第一章　基于 K8s 的 DevOps 平台

K8s 技术交流群　　和宽哥一对一交流

课程推荐，提供跪舔式售后服务：

K8s 高薪全栈架构师课程：https://edu.51cto.com/sd/518e5

K8s CKA 认证课程：https://edu.51cto.com/sd/fbbc8

K8s CKS 认证课程：https://edu.51cto.com/sd/affb5

K8s 全栈架构师+CKA 套餐：https://edu.51cto.com/topic/4973.html

超级套购：

K8s 全栈架构师+CKA+CKS：https://edu.51cto.com/topic/5174.html

关注宽哥得永生：

https://edu.51cto.com/lecturer/11062970.html?type=2

1.1 什么是流水线

1.1.1 声明式流水线

在声明式流水线语法中，流水线过程定义在 Pipeline{}中，Pipeline 块定义了整个流水线中完成的所有工作，比如：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                //
            }
        }
        stage('Test') {
            steps {
                //
            }
        }
        stage('Deploy') {
            steps {
                //
```

```
            }
        }
    }
}
```

参数说明：

- agent any：在任何可用的代理上执行流水线或它的任何阶段，也就是执行流水线过程的位置，也可以指定到具体的节点；
- stage：定义流水线的执行过程（相当于一个阶段），比如上文所示的 Build、Test、Deploy，但是这个名字是根据实际情况进行定义的，并非固定的名字；
- steps：执行某阶段具体的步骤。

一个以声明式流水线的语法编写的 Jenkinsfile 文件如下：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'make'
            }
        }
        stage('Test'){
            steps {
                sh 'make check'
                junit 'reports/**/*.xml'
            }
        }
        stage('Deploy') {
            steps {
                sh 'make publish'
            }
        }
    }
}
```

常用参数说明：

- pipeline：声明式流水线的一种特定语法，定义了包含执行整个流水线的所有内容和指令，也是声明式流水线的开始；
- agent：声明式流水线的一种特定语法，指示 Jenkins 为整个流水线分配一个执行器和工作区，也就是在流水线中的步骤在哪个 Agent 上执行，该参数同样可以在 stage 中配置；
- stage：描述流水线阶段的语法块，在脚本式流水线语法中，stage（阶段）块是可选的；
- steps：声明式流水线的一种特定语法，它描述了在这个 stage 中要运行的步骤；
- sh：执行一个 shell 命令；
- junit：用于聚合测试报告。

1.1.2 脚本化流水线

在脚本化流水线语法中，会有一个或多个 Node（节点）块在整个流水线中执行核心工作，比如：

```
Jenkinsfile (Scripted Pipeline)
node {
```

```
    stage('Build') {
        //
    }
    stage('Test') {
        //
    }
    stage('Deploy') {
        //
    }
}
```

参数说明：

- node：在任何可用的代理上执行流水线或它的任何阶段，也可以指定到具体的节点；
- stage：和声明式的含义一致，定义流水线的阶段。Stage 块在脚本化流水线语法中是可选的，然而在脚本化流水线中实现 stage 块，可以清楚地在 Jenkins UI 界面中显示每个 stage 的任务子集。

上一小节的声明式流水线也可以用以下脚本式流水线替代：

```
Jenkinsfile (Scripted Pipeline)
node {
    stage('Build') {
        sh 'make'
    }
    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml'
    }
    stage('Deploy') {
        sh 'make publish'
    }
}
```

1.2 声明式 Pipeline 语法

声明式流水线必须包含在一个 Pipeline 块中，比如以下是一个 Pipeline 块的格式：

```
pipeline {
    /* insert Declarative Pipeline here */
}
```

在声明式流水线中有效的基本语句和表达式遵循与 Groovy 的语法同样的规则，但有以下例外：

- 流水线顶层必须是一个 block，即 pipeline{}；
- 分隔符可以不需要分号，但是每条语句都必须在自己的行上；
- 块只能由 Sections、Directives、Steps 或 assignment statements 组成；
- 属性引用语句被当做是无参数的方法调用，比如 input 会被当做 input()。

1.2.1 Sections

声明式流水线中的 Sections 不是一个关键字或指令，而是包含一个或多个 Agent、Stages、post、Directives 和 Steps 的代码区域块。

1.2.1.1 Agent

Agent 表示整个流水线或特定阶段中的步骤和命令执行的位置，该部分必须在 pipeline 块的顶层被定义，也可以在 stage 中再次定义，但是 stage 级别是可选的。

- any：在任何可用的代理上执行流水线，配置语法：

```
pipeline {
    agent any
}
```

- none：表示该 Pipeline 脚本没有全局的 agent 配置。当顶层的 agent 配置为 none 时，每个 stage 部分都需要包含它自己的 agent。配置语法：

```
pipeline {
    agent none
    stages {
        stage('Stage For Build'){
            agent any
        }
    }
}
```

- label：选择某个具体的节点执行 Pipeline 命令，例如：agent{label 'my-defined-label'}。配置语法：

```
pipeline {
    agent none
    stages {
        stage('Stage For Build'){
           agent { label 'my-slave-label' }
        }
    }
}
```

- node：和 label 配置类似，只不过是可以添加一些额外的配置，比如 customWorkspace：
- dockerfile：使用从源码中包含的 Dockerfile 所构建的容器执行流水线或 stage。此时对应的 agent 写法如下：

```
agent {
    dockerfile {
        filename 'Dockerfile.build'
        dir 'build'
        label 'my-defined-label'
        additionalBuildArgs '--build-arg version=1.0.2'
    }
}
```

- docker：相当于 dockerfile，可以直接使用 docker 字段指定外部镜像即可，可以省去构建的时间。比如使用 maven 镜像进行打包，同时可以指定 args：

```
agent{
    docker{
        image 'maven:3-alpine'
        label 'my-defined-label'
        args '-v /tmp:/tmp'
    }
}
```

- kubernetes：Jenkins 也支持使用 Kubernetes 创建 Slave，也就是常说的动态 Slave。配置示例如下：

```
agent {
    kubernetes {
        label podlabel
        yaml """
```

```
    kind: Pod
    metadata:
      name: jenkins-agent
    spec:
      containers:
      - name: kaniko
        image: gcr.io/kaniko-project/executor:debug
        imagePullPolicy: Always
        command:
        - /busybox/cat
        tty: true
        volumeMounts:
          - name: aws-secret
            mountPath: /root/.aws/
          - name: docker-registry-config
            mountPath: /kaniko/.docker
      restartPolicy: Never
      volumes:
        - name: aws-secret
          secret:
            secretName: aws-secret
        - name: docker-registry-config
          configMap:
            name: docker-registry-config
    """
       }
```

1.2.1.2 配置示例

示例 1：假设有一个 Java 项目，需要用 mvn 命令进行编译，此时可以使用 maven 的镜像作为 agent。配置如下：

```
Jenkinsfile (Declarative Pipeline) // 可以不要此行
pipeline {
    agent { docker 'maven:3-alpine' }
    stages {
        stage('Example Build') {
            steps {
                sh 'mvn -B clean verify'
            }
        }
    }
}
```

示例 2：本示例在流水线顶层将 agent 定义为 none，那么此时 stage 部分就需要必须包含它自己的 agent 部分。在 stage('Example Build')部分使用 maven:3-alpine 执行该阶段步骤，在 stage('Example Test')部分使用 openjdk:8-jre 执行该阶段步骤。此时 Pipeline 如下：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent none
    stages {
        stage('Example Build') {
            agent { docker 'maven:3-alpine' }
            steps {
                echo 'Hello, Maven'
                sh 'mvn --version'
            }
        }
        stage('Example Test') {
```

```
            agent { docker 'openjdk:8-jre' }
            steps {
                echo 'Hello, JDK'
                sh 'java -version'
            }
        }
    }
}
```

示例 3：上述的示例也可以用基于 Kubernetes 的 agent 实现。比如定义具有三个容器的 Pod，分别为 jnlp（负责和 Jenkins Master 通信）、build（负责执行构建命令）、kubectl（负责执行 Kubernetes 相关命令），在 steps 中可以通过 containers 字段，选择在某个容器执行命令：

```
pipeline {
  agent {
    kubernetes {
      cloud 'kubernetes-default'
      slaveConnectTimeout 1200
      yaml '''
apiVersion: v1
kind: Pod
spec:
  containers:
    - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
      image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
      name: jnlp
      imagePullPolicy: IfNotPresent
    - command:
        - "cat"
      image: "registry.cn-beijing.aliyuncs.com/citools/maven:3.5.3"
      imagePullPolicy: "IfNotPresent"
      name: "build"
      tty: true
    - command:
        - "cat"
      image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
      imagePullPolicy: "IfNotPresent"
      name: "kubectl"
      tty: true
'''
}
}

  stages {
    stage('Building') {
      steps {
        container(name: 'build') {
        sh """
          mvm clean install
        """
        }

      }
    }
    stage('Deploy') {
      steps {
      container(name: 'kubectl') {
        sh """
            kubectl get node
        """
        }

      }
    }
```

```
    }
}
```

1.2.1.3 Post

Post 一般用于流水线结束后的进一步处理，比如错误通知等。Post 可以针对流水线不同的结果做出不同的处理，就像开发程序的错误处理，比如 Python 语言的 try catch。Post 可以定义在 Pipeline 或 stage 中，目前支持以下条件：

- always：无论 Pipeline 或 stage 的完成状态如何，都允许运行该 post 中定义的指令；
- changed：只有当前 Pipeline 或 stage 的完成状态与它之前的运行不同时，才允许在该 post 部分运行该步骤；
- fixed：当本次 Pipeline 或 stage 成功，且上一次构建是失败或不稳定时，允许运行该 post 中定义的指令；
- regression：当本次 Pipeline 或 stage 的状态为失败、不稳定或终止，且上一次构建的状态为成功时，允许运行该 post 中定义的指令；
- failure：只有当前 Pipeline 或 stage 的完成状态为失败（failure），才允许在 post 部分运行该步骤，通常这时在 Web 界面中显示为红色；
- success：当前状态为成功（success），执行 post 步骤，通常在 Web 界面中显示为蓝色或绿色；
- unstable：当前状态为不稳定（unstable），执行 post 步骤，通常由于测试失败或代码违规等造成，在 Web 界面中显示为黄色；
- aborted：当前状态为终止（aborted），执行该 post 步骤，通常由于流水线被手动终止触发，这时在 Web 界面中显示为灰色；
- unsuccessful：当前状态不是 success 时，执行该 post 步骤；
- cleanup：无论 pipeline 或 stage 的完成状态如何，都允许运行该 post 中定义的指令。和 always 的区别在于，cleanup 会在其它执行之后执行。

示例：一般情况下 post 部分放在流水线的底部，比如本实例，无论 stage 的完成状态如何，都会输出一条 I will always say Hello again!信息：

```
Jenkinsfile (Declarative Pipeline) // 可以不写该行
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
    post {
        always {
            echo 'I will always say Hello again!'
        }
    }
}
```

也可以将 post 写在 stage：

```
pipeline {
    agent any
```

```
        stages {
            stage('Test') {
                steps {
                    sh 'EXECUTE_TEST_COMMAND'
                }
                post {
                    failure {
                        echo "Pipeline Testing failure..."
                    }
                }
            }
        }
    }
```

1.2.1.4 Stages

Stages 包含一个或多个 stage 指令，同时可以在 stage 中的 steps 块中定义真正执行的指令。

比如创建一个流水线，stages 包含一个名为 Example 的 stage，该 stage 执行 echo 'Hello World' 命令输出 Hello World 字符串：

```
    Jenkinsfile (Declarative Pipeline)
    pipeline {
        agent any
        stages {
            stage('Example') {
                steps {
                    echo 'Hello World ${env.BUILD_ID}'
                }
            }
        }
    }
```

1.2.1.5 Steps

Steps 部分在给定的 stage 指令中执行的一个或多个步骤，比如在 steps 定义执行一条 shell 命令：

```
    Jenkinsfile (Declarative Pipeline)
    pipeline {
        agent any
        stages {
            stage('Example') {
                steps {
                    echo 'Hello World'
                }
            }
        }
    }
```

或者是使用 sh 字段执行多条指令：

```
    Jenkinsfile (Declarative Pipeline)
    pipeline {
        agent any
        stages {
            stage('Example') {
                steps {
                    sh """
                        echo 'Execute building...'
```

```
                    mvn clean install
                """
            }
        }
    }
}
```

1.2.2 Directives

Directives 可用于一些执行 stage 时的条件判断或预处理一些数据，和 Sections 一致，Directives 不是一个关键字或指令，而是包含了 environment、options、parameters、triggers、stage、tools、input、when 等配置。

1.2.2.1 Environment

Environment 主要用于在流水线中配置的一些环境变量，根据配置的位置决定环境变量的作用域。可以定义在 pipeline 中作为全局变量，也可以配置在 stage 中作为该 stage 的环境变量。

该指令支持一个特殊的方法 credentials()，该方法可用于在 Jenkins 环境中通过标识符访问预定义的凭证。对于类型为 Secret Text 的凭证，credentials()可以将该 Secret 中的文本内容赋值给环境变量。对于类型为标准的账号密码型的凭证，指定的环境变量为 username 和 password，并且也会定义两个额外的环境变量，分别为 MYVARNAME_USR 和 MYVARNAME_PSW。

假如需要定义个变量名为 CC 的全局变量和一个名为 AN_ACCESS_KEY 的局部变量，并且用 credentials 读取一个 Secret 文本，可以通过以下方式定义：


```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    environment {  // Pipeline 中定义，属于全局变量
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment {  // 定义在 stage 中，属于局部变量
                AN_ACCESS_KEY = credentials('my-prefined-secret-text')
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

1.2.2.2 Options

Jenkins 流水线支持很多内置指令，比如 retry 可以对失败的步骤进行重复执行 n 次，可以根据不同的指令实现不同的效果。比较常用的指令如下：

- buildDiscarder：保留多少个流水线的构建记录。比如：options { buildDiscarder(logRotator(numToKeepStr: '1')) }；
- disableConcurrentBuilds：禁止流水线并行执行，防止并行流水线同时访问共享资源导

致流水线失败。比如：options { disableConcurrentBuilds() }；

- disableResume：如果控制器重启，禁止流水线自动恢复。比如：options { disableResume() }；
- newContainerPerStage：agent 为 docker 或 dockerfile 时，每个阶段将在同一个节点的新容器中运行，而不是所有的阶段都在同一个容器中运行。比如：options { newContainerPerStage () }；
- quietPeriod：流水线静默期，也就是触发流水线后等待一会在执行。比如：options { quietPeriod(30) }；
- retry：流水线失败后重试次数。比如：options { retry(3) }；
- timeout：设置流水线的超时时间，超过流水线时间，job 会自动终止。比如：options { timeout(time: 1, unit: 'HOURS') }；
- timestamps：为控制台输出时间戳。比如：options { timestamps() }。

配置示例如下，只需要添加 options 字段即可：

```
pipeline {
    agent any
    options {
        timeout(time: 1, unit: 'HOURS')
        timestamps()
    }
    stages {
        stage('Example') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

Option 除了写在 Pipeline 顶层，还可以写在 stage 中，但是写在 stage 中的 option 仅支持 retry、timeout、timestamps，或者是和 stage 相关的声明式选项，比如 skipDefaultCheckout。处于 stage 级别的 options 写法如下：

```
pipeline {
    agent any
    stages {
        stage('Example') {
            options {
                timeout(time: 1, unit: 'HOURS')
            }
            steps {
                echo 'Hello World'
            }
        }
    }
}
```

1.2.2.3 Parameters

Parameters 提供了一个用户在触发流水线时应该提供的参数列表，这些用户指定参数的值可以通过 params 对象提供给流水线的 step（步骤）。

目前支持的参数类型如下：

- string：字符串类型的参数，例如：parameters { string(name: 'DEPLOY_ENV', defaultValue:

'staging', description: '') }，表示定义一个名为 DEPLOY_ENV 的字符型变量，默认值为 staging；

- text：文本型参数，一般用于定义多行文本内容的变量。例如 parameters { text(name: 'DEPLOY_TEXT', defaultValue: 'One\nTwo\nThree\n', description: '') }，表示定义一个名为 DEPLOY_TEXT 的变量，默认值是'One\nTwo\nThree\n'；
- booleanParam：布尔型参数，例如: parameters { booleanParam(name: 'DEBUG_BUILD', defaultValue: true, description: '') }；
- choice：选择型参数，一般用于给定几个可选的值，然后选择其中一个进行赋值，例如：parameters { choice(name: 'CHOICES', choices: ['one', 'two', 'three'], description: '') }，表示定义一个名为 CHOICES 的变量，可选的值为 one、two、three；
- password：密码型变量，一般用于定义敏感型变量，在 Jenkins 控制台会输出为*。例如：parameters { password(name: 'PASSWORD', defaultValue: 'SECRET', description: 'A secret password') }，表示定义一个名为 PASSWORD 的变量，其默认值为 SECRET。

Parameters 用法如下：

```
    pipeline {
        agent any
        parameters {
            string(name: 'PERSON', defaultValue: 'Mr Jenkins', description:
'Who should I say hello to?')

            text(name: 'BIOGRAPHY', defaultValue: '', description: 'Enter some
information about the person')

            booleanParam(name: 'TOGGLE', defaultValue: true, description:
'Toggle this value')

            choice(name: 'CHOICE', choices: ['One', 'Two', 'Three'],
description: 'Pick something')

            password(name: 'PASSWORD', defaultValue: 'SECRET', description:
'Enter a password')
        }
        stages {
            stage('Example') {
                steps {
                    echo "Hello ${params.PERSON}"

                    echo "Biography: ${params.BIOGRAPHY}"

                    echo "Toggle: ${params.TOGGLE}"

                    echo "Choice: ${params.CHOICE}"

                    echo "Password: ${params.PASSWORD}"
                }
            }
        }
    }
```

1.2.2.4 Triggers

在 Pipeline 中可以用 triggers 实现自动触发流水线执行任务，可以通过 Webhook、Cron、pollSCM 和 upstream 等方式触发流水线。

假如某个流水线构建的时间比较长，或者某个流水线需要定期在某个时间段执行构建，可以使用 **cron** 配置触发器，比如周一到周五每隔四个小时执行一次：

```
    pipeline {
        agent any
        triggers {
            cron('H */4 * * 1-5')
        }
        stages {
            stage('Example') {
                steps {
                    echo 'Hello World'
                }
            }
        }
    }
```

注意：H 的意思不是 HOURS 的意思，而是 Hash 的缩写。主要为了解决多个流水线在同一时间同时运行带来的系统负载压力。

使用 **cron** 字段可以定期执行流水线，如果代码更新想要重新触发流水线，可以使用 **pollSCM** 字段：

```
    pipeline {
        agent any
        triggers {
            pollSCM('H */4 * * 1-5')
        }
        stages {
            stage('Example') {
                steps {
                    echo 'Hello World'
                }
            }
        }
    }
```

Upstream 可以根据上游 job 的执行结果决定是否触发该流水线。比如当 job1 或 job2 执行成功时触发该流水线：

```
    pipeline {
        agent any
        triggers {
            upstream(upstreamProjects: 'job1,job2', threshold:
hudson.model.Result.SUCCESS)
        }
        stages {
            stage('Example') {
                steps {
                    echo 'Hello World'
                }
            }
        }
    }
```

目前支持的状态有 SUCCESS、UNSTABLE、FAILURE、NOT_BUILT、ABORTED 等。

1.2.2.5 Input

Input 字段可以实现在流水线中进行交互式操作，比如选择要部署的环境、是否继续执行某个阶段等。

配置 Input 支持以下选项：

- message：必选，需要用户进行 input 的提示信息，比如：“是否发布到生产环境？”；
- id：可选，input 的标识符，默认为 stage 的名称；
- ok：可选，确认按钮的显示信息，比如：“确定”、“允许”；
- submitter：可选，允许提交 input 操作的用户或组的名称，如果为空，任何登录用户均可提交 input；
- parameters：提供一个参数列表供 input 使用。

假如需要配置一个提示消息为“还继续么”、确认按钮为“继续”、提供一个 PERSON 的变量的参数，并且只能由登录用户为 alice 和 bob 提交的 input 流水线：

```
    pipeline {
        agent any
        stages {
            stage('Example') {
                input {
                    message "还继续么?"
                    ok "继续"
                    submitter "alice,bob"
                    parameters {
                        string(name: 'PERSON', defaultValue: 'Mr Jenkins',
description: 'Who should I say hello to?')
                    }
                }
                steps {
                    echo "Hello, ${PERSON}, nice to meet you."
                }
            }
        }
    }
```


1.2.2.6 When

When 指令允许流水线根据给定的条件决定是否应该执行该 stage，when 指令必须包含至少一个条件。如果 when 包含多个条件，所有的子条件必须都返回 True，stage 才能执行。

When 也可以结合 not、allOf、anyOf 语法达到更灵活的条件匹配。

目前比较常用的内置条件如下：

- branch：当正在构建的分支与给定的分支匹配时，执行这个 stage，例如：when { branch 'master' }。注意，branch 只适用于多分支流水线；
- changelog：匹配提交的 changeLog 决定是否构建，例如：when { changelog '.*^\\[DEPENDENCY\\] .+$' }；
- environment：当指定的环境变量和给定的变量匹配时，执行这个 stage，例如：when { environment name: 'DEPLOY_TO', value: 'production' }；
- equals：当期望值和实际值相同时，执行这个 stage，例如：when { equals expected: 2, actual: currentBuild.number }；
- expression：当指定的 Groovy 表达式评估为 True，执行这个 stage，例如：when { expression { return params.DEBUG_BUILD } }；
- tag：如果 TAG_NAME 的值和给定的条件匹配，执行这个 stage，例如：when { tag "release-*" }；
- not：当嵌套条件出现错误时，执行这个 stage，必须包含一个条件，例如：when { not { branch 'master' } }；
- allOf：当所有的嵌套条件都正确时，执行这个 stage，必须包含至少一个条件，例如：

when { allOf { branch 'master'; environment name: 'DEPLOY_TO', value: 'production' } }；

- anyOf：当至少有一个嵌套条件为 True 时，执行这个 stage，例如：when { anyOf { branch 'master'; branch 'staging' } }。

示例 1：当分支为 production 时，执行 **Example Deploy** 步骤：

```
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

也可以同时配置多个条件，比如分支是 production，而且 DEPLOY_TO 变量的值为 production 时，才执行 **Example Deploy**：

```
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
                environment name: 'DEPLOY_TO', value: 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

也可以使用 anyOf 进行匹配其中一个条件即可，比如分支为 production，DEPLOY_TO 为 production 或 staging 时执行 Deploy：

```
pipeline {
    agent any
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                branch 'production'
                anyOf {
                    environment name: 'DEPLOY_TO', value: 'production'
                    environment name: 'DEPLOY_TO', value: 'staging'
                }
```

```
                }
                steps {
                    echo 'Deploying'
                }
            }
        }
    }
```

也可以使用 expression 进行正则匹配，比如当 BRANCH_NAME 为 production 或 staging，并且 DEPLOY_TO 为 production 或 staging 时才会执行 Example Deploy：

```
    pipeline {
        agent any
        stages {
            stage('Example Build') {
                steps {
                    echo 'Hello World'
                }
            }
            stage('Example Deploy') {
                when {
                    expression { BRANCH_NAME ==~ /(production|staging)/ }
                    anyOf {
                        environment name: 'DEPLOY_TO', value: 'production'
                        environment name: 'DEPLOY_TO', value: 'staging'
                    }
                }
                steps {
                    echo 'Deploying'
                }
            }
        }
    }
```

默认情况下，如果定义了某个 stage 的 agent，在进入该 stage 的 agent 后，该 stage 的 when 条件才会被评估，但是可以通过一些选项更改此选项。比如在进入 stage 的 agent 前评估 when，可以使用 beforeAgent，当 when 为 true 时才进行该 stage。

目前支持的前置条件如下：

- beforeAgent：如果 beforeAgent 为 true，则会先评估 when 条件。在 when 条件为 true 时，才会进入该 stage；
- beforeInput：如果 beforeInput 为 true，则会先评估 when 条件。在 when 条件为 true 时，才会进入到 input 阶段；
- beforeOptions：如果 beforeInput 为 true，则会先评估 when 条件。在 when 条件为 true 时，才会进入到 options 阶段；

注意
beforeOptions **优先级大于** beforeInput **大于** beforeAgent

配置一个 beforeAgent 示例如下：

```
    pipeline {
        agent none
        stages {
            stage('Example Build') {
                steps {
                    echo 'Hello World'
                }
            }
            stage('Example Deploy') {
                agent {
```

```
                label "some-label"
            }
            when {
                beforeAgent true
                branch 'production'
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

配置一个 beforeInput 示例如下：

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                beforeInput true
                branch 'production'
            }
            input {
                message "Deploy to production?"
                id "simple-input"
            }
            steps {
                echo 'Deploying'
            }
        }
    }
}
```

配置一个 beforeOptions 示例如下：

```
pipeline {
    agent none
    stages {
        stage('Example Build') {
            steps {
                echo 'Hello World'
            }
        }
        stage('Example Deploy') {
            when {
                beforeOptions true
                branch 'testing'
            }
            options {
                lock label: 'testing-deploy-envs', quantity: 1, variable:
'deployEnv'
            }
            steps {
                echo "Deploying to ${deployEnv}"
            }
        }
    }
}
```

1.2.3 Parallel

在声明式流水线中可以使用 Parallel 字段，即可很方便的实现并发构建，比如对分支 A、B、C 进行并行处理：

```
pipeline {
    agent any
    stages {
        stage('Non-Parallel Stage') {
            steps {
                echo 'This stage will be executed first.'
            }
        }
        stage('Parallel Stage') {
            when {
                branch 'master'
            }
            failFast true
            parallel {
                stage('Branch A') {
                    agent {
                        label "for-branch-a"
                    }
                    steps {
                        echo "On Branch A"
                    }
                }
                stage('Branch B') {
                    agent {
                        label "for-branch-b"
                    }
                    steps {
                        echo "On Branch B"
                    }
                }
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

设置 failFast 为 true 表示并行流水线中任意一个 stage 出现错误，其它 stage 也会立即终止。也可以通过 options 配置在全局：

```
pipeline {
    agent any
```

```
    options {
        parallelsAlwaysFailFast()
    }
    stages {
        stage('Non-Parallel Stage') {
            steps {
                echo 'This stage will be executed first.'
            }
        }
        stage('Parallel Stage') {
            when {
                branch 'master'
            }
            parallel {
                stage('Branch A') {
                    agent {
                        label "for-branch-a"
                    }
                    steps {
                        echo "On Branch A"
                    }
                }
                stage('Branch B') {
                    agent {
                        label "for-branch-b"
                    }
                    steps {
                        echo "On Branch B"
                    }
                }
                stage('Branch C') {
                    agent {
                        label "for-branch-c"
                    }
                    stages {
                        stage('Nested 1') {
                            steps {
                                echo "In stage Nested 1 within Branch C"
                            }
                        }
                        stage('Nested 2') {
                            steps {
                                echo "In stage Nested 2 within Branch C"
                            }
                        }
                    }
                }
            }
        }
    }
}
```

1.3 Jenkinsfile 的使用

上面讲过流水线支持两种语法，即声明式和脚本式，这两种语法都支持构建持续交付流水线。并且都可以用来在 Web UI 或 Jenkinsfile 中定义流水线，不过通常将 Jenkinsfile 放置于代码仓库中（当然也可以放在单独的代码仓库中进行管理）。

创建一个 Jenkinsfile 并将其放置于代码仓库中，有以下好处：

- 方便对流水线上的代码进行复查/迭代；
- 对管道进行审计跟踪；
- 流水线真正的源代码能够被项目的多个成员查看和编辑。

1.3.1 环境变量

1. 静态变量

Jenkins 有许多内置变量可以直接在 Jenkinsfile 中使用，可以通过 JENKINS_URL/pipeline-syntax/globals#env 获取完整列表。目前比较常用的环境变量如下：

- BUILD_ID：当前构建的 ID，与 Jenkins 版本 1.597+中的 BUILD_NUMBER 完全相同；
- BUILD_NUMBER：当前构建的 ID，和 BUILD_ID 一致；
- BUILD_TAG：用来标识构建的版本号，格式为：jenkins-${JOB_NAME}-${BUILD_NUMBER}，可以对产物进行命名，比如生产的 jar 包名字、镜像的 TAG 等；
- BUILD_URL：本次构建的完整 URL，比如：http://buildserver/jenkins/job/MyJobName/17/；
- JOB_NAME：本次构建的项目名称；
- NODE_NAME：当前构建节点的名称；
- JENKINS_URL：Jenkins 完整的 URL，需要在 System Configuration 设置；
- WORKSPACE：执行构建的工作目录。

上述变量会保存在一个 Map 中，可以使用 env.BUILD_ID 或 env.JENKINS_URL 引用某个内置变量：


```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    stages {
        stage('Example') {
            steps {
                echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
            }
        }
    }
}
```

对应的脚本式流水线如下：

```
Jenkinsfile (Scripted Pipeline)
node {
    echo "Running ${env.BUILD_ID} on ${env.JENKINS_URL}"
}
```

除了上述默认的环境变量，也可以手动配置一些环境变量：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    environment {
        CC = 'clang'
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
```

```
            steps {
                sh 'printenv'
            }
        }
    }
}
```

上述配置了两个环境变量，一个 CC 值为 clang，另一个是 DEBUG_FLAGS 值为-g。但是两者定义的位置不一样，CC 位于顶层，适用于整个流水线，而 DEBUG_FLAGS 位于 stage 中，只适用于当前 stage。

2. 动态变量

动态变量是根据某个指令的结果进行动态赋值，变量的值根据指令的执行结果而不同。如下所示：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent any
    environment {
        // 使用 returnStdout
        CC = """${sh(
                returnStdout: true,
                script: 'echo "clang"'
            )}"""
        // 使用 returnStatus
        EXIT_STATUS = """${sh(
                returnStatus: true,
                script: 'exit 1'
            )}"""
    }
    stages {
        stage('Example') {
            environment {
                DEBUG_FLAGS = '-g'
            }
            steps {
                sh 'printenv'
            }
        }
    }
}
```

- returnStdout：将命令的执行结果赋值给变量，比如上述的命令返回的是 clang，此时 CC 的值为“clang ”。注意后面多了一个空格，可以用.trim()将其删除；
- returnStatus：将命令的执行状态赋值给变量，比如上述命令的执行状态为 1，此时 EXIT_STATUS 的值为 1。

1.3.2 凭证管理

Jenkins 的声明式流水线语法有一个 credentials()函数，它支持 secret text（加密文本）、username 和 password（用户名和密码）以及 secret file（加密文件）等。接下来看一下一些常用的凭证处理方法。

1. 加密文本

本实例演示将两个 Secret 文本凭证分配给单独的环境变量来访问 Amazon Web 服务，需要提前创建这两个文件的 credentials（实践的章节会有演示），Jenkinsfile 文件的内容如下：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
```

```
        agent {
            // Define agent details here
        }
        environment {
            AWS_ACCESS_KEY_ID     = credentials('jenkins-aws-secret-key-id')
            AWS_SECRET_ACCESS_KEY = credentials('jenkins-aws-secret-access-
key')
        }
        stages {
            stage('Example stage 1') {
                steps {
                    //
                }
            }
            stage('Example stage 2') {
                steps {
                    //
                }
            }
        }
    }
```

说明：

上述示例定义了两个全局变量 AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY，这两个变量引用的是 credentials 的两个加密文本，并且这两个变量均可以在 stages 直接引用（通过$AWS_SECRET_ACCESS_KEY 和$AWS_ACCESS_KEY_ID）。

注　意
如果在 steps 中使用 echo $AWS_ACCESS_KEY_ID，此时返回的是****，加密内容不会被显示出来。

2. 用户名密码

本示例用来演示 credentials 账号密码的使用，比如使用一个公用账户访问 Bitbucket、GitLab、Harbor 等。假设已经配置完成了用户名密码形式的 credentials，凭证 ID 为 jenkins-bitbucket-common-creds。

可以用以下方式设置凭证环境变量（BITBUCKET_COMMON_CREDS 名称可以自定义）：

```
    environment {
        BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-common-
creds')
    }
```

上述的配置会自动生成 3 个环境变量：

- BITBUCKET_COMMON_CREDS：包含一个以冒号分隔的用户名和密码，格式为 username:password；
- BITBUCKET_COMMON_CREDS_USR：仅包含用户名的附加变量；
- BITBUCKET_COMMON_CREDS_PSW：仅包含密码的附加变量。

此时，调用用户名密码的 Jenkinsfile 如下：

```
    Jenkinsfile (Declarative Pipeline)
    pipeline {
        agent {
            // Define agent details here
        }
        stages {
```

```
        stage('Example stage 1') {
            environment {
                BITBUCKET_COMMON_CREDS = credentials('jenkins-bitbucket-
common-creds')
            }
            steps {
                //
            }
        }
        stage('Example stage 2') {
            steps {
                //
            }
        }
    }
}
```

注 意
此时环境变量的凭证仅作用于 stage 1，也可以配置在顶层对全局生效

3. 加密文件

需要加密保存的文件，也可以使用 credential，比如链接到 Kubernetes 集群的 kubeconfig 文件等。

假如已经配置好了一个 kubeconfig 文件，此时可以在 Pipeline 中引用该文件：

```
Jenkinsfile (Declarative Pipeline)
pipeline {
    agent {
        // Define agent details here
    }
    environment {
        MY_KUBECONFIG = credentials('my-kubeconfig')
    }
    stages {
        stage('Example stage 1') {
            steps {
                sh("kubectl --kubeconfig $MY_KUBECONFIG get pods")
            }
        }
    }
}
```

更多其它类型的凭证可以参考：https://www.jenkins.io/doc/book/pipeline/jenkinsfile/#handling-credentials。

1.3.3 参数处理

声明式流水线支持很多开箱即用的参数，可以让流水线接收不同的参数以达到不同的构建效果，在 Directives 小节讲解的参数均可用在流水线中。

在 Jenkinsfile 中指定的 parameters 会在 Jenkins Web UI 自动生成对应的参数列表，此时可以在 Jenkins 页面点击 Build With Parameters 来指定参数的值，这些参数可以通过 params 变量被成员访问。

假设在 Jenkinsfile 中配置了名为 Greeting 的字符串参数，可以通过${params.Greeting}访问该参数，比如：

```
Jenkinsfile (Declarative Pipeline)
```

```
    pipeline {
        agent any
        parameters {
            string(name: 'Greeting', defaultValue: 'Hello', description: 'How
should I greet the world?')
        }
        stages {
            stage('Example') {
                steps {
                    echo "${params.Greeting} World!"
                }
            }
        }
    }
```

对应的脚本式流水线如下：

```
    Jenkinsfile (Scripted Pipeline)
    properties([parameters([string(defaultValue: 'Hello', description: 'How
should I greet the world?', name: 'Greeting')])])

    node {
        echo "${params.Greeting} World!"
    }
```

1.3.4 使用多个代理

流水线允许在 Jenkins 环境中使用多个代理，这有助于更高级的用例，例如跨多个平台执行构建、测试等。

比如，在 Linux 和 Windows 系统的不同 agent 上进行测试：

```
    Jenkinsfile (Declarative Pipeline)
    pipeline {
        agent none
        stages {
            stage('Build') {
                agent any
                steps {
                    checkout scm
                    sh 'make'
                    stash includes: '**/target/*.jar', name: 'app'
                }
            }
            stage('Test on Linux') {
                agent {
                    label 'linux'
                }
                steps {
                    unstash 'app'
                    sh 'make check'
                }
                post {
                    always {
                        junit '**/target/*.xml'
                    }
                }
            }
            stage('Test on Windows') {
                agent {
                    label 'windows'
```

```
            }
            steps {
                unstash 'app'
                bat 'make check'
            }
            post {
                always {
                    junit '**/target/*.xml'
                }
            }
        }
    }
}
```

1.4 DevOps 平台建设

首先先来学习下在 Kubernetes 中进行 CICD 的过程，一般的步骤如下：

1) 在 GitLab 中创建对应的项目；
2) 配置 Jenkins 集成 Kubernetes 集群，后期 Jenkins 的 Slave 将为在 Kubernetes 中动态创建的 Slave；
3) Jenkins 创建对应的任务（Job），集成该项目的 Git 地址和 Kubernetes 集群；
4) 开发者将代码提交到 GitLab；
5) 如有配置钩子，推送（Push）代码会自动触发 Jenkins 构建，如没有配置钩子，需要手动构建；
6) Jenkins 控制 Kubernetes（使用的是 Kubernetes 插件）创建 Jenkins Slave（Pod 形式）；
7) Jenkins Slave 根据流水线（Pipeline）定义的步骤执行构建；
8) 通过 Dockerfile 生成镜像；
9) 将镜像提送（Push）到私有 Harbor（或者其它的镜像仓库）；
10) Jenkins 再次控制 Kubernetes 进行最新的镜像部署；
11) 流水线结束删除 Jenkins Slave。

1.4.1 Jenkins 安装

1.4.1.1 Jenkins 安装

首先需要一个 Linux 服务器，配置不低于 2C4G 和 40G 硬盘。首先安装 Docker：

```
    # yum install -y yum-utils device-mapper-persistent-data lvm2
    # yum-config-manager --add-repo https://mirrors.aliyun.com/docker-
ce/linux/centos/docker-ce.repo
    # sed -i -e '/mirrors.cloud.aliyuncs.com/d' -e '/mirrors.aliyuncs.com/d'
/etc/yum.repos.d/CentOS-Base.repo
    # yum install docker-ce-19.03.* docker-ce-cli-19.03.* -y
    # systemctl daemon-reload && systemctl enable --now docker
```

创建 Jenkins 的数据目录，防止容器重启后数据丢失：

```
    # mkdir /data/jenkins_data -p
    # chmod -R 777 /data/jenkins_data
```

启动 Jenkins，并配置管理员账号密码为 admin / admin123：

```
    # docker run -d --name=jenkins --restart=always -e
JENKINS_PASSWORD=admin123 -e JENKINS_USERNAME=admin -e
```

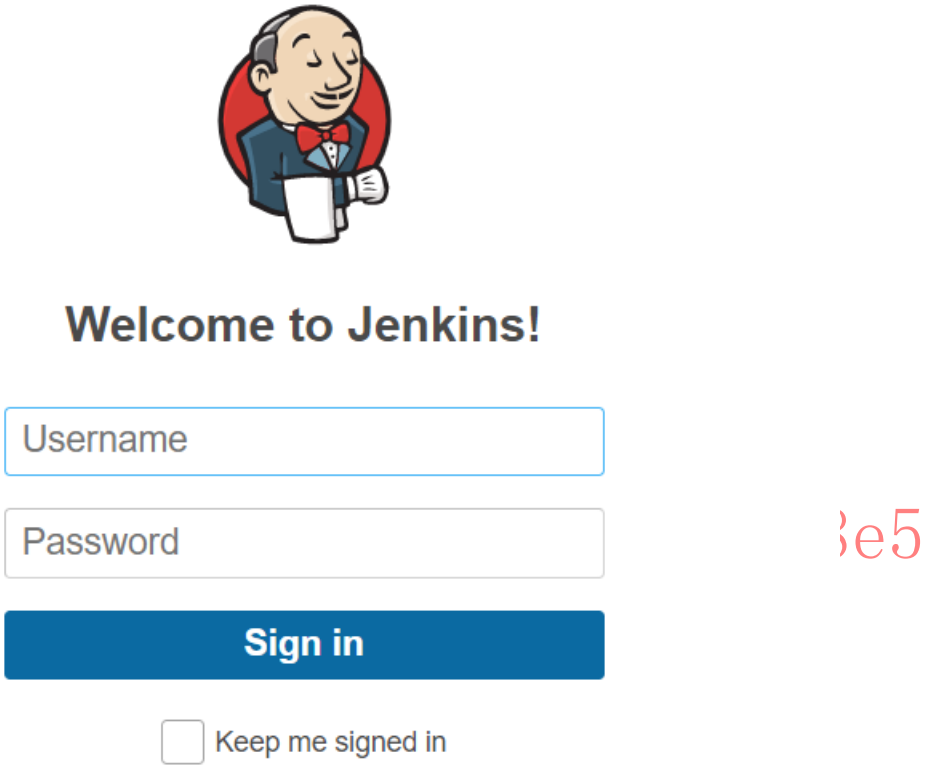
```
JENKINS_HTTP_PORT_NUMBER=8080 -p 8080:8080 -p 50000:50000 -v
/data/jenkins_data:/bitnami/jenkins bitnami/jenkins:2.303.1-debian-10-r29
    e987d004e62b8412e33911eea2c37ffc81b559fb6c3075060b2ea78bac3cae45
```

其中 8080 端口为 Jenkins Web 界面的端口，50000 是 jnlp 使用的端口，后期 Jenkins Slave 需要使用 50000 端口和 Jenkins 主节点通信。

查看 Jenkins 日志：

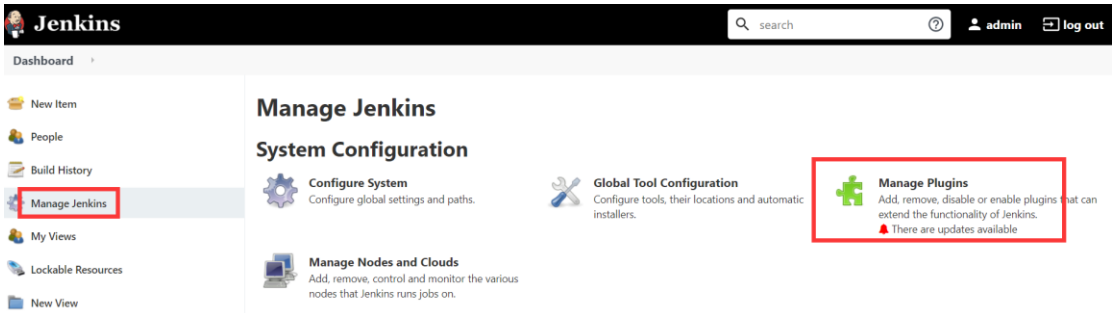
```
# docker logs -f  Jenkins
... # 查看到这条日志说明 Jenkins 已完成启动
INFO: Jenkins is fully up and running
```

之后通过 Jenkins 宿主机的 IP+8080 即可访问 Jenkins



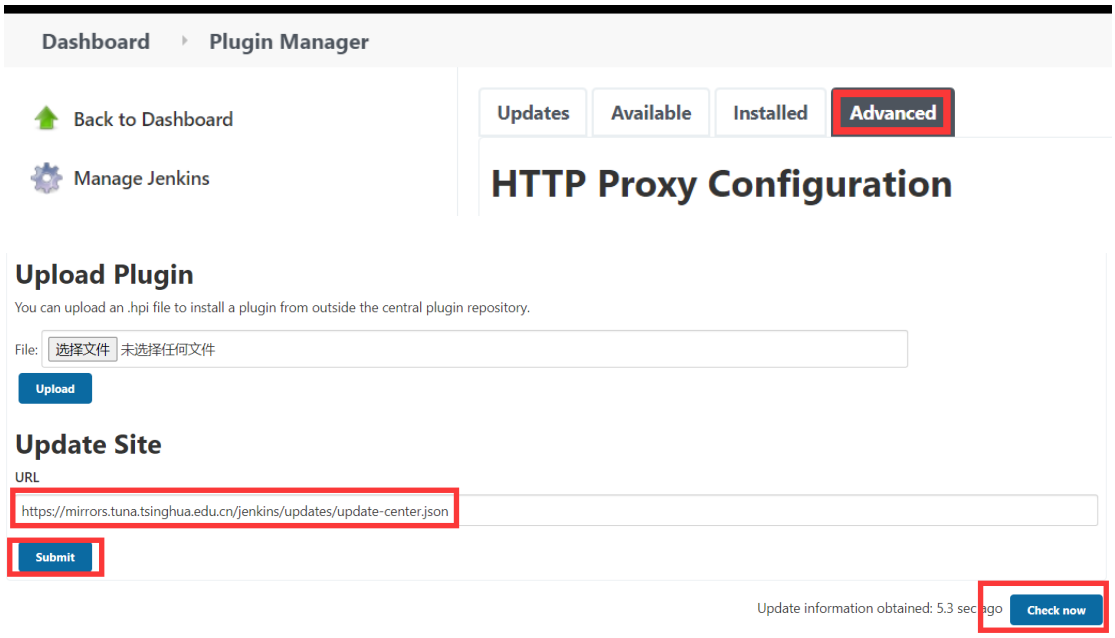
1.4.1.2 插件安装

登录后点击 Manage Jenkins → Manage Plugins 安装需要使用的插件：

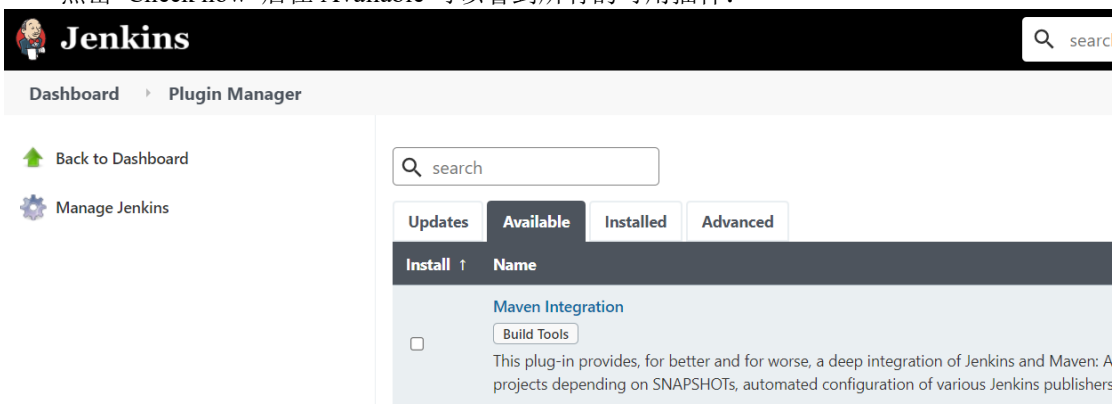


在安装之前首先配置国内的插件源，点击 Advanced，将插件源更改为国内插件源

（https://mirrors.tuna.tsinghua.edu.cn/jenkins/updates/update-center.json）：



点击 Check now 后在 Available 可以看到所有的可用插件：

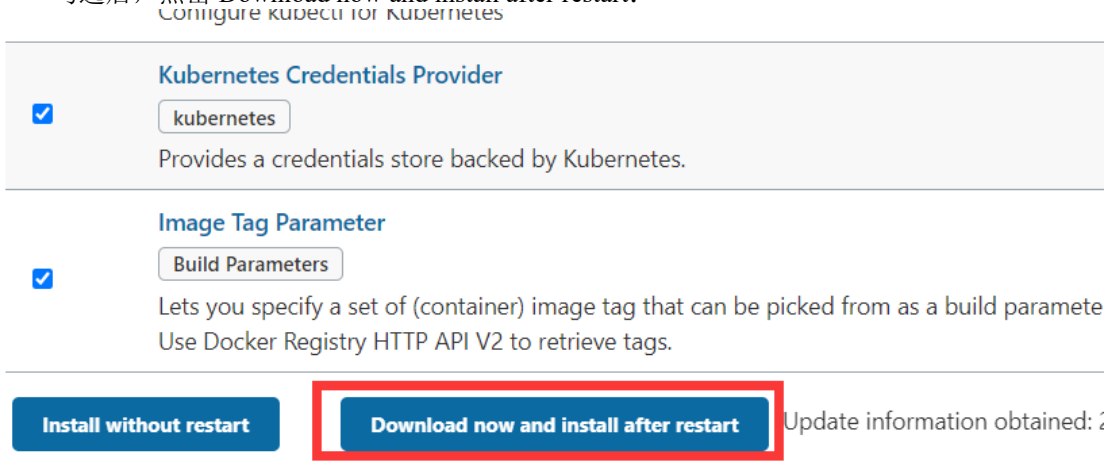


本课程需要的插件如下所示：

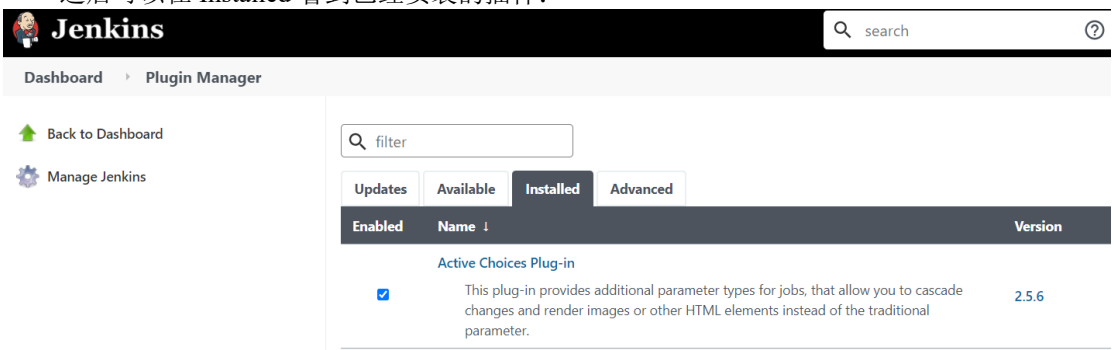
```
Git
Git Parameter
Git Pipeline for Blue Ocean
GitLab
Credentials
Credentials Binding
Blue Ocean
Blue Ocean Pipeline Editor
Blue Ocean Core JS
Pipeline SCM API for Blue Ocean
Dashboard for Blue Ocean
Build With Parameters
Dynamic Extended Choice Parameter Plug-In
Dynamic Parameter Plug-in
Extended Choice Parameter
List Git Branches Parameter
Pipeline
Pipeline: Declarative
Kubernetes
Kubernetes CLI
Kubernetes Credentials
```

```
Image Tag Parameter
Active Choices
```

勾选后，点击 Download now and install after restart：



之后可以在 Installed 看到已经安装的插件：



至此 Jenkins 和 Jenkins 插件的安装就完成了，接下来安装 Harbor 和 GitLab。

1.4.2 GitLab 安装

GitLab 在企业内经常用于代码的版本控制，也是 DevOps 平台中尤为重要的一个工具，接下来在另一台服务器（4C4G40G 以上）上安装 GitLab（如果同样有可用的 GitLab，也可无需安装）。

首先在 GitLab 国内源下载 GitLab 的安装包：https://mirrors.tuna.tsinghua.edu.cn/gitlab-ce/yum/el7/。

将下载后的 rpm 包，上传到服务器，之后通过 yum 直接安装即可：

```
# yum install gitlab-ce-14.2.3-ce.0.el7.x86_64.rpm -y
```

安装完成后，需要更改几处配置：

```
# vim /etc/gitlab/gitlab.rb
```

将 external_url 更改为自己的发布地址，可以是服务器的 IP，也可以是一个可被解析的域名：

```
##! On AWS EC2 instances, we also attempt to fetch the public hostname/IP
##! address from AWS. For more details, see:
##! https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval
external_url 'http://[redacted]'

## Roles for multi-instance GitLab
```

大部分公司内可能已经有了 Prometheus 监控平台，所以 GitLab 自带的 Prometheus 可以无需安装，后期可以安装 GitLab Exporter 即可进行监控。关闭 Prometheus 插件（可选）：

```
# monitoring_role['enable'] = true

prometheus['enable'] = false
# prometheus['monitor_kubernetes'] = true
# prometheus['username'] = 'gitlab-prometheus'
# prometheus['group'] = 'gitlab-prometheus'
```

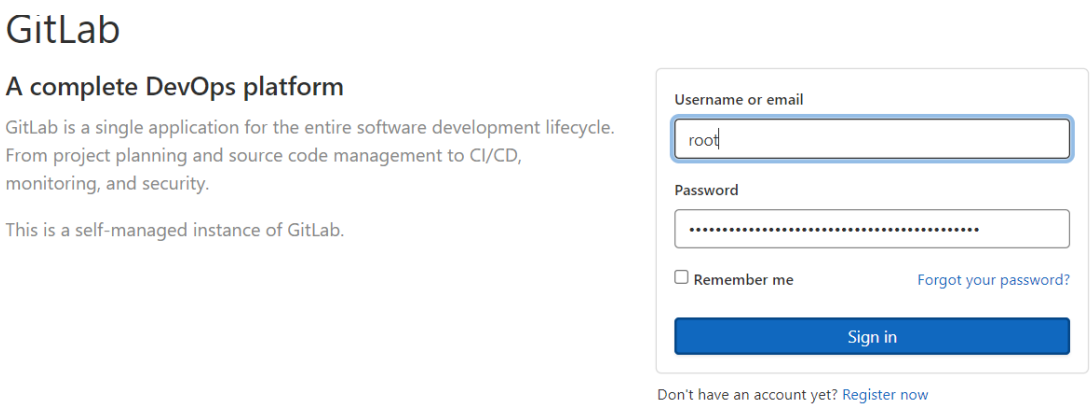
更改完成后需要重新加载配置文件：

```
# gitlab-ctl reconfigure
```

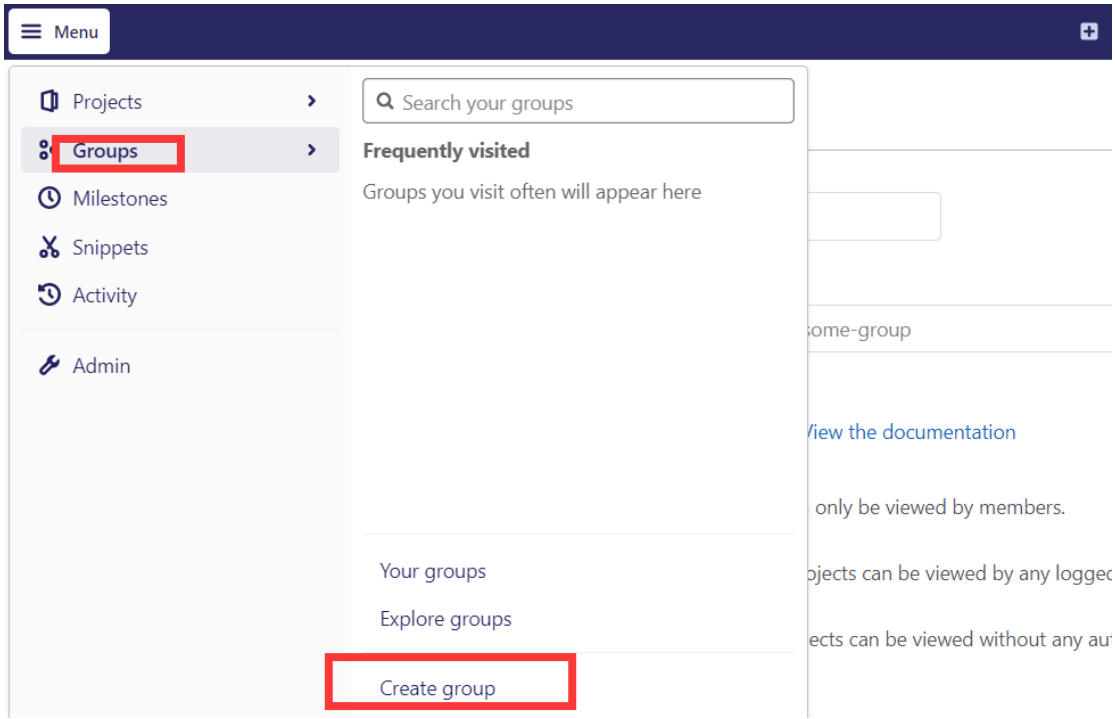
加载配置完成以后，可以看到如下信息：

```
Notes:
Default admin account has been configured with following details:
Username: root
Password: You didn't opt-in to print initial root password to STDOUT.
Password stored to /etc/gitlab/initial_root_password. This file will be
cleaned up in first reconfigure run after 24 hours.

NOTE: Because these credentials might be present in your log files in
plain text, it is highly recommended to reset the password following
https://docs.gitlab.com/ee/security/reset_user_password.html#reset-your-
root-password.

gitlab Reconfigured!
```

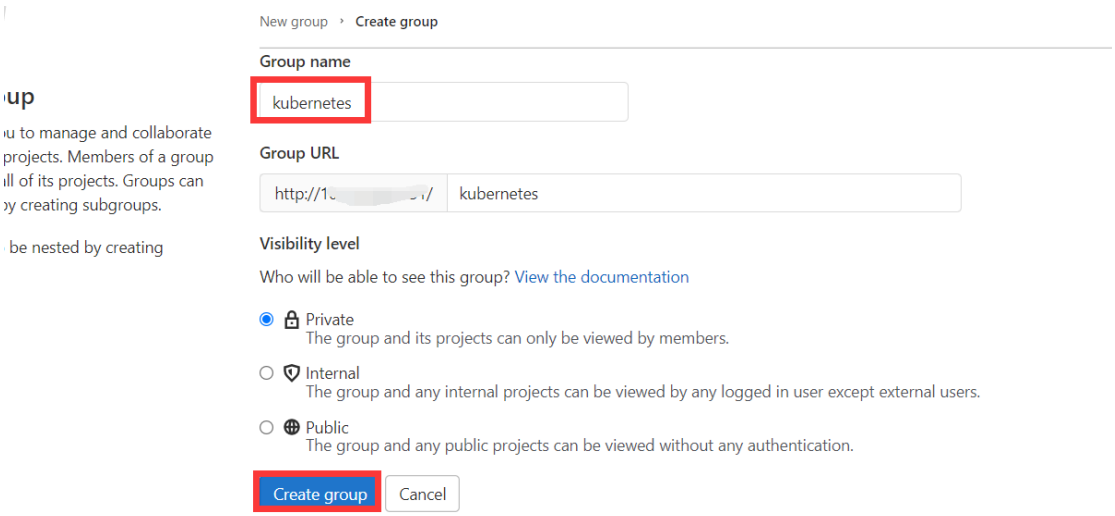
之后可以通过浏览器访问 GitLab，账号 root，默认密码在**/etc/gitlab/initial_root_password**：



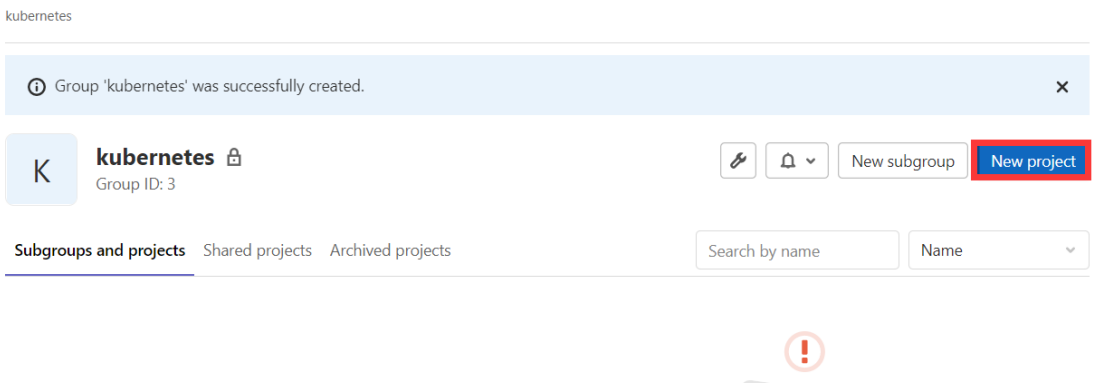
登录后，可以创建一个测试项目，进行一些简单的测试。首先创建一个组：



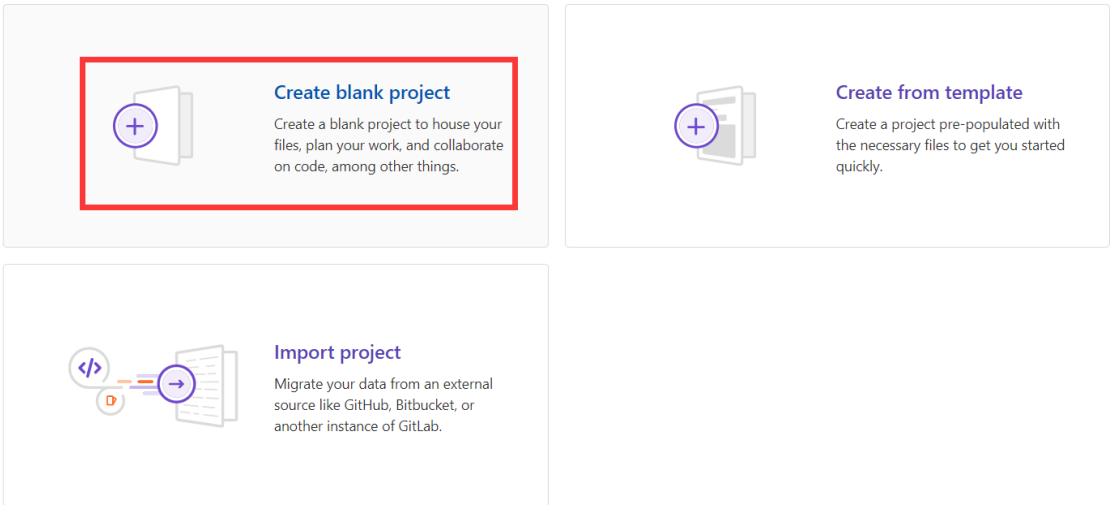
组名为 kubernetes，类型为 Private，之后点击 Create group 即可：



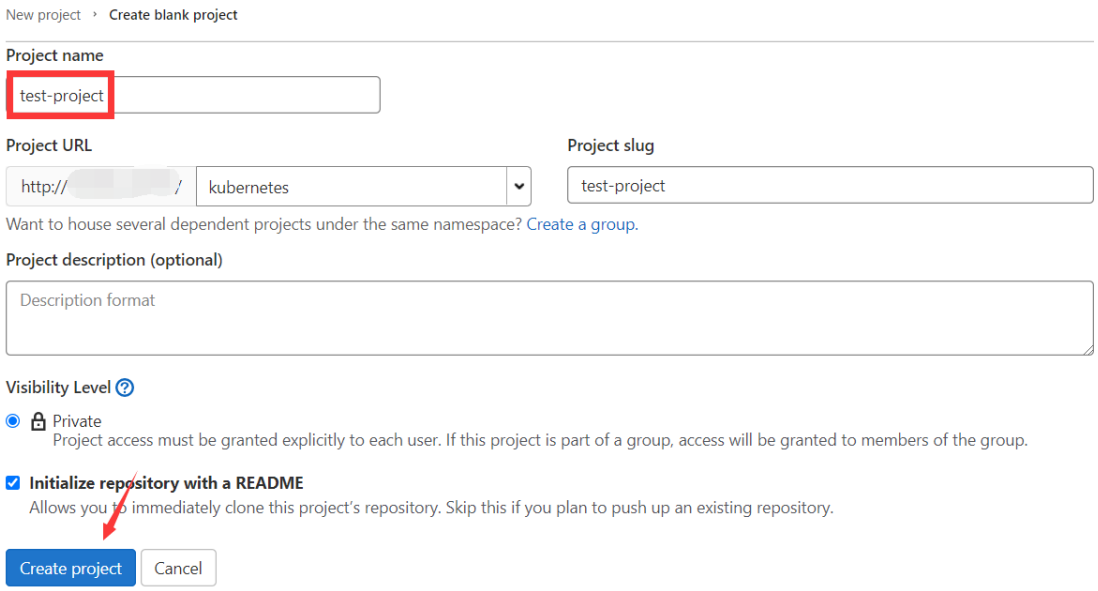
之后在该组下创建一个 Project：



选择创建一个空的项目：



输入项目名称，然后点击 Create project 即可：



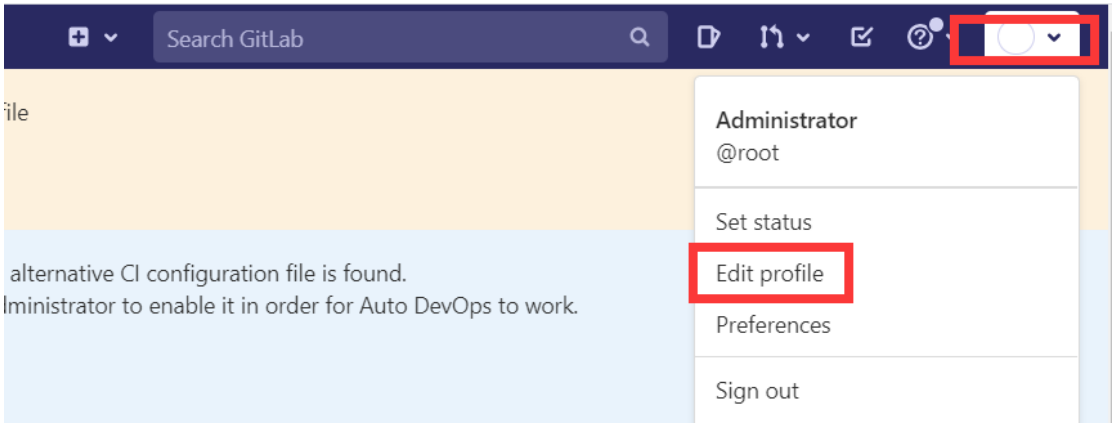
之后可以将 Jenkins 服务器的 key 导入到 GitLab，首先生成密钥（如有可以无需生成）：

```
# ssh-keygen -t rsa -C "YOUR_EMAIL@ADDRESS.COM"
```

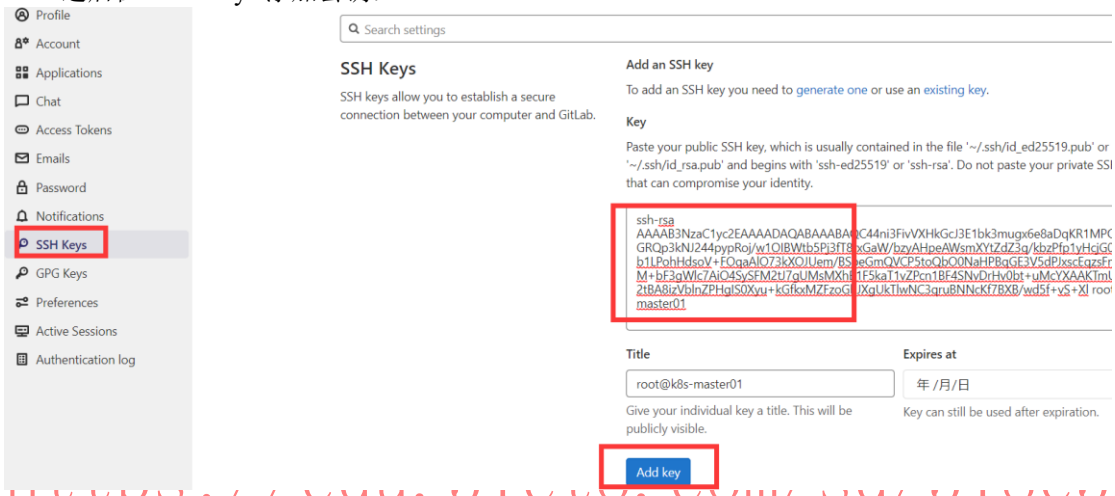
将公钥的内容放在 GitLab 中即可：

```
# cat ~/.ssh/id_rsa.pub
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC44ni3FivVXHkGcJ3E1bk3mugx6e8aDqKR1MPQP0idb0XG
RQp3kNJ244pypRoj/w1OIBWtb5Pj3fT8rxGaW/bzyAHpeAWsmXYtZdZ3q/kbzPfp1yHcjG0RrWs9
Fb1LPohHdsoV+FOqaAlO73kXOJUem/BSbeGmQVCP5toQbO0NaHPBqGE3V5dPJxscEqzsFmlrH/Wh
M+bF3gWlc7AiO4SySFM2tJ7gUMsMXhB1F5kaT1vZPcn1BF4SNvDrHv0bt+uMcYXAAKTmUUdZcpG2
tBA8izVblnZPHgIS0Xyu+kGfkxMZFzoGhJXgUkTlwNC3qruBNNcKf7BXB/wd5f+vS+Xl
root@k8s-master01
```

在 GitLab 找到 Profile：



之后在 SSH Keys 添加公钥：



添加后就可以在 Jenkins 服务器拉取代码：

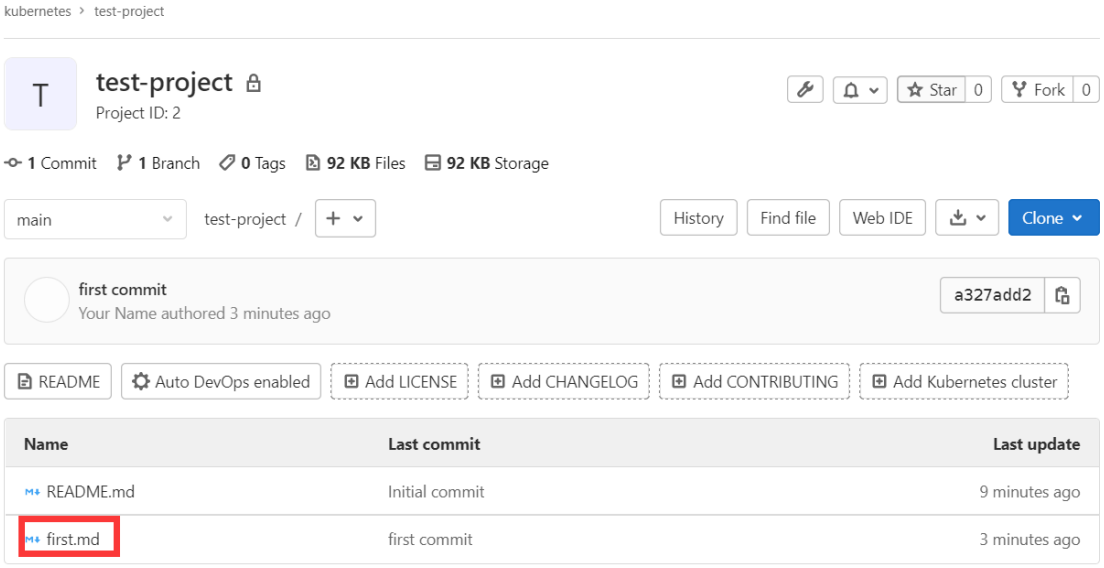
```
    # yum install git -y
    # git clone git@YOUR_GITLAB_ADDRESS:kubernetes/test-project.git
    Cloning into 'test-project'...
    The authenticity of host 'YOUR_GITLAB_ADDRESS (YOUR_GITLAB_ADDRESS)'
can't be established.
    ECDSA key fingerprint is
SHA256:+S7xLeFMNznoIzwOQDMHHj2spo08wxP4+HI6MgZPlP0.
    ECDSA key fingerprint is
MD5:e0:92:c3:aa:4e:dc:26:8e:bc:92:f0:94:b3:fb:26:61.
    Are you sure you want to continue connecting (yes/no)? yes
    Warning: Permanently added 'YOUR_GITLAB_ADDRESS' (ECDSA) to the list of
known hosts.
    remote: Enumerating objects: 3, done.
    remote: Counting objects: 100% (3/3), done.
    remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
    Receiving objects: 100% (3/3), done.
```

创建几个文件，然后提交测试：

```
    # ls
    test-project
    # cd test-project/
    # ls
    README.md
    # echo "# Frist Commit For DevOps" > first.md
    # ls
    first.md  README.md
    # git add .
    # git commit -am "first commit"
    [main a327add] first commit
     1 file changed, 1 insertion(+)
```

```
 create mode 100644 first.md
# git push origin main
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 302 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@YOUR_GITLAB_ADDRESS:kubernetes/test-project.git
   b64564e..a327add  main -> main
```

提交之后在 GitLab 即可看到该文件：



1.4.3 安装 Harbor

首先在 GitHub 下载最新的 Harbor 离线包，并上传至 Harbor 服务器，官方下载地址：https://github.com/goharbor/harbor/releases/

由于 Harbor 是采用 docker-compose 一键部署的，所以 Harbor 服务器也需要安装 Docker：

```
# yum install -y yum-utils device-mapper-persistent-data lvm2
# yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
# sed -i -e '/mirrors.cloud.aliyuncs.com/d' -e '/mirrors.aliyuncs.com/d' /etc/yum.repos.d/CentOS-Base.repo
# yum install docker-ce-19.03.* docker-ce-cli-19.03.* -y
# systemctl daemon-reload && systemctl enable --now docker
```

安装完成后，将下载的 Harbor 离线包解压并载入 Harbor 镜像：

```
# tar xf harbor-offline-installer-v2.3.2.tgz
# cd harbor
# docker load -i harbor.v2.3.2.tar.gz
```

之后安装 Compose：

```
# curl  -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
# chmod +x /usr/local/bin/docker-compose
# docker-compose -v
docker-compose version 1.29.2, build 5becea4c
```

Harbor 默认提供了一个配置文件模板，需要更改如下字段：

```
# cp harbor.yml.tmpl harbor.yml
# vim harbor.yml
```

```
# The IP address or hostname to access admin UI and registry service.
# DO NOT use localhost or 127.0.0.1, because Harbor needs to be accessed by external clients.
hostname: 1[redacted]

# http related config
http:
  # port for http, default is 80. If https enabled, this port will redirect to https port
  port: 80

# https related config
#https:
#  # https port for harbor, default is 443
#  port: 443
#  # The path of cert and key files for nginx
#  certificate: /your/certificate/path
#  private_key: /your/private/key/path

# # Uncomment following will enable tls communication between all harbor components
# internal_tls:
#   # set enabled to true means internal tls is enabled
#   enabled: true
#   # put your cert and key files on dir
#   dir: /etc/harbor/tls/internal
```

- hostname：Harbor 的访问地址，可以是域名或者 IP，生产推荐使用域名，并且带有证书；
- https：域名证书的配置，生产环境需要配置权威证书供 Harbor 使用，否则需要添加 insecure-registry 配置，由于是学习环境，所以本示例未配置证书；
- 账号密码按需修改即可，默认为 admin:Harbor12345。

之后修改 Harbor 的数据目录：

```
  max_idle_conns: 100
  # The maximum number of open connections to the database. If it
  # Note: the default number of connections is 1024 for postgres o
  max_open_conns: 900

#[redacted]
data_volume: /data/harbor

# Harbor Storage settings by default is using /data dir on local f
```

18e5

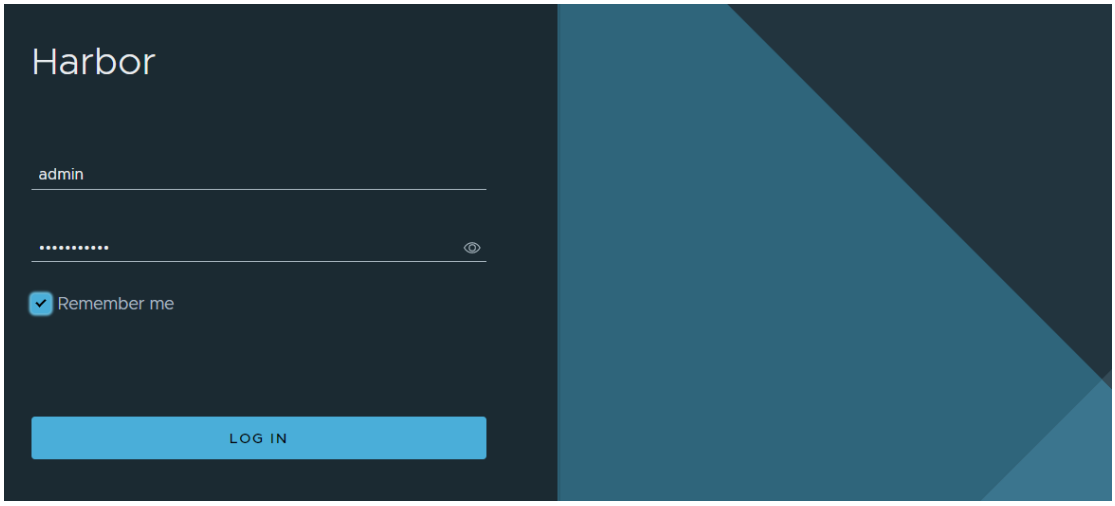
创建 Harbor 数据目录并进行预配置：

```
# mkdir /data/harbor /var/log/harbor -p
# ./prepare
prepare base dir is set to /root/harbor
WARNING:root:WARNING: HTTP protocol is insecure. Harbor will deprecate
http protocol in the future. Please make sure to upgrade to https
...
Generated configuration file: /compose_location/docker-compose.yml
Clean up the input dir
```

执行安装：

```
# ./install.sh
Creating network "harbor_harbor" with the default driver
Creating harbor-log ... done
Creating redis        ... done
Creating harbor-db    ... done
Creating harbor-portal ... done
Creating registryctl   ... done
Creating registry      ... done
Creating harbor-core   ... done
Creating nginx             ... done
Creating harbor-jobservice ... done
✔ ----Harbor has been installed and started successfully.----
```

成功启动后，即可通过配置的地址或域名访问：



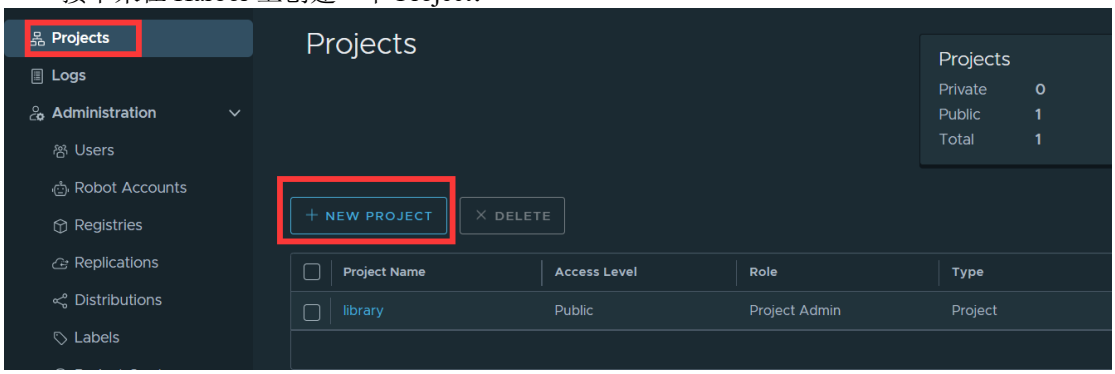
如果配置不是 https 协议，所有的 Kubernetes 节点的 Docker（如果是 containerd 作为 Runtime，可以参考下文配置 insecure-registry）都需要添加 insecure-registries 配置：

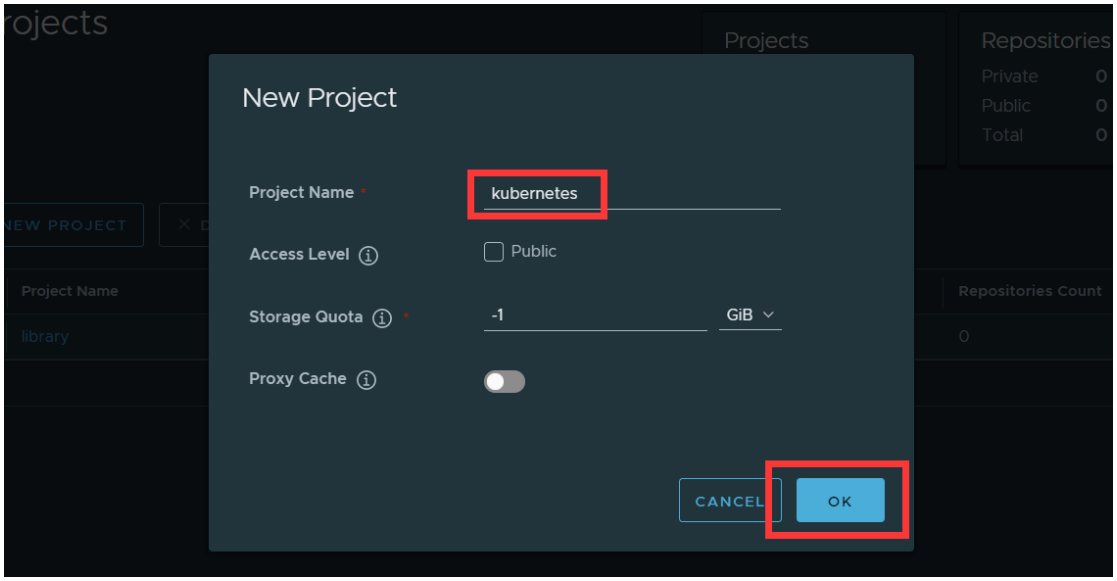
```
# vi /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "insecure-registries": ["YOUR_HARBOR_ADDRESS"]
}
# systemctl daemon-reload
# systemctl restart docker
```

登录测试：

```
# docker login YOUR_HARBOR_ADDRESS
Username: admin
Password:
...
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

接下来在 Harbor 上创建一个 Project：

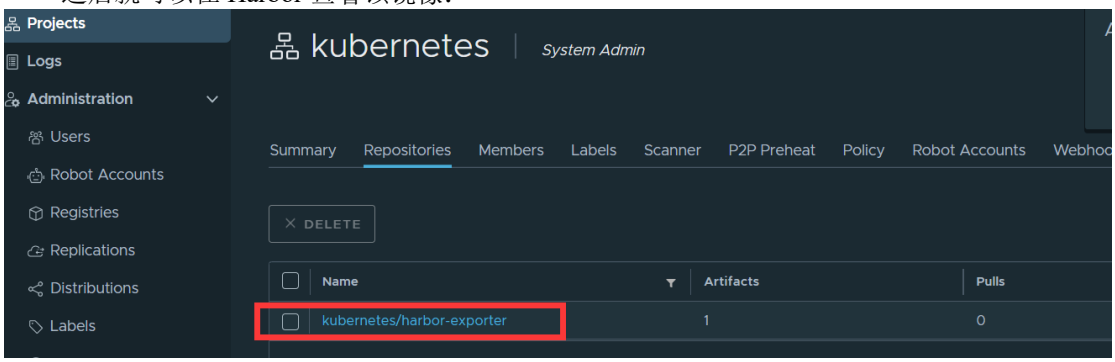




之后在服务器上找任意一个镜像，修改 tag 为 Harbor 的地址，并进行 Push 测试：

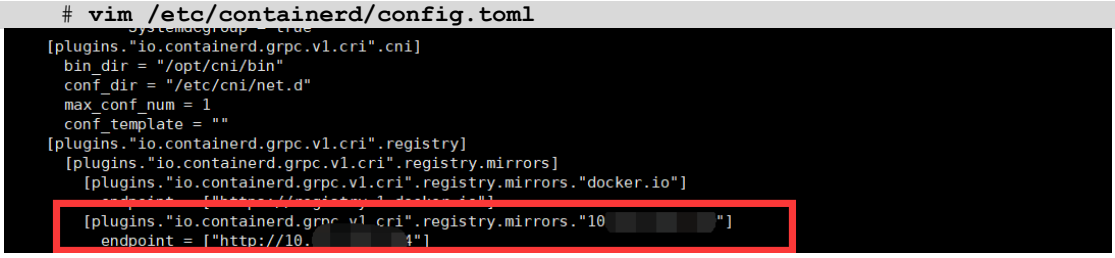
```
    # docker tag goharbor/harbor-exporter:v2.3.2
YOUR_HARBOR_ADDRESS/kubernetes/harbor-exporter:v2.3.2
    # docker push YOUR_HARBOR_ADDRESS/kubernetes/harbor-exporter:v2.3.2
    The push refers to repository [YOUR_HARBOR_ADDRESS/kubernetes/harbor-
exporter]
    7c944a564b1f: Pushed
    3cd80dd3cb0c: Pushed
    8ac2885dd0bc: Pushed
    6b9b715ecb6e: Pushed
    f6e68d4c9b22: Pushed
    v2.3.2: digest:
sha256:a4f8a11ffaa67bcf19424b81ff389c687109598f7e72cf00c49b2cb11b4cc34f
size: 1369
```

之后就可以在 Harbor 查看该镜像：



如果 Kubernetes 集群采用的是 Containerd 作为的 Runtime，配置 insecure-registry 只需要在 Containerd 配置文件的 mirrors 下添加自己的镜像仓库地址即可：

```
    # vim /etc/containerd/config.toml
```



配置完成后，重启 Containerd，之后进行 pull 测试：

```
    # systemctl restart containerd
    # ctr -n k8s.io image pull
CHANGE_HERE_FOR_YOUR_HARBOR_ADDRESS/kubernetes/harbor-exporter:v2.3.2  --
```

```
plain-http --user admin:Harbor12345
```

至此用到的工具都已经安装完成，接下来需要做一些配置。

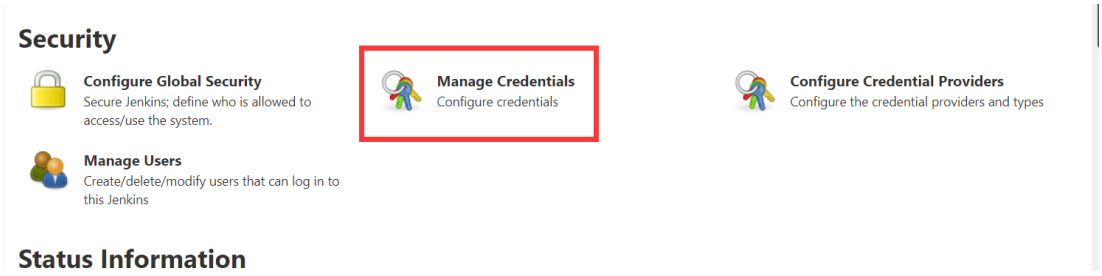
1.4.4 Jenkins 凭证 Credentials

Harbor 的账号密码、GitLab 的私钥、Kubernetes 的证书均使用 Jenkins 的 Credentials 管理。

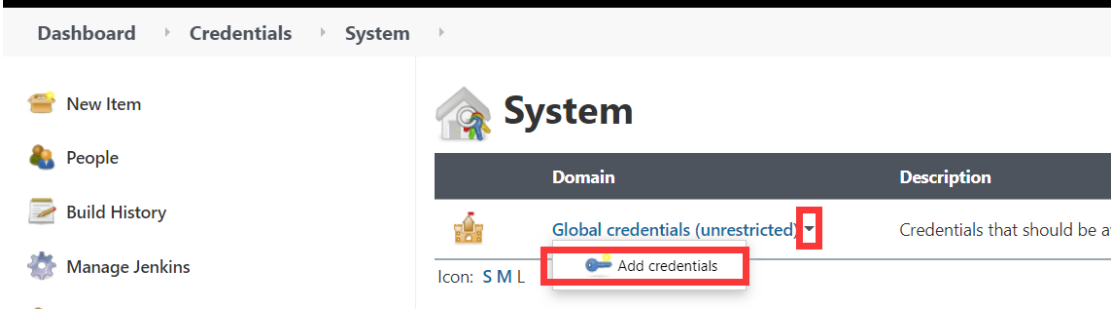
1.4.4.1 配置 Kubernetes 证书

首先需要找到集群中的 KUBECONFIG，一般是 kubectl 节点的~/.kube/config 文件，或者是 KUBECONFIG 环境变量所指向的文件。

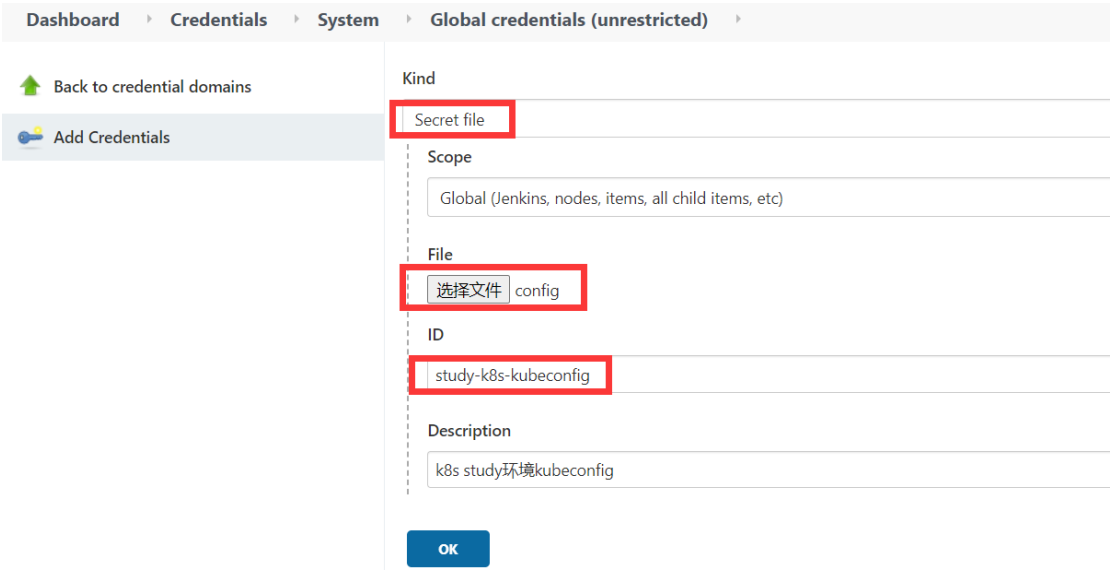
接下来只需要把证书文件放置于 Jenkins 的 Credentials 中即可。首先点击 Manage Jenkins，之后点击 Manage Credentials：



然后在点击 Jenkins，之后点击 Add Credentials：



在打开添加凭证页面时，需要将凭证类型改为 Secret file：

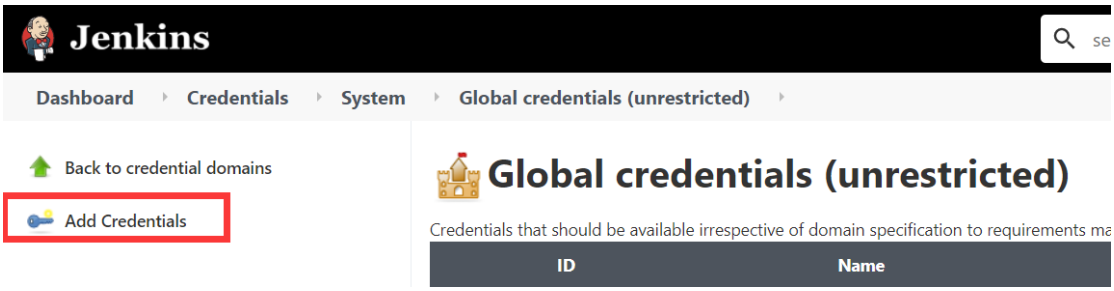


- File：KUBECONFIG 文件或其它加密文件；
- ID：该凭证的 ID；
- Description：证书的描述。

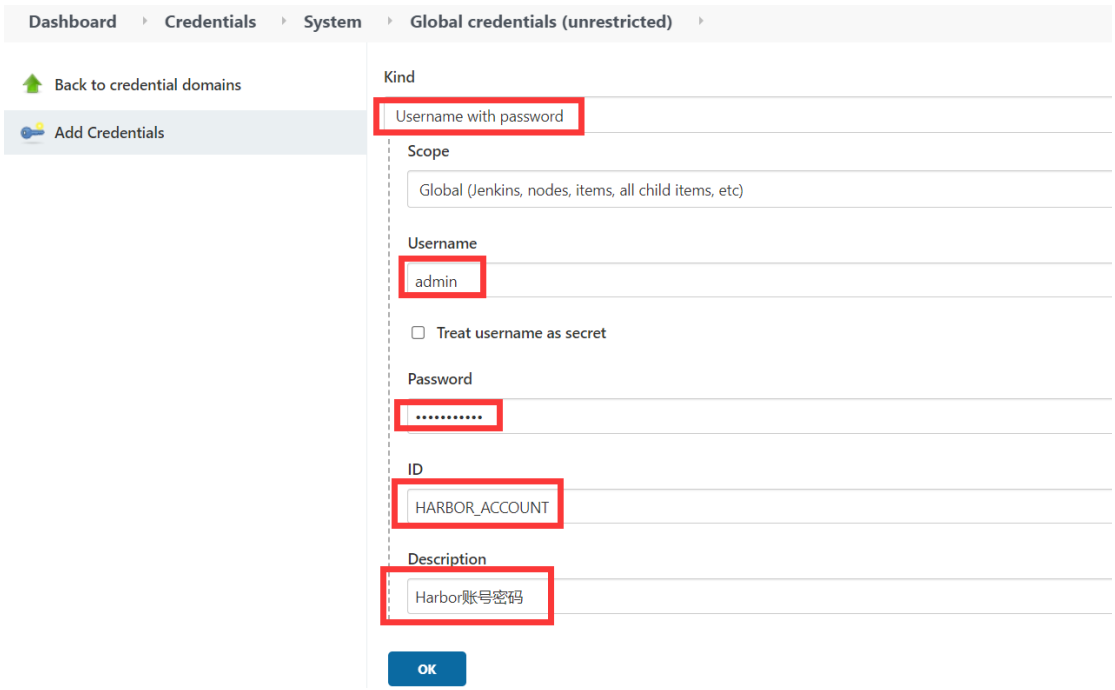
1.4.4.2 配置 Harbor 账号密码

对于账号密码和 Key 类型的凭证，配置步骤是一致的，只是选择的凭证类型不一样。接下来通过 Jenkins 凭证管理 Harbor 的账号密码。

在同样的位置点击 Add Credentials：



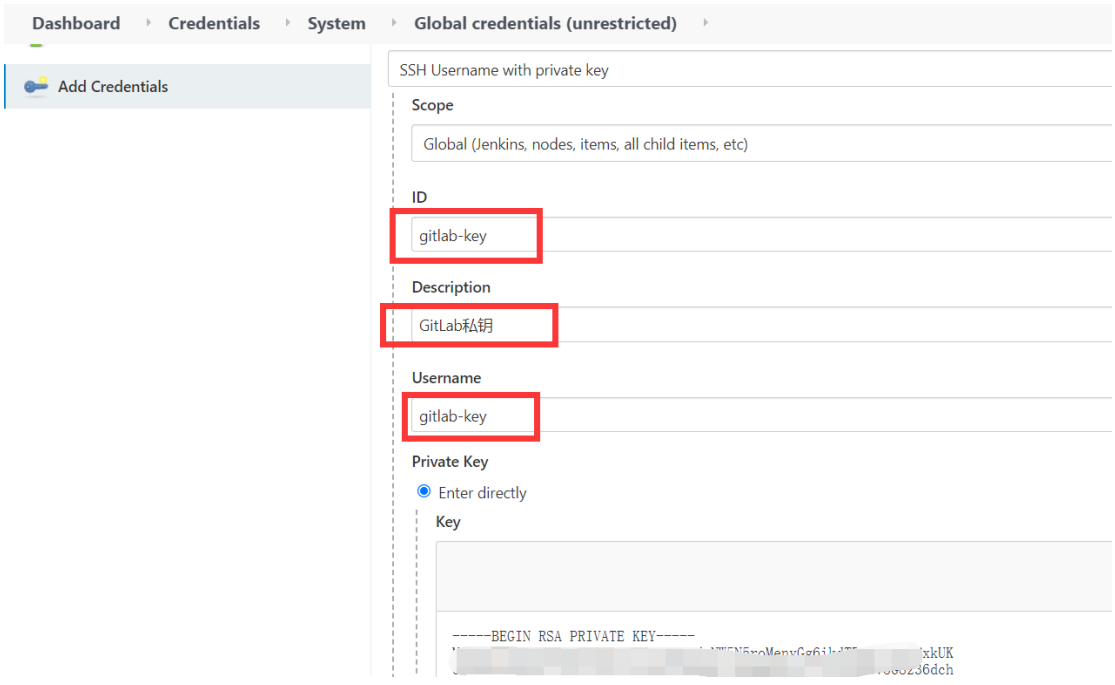
选择类型为 Username with password：



- Username：Harbor 或者其它平台的用户名；
- Password：Harbor 或者其它平台的密码；
- ID：该凭证的 ID；
- Description：证书的描述。

1.4.4.3 配置 GitLab Key

点击 Add Credentials，类型选择为 SSH Username with private key：



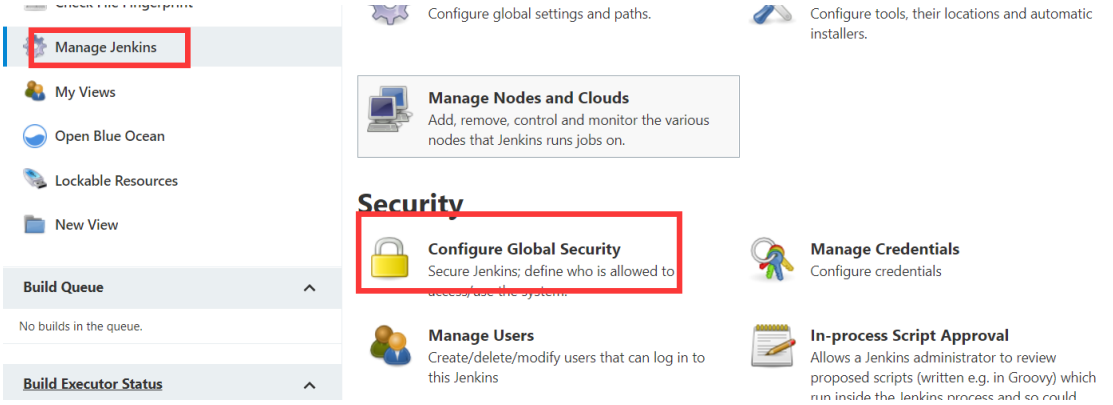
- Username：用户名，无强制性；

- Private Key：Jenkins 服务器的私钥，一般位于~/.ssh/id_rsa。

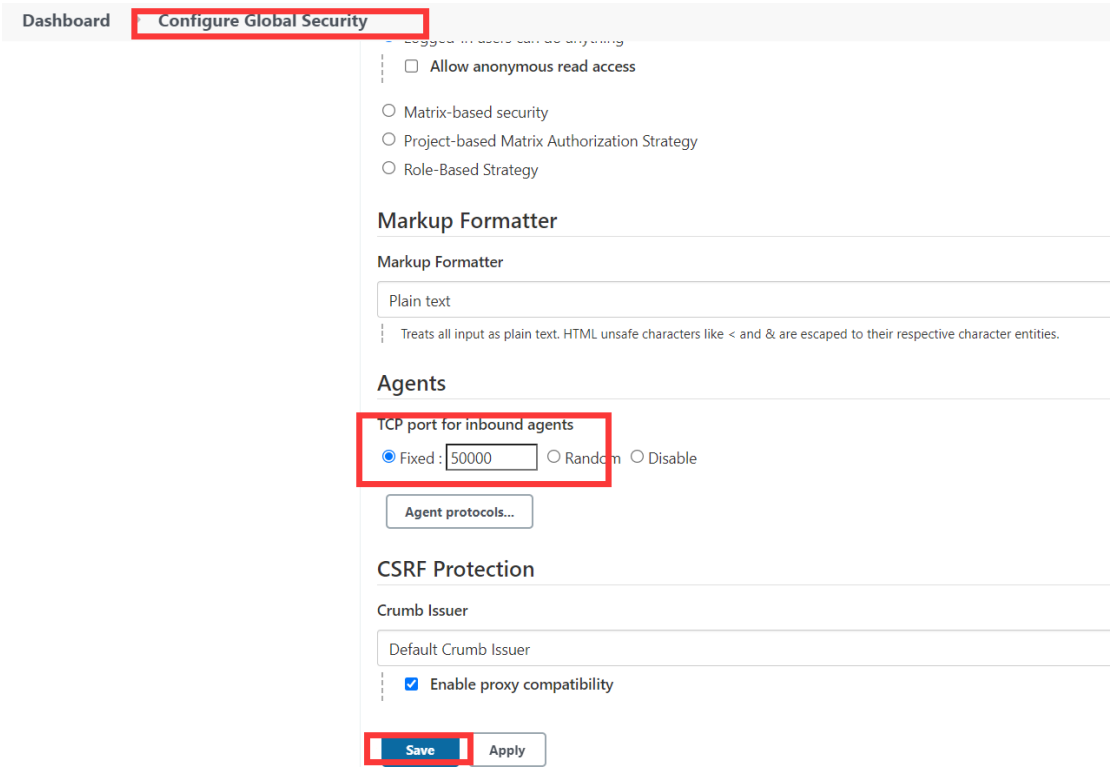
1.4.5 配置 Agent

通常情况下，Jenkins Slave 会通过 Jenkins Master 节点的 50000 端口与之通信，所以需要开启 Agent 的 50000 端口。

点击 Manage Jenkins，然后点击 Configure Global Security：



在安全配置下方找到 Agents，点击 Fixed，输入 50000 即可：



实际使用时，没有必要把整个 Kubernetes 集群的节点都充当创建 Jenkins Slave Pod 的节点，可以选择任意的一个或多个节点作为创建 Slave Pod 的节点。

假设 k8s-node01 作为 Slave 节点：

```
# kubectl label node k8s-node01 build=true
```

如果集群并非使用 Docker 作为 Runtime，但是由于构建镜像时，需要使用 Docker，所以该节点需要安装 Docker：

```
# yum install -y yum-utils device-mapper-persistent-data lvm2
```

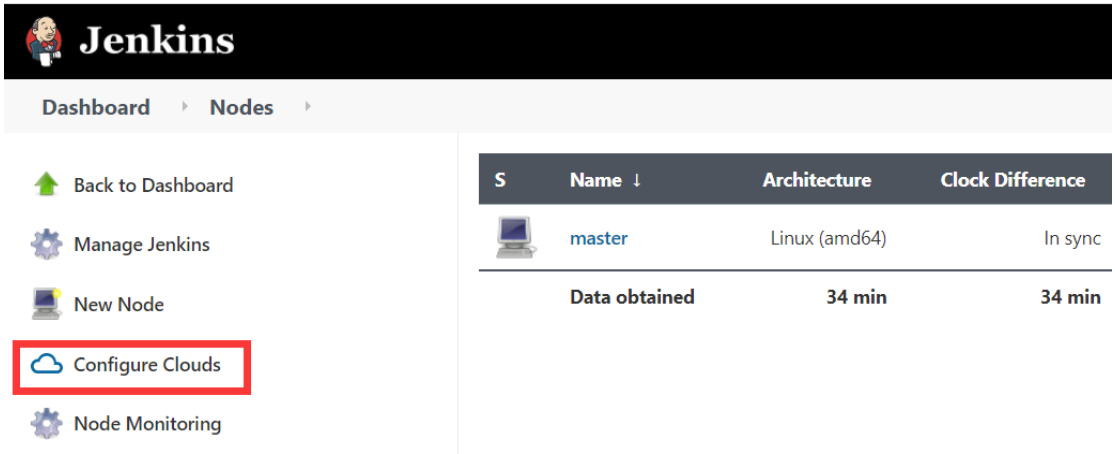
```
    # yum-config-manager --add-repo https://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo
    # sed -i -e '/mirrors.cloud.aliyuncs.com/d' -e '/mirrors.aliyuncs.com/d' /etc/yum.repos.d/CentOS-Base.repo
    # yum install docker-ce-19.03.* docker-ce-cli-19.03.* -y
    # systemctl daemon-reload && systemctl enable --now docker
```

如果镜像仓库未配置证书，需要配置 insecure-registry：

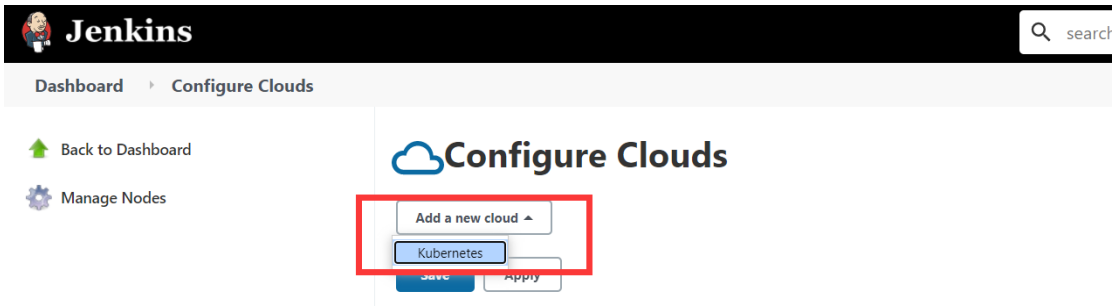
```
    # cat /etc/docker/daemon.json
    {
      "exec-opts": ["native.cgroupdriver=systemd"],
      "insecure-registries": ["CHANGE_HERE_FOR_YOUR_HARBOR_ADDRESS"] # 添加此行
    }
```

1.4.1 Jenkins 配置 Kubernetes 多集群

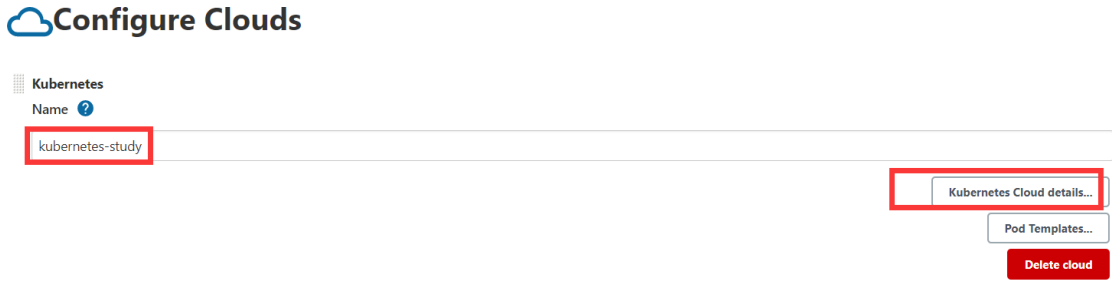
首先点击 Manage Jenkins，之后点击 Configure Clouds：



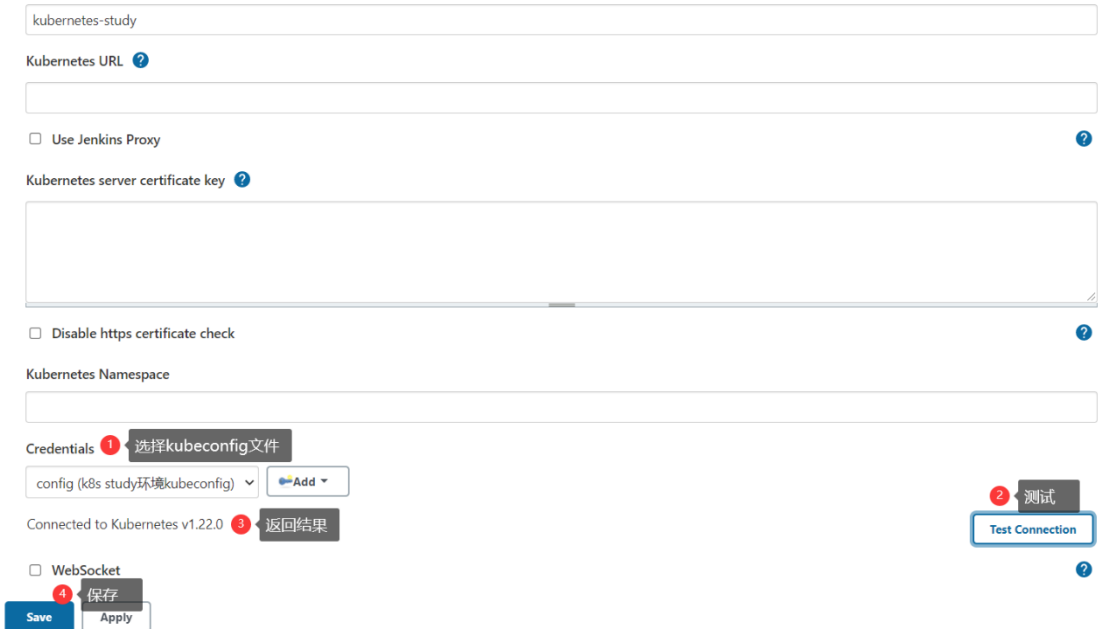
点击 Add a new cloud，选择 Kubernetes：



之后在 Name 字段，输入集群的名称，一般按照可识别的名称即可，比如现在是学习环境的 Kubernetes 可以叫做“kubernetes-study”。之后点击 Kubernetes Cloud details：



然后在 Credentials 处选择之前添加 Kubernetes 证书，选择后点击 Test Connection，最后在凭证下方即可看到能否正常连接的结果：



最后点击 Save 即可，添加完 Kubernetes 后，在 Jenkinsfile 的 Agent 中，就可以选择该集群作为创建 Slave 的集群。

如果想要添加多个集群，重复上述的步骤即可。首先添加 Kubernetes 凭证，然后添加 Cloud 即可。

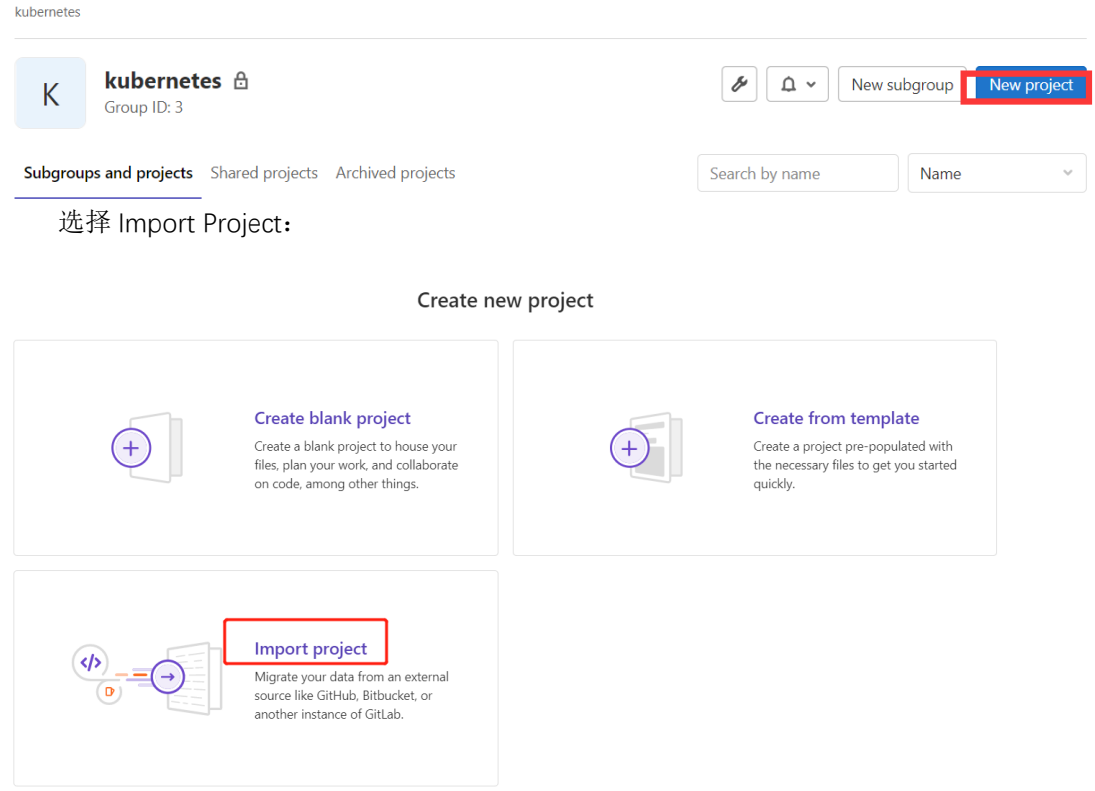
1.5 自动化构建 Java 应用

1.5.1 创建 Java 测试用例

示例项目可以从 https://gitee.com/dukuan/spring-boot-project.git 找到该项目（也可以使用公司的 Java 项目也是一样的）。

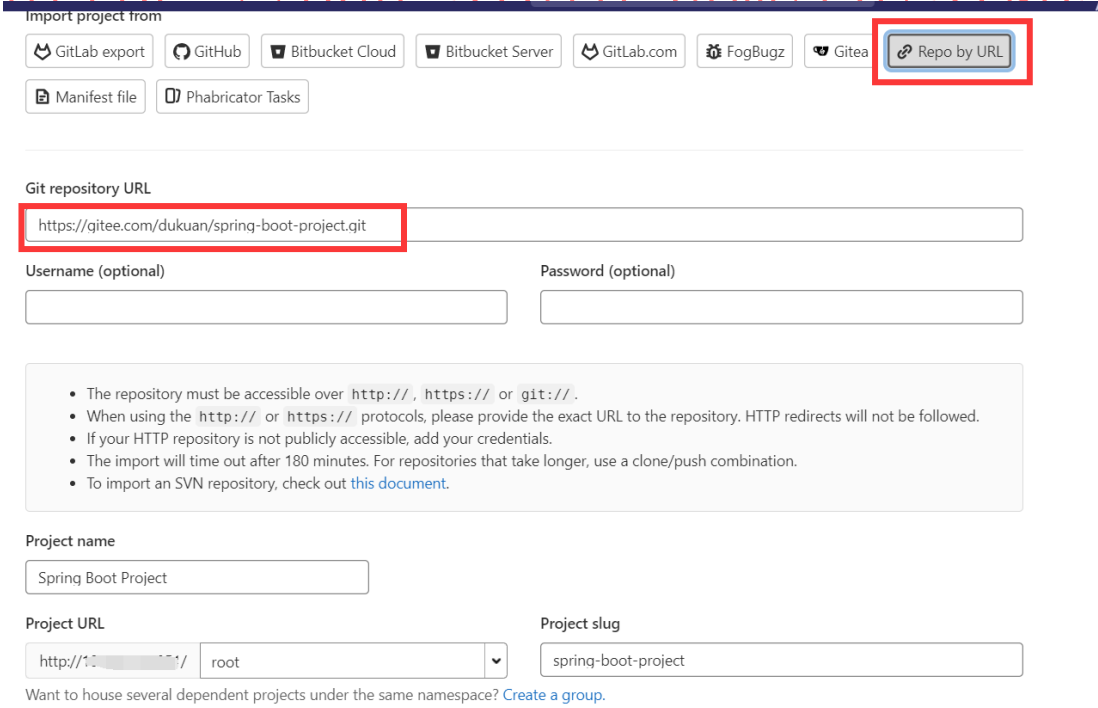
接下来将该项目导入到自己的 GitLab 中。首先找到之前创建的 Kubernetes 组，然后点击

New Project：

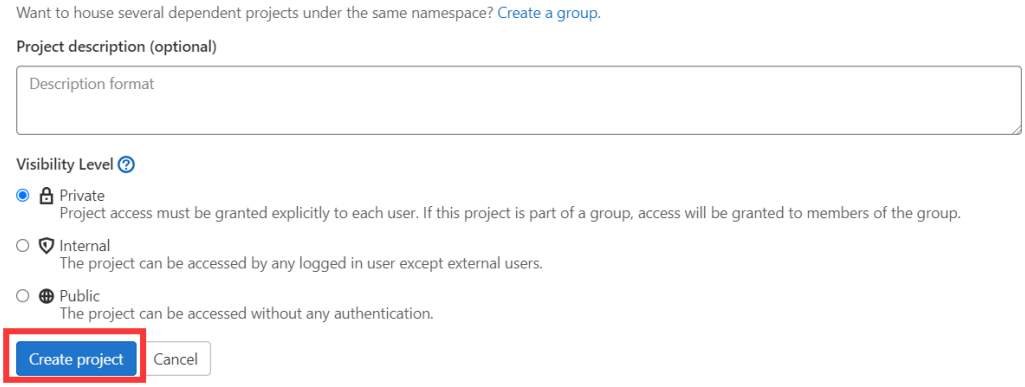


选择 Import Project：

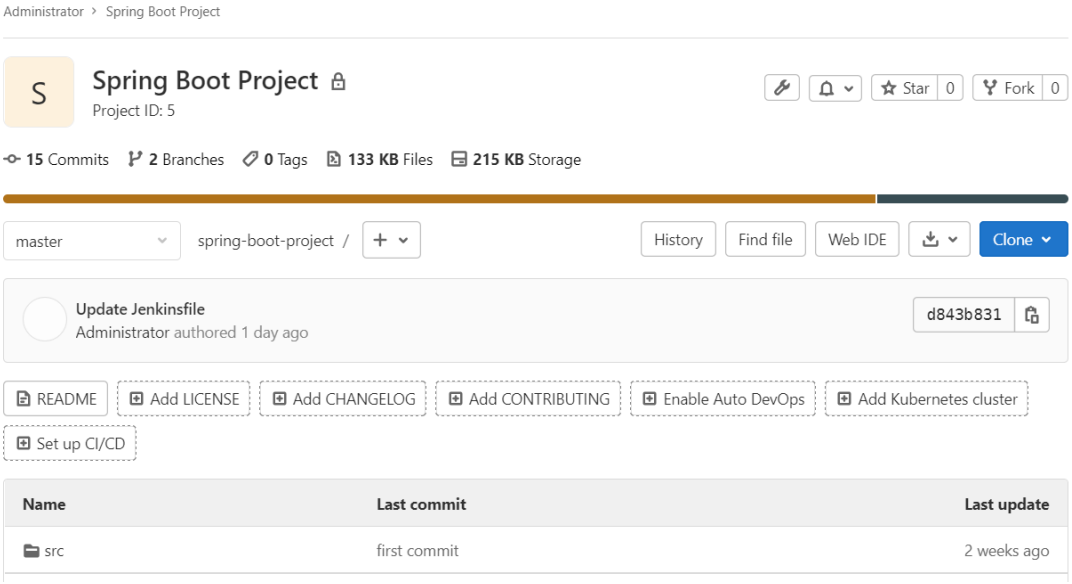
点击 Repo by URL：



在 Git repository URL 输入示例地址，然后点击 Create Project 即可：

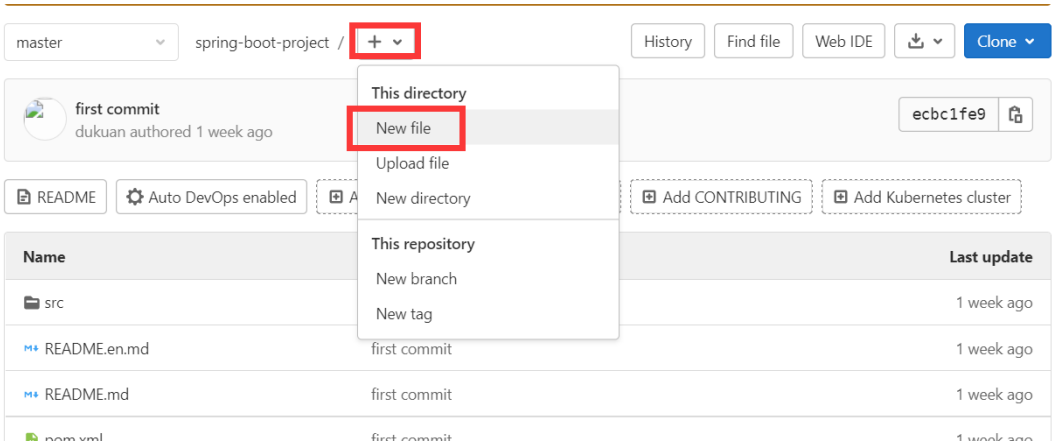


导入后，如下所示：



1.5.2 定义 Jenkinsfile

接下来再 GitLab 的源代码中添加 Jenkinsfile。首先点击代码首页的“+”号，然后点击 New file：



在窗口中，添加如下内容：

```
pipeline {
```

```
    agent {
      kubernetes {
        cloud 'kubernetes-study'
        slaveConnectTimeout 1200
        workspaceVolume hostPathWorkspaceVolume(hostPath: "/opt/workspace", readOnly: false)
        yaml '''
apiVersion: v1
kind: Pod
spec:
  containers:
    - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
      image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
      name: jnlp
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - mountPath: "/etc/localtime"
          name: "localtime"
          readOnly: false
    - command:
        - "cat"
      env:
        - name: "LANGUAGE"
          value: "en_US:en"
        - name: "LC_ALL"
          value: "en_US.UTF-8"
        - name: "LANG"
          value: "en_US.UTF-8"
      image: "registry.cn-beijing.aliyuncs.com/citools/maven:3.5.3"
      imagePullPolicy: "IfNotPresent"
      name: "build"
      tty: true
      volumeMounts:
        - mountPath: "/etc/localtime"
          name: "localtime"
        - mountPath: "/root/.m2/"
          name: "cachedir"
          readOnly: false
    - command:
        - "cat"
      env:
        - name: "LANGUAGE"
          value: "en_US:en"
        - name: "LC_ALL"
          value: "en_US.UTF-8"
```

```
      - name: "LANG"
        value: "en_US.UTF-8"
    image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
    imagePullPolicy: "IfNotPresent"
    name: "kubectl"
    tty: true
    volumeMounts:
      - mountPath: "/etc/localtime"
        name: "localtime"
        readOnly: false
  - command:
      - "cat"
    env:
      - name: "LANGUAGE"
        value: "en_US:en"
      - name: "LC_ALL"
        value: "en_US.UTF-8"
      - name: "LANG"
        value: "en_US.UTF-8"
    image: "registry.cn-beijing.aliyuncs.com/citools/docker:19.03.9-git"
    imagePullPolicy: "IfNotPresent"
    name: "docker"
    tty: true
    volumeMounts:
      - mountPath: "/etc/localtime"
        name: "localtime"
        readOnly: false
      - mountPath: "/var/run/docker.sock"
        name: "dockersock"
        readOnly: false
restartPolicy: "Never"
nodeSelector:
  build: "true"
securityContext: {}
volumes:
  - hostPath:
      path: "/var/run/docker.sock"
    name: "dockersock"
  - hostPath:
      path: "/usr/share/zoneinfo/Asia/Shanghai"
    name: "localtime"
  - name: "cachedir"
    hostPath:
      path: "/opt/m2"
```

```
'''
    }
}
  stages {
    stage('Pulling Code') {
      parallel {
        stage('Pulling Code by Jenkins') {
          when {
            expression {
              env.gitlabBranch == null
            }

          }
          steps {
            git(changelog: true, poll: true, url: 'git@CHANGE_HERE_FOR_YOUR_GITLAB_URL:root/spring-boot-project.git', branch: "${BRANCH}", credentialsId: 'gitlab-key')
            script {
              COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --pretty=format:'%h'").trim()
              TAG = BUILD_TAG + '-' + COMMIT_ID
              println "Current branch is ${BRANCH}, Commit ID is ${COMMIT_ID}, Image TAG is ${TAG}"

            }

          }
        }

        stage('Pulling Code by trigger') {
          when {
            expression {
              env.gitlabBranch != null
            }

          }
          steps {
            git(url: 'git@CHANGE_HERE_FOR_YOUR_GITLAB_URL:root/spring-boot-project.git', branch: env.gitlabBranch, changelog: true, poll: true, credentialsId: 'gitlab-key')
            script {
              COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --pretty=format:'%h'").trim()
              TAG = BUILD_TAG + '-' + COMMIT_ID
              println "Current branch is ${env.gitlabBranch}, Commit ID is ${COMMIT_ID}, Image TAG is ${TAG}"
            }
```

```
            }
          }

        }
      }

      stage('Building') {
        steps {
          container(name: 'build') {
              sh """
                curl repo.maven.apache.org
                mvn clean install -DskipTests
                ls target/*
              """
          }
        }
      }

      stage('Docker build for creating image') {
        environment {
          HARBOR_USER        = credentials('HARBOR_ACCOUNT')
      }
        steps {
          container(name: 'docker') {
            sh """
            echo ${HARBOR_USER_USR} ${HARBOR_USER_PSW} ${TAG}
            docker build -t ${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} .
            docker login -u ${HARBOR_USER_USR} -p ${HARBOR_USER_PSW}
${HARBOR_ADDRESS}
            docker push ${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG}
            """
          }
        }
      }

      stage('Deploying to K8s') {
        environment {
          MY_KUBECONFIG = credentials('study-k8s-kubeconfig')
      }
        steps {
          container(name: 'kubectl'){
            sh """
            /usr/local/bin/kubectl --kubeconfig $MY_KUBECONFIG set image deploy -l
```

```
app=${IMAGE_NAME}
${IMAGE_NAME}=${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG}          -n
$NAMESPACE
            """
          }
        }
      }

  }
  environment {
    COMMIT_ID = ""
    HARBOR_ADDRESS = "CHANGE_HERE_FOR_YOUR_HARBOR_URL"
    REGISTRY_DIR = "kubernetes"
    IMAGE_NAME = "spring-boot-project"
    NAMESPACE = "kubernetes"
    TAG = ""
  }
  parameters {
    gitParameter(branch: '', branchFilter: 'origin/(.*)', defaultValue: '', description: 'Branch for build
and deploy', name: 'BRANCH', quickFilterEnabled: false, selectedValue: 'NONE', sortMode: 'NONE',
tagFilter: '*', type: 'PT_BRANCH')
  }
}
```

1.5.3 Jenkinsfile 详解

首先是顶层的 Agent，定义的是 Kubernetes 的 Pod 作为 Jenkins 的 Slave：

```
    agent {
        # 定义使用 Kubernetes 作为 agent
        kubernetes {
        # 选择的云为之前配置的名字
          cloud 'kubernetes-study'
          slaveConnectTimeout 1200
        # 将 workspace 改成 hostPath，因为该 Slave 会固定节点创建，如果有存储可用，可以
改成 PVC 的模式
          workspaceVolume hostPathWorkspaceVolume(hostPath: "/opt/workspace",
readOnly: false)
          yaml '''
    apiVersion: v1
    kind: Pod
    spec:
      containers:
        # jnlp 容器，和 Jenkins 主节点通信
        - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
          image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
          name: jnlp
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - mountPath: "/etc/localtime"
              name: "localtime"
```

```
            readOnly: false
        # build 容器，包含执行构建的命令，比如 Java 的需要 mvn 构建，就可以用一个 maven 的
镜像
        - command:
            - "cat"
          env:
            - name: "LANGUAGE"
              value: "en_US:en"
            - name: "LC_ALL"
              value: "en_US.UTF-8"
            - name: "LANG"
              value: "en_US.UTF-8"
          # 使用 Maven 镜像，包含 mvn 工具。NodeJS 可以用 node 的镜像
          image: "registry.cn-beijing.aliyuncs.com/citools/maven:3.5.3"
          imagePullPolicy: "IfNotPresent"
          # 容器的名字，流水线的 stage 可以直接使用该名字
          name: "build"
          tty: true
          volumeMounts:
            - mountPath: "/etc/localtime"
              name: "localtime"
             # Pod 单独创建了一个缓存的 volume，将其挂载到了 maven 插件的缓存目录，默认是
/root/.m2
            - mountPath: "/root/.m2/"
              name: "cachedir"
              readOnly: false
          # 发版容器，因为最终是发版至 Kubernetes 的，所以需要有一个 kubectl 命令
        - command:
            - "cat"
          env:
            - name: "LANGUAGE"
              value: "en_US:en"
            - name: "LC_ALL"
              value: "en_US.UTF-8"
            - name: "LANG"
              value: "en_US.UTF-8"
          # 镜像的版本可以替换为其它的版本，也可以不进行替换，因为只执行 set 命令，所以版本
是兼容的
          image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
          imagePullPolicy: "IfNotPresent"
          name: "kubectl"
          tty: true
          volumeMounts:
            - mountPath: "/etc/localtime"
              name: "localtime"
              readOnly: false
          # 用于生成镜像的容器，需要包含 docker 命令
        - command:
            - "cat"
          env:
            - name: "LANGUAGE"
              value: "en_US:en"
            - name: "LC_ALL"
              value: "en_US.UTF-8"
            - name: "LANG"
              value: "en_US.UTF-8"
          image: "registry.cn-beijing.aliyuncs.com/citools/docker:19.03.9-
git"
          imagePullPolicy: "IfNotPresent"
          name: "docker"
          tty: true
          volumeMounts:
```

```
          - mountPath: "/etc/localtime"
            name: "localtime"
            readOnly: false
           # 由于容器没有启动 docker 服务，所以将宿主机的 docker 经常挂载至容器即可
          - mountPath: "/var/run/docker.sock"
            name: "dockersock"
            readOnly: false
    restartPolicy: "Never"
  # 固定节点部署
    nodeSelector:
      build: "true"
    securityContext: {}
    volumes:
      # Docker socket volume
      - hostPath:
          path: "/var/run/docker.sock"
        name: "dockersock"
      - hostPath:
          path: "/usr/share/zoneinfo/Asia/Shanghai"
        name: "localtime"
      # 缓存目录
      - name: "cachedir"
        hostPath:
          path: "/opt/m2"
  '''
    }
```

之后看一下 Jenkinsfile 最后的环境变量和 parameters 的配置：

```
  # 定义一些全局的环境变量
  environment {
    COMMIT_ID = ""
    HARBOR_ADDRESS = "CHANGE_HERE_FOR_YOUR_HARBOR_URL" # Harbor 地址
    REGISTRY_DIR = "kubernetes" # Harbor 的项目目录
    IMAGE_NAME = "spring-boot-project" # 镜像的名称
    NAMESPACE = "kubernetes" # 该应用在 Kubernetes 中的命名空间
    TAG = "" # 镜像的 Tag，在此用 BUILD_TAG+COMMIT_ID 组成
  }
  parameters {
    # 之前讲过一些 choice、input 类型的参数，本次使用的是 GitParameter 插件
    # 该字段会在 Jenkins 页面生成一个选择分支的选项
    gitParameter(branch: '', branchFilter: 'origin/(.*)', defaultValue:
'', description: 'Branch for build and deploy', name: 'BRANCH',
quickFilterEnabled: false, selectedValue: 'NONE', sortMode: 'NONE',
tagFilter: '*', type: 'PT_BRANCH')
  }
```

接下来是拉代码的 stage，这个 stage 是一个并行的 stage，因为考虑了该流水线是手动触发还是触发：

```
  stage('Pulling Code') {
    parallel {
      stage('Pulling Code by Jenkins') {
        when {
          expression {
            # 假如 env.gitlabBranch 为空，则该流水线为手动触发，那么就会执行该
stage，如果不为空则会执行同级的另外一个 stage
            env.gitlabBranch == null
          }

        }
        steps {
          # 这里使用的是 git 插件拉取代码，BRANCH 变量取自于前面介绍的 parameters
配置
```

```
              # git@xxxxxx:root/spring-boot-project.git 代码地址
              # credentialsId: 'gitlab-key'，之前创建的拉取代码的 key
              git(changelog: true, poll: true, url: 'git@xxxxxx:root/spring-
boot-project.git', branch: "${BRANCH}", credentialsId: 'gitlab-key')
              script {
                # 定义一些变量用于生成镜像的 Tag
                # 获取最近一次提交的 Commit ID
                COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --
pretty=format:'%h'").trim()
                # 赋值给 TAG 变量，后面的 docker build 可以取到该 TAG 的值
                TAG = BUILD_TAG + '-' + COMMIT_ID
                println "Current branch is ${BRANCH}, Commit ID is
${COMMIT_ID}, Image TAG is ${TAG}"

              }

            }
          }
          stage('Pulling Code by trigger') {
            when {
              expression {
                # 如果 env.gitlabBranch 不为空，说明该流水线是通过 webhook 触发，则此
时执行该 stage，上述的 stage 不再执行。此时 BRANCH 变量为空
                env.gitlabBranch != null
              }

            }
            steps {
              # 以下配置和上述一致，只是此时 branch: env.gitlabBranch 取的值为
env.gitlabBranch
              git(url: 'git@xxxxxxxxxxx:root/spring-boot-project.git',
branch: env.gitlabBranch, changelog: true, poll: true, credentialsId:
'gitlab-key')
              script {
                COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --
pretty=format:'%h'").trim()
                TAG = BUILD_TAG + '-' + COMMIT_ID
                println "Current branch is ${env.gitlabBranch}, Commit ID is
${COMMIT_ID}, Image TAG is ${TAG}"
              }

            }
          }

        }
      }
```

代码拉下来后，就可以执行构建命令，由于本次实验是 Java 示例，所以需要使用 mvn 命令进行构建：

```
    stage('Building') {
         steps {
           # 使用 Pod 模板里面的 build 容器进行构建
           container(name: 'build') {
              sh """
                # 编译命令， 需要根据自己项目的实际情况进行修改，可能会不一致
                mvn clean install -DskipTests
                # 构建完成后，一般情况下会在 target 目录下生成 jar 包
                ls target/*
              """
           }
         }
       }
```

生成编译产物后，需要根据该产物生成对应的镜像，此时可以使用 Pod 模板的 docker 容器：

```
    stage('Docker build for creating image') {
       # 首先取出来 HARBOR 的账号密码
       environment {
           HARBOR_USER    = credentials('HARBOR_ACCOUNT')
        }
          steps {
            # 指定使用 docker 容器
            container(name: 'docker') {
              sh """
               # 执行 build 命令，Dockerfile 会在下一小节创建，也是放在代码仓库，和
Jenkinsfile 同级
               docker build -t
${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} .
              # 登录 Harbor，HARBOR_USER_USR 和 HARBOR_USER_PSW 由上述 environment 生
成
               docker login -u ${HARBOR_USER_USR} -p ${HARBOR_USER_PSW}
${HARBOR_ADDRESS}
              # 将镜像推送至镜像仓库
               docker push
${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG}
               """
            }
          }
        }
```

最后一步就是将该镜像发版至 Kubernetes 集群中，此时使用的是包含 kubectl 命令的容器：

```
    stage('Deploying to K8s') {
       # 获取连接 Kubernetes 集群证书
       environment {
           MY_KUBECONFIG = credentials('study-k8s-kubeconfig')
        }
          steps {
            # 指定使用 kubectl 容器
            container(name: 'kubectl'){
               sh """
               # 直接 set 更改 Deployment 的镜像即可
               /usr/local/bin/kubectl --kubeconfig $MY_KUBECONFIG set image
deploy -l app=${IMAGE_NAME}
${IMAGE_NAME}=${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} -n
$NAMESPACE
               """
            }
          }
        }
      }
```

注意
本次发版的命令为/usr/local/bin/kubectl --kubeconfig $MY_KUBECONFIG set image deploy -l app=${IMAGE_NAME} ${IMAGE_NAME}=${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} -n $NAMESPACE。由于本示例的 Deployment 名称、Label、容器名称均和项目名称一致，所以就统一了 IMAGE_NAME，如果示例不是按照此规范命名，需要进行更改。

1.5.4 定义 Dockerfile

在执行流水线过程时，需要将代码的编译产物做成镜像。Dockerfile 主要写的是如何生成公司业务的镜像。而本次示例是 Java 项目，只需要把 Jar 包放在有 Jre 环境的镜像中，然后启动该 Jar 包即可：

```
    #  基础镜像可以按需修改，可以更改为公司自有镜像
    FROM registry.cn-beijing.aliyuncs.com/dotbalo/jre:8u211-data
    # jar 包名称改成实际的名称，本示例为 spring-cloud-eureka-0.0.1-SNAPSHOT.jar
    COPY target/spring-cloud-eureka-0.0.1-SNAPSHOT.jar ./
    # 启动 Jar 包
    CMD java -jar spring-cloud-eureka-0.0.1-SNAPSHOT.jar
```

1.5.5 定义 Kubernetes 资源

本示例在 GitLab 创建的 Group 为 kubernetes，可以将其认为是一个项目，同一个项目可以部署至 Kubernetes 集群中同一个 Namespace 中，本示例为 kubernetes 命名空间。由于使用的是私有仓库，因此也需要先配置拉取私有仓库镜像的密钥：

```
    # kubectl create ns kubernetes
    namespace/kubernetes created
    # kubectl create secret docker-registry harborkey --docker-
server=CHANGE_HERE_FOR_YOUR_HARBOR_ADDRESS --docker-username=admin --docker-
password=Harbor12345 --docker-email=mail@kubeasy.com -n kubernetes
    secret/harborkey created
```


配置该应用的 Deployment（部分内容），需要注意文件的加粗部分：

```
    apiVersion: apps/v1
    kind: Deployment
    metadata:
      creationTimestamp: null
      labels:
        app: spring-boot-project # Deployment 标签，和流水线的 set -l 一致
      name: spring-boot-project  # Deployment 名称
      namespace: kubernetes
    spec:
      replicas: 1
      selector:
        matchLabels:
          app: spring-boot-project # Pod 的标签
      template:
        metadata:
          creationTimestamp: null
          labels:
            app: spring-boot-project # Pod 的标签
        spec:
          ...
            - name: LANG
              value: C.UTF-8
            image: nginx # 此处使用的 nginx 作为原始的镜像，通过 Jenkins 构建并发版后，
变成 Java 应用的镜像
            imagePullPolicy: IfNotPresent
            lifecycle: {}
            livenessProbe:
              failureThreshold: 2
              initialDelaySeconds: 30
              periodSeconds: 10
```

```
            successThreshold: 1
            tcpSocket:
              port: 8761 # 端口号和健康检查按照实际情况进行修改
            timeoutSeconds: 2
          name: spring-boot-project # 容器的名称，需要和流水线 set 命令的容器名称一
致
          ports:
          - containerPort: 8761 # 端口号按照实际情况进行修改
            name: web
            protocol: TCP
          readinessProbe:
            failureThreshold: 2
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 8761 # 端口号和健康检查按照实际情况进行修改
            timeoutSeconds: 2
          resources: # 资源请求按照实际情况修改
            limits:
              cpu: 994m
              memory: 1170Mi
            requests:
              cpu: 10m
              memory: 55Mi
        dnsPolicy: ClusterFirst
        imagePullSecrets:
        - name: harborkey # Harbor 仓库密钥，需要和上述创建的 Secret 一致
        restartPolicy: Always
        securityContext: {}
        serviceAccountName: default
```


定义该应用的 Service 和 Ingress：

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: spring-boot-project
  name: spring-boot-project
  namespace: kubernetes
spec:
  ports: # 端口按照实际情况进行修改
  - name: web
    port: 8761
    protocol: TCP
    targetPort: 8761
  selector:
    app: spring-boot-project
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: spring-boot-project
  namespace: kubernetes
spec:
```

```
  rules:
  - host: spring-boot-project.test.com
    http:
      paths:
      - backend:
          service:
            name: spring-boot-project
            port:
              number: 8761
        path: /
        pathType: ImplementationSpecific
```

完整文件：

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: spring-boot-project
  name: spring-boot-project
  namespace: kubernetes
spec:
  ports:
  - name: web
    port: 8761
    protocol: TCP
    targetPort: 8761
  selector:
    app: spring-boot-project
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: spring-boot-project
  namespace: kubernetes
spec:
  rules:
  - host: spring-boot-project.test.com
    http:
      paths:
      - backend:
          service:
            name: spring-boot-project
            port:
              number: 8761
        path: /
        pathType: ImplementationSpecific
status:
  loadBalancer: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: spring-boot-project
  name: spring-boot-project
  namespace: kubernetes
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-boot-project
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: spring-boot-project
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - spring-boot-project
              topologyKey: kubernetes.io/hostname
            weight: 100
      containers:
      - env:
        - name: TZ
          value: Asia/Shanghai
        - name: LANG
          value: C.UTF-8
        image: nginx
        imagePullPolicy: IfNotPresent
        lifecycle: {}
        livenessProbe:
          failureThreshold: 2
          initialDelaySeconds: 30
          periodSeconds: 10
          successThreshold: 1
          tcpSocket:
            port: 8761
          timeoutSeconds: 2
        name: spring-boot-project
        ports:
        - containerPort: 8761
          name: web
          protocol: TCP
        readinessProbe:
          failureThreshold: 2
          initialDelaySeconds: 30
          periodSeconds: 10
          successThreshold: 1
          tcpSocket:
            port: 8761
          timeoutSeconds: 2
        resources:
          limits:
            cpu: 994m
            memory: 1170Mi
          requests:
```

```
            cpu: 10m
            memory: 55Mi
        dnsPolicy: ClusterFirst
        imagePullSecrets:
        - name: harborkey
        restartPolicy: Always
        securityContext: {}
        serviceAccountName: default
```

创建该资源：

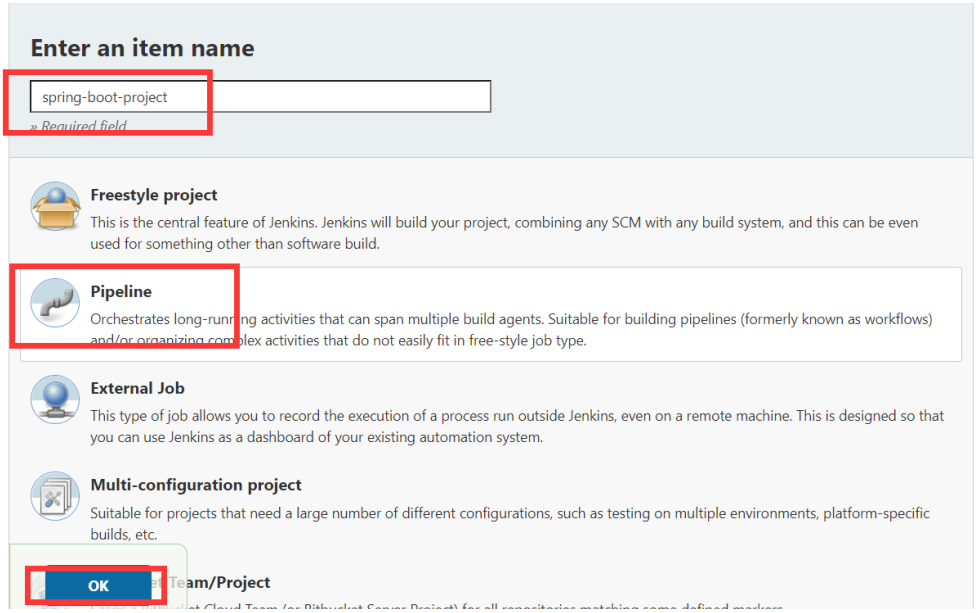
```
# kubectl create -f spring-boot-project.yaml
service/spring-boot-project created
ingress.networking.k8s.io/spring-boot-project created
deployment.apps/spring-boot-project created
```

提示
由于在 Harbor 安装小节已经配置了 kubernetes 的项目目录，在此不再重复创建，如果需要将镜像推送至其它目录，按照同样的步骤创建即可。

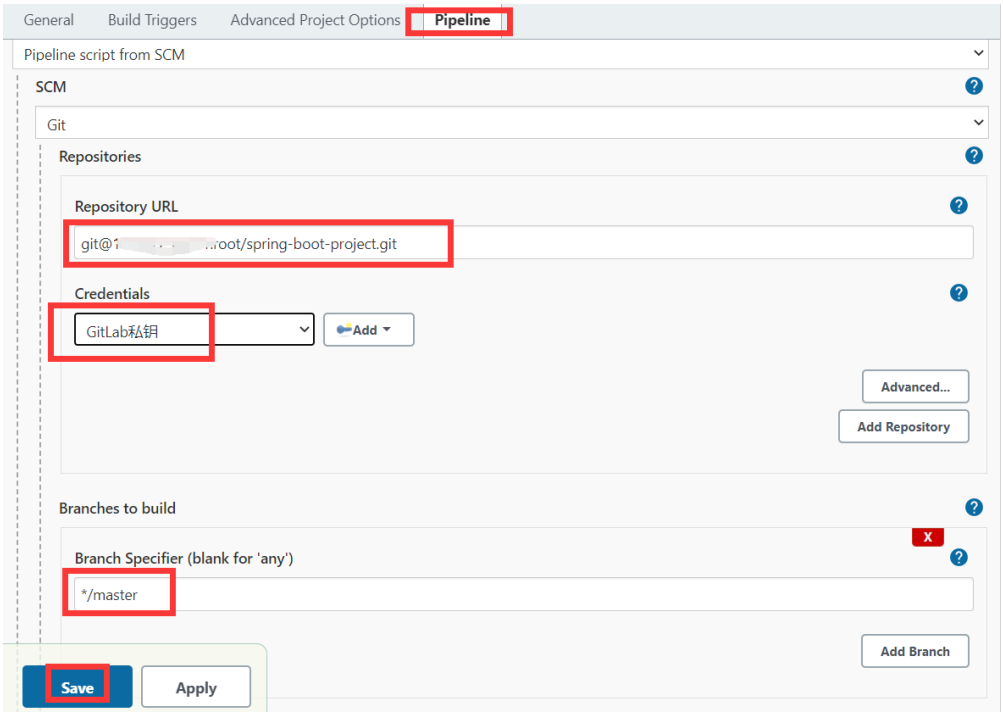
1.5.6 创建 Jenkins 任务（Job）

单击首页的创建任务（Job）选项（或者是 New Item）并配置任务信息，如图所示：

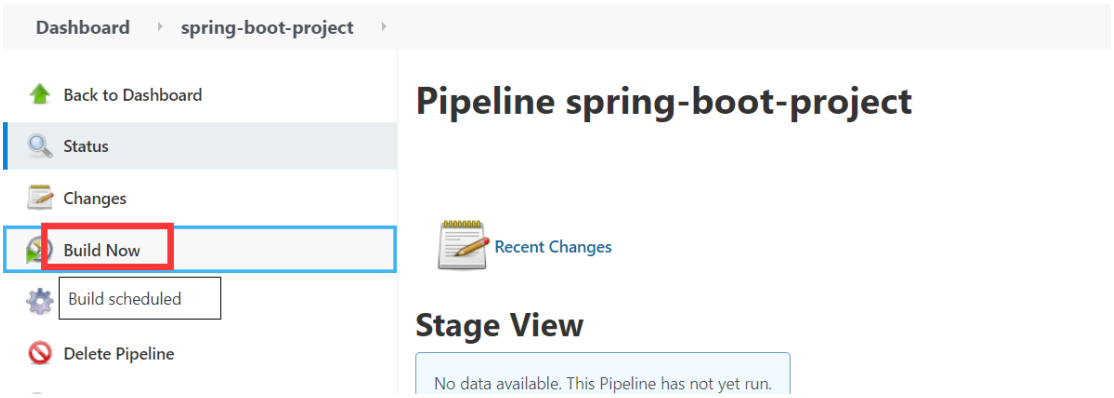
输入的 Job 的名称（一般和 GitLab 的仓库名字一致，便于区分），类型为 Pipeline，最后点击 OK 即可：



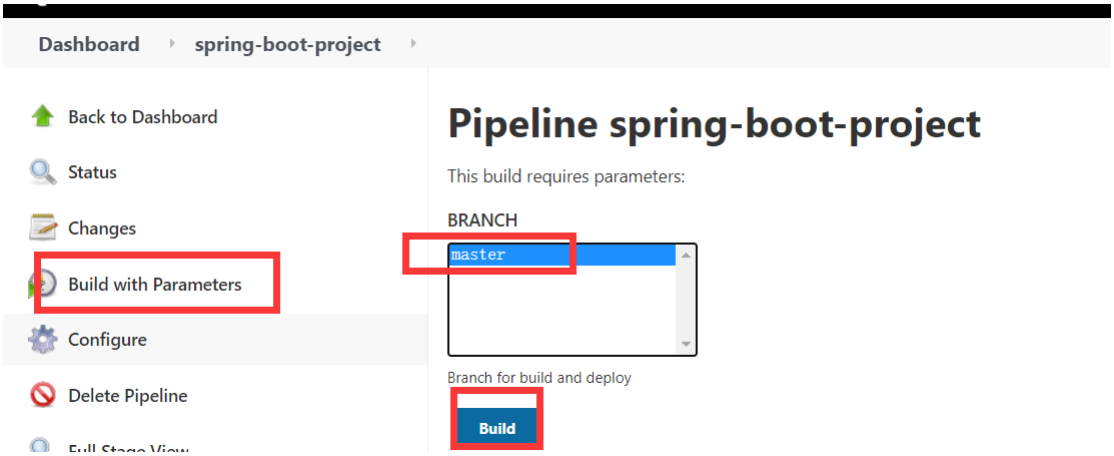
在新页面，点击 Pipeline，输入 Git 仓库地址、选择之前配置 GitLab Key、分支为 master，点击 Save 即可：



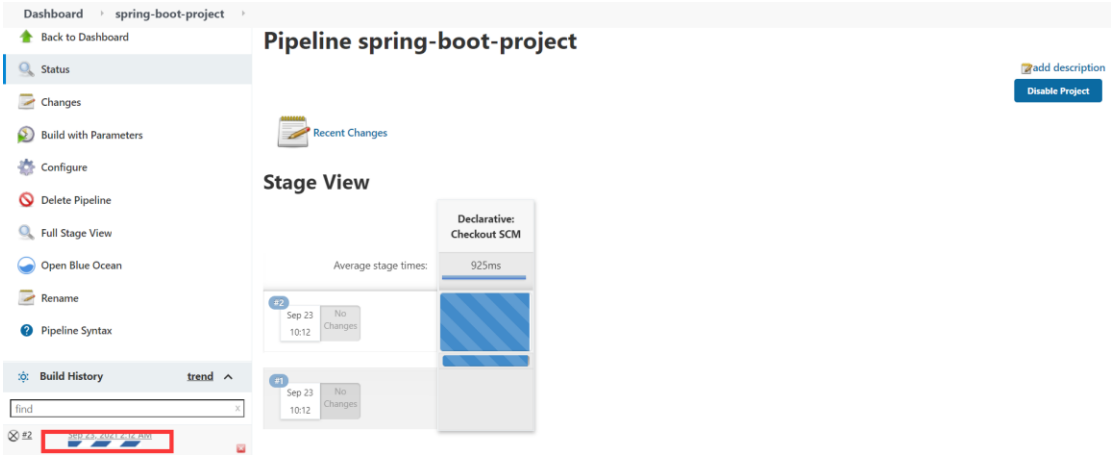
创建完成后，点击 Build Now（由于 Jenkins 参数由 Jenkinsfile 生成，所以第一次执行流水线会失败）：



第一次构建结束后，可以看到 Build Now 变成了 Build with Parameters。点击 Build with Parameters 后，可以读取到 Git 仓库的分支，之后可以选择分支进行手动构建（后面的章节会介绍自动触发）：



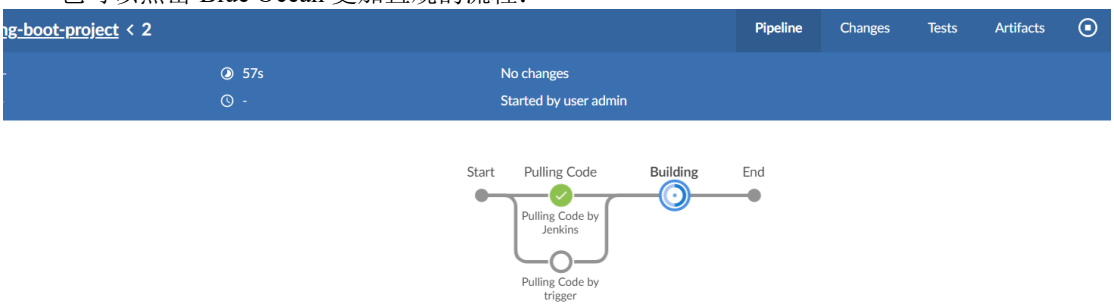
选择分支，之后点击 Build，然后点击 Build History 的进度条即可看到构建日志：



构建日志上部分为创建 Pod 的日志，可以看到 Pod 为 Agent 指定的 Pod：

```
Running in Durability level: MAX_SURVIVABILITY
[Pipeline] Start of Pipeline
[Pipeline] podTemplate
[Pipeline] {
[Pipeline] node
Created Pod: kubernetes-study default/spring-boot-project-2-w5nlp-h3j06-r01w3
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Scheduled] Successfully assigned default/spring-boot-project-2-w5nlp-h3j06-r01w3 to k8s-node01
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Pulled] Container image "registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine" already present on machine
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Created] Created container jnlp
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Started] Started container jnlp
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Pulled] Container image "registry.cn-beijing.aliyuncs.com/citools/maven:3.5.3" already present on machine
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Created] Created container build
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Started] Started container build
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Pulled] Container image "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17" already present on machine
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Created] Created container kubectl
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Started] Started container kubectl
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Pulled] Container image "registry.cn-beijing.aliyuncs.com/citools/docker:19.03.9-git" already present on machine
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Created] Created container docker
[Normal][default/spring-boot-project-2-w5nlp-h3j06-r01w3][Started] Started container docker
Agent spring-boot-project-2-w5nlp-h3j06-r01w3 is provisioned from template spring-boot-project_2-w5nlp-h3j06
---
apiVersion: "v1"
kind: "Pod"
metadata:
  annotations:
```


也可以点击 Blue Ocean 更加直观的流程：



如果是 Java 应用，可以看到 BUILD SUCCESS 说明 mvn 编译已经通过：

```
[INFO] Installing /home/jenkins/agent/workspace/spring-boot-project/pom.xml to /root/.m2/repository/com/testjava/spring-cloud-eureka/0.0.1-SNAPSHOT/sp
cloud-eureka-0.0.1-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3.407 s
[INFO] Finished at: 2021-09-23T02:54:18Z
[INFO] ------------------------------------------------------------------------
+ ls target/classes target/generated-sources target/generated-test-sources target/maven-archiver target/maven-status target/spring-cloud-eureka-0.0.1-
SNAPSHOT.jar target/spring-cloud-eureka-0.0.1-SNAPSHOT.jar.original target/test-classes
target/spring-cloud-eureka-0.0.1-SNAPSHOT.jar
target/spring-cloud-eureka-0.0.1-SNAPSHOT.jar.original
```

编译结束后，可以看到制作镜像的日志：

```
+ docker build -t [redacted]/kubernetes/spring-boot-project:jenkins-spring-boot-project-4-959387c .
Sending build context to Docker daemon  47.47MB

Step 1/3 : FROM registry.cn-beijing.aliyuncs.com/dotbalo/jre:8u211-data
 ---> b1c16eb1456d
Step 2/3 : COPY target/spring-cloud-eureka-0.0.1-SNAPSHOT.jar ./
 ---> 98746b5b8bb2
Step 3/3 : CMD java -jar spring-cloud-eureka-0.0.1-SNAPSHOT.jar
 ---> Running in 2e3063d56e79
Removing intermediate container 2e3063d56e79
 ---> a7a6404e5a2f
Successfully built a7a6404e5a2f
Successfully tagged [redacted]/kubernetes/spring-boot-project:jenkins-spring-boot-project-4-959387c
+ docker login -u admin -p **** [redacted]
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store
```

之后镜像被推送至 Harbor：

```
+ docker push [redacted]/kubernetes/spring-boot-project:jenkins-spring-boot-project-4-959387c
The push refers to repository [[redacted]/kubernetes/spring-boot-project]
a1ba4632765d: Preparing
eff47b948517: Preparing
f25edfe5114d: Preparing
46f3c01a700a: Preparing
37cd0af63488: Preparing
ed211f7d10e3: Preparing
f1b5933fe4b5: Preparing
ed211f7d10e3: Waiting
37cd0af63488: Layer already exists
eff47b948517: Layer already exists
46f3c01a700a: Layer already exists
f25edfe5114d: Layer already exists
ed211f7d10e3: Layer already exists
f1b5933fe4b5: Layer already exists
a1ba4632765d: Pushed
jenkins-spring-boot-project-4-959387c: digest: sha256:aaa7704f3888d54cd3983eac3c337aa393773d08cf43b98237767c3c5713852e size: 1785
```

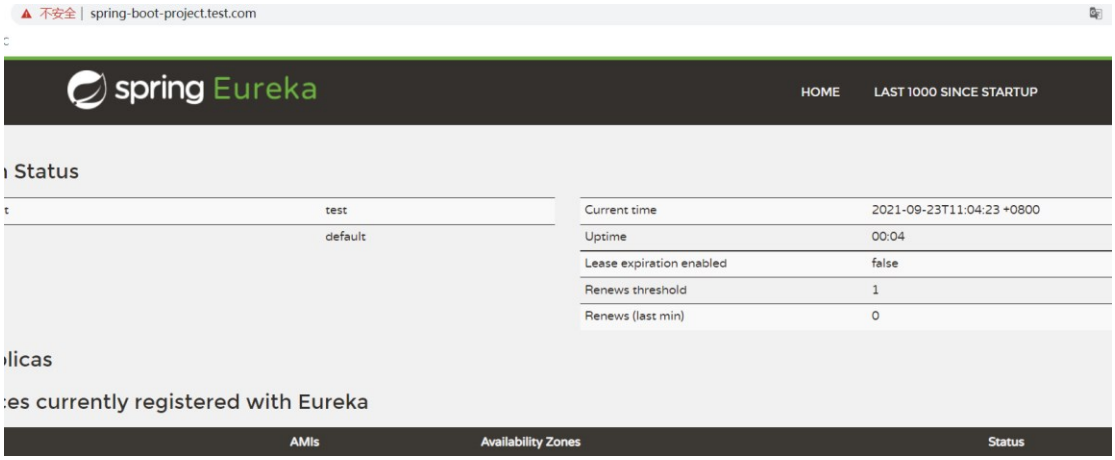
最后为发版至 Kubernetes：


```
Warning: A secret was passed to "sh" using Groovy String interpolation, which is insecure.
		 Affected argument(s) used the following variable(s): [MY_KUBECONFIG]
		 See https://jenkins.io/redirect/groovy-string-interpolation for details.
+ /usr/local/bin/kubectl --kubeconfig **** set image deploy -l 'app=spring-boot-project' 'spring-boot-project=[redacted]/kubernetes/spring-boot-project:jenkins-spring-boot-project-4-959387c' -n kubernetes
deployment.apps/spring-boot-project image updated
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Finished：SUCCESS 说明该流水线正常结束。接下来可以查看 Deployment 的镜像：

```
    # kubectl get deploy -n kubernetes spring-boot-project -oyaml | grep "image:"
          image: xxxxxxxxxxxx.com/kubernetes/spring-boot-project:jenkins-spring-boot-project-4-959387c
    # kubectl get po -n kubernetes
    NAME                                  READY   STATUS    RESTARTS   AGE
    spring-boot-project-869589667d-2zqjm   1/1     Running   0          6m53s
```

如果配置了域名，可以通过域名访问（测试域名需要配置 hosts）：



1.6 自动化构建 Vue/H5 前端应用

本节介绍自动化构建 Vue/H5 应用，其构建方式和自动化构建 Java 基本相同，重点是更改 Deployment、Jenkinsfile 和 Dockerfile 即可。

前端应用测试项目地址：https://gitee.com/dukuan/vue-project.git，可以参考 Java 小节的方式，导入前端项目到 GitLab 中，当然也可以使用公司自己的项目。

1.6.1 定义 Jenkinsfile

完整文件：

```
pipeline {
  agent {
    kubernetes {
      cloud 'kubernetes-study'
      slaveConnectTimeout 1200
      workspaceVolume hostPathWorkspaceVolume(hostPath: "/opt/workspace", readOnly: false)
      yaml ""
apiVersion: v1
kind: Pod
spec:
  containers:
    - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
      image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
      name: jnlp
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - mountPath: "/etc/localtime"
          name: "localtime"
          readOnly: false
    - command:
        - "cat"
```

```
    env:
      - name: "LANGUAGE"
        value: "en_US:en"
      - name: "LC_ALL"
        value: "en_US.UTF-8"
      - name: "LANG"
        value: "en_US.UTF-8"
    image: "registry.cn-beijing.aliyuncs.com/citools/node:lts"
    imagePullPolicy: "IfNotPresent"
    name: "build"
    tty: true
    volumeMounts:
      - mountPath: "/etc/localtime"
        name: "localtime"
      - mountPath: "/root/.m2/"
        name: "cachedir"
        readOnly: false
  - command:
      - "cat"
    env:
      - name: "LANGUAGE"
        value: "en_US:en"
      - name: "LC_ALL"
        value: "en_US.UTF-8"
      - name: "LANG"
        value: "en_US.UTF-8"
    image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
    imagePullPolicy: "IfNotPresent"
    name: "kubectl"
    tty: true
    volumeMounts:
      - mountPath: "/etc/localtime"
        name: "localtime"
        readOnly: false
  - command:
      - "cat"
    env:
      - name: "LANGUAGE"
        value: "en_US:en"
      - name: "LC_ALL"
        value: "en_US.UTF-8"
      - name: "LANG"
        value: "en_US.UTF-8"
    image: "registry.cn-beijing.aliyuncs.com/citools/docker:19.03.9-git"
```

```
        imagePullPolicy: "IfNotPresent"
        name: "docker"
        tty: true
        volumeMounts:
          - mountPath: "/etc/localtime"
            name: "localtime"
            readOnly: false
          - mountPath: "/var/run/docker.sock"
            name: "dockersock"
            readOnly: false
  restartPolicy: "Never"
  nodeSelector:
    build: "true"
  securityContext: {}
  volumes:
    - hostPath:
        path: "/var/run/docker.sock"
      name: "dockersock"
    - hostPath:
        path: "/usr/share/zoneinfo/Asia/Shanghai"
      name: "localtime"
    - name: "cachedir"
      hostPath:
        path: "/opt/m2"
"""
    }
}
  stages {
    stage('Pulling Code') {
      parallel {
        stage('Pulling Code by Jenkins') {
          when {
            expression {
              env.gitlabBranch == null
            }

          }
          steps {
            git(changelog: true, poll: true, url: 'git@10.103.236.251:kubernetes/vue-project.git',
branch: "${BRANCH}", credentialsId: 'gitlab-key')
            script {
              COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --pretty=format:'%h'").trim()
              TAG = BUILD_TAG + '-' + COMMIT_ID
              println "Current branch is ${BRANCH}, Commit ID is ${COMMIT_ID}, Image TAG is
```

```
${TAG}"

                }

            }
        }

        stage('Pulling Code by trigger') {
            when {
                expression {
                    env.gitlabBranch != null
                }

            }
            steps {
                git(url: 'git@10.103.236.251:kubernetes/vue-project.git', branch: env.gitlabBranch, changelog: true, poll: true, credentialsId: 'gitlab-key')
                script {
                    COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --pretty=format:'%h'").trim()
                    TAG = BUILD_TAG + '-' + COMMIT_ID
                    println "Current branch is ${env.gitlabBranch}, Commit ID is ${COMMIT_ID}, Image TAG is ${TAG}"
                }

            }
        }

    }
}

stage('Building') {
    steps {
        container(name: 'build') {
            sh """
                npm install --registry=https://registry.npm.taobao.org
                npm run build
            """
        }
    }
}

stage('Docker build for creating image') {
    environment {
        HARBOR_USER         = credentials('HARBOR_ACCOUNT')
```

```
    }
        steps {
            container(name: 'docker') {
                sh """
                echo ${HARBOR_USER_USR} ${HARBOR_USER_PSW} ${TAG}
                docker build -t ${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} .
                docker login -u ${HARBOR_USER_USR} -p ${HARBOR_USER_PSW} ${HARBOR_ADDRESS}
                docker push ${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG}
                """
            }
        }
    }

    stage('Deploying to K8s') {
        environment {
            MY_KUBECONFIG = credentials('study-k8s-kubeconfig')
    }
        steps {
            container(name: 'kubectl'){
                sh """
                /usr/local/bin/kubectl --kubeconfig $MY_KUBECONFIG set image deploy -l app=${IMAGE_NAME} ${IMAGE_NAME}=${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} -n $NAMESPACE
                """
            }
        }
    }

  }
  environment {
    COMMIT_ID = ""
    HARBOR_ADDRESS = "10.103.236.204"
    REGISTRY_DIR = "kubernetes"
    IMAGE_NAME = "vue-project"
    NAMESPACE = "kubernetes"
    TAG = ""
  }
  parameters {
    gitParameter(branch: '', branchFilter: 'origin/(.*)', defaultValue: '', description: 'Branch for build and deploy', name: 'BRANCH', quickFilterEnabled: false, selectedValue: 'NONE', sortMode: 'NONE', tagFilter: '*', type: 'PT_BRANCH')
  }
```

```
}
```

Jenkinsfile 和 Java 项目并无太大区别，需要更改的位置如下：

```
    pipeline {
      agent {
        kubernetes {
       ... # 此处代码有省略
           # 构建容器改为具有 NodeJS 环境的镜像，版本需要和公司项目一致
          image: "registry.cn-beijing.aliyuncs.com/citools/node:lts"
          imagePullPolicy: "IfNotPresent"
          name: "build"
          tty: true
          volumeMounts:
            - mountPath: "/etc/localtime"
              name: "localtime"
            - mountPath: "/root/.m2/" # NodeJS 的缓存目录为 node_modules，此处可以不配置，因为 workspace 采用的是 hostPath，该目录会被缓存到创建 Pod 节点的/opt/目录
              name: "cachedir"
              readOnly: false
       ... # 此处代码有省略
                # 此处的地址需要改为自己的 Git 地址
                git(changelog: true, poll: true, url: 'git@xxxxxxxxxx:kubernetes/vue-project.git', branch: "${BRANCH}", credentialsId: 'gitlab-key')
                ...

            }

            stage('Pulling Code by trigger') {
              when {
                expression {
                  env.gitlabBranch != null
                }

              }
              steps {
                # 此处的地址需要改为自己的 Git 地址
                git(url: 'git@xxxxxxxx:kubernetes/vue-project.git', branch: env.gitlabBranch, changelog: true, poll: true, credentialsId: 'gitlab-key')
                ...


         stage('Building') {
           steps {
             container(name: 'build') {
                sh """
                  # 此处为 NodeJS/Vue 项目的构建命令，根据实际情况进行修改
                  npm install --registry=https://registry.npm.taobao.org
                  npm run build
                """
             }
           }
         }

        ...
         REGISTRY_DIR = "kubernetes"
         IMAGE_NAME = "vue-project"  # 名字为 vue-project
         NAMESPACE = "kubernetes"
         TAG = ""
       ...
```

1.6.2 定义 Dockerfile

前端应用构建后一般会在 dist 文件下产生 html 文件，只需要拷贝到 nginx 的根目录下即可：

```
FROM registry.cn-beijing.aliyuncs.com/dotbalo/nginx:1.15.12

COPY dist/* /usr/share/nginx/html/
```

1.6.3 定义 Kubernetes 资源

对于 Kubernetes 的资源也是类似的，只需要更改资源名称和端口号即可（加粗部分）：

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: vue-project
  name: vue-project
  namespace: kubernetes
spec:
  ports:
  - name: web
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: vue-project
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: vue-project
  namespace: kubernetes
spec:
  rules:
  - host: vue-project.test.com
    http:
      paths:
      - backend:
          service:
            name: vue-project
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
```

```
      app: vue-project
    name: vue-project
    namespace: kubernetes
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: vue-project
    template:
      metadata:
        creationTimestamp: null
        labels:
          app: vue-project
      spec:
        affinity:
          podAntiAffinity:
            preferredDuringSchedulingIgnoredDuringExecution:
            - podAffinityTerm:
                labelSelector:
                  matchExpressions:
                  - key: app
                    operator: In
                    values:
                    - vue-project
                topologyKey: kubernetes.io/hostname
              weight: 100
  ...
          lifecycle: {}
          livenessProbe:
            failureThreshold: 2
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 80
            timeoutSeconds: 2
          name: vue-project
          ports:
          - containerPort: 80
            name: web
            protocol: TCP
          readinessProbe:
            failureThreshold: 2
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 80
            timeoutSeconds: 2
    ...
```

完整文件：

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: vue-project
  name: vue-project
  namespace: kubernetes
spec:
  ports:
  - name: web
```

```
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: vue-project
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: vue-project
  namespace: kubernetes
spec:
  rules:
  - host: vue-project.test.com
    http:
      paths:
      - backend:
          service:
            name: vue-project
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: vue-project
  name: vue-project
  namespace: kubernetes
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vue-project
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: vue-project
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - vue-project
```

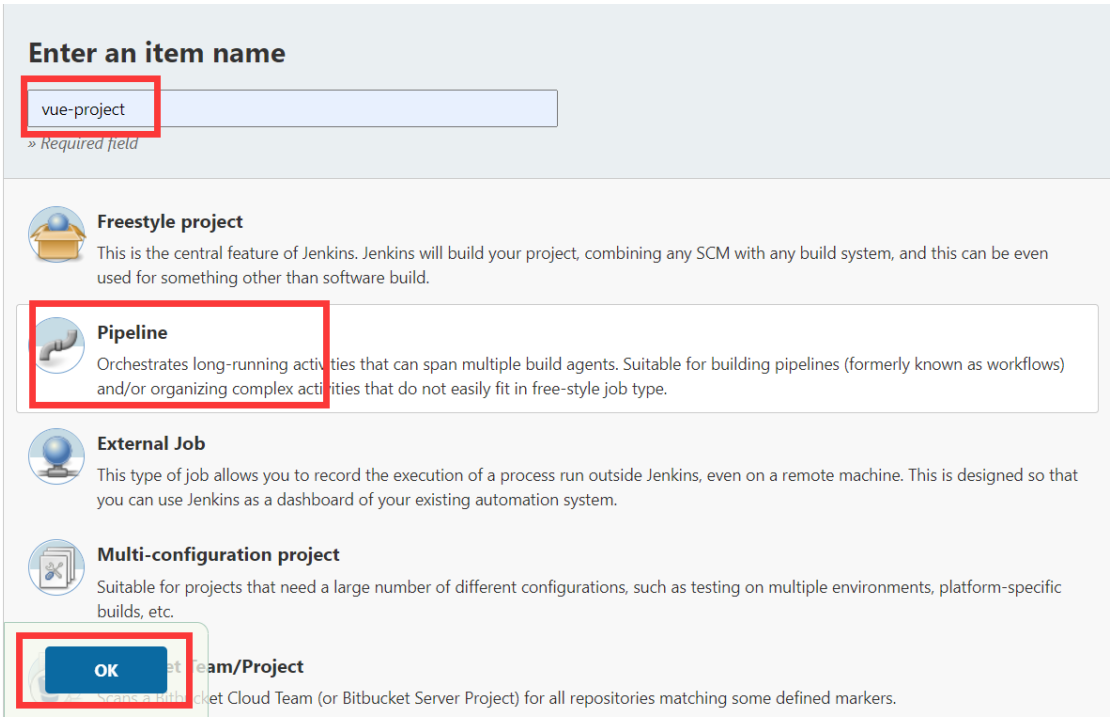
```
              topologyKey: kubernetes.io/hostname
            weight: 100
        containers:
        - env:
          - name: TZ
            value: Asia/Shanghai
          - name: LANG
            value: C.UTF-8
          image: nginx
          imagePullPolicy: IfNotPresent
          lifecycle: {}
          livenessProbe:
            failureThreshold: 2
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 80
            timeoutSeconds: 2
          name: vue-project
          ports:
          - containerPort: 80
            name: web
            protocol: TCP
          readinessProbe:
            failureThreshold: 2
            initialDelaySeconds: 30
            periodSeconds: 10
            successThreshold: 1
            tcpSocket:
              port: 80
            timeoutSeconds: 2
          resources:
            limits:
              cpu: 994m
              memory: 1170Mi
            requests:
              cpu: 10m
              memory: 55Mi
        dnsPolicy: ClusterFirst
        imagePullSecrets:
        - name: harborkey
        restartPolicy: Always
        securityContext: {}
        serviceAccountName: default
```

创建该资源：

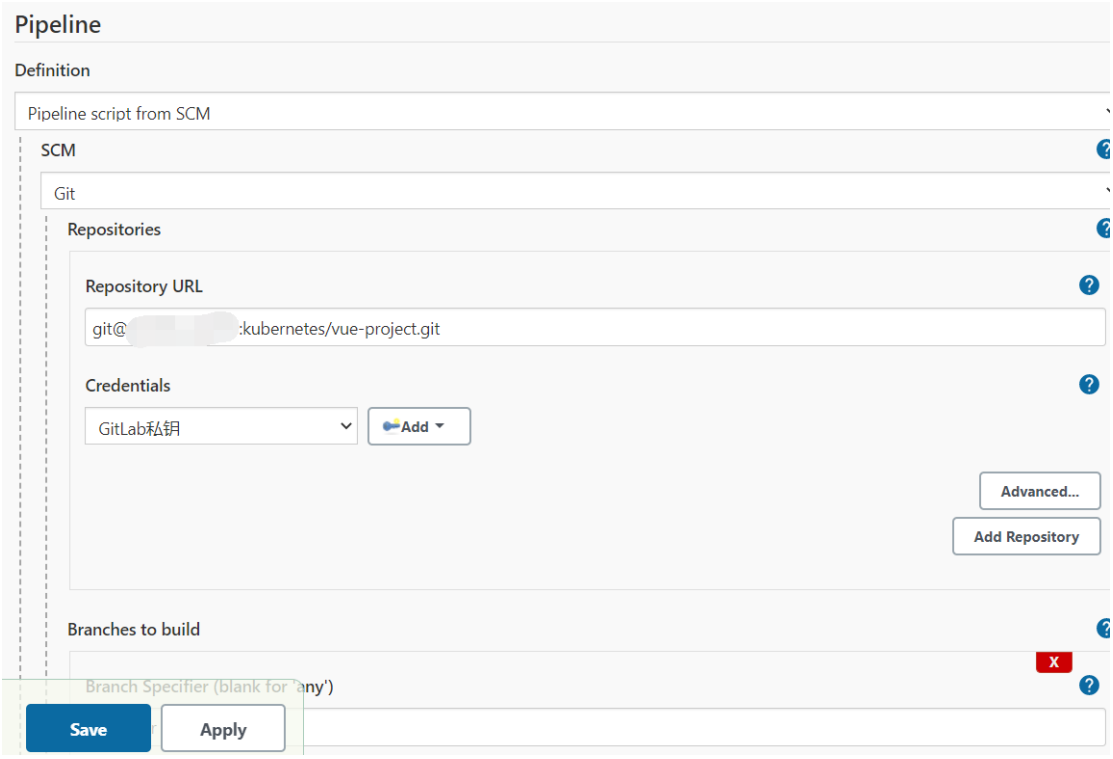
```
# kubectl create -f vue-project.yaml
service/vue-project created
ingress.networking.k8s.io/vue-project created
deployment.apps/vue-project created
```

1.6.4 创建 Jenkins Job

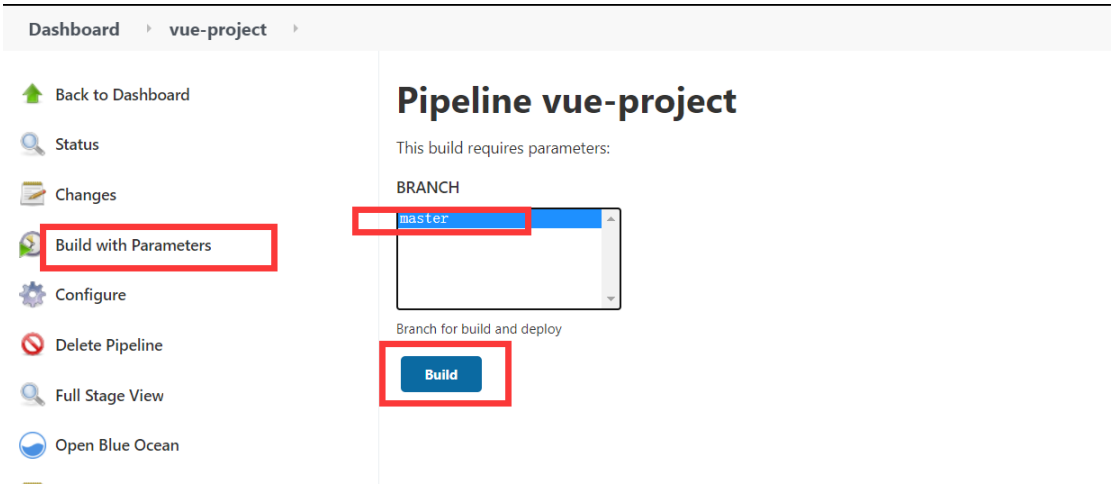
创建 Jenkins Job 和之前并无太大区别。首先创建 Pipeline 类型的 Job，名称为 vue-project：



同样在 Pipeline 栏，写上地址、Key 和分支：



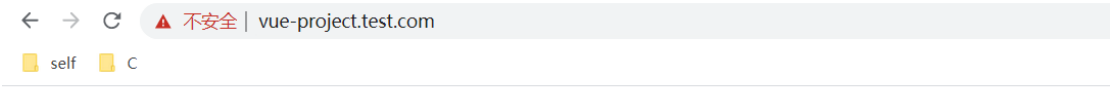
同样的，第一次构建也会失败，第二次可以选择分支构建：



当出现 SUCCESS 时，表示构建结束：

```
bf9c09d' -n kubernetes
deployment.apps/vue-project image updated
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

之后可以在浏览器访问该域名：



Vue Project For DevOps

1.7 自动化构建 Golang 项目

上述演示了 Java 和前端应用的自动化，接下来演示一下对于 Golang 的自动化构建，本次示例的代码地址：https://gitee.com/dukuan/go-project.git。

1.7.1 定义 Jenkinsfile

本次示例的 Jenkinsfile 和之前的也无太大区别，需要的更改的位置是构建容器的镜像、缓存目录、Git 地址和项目名称：

```
    pipeline {
      agent {
        kubernetes {
          ...
          # 构建镜像改为 golang，需要根据实际情况更改版本
          image: "registry.cn-beijing.aliyuncs.com/citools/golang:1.15"
```

```
      ...
         # 缓存目录为/go/pkg/，执行 go build 时下载的依赖包会缓存在该目录
         - mountPath: "/go/pkg/"
           name: "cachedir"
           readOnly: false
    ... # 缓存目录保存的位置
      - name: "cachedir"
        hostPath:
          path: "/opt/gopkg"
  '''
      }
  }
  ...

      # 构建命令
      stage('Building') {
        steps {
          container(name: 'build') {
              sh """
                export GO111MODULE=on
                go env -w GOPROXY=https://goproxy.cn,direct
                go build
              """
          }
        }
      }

      ...
```

完整文件：

```
  pipeline {
    agent {
      kubernetes {
        cloud 'kubernetes-study'
        slaveConnectTimeout 1200
        workspaceVolume hostPathWorkspaceVolume(hostPath: "/opt/workspace",
readOnly: false)
        yaml '''
  apiVersion: v1
  kind: Pod
  spec:
    containers:
      - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
        image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
        name: jnlp
        imagePullPolicy: IfNotPresent
        volumeMounts:
          - mountPath: "/etc/localtime"
            name: "localtime"
            readOnly: false
      - command:
          - "cat"
        env:
          - name: "LANGUAGE"
            value: "en_US:en"
          - name: "LC_ALL"
            value: "en_US.UTF-8"
          - name: "LANG"
            value: "en_US.UTF-8"
        image: "registry.cn-beijing.aliyuncs.com/citools/golang:1.15"
        imagePullPolicy: "IfNotPresent"
        name: "build"
        tty: true
```

```
        volumeMounts:
          - mountPath: "/etc/localtime"
            name: "localtime"
          - mountPath: "/go/pkg/"
            name: "cachedir"
            readOnly: false
      - command:
          - "cat"
        env:
          - name: "LANGUAGE"
            value: "en_US:en"
          - name: "LC_ALL"
            value: "en_US.UTF-8"
          - name: "LANG"
            value: "en_US.UTF-8"
        image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
        imagePullPolicy: "IfNotPresent"
        name: "kubectl"
        tty: true
        volumeMounts:
          - mountPath: "/etc/localtime"
            name: "localtime"
            readOnly: false
      - command:
          - "cat"
        env:
          - name: "LANGUAGE"
            value: "en_US:en"
          - name: "LC_ALL"
            value: "en_US.UTF-8"
          - name: "LANG"
            value: "en_US.UTF-8"
        image: "registry.cn-beijing.aliyuncs.com/citools/docker:19.03.9-
git"
        imagePullPolicy: "IfNotPresent"
        name: "docker"
        tty: true
        volumeMounts:
          - mountPath: "/etc/localtime"
            name: "localtime"
            readOnly: false
          - mountPath: "/var/run/docker.sock"
            name: "dockersock"
            readOnly: false
    restartPolicy: "Never"
    nodeSelector:
      build: "true"
    securityContext: {}
    volumes:
      - hostPath:
          path: "/var/run/docker.sock"
        name: "dockersock"
      - hostPath:
          path: "/usr/share/zoneinfo/Asia/Shanghai"
        name: "localtime"
      - name: "cachedir"
        hostPath:
          path: "/opt/gopkg"
  '''
    }
  }
    stages {
      stage('Pulling Code') {
```

```
        parallel {
          stage('Pulling Code by Jenkins') {
            when {
              expression {
                env.gitlabBranch == null
              }

            }
            steps {
              git(changelog: true, poll: true, url:
'git@10.103.236.251:kubernetes/go-project.git', branch: "${BRANCH}",
credentialsId: 'gitlab-key')
              script {
                COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --
pretty=format:'%h'").trim()
                TAG = BUILD_TAG + '-' + COMMIT_ID
                println "Current branch is ${BRANCH}, Commit ID is
${COMMIT_ID}, Image TAG is ${TAG}"

              }

            }
          }

          stage('Pulling Code by trigger') {
            when {
              expression {
                env.gitlabBranch != null
              }

            }
            steps {
              git(url: 'git@10.103.236.251:kubernetes/go-project.git',
branch: env.gitlabBranch, changelog: true, poll: true, credentialsId:
'gitlab-key')
              script {
                COMMIT_ID = sh(returnStdout: true, script: "git log -n 1 --
pretty=format:'%h'").trim()
                TAG = BUILD_TAG + '-' + COMMIT_ID
                println "Current branch is ${env.gitlabBranch}, Commit ID is
${COMMIT_ID}, Image TAG is ${TAG}"
              }

            }
          }

        }
      }

      stage('Building') {
        steps {
          container(name: 'build') {
            sh """
              export GO111MODULE=on
              go env -w GOPROXY=https://goproxy.cn,direct
              go build
            """
          }
        }
      }

      stage('Docker build for creating image') {
        environment {
```

```
          HARBOR_USER    = credentials('HARBOR_ACCOUNT')
       }
         steps {
           container(name: 'docker') {
             sh """
             pwd
             ls
             echo ${HARBOR_USER_USR} ${HARBOR_USER_PSW} ${TAG}
             docker build -t
${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} .
             docker login -u ${HARBOR_USER_USR} -p ${HARBOR_USER_PSW}
${HARBOR_ADDRESS}
             docker push
${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG}
             """
           }
         }
       }

       stage('Deploying to K8s') {
         environment {
           MY_KUBECONFIG = credentials('study-k8s-kubeconfig')
       }
         steps {
           container(name: 'kubectl'){
              sh """
              /usr/local/bin/kubectl --kubeconfig $MY_KUBECONFIG set image
deploy -l app=${IMAGE_NAME}
${IMAGE_NAME}=${HARBOR_ADDRESS}/${REGISTRY_DIR}/${IMAGE_NAME}:${TAG} -n
$NAMESPACE
             """
           }
         }
       }

     }
     environment {
       COMMIT_ID = ""
       HARBOR_ADDRESS = "10.103.236.203"
       REGISTRY_DIR = "kubernetes"
       IMAGE_NAME = "go-project"
       NAMESPACE = "kubernetes"
       TAG = ""
     }
     parameters {
       gitParameter(branch: '', branchFilter: 'origin/(.*)', defaultValue:
'', description: 'Branch for build and deploy', name: 'BRANCH',
quickFilterEnabled: false, selectedValue: 'NONE', sortMode: 'NONE',
tagFilter: '*', type: 'PT_BRANCH')
     }
   }
```

1.7.2 定义 Dockerfile

和之前不一样的地方是，Golang 编译后生成的是一个二进制文件，可以直接执行，所以底层镜像设置为 alpine 或者其它的小镜像即可：

```
FROM registry.cn-beijing.aliyuncs.com/dotbalo/alpine-glibc:alpine-3.9

COPY conf/  ./conf # 如果定义了单独的配置文件，可能需要拷贝到镜像中
COPY ./go-project ./ # 包名按照实际情况进行修改
```

```
ENTRYPOINT [ "./go-project"] # 启动该应用
```

完整文件：

```
FROM registry.cn-beijing.aliyuncs.com/dotbalo/alpine-glibc:alpine-3.9
 # 如果定义了单独的配置文件，可能需要拷贝到镜像中
COPY conf/  ./conf
 # 包名按照实际情况进行修改
COPY ./go-project ./

ENTRYPOINT [ "./go-project"] # 启动该应用
```

1.7.3 定义 Kubernetes 资源

```
---
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: go-project
  name: go-project
  namespace: kubernetes
spec:
  ports:
  - name: web
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: go-project
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  creationTimestamp: null
  name: go-project
  namespace: kubernetes
spec:
  rules:
  - host: go-project.test.com
    http:
      paths:
      - backend:
          service:
            name: go-project
            port:
              number: 8080
        path: /
        pathType: ImplementationSpecific
---
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: go-project
```

```
  name: go-project
  namespace: kubernetes
spec:
  replicas: 1
  selector:
    matchLabels:
      app: go-project
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: go-project
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - go-project
              topologyKey: kubernetes.io/hostname
            weight: 100
      containers:
      - env:
        - name: TZ
          value: Asia/Shanghai
        - name: LANG
          value: C.UTF-8
        image: nginx
        imagePullPolicy: IfNotPresent
        lifecycle: {}
        livenessProbe:
          failureThreshold: 2
          initialDelaySeconds: 30
          periodSeconds: 10
          successThreshold: 1
          tcpSocket:
            port: 8080
          timeoutSeconds: 2
        name: go-project
        ports:
        - containerPort: 8080
          name: web
          protocol: TCP
        readinessProbe:
          failureThreshold: 2
          initialDelaySeconds: 30
          periodSeconds: 10
          successThreshold: 1
          tcpSocket:
            port: 8080
          timeoutSeconds: 2
        resources:
          limits:
            cpu: 994m
```

```
            memory: 1170Mi
          requests:
            cpu: 10m
            memory: 55Mi
      dnsPolicy: ClusterFirst
      imagePullSecrets:
      - name: harborkey
      restartPolicy: Always
      securityContext: {}
      serviceAccountName: default
```

1.7.4 创建 Jenkins Job

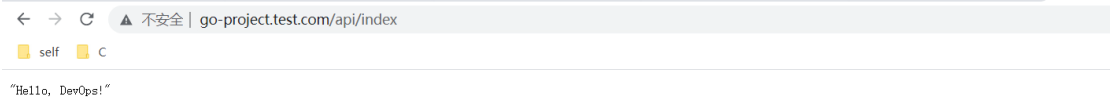
直接创建即可：

```
# kubectl create -f go-project.yaml
service/go-project created
ingress.networking.k8s.io/go-project created
deployment.apps/go-project created
```

创建 Jenkins Job 和上述小节也并无区别，在此不再重复演示。创建完成后，可以自行构建，看到如下信息表示创建成功：

```
deployment.apps/go-project image updated
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

之后即可访问该应用（如果是后端项目可以无需创建 Ingress 暴露服务）：



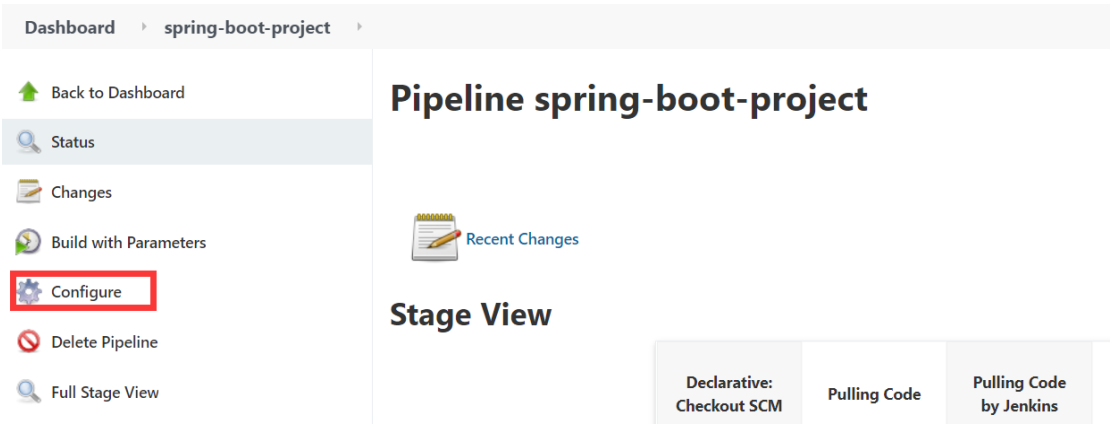
之后在创建 Pod 的节点，可以看到缓存目录：

```
[root@k8s-node01 gopkg]# cd /opt/gopkg/
[root@k8s-node01 gopkg]# ls
mod  sumdb
```

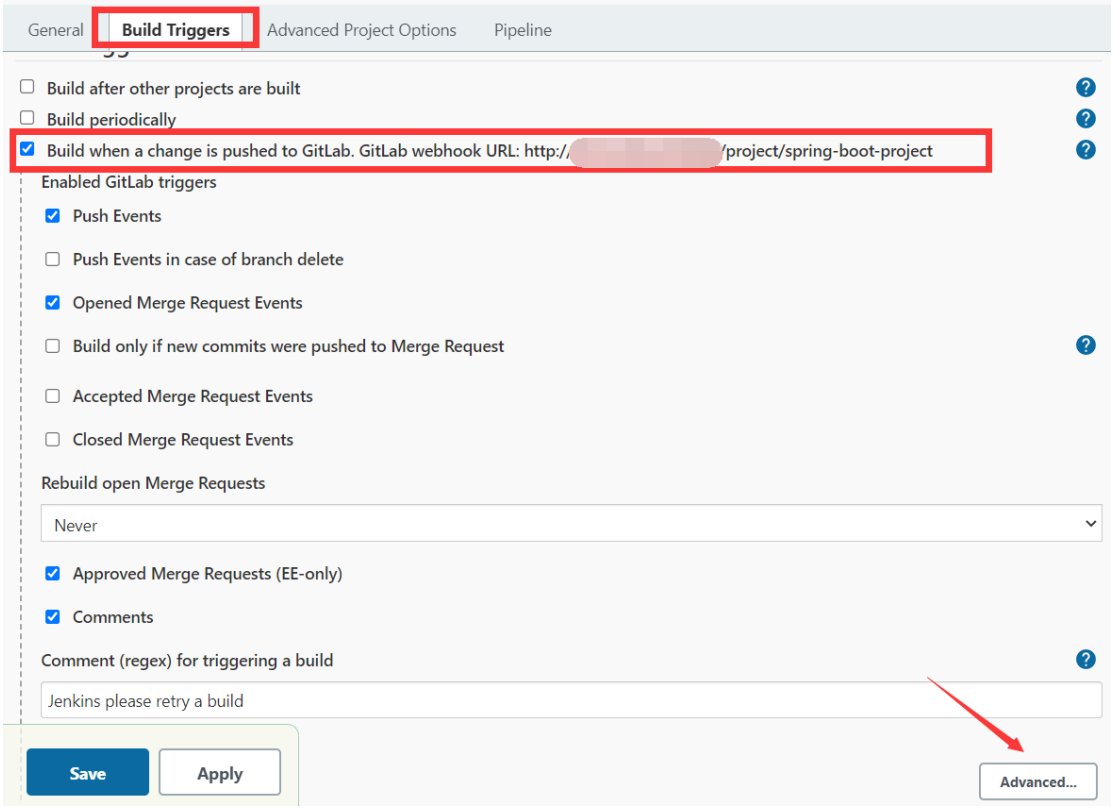
1.8 自动触发构建

之前的构建都是采用手动选择分支进行构建的，实际使用时，项目可能有很多，如果都是手动触发可能比较消耗人力。所以推荐可以按需配置自动触发，即提交代码后自动触发 Jenkins 进行构建任务。

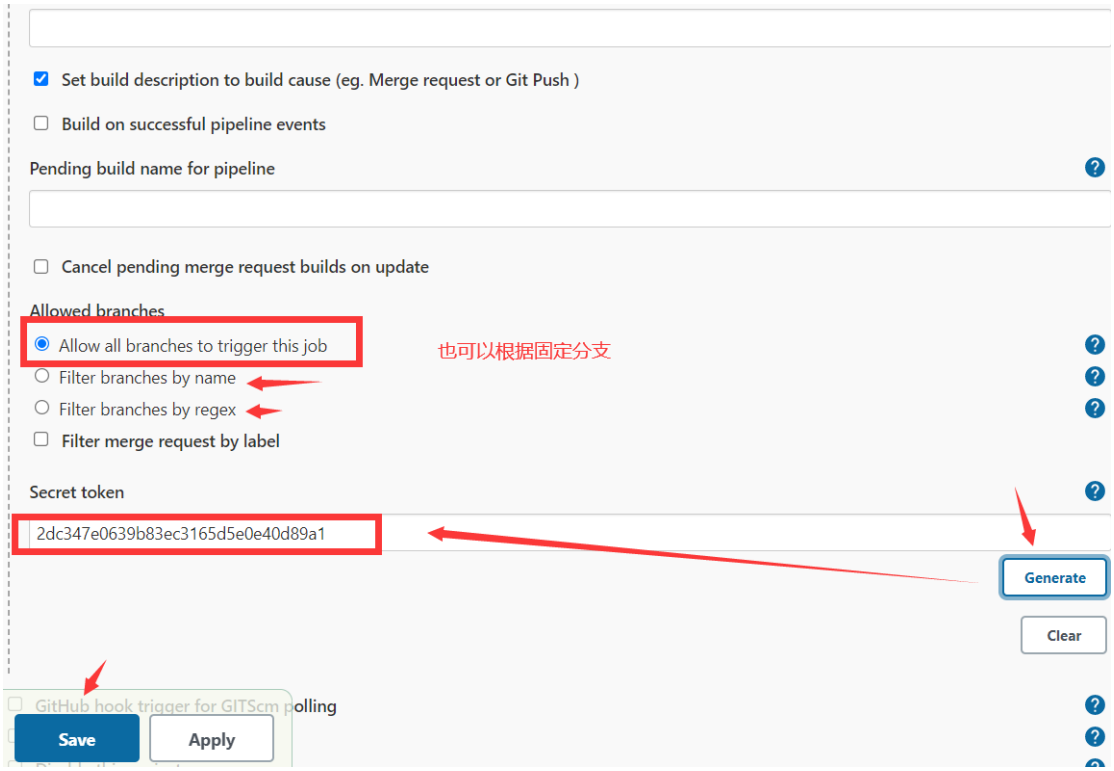
本次用 Java 项目进行演示。首先找到 Java 项目的 Job，点击 Configure：



之后选择 Build Triggers，勾选 Build when a change…，记录 webhook URL：

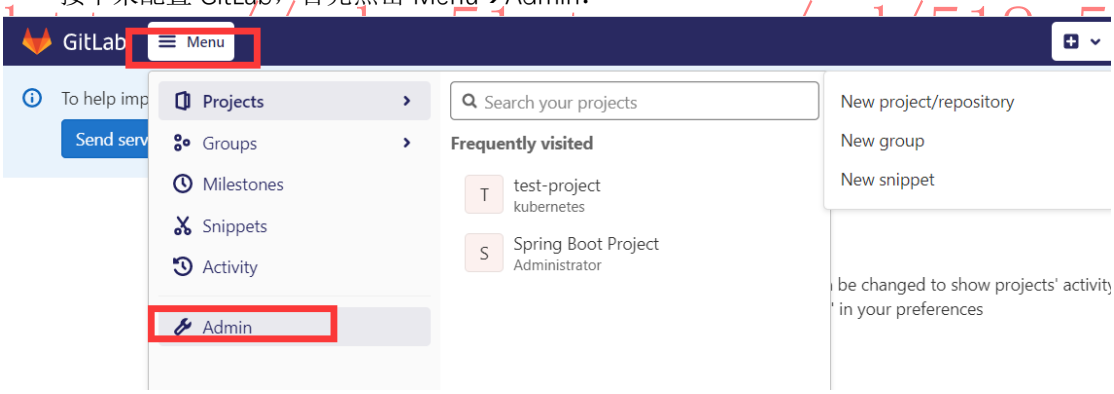


之后点击 Advanced：

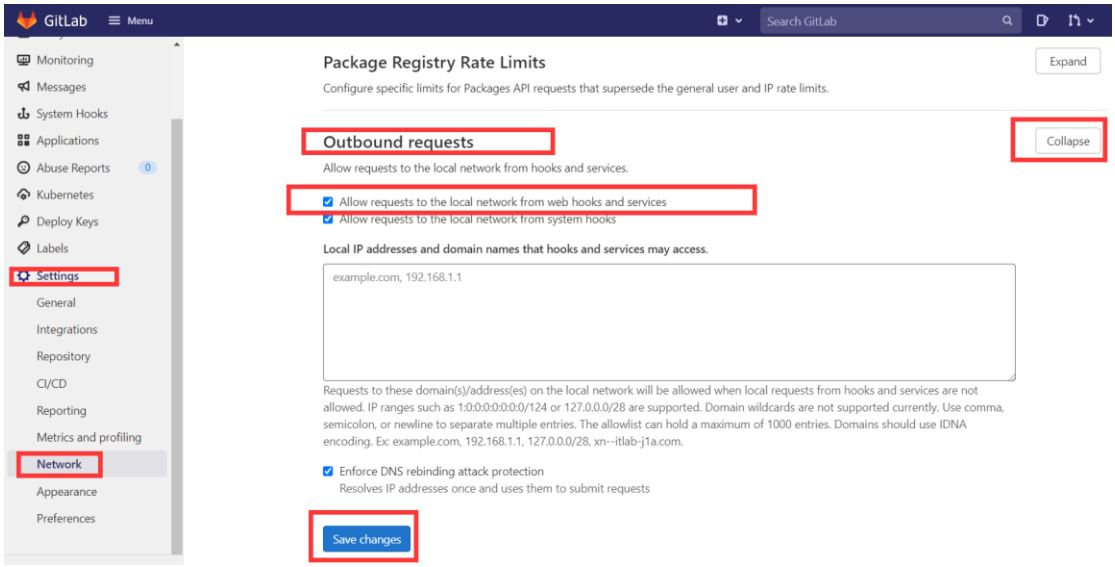


选择 Allow all branches，如果不想任何分支都可以触发该流水线，可以选择 Filter 进行条件匹配。之后点击 Generate 生成 Secret token，最后点击 Save 即可。

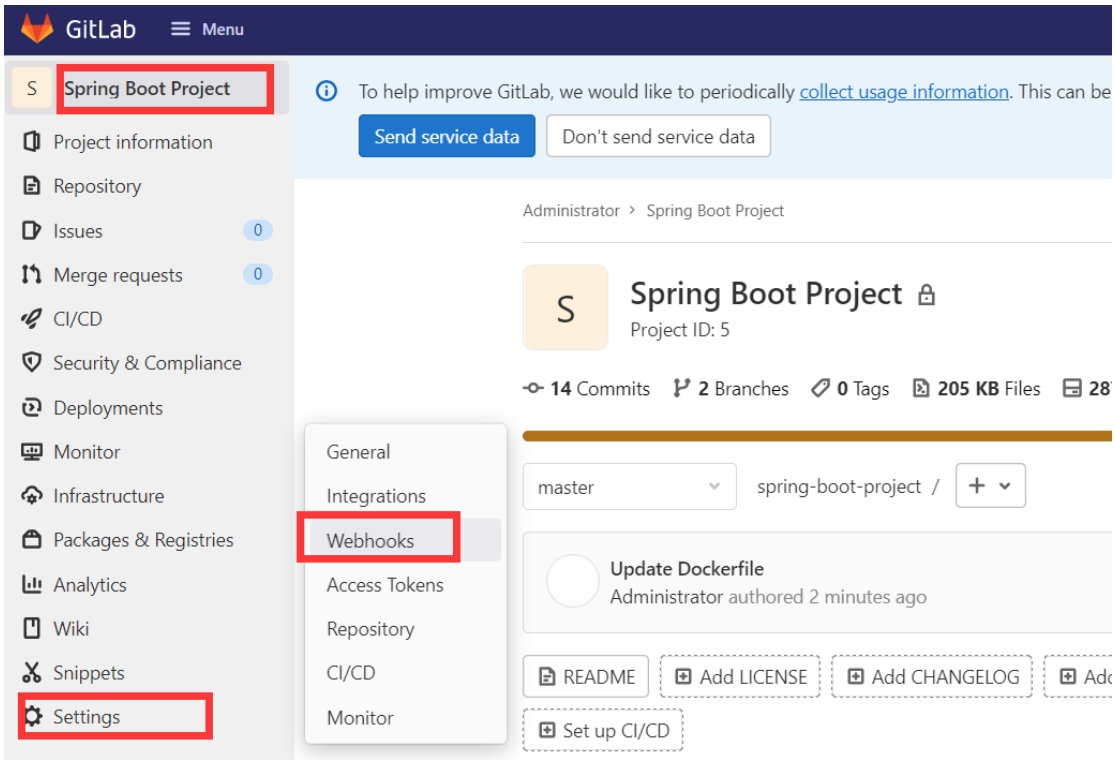
接下来配置 GitLab，首先点击 Menu→Admin：



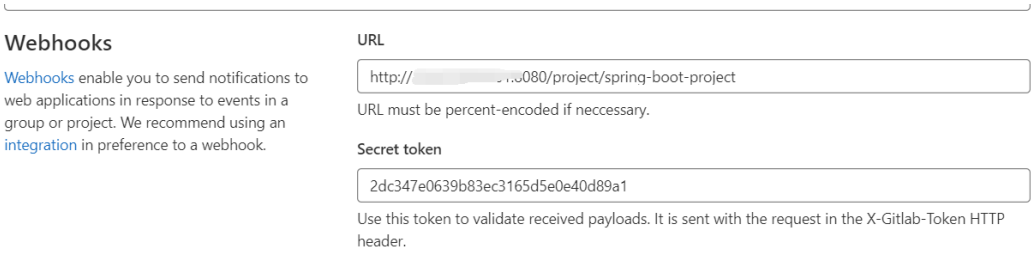
选择 Settings→Network，之后允许访问外部的请求：



保存后，找到 Java 项目，点击 Settings→WebHooks：

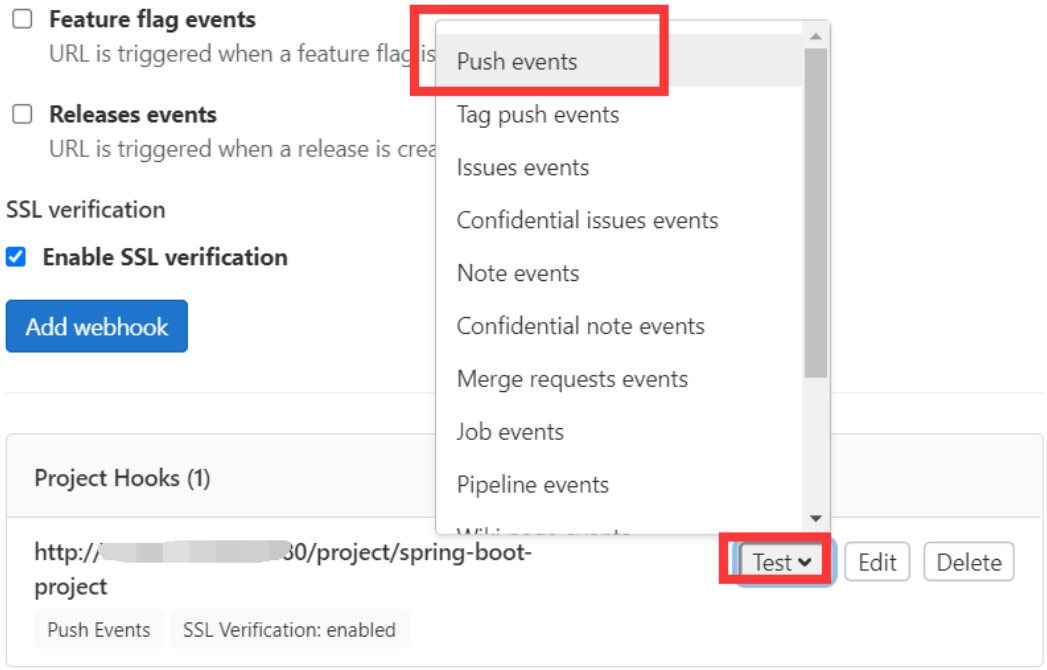


在新页面输入 Webhook 地址和 token：

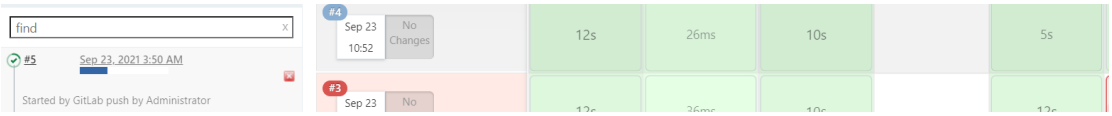


确认无误后，点击 Add webhook：

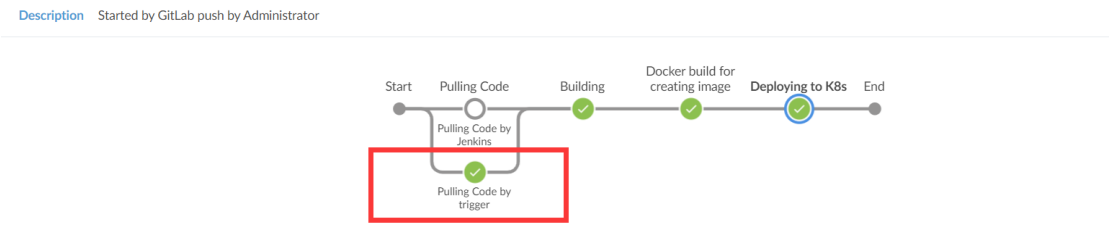
之后下方会添加一个新的 Project Hooks，可以点击 Test 进行 Push 测试：



点击后，即可在 Jenkins 页面看到任务被触发：



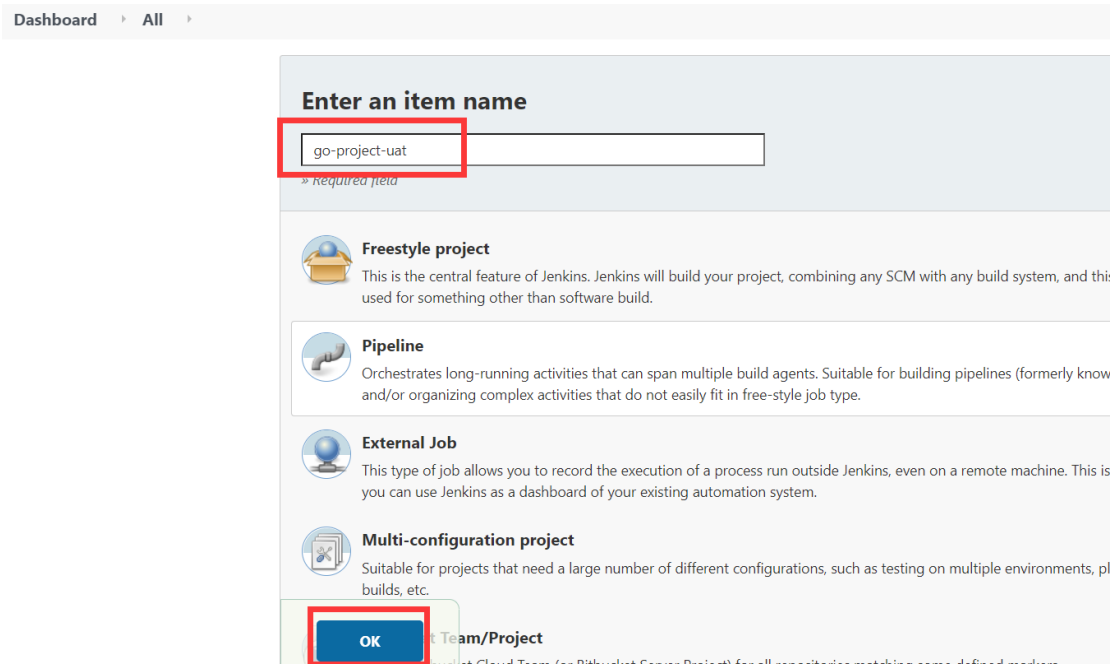
也可以通过 Blue Ocean 看到是自动触发的 stage 被执行：



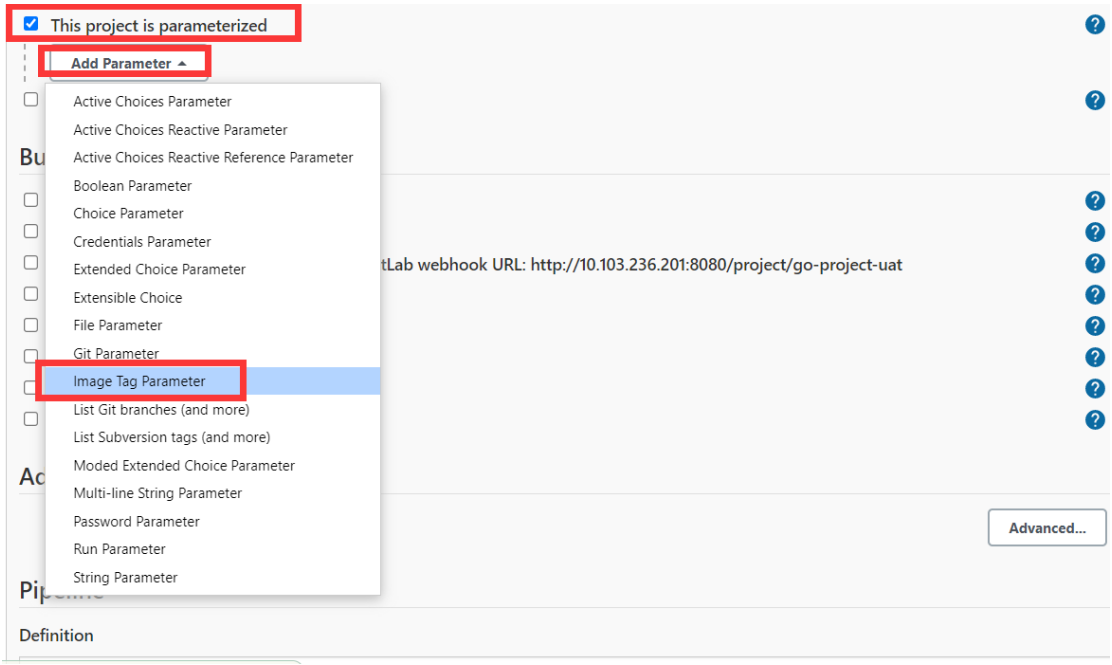
以上就是通过 GitLab 的事件触发 Jenkins 任务，在实际使用时，此功能非常常用，一般会用于开发、测试等环境，省去了手动构建的过程。而在 UAT 和生产环境，一般不需要再次构建，而是选择其它环境产生的镜像进行发版，接下来看一下如何进行不构建进行发版。

1.9 一次构建多次部署

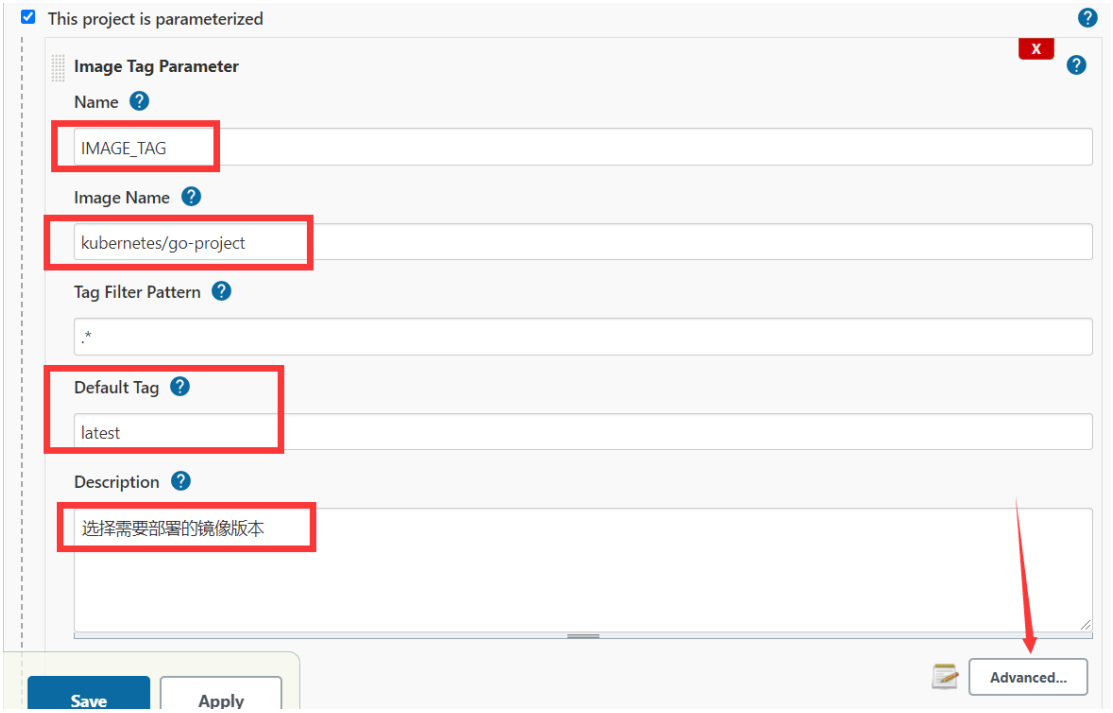
创建一个新的 Job，名字为 go-project-uat，类型 Pipeline：



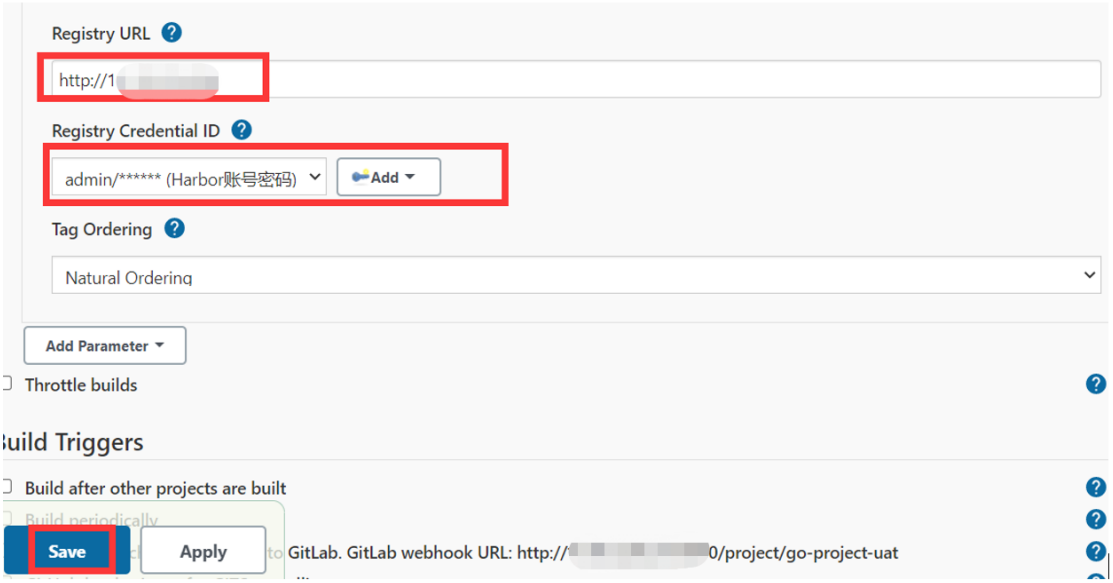
点击页面的 This Project is parameterized（参数化构建）：



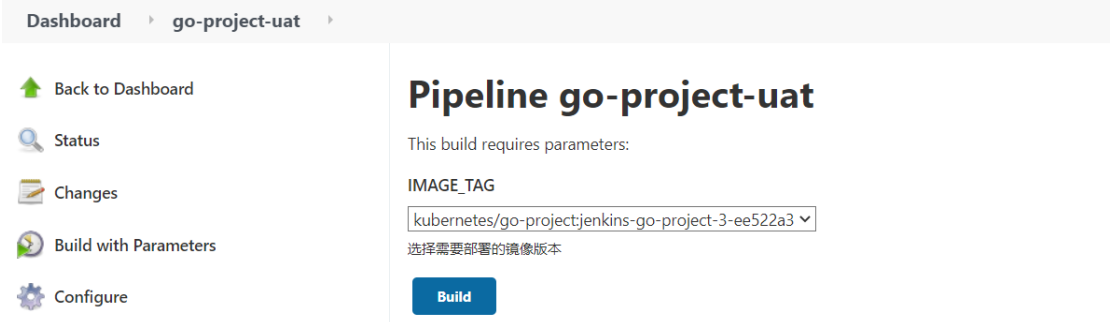
选择参数类型为 Image Tag Parameter（需要安装 Image Tag 插件），之后定义 Name 为变量的名称，Iamge Name 为 Harbor 的目录和镜像名称：



点击 Advance，输入仓库的地址，注意如果配置了证书，需要配置 https：



之后可以先点击 Save，然后测试能否获取到进行 Tag。保存后点击 Build with Parameters：



之后点击 Configure，添加 Pipeline 脚本：

```
pipeline {
    agent {
    kubernetes {
```

```
        cloud 'kubernetes-study'
        slaveConnectTimeout 1200
        yaml '''
    apiVersion: v1
    kind: Pod
    spec:
      containers:
        # 只需要配置 jnlp 和 kubectl 镜像即可
        - args: [\'$(JENKINS_SECRET)\', \'$(JENKINS_NAME)\']
          image: 'registry.cn-beijing.aliyuncs.com/citools/jnlp:alpine'
          name: jnlp
          imagePullPolicy: IfNotPresent
        - command:
            - "cat"
          env:
            - name: "LANGUAGE"
              value: "en_US:en"
            - name: "LC_ALL"
              value: "en_US.UTF-8"
            - name: "LANG"
              value: "en_US.UTF-8"
          image: "registry.cn-beijing.aliyuncs.com/citools/kubectl:self-1.17"
          imagePullPolicy: "IfNotPresent"
          name: "kubectl"
          tty: true
      restartPolicy: "Never"
    '''
        }
    }

       stages {
          stage('Deploy') {
             environment {
                 MY_KUBECONFIG = credentials('study-k8s-kubeconfig')
             }
             steps {
              container(name: 'kubectl'){
                sh """
                   echo ${IMAGE_TAG} # 该变量即为前台选择的镜像
                   kubectl --kubeconfig=${MY_KUBECONFIG} set image deployment -
l app=${IMAGE_NAME} ${IMAGE_NAME}=${HARBOR_ADDRESS}/${IMAGE_TAG} -n
${NAMESPACE}
                   kubectl --kubeconfig=${MY_KUBECONFIG} get po  -l
app=${IMAGE_NAME} -n ${NAMESPACE} -w
                """
             }
             }
          }
       }
       environment {
        HARBOR_ADDRESS = "HARBOR_ADDRESS"
        NAMESPACE = "kubernetes"
         IMAGE_NAME = "go-project"
        TAG = ""
      }
    }
```

保存后，选择一个镜像，点击 Build：

```
+ echo kubernetes/go-project:jenkins-go-project-3-ee522a3
kubernetes/go-project:jenkins-go-project-3-ee522a3
+ kubectl '--kubeconfig=****' set image deployment -l 'app=go-project' 'go-project=10.103.236.204/kubernetes/go-project:jenkins-go-project-3-ee522a3' -n kubernetes
deployment.apps/go-project image updated
+ kubectl '--kubeconfig=****' get po -l 'app=go-project' -n kubernetes -w
NAME                          READY   STATUS    RESTARTS   AGE
go-project-5ffb6d7f44-gg2kd   0/1     Pending   0          0s
go-project-66b78cd967-27xgd   1/1     Running   0          3h5m
go-project-5ffb6d7f44-gg2kd   0/1     ContainerCreating   0          0s
go-project-5ffb6d7f44-gg2kd   0/1     ContainerCreating   0          1s
go-project-5ffb6d7f44-gg2kd   0/1     Running             0          2s
go-project-5ffb6d7f44-gg2kd   1/1     Running             0          40s
go-project-66b78cd967-27xgd   1/1     Terminating         0          3h5m
go-project-66b78cd967-27xgd   1/1     Terminating         0          3h5m
go-project-66b78cd967-27xgd   0/1     Terminating         0          3h5m
go-project-66b78cd967-27xgd   0/1     Terminating         0          3h5m
go-project-66b78cd967-27xgd   0/1     Terminating         0          3h5m
[Pipeline] }
[Pipeline] // container
[Pipeline] }
[Pipeline] // withCredentials
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // podTemplate
[Pipeline] End of Pipeline
Finished: SUCCESS
```

即可看到是直接将镜像版本更新至 Kubernetes，并无构建过程，可以省下很多时间。该流水线也可以选择之前的版本进行回滚操作。