

一. 配置:

1. 配置全局用户名:

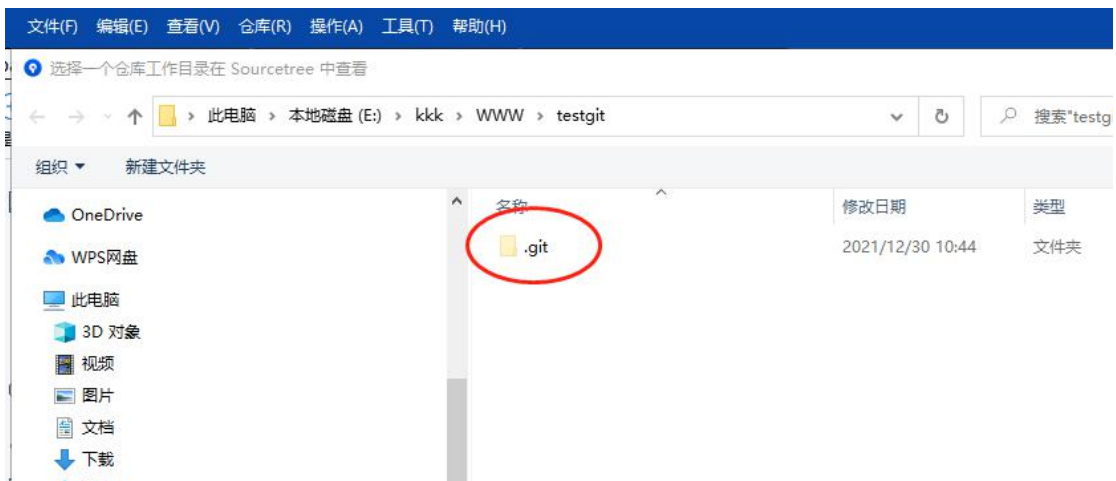
作用就是相当于在你的电脑上创建了一张名片，保存了登录 github 的邮箱和用户名，以后用 git 就不用再重新输入了

- A. `git config --global user.name 'kkk'` (kkk 为你的 github 邮箱所对应的用户名)
- B. `git config --global user.email wangyongzhang@keyirobot.com` (为你注册的 github 账号的邮箱)

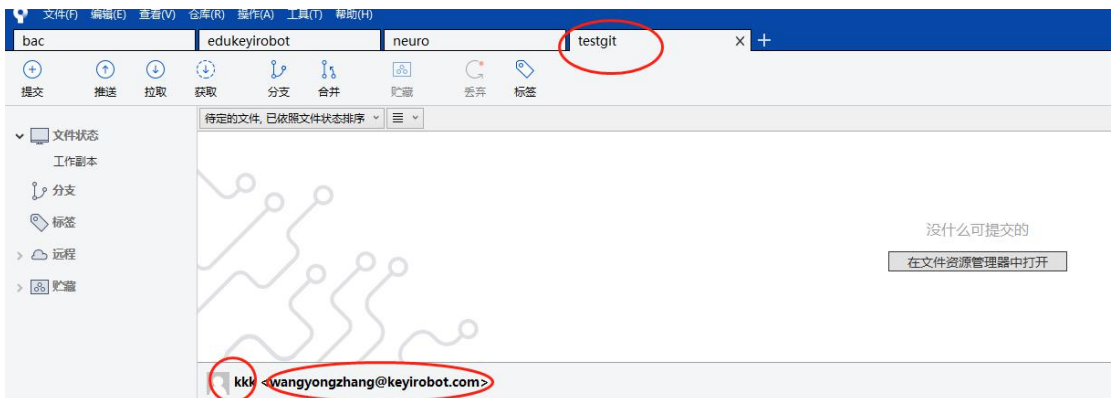
备注: github 可以用 name 和 email 当作用户名登录

2. 配置本地仓库

- A. `git init` // 初始化 git 仓库 (`git init` 后的生成 “.git” 版本文件控制目录)



在 sourceTree 可视化中可见



- B. 连接远程仓库 `git remote add origin + "url"` 配置远程 origin 路径

作用是:将本地仓库与远程仓库关联起来(上传和下载代码)

说明:

a).git 是最大的指令集, git remote 是子指令集, add origin + “url” 是 git remote 的一个具体指令

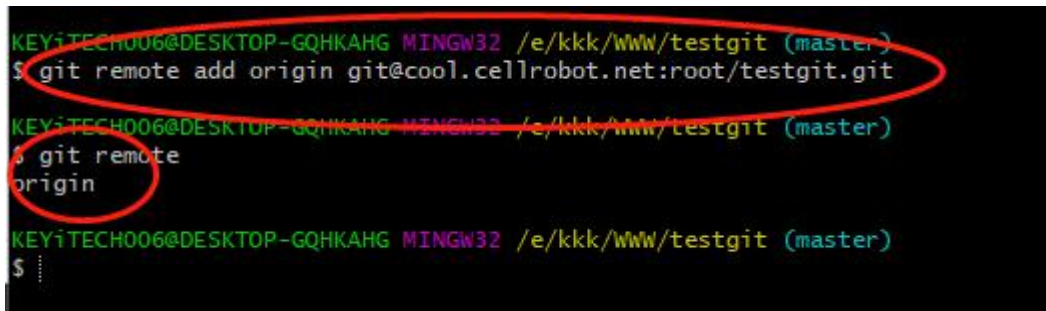
b).origin 是给远程路径起的缩写别名 (可以不叫作 origin 但一般习惯写做 origin)

c).url 是你想链接的远程仓库的地址

eg: git remote add origin [git@github.com:test/test.git](https://github.com/test/test.git)

备注: 查看你已经配置的远程仓库服务器, 可以运行 git remote / git remote show 命令。它会列出你指定的每一个远程服务器的简写。如果你已经克隆了自己的仓库, 那么至少应该能看到 origin — 这是 Git 给你克隆的仓库服务器的默认名字

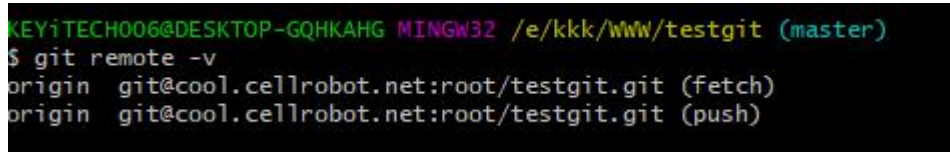
\$ git remote 查看远程分支的别名
origin

A terminal window with a black background and green text. The prompt is 'KEYITECH006@DESKTOP-GQHKAHG MINGW32 /e/xxx/www/testgit (master)'. The first command is '\$ git remote add origin git@cool.cellrobot.net:root/testgit.git'. The second command is '\$ git remote origin', which outputs 'origin'. A red circle is drawn around the 'git remote origin' command and its output.

git remote -v : 查看需要读写远程给仓库 git 保存的简写和对应的 URL

origin git@cool.cellrobot.net:cool/neuro.git (fetch)

origin git@cool.cellrobot.net:cool/neuro.git (push)

A terminal window with a black background and green text. The prompt is 'KEYITECH006@DESKTOP-GQHKAHG MINGW32 /e/xxx/www/testgit (master)'. The command is '\$ git remote -v'. The output is 'origin git@cool.cellrobot.net:root/testgit.git (fetch)' and 'origin git@cool.cellrobot.net:root/testgit.git (push)'.

下面命令看看就行了, 不常用:

远程仓库的重命名与移除

你可以运行 git remote rename 来修改一个远程仓库的简写名。例如, 想要将 pb 重命名为 paul, 可以用 git remote rename 这样做:

```
$ git remote rename pb paul
```

```
$ git remote remove paul
```

三. Git 常用命令

1. git init 新建仓库 (同时自动创建了第一个分支: master 主分支) git clone 同样可以创建新仓库

2. `git clone username@host:/path/repository` 克隆仓库

3. `git add filename1 filename2` (指定一个或多个文件 并把文件放到“暂存区”)

未执行任何动作的文件所在区叫做“工作区” 即是未暂存文件所在区(未执行 `git add`)

`git add .` (“.” 表示当前目录下: 不包括父级目录)

(`git add a.txt` 此命令把 `a.txt` 放到了 `stage/index` 或者叫“暂存区”)

注意: `git add .` 和 `git add *` 区别

`git add .` 会把本地所有 `untrack` 的文件都加入暂存区, 并且会根据 `.gitignore` 做过滤,

但是 `git add *` (一般不用、慎用) 会忽略 `.gitignore` 把任何文件都加入

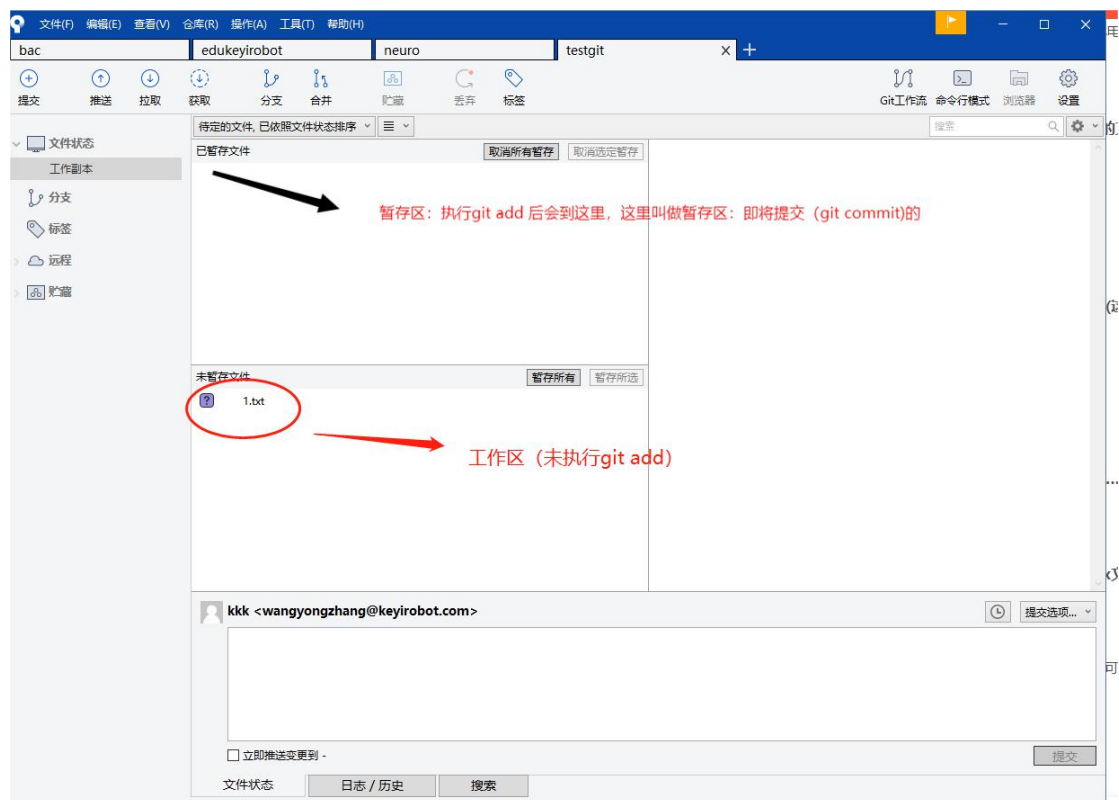
4. `git commit -m ‘提交代码备注信息’`

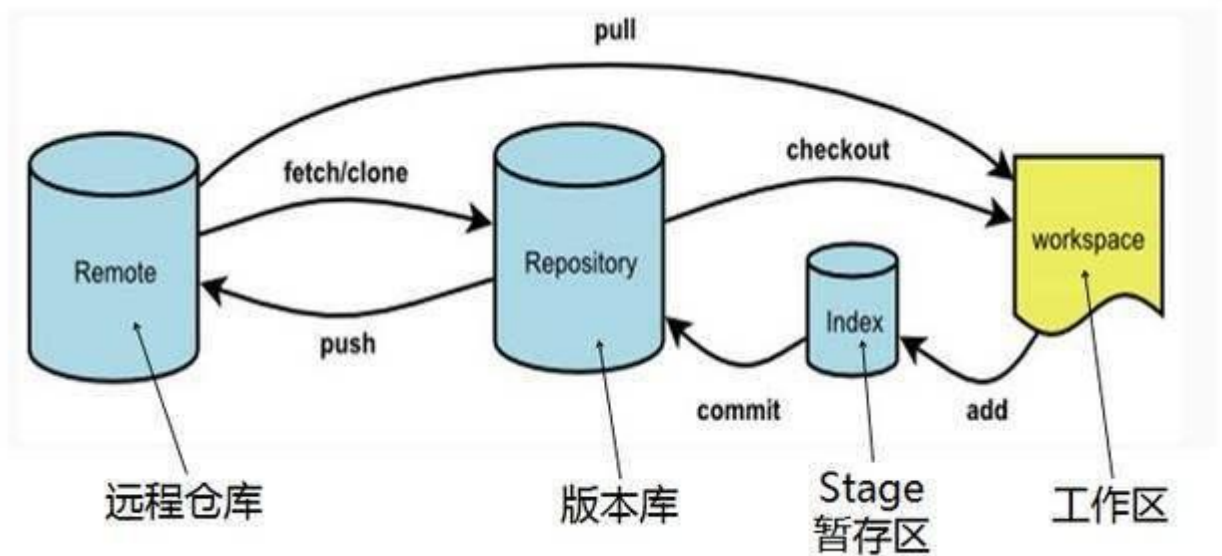
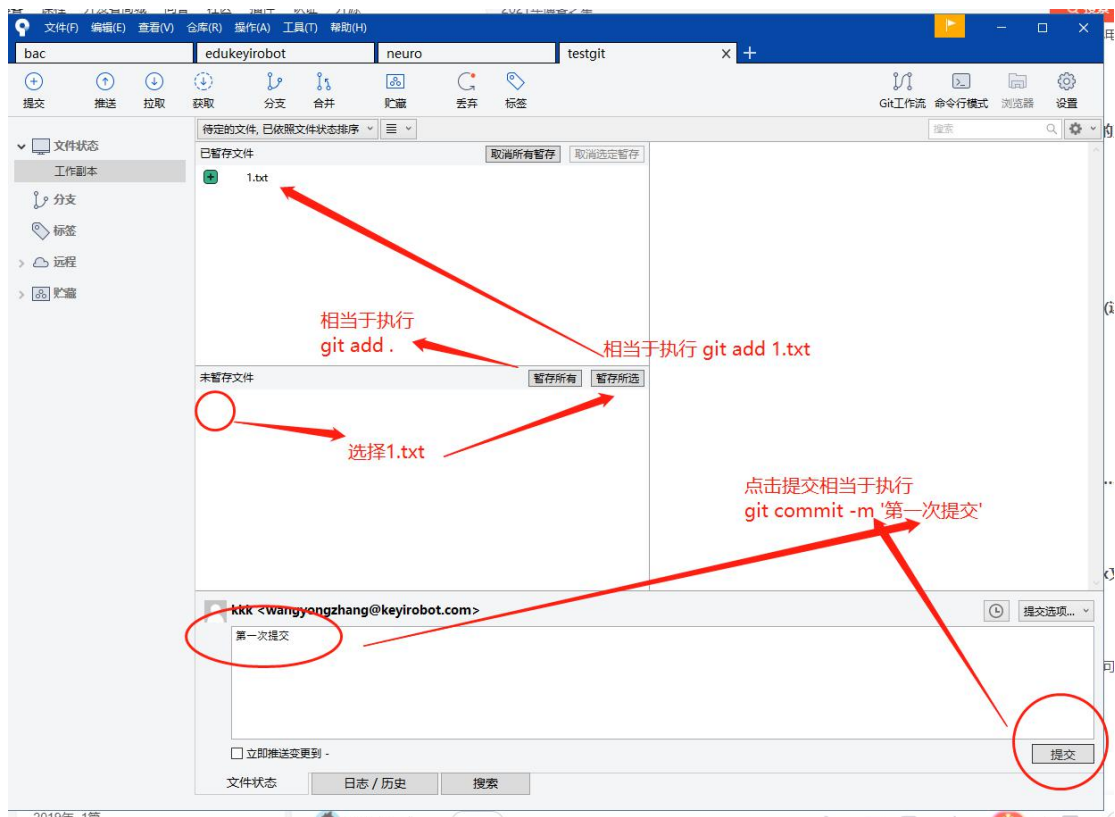
综上 Git 中添加, 是分两步执行的;

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

第二步是 `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。

可以理解为需要提交的文件统统放在暂存区, 然后, 一次性提交暂存区的所有修改。





5. git status (working tree clean) 查看本地版本状态

git status

on branch master

nothing to commit, working tree clean

6. `git push (-u) origin master/fix-user` 把代码 push 到远程

7. `git pull origin master/fix-user`

下面看看就行, 不常用操作

(`pull` 拉取远程仓库的内容更新到本地仓库, 使用 `--allow-unrelated-histories` (一般不用) 忽略本地仓库和远程仓库的无关性, 强行合并)

```
$ git push <远程主机名> <本地分支名>:<远程分支名>
```

如果本地分支名与远程分支名相同, 则可以省略冒号: `git push <远程主机名> <本地分支名>`

`git push origin master` 等价于 `git push origin master:master`

如果本地版本和远程有差异, 但是又要强制推送可以用: `--force` 参数

```
git push --force origin master
```

删除主机的分支可以使用 `--delete` 参数, 以下命令表示删除 `origin` 主机的 `master` 分支

```
git push origin --delete master
```

分支推送顺序的写法是<来源地>:<目的地>, 所以 `git pull` 是<远程分支>:<本地分支>, 而 `git push` 是<本地分支>:<远程分支>

如果省略远程分支名, 则表示将本地分支推送与之存在“追踪关系”的远程分支 (通常两者同名), 如果该远程分支不存在, 则会被新建

如果当前分支与多个主机存在追踪关系, 则可以使用 `-u` 选项指定一个默认主机, 这样后面就可以不加任何参数使用 `git push`

如果省略本地分支名, 则表示删除指定的远程分支, 因为这等同于推送一个空的本地分支到远程分支

```
git push origin :master 等同 git push origin --delete master
```

上面命令表示删除 `origin` 主机的 `master` 分支

`Git push -u origin master`

上面命令将本地的 `master` 分支推送到 `origin` 主分支, 同时指定 `origin` 为默认主分支, 后面就可以不加任何参数使用 `git push` 了。

不带任何参数的 `git push`, 默认只推送当前分支, 这叫做 `simple` 方式。此外, 还有一种 `matching` 方式, 会推送所有有对应的远程分支的本地分支。Git 2.0 版本之前, 默认采用 `matching` 方法, 现在改为默认采用 `simple` 方式。如果要修改这个设置, 可以采用 `git config` 命令

```
git config --global push.default matching/simple
```

还有一种情况, 就是不管是否存在对应的远程分支, 将本地的所有分支都推送到远程主机, 这时需要使用 `-all` 选项。

`git push --all origin`

上面命令表示，将所有本地分支都推送到 `origin` 主机。

如果远程主机的版本比本地版本更新，推送时 `Git` 会报错，要求先在本地做 `git pull` 合并差异，然后再推送到远程主机。这时，如果你一定要推送，可以使用 `- force` 选项。

`git push --force origin`

上面命令使用 `- force` 选项，结果导致在远程主机产生一个“非直进式”的合并 (`non-fast-forward merge`)。除非你很确定要这样做，否则应该尽量避免使用 `- force` 选项。

最后，`git push` 不会推送标签(`tag`)，除非使用 `- tags` 选项。

`git push origin --tags`

8. `git checkout -b fix-bug` 新建分支并切换分支 (Switched to a new branch 'fix-bug')

9. `git checkout master` 切换回主分支

10. `git branch -d fix-bug` 删除本地分支

11. `git push origin --delete fix-bug` 删除远程 (服务器上的分支)

12. `git branch -r` 查看远程仓库多少分支

13. `git merge feature-user-login` 合并分支

14. `git checkout -- filename`, eg: `git checkout a.txt` 取消 `a.txt` 的修改 (工作区的修改)

15. 用于 `git-pull` 中，来整合另一代码仓库中的变化 (即: `git pull = git fetch + git merge`)

四. 撤销的时机:

1. 撤销 `add` 之前的 (删除或者撤销 “工作区” 中的内容)

说明: 如果是新建的文件或者文件夹处于 `Untracked files` 状态, 需要手动 `git add` 才能纳入 `git` 仓, 否则 `git` 还受控制了

如果文件或者文件夹已经在 `git` 仓库管理之下的操作:

A. `git checkout -- 1.txt` 撤销对 `1.txt` 的修改, (未 `git add 1.txt` 的修改)

B. `git reset --hard HEAD` 撤销之前未 `add` 的所有修改 (同样不会删除新建的文件或者文件夹)

二者的区别是: `git checkout`. 只会影响当前目录或者其子目录

git reset --hard HEAD 会影响整个项目的

如果是新建的文件或者文件夹(处于: **untracked files** 状态的)想撤销:

A: `git clean -fd` 撤销不在 git 控制中的文件或者文件夹, 注意此操作不影响在 git 记录中的文件或者文件夹

案例: 新建了 2.txt 文件同时修改 1.txt, 如果想撤销执行命令: `git clean -fd` 即可删除 2.txt

```
$ git clean -fd
```

Removing 2.txt

注意:

此时如果 `git status` 发现 1.txt 还是被修改后的状态值, 没有受到影响

如果还需要撤销 1.txt 的修改, 则需要执行 `git checkout 1.txt`

2. 撤销 add 的 (从 “暂存区” 回退到 “工作区”)

A. `git reset head a.txt` 取消某个文件的 `git add a.txt`

B. `git reset head` 取消 `git add .` 的操作

二者的区别是: 影响的范围不同

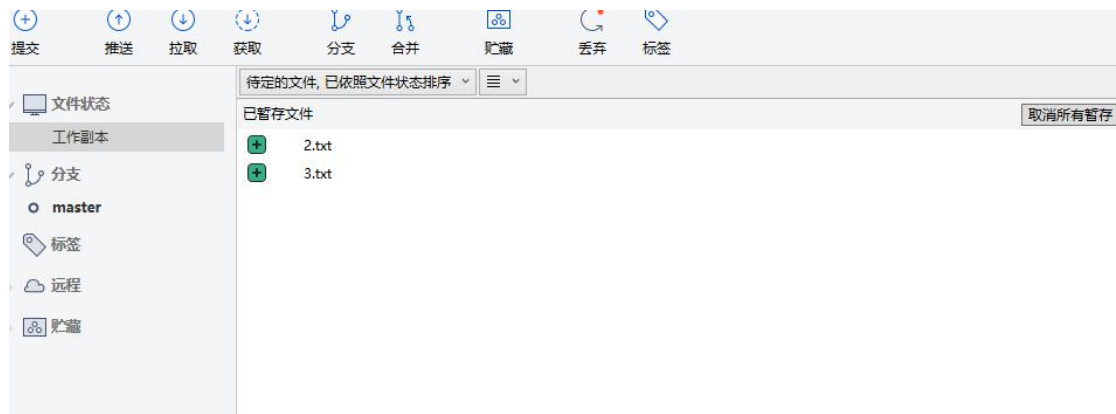
`git reset head xxx` 对单个文件处理, `git reset head` 是回退整个 `git add` 的所有操作

3. 撤销 git commit 的几种操作方式:

A. `git reset --soft HEAD^` (直接删除了最后一次撤销, 不会留下撤销记录, 但是会把上次的提交撤回暂存区 (INDEX 区域))

其实就是撤销最后一次 `git commit -m ‘最后一次提交’` 的操作

工作区 (work tree) `git add` 后去暂存区 (INDEX 区域)



第二次提交 2.txt, 第三次提交 3.txt 回退两次后 2.txt 和 3.txt 的修改都退回到了暂存区

HEAD^ 表示上一个版本, 即上一次的 commit, 也可以写成 HEAD~1

如果要回退到两次前的 commit, 想要都撤回, 可以使用 HEAD~2

如果要回退到三次的 commit, 想要都撤回, 可以使用 HEAD~3

会把这两次的提交回退叫暂存区 (INDEX 区域)

注意: 无论回退多少次, 回退后的状态的代码都是不变的, 因为代码只是到了暂存区, 并没有改变最新代

码的任何变化，这种只适合需要再次修改的情况，不适合回退线上运行的代码的紧急撤销。

B. `git reset --hard head^ / head~1` 等价

删除了最后一次的提交，不会留下撤销记录(硬删除，直接删除 `git commit` 记录及其代码，而且不会把代码回退到暂存区：野蛮删除慎用)，返回到了上一次的提交

而且此时本地的代码也是上一次提交的代码（这中回退会改变工作区的代码）

C. `git revert HEAD` 撤销上一次提交，并生成一次新的提交（会保留上一次提交的记录版本，代码也被真实回退到之前的上一个版本了）

* <code>git revert HEAD</code>	撤销前一次 <code>commit</code>
* <code>git revert HEAD^</code>	撤销前前一次 <code>commit</code>

注意：如果要回退的版本非久远或者不清楚第几次，可以用 `commit_id`

`git reset --hard 46b66217d92af8c64bcd1d796fe67695022c9d54`

最重要的一点：`revert` 是回滚某个 `commit`，不是回滚“到”某个

``git revert commit_id``之后并不会回滚到该 `id` 的内容，而是将该 `id` 的内容给逆向操作一遍，比如说，`a` 操作添加了“haha”，`commit` 了 `a`，`b` 操作添加了“xixi”，`commit b`。现在想回滚到只添加了“haha”，需要的是删除“xixi”，也就是逆向操作 `b`，所以应该 ``git revert b`` 的 `commit_id``。``git revert`` 应该翻译成“反转、逆转”比较好理解，而不是回退

`git revert ae7102ea99207e40733e0`（指定的提交 `COMID`）

这种有回退记录，但是回退的是指定版本的前一个版本（并且会保留最后一次提交的版本）

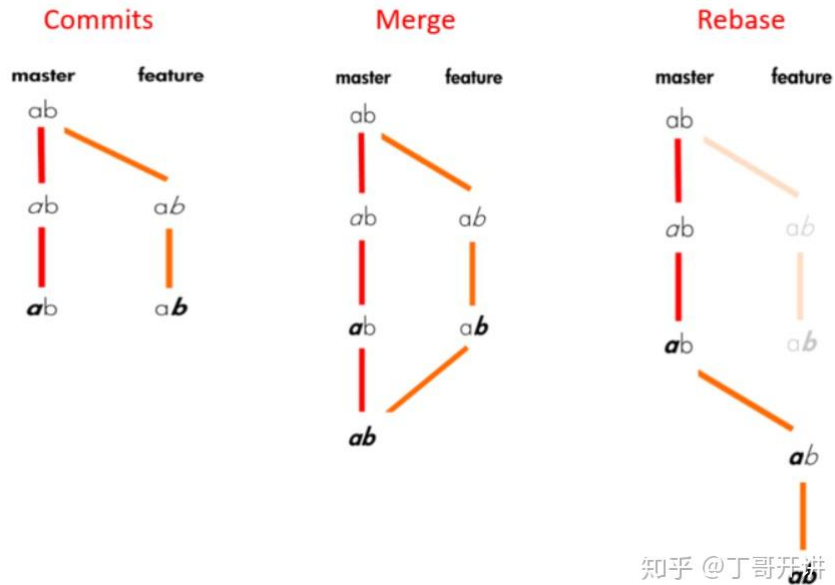
`git reset` 是把 `HEAD` 向后移动了一下，而 `git revert` 是 `HEAD` 继续前进，只是新的 `commit` 的内容和要 `revert` 的内容正好相反，能够抵消要被 `revert` 的内容

注意：这里的 `git revert head^` 会把最近一次提交的上一次提交给清除

比如现在 1,2,3,4 然后 `revert head^`后变成了 1,2,4 把 3 给清除了

五. 合并代码

开发分支（`dev`）上的代码达到上线的标准后，要合并到 `master` 分支



1. `git merge dev`
2. `git rebase`

3. 二者区别:

A. `git merge` 会生成一个新的合并点, 而 `git rebase` 不会, 比如下面: 当前存在两个分支, `master` 和 `test` 分支

```

D---E test (待合并的分支)
/
A---B---C---F master (主分支)

```

如果使用 `merge` 合并, 将为分支合并自动识别出最佳的同源合并点: 并新增合并点 G

```

D-----E
/   \
A---B---C---F----G test, master

```

如果使用 `rebase` 合并, 则合并结果为:

```

A---B---D---E---C'---F' test, master

```

即 `git rebase` 可以线性的看到每次提交, 而 `git merge` 可以更加精确的看到每次提交。

所以想要更好的提交树, 使用 `rebase` 操作会更好一点。这样可以线性的看到每一次提交, 并且没有增加提交节点。

4. 合并遇到冲突时的如何处理

`merge` 操作遇到冲突的时候, 当前 `merge` 不能继续进行下去。手动修改冲突内容后, `add` 修改, `commit` 就可以继续往下操作, 而 `rebase` 操作的话, 会中断 `rebase`, 同时会提示去解决冲突。解决冲突后, 将修改 `add` 后执行 `git rebase --continue` 继续操作, 或者 `git rebase --skip` 忽略冲突

辨析: git merge 冲突的时机

如果分支的修改时间落后于 master, 那么会分支代码更新会直接覆盖掉 master 上的代码, 并且不会冲突。

如果分支的修改时间更新于 master, 那么 merge 合并的时候会提示冲突 (而且此时已经发生了冲突需要解决冲突, <<<<<<< HEAD 自己的代码

五. GITLAB 使用的问题:

1. 新建 git 项目并 vF 为全球 2 皮 0/+ 【

分组 (用不到一般)

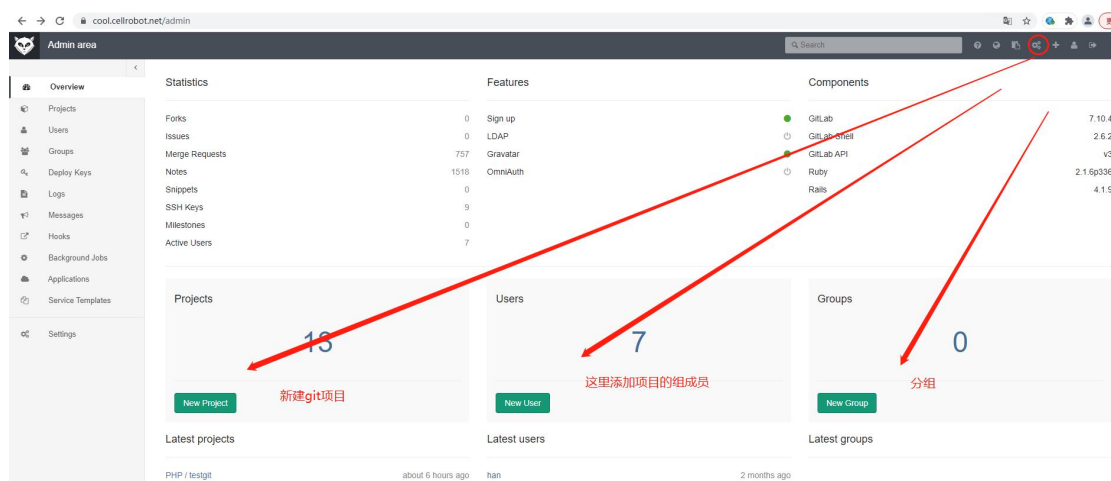
0 一个组是几个项目的集合

默认情况下, 组是私有的

群组成员只能查看他们有权访问的项目

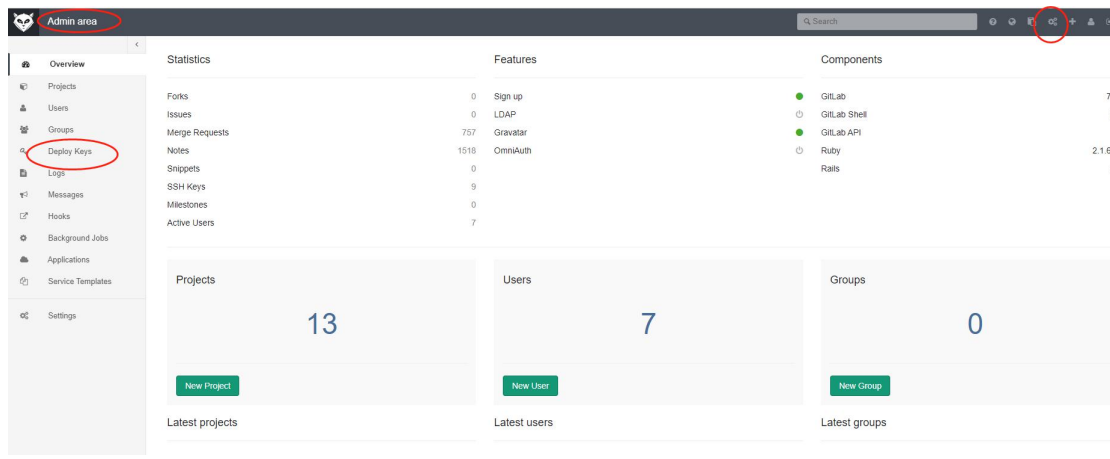
组项目 URL 以组命名空间为前缀

现有项目可移入组



2. 给项目分配谁能拉取或者克隆的权限 (因为默认项目必须都是私有的)

免密码拉取推送代码到私有仓库



SSH

SSH 密钥

SSH 密钥允许您在计算机和 GitLab 之间建立安全连接。

在生成 SSH 密钥之前，通过运行 `cat ~/.ssh/id_rsa.pub`。如果您看到以 `ssh-rsa` 或开头的长字符串 `ssh-dsa`，则可以跳过 `ssh-keygen` 步骤。

要生成新的 SSH 密钥，只需打开您的终端并使用下面的代码。ssh-keygen 命令会提示您输入用于存储密钥对的位置和文件名以及密码。当提示输入位置和文件名时，您可以按 Enter 以使用默认值。

最佳做法是为 SSH 密钥使用密码，但这不是必需的。您可以按 Enter 跳过创建密码。请注意，您在此处选择的密码无法更改或检索。

```
ssh-keygen -t rsa -C wangyongzhang@keyirobot.com
```

使用下面的代码显示您的公钥。

```
cat ~/.ssh/id_rsa.pub
```

将密钥复制粘贴到用户配置文件中“SSH”选项卡下的“我的 SSH 密钥”部分。请复制 ssh- 以您的用户名和主机开头和结尾的完整密钥。

使用下面的代码将您的公钥复制到剪贴板。根据您的操作系统，您需要使用不同的命令：

视窗：

```
clip < ~/.ssh/id_rsa.pub
```

苹果电脑：

```
pbcopy < ~/.ssh/id_rsa.pub
```

GNU/Linux (需要 xclip)：

```
xclip -sel clip < ~/.ssh/id_rsa.pub
```

部署密钥

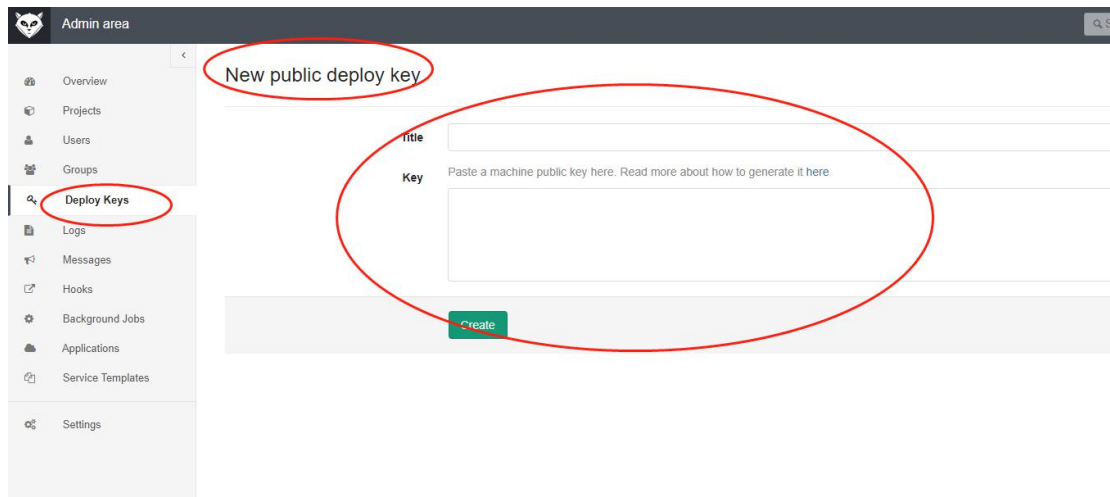
部署密钥允许使用单个 SSH 密钥对多个项目进行只读访问。

这对于将存储库克隆到持续集成 (CI) 服务器非常有用。通过使用部署密钥，您不必设置虚拟用户帐户。

如果您是项目主人或所有者，则可以在“部署密钥”部分下的项目设置中添加部署密钥。按“新建部署密钥”按钮并上传公共 SSH 密钥。此后，使用相应私钥的机器对项目具有只读访问权限。

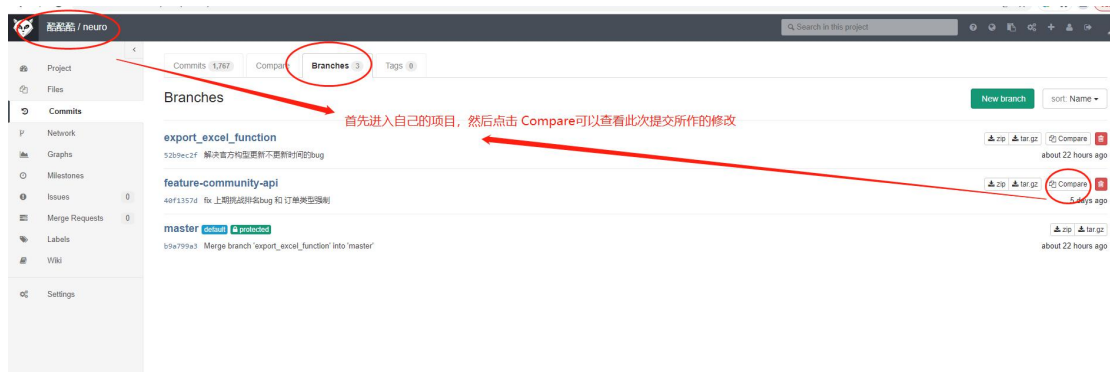
您不能使用“新建部署密钥”选项两次添加相同的部署密钥。如果您想将相同的密钥添加到另一个项目，请在显示“从您可用的项目中部署密钥”的列表中启用它。您有权访问的所有项目的部署密钥都可用。这种项目访问可以通过成为项目的直接成员或通过一个小组来实现。见 `def accessible_deploy_keys` 在 `app/models/user.rb` 获取更多信息。

把此 ssh 方式生成密钥后放到 New public deploy key

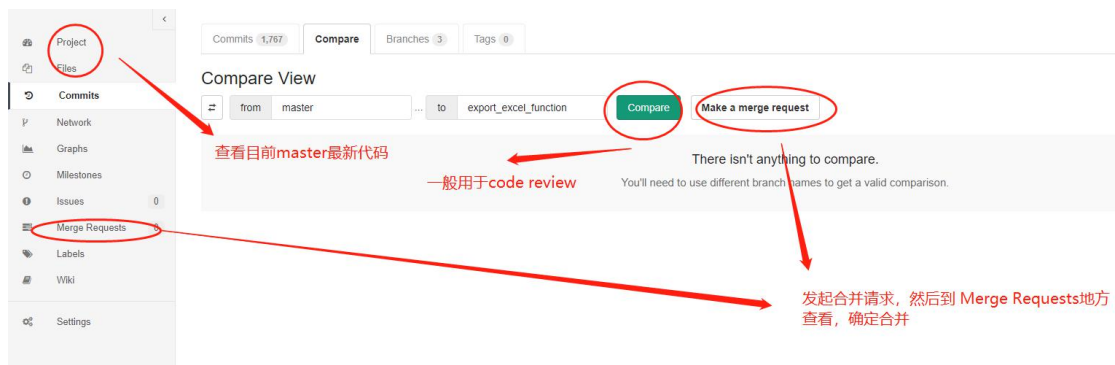


即可获取拉取此私有项目的权限

3. 查看提交记录然后合并到主分支



4. 查看分支提交记录后两个选择（compare 对比查看修改内容，OK 后点击 “make a merge request”）向 master 提交合并申请



七. GIT 使用常见问题：

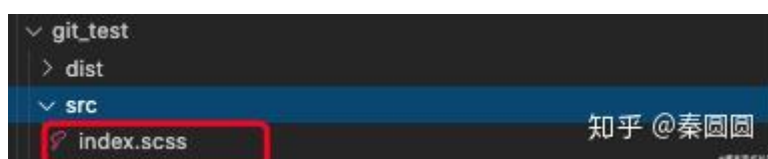
1. 修改了 .gitignore 未生效如何处理：(.gitignore 用于忽略不想加入版本控制的文件或者文件夹)

常用于：配置文件，线上线下不一样的配置以及没必要管理的资源文件或者拓展包

(<https://www.cnblogs.com/liangzhixiaolaohu/p/15258309.html>)

写好 .gitignore 后发现要忽视文件照样上传了，很可能的原因是 你已经 `trace` 了，比如之前使用过 `git add`。这样的命令 (.gitignore 只对未跟踪的文件起作用！已跟踪的文件是指那些被纳入了版本控制的文件，在上一次提交中有它们的记录。那么未跟踪文件就是指那些从没提交过的文件)

Demo：



解决方案

我与 git 的对话

【我】：我的目的是在仓库里面保留 index.scss，但是要所有人忽略它的改动。你能做到吗？

【git】：我做不到，我是个版本控制的仓库，我要为所有文件的版本负责，而不是你存储文件的中转站！但是你可以这样：

别让我管理你的 index.scss，踢出我的版本控制，存在自个儿的电脑里。

暂时只忽略你的文件改动，别人已加到版本库的文件改动，你可管不着。

让我们来看看根据 git 提供的思路，如何实践

所以取消 trace，在重新 add 即可

修改 .gitignore 文件 立即生效

```
git rm -r --cached . #清除缓存
```

```
git add . #重新 trace file
```

```
git commit -m "update .gitignore" #提交和注释
```

```
git push origin master
```

2. 项目开发过程中来了新需求如何处理

A. 先拉取线上最新代码，

B. 然后从 master 新建分支进行新的项目开发

3. 合并冲突解决方案：

a. 找到冲突文件选择要保留的代码后提交新版本

```
<<<<<<< HEAD
```

本地代码 （自己写的代码）

```
=====
```

拉下来的代码（其他分支的或者线上的代码）

```
>>>>>>>
```

然后选择删除，调试合并再次提交 commit 记录

4. 已上线代码发现重大 bug 需要回滚的操作

用：git revert HEAD 靠谱

5. 实操中的建议工作流：

- A. 新项目开始的时候必须保持自己的代码是最新代码: `git pull origin master`
- B. 在最新代码上新建开发分支 `git checkout -b fix/feature-code`
- C. 为了防止在服务器端合并冲突, 可以提交上线前先拉取服务器最新代码到本地, 在本地合并后解除冲突后在 push , `git pull origin fix-code`
- D. 如果没有冲突 `git push origin fix-code`(如果服务器没有会新建)
- E. 如果有冲突解决冲突提交新的版本记录后 `git push origin fix-code`

参考链接:

<https://www.cnblogs.com/arxive/p/14802613.html>

<https://www.cnblogs.com/limuma/p/9318712.html>

man git clean 查看命令