### **git操作手册**

**前言：**

1.蓝色字体表示命令行命令，正式执行时不要复制前面的#号，#号只是提示应该使用root权限操作

2.绿色字体表示注释，有时注释太多就不用绿色表示了

3.注意：本文档的所有操作请先在测环境进行实践，请不要直接在真实的服务器中操作！

**版权声明**：

本文档以开源的形式发布，所有条款如下：

1. 无担保：作者不保证文档内容的准确无误，亦不承担由于使用此文档所导致的任何后果

2. 自由使用：任何人可以出于任何目的而自由地 阅读/链接/打印/转载/引用/再创作 此文档，无需任何附加条件

若您 阅读/链接/打印/转载/引用/再创作 本文档，则说明接受以上2个条款。

作者：李茂福

更新日期：2024-01-11

**★git命令操作**

Git是一个开源的分布式版本控制系统，可以高效地处理从很小到非常大的项目版本管理。

**★git命令行客户端安装及设置**

# yum install git -y #自带光盘里有

# git --version #查看git软件版本

git version 2.39.1

**★全局设置**

设置用户名和邮箱，只用于提交commit时做metadata信息，不用于身份验证

#全局设置信息保存在 ~/.gitconfig 文件里

# git config --global user.name cof #设置用户名

# git config --global user.email cof@cof-lee.com #设置邮箱

# git config --global http.sslVerify false #不验证ssl证书

# git config --global --list #查看全局设置

**★本地设置**

针对单一项目的设置，使用 --local参数，本地设置信息保存在 项目目录/.git/config文件里

# cd 项目目录

# git init #初始化仓库，将在当前路径下生成.git子目录，子目录里有此git仓库的相关信息

# git config --local user.name cof #设置用户名

# git config --local user.email cof@cof-lee.com #设置邮箱

# git config --local http.sslVerify false #不验证ssl证书

# git config --local --list #查看本地设置

**★系统设置**

系统设置信息保存在/etc/gitconfig文件里

# git config --system user.name cof

# git config --system user.email cof@cof-lee.com

# git config --system --list #查看系统设置

# git config --list #查看所有设置（全局+本地+系统）

# git config user.name #查看用户名

**★以上3个范围的设置优先级：** 本地设置 > 全局设置 > 系统设置

**★push.default配置项**

# git config --local push.default simple #设置推送模式

|  |  |
| --- | --- |
| nothing | 直接push会出错，需要显式的指出推送的远程分支，例如 git push origin master |
| current | 推送时只会推送当前所在的分支到远程同名分支，如果远程分支不存在相应的同名分支，则创建该分支 |
| upstream | 推送当前分支到它的upstream分支上，这个模式只适用于推送到与拉取数据相同的仓库(比如central workflow) |
| simple | 在中央仓库工作流程模式下，只能推送到与本地分支名一致的upstream分支中，如果推送的远程仓库和拉取数据的远程仓库不一致，那么该模式会像current模式一样进行操作。因为该选项对于新手来说是最安全的，所以在git 2.0中，simple是push.default的默认值配置项（2.0以前的默认配置项是matching） |
| matching | 推送本地和远程都存在的同名分支 |

**★仓库操作**

# mkdir ~/testgit #创建项目目录

# cd ~/testgit

# git init #初始化仓库，将在当前路径下生成.git子目录，子目录里有此git仓库的相关信息，初始化后要设置当前工作的用户名等相关信息，否则使用全局设置

# git status #查看当前工作目录和暂存区的状态

On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

# echo 'print("hello world")' > main.py #随便创建一个文件

# git add . #提交当前工作目录的所有文件到暂存区（add为暂存操作），命令最后有一个点.

# git status #再次查看状态

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: main.py #暂存区有了一个新的文件：main.py

# git commit -m "msgxxx" #将暂存区提交到仓库区（commit为提交操作），-m后指定本次提交的说明信息

1 file changed, 1 insertion(+)

create mode 100644 main.py

# git log #查看提交记录；显示完整的hash值

commit 7b8ff6902bf14780a413a021cc1d946fd82fa82e (HEAD -> master) # commit后面为hash值，HEAD表示

Author: Cof <cof@cof-lee.com> # 当前分支指向这个提交（commit）

Date: Mon Oct 30 10:47:04 2023 +0800

msgxxx #提交的说明信息

# git log --oneline #同上，提交记录以一行显示一次提交；只显示hash值前6个字符

7b8ff69 (HEAD -> master) msgxxx

# git log 文件名 #查看目标文件的提交记录

# git log -p -3 #查看最近3次提交记录

# git reflog #查看git操作记录

# git reset HEAD^ #恢复到上次提交的版本（回退一次）

# git reset HEAD^^ #恢复到上上次提交的版本（回退2次）

# git diff #查看未暂存的修改，在工作目录增删改文件后，未add的

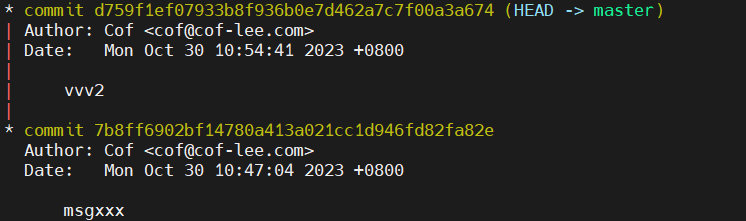
# git diff --cached #查看未提交的暂存

**★分支操作**

分支（branch）的本质其实就是一个提交对象（commit），HEAD是一个指针，它默认指向master分支，切换分支时就是让HEAD指向不同的分支。每次有新的提交时，HEAD都会自动指向最新的提交。

首次创建分支前，得先进行一次提交，否则git无法创建分支（因为分支是基于提交去创建的）。

# git log --decorate --graph --all #查看整个项目的分支图



# git branch #查看分支列表，默认只有一个分支，分支名默认为master

\* master

# git branch xxx #在当前提交的对象上创建新的分支，名为xxx

# git branch yyy 7b8ff6 #在指定的提交对象上创建新的分支；提交的hash值可写全，也可只写前6字符

# git checkout xxx #切换分支

# git branch

master

\* xxx #前面有\*星号，表示为当前所处的分支

yyy

# git branch -v #查看所有分支的最新的提交

# git branch -D xxxxx #强制删除分支（不能删除当前所处的分支）

# git branch -d xxxxx #删除空的分支，删除已被合并的分支

# git merge yyy #将yyy分支合并到当前分支（当前为xxx），源为yyy，目的为xxx

# git branch --no-merged #查看没有合并到当前分支的分支列表

# git config --global init.defaultBranch main #设置默认分支名为main，默认为master；设置完成后，再init其他仓库，则默认名称生效（只对新创建仓库生效）

# git branch -m main #将当前分支更名为main

**★tag操作**

tag是git版本库的一个标记命令，指向了某个commit的指针，tag命令主要用于发布版本管理，当一个版本发布时，打上tag标签。

有了分支（branch）为什么还要引入tag这个概念呢？

因为tag是相对于同一个分支而言的，同一分支不停地开发，有新增的文件，也有修改旧的文件，每次提交（commit）后当前分支的HEAD指针会指向最新的提交，那么如何管理旧的提交呢？可以对某个时间点的提交打个标记，方便后续的版本追溯及管理。这个标记就是tag

# git tag v1.0.1 #将当前所处分支的最新提交打上tag，名为v1.0.1

# git tag v1.2 -m 'v2' #创建名为v1.2的tag，-m指定说明信息，未指定-m说明信息时，则默认使用提交（commit）时的说明信息作为此tag的说明信息

# git tag -n #查看所有tag及对应的提交

v1.0.1 msgxxx

v1.2 v2

# git show v1.0.1 #查看指定的tag信息

# git push rmtRepo v1.2 #将指定的tag推送到远程仓库，**先参考下一节进行远程仓库的映射**

# git push rmtRepo --tags #推送所有tag到远程仓库

# git tag -d v1.0.1 #删除指定的tag

# git push rmtRepo --delete v1.2 #删除远程tag

# git tag new-tag old-tag #基于原有的old-tag创建新tag（可用于重命名，生成新tag后，再删除旧的tag）

**★git远程协作开发**

**远程协作的基本流程：**

① 管理员在创建一个空的远程仓库（比如在gitlab上创建一个空的项目）

② 管理员创建一个待推送的本地仓库

③ 管理员关联本地仓库与远程仓库

④ 管理员在本地仓库中编辑代码，提交代码

⑤ 管理员推送本地仓库代码到远程仓库

⑥ 管理员邀请协作成员

⑦ 成员克隆远程仓库

⑧ 成员做出修改

⑨ 成员推送自己的修改

⑩ 管理员拉取成员的修改

**★管理员操作①到⑤**

# git init #初始化一个git仓库

#进行代码的编辑，过程省略

# git add . #将“增删改后的代码”从当前工作区存放到暂存区

# git commit -m "first commit" #将暂存区中存放的文件提交到仓库中（本地的仓库）

# git remote add rmtRepo https://gitlab.cof-lee.com/username/*RemoteRepoName*.git #将本地代码库的当前分支与远程的代码库相关联，远程库在本地的别名设置为rmtRepo（当前本地分支与远程仓库的分支名称要保持一致）git用户名及仓库名称要根据实际情况修改，比如：

# git remote add rmtRepo https://gitlab.cof-lee.com/cof/pro-1.git

# git push -uf rmtRepo main #将本地代码库的当前分支（main）推送到远程的代码库的main分支

# git pull rmtRepo main #从远程代码库下载main分支的最新修改，若未指定分支名，则默认拉取同本地分支名的分支

# git remote -v #查看当前仓库关联的远程代码库的URL

# git remote rm rmtRepo #删除远程仓库

**★协作成员操作⑦到⑨，详见下一小节：克隆远程仓库**

**★克隆远程仓库**

# git clone https://gitlab.cof-lee.com/cof/pro-1.git #克隆远程仓库到本地，默认克隆的是远程的默认分支，本例中远程仓库默认分支为main（克隆后会自动在当前目录下生成项目目录）

# cd pro-1/ #进入克隆后的项目目录

# git remote -v #直接克隆的话，默认将远程仓库映射到本地，别名为origin

origin https://gitlab.cof-lee.com/cof/pro-1.git (fetch)

origin https://gitlab.cof-lee.com/cof/pro-1.git (push)

# git branch -a #查看所有分支

\* main #本地只有main分支

remotes/origin/HEAD -> origin/main

remotes/origin/bran3

remotes/origin/main

# git branch bran4 #在当前分支main的基础上，创建新的分支bran4

# git checkout bran4 #切换分支

# git branch

\* bran4

main

#成员进行代码的编辑，并提交，过程省略

# git diff bran4 remotes/origin/main #查看本地的bran4分支与远程的main分支有啥区别

# git push -u origin bran4 #将本地代码库的当前分支（bran4）推送到远程的代码库的bran4分支，然后在远程仓库上就有了这个bran4的分支

# git branch -a #查看所有分支（包含本地及远程的）

\* bran4

main

remotes/origin/HEAD -> origin/main

remotes/origin/bran3

remotes/origin/bran4

remotes/origin/main

**#管理员再决定是否合并分支**

管理员电脑上：main分支# git merge remotes/rmtRepo/bran4 #将远程的bran4合并到本地的main分支

管理员电脑上：main分支# git push -uf rmtRepo main #将本地main推送到远程仓库

# git branch -d -r origin/bran4 #删除远程分支的映射，远程仓库上还有bran4，本地没有remotes/origin/bran4了

# git push origin --delete bran4 #删除本地上关于远程分支的映射以及在远程上也真的删除远程分支

Username for 'https://gitlab.cof-lee.com': cof

Password for 'https://cof@gitlab.cof-lee.com':

To https://gitlab.cof-lee.com/cof/pro-1.git

- [deleted] bran4

**★或者成员先创建一个空项目目录，进入目录后再映射仓库到本地**

# mkdir my-pro-1

# cd my-pro-1

# git init

# git remote add rmtRepo https://gitlab.cof-lee.com/cof/pro-1.git

# git fetch rmtRepo 远程分支:本地分支 #在本地新建分支，不会自动切换到该分支，需要手动checkout切换

**比如：**

# git fetch rmtRepo main:bran5 #将远程的main分支下载到本地并基于此创建bran5分支

# git checkout bran5 #切换到bran5分支

# git branch -a

\* bran5

remotes/rmtRepo/main

**或者：**

# git checkout -b bran5 rmtRepo/main #基于远程分支在本地新建分支，并**自动切换**到此分支

#成员再做增删改工作，过程省略

# git push -u rmtRepo bran5 #将本地代码库的当前分支（bran3）推送到远程的代码库的bran5分支

# git branch -a

\* bran5

remotes/rmtRepo/bran5 #可见远程仓库已有bran5分支了

remotes/rmtRepo/main

#如果管理员合并新分支内容到main了，则需要成员同步一下main分支

**成员电脑上操作：**

# git fetch rmtRepo main #同步远程代码库（同步远程的main分支到本地的当前分支）

# git merge remotes/origin/main main #合并同步远程代码库的main分支到本地的main分支

**★其他克隆操作**

# git clone -b bran5 --depth=1 http://gitlab.cof-lee.com/cof/pro-1.git #-b克隆指定版本，可以是branch，也可是tag，--depth=1表示史克隆最新一层提交，不克隆历史版本

**★git可视化客户端安装及设置**

https://git-scm.com/downloads/guis

**①git**

官网下载： https://git-scm.com/downloads



https://github.com/git-for-windows/git/releases/download/v2.42.0.windows.2/Git-2.42.0.2-64-bit.exe

**②TortoiseGit**

官网： https://tortoisegit.org/

使用TortoiseGit时，要求提前安装好git工具



**★部署gitlab仓库**

gitlab是一个基于git实现的在线代码仓库软件，提供web可视化管理界面，通常用于企业团队内部协作开发

**★安装gitlab-ce**

部署gitlab-ce要求服务器内存>=4GB，本例使用操作系统为 Rocky Linux 8.8，gitlab-ce版本16.5.0

**★首先gitlab服务器要安装ntp时间服务**

# yum install chrony -y

# systemctl enable chronyd

# systemctl start chronyd

# vi /etc/chrony.conf

server 10.99.1.1 iburst prefer #配置集群可访问的ntp服务器

# timedatectl set-timezone Asia/Shanghai #设置时区

# systemctl restart chronyd

# chronyc sources -v

**★配置yum源**

# curl -ks https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.rpm.sh -o script.rpm.sh

# bash script.rpm.sh #安装官方yum源

# cat /etc/yum.repos.d/gitlab\_gitlab-ce.repo #repo仓库内容如下：

[gitlab\_gitlab-ce]

name=gitlab\_gitlab-ce

baseurl=https://packages.gitlab.com/gitlab/gitlab-ce/el/8/$basearch

repo\_gpgcheck=0

gpgcheck=1

enabled=1

gpgkey=https://packages.gitlab.com/gitlab/gitlab-ce/gpgkey

https://packages.gitlab.com/gitlab/gitlab-ce/gpgkey/gitlab-gitlab-ce-3D645A26AB9FBD22.pub.gpg

sslverify=1

sslcacert=/etc/pki/tls/certs/ca-bundle.crt

metadata\_expire=300

[gitlab\_gitlab-ce-source]

name=gitlab\_gitlab-ce-source

baseurl=https://packages.gitlab.com/gitlab/gitlab-ce/el/8/SRPMS

repo\_gpgcheck=0

gpgcheck=1

enabled=1

gpgkey=https://packages.gitlab.com/gitlab/gitlab-ce/gpgkey

https://packages.gitlab.com/gitlab/gitlab-ce/gpgkey/gitlab-gitlab-ce-3D645A26AB9FBD22.pub.gpg

sslverify=1

sslcacert=/etc/pki/tls/certs/ca-bundle.crt

metadata\_expire=300

# yum install gitlab-ce -y #如果报gpg验证错误，可带上参数 --nogpgcheck

# cat /etc/gitlab/gitlab.rb | grep -v '^#' | grep -v '^$' #查看默认配置

external\_url 'http://gitlab.example.com'

**#修改配置文件**

# cat > /etc/gitlab/gitlab.rb <<EOF

external\_url 'https://gitlab.cof-lee.com'

#设置数据存放路径，默认 /var/opt/gitlab/git-data/repositories

git\_data\_dirs({

"default" => {

"path" => "/git-data"

}

})

#设置备份存放路径及备份默认保存时间，默认 /var/opt/gitlab/backups/

gitlab\_rails['backup\_path'] = "/git-backup"

gitlab\_rails['backup\_keep\_time'] = 604800

#设置时区

gitlab\_rails['time\_zone'] = "Asia/Shanghai"

#设置监听的端口号（https的端口号）

nginx['listen\_port'] = 443

#开启强制http转https

nginx['redirect\_http\_to\_https'] = true

#以下为http监听的端口号

nginx['redirect\_http\_to\_https\_port'] = 80

nginx['ssl\_certificate'] = "/etc/gitlab/gitlab.cof-lee.com.crt"

nginx['ssl\_certificate\_key'] = "/etc/gitlab/gitlab.cof-lee.com.key"

EOF

# mkdir /git-data

# mkdir /git-backup

#将证书及密钥文件复制到/etc/gitlab目录下

# gitlab-ctl reconfigure #重加载配置文件（耗时较长，约5到10分钟）

# gitlab-ctl restart #重新加载配置文件后需要重启gitlab

# cat /etc/gitlab/initial\_root\_password #查看初始化后自动生成的root密码，会在24小时后自动删除此文件

Password: Vx5HgVtCjOr09ZB6u0M+rHAb476Wp5BP5YmWqTyU5ng=

# gitlab-ctl status #查看gitlab状态

# gitlab-ctl start #启动

# gitlab-ctl stop #停止

# gitlab-ctl restart #重启

**★放通防火墙**

# firewall-cmd --add-service=http

# firewall-cmd --add-service=https

# firewall-cmd --runtime-to-permanent

**★查看http配置：**

gitlab-ce自带有nginx软件，查看默认生成的配置：

# cat /var/opt/gitlab/nginx/conf/nginx.conf

# cat /var/opt/gitlab/nginx/conf/gitlab-http.conf

# gitlab-ctl restart nginx #如果修改了nginx的配置，需要重启nginx服务

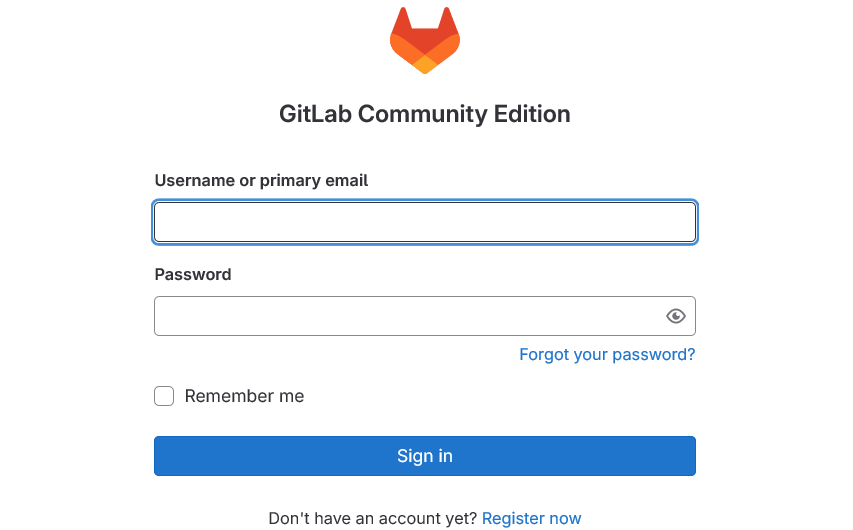
**★查看日志**

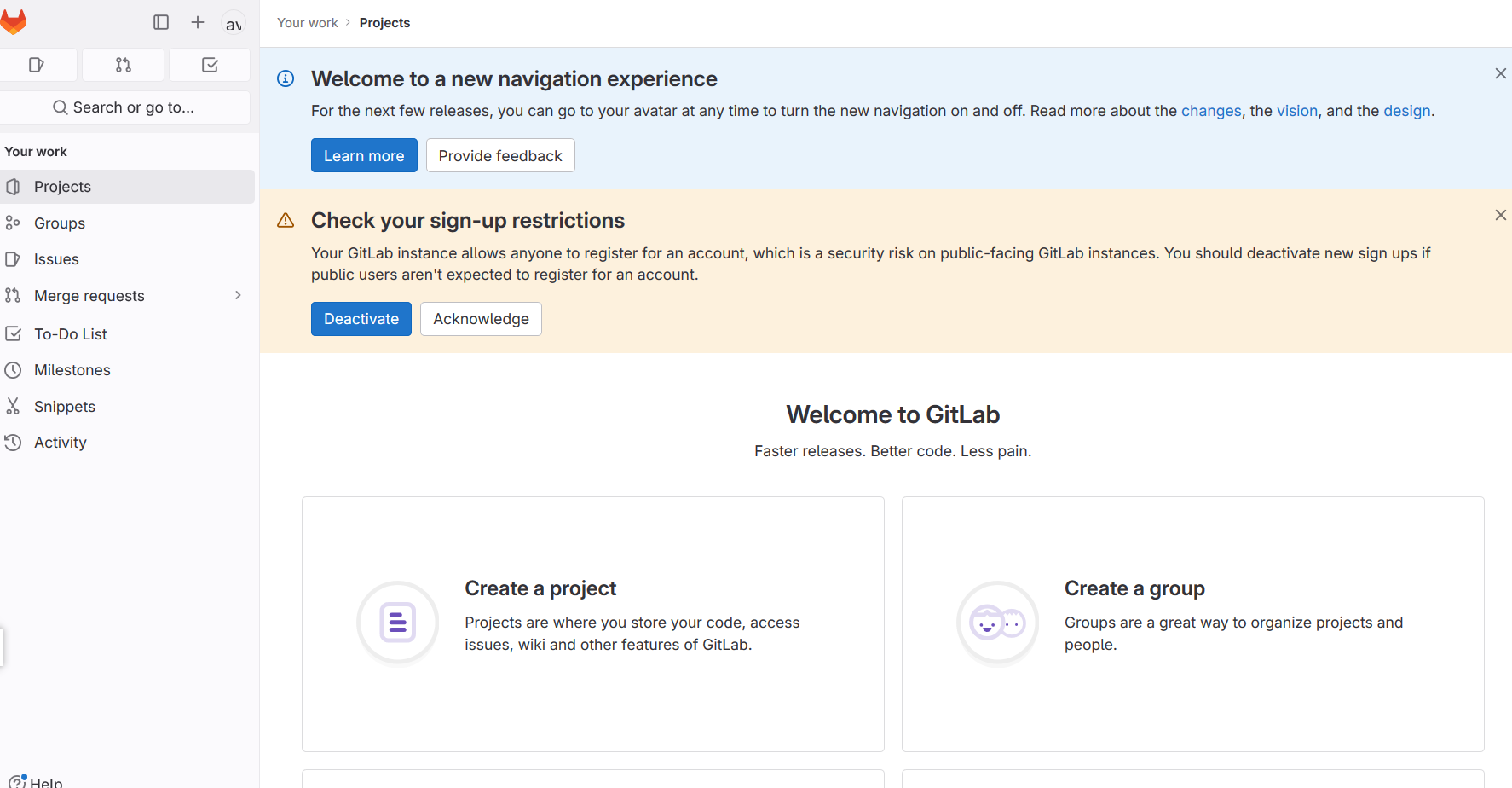
# gitlab-ctl tail #查看日志

# tail -f /var/log/gitlab/gitlab-rails/production.log

# tail -f /var/log/gitlab/gitlab-rails/production\_json.log

**★web界面**

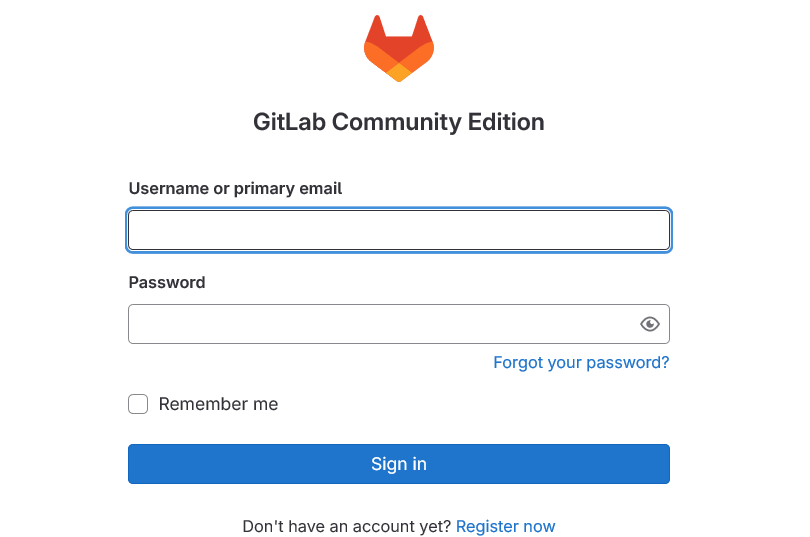




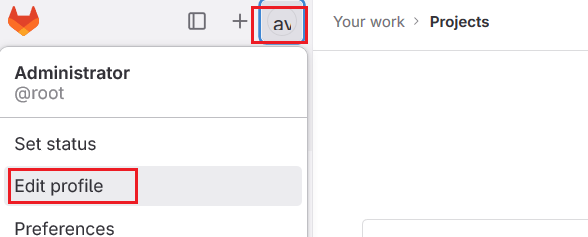
**★管理员账号登录web界面及修改密码**

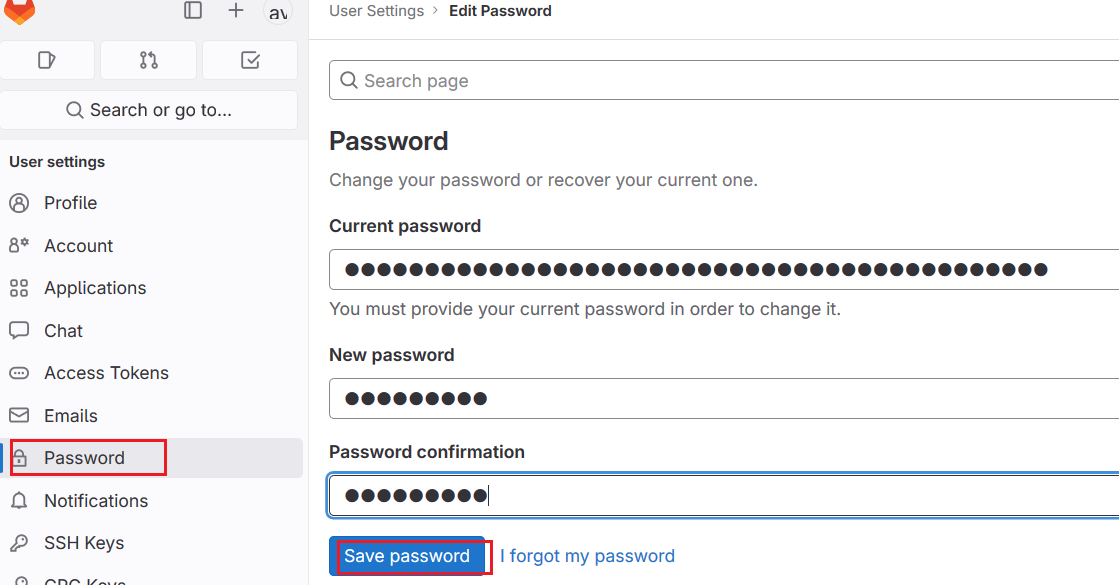
浏览器登录gitlab-ce，用户名root，初始化密码为/etc/gitlab/initial\_root\_password文件里的密码

https://x.x.x.x/



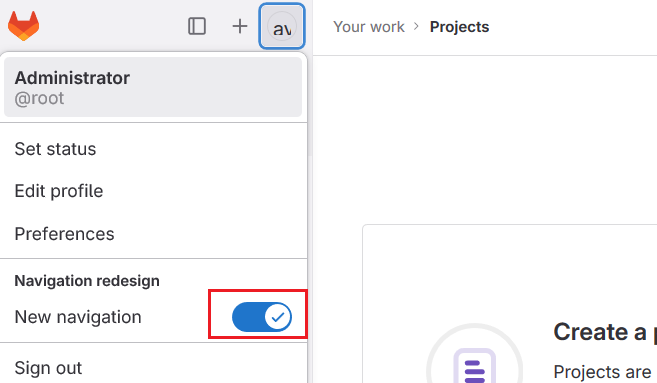
登录后，修改密码



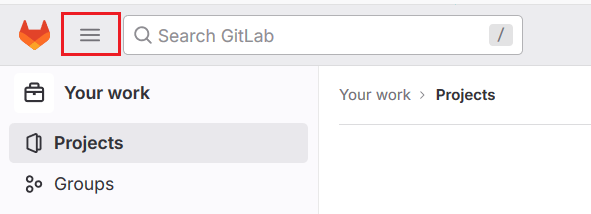


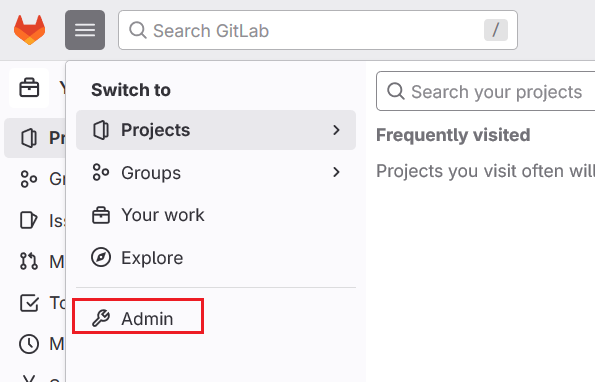
**★关闭用户申请创建账号的功能**

关闭“New navigation”

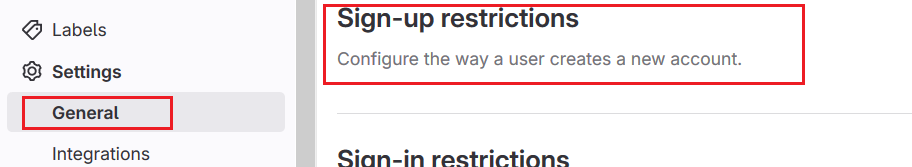


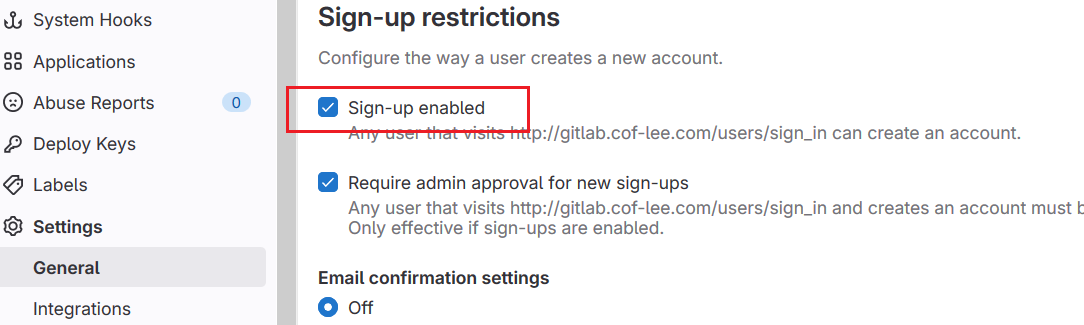
关闭后，左上角出现三道杠，右上角出现用户头像，点击左上角的三道杠





点击“Admin” 或者浏览器直接输入： https://x.x.x.x/admin



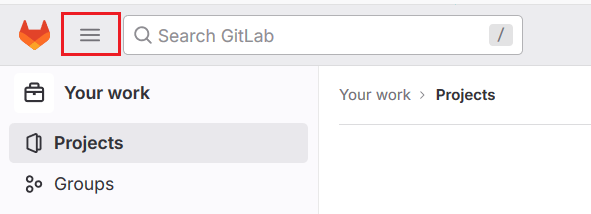


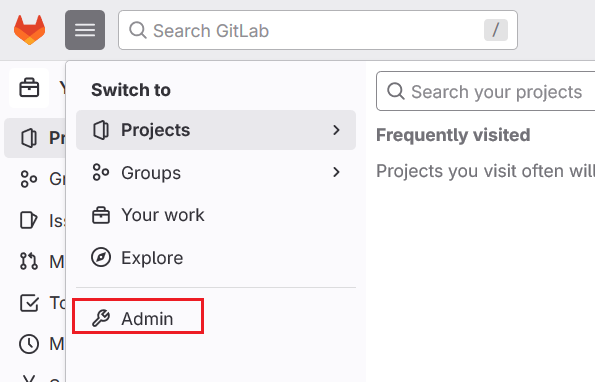
取消勾选“Sign-up enabled”，然后点击下面的“Save changes”之后可关闭“用户申请创建账号”的功能

**★禁用gravatar头像**

gitlab默认使用gravatar头像功能，会访问www.gravatar.com网站，内网环境下导致gitlab页面加载缓慢，可禁用此功能

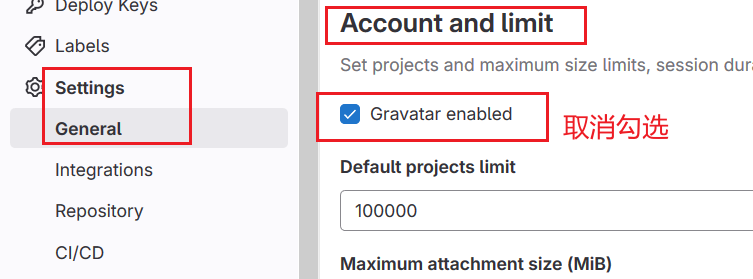
点击左上角的三道杠





点击“Admin” 或者浏览器直接输入： https://x.x.x.x/admin

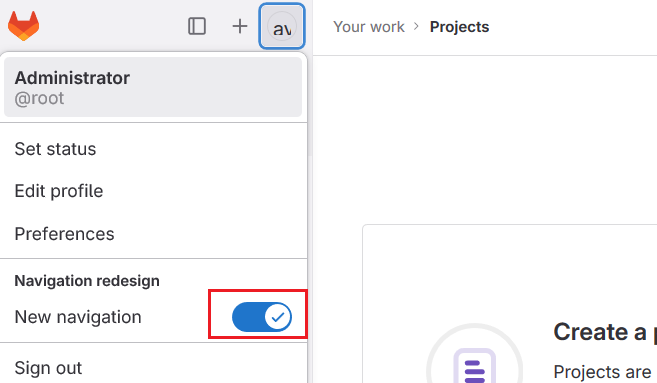
Settings→General→Account and limit



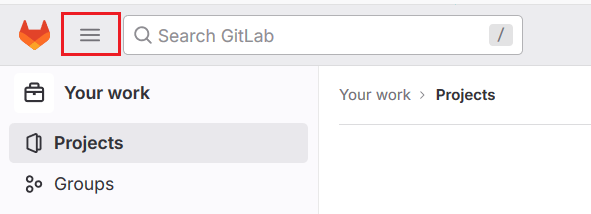
取消勾选“Gravatar enabled”然后点击下面的“Save changes”即可

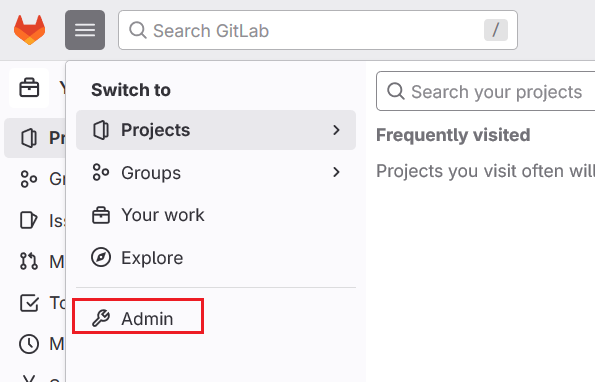
**★创建普通用户账号**

关闭“New navigation”

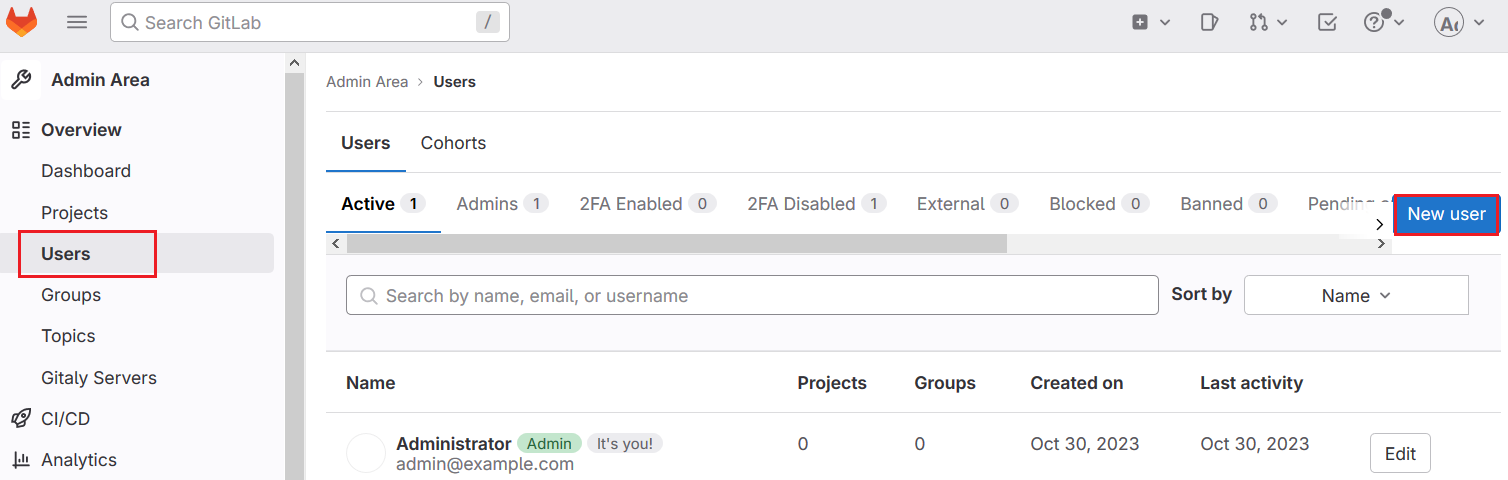


关闭后，左上角出现三道杠，右上角出现用户头像，点击左上角的三道杠

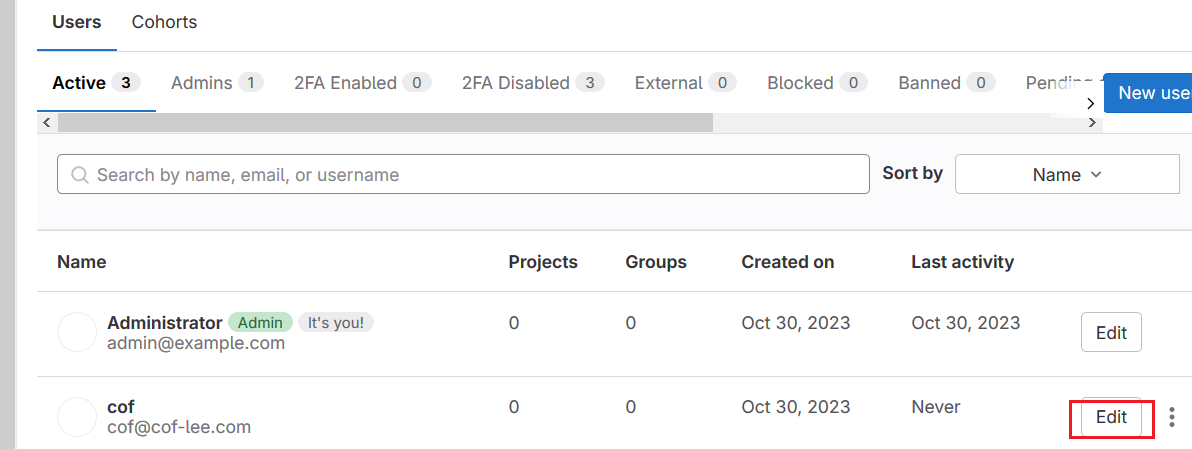


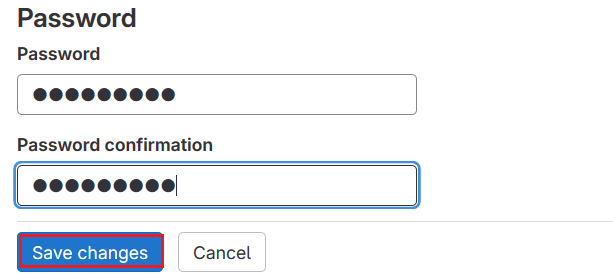


点击“Admin”



创建一个名为cof的用户账号，邮箱为cof@cof-lee.com

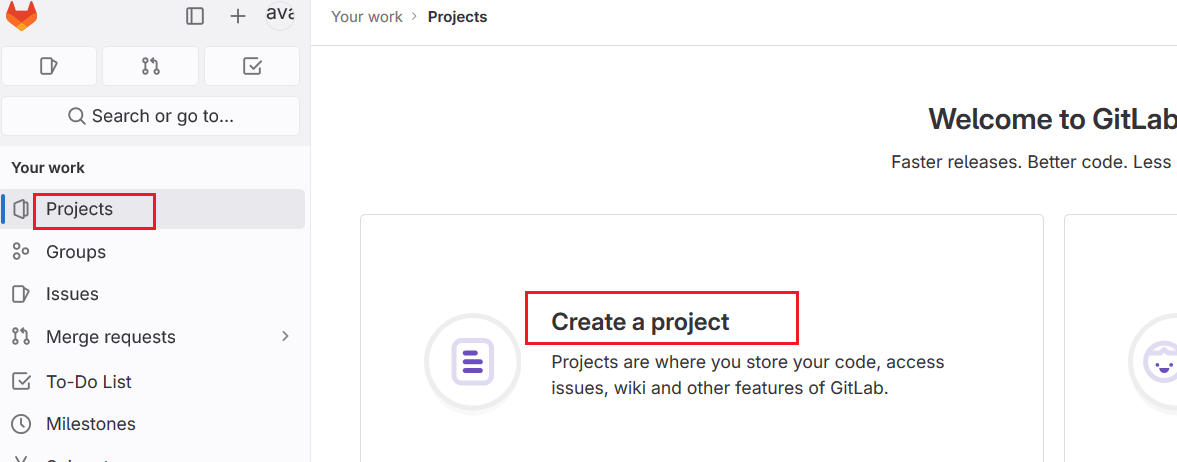


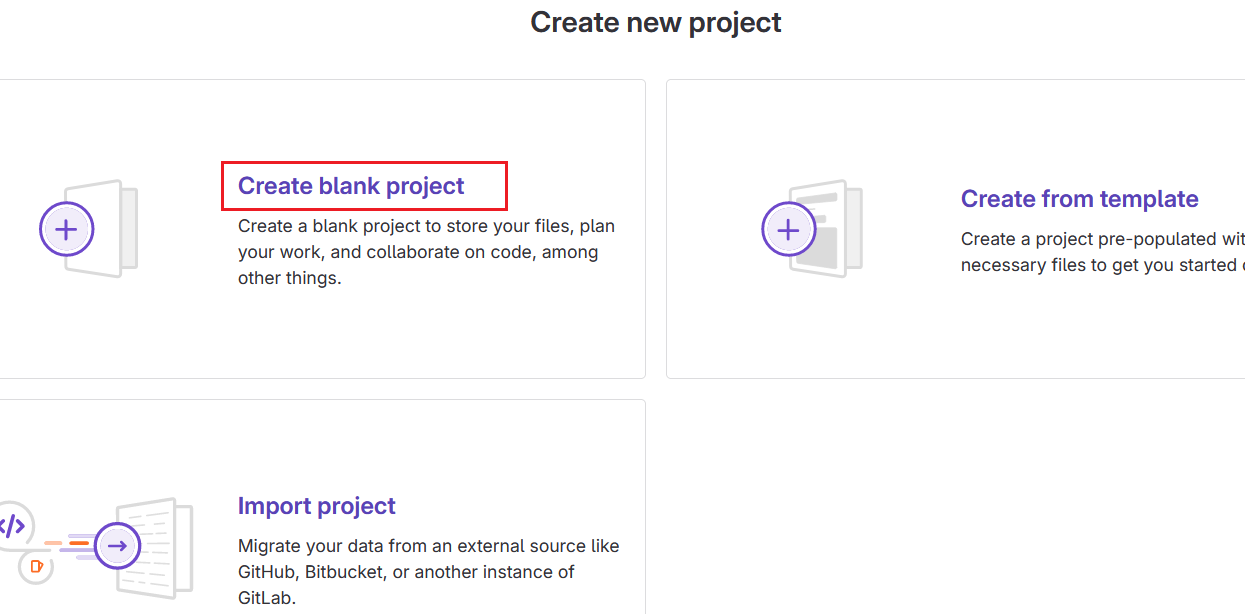


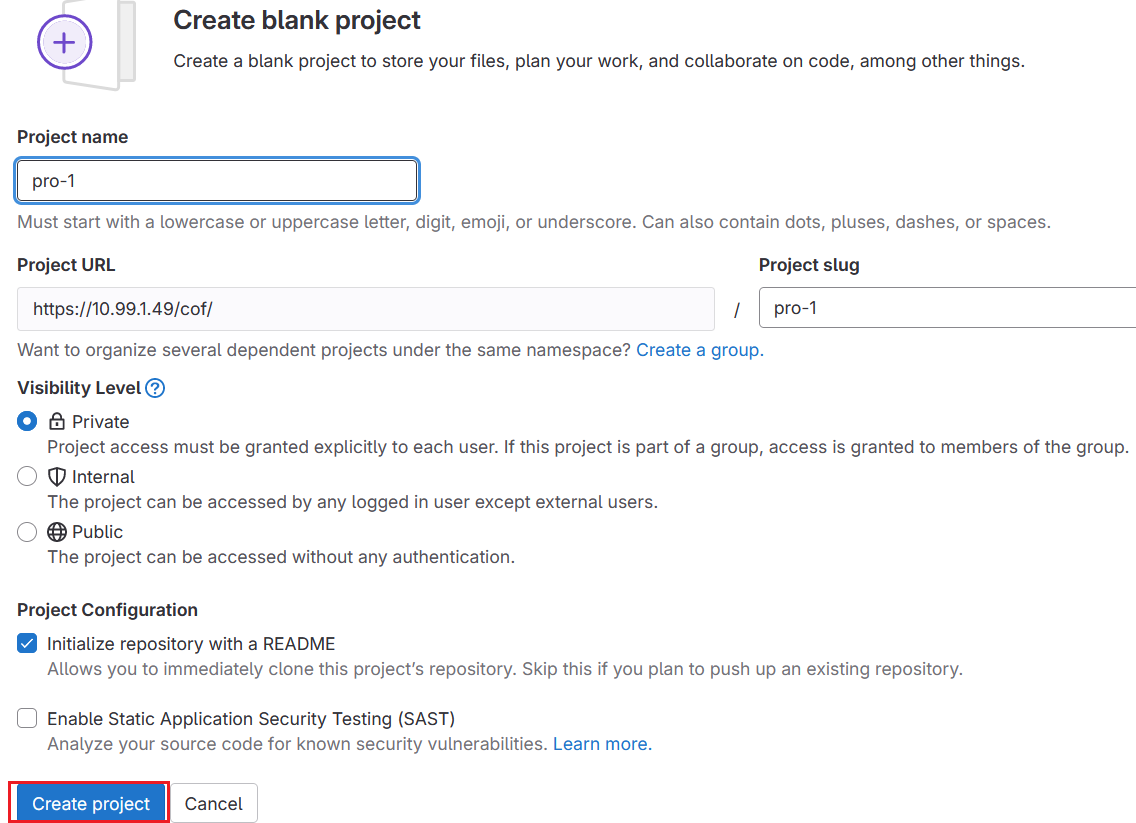
设置好用户密码后，用户初次登录时强制要求修改密码。

**★普通用户登录并创建项目**

普通用户首次登录要求修改密码，然后再次登录



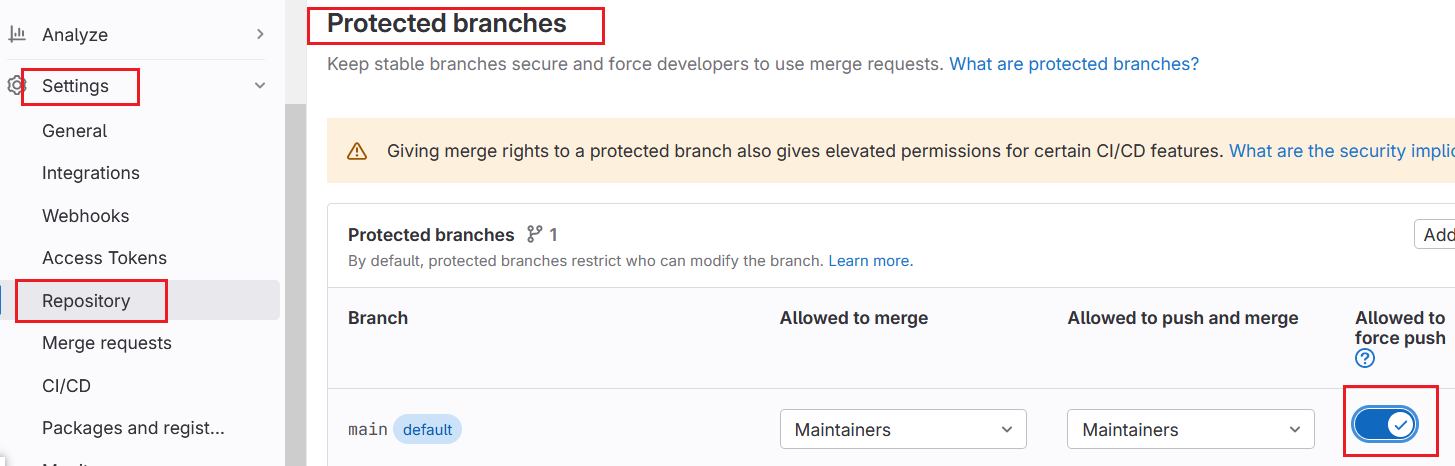




创建一个名为pro-1的项目（仓库）

**★允许客户端push到main分支**

Settings→Repository→Protected branches



勾选“Allowed to force push”

客户端上传分支代码：

# cd ~/testgit #进入项目目录，已初始化了的项目

# git remote add rmtRepo https://gitlab.cof-lee.com/cof/pro-1.git

# git push -uf rmtRepo main

Username for 'https://gitlab.cof-lee.com': cof

Password for 'https://cof@gitlab.cof-lee.com':

Enumerating objects: 3, done.

Counting objects: 100% (3/3), done.

Writing objects: 100% (3/3), 212 bytes | 212.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

To https://gitlab.cof-lee.com/cof/pro-1.git

+ 2447ea5...dd101f2 main -> main (forced update)

branch 'main' set up to track 'rmtRepo/main'.

**★添加用户的ssh密钥**

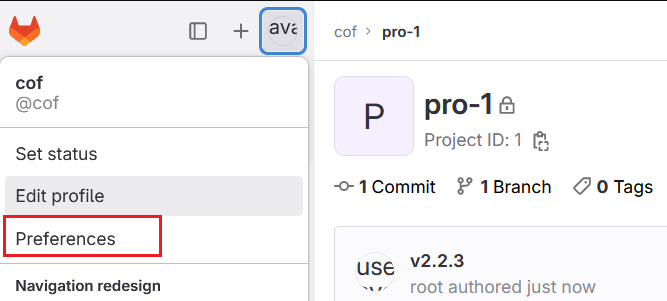
由用户登录gitlab界面，自行添加ssh key

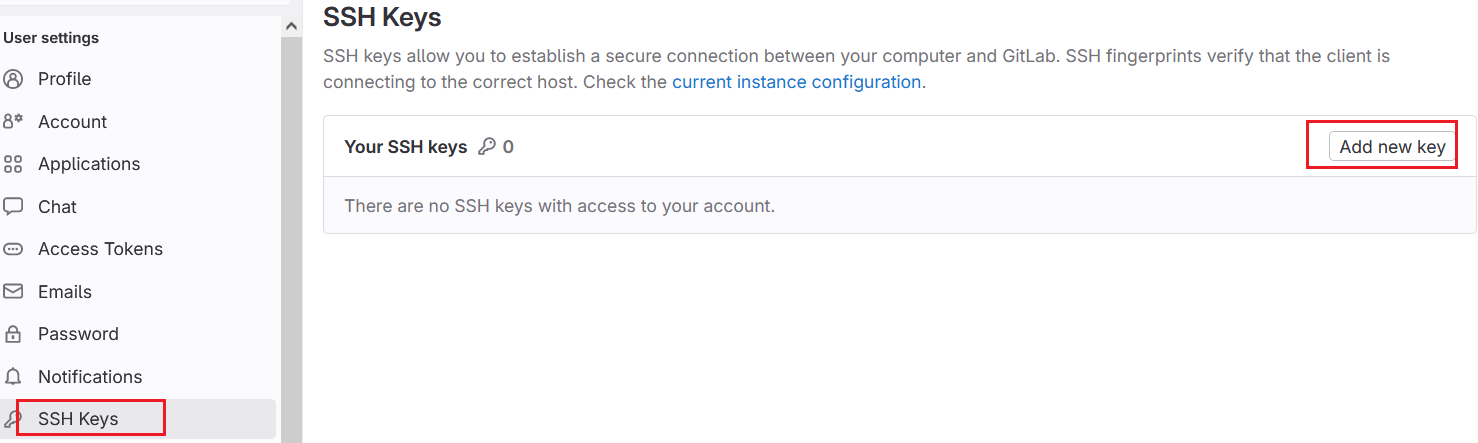
首先在linux服务器上生成用户的ssh密钥对文件

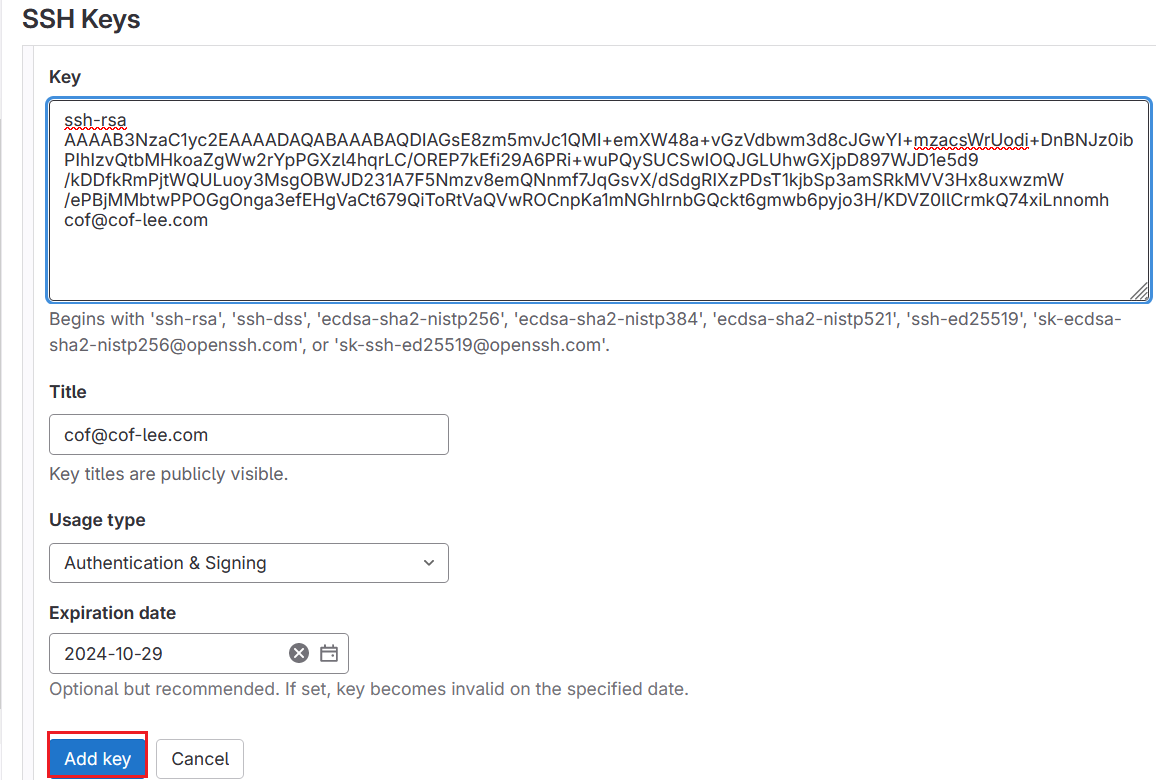
# ssh-keygen -t rsa -b 2048 -C "cof@cof-lee.com" -P "" -f cof-sshkey

#直接生成key文件及公钥.pub文件，**无需交互**，-C指定密钥的描述信息，-P指定加密密钥的密码，-f指定生成的文件名，生成的密钥文件： cof-sshkey为私钥，cof-sshkey.pub为公钥

将**公钥文件**的内容复制粘贴到gitlab上（gitlab上保存的是公钥，私钥留在客户端）







**★客户端使用ssh密钥登录gitlab**

编辑ssh的客户端配置文件，使用ssh登录gitlab.cof-lee.com主机时，指定用户名及ssh私钥

# vi ~/.ssh/config

Host gitlab.cof-lee.com

IdentityFile ~/cof-sshkey

User cof

IdentityAgent none

# ssh git@gitlab.cof-lee.com

PTY allocation request failed on channel 0

Welcome to GitLab, @cof! #出现这行说明已经登录成功了

Connection to gitlab.cof-lee.com closed. #关闭连接是正常的，因为gitlab不提供登录shell

**★使用ssh连接进行克隆操作**

# git clone git@gitlab.cof-lee.com:cof/pro-1.git #首先要配置好~/.ssh/config指定用户名及密钥

Cloning into 'pro-1'...

remote: Enumerating objects: 6, done.

remote: Counting objects: 100% (6/6), done.

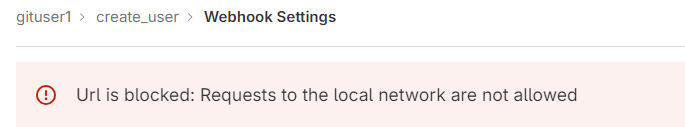
remote: Compressing objects: 100% (3/3), done.

remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 0

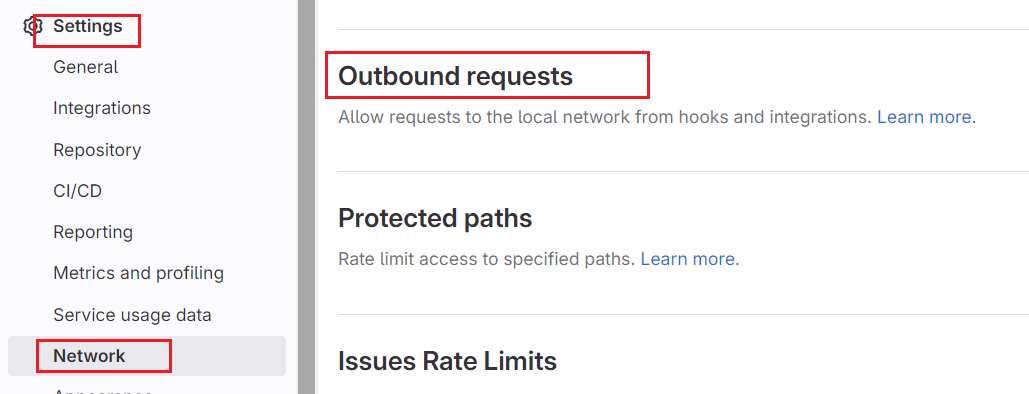
Receiving objects: 100% (6/6), done.

**★配置webhook自动化触发功能**

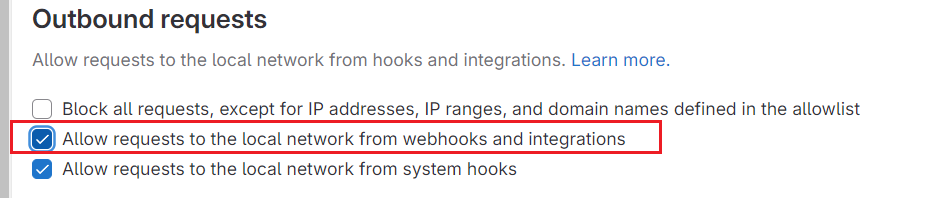
首先配置允许向本地网络（私有网段地址）发送webhook通知，如果不配置则会报错：



管理员用户登录，点击设置：



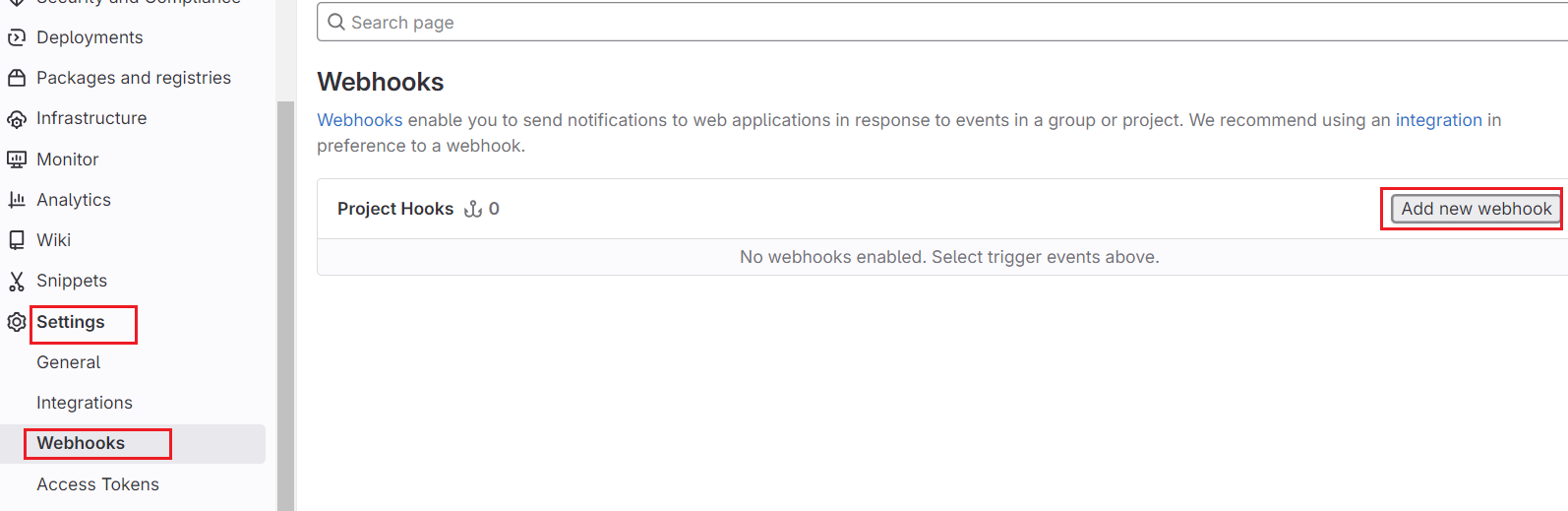
勾选“Allow requests to the local network from webhooks and integrations”

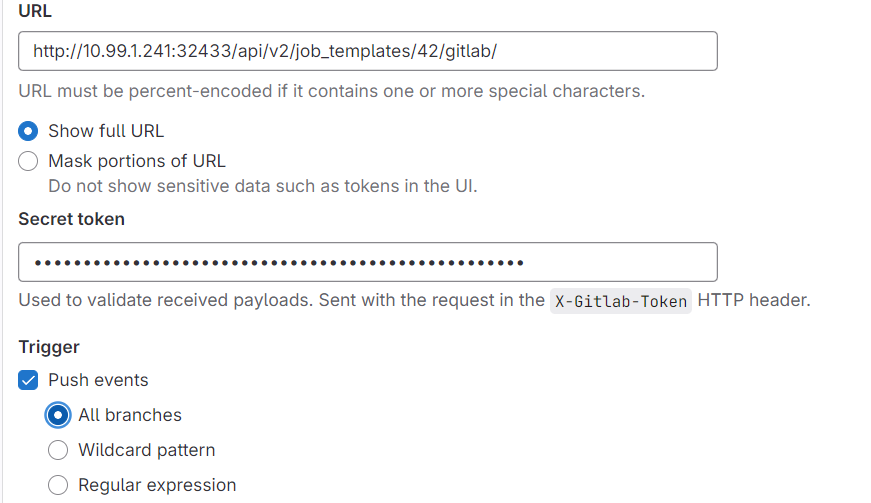


点击“保存”

然后普通用户就可在自己的项目里配置“WebHook”功能了

Settings→Webhooks→Add new webhook





Webhook URL： http://10.99.1.241:32433/api/v2/job\_templates/42/gitlab/

Webhook Key： vHRMIUlRQql9khxfr23sVlaeeMaLA7pZEmbKmmECeFL3bgF7kP

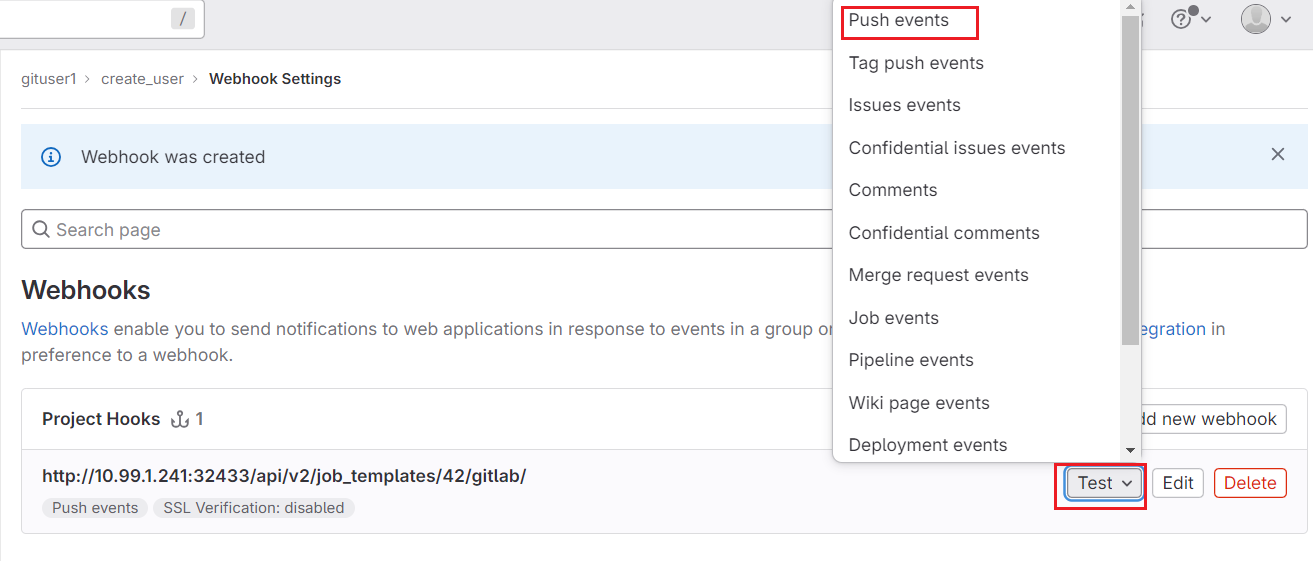
Trigger可选择“Push events”（表示更新代码时），All branches表示所有分支，也可使用“Regular expression”指定目标分支

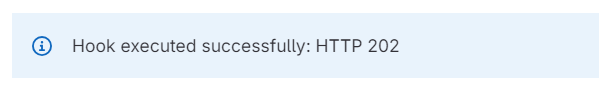


SSL verification根据目标webhook而定，如果Webhook URL不是https则不勾选“Enable SSL verification”

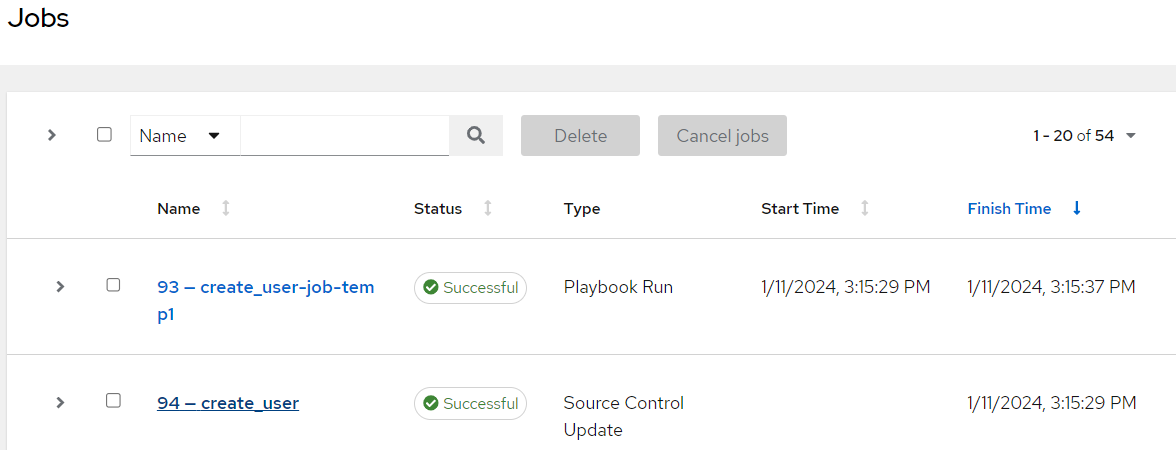
保存后：

可以测试一下：





此操作会触发webhook监听侧的动作（比如ansible-awx自动执行相应的job template）：



**★重置root用户密码**

安装完gitlab-ce后，初始管理密码保存在 /etc/gitlab/initial\_root\_password 文件中，如果文件已删除或忘记root用户密码了，可以重置，方法如下：

# cd /opt/gitlab/bin/

# gitlab-rails console -e production #进入交互界面

> User.all #查看所有用户，确认root用户id为1

> u=User.where(id:1).first

> u.password='pasdx2023' #设置密码

> u.password\_confirmation='pasdx2023'

> u.save!

> exit