OpenShift Container Platform 4.7

CI/CD

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

Last Updated: 2023-03-24

OpenShift Container Platform 4.7 CI/CD

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

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

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 Builds
- OpenShift Pipelines
- OpenShift GitOps

1.1. OPENSHIFT BUILDS

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，请参阅了解 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 BUILDS

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。有关每种源类型的更多信息，请参阅"创建构建输入"。
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 **ImageContentSourcePolicy** 对象来配置存储库镜像，则只能将全局 pull secret 用于镜像 registry。您不能在项目中添加 pull secret。

另外，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。

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。此外，也可通过 **ref** 字段来指定要使用的特定代码。有效的 **ref** 可以是 SHA1 标签或分支名称。

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**。

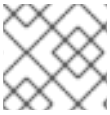
如果未提供 **ref** 值，OpenShift Container Platform 将执行浅克隆（**--depth=1**）。这时，仅下载与默认分支（通常为 **master**）上最近提交相关联的文件。这将使存储库下载速度加快，但不会有完整的提交历史记录。要对指定存储库的默认分支执行完整 **git clone**，请将 **ref** 设为默认分支（如 **master**）的名称。

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
- 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: "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
```

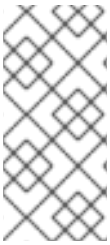
2.3.4.2.3. 从 .gitconfig 文件创建 secret

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

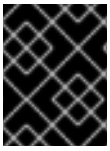
```
[user]
        name = <name>
        email = <email>
[http]
        sslVerify = false
        sslCert = /var/run/secrets/openshift.io/source/client.crt
        sslKey = /var/run/secrets/openshift.io/source/client.key
        sslCaInfo = /var/run/secrets/openshift.io/source/cacert.crt
```

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。

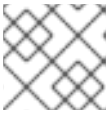
2. 在使用 SSH 密钥访问私有存储库之前，先创建 secret：

```
$ 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。使用以下命令：

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

a. 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

- 使用 **.gitconfig** 文件创建基于 SSH 的身份验证 secret：

```
$ 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

- 创建组合了 **.gitconfig** 文件和 CA 证书的 secret：

```
$ 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

- 使用 CA 证书创建基本身份验证 secret：

```
$ 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 .gitconfig 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

- 使用 **.gitconfig** 文件创建基本身份验证 secret：

```
$ 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

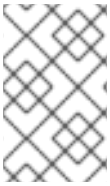
- 使用 **.gitconfig** 文件和 CA 证书创建基本身份验证 secret：

```
$ 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

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: dmFsdWUtMQ0K (3)
  password: dmFsdWUtMg0KDQo=
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。

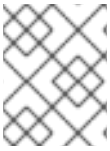
2.3.6.1.2. secret 的类型

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

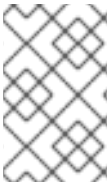
您可以指定其他任意类型，如 **example.com/my-secret-type**。这些类型不在服务器端强制执行，但代表 secret 的创建者意在符合该类型的键/值要求。

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

- 使用创建命令从 JSON 或 YAML 文件创建 secret 对象：

```
$ 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: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

(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. 创建要引用您的 secret 的 Pod：

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

2. 获取日志：

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

3. 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

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: dmFsdWUtMQ0K      1
  password: dmFsdWUtMQ0KDQo=  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 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 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
```

一个 Build Config 的 YAML 定义，在环境变量中使用 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

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。

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，则进行创建：

```
$ 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。

2. 如果 **Secret** 对象不存在，则进行创建：

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

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。

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
```

要在新 **BuildConfig** 对象中包含 secret 和配置映射，请运行以下命令：

```
$ 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，**settings.xml** 和 **id_rsa** 文件将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"
```

您还可以指定创建新 **BuildConfig** 对象时的目标目录：

```
$ 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"]
```

注意

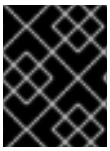
用户应该从最终的应用程序镜像中移除输入 secret，以便从该镜像运行的容器中不会存在这些 secret。但是，secret 仍然存在于它们添加到的层中的镜像本身内。这一移除是 Dockerfile 本身的一部分。

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** 中设置
- 使用 **oc start-build --env** 明确提供（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。

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:
  https://index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

1. registry URL。
2. 加密的密码。
3. 用于登录的电子邮件地址。

您可以在此文件中定义多个容器镜像 registry 条目。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. 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。

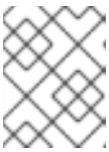
```
$ oc secrets link builder dockerhub
```

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。将 pull secret 链接到构建使用的服务帐户，而不指定 **pullSecret** 字段：

```
$ 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 Container Platform 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) 构建设置输出镜像的以下标签：

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

标签	描述
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。

应用到所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 中的 **FROM** 指令替换为 **BuildConfig** 对象的 **from**。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

将 Dockerfile 中的 **FROM** 指令替换为 **BuildConfig** 中的 **from**。

```
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

要通过构建的 **dockerfilePath** 字段使用不同的路径来定位 Dockerfile，请设置：

```
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。

流程

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 build 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: "foo"
      value: "bar"
```

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. 对 Docker 的构建层进行压缩

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
```

2.5.2. 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。

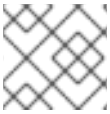
流程

要覆盖构建器镜像提供的 **assemble**、**run** 和 **save-artifacts** S2I 脚本，请执行以下任一操作：

- 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的 URL，作为策略定义的一部分。例如：

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

1 此路径会将 **run**、**assemble** 和 **save-artifacts** 附加到其中。如果找到任何或所有脚本，将使用它们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和生成的镜像：环境文件和 BuildConfig 环境值。提供的变量将存在于构建过程和输出镜像中。

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

2.5.3. 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。

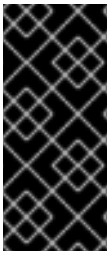
对于项目中的 OpenShift Container Platform 管道，必须使用 Jenkins Pipeline 构建策略。此策略默认使用源存储库根目录下的 **jenkinsfile**，但也提供以下配置选项：

- 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'
```

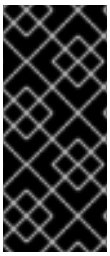
```
      openshiftBuild(buildConfig: 'ruby-sample-build', showBuildLogs: 'true')
      stage 'deploy'
      openshiftDeploy(deploymentConfig: 'frontend')
    }
```

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。

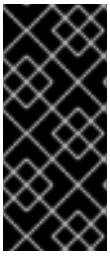
为 Jenkins 任务启动构建的方式决定了如何设置参数。

- 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 **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 4.7 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/ubi8/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. 创建 **BuildConfig**:

```
$ 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. 创建镜像流：

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

5. 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. 执行基本构建

以下小节提供了基本构建操作的说明，包括启动和取消构建、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

- 使用以下命令从源启动构建，以将本地 Git 存储库的内容作为标签 **v2** 的存档发送：

```
$ 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. 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>
```

2.7.4. 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>
```

2.7.5. 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>
```

2.7.5.1. 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>
```

2.7.5.2. 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。

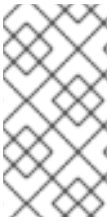
流程

- 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>
```

2.7.5.3. 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（也称为 master 主机）确认事件内的分支引用是否与相应 **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。

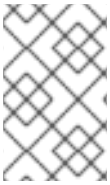
```
- kind: Secret
  apiVersion: v1
  metadata:
    name: mysecret
    creationTimestamp:
  data:
    WebHookSecretKey: c2VjcmV0dmFsdWUx
```

2.8.1.1.1. 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。前者使 Webhook URL 唯一且难以预测，后者是一个可选的字符串字段，用于创建正文的 HMAC 十六进制摘要，作为 **X-Hub-Signature** 标头来发送。

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**。

流程

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

   ```
   $ oc describe bc/<name-of-your-BuildConfig>
   ```

   这会生成一个 Webhook GitHub URL，如下所示：

   输出示例

   ```
   <https://api.starter-us-east-
   1.openshift.com:443/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<na
   me>/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. 提供含有有效 JSON 内容的文件后，如 **payload.json**，您可以使用 **curl** 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>/buildcon
   gs/<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。

其他资源

- Gogs

2.8.1.1.2. 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
```

流程

1. 配置 GitLab Webhook：
   a. 描述 **BuildConfig** 以获取 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. 提供含有有效 JSON 内容的文件后，如 **payload.json**，您可以使用 **curl** 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>/buildconfigs/<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。

2.8.1.1.3. 使用 Bitbucket Webhook

当存储库更新时，Bitbucket Webhook 处理 Bitbucket 发出的调用。与前面的触发器类似，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/<name>/webhooks/<secret>/bitbucket
```

-

流程

1. 配置 Bitbucket Webhook：
   a. 描述 BuildConfig 以获取 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. 提供含有有效 JSON 内容的文件后，如 **payload.json**，您可以使用 **curl** 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。

2.8.1.1.4. 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：

   输出示例

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

   调用者必须以 **POST** 操作形式调用 Webhook。

2. 要手动调用 Webhook，您可以使用 **curl**：

   ```
   $ 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，再运行以下命令：

```
$ 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>/buildcon
gs/<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。

2.8.1.1.5. 显示 Webhook URL

您可以使用以下命令来显示与构建配置关联的 Webhook URL。如果命令不显示任何 Webhook URL，则没有为该构建配置定义任何 Webhook 触发器。

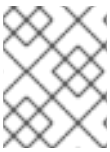
流程

- 显示与 **BuildConfig** 关联的任何 Webhook URL：

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

2.8.1.2. 使用镜像更改触发器

通过镜像更改触发器，您可以在上游镜像有新版本可用时自动调用构建。例如，如果构建以 RHEL 镜像为基础，那么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

配置镜像更改触发器需要以下操作：

1. 定义指向要触发的上游镜像的 **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. 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。

2.8.1.3.1. 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，请使用：

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

- 要设置镜像更改触发器，请使用：

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

- 要移除触发器，请添加 **--remove**：

```
$ 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

配置提交后构建 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

- 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 时可使用此选项。例如，用于运行前文所述的单元测试。若要控制镜像入口点，或者如果镜像没有 **/bin/sh**，可使用 **command** 和/或 **args**。

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。

- 命令作为镜像入口点：

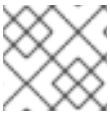
```
postCommit:
  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]
```

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。

- 命令带有参数：

```
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. 将命令设置为提交后构建 hook：

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

2. 将脚本设置为提交后构建 hook：

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

2.9. 执行高级构建

以下小节提供了有关高级构建操作的说明，包括设置构建资源和最长持续时间、将构建分配给节点、串联构建、修剪构建，以及构建运行策略。

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。
- 在构建配置中通过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
```

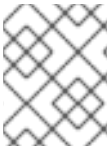
注意

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
```

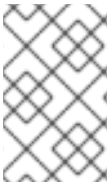
(1) 与此构建配置关联的构建将仅在具有 **key1=value2** 和 **key2=value2** 标签的节点上运行。

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

按照以下小节中的内容在 OpenShift Container Platform 上运行授权构建。

2.10.1. 为红帽通用基础镜像创建镜像流标签

要在构建中使用红帽订阅，您可以创建一个镜像流标签来引用通用基础镜像（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。

以这种方式使用镜像流标签的好处是，根据安装 pull secret 中的 **registry.redhat.io** 凭证授予对 UBI 的访问权限，而不会向其他用户公开 pull secret。这比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**，因此所有项目中的开发人员可使用它，请输入：

```
$ oc tag --source=docker registry.redhat.io/ubi7/ubi:latest ubi:latest -n openshift
```

- 要在单个项目中创建 **ImageStreamTag**，请输入：

```
$ oc tag --source=docker registry.redhat.io/ubi7/ubi:latest ubi:latest
```

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

您的订阅必须可以访问红帽权利，而且权利必须具有单独的公钥和私钥文件。

提示

使用 Red Hat Enterprise Linux(RHEL)7 执行 Entitlement Build 时，在运行任何 **yum** 命令前，必须在 Dockerfile 中包含以下指令：

```
RUN rm /etc/rhsm-host
```

流程

1. 创建包含权利的 secret，确保存在含有权利公钥和私钥的单独文件：

```
$  oc create secret generic etc-pki-entitlement --from-file /path/to/entitlement/{ID}.pem \
> --from-file /path/to/entitlement/{ID}-key.pem ...
```

2. 在构建配置中将 secret 添加为构建输入：

```
source:
  secrets:
  - secret:
      name: etc-pki-entitlement
    destinationDir: 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 Subscription Manager 来安装订阅内容。

先决条件

必须添加授权密钥、Subscription Manager 配置和 Subscription Manager 证书颁发机构，作为构建输入。

流程

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

```
FROM registry.redhat.io/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy subscription manager configurations
COPY ./rhsm-conf /etc/rhsm
COPY ./rhsm-ca /etc/rhsm/ca
# Delete /etc/rhsm-host to use entitlements from the build container
RUN rm /etc/rhsm-host && \
    # Initialize /etc/yum.repos.d/redhat.repo
    # See https://access.redhat.com/solutions/1443553
    yum repolist --disablerepo=* && \
    subscription-manager repos --enable <enabled-repo> && \
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements and Subscription Manager configs
    rm -rf /etc/pki/entitlement && \
    rm -rf /etc/rhsm
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

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. 创建包含 Satellite 存储库配置文件的 **ConfigMap**：

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

2. 在 **BuildConfig** 中添加 Satellite 存储库配置：

```
source:
  configMaps:
  - configMap:
      name: yum-repos-d
    destinationDir: yum.repos.d
```

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

- 必须将权利密钥和 Satellite 存储库配置添加为构建输入。

流程

使用以下示例 Dockerfile 来通过 Satellite 安装内容：

```
FROM registry.redhat.io/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy repository configuration
COPY ./yum.repos.d /etc/yum.repos.d
# Delete /etc/rhsm-host to use entitlements from the build container
RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf
(1)
#RUN cat /etc/yum/pluginconf.d/subscription-manager.conf
RUN yum clean all
```

```
#RUN yum-config-manager
RUN rm /etc/rhsm-host && \
    # yum repository info provided by Satellite
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements
    rm -rf /etc/pki/entitlement
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

1 如果使用 **enabled=1** 在构建中添加 Satellite 配置失败，请将 **RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf** 添加到 Dockerfile。

2.10.5. 其他资源

- 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. 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. 移除角色：

```
$ 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

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。

参数	描述
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

- 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 并清除服务注解 service **.beta.openshift.io/serving-cert-generation-error 和 service.beta.openshift.io/serving-cert-generation-error-num，以强制重新生成** 证书：

```
$ 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

- 您必须具有集群管理员特权。
- 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. RED HAT OPENSHIFT PIPELINES 发行注记

Red Hat OpenShift Pipelines 是基于 Tekton 项目的一个云原生 CI/CD 环境，它提供：

- 标准 Kubernetes 原生管道定义 (CRD)。
- 无需 CI 服务器管理开销的无服务器管道。
- 使用任何 Kubernetes 工具（如 S2I、Buildah、JIB 和 Kaniko）构建镜像。
- 不同 Kubernetes 发布系统间的可移植性。
- 用于与管道交互的强大 CLI。
- 使用 OpenShift Container Platform Web 控制台的 **Developer** 视角集成用户体验。

如需了解 Red Hat OpenShift Pipelines 的概述，请参阅了解 OpenShift Pipelines。

3.1.1. 使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 Red Hat CTO Chris Wright 信息。

3.1.2. Red Hat OpenShift Pipelines 正式发布 1.4 发行注记

Red Hat OpenShift Pipelines General Availability（GA）1.4 现在包括在 OpenShift Container Platform 4.7 中。

注意

除了 stable 和 preview Operator 频道外，Red Hat OpenShift Pipelines Operator 1.4.0 还带有 ocp-4.6、ocp-4.5 和 ocp-4.4 弃用的频道。这些过时的频道及其支持将在以下 Red Hat OpenShift Pipelines 发行版本中删除。

3.1.2.1. 兼容性和支持列表

这个版本中的一些功能当前还处于技术预览状态。它们并不适用于在生产环境中使用。

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 3.1. 兼容性和支持列表

功能	Version	支持状态
Pipelines	0.22	GA
CLI	0.17	GA
Catalog	0.22	GA
触发器	0.12	TP
Pipeline 资源	-	TP

如果您有疑问或希望提供反馈信息，请向产品团队发送邮件 pipelines-interest@redhat.com。

3.1.2.2. 新功能

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift Pipelines 1.4 中的新内容。

- 自定义任务有以下改进：
  - Pipeline 结果现在可以引用自定义任务生成的结果。
  - 自定义任务现在可以使用工作区、服务帐户和 pod 模板来构建更复杂的自定义任务。
- **finally** 的任务包括以下改进：
  - 在 **finally** 任务中支持 **when** 表达式，它可以有效地保证执行，并提高了任务的可重复使用性。
  - **finally** 任务可以被配置为消耗同一管道中任何任务的结果。

注意

OpenShift Container Platform 4.7 web 控制台中不支持 **when** 表达式和 **finally** 任务。

- 添加了对 **dockercfg** 或 **dockerconfigjson** 类型的多个 secret 的支持，以便在运行时进行身份验证。
- 添加支持 **git-clone** 任务的 sparse-checkout 的功能。这可让您将存储库的子集克隆为本地副本，并帮助您限制克隆的存储库的大小。
- 您可以创建管道以待处理状态运行，而无需实际启动它们。在负载非常重的集群中，这允许 Operator 控制管道运行的开始时间。
- 确保为控制器手动设置 **SYSTEM_NAMESPACE** 环境变量；这在之前被默认设置。
- 现在，一个非 root 用户被添加到管道的构建基础镜像中，以便 **git-init** 能够以非 root 用户身份克隆存储库。

- 支持在管道运行启动前在已解析的资源间验证依赖项。管道中的所有结果变量都必须有效，管道中的可选工作区只能传递给期望管道启动运行的任务。
- controller 和 Webhook 作为非 root 组运行，并且删除了它们的多余功能，以使它们更加安全。
- 您可以使用 **tkn pr logs** 命令查看重试任务运行的日志流。
- 您可以使用 **tkn tr delete** 命令中的 **--clustertask** 选项删除与特定集群任务关联的所有任务。
- 通过引入一个新的 **customResource** 字段，增加了对在 **EventListener** 资源中使用 Knative 服务的支持。
- 当事件有效负载没有使用 JSON 格式时，会显示错误消息。
- 源控制拦截器（如 GitLab、BitBucket 和 GitHub）现在使用新的 **InterceptorRequest** 或 **InterceptorResponse** 类型的接口。
- 新的 CEL 功能 **marsagingJSON** 被实现，以便您可以将 JSON 对象或数组编码到字符串。
- 为 CEL 添加了 HTTP 处理器，并添加了源控制内核拦截器。它将四个核心拦截器打包到单一 HTTP 服务器中，该服务器部署在 **tekton-pipelines** 命名空间中。**EventListener** 对象通过 HTTP 服务器将事件转发到拦截器。每个拦截器都位于不同的路径。例如，CEL 拦截器位于 **/cel** 路径中。
- **pipelines-scc** 安全性上下文约束（SCC）与默认的 **pipeline** 服务帐户一同使用。此新服务帐户与 **anyuid** 类似，但 OpenShift Container Platform 4.7 的 SCC 在 YAML 中定义中有一个小的差别：

```
fsGroup:
  type: MustRunAs
```

3.1.2.3. 已弃用的功能

- 不支持 pipeline 资源存储中的 **build-gcs** 子类型和 **gcs-fetcher** 镜像。
- 在集群任务的 **TaskRun** 字段中，删除标签 **tekton.dev/task**。
- 对于 webhook，移除了与字段 **admissionReviewVersions** 对应的 **v1beta1** 值。
- 用于构建和部署的 **creds-init** 帮助程序镜像已被删除。
- 在触发器 spec 和绑定中，弃用的字段 **template.name** 已被删除，并使用 **template.ref** 替代。您应该更新所有 **eventListener** 定义，以使用 **ref** 字段。

注意

从 Pipelines 1.3.x 和早期版本升级到 Pipelines 1.4.0 会中断事件监听程序，因为 **template.name** 字段不可用。在这种情况下，使用 Pipelines 1.4.1 来提供恢复的 **template.name** 字段。

- 对于 **EventListener** 自定义资源/对象，**PodTemplate** 和 **ServiceType** 字段已弃用，并使用 **Resource** 替代。
- 过时的 spec 风格内嵌绑定已被删除。
- **spec** 字段已从 **triggerSpecBinding** 中删除。

- 事件 ID 已从包括 5 个字符的随机字符串改为 UUID。

3.1.2.4. 已知问题

- 在 **Developer** 视角中，管道指标和触发器功能仅适用于 OpenShift Container Platform 4.7.6 或更高版本。
- 在 IBM Power Systems、IBM Z 和 LinuxONE 中，不支持 **tkn hub** 命令。
- 当您在 IBM Power Systems（ppc64le）、IBM Z 和 LinuxONE（s390x）集群上运行 Maven 和 Jib Maven 集群任务时，将 **MAVEN_IMAGE** 参数值设置为 **maven:3.6.3-adoptopenjdk-11**。
- 如果您在触发器绑定中有以下配置，触发器会因为不正确处理 JSON 格式抛出错误：

```
params:
  - name: github_json
    value: $(body)
```

  要解决这个问题：

  - 如果您使用触发器 v0.11.0 及更高版本，请使用 **marshalJSON** CEL 函数，该函数使用 JSON 对象或数组，并将该对象或数组的 JSON 编码作为字符串返回。
  - 如果使用旧的触发器版本，请在触发器模板中添加以下注解：

```
annotations:
  triggers.tekton.dev/old-escape-quotes: "true"
```

- 当从 Pipelines 1.3.x 升级到 1.4.x 时，您必须重新创建路由。

3.1.2.5. 修复的问题

- 在以前的版本中，**tekton.dev/task** 标签已从集群任务运行中删除，并且引进了 **tekton.dev/clusterTask** 标签。此更改导致的问题可以通过修复 **clustertask describe** 和 **delete** 命令来解决。另外，也修改了任务的 **lastrun** 功能，从而解决应用到在旧版管道中运行任务中的 **tekton.dev/task** 标签的问题。
- 当进行交互式 **tkn pipeline start pipelinename** 时，会以互动方式创建一个 **PipelineResource**。如果资源状态不是 **nil**，**tkn p start** 命令会输出资源状态。
- 在以前的版本中，**tekton.dev/task=name** 标签已从集群任务创建的任务中删除。在这个版本中，使用 **--last** 标志修改 **tkn clustertask start** 命令，以检查所创建的任务运行中的 **tekton.dev/task=name** 标签。
- 当任务使用内联任务规格时，在运行 **tkn pipeline describe** 命令时对应的任务运行会被嵌入到管道中，任务名称返回为内嵌。
- 修复了 **tkn version** 命令，显示已安装的 Tekton CLI 工具的版本，无需配置的 **kubeConfiguration** 命名空间或对集群的访问。
- 如果使用意外的参数或者使用多个参数，则 **tkn completion** 命令会出错。
- 在以前的版本中，当管道转换为 **v1alpha1** 版本并恢复到 **v1beta1** 版本时，带有嵌套在一个管道规格中的 **finally** 的任务将会丢失那些 **finally** 任务。修复了转换过程中发生的这个错误，以避免潜在的数据丢失。现在，带有嵌套在管道规格中的 **finally** 任务的管道运行会被序列化并存储在 alpha 版本中，它们只会在以后进行反序列化。

- 在以前的版本中，当服务帐户中的 **secrets** 字段被设置为 **{}** 时，pod 生成中会出现一个错误。任务运行失败，显示 **CouldntGetTask**，因为带有空 secret 名称的 GET 请求返回了一个错误，表示资源名称可能不是空的。这个问题已通过避免 **kubeclient** GET 请求中的空 secret 名称来解决。

- 现在，可以请求带有 **v1beta1** API 版本的管道和 **v1alpha1** 版本，而不会丢失 **finall** 任务。应用返回的 **v1alpha1** 版本将以 **v1beta1** 来保存资源，**finally** 部分恢复到其原始状态。

- 在以前的版本中，控制器中的一个未设置的 **selfLink** 字段在 Kubernetes v1.20 集群中造成错误。作为一个临时修复，在没有自动填充的 **selfLink** 字段的值时，**CloudEvent** source 字段被设置为与当前源 URI 匹配的值。

- 在以前的版本中，带有点（如 **gcr.io** ）的 secret 名称会导致任务运行创建失败。这是因为内部使用的 secret 名称作为卷挂载名称的一部分。卷挂载名称遵循 RFC1123 DNS 标签标准，它不允许使用点作为名称的一部分。这个问题已通过将点替换为横线来解决。

- 现在，上下文变量会在 **finally** 任务中进行验证。

- 在以前的版本中，当任务运行协调器传递了一个任务运行，且没有之前的状态更新，其中包含它创建的 pod 的名称时，任务运行协调程序会列出与任务运行关联的 pod。任务运行协调程序使用任务运行标签（被传播到 pod）来查找 pod。在任务运行期间更改这些标签，会导致代码找不到现有的 pod。因此，会创建重复的 pod。这个问题已通过在查找 pod 时将任务运行协调器更改为只使用 **tekton.dev/taskRun** Tekton 控制的标签来解决。

- 在以前的版本中，当管道接受一个可选的工作区并将其传递给管道任务时，如果未提供工作区，管道运行协调器会停止并出错，即使缺少的工作区绑定是可选工作区的有效状态。这个问题已被解决，确保管道运行的协调器不会无法创建任务运行，即使未提供可选的工作区。

- 排序步骤状态的顺序与步骤容器的顺序匹配。

- 在以前的版本中，当 Pod 遇到 **CreateContainerConfigError** 原因时，任务运行状态被设置为 **unknown**，这意味着任务和管道会运行，直到 pod 超时为止。这个问题已通过将任务运行状态设置为 **false** 来解决这个问题，因此当 pod 遇到 **CreateContainerConfigError** 原因时，任务被设置为 failed。

- 在以前的版本中，管道结果会在管道运行完成后在第一次协调时解析。这可能会导致管道运行的 **Succeeded** 条件被覆盖。因此，最终状态信息丢失，可能会使监视管道运行条件的任何服务混淆。当管道运行进入 **Succeeded** 或 **True** 条件时，这个问题可以通过将管道结果解析移到协调结束时来解决。

- 现在，执行状态变量已被验证。这可避免在验证上下文变量来访问执行状态时验证任务结果。

- 在以前的版本中，包含无效变量的管道结果将添加到管道运行中，并包含变量的字面表达式。因此，很难评估结果是否正确填充。这个问题已通过过滤管道运行结果来解决，该结果引用了失败的任务运行。现在，包含无效变量的管道结果将完全不会被管道运行发送。

- 现在，**tkn eventlistener describe** 命令已被修复，以避免在没有模板的情况下崩溃。它还显示有关触发器引用的详情。

- 由于 **template.name** 不可用，从 Pipelines 1.3.x 及早期版本升级到 Pipelines 1.4.0 会破坏事件监听程序。在 Pipelines 1.4.1 中，**template.name** 已恢复，以避免触发器中的事件监听程序。

- 在 Pipelines 1.4.1 中，**ConsoleQuickStart** 自定义资源已更新，以符合 OpenShift Container Platform 4.7 的功能和行为。

3.1.3. Red Hat OpenShift Pipelines 技术预览 1.3 发行注记

3.1.3.1. 新功能

Red Hat OpenShift Pipelines 技术预览（TP）1.3 现在包括在 OpenShift Container Platform 4.7 中。Red Hat OpenShift Pipelines TP 1.3 更新为支持：

- Tekton Pipelines 0.19.0
- Tekton **tkn** CLI 0.15.0
- Tekton Triggers 0.10.2
- 基于 Tekton Catalog 0.19.0 的集群任务
- OpenShift Container Platform 4.7 中的 IBM Power Systems
- OpenShift Container Platform 4.7 上的 IBM Z 和 LinuxONE

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift Pipelines 1.3 中的新内容。

3.1.3.1.1. Pipelines

- 构建镜像的任务，如 S2I 和 Buildah 任务，现在发出构建的镜像 URL，其中包含镜像 SHA。
- 由于 **Conditions** 自定义资源定义（CRD）已弃用，管道任务中引用自定义资源的条件会被禁止。
- 现在，在 **Task** CRD 中为以下字段添加了变量扩展： **spec.steps[].imagePullPolicy** 和 **spec.sidecar[].imagePullPolicy**。
- 您可以通过将 **disable-creds-init** feature-flag 设置为 **true** 来禁用 Tekton 中的内置凭证机制。
- 现在，当在 **PipelineRun** 配置的 **Status** 字段中的 **Skipped Tasks** 和 **Task Runs** 部分中列出了表达式时，可以解析。
- **git init** 命令现在可以克隆递归子模块。
- 现在，**Task** CR 的作者可以为 **Task** spec 的一个步骤指定超时。
- 现在，您可以将入口点镜像基于 **distroless/static:nonroot** 镜像，赋予将其复制到目的地的模式，而无需依赖基础镜像中存在的 **cp** 命令。
- 现在，您可以使用配置标记 **require-git-ssh-secret-known-hosts** 来禁止在 Git SSH secret 中省略已知主机。当标志值设为 **true** 时，必须在 Git SSH secret 中包含 **known_host** 字段。标志的默认值为 **false**。
- 现在引进了可选工作区的概念。任务或管道可能会声明一个工作区（workspace），并有条件地更改其行为。任务运行或管道运行可能省略了工作区，因此修改任务或管道行为。默认任务运行工作区不会添加到忽略的可选工作区。
- 现在，在 Tekton 中进行凭证初始化会检测一个与非 SSH URL 搭配使用的 SSH 凭证，而 Git pipeline 资源与 Git pipeline 资源相同，并在步骤容器中记录警告。
- 如果 pod 模板指定的关联性被关联性代理覆盖，则任务运行控制器会发出警告事件。
- 任务运行协调程序现在记录了在任务运行完成后发送的云事件的指标。这包括重试。

3.1.3.1.2. Pipelines CLI

- 现在，在以下命令中添加了对 **--no-headers flag** 的支持： **tkn condition list**、**tkn triggerbinding list**、**tkn eventlistener list**、**tkn clustertask list**、**tkn clustertriggerbinding list**。
- 当一起使用时，**--last** 或 **--use** 选项会覆盖 **--prefix-name** 和 **--timeout** 选项。
- 现在，添加了 **tkn eventlistener logs** 命令来查看 **EventListener** 日志。
- **tekton hub** 命令现在被集成到 **tkn** CLI。
- **--nocolour** 选项现在改为 **--no-color**。
- **--all-namespaces** 标志添加到以下命令中： **tkn triggertemplate list**、**tkn condition list**、**tkn triggerbinding list**、**tkn eventlistener list**。

3.1.3.1.3. 触发器

- 现在，您可以在 **EventListener** 模板中指定资源信息。
- 现在，**EventListener** 服务帐户除具有所有触发器资源的 **get** verb 外，还具有 **list** 和 **watch** verb。这可让您使用 **Listers** 从 **EventListener**、**Trigger**、**TriggerBinding**、**TriggerTemplate** 和 **ClusterTriggerBinding** 资源中获取数据。您可以使用此功能创建 **Sink** 对象，而不是指定多个通知器，直接向 API 服务器发出调用。
- 添加了一个新的 **Interceptor** 接口，以支持不可变的输入事件正文。拦截器现在可以在一个新的 **extensions** 字段中添加数据或字段，且无法修改输入正文使其不可变。CEL 拦截器使用这个新的 **Interceptor** 接口。
- 在 **EventListener** 资源中添加了一个 **namespaceSelector** 字段。使用它来指定 **EventListener** 资源可以从中获取用于处理事件的 **Trigger** 对象的命名空间。要使用 **namespaceSelector** 字段，**EventListener** 资源的服务帐户必须具有集群角色。
- 触发器 **EventListener** 资源现在支持到 **eventlistener** pod 的端到端安全连接。
- 在 **TriggerTemplates** 资源中把 **''** 替换为 **\''** 的转义行为现在已被删除。
- 一个支持 Kubernetes 资源的、新的 **resources** 项已作为 **EventListener** spec 的一部分被添加。
- 添加了对 CEL 拦截器的新功能，它支持 ASCII 字符串的大写和小写。
- 您可以使用触发器中的 **name** 和 **value** 字段，或事件监听程序来嵌入 **TriggerBinding** 资源。
- **PodSecurityPolicy** 配置已更新，可在受限环境中运行。它确保容器必须以非 root 运行。另外，使用 Pod 安全策略的基于角色的访问控制也从集群范围移到命名空间范围。这样可确保触发器无法使用与命名空间不相关的其他 Pod 安全策略。
- 现在，添加了对内嵌触发器模板的支持。您可以使用 **name** 字段来指代嵌入的模板，或者在 **spec** 字段中嵌入模板。

3.1.3.2. 已弃用的功能

- 使用 **PipelineResources** CRD 的管道模板现已弃用，并将在以后的发行版本中删除。
- **template.name** 字段已弃用，被 **template.ref** 字段替代，并将在以后的发行版本中删除。

- 使用 **-c** 作为 **--check** 命令的缩写已被删除。另外，全局 **tkn** 标志被添加到 **version** 命令中。

3.1.3.3. 已知问题

- CEL 覆盖在新的顶层 **extensions** 功能中添加字段，而不是修改传入的事件正文。**TriggerBinding** 资源可以使用 **$(extensions.<key>)** 语法访问这个新 **extensions** 功能中的值。更新您的绑定，使用 **$(extensions.<key>)** 语法而不是 **$(body.<overlay-key>)** 语法。

- 把 **"** 替换为 **\"** 的参数转义行为现在已被删除。如果您需要保留旧的转义参数行为，请在 **TriggerTemplate** 规格中添加 **tekton.dev/old-escape-quotes: true"** 注解。

- 您可以使用触发器中的 **name** 和 **value** 字段，或事件监听程序来嵌入 **TriggerBinding** 资源。但是，您无法为单个绑定指定 **name** 和 **ref** 字段。使用 **ref** 字段引用 **TriggerBinding** 资源以及内嵌绑定的 **name** 字段。

- 拦截器无法试图引用 **EventListener** 资源命名空间以外的 **secret**。您必须在 'EventListener' 资源的命名空间中包含 secret。

- 在 Triggers 0.9.0 及之后的版本中，如果基于正文或标头的 **TriggerBinding** 参数在事件有效负载中缺失或格式不正确，则使用默认值而不是显示错误。

- 通过使用 Tekton Pipelines 0.16.x 的 **WhenExpression** 对象创建的任务和管道必须重新应用来修复它们的 JSON 注解。

- 当管道接受一个可选的工作区，并将其提供给某个任务时，如果未提供工作区，管道会运行停止。

- 要在断开连接的环境中使用 Buildah 集群任务，请确保 Dockerfile 使用作为基础镜像的内部镜像流，然后使用与任何 S2I 集群任务相同的方法。

3.1.3.4. 修复的问题

- CEL 拦截器添加的扩展通过在事件正文中添加 **Extensions** 字段传递给 Webhook 拦截器。

- 现在，日志读取器的活动超时可以使用 **LogOptions** 字段进行配置。但是，10 秒的默认超时行为会被保留。

- 当一个任务运行或管道运行完成后，**log** 命令会忽略 **--follow** 标记，它会读取可用的日志而不是实时的日志。

- 对以下 Tekton 资源的引用现在标准化，并在 **tkn** 命令中的所有面向用户的信息中保持一致：**EventListener**、**TriggerBinding**、**ClusterTriggerBinding**、**Condition** 和 **TriggerTemplate**。

- 在以前的版本中，如果您启动了使用 **--use-taskrun <canceled-task-run-name>**、**--use-pipelinerun <canceled-pipeline-run-name>** 或 **--last** 标记的、已被取消的任务运行或管道运行，新的运行将会被取消。这个程序漏洞现已解决。

- 现在，**tkn pr desc** 命令已被改进，以便在管道有条件运行时不会失败。

- 当使用 **--task** 选项删除使用 **tkn tr delete** 运行的任务时，且集群任务具有相同名称的集群任务存在时，集群任务运行的任务也会被删除。作为临时解决方案，使用 **TaskRefKind** 字段过滤运行的任务。

- **tkn triggertemplate describe** 命令在输出中只会显示 **apiVersion** 值的一部分。例如，只显示 **triggers.tekton.dev**，而不是 **triggers.tekton.dev/v1alpha1**。这个程序漏洞现已解决。

- 在某些情况下，webhook 无法获取租期且无法正常工作。这个程序漏洞现已解决。

- 在 v0.16.3 中创建的带有 when 表达式的管道现在可以在 v0.17.1 及之后的版本中运行。升级后，您不需要重新应用之前版本中创建的管道定义，因为现在支持注解中第一个字母的大写和小写。
- 默认情况下，**leader-election-ha** 字段为高可用性启用。当 **disable-ha** 控制器标记设置为 **true** 时，它会禁用高可用性支持。
- 现在，解决了重复云事件的问题。现在，只有在条件改变状态、原因或消息时才发送云事件。
- 当 **PipelineRun** 或 **TaskRun** spec 中没有服务帐户名称时，控制器会使用 **config-defaults** 配置映射中的服务帐户名称。如果在 **config-defaults** 配置映射中也缺少服务帐户名称，控制器现在会在 spec 中将其设置为 **default**。
- 现在，当同一个持久性卷声明用于多个工作区，但具有不同子路径时，支持验证是否与关联性偏好兼容。

3.1.4. Red Hat OpenShift Pipelines 技术预览 1.2 发行注记

3.1.4.1. 新功能

Red Hat OpenShift Pipelines 技术预览（TP）1.2 现在包括在 OpenShift Container Platform 4.6 中。Red Hat OpenShift Pipelines TP 1.2 更新为支持：

- Tekton Pipelines 0.16.3
- Tekton **tkn** CLI 0.13.1
- Tekton Triggers 0.8.1
- 基于 Tekton Catalog 0.16 的集群任务
- OpenShift Container Platform 4.6 中的 IBM Power Systems
- OpenShift Container Platform 4.6 上的 IBM Z 和 LinuxONE

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift Pipelines 1.2 中的新内容。

3.1.4.1.1. Pipelines

- 此 Red Hat OpenShift Pipelines 发行版本添加了对断开连接的安装的支持。

注意

IBM Power Systems、IBM Z 和 LinuxONE 目前不支持在受限环境中安装。

- 现在，您可以使用 **when** 字段而不是 **conditions** 资源，仅在满足特定条件时运行任务。**WhenExpression** 资源的关键组件是 **Input**、**Operator** 和 **Values**。如果所有表达式都评估的结果都为 **True**，则任务运行。如果表达式评估的结果为 **False**，则任务被跳过。
- 现在，如果某个任务运行被取消或超时，则步骤（Step）状态被更新。
- 现在，支持 Git 大文件存储（LFS）来使用 **git-init** 构建基础镜像。
- 现在，当某个任务嵌入到管道中时，您可以使用 **taskSpec** 字段来指定元数据，如标识（label）和注解（annotation）。

- 现在，Pipeline 运行支持云事件。现在，对于云事件管道资源发送的带有 **backoff** 的云事件会进行重试。
- 现在，可以为声明了 **Task**、但没有明确指定 **TaskRun** 资源的工作区（workspace）设置一个默认的 **Workspace** 配置。
- 支持 **PipelineRun** 命名空间和 **TaskRun** 命名空间的命名空间变量插入。
- 现在，添加了对 **TaskRun** 对象的验证，以检查当 **TaskRun** 资源与 Affinity Assistant 关联时，是否使用一个以上的持久性卷声明工作区。如果使用多个持久性卷声明工作区，则任务运行会失败，并且有一个 **TaskRunValidationFailed** 条件。请注意，默认情况下， Affinity Assistant 在 Red Hat OpenShift Pipelines 中被禁用，因此您需要启用 Affinity Assistant 来使用它。

3.1.4.1.2. Pipelines CLI

- **tkn task describe**、**tkn taskrun describe**、**tkn clustertask describe**、**tkn pipeline describe** 和 **tkn pipelinerun describe** 命令现在：
  - 如果存在其中之一，会自动选择 **Task**、**TaskRun**、**ClusterTask**、**Pipeline** 和 **PipelineRun**。
  - 在相应的输出中显示 **Task**、**TaskRun**、**ClusterTask**、**Pipeline** 和 **PipelineRun** 资源的结果。
  - 在相应的输出中显示 **Task**、**TaskRun**、**ClusterTask**、**Pipeline** 和 **PipelineRun** 资源中声明的工作区。
- 现在，您可以使用 **tkn clustertask start** 命令的 **--prefix-name** 选项指定任务运行名称前缀。
- 现在为 **tkn clustertask start** 命令提供了互动模式支持。
- 现在，您可以使用 **TaskRun** 和 **PipelineRun** 对象的本地或远程文件定义指定管道支持的 **PodTemplate** 属性。
- 现在，您可以在 **tkn clustertask start** 命令中使用 **--use-params-defaults** 选项，使用 **ClusterTask** 配置中设置的默认值并创建任务运行。
- 现在，如果有些参数没有指定默认值，**tkn pipeline start** 命令的 **--use-param-defaults** 标志会提示以互动模式提供。

3.1.4.1.3. 触发器

- 添加了一个名为 **parseYAML** 的通用表达语言（CEL）函数，用来将 YAML 字符串解析为一个映射的字符串。
- 在评估表达式和解析 hook 正文以创建评估环境时，改进了解析 CEL 表达式的错误消息，使其更加精细。
- 现在，可以支持 marsing 布尔值和映射，如果它们被用作 CEL 覆盖机制中的表达式值。
- 在 **EventListener** 对象中添加了以下字段：
  - **replicas** 字段通过在 YAML 文件中指定副本数，使事件监听程序能够运行多个 pod。
  - **NodeSelector** 字段使 **EventListener** 对象能够将事件监听器 pod 调度到特定的节点。

- Webhook 拦截器现在可以解析 **EventListener-Request-URL** 标头，从事件监听器处理的原始请求 URL 中提取参数。
- 现在，事件监听器的注解可以被传播到部署、服务和其他 pod。请注意，服务或部署的自定义注解将被覆盖，因此必须在事件监听程序注解中添加它们以便传播它们。
- 现在，当用户将 **spec.replicas** 值指定为 **负数** 或 **零**时，可以正确验证 **EventListener** 规格中的副本。
- 现在，您可以在 **EventListener** spec 中指定 **TriggerCRD** 项，作为一个使用 **TriggerRef** 项的引用来独立创建 **TriggerCRD** 项，然后在**EventListener** spec 中绑定它。
- 现在，提供了对 **TriggerCRD** 对象的验证和默认值。

3.1.4.2. 已弃用的功能

- **$(params)** 参数现已从 **triggertemplate** 资源中删除，由 **$(tt.params)** 替代，以避免 **resourcetemplate** 和 **triggertemplate** 资源参数间的混淆。
- 基于可选的基于 **EventListenerTrigger** 的身份验证级别的 **ServiceAccount** 引用，已从对象引用改为一个 **ServiceAccountName** 字符串。这样可确保 **ServiceAccount** 引用与 **EventListenerTrigger** 对象位于同一个命名空间中。
- **Conditions** 自定义资源定义（CRD）现已弃用，已使用 **WhenExpressions** CRD 替代。
- **PipelineRun.Spec.ServiceAccountNames** 对象已启用，被 **PipelineRun.Spec.TaskRunSpec[].ServiceAccountName** 对象替代。

3.1.4.3. 已知问题

- 此 Red Hat OpenShift Pipelines 发行版本添加了对断开连接的安装的支持。但是，集群任务使用的一些镜像必须进行镜像（mirror）才能在断开连接的集群中工作。
- 在卸载 Red Hat OpenShift Pipelines Operator 后，**openshift** 命名空间中的管道不会被删除。使用 **oc delete pipelines -n openshift --all** 命令删除管道。
- 卸载 Red Hat OpenShift Pipelines Operator 不会删除事件监听程序。
  作为临时解决方案，删除 **EventListener** 和 **Pod** CRD:

  1. 使用 **foregroundDeletion** 终结器编辑 **EventListener** 对象：

     ```
     $ oc patch el/<eventlistener_name> -p '{"metadata":{"finalizers":["foregroundDeletion"]}}'
     --type=merge
     ```

     例如：

     ```
     $ oc patch el/github-listener-interceptor -p '{"metadata":{"finalizers":
     ["foregroundDeletion"]}}' --type=merge
     ```

  2. 删除 **EventListener** CRD:

     ```
     $ oc patch crd/eventlisteners.triggers.tekton.dev -p '{"metadata":{"finalizers":[]}}' --
     type=merge
     ```

- 当您运行多架构容器镜像任务时，如果在 IBM Power Systems(ppc64le)或 IBM Z(s390x)集群上没有命令规格，则 **TaskRun** 资源会失败，并显示以下错误：

  ```
  Error executing command: fork/exec /bin/bash: exec format error
  ```

  作为临时解决方案，使用特定架构的容器镜像或指定 sha256 摘要指向正确的架构。要获得 sha256 摘要，请输入：

  ```
  $ skopeo inspect --raw <image_name>| jq '.manifests[] | select(.platform.architecture == "
  <architecture>") | .digest'
  ```

3.1.4.4. 修复的问题

- 现在，添加了一个简单的语法验证用于检查 CEL 过滤器、Webhook 验证器中的覆盖以及拦截器中的表达式。
- 触发器不再覆盖底层部署和服务对象上的注解。
- 在以前的版本中，事件监听器将停止接受事件。**EventListener** sink 增加了一个 120 秒的空闲超时来解决这个问题。
- 在以前的版本中，取消一个带有 **Failed(Canceled)** 状态的管道运行会给出一个成功信息。这个问题已被解决，现在在这种情况下会显示错误。
- **tkn eventlistener list** 命令现在提供列出的事件监听器的状态，从而使您可以轻松地识别可用的事件。
- 现在，当没有安装触发器或没有找到资源时，**triggers list** 和 **triggers describe** 命令会显示一致的错误信息。
- 在以前的版本中，在云事件交付过程中会产生大量闲置连接。**DisableKeepAlives: true** 参数添加到 **cloudeventclient** 配置来修复这个问题。因此，会为每个云事件设置一个新的连接。
- 在以前的版本中，**creds-init** 代码也会向磁盘写入空文件，即使未提供给定类型的凭证。在这个版本中，**creds-init** 代码只为从正确注解的 secret 中挂载的凭证写入文件。

3.1.5. Red Hat OpenShift Pipelines 技术预览 1.1 发行注记

3.1.5.1. 新功能

Red Hat OpenShift Pipelines 技术预览（TP）1.1 现在包括在 OpenShift Container Platform 4.5 中。Red Hat OpenShift Pipelines TP 1.1 更新为支持：

- Tekton Pipelines 0.14.3
- Tekton **tkn** CLI 0.11.0
- Tekton Triggers 0.6.1
- 基于 Tekton Catalog 0.14 的集群任务

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift Pipelines 1.1 中的新内容。

3.1.5.1.1. Pipelines

- 现在可以使用工作区而不是管道资源。建议您在 OpenShift Pipelines 中使用 Workspaces 而不是 PipelineResources，因为 PipelineResources 很难调试，范围有限，且不容易重复使用。如需有关 Workspaces 的更多信息，请参阅了解 OpenShift Pipelines。

- 添加了对卷声明模板的工作空间支持：

  - 管道运行和任务运行的卷声明模板现在可以添加为工作区的卷源。然后，tkton-controller 使用模板创建一个持久性卷声明（PVC），该模板被视为管道中运行的所有任务的 PVC。因此，您不需要在每次绑定多个任务的工作空间时都定义 PVC 配置。

  - 当卷声明模板用作卷源时，支持使用变量替换来查找 PVC 名称。

- 支持改进的审核：

  - **PipelineRun.Status** 字段现在包含管道中运行的每个任务的状态，以及用于实例化用于监控管道运行进度的管道规格。

  - Pipeline 结果已添加到 pipeline 规格和 **PipelineRun** 状态中。

  - **TaskRun.Status** 字段现在包含用于实例化 **TaskRun** 资源的具体任务规格。

- 支持在条件中应用默认参数。

- 现在，通过引用集群任务创建的任务运行会添加 **tekton.dev/clusterTask** 标签，而不是 **tekton.dev/task** 标签。

- kube config writer 现在在资源结构中添加了 **ClientKeyData** 和 **ClientCertificateData** 配置，以便使用 kubeconfig-creator 任务替换 pipeline 资源类型集群。

- 现在，**feature-flags** 和 **config-defaults** 配置映射的名称可以自定义。

- 现在，在任务运行使用的 pod 模板中支持主机网络。

- 现在，可以使用 Affinity Assistant 支持任务运行中共享工作空间卷的节点关联性。默认情况下，这在 OpenShift Pipelines 上被禁用。

- Pod 模板已更新，使用 **imagePullSecrets** 指定在启动一个 pod 时，容器运行时用来拉取容器镜像的 secret。

- 如果控制器无法更新任务运行，则支持从任务运行控制器发出警告事件。

- 在所有资源中添加了标准或者推荐的 k8s 标签，以标识属于应用程序或组件的资源。

- 现在，**Entrypoint** 进程被通知有信号，然后这些信号会使用一个 **Entrypoint** 进程的专用 PID 组来传播这些信号。

- pod 模板现在可以在运行时使用任务运行 specs 在任务级别设置。

- 支持放出 Kubernetes 事件：

  - 控制器现在会为其他任务运行生命周期事件发出事件 - **taskrun started** 和 **taskrun running**。

  - 频道运行控制器现在会在管道每次启动时放出一个事件。

- 除了默认的 Kubernetes 事件外，现在还提供对任务运行的支持。可将控制器配置为发送任何任务运行事件（如创建、启动和失败）作为云事件。

- 支持使用 **$context.<task|taskRun|pipelineRun>.name** 变量来引用管道运行和任务运行时的适当名称。
- 现在提供了管道运行参数的验证，以确保管道运行提供了管道所需的所有参数。这也允许管道运行在所需参数之外提供额外的参数。
- 现在，您可以使用管道 YAML 文件中的 **finally** 字段指定管道中的任务，这些任务会在管道退出前始终执行。
- **git-clone** 集群任务现在可用。

3.1.5.1.2. Pipelines CLI

- **tkn dlistener describe** 命令现在可以支持内嵌触发器绑定。
- 支持在使用不正确的子命令时推荐子命令并给出建议。
- 现在，如果管道中只有一个任务存在，**tkn task describe** 命令会自动选择该任务。
- 现在您可以使用默认参数值启动任务，方法是在 **tkn task start** 命令中指定 **--use-param-defaults** 标记。
- 现在，您可以使用 **tkn pipeline start** 或 **tkn task start** 命令的 **--workspace** 选项为管道运行或任务指定卷声明模板。
- **tkn pipelinerun logs** 命令现在会显示 **finally** 部分中列出的最终任务的日志。
- 现在，为 **tkn task start** 命令提供了互动模式支持，并为以下 **tkn** 资源提供 **describe** 子命令：**pipeline**, **PipelineRun**、**task**、**taskrun**、**clustertask** 和 **pipelineresource**。
- **tkn version** 命令现在显示集群中安装的触发器版本。
- **tkn pipeline describe** 命令现在显示为频道中使用的任务指定的参数值和超时。
- 添加了对 **tkn pipelinerun describe** 和 **tkn taskrun describe** 命令的 **--last** 选项的支持，以分别描述最新的频道运行或任务运行。
- **tkn pipeline describe** 命令现在显示管道中适用于任务的条件。
- 现在，您可以在 **tkn resource list** 命令中使用 **--no-headers** 和 **--all-namespaces** 标记。

3.1.5.1.3. 触发器

- 现在以下通用表达式语言（CEL）功能可用：
  - **parseURL** 用来解析和提取一个 URL 的部分内容
  - **parseJSON** 用来解析嵌入在 **deployment** webhook 中的 **payload** 字段中的字符串中的 JSON 值类型
- 添加了来自 Bitbucket 的 webhook 的新拦截器。
- 现在，在使用 **kubectl get** 列出时，事件监听器会显示 **Address URL** 和 **Available status** 作为额外的项。
- 触发器模板参数现在使用 **$(tt.params.<paramName>)** 语法而不是 **$(params.<paramName>)** 来减少触发器模板和资源模板参数之间的混淆。

- 现在，您可以在 **EventListener** CRD 中添加 **容限**，以确保事件监听程序使用相同的配置，即使所有节点都因为安全或管理问题而产生污点也是如此。
- 现在，您可以在 **URL/live** 中为事件监听器添加就绪探测（Readiness Probe）。
- 现在，添加了对在事件监听器触发器中嵌入 **TriggerBinding** 规格的支持。
- 触发器资源现在附带推荐的 **app.kubernetes.io** 标签注解。

3.1.5.2. 已弃用的功能

本发行版本中已弃用了以下内容：

- 所有集群范围命令（包括 **clustertask** 和 **clustertriggerbinding** 命令）的 **--namespace** 或 **-n** 标志都已弃用。它将在以后的发行版本中被删除。
- 事件监听器中的 **triggers.bindings** 中的 **name** 字段已弃用。现在使用 **ref** 字段替代，并将在以后的发行版本中删除。
- 使用 **$(params)** 的触发器模板中的变量插值已经被弃用，现在使用 **$(tt.params)** 来减少与管道变量插入语法的混乱。在以后的发行版本中会删除 **$(params.<paramName>)** 语法。
- 在集群任务中弃用了 **tekton.dev/task** 标签。
- **TaskRun.Status.ResourceResults.ResourceRef** 字段已弃用，并将被删除。
- **tkn pipeline create**、**tkn task create** 和 **tkn resource create -f** 子命令已被删除。
- 从 **tkn** 命令中删除了命名空间验证。
- **tkn ct start** 命令中的默认超时时间（**1h**）以及 **-t** 标志已被删除。
- **s2i** 集群任务已弃用。

3.1.5.3. 已知问题

- 条件（Conditions）不支持工作区。
- **tkn clustertask start** 命令不支持 **--workspace** 选项和互动模式。
- 支持 **$(params.<paramName>)** 语法的向后兼容性会强制您使用带有特定管道参数的触发器模板，因为触发器 s Webhook 无法将触发器参数与管道参数区分开。
- 当针对 **tekton_taskrun_count** 和 **tekton_taskrun_duration_seconds_count** 运行一个 promQL 查询时，Pipeline metrics 会报告不正确的值。
- 当为一个工作区指定了一个不存在的 PVC 名称时，管道运行和任务运行会维持在 **Running** 和 **Running(Pending)** 的状态。

3.1.5.4. 修复的问题

- 在以前的版本中，如果任务和集群任务的名称是相同的，则 **tkn task delete <name> --trs** 命令会同时删除 Task 和 ClusterTask。在这个版本中，该命令只删除任务 **<name>** 创建的任务运行。

- 以前，**tkn pr delete -p <name> --keep 2** 命令会在使用 **--keep** 是忽略 **-p** 标志，并将删除除最后两个以外的所有管道运行。在这个版本中，命令只删除由管道 **<name>** 创建的管道运行，但最后两个除外。
- **tkn triggertemplate describe** 输出现在以表格式而不是 YAML 格式显示资源模板。
- 在以前的版本中，当一个新用户添加到容器时，**buildah** 集群任务会失败。在这个版本中，这个问题已被解决。

3.1.6. Red Hat OpenShift Pipelines 技术预览 1.0 发行注记

3.1.6.1. 新功能

Red Hat OpenShift Pipelines 技术预览（TP）1.0 现在包括在 OpenShift Container Platform 4.4 中。Red Hat OpenShift Pipelines TP 1.0 更新为支持：

- Tekton Pipelines 0.11.3
- Tekton **tkn** CLI 0.9.0
- Tekton Triggers 0.4.0
- 基于 Tekton Catalog 0.11 的集群任务

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift Pipelines 1.0 中的新内容。

3.1.6.1.1. Pipelines

- 支持 v1beta1 API 版本。
- 支持改进的限制范围。在以前的版本中，限制范围仅为任务运行和管道运行指定。现在不需要显式指定限制范围。使用命名空间中的最小限制范围。
- 支持使用任务结果和任务参数在任务间共享数据。
- 现在，管道可以被配置为不覆盖 **HOME** 环境变量和步骤的工作目录。
- 与任务步骤类似，**sidecar** 现在支持脚本模式。
- 现在，您可以在任务运行 **podTemplate** 资源中指定不同的调度程序名称。
- 支持使用 Star Array Notation 替换变量。
- Tekton 控制器现在可以配置为监控单个命名空间。
- 现在，在管道、任务、集群任务、资源和条件规格中添加了一个新的 description 字段。
- 在 Git pipeline 资源中添加代理参数。

3.1.6.1.2. Pipelines CLI

- 现在为以下 **tkn** 资源添加了 **describe** 子命令： **EventListener**、**Condition**、**triggerTemplate**、**ClusterTask** 和 **TriggerSBinding**。

- 在以下资源中添加 **v1beta1** 支持以及 **v1alpha1** 的向后兼容性： **ClusterTask**、**Task**、**Pipeline**、**PipelineRun** 和 **TaskRun**。
- 以下命令现在可以使用 **--all-namespaces** 标志选项列出所有命名空间的输出结果： **tkn task list**、**tkn pipeline list**、**tkn taskrun list** 和 **tkn pipelinerun list**
  这些命令的输出也可以通过 **--no-headers** 选项在没有标头的情况下显示信息。
- 现在您可以使用默认参数值启动管道，方法是在 **tkn pipelines start** 命令中指定 **--use-param-defaults** 标记。
- 现在，在 **tkn pipeline start** 和 **tkn task start** 命令中增加了对工作区的支持。
- 现在增加了一个新命令 **clustertriggerbinding**，它带有以下子命令：**describe**、**delete** 和 **list**。
- 现在，您可以使用本地或远程 **yaml** 文件直接启动管道运行。
- **describe** 子命令现在显示一个改进的详细输出。现在，除了新的项，如 **description**、**timeout**、**param description** 和 **sidecar status**，命令输出还提供了关于一个特定 **tkn** 资源的更详细的信息。
- 现在，如果命名空间中只有一个任务，**tkn task log** 命令会直接显示日志。

3.1.6.1.3. 触发器

- 现在触发器可以同时创建 **v1alpha1** 和 **v1beta1** 管道资源。
- 支持新的通用表达式语言(CEL)拦截器功能 - **compareSecret**。此功能安全地将字符串与 CEL 表达式中的 secret 进行比较。
- 支持在事件监听器触发器级别进行身份验证和授权。

3.1.6.2. 已弃用的功能

本发行版本中已弃用了以下内容：

- **Steps** 规格中的环境变量 **$HOME**，变量 **workingDir** 已被弃用，并可能在以后的发行版本中有所变化。目前，在 **Step** 容器中，**HOME** 和 **workingDir** 变量会分别被 **/tekton/home** 和 **/workspace** 变量覆盖。
  在以后的发行版本中，这两个字段将不会被修改，它将被设置为容器镜像和 **Task** YAML 中定义的值。在本发行版本中，使用 **disable-home-env-overwrite** 和 **disable-working-directory-overwrite** 标记来禁用覆盖 **HOME** 和 **workingDir** 变量。
- 以下命令已弃用，并可能在以后的发行版本中删除： **tkn pipeline create**、**tkn task create**。
- 在 **tkn resource create** 命令中使用 **-f** 标志现已弃用。以后的发行版本中可能会删除它。
- **tkn clustertask create** 命令中的 **-t** 标记和 **--timeout** 标记（使用秒格式）现已被弃用。现在只支持持续超时格式，例如 **1h30s**。这些已弃用的标记可能会在以后的版本中删除。

3.1.6.3. 已知问题

- 如果您要从 Red Hat OpenShift Pipelines 的旧版本升级，则必须删除您现有的部署，然后再升级到 Red Hat OpenShift Pipelines 版本 1.0。要删除现有的部署，您必须首先删除自定义资源，然后卸载 Red Hat OpenShift Pipelines Operator。如需了解更多详细信息，请参阅卸载 Red Hat OpenShift Pipelines 部分。

- 提交相同的 **v1alpha1** 任务多次会导致错误。在重新提交一个 **v1alpha1** 任务时，使用 **oc replace** 命令而不是 **oc apply**。

- 当一个新用户添加到容器时，**buildah** 集群任务无法正常工作。
当安装 Operator 时，**buildah** 集群任务的 **--storage-driver** 标志没有指定，因此它会被设置为默认值。在某些情况下，这会导致存储驱动程序设置不正确。当添加一个新用户时，错误的 storage-driver 会造成 **buildah** 集群任务失败并带有以下错误：

```
useradd: /etc/passwd.8: lock file already used
useradd: cannot lock /etc/passwd; try again later.
```

作为临时解决方案，在 **buildah-task.yaml** 文件中手工把 **--storage-driver** 标识的值设置为 **overlay**：

1. 以 **cluster-admin** 身份登录到集群:

```
$ oc login -u <login> -p <password> https://openshift.example.com:6443
```

2. 使用 **oc edit** 命令编辑 **buildah** 集群任务：

```
$ oc edit clustertask buildah
```

buildah clustertask YAML 文件的最新版本会在由 **EDITOR** 环境变量指定的编辑器中打开。

3. 在 **Steps** 字段中找到以下 **command** 字段：

```
 command: ['buildah', 'bud', '--format=$(params.FORMAT)', '--tls-
verify=$(params.TLSVERIFY)', '--layers', '-f', '$(params.DOCKERFILE)', '-t',
'$(resources.outputs.image.url)', '$(params.CONTEXT)']
```

4. 使用以下内容替换 **command** 字段：

```
 command: ['buildah', '--storage-driver=overlay', 'bud', '--format=$(params.FORMAT)', '--
tls-verify=$(params.TLSVERIFY)', '--no-cache', '-f', '$(params.DOCKERFILE)', '-t',
'$(params.IMAGE)', '$(params.CONTEXT)']
```

5. 保存文件并退出。

另外，您还可以直接在 web 控制台中直接修改 **buildah** 集群任务 YAML 文件：进入 **Pipelines → Cluster Tasks → buildah**。从 **Actions** 菜单中选择 **Edit Cluster Task**，如前所示替换 **command** 项。

3.1.6.4. 修复的问题

- 在以前的版本中，即使镜像构建已在进行中，**DeploymentConfig** 任务也会触发新的部署构建。这会导致管道部署失败。在这个版本中，**deploy task** 命令被 **oc rollout status** 命令替代，它会等待正在进行中的部署完成。

- 现在在管道模板中添加了对 **APP_NAME** 参数的支持。

- 在以前的版本中，Java S2I 的管道模板无法在 registry 中查找镜像。在这个版本中，使用现有镜像管道资源而不是用户提供的 **IMAGE_NAME** 参数来查找镜像。

- 所有 OpenShift Pipelines 镜像现在都基于 Red Hat Universal Base Images（UBI）。

- 在以前的版本中，当管道在 **tekton-pipelines** 以外的命名空间中安装时，**tkn version** 命令会将管道版本显示为 **unknown**。在这个版本中，**tkn version** 命令会在任意命名空间中显示正确的管道版本。
- **tkn version** 命令不再支持 **-c** 标志。
- 非管理员用户现在可以列出集群触发器绑定。
- 现在为 CEL 拦截器修复了事件监听程序 **CompareSecret** 功能。
- 现在，当任务和集群任务的名称相同时，任务和集群任务的 **list**、**describe** 和 **start** 子命令可以正确地显示输出。
- 在以前的版本中，OpenShift Pipelines Operator 修改了特权安全性上下文约束 (SCC)，这会在集群升级过程中造成错误。这个错误现已解决。
- 在 **tekton-pipelines** 命名空间中，现在将所有任务运行和管道运行的超时设置为使用配置映射的 **default-timeout-minutes** 字段。
- 在以前的版本中，Web 控制台中的管道部分没有为非管理员用户显示。这个问题现已解决。

3.2. 了解 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。

3.2.1. 主要特性

- Red Hat OpenShift Pipelines 是一个无服务器的 CI/CD 系统，它在独立的容器中运行 Pipelines，以及所有需要的依赖组件。
- Red Hat OpenShift Pipelines 是为开发基于微服务架构的非中心化团队设计的。
- Red Hat OpenShift Pipelines 使用标准 CI/CD 管道（pipeline）定义，这些定义可轻松扩展并与现有 Kubernetes 工具集成，可让您按需扩展。
- 您可以通过 Red Hat OpenShift Pipelines 使用 Kubernetes 工具（如 Source-to-Image (S2I)、Buildah、Buildpacks 和 Kaniko）构建镜像，这些工具可移植到任何 Kubernetes 平台。
- 您可以使用 OpenShift Container Platform 开发控制台来创建 Tekton 资源，查看管道运行的日志，并管理 OpenShift Container Platform 命名空间中的管道。

3.2.2. OpenShift Pipeline 概念

本指南提供了对管道（pipeline）概念的详细论述。

3.2.2.1. 任务（Task）

*Task（任务）*是 Pipeline 的构建块，它由按顺序执行的步骤组成。它基本上是一个输入和输出的功能。一个任务可以单独运行，也可以作为管道的一部分运行。任务（Task）可以重复使用，并可用于多个 Pipelines。

*Step（步骤）*是由任务顺序执行并实现特定目标（如构建镜像）的一系列命令。每个任务都作为 pod 运行，每个步骤都作为该 pod 中的容器运行。由于步骤在同一个 pod 中运行，所以它们可以访问同一卷来缓存文件、配置映射和 secret。

以下示例显示了 **apply-manifests** 任务。

```
apiVersion: tekton.dev/v1beta1 1
kind: Task 2
metadata:
  name: apply-manifests 3
spec: 4
  workspaces:
  - name: source
  params:
    - name: manifest_dir
      description: The directory in source that contains yaml manifests
      type: string
      default: "k8s"
  steps:
    - name: apply
      image: image-registry.openshift-image-registry.svc:5000/openshift/cli:latest
      workingDir: /workspace/source
      command: ["/bin/bash", "-c"]
      args:
        - |-
          echo Applying manifests in $(params.manifest_dir) directory
          oc apply -f $(params.manifest_dir)
          echo -----------------------------------
```

1. Task API 版本 **v1beta1**。
2. Kubernetes 对象的类型，**任务**。
3. 此任务的唯一名称。
4. 列出任务中的参数和步骤，以及任务使用的工作区（workspace）。

此任务启动 pod，并在该 pod 中使用指定镜像运行一个容器，以运行指定的命令。

3.2.2.2. TaskRun

TaskRun 使用集群上的特定输入、输出和执行参数来实例化一个任务用来执行它。它可自行调用，或作为管道中每个任务的 PipelineRun 的一部分。

任务由执行容器镜像的一个或多个步骤组成，每个容器镜像执行特定的构建工作。TaskRun 以指定顺序在任务中执行步骤，直到所有步骤都成功执行或发生失败为止。PipelineRun 由 Pipeline 中每个任务的 PipelineRun 自动创建。

以下示例显示了一个带有相关输入参数运行 **apply-manifests** 任务的 TaskRun:

```
apiVersion: tekton.dev/v1beta1 1
kind: TaskRun 2
metadata:
  name: apply-manifests-taskrun 3
```

```
spec: 4
  serviceAccountName: pipeline
  taskRef: 5
    kind: Task
    name: apply-manifests
  workspaces: 6
  - name: source
    persistentVolumeClaim:
      claimName: source-pvc
```

1. TaskRun API 版本 **v1beta1**。
2. 指定 Kubernetes 对象的类型。在本例中，**TaskRun**。
3. 用于标识此 TaskRun 的唯一名称。
4. TaskRun 的定义。对于这个 TaskRun，指定了任务和所需的工作区。
5. 用于此 TaskRun 的任务引用的名称。此 TaskRun 会执行 **apply-manifests** 任务。
6. TaskRun 使用的工作空间。

3.2.2.3. Pipelines

Pipeline 一组 **Task（任务）**资源，它们按特定顺序执行。执行它们是为了构建复杂的工作流，以自动化应用程序的构建、部署和交付。您可以使用包含一个或多个任务的管道为应用程序定义 CI/CD 工作流。

Pipeline 资源的定义由多个字段或属性组成，它们一起可让管道实现一个特定目标。每个 **Pipeline** 资源定义必须至少包含一个**Task（任务）**资源，用于控制特定输入并生成特定的输出。Pipeline 定义也可以根据应用程序要求包括 *Conditions*、*Workspaces*、*Parameters* 或 *Resources*。

以下示例显示了 **build-and-deploy** pipeline，它使用 **buildah ClusterTask** 资源从 Git 存储库构建应用程序镜像：

```
apiVersion: tekton.dev/v1beta1 1
kind: Pipeline 2
metadata:
  name: build-and-deploy 3
spec: 4
  workspaces: 5
  - name: shared-workspace
  params: 6
  - name: deployment-name
    type: string
    description: name of the deployment to be patched
  - name: git-url
    type: string
    description: url of the git repo for the code of deployment
  - name: git-revision
    type: string
    description: revision to be used from repo of the code for deployment
    default: "pipelines-1.4"
  - name: IMAGE
    type: string
```

```
    description: image to be built from the code
  tasks: (7)
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.git-url)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: $(params.git-revision)
  - name: build-image (8)
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: TLSVERIFY
      value: "false"
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - fetch-repository
  - name: apply-manifests (9)
    taskRef:
      name: apply-manifests
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter: (10)
    - build-image
  - name: update-deployment
    taskRef:
      name: update-deployment
    workspaces:
    - name: source
      workspace: shared-workspace
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

(1) Pipeline API 版本 **v1beta1**。

2 指定 Kubernetes 对象的类型。在本例中, **Pipeline**。

3 此 Pipeline 的唯一名称。

4 指定 Pipeline 的定义和结构。

5 Pipeline 中所有任务使用的工作区。

6 Pipeline 中所有任务使用的参数。

7 指定 Pipeline 中使用的任务列表。

8 任务 **build-image** 使用 **buildah** ClusterTask 从给定的 Git 仓库构建应用程序镜像。

9 任务 **apply-manifests** 使用相同名称的用户定义的任务。

10 指定在 Pipeline 中运行任务的顺序。在本例中，**apply-manifests** 任务仅在 **build-image** 任务完成后运行。

3.2.2.4. PipelineRun

PipelineRun 是一个 Pipeline 的运行实例。它使用集群上的特定输入、输出和执行参数来实例化 Pipeline 执行。在 PipelineRun 中为每个任务自动创建一个对应的 TaskRun。

Pipeline 中的所有任务均按定义的顺序执行，直到所有任务成功或任务失败为止。**status** 字段跟踪并存储 PipelineRun 中的每个 TaskRun 的进度，用于监控和审核目的。

以下示例显示了一个 PipelineRun，用于运行带有相关资源和参数的 **build-and-deploy** Pipeline:

```
apiVersion: tekton.dev/v1beta1 1
kind: PipelineRun 2
metadata:
  name: build-deploy-api-pipelinerun 3
spec:
  pipelineRef:
    name: build-and-deploy 4
  params: 5
  - name: deployment-name
    value: vote-api
  - name: git-url
    value: https://github.com/openshift-pipelines/vote-api.git
  - name: IMAGE
    value: image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/vote-api
  workspaces: 6
  - name: shared-workspace
    volumeClaimTemplate:
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 500Mi
```

1 PipelineRun API 版本 **v1beta1**。

2 指定 Kubernetes 对象的类型。在本例中，**PipelineRun**。

3 用于标识此 PipelineRun 的唯一名称。

4 要运行的 Pipeline 的名称。在本例中，**build-and-deploy**。

5 指定运行 Pipeline 所需的参数列表。

6 PipelineRun 使用的 Workspace。

3.2.2.5. Workspaces（工作区）

注意

建议您在 OpenShift Pipelines 中使用 Workspaces 而不是 PipelineResources，因为 PipelineResources 很难调试，范围有限，且不容易重复使用。

Workspace 声明 Pipeline 中的任务在运行时需要的共享存储卷来接收输入或提供输出。Workspaces 不指定卷的实际位置，它允许您定义运行时所需的文件系统或部分文件系统。Task 或 Pipeline 会声明 Workspace，您必须提供卷的特定位置详情。然后，它会挂载到 TaskRun 或 PipelineRun 中的 Workspace 中。这种将卷声明与运行时存储卷分开来使得任务可以被重复使用、灵活且独立于用户环境。

使用 Workspaces，您可以：

- 存储任务输入和输出
- 任务间共享数据
- 使用它作为 Secret 中持有的凭证的挂载点
- 使用它作为 ConfigMap 中保存的配置的挂载点
- 使用它作为机构共享的通用工具的挂载点
- 创建可加快作业的构建工件缓存

您可以使用以下方法在 TaskRun 或 PipelineRun 中指定 Workspaces:

- 只读 ConfigMap 或 Secret
- 与其他任务共享的现有 PersistentVolumeClaim
- 来自提供的 VolumeClaimTemplate 的 PersistentVolumeClaim
- TaskRun 完成后丢弃的 emptyDir

以下显示了 **build-and-deploy** Pipeline 的代码片段，它为任务 **build-image** 和 **apply-manifests** 声明了一个 **shared-workspace** Workspace。

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces: 1
```

```
  - name: shared-workspace
  params:
...
  tasks: (2)
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: TLSVERIFY
      value: "false"
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces: (3)
    - name: source (4)
      workspace: shared-workspace (5)
    runAfter:
    - fetch-repository
  - name: apply-manifests
    taskRef:
      name: apply-manifests
    workspaces: (6)
    - name: source
      workspace: shared-workspace
    runAfter:
      - build-image
...
```

1. Pipeline 中定义的任务共享的 Workspace 列表。Pipeline 可以根据需要定义 Workspace。在这个示例中，只声明了一个名为 **shared-workspace** 的 Workspace。
2. Pipeline 中使用的任务定义。此片段定义了两个任务，**build-image** 和 **apply-manifests**。这两个任务共享一个 Workspace。
3. **build-image** 任务中使用的 Workspaces 列表。任务定义可以根据需要包含多个 Workspace。但建议任务最多使用一个可写 Workspace。
4. 唯一标识任务中使用的 Workspace 的名称。此任务使用一个名为 **source** 的 Workspace。
5. 任务使用的 Pipeline Workspace 的名称。请注意，Workspace **source** 使用 Pipeline Workspace **shared-workspace**。
6. **apply-manifests** 任务中使用的 Workspace 列表。请注意，此任务与 **build-image** 任务共享 **source** Workspace。

工作区可帮助任务共享数据，并允许您指定 Pipeline 中每个任务在执行过程中所需的一个或多个卷。您可以创建持久性卷声明，或者提供一个卷声明模板，用于为您创建持久性卷声明。

以下 **build-deploy-api-pipelinerun** PipelineRun 的代码片段使用卷声明模板创建持久性卷声明来为 **build-and-deploy** Pipeline 中使用的 **shared-workspace** Workspace 定义存储卷。

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: build-deploy-api-pipelinerun
```

```
spec:
  pipelineRef:
    name: build-and-deploy
  params:
...

  workspaces: 1
  - name: shared-workspace 2
    volumeClaimTemplate: 3
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 500Mi
```

1 指定 Pipeline Workspaces 列表，用于在 PipelineRun 中提供卷绑定。

2 提供卷的 Pipeline 中的 Workspace 的名称。

3 指定卷声明模板，该模板可创建一个持久性卷声明来为工作区定义存储卷。

3.2.2.6. 触发器

使用*触发器（Trigger）*和 Pipelines 一起创建一个完整的 CI/CD 系统，其中 Kubernetes 资源定义整个 CI/CD 执行。触发器捕获外部事件，如 Git 拉取请求，并处理它们以获取关键信息。将这个事件数据映射到一组预定义的参数会触发一系列任务，然后创建和部署 Kubernetes 资源并实例化管道。

例如，您可以使用 Red Hat OpenShift Pipelines 为应用程序定义 CI/CD 工作流。管道必须启动，才能在应用程序存储库中使任何新的更改生效。通过捕获和处理任何更改事件，并通过触发器部署新镜像的管道运行来自动触发这个过程。

触发器由以下主要资源组成，它们可一起组成可重复使用、分离和自力更生的 CI/CD 系统：

- **TriggerBinding** 资源验证事件，从事件有效负载中提取字段，并将它们保存为参数。
  以下示例显示了 **TriggerBinding** 资源的代码片段,它从接收的事件有效负载中提取 Git 存储库信息：

```
apiVersion: triggers.tekton.dev/v1alpha1 1
kind: TriggerBinding 2
metadata:
  name: vote-app 3
spec:
  params: 4
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

1 **TriggerBinding** 资源的 API 版本。在本例中，**v1alpha1**。

2 指定 Kubernetes 对象的类型。在本例中，**TriggerBinding**。

3 用于标识 **TriggerBinding** 资源的唯一名称。

4 从接收的事件有效负载中提取并传递给 **TriggerTemplate** 的参数列表。在本例中，Git 仓库 URL、名称和修订版本是从事件有效负载主体中提取的。

- **TriggerTemplate** 资源充当创建资源的方式标准。它指定了 **TriggerBinding** 资源中参数化数据的方式。触发器模板从触发器绑定接收输入，然后执行一系列操作来创建新管道资源，并启动新管道运行。
  以下示例显示了 **TriggerTemplate** 资源的代码片段，它使用您刚创建的 **TriggerBinding** 资源提供的 Git 存储库信息创建一个管道运行：

```
apiVersion: triggers.tekton.dev/v1alpha1 1
kind: TriggerTemplate 2
metadata:
  name: vote-app 3
spec:
  params: 4
  - name: git-repo-url
    description: The git repository url
  - name: git-revision
    description: The git revision
    default: pipelines-1.4
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates: 5
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
      name: build-deploy-$(tt.params.git-repo-name)-$(uid)
    spec:
      serviceAccountName: pipeline
      pipelineRef:
        name: build-and-deploy
      params:
      - name: deployment-name
        value: $(tt.params.git-repo-name)
      - name: git-url
        value: $(tt.params.git-repo-url)
      - name: git-revision
        value: $(tt.params.git-revision)
      - name: IMAGE
        value: image-registry.openshift-image-registry.svc:5000/pipelines-
tutorial/$(tt.params.git-repo-name)
      workspaces:
      - name: shared-workspace
        volumeClaimTemplate:
          spec:
            accessModes:
             - ReadWriteOnce
            resources:
              requests:
                storage: 500Mi
```

1. **TriggerTemplate** 资源的 API 版本。在本例中， **v1alpha1**。
2. 指定 Kubernetes 对象的类型。在本例中，**TriggerTemplate**。
3. 用于标识 **TriggerTemplate** 资源的唯一名称。
4. **TriggerBinding** 或 **EventListerner** 资源提供的参数。
5. 指定使用 **TriggerBinding** 或 **EventListener** 资源接收的参数创建资源方法的模板列表。

- **Trigger** 资源连接 **TriggerBinding** 和 **TriggerTemplate** 资源，这个 **Trigger** 资源在 **EventListener** 规格中被引用。
以下示例显示了一个 **Trigger** 资源的代码片段，名为 **vote-trigger**，它连接 **TriggerBinding** 和 **TriggerTemplate** 资源。

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: Trigger (2)
metadata:
  name: vote-trigger (3)
spec:
  serviceAccountName: pipeline (4)
  bindings:
    - ref: vote-app (5)
  template: (6)
    ref: vote-app
```

1. **Trigger** 资源的 API 版本。在本例中， **v1alpha1**。
2. 指定 Kubernetes 对象的类型。在本例中, **Trigger**。
3. 用于标识 **Trigger** 资源的唯一名称。
4. 要使用的服务帐户名称。
5. 连接到 **TriggerTemplate** 资源的 **TriggerBinding** 资源的名称。
6. 连接到 **TriggerBinding** 资源的 **TriggerTemplate** 资源的名称。

- **EventListener** 资源提供一个端点或事件接收器（sink），用于使用 JSON 有效负载侦听传入的基于 HTTP 的事件。它从每个 **TriggerBinding** 资源提取事件参数，然后处理此数据以按照对应的 **TriggerTemplate** 资源指定的 Kubernetes 资源创建 Kubernetes 资源。 **EventListener** 资源还使用事件 **interceptors（拦截器）** 在有效负载上执行轻量级事件处理或基本过滤，这可识别有效负载类型并进行自选修改。目前，管道触发器支持四种拦截器： *Webhook Interceptors*、*GitHub Interceptors*、*GitLab Interceptors* 和 *Common Expression Language(CEL)Interceptors*。
以下示例显示了一个 **EventListener** 资源，它引用名为 **vote-trigger** 的 **Trigger** 资源。

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: EventListener (2)
metadata:
  name: vote-app (3)
spec:
```

```
serviceAccountName: pipeline 4
triggers:
  - triggerRef: vote-trigger 5
```

1. **EventListener** 资源的 API 版本。在本例中，**v1alpha1**。
2. 指定 Kubernetes 对象的类型。在本例中，**EventListener**。
3. 用于标识 **EventListener** 资源的唯一名称。
4. 要使用的服务帐户名称。
5. **EventListener** 资源引用的 **Trigger** 资源的名称。

Red Hat OpenShift Pipelines 中的触发器支持 HTTP（不安全）和 HTTPS（安全 HTTP）连接到 **Eventlistener** 资源。使用安全 HTTPS 连接，您可以在集群内部和外部获得端到端安全连接。创建命名空间后，您可以通过将 **operator.tekton.dev/enable-annotation=enabled** 标签添加到命名空间，然后创建一个 **Trigger** 资源以及使用重新加密 TLS 终止的安全路由，从而为 **Eventlistener** 资源启用这个安全 HTTPS 连接。

3.2.3. 其他资源

- 有关安装管道的详情，请参阅安装 OpenShift Pipelines。
- 有关创建自定义 CI/CD 解决方案的详情，请参阅使用 CI/CD Pipelines 创建应用程序。
- 有关重新加密 TLS 终止的详情，请参阅重新加密终止。
- 有关安全路由的详情，请参阅安全路由部分。

3.3. 安装 OPENSHIFT PIPELINES

本指南帮助集群管理员了解将 Red Hat OpenShift Pipelines Operator 安装到 OpenShift Container Platform 集群的整个过程。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已安装了 **oc** CLI。
- 您已在本地系统中安装了 OpenShift Pipelines (tkn)CLI。

3.3.1. 在 Web 控制台中安装 Red Hat OpenShift Pipelines Operator

您可以使用 OpenShift Container Platform OperatorHub 中列出的 Operator 来安装 Red Hat OpenShift Pipelines。安装 Red Hat OpenShift Pipelines Operator 时，管道配置所需的自定义资源（CR）与 Operator 一起自动安装。

默认 Operator 自定义资源定义（CRD）**config.operator.tekton.dev** 现在被 **tektonconfigs.operator.tekton.dev** 替代。另外，Operator 提供以下额外的 CRD 来单独管理 OpenShift Pipelines 组
件：**tektonpipelines.operator.tekton.dev**、**tektontriggers.operator.tekton.dev** 和 **tektonaddons.operator.tekton.dev**。

如果在集群中安装了 OpenShift Pipelines，现有安装会无缝升级。Operator 会根据需要将集群中的 **config.operator.tekton.dev** 实例替换为 **tektonconfigs.operator.tekton.dev** 实例，以及其它 CRD 的额外对象。

警告

如果您手动更改现有安装，例如，在 **config.operator.tekton.dev** CRD 实例中修改了目标命名空间（更改了 **resource name - cluster** 的项），则升级过程将不会非常流畅。在这种情况下，推荐的工作流是，先卸载安装，然后再重新安装 Red Hat OpenShift Pipelines Operator。

Red Hat OpenShift Pipelines Operator 现在提供了选择您要安装的组件的选项，方法是作为 **TektonConfig** CR 的一部分来指定配置集。在安装 Operator 时会自动安装 **TektonConfig** CR。支持的配置集有：

- Base：只安装 Tekton 管道。
- Default：安装 Tekton 管道和 Tekton 触发器。
- All：在安装了 **TektonConfig** CR 时，使用的默认配置集。此配置集安装所有 Tekton 组件：Tekton Pipelines、Tekton Triggers、Tekton Addons（包括 **ClusterTasks**、**ClusterTriggerBindings**、**ConsoleCLIDownload**、**ConsoleQuickStart** 和 **ConsoleYAMLSample** 资源）。

流程

1. 在控制台的 **Administrator** 视角中，导航到 **Operators → OperatorHub**。
2. 使用 **Filter by keyword** 复选框在目录中搜索 **Red Hat OpenShift Pipelines** Operator。点 **Red Hat OpenShift Pipelines** Operator 标题。
3. 参阅 **Red Hat OpenShift Pipelines** Operator 页中有关 Operator 的简单描述。点击 **Install**。
4. 在 **Install Operator** 页面中：
   a. 为 **Installation Mode** 选择 **All namespaces on the cluster (default)**。选择该项会将 Operator 安装至默认 **openshift-operators** 命名空间，这将启用 Operator 以进行监视并在集群中的所有命名空间中可用。
   b. 为 **Approval Strategy** 选择 **Automatic**。这样可确保以后对 Operator 的升级由 Operator Lifecycle Manager (OLM) 自动进行。如果您选择 **Manual** 批准策略，OLM 会创建一个更新请求。作为集群管理员，您必须手动批准 OLM 更新请求，才可将 Operator 更新至新版本。
   c. 选择一个 **Update Channel**。
      - **stable** 频道启用 Red Hat OpenShift Pipelines Operator 最新稳定和支持版本的安装。
      - **preview** 频道启用 Red Hat OpenShift Pipelines Operator 的最新预览版本，该版本可能包含 **stable** 频道中尚不提供且不受支持的功能。
5. 点击 **Install**。您会看到 **Installed Operators** 页面中列出的 Operator。

注意

Operator 会自动安装到 **openshift-operators** 命名空间中。

6. 检查 **Status** 是否已被设置为 **Succeeded Up to date** 来确认 Red Hat OpenShift Pipelines Operator 已安装成功。

3.3.2. 使用 CLI 安装 OpenShift Pipelines Operator

您可以使用 CLI 从 OperatorHub 安装 Red Hat OpenShift Pipelines Operator。

流程

1. 创建一个订阅对象 YAML 文件，以便为 Red Hat OpenShift Pipelines Operator 订阅一个命名空间，如 **sub.yaml**：

   订阅示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-pipelines-operator
  namespace: openshift-operators
spec:
  channel:  <channel name> 1
  name: openshift-pipelines-operator-rh 2
  source: redhat-operators 3
  sourceNamespace: openshift-marketplace 4
```

   1. 指定您要订阅 Operator 的频道名称
   2. 要订阅的 Operator 的名称。
   3. 提供 Operator 的 CatalogSource 的名称。
   4. CatalogSource 的命名空间。将 **openshift-marketplace** 用于默认的 OperatorHub CatalogSource。

2. 创建订阅对象：

```
$ oc apply -f sub.yaml
```

   Red Hat OpenShift Pipelines Operator 现在安装在默认目标命名空间 **openshift-operators** 中。

3.3.3. 在受限环境中的 Red Hat OpenShift Pipelines Operator

Red Hat OpenShift Pipelines Operator 支持在受限网络环境中安装管道。

Operator 会安装代理 Webhook，它会根据 **cluster** 代理对象在由 tekton-controllers 创建的 pod 容器中设置代理环境变量。它还会在 **TektonPipelines**、**TektonTriggers**、**Controllers**、**Webhooks** 和 **Operator Proxy Webhook** 资源中设置代理环境变量。

默认情况下，**openshift-pipelines** 命名空间禁用代理 Webhook。要为任何其他命名空间禁用它，您可以在 **namespace** 对象中添加 **operator.tekton.dev/disable-proxy: true** 标签。

3.3.4. 其他资源

- 您可以参阅将 Operators 添加到集群一节中的内容来了解更多有关在 OpenShift Container Platform 上安装 Operator 的信息。
- 有关在受限环境中使用管道的更多信息，请参阅：
  - 镜像以在受限环境中运行管道
  - 为受限集群配置 Samples Operator
  - 创建带有镜像 registry 的集群

3.4. 卸载 OPENSHIFT PIPELINES

卸载 Red Hat OpenShift Pipelines Operator 分为两个步骤：

1. 删除安装 Red Hat OpenShift Pipelines Operator 时默认添加的自定义资源 (CR)。
2. 卸载 Red Hat OpenShift Pipelines Operator。

安装 Operator 时，仅卸载 Operator 不会删除默认创建的 Red Hat OpenShift Pipelines 组件。

3.4.1. 删除 Red Hat OpenShift Pipelines 组件和自定义资源

删除安装 Red Hat OpenShift Pipelines Operator 期间默认创建的自定义资源（CR）。

流程

1. 在 Web 控制台的 **Administrator** 视角中，导航至 **Administration → Custom Resource Definition**。
2. 在 **Filter by name** 框中键入 **config.operator.tekton.dev** 来搜索 Red Hat OpenShift Pipelines Operator CR。
3. 点击 **CRD Config** 查看 **Custom Resource Definition Details** 页面。
4. 点击 **Actions** 下拉菜单并选择 **Delete Custom Resource Definition**。

注意

删除 CR 将删除 Red Hat OpenShift Pipelines 组件，并丢失集群上的所有任务和管道。

5. 点击 **Delete** 以确认删除 CR。

3.4.2. 卸载 Red Hat OpenShift Pipelines Operator

流程

1. 在 **Operators → OperatorHub** 页面中，使用 **Filter by keyword** 复选框来搜索 **Red Hat OpenShift Pipelines Operator**。
2. 点 **OpenShift Pipelines Operator**。Operator 标题表示已安装该 Operator。
3. 在 **OpenShift Pipelines Operator** 描述符页面中，点击 **Uninstall**。

其他资源

- 您可以参阅从集群中卸载 Operators一节中的内容来了解更多有关从 OpenShift Container Platform 上卸载 Operator 的信息。

3.5. 为使用 OPENSHIFT PIPELINES 的应用程序创建 CI/CD 解决方案

使用 Red Hat OpenShift Pipelines，您可以创建一个自定义的 CI/CD 解决方案来构建、测试和部署应用程序。

要为应用程序创建一个完整的自助 CI/CD 管道，请执行以下任务：

- 创建自定义任务，或安装现有的可重复使用的任务。
- 为应用程序创建并定义交付管道。
- 使用以下方法之一提供附加到管道执行的工作区中的存储卷或文件系统：
  - 指定创建持久性卷声明的卷声明模板
  - 指定一个持久性卷声明
- 创建一个 **PipelineRun** 对象来实例化并调用管道。
- 添加触发器以捕获源仓库中的事件。

本节使用 **pipelines-tutorial** 示例来演示前面的任务。这个示例使用一个简单的应用程序，它由以下部分组成：

- 一个前端接口，**pipelines-vote-ui**，它的源代码在 **pipelines-vote-ui** Git 存储库中。
- 一个后端接口，**pipelines-vote-api**，它的源代码在 **pipelines-vote-api** Git 存储库中。
- **apply-manifests** 和 **update-deployment** 任务在 **pipelines-tutorial** Git 存储库中。

3.5.1. 先决条件

- 有访问 OpenShift Container Platform 集群的权限。
- 已使用在 OpenShift OperatorHub 中列出的 Red Hat OpenShift Pipelines Operator 安装了 OpenShift Pipelines。在安装后，它可用于整个集群。
- 已安装 OpenShift Pipelines CLI。
- 使用您的 GitHub ID fork 前端 **pipelines-vote-ui** 和后端 **pipelines-vote-api** Git 存储库，并具有对这些存储库的管理员访问权限。
- 可选：克隆了 **pipelines-tutorial** Git 存储库。

3.5.2. 创建项目并检查管道服务帐户

流程

1. 登录您的 OpenShift Container Platform 集群：

```
$ oc login -u <login> -p <password> https://openshift.example.com:6443
```

2. 为示例应用程序创建一个项目。在本例中，创建 **pipelines-tutorial** 项目：

```
$ oc new-project pipelines-tutorial
```

注意

如果您使用其他名称创建项目，请确定使用您的项目名称更新示例中使用的资源 URL。

3. 查看 **pipeline** 服务帐户：
Red Hat OpenShift Pipelines Operator 添加并配置一个名为 **pipeline** 的服务帐户，该帐户有足够的权限来构建和推送镜像。**PipelineRun** 对象使用此服务帐户。

```
$ oc get serviceaccount pipeline
```

3.5.3. 创建管道任务

流程

1. 从 **pipelines-tutorial** 存储库安装 **apply-manifests** 和 **update-deployment** 任务资源，其中包含可为管道重复使用的任务列表：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/01_apply_manifest_task.yaml
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/02_update_deployment_task.yaml
```

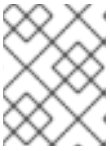
2. 使用 **tkn task list** 命令列出您创建的任务：

```
$ tkn task list
```

输出会确认创建了 **apply-manifests** 和 **update-deployment** 任务：

```
NAME                DESCRIPTION   AGE
apply-manifests                   1 minute ago
update-deployment                 48 seconds ago
```

3. 使用 **tkn clustertasks list** 命令列出由 Operator 安装的额外集群任务，如 **buildah** 和 **s2i-python**：



注意

要在受限环境中使用 **buildah** 集群任务，您必须确保 Dockerfile 使用内部镜像流作为基础镜像。

```
$ tkn clustertasks list
```

输出列出了 Operator 安装的 **ClusterTask** 资源：

```
NAME                DESCRIPTION   AGE
buildah                           1 day ago
git-clone                         1 day ago
s2i-python                        1 day ago
tkn                               1 day ago
```

3.5.4. 组装管道

管道（pipeline）代表一个 CI/CD 流，由要执行的任务定义。它被设计为在多个应用程序和环境中通用且可重复使用。

管道指定任务如何使用 **from** 和 **runAfter** 参数相互交互以及它们执行的顺序。它使用 **workspaces** 字段指定管道中每个任务在执行过程中所需的一个或多个卷。

在本小节中，您将创建一个管道，从 GitHub 获取应用程序的源代码，然后在 OpenShift Container Platform 上构建和部署应用程序。

管道为后端应用程序 **pipelines-vote-api** 和前端应用程序 **pipelines-vote-ui** 执行以下任务：

- 通过引用 **git-url** 和 **git-revision** 参数，从 Git 存储库中克隆应用程序的源代码。
- 使用 **buildah** 集群任务构建容器镜像。
- 通过引用 **image** 参数将镜像推送到内部 镜像 registry。
- 通过使用 **apply-manifests** 和 **update-deployment** 任务在 OpenShift Container Platform 上部署新镜像。

流程

1. 复制以下管道 YAML 文件示例内容并保存：

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces:
  - name: shared-workspace
  params:
  - name: deployment-name
    type: string
    description: name of the deployment to be patched
  - name: git-url
    type: string
```

```
    description: url of the git repo for the code of deployment
  - name: git-revision
    type: string
    description: revision to be used from repo of the code for deployment
    default: "pipelines-1.4"
  - name: IMAGE
    type: string
    description: image to be built from the code
  tasks:
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.git-url)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: $(params.git-revision)
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - fetch-repository
  - name: apply-manifests
    taskRef:
      name: apply-manifests
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - build-image
  - name: update-deployment
    taskRef:
      name: update-deployment
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

Pipeline 定义提取 Git 源存储库和镜像 registry 的特定内容。当一个管道被触发并执行时，这些详细信息会作为 **params** 添加。

2. 创建管道：

```
$ oc create -f <pipeline-yaml-file-name.yaml>
```

或者，还可以从 Git 存储库直接执行 YAML 文件：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/04_pipeline.yaml
```

3. 使用 **tkn pipeline list** 命令来验证管道是否已添加到应用程序中：

```
$ tkn pipeline list
```

检查输出来验证创建了 **build-and-deploy** pipeline：

```
NAME              AGE            LAST RUN   STARTED   DURATION   STATUS
build-and-deploy  1 minute ago   ---        ---       ---        ---
```

3.5.5. 镜像以在受限环境中运行管道

要在断开连接的集群或受限环境中置备的集群中运行 OpenShift Pipelines，请确保为受限网络配置了 Samples Operator，或者集群管理员创建了带有镜像 registry 的集群。

以下流程使用 **pipelines-tutorial** 示例，使用带有镜像 registry 的集群在受限环境中为应用程序创建管道。为确保 **pipelines-tutorial** 示例在受限环境中工作，您必须为前端接口（**pipelines-vote-ui**）后端接口（**pipelines-vote-api**）和 **cli** 从 mirror registry 中镜像相应的构建器镜像。

流程

1. 为前端接口 **pipelines-vote-ui** 从 mirror registry 中镜像构建器镜像。

   a. 验证所需镜像标签没有导入：

```
$ oc describe imagestream python -n openshift
```

输出示例

```
Name:   python
Namespace:  openshift
[...]

3.8-ubi8 (latest)
  tagged from registry.redhat.io/ubi8/python-38:latest
    prefer registry pullthrough when referencing this tag

  Build and run Python 3.8 applications on UBI 8. For more information about using this builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-python-container/blob/master/3.8/README.md.
  Tags: builder, python
  Supports: python:3.8, python
```

```
  Example Repo: https://github.com/sclorg/django-ex.git

[...]
```

b. 将支持的镜像标签镜像到私有 registry：

```
$ oc image mirror registry.redhat.io/ubi8/python-38:latest <mirror-registry>:
<port>/ubi8/python-38
```

c. 导入镜像：

```
$ oc tag <mirror-registry>:<port>/ubi8/python-38 python:latest --scheduled -n openshift
```

您必须定期重新导入镜像。**--scheduled** 标志启用镜像自动重新导入。

d. 验证带有指定标签的镜像已被导入：

```
$ oc describe imagestream python -n openshift
```

输出示例

```
Name:   python
Namespace:  openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/ubi8/python-38

  * <mirror-registry>:<port>/ubi8/python-
38@sha256:3ee3c2e70251e75bfeac25c0c33356add9cc4abcbc9c51d858f39e4dc29c5f58


[...]
```

2. 为后端接口 **pipelines-vote-api** 从 mirror registry 中镜像构建器镜像。

a. 验证所需镜像标签没有导入：

```
$ oc describe imagestream golang -n openshift
```

输出示例

```
Name:   golang
Namespace:  openshift
[...]

1.14.7-ubi8 (latest)
  tagged from registry.redhat.io/ubi8/go-toolset:1.14.7
    prefer registry pullthrough when referencing this tag

  Build and run Go applications on UBI 8. For more information about using this builder
image, including OpenShift considerations, see https://github.com/sclorg/golang-
container/blob/master/README.md.
  Tags: builder, golang, go
```

```
  Supports: golang
  Example Repo: https://github.com/sclorg/golang-ex.git

[...]
```

b. 将支持的镜像标签镜像到私有 registry：

```
$ oc image mirror registry.redhat.io/ubi8/go-toolset:1.14.7 <mirror-registry>:
<port>/ubi8/go-toolset
```

c. 导入镜像：

```
$ oc tag <mirror-registry>:<port>/ubi8/go-toolset golang:latest --scheduled -n openshift
```

您必须定期重新导入镜像。**--scheduled** 标志启用镜像自动重新导入。

d. 验证带有指定标签的镜像已被导入：

```
$ oc describe imagestream golang -n openshift
```

输出示例

```
Name:   golang
Namespace:  openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/ubi8/go-toolset

  * <mirror-registry>:<port>/ubi8/go-
toolset@sha256:59a74d581df3a2bd63ab55f7ac106677694bf612a1fe9e7e3e1487f55c421
b37

[...]
```

3. 从 **cli** 的镜像 registry 中镜像构建器镜像。

a. 验证所需镜像标签没有导入：

```
$ oc describe imagestream cli -n openshift
```

输出示例

```
Name:                cli
Namespace:           openshift
[...]

latest
  updates automatically from registry quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


  * quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551
```

```
[...]
```

b. 将支持的镜像标签镜像到私有 registry：

```
$ oc image mirror quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551
<mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev:latest
```

c. 导入镜像：

```
$ oc tag <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev cli:latest --
scheduled -n openshift
```

您必须定期重新导入镜像。**--scheduled** 标志启用镜像自动重新导入。

d. 验证带有指定标签的镜像已被导入：

```
$ oc describe imagestream cli -n openshift
```

输出示例

```
Name:                   cli
Namespace:              openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/openshift-release-dev/ocp-
v4.0-art-dev

  * <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


[...]
```

其他资源

- 为受限集群配置 Samples Operator
- 创建带有镜像 registry 的集群

3.5.6. 运行管道

PipelineRun 资源启动管道，并将其与 Git 和用于特定调用的镜像资源相关联。它为管道中的每个任务自动创建并启动 **TaskRun** 资源。

流程

1. 启动后端应用程序的管道：

```
$ tkn pipeline start build-and-deploy \
```

```
    -w name=shared-
workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
    -p deployment-name=pipelines-vote-api \
    -p git-url=https://github.com/openshift/pipelines-vote-api.git \
    -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
vote-api
```

上一命令使用卷声明模板，该模板为管道执行创建持久性卷声明。

2. 要跟踪管道运行的进度，请输入以下命令：

```
$ tkn pipelinerun logs <pipelinerun_id> -f
```

上述命令中的 <pipelinerun_id> 是上一命令输出返回的 **PipelineRun** 的 ID。

3. 启动前端应用程序的管道：

```
$ tkn pipeline start build-and-deploy \
    -w name=shared-
workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
    -p deployment-name=pipelines-vote-ui \
    -p git-url=https://github.com/openshift/pipelines-vote-ui.git \
    -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
vote-ui
```

4. 要跟踪管道运行的进度，请输入以下命令：

```
$ tkn pipelinerun logs <pipelinerun_id> -f
```

上述命令中的 <pipelinerun_id> 是上一命令输出返回的 **PipelineRun** 的 ID。

5. 几分钟后，使用 **tkn pipelinerun list** 命令列出所有管道运行来验证管道是否成功运行：

```
$ tkn pipelinerun list
```

输出列出了管道运行：

```
NAME                     STARTED      DURATION     STATUS
build-and-deploy-run-xy7rw   1 hour ago   2 minutes    Succeeded
build-and-deploy-run-z2rz8   1 hour ago   19 minutes   Succeeded
```

6. 获取应用程序路由：

```
$ oc get route pipelines-vote-ui --template='http://{{.spec.host}}'
```

记录上一个命令的输出。您可以使用此路由来访问应用程序。

7. 要重新运行最后的管道运行,请使用上一管道的管道资源和服务帐户运行：

```
$ tkn pipeline start build-and-deploy --last
```

3.5.7. 在管道中添加触发器

触发器（Trigger）使 Pipelines 可以响应外部 GitHub 事件，如推送事件和拉取请求。在为应用程序组装并启动管道后，添加 **TriggerBinding**、**TriggerTemplate**、**Trigger** 和 **EventListener** 资源来捕获 GitHub 事件。

流程

1. 复制以下 **TriggerBinding** YAML 示例文件的内容并保存：

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerBinding
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

2. 创建 **TriggerBinding** 资源：

```
$ oc create -f <triggerbinding-yaml-file-name.yaml>
```

或者，您可以直接从 **pipelines-tutorial** Git 仓库创建 **TriggerBinding** 资源：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/01_binding.yaml
```

3. 复制以下 **TriggerTemplate** YAML 示例文件的内容并保存：

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerTemplate
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    description: The git repository url
  - name: git-revision
    description: The git revision
    default: pipelines-1.4
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates:
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
      generateName: build-deploy-$(tt.params.git-repo-name)-
    spec:
      serviceAccountName: pipeline
```

```
    pipelineRef:
      name: build-and-deploy
    params:
    - name: deployment-name
      value: $(tt.params.git-repo-name)
    - name: git-url
      value: $(tt.params.git-repo-url)
    - name: git-revision
      value: $(tt.params.git-revision)
    - name: IMAGE
      value: image-registry.openshift-image-registry.svc:5000/pipelines-
tutorial/$(tt.params.git-repo-name)
    workspaces:
    - name: shared-workspace
      volumeClaimTemplate:
        spec:
          accessModes:
            - ReadWriteOnce
          resources:
            requests:
              storage: 500Mi
```

模板指定一个卷声明模板，用于创建用于为工作空间定义存储卷的持久性卷声明。因此，您不需要创建持久性卷声明来提供数据存储。

4. 创建 **TriggerTemplate** 资源：

```
$ oc create -f <triggertemplate-yaml-file-name.yaml>
```

另外，您还可以从 **pipelines-tutorial** Git 仓库直接创建 **TriggerTemplate** 资源：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-
1.4/03_triggers/02_template.yaml
```

5. 复制以下 **Trigger** YAML 示例文件的内容并保存：

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: Trigger
metadata:
  name: vote-trigger
spec:
  serviceAccountName: pipeline
  bindings:
    - ref: vote-app
  template:
    ref: vote-app
```

6. 创建 **Trigger** 资源：

```
$ oc create -f <trigger-yaml-file-name.yaml>
```

另外，您还可以直接从 **pipelines-tutorial** Git 仓库创建 **Trigger** 资源：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/03_trigger.yaml
```

7. 复制以下 **EventListener** YAML 示例文件的内容并保存：

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
    - triggerRef: vote-trigger
```

或者，如果您还没有定义触发器自定义资源，将绑定和模板规格添加到 **EventListener** YAML 文件中，而不是引用触发器的名称：

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
  - bindings:
    - ref: vote-app
    template:
      ref: vote-app
```

8. 通过执行以下步骤来创建 **EventListener** 资源：

- 使用安全 HTTPS 连接创建 **EventListener** 资源：

a. 添加一个标签，在 Eventlistener 资源中启用安全 **HTTPS** 连接：

```
$ oc label namespace <ns-name> operator.tekton.dev/enable-annotation=enabled
```

b. 创建 **EventListener** 资源：

```
$ oc create -f <eventlistener-yaml-file-name.yaml>
```

或者，您可以直接从 **pipelines-tutorial** Git 仓库创建 **EvenListener** 资源：

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/04_event_listener.yaml
```

c. 使用重新加密 TLS 终止创建路由：

```
$ oc create route reencrypt --service=<svc-name> --cert=tls.crt --key=tls.key --ca-cert=ca.crt --hostname=<hostname>
```

另外，您可以创建一个重新加密 TLS 终止 YAML 文件，以创建安全路由。

安全路由重新加密 TLS 终止 YAML 示例

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-passthrough-secured (1)
spec:
  host: <hostname>
  to:
    kind: Service
    name: frontend (2)
  tls:
    termination: reencrypt (3)
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |- (4)
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

(1) (2) 对象的名称，长度限于 63 个字符。

(3) **termination** 字段设置为 **reencrypt**。这是唯一需要 **tls** 的字段。

(4) 重新加密需要。**destinationCACertificate** 指定用来验证端点证书的 CA 证书，保护从路由器到目标 pod 的连接。如果服务使用服务签名证书，或者管理员为路由器指定默认 CA 证书，且服务有由该 CA 签名的证书，则可以省略此字段。

如需了解更多选项，请参阅 **oc create route reencrypt --help**。

- 使用不安全的 HTTP 连接创建 **EventListener** 资源：

  a. 创建 **EventListener** 资源。

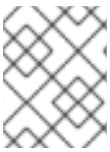
  b. 将 **EventListener** 服务公开为 OpenShift Container Platform 路由，使其可以被公开访问：

```
$ oc expose svc el-vote-app
```

3.5.8. 创建 Webhook

Webhook 是事件监听程序在存储库中配置事件时接收到的 HTTP POST 信息。然后，事件有效负载映射到触发器绑定，并由触发器模板处理。触发器模板最终启动一个或多个管道运行，从而创建并部署 Kubernetes 资源。

在本小节中，您将在 Git 存储库 **pipelines-vote-ui** 和 **pipelines-vote-api** 的副本中配置 webhook URL。这个 URL 指向公开访问的 **EventListener** 服务路由。



注意

添加 Webhook 需要对该存储库有管理特权。如果您没有对库的管理权限，请联络您的系统管理员来添加 webhook。

流程

1. 获取 Webhook URL：
   - 对于安全 HTTPS 连接：

     ```
     $ echo "URL: $(oc  get route el-vote-app --template='https://{{.spec.host}}')"
     ```

   - 对于 HTTP（不安全）连接：

     ```
     $ echo "URL: $(oc  get route el-vote-app --template='http://{{.spec.host}}')"
     ```

     记录下输出中的 URL。
2. 在前端存储库中手动配置 Webhook：
   a. 在浏览器中打开前端 Git 存储库 **pipelines-vote-ui**。
   b. 点 **Settings → Webhooks → Add Webhook**
   c. 在 **Webhooks/Add Webhook** 页面中：
      i. 在 **Payload URL** 字段中输入第 1 步中的 webhook URL
      ii. 为 **Content type** 选择 **application/json**
      iii. 在 **Secret** 字段中指定 secret
      iv. 确定选择了 **Just the push event**
      v. 选择 **Active**
      vi. 点击 **Add webhook**。
3. 重复步骤 2 来使用后端存储库 **pipelines-vote-api**。

3.5.9. 触发一个管道运行

每当 Git 仓库中发生 **push** 事件时，配置的 Webhook 会将事件有效负载发送到公开的 **EventListener** 服务路由。应用程序的 **EventListener** 服务处理有效负载，并将其传递给相关的 **TriggerBinding** 和 **TriggerTemplate** 资源对。**TriggerBinding** 资源提取参数，**TriggerTemplate** 资源使用这些参数并指定必须创建资源的方式。这可能会重建并重新部署应用程序。

在本小节中，您将把一个空的提交推送到前端 **pipelines-vote-ui** 存储库，该存储库将触发管道运行。

流程

1. 在终端中，克隆 fork 的 Git 存储库 **pipelines-vote-ui**：

   ```
   $ git clone git@github.com:<your GitHub ID>/pipelines-vote-ui.git -b pipelines-1.4
   ```

2. 推送空提交：

   ```
   $ git commit -m "empty-commit" --allow-empty && git push origin pipelines-1.4
   ```

3. 检查管道运行是否已触发：

```
$ tkn pipelinerun list
```

请注意，一个新的管道运行被启动。

3.5.10. 其他资源

- 如需了解更多频道在 **Developer** 视角的信息，请参阅在 **Developer** 视角中使用频道小节中的内容。
- 要了解更多有关安全性上下文约束（SCC）的信息，请参阅 管理安全性上下文约束部分。
- 如需有关可重复使用的任务的更多示例，请参阅 OpenShift Catalog 仓库。另外，您还可以在 Tekton 项目中看到 Tekton Catalog。
- 有关重新加密 TLS 终止的详情，请参阅重新加密终止。
- 有关安全路由的详情，请参阅安全路由部分。

3.6. 在 DEVELOPER 视角中使用 RED HAT OPENSHIFT PIPELINES

您可以使用 OpenShift Container Platform web 控制台的 **Developer** 视角为软件交付过程创建 CI/CD 管道.

在 **Developer** 视角中：

- 使用 **Add → Pipeline → Pipeline Builder** 选项为您的应用程序创建自定义管道。
- 在 OpenShift Container Platform 上创建应用程序时，通过 **Add → From Git** 选项来使用 Operator 安装的频道模板和资源来创建频道。

在为应用程序创建管道后，可以在 **Pipelines** 视图中查看并以视觉化的形式与部署进行交互。您还可以使用 **Topology** 视图与使用 **From Git** 选项创建的管道交互。您需要将自定义标识应用到使用 **Pipeline Builder** 创建的管道，以便在 **Topology** 视图中查看它。

先决条件

- 您可以访问 OpenShift Container Platform 集群，并切换到 **Developer** 视角。
- 已在集群中 安装了 OpenShift Pipelines Operator。
- 您是集群管理员，或是有创建和编辑权限的用户。
- 您已创建了一个项目。

3.6.1. 使用 Pipeline Builder 构建管道

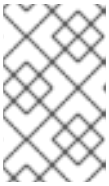
在控制台的 **Developer** 视角中，您可以使用 **+Add → Pipeline → Pipeline Builder** 选项：

- 使用 **Pipeline 构建器** 或 **YAML 视图** 配置管道。
- 使用现有任务和集群任务构建管道流。安装 OpenShift Pipelines Operator 时，它会在集群中添加可重复使用的管道集群任务。
- 指定管道运行所需的资源类型，如有必要，将额外参数添加到管道。
- 引用管道中的每个任务中的这些管道资源作为输入和输出资源。

- 如果需要，引用任务中添加至管道的任何额外参数。任务的参数会根据任务的规格预先填充。
- 使用 Operator 安装的、可重复使用的片断和示例来创建详细的管道。

流程

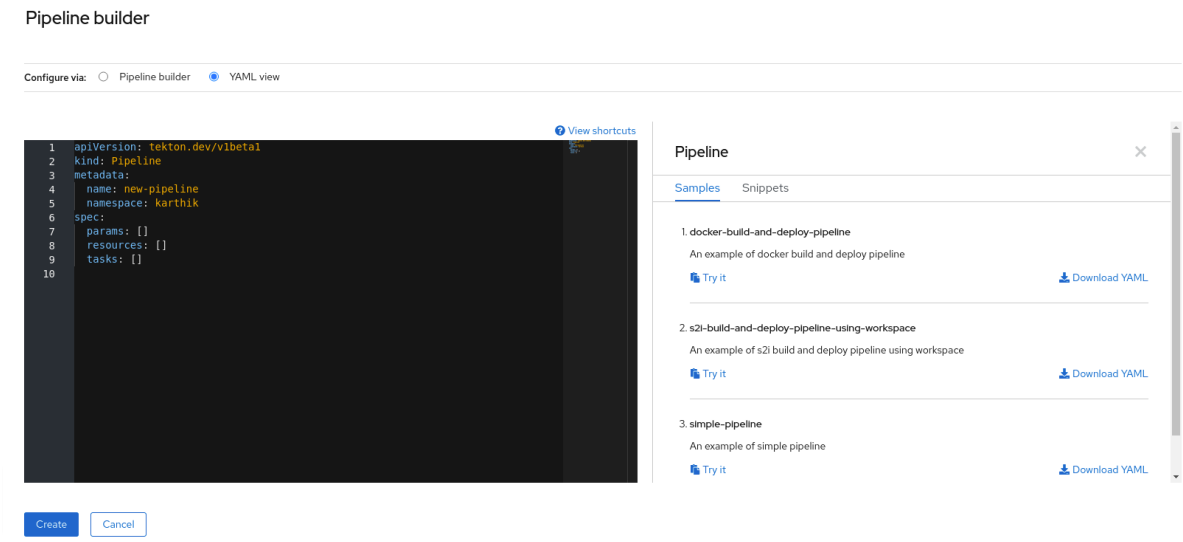
1. 在 **Developer** 视角的 **Add** 视图中，点 **Pipeline** 标题查看 **Pipeline Builder** 页面。
2. 使用 **Pipeline 构建器** 视图或 **YAML 视图**配置管道。



注意

Pipeline 构建器视图支持有限的字段，而 **YAML 视图**支持所有可用字段。（可选）您还可以使用 Operator 安装的、可重复使用的代码片断和样本来创建详细的管道。

图 3.1. YAML 视图



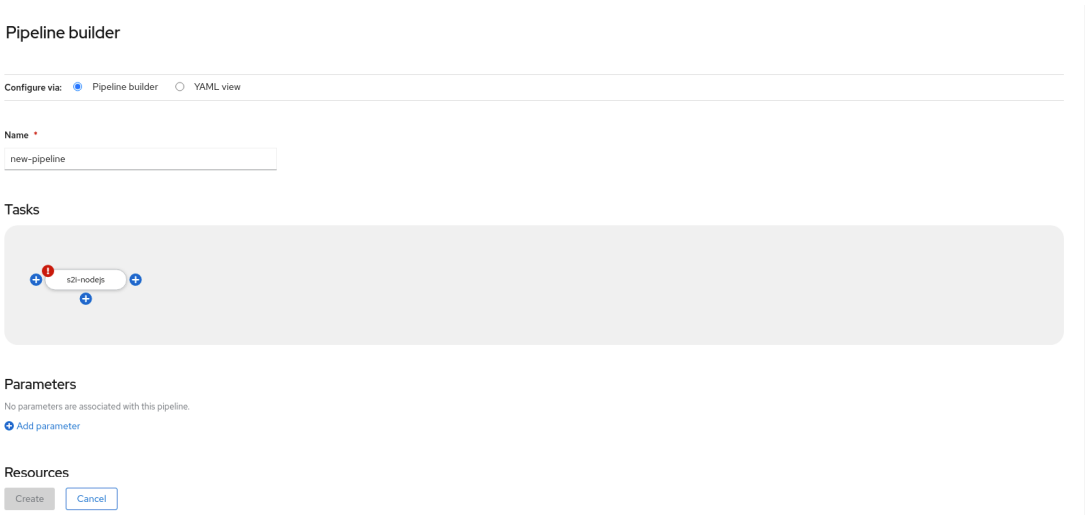
使用 **Pipeline Builder** 配置管道：

a. 为管道输入唯一名称。

b. 从 **Select Task** 列表中选择一个任务，将任务添加到管道。这个示例使用 **s2i-nodejs** 任务。

- 要为管道添加后续任务，点任务右侧或左边的加号图标，从 **Select Task** 列表中选择您要添加到管道的任务。在本例中，使用 **s2i-nodejs** 任务右侧的加号图标来添加 **openshift-client** 任务。
- 要为现有任务添加并行任务，请点击任务旁边显示的加号图标，从 **Select Task** 列表中选择您要添加到管道的并行任务。

图 3.2. Pipeline Builder



c. 点 Add Resources 指定管道运行要使用的资源的名称和类型。这些资源然后会被管道中的任务使用作为输入和输出。在此例中：

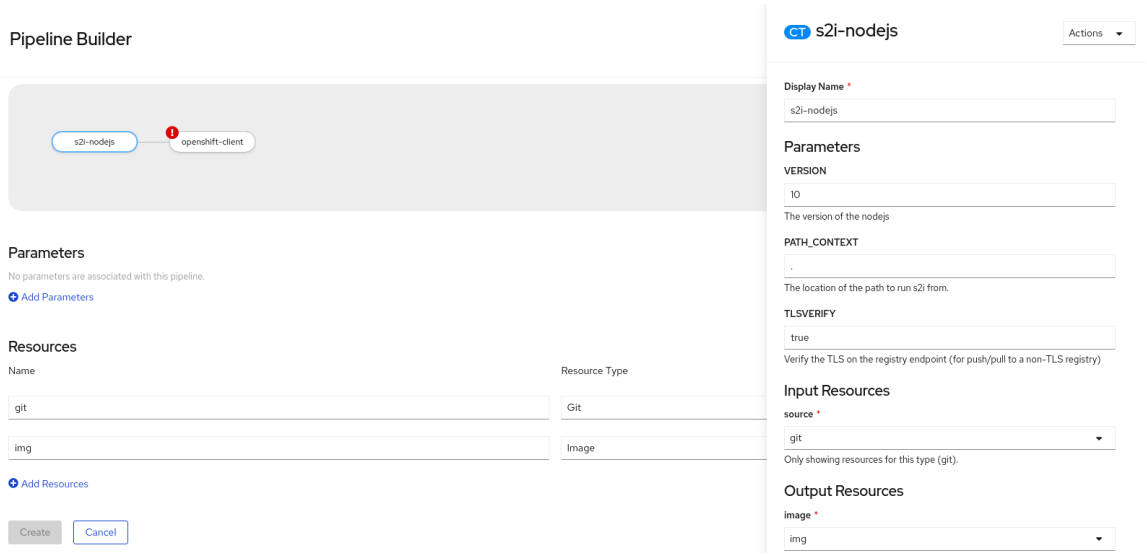
   i. 添加一个输入资源。在 Name 字段中输入 **Source**，从 Resource Type 下拉列表中选择 **Git**。

   ii. 添加一个输出资源。在 Name 字段中输入 **img**，从 Resource Type 下拉列表中选择 **Image**。

d. 可选：任务的**参数**会根据任务的规格预先填充。如果需要，使用 Add Parameters 链接来添加额外的参数。

e. 如果没有指定任务的资源，则会在任务中显示 Missing Resources 警告。点 s2i-nodejs 任务，查看任务的详情侧面板。

图 3.3. Pipelines Builder 中的任务详情



f. 在任务侧面板中，为 s2i-nodejs 任务指定资源和参数：

   i. 在 Input Resources → Source 部分中, Select Resources 下拉列表显示添加到管道中的资源。在本例中，选择 **Source**。

   ii. 在 Output Resources → Image 部分，点 Select Resources 列表，然后选择 **img**。

iii. 如果需要，在 Parameters 项中，使用 **$(params.<param-name>)** 语法在默认参数外添加更多参数。

iv. 同样，为 **openshift-client** 任务添加输入资源。

3. 点 **Create** 在 **Pipeline Details** 页面中创建并查看管道。

4. 点 **Actions** 下拉菜单，然后点 **Start** 启动管道。

3.6.2. 使用 OpenShift Pipelines 创建应用程序

要与应用程序一同创建管道，使用 **Developer** 视角的 **Add 视图** 中的 **From Git** 选项。如需更多信息，请参阅使用 Developer 视角创建应用程序。

3.6.3. 使用 Developer 视角与管道交互

开发者视角中的 **Pipelines** 视图列出了项目中的所有管道，以及以下详细信息：

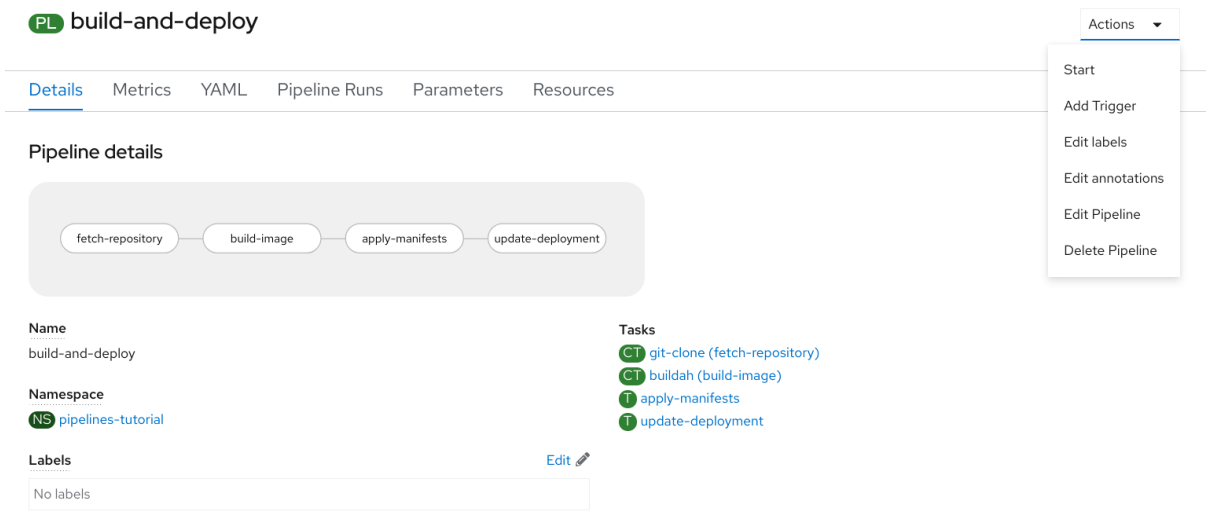
- 创建管道的命名空间
- 最后一次管道运行
- 管道运行中的任务状态
- 管道运行的状态
- 最后一次管道运行的创建时间

流程

1. 在 **Developer** 视角的 **Pipelines** 视图中，从 **Project** 下拉列表中选择一个项目，以查看该项目中的管道。

2. 点击所需管道查看 **Pipeline 详情**页面。默认情况下，**Details** 选项卡会打开，并显示管道中所有串行和并行任务。这些任务也列在本页的右下角。您可以点击列出的**任务**来查看任务详情。

图 3.4. Pipeline 详情

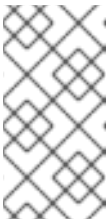


3. 另外，在 **Pipeline 详情**页面中：

- 点 **Metrics** 选项卡查看有关管道的以下信息：

- **Pipeline 成功率**
- **Pipeline 运行数量**
- **管道运行持续时间**
- **任务运行持续时间**
  您可以使用这些信息来改进管道工作流，并在管道生命周期的早期解决问题。

- 点 **YAML** 选项卡编辑管道的 YAML 文件。
- 点击 **Pipeline Runs** 标签页，查看完成、正在运行或运行失败的管道。



注意

Pipeline Run Details 页面的 **Details** 部分显示失败管道运行的**日志片段**。**日志片断** 提供一般错误信息和日志片断。**Logs** 部分的链接可让您快速访问失败运行的详细信息。**日志片断**也会显示在 **Task Run Details** 页面的 **Details** 部分。

您可以使用 Options 菜单  来停止正在运行的管道，使用与之前的管道执行相同的参数和资源重新运行管道，或删除管道运行。

- 点 **Parameters** 标签，查看管道中定义的参数。您还可以根据需要添加或者编辑附加参数。
- 点 **Resources** 标签页，查看管道中定义的资源。您还可以根据需要添加或编辑附加资源。

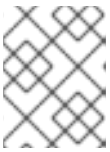
3.6.4. 启动管道

创建管道后，您需要启动它以在定义的顺序中执行包含的任务。您可从 **Pipelines** 视图、**Pipeline Details** 页面或 **Topology** 视图启动管道。

流程

使用 **Pipelines** 视图启动管道：

1. 在 **Developer 视角** 的 **Pipelines** 视图中，点附加到 Pipeline 的 **Options**  菜单，然后选择 **Start**。
2. **Start Pipeline** 对话框显示 **Git Resources** 以及基于管道定义的 **Image Resources**。



注意

对于使用 **From Git** 选项创建的管道，**Start Pipeline** 对话框也会在 **Parameters** 部分显示 **APP_NAME** 字段，对话框中的所有字段都由管道模板预先填充。

   a. 如果您在命名空间中有资源，**Git Resources** 和 **Image Resources** 字段会预先填充这些资源。如果需要，使用下拉菜单选择或创建所需资源并自定义管道运行实例。

3. 可选：修改 **Advanced Options** 以添加验证指定私有 Git 服务器或镜像 registry 的凭证。
   a. 在 **Advanced Options** 下，点 **Show Credentials Options** 并选择 **Add Secret**。

b. 在 **Create Source Secret**部分，指定以下内容：

i. secret 的唯一 **Secret Name**。

ii. 在 **要被验证的指定供应商** 部分，在 **Access to** 字段中指定要验证的供应商，以及基本 **服务器 URL**。

iii. 选择 **Authentication Type** 并提供凭证：

- 对于 **Authentication Type Image Registry Credentials**，请指定您要进行身份验证的 **Registry 服务器地址**，并通过**Username**、**Password** 和 **Email** 项中提供您的凭证。
  如果要指定额外的 **Registry 服务器地址**，选择 **Add Credentials**。

- 如果 **Authentication Type** 为 **Basic Authentication**，在 **UserName** 和 **Password or Token** 项中指定相关的值。

- 如果 **Authentication Type** 为 **SSH Keys** 时，在 **SSH Private Key** 字段中指定相关的值。

iv. 选择**要添加** secret 的检查标记。

您可以根据频道中的资源数量添加多个 secret。

4. 点 **Start** 启动管道。

5. **Pipeline Run Details** 页面显示正在执行的管道。管道启动后，每个任务中的任务和步骤都会被执行。您可以：

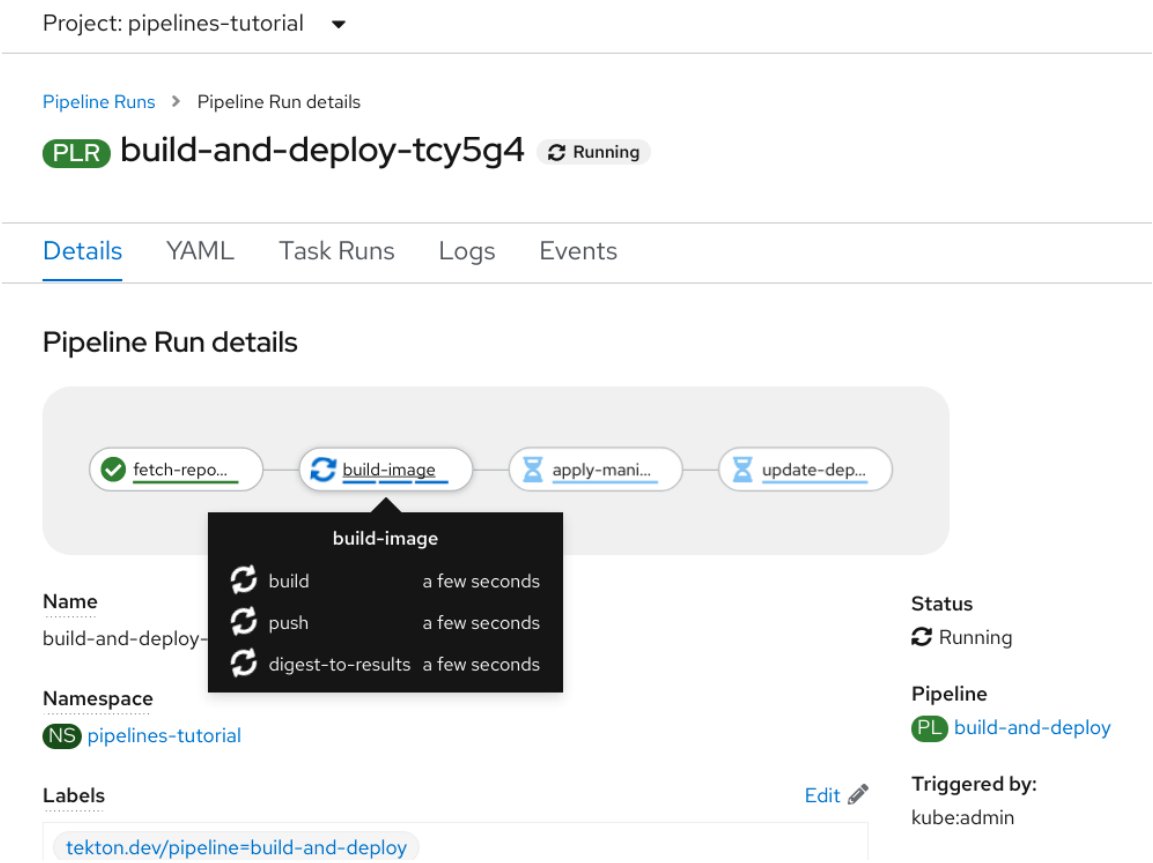
- 将鼠标悬停在任务上，以查看执行每一步骤所需时间。

- 点一个任务来查看任务中每一步骤的日志。

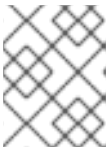
- 点 **Logs** 选项卡查看与任务执行顺序相关的日志。您还可以使用相关按钮扩展窗格，单独或批量下载日志。

- 点 **Events** 选项卡查看管道运行生成的事件流。
  您可以使用 **Task Runs**、**Logs** 和 **Events** 选项卡来帮助调试失败的管道运行或失败的任务运行。

图 3.5. Pipeline 运行详情



6. 对于使用 **From Git** 选项创建的管道，您可以在启动后使用 **Topology** 视图来与管道进行交互：



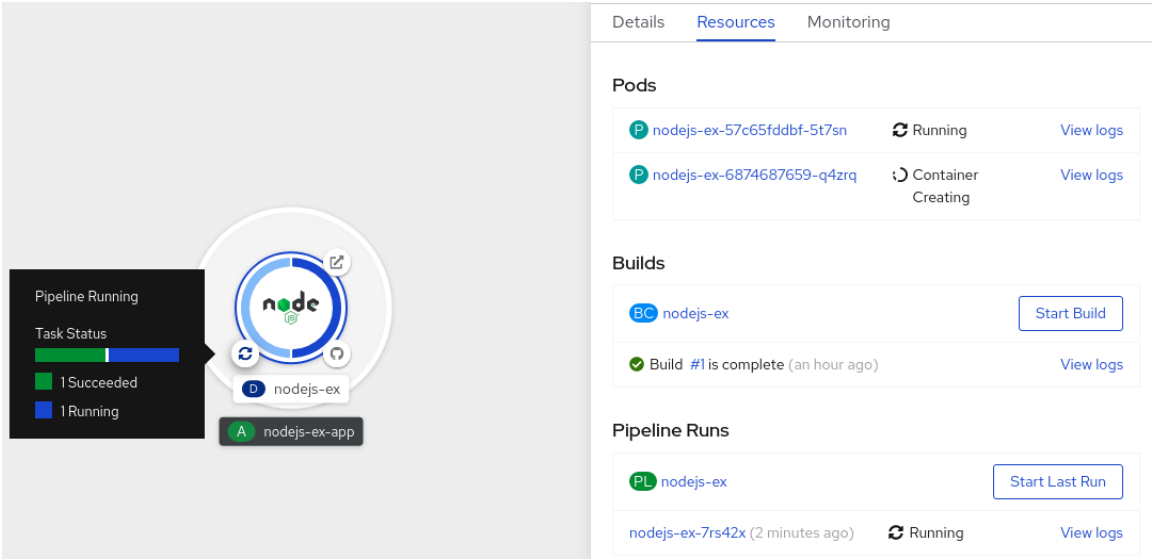
注意

要使用 **Pipeline Builder** 在 **Topology** 视图中查看创建的管道，自定义管道标识来把 Pipeline 与应用程序负载相连接。

a. 在左侧导航面板中，点 **Topology**，然后点应用程序来查看在侧面面板中列出的管道运行。

b. 在 **Pipeline Runs** 部分,点击 **Start Last Run** 来启动一个新的管道运行，使用与前一个相同的参数和资源。如果没有启动管道运行，这个选项会被禁用。

图 3.6. Topology 视图中的管道



c. 在 **Topology** 页面中，把鼠标移到应用程序的左侧，以查看应用程序的管道运行状态。

注意

当管道在特定任务运行时失败时，**Topology** 页面中的应用程序节点的侧面板会显示一个**日志片段**。您可以在 **Pipeline Runs** 部分的 **Resources** 选项卡中查看**日志片断**。**日志片断** 提供一般错误信息和日志片断。**Logs** 部分的链接可让您快速访问失败运行的详细信息。

3.6.5. 编辑管道

您可以使用 web 控制台的 **Developer** 视角编辑集群中的 Pipelines:

流程

1. 在 **Developer** 视角的 **Pipelines** 视图中，选择您要编辑的管道来查看 Pipeline 的详情。在 **Pipeline Details** 页中，点 **Actions** 并选择 **Edit Pipeline**。
2. 在 **Pipeline Builder** 页中：
   - 您可以将任务、参数或资源添加到管道。
   - 您可以点要修改的任务来查看侧面面板中的任务详情，并修改所需的任务详情，如显示名称、参数和资源。
   - 或者，要删除此任务，点任务，在侧面面板中点 **Actions**，并选择 **Remove Task**。
3. 点 **Save** 来保存修改的管道。

3.6.6. 删除管道

您可以使用 web 控制台的 **Developer** 视角删除集群中的管道。

流程

1. 在 **Developer 视角** 的 **Pipelines** 视图中，点附加到 Pipeline 的 **Options** 菜单，然后选择 **Delete Pipeline**。
2. 在 **Delete Pipeline** 确认提示下，点 **Delete** 以确认删除。

3.7. 减少管道的资源消耗

如果在多租户环境中使用集群，您必须控制各个项目和 Kubernetes 对象的 CPU、内存和存储资源的消耗。这有助于防止一个应用程序消耗太多资源并影响其他应用程序。

要定义在生成的 pod 上设置的最终资源限值，Red Hat OpenShift Pipelines 使用资源配额限制和/或限制执行项目的范围。

要限制项目中的资源消耗，您可以：

- 设置和管理资源配额来限制聚合资源消耗。
- 使用限制范围来限制特定对象的资源消耗，如 pod、镜像、镜像流和持久性卷声明。

3.7.1. 了解管道中的资源消耗

每个任务都由 **Task** 资源中 **steps** 字段中定义的特定顺序执行的一些所需步骤组成。每个任务都作为 pod 运行，每个步骤都作为该 pod 中的容器运行。

每次都会执行一个步骤。执行此任务的 Pod 仅请求足够的资源，以一次在任务中运行单个容器镜像（步骤），因此不会为任务中的所有步骤存储资源。

steps spec 中的 **Resources** 字段指定资源消耗的限值。默认情况下，CPU、内存和临时存储的资源请求设置为 **BestEffort** （零）值，或通过该项目中限制范围设置的最小值。

步骤的资源请求和限值配置示例

```
spec:
  steps:
  - name: <step_name>
    resources:
      requests:
        memory: 2Gi
        cpu: 600m
      limits:
        memory: 4Gi
        cpu: 900m
```

当在执行管道和任务运行的项目中指定 **LimitRange** 参数和容器资源请求的最小值时，Red Hat OpenShift Pipelines 会查看项目中的所有 **LimitRange** 值，并使用最小值而不是零。

在项目级别配置限制范围参数示例

```
apiVersion: v1
kind: LimitRange
metadata:
  name: <limit_container_resource>
spec:
  limits:
  - max:
      cpu: "600m"
      memory: "2Gi"
    min:
      cpu: "200m"
      memory: "100Mi"
    default:
      cpu: "500m"
      memory: "800Mi"
    defaultRequest:
      cpu: "100m"
      memory: "100Mi"
    type: Container
...
```

3.7.2. 缓解管道中的额外资源消耗

当您在 pod 中的容器上设置了资源限制时，OpenShift Container Platform 会将请求的资源限值作为同时运行的所有容器的总和。

为了消耗调用任务一次执行一个步骤所需的最小资源，Red Hat OpenShift Pipelines 请求最大 CPU、内存和临时存储，这在需要最多资源的步骤中指定。这样可确保满足所有步骤的资源要求。最大值以外的请求设置为零。

但是，这种行为的资源使用量会超过要求。如果使用资源配额，这也可能导致无法调度的 pod。

例如，有一个任务，它带有两个使用脚本的步骤，且没有定义任何资源限值和请求。生成的 pod 有两个 init 容器（一个用于入口点复制，另一个用于编写脚本）和两个容器（每个步骤一个）。

OpenShift Container Platform 使用为项目设置的限制范围来计算所需资源请求和限值。在本例中，在项目中设置以下限制范围：

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

在这种情况下，每个 init 容器使用 1Gi 请求内存（限制范围的最大限制），每个容器都使用 500Mi 请求内存。因此，pod 的内存请求总数为 2Gi。

如果在 10 个步骤中使用相同的限制范围，则最终内存请求为 5Gi，高于每个步骤实际需要的 5Gi，即 500Mi（因为每个步骤均在另一个步骤之后运行）。

因此，为了减少资源消耗，您可以：

- 通过将不同的步骤分组到一个较大的步骤，使用脚本功能和同一镜像来减少给定任务中的步骤数量。这可减少最低请求资源。
- 分发步骤，它们相互独立，可以对于多个任务而不是单个任务运行。这样可减少每个任务中的步骤数量，使每个任务的请求更小，那么调度程序可在资源可用时运行它们。

3.7.3. 其他资源

- 资源配额
- 使用限制范围限制资源消耗
- Kubernetes 中的资源请求和限值

3.8. 在特权安全上下文中使用 POD

如果 Pod 由管道运行或任务运行，则 OpenShift Pipelines 1.3.x 及更新版本的默认配置不允许使用特权安全上下文运行 Pod。对于这样的容器集，默认服务帐户为 **pipeline**，与 **pipelines** 服务帐户关联的安全性上下文约束（SCC）是 **pipelines-scc**。**pipelines-scc** SCC 与 **anyuid** SCC 类似，但存在细微差别，如管道 SCC 的 YAML 文件中所定义：

SecurityContextConstraints 对象示例

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
...
fsGroup:
  type: MustRunAs
...
```

另外，**Buildah** 集群任务作为 OpenShift Pipelines 的一部分提供，使用 **vfs** 作为默认存储驱动程序。

3.8.1. 运行管道运行和任务运行带有特权安全上下文的 Pod

流程

要使用 **特权** 安全上下文运行 pod（从管道运行或任务运行中），请执行以下修改：

- 将关联的用户帐户或服务帐户配置为具有显式 SCC。您可以使用以下任一方法执行配置：
  - 执行以下 OpenShift 命令：

    ```
    $ oc adm policy add-scc-to-user <scc-name> -z <service-account-name>
    ```

  - 或者，修改 **RoleBinding** 和 **Role** 或 **ClusterRole** 的 YAML 文件：

    RoleBinding 对象示例

    ```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: RoleBinding
    metadata:
      name: service-account-name 1
      namespace: default
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: ClusterRole
      name: pipelines-scc-clusterrole 2
    subjects:
    - kind: ServiceAccount
      name: pipeline
      namespace: default
    ```

    1. 使用适当的服务帐户名称替换。
    2. 根据您使用的角色绑定，使用适当的集群角色替换。

    ClusterRole 对象示例

    ```
    apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRole
    metadata:
      name: pipelines-scc-clusterrole 1
    rules:
    - apiGroups:
      - security.openshift.io
      resourceNames:
    ```

```
- nonroot
resources:
- securitycontextconstraints
verbs:
- use
```

1 根据您使用的角色绑定，使用适当的集群角色替换。

注意

作为最佳实践，请创建默认 YAML 文件的副本并在重复文件中进行更改。

- 如果您不使用 **vfs** 存储驱动程序，请将与任务运行或管道运行关联的服务帐户配置为具有特权 SCC，并将安全上下文设置为 **privileged: true**。

3.8.2. 使用自定义 SCC 和自定义服务帐户运行管道运行和任务

使用与默认管道 **服务帐户** 关联的 **pipelines-scc** 安全性上下文约束(SCC)时，管道运行和任务运行 pod 可能会面临超时问题。这是因为在默认的 **pipelines-scc** SCC 中， **fsGroup.type** 参数设置为 **MustRunAs**。

注意

有关 pod 超时的更多信息，请参阅 BZ#779。

为避免 pod 超时，您可以创建一个自定义 SCC，并将 **fsGroup.type** 参数设置为 **RunAsAny**，并将它与自定义服务帐户关联。

注意

作为最佳实践，使用自定义 SCC 和自定义服务帐户来运行管道运行和任务运行。这种方法具有更大的灵活性，在升级过程中修改默认值时不会中断运行。

流程

1. 定义自定义 SCC，并将 **fsGroup.type** 参数设置为 **RunAsAny** ：

示例：自定义 SCC

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: my-scc is a close replica of anyuid scc. pipelines-scc has
fsGroup - RunAsAny.
  name: my-scc
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
```

```
allowPrivilegedContainer: false
allowedCapabilities: null
defaultAddCapabilities: null
fsGroup:
  type: RunAsAny
groups:
- system:cluster-admins
priority: 10
readOnlyRootFilesystem: false
requiredDropCapabilities:
- MKNOD
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

2. 创建自定义 SCC：

   示例：创建 my-scc SCC

   ```
   $ oc create -f my-scc.yaml
   ```

3. 创建自定义服务帐户：

   示例：创建一个 fsgroup-runasany 服务帐户

   ```
   $ oc create serviceaccount fsgroup-runasany
   ```

4. 将自定义 SCC 与自定义服务帐户关联：

   示例： 将 my-scc SCC 与 fsgroup-runasany 服务帐户关联

   ```
   $ oc adm policy add-scc-to-user my-scc -z fsgroup-runasany
   ```

   如果要将自定义服务帐户用于特权任务，您可以通过运行以下命令将 **特权** SCC 与自定义服务帐户关联：

   示例：将 特权 SCC 与 fsgroup-runasany 服务帐户关联

   ```
   $ oc adm policy add-scc-to-user privileged -z fsgroup-runasany
   ```

5. 在管道运行和任务运行中使用自定义服务帐户：

   示例：Pipeline 使用 fsgroup-runasany 自定义服务帐户运行 YAML

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: <pipeline-run-name>
spec:
  pipelineRef:
    name: <pipeline-cluster-task-name>
  serviceAccountName: 'fsgroup-runasany'
```

示例：任务使用 fsgroup-runasany 自定义服务帐户运行 YAML

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: <task-run-name>
spec:
  taskRef:
    name: <cluster-task-name>
  serviceAccountName: 'fsgroup-runasany'
```

3.8.3. 其他资源

- 有关管理 SCC 的信息，请参阅管理安全性上下文约束。

3.9. 使用 OPENSHIFT LOGGING OPERATOR 查看管道日志

管道运行、任务运行和事件侦听器生成的日志存储在其各自的 pod 中。检查和分析用于故障排除和审计的日志非常有用。

但是，保留 pod 不会造成不必要的资源消耗和杂乱的命名空间。

要消除对 pod 查看管道日志的依赖，您可以使用 OpenShift Elasticsearch Operator 和 OpenShift Logging Operator。这些 Operator 可帮助您使用 Elasticsearch Kibana 堆栈查看管道日志，即使您删除了包含日志的 pod。

3.9.1. 先决条件

在 Kibana 仪表板中尝试查看管道日志前，请确保以下内容：

- 步骤由集群管理员执行。
- 管道运行和任务运行的日志可用。
- 安装了 OpenShift Elasticsearch Operator 和 OpenShift Logging Operator。

3.9.2. 在 Kibana 中查看管道日志

在 Kibana web 控制台中查看管道日志：

流程

1. 以集群管理员身份登录到 OpenShift Container Platform Web 控制台。

2. 在菜单栏右上角，点击 **grid** 图标 → **Observability** → **Logging**。这时会显示 Kibana Web 控制台。

3. 创建索引模式：

   a. 在 **Kibana** Web 控制台左侧导航面板中，点击 **Management**。

   b. 单击 **Create index pattern**。

   c. 在 **Step 1 of 2: Define index pattern**→ **Index pattern** 中输入一个 * 特征并点 **Next Step**。

   d. 在 **Step 2 of 2: Configure settings**→ **Time filter field name** 中，从下来菜单中选择 **@timestamp**，点 **Create index pattern**。

4. 添加过滤器：

   a. 在 **Kibana** Web 控制台左侧导航面板中，点 **Discover**。

   b. 点 **Add a filter +**→ **Edit Query DSL**。

注意

- 对于以下每个示例过滤器，编辑查询并单击 **Save**。
- 这些过滤器会逐个应用。

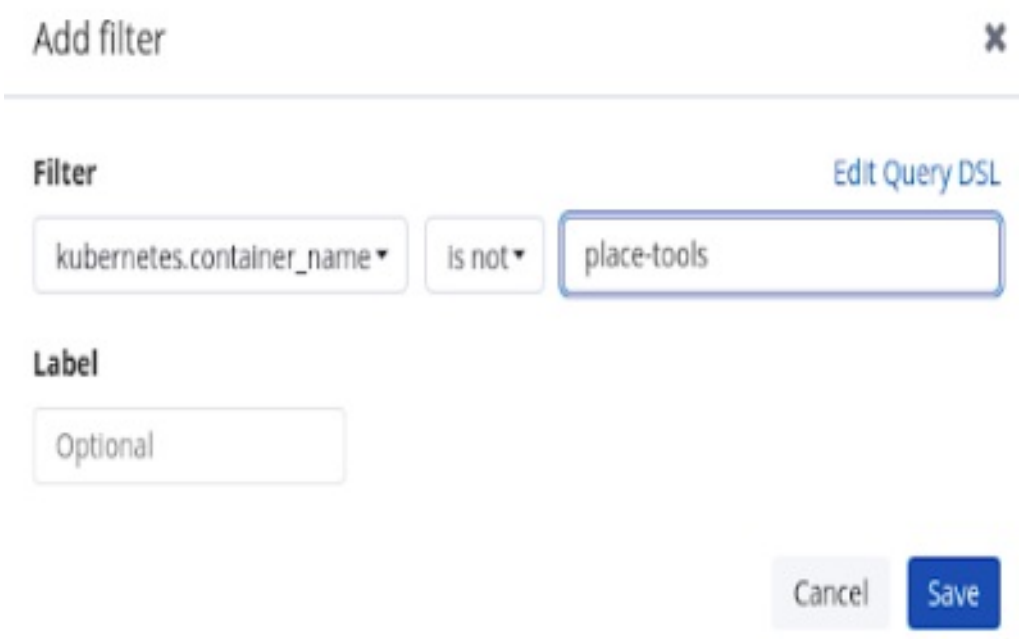
   i. 过滤与管道相关的容器：

过滤管道容器的查询示例

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "app_kubernetes_io/managed-by=tekton-pipelines",
     "type": "phrase"
   }
 }
  }
}
```

   ii. 过滤所有不是 **place-tools** 容器的容器。作为使用图形下拉菜单而不是编辑查询 DSL 的一个示例，请考虑以下方法：

图 3.7. 使用下拉列表字段进行过滤示例



iii. 在标签中过滤 **pipelinerun** 以高亮显示：

在标签中过滤 pipelinerun 的查询示例

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "tekton_dev/pipelineRun=",
     "type": "phrase"
   }
 }
  }
}
```

iv. 在标签中过滤 **pipeline** 以高亮显示：

在标签中过滤 pipeline 以高亮显示的查询示例

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "tekton_dev/pipeline=",
     "type": "phrase"
   }
 }
  }
}
```

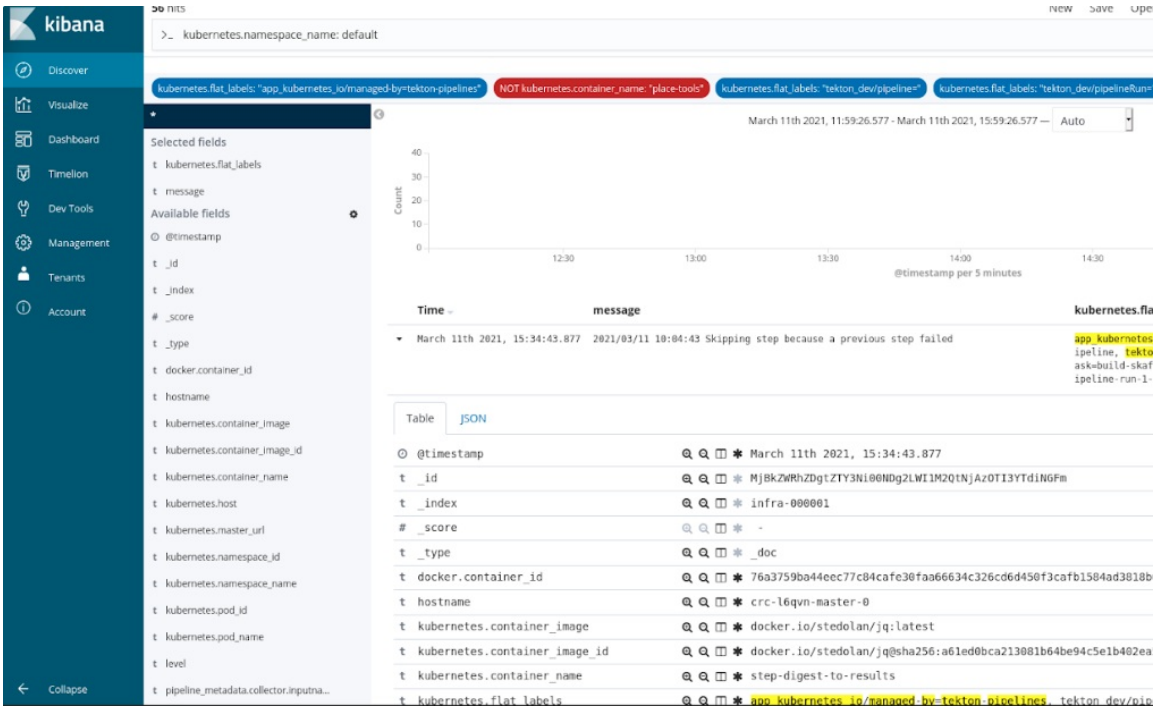
c. 从 **Available fields** 列表中，选择以下字段：

- **kubernetes.flat_labels**

- **message**
  确保所选字段显示在 **Selected fields** 列表下。

d. 日志显示在 **message** 字段下。

图 3.8. 过滤的消息



3.9.3. 其他资源

- 安装 OpenShift Logging
- 查看资源的日志
- 使用 Kibana 查看集群日志

第 4 章 GITOPS

4.1. RED HAT OPENSHIFT GITOPS 发行注记

Red Hat OpenShift GitOps 是为云原生应用程序实施持续部署的一种声明方法。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 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
- 从已知状态恢复或重新创建集群
- 对多个 OpenShift Container Platform 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。

如需了解 Red Hat OpenShift GitOps 的概述，请参阅了解 OpenShift GitOps。

4.1.1. 使开源包含更多

红帽承诺替换我们的代码、文档和网页属性中存在问题的语言。我们从这四个术语开始： master、slave、blacklist 和 whitelist。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 Red Hat CTO Chris Wright 信息。

4.1.2. Red Hat OpenShift GitOps 1.2.1 发行注记

Red Hat OpenShift GitOps 1.2.1 现在包括在 OpenShift Container Platform 4.7 和 4.8 中。

4.1.2.1. 支持列表

这个版本中的一些功能当前还处于技术预览状态。它们并不适用于在生产环境中使用。

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 4.1. 支持列表

功能	Red Hat OpenShift GitOps 1.2.1
Argo CD	GA
Argo CD ApplicationSet	TP

功能	Red Hat OpenShift GitOps 1.2.1
Red Hat OpenShift GitOps Application Manager (kam)	TP

4.1.2.2. 修复的问题

在当前发行版本中解决了以下问题：

- 在以前的版本中，应用程序控制器启动时会在应用程序控制器中观察到大量内存高峰。现在，应用程序控制器的 **--kubectl-parallelism-limit** 标志默认设置为 10，但可以通过在 Argo CD CR 规格中为 **.spec.controller.kubeParallelismLimit** 指定数字来覆盖这个值。GITOPS-1255
- 最新的 Triggers API 会导致在使用 **kam bootstrap** 命令时因为 kustomization.yaml 中的重复条目导致 Kubernetes 构建失败。Pipelines 和 Tekton 会在 v0.24.2 和 v0.14.2 中分别更新来解决这个问题。GITOPS-1273
- 现在，当从源命名空间中删除 Argo CD 实例时，持久化 RBAC 角色和绑定会自动从目标命名空间中移除。GITOPS-1228
- 在以前的版本中，当将 Argo CD 实例部署到命名空间中时，Argo CD 实例会将"managed-by"标签更改为自己的命名空间。在这个版本中，命名空间会取消标记，同时确保为命名空间创建和删除所需的 RBAC 角色和绑定。GITOPS-1247
- 在以前的版本中，Argo CD 工作负载中的默认资源请求限制（特别是 repo-server 和应用程序控制器）被发现非常严格的限制。现有资源配额现已被删除，在仓库服务器中默认内存限值已增加到 1024M。请注意，这个更改只会影响新的安装；现有的 Argo CD 实例工作负载不会受到影响。GITOPS-1274

4.1.3. Red Hat OpenShift GitOps 1.2 发行注记

Red Hat OpenShift GitOps 1.2 现在包括在 OpenShift Container Platform 4.7 和 4.8 中。

4.1.3.1. 支持列表

这个版本中的一些功能当前还处于技术预览状态。它们并不适用于在生产环境中使用。

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 4.2. 支持列表

功能	Red Hat OpenShift GitOps 1.2
Argo CD	GA

功能	Red Hat OpenShift GitOps 1.2
Argo CD ApplicationSet	TP
Red Hat OpenShift GitOps Application Manager (kam)	TP

4.1.3.2. 新功能

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift GitOps 1.2 中的新内容。

- 如果您没有对 openshift-gitops 命名空间的读写访问权限，现在可以使用 GitOps Operator 中的 **DISABLE_DEFAULT_ARGOCD_INSTANCE** 环境变量，并将值设置为 **TRUE** 以防止默认的 Argo CD 实例从 **openshift-gitops** 命名空间启动。

- 现在，在 Argo CD 工作负载中配置了资源请求和限制。在 **openshift-gitops** 命名空间中启用资源配额。因此，手动在 openshift-gitops 命名空间中部署的带外工作负载必须使用资源请求和限制进行配置，并且可能需要增加资源配额。

- Argo CD 身份验证现已与红帽 SSO 集成，并在集群中自动配置 OpenShift 4 身份提供商。此功能默认为禁用。要启用红帽 SSO，请在 **ArgoCD** CR 中添加 SSO 配置，如下所示。目前，**keycloak** 是唯一受支持的提供程序。

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: example-argocd
  labels:
    example: basic
spec:
  sso:
    provider: keycloak
  server:
    route:
     enabled: true
```

- 现在，您可以使用路由标签定义主机名来支持路由器分片。现在，支持在 **server** (argocd server)、**grafana** 和 **prometheus** 路由上设置标签。要在路由上设置标签，请在 **ArgoCD** CR 中的服务器的路由配置下添加标签。

  在 argocd 服务器上设置标签的 ArgoCD CR YAML 示例

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: example-argocd
  labels:
    example: basic
spec:
  server:
    route:
     enabled: true
```

```
labels:
  key1: value1
  key2: value2
```

- GitOps Operator 现在会自动向 Argo CD 实例授予权限，以通过应用标签来管理目标命名空间中的资源。用户可以使用标签 **argocd.argoproj.io/managed-by: <source-namespace>** 标记目标命名空间，其中 **source-namespace** 是部署 argocd 实例的命名空间。

4.1.3.3. 修复的问题

在当前发行版本中解决了以下问题：

- 在以前的版本中，如果用户在 openshift-gitops 命名空间中创建了由默认集群实例管理的 Argo CD 实例，则负责新 Argo CD 实例的应用程序会停留在 **OutOfSync** 状态。现在，通过添加对集群 secret 的所有者引用解决了这个问题。GITOPS-1025

4.1.3.4. 已知问题

Red Hat OpenShift GitOps 1.2 中已知的问题。

- 当从源命名空间中删除 Argo CD 实例时，目标命名空间中的 **argocd.argoproj.io/managed-by** 标签不会被删除。GITOPS-1228

- Red Hat OpenShift GitOps 1.2 中的 openshift-gitops 命名空间中启用了资源配额。这会影响手动部署的带外工作负载，以及 **openshift-gitops** 命名空间中默认 Argo CD 实例部署的工作负载。当您从 Red Hat OpenShift GitOps **v1.1.2** 升级到 **v1.2** 时，此类工作负载必须使用资源请求和限制来配置。如果存在额外的工作负载，则必须增加 openshift-gitops 命名空间中的资源配额。**openshift-gitops** 命名空间的当前资源配额。

资源	Requests	Limits
CPU	6688m	13750m
内存	4544Mi	9070Mi

您可以使用以下命令来更新 CPU 限值。

```
$ oc patch resourcequota openshift-gitops-compute-resources -n openshift-gitops --
type='json' -p='[{"op": "replace", "path": "/spec/hard/limits.cpu", "value":"9000m"}]'
```

您可以使用以下命令来更新 CPU 请求。

```
$ oc patch resourcequota openshift-gitops-compute-resources -n openshift-gitops --
type='json' -p='[{"op": "replace", "path": "/spec/hard/cpu", "value":"7000m"}]
```

您可以替换以上命令中的路径从 **cpu** 到 **memory** 来更新内存。

4.1.4. Red Hat OpenShift GitOps 1.1 发行注记

Red Hat OpenShift GitOps 1.1 现在包括在 OpenShift Container Platform 4.7 中。

4.1.4.1. 支持列表

这个版本中的一些功能当前还处于技术预览状态。它们并不适用于在生产环境中使用。

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 4.3. 支持列表

功能	Red Hat OpenShift GitOps 1.1
Argo CD	GA
Argo CD ApplicationSet	TP
Red Hat OpenShift GitOps Application Manager (kam)	TP

4.1.4.2. 新功能

除了包括修复和稳定性改进的信息外，以下小节突出介绍了 Red Hat OpenShift GitOps 1.1 中的新内容：

- 现在添加了 **ApplicationSet** 功能（技术预览）。**ApplicationSet** 功能可在大量集群和 monorepos 中管理 Argo CD 应用程序时实现自动化和更大的灵活性。它还可在多租户 Kubernetes 集群中实现自助服务。
- Argo CD 现在与集群日志记录堆栈以及 OpenShift Container Platform Monitoring 和 Alerting 功能集成。
- Argo CD auth 现在与 OpenShift Container Platform 集成。
- Argo CD 应用程序控制器现在支持横向扩展。
- Argo CD Redis 服务器现在支持高可用性（HA）。

4.1.4.3. 修复的问题

在当前发行版本中解决了以下问题：

- 在以前的版本中，Red Hat OpenShift GitOps 在带有活跃全局代理设置的代理服务器设置中无法正常工作。这个问题已被解决。现在，Red Hat OpenShift GitOps Operator 会使用 pod 的完全限定域名（FQDN）配置 Argo CD，以启用组件间的通信。GITOPS-703
- Red Hat OpenShift GitOps 后端依赖于 Red Hat OpenShift GitOps URL 中的 **?ref=** 查询参数来发出 API 调用。在以前的版本中，这个参数没有从 URL 中读取，从而导致后端始终考虑默认引用。这个问题已被解决。Red Hat OpenShift GitOps 后端现在从 Red Hat OpenShift GitOps URL 提取引用查询参数，且仅在未提供输入引用时使用默认引用。GITOPS-817

- 在以前的版本中，Red Hat OpenShift GitOps 后端无法找到有效的 GitLab 存储库。这是因为 Red Hat OpenShift GitOps 后端检查 **main** 作为分支引用，而不是 GitLab 存储库中的 **master**。这个问题现已解决。GITOPS-768

- OpenShift Container Platform Web 控制台的 **Developer** 视角中的 **Environments** 页面现在显示应用程序列表和环境数量。本页还显示 Argo CD 链接，它将您定向到列出所有应用程序的 Argo CD **Applications** 页面。Argo CD **Applications** 页面带有 **LABELS**（如 **app.kubernetes.io/name=appName**），它只帮助您过滤您选择的应用程序。GITOPS-544

4.1.4.4. 已知问题

Red Hat OpenShift GitOps 1.1 中已知的问题：

- Red Hat OpenShift GitOps 不支持 Helm v2 和 ksonnet。

- 在断开连接的集群中不支持 Red Hat SSO（RH SSO）Operator。因此，断开连接的集群中不支持 Red Hat OpenShift GitOps Operator 和 RH SSO 集成。

- 当您从 OpenShift Container Platform web 控制台删除 Argo CD 应用程序时，Argo CD 应用程序会从用户界面中删除，但部署仍存在于集群中。作为临时解决方案，请从 Argo CD 控制台删除 Argo CD 应用程序。GITOPS-830

4.1.4.5. 有问题的更改

4.1.4.5.1. 从 Red Hat OpenShift GitOps v1.0.1 升级

当您从 Red Hat OpenShift GitOps **v1.0.1** 升级到 **v1.1** 时，Red Hat OpenShift GitOps Operator 会将 **openshift-gitops** 命名空间中创建的默认 Argo CD 实例从 **argocd-cluster** 重命名到 **openshift-gitops**。

这是一个有问题的变化，需要在升级前手动执行以下步骤：

1. 进入 OpenShift Container Platform web 控制台，将 **openshift-gitops** 命名空间中的 **argocd-cm.yml** 配置映射文件的内容复制到本地文件中。内容可能类似以下示例：

   argocd 配置映射 YAML 示例

```
kind: ConfigMap
apiVersion: v1
metadata:
selfLink: /api/v1/namespaces/openshift-gitops/configmaps/argocd-cm
resourceVersion: '112532'
name: argocd-cm
uid: f5226fbc-883d-47db-8b53-b5e363f007af
creationTimestamp: '2021-04-16T19:24:08Z'
managedFields:
...
namespace: openshift-gitops
labels:
  app.kubernetes.io/managed-by: argocd-cluster
  app.kubernetes.io/name: argocd-cm
  app.kubernetes.io/part-of: argocd
data: "" (1)
admin.enabled: 'true'
statusbadge.enabled: 'false'
resource.exclusions: |
```

```
  - apiGroups:
    - tekton.dev
    clusters:
    - '*'
    kinds:
    - TaskRun
    - PipelineRun
ga.trackingid: ''
repositories: |
  - type: git
    url: https://github.com/user-name/argocd-example-apps
ga.anonymizeusers: 'false'
help.chatUrl: ''
url: >-
  https://argocd-cluster-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com    "" 2
help.chatText: ''
kustomize.buildOptions: ''
resource.inclusions: ''
repository.credentials: ''
users.anonymous.enabled: 'false'
configManagementPlugins: ''
application.instanceLabelKey: ''
```

1. 手工恢复 **argocd-cm.yml** 配置映射文件中内容的 **data** 部分。
2. 将配置映射条目中的 URL 值替换为新实例名称 **openshift-gitops**。

2. 删除默认的 **argocd-cluster** 实例。
3. 编辑新的 **argocd-cm.yml** 配置映射文件，手动恢复整个 **data** 部分。
4. 将配置映射条目中的 URL 值替换为新实例名称 **openshift-gitops**。例如，在上例中将 URL 值替换为以下 URL 值：

```
url: >-
  https://openshift-gitops-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com
```

5. 登录到 Argo CD 集群，验证之前的配置是否存在。

4.2. 了解 OPENSHIFT GITOPS

4.2.1. 关于 GitOps

GitOps 是为云原生应用程序实施持续部署的一种声明方式。您可以使用 GitOps 创建可重复进程，用于在多集群 Kubernetes 环境间管理 OpenShift Container Platform 集群和应用程序。GitOps 以快速的速度处理和自动化复杂部署，节省部署和发行周期期间的时间。

GitOps 工作流通过开发、测试、临时和生产环境来推送应用程序。GitOps 部署新应用程序或更新现有应用程序，因此您只需要更新存储库，GitOps 会自动执行所有操作。

GitOps 是一组使用 Git 拉取请求来管理基础架构和应用程序配置的实践。GitOps 中的 Git 存储库是系统和应用程序配置的唯一来源。此 Git 存储库包含指定环境中所需的基础架构声明描述，并包含自动流程，

以使您的环境与上述状态匹配。它还包含该系统的完整状态，以便可查看并可审核更改到系统状态。通过使用 GitOps，您可以处理基础架构和应用程序配置 sprawl 的问题。

GitOps 将基础架构和应用程序定义定义为代码。然后，它会使用此代码来管理多个工作区和集群来简化基础架构和应用程序配置的创建过程。根据代码原则，您可以在 Git 存储库中存储集群和应用程序的配置，然后按照 Git 工作流将这些存储库应用到所选集群中。您可以将在 Git 存储库中开发和维护软件的核心原则应用到创建和管理集群和应用程序配置文件。

4.2.2. 关于 Red Hat OpenShift GitOps

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，ArgoCD 可让您提供全局自定义资源，如用于配置 OpenShift Container Platform 集群的资源。

4.2.2.1. 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
- 从已知状态恢复或重新创建集群
- 对多个 OpenShift Container Platform 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。

4.3. OPENSHIFT GITOPS 入门

Red Hat OpenShift GitOps 使用 Argo CD 管理特定集群范围的资源，包括：平台操作员、可选 Operator Lifecycle Manager（OLM）Operator 和用户管理。

本指南介绍如何将 Red Hat OpenShift GitOps Operator 安装到 OpenShift Container Platform 集群，并登录 Argo CD 实例。

4.3.1. 在 Web 控制台中安装 GitOps Operator

先决条件

- 访问 OpenShift Container Platform Web 控制台。
- 具有 **cluster-admin** 角色的帐户。
- 以管理员身份登录 OpenShift 集群。

警告

如果您已安装 Argo CD Operator 的 Community 版本，请在安装 Red Hat OpenShift GitOps Operator 前删除 Argo CD Community Operator。

流程

1. 打开 Web 控制台的 **Administrator** 视角，并进入左侧菜单中的 **Operators → OperatorHub**。
2. 搜索 **OpenShift GitOps**，单击 **Red Hat OpenShift GitOps** 标题，然后单击 **Install**。
   Red Hat OpenShift GitOps 将安装在集群的所有命名空间中。

安装 Red Hat OpenShift GitOps Operator 后，它会自动设置 **openshift-gitops** 命名空间中的已就绪的 Argo CD 实例，并在控制台工具栏中显示 Argo CD 图标。您可以在项目下为您的应用程序创建后续的 Argo CD 实例。

4.4. 配置 ARGO CD 与应用程序递归同步 GIT 存储库

4.4.1. 通过部署带有集群配置的应用程序来配置 OpenShift 集群

使用 Red Hat OpenShift GitOps，您可以将 Argo CD 配置为将 Git 目录的内容与包含集群自定义配置的应用程序递归同步。

先决条件

- 在集群中安装了 Red Hat OpenShift GitOps。

4.4.1.1. 使用您的 OpenShift 凭证登录到 Argo CD 实例

Red Hat OpenShift GitOps Operator 会自动创建一个可用的 Argo CD 实例，可在 **openshift-gitops** 命名空间中使用。

先决条件

- 在集群中安装了 Red Hat OpenShift GitOps Operator。

流程

1. 在 Web 控制台的 **Administrator** 视角中，导航到 **Operators → Installed Operators**，以验证是否安装了 Red Hat OpenShift GitOps Operator。

2. 导航到  菜单 → **OpenShift GitOps → Cluster Argo CD**。Argo CD UI 的登录页面显示在新窗口中。

3. 获取 Argo CD 实例的密码：

   a. 导航到 web 控制台的 **Developer** 视角。此时会显示可用项目列表。

   b. 导航到 **openshift-gitops** 项目。

   c. 使用左侧导航面板导航到 **Secrets** 页面。

   d. 选择 **openshift-gitops-cluster** 实例来显示密码。

   e. 复制密码。

4. 使用此密码和 **admin** 作为用户名在新窗口中登录到 Argo CD UI。

4.4.1.2. 使用 Argo CD 仪表板创建应用程序

Argo CD 提供了一个仪表板，供您创建应用程序。

此示例工作流逐步指导您完成将 Argo CD 配置为递归将 **cluster** 目录中的内容同步到 **cluster-configs-** 应用程序。目录定义 OpenShift Container Platform Web 控制台集群配置，在 web 控制台的  菜单下添加指向 **Red Hat Developer Blog - Kubernetes**的链接，并在集群中定义命名空间 **spring-petclinic**。

流程

1. 在 Argo CD 控制面板中，单击 **NEW APP** 以添加新 Argo CD 应用。

2. 对于此工作流，使用以下配置创建一个 **cluster-configs** 应用程序：

   应用程序名称
   cluster-configs
   Project（项目）
   default
   同步策略
   Manual
   仓库 URL
   https://github.com/redhat-developer/openshift-gitops-getting-started
   修订
   HEAD
   路径
   集群
   目的地
   https://kubernetes.default.svc
   命名空间
   spring-petclinic
   Directory Recurse

checked

3. 单击 **CREATE** 以创建应用程序。
4. 打开 Web 控制台的 **Administrator** 视角，并导航到左侧菜单中的 **Administration → Namespaces**。
5. 搜索并选择命名空间，然后在 **Label** 字段中输入 **argocd.argoproj.io/managed-by=openshift-gitops**，以便 **openshift-gitops** 命名空间中的 Argo CD 实例可以管理您的命名空间。

4.4.1.3. 使用 oc 工具创建应用程序

您可以使用 **oc** 工具在终端中创建 Argo CD 应用程序。

流程

1. 下载 示例应用程序：

```
$ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
```

2. 创建应用程序：

```
$ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
```

3. 运行 **oc get** 命令以查看所创建的应用程序：

```
$ oc get application -n openshift-gitops
```

4. 在部署应用程序的命名空间中添加标签，以便 **openshift-gitops** 命名空间中的 Argo CD 实例可以管理它：

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

4.4.1.4. 将应用程序与 Git 存储库同步

流程

1. 在 Argo CD 仪表板中，**cluster-configs** Argo CD 应用程序的状态为 **Missing** 和 **OutOfSync**。因为应用程序配置了手动同步策略，所以 Argo CD 不会自动同步。
2. 单击 **cluster-configs** 标题上的 **SYNC**，查看更改，然后单击 **SYNCHRONIZE**。Argo CD 将自动检测 Git 存储库中的任何更改。如果更改了配置，Argo CD 会将 **cluster-configs** 的状态改为 **OutOfSync**。您可以修改 Argo CD 的同步策略，以自动将 Git 存储库中的更改应用到集群。
3. 现在，**cluster-configs** Argo CD 应用程序的状态为 **Healthy** 和 **Synced**。点 **cluster-configs** 标题检查同步资源的详情及其在集群中的状态。
4. 导航到 OpenShift Container Platform Web 控制台并点  以验证现在是否存在到 **Red Hat Developer Blog - Kubernetes** 的链接。
5. 导航到 **Project** 页面并搜索 **spring-petclinic** 命名空间，以验证它是否已添加到集群中。集群配置已成功与集群同步。

4.4.2. 使用 Argo CD 部署 Spring Boot 应用程序

通过 Argo CD，您可以使用 Argo CD 仪表板或使用 **oc** 工具将应用程序部署到 OpenShift 集群。

先决条件

- 在集群中安装了 Red Hat OpenShift GitOps。

4.4.2.1. 使用您的 OpenShift 凭证登录到 Argo CD 实例

Red Hat OpenShift GitOps Operator 会自动创建一个可用的 Argo CD 实例，可在 **openshift-gitops** 命名空间中使用。

先决条件

- 在集群中安装了 Red Hat OpenShift GitOps Operator。

流程

1. 在 Web 控制台的 **Administrator** 视角中，导航到 **Operators → Installed Operators**，以验证是否安装了 Red Hat OpenShift GitOps Operator。
2. 导航到  菜单 → **OpenShift GitOps → Cluster Argo CD**。Argo CD UI 的登录页面显示在新窗口中。
3. 获取 Argo CD 实例的密码：
   a. 导航到 web 控制台的 **Developer** 视角。此时会显示可用项目列表。
   b. 导航到 **openshift-gitops** 项目。
   c. 使用左侧导航面板导航到 **Secrets** 页面。
   d. 选择 **openshift-gitops-cluster** 实例来显示密码。
   e. 复制密码。
4. 使用此密码和 **admin** 作为用户名在新窗口中登录到 Argo CD UI。

4.4.2.2. 使用 Argo CD 仪表板创建应用程序

Argo CD 提供了一个仪表板，供您创建应用程序。

此示例工作流逐步指导您完成将 Argo CD 配置为递归将 **cluster** 目录中的内容同步到 **cluster-configs-** 应用程序。目录定义 OpenShift Container Platform Web 控制台集群配置，在 web 控制台的  菜单下添加指向 **Red Hat Developer Blog - Kubernetes**的链接，并在集群中定义命名空间 **spring-petclinic**。

流程

1. 在 Argo CD 控制面板中，单击 **NEW APP** 以添加新 Argo CD 应用。
2. 对于此工作流，使用以下配置创建一个 **cluster-configs** 应用程序：

应用程序名称

cluster-configs

Project（项目）

default

同步策略

Manual

仓库 URL

https://github.com/redhat-developer/openshift-gitops-getting-started

修订

HEAD

路径

集群

目的地

https://kubernetes.default.svc

命名空间

spring-petclinic

Directory Recurse

checked

3. 对于此工作流，使用以下配置创建一个 spring-petclinic 应用程序：

应用程序名称

spring-petclinic

Project（项目）

default

同步策略

自动

仓库 URL

https://github.com/redhat-developer/openshift-gitops-getting-started

修订

HEAD

路径

app

目的地

https://kubernetes.default.svc

命名空间

spring-petclinic

4. 单击 **CREATE** 以创建应用程序。

5. 打开 Web 控制台的 **Administrator** 视角，并导航到左侧菜单中的 **Administration → Namespaces**。

6. 搜索并选择命名空间，然后在 **Label** 字段中输入 **argocd.argoproj.io/managed-by=openshift-gitops**，以便 **openshift-gitops** 命名空间中的 Argo CD 实例可以管理您的命名空间。

4.4.2.3. 使用 oc 工具创建应用程序

您可以使用 **oc** 工具在终端中创建 Argo CD 应用程序。

流程

1. 下载 示例应用程序：

```
$ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
```

2. 创建应用程序：

```
$ oc create -f openshift-gitops-getting-started/argo/app.yaml
```

```
$ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
```

3. 运行 **oc get** 命令以查看所创建的应用程序：

```
$ oc get application -n openshift-gitops
```

4. 在部署应用程序的命名空间中添加标签，以便 **openshift-gitops** 命名空间中的 Argo CD 实例可以管理它：

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

4.4.2.4. 验证 Argo CD 自助行为

Argo CD 持续监控已部署应用程序的状态，检测 Git 中指定清单和集群中的实时更改之间的差别，然后自动更正它们。这个行为被称为自我管理。

您可以在 Argo CD 中测试并观察自我管理的行为。

先决条件

- 已部署并配置 **app-spring-petclinic** 应用程序示例。

流程

1. 在 Argo CD 仪表板中，验证您的应用程序是否具有 **Synced** 状态。
2. 点 Argo CD 仪表板中的 **app-spring-petclinic** 标题，查看部署到集群中的应用程序资源。
3. 在 OpenShift web 控制台中导航到 **Developer** 视角。
4. 修改 Spring PetClinic 部署，并将更改提交到 Git 仓库的 **app/** 目录。Argo CD 将自动将更改部署到集群。
5. 通过修改集群的部署并将其扩展至两个 pod，同时在 OpenShift web 控制台中查看应用程序，以测试自我管理的行为。
    a. 运行以下命令修改部署：

```
$ oc scale deployment spring-petclinic --replicas 2  -n spring-petclinic
```

b. 在 OpenShift web 控制台中，部署扩展至两个 pod，并立即缩减到一个 pod。Argo CD 检测到 Git 存储库的不同，并在 OpenShift 集群上自动管理应用程序。

6. 在 Argo CD 仪表板中，点击 **app-spring-petclinic** 标题 → **APP DETAILS** → **EVENTS**。The **EVENTS** 选项卡显示以下事件： Argo CD 检测集群中缺少同步部署资源，然后重新同步 Git 存储库进行更正。

4.5. 在 OPENSHIFT 上为 ARGO CD 配置 SSO

安装 Red Hat OpenShift GitOps Operator 后，Argo CD 会自动创建一个具有 **admin** 权限的用户。要管理多个用户，Argo CD 允许集群管理员配置 SSO。

注意

不支持捆绑的 Dex OIDC 供应商。

先决条件

- 在集群中安装了 Red Hat SSO。

4.5.1. 在 Keycloak 创建新客户端

流程

1. 登录您的 Keycloak 服务器，选择要使用的域，导航到 **Clients** 页面，然后单击屏幕右上角的 **Create**。
2. 指定以下值：

客户端 ID

argocd

客户端协议

openid-connect

路由 URL

<your-argo-cd-route-url>

访问类型

confidential

有效的 Redirect URI

<your-argo-cd-route-url>/auth/callback

基本 URL

/applications

3. 点 **Save** 以查看添加到 **Client** 页面中的 **Credentials** 标签页。
4. 从 **Credentials** 选项卡中复制 secret 以进行进一步配置。

4.5.2. 配置组声明

要管理 Argo CD 中的用户，您必须配置可包含在身份验证令牌中的组声明。

流程

1. 在 Keycloak 仪表板中，导航到 **Client Scope**，并使用以下值添加新客户端：

   Name
   groups
   协议
   openid-connect
   显示内容范围
   On
   包含在令牌范围
   On

2. 点 **Save** 并导航到 **groups → Mappers**。

3. 使用以下值添加新令牌映射器：

   Name
   groups
   映射器类型
   Group Membership
   令牌声明名称
   groups
   当客户端请求 **groups** 时，令牌映射器会将 **groups** 声明添加到令牌中。

4. 导航到 **Clients → Client Scopes**，并将客户端配置为提供组范围。在 **Assigned Default Client Scopes** 表中选择选 **groups**，点 **Add selected**。**groups** 范围必须位于 **Available Client Scopes** 表中。

5. 导航到 **Users → Admin → Groups** 并创建一个组 **ArgoCDAdmins**。

4.5.3. 配置 Argo CD OIDC

要配置 Argo CD OpenID Connect（OIDC），您必须生成客户端 secret，对其进行编码并将其添加到自定义资源中。

先决条件

- 获取了客户端 secret。

流程

1. 存储您生成的客户端 secret。

   a. 使用 base64 对客户端 secret 进行编码：

   ```
   $ echo -n '83083958-8ec6-47b0-a411-a8c55381fbd2' | base64
   ```

b. 编辑 secret，将 base64 值添加到 **oidc.keycloak.clientSecret** 键中：

```
$ oc edit secret argocd-secret -n <namespace>
```

secret 的 YAML 示例

```
apiVersion: v1
kind: Secret
metadata:
  name: argocd-secret
data:
  oidc.keycloak.clientSecret:
ODMwODM5NTgtOGVjNi00N2IwLWE0MTEtYThjNTUzODFmYmQy
```

2. 编辑 **argocd** 自定义资源并添加 OIDC 配置来启用 Keycloak 验证：

```
$ oc edit argocd -n <your_namespace>
```

argocd 自定义资源示例

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  creationTimestamp: null
  name: argocd
  namespace: argocd
spec:
  resourceExclusions: |
    - apiGroups:
      - tekton.dev
      clusters:
      - '*'
      kinds:
      - TaskRun
      - PipelineRun
  oidcConfig: |
    name: OpenShift Single Sign-On
    issuer: https://keycloak.example.com/auth/realms/myrealm (1)
    clientID: argocd (2)
    clientSecret: $oidc.keycloak.clientSecret (3)
    requestedScopes: ["openid", "profile", "email", "groups"] (4)
  server:
    route:
      enabled: true
```

1. **issuer** 必须使用正确的域名（在这个示例中为 **myrealm**）。
2. **clientID** 是您在 Keycloak 帐户中配置的客户端 ID。
3. **clientSecret** 指向您在 argocd-secret secret 中创建的正确密钥。
4. 如果没有将其添加到 Default 范围，则 **requestsScopes** 包含组声明。

4.5.4. OpenShift 的 keycloak Identity Brokering

您可以将 Keycloak 实例配置为使用 OpenShift 通过 Identity Brokering 进行身份验证。这允许 OpenShift 集群和 Keycloak 实例间的单点登录（SSO）。

先决条件

- 已安装**jq** CLI 工具。

流程

1. 获取 OpenShift Container Platform API URL：

```
$ curl -s -k -H "Authorization: Bearer $(oc whoami -t)" https://<openshift-user-facing-api-url>/apis/config.openshift.io/v1/infrastructures/cluster | jq ".status.apiServerURL".
```

注意

OpenShift Container Platform API 的地址通常受到 HTTPS 的保护。因此，您必须在容器中配置 X509_CA_BUNDLE，并将其设置为 **/var/run/secrets/kubernetes.io/serviceaccount/ca.crt**。否则 Keycloak 无法与 API 服务器通信。

2. 在 Keycloak 服务器仪表板中，导航到 **Identity Providers** 并选择 **Openshift v4**。指定以下值：

Base Url

OpenShift 4 API URL

客户端 ID

keycloak-broker

Client Secret

您要定义的 secret
现在，您可以使用您的 OpenShift 凭证作为 Identity Broker 通过 Keycloak 登录 Argo CD。

4.5.5. 注册其他 OAuth 客户端

如果需要其他 OAuth 客户端来管理 OpenShift Container Platform 集群的身份验证，则可以注册一个。

流程

- 注册您的客户端：

```
$ oc create -f <(echo '
kind: OAuthClient
apiVersion: oauth.openshift.io/v1
metadata:
 name: keycloak-broker 1
secret: "..." 2
redirectURIs:
- "https://keycloak-keycloak.apps.dev-svc-4.7-
```

```
020201.devcluster.openshift.com/auth/realms/myrealm/broker/openshift-v4/endpoint" 3
grantMethod: prompt 4
')
```

1. OAuth 客户端的名称用作提交对 **<namespace_route>/oauth/authorize** 和 **<namespace_route>/oauth/token** 的请求时的 **client_id** 参数。
2. **secret** 用作提交对 **<namespace_route>/oauth/token** 的请求时的 client_secret 参数。
3. 对 **<namespace_route>/oauth/authorize** 和 **<namespace_route>/oauth/token** 的请求中指定的 **redirect_uri** 参数必须等于 **redirectURIs** 参数值中所列的某一 URI 或以其为前缀。
4. 如果用户没有授予这个客户端的访问权限，**grantMethod** 决定当此客户端请求令牌时要执行的操作。指定 **auto** 可自动批准授权并重试请求，或指定 **prompt** 以提示用户批准或拒绝授权。

4.5.6. 配置组和 Argo CD RBAC

基于角色的访问控制（RBAC）可让您为用户提供相关权限。

先决条件

- 您已在 Keycloak 中创建了 **ArgoCDAdmins** 组。
- 您要授予登录到 Argo CD 的权限的用户。

流程

1. 在 Keycloak 仪表板中导航到 **Users → Groups**。将用户添加到 Keycloak 组 **ArgoCDAdmins** 中。
2. 在 **argocd-rbac** 配置映射中确保 **ArgoCDAdmins** 组具有所需的权限。
   - 编辑配置映射：

     ```
     $ oc edit configmap argocd-rbac-cm -n <namespace>
     ```

     定义 admin 权限的配置映射示例。

     ```
     apiVersion: v1
     kind: ConfigMap
     metadata:
       name: argocd-rbac-cm
     data:
       policy.csv: |
         g, /ArgoCDAdmins, role:admin
     ```

4.5.7. Argo CD 的内置权限

本节列出了授予 ArgoCD 管理特定集群范围资源的权限，其中包括集群操作员、可选 OLM 操作者和用户管理。请注意，ArgoCD 没有被授予 **cluster-admin** 权限。

表 4.4. 为 Argo CD 授予权限

资源组	它为用户或管理员配置的内容
operators.coreos.com	由 OLM 管理的可选 Operator
user.openshift.io、rbac.authorization.k8s.io	组、用户及其权限
config.openshift.io	由 CVO 管理的 control plane Operator，用于配置集群范围的构建配置、registry 配置和调度程序策略
storage.k8s.io	存储
console.openshift.io	控制台自定义

4.6. GITOPS OPERATOR 的大小要求

大小要求页面显示在 OpenShift Container Platform 上安装 Red Hat OpenShift GitOps 的大小要求。它还提供由 GitOps Operator 实例化的默认 ArgoCD 实例的大小详情。

4.6.1. GitOps 的大小要求

Red Hat OpenShift GitOps 是为云原生应用程序实施持续部署的一种声明方法。通过 GitOps，您可以定义并配置应用程序的 CPU 和内存要求。

每次安装 Red Hat OpenShift GitOps Operator 时，命名空间上的资源都会在定义的限制中安装。如果默认安装没有设置任何限制或请求，Operator 会使用配额在命名空间中失败。如果没有足够资源，集群无法调度 ArgoCD 相关 pod。下表列出了默认工作负载的资源请求和限值：

Workload	CPU 请求	CPU 限值	内存请求	内存限值
argocd-application-controller	1	2	1024M	2048M
applicationset-controller	1	2	512M	1024M
argocd-server	0.125	0.5	128M	256M
argocd-repo-server	0.5	1	256M	1024M
argocd-redis	0.25	0.5	128M	256M
argocd-dex	0.25	0.5	128M	256M
HAProxy	0.25	0.5	128M	256M

另外，您还可以在 **oc** 命令中使用 ArgoCD 自定义资源来查看特定并修改它们：

```
oc edit argocd <name of argo cd> -n namespace
```