OpenShift Dedicated 4

使用 BuildConfig 进行构建

包含有关 OpenShift Dedicated 构建的信息

Last Updated: 2024-06-28

OpenShift Dedicated 4 使用 BuildConfig 进行构建

包含有关 OpenShift Dedicated 构建的信息

法律通告

摘要

OpenShift Dedicated 集群的构建。

目录

第 1 章 理解镜像构建

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 Dedicated 通过从构建镜像创建容器并将它们推送到容器镜像 registry 来使用 Kubernetes。

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。

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

1.1.1. Docker 构建

OpenShift Dedicated 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。

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。

第 2 章 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。

2.1. 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 Dedicated 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**，并可以随时对其进行编辑。如果选择稍后手动更改配置，则了解 **BuildConfig** 的组成部分及可用选项可能会有所帮助。

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

BuildConfig 对象定义

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: "ruby-sample-build" 1
spec:
  runPolicy: "Serial" 2
  triggers: 3
    -
      type: "GitHub"
      github:
        secret: "secret101"
    - type: "Generic"
      generic:
        secret: "secret101"
    -
      type: "ImageChange"
  source: 4
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy: 5
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "ruby-20-centos7:latest"
  output: 6
    to:
      kind: "ImageStreamTag"
      name: "origin-ruby-sample:latest"
  postCommit: 7
      script: "bundle exec rake test"
```

1. 此规格会创建一个名为 **ruby-sample-build** 的新 **BuildConfig**。
2. **runPolicy** 字段控制从此构建配置创建的构建能否同时运行。默认值为 **Serial**，即新构建将按顺序运行，而不是同时运行。

3 您可以指定导致创建新构建的触发器的列表。

4 **source** 部分定义构建的来源。源类型决定主要的输入源，可以是 **Git**（指向代码库存储位置）、**Dockerfile**（从内联 Dockerfile 构建）或 **Binary**（接受二进制有效负载）。可以同时拥有多个源。详情请参阅每种源类型的文档。

5 **strategy** 部分描述用于执行构建的构建策略。您可以在此处指定 **Source**、**Docker** 或 **Custom** 策略。本例使用 **ruby-20-centos7** 容器镜像， Source-to-image（S2I）用于应用程序构建。

6 成功构建容器镜像后，它将被推送到 **output** 部分中描述的存储库。

7 **postCommit** 部分定义一个可选构建 hook。

第 3 章 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。

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 Dedicated 中提供源，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。

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。

3.3. 镜像源

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

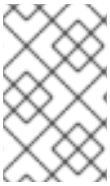
源路径可以是指定镜像内的任何绝对路径。目的地必须是相对目录路径。构建时会加载镜像，并将指定的文件和目录复制到构建过程上下文目录中。这与源存储库内容要克隆到的目录相同。如果源路径以 **/.** 结尾，则复制目录的内容，但不在目的地上创建该目录本身。

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

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
    ref: "master"
  images: 1
  - from: 2
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths: 3
    - destinationDir: injected/dir 4
```

```
      sourcePath: /usr/lib/somefile.jar 5
  - from:
      kind: ImageStreamTag
      name: myotherinputimage:latest
      namespace: myothernamespace
    pullSecret: mysecret 6
    paths:
    - destinationDir: injected/dir
      sourcePath: /usr/lib/somefile.jar
```

1. 由一个或多个输入镜像和文件组成的数组。
2. 对包含要复制的文件的镜像的引用。
3. 源/目标路径的数组。
4. 相对于构建过程能够处理文件的构建根目录的目录。
5. 要从所引用镜像中复制文件的位置。
6. 提供的可选 secret，如需要凭证才能访问输入镜像。



注意

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

需要 pull secret 的镜像

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

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

注意

使用自定义策略的构建不支持此功能。

位于 mirrorr registry 中的需要 pull secret 的镜像

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

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

3.4. GIT 源

指定之后，从提供的位置获取源代码。

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

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

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

1. **git** 字段包含源代码的远程 Git 存储库的 URI (Uniform Resource Identifier)。您必须指定 **ref** 字段的值来签出特定的 Git 引用。有效的 **ref** 可以是 SHA1 标签或分支名称。 **ref** 字段的默认值为 **master**。
2. **contextDir** 字段允许您覆盖源代码存储库中构建查找应用程序源代码的默认位置。如果应用程序位于子目录中，您可以使用此字段覆盖默认位置（根文件夹）。
3. 如果提供可选的 **dockerfile** 字段，它应该是包含 Dockerfile 的字符串，此文件将覆盖源存储库中可能存在的任何 Dockerfile。

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

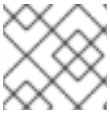
如果没有提供 **ref** 值，OpenShift Dedicated 会执行一个粗略克隆(**--depth=1**)。这时，仅下载与默认分支（通常为 **master**）上最近提交相关联的文件。这将使存储库下载速度加快，但不会有完整的提交历史记录。要执行指定存储库的默认分支的完整 **git clone**，请将 **ref** 设置为默认分支的名称（如 **main**）。

警告

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

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。

3.4.2. 源克隆 secret

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

支持以下源克隆 secret 配置：

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

注意

您还可以组合使用这些配置来满足特定的需求。

3.4.2.1. 自动把源克隆 secret 添加到构建配置

创建 **BuildConfig** 时，OpenShift Dedicated 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 引用的情况下创建的，并且其 Git 源 URI 与 secret 注解中的 URI 模式匹配，OpenShift Dedicated 会在 **BuildConfig** 中插入对该 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 Dedicated 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/*'
```

3.4.2.2. 手动添加源克隆 secret

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

```
apiVersion: "build.openshift.io/v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
```

```
output:
  to:
    kind: "ImageStreamTag"
    name: "sample-image:latest"
source:
  git:
    uri: "https://github.com/user/app.git"
  sourceSecret:
    name: "basicsecret"
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "python-33-centos7:latest"
```

流程

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

- 要在现有构建配置上设置源克隆 secret，请输入以下命令：

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

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
```

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。

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
```

3.4.2.6. 从源代码 SSH 密钥身份验证创建 secret

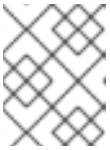
基于 SSH 密钥的身份验证需要 SSH 私钥。

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

为 SSH 密钥创建密码短语可防止 OpenShift Dedicated 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。

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一组传输层安全(TLS)证书颁发机构(CA)内置到 OpenShift Dedicated 基础架构镜像中。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。

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

流程

使用 CA 证书文件创建 secret。

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

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

2. 运行以下命令来创建 secret：

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

1 您必须使用密钥名称 **ca.crt**。

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。

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

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

先决条件

- SSH 身份验证
- **.gitconfig** 文件

流程

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

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

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

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

先决条件

- **.gitconfig** 文件
- CA 证书

流程

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

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

3.4.2.8.3. 使用 CA 证书创建基本身份验证 secret

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

先决条件

- 基本身份验证凭证
- CA 证书

流程

- 要使用 CA 证书创建基本身份验证 secret，请输入以下命令：

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

3.4.2.8.4. 使用 Git 配置文件创建基本身份验证 secret

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

先决条件

- 基本身份验证凭证
- **.gitconfig** 文件

流程

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

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

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

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

先决条件

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

流程

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

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

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。

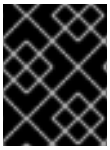
使用 **oc new-build --binary =true** 时，该命令可确保强制执行与二进制构建关联的限制。生成的 **BuildConfig** 具有 **Binary** 源类型，这意味着为此 **BuildConfig** 运行构建的唯一有效方法是使用 **oc start-build** 和其中一个 **--from** 选项来提供必需的二进制数据。

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。

3.6. 输入 SECRET 和配置映射



重要

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

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

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

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

为安全起见，不应在应用程序镜像中公开您的凭证。

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

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 Dedicated 客户端配置文件、**dockercfg** 文件、私有源存储库凭证等。secret 将敏感内容与 Pod 分离。您可以使用卷插件将 secret 信息挂载到容器中，系统也可以使用 secret 代表 Pod 执行操作。

YAML Secret 对象定义

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

1. 指示 secret 的键和值的结构。
2. **data** 字段中允许的键格式必须符合 Kubernetes 标识符术语表中 **DNS_SUBDOMAIN** 值的规范。
3. 与 **data** 映射中键关联的值必须采用 base64 编码。
4. **stringData** 映射中的条目将转换为 base64，然后该条目将自动移动到 **data** 映射中。此字段是只读的。这个值只能由 **data** 字段返回。
5. 与 **stringData** 映射中键关联的值由纯文本字符串组成。

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。

3.6.1.2. secret 的类型

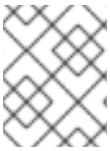
type 字段中的值指明 secret 的键名称和值的结构。此类型可用于强制使 secret 对象中存在用户名和密钥。如果您不想进行验证，请使用 **opaque** 类型，这也是默认类型。

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。

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。

3.6.2. 创建 secret

您必须先创建 secret，然后创建依赖于此 secret 的 Pod。

在创建 secret 时：

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

流程

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

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

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

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

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

YAML Opaque Secret 对象定义

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

1. 指定一个 *opaque* secret。

Docker 配置 JSON 文件对象定义

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

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXIzCg== (2)
```

1. 指定该 secret 使用 docker 配置 JSON 文件。
2. base64 编码的 docker 配置 JSON 文件。

3.6.3. 使用 secret

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

流程

1. 输入以下命令创建 pod 来引用您的 secret：

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

2. 输入以下命令来获取日志：

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

3. 输入以下命令删除 pod：

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

其他资源

- 带有 secret 数据的 YAML 文件示例：

将创建四个文件的 secret 的 YAML 文件

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

1. 文件包含已解码的值。
2. 文件包含已解码的值。
3. 文件包含提供的字符串。
4. 文件包含提供的数据。

pod 的 YAML 文件使用 secret 数据填充卷中的文件

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

pod 的 YAML 文件使用 secret 数据填充环境变量

```
apiVersion: v1
kind: Pod
metadata:
```

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

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

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

3.6.4. 添加输入 secret 和配置映射

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

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

流程

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

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

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

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

提示

您还可以应用以下 YAML 来创建配置映射：

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

2. 如果 **Secret** 对象不存在，请输入以下命令来创建它：

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

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

提示

您还可以应用以下 YAML 来创建输入 secret：

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

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

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

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

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

```
    --build-config-map "settings-mvn"
```

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

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

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

```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn:.ssh” \
    --build-config-map "settings-mvn:.m2"
```

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

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。

3.7. 外部工件 (ARTIFACT)

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

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

.s2i/bin/assemble 文件

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

.s2i/bin/run 文件

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

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

Dockerfile 摘录

```
FROM jboss/base-jdk:8

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

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

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

您可以选择不同方法来定义环境变量：

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

3.8. 将 DOCKER 凭证用于私有容器镜像仓库

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

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

注意

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

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

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

```
    auth: "GzhYWRGU6R2fbclabnRgbkSp=""
    email: "user@example.com"
  docker.io/my-namespace: 5
    auth: "GzhYWRGU6R2deesfrRgbkSp=""
    email: "user@example.com"
```

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

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

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

先决条件

- 您必须有一个 **.docker/config.json** 文件。

流程

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

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

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

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

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

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

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

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

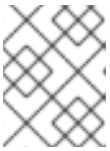
```
$ oc secrets link builder dockerhub
```

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

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

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

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



注意

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

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

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

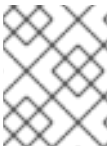
注意

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

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。

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。

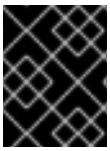
流程

- 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
```

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

3.10. 服务用（SERVICE SERVING）证书 SECRET

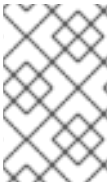
服务用证书 secret 旨在支持需要开箱即用证书的复杂中间件应用程序。它的设置与管理员工具为节点和 master 生成的服务器证书相同。

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。

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。

第 4 章 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。

4.1. 构建输出

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

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

输出到 ImageStreamTag

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

输出到 docker Push 规格

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

4.2. 输出镜像环境变量

Source-to-Image (S2I)策略构建在输出镜像上设置以下环境变量：

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 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。

4.3. 输出镜像标签

Source-to-image (S2I)构建在输出镜像上设置以下标签：

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

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

构建镜像的自定义标签

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "my-image:latest"
    imageLabels:
    - name: "vendor"
      value: "MyCompany"
    - name: "authoritative-source-url"
      value: "registry.mycompany.com"
```

第 5 章 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。

5.1. DOCKER 构建

OpenShift Dedicated 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。

5.1.1. 替换 Dockerfile FROM 镜像

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

流程

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

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

5.1.2. 使用 Dockerfile 路径

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

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

流程

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

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

5.1.3. 使用 Docker 环境变量

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

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

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

例如，定义要在构建和运行时使用的自定义 HTTP 代理：

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

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

5.1.4. 添加 Docker 构建参数

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

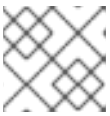
提示

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

流程

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

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



注意

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

5.1.5. 使用 docker 构建的 Squash 层

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

流程

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

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

5.1.6. 使用构建卷

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

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

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

先决条件

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

流程

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

```
spec:
  dockerStrategy:
    volumes:
      - name: secret-mvn (1)
        mounts:
        - destinationPath: /opt/app-root/src/.ssh (2)
        source:
          type: Secret (3)
          secret:
            secretName: my-secret (4)
      - name: settings-mvn (5)
        mounts:
        - destinationPath: /opt/app-root/src/.m2 (6)
        source:
          type: ConfigMap (7)
          configMap:
            name: my-config (8)
```

(1) (5) 必需。唯一的名称。

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

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

(4) (8) 必需。源的名称。

其他资源

- 构建输入
- 输入 secret 和配置映射

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。

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。

5.2.2. 覆盖 source-to-image 构建器镜像脚本

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

流程

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

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

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

注意

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

5.2.3. Source-to-image 环境变量

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

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。

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。

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。

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。

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。

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 5.1. 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。

脚本	描述
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。
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

5.2.6. 使用构建卷

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

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

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

先决条件

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

流程

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

```
spec:
  sourceStrategy:
    volumes:
      - name: secret-mvn (1)
        mounts:
        - destinationPath: /opt/app-root/src/.ssh (2)
        source:
          type: Secret (3)
          secret:
            secretName: my-secret (4)
      - name: settings-mvn (5)
        mounts:
        - destinationPath: /opt/app-root/src/.m2 (6)
        source:
          type: ConfigMap (7)
          configMap:
            name: my-config (8)
```

(1) (5) 必需。唯一的名称。

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

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

(4) (8) 必需。源的名称。

其他资源

- 构建输入
- 输入 secret 和配置映射

5.3. 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

在构建配置中添加 secret，以便它可以从 OpenShift Dedicated Web 控制台访问私有存储库：

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

5.4. 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。

第 6 章 执行和配置基本构建

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

6.1. 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>
```

6.1.1. 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>
```

6.1.2. 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
```

6.1.3. 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：

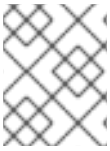
```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

6.1.4. 使用源启动构建

您还可以直接推送源来启动构建，而不依赖于 Git 源拉取(pull)构建，该源可以是 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，或单个文件。这可以通过为 **start-build** 命令指定以下选项之一来完成：

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

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



注意

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

流程

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

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

6.2. 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>
```

6.2.1. 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>
```

6.2.2. 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>
```

6.2.3. 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>
```

6.3. 编辑 BUILDCONFIG

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

您可以使用以下任一视图编辑 **BuildConfig** ：

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

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

流程

1. 在 **Developer 视角** 的 **Builds** 视图中，点击菜单  来查看 **Edit BuildConfig** 选项。
2. 点击 **Edit BuildConfig** 以查看 **Form view** 选项。
3. 在 **Git** 部分中，输入您要用来创建应用程序的代码库的 Git 存储库 URL。这个 URL 随后会被验证。
   - 可选：点击 **Show Advanced Git Options** 来添加详情，例如：
     - **Git Reference**，用于指定包含您要用来构建应用程序的代码的分支、标签或提交。
     - **Context Dir**，用于指定包含您要用来构建应用程序的代码的子目录。
     - **Source Secret**，创建一个具有用来从私有存储库拉取源代码的凭证的 **Secret Name**。
4. 在 **Build from** 部分中，选择您要从中构建的选项。您可以使用以下选项：
   - **镜像流标签** 引用给定镜像流和标签的镜像。输入您要从构建并推送到的位置的项目、镜像流和标签。
   - **镜像流镜像** 引用给定镜像流和镜像名称的镜像。输入您要从中构建的镜像流镜像。另外，进入要推送到的项目、镜像流和标签。

- **Docker 镜像** ：通过 Docker 镜像存储库引用 Docker 镜像。您还需要进入项目、镜像流和标签，以引用您要推送到的位置。

5. 可选：在 **Environment Variables** 部分中，使用 **Name** 和 **Value** 字段添加与项目关联的环境变量。要添加更多环境变量，请使用 **Add Value** 或 **Add from ConfigMap** 和 **Secret**。

6. 可选： 要进一步自定义应用程序，请使用以下高级选项：

   Trigger

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

   Secrets

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

   策略

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

   Hook

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

7. 单击 **Save** 以保存 **BuildConfig**。

6.4. 删除 BUILDCONFIG

您可以使用以下命令来删除 **BuildConfig**。

流程

- 要删除 **BuildConfig**，请输入以下命令：

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

  这也会删除从此 **BuildConfig** 实例化的所有构建。

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

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

6.5. 查看构建详情

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

这会显示，包括：

- 构建源。
- 构建策略。
- 输出目的地。
- 目标 registry 中的镜像摘要。

- 构建的创建方式。

如果构建使用 **Source** 策略，**oc describe** 输出还包括用于构建的源修订的信息，包括提交 ID、作者、提交者和消息。

流程

- 要查看构建详情，请输入以下命令：

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

6.6. 访问构建日志

您可以使用 Web 控制台或 CLI 访问构建日志。

流程

- 要直接使用构建来流传输日志，请输入以下命令：

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

6.6.1. 访问 BuildConfig 日志

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

流程

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

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

6.6.2. 访问给定版本构建的 BuildConfig 日志

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

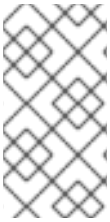
流程

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

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

6.6.3. 启用日志详细程度

您可以通过将 **BUILD_LOGLEVEL** 环境变量作为 **BuildConfig** 中的 **sourceStrategy** 的一部分来启用更详细的输出。



注意

管理员可以通过配置 **env/BUILD_LOGLEVEL**，为整个 OpenShift Dedicated 实例设置默认构建详细程度。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。

第 7 章 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。

7.1. 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

7.1.1. Webhook 触发器

Webhook 触发器允许您将请求发送到 OpenShift Dedicated 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 Dedicated Webhook 仅支持每个基于 Git 的源代码管理(SCM)系统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 Dedicated control plane 主机确认事件内的分支引用是否与对应 **BuildConfig** 中的分支引用匹配。如果是这样，它会检查 OpenShift Dedicated 构建 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
```

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

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

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

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

重要

在修改未经身份验证的访问时，始终验证符合您机构的安全标准。

先决条件

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

流程

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

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

(1) **BuildConfig** 的命名空间。

2. 运行以下命令来应用配置：

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

其他资源

- 未经身份验证的组的集群角色绑定

7.1.1.2. 使用 GitHub Webhook

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

GitHub Webhook 定义示例：

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

注意

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

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

输出示例

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

先决条件

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

流程

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

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

   此命令生成 webhook GitHub URL。

   输出示例

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

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

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

d. 将 URL 输出粘贴到 **Payload URL** 字段。

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

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

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

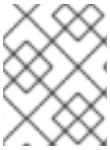
注意

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

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

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

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。



注意

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

其他资源

- Gogs

7.1.1.3. 使用 GitLab Webhook

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

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

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

输出示例

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

先决条件

先决条件

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

流程

1. 配置 GitLab Webhook。
   a. 输入以下命令来获取 Webhook URL：

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

   b. 复制 Webhook URL，将 **<secret>** 替换为您的 secret 值。
   c. 按照 GitLab 设置说明，将 Webhook URL 粘贴到 GitLab 存储库设置中。
2. 假设包含有效 JSON 有效负载的文件，如 **payload.json**，您可以使用以下 **curl** 命令手动触发 Webhook：

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

7.1.1.4. 使用 Bitbucket Webhook

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

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

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

输出示例

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

先决条件

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

流程

1. 配置 Bitbucket Webhook。
   a. 输入以下命令来获取 Webhook URL：

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

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

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

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

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

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

7.1.1.5. 使用通用 Webhook

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

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

1 设置为 **true**，以允许通用 Webhook 传入环境变量。

流程

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

通用 webhook 端点 URL 示例

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

调用者必须以 **POST** 操作的形式调用 Webhook。

2. 要手动调用 webhook，请输入以下 **curl** 命令：

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

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

端点可以接受具有以下格式的可选有效负载：

```
git:
```

```
    uri: "<url to git repository>"
    ref: "<optional git reference>"
    commit: "<commit hash identifying a specific git commit>"
    author:
      name: "<author name>"
      email: "<author e-mail>"
    committer:
      name: "<committer name>"
      email: "<committer e-mail>"
    message: "<commit message>"
  env: 1
    - name: "<variable name>"
      value: "<variable value>"
```

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

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

```
$ curl -H "Content-Type: application/yaml" --data-binary @payload_file.yaml -X POST -k
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/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有效负载，如无效的内容类型、无法解析或无效的内容等，OpenShift Dedicated 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有效负载，OpenShift Dedicated 会返回 JSON 格式的警告，作为其 **HTTP 200 OK** 响应的一部分。

7.1.1.6. 显示 Webhook URL

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

流程

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

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

7.1.2. 使用镜像更改触发器

作为开发人员，您可以将构建配置为在每次基础镜像更改时自动运行。

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

注意

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

流程

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

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

这将定义绑定到位于 ***<system-registry>*/*<namespace>*/ruby-20-centos7** 的容器镜像存储库的镜像流。& **lt;system-registry** > 定义为 OpenShift Dedicated 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
```

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

7.1.3. 识别构建的镜像更改触发器

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

BuildConfig示例

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

# ...

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

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

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

  lastVersion: 1
```

注意

本例省略了与镜像更改触发器无关的元素。

先决条件

- 您已配置了多个镜像更改触发器。这些触发器已触发一个或多个构建。

流程

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

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

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

镜像更改触发器

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

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

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

- **lastTriggeredImageID** ：此字段在 OpenShift Dedicated 4.8 中已弃用，并将在以后的发行版本中被忽略。它包含从此 **BuildConfig** 触发最后一次构建时的 **ImageStreamTag** 的已解析镜像引用。

- **paused**：您可以使用此字段（示例中未显示）暂时禁用此特定镜像更改触发器。

- **from** ：使用此字段引用驱动此镜像更改触发器的 **ImageStreamTag**。其类型是核心 Kubernetes 类型 **OwnerReference**。

from 字段有以下字段：

- **kind** ：对于镜像更改触发器，唯一支持的值是 **ImageStreamTag**。
- **命名空间** ：使用此字段指定 **ImageStreamTag** 的命名空间。
- **名称** ：使用此字段指定 **ImageStreamTag**。

镜像更改触发器状态

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

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

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

其他资源

- v1 容器 registry

7.1.4. 配置更改触发器

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

如下是 **BuildConfig** 中的示例触发器定义 YAML：

```
type: "ConfigChange"
```

注意

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

7.1.4.1. 手动设置触发器

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

流程

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

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

- 要设置镜像更改触发器，请输入以下命令：

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

- 要删除触发器，请输入以下命令：

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

注意

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

如需更多信息，请输入以下命令查阅帮助文档：

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

7.2. 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。

7.2.1. 配置提交后构建 hook

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

流程

- 使用以下选项之一配置构建后 hook：

选项	描述

选项	描述
shell 脚本	``` postCommit: script: "bundle exec rake test --verbose" ``` **script** 值是通过 **/bin/sh -ic** 执行的 shell 脚本。当 shell 脚本适合执行构建 hook 时，请使用此选项。例如，用于运行前文所述的单元测试。要控制镜像入口点，或者镜像没有 **/bin/sh**，**请使用命令**、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。

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

注意

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

7.2.2. 使用 CLI 设置提交后构建 hook

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

流程

1. 完成以下操作之一：
   - 要将命令设置为 post-commit 构建 hook，请输入以下命令：

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

- 要将脚本设置为提交后构建 hook，请输入以下命令：

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

第 8 章 执行高级构建

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

8.1. 设置构建资源

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

流程

您可以以两种方式限制资源使用：

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

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

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

不过，如果您的项目定义了配额，则需要以下两项之一：

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

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

1. **requests** 对象包含与配额中资源列表对应的资源列表。

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

8.2. 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 Dedicated 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
```

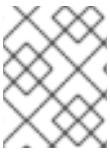
注意

Pipeline 策略选项不支持此设置。

8.3. 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。

8.4. 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。

8.5. 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。

8.6. 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。

第 9 章 在构建中使用红帽订阅

使用以下部分在 OpenShift Dedicated 构建中安装红帽订阅内容。

9.1. 为红帽通用基础镜像创建镜像流标签

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

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

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

流程

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

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

提示

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

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

9.2. 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

- 您必须可以使用具有 **cluster-admin** 角色的用户访问集群，或者具有访问 **openshift-config-managed** 项目中的 secret 的权限。

流程

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

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

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

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

9.3. 使用 SUBSCRIPTION MANAGER 运行构建

9.3.1. 使用 Subscription Manager 执行 Docker 构建

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

先决条件

- 必须将授权密钥添加为构建策略卷。

流程

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

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

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

2. 使用红帽软件包浏览器查找已安装软件包的正确存储库。
3. 您需要恢复 **/etc/rhsm-host** 符号链接，以便使您的镜像与其他红帽容器镜像兼容。

9.4. 使用 RED HAT SATELLITE 订阅运行构建

9.4.1. 将 Red Hat Satellite 配置添加到构建中

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

先决条件

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

  仓库配置示例

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

流程

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

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

2. 将 Satellite 存储库配置和授权密钥添加为构建卷：

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

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

9.4.2. 使用 Red Hat Satellite 订阅构建 Docker

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

先决条件

- 您已将授权密钥和 Satellite 存储库配置添加为构建卷。

流程

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

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

1. 在执行任何 **yum** 或 **dnf** 命令前，您必须包含删除 **/etc/rhsm-host** 目录及其所有内容的命令。
2. 请联系您的 Satellite 系统管理员，以查找构建安装的软件包的正确仓库。
3. 您需要恢复 **/etc/rhsm-host** 符号链接，以便使您的镜像与其他红帽容器镜像兼容。

其他资源

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

9.5. 其他资源

- 管理镜像流
- 构建策略

第 10 章 构建故障排除

使用以下内容来排除构建问题。

10.1. 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
```

10.2. 服务证书生成失败

如果您的资源访问请求遭到拒绝：

问题

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

输出示例

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

解决方案

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

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

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

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

注意

删除注解的命令在要删除的注解名称后面有一个 **-**。