**Gitlab使用手册**

文档更新记录

|  |  |  |  |
| --- | --- | --- | --- |
| 版本历史 | 作者 | 日期 | 备注 |
|  |  |  |  |

目录

[1使用准备 4](#_Toc498501716)

[1.1用户登录 4](#_Toc498501717)

[1.2修改密码 4](#_Toc498501718)

[1.2生成和上传 ssh key 5](#_Toc498501719)

[1.2.1生成ssh key 5](#_Toc498501720)

[1.2.2上传ssh key 5](#_Toc498501721)

[2项目和用户组 6](#_Toc498501722)

[2.1 创建组 6](#_Toc498501723)

[2.2 创建新项目 7](#_Toc498501724)

[2.3添加成员 8](#_Toc498501725)

[2.3.1 添加组成员 8](#_Toc498501726)

[2.3.2 添加项目成员 9](#_Toc498501727)

[2.4查看项目详情 11](#_Toc498501728)

[2.4.1 Overview 11](#_Toc498501729)

[2.4.2 Repository 12](#_Toc498501730)

[2.4.3 Issues 12](#_Toc498501731)

[2.4.4 Wikis 14](#_Toc498501732)

[2.4.5 Snippets 15](#_Toc498501733)

[2.4.6 Notifications 15](#_Toc498501734)

[3 Git仓库和项目 16](#_Toc498501735)

[3.1提交代码 16](#_Toc498501736)

[3.2添加评论 16](#_Toc498501737)

[3.3新建标签 17](#_Toc498501738)

[3.4新建分支 18](#_Toc498501739)

[3.5保护分支和保护标签 19](#_Toc498501740)

[3.5.1 保护分支 19](#_Toc498501741)

[3.5.2 保护标签 20](#_Toc498501742)

[3.6新建合并请求 21](#_Toc498501743)

[3.7代码审查 22](#_Toc498501744)

[3.8待办事项 23](#_Toc498501745)

[4 git常用命令汇总 23](#_Toc498501746)

[4.1第一个Git仓库 23](#_Toc498501747)

[4.2查看工作区状态 23](#_Toc498501748)

[4.3添加工作目录的文件到暂存区 24](#_Toc498501749)

[4.4提交暂存区的文件到仓库 24](#_Toc498501750)

[4.5撤销工作目录的修改 24](#_Toc498501751)

[4.6撤销暂存区的修改 24](#_Toc498501752)

[4.7查看修改前后的异同 25](#_Toc498501753)

[4.8查看提交信息 25](#_Toc498501754)

[4.9从暂存区删除文件 26](#_Toc498501755)

[4.10克隆远程仓库到本地 26](#_Toc498501756)

[4.11关联本地仓库与远程仓库 26](#_Toc498501757)

[4.12多分支管理 26](#_Toc498501758)

[4.13标签管理 27](#_Toc498501759)

[4.14远程仓库及多分支 28](#_Toc498501760)

[5 Gitlab官方帮助文档 28](#_Toc498501761)

# 1使用准备

## 1.1用户登录

创建成功用户后，新建的用户会收到一封来自gitlab的Confirmation instructions邮件，需要确认了才能登陆使用。

新建用户确认流程：

   1、**请点击**[Confirm your account](http://172.16.1.161/users/confirmation?confirmation_token=1CpskayDp2Uk3WqDJ2uj)

   2、复制如下地址到个人虚拟机**chrome**浏览器里登录确认。

C:\Users\p\AppData\Roaming\Foxmail7\Temp-3424-20171128085834\Attach\Catch6B2C(12-05-17-27-45).jpg

  3、**登录**

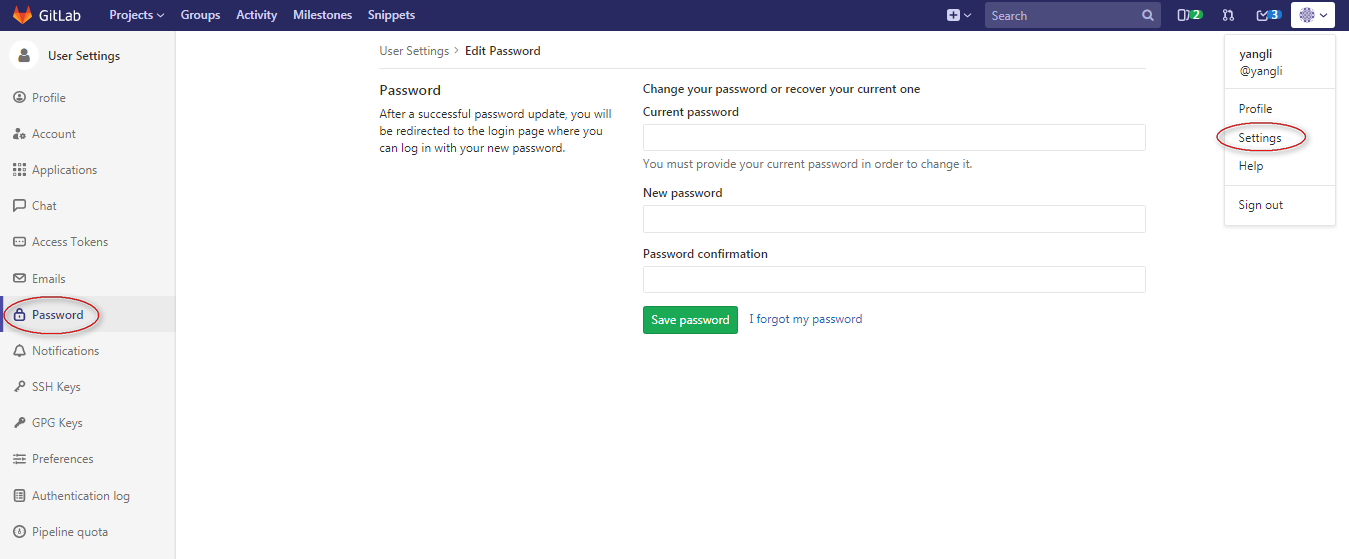
   gitlab地址：<http://192.168.31.76/> （内网）

   gitlab登陆名：完整邮箱或邮箱前缀不带@raisecom.com

   gitlab密码：raisecom

## 1.2修改密码

在面板上个人账户下，依次点击【Settings】、【Password】，进入密码修改界面，重新设置即可。如果忘记密码就点击“I forgot my password”，gitlab将会给你的邮箱发送重置密码邮件。



## 1.3生成和上传 ssh key

### 1.3.1生成ssh key

在命令行下执行：ssh-keygen

执行过程中，会要求你确认保存公钥的位置（默认为：/home/user/.ssh/id\_rsa），

然后它会让重复输入一个密码两次，如果不想在以后每次使用推送代码的时候都输入密码，可以直接留空。

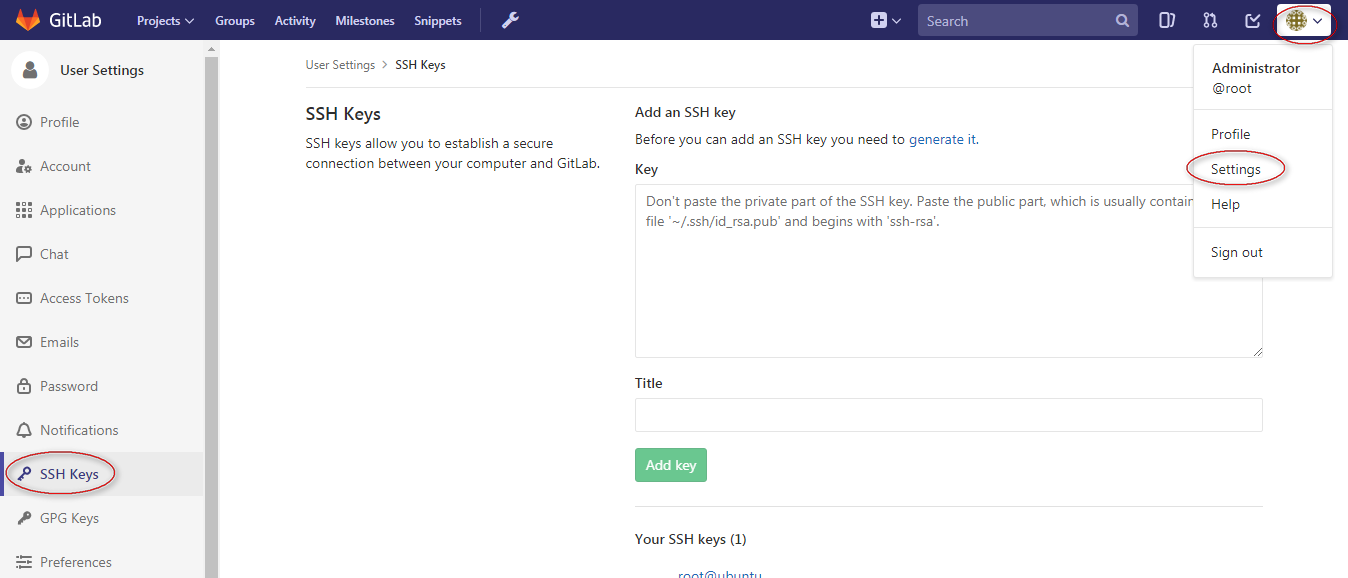
执行完毕后，就会生成一对密钥，id\_rsa是private key，id\_rsa.pub是public key。

### 1.3.2上传ssh key

用户必须先将ssh public key导入gitlab才可以进行代码的上传与下载等操作。SSH KEY由用户自己添加，可以添加多个KEY。

方法如下：

1. 复制本地id\_rsa.pub里的内容；
2. 在面板上个人账户下，依次点击【Settings】、【SSH Keys】；
3. Add key。

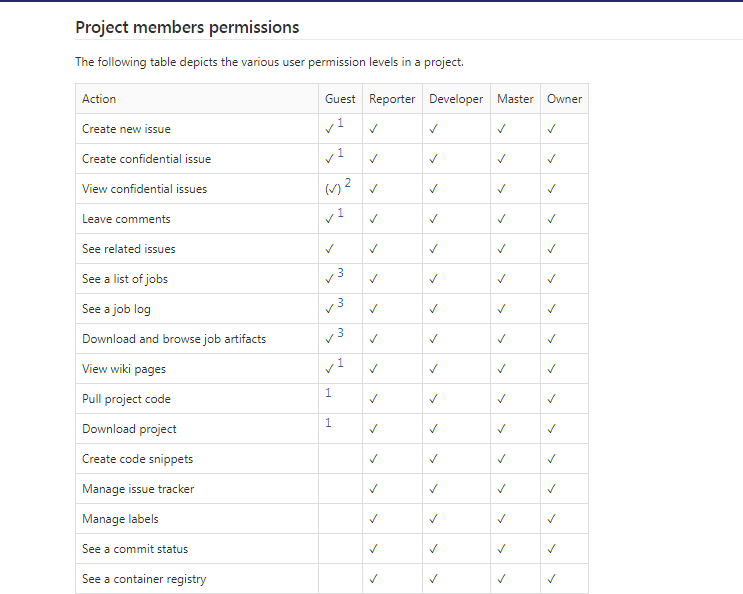
****

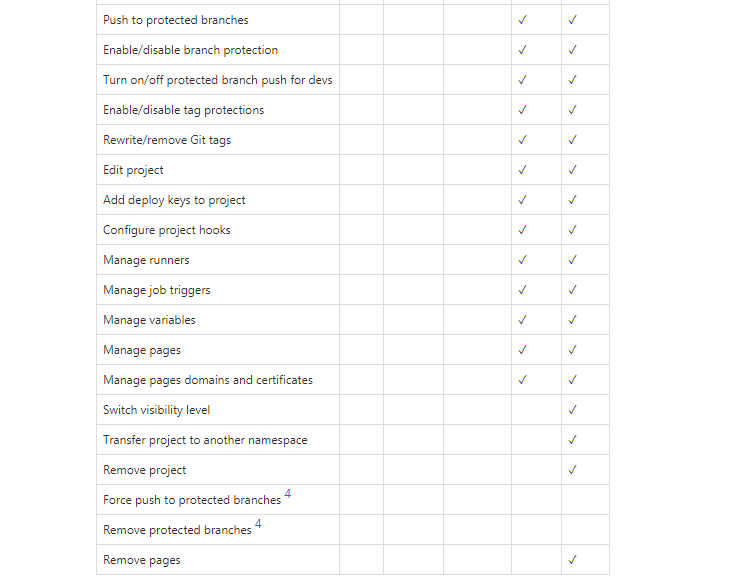
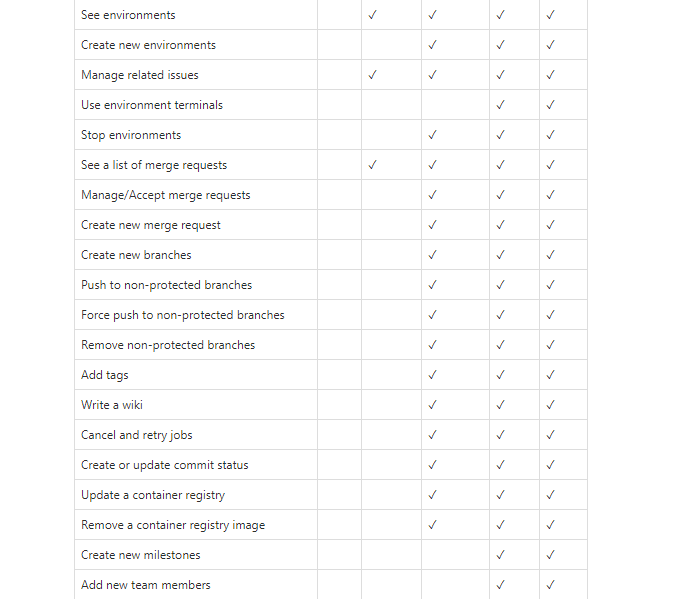
# 2权限相关

## 2.1五中权限介绍

用户project权限有Guest、Reporter、Developer、Master、Owner 5种，默认设为Developer，具体各角色权限如下所示：

* Owner：项目所有者，拥有所有的操作权限。
* Master：项目的管理者，除更改、删除项目元信息外其它操作均可。
* Developer：项目的开发人员，做一些开发工作，对受保护内容无权限。
* Reporter：项目的报告者，只有项目的读权限，可以创建代码片断,不能提交代码，不能创建分支。
* Guest：项目的游客，只能提交问题和评论内容。





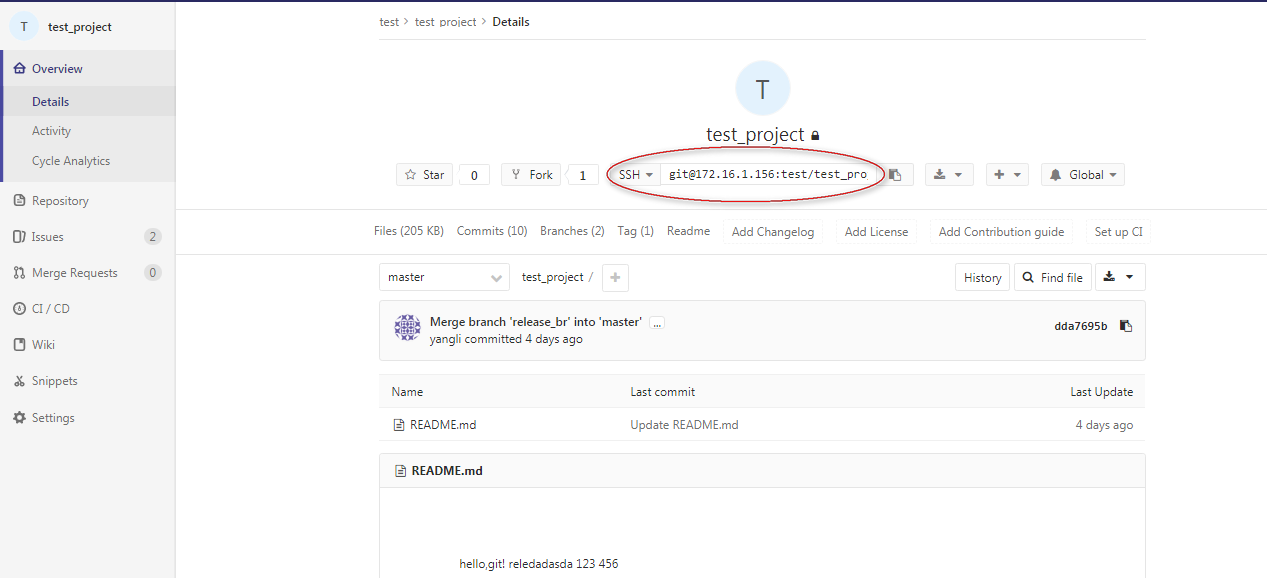
组权限和项目权限说明：

如果在组里给某成员添加了权限如developer，则该成员会在该组下的所有项目都继承developer权限。而如果再单独针对项目对该成员进行权限设置，比如调整为master，则他在这个项目里就是master权限。

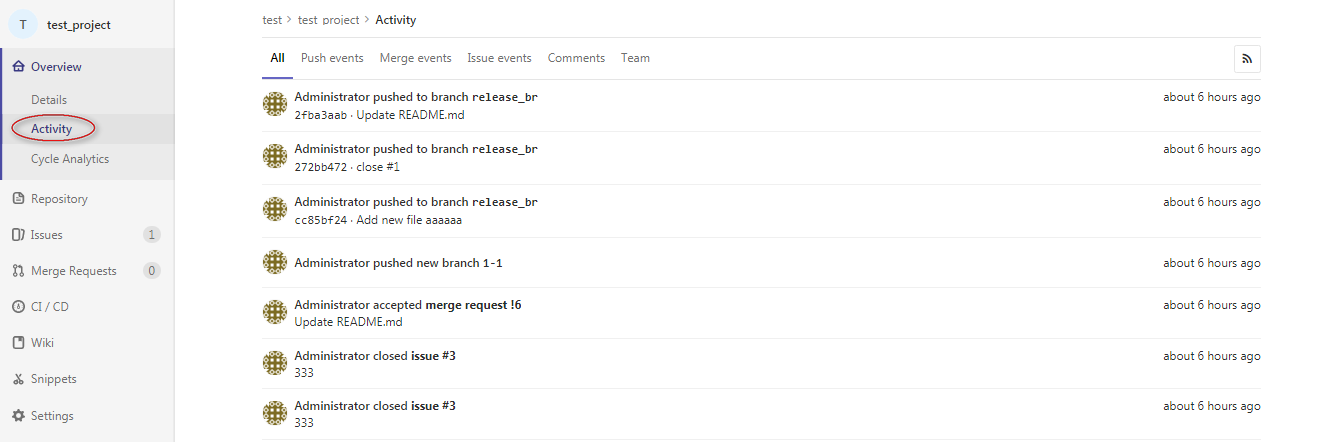
## 2.2查看项目详情

### 2.4.1 Overview

用户可以通过首页 projects查看所有参与的项目，点击进入相应的项目，选中【Overview】。【Details】下可以看到本项目的SSH地址和提交次数、分支个数等基本信息。

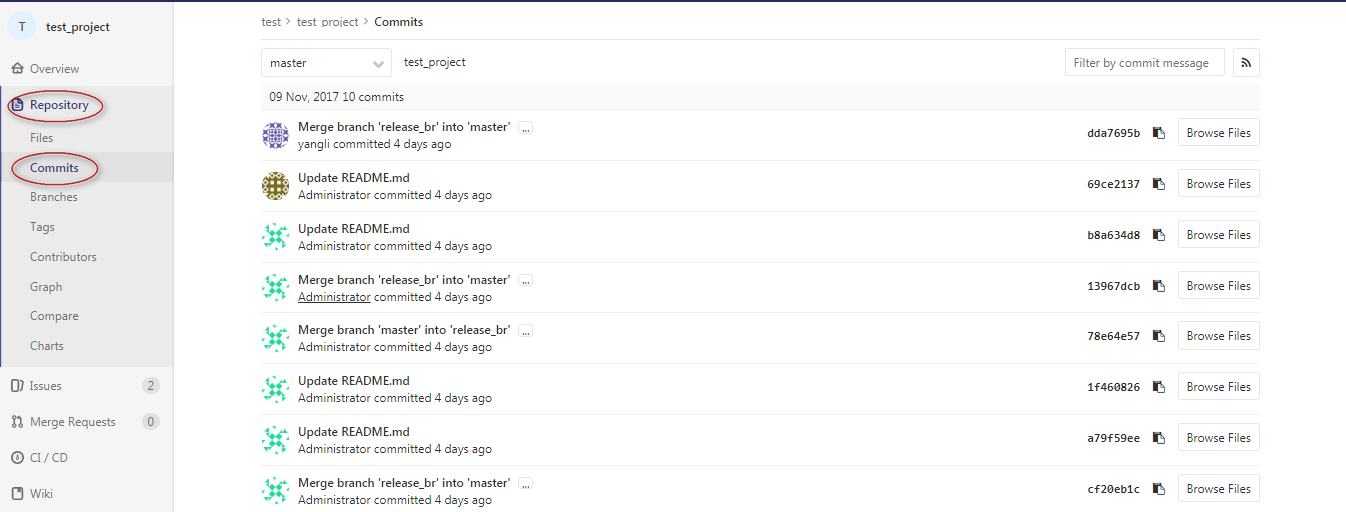


【Activity】下可以看到代码的提交记录、merge记录、问题单记录、评论记录等信息。



### 2.4.2 Repository

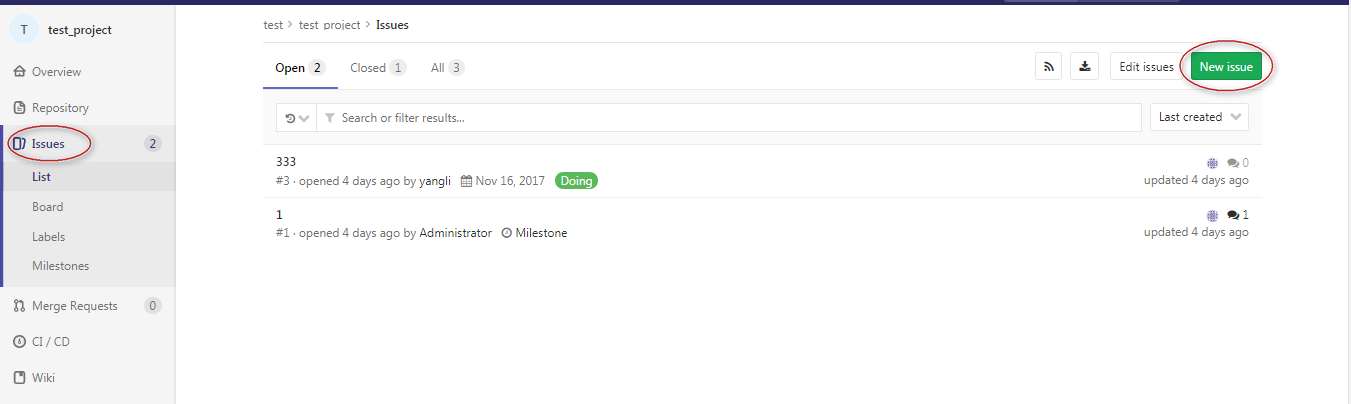
点击项目左侧栏的【Repository】可以查看项目包含的文件（Files）、提交记录（Commits）、分支（Branches）、标签（Tags）、版本树（Graph）、两个分支之间的代码差异比较（Compare）等信息。



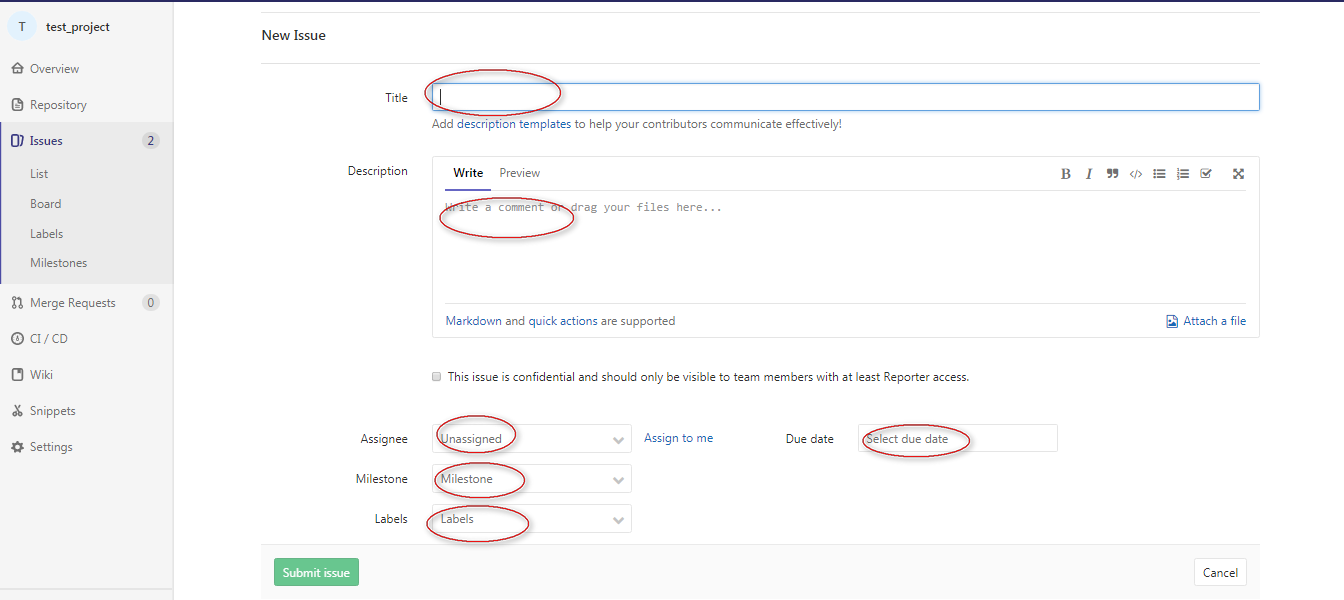
### 2.4.3 Issues

GitLab问题跟踪器可以用来跟踪解决bug的过程或feature开发的过程。

（1）创建issue

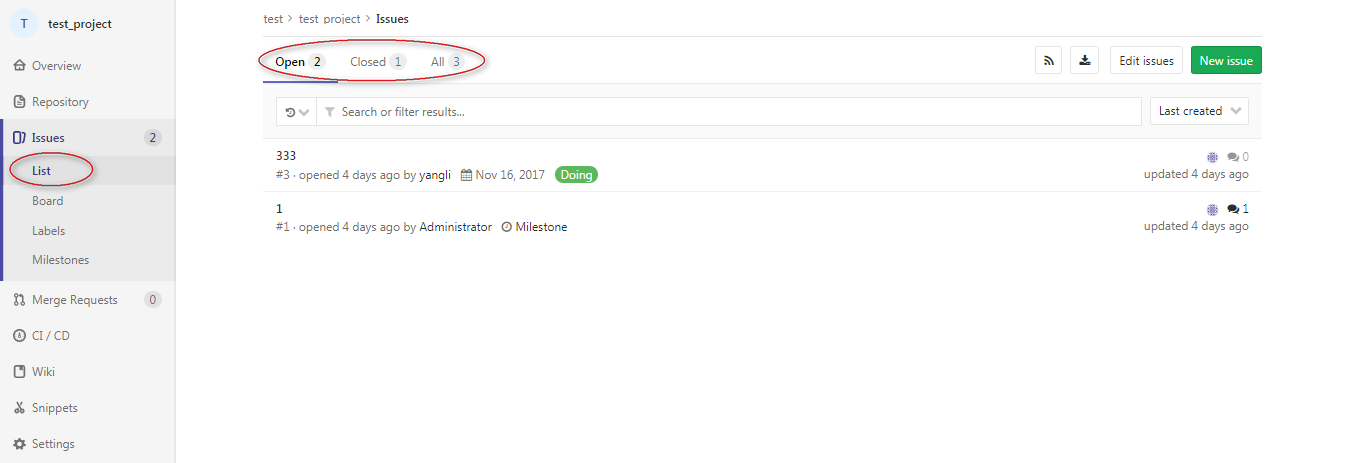


填写Issue Title，Description对问题单进行描述，Assignee指派问题单，Milestone里程碑。



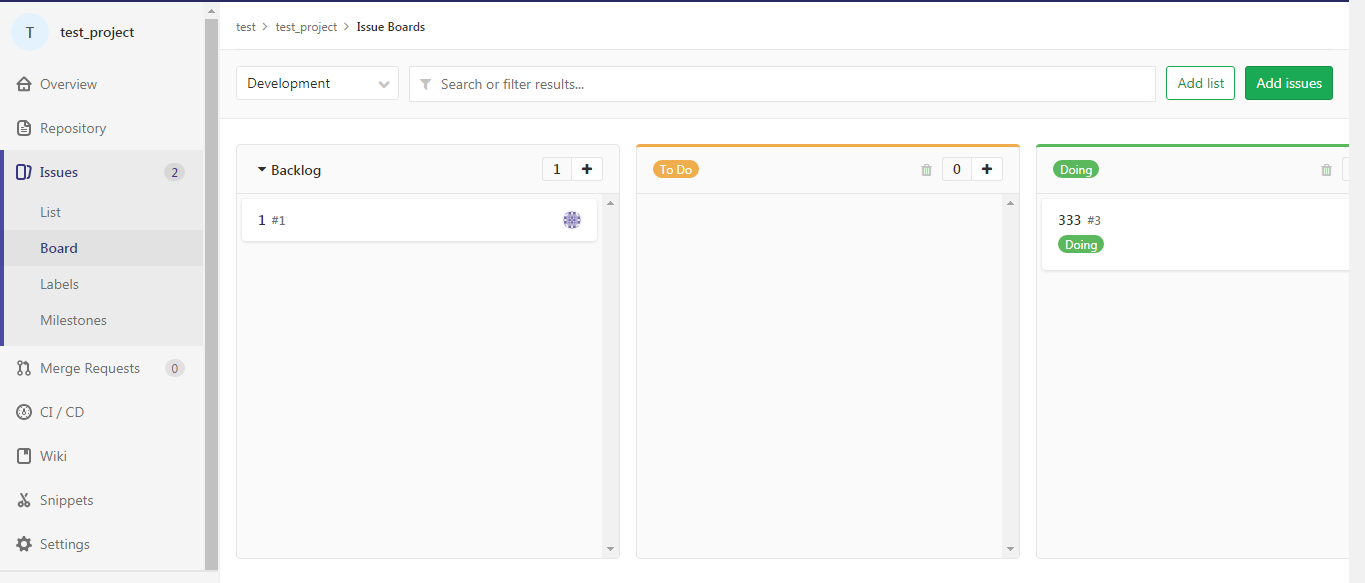
（2）List

当访问一个项目的Issues时，GitLab会将它们显示在一个列表中，可以使用可用的选项卡来快速筛选和关闭问题。



（3）Board

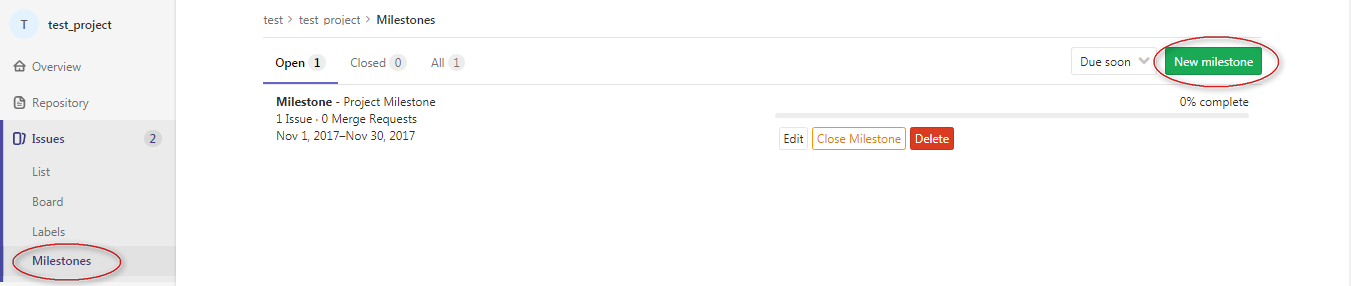
Board是一个看板，通过Board可以很直观的看到处于各个状态的问题单有多少，也可以很方便的拖拽问题单对状态进行变更。这里的“To Do”、“Doing”等状态标签可以根据需要自定义。



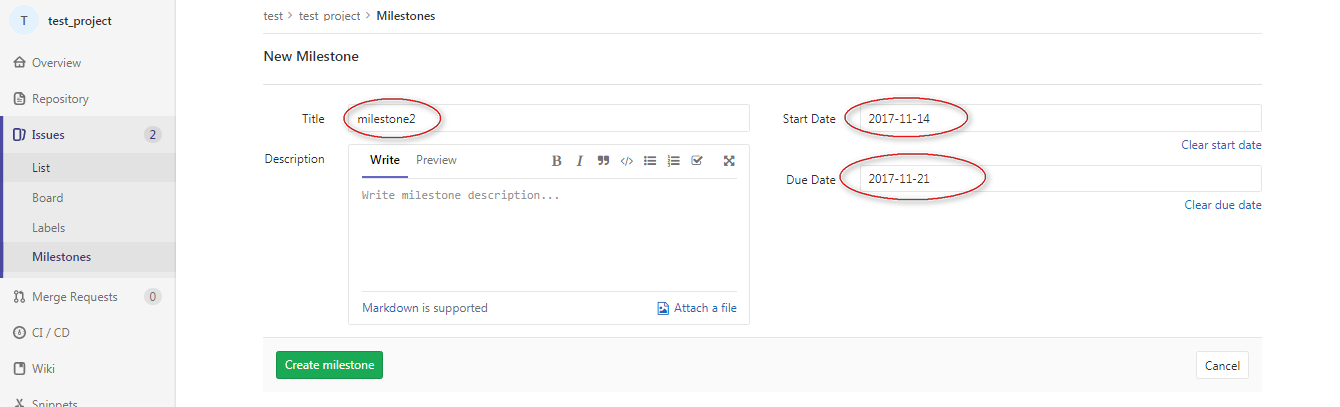
（4）Milestone

创建项目里程碑，评估工作量和时间。

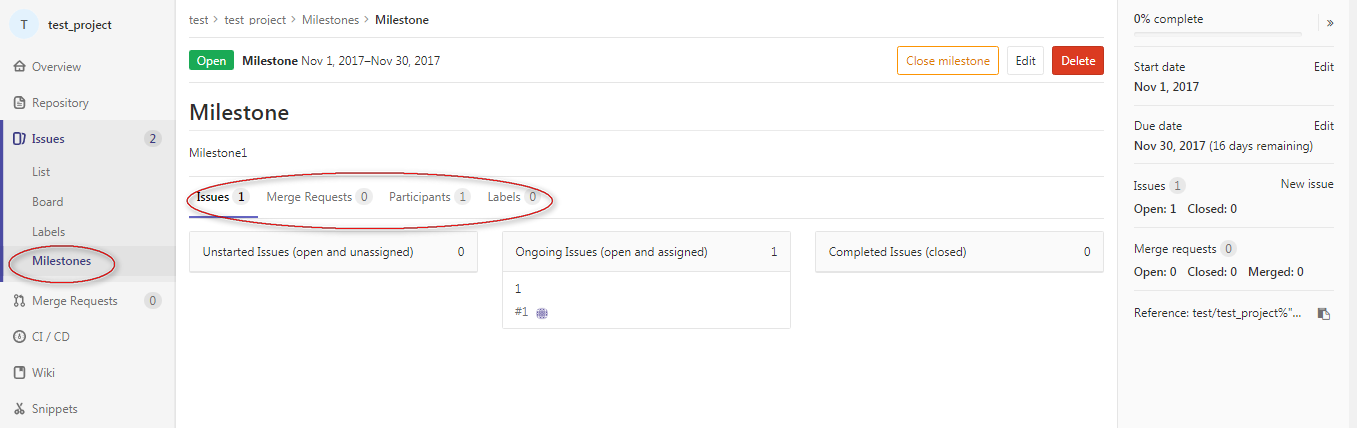
选择【Issues】下的【Milestone】，可以看到当前项目所有的milestone。



填写Title，开始日期，结束日期，创建一个新的milestone。



如选择进入milestone1可以看到这个milestone下所有的Issues、Merge requests等信息。

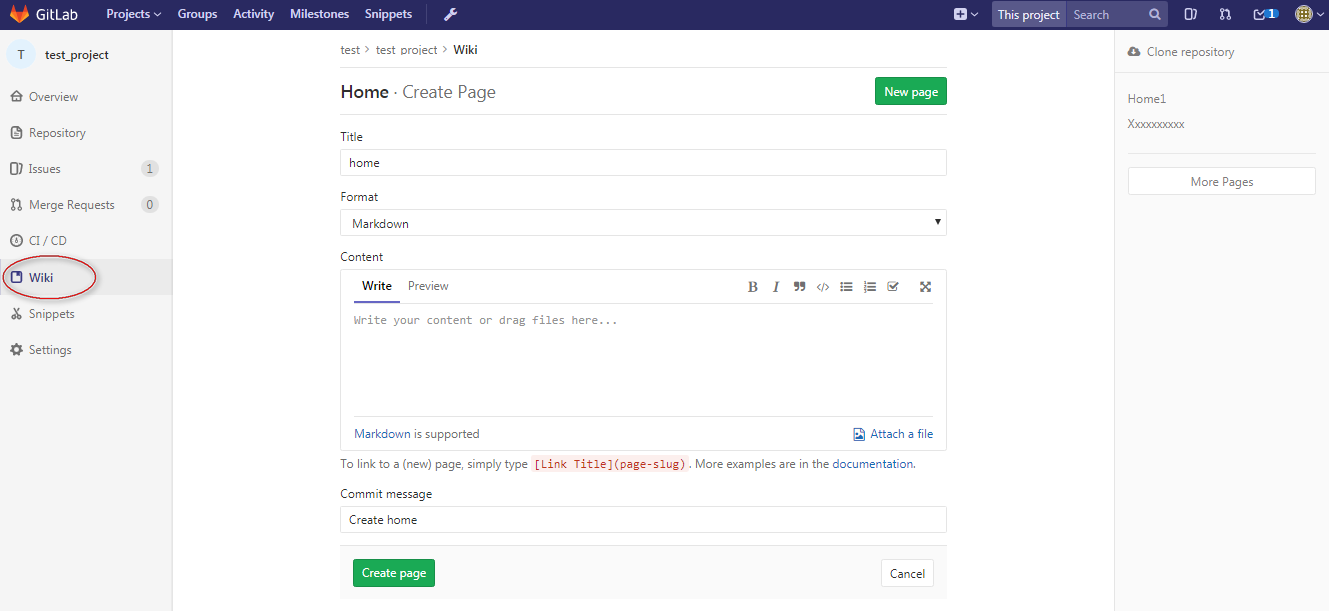


（5）Issues如何和commit关联

当提交一个commit的时候在commit message里面使用#issue, 比如#8, gitlab就会自动关联issue 8和这个commit。那么如何跟随着commit关闭一个issue呢? 当一个Merge Request解决了一个issue时，并且在Merge Request的Description里写了如下语法格式的命令：fixes #xx、fixed #xx、fix #xx、closes #xx、close #xx、closed #xx，假如这个merge request被接受，则这个issue将会自动被关闭。

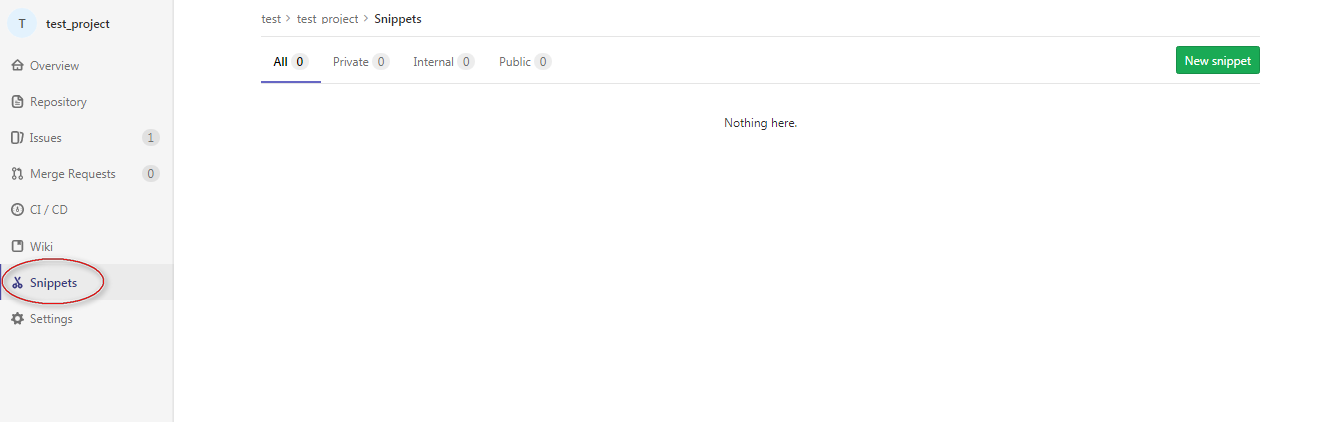
### 2.4.4 Wikis

WiKi 可以存放项目的共享文档等。



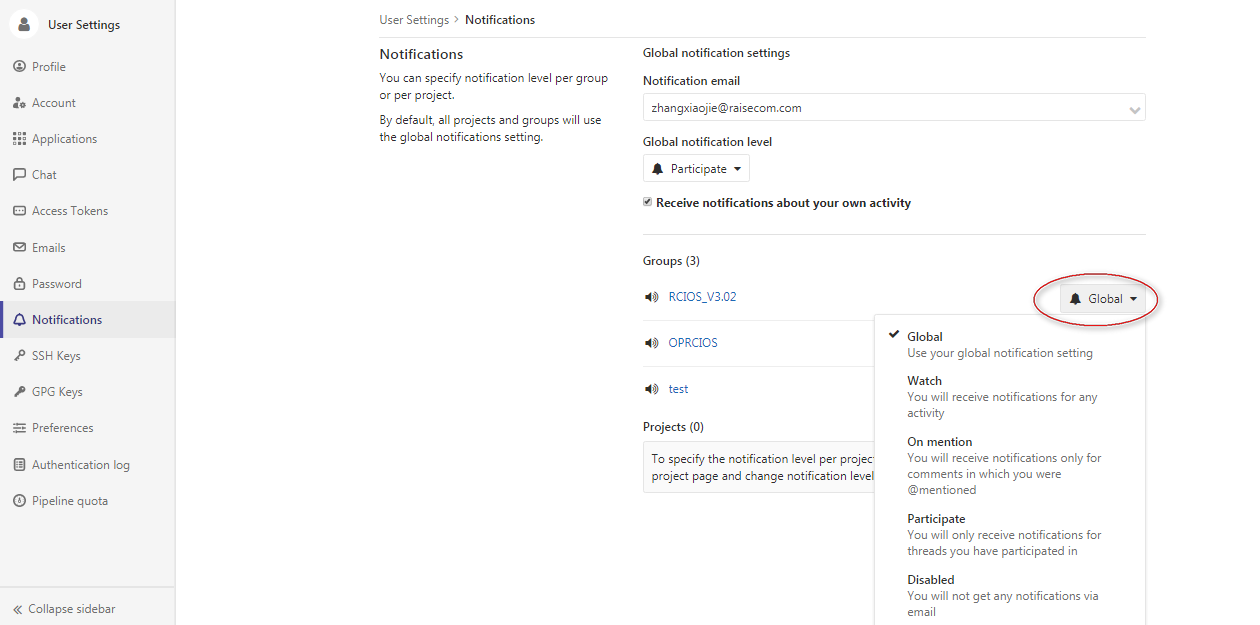
### 2.4.5 Snippets

GitLab的代码片段可以是私有的，也可以是内部的，也可以是公开的。每个人都可以将自己常用的代码片段保存到系统并分享给大家。



### 2.4.6 Notifications

在个人账户Profile Settings下选中Notifications，可以设置消息提醒级别和范围。消息可以在项目的activity中看到。



# 3 Git工作流程

## 3.1本地工作

先进行git全局设置

git config –global user.name "xxxxx"

git config –global user.email "xxxxxx"

工作方式举例：

**贺永芳组工作方式（基于master分支开发）**

以master分支举例，如锁定master分支，保持本地分支一直和远程master同步

**流程：**基于远程master创建本地分支<branchname>，修改完之后将本地分支推送到远程<branchname>，创建merge request

**git操作命令：**

**（1）下载代码到本地**

git clone git@192.168.31.76:<groupname>/<projectname>.git  （下载的是master分支代码）

举例：git clone git@192.168.31.76:RCIOS/RCIOS\_V3.01.git

如果想切换其他分支代码，在执行git clone之后，执行如下命令

git checkout <branchname>

**（2）本地创建工作分支**

在自己本地基于远程master分支创建一个<branchname>分支，并切换到该分支，这样本地<branchname>就和远程master分支建立了连接关系

git checkout -b <branchname> origin/master

**（3）本地修改后提交**

本地修改后提交

git status   查看本地工作区状态

git add <filename>     添加修改文件

git commit -m "message"    提交修改文件

**（4）拉取远程更新到本地**

推送到远程前拉取master最新到本地，因为本地分支已经建立了和master的连接关系，可以直接使用git pull拉取master最新到本地。若有冲突则编辑修改，再次提交。

git pull

**（5）推送本地修改到远程**

推送本地修改到远程<branchname>分支，如果远程没有这个分支则会创建一个<branchname>

git push origin <branchname>

**（6）merge request**

最后登录gitlab网页，创建一个merge request，指定合并人，如果需要多人评审，则在Discription里@评审人。

**官方推荐工作方式：多分支开发**

**master 主分支**

**develop 开发分支**

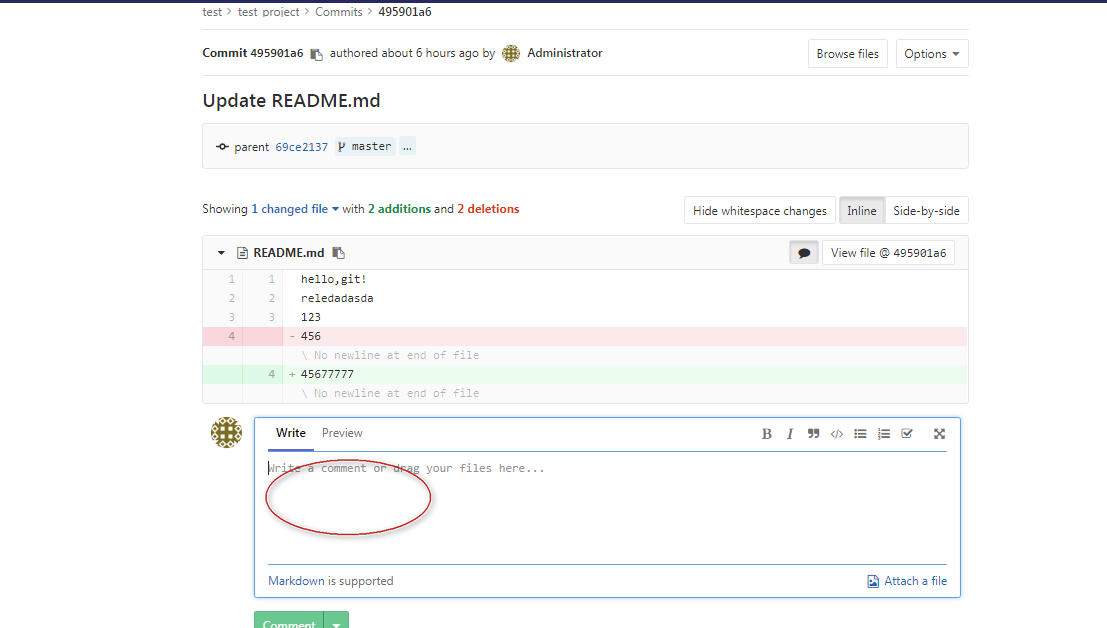
**feature-功能开发分支**

**bug- bug修改分支**

**release- 发布分支**

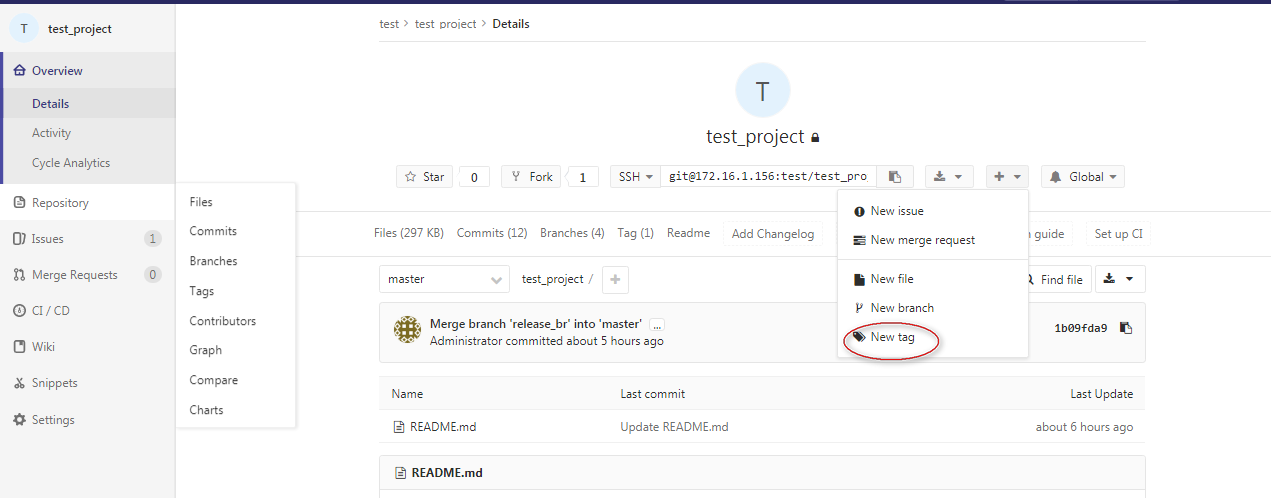
## 3.2添加评论

选择项目，选择左侧栏“Commits”，选中需要添加注释的提交代码，可以对他人提交的代码添加comment，可以针对提交的每一行添加comments。

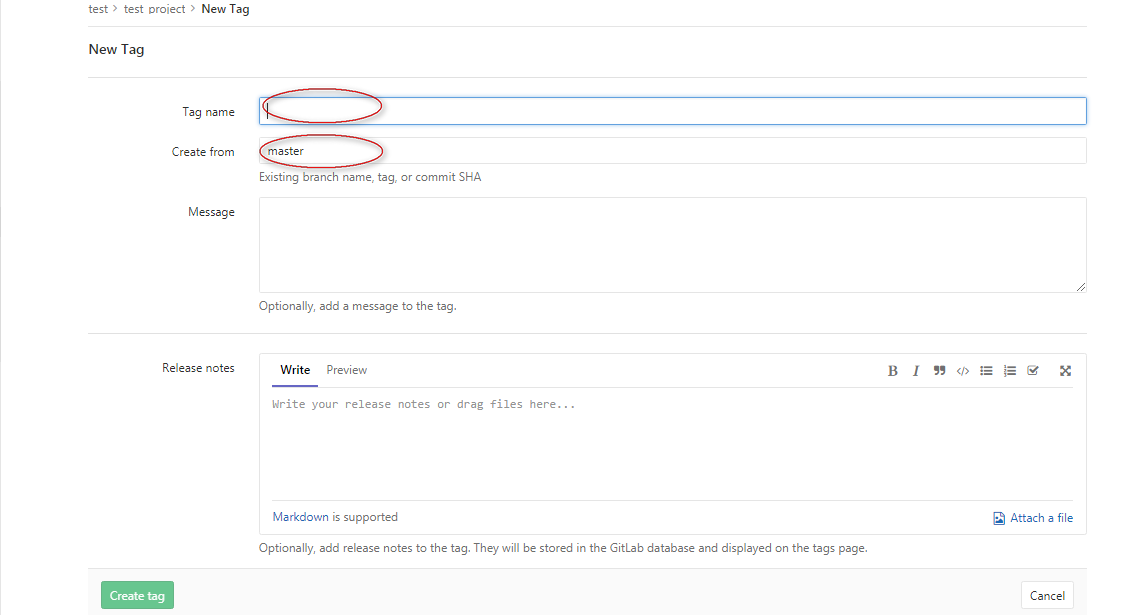


## 3.3新建标签

在下图中Project主页下“+”下拉框中选择new tag，如下所示。

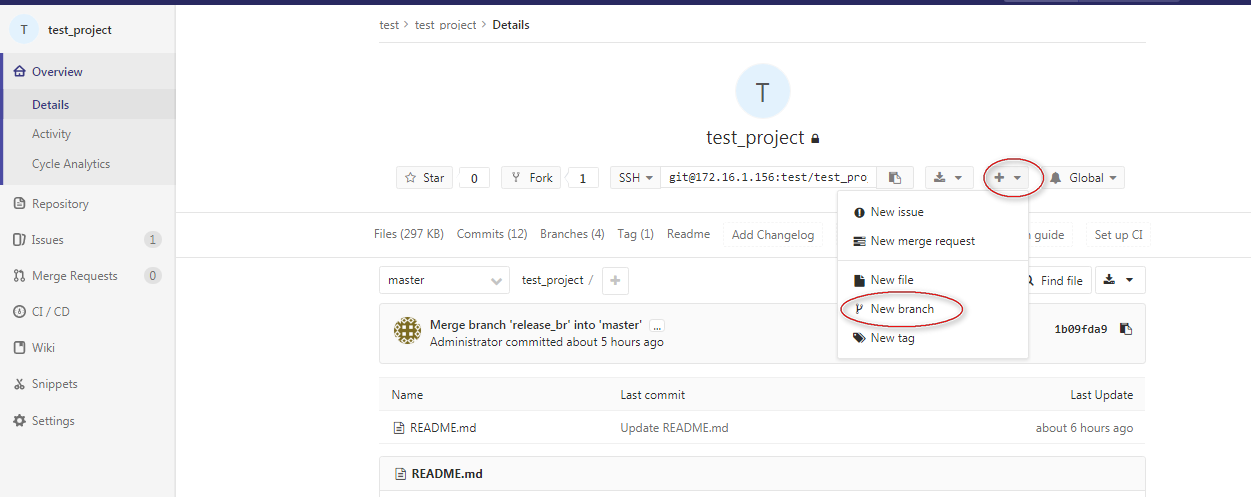


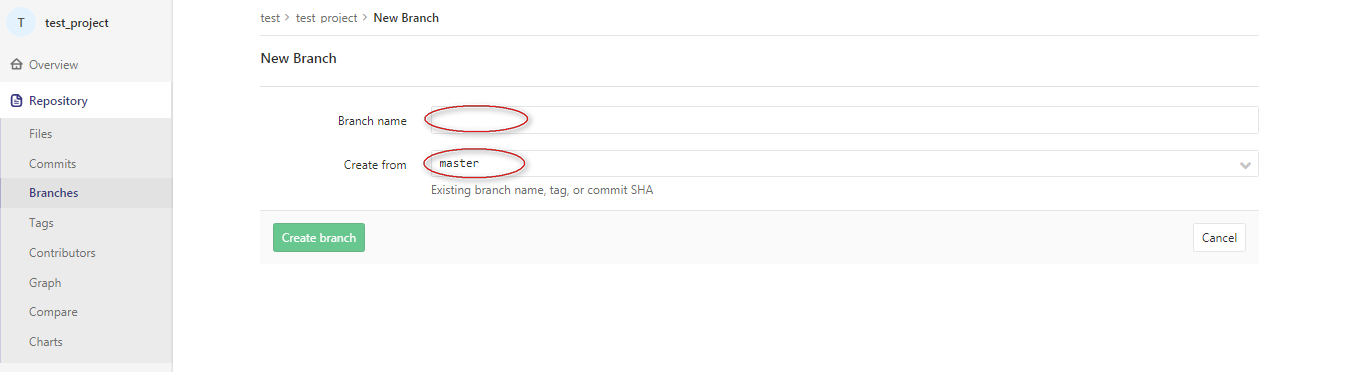
填写Tag name，选择要打标签的分支或提交记录。



## 3.4新建分支

可以在下图中Project主页下“+”下拉框中选择new branch，填写新分支名称，选择源分支，如下所示。





## 3.5保护分支和保护标签

### 3.5.1 保护分支

gitlab默认一个项目的master分支是protected branch。对于protected branch，developer默认不能再直接提交代码，只有Master和Owner有权限提交。

Master和Owner有权限添加或取消protected branch，同时Master和Owner可以根据需要将一个protected branch设置为允许Developer提交。

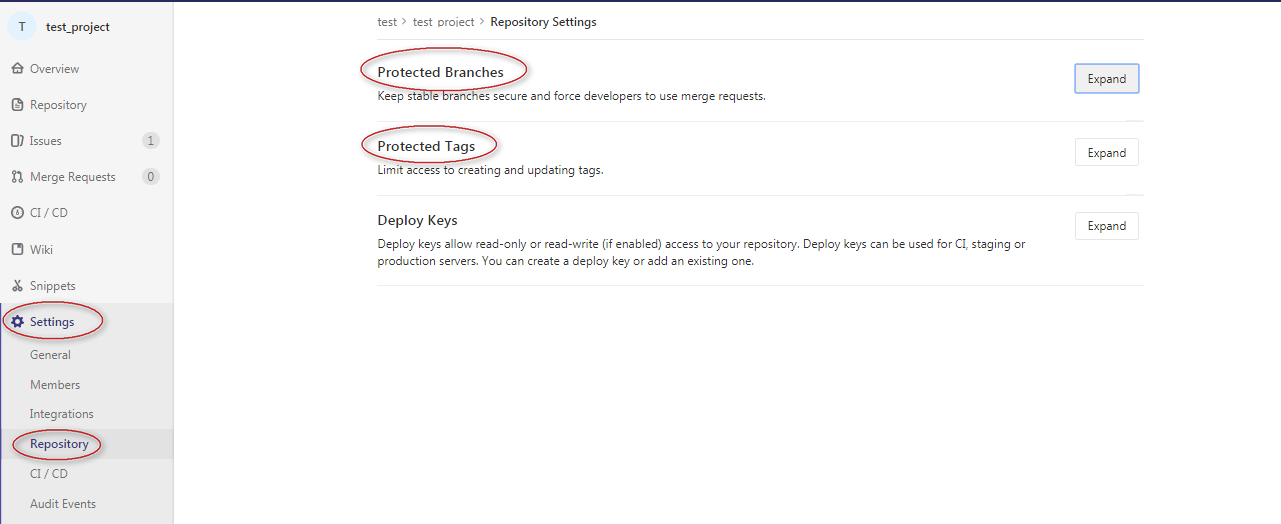
使用需求：

1、版本正式发布后为避免开发者继续在原发布版本分支上提交代码，这种情况下可以锁定分支。如果要继续基于此分支开发，可重新拉出一个分支。

2、为了控制版本的发布时间和质量，限制开发者的提交。

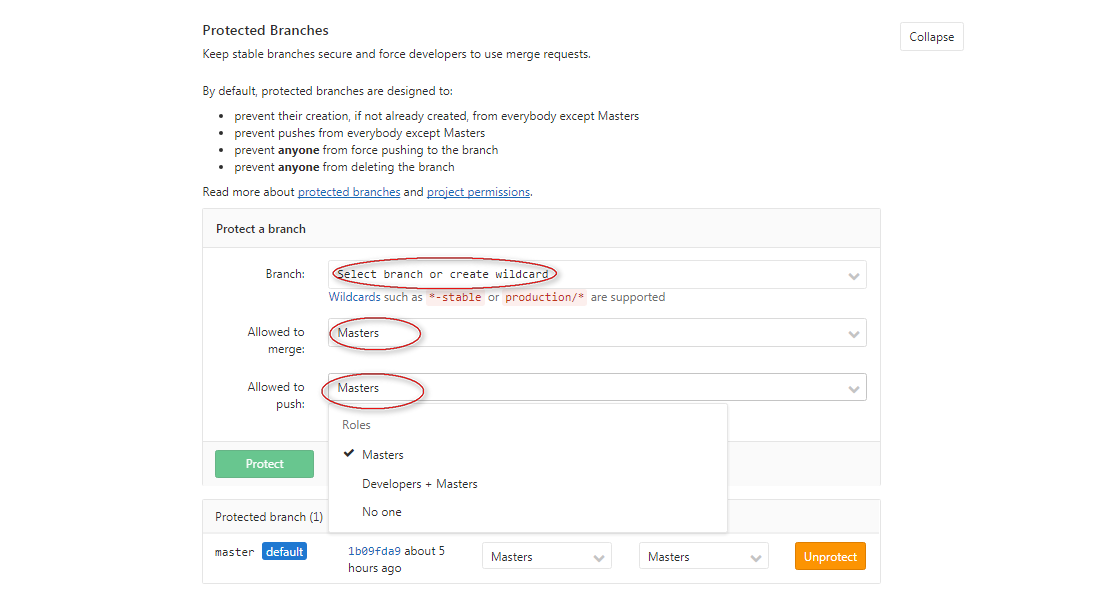
具体操作如下：

在Projects主页左下方点击【Settings】，点击【Repository】，选择【Protected branches】，如下图所示。

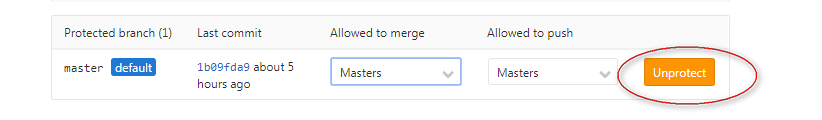


点击【Expand】进入保护分支设置界面，如下图所示。

选择需要保护的分支，设定允许merge的角色和允许push的角色，只有三种选项，Masters、Developers+Masters、No one，默认选择Masters。



如果要取消保护状态，在Protected Branches下选中分支，点击右侧Unprotect，即取消了被保护状态。



### 3.5.2 保护标签

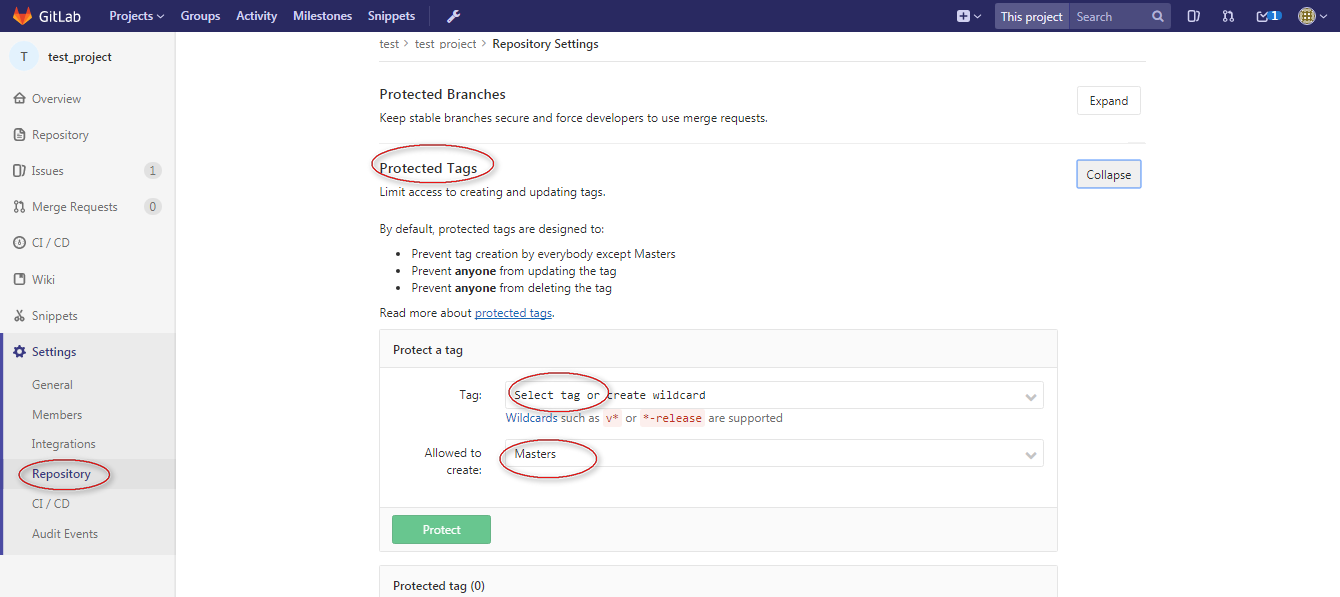
受保护的tag将防止任何人更新或删除标签，并将根据你所选择的权限防止创建匹配标签。默认情况下，没有Master权限的任何人都不能创建tag。

另外可以根据项目需要指定一个通配符保护标签，它将保护所有匹配通配符的tags。

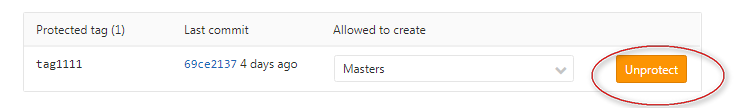
具体操作如下：

在Projects主页左下方点击【Settings】，点击【Repository】，选择【Protected tags】，点击【Expand】进入保护标签设置界面，如下所示。

选择需要保护的tag，或设定通配符，设定允许create的角色，只有三种选项，Masters、Developers+Masters、No one，默认选择Masters。

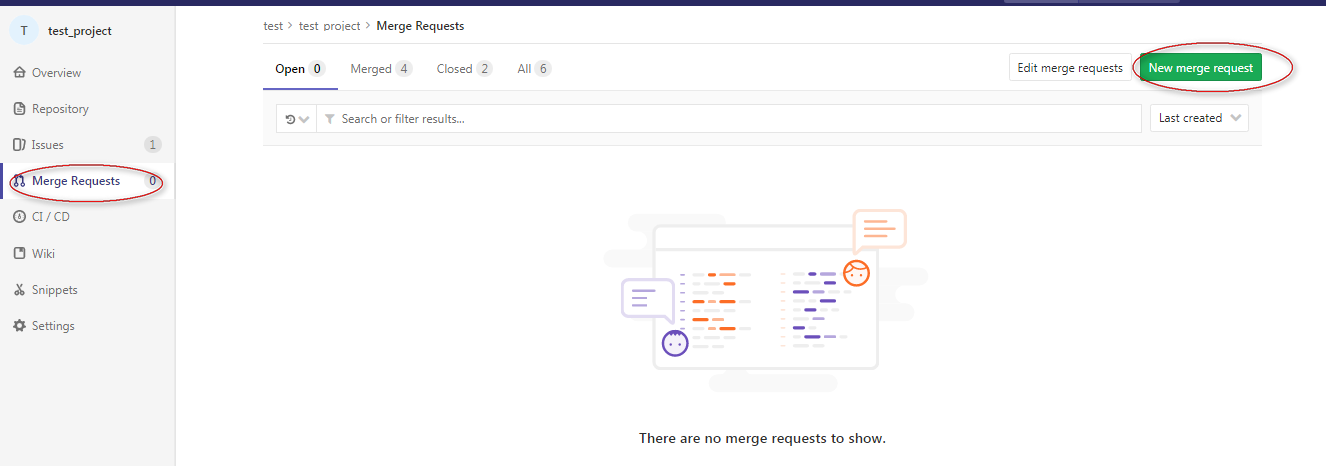


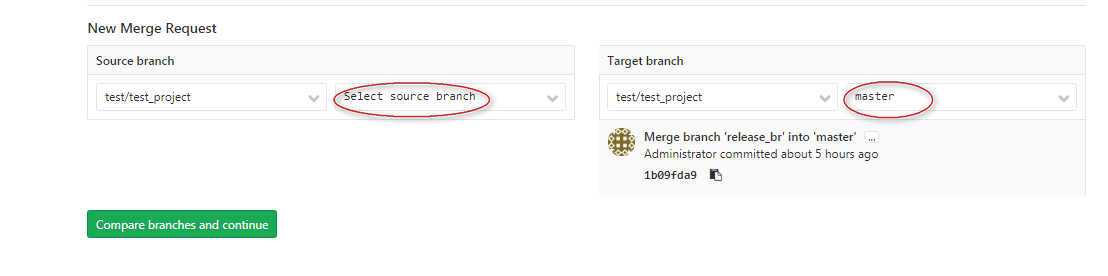
如果要取消保护状态，在Protected tag下选中tag，点击右侧UNPROJECT，即取消了被保护状态。

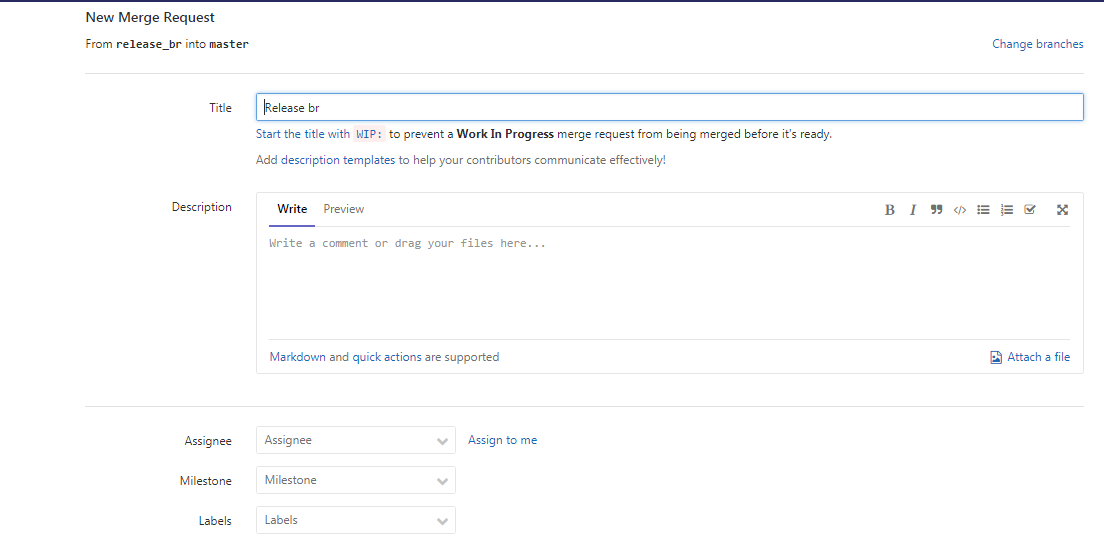


## 3.6新建合并请求

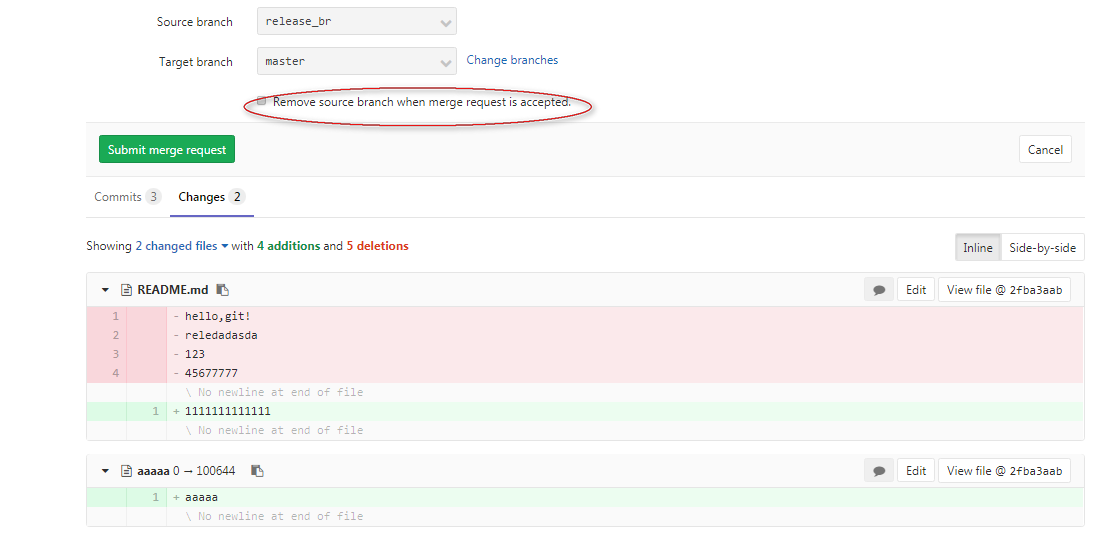
选择项目，点击左侧栏的Merge Requests，在右上方将出现“New merge request”的绿色按钮，选中，需要填写“soure branch”和“target branch”和评审人。如下图所示。





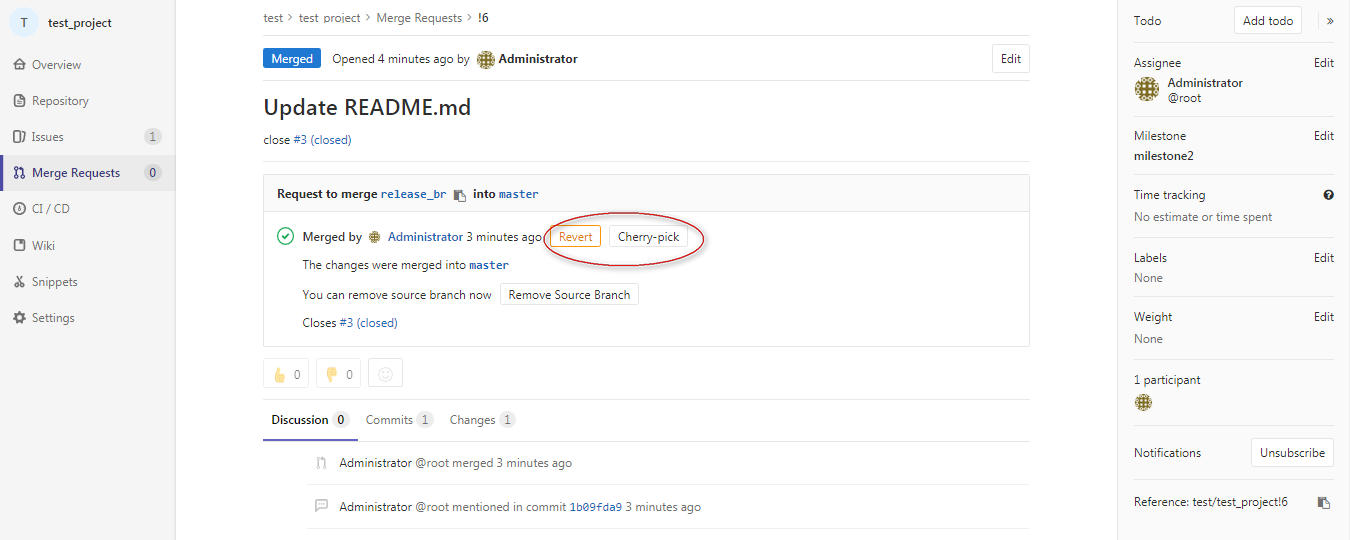


在Submit merge request下方有Commits和Changes信息，可以看到两个分支之间的差异，提交时可以选择当合并被接受的时候删除source branch以免分支过多。



当一个合并请求接受后，将有一个cherry-pick按钮来选择合并请求所引入的更改。单击cherry-pick按钮后，可以选择将更改直接选择到选择的分支，或者可以选择使用cherry - pick更改创建一个新的合并请求。

同时当一个合并请求接受后，也会有一个revert按钮来恢复合并请求所引入的更改。单击该按钮后，可以选择将更改直接恢复到所选的分支，或者可以选择使用恢复更改创建一个新的合并请求。



建议：

如果你在一个feature分支工作比较长时间，并且没有准备好merge，但是又希望大家来提意见，此时可以建立merge request而不指向任何人。团队成员可以填写comment提出意见，这就是将merge requests作为code review工具来使用。如果review时发现了一些问题，你修改完以后push了一个新的commit到source branch， merge request的代码也将会自动更新。

当你觉得合适了，就把merge request指定给评审人，提醒他来查看代码给你反馈。如果别人觉得你的代码不太好，他可以直接关掉merge request。

## 3.7代码审查

通过锁定分支和merge request实现强制code review。

使用需求：

把控提交代码的质量，代码必须审核通过后才允许提交。

具体操作说明：

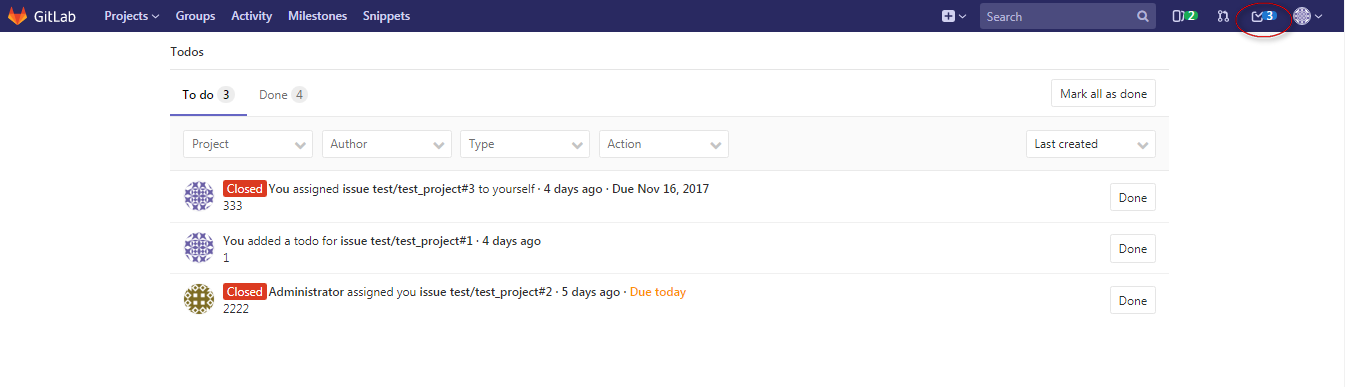
举例：为了把控提交的代码，现将master分支锁定，不允许developer随意提交代码。开发者必须在本地拉一个分支开发，开发完成后在gitlab页面上发出merge request，相关负责人审批通过后才能merge到master分支。

## 3.8待办事项

当你登录GitLab时，通过Todos可以很方便的看到有哪些需要你留意的地方。

Todos是一个按时间顺序排列的待办事项列表，当一个问题或合并请求分配给你，或你在一个问题或合并请求中被@到，在你的Todos面板里将会新增一个待办事项。

Todos是私人性质的， 如果其他人关闭问题或合并请求，你的任务将仍然出现在待办事项里。这可以防止其他用户在没有通知你的情况下关闭问题或合并请求。每个问题或合并请求只有一个Todo，所以提到一个用户在一个问题上，只会触发一个Todo。



# 4 git常用命令汇总

## 4.1取得项目的git仓库

### 4.1.1在工作目录中初始化新仓库

要对现有的某个项目开始用 Git 管理，只需到此项目所在的目录（可以是空目录也可以是非空目录），执行：

$ git init

初始化后，在当前目录下会出现一个名为 .git 的目录，所有 Git 需要的数据和资源都存放在这个目录中。

如果当前目录下有几个文件想要纳入版本控制，需要先用 git add 命令告诉 Git 开始对这些文件进行跟踪，然后提交。

### 4.1.2从现有仓库克隆

git clone是把服务器上代码仓库完整地下载下来。

所以，即使服务器的磁盘发生故障，用任何一个克隆出来的客户端都可以重建服务器上的仓库，回到当初克隆时的状态。

git clone命令：

$ git clone git@<url>:<group name>/<project name>.git

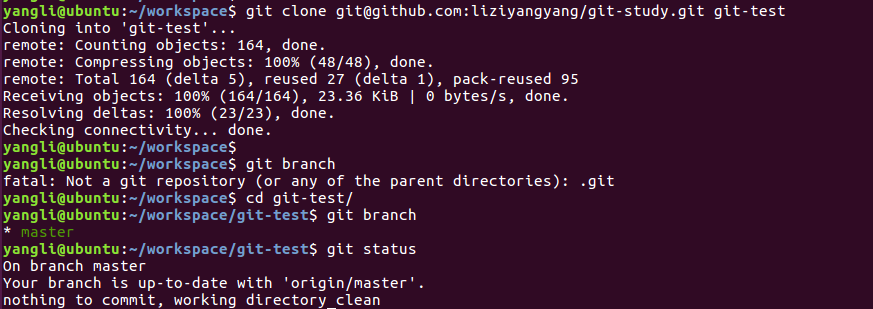
如果希望在克隆的时候，自己定义要新建的项目目录名称，可以在上面的命令末尾指定新的名字。

执行完clone命令后这会在当前目录下创建一个名为<project name>的目录，其中包含一个 .git 的隐藏目录，它是 Git 用来保存元数据和对象数据库的地方。

执行git branch查看，本地只有一个master分支，因为git clone默认会在本地创建一个对应于远程默认分支的本地分支，一般都是master分支，并取出master分支的最新版本文件到本地，这些文件都是从git目录中的压缩对象数据库中提取出来的。

注：远程默认分支可修改。

\*所指就是当前所在分支

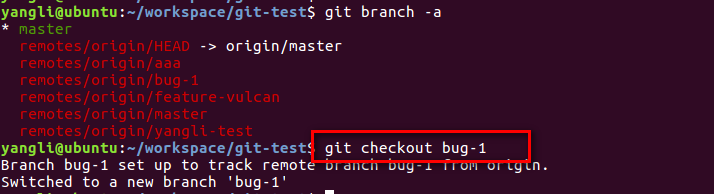


如果想切换到其他分支

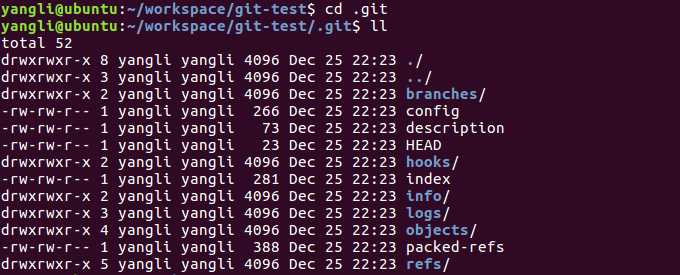
执行git branch –a可以查看到本地和远程所有的分支，如果想切换到bug-1分支，可以直接执行git checkout bug-1，就切换到bug-1分支了，并会同时建立本地bug-1分支和远程bug-1分支的跟踪关系。

如果想基于远程remotebranch分支创建一个本地localbranch分支并切换到loaclbranch，执行下述命令：

git checkout –b loaclbranch origin/remotebranch



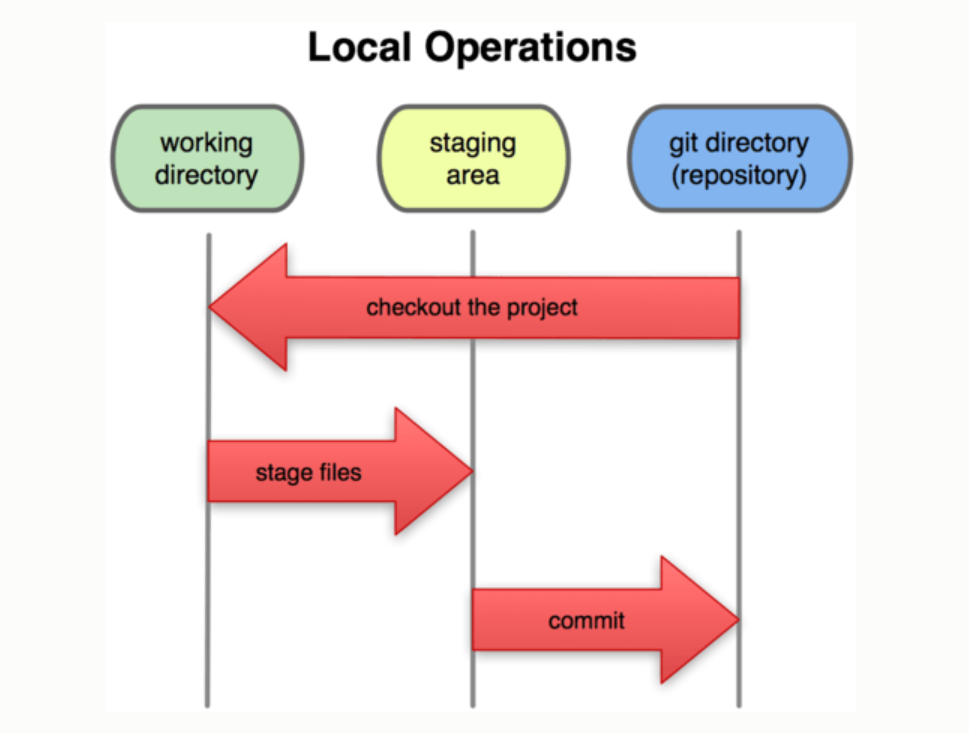
.git目录结构



## 4.2记录每次更新到仓库

### 4.2.1文件流转

文件流转的三个工作区域：Git 的工作目录，暂存区域，以及本地仓库



工作目录下面的所有文件都不外乎这两种状态：已跟踪或未跟踪。已跟踪的文件是指本来就被纳入版本控制管理的文件，在上次快照中有它们的记录，工作一段时间后，它们的状态可能是未更新，已修改或者已放入暂存区。而所有其他文件都属于未跟踪文件。它们既没有上次更新时的快照，也不在当前的暂存区域。初次克隆某个仓库时，工作目录中的所有文件都属于已跟踪文件，状态为未修改。

### 4.2.2 git status

要确定哪些文件当前处于什么状态，可以用 git status 命令。

“status”是一个常用的命令，用来查看工作区状态，工作目录下新添加文件的状态是untracked，其它文件修改后的状态是not staged，这两种状态的文件都是需要添加到暂存区以便提交到仓库的，暂存区下的文件状态可以是new或者modified。

untracked未跟踪的文件意味着Git在之前的提交中没有这些文件；Git 不会自动将它纳入跟踪范围，除非你明明白白地告诉它“我需要跟踪该文件”，因而不用担心把临时文件什么的也归入版本管理。

只要在 “Changes to be committed” 这行下面的，就说明是已暂存状态。如果此时提交，那么该文件此时此刻的版本将被留存在历史记录中。

### 4.2.3 git add

$ git add <file>

添加单个文件。

$ git add <file> <file2>

添加多个文件。

$ git add -A

添加所有文件。

$ git add --all

添加所有文件。

$ git add -u

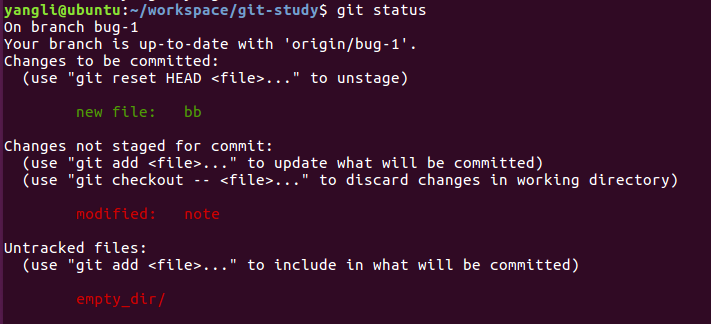
提交被修改(modified)和被删除(deleted)文件，不包括新文件(new)

$git add .

提交新文件(new)和被修改(modified)文件，不包括被删除(deleted)文件

在 git add 后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下的所有文件。

git add 命令根据目标文件的状态不同，此命令的效果也不同：可以用它开始跟踪新文件，或者把已跟踪的文件放到暂存区，还能用于合并时把有冲突的文件标记为已解决状态等。



### 4.2.4 git commit

$ git commit -m “comment”

“comment”是本次提交的注释，配合“-m”参数，提交成功后会产生一个新的commit id。

$ git commit -a -m “comment”

“-a”等同于“--all”，这个参数的意思是自动暂存，也就是说已经处于暂存区的文件如果被修改了，不用进行传统意义上的先“git add”后“git commit”，而是直接“git commit -a”即可。

$ git commit --amend -m “comment”

“--amend”参数的意思是追加一次提交，不产生新的commit id。

在提交之前，请一定要确认还有什么修改过的或新建的文件还没有 git add 过，否则提交的时候不会记录这些还没暂存起来的变化。所以，每次准备提交前，先用 git status 看下，是不是都已暂存起来了，然后再运行提交命令 git commit –m “message”，则将本地暂存区修改提交到了本地仓库。



可以看到，提交后它会告诉你，当前是在哪个分支（master）提交的，本次提交的完整 SHA-1 校验和是什么（463dc4f），以及在本次提交中，有多少文件修订过，多少行添改和删改过。

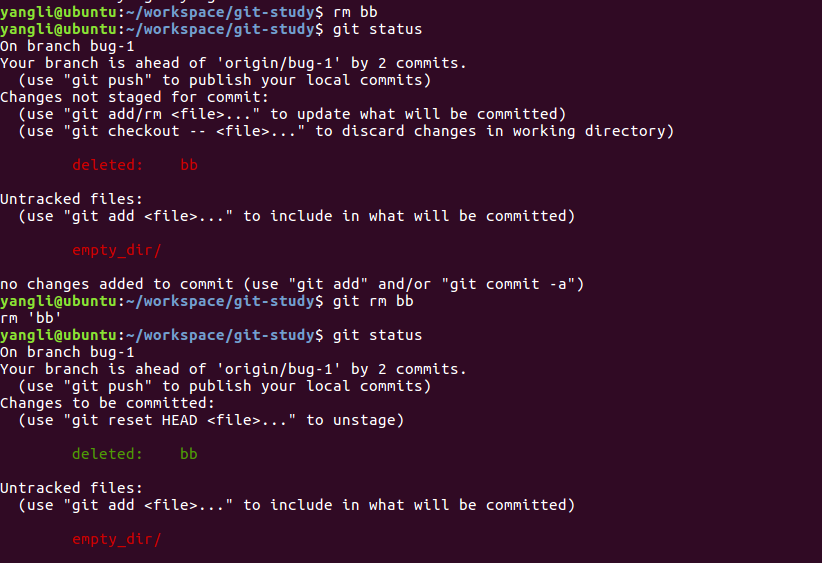
记住，提交时记录的是放在暂存区域的快照，任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对你项目作一次快照，以后可以回到这个状态，或者进行比较。

建议：随时执行git status查看工作区状态

### 4.2.5 git rm

要从 Git 中移除某个文件，就必须要从已跟踪文件清单中移除（确切地说，是从暂存区域移除），然后提交。可以用 git rm 命令完成此项工作，并连带从工作目录中删除指定的文件，这样以后就不会出现在未跟踪文件清单中了。

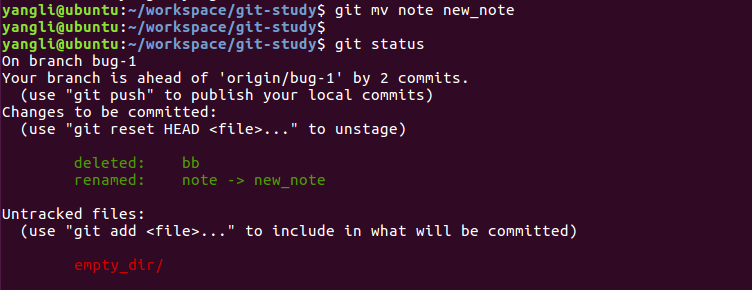
如果只是简单地从工作目录中手工删除文件，运行 git status 时就会在 “Changes not staged for commit” 部分（也就是未暂存清单）看到。



再执行git rm之后等最后提交的时候，该文件就不再纳入版本管理了。

### 4.2.6 git mv

重命名命令：git mv old\_file new\_file



## 4.3 Git撤消操作

### 4.3.1 撤消工作目录修改

$ git checkout <file>

撤销单个文件的修改。

$ git checkout <file> <file2>

撤销多个文件的修改。

### 4.3.2 撤消暂存区修改

$ git reset HEAD <file>

撤销单个文件的修改。

$ git reset HEAD <file> <file2>

撤销多个文件的修改。

$ git reset HEAD

$ git reset HEAD --soft

$ git reset HEAD --hard

撤销所有文件的修改，这种情况下可以使用“--soft”或“--hard”参数，默认是“--soft”模式，撤销并保留修改，“--hard”则是撤销并删除修改。

上面的“HEAD”是最近一次提交到仓库的状态，前面几次依次为“HEAD^”、“HEAD^^”等，如果是前面n次呢，那就显得很不方面了，不过可以这样表示“HEAD~n”，n是一个具体的数字，这个数字是需要我们自己算出来的，还是有点麻烦。“HEAD”其实就是个commit id，直接用这个id就方便多了。

回退前，用git log可以查看提交历史，以便确定要回退到哪个版本。

要重返未来，用git reflog查看命令历史，以便确定要回到未来的哪个版本。

### 4.3.3 撤消远程修改

对于已经把代码push到线上仓库,你回退本地代码其实也想同时回退线上代码,回滚到某个指定的版本,线上,线下代码保持一致.你要用到下面的命令

git revert用一个新提交来消除一个历史提交所做的任何修改.

此次操作之前和之后的commit和history都会保留，并且把这次撤销作为一次最新的提交   
git revert HEAD 撤销前一次 commit   
git revert HEAD^ 撤销前前一次 commit   
git revert commit-id (撤销指定的版本，撤销也会作为一次提交进行保存） 

git revert是提交一个新的版本，将需要revert的版本的内容再反向修改回去，版本会递增，不影响之前提交的内容。

## 4.4查看修改前后的异同

$ git diff <file>

查看单个文件修改前后的异同。

$ git diff <file> <file2>

查看多个文件修改前后的异同。

$ git diff

查看所有文件修改前后的异同。

单单 git diff 是显示还没有暂存起来的改动。所以有时候你一下子暂存了所有更新过的文件后，运行 git diff 后却什么也没有，就是这个原因。

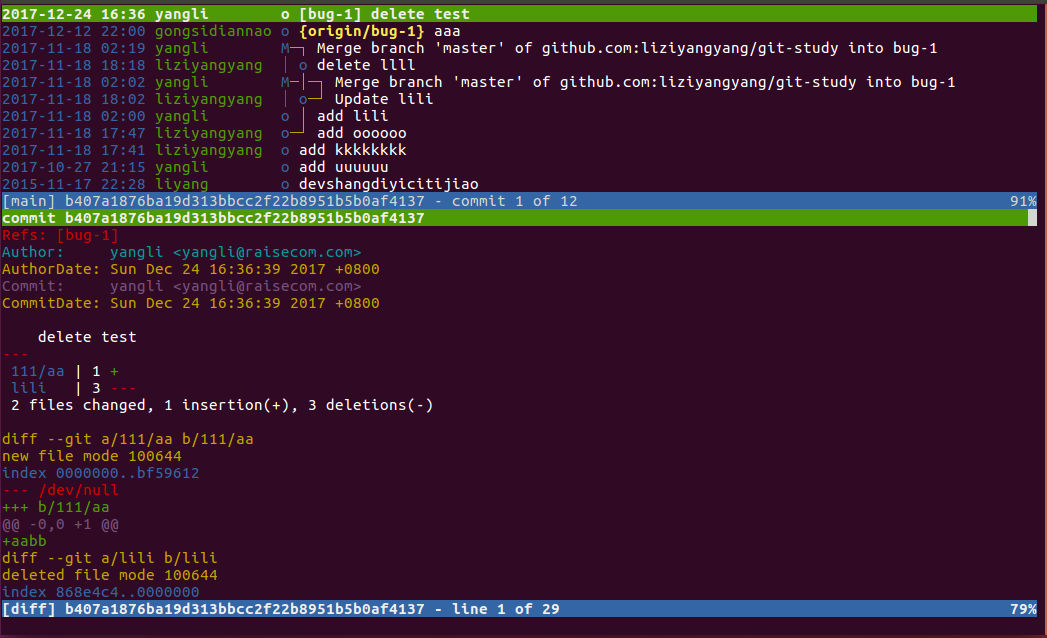
$ git diff <commit id> <file>

这个对比是在本次修改和某次提交之间进行的。

若要看已经暂存起来的文件和上次提交时的快照之间的差异，可以用 git diff --cached 命令。

推荐使用图形化工具tig

安装方式 sudo apt-get install tig



## 4.5查看历史提交信息

$ git log

显示所有的提交记录，包括commit id、提交者、提交时间、注释。

$ git log --pretty=oneline

“--pretty=oneline”参数的好处是在一行上显示每次的提交记录，包括commit id、注释。

$ git log --graph

“--graph”参数在屏幕左边显示每次提交之间的关系。

（1）按数量查找提交信息

限制显示的提交数量，如想显示最新的3次提交

$git log -3

（2）按日期查找

如果想查找某个特定时间段的提交，可用使用—after或—before标记来按日期查询，比如查看2014年7月1日到2014年7月4日之间的提交，可以这么写

$git log --after=”2014-7-1” --before=”2014-7-4”

--since、--until和--after、--before标记等价

（3）按作者查找

当只想查看某一特定作者的提交时，可以使用—author标记

$git log –author=”john”

如想检索多个作者的提交

$git log –author=”john\|mary”

（4）按提交信息查找

如果想根据commit message来过滤，可以用—grep来标记

$git log –grep=”message”

（5）按文件

如果只对某个特定文件的更改感兴趣，下面这个命令将返回所有foo.py和bar.py文件的提交记录

$git log – foo.py bar.py

## 4.6分支管理

$ git branch

查看本地的所有分支分支，当前所在的分支前面会有一个 \* 做标注。

$ git branch -a

查看远程仓库的分支。

$ git branch –vv

查看本地分支和远程分支的追踪对应关系。

$ git branch <name>

创建分支。

$ git checkout <name>

切换分支。

$ git checkout -b <name>

创建并切换分支。

$ git checkout -b <name> origin/<branch\_name>

创建并切换分支，同时与远程仓库“origin”的“branch\_name”分支关联起来，如果远程仓库“origin”没有“branch\_name”分支的话，则会创建出来，这样做的目的是方便以后本地仓库和远程仓库的同步，如果不这样做的话，在后面的pull或push代码时会有问题，不过这也是有补救措施的，可以执行如下命令建立本地分支和远程分支的连接关系：git branch --set-upstream <name> origin/<branch\_name>

$ git merge <name>

合并某分支到当前分支。

$ git merge --no-ff -m “comment”<name>

合并某分支到当前分支， merge时会产生一个新的commit id，所以加了注释。

注意：在合并分支时，有可能会产生冲突，这是因为多个分支同时修改了某个文件导致的，这时候只要手动修改产生冲突的文件然后重新提交即可。

$ git branch -d <name>

删除分支。

$ git branch -D <name>

如果在被删除的分支上提交了东西却没有merge到当前分支，用“-d”参数删除是会失败的，这时就要用到“-D”参数了，忽略被删除分支上的提交。

## 4.7 git pull

git pull <远程主机名> <远程分支名>:<本地分支名>

比如，要取回origin主机的next分支，与本地的master分支合并，需要写成下面这样 -

$ git pull origin next:master

如果远程分支(next)要与当前分支合并，则冒号后面的部分可以省略。上面命令可以简写为：

$ git pull origin next

上面命令表示，取回origin/next分支，再与当前分支合并。实质上，这等同于先做git fetch，再执行git merge。

## 4.8 git push

git push命令用于将本地分支的更新，推送到远程主机。它的格式与git pull命令相仿。

$ git push <远程主机名> <本地分支名>:<远程分支名>

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

$ git push origin master

上面命令表示，将本地的master分支推送到origin主机的master分支。如果后者不存在，则会被新建。

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

$ git push origin :master # 等同于 $ git push origin --delete master

上面命令表示删除origin主机的master分支。

如果当前分支与远程分支之间存在追踪关系，则本地分支和远程分支都可以省略。

$ git push origin

上面命令表示，将当前分支推送到origin主机的对应分支。

如果当前分支只有一个追踪分支，那么主机名都可以省略。

$ git push

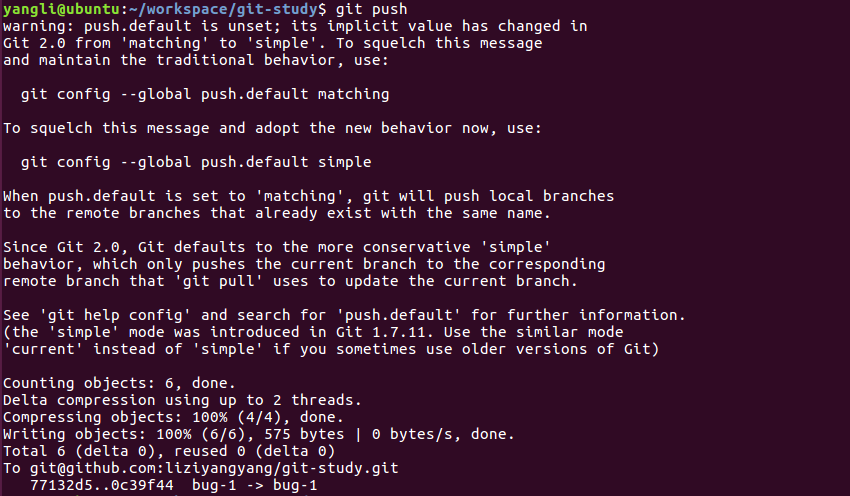
如果当前分支与多个主机存在追踪关系，则可以使用-u选项指定一个默认主机，这样后面就可以不加任何参数使用git push。

$ git push -u origin master

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

不带任何参数的git push，默认只推送当前分支，这叫做simple方式。还有一种模式是'matching'，git 将推送和远程同名的所有本地分支。

Git 2.0版本之后，改为默认采用simple方式。如果要修改这个设置，可以采用git config命令。



若要不再显示本信息并从现在开始采用新的使用习惯，设置：

git config --global push.default simple

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

$ git push --all origin

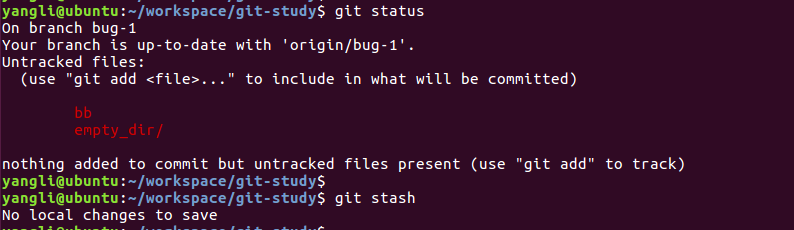
上面命令表示，将所有本地分支都推送到origin主机。

## 4.9 git stash

可能有这样的事情发生，当你正在进行项目中某一部分的工作，里面的东西处于一个比较杂乱的状态，而你想转到其他分支上进行一些工作。问题是，你不想提交进行了一半的工作，否则以后你无法回到这个工作点。解决这个问题的办法就是git stash命令。

“‘储藏”“可以获取你工作目录的中间状态——也就是你修改过的被追踪的文件和暂存的变更——并将它保存到一个未完结变更的堆栈中，随时可以重新应用。

需注意上面那句话所对作用范围的定义，如果你的文件没add，即没有“被跟踪”和“暂存”，stash是不会帮你保存的。



$ git stash

往堆栈推送一个新的储藏

$ git stash list

查看所有的stash

$ git stash pop stash@{版本号}

恢复储藏

## 4.10标签管理

$ git tag <tagname>

对最新提交的commit打标签

$ git tag <tagname> commitid

对某一次提交打标签

$ git tag

查看标签

$ git tag –a tagname “message” commitid

创建带有说明的标签

$ git tag –d tagname

删除本地标签

$ git push origin tagname

推送标签到远程

$ git push origin –tags

推送所有的本地标签

$ git push origin :refs/tags/tagname

删除远程标签

## 4.11远程仓库及多分支

$ git remote

查看远程仓库的名字。

$ git remote -v

“-v”即“--verbose”，查看远程仓库的名字及url。

列出所有当前已配置的远程版本库（-v为详细显示）

$ git remote -v

显示一个远程版本库的信息

$ git remote show <remote>

添加并命名一个新的远程版本库

$ git remote add <remote> <url>

$ git remote rm <remote>

删除远程版本库

# 5 Gitlab官方帮助文档

访问地址：http://url /help

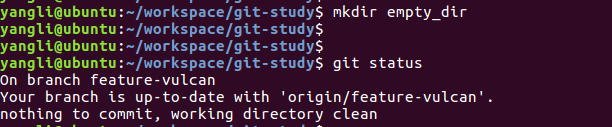
# 6 Git常见问题总结

## 6.1如何提交空目录

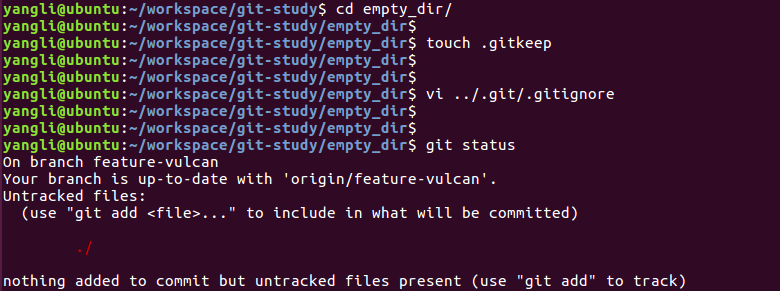
Git仅仅跟踪文件的变动，不跟踪目录

在要上传的空目录下创建.gitkeep文件，然后在项目的.gitignore中设置不忽略.gitkeep，即在.gitignore中添加!.gitkeep

设置之前



设置之后



## 6.2如何设置忽略文件

在仓库根目录下创建.gitignore文件

下面看一个 .gitignore 文件的例子：

# 此为注释 – 将被 Git 忽略

# 忽略所有 .a 结尾的文件

\*.a

# 但 lib.a 除外

!lib.a

# 仅仅忽略项目根目录下的 TODO 文件，不包括 subdir/TODO

/TODO

# 忽略 build/ 目录下的所有文件

build/

# 会忽略 doc/notes.txt 但不包括 doc/server/arch.txt

doc/\*.txt

# 忽略 doc/ 目录下所有扩展名为 txt 的文件

doc/\*\*/\*.txt

#跟踪某类文件

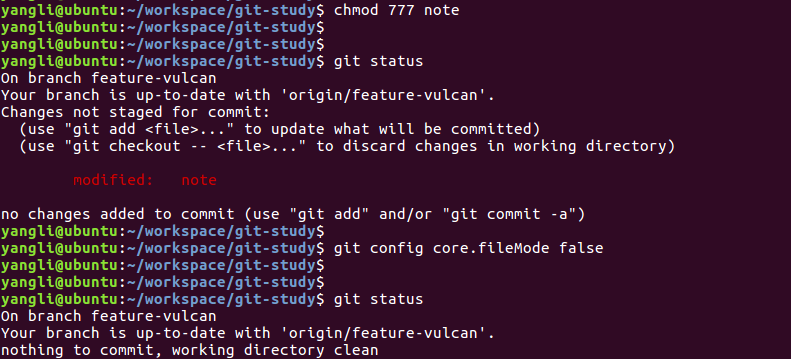
!\*.c

#跟踪某个文件夹

!/plutommi/mmi

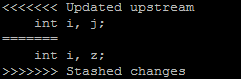
## 6.3如何设置忽略文件权限变更

在git仓库中命令行下执行git config core.fileMode false



## 6.4合并冲突

当我们git pull 时发现本地的修改和远程代码库的修改有冲突，该怎么解决呢？



这个时候，我们可以根据提示来找到冲突的位置，其中Updated upstream 和=====之间的内容就是pull下来的内容，====和stashedchanges之间的内容就是本地修改的内容，需要我们手动进行确认需要怎么合并。合并好之后重新add提交。