OpenShift Container Platform 4.2

构建（build）

在 OpenShift Container Platform 4.2 中执行构建并与之交互

Last Updated: 2020-08-21

OpenShift Container Platform 4.2 构建（build）

在 OpenShift Container Platform 4.2 中执行构建并与之交互

法律通告

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

摘要

本文档概述了 OpenShift Container Platform 4.2 中的构建和构建配置，并且说明了执行和管理构建的各种方法。

目录

第 1 章 理解镜像构建 **4**
1.1. 构建（BUILD） 4

第 2 章 了解构建配置 **6**
2.1. BUILDCONFIG 6

第 3 章 创建构建输入 **8**
3.1. 构建输入 8
3.2. DOCKERFILE 源 9
3.3. 镜像源 9
3.4. GIT 源 10
3.5. 二进制（本地）来源 19
3.6. 输入 SECRET 和 CONFIGMAP 20
3.7. 外部工件 (ARTIFACT) 22
3.8. 将 DOCKER 凭证用于私有 REGISTRY 23
3.9. 构建环境 25
3.10. 什么是 SECRET？ 26
3.11. 服务用（SERVICE SERVING）证书 SECRET 30
3.12. SECRET 限制 31

第 4 章 管理构建输出 **32**
4.1. 构建输出 32
4.2. 输出镜像环境变量 32
4.3. 输出镜像标签 33

第 5 章 使用构建策略 **34**
5.1. DOCKER 构建 34
5.2. SOURCE-TO-IMAGE (S2I) 构建 35
5.3. CUSTOM 构建 42
5.4. PIPELINE 构建 44
5.5. 使用 WEB 控制台添加 SECRET 51
5.6. 启用拉取 (PULL) 和推送 (PUSH) 52

第 6 章 使用 BUILDAH 自定义镜像构建 **53**
6.1. 创建自定义构建工件 53
6.2. 构建自定义构建器镜像 54
6.3. 使用自定义构建器镜像 54

第 7 章 执行基本构建 **56**
7.1. 启动构建 56
7.2. 取消构建 57
7.3. 删除 BUILDCONFIG 58
7.4. 查看构建详情 58
7.5. 访问构建日志 58

第 8 章 触发和修改构建 **61**
8.1. 构建触发器 61
8.2. 构建 HOOK 69

第 9 章 执行高级构建 **71**
9.1. 设置构建资源 71
9.2. 设置最长持续时间 71
9.3. 将构建分配给特定的节点 72

9.4. 串联构建 72
9.5. 修剪构建 74
9.6. 构建运行策略 74
第 10 章 在构建中使用红帽订阅 76
10.1. 创建 RED HAT UNIVERSAL BASE IMAGE 的 IMAGESTREAMTAG 76
10.2. 将订阅权利添加为构建 SECRET 76
10.3. 使用 SUBSCRIPTION MANAGER 运行构建 77
10.4. 使用 SATELLITE 订阅运行构建 78
10.5. 对 DOCKER 的构建层进行压缩 79
10.6. 其他资源 79
第 11 章 通过策略保护构建 80
11.1. 在全局范围内禁用构建策略访问 80
11.2. 在全局范围内限制用户使用构建策略 81
11.3. 在项目范围内限制用户使用构建策略 82
第 12 章 构建配置资源 83
12.1. 构建控制器配置参数 83
12.2. 配置构建设置 83
第 13 章 构建故障排除 86
13.1. 解决资源访问遭到拒绝的问题 86
13.2. 服务证书生成失败 86
第 14 章 为构建设置其他可信证书颁发机构 87
14.1. 在集群中添加证书颁发机构 87
14.2. 其他资源 87

第 1 章 理解镜像构建

1.1. 构建（BUILD）

*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 Custom 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 Pipeline 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

Docker 构建策略调用 docker build 命令，它需要一个含有 ***Dockerfile*** 的存储库并且其中包含所有必要的工件，从而能生成可运行的镜像。

1.1.2. Source-to-Image (S2I) 构建

Source-to-Image (S2I) 是一种用于构建可重复生成的 Docker 格式容器镜像的工具。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。

S2I 的优点包括：

镜像灵活性	可以编写 S2I 脚本，将应用程序代码注入到几乎所有现有的 Docker 格式容器镜像，以此利用现有的生态系统。请注意，S2I 目前依靠 **tar** 来注入应用程序源代码，因此镜像需要能够处理 tar 压缩的内容。

速度	使用 S2I 时，汇编过程可以执行大量复杂操作，无需在每一步创建新层，进而能实现快速的流程。此外，可以编写 S2I 脚本来重复利用应用程序镜像的旧版本，而不必在每次运行构建时下载或构建它们。
可修补性	如果基础镜像因为安全问题而需要补丁，则 S2I 允许基于新的基础镜像重新构建应用程序。
操作效率	通过限制构建操作而不许随意进行 ***Dockerfile*** 允许的操作，PaaS 运维人员可以避免意外或故意滥用构建系统。
操作安全性	构建任意 ***Dockerfile*** 会将主机系统暴露于 root 特权提升。因为整个 Docker 构建过程都通过具备 Docker 特权的用户运行，这可能被恶意用户利用。S2I 限制以 root 用户执行操作，而能够以非 root 用户运行脚本。
用户效率	S2I 禁止开发人员在应用程序构建期间执行任意 **yum install** 类型的操作。因为这类操作可能会减慢开发迭代速度。
生态系统	S2I 倡导共享镜像生态系统，您可以将其中的最佳实践运用于自己的应用程序。
可重复生成性	生成的镜像可以包含所有输入，包括构建工具和依赖项的特定版本。这可确保精确地重新生成镜像。

1.1.3. Custom 构建

采用 Custom 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（Custom builder）镜像是嵌入了构建过程逻辑的普通 Docker 格式容器镜像，例如用于构建 RPM 或基础镜像。

Custom 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。

1.1.4. Pipeline 构建

采用 Pipeline 构建策略时，开发人员可以定义 *Jenkins Pipeline* 由 Jenkins Pipeline 插件执行。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。

第 2 章 了解构建配置

以下小节定义了构建概念 **BuildConfig**，并概述了可用的主要构建策略。

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。如需详细信息，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。示例中使用了 **ruby-20-centos7** 容器镜像，Source-To-Image 使用该镜像来进行应用程序构建。
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 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，构建将忽略所有驻留在 **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)
  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* 将存储到 *<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* 中将创建含有此内容的 Dockerfile，并覆盖具有此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** 字段包含将要构建的内联 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。也就是说，可以使用容器镜像和镜像流标签（imagestreamtag）进行引用。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。

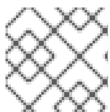
源路径可以是指定镜像内的任何绝对路径。目的地必须是相对目录路径。构建时会加载镜像，并将指定的文件和目录复制到构建过程上下文目录中。这与源存储库内容（若有）要克隆到的目录相同。如果源路径以 /. 结尾，则复制目录的内容，但不在目的地上创建该目录本身。

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

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git
  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

使用 Custom 策略的构建不支持此功能。

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*（若有）。

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**。

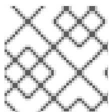
如果未提供 **ref** 值，OpenShift Container Platform 将执行浅克隆 (**--depth=1**)。这时，仅下载与默认分支（通常为 **master**）上最近提交相关联的文件。这将使存储库下载速度加快，但不会有完整的提交历史记录。要对指定存储库的默认分支执行完整 **git clone**，请将 **ref** 设为默认分支（如 **master**）的名称。

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，该操作将不起作用。

3.4.1. 使用代理

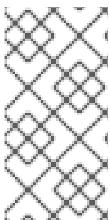
如果 Git 存储库需要使用代理才能访问，您可以在 **BuildConfig** 的 **source** 部分中定义要使用的代理。您可以同时配置要使用的 HTTP 和 HTTPS 代理。两个字段都是可选的。也可以在 **NoProxy** 字段中指定不应执行代理的域。



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"
    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。

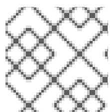
3.4.2. 添加源克隆 secret

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。

先决条件

- 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

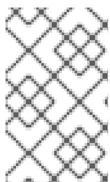
您还可以组合使用这些配置来满足特定的需求。

流程

使用 **builder** 服务帐户运行构建，该帐户必须能够访问所使用的任何源克隆 secret。

- 运行以下命令来授予访问权限：

```
$ oc secrets link builder mysecret
```



注意

默认情况下，“将 secret 仅限于引用它们的服务帐户”的功能被禁用。这意味着，如果在主配置文件中将 **serviceAccountConfig.limitSecretReferences** 设置为 **false**（默认设置），则不需要将 secret 连接到一个特定的服务。

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 **Build** 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**Secret** 必须存在于稍后创建 **BuildConfig** 的命名空间中。此 **Secret** 还必须包含前缀为 **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 URI 模式，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：

- 一个有效的方案（***:**//、**git:**//、**http:**//**https:**// 或 **ssh:**//）。
- 一个主机（*，或一个有效的主机名或 IP 地址（可以在之前使用 ***.**））。
- 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 将选择匹配内容最长的 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** 内的 **source** 部分，并将它设置为您要创建的 **secret** 的名称（本例中为 **basicsecret**），您可以手动将源克隆 secret 添加到构建配置中。

```
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。

- 要在现有构建配置中设置源克隆 secret，请运行：

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

其他资源

- 在 **BuildConfig** 中定义 Secret 提供了有关此主题的更多信息。

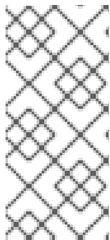
3.4.2.3. 从 .gitconfig 文件创建 secret

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

- 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中的 */var/run/secrets/openshift.io/source/* 文件夹。

2. 在服务器的 *.gitconfig* 文件中，添加下例中所示的 **[http]** 部分：

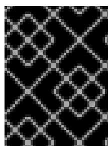
```
# cat .gitconfig
[user]
        name = <name>
        email = <email>
[http]
        sslVerify = false
        sslCert = /var/run/secrets/openshift.io/source/client.crt
        sslKey = /var/run/secrets/openshift.io/source/client.key
        sslCaInfo = /var/run/secrets/openshift.io/source/cacert.crt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

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

为避免必须再次输入密码，请务必在构建中指定 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 **token** 才能与 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. 先创建 secret，再使用用户名和密码访问私有存储库：

```
$ 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

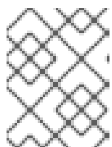
基于 SSH 密钥的身份验证需要 SSH 私钥。

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 rsa -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> \
    --type=kubernetes.io/ssh-auth
```

3.4.2.7. 从源代码可信证书颁发机构创建 secret

git clone 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。

如果为 **CA 证书**创建 secret，OpenShift Container Platform 会在 **git clone** 操作期间使用它来访问您的 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。

a. 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。运行以下命令：

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

b. 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

- 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
```

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>
```

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
```

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
```

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 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

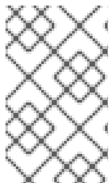
- 使用 *.gitconfig* 文件和 CA 证书创建基本身份验证 secret

```
$ 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
```

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（例如，通过镜像更改触发器），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，还可以指定要提取的存档的 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** 将具有 **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。如果结合使用 **dockerfile** 和 **--from-file** 参数，并且文件参数指定为 **dockerfile**，则 **dockerfile** 的值将取代二进制流中的值。

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 CONFIGMAP

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 ConfigMap* 来实现这一目的。

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. 添加输入 secret 和 ConfigMap

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 ConfigMap* 来实现这一目的。

流程

将输入 secret 和/或 ConfigMap 添加到现有的 **BuildConfig** 中：

1. 如果 ConfigMap 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 ConfigMap，其包含 ***settings.xml*** 文件的纯文本内容。

2. 如果 secret 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 ConfigMap 和 secret 添加到现有 **BuildConfig** 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** 中包含 secret 和 ConfigMap，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** 时的目标目录：

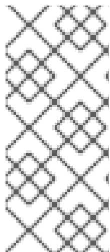
```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn:.ssh” \
    --build-config-map "settings-mvn:.m2"
```

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。

3.6.2. 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。

destinationDir 是相对路径时采用相同的规则；secret 将放置到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 ConfigMap。

3.6.3. 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 将复制到相对于 *Dockerfile* 所在位置的这个目录中。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。

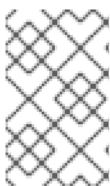
引用 secret 和 ConfigMap 数据的 Dockerfile 示例

```
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

用户通常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。这一移除应该是 *Dockerfile* 本身的一部分。

3.6.4. Custom 策略

使用 **Custom** 策略时，所有定义的输入 secret 和 ConfigMap 都位于 */var/run/secrets/openshift.io/build* 目录下的构建器容器内。自定义构建镜像负责适当地使用这些 secret 和 ConfigMap。**Custom** 策略也允许按照 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）。

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，您可能会发现有必要定义一个构建，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" ]
```

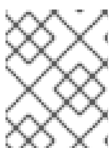
在实践中，您可能希望将环境变量用于文件位置，以便要下载的具体文件能够使用 **BuildConfig** 中定义的环境变量来自定义，而不必更新 *Dockerfile* 或 *assemble* 脚本。

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）

3.8. 将 DOCKER 凭证用于私有 REGISTRY

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

- *.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. 创建 secret 后，将其添加到 builder 服务帐户。所有构建都使用 **builder** 角色来运行，因此您必须使用以下命令使其能访问您的 secret：

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

3. 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
```

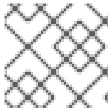
4. 通过指定 **pullSecret** 字段（构建策略定义的一部分），从私有容器镜像 registry 拉取构建器容器镜像：

```
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
```



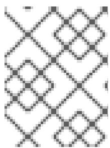
注意

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

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。

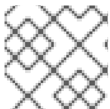


注意

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。

流程

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

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

3.10. 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。此字段是只写的；其值仅通过 **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。

3.10.1. 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。

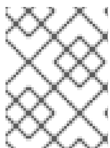
3.10.2. secret 的类型

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**。搭配基本身份验证使用。
- **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

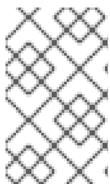
您可以指定其他任意类型，如 **example.com/my-secret-type**。这些类型不在服务器端强制执行，但代表 secret 的创建者意在符合该类型的键/值要求。

3.10.3. 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。若要更改 secret，需要删除原始 Pod 并创建一个新 Pod（可能具有相同的 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。

3.10.4. 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）形式使用 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。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
data:
  username: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

1 指定一个 *opaque* secret。

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

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXlzCg== 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

3.10.4.1. 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。

一个 Pod 的 YAML 定义，使用卷中的 secret 数据。

```
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
```

一个 Pod 的 YAML 定义，在环境变量中使用 secret 数据

```
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: 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
```

3.11. 服务用（SERVICE SERVING）证书 SECRET

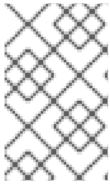
服务用证书 secret 旨在支持需要开箱即用证书的复杂中间件应用程序。它的设置与管理员工具为节点和 master 生成的服务器证书相同。

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。

- 设置服务的 **service.alpha.openshift.io/serving-cert-secret-name** 注解，并将其值设为您要用于 secret 的名称。
  然后，您的 **PodSpec** 可以挂载该 secret。当它可用时，您的 Pod 就可运行。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.alpha.openshift.io/expiry** 注解中查看到期日期，其采用 RFC3339 格式。



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 可以信任集群创建的证书（它仅对内部 DNS 名称进行签名），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 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。

3.12. 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容器。**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

采用 **Docker** 或 **Source-to-Image (S2I)** 策略的构建会创建新的容器镜像。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。如果输出类型是 **DockerImage**，则输出引用的名称将用作 Docker 推送规格。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 推送规范

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

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。

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
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** 字段指定将应用到从 **BuildConfig** 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"
```

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

Docker 构建策略调用 docker build 命令，它需要一个含有 *Dockerfile* 的存储库并且其中包含所有必要的工件，从而能生成可运行的镜像。

5.1.1. 替换 Dockerfile FROM 镜像

您可以将 *Dockerfile* 中的 **FROM** 指令替换为 **BuildConfig** 中的 **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"
```

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的 Dockerfile（名为 *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*），或子目录中 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
```

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 **BuildConfig** 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/"
```

集群管理员还可以使用 Ansible 配置全局构建设置。

您也可以使用 **oc set env** 命令管理 **BuildConfig** 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。

流程

要设置 Docker 构建参数，请在 **BuildArgs** 中添加条目，它位于 **BuildConfig** 的 **dockerStrategy** 定义中。例如：

```
dockerStrategy:
...
  buildArgs:
    - name: "foo"
      value: "bar"
```

5.2. SOURCE-TO-IMAGE (S2I) 构建

Source-to-Image (S2I) 是一种用于构建可重复生成的 Docker 格式容器镜像的工具。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。

S2I 的优点包括：

镜像灵活性	可以编写 S2I 脚本，将应用程序代码注入到几乎所有现有的 Docker 格式容器镜像，以此利用现有的生态系统。请注意，S2I 目前依靠 **tar** 来注入应用程序源代码，因此镜像需要能够处理 tar 压缩的内容。
速度	使用 S2I 时，汇编过程可以执行大量复杂操作，无需在每一步创建新层，进而能实现快速的流程。此外，可以编写 S2I 脚本来重复利用应用程序镜像的旧版本，而不必在每次运行构建时下载或构建它们。
可修补性	如果基础镜像因为安全问题而需要补丁，则 S2I 允许基于新的基础镜像重新构建应用程序。
操作效率	通过限制构建操作而不许随意进行 ***Dockerfile*** 允许的操作，PaaS 运维人员可以避免意外或故意滥用构建系统。
操作安全性	构建任意 ***Dockerfile*** 会将主机系统暴露于 root 特权提升。因为整个 Docker 构建过程都通过具备 Docker 特权的用户运行，这可能被恶意用户利用。S2I 限制以 root 用户执行操作，而能够以非 root 用户运行脚本。

用户效率	S2I 禁止开发人员在应用程序构建期间执行任意 **yum install** 类型的操作。因为这类操作可能会减慢开发迭代速度。
生态系统	S2I 倡导共享镜像生态系统，您可以将其中的最佳实践运用于自己的应用程序。
可重复生成性	生成的镜像可以包含所有输入，包括构建工具和依赖项的特定版本。这可确保精确地重新生成镜像。

5.2.1. 执行 Source-to-Image (S2I) 增量构建

S2I 可以执行增量构建，也就是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 **BuildConfig** 并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 (S2I) 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* S2I 脚本。

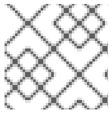
流程

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：

1. 在应用程序源存储库的 *.s2i/bin* 目录中提供 *assemble*、*run* 或 *save-artifacts* 脚本；或者
2. 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。

5.2.3. Source-to-Image (S2I) 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。

5.2.3.1. 使用 Source-to-Image (S2I) 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。

如果在源存储库中提供 *.s2i/environment* 文件，则 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**。

其他资源

- 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 (S2I) BuildConfig 环境

您可以在 **BuildConfig** 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。

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

5.2.4. 忽略 Source-to-Image (S2I) 源文件

Source-to-Image 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。

如需有关 **.s2iignore** 文件格式的更多详细信息，请参阅 Source-to-Image 文档。

5.2.5. 使用 S2I 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，才能让您的镜像提供最佳的 S2I 性能：构建过程和 S2I 脚本。

5.2.5.1. 了解 S2I 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：

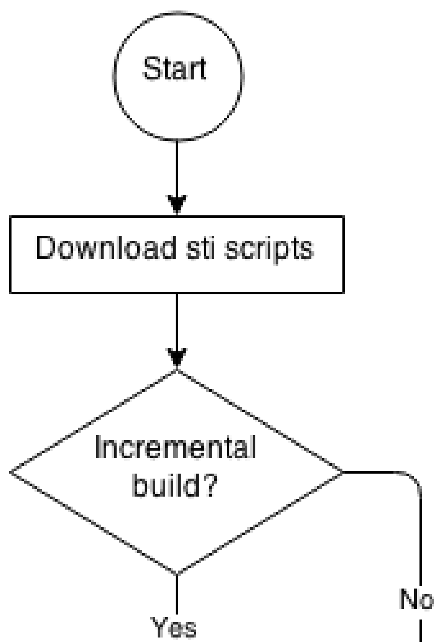
- 源代码
- S2I 脚本
- 构建器镜像

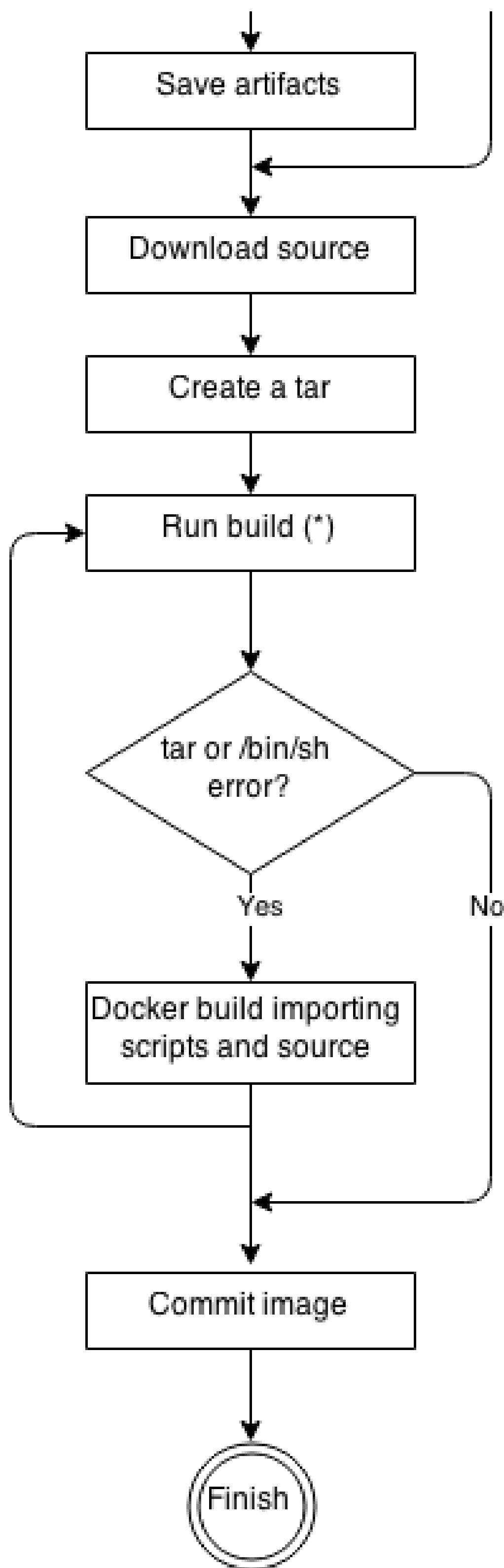
在构建过程中，S2I 必须将源代码和脚本放在构建器镜像中。为此，S2I 创建一个包含源和脚本的 **tar** 文件，然后将该文件流传输到构建器镜像中。在执行 **assemble** 脚本前，S2I 解压缩该文件并将其内容放到构建器镜像的 **io.openshift.s2i.destination** 标签中指定的位置，默认位置为 **/tmp** 目录。

为了使这个过程能够发生，您必须提供 **tar** 存档实用程序（可在 **$PATH** 中使用的 **tar** 命令）和命令行解释器（**/bin/sh** 命令）；这样，您的镜像能够使用最快捷的构建路径。如果 **tar** 或 **/bin/sh** 命令不可用，则强制 **s2i build** 过程自动执行额外容器构建，将源和脚本放进镜像中，然后才运行常规构建。

参见下图中的基本 S2I 构建工作流：

图 5.1. 构建工作流





运行构建的过程包括解压源代码、脚本和工件（若存在），并且调用 **assemble** 脚本。如果这是二次运行（在捕获了"未找到 **tar** 或 **/bin/sh**"错误后），它将仅负责调用 **assemble** 脚本，因为脚本和源代码都已就位。

5.2.5.2. 编写 S2I 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. BuildConfig 中指定的脚本

2. 在应用程序源 **.s2i/bin** 目录中找到的脚本

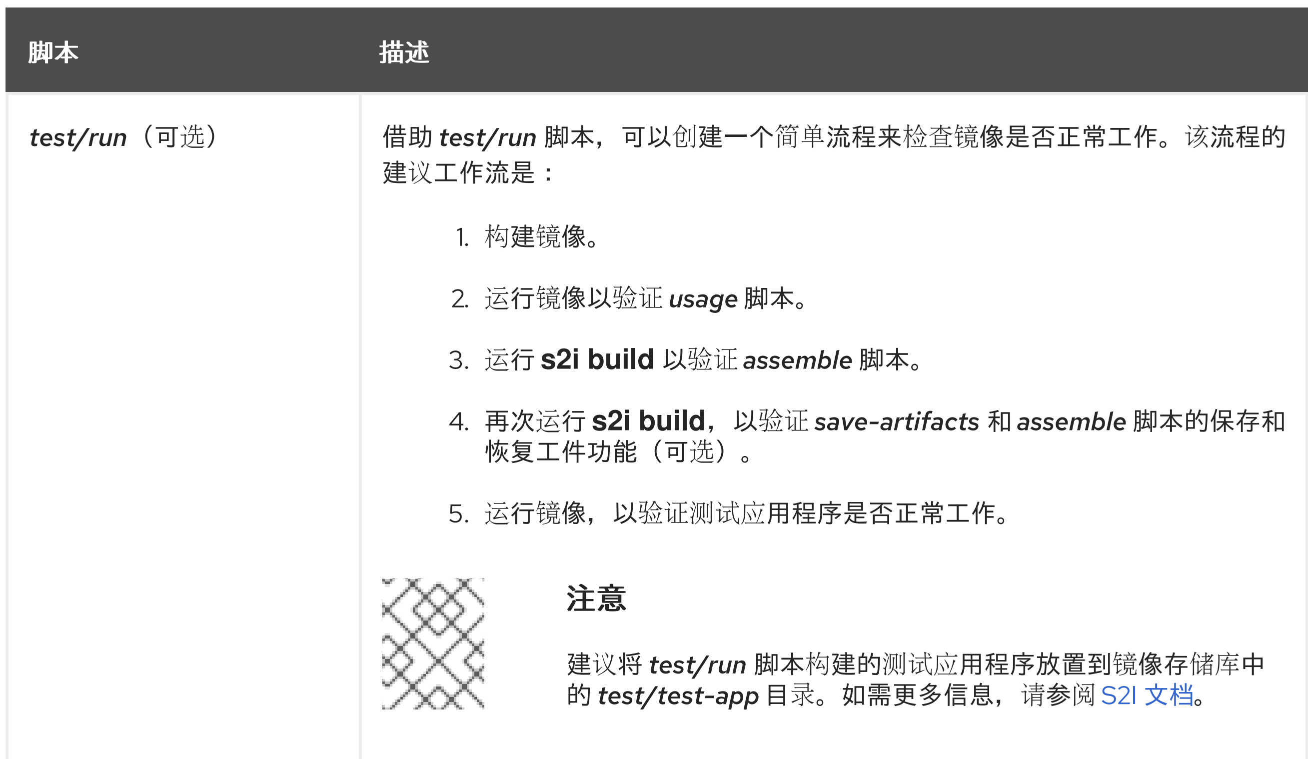
3. 在默认镜像 URL（**io.openshift.s2i.scripts-url** 标签）中找到的脚本

镜像中指定的 **io.openshift.s2i.scripts-url** 标签和 **BuildConfig** 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（必需）	*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。此脚本的工作流为： 1. 恢复构建工件。如果要支持增量构建，请确保同时定义了 *save-artifacts*（可选）。 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（必需）	*run* 脚本将执行您的应用程序。
save-artifacts（可选）	*save-artifacts* 脚本将收集所有可加快后续构建过程的依赖项。例如： • 对于 Ruby，由 Bundler 安装的 **gem**。 • 对于 Java，**.m2** 内容。 这些依赖项收集到一个 tar 文件中，再传输到标准输出。
usage（可选）	借助 *usage* 脚本，可以告知用户如何正确使用您的镜像。



脚本	描述
test/run（可选）	借助 *test/run* 脚本，可以创建一个简单流程来检查镜像是否正常工作。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. 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（可选）。 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。如需更多信息，请参阅 S2I 文档。

5.2.5.2.1. 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 内容解压缩到 **/tmp/s2i** 目录中。

例 5.1. 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
```

例 5.2. run 脚本：

```
#!/bin/bash

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

例 5.3. *save-artifacts* 脚本：

```
#!/bin/bash

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

例 5.4. 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
```

5.3. CUSTOM 构建

采用 Custom 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（Custom builder）镜像是嵌入了构建过程逻辑的普通 Docker 格式容器镜像，例如用于构建 RPM 或基础镜像。

Custom 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。

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"
```

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

将各个 secret 挂载到特定位置：

```
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。

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 Custom 构建过程使用，可在 **BuildConfig** 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

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/"
```

集群管理员还可以使用 Ansible 配置全局构建设置。

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

5.3.4. 使用自定义构建器镜像

OpenShift Container Platform 的 Custom 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，旨在填补因创建容器器镜像日益普及而造成的空缺。如果构建仍然需要生成各种工件（例如，软件包、JAR、WAR、可安装 ZIP 和基础镜像），采用 Custom 构建策略的*自定义构建器镜像*是填补这一空缺的理想选择。

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 或基本容器镜像的逻辑。

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

要充分利用 Custom 构建策略的优势，您必须了解如何创建能够构建所需对象的自定义构建器镜像。

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 5.2. 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** 或 **docker push** 操作的容器 registry 凭证的路径。

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。

5.4. PIPELINE 构建

采用 Pipeline 构建策略时，开发人员可以定义 *Jenkins Pipeline* 由 Jenkins Pipeline 插件执行。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。

5.4.1. 了解 OpenShift Container Platform 管道

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 OpenShift Container Platform 域特定语言 (DSL)（由 Jenkins 客户端插件提供），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 **BuildConfig** 和 Build 对象与 Jenkins 任务和构建保持同步，并提供以下功能：

- Jenkins 中动态创建任务/运行。
- 从 ImageStreams、ImageStreamTag 或 ConfigMap 动态创建 slave Pod 模板。
- 注入环境变量。
- OpenShift web 控制台中的管道视觉化。
- 与 Jenkins Git 插件集成，后者传递提交信息
- 将 secret 同步到 OpenShift 为 Jenkins Git 插件构建的 Jenkins 凭证条目。

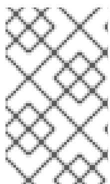
OpenShift Container Platform Jenkins 客户端插件

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。该插件利用了 OpenShift 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：

- **BuildConfig** 中的内联 **jenkinsfile** 字段。
- **BuildConfig** 中的 **jenkinsfilePath** 字段，该字段引用要使用的 **jenkinsfile** 的位置，路径相对于源 **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**。

5.4.2. 为 Pipeline 构建提供 Jenkinsfile

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 slave Pod 上执行，如果您打算使用 OpenShift DSL，它必须具有 OpenShift Client 二进制文件。

流程

若要提供 Jenkinsfile，您可以执行以下操作之一：

1. 在构建配置中嵌入 Jenkinsfile。
2. 在构建配置中包含对含有 Jenkinsfile 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***。

5.4.3. 使用环境变量进行 Pipeline 构建

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

定义之后，环境变量将设置为与 **BuildConfig** 关联的任何 Jenkins 任务的参数。

流程

定义要在构建期间使用的环境变量：

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

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

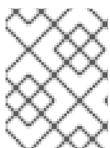
5.4.3.1. BuildConfig 环境变量和 Jenkins 任务参数之间的映射

基于对 Pipeline 策略的 **BuildConfig** 的更改创建或更新 Jenkins 任务时， **BuildConfig** 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相关联的环境变量的当前值。

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 **BuildConfig** 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 **BuildConfig** 中环境变量的值是为对应任务实例设置的参数。您在 Jenkins 控制台中对参数默认值所做的更改都将被忽略。**BuildConfig** 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 **BuildConfig** 中的环境变量，它们会添加为 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。**BuildConfig** 以及您通过 **oc start-build -e** 指定的值将具有优先权。
- 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 **BuildConfig** 中指定与任务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。

5.4.4. Pipeline 构建教程

本例演示如何创建 OpenShift 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

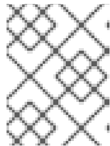
1. 创建 Jenkins master：

```
$ oc project <project_name> (1)
$ oc new-app jenkins-ephemeral (2)
```

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

2. 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：

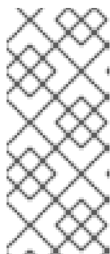


注意

这将创建一个 **BuildConfig**，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** 后，请通过使用内联 **jenkinsfile** 告知管道接下来做什么：



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 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。
2. 要创建的模板的名称。
3. 启动 **node.js** slave Pod，以运行此构建。
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 集群中创建管道 **BuildConfig**：

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

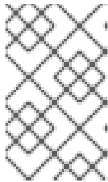
■

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
```

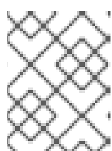


注意

此外，也可以通过 OpenShift 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。
- 启动了 slave Pod（如果管道需要）。
- 管道在 slave Pod 上运行；如果不需要 slave，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**。
- slave Pod（如果管道过去需要）被删除。



注意

视觉化管道执行的最佳方法是在 OpenShift 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。

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

在构建配置中添加 secret 使其能访问私有存储库：

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** 按钮。

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**。

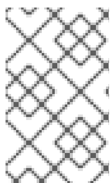
启用推送：

- 在构建配置中设置 **Push Secret**。

第 6 章 使用 BUILDAH 自定义镜像构建

使用 OpenShift Container Platform 4.2 时，主机节点上不会出现 Docker 套接字。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 套接字。

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。

先决条件

- 查看如何授予自定义构建权限。

6.1. 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 docker.io/centos:7
RUN yum install -y buildah
# For simplicity, /tmp/build contains the inputs we'll be building when we
# run this custom builder image. Normally the custom builder image would
# fetch this content from some location at build time. (e.g. via git clone).
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 executed when
# this custom builder image is executed.
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 docker.io/centos:7
RUN touch /tmp/built
```

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
# would be 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 in order 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}
```

6.2. 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**：

```
$ 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。

6.3. 使用自定义构建器镜像

您可以定义一个 **BuildConfig**，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**：

```
kind: BuildConfig
```

```
apiVersion: 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: 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 imagestream**。

第 7 章 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。

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，请运行：

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

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，请运行：

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

7.1.2. 流传输构建日志

您可以指定 **--follow** 标志，在标准输出中流传输构建日志。

流程

要在标准输出中手动流传输构建日志，请运行：

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

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，请运行：

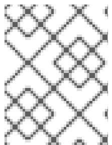
```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

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

例如，以下命令将发送本地 Git 存储库的内容作为标签 **v2** 的存档，再启动构建：

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

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，请运行：

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

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，请运行：

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

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，请运行：

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

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（例如 **new** 或 **pending** 状态），并忽略其他状态下的构建。

流程

要取消给定状态下的所有构建，请运行：

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

7.3. 删除 BUILDCONFIG

您可以使用以下命令来删除 **BuildConfig**。

流程

要删除 **BuildConfig**，请运行：

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

这也会删除从此 **BuildConfig** 实例化的所有构建。如果您不想删除构建，请指定 **--cascade=false** 标志：

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

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。

这将显示诸如以下信息：

- 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，请运行：

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

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，请运行：

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

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

要流传输 BuildConfig 的最新构建的日志，请运行：

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

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。

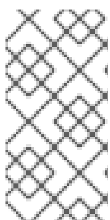
流程

要流传输 BuildConfig 的给定版本构建的日志，请运行：

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

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。

Source 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 8 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。

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：

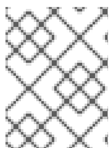
- Webhook
- 镜像更改
- 配置更改

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 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** 将自动创建 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（请参阅"设置触发器"）。

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
```

其他资源

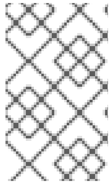
- GitHub
- GitLab
- Bitbucket

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：

```
http://<openshift_api_host:port>/oapi/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：

    ```
    <https://api.starter-us-east-1.openshift.com:443/oapi/v1/namespaces/nsname/buildconfigs/bcname/webhooks/&lt;secret&gt;/github>.
    ```

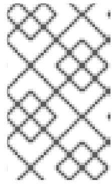
    b. 从 GitHub Web 控制台将此 URL 剪切并粘贴到 GitHub 中。
    c. 在 GitHub 存储库中，从 **Settings → Webhooks** 中选择 **Add Webhook**。

d. 将 URL 输出（与上方相似）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>/oapi/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。

其他资源

- GitHub
- Gogs

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（请参阅"显示 Webhook URL"），其结构如下：

```
http://<openshift_api_host:port>/oapi/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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
hooks/<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。

其他资源

- GitLab

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（请参阅"显示 Webhook URL"），其结构如下：

```
http://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<se
cret>/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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
hooks/<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。

其他资源

- Bitbucket

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. 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：

   ```
   http://<openshift_api_host:port>/oapi/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>/oapi/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>"
   ```

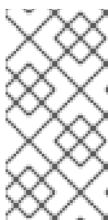
```
env: (1)
  - name: "<variable name>"
    value: "<variable value>"
```

(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>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/web
hooks/<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。

8.1.1.5. 显示 Webhook URL

您可以使用以下命令来显示与 **BuildConfig** 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。请参阅"设置触发器"来手动添加触发器。

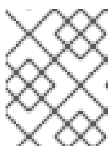
流程

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

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

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 字段设置为指向该镜像流：

```
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. 使用指向镜像流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，策略将使用此新镜像引用。

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

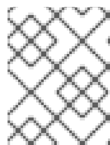
- v1 容器 registry

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。

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：

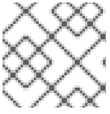
  ```
  $ oc set triggers bc <name> --from-github
  ```

- 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，请使用 **oc set triggers --help** 来查阅帮助文档

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。

8.2.1. 配置提交后构建 hook

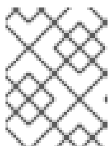
配置提交后构建 hook 的方法有多种。以下示例中所有形式具有同等作用，也都执行 **bundle exec rake test --verbose**。

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：

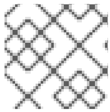
```
postCommit:
  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]
```

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。

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"
```

第 9 章 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。

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

- 通过在项目的默认容器限值中指定资源限值来限制资源。
- 或者，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。

9.2. 设置最长持续时间

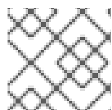
定义 **BuildConfig** 时，您可以通过设置 **completionDeadlineSeconds** 字段来定义其最长持续时间。以秒为单位指定，默认情况下不设置。若未设置，则不强制执行最长持续时间。

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
```



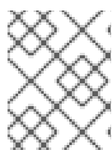
注意

Pipeline 策略选项不支持此设置。

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
```

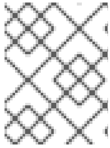
(1) 与此构建配置关联的构建将仅在具有 **key1=value2** 和 **key2=value2** 标签的节点上运行。

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：一个用于生成编译的工件，另一个将该工件放置在运行工件的独立镜像中。

在以下示例中，Source-to-Image 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 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。输出工件的路径将取决于所用 Source-to-Image 构建器的 ***assemble*** 脚本。在本例中，它将输出到 ***/wildfly/standalone/deployments/ROOT.war***。

```
apiVersion: 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
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
```

第二个构建使用了路径指向第一个构建中输入镜像内的 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: 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。

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

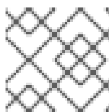
1. 通过为 **BuildConfig** 中的 **successfulBuildsHistoryLimit** 或 **failedBuildsHistoryLimit** 提供正整数值，限制要保留的旧构建的数量，如下例中所示：

```
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。

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。

也可以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。最新的构建接着就会执行。

第 10 章 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。

10.1. 创建 RED HAT UNIVERSAL BASE IMAGE 的 IMAGESTREAMTAG

要在构建中使用红帽订阅，您需要创建一个 **ImageStream** 来引用通用基础镜像 (UBI) 。

直接从 registry.redhat.io 引用 UBI 的构建需要一个 pull secret。

先决条件

- 您必须为 registry.redhat.io 创建一个 pull secret，并将其链接到用户项目。

流程

- 在单个项目中创建 **imagestreamtag**：

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

- 在 OpenShift Container Platform 命名空间中创建 **imagestreamtag** 供所有项目中的开发人员使用：

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

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。

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
```

可以通过两个途径来拉取基础 RHEL 镜像：

- 将 registry.redhat.io 的 pull secret 添加到您的项目中。
- 在 OpenShift 命名空间中为基于 RHEL 的镜像创建镜像流。这样可在整个集群中使用该镜像流。

10.3. 使用 SUBSCRIPTION MANAGER 运行构建

10.3.1. 将 Subscription Manager 配置添加到构建中

使用 Subscription Manager 安装内容的构建必须为订阅的存储库提供适当的配置文件和证书颁发机构。

先决条件

您必须有权访问 Subscription Manager 的配置和证书颁发机构文件。

流程

1. 为 Subscription Manager 配置创建 ConfigMap：

```
$ oc create configmap rhsm-conf --from-file /path/to/rhsm/rhsm.conf
```

2. 为证书颁发机构创建 ConfigMap：

```
$ oc create configmap rhsm-ca --from-file /path/to/rhsm/ca/redhat-uep.pem
```

3. 将 Subscription Manager 配置和证书颁发机构添加到 BuildConfig 中：

```
source:
  configMaps:
  - configMap:
      name: rhsm-conf
    destinationDir: rhsm-conf
  - configMap:
      name: rhsm-ca
    destinationDir: rhsm-ca
```

10.3.2. 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"]
```

10.4. 使用 SATELLITE 订阅运行构建

10.4.1. 将 Satellite 配置添加到构建中

使用 Satellite 安装内容的构建必须提供适当的配置，以从 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。

```
[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. 将 Satellite 存储库配置添加到 BuildConfig 中：

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

10.4.2. 使用 Satellite 订阅进行 Docker 构建

Docker 策略构建可以使用 Satellite 存储库来安装订阅内容。

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 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"]
```

10.5. 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
```

10.6. 其他资源

- 如需更多信息，请参阅管理镜像流。

第 11 章 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，这允许能运行构建的用户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 **cluster-admin** 特权的用户可以启用 Custom 构建策略，如有关在全局范围内限制用户使用构建策略章节中所述。

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。用户必须具有创建构建的权限*以及*创建构建策略子资源的权限，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 11.1. 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

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 **cluster-admin** 特权的用户身份登录，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

apiVersion: v1
groupNames:
- system:authenticated
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: /oapi/v1/clusterrolebindings/system%3Abuild-strategy-docker-binding
```

```
  uid: 17b1f3d4-9c3c-11e8-be62-0800277d20bf
roleRef:
  name: system:build-strategy-docker
subjects:
- kind: SystemGroup
  name: system:authenticated
userNames:
- system:serviceaccount:management-infra:management-admin
```

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，移除与要禁用的策略资源对应的行。

   a. 为 **admin** 禁用 Docker 构建策略：

   ```
   kind: ClusterRole
   metadata:
     name: admin
   ...
   rules:
   - resources:
     - builds/custom
     - builds/docker 1
     - builds/source
     ...
   ...
   ```

   1 删除此行，以在全局范围内禁止具有 **admin** 角色的用户进行 Docker 构建。

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。

11.3. 在项目范围内限制用户使用构建策略

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

先决条件

- 禁用构建策略的全局访问。

流程

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

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

第 12 章 构建配置资源

使用以下步骤来配置构建设置。

12.1. 构建控制器配置参数

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

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

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。

第 13 章 构建故障排除

使用以下内容来排除构建问题。

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
```

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

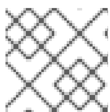
如果服务证书生成失败并显示以下信息（服务的 **service.alpha.openshift.io/serving-cert-generation-error** 注解包含以下信息）：

```
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** 。您必须移除旧 secret 并清除服务注解 **service.alpha.openshift.io/serving-cert-generation-error** 和 **service.alpha.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.alpha.openshift.io/serving-cert-generation-error-
$ oc annotate service <service_name> service.alpha.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 **-**。

第 14 章 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。

14.1. 在集群中添加证书颁发机构

您可以将证书颁发机构 (CA) 添加到集群，以便按照以下方法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，请确保 ConfigMap 中的键是 **hostname[..port]** 格式的 registry 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
```

14.2. 其他资源

- 创建 ConfigMap
- secret 和 ConfigMap
- 配置自定义 PKI