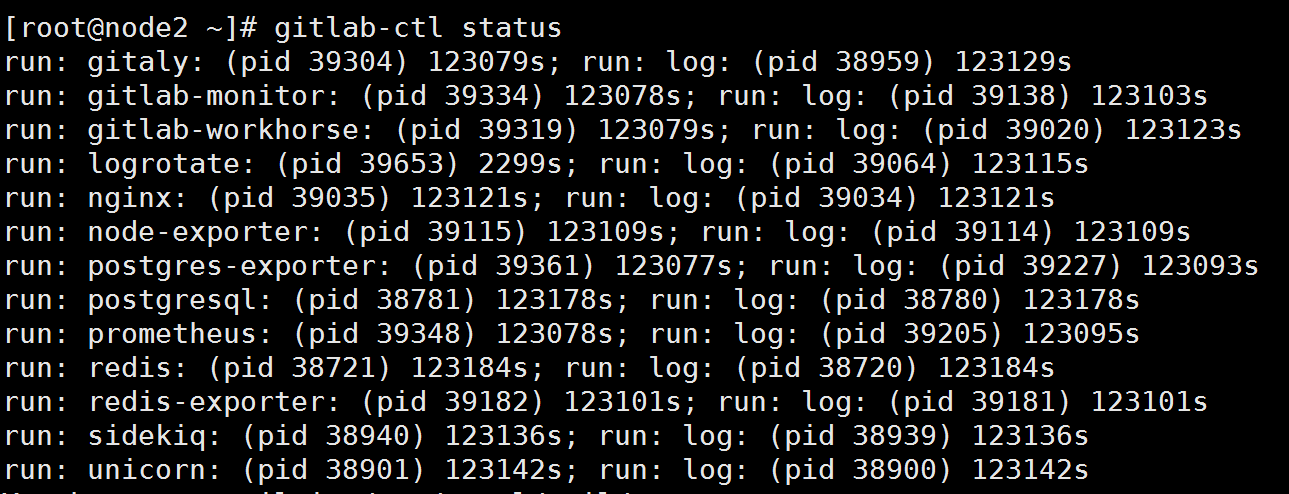
# Gitlab使用

# 一、常用命令

gitlab-ctl status:查看gitlab组件状态



gitlab-ctl start:启动全部服务

gitlab-ctl restart:重启全部服务

gitlab-ctl stop：停止全部服务

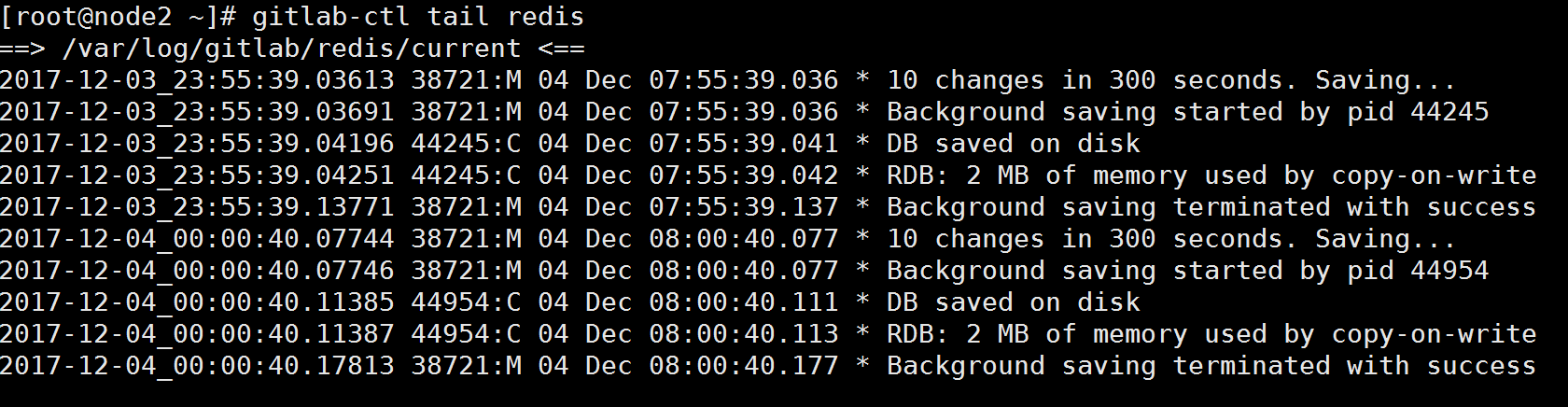
gitlab-ctl reconfigure:使配置文件生效（一般修改完主配置文件/etc/gitlab/gitlab.rb，需要执行此命令）

gitlab-ctl show-config ：验证配置文件

gitlab-ctl uninstall:删除gitlab（保留数据）

gitlab-ctl cleanse:删除所有数据，从新开始

gitlab-ctl tail <service name>查看服务的日志



# 二、常用的组件

nginx：静态Web服务器

gitlab-shell：用于处理Git命令和修改authorized keys列表，我们的gitlab是以Git做为最层的，你操作实际上最后就是调用gitlab-shell命令进行处理。

gitlab-workhorse:轻量级的反向代理服务器

logrotate：日志文件管理工具

postgresql：数据库

redis：缓存数据库

sidekiq：用于在后台执行队列任务（异步执行）

unicorn：GitLab Rails应用是托管在这个服务器上面的

# 三、基础目录：

/var/opt/gitlab/git-data/repositories：库默认存储目录

/opt/gitlab： 应用代码和相应的依赖程序

/var/opt/gitlab：gitlab-ctl reconfigure命令编译后的应用数据和配置文件，不需要人为修改配置

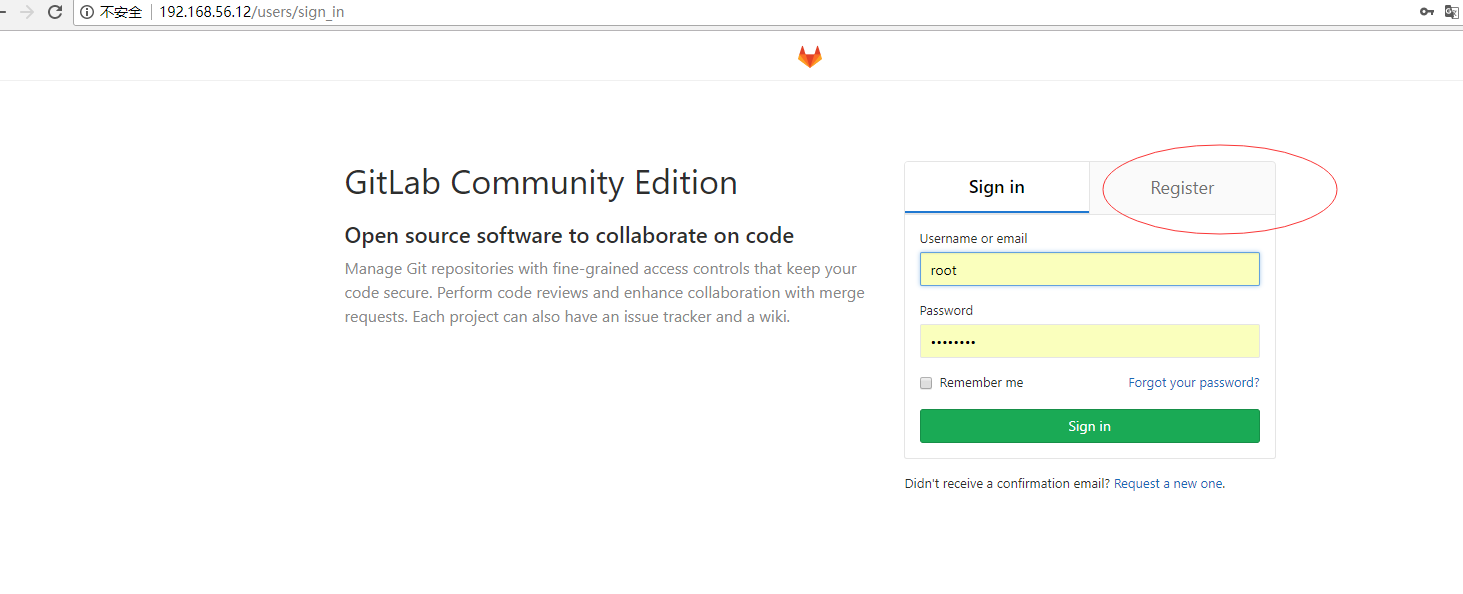
/etc/gitlab： 配置文件目录

/var/log/gitlab：此目录下存放了gitlab各个组件产生的日志

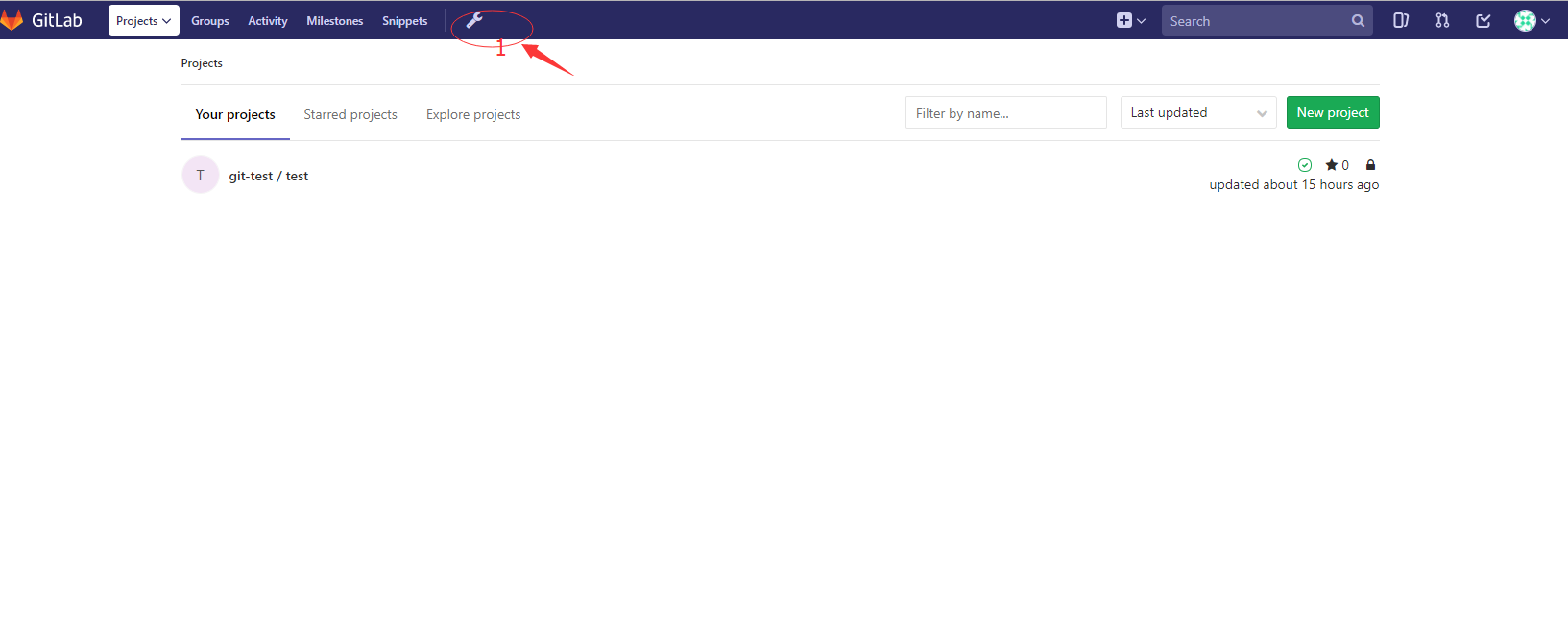
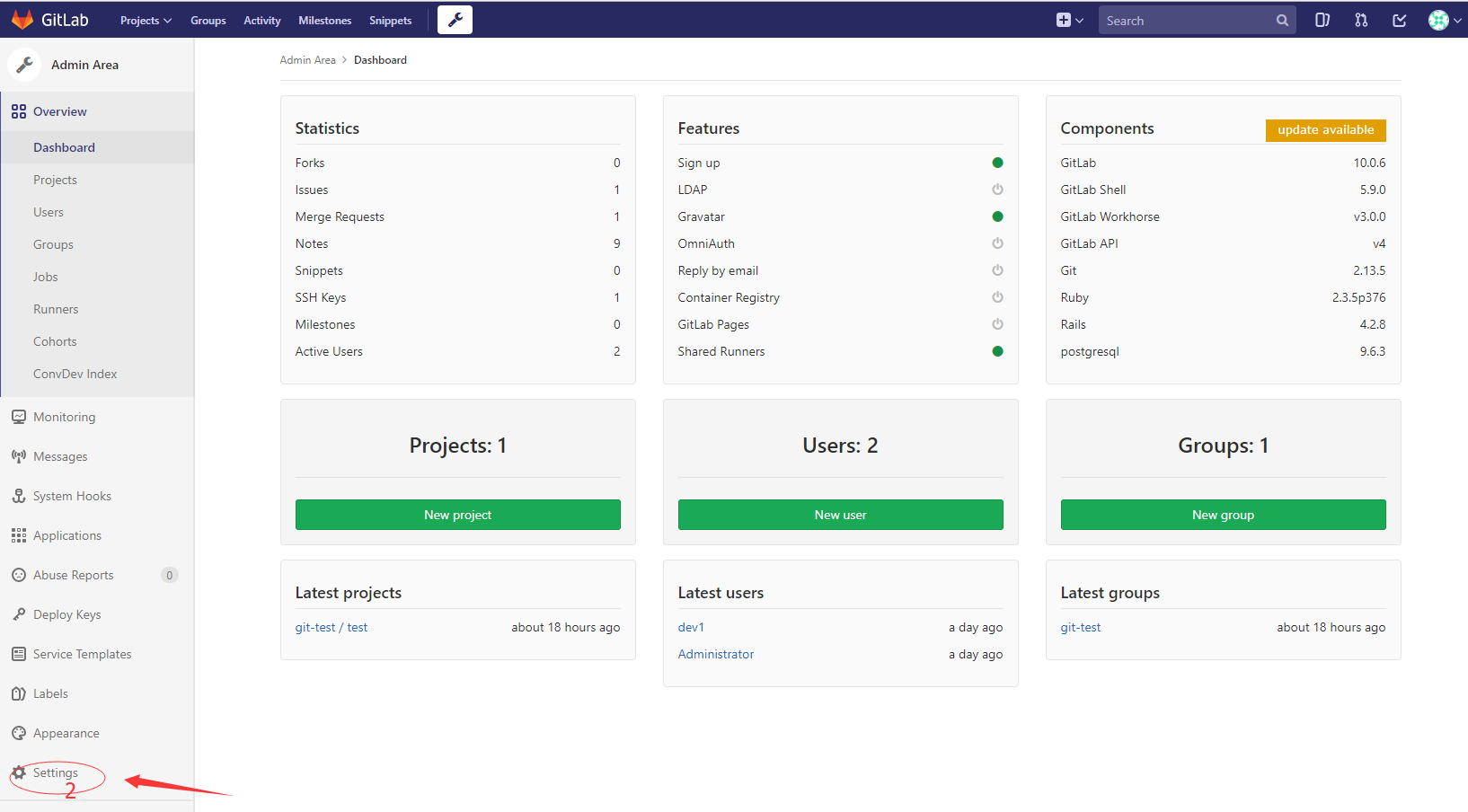
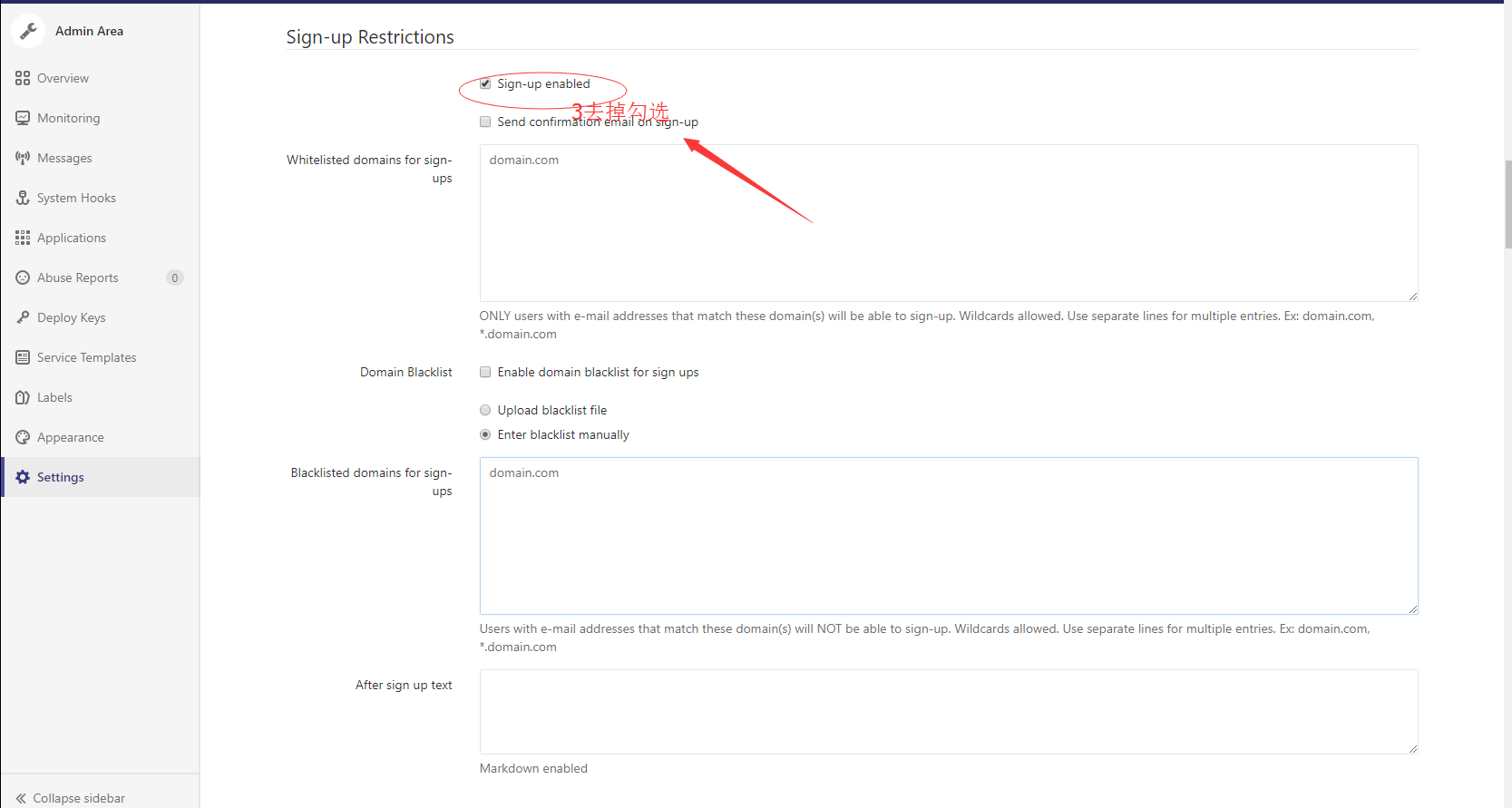
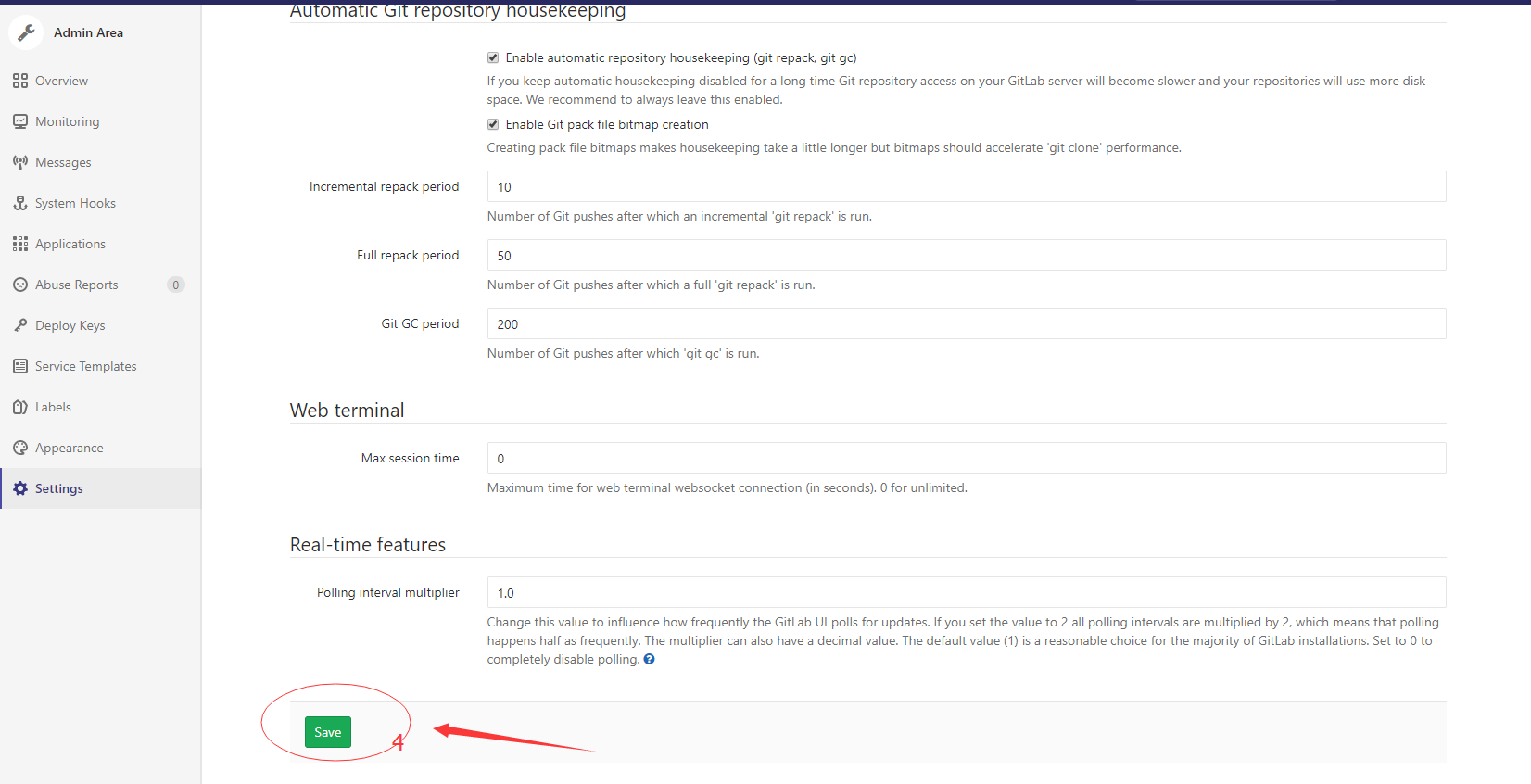
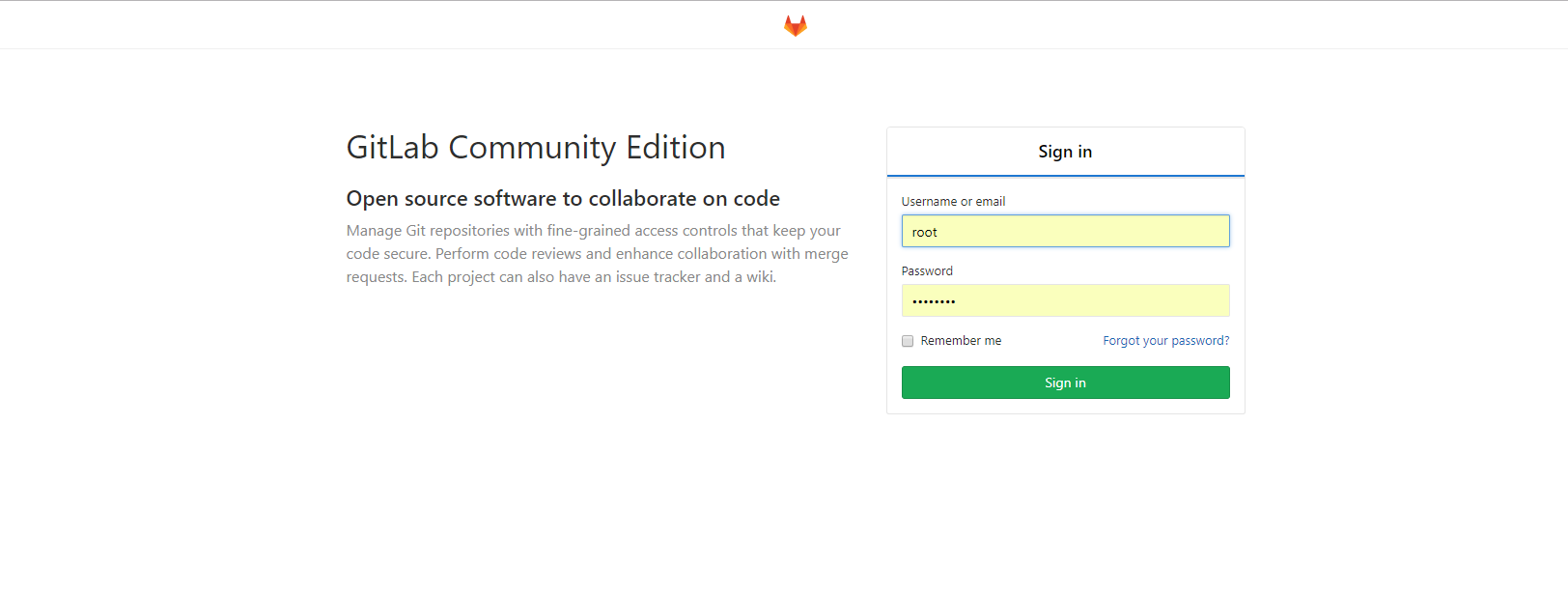
/var/opt/gitlab/backups/：备份文件生成的目录

# 四、Gitlab基本配置

## 1、关闭注册



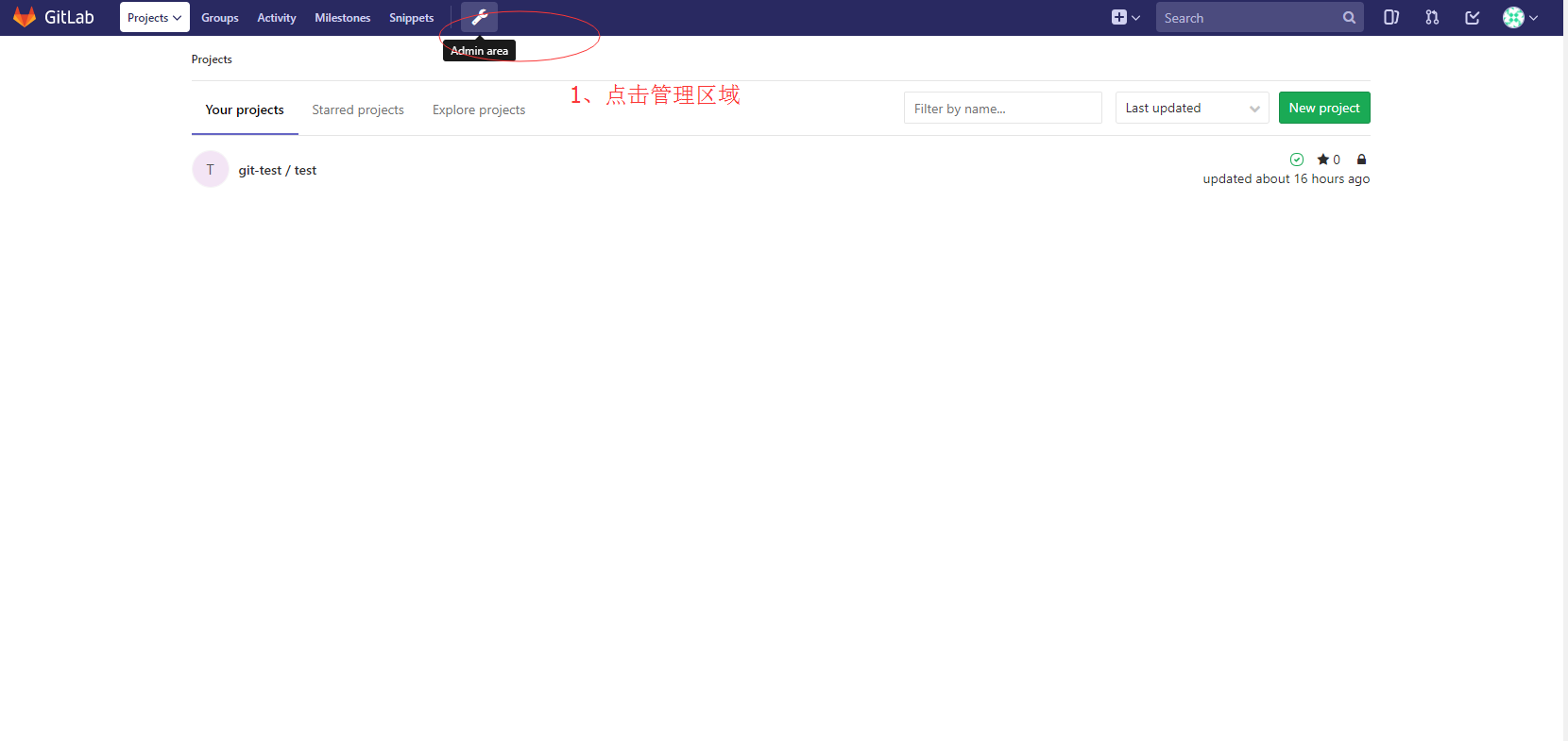
由于我们Gitlab系统是私有仓库，一般用户都是由管理员创建和分派的，所以我们需要关闭注册。

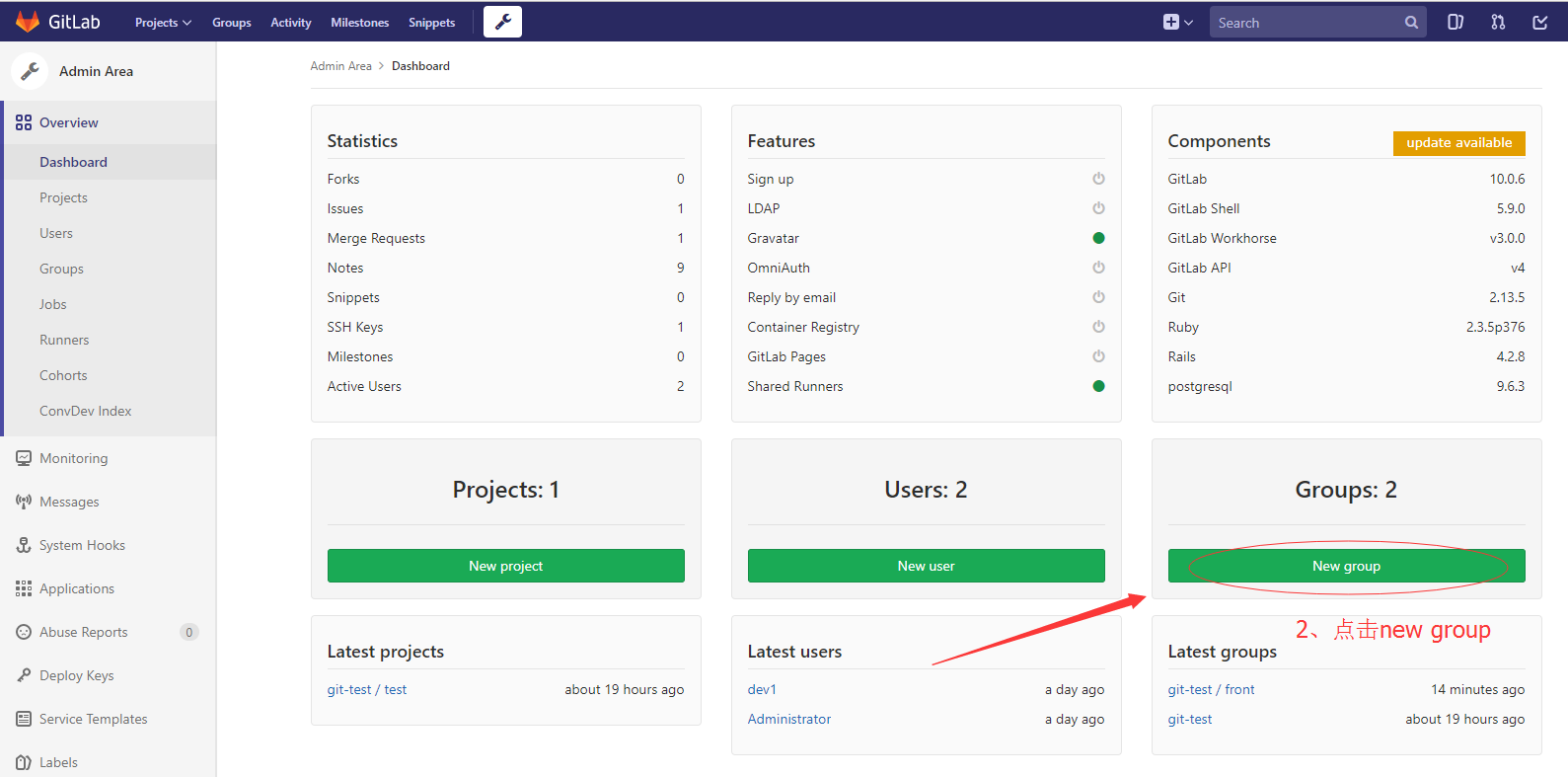
    

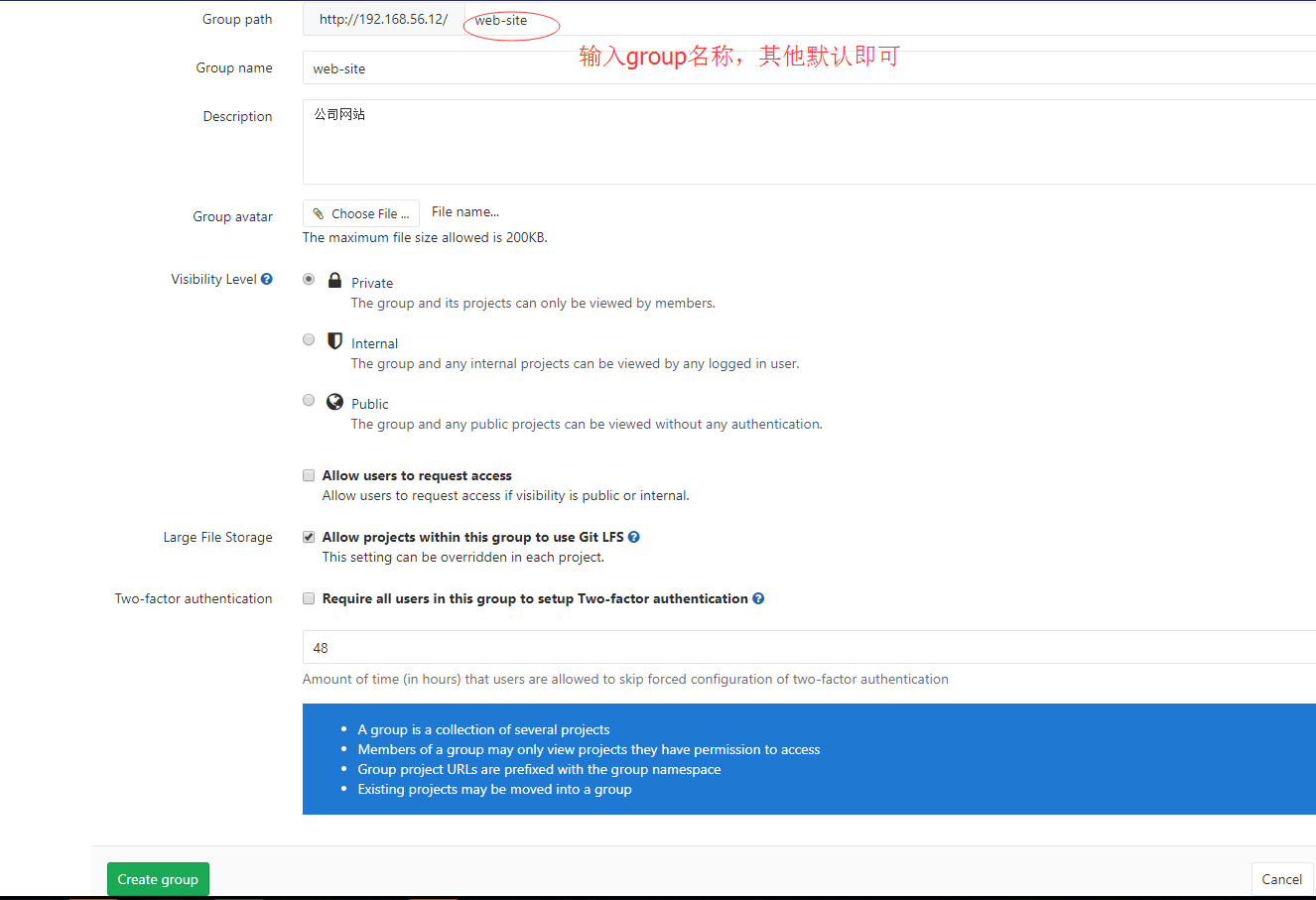
## 2、create group

group（项目）下面可以创建subgroup，创建project（项目下的具体工程），添加user。group就是把相关的project或者user放在一起，进行统一的权限管理。

我们创建一公司网站的项目web-site，项目下面有两个工程，一个是前台pro-frontend， 一个是后台管理pro-backend。





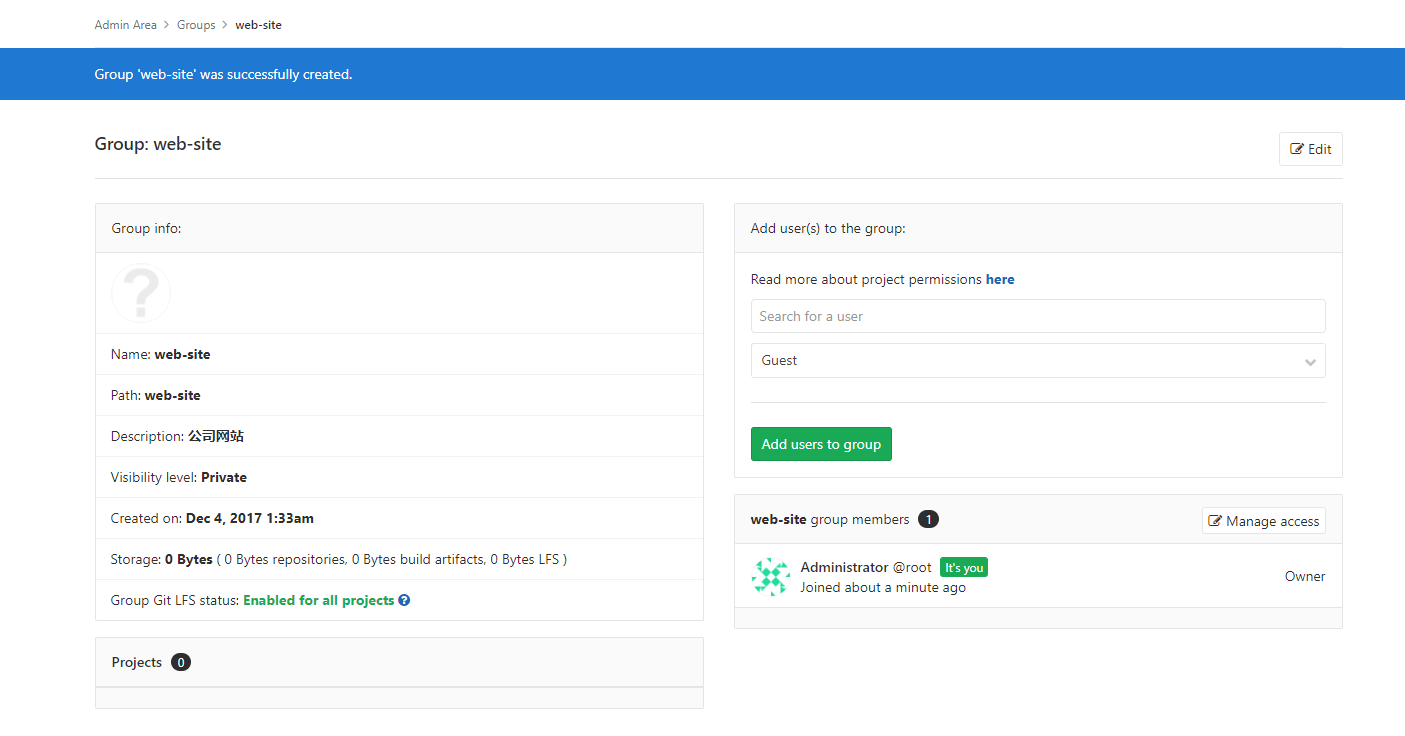
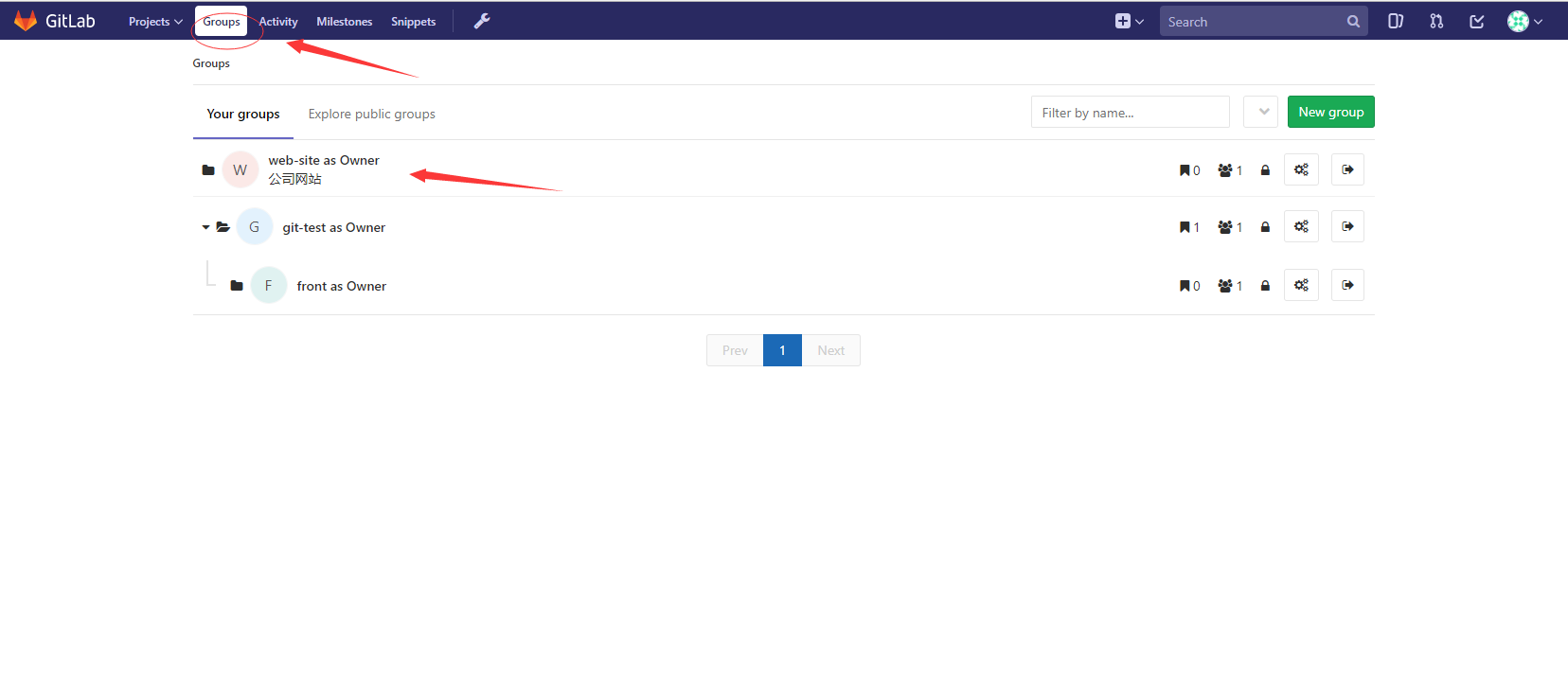


visibility Level：选择谁可以访问该组：我们默认选择private,因为我建设的是私有仓库

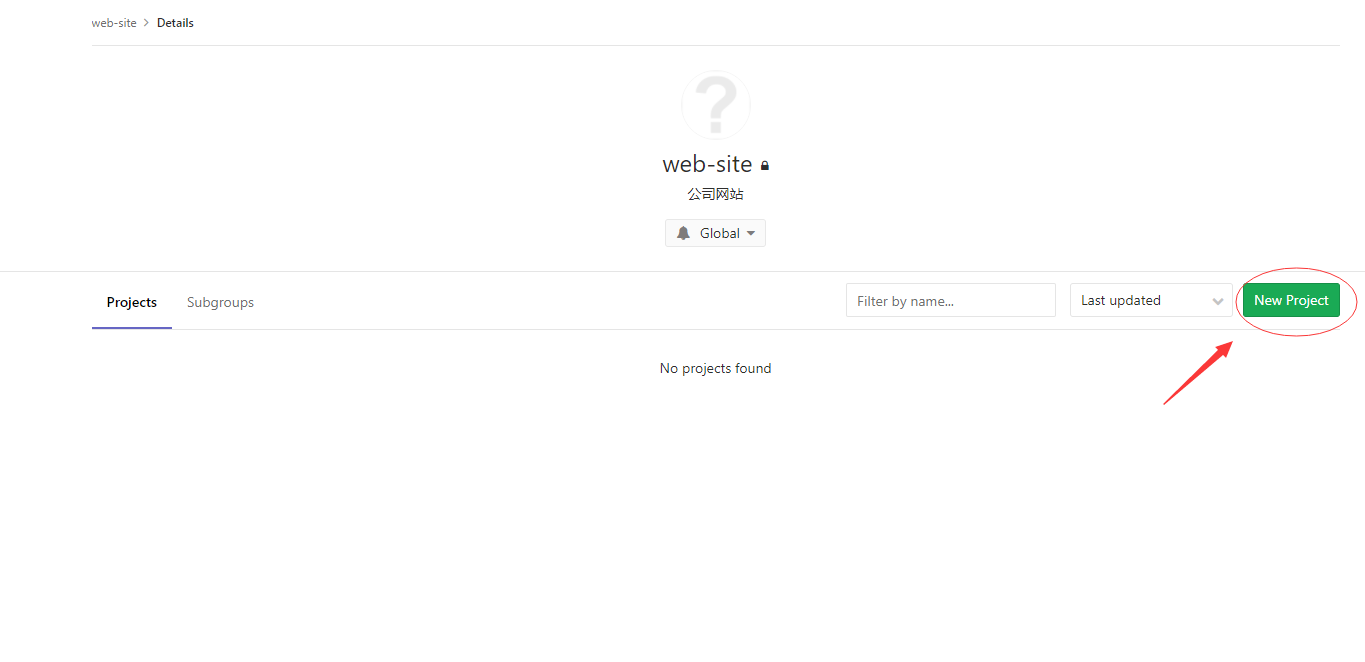
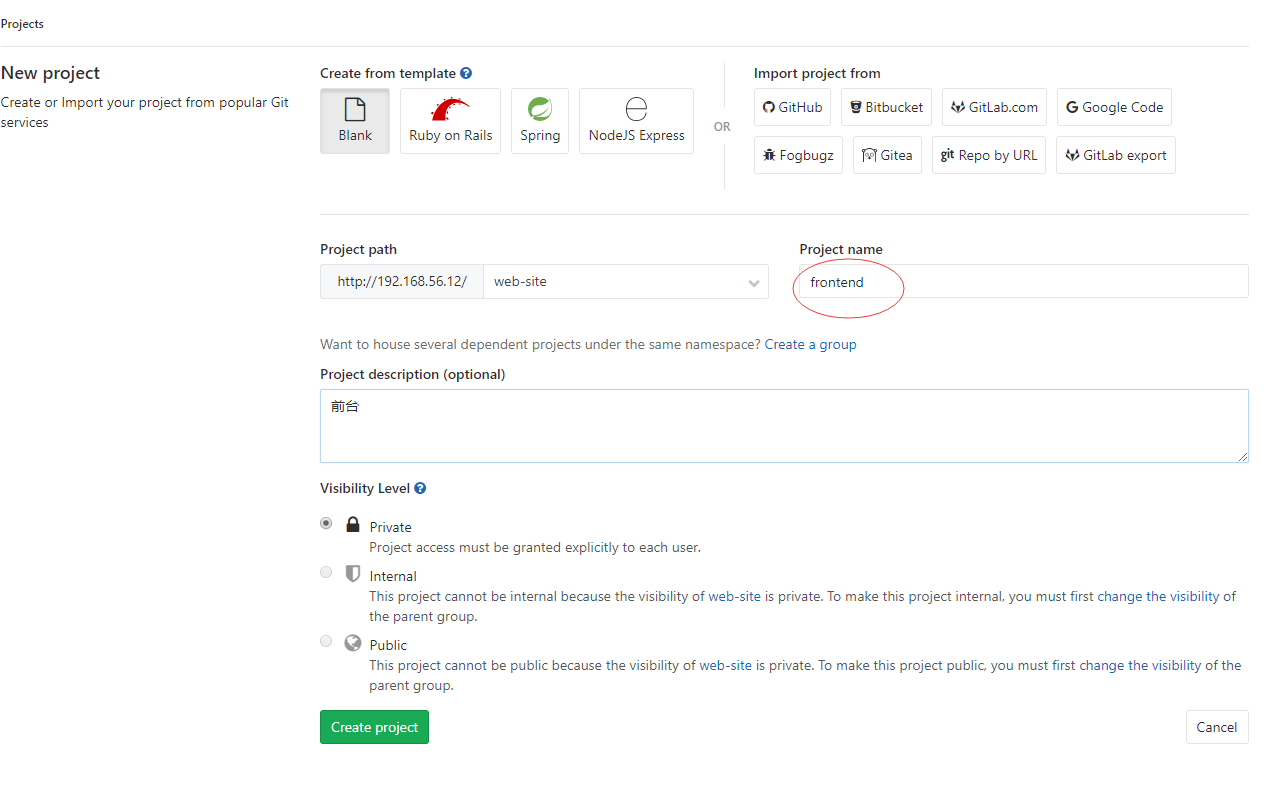
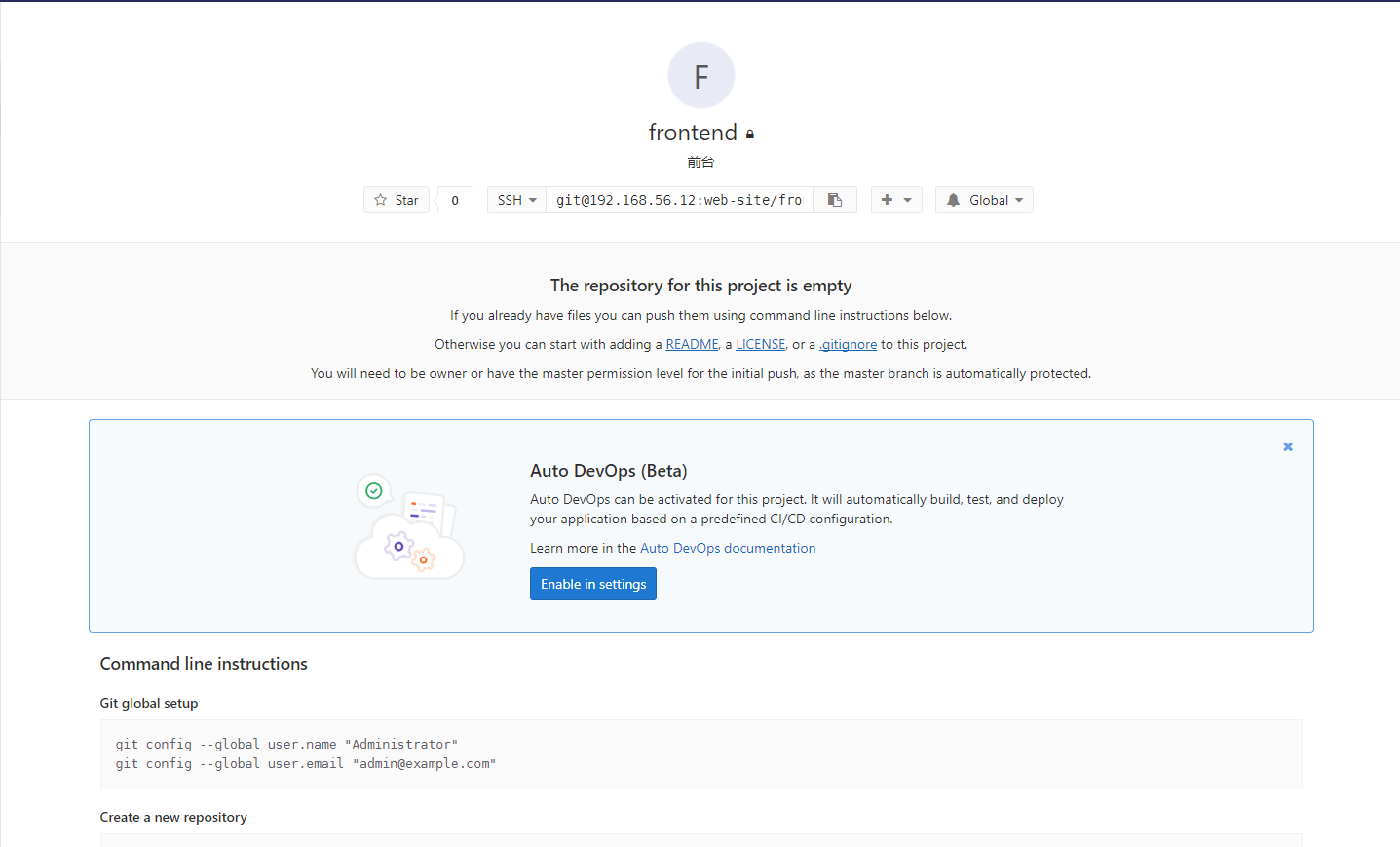
Private:只有授权的用户才可以看到

Internal：只要是登录gitlab的用户就可以看到

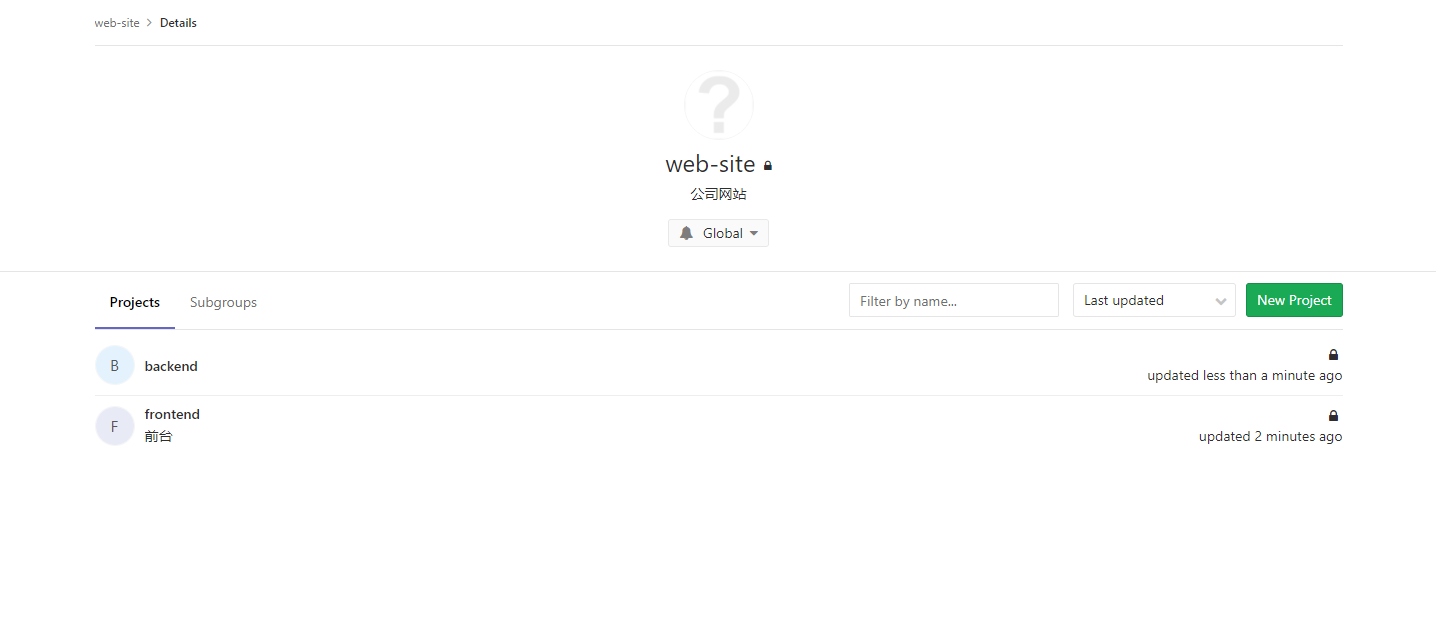
Public：只要可以访问gitlab web页面的人就可以看到

点击group名称进去，我们在web-site下面创建project:

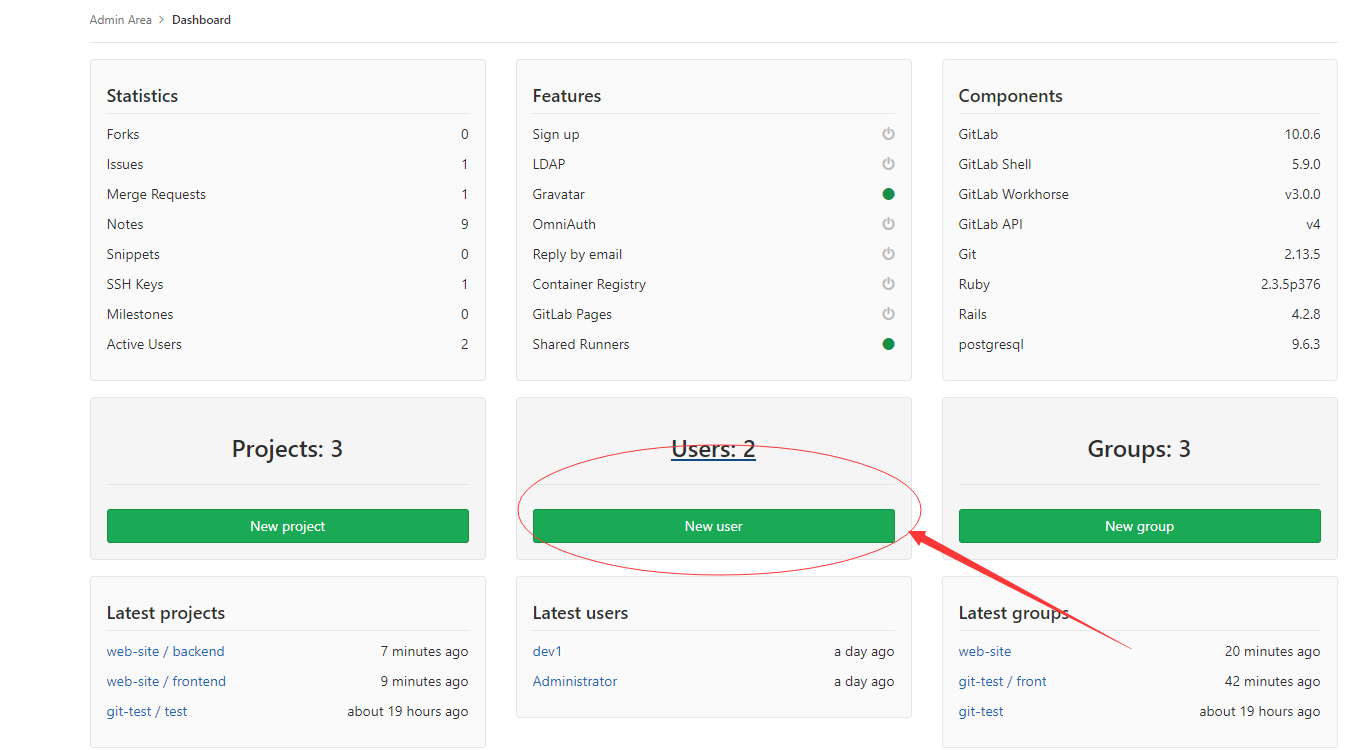
此时我们已经创建了一个frontend的project,是一个空的工程。我们暂时先不管其他的，使用同样的方法创建backend project。

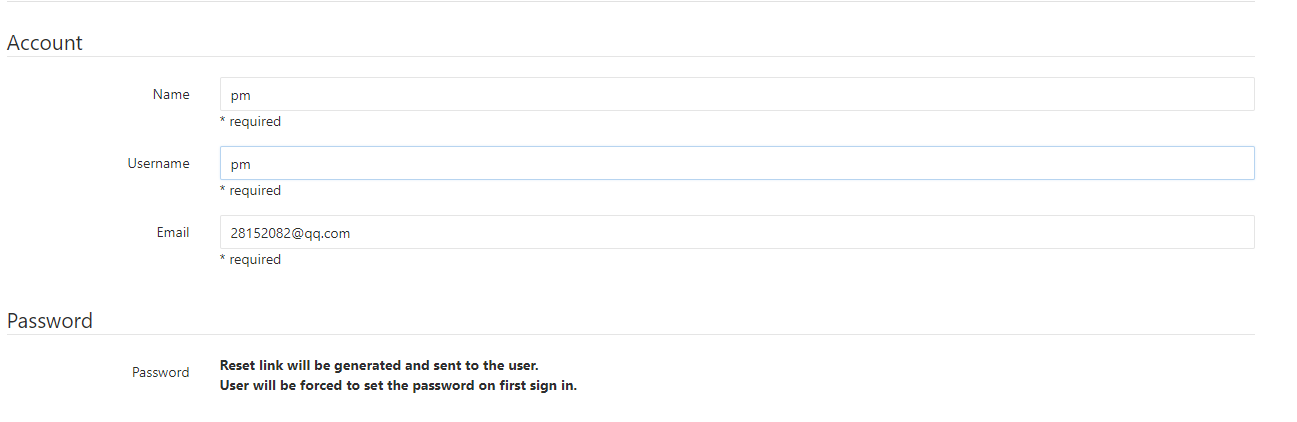


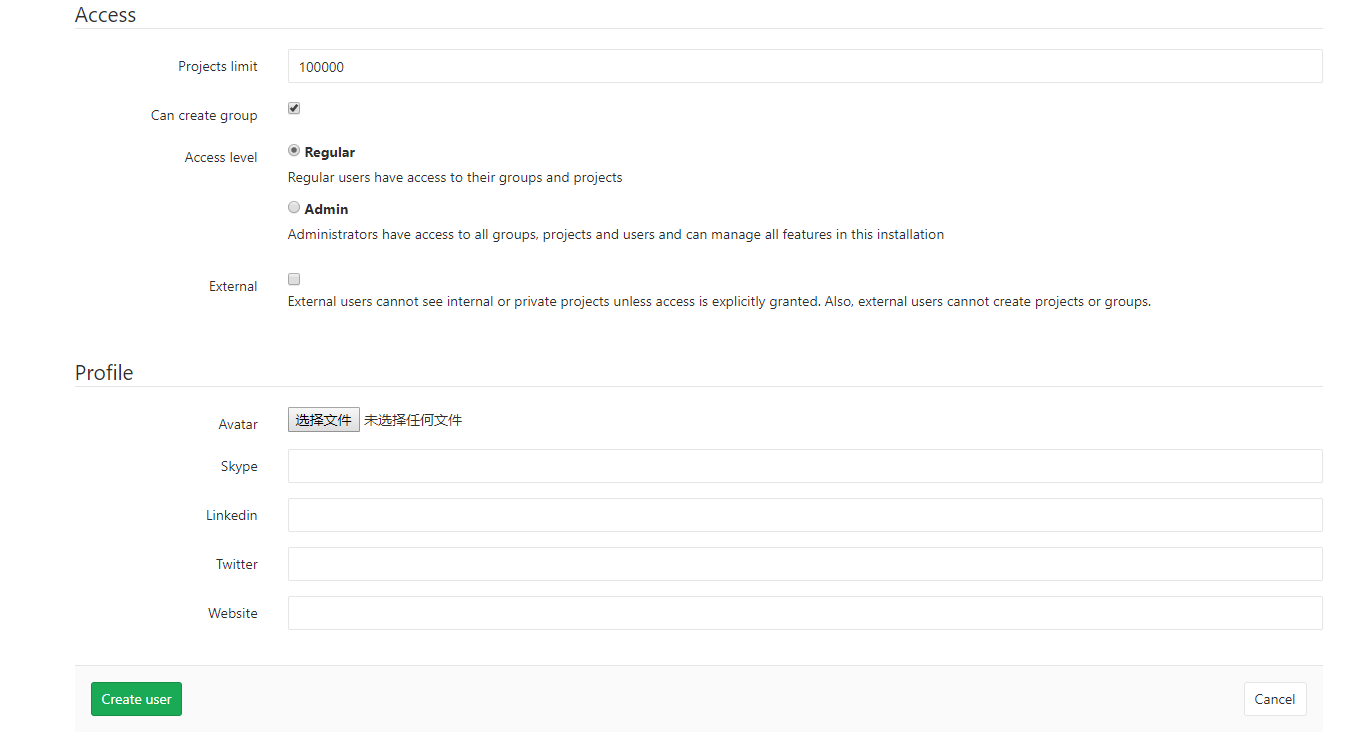
subgroup创建同group同样，在此我们不再详细说明。

## 3、create user

我们主要创建这两类用户，一类是项目经理PM（用来管理项目），另一类是开发DEV（项目功能的实现）。

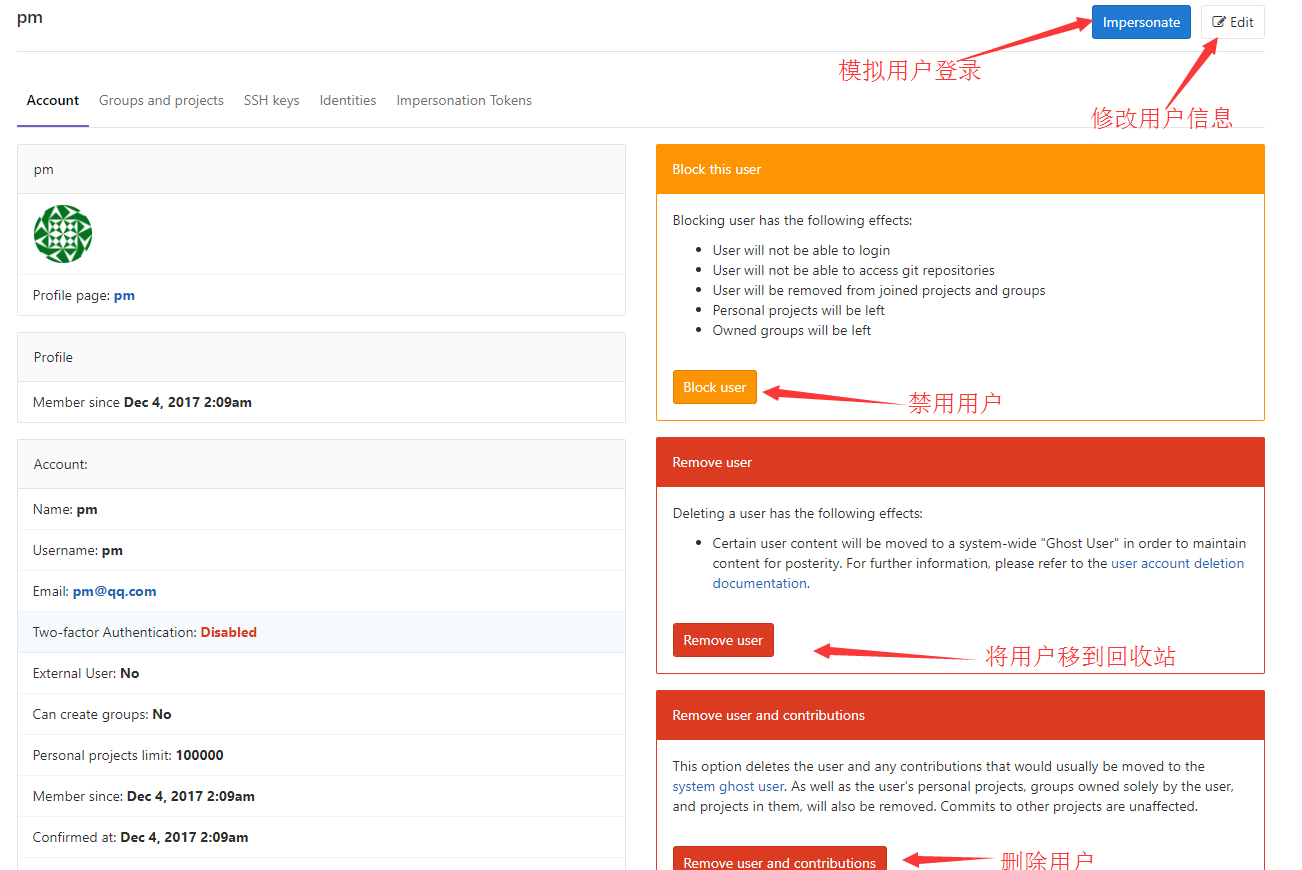




注：1、邮件不能重复；

2、新建用户不能设置密码，需要我们在添加完用户名，编辑用户并为用户设置一个初始密码，用户第一次登录时系统要求用户更改密码；

3、去掉勾选，普通用户我们一般不需要create group。

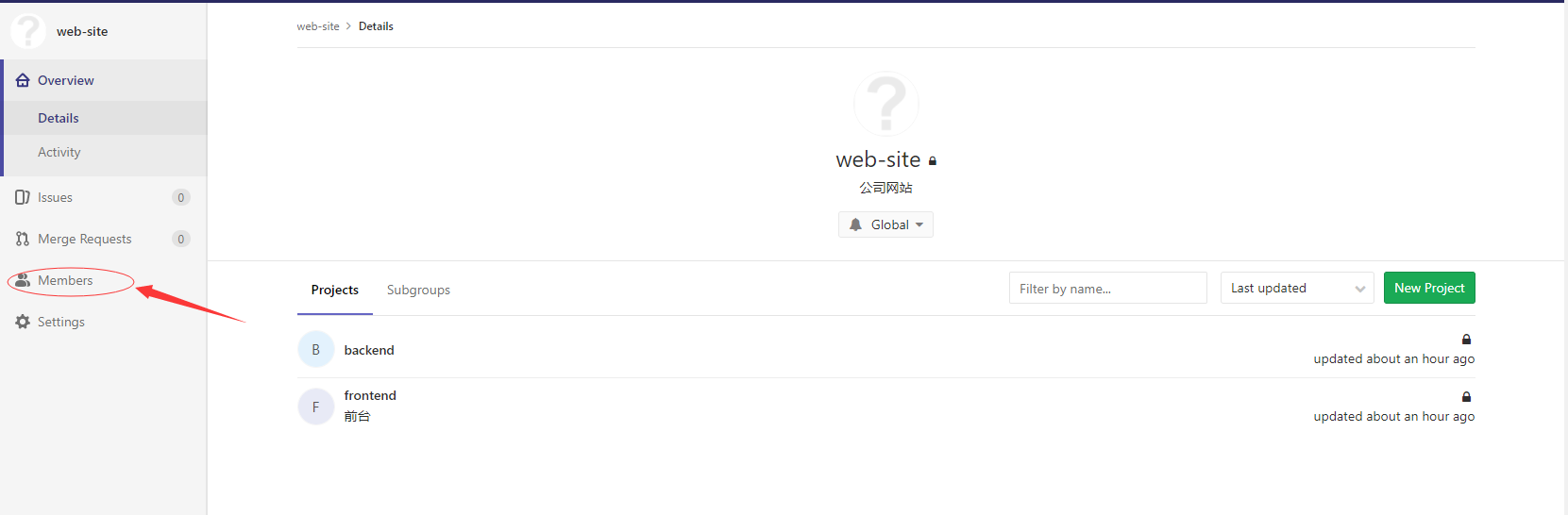


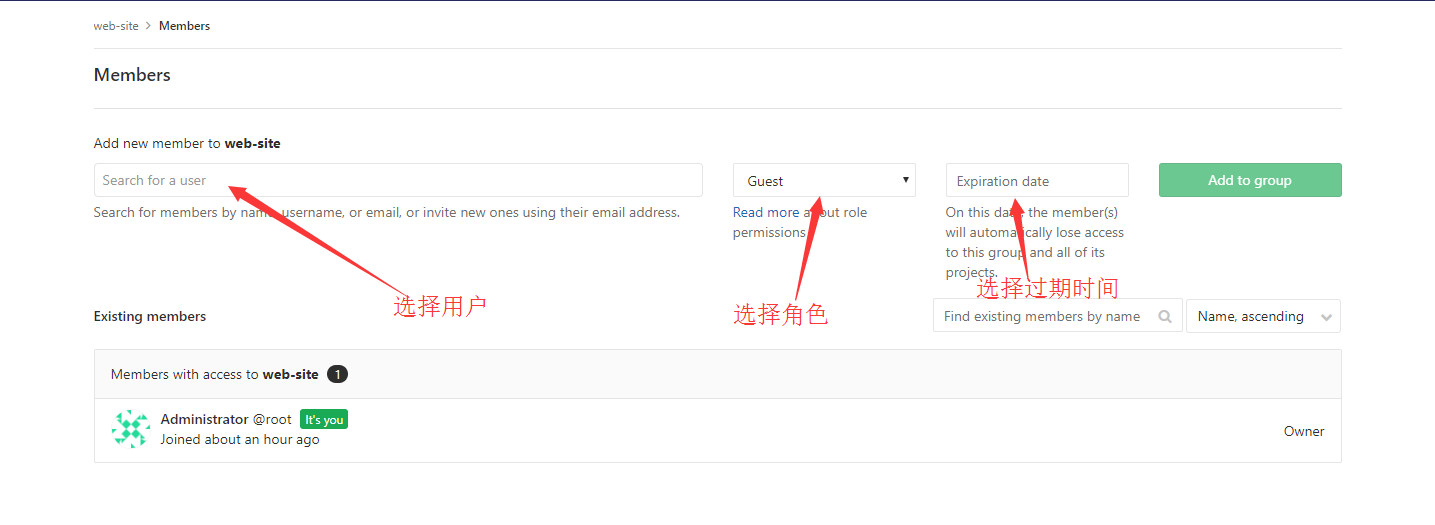
重复以上步骤，我们创建dev，dev1，dev2用户。

## 4、grant user

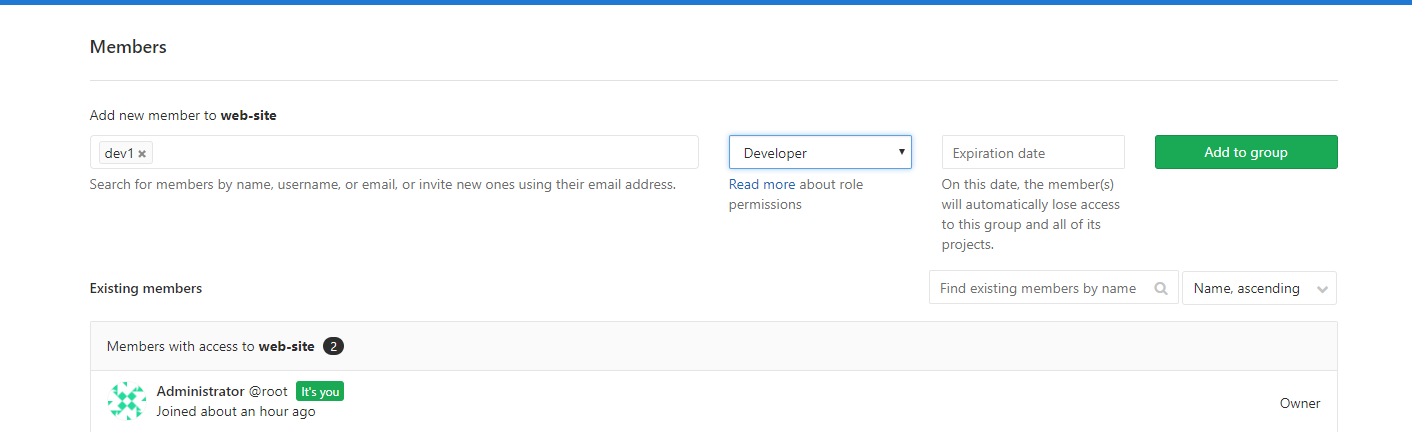
创建完user后，我们需要将user添加到组或者project上，并选择不同的role。

首先我们add user to group

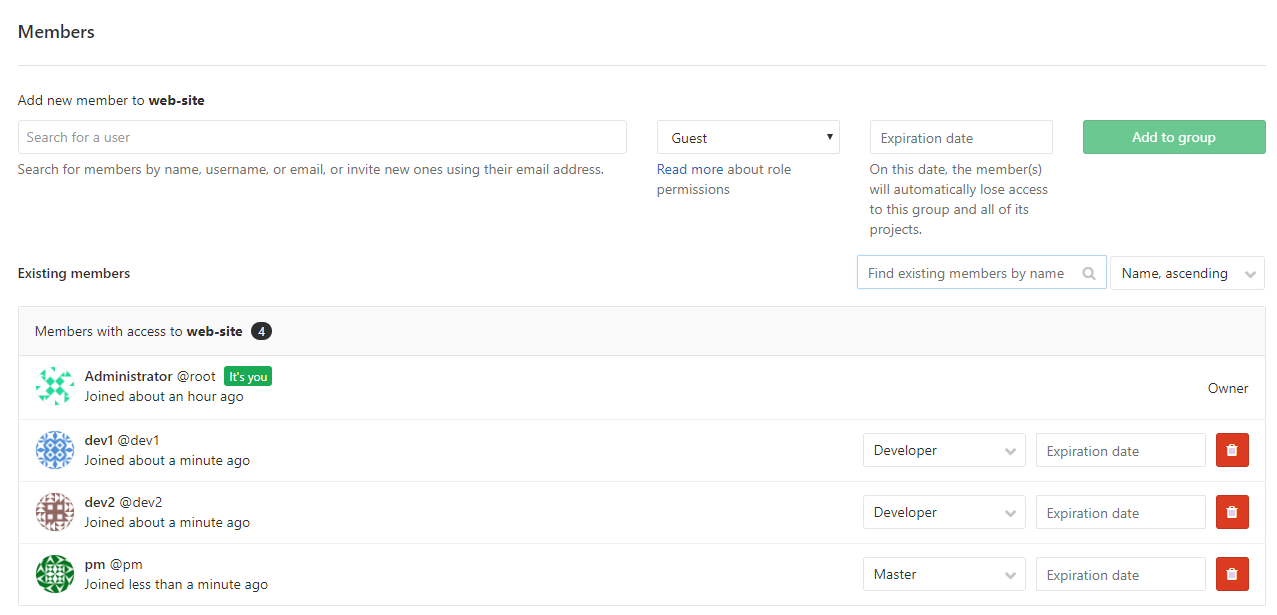




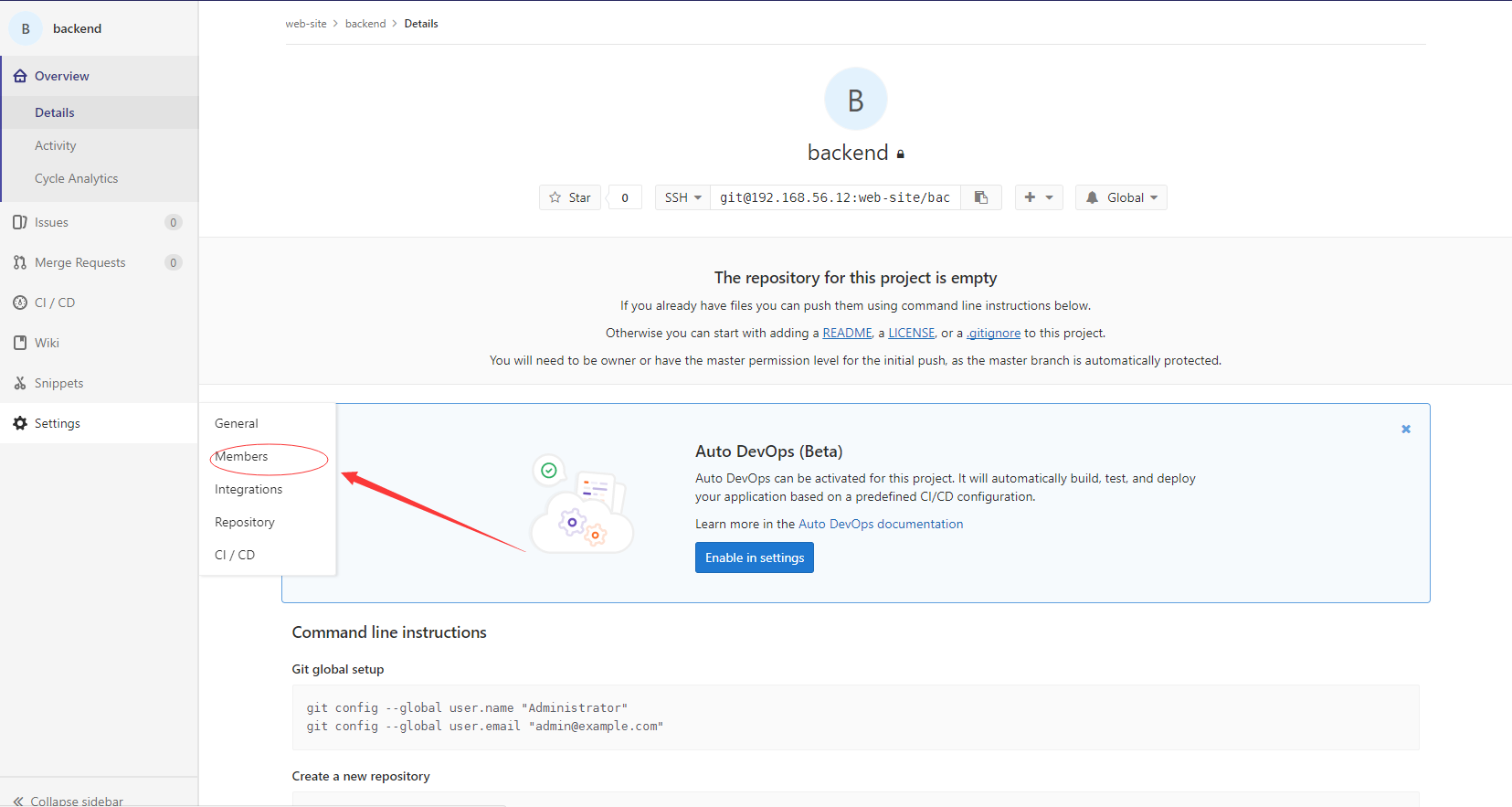
我们将选择dev1用户，角色选择Developer，过期日期不设置为永远不过期。

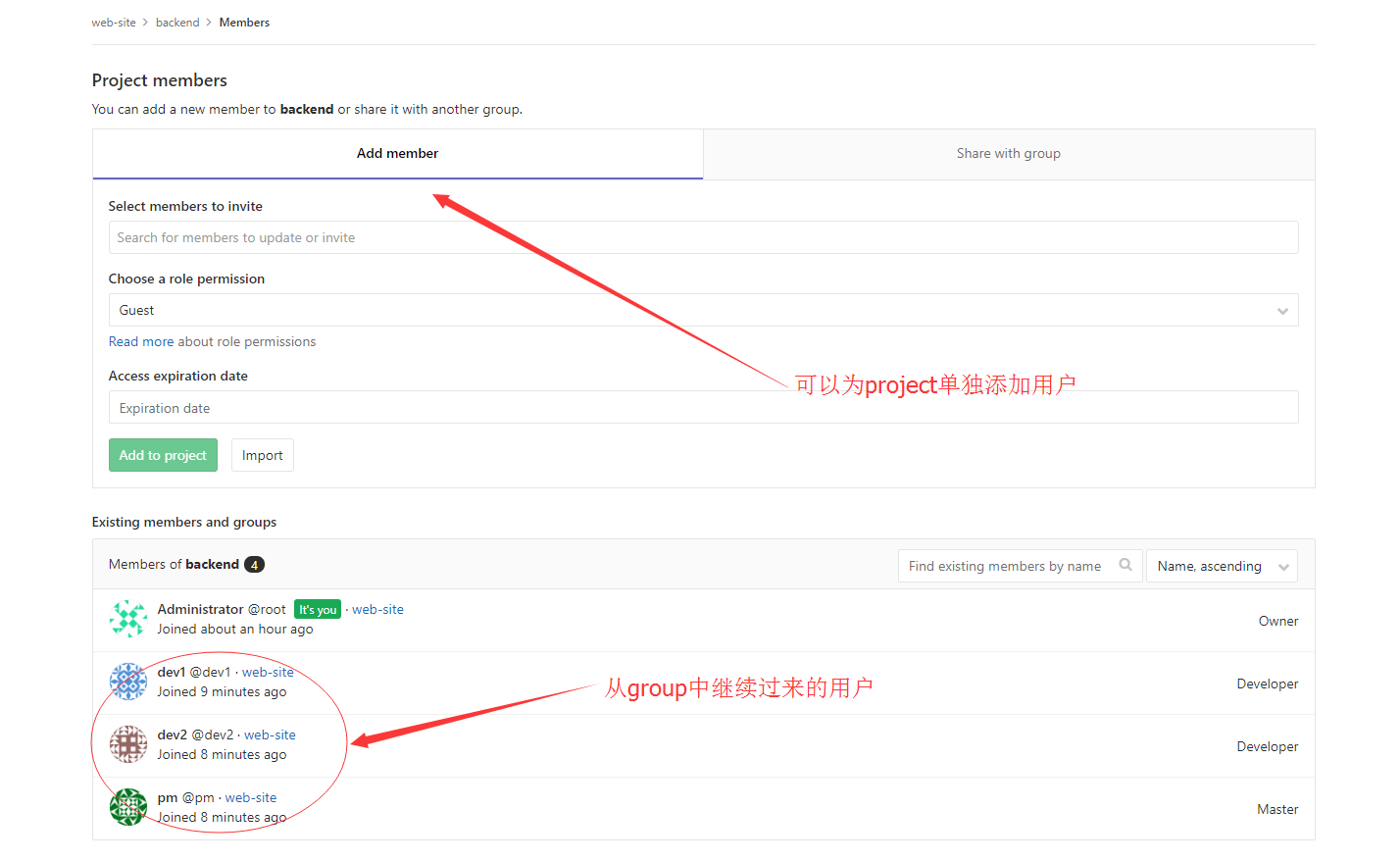


用同样的方法添加dev2,pm(角色Master)，添加到group.

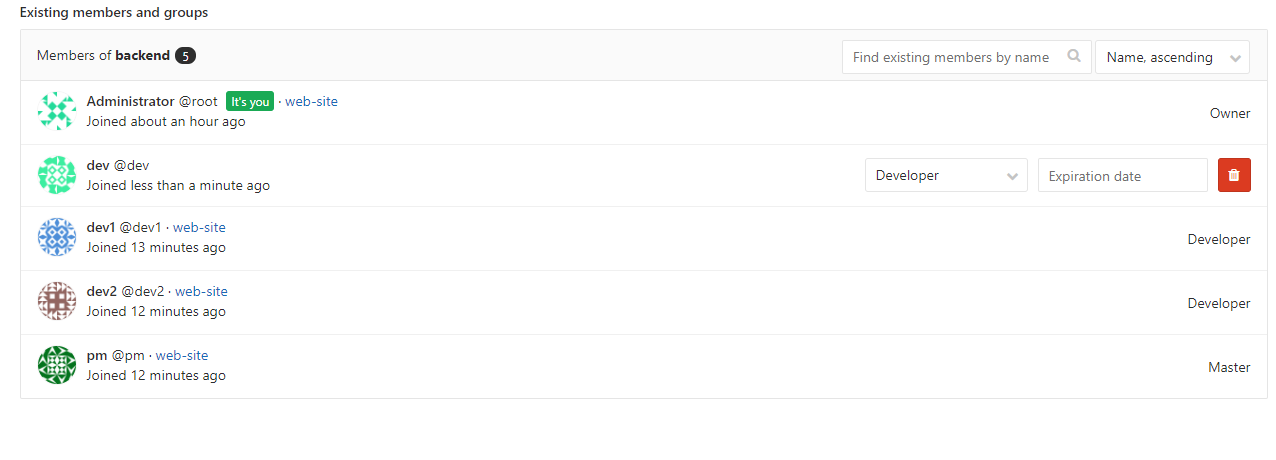


将用户添加到组后，我们发现组里的每个project下自动添加了我们刚才添加的用户。

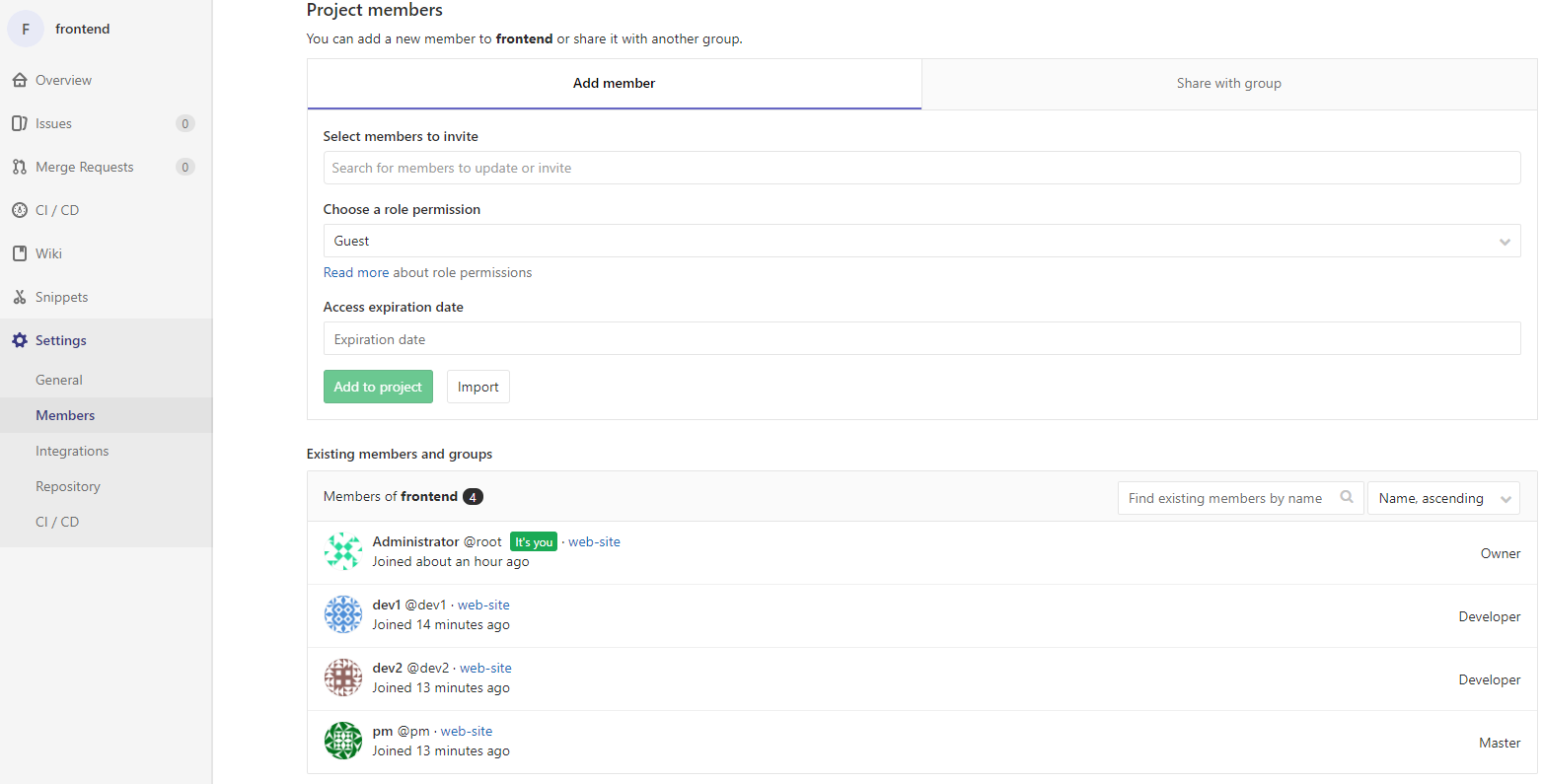




我们将dev用户添加到backend中，



我们切换到frontend project看一下：发现没有dev用户。



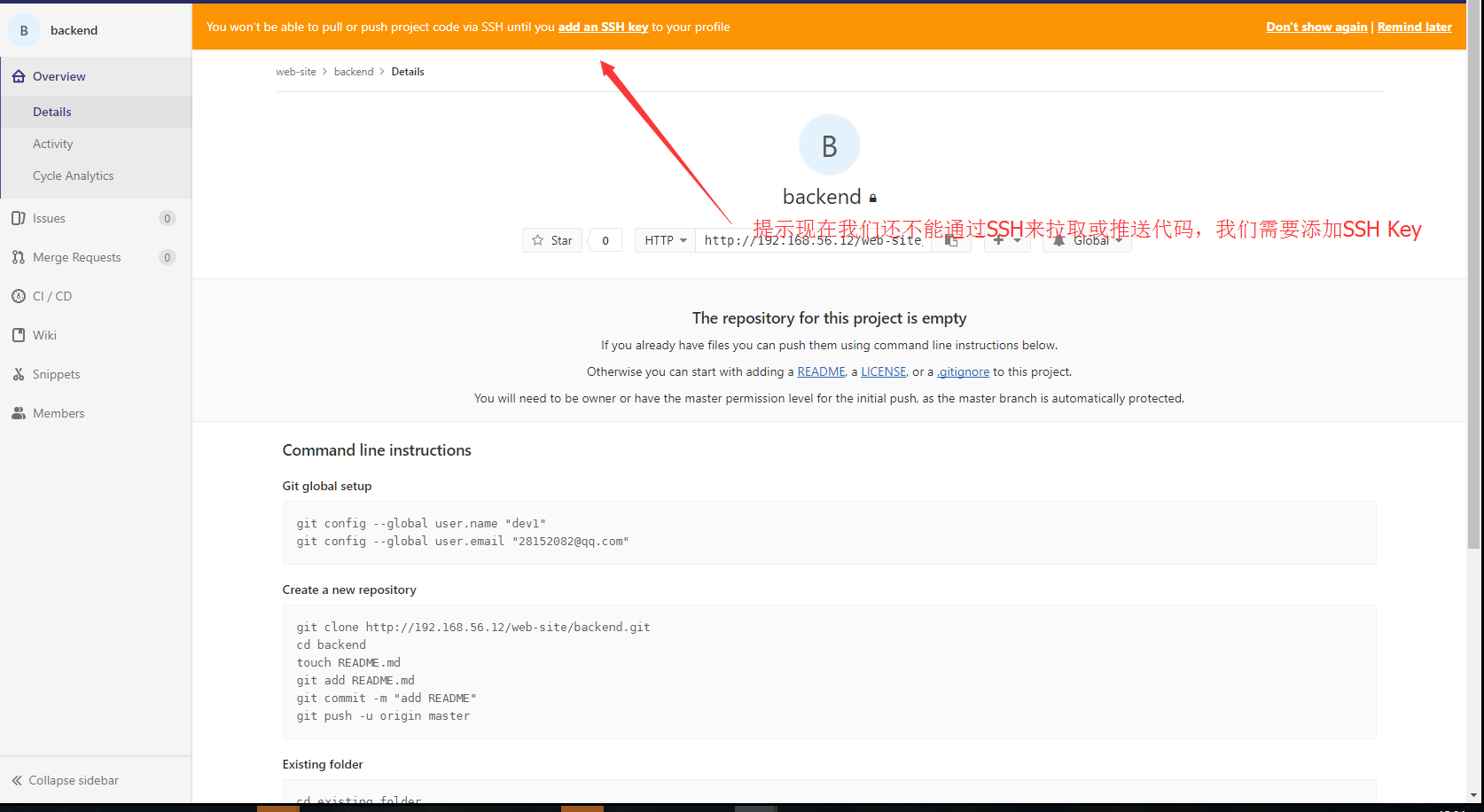
现在我们可以使用用户模拟的功能看一下每个用户的权限。

具体每一种角色的权限，可以参考如下地址：

<http://192.168.56.12/help/user/permissions> ，地址换成你的Gitlab地址即可。

## 5、添加SSH Key

我们使用dev1帐号登录到Gitlab，然后切换到一个具体的project下：



我们为dev1用户添加一个SSH Key，SSH Key可以让我以SSH的方式链接到代码仓库，然后就可以在本地和Gitlab仓库之间拉取和推送代码。SSH Key全局唯一。

我们在node1节点上生成SSH Key:

[root@node1 ~]# ssh-keygen -t rsa

Generating public/private rsa key pair.

Enter file in which to save the key (/root/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /root/.ssh/id\_rsa.

Your public key has been saved in /root/.ssh/id\_rsa.pub.

The key fingerprint is:

SHA256:eEoexwPCixJX99Gr7H3Sn29BS4RIJ4IBIbnr2fGme1M root@node1

The key's randomart image is:

+---[RSA 2048]----+

| .oo+.+o.o... |

| +.. o .o.o. . |

|. . + . . . . |

| o o o + . o |

|. . o +.S. o .|

| . . + =oE o |

| . o =... . .|

| o . =. o o ..|

| o= . o .oo.|

+----[SHA256]-----+

[root@node1 ~]# ll .ssh/

total 12

-rw------- 1 root root 1679 Dec 4 15:17 id\_rsa

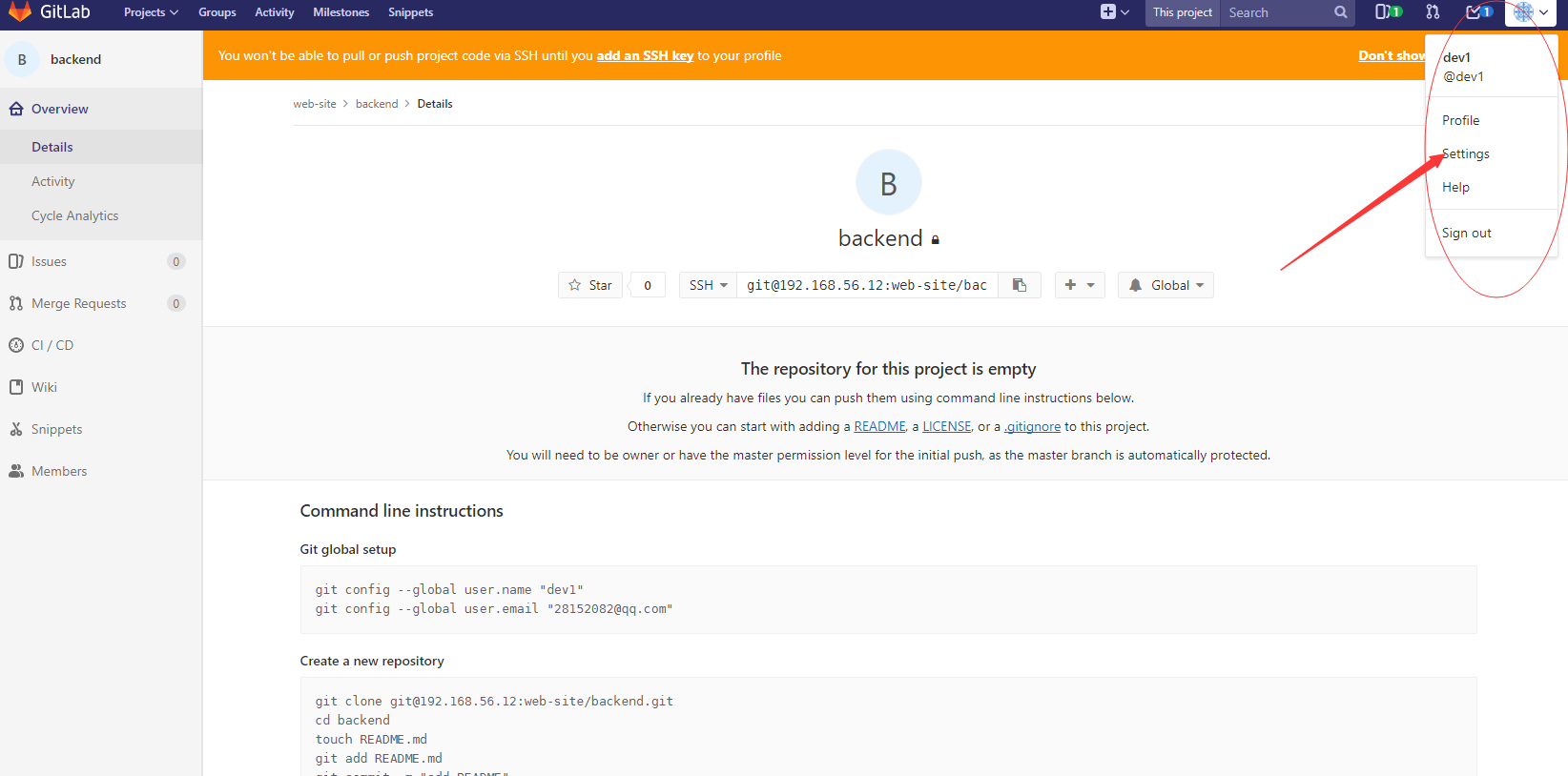
-rw-r--r-- 1 root root 392 Dec 4 15:17 id\_rsa.pub

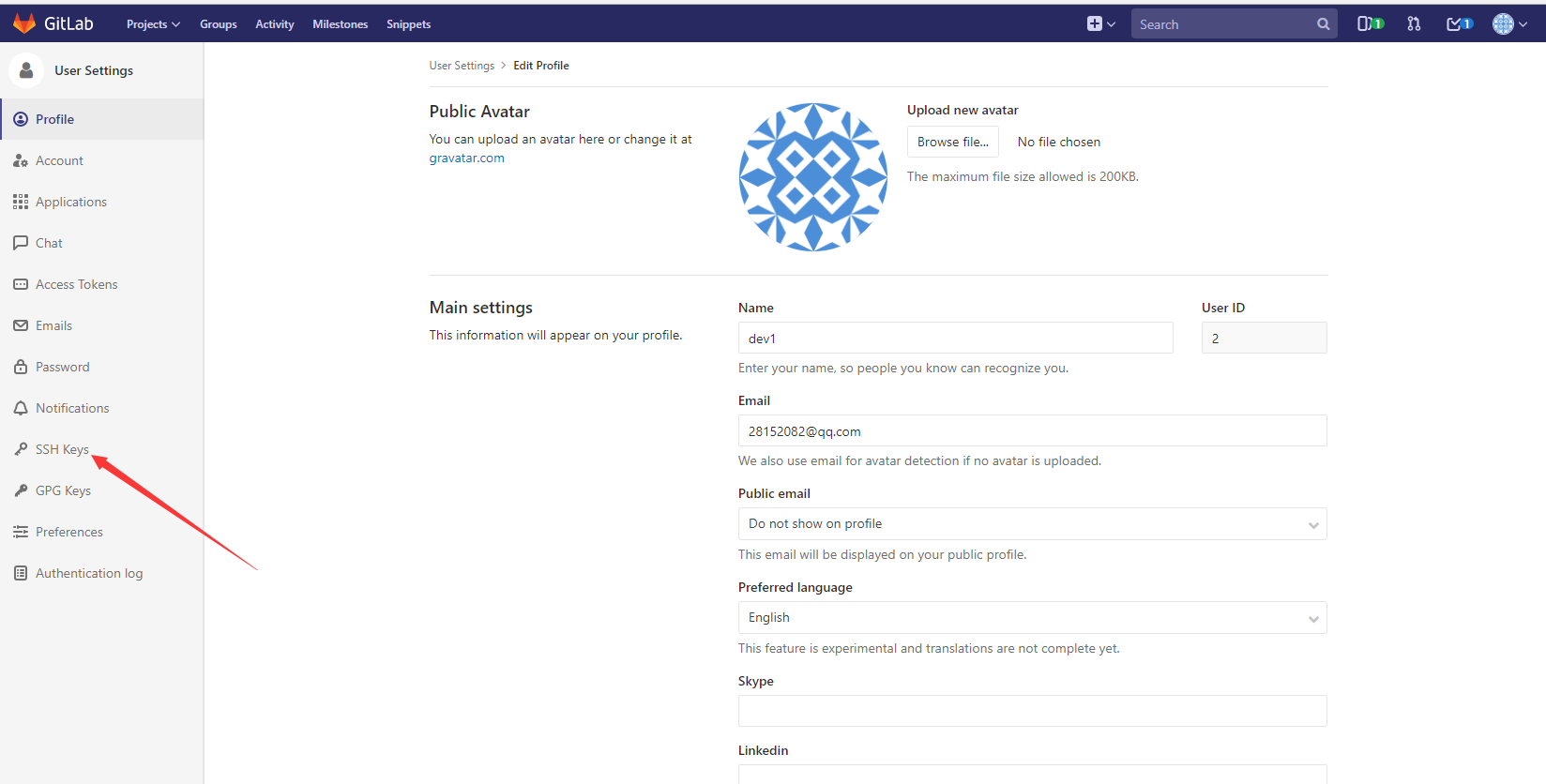
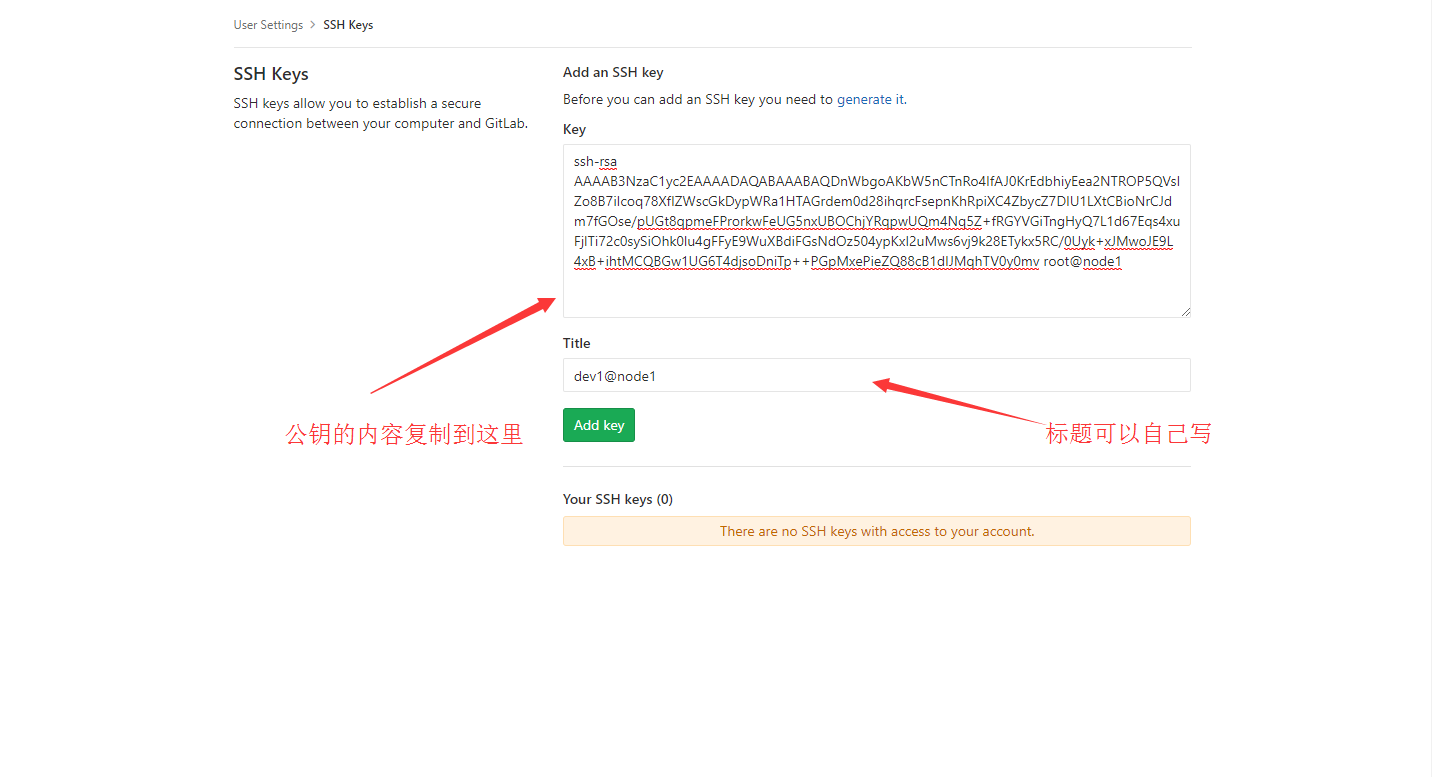
-rw-r--r-- 1 root root 803 Dec 1 13:26 known\_hosts

[root@node1 ~]# cat .ssh/id\_rsa.pub

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDnWbgoAKbW5nCTnRo4IfAJ0KrEdbhiyEea2NTROP5QVsIZo8B7iIcoq78XfIZWscGkDypWRa1HTAGrdem0d28ihqrcFsepnKhRpiXC4ZbycZ7DIU1LXtCBioNrCJdm7fGOse/pUGt8qpmeFProrkwFeUG5nxUBOChjYRqpwUQm4Nq5Z+fRGYVGiTngHyQ7L1d67Eqs4xuFjlTi72c0sySiOhk0lu4gFFyE9WuXBdiFGsNdOz504ypKxI2uMws6vj9k28ETykx5RC/0Uyk+xJMwoJE9L4xB+ihtMCQBGw1UG6T4djsoDniTp++PGpMxePieZQ88cB1dIJMqhTV0y0mv root@node1

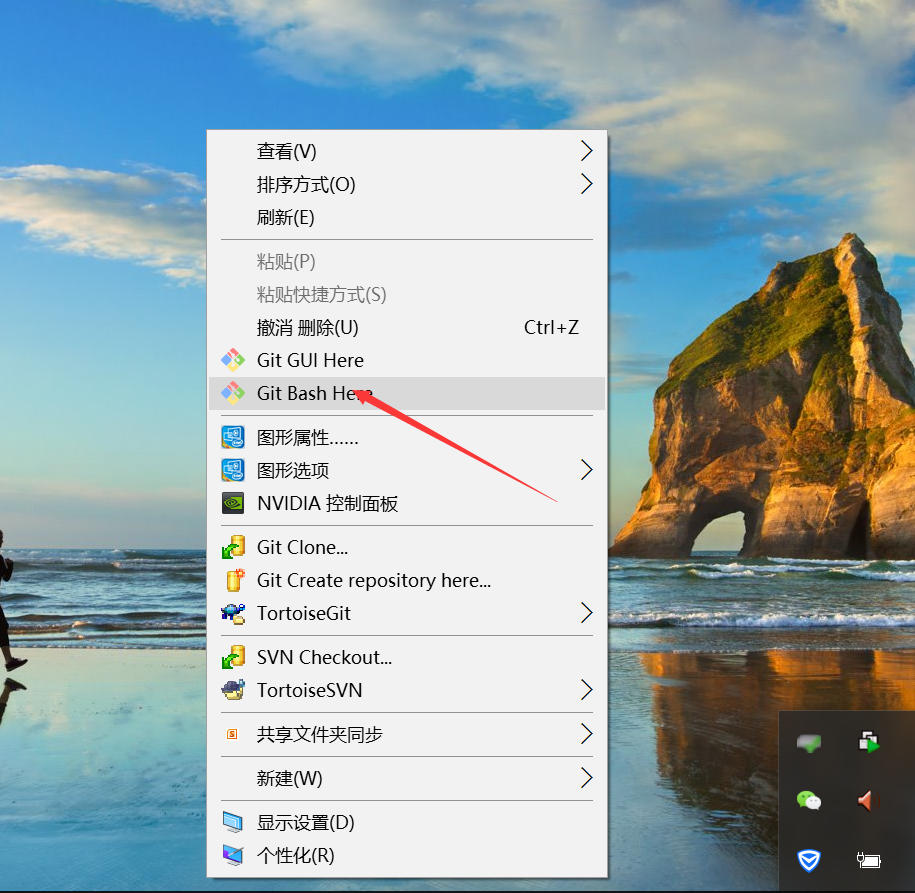
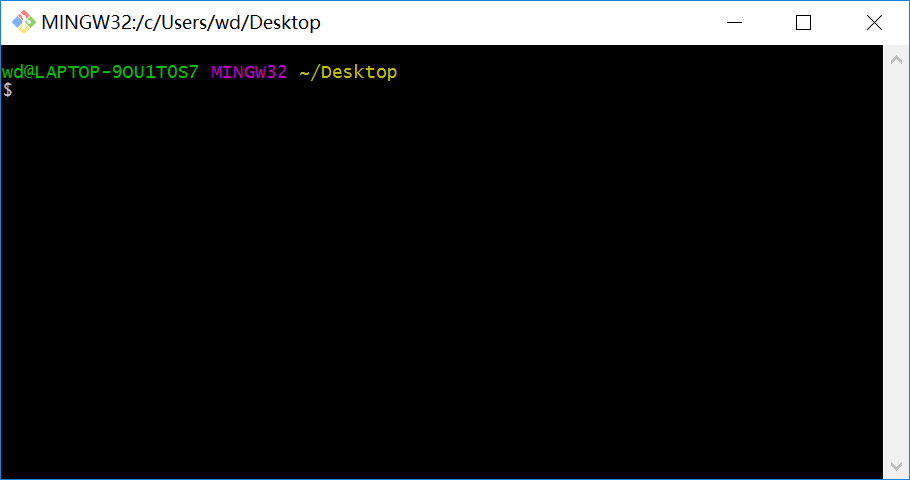
我们将公钥id\_rsa.pub的内容添加到Gitlab dev1用户的SSH Key中。



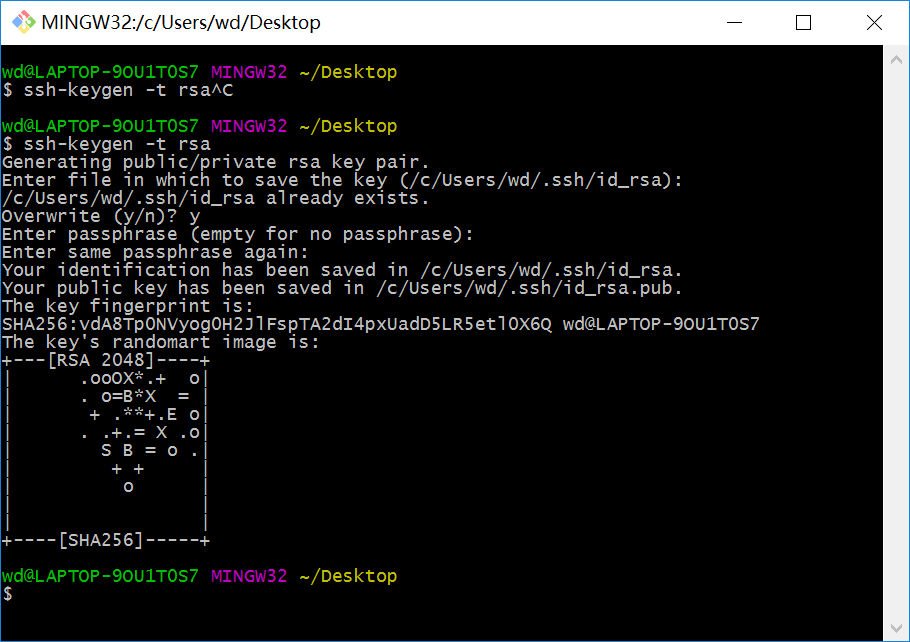
 

至此，我们已经完成打通了dev1的本机与Gitlab仓库之间的通道。

我们再将我们的windows的SSH Key添加到Gitlab，首先我们要安装Git GUI（参考gitlab安装文档）：

在窗口中重复执行我们在linux中创建SSH Key的过程

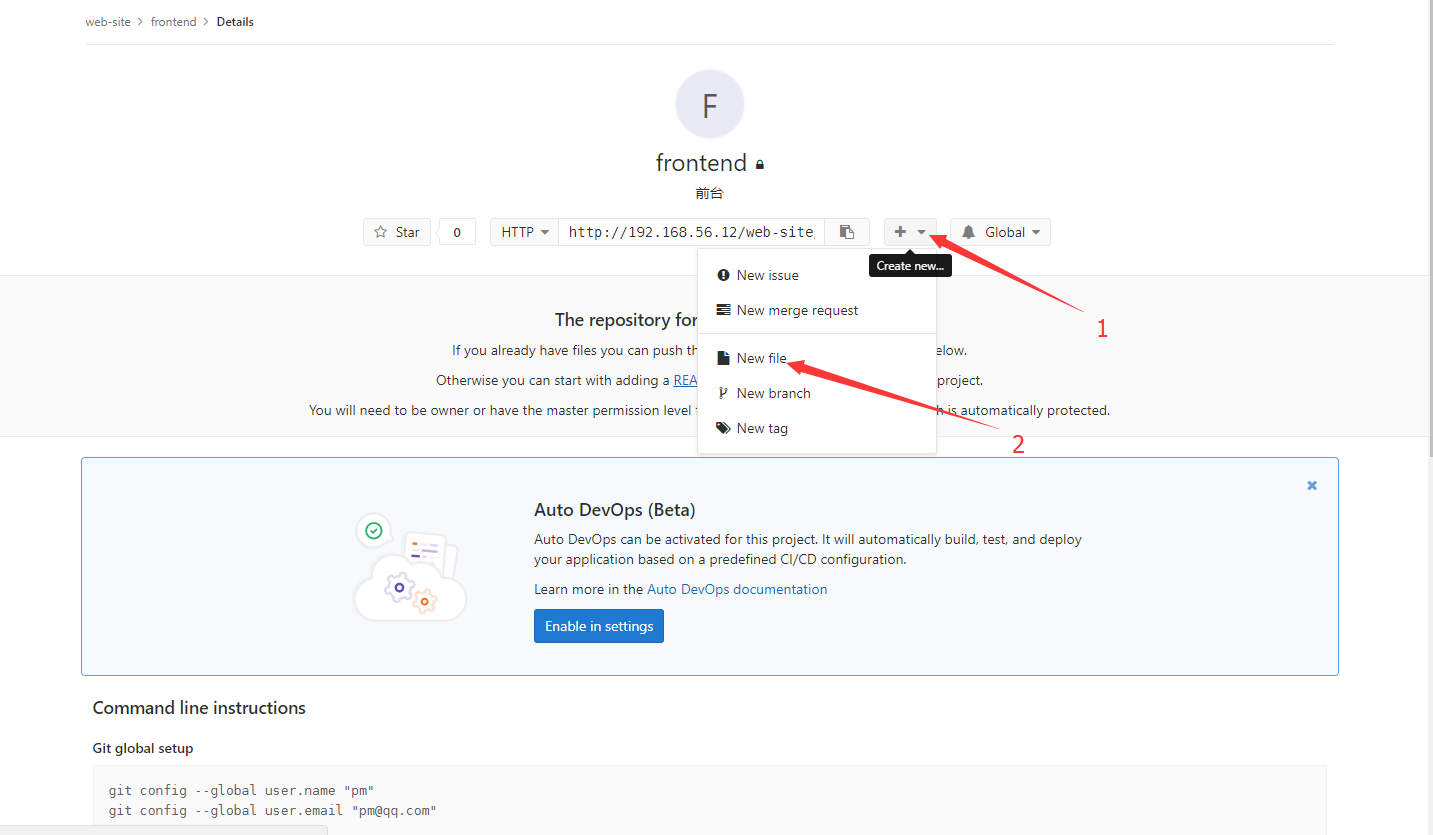
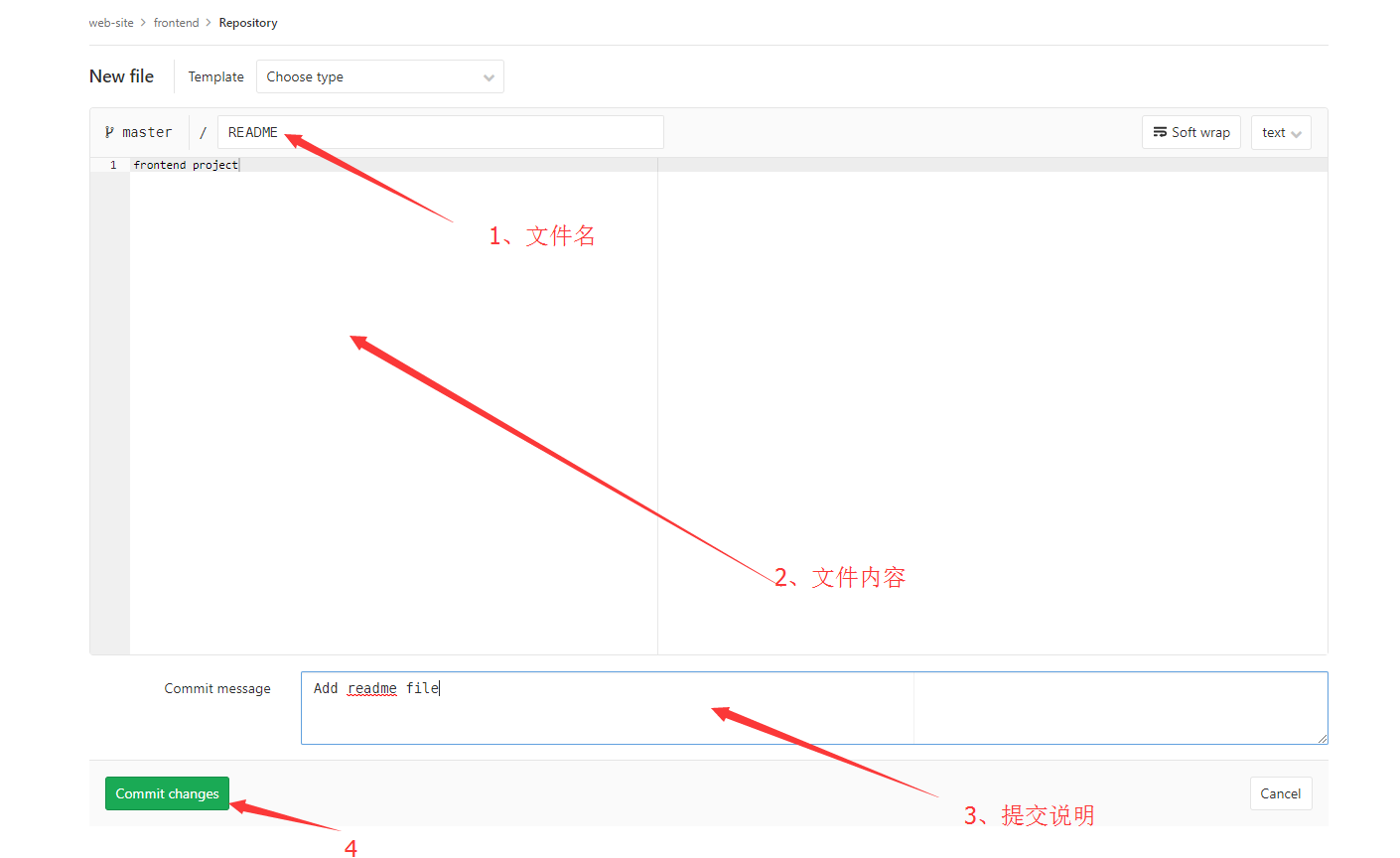


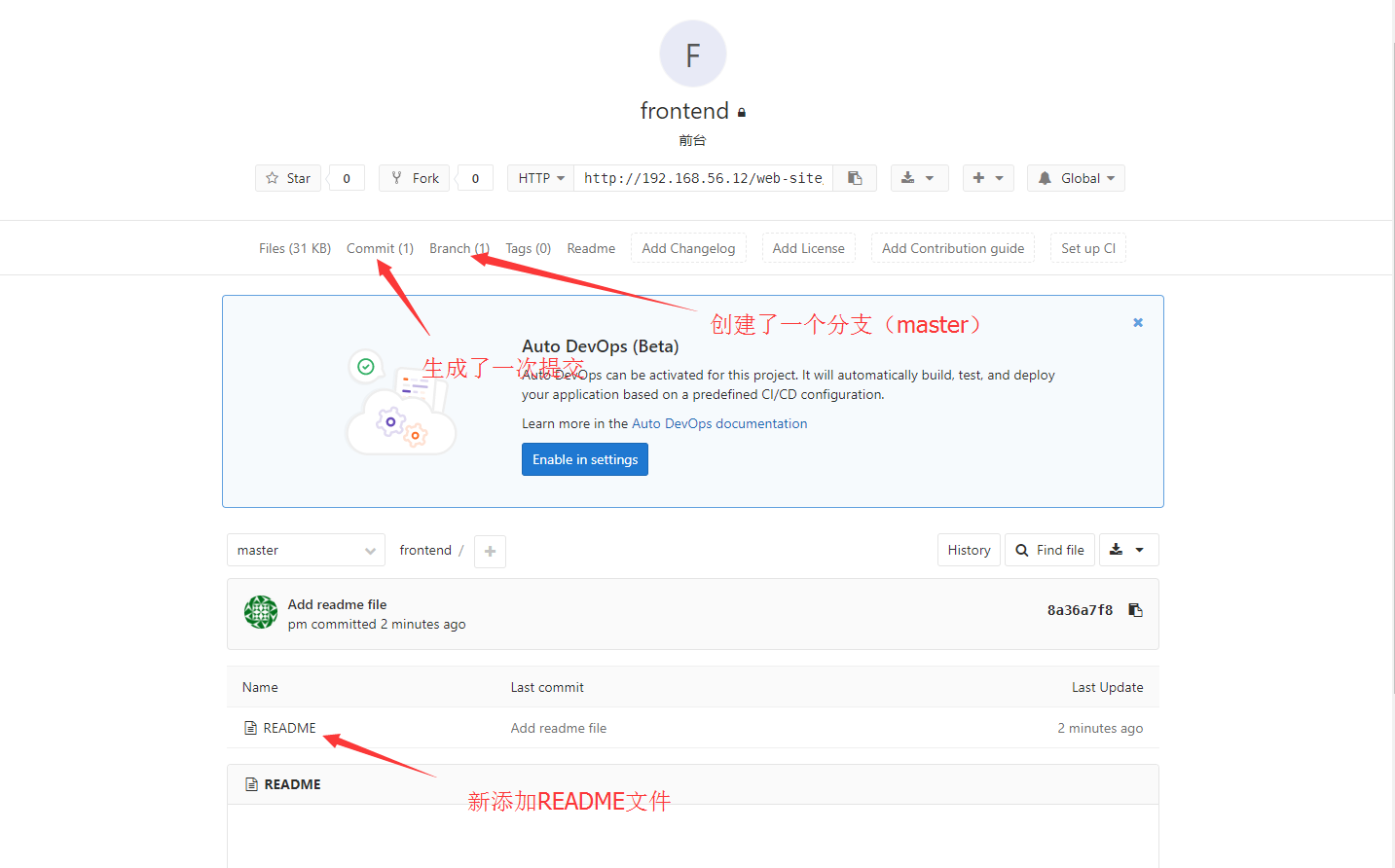
然后根据提示找到id\_rsa.pub,将该文件中的内容添加到Gitlab中dev2的SSH Key。

由于SSH Key全局唯一，所以我只要任何一个用户中添加都可以。

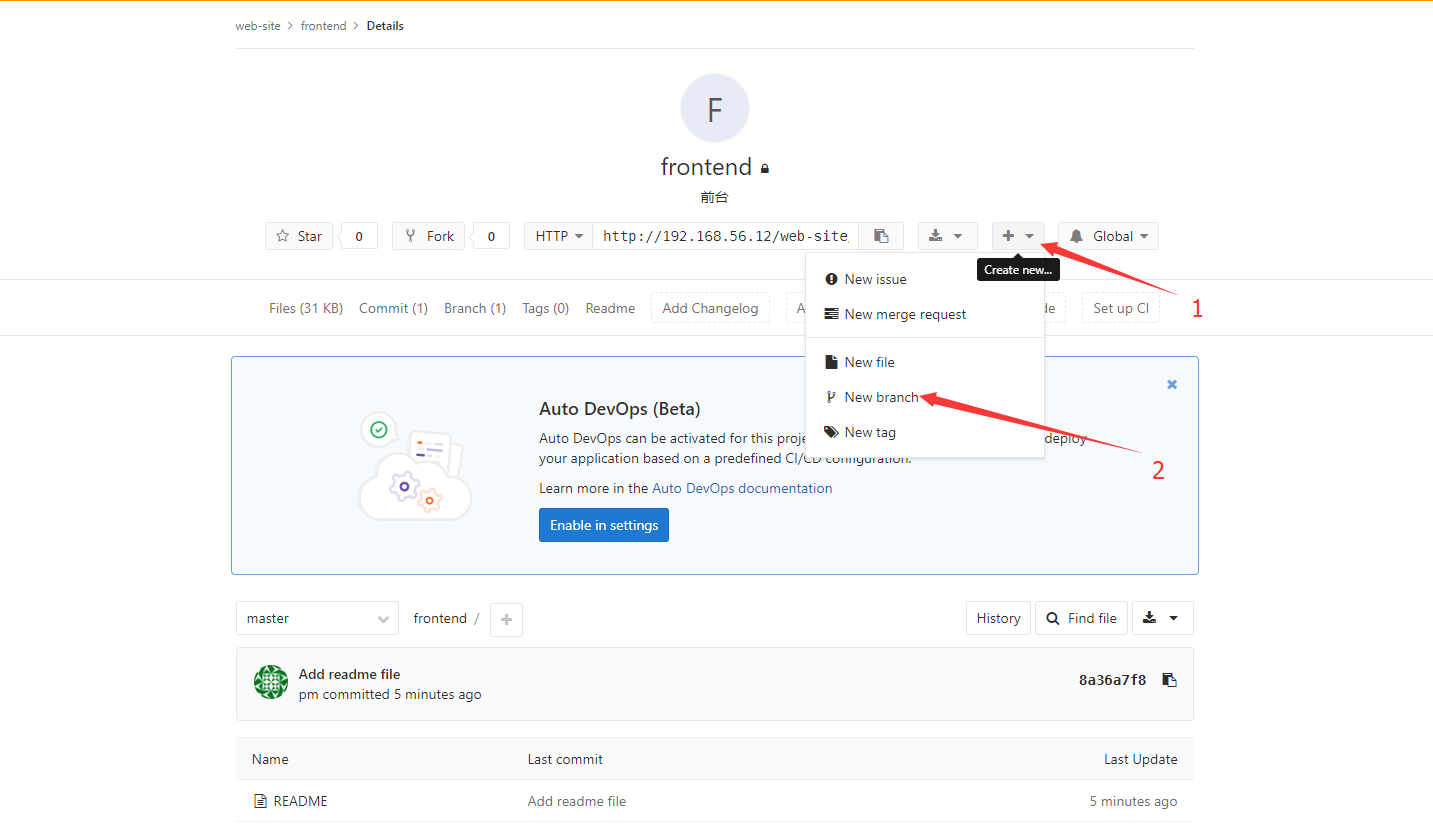
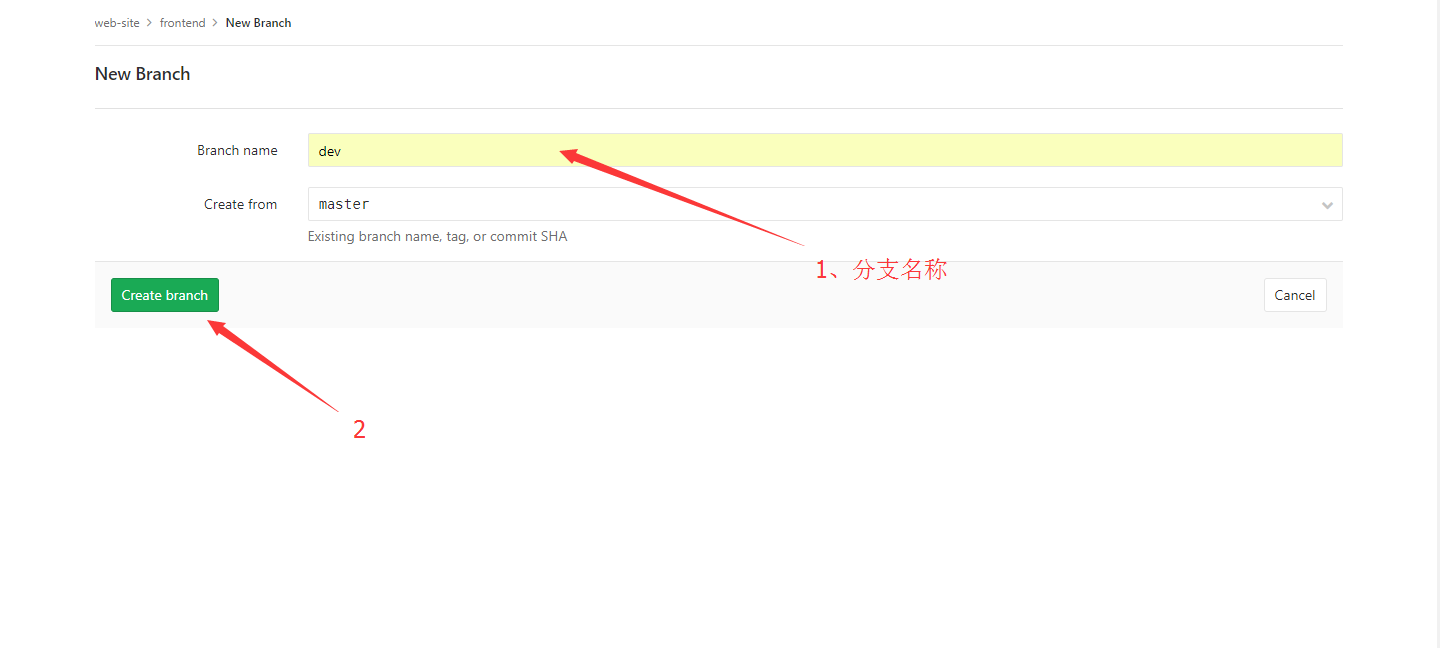
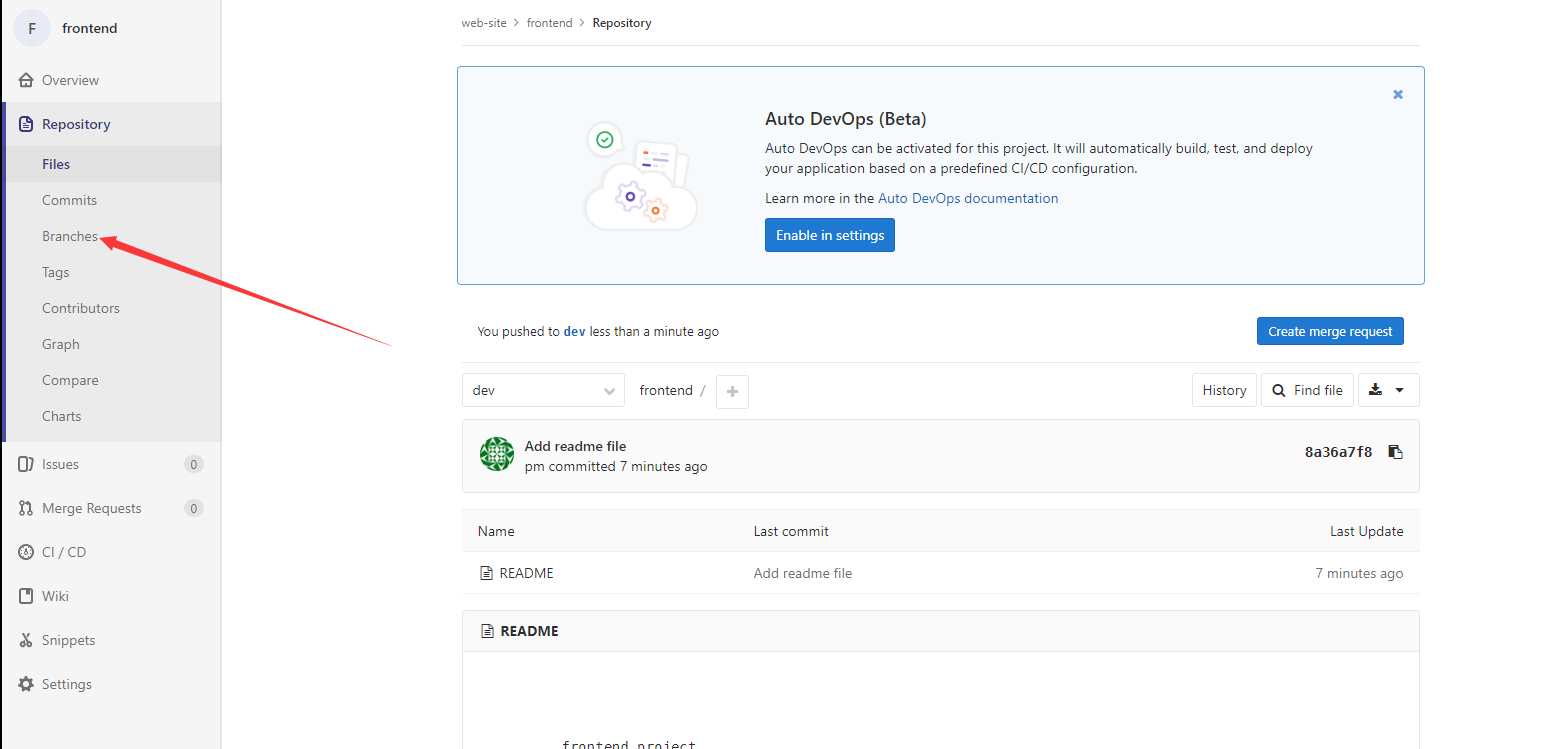
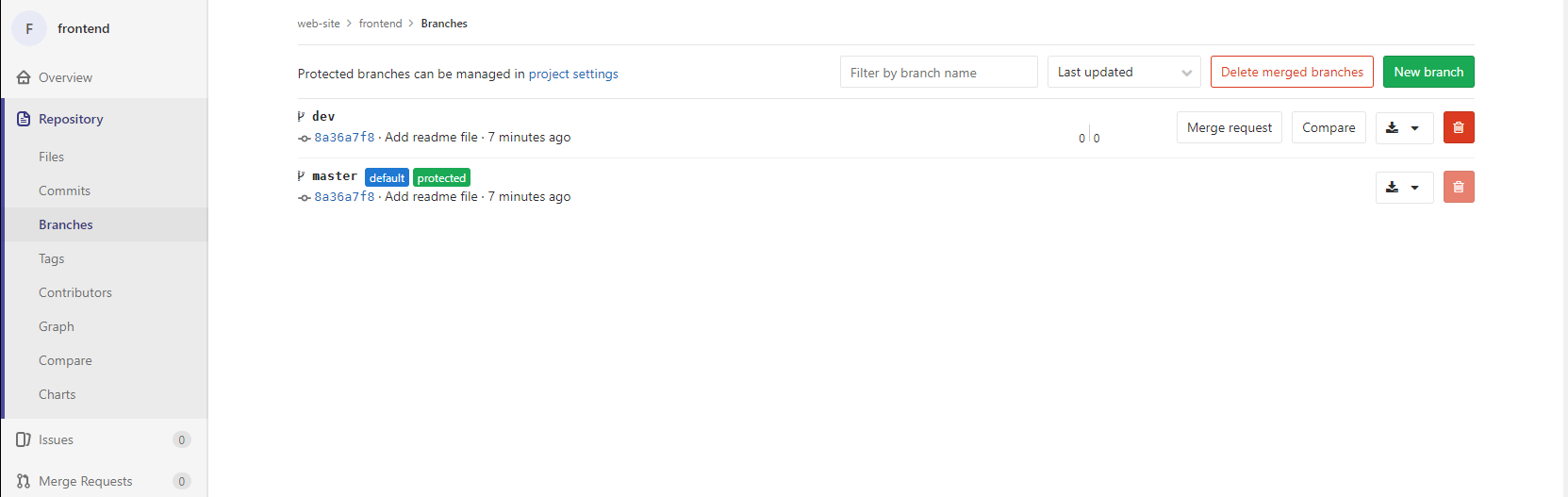
## 6、初始化GitLab仓库

刚才我们创建的前台和后台两个project现在还是空的，本次教程我们只使用其中的frontend，现在我们首先对这两个仓库进行初始化，并创建两个分支master，dev。我们使用pm用户对frontend进行初始化操作。



接下来我们自动创建dev分支：

我们可以mster分支是默认的分支，而且是受保护的。

在实际的开发过程中，master分支一般用来版本发布，dev分支用于存放开发代码。

## 7、使用Gitlab远程仓库

前面我们已经创建了frontend的远程仓库，现在我们分别在linux下和windows下连接远程仓库，实现代码的摘取与推送。

linux:

使用git clone 命令克隆远程frontend仓库到本地

[root@node1 ~]# git clone git@192.168.56.12:web-site/frontend.git

Cloning into 'frontend'...

The authenticity of host '192.168.56.12 (192.168.56.12)' can't be established.

ECDSA key fingerprint is SHA256:SjPoetHYvGBI08VxTdzYOys+QpjR5vLNbU9Obs2Lx9Q.

ECDSA key fingerprint is MD5:39:3e:52:a1:45:9b:3e:23:72:e6:0d:0e:76:00:17:55.

Are you sure you want to continue connecting (yes/no)? yes /\*第一次连接时会出现该提示

Warning: Permanently added '192.168.56.12' (ECDSA) to the list of known hosts.

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0)

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

Checking connectivity... done.

[root@node1 ~]# ll

total 4

-rw-------. 1 root root 1374 Nov 27 00:14 anaconda-ks.cfg

drwxr-xr-x 3 root root 94 Nov 30 22:28 demo

drwxr-xr-x 3 root root 32 Dec 4 21:02 frontend

drwxr-xr-x 3 root root 97 Dec 2 19:48 git-test

drwxr-xr-x 4 root root 49 Nov 30 15:53 web-site

[root@node1 ~]# cd frontend/

[root@node1 frontend]# ll

total 4

-rw-r--r-- 1 root root 16 Dec 4 21:02 README

[root@node1 frontend]# git branch

\* master

[root@node1 frontend]# git remote -v

origin git@192.168.56.12:web-site/frontend.git (fetch)

origin git@192.168.56.12:web-site/frontend.git (push)

从远程仓库拉下dev分支到本地

[root@node1 frontend]# git pull origin dev

From 192.168.56.12:web-site/frontend

\* branch dev -> FETCH\_HEAD

Already up-to-date.

切换到dev分支

[root@node1 frontend]# git checkout dev

Branch dev set up to track remote branch dev from origin.

Switched to a new branch 'dev'

[root@node1 frontend]# git branch

\* dev

master

在dev分支下添加linux.txt文件，然后commit。

[root@node1 frontend]# echo "this is linux agnet" >> linux.txt

You have new mail in /var/spool/mail/root

[root@node1 frontend]# git add .

[root@node1 frontend]# git commit -m "add linux.txt file"

[dev cdd58d8] add linux.txt file

1 file changed, 1 insertion(+)

create mode 100644 linux.txt

推送本地dev分支到远程仓库的dev分支

[root@node1 frontend]# git push origin dev

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 291 bytes | 0 bytes/s, done.

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

remote:

remote: To create a merge request for dev, visit:

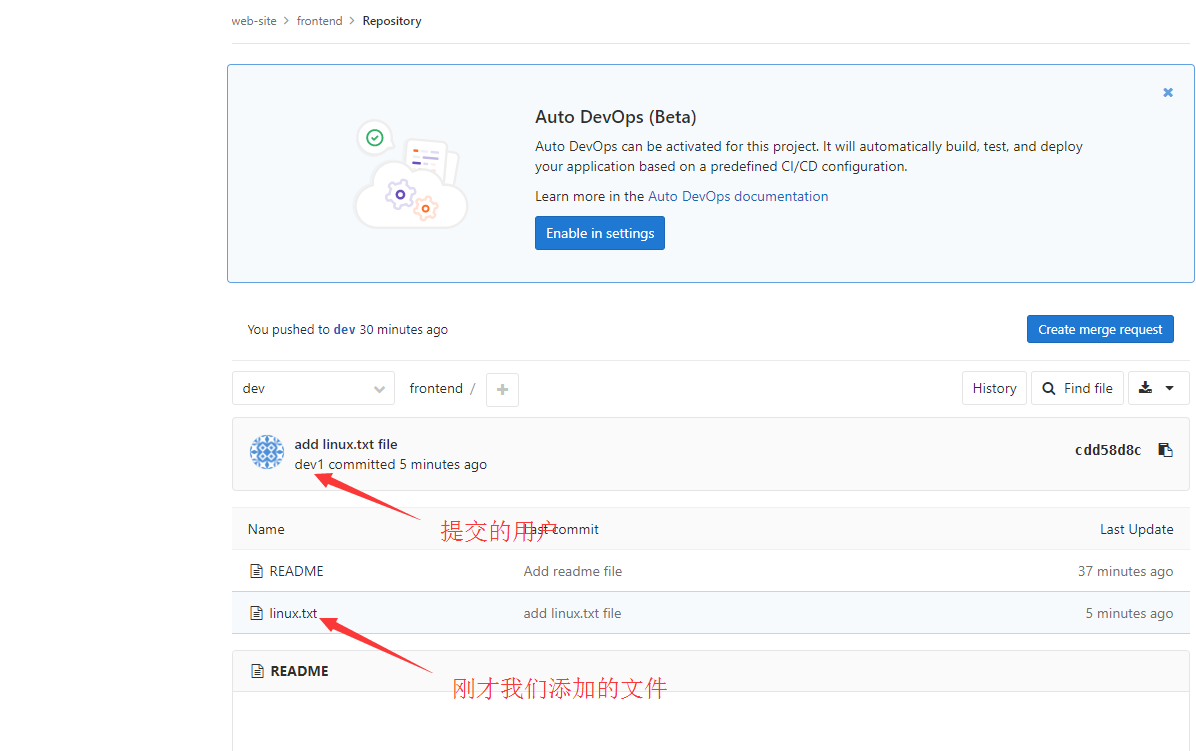
remote: http://192.168.56.12/web-site/frontend/merge\_requests/new?merge\_request%5Bsource\_branch%5D=dev

remote:

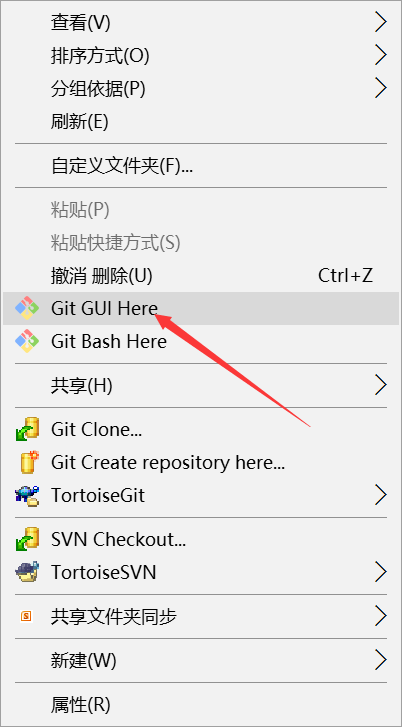
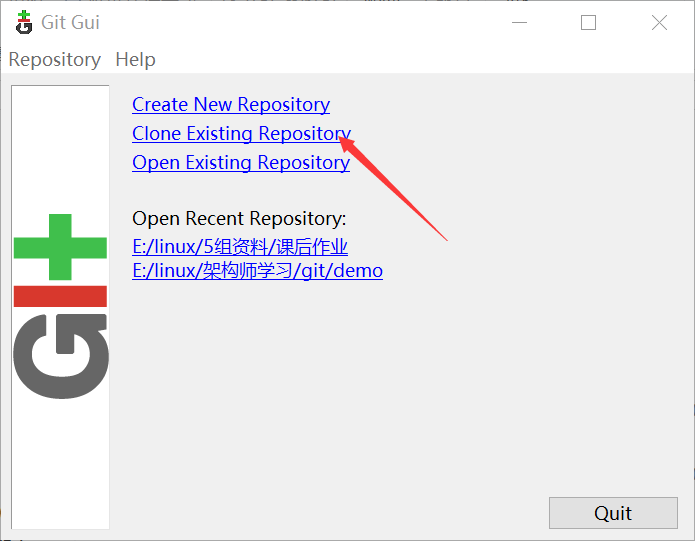
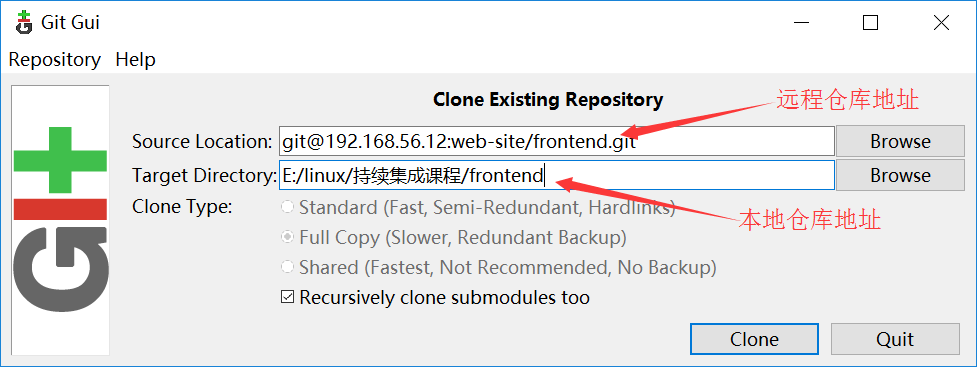
To 192.168.56.12:web-site/frontend.git

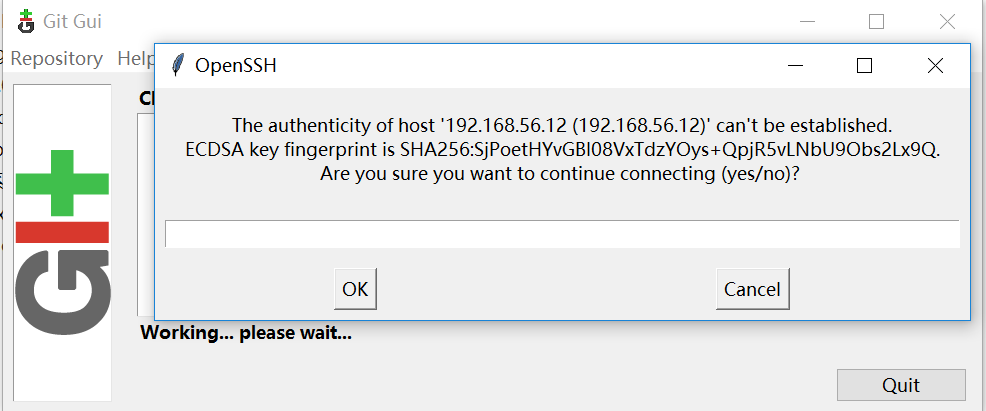
8a36a7f..cdd58d8 dev -> dev

然后我们在Gitlab上查看：

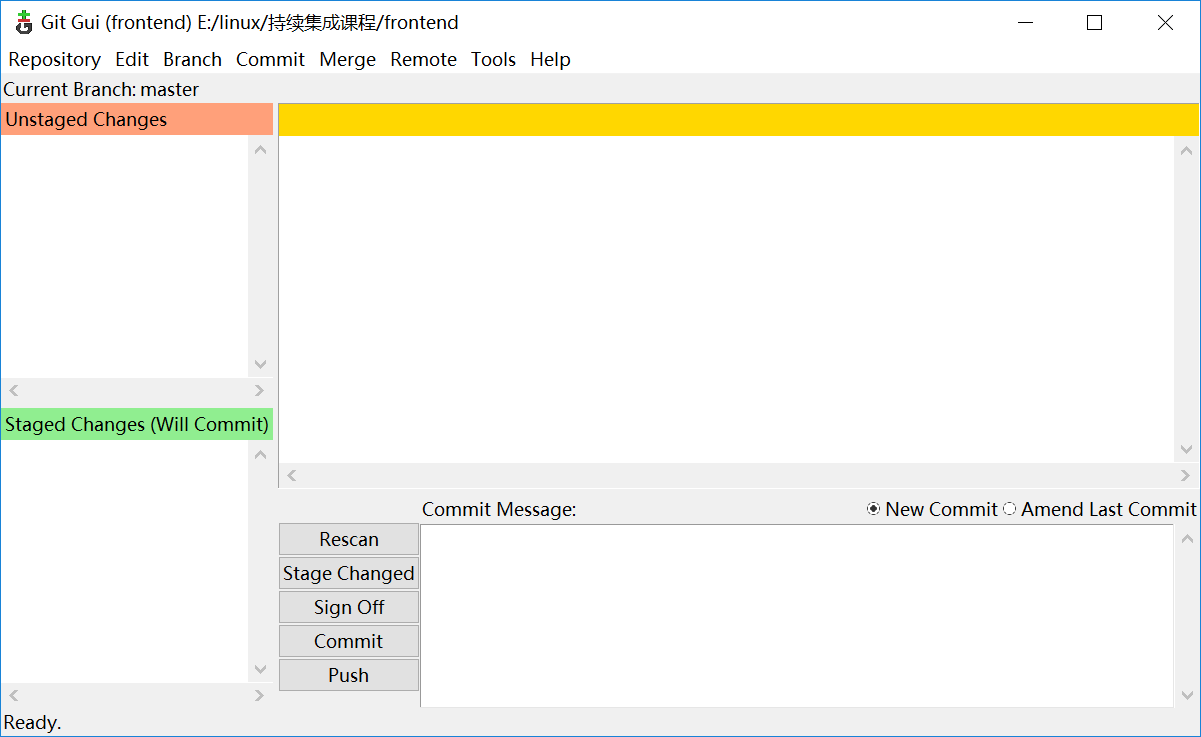


windows:

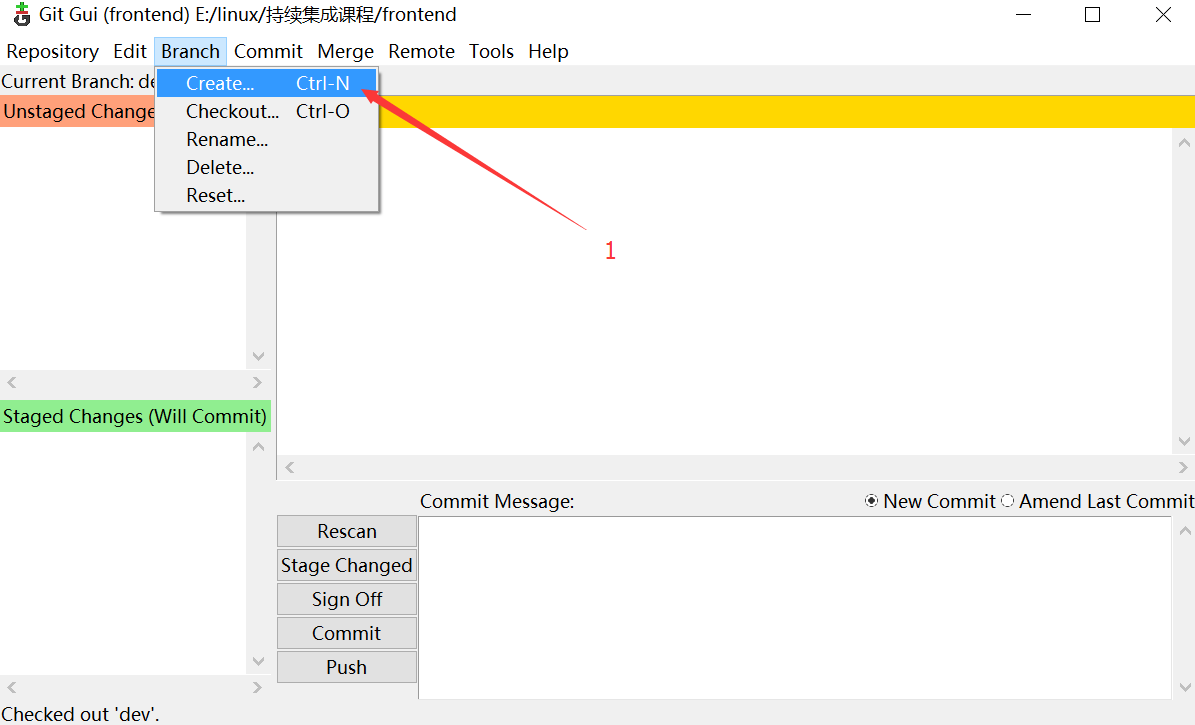
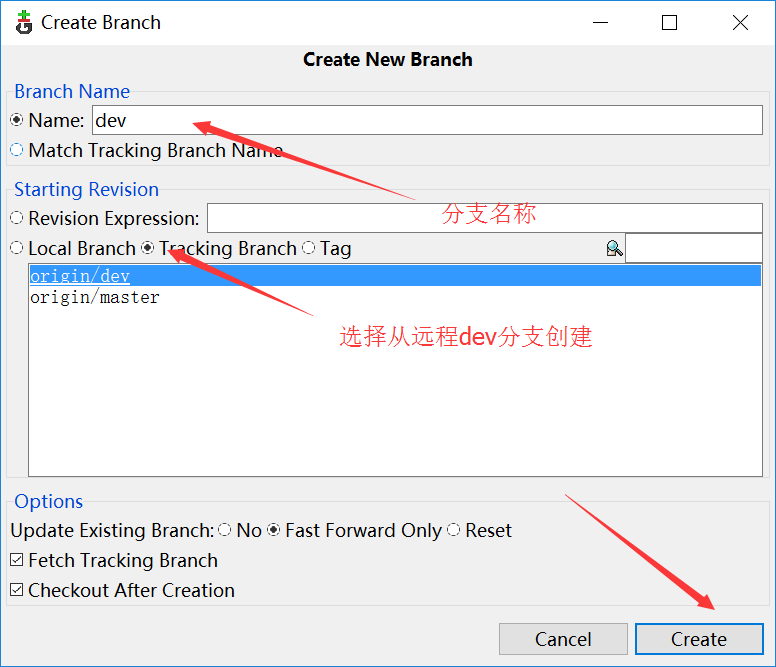
  

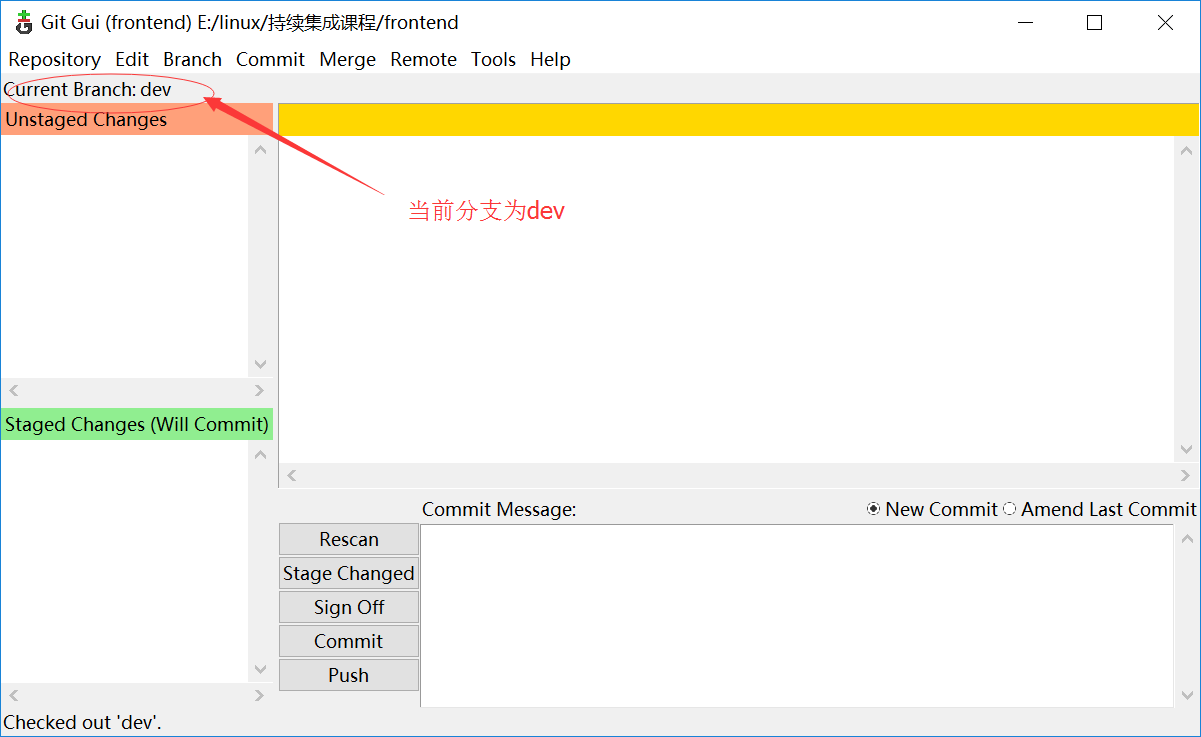


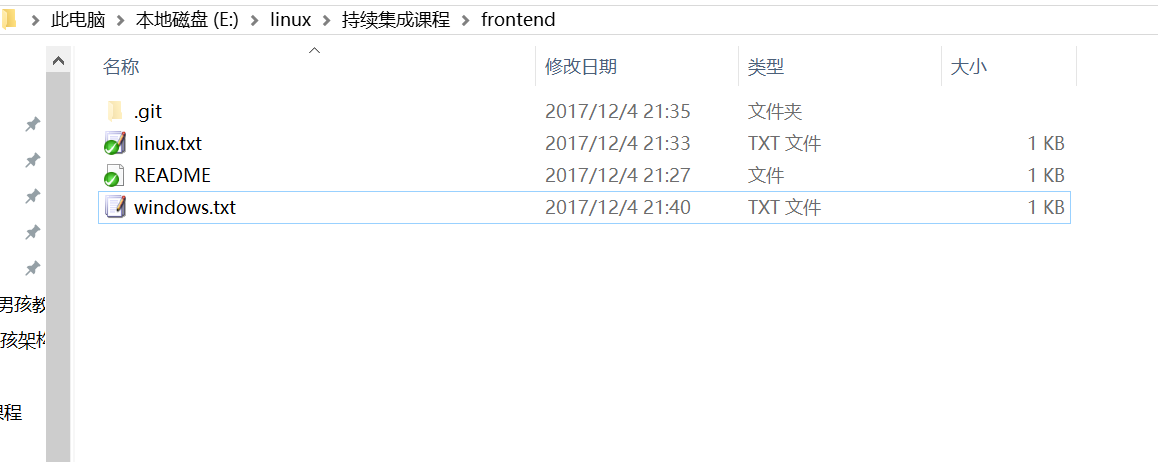
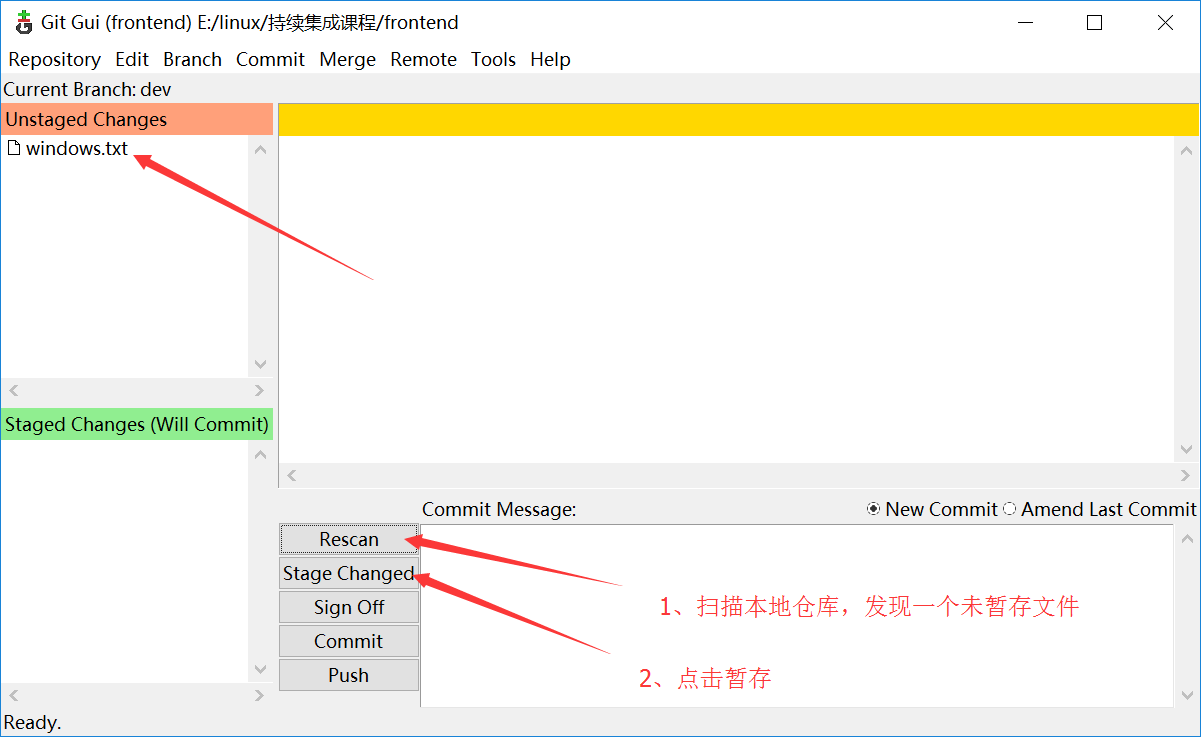
第一次连接会出现此对话框，请输入yes。

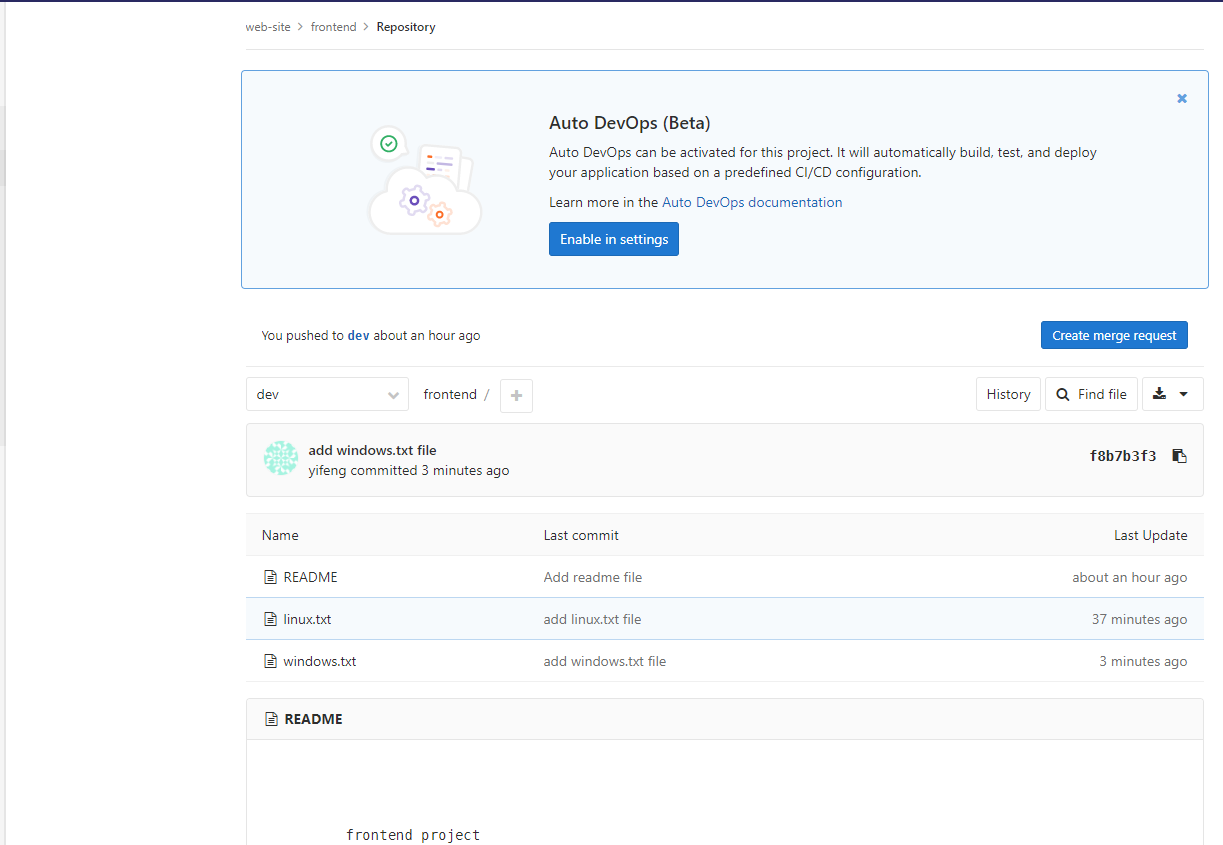


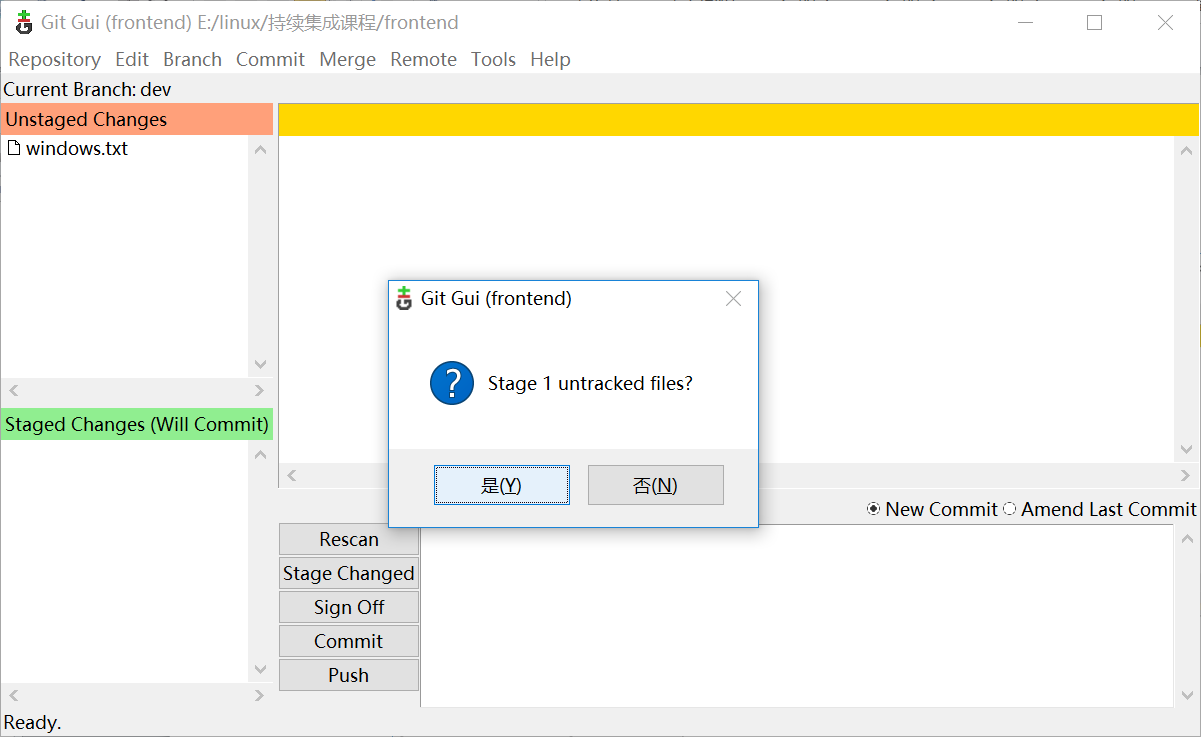
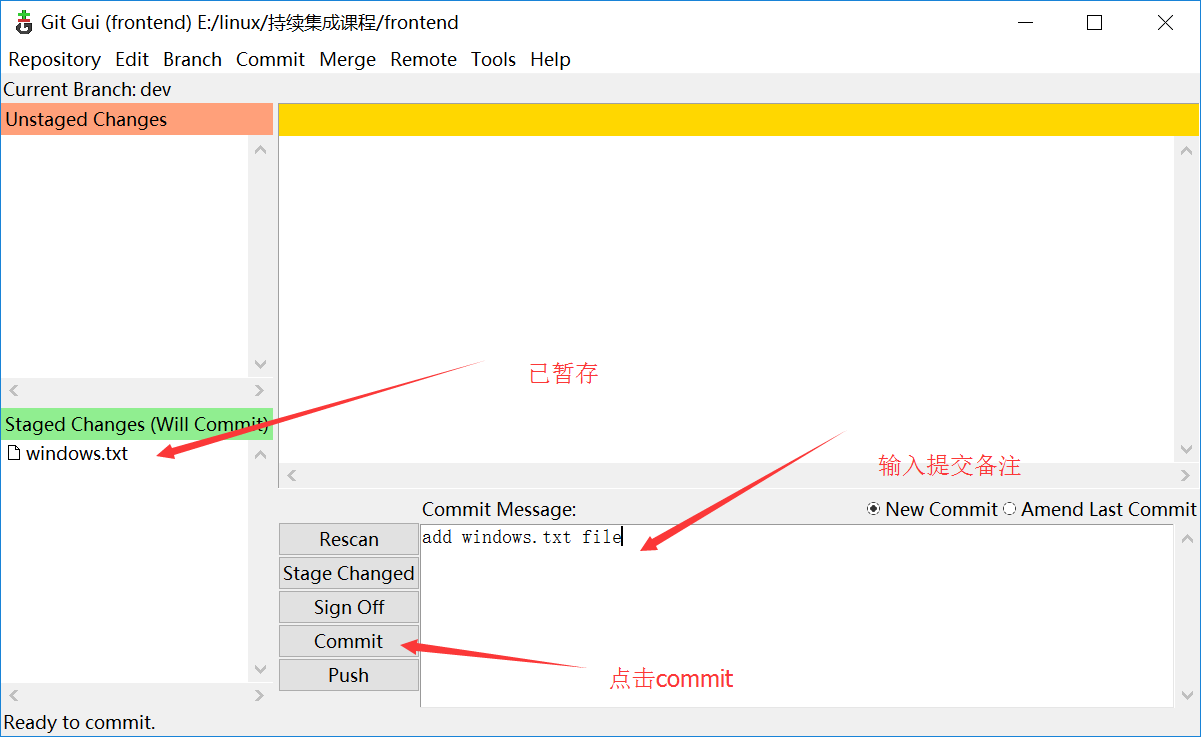
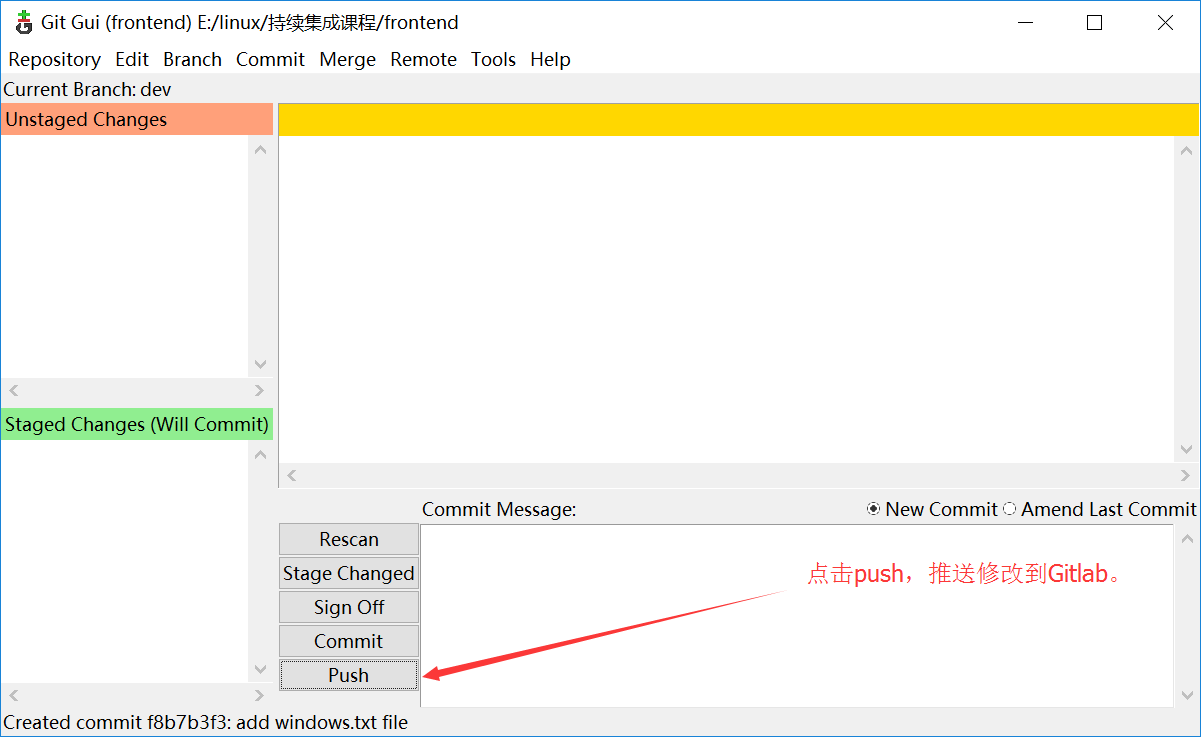
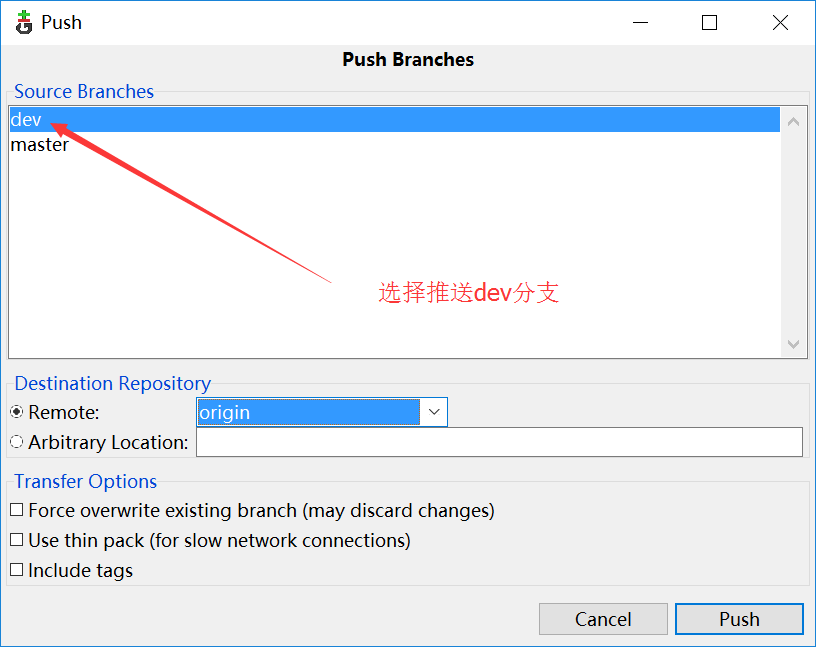
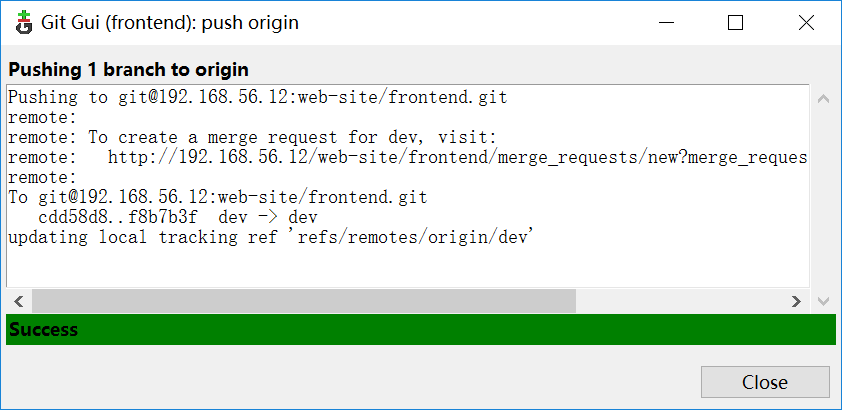
拉取远程dev分支，然后切换

我们在frontend文件夹里添加windows文件



上面的两个过程我们都是从本地的dev推送到Gitlab仓库的dev分支，那我是不是可以直接在本地合并到master分支，然后推送master分支到Gitlab呢？这个问题留给大家课后实际操作一下。

实验结果：

[root@node1 frontend]# git checkout master

Switched to branch 'master'

[root@node1 frontend]# git merge dev

Updating 8a36a7f..cdd58d8

Fast-forward

linux.txt | 1 +

1 file changed, 1 insertion(+)

create mode 100644 linux.txt

[root@node1 frontend]# ll

total 8

-rw-r--r-- 1 root root 20 Dec 4 21:50 linux.txt

-rw-r--r-- 1 root root 16 Dec 4 21:02 README

[root@node1 frontend]# git push origin master

Counting objects: 3, done.

Delta compression using up to 2 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 291 bytes | 0 bytes/s, done.

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

**remote: GitLab: You are not allowed to push code to protected branches on this project.**

To 192.168.56.12:web-site/frontend.git

! [remote rejected] master -> master (pre-receive hook declined)

error: failed to push some refs to 'git@192.168.56.12:web-site/frontend.git'

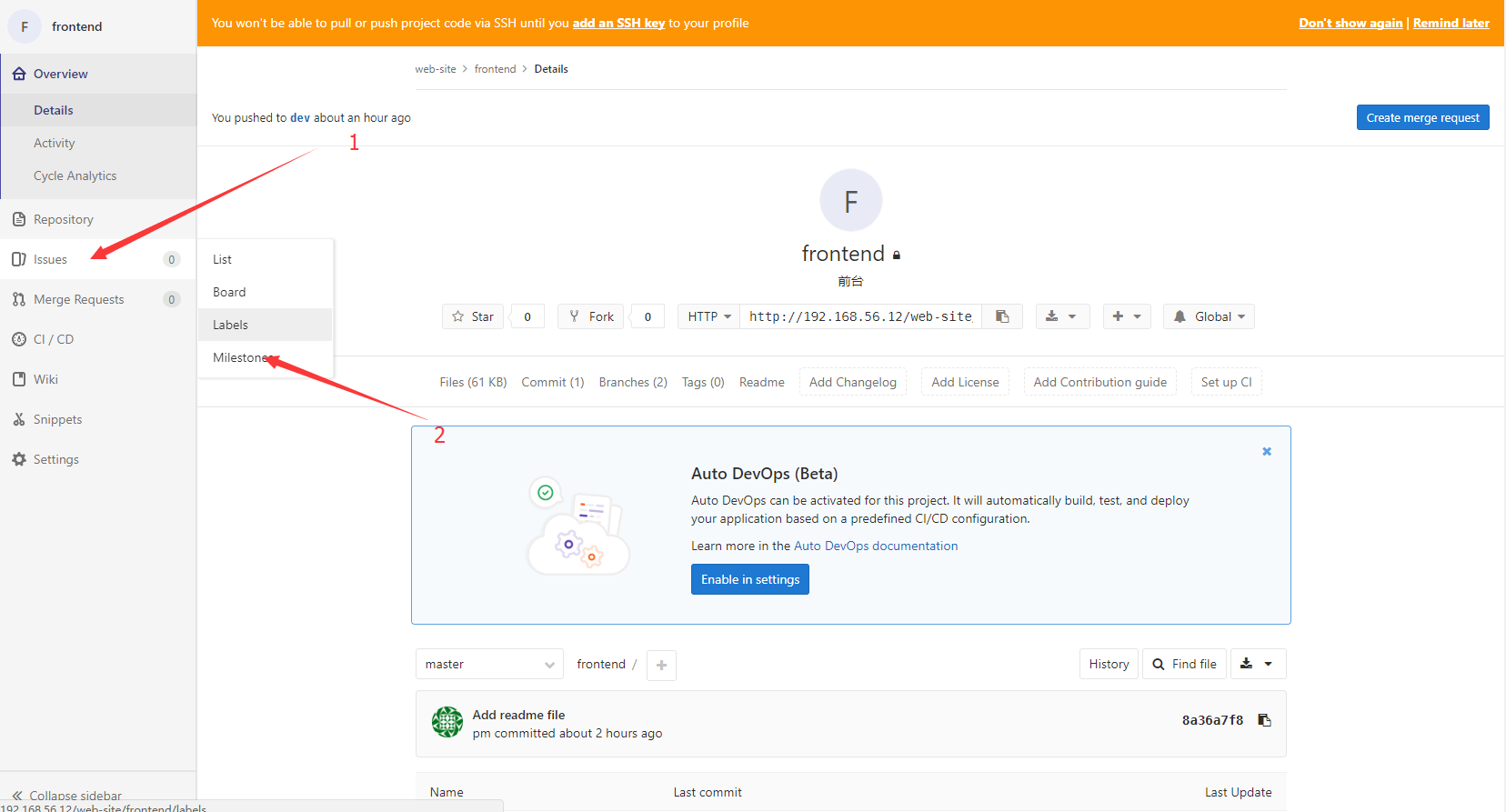
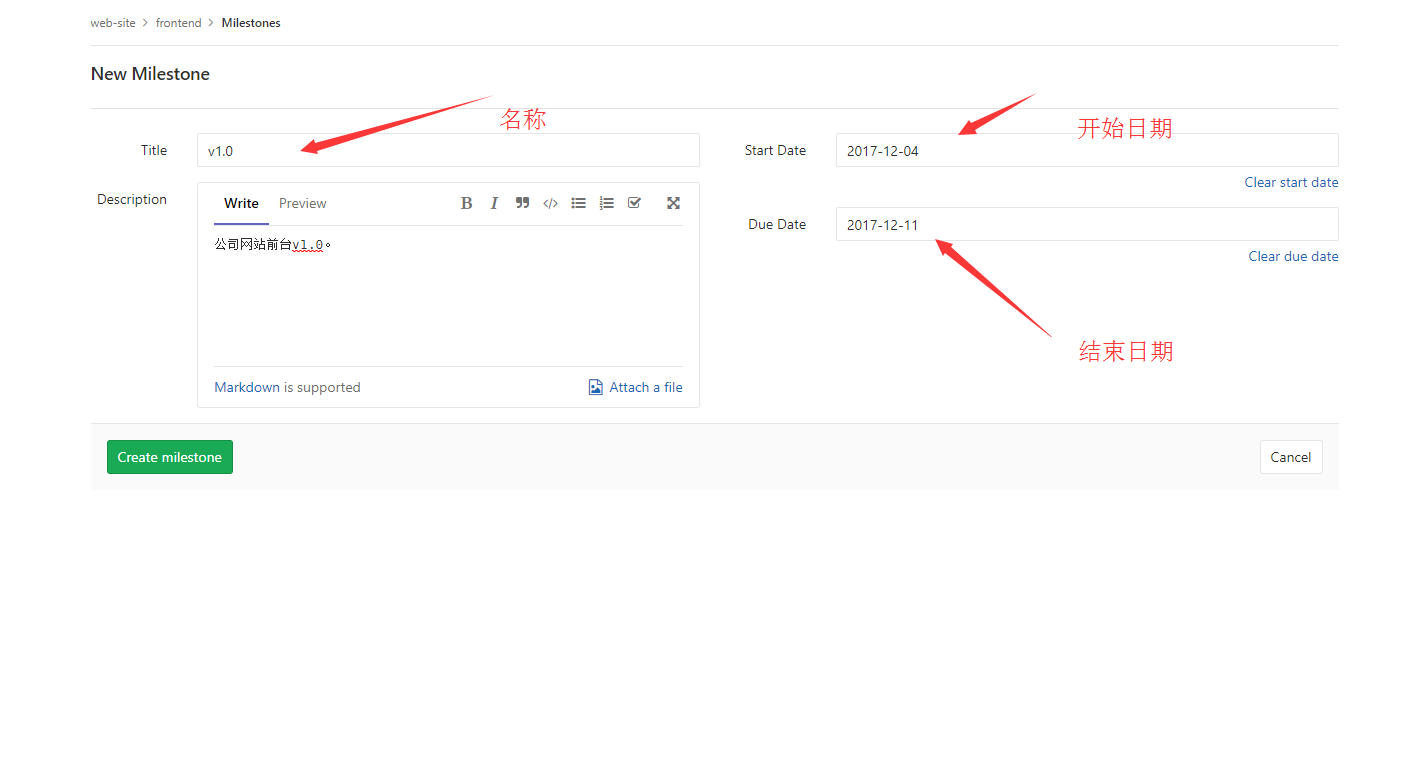
# 五、Gitlab高级使用

在开始我们的讲解前，我们先为我们的frontend项目做一个开发计划：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| v1.0版本：开始时间为12月5日，结束时间为12月11，工期5天 | | | | |
| 模块 | 开发者 | 完成日期 |  |  |
| 首页 | dev1 | 7 |  |  |
| 新闻 | dev1 | 8 |  |  |
| 产品 | dev2 | 9 |  |  |
| 关于我们 | dev2 | 10 |  |  |

## 1、Milestones（里程碑）

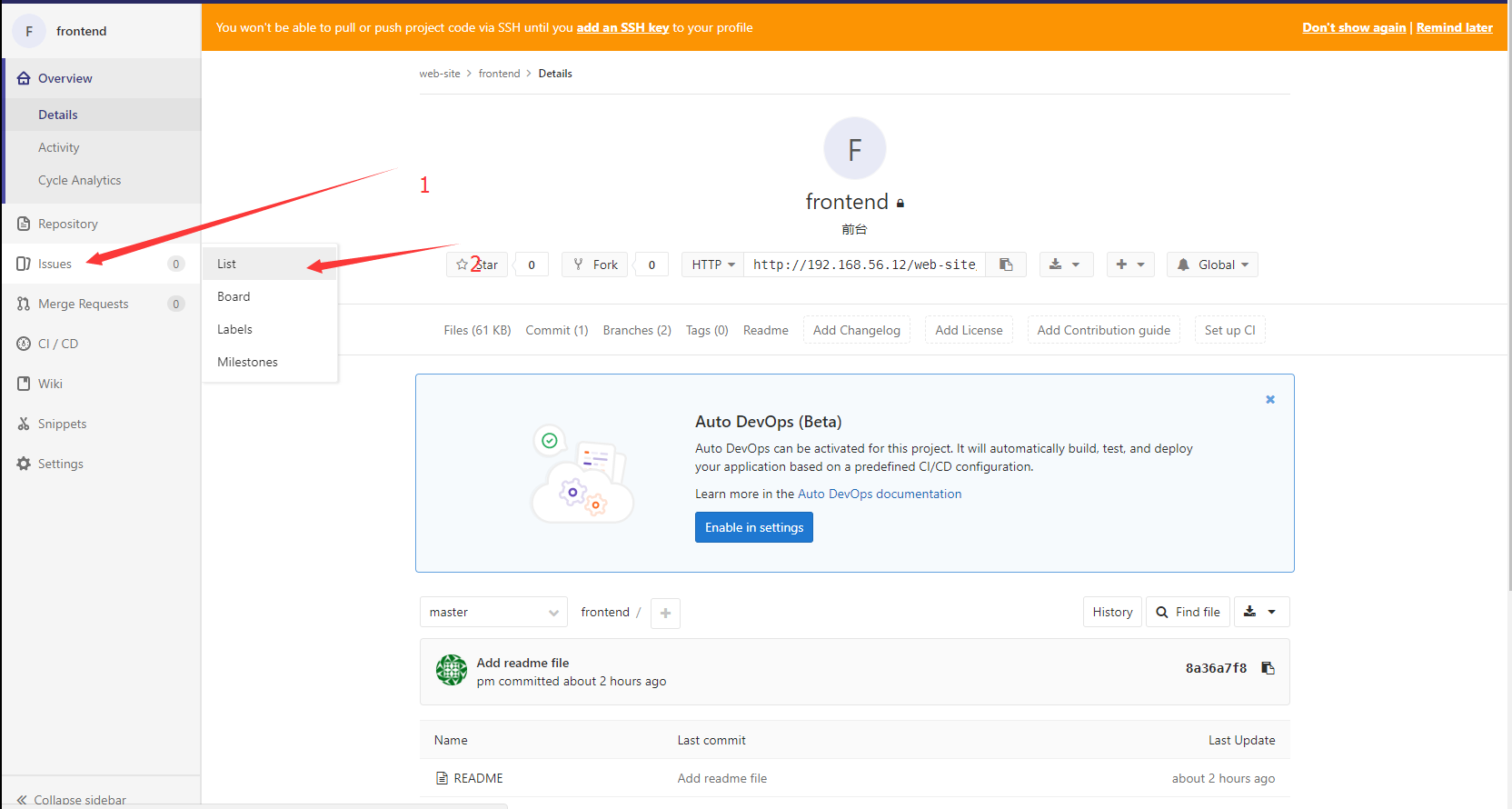
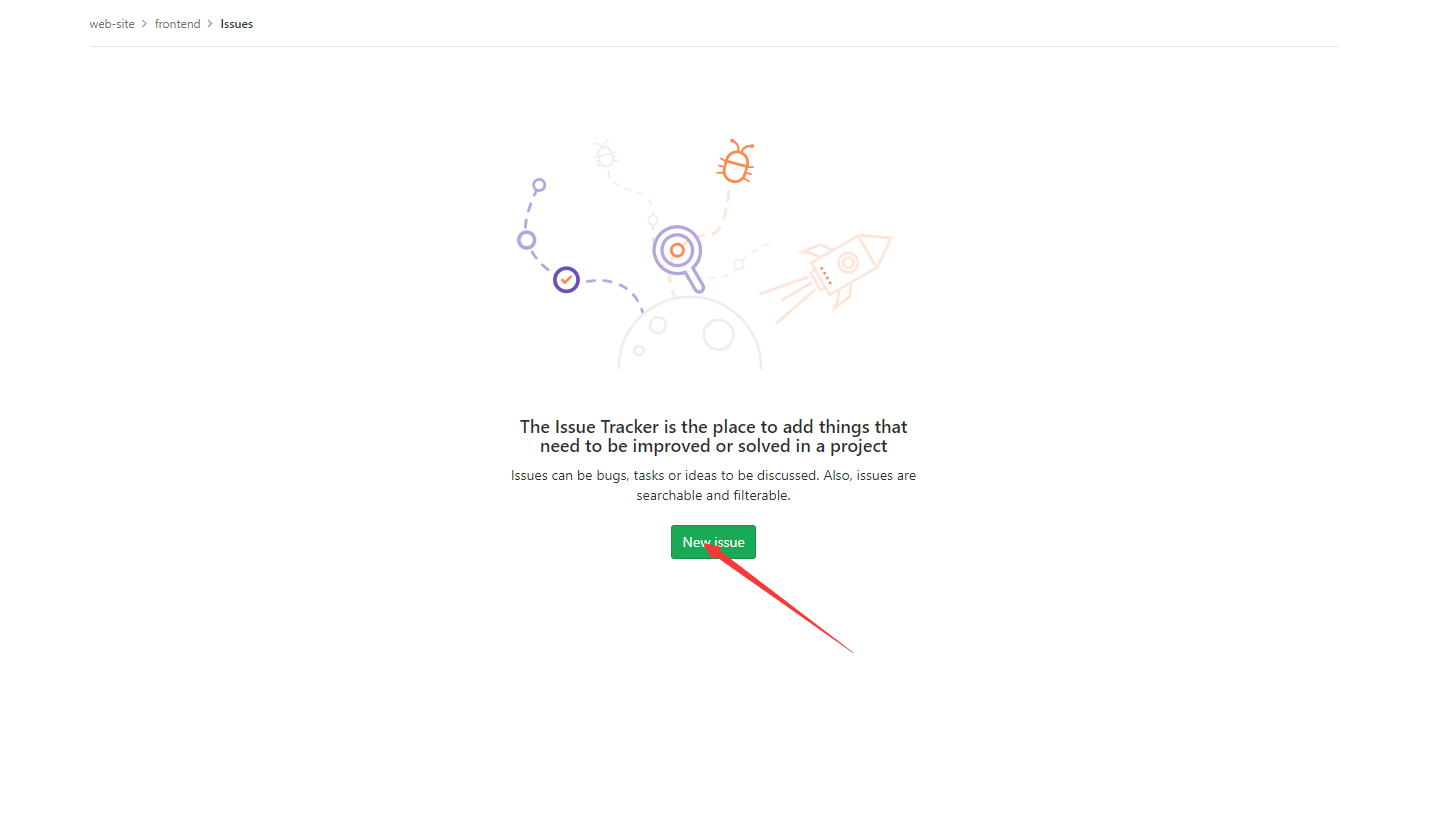
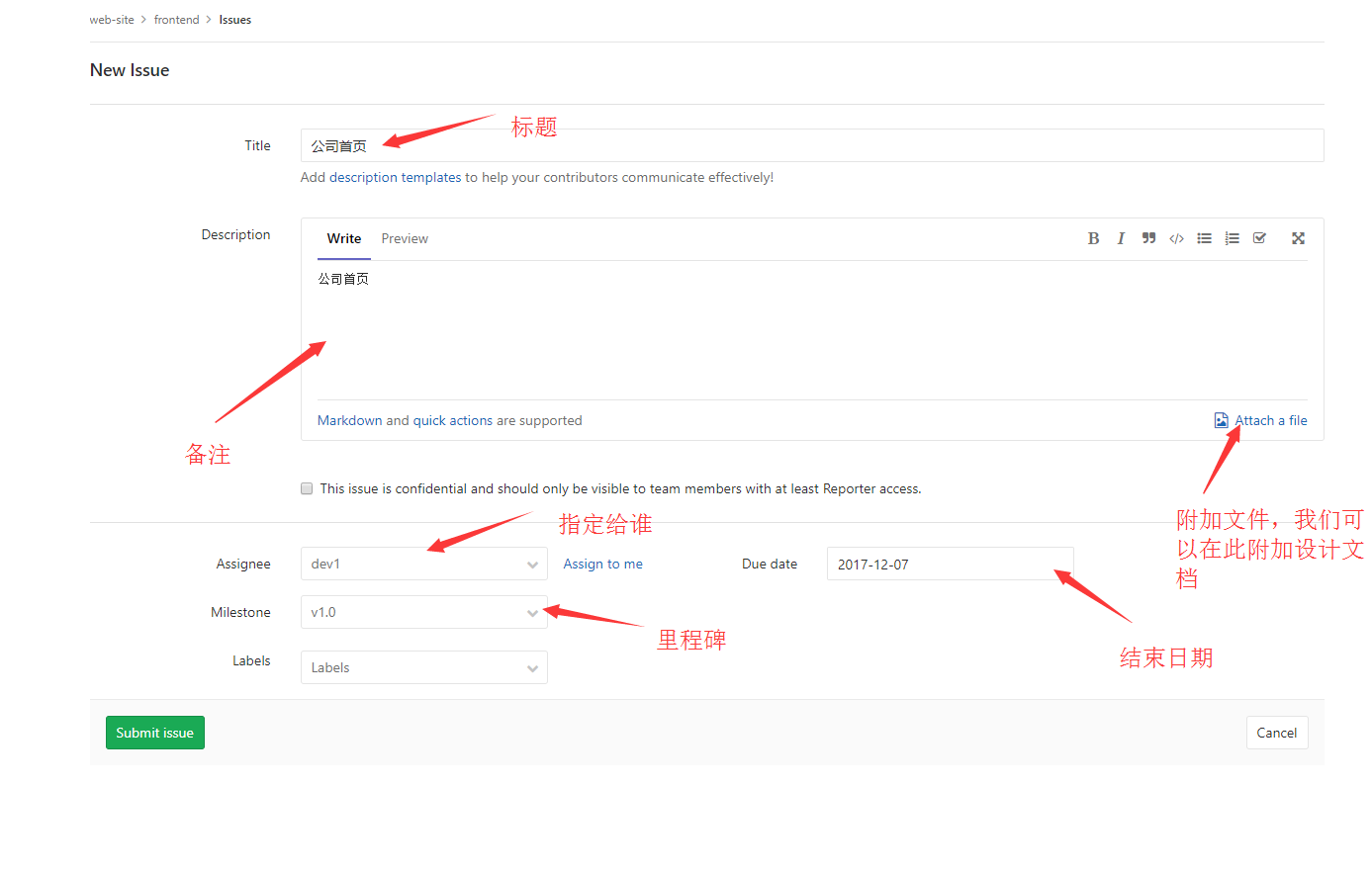
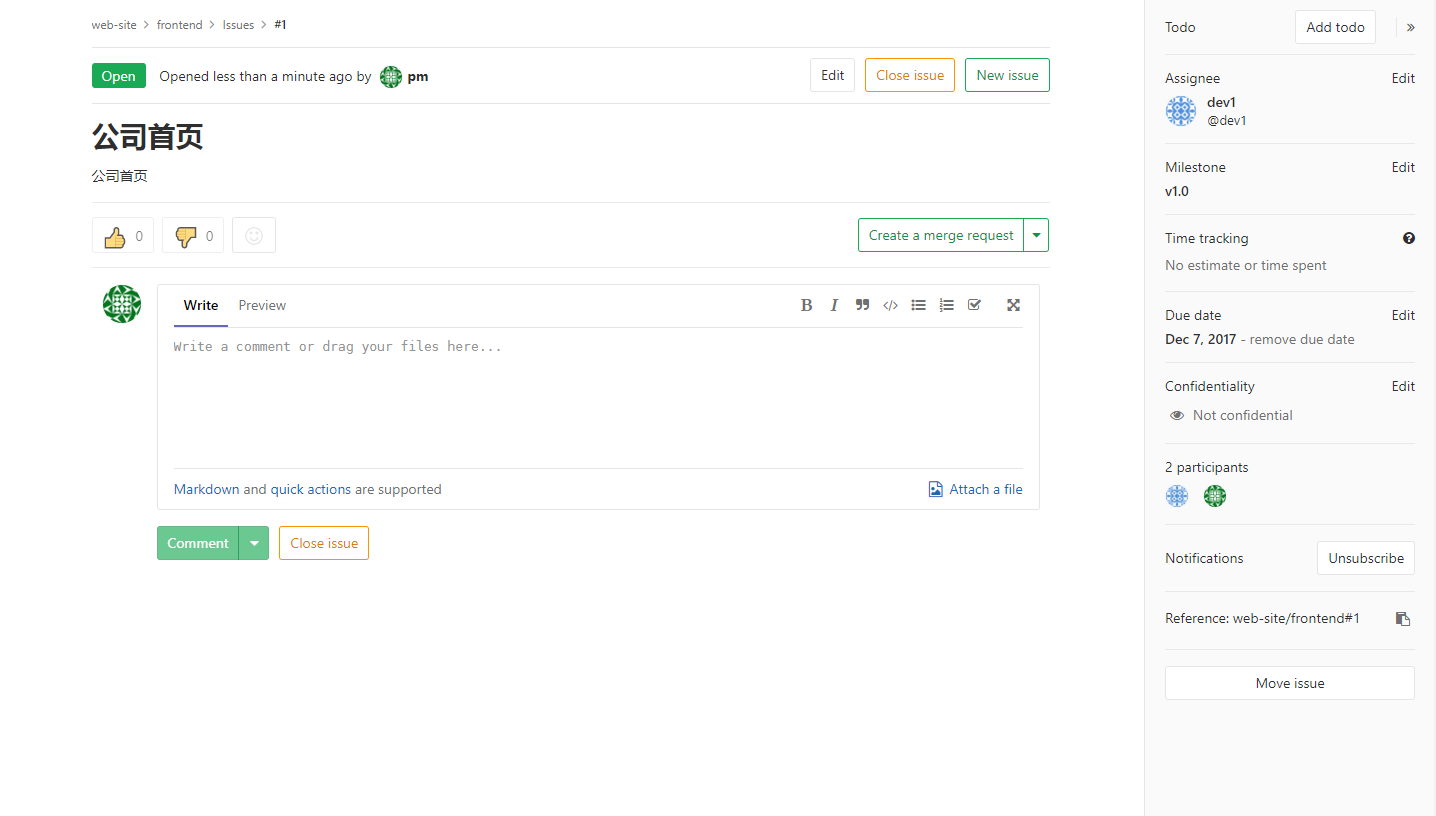
里程碑计划是一个目标计划，它表明为了达到特定的里程碑，去完成一系列活动。根据frontend的开发计划，我们为frontend建立一个v1.0的里程碑。

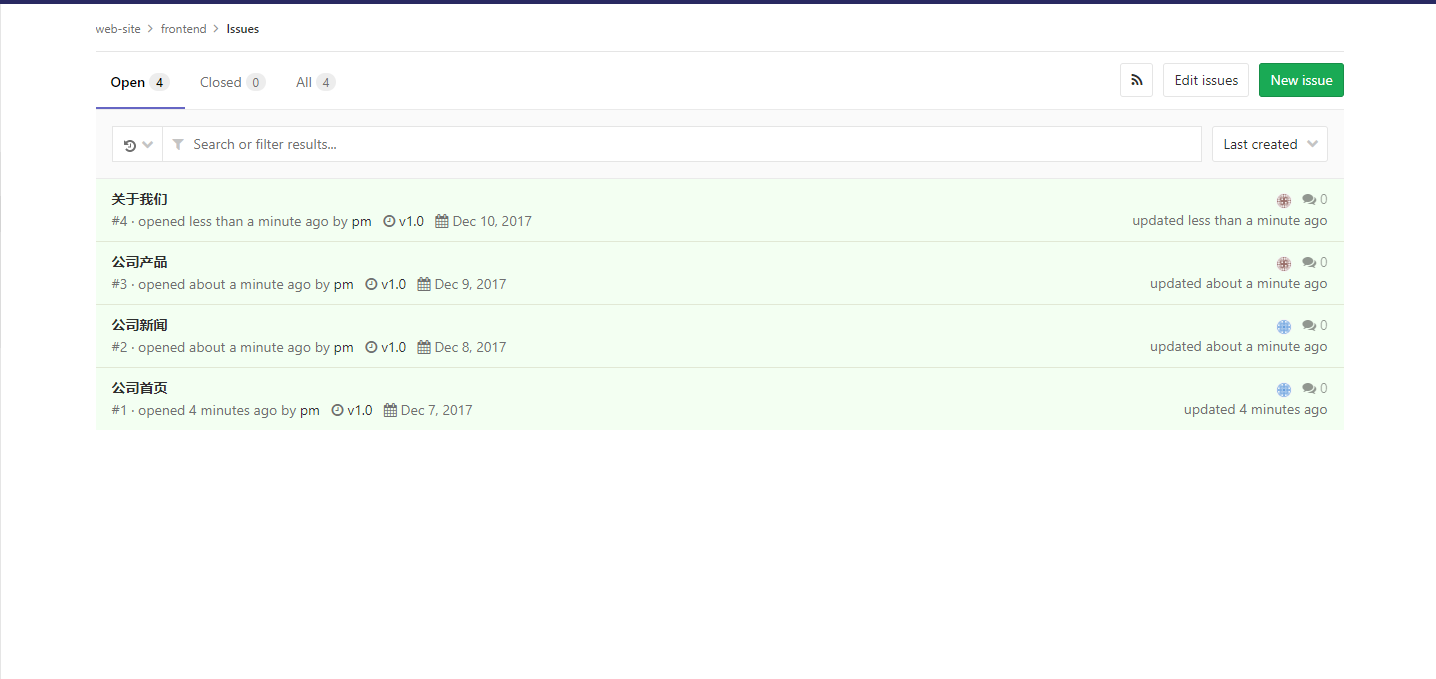
## 2、Issue and Issue Tracker（问题跟踪器）

Issue可以有很多的作用：1、阐述一个新的想法；2、提交功能建议；3、报告bugs等。

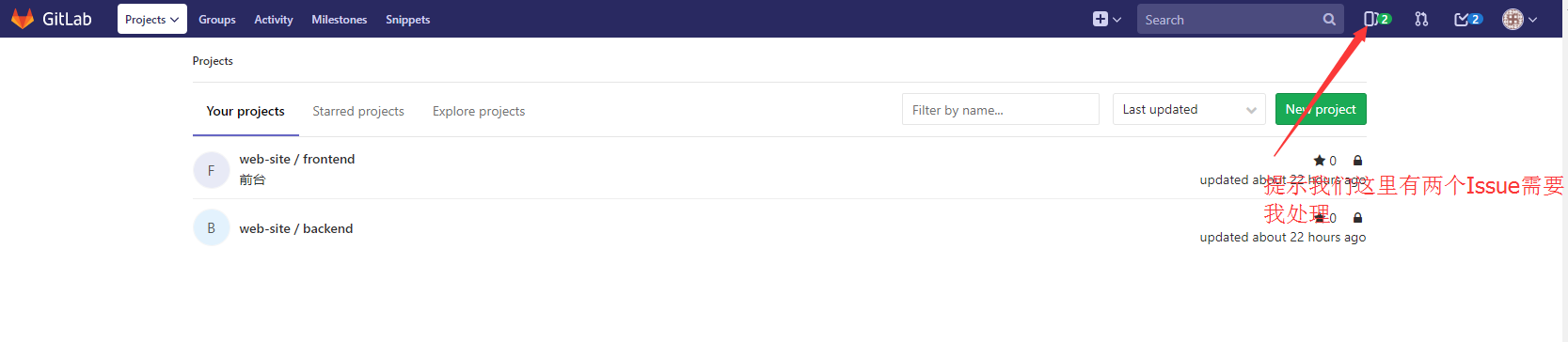
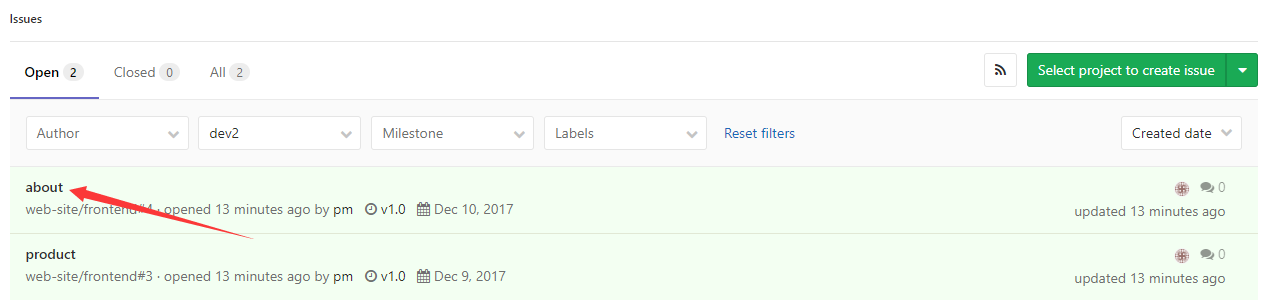
根据我们frontend的开发计划，我们使用Issue来分派计划中的任务：

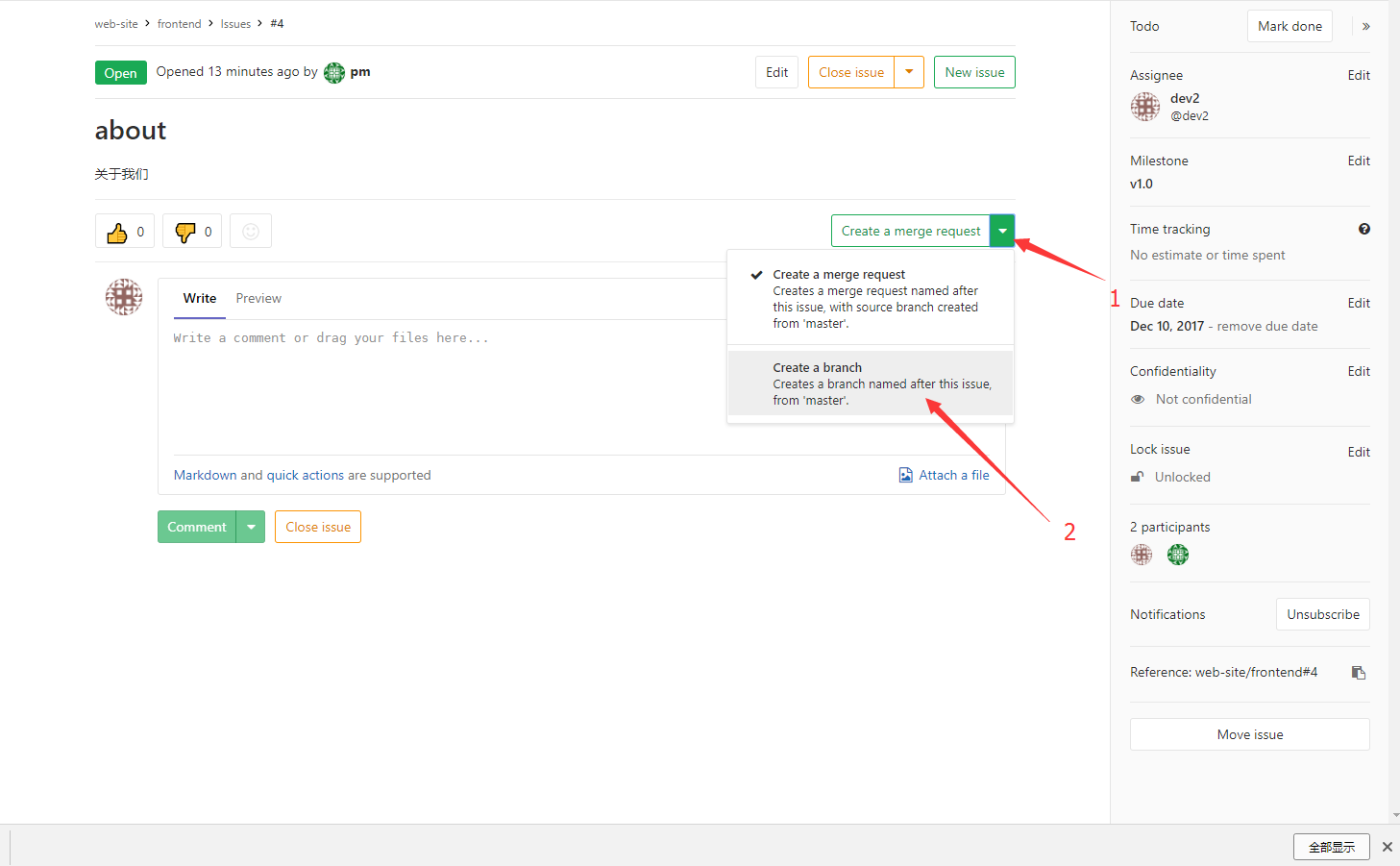
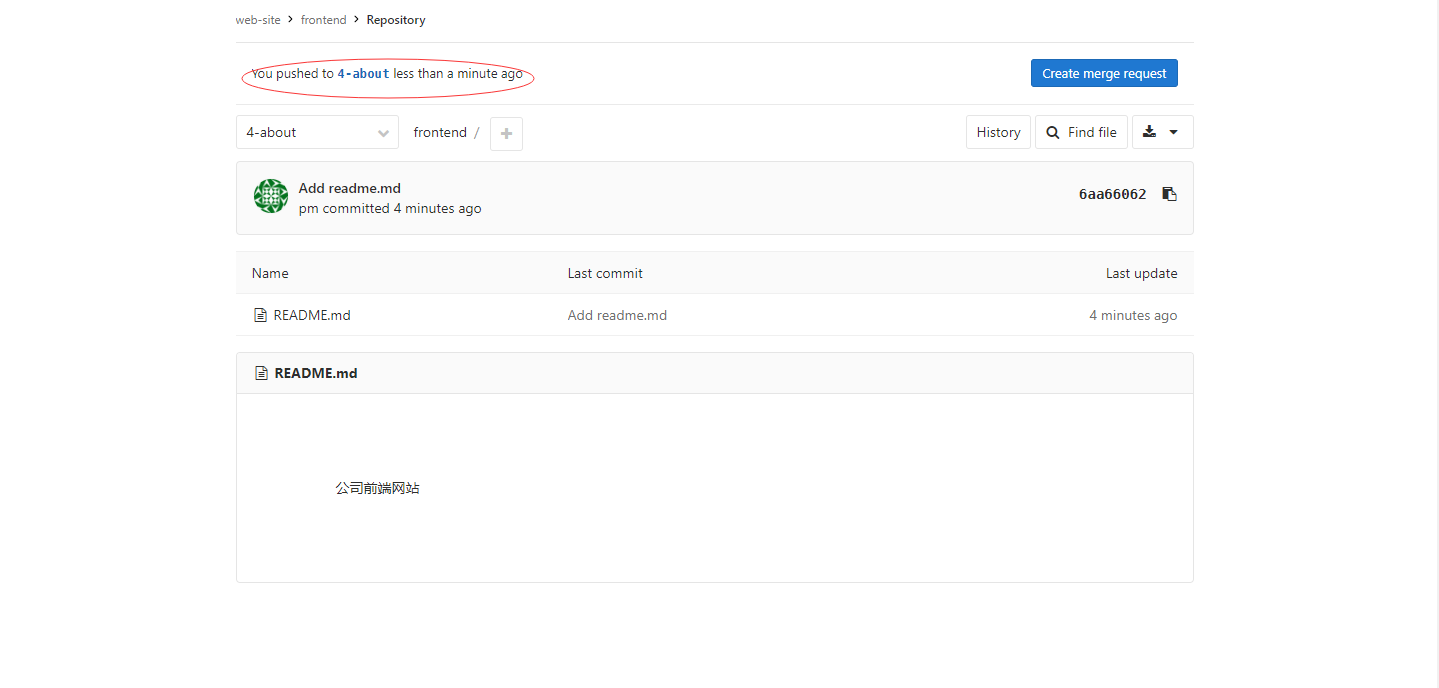
   

使用同样的方法，添加剩下的三个计划。完成后如下：

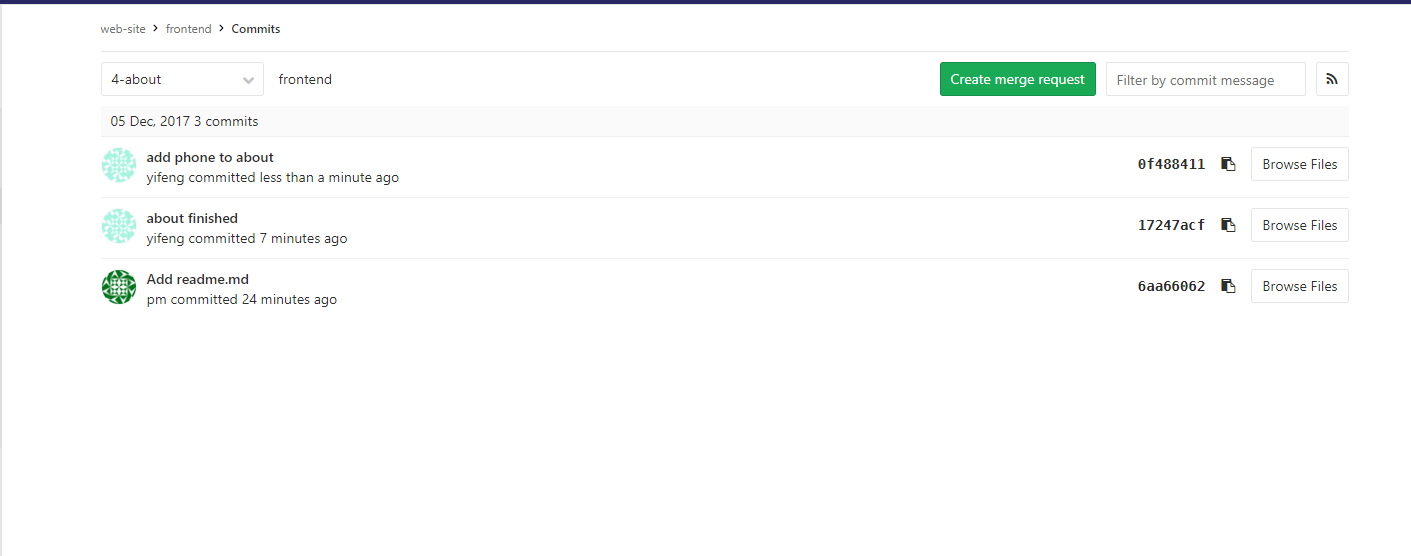


我们使用dev2用户登录，可以看到：

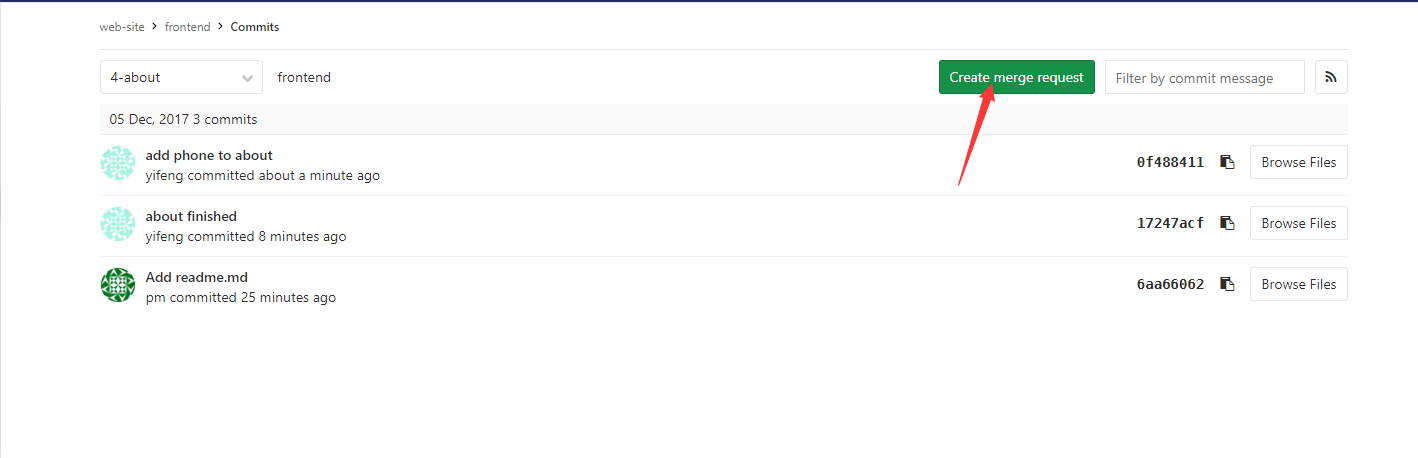
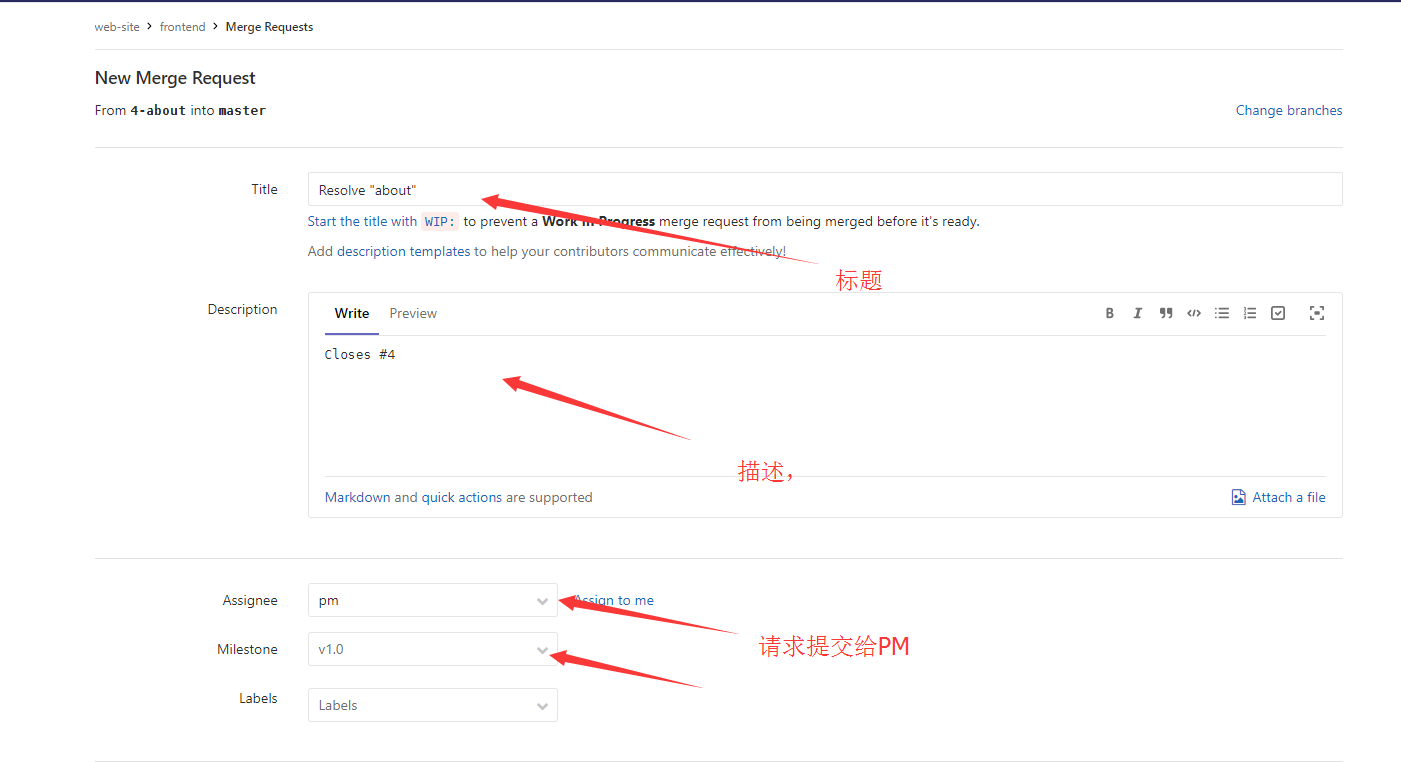
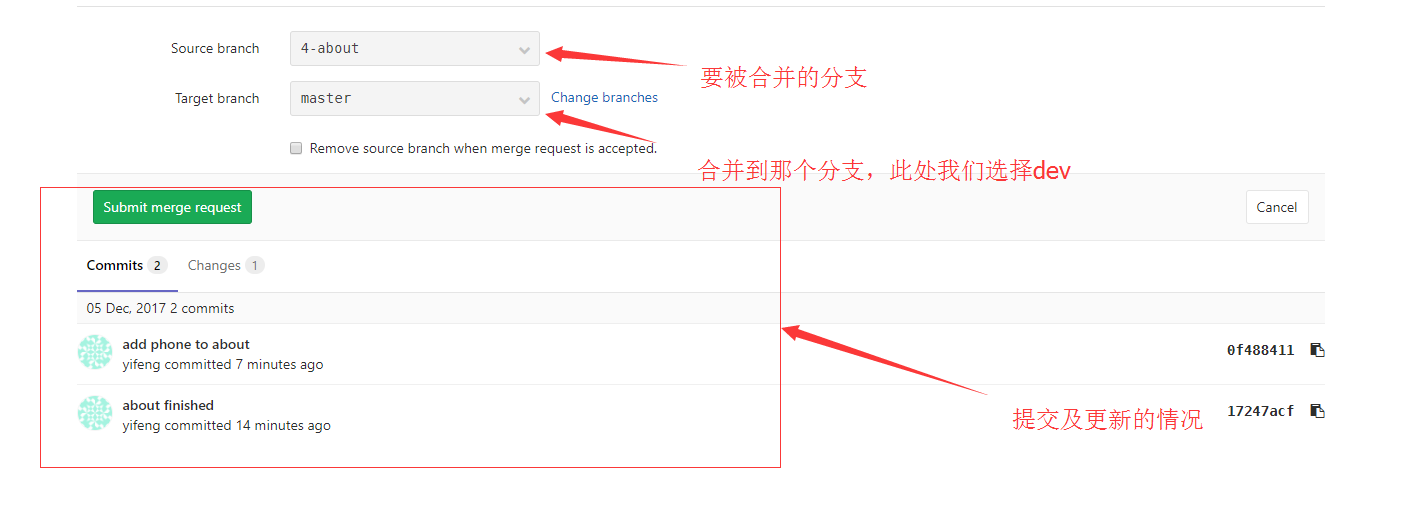
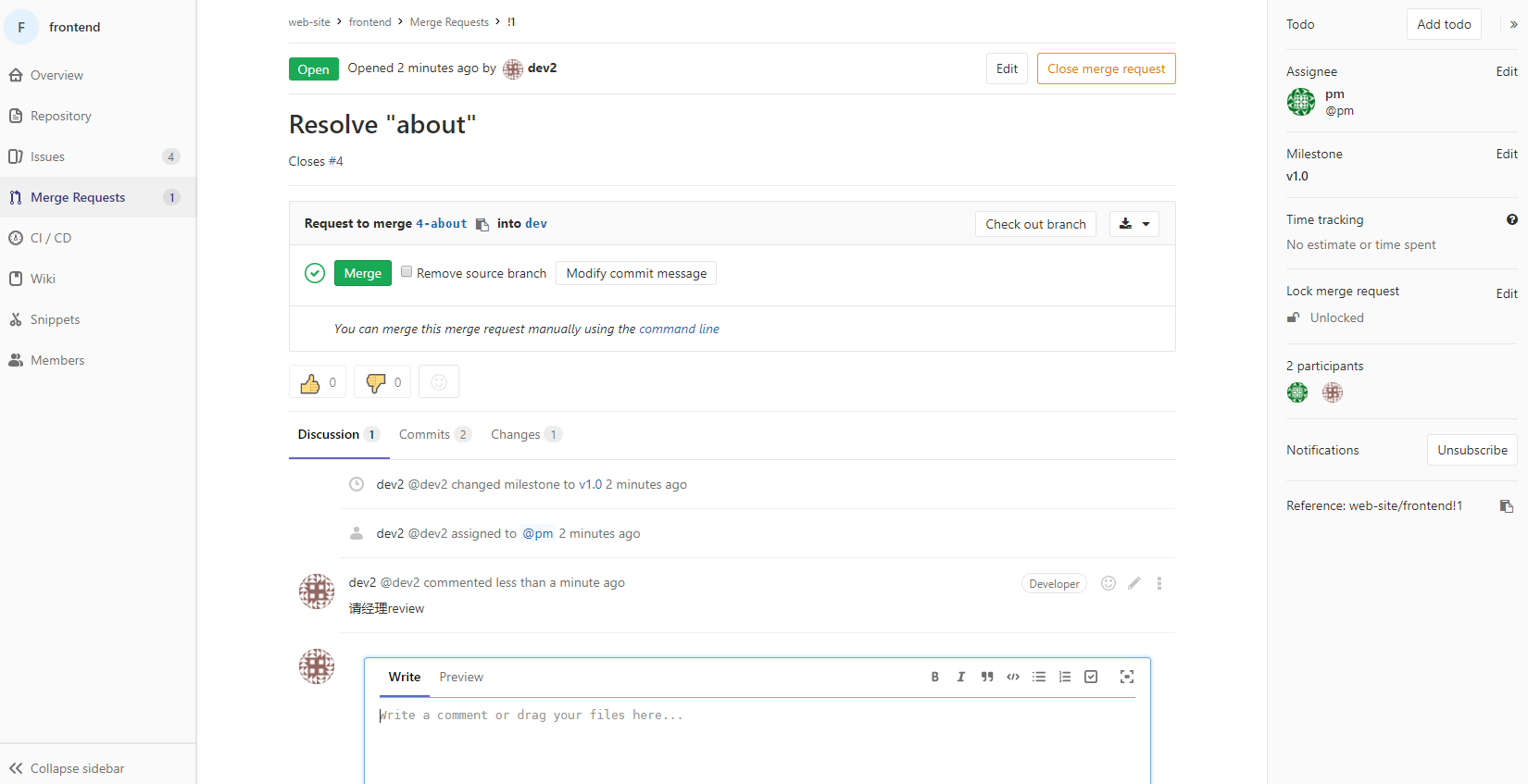
 

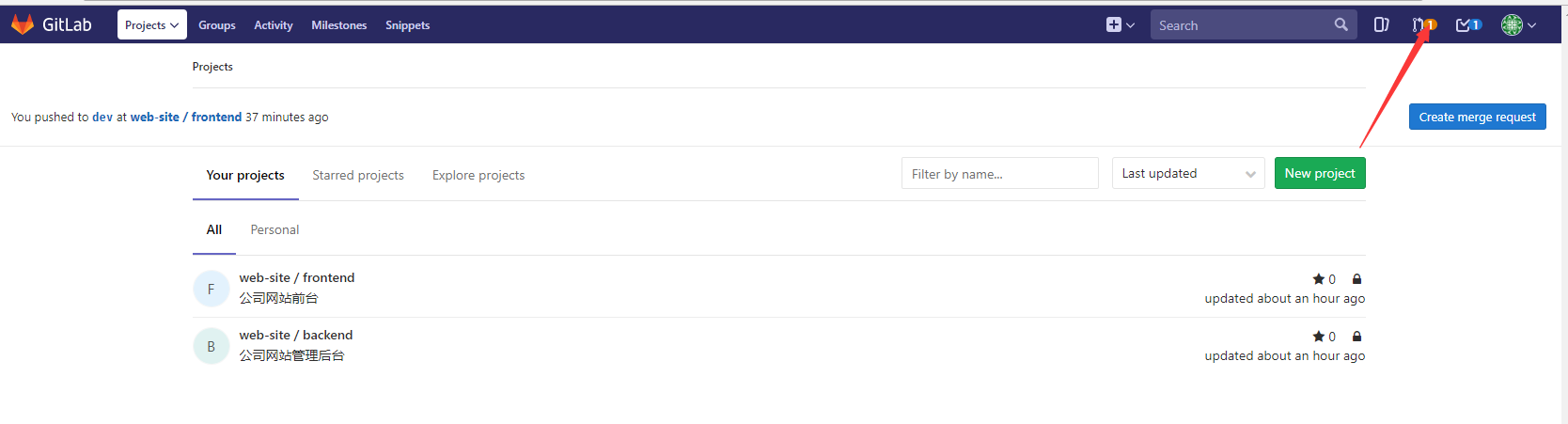
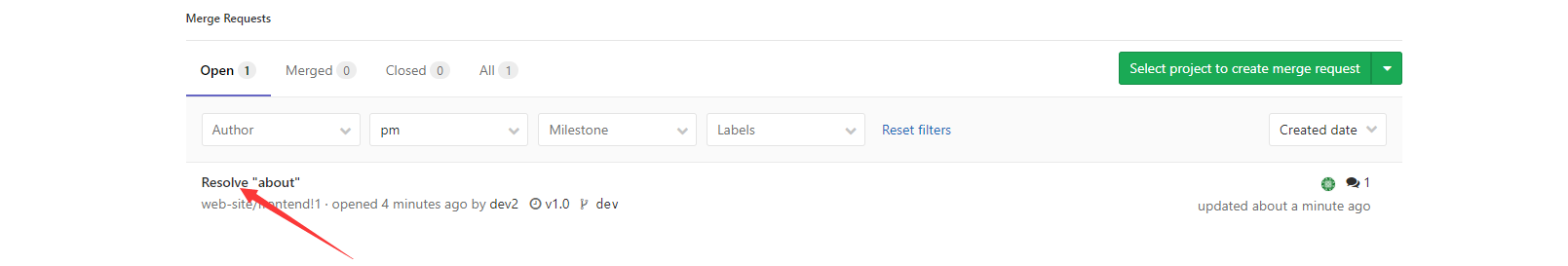
然后我们模拟dev2用户在本地开发about功能，并将开发的代码上传gitlab，然后合并到dev分支。本地编辑文件并上传部分请参考前面（4.7章节），完成后gitlab上4-about分支的状态如下：

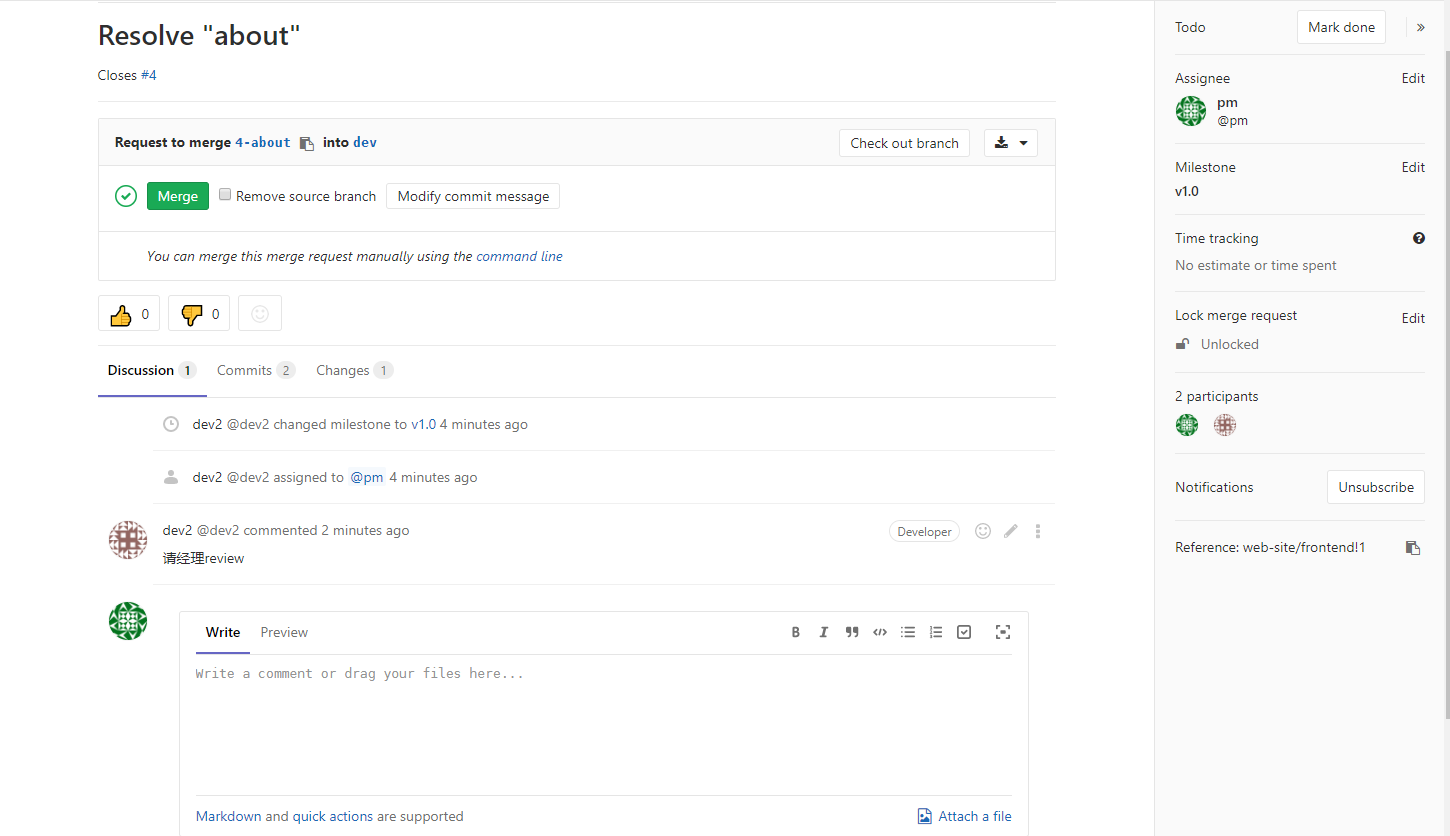


然后我们发送一个合并请求给pm，请我们的about功能合并到开发分支。

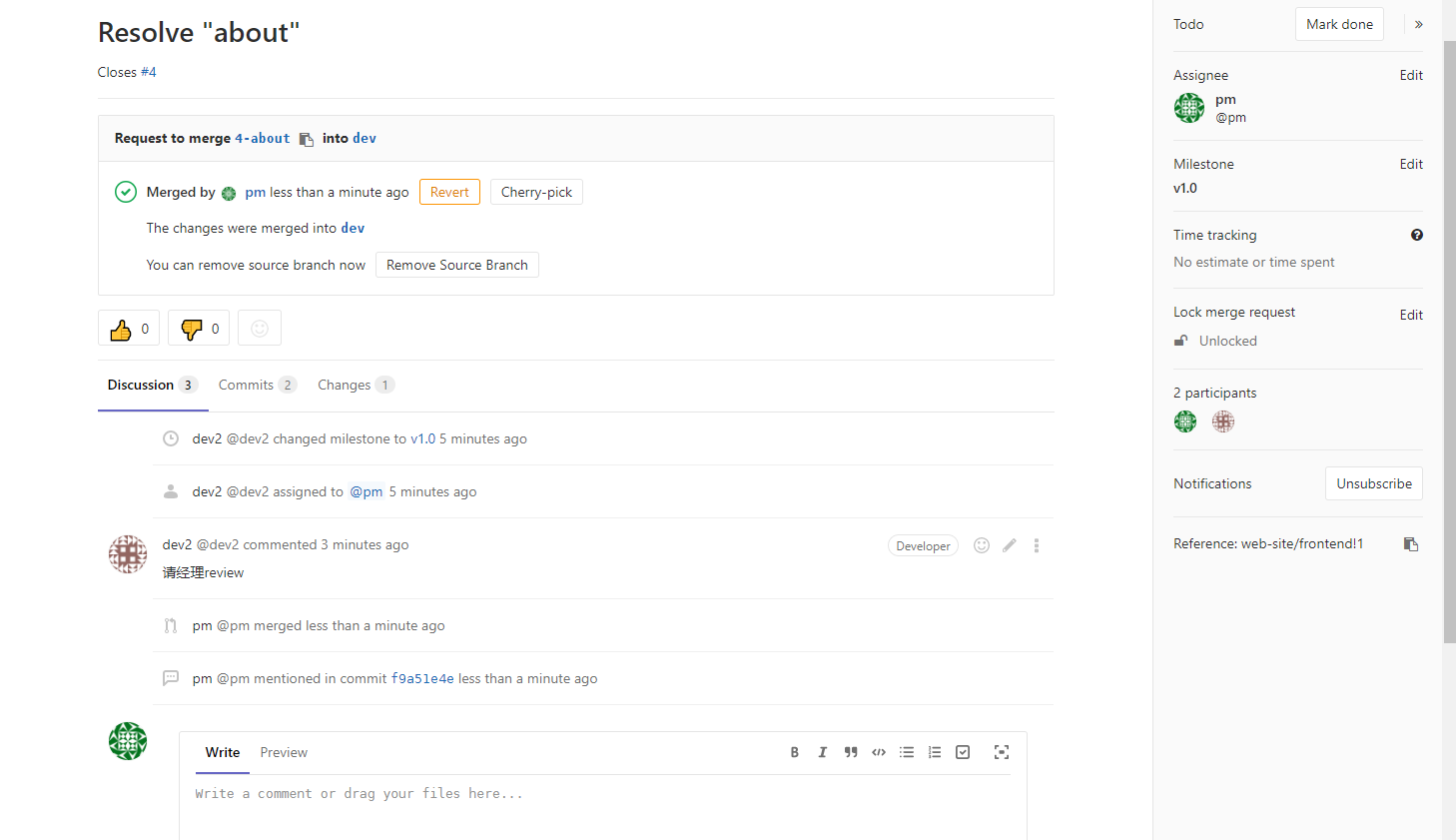
   

然后我们使用pm用户登录：

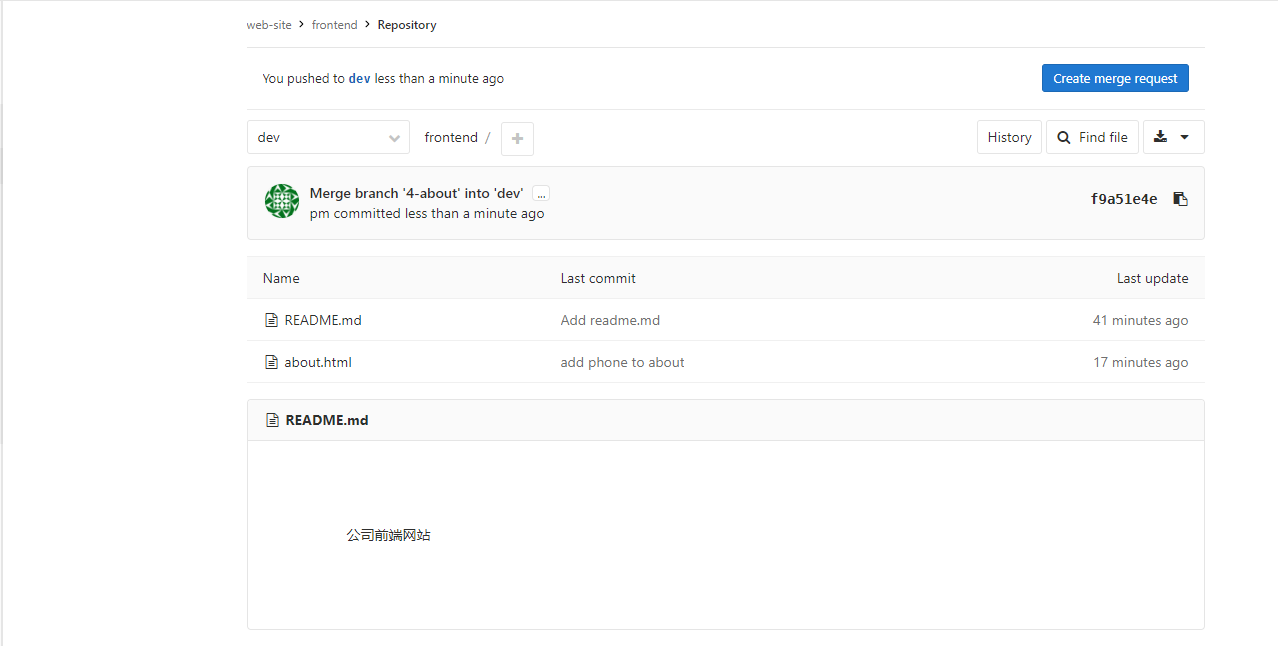
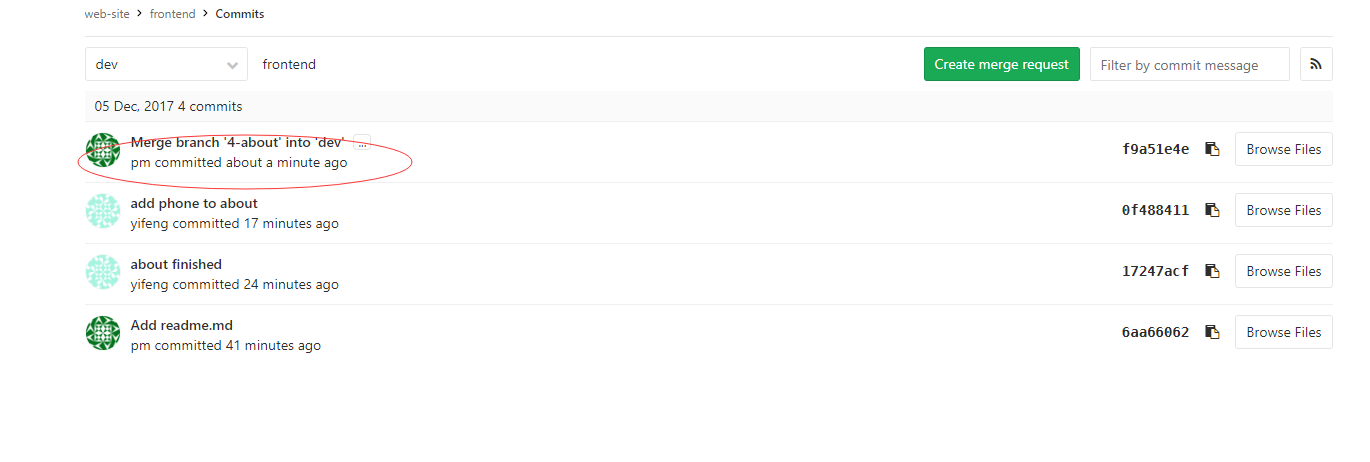
 



检查正确后，点击merge



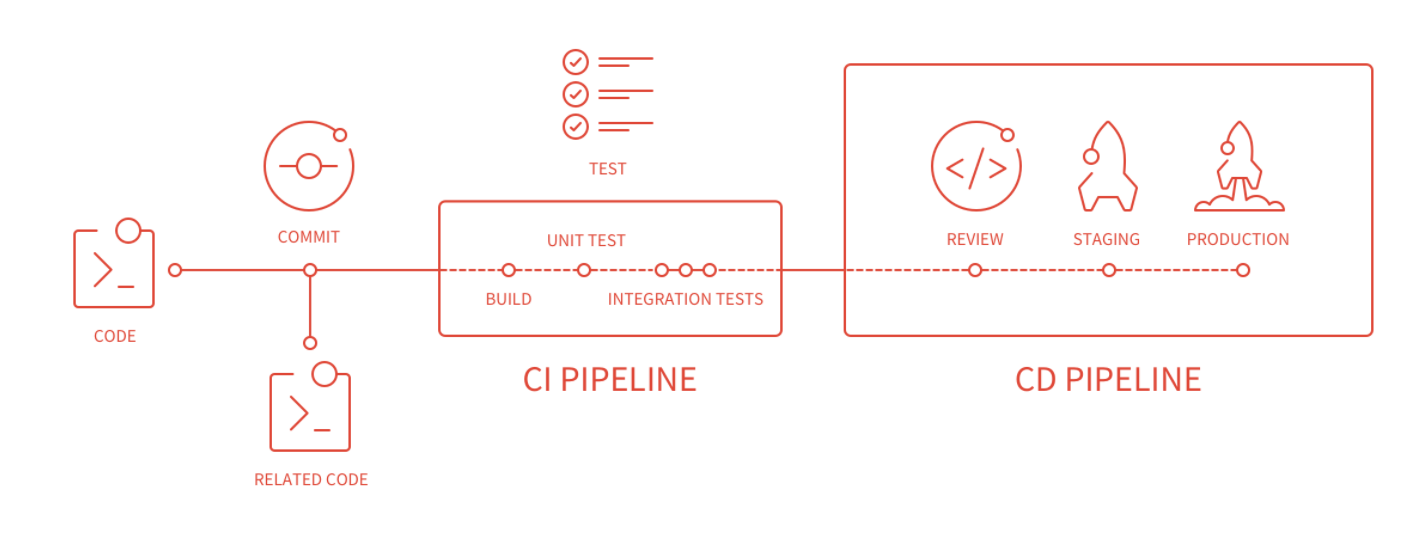
我们可以看about.html文件已经合并到了dev分支，并且生成了一次Merge branch的commit记录。然后我们就可以关闭about 的issue。我们可以使用同样的方法完成其他几个模块的开发，并最后合并到dev分支。当所有的开发工作完成后，由PM将dev分支合并到mster分支，然后我们可以在Master分支创建一个发布版本进行发布。

## 3、GitLab CI

### 3.1简介

gitlab-ci全称是gitlab continuous integration的意思，也就是持续集成。中心思想是当每一次push到gitlab的时候，都会触发一次脚本执行，然后脚本的内容包括了测试，编译，部署等一系列自定义的内容。



GitLab CI是 GitLab 提供的持续集成服务，只要在你的仓库根目录 创建一个.gitlab-ci.yml 文件， 并为该项目指派一个Runner，当有合并请求或者 push的时候就会触发build。

这个.gitlab-ci.yml 文件定义GitLab runner要做哪些操作。 默认有3个[stages(阶段)]: build、test、deploy。

所以简单的说，要让CI工作可总结为以下几点:

在仓库根目录创建一个名为.gitlab-ci.yml 的文件

为该项目配置一个Runner

完成上面的步骤后，每次push代码到Git仓库， Runner就会自动开始pipeline。

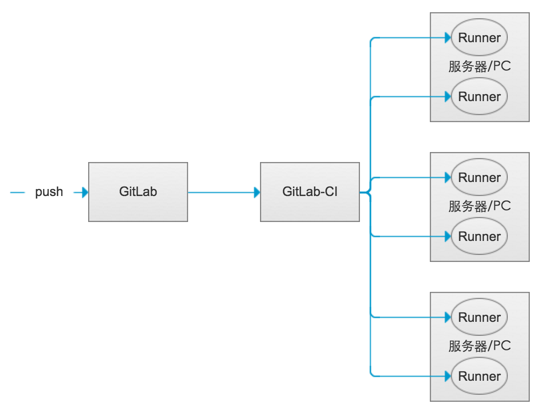
### 3.2组件

GitLab-CI

这个是一套配合GitLab使用的持续集成系统，是GitLab自带的，也就是你装GitLab的那台服务器上就带有的。无需多考虑。.gitlab-ci.yml的脚本解析就由它来负责。

GitLab-Runner

这个是脚本执行的承载者，.gitlab-ci.yml的script部分的运行就是由runner来负责的。GitLab-CI浏览过项目里的.gitlab-ci.yml文件之后，根据里面的规则，分配到各个Runner来运行相应的脚本script。这些脚本有的是测试项目用的，有的是部署用的。



因为 GitLab Runner 可以安装到不同的机器上，所以在构建任务运行期间并不会影响到 GitLab 的性能

.gitlab-ci.yml

这个是在git项目的根目录下的一个文件，记录了一系列的阶段和执行规则。GitLab-CI在push后会解析它，根据里面的内容调用runner来运行。

### 3.3安装

安装GitLab-CI  
这个不用安装了，装好GitLab就自带了

安装GitLab-Runner

[root@node1 src]# cd /usr/local/src/

[root@node1 src]# wget https://mirrors.tuna.tsinghua.edu.cn/gitlab-ci-multi-runner/yum/el7/gitlab-ci-multi-runner-9.5.1-1.x86\_64.rpm

[root@node1 src]# rpm -ivh gitlab-ci-multi-runner-9.5.1-1.x86\_64.rpm

### 3.4注册

向gitlab-CI注册这个runner

[root@node1 src]# gitlab-ci-multi-runner register

Running in system-mode.

Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):

http://192.168.56.12 gitlab地址

Please enter the gitlab-ci token for this runner:

WUsLpmDFu6q4snDSqmds project的token

Please enter the gitlab-ci description for this runner:

[node1]: node1 runner的描述

Please enter the gitlab-ci tags for this runner (comma separated):

node1-runner runner标签

Whether to run untagged builds [true/false]:

[false]: true

Whether to lock Runner to current project [true/false]:

[false]: false

Registering runner... succeeded runner=WUsLpmDF

Please enter the executor: docker, docker-ssh, virtualbox, kubernetes, parallels, shell, ssh, docker+machine, docker-ssh+machine:

shell 选择runner的类型

Runner registered successfully. Feel free to start it, but if it's running already the config should be automatically reloaded!

[root@node1 src]# gitlab-ci-multi-runner list

Listing configured runners ConfigFile=/etc/gitlab-runner/config.toml

node1 Executor=shell Token=bcb3699e350002b905bcfead2f3f6d URL=http://192.168.56.12

[root@node1 src]#

### 3.5启动

[root@node1 src]# gitlab-ci-multi-runner start

### 3.6编写gitlab-ci.yml

我们先介绍几个基本概念：

Pipeline

一次 Pipeline 其实相当于一次构建任务，里面可以包含多个流程，如安装依赖、运行测试、编译、部署测试服务器、部署生产服务器等流程。

Stages

Stages 表示构建阶段，说白了就是上面提到的流程。

我们可以在一次 Pipeline 中定义多个 Stages，这些 Stages 会有以下特点：

所有 Stages 会按照顺序运行，即当一个 Stage 完成后，下一个 Stage 才会开始

只有当所有 Stages 完成后，该构建任务 (Pipeline) 才会成功

如果任何一个 Stage 失败，那么后面的 Stages 不会执行，该构建任务 (Pipeline) 失败

Jobs

Jobs 表示构建工作，表示某个 Stage 里面执行的工作。

我们可以在 Stages 里面定义多个 Jobs，这些 Jobs 会有以下特点：

相同 Stage 中的 Jobs 会并行执行

相同 Stage 中的 Jobs 都执行成功时，该 Stage 才会成功

如果任何一个 Job 失败，那么该 Stage 失败，即该构建任务 (Pipeline) 失败

基本写法

# 定义 stages

stages:

- build

- deploy

# 定义 job

job1:

stage: deploy

script:

- echo "I am job1"

- scp web.tar.gz /tmp

- pwd

# 定义 job

job2:

stage: build

script:

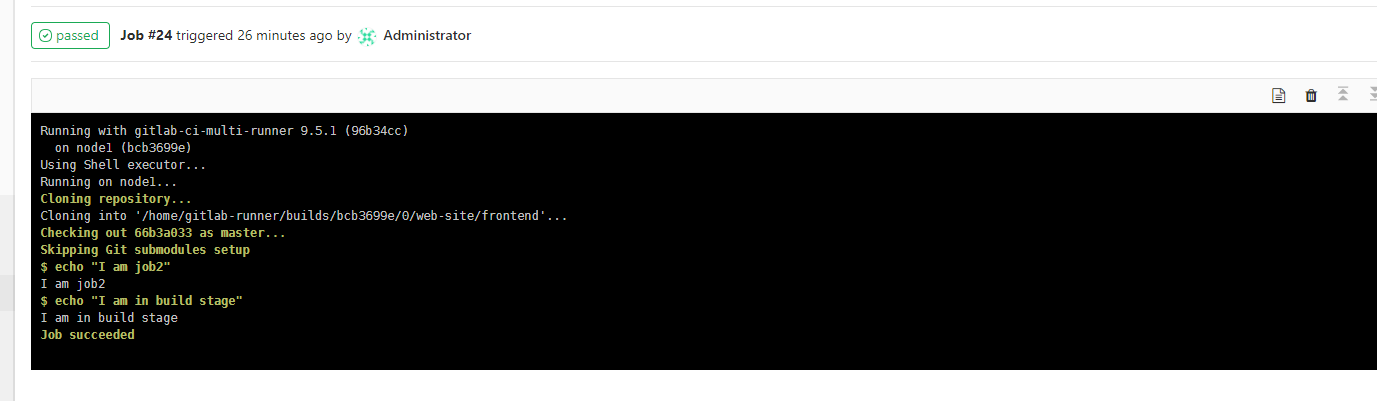
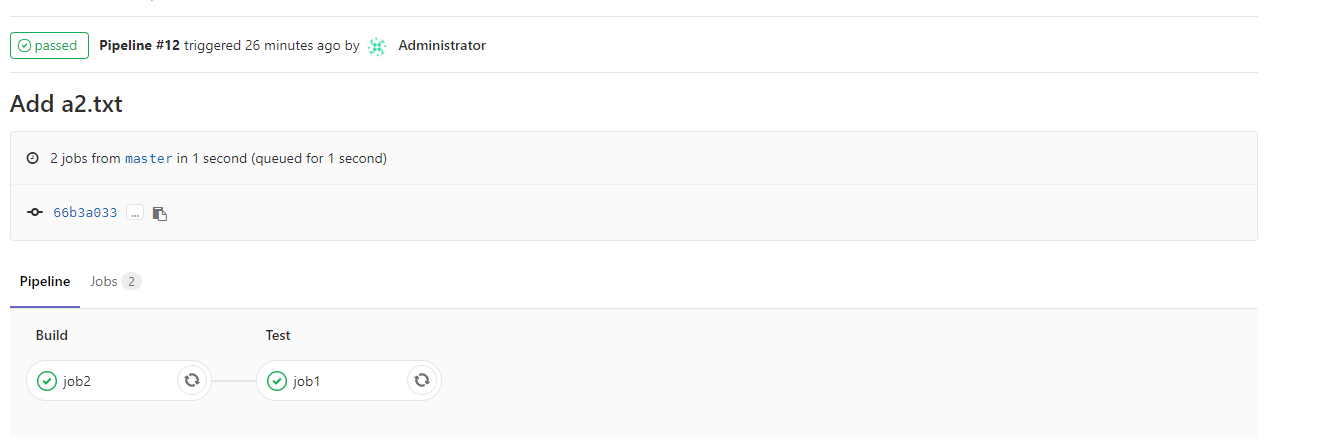
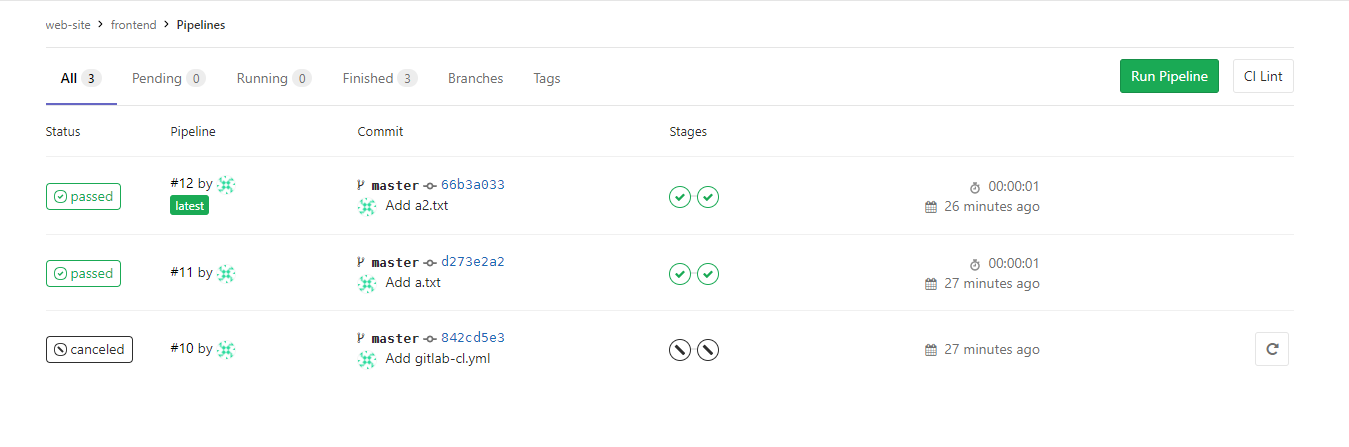
- echo "I am job2"

- tar czf web.tar.gz ./\*

用 stages 关键字来定义 Pipeline 中的各个构建阶段，然后用一些非关键字来定义 jobs。

每个 job 中可以可以再用 stage 关键字来指定该 job 对应哪个 stage。

job 里面的 script 关键字是最关键的地方了，也是每个 job 中必须要包含的，它表示每个 job 要执行的命令。



# 六、GitLab备份

对gitlab进行备份将会创建一个包含所有库和附件的归档文件。对备份的恢复只能恢复到与备份时的gitlab相同的版本。将gitlab迁移到另一台服务器上的最佳方法就是通过备份和还原。

gitlab提供了一个简单的命令行来备份整个gitlab，并且能灵活的满足需求。

备份文件将保存在配置文件中定义的backup\_path中，文件名为TIMESTAMP\_gitlab\_backup.tar,TIMESTAMP为备份时的时间戳。TIMESTAMP的格式为：EPOCH\_YYYY\_MM\_DD\_Gitlab-version。

如果自定义备份目录需要赋予git权限

配置文件中加入

gitlab\_rails['backup\_path'] = '/data/backup/gitlab'

gitlab\_rails['backup\_keep\_time'] = 604800 备份保留的时间（以秒为单位，这个是七天默认值），

mkdir /data/backup/gitlab

chown -R git.git /data/backup/gitlab

完成后执行gitlab-ctl reconfigure

### 1、手动备份

执行：gitlab-rake gitlab:backup:create生成一次备份。

[root@node2 ~]# gitlab-rake gitlab:backup:create

Dumping database ...

Dumping PostgreSQL database gitlabhq\_production ... [DONE]

done

Dumping repositories ...

\* web-site/frontend ... [DONE]

\* web-site/frontend.wiki ... [SKIPPED]

\* web-site/backend ... [SKIPPED]

\* web-site/backend.wiki ... [SKIPPED]

\* devops/accout ... [DONE]

\* devops/accout.wiki ... [SKIPPED]

\* devops/user ... [DONE]

\* devops/user.wiki ... [SKIPPED]

\* web-site/accout ... [DONE]

\* web-site/accout.wiki ... [SKIPPED]

done

Dumping uploads ...

done

Dumping builds ...

done

Dumping artifacts ...

done

Dumping pages ...

done

Dumping lfs objects ...

done

Dumping container registry images ...

[DISABLED]

Creating backup archive: 1512811475\_2017\_12\_09\_10.2.2\_gitlab\_backup.tar ... done

Uploading backup archive to remote storage ... skipped

Deleting tmp directories ... done

done

done

done

done

done

done

done

Deleting old backups ... skipping

[root@node2 ~]# ll /var/opt/gitlab/backups/

total 272

-rw------- 1 git git 276480 Dec 9 17:24 1512811475\_2017\_12\_09\_10.2.2\_gitlab\_backup.tar

### 2、定时备份

在定时任务里添加：

0 2 \* \* \* /opt/gitlab/bin/gitlab-rake gitlab:backup:create CRON=1

环境变量CRON=1的作用是如果没有任何错误发生时， 抑制备份脚本的所有进度输出。

### 3、恢复

只能还原到与备份文件相同的gitlab版本。

执行恢复操作时，需要gitlab处于运行状态，备份文件位于gitlab\_rails['backup\_path']。

[root@node2 ~]# ll /var/opt/gitlab/backups/

total 272

-rw------- 1 git git 276480 Dec 9 17:24 1512811475\_2017\_12\_09\_10.2.2\_gitlab\_backup.tar

停止连接到数据库的进程（也就是停止数据写入服务），但是保持GitLab是运行的。

[root@node2 ~]# gitlab-ctl stop unicorn

ok: down: unicorn: 0s, normally up

[root@node2 ~]# gitlab-ctl stop sidekiq

ok: down: sidekiq: 0s, normally up

确认：

[root@node2 ~]# gitlab-ctl status

run: gitaly: (pid 1497) 0s; run: log: (pid 540) 0s

run: gitlab-monitor: (pid 1507) 0s; run: log: (pid 543) 0s

run: gitlab-workhorse: (pid 1517) 0s; run: log: (pid 508) 0s

run: logrotate: (pid 14405) 1564s; run: log: (pid 510) 0s

run: nginx: (pid 1532) 0s; run: log: (pid 507) 0s

run: node-exporter: (pid 1538) 0s; run: log: (pid 525) 0s

run: postgres-exporter: (pid 1543) 0s; run: log: (pid 530) 0s

run: postgresql: (pid 1551) 0s; run: log: (pid 492) 0s

run: prometheus: (pid 1559) 0s; run: log: (pid 535) 0s

run: redis: (pid 1567) 0s; run: log: (pid 491) 0s

run: redis-exporter: (pid 1572) 0s; run: log: (pid 547) 0s

down: sidekiq: 121s, normally up; run: log: (pid 500) 0s

down: unicorn: 133s, normally up; run: log: (pid 502) 0s

[root@node2 ~]#

接下我们进行恢复，指定时间戳你要从那个备份恢复：

[root@node2 ~]# gitlab-rake gitlab:backup:restore BACKUP=1512811475\_2017\_12\_09\_10.2.2

Unpacking backup ... done

Before restoring the database, we will remove all existing

tables to avoid future upgrade problems. Be aware that if you have

custom tables in the GitLab database these tables and all data will be

removed.

Do you want to continue (yes/no)?

将移除我们自建的表。回答yes

Restoring uploads ...

done

Restoring builds ...

done

Restoring artifacts ...

done

Restoring pages ...

done

Restoring lfs objects ...

done

This will rebuild an authorized\_keys file.

You will lose any data stored in authorized\_keys file.

Do you want to continue (yes/no)?

将移除所有的认证Key。回答yes

....

Deleting tmp directories ... done

done

done

done

done

done

done

done

完成后重启GitLab服务

[root@node2 ~]# gitlab-ctl restart

ok: run: gitaly: (pid 18194) 0s

ok: run: gitlab-monitor: (pid 18204) 0s

ok: run: gitlab-workhorse: (pid 18209) 1s

ok: run: logrotate: (pid 18224) 0s

ok: run: nginx: (pid 18231) 1s

ok: run: node-exporter: (pid 18237) 0s

ok: run: postgres-exporter: (pid 18242) 0s

ok: run: postgresql: (pid 18314) 0s

ok: run: prometheus: (pid 18317) 1s

ok: run: redis: (pid 18326) 0s

ok: run: redis-exporter: (pid 18330) 0s

ok: run: sidekiq: (pid 18345) 0s

ok: run: unicorn: (pid 18354) 0s

检查GitLab的服务

[root@node2 ~]# gitlab-rake gitlab:check SANITIZE=true

Checking GitLab Shell ...

GitLab Shell version >= 5.9.4 ? ... OK (5.9.4)

Repo base directory exists?

default... yes

Repo storage directories are symlinks?

default... no

Repo paths owned by git:root, or git:git?

default... yes

Repo paths access is drwxrws---?

default... yes

hooks directories in repos are links: ...

2/3 ... ok

2/4 ... repository is empty

7/5 ... ok

7/7 ... ok

2/8 ... ok

Running /opt/gitlab/embedded/service/gitlab-shell/bin/check

Check GitLab API access: OK

Redis available via internal API: OK

Access to /var/opt/gitlab/.ssh/authorized\_keys: OK

gitlab-shell self-check successful

Checking GitLab Shell ... Finished

Checking Sidekiq ...

Running? ... yes

Number of Sidekiq processes ... 1

Checking Sidekiq ... Finished

Reply by email is disabled in config/gitlab.yml

Checking LDAP ...

LDAP is disabled in config/gitlab.yml

Checking LDAP ... Finished

Checking GitLab ...

Git configured correctly? ... yes

Database config exists? ... yes

All migrations up? ... yes

Database contains orphaned GroupMembers? ... no

GitLab config exists? ... yes

GitLab config up to date? ... yes

Log directory writable? ... yes

Tmp directory writable? ... yes

Uploads directory exists? ... yes

Uploads directory has correct permissions? ... yes

Uploads directory tmp has correct permissions? ... skipped (no tmp uploads folder yet)

Init script exists? ... skipped (omnibus-gitlab has no init script)

Init script up-to-date? ... skipped (omnibus-gitlab has no init script)

Projects have namespace: ...

2/3 ... yes

2/4 ... yes

7/5 ... yes

7/7 ... yes

2/8 ... yes

Redis version >= 2.8.0? ... yes

Ruby version >= 2.3.5 ? ... yes (2.3.5)

Git version >= 2.7.3 ? ... yes (2.13.6)

Git user has default SSH configuration? ... yes

Active users: ... 5

Checking GitLab ... Finished