Jenkins

概念

概念

CI/CD 是一种通过在应用开发阶段引入自动化来频繁向客户交付应用的方法。CI/CD 的核心概念是持续集成、持续交付和持续部署。作为一个面向开发和运营团队的解决方案，CI/CD 主要针对在集成新代码时所引发的问题。
具体而言，CI/CD 可让持续自动化和持续监控贯穿于应用的整个生命周期（从集成和测试阶段，到交付和部署）。这些关联的事务通常被统称为“CI/CD 管道”，由开发和运维团队以敏捷方式协同支持。

持续集成（CI：Continuous Integration）

帮助开发人员更加频繁的将代码合并到共享分支或主干中，合并之后会自动触发构建应用，运行不同级别的代码扫描（sonarqube）和自动化测试（单元和集成测试）。

CI/CD 中的“CI”始终指持续集成，它属于开发人员的自动化流程。成功的 CI 意味着应用代码的新更改会定期构建、测试并合并到共享存储库中。该解决方案可以解决在一次开发中有太多应用分支，从而导致相互冲突的问题。
持续集成（CI）可以帮助开发人员更加频繁地（有时甚至每天）将代码更改合并到共享分支或“主干”中。一旦开发人员对应用所做的更改被合并，系统就会通过自动构建应用并运行不同级别的自动化测试（通常是单元测试和集成测试）来验证这些更改，确保这些更改没有对应用造成破坏。这意味着测试内容涵盖了从类和函数到构成整个应用的不同模块。如果自动化测试发现新代码和现有代码之间存在冲突，CI 可以更加轻松地快速修复这些错误。

持续交付（CD：Continuous Delivery）

将通过集成测试的代码合并到一个可以随时部署到生产环境的代码库。

CI/CD 中的“CD”指的是持续交付和/或持续部署，这些相关概念有时会交叉使用。两者都事关管道后续阶段的自动化，但它们有时也会单独使用，用于说明自动化程度。
持续交付通常是指开发人员对应用的更改会自动进行错误测试并上传到存储库，然后由运维团队将其部署到实时生产环境中。旨在解决开发和运维团队之间可见性及沟通较差的问题。因此，持续交付的目的就是确保尽可能减少部署新代码时所需的工作量。
持续部署（另一种“CD”）指的是自动将开发人员的更改从存储库发布到生产环境，以供客户使用。它主要为了解决因手动流程降低应用交付速度，从而使运维团队超负荷的问题。持续部署以持续交付的优势为根基，实现了管道后续阶段的自动化。

持续部署

持续交付的延伸，就是将代码自动发布到生产环境中。

jenkins 安装

1. 下载地址

www.jenkins.io/download

mirrors.jenkins.io/war-stable/　2.222.4

2. 安装java 1.8

```
```

3. install

```
java -jar jenkins.war --httpPort=28080 &
```

浏览器访问：http://ip:28080

安装插件：

> Active Choices Plug-in
>
> Blue Ocean
>
> Blue Ocean Core JS
>
> Blue Ocean Executor Info
>
> Blue Ocean Pipeline Editor
>
> Build Pipline Plugin
>
> Credentials Binding Plugin
>
> Credentials Plugin
>
> Dashboard for Blue Ocean
>
> Declarative Pipline Migration Assistant
>
> Declarative Pipline Migration Assistant API
>
> Display URL API
>
> Display URL for Blue Ocean
>
> Git Parameter Plug-In
>
> Hidden Parameter plugin
>
> Kubernetes CLI Plugin
>
> Kubernetes Plugin
>
> List Git Branches Parameter PlugIn
>
> Parameterized Remote Trigger Plugin
>
> Parameterized Trigger Plugin
>
> Pipeline

声明式流水线

Jenkins pipeline 语法文档：

https://www.jenkins.io/doc/book/pipeline/syntax/

https://www.jenkins.io/zh/doc/book/pipeline/syntax/

变量

定义环境变量

定义流水线环境变量，可以定义全局变量或者stage中定义局部变量，这取决于 environment 指令在流水线内的位置。

```
pipeline {
    agent any
    //全局变量
    environment {
        DISABLE_AUTH = 'true'
        DB_ENGINE    = 'sqlite'
    }

    stages {
        stage('Build') {
            //局部变量
            environment {
                AN_ACCESS_KEY = credentials('my-prefined-secret-text')
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

jenkins自定义环境变量

将变量从shell脚本传递给jenkins，然后在Jenkinsfile中，可以像其他变量一样使用$foo。

```
foo = sh(
  returnStdout: true,
  script: 'date'
)
```

获取git commit短ID和时间戳

```
pipeline {
    agent any
```

```
    stages {
        stage('docker build & push') {
            steps {
                script {
                    env.COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --
pretty=format:'%h'").trim()
                    env.TIMESTRAP = sh(returnStdout: true, script: 'date
+%Y%m%d%H%M%S').trim()
                    env.DOCKER_TAG = "dev_${TIMESTRAP}_${COMMIT_ID}_${BUILD_NUMBER}"
                }
            }
        }
    }
}
```

jenkins默认环境变量

Jenkins中内置了很多环境变量，比如JENKINS_HOME，还有BUILD_NUMBER等。

Jenkins的脚本中能够引用的常用环境变量如下所示。

环境变量	说明	备考
BRANCH_NAME	在multibranch项目中，BRANCH_NAME用于标明构建分支的名称。	-
CHANGE_ID	在multibranch的项目中，相较于特定的变更请求，用于标明变更ID，比如Pull Request	不支持的情况下此环境变量会被unset
CHANGE_URL	在multibranch的项目中，相较于特定的变更请求，用于标明变更的URL	不支持的情况下此环境变量会被unset
CHANGE_TITLE	在multibranch的项目中，相较于特定的变更请求，用于标明变更的标题	不支持的情况下此环境变量会被unset
CHANGE_AUTHOR	在multibranch的项目中，相较于特定的变更请求，用于标明提交变更的人员的名称	不支持的情况下此环境变量会被unset
CHANGE_AUTHOR_DISPLAY_NAME	在multibranch的项目中，相较于特定的变更请求，用于标明提交变更的人员的显示名称	不支持的情况下此环境变量会被unset
CHANGE_AUTHOR_EMAIL	在multibranch的项目中，相较于特定的变更请求，用于标明提交变更的人员的邮件地址	不支持的情况下此环境变量会被unset
CHANGE_TARGET	在multibranch的项目中，相较于特定的变更请求，用于合并后的分支信息等	不支持的情况下此环境变量会被unset
BUILD_NUMBER	当前的构建编号	-
BUILD_ID	在1.597版本后引进，表示当前构建ID	使用YYYY-MM-DD_hh-mm-ss的时间戳以表示之前的构建信息

BUILD_DISPLAY_NAME	当前构建的显示信息	-
JOB_NAME	构建Job的全称，包含项目信息	-
JOB_BASE_NAME	除去项目信息的Job名称	-
BUILD_TAG	构建标签	生成的形为jenkins-JOB_NAME的构建标签
EXECUTOR_NUMBER	执行器编号，用于标识构建器的不同编号。	编号从0开始
NODE_NAME	构建节点的名称	如果在master节点上执行的话，名称为master
NODE_LABELS	节点标签	-
WORKSPACE	构建时使用的工作空间的绝对路径	-
JENKINS_HOME	JENKINS根目录的绝对路径	用于指定Jenkins的Master节点数据存储的路径
JENKINS_URL	Jenkins的URL信息	只有当系统配置中被设定才会显示。
BUILD_URL	构建的URL信息	只有当系统配置中被设定才会显示。
JOB_URL	构建Job的URL信息	只有当系统配置中被设定才会显示。
GIT_COMMIT	git提交的hash码	-
GIT_PREVIOUS_COMMIT	当前分支上次提交的hash码	仅在信息存在的情况下才会显示
GIT_PREVIOUS_SUCCESSFUL_COMMIT	当前分支上次成功构建时提交的hash码	仅在信息存在的情况下才会显示
GIT_BRANCH	远程分支名称	仅在信息存在的情况下才会显示
GIT_LOCAL_BRANCH	本地分支名称	-
GIT_URL	远程URL地址	当存在多个URL地址的情况下，引用方式依次为GIT_URL_1、 GIT_URL_2等
GIT_COMMITTER_NAME	Git提交者的名称	-
GIT_AUTHOR_NAME	Git Author的名称	-
GIT_COMMITTER_EMAIL	Git提交者的email地址	-
GIT_AUTHOR_EMAIL	Git Author的email地址	-
MERCURIAL_REVISION	Mercurial的版本ID信息	-
MERCURIAL_REVISION_SHORT	Mercurial的版本ID缩写	-
MERCURIAL_REVISION_NUMBER	Mercurial的版本号信息	-
MERCURIAL_REVISION_BRANCH	分支版本信息	-
MERCURIAL_REPOSITORY_URL	仓库URL信息	-
SVN_REVISION	Subversion的当前版本信息	-
SVN_URL	当前工作空间中被checkout的Subversion工程的URL地址信息	-

引用方式

```
${env.JENKINS_HOME}
```

以JENKINS_HOME为例，在Jenkinsfile中使用环境变量的方式如下所示

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                sh 'echo Build stage ...'
                echo "[BUILD_ID]           : ${env.BUILD_ID}"
                echo "[BUILD_NUMBER]       : ${env.BUILD_NUMBER}"
                echo "[BUILD_DISPLAY_NAME] : ${env.BUILD_DISPLAY_NAME}"
            }
        }
        stage('Test'){
            steps {
                sh 'echo Test stage ...'
                echo "[JOB_NAME]           : ${env.JOB_NAME}"
                echo "[JOB_BASE_NAME]      : ${env.JOB_BASE_NAME}"
                echo "[BUILD_TAG]          : ${env.BUILD_TAG}"
                echo "[EXECUTOR_NUMBER]    : ${env.EXECUTOR_NUMBER}"
                echo "[NODE_NAME]          : ${env.NODE_NAME}"
                echo "[NODE_LABELS]        : ${env.NODE_LABELS}"
            }
        }
        stage('Deploy') {
            steps {
                sh 'echo Deploy stage ...'
                echo "[WORKSPACE]          : ${env.WORKSPACE}"
                echo "[JENKINS_HOME]       : ${env.JENKINS_HOME}"
                echo "[JENKINS_URL]        : ${env.JENKINS_URL}"
                echo "[BUILD_URL]          : ${env.BUILD_URL}"
                echo "[JOB_URL]            : ${env.JOB_URL}"
                echo "[GIT_COMMIT]         : ${env.GIT_COMMIT}"
            }
        }
    }
  }
```

Jenkins pipeline单引号、双引号和转义字符

Jenkins pipeline的单引号、双引号和转义字符的语法和Linux shell中的语法一致。

在单引号之间的所有特殊字符都失去了特殊含义；

在双引号之间的绝大多数特殊字符都失去了特殊含义，除了以下特例：

- `$`美元号用来提取变量的值
- ` 反冒号用执行命令
- `\`反斜杠用来转义字符

Jenkins pipeline例子

例子1：打印I have $100

```
echo 'I have $100'
echo "I have \$100"
```

例子2：打印PATH环境变量

```
echo "PATH = ${PATH}"
```

例子3：打印It's a dog.

```
echo "It's a dog"
```

例子4：查看python进程号

```
sh "ps -ef | grep python | grep -v grep | awk \'{print \$2}\'"
```

例子5：执行多条命令

```
sh """
    whoami
    pwd
    ls -ltra
"""
```

或

```
sh '''
    whoami
    pwd
    ls -ltra
'''
```

小结

在Jenkins pipeline中，如果没有特殊字符，用单引号和双引号是一样的，如果要执行多行命令，用三个单引号或三个双引号。

如果有特殊字符需要解释，用双引号。需要转义的情况，用`\`转义符。

commit短ID变量

```
stage('get_commit_msg') {
    steps {
        script {
              env.imageTag = sh (script: 'git rev-parse --short HEAD ${GIT_COMMIT}',
returnStdout: true).trim()
            }
       }
}
```

级联变量

联动参数：根据选择不同环境，而调出不同的机器

安装插件：Active Choices Plug-in

Credentials配置

配置harbor的 账号密码，这里是阿里云的

配置gitlba的 SSH 方式的账号密码（参考之前的运维篇），导入私钥.

BlueOcean入门

推荐方式：

```
1.BlueOcean 创建Pipline
2.在gitlab上创建一个独立的项目
3.BlueOcean应用上面创建的gitlab
4.会在gitlab上创建的独立项目中生成Jenkinsfile
5.拷贝上面的Jenkinsfile，修改适配自己真实项目的Jenkinsfile 应用到自己的真实项目中
6.job调用job的方式，去复用公共模块。比如说代码扫描等
```

官方文档：https://www.jenkins.io/projects/blueocean/

使用 Jenkins Pipeline 来自动化部署一个 Kubernetes 应用的方法，在实际的项目中，往往一个代码仓库都会有很多分支的，比如开发、测试、线上这些分支都是分开的，一般情况下开发或者测试的分支希望提交代码后就直接进行 CI/CD 操作，而线上的话最好增加一个人工干预的步骤，这就需要 Jenkins 对代码仓库有多分支的支持，当然这个特性是被 Jenkins 支持的。

Jenkinsfile

```
node('haimaxy-jnlp') {
    stage('Prepare') {
        echo "1.Prepare Stage"
        checkout scm
        script {
            build_tag = sh(returnStdout: true, script: 'git rev-parse --short
HEAD').trim()
            if (env.BRANCH_NAME != 'master') {
                build_tag = "${env.BRANCH_NAME}-${build_tag}"
            }
        }
    }
    stage('Test') {
      echo "2.Test Stage"
    }
    stage('Build') {
        echo "3.Build Docker Image Stage"
        sh "docker build -t cnych/jenkins-demo:${build_tag} ."
    }
    stage('Push') {
        echo "4.Push Docker Image Stage"
        withCredentials([usernamePassword(credentialsId: 'dockerHub', passwordVariable:
'dockerHubPassword', usernameVariable: 'dockerHubUser')]) {
            sh "docker login -u ${dockerHubUser} -p ${dockerHubPassword}"
            sh "docker push cnych/jenkins-demo:${build_tag}"
        }
    }
    stage('Deploy') {
        echo "5. Deploy Stage"
        if (env.BRANCH_NAME == 'master') {
            input "确认要部署线上环境吗? "
        }
        sh "sed -i 's/<BUILD_TAG>/${build_tag}/' k8s.yaml"
        sh "sed -i 's/<BRANCH_NAME>/${env.BRANCH_NAME}/' k8s.yaml"
        sh "kubectl apply -f k8s.yaml --record"
    }
}
```

在第一步中增加了checkout scm命令，用来检出代码仓库中当前分支的代码，为了避免各个环境的镜像 tag 产生冲突，为非 master 分支的代码构建的镜像增加了一个分支的前缀，在第五步中如果是 master 分支的话才增加一个确认部署的流程，其他分支都自动部署，并且还需要替换 k8s.yaml 文件中的环境变量的值。

更改完成后，提交 dev 分支到 github 仓库中。

BlueOcean

官方文档：https://www.jenkins.io/projects/blueocean/

使用 BlueOcean 这种方式来完成此处 CI/CD 的工作，BlueOcean 是 Jenkins 团队从用户体验角度出发，专为 Jenkins Pipeline 重新设计的一套 UI 界面，仍然兼容以前的 fressstyle 类型的 job，BlueOcean 具有以下的一些特性：

- 连续交付（CD）Pipeline 的复杂可视化，允许快速直观的了解 Pipeline 的状态
- 可以通过 Pipeline 编辑器直观的创建 Pipeline
- 需要干预或者出现问题时快速定位，BlueOcean 显示了 Pipeline 需要注意的地方，便于异常处理和提高生产力
- 用于分支和拉取请求的本地集成可以在 GitHub 或者 Bitbucket 中与其他人进行代码协作时最大限度提高开发人员的生产力。

> BlueOcean 可以安装在现有的 Jenkins 环境中，也可以使用 Docker 镜像的方式直接运行，这里直接在现有的 Jenkins 环境中安装 BlueOcean 插件：登录 Jenkins Web UI -> 点击左侧的 Manage Jenkins -> Manage Plugins -> Available -> 搜索查找 BlueOcean -> 点击下载安装并重启

安装完成后，可以在 Jenkins Web UI 首页左侧看到会多一个 Open Blue Ocean 的入口，点击就可以打开，如果之前没有创建过 Pipeline，则打开 Blue Ocean 后会看到一个Create a new pipeline的对话框：

然后点击开始创建一个新的 Pipeline，可以看到可以选择 Git、Bitbucket、GitHub，这里选择 GitHub，可以看到这里需要一个访问 GitHub 仓库权限的 token，在 GitHub 的仓库中创建一个 Personal access token

然后将生成的 token 填入下面的创建 Pipeline 的流程中，然后就有权限选择自己的仓库，包括下面需要构建的仓库，比如这里需要构建的是 jenkins-demo 这个仓库，然后创建 Pipeline 即可：

Blue Ocean 会自动扫描仓库中的每个分支，会为根文件夹中包含Jenkinsfile的每个分支创建一个 Pipeline，比如这里有 master 和 dev 两个分支，并且两个分支下面都有 Jenkinsfile 文件，所以创建完成后会生成两个 Pipeline:

可以看到有两个任务在运行了，可以把 master 分支的任务停止掉，只运行 dev 分支即可，然后点击 dev 这个 pipeline 就可以进入本次构建的详细页面：

在上面的图中每个阶段都可以点击进去查看对应的构建结果，比如可以查看 Push 阶段下面的日志信息：

```
[jenkins-demo_dev-I2WMFUIFQCIFGRPNHN3HU7IZIMHEQMHWPUN2TP6DCYSWHFFFFHOA] Running shell
script

+ docker push ****/jenkins-demo:dev-ee90aa5

The push refers to a repository [docker.io/****/jenkins-demo]

...
```

Docker 镜像的 Tag 为dev-ee90aa5，是符合在 Jenkinsfile 中的定义

更改下 k8s.yaml 将 环境变量的值的标记改成 BRANCH_NAME，当然 Jenkinsfile 也要对应的更改，然后提交代码到 dev 分支并且 push 到 Github 仓库，可以看到 Jenkins Blue Ocean 里面自动触发了一次构建工作，最好同样可以看到本次构建能够正常完成，最后查看下本次构建的结果：

```
$ kubectl get pods
NAME                                  READY     STATUS        RESTARTS   AGE
...
jenkins-demo-648876568d-q5mbx         0/1       Completed     3          57s
...
$ kubectl logs jenkins-demo-648876568d-q5mbx
Hello, Kubernetes! I'm from Jenkins CI!
BRANCH: dev
```

可以看到打印了一句 BRANCH: dev ，证明我本次 CI/CD 是正常的。

现在来把 dev 分支的代码合并到 master 分支，然后来触发一次自动构建：

```
☁  jenkins-demo [dev] git status
On branch dev
nothing to commit, working directory clean
☁  jenkins-demo [dev] git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
☁  jenkins-demo [master] git merge dev
Updating 50e0401..ee90aa5
Fast-forward
 Jenkinsfile | 29 +++++++++--------------------
 k8s.yaml    |  3 +++
 main.go     |  2 ++
 3 files changed, 14 insertions(+), 20 deletions(-)
☁  jenkins-demo [master] git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:cnych/jenkins-demo.git
   50e0401..ee90aa5  master -> master
```

回到 Jenkins 的 Blue Ocean 页面中，可以看到一个 master 分支下面的任务被自动触发了，同样进入详情页可以查看 Push 阶段下面的日志：

```
...
[jenkins-demo_master-XA3VZ5LP4XTCFAHHXIN3G5ZB4XA4J5H6I4DNKOH6JAXZXARF7LYQ] Running
shell script

+ docker push ****/jenkins-demo:ee90aa5
...
```

可以查看到此处推送的镜像 TAG 为 ee90aa5，没有分支的前缀，是不是和前面在 Jenkinsfile 中的定义是一致的，镜像推送完成后，进入 Deploy 阶段的时候可以看到出现了一个暂停的操作，让选择是否需要部署到线上，前面是不是定义的如果是 master 分支的话，在部署的阶段需要人工确认：

点击Proceed才会继续后面的部署工作，确认后，同样可以去 Kubernetes 环境中查看下部署的结果：

```
$ kubectl get pods
NAME                                  READY     STATUS              RESTARTS   AGE
...
jenkins-demo-c69dc6fdf-6ssjf              0/1       Completed   5          4m
...
$ kubectl logs jenkins-demo-c69dc6fdf-6ssjf
Hello, Kubernetes! I'm from Jenkins CI!
BRANCH: maste
```

现在可以看到打印出来的信息是 master，证明部署是没有问题的。

到这里就实现了多分支代码仓库的完整的 CI/CD 流程。

当然这里的示例还是太简单，只是单纯为了说明 CI/CD 的步骤，在后面的课程中，会结合其他的工具进一步对现有的方式进行改造，比如使用 Helm、Gitlab 等等。

另外如果你对声明式的 Pipeline 比较熟悉的话，推荐使用这种方式来编写 Jenkinsfile 文件，因为使用声明式的方式编写的 Jenkinsfile 文件在 Blue Ocean 中不但支持得非常好，还可以直接在 Blue Ocean Editor 中可视化的对的 Pipeline 进行编辑操作，非常方便。

自动化构建流水线设计

1.gitlab代码仓库创建项目

2.开发开发代码

3.Push到 gitlab后执行构建

a) 自动构建

env.gitlabbranch

b) 手动构建

c) 定时构建

4.在 jenkins 调用k8s 创建 Pod 执行构建

a) 代码编译

b) 代码扫描

5.根据Dockerfile 生成镜像

a）对应项目的根目录下

b) 或者放在gitlab 统一管理

c) 每个job 配置单独的变量

- jar 、war 放到基础镜像中
- H5项目， html放到nginx根目录下

6.制作镜像，Push 镜像到镜像仓库

7.Jenkins Slave kubectl -> set 命令更新镜像

```
* 只更新镜像
* Helm更新
```

8.判断程序是否启动

a) -w参数

b) 写脚本判断

9.程序启动，调用测试Job

不构建的流水线：

1.Jenkins 调用镜像仓库接口，返回镜像tag

2.选择对应的tag镜像发版到其他环境

图形化创建Jenkins案例

文档：https://www.jenkins.io/zh/doc/tutorials/create-a-pipeline-in-blue-ocean/

Harbor

阿里云镜像仓库

https://cr.console.aliyun.com/cn-hangzhou/instance/dashboard

```
# login
sudo docker login --username=jet**** registry.cn-hangzhou.aliyuncs.com
```

1. 配置accessKey
2. 在linux安装 CLI工具

https://www.alibabacloud.com/help/zh/doc-detail/139508.htm

```
tar xzvf aliyun-cli-linux-3.0.32-amd64.tgz
sudo cp aliyun /usr/local/bin
aliyun --help

aliyun  configure
# 输入accessKey
```

```
# 获取镜像tag
aliyun cr GetRepoTags --RepoNamespace 命名空间名称 --RepoName 镜像名称 | jq
".data.tags[].tag" -r
```

harbor

具体使用参考，运维篇

```
Harbor镜像仓库地址：172.168.1.249

# 获取项目信息

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/projects/2"

# 获取所有项目信息

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/projects?"

# 搜索镜像

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/search?q=asset"

# 删除项目
```

```
curl -u "admin:Harbor12345" -X DELETE -H "Content-Type: application/json"
"http://172.168.1.249/api/projects/3"

# 创建项目

curl -u "admin:Harbor12345" -X POST -H "Content-Type: application/json"
"http://172.168.1.249/api/projects" -d @createproject.json

createproject.json为文件名,文件内容参考createproject.json

# 0为私有

{

    "project_name": "项目名",

    "public": 0

}

# 创建用户

curl -u "admin:Harbor12345" -X POST -H "Content-Type: application/json"
"http://172.168.1.249/api/users" -d @user.json

文件内容参考user.json

{

    "user_id": 5,

    "username": "test",

    "email": "test@qq.com",

    "password": "Harbor12345",

    "realname": "test",

    "role_id": 0

}

# 获取用户信息,除admin外

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/users"

# 查看当前用户信息
```

```
curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/users/current"

# 删除用户,3是用户user_id

curl -u "admin:Harbor12345" -X DELETE -H "Content-Type: application/json"
"http://172.168.1.249/api/users/34"

# 修改用户密码

curl -u "admin:Harbor12345" -X PUT -H "Content-Type: application/json"
"http://172.168.1.249/api/users/4/password" -d @uppwd.json

# 查看项目相关角色

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/projects/2/members/"

# 项目添加角色

curl -u "jaymarco:Harbor123456" -X POST -H "Content-Type: application/json"
"http://172.168.1.249/api/projects/2/members/" -d @role.json

# 查看镜像

curl -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/repositories?project_id=2&q=镜像名"

# 删除镜像

curl -u "admin:Harbor12345" -X DELETE -H "Content-Type: application/json"
"http://172.168.1.249/api/repositories/marktrace%2Fasset/tags/latest"

# 获取镜像标签

curl -s -u "admin:Harbor12345" -X GET -H "Content-Type: application/json"
"http://172.168.1.249/api/repositories/marktrace%2Fasset/tags/" |grep "digest" -C 2
|grep ""name""
```

Gitlab

具体使用参考，运维篇

下载地址：https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el7/

```
wget https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el7/gitlab-ce-13.0.3-ce.0.el7.x86_64.rpm
yum install gitlab-ce-13.0.3-ce.0.el7.x86_64.rpm -y
```

```
vim /etc/gitlab/gitlab.rb
external_url #修改  gitlab.test.com
gitlab-ctl reconfigure

# win 修改
win hosts
10.4.7.107 gitlab.test.com
```