OpenShift Container Platform 4.16

CI/CD

包含有关 OpenShift Container Platform 构建、管道和 GitOps 的信息

Last Updated: 2024-07-03

OpenShift Container Platform 4.16 CI/CD

包含有关 OpenShift Container Platform 构建、管道和 GitOps 的信息

法律通告

摘要

OpenShift Container Platform 的 CI/CD

目录

第 1 章 OPENSHIFT CONTAINER PLATFORM CI/CD 概述

OpenShift Container Platform 是面向开发人员的企业就绪 Kubernetes 平台，使组织能够通过 DevOps 实践（如持续集成(CI)和持续交付(CD)）自动化应用程序交付流程。为了满足您的机构需求，OpenShift Container Platform 提供以下 CI/CD 解决方案：

- OpenShift 构建
- OpenShift Pipelines
- OpenShift GitOps

1.1. OPENSHIFT 构建

使用 OpenShift 构建时，您可以使用声明性构建过程创建云原生应用程序。您可以在用于创建 BuildConfig 对象的 YAML 文件中定义构建过程。此定义包括构建触发器、输入参数和源代码等属性。部署之后，BuildConfig 对象通常构建可运行的镜像并将其推送到容器镜像 registry。

OpenShift 构建为构建策略提供以下可扩展的支持：

- Docker 构建
- Source-to-image（S2I）构建
- Custom 构建

如需更多信息，请参阅了解镜像构建

1.2. OPENSHIFT PIPELINES

OpenShift Pipelines 提供了一个 Kubernetes 原生 CI/CD 框架，用于在其自己的容器中设计和运行 CI/CD 管道的每个步骤。它可以通过可预测的结果独立扩展以满足按需管道。

如需更多信息，请参阅了解 OpenShift Pipelines。

1.3. OPENSHIFT GITOPS

OpenShift GitOps 是一个使用 Argo CD 作为声明性 GitOps 引擎的 Operator。它启用了多集群 OpenShift 和 Kubernetes 基础架构的 GitOps 工作流。使用 OpenShift GitOps，管理员可以在集群和开发生命周期中一致地配置和部署基于 Kubernetes 的基础架构和应用程序。

如需更多信息，请参阅关于 Red Hat OpenShift GitOps 。

1.4. JENKINS

Jenkins 自动化了构建、测试和部署应用和项目的过程。OpenShift 开发者工具提供 Jenkins 镜像，它直接与 OpenShift Container Platform 集成。Jenkins 可通过使用 Samples Operator 模板或认证的 Helm Chart 在 OpenShift 上部署。

第 2 章 构建（BUILD）

2.1. 理解镜像构建

2.1.1. Builds

构建 (build) 是将输入参数转换为结果对象的过程。此过程最常用于将输入参数或源代码转换为可运行的镜像。**BuildConfig** 对象是整个构建过程的定义。

OpenShift Container Platform 使用 Kubernetes，从构建镜像创建容器并将它们推送到容器镜像 registry。

构建对象具有共同的特征，包括构建的输入，完成构建过程要满足的要求、构建过程日志记录、从成功构建中发布资源，以及发布构建的最终状态。构建会使用资源限制，具体是指定资源限值，如 CPU 使用量、内存使用量，以及构建或 Pod 执行时间。

OpenShift Container Platform 构建系统提供对构建策略的可扩展支持，它们基于构建 API 中指定的可选择类型。可用的构建策略主要有三种：

- Docker 构建
- Source-to-image（S2I）构建
- Custom 构建

默认情况下，支持 docker 构建和 S2I 构建。

构建生成的对象取决于用于创建它的构建器（builder）。对于 docker 和 S2I 构建，生成的对象为可运行的镜像。对于自定义构建，生成的对象是构建器镜像作者指定的任何事物。

此外，也可利用管道构建策略来实现复杂的工作流：

- 持续集成
- 持续部署

2.1.1.1. Docker 构建

OpenShift Container Platform 使用 Buildah 从 Dockerfile 构建容器镜像。有关使用 Dockerfile 构建容器镜像的更多信息，请参阅 Dockerfile 参考文档。

提示

如果使用 **buildArgs** 数组设置 Docker 构建参数，请参阅 Dockerfile 参考文档中 了解 ARG 和 FROM 如何交互。

2.1.1.2. Source-to-image 构建

Source-to-Image (S2I) 是一种用于构建可重复生成的容器镜像的工具。它通过将应用程序源代码注入容器镜像并汇编新镜像来生成可随时运行的镜像。新镜像融合了基础镜像（构建器）和构建的源代码，并可搭配 **buildah run** 命令使用。S2I 支持递增构建，可重复利用以前下载的依赖项和过去构建的工件等。

2.1.1.3. Custom 构建

采用自定义构建策略时，开发人员可以定义负责整个构建过程的特定构建器镜像。通过利用自己的构建器镜像，可以自定义构建流程。

自定义构建器镜像是嵌入了构建过程逻辑的普通容器镜像，例如用于构建 RPM 或基本镜像的逻辑。

自定义构建以级别很高的特权运行，默认情况下不可供用户使用。只有可赋予集群管理权限的用户才应被授予运行自定义构建的权限。

2.1.1.4. Pipeline 构建

重要

Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

采用 Pipeline 构建策略时，开发人员可以定义 Jenkins 管道，供 Jenkins 管道插件使用。构建可以由 OpenShift Container Platform 启动、监控和管理，其方式与任何其他构建类型相同。

Pipeline 工作流在 **jenkinsfile** 中定义，或直接嵌入在构建配置中，或者在 Git 存储库中提供并由构建配置引用。

2.2. 了解构建配置

以下小节定义了构建、构建配置和可用的主要构建策略的概念。

2.2.1. BuildConfig

构建配置描述单个构建定义，以及一组规定何时创建新构建的触发器（trigger）。构建配置通过 **BuildConfig** 定义，它是一种 REST 对象，可在对 API 服务器的 POST 中使用以创建新实例。

构建配置或 **BuildConfig** 的特征就是构建策略和一个或多个源。策略决定流程，而源则提供输入。

根据您选择使用 OpenShift Container Platform 创建应用程序的方式，如果使用 Web 控制台或 CLI，通常会自动生成 **BuildConfig**，并且可以随时对其进行编辑。如果选择稍后手动更改配置，则了解 **BuildConfig** 的组成部分及可用选项可能会有所帮助。

以下示例 **BuildConfig** 在每次容器镜像标签或源代码改变时产生新的构建：

BuildConfig 对象定义

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: "ruby-sample-build" 1
spec:
  runPolicy: "Serial" 2
  triggers: 3
    -
      type: "GitHub"
      github:
        secret: "secret101"
```

```
  - type: "Generic"
    generic:
      secret: "secret101"
  -
    type: "ImageChange"
source: 4
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
strategy: 5
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
output: 6
  to:
    kind: "ImageStreamTag"
    name: "origin-ruby-sample:latest"
postCommit: 7
    script: "bundle exec rake test"
```

1. 此规格会创建一个名为 **ruby-sample-build** 的新 **BuildConfig**。
2. **runPolicy** 字段控制从此构建配置创建的构建能否同时运行。默认值为 **Serial**，即新构建将按顺序运行，而不是同时运行。
3. 您可以指定导致创建新构建的触发器的列表。
4. **source** 部分定义构建的来源。源类型决定主要的输入源，可以是 **Git**（指向代码库存储位置）、**Dockerfile**（从内联 Dockerfile 构建）或 **Binary**（接受二进制有效负载）。可以同时拥有多个源。详情请参阅每种源类型的文档。
5. **strategy** 部分描述用于执行构建的构建策略。您可以在此处指定 **Source**、**Docker** 或 **Custom** 策略。本例使用 **ruby-20-centos7** 容器镜像， Source-to-image（S2I）用于应用程序构建。
6. 成功构建容器镜像后，它将被推送到 **output** 部分中描述的存储库。
7. **postCommit** 部分定义一个可选构建 hook。

2.3. 创建构建输入

通过以下小节查看构建输入的概述，并了解如何使用输入提供构建操作的源内容，以及如何使用构建环境和创建 secret。

2.3.1. 构建输入

构建输入提供构建操作的源内容。您可以使用以下构建输入在 OpenShift Container Platform 中提供源，它们按优先顺序列出：

- 内联 Dockerfile 定义
- 从现有镜像中提取内容
- Git 存储库

- 二进制（本地）输入
- 输入 secret
- 外部工件 (artifact)

您可以在单个构建中组合多个输入。但是，由于内联 Dockerfile 具有优先权，它可能会覆盖任何由其他输入提供的名为 Dockerfile 的文件。二进制（本地）和 Git 存储库是互斥的输入。

如果不希望在构建生成的最终应用程序镜像中提供构建期间使用的某些资源或凭证，或者想要消耗在 secret 资源中定义的值，您可以使用输入 secret。外部工件可用于拉取不以其他任一构建输入类型提供的额外文件。

在运行构建时：

1. 构造工作目录，并将所有输入内容放进工作目录中。例如，把输入 Git 存储库克隆到工作目录中，并且把由输入镜像指定的文件通过目标目录复制到工作目录中。
2. 构建过程将目录更改到 **contextDir**（若已指定）。
3. 内联 Dockerfile（若有）写入当前目录中。
4. 当前目录中的内容提供给构建过程，供 Dockerfile、自定义构建器逻辑或 **assemble** 脚本引用。这意味着，构建会忽略所有驻留在 **contextDir** 之外的输入内容。

以下源定义示例包括多种输入类型，以及它们如何组合的说明。如需有关如何定义各种输入类型的更多详细信息，请参阅每种输入类型的具体小节。

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git 1
    ref: "master"
  images:
  - from:
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths:
    - destinationDir: app/dir/injected/dir 2
      sourcePath: /usr/lib/somefile.jar
  contextDir: "app/dir" 3
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" 4
```

1. 要克隆到构建的工作目录中的存储库。
2. 来自 **myinputimage** 的 **/usr/lib/somefile.jar** 存储在 **<workingdir> /app/dir/injected/dir** 中。
3. 构建的工作目录将变为 **<original_workingdir>/app/dir**。
4. **<original_workingdir>/app/dir** 中创建了含有此内容的 Dockerfile，并覆盖具有该名称的任何现有文件。

2.3.2. Dockerfile 源

提供 **dockerfile** 值时，此字段的内容将写到磁盘上，存为名为 **Dockerfile** 的文件。这是处理完其他输入源之后完成的；因此，如果输入源存储库的根目录中包含 Dockerfile，它会被此内容覆盖。

源定义是 **BuildConfig** 的 **spec** 部分的一部分：

```
source:
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" (1)
```

(1) **dockerfile** 字段包含要构建的内联 Dockerfile。

其他资源

- 此字段的典型用途是为 Docker 策略构建提供 Dockerfile。

2.3.3. 镜像源

您可以使用镜像向构建过程添加其他文件。输入镜像的引用方式与定义 **From** 和 **To** 镜像目标的方式相同。这意味着可以引用容器镜像和镜像流标签。在使用镜像时，必须提供一个或多个路径对，以指示要复制镜像的文件或目录的路径以及构建上下文中要放置它们的目的地。

源路径可以是指定镜像内的任何绝对路径。目的地必须是相对目录路径。构建时会加载镜像，并将指定的文件和目录复制到构建过程上下文目录中。这与源存储库内容要克隆到的目录相同。如果源路径以 **/.** 结尾，则复制目录的内容，但不在目的地上创建该目录本身。

镜像输入在 **BuildConfig** 的 **source** 定义中指定：

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
    ref: "master"
  images: (1)
  - from: (2)
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths: (3)
    - destinationDir: injected/dir (4)
      sourcePath: /usr/lib/somefile.jar (5)
  - from:
      kind: ImageStreamTag
      name: myotherinputimage:latest
      namespace: myothernamespace
    pullSecret: mysecret (6)
    paths:
    - destinationDir: injected/dir
      sourcePath: /usr/lib/somefile.jar
```

(1) 由一个或多个输入镜像和文件组成的数组。

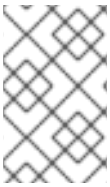
(2) 对包含要复制的文件的镜像的引用。

(3) 源/目标路径的数组。

4 相对于构建过程能够处理文件的构建根目录的目录。

5 要从所引用镜像中复制文件的位置。

6 提供的可选 secret，如需要凭证才能访问输入镜像。



注意

如果您的集群使用 **ImageDigestMirrorSet**、**ImageTagMirrorSet** 或 **ImageContentSourcePolicy** 对象来配置存储库镜像，则只能使用镜像的 registry 的全局 pull secret。您不能在项目中添加 pull secret。

需要 pull secret 的镜像

如果输入镜像需要 pull secret，您可以将 pull secret 链接到构建所使用的服务帐户。默认情况下，构建使用 **builder** 服务帐户。如果 secret 包含与托管输入镜像的存储库匹配的凭证，pull secret 会自动添加到构建中。要将 pull secret 连接到构建使用的服务帐户，请运行：

```
$ oc secrets link builder dockerhub
```

注意

使用自定义策略的构建不支持此功能。

位于 mirrorr registry 中的需要 pull secret 的镜像

当使用 mirrorr registry 中的输入镜像时，如果出现 **build error: failed to pull image** 信息，您可以使用以下方法之一解决这个错误：

- 创建一个输入 secret，其中包含构建器镜像的仓库和所有已知 mirror 系统的身份验证凭据。在本例中，为到镜像 registry 及其 mirror 的凭证创建一个 pull secret。
- 使用输入 secret 作为 **BuildConfig** 对象上的 pull secret。

2.3.4. Git 源

指定之后，从提供的位置获取源代码。

如果您提供内联 Dockerfile，它将覆盖 Git 存储库的 **contextDir** 中的 Dockerfile。

源定义是 **BuildConfig** 的 **spec** 部分的一部分：

```
source:
  git: 1
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
  contextDir: "app/dir" 2
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example" 3
```

1 **git** 字段包含源代码的远程 Git 存储库的 URI (Uniform Resource Identifier)。您必须指定 **ref** 字段的值来签出特定的 Git 引用。有效的 **ref** 可以是 SHA1 标签或分支名称。 **ref** 字段的默认值为 **master**。

2

contextDir 字段允许您覆盖源代码存储库中构建查找应用程序源代码的默认位置。如果应用程序位于子目录中，您可以使用此字段覆盖默认位置（根文件夹）。

3 如果提供可选的 **dockerfile** 字段，它应该是包含 Dockerfile 的字符串，此文件将覆盖源存储库中可能存在的任何 Dockerfile。

如果 **ref** 字段注明拉取请求，则系统将使用 **git fetch** 操作，然后 checkout **FETCH_HEAD**。

如果未提供 **ref** 值，OpenShift Container Platform 将执行浅克隆（**--depth=1**）。这时，仅下载与默认分支（通常为 **master**）上最近提交相关联的文件。这将使存储库下载速度加快，但不会有完整的提交历史记录。要执行指定存储库的默认分支的完整 **git clone**，请将 **ref** 设置为默认分支的名称（如 **main**）。

警告

如果 Git 克隆操作要经过执行中间人 (MITM) TLS 劫持或重新加密被代理连接的代理，该操作不起作用。

2.3.4.1. 使用代理

如果 Git 存储库只能使用代理访问，您可以在构建配置的 **source** 部分中定义要使用的代理。您可以同时配置要使用的 HTTP 和 HTTPS 代理。两个字段都是可选的。也可以在 **NoProxy** 字段中指定不应执行代理的域。

注意

源 URI 必须使用 HTTP 或 HTTPS 协议才可以正常工作。

```
source:
  git:
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
    httpProxy: http://proxy.example.com
    httpsProxy: https://proxy.example.com
    noProxy: somedomain.com, otherdomain.com
```

注意

对于 Pipeline 策略构建，因为 Jenkins Git 插件当前限制的缘故，通过 Git 插件执行的任何 Git 操作都不会利用 **BuildConfig** 中定义的 HTTP 或 HTTPS 代理。Git 插件将仅使用 Plugin Manager 面板上 Jenkins UI 中配置的代理。然后，在所有任务中，此代理都会被用于 Jenkins 内部与 git 的所有交互。

其他资源

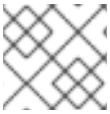
- 您可以在 JenkinsBehindProxy 上找到有关如何通过 Jenkins UI 配置代理的说明。

2.3.4.2. 源克隆 secret

构建器 pod 需要访问定义为构建源的任何 Git 存储库。源克隆 secret 为构建器 pod 提供了通常无权访问的资源的访问权限，例如私有存储库或具有自签名或不可信 SSL 证书的存储库。

支持以下源克隆 secret 配置：

- **.gitconfig** 文件
- 基本身份验证（Basic authentication）
- SSH 密钥身份验证
- 可信证书颁发机构



注意

您还可以组合使用这些配置来满足特定的需求。

2.3.4.2.1. 自动把源克隆 secret 添加到构建配置

创建 **BuildConfig**，OpenShift Container Platform 可以自动填充其源克隆 secret 引用。这会使生成的构建自动使用存储在引用的 secret 中的凭证与远程 Git 存储库进行身份验证，而无需进一步配置。

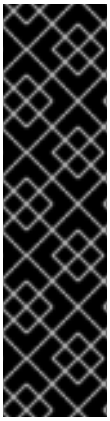
若要使用此功能，包含 Git 存储库凭证的一个 secret 必须存在于稍后创建 **BuildConfig** 的命名空间中。此 secret 必须包含前缀为 **build.openshift.io/source-secret-match-uri-** 的一个或多个注解。这些注解中的每一个值都是统一资源标识符（URI）模式，其定义如下。如果 **BuildConfig** 是在没有源克隆 secret 引用的前提下创建的，并且其 Git 源 URI 与 secret 注解中的 URI 模式匹配，OpenShift Container Platform 将自动在 **BuildConfig** 插入对该 secret 的引用。

先决条件

URI 模式必须包含：

- 有效的方案包括： ***://**、**git://**、**http://**、**https://** 或 **ssh://**
- 一个主机：*,或一个有效的主机名或 IP 地址（可选）在之前使用 ***。**
- 一个路径： **/***，或 **/**（后面包括任意字符并可以包括 * 字符）

在上述所有内容中，* 字符被认为是通配符。



重要

URI 模式必须与符合 RFC3986 的 Git 源 URI 匹配。不要在 URI 模式中包含用户名（或密码）组件。

例如，如果使用 **ssh://git@bitbucket.atlassian.com:7999/ATLASSIAN jira.git** 作为 git 存储库 URL，则源 secret 必须指定为 **ssh://bitbucket.atlassian.com:7999/***（而非 **ssh://git@bitbucket.atlassian.com:7999/***）。

```
$ oc annotate secret mysecret \
    'build.openshift.io/source-secret-match-uri-1=ssh://bitbucket.atlassian.com:7999/*'
```

流程

如果多个 secret 与特定 **BuildConfig** 的 Git URI 匹配，OpenShift Container Platform 会选择匹配最多的 secret。这可以实现下例中所示的基本覆盖。

以下片段显示了两个部分源克隆 secret，第一个匹配通过 HTTPS 访问的 **mycorp.com** 域中的任意服务器，第二个则覆盖对服务器 **mydev1.mycorp.com** 和 **mydev2.mycorp.com** 的访问：

```
kind: Secret
apiVersion: v1
metadata:
  name: matches-all-corporate-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://*.mycorp.com/*
data:
  ...
---
kind: Secret
apiVersion: v1
metadata:
  name: override-for-my-dev-servers-https-only
  annotations:
    build.openshift.io/source-secret-match-uri-1: https://mydev1.mycorp.com/*
    build.openshift.io/source-secret-match-uri-2: https://mydev2.mycorp.com/*
data:
  ...
```

- 使用以下命令将 **build.openshift.io/source-secret-match-uri-** 注解添加到预先存在的 secret：

  ```
  $ oc annotate secret mysecret \
      'build.openshift.io/source-secret-match-uri-1=https://*.mycorp.com/*'
  ```

2.3.4.2.2. 手动添加源克隆 secret

通过将 **sourceSecret** 字段添加到 **BuildConfig** 中的 **source** 部分，并将它设置为您创建的 secret 的名称，可以手动将源克隆 secret 添加到构建配置中。在本例中，是 **basicsecret**。

```
apiVersion: "build.openshift.io/v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
  source:
    git:
      uri: "https://github.com/user/app.git"
    sourceSecret:
      name: "basicsecret"
  strategy:
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "python-33-centos7:latest"
```

流程

您还可以使用 **oc set build-secret** 命令在现有构建配置中设置源克隆 secret。

- 要在现有构建配置上设置源克隆 secret，请输入以下命令：

```
$ oc set build-secret --source bc/sample-build basicsecret
```

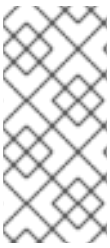
2.3.4.2.3. 从 .gitconfig 文件创建 secret

如果克隆应用程序要依赖于 **.gitconfig** 文件，您可以创建包含它的 secret。将它添加到 builder 服务帐户中，再添加到您的 **BuildConfig** 。

流程

- 从 **.gitconfig** 文件创建 secret：

```
$ oc create secret generic <secret_name> --from-file=<path/to/.gitconfig>
```



注意

如果 **.gitconfig** 文件的 **http** 部分设置了 **sslVerify=false**，则可以关闭 iVSSL 验证：

```
[http]
        sslVerify=false
```

2.3.4.2.4. 从 .gitconfig 文件为安全 Git 创建 secret

如果 Git 服务器使用双向 SSL 和用户名加密码进行保护，您必须将证书文件添加到源构建中，并在 **.gitconfig** 文件中添加对证书文件的引用。

先决条件

- 您必须具有 Git 凭证。

流程

将证书文件添加到源构建中，并在 **gitconfig** 文件中添加对证书文件的引用。

1. 将 **client.crt**、**cacert.crt** 和 **client.key** 文件添加到应用程序源代码的 **/var/run/secrets/openshift.io/source/** 目录中。
2. 在服务器的 **.gitconfig** 文件中，添加下例中所示的 **[http]** 部分：

```
# cat .gitconfig
```

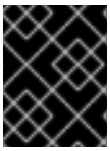
输出示例

```
[user]
        name = <name>
        email = <email>
[http]
        sslVerify = false
        sslCert = /var/run/secrets/openshift.io/source/client.crt
        sslKey = /var/run/secrets/openshift.io/source/client.key
        sslCaInfo = /var/run/secrets/openshift.io/source/cacert.crt
```

3. 创建 secret：

```
$ oc create secret generic <secret_name> \
--from-literal=username=<user_name> \ 1
--from-literal=password=<password> \ 2
--from-file=.gitconfig=.gitconfig \
--from-file=client.crt=/var/run/secrets/openshift.io/source/client.crt \
--from-file=cacert.crt=/var/run/secrets/openshift.io/source/cacert.crt \
--from-file=client.key=/var/run/secrets/openshift.io/source/client.key
```

1. 用户的 Git 用户名。
2. 此用户的密码。



重要

为了避免必须再次输入密码，需要在构建中指定 source-to-image（S2I）镜像。但是，如果无法克隆存储库，您仍然必须指定用户名和密码才能推进构建。

其他资源

- 应用程序源代码中的 **/var/run/secrets/openshift.io/source/** 文件夹。

2.3.4.2.5. 从源代码基本身份验证创建 secret

基本身份验证需要 **--username** 和 **--password** 的组合，或者令牌方可与软件配置管理（SCM）服务器进行身份验证。

先决条件

- 用于访问私有存储库的用户名和密码。

流程

1. 在使用 **--username** 和 **--password** 访问私有存储库前首先创建 secret:

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --type=kubernetes.io/basic-auth
```

2. 使用令牌创建基本身份验证 secret：

```
$ oc create secret generic <secret_name> \
    --from-literal=password=<token> \
    --type=kubernetes.io/basic-auth
```

2.3.4.2.6. 从源代码 SSH 密钥身份验证创建 secret

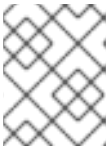
基于 SSH 密钥的身份验证需要 SSH 私钥。

存储库密钥通常位于 **$HOME/.ssh/** 目录中，但默认名称为 **id_dsa.pub**、**id_ecdsa.pub**、**id_ed25519.pub** 或 **id_rsa.pub**。

流程

1. 生成 SSH 密钥凭证：

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```



注意

使用带有密语保护的 SSH 密钥会导致 OpenShift Container Platform 无法进行构建。提示输入密语（passphrase）时，请将其留空。

创建两个文件：公钥和对应的私钥（**id_dsa**、**id_ecdsa**、**id_ed25519** 或 **id_rsa** 之一）。这两项就位后，请查阅源代码控制管理 (SCM) 系统的手册来了解如何上传公钥。私钥用于访问您的私有存储库。

2. 在使用 SSH 密钥访问私有存储库之前，先创建 secret：

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/known_hosts> \ 1
    --type=kubernetes.io/ssh-auth
```

1 可选：添加此字段可启用严格的服务器主机密钥检查。

警告

在创建 secret 时跳过 **known_hosts** 文件会使构建容易受到中间人 (MITM) 攻击的影响。

注意

确保 **known_hosts** 文件中包含源代码主机条目。

2.3.4.2.7. 从源代码可信证书颁发机构创建 secret

在 Git 克隆操作期间受信任的 TLS 证书颁发机构（CA）集合内置于 OpenShift Container Platform 基础结构镜像中。如果 Git 服务器使用自签名证书或由镜像不信任的颁发机构签名的证书，您可以创建包含证书的 secret 或者禁用 TLS 验证。

如果您为 CA 证书创建 secret，OpenShift Container Platform 会在 Git 克隆操作过程中使用它来访问您的 Git 服务器。使用此方法比禁用 Git 的 SSL 验证要安全得多，后者接受所出示的任何 TLS 证书。

流程

使用 CA 证书文件创建 secret。

1. 如果您的 CA 使用中间证书颁发机构，请合并 **ca.crt** 文件中所有 CA 的证书。输入以下命令：

```
$ cat intermediateCA.crt intermediateCA.crt rootCA.crt > ca.crt
```

2. 运行以下命令来创建 secret：

```
$ oc create secret generic mycert --from-file=ca.crt=</path/to/file> 1
```

1 您必须使用密钥名称 **ca.crt**。

2.3.4.2.8. 源 secret 组合

您可以组合使用不同的源克隆 secret 创建方法来满足特定的需求。

2.3.4.2.8.1. 使用 .gitconfig 文件创建基于 SSH 的身份验证 secret

您可以组合不同的方法开创建源克隆 secret 以满足特定的需求，例如使用 **.gitconfig** 文件的基于 SSH 的身份验证 secret。

先决条件

- SSH 身份验证
- **.gitconfig** 文件

流程

- 要使用 **.gitconfig** 文件创建基于 SSH 的身份验证 secret，请输入以下命令：

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/.gitconfig> \
    --type=kubernetes.io/ssh-auth
```

2.3.4.2.8.2. 创建组合了 .gitconfig 文件和 CA 证书的 secret

您可以组合使用不同的源克隆 secret 创建方法来满足特定的需求，例如组合了 **.gitconfig** 文件和 CA 证书的 Secret。

先决条件

- **.gitconfig** 文件
- CA 证书

流程

- 要创建组合了 **.gitconfig** 文件和 CA 证书的 secret，请输入以下命令：

```
$ oc create secret generic <secret_name> \
    --from-file=ca.crt=<path/to/certificate> \
    --from-file=<path/to/.gitconfig>
```

2.3.4.2.8.3. 使用 CA 证书创建基本身份验证 secret

您可以组合使用不同的源克隆 secret 创建方法来满足特定的需求，例如组合了基本身份验证和 CA 证书的 secret。

先决条件

- 基本身份验证凭证
- CA 证书

流程

- 要使用 CA 证书创建基本身份验证 secret，请输入以下命令：

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=ca-cert=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

2.3.4.2.8.4. 使用 Git 配置文件创建基本身份验证 secret

您可以组合使用不同的源克隆 secret 创建方法来满足特定的需求，例如组合了基本身份验证和 **.gitconfig** 文件的 secret。

先决条件

- 基本身份验证凭证
- **.gitconfig** 文件

流程

- 要使用 **.gitconfig** 文件创建基本身份验证 secret，请输入以下命令：

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=</path/to/.gitconfig> \
    --type=kubernetes.io/basic-auth
```

2.3.4.2.8.5. 使用 .gitconfig 文件和 CA 证书创建基本身份验证 secret

您可以组合使用不同的源克隆 secret 创建方法来满足特定的需求，例如组合了基本身份验证、**.gitconfig** 文件和证书颁发机构（CA）证书的 Secret。

先决条件

- 基本身份验证凭证
- **.gitconfig** 文件
- CA 证书

流程

- 要使用 **.gitconfig** 文件和 CA 证书创建基本身份验证 secret，请输入以下命令：

```
$ oc create secret generic <secret_name> \
    --from-literal=username=<user_name> \
    --from-literal=password=<password> \
    --from-file=</path/to/.gitconfig> \
    --from-file=ca-cert=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

2.3.5. 二进制（本地）来源

从本地文件系统流传输内容到构建器称为 **Binary** 类型构建。对于此类构建，**BuildConfig.spec.source.type** 的对应值为 **Binary**。

这种源类型的独特之处在于，它仅基于您对 **oc start-build** 的使用而加以利用。

注意

二进制类型构建需要从本地文件系统流传输内容，因此无法自动触发二进制类型构建，如镜像更改触发器。这是因为无法提供二进制文件。同样，您无法从 web 控制台启动二进制类型构建。

要使用二进制构建，请使用以下选项之一调用 **oc start-build**：

- **--from-file**：指定的文件内容作为二进制流发送到构建器。您还可以指定文件的 URL。然后，构建器将数据存储在构建上下文顶端的同名文件中。
- **--from-dir** 和 **--from-repo**：内容存档下来，并作为二进制流发送给构建器。然后，构建器在构建上下文目录中提取存档的内容。使用 **--from-dir** 时，您还可以指定提取的存档的 URL。
- **--from-archive**：指定的存档发送到构建器，并在构建器上下文目录中提取。此选项与 **--from-dir** 的行为相同；只要这些选项的参数是目录，就会首先在主机上创建存档。

在上方列出的每种情形中：

- 如果 **BuildConfig** 已经定义了 **Binary** 源类型，它会有效地被忽略并且替换成客户端发送的内容。
- 如果 **BuildConfig** 定义了 **Git** 源类型，则会动态禁用它，因为 **Binary** 和 **Git** 是互斥的，并且二进制流中提供给构建器的数据将具有优先权。

您可以将 HTTP 或 HTTPS 方案的 URL 传递给 **--from-file** 和 **--from-archive**，而不传递文件名。将 **-from-file** 与 URL 结合使用时，构建器镜像中文件的名称由 web 服务器发送的 **Content-Disposition** 标头决定，如果该标头不存在，则由 URL 路径的最后一个组件决定。不支持任何形式的身份验证，也无法使用自定义 TLS 证书或禁用证书验证。

使用 **oc new-build --binary =true** 时，该命令可确保强制执行与二进制构建关联的限制。生成的 **BuildConfig** 具有 **Binary** 源类型，这意味着为此 **BuildConfig** 运行构建的唯一有效方法是使用 **oc start-build** 和其中一个 **--from** 选项来提供必需的二进制数据。

Dockerfile 和 **contextDir** 源选项对二进制构建具有特殊含义。

Dockerfile 可以与任何二进制构建源一起使用。如果使用 Dockerfile 且二进制流是存档，则其内容将充当存档中任何 Dockerfile 的替代 Dockerfile。如果将 Dockerfile 与 **--from-file** 参数搭配 使用，且文件参数命名为 Dockerfile，则 Dockerfile 的值将替换二进制流中的值。

如果是二进制流封装提取的存档内容，**contextDir** 字段的值将解释为存档中的子目录，并且在有效时，构建器将在执行构建之前更改到该子目录。

2.3.6. 输入 secret 和配置映射

重要

要防止输入 secret 和配置映射的内容出现在构建输出容器镜像中，请使用 Docker 构建和源至镜像构建策略中的构建 卷。

有时候，构建操作需要凭证或其他配置数据才能访问依赖的资源，但又不希望将这些信息放在源代码控制中。您可以定义输入 secret 和输入配置映射。

例如，在使用 Maven 构建 Java 应用程序时，可以设置通过私钥访问的 Maven Central 或 JCenter 的私有镜像。要从该私有镜像下载库，您必须提供以下内容：

1. 配置了镜像的 URL 和连接设置的 **settings.xml** 文件。
2. 设置文件中引用的私钥，例如 ~/**.ssh/id_rsa**。

为安全起见，不应在应用程序镜像中公开您的凭证。

示例中描述的是 Java 应用程序，但您可以使用相同的方法将 SSL 证书添加到 **/etc/ssl/certs** 目录，以及添加 API 密钥或令牌、许可证文件等。

2.3.6.1. 什么是 secret？

Secret 对象类型提供了一种机制来保存敏感信息，如密码、OpenShift Container Platform 客户端配置文件、**dockercfg** 文件和私有源存储库凭证等。secret 将敏感内容与 Pod 分离。您可以使用卷插件将 secret 信息挂载到容器中，系统也可以使用 secret 代表 Pod 执行操作。

YAML Secret 对象定义

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque (1)
data: (2)
  username: <username> (3)
  password: <password>
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

(1) 指示 secret 的键和值的结构。

(2) **data** 字段中允许的键格式必须符合 Kubernetes 标识符术语表中 **DNS_SUBDOMAIN** 值的规范。

(3) 与 **data** 映射中键关联的值必须采用 base64 编码。

4 **stringData** 映射中的条目将转换为 base64，然后该条目将自动移动到 **data** 映射中。此字段是只读的。这个值只能由 **data** 字段返回。

5 与 **stringData** 映射中键关联的值由纯文本字符串组成。

2.3.6.1.1. secret 的属性

主要属性包括：

- Secret 数据可以独立于其定义来引用。
- Secret 数据卷由临时文件工具 (tmpfs) 支持，永远不会停留在节点上。
- secret 数据可以在命名空间内共享。

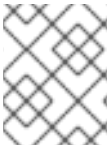
2.3.6.1.2. secret 的类型

type 字段中的值指明 secret 的键名称和值的结构。此类型可用于强制使 secret 对象中存在用户名和密钥。如果您不想进行验证，请使用 **opaque** 类型，这也是默认类型。

指定以下一种类型来触发最小服务器端验证，确保 secret 数据中存在特定的键名称：

- **kubernetes.io/service-account-token**。使用服务帐户令牌。
- **kubernetes.io/dockercfg**。将 **.dockercfg** 文件用于所需的 Docker 凭证。
- **kubernetes.io/dockerconfigjson**。将 **.docker/config.json** 文件用于所需的 Docker 凭证。
- **kubernetes.io/basic-auth**。与基本身份验证搭配使用。
- **kubernetes.io/ssh-auth**。搭配 SSH 密钥身份验证使用。
- **kubernetes.io/tls**。搭配 TLS 证书颁发机构使用。

如果不想进行验证，设置 **type= Opaque**。这意味着，secret 不声明符合键名称或值的任何约定。**opaque** secret 允许使用无结构 **key:value** 对，可以包含任意值。



注意

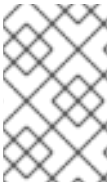
您可以指定其他任意类型，如 **example.com/my-secret-type**。这些类型不在服务器端强制执行，但代表 secret 的创建者意在符合该类型的键/值要求。

2.3.6.1.3. 更新 secret

当修改 secret 的值时，已在被运行的 Pod 使用的 secret 值不会被动态更新。要更改 secret，必须删除原始 pod 并创建一个新 pod，在某些情况下，具有相同的 **PodSpec**。

更新 secret 遵循与部署新容器镜像相同的工作流。您可以使用 **kubectl rolling-update** 命令。

secret 中的 **resourceVersion** 值不在引用时指定。因此，如果在 pod 启动的同时更新 secret，则将不能定义用于 pod 的 secret 版本。



注意

目前，无法检查 Pod 创建时使用的 secret 对象的资源版本。按照计划 Pod 将报告此信息，以便控制器可以重启使用旧 **resourceVersion** 的 Pod。在此期间，请勿更新现有 secret 的数据，而应创建具有不同名称的新数据。

2.3.6.2. 创建 secret

您必须先创建 secret，然后创建依赖于此 secret 的 Pod。

在创建 secret 时：

- 使用 secret 数据创建 secret 对象。
- 更新 pod 的服务帐户以允许引用该 secret。
- 创建以环境变量或文件（使用 **secret** 卷）形式消耗 secret 的 pod。

流程

- 要从 JSON 或 YAML 文件创建 secret 对象，请输入以下命令：

```
$ oc create -f <filename>
```

例如，您可以从本地的 **.docker/config.json** 文件创建一个 secret：

```
$ oc create secret generic dockerhub \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

此命令将生成名为 **dockerhub** 的 secret JSON 规格并创建该对象。

YAML Opaque Secret 对象定义

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
data:
  username: <username>
  password: <password>
```

1 指定一个 *opaque* secret。

Docker 配置 JSON 文件对象定义

```
apiVersion: v1
kind: Secret
metadata:
  name: aregistrykey
  namespace: myapps
type: kubernetes.io/dockerconfigjson 1
```

```
data:

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXIzCg== 2
```

1 指定该 secret 使用 docker 配置 JSON 文件。

2 base64 编码的 Docker 配置 JSON 文件

2.3.6.3. 使用 secret

创建 secret 后，可以创建一个 Pod 来引用您的 secret，再获取日志，然后删除 Pod。

流程

1. 输入以下命令创建 pod 来引用您的 secret：

```
$ oc create -f <your_yaml_file>.yaml
```

2. 输入以下命令来获取日志：

```
$ oc logs secret-example-pod
```

3. 输入以下命令删除 pod：

```
$ oc delete pod secret-example-pod
```

其他资源

- 带有 secret 数据的 YAML 文件示例：

  将创建四个文件的 secret 的 YAML 文件

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: <username> 1
  password: <password> 2
stringData:
  hostname: myapp.mydomain.com 3
  secret.properties: |- 4
    property1=valueA
    property2=valueB
```

1 文件包含已解码的值。

2 文件包含已解码的值。

3 文件包含提供的字符串。

4 文件包含提供的数据。

pod 的 YAML 文件使用 secret 数据填充卷中的文件

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
          # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/secret-volume
            readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
  restartPolicy: Never
```

pod 的 YAML 文件使用 secret 数据填充环境变量

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
  restartPolicy: Never
```

BuildConfig 对象的 YAML 文件，用于在环境变量中填充 secret 数据

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
```

```
      valueFrom:
        secretKeyRef:
          name: test-secret
          key: username
```

2.3.6.4. 添加输入 secret 和配置映射

要向构建提供凭证和其他配置数据，而不将其放在源控制中，您可以定义输入 secret 和输入配置映射。

在某些情况下，构建操作需要凭证或其他配置数据才能访问依赖的资源。要使该信息在不置于源控制中的情况下可用，您可以定义输入 secret 和输入配置映射。

流程

将输入 secret 和配置映射添加到现有的 **BuildConfig** 对象中：

1. 如果 **ConfigMap** 对象不存在，请输入以下命令来创建它：

```
$ oc create configmap settings-mvn \
    --from-file=settings.xml=<path/to/settings.xml>
```

这会创建一个名为 **settings-mvn** 的新配置映射，其中包含 **settings.xml** 文件的纯文本内容。

提示

您还可以应用以下 YAML 来创建配置映射：

```
apiVersion: core/v1
kind: ConfigMap
metadata:
  name: settings-mvn
data:
  settings.xml: |
    <settings>
    … # Insert maven settings here
    </settings>
```

2. 如果 **Secret** 对象不存在，请输入以下命令来创建它：

```
$ oc create secret generic secret-mvn \
    --from-file=ssh-privatekey=<path/to/.ssh/id_rsa> \
    --type=kubernetes.io/ssh-auth
```

这会创建一个名为 **secret-mvn** 的新 secret，其包含 **id_rsa** 私钥的 base64 编码内容。

提示

您还可以应用以下 YAML 来创建输入 secret：

```
apiVersion: core/v1
kind: Secret
metadata:
  name: secret-mvn
type: kubernetes.io/ssh-auth
data:
  ssh-privatekey: |
    # Insert ssh private key, base64 encoded
```

3. 将配置映射和 secret 添加到现有 **BuildConfig** 对象的 **source** 部分中：

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
  secrets:
    - secret:
        name: secret-mvn
```

4. 要在新 **BuildConfig** 对象中包含 secret 和配置映射，请输入以下命令：

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn” \
    --build-config-map "settings-mvn"
```

在构建期间，构建过程将 **settings.xml** 和 **id_rsa** 文件复制到源代码所在的目录中。在 OpenShift Container Platform S2I 构建器镜像中，这是镜像的工作目录，使用 **Dockerfile** 中的 **WORKDIR** 指令设置。如果要指定其他目录，请在定义中添加 **destinationDir**：

```
source:
  git:
    uri: https://github.com/wildfly/quickstart.git
  contextDir: helloworld
  configMaps:
    - configMap:
        name: settings-mvn
      destinationDir: ".m2"
  secrets:
    - secret:
        name: secret-mvn
      destinationDir: ".ssh"
```

您还可以输入以下命令在创建新 **BuildConfig** 对象时指定目标目录：

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
```

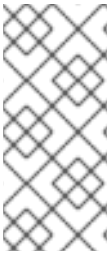
```
--context-dir helloworld --build-secret “secret-mvn:.ssh” \
--build-config-map "settings-mvn:.m2"
```

在这两种情况下，**settings.xml** 文件都添加到构建环境的 **./.m2** 目录中，而 **id_rsa** 密钥则添加到 **./.ssh**目录中。

2.3.6.5. Source-to-Image 策略

采用 **Source** 策略时，所有定义的输入 secret 都复制到对应的 **destinationDir** 中。如果 **destinationDir** 留空，则 secret 会放置到构建器镜像的工作目录中。

当 **destinationDir** 是一个相对路径时，使用相同的规则。secret 放置在相对于镜像工作目录的路径中。如果构建器镜像中不存在 **destinationDir** 路径中的最终目录，则会创建该目录。**destinationDir** 中的所有上述目录都必须存在，否则会发生错误。



注意

输入 secret 将以全局可写（具有 **0666** 权限）形式添加，并且在执行 **assemble** 脚本后其大小会被截断为零。也就是说，生成的镜像中会包括这些 secret 文件，但出于安全原因，它们将为空。

assemble 脚本完成后不会截断输入配置映射。

2.3.6.6. Docker 策略

采用 docker 策略时，您可以使用 Dockerfile 中的 **ADD** 和 **COPY** 指令，将所有定义的输入 secret 添加到容器镜像中。

如果没有为 secret 指定 **destinationDir**，则文件将复制到 Dockerfile 所在的同一目录中。如果将一个相对路径指定为 **destinationDir**，则 secret 复制到相对于 Dockerfile 位置的该目录中。这样，secret 文件可供 Docker 构建操作使用，作为构建期间使用的上下文目录的一部分。

引用 secret 和配置映射数据的 Dockerfile 示例

```
FROM centos/ruby-22-centos7

USER root
COPY ./secret-dir /secrets
COPY ./config /

# Create a shell script that will output secrets and ConfigMaps when the image is run
RUN echo '#!/bin/sh' > /input_report.sh
RUN echo '(test -f /secrets/secret1 && echo -n "secret1=" && cat /secrets/secret1)' >>
/input_report.sh
RUN echo '(test -f /config && echo -n "relative-configMap=" && cat /config)' >> /input_report.sh
RUN chmod 755 /input_report.sh

CMD ["/bin/sh", "-c", "/input_report.sh"]
```

重要

用户应该从最终的应用程序镜像中移除输入 secret，以便从该镜像运行的容器中不会存在这些 secret。但是，secret 仍然存在于它们添加到的层中的镜像本身内。这一移除是 Dockerfile 本身的一部分。

为防止输入 secret 和配置映射的内容出现在构建输出容器镜像中并完全避免此移除过程，请在 Docker 构建策略中使用构建卷。

2.3.6.7. Custom 策略

使用 Custom 策略时，所有定义的输入 secret 和配置映射都位于 **/var/run/secrets/openshift.io/build** 目录中的构建器容器中。自定义构建镜像必须正确使用这些 secret 和配置映射。使用 Custom 策略时，您可以按照 Custom 策略选项中所述定义 secret。

现有策略 secret 与输入 secret 之间没有技术差异。但是，构建器镜像可以区分它们并以不同的方式加以使用，具体取决于您的构建用例。

输入 secret 始终挂载到 **/var/run/secrets/openshift.io/build** 目录中，或您的构建器可以解析 **$BUILD** 环境变量（包含完整构建对象）。

重要

如果命名空间和节点上存在 registry 的 pull secret，构建会默认使用命名空间中的 pull secret。

2.3.7. 外部工件 (artifact)

建议不要将二进制文件存储在源存储库中。因此，您必须定义一个构建，在构建过程中拉取其他文件，如 Java **.jar** 依赖项。具体方法取决于使用的构建策略。

对于 Source 构建策略，必须在 **assemble** 脚本中放入适当的 shell 命令：

.s2i/bin/assemble 文件

```
#!/bin/sh
APP_VERSION=1.0
wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

.s2i/bin/run 文件

```
#!/bin/sh
exec java -jar app.jar
```

对于 Docker 构建策略，您必须修改 Dockerfile 并通过 **RUN** 指令 调用 shell 命令：

Dockerfile 摘录

```
FROM jboss/base-jdk:8

ENV APP_VERSION 1.0
RUN wget http://repository.example.com/app/app-$APP_VERSION.jar -O app.jar
```

```
EXPOSE 8080
CMD [ "java", "-jar", "app.jar" ]
```

在实践中，您可能希望将环境变量用于文件位置，以便要下载的具体文件能够使用 **BuildConfig** 中定义的环境变量来自定义，而不必更新 Dockerfile 或 **assemble** 脚本。

您可以选择不同方法来定义环境变量：

- 使用 **.s2i/environment** 文件（仅适用于 **Source** 构建策略）
- 在 **BuildConfig** 对象中设置变量
- 使用 **oc start-build --env** 命令显式提供变量（仅适用于手动触发的构建）

2.3.8. 将 docker 凭证用于私有容器镜像仓库

您可以为构建提供 **.docker/config.json** 文件，在文件中包含私有容器 registry 的有效凭证。这样，您可以将输出镜像推送到私有容器镜像 registry 中，或从需要身份验证的私有容器镜像 registry 中拉取构建器镜像。

您可以为同一 registry 中的多个存储库提供凭证，每个软件仓库都有特定于该 registry 路径的凭证。

注意

对于 OpenShift Container Platform 容器镜像 registry，这不是必需的，因为 OpenShift Container Platform 会自动为您生成 secret。

默认情况下，**.docker/config.json** 文件位于您的主目录中，并具有如下格式：

```
auths:
  index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
  docker.io/my-namespace/my-user/my-image: 4
    auth: "GzhYWRGU6R2fbclabnRgbkSp=""
    email: "user@example.com"
  docker.io/my-namespace: 5
    auth: "GzhYWRGU6R2deesfrRgbkSp=""
    email: "user@example.com"
```

1. registry URL。
2. 加密的密码。
3. 用于登录的电子邮件地址。
4. 命名空间中的特定镜像的 URL 和凭证。
5. registry 命名空间的 URL 和凭证。

您可以定义多个容器镜像 registry，或在同一 registry 中定义多个存储库。或者，也可以通过运行 **docker login** 命令将身份验证条目添加到此文件中。如果文件不存在，则会创建此文件。

Kubernetes 提供**Secret** 对象，可用于存储配置和密码。

先决条件

- 您必须有一个 **.docker/config.json** 文件。

流程

1. 输入以下命令从本地 **.docker/config.json** 文件创建 secret：

```
$ oc create secret generic dockerhub \
    --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
    --type=kubernetes.io/dockerconfigjson
```

这将生成名为 **dockerhub** 的 secret 的 JSON 规格并创建该对象。

2. 将 **pushSecret** 字段添加到 **BuildConfig** 中的 **output** 部分，并将它设为您创建的 **secret** 的名称，上例中为 **dockerhub**：

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "private.registry.com/org/private-image:latest"
    pushSecret:
      name: "dockerhub"
```

您可以使用 **oc set build-secret** 命令在构建配置上设置推送 secret：

```
$ oc set build-secret --push bc/sample-build dockerhub
```

您还可以将 push secret 与构建使用的服务帐户链接，而不指定 **pushSecret** 字段。默认情况下，构建使用 **builder** 服务帐户。如果 secret 包含与托管构建输出镜像的存储库匹配的凭证，则 push secret 会自动添加到构建中。

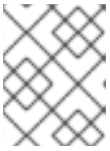
```
$ oc secrets link builder dockerhub
```

3. 通过指定 **pullSecret** 字段（构建策略定义的一部分），从私有容器镜像 registry 拉取构建器容器镜像：

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
      name: "docker.io/user/private_repository"
    pullSecret:
      name: "dockerhub"
```

您可以使用 **oc set build-secret** 命令在构建配置上设置拉取 secret：

```
$ oc set build-secret --pull bc/sample-build dockerhub
```



注意

本例在 Source 构建中使用 **pullSecret**，但也适用于 Docker 构建和 Custom 构建。

您还可以将 pull secret 链接到构建使用的服务帐户，而不指定 **pullSecret** 字段。默认情况下，构建使用 **builder** 服务帐户。如果 secret 包含与托管构建的输入镜像的存储库匹配的凭证，pull secret 会自动添加到构建中。要将 pull secret 链接到构建使用的服务帐户，而不指定 **pullSecret** 字段，请输入以下命令：

```
$ oc secrets link builder dockerhub
```

注意

您必须在 **BuildConfig** spec 中指定一个 **from** 镜像，才能利用此功能。由 **oc new-build** 或 **oc new-app** 生成的 Docker 策略构建在某些情况下可能无法进行这个操作。

2.3.9. 构建环境

与 Pod 环境变量一样，可以定义构建环境变量，在使用 Downward API 时引用其他源或变量。需要注意一些例外情况。

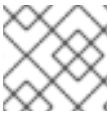
您也可以使用 **oc set env** 命令管理 **BuildConfig** 中定义的环境变量。

注意

不支持在构建环境变量中使用 **valueFrom** 引用容器资源，因为这种引用在创建容器之前解析。

2.3.9.1. 使用构建字段作为环境变量

您可以注入构建对象的信息，使用 **fieldPath** 环境变量源指定要获取值的字段的 **JsonPath**。



注意

Jenkins Pipeline 策略不支持将 **valueFrom** 语法用于环境变量。

流程

- 将 **fieldPath** 环境变量源设置为您有兴趣获取其值的字段的 **JsonPath**：

```
env:
  - name: FIELDREF_ENV
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

2.3.9.2. 使用 secret 作为环境变量

您可以使用 **valueFrom** 语法，将 secret 的键值作为环境变量提供。

重要

此方法在构建容器集控制台的输出中以纯文本形式显示机密。要避免这种情况，请使用输入 secret 和配置映射。

流程

- 要将 secret 用作环境变量，请设置 **valueFrom** 语法：

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: MYVAL
        valueFrom:
          secretKeyRef:
            key: myval
            name: mysecret
```

其他资源

- 输入 secret 和配置映射

2.3.10. 服务用（service serving）证书 secret

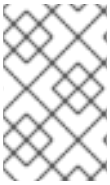
服务用证书 secret 旨在支持需要开箱即用证书的复杂中间件应用程序。它的设置与管理员工具为节点和 master 生成的服务器证书相同。

流程

要保护与您服务的通信，请让集群生成的签名的服务证书/密钥对保存在您的命令空间的 secret 中。

- 在服务上设置 **service.beta.openshift.io/serving-cert-secret-name** 注解，并将值设置为您要用于 secret 的名称。
  然后，您的 **PodSpec** 可以挂载该 secret。当它可用时，您的 Pod 会运行。该证书对内部服务 DNS 名称 **<service.name>.<service.namespace>.svc** 有效。

  证书和密钥采用 PEM 格式，分别存储在 **tls.crt** 和 **tls.key** 中。证书/密钥对在接近到期时自动替换。在 secret 的 **service.beta.openshift.io/expiry** 注解中查看过期日期，其格式为 RFC3339。



注意

在大多数情形中，服务 DNS 名称 **<service.name>.<service.namespace>.svc** 不可从外部路由。**<service.name>.<service.namespace>.svc** 的主要用途是集群内或服务内通信，也用于重新加密路由。

其他 pod 可以信任由集群创建的证书，这些证书只为内部 DNS 名称签名，方法是使用 pod 中自动挂载的 **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt** 文件中的证书颁发机构（CA）捆绑包。

此功能的签名算法是 **x509.SHA256WithRSA**。要手动轮转，请删除生成的 secret。这会创建新的证书。

2.3.11. secret 限制

若要使用一个 secret，Pod 需要引用该 secret。可以通过三种方式将 secret 用于 Pod：

- 为容器产生环境变量。
- 作为挂载到一个或多个容器上的卷中的文件。
- 在拉取 Pod 的镜像时通过 kubelet 使用。

卷类型 secret 使用卷机制将数据作为文件写入到容器中。**imagePullSecrets** 使用服务帐户将 secret 自动注入到命名空间中的所有 Pod。

当模板包含 secret 定义时，模板使用提供的 secret 的唯一方法是确保验证 secret 卷源通过验证，并且指定的对象引用实际指向 **Secret** 类型的对象。因此，secret 需要在依赖它的任何 Pod 之前创建。确保这一点的最有效方法是通过使用服务帐户自动注入。

Secret API 对象驻留在命名空间中。它们只能由同一命名空间中的 pod 引用。

每个 secret 的大小限制为 1MB。这是为了防止创建可能会耗尽 apiserver 和 kubelet 内存的大型 secret。不过，创建大量较小的 secret 也可能耗尽内存。

2.4. 管理构建输出

以下小节提供了管理构建输出的概览和说明。

2.4.1. 构建输出

使用 docker 或 source-to-image（S2I）策略的构建会导致创建新的容器镜像。镜像而后被推送到由 **Build** 规格的 **output** 部分中指定的容器镜像 registry 中。

如果输出类型是 **ImageStreamTag**，则镜像将推送到集成的 OpenShift 镜像 registry 并在指定的镜像流中标记。如果输出类型为 **DockerImage**，则输出引用的名称将用作 docker push 规格。规格中可以包含 registry；如果没有指定 registry，则默认为 DockerHub。如果 Build 规格的 output 部分为空，则构建结束时不推送镜像。

输出到 ImageStreamTag

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "sample-image:latest"
```

输出到 docker Push 规格

```
spec:
  output:
    to:
      kind: "DockerImage"
      name: "my-registry.mycompany.com:5000/myimages/myimage:tag"
```

2.4.2. 输出镜像环境变量

Docker 和 source-to-image（S2I）策略构建设置输出镜像的以下环境变量：

变量	描述
OPENSHIFT_BUILD_NAME	构建的名称
OPENSHIFT_BUILD_NAMESPACE	构建的命名空间
OPENSHIFT_BUILD_SOURCE	构建的源 URL
OPENSHIFT_BUILD_REFERENCE	构建中使用的 Git 引用
OPENSHIFT_BUILD_COMMIT	构建中使用的源提交

此外，任何用户定义的环境变量（例如，使用 S2I 或 docker 策略选项配置的环境变量）也将是输出镜像环境变量列表的一部分。

2.4.3. 输出镜像标签

Docker 和 Source-to-image (S2I) 构建在输出镜像上设置以下标签：

标签	描述
io.openshift.build.commit.author	构建中使用的源提交的作者
io.openshift.build.commit.date	构建中使用的源提交的日期
io.openshift.build.commit.id	构建中使用的源提交的哈希值
io.openshift.build.commit.message	构建中使用的源提交的消息
io.openshift.build.commit.ref	源中指定的分支或引用
io.openshift.build.source-location	构建的源 URL

您还可以使用 **BuildConfig.spec.output.imageLabels** 字段指定将应用到构建配置构建的每个镜像的自定义标签列表。

构建镜像的自定义标签

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "my-image:latest"
    imageLabels:
    - name: "vendor"
```

```
    value: "MyCompany"
  - name: "authoritative-source-url"
    value: "registry.mycompany.com"
```

2.5. 使用构建策略

以下小节定义了受支持的主要构建策略，以及它们的使用方法。

2.5.1. Docker 构建

OpenShift Container Platform 使用 Buildah 从 Dockerfile 构建容器镜像。有关使用 Dockerfile 构建容器镜像的更多信息，请参阅 Dockerfile 参考文档。

提示

如果使用 **buildArgs** 数组设置 Docker 构建参数，请参阅 Dockerfile 参考文档中 了解 ARG 和 FROM 如何交互。

2.5.1.1. 替换 Dockerfile FROM 镜像

您可以将 Dockerfile 的 **FROM** 指令替换为 **BuildConfig** 对象的 **from** 参数。如果 Dockerfile 使用多阶段构建，最后一个 **FROM** 指令中的镜像将被替换。

流程

- 要将 Dockerfile 的 **FROM** 指令替换为 **BuildConfig** 对象的 **from** 参数，请在 **BuildConfig** 对象中添加以下设置：

```
strategy:
  dockerStrategy:
    from:
      kind: "ImageStreamTag"
      name: "debian:latest"
```

2.5.1.2. 使用 Dockerfile 路径

默认情况下，docker 构建使用位于 **BuildConfig.spec.source.contextDir** 字段中指定的上下文的根目录下的 Dockerfile。

dockerfilePath 字段允许构建使用不同的路径来定位 Dockerfile，该路径相对于 **BuildConfig.spec.source.contextDir** 字段。它可以是不同于默认 Dockerfile 的其他文件名，如 **MyDockerfile**，也可以是子目录中 Dockerfile 的路径，如 **dockerfiles/app1/Dockerfile**。

流程

- 设置构建的 **dockerfilePath** 字段，以使用不同的路径来定位 Dockerfile：

```
strategy:
  dockerStrategy:
    dockerfilePath: dockerfiles/app1/Dockerfile
```

2.5.1.3. 使用 Docker 环境变量

要将环境变量提供给 docker 构建过程和生成的镜像使用，您可以在构建配置的 **dockerStrategy** 定义中添加环境变量。

这里定义的环境变量作为单个 **ENV** Dockerfile 指令直接插入到 **FROM** 指令后，以便稍后可在 Dockerfile 内引用该变量。

变量在构建期间定义并保留在输出镜像中，因此它们也会出现在运行该镜像的任何容器中。

例如，定义要在构建和运行时使用的自定义 HTTP 代理：

```
dockerStrategy:
...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

您还可以使用 **oc set env** 命令管理构建配置中定义的环境变量。

2.5.1.4. 添加 Docker 构建参数

您可以使用 **buildArgs** 数组来设置 Docker 构建参数。构建参数将在构建启动时传递给 Docker。

提示

请参阅 Dockerfile 参考文档中的 ARG 和 FROM 如何交互。

流程

- 要设置 Docker 构建参数，请在 **buildArgs** 中添加条目，它位于 **BuildConfig** 对象的 **dockerStrategy** 定义中。例如：

```
dockerStrategy:
...
  buildArgs:
    - name: "version"
      value: "latest"
```

注意

只支持 **name** 和 **value** 字段。**valueFrom** 字段上的任何设置都会被忽略。

2.5.1.5. 使用 docker 构建的 Squash 层

通常，Docker 构建会为 Dockerfile 中的每条指令都创建一个层。将 **imageOptimizationPolicy** 设置为 **SkipLayers**，可将所有指令合并到基础镜像顶部的单个层中。

流程

- 将 **imageOptimizationPolicy** 设置为 **SkipLayers**：

```
strategy:
  dockerStrategy:
    imageOptimizationPolicy: SkipLayers
```

2.5.1.6. 使用构建卷

您可以挂载构建卷，为运行的构建授予您不想在输出容器镜像中保留的信息的访问权限。

构建卷提供仅在构建时需要的敏感信息，如存储库凭据。构建卷与构建输入不同，后者的数据可以保留在输出容器镜像中。

构建卷的挂载点（运行中的构建从中读取数据）在功能上与 pod 卷挂载 类似。

先决条件

- 您已将输入 secret、配置映射或两者添加到 BuildConfig 对象中。

流程

- 在 **BuildConfig** 对象的 **dockerStrategy** 定义中，将任何构建卷添加到 **volumes** 数组中。例如：

```
spec:
  dockerStrategy:
    volumes:
      - name: secret-mvn 1
        mounts:
        - destinationPath: /opt/app-root/src/.ssh 2
        source:
          type: Secret 3
          secret:
            secretName: my-secret 4
      - name: settings-mvn 5
        mounts:
        - destinationPath: /opt/app-root/src/.m2 6
        source:
          type: ConfigMap 7
          configMap:
            name: my-config 8
      - name: my-csi-volume 9
        mounts:
        - destinationPath: /opt/app-root/src/some_path 10
        source:
          type: CSI 11
          csi:
            driver: csi.sharedresource.openshift.io 12
            readOnly: true 13
            volumeAttributes: 14
              attribute: value
```

1 5 9 必需。唯一的名称。

2 6 10 必需。挂载点的绝对路径。它不能包含 **..** 或 **:** 且不与构建器生成的目的地路径冲突。**/opt/app-root/src** 是许多支持 Red Hat S2I 的镜像的默认主目录。

3 7 11 必需。源类型，**ConfigMap**、**Secret** 或 **CSI**。

4 8 必需。源的名称。

12 必需。提供临时 CSI 卷的驱动程序。

13 必需。这个值必须设为 **true**。提供只读卷。

14 可选。临时 CSI 卷的卷属性。如需支持的属性键和值，请参阅 CSI 驱动程序的文档。

重要

共享资源 CSI 驱动程序只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

重要

共享资源 CSI 驱动程序只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

其他资源

- 构建输入
- 输入 secret 和配置映射

2.5.2. Source-to-image 构建

Source-to-Image (S2I) 是一种用于构建可重复生成的容器镜像的工具。它通过将应用程序源代码注入容器镜像并汇编新镜像来生成可随时运行的镜像。新镜像融合了基础镜像（构建器）和构建的源代码，并可搭配 **buildah run** 命令使用。S2I 支持递增构建，可重复利用以前下载的依赖项和过去构建的工件等。

2.5.2.1. 执行 source-to-image 增量构建

Source-to-image (S2I) 可以执行增量构建，也就是能够重复利用过去构建的镜像中的工件。

流程

- 要创建增量构建，请创建对策略定义进行以下修改：

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "incremental-image:latest" 1
    incremental: true 2
```

1 指定支持增量构建的镜像。请参考构建器镜像的文档，以确定它是否支持此行为。

2 此标志（flag）控制是否尝试增量构建。如果构建器镜像不支持增量构建，则构建仍将成功，但您会收到一条日志消息，指出增量构建因为缺少 **save-artifacts** 脚本而未能成功。

其他资源

- 如需有关如何创建支持增量构建的构建器镜像的信息，请参阅 S2I 要求。

2.5.2.2. 覆盖 source-to-image 构建器镜像脚本

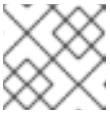
您可以覆盖构建器镜像提供的 **assemble**、**run** 和 **save-artifacts** source-to-image(S2I)脚本。

流程

- 要覆盖构建器镜像提供的 **assemble**、**run** 和 **save-artifacts** S2I 脚本，请完成以下操作之一：
  - 在应用程序源存储库的 **.s2i/bin** 目录中提供 **assemble**、**run** 或 **save-artifacts** 脚本。
  - 提供包含脚本的目录 URL，作为 **BuildConfig** 对象中的策略定义的一部分。例如：

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest"
    scripts: "http://somehost.com/scripts_directory" 1
```

1 构建过程将 **run**、**assemble** 和 **save-artifacts** 附加到路径中。如果存在具有这些名称的任何或所有脚本，构建过程将使用这些脚本代替镜像中提供的同名脚本。



注意

位于 **scripts** URL 的文件优先于源存储库的 **.s2i/bin** 中的文件。

2.5.2.3. Source-to-image 环境变量

可以通过两种方式将环境变量提供给源构建过程使用，并生成镜像：环境文件和 **BuildConfig** 环境值。使用任一方法提供的变量将在构建过程中和输出镜像中存在。

2.5.2.3.1. 使用 Source-to-image 环境文件

利用源代码构建，您可以在应用程序内设置环境值（每行一个），方法是在源存储库中的 **.s2i/environment** 文件中指定它们。此文件中指定的环境变量存在于构建过程和输出镜像。

如果您在源存储库中提供 **.s2i/environment** 文件，则 source-to-image(S2I)会在构建期间读取此文件。这允许自定义构建行为，因为 **assembe** 脚本可能会使用这些变量。

流程

例如，在构建期间禁用 Rails 应用程序的资产编译：

- 在 **.s2i/environment** 文件中添加 **DISABLE_ASSET_COMPILATION=true**。

除了构建之外，指定的环境变量也可以在运行的应用程序本身中使用。例如，使 Rails 应用程序在 **development** 模式而非 **production** 模式中启动：

- 在 **.s2i/environment** 文件中添加 **RAILS_ENV=development**。

使用镜像部分中提供了各个镜像支持的环境变量的完整列表。

2.5.2.3.2. 使用 Source-to-image 构建配置环境

您可以在构建配置的 **sourceStrategy** 定义中添加环境变量。这里定义的环境变量可在 **assemble** 脚本执行期间看到，也会在输出镜像中定义，使它们能够供 **run** 脚本和应用程序代码使用。

流程

- 例如，禁用 Rails 应用程序的资产编译：

```
sourceStrategy:
...
  env:
    - name: "DISABLE_ASSET_COMPILATION"
      value: "true"
```

其他资源

- “构建环境”部分提供了更多高级指导。
- 您还可以使用 **oc set env** 命令管理构建配置中定义的环境变量。

2.5.2.4. 忽略 source-to-image 源文件

Source-to-Image（S2I） 支持 **.s2iignore** 文件，该文件包含了需要被忽略的文件列表。构建工作目录中的文件（由各种输入源提供）若与 **.s2iignore** 文件中指定的文件匹配，将不会提供给 **assemble** 脚本使用。

2.5.2.5. 使用 Source-to-image 从源代码创建镜像

Source-to-Image (S2I) 是一种框架，它可以轻松地将应用程序源代码作为输入，生成可运行编译的应用程序的新镜像。

使用 S2I 构建可重复生成的容器镜像的主要优点是便于开发人员使用。作为构建器镜像作者，您必须理解两个基本概念，构建过程和 S2I 脚本，才能让您的镜像提供最佳的 S2I 性能。

2.5.2.5.1. 了解 source-to-image 构建过程

构建过程包含以下三个基本元素，这些元素组合成最终的容器镜像：

- 源
- Source-to-image(S2I)脚本
- 构建器镜像

S2I 生成带有构建器镜像的 Dockerfile 作为第一个 **FROM** 指令。然后，由 S2I 生成的 Dockerfile 会被传递给 Buildah。

2.5.2.5.2. 如何编写 Source-to-image 脚本

您可以使用任何编程语言编写 S2I 脚本，只要脚本可在构建器镜像中执行。S2I 支持多种提供 **assemble/run/save-artifacts** 脚本的选项。每次构建时按以下顺序检查所有这些位置：

1. 构建配置中指定的脚本。
2. 在应用程序源 **.s2i/bin** 目录中找到的脚本。
3. 在默认镜像 URL 中找到的带有 **io.openshift.s2i.scripts-url** 标签的脚本。

镜像中指定的 **io.openshift.s2i.scripts-url** 标签和构建配置中指定的脚本都可以采用以下形式之一：

- **image:///path_to_scripts_dir**：镜像中 S2I 脚本所处目录的绝对路径
- **file:///path_to_scripts_dir**：主机上 S2I 脚本所处目录的相对或绝对路径
- **http(s)://path_to_scripts_dir**：S2I 脚本所处目录的 URL

表 2.1. S2I 脚本

脚本	描述
assemble	**assemble** 用来从源代码构建应用程序工件，并将其放置在镜像内部的适当目录中的脚本。这个脚本是必需的。此脚本的工作流为： 1. 可选：恢复构建工件。如果要支持增量构建，确保同时定义了 **save-artifacts**。 2. 将应用程序源放在所需的位置。 3. 构建应用程序工件。 4. 将工件安装到适合它们运行的位置。
run	**run** 脚本将执行您的应用程序。这个脚本是必需的。
save-artifacts	**save-artifacts** 脚本将收集所有可加快后续构建过程的依赖项。这个脚本是可选的。例如： - 对于 Ruby，由 Bundler 安装的 **gem**。 - 对于 Java，**.m2** 内容。 这些依赖项会收集到一个 **tar** 文件中，并传输到标准输出。
usage	借助 **usage** 脚本，可以告知用户如何正确使用您的镜像。这个脚本是可选的。

脚本	描述
test/run	借助 **test/run** 脚本，可以创建一个进程来检查镜像是否正常工作。这个脚本是可选的。该流程的建议工作流是： 1. 构建镜像。 2. 运行镜像以验证 **usage** 脚本。 3. 运行 **s2i build** 以验证 **assemble** 脚本。 4. 可选：再次运行 **s2i build**，以验证 **save-artifacts** 和 **assemble** 脚本的保存和恢复工件功能。 5. 运行镜像，以验证测试应用程序是否正常工作。  **注意** 建议将 **test/run** 脚本构建的测试应用程序放置到镜像存储库中的 **test/test-app** 目录。

S2I 脚本示例

以下示例 S2I 脚本采用 Bash 编写。每个示例都假定其 **tar** 内容解包到 **/tmp/s2i** 目录中。

assemble 脚本：

```
#!/bin/bash

# restore build artifacts
if [ "$(ls /tmp/s2i/artifacts/ 2>/dev/null)" ]; then
    mv /tmp/s2i/artifacts/* $HOME/.
fi

# move the application source
mv /tmp/s2i/src $HOME/src

# build application artifacts
pushd ${HOME}
make all

# install the artifacts
make install
popd
```

run 脚本：

```
#!/bin/bash

# run the application
/opt/application/run.sh
```

save-artifacts 脚本：

```
#!/bin/bash

pushd ${HOME}
if [ -d deps ]; then
    # all deps contents to tar stream
    tar cf - deps
fi
popd
```

usage 脚本：

```
#!/bin/bash

# inform the user how to use the image
cat <<EOF
This is a S2I sample builder image, to use it, install
https://github.com/openshift/source-to-image
EOF
```

其他资源

- S2I 镜像创建教程

2.5.2.6. 使用构建卷

您可以挂载构建卷，为运行的构建授予您不想在输出容器镜像中保留的信息的访问权限。

构建卷提供仅在构建时需要的敏感信息，如存储库凭据。构建卷与构建输入不同，后者的数据可以保留在输出容器镜像中。

构建卷的挂载点（运行中的构建从中读取数据）在功能上与 pod 卷挂载 类似。

先决条件

- 您已将输入 secret、配置映射或两者添加到 BuildConfig 对象中。

流程

- 在 **BuildConfig** 对象的 **sourceStrategy** 定义中，将任何构建卷添加到 **volumes** 数组中。例如：

```
spec:
  sourceStrategy:
    volumes:
      - name: secret-mvn (1)
        mounts:
        - destinationPath: /opt/app-root/src/.ssh (2)
        source:
          type: Secret (3)
          secret:
            secretName: my-secret (4)
      - name: settings-mvn (5)
        mounts:
        - destinationPath: /opt/app-root/src/.m2 (6)
```

```
    source:
      type: ConfigMap (7)
      configMap:
        name: my-config (8)
  - name: my-csi-volume (9)
    mounts:
    - destinationPath: /opt/app-root/src/some_path (10)
    source:
      type: CSI (11)
      csi:
        driver: csi.sharedresource.openshift.io (12)
        readOnly: true (13)
        volumeAttributes: (14)
          attribute: value
```

(1) (5) (9) 必需。唯一的名称。

(2) (6) (10) 必需。挂载点的绝对路径。它不能包含 **..** 或 **:** 且不与构建器生成的目的地路径冲突。**/opt/app-root/src** 是许多支持 Red Hat S2I 的镜像的默认主目录。

(3) (7) (11) 必需。源类型，**ConfigMap**、**Secret** 或 **CSI**。

(4) (8) 必需。源的名称。

(12) 必需。提供临时 CSI 卷的驱动程序。

(13) 必需。这个值必须设为 **true**。提供只读卷。

(14) 可选。临时 CSI 卷的卷属性。如需支持的属性键和值，请参阅 CSI 驱动程序的文档。

重要

共享资源 CSI 驱动程序只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

其他资源

- 构建输入
- 输入 secret 和配置映射

2.5.3. Custom 构建

采用自定义构建策略时，开发人员可以定义负责整个构建过程的特定构建器镜像。通过利用自己的构建器镜像，可以自定义构建流程。

自定义构建器镜像是嵌入了构建过程逻辑的普通容器镜像，例如用于构建 RPM 或基本镜像的逻辑。

自定义构建以级别很高的特权运行，默认情况下不可供用户使用。只有可赋予集群管理权限的用户才应被授予运行自定义构建的权限。

2.5.3.1. 使用 FROM 镜像进行自定义构建

您可以使用 **customStrategy.from** 部分来指示要用于自定义构建的镜像。

流程

- 设置 **customStrategy.from** 部分：

```
strategy:
  customStrategy:
    from:
      kind: "DockerImage"
      name: "openshift/sti-image-builder"
```

2.5.3.2. 在自定义构建中使用 secret

除了可以添加到所有构建类型的源和镜像的 secret 之外，自定义策略还允许向构建器 Pod 添加任意 secret 列表。

流程

- 要将各个 secret 挂载到特定位置，编辑 **策略** YAML 文件的 **secretSource** 和 **mountPath** 字段：

```
strategy:
  customStrategy:
    secrets:
      - secretSource: 1
          name: "secret1"
        mountPath: "/tmp/secret1" 2
      - secretSource:
          name: "secret2"
        mountPath: "/tmp/secret2"
```

1. **secretSource** 是对与构建相同的命名空间中的 secret 的引用。
2. **mountPath** 是自定义构建器中应挂载 secret 的路径。

2.5.3.3. 使用环境变量进行自定义构建

要将环境变量提供给自定义构建过程使用，您可以在构建配置的 **customStrategy** 定义中添加环境变量。

这里定义的环境变量将传递给运行自定义构建的 Pod。

流程

1. 定义在构建期间使用的自定义 HTTP 代理：

```
customStrategy:
...
  env:
    - name: "HTTP_PROXY"
      value: "http://myproxy.net:5187/"
```

2. 要管理构建配置中定义的环境变量，请输入以下命令：

```
$ oc set env <enter_variables>
```

2.5.3.4. 使用自定义构建器镜像

OpenShift Container Platform 的自定义构建策略允许您定义负责整个构建过程的特定构建器镜像。当您需要构建来生成单独的工件，如软件包、JAR、WAR、可安装的 ZIP 或基础镜像时，请使用自定义构建器镜像。

自定义构建器镜像是嵌入构建过程逻辑的普通容器镜像，用于构建工件，如 RPM 或基础容器镜像。

另外，自定义构建器允许实施任何扩展构建过程，如运行单元或集成测试的 CI/CD 流。

2.5.3.4.1. 自定义构建器镜像

在调用时，自定义构建器镜像将接收以下环境变量以及继续进行构建所需要的信息：

表 2.2. 自定义构建器环境变量

变量名称	描述
BUILD	**Build** 对象定义的完整序列化 JSON。如果必须使用特定的 API 版本进行序列化，您可以在构建配置的自定义策略规格中设置 **buildAPIVersion** 参数。
SOURCE_REPOSITORY	包含要构建的源代码的 Git 存储库的 URL。
SOURCE_URI	使用与 **SOURCE_REPOSITORY** 相同的值。可以使用其中任一个。
SOURCE_CONTEXT_DIR	指定要在构建时使用的 Git 存储库的子目录。只有定义后才出现。
SOURCE_REF	要构建的 Git 引用。
ORIGIN_VERSION	创建此构建对象的 OpenShift Container Platform master 的版本。
OUTPUT_REGISTRY	镜像要推送到的容器镜像 registry。
OUTPUT_IMAGE	所构建镜像的容器镜像标签名称。
PUSH_DOCKERCFG_PATH	用于运行 **podman push** 操作的容器 registry 凭证的路径。

2.5.3.4.2. 自定义构建器工作流

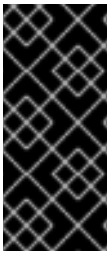
虽然自定义构建器镜像作者在定义构建过程时具有很大的灵活性，但构建器镜像仍必须遵循如下必要的步骤，才能在 OpenShift Container Platform 内无缝运行构建：

1. **Build** 对象定义包含有关构建的输入参数的所有必要信息。

2. 运行构建过程。

3. 如果构建生成了镜像，则将其推送到构建的输出位置（若已定义）。可通过环境变量传递其他输出位置。

2.5.4. Pipeline 构建



重要

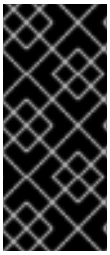
Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

采用 Pipeline 构建策略时，开发人员可以定义 Jenkins 管道，供 Jenkins 管道插件使用。构建可以由 OpenShift Container Platform 启动、监控和管理，其方式与任何其他构建类型相同。

Pipeline 工作流在 **jenkinsfile** 中定义，或直接嵌入在构建配置中，或者在 Git 存储库中提供并由构建配置引用。

2.5.4.1. 了解 OpenShift Container Platform 管道



重要

Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

通过管道（pipeline），您可以控制在 OpenShift Container Platform 上构建、部署和推进您的应用程序。通过结合使用 Jenkins Pipeline 构建策略、**jenkinsfile** 和 Jenkins 客户端插件提供的 OpenShift Container Platform 域特定语言（DSL），您可以为任何场景创建高级构建、测试、部署和推进管道。

OpenShift Container Platform Jenkins 同步插件

OpenShift Container Platform Jenkins 同步插件使构建配置和构建对象与 Jenkins 任务和构建保持同步，并提供以下功能：

- Jenkins 中动态作业并运行创建。
- 从镜像流、镜像流标签或配置映射动态创建代理 Pod 模板。
- 注入环境变量。
- OpenShift Container Platform Web 控制台中的管道视觉化。
- 与 Jenkins Git 插件集成，后者将 OpenShift Container Platform 构建的提交信息传递给 Jenkins Git 插件。
- 将 secret 同步到 Jenkins 凭证条目。

OpenShift Container Platform Jenkins 客户端插件

OpenShift Container Platform Jenkins 客户端插件是一种 Jenkins 插件，旨在提供易读、简洁、全面且流畅的 Jenkins Pipeline 语法，以便与 OpenShift Container Platform API 服务器进行丰富的交互。该插件使用 OpenShift Container Platform 命令行工具 **oc**，此工具必须在执行脚本的节点上可用。

Jenkins 客户端插件必须安装到 Jenkins master上，这样才能在您的应用程序的 **jenkinsfile** 中使用 OpenShift Container Platform DSL。使用 OpenShift Container Platform Jenkins 镜像时，默认安装并启用此插件。

对于项目中的 OpenShift Container Platform 管道，必须使用 Jenkins Pipeline 构建策略。此策略默认使用源存储库根目录下的 **jenkinsfile**，但也提供以下配置选项：

- 构建配置中的内联 **jenkinsfile** 字段。
- 构建配置中的 **jenkinsfilePath** 字段，该字段引用要使用的 **jenkinsfile** 的位置，该位置相对于源 **contextDir**。

注意

可选的 **jenkinsfilePath** 字段指定要使用的文件的名称，其路径相对于源 **contextDir**。如果省略了 **contextDir**，则默认为存储库的根目录。如果省略了 **jenkinsfilePath**，则默认为 **jenkinsfile**。

2.5.4.2. 为管道构建提供 Jenkins 文件

重要

Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

jenkinsfile 使用标准的 Groovy 语言语法，允许对应用程序的配置、构建和部署进行精细控制。

您可以通过以下一种方式提供 **jenkinsfile**：

- 位于源代码存储库中的文件。
- 使用 **jenkinsfile** 字段嵌入为构建配置的一部分。

使用第一个选项时，**jenkinsfile** 必须包含在以下位置之一的应用程序源代码存储库中：

- 存储库根目录下名为 **jenkinsfile** 的文件。
- 存储库的源 **contextDir** 的根目录下名为 **jenkinsfile** 的文件。
- 通过 BuildConfig 的 **JenkinsPipelineStrategy** 部分的 **jenkinsfilePath** 字段指定的文件名；若提供，则路径相对于源 **contextDir**，否则默认为存储库的根目录。

jenkinsfile 在 Jenkins 代理 Pod 上运行，如果您打算使用 OpenShift Container Platform DSL，它必须具有 OpenShift Container Platform 客户端二进制文件。

流程

要提供 Jenkins 文件，您可以：

- 在构建配置中嵌入 Jenkins 文件。
- 在构建配置中包含对包含 Jenkins 文件的 Git 存储库的引用。

嵌入式定义

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node('agent') {
          stage 'build'
          openshiftBuild(buildConfig: 'ruby-sample-build', showBuildLogs: 'true')
          stage 'deploy'
          openshiftDeploy(deploymentConfig: 'frontend')
        }
```

引用 Git 存储库

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  source:
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfilePath: some/repo/dir/filename 1
```

1 可选的 **jenkinsfilePath** 字段指定要使用的文件的名称，其路径相对于源 **contextDir**。如果省略了 **contextDir**，则默认为存储库的根目录。如果省略了 **jenkinsfilePath**，则默认为 **jenkinsfile**。

2.5.4.3. 使用环境变量进行 Pipeline 构建

重要

Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

要将环境变量提供给 Pipeline 构建过程使用，您可以在构建配置的 **jenkinsPipelineStrategy** 定义中添加环境变量。

定义后，环境变量将设置为与构建配置关联的任何 Jenkins 任务的参数。

流程

- 要定义在构建期间使用的环境变量，编辑 YAML 文件：

```
jenkinsPipelineStrategy:
...
  env:
    - name: "FOO"
      value: "BAR"
```

您还可以使用 **oc set env** 命令管理构建配置中定义的环境变量。

2.5.4.3.1. BuildConfig 环境变量和 Jenkins 任务参数之间的映射

根据对 Pipeline 策略构建配置的更改创建或更新 Jenkins 任务时，构建配置中的任何环境变量都会映射到 Jenkins 任务参数定义，其中 Jenkins 任务参数定义的默认值是关联环境变量的当前值。

在 Jenkins 任务初始创建之后，您仍然可以从 Jenkins 控制台向任务添加其他参数。参数名称与构建配置中的环境变量名称不同。为这些 Jenkins 任务启动构建时，将遵循这些参数。

为 Jenkins 任务启动构建的方式决定了如何设置参数。

- 如果使用 **oc start-build** 启动，则构建配置中的环境变量值是为对应作业实例设置的参数。您在 Jenkins 控制台中对参数默认值所做的更改都将被忽略。构建配置值具有优先权。
- 如果使用 **oc start-build -e** 启动，则 **-e** 选项中指定的环境变量值具有优先权。
  - 如果指定没有列在构建配置中列出的环境变量，它们将添加为 Jenkins 任务参数定义。
  - 您在 Jenkins 控制台中对与环境变量对应的参数所做的更改都将被忽略。构建配置以及您通过 **oc start-build -e** 指定的值具有优先权。
- 如果使用 Jenkins 控制台启动 Jenkins 任务，您可以使用 Jenkins 控制台控制参数的设置，作为启动任务构建的一部分。

注意

建议您在构建配置中指定与作业参数关联的所有可能环境变量。这样做可以减少磁盘 I/O 并提高 Jenkins 处理期间的性能。

2.5.4.4. Pipeline 构建教程

重要

Pipeline 构建策略在 OpenShift Container Platform 4 中弃用。基于 Tekton 的 OpenShift Container Platform Pipelines 中带有等效且改进的功能。

OpenShift Container Platform 上的 Jenkins 镜像被完全支持，用户可以按照 Jenkins 用户文档在作业中定义 **jenkinsfile**，或者将其存储在 Source Control Management 系统中。

本例演示如何创建 OpenShift Container Platform Pipeline，以使用 **nodejs-mongodb.json** 模板构建、部署和验证 **Node.js/MongoDB** 应用程序。

流程

1. 创建 Jenkins master：

```
$ oc project <project_name>
```

选择要使用的项目，或使用 **oc new-project <project_name>** 创建一个新项目。

```
$ oc new-app jenkins-ephemeral 1
```

如果要使用持久性存储，请改用 **jenkins-persistent**。

2. 使用以下内容，创建名为 **nodejs-sample-pipeline.yaml** 的文件：

注意

这将创建一个 **BuildConfig** 对象，它将使用 Jenkins Pipeline 策略来构建、部署和扩展 **Node.js/MongoDB** 示例应用程序。

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "nodejs-sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: <pipeline content from below>
    type: JenkinsPipeline
```

3. 使用 **jenkinsPipelineStrategy** 创建 **BuildConfig** 对象后，通过使用内联 **jenkinsfile** 告知管道做什么：

注意

本例没有为应用程序设置 Git 存储库。

以下 **jenkinsfile** 内容使用 OpenShift Container Platform DSL 以 Groovy 语言编写。在本例中，请使用 YAML Literal Style 在 **BuildConfig** 中包含内联内容，但首选的方法是使用源存储库中的 **jenkinsfile**。

```
def templatePath = 'https://raw.githubusercontent.com/openshift/nodejs-
ex/master/openshift/templates/nodejs-mongodb.json' 1
def templateName = 'nodejs-mongodb-example' 2
pipeline {
  agent {
    node {
      label 'nodejs' 3
    }
  }
  options {
    timeout(time: 20, unit: 'MINUTES') 4
  }
  stages {
    stage('preamble') {
```

```
    steps {
      script {
        openshift.withCluster() {
          openshift.withProject() {
            echo "Using project: ${openshift.project()}"
          }
        }
      }
    }
  }
  stage('cleanup') {
    steps {
      script {
        openshift.withCluster() {
          openshift.withProject() {
            openshift.selector("all", [ template : templateName ]).delete() 5
            if (openshift.selector("secrets", templateName).exists()) { 6
              openshift.selector("secrets", templateName).delete()
            }
          }
        }
      }
    }
  }
  stage('create') {
    steps {
      script {
        openshift.withCluster() {
          openshift.withProject() {
            openshift.newApp(templatePath) 7
          }
        }
      }
    }
  }
  stage('build') {
    steps {
      script {
        openshift.withCluster() {
          openshift.withProject() {
            def builds = openshift.selector("bc", templateName).related('builds')
            timeout(5) { 8
              builds.untilEach(1) {
                return (it.object().status.phase == "Complete")
              }
            }
          }
        }
      }
    }
  }
  stage('deploy') {
    steps {
      script {
        openshift.withCluster() {
          openshift.withProject() {
```

```
            def rm = openshift.selector("dc", templateName).rollout()
            timeout(5) { 9
              openshift.selector("dc", templateName).related('pods').untilEach(1) {
                return (it.object().status.phase == "Running")
              }
            }
          }
        }
      }
    }
    stage('tag') {
      steps {
        script {
          openshift.withCluster() {
            openshift.withProject() {
              openshift.tag("${templateName}:latest", "${templateName}-staging:latest") 10
            }
          }
        }
      }
    }
  }
}
```

1 要使用的模板的路径。

1 2 要创建的模板的名称。

3 启动 **node.js** 代理 pod 以针对其运行此构建。

4 为此管道设置 20 分钟超时。

5 使用此模板标签删除所有内容。

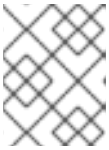
6 使用此模板标签删除任何 secret。

7 从 **templatePath** 创建一个新应用程序。

8 等待最多五分钟以完成构建。

9 等待最多五分钟以完成部署。

10 如果其余部分都成功，则将 **$ {templateName}:latest** 镜像标记为 **$ {templateName}-staging:latest**。stage 环境的管道 BuildConfig 可以监控 **$ {templateName}-staging:latest** 镜像更改，并将它部署到 stage 环境中。



注意

上例使用 declarative pipeline 风格编写，但较旧的 scripted pipeline 风格也受到支持。

4. 在 OpenShift Container Platform 集群中创建管道 **BuildConfig**：

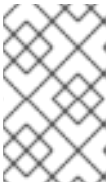
```
$ oc create -f nodejs-sample-pipeline.yaml
```

a. 如果您不想自行创建文件，可以通过运行以下命令来使用 Origin 存储库中的示例：

```
$ oc create -f
https://raw.githubusercontent.com/openshift/origin/master/examples/jenkins/pipeline/nodejs-
sample-pipeline.yaml
```

5. 启动管道：

```
$ oc start-build nodejs-sample-pipeline
```

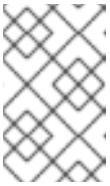


注意

此外，也可以通过 OpenShift Container Platform Web 控制台启动管道，方法是导航到 Builds → Pipeline 部分并点击 **Start Pipeline**，或者访问 Jenkins 控制台，再导航到您创建的管道并点击 **Build Now**。

管道启动之后，您应该看到项目中执行了以下操作：

- 在 Jenkins 服务器上创建了作业实例。
- 如果管道需要，启动一个代理 pod。
- 管道在代理 Pod 上运行，如果不需要代理，则管道在 master 上运行。
  - 将删除之前创建的具有 **template=nodejs-mongodb-example** 标签的所有资源。
  - 从 **nodejs-mongodb-example** 模板创建一个新应用程序及其所有相关资源。
  - 使用 **nodejs-mongodb-example BuildConfig** 启动构建。
    - 管道将等待到构建完成后触发下一阶段。
  - 使用 **nodejs-mongodb-example** 部署配置启动部署。
    - 管道将等待到部署完成后触发下一阶段。
  - 如果构建和部署都成功，则 **nodejs-mongodb-example:latest** 镜像将标记为 **nodejs-mongodb-example:stage**。
- 如果管道需要，则代理 pod 会被删除。



注意

视觉化管道执行的最佳方法是在 OpenShift Container Platform Web 控制台中查看它。您可以通过登录 Web 控制台并导航到 Builds → Pipelines 来查看管道。

2.5.5. 使用 web 控制台添加 secret

您可以在构建配置中添加 secret，以便它可以访问私有存储库。

流程

将 secret 添加到构建配置中，以便它可以从 OpenShift Container Platform Web 控制台访问私有存储库：

1. 创建一个新的 OpenShift Container Platform 项目。
2. 创建一个包含用于访问私有源代码存储库的凭证的 secret。
3. 创建构建配置。
4. 在构建配置编辑器页面上或在 Web 控制台的 **create app from builder image** 页面中，设置 **Source Secret**。
5. 点击 **Save**。

2.5.6. 启用拉取 (pull) 和推送 (push)

您可以通过在构建配置中设置 pull secret 来启用拉取到私有 registry，也可以通过设置 push secret 来启用推送。

流程

启用拉取到私有 registry：

- 在构建配置中设置 pull secret。

启用推送：

- 在构建配置中设置 push secret。

2.6. 使用 BUILDAH 自定义镜像构建

在 OpenShift Container Platform 4.15 中，主机节点上没有 docker socket。这意味着，不能保证自定义构建的 *mount docker socket* 选项会提供可在自定义构建镜像中使用的可访问 docker socket。

如果您需要此功能来构建和推送镜像，请将 Buildah 工具添加到自定义构建镜像中，并在自定义构建逻辑中使用它来构建并推送镜像。以下是如何使用 Buildah 运行自定义构建的示例。

注意

使用自定义构建策略需要普通用户默认情况下不具备的权限，因为它允许用户在集群上运行的特权容器内执行任意代码。此级别的访问权限可被用来进行可能对集群造成损害的操作，因此应仅授权给信任的用户。

2.6.1. 先决条件

- 查看如何授予自定义构建权限。

2.6.2. 创建自定义构建工件

您必须创建要用作自定义构建镜像的镜像。

流程

1. 从空目录着手，使用以下内容创建名为 **Dockerfile** 的文件：

```
FROM registry.redhat.io/rhel8/buildah
# In this example, `/tmp/build` contains the inputs that build when this
# custom builder image is run. Normally the custom builder image fetches
# this content from some location at build time, by using git clone as an example.
ADD dockerfile.sample /tmp/input/Dockerfile
ADD build.sh /usr/bin
RUN chmod a+x /usr/bin/build.sh
# /usr/bin/build.sh contains the actual custom build logic that will be run when
# this custom builder image is run.
ENTRYPOINT ["/usr/bin/build.sh"]
```

2. 在同一目录中，创建名为 **dockerfile.sample** 的文件。此文件将包含在自定义构建镜像中，并且定义将由自定义构建生成的镜像：

```
FROM registry.access.redhat.com/ubi9/ubi
RUN touch /tmp/build
```

3. 在同一目录中，创建名为 **build.sh** 的文件。此文件包含自定义生成运行时将要执行的逻辑：

```
#!/bin/sh
# Note that in this case the build inputs are part of the custom builder image, but normally this
# is retrieved from an external source.
cd /tmp/input
# OUTPUT_REGISTRY and OUTPUT_IMAGE are env variables provided by the custom
# build framework
TAG="${OUTPUT_REGISTRY}/${OUTPUT_IMAGE}"


# performs the build of the new image defined by dockerfile.sample
buildah --storage-driver vfs bud --isolation chroot -t ${TAG} .


# buildah requires a slight modification to the push secret provided by the service
# account to use it for pushing the image
cp /var/run/secrets/openshift.io/push/.dockercfg /tmp
(echo "{ \"auths\": " ; cat /var/run/secrets/openshift.io/push/.dockercfg ; echo "}") >
/tmp/.dockercfg


# push the new image to the target for the build
buildah --storage-driver vfs push --tls-verify=false --authfile /tmp/.dockercfg ${TAG}
```

2.6.3. 构建自定义构建器镜像

您可以使用 OpenShift Container Platform 构建和推送要在 Custom 策略中使用的自定义构建器镜像。

先决条件

- 定义要用于创建新的自定义构建器镜像的所有输入。

流程

1. 定义要用于构建自定义构建器镜像的 **BuildConfig** 对象：

```
$ oc new-build --binary --strategy=docker --name custom-builder-image
```

2. 从您在其中创建自定义构建镜像的目录中，运行构建：

```
$ oc start-build custom-builder-image --from-dir . -F
```

构建完成后，新自定义构建器镜像将在名为 **custom-builder-image:latest** 的镜像流标签中的项目内可用。

2.6.4. 使用自定义构建器镜像

您可以定义一个 **BuildConfig** 对象，它将结合使用 Custom 策略与自定义构建器镜像来执行您的自定义构建逻辑。

先决条件

- 为新自定义构建器镜像定义所有必要的输入。
- 构建您的自定义构建器镜像。

流程

1. 创建名为 **buildconfig.yaml** 的文件。此文件定义要在项目中创建并执行的 **BuildConfig** 对象：

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: sample-custom-build
  labels:
    name: sample-custom-build
  annotations:
    template.alpha.openshift.io/wait-for-ready: 'true'
spec:
  strategy:
    type: Custom
    customStrategy:
      forcePull: true
      from:
        kind: ImageStreamTag
        name: custom-builder-image:latest
        namespace: <yourproject> (1)
  output:
    to:
      kind: ImageStreamTag
      name: sample-custom:latest
```

(1) 指定项目的名称。

2. 运行以下命令来创建 **BuildConfig** 对象：

```
$ oc create -f buildconfig.yaml
```

3. 创建名为 **imagestream.yaml** 的文件。此文件定义构建要将镜像推送到的镜像流：

```
kind: ImageStream
apiVersion: image.openshift.io/v1
metadata:
  name: sample-custom
spec: {}
```

4. 运行以下命令来创建镜像流：

```
$ oc create -f imagestream.yaml
```

5. 输入以下命令运行自定义构建：

```
$ oc start-build sample-custom-build -F
```

构建运行时，它会启动一个 Pod 来运行之前构建的自定义构建器镜像。该 Pod 将运行定义为自定义构建器镜像入口点的 **build.sh** 逻辑。**build.sh** 逻辑调用 Buildah 来构建自定义构建器镜像中嵌入的 **dockerfile.sample**，然后使用 Buildah 将新镜像推送到 **sample-custom 镜像流**。

2.7. 执行和配置基本构建

以下小节提供了有关基本构建操作的说明，包括启动和取消构建、编辑 **BuildConfig**、删除 **BuildConfig**、查看构建详情以及访问构建日志。

2.7.1. 启动构建

您可以从当前项目中的现有构建配置手动启动新构建。

流程

- 要手动启动构建，请输入以下命令：

```
$ oc start-build <buildconfig_name>
```

2.7.1.1. 重新运行构建

您可以使用 **--from-build** 标志，手动重新运行构建。

流程

- 要手动重新运行构建，请输入以下命令：

```
$ oc start-build --from-build=<build_name>
```

2.7.1.2. 流传输构建日志

您可以指定 **--follow** 标志，在 **stdout** 中输出构建日志。

流程

- 要在 **stdout** 中手动输出构建日志，请输入以下命令：

```
$ oc start-build <buildconfig_name> --follow
```

2.7.1.3. 在启动构建时设置环境变量

您可以指定 **--env** 标志，为构建设置任何所需的环境变量。

流程

- 要指定所需的环境变量，请输入以下命令：

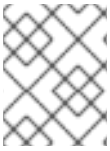
```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

2.7.1.4. 使用源启动构建

您可以通过直接推送源来启动构建，而不依赖于 Git 源拉取或构建的 Dockerfile；源可以是 Git 或 SVN 工作目录的内容、您想要部署的一组预构建二进制工件，或者单个文件。这可以通过为 **start-build** 命令指定以下选项之一来完成：

选项	描述
--from-dir=<directory>	指定将要存档并用作构建的二进制输入的目录。
--from-file=<file>	指定将成为构建源中唯一文件的单个文件。该文件放在空目录的根目录中，其文件名与提供的原始文件相同。
--from-repo=<local_source_repo>	指定用作构建二进制输入的本地存储库的路径。添加 **--commit** 选项以控制要用于构建的分支、标签或提交。

将任何这些选项直接传递给构建时，内容将流传输到构建中并覆盖当前的构建源设置。



注意

从二进制输入触发的构建不会在服务器上保留源，因此基础镜像更改触发的重新构建将使用构建配置中指定的源。

流程

- 要从源代码存储库启动构建，并将本地 Git 存储库的内容作为标签 **v2** 的存档发送，请输入以下命令：

```
$ oc start-build hello-world --from-repo=../hello-world --commit=v2
```

2.7.2. 取消构建

您可以使用 Web 控制台或通过以下 CLI 命令来取消构建。

流程

- 要手动取消构建，请输入以下命令：

```
$ oc cancel-build <build_name>
```

2.7.2.1. 取消多个构建

您可以使用以下 CLI 命令取消多个构建。

流程

- 要手动取消多个构建，请输入以下命令：

```
$ oc cancel-build <build1_name> <build2_name> <build3_name>
```

2.7.2.2. 取消所有构建

您可以使用以下 CLI 命令取消构建配置中的所有构建。

流程

- 要取消所有构建，请输入以下命令：

```
$ oc cancel-build bc/<buildconfig_name>
```

2.7.2.3. 取消给定状态下的所有构建

您可以取消给定状态下的所有构建，如 **new** 或 **pending** 状态，同时忽略其他状态下的构建。

流程

- 要取消给定状态下的所有内容，请输入以下命令：

```
$ oc cancel-build bc/<buildconfig_name>
```

2.7.3. 编辑 BuildConfig

要编辑构建配置，您可以使用 Developer 视角的 **Builds** 视图中的 **Edit BuildConfig** 选项。

您可以使用以下任一视图编辑 **BuildConfig** ：

- **Form 视图** 允许您使用标准表单字段和复选框编辑 **BuildConfig**。
- **YAML 视图** 允许您编辑 **BuildConfig**，完全控制操作。

您可以在 **Form view** 和 **YAML 视图**间切换，而不丢失任何数据。**Form 视图**中的数据传输到 **YAML 视图**，反之亦然。

流程

1. 在 **Developer 视角** 的 **Builds** 视图中，点击菜单  来查看 **Edit BuildConfig** 选项。
2. 点击 **Edit BuildConfig** 以查看 **Form view** 选项。
3. 在 **Git** 部分中，输入您要用来创建应用程序的代码库的 Git 存储库 URL。这个 URL 随后会被验证。

- 可选：点击 **Show Advanced Git Options** 来添加详情，例如：
  - **Git Reference**，用于指定包含您要用来构建应用程序的代码的分支、标签或提交。
  - **Context Dir**，用于指定包含您要用来构建应用程序的代码的子目录。
  - **Source Secret**，创建一个具有用来从私有存储库拉取源代码的凭证的 **Secret Name**。

4. 在 **Build from** 部分中，选择您要从中构建的选项。您可以使用以下选项：
   - **镜像流标签** 引用给定镜像流和标签的镜像。输入您要从构建并推送到的位置的项目、镜像流和标签。
   - **镜像流镜像** 引用给定镜像流和镜像名称的镜像。输入您要从中构建的镜像流镜像。另外，进入要推送到的项目、镜像流和标签。
   - **Docker 镜像** ：通过 Docker 镜像存储库引用 Docker 镜像。您还需要进入项目、镜像流和标签，以引用您要推送到的位置。
5. 可选：在 **Environment Variables** 部分中，使用 **Name** 和 **Value** 字段添加与项目关联的环境变量。要添加更多环境变量，请使用 **Add Value** 或 **Add from ConfigMap** 和 **Secret**。
6. 可选： 要进一步自定义应用程序，请使用以下高级选项：

   Trigger

   构建器镜像更改时触发新镜像构建。点 **Add Trigger** 并选择 **Type** 和 **Secret** 来添加更多触发器。

   Secrets

   为应用添加 secret。点 **Add secret** 并选择 **Secret** 和 **Mountpoint** 来添加更多 secret。

   策略

   单击 **Run policy** 以选择构建运行策略。所选策略决定从构建配置创建的构建必须运行的顺序。

   Hook

   选择择 **Run build hooks after image is built**，以在构建结束时运行命令并验证镜像。添加 **Hook 类型**、**命令** 和 **参数**，以附加到 命令。

7. 单击 **Save** 以保存 **BuildConfig**。

2.7.4. 删除 BuildConfig

您可以使用以下命令来删除 **BuildConfig**。

流程

- 要删除 **BuildConfig**，请输入以下命令：

```
$ oc delete bc <BuildConfigName>
```

  这也会删除从此 **BuildConfig** 实例化的所有构建。

- 要删除 **BuildConfig** 并保留从 **BuildConfig** 中初始化的构建，在输入以下命令时指定 **--cascade=false** 标志：

```
$ oc delete --cascade=false bc <BuildConfigName>
```

2.7.5. 查看构建详情

您可以使用 Web 控制台或 **oc describe** CLI 命令查看构建详情。

这会显示，包括：

- 构建源。
- 构建策略。
- 输出目的地。
- 目标 registry 中的镜像摘要。
- 构建的创建方式。

如果构建采用 **Docker** 或 **Source** 策略，则 **oc describe** 输出还包括用于构建的源修订的相关信息，包括提交 ID、作者、提交者和消息等。

流程

- 要查看构建详情，请输入以下命令：

```
$ oc describe build <build_name>
```

2.7.6. 访问构建日志

您可以使用 Web 控制台或 CLI 访问构建日志。

流程

- 要直接使用构建来流传输日志，请输入以下命令：

```
$ oc describe build <build_name>
```

2.7.6.1. 访问 BuildConfig 日志

您可以使用 Web 控制台或 CLI 访问 **BuildConfig** 日志。

流程

- 要输出 **BuildConfig** 的最新构建的日志，请输入以下命令：

```
$ oc logs -f bc/<buildconfig_name>
```

2.7.6.2. 访问给定版本构建的 BuildConfig 日志

您可以使用 Web 控制台或 CLI 访问 **BuildConfig** 的给定版本构建的日志。

流程

- 要输出 **BuildConfig** 的给定版本构建的日志，请输入以下命令：

```
$ oc logs --version=<number> bc/<buildconfig_name>
```

2.7.6.3. 启用日志详细程度

您可以传递 **BUILD_LOGLEVEL** 环境变量作为 **BuildConfig** 中 **sourceStrategy** 或 **dockerStrategy** 的一部分，来实现更为详细的输出。

注意

管理员可以通过配置 **env/BUILD_LOGLEVEL**，为整个 OpenShift Container Platform 实例设置默认的构建详细程度。此默认值可以通过在给定的 **BuildConfig** 中指定 **BUILD_LOGLEVEL** 来覆盖。您可以通过将 **--build-loglevel** 传递给 **oc start-build**，在命令行中为非二进制构建指定优先级更高的覆盖。

源构建的可用日志级别如下：

0 级	生成运行 **assemble** 脚本的容器的输出，以及所有遇到的错误。这是默认值。
1 级	生成有关已执行进程的基本信息。
2 级	生成有关已执行进程的非常详细的信息。
3 级	生成有关已执行进程的非常详细的信息，以及存档内容的列表。
4 级	目前生成与 3 级相同的信息。
5 级	生成以上级别中包括的所有内容，另外还提供 Docker 推送消息。

流程

- 要启用更为详细的输出，请传递 **BUILD_LOGLEVEL** 环境变量作为 **BuildConfig** 中 **sourceStrategy** 或 **dockerStrategy** 的一部分：

```
sourceStrategy:
...
  env:
    - name: "BUILD_LOGLEVEL"
      value: "2" 1
```

1 将此值调整为所需的日志级别。

2.8. 触发和修改构建

以下小节概述了如何使用构建 hook 触发构建和修改构建。

2.8.1. 构建触发器

在定义 **BuildConfig** 时，您可以定义触发器来控制应该运行 **BuildConfig** 的环境。可用的构建触发器如下：

- Webhook
- 镜像更改
- 配置更改

2.8.1.1. Webhook 触发器

Webhook 触发器通过发送请求到 OpenShift Container Platform API 端点来触发新构建。您可以使用 GitHub、GitLab、Bitbucket 或通用 Webhook 来定义这些触发器。

目前，OpenShift Container Platform Webhook 仅支持各种基于 Git 的源代码管理系统 (SCM) 的推送事件的类同版本。所有其他事件类型都会忽略。

处理推送事件时，OpenShift Container Platform control plane 主机确认事件内的分支引用是否与对应 **BuildConfig** 中的分支引用匹配。如果匹配，它会检查 OpenShift Container Platform 构建的 Webhook 事件中记录的确切提交引用。如果不匹配，则不触发构建。

注意

oc new-app 和 **oc new-build** 会自动创建 GitHub 和通用 Webhook 触发器，但其他所需的 Webhook 触发器都必须手动添加。您可以通过设置触发器来手动添加触发器。

对于所有 Webhook，您必须使用名为 **WebHookSecretKey** 的键定义 secret，并且其值是调用 Webhook 时要提供的值。然后，Webhook 定义必须引用该 secret。secret可确保 URL 的唯一性，防止他人触发构建。键的值将与 Webhook 调用期间提供的 secret 进行比较。

例如，此处的 GitHub Webhook 具有对名为 **mysecret** 的 secret 的引用：

```
type: "GitHub"
github:
  secretReference:
    name: "mysecret"
```

该 secret 的定义如下。注意 secret 的值采用 base64 编码，如 **Secret** 对象的 **data** 字段所要求。

```
- kind: Secret
  apiVersion: v1
  metadata:
    name: mysecret
    creationTimestamp:
  data:
    WebHookSecretKey: c2VjcmV0dmFsdWUx
```

2.8.1.1.1. 将未经身份验证的用户添加到 system:webhook 角色绑定

作为集群管理员，您可以为特定命名空间将未经身份验证的用户添加到 OpenShift Container Platform 中的 **system:webhook** 角色绑定中。**system:webhook** 角色绑定允许用户触发不使用 OpenShift Container Platform 身份验证机制的外部系统的构建。默认情况下，未经身份验证的用户无法访问非公共角色绑定。这是在 4.16 之前的 OpenShift Container Platform 版本的变化。

需要将未经身份验证的用户添加到 **system:webhook** 角色绑定，才能成功触发从 GitHub、GitLab 和 Bitbucket 的构建。

如果需要允许未经身份验证的用户访问集群，您可以通过将未经身份验证的用户添加到每个所需命名空间中的 **system:webhook** 角色绑定来完成此操作。这个方法比将未经身份验证的用户添加到 **system:webhook** 集群角色绑定更安全。但是，如果您有大量命名空间，可以将未经身份验证的用户添加到 **system:webhook** 集群角色绑定中，该绑定会将更改应用到所有命名空间。

重要

在修改未经身份验证的访问时，始终验证符合您机构的安全标准。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
- 已安装 OpenShift CLI(**oc**)。

流程

1. 创建名为 **add-webhooks-unauth.yaml** 的 YAML 文件，并添加以下内容：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  name: webhook-access-unauthenticated
  namespace: <namespace> (1)
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: "system:webhook"
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: Group
    name: "system:unauthenticated"
```

(1) **BuildConfig** 的命名空间。

2. 运行以下命令来应用配置：

```
$ oc apply -f add-webhooks-unauth.yaml
```

其他资源

- 未经身份验证的组的集群角色绑定

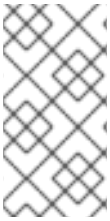
2.8.1.1.2. 使用 GitHub Webhook

当存储库更新时，GitHub Webhook 处理 GitHub 发出的调用。在定义触发器时，您必须指定一个 secret，它将是您在配置 Webhook 时提供给 GitHub 的 URL 的一部分。

GitHub Webhook 定义示例：

```
type: "GitHub"
```

```
github:
  secretReference:
    name: "mysecret"
```



注意

Webhook 触发器配置中使用的 secret 与在 GitHub UI 中配置 Webhook 时遇到的 **secret** 字段不同。Webhook 触发器配置中的 secret 使 Webhook URL 是唯一的，且难以预测。GitHub UI 中配置的 secret 是一个可选字符串字段，用于创建正文的 HMAC 十六进制摘要，该摘要作为 **X-Hub-Signature** 标头发送。

oc describe 命令将有效负载 URL 返回为 GitHub Webhook URL（请参阅"显示 Webhook URL"），其结构如下：

输出示例

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github
```

先决条件

- 从 GitHub 存储库创建 **BuildConfig**。
- **system:unauthenticated** 在所需命名空间中可以访问 **system:webhook** 角色。或者，**system:unauthenticated** 能够访问 **system:webhook** 集群角色。

流程

1. 配置 GitHub Webhook。
   a. 从 GitHub 存储库创建 **BuildConfig** 对象后，运行以下命令：

   ```
   $ oc describe bc/<name_of_your_BuildConfig>
   ```

   此命令生成 webhook GitHub URL。

   输出示例

   ```
   https://api.starter-us-east-1.openshift.com:443/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github
   ```

   b. 从 GitHub Web 控制台将此 URL 剪切并粘贴到 GitHub 中。

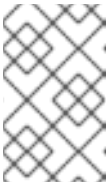
   c. 在 GitHub 存储库中，从 **Settings → Webhooks** 中选择 **Add Webhook**。

   d. 将 URL 输出粘贴到 **Payload URL** 字段。

   e. 将 **Content Type** 从 GitHub 默认的 **application/x-www-form-urlencoded** 更改为 **application/json**。

   f. 点击 **Add webhook**。
   您应该看到一条来自 GitHub 的消息，说明您的 Webhook 已配置成功。

现在，每当您将更改推送到 GitHub 存储库时，新构建会自动启动，成功构建后也会启动新部署。



注意

Gogs 支持与 GitHub 相同的 Webhook 有效负载格式。因此，如果您使用的是 Gogs 服务器，也可以在 **BuildConfig** 中定义 GitHub Webhook 触发器，并由 Gogs 服务器触发它。

2. 假设包含有效 JSON 有效负载的文件，如 **payload.json**，您可以使用以下 **curl** 命令手动触发 Webhook：

```
$ curl -H "X-GitHub-Event: push" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/github
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

注意

只有 GitHub Webhook 事件的 **ref** 值与 **BuildConfig** 资源中的 **source.git** 字段中指定的 **ref** 值匹配时，才会触发构建。

其他资源

- Gogs

2.8.1.1.3. 使用 GitLab Webhook

当存储库更新时，GitLab Webhook 处理 GitLab 发出的调用。与 GitHub 触发器一样，您必须指定一个 secret。以下示例是 **BuildConfig** 中的触发器定义 YAML：

```
type: "GitLab"
gitlab:
  secretReference:
    name: "mysecret"
```

oc describe 命令将有效负载 URL 返回为 GitLab Webhook URL，其结构如下：

输出示例

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/gitlab
```

先决条件

- **system:unauthenticated** 在所需命名空间中可以访问 **system:webhook** 角色。或者，**system:unauthenticated** 能够访问 **system:webhook** 集群角色。

流程

1. 配置 GitLab Webhook。

a. 输入以下命令来获取 Webhook URL：

```
$ oc describe bc <name>
```

b. 复制 Webhook URL，将 **<secret>** 替换为您的 secret 值。

c. 按照 GitLab 设置说明，将 Webhook URL 粘贴到 GitLab 存储库设置中。

2. 假设包含有效 JSON 有效负载的文件，如 **payload.json**，您可以使用以下 **curl** 命令手动触发 Webhook：

```
$ curl -H "X-GitLab-Event: Push Hook" -H "Content-Type: application/json" -k -X POST --
data-binary @payload.json
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildcon
gs/<name>/webhooks/<secret>/gitlab
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

2.8.1.1.4. 使用 Bitbucket Webhook

当存储库更新时，Bitbucket Webhook 处理 Bitbucket 发出的调用。与 GitHub 和 GitLab 触发器类似，您必须指定一个 secret。以下示例是 **BuildConfig** 中的触发器定义 YAML：

```
type: "Bitbucket"
bitbucket:
  secretReference:
    name: "mysecret"
```

oc describe 命令将有效负载 URL 返回为 Bitbucket Webhook URL，其结构如下：

输出示例

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<na
me>/webhooks/<secret>/bitbucket
```

先决条件

- **system:unauthenticated** 在所需命名空间中可以访问 **system:webhook** 角色。或者，**system:unauthenticated** 能够访问 **system:webhook** 集群角色。

流程

1. 配置 Bitbucket Webhook。

a. 输入以下命令来获取 Webhook URL：

```
$ oc describe bc <name>
```

b. 复制 Webhook URL，将 **<secret>** 替换为您的 secret 值。

c. 按照 Bitbucket 设置说明，将 Webhook URL 粘贴到 Bitbucket 存储库设置中。

2. 假设包含有效 JSON 有效负载的文件，如 **payload.json**，您可以使用以下 **curl** 命令手动触发 Webhook：

```
$ curl -H "X-Event-Key: repo:push" -H "Content-Type: application/json" -k -X POST --data-binary @payload.json
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/bitbucket
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

2.8.1.1.5. 使用通用 Webhook

通用 Webhook 可从能够发出 Web 请求的任何系统调用。与其他 Webhook一样，您必须指定一个 secret，该 secret 将成为调用者必须用于触发构建的 URL 的一部分。secret可确保 URL 的唯一性，防止他人触发构建。如下是 **BuildConfig** 中的示例触发器定义 YAML：

```
type: "Generic"
generic:
  secretReference:
    name: "mysecret"
  allowEnv: true 1
```

1 设置为 **true**，以允许通用 Webhook 传入环境变量。

流程

1. 要设置调用者，请为调用系统提供构建的通用 Webhook 端点的 URL：

   通用 webhook 端点 URL 示例

   ```
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
   ```

   调用者必须以 **POST** 操作的形式调用 Webhook。

2. 要手动调用 webhook，请输入以下 **curl** 命令：

   ```
   $ curl -X POST -k
   https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
   ```

   HTTP 操作动词必须设置为 **POST**。指定了不安全 **-k** 标志以忽略证书验证。如果集群拥有正确签名的证书，则不需要此第二个标志。

   端点可以接受具有以下格式的可选有效负载：

   ```
   git:
     uri: "<url to git repository>"
     ref: "<optional git reference>"
     commit: "<commit hash identifying a specific git commit>"
     author:
       name: "<author name>"
       email: "<author e-mail>"
     committer:
       name: "<committer name>"
       email: "<committer e-mail>"
   ```

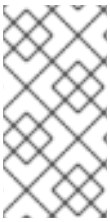
```
  message: "<commit message>"
env: 1
  - name: "<variable name>"
    value: "<variable value>"
```

1 与 **BuildConfig** 环境变量类似，此处定义的环境变量也可供您的构建使用。如果这些变量与 **BuildConfig** 环境变量发生冲突，则以这些变量为准。默认情况下，Webhook 传递的环境变量将被忽略。在 Webhook 定义上将 **allowEnv** 字段设为 **true** 即可启用此行为。

3. 要使用 **curl** 传递此有效负载，请在名为 **payload_file.yaml** 的文件中进行定义，并运行以下命令：

```
$ curl -H "Content-Type: application/yaml" --data-binary @payload_file.yaml -X POST -k https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic
```

参数与前一个示例相同，但添加了标头和 payload。**-H** 参数将 **Content-Type** 标头设置为 **application/yaml** 或 **application/json**，具体取决于您的 payload 格式。**--data-binary** 参数用于通过 **POST** 请求发送带有换行符的二进制 payload。



注意

即使出示了无效的请求 payload（例如，无效的内容类型，或者无法解析或无效的内容等），OpenShift Container Platform 也允许通用 Webhook 触发构建。保留此行为是为了向后兼容。如果出示无效的请求 payload，OpenShift Container Platform 将以 JSON 格式返回警告，作为其 **HTTP 200 OK** 响应的一部分。

2.8.1.1.6. 显示 Webhook URL

您可以使用 **oc describe** 命令显示与构建配置关联的 Webhook URL。如果命令没有显示任何 Webhook URL，则目前不会为该构建配置定义任何 Webhook 触发器。

流程

- 要显示与 **BuildConfig** 关联的任何 Webhook URL，请运行以下命令：

```
$ oc describe bc <name>
```

2.8.1.2. 使用镜像更改触发器

作为开发人员，您可以将构建配置为在每次基础镜像更改时自动运行。

当上游镜像有新版本可用时，您可以使用镜像更改触发器自动调用构建。例如，如果构建基于 RHEL 镜像，您可以触发该构建在 RHEL 镜像更改时运行。因此，应用程序镜像始终在最新的 RHEL 基础镜像上运行。

注意

指向 v1 容器 registry 中的容器镜像的镜像流仅在镜像流标签可用时触发一次构建，后续镜像更新时则不会触发。这是因为 v1 容器 registry 中缺少可唯一标识的镜像。

流程

1. 定义指向您要用作触发器的上游镜像的 **ImageStream** ：

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby-20-centos7"
```

这将定义绑定到位于 ***<system-registry>/<namespace>*/ruby-20-centos7** 的容器镜像存储库的镜像流。**<system-registry>** 定义为 OpenShift Container Platform 中运行的名为 **docker-registry** 的服务。

2. 如果镜像流是构建的基础镜像，请将构建策略中的 **from** 字段设置为指向 **ImageStream**：

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
```

在这种情形中，**sourceStrategy** 定义将消耗此命名空间中名为 **ruby-20-centos7** 的镜像流的 **latest** 标签。

3. 使用指向 **ImageStreams** 的一个或多个触发器定义构建：

```
type: "ImageChange" (1)
imageChange: {}
type: "ImageChange" (2)
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
```

(1) 监控构建策略的 **from** 字段中定义的 **ImageStream** 和 **Tag** 的镜像更改触发器。此处的 **imageChange** 对象必须留空。

(2) 监控任意镜像流的镜像更改触发器。此时 **imageChange** 部分必须包含一个 **from** 字段，以引用要监控的 **ImageStreamTag**。

将镜像更改触发器用于策略镜像流时，生成的构建会获得一个不可变 docker 标签，指向与该标签对应的最新镜像。在执行构建时，策略会使用此新镜像引用。

对于不引用策略镜像流的其他镜像更改触发器，系统会启动新构建，但不会使用唯一镜像引用来更新构建策略。

由于此示例具有策略的镜像更改触发器，因此生成的构建将是：

```
strategy:
  sourceStrategy:
    from:
      kind: "DockerImage"
      name: "172.30.17.3:5001/mynamespace/ruby-20-centos7:<immutableid>"
```

这将确保触发的构建使用刚才推送到存储库的新镜像，并且可以使用相同的输入随时重新运行构建。

您可以暂停镜像更改触发器，以便在构建开始之前对引用的镜像流进行多次更改。在将 **ImageChangeTrigger** 添加到 **BuildConfig** 时，您也可以将 **paused** 属性设为 true，以避免立即触发构建。

```
type: "ImageChange"
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
  paused: true
```

除了设置适用于所有 **Strategy** 类型的镜像字段外，自定义构建还需要检查 **OPENSHIFT_CUSTOM_BUILD_BASE_IMAGE** 环境变量。如果不存在，则使用不可变镜像引用来创建它。如果存在，则使用不可变镜像引用进行更新。

如果因为 Webhook 触发器或手动请求而触发构建，则创建的构建将使用从 **Strategy** 引用的 **ImageStream** 解析而来的 **<immutableid>**。这将确保使用一致的镜像标签来执行构建，以方便再生。

其他资源

- v1 容器 registry

2.8.1.3. 识别构建的镜像更改触发器

作为开发人员，如果您有镜像更改触发器，您可以识别启动了上一次构建的镜像更改。这对于调试或故障排除构建非常有用。

BuildConfig示例

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: bc-ict-example
  namespace: bc-ict-example-namespace
spec:

# ...

  triggers:
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input:latest
        namespace: bc-ict-example-namespace
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input2:latest
        namespace: bc-ict-example-namespace
    type: ImageChange
status:
  imageChangeTriggers:
  - from:
      name: input:latest
      namespace: bc-ict-example-namespace
```

```
    lastTriggerTime: "2021-06-30T13:47:53Z"
    lastTriggeredImageID: image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input@sha256:0f88ffbeb9d25525720bfa3524cb1bf0908b7f791057cf1acfae917b11266a69

  - from:
      name: input2:latest
      namespace: bc-ict-example-namespace
    lastTriggeredImageID:  image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input2@sha256:0f88ffbeb9d25525720bfa3524cb2ce0908b7f791057cf1acfae917b11266a6
9

  lastVersion: 1
```

注意

本例省略了与镜像更改触发器无关的元素。

先决条件

- 您已配置了多个镜像更改触发器。这些触发器已触发一个或多个构建。

流程

1. 在 **BuildConfig** CR 中，在 **status.imageChangeTriggers** 中，标识具有最新时间戳的 **lastTriggerTime**。
   这个 **ImageChangeTriggerStatus**

   > Then you use the `name` and `namespace` from that build to find the corresponding image change trigger in `buildConfig.spec.triggers`.

2. 在 **UnderimageChangeTriggers** 下，比较时间戳以标识最新的

镜像更改触发器

在构建配置中，**buildConfig.spec.triggers** 是构建触发器策略 **BuildTriggerPolicy** 的数组。

每个 **BuildTriggerPolicy** 都有 **type** 字段和指针字段。每个指针字段对应于 **type** 字段允许的值之一。因此，您只能将 **BuildTriggerPolicy** 设置为一个指针字段。

对于镜像更改触发器，**type** 的值为 **ImageChange**。然后，**imageChange** 字段是指向 **ImageChangeTrigger** 对象的指针，其具有以下字段：

- **lastTriggeredImageID**：此字段在 OpenShift Container Platform 4.8 中已弃用，并将在以后的发行版本中被忽略。它包含从此 **BuildConfig** 触发最后一次构建时的 **ImageStreamTag** 的已解析镜像引用。
- **paused**：您可以使用此字段（示例中未显示）暂时禁用此特定镜像更改触发器。
- **from** ：使用此字段引用驱动此镜像更改触发器的 **ImageStreamTag**。其类型是核心 Kubernetes 类型 **OwnerReference**。

from 字段有以下字段：

- **kind** ：对于镜像更改触发器，唯一支持的值是 **ImageStreamTag**。

- **namespace**：使用此字段指定 **ImageStreamTag** 的命名空间。
- **name**：使用此字段指定 **ImageStreamTag**。

镜像更改触发器状态

在构建配置中，**buildConfig.status.imageChangeTriggers** 是 **ImageChangeTriggerStatus** 元素的数组。每个 **ImageChangeTriggerStatus** 元素都包含上例中所示的 **from**、**lastTriggeredImageID** 和 **lastTriggerTime** 元素。

具有最新 **lastTriggerTime** 的 **ImageChangeTriggerStatus** 触发了最新的构建。您可以使用其 **name** 和 **namespace** 来识别触发构建的 **buildConfig.spec.triggers** 中的镜像更改触发器。

带有最新时间戳的 **lastTriggerTime** 表示最后一个构建的 **ImageChangeTriggerStatus**。此 **ImageChangeTriggerStatus** 的**name** 和 **namespace** 与触发构建的 **buildConfig.spec.triggers** 中的镜像更改触发器相同。

其他资源

- v1 容器 registry

2.8.1.4. 配置更改触发器

通过配置更改触发器，您可以在创建新 **BuildConfig** 时立即自动调用构建。

如下是 **BuildConfig** 中的示例触发器定义 YAML：

```
type: "ConfigChange"
```

注意

配置更改触发器目前仅在创建新 **BuildConfig** 时运作。在未来的版本中，配置更改触发器也可以在每当 **BuildConfig** 更新时启动构建。

2.8.1.4.1. 手动设置触发器

您可以使用 **oc set triggers** 在构建配置中添加和移除触发器。

流程

- 要在构建配置上设置 GitHub Webhook 触发器，请输入以下命令：

```
$ oc set triggers bc <name> --from-github
```

- 要设置镜像更改触发器，请输入以下命令：

```
$ oc set triggers bc <name> --from-image='<image>'
```

- 要删除触发器，请输入以下命令：

```
$ oc set triggers bc <name> --from-bitbucket --remove
```

注意

如果 Webhook 触发器已存在，再次添加它会重新生成 Webhook secret。

如需更多信息，请输入以下命令查阅帮助文档：

```
$ oc set triggers --help
```

2.8.2. 构建 hook

通过构建 hook，可以将行为注入到构建过程中。

BuildConfig 对象的 **postCommit** 字段在运行构建输出镜像的临时容器内执行命令。Hook 的执行时间是紧接在提交镜像的最后一层后，并且在镜像推送到 registry 之前。

当前工作目录设置为镜像的 **WORKDIR**，即容器镜像的默认工作目录。对于大多数镜像，这是源代码所处的位置。

如果脚本或命令返回非零退出代码，或者启动临时容器失败，则 hook 将失败。当 hook 失败时，它会将构建标记为失败，并且镜像也不会推送到 registry。可以通过查看构建日志来检查失败的原因。

构建 hook 可用于运行单元测试，以在构建标记为完成并在 registry 中提供镜像之前验证镜像。如果所有测试都通过并且测试运行器返回退出代码 **0**，则构建标记为成功。如果有任何测试失败，则构建标记为失败。在所有情况下，构建日志将包含测试运行器的输出，这可用于识别失败的测试。

postCommit hook 不仅限于运行测试，也可用于运行其他命令。由于它在临时容器内运行，因此 hook 所做的更改不会持久存在；也就是说，hook 执行无法对最终镜像造成影响。除了其他用途外，也可借助此行为来安装和使用会自动丢弃并且不出现在最终镜像中的测试依赖项。

2.8.2.1. 配置提交后构建 hook

配置构建后 hook 的方法有多种。以下示例中所有形式具有同等作用，也都执行 **bundle exec rake test --verbose**。

流程

- 使用以下选项之一配置构建后 hook：

选项	描述

选项	描述

shell 脚本	`postCommit:`
	`  script: "bundle exec rake test --verbose"`
	
	script 值是通过 **/bin/sh -ic** 执行的 shell 脚本。当 shell 脚本适合执行构建 hook 时，请使用此选项。例如，用于运行前文所述的单元测试。要控制镜像入口点，或者镜像没有 **/bin/sh**，使用 **command**, 或 **args**（或两个都使用）。
	
	 
	注意
	
	引入的额外 **-i** 标志用于改进搭配 CentOS 和 RHEL 镜像时的体验，未来的发行版中可能会剔除。

作为镜像入口点的命令	`postCommit:`
	`  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]`
	
	在这种形式中，**command** 是要运行的命令，它会覆盖 exec 形式中的镜像入口点，如 Dockerfile 引用中所述。如果镜像没有 **/bin/sh**，或者您不想使用 shell，则需要这样做。在所有其他情形中，使用 **script** 可能更为方便。

带有参数的命令	`postCommit:`
	`  command: ["bundle", "exec", "rake", "test"]`
	`  args: ["--verbose"]`
	
	这种形式相当于将参数附加到 **command**。

注意

同时提供 **script** 和 **command** 会产生无效的构建 hook。

2.8.2.2. 使用 CLI 设置提交后构建 hook

oc set build-hook 命令可用于为构建配置设置构建 hook。

流程

1. 完成以下操作之一：

- 要将命令设置为 post-commit 构建 hook，请输入以下命令：

```
$ oc set build-hook bc/mybc \
    --post-commit \
    --command \
    -- bundle exec rake test --verbose
```

- 要将脚本设置为提交后构建 hook，请输入以下命令：

```
$ oc set build-hook bc/mybc --post-commit --script="bundle exec rake test --verbose"
```

2.9. 执行高级构建

您可以设置构建资源和最长持续时间，将构建分配给节点、链构建、修剪构建和配置构建运行策略。

2.9.1. 设置构建资源

默认情况下，构建由 Pod 使用未绑定的资源（如内存和 CPU）来完成。这些资源可能会有限制。

流程

您可以以两种方式限制资源使用：

- 通过在项目的默认容器限值中指定资源限值来限制资源使用。
- 通过在构建配置中指定资源限值来限制资源使用。
  - 在以下示例中，每个 **resources**、**cpu** 和 **memory** 参数都是可选的。

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  resources:
    limits:
      cpu: "100m" 1
      memory: "256Mi" 2
```

1. **cpu** 以 CPU 单元数为单位：**100m** 表示 0.1 个 CPU 单元（100 * 1e-3）。
2. **memory** 以字节为单位：**256Mi** 表示 268435456 字节 (256 * 2 ^ 20)。

不过，如果您的项目定义了配额，则需要以下两项之一：

- 设定了显式 **requests** 的 **resources** 部分：

```
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1. **requests** 对象包含与配额中资源列表对应的资源列表。

- 项目中定义的限值范围，其中 **LimitRange** 对象中的默认值应用到构建过程中创建的 Pod。
  否则，构建 Pod 创建将失败，说明无法满足配额要求。

2.9.2. 设置最长持续时间

定义 **BuildConfig** 对象时，您可以通过设置 **completionDeadlineSeconds** 字段来定义其最长持续时间。以秒为单位指定，默认情况下不设置。若未设置，则不强制执行最长持续时间。

最长持续时间从构建 Pod 调度到系统中的时间开始计算，并且定义它在多久时间内处于活跃状态，这包括拉取构建器镜像所需的时间。达到指定的超时后，OpenShift Container Platform 将终止构建。

流程

- 要设置最长持续时间，请在 **BuildConfig** 中指定 **completionDeadlineSeconds**。下例显示了 **BuildConfig** 的部分内容，它指定了值为 30 分钟的 **completionDeadlineSeconds** 字段：

```
spec:
  completionDeadlineSeconds: 1800
```

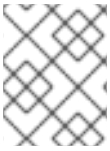
注意

Pipeline 策略选项不支持此设置。

2.9.3. 将构建分配给特定的节点

通过在构建配置的 **nodeSelector** 字段中指定标签，可以将构建定位到在特定节点上运行。**nodeSelector** 值是一组键值对，在调度构建 pod 时与 **Node** 标签匹配。

nodeSelector 值也可以由集群范围的默认值和覆盖值控制。只有构建配置没有为 nodeSelector 定义任何键值对，也没有为 **nodeSelector :{}** 定义显式的空映射值，才会应用默认值。覆盖值将逐个键地替换构建配置中的值。



注意

如果指定的 **NodeSelector** 无法与具有这些标签的节点匹配，则构建仍将无限期地保持在 **Pending** 状态。

流程

- 通过在 **BuildConfig** 的 **nodeSelector** 字段中指定标签，将构建分配到特定的节点上运行，如下例所示：

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  nodeSelector: (1)
    key1: value1
    key2: value2
```

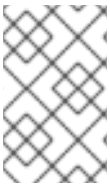
(1) 与此构建配置关联的构建将仅在具有 **key1=value2** 和 **key2=value2** 标签的节点上运行。

2.9.4. 串联构建

对于编译语言（例如 Go、C、C ++ 和 Java），在应用程序镜像中包含编译所需的依赖项可能会增加镜像的大小，或者引入可被利用的漏洞。

为避免这些问题，可以将两个构建串联在一起。一个生成编译工件的构建，另一个构建将工件放置在运行工件的独立镜像中。

在以下示例中，Source-to-Image（S2I）构建与 Docker 构建相结合，以编译工件并将其置于单独的运行时镜像中。



注意

虽然本例串联了 Source-to-Image（S2I） 构建和 Docker 构建，但第一个构建可以使用任何策略来生成包含所需工件的镜像，第二个构建则可以使用任何策略来消耗镜像中的输入内容。

第一个构建获取应用程序源，并生成含有 **WAR** 文件的镜像。镜像推送到 **artifact-image** 镜像流。输出工件的路径取决于使用的 S2I 构建器的 **assemble** 脚本。在这种情况下，它会输出到 **/wildfly/standalone/deployments/ROOT.war**。

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: artifact-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: artifact-image:latest
  source:
    git:
      uri: https://github.com/openshift/openshift-jee-sample.git
      ref: "master"
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
```

第二个构建使用路径指向第一个构建中输出镜像内的 WAR 文件的镜像源。内联 **dockerfile** 将该 **WAR** 文件复制到运行时镜像中。

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: image-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: image-build:latest
  source:
    dockerfile: |-
```

```
    FROM jee-runtime:latest
    COPY ROOT.war /deployments/ROOT.war
  images:
  - from: 1
      kind: ImageStreamTag
      name: artifact-image:latest
    paths: 2
    - sourcePath: /wildfly/standalone/deployments/ROOT.war
      destinationDir: "."
strategy:
  dockerStrategy:
    from: 3
      kind: ImageStreamTag
      name: jee-runtime:latest
triggers:
- imageChange: {}
  type: ImageChange
```

1. **from** 指定 docker 构建应包含来自 **artifact-image** 镜像流的镜像输出，而这是上一个构建的目标。
2. **paths** 指定要在当前 docker 构建中包含目标镜像的哪些路径。
3. 运行时镜像用作 docker 构建的源镜像。

此设置的结果是，第二个构建的输出镜像不需要包含创建 **WAR** 文件所需的任何构建工具。此外，由于第二个构建包含镜像更改触发器，因此每当运行第一个构建并生成含有二进制工件的新镜像时，将自动触发第二个构建，以生成包含该工件的运行时镜像。所以，两个构建表现为一个具有两个阶段的构建。

2.9.5. 修剪构建

默认情况下，生命周期已结束的构建将无限期保留。您可以限制要保留的旧构建数量。

流程

1. 通过为 **BuildConfig** 中的 **successfulBuildsHistoryLimit** 或 **failedBuildsHistoryLimit** 提供正整数值，限制要保留的旧构建的数量，如下例中所示：

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  successfulBuildsHistoryLimit: 2 1
  failedBuildsHistoryLimit: 2 2
```

1. **successfulBuildsHistoryLimit** 将保留最多两个状态为 **completed** 的构建。
2. **failedBuildsHistoryLimit** 将保留最多两个状态为 **failed**、**cancelled** 或 **error** 的构建。

2. 通过以下操作之一来触发构建修剪：

- 更新构建配置。

- 等待构建结束其生命周期。

构建按其创建时间戳排序，首先修剪最旧的构建。

注意

管理员可以使用 oc adm 对象修剪命令来手动修剪构建。

2.9.6. 构建运行策略

构建运行策略描述从构建配置创建的构建应运行的顺序。这可以通过更改 **Build** 规格的 **spec** 部分中的 **runPolicy** 字段的值来完成。

还可以通过以下方法更改现有构建配置的 **runPolicy** 值：

- 如果将 **Parallel** 改为 **Serial** 或 **SerialLatestOnly**，并从此配置触发新构建，这会导致新构建需要等待所有并行构建完成，因为串行构建只能单独运行。
- 如果将 **Serial** 更改为 **SerialLatestOnly** 并触发新构建，这会导致取消队列中的所有现有构建，但当前正在运行的构建和最近创建的构建除外。最新的构建接下来运行。

2.10. 在构建中使用红帽订阅

按照以下部分的内容，在 OpenShift Container Platform 构建中安装红帽订阅内容。

2.10.1. 为红帽通用基础镜像创建镜像流标签

要在构建中安装 Red Hat Enterprise Linux (RHEL) 软件包，您可以创建一个镜像流标签来引用 Red Hat Universal Base Image (UBI)。

要让 UBI 在集群中的**每个项目**中都可用，您需要将镜像流标签添加到 **openshift** 命名空间中。否则，若要使其**在一个特定项目中**可用，您要将镜像流标签添加到该项目。

镜像流标签使用安装 pull secret 中的 **registry.redhat.io** 凭证授予对 UBI 的访问权限，而无需向其他用户公开 pull secret。这个方法要求每个开发人员使用项目中的 **registry.redhat.io** 凭证安装 pull secret 更为方便。

流程

- 要在 **openshift** 命名空间中创建 **ImageStreamTag** 资源，以便在所有项目中都可以使用它，请输入以下命令：

```
$ oc tag --source=docker registry.redhat.io/ubi9/ubi:latest ubi9:latest -n openshift
```

提示

您还可以应用以下 YAML 在 **openshift** 命名空间中创建 **ImageStreamTag** 资源：

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: ubi9
  namespace: openshift
spec:
  tags:
  - from:
      kind: DockerImage
      name: registry.redhat.io/ubi9/ubi:latest
    name: latest
    referencePolicy:
      type: Source
```

- 要在单个项目中创建 **ImageStreamTag** 资源，请输入以下命令：

```
$ oc tag --source=docker registry.redhat.io/ubi9/ubi:latest ubi:latest
```

提示

您还可以应用以下 YAML 在单个项目中创建 **ImageStreamTag** 资源：

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: ubi9
spec:
  tags:
  - from:
      kind: DockerImage
      name: registry.redhat.io/ubi9/ubi:latest
    name: latest
    referencePolicy:
      type: Source
```

2.10.2. 将订阅权利添加为构建 secret

使用红帽订阅安装内容的构建需要包括做为一个构件 secret 的权利密钥。

先决条件

- 您可以通过您的订阅访问 Red Hat Enterprise Linux (RHEL) 软件包仓库。当集群已正确订阅时，Insights Operator 会自动创建用于访问这些仓库的权利 secret。
- 您必须可以使用具有 **cluster-admin** 角色的用户访问集群，或者具有访问 **openshift-config-managed** 项目中的 secret 的权限。

流程

1. 输入以下命令将 **openshift-config-managed** 命名空间中的授权 secret 复制到构建的命名空间：

```
$ cat << EOF > secret-template.txt
kind: Secret
apiVersion: v1
metadata:
  name: etc-pki-entitlement
type: Opaque
data: {{ range \$key, \$value := .data }}
  {{ \$key }}: {{ \$value }} {{ end }}
EOF
$ oc get secret etc-pki-entitlement -n openshift-config-managed -o=go-template-file --
template=secret-template.txt | oc apply -f -
```

2. 在构建配置的 Docker 策略中将 etc-pki-entitlement secret 添加为构建卷：

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi9:latest
    volumes:
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
        type: Secret
        secret:
          secretName: etc-pki-entitlement
```

2.10.3. 使用 Subscription Manager 运行构建

2.10.3.1. 使用 Subscription Manager 执行 Docker 构建

Docker 策略构建可以使用 **yum** 或 **dnf** 安装其他 Red Hat Enterprise Linux (RHEL)软件包。

先决条件

- 必须将授权密钥添加为构建策略卷。

流程

- 使用以下示例 Dockerfile 来通过 Subscription Manager 安装内容：

```
FROM registry.redhat.io/ubi9/ubi:latest
RUN rm -rf /etc/rhsm-host (1)
RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
   nss_wrapper \
   uid_wrapper -y && \
   yum clean all -y
RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
```

1. 在执行任何 **yum** 或 **dnf** 命令前，您必须包含删除 **/etc/rhsm-host** 目录及其所有内容的命令。
2. 使用红帽软件包浏览器查找已安装软件包的正确存储库。
3. 您需要恢复 **/etc/rhsm-host** 符号链接，以便使您的镜像与其他红帽容器镜像兼容。

2.10.4. 使用 Red Hat Satellite 订阅运行构建

2.10.4.1. 将 Red Hat Satellite 配置添加到构建中

使用 Red Hat Satellite 安装内容的构建必须提供适当的配置，以便从 Satellite 存储库获取内容。

先决条件

- 您必须提供或创建与 **yum** 兼容的存储库配置文件，该文件将从 Satellite 实例下载内容。

  仓库配置示例

```
[test-<name>]
name=test-<number>
baseurl = https://satellite.../content/dist/rhel/server/7/7Server/x86_64/os
enabled=1
gpgcheck=0
sslverify=0
sslclientkey = /etc/pki/entitlement/...-key.pem
sslclientcert = /etc/pki/entitlement/....pem
```

流程

1. 输入以下命令来创建包含 Satellite 存储库配置文件的 **ConfigMap** 对象：

```
$ oc create configmap yum-repos-d --from-file /path/to/satellite.repo
```

2. 将 Satellite 存储库配置和授权密钥添加为构建卷：

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi9:latest
    volumes:
    - name: yum-repos-d
      mounts:
      - destinationPath: /etc/yum.repos.d
      source:
        type: ConfigMap
        configMap:
          name: yum-repos-d
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
```

```
    type: Secret
    secret:
      secretName: etc-pki-entitlement
```

2.10.4.2. 使用 Red Hat Satellite 订阅构建 Docker

Docker 策略构建可以使用 Red Hat Satellite 软件仓库来安装订阅内容。

先决条件

- 您已将授权密钥和 Satellite 存储库配置添加为构建卷。

流程

- 使用以下示例为使用 Satellite 安装内容创建 **Dockerfile** ：

```
FROM registry.redhat.io/ubi9/ubi:latest
RUN rm -rf /etc/rhsm-host (1)
RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
    nss_wrapper \
    uid_wrapper -y && \
    yum clean all -y
RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
```

(1) 在执行任何 **yum** 或 **dnf** 命令前，您必须包含删除 **/etc/rhsm-host** 目录及其所有内容的命令。

(2) 请联系您的 Satellite 系统管理员，以查找构建安装的软件包的正确仓库。

(3) 您需要恢复 **/etc/rhsm-host** 符号链接，以便使您的镜像与其他红帽容器镜像兼容。

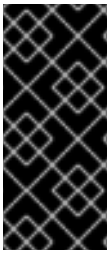
其他资源

- 如何使用 Red Hat Satellite 订阅和使用哪个证书进行构建

2.10.5. 使用 SharedSecret 对象运行构建

您可以使用 **SharedSecret** 对象安全地访问构建中的集群的授权密钥。

SharedSecret 对象允许您在命名空间间共享和同步 secret。



重要

共享资源 CSI 驱动程序只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

先决条件

- 已使用功能门启用 **TechPreviewNoUpgrade** 功能集。如需更多信息，请参阅 使用功能门启用功能。

- 您必须具有执行以下操作的权限：
  - 创建构建配置和启动构建。
  - 输入 **oc get sharedsecrets** 命令并返回非空列表来发现哪些 **SharedSecret** CR 实例可用。
  - 确定命名空间中的 **builder** 服务帐户是否可以使用给定的 **SharedSecret** CR 实例。换句话说，您可以运行 **oc adm policy who-can use <identifier of specific SharedSecret>** 来查看是否列出命名空间中的 **builder** 服务帐户。

注意

如果没有满足此列表中的最后两个先决条件，则建立或询问某人建立所需的基于角色的访问控制(RBAC)，以便您可以发现 **SharedSecret** CR 实例，并启用服务帐户使用 **SharedSecret** CR 实例。

流程

1. 使用 **oc apply** 创建带有集群权利 secret 的 **SharedSecret** 对象实例。

重要

您必须具有集群管理员权限来创建 **SharedSecret** 对象。

使用带有 YAML 角色对象定义的 oc apply -f 命令示例

```
$ oc apply -f - <<EOF
kind: SharedSecret
apiVersion: sharedresource.openshift.io/v1alpha1
metadata:
  name: etc-pki-entitlement
spec:
  secretRef:
    name: etc-pki-entitlement
    namespace: openshift-config-managed
EOF
```

2. 创建一个角色来授予 **builder** 服务帐户权限来访问 **SharedSecret** 对象：

oc apply -f 命令示例

```
$ oc apply -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: builder-etc-pki-entitlement
  namespace: build-namespace
rules:
  - apiGroups:
      - sharedresource.openshift.io
    resources:
      - sharedsecrets
    resourceNames:
      - etc-pki-entitlement
```

```
    verbs:
      - use
EOF
```

3. 运行以下命令，创建一个 **RoleBinding** 对象，授予 **builder** 服务帐户权限来访问 **SharedSecret** 对象：

oc create rolebinding 命令示例

```
$ oc create rolebinding builder-etc-pki-entitlement --role=builder-etc-pki-entitlement --
serviceaccount=build-namespace:builder
```

4. 使用 CSI 卷挂载将授权 secret 添加到 **BuildConfig** 对象中：

YAML BuildConfig 对象定义示例

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: uid-wrapper-rhel9
  namespace: build-namespace
spec:
  runPolicy: Serial
  source:
    dockerfile: |
      FROM registry.redhat.io/ubi9/ubi:latest
      RUN rm -rf /etc/rhsm-host (1)
      RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
          nss_wrapper \
          uid_wrapper -y && \
          yum clean all -y
      RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
  strategy:
    type: Docker
    dockerStrategy:
      volumes:
        - mounts:
            - destinationPath: "/etc/pki/entitlement"
          name: etc-pki-entitlement
          source:
            csi:
              driver: csi.sharedresource.openshift.io
              readOnly: true (4)
              volumeAttributes:
                sharedSecret: etc-pki-entitlement (5)
            type: CSI
```

(1) 在执行任何 **yum** 或 **dnf** 命令前，您必须包含删除 **/etc/rhsm-host** 目录及其所有内容的命令。

(2) 使用红帽软件包浏览器查找已安装软件包的正确存储库。

(3) 您需要恢复 **/etc/rhsm-host** 符号链接，以便使您的镜像与其他红帽容器镜像兼容。

4. 您必须将 **readOnly** 设置为 **true**，以便在构建中挂载共享资源。
5. 引用构建中包含的 **SharedSecret** 对象的名称。

5. 从 **BuildConfig** 对象启动构建，并使用 **oc** 命令跟踪日志。

```
$ oc start-build uid-wrapper-rhel9 -n build-namespace -F
```

2.10.6. 其他资源

- 使用 Insights Operator 导入简单的内容访问证书
- 使用功能门启用功能
- 管理镜像流
- 构建策略

2.11. 通过策略保护构建

OpenShift Container Platform 中的构建在特权容器中运行。根据所用的构建策略，如果您有权限，可以运行构建来升级其在集群和主机节点上的权限。为安全起见，请限制可以运行构建的人员以及用于这些构建的策略。Custom 构建本质上不如 Source 构建安全，因为它们可以在特权容器内执行任何代码，这在默认情况下是禁用的。请谨慎授予 docker 构建权限，因为 Dockerfile 处理逻辑中的漏洞可能会导致在主机节点上授予特权。

默认情况下，所有能够创建构建的用户都被授予相应的权限，可以使用 docker 和 Source-to-Image (S2I) 构建策略。具有集群管理员特权的用户可启用自定义构建策略,如在全局范围内限制用户使用构建策略部分中所述。

您可以使用授权策略来控制谁能够构建以及他们可以使用哪些构建策略。每个构建策略都有一个对应的构建子资源。用户必须有权创建构建，并在构建策略子资源上创建构建的权限，才能使用该策略创建构建。提供的默认角色用于授予构建策略子资源的 create 权限。

表 2.3. 构建策略子资源和角色

策略	子资源	角色
Docker	builds/docker	system:build-strategy-docker
Source-to-Image	builds/source	system:build-strategy-source
Custom	builds/custom	system:build-strategy-custom
JenkinsPipeline	builds/jenkinspipeline	system:build-strategy-jenkinspipeline

2.11.1. 在全局范围内禁用构建策略访问

要在全局范围内阻止对特定构建策略的访问，请以具有集群管理源权限的用户身份登录，从 **system:authenticated** 组中移除对应的角色，再应用注解 **rbac.authorization.kubernetes.io/autoupdate: "false"** 以防止它们在 API 重启后更改。以下示例演示了

如何禁用 Docker 构建策略。

流程

1. 输入以下命令应用 **rbac.authorization.kubernetes.io/autoupdate** 注解：

```
$ oc edit clusterrolebinding system:build-strategy-docker-binding
```

输出示例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "false" (1)
  creationTimestamp: 2018-08-10T01:24:14Z
  name: system:build-strategy-docker-binding
  resourceVersion: "225"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Abuild-strategy-
docker-binding
  uid: 17b1f3d4-9c3c-11e8-be62-0800277d20bf
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:build-strategy-docker
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
```

(1) 将 **rbac.authorization.kubernetes.io/autoupdate** 注解的值更改为 **"false"**。

2. 输入以下命令删除角色：

```
$ oc adm policy remove-cluster-role-from-group system:build-strategy-docker
system:authenticated
```

3. 确保也从这些角色中移除构建策略子资源：

```
$ oc edit clusterrole admin
```

```
$ oc edit clusterrole edit
```

4. 对于每个角色，指定与要禁用的策略资源对应的子资源。

   a. 为 **admin** 禁用 docker Build 策略：

```
kind: ClusterRole
metadata:
  name: admin
...
- apiGroups:
  - ""
```

```
  - build.openshift.io
  resources:
  - buildconfigs
  - buildconfigs/webhooks
  - builds/custom 1
  - builds/source
  verbs:
  - create
  - delete
  - deletecollection
  - get
  - list
  - patch
  - update
  - watch
...
```

1 添加 **builds/custom** 和 **builds/source**，以在全局范围内为具有 admin 角色的用户禁用 docker 构建。

2.11.2. 在全局范围内限制用户使用构建策略

您可以允许某一组用户使用特定策略来创建构建。

先决条件

- 禁用构建策略的全局访问。

流程

- 将与构建策略对应的角色分配给特定用户。例如，将 **system:build-strategy-docker** 集群角色添加到用户 **devuser**：

```
$ oc adm policy add-cluster-role-to-user system:build-strategy-docker devuser
```

警告

如果在集群级别授予用户对 **builds/docker** 子资源的访问权限，那么该用户将能够在他们可以创建构建的任何项目中使用 docker 策略来创建构建。

2.11.3. 在项目范围内限制用户使用构建策略

与在全局范围内向用户授予构建策略角色类似，您只能允许项目中的某一组特定用户使用特定策略来创建构建。

先决条件

- 禁用构建策略的全局访问。

流程

- 将与构建策略对应的角色分配给项目中的特定用户。例如，将 **devproject** 项目中的 **system:build-strategy-docker** 角色添加到用户 **devuser**：

```
$ oc adm policy add-role-to-user system:build-strategy-docker devuser -n devproject
```

2.12. 构建配置资源

使用以下步骤来配置构建设置。

2.12.1. 构建控制器配置参数

build.config.openshift.io/cluster 资源提供以下配置参数。

参数	描述
Build	包含有关如何处理构建的集群范围内信息。规范且唯一有效的名称是 **cluster**。 **spec**：包含构建控制器配置的用户可设置值。
buildDefaults	控制构建的默认信息。 **defaultProxy**：包含所有构建操作的默认代理设置，包括镜像拉取或推送以及源代码下载。 您可以通过设置 **BuildConfig** 策略中的 **HTTP_PROXY**、**HTTPS_PROXY** 和 **NO_PROXY** 环境变量来覆盖值。 **gitProxy**：仅包含 Git 操作的代理设置。如果设置，这将覆盖所有 Git 命令的任何代理设置，例如 **git clone**。 此处未设置的值将从 DefaultProxy 继承。 **env**：一组应用到构建的默认环境变量，条件是构建中不存在指定的变量。 **imageLabels**：应用到生成的镜像的标签列表。您可以通过在 **BuildConfig** 中提供具有相同名称的标签来覆盖默认标签。 **resources**：定义执行构建的资源要求。
ImageLabel	**name**：定义标签的名称。它必须具有非零长度。
buildOverrides	控制构建的覆盖设置。 **imageLabels**：应用到生成的镜像的标签列表。如果您在 **BuildConfig** 中提供了与此表中名称相同的标签，您的标签将会被覆盖。 **nodeSelector**：一个选择器，必须为 true 才能使构建 Pod 适合节点。 **tolerations**：一个容忍度列表，覆盖构建 Pod 上设置的现有容忍度。
BuildList	**items**：标准对象的元数据。

2.12.2. 配置构建设置

您可以通过编辑 **build.config.openshift.io/cluster** 资源来配置构建设置。

流程

- 输入以下命令来编辑 **build.config.openshift.io/cluster** 资源：

```
$ oc edit build.config.openshift.io/cluster
```

以下是 **build.config.openshift.io/cluster** 资源的示例：

```
apiVersion: config.openshift.io/v1
kind: Build (1)
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 2
  name: cluster
  resourceVersion: "107233"
  selfLink: /apis/config.openshift.io/v1/builds/cluster
  uid: e2e9cc14-78a9-11e9-b92b-06d6c7da38dc
spec:
  buildDefaults: (2)
    defaultProxy: (3)
      httpProxy: http://proxy.com
      httpsProxy: https://proxy.com
      noProxy: internal.com
    env: (4)
    - name: envkey
      value: envvalue
    gitProxy: (5)
      httpProxy: http://gitproxy.com
      httpsProxy: https://gitproxy.com
      noProxy: internalgit.com
    imageLabels: (6)
    - name: labelkey
      value: labelvalue
    resources: (7)
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 10m
        memory: 10Mi
  buildOverrides: (8)
    imageLabels: (9)
    - name: labelkey
      value: labelvalue
    nodeSelector: (10)
      selectorkey: selectorvalue
    tolerations: (11)
```

```
    - effect: NoSchedule
      key: node-role.kubernetes.io/builds
  operator: Exists
```

1. **Build**：包含有关如何处理构建的集群范围内信息。规范且唯一有效的名称是 **cluster**。
2. **buildDefaults**：控制构建的默认信息。
3. **defaultProxy**：包含所有构建操作的默认代理设置，包括镜像拉取或推送以及源代码下载。
4. **env**：一组应用到构建的默认环境变量，条件是构建中不存在指定的变量。
5. **gitProxy**：仅包含 Git 操作的代理设置。如果设置，这将覆盖所有 Git 命令的任何代理设置，例如 **git clone**。
6. **imageLabels**：应用到生成的镜像的标签列表。您可以通过在 **BuildConfig** 中提供具有相同名称的标签来覆盖默认标签。
7. **resources**：定义执行构建的资源要求。
8. **buildOverrides**：控制构建的覆盖设置。
9. **imageLabels**：应用到生成的镜像的标签列表。如果您在 **BuildConfig** 中提供了与此表中名称相同的标签，您的标签将会被覆盖。
10. **nodeSelector**：一个选择器，必须为 true 才能使构建 Pod 适合节点。
11. **tolerations**：一个容忍度列表，覆盖构建 Pod 上设置的现有容忍度。

2.13. 构建故障排除

使用以下内容来排除构建问题。

2.13.1. 解决资源访问遭到拒绝的问题

如果您的资源访问请求遭到拒绝：

问题

构建失败并显示以下信息：

```
requested access to the resource is denied
```

解决方案

您已超过项目中设置的某一镜像配额。检查当前的配额，并验证应用的限值和正在使用的存储：

```
$ oc describe quota
```

2.13.2. 服务证书生成失败

如果您的资源访问请求遭到拒绝：

问题

如果服务证书生成失败并显示以下信息（ 服务的 **service.beta.openshift.io/serving-cert-generation-error** 注解包含）：

输出示例

```
secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not match 77b6dd80-d716-11e6-9d6f-0e9c0057b60
```

解决方案

生成证书的服务不再存在，或者具有不同的 **serviceUID** 。您必须删除旧 secret 并清除服务上的以下注解 **service.beta.openshift.io/serving-cert-generation-error** 和 **service.beta.openshift.io/serving-cert-generation-error-num** 来强制重新生成证书。要清除注解，请输入以下命令：

```
$ oc delete secret <secret_name>
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-
```

注意

在用于移除注解的命令中，要移除的注解后面有一个 **-**。

2.14. 为构建设置其他可信证书颁发机构

在从镜像 registry 中拉取镜像时，参照以下部分设置构建可信任的额外证书颁发机构 (CA) 。

此流程要求集群管理员创建 **ConfigMap**，并在 **ConfigMap** 中添加额外的 CA 作为密钥。

- **ConfigMap** 必须在 **openshift-config** 命名空间中创建。
- **domain** 是 **ConfigMap** 中的键，**value** 是 PEM 编码的证书。
  - 每个 CA 必须与某个域关联。域格式是 **hostname[..port]**。
- **ConfigMap** 名称必须在 **image.config.openshift.io/cluster** 集群范围配置资源的 **spec.additionalTrustedCA** 字段中设置。

2.14.1. 在集群中添加证书颁发机构

您可以按照以下流程将证书颁发机构 (CA) 添加到集群，以便在推送和拉取镜像时使用。

先决条件

- 您必须有权访问 registry 的公共证书，通常是位于 **/etc/docker/certs.d/** 目录中的 **hostname/ca.crt** 文件。

流程

1. 在 **openshift-config** 命名空间中创建一个 **ConfigMap**，其中包含使用自签名证书的 registry 的可信证书。对于每个 CA 文件，确保 **ConfigMap** 中的键是 **hostname[..port]** 格式的容器镜像仓库的主机名：

```
$ oc create configmap registry-cas -n openshift-config \
--from-file=myregistry.corp.com..5000=/etc/docker/certs.d/myregistry.corp.com:5000/ca.crt \
--from-file=otherregistry.com=/etc/docker/certs.d/otherregistry.com/ca.crt
```

2. 更新集群镜像配置：

```
$ oc patch image.config.openshift.io/cluster --patch '{"spec":{"additionalTrustedCA":
{"name":"registry-cas"}}}' --type=merge
```

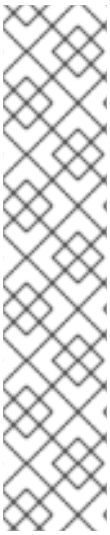
2.14.2. 其他资源

- 创建一个 **ConfigMap**
- Secrets 和 **ConfigMaps**
- 配置自定义 PKI

第 3 章 PIPELINES

3.1. 关于 RED HAT OPENSHIFT PIPELINES

Red Hat OpenShift Pipelines 是一个基于 Kubernetes 资源的云原生的持续集成和持续交付（continuous integration and continuous delivery，简称 CI/CD）的解决方案。它通过提取底层实现的详情，使用 Tekton 构建块进行跨多个平台的自动部署。Tekton 引入了多个标准自定义资源定义 (CRD)，用于定义可跨 Kubernetes 分布的 CI/CD 管道。



注意

因为 Red Hat OpenShift Pipelines 的发行节奏与 OpenShift Container Platform 不同，所以 Red Hat OpenShift Pipelines 文档现在为每个产品的次版本提供单独的文档。

Red Hat OpenShift Pipelines 文档包括在 https://docs.openshift.com/pipelines/。

特定版本的文档使用版本选择器下拉列表，或者直接将版本添加到 URL，例如 https://docs.openshift.com/pipelines/1.11。

另外，Red Hat OpenShift Pipelines 文档也包括在红帽客户门户网站 https://access.redhat.com/documentation/zh-cn/red_hat_openshift_pipelines/ 中。

如需有关 Red Hat OpenShift Pipelines 生命周期和支持的平台的更多信息，请参阅平台生命周期政策。

第 4 章 GITOPS

4.1. 关于 RED HAT OPENSHIFT GITOPS

Red Hat OpenShift GitOps 是一个使用 Argo CD 作为声明性 GitOps 引擎的 Operator。它启用了多集群 OpenShift 和 Kubernetes 基础架构的 GitOps 工作流。使用 Red Hat OpenShift GitOps，管理员可以在集群和开发生命周期中一致地配置和部署基于 Kubernetes 的基础架构和应用程序。Red Hat OpenShift GitOps 基于开源项目 Argo CD，为上游提供的功能提供类似的功能，并提供额外的自动化功能，集成到 Red Hat {OCP} 中，以及 Red Hat Enterprise 支持、质量保证和专注于企业安全性的好处。

注意

因为 Red Hat OpenShift GitOps 的发行节奏与 {OCP} 不同，所以 Red Hat OpenShift GitOps 文档现在作为每个产品的次版本提供单独的文档。

Red Hat OpenShift GitOps 文档包括在 https://docs.openshift.com/gitops/。

特定版本的文档使用版本选择器下拉列表，或者直接将版本添加到 URL，例如 https://docs.openshift.com/gitops/1.8。

另外，Red Hat OpenShift GitOps 文档也包括在 Red Hat Portal https://access.redhat.com/documentation/zh-cn/red_hat_openshift_gitops/ 中。

如需有关 Red Hat OpenShift GitOps 生命周期和支持的平台的更多信息，请参阅平台生命周期政策。

当应用程序部署到不同环境中的不同集群时，Red Hat OpenShift GitOps 可确保应用程序的一致性，如开发、临时和生产环境。Red Hat OpenShift GitOps 整理与配置仓库相关的部署过程，并将其作为核心元素。它总会保持至少有两个软件仓库：

1. 源代码的应用程序仓库
2. 定义应用程序所需状态的环境配置仓库

这些软件仓库包含您指定环境中所需的基础架构声明信息。它们还包含可让您的环境与上述状态匹配的自动过程。

Red Hat OpenShift GitOps 使用 Argo CD 来维护集群资源。Argo CD 是一个开源声明工具，用于应用程序的持续集成和持续部署（CI/CD）。Red Hat OpenShift GitOps 将 Argo CD 实现作为一个控制器，以便持续监控 Git 存储库中定义的应用程序定义和配置。然后，Argo CD 将这些配置的指定状态与集群中的实时状态进行比较。

Argo CD 报告与指定状态不同的配置。报告允许管理员自动或者手动将配置重新同步到定义的状态。因此，Argo CD 允许您提供全局自定义资源，如用于配置 {OCP} 集群的资源。

4.1.1. 主要特性

Red Hat OpenShift GitOps 可帮助您自动执行以下任务：

- 确保集群具有类似的配置、监控和存储状态
- 对多个 {OCP} 集群应用或恢复配置更改
- 将模板配置与不同环境关联

- 在集群间（从调试到生产阶段）推广应用程序。

4.1.2. 其他资源

- 使用自定义资源定义来扩展 Kubernetes API
- 管理自定义资源定义中的资源
- 什么是 GitOps？

第 5 章 JENKINS

5.1. 配置 JENKINS 镜像

OpenShift Container Platform 为运行 Jenkins 提供容器镜像。此镜像提供 Jenkins 服务器实例，可用于为连续测试、集成和交付设置基本流程。

该镜像基于 Red Hat Universal Base Images（UBI）。

OpenShift Container Platform 遵从 Jenkins 的 LTS 的发行版本。OpenShift Container Platform 提供一个包含 Jenkins 2.x 的镜像。

OpenShift Container Platform Jenkins 镜像在 Quay.io 或 registry.redhat.io 上提供。

例如：

```
$ podman pull registry.redhat.io/ocp-tools-4/jenkins-rhel8:<image_tag>
```

要使用这些镜像，您可直接从这些 registry 访问镜像或将其推送（push）到 OpenShift Container Platform 容器镜像 registry 中。另外，您还可在容器镜像 registry 或外部位置创建一个指向镜像的镜像流。然后，OpenShift Container Platform 资源便可引用镜像流。

但为方便起见,OpenShift Container Platform 会在 **openshift** 命名空间中为核心 Jenkins 镜像以及针对 OpenShift Container Platform 与 Jenkins 集成提供的示例代理镜像提供镜像流。

5.1.1. 配置和自定义

您可采用两种方式管理 Jenkins 身份验证：

- OpenShift Container Platform OAuth 身份验证由 OpenShift Container Platform Login 插件提供。
- 由 Jenkins 提供的标准身份验证。

5.1.1.1. OpenShift Container Platform OAuth 身份验证

OAUTH 身份验证激活方法：配置 Jenkins UI 中 **Configure Global Security** 面板上的选项，或者将 Jenkins **Deployment configuration** 上的 **OPENSHIFT_ENABLE_OAUTH** 环境变量设置为非 **false**。这会激活 OpenShift Container Platform Login 插件，该插件从 Pod 数据或通过与 OpenShift Container Platform API 服务器交互来检索配置信息。

有效凭证由 OpenShift Container Platform 身份提供程序控制。

Jenkins 支持浏览器和非浏览器访问。

登录时，有效用户会自动添加到 Jenkins 授权列表中，其中的 OpenShift Container Platform 角色规定了用户拥有的特定 Jenkins 权限。默认使用的角色是预定义的 **admin**、**edit** 和 **view**。登录插件对 Jenkins 正在其中运行的项目或命名空间中的那些角色执行自身 SAR 请求。

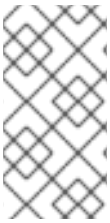
具有 **Admin** 角色的用户拥有传统 Jenkins 管理用户权限，而具有 **edit** 或 **view** 角色的用户的权限逐渐减少。

默认的 OpenShift Container Platform **admin**、**edit** 和 **view** 角色以及这些角色在 Jenkins 实例中分配的 Jenkins 权限均可配置。

在 OpenShift Container Platform pod 中运行 Jenkins 时，登录插件会在 Jenkins 正在其中运行的命名空间中查找名为 **openshift-jenkins-login-plugin-config** 的配置映射。

如果该插件找到并可以在该配置映射中读取，您可以定义到 Jenkins 权限映射的角色。具体来说：

- 登录插件将配置映射中的键值对视为 Jenkins 权限到 OpenShift Container Platform 角色映射。
- 其中，键是 Jenkins 权限组短 ID 和 Jenkins 权限短 ID，两者之间用连字符隔开。
- 如果要向 OpenShift Container Platform 角色添加 **Overall Jenkins Administer** 权限，键应为 **Overall-Administer**。
- 要了解有哪些权限组和权限 ID 可用，请转至 Jenkins 控制台中的列表授权页，并在它们提供的表中查找组 ID 和个别权限。
- 键值对的值是权限应当应用到的 OpenShift Container Platform 角色的列表，各个角色之间用逗号隔开。
- 如果要将 **Overall Jenkins Administer** 权限添加到默认的 **admin** 和 **edit** 角色以及您创建的新 Jenkins 角色，则 **Overall-Administer** 键的值将为 **admin,edit,jenkins**。



注意

使用 OpenShift Container Platform OAuth 时，OpenShift Container Platform Jenkins 镜像中预填充了管理特权的 **admin** 用户不会被授予这些特权。要授予这些权限，OpenShift Container Platform 集群管理员必须在 OpenShift Container Platform 身份提供程序中显式定义该用户，并为该用户分配 **admin** 角色。

最初建立用户后，可对存储的 Jenkins 用户权限进行更改。OpenShift Container Platform Login 插件轮询 OpenShift Container Platform API 服务器以获取权限，并使用从 OpenShift Container Platform 检索的权限更新存储在 Jenkins 中的每个用户的权限。如果 Jenkins UI 用于为 Jenkins 用户更新权限，则权限更改将在插件下次轮询 OpenShift Container Platform 时被覆盖。

您可通过 **OPENSHIFT_permissions_poll_interval** 环境变量来控制轮询频率。默认轮询间隔为五分钟。

使用 Oauth 身份验证创建新的 Jenkins 服务的最简单方式是借助模板。

5.1.1.2. Jenkins 身份验证

如果镜像未使用模板直接运行，则默认使用 Jenkins 身份验证。

Jenkins 首次启动时，配置与管理员用户和密码一同创建。默认用户凭证为 **admin** 和 **password**。在使用标准 Jenkins 身份验证时，且仅这种情况下，通过设置 **JENKINS_PASSWORD** 环境变量来配置默认密码。

流程

- 输入以下命令来创建使用标准 Jenkins 身份验证的 Jenkins 应用程序：

```
$ oc new-app -e \
    JENKINS_PASSWORD=<password> \
    ocp-tools-4/jenkins-rhel8
```

5.1.2. Jenkins 环境变量

Jenkins 服务器可通过以下环境变量进行配置：

变量	定义	值和设置示例
OPENSHIFT_ENABLE_OAUTH	决定在登录 Jenkins 时，OpenShift Container Platform Login 插件可否管理身份验证。要启用，请设为 **true**。	默认：**false**
JENKINS_PASSWORD	使用标准 Jenkins 身份验证时 **admin** 用户的密码。**OPENSHIFT_ENABLE_OAUTH** 设置为 **true** 时不适用。	默认：**password**
JAVA_MAX_HEAP_PARAM、CONTAINER_HEAP_PERCENT、JENKINS_MAX_HEAP_UPPER_BOUND_MB	这些值控制 Jenkins JVM 的最大堆大小。如果设置了 **JAVA_MAX_heap_PARAM**，则优先使用其值。否则，最大堆大小将动态计算为容器内存限值的 **CONTAINER_HEAP_PERCENT**，可选上限为 **JENKINS_MAX_HEAP_UPPER_BOUND_MB** MiB。 默认情况下，Jenkins JVM 的最大堆大小设置为容器内存限值的 50%，且无上限。	**Java_MAX_heap_PARAM** 示例设置：**-Xmx512m** **container_heap_percent** 默认：**0.5** 或 50% **Jenkins_MAX_heap_UPPER_BOUND_MB** 示例设置：**512 MiB**
JAVA_INITIAL_HEAP_PARAM、CONTAINER_INITIAL_PERCENT	这些值控制 Jenkins JVM 的初始堆大小。如果设置了 **JAVA_INITIAL_heap_PARAM**，则优先使用其值。否则，初始堆大小将动态计算为动态计算的最大堆大小的 **CONTAINER_INITIAL_PERCENT**。 默认情况下，JVM 设置初始堆大小。	**java_INITIAL_heap_PARAM** 示例设置：**-Xms32m** **container_INITIAL_percent** 示例设置：**0.1**或 10%
CONTAINER_CORE_LIMIT	如果设置，请将用于调整内部 JVM 线程数的内核数指定为整数。	示例设置：**2**
JAVA_TOOL_OPTIONS	指定应用于该容器中运行的所有 JVM 的选项。不建议覆盖该值。	默认：**-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true**

变量	定义	值和设置示例
JAVA_GC_OPTS	指定 Jenkins JVM 垃圾回收参数。不建议覆盖该值。	默认：**-XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90**
JENKINS_JAVA_OVERRIDES	指定适用于 Jenkins JVM 的附加选项。这些选项附加到所有其他选项中，包括上面的 Java 选项，必要时可用于覆盖其中任何一个选项。用空格分开各个附加选项；如有任意选项包含空格字符，请使用反斜杠转义。	示例设置：**-Dfoo -Dbar; -Dfoo=first\ value -Dbar=second\ value**。
JENKINS_OPTS	为 Jenkins 指定参数。	
INSTALL_PLUGINS	指定在容器首次运行或 **OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS** 设置为 **true** 时需要安装的 Jenkins 附加插件。插件被指定为用逗号分隔的“名称:版本”对列表。	示例设置：**git:3.7.0,subversion:2.10.2**。
OPENSHIFT_PERMISSIONS_POLL_INTERVAL	指定 OpenShift Container Platform Login 插件轮询 OpenShift Container Platform 的时间间隔（以毫秒为单位），以获取与 Jenkins 中定义的每个用户关联的权限。	默认：**300000** - 5 分钟
OVERRIDE_PV_CONFIG_WITH_IMAGE_CONFIG	当使用 Jenkins 配置目录的 OpenShift Container Platform 持久性卷（PV）运行此镜像时，从镜像到 PV 的配置传输仅在镜像首次启动时执行，因为在创建持久性卷声明（PVC）时会分配 PV。如果您在初始启动后创建自定义镜像来扩展此镜像并更新自定义镜像中的配置，则不会复制该配置，除非将该环境变量设置为 **true**。	默认：**false**

变量	定义	值和设置示例
OVERRIDE_PV_PLUGINS_WITH_IMAGE_PLUGINS	当使用 Jenkins 配置目录的 OpenShift Container Platform PV 运行此镜像时，从镜像到 PV 的插件传输仅在镜像首次启动时执行，因为在创建 PVC 时会分配 PV。如果您在初始启动后创建自定义镜像来扩展此镜像并更新自定义镜像中的插件，则不会复制该插件，除非将该环境变量设置为 **true**。	默认：**false**
ENABLE_FATAL_ERROR_LOG_FILE	当使用 Jenkins 配置目录的 OpenShift Container Platform PVC 运行此镜像时，该环境变量允许在严重错误发生时，严重错误日志文件保留。严重错误文件保存在：**/var/lib/jenkins/logs**。	默认：**false**
AGENT_BASE_IMAGE	设置此值将覆盖使用此镜像提供的 Kubernetes 插件 pod 模板中的 **jnlp** 容器的镜像。否则，会使用 **openshift** 命名空间中的 **jenkins-agent-base-rhel8:latest** 镜像流标签中的镜像。	默认：**image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest**
JAVA_BUILDER_IMAGE	设置此值将覆盖使用此镜像提供的 **java-builder** 示例 Kubernetes 插件 Pod 模板中用于 **java-builder** 容器的镜像。否则，会使用 **openshift** 命名空间中的 **java:latest** 镜像流标签的镜像。	默认：**image-registry.openshift-image-registry.svc:5000/openshift/java:latest**
JAVA_FIPS_OPTIONS	设置此值控制 JVM 在 FIPS 节点上运行的运行时如何运行。如需更多信息，请参阅在 FIPS 模式中配置红帽构建的 OpenJDK 11。	默认：**-Dcom.redhat.fips=false**

5.1.3. 向 Jenkins 提供跨项目访问权限

如果要在与您的项目不同的其他位置运行 Jenkins，则必须向 Jenkins 提供访问令牌来访问您的项目。

流程

1. 输入以下命令识别具有适当权限访问 Jenkins 必须访问的项目的服务帐户的 secret：

```
$ oc describe serviceaccount jenkins
```

输出示例

```
Name:       default
Labels:     <none>
Secrets:    {  jenkins-token-uyswp    }
            {  jenkins-dockercfg-xcr3d    }
Tokens:     jenkins-token-izv1u
            jenkins-token-uyswp
```

这种情况下，secret 被命名为 **jenkins-token-extensionswp**。

2. 输入以下命令从 secret 检索令牌：

```
$ oc describe secret <secret name from above>
```

输出示例

```
Name:       jenkins-token-uyswp
Labels:     <none>
Annotations:    kubernetes.io/service-account.name=jenkins,kubernetes.io/service-
account.uid=32f5b661-2a8f-11e5-9528-3c970e3bf0b7
Type:   kubernetes.io/service-account-token
Data
====
ca.crt: 1066 bytes
token:  eyJhbGc..<content cut>....wRA
```

令牌参数包含 Jenkins 访问项目所需的令牌值。

5.1.4. Jenkins 跨卷挂载点

可使用挂载卷运行 Jenkins 镜像，以便为配置启用持久性存储：

- **/var/lib/jenkins** 是 Jenkins 存储配置文件的数据目录，包括任务定义。

5.1.5. 通过 Source-to-image 自定义 Jenkins 镜像

要自定义官方 OpenShift Container Platform Jenkins 镜像，您可以使用该镜像作为 Source-to-image（S2I）构建程序。

您可使用 S2I 来复制自定义 Jenkins 任务定义，添加其它插件，或使用您自己的自定义配置来替换所提供的 **config.xml** 文件。

要在 Jenkins 镜像中包括您的修改，必须要有采用以下目录结构的 Git 存储库：

plugins

该目录包含要复制到 Jenkins 中的二进制 Jenkins 插件。

plugins.txt

该文件使用以下语法列出要安装的插件:

```
pluginId:pluginVersion
```

configuration/jobs

该目录包含 Jenkins 任务定义。

configuration/config.xml

该文件包含您的自定义 Jenkins 配置。

configuration/ 目录的内容会被复制到 **/var/lib/jenkins/** 目录中，以便也可以包括其他文件，如 **credentials.xml**。

在 OpenShift Container Platform 中自定义 Jenkins 镜像的构建配置示例

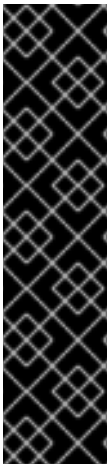
```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: custom-jenkins-build
spec:
  source:                    1
    git:
      uri: https://github.com/custom/repository
    type: Git
  strategy:                  2
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: jenkins:2
        namespace: openshift
    type: Source
  output:                    3
    to:
      kind: ImageStreamTag
      name: custom-jenkins:latest
```

1. **source** 参数使用上述布局定义源 Git 存储库。
2. **strategy** 参数定义用作构建的源镜像的原始 Jenkins 镜像。
3. **output** 参数定义可用于部署配置的生成自定义 Jenkins 镜像，而非官方 Jenkins 镜像。

5.1.6. 配置 Jenkins Kubernetes 插件

OpenShift Jenkins 镜像包含预安装的用于 Jenkins 的 Kubernetes 插件，以便使用 Kubernetes 和 OpenShift Container Platform 在多个容器主机上动态置备 Jenkins 代理。

为了使用 Kubernetes 插件，OpenShift Container Platform 提供了一个适合用作 Jenkins 代理的 OpenShift Agent Base 镜像。



重要

OpenShift Container Platform 4.11 将 OpenShift Jenkins 和 OpenShift Agent Base 镜像移到 **registry.redhat.io** 的 **ocp-tools-4** 仓库中，以便红帽可以在 OpenShift Container Platform 生命周期外生成和更新镜像。在以前的版本中，这些镜像位于 OpenShift Container Platform 安装有效负载以及 **registry.redhat.io** 的 **openshift4** 存储库中。

OpenShift Jenkins Maven 和 NodeJS Agent 镜像已从 OpenShift Container Platform 4.11 有效负载中删除。红帽不再生成这些镜像，它们不能从 **registry.redhat.io** 的 **ocp-tools-4** 存储库中提供。根据 OpenShift Container Platform 生命周期政策，红帽会维护这些镜像的 4.10 及更早的版本，适用于任何重要的程序错误修复或安全 CVE。

如需更多信息，请参阅以下"添加资源"部分中的"重要更改 OpenShift Jenkins 镜像"链接。

Maven 和 Node.js 代理镜像均会在 OpenShift Container Platform Jenkins 镜像的 Kubernetes 插件配置中自动配置为 Kubernetes Pod 模板镜像。该配置包含每个镜像的标签，您可以在 **Restrict where this project can run** 设置下应用到任何 Jenkins 任务。如果应用了标签，任务将在运行相应代理镜像的 OpenShift Container Platform Pod 下运行。

重要

在 OpenShift Container Platform 4.10 及更新的版本中，使用 Kubernetes 插件运行 Jenkins 代理的建议模式是使用带有 **jnlp** 和 **sidecar** 容器的 pod 模板。**jnlp** 容器使用 OpenShift Container Platform Jenkins Base 代理镜像来简化为构建启动一个单独的 pod。**sidecar** 容器镜像具有在启动的独立 pod 中的特定语言构建所需的工具。Red Hat Container Catalog 的许多容器镜像在 **openshift** 命名空间中的示例镜像流中引用。OpenShift Container Platform Jenkins 镜像有一个名为 **java-build** 的 pod 模板，它带有演示此方法的 sidecar 容器。此 pod 模板使用由 **openshift** 命名空间中的 **java** 镜像流提供的最新 Java 版本。

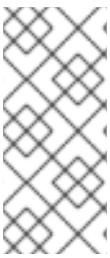
Jenkins 镜像还为 Kubernetes 插件提供附加代理镜像的自动发现和自动配置。

在 OpenShift Container Platform 同步插件中，在 Jenkins 启动时，Jenkins 镜像会在其运行的项目中搜索，或者插件配置中列出的项目，用于以下项目：

- 将 **role** 标签设置为 **jenkins-agent** 的镜像流。
- 将 **role** 注解设置为 **jenkins-agent** 的镜像流标签。
- 将 **role** 标签设置为 **jenkins-agent** 的配置映射。

当 Jenkins 镜像找到具有适当标签的镜像流或带有适当注解的镜像流标签时，它会生成对应的 Kubernetes 插件配置。这样，您可以将 Jenkins 任务分配到运行镜像流提供的容器镜像的 pod 中运行。

镜像流或镜像流标签的名称和镜像引用映射到 Kubernetes 插件 pod 模板中的名称和镜像字段。您可以通过使用 **agent-label** 键在镜像流或镜像流标签对象上设置注解来控制 Kubernetes 插件 pod 模板的标签字段。否则，名称将用作标签。



注意

不要登录到 Jenkins 控制台并更改 pod 模板配置。如果您在创建 pod 模板后这样做，且 OpenShift Container Platform Sync 插件检测到与镜像流或镜像流标签关联的镜像已更改，则该插件会替换 pod 模板并覆盖这些配置更改。您无法将新配置与现有配置合并。

如果您的配置需要更复杂的配置需求，请考虑配置映射方法。

当找到具有适当标签的配置映射时，Jenkins 镜像会假定配置映射的键值数据有效负载中的任何值都与 Jenkins 和 Kubernetes 插件 pod 模板的配置格式一致。配置映射与镜像流和镜像流标签的一个关键优点是，您可以控制所有 Kubernetes 插件 pod 模板参数。

jenkins-agent 的配置映射示例

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-agent
data:
  template1: |-
    <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
      <inheritFrom></inheritFrom>
      <name>template1</name>
      <instanceCap>2147483647</instanceCap>
      <idleMinutes>0</idleMinutes>
      <label>template1</label>
      <serviceAccount>jenkins</serviceAccount>
      <nodeSelector></nodeSelector>
      <volumes/>
      <containers>
        <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
          <name>jnlp</name>
          <image>openshift/jenkins-agent-maven-35-centos7:v3.10</image>
          <privileged>false</privileged>
          <alwaysPullImage>true</alwaysPullImage>
          <workingDir>/tmp</workingDir>
          <command></command>
          <args>${computer.jnlpmac} ${computer.name}</args>
          <ttyEnabled>false</ttyEnabled>
          <resourceRequestCpu></resourceRequestCpu>
          <resourceRequestMemory></resourceRequestMemory>
          <resourceLimitCpu></resourceLimitCpu>
          <resourceLimitMemory></resourceLimitMemory>
          <envVars/>
        </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
      </containers>
      <envVars/>
      <annotations/>
      <imagePullSecrets/>
      <nodeProperties/>
    </org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```

以下示例显示了引用 **openshift** 命名空间中镜像流的两个容器。一个容器处理作为 Jenkins 代理启动 Pod 的 JNLP 合同。其他容器使用带有工具的镜像，以特定的编码语言构建代码：

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: jenkins-agent
  labels:
    role: jenkins-agent
```

```
data:
  template2: |-
      <org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
        <inheritFrom></inheritFrom>
        <name>template2</name>
        <instanceCap>2147483647</instanceCap>
        <idleMinutes>0</idleMinutes>
        <label>template2</label>
        <serviceAccount>jenkins</serviceAccount>
        <nodeSelector></nodeSelector>
        <volumes/>
        <containers>
          <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
            <name>jnlp</name>
            <image>image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-
rhel8:latest</image>
            <privileged>false</privileged>
            <alwaysPullImage>true</alwaysPullImage>
            <workingDir>/home/jenkins/agent</workingDir>
            <command></command>
            <args>\$(JENKINS_SECRET) \$(JENKINS_NAME)</args>
            <ttyEnabled>false</ttyEnabled>
            <resourceRequestCpu></resourceRequestCpu>
            <resourceRequestMemory></resourceRequestMemory>
            <resourceLimitCpu></resourceLimitCpu>
            <resourceLimitMemory></resourceLimitMemory>
            <envVars/>
          </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
          <org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
            <name>java</name>
            <image>image-registry.openshift-image-registry.svc:5000/openshift/java:latest</image>
            <privileged>false</privileged>
            <alwaysPullImage>true</alwaysPullImage>
            <workingDir>/home/jenkins/agent</workingDir>
            <command>cat</command>
            <args></args>
            <ttyEnabled>true</ttyEnabled>
            <resourceRequestCpu></resourceRequestCpu>
            <resourceRequestMemory></resourceRequestMemory>
            <resourceLimitCpu></resourceLimitCpu>
            <resourceLimitMemory></resourceLimitMemory>
            <envVars/>
          </org.csanchez.jenkins.plugins.kubernetes.ContainerTemplate>
        </containers>
        <envVars/>
        <annotations/>
        <imagePullSecrets/>
        <nodeProperties/>
      </org.csanchez.jenkins.plugins.kubernetes.PodTemplate>
```

注意

不要登录到 Jenkins 控制台并更改 pod 模板配置。如果您在创建 pod 模板后这样做，且 OpenShift Container Platform Sync 插件检测到与镜像流或镜像流标签关联的镜像已更改，则该插件会替换 pod 模板并覆盖这些配置更改。您无法将新配置与现有配置合并。

如果您的配置需要更复杂的配置需求，请考虑配置映射方法。

安装后，OpenShift Container Platform Sync 插件会监控 OpenShift Container Platform 的 API 服务器，以获取对镜像流、镜像流标签和配置映射的更新，并调整 Kubernetes 插件的配置。

适用以下规则：

- 从配置映射、镜像流或镜像流标签中删除标签或注解，从 Kubernetes 插件配置中删除任何现有 **PodTemplate**。
- 如果删除了这些对象，相应配置也会从 Kubernetes 插件中删除。
- 如果您创建适当标记或注解的 **ConfigMap**、**ImageStream** 或 **ImageStreamTag** 对象，或者在初始创建后添加标签，这会导致在 Kubernetes-plugin 配置中创建 **PodTemplate**。
- 对于按照配置映射表单的 **PodTemplate**，对 **PodTemplate** 的配置映射数据的更改将应用到 Kubernetes 插件配置中的 **PodTemplate** 设置。更改还会覆盖通过 Jenkins UI 在配置映射更改之间对 **PodTemplate** 所做的任何更改。

要将容器镜像用作 Jenkins 代理，镜像必须运行该代理作为入口点。如需了解更多详细信息，请参阅官方 Jenkins 文档。

其他资源

- OpenShift Jenkins 镜像的重要变化

5.1.7. Jenkins 权限

在 ConfigMap 中，如果 Pod 模板 XML 的 **<serviceAccount>** 元素是用于生成的 pod 的 OpenShift Container Platform 服务帐户，则服务帐户凭证将挂载到 pod 中。权限与服务帐户关联，并控制允许从 pod 对 OpenShift Container Platform master 执行的操作。

考虑以下场景，服务帐户用于 OpenShift Container Platform Jenkins 镜像中运行的 Kubernetes 插件启动的 pod：

如果您使用 OpenShift Container Platform 提供的 Jenkins 示例模板，则 **jenkins** 服务帐户将由运行 Jenkins 的项目的 **edit** 角色定义，且 master Jenkins Pod 已挂载了该服务帐户。

注入 Jenkins 配置的两个默认 Maven 和 NodeJS Pod 模板也将设置为使用与 Jenkins master 相同的服务帐户。

- 由于镜像流或 imagestreamtag 具有所需的标签或注解而被 OpenShift Container Platform Sync 插件自动发现的任何 Pod 模板均会被配置为使用 Jenkins master 的服务帐户作为其服务帐户。
- 对于其他方法，您可在 Jenkins 和 Kubernetes 插件中提供 Pod 模板定义，但必须明确指定要使用的服务帐户。其它方法包括 Jenkins 控制台、由 Kubernetes 插件提供的 **podTemplate** 管道 DSL，或标记其数据为 Pod 模板的 XML 配置的 ConfigMap。
- 如果没有为服务帐户指定值，则将使用 **default** 服务帐户。

- 确保所使用的任何服务帐户均具有 OpenShift Container Platform 中定义的必要权限、角色等，以操作您选择从 pod 中操作的任何项目。

5.1.8. 从模板创建 Jenkins 服务

模板提供参数字段来定义具有预定义默认值的所有环境变量。OpenShift Container Platform 提供模板，以简化 Jenkins 服务的新建操作。在初始集群设置期间，您的集群管理员应在默认 **openshift** 项目中注册 Jenkins 模板。

提供的两个模板均定义了部署配置和服务。模板在不同的存储策略中会有所不同，存储策略决定 Pod 重启后是否保留 Jenkins 内容。

注意

当 Pod 移到另一节点，或当对部署配置的更新触发了重新部署时，Pod 可能会重启。

- **jenkins-ephemeral** 使用临时存储。Pod 重启时，所有数据都会丢失。该模板仅适用于开发或测试。
- **jenkins-persistent** 使用持久性卷（PV）存储。数据不会因 Pod 重启而丢失。

要使用 PV 存储，集群管理员必须在 OpenShift Container Platform 部署中定义一个 PV 池。

选好所需模板后，您还必须对模板进行实例化，才能使用 Jenkins。

流程

- 使用以下任一方法新建 Jenkins 应用程序：
  - 一个 PV：

```
$ oc new-app jenkins-persistent
```

  - 或 **emptyDir** 类型卷，其配置在 Pod 重启后不保留：

```
$ oc new-app jenkins-ephemeral
```

使用这两个模板，您可以在它们上运行 **oc describe**，以查看可用于覆盖的所有参数。

例如：

```
$ oc describe jenkins-ephemeral
```

5.1.9. 使用 Jenkins Kubernetes 插件

在以下示例中，**openshift-jee-sample BuildConfig** 对象会导致 Jenkins Maven 代理 pod 动态置备。pod 会克隆一些 Java 源代码，构建一个 WAR 文件，并导致第二个 **BuildConfig openshift-jee-sample-docker** 运行。第二个 **BuildConfig** 会将新的 WAR 文件分层到一个容器镜像中。

重要

OpenShift Container Platform 4.11 从其有效负载中删除 OpenShift Jenkins Maven 和 NodeJS Agent 镜像。红帽不再生成这些镜像，它们不能从 **registry.redhat.io** 的 **ocp-tools-4** 存储库中提供。根据 OpenShift Container Platform 生命周期政策，红帽会维护这些镜像的 4.10 及更早的版本，适用于任何重要的程序错误修复或安全 CVE。

如需更多信息，请参阅以下"添加资源"部分中的"重要更改 OpenShift Jenkins 镜像"链接。

使用 Jenkins Kubernetes 插件的 BuildConfig 示例

```
kind: List
apiVersion: v1
items:
- kind: ImageStream
  apiVersion: image.openshift.io/v1
  metadata:
    name: openshift-jee-sample
- kind: BuildConfig
  apiVersion: build.openshift.io/v1
  metadata:
    name: openshift-jee-sample-docker
  spec:
    strategy:
      type: Docker
    source:
      type: Docker
      dockerfile: |-
        FROM openshift/wildfly-101-centos7:latest
        COPY ROOT.war /wildfly/standalone/deployments/ROOT.war
        CMD $STI_SCRIPTS_PATH/run
      binary:
        asFile: ROOT.war
    output:
      to:
        kind: ImageStreamTag
        name: openshift-jee-sample:latest
- kind: BuildConfig
  apiVersion: build.openshift.io/v1
  metadata:
    name: openshift-jee-sample
  spec:
    strategy:
      type: JenkinsPipeline
      jenkinsPipelineStrategy:
        jenkinsfile: |-
          node("maven") {
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
    triggers:
    - type: ConfigChange
```

它还可覆盖动态创建的 Jenkins 代理 pod 的规范。下面是对前一示例的修改，可覆盖容器内存并指定环境变量：

使用 Jenkins Kubernetes 插件的 BuildConfig 示例，指定内存限制和环境变量

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: openshift-jee-sample
spec:
  strategy:
    type: JenkinsPipeline
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        podTemplate(label: "mypod", 1
                cloud: "openshift", 2
                inheritFrom: "maven", 3
                containers: [
          containerTemplate(name: "jnlp", 4
                    image: "openshift/jenkins-agent-maven-35-centos7:v3.10", 5
                    resourceRequestMemory: "512Mi", 6
                    resourceLimitMemory: "512Mi", 7
                    envVars: [
            envVar(key: "CONTAINER_HEAP_PERCENT", value: "0.25") 8
          ])
        ]) {
          node("mypod") { 9
            sh "git clone https://github.com/openshift/openshift-jee-sample.git ."
            sh "mvn -B -Popenshift package"
            sh "oc start-build -F openshift-jee-sample-docker --from-file=target/ROOT.war"
          }
        }
  triggers:
  - type: ConfigChange
```

1. 动态定义的名为 **mypod** 的新 Pod 模板。新 pod 模板名称在节点片段中引用。
2. **cloud** 值必须设置为 **openshift**。
3. 新 pod 模板可以从现有 pod 模板继承其配置。在本例中，继承自 OpenShift Container Platform 预定义的 Maven Pod 模板。
4. 本例覆盖了预先存在容器中的值，且必须按名称指定。OpenShift Container Platform 附带的所有 Jenkins 代理镜像均使用容器名称 **jnlp**。
5. 再次指定容器镜像名称。这是个已知问题。
6. 指定了 **512 Mi** 的内存请求。
7. 指定了 **512 Mi** 的内存限值。
8. **CONTAINER_HEAP_PERCENT** 环境变量，其值指定为 **0.25**。
9. 节点片段引用定义的 pod 模板的名称。

构建完成后会默认删除 pod。可以使用插件或在 Jenkinsfile 管道中修改此行为。

上游 Jenkins 最近引入了一个 YAML 声明格式，用于定义带有您的管道的 **podTemplate** 管道 DSL。一个这种格式的示例，它使用 OpenShift Container Platform Jenkins 镜像中定义的 **java-builder** pod 模板示例：

```
def nodeLabel = 'java-buidler'

pipeline {
  agent {
    kubernetes {
      cloud 'openshift'
      label nodeLabel
      yaml """
apiVersion: v1
kind: Pod
metadata:
  labels:
    worker: ${nodeLabel}
spec:
  containers:
  - name: jnlp
    image: image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest
    args: ['\$(JENKINS_SECRET)', '\$(JENKINS_NAME)']
  - name: java
    image: image-registry.openshift-image-registry.svc:5000/openshift/java:latest
    command:
    - cat
    tty: true
"""
    }
  }

  options {
    timeout(time: 20, unit: 'MINUTES')
  }

  stages {
    stage('Build App') {
      steps {
        container("java") {
          sh "mvn --version"
        }
      }
    }
  }
}
```

其他资源

- OpenShift Jenkins 镜像的重要变化

5.1.10. Jenkins 内存要求

使用所提供的 Jenkins Ephemeral 或 Jenkins Persistent 模板部署时，默认内存限值为 **1 Gi**。

默认情况下，Jenkins 容器中运行的所有其他进程使用的内存总量不超过 **512 MiB**。如果这些进程需要更多内存，容器将停止。因此，我们强烈建议管道尽可能在代理容器中运行外部命令。

如果 **Project** 配额允许，请参阅 Jenkins 文档，了解 Jenkins master 应具有多少内存的建议。这些建议禁止为 Jenkins master 分配更多内存。

建议您为 Jenkins Kubernetes 插件创建的代理容器指定内存请求和限制值。管理员用户可通过 Jenkins 配置基于每个代理镜像设置默认值。内存请求和限值参数也可基于每个容器覆盖。

在实例化 Jenkins Ephemeral 或 Jenkins Persistent 模板时，您可通过覆盖 **MEMORY_LIMIT** 参数来增加 Jenkins 的可用内存量。

5.1.11. 其他资源

- 有关 Red Hat Universal Base Images (UBI)的更多信息，请参阅基础镜像选项 。
- OpenShift Jenkins 镜像的重要变化

5.2. JENKINS 代理

OpenShift Container Platform 提供了一个基础镜像，用作 Jenkins 代理。

Jenkins 代理的 Base 镜像执行以下操作：

- 拉取(pull)所需工具、无头 Java、Jenkins JNLP 客户端以及有用的工具，包括 **git**、**tar**、**zip** 和 **nss** 等。
- 建立 JNLP 代理作为入口点。
- 包括 **oc** 客户端工具，用于从 Jenkins 任务调用命令行操作。
- 为 Red Hat Enterprise Linux(RHEL)和 **localdev** 镜像提供 Dockerfile。

重要

使用适合您的 OpenShift Container Platform 发行版本的代理镜像版本。嵌入了与 OpenShift Container Platform 版本不兼容的 **oc** 客户端版本可能会导致意外行为。

OpenShift Container Platform Jenkins 镜像还定义了以下示例 **java-builder** pod 模板，以说明如何将代理镜像用于 Jenkins Kubernetes 插件。

java-builder pod 模板采用两个容器：

- 使用 OpenShift Container Platform Base 代理镜像的 **jnlp** 容器，并处理 JNLP 合同来启动和停止 Jenkins 代理。
- 一个是 **java** 容器，它使用 **java** OpenShift Container Platform Sample ImageStream，其包含各种 Java 二进制文件，包括 Maven 二进制文件 **mvn**，用于构建代码。

5.2.1. Jenkins 代理镜像

OpenShift Container Platform Jenkins 代理镜像在 Quay.io 或 registry.redhat.io 上提供。

Jenkins 镜像通过 Red Hat Registry 提供：

```
$ docker pull registry.redhat.io/ocp-tools-4/jenkins-rhel8:<image_tag>
```

```
$ docker pull registry.redhat.io/ocp-tools-4/jenkins-agent-base-rhel8:<image_tag>
```

要使用这些镜像，您可直接从 Quay.io 或 registry.redhat.io 访问或将其推送（push）到 OpenShift Container Platform 容器镜像 registry 中。

5.2.2. Jenkins 代理环境变量

每个 Jenkins 代理容器均可通过以下环境变量进行配置。

变量	定义	值和设置示例
JAVA_MAX_HEAP_PARAM、CONTAINER_HEAP_PERCENT、JENKINS_MAX_HEAP_UPPER_BOUND_MB	这些值控制 Jenkins JVM 的最大堆大小。如果设置了 **JAVA_MAX_heap_PARAM**，则优先使用其值。否则，最大堆大小将动态计算为容器内存限值的 **CONTAINER_HEAP_PERCENT**，可选上限为 **JENKINS_MAX_HEAP_UPPER_BOUND_MB** MiB。 默认情况下，Jenkins JVM 的最大堆大小设置为容器内存限值的 50%，且无上限。	**Java_MAX_heap_PARAM** 示例设置：**-Xmx512m** **container_heap_percent** 默认：**0.5** 或 50% **Jenkins_MAX_heap_UPPER_BOUND_MB** 示例设置：**512 MiB**
JAVA_INITIAL_HEAP_PARAM、CONTAINER_INITIAL_PERCENT	这些值控制 Jenkins JVM 的初始堆大小。如果设置了 **JAVA_INITIAL_heap_PARAM**，则优先使用其值。否则，初始堆大小将动态计算为动态计算的最大堆大小的 **CONTAINER_INITIAL_PERCENT**。 默认情况下，JVM 设置初始堆大小。	**java_INITIAL_heap_PARAM** 示例设置：**-Xms32m** **container_INITIAL_percent** 示例设置：**0.1**或 10%
CONTAINER_CORE_LIMIT	如果设置，请将用于调整内部 JVM 线程数的内核数指定为整数。	示例设置：**2**
JAVA_TOOL_OPTIONS	指定应用于该容器中运行的所有 JVM 的选项。不建议覆盖该值。	默认：**-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true**

变量	定义	值和设置示例
JAVA_GC_OPTS	指定 Jenkins JVM 垃圾回收参数。不建议覆盖该值。	默认：**-XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90**
JENKINS_JAVA_OVERRIDES	指定适用于 Jenkins JVM 的附加选项。这些选项附加至所有其他选项中，包括上面的 Java 选项，必要时可用于覆盖其中任何一个选项。用空格分开各个附加选项；如有任意选项包含空格字符，请使用反斜杠转义。	示例设置：**-Dfoo -Dbar**; **-Dfoo=first\ value -Dbar=second\ value**
USE_JAVA_VERSION	指定用来在容器中运行代理的 Java 版本版本。容器基础镜像安装了两个 java 版本： **java-11** 和 **java-1.8.0**。如果扩展容器基础镜像，您可以使用其关联的后缀指定 java 的任何替代版本。	默认值为 **java-11**。 示例设置： **java-1.8.0**

5.2.3. Jenkins 代理内存要求

所有 Jenkins 代理均使用 JVM 来托管 Jenkins JNLP 代理和运行任何 Java 应用程序，如 **javac**、Maven 或 Gradle。

默认情况下，Jenkins JNLP 代理 JVM 会将容器内存限值的 50% 用于其堆。该值可通过 **CONTAINER_HEAP_PERCENT** 环境变量修改，还可设置上限或整个覆盖。

默认情况下，Jenkins 代理容器中运行的其它进程（如 shell 脚本或从管道运行的 **oc** 命令）在不引发 OOM 终止的情况下，所用内存均不得超过剩余的 50% 内存限值。

默认情况下，Jenkins 代理容器中运行的每个其他 JVM 进程最多可将 25% 的容器内存限值用于其堆。对于很多构建工作负载，可能还需调整此限值。

5.2.4. Jenkins 代理 Gradle 构建

在 OpenShift Container Platform 上的 Jenkins 代理中托管 Gradle 构建会出现其他复杂情况，因为除了 Jenkins JNLP 代理和 Gradle JVM 外，Gradle 还会生成第三个 JVM 来运行测试（若已指定）。

建议将以下设置作为起始点，在 OpenShift Container Platform 上内存受限的 Jenkins 代理中运行 Gradle 构建。您还可按需修改这些设置。

- 通过将 **org.gradle.daemon=false** 添加到 **gradle.properties** 文件中来确保禁用长期 Gradle 守护进程。
- 通过确保 **gradle.properties** 文件中未设置 **org.gradle.parallel=true** 且 **--parallel** 未设置为命令行参数来禁用并行构建执行。

- 要防止 Java 编译超出进程范围，请在 **build.gradle** 文件中设置 **java { options.fork = false }**。
- 通过确保在 **build.gradle** 文件中设置 **test { maxParallelForks = 1 }** 来禁用多个附加测试进程。
- 使用 **GRADLE_OPTS**、**JAVA_OPTS** 或 **JAVA_TOOL_OPTIONS** 环境变量覆盖 Gradle JVM 内存参数。
- 通过在 **build.gradle** 中定义 **maxHeapSize** 和 **jvmArgs** 设置，或通过 **-Dorg.gradle.jvmargs** 命令行参数来为任何 Gradle 测试 JVM 设置最大堆大小和 JVM 参数。

5.2.5. Jenkins 代理 pod 保留

构建完成后或停止后会默认删除 Jenkins 代理 pod。此行为可通过 Kubernetes 插件 pod 保留设置来更改。Pod 保留可针对所有 Jenkins 构建设置，并覆盖每个 pod 模板。支持以下行为：

- **Always** 保留构建 pod，不受构建结果的限制。
- **Default** 使用插件值，即仅限 pod 模板。
- **Never** 始终删除 pod。
- **On Failure** 如果构建过程中失败，则保留 pod。

您可覆盖管道 Jenkinsfile 中的 pod 保留：

```
podTemplate(label: "mypod",
  cloud: "openshift",
  inheritFrom: "maven",
  podRetention: onFailure(), 1
  containers: [
    ...
  ]) {
  node("mypod") {
    ...
  }
}
```

1 **podRetention** 允许的值为 **never()**、**onFailure()**、**always()** 和 **default()**。

警告

保留的 Pod 可能会根据资源配额继续运行和计数。

5.3. 从 JENKINS 迁移到 OPENSHIFT PIPELINES 或 TEKTON

您可以将 CI/CD 工作流从 Jenkins 迁移到 Red Hat OpenShift Pipelines，这是基于 Tekton 项目的云原生 CI/CD 体验。

5.3.1. Jenkins 和 OpenShift Pipelines 概念的比较

您可以查看并比较 Jenkins 和 OpenShift Pipelines 中使用的以下等效术语。

5.3.1.1. Jenkins 术语

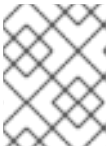
Jenkins 提供声明式和脚本化管道，它们可以使用共享库和插件扩展。Jenkins 中的一些基本术语如下：

- **管道（Pipeline）**：利用 Groovy 语法自动化构建、测试和部署应用的整个流程。
- **节点（Node）**：能够编配或执行脚本化管道的计算机。
- **阶段（Stage）**：在管道中执行的概念上不同的任务子集。插件或用户界面通常使用此块来显示任务的状态或进度。
- **步骤（Step）**：一项任务指定要执行的确切操作，可使用命令或脚本。

5.3.1.2. OpenShift Pipelines 术语

OpenShift Pipelines 使用 YAML 语法用于声明管道，由任务组成。OpenShift Pipelines 中的一些基本术语如下：

- **频道（Pipeline）**：一组串行或并行（或两者）任务。
- **任务（Task）**：作为命令、二进制文件或脚本的步骤序列。
- **管道运行（PipelineRun）**：执行包含一个或多个任务的管道。
- **任务运行（TaskRun）**：通过一个或多个步骤执行任务。



注意

您可以使用一组输入（如参数和工作区）启动 PipelineRun 或 TaskRun，执行会产生一组输出和工件。

- **workspace**：在 OpenShift Pipelines 中，工作区是概念块，用于以下目的：
  - 存储输入、输出和构建工件.
  - 在任务间共享数据的通用空间.
  - 在 secret 中保存的凭证挂载点、配置映射中保存的配置以及机构共享的通用工具。

注意

在 Jenkins 中，没有 OpenShift Pipelines 工作区直接对应的工作区。您可以将控制节点视为工作区，因为它存储克隆的代码存储库、构建历史记录和工件。当作业分配给其他节点时，克隆的代码和生成的工件会存储在该节点中，但控制节点维护构建历史记录。

5.3.1.3. 概念映射

Jenkins 和 OpenShift Pipelines 的构建块并不等同，特定比较并不提供技术准确的映射。Jenkins 和 OpenShift Pipelines 中的以下术语和概念一般关联：

表 5.1. Jenkins 和 OpenShift Pipelines - 基本比较

Jenkins	OpenShift Pipelines
Pipeline	Pipeline 和 PipelineRun
Stage	任务
Step	一个任务中的一个步骤

5.3.2. 将示例管道从 Jenkins 迁移到 OpenShift Pipelines

您可以使用以下等同的示例来帮助将构建、测试和部署管道从 Jenkins 迁移到 OpenShift Pipelines。

5.3.2.1. Jenkins 管道

考虑在 Groovy 中编写的 Jenkins 管道，用于构建、测试和部署：

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

5.3.2.2. OpenShift Pipelines 管道

要在 OpenShift Pipelines 中创建相当于前面的 Jenkins 管道的管道，您需要创建以下三个任务：

build 任务 YAML 定义文件示例

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: myproject-build
spec:
  workspaces:
  - name: source
  steps:
```

```
  - image: my-ci-image
    command: ["make"]
    workingDir: $(workspaces.source.path)
```

test 任务 YAML 定义文件示例

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: myproject-test
spec:
  workspaces:
  - name: source
  steps:
  - image: my-ci-image
    command: ["make check"]
    workingDir: $(workspaces.source.path)
  - image: junit-report-image
    script: |
      #!/usr/bin/env bash
      junit-report reports/**/*.xml
    workingDir: $(workspaces.source.path)
```

deploy 任务 YAML 定义文件示例

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: myprojectd-deploy
spec:
  workspaces:
  - name: source
  steps:
  - image: my-deploy-image
    command: ["make deploy"]
    workingDir: $(workspaces.source.path)
```

您可以按顺序组合三个任务以在 OpenShift Pipelines 中组成管道：

示例：用于构建、测试和部署的 OpenShift Pipelines 管道

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: myproject-pipeline
spec:
  workspaces:
  - name: shared-dir
  tasks:
  - name: build
    taskRef:
      name: myproject-build
    workspaces:
    - name: source
```

```
      workspace: shared-dir
  - name: test
    taskRef:
      name: myproject-test
    workspaces:
    - name: source
      workspace: shared-dir
  - name: deploy
    taskRef:
      name: myproject-deploy
    workspaces:
    - name: source
      workspace: shared-dir
```

5.3.3. 从 Jenkins 插件迁移到 Tekton Hub 任务

您可以使用 插件 来扩展 Jenkins 的功能。要在 OpenShift Pipelines 中实现类似的可扩展性，请使用 Tekton Hub 中提供的任何任务。

例如，考虑 Tekton Hub 中的 git-clone 任务，它对应于 Jenkins 的 git 插件。

示例：Tekton Hub 中的 git-clone 任务

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
 name: demo-pipeline
spec:
 params:
   - name: repo_url
   - name: revision
 workspaces:
   - name: source
 tasks:
   - name: fetch-from-git
     taskRef:
       name: git-clone
     params:
       - name: url
         value: $(params.repo_url)
       - name: revision
         value: $(params.revision)
     workspaces:
     - name: output
       workspace: source
```

5.3.4. 使用自定义任务和脚本扩展 OpenShift Pipelines 功能

在 OpenShift Pipelines 中，如果您在 Tekton Hub 中找不到正确的任务，或需要对任务进行更大的控制，您可以创建自定义任务和脚本来扩展 OpenShift Pipelines 的功能。

示例：运行 maven test 命令的自定义任务

```
apiVersion: tekton.dev/v1beta1
```

```
kind: Task
metadata:
  name: maven-test
spec:
  workspaces:
  - name: source
  steps:
  - image: my-maven-image
    command: ["mvn test"]
    workingDir: $(workspaces.source.path)
```

示例：通过提供自定义 shell 脚本的路径

```
...
steps:
  image: ubuntu
  script: |
      #!/usr/bin/env bash
      /workspace/my-script.sh
...
```

示例：在 YAML 文件中写入自定义 Python 脚本

```
...
steps:
  image: python
  script: |
      #!/usr/bin/env python3
      print(“hello from python!”)
...
```

5.3.5. Jenkins 和 OpenShift Pipelines 执行模型的比较

Jenkins 和 OpenShift Pipelines 提供了类似的功能，但在架构和执行方面有所不同。

表 5.2. Jenkins 和 OpenShift Pipelines 中的执行模型比较

Jenkins	OpenShift Pipelines
Jenkins 有一个控制器节点。Jenkins 集中运行管道和步骤，或编排其他节点上运行的作业。	OpenShift Pipelines 是无服务器且分布式的，它没有执行中央依赖项。
容器由 Jenkins 控制器节点通过管道启动。	OpenShift Pipelines 采用"容器先行"方法，其中的每个步骤都作为 pod 中的容器运行（等同于 Jenkins 中的节点）。
使用插件可实现可扩展性。	使用 Tekton Hub 中的任务或创建自定义任务和脚本来实现可扩展性。

5.3.6. 常见使用案例示例

Jenkins 和 OpenShift Pipelines 都为常见 CI/CD 用例提供功能，例如：

- 使用 Apache Maven 编译、构建和部署镜像
- 使用插件扩展核心功能
- 重新使用可共享库和自定义脚本

5.3.6.1. 在 Jenkins 和 OpenShift Pipelines 中运行 Maven 管道

您可以在 Jenkins 和 OpenShift Pipelines 工作流中使用 Maven 来编译、构建和部署镜像。要将现有的 Jenkins 工作流映射到 OpenShift Pipelines，请考虑以下示例：

示例：完成并构建镜像，并使用 Jenkins 中的 Maven 部署到 OpenShift

```
#!/usr/bin/groovy
node('maven') {
   stage 'Checkout'
   checkout scm

   stage 'Build'
   sh 'cd helloworld && mvn clean'
   sh 'cd helloworld && mvn compile'

   stage 'Run Unit Tests'
   sh 'cd helloworld && mvn test'

   stage 'Package'
   sh 'cd helloworld && mvn package'

   stage 'Archive artifact'
   sh 'mkdir -p artifacts/deployments && cp helloworld/target/*.war artifacts/deployments'
   archive 'helloworld/target/*.war'

   stage 'Create Image'
   sh 'oc login https://kubernetes.default -u admin -p admin --insecure-skip-tls-verify=true'
   sh 'oc new-project helloworldproject'
   sh 'oc project helloworldproject'
   sh 'oc process -f helloworld/jboss-eap70-binary-build.json | oc create -f -'
   sh 'oc start-build eap-helloworld-app --from-dir=artifacts/'

   stage 'Deploy'
   sh 'oc new-app helloworld/jboss-eap70-deploy.json' }
```

示例：完成并构建镜像，并使用 OpenShift Pipelines 中的 Maven 部署到 OpenShift 中。

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: maven-pipeline
spec:
  workspaces:
    - name: shared-workspace
    - name: maven-settings
    - name: kubeconfig-dir
      optional: true
  params:
```

```
    - name: repo-url
    - name: revision
    - name: context-path
  tasks:
    - name: fetch-repo
      taskRef:
        name: git-clone
      workspaces:
        - name: output
          workspace: shared-workspace
      params:
        - name: url
          value: "$(params.repo-url)"
        - name: subdirectory
          value: ""
        - name: deleteExisting
          value: "true"
        - name: revision
          value: $(params.revision)
    - name: mvn-build
      taskRef:
        name: maven
      runAfter:
        - fetch-repo
      workspaces:
        - name: source
          workspace: shared-workspace
        - name: maven-settings
          workspace: maven-settings
      params:
        - name: CONTEXT_DIR
          value: "$(params.context-path)"
        - name: GOALS
          value: ["-DskipTests", "clean", "compile"]
    - name: mvn-tests
      taskRef:
        name: maven
      runAfter:
        - mvn-build
      workspaces:
        - name: source
          workspace: shared-workspace
        - name: maven-settings
          workspace: maven-settings
      params:
        - name: CONTEXT_DIR
          value: "$(params.context-path)"
        - name: GOALS
          value: ["test"]
    - name: mvn-package
      taskRef:
        name: maven
      runAfter:
        - mvn-tests
      workspaces:
        - name: source
```

```
      workspace: shared-workspace
    - name: maven-settings
      workspace: maven-settings
  params:
    - name: CONTEXT_DIR
      value: "$(params.context-path)"
    - name: GOALS
      value: ["package"]
- name: create-image-and-deploy
  taskRef:
    name: openshift-client
  runAfter:
    - mvn-package
  workspaces:
    - name: manifest-dir
      workspace: shared-workspace
    - name: kubeconfig-dir
      workspace: kubeconfig-dir
  params:
    - name: SCRIPT
      value: |
        cd "$(params.context-path)"
        mkdir -p ./artifacts/deployments && cp ./target/*.war ./artifacts/deployments
        oc new-project helloworldproject
        oc project helloworldproject
        oc process -f jboss-eap70-binary-build.json | oc create -f -
        oc start-build eap-helloworld-app --from-dir=artifacts/
        oc new-app jboss-eap70-deploy.json
```

5.3.6.2. 使用插件扩展 Jenkins 和 OpenShift Pipelines 的核心功能

Jenkins 利用了其广泛的用户群来多年开发的大量插件生态系统。您可以在 Jenkins 插件索引中搜索和浏览插件。

OpenShift Pipelines 还有许多任务由社区和企业用户开发并贡献。Tekton Hub 中提供了可重复使用的 OpenShift Pipelines 任务的公开目录。

另外，OpenShift Pipelines 包含在其核心功能中 Jenkins 生态系统的许多插件。例如，授权是 Jenkins 和 OpenShift Pipelines 中的关键功能。虽然 Jenkins 使用基于角色的访问控制插件来确保授权，OpenShift Pipelines 使用 OpenShift 的内置基于角色的访问控制系统。

5.3.6.3. 在 Jenkins 和 OpenShift Pipelines 中共享可重复使用的代码

Jenkins 共享库为 Jenkins 管道的各部分提供可重复使用的代码。该库在 Jenkinsfile 之间共享，以创建高度模块化的管道，而不重复代码。

虽然 OpenShift Pipelines 中没有直接等效的 Jenkins 共享库，但您可以使用 Tekton Hub 中的任务与自定义任务和脚本相结合来实现类似的工作流。

5.3.7. 其他资源

- 了解 OpenShift Pipelines
- 基于角色的访问控制

5.4. OPENSHIFT JENKINS 镜像的重要变化

OpenShift Container Platform 4.11 将 OpenShift Jenkins 和 OpenShift Agent Base 镜像移到 **registry.redhat.io** 的 **ocp-tools-4** 仓库中。它还从其有效负载中删除 OpenShift Jenkins Maven 和 NodeJS Agent 镜像：

- OpenShift Container Platform 4.11 将 OpenShift Jenkins 和 OpenShift Agent Base 镜像移到 **registry.redhat.io** 的 **ocp-tools-4** 仓库中，以便红帽可以在 OpenShift Container Platform 生命周期外生成和更新镜像。在以前的版本中，这些镜像位于 OpenShift Container Platform 安装有效负载以及 **registry.redhat.io** 的 **openshift4** 存储库中。

- OpenShift Container Platform 4.10 弃用了 OpenShift Jenkins Maven 和 NodeJS Agent 镜像。OpenShift Container Platform 4.11 从其有效负载中删除这些镜像。红帽不再生成这些镜像，它们不能从 **registry.redhat.io** 的 **ocp-tools-4** 存储库中提供。根据 OpenShift Container Platform 生命周期政策，红帽会维护这些镜像的 4.10 及更早的版本，适用于任何重要的程序错误修复或安全 CVE。

这些更改支持 OpenShift Container Platform 4.10 建议在 Jenkins Kubernetes 插件中使用多个容器 Pod 模板。

5.4.1. OpenShift Jenkins 镜像重新定位

OpenShift Container Platform 4.11 对特定 OpenShift Jenkins 镜像的位置和可用性进行了大量更改。另外，您还可以配置何时以及如何更新这些镜像。

什么与 OpenShift Jenkins 镜像保持相同？

- Cluster Samples Operator 管理用于运行 OpenShift Jenkins 镜像的 **ImageStream** 和 **Template** 对象。

- 默认情况下，Jenkins pod 模板的 Jenkins **DeploymentConfig** 对象会在 Jenkins 镜像更改时触发重新部署。默认情况下，此镜像由 Samples Operator 有效负载中的 **ImageStream** YAML 文件中的 **openshift** 命名空间的 Jenkins 镜像流的 **jenkins:2** 镜像流标签引用。

- 如果您从 OpenShift Container Platform 4.10 及更早版本升级到 4.11，弃用的 **maven** 和 **nodejs** pod 模板仍处于默认镜像配置中。

- 如果您从 OpenShift Container Platform 4.10 及更新版本升级到 4.11，则 **jenkins-agent-maven** 和 **jenkins-agent-nodejs** 镜像流仍存在于集群中。要维护这些镜像流，请参见以下部分："**openshift** 命名空间中的 **jenkins-agent-maven** 和 **jenkins-agent-nodejs** 镜像流有什么变化？"

OpenShift Jenkins 镜像支持列表中的哪些变化？

registry.redhat.io registry 中的 **ocp-tools-4** 仓库中的每个新镜像都支持多个 OpenShift Container Platform 版本。当红帽更新其中一个新镜像时，所有版本都同时可用。当红帽更新镜像以响应安全公告时，此可用性是理想的选择。最初，这个更改适用于 OpenShift Container Platform 4.11 及更新的版本。计划此更改最终会应用到 OpenShift Container Platform 4.9 及更新的版本。

在以前的版本中，每个 Jenkins 镜像只支持一个 OpenShift Container Platform 版本，红帽可能会随时间顺序更新这些镜像。

OpenShift Jenkins 和 Jenkins Agent Base ImageStream 和 ImageStreamTag 对象增加了什么？

通过从 in-payload 镜像流移到引用非备份镜像的镜像流，OpenShift Container Platform 可以定义额外的镜像流标签。红帽创建了一系列新镜像流标签，并附带现有的 **"value": "jenkins:2"** 和 **"value": "image-registry.openshift-image-registry.svc:5000/openshift/jenkins-agent-base-rhel8:latest"** 镜像流标签

（位于 OpenShift Container Platform 4.10 及更早版本）。这些新镜像流标签处理了一些请求，以改进维护 Jenkins 相关镜像流的方式。

关于新镜像流标签：

ocp-upgrade-redeploy

要在升级 OpenShift Container Platform 时更新 Jenkins 镜像，请在 Jenkins 部署配置中使用此镜像流标签。此镜像流标签与 jenkins 镜像流现有的 **jenkins** 镜像流的 **2** 镜像流标签以及 **jenkins-agent-base-rhel8** 镜像流的 **latest** 镜像流标签对应。它仅使用特定于一个 SHA 或镜像摘要的镜像标签。当 **ocp-tools-4** 镜像更改时，如 Jenkins 安全公告，红帽工程团队会更新 Cluster Samples Operator 有效负载。

user-maintained-upgrade-redeploy

要在升级 OpenShift Container Platform 后手动重新部署 Jenkins，请在 Jenkins 部署配置中使用此镜像流标签。此镜像流标签使用至少可用的特定镜像版本指示符。重新部署 Jenkins 时，运行以下命令：**$ oc import-image jenkins:user-maintained-upgrade-redeploy -n openshift**。当您发出此命令时，OpenShift Container Platform **ImageStream** 控制器可以访问 **registry.redhat.io** 镜像 registry，并将任何更新的镜像存储在 OpenShift 镜像 registry 的 Jenkins **ImageStreamTag** 对象的插槽中。否则，如果您没有运行此命令，您的 Jenkins 部署配置不会触发重新部署。

scheduled-upgrade-redeploy

要在释放后自动重新部署 Jenkins 镜像的最新版本，请在 Jenkins 部署配置中使用此镜像流标签。此镜像流标签使用 OpenShift Container Platform 镜像流控制器的定期导入镜像流标签功能，用于检查后备镜像中的更改。例如，如果镜像更改（例如由于最新的 Jenkins 安全公告），OpenShift Container Platform 会触发重新部署 Jenkins 部署配置。在以下"添加资源"中，请参阅"定期导入镜像流标签"。

openshift 命名空间中的 jenkins-agent-maven 和 jenkins-agent-nodejs 镜像流会发生什么？

OpenShift Container Platform 的 OpenShift Jenkins Maven 和 NodeJS Agent 镜像已在 4.10 中弃用，并在 4.11 中从 OpenShift Container Platform 安装有效负载中删除。它们没有在 **ocp-tools-4** 仓库中定义 alternatives。但是，您可以使用以下"Additional resources"部分中提到的"Jenkins 代理"主题描述的 sidecar 模式来解决这个问题。

但是，Cluster Samples Operator 不会删除之前版本创建的 **jenkins-agent-maven** 和 **jenkins-agent-nodejs** 镜像流，指向 **registry.redhat.io** 上对应 OpenShift Container Platform 有效负载镜像的标签。因此，您可以通过运行以下命令拉取 (pull) 镜像的更新：

```
$ oc import-image jenkins-agent-nodejs -n openshift
```

```
$ oc import-image jenkins-agent-maven -n openshift
```

5.4.2. 自定义 Jenkins 镜像流标签

要覆盖默认的升级行为，并控制 Jenkins 镜像的升级方式，您可以设置 Jenkins 部署配置使用的镜像流标签值。

默认升级行为是在 Jenkins 镜像是安装有效负载的一部分时存在的行为。**jenkins-rhel.json** 镜像流文件中的镜像流标签名称 **2** 和 **ocp-upgrade-redeploy** 使用 SHA 特定镜像引用。因此，当这些标签使用新的 SHA 更新时，OpenShift Container Platform 镜像更改控制器会自动从关联的模板重新部署 Jenkins 部署配置，如 **jenkins-ephemeral.json** 或 **jenkins-persistent.json**。

对于新部署，要覆盖该默认值，您可以在 **jenkins-ephemeral.json** Jenkins 模板中更改 **JENKINS_IMAGE_STREAM_TAG** 的值。例如，将 **"value": "jenkins:2"** 中的 **2** 替换为以下镜像流标签之一：

- 升级 OpenShift Container Platform 时，OCP **-upgrade-redeploy** （默认值）会更新 Jenkins 镜像。
- **user-maintained-upgrade-redeploy** 要求您在升级 OpenShift Container Platform 后运行 **$ oc import-image jenkins:user-maintained-upgrade-redeploy -n openshift** 来手动重新部署 Jenkins。
- **scheduled-upgrade-redeploy** 会定期检查给定的 **<image>:<tag>** 组合，以便在镜像更改时进行更改和升级。镜像更改控制器拉取更改的镜像，并重新部署由模板置备的 Jenkins 部署配置。有关此调度导入策略的更多信息，请参阅以下"Additional resources"中的"将标签添加到镜像流"。

注意

要覆盖现有部署的当前升级值，请更改与这些模板参数对应的环境变量值。

先决条件

- 您已在 OpenShift Container Platform 4.15 上运行了 OpenShift Jenkins。
- 您知道部署 OpenShift Jenkins 的命名空间。

流程

- 设置镜像流标签值，将 **<namespace>** 替换为部署 OpenShift Jenkins 的命名空间，并将 **<image_stream_tag>** 替换为镜像流标签：

  Example

  ```
  $ oc patch dc jenkins -p '{"spec":{"triggers":[{"type":"ImageChange","imageChangeParams":
  {"automatic":true,"containerNames":["jenkins"],"from":
  {"kind":"ImageStreamTag","namespace":"<namespace>","name":"jenkins:
  <image_stream_tag>"}}}]}}'
  ```

  提示

  另外，要编辑 Jenkins 部署配置 YAML，请输入 **$ oc edit dc/jenkins -n <namespace>** 并更新 **value: 'jenkins:<image_stream_tag>'** 行。

5.4.3. 其他资源

- 向镜像流中添加标签
- 配置定期导入镜像流标签
- Jenkins 代理
- 认证的 **Jenkins** 镜像
- 认证的 **jenkins-agent-base** 镜像
- 认证的 **jenkins-agent-maven** 镜像
- 认证的 **jenkins-agent-nodejs** 镜像