代码质量监控平台

**简述：**

Gitlab cicd 结合sonarqube构建代码质量监控平台

**实验环境：**

Gitlab11.1.0

GitRunner

Sonarqube7.4

Mysql5.7

Jdk 8

Maven3.6.0

# 1.基础环境准备

## 1.1 Maven+jdk8+mysql安装步骤省略

## 1.2 Mysql用于sonarqube 存放质量监控数据

或使用现有的mysql数据创建一个sonar账号即可，创建语法：

> CREATE DATABASE sonar CHARACTER SET utf8 COLLATE utf8\_general\_ci;> grant all privileges on sonar.\* to "sonar"@"%" identified by "sonar";> flush privileges;

另sonarqube7.9以上版本不再支持mysql,支持es和Postgresql、MS-Sql

# 2.Gitlab安装

参考《Gitlab安装手册.doc》

# 3.Sonarqube安装

## 3.1 创建sonar用户

useradd sonar;echo "123456"|passwd --stdin sonar

*tips: 内置es不能以root用户启动*

## 3.2 安装sonarqube

wget <https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-7.4.zip>

unzip sonarqube-7.4.zip

chown -R sonar:sonar sonarqube-7.4/

## 3.3 修改系统配置，内置es需要

echo -e "\* soft nofile 65536\n\* hard nofile 131072\n\* soft nproc 2048\n\* hard nproc 4096\n" >>/etc/security/limits.conf

echo "vm.max\_map\_count=655360" >>/etc/sysctl.conf

sysctl -p

## 3.4 配置sonarqube

vi sonarqube-7.4/conf/sonar.properties

#jdbc连接账号密码

sonar.jdbc.username=sonar

sonar.jdbc.password=sonar

# 数据连接

sonar.jdbc.url=jdbc:mysql://172.16.68.164:3306/sonar?useUnicode=true&characterEncoding=utf8&rewriteBatchedStatements=true&useConfigs=maxPerformance&useSSL=false

# web配置绑定IP和端口

sonar.web.host=172.16.68.164

sonar.web.port=9000

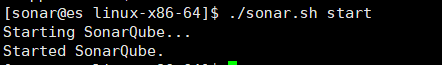
## 3.5 启动

su - sonar

*Tips: 一定要切换成sonar用户，内置的es不允许root启动*

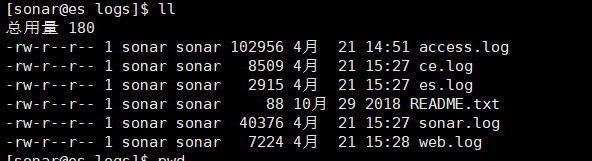
cd ${sonarqube\_home}/ bin/linux-x86-64/

./sonar.sh start

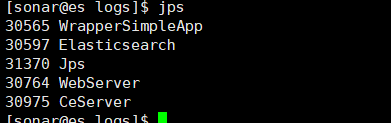


查看日志目录启动情况：

cd ${sonarqube\_home}/logs



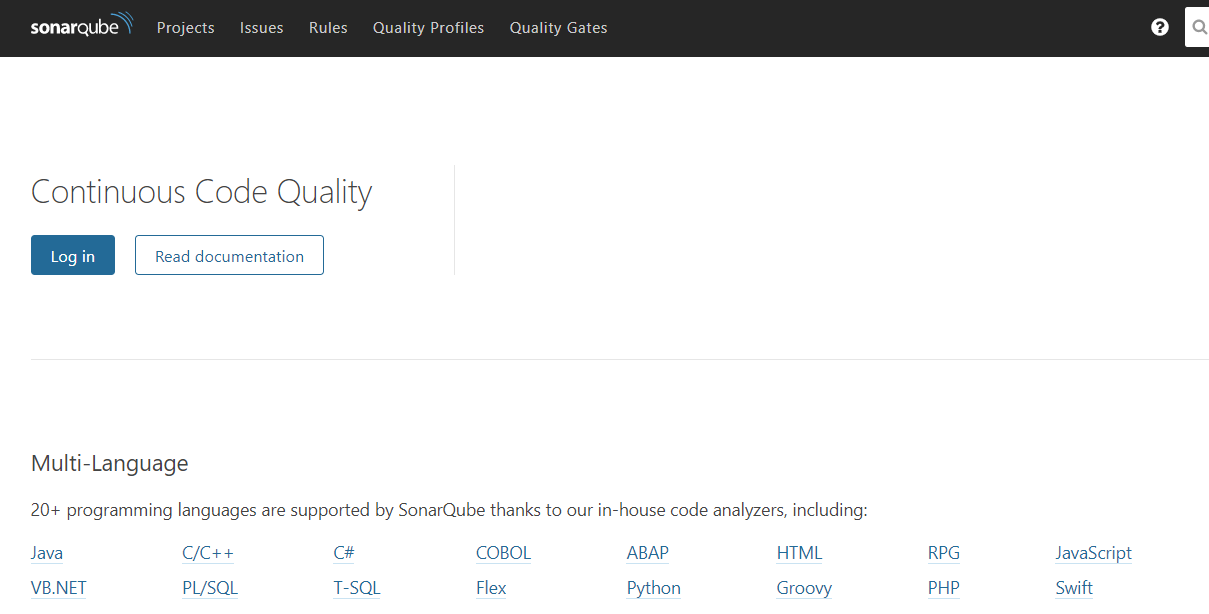
主要查看sonar.log和web.log的日志文件



## 3.6 启动验证

<http://172.16.68.164:9000/>

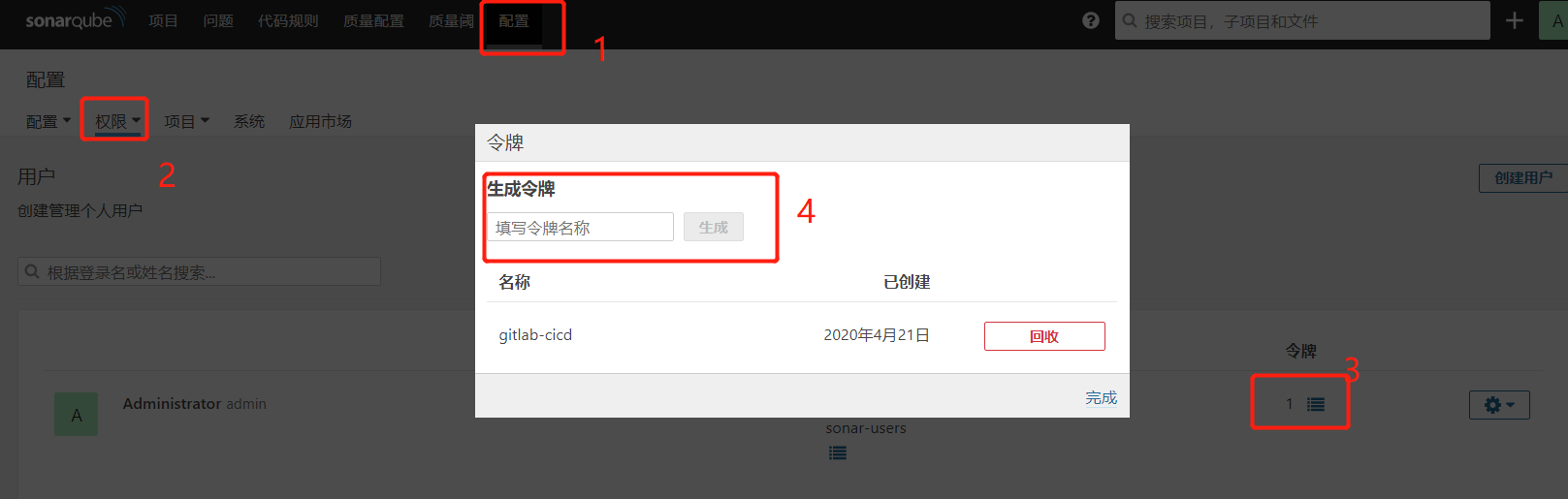
默认账号密码admin/admin



*Tips:如无法访问确定是否启动以及端口是否开放*

第一次登录会提示创建token,可以跳过后续添加。此token用于cicd上传报告时权限token配置，token仅显示一次需要记录好。

两种方式获取新的token:

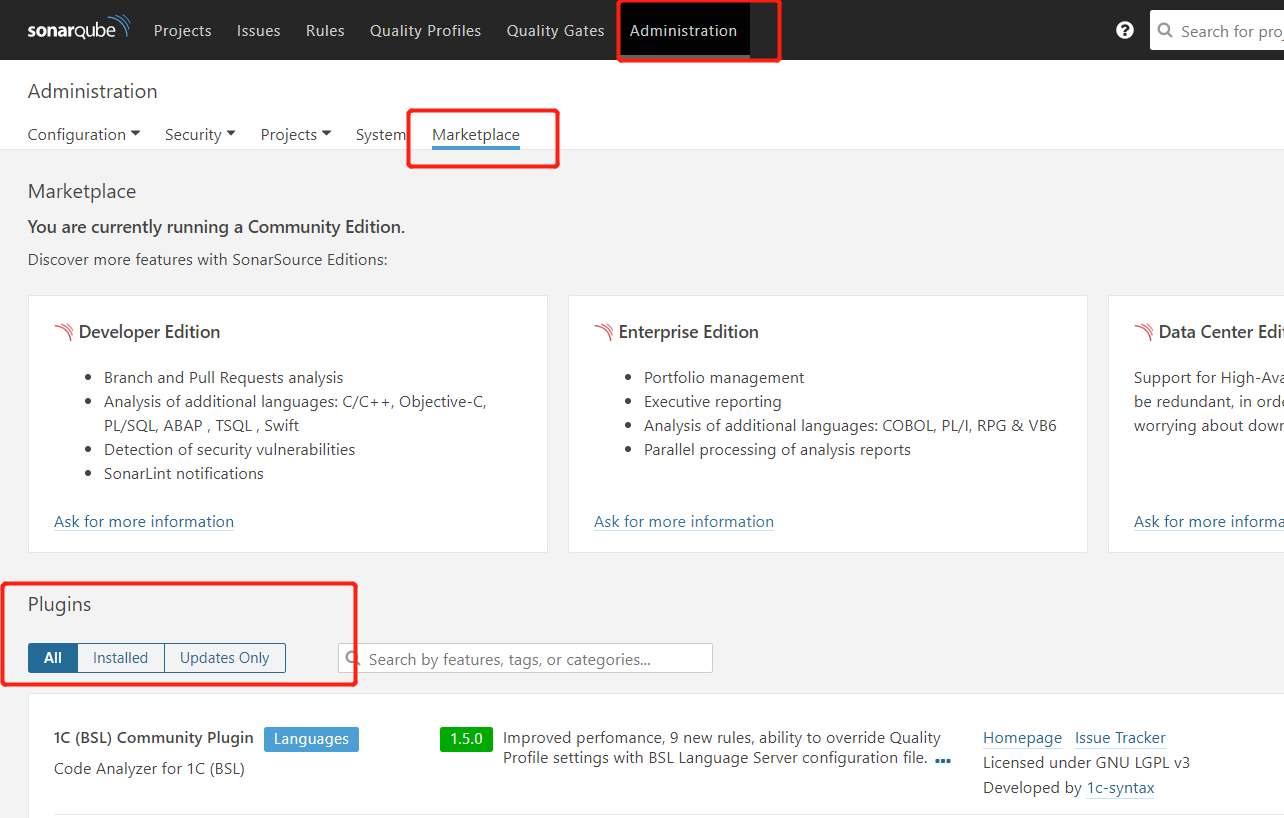


或者



## 3.7 汉化以及插件安装

在线安装方式：administration--->Marketplace--->Plugins



版本兼容问题我们采取离线安装方式，需要安装的离线插件：

**A.离线安装一定要找到对应版本，否则sonarqube无法启动**

**B.下载离线安装包.jar文件放到${sonarqube}/ extensions/plugins目录下**

**1)汉化包sonar-l10n-zh 1.24**

<https://github.com/SonarQubeCommunity/sonar-l10n-zh/releases>

**2) CheckStyle 4.27**

<https://github.com/checkstyle/sonar-checkstyle/releases>

**3)sonar-gitlab-plugin 4.0.0**

https://github.com/gabrie-allaigre/sonar-gitlab-plugin/releases

**4)pmd 3.0.1 如果使用集成阿里云的此插件可不用**

https://github.com/jensgerdes/sonar-pmd/releases

**5)findbugs 3.9.1**

https://github.com/spotbugs/sonar-findbugs/releases

**6)sonar-auth-gitlab-plugin 1.2.2**

https://github.com/gabrie-allaigre/sonar-auth-gitlab-plugin/releases

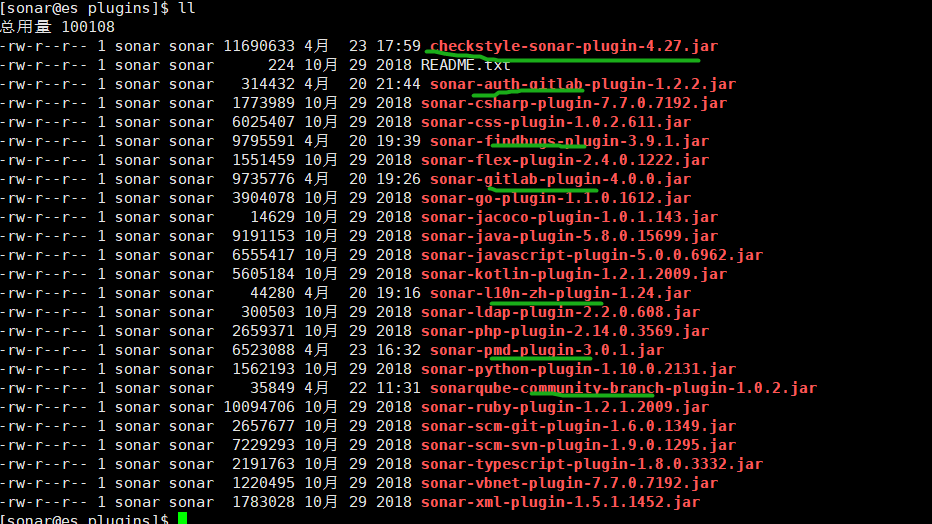
**7)此pmd已经集成 pmd和阿里云的p3c-pmd**

<https://github.com/Xlinlin/sonar/tree/master/sonar-p3c-pmd>

**8)sonarqube-community-branch 1.0.2** ui多版本展示支持

<https://github.com/mc1arke/sonarqube-community-branch-plugin/releases>

***tips：此插件除需要放到extensions/plugins目录下之外，还需拷贝一份放到${sonarqube}/lib/commo目录下***

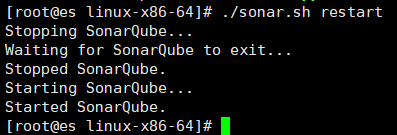


*Tips：请注意插件包的权限*

重启sonarqube

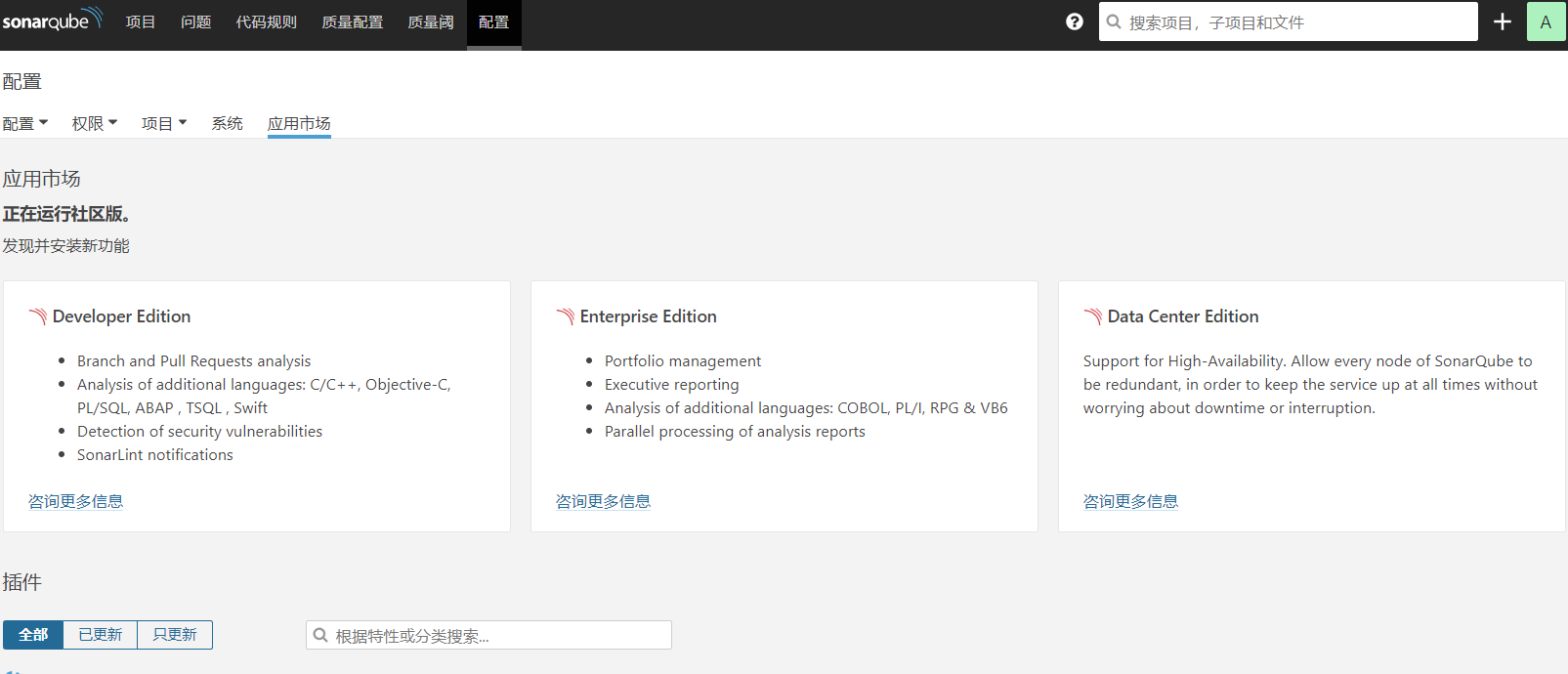
cd ${sonarqube\_home}/ bin/linux-x86-64/

./sonar.sh restart



验证插件安装：

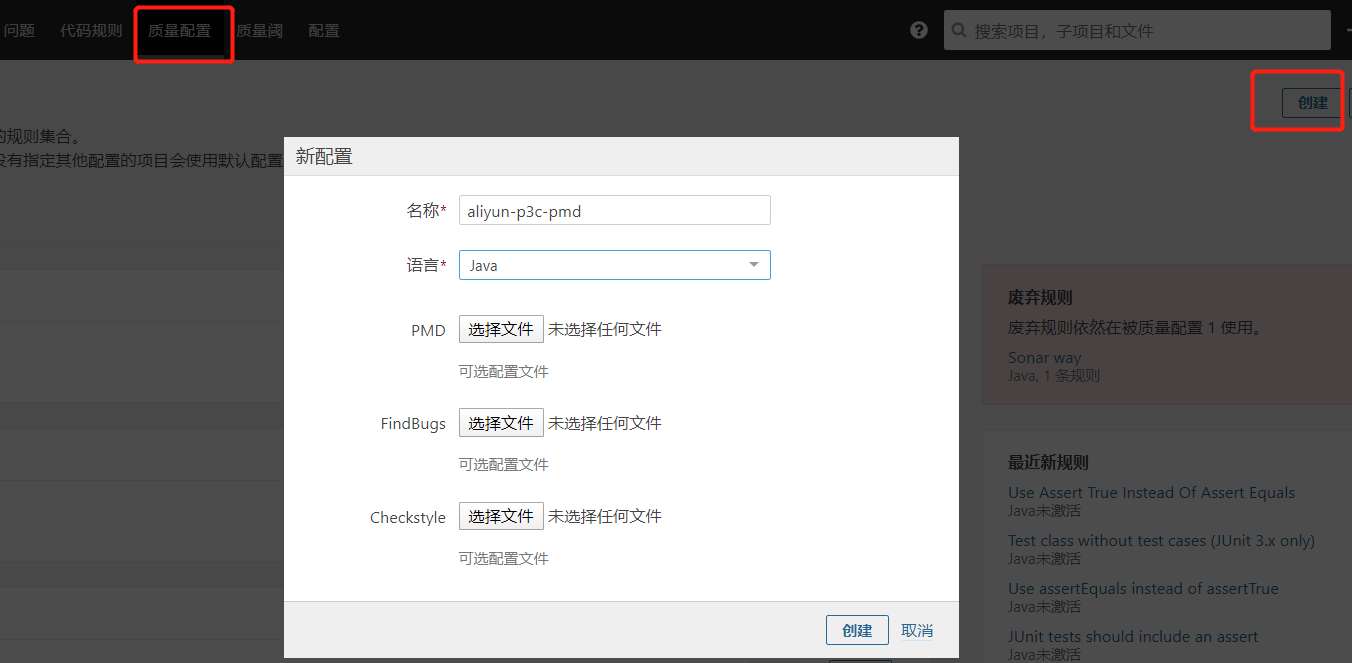
<http://172.16.68.164:9000/>



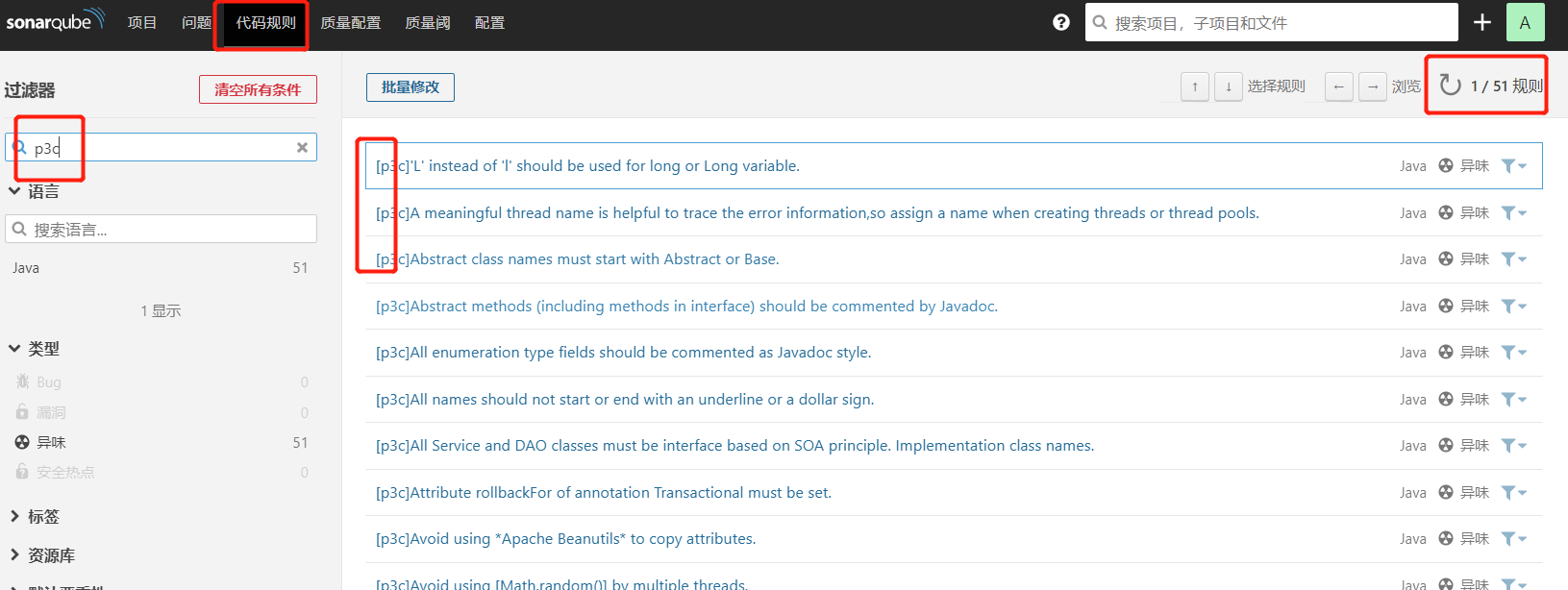
## 3.8 使用阿里云P3C质量检测规则(推荐)

### 3.8.1 创建新的质量配置规则

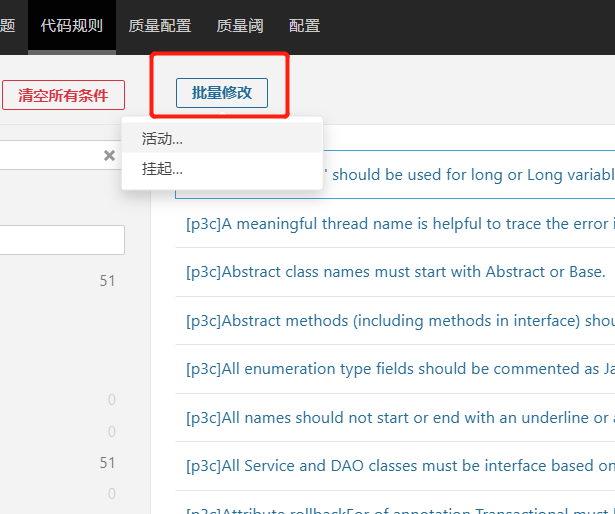
质量配置-创建

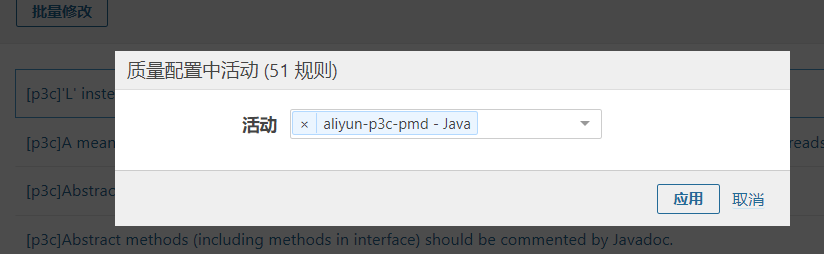
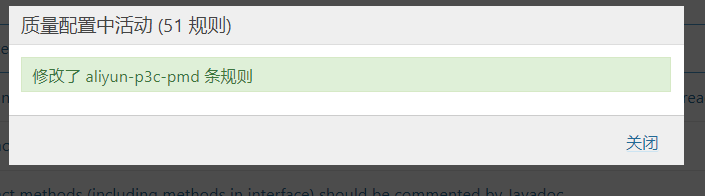


### 3.8.2 添加p3c的规则到新创建的配置中 51条规则

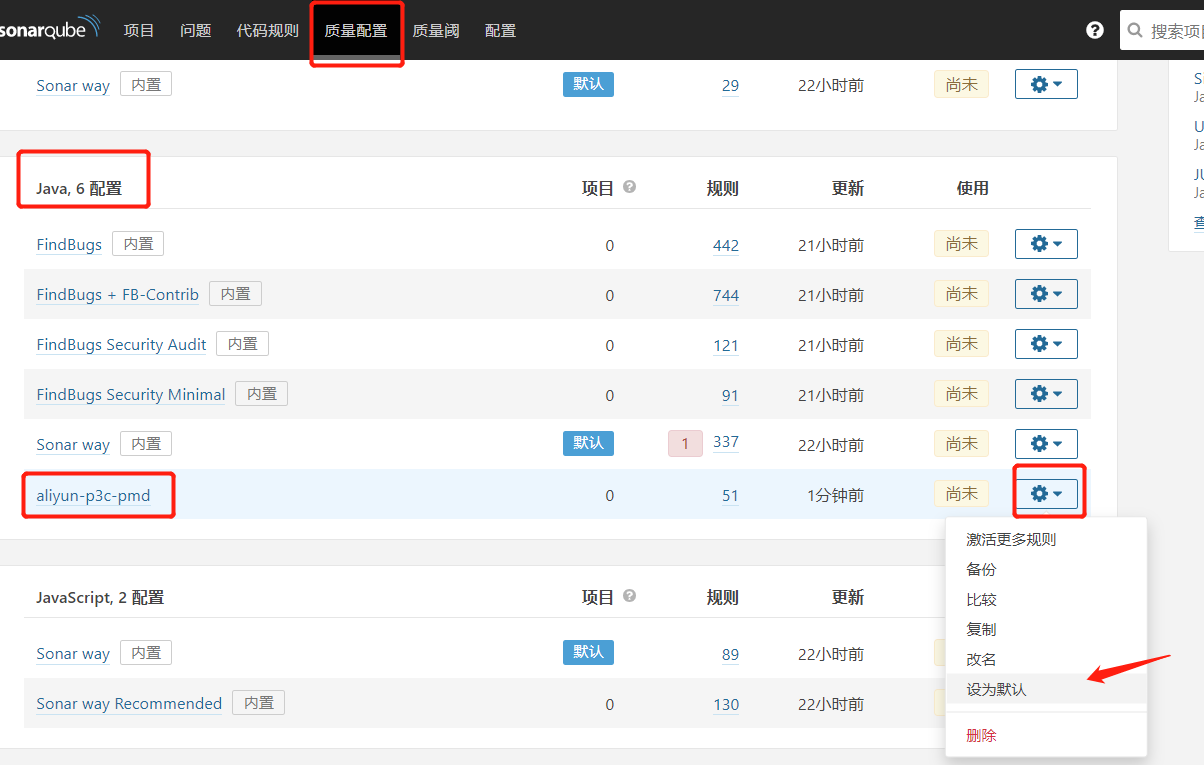


点击批量修--活动添加到新建的规则中



### 3.8.3 设置P3C为默认



## 3.9 使用自定义CheckStyle做质量检测

### 3.9.1 需要安装CheckStyle插件

具体安装方法参考3.7章节的checkStyle插件

Tips:如在这一步才安装记得重启sonarqube

### 3.9.2 创建自定义的CheckStyle规则配置



***郑重提示：***

***1.在Sonarqube里，CheckStyle有两者规则形式的存在，一种是普通规则，一种是规则模板。如果提供的CheckStyle里面有规则模板的存在将不允许上传，而此种规则模板的规则只能通过Sonarqube控制台去创建，稍后补充创建步骤***

***2.CheckStyle.xml模板里不允许出现重复，也会导致上传失败。***

***3.现有的checkStyle.xml模板中已发现规则模板有：***

***IllegalTokenText、MethodLength、NoLineWrap、NeedBraces、LeftCurly、WhitespaceAround、EmptyLineSeparator、TypeName、MemberName、LocalVariableName、MethodParamPad、WhitespaceAfter、NoWhitespaceAfter、NoWhitespaceBefore、ParenPad、OperatorWrap、JavadocType、JavadocMethod、JavadocVariable、JavadocStyle***

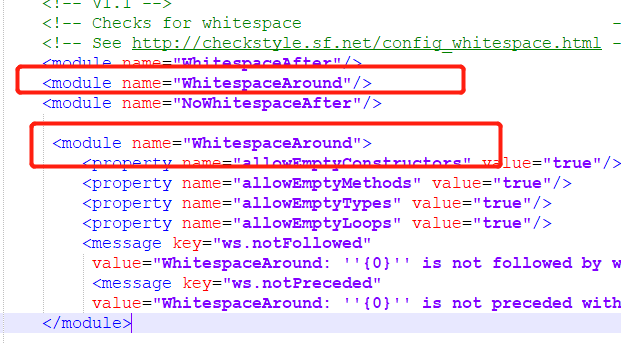


失败的大概三种提示：

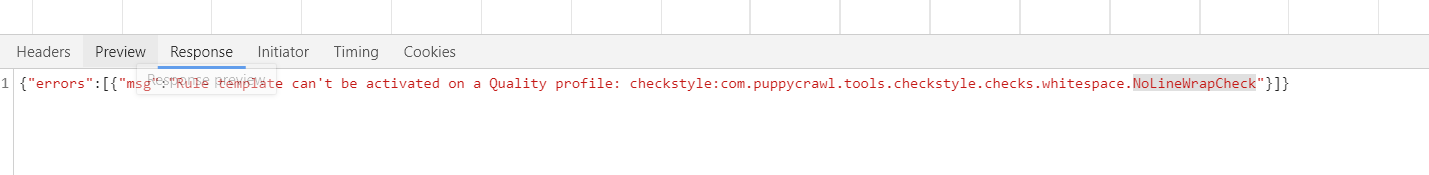
1. 控制台错误提示，标识有规则module重复



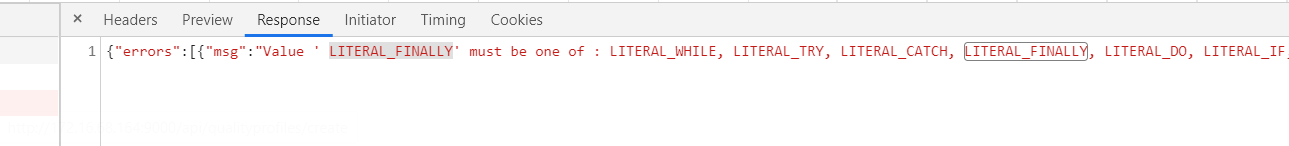
比如：



1. checkStyle中规则无法激活提示错误，比如No Line Wrap



1. 参数填写错误的，比如多了空格



导入成功后如下展示：

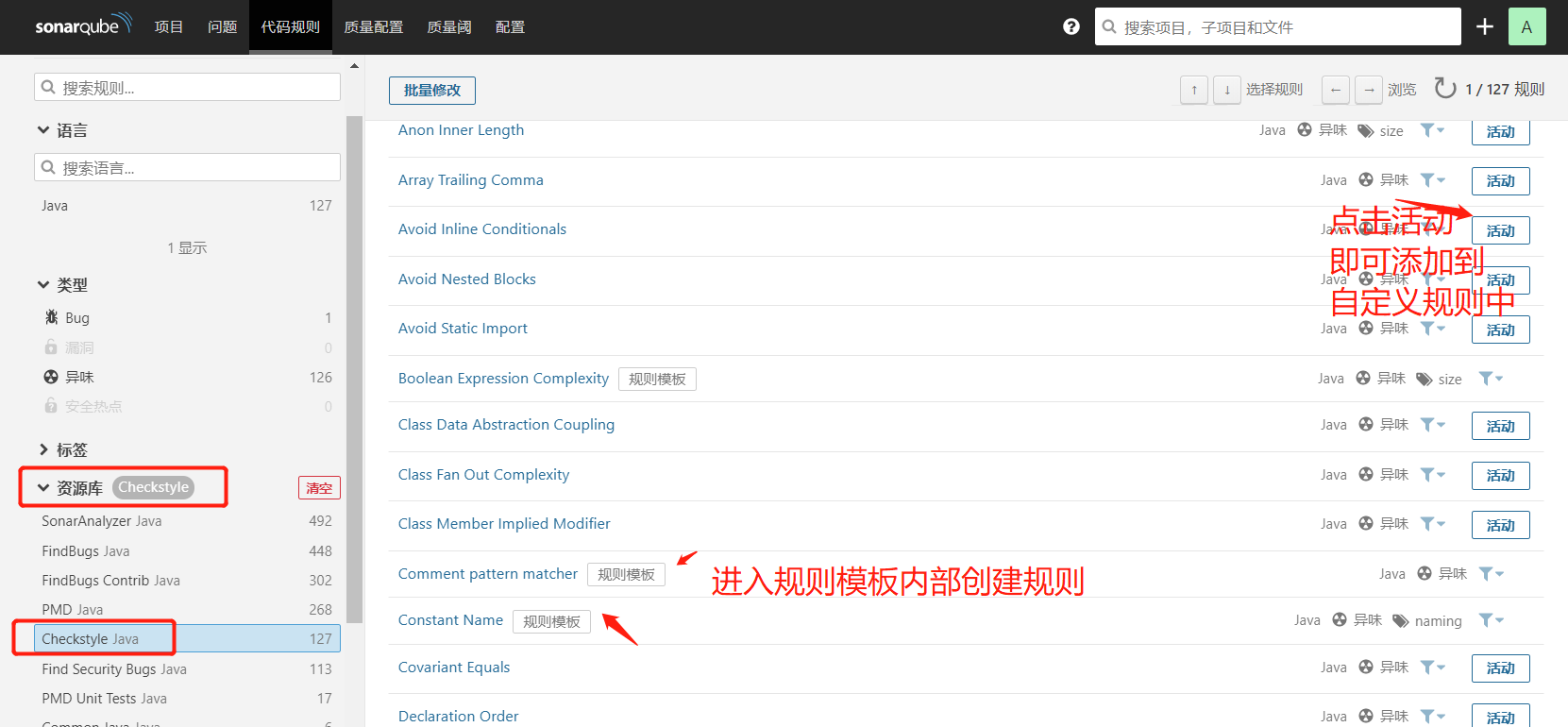


### 3.9.3 sonarqube中添加规则模板中的规则

1).点击进入添加的自定义checkStyle规则

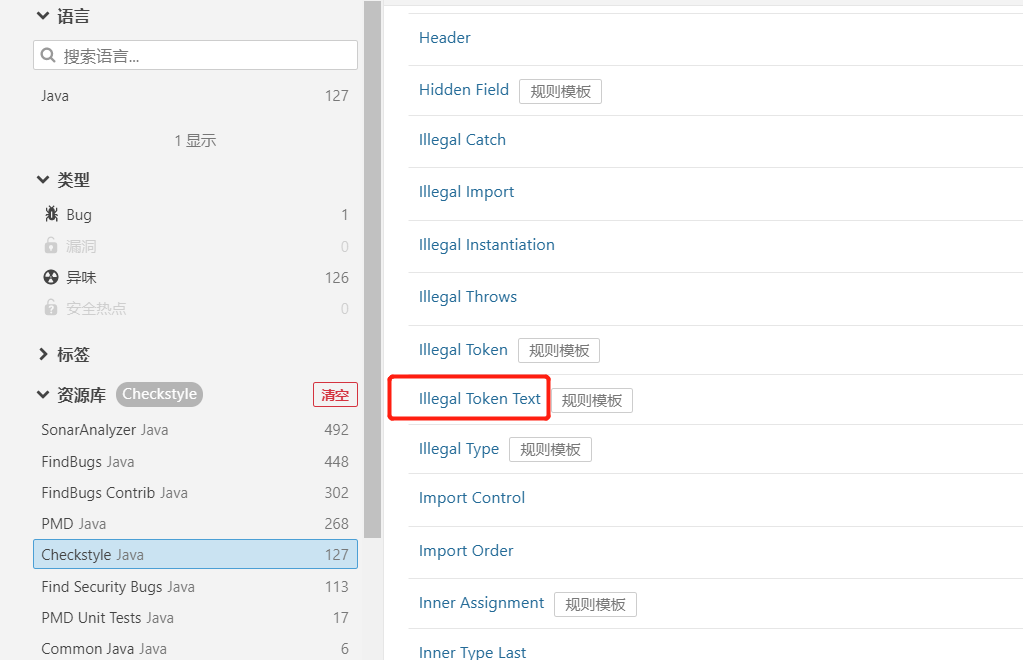


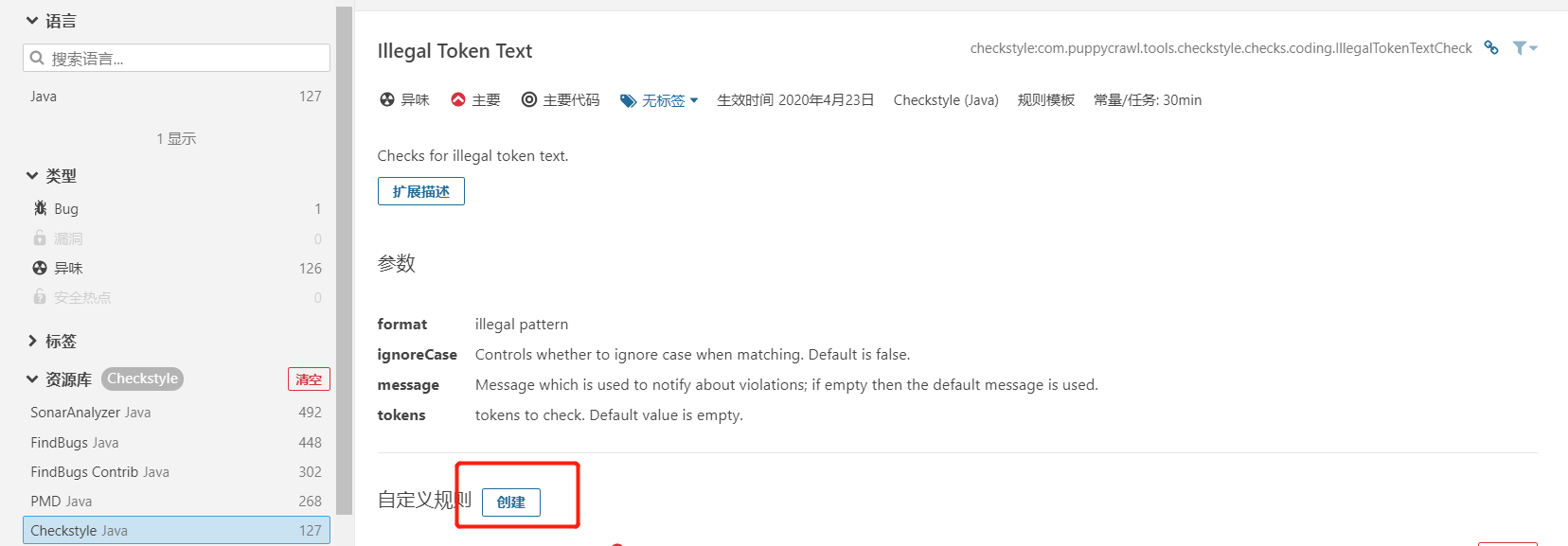
2).更多激活规则



3) 规则模板创建规则，以Illegal Token Text 为例

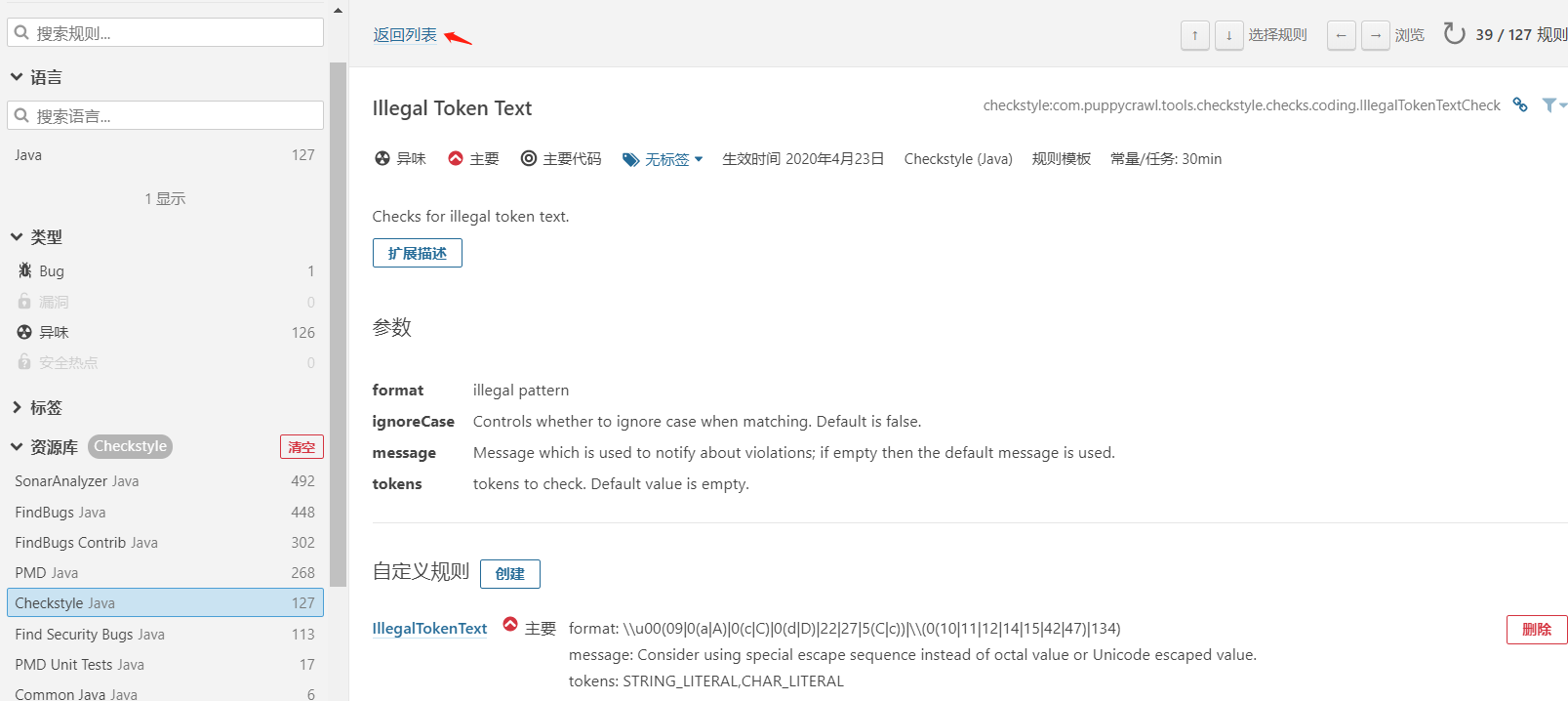
点击进入规则模板：



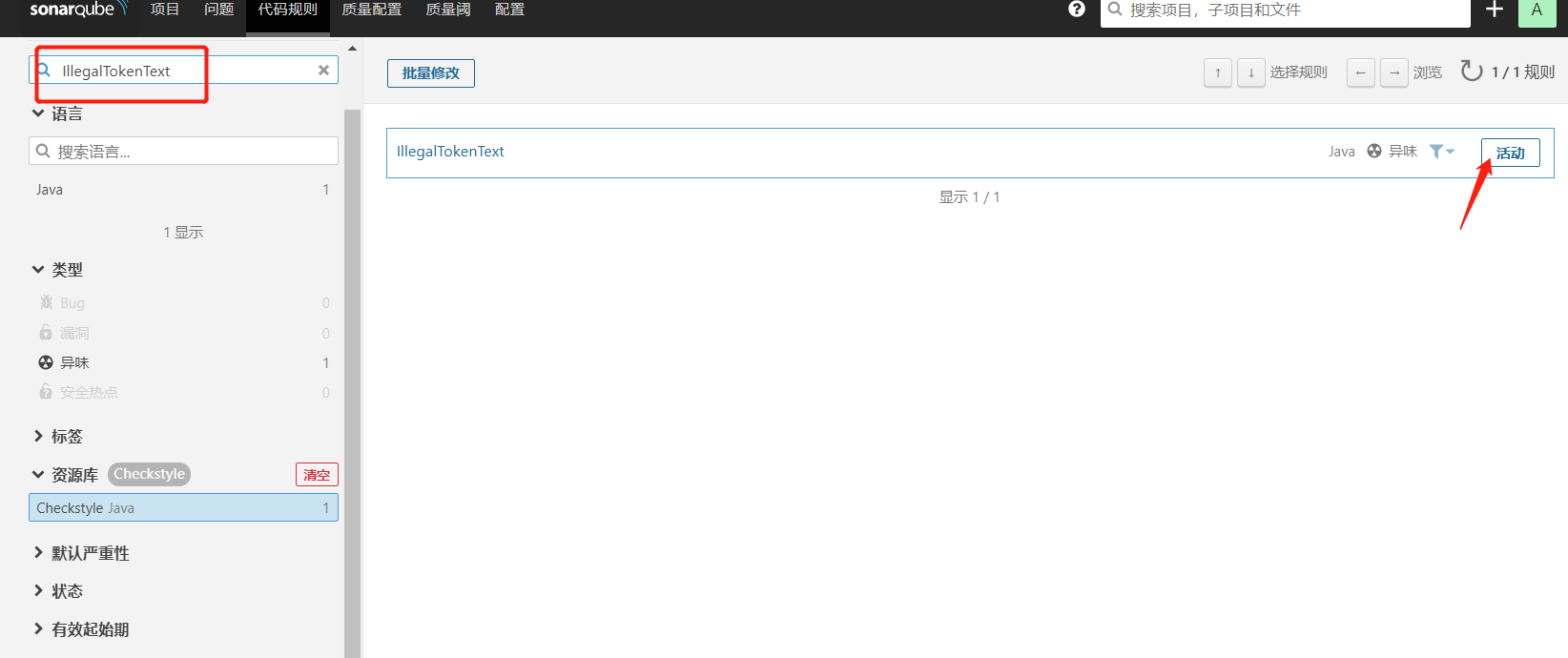


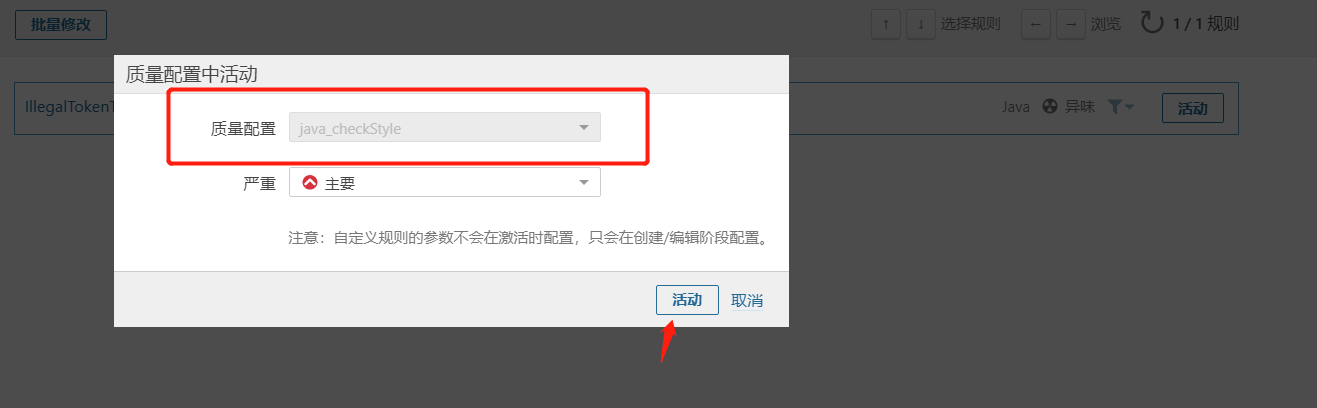
创建规则





返回到列表，添加改规则至自定义的质量规则中





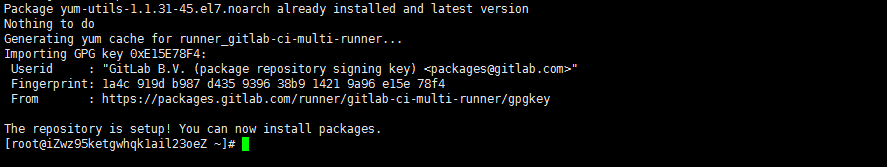
添加成功后总规则数量会增加，我这里已经添加更多规则：



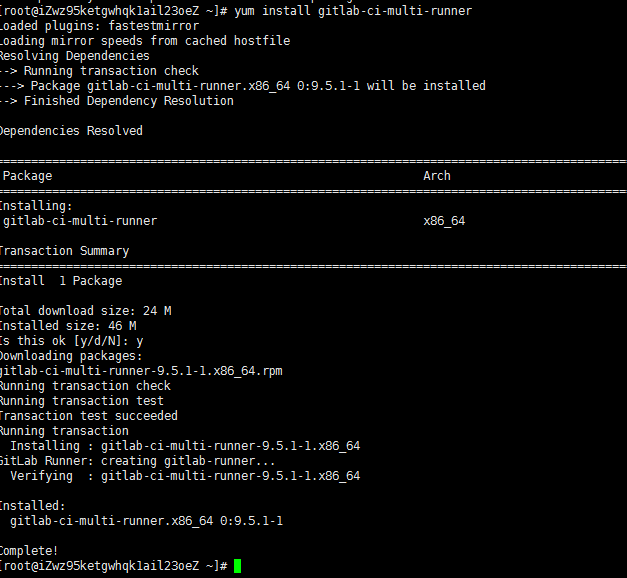
# 4.Git-runner安装

## 4.1 安装

[root@iZwz95ketgwhqk1ail23oeZ ~]# curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-ci-multi-runner/script.rpm.sh | sudo bash



[root@iZwz95ketgwhqk1ail23oeZ ~]# yum install gitlab-ci-multi-runner

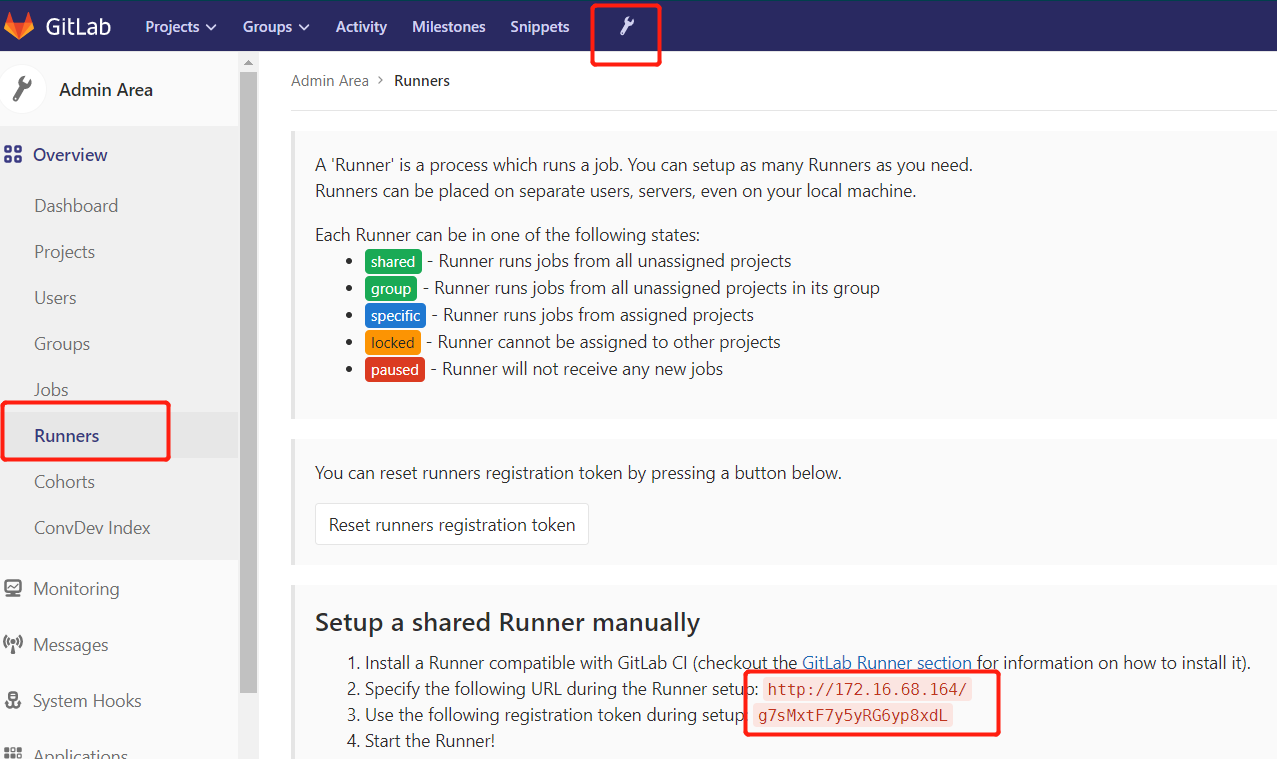


## 4.2 gitlab注册 runner

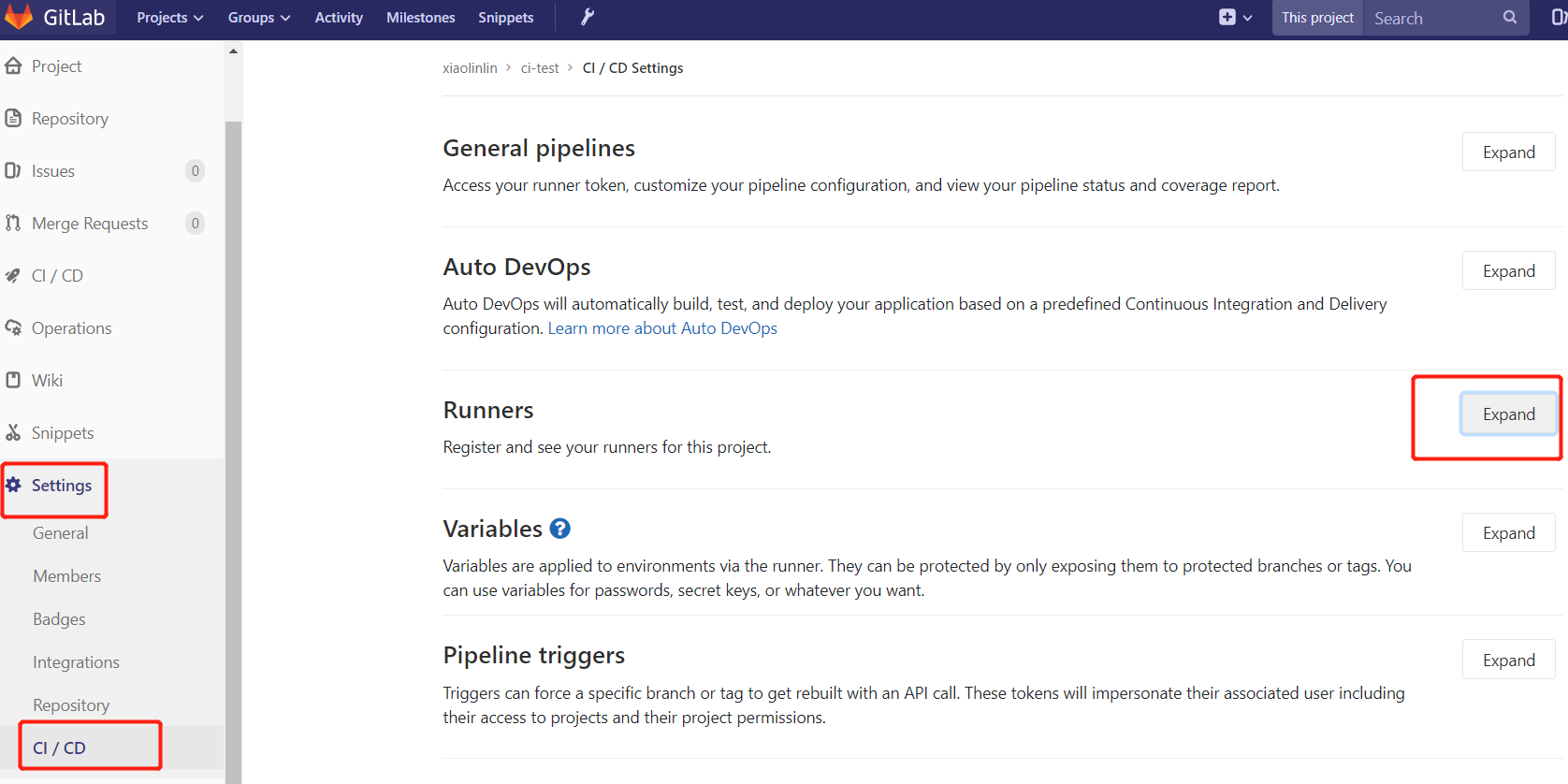
分全局共享runner和项目独享runner,二者注册方式一样，获取的token方式不一样

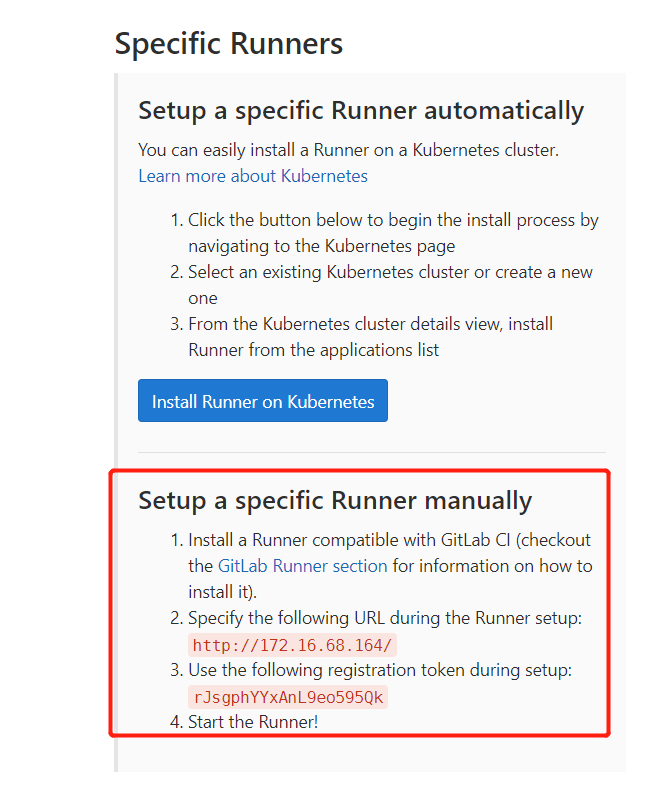
### 4.2.1 全局共享runner获取token

Tips:需要gitlab管理员权限



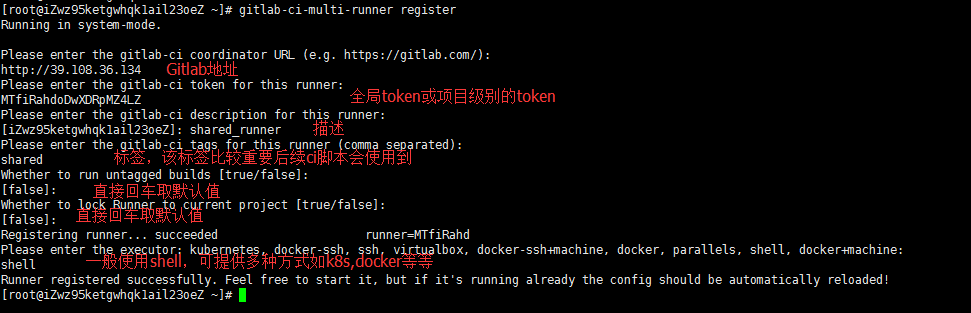
### 4.2.2 项目独享runner获取token



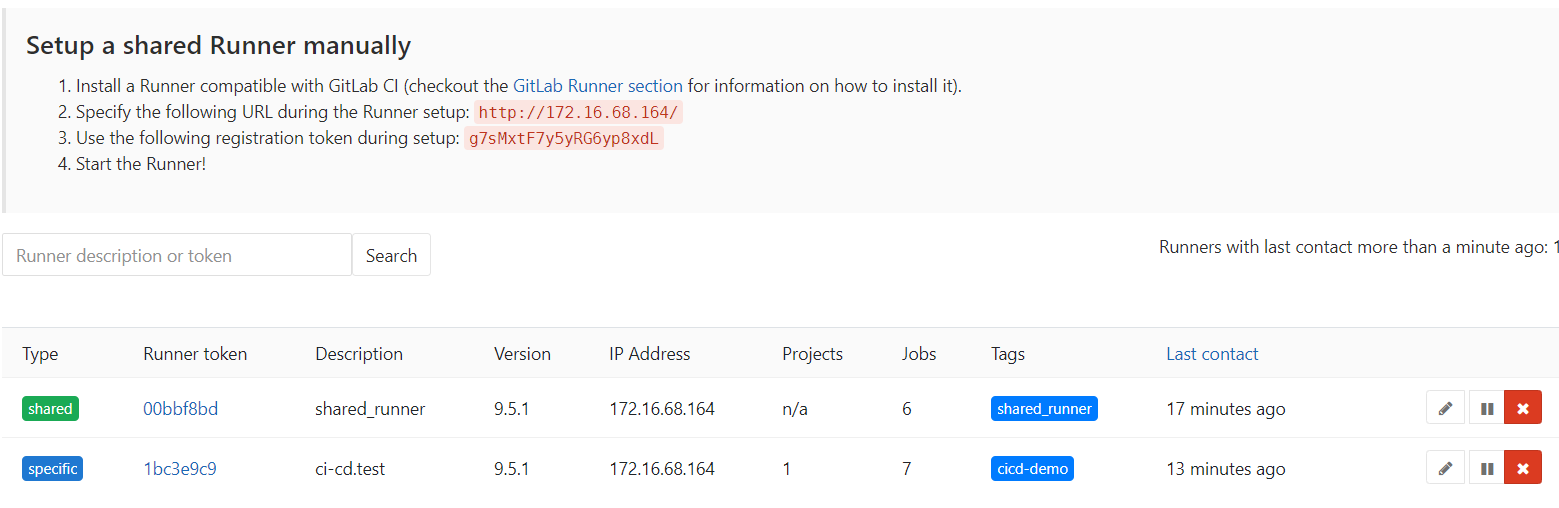


### 4.3.3 注册

[root@iZwz95ketgwhqk1ail23oeZ ~]# gitlab-ci-multi-runner register



全局注册成功后可以全局runner看到runner列表：

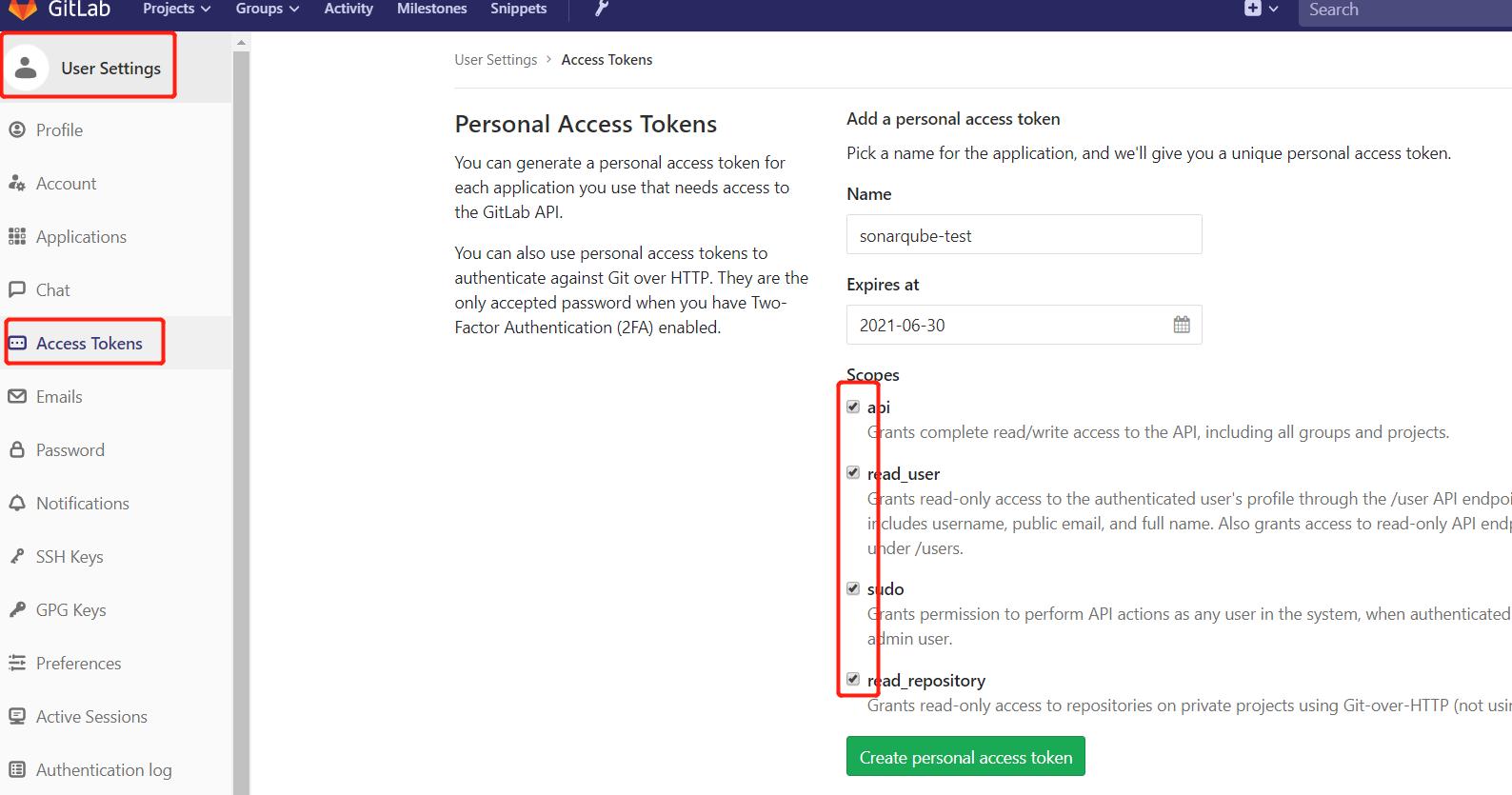


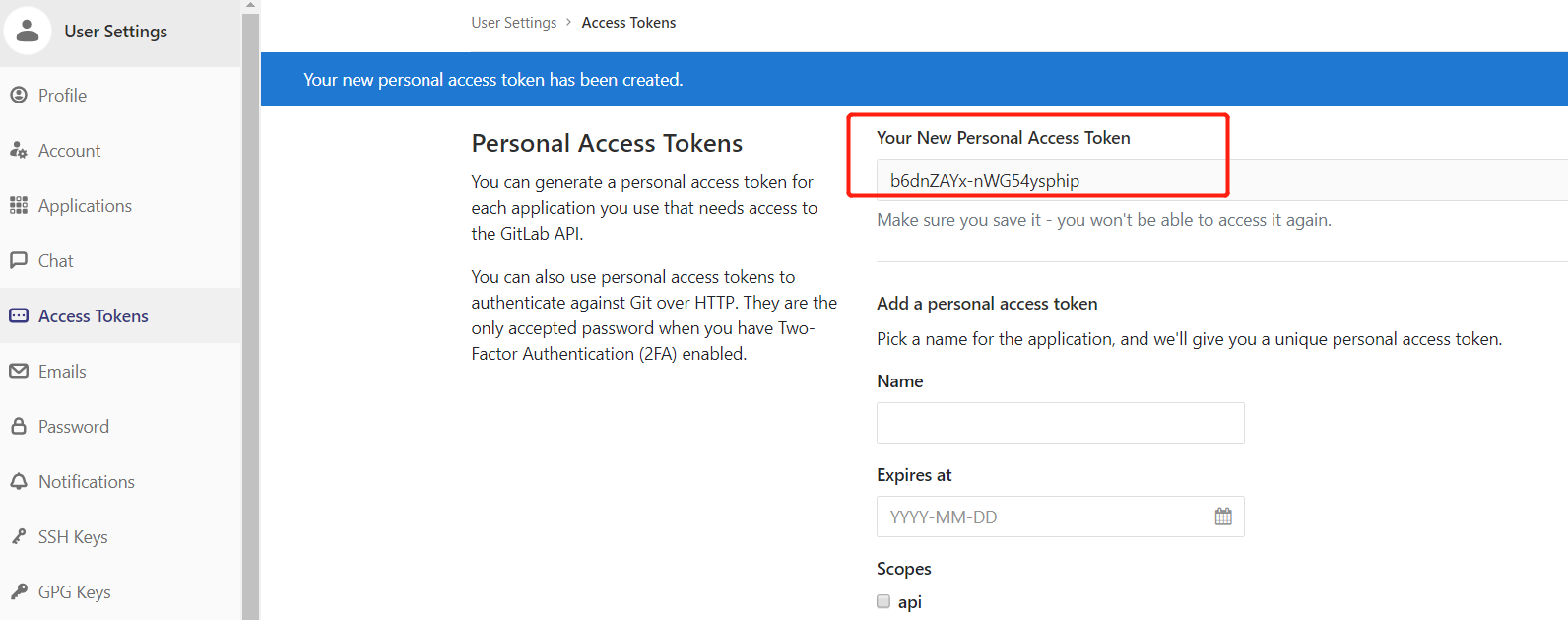
# 5.Gitlab-CICD和Sonarqube配合使用

## 5.1 Sonarqube Gitlab配置

### 5.1.1 gitlab中获取权限

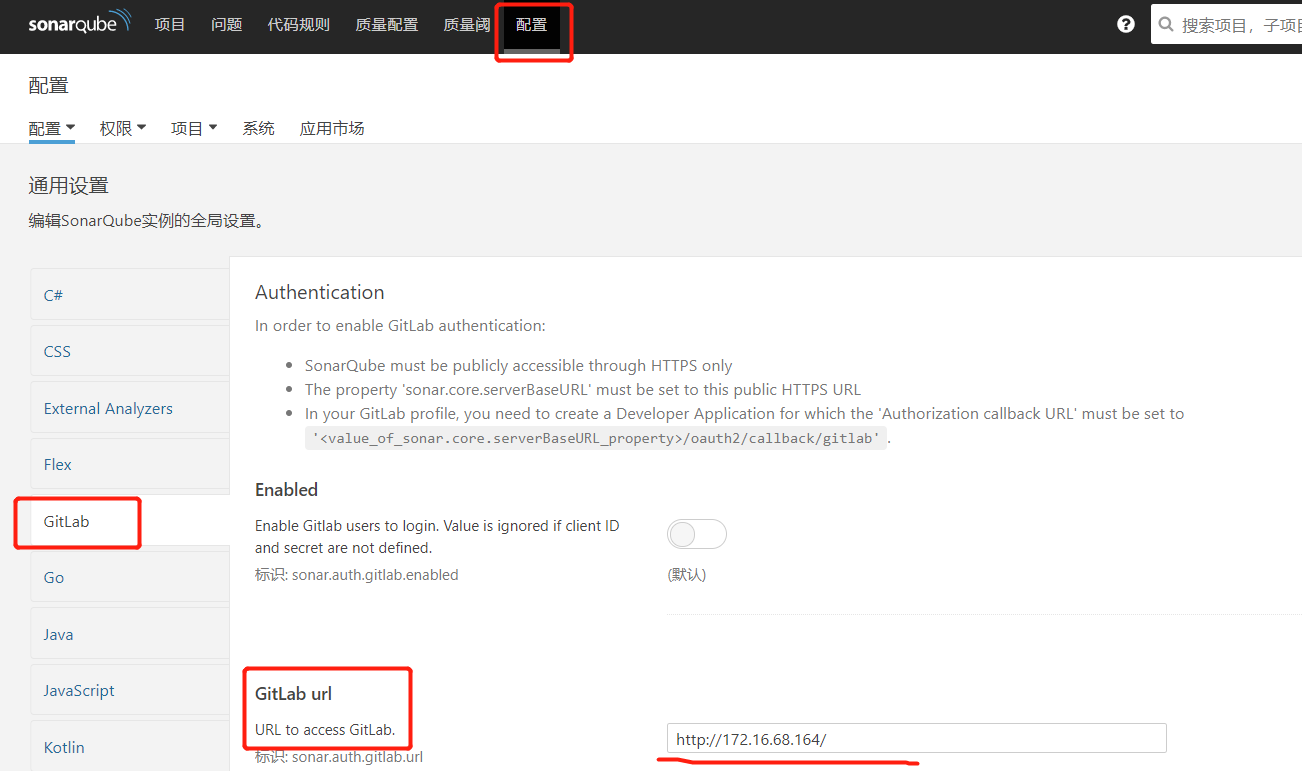
User==>Setting==>AccessTokens

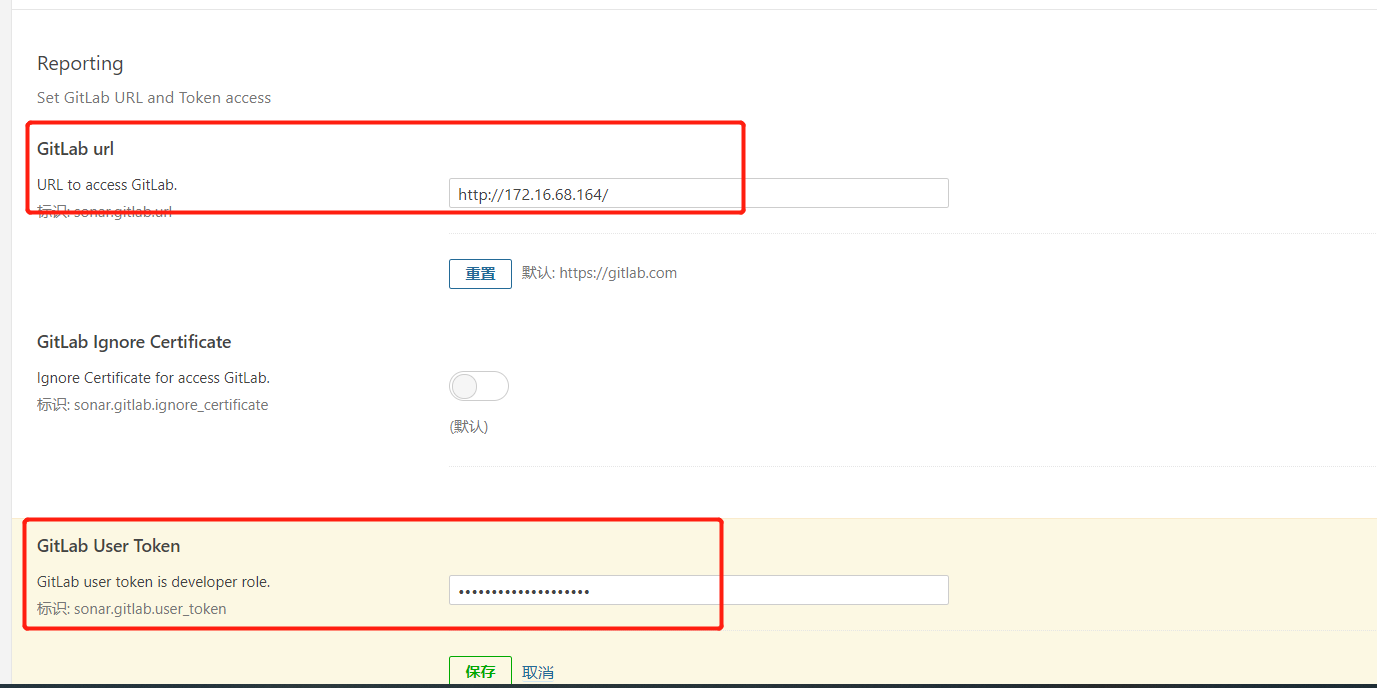




Tips：记住这个Token需要到sonarquebe中配置

### 5.1.2 Sonarqube配置Gitlab信息

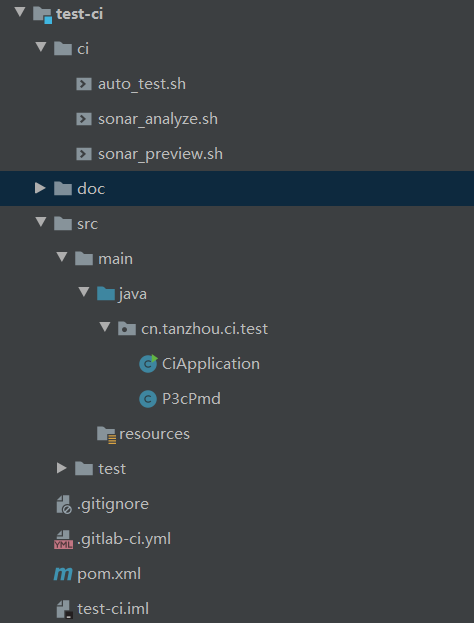




## 5.2 CICD配置

### 5.2.1 Gitlab上创建项目

目录结构大致如下：



### 5.2.2 添加CICD文件 .gitlab-ci.yml 内容：

|  |
| --- |
| stages:  - build  - test  - deploy  variables:  SONAR\_HOST: http://172.16.68.164:9000  SONAR\_LOGIN: 55b3358419c9543d761fb808731288df5cc8ed98  GITLAB\_URL: http://172.16.68.164  GITLAB\_TOKEN: kxaCFitFKWzfMVehyRS9  sonar\_preview:  stage: build  script:  - echo "Starting verify code!"  - |  mvn -batch-mode verify sonar:sonar \  -Dmaven.test.skip=true \  -Dsonar.host.url=${SONAR\_HOST} \  -Dsonar.login=${SONAR\_LOGIN} \  -Dsonar.gitlab.project\_id=$CI\_PROJECT\_ID \  -Dsonar.gitlab.commit\_sha=$CI\_BUILD\_REF \  -Dsonar.gitlab.ref\_name=$CI\_BUILD\_REF\_NAME \  -Dsonar.gitlab.url=${GITLAB\_URL} \  -Dsonar.gitlab.user\_token=${GITLAB\_TOKEN} \  -Dsonar.gitlab.ci\_merge\_request\_iid=$CI\_MERGE\_REQUEST\_IID \  -Dsonar.gitlab.merge\_request\_discussion=true \  -Dsonar.projectKey=$CI\_PROJECT\_NAME \  -Dsonar.projectName=$CI\_PROJECT\_NAME \  -Dsonar.gitlab.comment\_no\_issue=true -X \  -Dsonar.branch.name=$CI\_COMMIT\_REF\_NAME  - echo "Finished verify for without master branch"  except:  - master  tags:  - shared\_runner  master\_analyze:  stage: build  script:  - echo "Starting analyze code to master branch!"  - mvn clean install -DskipTest  - |  mvn sonar:sonar \  -Dsonar.host.url=${SONAR\_HOST} \  -Dsonar.login=${SONAR\_LOGIN} \  -Dsonar.analysis.mode=preview \  -Dsonar.gitlab.project\_id=$CI\_PROJECT\_PATH \  -Dsonar.gitlab.commit\_sha=$CI\_COMMIT\_SHA \  -Dsonar.gitlab.ref\_name=$CI\_COMMIT\_REF\_NAME \  -Dsonar.projectKey=$CI\_PROJECT\_NAME \  -Dsonar.projectName=$CI\_PROJECT\_NAME \  -Dsonar.branch.name=$CI\_COMMIT\_REF\_NAME  - echo "Finished analyze for master branch"  only:  - master  tags:  - shared\_runner  allow\_failure: false |

参考文件：



变量参数解释和查询路径：

|  |
| --- |
| sonar.branch.name 支持多分支 需要配合sonarqube-community-branch-plugin插件  sonar.gitlab.comment\_no\_issue 讲检测结果回写到gitlab中，需要与sonar.gitlab.url、sonar.gitlab.user\_token两个参数配合使用  sonar gitlab相关参数：  <https://github.com/gabrie-allaigre/sonar-gitlab-plugin>  sonar qube相关参数：  <https://docs.sonarqube.org/latest/analysis/analysis-parameters/>  gitlab cicd一些内置变量：  <https://docs.gitlab.com/ee/ci/variables/predefined_variables.html> |

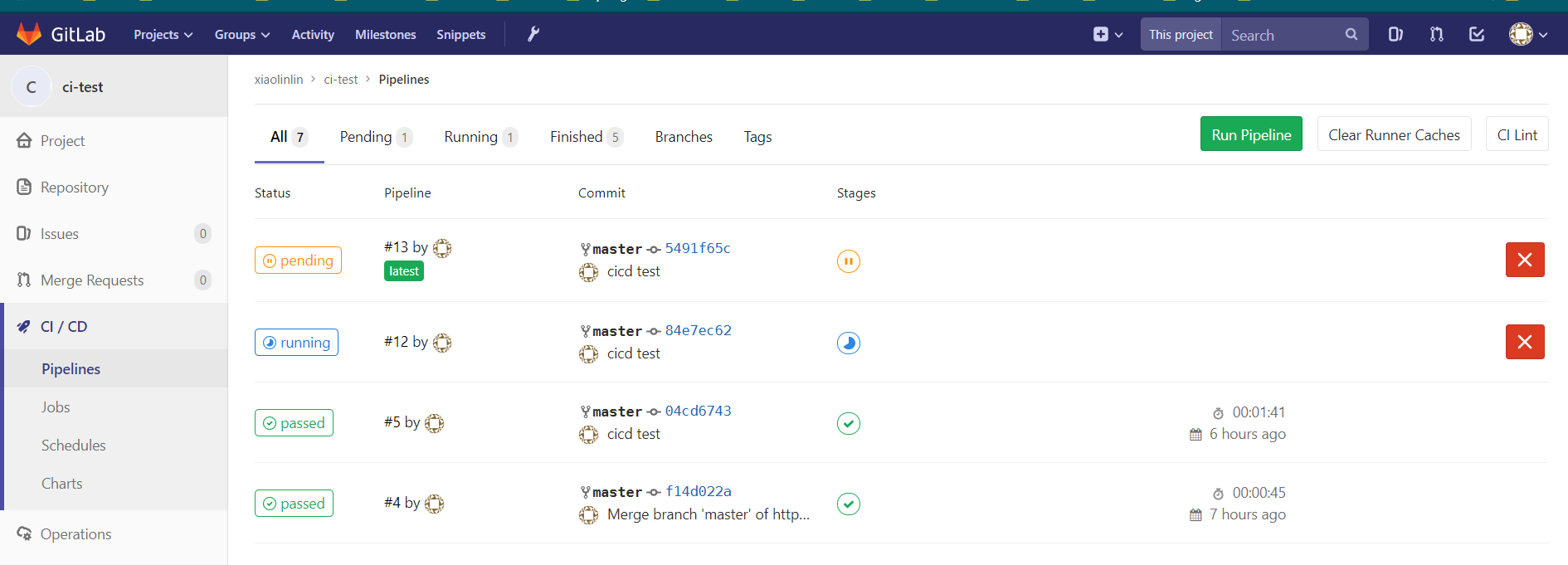
Tips：如果不想构建cicd，提交代码注释中添加：[]

### 5.2.3 添加java文件内容触发gitlab提交：

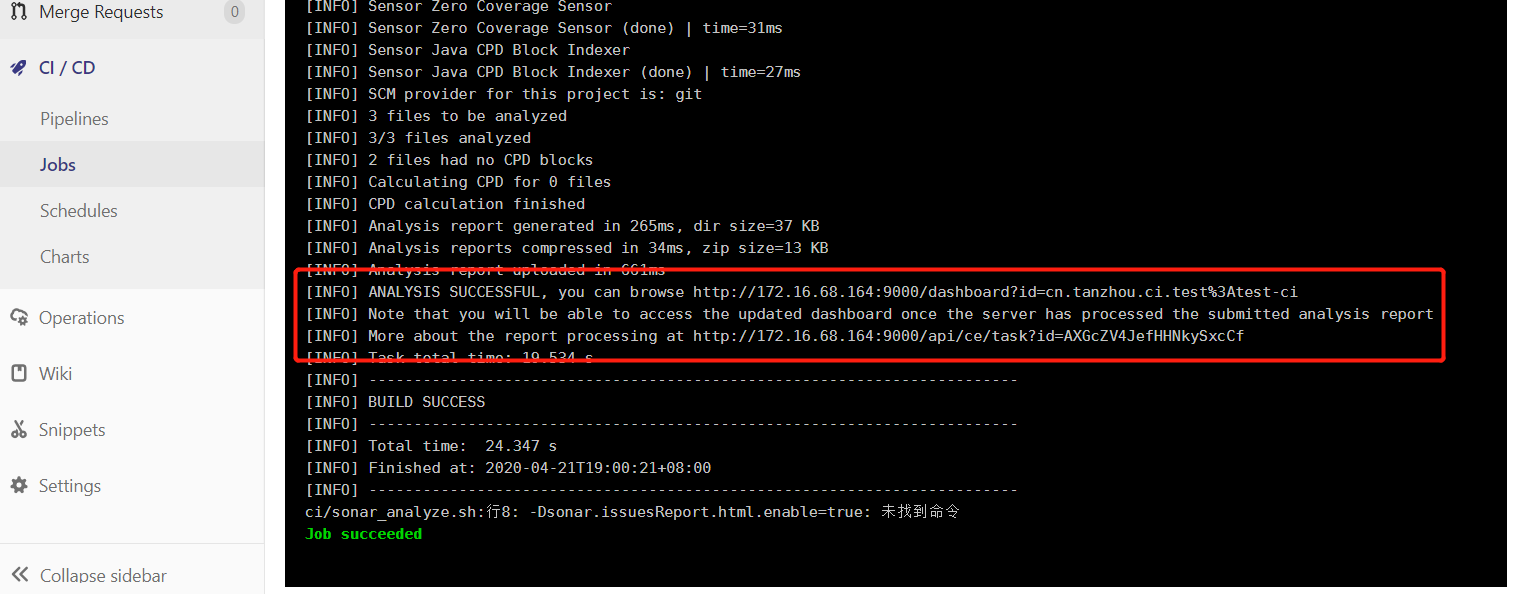
|  |
| --- |
| /\*\*  \* [简要描述]: ci 测试  \* [详细描述]:  \*  \* @author xiaolinlin  \* @version 1.0, 2020/4/20 16:47  \* @since JDK 1.8  \*/  public class CiApplication  {  public static final String testfinal = "00001";  public static void main(String[] args)  {  System.out.printf("CI-test");  }  } |

### 5.2.4 提交代码触发CICD

Gitlab触发CICD

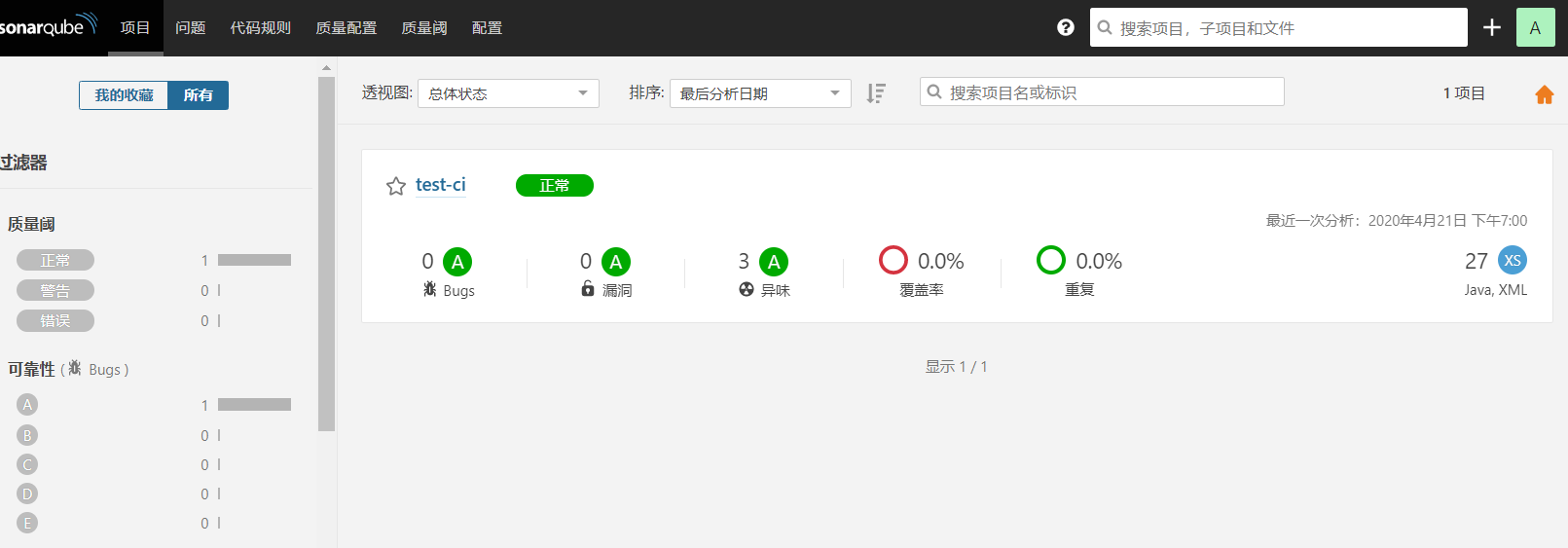


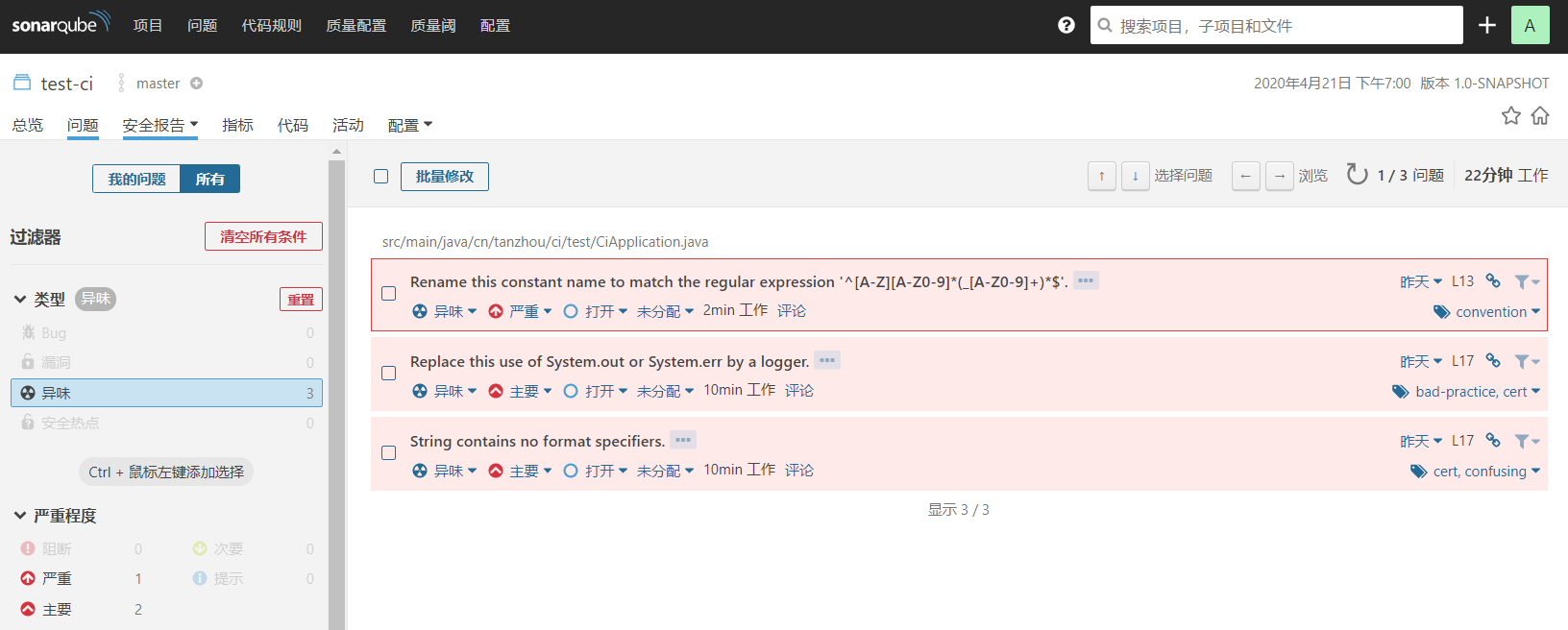
点击running，可以查看执行日志



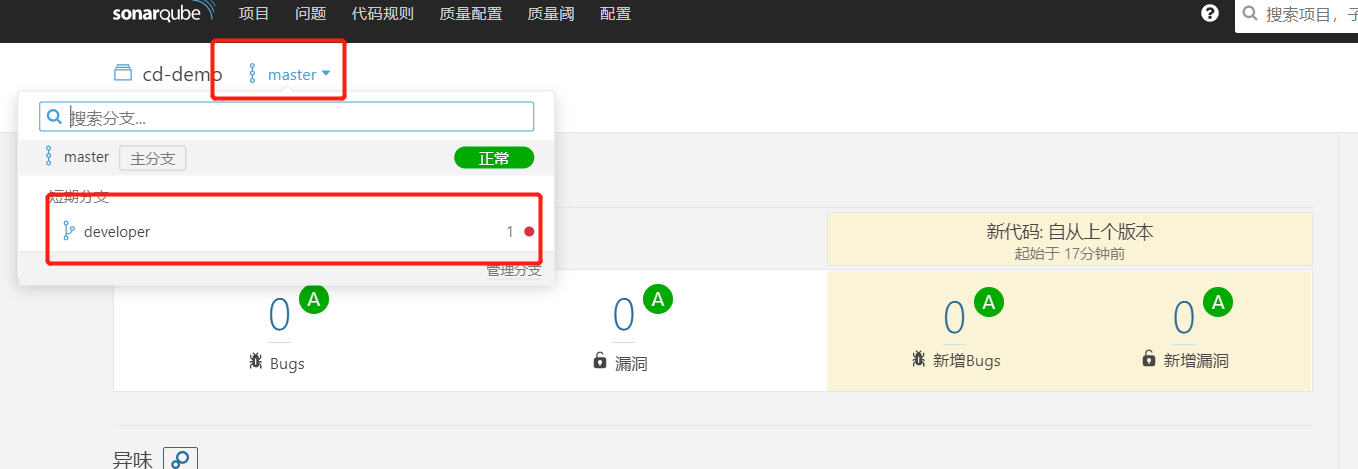
*Tips:如果无图中红框报告连接打印，请检查Dsonar.login参数是否正确，由于无法连接sonarqube上报结果导致*

### 5.2.5 sonarqube代码展示(自动创建项目)

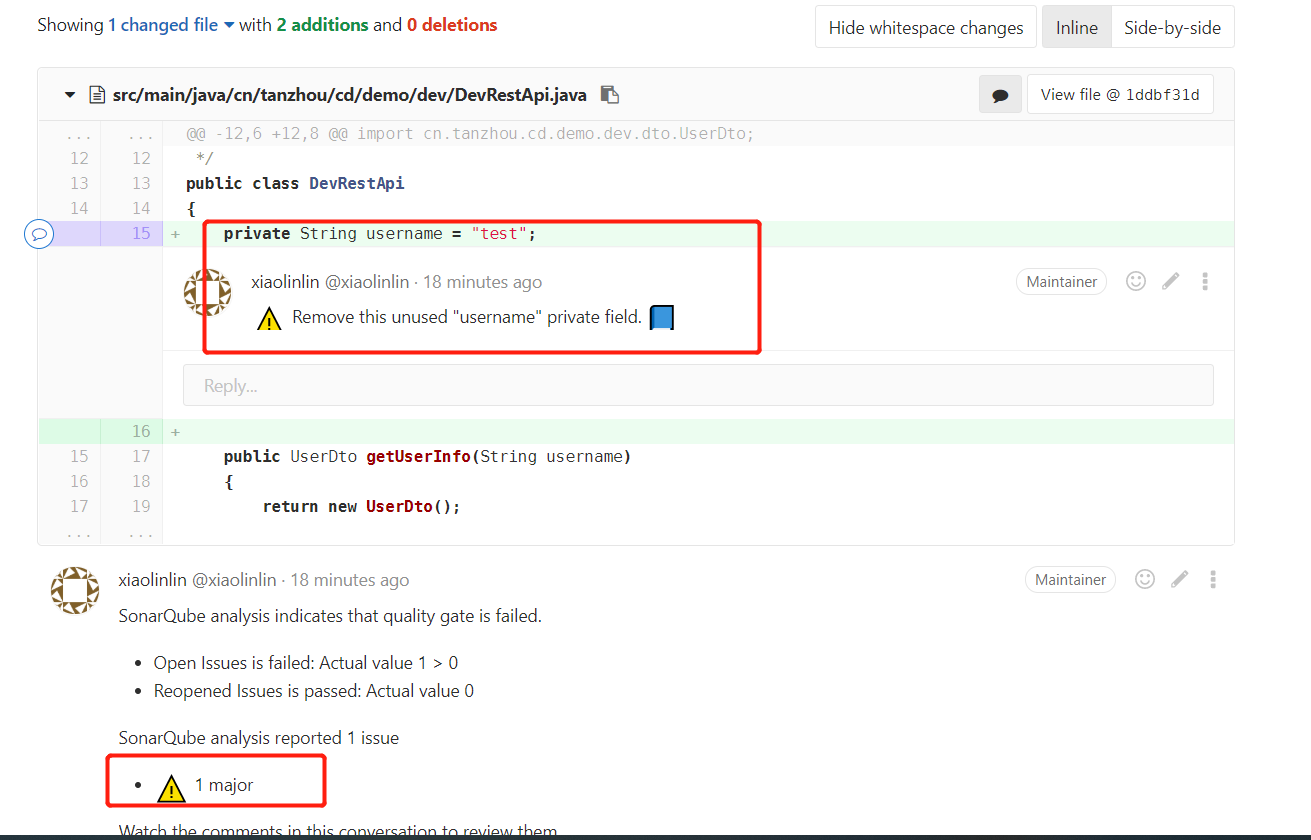




多项目展示：



非master分之回写报告到commit的comment中：



## 5.3 gitlab-ci.yml配置详解

|  |
| --- |
| before\_script:  - echo "每个job之前都会执行"  after\_script:  - echo "每个job之后都会执行"  #全局变量定义，在job中使用 ${variableName}  #内置变量：CI\_COMMIT\_REF\_NAME（项目所在的分支或标签名称）、CI\_JOB\_NAME（任务名称）、CI\_JOB\_STAGE（任务阶段）  variables:  # 定义拉取代码的方式，有3种：clone/fetch/none，默认为clone，速度最慢，每步job都会重新clone一次代码。我们一般将它设置为none，  # 在具体任务里设置为fetch就可以满足需求，毕竟不是每步都需要新代码，那也不符合我们测试的流程  GIT\_STRATEGY: "none" #GIT策略，  # 来指定需要在job之间缓存的文件或目录  cache:  #因为缓存为不同管道和任务间共享，可能会覆盖，所以有时需要设置key  key: ${CI\_COMMIT\_REF\_NAME} # 启用每分支缓存  #key: "$CI\_JOB\_NAME/$CI\_COMMIT\_REF\_NAME" # 启用每个任务和每个分支缓存  untracked: true #缓存所有Git未跟踪的文件  #paths: #缓存哪些文件/文件夹  #- node\_modules/  #- dist/  #表示构建阶段，主要包含 build、test、deploy三个阶段，任务将按此顺序执行  # build阶段job并行，所有build执行ok并行test阶段job，所有test执行ok并行deploy，全部ok标记success，否则标记failure  # yml文件中如未明确定义stages，默认取值就是：build、test、deploy，job中未指定stage，默认取值test  stages:  - build  - test  - deploy  # Jobs 表示构建工作，表示某个 Stage 里面执行的工作,可以在 Stages 里面定义多个 Jobs  # 定义job任务，job\_name 全局唯一且不能是关键字  test-job1:  # test阶段执行  stage: test  tags:  - xx #注册runner的标签  only:  - dev # 只有dev分支提交代码才会执行这个任务  - /^future-.\*$/ #正则匹配，只有future-开头的分支执行  script: #要执行的内容，脚本  - echo "I am job1"  - echo "Just from dev&future-\* branch and test phase!"  # 代码覆盖率设置  coverage: '/Code coverage: \d+\.\d+/'  # 指定该任务执行完毕后，哪些目录或文件需要保留，打成zip包供后续使用  artifacts:  path:  - public  # 定义此作业完成部署的环境名称，job将会执行部署到production环境  #environment:  #name: production  test-job2:  # 默认stage阶段为 test  stage: test  only:  - master # 只有Master分支有提交代码才会执行  script:  - echo "Trigger for master branch!"  allow\_failure: true #允许失败，即不影响下一步构建  build-job:  stage: build  except:  - dev # 除dev环境，其他分之提交点都会执行  script:  - echo "Trigger without dev branc!"  # when触发条件  # on\_success、on\_failure、always、manual(手动)  # 不管前面步骤是否成功，都会执行  when: always  ############################ yml高级用法###########################  # .开头的job忽略执行，做继承、模板方法等  # &(锚)使用隐藏的key,\*(aliases)别名做job模板，后续job可以<<(继承)  .general: &job\_template  stage: deploy  script:  - echo 'Job template'  - path/shell.sh #执行一个脚本  only:  - master  tags:  - runnerTag  # 生产job  .prod: &prod  # 继承general  <<: \*job\_template  # 重写script  script:  - echo 'Prod branch!'  # 各环境配置  .test: &test  <<: \*job\_template  script:  - echo 'Dev branch!'  only:  - test  .dev: &dev  <<: \*job\_template  script:  - echo 'Dev branch!'  only:  - dev  - /^feture-\*$/  prod-job:  <<: \*prod  test-job:  <<: \*test  dev-job:  <<: \*dev |



更多语法详见：<https://docs.gitlab.com/ee/ci/yaml/>