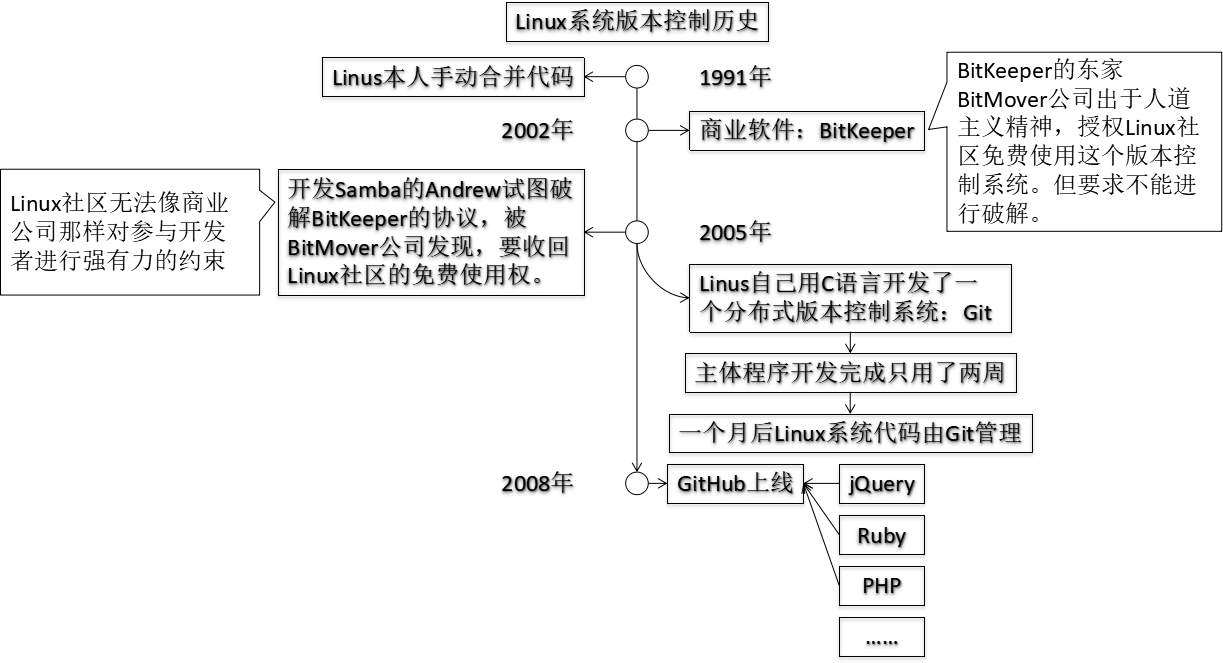
Git

# 一、Git的起源

## 1. Git简史



## 2. Git是什么

Git是目前世界上最先进的分布式版本控制系统（没有之一）

## 3. Git优势

* 速度
* 简单的设计
* 对非线性开发模式的强力支持（允许成千上万个并行开发的分支）
* 完全分布式
* 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

# 二、Git的安装

整体都是一键傻瓜式安装（有下一步点下一步，没有下一步选择第一个选项，再点击下一步，最后完成） ，这种默认选项就足够用了

## Windows

选择Windows安装包，安装即可。

也可以从Git官方网址下载最新的安装包：<https://git-scm.com/download/win>

但是Git官方网址下载很慢，所以有国内镜像Git安装包下载地址：<http://npm.taobao.org/mirrors/git-for-windows>

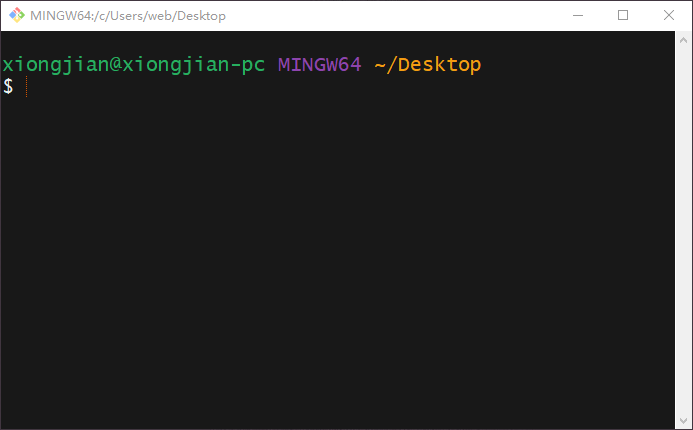
国内镜像拉到最下面找最新的安装包安装即可

## Mac

安装方式，详见官方网址：<https://git-scm.com/download/mac>

## 检查

以Windows为例：安装完成后，右键打开菜单栏找到“Git”->“Git Bash”，蹦出一个类似命令行窗口的东西，就说明Git安装成功！



# 三、Git的基本使用

## 1. 初次配置

第一次使用Git之前，我们需要进行一步配置，只需要配置一次。

因为Git是分布式版本控制系统，所以，每个机器都必须自报家门。我们需要配置：你的名字和Email地址

Git指令：

* git config --global user.name "Your Name" 配置git用户名
* git config --global user.email "email@example.com" 配置git用户邮箱
* git config user.name 查看配置的姓名
* git config user.email 查看配置的邮箱

## 2. 理解Git三区

将来我们存储在Git的文件一共有三种状态，分别位于三个不同的区域内：

* 工作区（Working Directory）：表示修改了文件，但还没保存到数据库中。
* 暂存区（Staging Area）：表示对一个已修改文件的当前版本做了标记，使之包含在下次提交的快照中。
* 版本区（Repository）：表示数据已经安全地保存在本地数据库中。

基本的 Git 工作流程如下：

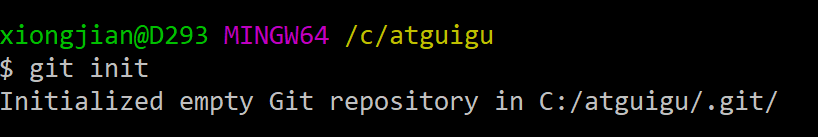
* 在工作区中新增或修改文件。（此时文件位于工作区）
* 你想要下次提交的更改选择性地暂存，这样只会将更改的部分添加到暂存区。（此时文件位于暂存区）
* 提交更新，找到暂存区的文件，将快照永久性存储到 Git 仓库。（此时文件位于版本区）

## 3. 基本指令

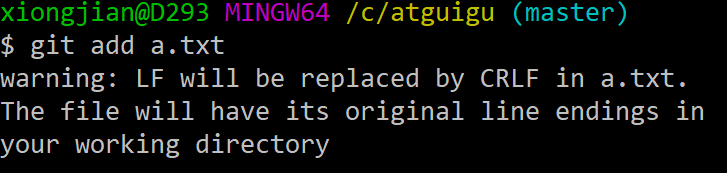
* git help 查看所有Git指令的帮助
  + 如要查看具体某个Git指令的帮助：git help xxx
* git –version 查看当前Git的版本

## 基本工作流程指令

* git init 将当前目录变成一个Git仓库

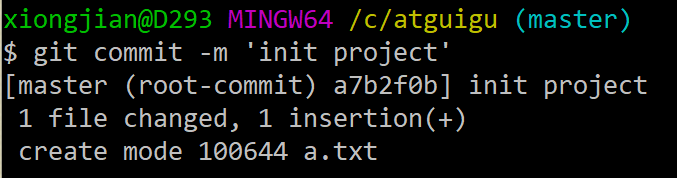


* + 注意：只能初始化一次
* git add 将新增或修改的文件（工作区文件）到暂存区中暂存起来
  + git add 某个文件 将某个文件添加到暂存区中暂存起来
  + git add [. | \* | -A] 将所有文件添加到暂存区中暂存起来

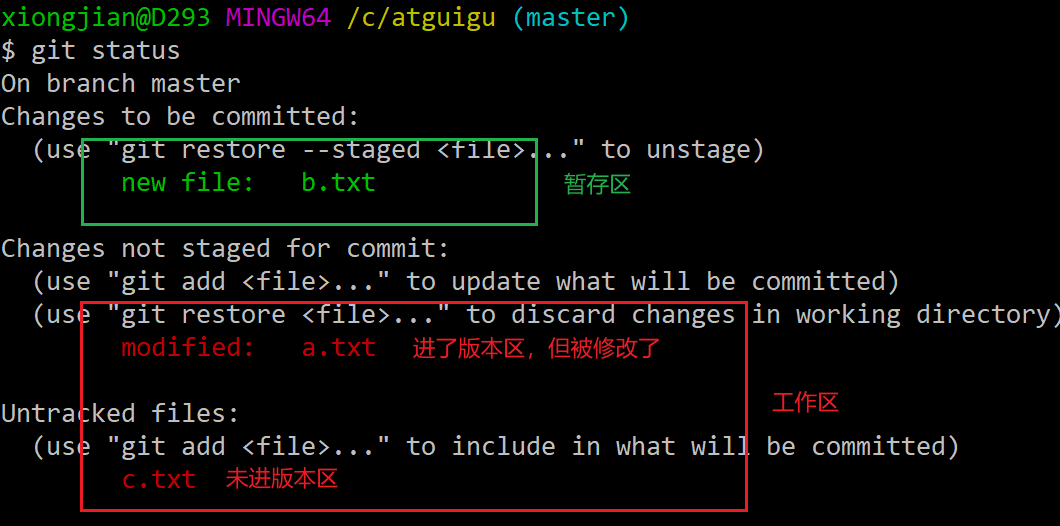


此时第一次会出现一个警告，原因：由于Linux(LF)和Windows(CRLF)的换行符不一致导致的，我们关闭掉它就好了：git config --global core.autocrlf false

* git commit –m "commit message" 将暂存区所有文件提交到版本区



* + 提交注释写错了，怎么办？ git commit –amend 修改提交的注释内容
* git status 查看文件的状态（位于哪个区）



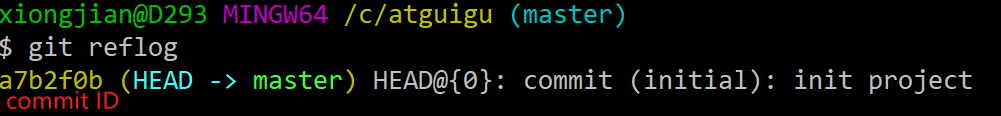
* + 红色文件位于工作区
  + 绿色文件位于暂存区
  + 没有显示的文件位于版本区

## 查看版本历史记录

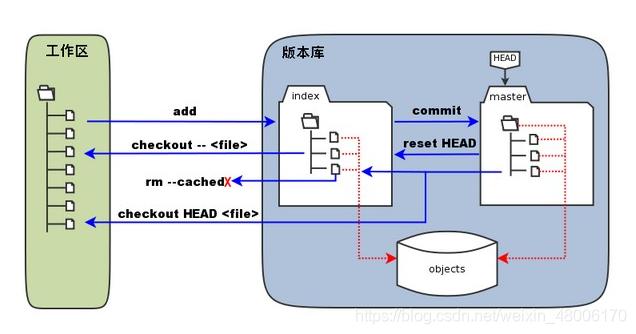
* git log 查看详细版本历史记录



* git reflog查看简单版本历史记录



## 版本回退



版本回退主要分为以下三个任务：

* 暂存区代码回退到工作区，覆盖工作区的内容
* 版本区代码仅回退到暂存区，覆盖暂存区内容，工作区内容不变
* 版本区代码即回退到工作区，也回退暂存区，此时会覆盖工作区和暂存区的内容

### 暂存区 –> 工作区

* git checkout – 文件名
* 需求：如果暂存区所有文件都需要回退到工作区呢?
* git checkout – \*

### 版本区 –> 暂存区

* git reset HEAD
* git reset --mixed

### 版本区 –> 暂存区 –> 工作区

* git checkout HEAD 文件名
* git reset –hard HEAD^

### reset指令详解

* --soft 参数：仅回退版本区代码
  + git reset –soft 版本区回退一个版本
  + git reset –soft HEAD^ 版本区回退一个版本
  + git reset –soft HEAD^^ 版本区回退两个版本（以此类推，可以叠加n个^）
  + git reset –soft HEAD~n 版本区回退n个版本
  + git reset –soft commitID 版本区回退到commitID的版本
* --mixed参数：回退版本区和暂存区代码
  + git reset --mixed
* --hard参数：回退版本区、暂存区和工作区代码
  + git reset –hard 三区都回退一个版本
  + git reset –hard HEAD^ 三区都回退一个版本
  + git reset –hard HEAD^^ 三区都回退两个版本（以此类推，可以叠加n个^）
  + git reset –hard HEAD~n 三区都回退n个版本
  + git reset –hard commitID 三区都回退到commitID的版本

# 四、Git的分支管理

## 1. 分支简介

Git 保存文件，保存的不是文件的变化或者差异，保存的是一系列不同时刻的**快照。**

Git 会保存一个提交对象（commit object）。 知道了 Git 保存数据的方式，我们可以很自然的想到——该提交对象会包含一个指向暂存内容快照的指针。 但不仅仅是这样，该提交对象还包含了作者的姓名和邮箱、提交时输入的信息以及指向它的父对象的指针。 首次提交产生的提交对象没有父对象，普通提交操作产生的提交对象有一个父对象， 而由多个分支合并产生的提交对象有多个父对象。

Git 的分支，其实本质上仅仅是**指向提交对象的可变指针**。 Git 的默认分支名字是 master。 在多次提交操作之后，你其实已经有一个指向最后那个提交对象的 master 分支。 master 分支会在每次提交时自动向前移动。

* Git 的 master 分支并不是一个特殊分支。 它就跟其它分支完全没有区别。 之所以几乎每一个仓库都有 master 分支，是因为 git init 命令默认创建它，并且大多数人都懒得去改动它。

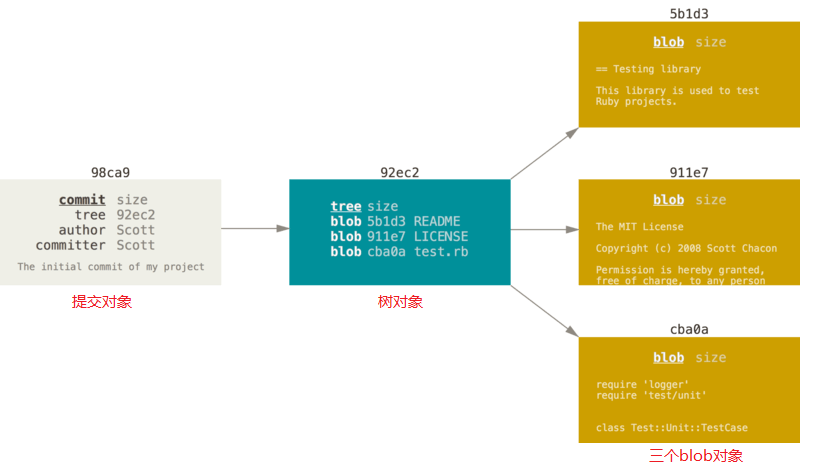
为了更加形象地说明，我们假设有一个工作目录，里面有三个文件：README test.rb LICENSE

首先我们将文件首次添加到版本区中进行版本控制：

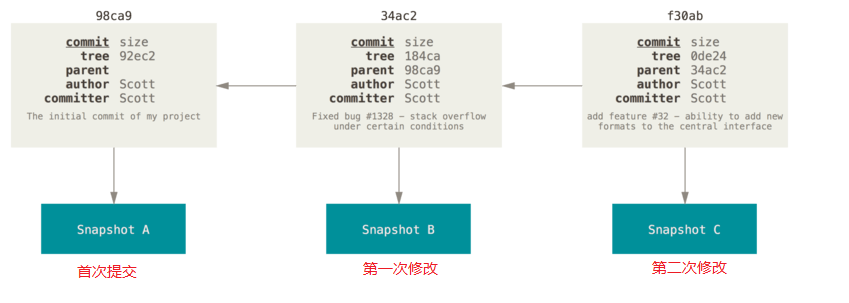
* git add README test.rb LICENSE
* git commit -m 'The initial commit of my project'

当使用 git commit 进行提交操作时，Git 会先计算每一个子目录（本例中只有项目根目录）的校验和， 然后在 Git 仓库中这些校验和保存为树对象。随后，Git 便会创建一个提交对象， 它除了包含上面提到的那些信息外，还包含指向这个树对象（项目根目录）的指针。 如此一来，Git 就可以在需要的时候重现此次保存的快照。

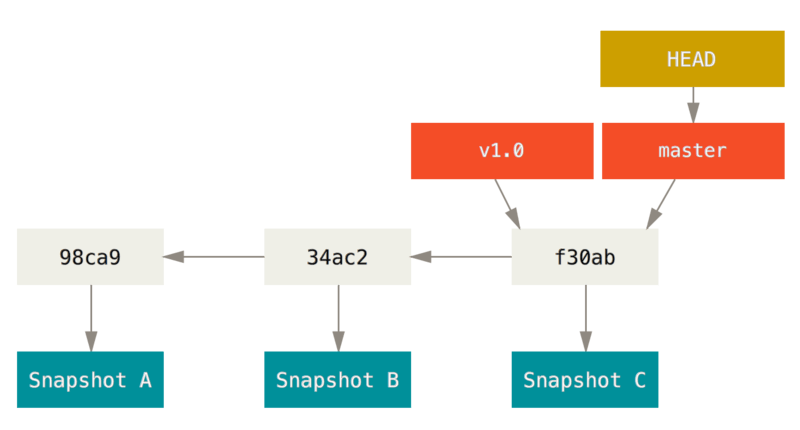
现在，Git 仓库中有五个对象：三个 blob 对象（保存着不同文件快照）、一个 树 对象 （记录着目录结构和 blob 对象索引）以及一个 提交 对象（包含着指向前述树对象的指针和所有提交信息），如下图所示：



对文件做些修改后再次提交，那么这次产生的提交对象会包含一个指向上次提交对象（父对象）的指针。



Git 的分支，其实本质上仅仅是**指向提交对象的可变指针。**默认叫master



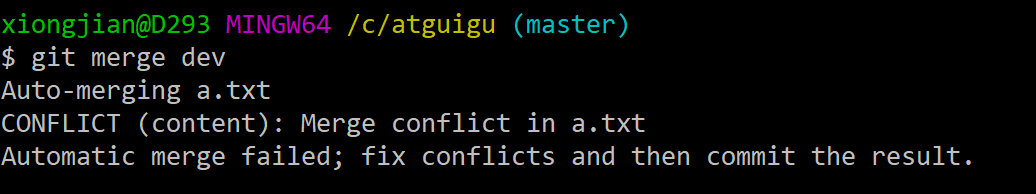
## 2. 分支操作

* git branch 查看分支
* git branch <branch name> 新建分支
* git checkout <branch name> 切换分支
* git checkout -b <branch name> 新建并切换分支
* git branch -D <branch name> 删除分支
* git merge <branch name> 当前分支合并其他分支

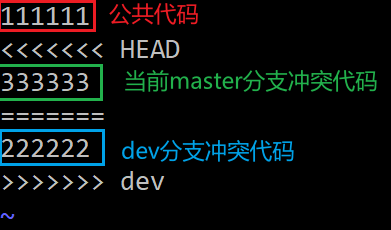
## 3. 合并冲突

### 3.1 原因

如果你在两个不同的分支中，对同一个文件的同一个部分进行了不同的修改，在合并它们的时候就会产生合并冲突。

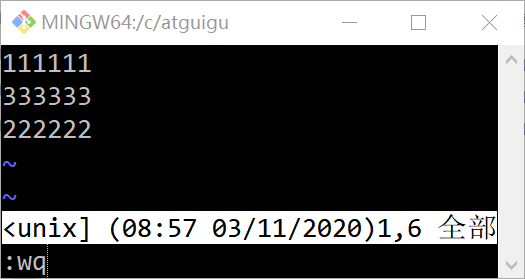


### 3.2 冲突文件



### 3.3 解决冲突

直接去文件中删除<<<<<<< , ======= , 和 >>>>>>>符号，保留需要保留的代码。



# 五、Git的高级操作

## 差异对比

对比不同区域或不同分支的文件差异

* git diff : 比较工作区与暂存区文件差异
* git diff --cached : 比较暂存区与版本区文件差异
* git diff HEAD: 比较工作区与版本区文件差异
* git diff <commit ID> <commit ID> 比较两个commit之间内容差异
* git diff <branch name> <branch name> 比较两个branch之间内容差异
  + 后面加上--stat参数，只会显示出所有有差异的文件名称
  + 后面加上具体文件名称，只会显示这个具体文件的差异

## 文件暂存

把工作区内容缓存到一个栈里，之后可以取出。在未提交工作区内容，但是想切到其他分支时非常有用。

* git stash save "message" 添加改动到stash
* git stash drop <stash@{ID}> 删除单个stash
* git stash pop <stash@{ID}> 恢复某个stash改动
* git stash list 查看stash列表
* git stash clear 删除全部stash

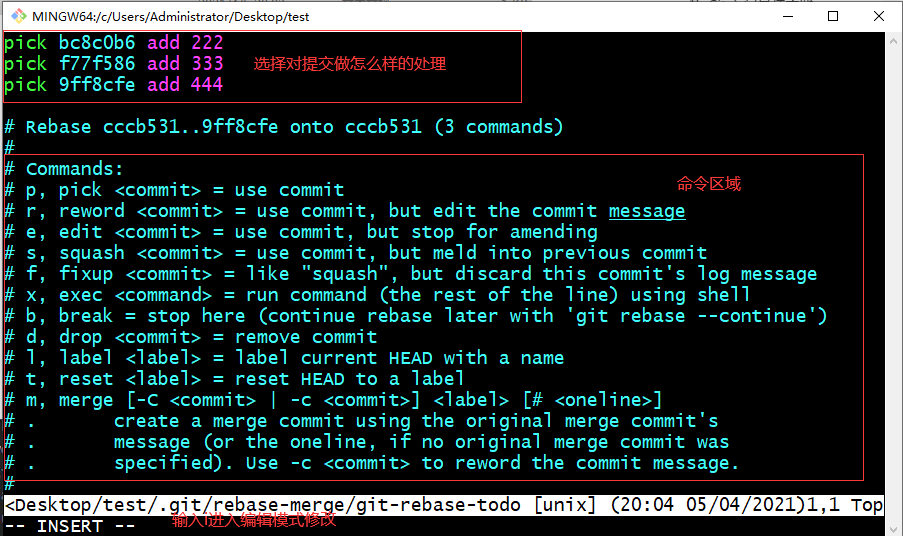
## Rebase

### 合并多个提交

将一个分支上多个commit提交内容合并为一个，让整体Git提交信息更加简介

* git rebase –i [startpoint] [endpoint]
  + startpoint 代表开始的提交
  + endpoint代表结束的提交

输入指令完后界面显示如下：

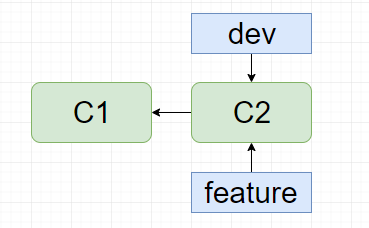


其中命令区域的命令含义为：

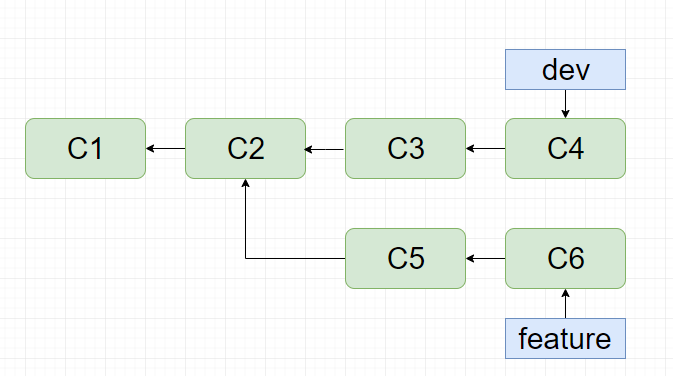
* pick：保留该commit（缩写:p）
* reword：保留该commit，但我需要修改该commit的注释（缩写:r）
* edit：保留该commit, 但我要停下来修改该提交(不仅仅修改注释)（缩写:e）
* squash：将该commit和前一个commit合并（缩写:s）
* fixup：将该commit和前一个commit合并，但我不要保留该提交的注释信息（缩写:f）
* exec：执行shell命令（缩写:x）
* drop：我要丢弃该commit（缩写:d）

### 合并两个分支

假设Git仓库已经有两个提交：C1和C2，同时也有两个分支：dev和feature，它们内容都是一样的



dev分支开发提交了两次：C3和C4，feature分支开发提交了两次：C5和C6

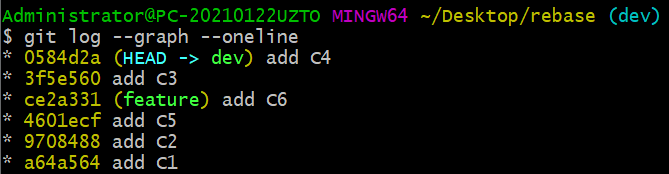


来到dev分支，使用git merge feature，然后通过 git log --graph --oneline 查看提交图形



能看到此处有两条线

来到dev分支，使用git rebase feature，然后通过 git log --graph –oneline 查看提交图形



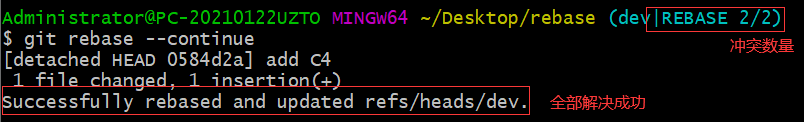
能看到此处只有一条线，**更加直观，有利于回滚代码**

### 解决冲突

使用rebase合并出现冲突要一个一个分别解决

解决办法：

* 编辑冲突文件，解决冲突
* git add .
* git rebase –continue 继续下一个冲突
* 此时会弹出一个界面，让你确认当前提交注释，确认即可
* 一直循环到成功为止

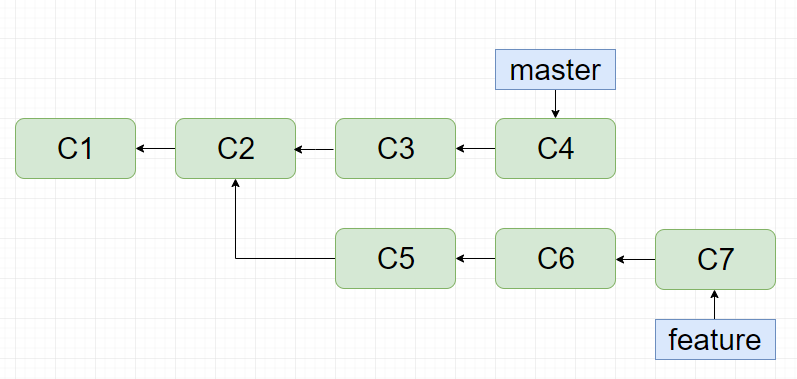


## Cherry Pick

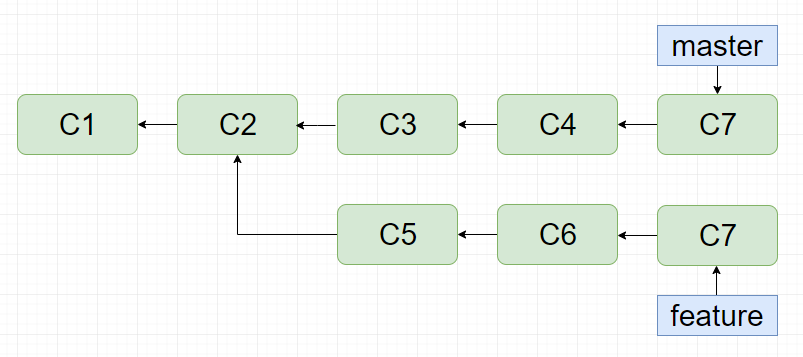
对于多分支的代码库，将代码从一个分支转移到另一个分支是常见需求。

这时分两种情况。一种情况是，你需要另一个分支的所有代码变动，那么就采用合并（git merge）。另一种情况是，你只需要部分代码变动（某几个提交），这时可以采用 Cherry Pick。

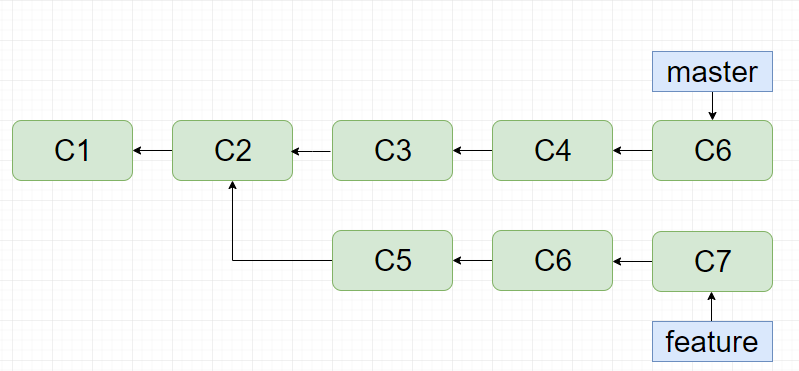
比如：代码仓库有master和feature两个分支。



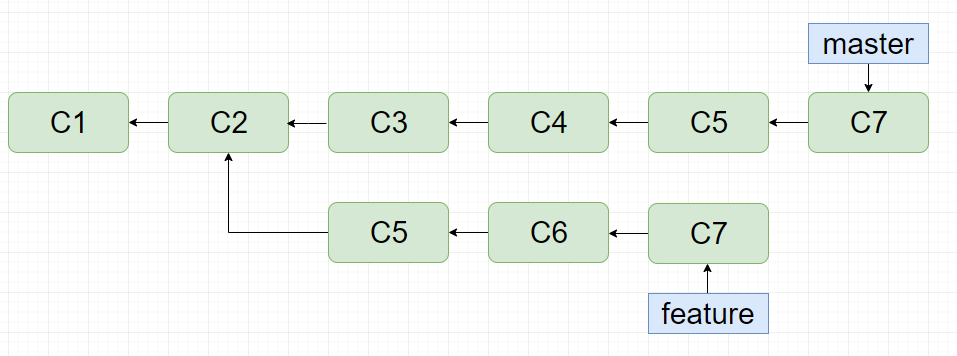
* 首先，切换到master分支：git checkout master
* 此时不同的做法结果不一样：
  + git cherry-pick feature 拉取feature 分支最近的提交到master分支来



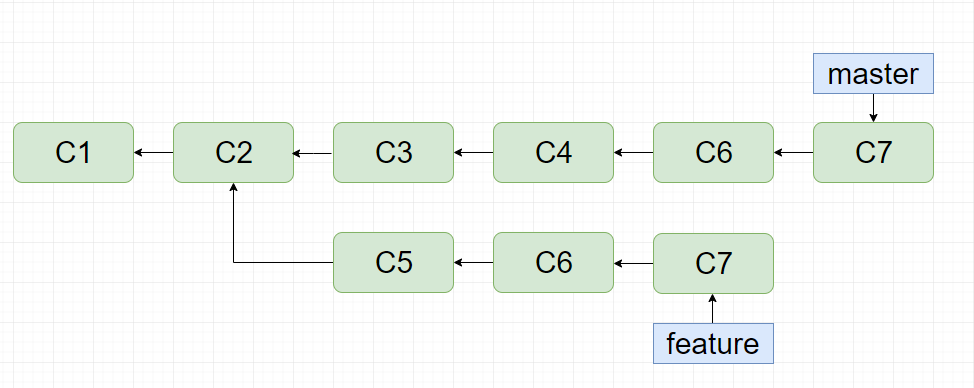
* + git cherry-pick C6拉取feature 分支C6提交到master分支来



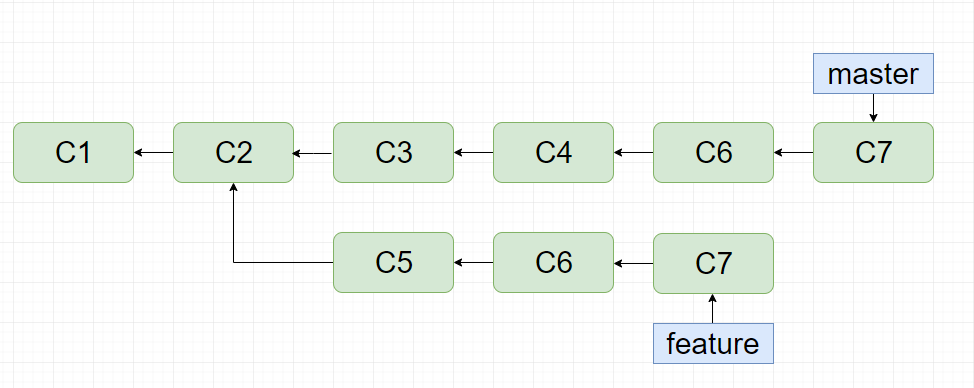
* + git cherry-pick C5 C7拉取feature 分支C5和C7提交到master分支来



* + git cherry-pick C5..C7拉取feature 分支C5(不包含)到C7提交到master分支来



* + git cherry-pick C6^..C7拉取feature 分支C6(包含)到C7提交到master分支来



* 解决冲突（一个一个分别解决）
  + 修改冲突文件
  + git add .
  + git cherry-pick –continue
  + 输入注释信息，完成

# 六、远程仓库

## 1. Github

### 1.1 是什么

GitHub是一个Git项目托管网站，也是全球最大开源项目托管网站。

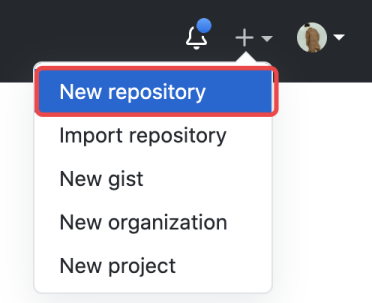
### 1.2 网址

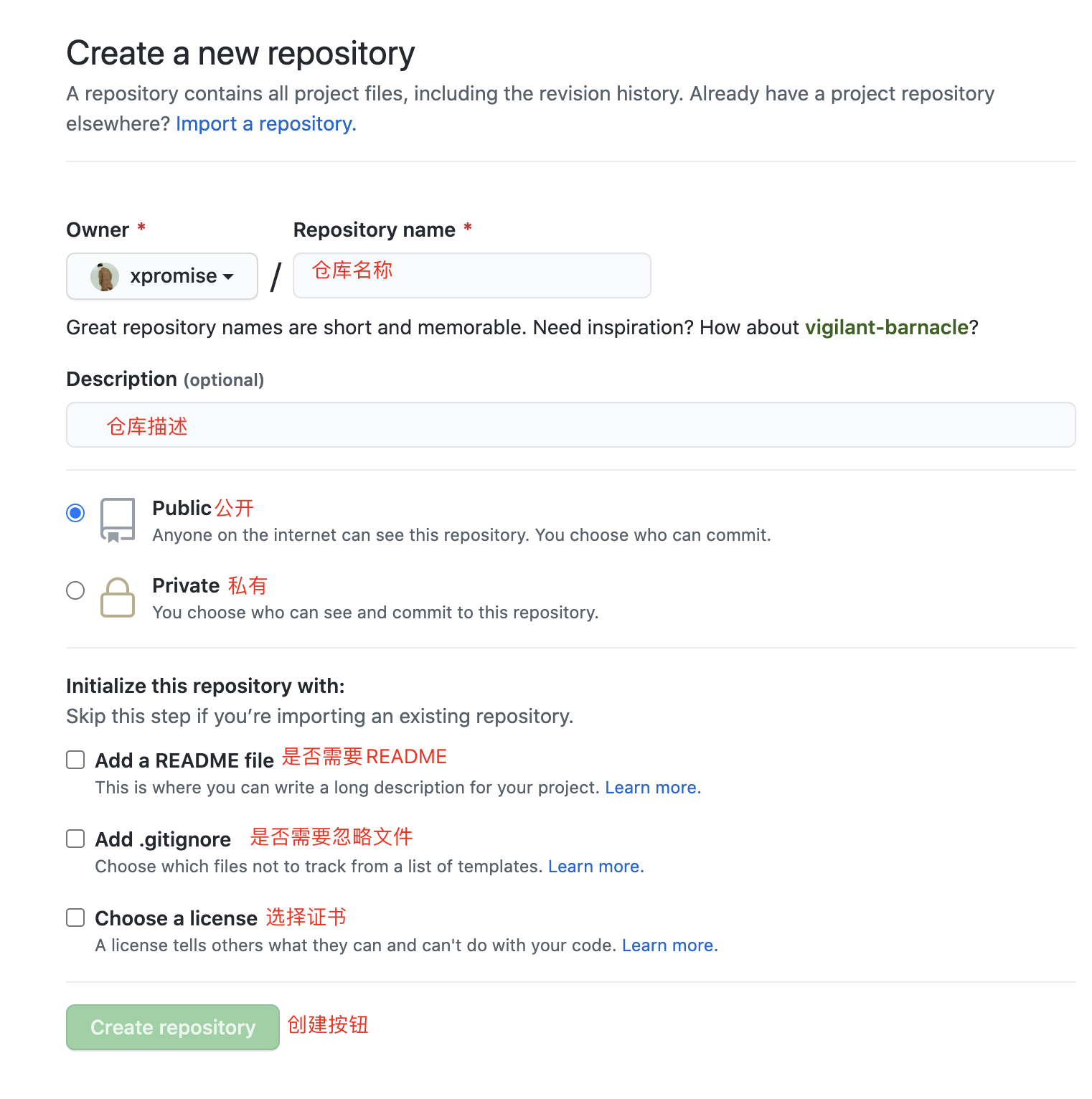
<https://github.com/>

### 1.3 使用

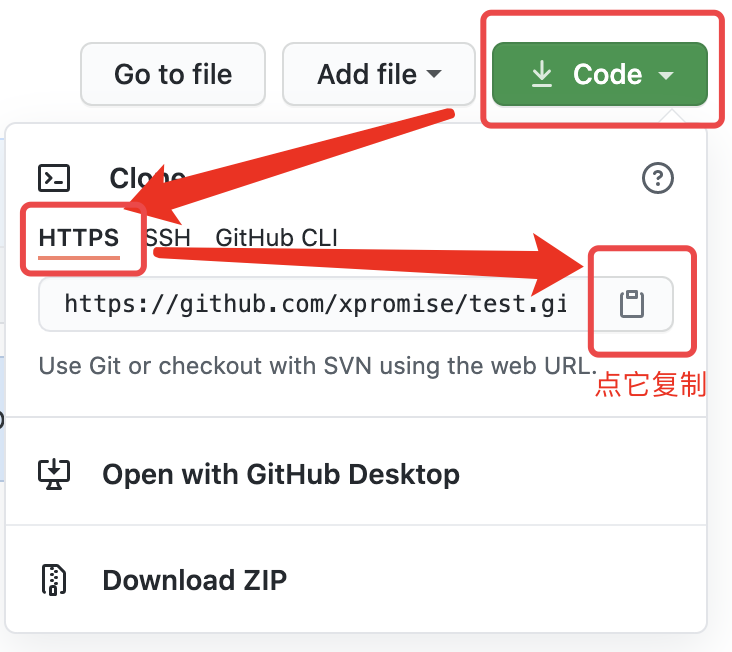
#### 将我们本地代码托管到Github上去

* 在Github上创建一个仓库



* 本地仓库进行Git版本控制
  + git add .
  + git commit –m "commit message"
* 将远程仓库和本地仓库关联起来
  + 关联Github仓库地址



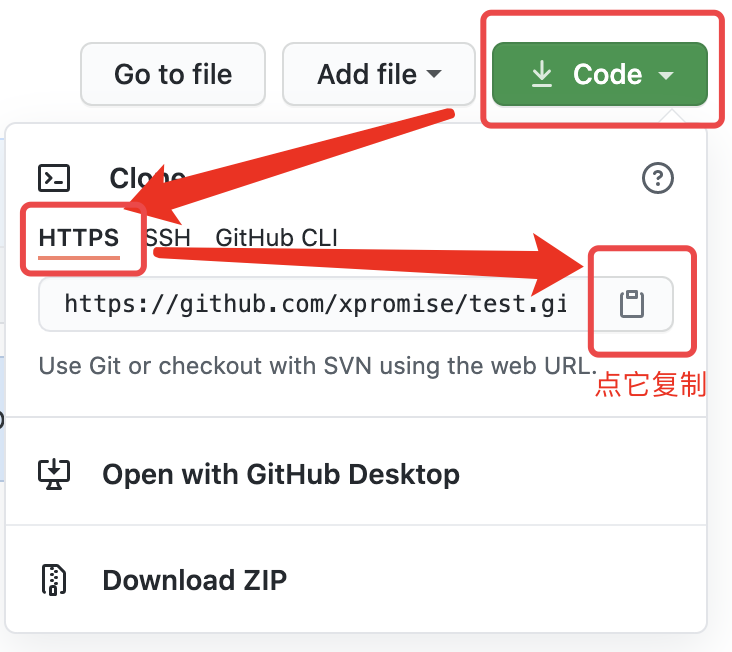
此处有两种地址：https和ssh

默认https就可以直接用，而ssh需要配置，后面章节会讲，此处先用https的地址

* + 关联命令：git remote add origin [远程仓库地址]
* 将本地仓库代码推送到远程仓库保管
  + git push –u origin [分支名称]
  + 首次加上-u参数，后续可以不用，加上后将来推送分支可以省略为git push

#### 将Github上的开源项目拉取到本地进行开发

* 找到开源项目的地址



* 将其克隆下来
  + git clone [远程仓库地址]

#### 基于前两个，开发完成又如何交互

* 本地代码修改好了，要推送到远程仓库保管
  + git push origin [分支名称]
* 远程开源项目更新了，要看最新代码
  + git pull origin [分支名称]、

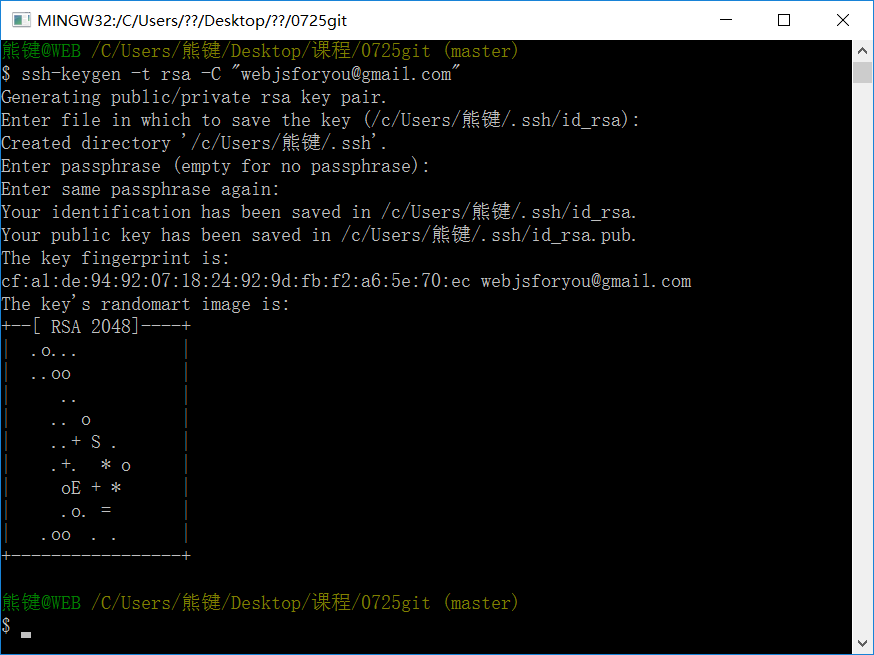
#### 拉取远程仓库的dev分支

* 克隆仓库：git clone xxx
  + 此时拉取下来的是master分支
* 拉取dev分支：git fetch origin dev1:dev2
  + 拉取远程仓库dev1分支到本地的dev2分支来
  + 注意：实际写的时候都是dev，不改名称
* 切换到dev分支：git checkout dev

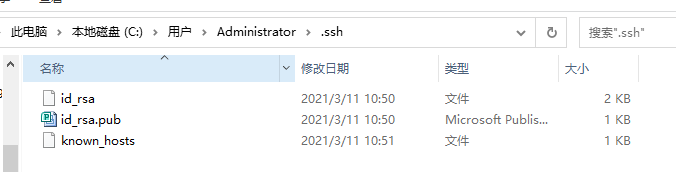
#### 配置SSH

默认情况下，我们只能使用HTTPS的地址进行操作远程仓库，如果想要使用ssh必须进行一步配置，也只需要配置一次即可。

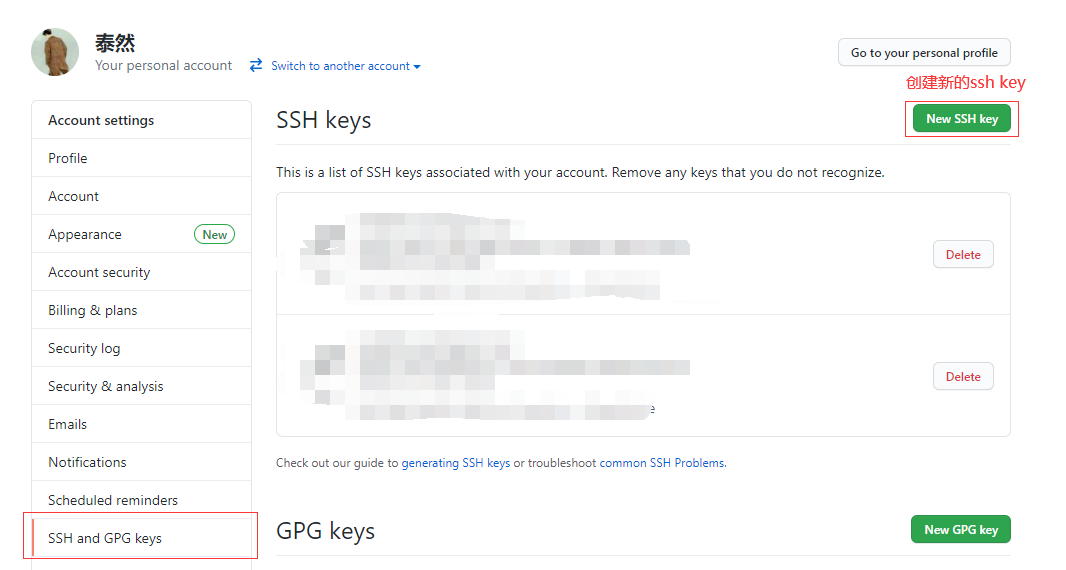
* 首先，本地生成SSH公钥
  + 打开CMD窗口，输入ssh-keygen -t rsa -C "您的邮箱地址"



* + 按三次回车，您的SSH KEY就生成好了
* 来到 ~/.ssh 目录下，打开id\_rsa.pub 文件，复制里面所有的内容



* 打开网址：<https://github.com/settings/keys>



* 新增SSH KEY



操作完成，可以试试使用SSH的方式来操作仓库了~

## 2. Gitee

### 1.1 是什么

Gitee.com(码云) 是 OSCHINA.NET 推出的代码托管平台,支持 Git 和 SVN,提供免费的私有仓库托管。

其实就是中国版Github，相对于Github而言，访问速度非常流畅。

### 1.2 网址

<https://gitee.com/>

### 1.3 使用

与Github操作基本一致

## 3. Gitlab

### 3.1 是什么

GitLab 是一个用于仓库管理系统的开源项目，使用Git作为代码管理工具，并在此基础上搭建起来的Web服务

### 3.2 优势

* 提供本地私有化部署，这样项目就不用开源出去，而是部署在自己服务器上
* 简单理解：就是使用Gitlab搭建一个类似于Github私有平台在自己服务器上，但是功能比Github更加强大

### 3.3 安装

#### 免安装，直接网址访问

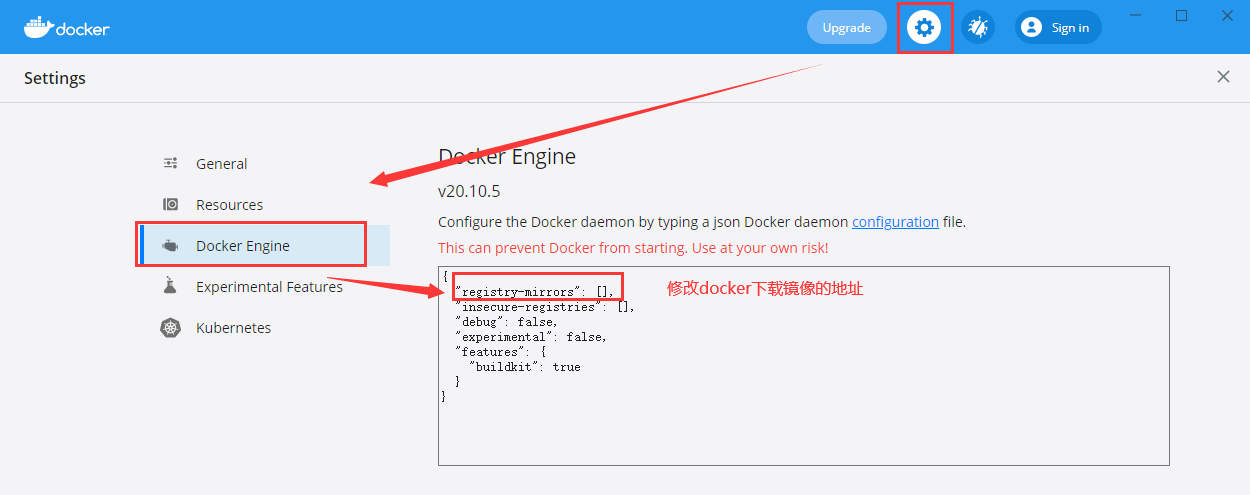
* 访问网址：<https://gitlab.com/dashboard/projects>
* 可以使用github登录，免费体验30天
* 问题：国内访问速度很慢

#### 使用Docker安装Gitlab到本地

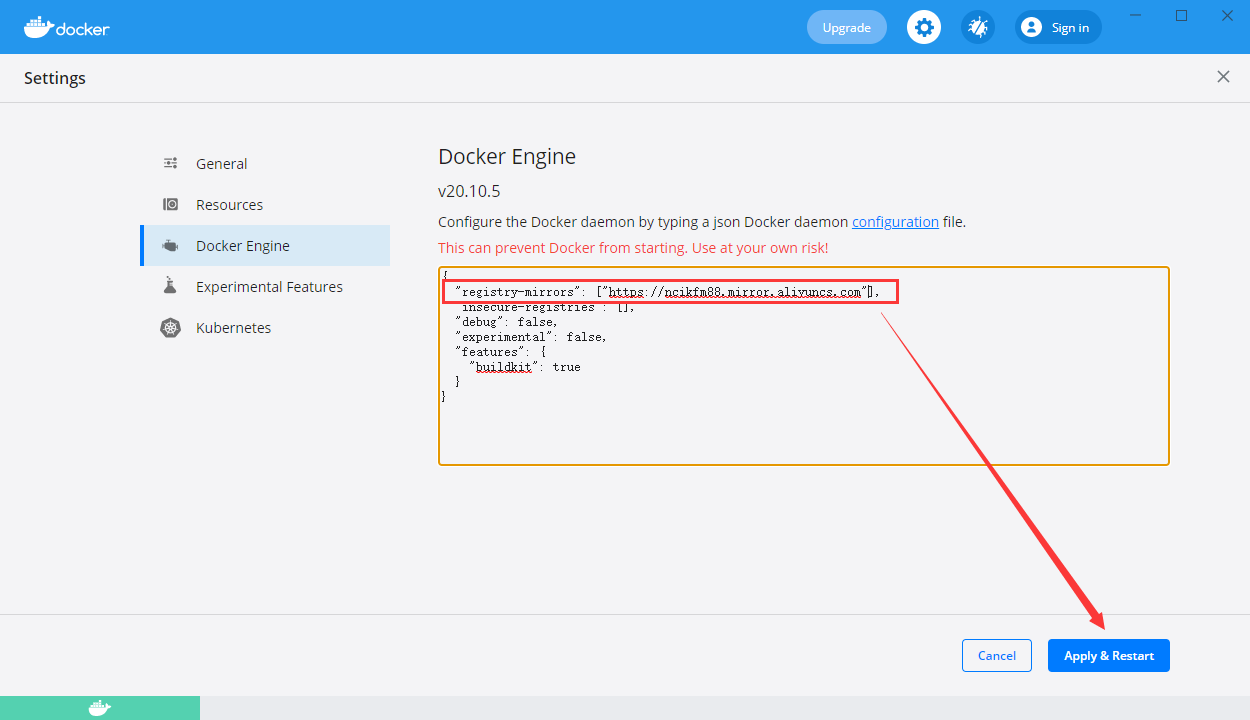
* Docker介绍
  + Docker 是一个开源的应用容器引擎，让开发者可以打包他们的应用以及依赖包到一个可移植的镜像中，然后发布到任何流行的 Linux或Windows 机器上，也可以实现虚拟化。容器是完全使用沙箱机制，相互之间不会有任何接口
  + 比如：我要运行一个JAVA服务，我需要安装JAVA、MYSQL、REDIS等一大堆软件才能运行起来，一个个下载配置特别麻烦。docker提供容器（可以看作运行代码的环境），在容器中下载镜像（各种需要安装的软件包），下载完成即配置完成了，此时就可以在容器中运行我们的JAVA服务。
* 安装Docker
  + 安装地址：[https://docs.docker.com/get-started/#download-and-install-docker](https://docs.docker.com/get-started/" \l "download-and-install-docker)
  + 我们安装Windows版本的Docker
  + 此时也安装好了Docker Compose
  + 安装完成初次打开可能会有一个警告提示



* + 安装wsl\_update\_x64.msi 即可解决
* 修改Docker下载镜像的地址
  + 默认地址是国外服务器，太卡了



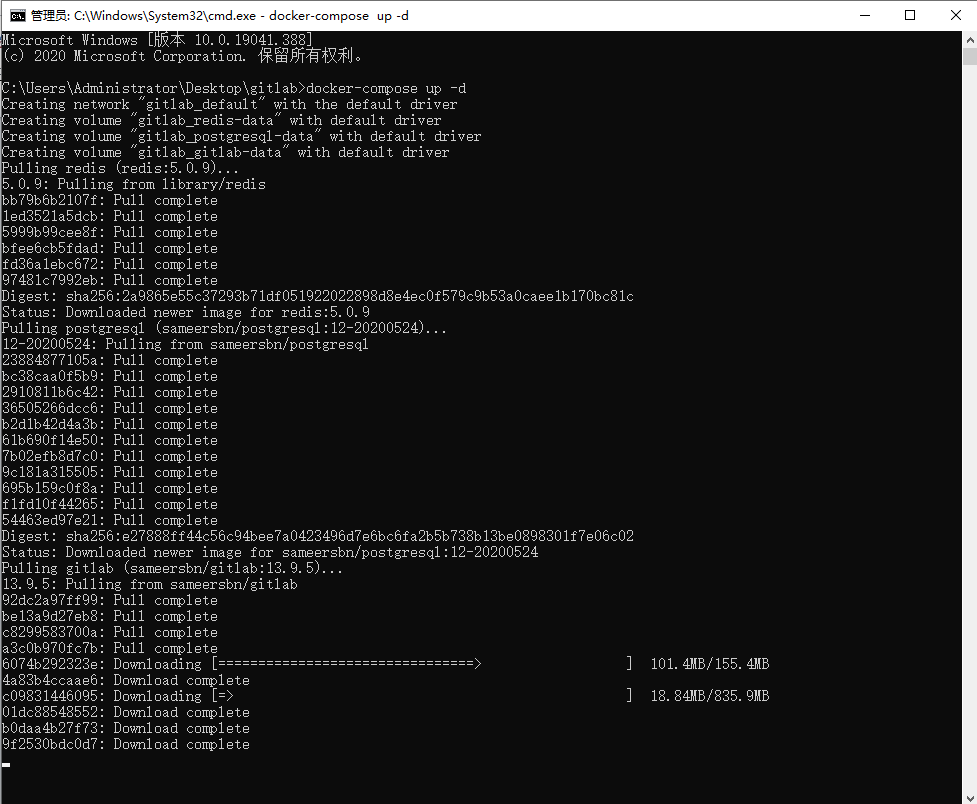
* 国内镜像地址
  + Docker官方中国区：https://registry.docker-cn.com
  + 网易：http://hub-mirror.c.163.com
  + 中国科学技术大学：https://docker.mirrors.ustc.edu.cn
  + 阿里云：
    - 进入阿里云镜像加速页面（需要登录）：<https://cr.console.aliyun.com/undefined/instances/mirrors>
    - 获取如下加速器地址：<https://XXXXXX.mirror.aliyuncs.com>
  + 选择其一即可

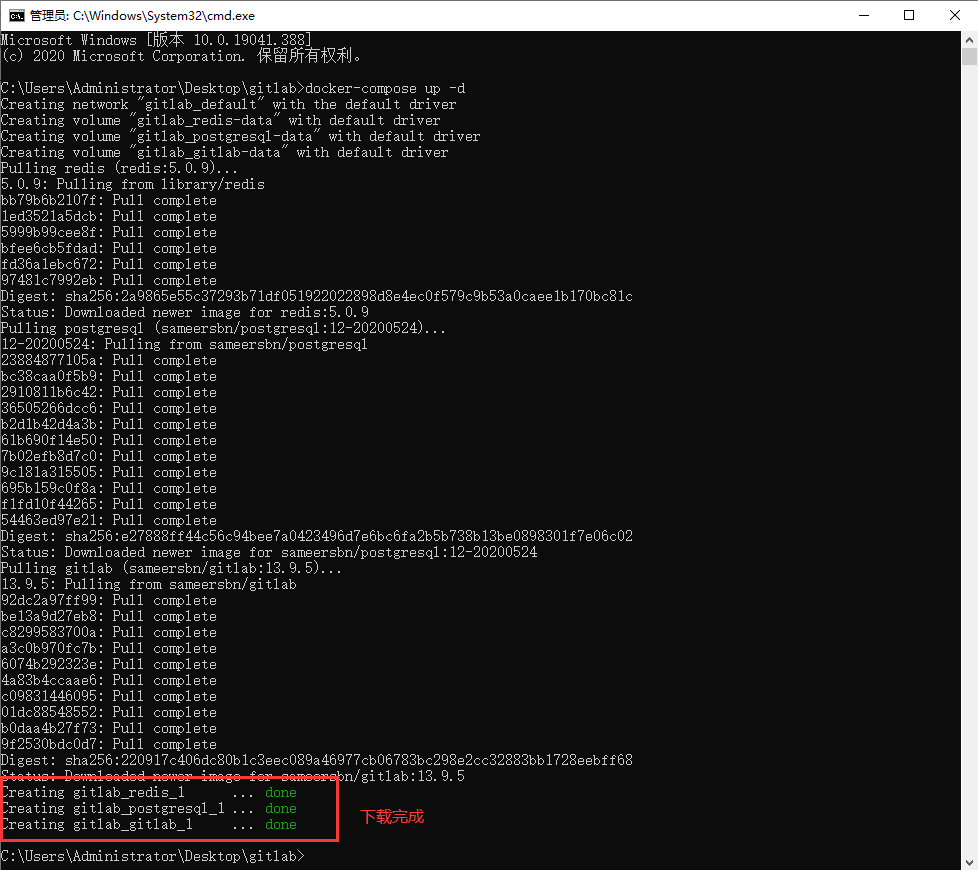


* 创建 Docker Compose 配置文件 docker-compose.yml
  + 配置文件地址： <https://github.com/sameersbn/docker-gitlab/blob/master/docker-compose.yml>
  + 我们需要稍作修改：添加管理员账户邮箱和密码

|  |
| --- |
| version: '2.3'  services:  redis:  restart: always  image: redis:5.0.9  command:  - --loglevel warning  volumes:  - redis-data:/var/lib/redis:Z  postgresql:  restart: always  image: sameersbn/postgresql:12-20200524  volumes:  - postgresql-data:/var/lib/postgresql:Z  environment:  - DB\_USER=gitlab  - DB\_PASS=password  - DB\_NAME=gitlabhq\_production  - DB\_EXTENSION=pg\_trgm,btree\_gist  gitlab:  restart: always  image: sameersbn/gitlab:13.9.5  depends\_on:  - redis  - postgresql  ports:  - "10080:80"  - "10022:22"  volumes:  - gitlab-data:/home/git/data:Z  healthcheck:  test: ["CMD", "/usr/local/sbin/healthcheck"]  interval: 5m  timeout: 10s  retries: 3  start\_period: 5m  environment:  - DEBUG=false  - DB\_ADAPTER=postgresql  - DB\_HOST=postgresql  - DB\_PORT=5432  - DB\_USER=gitlab  - DB\_PASS=password  - DB\_NAME=gitlabhq\_production  - REDIS\_HOST=redis  - REDIS\_PORT=6379  - TZ=Asia/Kolkata  - GITLAB\_TIMEZONE=Kolkata  - GITLAB\_HTTPS=false  - SSL\_SELF\_SIGNED=false  - GITLAB\_HOST=localhost  - GITLAB\_PORT=10080  - GITLAB\_SSH\_PORT=10022  - GITLAB\_RELATIVE\_URL\_ROOT=  - GITLAB\_SECRETS\_DB\_KEY\_BASE=long-and-random-alphanumeric-string  - GITLAB\_SECRETS\_SECRET\_KEY\_BASE=long-and-random-alphanumeric-string  - GITLAB\_SECRETS\_OTP\_KEY\_BASE=long-and-random-alphanumeric-string  - GITLAB\_ROOT\_PASSWORD=12345678  - GITLAB\_ROOT\_EMAIL=xpromise@atguigu.com  - GITLAB\_NOTIFY\_ON\_BROKEN\_BUILDS=true  - GITLAB\_NOTIFY\_PUSHER=false  - GITLAB\_EMAIL=notifications@example.com  - GITLAB\_EMAIL\_REPLY\_TO=noreply@example.com  - GITLAB\_INCOMING\_EMAIL\_ADDRESS=reply@example.com  - GITLAB\_BACKUP\_SCHEDULE=daily  - GITLAB\_BACKUP\_TIME=01:00  - SMTP\_ENABLED=false  - SMTP\_DOMAIN=www.example.com  - SMTP\_HOST=smtp.gmail.com  - SMTP\_PORT=587  - SMTP\_USER=mailer@example.com  - SMTP\_PASS=password  - SMTP\_STARTTLS=true  - SMTP\_AUTHENTICATION=login  - IMAP\_ENABLED=false  - IMAP\_HOST=imap.gmail.com  - IMAP\_PORT=993  - IMAP\_USER=mailer@example.com  - IMAP\_PASS=password  - IMAP\_SSL=true  - IMAP\_STARTTLS=false  - OAUTH\_ENABLED=false  - OAUTH\_AUTO\_SIGN\_IN\_WITH\_PROVIDER=  - OAUTH\_ALLOW\_SSO=  - OAUTH\_BLOCK\_AUTO\_CREATED\_USERS=true  - OAUTH\_AUTO\_LINK\_LDAP\_USER=false  - OAUTH\_AUTO\_LINK\_SAML\_USER=false  - OAUTH\_EXTERNAL\_PROVIDERS=  - OAUTH\_CAS3\_LABEL=cas3  - OAUTH\_CAS3\_SERVER=  - OAUTH\_CAS3\_DISABLE\_SSL\_VERIFICATION=false  - OAUTH\_CAS3\_LOGIN\_URL=/cas/login  - OAUTH\_CAS3\_VALIDATE\_URL=/cas/p3/serviceValidate  - OAUTH\_CAS3\_LOGOUT\_URL=/cas/logout  - OAUTH\_GOOGLE\_API\_KEY=  - OAUTH\_GOOGLE\_APP\_SECRET=  - OAUTH\_GOOGLE\_RESTRICT\_DOMAIN=  - OAUTH\_FACEBOOK\_API\_KEY=  - OAUTH\_FACEBOOK\_APP\_SECRET=  - OAUTH\_TWITTER\_API\_KEY=  - OAUTH\_TWITTER\_APP\_SECRET=  - OAUTH\_GITHUB\_API\_KEY=  - OAUTH\_GITHUB\_APP\_SECRET=  - OAUTH\_GITHUB\_URL=  - OAUTH\_GITHUB\_VERIFY\_SSL=  - OAUTH\_GITLAB\_API\_KEY=  - OAUTH\_GITLAB\_APP\_SECRET=  - OAUTH\_BITBUCKET\_API\_KEY=  - OAUTH\_BITBUCKET\_APP\_SECRET=  - OAUTH\_BITBUCKET\_URL=  - OAUTH\_SAML\_ASSERTION\_CONSUMER\_SERVICE\_URL=  - OAUTH\_SAML\_IDP\_CERT\_FINGERPRINT=  - OAUTH\_SAML\_IDP\_SSO\_TARGET\_URL=  - OAUTH\_SAML\_ISSUER=  - OAUTH\_SAML\_LABEL="Our SAML Provider"  - OAUTH\_SAML\_NAME\_IDENTIFIER\_FORMAT=urn:oasis:names:tc:SAML:2.0:nameid-format:transient  - OAUTH\_SAML\_GROUPS\_ATTRIBUTE=  - OAUTH\_SAML\_EXTERNAL\_GROUPS=  - OAUTH\_SAML\_ATTRIBUTE\_STATEMENTS\_EMAIL=  - OAUTH\_SAML\_ATTRIBUTE\_STATEMENTS\_NAME=  - OAUTH\_SAML\_ATTRIBUTE\_STATEMENTS\_USERNAME=  - OAUTH\_SAML\_ATTRIBUTE\_STATEMENTS\_FIRST\_NAME=  - OAUTH\_SAML\_ATTRIBUTE\_STATEMENTS\_LAST\_NAME=  - OAUTH\_CROWD\_SERVER\_URL=  - OAUTH\_CROWD\_APP\_NAME=  - OAUTH\_CROWD\_APP\_PASSWORD=  - OAUTH\_AUTH0\_CLIENT\_ID=  - OAUTH\_AUTH0\_CLIENT\_SECRET=  - OAUTH\_AUTH0\_DOMAIN=  - OAUTH\_AUTH0\_SCOPE=  - OAUTH\_AZURE\_API\_KEY=  - OAUTH\_AZURE\_API\_SECRET=  - OAUTH\_AZURE\_TENANT\_ID=  volumes:  redis-data:  postgresql-data:  gitlab-data: |

* 使用 Docker Compose 来安装Gitlab
  + 来到docker-compose.yml文件目录，打开CMD
  + 输入命令：docker-compose up –d
  + 注意：安装过程有1GB多内容，所以请使用较好的网速安装



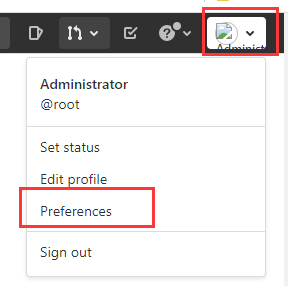


* 安装成功, 打开网址<http://localhost:10080/> 访问Gitlab
* 问题：所有镜像都安装到C盘，C盘太大了
* 解决：将Docker安装的镜像从C盘移到E盘
  + 在E盘创建一个文件夹：docker/wsl/data
  + 退出Docker应用程序
  + 打开管理员身份的CMD
  + 首先，将Docker镜像数据进行备份，命令：wsl --export docker-desktop-data "E:\docker\wsl\data\docker-desktop-data.tar"
  + 然后，删除Docker镜像数据，命令：wsl --unregister docker-desktop-data
  + 最后，恢复Docker镜像数据到E盘，命令：wsl --import docker-desktop-data "E:\docker" "E:\docker\wsl\data\docker-desktop-data.tar" --version 2

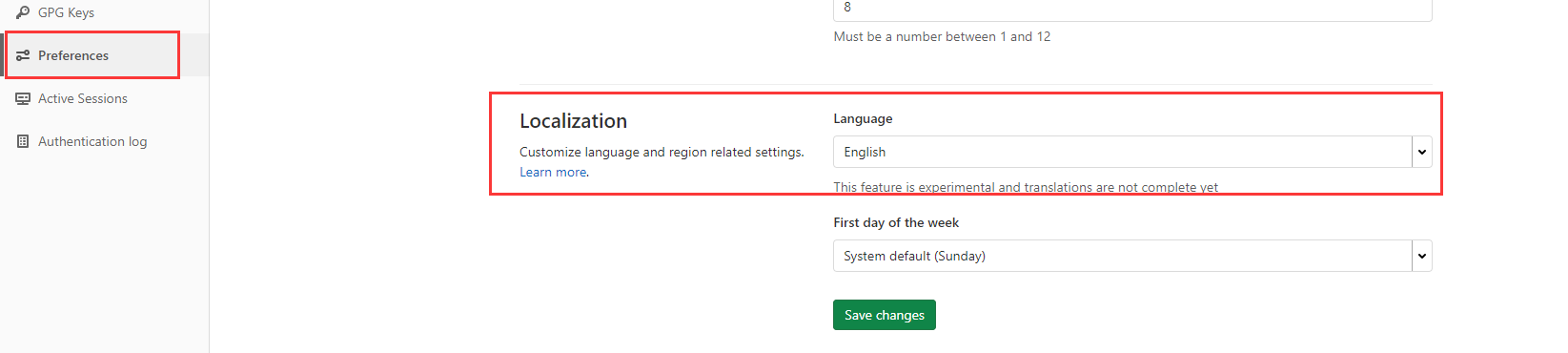
### 使用

#### 设置语言为中文

点击右上角头像处， 选择Preferences



滚动到网页最下面，将Language 选择为简体中文，点击save changes保存

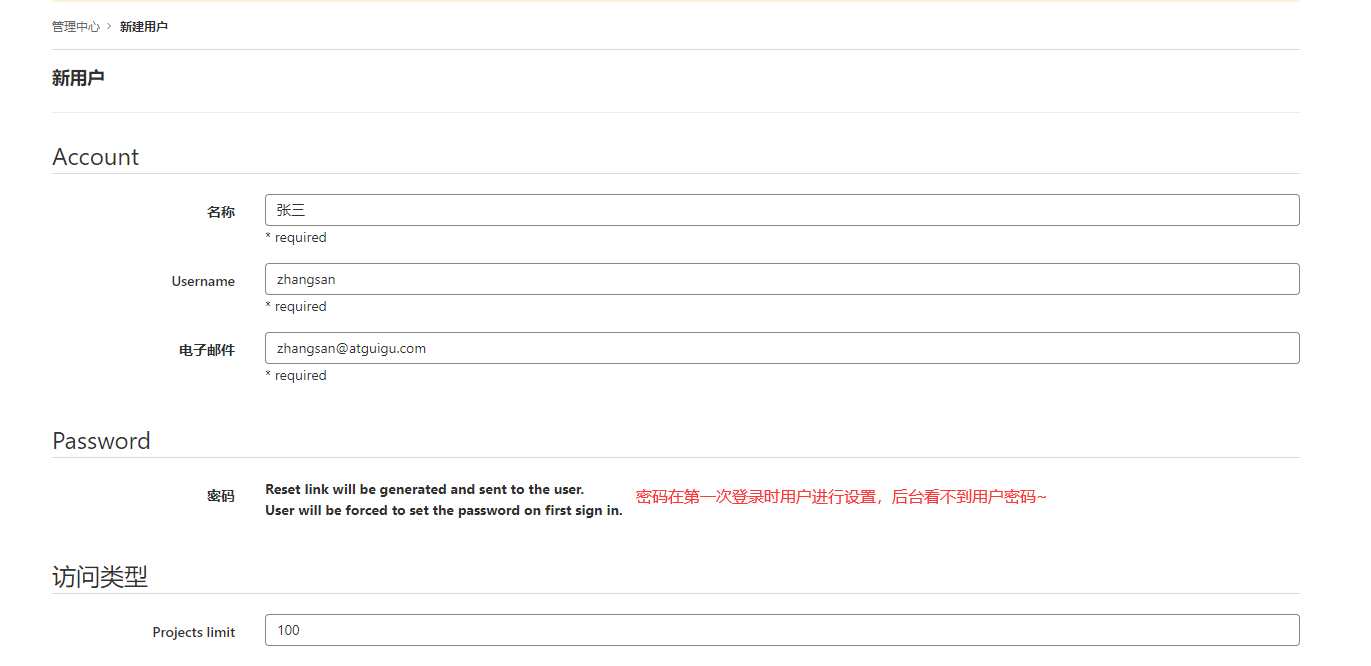


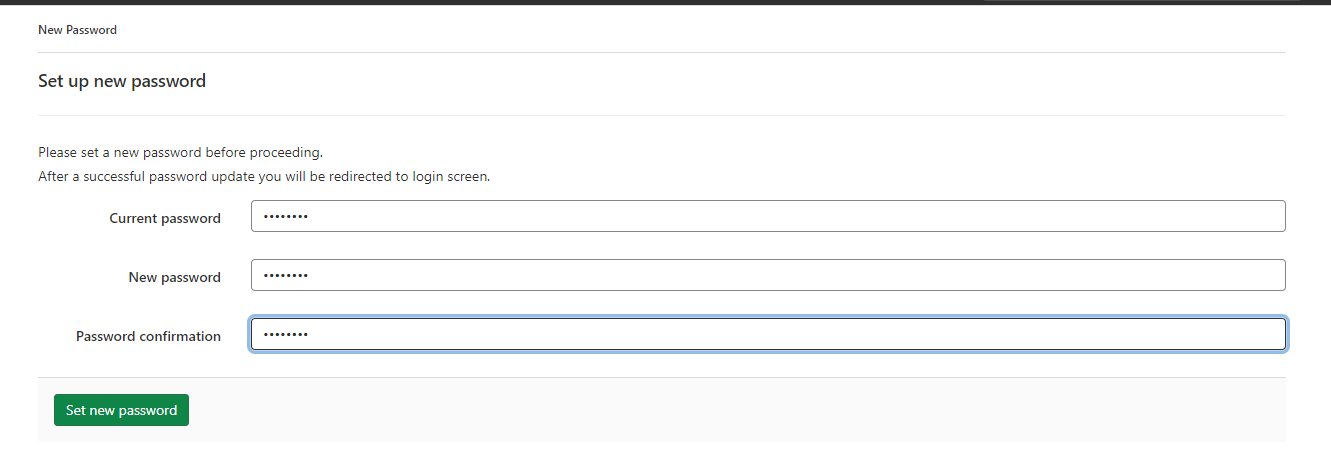
最后刷新页面生效

#### 创建用户

创建一个用来登录的用户，将来分配给每一个参与项目的成员







#### 创建群组

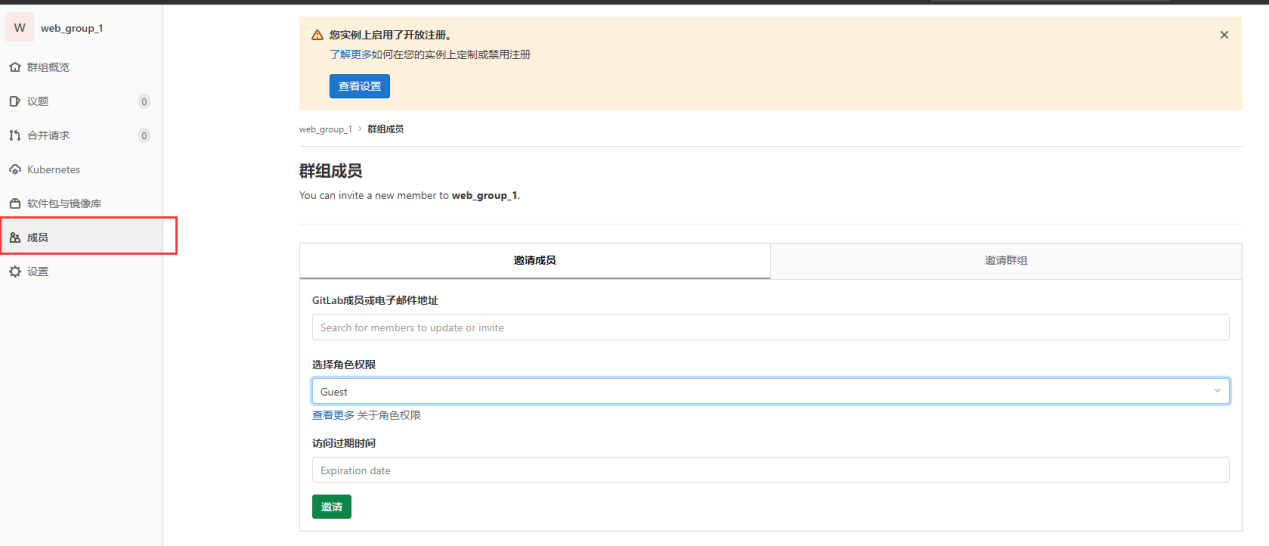
用户没有访问项目的权限，一个个添加太麻烦了，创建一个群组，拥有访问某个项目的权限，再将用户添加到群组中即可

首先，创建群组



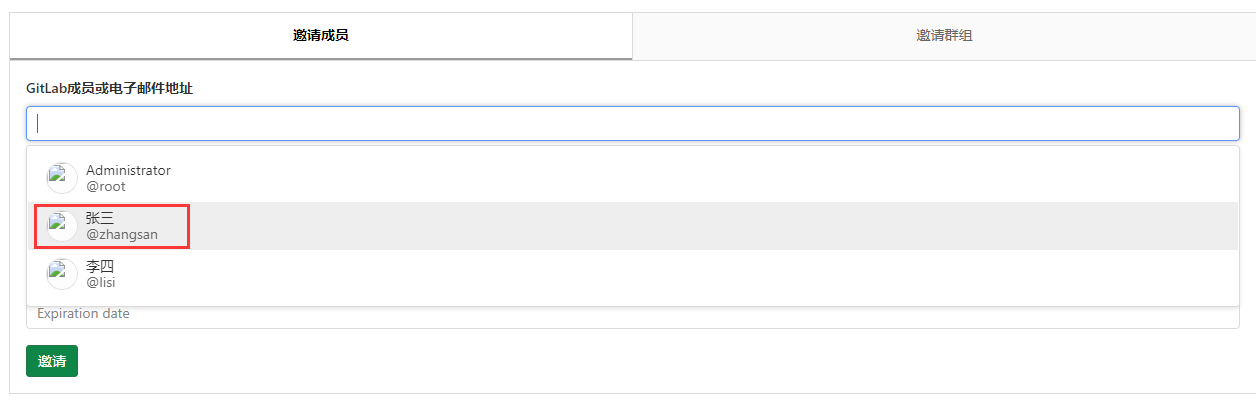


然后，邀请用户成为群组成员



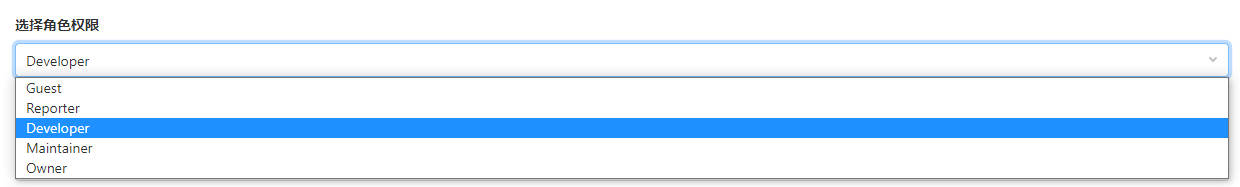
此处有三个设置：

1. 邀请成员：选择要邀请的人即可，比如我们创建的张三

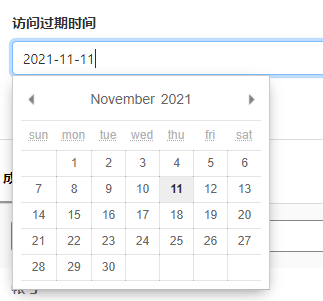


1. 选择角色权限:

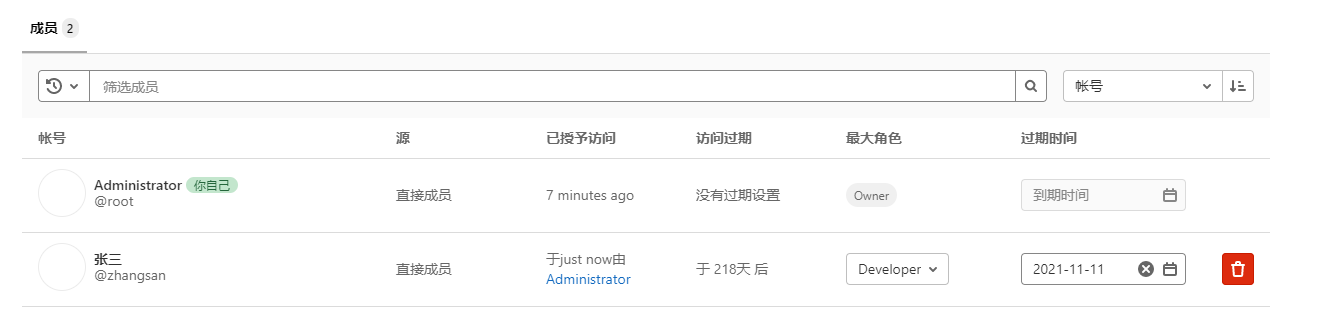
* Guest：可以创建issue、发表评论，不能读写版本库
* Reporter：可以克隆代码，不能提交，QA、PM可以赋予这个权限
* Developer：可以克隆代码、开发、提交、push，RD可以赋予这个权限
* Master：可以创建项目、添加tag、保护分支、添加项目成员、编辑项目，核心RD负责人可以赋予这个权限
* Owner：可以设置项目访问权限 - Visibility Level、删除项目、迁移项目、管理组成员，开发组leader可以赋予这个权限



1. 设置成员权限到期时间：设置为项目dead line 时间就好了

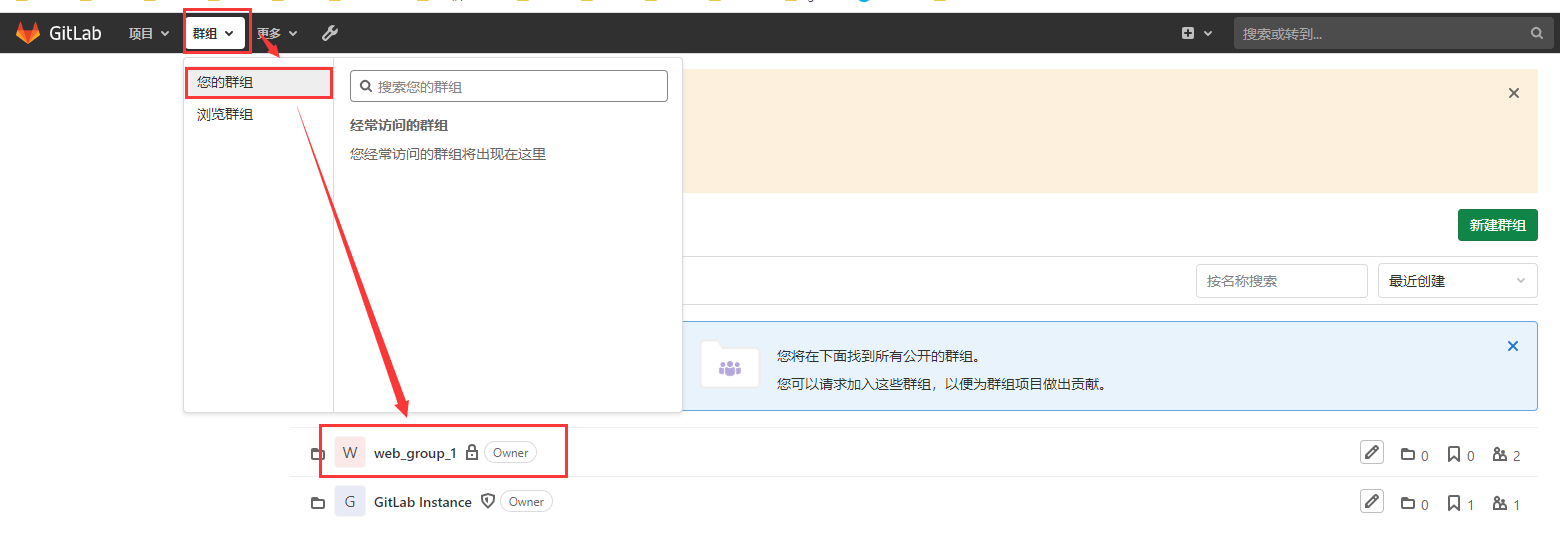


1. 点击邀请
2. 将来下面还可以直接修改前面设置的权限和到期时间



#### 创建项目仓库

回到前面我们创建的群组





选择空白项目即可





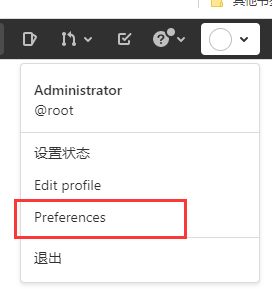
在 **克隆** 按钮位置查看仓库地址



#### 配置SSH

如果使用HTTPS方式操作仓库，请使用登录用的账户名和密码操作即可

而SSH需要进行额外配置，首先生成SSH KEY（生成方式见GitHub章节的配置SSH）



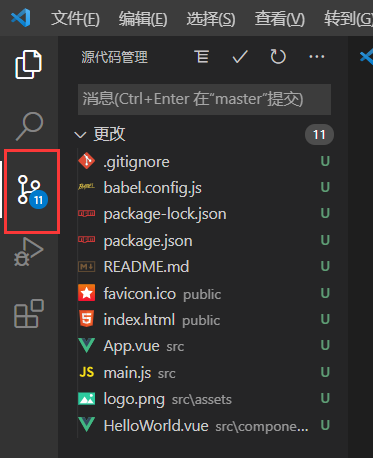




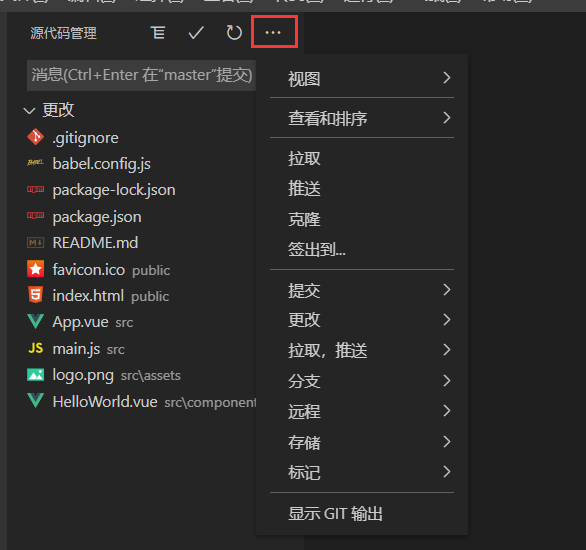
# 七、图形化工具Source Tree

# 八、结合VSCode使用

## 1. 功能简介







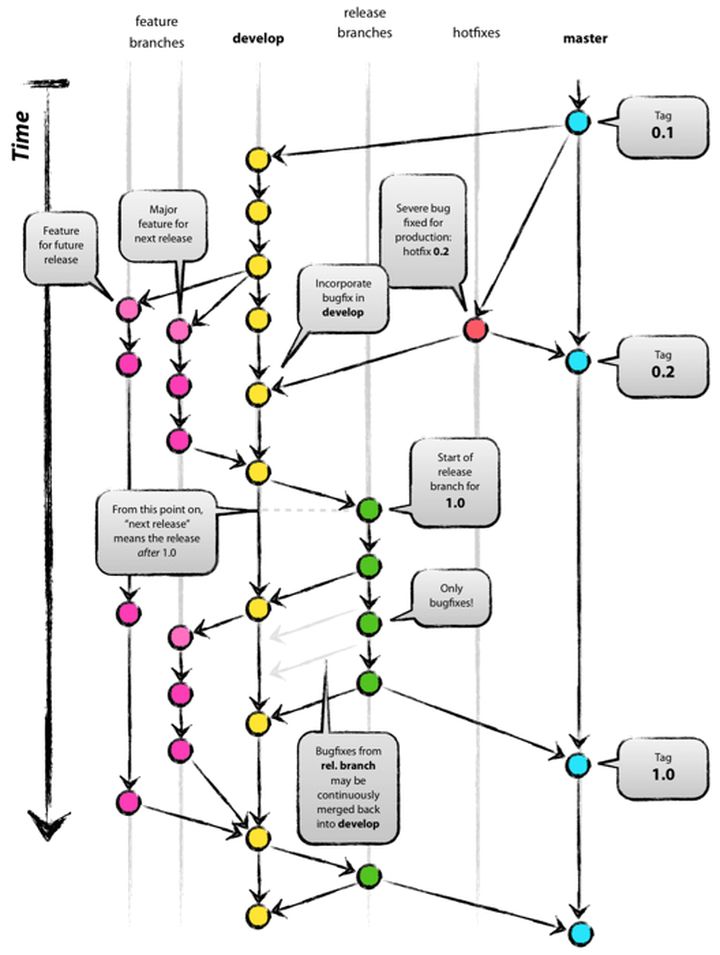
# 九、Git Flow

## 1. 简介

Git 作为一个源码管理系统，不可避免涉及到多人协作。

协作必须有一个规范的工作流程，让大家有效地合作，使得项目井井有条地发展下去。"工作流程"在英语里，叫做"workFlow"或者"Flow"，原意是水流，比喻项目像水流那样，顺畅、自然地向前流动，不会发生冲击、对撞、甚至漩涡。

## 2. Git Flow



它最主要的特点有两个：

* 首先，项目存在两个长期分支。
  + 主分支master
  + 开发分支develop
  + 前者用于存放对外发布的版本，任何时候在这个分支拿到的，都是稳定的分布版；后者用于日常开发，存放最新的开发版。
* 其次，项目存在三种短期分支。
  + 功能分支（feature branch）
  + 补丁分支（hotfix branch）
  + 预发分支（release branch）
  + 一旦完成开发，它们就会被合并进develop或master，然后被删除。

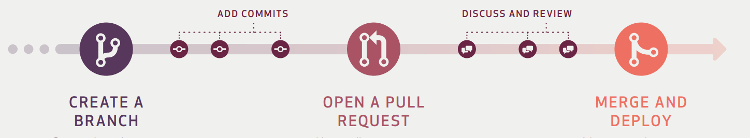
Git Flow的优点是清晰可控，缺点是相对复杂，需要同时维护两个长期分支。大多数工具都将master当作默认分支，可是开发是在develop分支进行的，这导致经常要切换分支，非常烦人。

更大问题在于，这个模式是基于"版本发布"的，目标是一段时间以后产出一个新版本。但是，很多网站项目是"持续发布"，代码一有变动，就部署一次。这时，master分支和develop分支的差别不大，没必要维护两个长期分支。

## 3. Github Flow

Github Flow 是Git Flow的简化版，专门配合"持续发布"。它是 Github.com 使用的工作流程。

它只有一个长期分支，就是master，因此用起来非常简单。



* 第一步：根据需求，从master拉出新分支，不区分功能分支或补丁分支。
* 第二步：新分支开发完成后，或者需要讨论的时候，就向master发起一个pull request（简称PR）。
* 第三步：Pull Request既是一个通知，让别人注意到你的请求，又是一种对话机制，大家一起评审和讨论你的代码。对话过程中，你还可以不断提交代码。
* 第四步：你的Pull Request被接受，合并进master，重新部署后，原来你拉出来的那个分支就被删除。（先部署再合并也可。）

Github Flow 的最大优点就是简单，对于"持续发布"的产品，可以说是最合适的流程。

问题在于它的假设：master分支的更新与产品的发布是一致的。也就是说，master分支的最新代码，默认就是当前的线上代码。

可是，有些时候并非如此，代码合并进入master分支，并不代表它就能立刻发布。比如，苹果商店的APP提交审核以后，等一段时间才能上架。这时，如果还有新的代码提交，master分支就会与刚发布的版本不一致。另一个例子是，有些公司有发布窗口，只有指定时间才能发布，这也会导致线上版本落后于master分支。

上面这种情况，只有master一个主分支就不够用了。通常，你不得不在master分支以外，另外新建一个production分支跟踪线上版本。

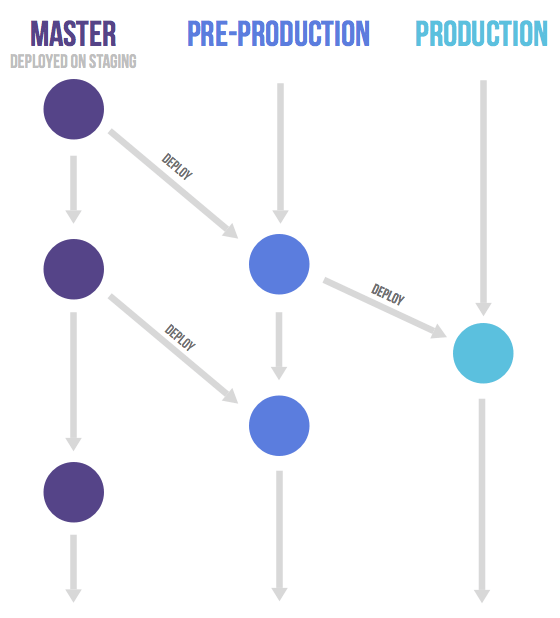
## Gitlab Flow

Gitlab Flow 是 Git Flow 与 Github Flow 的综合。它吸取了两者的优点，既有适应不同开发环境的弹性，又有单一主分支的简单和便利。它是 Gitlab.com 推荐的做法。

1. 上游优先

Gitlab Flow 的最大原则叫做"上游优先"（upsteam first），即只存在一个主分支master，它是所有其他分支的"上游"。只有上游分支采纳的代码变化，才能应用到其他分支。

1. 持续发布

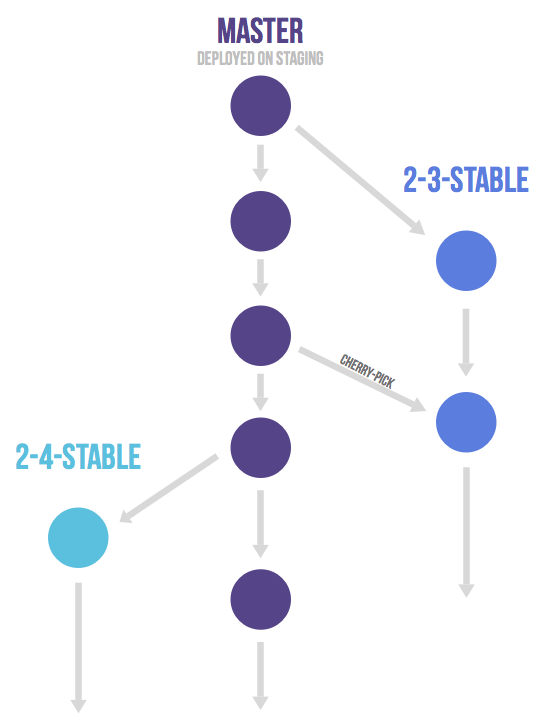


对于"持续发布"的项目，它建议在master分支以外，再建立不同的环境分支。比如，"开发环境"的分支是master，"预发环境"的分支是pre-production，"生产环境"的分支是production。

开发分支是预发分支的"上游"，预发分支又是生产分支的"上游"。代码的变化，必须由"上游"向"下游"发展。比如，生产环境出现了bug，这时就要新建一个功能分支，先把它合并到master，确认没有问题，再cherry-pick到pre-production，这一步也没有问题，才进入production。

只有紧急情况，才允许跳过上游，直接合并到下游分支。

1. 版本发布



对于"版本发布"的项目，建议的做法是每一个稳定版本，都要从master分支拉出一个分支，比如2-3-stable、2-4-stable等等。

以后，只有修补bug，才允许将代码合并到这些分支，并且此时要更新小版本号。

# 十、扩展：CI/CD

## 概念

* CI(Continuous Integration)：持续集成，当代码仓库代码发生变更，就会自动对代码进行测试和构建，反馈运行结果。
* CD(Continuous Delivery)：持续交付，是在持续集成的基础上，可以将集成后的代码依次部署到测试环境、予发布环境、生产环境等中

## 准备

准备一台云服务器ECS，我们需要在云服务器配置CI/CD。

云服务器ESC可以去任意平台购买，比如：阿里云、腾讯云、华为云、七牛云等

本文以阿里云服务器为例，最低的云服务器配置也可以使用

配置如下：

|  |  |  |  |
| --- | --- | --- | --- |
| CPU&内存 | 操作系统 | 实例规格 | 带宽 |
| 1核 2 GiB | CentOS 8.3 64位 | [ecs.t5-lc1m2.small(性能约束实例)](https://help.aliyun.com/document_detail/25378.html" \l "t5" \t "_blank) | 1Mbps |

还需要配置云服务器的安全组，配置安全组目的在于外面客户端可以访问服务器的哪些端口号，服务器需要暴露出去的就在安全组配置，不需要的就不暴露了

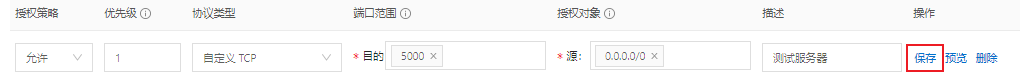
1. 来到阿里云云服务器ESC页面，打开安全组

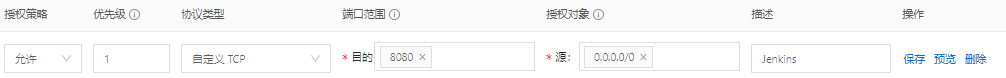


1. 配置安全组规则



1. 添加新的规则





1. 最终效果

## 本地电脑安装

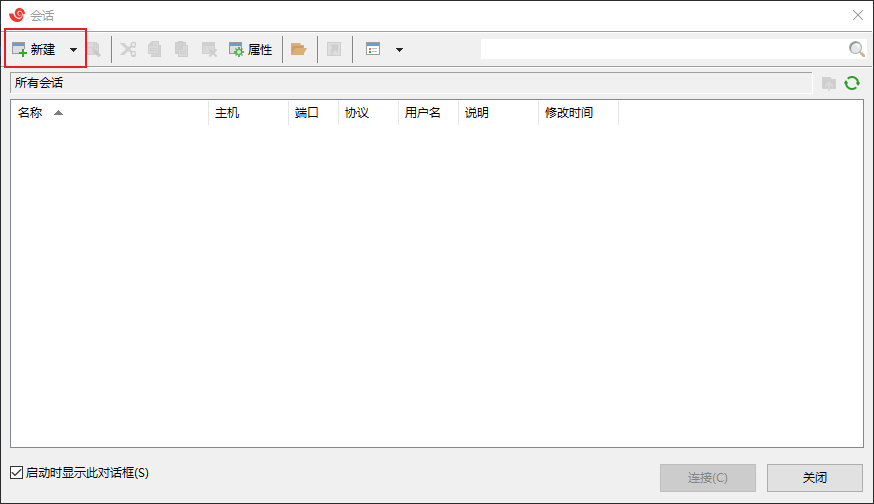
我们需要安装xshell、xftp帮助我们连接云服务器，方便操作云服务器和上传文件

## 远程云服务器安装

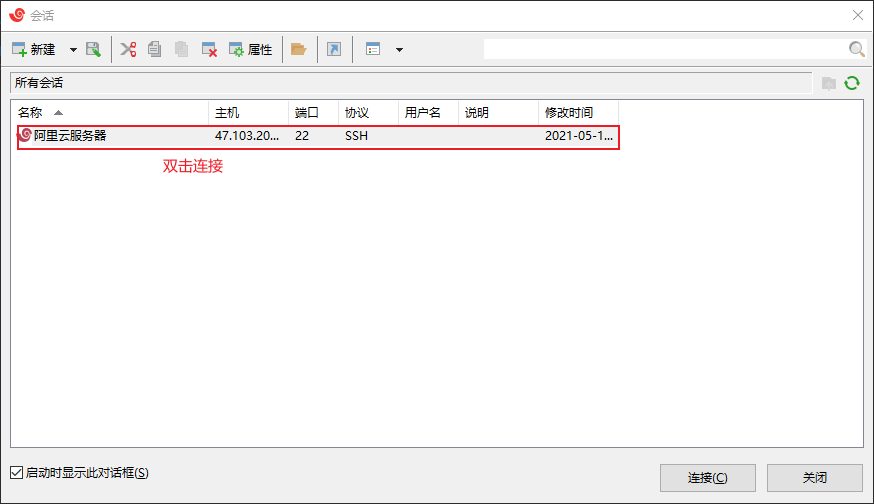
我们需要在云服务器安装nodejs、Git、JDK、Jenkins等工具来完成CI/CD

#### 使用xshell连接远程云服务器

1. 新建一个连接



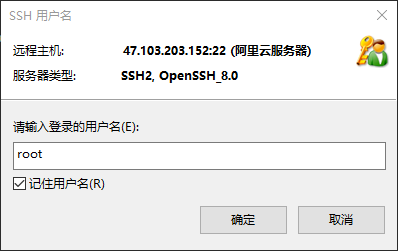
1. 填写名称和主机（公网IP：云服务器ESC - 实例 - 此时在实例列表中就能看到ip地址）
2. 确认后，双击连接，连接阿里云服务器

4. 此时验证ssh秘钥，我们选择接受并保存



5. 接下来就开始输入远程云服务器的账户名和密码

输入时请选择记住用户名和记住密码



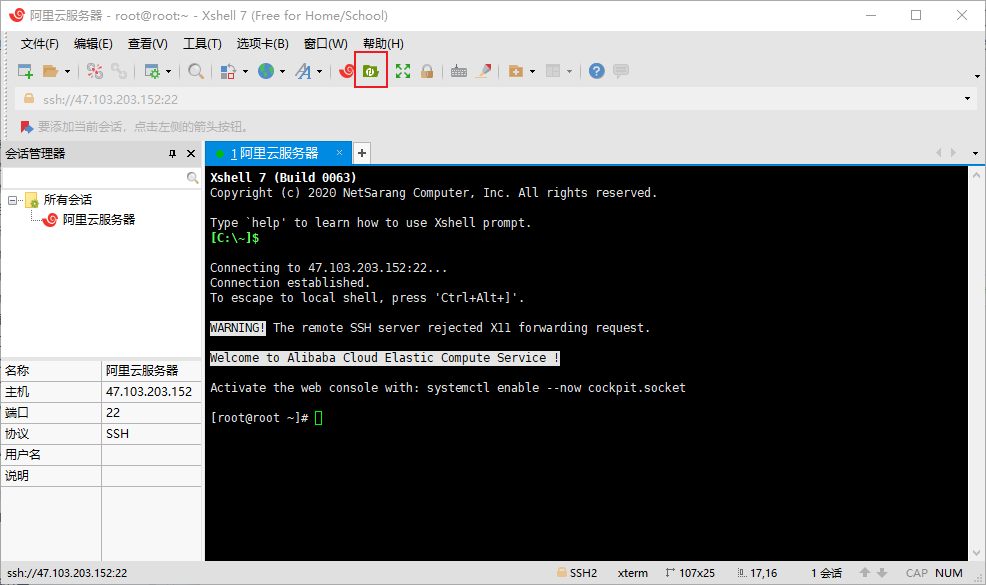
6. 连接成功，会提示：Welcome to Alibaba Cloud Elastic Compute Service !

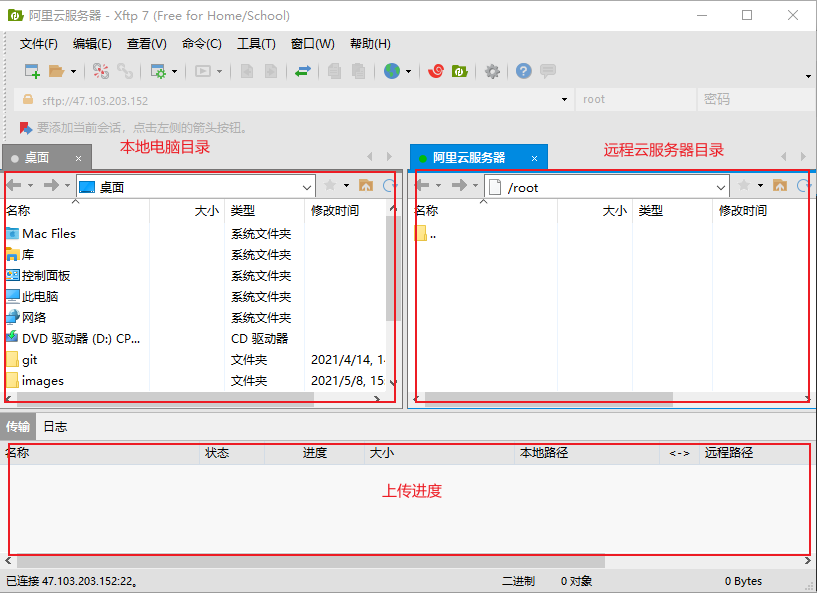


#### 安装nodejs

1. 将nodejs安装包上传到云服务器上

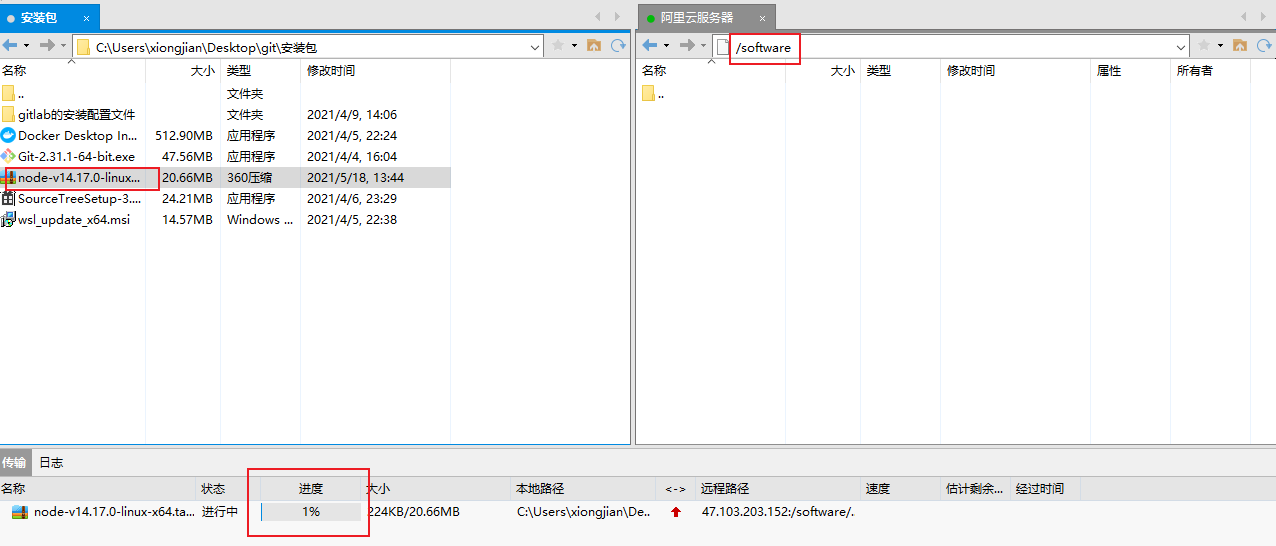
打开xftp，在远程云服务器根目录下创建一个文件夹software，将本地nodejs安装包上传到software中







4



1. 解压nodejs

cd /software

1

tar -xvf node-v14.17.0-linux-x64.tar.xz

3

1. 将nodejs移动到/usr/local/node目录下

mv node-v14.17.0-linux-x64 /usr/local/node

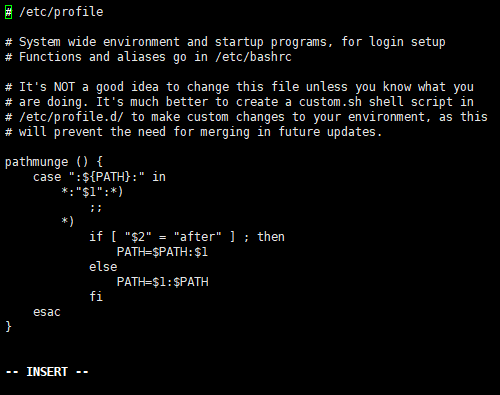
1

1. 配置环境变量，让nodejs可以在任意目录访问

vi /etc/profile 编辑文件

1

进入文件输入 i，进入insert模式



按方向键下键，来到最后，添加nodejs环境变量配置

export PATH=$PATH:/usr/local/node/bin



esc 退出编辑模式

: 开始输入模式

wq 保存并退出

4

让配置的环境变量生效

source /etc/profile

5

测试

6

最后为了下载包速度更快，我们配置淘宝镜像地址

npm config set registry https://registry.npm.taobao.org

1

#### 安装Git

输入命令：yum install -y git

等待安装即可

查看是否安装成功：git --version

1

首次使用，需要给其配置邮箱和用户名

1

#### 安装JDK

输入命令：yum install -y java-1.8.0-openjdk

等待安装即可

查看是否安装成功：java -version

1

#### 安装Jenkins

1. 下载repo的安装包

sudo wget -O /etc/yum.repos.d/jenkins.repo <https://pkg.jenkins.io/redhat-stable/jenkins.repo>



1. 下载秘钥

sudo rpm --import https://pkg.jenkins.io/redhat-stable/jenkins.io.key

2

1. 安装

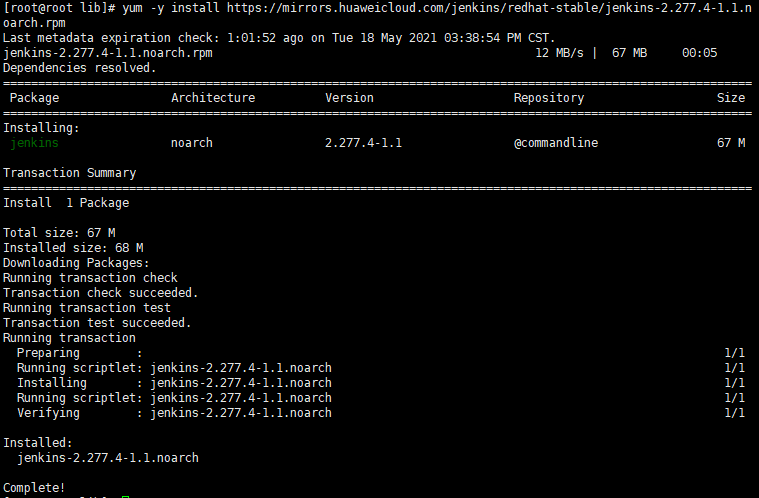
官方jenkins使用的是国外镜像，安装很慢

我们使用华为云镜像安装，可以去这个地址查看最新的安装包

<https://mirrors.huaweicloud.com/jenkins/redhat-stable>

输入命令：

yum -y install https://mirrors.huaweicloud.com/jenkins/redhat-stable/jenkins-2.277.4-1.1.noarch.rpm



1. 启动

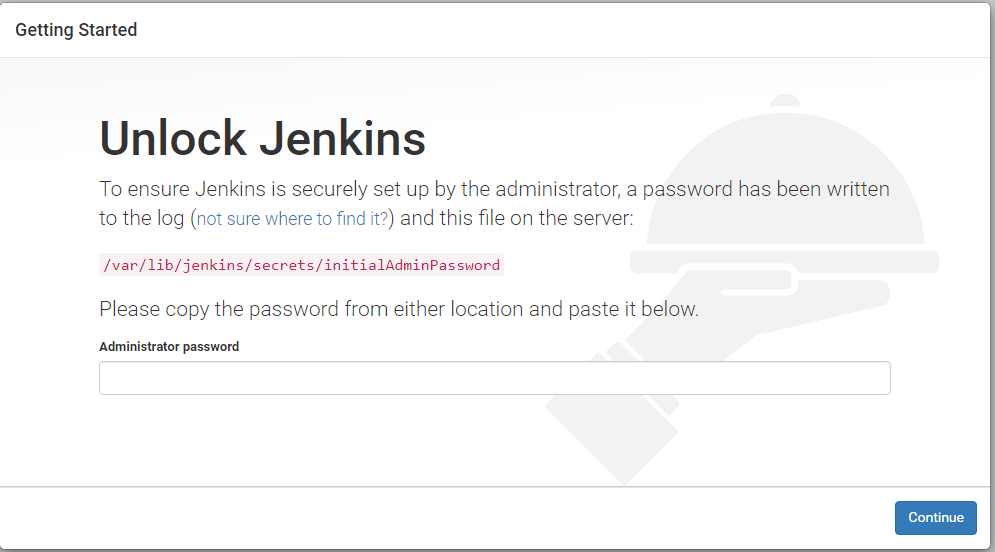
systemctl start jenkins

2

1. 在本地电脑，打开网址访问Jenkins

http://您的公网ip:8080

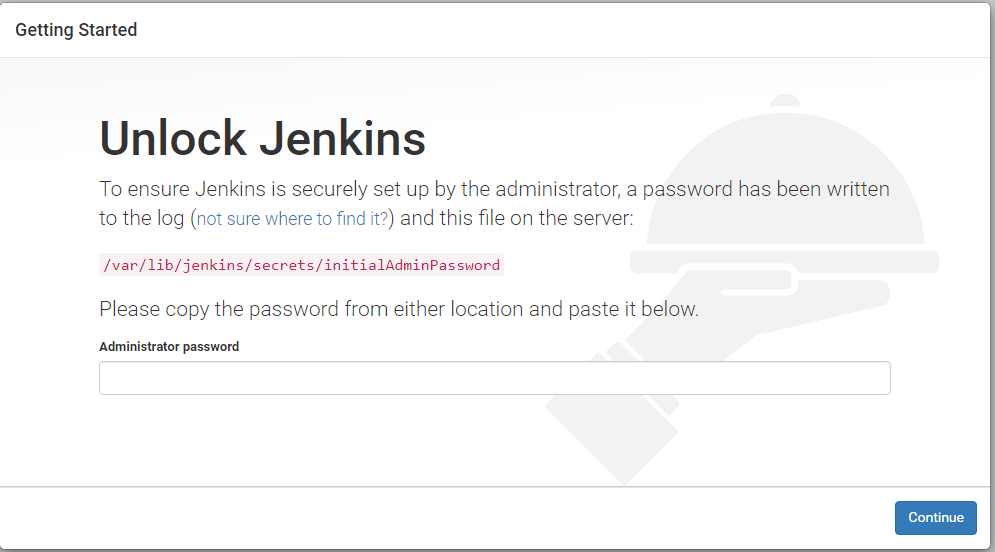
比如：http://47.103.203.152:8080



## 配置Jenkins并安装插件

初次使用Jenkins需要导入秘钥

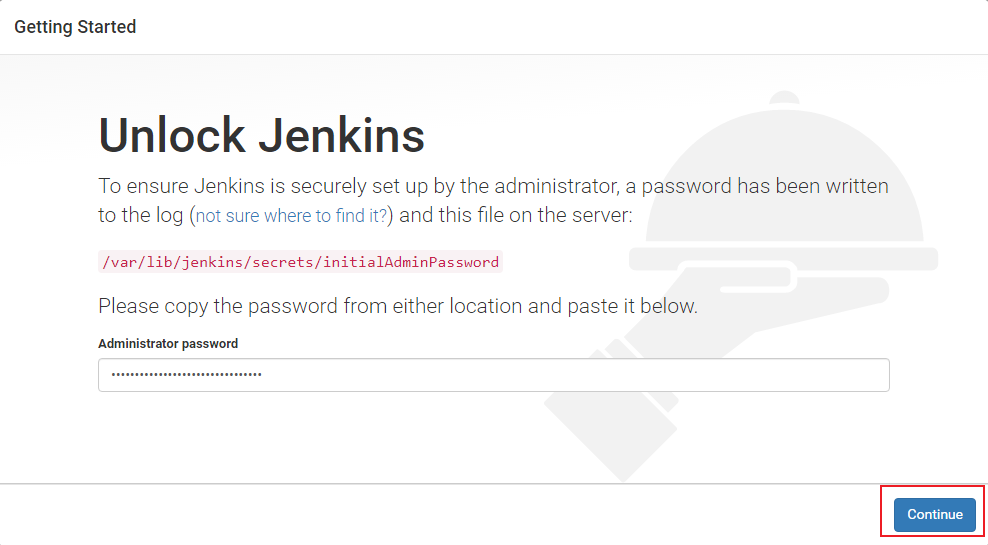
图中给出了秘钥地址：/var/lib/jenkins/secrets/initialAdminPassword



进入/var/lib/jenkins/secrets目录，查看initialAdminPassword文件内容

1

复制initialAdminPassword文件内容到网站上，点击继续



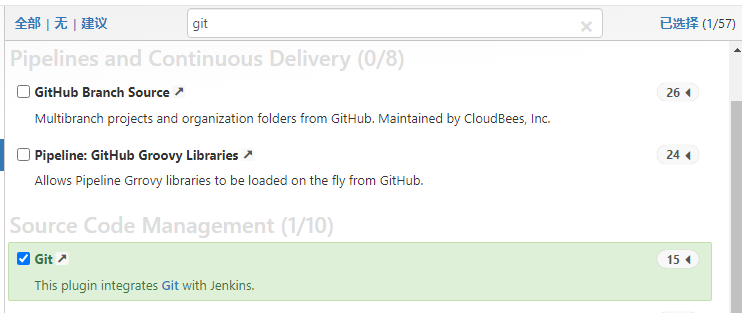
选择自定义安装插件

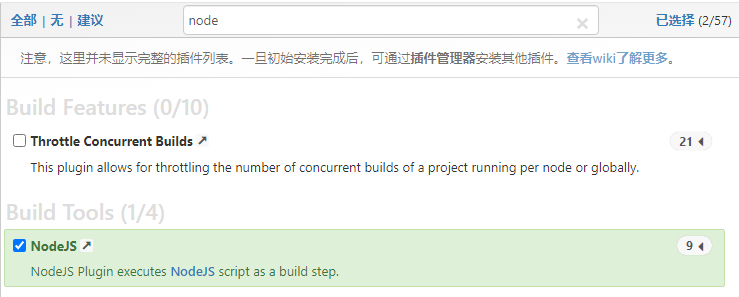


选择无，取消默认安装的插件



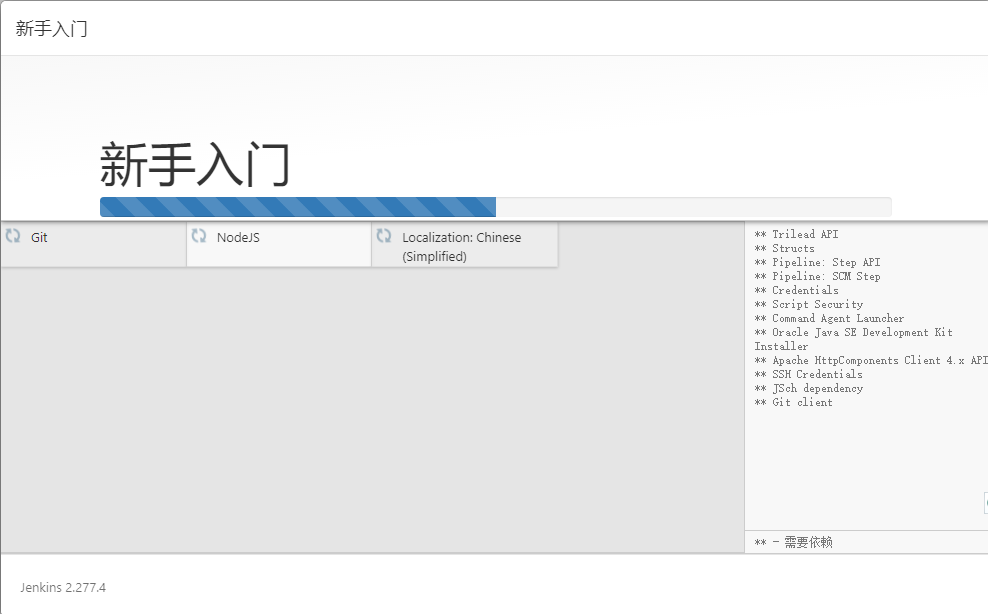
选择安装git、nodejs、Localization: Chinese (Simplified)三个插件







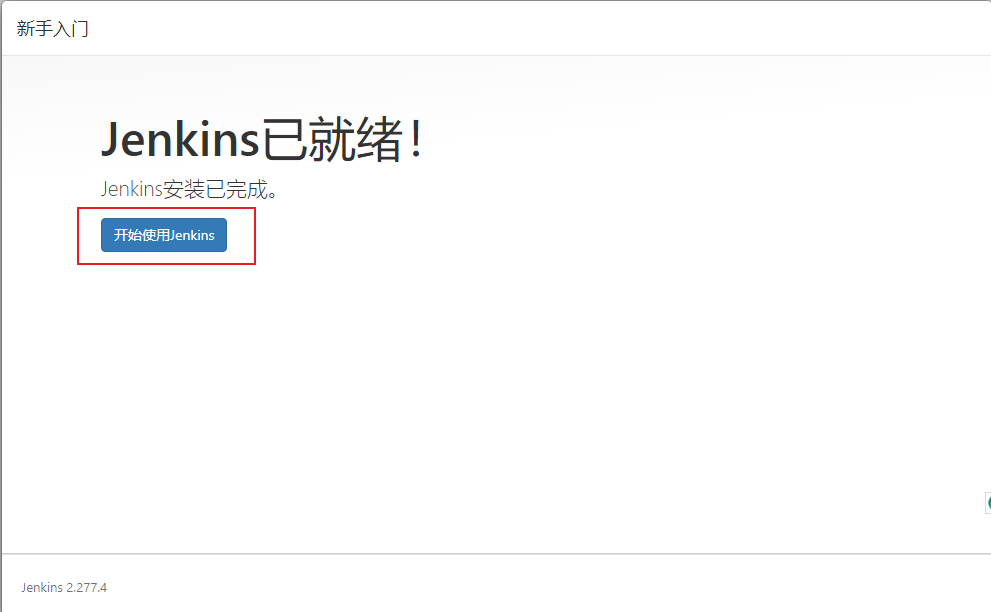
点击安装，进入安装页面



创建一个用户名和密码，请记住，以后不能找回的



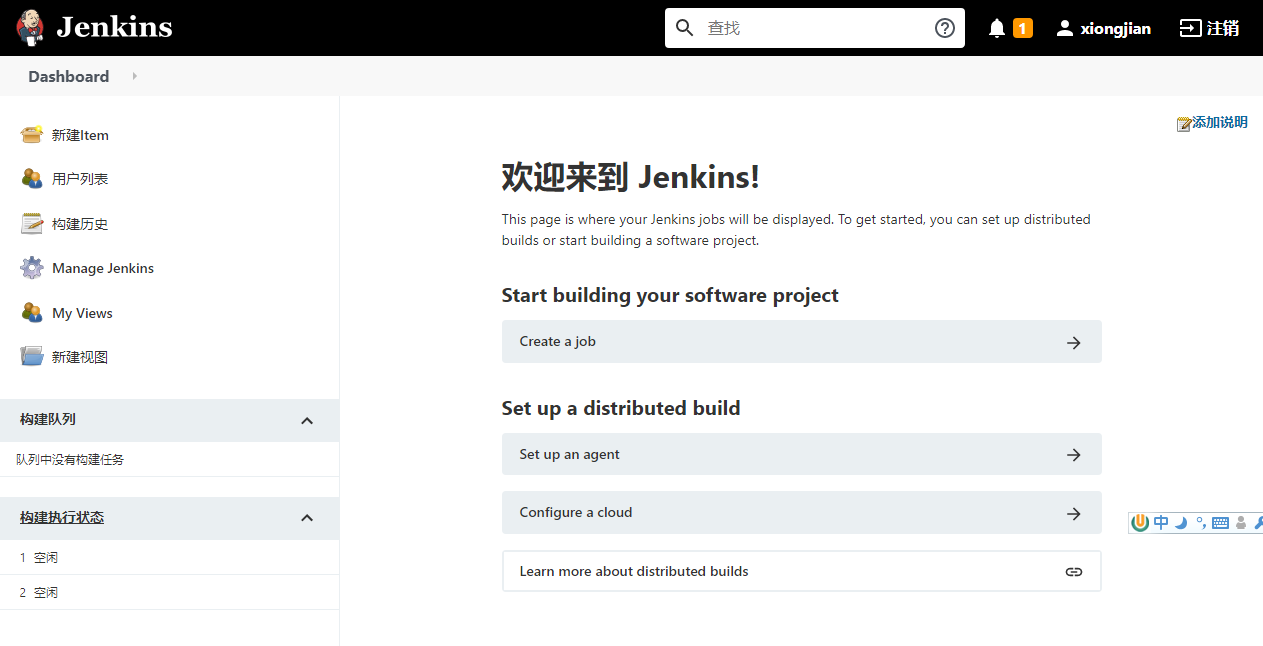




输入网址：http://您的公网ip地址:8080

如：http://47.103.203.152:8080

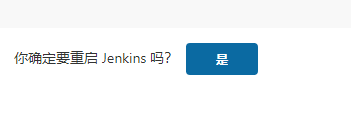
开始访问Jenkins



进入后改变地址为：http://您的公网ip地址:8080/restart

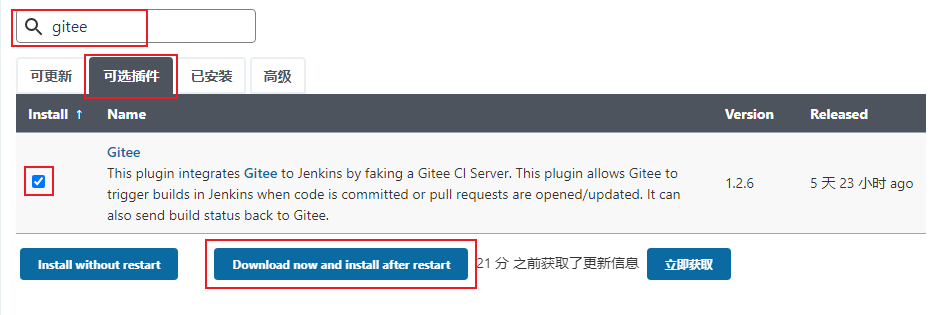
重启Jenkins，使我们安装的插件生效

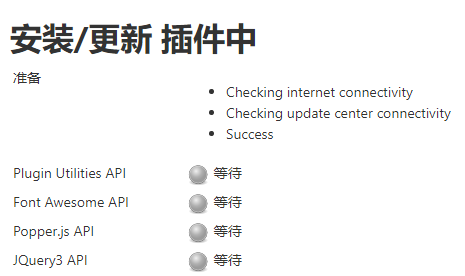
1



因为我们使用Gitee来管理源码仓库，所以还需要安装Gitee插件，而这个插件在最开始时是搜不到的，所以要进来搜索





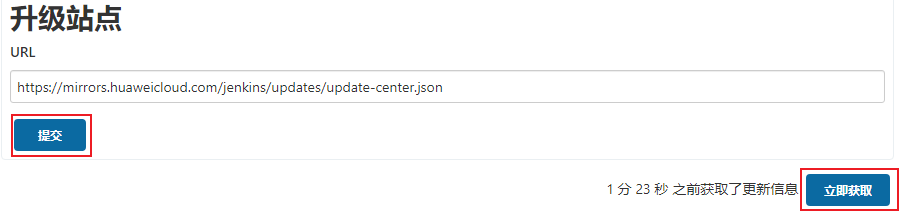


等待插件安装成功，即可下一步

如果**安装插件失败**，可以进行以下配置，再重新安装

来到系统管理 - 插件管理 - 高级

将url改为华为云镜像地址 https://mirrors.huaweicloud.com/jenkins/updates/update-center.json



再将Jenkins重启在安装缺少的插件

## 配置Jenkins插件

#### 配置nodejs插件

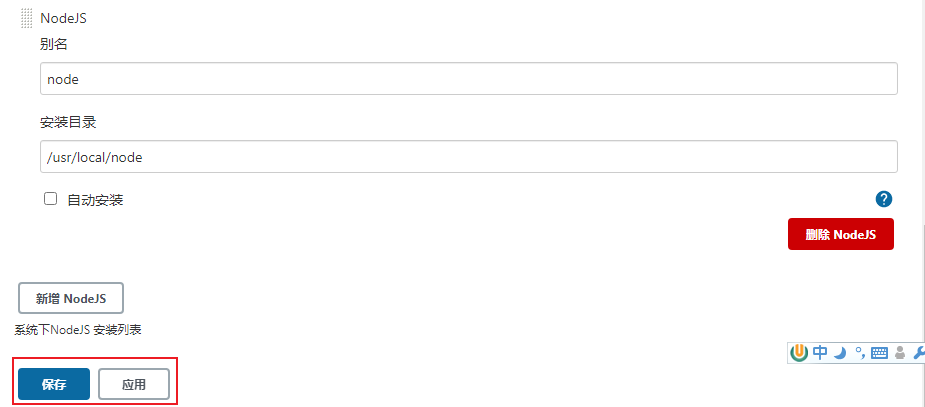
打开系统管理-全局工具配置



来到最下面，点击新增NodeJS



取消自动安装，填写以下内容，点击保存



#### 配置Gitee插件

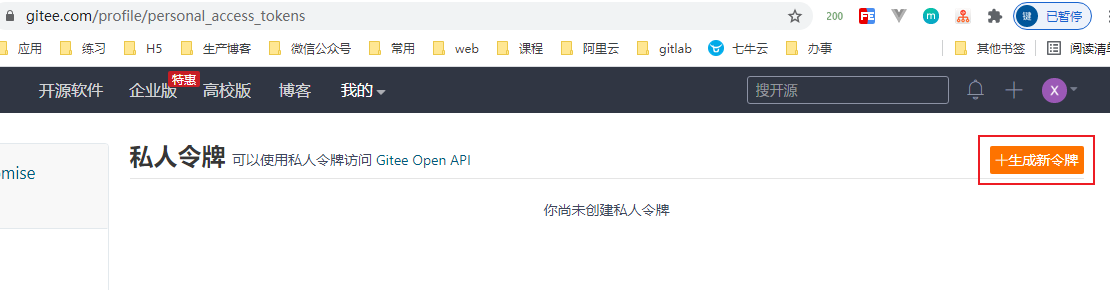
1. 来到系统管理-系统配置



1. 最下面配置Gitee



1. 生成GItee私人令牌，并复制令牌内容



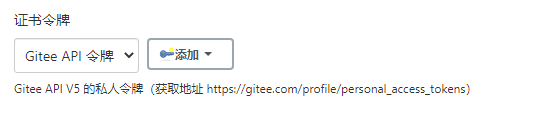




1. 来到Jenkins的Gitee配置处，点击添加 - 选择Gitee API 令牌，最后点击添加



1. 将刚添加的令牌选中



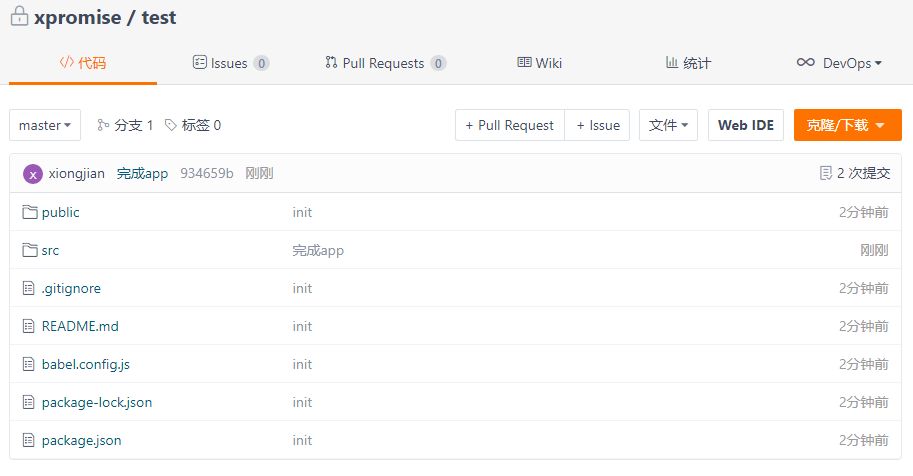
1. 测试链接是否成功



1. 最后记得保存起来才能生效~

## 准备一个Gitee项目仓库

新建一个Gitee仓库，并将项目代码上传上去



## 创建流水线

#### 新建任务



#### 输入任务名称和构建自由风格的软件项目



#### 添加Gitee的配置

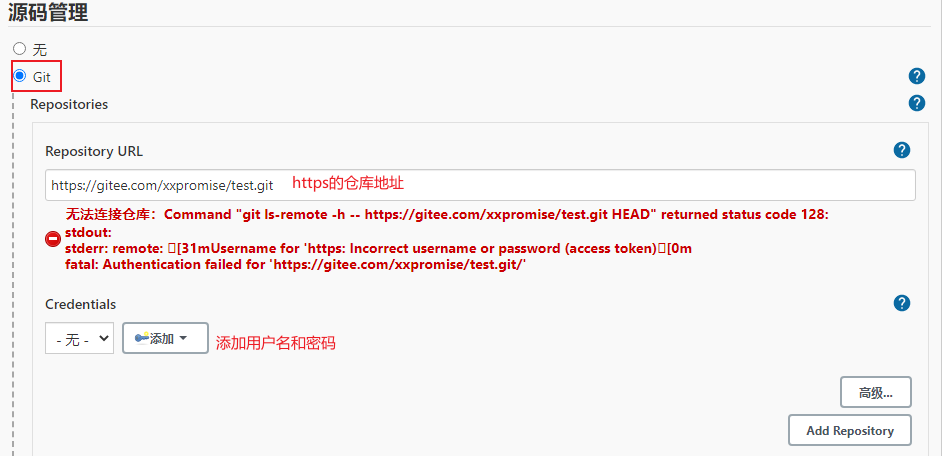
作用：配置打包后的项目保存多久和数量



#### 配置Git

添加之前创建的Gitee仓库地址，此时有一个报错：连接不上仓库，是正常的，因为我们还没有填写用户名和密码

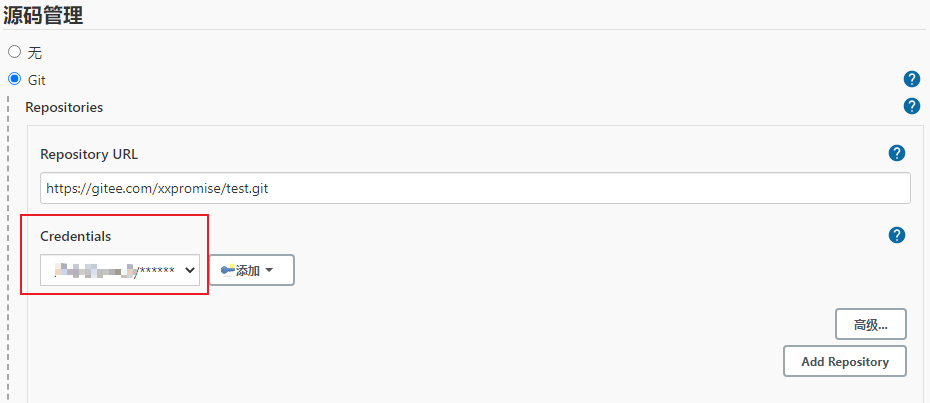
作用：可以让我们获取项目源代码，从而打包



添加一个Credentials，输入Gitee账户的用户名和密码，并点击添加



选中刚添加的凭据



#### 配置Gitee WebHook

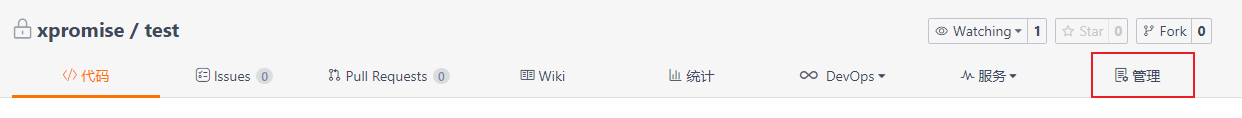
作用：什么时候需要重新打包项目呢？当用户进行一次push时，项目源代码发生变化，此时需要重新打包。WebHook就是一个通知机制，当用户push时，会通知Jenkins进行打包



来到下面，生成随机的WebHook密码



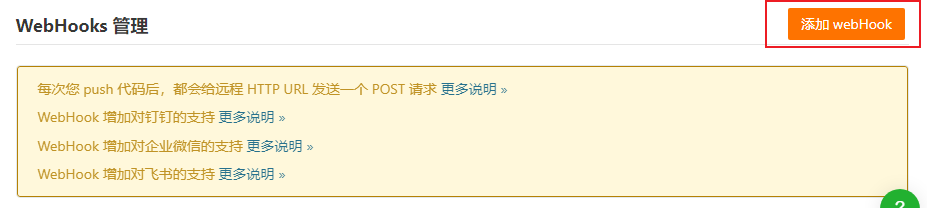
接下来要来到Gitee项目仓库处，点击管理



点击WebHooks



添加WebHook



此时需要填写两个内容：

一个是url，要返回Jenkins任务处复制地址



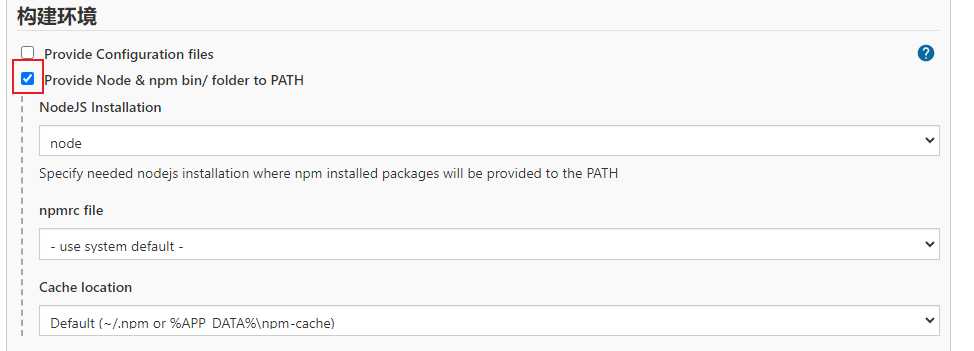
另一个是WebHook密码，来到Jenkins任务生成随机密码处复制密码

两个都填好后，点击添加

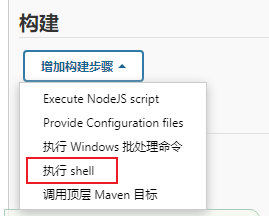


#### 添加node

作用：让后续执行shell能找到npm

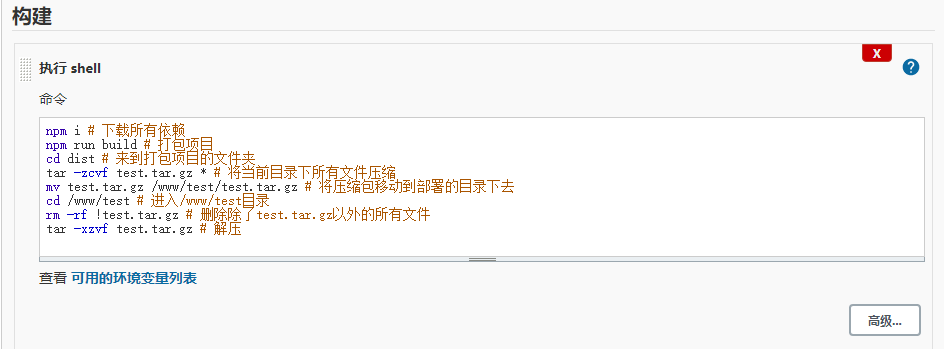


#### 执行shell



脚本内容：

|  |
| --- |
| npm i # 下载所有依赖  npm run build # 打包项目  cd dist # 来到打包项目的文件夹  tar -zcvf test.tar.gz \* # 将当前目录下所有文件压缩  mv test.tar.gz /www/test/test.tar.gz # 将压缩包移动到部署的目录下去  cd /www/test # 进入/www/test目录  rm -rf !test.tar.gz # 删除除了test.tar.gz以外的所有文件  tar -xzvf test.tar.gz # 解压 |



#### 整体流程分析

1. 当开发者进行一次push代码到指定仓库中，Gitee仓库会触发一个WebHook
2. WebHook会发送一个POST请求到Jenkins服务器
3. Jenkins服务器接受到，就会触发指定流水线任务
4. 拉取Gitee仓库代码到Jenkins工作空间来
5. 执行shell脚本
6. shell脚本就会下载依赖，打包，并部署到指定目录

#### 其他配置

我们需要将部署的目录新建好，并且使用nodejs搭建服务器，pm2进程守卫让服务更健壮

1. 新建目录

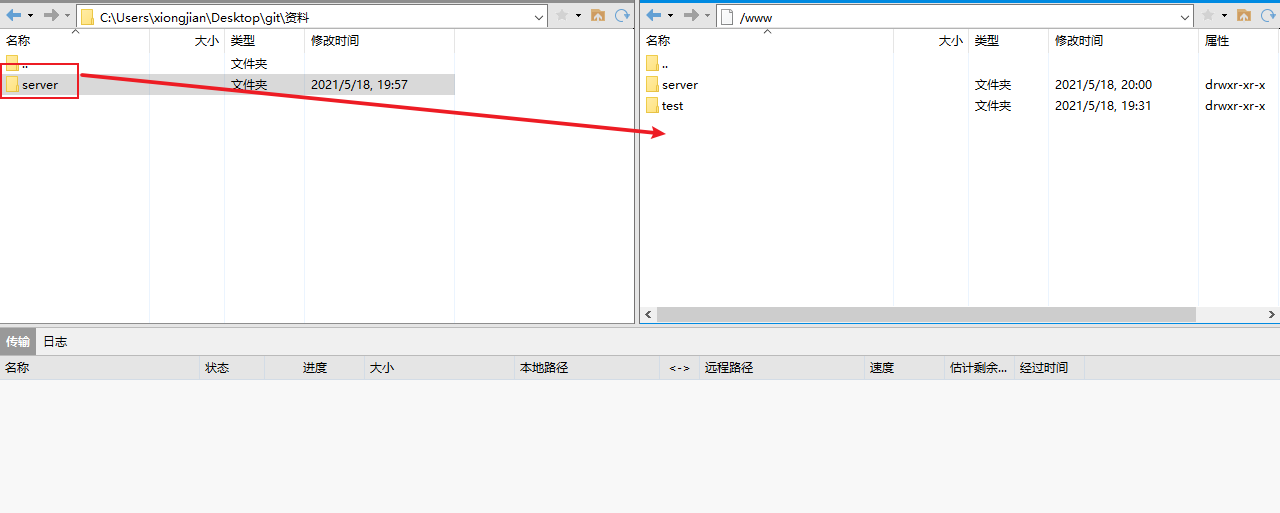
mkdir -p /www/test

1

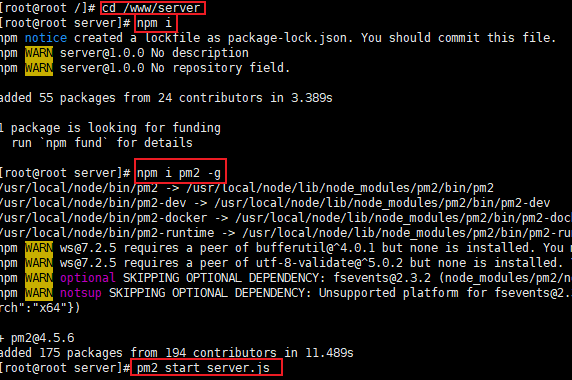
1. 给目录授权

chmod -R 777 /www

1. 将写好的nodejs脚本复制到云服务器去



1. 下载依赖并运行



1. 访问地址是：http://您的ip地址:5000

还没有进行一次打包，所以第一次访问不到任何内容

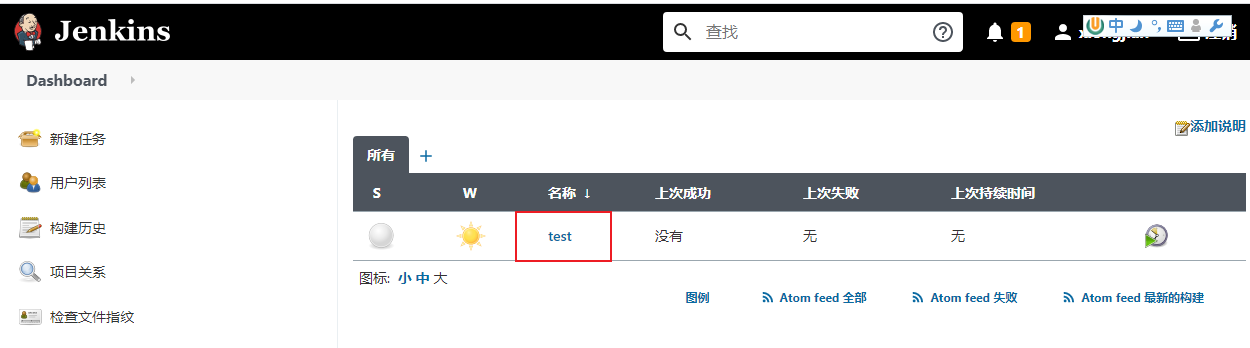
注意：云服务器安全组配置了5000端口是放行的才能访问！

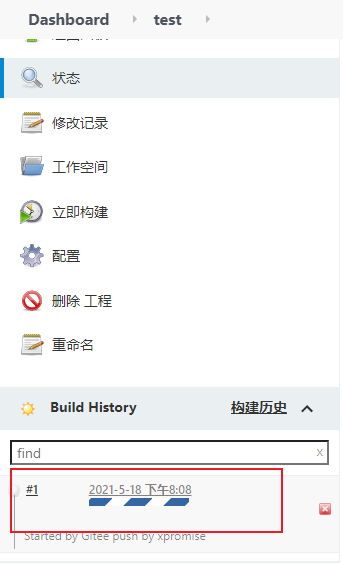
## 测试访问

#### 进行一次Gitee Push



#### 查看Jenkins工作进度





蓝白相间：代表工作中

红色：工作失败

蓝色：工作成功

#### 打开网址，查看网址

打开http://您的ip地址:5000

查看有无内容，并且有没有变化