OpenShift Container Platform 3.11

开发人员指南

OpenShift Container Platform 3.11 开发人员参考

Last Updated: 2023-03-25

OpenShift Container Platform 3.11 开发人员指南

OpenShift Container Platform 3.11 开发人员参考

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

法律通告

摘要

这些课题可帮助开发人员设置并配置工作站，以通过命令行界面(CLI)在 OpenShift Container Platform 云环境中开发和部署应用程序。本指南为帮助开发人员提供详细的说明和示例，以帮助开发人员使用 Web 控制台配置并浏览项目，并使用 templatesManage 构建和 webhookDefine 和 trigger deploymentsIntegrate 外部服务（数据库、SaaS 端点）来利用 CLI 生成配置。

目录

第 1 章 概述

本指南面向应用程序开发人员，提供设置和配置工作站在 OpenShift Container Platform 云环境中开发和部署应用程序的说明。这包括帮助开发人员的详细说明和示例：

1. 创建新应用程序
2. 监控并配置项目
3. 使用模板生成配置
4. 管理构建，包括构建策略选项和 Webhook
5. 定义部署，包括部署策略
6. 创建和管理路由
7. 创建和配置 secret
8. 集成外部服务，如数据库和 SaaS 端点
9. 使用探测检查应用程序的健康状态

第 2 章 应用程序生命周期管理

2.1. 规划您的开发流程

2.1.1. 概述

OpenShift Container Platform 专为构建和部署应用程序而设计。根据开发过程中涉及的 OpenShift Container Platform 量，您可以选择：

- 专注于 OpenShift Container Platform 项目中的开发，使用它从头开始构建应用，然后持续开发和管理其生命周期，或者
- 使应用程序（如二进制、容器镜像、源代码）已在单独的环境中开发，并将其部署到 OpenShift Container Platform。

2.1.2. 使用 OpenShift Container Platform 作为您的开发环境

OPENSHIFT_396538_0316

您可以直接使用 OpenShift Container Platform 从头开始开始应用程序的开发。在规划此类型的开发过程中请考虑以下步骤：

初始规划

- 您的应用程序有什么作用？
- 将在什么编程语言中开发？

访问 OpenShift Container Platform

- OpenShift Container Platform 应该由您自己或您所在机构的管理员安装。

开发

- 使用您的编辑器或选择的 IDE，创建一个应用程序的基本框架。它应该足以告知 OpenShift Container Platform 是什么应用程序。
- 将代码推送到您的 Git 存储库。

Generate

- 使用 **oc new-app** 命令创建基本应用程序。OpenShift Container Platform 生成构建和部署配置。

管理

- 开始开发您的应用程序代码。
- 确保您的应用构建成功。
- 继续在本地开发和管理您的代码。

- 将您的代码推送到 Git 存储库。
- 是否需要额外的配置？更多相关信息，请参阅开发人员指南。

验证

- 您可以通过多种方法验证您的应用程序。您可以将更改推送到应用程序的 Git 存储库，并使用 OpenShift Container Platform 重建并重新部署应用程序。另外，您可以使用 **rsync** 热部署将代码更改同步到正在运行的 pod。

2.1.3. 使应用程序能够部署到 OpenShift Container Platform

OPENSHIFT_396538_0316

另一种可能的应用程序开发策略是在本地开发，然后使用 OpenShift Container Platform 部署您完全开发的应用程序。如果计划已有应用程序代码，请使用以下步骤，然后在完成后构建并部署到 OpenShift Container Platform 安装中：

初始规划

- 您的应用程序有什么作用？
- 将在什么编程语言中开发？

开发

- 使用您的编辑器或您选择的 IDE 开发您的应用程序代码。
- 本地构建并测试应用程序代码。
- 将您的代码推送到 Git 存储库。

访问 OpenShift Container Platform

- OpenShift Container Platform 应该由您自己或您所在机构的管理员安装。

Generate

- 使用 **oc new-app** 命令创建基本应用程序。OpenShift Container Platform 生成构建和部署配置。

验证

- 确保已在以上 Generate 步骤中构建和部署的应用程序在 OpenShift Container Platform 上成功运行。

管理

- 继续开发应用程序代码，直到您对此结果很满意。
- 在 OpenShift Container Platform 中重新构建您的应用程序，以接受任何新推送的代码。
- 是否需要额外的配置？更多相关信息，请参阅开发人员指南。

2.2. 创建新应用程序

2.2.1. 概述

您可以使用 OpenShift CLI 或 Web 控制台，从包括源代码或二进制代码、镜像和/或模板在内的组件创建一个新的 OpenShift Container Platform 应用程序。

2.2.2. 使用 CLI 创建应用程序

2.2.2.1. 从源代码创建应用程序

您可以使用 **new-app** 命令从本地或远程 Git 存储库中的源代码创建应用程序。

使用本地目录中的 Git 存储库创建应用程序：

```
$ oc new-app /path/to/source/code
```

注意

如果使用本地 Git 存储库，该存储库应具有一个名为 **origin** 的远程源，指向 OpenShift Container Platform 集群可访问的 URL。如果没有 recognized remote，**new-app** 将创建一个 二进制构建。

使用远程 Git 存储库创建应用程序：

```
$ oc new-app https://github.com/sclorg/cakephp-ex
```

使用私有远程 Git 存储库创建应用程序：

```
$ oc new-app https://github.com/youruser/yourprivaterepo --source-secret=yoursecret
```

注意

如果使用私有远程 Git 存储库，您可以使用 **--source-secret** 标志指定一个现有的 source clone secret，该 secret 将注入到 **BuildConfig** 中以访问存储库。

您可以通过指定 **--context-dir** 标志来使用源代码存储库的子目录。使用远程 Git 存储库和上下文子目录创建应用程序：

```
$ oc new-app https://github.com/sclorg/s2i-ruby-container.git \
    --context-dir=2.0/test/puma-test-app
```

另外，在指定远程 URL 时，您可以通过在 URL 末尾附加 **#<branch_name>** 来指定要使用的 Git 分支：

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

new-app 命令将创建一个构建配置，它本身会从您的源代码中创建一个新的应用程序镜像。**new-app** 命令通常还会创建用于部署新镜像的部署配置，以及为运行您的镜像的部署提供负载均衡访问的服务。

OpenShift Container Platform 会自动 检测 是否应使用 **Docker**、**Pipeline** 或 **Source** 构建策略，如果进行 **Source** 构建，则 检测相应的语言构建器镜像。

构建策略检测

在创建新应用程序时，如果源存储库的根目录或指定上下文目录中存在 ***Jenkinsfile***，则 OpenShift Container Platform 会生成 **Pipeline** 构建策略。否则，如果找到 ***Dockerfile***，OpenShift Container Platform 会生成 **Docker** 构建策略。否则，它会生成 **Source** 构建策略。

您可以通过将 **--strategy** 标志设置为 **docker**、**pipeline** 或 **source** 来覆盖构建策略。

```
$ oc new-app /home/user/code/myapp --strategy=docker
```

注意

oc 命令要求包含构建源的文件在远程 Git 存储库中可用。对于所有 Source 构建，您必须使用 **git remote -v**。

语言检测

如果使用 **Source** 构建策略，**new-app** 会尝试根据存储库根目录或指定上下文目录中是否存在特定文件来确定要使用的语言构建器：

表 2.1. new-app 检测的语言

语言	文件
dotnet	*project.json*, **.csproj*
jee	*pom.xml*
nodejs	*app.json*, *package.json*
perl	*cpanfile*, *index.pl*
php	*composer.json*, *index.php*
python	*requirements.txt*, *setup.py*
ruby	*Gemfile*, *Rakefile*, *config.ru*
scala	*build.sbt*
golang	*Godeps*, *main.go*

检测了语言后，**new-app** 会在 OpenShift Container Platform 服务器上搜索具有与所检测语言匹配的 **支持**注解的镜像流标签，或与所检测语言名称匹配 的镜像流。如果找不到匹配项，**new-app** 会在 Docker Hub registry 中搜索名称上与所检测语言匹配的镜像。

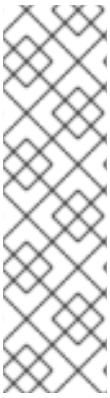
您可以通过指定镜像（镜像流或容器规格）和存储库（以 ~ 作为分隔符），来覆盖构建器用于特定源存储库的镜像。请注意，如果这样做，不会执行 构建策略检测和语言检测。

例如，将 **myproject/my-ruby** 镜像流与远程存储库中的源一起使用：

```
$ oc new-app myproject/my-ruby~https://github.com/openshift/ruby-hello-world.git
```

使用 **openshift/ruby-20-centos7:latest** 容器镜像流以及本地仓库中的源：

```
$ oc new-app openshift/ruby-20-centos7:latest~/home/user/code/my-ruby-app
```



注意

语言检测需要在本地安装 Git 客户端，以便克隆并检查您的存储库。如果 Git 不可用，您可以使用 **<image>~<repository>** 语法指定要与存储库搭配使用的构建器镜像，以避免语言检测步骤。

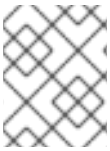
调用 **-i <image> <repository>** 要求 **new-app** 尝试克隆 **repository**，从而能判断其工件类型；如果 Git 不可用，此操作会失败。

使用 **-i <image> --code <repository>** 要求 **new-app** 克隆 **repository**，从而能判断 **image** 应用作源代码的构建器，还是另外部署（使用数据库镜像时）。

2.2.2.2. 从镜像创建应用程序

您可以从现有镜像部署应用程序。镜像可以来自 OpenShift Container Platform 服务器中的镜像流、特定 registry 或 Docker Hub registry 中的镜像，或本地 Docker 服务器中的镜像。

new-app 命令尝试确定传递给它的参数中指定的镜像类型。但是，您可以明确告知 **new-app** 镜像是容器镜像（使用 **--docker-image** 参数）还是镜像流（使用 **-i|--image** 参数）。



注意

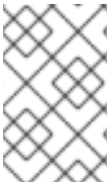
如果指定本地 Docker 存储库中的镜像，必须确保同一镜像可供 OpenShift Container Platform 节点使用。

例如，从 DockerHub MySQL 镜像创建应用程序：

```
$ oc new-app mysql
```

要使用私有 registry 中的镜像创建应用程序，请指定完整容器镜像规格：

```
$ oc new-app myregistry:5000/example/myimage
```



注意

如果包含镜像的 registry 没有使用 SSL 保护，集群管理员必须确保 OpenShift Container Platform 节点主机上的 Docker 守护进程使用指向该 registry 的 **--insecure-registry** 标志运行。您还必须告知 **new-app** 镜像来自带有 **--insecure-registry** 标志的不安全 registry。

您可以从现有镜像流和可选镜像流标签创建应用程序：

```
$ oc new-app my-stream:v1
```

2.2.2.3. 从模板创建应用程序

您可以通过将 模板名称 指定为参数，从之前存储的模板或模板文件创建应用程序。例如，您可以存储一个示例应用程序模板，并使用它来创建应用程序。

从存储的模板创建应用程序：

```
$ oc create -f examples/sample-app/application-template-stibuild.json
$ oc new-app ruby-helloworld-sample
```

要直接使用本地文件系统中的模板，而不先将它保存到 OpenShift Container Platform 中，请使用 **-f|--file** 参数：

```
$ oc new-app -f examples/sample-app/application-template-stibuild.json
```

模板参数

在基于模板创建应用程序时，请使用 **-p|--param** 参数来设置模板定义的参数值：

```
$ oc new-app ruby-helloworld-sample \
    -p ADMIN_USERNAME=admin -p ADMIN_PASSWORD=mypassword
```

您可以将参数保存到文件中，然后在实例化模板时通过 **--param-file** 来使用该文件。如果要从标准输入中读取参数，请使用 **--param-file=-**：

```
$ cat helloworld.params
ADMIN_USERNAME=admin
ADMIN_PASSWORD=mypassword
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
$ cat helloworld.params | oc new-app ruby-helloworld-sample --param-file=-
```

2.2.2.4. 进一步修改应用程序创建

new-app 命令生成将构建、部署和运行正在创建应用程序的 OpenShift Container Platform 对象。通常，这些对象使用从输入源存储库或输入镜像派生的名称在当前项目中创建。但是，**new-app** 允许您修改此行为。

由 **new-app** 创建的对象集合取决于作为输入传递的工件，如输入源存储库、镜像或模板。

表 2.2. new-app 输出对象

对象	描述
BuildConfig	为命令行中指定的每个源存储库创建一个 **BuildConfig**。**BuildConfig** 指定要使用的策略、源位置和构建输出位置。
ImageStreams	对于 **BuildConfig**，通常创建两个 **ImageStreams**。其一代表输入镜像。进行 **Source** 构建时，这是构建器镜像。进行 **Docker** 构建时，这是 FROM 镜像。其二代表输出镜像。如果容器镜像指定为 **new-app** 的输入，那么也会为该镜像创建镜像流。
DeploymentConfig	创建一个 **DeploymentConfig** 来部署构建的输出或指定的镜像。对于生成的 **DeploymentConfig** 中包含的容器，**new-app** 命令为容器中指定的所有 Docker 卷创建 **emptyDir** 卷。

对象	描述
Service	**new-app** 命令会尝试检测输入镜像中公开的端口。它使用编号最小的已公开端口来生成公开该端口的服务。若要公开其他端口，只需在 **new-app** 完成后使用 **oc expose** 命令生成其他服务。
其他	根据模板，可在实例化模板时生成其他对象。

2.2.2.4.1. 指定环境变量

从模板、源 或 镜像 生成应用程序时，您可以在运行时使用 **-e|--env** 参数将环境变量传递给应用程序容器：

```
$ oc new-app openshift/postgresql-92-centos7 \
    -e POSTGRESQL_USER=user \
    -e POSTGRESQL_DATABASE=db \
    -e POSTGRESQL_PASSWORD=password
```

这些变量可使用 **--env-file** 参数从文件中读取：

```
$ cat postgresql.env
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
$ oc new-app openshift/postgresql-92-centos7 --env-file=postgresql.env
```

另外，也可使用 **--env-file=-** 在标准输入上给定环境变量：

```
$ cat postgresql.env | oc new-app openshift/postgresql-92-centos7 --env-file=-
```

如需更多信息，请参阅管理环境变量。

注意

在 **new-app** 处理过程中创建的任何 **BuildConfig** 对象，都不能使用通过 **-e|--env** 或 **--env-file** 参数传递的环境变量进行更新。

2.2.2.4.2. 指定构建环境变量

从模板、源 或 镜像 生成应用程序时，您可以在运行时使用 **--build-env** 参数将环境变量传递给构建容器：

```
$ oc new-app openshift/ruby-23-centos7 \
    --build-env HTTP_PROXY=http://myproxy.net:1337/ \
    --build-env GEM_HOME=~/.gem
```

这些变量可使用 **--build-env-file** 参数从文件中读取：

```
$ cat ruby.env
```

```
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
$ oc new-app openshift/ruby-23-centos7 --build-env-file=ruby.env
```

另外，也可使用 **--build-env-file=-** 在标准输入上给定环境变量：

```
$ cat ruby.env | oc new-app openshift/ruby-23-centos7 --build-env-file=-
```

2.2.2.4.3. 指定标签

从源、镜像或模板生成 应用程序时，您可以使用 **-l|--label** 参数向创建的对象添加标签。借助标签，您可以轻松地集中选择、配置和删除与应用程序关联的对象。

```
$ oc new-app https://github.com/openshift/ruby-hello-world -l name=hello-world
```

2.2.2.4.4. 查看输出（不创建）

要查看要创建的 **new-app** 的空运行，您可以使用 **-o|--output** 参数及 **yaml** 或 **json** 值。然后，您可以使用输出来预览要创建的对象，或者将其重定向到您可以编辑的文件。满意后，您可以使用 **oc create** 创建 OpenShift Container Platform 对象。

将 **new-app** 工件输出到文件中，编辑工件，再创建工件：

```
$ oc new-app https://github.com/openshift/ruby-hello-world \
    -o yaml > myapp.yaml
$ vi myapp.yaml
$ oc create -f myapp.yaml
```

2.2.2.4.5. 使用不同名称创建对象

new-app 创建的对象通常命名自用于生成它们的源存储库或镜像。您可以通过在命令中添加 **--name** 标志来设置生成的对象名称：

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=myapp
```

2.2.2.4.6. 在不同的项目中创建对象

通常，**new-app** 会在当前项目中创建对象。但是，您可以使用 **-n|--namespace** 参数在有权访问的不同项目中创建对象：

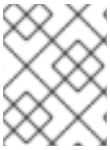
```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

2.2.2.4.7. 创建多个对象

new-app 命令允许创建多个应用程序，为 **new-app** 指定多个参数便可实现。命令行中指定的标签将应用到单一命令创建的所有对象。环境变量应用到从源或镜像创建的所有组件。

从源存储库和 Docker Hub 镜像创建应用程序：

```
$ oc new-app https://github.com/openshift/ruby-hello-world mysql
```



注意

如果以独立参数形式指定源代码存储库和构建器镜像，**new-app** 会将构建器镜像用作源代码存储库的构建器。如果这不是您的用意，请使用 ~ 分隔符为源指定所需的构建器镜像。

2.2.2.4.8. 在单个 Pod 中对镜像和源进行分组

new-app 命令允许在一个 pod 中一起部署多个镜像。要指定哪些镜像要分组在一起，请使用 **+** 分隔符。也可使用 **--group** 命令行参数来指定应分组在一起的镜像。要将源存储库中构建的镜像与其他镜像一起分组，请在组中指定其构建器镜像：

```
$ oc new-app ruby+mysql
```

将通过源构建的镜像和外部镜像一起部署：

```
$ oc new-app \
    ruby~https://github.com/openshift/ruby-hello-world \
    mysql \
    --group=ruby+mysql
```

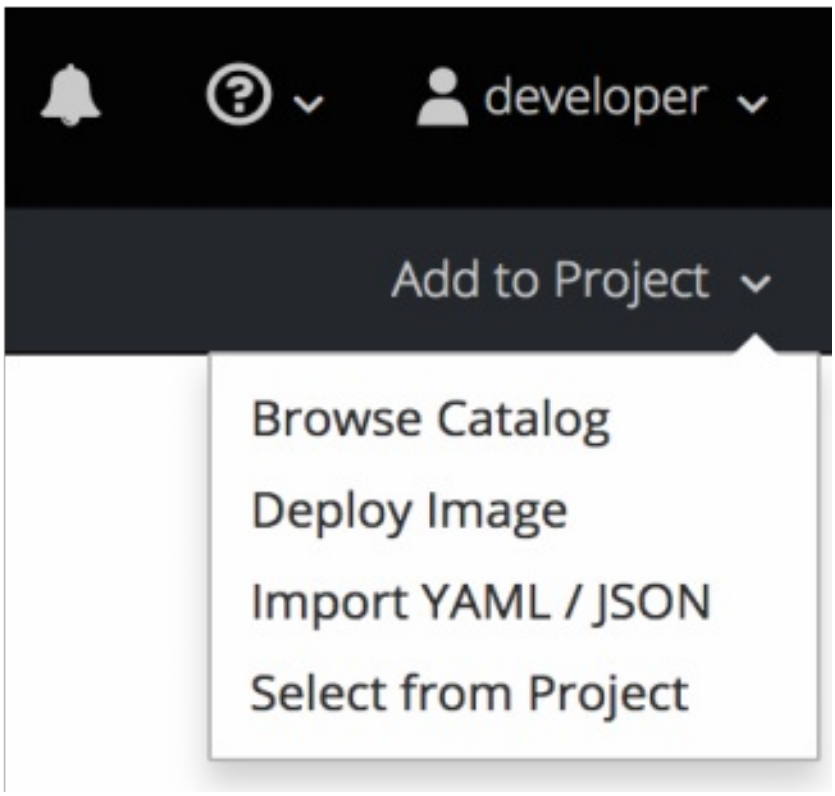
2.2.2.4.9. 搜索镜像、模板和其他输入

要搜索镜像、模板和 **oc new-app** 命令的其他输入，使用 **--search** 和 **--list**。例如，查找包含 PHP 的所有镜像或模板：

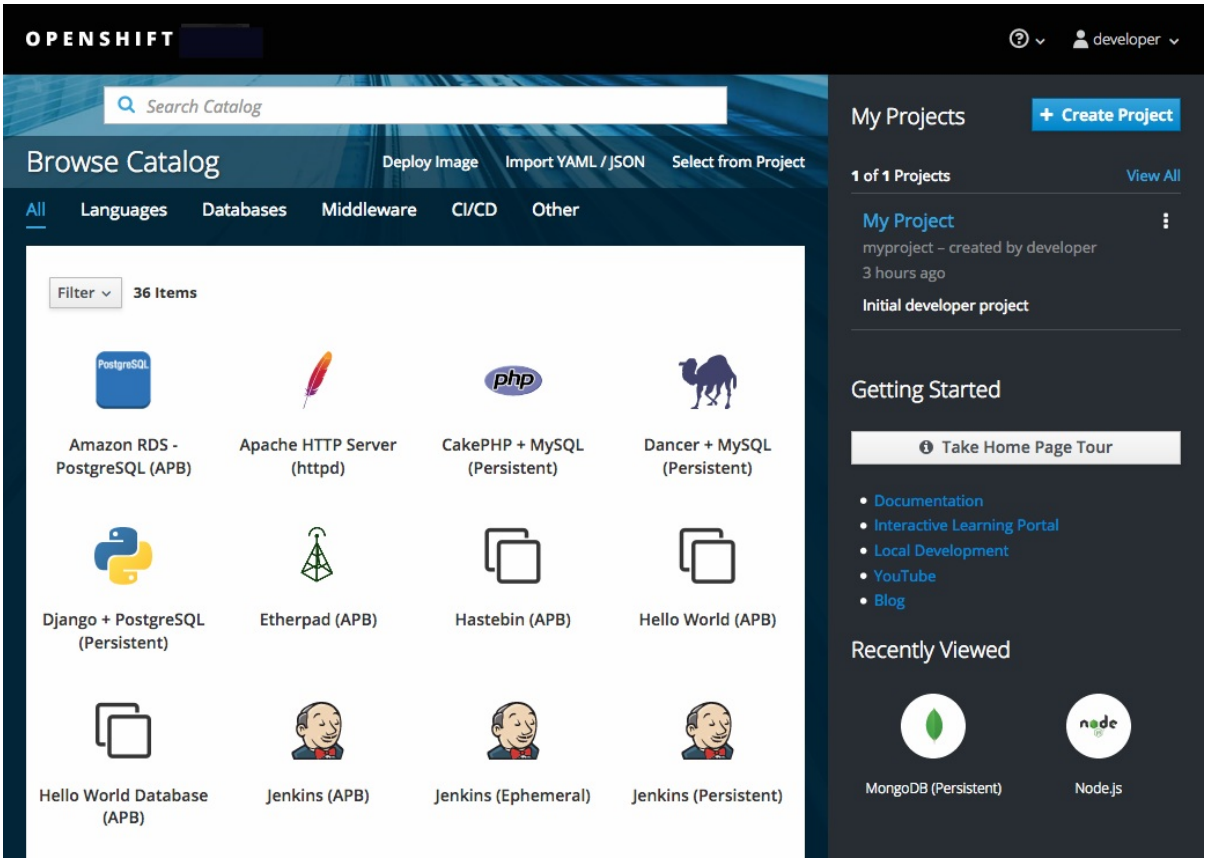
```
$ oc new-app --search php
```

2.2.3. 使用 Web 控制台创建应用程序

1. 在所需项目中，点击 **Add to Project** ：



2. 从项目中的镜像列表或从服务目录中选择构建程序镜像。



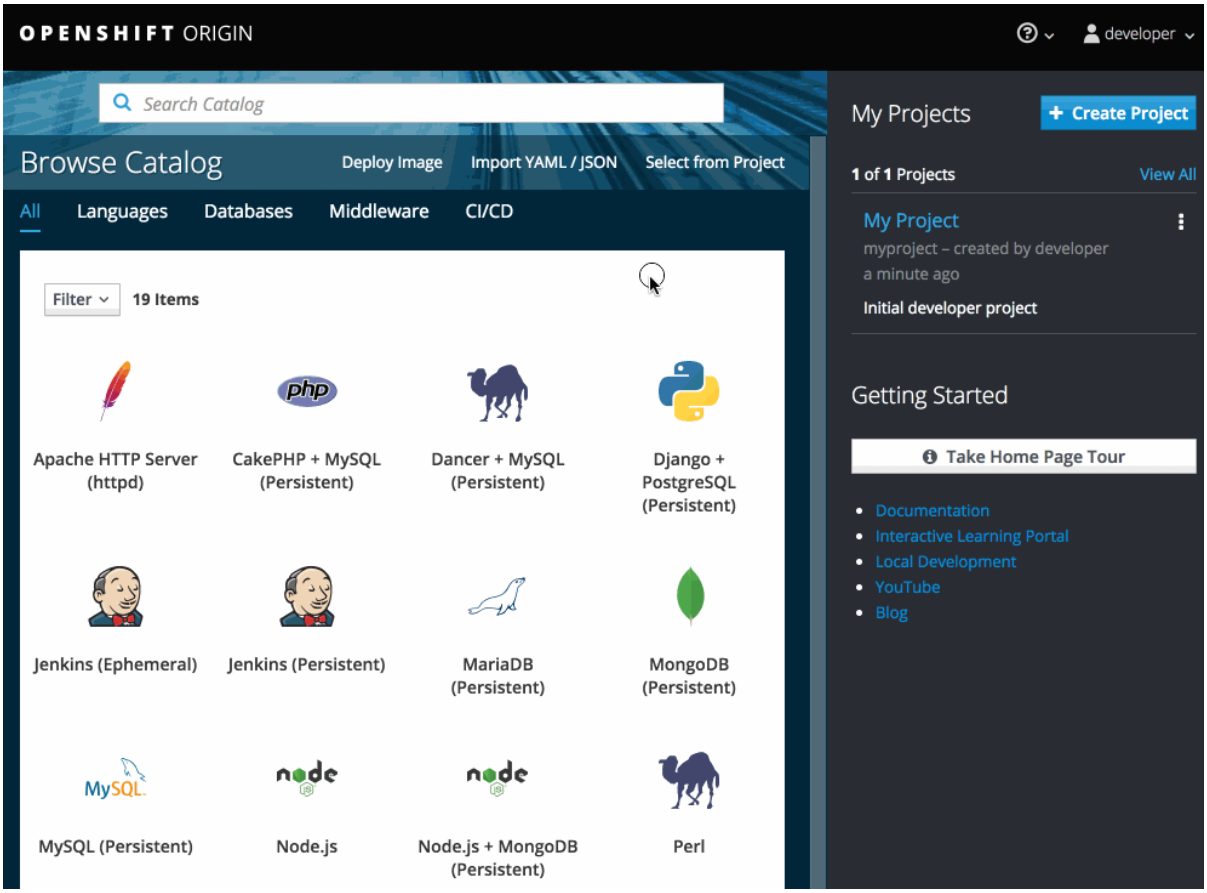
注意

只有其注解中列出 **builder** 标签的 image stream tags 才会出现在此列表中，如下所示：

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository: "registry.redhat.io/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" (1)
        supports: "ruby:2.0,ruby"
        version: "2.0"
```

(1) 此处包含 **builder** 可确保该 **ImageStreamTag** 作为构建程序出现在 web 控制台中。

3. 修改新应用程序屏幕中的设置，以配置对象来支持您的应用程序：



2.3. 在跨环境中提升应用程序

2.3.1. 概述

应用程序提升意味着通过各种运行时环境迁移应用程序，通常具有更高程度的成熟度。例如，应用程序可能会在开发环境中启动，然后提升到暂存环境以进一步进行测试，然后才会出现在生产环境中。随着应用中引入的变化，再次更改将在开发中开始，并在 stage 和生产环境中推广。

当今的"应用程序"不仅仅是使用 Java、Perl 和 Python 等编写的源代码。现在，它比静态 Web 内容、集成脚本或应用特定运行时的相关配置更多。它不仅仅是由那些语言特定运行时使用的应用程序特定存档。

在 OpenShift Container Platform 及其 Kubernetes 和 Docker 的组合部分中，额外的应用程序工件包括：

- 带有大量元数据和相关工具的*容器镜像*。
- 注入容器的*环境变量*以供应用程序使用。
- *API 对象*（也称为资源定义）；请参阅 OpenShift Container Platform 的 核心概念，它：
  - 注入到容器中供应用使用。
  - OpenShift Container Platform 如何管理容器和 pod。

在 OpenShift Container Platform 中如何推广应用程序，这个主题将：

- 详细阐述引入到应用程序定义中的这些新工件。
- 描述您可以为应用程序提升管道分离不同的环境。

- 讨论用于管理这些新工件的方法和工具。
- 提供将各种概念、构建、方法和工具应用到应用程序提升的示例。

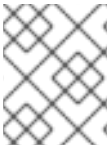
2.3.2. 应用程序组件

2.3.2.1. API 对象

对于 OpenShift Container Platform 和 Kubernetes 资源定义（新向应用程序清单引入的项目），这些 API 对象有几个关键设计点，在考虑应用程序提升主题时，需要重新查看这些 API 对象。

首先，作为在 OpenShift Container Platform 文档中突出显示的，每个 API 对象可以通过 JSON 或 YAML 来表达，从而可以通过传统的源控制和脚本管理这些资源定义。

另外，API 对象也被设计为设计为存在部分对象，用于指定系统所需状态，其他部分则反映了系统的状态或当前状态。这可以被认为为输入和输出。以 JSON 或 YAML 格式表示的输入部分，特别是作为源控制管理(SCM)工件自然而符合的项目。



注意

请记住，API 对象的输入或规格部分可以全面静态或动态，该变量在实例化时 可以通过模板处理替换。

与 API 对象相关的结果是，其表达式用作 JSON 或 YAML 文件，您可以将应用程序的配置视为代码。

可以说，几乎所有 API 对象都可能会被视为您的机构的应用程序工件。下面是与部署和管理应用程序最常见相关的对象：

BuildConfig

在应用程序提升的情况下是一个特殊情况资源。虽然 **BuildConfig** 是应用程序的一部分，特别是开发人员透视图中，但通常 **BuildConfig** 不会通过管道来提升。它会生成通过管道提升（以及其他项目）的**镜像**。

模板

在应用程序提升方面，**模板** 可以充当在给定暂存环境中设置资源的起点，特别是参数化功能。当应用程序通过提升管道进行移动时，额外的后后修改会非常有效。有关此方面的更多信息，请参阅场景和示例。

Routes

这些是应用程序提升管道中阶段阶段的最典型资源，作为通过其 **Route** 对应用程序的不同阶段进行测试。另外，请记住，您拥有与手动规格或自动生成主机名相关的选项，以及 **Route** 的 HTTP 级别的安全性。

服务

对于给定应用程序提升阶段存在避免 **Routers** 和 **Routes** 的原因（例如在开发的早期阶段为开发人员提供简便），可以通过**集群** IP 地址和端口访问应用程序。如果是这种情况，则在不同阶段之间的地址和端口的某些管理可能需要保证。

Endpoints

某些应用程序级别服务（例如，许多企业中的数据库实例）可能无法由 OpenShift Container Platform 管理。如果是这样，则自行创建这些**端点**，以及对相关**服**务（忽略 **Service**上的选择器字段）的修改已启用（基于您的具体环境，在不同阶段之间重复或共享）。

Secrets

当相应实体（由 OpenShift Container Platform 管理的**服**务或 OpenShift Container Platform 之外的外部服务管理的服务）时，由 **Secret** 封装的敏感信息是在暂存环境之间共享的。如果您的应用程序提

升管道的不同阶段有不同版本的实体，则可能需要在管道的每个阶段维护不同的 **Secret**，或者在它遍历管道时对其进行修改。另外，请注意，如果您在 SCM 中将 **Secret** 存储为 JSON 或 YAML，则可能会对一些加密格式进行加密，以保护敏感信息。

DeploymentConfig

此对象是用于定义和缩进给定应用程序提升管道阶段的环境的主要资源；它控制应用程序启动的方式。虽然所有不同阶段都很常见，但为了加快应用程序的提升管道进行，但撤销会对这个对象进行修改，以反映每个阶段的不同环境的不同，或者系统行为的变化，以协助测试应用程序必须支持的不同场景。

ImageStream、ImageStreamTags 和 ImageStreamImage

在 Images 和 Image Streams 部分中详述，这些对象是管理容器镜像的 OpenShift Container Platform 补充。

ServiceAccounts 和 RoleBindings

OpenShift Container Platform 中其他 API 对象的权限管理以及外部服务，用于管理您的应用程序。与 **Secrets** 类似，**ServiceAccounts** 和 **RoleBindings** 对象可能会因您需要共享或隔离这些不同环境的需求在应用程序提升管道的不同阶段之间共享的不同而有所不同。

PersistentVolumeClaims

与数据库等有状态服务相关，不同的应用程序提升阶段将共享量与您的组织共享或隔离应用程序数据副本的关联。

ConfigMaps

从 **Pod** 本身分离 **Pod** 配置的一个实用性（思考环境变量风格配置）可在需要一致的 **Pod** 行为时由各种暂存环境共享。它们也可以在阶段修改，以更改 **Pod** 行为（通常因为应用程序的不同方面被检查到不同的阶段）。

2.3.2.2. 镜像

如前所述，容器镜像现在是应用程序的工件。事实上，新应用工件、镜像和管理是应用程序提升的关键部分。在某些情况下，镜像可能会封装整个应用，应用提升流程则专门管理镜像。

镜像通常不在 SCM 系统中管理，因为应用二进制文件不在以前的系统中。但是，就像使用二进制、可安装工件和相应的存储库（如 RPM、RPM 存储库或 Nexus）一样，与 SCM 相同的语义一样。因此，与 SCM 类似的镜像管理构建和术语发生：

- 镜像 registry == SCM 服务器
- 镜像存储库 == SCM 存储库

当镜像位于 registry 中时，需要确保 registry 中存在适当的镜像，这些镜像可从需要运行该镜像代表的应用程序的环境中访问。

应用程序定义通常抽象到镜像流的引用，而不是直接引用镜像。这意味着镜像流将是组成应用程序组件的另一个 API 对象。有关镜像流的详情，请参阅 核心概念。

2.3.2.3. 概述

现在，在 OpenShift Container Platform 内应用程序推广应用程序工件、镜像和 API 对象的应用程序工件已包括在 OpenShift Container Platform 中，您在提升管道的不同阶段运行应用程序的行为是接下来讨论的。

2.3.3. 部署环境

在这种情况下，部署环境描述了在 CI/CD 管道的特定阶段运行应用程序的不同空间。典型环境包括 **开发**、**测试**、**stage** 和**生产**，例如：环境边界可以以不同的方式定义，例如：

- 通过单一项目中的标签和唯一命名。
- 通过集群中的不同项目。
- 通过不同的集群。

它取决于您的组织是否全部利用了所有三个产品。

2.3.3.1. 注意事项

通常，您会在部署环境结构时考虑以下 heuristics：

- 共享您的提升流程的不同阶段的资源量允许
- 隔离您的提升流程的不同阶段需要多少
- 如何集中位置（或地理位置分散的）促销流程的不同阶段

另外，一些重要的提醒有关 OpenShift Container Platform 集群和项目与镜像 registry 的关系：

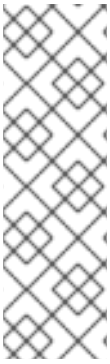
- 同一集群中的多个项目可以访问同一镜像流。
- 多个集群可以访问同一外部 registry。
- 只有 OpenShift Container Platform 内部镜像 registry 通过路由公开时，集群才可以共享 registry。

2.3.3.2. 概述

定义了部署环境后，可以实施管道中的下线阶段的提升流程。构造这些提升流实施的方法和工具是下一个讨论点。

2.3.4. 方法和工具

从根本上而言，应用程序提升是一种将上述应用程序组件从一个环境移至另一个环境的过程。以下小节概述了可用于手动移动各种组件的工具，然后再讨论用于自动化应用程序提升的完整解决方案。



注意

构建和部署过程中都提供了很多插入点。它们在 **BuildConfig** 和 **DeploymentConfig** API 对象中定义。这些 hook 允许调用与部署的组件（如数据库和 OpenShift Container Platform 集群本身）交互的自定义脚本。

因此，可以使用这些 hook 来执行组件管理操作，在环境间有效移动应用程序，例如，从 hook 中执行镜像标签操作。但是，各种 hook 得分最适合在给定环境中管理应用程序的生命周期（例如，在部署新版本的应用程序时执行数据库架构迁移），而不是在环境之间移动应用程序组件。

2.3.4.1. 管理 API 对象

在单一环境中定义的资源将导出为 JSON 或 YAML 文件内容，以准备将其导入到新环境中。因此，当您通过应用程序管道提升 API 对象时，作为 JSON 或 YAML 的表达式作为工作单元。**oc** CLI 用于导出和导入此内容。

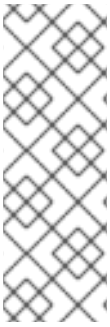
提示

虽然在 OpenShift Container Platform 中不需要提升流，但 JSON 或 YAML 存储在文件中，但您可以考虑从 SCM 系统中存储和检索内容。这可让您利用 SCM 的与版本相关的功能，包括创建分支，以及针对与版本关联的不同标签或标签分配和查询。

2.3.4.1.1. 导出 API 对象状态

API 对象规格应该被 **oc get --export** 捕获。此操作会从对象定义（如当前命名空间或分配的 IP 地址）中移除环境特定数据，允许在不同环境中重新创建它们（与 **oc get** 操作不同，它输出对象的不过滤状态）。

使用 **oc label**，它允许在 API 对象上添加、修改或删除标签，这在组织为提升流程而收集的一组对象时很有用，因为标签允许在单一操作中选择和管理 pod 组。这样可以更轻松地导出正确组的对象，因为标签将在新环境中创建对象时向前工作，所以它们还可以更轻松地管理每个环境中的应用程序组件。



注意

API 对象通常包含引用 **Secret** 的 **DeploymentConfig** 等。当将 API 对象从一个环境迁移到另一个环境时，您必须确保这样的引用也会移到新环境中。

同样，**DeploymentConfig** 等 API 对象通常包含引用外部 registry 的 **ImageStream** 的引用。当将 API 对象从一个环境移到另一个环境时，您必须确保在新环境中可以解析此类引用，这意味着引用必须可以被解析，并且 **ImageStream** 必须引用新环境中可访问的 registry。如需了解更多详细信息，请参阅 Moving Images 和 Promotion Caveats。

2.3.4.1.2. 导入 API 对象状态

2.3.4.1.2.1. 初始创建

当应用程序第一次出现在新环境中时，使用 JSON 或 YAML 来表达 API 对象规格，并运行 **oc create** 在适当的环境中创建它们。使用 **oc create** 时，请记住 **--save-config** 选项。在其注解列表中保存对象的配置元素有助于稍后使用 **oc apply** 修改对象。

2.3.4.1.2.2. 迭代修改

最初建立各种暂存环境后，在提升周期开始并且应用程序从阶段移到 stage 后，对应用程序的更新会包括修改作为应用程序一部分的 API 对象。这些 API 对象中的更改是可激活的，因为它们代表 OpenShift Container Platform 系统的配置。这种变化的动机包括：

- 衡量暂存环境之间的环境差异。
- 验证应用支持的各种场景。

API 对象传输到下一阶段的环境可以通过 **oc** CLI 来完成。虽然存在修改 API 对象的多组 **oc** 命令，但此主题侧重于 **oc apply**，其计算并应用对象之间的差别。

特别是，您可以把 **oc apply** 作为三向合并，在文件或 stdin 中取为输入以及现有的对象定义。它执行三向合并：

1. 命令的输入，
2. 对象的当前版本，以及
3. 最近的用户指定的对象定义存储为当前对象中的注解。

然后，使用结果更新现有对象。

如果需要进一步自定义 API 对象，如在源和目标环境之间对象不能相同时，可以使用 **oc set** 等 **oc** 命令在应用来自上游环境的最新对象定义后修改对象。

某些特定的用途在 Scenarios 和示例中加以控制。

2.3.4.2. 管理镜像和镜像流

OpenShift Container Platform 中的镜像也通过一系列 API 对象进行管理。但是，管理镜像是应用程序提升的核心，讨论这些工具和 API 对象最直接关联镜像保证了单独讨论。存在手动和自动化的形式，可帮助您管理镜像提升（通过管道传播镜像）。

2.3.4.2.1. 移动镜像

注意

有关管理镜像的所有详细信息，请参阅管理镜像主题。

2.3.4.2.1.1. 当暂存环境共享 registry 时

当暂存环境共享相同的 OpenShift Container Platform registry 时，例如它们都在同一个 OpenShift Container Platform 集群中，则有两个操作，代表在应用程序提升管道的不同 stage 间*移动*镜像：

1. 首先，与 **docker tag** 和 **git tag** 类似，**oc tag** 命令允许您更新具有特定镜像的 OpenShift Container Platform 镜像流。它还允许您将引用从一个镜像流的特定版本复制到另一个镜像流，即使在集群中的不同项目也是如此。

2. 其次，**oc import-image** 充当外部 Registry 和镜像流之间的桥接。它从 registry 中导入给定镜像的元数据，并将其作为镜像流标签存储在镜像流中。项目中的各种 **BuildConfigs** 和 **DeploymentConfig** 可以引用这些特定的镜像。

2.3.4.2.1.2. 当 Staging 环境使用不同的 registry 时

当 staging 环境使用不同的 OpenShift Container Platform registry 时，会进行更高级的使用。

访问内部 registry 会详细介绍这些步骤，但总体来说您可以：

1. 使用 **docker** 命令组合获取 OpenShift Container Platform 访问令牌，以提供 **docker login** 命令。

2. 登录 OpenShift Container Platform registry 后，请使用 **docker pull**、**docker tag** 和 **docker push** 来传输镜像。

3. 在管道下一环境的 registry 中提供了镜像后，根据需要使用 **oc tag** 来填充任何镜像流。

2.3.4.2.2. 部署

无论是更改底层应用程序镜像还是配置应用程序的 API 对象，部署通常需要提取升级的更改。如果应用程序的镜像改变（例如，由于 **oc tag** 操作或 **docker push** 作为从上游环境提升镜像的一部分），则您的 **DeploymentConfig** 上的 **ImageChangeTriggers** 可以触发新部署。同样，如果 **DeploymentConfig** API 对象本身被更改，则当由提升步骤更新 API 对象时，**ConfigChangeTrigger** 可以启动部署（如 **oc apply**）。

否则，有助于手动部署的 **oc** 命令包括：

- **oc rollout**:管理部署的新方法，包括暂停和恢复有关管理历史记录的语义和丰富的功能。
- **oc rollback**:允许将版本重新升级到以前的部署；在提升情景中，如果测试新版本遇到问题，请确认它仍能与上一版本合作。

2.3.4.2.3. 使用 Jenkins 自动化促销流

在了解了应用程序的组件后，在提升环境以及移动组件所需的步骤时需要在环境间进行移动后，您可以开始编配和自动化工作流。OpenShift Container Platform 提供了一个 Jenkins 镜像和插件来解决此问题。

使用镜像 (image)中详细介绍了 OpenShift Container Platform Jenkins 镜像，其中包括有助于 Jenkins 集成 Jenkins 和 Jenkins Pipelines 的 OpenShift Container Platform 以插件集。另外，Pipeline 构建策略 有助于集成 Jenkins Pipelines 和 OpenShift Container Platform。所有这些侧重于启用 CI/CD 的各个方面，包括应用程序提升。

在手动执行应用程序提升步骤外，应该考虑 OpenShift Container Platform 提供的与 Jenkins 相关的功能：

- OpenShift Container Platform 提供了一个 Jenkins 镜像，它被定制来大大简化 OpenShift Container Platform 集群中的部署。
- Jenkins 镜像包含 OpenShift Pipeline 插件，它为实施提升工作流提供构建块。这些构建块包括 Jenkins 任务触发作为镜像流更改，以及在这些作业中触发构建和部署。
- 采用 OpenShift Container Platform Jenkins Pipeline 构建策略的 **BuildConfig** 可执行基于 Jenkinsfile 的 Jenkins 管道作业。Pipeline 作业是 Jenkins 中用于复杂提升流程的战略方向，可以利用 OpenShift Pipeline 插件提供的步骤。

2.3.4.2.4. Promotion Caveats

2.3.4.2.4.1. API 对象参考

API 对象可以引用其他对象。这种情况的一个常见用途是具有引用镜像流的 **DeploymentConfig**，但也有其他引用关系。

将 API 对象从一个环境复制到另一个环境时，所有引用仍可以在目标环境中解析。有几个参考场景需要考虑：

- 引用是项目的"本地"。在这种情况下，引用的对象位于与引用它的对象位于同一项目中。通常，要进行的正确操作是，请确保将引用的对象复制到与引用对象相同的项目中的目标环境中。
- 引用是另一个项目中的对象。当共享项目中的镜像流被多个应用项目使用时，这比较典型。在这种情况下，当将引用对象复制到新环境中时，您必须根据需要更新引用，以便在目标环境中解析它。这可能意味着：
  - 如果共享项目在目标环境中具有不同的名称，则更改引用指向的项目。
  - 将引用对象从目标环境中的本地项目移动到本地项目，并将主对象移到目标环境中时，将引用更新为指向本地项目。
  - 将引用对象复制到目标环境中的某些其他组合，并更新对其的引用。

通常，其指导是考虑复制到新环境中的对象引用的对象，并确保引用可在目标环境中解析。如果没有，采取适当的操作来修复引用，并使目标环境中有引用的对象可用。

2.3.4.2.4.2. 镜像 Registry 参考

镜像流指向镜像存储库，以指明它们所代表的镜像源。当镜像流从一个环境移到另一个环境时，务必要考虑 registry 和存储库引用是否还应更改：

- 如果使用不同的镜像 registry 来在测试环境和生产环境之间断言隔离。
- 如果使用不同的镜像存储库来分隔测试和生产就绪的镜像。

如果其中任何一个情况是，在从源环境复制到目标环境时，必须修改镜像流，以便其解析为正确的镜像。这还执行 Scenarios 和 Examples 中描述的步骤，将镜像从一个 registry 和 repository 复制到另一个。

2.3.4.3. 概述

此时定义了以下内容：

- 组成已部署应用程序的新应用程序工件。
- 将应用程序提升活动与 OpenShift Container Platform 提供的工具和概念关联。
- OpenShift Container Platform 和 CI/CD 管道引擎 Jenkins 之间的集成。

将应用程序提升流的示例放在 OpenShift Container Platform 中，是本主题的最后一步。

2.3.5. 场景和示例

在 Docker、Kubernetes 和 OpenShift Container Platform 生态系统中定义了新的应用程序工件组件，本节介绍了如何使用 OpenShift Container Platform 提供的机制和工具在环境之间提升这些组件。

镜像是组成应用程序的组件的主要工件。采用该内部环境并将其扩展至应用程序提升、核心、基本应用程序提升模式是映像提升，其中工作单元是镜像。大多数应用程序促销方案通过提升管道管理和传播镜像。

仅仅通过管道管理和传播镜像的简单场景。随着推广方案的范围广泛，其他应用程序工件（特别是 API 对象）都包含在通过管道进行管理和传播的项目清单中。

本主题介绍了一些有关使用手动和自动化方法来提升镜像以及 API 对象的特定示例。但请注意以下设置应用程序提升管道的环境。

2.3.5.1. 为提升设置

完成应用程序初始修订的开发后，下一步是打包应用程序的内容，以便您可以转移到提升管道的后续暂存环境。

1. 首先，将您查看的所有 API 对象作为传输进行分组，并为它们应用通用标签 ：

   ```
   labels:
     promotion-group: <application_name>
   ```

   如前文所述，**oc label** 命令协助使用各种 API 对象管理标签。

   提示

   如果您最初在 OpenShift Container Platform 模板中定义 API 对象，您可以轻松确保所有相关对象在导出以准备提升时用于查询。

2. 您可以在后续查询中使用该标签。例如，请考虑以下一组 **oc** 命令调用，然后达到应用程序的 API 对象的传输：

```
$ oc login <source_environment>
$ oc project <source_project>
$ oc get -o yaml --export dc,is,svc,route,secret,sa -l promotion-group=<application_name> >
export.yaml
$ oc login <target_environment>
$ oc new-project <target_project> 1
$ oc create -f export.yaml
```

1 或者，如果已存在则为 **oc project <target_project>**。

注意

在 **oc get --export** 命令中，无论您是否包含镜像流的 **is** 类型都取决于您如何选择在管道中不同环境中管理镜像、镜像流和 registry。下文将讨论有关此问题的注意事项。另请参阅 管理镜像 主题。

3. 您还必须获取针对提升管道中不同暂存环境中使用的每个 registry 操作所需的令牌。对于每个环境：

   a. 登录到环境：

   ```
   $ oc login <each_environment_with_a_unique_registry>
   ```

   b. 使用以下命令获取访问令牌：

   ```
   $ oc whoami -t
   ```

   c. 复制并粘贴令牌值供以后使用。

2.3.5.2. 可重复提升流程

在为您的管道进行另一个暂存环境的初始设置后，通过提升管道验证应用程序的每个迭代可以启动一组可重复的步骤。每次源环境中的镜像或 API 对象改变时会执行这些基本步骤：

移动更新的镜像 → Move updated API 对象 → Apply 环境特定自定义

1. 通常，第一步是将与应用程序关联的镜像的任何更新提升到管道中的下一阶段。如上面所述，提升镜像的关键差异化是 OpenShift Container Platform registry 是否在暂存环境间共享。

   a. 如果 registry 共享，则只利用 **oc tag** ：

   ```
   $ oc tag <project_for_stage_N>/<imagestream_name_for_stage_N>:<tag_for_stage_N>
   <project_for_stage_N+1>/<imagestream_name_for_stage_N+1>:<tag_for_stage_N+1>
   ```

   b. 如果没有共享 registry，您可以在登录到源和目标 registry 时利用每个提升管道 registry 的访问令牌，相应地拉取、标记和推送应用程序镜像：

      i. 登录到源环境 registry：

      ```
      $ docker login -u <username> -e <any_email_address> -p <token_value>
      <src_env_registry_ip>:<port>
      ```

      ii. 拉取应用程序的镜像：

```
$ docker pull <src_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

iii. 将应用程序的镜像标记到目标 registry 的位置，根据需要更新命名空间、名称和标签，以符合目标暂存环境：

```
$ docker tag <src_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
<dest_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

iv. 登录到目标暂存环境 registry：

```
$ docker login -u <username> -e <any_email_address> -p <token_value>
<dest_env_registry_ip>:<port>
```

v. 将镜像推送到其目的地：

```
$ docker push <dest_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

提示

要从外部 registry 自动导入镜像的新版本，**oc tag** 命令有一个 **--scheduled** 选项。如果使用，则 **ImageStreamTag** 引用的镜像将定期从托管镜像的 registry 中拉取。

2. 接下来，在有些情况下，应用程序的发展需要对您的 API 对象进行基本更改，或者从组成应用程序的 API 对象集合中移除。当应用程序 API 对象出现这种演变时，OpenShift Container Platform CLI 提供了广泛的选项来从一个暂存环境转移到下一个阶段。

a. 从最初设置提升管道时启动的方式相同：

```
$ oc login <source_environment>
$ oc project <source_project>
$ oc get -o yaml --export dc,is,svc,route,secret,sa -l promotion-group=
<application_name> > export.yaml
$ oc login <target_environment>
$ oc <target_project>
```

b. 更新它们，而不是简单地在新环境中创建资源。您可以按照几种不同的方式实现：

i. 更保守的方法是使用 **oc apply** 并合并对目标环境中每个 API 对象的新更改。要做到这一点，您可以在实际更改对象前 **--dry-run=true** 选项并检查生成的对象：

```
$ oc apply -f export.yaml --dry-run=true
```

如果满意，则实际运行 **apply** 命令：

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

apply 命令可以选择使用额外的参数，以帮助实现更复杂的场景。如需了解更多详细信息，请参阅 **oc apply --help**。

ii. 或者，更简单但更积极的方法是使用 **oc replace**。没有使用此更新且替换的空运行。在最基本的形式中，这涉及执行：

```
$ oc replace -f export.yaml
```

- 与 **apply** 一样，**replace** （可选）为更复杂的行为采用附加参数。如需了解更多详细信息，请参阅 **oc replace --help**。

3. 前面的步骤会自动处理引入的新 API 对象，但如果从源环境中删除 API 对象，必须使用 **oc delete** 从目标环境中手动删除它们。

4. 对于任何 API 对象而言，可能需要调整环境变量，因为这些对象所需的值可能会在暂存环境之间有所不同。为此，请使用 **oc set env** ：

```
$ oc set env <api_object_type>/<api_object_ID> <env_var_name>=<env_var_value>
```

5. 最后，使用 **oc rollout** 命令或上方 Deployments 部分中讨论的其他机制之一来触发更新的应用程序的新部署。

2.3.5.3. 使用 Jenkins 可重复提升过程

用于 OpenShift Container Platform 的 Jenkins Docker 镜像 中定义的 OpenShift Sample 作业是一个在 Jenkins 构造中 OpenShift Container Platform 中的镜像提升示例。本示例的设置位于 OpenShift Origin 源存储库中。

这个示例包括：

- 使用 **Jenkins 作为 CI/CD 引擎**。
- 将 **OpenShift Pipeline 插件用于 Jenkins**。此插件提供了 **oc** CLI 为打包为 Jenkins Freestyle 和 DSL 作业步骤的 oc CLI 提供的功能子集。请注意，**oc** 二进制文件也包含在用于 OpenShift Container Platform 的 Jenkins Docker 镜像中，也可用于与 Jenkins 任务中的 OpenShift Container Platform 交互。
- OpenShift Container Platform 提供的 **Jenkins 的模板**。适用于临时存储和持久存储的模板。
- **示例应用** ：在 OpenShift Origin 源存储库 中定义，此应用利用 **ImageStreams**、**imageChangeTriggers**、**ImageStreamTags**、**BuildConfigs** 和与提升管道中不同阶段对应的 **DeploymentConfig** 和服务。

下面将更加详细地检查 OpenShift Sample 作业的各种部分：

1. 第一步是 等同于 **oc scale dc frontend --replicas=0** 调用。此步骤旨在关闭可能正在运行的应用程序镜像的任何早期版本。
2. 第二个步骤 等同于 **oc start-build frontend** 调用。
3. 第三个步骤 等同于 **oc rollout latest dc/frontend** 调用。
4. 第四个步骤是本例的"测试"步骤。它确保此应用程序的相关服务实际上可从网络角度访问。在覆盖范围内，针对与 OpenShift Container Platform 服务关联的 IP 地址和端口尝试套接字连接。当然，可以添加额外的测试（如果没有通过 OpenShift Pipepline 插件步骤），然后通过使用 Jenkins Shell 步骤来利用操作系统级命令和脚本来测试应用程序。
5. 第五个步骤开始假设测试通过的应用程序，因此希望将镜像标记为"就绪"。在这一步中，会从 **latest** 镜像为应用镜像创建一个新的 **prod** 标签。随着 **frontend DeploymentConfig** 有一个为该标签定义的 **ImageChangeTrigger**，则会启动对应的"production"部署。
6. 第六个和最后一个步骤 是一个验证步骤，插件确认 OpenShift Container Platform 为 "production" 部署启动了所需的副本数。

第 3 章 身份验证

3.1. WEB 控制台身份验证

从位于 **<master_public_addr>:8443** 的浏览器访问 Web 控制台时，您会自动重定向到登录页面。

查看可用于访问 Web 控制台的 浏览器版本和操作系统。

您可以在此页面中提供您的登录凭证，以获取令牌来发出 API 调用。登录后，您可以使用 Web 控制台 导航到项目。

3.2. CLI 身份验证

您可以在命令行中使用 CLI 命令 **oc login** 进行身份验证。您可以在没有选项的情况下运行这个命令来 开始使用 CLI ：

```
$ oc login
```

命令的交互式流可帮助您使用提供的凭证建立到 OpenShift Container Platform 服务器的会话。如果没有提供成功登录到 OpenShift Container Platform 服务器所需的信息，命令会根据需要提示用户输入。配置会自动保存，然后用于后续命令。

oc login --help 中列出的 **oc login** 命令的所有配置选项都是可选的。以下示例显示一些常见选项的使用：

```
$ oc login [-u=<username>] \
  [-p=<password>] \
  [-s=<server>] \
  [-n=<project>] \
  [--certificate-authority=</path/to/file.crt>|--insecure-skip-tls-verify]
```

下表描述了这些通用选项：

表 3.1. 常见 CLI 配置选项

选项	语法	描述
-s, --server	`$ oc login -s=<server>`	指定 OpenShift Container Platform 服务器的主机名。如果通过此标志提供服务器，命令不会以交互方式询问服务器。如果您已有 CLI 配置文件并希望登录并切换到其他服务器，也可以使用此标志。
-u、--username 和 -p, --password	`$ oc login -u=<username> -p=<password>`	允许您指定凭证登录 OpenShift Container Platform 服务器。如果通过这些标志提供用户名和密码，命令不会以互动方式询问。如果您已建立会话令牌的配置文件并希望登录并切换到另一个用户名，也可以使用这些标志。

选项	语法	描述
-n, --namespace	$ oc login -u=<username> -p=<password> -n=<project>	一个全局 CLI 选项，它用于 **oc login** 时，允许您指定在以给定用户身份登录时要切换到的项目。
--certificate-authority	$ oc login --certificate-authority=<path/to/file.crt>	使用 HTTPS 的 OpenShift Container Platform 服务器正确且安全地进行身份验证。必须提供证书颁发机构文件的路径。
--insecure-skip-tls-verify	$ oc login --insecure-skip-tls-verify	允许与 HTTPS 服务器交互绕过服务器证书检查，但请注意，这不太安全。如果您尝试 **oc login** 到不提供有效证书的 HTTPS 服务器，并且不提供 **--certificate-authority** 标志，oc login 将提示用户输入来确认（**y/N** 型输入）与连接不安全。

通过 CLI 配置文件，您可以轻松管理多个 CLI 配置文件。

注意

如果您可以访问管理员凭据，但不再作为默认系统用户 **system:admin** 登录，只要仍存在于 CLI 配置文件 中，您可以随时以这个用户身份登录。以下命令登录并切换到 **default** 项目 ：

```
$ oc login -u system:admin -n default
```

第 4 章 授权

4.1. 概述

本主题包含应用程序开发人员及其功能的授权任务，具体由集群管理员规定。

4.2. 检查用户是否可以创建 POD

使用 **scc-review** 和 **scc-subject-review** 选项，您可以看到单个用户或特定服务帐户下的用户能否创建或更新 pod。

使用 **scc-review** 选项，您可以检查服务帐户是否可以创建或更新 pod。命令输出接受该资源的安全性上下文约束。

例如，要检查具有 **system:serviceaccount:projectname:default** 服务帐户的用户是否可以创建一个 pod：

```
$ oc policy scc-review -z system:serviceaccount:projectname:default -f my_resource.yaml
```

您还可以使用 **scc-subject-review** 选项检查特定用户是否可以创建或更新 pod:

```
$ oc policy scc-subject-review -u <username> -f my_resource.yaml
```

要检查属于特定组的用户是否可以在特定文件中创建 pod：

```
$ oc policy scc-subject-review -u <username> -g <groupname> -f my_resource.yaml
```

4.3. 确定您可以作为经过身份验证的用户执行什么操作

在 OpenShift Container Platform 项目中，您可以决定对所有命名空间范围的资源（包括第三方资源）执行的操作 verbs。

can-i 命令选项测试用户和角色方面的范围。

```
$ oc policy can-i --list --loglevel=8
```

输出可帮助您确定提出哪些 API 请求来收集信息。

要以用户可读格式接收信息，请运行：

```
$ oc policy can-i --list
```

输出提供了一个完整的列表。

要确定您可以执行特定的操作动词，请运行：

```
$ oc policy can-i <verb> <resource>
```

用户范围 可以提供有关给定范围的更多信息。例如：

```
$ oc policy can-i <verb> <resource> --scopes=user:info
```

第 5 章 项目

5.1. 概述

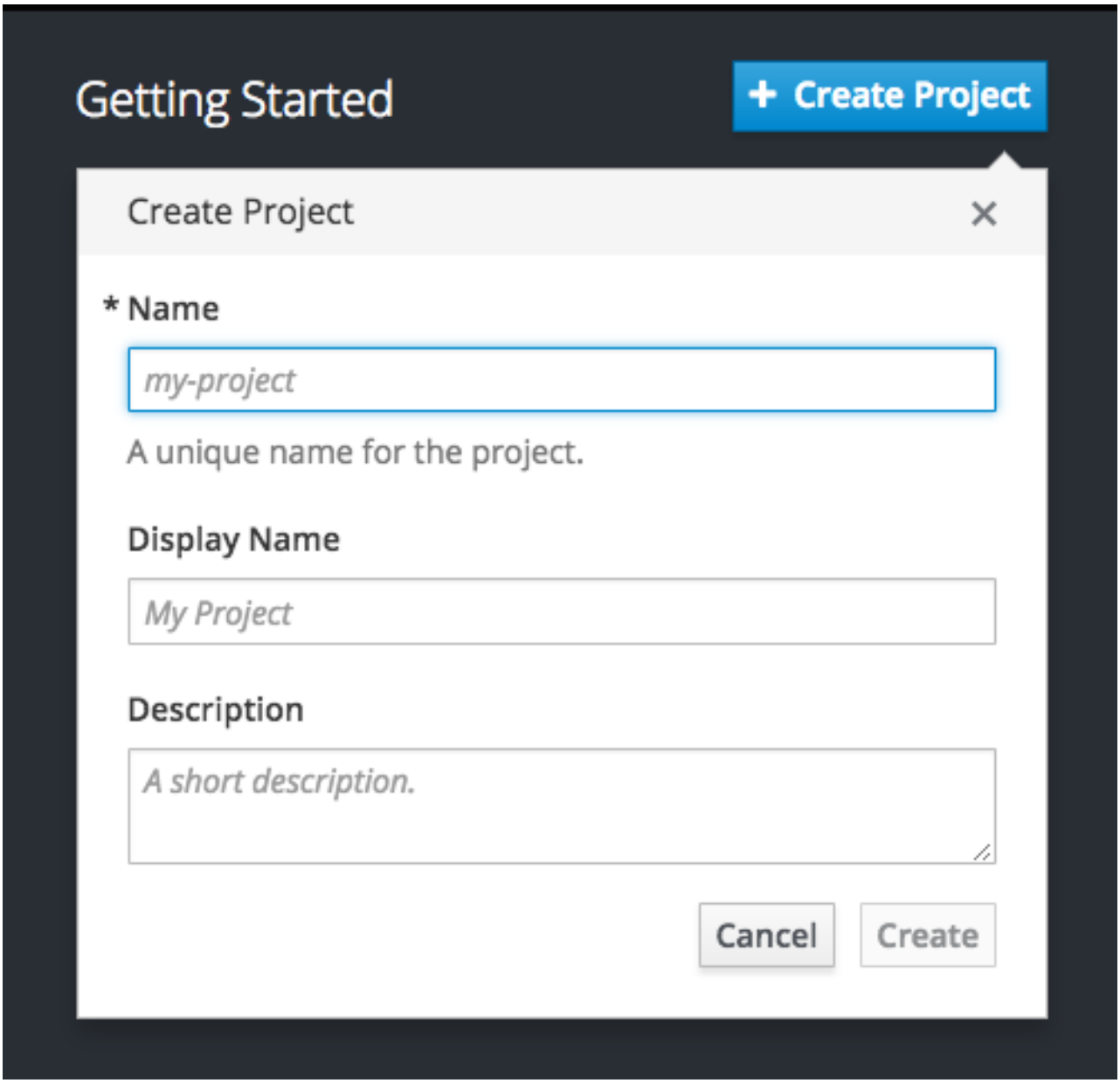
通过项目（project），一个社区用户可以在与其他社区隔离的前提下组织和管理其内容。

5.2. 创建一个项目

如果集群管理员允许，您可以使用 CLI 或 Web 控制台 创建新项目。

5.2.1. 使用 Web 控制台

若要使用 Web 控制台创建新项目，请单击 Projects 面板或 Projects 页面上的 **Create Project** 按钮。



默认情况下会显示 **Create Project** 按钮，但可以选择进行隐藏或自定义。

5.2.2. 使用 CLI

使用 CLI 创建新项目：

```
$ oc new-project <project_name> \
    --description="<description>" --display-name="<display_name>"
```

例如：

```
$ oc new-project hello-openshift \
    --description="This is an example project to demonstrate OpenShift v3" \
    --display-name="Hello OpenShift"
```

注意

系统管理员可能会限制允许创建的项目数量。达到限制后，可能需要删除现有项目来创建新项目。

5.3. 查看项目

查看项目时，只能看到根据授权策略您有权访问的项目。

要查看项目列表，请运行：

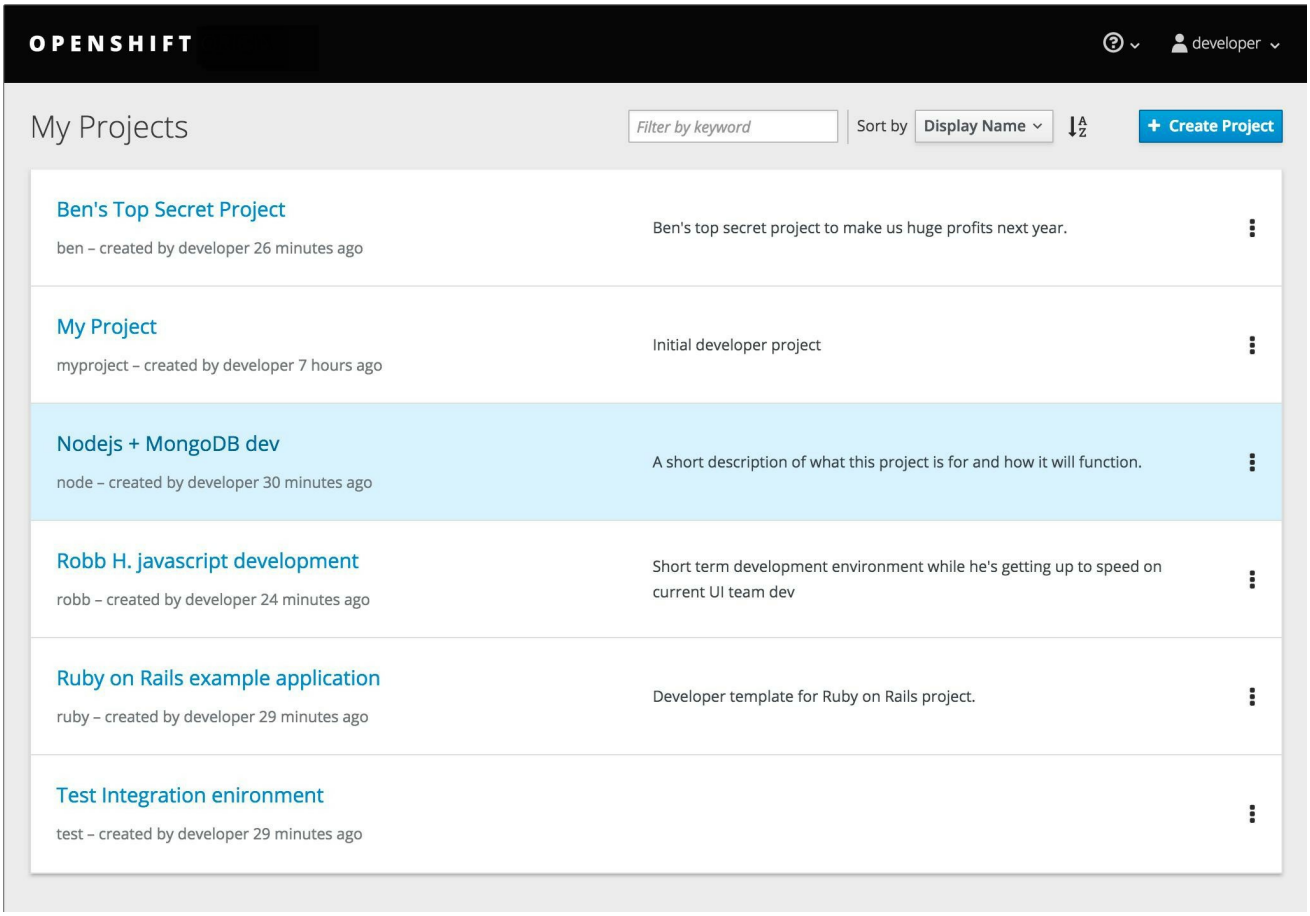
```
$ oc get projects
```

您可以从当前项目更改到其他项目，以进行 CLI 操作。然后，所有操控项目范围内容的后续操作都会使用指定的项目：

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

您还可以使用 Web 控制台在 项目之间查看和更改。身份验证并登录后，您会看到您有权访问的项目列表。

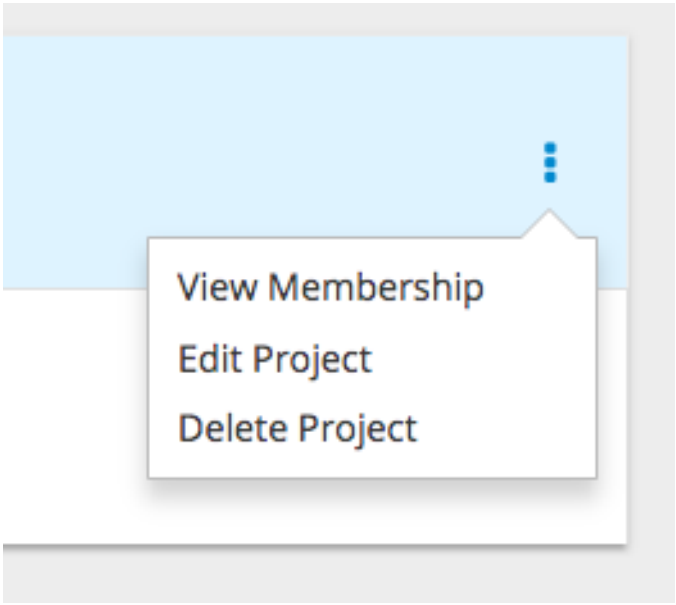
服务目录显示的右侧面板提供对最新访问的项目（最多五个项目）的快速访问。如需项目的完整列表，请使用右侧面板顶部的 **View All** 链接。



如果使用 CLI 创建新项目，您可以在浏览器中刷新页面以查看新项目。

选择一个项目可进入项目的项目概述。

点击特定项目的 kebab 菜单来为您提供以下选项：



5.4. 检查项目状态

oc status 命令提供当前项目的高级概述，及其组件及其关系。这个命令没有参数：

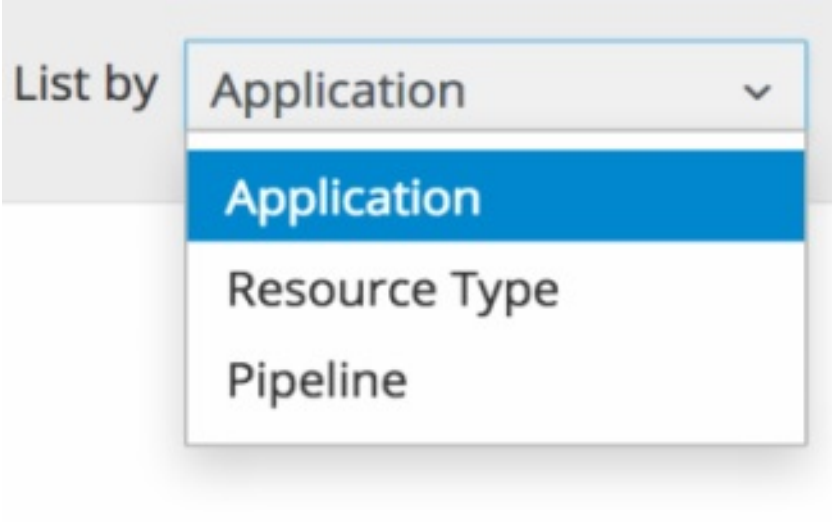
```
$ oc status
```

5.5. 按标签过滤

您可以使用资源标签过滤 Web 控制台中 的项目页面的内容。您可以从推荐的标签名称和值中选择，或者自行输入。可以添加多个过滤器。应用多个过滤器时，资源必须匹配所有过滤器才能保持可见。

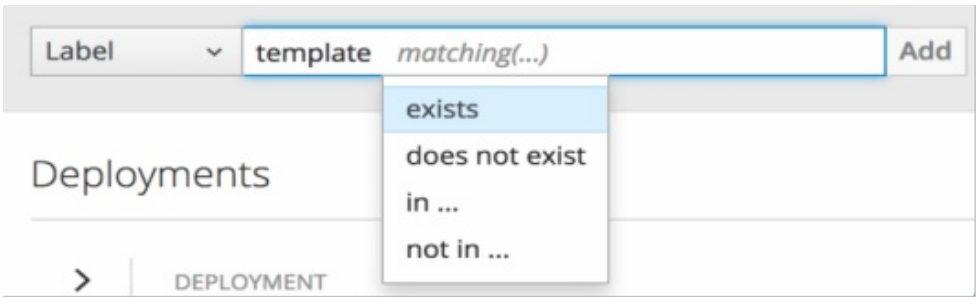
按标签进行过滤：

1. 选择标签类型：

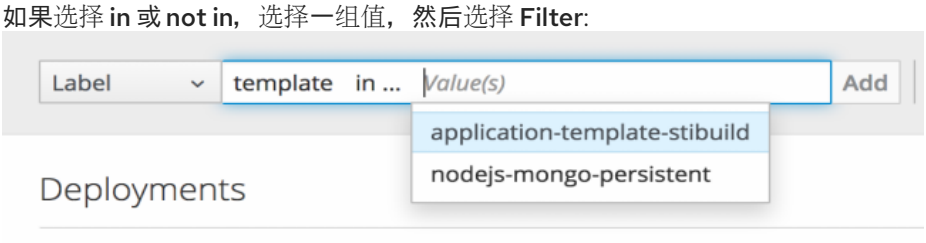


2. 任选以下一项：

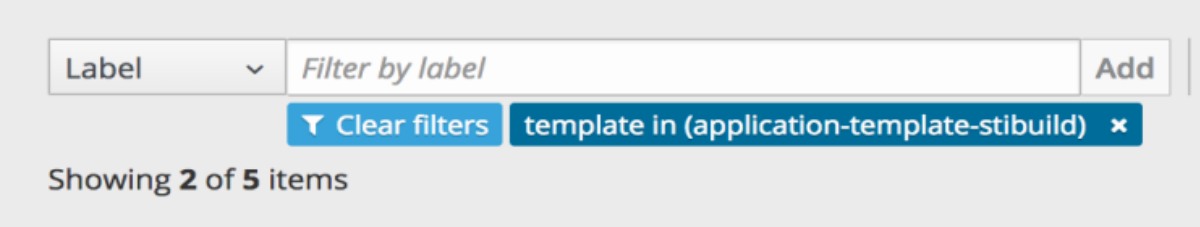
exists	验证标签名称是否存在，但忽略其值。
does not exist	验证标签名称不存在，但忽略其值。
in	验证标签名称是否存在，并且等于所选值之一。
not in	验证标签名称不存在，或者不等于任何所选值。



a. 如果选择 **in** 或 **not in**，选择一组值，然后选择 **Filter**:



3. 添加过滤器后，您可以通过选择 **Clear all 过滤器**来停止过滤，或者点击单个过滤器来删除它们：



5.6. 书签页面状态

OpenShift Container Platform Web 控制台现在书签页状态，这有助于保存标签过滤器和其他设置。

当您执行更改页面的状态时，如在标签页间切换时，浏览器导航栏中的 URL 会被自动更新。

5.7. 删除项目

当您删除项目时，服务器会将项目状态从 Active 更新为 Terminating。然后，服务器会清除正在终止的项目中的所有内容，然后最终删除项目。在项目处于 Terminating 状态时，用户无法向项目添加新内容。可以从 CLI 或 Web 控制台删除项目。

使用 CLI 删除项目：

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

第 6 章 迁移应用程序

6.1. 概述

本主题涵盖 OpenShift 版本 2(v2)应用迁移到 OpenShift 版本 3(v3)的迁移步骤。

注意

本主题使用一些特定于 OpenShift v2 的术语。比较 OpenShift Enterprise 2 和 OpenShift Enterprise 3 可深入了解两个版本与所用的语言之间的差别。

要将 OpenShift v2 应用程序迁移到 OpenShift Container Platform v3，v2 应用程序中的所有cartridge 必须记录，因为每个 v2 cartridge 等同于 OpenShift Container Platform v3 中的相应镜像或模板，且必须单独迁移。对于每个 cartridge，还必须记录所有依赖项或所需软件包，因为它们必须包含在 v3 镜像中。

常规迁移步骤为：

1. 备份 v2 应用。
   - Web cartridge:源代码可以备份到 Git 存储库，例如通过推送到 GitHub 上的存储库。
   - 数据库 cartridge:可使用转储命令（**mongodump**、**mysqldump**、**pg_dump**）备份数据库，以备份数据库。
   - Web 和数据库模块：**rhc** 客户端工具提供备份多个模块的快照功能：

     ```
     $ rhc snapshot save <app_name>
     ```

     快照是可以解压缩的 tar 文件，其内容是应用源代码和数据库转储。
2. 如果应用程序有数据库 cartridge，请创建一个 v3 数据库应用程序，将数据库转储同步到新 v3 数据库应用程序的 pod，然后使用数据库恢复命令恢复 v3 数据库应用程序中的 v2 数据库。
3. 对于 Web 框架应用程序，编辑应用源代码，使其与 v3 兼容。然后，添加 Git 存储库中的相应文件中所需的任何依赖项或软件包。将 v2 环境变量转换为对应的 v3 环境变量。
4. 从源（您的 Git 存储库）或者从带有 Git URL 的快速入门中创建 v3 应用程序。另外，将数据库服务参数添加到新应用，将数据库应用链接到 Web 应用。
5. 在 v2 中，有一个集成的 Git 环境，应用程序会在更改推送到 v2 Git 存储库时自动重新构建和重启。在 v3 中，若要由推送到公共 Git 存储库的源代码更改自动触发构建，您必须在 v3 的初始构建完成后设置 webhook。

6.2. 迁移数据库应用程序

6.2.1. 概述

本节复习如何将 MySQL、PostgreSQL 和 MongoDB 数据库应用程序从 OpenShift 版本 2(v2)迁移到 OpenShift 版本 3(v3)。

6.2.2. 支持的数据库

v2	v3
MongoDB:2.4	MongoDB:2.4, 2.6
MySQL:5.5	MySQL:5.5, 5.6
PostgreSQL:9.2	PostgreSQL:9.2, 9.4

6.2.3. MySQL

1. 将所有数据库导出到转储文件并将其复制到本地计算机（位于当前目录中）：

```
$ rhc ssh <v2_application_name>
$ mysqldump --skip-lock-tables -h $OPENSHIFT_MYSQL_DB_HOST -P
${OPENSHIFT_MYSQL_DB_PORT:-3306} -u ${OPENSHIFT_MYSQL_DB_USERNAME:-
'admin'} \
 --password="$OPENSHIFT_MYSQL_DB_PASSWORD" --all-databases > ~/app-
root/data/all.sql
$ exit
```

2. 将 **dbdump** 下载到您的本地机器中：

```
$ mkdir mysqldumpdir
$ rhc scp -a <v2_application_name> download mysqldumpdir app-root/data/all.sql
```

3. 从模板创建 v3 **mysql-persistent** pod：

```
$ oc new-app mysql-persistent -p \
  MYSQL_USER=<your_V2_mysql_username> -p \
  MYSQL_PASSWORD=<your_v2_mysql_password> -p MYSQL_DATABASE=
<your_v2_database_name>
```

4. 检查 pod 是否准备就绪：

```
$ oc get pods
```

5. 当 pod 上线并运行时，将数据库存档文件复制到您的 v3 MySQL pod 中：

```
$ oc rsync /local/mysqldumpdir <mysql_pod_name>:/var/lib/mysql/data
```

6. 恢复运行 v3 的 pod 中的数据库：

```
$ oc rsh <mysql_pod>
$ cd /var/lib/mysql/data/mysqldumpdir
```

在 v3 中，若要恢复您需要以 **root** 用户身份访问 MySQL 的数据库。

在 v2 中，**$OPENSHIFT_MYSQL_DB_USERNAME** 对所有数据库具有完整权限。在 v3 中，您必须为每个数据库授予 **$MYSQL_USER** 的权限。

```
$ mysql -u root
$ source all.sql
```

将 **<dbname>** 的所有权限授予 **<your_v2_username>@localhost**，然后清空权限。

7. 从 pod 中删除转储目录：

```
$ cd ../; rm -rf /var/lib/mysql/data/mysqldumpdir
```

支持的 MySQL 环境变量

v2	v3
OPENSHIFT_MYSQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_MYSQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_MYSQL_DB_USERNAME	**MYSQL_USER**
OPENSHIFT_MYSQL_DB_PASSWORD	**MYSQL_PASSWORD**
OPENSHIFT_MYSQL_DB_URL	
OPENSHIFT_MYSQL_DB_LOG_DIR	
OPENSHIFT_MYSQL_VERSION	
OPENSHIFT_MYSQL_DIR	
OPENSHIFT_MYSQL_DB_SOCKET	
OPENSHIFT_MYSQL_IDENT	
OPENSHIFT_MYSQL_AIO	**MYSQL_AIO**
OPENSHIFT_MYSQL_MAX_ALLOWED_PACKET	**MYSQL_MAX_ALLOWED_PACKET**
OPENSHIFT_MYSQL_TABLE_OPEN_CACHE	**MYSQL_TABLE_OPEN_CACHE**
OPENSHIFT_MYSQL_SORT_BUFFER_SIZE	**MYSQL_SORT_BUFFER_SIZE**
OPENSHIFT_MYSQL_LOWER_CASE_TABLE_NAMES	**MYSQL_LOWER_CASE_TABLE_NAMES**
OPENSHIFT_MYSQL_MAX_CONNECTIONS	**MYSQL_MAX_CONNECTIONS**
OPENSHIFT_MYSQL_FT_MIN_WORD_LEN	**MYSQL_FT_MIN_WORD_LEN**

v2	v3
OPENSHIFT_MYSQL_FT_MAX_WORD_LEN	**MYSQL_FT_MAX_WORD_LEN**
OPENSHIFT_MYSQL_DEFAULT_STORAGE_ENGINE	
OPENSHIFT_MYSQL_TIMEZONE	
	MYSQL_DATABASE
	MYSQL_ROOT_PASSWORD
	MYSQL_MASTER_USER
	MYSQL_MASTER_PASSWORD

6.2.4. PostgreSQL

1. 从 gear 备份 v2 PostgreSQL 数据库：

```
$ rhc ssh -a <v2-application_name>
$ mkdir ~/app-root/data/tmp
$ pg_dump <database_name> | gzip > ~/app-root/data/tmp/<database_name>.gz
```

2. 将备份文件提取到本地机器：

```
$ rhc scp -a <v2_application_name> download <local_dest> app-root/data/tmp/<db-name>.gz
$ gzip -d <database-name>.gz
```

注意

将备份文件保存到单独的文件夹，以供第 4 步使用。

3. 使用 v2 应用程序数据库名称、用户名和密码来创建 PostgreSQL 服务，以创建新服务：

```
$ oc new-app postgresql-persistent -p POSTGRESQL_DATABASE=dbname -p
POSTGRESQL_PASSWORD=password -p POSTGRESQL_USER=username
```

4. 检查 pod 是否准备就绪：

```
$ oc get pods
```

5. 当 pod 上线并运行时，将备份目录同步到 pod：

```
$ oc rsync /local/path/to/dir <postgresql_pod_name>:/var/lib/pgsql/data
```

6. 远程访问 pod:

```
$ oc rsh <pod_name>
```

7. 恢复数据库：

```
psql dbname < /var/lib/pgsql/data/<database_backup_file>
```

8. 删除不再需要的所有备份文件：

```
$ rm /var/lib/pgsql/data/<database-backup-file>
```

支持的 PostgreSQL 环境变量

v2	v3
OPENSHIFT_POSTGRESQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_POSTGRESQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_POSTGRESQL_DB_USERNAME	**POSTGRESQL_USER**
OPENSHIFT_POSTGRESQL_DB_PASSWORD	**POSTGRESQL_PASSWORD**
OPENSHIFT_POSTGRESQL_DB_LOG_DIR	
OPENSHIFT_POSTGRESQL_DB_PID	
OPENSHIFT_POSTGRESQL_DB_SOCKET_DIR	
OPENSHIFT_POSTGRESQL_DB_URL	
OPENSHIFT_POSTGRESQL_VERSION	
OPENSHIFT_POSTGRESQL_SHARED_BUFFERS	
OPENSHIFT_POSTGRESQL_MAX_CONNECTIONS	
OPENSHIFT_POSTGRESQL_MAX_PREPARED_TRANSACTIONS	
OPENSHIFT_POSTGRESQL_DATESTYLE	
OPENSHIFT_POSTGRESQL_LOCALE	
OPENSHIFT_POSTGRESQL_CONFIG	

v2	v3
OPENSHIFT_POSTGRESQL_SSL_ENABLED	
	POSTGRESQL_DATABASE
	POSTGRESQL_ADMIN_PASSWORD

6.2.5. MongoDB

注意

- 对于 OpenShift v3:MongoDB shell 版本 3.2.6
- 对于 OpenShift v2：MongoDB shell 版本 2.4.9

1. 通过 **ssh** 命令远程访问 v2 应用程序：

```
$ rhc ssh <v2_application_name>
```

2. 运行 **mongodump**，使用 **-d <database_name> -c <collections>** 指定单个数据库。如果没有这些选项，转储所有数据库。每个数据库在其自己的目录中转储：

```
$ mongodump -h $OPENSHIFT_MONGODB_DB_HOST -o app-root/repo/mydbdump -u
'admin' -p $OPENSHIFT_MONGODB_DB_PASSWORD
$ cd app-root/repo/mydbdump/<database_name>; tar -cvzf dbname.tar.gz
$ exit
```

3. 将 **dbdump** 下载到 **mongodump** 目录中的本地机器：

```
$ mkdir mongodump
$ rhc scp -a <v2 appname> download mongodump \
  app-root/repo/mydbdump/<dbname>/dbname.tar.gz
```

4. 在 v3 中启动 MongoDB pod。由于最新的镜像(3.2.6)不包括 **mongo-tools**，若要使用 **mongorestore** 或 **mongoimport** 命令，您需要编辑默认的 **mongodb-persistent** 模板，以指定包含 **mongo-tools, "mongodb:2.4"** 的镜像标签。因此，需要以下 **oc get --export** 命令并编辑：

```
$ oc get -o json --export template mongodb-persistent -n openshift > mongodb-
24persistent.json
```

编辑 *mongodb-24persistent.json* 的第 80 行；将 **mongodb:latest** 替换为 **mongodb:2.4**。

```
$ oc new-app --template=mongodb-persistent -n <project-name-that-template-was-created-
in> \
  MONGODB_USER=user_from_v2_app -p \
  MONGODB_PASSWORD=password_from_v2_db -p \
  MONGODB_DATABASE=v2_dbname -p \
  MONGODB_ADMIN_PASSWORD=password_from_v2_db
$ oc get pods
```

5. 当 mongodb pod 上线并运行时，将数据库存档文件复制到 v3 MongoDB pod 中：

```
$ oc rsync local/path/to/mongodump <mongodb_pod_name>:/var/lib/mongodb/data
$ oc rsh <mongodb_pod>
```

6. 在 MongoDB pod 中，为您要恢复的每个数据库完成以下内容：

```
$ cd /var/lib/mongodb/data/mongodump
$ tar -xzvf dbname.tar.gz
$ mongorestore -u $MONGODB_USER -p $MONGODB_PASSWORD -d dbname -v
/var/lib/mongodb/data/mongodump
```

7. 检查数据库是否已恢复：

```
$ mongo admin -u $MONGODB_USER -p $MONGODB_ADMIN_PASSWORD
$ use dbname
$ show collections
$ exit
```

8. 从 pod 中删除 **mongodump** 目录：

```
$ rm -rf /var/lib/mongodb/data/mongodump
```

支持的 MongoDB 环境变量

v2	v3
OPENSHIFT_MONGODB_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_MONGODB_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_MONGODB_DB_USERNAME	**MONGODB_USER**
OPENSHIFT_MONGODB_DB_PASSWORD	**MONGODB_PASSWORD**
OPENSHIFT_MONGODB_DB_URL	
OPENSHIFT_MONGODB_DB_LOG_DIR	
	MONGODB_DATABASE
	MONGODB_ADMIN_PASSWORD
	MONGODB_NOPREALLOC
	MONGODB_SMALLFILES
	MONGODB_QUIET

v2	v3
	MONGODB_REPLICA_NAME
	MONGODB_KEYFILE_VALUE

6.3. 迁移 WEB 框架应用程序

6.3.1. 概述

本主题介绍了如何将 Python、Ruby、PHP、Perl、Node.js、WordPress、Ghost、JBoss EAP、JBoss WS(Tomcat)和 Wildfly 10(JBoss AS)Web 框架应用程序从 OpenShift 版本 2(v2)迁移到 OpenShift 版本 3(v3)。

6.3.2. Python

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 v2 Git 存储库中：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 确保所有重要文件（如 ***setup.py***、***wsgi.py***、***requirements.txt etc***）都应推送到新存储库。
   - 确定您的应用程序所需的所有软件包都包含在 ***requirements.txt*** 中。
4. 使用 **oc** 命令从构建器镜像和源代码中启动新的 Python 应用程序：

```
$ oc new-app --strategy=source
python:3.3~https://github.com/<github-id>/<repo-name> --name=<app-name> -e
<ENV_VAR_NAME>=<env_var_value>
```

支持的 Python 版本

v2	v3
python:2.6, 2.7, 3.3	支持的容器镜像
Django	Django-psql-example（快速入门）

6.3.3. Ruby

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 v2 Git 存储库中：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 如果您没有 Gemfile 并运行一个简单的机架应用程序，请将这个 Gemfile 复制到您的源的根目录中：

```
https://github.com/sclorg/ruby-ex/blob/master/Gemfile
```

注意

支持 Ruby 2.0 的 **rack** gem 的最新版本是 1.6.4，因此需要将 Gemfile 修改为 **gem 'rack', "1.6.4"**。

对于 Ruby 2.2 或更高版本，使用 **rack** gem 2.0 或更高版本。

4. 使用 **oc** 命令从构建器镜像和源代码中启动新的 Ruby 应用程序：

```
$ oc new-app --strategy=source
ruby:2.0~https://github.com/<github-id>/<repo-name>.git
```

支持的 Ruby 版本

v2	v3
Ruby：1.8, 1.9, 2.0	支持的容器镜像
Ruby on Rails：3, 4	Rails-postgresql-example（快速入门）
Sinatra	

6.3.4. PHP

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 v2 Git 存储库中：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 使用 **oc** 命令从构建器镜像和源代码中启动新的 PHP 应用程序：

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

支持的 PHP 版本

v2	v3
PHP:5.3, 5.4	支持的容器镜像
带有 Zend Server 6.1 的 PHP 5.4	
CodeIgniter 2	
HHVM	
Laravel 5.0	
	cakephp-mysql-example (quickstart)

6.3.5. Perl

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 v2 Git 存储库中：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 编辑本地 Git 存储库并推送上游更改，使其与 v3 兼容：

   a. 在 v2 中，CPAN 模块驻留在 ***.openshift/cpan.txt*** 中。在 v3 中，s2i 构建器在源的根目录中查找名为 ***cpanfile*** 的文件。

```
$ cd <local-git-repository>
$ mv .openshift/cpan.txt cpanfile
```

   编辑 cpanfile，因为它的格式略有不同：

cpanfile 格式	cpan.txt 格式
需要 'cpan::mod';	cpan::mod
需要"Dancer"；	Dancer
需要 'YAML';	YAML

   b. 删除 ***.openshift*** 目录

注意

在 v3 中，**action_hooks** 和 **cron** 任务的支持方式不同。如需更多信息，请参阅 Action Hook。

4. 使用 **oc** 命令从构建器镜像和源代码中启动新的 Perl 应用程序：

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
```

支持的 Perl 版本

v2	v3
Perl:5.10	支持的容器镜像
	dancer-mysql-example(Quickstart)

6.3.6. Node.js

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 Git 存储库：

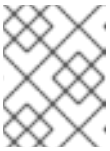
```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 编辑本地 Git 存储库并推送上游更改，使其与 v3 兼容：

   a. 删除 ***.openshift*** 目录。

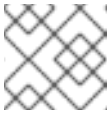
   

   注意

   在 v3 中，**action_hooks** 和 **cron** 任务的支持方式不同。如需更多信息，请参阅 Action Hook。

   b. 编辑 ***server.js***。

   - L116 server.js: 'self.app = express();'
   - L25 server.js: self.ipaddress = '0.0.0.0';
   - L26 server.js: self.port = 8080;

   注意

   行(L)来自基础 V2 cartridge **server.js**。

4. 使用 **oc** 命令从构建器镜像和源代码中启动新的 Node.js 应用程序：

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

支持的 Node.js 版本

v2	v3
Node.js 0.10	支持的容器镜像
	nodejs-mongodb-example。此快速入门模板只支持 Node.js 版本 6。

6.3.7. WordPress

重要

目前，迁移 WordPress 应用程序的支持仅由社区提供，而不受红帽支持提供。

有关将 WordPress 应用程序迁移到 OpenShift Container Platform v3 的指南，请参阅 OpenShift 博客。

6.3.8. Ghost

重要

目前，对迁移 Ghost 应用程序的支持仅由社区提供，而不受红帽支持。

有关将 Ghost 应用程序迁移到 OpenShift Container Platform v3 的指导，请参阅 OpenShift 博客。

6.3.9. JBoss EAP

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 Git 存储库：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 如果存储库包含预构建的 ***.war*** 文件，它们需要驻留在存储库的根目录的 ***deployments*** 目录中。
4. 使用 JBoss EAP 7 构建器镜像(jboss-eap70-openshift)和来自 GitHub 的源代码存储库创建新应用：

```
$ oc new-app --strategy=source jboss-eap70-openshift:1.6~https://github.com/<github-id>/<repo-name>.git
```

6.3.10. JBoss WS(Tomcat)

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 Git 存储库：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 如果存储库包含预构建的 *.war* 文件，它们需要驻留在存储库的根目录的 *deployments* 目录中。

4. 使用 JBoss Web 服务器 3(Tomcat 7)构建器镜像(jboss-webserver30-tomcat7)和来自 GitHub 的源代码存储库创建新应用程序：

```
$ oc new-app --strategy=source
jboss-webserver30-tomcat7-openshift~https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

6.3.11. JBoss AS (Wildfly 10)

1. 设置新的 GitHub 存储库，并将它作为远程分支添加到当前本地 Git 存储库：

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. 将本地 v2 源代码推送到新存储库：

```
$ git push -u <remote-name> master
```

3. 编辑本地 Git 存储库并推送上游更改，使其与 v3 兼容：

   a. 删除 *.openshift* 目录。

注意

在 v3 中，**action_hooks** 和 **cron** 任务的支持方式不同。如需更多信息，请参阅 Action Hook。

   b. 将 *deployments* 目录添加到源存储库的根目录中。将 *.war* 文件移到 'deployments' 目录。

4. 使用 **oc** 命令从构建器镜像和源代码中启动新的 Wildfly 应用程序：

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
 --image-stream=”openshift/wildfly:10.0" --name=<app-name> -e
 <ENV_VAR_NAME>=<env_var_value>
```

注意

参数 **--name** 是可选的，用于指定应用程序的名称。参数 **-e** 是可选的，可添加构建和部署流程所需的环境变量，如 **OPENSHIFT_PYTHON_DIR**。

6.3.12. 支持的 JBoss 版本

v2	v3
JBoss App Server 7	
Tomcat 6(JBoss EWS 1.0)	支持的容器镜像

v2	v3
Tomcat 7(JBoss EWS 2.0)	支持的容器镜像
Vert.x 2.1	
WildFly App Server 10	
WildFly App Server 8.2.1.Final	
WildFly App Server 9	
CapeDwarf	
JBoss Data Virtualization 6	支持的容器镜像
JBoss Enterprise App Platform(EAP)6	支持的容器镜像
JBoss Unified Push Server 1.0.0.Beta1, Beta2	
JBoss BPM 套件	支持的容器镜像
JBoss BRMS	支持的容器镜像
	jboss-eap70-openshift:1.3-Beta
	eap64-https-s2i
	eap64-mongodb-persistent-s2i
	eap64-mysql-persistent-s2i
	eap64-psql-persistent-s2i

6.4. QUICKSTART 示例

6.4.1. 概述

虽然 v2 quickstart to v3 Quickstart 没有清晰的迁移路径，但 v3 当前有以下快速入门。如果您的应用程序带有数据库，而不是使用 **oc new-app** 创建应用程序，则 **oc new-app** 会再次启动一个单独的数据库服务并使用通用环境变量将这两个应用程序链接到一起，您可以使用以下方法一次性从包含您的源代码的 GitHub 存储库中实例化链接的应用程序和数据库。您可以使用 **oc get templates -n openshift** 列出所有可用模板：

- CakePHP MySQL https://github.com/sclorg/cakephp-ex
  - template: cakephp-mysql-example

- Node.js MongoDB https://github.com/sclorg/nodejs-ex
  - template: nodejs-mongodb-example
- Django PosgreSQL https://github.com/sclorg/django-ex
  - template: django-psql-example
- Dancer MySQL https://github.com/sclorg/dancer-ex
  - template: dancer-mysql-example
- Rails PostgreSQL https://github.com/sclorg/rails-ex
  - template: rails-postgresql-example

6.4.2. 工作流

本地运行上述模板 URL 的 **git clone**。添加并提交应用程序源代码并推送 GitHub 存储库，然后从上面列出的其中一个模板启动 v3 Quickstart 应用程序：

1. 为您的应用程序创建 GitHub 存储库。
2. 克隆快速入门模板，并将 GitHub 存储库添加为远程：

```
$ git clone <one-of-the-template-URLs-listed-above>
$ cd <your local git repository>
$ git remote add upstream <https://github.com/<git-id>/<quickstart-repo>.git>
$ git push -u upstream master
```

3. 将您的源代码提交并推送到 GitHub：

```
$ cd <your local repository>
$ git commit -am “added code for my app”
$ git push origin master
```

4. 在 v3 中创建一个新应用程序：

```
$ oc new-app --template=<template> \
-p SOURCE_REPOSITORY_URL=<https://github.com/<git-id>/<quickstart_repo>.git> \
-p DATABASE_USER=<your_db_user> \
-p DATABASE_NAME=<your_db_name> \
-p DATABASE_PASSWORD=<your_db_password> \
-p DATABASE_ADMIN_PASSWORD=<your_db_admin_password> (1)
```

(1) 仅适用于 MongoDB。

您现在应该有 2 个 pod 运行、Web 框架 Pod 和数据库 Pod。Web 框架 pod 环境应当与数据库 pod 环境匹配。您可以使用 **oc set env pod/<pod_name> --list** 列出环境变量：

- **DATABASE_NAME** 现在为 **<DB_SERVICE>_DATABASE**
- **DATABASE_USER** 现在是 **<DB_SERVICE>_USER**
- **DATABASE_PASSWORD** 现在为 **<DB_SERVICE>_PASSWORD**

- **DATABASE_ADMIN_PASSWORD** 现在是 **MONGODB_ADMIN_PASSWORD** （仅适用于 MongoDB）
  如果没有指定 **SOURCE_REPOSITORY_URL**，该模板将使用上面列出的模板 URL(***https://github.com/openshift/<quickstart>-ex***)作为源存储库，并且将 **hello-welcome** 应用启动。

5. 如果您要迁移数据库，请将数据库导出到转储文件，并在新的 v3 数据库 pod 中恢复数据库。请参阅 Database Applications 中介绍的步骤，跳过 **oc new-app** 步骤，因为数据库 pod 已启动并在运行。

6.5. 持续集成和部署(CI/CD)

6.5.1. 概述

本节回顾 OpenShift 版本 2(v2)和 OpenShift 版本 3(v3)之间的持续集成和部署(CI/CD)应用的不同，以及如何将这些应用程序迁移到 v3 环境中。

6.5.2. Jenkins

OpenShift 版本 2(v2)和 OpenShift 版本 3(v3)中的 Jenkins 应用会因为架构中的基本差异而进行不同的配置。例如，在 v2 中，应用程序使用托管在 gear 中的集成 Git 存储库来存储源代码。在 v3 中，源代码位于托管在 pod 之外的公共或私有 Git 存储库中。

另外，在 OpenShift v3 中，Jenkins 作业只能被源代码更改触发，还由 ImageStream 中的更改，后者会更改用于构建应用的镜像及其源代码。因此，强烈建议您通过在 v3 中创建新的 Jenkins 应用来手动迁移 Jenkins 应用，然后使用适合 OpenShift v3 环境的配置重新创建作业。

请参考这些资源以了解有关如何创建 Jenkins 应用程序、配置作业和正确使用 Jenkins 插件的更多信息：

- https://github.com/openshift/origin/blob/master/examples/jenkins/README.md
- https://github.com/openshift/jenkins-plugin/blob/master/README.md
- https://github.com/openshift/origin/blob/master/examples/sample-app/README.md

6.6. WEBHOOK 和 ACTION HOOK

6.6.1. 概述

本节回顾 OpenShift 版本 2(v2)和 OpenShift 版本 3(v3)之间的 webhook 和操作 hook，以及如何将这些应用迁移到 v3 环境中。

6.6.2. Webhook

1. 从 GitHub 存储库创建 **BuildConfig** 后，请运行：

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

这将输出一个 Webhook GitHub URL，如下所示：

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

2. 从 GitHub Web 控制台将此 URL 剪切并粘贴到 GitHub 中。
3. 在 GitHub 存储库中，从 **Settings → Webhooks & Services** 中选择 **Add Webhook**。
4. 将 URL 输出（与上方相似）粘贴到 **Payload URL** 字段。
5. 将 **Content Type** 设置为 **application/json**。
6. 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

如果删除或重新创建应用程序，您必须使用新的 **BuildConfig** webhook url 更新 GitHub 中的 **Payload URL** 字段。

6.6.3. 操作 Hook

在 OpenShift 版本 2(v2)中，有构建、部署、post_deploy 和 pre_build 脚本或 action_hooks 位于 ***.openshift/action_hooks*** 目录中。虽然 v3 中没有适用于这些功能的一个一对一映射，但 v3 中的 S2I 工具有添加可自定义脚本的选项，在指定的 URL 或您的源存储库的 ***.s2i/bin*** 目录中。

OpenShift 版本 3(v3)还提供 post-build hook，用于运行镜像构建后镜像的基本测试，然后再推送到 registry。部署 hook 在部署配置中配置。

在 v2 中，action_hooks 通常用于设置环境变量。在 v2 中，任何环境变量都应该传递：

```
$ oc new-app <source-url> -e ENV_VAR=env_var
```

或：

```
$ oc new-app <template-name> -p ENV_VAR=env_var
```

另外，也可使用以下方法添加或更改环境变量：

```
$ oc set env dc/<name-of-dc>
ENV_VAR1=env_var1 ENV_VAR2=env_var2'
```

6.7. S2I 工具

6.7.1. 概述

Source-to-Image(S2I)工具将应用源代码注入到容器镜像中，最终产品是一个新的、可直接运行的容器镜像，它融合了构建器镜像和构建的源代码。S2I 工具可以安装到本地机器上，而没有来自存储库的 OpenShift Container Platform。

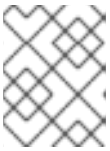
S2I 工具是一个非常强大的工具，用于在 OpenShift Container Platform 上使用应用程序之前在本地测试和验证应用程序和镜像。

6.7.2. 创建容器镜像

1. 识别应用所需的构建器镜像。红帽为不同的语言提供多个构建器镜像，包括 Python、Ruby、Perl、PHP 和 Node.js。其他镜像可从社区空间获取。

2. S2I 可以从本地文件系统中的源代码或从 Git 存储库构建镜像。从构建器镜像和源代码构建新容器镜像：

```
$ s2i build <source-location> <builder-image-name> <output-image-name>
```



注意

<source-location> 可以是 Git 存储库 URL，也可以是本地文件系统中源代码的目录。

3. 使用 Docker 守护进程测试构建的镜像：

```
$ docker run -d --name <new-name> -p <port-number>:<port-number> <output-image-name>
$ curl localhost:<port-number>
```

4. 将新镜像推送到 OpenShift registry。

5. 使用 **oc** 命令从 OpenShift registry 中的镜像创建新应用程序：

```
$ oc new-app <image-name>
```

6.8. 支持指南

6.8.1. 概述

本主题回顾了 OpenShift 版本 2(v2)和 OpenShift 版本 3(v3)支持的语言、框架、数据库和标记。

如需有关 OpenShift Container Platform 客户使用的通用组合的更多信息，请参阅 OpenShift Container Platform 测试的集成。

6.8.2. 支持的数据库

请参阅 Database Applications 的支持的 Databases 部分。

6.8.3. 支持的语言

- PHP
- Python
- Perl
- Node.js
- Ruby
- JBoss/xPaaS

6.8.4. 支持的框架

表 6.1. 支持的框架

v2	v3
Jenkins 服务器	jenkins-persistent
Drupal 7	
Ghost 0.7.5	
WordPress 4	
Ceylon	
Go	
MEAN	

6.8.5. 支持的标记

表 6.2. Python

v2	v3
pip_install	如果您的存储库包含 ***requirements.txt***，则默认调用 pip。否则，不使用 pip。

表 6.3. Ruby

v2	v3
disable_asset_compilation	这可以通过在 **buildconfig** 策略定义中将 **DISABLE_ASSET_COMPILATION** 环境变量设置为 **true** 来实现。

表 6.4. Perl

v2	v3
enable_cpan_tests	这可以通过在构建配置中将 **ENABLE_CPAN_TEST** 环境变量设置为 **true** 来实现。

表 6.5. PHP

v2	v3
use_composer	如果源存储库在根目录中包含 ***composer.json***，则始终使用 Composer。

表 6.6. Node.js

v2	v3
NODEJS_VERSION	N/A
use_npm	**npm** 总是用于启动应用程序，除非将 **DEV_MODE** 设为 **true**，本例中为 **nodemon**。

表 6.7. JBoss EAP, JBoss WS, WildFly

v2	v3
enable_debugging	这个选项通过设置部署配置上设置的 **ENABLE_JPDA** 环境变量来控制，方法是将其设置为任何非空值。
skip_maven_build	如果存在 ***pom.xml***，则会运行 maven。
java7	N/A
java8	JavaEE 使用 JDK8。

表 6.8. Jenkins

v2	v3
enable_debugging	N/A

表 6.9. All

v2	v3
force_clean_build	v3 中有一个类似的概念，因为 **buildconfig** 中的 **noCache** 字段会强制容器构建重新运行每个层。在 S2I 构建中，**incremental** 标记默认为 false，这代表**清理构建**。
hot_deploy	Ruby, Python, Perl, PHP, Node.js

v2	v3
enable_public_server_status	N/A
disable_auto_scaling	在默认情况下，自动缩放功能可以通过 pod 自动扩展来开启。

6.8.6. 支持的环境变量

- MySQL
- MongoDB
- PostgreSQL

第 7 章 教程

7.1. 概述

本节组包含如何在 OpenShift Container Platform 中启动并运行应用程序的信息，涵盖了不同的语言及其框架。

7.2. QUICKSTART 模板

7.2.1. 概述

快速入门是 OpenShift Container Platform 上运行的应用程序的基本示例。Quickstarts 提供多种语言和框架，并在模板中定义，模板由一组服务、构建配置和部署配置组成。该模板引用了构建和部署应用程序所需的镜像和源存储库。

要探索快速入门，请从模板创建应用程序。您的管理员可能已在您的 OpenShift Container Platform 集群中安装了这些模板，在这种情况下，您只需从 web 控制台中选择即可。有关如何上传、从中创建和修改模板 的更多信息，请参阅模板文档。

Quickstarts 是指包含应用程序源代码的源存储库。要自定义 Quickstart，请分叉存储库，并在从模板创建应用程序时，用分叉的存储库替换默认的源存储库名称。这将导致使用您的源代码而非所提供的示例源来执行构建。然后，您可以更新源存储库中的代码，并启动新的构建来查看反映在所部署的应用程序中的更改。

7.2.2. Web 框架 Quickstart 模板

这些快速入门提供了指定框架和语言的基本应用程序：

- Cakephp：PHP web 框架（包括 MySQL 数据库）
  - 模板定义
  - 源存储库
- Dancer：Perl web 框架（包括 MySQL 数据库）
  - 模板定义
  - 源存储库
- Django：Python web 框架（包括 PostgreSQL 数据库）
  - 模板定义
  - 源存储库
- NodeJS：NodeJS web 应用程序（包括 MongoDB 数据库）
  - 模板定义
  - 源存储库
- Rails：Ruby web 框架（包括 PostgreSQL 数据库）
  - 模板定义

- 源存储库

7.3. RUBY ON RAILS

7.3.1. 概述

Ruby on Rails 是使用 Ruby 编写的流行 web 框架。本指南介绍在 OpenShift Container Platform 上使用 Rails 4。

警告

我们强烈建议您完成整个教程，以概述在 OpenShift Container Platform 上运行应用程序所需的所有步骤。如果遇到问题，请尝试通读整个教程，然后再回看问题。该教程还可用于审查您之前采取的步骤，以确保正确执行了所有步骤。

对于本指南，您需要：

- 基本 Ruby/Rails 知识
- 本地安装的 Ruby 2.0.0+ 版本、Rubygems、Bundler
- 基本的 Git 知识
- 运行 OpenShift Container Platform v3 实例

7.3.2. 本地工作站设置

首先，确保 OpenShift Container Platform 实例正在运行且可用。有关如何启动并运行 OpenShift Container Platform 的更多信息，请检查 安装方法。另外，确保已安装 **oc** CLI 客户端，且可从命令 shell 访问命令，因此您可以使用您的电子邮件地址和密码使用它来 登录。

7.3.2.1. 设置数据库

Rails 应用程序几乎总是与数据库一同使用。对于本地开发，我们选择 PostgreSQL 数据库。要安装：

```
$ sudo yum install -y postgresql postgresql-server postgresql-devel
```

接下来，您需要使用以下方法初始化数据库：

```
$ sudo postgresql-setup initdb
```

该命令将创建 **/var/lib/pgsql/data** 目录，数据将存储在其中。

通过键入以下内容启动数据库：

```
$ sudo systemctl start postgresql.service
```

数据库运行时，创建 **rails** 用户：

```
$ sudo -u postgres createuser -s rails
```

请注意，我们创建的用户没有密码。

7.3.3. 编写应用程序

如果您要从头开始启动 Rails 应用程序，则需要首先安装 Rails gem。

```
$ gem install rails
Successfully installed rails-4.2.0
1 gem installed
```

安装完 Rails gem 后，使用 PostgreSQL 创建一个新应用程序，作为数据库：

```
$ rails new rails-app --database=postgresql
```

然后，更改到您的新应用目录。

```
$ cd rails-app
```

如果您已有应用程序，请确保 **Gemfile** 中存在 **pg** (postgresql) gem。如果没有通过添加 gem 来编辑 **Gemfile** ：

```
gem 'pg'
```

使用您的所有依赖项生成一个新的 **Gemfile.lock** ：

```
$ bundle install
```

除了将 **postgresql** 数据库与 **pg** gem 结合使用外，还需要确保 **config/database.yml** 正在使用 **postgresql** 适配器。

请确保更新了 **config/database.yml** 文件中的 **default** 部分，如下所示：

```
default: &default
  adapter: postgresql
  encoding: unicode
  pool: 5
  host: localhost
  username: rails
  password:
```

使用此 **rake** 命令创建应用程序的开发和测试数据库：

```
$ rake db:create
```

这将在您的 PostgreSQL 服务器中创建 **development** 和 **test** 数据库。

7.3.3.1. 创建欢迎页面

由于 Rails 4 在生产中不再提供静态 **public/index.html** 页面，因此我们需要创建一个新的 root 页面。

要具有自定义欢迎页面，需要执行以下步骤：

- 使用索引操作创建 **controller**
- 为 **welcome** 控制器 **index** 操作创建 **view** 页面
- 使用所创建的 **controller** 和 **view** 创建一个提供应用程序 root 页面的 **route**

Rails 提供了一个生成器，可为您执行所有必要的步骤。

```
$ rails generate controller welcome index
```

现在，已创建所有必需的文件，现在只需要在 **config/routes.rb** 文件中编辑行 2，如下所示：

```
root 'welcome#index'
```

运行 rails 服务器以验证页面是否可用。

```
$ rails server
```

在浏览器中访问 http://localhost:3000 即可查看您的页面。如果没有看到该页面，请检查输出至服务器的日志进行调试。

7.3.3.2. 为 OpenShift Container Platform 配置应用程序

要让您的应用程序与将在 OpenShift Container Platform 中运行的 PostgreSQL 数据库服务通信，您需要编辑 **config/database.yml** 中的 **default** 部分，以便在创建数据库服务时使用环境变量。

您编辑的 **config/database.yml** 中的 **default** 部分和预定义的变量应类似于如下：

```
<% user = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ? "root" :
ENV["POSTGRESQL_USER"] %>
<% password = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ?
ENV["POSTGRESQL_ADMIN_PASSWORD"] : ENV["POSTGRESQL_PASSWORD"] %>
<% db_service = ENV.fetch("DATABASE_SERVICE_NAME","").upcase %>

default: &default
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see rails configuration guide
  # http://guides.rubyonrails.org/configuring.html#database-pooling
  pool: <%= ENV["POSTGRESQL_MAX_CONNECTIONS"] || 5 %>
  username: <%= user %>
  password: <%= password %>
  host: <%= ENV["#{db_service}_SERVICE_HOST"] %>
  port: <%= ENV["#{db_service}_SERVICE_PORT"] %>
  database: <%= ENV["POSTGRESQL_DATABASE"] %>
```

有关最终文件应该如何查找的示例，请参阅 Ruby on Rails 示例应用程序 config/database.yml。

7.3.3.3. 将应用程序存储在 Git 中

如果您尚未安装 OpenShift Container Platform，则需要安装 git。

在 OpenShift Container Platform 中构建应用程序通常需要将源代码存储在 git 存储库中，因此如果您还没有 **git**，则需要安装 git。

运行 **ls -1** 命令，确保已在 Rails 应用程序目录中。命令输出应类似于：

```
$ ls -1
app
bin
config
config.ru
db
Gemfile
Gemfile.lock
lib
log
public
Rakefile
README.rdoc
test
tmp
vendor
```

现在，在 Rails 应用程序目录中运行这些命令以初始化代码并将其提交给 git：

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

提交应用程序后，您需要将其推送(push)到远程存储库。为此，您将需要一个 GitHub 帐户，用于 创建新存储库。

设置指向 **git** 存储库的远程存储库：

```
$ git remote add origin git@github.com:<namespace/repository-name>.git
```

之后，将应用程序推送到远程 git 存储库。

```
$ git push
```

7.3.4. 将应用程序部署到 OpenShift Container Platform

要部署 Ruby on Rails 应用程序，为应用程序创建一个新项目：

```
$ oc new-project rails-app --description="My Rails application" --display-name="Rails Application"
```

创建 **rails-app** 项目 后，您将自动切换到新的项目命名空间。

在 OpenShift Container Platform 中部署应用程序涉及三个步骤：

- 从 OpenShift Container Platform 的 PostgreSQL 镜像创建数据库服务。
- 从 OpenShift Container Platform 的 Ruby 2.0 构建器镜像 和用来与数据库服务进行连接的 Ruby on Rails 源代码创建一个前端 service

- 为应用程序创建路由。

7.3.4.1. 创建数据库服务

您的 Rails 应用程序需要一个正在运行的 数据库服务。对于此服务，请使用 PostgeSQL 数据库镜像。

要创建 数据库服务，请使用 oc new-app 命令。您必须将一些要在数据库容器内使用的必要环境变量传递给此命令。设置用户名、密码和数据库名称需要这些环境变量。您可随意更改这些环境变量的值。我们将设置的变量如下：

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

设置这些变量可确保：

- 存在具有指定名称的数据库
- 存在具有指定名称的用户
- 用户可以使用指定密码访问指定数据库

例如：

```
$ oc new-app postgresql -e POSTGRESQL_DATABASE=db_name -e
POSTGRESQL_USER=username -e POSTGRESQL_PASSWORD=password
```

若也要为数据库管理员设置密码，请将以下内容附加至上一命令中：

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

监控这个命令的进度：

```
$ oc get pods --watch
```

7.3.4.2. 创建 Frontend 服务

要将应用程序添加到 OpenShift Container Platform 中，您需要再次使用 **oc new-app** 命令指定应用程序所处的存储库，该仓库需要在创建数据库服务中设置数据库相关的环境变量 ：

```
$ oc new-app path/to/source/code --name=rails-app -e POSTGRESQL_USER=username -e
POSTGRESQL_PASSWORD=password -e POSTGRESQL_DATABASE=db_name -e
DATABASE_SERVICE_NAME=postgresql
```

通过此命令，OpenShift Container Platform 会获取源代码，设置构建器镜像，构建应用程序镜像，并将新创建的镜像与 指定的环境变量 一起部署。该应用程序命名为 **rails-app**。

您可以通过查看 **rails-app** DeploymentConfig 的 JSON 文档来验证环境变量是否已添加：

```
$ oc get dc rails-app -o json
```

您应看到以下部分：

```
env": [
   {
      "name": "POSTGRESQL_USER",
      "value": "username"
   },
   {
      "name": "POSTGRESQL_PASSWORD",
      "value": "password"
   },
   {
      "name": "POSTGRESQL_DATABASE",
      "value": "db_name"
   },
   {
      "name": "DATABASE_SERVICE_NAME",
      "value": "postgresql"
   }

],
```

检查构建过程：

```
$ oc logs -f build/rails-app-1
```

构建完成后，您可以查看 OpenShift Container Platform 中运行的 pod。

```
$ oc get pods
```

您应看到其中一行命令以 **myapp-<number>-<hash>** 开头，这是您在 OpenShift Container Platform 中运行的应用程序。

在应用程序正常工作前，您需要通过运行数据库迁移脚本来初始化数据库。具体可通过两种方式实现：

- 从正在运行的前端容器手动实现：

首先，您需要使用 rsh 命令执行到 frontend 容器：

```
$ oc rsh <FRONTEND_POD_ID>
```

从容器内部运行迁移：

```
$ RAILS_ENV=production bundle exec rake db:migrate
```

如果在 **development** 或 **test** 环境中运行 Rails 应用程序，则不必指定 **RAILS_ENV** 环境变量。

- 通过在模板中添加部署前生命周期 hook。例如，检查 Rails 示例应用程序中的 hook 示例。

7.3.4.3. 为您的应用程序创建路由

要通过向服务提供外部可访问的主机名（如 **www.example.com** ）来公开服务，请使用 OpenShift Container Platform 路由。对于您的情况，需要通过键入以下命令来公开前端服务：

```
$ oc expose service rails-app --hostname=www.example.com
```

警告

这是用户的责任来确保它们指定的主机名解析为路由器的 IP 地址。如需更多信息，请参阅 OpenShift Container Platform 文档：

- Routes
- 配置高可用性路由服务

7.4. 为 MAVEN 设置 NEXUS 镜像

7.4.1. 简介

使用 Java 和 Maven 开发应用时，您很可能会构建多次。为了缩短容器集的构建时间，可将 Maven 依赖项缓存在本地 Nexus 存储库中。本教程将指导您在群集上创建 Nexus 存储库。

本教程假设您正在处理已经设置用于 Maven 的项目。如果您有兴趣将 Maven 与 Java 项目搭配使用，则强烈建议您查阅 其指南。

另外，请确定检查应用程序的镜像以获取 Maven 镜像功能。使用 Maven 的许多镜像都有 **MAVEN_MIRROR_URL** 环境变量，可用于简化此过程。如果它没有这一功能，请阅读 Nexus 文档 以正确配置您的构建。

另外，请确保为每个 pod 有足够的资源才能正常工作。您可能必须 编辑 Nexus 部署配置中的 Pod 模板，以请求更多资源。

7.4.2. 设置 Nexus

1. 下载并部署官方 Nexus 容器镜像：

```
oc new-app sonatype/nexus
```

2. 通过公开新创建的 Nexus 服务来创建路由：

```
oc expose svc/nexus
```

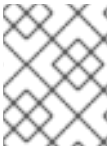
3. 使用 **oc get routes** 查找 pod 的新外部地址。

```
oc get routes
```

输出应类似：

```
NAME      HOST/PORT                         PATH      SERVICES   PORT       TERMINATION
nexus     nexus-myproject.192.168.1.173.xip.io          nexus      8081-tcp
```

4. 通过导航到 **HOST/PORT** 下的 URL，以确认 Nexus 正在运行。若要登录 Nexus，默认的管理员用户名是 **admin**，密码则为 **admin123**。



注意

Nexus 预配置了中央存储库，但您可能需要其他应用程序。对于许多红帽镜像，建议在 Maven 存储库中添加 **jboss-ga** 存储库。

7.4.2.1. 使用探测检查成功

这是设置 就绪度和存活度探测 的良好时机。这些将定期检查以查看 Nexus 是否正确运行。

```
$ oc set probe dc/nexus \
 --liveness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 -- echo ok
$ oc set probe dc/nexus \
 --readiness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 --get-url=http://:8081/nexus/content/groups/public
```

7.4.2.2. 在 Nexus 中添加持久性

注意

如果您不想持久存储，请继续 连接到 Nexus。但是，因为任何原因，您的缓存的依赖项和任何配置自定义都会丢失。

为 Nexus 创建持久卷声明(PVC)，以便在运行服务器的容器集终止时不会丢失缓存的依赖项。PVC 需要集群中的可用持久性卷(PV)。如果没有 PV 可用，且您没有集群中的管理员访问权限，请系统管理员为您创建 Read/Write 持久性卷。

否则，请参阅 OpenShift Container Platform 中的持久性存储 部分。

在 Nexus 部署配置中添加 PVC。

```
$ oc set volume dc/nexus --add \
 --name 'nexus-volume-1' \
 --type 'pvc' \
 --mount-path '/sonatype-work/' \
 --claim-name 'nexus-pv' \
 --claim-size '1G' \
 --overwrite
```

这会移除之前用于部署配置的 **emptyDir** 卷，并为挂载于 **/sonatype-work** 的 1GB 持久性存储添加一个声明（即存储依赖项的位置）。由于配置更改，Nexus 容器集将自动重新部署。

要验证 Nexus 是否正在运行，请在浏览器中刷新 Nexus 页面。您可以使用以下方法监控部署的进度：

```
$ oc get pods -w
```

7.4.3. 连接到 Nexus

后续步骤中演示定义使用新的 Nexus 存储库的构建。教程的其余部分将这个示例存储库 与 **wildfly-100-centos7** 一起用作构建器，但这些更改应该可用于任何项目。

示例构建器镜像 支持 **MAVEN_MIRROR_URL** 作为其环境的一部分，因此我们可以使用此 将构建器镜像指向我们的 Nexus 存储库。如果您的镜像不支持使用环境变量来配置 Maven 镜像，您可能需要修改构建器镜像，以提供正确的 Maven 设置以指向 Nexus 镜像。

```
$ oc new-build openshift/wildfly-100-centos7:latest~https://github.com/openshift/jee-ex.git \
 -e MAVEN_MIRROR_URL='http://nexus.<Nexus_Project>:8081/nexus/content/groups/public'
$ oc logs build/jee-ex-1 --follow
```

将 **<Nexus_Project>** 替换为 Nexus 存储库的项目名称。如果该项目与正在使用的应用程序位于同一个项目中，您可以删除 **<Nexus_Project>.**。在 OpenShift Container Platform 中了解更多有关 DNS 解析的信息。

7.4.4. 确认是否成功

在 Web 浏览器中，导航到 ***http://<NexusIP>:8081/nexus/content/groups/public***，以确认它已存储了应用程序的依赖项。您还可以检查构建日志，以查看 Maven 是否使用 Nexus 镜像。如果成功，您应该会看到引用 URL ***http://nexus:8081*** 的输出。

7.4.5. 其它资源

- 在 OpenShift Container Platform 中管理卷
- 改进 OpenShift Container Platform 上的 Java 构建时间
- Nexus 存储库文档

7.5. OPENSHIFT PIPELINE 构建

7.5.1. 简介

无论您创建简单的网站还是复杂的微服务 Web，请使用 OpenShift Pipelines 在 OpenShift 中构建、测试、部署和推进您的应用程序。

除了标准的 Jenkins 管道语法外，OpenShift Jenkins 镜像还提供 OpenShift 域特定语言(DSL)（通过 OpenShift Jenkins 客户端插件），其旨在提供易读、简洁、全面和流畅的语法，以便与 OpenShift API 服务器进行丰富的交互，甚至能更好地控制 OpenShift 集群上的应用程序的构建、部署和推广。

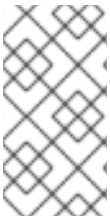
本例演示如何创建 OpenShift Pipeline，以使用 **nodejs-mongodb.json** 模板构建、部署和验证 **Node.js/MongoDB** 应用程序。

7.5.2. 创建 Jenkins Master

要创建 Jenkins master，请运行：

```
$ 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**。



注意

如果在集群中启用了 Jenkins 自动配置，您不需要对 Jenkins master 进行任何自定义，您可以跳过上一步。

有关 Jenkins 自动置备的更多信息，请参阅配置管道执行。

7.5.3. Pipeline 构建配置

现在 Jenkins master 已启动并运行，请创建一个 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。

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

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

有关配置 Pipeline 构建策略的更多信息，请参阅 Pipeline 策略选项。

7.5.4. Jenkinsfile

使用 **jenkinsPipelineStrategy** 创建 BuildConfig 后，通过使用内联 **jenkinsfile** 告知管道做什么。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**。

完成的 BuildConfig 可以在示例目录 **nodejs-sample-pipeline.yaml** 中的 OpenShift Origin 存储库中查看。

```
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().latest()
              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**。staging 环境的管道 BuildConfig 可以监控 **${templateName}-staging:latest** 镜像更改，然后将其部署到 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。

7.5.5. 创建管道

您可以通过运行以下命令，在 OpenShift 集群中创建 BuildConfig：

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

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
```

有关此处使用的 OpenShift DSL 语法的更多信息，请参阅 OpenShift Jenkins 客户端插件。

7.5.6. 启动管道

使用以下命令启动管道：

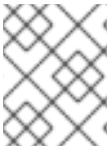
```
$ oc start-build nodejs-sample-pipeline
```

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。



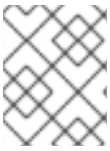
注意

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

7.5.7. OpenShift Pipelines 的高级选项

使用 OpenShift Pipelines，您可以在一个项目中启动 Jenkins，然后让 OpenShift 同步插件监控开发人员在其中工作的一组项目。以下小节概述了完成此过程的步骤。

- 要禁用 Jenkins auto=provisioning，请参阅配置管道执行。
- 要使 Jenkins 服务帐户能够访问要运行 OpenShift Pipelines 的每个项目，请参阅跨项目访问。
- 要添加项目来监控，请执行以下任一操作：
  - 登录 Jenkins 控制台。
    - 导航到 **Manage Jenkins**，再单击**配置系统**。
    - 更新 **OpenShift Jenkins Sync** 下的 **Namespace** 字段。
  - 使用 S2I 扩展名选项或扩展 OpenShift Jenkins 镜像，以更新 Jenkins 配置文件。



注意

避免从运行 OpenShift 同步插件的多个 Jenkins 部署中监控同一项目。这些实例之间没有协调，可能会发生不可预测的结果。

7.6. 二进制构建

7.6.1. 简介

OpenShift 中的二进制构建功能使开发人员能够将源或工件直接上传到构建中，而不是从 Git 存储库 URL 中拉取源。任何带有 source、Docker 或 custom 的 BuildConfig 都可以作为二进制构建启动。从本地工件启动构建时，现有源引用将被替换为来自本地用户机器的源。

源可以通过几种方法提供，它们对应于使用 start-build 命令时可用的参数：

- 来自一个文件(**--from-file**)：构建整个源都由一个文件组成，就会出现这种情况。例如，它可能是用于 Wildfly 构建的 Docker 构建、**pom.xml** 的 **Dockerfile**，或用于 Ruby 构建的 **Gemfile**。
- 来自一个目录(**--directory**)：当源位于本地目录且未提交到 Git 存储库时，请使用此选项。**start-build** 命令将创建给定目录的存档，并将它上传到构建器，作为来源。
- 来自存档(**--from-archive**）：当存档和源已存在时，请使用此选项。归档可以是 **tar**、**tar.gz** 或 **zip** 格式。
- 来自 Git 存储库(**--from-repo**)：这是用于当前用户本地计算机上的 Git 存储库一部分的源。当前存储库的 HEAD 提交将存档，并发送到 OpenShift 以进行构建。

7.6.1.1. 使用案例

二进制构建删除了构建从现有 Git 存储库拉取源的要求。使用二进制构建的原因包括：

- 构建和测试本地代码更改.可以从公共存储库中的源克隆，可以将本地更改上传到 OpenShift 中以进行构建。本地更改不必在任何位置提交或推送。
- 构建私有代码.新的构建可以从头开始作为二进制构建。然后源可以直接从本地工作站上传到 OpenShift，而无需将其签入到 SCM。
- 使用来自其他源的工件构建镜像。使用 Jenkins 管道时，二进制构建可用于将构建工件与 Maven 或 C 编译器等工具组合使用这些构建的运行时镜像。

7.6.1.2. 限制：

- 二进制构建不可重复。因为二进制构建需要用户在启动时上传工件，所以 OpenShift 无法重复同一构建，而无需用户每次都重复执行相同的上传。
- 无法自动触发二进制构建。只有当用户上传所需的二进制工件时，才能手动启动。

注意

作为二进制构建启动的构建可能也具有配置的源 URL。如果是这种情况，触发器将成功启动构建，但源来自配置的源 URL，而不是用户上次运行构建时提供的源 URL。

7.6.2. 教程概述

以下教程假设您有一个可用的 OpenShift 集群，并且具有可在创建工件的项目。它需要在本地同时拥有 **git** 和 **oc**。

7.6.2.1. 教程：构建本地代码更改

1. 根据现有源存储库创建新应用，并为它创建路由：

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git
$ oc expose svc/ruby-hello-world
```

2. 通过导航到路由的主机，等待初始构建完成并查看应用程序的页面。您应会收到欢迎页面：

```
$ oc get route ruby-hello-world
```

3. 本地克隆存储库：

```
$ git clone https://github.com/openshift/ruby-hello-world.git
$ cd ruby-hello-world
```

4. 更改应用程序视图。使用您最喜欢的编辑器编辑 **view/main.rb**:将 **<body>** 标签更改为 **<body style="background-color:blue">**。

5. 使用您的本地修改源启动新构建。在仓库的本地目录中运行：

```
----
$ oc start-build ruby-hello-world --from-dir="." --follow
----
```

构建完成后，应用程序重新部署后，导航到应用程序的路由主机会导致页面具有蓝色背景。

您可以在本地进行更改，并使用 **oc start-build --from-dir** 来构建代码。

您还可以创建代码分支，在本地提交更改，并使用存储库的 HEAD 作为构建的源：

```
$ git checkout -b my_branch
$ git add .
$ git commit -m "My changes"
$ oc start-build ruby-hello-world --from-repo="." --follow
```

7.6.2.2. 教程：构建私有代码

1. 创建存放您的代码的本地目录：

```
$ mkdir myapp
$ cd myapp
```

2. 在目录中，创建一个名为 **Dockerfile** 的文件，其内容如下：

```
FROM centos:centos7

EXPOSE 8080

COPY index.html /var/run/web/index.html

CMD cd /var/run/web && python -m SimpleHTTPServer 8080
```

3. 创建名为 **index.html** 的文件，其内容如下：

```
<html>
  <head>
    <title>My local app</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is my local application</p>
  </body>
</html>
```

4. 为您的应用程序创建新构建：

```
$ oc new-build --strategy docker --binary --docker-image centos:centos7 --name myapp
```

5. 使用本地目录的内容启动一个二进制构建：

```
$ oc start-build myapp --from-dir . --follow
```

6. 使用 **new-app** 部署应用程序，然后为其创建路由：

```
$ oc new-app myapp
$ oc expose svc/myapp
```

7. 获取路由的主机名并导航到它：

```
$ oc get route myapp
```

在构建并部署您的代码后，您可以通过调用 **oc start-build myapp --from-dir**，对本地文件进行更改并启动新的构建。构建后，代码将被自动部署，当您刷新页面时，更改将会被反映到您的浏览器中。

7.6.2.3. 教程：来自管道的二进制工件

OpenShift 上的 Jenkins 允许在适当的工具中使用从属镜像来构建您的代码。例如，您可以使用 **maven** slave 从代码存储库构建 WAR。但是，构建此工件后，您需要将其提交到包含运行代码的适当运行时工件的镜像。二进制构建可用于将这些工件添加到运行时镜像中。在以下教程中，我们将创建一个 Jenkins 管

道，以使用 **maven** slave 构建 WAR，然后使用带有 **Dockerfile** 的二进制构建将该 WAR 添加到 wildfly 运行时镜像中。

1. 为您的应用程序创建新目录：

```
$ mkdir mavenapp
$ cd mavenapp
```

2. 创建一个 **Dockerfile**，它将 WAR 复制到 wildfly 镜像内部的适当位置，以进行执行。将以下内容复制到名为 **Dockerfile** 的本地文件中：

```
FROM wildfly:latest
COPY ROOT.war /wildfly/standalone/deployments/ROOT.war
CMD  $STI_SCRIPTS_PATH/run
```

3. 为该 Dockerfile 创建新的 BuildConfig：

注意

这将自动启动构建，该构建最初将失败，因为 **ROOT.war** 构件还不可用。以下管道将使用二进制构建将该 WAR 传递给构建。

```
$ cat Dockerfile | oc new-build -D - --name mavenapp
```

4. 使用 Jenkins 管道创建 BuildConfig，它将构建 WAR，然后使用该 WAR 使用之前创建的 **Dockerfile** 构建镜像。同一模式可用于由一组工具构建二进制工件的其他平台，然后与最终软件包的不同运行时镜像组合。将以下代码保存到 **mavenapp-pipeline.yml** ：

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: mavenapp-pipeline
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        pipeline {
          agent { label "maven" }
          stages {
            stage("Clone Source") {
              steps {
                checkout([$class: 'GitSCM',
                          branches: [[name: '*/master']],
                          extensions: [
                            [$class: 'RelativeTargetDirectory', relativeTargetDir: 'mavenapp']
                          ],
                          userRemoteConfigs: [[url: 'https://github.com/openshift/openshift-jee-
sample.git']]
                        ])
              }
            }
            stage("Build WAR") {
              steps {
                dir('mavenapp') {
```

```
            sh 'mvn clean package -Popenshift'
          }
        }
      }
      stage("Build Image") {
        steps {
          dir('mavenapp/target') {
            sh 'oc start-build mavenapp --from-dir . --follow'
          }
        }
      }
    }
  }
  type: JenkinsPipeline
triggers: []
```

5. 创建管道构建。如果 Jenkins 没有部署到项目中，使用管道创建 BuildConfig 将导致 Jenkins 部署。Jenkins 准备好构建管道前可能需要几分钟时间。您可以通过调用 **oc rollout status dc/jenkins** 检查 Jenkins rollout 的状态：

```
$ oc create -f ./mavenapp-pipeline.yml
```

6. Jenkins 就绪后，启动前面定义的管道：

```
$ oc start-build mavenapp-pipeline
```

7. 当管道完成构建后，使用 new-app 部署新应用程序并公开其路由：

```
$ oc new-app mavenapp
$ oc expose svc/mavenapp
```

8. 使用您的浏览器，进入应用程序的路由：

```
$ oc get route mavenapp
```

第 8 章 BUILDS

8.1. 构建如何工作

8.1.1. 什么是构建？

OpenShift Container Platform 中的 *构建* 是将输入参数转换为结果对象的过程。大多数情况下，构建用于将源代码转换为可运行的容器镜像。

*构建配置*或 **BuildConfig** 的特征就是 *构建策略*和一个或多个源。策略决定上述过程，而源则提供输入。

构建策略有：

- Source-to-Image (S2I) (description, options)
- Pipeline (description, options)
- Docker (description, options)
- Custom (description, options)

有六种类型的源作为 *构建输入* 提供 ：

- Git
- Docker
- 二进制
- Image
- 输入 secret
- 外部工件 (artifact)

每个构建策略最多可考虑或忽略特定类型的源，并确定要使用方法。Binary 和 Git 是互斥源类型。Dockerfile 和镜像可以单独单独使用，也可以与 Git 或 Binary 一起使用。Binary 源类型在 向系统指定选项中的其它选项 中是唯一的。

8.1.2. BuildConfig 是什么？

构建配置描述了单个构建定义，以及应该创建新构建时的一组 触发器。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。

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。如果选择稍后手动调整配置，则了解 **BuildConfig** 的组成部分及其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 Object Definition

```
kind: "BuildConfig"
apiVersion: "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) 此规格将创建一个名为 **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，请参考每种源类型的文档。

(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。上面的示例使用 **ruby-20-centos7** 容器镜像，Source-To-Image 将用于应用程序构建。

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

(7) **postCommit** 部分定义一个可选的 构建 hook。

8.2. 基本构建操作

8.2.1. 启动构建

使用以下命令，从当前项目中的现有构建配置手动启动新构建：

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

使用 **--from-build** 标志重新运行构建：

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

指定 **--follow** 标志，在 stdout 中流传输构建日志：

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

指定 **--env** 标志来为构建设置任何所需的环境变量：

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

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。

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
```

8.2.2. 取消构建

使用 Web 控制台或以下 CLI 命令手动取消构建：

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

同时取消多个构建：

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

取消从构建配置创建的所有构建：

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

取消给定状态下的所有构建（如 **new** 或 **pending**状态），忽略其他状态下的构建：

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

8.2.3. 删除 BuildConfig

使用以下命令删除 **BuildConfig** ：

```
$ 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>
```

8.2.4. 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：

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

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。

8.2.5. 访问构建日志

您可以使用 Web 控制台或 CLI 访问构建日志。

直接使用构建来流传输日志：

```
$ oc logs -f build/<build_name>
```

流传输构建配置的最新构建的日志：

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

返回构建配置的给定版本构建的日志：

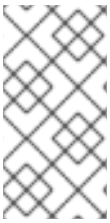
```
$ oc logs --version=<number> bc/<buildconfig_name>
```

日志详细程度

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。



注意

平台管理员可以通过为 **BuildDefaults** 准入控制器配置 **env/BUILD_LOGLEVEL** 来设置整个 OpenShift Container Platform 实例的默认构建详细程度。此默认值可以通过在给定的 **BuildConfig** 中指定 **BUILD_LOGLEVEL** 来覆盖。您可以通过将 **--build-loglevel** 传递给 **oc start-build**，在命令行中为非二进制构建指定优先级更高的覆盖。

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。

8.3. 构建输入

8.3.1. 构建输入如何工作

*构建输入*提供构建操作的源内容。在 OpenShift Container Platform 中，可以通过几种方法提供源。顺序排列：

- 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 和 ConfigMap

- 外部工件 (artifact)

不同的输入可以合并为一个构建。由于内联 Dockerfile 具有优先权，它可覆盖由另一个输入提供的名为 ***Dockerfile*** 的任何其他文件。二进制（本地）和 Git 存储库是互斥的输入。

当您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** 资源中定义的值时，输入 secret 很有用。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。

每当运行构建时：

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。

8.3.2. 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。

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

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。

8.3.3. 镜像源

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

源路径可以是指定镜像内的任何绝对路径。目的地必须是相对目录路径。构建时会加载镜像，并将指定的文件和目录复制到构建过程上下文目录中。这与源存储库内容（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。

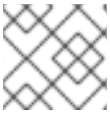
(2) 对包含要复制的文件的镜像的引用。

(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。

8.3.4. Git Source

指定后，将从提供的位置获取源代码。

如果提供了内联 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。

8.3.4.1. 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。您可以配置要使用的 HTTP 和 HTTPS 代理。两个字段都是可选的。也可以通过 **NoProxy** 字段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
```

集群管理员也可使用 Ansible 为 Git 克隆配置全局代理。

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

8.3.4.2. 源克隆 secret

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

支持以下源克隆 secret 配置。

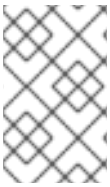
- *.gitconfig* 文件
- 基本身份验证
- SSH 密钥身份验证
- 可信证书颁发机构

注意

您还可以 组合使用 这些配置来满足特定的需求。

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。使用以下命令授予访问权限：

```
$ 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。

8.3.4.2.1. 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。此行为允许生成的构建自动使用存储在引用的 **Secret** 中的凭证与远程 Git 存储库进行身份验证，而无需进一步配置。

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。

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/*'
```

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：

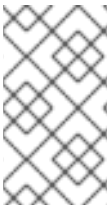
```
$ oc annotate secret mysecret \
    'build.openshift.io/source-secret-match-uri-1=https://*.mycorp.com/*'
```

8.3.4.2.2. 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"
```



注意

您还可以使用 **oc set build-secret** 命令在现有构建配置上设置源克隆 secret：

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

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

8.3.4.2.3. .gitconfig 文件

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

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
```

8.3.4.2.4. 安全 Git 的 .gitconfig 文件

如果 Git 服务器使用双向 SSL 和用户名进行保护，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：

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，您仍然需要指定用户名和密码来推进构建。

8.3.4.2.5. 基本身份验证

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
```

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
```

8.3.4.2.6. SSH 密钥身份验证

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**。使用以下命令生成 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。

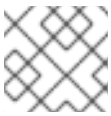
在使用 SSH 密钥访问私有存储库之前，请先创建 secret：

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

1 可选：添加此字段可启用严格的服务器主机密钥检查。

警告

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



注意

确保 **known_hosts** 文件中包含源代码主机条目。

8.3.4.2.7. 可信证书颁发机构

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

如果为 **CA 证书**创建 secret，OpenShift 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。

完成以下进程之一：

- 使用 CA 证书文件（推荐）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*。

- 禁用 Git TLS 验证。
  在构建配置的相应 strategy 部分中，将 **GIT_SSL_NO_VERIFY** 环境变量设置为 **true**。您可以使用 **oc set env** 命令管理 **BuildConfig** 环境变量。

8.3.4.2.8. 组合

以下是如何组合上述方法的几个示例，以便根据您的特定需求创建源克隆 secret。

a. 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
```

b. 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>
```

c. 使用 CA 证书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=ca.crt=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

d. 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
```

e. 使用 *.gitconfig* 文件和 CA 证书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> \
    --from-file=ca.crt=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

8.3.5. 二进制（本地）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**:您指定的文件的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，并作为二进制流发送到构建器。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**:您指定的存档将发送到构建器，在其中将其提取到构建上下文目录中。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。

在以上每个情况下：

- 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。

8.3.6. 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，在通过 Maven 构建 Java 应用程序时，您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。

8.3.6.1. 添加输入 Secret 和 ConfigMap

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。请注意，对于 Docker 策略，目标目录必须是相对路径。

8.3.6.2. 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。

8.3.6.3. 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。

例 8.1. 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。

8.3.6.4. 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** 策略还允许按照 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）。

8.3.7. 使用外部 Artifacts

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。

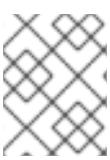
对于 **Source** 构建策略，必须在 ***assemble*** 脚本中放入适当的 shell 命令：

***.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
```



注意

有关如何控制 Source 构建使用哪个 ***assemble*** 和 ***run*** 脚本的更多信息，请参阅覆盖构建器镜像脚本。

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）

8.3.8. 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。

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"
```

您还可以使用 **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"
```

您还可以使用 **oc set build-secret** 命令在构建配置上设置 pull 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。

8.4. 构建输出

8.4.1. 构建输出概述

使用 **Docker** 或 **Source** 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"
```

8.4.2. 输出镜像环境变量

Docker 和 **Source** 策略构建在输出镜像上设置以下环境变量：

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

此外，任何用户定义的环境变量（例如，通过 **Source** 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。

8.4.3. 输出镜像标签

Docker 和 **Source** 构建在输出镜像上设置以下标签：

标签	描述
io.openshift.build.commit.author	构建中使用的源提交的作者
io.openshift.build.commit.date	构建中使用的源提交的日期
io.openshift.build.commit.id	构建中使用的源提交的哈希值
io.openshift.build.commit.message	构建中使用的源提交的消息

标签	描述
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"
```

8.4.4. 输出镜像 Digest

构建的镜像可由其 digest 唯一识别，后来可用于通过 digest（无论其当前标签是什么）来拉取镜像。

Docker 和 **Source** 构建会在镜像推送到 registry 后将摘要存储在 **Build.status.output.to.imageDigest** 中。摘要在 registry 中计算。因此，它可能无法始终存在，例如当 registry 没有返回摘要时，或者构建器镜像不知道其格式时。

在 registry 的 Successful Push 后构建镜像 Digest

```
status:
  output:
    to:
      imageDigest:
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
```

8.4.5. 将 Docker 凭证用于私有 registry

要将镜像推送到私有容器镜像 registry，可以使用 secret 来提供凭证。具体步骤请参阅构建输入。

8.5. 构建策略选项

8.5.1. Source-to-Image 策略选项

以下选项特定于 S2I 构建策略 ：

8.5.1.1. 强制 Pull

默认情况下，如果构建配置中指定的构建器镜像在节点上本地可用，则将使用该镜像。但是，要覆盖本地镜像并从镜像流指向的 registry 中刷新它，请创建一个 **BuildConfig**，将 **forcePull** 标志设为 **true** ：

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

1 使用的构建器镜像，其中节点上的本地版本可能不与镜像流指向的 registry 中的版本保持同步。

2 此标志会导致本地构建器镜像被忽略，并从镜像流指向的 registry 中拉取新版本。将 **forcePull** 设置为 **false** 将导致默认行为是遵守本地存储的镜像。

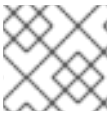
8.5.1.2. 增量构建

S2I 可以执行增量构建，也就是能够重复利用过去构建的镜像中的工件。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。



注意

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 主题。

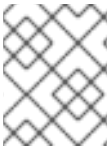
8.5.1.3. 覆盖构建器镜像脚本

您可以通过以下两种方式之一覆盖构建器镜像提供的 ***assemble***、***run*** 和 ***save-artifacts*** S2I 脚本。任一：

1. 在应用程序源存储库的 ***.s2i/bin*** 目录中提供 ***assemble***、***run***、和/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。有关如何使用 S2I 脚本的信息，请参阅 S2I 要求主题和 S2I 文档。

8.5.1.4. 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。

8.5.1.4.1. 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。各个镜像的文档中有完整的支持环境变量列表。

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。

例如，如果要禁用 Rails 应用程序的资产编译，您可以在 *.s2i/environment* 文件中添加 **DISABLE_ASSET_COMPILATION=true**，这样就可以在构建期间跳过资产编译。

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，您可以在 *.s2i/environment* 文件中添加 **RAILS_ENV=development**，使 Rails 应用程序以 **开发模式** 而非 **生产** 模式启动。

8.5.1.4.2. BuildConfig Environment

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。

例如，禁用 Rails 应用程序的资产编译：

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

构建环境部分提供了更多高级指导。

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

8.5.1.5. 通过 Web 控制台添加 Secret

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 Source-to-Image(S2I)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。

8.5.1.5.1. 启用拉取和推送

通过在构建配置中设置 **Pull Secret** 来启用拉取到私有 registry，并通过设置 **Push Secret** 来启用推送。

8.5.1.6. 忽略源文件

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）与 **.s2iignore** 文件中提供的模式匹配，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。

8.5.2. Docker 策略选项

以下选项特定于 Docker 构建策略 ：

8.5.2.1. FROM 镜像

Dockerfile 的 **FROM** 指令将被 **BuildConfig** 中的 **from** 替换：

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

8.5.2.2. 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。它可以只是默认 ***Dockerfile*** 以外的其他文件名（如 ***MyDockerfile***），或子目录中 Dockerfile 的路径（如 ***dockerfiles/app1/Dockerfile***）：

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

8.5.2.3. No Cache

Docker 构建通常重复使用执行构建的主机上找到的缓存层。将 **noCache** 选项设置为 **true** 会强制构建忽略缓存的层并重新运行 ***Dockerfile*** 的所有步骤：

```
strategy:
  dockerStrategy:
    noCache: true
```

8.5.2.4. 强制 Pull

默认情况下，如果构建配置中指定的构建器镜像在节点上本地可用，则将使用该镜像。但是，要覆盖本地镜像并从镜像流指向的 registry 中刷新它，请创建一个 **BuildConfig**，将 **forcePull** 标志设为 **true** ：

```
strategy:
  dockerStrategy:
    forcePull: true (1)
```

1. 此标志会导致本地构建器镜像被忽略，并从镜像流指向的 registry 中拉取新版本。将 **forcePull** 设置为 **false** 将导致默认行为是遵守本地存储的镜像。

8.5.2.5. 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。

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/"
```

集群管理员也可使用 Ansible 配置全局构建设置。

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

8.5.2.6. 通过 Web 控制台添加 Secret

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 docker 构建配置。
4. 在构建配置编辑器页面上或在 Web 控制台的 **fromimage** 页面中设置 **Source Secret**。
5. 点 **Save** 按钮。

8.5.2.7. Docker 构建参数

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"
```

构建参数将在构建启动时传递给 Docker。

8.5.2.7.1. 启用拉取和推送

通过在构建配置中设置 **Pull Secret** 来启用拉取到私有 registry，并通过设置 **Push Secret** 来启用推送。

8.5.3. Custom 策略选项

以下选项特定于 Custom 构建策略。

8.5.3.1. FROM 镜像

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

```
strategy:
  customStrategy:
    from:
      kind: "DockerImage"
      name: "registry.access.redhat.com/openshift3/ose-docker-builder"
```

8.5.3.2. 公开 Docker 套接字

为了能运行 Docker 命令并从容器内构建容器镜像，构建容器必须绑定到可访问的套接字。要做到这一点，将 **exposeDockerSocket** 选项设置为 **true** ：

```
strategy:
  customStrategy:
    exposeDockerSocket: true
```

8.5.3.3. Secrets

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。

每个 secret 都可以挂载到特定位置：

```
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。

8.5.3.3.1. 通过 Web 控制台添加 Secret

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. 创建自定义构建配置。
4. 在构建配置编辑器页面上或在 Web 控制台的 **fromimage** 页面中设置 **Source Secret**。
5. 点 **Save** 按钮。

8.5.3.3.2. 启用拉取和推送

通过在构建配置中设置 **Pull Secret** 来启用拉取到私有 registry，并通过设置 **Push Secret** 来启用推送。

8.5.3.4. 强制 Pull

默认情况下，在设置构建 pod 时，构建控制器会检查构建配置中指定的镜像是否本地可用。如果是这样，则使用该镜像。但是，要覆盖本地镜像并从镜像流指向的 registry 中刷新它，请创建一个 **BuildConfig**，将 **forcePull** 标志设为 **true** ：

```
strategy:
  customStrategy:
    forcePull: true 1
```

1 此标志会导致本地构建器镜像被忽略，并从镜像流指向的 registry 中拉取新版本。将 **forcePull** 设置为 **false** 将导致默认行为是遵守本地存储的镜像。

8.5.3.5. 环境变量

要将环境变量提供给 Custom 构建过程使用，您可以在 **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。

例如，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/"
```

集群管理员也可使用 Ansible 配置全局构建设置。

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

8.5.4. Pipeline 策略选项

以下选项特定于 Pipeline 构建策略。

8.5.4.1. 提供 Jenkinsfile

您可以通过两种方式之一提供 Jenkinsfile：

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***。

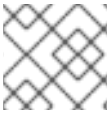
8.5.4.2. 环境变量

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。

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

例如：

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



注意

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

8.5.4.2.1. 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。而且，如果指定没有列在 **BuildConfig** 中的环境变量，它们将添加为 Jenkins 任务参数定义。此外，您从 Jenkins 控制台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。

如果通过 Jenkins 控制台启动 Jenkins 任务，则您可以通过 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

在 **BuildConfig** 中指定所有可能的环境变量和作业参数会减少磁盘 I/O，并在 Jenkins 处理过程中提高性能。

8.6. 构建环境

8.6.1. 概述

与 pod 环境变量一样，可以使用 Downward API 在引用其他资源/变量时定义构建环境变量。但是，有几个例外情况如下。

注意

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

8.6.2. 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**。

```
env:
  - name: FIELDREF_ENV
    valueFrom:
```

```
    fieldRef:
      fieldPath: metadata.name
```

注意

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

8.6.3. 使用容器资源作为环境变量

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。

8.6.4. 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。

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

8.7. 触发构建

8.7.1. 构建触发器概述

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

8.7.2. 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，会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，则针对 OpenShift Container Platform 构建签出 Webhook 事件中记录的准确提交引用。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
```

8.7.2.1. 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：

```
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/github
```

配置 GitHub Webhook：

1. 从 GitHub 存储库创建 **BuildConfig** 后，请运行：

```
$ 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/<secret>/gith
ub>.
```

2. 从 GitHub Web 控制台将此 URL 剪切并粘贴到 GitHub 中。
3. 在 GitHub 存储库中，从 **Settings → Webhooks & Services** 中选择 **Add Webhook**。
4. 将 URL 输出（与上方相似）粘贴到 **Payload URL** 字段。
5. 将 **Content Type** 从 GitHub 默认的 **application/x-www-form-urlencoded** 更改为 **application/json**。
6. 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 Gogs 服务器，您可以在 **BuildConfig** 中定义 GitHub Webhook 触发器，并通过 Gogs 服务器触发它。

根据包含有效 JSON 有效负载的文件，如 **payload.json**，您可以通过 **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/<s
ecret>/github
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

8.7.2.2. GitLab Webhooks

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：

```
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/gitlab
```

配置 GitLab Webhook：

1. 描述构建配置以获取 Webhook URL：

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

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

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

根据包含有效 JSON 有效负载的文件，如 **payload.json**，您可以通过 **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>/webhooks/<s
ecret>/gitlab
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

8.7.2.3. 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：

```
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/bitbucket
```

配置 Bitbucket Webhook：

1. 描述构建配置以获取 Webhook URL：

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

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

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

根据包含有效 JSON 有效负载的文件，如 **payload.json**，您可以通过 **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>/webhooks/<s
ecret>/bitbucket
```

只有在 API 服务器没有适当签名的证书时，才需要 **-k** 参数。

8.7.2.4. 通用 Webhook

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

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

1 设置为 **true**，以允许通用 Webhook 传入环境变量。

要设置调用者，请为调用系统提供构建的通用 Webhook 端点的 URL：

```
https://<openshift_api_host:port>/oapi/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<s
ecret>/generic
```

调用者必须以 **POST** 操作形式调用 Webhook。

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/<s
ecret>/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，通过 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。

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>/webhooks/<s
ecret>/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。

8.7.2.5. 显示 Webhook URL

使用以下命令来显示与构建配置关联的任何 Webhook URL:

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

如果上述命令不显示任何 Webhook URL，则不会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。

8.7.3. 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。

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。

在上例中，如果策略有镜像更改触发器，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。

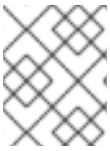
注意

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。

8.7.4. 配置更改触发器

通过配置更改触发器，您可以在创建新 **BuildConfig** 时立即自动调用构建。如下是 **BuildConfig** 中的示例触发器定义 YAML：

```
type: "ConfigChange"
```



注意

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

8.7.4.1. 手动设置触发器

您可以使用 **oc set triggers** 在构建配置中添加和移除触发器。例如，要在构建配置上设置 GitHub Webhook 触发器，请使用：

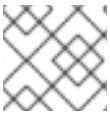
```
$ oc set triggers bc <name> --from-github
```

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

8.8. 构建 HOOK

8.8.1. 构建 hook 概述

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。

8.8.2. 配置 Post Commit 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**：

- 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。

- 将参数传递给默认入口点：

```
postCommit:
  args: ["bundle", "exec", "rake", "test", "--verbose"]
```

在这种形式中，**args** 是提供给镜像的默认入口点的参数列表。镜像入口点必须能够处理参数。

- Shell 脚本带有参数：

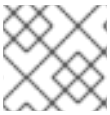
```
postCommit:
  script: "bundle exec rake test $1"
  args: ["--verbose"]
```

如果您需要在 shell 脚本中正确传递参数，请使用此表单。在 **script** 中，**$0** 将为 "/bin/sh"，**$1**，**$2** 等是 **args** 中的位置参数。

- 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。

8.8.2.1. 使用 CLI

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

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
```

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"
```

8.9. 构建运行策略

8.9.1. 构建运行策略概述

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。

8.9.2. 串行运行策略

将 **runPolicy** 字段设置为 **Serial** 将会导致从 **Build** 配置创建的所有新构建按顺序运行。这意味着每次只有一个构建运行，每次新构建将等候到上一构建完成。使用此策略将导致一致且可预测的构建输出。这是默认的 **runPolicy**。

使用 **Serial** 策略从 **sample-build** 配置触发三个构建将导致：

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git          New
sample-build-3   Source   Git          New
```

当 **sample-build-1** 构建完成时，**sample-build-2** 构建将运行：

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Completed 43 seconds ago   34s
```

```
sample-build-2   Source   Git@1aa381b   Running   2 seconds ago   2s
sample-build-3   Source   Git           New
```

8.9.3. SerialLatestOnly Run Policy

将 **runPolicy** 字段设置为 **SerialLatestOnly** 将使 **Build** 配置创建的所有新构建都按顺序运行，这与使用 **Serial** run 策略相同。区别在于，当当前运行的构建完成时，将运行的下一个构建是最新构建。换句话说，您不会等待排队的构建运行，因为跳过它们。跳过的构建标记为 **Cancelled**。此策略可用于快速迭代开发。

使用 **SerialLatestOnly** 策略从 **sample-build** 配置触发三个构建将导致：

```
NAME            TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git           Cancelled
sample-build-3   Source   Git           New
```

sample-build-2 构建将被取消(skipped)，在 **sample-build-1** 完成后运行的下一个构建将是 **sample-build-3** 构建：

```
NAME            TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Completed 43 seconds ago   34s
sample-build-2   Source   Git           Cancelled
sample-build-3   Source   Git@1aa381b   Running   2 seconds ago   2s
```

8.9.4. 并行运行策略

将 **runPolicy** 字段设置为 **Parallel** 会导致从 **Build** 配置创建的所有新构建并行运行。这可以产生无法预计的结果，因为第一次创建的构建可以最后完成，这会替换之前由最后一个构建生成的推送容器镜像。

在您不关心构建完成的顺序时，请使用并行 run 策略。

使用 **Parallel** 策略从 **sample-build** 配置触发三个构建将导致三个同时构建：

```
NAME            TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source   Git@689d111   Running   17 seconds ago   3s
```

无法保证完成顺序：

```
NAME            TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source   Git@689d111   Completed 17 seconds ago   5s
```

8.10. 高级构建操作

8.10.1. 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。通过在项目的默认容器限值中指定资源限值来限制这些资源。

您还可以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 单元表示：**100m** 代表 0.1 CPU 单元（100 * 1e-3）。

2 **内存** 以字节为单位：**256Mi** 代表 268435456 字节(256 * 2 ^ 20)。

但是，如果您的项目定义了 配额，则需要以下两项之一：

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

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

1 **requests** 对象包含与配额中资源列表对应的资源列表。

- 项目中定义的限值范围，其中 **LimitRange** 对象中的默认值应用到构建过程中创建的 pod。

否则，构建 Pod 创建将失败，说明无法满足配额要求。

8.10.2. 设置最大持续时间

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。

下例显示了 **BuildConfig** 的部分内容，它指定了值为 30 分钟的 **completionDeadlineSeconds** 字段：

```
spec:
  completionDeadlineSeconds: 1800
```

注意

Pipeline 策略选项不支持此设置。

8.10.3. 将构建分配给特定的节点

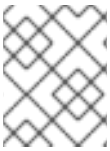
通过在构建配置的 **nodeSelector** 字段中指定标签，可以将构建定位到在特定节点上运行。**nodeSelector** 值是一组键/值对，在调度构建 Pod 时与 **node** 标签匹配。

```
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。

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。

如需更多信息，请参阅配置全局构建默认值和覆盖。



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。

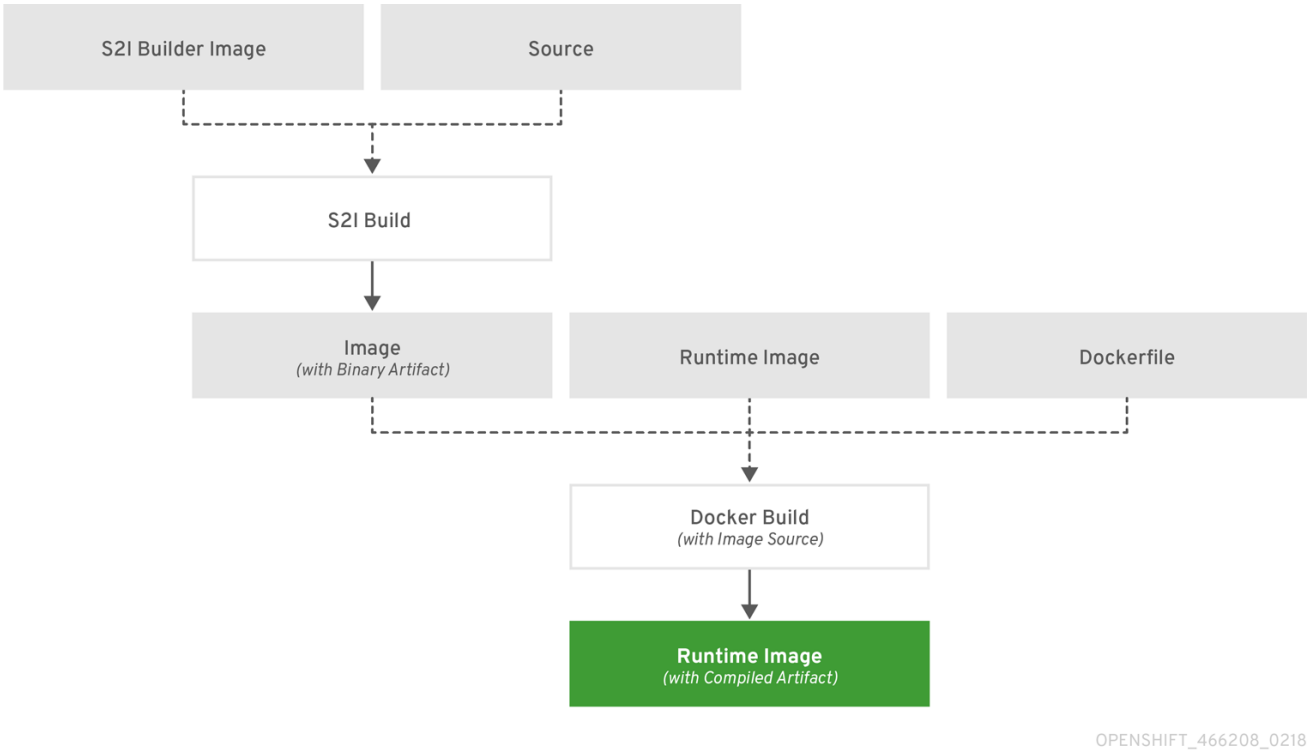
8.10.4. 串联构建

对于编译语言（Go、C、C ++、Java 等），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
    type: Git
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
    type: Source
```

第二个构建使用带有指向第一个构建中输出镜像内的 WAR 文件的路径的镜像源。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:
  type: Dockerfile
  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
  type: Docker
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。

8.10.5. 构建修剪

默认情况下，生命周期已结束的构建将无限期保留。您可以通过为 **successfulBuildsHistoryLimit** 或 **failedBuildsHistoryLimit** 提供正整数值来限制保留的旧构建数量，如以下示例构建配置中所示。

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

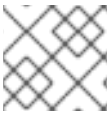
1 **successfulBuildsHistoryLimit** 将保留最多两个状态为 **completed** 的构建。

2 **failedBuildsHistoryLimit** 将保留最多两个状态为 **failed**, **cancelled**, 或 **error** 的构建。

构建修剪由以下操作触发：

- 更新构建配置。
- 构建可以完成其生命周期。

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' object pruning command 来手动修剪构建。

8.11. 构建故障排除

8.11.1. 请求对资源的访问

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
```

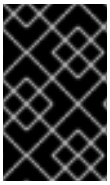
第 9 章 DEPLOYMENTS

9.1. 部署如何工作

9.1.1. 部署是什么？

OpenShift Container Platform 部署提供对常见用户应用程序的精细管理。它们使用三个独立的 API 对象进行描述：

- 部署配置，它将应用程序特定组件的所需状态描述为 pod 模板。
- 一个或多个复制控制器，其中包含部署配置状态作为 pod 模板的时间点记录。
- 一个或多个 pod，表应用程序某一特定版本的实例。



重要

用户不需要操作由部署配置拥有的复制控制器或 pod。部署系统可确保正确传播对部署配置的更改。如果现有部署策略不适用于您的用例，而且您需要在部署的生命周期内执行手动步骤，那么应考虑创建自定义策略。

在创建部署配置时，会创建一个复制控制器来代表部署配置的 pod 模板。如果部署配置更改，则使用最新的 pod 模板创建一个新的复制控制器，并运行部署过程来缩减旧复制控制器并扩展新的复制控制器。

在创建时，自动从服务负载均衡器和路由器中添加和移除应用程序的实例。只要您的应用程序支持收到 **TERM** 信号时 安全关闭，您可以确保运行的用户连接能够正常完成。

部署系统提供的功能：

- 部署配置，这是用于运行应用程序的模板。
- 为响应事件而触发自动化部署的触发器。
- User-customizable strategies，用于从上一版本过渡到新版本。在 pod 内运行的策略，通常称为部署过程。
- 一组 hook，用于在部署生命周期的不同点上执行自定义行为。
- 应用程序的版本控制，以便在部署失败时支持手动或自动的回滚。
- 手动复制扩展和自动扩展。

9.1.2. 创建部署配置

部署配置是 **deploymentConfig** OpenShift Container Platform API 资源，可以像任何其他资源一样通过 **oc** 命令进行管理。以下是 **deploymentConfig** 资源的示例：

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend"
spec:
  template: (1)
    metadata:
```

```
    labels:
      name: "frontend"
  spec:
    containers:
      - name: "helloworld"
        image: "openshift/origin-ruby-sample"
        ports:
          - containerPort: 8080
            protocol: "TCP"
replicas: 5 (2)
triggers:
  - type: "ConfigChange" (3)
  - type: "ImageChange" (4)
    imageChangeParams:
      automatic: true
      containerNames:
        - "helloworld"
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
strategy: (5)
  type: "Rolling"
paused: false (6)
revisionHistoryLimit: 2 (7)
minReadySeconds: 0 (8)
```

1. **frontend** 部署配置的 pod 模板描述了一个简单的 Ruby 应用程序。
2. **frontend** 将有 5 个副本。
3. 配置更改触发器会在每次 Pod 模板更改时创建新复制控制器。
4. 镜像更改触发器会在每次有新版本的 **origin-ruby-sample:latest** 镜像流标签时触发新复制控制器的创建。
5. Rolling 策略是部署 Pod 的默认方法。可以省略。
6. 暂停部署配置。这将禁用所有触发器的功能，并允许在实际部署 pod 模板前对 pod 模板进行多次更改。
7. 修订历史记录限制是您要保留的旧复制控制器的限制，用于回滚。可以省略。如果省略，则不会清理旧的复制控制器。
8. pod 被视为可用前等待的最短秒数（在就绪度检查成功后）。默认值为 0。

9.2. 基本部署操作

9.2.1. 启动部署

您可以使用 Web 控制台或 CLI 手动启动新的部署过程：

```
$ oc rollout latest dc/<name>
```

注意

如果部署过程已在进行中，命令将显示一条消息，不会部署新的复制控制器。

9.2.2. 查看部署

要获取有关应用程序所有可用修订的基本信息：

```
$ oc rollout history dc/<name>
```

这将显示有关所有最近为提供的部署配置创建的复制控制器的详细信息，包括任何当前运行的部署过程。

您可以使用 **--revision** 标志查看特定于修订版本的详情：

```
$ oc rollout history dc/<name> --revision=1
```

如需有关部署配置和最新修订的详细信息：

```
$ oc describe dc <name>
```

注意

Web 控制台在 **Browse** 标签页中显示部署。

9.2.3. 重试部署

如果部署配置的当前修订无法部署，您可以使用以下方法重启部署过程：

```
$ oc rollout retry dc/<name>
```

如果成功部署了最新的修订，命令会显示一条消息，且不会重试部署过程。

注意

重试部署会重启部署过程，且不创建新的部署修订。重启的复制控制器将具有与失败时相同的配置。

9.2.4. 回滚部署

回滚将应用恢复到上一修订，可通过 REST API、命令行或 Web 控制台进行。

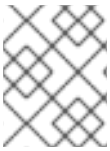
回滚到配置的最近一次部署成功的修订：

```
$ oc rollout undo dc/<name>
```

将恢复部署配置的模板以匹配 undo 命令中指定的部署修订，并且会启动新的复制控制器。如果没有通过 **--to-revision** 指定修订，则使用上一次成功部署的版本。

在回滚过程中，部署配置的镜像更改触发器会被禁用，以防止在回滚完成不久后意外启动新的部署过程。重新启用镜像更改触发器：

```
$ oc set triggers dc/<name> --auto
```



注意

部署配置也支持最新部署过程失败时自动回滚到配置的最近一次成功修订。这时，系统会原样保留部署失败的最新模板，由用户来修复其配置。

9.2.5. 在容器内执行命令

您可以为容器添加命令，用来覆盖决镜像的 **ENTRYPOINT** 设置来改变容器的启动行为。这与生命周期 hook 不同，后者在每个部署的指定时间点上运行一次。

在部署配置的 **spec** 字段中添加 **command** 参数。您也可以添加 **args** 字段来修改 **command**（如果 **command** 不存在，则修改 **ENTRYPOINT**）。

```
...
spec:
  containers:
    -
    name: <container_name>
    image: 'image'
    command:
      - '<command>'
    args:
      - '<argument_1>'
      - '<argument_2>'
      - '<argument_3>'
...
```

例如，使用 **-jar** 和 */opt/app-root/springboots2idemo.jar* 参数执行 **java** 命令：

```
...
spec:
  containers:
    -
    name: example-spring-boot
    image: 'image'
    command:
      - java
    args:
      - '-jar'
      - /opt/app-root/springboots2idemo.jar
...
```

9.2.6. 查看部署日志

输出给定部署配置的最新修订的日志：

```
$ oc logs -f dc/<name>
```

如果最新的修订正在运行或失败，**oc logs** 将返回负责部署 pod 的进程的日志。如果成功，**oc logs** 会从应用程序的 pod 返回日志。

您还可以查看来自旧的失败部署进程的日志，只要存在这些进程（旧的复制控制器及其部署器 pod）并且没有手动清理或删除：

```
$ oc logs --version=1 dc/<name>
```

有关获取日志的更多选项，请参阅：

```
$ oc logs --help
```

9.2.7. 设置部署触发器

部署配置可以包含触发器，推动创建新部署过程以响应集群内的事件。

警告

如果在部署配置上没有定义任何触发器，则默认添加 **ConfigChange** 触发器。如果触发器定义为空白字段，则必须手动启动部署。

9.2.7.1. 配置更改触发器

每当部署配置的容器集模板中检测到更改时，**ConfigChange** 触发器都会生成新的复制控制器。

注意

如果在部署配置上定义了 **ConfigChange** 触发器，则在部署配置本身创建后会自动创建第一个复制控制器，且不会暂停。

一个 ConfigChange 触发器

```
triggers:
  - type: "ConfigChange"
```

9.2.7.2. ImageChange Trigger

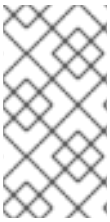
ImageChange 触发器会在镜像流标签内容更改时（推送镜像的新版本时）产生新的复制控制器。

ImageChange 触发器

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true (1)
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

(1) 如果 **imageChangeParams.automatic** 字段设置为 **false**，则触发器被禁用。

在上例中，当 **origin-ruby-sample** 镜像流的 **latest** 标签值更改并且新镜像值与部署配置的 **helloworld** 容器中指定的当前镜像不同时，会使用 **helloworld** 容器的新镜像创建新的复制控制器。



注意

如果在部署配置上定义了 **ImageChange** 触发器（带有 **ConfigChange** 触发器和 **automatic=false**，或者 **automatic=true**）并且 **ImageChange** 触发器指向的 **ImageStreamTag** 尚不存在，则初始部署过程将在镜像导入时或由构建推送到 **ImageStreamTag** 时立即自动启动。

9.2.7.2.1. 使用命令行

oc set triggers 命令可以用来为部署配置设置部署触发器。在上例中，您可以使用以下命令设置 **ImageChangeTrigger** ：

```
$ oc set triggers dc/frontend --from-image=myproject/origin-ruby-sample:latest -c helloworld
```

如需更多信息，请参阅：

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

9.2.8. 设置部署资源

部署由部署 Pod 完成。默认情况下，部署 pod 会在调度它的计算节点上消耗无限的节点资源。在大多数情况下，未绑定资源消耗不会导致问题，因为部署 pod 会消耗较低资源，并在短时间内运行。如果项目指定了默认容器限值，则部署 Pod 使用的资源以及其他 pod 数（针对这些限制）。

您可以通过在部署配置中通过部署策略限制部署 Pod 使用的资源。部署 pod 的资源限制可与 Recreate、Rolling 或 Custom 部署策略一起使用。

注意

只有在管理员启用了临时存储技术预览功能时，才能限制 临时存储。此功能默认为禁用。

在以下示例中，**resources**、**cpu**、**memory** 和 **ephemeral-storage** 中每一个都是可选的：

```
type: "Recreate"
resources:
  limits:
    cpu: "100m" (1)
    memory: "256Mi" (2)
    ephemeral-storage: "1Gi" (3)
```

1. **cpu** 以 CPU 单元表示：**100m** 代表 0.1 CPU 单元（100 * 1e-3）。
2. **内存** 以字节为单位：**256Mi** 代表 268435456 字节(256 * 2 ^ 20)。
3. **ephemeral-storage** 以字节为单位：**1Gi** 表示 1073741824 字节(2 ^ 30)。只有管理员启用了临时存储技术预览功能时，才会提供 **ephemeral** -storage 参数。

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：

```
type: "Recreate"
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
    ephemeral-storage: "1Gi"
```

1 **requests** 对象包含与配额中资源列表对应的资源列表。

请参阅 quota 和 Limit Ranges，以了解更多有关计算资源和请求与限值之间的区别的信息。

- 项目中定义的限值范围，其中 **LimitRange** 对象中的默认值应用到部署过程中创建的 pod。

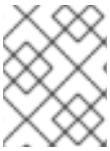
否则，部署 pod 创建将失败，显示无法满足配额要求。

9.2.9. 手动扩展

除了回滚外，您还可以对 Web 控制台中的副本数量进行精细控制，或者使用 **oc scale** 命令进行精细的控制。例如，以下命令将部署配置 **frontend** 中的副本设置为 3。

```
$ oc scale dc frontend --replicas=3
```

副本数量最终会传播到部署配置 **frontend** 配置的部署的预期和当前状态。

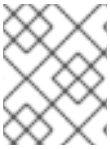


注意

也可以使用 **oc autoscale** 命令自动扩展 Pod。如需了解更多详细信息，请参阅 Pod 自动扩展。

9.2.10. 将 Pod 分配给特定的节点

您可以结合使用节点选择器和带标签的节点来控制 pod 的放置。



注意

OpenShift Container Platform 管理员可以在集群安装过程中分配标签，或者在安装后添加到节点。

集群管理员可为项目设置默认节点选择器，以便将 pod 放置限制到特定的节点。作为 OpenShift Container Platform 开发人员，您可以设置 pod 配置的节点选择器来进一步限制节点。

要在创建 pod 时添加节点选择器，请编辑 pod 配置并添加 **nodeSelector** 值。这可添加到单个 pod 配置中，也可以添加到 pod 模板中：

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
...
```

具有节点选择器时创建的 Pod 会分配给带有指定标签的节点。

这里指定的标签将与集群管理员添加的标签结合使用。

例如，如果项目中包含由集群管理员添加的 **type=user-node** 和 **region=east** 标签，并且您将上述 **disktype: ssd** 标签添加到某一 pod，则 pod 只会调度到具有所有这三个标签的节点上。

注意

标签只能设置为一个值，因此在具有管理员设置的默认值 **region=east** 的 pod 配置中设置节点选择器 **region=west**，这会导致 pod 永不会被调度。

9.2.11. 使用不同服务帐户运行 Pod

您可以使用非默认服务帐户运行 pod：

1. 编辑部署配置：

```
$ oc edit dc/<deployment_config>
```

2. 将 **serviceAccount** 和 **serviceAccountName** 参数添加到 **spec** 字段，再指定您要使用的服务帐户：

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

9.2.12. 从 Web 控制台将 secret 添加到部署配置

将 secret 添加到部署配置中，以便它可以访问私有存储库。

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镜像存储库的凭证的 secret。
3. 创建部署配置。
4. 在部署配置编辑器页面中，或在 Web 控制台的 **fromimage** 页面中设置 **Pull Secret**。
5. 点 **Save** 按钮。

9.3. 部署策略

9.3.1. 什么是部署策略？

部署策略是更改或升级应用程序的方法。其目的是在无需停机的前提下进行修改，从而使用户几乎不会注意到这些变化。

最常见的策略是使用蓝绿部署。新版本（蓝色版本）上线进行测试和评估，同时用户仍然使用稳定版本（绿色版本）。准备就绪后，用户切换到蓝色版本。如果出现问题，您可以切回到绿色版本。

常见的替代策略是使用同时活跃的 A/B 版本，有些用户使用一个版本，而一些用户使用另一个版本。这可用于试验用户界面变化和其他功能，以获取用户反馈。它还可用来在影响有限用户的生产环境中验证正确的操作。

Canary 部署会测试新版本，但在检测到问题时，迅速回退到上一版本。这可以通过以上两个策略实现。

基于路由的部署策略不会缩放服务中的 pod 数量。为了保持所需的性能特性，部署配置可能需要扩展。

选择部署策略时需要考虑一些事项。

- 长时间运行的连接需要被安全处理。
- 数据库转换会很棘手，需要与应用程序一起执行并回滚。
- 如果应用是微服务混合且传统组件停机时间，则可能需要完成转换。
- 您需要基础架构来做到这一点。
- 如果您的测试环境没有被隔离，则可能会破坏到新版本和旧版本。

由于最终用户通常通过由路由器控制的路由访问应用程序，因此部署策略侧重于部署配置的功能或路由功能。

注重部署配置的策略会影响所有使用该应用程序的路由。侧重于路由功能的策略则会影响到单个的路由。

许多部署策略通过部署配置支持，一些额外的策略则通过路由器功能支持。本节将讨论基于部署配置的策略。

- Rolling 策略和 Canary 部署
- Recreate 策略
- Custom 策略
- 使用路由 进行蓝绿部署
- 使用路由进行/B Deployment 和 Canary 部署
- 一个 Service，多个部署配置

如果在部署配置上没有指定任何策略，则 Rolling 策略是使用的默认策略。

部署策略使用 就绪度检查 来确定新 pod 是否准备就绪。如果就绪度检查失败，部署配置将重试运行 Pod，直到超时为止。默认超时为 **10m**，其值在 **dc.spec.strategy.*params** 的 **TimeoutSeconds** 中设置。

9.3.2. Rolling 策略

滚动部署会逐渐将应用程序旧版本实例替换为应用程序的新版本实例。在缩减旧组件前，滚动部署通常会借助 **readiness check** 等待新 Pod 变为 **ready**。如果发生严重问题，可以中止 Rolling 部署。

9.3.2.1. Canary 部署

OpenShift Container Platform 中的所有滚动部署都属于 *Canary 部署*；在替换所有旧实例前测试新的版本（Canary）。如果就绪度检查永不成功，则移除 Canary 实例，并且自动回滚部署配置。就绪度检查是应用代码的一部分，可能根据需要复杂，以确保新实例就绪可用。如果您需要对应用程序进行更复杂的检查（如向新实例发送真实用户工作负载），请考虑实施自定义部署 或使用蓝绿部署 策略。

9.3.2.2. 使用 Rolling 部署

- 希望在应用程序更新过程中不需要停机时。
- 应用程序同时支持运行旧代码和新代码时。

滚动部署意味着您同时运行旧版和新版本的代码。这通常需要您的应用程序可以处理 N-1 兼容性。

以下是 Rolling 策略的示例：

```
strategy:
  type: Rolling
  rollingParams:
    updatePeriodSeconds: 1 (1)
    intervalSeconds: 1 (2)
    timeoutSeconds: 120 (3)
    maxSurge: "20%" (4)
    maxUnavailable: "10%" (5)
    pre: {} (6)
    post: {}
```

1. 各个 pod 更新之间等待的时间。如果未指定，则默认值为 **1**。
2. 更新后轮询部署状态之间等待的时间。如果未指定，则默认值为 **1**。
3. 放弃前等待扩展事件的时间。可选，默认值为 **600**。这里的*放弃*表示自动回滚到以前的完整部署。
4. **maxSurge** 是可选的；如果未指定，则默认为 **25%**。请参考以下流程中的信息。
5. **maxUnavailable** 是可选的；如果未指定，则默认为 **25%** 。请参考以下流程中的信息。
6. **pre** 和 **post** 都是 生命周期 hook。

Rolling 策略将：

1. 执行任何 **pre** 生命周期 hook。
2. 根据数量扩展新的复制控制器。
3. 根据最大不可用数，缩减旧的复制控制器。
4. 重复这个扩展，直到新的复制控制器达到所需的副本数，并且旧的复制控制器已缩减为零。
5. 执行任何 **post** 生命周期 hook。

重要

在缩减时，Rolling 策略会等待 pod 准备就绪，以便它能决定进一步缩放是否会影响到可用性。如果扩展 pod 永不就绪，部署过程将最终超时并导致部署失败。

maxUnavailable 参数是在更新过程中不可用的 pod 的最大数量。**maxSurge** 参数是原始 pod 数量之上最多可以调度的 pod 数量。这两个参数可以设定为百分比（如 **10%**）或绝对值（如 **2**）。两者的默认值都是 **25%**。

这些参数允许对部署的可用性和速度进行调优。例如：

- **maxUnavailable=0** 和 **maxSurge=20%** 可确保在更新和快速扩展过程中保持全部容量。
- **maxUnavailable=10%** 和 **maxSurge** =0 使用没有额外容量（原位更新）来执行更新。
- **maxUnavailable=10%** 和 **maxSurge=10%**，会快速扩展和缩减容量损失。

一般而言，如果您想要快速推出部署，请使用 **maxSurge**。如果您需要考虑资源配额并可以接受资源部分不可用的情况，请使用 **maxUnavailable**。

9.3.2.3. 滚动示例

在 OpenShift Container Platform 中，Rolling 部署是默认设置。要查看滚动更新，请按照以下步骤执行：

1. 根据 DockerHub 中找到的示例部署镜像创建一个应用程序：

```
$ oc new-app openshift/deployment-example
```

   如果您安装了路由器，请通过路由（或直接使用服务 IP）提供应用程序。

```
$ oc expose svc/deployment-example
```

   通过 **deployment-example.<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

2. 将部署配置扩展至三个副本：

```
$ oc scale dc/deployment-example --replicas=3
```

3. 通过为示例的新版本标上 **latest** 标签（tag），自动触发新部署：

```
$ oc tag deployment-example:v2 deployment-example:latest
```

4. 在浏览器中刷新页面，直到您看到 **v2** 镜像。
5. 如果使用 CLI，以下命令会显示版本 1 上的 pod 数以及版本 2 上的数量。在 Web 控制台中，您应该会看到 pod 缓慢添加到 v2 中，并从 v1 中删除。

```
$ oc describe dc deployment-example
```

在部署过程中，新复制控制器以递增方式扩展。新 pod 标记为 **ready**（通过就绪度检查）后，部署过程将继续。如果 pod 尚未就绪，该过程将中止，并且部署配置将回滚到之前的版本。

9.3.3. Recreate 策略

Recreate 策略具有基本的推出部署行为，并支持用于注入代码到部署流程的生命周期 hook。

以下是 Recreate 策略的示例：

```
strategy:
  type: Recreate
  recreateParams: 1
```

```
    pre: {} 2
    mid: {}
    post: {}
```

1 **recreateParams** 可选。

2 **pre**、**mid** 和 **post** 是 生命周期 hook。

Recreate 策略将：

1. 执行任何 **pre** 生命周期 hook。
2. 将上一部署缩减为零。
3. 执行任何 **mid** 生命周期 hook。
4. 扩展新部署。
5. 执行任何 **post** 生命周期 hook。

重要

在扩展过程中，如果部署副本数大于一，则先对部署的第一副本进行就绪状态验证，然后再全面扩展部署。如果第一副本验证失败，部署将被视为失败。

9.3.3.1. 何时使用 Recreate 部署

- 需要在新代码启动前进行迁移或进行其他数据转换时。
- 不支持同时运行应用程序代码的新旧版本时。
- 当使用 RWO 卷时，不支持在多个副本间共享该卷。

重新创建部署会导致停机，这是因为在短时间内没有运行应用程序实例。然而，旧代码和新代码不会被同时运行。

9.3.4. Custom 策略

Custom 策略允许您提供自己的部署行为。

以下是 Custom 策略的示例：

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
      - name: ENV_1
        value: VALUE_1
```

在上例中，**organization/strategy** 容器镜像提供部署行为。可选的 **command** 数组覆盖镜像的 *Dockerfile* 中指定的任何 **CMD** 指令。提供的可选环境变量添加到策略过程的执行环境中。

另外，OpenShift Container Platform 为部署过程提供以下环境变量：

环境变量	描述
OPENSHIFT_DEPLOYMENT_NAME	新部署（复制控制器）的名称。
OPENSHIFT_DEPLOYMENT_NAMESPACE	新部署的命名空间。

新部署的副本数最初为零。该策略负责使新部署积极使用最能满足用户需求的逻辑。

了解有关 高级部署策略 的更多信息.

另外，也可使用 **customParams** 将自定义部署逻辑注入现有的部署策略中。提供自定义 shell 脚本逻辑并调用 **openshift-deploy** 二进制文件。用户不必提供自定义部署器容器镜像，但需要使用默认的 OpenShift Container Platform 部署器镜像：

```
strategy:
  type: Rolling
  customParams:
    command:
    - /bin/sh
    - -c
    - |
      set -e
      openshift-deploy --until=50%
      echo Halfway there
      openshift-deploy
      echo Complete
```

这会导致以下部署：

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-1 down to 1
    Scaling custom-deployment-2 up to 2
    Scaling custom-deployment-1 down to 0
--> Success
Complete
```

如果自定义部署策略过程需要访问 OpenShift Container Platform API 或 Kubernetes API，执行该策略的容器可以使用容器中的服务帐户令牌进行身份验证。

9.3.5. 生命周期 Hook

Recreate 和 Rolling 策略支持生命周期 hook，它允许在策略的预定义点将行为注入到部署过程中：

以下是 **pre** 生命周期 hook 示例：

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1 **execNewPod** 是基于 pod 的生命周期 hook。

每个 hook 都有 **failurePolicy**，定义在遇到 hook 失败时策略应执行的操作：

Abort	如果 hook 失败，部署过程将被视为失败。
Retry	应重试 hook 执行过程，直到成功为止。
Ignore	所有 hook 失败都应忽略，部署应继续进行。

Hook 具有特定类型的字段，用于描述如何执行 Hook。目前，pod-based hooks 是唯一受支持的 hook 类型，通过 **execNewPod** 字段指定。

9.3.5.1. 基于 Pod 的生命周期 Hook

基于 Pod 的生命周期 hook 在从部署配置中模板派生的新 pod 中执行 hook 代码。

以下简化的部署配置示例使用了 Rolling 策略。为简明起见，省略了触发器和其他一些次要的细节：

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld 1
          command: [ "/usr/bin/command", "arg1", "arg2" ] 2
          env: 3
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data 4
```

■

1. **helloworld** 名称指代 **spec.template.spec.containers[0].name**。
2. 此 **command** 覆盖 **openshift/origin-ruby-sample** 镜像中定义的任何 **ENTRYPOINT**。
3. **env** 是 hook 容器的一组可选环境变量。
4. **volumes** 是 hook 容器的一组可选的卷引用。

在本例中，将使用 **helloworld** 容器中的 **openshift/origin-ruby-sample** 镜像在新 pod 中执行 **pre** hook。hook pod 将具有以下属性：

- hook 命令将是 **/usr/bin/command arg1 arg2**。
- hook 容器将具有 **CUSTOM_VAR1=custom_value1** 环境变量。
- hook 失败策略是 **Abort**，这意味着如果 hook 失败，部署过程将失败。
- hook pod 将从部署配置 pod 中继承 **data** 卷。

9.3.5.2. 使用命令行

oc set deployment-hook 命令可用于为部署配置设置部署 hook。在上例中，您可以使用以下命令设置部署前 hook：

```
$ oc set deployment-hook dc/frontend --pre -c helloworld -e CUSTOM_VAR1=custom_value1 \
  -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

9.4. 高级部署策略

9.4.1. 高级部署策略

部署策略 为应用程序的演进提供了一个途径。有些策略使用 部署配置 进行解析到应用程序的所有路由用户可见的更改。其他策略，如此处描述的那样，使用路由器功能影响特定路由。

9.4.2. 蓝绿部署

蓝绿部署涉及同时运行应用程序的两个版本，并将流量从生产版本（绿色版本）移动到更新版本（蓝色版本）。您可以使用 滚动策略 或切换路由中的服务。

注意

由于许多应用程序依赖于持久数据，因此您将需要有一个支持 N-1 兼容性 的应用程序，这意味着您可以通过创建数据层的两个副本在数据库、存储或磁盘间共享数据并实现实时迁移。

以测试新版本时使用的数据为例。如果是生产数据，新版本中的错误可能会破坏生产版本。

9.4.2.1. 使用蓝绿部署

蓝绿部署使用两个部署配置。这两者都在运行，生产环境中的部署配置依赖于路由指定的服务，每个部署配置公开给不同的服务。您可以创建指向新版本的新路由并进行测试。准备就绪后，将生产路由中的服务更改为指向新服务和新的 blue 版本。

如果需要，可以通过将服务切回到之前的版本以回滚到老的绿色版本。

使用路由和两个服务
这个示例设置了两个部署配置：一个用于稳定版本（绿色版本），另一个用于较新的版本（蓝色版本）。

路由指向某个服务，可以随时更改为指向不同服务。作为开发人员，您可以在生产流量路由到前连接到新服务来测试新版代码。

路由适用于 Web（HTTP 和 HTTPS）流量，因此这种技术最适合 Web 应用程序。

1. 创建示例应用程序的两个副本：

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

   这会创建两个独立的应用程序组件：一个在 **example-green** 服务下运行 **v1** 镜像，另一个使用 **example-blue** 服务下的 **v2** 镜像。

2. 创建指向旧服务的路由：

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. 通过 **example-green.<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

4. 编辑路由并将服务名称改为 **example-blue**：

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

5. 要验证路由是否已改变，请刷新浏览器直到您看到 **v2** 镜像。

9.4.3. A/B 部署

A/B 部署策略允许您在生产环境中以有限的方式尝试应用程序的新版本。您可以指定生产版本获得大多数用户请求，同时让有限比例的请求进入新版本。由于您控制对每个版本的请求部分，在测试过程中，您可以增加对新版本的请求的比例，最终停止使用上一版本。当您调整每个版本的请求负载时，可能需要扩展各个服务中的 pod 数量，以提供预期的性能。

除了升级软件外，您还可以使用此功能来试验用户界面的不同版本。由于部分用户会使用旧版本，而另外的一部分用户会使用新版本，因此您可以评估用户对不同版本的反应，以做出明智的设计决策。

要实现这一目的，旧的和新版本需要足够相似，两者可以同时运行。这常用于对程序错误修复的发布，也适用于新功能不会影响到旧功能的情况。版本需要 N-1 兼容性 才能正常工作。

OpenShift Container Platform 通过 Web 控制台和命令行界面支持 N-1 兼容性。

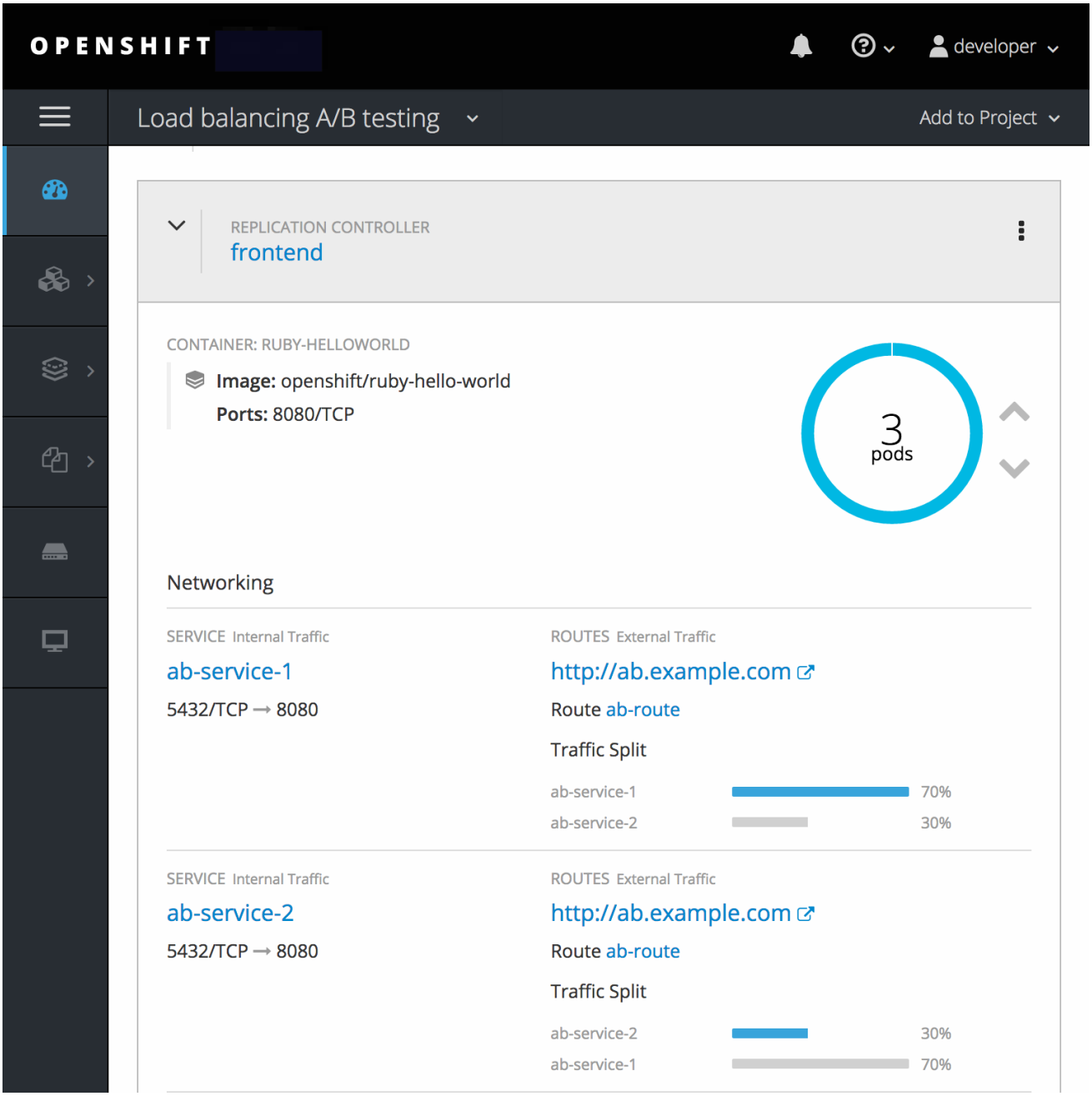
9.4.3.1. A/B 测试的负载均衡

用户设置使用多个服务设置路由。每个服务负责应用程序的一个版本。

每个服务分配到一个 **weight**，进入每个服务的请求的比例等于 **service_weight** 除以 **sum_of_weights**。每个服务的 **weight** 分布到该服务的端点，使得端点 **weight** 的总和等于服务 **weight**。

路由最多可有四个服务。服务的 **weight** 可以在 **0** 到 **256** 范围内。当 **weight** 等于 **0** 时，服务不参与负载均衡，但继续为现有的持久连接服务。当服务 **weight** 不为 **0** 时，每个端点的最小 **weight** 为 **1**。因此，拥有大量端点的服务会得到高于必要值的 **weight**。这时，可以减少 pod 数量来获得所需的负载均衡 **weight**。如需更多信息，请参阅 Alternate Backends 和 Weights 部分。

Web 控制台允许用户设置权重并在它们之间显示平衡：



设置 A/B 环境：

1. 创建两个应用程序并使用不同的名称。每一个都会创建一个部署配置。应用程序是同一程序的不同版本；一个是当前生产版本，另一个是提议的新版本：

```
$ oc new-app openshift/deployment-example1 --name=ab-example-a
$ oc new-app openshift/deployment-example2 --name=ab-example-b
```

2. 公开部署配置以创建服务：

```
$ oc expose dc/ab-example-a --name=ab-example-A
$ oc expose dc/ab-example-b --name=ab-example-B
```

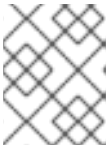
此时两个应用都已部署，并且正在运行并且具有服务。

3. 通过路由对外提供应用程序。此时可以公开任何服务，公开当前生产版本和后来修改路由以添加新版本。

```
$ oc expose svc/ab-example-A
```

通过 **ab-example.<project>.<router_domain>** 访问应用程序，验证您能否看到所需的版本。

4. 当您部署路由时，路由器将根据为服务指定的 **weight** 来均衡流量。此时，有一个带有默认 **weight=1** 的单个服务，因此所有请求都会进入该服务。将其他服务添加为 **alternateBackend**，调整 **weight** 会使 A/B 设置生效。这可通过 **oc set route-backends** 命令或编辑路由来完成。将 **oc set route-backend** 设置为 0 意味着服务不参与负载均衡，而是继续为现有的持久连接服务。



注意

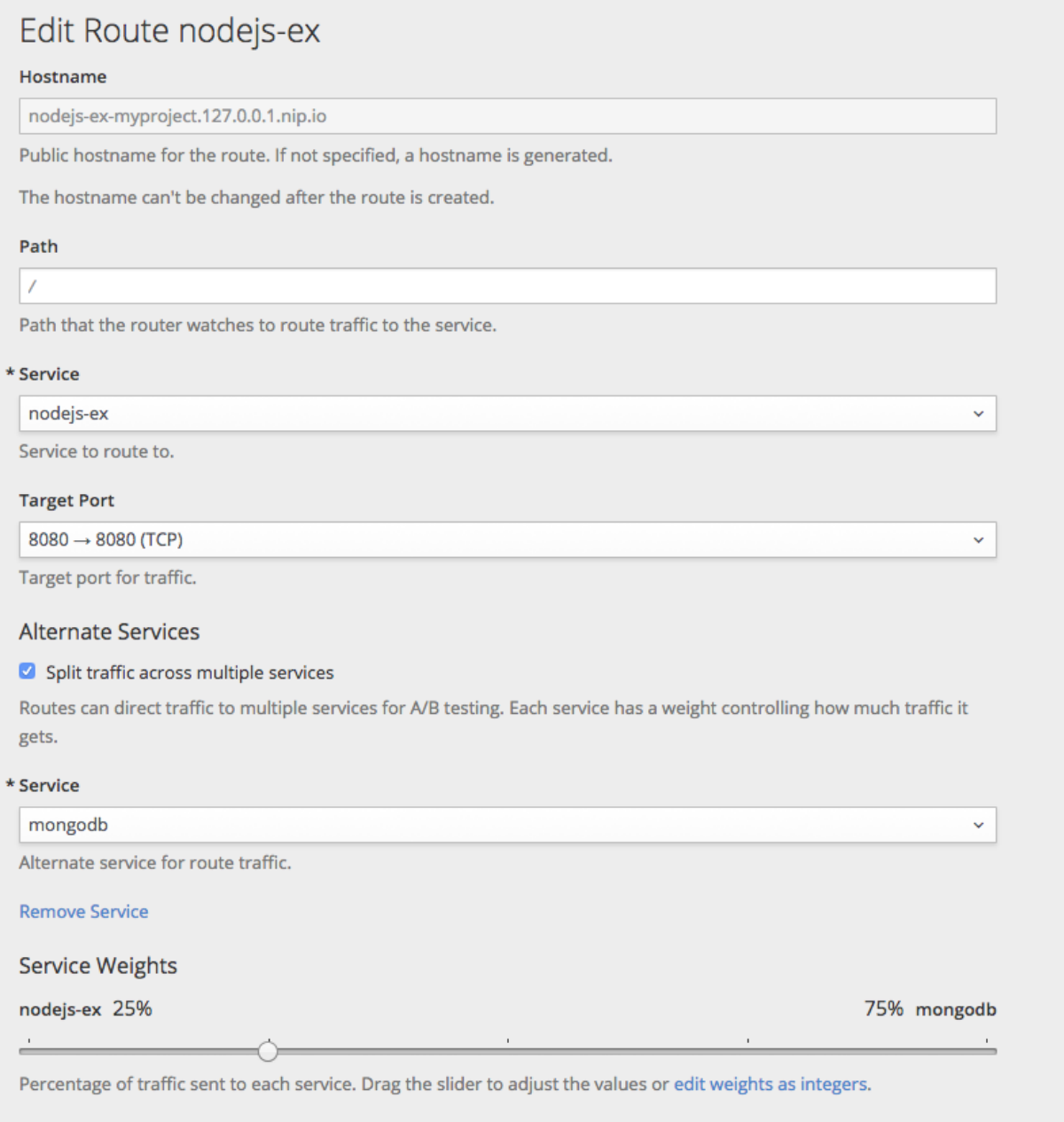
对路由的更改只会改变流量进入各个服务的比例。您可能需要扩展部署配置，以调整 pod 数量，以处理预期的负载。

若要编辑路由，请运行：

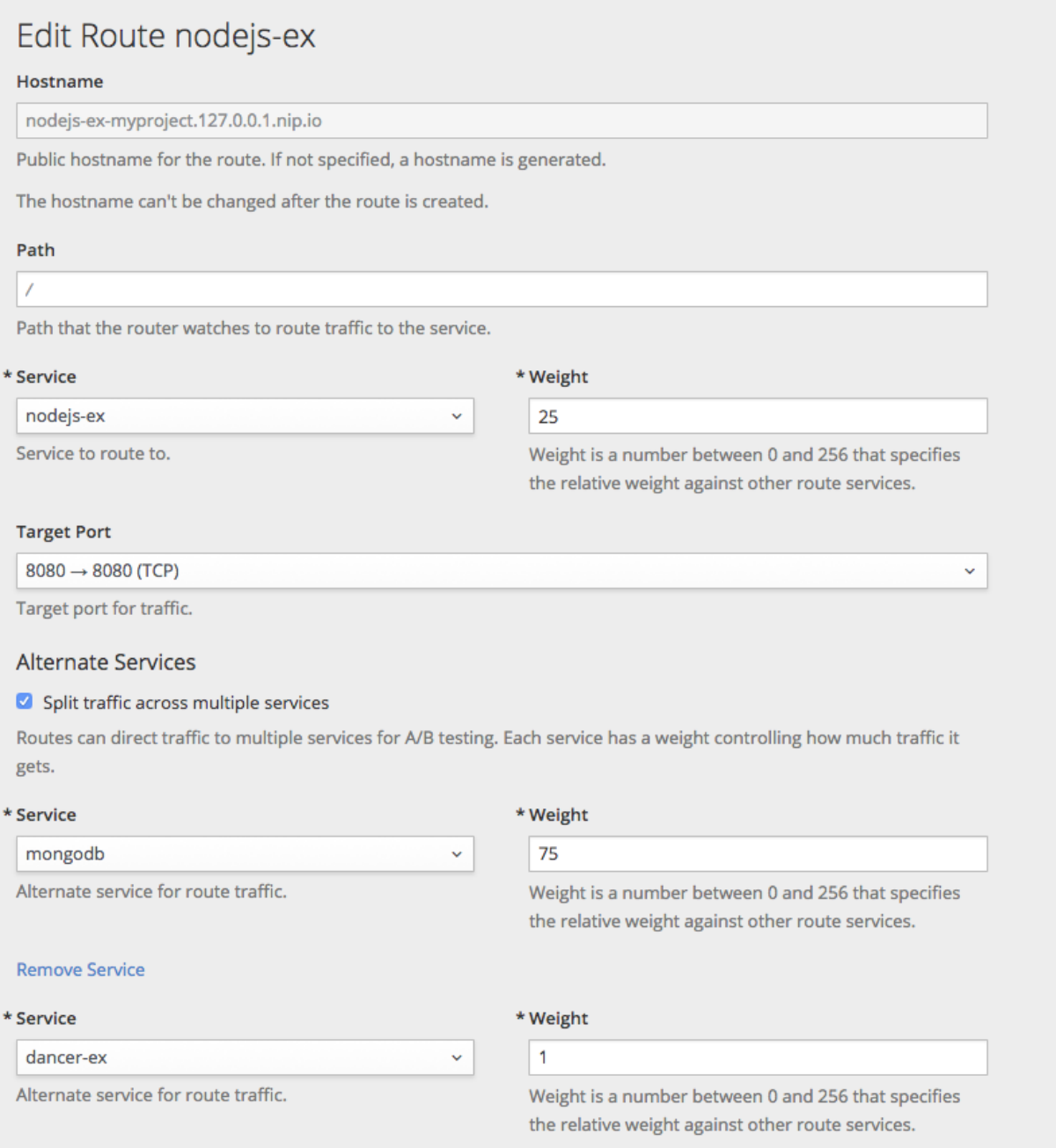
```
$ oc edit route <route-name>
...
metadata:
  name: route-alternate-service
  annotations:
    haproxy.router.openshift.io/balance: roundrobin
spec:
  host: ab-example.my-project.my-domain
  to:
    kind: Service
    name: ab-example-A
    weight: 10
  alternateBackends:
  - kind: Service
    name: ab-example-B
    weight: 15
...
```

9.4.3.1.1. 使用 Web 控制台管理 Weights

1. 导航到 Route 详情页面 (Applications/Routes)。
2. 从 Actions 菜单中选择 **Edit**。
3. 选中 **Split traffic across multiple services**。
4. **Service Weights** 滑块设置发送到各个服务的流量的百分比。



如果在超过两个服务之间进行流量分割，各个服务的相对权重通过 0 到 256 范围内的整数来指定。



在分割流量的应用程序的展开行中 **Overview** 上会显示流量加权情况。

9.4.3.1.2. 使用 CLI 管理 Weights

此命令会由路由管理服务和对应的权重负载均衡。

```
$ oc set route-backends ROUTENAME [--zero|--equal] [--adjust] SERVICE=WEIGHT[%] [...]
[options]
```

例如，以下命令将 **ab-example-A** 设置为主服务，**weight=198** 和 **ab-example-B** 作为第一个替代服务，值为 **weight=2** ：

```
$ oc set route-backends web ab-example-A=198 ab-example-B=2
```

这意味着 99% 的流量将发送到服务 **ab-example-A**，1% 发送到服务 **ab-example-B**。

此命令不扩展部署配置。您可能需要进行此操作，才能有足够的 pod 来处理请求负载。

不带标志的命令会显示当前的配置。

```
$ oc set route-backends web
NAME                KIND     TO          WEIGHT
routes/web          Service  ab-example-A 198 (99%)
routes/web          Service  ab-example-B 2   (1%)
```

--adjust 标志可让您更改单个服务相对于自身或主服务的权重。指定百分比将调整相对于主服务或第一个替代服务（如果指定了主服务）的服务。如果还有其他后端，它们的权重会与更改的比例保持比例。

```
$ oc set route-backends web --adjust ab-example-A=200 ab-example-B=10
$ oc set route-backends web --adjust ab-example-B=5%
$ oc set route-backends web --adjust ab-example-B=+15%
```

--equal 标志将所有服务的**权重**设置为 100

```
$ oc set route-backends web --equal
```

--zero 标志将所有服务的 **weight** 设为 0。所有请求都将返回 503 错误。

注意

并非所有路由器都支持多个后端或加权后端。

9.4.3.1.3. 一个 Service，多个部署配置

如果您安装了路由器，请通过路由（或直接使用服务 IP）提供应用程序：

```
$ oc expose svc/ab-example
```

通过 **ab-example.<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

1. 根据与第一个分片相同的源镜像创建第二个分片，但使用不同的标记版本，并设置唯一值：

   ```
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b --labels=ab-
   example=true SUBTITLE="shard B" COLOR="red"
   ```

2. 编辑新创建的分片，以设置对所有分片通用的 **ab-example=true** 标签：

   ```
   $ oc edit dc/ab-example-b
   ```

   在编辑器中，将 **spec.selector** 和 **spec.template.metadata.labels** 所在的 **ab-example: "true"** 行与现有的 **deploymentconfig=ab-example-b** 标签一起添加。保存并退出编辑器。

3. 触发第二个分片的重新部署以获取新标签：

   ```
   $ oc rollout latest dc/ab-example-b
   ```

4. 在这一刻，路由下同时提供了两组 pod。但是，由于两个浏览器（通过保持连接打开）和路由器（默认为 Cookie）将尝试保留与后端服务器的连接，所以您可能不会看到两个分片都返回给您。要将浏览器强制到其中一个分片，请使用 scale 命令：

   ```
   $ oc scale dc/ab-example-a --replicas=0
   ```

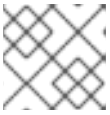
-

刷新浏览器应该会显示 **v2** 和 **shard B** （红色）。

```
$ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
```

刷新浏览器应该会显示 **v1** 和 **shard A** （蓝色）。

如果您在任一分片上触发部署，则只有该分片中的 pod 会受到影响。您可以通过 **oc edit dc/ab-example-a** 或 **oc edit dc/ab-example-b** 更改 **SUBTITLE** 环境变量来轻松地触发部署。您可以通过重复步骤 5-7 来添加额外的分片。



注意

这些步骤将在以后的 OpenShift Container Platform 版本中简化。

9.4.4. 代理分片/流量分割

在生产环境中，您可以精确控制特定分片上的流量分布。在处理大量实例时，可以使用相对比例的独立分片来实现基于百分比的流量分布。这可与**代理分片**良好结合，将接收到的流量转发或分割到在其他位置运行的单独服务或应用程序。

在最简单的配置中，代理会原封不动转发请求。在比较复杂的设置中，可以复制传入的请求，同时将它们发送到独立集群以及应用程序的本地实例，并且比较其结果。其他模式包括使 DR 安装的缓存保持活跃，或抽样传入的流量来满足分析需要。

虽然实施超出本例的范围，但任何 TCP（或 UDP）代理都可以在所需分片下运行。使用 **oc scale** 命令更改代理分片下服务请求的相对数量。对于更复杂的流量管理，请考虑使用比例平衡功能自定义 OpenShift Container Platform 路由器。

9.4.5. N-1 兼容性

同时运行新旧代码的应用程序必须要谨慎处理，以确保新代码写入的数据能被旧版代码读取和处理（或恰当忽略）。这有时被称为*架构演进*，而且是一个复杂的问题。

这可以采用磁盘上、数据库、临时缓存或者作为用户浏览器会话的一部分存储在磁盘上的多个形式。虽然大多数 Web 应用程序都支持滚动部署，但务必要测试并设计您的应用程序以便能处理它。

在一些应用程序中，同时运行新旧代码的时间是短暂的，因此程序错误或一些用户事务失败是可以接受的。至于其他应用程序，失败模式可能会导致整个应用程序无法运作。

验证 N-1 兼容性的一种方法是使用 A/B 部署。在测试环境中以受控的方式同时运行旧代码和新代码，并验证流到新部署的流量不会导致旧部署失败。

9.4.6. 正常终止

OpenShift Container Platform 和 Kubernetes 会留出时间，让应用程序实例关机后再从负载均衡轮转中移除。但是，应用程序必须保证在用户退出前彻底终止用户连接。

在关闭时，OpenShift Container Platform 会将 **TERM** 信号发送到容器中的进程。接收 **SIGTERM** 时的应用程序代码应该停止接受新的连接。这将确保负载均衡器将流量路由到其他活跃实例。然后，应用程序代码应等到所有打开的连接都关闭（或在下次机会出现时恰当终止独立的连接）后再退出。

在恰当终止期限到期后，未退出的进程将发送 **KILL** 信号，该信号会立即结束该进程。pod 或 pod 模板的 **terminationGracePeriodSeconds** 属性控制恰当终止期限（默认值 30 秒），并可根据需要自定义每个应用程序。

9.5. KUBERNETES DEPLOYMENTS 支持

9.5.1. 部署对象类型

Kubernetes 在 OpenShift Container Platform 中提供了一流的对象类型，名为 *deployments*。此对象类型（这里称为 *Kubernetes 部署*）用作部署配置对象类型的后代。

与部署配置一样，Kubernetes 部署将应用程序特定组件的所需状态描述为 pod 模板。Kubernetes 部署创建 *副本集*（复制控制器迭代），用于编配 pod 生命周期。

例如，此 Kubernetes 部署的定义会创建一个副本集来启动一个 **hello-openshift** pod:

Kubernetes 部署定义 *hello-openshift-deployment.yaml*示例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-openshift
  template:
    metadata:
      labels:
        app: hello-openshift
    spec:
      containers:
      - name: hello-openshift
        image: openshift/hello-openshift:latest
        ports:
        - containerPort: 80
```

将定义保存到本地文件后，就可以使用它来创建 Kubernetes 部署：

```
$ oc create -f hello-openshift-deployment.yaml
```

您可以使用 CLI 检查并运行 Kubernetes 部署，以及与其他对象类型一样的副本集，如 Common Operations（例如 **get** 和 **describe**）所述。对于对象类型，为 Kubernetes 部署使用 **deployments** 或 **deploy**，为副本集使用 **replicasets** 或 **rs**。

如需有关 Deployment 和 Replica Sets 的更多详细信息，请参阅 Kubernetes 文档，在 CLI 用法示例中使用 **oc** 替换 **kubectl**。

9.5.2. Kubernetes Deployments 和部署配置

由于在 Kubernetes 1.2 中添加了部署之前，OpenShift Container Platform 中存在部署配置，因此后者的对象类型与之前稍有的稍有不同。OpenShift Container Platform 中的长期目标是访问 Kubernetes 部署中的完整功能奇偶校验，并通过应用程序将其切换到使用单一的对象类型来提供精细管理。

支持 Kubernetes 部署，以确保使用新对象类型的上游项目和示例可以在 OpenShift Container Platform 上平稳运行。根据当前的 Kubernetes 部署功能集，如果不计划使用以下任一操作，则在 OpenShift Container Platform 中使用它们而不是部署配置：

- 镜像流
- 生命周期 hook
- 自定义部署策略

以下部分详细阐述两种对象类型之间的区别，以进一步协助您决定何时想通过部署配置使用 Kubernetes 部署。

9.5.2.1. 部署配置特定功能

9.5.2.1.1. 自动回滚

Kubernetes 部署不支持在出现故障时自动回滚到上次成功部署的副本集。此功能会很快被添加。

9.5.2.1.2. 触发器

Kubernetes 部署有一个隐式 **ConfigChange** 触发器，每次更改部署的 Pod 模板都会自动触发新的推出部署。如果您不想在 Pod 模板更改时进行新的推出部署，请暂停部署：

```
$ oc rollout pause deployments/<name>
```

目前，Kubernetes 部署不支持 **ImageChange** 触发器。上游建议了一个通用触发机制，但如果它被接受，则代表未知。最终，OpenShift Container Platform 的特定机制可以实施为 Kubernetes 部署之上的层，但最好作为 Kubernetes 内核的一部分存在。

9.5.2.1.3. 生命周期 Hook

Kubernetes 部署不支持任何生命周期 hook。

9.5.2.1.4. 自定义策略

Kubernetes 部署尚不支持用户指定的自定义部署策略。

9.5.2.1.5. Canary 部署

Kubernetes 部署还没有作为新推出部署的一部分运行。

9.5.2.1.6. 测试部署

Kubernetes 部署不支持运行测试跟踪。

9.5.2.2. 特定于 Kubernetes 部署的功能

9.5.2.2.1. 滚动

Kubernetes 部署的部署过程是由控制器循环推动的，这与使用部署器 Pod 进行每次新推出部署的部署配置相反。这意味着，Kubernetes 部署可以拥有尽可能多的活跃副本集，最终部署控制器将缩减所有旧副本集，并扩展最新的副本集。

部署配置最多可以有一个部署器 pod 运行，否则多个部署器最终会相互努力以扩展它们认为是最新的复制控制器。因此，任何时间点上只能有两个复制控制器处于活跃状态。最终，这会转化为 Kubernetes 部署的快速推出部署。

9.5.2.2.2. 按比例扩展

因为 Kubernetes 部署控制器是部署所拥有的新旧副本集的大小的唯一来源，所以它可以扩展正在进行的推出部署。额外副本会根据每个副本集的大小按比例分发。

当一个推出部署(rollout)正在进行时无法扩展部署配置，因为部署配置控制器最终与部署器进程有关新复制控制器的大小。

9.5.2.2.3. 暂停 Mid-rollout

Kubernetes 部署可以在任何时间点上暂停，这意味着您可以暂停正在进行的推出部署。另一方面，当前无法暂停部署器 Pod。因此，如果您尝试在推出部署进行期间暂停部署配置，则部署器进程不受影响，它会继续运行直到完成为止。

第 10 章 模板

10.1. 概述

模板描述了一组可参数化和处理的对象，用于生成对象列表，供 OpenShift Container Platform 用于创建。可对模板进行处理，以创建您有权在项目中创建的任何内容，如 服务、构建配置和部署配置。模板还可定义一系列标签（label），以应用到该模板中定义的每个对象。

您可以使用 CLI 从模板创建对象列表，或者如果模板已上传至项目或全局模板库，则可使用 web 控制台来创建。有关一组策展的模板，请参阅 OpenShift 镜像流和模板库。

10.2. 上传模板

如果您有定义模板的 JSON 或 YAML 文件，如 本例中所示，您可以使用 CLI 将模板上传到项目。此操作将模板保存到项目，供任何有适当权限访问该项目的用户重复使用。本主题后面会介绍如何编写自己的模板。

要将模板上传到当前项目的模板库，请使用以下命令传递 JSON 或 YAML 文件：

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

您可以使用 **-n** 选项以及项目名称将模板上传到其他项目：

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

现在可使用 web 控制台或 CLI 选择该模板。

10.3. 使用 WEB 控制台从模板创建

请参阅使用 Web 控制台创建应用程序。

10.4. 使用 CLI 从模板创建

您可以使用 CLI 来处理模板，并使用所生成的配置来创建对象。

10.4.1. 标签

标签 (label)用于管理和组织生成的对象，如 pod。模板中指定的标签应用于从模板生成的每个对象。

也可以从命令行在模板中添加标签。

```
$ oc process -f <filename> -l name=otherLabel
```

10.4.2. 参数

模板的 **parameter** 部分列出了可覆盖的参数列表。您可以使用以下命令并指定要使用的文件来通过 CLI 列出它们：

```
$ oc process --parameters -f <filename>
```

或者，如果模板已上传：

```
$ oc process --parameters -n <project> <template_name>
```

例如，下面显示了在默认 **openshift** 项目中列出其中一个 Quickstart 模板的参数时的输出：

```
$ oc process --parameters -n openshift rails-postgresql-example
NAME                     DESCRIPTION
GENERATOR          VALUE
SOURCE_REPOSITORY_URL        The URL of the repository with your application source code
https://github.com/sclorg/rails-ex.git
SOURCE_REPOSITORY_REF        Set this to a branch name, tag or other ref of your repository if
you are not using the default branch
CONTEXT_DIR                  Set this to the relative path to your project if it is not in the root of your
repository
APPLICATION_DOMAIN           The exposed hostname that will route to the Rails service
rails-postgresql-example.openshiftapps.com
GITHUB_WEBHOOK_SECRET        A secret string used to configure the GitHub webhook
expression         [a-zA-Z0-9]{40}
SECRET_KEY_BASE              Your secret key for verifying the integrity of signed cookies
expression         [a-z0-9]{127}
APPLICATION_USER             The application user that is used within the sample application to
authorize access on pages                        openshift
APPLICATION_PASSWORD         The application password that is used within the sample
application to authorize access on pages                         secret
DATABASE_SERVICE_NAME        Database service name
postgresql
POSTGRESQL_USER              database username
expression         user[A-Z0-9]{3}
POSTGRESQL_PASSWORD          database password
expression         [a-zA-Z0-9]{8}
POSTGRESQL_DATABASE          database name
root
POSTGRESQL_MAX_CONNECTIONS   database max connections
10
POSTGRESQL_SHARED_BUFFERS    database shared buffers
12MB
```

该输出标识了在处理模板时使用类似正则表达式的生成器生成的几个参数。

10.4.3. 生成对象列表

使用 CLI，您可以处理定义模板的文件，以将对象列表返回到标准输出：

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

或者，如果模板已上传到当前项目：

```
$ oc process <template_name>
```

您可以通过处理模板并将输出传送至 **oc create** 来从模板创建对象：

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

或者，如果模板已上传到当前项目：

```
-
```

```
$ oc process <template> | oc create -f -
```

您可以为每个要覆盖的 **<name>=<value>** 对添加 **-p** 选项，以覆盖文件中定义的任何参数值。参数引用可能会出现在模板项目内的任何文本字段中。

例如，在以下部分中，模板的 **POSTGRESQL_USER** 和 **POSTGRESQL_DATABASE** 参数被覆盖，以输出带有自定义环境变量的配置：

例 10.1. 从模板创建对象列表

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase
```

JSON 文件可重定向到文件，也可直接应用，而无需通过将已处理的输出传送到 **oc create** 命令来上传模板：

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase \
    | oc create -f -
```

如有大量参数，可将其保存到文件中，然后将此文件传递到 **oc process**：

```
$ cat postgres.env
POSTGRESQL_USER=bob
POSTGRESQL_DATABASE=mydatabase
$ oc process -f my-rails-postgresql --param-file=postgres.env
```

此外，您还可使用 **"-"** 作为 **--param-file** 的参数，从标准输入中读取环境：

```
$ sed s/bob/alice/ postgres.env | oc process -f my-rails-postgresql --param-file=-
```

10.5. 修改上传的模板

您可以使用以下命令编辑已上传至项目中的模板：

```
$ oc edit template <template>
```

10.6. 使用 INSTANT APP 和 QUICKSTART TEMPLATES

OpenShift Container Platform 提供很多默认的 Instant App 和 Quickstart 模板，有助于快速开始为不同语言创建新应用程序。提供了适用于 Rails (Ruby)、Django (Python)、Node.js、CakePHP (PHP) 和 Dancer (Perl) 的模板。您的集群管理员应已在默认的全局 **openshift** 项目中创建了这些模板，以便您访问。您可通过以下命令列出可用的默认 Instant App 和 Quickstart 模板：

```
$ oc get templates -n openshift
```

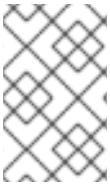
如果没有这些内容，请推荐集群管理员参阅 Loading the Default Image Streams and Templates 中的内容。

默认情况下，模板会使用 GitHub 上包含所需应用程序代码的公共源存储库进行构建。要修改源并构建您自己的应用程序版本，您必须：

1. 对模板默认的 **SOURCE_REPOSITORY_URL** 参数引用的存储库进行分叉。
2. 在从模板创建时，覆盖 **SOURCE_REPOSITORY_URL** 参数的值，从而指定您的分叉而非默认值。

这样，模板创建的构建配置将指向应用程序代码的分叉，您可随意修改代码和重新构建应用程序。

使用 Web 控制台进行这个过程的内容包括在开发人员入门：Web 控制台。



注意

某些 Instant App 和 Quickstart 模板会定义一个 数据库部署配置。它们定义的配置对数据库内容使用临时存储。这些模板仅限于演示目的，因为如果数据库 pod 因任何原因重启，所有数据库数据都将丢失。

10.7. 编写模板

您可以定义新模板，以便轻松重新创建应用程序的所有对象。该模板将定义由其创建的对象以及一些元数据，以指导创建这些对象。

例 10.2. 简单模板对象定义(YAML)

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects:
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD}
      image: dockerfile/redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
parameters:
- description: Password used for Redis authentication
  from: '[A-Z0-9]{8}'
  generate: expression
  name: REDIS_PASSWORD
labels:
  redis: master
```

10.7.1. 描述

模板描述向用户介绍模板的作用，有助于用户在 web 控制台中搜索查找模板。除模板名称以外的其他元数据均为可选，但若有则会非常有用。除常规描述性信息外，元数据还应包含一组标签。实用标签包括与模板相关的语言名称（如 **java**、**php**、**ruby** 等）。

例 10.3. 模板描述元数据

```
kind: Template
apiVersion: v1
metadata:
  name: cakephp-mysql-example (1)
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" (2)
    description: >-
      An example CakePHP application with a MySQL database. For more information
      about using this template, including OpenShift considerations, see
      https://github.com/sclorg/cakephp-ex/blob/master/README.md.


      WARNING: Any data stored will be lost upon pod destruction. Only use this
      template for testing." (3)
    openshift.io/long-description: >-
      This template defines resources needed to develop a CakePHP application,
      including a build configuration, application deployment configuration, and
      database deployment configuration.  The database is stored in
      non-persistent storage, so this configuration should be used for
      experimental purposes only. (4)
    tags: "quickstart,php,cakephp" (5)
    iconClass: icon-php (6)
    openshift.io/provider-display-name: "Red Hat, Inc." (7)
    openshift.io/documentation-url: "https://github.com/sclorg/cakephp-ex" (8)
    openshift.io/support-url: "https://access.redhat.com" (9)
message: "Your admin credentials are ${ADMIN_USERNAME}:${ADMIN_PASSWORD}" (10)
```

(1) 模板的唯一名称。

(2) 可由用户界面使用的简单、用户友好型名称。

(3) 模板的描述。包含充足的详细信息，供用户了解所部署的内容以及部署前须知的注意事项。此外，还应提供其他信息链接，如 ***README*** 文件。可包括换行符来创建段落。

(4) 其他模板描述。例如，这可按照服务目录显示。

(5) 要与模板关联的标签，用于搜索和分组。添加将包含在其中一个提供的目录类别中的标签。请参见控制台常量文件的 **CATALOG_CATEGORIES** 中的 **id** 和 **categoryAliases**。此外，还可为整个集群自定义类别。

(6) 在 web 控制台中与模板一同显示的图标。尽可能从现有 徽标图标 中进行选择。也可使用 FontAwesome 和 PatternFly 中的图标。另外，也可通过 CSS 自定义 提供图标，它们可添加到使用模板的 OpenShift Container Platform 集群中。您必须指定一个存在的图标类，否则它将阻止回退到通用图标。

7 提供模板的个人或组织的名称。

8 用于参考更多模板文档的 URL。

9 用于获取模板支持的 URL。

10 模板实例化时显示的说明消息。该字段应向用户介绍如何使用新建资源。显示消息前，对消息进行参数替换，以便输出中包含所生成的凭据和其他参数。其中包括用户应遵守的所有后续步骤文档链接。

10.7.2. 标签

模板可包含一组标签。这些标签将添加至模板实例化时创建的各个对象中。采用这种方式定义标签可方便用户查找和管理从特定模板创建的所有对象。

例 10.4. 模板对象标签

```
kind: "Template"
apiVersion: "v1"
...
labels:
  template: "cakephp-mysql-example" 1
  app: "${NAME}" 2
```

1 标签，将应用于从该模板创建的所有对象。

2 参数化标签，也将用于从该模板创建的所有对象。对标签键和值均执行参数扩展。

10.7.3. 参数

允许用户提供一个值或在实例化模板时生成一个值作为参数。然后，该值将在引用参数的任意位置上被替换。可在对象列表字段中的任意字段中定义引用。这有助于生成随机密码，或允许用户提供自定义模板时所需的主机名或其他用户特定值。可通过以下两种方式引用参数：

- 作为字符串值，将格式为 **${PARAMETER_NAME}** 的值放在模板的任意字符串字段中。
- 作为 json/yaml 值，将格式为 **${PARAMETER_NAME}** 的值放在模板的任意字段中。

使用 **${PARAMETER_NAME}** 语法时，可将多个参数引用合并到一个字段中，并可将引用嵌入到固定数据中，如 **"http://${PARAMETER_1}${PARAMETER_2}"**。两个参数值均将被替换，结果值将是一个带引号的字符串。

使用 **${{PARAMETER_NAME}}** 语法时，仅允许单个参数引用，不允许使用前导/尾随字符。执行替换后，结果值将不加引号，除非结果不是有效的 json 对象。如果结果不是有效的 json 值，则结果值将加引号并被视为标准字符串。

单个参数可在模板中多次引用，且可在单个模板中使用两种替换语法来引用。

可提供默认值，如果用户未提供其他值则使用默认值：

例 10.5. 将一个明确的值设置为默认值

```
parameters:
  - name: USERNAME
    description: "The user name for Joe"
    value: joe
```

也可通过参数定义中指定的规则生成参数值：

例 10.6. 生成参数值

```
parameters:
  - name: PASSWORD
    description: "The random user password"
    generate: expression
    from: "[a-zA-Z0-9]{12}"
```

在上例中，处理将生成一个由大小写字母和数字组成的 12 个字符长的随机密码。

可用语法并非完整的正则表达式语法。但是，您可以使用 **\w**、**\d**和 **\a** 修饰符：

- **[\w]{10}** 生成 10 个字母字符、数字和下划线。它遵循 PCRE 标准，等同于 **[a-zA-Z0-9_]{10}**。
- **[\d]{10}** 生成 10 个数字。等同于 **[0-9]{10}**。
- **[\a]{10}** 生成 10 个字母字符。这等同于 **[a-zA-Z]{10}**。

下面是附带参数定义和参考的完整模板示例：

例 10.7. 带有参数定义和参考的完整模板

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: cakephp-mysql-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}" 1
          ref: "${SOURCE_REPOSITORY_REF}"
        contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
```

```
      replicas: "${{REPLICA_COUNT}}" 2
parameters:
  - name: SOURCE_REPOSITORY_URL 3
    displayName: Source Repository URL 4
    description: The URL of the repository with your application source code 5
    value: https://github.com/sclorg/cakephp-ex.git 6
    required: true 7
  - name: GITHUB_WEBHOOK_SECRET
    description: A secret string used to configure the GitHub webhook
    generate: expression 8
    from: "[a-zA-Z0-9]{40}" 9
  - name: REPLICA_COUNT
    description: Number of replicas to run
    value: "2"
    required: true
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET} ..." 10
```

1. 模板实例化时，该值将被替换为 **SOURCE_REPOSITORY_URL** 参数的值。
2. 模板实例化时，该值将被替换为 **REPLICA_COUNT** 参数的不加引号值。
3. 参数的名称。该值用于引用模板中的参数。
4. 参数的用户友好型名称。这将对用户显示。
5. 参数的描述。出于参数目的提供更详细的信息，包括对预期值的任何限制。描述应当按照控制台的文本标准使用完整句子。不可与显示名称重复。
6. 如果用户实例化该模板时未覆盖该值，则将使用该参数的默认值。密码之类避免使用默认值，而应结合使用生成的参数与 Secret。
7. 指示此参数为必填项，表示用户无法用空白值覆盖它。如果该参数未提供默认值或生成值，则用户必须提供一个值。
8. 生成其值的参数。
9. 生成器的输入。这种情况下，生成器会生成一个 40 个字符的字母数字值，其中包括大写和小写字符。
10. 参数可包含在模板消息中。此项告知用户生成的值。

10.7.4. 对象列表

模板主要部分为对象列表，将在模板实例化时创建。这可以是任何有效的 API 对象，如 **BuildConfig**、**DeploymentConfig**、**Service** 等。该对象将完全按照此处定义创建，创建前替换任意参数值。这些对象的定义可引用前面定义的参数。

```
kind: "Template"
apiVersion: "v1"
metadata:
  name: my-template
objects:
```

```
- kind: "Service" (1)
  apiVersion: "v1"
  metadata:
    name: "cakephp-mysql-example"
    annotations:
      description: "Exposes and load balances the application pods"
  spec:
    ports:
      - name: "web"
        port: 8080
        targetPort: 8080
    selector:
      name: "cakephp-mysql-example"
```

(1) 将由该模板创建的 **Service** 的定义。

注意

如果对象定义的元数据包含固定的 **namespace** 字段值，则在模板实例化过程中，该字段将从定义中剥离出来。如果 **namespace** 字段包含参数引用，则将执行正常的参数替换，并参数替换将值解析到的任何命名空间中创建对象，假定用户有权在该命名空间中创建对象。

10.7.5. 将模板标记为可绑定

模板服务代理在目录中为其了解的每个模板对象公告一个服务。默认情况下，每个服务均会公告为"可绑定"，表示允许最终用户绑定制备的服务。

模板创建者可将注解 **template.openshift.io/bindable: "false"** 添加到模板中，防止最终用户绑定从给定模板制备的服务。

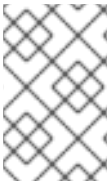
10.7.6. 公开对象字段

模板创建者可指定模板中的特定对象字段应公开。模板服务代理识别 ConfigMap、Secret、Service 和 Route 上的公开字段，并在用户绑定代理支持的服务时返回公开字段的值。

要公开对象的一个或多个字段，请在模板中为对象添加以 **template.openshift.io/expose-** 或 **template.openshift.io/base64-expose-** 为前缀的注解。

每个移除前缀的注解键均会被传递成为 **bind** 响应中的一个键。

每个注解值是一个 Kubernetes JSONPath 表达式，它会在绑定时解析，以指示应在 **bind** 响应中返回值的对象字段。



注意

Bind 响应键/值对可在系统其他部分用作环境变量。因此，建议删除前缀的每个注解键均应为有效的环境变量名称，以字符 **A-Z**、**a-z** 或下划线开头，后跟 0 个或多个 **A-Z**、**a-z**、**0-9** 或下划线字符。

使用 **template.openshift.io/expose-** 注解来以字符串形式返回字段值。这样很方便，尽管没有处理任意二进制数据。如果要返回二进制数据，请在返回前使用 **template.openshift.io/base64-expose-** 注解对数据进行 base64 编码。

注意

除非用反斜杠转义，否则 Kubernetes 的 JSONPath 实现会将 **.**、**@** 字符等解析为元字符，而无关其在表达式中的位置。因此，例如要引用名为 **my.key** 的 **ConfigMap** 数据，所需 JSONPath 表达式应为 **{.data['my\.key']}**。根据 JSONPath 表达式在 YAML 中的编写方式，可能需要额外增加反斜杠，如 **"{.data['my\\.key']}"**。

以下是被公开的不同对象字段的示例：

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: ConfigMap
  apiVersion: v1
  metadata:
    name: my-template-config
    annotations:
      template.openshift.io/expose-username: "{.data['my\\.username']}"
  data:
    my.username: foo
- kind: Secret
  apiVersion: v1
  metadata:
    name: my-template-config-secret
    annotations:
      template.openshift.io/base64-expose-password: "{.data['password']}"
  stringData:
    password: bar
- kind: Service
  apiVersion: v1
  metadata:
    name: my-template-service
    annotations:
      template.openshift.io/expose-service_ip_port: "{.spec.clusterIP}:{.spec.ports[?
(.name==\"web\")].port}"
  spec:
    ports:
    - name: "web"
      port: 8080
- kind: Route
  apiVersion: v1
  metadata:
    name: my-template-route
    annotations:
      template.openshift.io/expose-uri: "http://{.spec.host}{.spec.path}"
  spec:
    path: mypath
```

下面是在遵守上述部分模板情况下，对 **bind** 操作的一个响应示例：

```
{
  "credentials": {
    "username": "foo",
```

```
    "password": "YmFy",
    "service_ip_port": "172.30.12.34:8080",
    "uri": "http://route-test.router.default.svc.cluster.local/mypath"
  }
}
```

10.7.7. 等待模板就绪

模板创建者可指定：在服务目录、Template Service Broker 或 TemplateInstance API 进行的模板实例化被视为完成之前，应等待模板中的某些对象。

要使用该功能，请使用以下注解在模板中标记一个或多个**Build**、**BuildConfig**、**Deployment**、**DeploymentConfig**、**Job** 或 **StatefulSet** 类型的对象：

```
"template.alpha.openshift.io/wait-for-ready": "true"
```

直到标有注解的所有对象报告就绪时，模板实例化才算完成。同样，如果任何注解的对象报告失败，或者模板未能在一小时的固定超时内就绪，则模板实例化将失败。

就实例化而言，各种对象类型的就绪和失败定义如下：

类型	就绪	失败
Build	对象报告阶段完成	对象报告阶段取消、错误或失败
BuildConfig	最新关联构建对象报告阶段完成	最新关联构建对象报告阶段取消、错误或失败
Deployment	对象报告新的 ReplicaSet 和部署可用（这遵循对象上定义的就绪探针）	对象报告进度状况为错误
DeploymentConfig	对象报告新的 ReplicationController 和部署可用（这遵循对象上定义的就绪探针）	对象报告进度状况为错误
Job	对象报告完成	对象报告出现一个或多个故障
StatefulSet	对象报告所有副本就绪（这遵循对象上定义的就绪探针）	不适用

以下是使用 **wait-for-ready** 注解的模板提取示例。更多示例可在 OpenShift quickstart 模板中找到。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
```

```
      # wait-for-ready used on BuildConfig ensures that template instantiation
      # will fail immediately if build fails
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: DeploymentConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: Service
  apiVersion: v1
  metadata:
    name: ...
  spec:
    ...
```

10.7.8. 其他建议

- 设置内存、CPU 和存储的默认大小，以确保您的应用程序获得足够资源使其平稳运行。
- 如果要在主版本中使用该标签，请避免引用来自镜的 **latest** 标签。当新镜像被推送（push）到该标签时，这可能会导致运行中的应用程序中断。
- 良好的模板可整洁地构建和部署，无需在部署模板后进行修改。

10.7.9. 从现有对象创建模板

您可以 YAML 格式从项目中导出现有对象，然后通过添加参数和其他自定义作为模板表单来修改 YAML，而无需从头开始编写整个模板。要以 YAML 格式导出项目中的对象，请运行：

```
$ oc get -o yaml --export all > <yaml_filename>
```

您还可替换特定资源类型或多个资源，而非 **all** 资源。运行 **oc get -h** 获取更多示例。

oc get --export all 中包括的对象类型是：

- BuildConfig
- Build
- DeploymentConfig
- ImageStream
- Pod
- ReplicationController
- Route
- 服务

第 11 章 打开远程 SHELL 至容器

11.1. 概述

oc rsh 命令允许您在本地访问和管理系统中的工具。安全 shell(SSH)是底层技术和行业标准，可提供到应用程序的安全连接。使用 shell 环境访问应用程序通过安全增强型 Linux(SELinux)策略受到保护限制。

11.2. 启动 SECURE SHELL 会话

打开到容器的远程 shell 会话：

```
$ oc rsh <pod>
```

在远程 shell 中，您可以像在容器内部一样发出命令进行内部操作，如监控、故障调试等，并执行特定于容器中运行的项的 CLI 命令。

例如，在 MySQL 容器中，您可以通过调用 **mysql** 命令 来计算数据库中的记录数量，然后使用 提示符在 **SELECT** 命令中键入。您还可以使用 **ps(1)** 和 **ls(1)** 等命令进行验证。

BuildConfig 和 **DeployConfigs** 用于定义您需要的项，pod（内部使用容器）会根据需要被创建或销毁。您的更改不持久。如果您直接在容器内进行更改，且该容器被销毁并重新构建，则您的更改将不再存在。

注意

oc exec 可用于远程执行命令。但是，**oc rsh** 命令提供了一种更容易地打开远程 shell 的方法。

11.3. SECURE SHELL 会话帮助

有关使用、选项以及查看示例的帮助信息：

```
$ oc rsh -h
```

第 12 章 服务帐户

12.1. 概述

当用户使用 OpenShift Container Platform CLI 或 Web 控制台时，其 API 令牌向 OpenShift API 进行身份验证。但是，当普通用户的凭据不可用时，组件通常会单独进行 API 调用。例如：

- 复制控制器进行 API 调用来创建或删除容器集。
- 容器内的应用程序可能会发出 API 调用来进行发现。
- 外部应用程序可以发出 API 调用来进行监控或集成。

服务帐户为控制 API 访问提供了灵活的方式，不需要共享常规用户的凭证。

12.2. 用户名和组

每个服务帐户都有一个关联的用户名，可以像普通用户一样被授予角色。用户名派生自其项目和名称：

```
system:serviceaccount:<project>:<name>
```

例如，将 **view** 角色添加到 **top-secret** 项目中的 **robot** 服务帐户：

```
$ oc policy add-role-to-user view system:serviceaccount:top-secret:robot
```

重要

如果要向项目中的特定服务帐户授予访问权限，您可以使用 **-z** 标志。在服务帐户所属的项目中，使用 **-z** 标志并指定 **<serviceaccount_name>**。强烈建议您这样做，因为它有助于防止拼写错误，并确保只为指定的服务帐户授予访问权限。例如：

```
$ oc policy add-role-to-user <role_name> -z <serviceaccount_name>
```

如果没有在项目中，请使用 **-n** 选项来指示它应用到的项目命名空间，如下例所示。

每一服务帐户也是以下两个组的成员：

system:serviceaccounts
包含系统中的所有服务帐户。

system:serviceaccounts:<project>
包含指定项目中的所有服务帐户。

例如，允许所有项目中的所有服务帐户查看 **top-secret** 项目中的资源：

```
$ oc policy add-role-to-group view system:serviceaccounts -n top-secret
```

允许 **managers** 项目中的所有服务帐户编辑 **top-secret** 项目中的资源：

```
$ oc policy add-role-to-group edit system:serviceaccounts:managers -n top-secret
```

12.3. 默认服务帐户和角色

每个项目中会自动创建三个服务帐户：

服务帐户	使用方法
builder	由构建 Pod 使用。被授予 **system:image-builder** 角色，允许使用内部 Docker registry 将镜像推送到项目中的任何镜像流。
deployer	部署 Pod 使用并被授予 **system:deployer** 角色，它允许查看和修改项目中的复制控制器和 pod。
default	用来运行其他所有 Pod，除非指定了不同的服务帐户。

项目中的所有服务帐户都会被授予 **system:image-puller** 角色，允许使用内部容器镜像 registry 从项目中的任何镜像流拉取镜像。

12.4. 管理服务帐户

服务帐户是各个项目中存在的 API 对象。要管理服务帐户，您可以使用 **oc** 命令和 **sa** 或 **serviceaccount** 对象类型，或者使用 Web 控制台。

获取当前项目中现有服务帐户的列表：

```
$ oc get sa
NAME      SECRETS   AGE
builder   2         2d
default   2         2d
deployer  2         2d
```

要创建新服务帐户，请执行以下操作：

```
$ oc create sa robot
serviceaccount "robot" created
```

一旦创建了服务帐户，会自动向其中添加两个 secret：

- API 令牌
- OpenShift Container Registry 的凭证

可以通过描述服务帐户来查看它们：

```
$ oc describe sa robot
Name:  robot
Namespace: project1
Labels:  <none>
Annotations: <none>

Image pull secrets: robot-dockercfg-qzbhb

Mountable secrets:  robot-token-f4khf
```

```
                robot-dockercfg-qzbhb

Tokens:         robot-token-f4khf
                robot-token-z8h44
```

系统确保服务帐户始终具有 API 令牌和 registry 凭据。

生成的 API 令牌和 registry 凭据不会过期，但可通过删除 secret 来撤销。删除 secret 时，会自动生成一个新 secret 来代替它。

12.5. 管理允许的机密

除了提供 API 凭证外，pod 的服务帐户也决定允许 pod 使用哪些 secret。

Pod 以两种方式使用 secret：

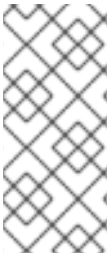
- 镜像拉取 secret，提供用于为 pod 容器拉取镜像的凭证
- 可挂载 secret，将 secret 的内容作为环境变量注入容器中

要允许由服务帐户的 pod 使用 secret 作为镜像 pull secret，请运行：

```
$ oc secrets link --for=pull <serviceaccount-name> <secret-name>
```

要允许由服务帐户的 pod 挂载 secret，请运行：

```
$ oc secrets link --for=mount <serviceaccount-name> <secret-name>
```



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 master 配置文件中将 **serviceAccountConfig.limitSecretReferences** 设置为 **false**（默认设置），则不需要将 secret 挂载到服务帐户的 pod 中，且不需要使用 **--for=mount** 选项。但是，无论 **serviceAccountConfig.limitSecretReferences** 值是什么，需要使用 **--for=pull** 选项启用镜像 pull secret。

这个示例创建并将 secret 添加到服务帐户中：

```
$ oc create secret generic secret-plans \
    --from-file=plan1.txt \
    --from-file=plan2.txt
secret/secret-plans

$ oc create secret docker-registry my-pull-secret \
    --docker-username=mastermind \
    --docker-password=12345 \
    --docker-email=mastermind@example.com
secret/my-pull-secret

$ oc secrets link robot secret-plans --for=mount

$ oc secrets link robot my-pull-secret --for=pull

$ oc describe serviceaccount robot
```

```
Name:               robot
Labels:             <none>
Image pull secrets: robot-dockercfg-624cx
                    my-pull-secret

Mountable secrets:  robot-token-uzkbh
                    robot-dockercfg-624cx
                    secret-plans

Tokens:             robot-token-8bhpp
                    robot-token-uzkbh
```

12.6. 在容器中使用服务帐户凭证

创建 pod 时，它指定一个服务帐户（或使用默认服务帐户），并使用该服务帐户的 API 凭证并引用 secret。

包含 pod 服务帐户的 API 令牌的文件会自动挂载到 */var/run/secrets/kubernetes.io/serviceaccount/token*。

该令牌可用于发出 API 调用作为 pod 的服务帐户。这个示例调用 ***users/~*** API 来获取有关令牌标识的用户的信息：

```
$ TOKEN="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"

$ curl --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    "https://openshift.default.svc.cluster.local/oapi/v1/users/~" \
    -H "Authorization: Bearer $TOKEN"

kind: "User"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "system:serviceaccount:top-secret:robot"
  selflink: "/oapi/v1/users/system:serviceaccount:top-secret:robot"
  creationTimestamp: null
identities: null
groups:
  - "system:serviceaccount"
  - "system:serviceaccount:top-secret"
```

12.7. 从外部使用服务帐户的凭证

同一令牌可以分发到需要向 API 进行身份验证的外部应用程序。

使用以下语法查看服务帐户的 API 令牌：

```
$ oc describe secret <secret-name>
```

例如：

```
$ oc describe secret robot-token-uzkbh -n top-secret
Name:  robot-token-uzkbh
Labels:  <none>
Annotations: kubernetes.io/service-account.name=robot,kubernetes.io/service-
```

```
account.uid=49f19e2e-16c6-11e5-afdc-3c970e4b7ffe

Type: kubernetes.io/service-account-token

Data

token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...

$ oc login --token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
Logged into "https://server:8443" as "system:serviceaccount:top-secret:robot" using the token
provided.

You don't have any projects. You can try to create a new project, by running

    $ oc new-project <projectname>

$ oc whoami
system:serviceaccount:top-secret:robot
```

第 13 章 管理镜像

13.1. 概述

镜像流 包含由标签识别的任意数量的容器镜像。提供相关镜像的单一虚拟视图，类似于容器镜存储库。

通过监控镜像流，构建和部署可在添加或修改新镜像时收到通知，并通过分别执行构建或部署来作出反应。

您可以通过多种方式与镜像交互并设置镜像流，具体取决于镜像 registry 所处的位置、这些 registry 的任何身份验证要求以及您预期的构建和部署性能。以下部分涵盖了这些主题的范围。

13.2. 标记镜像

在使用 OpenShift Container Platform 镜像流及其标签前，它有助于首先了解容器镜像的上下文中的镜像标签。

容器镜像可向它们添加名称，以便更直观地确定它们所包含的内容，称为 *标签*。使用标签指定镜像中包含的内容版本是常见用例。如果您有一个名为 **ruby** 的镜像，则可以将名为 **2.0** 的 2.0 标签用于 Ruby，另一个名为 **latest** 来指示该仓库中最新构建的镜像。

使用 **docker** CLI 直接与镜像交互时，**docker tag** 命令可以添加标签，后者本质上为由多个部分组成的镜像添加一个别名。这些部分包括：

```
<registry_server>/<user_name>/<image_name>:<tag>
```

如果镜像存储在带有内部 registry(OpenShift Container Registry)的 OpenShift Container Platform 环境中，则上面的 **<user_name>** 部分也可以引用一个项目或命名空间。

OpenShift Container Platform 提供 **oc tag** 命令，该命令类似于 **docker tag** 命令，但是在镜像流上运行，而非直接在镜像上运行。

注意

如需有关直接使用 **docker** CLI 标记镜像的更多信息，请参阅 Red Hat Enterprise Linux 7 的 Getting Started with Containers 文档。

13.2.1. 将标签添加到镜像流

请记住，OpenShift Container Platform 中的镜像流包含 0 个或更多由标签标识的容器镜像，您可以使用 **oc tag** 命令向镜像流中添加标签：

```
$ oc tag <source> <destination>
```

例如，要将 **ruby** 镜像流 **static-2.0** 标签配置为始终引用 **ruby** 镜像流 2.0 标签的当前镜像：

```
$ oc tag ruby:2.0 ruby:static-2.0
```

这会在 **ruby** 镜像流中创建名为 **static-2.0** 的新镜像流标签。运行 **oc tag** 时，新标签会直接引用 **ruby:2.0** 镜像流标签所指向的镜像 id，而所指向的镜像不会改变。

有各种不同类型的标签可用。默认行为是使用一个 *持久* 标签，指向一个特定的镜像；即使源有变化，新的（目标）标签不会改变。

*跟踪*标签表示，在导入源标签期间对目的地标签的元数据进行了更新。为确保目标标签在源标签更改时进行更新，请使用 **--alias=true** 标志：

```
$ oc tag --alias=true <source> <destination>
```

注意

使用*跟踪*标签创建持久性别名（例如： **latest** 或 **stable**）。该标签*只*在单一镜像流中正常工作。试图创建跨镜像流别名会出错。

您还可以添加 **--scheduled=true** 标志来定期刷新目的地标签（例如，重新导入）。周期在系统级别 进行全局配置。如需了解更多详细信息，请参阅导入标签和镜像元数据。

--reference 标志会创建一个非导入的镜像流标签。该标签持久指向源位置。

如果要指示 Docker 始终从集成的 registry 中获取标记的镜像，请使用 **--reference-policy=local**。registry 使用 pull-through 功能 为客户端提供镜像。默认情况下，镜像 Blob 由 registry 在本地进行镜像。因此，下次需要时便可更快拉取（pull）。只要镜像流有一个 insecure annotation，或标签有一个 insecure import policy，该标志也允许从不安全的 registry 拉取(pull)，无需向 Docker 守护进程提供 **--insecure-registry**。

13.2.2. 建议的标记惯例

镜像随时间不断发展，其标签反应了这一点。镜像标签始终指向最新镜像构建。

如果标签名称中嵌入了太多信息（如 **v2.0.1-may-2016**），标签仅指向镜像的某一个版本，永远不会更新。使用默认镜像修剪选项，此类镜像不会被删除。在庞大集群中，为每个修改后的镜像创建新标签这种模式最终可能会使用早已过期的镜像的多余标签元数据来填充 etcd 数据存储。

如果标签命名为 **v2.0**，则更多镜像修订的可能性更大。这会导致 标签历史记录 较长，镜像修剪器更有可能删除旧和未使用的镜像。如需更多信息，请参阅修剪镜像。

您可自行决定标签命名惯例，下面提供了一些 **<image_name>:<image_tag>** 格式的示例：

表 13.1. 镜像标签命名约定

描述	示例
修订	**myimage:v2.0.1**
架构	**myimage:v2.0-x86_64**
基础镜像	**myimage:v1.2-centos7**
最新（可能不稳定）	**myimage:latest**
最新稳定	**myimage:stable**

如果标签名称中需要日期，请定期检查旧的和不受支持的镜像以及 **istags**，并予以删除。否则，您可能会遇到旧镜像导致的资源使用量增加。

13.2.3. 从镜像流中删除标签

要从镜像流运行中完全删除标签：

```
$ oc delete istag/ruby:latest
```

或：

```
$ oc tag -d ruby:latest
```

13.2.4. 引用镜像流中的镜像

可以使用以下引用类型在镜像流中引用镜像：

- **ImageStreamTag** 用于引用或检索给定镜像流和标签的镜像。它的名称使用以下惯例：

  ```
  <image_stream_name>:<tag>
  ```

- **ImageStreamImage** 用于引用或检索给定镜像流和镜像名称的镜像。它的名称使用以下惯例：

  ```
  <image_stream_name>@<id>
  ```

  <id> 是针对特定镜像的不可变标识符，也称摘要。

- **DockerImage** 用于引用或检索给定外部 registry 的镜像。它使用标准 Docker *拉取规格* 作为名称，例如：

  ```
  openshift/ruby-20-centos7:2.0
  ```

  注意

  如果未指定标签，则会假定使用 **latest**标签。

  此外，您还可引用第三方 registry：

  ```
  registry.redhat.io/rhel7:latest
  ```

  或者带有摘要的镜像：

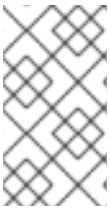
  ```
  centos/ruby-22-
  centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b2
  8e
  ```

当查看镜像流定义示例时，如 CentOS 镜像流示例，您可能会注意到它们包含 **ImageStreamTag** 的定义以及 **DockerImage** 的引用，但不与 **ImageStreamImage** 相关。

这是因为当您将镜像导入或标记到镜像流时，OpenShift Container Platform 中会自动创建 **ImageStreamImage** 对象。您不必在用于创建镜像流的任何镜像流定义中显式定义 **ImageStreamImage** 对象。

您可以使用镜像流名称和 ID 检索 **ImageStreamImage** 定义，来查看镜像的对象定义：

```
$ oc get -o yaml --export isimage <image_stream_name>@<id>
```



注意

您可以运行以下命令来找到给定镜像流的有效 **<id>** 值：

```
$ oc describe is <image_stream_name>
```

例如，在 **ruby** 镜像流中，要求 **ImageStreamImage** 的名称和 ID 为 **ruby@3a335d7** ：

通过 ImageStreamImage检索镜像对象的定义

```
$ oc get -o yaml --export isimage ruby@3a335d7

apiVersion: v1
image:
  dockerImageLayers:
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
  - name: sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c29
    size: 196634330
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
  - name: sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea092
    size: 177723024
  - name: sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b6e
    size: 55679776
  dockerImageMetadata:
    Architecture: amd64
    Author: SoftwareCollections.org <sclorg@redhat.com>
    Config:
      Cmd:
      - /bin/sh
      - -c
      - $STI_SCRIPTS_PATH/usage
      Entrypoint:
      - container-entrypoint
      Env:
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
      ExposedPorts:
        8080/tcp: {}
      Image: d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
      Labels:
        build-date: 2015-12-23
```

```
        io.k8s.description: Platform for building and running Ruby 2.2 applications
        io.k8s.display-name: Ruby 2.2
        io.openshift.builder-base-version: 8d95148
        io.openshift.builder-version: 8847438ba06307f86ac877465eadc835201241df
        io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
        io.openshift.tags: builder,ruby,ruby22
        io.s2i.scripts-url: image:///usr/libexec/s2i
        license: GPLv2
        name: CentOS Base Image
        vendor: CentOS
      User: "1001"
      WorkingDir: /opt/app-root/src
    ContainerConfig: {}
    Created: 2016-01-26T21:07:27Z
    DockerVersion: 1.8.2-el7
    Id: 57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
    Parent: d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
    Size: 430037130
    apiVersion: "1.0"
    kind: DockerImage
  dockerImageMetadataVersion: "1.0"
  dockerImageReference: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
  metadata:
    creationTimestamp: 2016-01-29T13:17:45Z
    name: sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
    resourceVersion: "352"
    uid: af2e7a0c-c68a-11e5-8a99-525400f25e34
kind: ImageStreamImage
metadata:
  creationTimestamp: null
  name: ruby@3a335d7
  namespace: openshift
  selflink: /oapi/v1/namespaces/openshift/imagestreamimages/ruby@3a335d7
```

13.3. KUBERNETES 资源使用镜像流

作为 OpenShift Container Platform 的原生资源，镜像流可以直接与 OpenShift Container Platform 中的所有其他原生资源一起来工作（如构建或部署）。当前，它也可能可以和其与原生 Kubernetes 资源一起工作，如作业、复制控制器、副本设置或 Kubernetes 部署。

集群管理员可以精确配置可以使用什么资源。

启用后，可以将镜像流的引用放在资源的 **image** 字段中。使用此功能时，只能引用位于与资源相同的项目中的镜像流。镜像流引用必须包含单个片段值，如 **ruby:2.5**，其中 **ruby** 是镜像流的名称，它具有名为 **2.5** 的标签，并位于与进行引用的资源相同的项目中。

有两种方法可以做到这一点：

1. 启用针对特定资源的镜像流解析。这只允许此资源使用 image 字段中的镜像流名称。
2. 在镜像流上启用镜像流解析。这允许指向此镜像流的所有资源在 image 字段中使用它。

这两个操作都可使用 **oc set image-lookup** 来完成。例如，以下命令允许所有资源引用名为 **mysql** 的镜像流：

```
$ oc set image-lookup mysql
```

这会将 **Imagestream.spec.lookupPolicy.local** 字段设置为 true。

启用镜像查询的镜像流

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/display-name: mysql
  name: mysql
  namespace: myproject
spec:
  lookupPolicy:
    local: true
```

启用后，会为镜像流中的所有标签启用此行为。

您可以查询镜像流并查看是否设置了选项：

```
$ oc set image-lookup
```

您还可以在特定资源上启用镜像查找。此命令允许名为 **mysql** 的 Kubernetes 部署使用镜像流：

```
$ oc set image-lookup deploy/mysql
```

这会在部署上设置 **alpha.image.policy.openshift.io/resolve-names** 注解。

启用镜像查询部署

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: myproject
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        alpha.image.policy.openshift.io/resolve-names: '*'
    spec:
      containers:
      - image: mysql:latest
        imagePullPolicy: Always
        name: mysql
```

要禁用镜像查找，使用 **--enabled=false** ：

```
$ oc set image-lookup deploy/mysql --enabled=false
```

13.4. 镜像拉取(PULL)策略

Pod 中的每个容器均有容器镜像。您创建了镜像并将其推送（push）到 registry 后，即可在 Pod 中引用它。

当 OpenShift Container Platform 创建容器时，会使用容器的 **imagePullPolicy** 来决定是否应在启动容器前拉取(pull)镜像。**imagePullPolicy** 有三个可能的值：

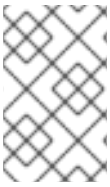
- **Always** - 始终拉取（pull）镜像。
- **IfNotPresent** - 只有节点上不存在的镜像时才拉取镜像。
- **Never** - 永不拉取(pull)镜像。

小心

您可以在全局构建配置中启用 **forcePull** 标志，以便在每次构建启动时强制从 registry 刷新镜像。这会强制镜像访问每个构建进行检查，从而可以避免用户访问他们没有访问权限的镜像的构建节点本地缓存。

如果没有指定容器的 **imagePullPolicy** 参数，OpenShift Container Platform 会根据镜像标签来设置它：

1. 如果标签是 **latest**，OpenShift Container Platform 会将 **imagePullPolicy** 默认设置为 **Always**。
2. 否则，OpenShift Container Platform 会将 **imagePullPolicy** 默认设置为 **IfNotPresent**。



注意

使用 **Never** Image Pull 策略时，您可以确保私有镜像只能供带有凭证的 pod 使用，以使用 AlwaysPullImages 准入控制器 拉取这些镜像。如果没有启用此准入控制器，节点上的任何用户的所有 pod 都可使用该镜像，而无需对镜像进行任何授权检查。

13.5. 访问内部 REGISTRY

您可以直接访问 OpenShift Container Platform 的内部 registry 来推送或拉取镜像。例如，如果您想 通过手动推送镜像来创建镜像流，或者只是直接进行 **docker pull**，则这很有用。

内部 registry 使用与 OpenShift Container Platform API 相同的令牌进行身份验证。要针对内部 registry 执行 **docker 登录**，您可以选择任何用户名和密码，但密码必须是有效的 OpenShift Container Platform 令牌。

登录到内部 registry：

1. 登录到 OpenShift Container Platform：

```
$ oc login
```

2. 获取您的访问令牌：

```
$ oc whoami -t
```

3. 使用该令牌登录到内部 registry。在您的系统中必须安装 **docker** ：

```
$ docker login -u <user_name> -e <email_address> \
    -p <token_value> <registry_server>:<port>
```

注意

如果您不知道要使用的 registry IP 或主机名和端口，请联络您的集群管理员。

若要拉取镜像，经过身份验证的用户必须具有所请求的 **imagestreams/layers** 的 **get** 权限。若要推送镜像，经过身份验证的用户必须具有所请求的 **imagestreams/layers** 的 **update** 权限。

默认情况下，一个项目中的所有服务帐户都有权拉取同一项目中的任何镜像，而 **builder** 服务帐户则有权在同一项目中推送任何镜像。

13.5.1. 列出软件仓库

/v2/_catalog 端点支持存储库列表或镜像流名称。

唯一的要求是经过身份验证的用户必须具有整个集群中 **镜像流**的**列表**权限。

要为用户授予列出镜像流的权限，请运行：

```
$ oc adm policy add-cluster-role-to-user registry-viewer user
```

列出软件仓库：

```
$ oc login -u user
$ curl -v -u unused:$(oc whoami -t) https://<registry_server>:<port>/v2/_catalog?n=100
```

重要

对于集群中的大量镜像流，这个 API 调用非常昂贵。建议您使用分页而不是列出所有镜像流。

13.6. 使用镜像提取 SECRET

Docker registries 可以被加以保护，以防止未授权方访问某些镜像。如果您在使用 OpenShift Container Platform 的内部 registry，且从位于同一项目中的镜像流拉取（pull），则您的 Pod 服务帐户应已具备正确权限，且无需额外操作。

然而，对于其他场景，例如在 OpenShift Container Platform 项目间或从安全 registry 引用镜像，则还需其他配置步骤。以下小节详细介绍了这些场景及其所需步骤。

13.6.1. 允许 Pod 在项目间引用镜像

使用内部 registry 时，要允许 **project-a** 中的 pod 引用 **project-b** 中的镜像，**project-a** 中的服务帐户必须绑定到 project-b 中的 **system:image-puller** 角色：

```
$ oc policy add-role-to-user \
    system:image-puller system:serviceaccount:project-a:default \
    --namespace=project-b
```

添加该角色后，**project-a** 中引用默认服务帐户的 pod 能够从 **project-b** 拉取（pull）镜像。

要允许访问 **project-a** 中的任意服务帐户，请使用组：

```
$ oc policy add-role-to-group \
    system:image-puller system:serviceaccounts:project-a \
    --namespace=project-b
```

13.6.2. 允许 Pod 引用其他安全 registry 中的镜像

.dockercfg 文件（或为较新的 Docker 客户端的 ***$HOME/.docker/config.json***）是一个 Docker 凭证文件，如果您之前已登录安全或不安全的 registry，则该文件会保存您的信息。

要拉取(pull)并非来自 OpenShift Container Platform 内部 registry 的安全容器镜像，您必须从 Docker 凭证创建一个 *pull secret*，并将其添加到您的服务帐户中。

如果您已经为安全 registry 有一个 ***.dockercfg*** 文件，则可运行以下命令从该文件中创建 secret：

```
$ oc create secret generic <pull_secret_name> \
    --from-file=.dockercfg=<path/to/.dockercfg> \
    --type=kubernetes.io/dockercfg
```

或者，如果您有一个 ***$HOME/.docker/config.json*** 文件：

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

如果您还没有安全 registry 的 Docker 凭证文件，则可运行以下命令创建一个 secret：

```
$ oc create secret docker-registry <pull_secret_name> \
    --docker-server=<registry_server> \
    --docker-username=<user_name> \
    --docker-password=<password> \
    --docker-email=<email>
```

要使用 secret 为 Pod 拉取镜像，您必须将 secret 添加到您的服务帐户中。本例中服务帐户的名称应与 Pod 使用的服务帐户的名称匹配；**default** 是默认服务帐户：

```
$ oc secrets link default <pull_secret_name> --for=pull
```

要使用 secret 来推送和拉取(pull)构建镜像，该 secret 必须可在 pod 内挂载。您可通过运行以下命令实现这一目的：

```
$ oc secrets link builder <pull_secret_name>
```

13.6.2.1. 使用委托身份验证从私有 registry 拉取(pull)

私有 registry 可将身份验证委托给单独服务。这种情况下，必须为身份验证和 registry 端点定义镜像 pull secret。

注意

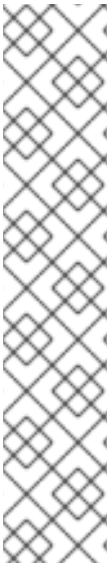
Red Hat Container Catalog 中的第三方镜像由 Red Hat Connect Partner Registry(**registry.connect.redhat.com**)提供。此 registry 将身份验证委托给 **sso.redhat.com**，因此适用以下步骤。

1. 为委托的身份验证服务器创建 secret：

```
$ oc create secret docker-registry \
    --docker-server=sso.redhat.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    redhat-connect-sso

secret/redhat-connect-sso
```

2. 为私有 registry 创建 secret：

```
$ oc create secret docker-registry \
    --docker-server=privateregistry.example.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    private-registry

secret/private-registry
```



注意

Red Hat Connect Partner Registry(**registry.connect.redhat.com**)不接受自动生成的 **dockercfg** secret 类型(BZ#1476330)。必须使用 **docker login** 命令生成的文件来创建基于文件的通用 secret：

```
$ docker login registry.connect.redhat.com --username developer@example.com

Password: **************
Login Succeeded

$ oc create secret generic redhat-connect --from-
file=.dockerconfigjson=.docker/config.json

$ oc secrets link default redhat-connect --for=pull
```

13.7. 导入标签和镜像元数据

镜像流可以被配置为从外部容器镜像 registry 中的镜像存储库导入标签和镜像元数据。您可以使用几种不同方法进行此操作。

- 您可以使用 **--from** 选项使用 **oc import-image** 命令手动导入 tag 和镜像信息：

```
$ oc import-image <image_stream_name>[:<tag>] --from=<docker_image_repo> --confirm
```

例如：

```
$ oc import-image my-ruby --from=docker.io/openshift/ruby-20-centos7 --confirm
The import completed successfully.

Name:   my-ruby
```

```
Created:  Less than a second ago
Labels:   <none>
Annotations:  openshift.io/image.dockerRepositoryCheck=2016-05-06T20:59:30Z
Docker Pull Spec: 172.30.94.234:5000/demo-project/my-ruby

Tag Spec     Created   PullSpec       Image
latest docker.io/openshift/ruby-20-centos7 Less than a second ago docker.io/openshift/ruby-20-centos7@sha256:772c5bf9b2d1e8... <same>
```

您还可以添加 **--all** 标志来导入镜像的所有标签，而不只导入 **latest**。

- 与 OpenShift Container Platform 中的大多数对象一样，您还可以将 JSON 或 YAML 定义写入文件，然后使用 CLI 创建对象。将 **spec.dockerImageRepository** 字段设置为镜像的 Docker pull spec：

```
apiVersion: "v1"
kind: "ImageStream"
metadata:
  name: "my-ruby"
spec:
  dockerImageRepository: "docker.io/openshift/ruby-20-centos7"
```

然后，创建对象：

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

当您创建引用外部 Docker registry 中的镜像的镜像流时，OpenShift Container Platform 会在短时间内与外部 registry 通信，以便获取镜像的最新信息。

同步 tag 和镜像元数据后，镜像流对象将类似如下：

```
apiVersion: v1
kind: ImageStream
metadata:
  name: my-ruby
  namespace: demo-project
  selflink: /oapi/v1/namespaces/demo-project/imagestreams/my-ruby
  uid: 5b9bd745-13d2-11e6-9a86-0ada84b8265d
  resourceVersion: '4699413'
  generation: 2
  creationTimestamp: '2016-05-06T21:34:48Z'
  annotations:
    openshift.io/image.dockerRepositoryCheck: '2016-05-06T21:34:48Z'
spec:
  dockerImageRepository: docker.io/openshift/ruby-20-centos7
  tags:
    -
      name: latest
      annotations: null
      from:
        kind: DockerImage
        name: 'docker.io/openshift/ruby-20-centos7:latest'
      generation: 2
      importPolicy: {  }
status:
```

```
  dockerImageRepository: '172.30.94.234:5000/demo-project/my-ruby'
  tags:
    -
      tag: latest
      items:
        -
          created: '2016-05-06T21:34:48Z'
          dockerImageReference: 'docker.io/openshift/ruby-20-
centos7@sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc3'
          image: 'sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc3'
          generation: 2
```

您可以将标签设置为以调度的间隔查询外部 registry，从而通过设置 **--scheduled=true** 标志和 **oc tag** 命令（如将标签添加到镜像流中所述） 来同步标签和镜像元数据。

另外，您可以在标签的定义中将 **importPolicy.scheduled** 设置为 **true** ：

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
spec:
  tags:
  - from:
      kind: DockerImage
      name: openshift/ruby-20-centos7
    name: latest
    importPolicy:
      scheduled: true
```

13.7.1. 从 Insecure Registries 导入镜像

镜像流可以被配置为从不安全的镜像 registry 中导入标签和镜像元数据，如使用自签名证书签名的镜像元数据或使用普通 HTTP 而不是 HTTPS 进行签名的镜像元数据。

要配置此功能，请添加 **openshift.io/image.insecureRepository** 注释，并将它设为 **true**。此设置会在连接到 registry 时绕过证书验证：

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  annotations:
    openshift.io/image.insecureRepository: "true" (1)
  spec:
    dockerImageRepository: my.repo.com:5000/myimage
```

(1) 将 **openshift.io/image.insecureRepository** 注解设置为 **true**

重要

这个选项指示集成 registry 在提供镜像流时回退到镜像流中标记的任何外部镜像的不安全传输，而这存在危险。如果可能，通过将 **istag** 标记为 insecure 来避免这一风险。

重要

以上定义仅影响导入标签和镜像元数据。要使此镜像在集群中使用（例如，要执行 **docker pull**），以下之一必须为 true：

1. 每个节点都使用 **--insecure-registry** 标志配置 Docker，与 **dockerImageRepository** 的 registry 部分匹配。如需更多信息，请参阅 主机准备。
2. 每个 **istag** 规格都必须将 **referencePolicy.type** 设置为 **Local**。如需更多信息，请参阅参考资料策略。

13.7.1.1. 镜像流标签策略

13.7.1.1.1. 不安全的标签导入策略

以上注解适用于特定 **ImageStream** 的所有镜像和标签。若要进行精细控制，可以在 **istags** 上设置策略。在标签的定义中将 **importPolicy.insecure** 设置为 **true**，以便只对该标签下的镜像进行回退到不安全传输。

注意

当镜像流被标注为不安全或者 **istag** 具有不安全的导入策略时，会在特定 **istag** 下启用回退到不安全的传输。**importPolicy.insecure** 设置为 **false** 可能无法覆盖镜像流注解。

13.7.1.1.2. 参考策略

Reference Policy 允许您指定引用此镜像流标签的资源的位置。它只适用于从外部 registry 导入的镜像。有两个选项可供选择：**Local** 和 **Source**.

Source 策略指示客户端直接从镜像的源 registry 中拉取。除非镜像由集群管理，否则不会涉及集成的 registry。（不是外部镜像。） 这是默认策略。

Local 策略指示客户端始终从集成的 registry 中拉取。如果要在不修改 Docker 守护进程设置的情况下从外部不安全的 registry 拉取(pull)非常有用。

此策略仅影响镜像流标签的使用。使用其外部 registry 位置直接引用或拉取镜像的组件或操作不会重定向到内部注册表。

registry 的 pull-through 功能 为客户端提供远程镜像。此功能（默认情况下为启用）必须启用本地参考策略。另外，默认会镜像所有 blobs，以便以后更快地访问。

您可以在镜像流标签规格中将策略设置为 **referencePolicy.type**。

带有本地参考策略的 Insecure Tag 示例

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  tags:
  - from:
      kind: DockerImage
      name: my.repo.com:5000/myimage
    name: mytag
```

```
importPolicy:
  insecure: true (1)
referencePolicy:
  type: Local (2)
```

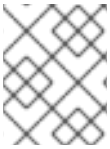
1. 将 tag **mytag** 设置为使用到该 registry 的不安全连接。
2. 设置 tag **mytag** 以使用集成 registry 来拉取外部镜像。如果引用策略类型设为 **Source**，客户端直接从 **my.repo.com:5000/myimage** 获取镜像。

13.7.2. 从私有 registry 导入镜像

镜像流可以配置为从私有镜像 registry 中导入标签和镜像元数据，从而需要身份验证。

要配置此功能，您需要创建一个用于存储您的凭据的 secret。有关使用 **oc create secret** 命令创建 secret 的说明，请参阅允许 Pod 引用其他安全 Registries 中的镜像 。

配置 secret 后，继续创建新镜像流或使用 **oc import-image** 命令。在导入过程中，OpenShift Container Platform 会提取 secret，并将其提供给远程方。



注意

从不安全的 registry 导入时，secret 中定义的 registry URL 必须包含 **:80** 端口后缀，或者在尝试从 registry 导入时不使用 secret。

13.7.3. 为外部 registry 添加可信证书

如果您要从中导入的 registry 使用标准证书颁发机构签名的证书，您需要明确将系统配置为信任 registry 的证书颁发机构或签名机构。这可以通过将 CA 证书或 registry 证书添加到运行 registry 导入控制器（通常是 master 节点）的主机系统中。

您必须在主机系统中分别将证书或 CA 证书添加到 **/etc/pki/tls/certs** 或 **/etc/pki/ca-trust** 中。您还需要在红帽分发版本中运行 **update-ca-trust** 命令，然后重启 master 服务以获取证书更改。

您还可以通过在 master 配置文件的 **Image Policy Configuration** 部分中设置 **AdditionalTrustedCA** 参数，将 registry 导入控制器指向另一个文件系统路径。

13.7.4. 在项目间导入镜像

镜像流可以被配置为从内部 registry 导入标签和镜像元数据，但来自不同项目。推荐的方法是使用 **oc tag** 命令，如为镜像流添加标签所示：

```
$ oc tag <source_project>/<image_stream>:<tag> <new_image_stream>:<new_tag>
```

另一种方法是使用 pull spec 手动从其他项目导入镜像：

警告

强烈建议不要使用以下方法，只有在以前的使用 **oc tag** 不足时才应使用。

1. 首先，添加所需的 策略 来访问其他项目：

```
$ oc policy add-role-to-group \
    system:image-puller \
    system:serviceaccounts:<destination_project> \
    -n <source_project>
```

   这允许 **<destination_project>** 从 **<source_project>** 拉取镜像。

2. 使用策略，您可以手动导入镜像：

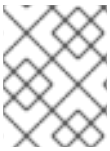
```
$ oc import-image <new_image_stream> --confirm \
    --from=<docker_registry>/<source_project>/<image_stream>
```

13.7.5. 通过手动推送镜像来创建镜像流

也可以通过手动将镜像推送到内部 registry 来自动创建镜像流。这只在使用 OpenShift Container Platform 内部 registry 时实现。

在执行此步骤前，必须满足以下条件：

- 您要推送到的目标项目必须已经存在。
- 用户必须获得该项目中的 **{get, update} "imagestream/layers"** 授权。另外，因为镜像流不存在，用户必须在该项目中被授权 **{create} "imagestream"**。如果您是项目管理员，则有这些权限。



注意

system:image-pusher 角色不授予创建新镜像流的权限，只是将镜像推送到现有镜像流，因此不能用来将镜像推送到还不存在的镜像流，除非还授予用户其他权限。

通过手动推送镜像来创建镜像流：

1. 首先，登录到内部 registry。
2. 然后，使用适当的内部 registry 位置标记您的镜像。例如，如果您已在本地拉取 **docker.io/centos:centos7** 镜像：

```
$ docker tag docker.io/centos:centos7 172.30.48.125:5000/test/my-image
```

3. 最后，将镜像推送到内部 registry。例如：

```
$ docker push 172.30.48.125:5000/test/my-image
The push refers to a repository [172.30.48.125:5000/test/my-image] (len: 1)
```

```
c8a648134623: Pushed
2bf4902415e3: Pushed
latest: digest:
sha256:be8bc4068b2f60cf274fc216e4caba6aa845fff5fa29139e6e7497bb57e48d67 size:
6273
```

4. 验证镜像流是否已创建：

```
$ oc get is
NAME       DOCKER REPO                       TAGS      UPDATED
my-image   172.30.48.125:5000/test/my-image   latest    3 seconds ago
```

13.8. 传输镜像

要将镜像从一个容器镜像 registry 移动到另一个容器镜像 registry，请使用 **oc image mirror** 命令。镜像从 registry 流传输到 registry，而不存储在本地。

例如，要将镜像从 **Docker Hub** 复制到集成的 registry 中，请使用以下命令：

```
$ oc image mirror docker.io/library/busybox:latest 172.30.0.0/16/myproject/toybox:latest
```

重要

如果您在源或目标中使用 **docker.io**，则无法省略 **docker.io** 和 **library** 部分。如果您想要获取 latest 标签，请不要省略 **latest**。

镜像可以一次性复制到多个位置。要做到这一点，您必须指定几个目的地：

```
$ oc image mirror 172.30.0.0/16/myproject/busybox:latest docker.io/myrepository/busybox:stable
docker.io/myrepository/toybox:dev
```

注意

oc image mirror 在本地运行，而不是在 OpenShift Container Platform 集群中运行。因此，**oc image mirror** 必须有权访问源和目标 registry。

如果容器镜像 registry 需要进行身份验证来拉取或推送镜像，您需要在执行 **oc image mirror** 命令前使用 **docker login** 命令手动登录。如果您没有对 docker 二进制文件和守护进程的访问权限，例如，由于您在 Jenkins 代理镜像中使用 命令，您可以在调用 **oc image mirror** 之前手动提供包含有效凭证的 **.docker/config.json** 文件。

13.9. 在镜像流更改时触发更新

当更新镜像流标签以指向新镜像时，OpenShift Container Platform 可以自动采取行动将新镜像推出到使用旧镜像的资源。这可根据引用镜像流标签的资源类型以不同的方式进行配置。

13.9.1. OpenShift 资源

OpenShift DeploymentConfig 和 BuildConfig 可通过更改 ImageStreamTags 自动触发。可使用更新的 ImageStreamTag 引用的镜像的新值运行触发的操作。有关使用此功能的更多详细信息，请参阅 BuildConfig 触发器和 DeploymentConfig 触发器的文档。

13.9.2. Kubernetes 资源

与 DeploymentConfig 和 BuildConfigs 不同，它包括作为 API 定义一组字段来控制触发器的字段，Kubernetes 资源没有用于触发的字段。OpenShift Container Platform 使用注解来允许用户请求触发器。该注解定义如下：

```
Key: image.openshift.io/triggers
Value: array of triggers, where each item has the schema:
[
 {
   "from" :{
     "kind": "ImageStreamTag", // required, the resource to trigger from, must be ImageStreamTag
     "name": "example:latest", // required, the name of an ImageStreamTag
     "namespace": "myapp",     // optional, defaults to the namespace of the object
   },
   // required, JSON path to change
   // Note that this field is limited today, and only accepts a very specific set
   // of inputs (a JSON path expression that precisely matches a container by ID or index).
   // For pods this would be "spec.containers[?(@.name='web')].image".
   "fieldPath": "spec.template.spec.containers[?(@.name='web')].image",
   // optional, set to true to temporarily disable this trigger.
   "paused": "false"
 },
 ...
]
```

当 OpenShift Container Platform 看到包含 pod 模板的核心 Kubernetes 资源之一时（即，只有 CronJob、Deployment、StatefulSets、DaemonSets、Job、ReplicaSet、ReplicaSets、ReplicationController 和 Pod）以及此注解，它会尝试使用当前与触发器引用的 ImageStreamTag 关联的镜像更新对象。更新针对指定的 **fieldPath** 进行。

在以下示例中，当 **example:latest** 镜像流标签被更新时，触发器会触发。触发后，**web** 容器的 pod 模板镜像引用会使用新的镜像值更新。如果 pod 模板是 Deployment 定义的一部分，则对 pod 模板的更改会自动触发部署，从而有效地推出新镜像。

```
image.openshift.io/triggers=[{"from":
{"kind":"ImageStreamTag","name":"example:latest"},"fieldPath":"spec.template.spec.containers[?
(@.name='web')].image"}]
```

在 Deployment 中添加镜像触发器时，也可以使用 **oc set triggers** 命令。例如，以下命令将镜像更改触发器添加到名为 **example** 的 Deployment 中，以便在更新 **example:latest** 镜像流标签时，部署中的 **web** 容器使用新镜像值更新：

```
$ oc set triggers deploy/example --from-image=example:latest -c web
```

除非 Deployment 已暂停，否则此 pod 模板更新会自动导致使用新镜像值进行部署。

13.10. 编写镜像流定义

您可以通过为整个镜像流编写镜像流定义来定义镜像流。这可让您在不运行 **oc** 命令的情况下将定义分发到不同的集群。

镜像流定义指定有关镜像流和要导入的特定标签的信息。

定义镜像流对象

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
  annotations:
    openshift.io/display-name: Ruby (1)
spec:
  tags:
    - name: '2.0' (2)
      annotations:
        openshift.io/display-name: Ruby 2.0 (3)
        description: >- (4)
          Build and run Ruby 2.0 applications on CentOS 7. For more information
          about using this builder image, including OpenShift considerations,
          see
          https://github.com/sclorg/s2i-ruby-container/tree/master/2.0/README.md.
        iconClass: icon-ruby (5)
        sampleRepo: 'https://github.com/sclorg/ruby-ex.git' (6)
        tags: 'builder,ruby' (7)
        supports: 'ruby' (8)
        version: '2.0' (9)
      from:
        kind: DockerImage (10)
        name: 'docker.io/openshift/ruby-20-centos7:latest' (11)
```

(1) 整个镜像流的简短、用户友好的名称。

(2) 该标签被称为版本。标签会出现在下拉菜单中。

(3) 此标签在镜像流中的用户友好名称。这应该是简要，并在适当的时候包括版本信息。

(4) 标签的描述，其中包含充足的详细信息，供用户了解提供该镜像的内容。它可以包含其他指令的链接。将描述内容限制为只包括几个句子。

(5) 此标签的图标。尽可能从现有 徽标图标 中进行选择。也可以使用来自 FontAwesome 和 Patternfly 的图标。另外，也可通过 CSS 自定义 提供图标，它们可添加到使用镜像流的 OpenShift Container Platform 集群中。您必须指定一个存在的图标类，或者防止回退到通用图标。

(6) 用于此构建器镜像标签的源存储库的 URL，并产生运行应用程序的示例。

(7) 镜像流标签关联的类别。需要 builder 标签才能显示在目录中。添加标签，将其与其中一个提供的目录类别相关联。请参见控制台常量文件的 **CATALOG_CATEGORIES** 中的 **id** 和 **categoryAliases**。此外，还可为整个集群自定义类别。

(8) 此镜像支持的语言。**oc new-app** 调用过程中使用这个值尝试将潜在的构建器镜像与提供的源存储库匹配。

(9) 此标签的版本信息。

(10) 此镜像流标签所引用的对象类型。有效值为：**DockerImage**, **ImageStreamTag**, 和 **ImageStreamImage**.

11 此镜像流标签导入的对象。

如需有关 **ImageStream** 中定义的字段的更多信息，请参阅镜像流 API 和 Imagestream Tag API。

第 14 章 配额和限值范围

14.1. 概述

通过使用配额和限值范围，集群管理员可以设置限制来限制项目中所用计算资源的数量或计算资源的数量。这有助于集群管理员更好地管理和分配所有项目的资源，并确保没有项目不适合于集群大小。

作为开发人员，您还可以在 Pod 和容器一级上设置计算资源的请求和限值。

以下小节帮助您了解如何检查配额和限值范围设置、它们可能会限制哪些类型，以及如何在自己的 pod 和容器中请求或限制计算资源。

14.2. 配额

资源配额 由 **ResourceQuota** 对象定义，提供约束来限制各个项目的聚合资源消耗。它可根据类型限制项目中创建的对象数量，以及该项目中资源可以消耗的计算资源和存储的总和。

注意

配额由集群管理员设置，并可限定到给定项目。

14.2.1. 查看配额

您可以在 Web 控制台导航到项目的 **Quota** 页面，查看与项目配额中定义的硬限值相关的使用量统计。

您还可以使用 CLI 查看配额详情：

1. 首先，获取项目中定义的配额列表。例如，对于名为 **demoproject** 的项目：

```
$ oc get quota -n demoproject
NAME              AGE
besteffort        11m
compute-resources   2m
core-object-counts  29m
```

2. 然后，描述您需要的配额，如 **core-object-counts** 配额：

```
$ oc describe quota core-object-counts -n demoproject
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
configmaps  3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

可以通过在对象上运行 **oc get --export** 来查看完整的配额定义。以下显示了一些配额定义示例：

core-object-counts.yaml

```
apiVersion: v1
```

```
kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" (1)
    persistentvolumeclaims: "4" (2)
    replicationcontrollers: "20" (3)
    secrets: "10" (4)
    services: "10" (5)
```

1. 项目中可以存在的 **ConfigMap** 对象的总数。
2. 项目中可以存在的持久性卷声明 (PVC) 的总数。
3. 项目中可以存在的复制控制器的总数。
4. 项目中可以存在的 secret 的总数。
5. 项目中可以存在的服务总数。

openshift-object-counts.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: openshift-object-counts
spec:
  hard:
    openshift.io/imagestreams: "10" (1)
```

1. 项目中可以存在的镜像流的总数。

compute-resources.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" (1)
    requests.cpu: "1" (2)
    requests.memory: 1Gi (3)
    requests.ephemeral-storage: 2Gi (4)
    limits.cpu: "2" (5)
    limits.memory: 2Gi (6)
    limits.ephemeral-storage: 4Gi (7)
```

1. 项目中可以存在的处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 请求总和不能超过 1 个内核。

3 在非终端状态的所有 Pod 中，内存请求总和不能超过 1Gi。

4 在非终端状态的所有 Pod 中，临时存储请求总和不能超过 2Gi。

5 在非终端状态的所有 Pod 中，CPU 限值总和不能超过 2 个内核。

6 在非终端状态的所有 Pod 中，内存限值总和不能超过 2Gi。

7 在非终端状态的所有 Pod 中，临时存储限值总和不能超过 4Gi。

besteffort.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
  - BestEffort 2
```

1 项目中可以存在的具有 **BestEffort** 服务质量的非终端状态 Pod 的总数。

2 将配额仅限为在内存或 CPU 方面具有 **BestEffort** 服务质量的匹配 Pod。

compute-resources-long-running.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
    limits.ephemeral-storage: "4Gi" 4
  scopes:
  - NotTerminating 5
```

1 处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 限值总和不能超过这个值。

3 在非终端状态的所有 Pod 中，内存限值总和不能超过这个值。

4 在非终端状态的所有 Pod 中，临时存储限值总和不能超过这个值。

5 将配额仅限为 **spec.activeDeadlineSeconds** 设为 **nil** 的匹配 Pod。构建 Pod 会归入 **NotTerminating** 下，除非应用了 **RestartNever** 策略。

compute-resources-time-bound.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" (1)
    limits.cpu: "1" (2)
    limits.memory: "1Gi" (3)
    limits.ephemeral-storage: "1Gi" (4)
  scopes:
  - Terminating (5)
```

1. 处于非终端状态的 Pod 总数。
2. 在非终端状态的所有 Pod 中，CPU 限值总和不能超过这个值。
3. 在非终端状态的所有 Pod 中，内存限值总和不能超过这个值。
4. 在非终端状态的所有 Pod 中，临时存储限值总和不能超过这个值。
5. 将配额仅限为 **spec.activeDeadlineSeconds >=0** 的匹配 Pod。例如，此配额适用于构建或部署器 Pod，而非 Web 服务器或数据库等长时间运行的 Pod。

storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" (1)
    requests.storage: "50Gi" (2)
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi" (3)
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" (4)
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" (5)
    bronze.storageclass.storage.k8s.io/requests.storage: "0" (6)
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" (7)
```

1. 项目中的持久性卷声明总数
2. 在一个项目中的所有持久性卷声明中，请求的存储总和不能超过这个值。
3. 在一个项目中的所有持久性卷声明中，金级存储类中请求的存储总和不能超过这个值。
4. 在一个项目中的所有持久性卷声明中，银级存储类中请求的存储总和不能超过这个值。
5. 在一个项目中的所有持久性卷声明中，银级存储类中声明总数不能超过这个值。
6. 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 0，则表示铜级存储类无法请求存储。

0，则表示铜级存储类无法请求存储。

7 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 0，则表示铜级存储类无法创建声明。

14.2.2. 由配额管理的资源

下文描述了可能通过配额管理的计算资源和对象类型的集合。

注意

如果 **status.phase in (Failed, Succeeded)** 为 true，则 Pod 处于终端状态。

表 14.1. 由配额管理的计算资源

资源名称	描述
cpu	非终端状态的所有 Pod 的 CPU 请求总和不能超过这个值。**cpu** 和 **requests.cpu** 的值相同，并可互换使用。
memory	非终端状态的所有 Pod 的内存请求总和不能超过这个值。**memory** 和 **requests.memory** 的值相同，并可互换使用。
ephemeral-storage	非终端状态的所有 Pod 的本地临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。
requests.cpu	非终端状态的所有 Pod 的 CPU 请求总和不能超过这个值。**cpu** 和 **requests.cpu** 的值相同，并可互换使用。
requests.memory	非终端状态的所有 Pod 的内存请求总和不能超过这个值。**memory** 和 **requests.memory** 的值相同，并可互换使用。
requests.ephemeral-storage	非终端状态的所有 Pod 的临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。
limits.cpu	非终端状态的所有 Pod 的 CPU 限值总和不能超过这个值。
limits.memory	非终端状态的所有 Pod 的内存限值总和不能超过这个值。
limits.ephemeral-storage	非终端状态的所有 Pod 的临时存储限值总和不能超过这个值。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。

表 14.2. 由配额管理的存储资源

资源名称	描述
requests.storage	处于任何状态的所有持久性卷声明的存储请求总和不能超过这个值。

资源名称	描述
PersistentVolumeClaims	项目中可以存在的持久性卷声明的总数。
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	在处于任何状态且具有匹配存储类的所有持久性卷声明中，存储请求总和不能超过这个值。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	项目中可以存在的具有匹配存储类的持久性卷声明的总数。

表 14.3. 由配额管理的对象计数

资源名称	描述
pods	项目中可以存在的处于非终端状态的 Pod 总数。
replicationcontrollers	项目中可以存在的复制控制器的总数。
resourcequotas	项目中可以存在的资源配额总数。
services	项目中可以存在的服务总数。
secrets	项目中可以存在的 secret 的总数。
configmaps	项目中可以存在的 **ConfigMap** 对象的总数。
PersistentVolumeClaims	项目中可以存在的持久性卷声明的总数。
openshift.io/imagestreams	项目中可以存在的镜像流的总数。

14.2.3. 配额范围

每个配额都有一组关联的*范围*。配额仅在与枚举的范围交集匹配时才会测量资源的使用量。

为配额添加范围会限制该配额可应用的资源集合。指定允许的集合之外的资源会导致验证错误。

影响范围	描述
Terminating	匹配 **spec.activeDeadlineSeconds >= 0** 的 Pod。

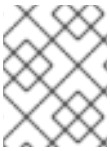
影响范围	描述
NotTerminating	匹配 **spec.activeDeadlineSeconds** 为 **nil** 的 Pod。
BestEffort	匹配带有最佳服务质量的 **cpu** 或**内存** 的 pod。如需更多提交计算资源，请参阅服务质量类。
NotBestEffort	匹配 **cpu** 和 **memory** 没有最佳服务质量的 Pod。

BestEffort 范围将配额仅限为限制以下资源：

- **pods**

Terminating、NotTerminating 和 NotBestEffort 范围将配额仅限为跟踪以下资源：

- **pods**
- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**
- **ephemeral-storage**
- **requests.ephemeral-storage**
- **limits.ephemeral-storage**



注意

只有在启用了临时存储技术预览功能时，才会应用临时存储请求和限值。此功能默认为禁用。

14.2.4. 配额强制

在项目中首次创建资源配额后，项目会限制您创建可能会违反配额约束的新资源，直到它计算了更新后的使用量统计。

在创建了配额并且更新了使用量统计后，项目会接受创建新的内容。当您创建或修改资源时，配额使用量会在请求创建或修改资源时立即递增。

在您删除资源时，配额使用量在下一次完整重新计算项目的配额统计时才会递减。如果项目修改超过配额使用量限制，服务器会拒绝该操作。返回了一条适当的错误消息，说明配额约束违反了，您当前观察到的用法统计位于系统中。

14.2.5. 请求与限制

在分配计算资源时，每个容器可能会为 CPU、内存和临时存储各自指定请求和限制值。配额可以限制任何这些值。

如果配额具有为 **requests.cpu** 或 **requests.memory** 指定的值，那么它要求每个传入的容器都明确请求这些资源。如果配额具有为 **limits.cpu** 或 **limits.memory** 的值，那么它要求每个传入的容器为那些资源指定一个显式限值。

有关设置 pod 和容器中的请求和限值的更多信息，请参阅计算资源。

14.3. 限制范围

一个限制范围，由 **LimitRange** 对象定义，在 pod、容器、镜像、镜像流和持久性卷声明一级的一个 项目中枚举的计算资源约束，并指定 pod、容器、镜像、镜像流和持久性卷声明一级可以消耗的资源数量。

要创建和修改资源的所有请求都会针对项目中的每个 **LimitRange** 对象进行评估。如果资源违反了任何限制，则会拒绝该资源。如果资源没有设置显式值，如果约束支持默认值，则默认值将应用到资源。

对于 CPU 和内存限值，如果您指定一个最大值，但没有指定最小限制，资源会消耗超过最大值的 CPU 和内存资源。

您可以使用临时存储技术预览功能指定临时存储的限值和请求。此功能默认为禁用。要启用此功能，请参阅为临时存储配置。

注意

限制范围由集群管理员设置，并可限定到给定项目。

14.3.1. 查看限制范围

您可以通过在 Web 控制台中导航到项目的 **Quota** 页面来查看项目中定义的任何限值范围。

您还可以通过执行以下步骤来使用 CLI 查看限制范围详情：

1. 获取项目中定义的限值范围对象列表。例如，对于名为 **demoproject** 的项目：

```
$ oc get limits -n demoproject
```

输出示例

```
NAME              AGE
resource-limits   6d
```

2. 描述限值范围。例如，对于名为 **resource-limits** 的限制范围：

```
$ oc describe limits resource-limits -n demoproject
```

输出示例

```
Name:                   resource-limits
Namespace:              demoproject
Type                    Resource    Min    Max    Default Request Default Limit   Max
Limit/Request Ratio
----                    --------    ---    ---    --------------- -------------   -----------------------
```

```
Pod                     cpu                 200m   2      -          -          -
Pod                     memory              6Mi    1Gi    -          -          -
Container               cpu                 100m   2      200m       300m       10
Container               memory              4Mi    1Gi    100Mi      200Mi      -
openshift.io/Image      storage             -      1Gi    -          -          -
openshift.io/ImageStream openshift.io/image -      12     -          -          -
openshift.io/ImageStream openshift.io/image-tags -     10     -          -          -
```

通过在对象中运行 **oc get --export** 可以查看完整的限制范围定义。下面显示了一个限制范围定义示例：

核心限制范围对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "core-resource-limits" 1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "200m" 4
        memory: "6Mi" 5
    - type: "Container"
      max:
        cpu: "2" 6
        memory: "1Gi" 7
      min:
        cpu: "100m" 8
        memory: "4Mi" 9
      default:
        cpu: "300m" 10
        memory: "200Mi" 11
      defaultRequest:
        cpu: "200m" 12
        memory: "100Mi" 13
      maxLimitRequestRatio:
        cpu: "10" 14
```

1. 限制范围对象的名称。
2. pod 可在所有容器间请求的最大 CPU 量。
3. pod 可在所有容器间请求的最大内存量。
4. pod 可在所有容器间请求的最小 CPU 量。如果没有设置 **min** 值，或者将 **min** 设置为 **0**，则结果为没有限制，pod 消耗的消耗可能会超过 **max** CPU 值。
5. pod 可在所有容器间请求的最小内存量。如果没有设置 **min** 值，或者将 **min** 设置为 **0**，则结果为没有限制，pod 消耗的消耗可能会超过 **max** 内存的值。

6. pod 中单个容器可以请求的最大 CPU 量。
7. pod 中单个容器可以请求的最大内存量。
8. pod 中单个容器可以请求的最小 CPU 量。如果没有设置 **min** 值，或者将 **min** 设置为 **0**，则结果为没有限制，pod 消耗的消耗可能会超过 **max** CPU 值。
9. pod 中单个容器可以请求的最小内存量。如果没有设置 **min** 值，或者将 **min** 设置为 **0**，则结果为没有限制，pod 消耗的消耗可能会超过 **max** 内存的值。
10. 如果没有在 pod 规格中指定限制，则容器的默认 CPU 限值。
11. 如果没有在 pod 规格中指定限制，则容器的默认内存限值。
12. 如果您没有在 pod 规格中指定请求，则容器的默认 CPU 请求。
13. 如果您没有在 pod 规格中指定请求，则容器的默认内存请求。
14. 容器最大的限制与请求的比率。

如需有关如何测量 CPU 和内存的更多信息，请参阅 Compute Resources。

14.3.2. 容器限制

支持的资源：

- CPU
- 内存

支持的限制：

根据容器，如果指定，则必须满足以下条件：

表 14.4. Container

约束	行为
Min	**Min[resource]** 小于或等于 **container.resources.requests[resource]** (必需)小于或等于 **container/resources.limits[resource]** (可选) 如果配置定义了 **min** CPU，则请求值必须大于 CPU 值。如果您没有设置 **min** 值，或将 **min** 设置为 **0**，则代表没有限制，pod 消耗的资源量可以超过 **max** 值。
Max	**container.resources.limits[resource]** （必需）小于或等于 **Max[resource]** 如果配置定义了 **max** CPU，则不需要定义 CPU 请求值。但是，您必须设置一个限制，用于满足在限制范围中指定的最大 CPU 约束。

约束	行为
MaxLimitRequestRatio	**MaxLimitRequestRatio[resource]** 小于或等于 (**container.resources.limits[resource]** / **container.resources.requests[resource]**) 如果限制范围定义了 **maxLimitRequestRatio** 约束，则任何新容器都必须具有 **request** 和 **limit** 值。另外，OpenShift Container Platform 会计算限制与请求的比率（**limit** 除以 **request**）。结果应该是大于 1 的整数。 例如，如果容器有 **cpu：500**（**limit** 值）和 **cpu：100**（**request** 值），**cpu** 的 limit-to-request 比率为 **5**。这个比例必须小于或等于 **maxLimitRequestRatio**。

支持的默认值：

Default[resource]

如果无，则默认为 **container.resources.limit[resource]** 作为指定的值。

默认请求[资源]

如果无，则默认为 **container.resources.requests[resource]** 作为指定的值。

14.3.3. Pod 限制

支持的资源：

- CPU
- 内存

支持的限制：

在 pod 中的所有容器中，需要满足以下条件：

表 14.5. Pod

约束	强制行为
Min	**Min[resource]** 小于或等于 **container.resources.requests[resource]** (必需)小于或等于 **container.resources.limits[resource]**。如果您没有设置 **min** 值，或将 **min** 设置为 **0**，则代表没有限制，pod 消耗的资源量可以超过 **max** 值。
Max	**container.resources.limits[resource]**（必需）小于或等于 **Max[resource]**。
MaxLimitRequestRatio	**MaxLimitRequestRatio[resource]** 小于或等于 (**container.resources.limits[resource]** / **container.resources.requests[resource]**)。

14.4. 计算资源

在节点上运行的每个容器都会消耗计算资源，这些资源是可处理、分配和使用的可测量数量。

在编写 pod 配置文件时，您可以选择性地指定 CPU、内存(RAM)和本地临时存储（如果管理员启用了临时存储技术预览）的数量，以便更好地在集群中调度 pod，并确保集群中调度 pod 的性能。

CPU 以名为 millicores 的单位来衡量。集群中的每个节点检查操作系统以确定节点上的 CPU 内核数，然后将该值乘以 1000 以表示其总容量。例如，如果某个节点有 2 个内核，则节点的 CPU 容量将代表为 2000m。如果您要使用单个内核的 1/10，它将表示为 100m。

内存和临时存储以字节为单位。另外，它可以被与 SI 后缀（E、P、T、G、M、K）或其电源一起使用（Ei、P、Ti、Gi、Mi Ki）。

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - image: openshift/hello-openshift
    name: hello-openshift
    resources:
      requests:
        cpu: 100m (1)
        memory: 200Mi (2)
        ephemeral-storage: 1Gi (3)
      limits:
        cpu: 200m (4)
        memory: 400Mi (5)
        ephemeral-storage: 2Gi (6)
```

1. 容器请求 100m CPU。
2. 容器请求 200Mi 内存。
3. 容器请求 1Gi 临时存储。您的管理员必须启用临时存储技术预览功能来指定这个参数值。
4. 容器限制 200m CPU。
5. 容器会限制 400Mi 内存。
6. 容器会限制 2Gi 临时存储。您的管理员必须启用临时存储技术预览功能来指定这个参数值。

14.4.1. CPU 请求

pod 中的每个容器可以指定其在节点上请求的 CPU 数量。调度程序使用 CPU 请求来查找适合容器的节点。

CPU 请求代表容器可能会消耗的最小 CPU 量，但如果没有 CPU 争用，它可以使用节点上的所有可用 CPU。如果节点上有 CPU 争用，则 CPU 请求将为系统上的所有容器提供相对权重，以获取有关容器可以使用的 CPU 时间。

在节点上，CPU 请求映射到内核 CFS 共享来强制实施此行为。

14.4.2. 查看计算资源

查看 pod 的计算资源：

```
$ oc describe pod ruby-hello-world-tfjxt
Name:       ruby-hello-world-tfjxt
Namespace:      default
Image(s):     ruby-hello-world
Node:       /
Labels:       run=ruby-hello-world
Status:       Pending
Reason:
Message:
IP:
Replication Controllers:  ruby-hello-world (1/1 replicas created)
Containers:
  ruby-hello-world:
    Container ID:
    Image ID:
    Image:    ruby-hello-world
    QoS Tier:
      cpu:  Burstable
      memory: Burstable
    Limits:
      cpu:  200m
      memory: 400Mi
      ephemeral-storage: 1Gi
    Requests:
      cpu:    100m
      memory:   200Mi
      ephemeral-storage: 2Gi
    State:    Waiting
    Ready:    False
    Restart Count:  0
    Environment Variables:
```

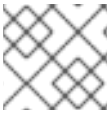
14.4.3. CPU 限制

pod 中的每个容器可以指定节点上要使用的 CPU 数量。CPU 限制控制容器独立于节点上争用的最大 CPU 数量。如果容器尝试超过指定的限制，系统会节流容器。这允许容器具有一致的服务级别，独立于调度到节点的 pod 数量。

14.4.4. 内存请求

默认情况下，容器可以尽可能消耗节点上的内存。为了提高集群中 pod 的放置，请指定运行容器所需的内存量。将 pod 绑定到节点前，调度程序会考虑可用的节点内存容量。即使指定请求时，容器仍然可以尽可能消耗节点上的内存。

14.4.5. 临时存储请求



注意

只有在管理员启用了临时存储技术预览功能时，才会应用此主题。

默认情况下，容器可以消耗节点上尽可能多的本地临时存储。为了提高集群中 pod 的放置，请指定要运行的容器所需的本地临时存储量。将 pod 绑定到节点前，调度程序会考虑可用的节点本地存储容量。即使指定请求，容器仍然可以尽可能消耗节点上的本地临时存储。

14.4.6. 内存限制

如果指定内存限制，您可以限制容器可以使用的内存量。例如，如果指定限制为 200Mi，则容器将限制为使用节点上的内存量。如果容器超过指定的内存限值，它将被终止，并可能因容器重启策略而重启。

14.4.7. 临时存储限值

注意

只有在管理员启用了临时存储技术预览功能时，才会应用此主题。

如果指定了临时存储限制，您可以限制容器可以使用的临时存储量。例如，如果您指定 2Gi 的限制，则容器将限制为使用节点上的这个临时存储。如果容器超过指定的内存限值，它将被终止，并可能因容器重启策略而重启。

14.4.8. Service Tiers 的质量

创建后，计算资源被分类为 *服务质量* (QoS)。有三个级别，每个层都基于为每个资源指定的请求和限制值：

服务质量	描述
BestEffort	未指定请求和限制时提供。
Burstable	指定了小于指定限制的请求时。
Guaranteed	指定了与可选指定请求相同的限制。

如果容器设置了请求和限值，这样可为每个计算资源产生不同的服务质量，则它将归类为 **Burstable**。

根据资源是否压缩，服务质量对不同资源有不同影响。CPU 是可压缩的资源，而 memory 则是不可压缩的资源。

使用 CPU 资源：

- **BestEffort CPU** 容器可以消耗节点上可用的 CPU 数量，但以最低优先级运行。
- 可以保证 **Burstable CPU** 容器获得请求的最小 CPU 数量，但可能或可能无法获得额外的 CPU 时间。过量 CPU 资源根据节点上所有容器间请求的数量进行分发。
- **Guaranteed CPU** 容器可以保证获得请求的数量，不再有其他数量，即使存在额外的 CPU 周期可用。这可独立于节点上的其他活动提供一致的性能级别。

使用内存资源：

- **BestEffort 内存** 容器可以消耗节点上可用的尽可能多的内存，但不能保证调度程序会将该容器放在具有足够内存来满足其需要的节点中。另外，如果节点上内存不足事件，则 **BestEffort** 容器的最大被终止机会。
- 在节点上调度 **Burstable 内存** 容器以获取所请求的内存量，但可能会消耗更多内存。如果节点上的内存事件不足，在尝试恢复内存时在 **BestEffort** 容器后终止 **Burstable** 容器。

- **Guaranteed 内存** 容器获得请求的内存量，但没有提供更多内存。如果出现内存不足事件，它只有在系统中没有更多的 **BestEffort** 或 **Burstable** 容器时才会被终止。

14.4.9. 通过 CLI 指定计算资源

通过 CLI 指定计算资源：

```
$ oc run ruby-hello-world --image=ruby-hello-world --limits=cpu=200m,memory=400Mi --
requests=cpu=100m,memory=200Mi
```

只有在管理员启用了临时存储技术预览功能时，才会应用临时存储。

14.5. 项目资源限制

集群管理员可按照项目设置资源限制。开发人员无法创建、编辑或删除这些限制，但 可以查看他们 有权访问的项目。

第 15 章 将流量传入集群

15.1. 将流量传入集群

OpenShift Container Platform 提供了多种方法从集群外部与集群中运行的服务进行通信。

注意

这部分中的流程需要由集群管理员执行先决条件。

管理员可以通过从一系列外部 IP 地址向该服务分配唯一外部 IP 地址来公开服务端点，供外部流量访问。管理员可以使用 CIDR 表示法来指定一系列地址，它允许应用程序用户为外部 IP 地址发出请求。

每个 IP 地址应只分配给一个服务，以确保每个服务都有唯一的端点。潜在的端口冲突是以先为先得原则处理的潜在端口。

建议（以首选程度进行排序）：

- 如果您有 HTTP/HTTPS，请使用 路由器。
- 如果您有 HTTPS 之外的 TLS 加密协议（例如，使用 SNI 标头的 TLS），请使用 路由器。
- 否则，请使用 负载均衡器、外部 IP 或 NodePort。

方法	用途
使用路由器	允许访问 HTTP/HTTPS 流量和 HTTPS 以外的 TLS 加密协议（例如，使用 SNI 标头的 TLS）。
使用 Load Balancer 服务自动分配公共 IP	允许流量通过从池分配的 IP 地址传到非标准端口。
手动将外部 IP 分配给服务	允许流量通过特定的 IP 地址传到非标准端口。
配置 NodePort	在集群中的所有节点上公开某一服务。

15.2. 使用路由器向集群获取流量

15.2.1. 概述

使用路由器是 允许从外部访问 OpenShift Container Platform 集群 的最常见方法。

路由器 配置为接受外部请求并根据 配置的路由 进行代理。这仅限于 HTTP/HTTPS(SNI)/TLS(SNI)，它涵盖 Web 应用程序。

15.2.2. 管理员先决条件

在开始此步骤前，管理员必须：

- 设置集群联网环境的外部端口，使请求能够到达集群。例如，可以将 name 配置为指向集群中的特定节点或其他 IP 地址。DNS 通配符 功能可用于将某个名称的子集配置为集群中的 IP 地址。这允许用户在集群中设置路由，而无需进一步关注管理员。
- 确保每个节点上的本地防火墙允许请求访问 IP 地址。
- 配置 OpenShift Container Platform 集群，以使用允许适当的用户访问的身份提供程序。
- 确保至少有一个用户具有 **cluster-admin** 角色。要将此角色添加到用户，请运行以下命令：

```
$ oc adm policy add-cluster-role-to-user cluster-admin <username>
```

- 有一个 OpenShift Container Platform 集群，其至少有一个 master 和至少一个节点，并且集群外有一个对集群具有网络访问权限的系统。此流程假设外部系统与集群位于同一个子网。不同子网上外部系统所需要的额外联网不在本主题的讨论范围内。

15.2.2.1. 定义公共 IP 地址范围

允许访问服务的第一步是在主配置文件中定义外部 IP 地址范围：

1. 以具有集群 admin 角色的用户身份登录 OpenShift Container Platform。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 在 */etc/origin/master/master-config.yaml* 文件中配置 **externalIPNetworkCIDRs** 参数，如下所示：

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

例如：

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 重启 OpenShift Container Platform master 服务以应用更改。

```
# master-restart api
# master-restart controllers
```

小心

IP 地址池必须在集群中的一个或多个节点上终止。

15.2.3. 创建一个项目和服务

如果您要公开的项目和服务尚不存在，请首先创建项目，再创建服务。

如果项目和服务都已存在，请进入下一步：**公开服务以创建路由。**

1. 登录 OpenShift Container Platform。
2. 为您的服务创建一个新项目：

```
$ oc new-project <project_name>
```

例如：

```
$ oc new-project external-ip
```

3. 使用 **oc new-app** 命令来创建服务：
例如：

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/openshift3/mysql-55-rhel7
```

4. 运行以下命令，以查看新服务是否已创建：

```
$ oc get svc
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7    172.30.131.89   <none>        3306/TCP   13m
```

默认情况下，新服务没有外部 IP 地址。

15.2.4. 将服务公开给创建路由

您必须使用 **oc expose** 命令将服务公开为路由。

公开服务：

1. 登录 OpenShift Container Platform。
2. 登录您想公开的服务所在的项目。

```
$ oc project project1
```

3. 运行以下命令以公开路由：

```
$ oc expose service <service_name>
```

例如：

```
$ oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. 在 master 上，使用 cURL 等工具来确保您可以使用服务的集群 IP 地址访问该服务：

```
$ curl <pod_ip>:<port>
```

例如：

```
$ curl 172.30.131.89:3306
```

此部分中的示例使用 MySQL 服务，这需要客户端应用程序。如果您得到一串字符并看到 **Got packets out of order** 消息，则您已连接到该服务。

如果您有 MySQL 客户端，请使用标准 CLI 命令登录：

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

15.2.5. 配置路由器

与管理员合作将路由器配置为接受外部请求并根据配置的路由进行代理。

管理员可以创建 通配符 DNS 条目，然后设置路由器。然后，您可以自助服务边缘路由器，无需与管理员联系。

路由器具有控制权，管理员可以指定用户是否可以自助置备主机名，或者主机名是否需要特定的模式。

在各种项目中创建一组路由时，整组路由都可用于一组路由器。每个路由器接受（或选择）来自于这一组路由的路由。默认情况下，所有路由器接受所有路由。

具有查看所有项目中所有标签的路由器，可以根据标签选择要接受的路由。这称为 路由器分片。这在在一组路由器之间平衡传入的流量负载时非常有用，当将流量隔离到特定路由器时。例如，A 公司使用一个路由器，将公司 B 转到另外一个路由器。

由于路由器在特定节点上运行，因此当或者节点无法停止流量时。通过在不同节点上创建冗余路由器并使用 高可用性 在节点出现故障时切换路由器 IP 地址，可以降低此问题的影响。

15.2.6. 使用 VIP 配置 IP 故障切换

另外，管理员可以配置 IP 故障转移。

IP 故障转移（IP failover）在一组节点上管理一个虚拟 IP（VIP）地址池。集合中的每个 VIP 都由从集合中选择的节点提供服务。只要单个节点可用，就会提供 VIP。无法将 VIP 显式分发到节点上。因此，可能存在没有 VIP 的节点，其他节点也有多个 VIP。如果只有一个节点，则所有 VIP 将位于其中。

VIP 必须可以从集群外部路由。

15.3. 使用负载均衡起来处理进入集群的网络数据

15.3.1. 概述

如果不需要具体的外部 IP 地址，您可以配置负载均衡器服务，以便从外部访问 OpenShift Container Platform 集群。

负载均衡器服务从配置的池分配唯一 IP。负载平衡器具有单一边缘路由器 IP（可以是 虚拟 IP(VIP) ），但仍然是一台用于初始负载均衡的计算机。

这个过程涉及以下内容：

- 管理员执行先决条件;
- 如果要公开的服务不存在， 开发人员会创建项目和服务 ；
- 开发人员公开服务以创建路由。
- 开发人员创建负载平衡器服务。
- 网络管理员将网络配置为服务。

15.3.2. 管理员先决条件

在开始此步骤前，管理员必须：

- 设置集群联网环境的外部端口，使请求能够到达集群。例如，可以将 name 配置为指向集群中的特定节点或其他 IP 地址。DNS 通配符 功能可用于将某个名称的子集配置为集群中的 IP 地址。这允许用户在集群中设置路由，而无需进一步关注管理员。
- 确保每个节点上的本地防火墙允许请求访问 IP 地址。
- 配置 OpenShift Container Platform 集群 ， 以使用允许适当的用户访问的身份提供程序。
- 确保至少有一个用户具有 **cluster-admin** 角色。要将此角色添加到用户，请运行以下命令：

```
$ oc adm policy add-cluster-role-to-user cluster-admin <username>
```

- 有一个 OpenShift Container Platform 集群，其至少有一个 master 和至少一个节点，并且集群外有一个对集群具有网络访问权限的系统。此流程假设外部系统与集群位于同一个子网。不同子网上外部系统所需要的额外联网不在本主题的讨论范围内。

15.3.2.1. 定义公共 IP 地址范围

允许访问服务的第一步是在主配置文件中定义外部 IP 地址范围：

1. 以具有集群 admin 角色的用户身份登录 OpenShift Container Platform。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 在 */etc/origin/master/master-config.yaml* 文件中配置 **externalIPNetworkCIDRs** 参数，如下所示：

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

例如：

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 重启 OpenShift Container Platform master 服务以应用更改。

```
# master-restart api
# master-restart controllers
```

小心

IP 地址池必须在集群中的一个或多个节点上终止。

15.3.3. 创建一个项目和服务

如果您要公开的项目和服务尚不存在，请首先创建项目，再创建服务。

如果项目和服务都已存在，请进入下一步：**公开服务以创建路由。**

1. 登录 OpenShift Container Platform。
2. 为您的服务创建一个新项目：

```
$ oc new-project <project_name>
```

例如：

```
$ oc new-project external-ip
```

3. 使用 **oc new-app** 命令来创建服务：
例如：

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/openshift3/mysql-55-rhel7
```

4. 运行以下命令，以查看新服务是否已创建：

```
$ oc get svc
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7    172.30.131.89   <none>        3306/TCP   13m
```

默认情况下，新服务没有外部 IP 地址。

15.3.4. 将服务公开给创建路由

您必须使用 **oc expose** 命令将服务公开为路由。

公开服务：

1. 登录 OpenShift Container Platform。
2. 登录您想公开的服务所在的项目。

```
$ oc project project1
```

3. 运行以下命令以公开路由：

```
$ oc expose service <service_name>
```

例如：

```
$ oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. 在 master 上，使用 cURL 等工具来确保您可以使用服务的集群 IP 地址访问该服务：

```
$ curl <pod_ip>:<port>
```

例如：

```
$ curl 172.30.131.89:3306
```

此部分中的示例使用 MySQL 服务，这需要客户端应用程序。如果您得到一串字符并看到 **Got packets out of order** 消息，则您已连接到该服务。

如果您有 MySQL 客户端，请使用标准 CLI 命令登录：

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

然后，执行以下任务：

- 创建 Load Balancer 服务
- 配置网络
- 配置 IP 故障切换

15.3.5. 创建 Load Balancer 服务

创建负载均衡器服务：

1. 登录 OpenShift Container Platform。
2. 加载您要公开的服务所在的项目。如果项目或服务不存在，请参阅创建项目和服务。

```
$ oc project project1
```

3. 在 master 节点上打开一个文本文件并粘贴以下文本，根据需要编辑该文件：

负载均衡器配置文件示例

```
apiVersion: v1
kind: Service
metadata:
  name: egress-2 1
spec:
  ports:
  - name: db
    port: 3306 2
  loadBalancerIP:
  type: LoadBalancer 3
  selector:
    name: mysql 4
```

1. 为负载均衡器服务输入一个描述性名称。
2. 输入您要公开的服务所侦听的同一个端口
3. 输入 **loadbalancer** 作为类型。
4. 输入服务的名称。

4. 保存并退出文件。
5. 运行以下命令来创建服务：

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

例如：

```
$ oc create -f mysql-lb.yaml
```

6. 执行以下命令以查看新服务：

```
$ oc get svc
NAME        CLUSTER-IP      EXTERNAL-IP                PORT(S)          AGE
egress-2    172.30.236.167  172.29.121.74,172.29.121.74  3306:30036/TCP   6s
```

请注意，该服务会自动分配一个外部 IP 地址。

7. 在 master 上，使用 cURL 等工具来确保您可以通过公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 172.29.121.74:3306
```

此部分中的示例使用 MySQL 服务，这需要客户端应用程序。如果您得到一串字符并看到 **Got packets out of order** 消息，则您已连接到该服务。

如果您有 MySQL 客户端，请使用标准 CLI 命令登录：

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

15.3.6. 配置网络

以下步骤是配置从其他节点访问公开的服务所需的网络的一般准则。随着网络环境的不同，请咨询您的网络管理员获取环境中需要进行的特定配置。

这些步骤假定所有系统都在同一个子网中。

在节点上：

1. 重新启动网络，以确保网络已启动。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

如果网络没有启动，在执行以下命令时，您将收到错误消息，如 **Network is unreachable**。

2. 在 master 上公开的服务的 IP 地址和 master 主机的 IP 地址之间添加路由。如果为网络路由使用子网掩码，请使用 **子网掩码** 选项以及要使用的子网掩码：

```
$ route add -net 172.29.121.74 netmask 255.255.0.0 gw 10.16.41.22 dev eth0
```

3. 使用 cURL 等工具来确保您可以使用公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 172.29.121.74:3306
```

如果您得到一串字符并看到 **Got packets out of order** 消息，则可从节点访问您的服务。

在没有位于集群中的系统中：

1. 重新启动网络，以确保网络已启动。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

如果网络没有启动，在执行以下命令时，您将收到错误消息，如 **Network is unreachable**。

2. 在 master 上公开的服务的 IP 地址和 master 主机的 IP 地址之间添加路由。如果为网络路由使用子网掩码，请使用 **子网掩码** 选项以及要使用的子网掩码：

```
$ route add -net 172.29.121.74 netmask 255.255.0.0 gw 10.16.41.22 dev eth0
```

3. 确保您可以使用公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 172.29.121.74:3306
```

如果您得到一串字符并看到 **Got packets out of order** 消息，则可以在集群外部访问您的服务。

15.3.7. 使用 VIP 配置 IP 故障切换

另外，管理员可以配置 IP 故障转移。

IP 故障转移（IP failover）在一组节点上管理一个虚拟 IP（VIP）地址池。集合中的每个 VIP 都由从集合中选择的节点提供服务。只要单个节点可用，就会提供 VIP。无法将 VIP 显式分发到节点上。因此，可能存在没有 VIP 的节点，其他节点也有多个 VIP。如果只有一个节点，则所有 VIP 将位于其中。

VIP 必须可以从集群外部路由。

15.4. 使用服务外部 IP 将流量传入集群

15.4.1. 概述

若要公开服务，一种方法是直接向您要从集群外部访问的服务分配外部 IP 访问权限。

请确定您创建了要使用的 IP 地址范围范围，如 定义公共 IP 地址范围 所示。

通过设置服务上的外部 IP，OpenShift Container Platform 会设置 IP 表规则，允许到达任何针对该 IP 地址发送到内部 Pod 的集群节点的流量。这与内部服务 IP 地址类似，但外部 IP 会告知 OpenShift Container Platform 此服务也应通过给定的 IP 对外部公开。管理员必须将该 IP 地址分配给集群中某一节点上的主机（节点）接口。另外，地址也可以用作 虚拟 IP(VIP)。

这些 IP 地址不由 OpenShift Container Platform 管理，管理员负责确保流量能通过此 IP 到达节点。

注意

以下是一个非 HA 解决方案，且不会配置 IP 故障转移。只用配置了 IP 故障转移才能使服务具有高可用性。

这个过程涉及以下内容：

- 管理员执行先决条件;
- 如果要公开的服务不存在，开发人员会创建项目和服务 ；

- 开发人员公开服务以创建路由。
- 开发人员为服务 分配 IP 地址。
- 网络管理员将网络配置为服务。

15.4.2. 管理员先决条件

在开始此步骤前，管理员必须：

- 设置集群联网环境的外部端口，使请求能够到达集群。例如，可以将 name 配置为指向集群中的特定节点或其他 IP 地址。DNS 通配符 功能可用于将某个名称的子集配置为集群中的 IP 地址。这允许用户在集群中设置路由，而无需进一步关注管理员。
- 确保每个节点上的本地防火墙允许请求访问 IP 地址。
- 配置 OpenShift Container Platform 集群，以使用允许适当的用户访问的身份提供程序。
- 确保至少有一个用户具有 **cluster-admin** 角色。要将此角色添加到用户，请运行以下命令：

```
$ oc adm policy add-cluster-role-to-user cluster-admin <username>
```

- 有一个 OpenShift Container Platform 集群，其至少有一个 master 和至少一个节点，并且集群外有一个对集群具有网络访问权限的系统。此流程假设外部系统与集群位于同一个子网。不同子网上外部系统所需要的额外联网不在本主题的讨论范围内。

15.4.2.1. 定义公共 IP 地址范围

允许访问服务的第一步是在主配置文件中定义外部 IP 地址范围：

1. 以具有集群 admin 角色的用户身份登录 OpenShift Container Platform。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 在 */etc/origin/master/master-config.yaml* 文件中配置 **externalIPNetworkCIDRs** 参数，如下所示：

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

例如：

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 重启 OpenShift Container Platform master 服务以应用更改。

```
# master-restart api
# master-restart controllers
```

小心

IP 地址池必须在集群中的一个或多个节点上终止。

15.4.3. 创建一个项目和服务

如果您要公开的项目和服务尚不存在，请首先创建项目，再创建服务。

如果项目和服务都已存在，请进入下一步：**公开服务以创建路由。**

1. 登录 OpenShift Container Platform。
2. 为您的服务创建一个新项目：

```
$ oc new-project <project_name>
```

例如：

```
$ oc new-project external-ip
```

3. 使用 **oc new-app** 命令来创建服务：
例如：

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.redhat.io/openshift3/mysql-55-rhel7
```

4. 运行以下命令，以查看新服务是否已创建：

```
$ oc get svc
NAME            CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7    172.30.131.89   <none>        3306/TCP   13m
```

默认情况下，新服务没有外部 IP 地址。

15.4.4. 将服务公开给创建路由

您必须使用 **oc expose** 命令将服务公开为路由。

公开服务：

1. 登录 OpenShift Container Platform。
2. 登录您想公开的服务所在的项目。

```
$ oc project project1
```

3. 运行以下命令以公开路由：

```
$ oc expose service <service_name>
```

例如：

```
$ oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. 在 master 上，使用 cURL 等工具来确保您可以使用服务的集群 IP 地址访问该服务：

```
$ curl <pod_ip>:<port>
```

例如：

```
$ curl 172.30.131.89:3306
```

此部分中的示例使用 MySQL 服务，这需要客户端应用程序。如果您得到一串字符并看到 **Got packets out of order** 消息，则您已连接到该服务。

如果您有 MySQL 客户端，请使用标准 CLI 命令登录：

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

然后，执行以下任务：

- 为服务分配一个 IP 地址
- 配置网络
- 配置 IP 故障切换

15.4.5. 为服务分配 IP 地址

为服务分配外部 IP 地址：

1. 登录 OpenShift Container Platform。
2. 加载您要公开的服务所在的项目。如果项目或服务不存在，请参阅先决条件中的创建项目和服务部分。
3. 运行以下命令，将外部 IP 地址分配给您要访问的服务。使用来自外部 IP 地址范围的 IP 地址 ：

```
$ oc patch svc <name> -p '{"spec":{"externalIPs":["<ip_address>"]}}'
```

<name> 是服务的名称，**-p** 表示要应用到服务 JSON 文件的补丁。括号中的表达式将指定 IP 地址分配给指定的服务。

例如：

```
$ oc patch svc mysql-55-rhel7 -p '{"spec":{"externalIPs":["192.174.120.10"]}}'

"mysql-55-rhel7" patched
```

4. 运行以下命令，以查看该服务具有公共 IP：

```
$ oc get svc
NAME            CLUSTER-IP      EXTERNAL-IP     PORT(S)    AGE
mysql-55-rhel7     172.30.131.89   192.174.120.10  3306/TCP   13m
```

5. 在 master 上，使用 cURL 等工具来确保您可以通过公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 192.168.120.10:3306
```

如果您得到一串字符并看到 **Got packets out of order** 消息，则您已连接到该服务。

如果您有 MySQL 客户端，请使用标准 CLI 命令登录：

```
$ mysql -h 192.168.120.10 -u admin -p
Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.

MySQL [(none)]>
```

15.4.6. 配置网络

分配外部 IP 地址后，您需要创建指向该 IP 的路由。

以下步骤是配置从其他节点访问公开的服务所需的网络的一般准则。随着网络环境的不同，请咨询您的网络管理员获取环境中需要进行的特定配置。

注意

这些步骤假定所有系统都在同一个子网中。

在 master 上：

1. 重新启动网络，以确保网络已启动。

```
# service network restart
Restarting network (via systemctl):  [  OK  ]
```

如果网络没有启动，在运行以下命令时，您将收到错误消息（如 **Network is unreachable**）。

2. 运行以下命令，使用您要公开的服务的外部 IP 地址以及与 **ifconfig** 命令输出中的主机 IP 关联的设备名称：

```
$ ip address add <external_ip> dev <device>
```

例如：

```
$ ip address add 192.168.120.10 dev eth0
```

如果您需要，运行以下命令来获取 master 所在的主机服务器的 IP 地址：

```
$ ifconfig
```

查找列出的设备类似如下：**UP,BROADCAST,RUNNING,MULTICAST**.

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.16.41.22  netmask 255.255.248.0  broadcast 10.16.47.255
      ...
```

3. 在 master 所在的主机的 IP 地址和 master 主机的网关 IP 地址之间添加路由。如果为网络路由使用子网掩码，请使用 **子网掩码** 选项以及要使用的子网掩码：

```
$ route add -host <host_ip_address> netmask <netmask> gw <gateway_ip_address> dev
<device>
```

例如：

```
$ route add -host 10.16.41.22 netmask 255.255.248.0 gw 10.16.41.254 dev eth0
```

netstat -nr 命令提供网关 IP 地址：

```
$ netstat -nr
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         10.16.41.254    0.0.0.0         UG        0 0          0 eth0
```

4. 在公开的服务的 IP 地址和 master 主机的 IP 地址间添加路由：

```
$ route add -net 192.174.120.0/24 gw 10.16.41.22 eth0
```

在节点上：

1. 重新启动网络，以确保网络已启动。

```
# service network restart
Restarting network (via systemctl):  [  OK  ]
```

如果网络没有启动，在执行以下命令时，您将收到错误消息，如 **Network is unreachable**。

2. 在节点所在主机的 IP 地址和节点主机的网关 IP 地址之间添加路由。如果为网络路由使用子网掩码，请使用 **子网掩码** 选项以及要使用的子网掩码：

```
$ route add -net 10.16.40.0 netmask 255.255.248.0 gw 10.16.47.254 eth0
```

ifconfig 命令显示主机 IP:

–

```
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.16.41.71  netmask 255.255.248.0  broadcast 10.19.41.255
```

netstat -nr 命令显示网关 IP：

```
$ netstat -nr
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         10.16.41.254    0.0.0.0         UG        0 0          0 eth0
```

3. 在公开的服务的 IP 地址和 master 节点所在主机系统的 IP 地址间添加路由：

```
$ route add -net 192.174.120.0 netmask 255.255.255.0 gw 10.16.41.22 dev eth0
```

4. 使用 cURL 等工具来确保您可以使用公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 192.168.120.10:3306
```

如果您得到一串字符并看到 **Got packets out of order** 消息，则可从节点访问您的服务。

在没有位于集群中的系统中：

1. 重新启动网络，以确保网络已启动。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

如果网络没有启动，在执行以下命令时，您将收到错误消息，如 **Network is unreachable**。

2. 在远程主机的 IP 地址和远程主机的网关 IP 之间添加路由。如果为网络路由使用子网掩码，请使用 **子网掩码** 选项以及要使用的子网掩码：

```
$ route add -net 10.16.64.0 netmask 255.255.248.0 gw 10.16.71.254 eno1
```

3. 在 master 上公开的服务的 IP 地址和 master 主机的 IP 地址间添加路由：

```
$ route add -net 192.174.120.0 netmask 255.255.255.0 gw 10.16.41.22
```

4. 使用 cURL 等工具来确保您可以使用公共 IP 地址访问该服务：

```
$ curl <public_ip>:<port>
```

例如：

```
$ curl 192.168.120.10:3306
```

如果您得到一串字符并看到 **Got packets out of order** 消息，则可以在集群外部访问您的服务。

15.4.7. 使用 VIP 配置 IP 故障切换

另外，管理员可以配置 IP 故障转移。

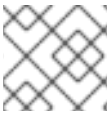
IP 故障转移（IP failover）在一组节点上管理一个虚拟 IP（VIP）地址池。集合中的每个 VIP 都由从集合中选择的节点提供服务。只要单个节点可用，就会提供 VIP。无法将 VIP 显式分发到节点上。因此，可能存在没有 VIP 的节点，其他节点也有多个 VIP。如果只有一个节点，则所有 VIP 将位于其中。

VIP 必须可以从集群外部路由。

15.5. 使用 NODEPORT 将流量获取到集群

15.5.1. 概述

使用 NodePorts 在集群中的所有节点上 公开服务 nodePort。



注意

使用 NodePort 需要额外的端口资源。

节点端口在节点 IP 地址的静态端口上公开该服务。

默认情况下，NodePort 在 30000-32767 范围内，这意味着 NodePort 不太可能与服务的预期端口匹配（例如，8080 可能会公开为 31020）。

管理员必须确保外部 IP 路由到所有节点上的节点，以及允许访问开放端口的本地防火墙规则。

NodePort 和外部 IP 互相独立，可以同时使用它们。

15.5.2. 管理员先决条件

在开始此步骤前，管理员必须：

- 设置集群联网环境的外部端口，使请求能够到达集群。例如，可以将 name 配置为指向集群中的特定节点或其他 IP 地址。DNS 通配符 功能可用于将某个名称的子集配置为集群中的 IP 地址。这允许用户在集群中设置路由，而无需进一步关注管理员。
- 确保每个节点上的本地防火墙允许请求访问 IP 地址。
- 配置 OpenShift Container Platform 集群，以使用允许适当的用户访问的身份提供程序。
- 确保至少有一个用户具有 **cluster-admin** 角色。要将此角色添加到用户，请运行以下命令：

```
$ oc adm policy add-cluster-role-to-user cluster-admin <username>
```

- 有一个 OpenShift Container Platform 集群，其至少有一个 master 和至少一个节点，并且集群外有一个对集群具有网络访问权限的系统。此流程假设外部系统与集群位于同一个子网。不同子网上外部系统所需要的额外联网不在本主题的讨论范围内。

15.5.3. 配置服务

在创建或修改服务时，您可以为 nodePort 指定端口号。如果您没有手动指定端口，系统将为您分配一个端口。

1. 登录 master 节点。
2. 如果您要使用的项目不存在，请为您的服务创建一个新项目：

```
$ oc new-project <project_name>
```

例如：

```
$ oc new-project external-ip
```

3. 编辑服务定义，以指定 **spec.type:NodePort**，并选择性地在 30000-32767 范围中指定端口。

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  type: NodePort
  ports:
    - port: 3306
      nodePort: 30036
      name: http
  selector:
    name: mysql
```

4. 运行以下命令来创建服务：

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

例如：

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

5. 运行以下命令，以查看新服务是否已创建：

```
$ oc get svc

NAME            CLUSTER_IP      EXTERNAL_IP   PORT(S)                 AGE
mysql           172.30.89.219   <none>        3306:30036/TCP          2m
```

请注意，外部 IP 列为 **<none>** 和节点端口。

您应能够使用 **<NodeIP>:<NodePort>** 地址访问该服务。

第 16 章 ROUTES

16.1. 概述

OpenShift Container Platform 路由以主机名（如 *www.example.com* ）公开服务，以便外部客户端能够通过名称访问该服务。

主机名的 DNS 解析是独立于路由处理的。您的管理员可能配置了一个云域，它总是可以解析到 OpenShift Container Platform 路由器，或者使用不相关的主机名单独修改其 DNS 记录来解析到路由器。

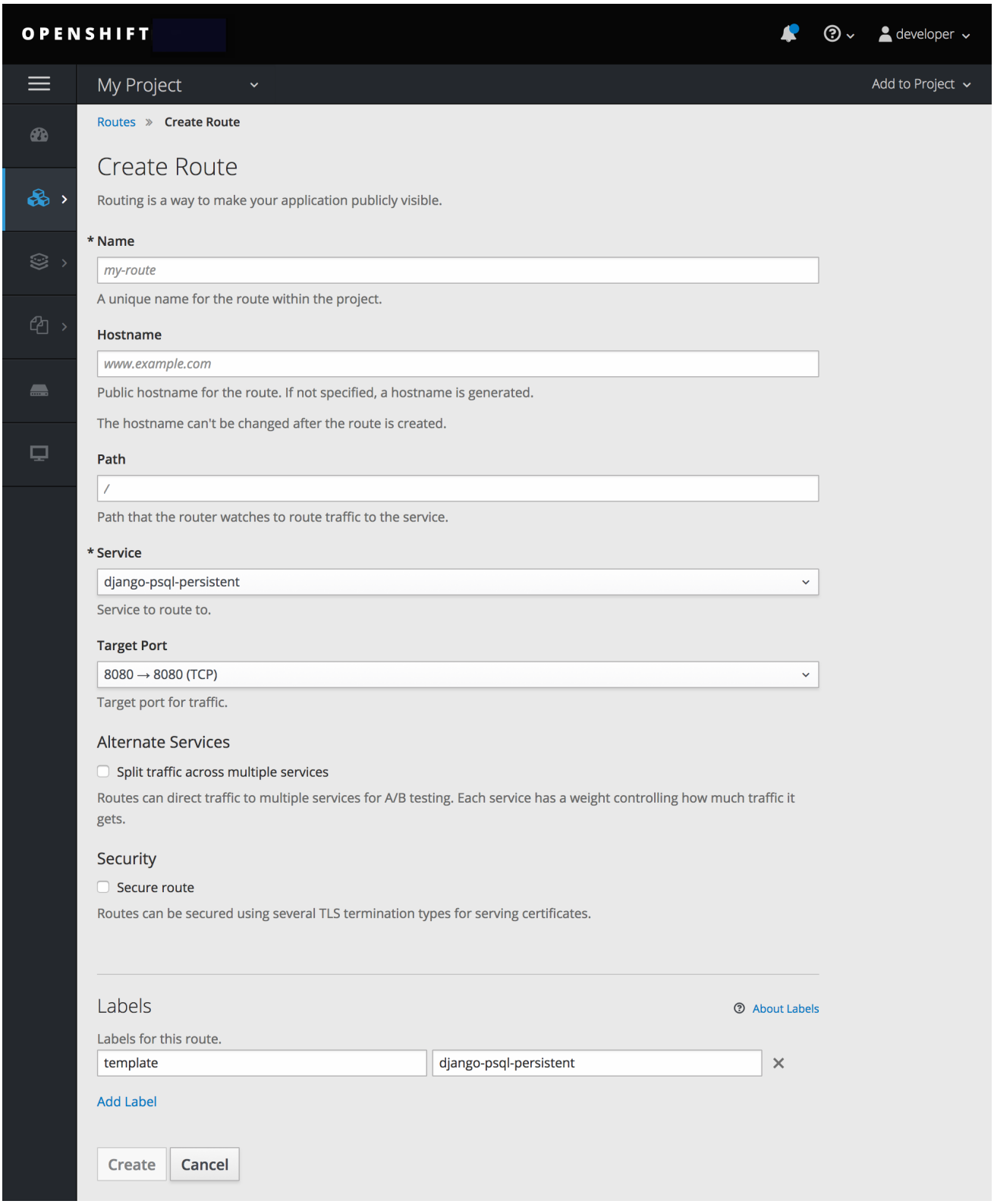
16.2. 创建路由

您可以使用 Web 控制台或 CLI 创建不受保护和安全的路由。

使用 Web 控制台，您可以进入 **Routes** 页面，在导航的 **Applications** 部分找到。

点击 **Create Route** 在项目中创建路由：

图 16.1. 使用 Web 控制台创建路由



使用 CLI 时，以下示例会创建一个不受保护的路由：

```
$ oc expose svc/frontend --hostname=www.example.com
```

新路由从服务继承名称，除非您使用 **--name** 选项指定名称。

上面创建的非安全路由的 YAML 定义

```
apiVersion: v1
```

```
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  path: "/test" 1
  to:
    kind: Service
    name: frontend
```

1 对于 基于路径的路由，请指定与 URL 相比的路径组件。

有关使用 CLI 配置路由的详情，请参考 Route Types。

非安全的路由是默认配置，因此最容易设置。但是，安全路由 为连接保持私有提供安全性。要创建使用密钥和证书加密的安全 HTTPS 路由（PEM-format 文件必须单独生成和签名），您可以使用 **create route** 命令并选择性地提供证书和密钥。

注意

TLS 是 HTTPS 和其他加密协议的 SSL 替代。

```
$ oc create route edge --service=frontend \
    --cert=${MASTER_CONFIG_DIR}/ca.crt \
    --key=${MASTER_CONFIG_DIR}/ca.key \
    --ca-cert=${MASTER_CONFIG_DIR}/ca.crt \
    --hostname=www.example.com
```

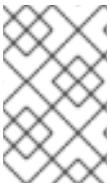
上面创建的安全路由的 YAML 定义

```
apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

目前，不支持密码保护的密钥文件。启动后，HAProxy 会提示输入密码，且无法自动执行此过程。要从密钥文件中删除密码短语，您可以运行以下命令：

```
# openssl rsa -in <passwordProtectedKey.key> -out <new.key>
```

您可以在不指定密钥和证书的前提下创建安全路由，在这种情况下，路由器的默认证书 将用于 TLS 终止。



注意

OpenShift Container Platform 中的 TLS 终止依赖于 SNI 来提供自定义证书。任何在端口 443 上收到的非SNI 流量都使用 TLS 终止处理，默认证书可能与请求的主机名不匹配，从而导致验证错误。

如需有关所有类型的 TLS 终止 以及 基于路径的路由 的更多信息，请参阅 架构部分。

16.3. 允许路由端点控制 COOKIE 名称

OpenShift Container Platform 提供粘性会话，通过确保所有流量都到达同一端点来实现有状态应用程序流量。但是，如果端点 pod 以重启、扩展或更改配置的方式被终止，这种有状态性可能会消失。

OpenShift Container Platform 可以使用 Cookie 来配置会话持久性。路由器选择一个端点来处理任何用户请求，并为会话创建一个 Cookie。Cookie 在响应请求时返回，用户则通过会话中的下一请求发回 Cookie。Cookie 告知路由器正在处理会话，确保客户端请求使用这个 Cookie 使请求路由到同一个 pod。

您可以设置 Cookie 名称来覆盖为路由自动生成的默认名称。通过删除 Cookie，它可以强制下一请求重新选择端点。因此，如果服务器过载，它会尝试从客户端中删除请求并重新分发它们。

1. 使用所需 Cookie 名称标注路由：

```
$ oc annotate route <route_name> router.openshift.io/cookie_name="<your_cookie_name>"
```

例如，将 **my_cookie** 指定为您的新 cookie 名称：

```
$ oc annotate route my_route router.openshift.io/cookie_name="my_cookie"
```

2. 保存 Cookie，再访问路由：

```
$ curl $my_route -k -c /tmp/my_cookie
```

第 17 章 集成外部服务

17.1. 概述

许多 OpenShift Container Platform 应用程序使用外部资源，如外部数据库或外部 SaaS 端点。这些外部资源可以建模为原生 OpenShift Container Platform 服务，以便应用程序可以与任何其他内部服务一起工作。

出口流量 可以由防火墙规则或 Egress 路由器控制。这允许其应用程序服务具有静态 IP 地址。

17.2. 为外部数据库定义服务

最常见的外部服务类型是外部数据库。要支持外部数据库，应用程序需要：

1. 一个要与之通信的端点。
2. 一组凭证和协调，包括：
   - 用户名
   - 密码短语
   - 数据库名称

与外部数据库集成的解决方案包括：

- 一个 **Service** 对象，它将 SaaS 供应商表示为 OpenShift Container Platform 服务。
- 服务的一个或多个**端点**。
- 含有凭据的适当 pod 中的环境变量。

下列步骤概述了与外部 MySQL 数据库集成的情况：

17.2.1. 第 1 步：定义服务

您可以通过提供 IP 地址和端点来定义服务，或者提供完全限定域名(FQDN)。

17.2.1.1. 使用 IP 地址

1. 创建一个 OpenShift Container Platform 服务来 代表您的外部数据库。这与创建内部服务类似，差异在服务的 **Selector** 字段中。
   内部 OpenShift Container Platform 服务使用 **Selector** 字段，通过 标签 将 pod 与服务关联。**EndpointsController** 系统组件将指定选择器的服务端点与与选择器匹配的 pod 同步。服务代理和 OpenShift Container Platform 路由器在服务端点之间对服务进行负载均衡。

   代表外部资源的服务不需要关联的 pod。相反，请保留 **Selector** 字段未设置。这代表了外部服务，使 **EndpointsController** 忽略该服务并允许手动指定端点：

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
```

```
  ports:
    -
      name: "mysql"
      protocol: "TCP"
      port: 3306
      targetPort: 3306 (1)
      nodePort: 0
selector: {} (2)
```

1. 可选：服务将连接转发到的后备 pod 上的端口。
2. 将 **selector** 字段留为空。

2. 接下来，为该服务创建所需的端点。这为服务代理和路由器提供发送到服务的流量的位置：

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "external-mysql-service" (1)
subsets: (2)
  -
    addresses:
      -
        ip: "10.0.0.0" (3)
    ports:
      -
        port: 3306 (4)
        name: "mysql"
```

1. **Service** 实例的名称，如上一步中所定义。
2. 如果提供了多个端点，则在提供的**端点**间对服务的流量进行负载平衡。
3. 端点 IP 不能是 loopback (127.0.0.0/8), link-local (169.254.0.0/16), 或 link-local multicast (224.0.0.0/24).
4. **port** 和 **name** 定义必须与上一步中定义的服务中的 **port** 和 **name** 的值相匹配。

17.2.1.2. 使用外部域名

使用外部域名可以更轻松地管理外部服务链接，因为您不必担心外部服务的 IP 地址更改。

ExternalName 服务没有选择器，或者任何定义的端口或端点，您可以使用 **ExternalName** 服务将流量定向到外部服务。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  type: ExternalName
  externalName: example.domain.name
selector: {} (1)
```

-

1 将 **selector** 字段留为空。

使用外部域名服务会通知系统在 **externalName** 字段中（上例中的**example.domain.name** ）中的 DNS 名称是恢复该服务的资源位置。针对 Kubernetes DNS 服务器发出 DNS 请求时，它会在 CNAME 记录中返回 **externalName**，告知客户端查找返回的名称以获取 IP 地址。

17.2.2. 第 2 步：使用服务

现在，定义了服务和端点，请通过在适当的容器中设置环境变量，授予适当的 pod 访问凭证以使用该服务：

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: (1)
  strategy:
    type: "Rolling"
    rollingParams:
      updatePeriodSeconds: 1 (2)
      intervalSeconds: 1 (3)
      timeoutSeconds: 120
  replicas: 2
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
          name: "helloworld"
          image: "origin-ruby-sample"
          ports:
            -
              containerPort: 3306
              protocol: "TCP"
          env:
            -
              name: "MYSQL_USER"
              value: "${MYSQL_USER}" (4)
            -
              name: "MYSQL_PASSWORD"
              value: "${MYSQL_PASSWORD}" (5)
            -
              name: "MYSQL_DATABASE"
              value: "${MYSQL_DATABASE}" (6)
```

1 **DeploymentConfig** 中的其他字段被省略

2 各个 pod 更新之间等待的时间。

3 更新后轮询部署状态之间等待的时间。

4 用于服务的用户名。

5 与服务一起使用的密码短语。

6 数据库名称。

外部数据库环境变量

在您的应用程序中使用外部服务与使用内部服务类似。您的应用程序将被分配该服务的环境变量，以及上一步中的凭据的其他环境变量。例如，MySQL 容器接收以下环境变量：

- **EXTERNAL_MYSQL_SERVICE_SERVICE_HOST=<ip_address>**
- **EXTERNAL_MYSQL_SERVICE_SERVICE_PORT=<port_number>**
- **MYSQL_USERNAME=<mysql_username>**
- **MYSQL_PASSWORD=<mysql_password>**
- **MYSQL_DATABASE_NAME=<mysql_database>**

应用程序负责从环境中读取服务的协调和凭证，并通过 服务与数据库建立连接。

17.3. 外部 SAAS 供应商

常见类型的外部服务是外部 SaaS 端点。要支持外部 SaaS 供应商，应用程序需要：

1. 要通信的端点
2. 一组凭证，例如：
   a. API 密钥
   b. 用户名
   c. 密码短语

下列步骤概述了与外部 SaaS 供应商集成的一个场景：

17.3.1. 使用 IP 地址和端点

1. 创建一个 OpenShift Container Platform 服务 来代表外部服务。这和创建内部服务类似，但差存在于服务的 **Selector** 字段中。
   内部 OpenShift Container Platform 服务使用 **Selector** 字段，通过 标签 将 pod 与服务关联。名为 **EndpointsController** 的系统组件同步了用于指定选择器和与选择器匹配的 pod 的服务的端点。服务代理和 OpenShift Container Platform 路由器在服务端点之间对服务进行负载均衡。

   代表外部资源的服务不需要与 pod 关联。相反，请保留 **Selector** 字段未设置。这会导致 **EndpointsController** 忽略该服务，并手动指定端点：

```
kind: "Service"
apiVersion: "v1"
metadata:
```

```
  name: "example-external-service"
spec:
  ports:
    -
      name: "mysql"
      protocol: "TCP"
      port: 3306
      targetPort: 3306 (1)
      nodePort: 0
selector: {} (2)
```

1. 可选：服务将连接转发到的后备 pod 上的端口。
2. 将 **selector** 字段留为空。

2. 然后，为该服务创建端点，其中包含有关发送流量发送到服务代理和路由器的信息：

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "example-external-service" (1)
subsets: (2)
- addresses:
  - ip: "10.10.1.1"
  ports:
  - name: "mysql"
    port: 3306
```

1. **Service** 实例的名称。
2. 到服务的流量在此处提供的**子集** 之间实现负载平衡。

3. 现在，定义了服务和端点，通过在适当的容器中设置环境变量来为 pod 提供凭证来使用该服务：

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: (1)
  strategy:
    type: "Rolling"
    rollingParams:
      timeoutSeconds: 120
  replicas: 1
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
```

```
      name: "helloworld"
      image: "openshift/openshift/origin-ruby-sample"
      ports:
        -
          containerPort: 3306
          protocol: "TCP"
      env:
        -
          name: "SAAS_API_KEY" 2
          value: "<SaaS service API key>"
        -
          name: "SAAS_USERNAME" 3
          value: "<SaaS service user>"
        -
          name: "SAAS_PASSPHRASE" 4
          value: "<SaaS service passphrase>"
```

1 **DeploymentConfig** 中的其他字段会被忽略。

2 **SAAS_API_KEY**:用于服务的 API 密钥。

3 **SAAS_USERNAME**:用于服务的用户名。

4 **SAAS_PASSPHRASE**:与服务一起使用的密码短语。

这些变量作为环境变量添加到容器中。使用环境变量允许服务对服务进行通信，并且可能需要其他参数，如 API 密钥、用户名和密码身份验证或证书。

外部 SaaS 供应商环境变量

同样，在使用内部服务时，会为您的服务分配环境变量，并使用前面步骤中描述的凭据提供额外的环境变量。在上例中，容器接收以下环境变量：

- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_HOST=<ip_address>**
- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_PORT=<port_number>**
- **SAAS_API_KEY=<saas_api_key>**
- **SAAS_USERNAME=<saas_username>**
- **SAAS_PASSPHRASE=<saas_passphrase>**

应用程序从环境中读取服务的协调和凭证，并使用该服务建立连接。

17.3.2. 使用外部域名

ExternalName 服务没有选择器，或者任何定义的端口或端点。您可以使用 **ExternalName** 服务将流量分配给集群外的外部服务。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
```

```
  type: ExternalName
  externalName: example.domain.name
selector: {} (1)
```

(1) 将 **selector** 字段留为空。

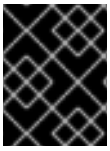
使用 **ExternalName** 服务将服务映射到 **externalName** 字段的值（上例中的**example.domain.name** ），方法是自动注入 CNAME 记录，将服务名称直接映射到外部 DNS 地址，并绕过对端点记录的需要。

第 18 章 使用设备管理器

18.1. 设备管理器的作用

设备管理器是一个 Kubelet 功能，它提供了一个使用 Kubelet 插件（称为 device plug-ins）公告专用节点硬件资源的机制。

任何供应商都可以实施设备插件来公告其专用硬件，而无需上游代码更改。



重要

OpenShift Container Platform 支持设备插件 API，但设备插件容器由各个供应商提供支持。

设备管理器将设备公告为**外部资源**。用户 pod 可以利用相同的**限制/请求**机制来使用设备管理器公告的设备，这一机制也用于请求任何其他**扩展资源**。

18.1.1. 注册

在启动时，设备插件通过在 */var/lib/kubelet/device-plugins/kubelet.sock* 上调用 **Register** 将自身注册到设备管理器，再启动位于 */var/lib/kubelet/device-plugins/<plugin>.sock* 的 gRPC 服务来服务设备管理器请求。

18.1.2. 设备发现和健康监控

在处理新的注册请求时，设备管理器会在设备插件服务中调用 **ListAndWatch** 远程过程调用 (RPC)。作为响应，设备管理器通过 gRPC 流从插件中获取设备对象的列表。设备管理器对流进行持续监控，以确认插件有没有新的更新。在插件一端，插件也会使流保持开放；只要任何设备的状态有所改变，就会通过相同的流传输连接将新设备列表发送到设备管理器。

18.1.3. 设备分配

在处理新的 pod 准入请求时，Kubelet 将请求的**扩展资源**传递给设备管理器以进行设备分配。设备管理器在其数据库中检查，以验证是否存在对应的插件。如果插件存在并且有可分配的设备及本地缓存，则在该特定设备插件上调用 **Allocate** RPC。

此外，设备插件也可以执行其他几个特定于设备的操作，如驱动程序安装、设备初始化和设备重置。这些功能视具体实现而异。

18.2. 启用设备管理器

启用设备管理器来实现设备插件，在不更改上游代码的前提下公告专用硬件。

1. 在目标节点或节点上启用设备管理器支持：

```
$ oc describe configmaps <name>
```

例如：

```
$ oc describe configmaps node-config-infra
...
kubeletArguments:
```

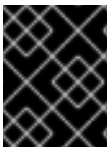
```
...
  feature-gates:
  - DevicePlugins=true
```

2. 通过确认节点上已创建了 */var/lib/kubelet/device-plugins/kubelet.sock*，确保已启用了设备管理器。这是设备管理器 gRPC 服务器在其上侦听新插件注册的 UNIX 域套接字。只有启用了设备管理器，才会在 Kubelet 启动时创建此 sock 文件。

第 19 章 使用设备插件

19.1. 设备插件的作用

借助设备插件，您无需编写自定义代码，就能在 OpenShift Container Platform pod 中使用特定的设备类型，如 GPU、InfiniBand 或其他需要供应商专用初始化和设置的类似计算资源。设备插件提供一致并可移植的解决方案，以便跨集群消耗硬件设备。设备插件通过一种扩展机制为这些设备提供支持，从而使这些设备可供容器使用，提供这些设备的健康检查，并安全地共享它们。



重要

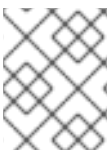
OpenShift Container Platform 支持设备插件 API，但设备插件容器由各个供应商提供支持。

设备插件是在节点(**atomic-openshift-node.service**)上运行的 gRPC 服务，负责管理特定的硬件资源。任何设备插件都必须支持以下远程过程调用 (RPC)：

```
service DevicePlugin {
    // GetDevicePluginOptions returns options to be communicated with Device
    // Manager
    rpc GetDevicePluginOptions(Empty) returns (DevicePluginOptions) {}

    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device disappears, ListAndWatch
    // returns the new list
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

    // Allocate is called during container creation so that the Device
    // Plug-in can run device specific operations and instruct Kubelet
    // of the steps to make the Device available in the container
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}

    // PreStartContainer is called, if indicated by Device Plug-in during
    // registration phase, before each container start. Device plug-in
    // can run device specific operations such as reseting the device
    // before making devices available to the container
    rpc PreStartContainer(PreStartContainerRequest) returns (PreStartContainerResponse) {}
}
```

19.1.1. 设备插件示例

- 适用于 COS 型操作系统的 Nvidia GPU 设备插件
- Nvidia 官方 GPU 设备插件
- KubeVirt 设备插件：vfio 和 kvm
- 适用于 GPU、FPGA 和 QuickAsist 设备的 Intel 设备插件



注意

对于简单设备插件参考实现，设备管理器代码中提供了一个存根设备插件：*vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go*。

19.2. 设备插件部署方法

- DaemonSet 是设备插件部署的推荐方法。
- 在启动时，设备插件会尝试在节点上 */var/lib/kubelet/device-plugin/* 创建一个 UNIX 域套接字，以便服务来自于 设备管理器 的 RPC。
- 由于设备插件必须管理硬件资源、主机文件系统的访问权以及套接字创建，它们必须在一个特权安全上下文中运行。
- 各种设备插件实现中提供了有关部署步骤的更多细节。

第 20 章 SECRETS

20.1. 使用 SECRET

本主题讨论 secret 的重要属性，并概述了开发人员如何使用它们。

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 的键名称和值 \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** 映射中的键关联的值必须采用 base64 编码。
4. 与 **stringData** 映射中的键关联的值由纯文本字符串组成。
5. **stringData** 映射中的条目将转换为 base64，然后该条目将自动移动到 **data** 映射中。此字段是只读的，只能通过 **data** 字段返回。

    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
    ```

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

YAML Opaque Secret 对象定义

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
```

```
data:
  username: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

1 指定一个 *opaque* secret。

Docker 配置 JSON 文件对象定义

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

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cgYXV0aC
BrZXIzCg== 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

20.1.1. 机密的属性

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。

20.1.2. 创建 Secret

您必须先创建 secret，然后创建依赖于此 secret 的 pod。

在创建 secret 时：

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

您可以使用 create 命令从 JSON 或 YAML 文件创建 secret 对象：

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

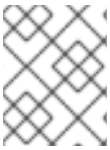
20.1.3. secret 的类型

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

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

- **kubernetes.io/service-account-token**。使用服务帐户令牌。
- **kubernetes.io/dockercfg**。将 *.dockercfg* 文件用于所需的 Docker 凭证。
- **kubernetes.io/dockerconfigjson**。将 *.docker/config.json* 文件用于所需的 Docker 凭证。
- **kubernetes.io/basic-auth**。与基本身份验证一起使用。
- **kubernetes.io/ssh-auth**。与 SSH 密钥身份验证 一起使用 .
- **kubernetes.io/tls**。使用 TLS 证书颁发机构

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**。这些类型不是在服务器端强制执行，而是表明 secret 的创建者意在符合该类型的键/值要求。

有关不同 secret 类型的示例，请参阅 *使用 Secrets* 中的代码示例。

20.1.4. 更新 secret

修改 secret 值时，值（由已在运行的 pod 使用）不会动态更改。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）。当您将 secret 挂载为卷时，您的 secret 会自动更新。

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。

20.2. 卷和环境变量中的 SECRET

如果容器使用 **secret** 作为环境变量，您必须重启容器来查看更新的 **secret**。请参阅包含 secret 数据的 YAML 文件示例。

创建 secret 后，您可以：

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
```

20.3. 镜像提取 SECRET

如需更多信息，请参阅使用镜像提取 Secret。

20.4. 源克隆 SECRET

如需有关在构建期间使用源克隆 secret 的更多信息，请参阅构建输入。

20.5. 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主控机生成的服务器证书相同。

表 20.1. 服务用（service serving）证书 secret

命名空间	Secret
default	**router-metrics-tls**
kube-service-catalog	**controllermanager-ssl**
openshift-ansible-service-broker	**asb-tls**
openshift-console	**console-serving-cert**
openshift-infra	**heapster-certs**
openshift-logging	**prometheus-tls**
openshift-monitoring	**alertmanager-main-tls**
	grafana-tls
	kube-state-metrics-tls
	node-exporter-tls
	prometheus-k8s-tls
openshift-template-service-broker	**apiserver-serving-cert**
openshift-web-console	**webconsole-serving-cert**

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。要做到这一点，将服务的 **service.alpha.openshift.io/serving-cert-secret-name** 注解设置为您要用于 secret 的名称。然后，您的 **PodSpec** 可以挂载该 secret。当它可用时，您的 Pod 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。当过期时间为一小时时，证书/密钥对会被自动替换。在 secret 的 **service.alpha.openshift.io/expiry** 注解中查看到期日期，其采用 RFC3339 格式。

其他 pod 可以信任集群创建的证书（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。

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。

20.6. 限制

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

- 容器的环境变量。
- 作为挂载到一个或多个容器上的卷中的文件。
- 在拉取 pod 的镜像时由 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 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，创建许多较小的 secret 也可能会耗尽内存。

20.6.1. Secret 数据密钥

Secret 密钥必须在 DNS 子域中。

20.7. 例子

例 20.1. 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。

例 20.2. 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
```

例 20.3. 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
```

例 20.4. 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
```

20.8. 故障排除

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
```

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 **service.alpha.openshift.io/serving-cert-generation-error**, **service.alpha.openshift.io/serving-cert-generation-error-num** 中的以下注解来强制重新生成证书：

```
$ 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，要移除的注解名称后面有一个 **-**。

第 21 章 CONFIGMAPS

21.1. 概述

许多应用程序需要使用配置文件、命令行参数和环境变量的某些组合来进行配置。这些配置工件应该与镜像内容分离，以便使容器化应用程序可以移植。

ConfigMap 对象提供向容器注入配置数据的机制，同时保持容器与 OpenShift Container Platform 无关。**ConfigMap** 可用于存储细粒度信息（如个别属性）或粒度信息（如完整配置文件或 JSON blob）。

ConfigMap API 对象包含配置数据的键值对，这些数据可在 Pod 中消耗或用于存储控制器等系统组件的配置数据。**ConfigMap** 与 secret 类似，但设计为能更加便捷地支持与不含敏感信息的字符串配合。

例如：

ConfigMap 对象定义

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData:
  bar: L3Jvb3QvMTAw 2
```

1 包含配置数据。

2 指向含有非 UTF8 数据的文件，如二进制 Java 密钥存储文件。在 Base 64 中输入文件路径。

注意

从文件 创建 configmap 时，您可以使用 **binaryData** 字段。

可以在 Pod 中以各种方式消耗配置数据。**ConfigMap** 可用于：

1. 填充环境变量的值。
2. 设置容器中的命令行参数。
3. 填充卷中的配置文件。

用户和系统组件都可以在 **ConfigMap** 中存储配置数据。

21.2. 创建 CONFIGMAP

您可以使用以下命令从目录、特定文件或文字值中轻松创建 **ConfigMap** ：

```
$ oc create configmap <configmap_name> [options]
```

以下部分涵盖了创建 **ConfigMap** 的不同方法。

21.2.1. 从目录创建

考虑包含一些已包含您要填充 **ConfigMap** 的数据的文件目录：

```
$ ls example-files
game.properties
ui.properties

$ cat example-files/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat example-files/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

您可以使用以下命令来创建包含此目录中每个文件内容的 **ConfigMap** ：

```
$ oc create configmap game-config \
    --from-file=example-files/
```

当 **--from-file** 选项指向某个目录时，该目录中的每个文件都直接用于在 **ConfigMap** 中填充密钥，其中键的名称是文件名称，键的值是文件的内容。

例如，上述命令会创建以下 **ConfigMap** ：

```
$ oc describe configmaps game-config
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data

game.properties:        121 bytes
ui.properties:          83 bytes
```

您可以在映射中看到从命令中指定的目录中的文件名称创建这两个键。由于这些密钥的内容可能较大，所以 **oc describe** 的输出只会显示键的名称及其大小。

如果要查看键的值，您可以使用 **-o** 选项运行 **oc get**：

```
$ oc get configmaps game-config -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"-
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

21.2.2. 从文件创建

您还可以使用特定文件传递 **--from-file** 选项，并多次将 --from-file 传递给 CLI。以下示例生成与 从目录创建示例相同的结果：

注意

如果从文件创建 configmap，您可以在不会破坏非 UTF8 数据的项中包含非 UTF8 数据的文件。OpenShift Container Platform 检测到二进制文件，并将该文件编码为 **MIME**。在服务器上，MIME 有效负载被解码并存储而不会损坏数据。

1. 创建 **ConfigMap** 指定特定文件：

```
$ oc create configmap game-config-2 \
    --from-file=example-files/game.properties \
    --from-file=example-files/ui.properties
```

2. 验证结果：

```
$ oc get configmaps game-config-2 -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
```

```
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

您还可以通过传递一个 **key=value** 表达式，将键设置为用于单独的文件（带有 **--from-file** 选项）。例如：

1. 创建 **ConfigMap** 指定键值对：

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

2. 验证结果：

```
$ oc get configmaps game-config-3 -o yaml

apiVersion: v1
data:
  game-special-key: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

21.2.3. 从 Literal 值创建

您还可以为 **ConfigMap** 提供字面值。**--from-literal** 选项使用 **key=value** 语法，允许直接在命令行中提供字面值：

1. 创建指定字面值的 **ConfigMap**:

```
$ oc create configmap special-config \
    --from-literal=special.how=very \
    --from-literal=special.type=charm
```

2. 验证结果：

```
$ oc get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

21.3. 使用案例：在 POD 中消耗 CONFIGMAP

以下小节描述了在 pod 中消耗 **ConfigMap** 对象时的一些用例。

21.3.1. 在环境变量中消耗

ConfigMap 可用于填充各个环境变量，或者从构成有效环境变量名称的所有键填充环境变量。例如，请考虑以下 **ConfigMap** ：

有两个环境变量的 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config (1)
  namespace: default
data:
  special.how: very (2)
  special.type: charm (3)
```

(1) ConfigMap 的名称。

(2) (3) 要注入的环境变量。

包含一个环境变量的 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config (1)
```

```
  namespace: default
data:
  log_level: INFO (2)
```

(1) ConfigMap 的名称。

(2) 要注入的环境变量。

您可以使用 **configMapKeyRef** 部分在 pod 中消耗此 **ConfigMap** 的键：

配置为注入特定环境变量的 pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: (1)
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config (2)
              key: special.how (3)
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config (4)
              key: special.type (5)
              optional: true (6)
      envFrom: (7)
        - configMapRef:
            name: env-config (8)
  restartPolicy: Never
```

(1) 从 **ConfigMap** 中拉取指定的环境变量的小节。

(2) (4) 要从中拉取特定环境变量的 **ConfigMap** 名称。

(3) (5) 要从 **ConfigMap** 中拉取的环境变量。

(6) 使环境变量成为可选。作为可选项，即使指定的 **ConfigMap** 和键不存在，也会启动 pod。

(7) 从 **ConfigMap** 中拉取所有环境变量的小节。

(8) 用于拉取所有环境变量的 **ConfigMap** 名称。

当此 pod 运行时，其输出将包含以下行：

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

21.3.2. 设置命令行参数

ConfigMap 还可用于设置容器中的命令或参数的值。这可以通过 Kubernetes 替换语法 **$(VAR_NAME)** 来完成。考虑以下 **ConfigMap** ：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

要将值注入命令行中，您需要以环境变量的形式使用您要使用的键，如环境变量用例所示。然后，您可以使用 **$(VAR_NAME)** 语法在容器的命令中引用它们。

配置为注入特定环境变量的 pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

当此 pod 运行时，**test-container** 容器的输出将是：

```
very charm
```

21.3.3. 在卷中消耗

ConfigMap 也可以被消耗在卷中。再次返回到以下示例 **ConfigMap** ：

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

在卷中消耗此 **ConfigMap** 时有两个不同的选项。最基本的方法是使用文件来填充卷，其中键为文件名，键值是文件的内容：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

当此 pod 运行时，输出将是：

```
very
```

您还可以控制投射 **ConfigMap** 键的卷中的路径：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never
```

当此 pod 运行时，输出将是：

```
very
```

21.4. 例如：配置 REDIS

对于真实示例，您可以使用 **ConfigMap** 配置 Redis。要将 Redis 注入用于将 Redis 用作缓存的推荐配置，Red Hat Redis 配置文件应包含以下内容：

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

如果您的配置文件位于 *example-files/redis/redis-config* 中，创建一个带有它的 **ConfigMap** ：

1. 创建指定配置文件的 **ConfigMap** ：

```
$ oc create configmap example-redis-config \
    --from-file=example-files/redis/redis-config
```

2. 验证结果：

```
$ oc get configmap example-redis-config -o yaml

apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-04-06T05:53:07Z
  name: example-redis-config
  namespace: default
  resourceVersion: "2985"
  selflink: /api/v1/namespaces/default/configmaps/example-redis-config
  uid: d65739c1-fbbb-11e5-8a72-68f728db1985
```

现在，创建使用此 **ConfigMap** 的 pod:

1. 创建类似以下内容的 pod 定义，并将它保存到文件中，如 *redis-pod.yaml* ：

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6379
```

```
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: example-redis-config
        items:
        - key: redis-config
          path: redis.conf
```

2. 创建 pod：

```
$ oc create -f redis-pod.yaml
```

新创建的 pod 有一个 **ConfigMap** 卷，它会将 **example-redis-config ConfigMap** 的 **redis-config** 键放入一个名为 ***redis.conf*** 的文件中。此卷挂载到 Redis 容器中的 ***/redis-master*** 目录，将配置文件放在 ***/redis-master/redis.conf*** 中，这是镜像查找 master 的 Redis 配置文件的位置。

如果您使用 **oc exec** 进入此 pod 并运行 **redis-cli** 工具，则可以检查是否正确应用了配置：

```
$ oc exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

21.5. 限制

在 pod 中使用前，需要先创建 **ConfigMap**。可以编写控制器来容许缺少的配置数据 ; 根据具体情况，通过 **ConfigMap** 配置的独立组件。

ConfigMap 对象驻留在一个项目中。它们只能被同一项目中的 pod 引用。

Kubelet 只支持为它从 API 服务器获取的 pod 使用 **ConfigMap**。这包括使用 CLI 创建或间接从复制控制器创建的任何 pod。它不包括使用 OpenShift Container Platform 节点的**--manifest-url** 标记，它的 **--config** 标记，或它的 REST API (这些都不是创建 pod 的常规方法) 创建的 pod。

第 22 章 DOWNWARD API

22.1. 概述

Downward API 是一种允许容器消耗 API 对象信息的机制，而无需与 OpenShift Container Platform 耦合。此类信息包括 pod 的名称、命名空间和资源值。容器可以使用环境变量或卷插件来消耗来自 Downward API 的信息。

22.2. 选择字段

pod 中的字段通过 **FieldRef** API 类型来选择。**FieldRef** 有两个字段：

字段	描述
fieldPath	要选择的字段的路径，这相对于 pod。
apiVersion	要在其中解释 **fieldPath** 选择器的 API 版本。

目前，v1 API 中的有效选择器包括：

选择器	描述
metadata.name	pod 的名称。在环境变量和卷中均受支持。
metadata.namespace	pod 的命名空间。在环境变量和卷中均受支持。
metadata.labels	pod 的标签。仅在卷中支持，环境变量中不支持。
metadata.annotations	pod 的注解。仅在卷中支持，环境变量中不支持。
status.podIP	pod 的 IP。仅在环境变量中支持，卷中不支持。

若未指定 **apiVersion** 字段，则默认为所属 pod 模板的 API 版本。

22.3. 使用 DOWNWARD API 消耗容器值

22.3.1. 使用环境变量

消耗 Downward API 的一种机制是使用容器的环境变量。**EnvVar** 类型的 **valueFrom** 字段（类型为 **EnvVarSource**）用于指定变量的值应来自一个 **FieldRef** 源，而不是 **value** 字段指定的字面值。以后可能会支持其他源 ; 当前，源的 **fieldRef** 字段用于从 Downward API 中选择一个字段。

只有 pod 常量属性可以这种方式消耗，因为一旦进程启动并且将变量值已更改的通知发送给进程，就无法更新环境变量。使用环境变量支持的字段包括：

- Pod 名称
- Pod 命名空间

1. 创建 ***pod.yaml*** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
  restartPolicy: Never
```

2. 从 ***pod.yaml*** 文件创建 pod：

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

3. 检查容器的日志，以查看 **MY_POD_NAME** 和 **MY_POD_NAMESPACE** 值：

```
$ oc logs -p dapi-env-test-pod
```

22.3.2. 使用卷插件

消耗 Downward API 的另一个机制是使用卷插件。Downward API 卷插件会创建一个将字段扩展到文件中的卷。**VolumeSource** API 对象的 **metadata** 字段用于配置这个卷。插件支持以下字段：

- Pod 名称
- Pod 命名空间
- Pod 注解
- Pod 标签

例 22.1. Downward API 卷插件配置

```
spec:
  volumes:
    - name: podinfo
      downwardAPI:: (1)
        items: (2)
          -name: "labels" (3)
           fieldRef:
             fieldPath: metadata.labels (4)
```

1. 卷源的 **metadata** 字段配置 Downward API 卷。
2. **items** 字段包含要扩展到卷的字段列表。
3. 将字段扩展到的文件的名称。
4. 要扩展的字段的选择器。

例如：

1. 创建 ***volume-pod.yaml*** 文件：

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: "345"
    annotation2: "456"
spec:
  containers:
    - name: volume-test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "cat /tmp/etc/pod_labels /tmp/etc/pod_annotations"]
      volumeMounts:
        - name: podinfo
          mountPath: /tmp/etc
          readOnly: false
  volumes:
  - name: podinfo
    downwardAPI:
      defaultMode: 420
      items:
      - fieldRef:
          fieldPath: metadata.name
        path: pod_name
      - fieldRef:
          fieldPath: metadata.namespace
        path: pod_namespace
      - fieldRef:
          fieldPath: metadata.labels
        path: pod_labels
      - fieldRef:
          fieldPath: metadata.annotations
        path: pod_annotations
  restartPolicy: Never
```

2. 从 ***volume-pod.yaml*** 文件创建 pod：

```
$ oc create -f volume-pod.yaml
```

3. 检查容器的日志，并验证配置的字段是否存在：

```
$ oc logs -p dapi-volume-test-pod
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

22.4. 使用 DOWNWARD API 消耗容器资源

在创建 pod 时，您可以使用 Downward API 注入关于计算资源请求和限制的信息，以便镜像和应用程序作者能够正确地为特定环境创建镜像。

您可以使用环境变量和卷插件方法进行此操作。

22.4.1. 使用环境变量

1. 在创建 pod 配置时，在 **spec.container** 字段中指定与 **resources** 字段的内容对应的环境变量：

```
....
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.memory
....
```

如果容器配置中没有包含资源限制，Downward API 会默认使用节点的 CPU 和内存可分配量。

2. 从 ***pod.yaml*** 文件创建 pod：

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

22.4.2. 使用卷插件

1. 在创建 pod 配置时，使用 **spec.volumes.downwardAPI.items** 字段来描述与 **spec.resources** 字段对应的所需资源：

```
....
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c", "while true; do echo; if [[ -e /etc/cpu_limit ]]; then cat /etc/cpu_limit;
fi; if [[ -e /etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e /etc/mem_limit ]]; then cat
/etc/mem_limit; fi; if [[ -e /etc/mem_request ]]; then cat /etc/mem_request; fi; sleep 5; done"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
....
```

如果容器配置中没有包含资源限制，Downward API 会默认使用节点的 CPU 和内存可分配量。

2. 从 ***volume-pod.yaml*** 文件创建 pod：

```
$ oc create -f volume-pod.yaml
```

22.5. 使用 DOWNWARD API 消耗 SECRET

在创建 pod 时，您可以使用 Downward API 注入 Secret，以便镜像和应用程序作者能够为特定环境创建镜像。

22.5.1. 使用环境变量

1. 创建 ***secret.yaml*** 文件：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  password: cGFzc3dvcmQ=
  username: ZGV2ZWxvcGVy
type: kubernetes.io/basic-auth
```

2. 从 secret.yaml 文件创建 **Secret**：

```
oc create -f secret.yaml
```

3. 创建 ***pod.yaml*** 文件来引用上述 **Secret** 中的 **username** 字段：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
  restartPolicy: Never
```

4. 从 ***pod.yaml*** 文件创建 pod：

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

5. 检查容器日志中的 **MY_SECRET_USERNAME** 值：

```
$ oc logs -p dapi-env-test-pod
```

22.6. 使用 DOWNWARD API 消耗 CONFIGMAP

在创建 pod 时，您可以使用 Downward API 注入 ConfigMap 值，以便镜像和应用程序作者能够为特定环境创建镜像。

22.6.1. 使用环境变量

1. 创建 ***configmap.yaml*** 文件：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

2. 从 ***configmap.yaml*** 文件创建 **ConfigMap**：

```
oc create -f configmap.yaml
```

3. 创建 ***pod.yaml*** 文件来引用上述 **ConfigMap**：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_CONFIGMAP_VALUE
          valueFrom:
            configMapKeyRef:
              name: myconfigmap
              key: mykey
  restartPolicy: Never
```

4. 从 ***pod.yaml*** 文件创建 pod：

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

5. 检查容器日志中的 **MY_CONFIGMAP_VALUE** 值：

```
$ oc logs -p dapi-env-test-pod
```

22.7. 环境变量参考

在创建 pod 时，您可以使用 **$()** 语法引用之前定义的环境变量的值。如果无法解析环境变量引用，则该值将保留为提供的字符串。

22.7.1. 使用环境变量引用

1. 创建 ***pod.yaml*** 文件来引用现有的环境变量：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_EXISTING_ENV
          value: my_value
        - name: MY_ENV_VAR_REF_ENV
          value: $(MY_EXISTING_ENV)
  restartPolicy: Never
```

2. 从 ***pod.yaml*** 文件创建 pod：

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

3. 检查容器日志中的 **MY_ENV_VAR_REF_ENV** 值：

```
$ oc logs -p dapi-env-test-pod
```

22.7.2. 转义环境变量参考

在创建 pod 时，您可以使用双美元符号来转义环境变量引用。然后，其值将设为所提供值的单美元符号版本。

1. 创建 ***pod.yaml*** 文件来引用现有的环境变量：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_NEW_ENV
          value: $$(SOME_OTHER_ENV)
  restartPolicy: Never
```

2. 从 ***pod.yaml*** 文件创建 pod：

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

3. 检查容器日志中的 **MY_NEW_ENV** 值：

```
$ oc logs -p dapi-env-test-pod
```

第 23 章 投射卷

23.1. 概述

一个*投射卷*会将几个现有的卷源映射到同一个目录中。

目前，可以投射以下类型的卷源：

- Secrets
- Config Map
- Downward API

注意

所有源都必须位于与 pod 相同的命名空间中。

投射卷可将这些卷源的任何组合映射到一个目录中，让用户能够：

- 使用来自多个 secret、configmap 的密钥和 downward API 信息自动填充单个卷，以便在一个目录中整合不同来源的信息；
- 使用来自多个 secret、configmap 的密钥和 downward API 信息填充单个卷，并且明确指定各个项目的路径，以便能够完全掌控卷中的内容。

23.2. 使用示例

以下一般情景演示了如何使用投射卷。

- **ConfigMap、Secret、Downward API。**通过投射卷，使用包含密码的配置数据来部署容器。使用这些资源的应用程序可以在 Kubernetes 上部署 OpenStack。根据服务要用于生产环境还是测试，可能需要对配置数据进行不同的编译。如果 pod 标记了生产或测试用途，可以使用 Downward API 选择器 **metadata.labels** 来生成正确的 OpenStack 配置。
- **ConfigMap + Secret。**借助投射卷来部署涉及配置数据和密码的容器。例如，您可以执行存储为 configmap 的 Ansible playbook，并且某些敏感加密任务使用 vault 密码文件解密。
- **ConfigMap + Downward API。**借助投射卷来生成包含 pod 名称的配置（可通过 **metadata.name** 选择器使用）。然后，此应用程序可以将 pod 名称与请求一起传递，以在不使用 IP 跟踪的前提下轻松地判断来源。
- **Secret + Downward API。**借助投射卷，将 secret 用作公钥来加密 pod 的命名空间（可通过 **metadata.namespace** 选择器使用）。这个示例允许操作员使用应用程序安全地传送命名空间信息，而不必使用加密传输。

23.3. POD 规格示例

以下是用于创建投射卷的 pod 规格示例。

例 23.1. 带有 secret、downward API 和 configmap 的 Pod

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts: 1
    - name: all-in-one
      mountPath: "/projected-volume" 2
      readOnly: true 3
  volumes: 4
  - name: all-in-one 5
    projected:
      defaultMode: 0400 6
      sources:
      - secret:
          name: mysecret 7
          items:
            - key: username
              path: my-group/my-username 8
      - downwardAPI: 9
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap: 10
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
              mode: 0777 11
```

1. 为每个需要 secret 的容器添加 **volumeMounts** 部分。
2. 指定一个到还未使用的目录的路径，secret 将出现在这个目录中。
3. 将 **readOnly** 设为 **true**。
4. 添加一个 **volumes** 块，以列出每个投射卷源。
5. 为卷指定任意名称。
6. 设置文件的执行权限。
7. 添加 secret。输入 secret 对象的名称。必须列出您要使用的每个 secret。
8. 指定 **mountPath** 下 secret 文件的路径。此处，secret 文件位于 */projected-volume/my-group/my-config*。
9. 添加 Downward API 源。

10 添加 ConfigMap 源。

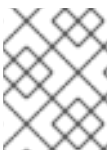
11 设置具体的投射模式

注意

如果 pod 中有多个容器，则每个容器都需要一个 **volumeMounts** 部分，但 **volumes** 部分只需一个即可。

例 23.2. 具有设定了非默认权限模式的多个 secret 的 Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      defaultMode: 0755
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```



注意

defaultMode 只能在投射级别上指定，而不针对每个卷源指定。但如上方所示，您可以明确设置每一个投射的 **mode**。

23.4. 路径注意事项

在创建投射卷时，请考虑以下与卷文件路径相关的情况。

配置路径相同时发生密钥间冲突

如果您使用同一路径配置多个密钥，则 pod 规格会视其为有效。以下示例中为 **mysecret** 和 **myconfigmap** 指定了相同的路径：

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/data
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/data
```

未配置路径的密钥之间发生冲突

只有在创建 pod 时所有路径都已知，才会进行运行时验证，这与上述情景类似。否则发生冲突时，最新指定的资源会覆盖所有之前指定的资源（在 pod 创建后更新的资源也是如此）。

一个路径为显式而另一个路径为自动投射时发生冲突

如果因为用户指定的路径与自动投射的数据匹配，从而发生冲突，则像前文所述一样，后面的资源将覆盖前面的资源

23.5. 为 POD 配置投射卷

以下示例演示了如何使用投射卷挂载现有 Secret 卷源。

可以使用这些步骤从本地文件创建用户名和密码 secret。然后，创建一个只运行一个容器的 pod，使用投射卷将 Secret 挂载到同一个共享目录中。

1. 创建包含 secret 的文件：
   例如：

   ```
   $ nano secret.yaml
   ```

   输入以下内容，根据需要替换密码和用户信息：

   ```
   apiVersion: v1
   kind: Secret
   ```

```
metadata:
  name: mysecret
type: Opaque
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
```

user 和 **pass** 值可以是采用 **base64** 编码的任意有效字符串。此处的示例使用了 base64 编码值 **user: admin** 和 **pass:1f2d1e2e67df**。

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFlMmU2N2Rm
```

2. 使用以下命令来创建 secret：

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

例如：

```
$ oc create -f secret.yaml
secret "mysecret" created
```

3. 您可以使用以下命令来检查是否创建了 secret：

```
$ oc get secret <secret-name>
$ oc get secret <secret-name> -o yaml
```

例如：

```
$ oc get secret mysecret
NAME       TYPE      DATA      AGE
mysecret   Opaque    2         17h
```

```
oc get secret mysecret -o yaml
apiVersion: v1
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-05-30T20:21:38Z
  name: mysecret
  namespace: default
  resourceVersion: "2107"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 959e0424-4575-11e7-9f97-fa163e4bd54c
type: Opaque
```

4. 创建类似以下示例的 pod 配置文件，其中包含一个 **volumes** 部分 ：

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: user
      - secret:
          name: pass
```

5. 从配置文件创建 pod：

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

例如：

```
$ oc create -f secret-pod.yaml
pod "test-projected-volume" created
```

6. 验证 pod 容器是否在运行，然后留意 Pod 的更改：

```
$ oc get pod <name>
```

输出结果应该类似以下示例：

```
$ oc get pod test-projected-volume
NAME                    READY     STATUS    RESTARTS   AGE
test-projected-volume   1/1       Running   0          14s
```

7. 在另一个终端中，使用 **oc exec** 命令打开到正在运行的容器的 shell：

```
$ oc exec -it <pod> <command>
```

例如：

```
$ oc exec -it test-projected-volume -- /bin/sh
```

8. 在 shell 中，验证 **projected-volumes** 目录是否包含您的投射源：

```
/ # ls
```

```
bin             home              root            tmp
dev             proc              run             usr
etc             projected-volume  sys             var
```

第 24 章 使用 DAEMONSET

24.1. 概述

daemonset 可用于在 OpenShift Container Platform 集群的特定节点或所有节点上运行 pod 副本。

使用 daemonset 创建共享存储，在集群的每一节点上运行日志 pod，或者在每个节点上部署监控代理。

为安全起见，只有集群管理员才能创建 daemonset。（授予用户守护进程设置权限。）

如需有关 daemonset 的更多信息，请参阅 Kubernetes 文档。

重要

Daemonset 调度与项目的默认节点选择器不相兼容。如果您没有成功禁用，daemonset 会与默认节点选择器合并，从而受到限制。这会造成在合并后节点选择器没有选中的节点上频繁地重新创建 pod，进而给集群带来意外的负载。

因此，

- 在开始使用 daemonset 之前，请通过将命名空间注解 **openshift.io/node-selector** 设置为空字符串来禁用命名空间中的默认项目范围节点选择器：

```
# oc patch namespace myproject -p \
    '{"metadata": {"annotations": {"openshift.io/node-selector": ""}}}'
```

- 如果您要创建新项目，请使用 **oc adm new-project --node-selector=""** 覆盖默认节点选择器。

24.2. 创建守护进程集

在创建 daemonset 时，请使用 **nodeSelector** 字段来指示 daemonset 应该在其上部署副本的节点。

1. 定义 daemonset yaml 文件：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
     matchLabels:
       name: hello-daemonset 1
  template:
    metadata:
      labels:
        name: hello-daemonset 2
    spec:
      nodeSelector: 3
        type: infra
      containers:
      - image: openshift/hello-openshift
        imagePullPolicy: Always
```

```
      name: registry
      ports:
      - containerPort: 80
        protocol: TCP
      resources: {}
      terminationMessagePath: /dev/termination-log
    serviceAccount: default
    terminationGracePeriodSeconds: 10
```

1. 决定哪些 pod 属于 daemonset 的标签选择器。
2. pod 模板的标签选择器。必须与上述标签选择器匹配。
3. 决定应该在哪些节点上部署 pod 副本的节点选择器。

2. 创建 daemonset 对象：

```
oc create -f daemonset.yaml
```

3. 验证 pod 是否已创建好，并且每个节点都有 pod 副本：

   a. 查找 daemonset pod：

```
$ oc get pods
hello-daemonset-cx6md   1/1       Running   0          2m
hello-daemonset-e3md9   1/1       Running   0          2m
```

   b. 查看 pod 以验证 pod 已放置到节点上：

```
$ oc describe pod/hello-daemonset-cx6md|grep Node
Node:        openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9|grep Node
Node:        openshift-node02.hostname.com/10.14.20.137
```

重要

- 如果更新 DaemonSet 的 pod 模板，现有的 pod 副本不受影响。
- 如果您删除了 DaemonSet，然后使用不同的模板创建新的 DaemonSet，但相同的标签选择器，它会将现有 pod 副本识别为具有匹配的标签，因此不会更新它们，也不会创建新的副本，尽管 pod 模板中存在不匹配。
- 如果您更改了节点标签，DaemonSet 会将 pod 添加到与新标签匹配的节点，并从不匹配新标签的节点中删除 pod。

要更新 DaemonSet，请通过删除旧副本或节点来强制创建新的 pod 副本。

第 25 章 POD 自动扩展

25.1. 概述

Pod 横向自动扩展（由 **HorizontalPodAutoscaler** 对象定义）指定系统应根据属于该复制控制器或部署配置的 pod 收集的指标来自动增加或减少扩展。

25.2. 使用 HORIZONTAL POD AUTOSCALER 的要求

要使用 pod 横向自动扩展，需要安装 OpenShift Container Platform 指标服务器：

```
$ ansible-playbook \
/usr/share/ansible/openshift-ansible/playbooks/metrics-server/config.yml \
-e openshift_metrics_server_install=true
```

您可以运行以下命令来验证服务器是否已正确安装：

```
$ oc adm top node
$ oc adm top pod
```

有关这些命令的更多信息，请参阅 查看节点和查看 Pod。

25.3. 支持的指标

pod 横向自动扩展支持以下指标：

表 25.1. 指标

指标	描述	API 版本
CPU 使用率	请求的 CPU 的百分比	**autoscaling/v1**, **autoscaling/v2beta1**
内存使用率	请求的内存百分比。	**autoscaling/v2beta1**

25.4. 自动缩放

您可以创建一个 pod 横向自动扩展来指定您要运行的 pod 的最小和最大数量，以及 pod 的目标 CPU 使用率 或 内存使用率。

创建横向 pod 自动缩放器后，它将开始尝试查询 Heapster 的指标。Heapster 获取初始指标之前，可能需要过两分钟。

在 Heapster 中提供指标后，pod 横向自动扩展会计算当前指标使用率与所需指标使用率的比率，并相应地扩展或缩减。该扩展会按常规间隔进行，但可能需要一到两分钟时间才能让指标变为 Heapster。

对于复制控制器，这种缩放直接与复制控制器的副本对应。对于部署配置，缩放直接与部署配置的副本计数对应。注意，自动缩放仅应用到 **Complete** 阶段的最新部署。

OpenShift Container Platform 会自动考虑资源情况，并防止在资源激增期间进行不必要的自动缩放，比如在启动过程中。处于 **unready** 状态的 pod 在扩展时具有 **0 CPU** 用量，自动扩展在缩减时会忽略这些

pod。没有已知指标的 Pod 在扩展时具有 **0% CPU** 用量，在缩减时具有 **100% CPU** 用量。这在 HPA 决策过程中提供更高的稳定性。要使用这个功能，您必须配置 就绪度检查 来确定新 pod 是否准备就绪。

25.4.1. 为 CPU 使用率自动扩展

为 CPU 使用率自动扩展时，您可以使用 **oc autoscale** 命令，并指定要在任意给定时间运行的 pod 的最大数量，以及 pod 的目标平均 CPU 使用率。您可以选择指定最小 pod 数量，否则 OpenShift Container Platform 服务器会为 pod 赋予默认值。

例如：

```
$ oc autoscale dc/frontend --max 10 --cpu-percent=80
deploymentconfig "frontend" autoscaled
```

example 命令为使用以下定义的现有 DeploymentConfig 创建一个 pod 横向自动扩展：

Pod 横向自动扩展对象定义

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend (1)
spec:
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1 (2)
    kind: DeploymentConfig (3)
    name: frontend (4)
    subresource: scale
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  targetCPUUtilizationPercentage: 80 (7)
```

1. 此 pod 横向自动扩展对象的名称。
2. 要缩放的对象 API 版本：
   - 对于 ReplicationController，使用 **v1**，
   - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
3. 要缩放的对象种类，可以是 **ReplicationController** 或 **DeploymentConfig**。
4. 您要扩展的现有对象的名称。
5. 缩减时的最小副本数量。默认值为 **1**。
6. 向上扩展时的最大副本数量。
7. 理想状态下每个 pod 应使用的请求 CPU 的百分比。

另外，**oc autoscale** 命令在使用 pod 横向自动扩展 **v2beta1** 版本时，会根据定义创建一个 pod 横向自动扩展：

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-cpu (1)
spec:
  scaleTargetRef:
    apiVersion: v1 (2)
    kind: ReplicationController (3)
    name: hello-hpa-cpu (4)
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50 (7)
```

1. 此 pod 横向自动扩展对象的名称。
2. 要缩放的对象 API 版本：
   - 对于 ReplicationController，使用 **v1**，
   - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
3. 要缩放的对象种类，可以是 **ReplicationController** 或 **DeploymentConfig**。
4. 您要扩展的现有对象的名称。
5. 缩减时的最小副本数量。默认值为 **1**。
6. 向上扩展时的最大副本数量。
7. 每个 pod 应使用的请求 CPU 的平均百分比。

25.4.2. Autoscaling for Memory Utilization

与基于 CPU 的自动缩放不同，基于内存的自动缩放要求使用 YAML 指定自动扩展，而不是使用 **oc autoscale** 命令。另外，您可以指定最小 pod 数量和 pod 的目标平均内存使用率，否则这些 pod 会从 OpenShift Container Platform 服务器给出默认值。

重要

根据内存使用率自动缩放是一项技术预览功能。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

如需红帽技术预览功能支持范围的更多信息，请参阅 https://access.redhat.com/support/offerings/techpreview/。

注意

基于内存的自动缩放仅可通过 **v2beta1** 版的自动扩展 API 使用。

使用基于内存的自动扩展：

1. 启用基于内存的自动扩展：

   a. 将以下内容添加到集群的 **master-config.yaml** 文件中：

```
...
apiServerArguments:
  runtime-config:
  - apis/autoscaling/v2beta1=true
...
```

   b. 重启 OpenShift Container Platform 服务：

```
$ master-restart api
$ master-restart controllers
```

2. 如果需要，获取您要缩放的对象名称：

```
$ oc get dc

NAME               REVISION   DESIRED   CURRENT   TRIGGERED BY
frontend           1          5         0         config
```

3. 将以下内容放入一个文件中，如 **hpa.yaml** ：

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory (1)
spec:
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1 (2)
    kind: DeploymentConfig (3)
    name: frontend (4)
  minReplicas: 2 (5)
  maxReplicas: 10 (6)
  metrics:
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: 50 (7)
```

(1) 此 pod 横向自动扩展对象的名称。

(2) 要缩放的对象 API 版本：

- 对于 ReplicationController，使用 **v1**，
- 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。

(3) 要缩放的对象种类，可以是 **ReplicationController** 或 **DeploymentConfig**。

(4) 您要扩展的现有对象的名称。

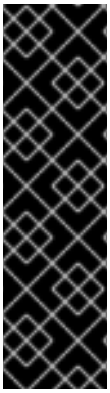
5 缩减时的最小副本数量。默认值为 **1**。

6 向上扩展时的最大副本数量。

7 每个 pod 应使用的所请求内存的平均百分比。

4. 然后，从以上文件创建自动扩展：

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



重要

要使基于内存的自动缩放正常工作，内存用量必须与副本数成比例增加和减小。平均而言：

- 增加副本数一定会导致每个 pod 的内存（工作集）用量总体降低。
- 减少副本数一定会导致每个 pod 的内存用量总体增高。

使用 OpenShift Web 控制台检查应用的内存行为，并确保应用程序在使用基于内存的自动扩展前满足这些要求。

25.5. 查看 HORIZONTAL POD AUTOSCALER

查看 pod 横向自动扩展的状态：

- 使用 **oc get** 命令查看 CPU 使用率和 pod 限制的信息：

```
$ oc get hpa/hpa-resource-metrics-cpu
NAME                     REFERENCE                                TARGET    CURRENT  MINPODS
MAXPODS    AGE
hpa-resource-metrics-cpu    DeploymentConfig/default/frontend/scale   80%      79%     1
10         8d
```

输出包括：

  - **Target**。部署配置控制的所有 Pod 的目标平均 CPU 使用率。
  - **Current**。由部署配置控制的所有 Pod 的当前 CPU 使用率。
  - **Minpods/Maxpods**.自动缩放器可设置的最小和最大副本数。

- 使用 **oc describe** 命令获取有关 pod 横向自动扩展对象的详细信息。

```
$ oc describe hpa/hpa-resource-metrics-cpu
Name:                        hpa-resource-metrics-cpu
Namespace:                   default
Labels:                      <none>
CreationTimestamp:           Mon, 26 Oct 2015 21:13:47 -0400
Reference:                   DeploymentConfig/default/frontend/scale
Target CPU utilization:      80% 1
Current CPU utilization:     79% 2
Min replicas:                1 3
Max replicas:                4 4
```

```
ReplicationController pods:    1 current / 1 desired
Conditions: 5
  Type              Status  Reason              Message
  ----              ------  ------              -------
  AbleToScale       True    ReadyForNewScale    the last scale time was sufficiently old
as to warrant a new scale
  ScalingActive     True    ValidMetricFound    the HPA was able to successfully calculate
a replica count from pods metric http_requests
  ScalingLimited    False   DesiredWithinRange  the desired replica count is within the
acceptable range
Events:
```

1. 每个 pod 应使用的所请求内存的平均百分比。
2. 由部署配置控制的所有 Pod 的当前 CPU 使用率。
3. 缩减到的最小副本数。
4. 向上扩展最多的副本数。
5. 如果对象使用 **v2alpha1** API，则会显示 状态条件。

25.5.1. 查看 Horizontal Pod Autoscaler 状态条件

您可以使用状态条件集来确定横向 pod 自动缩放器是否能够扩展以及是否以任何方式限制它。

注意

pod 横向自动扩展状态条件可通过 **v2beta1** 版的自动扩展 API 使用。

设定了以下状态条件：

- **AbleToScale** 指示 pod 横向自动扩展是否可以获取和更新扩展，以及任何 backoff 条件是否阻止扩展。
  - **True** 条件表示允许缩放。
  - **False** 条件表示因为指定原因不允许缩放。
- **ScalingActive** 指示 pod 横向自动扩展是否已启用（目标的副本计数不为零），并且可以计算所需的扩展。
  - **True** 条件表示指标工作正常。
  - **False** 条件通常表示获取指标时出现问题。
- **ScalingLimited** 表示不允许自动扩展，因为达到最大或最小副本数。
  - **True** 条件表示您需要提高或降低最小或最大副本数才能进行缩放。
  - **False** 条件表示允许请求的缩放。

要查看影响 pod 横向自动扩展的条件，请使用 **oc describe hpa**。条件会出现在 **status.conditions** 字段中：

```
$ oc describe hpa cm-test
Name:                          cm-test
Namespace:                     prom
Labels:                        <none>
Annotations:                   <none>
CreationTimestamp:             Fri, 16 Jun 2017 18:09:22 +0000
Reference:                     ReplicationController/cm-test
Metrics:                       ( current / target )
  "http_requests" on pods:     66m / 500m
Min replicas:                  1
Max replicas:                  4
ReplicationController pods:    1 current / 1 desired
Conditions:                    1
  Type                 Status  Reason               Message
  ----                 ------  ------               -------
  AbleToScale          True    ReadyForNewScale     the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive        True    ValidMetricFound     the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited       False   DesiredWithinRange   the desired replica count is within the acceptable
range
Events:
```

1 pod 横向自动扩展状态消息。

- **AbleToScale** 条件指示 HPA 是否能够获取和更新扩展，以及任何与退避相关的条件是否可防止扩展。
- **ScalingActive** 条件指示 HPA 是否已启用（例如，目标副本数不为零），并且可以计算所需的扩展。False 的状态通常表示获取指标时出现问题。
- **ScalingLimited** 条件表示所需的规模由 pod 横向自动扩展限定最大或最小限制。**True** 状态通常表示您可能需要提高或降低 pod 横向自动扩展中的最小和最大副本数限制。

下例中是一个无法缩放的 pod：

```
Conditions:
  Type          Status    Reason          Message
  ----          ------    ------          -------
  AbleToScale   False     FailedGetScale  the HPA controller was unable to get the target's current
scale: replicationcontrollers/scale.extensions "hello-hpa-cpu" not found
```

下例中是一个无法获得缩放所需指标的 pod：

```
Conditions:
  Type                 Status    Reason                  Message
  ----                 ------    ------                  -------
  AbleToScale          True      SucceededGetScale       the HPA controller was able to get the target's
current scale
  ScalingActive        False     FailedGetResourceMetric the HPA was unable to compute the replica
count: unable to get metrics for resource cpu: no metrics returned from heapster
```

下例中是一个请求的自动缩放低于所需下限的 pod：

```
Conditions:
  Type            Status    Reason              Message
  ----            ------    ------              -------
  AbleToScale     True      ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive   True      ValidMetricFound    the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited  False     DesiredWithinRange  the desired replica count is within the acceptable
range
Events:
```

第 26 章 管理卷

26.1. 概述

默认情况下，容器不具持久性；重启之后，其中的内容会被清除。卷是挂载的文件系统，供 pod 及其容器使用，可以通过多个主机上本地或网络附加存储端点来支持。

为确保卷上的文件系统不包含任何错误，并在出现错误时尽可能进行修复，OpenShift Container Platform 在调用 **mount** 实用程序之前会先调用 **fsck**。在添加卷或更新现有卷时会出现这种情况。

最简单的卷类型是 **emptyDir**，这是单一机器上的一个临时目录。管理员也可以允许您请求自动附加到 pod 的持久性卷。

注意

如果集群管理员启用了 FSGroup 参数，则 **emptyDir** 卷存储可能会受到基于 pod FSGroup 的配额的限制。

您可以使用 CLI 命令 **oc set volume**，为任何使用 pod 模板的对象（如 复制控制器或部署配置 ） 添加、更新或删除卷和卷挂载。您还可以 列出 pod 中的卷，或列出具有 pod 模板的任何对象。

26.2. 常规 CLI 用法

oc set volume 命令使用以下通用语法：

```
$ oc set volume <object_selection> <operation> <mandatory_parameters> <optional_parameters>
```

本主题使用 ***<object_type>/<name>*** 的形式用于 ***<object_selection>***。但是，您可以选择以下选项之一：

表 26.1. 对象选择

语法	描述	示例
<object_type> <name>	选择类型为 ***<object_type>*** 的 ***<name>***。	**deploymentConfig registry**
<object_type>/<name>	选择类型为 ***<object_type>*** 的 ***<name>***。	**deploymentConfig/registry**
<object_type> **--selector=*<object_label_selector>***	选择与给定标签选择器匹配且类型为 ***<object_type>*** 的资源。	**deploymentConfig --selector="name=registry"**
<object_type> **--all**	选择类型为 ***<object_type>*** 的所有资源。	**deploymentConfig --all**
-f 或 **--filename=*<file_name>***	用于编辑资源的文件名、目录或文件 URL。	**-f registry-deployment-config.json**

<operation> 可以是 **--add**, **--remove**, 或 **--list** 之一。

任何 ***<mandatory_parameters>*** 或 ***<optional_parameters>*** 都特定于所选操作，并在后续小节中讨论。

26.3. 添加卷

将卷和/或卷挂载添加到 pod 模板中：

```
$ oc set volume <object_type>/<name> --add [options]
```

表 26.2. 添加卷时支持的选项

选项	描述	默认
--name	卷的名称。	若未指定，则自动生成。
-t, --type	卷源的名称。支持的值有 **emptyDir**、**hostPath**、**secret**、**configmap**、**persistentVolumeClaim** 或 **projected**。	**emptyDir**
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	'*'
-m, --mount-path	所选容器内的挂载路径。	
--path	主机路径。**--type=hostPath** 的必要参数。	
--secret-name	secret 的名称。**--type=secret** 的必要参数。	
--configmap-name	configmap 的名称。**--type=configmap** 的必要参数。	
--claim-name	持久性卷声明的名称。**--type=persistentVolumeClaim** 的必要参数。	
--source	以 JSON 字符串表示的卷源详情。如果 **--type** 不支持所需的卷源，则建议使用此参数。	
-o, --output	显示修改后的对象，而不在服务器上更新它们。支持的值有 **json** 和 **yaml**。	
--output-version	输出给定版本的修改后对象。	**api-version**

例子

将新卷源 **emptyDir** 添加到部署配置 **registry** 中：

```
$ oc set volume dc/registry --add
```

为复制控制器 **r1** 添加含有 secret **secret1** 的卷 **v1**，并挂载到容器中的 ***/data*** ：

```
$ oc set volume rc/r1 --add --name=v1 --type=secret --secret-name='secret1' --mount-path=/data
```

使用声明名称 **pvc1** 将现有持久性卷 **v1** 添加到磁盘上的部署配置 ***dc.json***，将卷挂载到容器 **c1** 中的 ***/data*** 并更新服务器上的部署配置：

```
$ oc set volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

26.4. 更新卷

更新现有卷或卷挂载与 添加卷 相同，但使用 **--overwrite** 选项：

```
$ oc set volume <object_type>/<name> --add --overwrite [options]
```

示例
将复制控制器 **r1** 的现有卷 **v1** 替换为现有的持久性卷声明 **pvc1** ：

```
$ oc set volume rc/r1 --add --overwrite --name=v1 --type=persistentVolumeClaim --claim-name=pvc1
```

将卷 **v1** 的部署配置 **d1** 挂载点更改为 ***/opt*** ：

```
$ oc set volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

26.5. 删除卷

从 pod 模板中移除卷或卷挂载：

```
$ oc set volume <object_type>/<name> --remove [options]
```

表 26.3. 移除卷时支持的选项

选项	描述	默认
--name	卷的名称。	
-c, --containers	按名称选择容器。它还可以使用通配符 **'*'** 来匹配任意字符。	**'*'**
--confirm	指定您想要一次性移除多个卷。	
-o, --output	显示修改后的对象，而不在服务器上更新它们。支持的值有 **json** 和 **yaml**。	

选项	描述	默认
--output-version	输出给定版本的修改后对象。	**api-version**

例子
从部署配置 **d1** 中删除卷 **v1** ：

```
$ oc set volume dc/d1 --remove --name=v1
```

从部署配置 **d1** 的容器 **c1** 卸载卷 **v1**，并在卷 **v1** 未被 **d1** 上的任何容器引用时删除它：

```
$ oc set volume dc/d1 --remove --name=v1 --containers=c1
```

删除复制控制器 **r1** 的所有卷：

```
$ oc set volume rc/r1 --remove --confirm
```

26.6. 列出卷

列出 pod 或 pod 模板的卷或卷挂载：

```
$ oc set volume <object_type>/<name> --list [options]
```

列出卷支持的选项：

选项	描述	默认
--name	卷的名称。	
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	'*'

例子
列出 pod **p1** 的所有卷：

```
$ oc set volume pod/p1 --list
```

列出所有部署配置中定义的卷 **v1** ：

```
$ oc set volume dc --all --name=v1
```

26.7. 指定子路径

使用 **volumeMounts.subPath** 属性在卷中指定 **subPath**，而不是卷的根目录。subPath 允许您在一个 pod 中为多个卷共享一个卷。

要查看卷中的文件列表，请运行 **oc rsh** 命令：

```
$ oc rsh <pod>
sh-4.2$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

指定 **subPath**：

subPath 用法示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql (1)
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html (2)
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

1. 数据库存储在 **mysql** 文件夹中。
2. HTML 内容存储在 **html** 文件夹中。

第 27 章 使用持久性卷

27.1. 概述

PersistentVolume 对象是 OpenShift Container Platform 集群中的存储资源。通过从 GCE Persistent Disk、AWS Elastic Block Store(EBS)和 NFS 挂载等源创建 **PersistentVolume** 对象来置备存储。

注意

配置集群指南为集群管理员提供了使用 NFS, GlusterFS, Ceph RBD, OpenStack Cinder, AWS EBS, GCE Persistent Disk, iSCSI, 和 Fibre Channel 使用持久性存储置备 OpenShift Container Platform 集群的说明。

存储可通过向资源发送声明来为您提供。您可以使用 **PersistentVolumeClaim** 对象对存储资源发出请求；该声明与通常与您的请求匹配的卷配对。

27.2. 请求存储

您可以通过在项目中创建 **PersistentVolumeClaim** 对象来请求存储：

持久性卷声明对象定义

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "pv0001"
```

27.3. 卷和声明绑定

PersistentVolume 是一个特定资源。**PersistentVolumeClaim** 是具有特定属性的资源请求，如存储大小。在两者之间，是一个与一个可用卷的声明匹配的进程，并将它们绑定到一个卷。这允许将声明用作 pod 中的卷。OpenShift Container Platform 找到支持声明的卷，并将其挂载到 pod。

您可以使用 CLI 查询声明或卷是否绑定：

```
$ oc get pvc
NAME      LABELS    STATUS    VOLUME
claim1    map[]     Bound     pv0001

$ oc get pv
NAME            LABELS          CAPACITY        ACCESSMODES       STATUS    CLAIM
pv0001          map[]           5368709120      RWO               Bound     yournamespace / claim1
```

27.4. 在 POD 中作为卷声明

pod 将 **PersistentVolumeClaim** 用作卷。OpenShift Container Platform 在与 pod 相同的命名空间中找到给定名称的声明，然后使用该声明查找要挂载的对应卷。

具有声明的 Pod 定义

```
apiVersion: "v1"
kind: "Pod"
metadata:
  name: "mypod"
  labels:
    name: "frontendhttp"
spec:
  containers:
    -
      name: "myfrontend"
      image: openshift/hello-openshift
      ports:
        -
          containerPort: 80
          name: "http-server"
      volumeMounts:
        -
          mountPath: "/var/www/html"
          name: "pvol"
  volumes:
    -
      name: "pvol"
      persistentVolumeClaim:
        claimName: "claim1"
```

27.5. 卷和 CLAIM PRE-BINDING

如果您知道您希望 **PersistentVolumeClaim** 绑定到的 **PersistentVolume**，您可以使用 **volumeName** 字段在 PVC 中指定 PV。此方法跳过了正常的匹配和绑定进程。PVC 只会绑定到在 **volumeName** 中指定的同名的 PV。如果这个 PV 存在且为 **Available**，则 PV 和 PVC 将受到绑定，无论 PV 是否满足 PVC 的标签选择器、访问模式和资源请求。

例 27.1. 使用 volumeName 的持久性卷声明对象定义

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "pv0001"
```

重要

设置 **claimRefs** 的功能是针对上述用例的一个临时临时解决方案。用于限制某个卷在开发中的长期解决方案。

注意

在代表用户设置 **claimRefs** 前，集群管理员应首先考虑配置 selector-label 卷绑定。

您可能还希望集群管理员只为声明"reserve"卷，以便其他 nobody 的声明可以在您的操作之前绑定到它。在这种情况下，管理员可以使用 **claimRef** 字段在 PV 中指定 PVC。PV 只能绑定到一个 PVC，其名称和命名空间在 **claimRef** 中指定。PVC 的访问模式和资源请求必须仍满足，这样才能绑定 PV 和 PVC，但标签选择器被忽略。

使用 claimRef 的持久性卷对象定义

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  nfs:
    path: /tmp
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Retain
  claimRef:
    name: claim1
    namespace: default
```

在 PVC 中指定 **volumeName** 不会阻止不同的 PVC 在操作前绑定到指定的 PV。您的声明将保持 **Pending**，直到 PV 处于 **Available** 状态。

在 PV 中指定 **claimRef** 不会阻止指定的 PVC 绑定到不同的 PV。PVC 可以自由选择另外一个 PV，以根据常规绑定过程绑定。因此，为了避免这些情况，并确保您的声明绑定到您想要的卷，您必须确保同时指定了 **volumeName** 和 **claimRef**。

您可以通过检查 **pv.kubernetes.io/bound-by-controller** 注解的 **Bound** PV 和 PVC 对，告知设置 **volumeName** 和/或 **claimRef** 会影响匹配的和绑定过程。在您设置 **volumeName** 和/或 **claimRef** 的 PV 和 PVC 没有这个注解，但普通 PV 和 PVC 会被设置为 **"yes"**。

当 PV 的 **claimRef** 设置为某些 PVC 名称和命名空间时，并根据 **Retain** 重新声明策略重新声明，则其 **claimRef** 仍会保留为同一个 PVC 名称和命名空间，即使 PVC 或整个命名空间不再存在。

第 28 章 扩展持久性卷

28.1. 启用持久性卷声明扩展

要允许 OpenShift Container Platform 用户扩展持久性卷声明(PVC)，OpenShift Container Platform 管理员必须创建或更新将 **allowVolumeExpansion** 设置为 **true** 的 StorageClass。只有从该类创建的 PVC 可以扩展。

例如，OpenShift Container Platform 管理员可以将 **allowVolumeExpansion** 属性添加到 StorageClass 的配置中：

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gluster-container
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://heketi-storage-project.cloudapps.mystorage.com"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
allowVolumeExpansion: true
```

28.2. 扩展基于 GLUSTERFS 的持久性卷声明

当 OpenShift Container Platform 管理员创建了一个使用 **allowVolumeExpansion** 设置为 **true** 的 StorageClass 后，您可以从该类创建 PVC，之后您可以根据需要编辑 PVC 并请求新的大小。

例如：

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: gluster-mysql
spec:
  storageClass: "storageClassWithFlagSet"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi (1)
```

(1) 您可以通过更新 **spec.resources.requests** 来请求扩展卷。

28.3. 使用文件系统扩展 PVC

根据需要调整文件系统大小（如 GCE PD、EBS 和 Cinder）的卷类型扩展 PVC 分为两个步骤。这个过程通常涉及在 CloudProvider 中扩展卷对象，然后在实际节点上扩展文件系统。

只有在使用这个卷启动新的 pod 时，才会在该节点中扩展文件系统。

以下过程假设 PVC 先前是从将 **allowVolumeExpansion** 设置为 **true** 的 StorageClass 创建的：

1. 通过编辑 **spec.resources.requests**来修改 PVC 并请求一个新的大小。在 CloudProvider 对象完成重新定义大小后，PVC 被设置为 **FileSystemResizePending**。

2. 输入以下命令检查条件：

```
oc describe pvc <pvc_name>
```

当 CloudProvider 对象完成重新定义大小时，持久性卷(PV)对象会反映 **PersistentVolume.Spec.Capacity** 中新请求的大小。此时，您可以创建或者重新创建使用 PVC 的新 pod 来完成文件系统重新定义大小的过程。当 pod 运行后，新请求的大小就可用，同时 **FileSystemResizePending** 条件从 PVC 中删除。

28.4. 在扩展卷失败时进行恢复

如果扩展底层存储在 master 或节点上失败，OpenShift Container Platform 管理员可以手动恢复 PVC 状态，并取消控制器持续重试的大小请求，而无需管理员干预。

目前，可以通过完成以下步骤手动完成此操作：

1. 使用 **Retain** 重新声明策略标记绑定到声明(PVC)的 PV。编辑 PV，把 **persistentVolumeReclaimPolicy** 的值改为 **Retain**。

2. 删除 PVC（将在以后重新创建）。

3. 为了确保可以把 PVC 绑定到一个带有 **Retain** 设置的 PV，手工编辑 PV，把 **claimRef** 从 PV specs 中删除。这会将 PV 标记为 **Available**。有关绑定 PVC 的更多信息，请参阅 卷和声明绑定。

4. 以较小的大小或底层存储供应商分配的大小，重新创建 PVC。另外，将 PVC 的 **volumeName** 字段设置为 PV 的名称。这使 PVC 只会绑定到置备的 PV。

5. 恢复 PV 上的 reclaim 策略。

第 29 章 执行远程命令

29.1. 概述

您可以使用 CLI 在容器中执行远程命令。这可让您为容器中的常规操作运行常规 Linux 命令。

重要

为了安全起见，**oc exec** 命令在访问特权容器时无法工作，除非该命令由 **cluster-admin** 用户执行。如需更多信息，请参阅 CLI 操作主题。

29.2. 基本用法

CLI 中内置了对远程容器命令执行的支持：

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

例如：

```
$ oc exec mypod date
Thu Apr  9 02:21:53 UTC 2015
```

29.3. 协议

客户端通过向 Kubernetes API 服务器发出请求，来发起在容器中执行远程命令的操作：

```
/proxy/minions/<node_name>/exec/<namespace>/<pod>/<container>?command=<command>
```

在以上 URL 中：

- **<node_name>** 是节点的 FQDN。
- **<namespace>** 是目标 pod 的命名空间。
- **<pod>** 是目标 pod 的名称。
- **<container>** 是目标容器的名称。
- **<command>** 是要执行的命令。

例如：

```
/proxy/minions/node123.openshift.com/exec/myns/mypod/mycontainer?command=date
```

另外，客户端也可以在请求中添加参数来指示是否有以下要求：

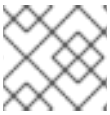
- 客户端应向远程容器的命令发送输入 (stdin)。
- 客户端的终端是 TTY。
- 远程容器的命令应该将来自 stdout 的输出发送到客户端。

- 远程容器的命令应该将来自 stderr 的输出发送到客户端。

在向 API 服务器发送 **exec** 请求后，客户端会将连接升级到支持多路复用的流；当前使用 SPDY。

客户端为 stdin、stdout 和 stderr 分别创建一个流。为了区分流，客户端将流的 **streamType** 标头设置为 **stdin**、**stdout** 或 **stderr** 之一。

在完成远程命令执行请求后，客户端关闭所有流、升级的连接和底层连接。



注意

管理员可以查看 架构指南 以了解更多信息。

第 30 章 将文件复制到容器或从容器中复制

30.1. 概述

您可以使用 CLI 将本地文件复制到容器中的远程目录，或从中复制文件。这是一个非常有用的工具，可将数据库存档复制到 pod 中或从 pod 中复制，以满足备份和恢复需要。当运行的 pod 支持源文件热重新加载时，它也可用于将源代码更改复制到正在运行的 pod 中以进行开发调试。

30.2. 基本用法

CLI 中内置了将本地文件复制到容器或从容器中复制本地文件的支持：

```
$ oc rsync <source> <destination> [-c <container>]
```

例如，将本地目录复制到 pod 目录中：

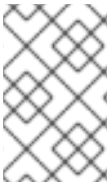
```
$ oc rsync /home/user/source devpod1234:/src
```

或者将 pod 目录复制到本地目录：

```
$ oc rsync devpod1234:/src /home/user/source
```

30.3. 备份和恢复数据库

使用 **oc rsync** 将现有数据库容器的数据库存档复制到新数据库容器的持久性卷目录中。



注意

MySQL 在以下示例中使用。将 **mysql|MYSQL** 替换为 **pgsql|PGSQL** 或 **mongodb|MONGODB**，并参阅 迁移指南 以查找每个支持的数据库镜像的确切命令。该示例假定现有的数据库容器。

1. 从正在运行的数据库 pod 备份现有数据库：

```
$ oc rsh <existing db container>
# mkdir /var/lib/mysql/data/db_archive_dir
# mysqldump --skip-lock-tables -h ${MYSQL_SERVICE_HOST} -P
${MYSQL_SERVICE_PORT:-3306} \
  -u ${MYSQL_USER} --password="$MYSQL_PASSWORD" --all-databases >
/var/lib/mysql/data/db_archive_dir/all.sql
# exit
```

2. 远程将存档文件同步到您的本地机器：

```
$ oc rsync <existing db container with db archive>:/var/lib/mysql/data/db_archive_dir /tmp/.
```

3. 启动第二个 MySQL pod，以便将上面创建的数据库存档文件加载到其中。MySQL pod 必须具有唯一的 **DATABASE_SERVICE_NAME**。

```
$ oc new-app mysql-persistent \
```

```
  -p MYSQL_USER=<archived mysql username> \
  -p MYSQL_PASSWORD=<archived mysql password> \
  -p MYSQL_DATABASE=<archived database name> \
  -p DATABASE_SERVICE_NAME='mysql2' 1
$ oc rsync /tmp/db_archive_dir new_dbpod1234:/var/lib/mysql/data
$ oc rsh new_dbpod1234
```

1 **MySQL** 是默认值。在本例中，已创建 **mysql2**。

4. 使用适当的命令从复制的数据库存档目录中恢复新数据库容器中的数据库：

MySQL

```
$ cd /var/lib/mysql/data/db_archive_dir
$ mysql -u root
$ source all.sql
$ GRANT ALL PRIVILEGES ON <dbname>.* TO '<your username>'@'localhost'; FLUSH
PRIVILEGES;
$ cd ../; rm -rf /var/lib/mysql/data/db_backup_dir
```

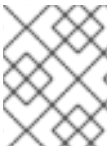
现在，您已使用存档数据库在项目中运行的两个 MySQL 数据库 pod。

30.4. 要求

如果客户端的机器上存在，**oc rsync** 命令会使用本地 **rsync** 命令。这要求远程容器也有 **rsync** 命令。

如果在本地或远程容器中未找到 **rsync**，则会在本地创建 tar 归档并发送到容器，其中将利用 **tar** 来提取文件。如果远程容器中没有 **tar**，则复制将失败。

tar 复制方法不提供与 **rsync** 相同的功能。例如，**rsync** 会创建目标目录（如果不存在），且只会发送源和目的地之间不同的文件。



注意

在 Windows 中，应当安装 **cwRsync** 客户端并添加到 PATH 中，以便与 **oc rsync** 命令搭配使用。

30.5. 指定复制来源

oc rsync 命令的 source 参数必须指向本地目录或 pod 目录。目前不支持单个文件。

指定 pod 目录时，目录名称必须加上 pod 名称前缀：

```
<pod name>:<dir>
```

与标准 **rsync** 一样，如果目录名称以路径分隔符(/)结尾，则只有目录的内容会复制到目的地。否则，目录本身将复制到具有所有内容的目标。

30.6. 指定复制目的地

oc rsync 命令的 destination 参数必须指向某个目录。如果该目录不存在，但使用 **rsync** 进行复制，系统会为您创建这个目录。

30.7. 删除目的地上的文件

可以使用 **--delete** 标志，在远程目录中删除本地目录中没有的文件。

30.8. 在文件更改时持续同步

如果使用 **--watch** 选项，命令可以监控源路径上的任何文件系统更改，并在发生更改时同步它们。使用这个参数时，命令会永久运行。

同步会在短暂的静默期后进行，以确保迅速更改的文件系统不会导致持续的同步调用。

使用 **--watch** 选项时，其行为实际上和手动反复调用 **oc rsync** 一致，通常传递给 **oc rsync** 的所有参数也一样。因此，您可以使用与手动调用 **oc rsync** 时相同的标记来控制其行为，比如 **--delete**。

30.9. 高级 RSYNC 功能

与标准的 **rsync** 相比，**oc rsync** 命令可用的命令行选项比较少。如果您想要使用某个标准 **rsync** 命令行选项，但 **oc rsync** 中没有这个选项（例如，**--exclude-from=FILE** 选项），您可以使用标准 **rsync** 的 **--rsh** (**-e**) 选项或 **RSYNC_RSH** 变量来作为权宜之计，如下所示：

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

或：

```
$ export RSYNC_RSH='oc rsh'
$ rsync --exclude-from=FILE SRC POD:DEST
```

以上两个示例将标准 **rsync** 配置为使用 **oc rsh** 作为远程 shell 程序，从而连接到远程 pod，它们是运行 **oc rsync** 的替代方法。

第 31 章 端口转发

31.1. 概述

OpenShift Container Platform 利用 Kubernetes 内置的功能来支持到 pod 的端口转发。如需更多信息，请参阅 架构。

您可以使用 CLI 将一个或多个本地端口转发到 pod。这样，您可以在本地侦听一个指定或随机端口，并且与 pod 中的指定端口来回转发数据。

31.2. 基本用法

CLI 中内置了端口转发支持：

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

CLI 侦听用户指定的本地端口，并通过以下协议进行转发。

可使用以下格式来指定端口：

5000	客户端在本地侦听端口 5000，并转发到 pod 中的 5000。
6000:5000	客户端在本地侦听端口 6000，并转发到 pod 中的 5000。
:5000 或 **0:5000**	客户端选择本地的一个空闲端口，并转发到 pod 中的 5000 。

例如，要在本地侦听端口 **5000** 和 **6000**，并从 pod 的端口 **5000** 和 **6000** 转发数据，请运行：

```
$ oc port-forward <pod> 5000 6000
```

要在本地侦听端口 **8888** 并转发到 pod 中的 **5000**，请运行：

```
$ oc port-forward <pod> 8888:5000
```

要在本地侦听一个空闲端口并转发到 pod 中的 **5000**，请运行：

```
$ oc port-forward <pod> :5000
```

或者：

```
$ oc port-forward <pod> 0:5000
```

31.3. 协议

客户端通过向 Kubernetes API 服务器发出请求，来发起向 pod 转发端口的操作：

```
/proxy/minions/<node_name>/portForward/<namespace>/<pod>
```

在以上 URL 中：

- **<node_name>** 是节点的 FQDN。
- **<namespace>** 是目标 pod 的命名空间。
- **<pod>** 是目标 pod 的名称。

例如：

```
/proxy/minions/node123.openshift.com/portForward/myns/mypod
```

向 API 服务器发送端口转发请求后，客户端会将连接升级到支持多路复用的流；当前使用 SPDY。

客户端创建 **port** 标头中包含 pod 中目标端口的流。写入流的所有数据都通过 Kubelet 传送到目标 pod 和端口。同样，针对被转发连接从 pod 发送的所有数据都会被传回客户端上的同一流。

在完成端口转发请求后，客户端关闭所有流、升级的连接和底层连接。

注意

管理员可以查看 架构指南 以了解更多信息。

第 32 章 共享内存

32.1. 概述

Linux 中有两个共享内存对象：System V 和 POSIX.pod 中的容器共享 pod 基础架构容器的 IPC 命名空间，因此能够共享 System V 共享内存对象。本文档描述了它们如何共享 POSIX 共享内存对象。

32.2. POSIX 共享内存

POSIX 共享内存要求将 tmpfs 挂载于 ***/dev/shm***。pod 中的容器不共享其挂载命名空间，因此我们使用卷将相同的 ***/dev/shm*** 提供给 pod 中的每个容器。以下示例演示了如何在两个容器之间设置 POSIX 共享内存。

shared-memory.yaml

```
---
apiVersion: v1
id: hello-openshift
kind: Pod
metadata:
  name: hello-openshift
  labels:
    name: hello-openshift
spec:
  volumes:                          1
    - name: dshm
      emptyDir:
        medium: Memory
  containers:
    - image: kubernetes/pause
      name: hello-container1
      ports:
        - containerPort: 8080
          hostPort: 6061
      volumeMounts:                 2
        - mountPath: /dev/shm
          name: dshm
    - image: kubernetes/pause
      name: hello-container2
      ports:
        - containerPort: 8081
          hostPort: 6062
      volumeMounts:                 3
        - mountPath: /dev/shm
          name: dshm
```

1. 指定 tmpfs 卷 **dshm**。
2. 通过 **dshm** 为 **hello-container1** 启用 POSIX 共享内存。
3. 通过 **dshm** 为 **hello-container2** 启用 POSIX 共享内存。

使用 ***shared-memory.yaml*** 文件创建 pod：

```
$ oc create -f shared-memory.yaml
```

第 33 章 应用程序健康状况

33.1. 概述

在软件系统中，组件可能会变得不健康，原因可能源自临时问题（如临时连接丢失）、配置错误或外部依赖项相关问题。OpenShift Container Platform 应用程序具有若干选项来探测和处理不健康的容器。

33.2. 使用探测的容器健康检查

探测（probe）是一种 Kubernetes 操作，它会定期对运行中的容器执行诊断。目前，存在两种类型的探测，各自满足不同的目的：

存活度（Liveness）探测	存活度探测检查在其中配置的容器是否仍然在运行。如果存活度探测失败，kubelet 会终止该容器，这将受到重启策略的影响。通过配置 Pod 配置的 **template.spec.containers.livenessprobe** 节来设置存活度检查。
就绪度（Readiness）探测	就绪度探测(Readiness probe)决定容器是否准备好服务请求。如果某一容器的就绪度探测失败，则端点控制器将确保从所有服务的端点中移除该容器的 IP 地址。就绪度探测也可用于向端点控制器发送信号，即使有容器在运行它也不应从代理接收任何流量。通过配置 Pod 配置的 **template.spec.containers.readinessprobe** 节来设置就绪度检查。

探测的确切时间由两个字段控制，它们以秒为单位表示：

字段	描述
initialDelaySeconds	容器启动后等待多久才能开始探测。
timeoutSeconds	等待探测完成的时间（默认：**1**).如果超过这一时间，OpenShift Container Platform 会将探测视为已失败。

可以通过两种方式配置两个探测：

HTTP 检查

kubelet 使用 Web hook 来确定容器的健康状态。如果 HTTP 响应代码介于 200 和 399 之间，则检查被认定为成功。以下是使用 HTTP 检查方法就绪度检查示例：

例 33.1. 就绪度 HTTP 检查

```
...
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

-

对于在完全初始化后返回 HTTP 状态代码的应用程序，HTTP 检查是理想的选择。

容器执行检查

kubelet 在容器内执行一个命令。退出检查时状态为 0 被视为成功。以下是使用容器执行方法的存活度检查示例：

例 33.2. 存活度容器执行检查

```
...
livenessProbe:
  exec:
    command:
    - cat
    - /tmp/health
  initialDelaySeconds: 15
...
```

注意

timeoutSeconds 参数不影响容器执行检查的就绪度和存活度探测。

注意

timeoutSeconds 参数不影响容器执行检查的就绪度和存活度探测。您可以在探测本身中使用超时机制，因为 OpenShift Container Platform 无法对进入容器的 exec 调用执行超时。在探测中实施超时的一种方法是使用 **timeout** 参数来运行存活度或就绪度探测：

```
[...]
      livenessProbe:
       exec:
         command:
           - /bin/bash
           - '-c'
           - timeout 60 /opt/eap/bin/livenessProbe.sh 1
       timeoutSeconds: 1
       periodSeconds: 10
       successThreshold: 1
       failureThreshold: 3
[...]
```

1 超时值和探测脚本路径。

TCP 套接字检查

kubelet 尝试向容器打开一个套接字。只有检查能够建立连接，容器才被视为健康。以下是使用 TCP 套接字检查方法的存活度检查示例：

例 33.3. 存活度 TCP 套接字检查

```
...
```

```
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

对于只有初始化完毕后才开始侦听的应用程序，TCP 套接字检查是理想的选择。

如需有关健康检查的更多信息，请参阅 Kubernetes 文档。

第 34 章 事件

34.1. 概述

OpenShift Container Platform 中的事件根据 OpenShift Container Platform 集群中 API 对象的事件进行建模。事件允许 OpenShift Container Platform 以无关资源的方式记录实际事件的信息。它们还允许开发人员和管理员以统一的方式消耗系统组件的信息。

34.2. 通过 CLI 查看事件

您可以使用以下命令获取给定项目中的事件列表：

```
$ oc get events [-n <project>]
```

34.3. 在控制台中查看事件

您可以在 web 控制台的 **Browse → Events** 页面查看项目中的事件。pod 和部署等许多其他对象也具有自己的 **Events** 选项卡，其中显示与该对象相关的事件。

34.4. 事件的完整列表

本节介绍 OpenShift Container Platform 的事件。

表 34.1. 配置事件

名称	描述
FailedValidation	pod 配置验证失败。

表 34.2. 容器事件

名称	描述
BackOff	避退重启使容器失败。
Created	已创建容器。
Failed	拉取/创建/启动失败。
Killing	正在终止容器。
Started	容器已启动。
Preempting	正在抢占其他 pod。
ExceededGracePeriod	在指定宽限期内，容器运行时没有停止 pod。

表 34.3. 健康事件

名称	描述
Unhealthy	容器不健康。

表 34.4. 镜像事件

名称	描述
BackOff	避退容器启动，镜像拉取。
ErrImageNeverPull	违反了镜像的 NeverPull 策略。
Failed	拉取镜像失败。
InspectFailed	检查镜像失败。
Pulled	成功拉取了镜像，或容器镜像已存在于机器上。
Pulling	正在拉取镜像。

表 34.5. 镜像管理器事件

名称	描述
FreeDiskSpaceFailed	可用磁盘空间失败。
InvalidDiskCapacity	磁盘容量无效。

表 34.6. 节点事件

名称	描述
FailedMount	卷挂载已失败。
HostNetworkNotSupported	主机网络不受支持。
HostPortConflict	主机/端口冲突。
InsufficientFreeCPU	可用 CPU 不足。

名称	描述
InsufficientFreeMemory	可用内存不足。
KubeletSetupFailed	kubelet 设置失败。
NilShaper	未定义整形器。
NodeNotReady	节点未就绪。
NodeNotSchedulable	节点不可调度。
NodeReady	节点已就绪。
NodeSchedulable	节点可以调度。
NodeSelectorMismatching	节点选择器不匹配。
OutOfDisk	磁盘空间不足。
Rebooted	节点已重启。
Starting	正在启动 kubelet。
FailedAttachVolume	附加卷失败。
FailedDetachVolume	分离卷失败。
VolumeResizeFailed	扩展/缩减卷失败。
VolumeResizeSuccessful	成功扩展/缩减卷。
FileSystemResizeFailed	扩展/缩减文件系统失败。
FileSystemResizeSuccessful	成功扩展/缩减文件系统。
FailedUnMount	卸载卷失败。

名称	描述
FailedMapVolume	映射卷失败。
FailedUnmapDevice	取消映射设备失败。
AlreadyMountedVolume	卷已经挂载。
SuccessfulDetachVolume	卷已被成功分离。
SuccessfulMountVolume	卷已被成功挂载。
SuccessfulUnMountVolume	卷已被成功卸载。
ContainerGCFailed	容器垃圾回收失败。
ImageGCFailed	镜像垃圾回收失败。
FailedNodeAllocatableEnforcement	未能强制实施系统保留的 Cgroup 限制。
NodeAllocatableEnforced	已强制实施系统保留的 Cgroup 限制。
UnsupportedMountOption	不支持的挂载选项。
SandboxChanged	Pod 沙盒已更改。
FailedCreatePodSandBox	未能创建 pod 沙盒。
FailedPodSandBoxStatus	pod 沙盒状态失败。

表 34.7. Pod Worker 事件

名称	描述
FailedSync	Pod 同步失败。

表 34.8. 系统事件

名称	描述
SystemOOM	集群遇到 OOM（内存不足）状况。

表 34.9. Pod 事件

名称	描述
FailedKillPod	停止 pod 失败。
FailedCreatePodContainer	创建 pod 容器失败。
Failed	创建 pod 数据目录失败。
NetworkNotReady	网络未就绪。
FailedCreate	创建时出错：**<error-msg>**。
SuccessfulCreate	已创建 pod：**<pod-name>**。
FailedDelete	删除时出错：**<error-msg>**。
SuccessfulDelete	已删除 pod：**<pod-id>**。

表 34.10. Pod 横向自动扩展事件

名称	描述
SelectorRequired	需要选择器。
InvalidSelector	无法将选择器转换为对应的内部选择器对象。
FailedGetObjectMetric	HPA 无法计算副本数。

名称	描述
InvalidMetricSourceType	未知的指标源类型。
ValidMetricFound	HPA 能够成功计算副本数。
FailedConvertHPA	未能转换给定的 HPA。
FailedGetScale	HPA 控制器无法获取目标的当前规模。
SucceededGetScale	HPA 控制器成功获取了目标的当前规模。
FailedComputeMetricsReplicas	未能根据列出的指标计算所需的副本数。
FailedRescale	新大小：**<size>**；原因：**<msg>**：错误：**<error-msg>**。
SuccessfulRescale	新大小：**<size>**；原因：**<msg>**。
FailedUpdateStatus	未能更新状态。

表 34.11. 网络事件 (openshift-sdn)

名称	描述
Starting	正在启动 OpenShift-SDN。
NetworkFailed	pod 的网络接口已经丢失，pod 也将被停止。

表 34.12. 网络事件 (kube-proxy)

名称	描述
NeedPods	服务端口 **<serviceName>:<port>** 需要 pod。

表 34.13. 卷事件

名称	描述
FailedBinding	没有可用的持久性卷，而且未设置存储类。

名称	描述
VolumeMismatch	卷大小或类与声明中请求的不同。
VolumeFailedRecycle	创建回收 pod 时出错。
VolumeRecycled	回收卷时发生。
RecyclerPod	回收 pod 时发生。
VolumeDelete	删除卷时发生。
VolumeFailedDelete	删除卷时出错。
ExternalProvisioning	在手动或通过外部软件置备声明的卷时发生。
ProvisioningFailed	未能置备卷。
ProvisioningCleanupFailed	清理置备的卷时出错。
ProvisioningSucceeded	在成功置备了卷时发生。
WaitForFirstConsumer	将绑定延迟到 pod 调度为止。

表 34.14. 生命周期 hook

名称	描述
FailedPostStartHook	处理程序因为 pod 启动而失败。
FailedPreStopHook	处理程序因为预停止而失败。
UnfinishedPreStopHook	预停止 hook 未完成。

表 34.15. 部署

名称	描述
DeploymentCancellationFailed	未能取消部署。
DeploymentCancelled	已取消部署。
DeploymentCreated	已创建新的复制控制器。
IngressIPRangeFull	没有可用的入口 IP 要分配给服务。

表 34.16. 调度程序事件

名称	描述
FailedScheduling	未能调度 pod：**<pod-namespace>/<pod-name>**。引发此事件有多个原因，例如：**AssumePodVolumes** 失败，绑定被拒绝等。
Preempted	被节点 **<node-name>** 上的 **<preemptor-namespace>/<preemptor-name>** 抢占。
Scheduled	成功将 **<pod-name>** 分配给 **<node-name>**。

表 34.17. DaemonSet 事件

名称	描述
SelectingAll	此 daemon 选择所有 pod。需要非空选择器。
FailedPlacement	未能将 pod 放置到 **<node-name>**。
FailedDaemonPod	在节点 **<node-name>** 上找到了失败的守护进程 pod**<pod-name>**，会尝试将它终止。

表 34.18. LoadBalancer 服务事件

名称	描述
CreatingLoadBalancerFailed	创建负载均衡器时出错。
DeletingLoadBalancer	正在删除负载均衡器。

名称	描述
EnsuringLoadBalancer	正在确保负载均衡器。
EnsuredLoadBalancer	已确保负载均衡器。
UnAvailableLoadBalancer	没有可用于 **LoadBalancer** 服务的节点。
LoadBalancerSourceRanges	列出新的 **LoadBalancerSourceRanges**。例如，**<old-source-range>** → **<new-source-range>**。
LoadbalancerIP	列出新 IP 地址。例如，**<old-ip>** → **<new-ip>**。
ExternalIP	列出外部 IP 地址。例如，**Added: <external-ip>**。
UID	列出新 UID。例如，**<old-service-uid>** → **<new-service-uid>**。
ExternalTrafficPolicy	列出新 **ExternalTrafficPolicy**。例如，**<old-policy>** → **<new-ploicy>**。
HealthCheckNodePort	列出新 **HealthCheckNodePort**。例如，**<old-node-port>** → **new-node-port>**。
UpdatedLoadBalancer	使用新主机更新负载均衡器。
LoadBalancerUpdateFailed	使用新主机更新负载均衡器时出错。
DeletingLoadBalancer	正在删除负载均衡器。
DeletingLoadBalancerFailed	删除负载均衡器时出错。
DeletedLoadBalancer	已删除负载均衡器。

第 35 章 管理环境变量

35.1. 设置和取消设置环境变量

OpenShift Container Platform 提供 **oc set env** 命令，为具有 pod 模板 的对象设置或取消设置环境变量，如复制控制器或部署配置。它还可以列出 pod 中的环境变量，或列出具有 pod 模板的任何对象。此命令也可以在 **BuildConfig** 对象中使用。

35.2. 列出环境变量

列出 pod 或 pod 模板中的环境变量：

```
$ oc set env <object-selection> --list [<common-options>]
```

本例列出了 pod **p1** 的所有环境变量：

```
$ oc set env pod/p1 --list
```

35.3. 设置环境变量

在 pod 模板中设置环境变量：

```
$ oc set env <object-selection> KEY_1=VAL_1 ... KEY_N=VAL_N [<set-env-options>] [<common-options>]
```

设置环境选项：

选项	描述
-e, --env=*<KEY>=<VAL>*	设置给定环境变量的键值对。
--overwrite	确认更新现有的环境变量。

在以下示例中，两个命令在部署配置 **registry** 中修改环境变量 **STORAGE**。第一个添加，值为 **/data**。第二个更新，值为 **/opt**。

```
$ oc set env dc/registry STORAGE=/data
$ oc set env dc/registry --overwrite STORAGE=/opt
```

以下示例在当前 shell 中找到环境变量，其名称以 **RAILS_** 开头，并将它们添加到服务器上的复制控制器 **r1** 中：

```
$ env | grep ^RAILS_ | oc set env rc/r1 -e -
```

以下示例不会修改文件 **rc.json** 中定义的复制控制器。相反，它会将带有更新的环境 **STORAGE=/local** 的 YAML 对象写入新文件 **rc.yaml**。

```
$ oc set env -f rc.json STORAGE=/opt -o yaml > rc.yaml
```

35.3.1. 自动添加的环境变量

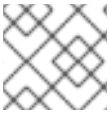
表 35.1. 自动添加的环境变量

变量名称
<SVCNAME>_SERVICE_HOST
<SVCNAME>_SERVICE_PORT

用法示例

服务 **KUBERNETES** 公开 TCP 端口 53，并分配了集群 IP 地址 10.0.0.11 会生成以下环境变量：

```
KUBERNETES_SERVICE_PORT=53
MYSQL_DATABASE=root
KUBERNETES_PORT_53_TCP=tcp://10.0.0.11:53
KUBERNETES_SERVICE_HOST=10.0.0.11
```



注意

使用 **oc rsh** 命令 SSH 到容器，并运行 **oc set env** 来列出所有可用的变量。

35.4. 取消设置环境变量

在 pod 模板中取消设置环境变量：

```
$ oc set env <object-selection> KEY_1- ... KEY_N- [<common-options>]
```

重要

末尾的连字符（**-**、U+2D）是必需的。

这个示例从部署配置 **d1** 中删除环境变量 **ENV1** 和 **ENV2** ：

```
$ oc set env dc/d1 ENV1- ENV2-
```

这会从所有复制控制器中删除环境变量 **ENV**:

```
$ oc set env rc --all ENV-
```

这会从复制控制器 **r1** 的容器 **c1** 中删除环境变量 **ENV**:

```
$ oc set env rc r1 --containers='c1' ENV-
```

第 36 章 JOBS

36.1. 概述

一个作业会运行带有任何副本数量的 pod，这与复制控制器不同。作业会跟踪任务的整体进度，并使用活跃、成功和失败 pod 的相关信息来更新其状态。删除作业会清理它创建的所有 pod 副本。作业是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

如需有关作业的更多信息，请参阅 Kubernetes 文档。

36.2. 创建作业

作业配置由以下关键部分组成：

- pod 模板，用于描述 Pod 将创建的应用程序。
- 可选的 **parallelism** 参数，用于指定并行运行多少个 pod 副本应执行作业。如果没有指定，则默认为 **completions** 参数中的值。
- 可选的 **completions** 参数，用于指定并发运行 pod 应该执行某个作业的数量。若不指定，这个值默认为一。

以下是一个**作业资源**的示例 ：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  template:         (3)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (4)
```

1. 可选值，定义一个作业应并行运行多少个 pod 副本 ; 默认为 **completions**。
2. 可选值，定义需要成功完成多少个 pod 才能将作业标记为完成；默认值为 1。
3. 控制器创建的 pod 模板。
4. pod 的重启策略。这不适用于作业控制器。详情请查看 第 36.2.1 节 “已知限制”。

您还可以使用 **oc run**，在一个命令中创建并启动作业。以下命令会创建并启动与上一示例中指定的相同的作业：

```
$ oc run pi --image=perl --replicas=1  --restart=OnFailure \
    --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

-

36.2.1. 已知限制

作业规格重启策略只适用于 *pod*，不适用于 *作业控制器*。不过，作业控制器被硬编码为可以一直重试直到作业完成为止。

因此，**restartPolicy:Never** 或 **--restart=Never** 会产生与 **restartPolicy 相同的行为：OnFailure** 或 **--restart=OnFailure**。也就是说，作业失败后会自动重启，直到成功（或被手动放弃）为止。策略仅设定由哪一子系统执行重启。

使用 **Never** 策略时，*作业控制器*负责执行重启。在每次尝试时，作业控制器会在作业状态中递增失败次数并创建新的 pod。这意味着，每次尝试失败都会增加 pod 的数量。

使用 **OnFailure** 策略时，*kubelet* 负责执行重启。每次尝试都不会在作业状态中递增失败次数。另外，kubelet 将通过在相同节点上启动 pod 来重试失败的作业。

36.3. 扩展作业

可以使用 **oc scale** 命令和 **--replicas** 选项（在作业案例中）修改 **spec.parallelism** 参数，来缩放作业。这会导致修改并行运行的 pod 副本数，并执行作业。

以下命令使用上面的示例作业，并将 **parallelism** 参数设置为三：

```
$ oc scale job pi --replicas=3
```

注意

扩展复制控制器也使用带有 **--replicas** 选项的 **oc scale** 命令，但改为更改复制控制器配置的 **replicas** 参数。

36.4. 设置最大持续时间

在定义一个**作业**时，您可以通过设置 **activeDeadlineSeconds** 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。

最长持续时间从系统中调度第一个 pod 的时间开始计算，并且定义作业在多久时间内处于活跃状态。它将跟踪执行整体时间，与完成的数量无关（执行任务所需的 pod 副本数）。达到指定的超时后，OpenShift Container Platform 将终止作业。

以下示例显示了一个 **作业** 的一部分，它指定了 **activeDeadlineSeconds** 字段（30 分钟）：

```
spec:
  activeDeadlineSeconds: 1800
```

36.5. 作业恢复失败策略

在因为配置中的逻辑错误或其他类似原因而重试了一定次数后，作业会被视为已经失败。指定作业的重试次数，设置 **.spec.backoffLimit** 属性。此字段默认值为 6。控制器以六分钟为上限，按指数避退延时（**10s**, **20s**, **40s** ...）重新创建与作业关联的失败 Pod。如果控制器检查之间没有出现新的失败 pod，则重置这个限制。

第 37 章 OPENSHIFT PIPELINE

37.1. 概述

OpenShift Pipelines 可让您控制在 OpenShift 中构建、部署和推广您的应用程序。通过结合使用 Jenkins Pipeline 构建策略、Jenkinsfile 和 OpenShift 域特定语言(DSL)（由 OpenShift 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。

37.2. OPENSHIFT JENKINS 客户端插件

OpenShift 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 DSL。使用 OpenShift Jenkins 镜像时，默认安装并启用此插件。

有关安装和配置此插件的更多信息，请参阅配置 Pipeline 执行。

37.2.1. OpenShift DSL

OpenShift Jenkins 客户端插件提供了一个流畅的 DSL，用于从 Jenkins 从中与 OpenShift API 通信。OpenShift DSL 基于 Groovy 语法，提供了控制应用生命周期的方法，如创建、构建、部署和删除。

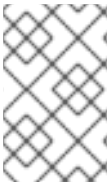
API 的完整详情被嵌入到插件的在线文档中，存储在运行的 Jenkins 实例中。查找它：

- 创建一个新的 Pipeline Item。
- 点 DSL 文本区域下的 **Pipeline 语法**。
- 在左侧导航菜单中点击 **Global Variables Reference**。

37.3. JENKINS PIPELINE 策略

为了利用项目中的 OpenShift Pipelines，您必须使用 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，该位置相对于源 **contextDir**。



注意

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

有关 Jenkins Pipeline 策略的详细信息，请参阅管道策略选项。

37.4. JENKINSFILE

jenkinsfile 采用标准的 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。

jenkinsfile 可以通过以下方法之一来提供：

- 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 **JenkinsPiplineStrategy** 部分的 **jenkinsfilePath** 字段指定的文件名，如果提供，则相对于源 **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。

37.5. 教程

有关使用 Jenkins 管道构建和部署应用的完整步骤，请参阅 Jenkins 管道教程。

37.6. 高级主题

37.6.1. 禁用 Jenkins AutoProvisioning

创建 Pipeline 构建配置时，OpenShift 会检查是否目前在当前项目中置备了 Jenkins master pod。如果没有找到 Jenkins master，则自动创建 master。如果这种行为不可取，或者您希望使用 OpenShift 外部的 Jenkins 服务器，可以禁用它。

如需更多信息请参阅配置管道执行。

37.6.2. 配置 Slave Pod

Kubernetes 插件 也预安装在官方 Jenkins 镜像中。此插件允许 Jenkins 主控机在 OpenShift 上创建从属 Pod，并将正在运行的作业委派给它们以实现可扩展性，并为 Pod 提供特定作业的特定运行时。

有关使用 Kubernetes 插件配置从属 pod 的详情，请参考 Kubernetes 插件。

第 38 章 CRON JOBS

38.1. 概述

cron job 基于常规作业构建，允许您特定调度作业的运行方式。Cron Job 是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

警告

Cron Job 大致会在每个计划的时间创建一个作业对象，但在有些情况下，它可能无法创建作业，或者可能会创建两个作业。因此，作业必须具有幂等性，而且 您必须配置历史限制。

38.2. 创建 CRON JOB

以下是 **CronJob** 资源的示例：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/1 * * * *"                    1
  concurrencyPolicy: "Replace"               2
  startingDeadlineSeconds: 200               3
  suspend: false                             4
  successfulJobsHistoryLimit: 3              5
  failedJobsHistoryLimit: 1                  6
  jobTemplate:                               7
    spec:
      template:
        metadata:
          labels:                            8
            parent: "cronjobpi"
        spec:
          containers:
          - name: pi
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: OnFailure           9
```

1 1 作业的 schedule 字段在 cron format 中指定。在本例中，作业会每分钟运行一次。

2 2 可选：concurrency 策略指定作业控制器如何处理 Cron Job 中的并发作业。只能指定以下策略之一：

- **Allow** 允许多个作业实例同时运行。**Allow** 是默认值。

- **Forbid** 会阻止多个作业实例同时运行。如果上一运行没有完成，则会跳过调度运行。
- **Replace** 会取消当前运行的作业实例，并使用一个新实例替换当前的作业。

3 3 可选：如果因任何原因而错过调度的运行，则起始截止时间（以秒为单位）。截止时间后，cron 任务不会启动该作业。错过的作业运行计为失败的作业。默认情况下，作业没有期限。请注意，当设置此字段时，cron 作业控制器会统计截止时间和当前时间的值间隔内发生的缺失的作业数量。例如，如果 **startDeadlineSeconds** 设为 **200**，则控制器会在最后 200 秒内统计错过的作业数量。如需更多信息，请参阅 Kubernetes 文档中的 cron 任务 概念的限制。

4 4 可选：suspend 字段用于防止后续运行 cron 作业。如果设置为 **true**，则后续所有运行都将阻止启动。默认情况下，值为 **false**，作业则运行。

5 可选：成功的作业历史记录限制指定要保留的已完成作业数量。默认情况下，保留三个作业。

6 可选：失败的作业历史记录限制指定要保留的失败作业数量。默认情况下保留一个作业。

7 作业模板指定要运行的作业。该字段类似于 作业示例。

8 可选：labels 字段指定为 cron 作业启动的作业设置的标签。在本例中，作业会接收标签 **parent=cronjobpi**。

9 可选：启动运行作业的 pod 的 重启策略。策略可以设置为 **Always**、**OnFailure** 或 **Never**。默认情况下，容器总是被重启。请注意，此字段不适用于作业控制器。详情请参阅已知的限制。

有关 **CronJob** 规格的更多信息，请参阅 Kubernetes 文档 来编写 cron 任务规格。

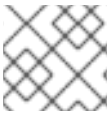
注意

所有 cron 任务**调度**时间均基于启动该作业的 master 的时区。

您还可以使用 **oc run** 在一个命令中创建并启动 cron 任务。以下命令会创建并启动与上一示例中指定的相同的 cron 任务：

```
$ oc run pi --image=perl --schedule='*/1 * * * *' \
    --restart=OnFailure --labels parent="cronjobpi" \
    --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

使用 **oc run** 时，**--schedule** 选项接受采用 cron 格式的调度计划。



注意

在创建 cron 任务时，**oc run** 仅支持 **Never** 或 **OnFailure** 重启策略(**--restart**)。

提示

删除您不再需要的 Cron Job：

```
$ oc delete cronjob/<cron_job_name>
```

这样可防止生成不必要的工件。

38.3. 在 CRON JOB 后清除

.spec.successfulJobsHistoryLimit 和 **.spec.failedJobsHistoryLimit** 字段是可选的，这些字段指定保留多少个完成和失败的作业。默认情况下，分别设置为 **3** 和 **1**。如果将限制设定为 **0**，则对应种类的作业完成后不予保留。

Cron Job 可能会遗留工件资源，如作业和 pod 等。作为用户，务必要配置历史限制，以便正确清理旧作业及其 pod。目前，cron 作业的 spec 中有两个字段负责这一事务：

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  successfulJobsHistoryLimit: 3 (1)
  failedJobsHistoryLimit: 1 (2)
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
  ...
```

1. 要保留的成功完成作业数量。默认值为 **3**。
2. 要保留的失败完成作业数量。默认值为：**1**。

第 39 章 从 URL 创建

39.1. 概述

从 URL 创建是一个功能，您可以从镜像流、镜像标签或模板中构造 URL。

从 URL 创建只适用于明确白名单的命名空间中的镜像流或模板。白名单默认包含 **openshift** 命名空间。要在白名单中添加命名空间，请参阅配置 Create From URL Namespace Whitelist 。

您可以定义自定义按钮。

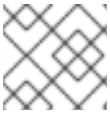
这些按钮利用带有适当的查询字符串的定义的 URL 模式。系统将提示用户选择该项目。然后，Create from URL 工作流将继续。

39.2. 使用镜像流和镜像标签

39.2.1. 查询字符串参数

名称	描述	必填	模式	默认
imageStream	要使用的镜像流中定义的 **metadata.name** 值。	true	字符串	
imageTag	要使用的镜像流中定义的 **spec.tags.name** 值。	true	字符串	
namespace	包含要使用的镜像流和镜像标签的命名空间名称。	false	字符串	**openshift**
name	标识为此应用程序创建的资源。	false	字符串	
sourceURI	包含应用源代码的 Git 存储库 URL。	false	字符串	
sourceRef	**sourceURI** 中指定的应用源代码的分支、标签或提交。	false	字符串	

名称	描述	必填	模式	默认
contextDir	**sourceURI** 中指定的应用源代码的子目录，用作构建的上下文目录。	false	字符串	



注意

参数值中的保留字符应为 URL 编码。

39.2.1.1. 示例

```
 create?
imageStream=nodejs&imageTag=4&name=nodejs&sourceURI=https%3A%2F%2Fgithub.com%2Fopenshift%2Fnodejs-ex.git&sourceRef=master&contextDir=%2F
```

39.3. 使用模板

39.3.1. 查询字符串参数

名称	描述	必填	模式	默认
模板	要使用的模板中定义的 **metadata.name** 值。	true	字符串	
templateParamsMap	包含模板参数名称和要覆盖的对应值的 JSON 参数映射。	false	JSON	
namespace	包含要使用的模板的命名空间名称。	false	字符串	**openshift**

注意

参数值中的保留字符应为 URL 编码。

39.3.1.1. 示例

```
 create?template=nodejs-mongodb-example&templateParamsMap=
{"SOURCE_REPOSITORY_URL"%3A"https%3A%2F%2Fgithub.com%2Fopenshift%2Fnodejs-ex.git"}
```

第 40 章 从自定义资源定义创建对象

40.1. KUBERNETES 自定义资源定义

在 Kubernetes API 中，资源是存储某一类 API 对象集合的端点。例如，内置 pod 资源包含一组 Pod 对象。

自定义资源 是一个扩展 Kubernetes API 的对象，或允许您在项目或集群中引入自己的 API。

自定义资源定义 (CRD)文件定义了自己的对象类型，并允许 API 服务器处理整个生命周期。

注意

虽然只有集群管理员可创建 CRD，但如果您具有读写权限，则可以从 CRD 创建对象。

40.2. 从 CRD 创建自定义对象

自定义对象可以包含包含任意 JSON 代码的自定义字段。

先决条件

- 创建 CRD。

流程

1. 为自定义对象创建 YAML 定义。在以下示例中，**cronSpec** 和 **镜像自定义** 字段在类型为 **CronTab** 的自定义对象中设置。kind 来自自定义资源定义对象的 **spec.kind** 字段。

   自定义对象的 YAML 文件示例

   ```
   apiVersion: "stable.example.com/v1" 1
   kind: CronTab 2
   metadata:
     name: my-new-cron-object 3
     finalizers: 4
     - finalizer.stable.example.com
   spec: 5
     cronSpec: "* * * * /5"
     image: my-awesome-cron-image
   ```

   1 指定自定义资源定义中的组名和 API 版本（名称/版本）。

   2 指定自定义资源定义中的类型。

   3 指定对象的名称。

   4 指定对象的结束程序（如有）。结束程序可让控制器实现在删除对象之前必须完成的条件。

   5 指定特定于对象类型的条件。

2. 创建对象文件后，创建对象：

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

40.3. 管理自定义对象

创建对象后，您可以管理自定义资源。

先决条件

- 创建自定义资源定义(CRD)。
- 从 CRD 创建对象。

流程

1. 要获取有关特定类型的自定义资源的信息，请输入：

```
oc get <kind>
```

例如：

```
oc get crontab

NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

请注意，资源名称不区分大小写，您可以使用 CRD 中定义的单数或复数形式，以及任何短名称。例如：

```
oc get crontabs
oc get crontab
oc get ct
```

2. 您还可以查看自定义资源的原始 YAML 数据：

```
oc get <kind> -o yaml
```

```
oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * /5' 1
    image: my-awesome-cron-image 2
```

-

1 2 显示用于创建对象的 YAML 的自定义数据。

第 41 章 应用程序内存大小调整

41.1. 概述

本页旨在提供有关使用 OpenShift Container Platform 的应用程序开发人员的指导信息：

1. 确定容器化应用程序组件的内存和风险要求，并配置容器内存参数以满足这些要求。
2. 配置容器化应用程序运行时（如 OpenJDK），以最佳的方式遵守配置的容器内存参数。
3. 诊断并解决与在容器中运行相关的内存错误情况。

41.2. 背景信息

在继续操作前，建议您先通篇阅读有关 OpenShift Container Platform 如何管理计算资源的概述。

对于调整应用程序内存的大小，关键点是：

- 对于每种种类的资源（内存、cpu 和存储），OpenShift Container Platform 允许将可选**请求**和**限制**值放在 pod 中的每个容器上。就此页面而言，我们只对内存请求和内存限值感兴趣。
- **内存请求**
  - 如果指定，内存请求值会影响 OpenShift Container Platform 调度程序。将容器调度到节点时，调度程序会考虑内存请求，然后在所选节点上隔离出请求的内存供该容器使用。
  - 如果节点的内存已用尽，OpenShift Container Platform 将优先驱除其内存用量超出内存请求最多的容器。在严重的内存耗尽情形中，节点 OOM 终止程序可以根据类似的指标选择并终止容器中的一个进程。
- **内存限制**
  - 如果指定，内存限制值针对可在容器中所有进程间分配的内存提供硬性限制。
  - 如果分配给容器中所有进程的内存超过内存限制，则节点 OOM 终止程序将立即选择并终止容器中的一个进程。
  - 如果同时指定了内存请求和限制，则内存限制必须大于或等于内存请求量。
- **管理**
  - 集群管理员可以根据内存请求值、限制值、两者或两者分配配额。
  - 集群管理员可以为内存请求值、限制值、两者或两者都分配默认值。
  - 集群管理员可以覆盖开发者指定的内存请求值，以便管理集群过量使用。例如，这在 OpenShift Online 上发生。

41.3. 策略

如下是 OpenShift Container Platform 上调整应用程序内存大小的步骤：

1. **确定预期的容器内存用量**
   从经验判断（例如，通过独立的负载测试），根据需要确定容器内存用量的预期平均值和峰值。需要考虑容器中有可能并行运行的所有进程：例如，主应用程序是否生成任何辅助脚本？

2. **确定风险嗜好**
   确定用于驱除的风险嗜好。如果风险嗜好较低，则容器应根据预期的峰值用量加上一个安全裕度百分比来请求内存。如果风险嗜好较高，那么根据预期的平均用量请求内存可能更为妥当。

3. **设定容器内存请求**
   根据以上所述设定容器内存请求。请求越能准确表示应用程序内存用量越好。如果请求过高，集群和配额用量效率低下。如果请求过低，应用程序驱除的几率就会提高。

4. **根据需要设定容器内存限制**
   在必要时，设定容器内存限制。如果容器中所有进程的总内存用量超过限制，那么设置限制会立即终止容器进程，所以这既有利也有弊。一方面，可能会导致过早出现意料之外的过量内存使用（"快速失败"）；另一方面，也会突然终止进程。

   需要注意的是，有些 OpenShift Container Platform 集群可能要求设置限制；有些集群可能会根据限制覆盖请求；而且有些应用程序镜像会依赖于设置的限制，因为这比请求值更容易检测。

   如果设置内存限制，其大小不应小于预期峰值容器内存用量加上安全裕度百分比。

5. **确保应用程序经过性能优化**
   在适当时，确保应用程序已根据配置的请求和限制进行了性能优化。对于池化内存的应用程序（如 JVM），这一步尤为相关。本页的其余部分将介绍这方面的内容。

41.4. 在 OPENSHIFT CONTAINER PLATFORM 上调整 OPENJDK 大小

遗憾的是，默认的 OpenJDK 设置无法在容器化环境中正常工作，因此每当在容器中运行 OpenJDK 时，一些额外的 Java 内存设置必须始终提供。

JVM 内存布局比较复杂，并且视版本而异，因此本文不做详细讨论。但作为在容器中运行 OpenJDK 的起点，至少以下三个于内存相关的任务非常重要：

1. 覆盖 JVM 最大堆大小。

2. 在可能的情况下，促使 JVM 向操作系统释放未使用的内存。

3. 确保正确配置了容器中的所有 JVM 进程。

优化容器中运行的 JVM 工作负载已超出本文讨论范畴，并且可能涉及设置多个额外的 JVM 选项。

41.4.1. 覆盖 JVM 最大堆大小

对于许多 Java 工作负载，JVM 堆是最大的内存用户。目前，OpenJDK 默认允许将计算节点最多 1/4 (1/**-XX:MaxRAMFraction**) 的内存用于该堆，不论 OpenJDK 是否在容器内运行。因此，务**必要**覆盖此行为，特别是设置了容器内存限制时。

达成以上目标至少有两种方式：

1. 如果设置了容器内存限制，并且 JVM 支持那些实验性选项，请设置 **-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap**。
   这会将 **-XX:MaxRAM** 设置为容器内存限制，并将最大堆大小 (**-XX:MaxHeapSize** / **-Xmx**) 设置为 1/**-XX:MaxRAMFraction**（默认为 1/4）。

2. 直接覆盖 **-XX:MaxRAM**、**-XX:MaxHeapSize** 或 **-Xmx**。
   这个选项涉及对值进行硬编码，但也有允许计算安全裕度的好处。

41.4.2. 把 JVM 更新到操作系统

默认情况下，OpenJDK 不会主动向操作系统退还未用的内存。这可能适合许多容器化的 Java 工作负载，但也有明显的例外，例如额外活跃进程与容器内 JVM 共存的工作负载，这些额外进程是原生或附加的 JVM，或者这两者的组合。

OpenShift Container Platform Jenkins maven slave 镜像 使用以下 JVM 参数鼓励 JVM 向操作系统释放未用的内存：**-XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight=90**。这些参数旨在当分配的内存超过 110% 使用中内存时 (**-XX:MaxHeapFreeRatio**) 将堆内存返还给操作系统，这将在垃圾回收器上最多花费 20% 的 CPU 时间 (**-XX:GCTimeRatio**)。应用程序堆分配一定不会小于初始堆分配（被 **-XX:InitialHeapSize** / **-Xms** 覆盖）。调节 Java 在 OpenShift 中的内存占用（第 1 部分） 、调节 Java 在 OpenShift 中的内存占用（第 2 部分）以及 OpenJDK 和容器提供了其他的详细信息。

41.4.3. 确保在容器中正确配置所有 JVM 进程

如果多个 JVM 在同一容器中运行，则必须保证它们的配置都正确无误。如果有许多工作负载，需要为每个 JVM 分配一个内存预算百分比，留出较大的额外安全裕度。

许多 Java 工具使用不同的环境变量（**JAVA_OPTS**、**GRADLE_OPTS** 和 **MAVEN_OPTS** 等）来配置它们的 JVM，或许难以确保将正确的设置传递给正确的 JVM。

OpenJDK 始终尊重 **JAVA_TOOL_OPTIONS** 环境变量，在 **JAVA_TOOL_OPTIONS** 中指定的值会被 JVM 命令行中指定的其他选项覆盖。默认情况下，OpenShift Container Platform Jenkins maven slave image 设置 **JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"**，确保在默认情况下，这些选项会为在 slave 镜像中运行的所有 JVM 工作负载使用。这不能保证不需要额外选项，只是用作一个实用的起点。

41.5. 从 POD 中查找内存请求和限制

希望从 pod 中动态发现内存请求和限制的应用程序应该使用 Downward API。下列代码片段演示了如何完成此操作。

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: test
    image: fedora:latest
    command:
    - sleep
    - "3600"
    env:
    - name: MEMORY_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: requests.memory
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: limits.memory
    resources:
```

```
    requests:
      memory: 384Mi
    limits:
      memory: 512Mi
```

```
# oc rsh test
$ env | grep MEMORY | sort
MEMORY_LIMIT=536870912
MEMORY_REQUEST=402653184
```

内存限制值也可由 **/sys/fs/cgroup/memory/memory.limit_in_bytes** 文件从容器内部读取。

41.6. 诊断 OOM KILL

如果容器中所有进程的内存总用量超过内存限制，或者在严重的节点内存耗尽情形下，OpenShift Container Platform 可以终止容器中的某个进程。

当一个进程被 OOM 终止时，这有可能会造成容器立即退出，但也不一定。如果容器 PID 1 进程收到 **SIGKILL**，则容器会立即退出。否则，容器行为将取决于其他进程的行为。

如果容器没有立即退出，则能够检测到 OOM 终止，如下所示：

1. 使用代码 137 退出的容器进程，表示它收到了 SIGKILL 信号
2. **/sys/fs/cgroup/memory/memory.oom_control** 中的 oom_kill 计数器递增

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 0
$ sed -e '' </dev/zero  # provoke an OOM kill
Killed
$ echo $?
137
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 1
```

如果 pod 中的一个或多个进程遭遇 OOM 终止，那么当 pod 随后退出时（不论是否立即发生），它都将会具有原因为 **OOMKilled** 的 **Failed** 阶段。OOM 终止的 pod 可以根据 **restartPolicy** 的值来重新启动。如果不重启，ReplicationController 等控制器会看到 pod 的失败状态，并创建一个新 pod 来取代旧 pod。

如果不重启，pod 状态如下：

```
$ oc get pod test
NAME      READY     STATUS      RESTARTS   AGE
test      0/1       OOMKilled   0          1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: false
    restartCount: 0
    state:
      terminated:
```

```
      exitCode: 137
      reason: OOMKilled
  phase: Failed
```

如果重启，其状态如下：

```
$ oc get pod test
NAME      READY     STATUS    RESTARTS   AGE
test      1/1       Running   1          1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
    lastState:
      terminated:
        exitCode: 137
        reason: OOMKilled
    state:
      running:
  phase: Running
```

41.7. 诊断被驱除的 POD

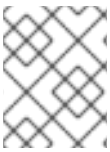
OpenShift Container Platform 可在节点内存耗尽时从节点上驱除 pod。根据内存耗尽的程度，驱除可能是安全操作，但也不一定。安全驱除表示，每个容器的主进程(PID 1)收到 SIGTERM 信号，在进程尚未退出的情况下，一段时间后会有一个 SIGKILL 信号。非安全驱除暗示着各个容器的主进程会立即收到 SIGKILL 信号。

被驱除的 pod 将具有原因为 **Evicted** 的 **Failed** 阶段。无论 **restartPolicy** 的值是什么，该 pod 都不会重启。但是，ReplicationController 等控制器会看到 pod 的失败状态，并且创建一个新 pod 来取代旧 pod。

```
$ oc get pod test
NAME      READY     STATUS    RESTARTS   AGE
test      0/1       Evicted   0          1m

$ oc get pod test -o yaml
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
  phase: Failed
  reason: Evicted
```

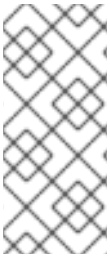
第 42 章 应用程序临时存储大小

42.1. 概述



注意

本节只有在您启用了临时存储技术预览功能时才适用。此功能默认为禁用。要启用此功能，请参阅为临时存储配置。



注意

红帽产品服务等级协议(SLA)不支持技术预览版本，且其功能可能并不完善，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。如需更多信息，请参阅链接：
https://access.redhat.com/support/offerings/techpreview/ [Red Hat 技术预览功能支持 Scope]。

您可以使用临时存储来：

- 确定容器化应用组件的临时存储和风险要求，并配置容器临时存储参数以满足这些要求。
- 配置容器化应用程序运行时（如 OpenJDK），以遵循配置的容器临时存储参数。
- 诊断和解决与在容器中运行存储关联的临时存储相关的错误条件。

42.2. 背景信息

注意

在使用临时存储前，请参阅 OpenShift Container Platform 如何使用 计算资源。

为了调整应用程序临时存储大小，关键点是：

- 对于每种资源（包括内存、CPU、存储和临时存储），OpenShift Container Platform 允许将可选**请求**和**限制**值放在 pod 中的每个容器上。

  临时存储请求

- 如果指定，临时存储请求值会影响 OpenShift Container Platform 调度程序。当将容器调度到节点时，调度程序会考虑临时存储请求，然后在所选节点上隔离出请求的临时存储以供使用容器。

  临时存储限制

- 如果指定，临时存储限制值会为可在容器中所有进程间分配的临时存储提供硬性限制。
- 如果同时指定了临时存储请求和限制，则临时存储限制值必须大于或等于临时存储请求。

  管理

- 集群管理员可以根据临时存储请求值、限制值、两者或两者分配配额。
- 集群管理员可以为临时存储请求值、限制值、或两者都分配默认值。

- 集群管理员可以覆盖开发人员指定的临时存储请求值，以便管理集群过量使用。例如，这在 OpenShift Online 上发生。

42.3. 策略

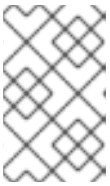
在 OpenShift Container Platform 上调整应用程序临时存储的大小：

1. 确定预期的容器临时用法。
   如果管理员启用了临时存储技术预览功能，请确定预期的平均和峰值容器临时存储使用情况（如有必要），例如通过独立的扩展。请记住，考虑容器中可能并行运行的所有进程。例如，主应用程序是否生成任何可能需要本地存储进行工作文件或日志记录的辅助脚本？

2. 评估驱除风险。
   确定用于驱除的风险 appetite。如果风险 appetite 较低，则根据预期的峰值用量加上一个安全边缘百分比来请求临时存储。如果风险 appetite 较高，则根据预期的平均用量将容器设置为请求临时存储。

3. 设置容器临时存储请求。
   根据您的风险评估设置容器临时存储请求。更加准确地是请求代表应用程序临时存储的使用，其更好。如果请求过高，集群和配额用量效率低下。如果请求过低，应用程序驱除的几率就会提高。

4. 根据需要，设置容器临时存储限值。
   根据需要，设置容器临时存储限值。如果容器中所有进程的组合临时存储使用情况超过限制，则设置限制会立即停止容器进程。例如，容器可能会在早期出现意外的过量临时存储使用，即快速失败，或者容器可能会立即停止进程。

   注意

   有些 OpenShift Container Platform 集群可能需要设置限制值；有些集群可能会根据限制覆盖请求；有些应用程序镜像依赖于设置的限制，因为这比请求值更容易检测。

   如果设置了这些限制，则不应小于预期的峰值容器资源使用情况加上安全边缘百分比。

5. 调优应用程序。
   如果适用，确保应用程序已根据配置的请求和限制进行了调整。此步骤与池临时存储的应用特别相关。

42.4. 诊断被驱除的 POD

当节点的临时存储被耗尽时，OpenShift Container Platform 可能会从其节点中驱除 pod。根据临时存储耗尽的程度，驱除可能是恰当的。在恰当的驱除中，每个容器的主进程 PID 1 会收到 SIGTERM 信号，之后会收到一个 SIGKILL 信号（如果进程仍在运行）。在非正常驱除中，每个容器的主要进程都会立即收到 SIGKILL 信号。

要获取所有 pod 的列表，以便您可以查看其状态：

```
$ oc get pod test
NAME      READY     STATUS    RESTARTS   AGE
test      0/1       Evicted   0          1m

$ oc get pod test -o yaml
...
```

```
status:
  message: 'Pod The node was low on resource: [DiskPressure].'
  phase: Failed
  reason: Evicted
```

被驱除的 pod 具有 **Failed** 阶段，原因为 **Evicted**。被驱除的 pod 不会重启，无论 **restartPolicy** 的值是什么。但是，ReplicationController 等控制器会看到 pod 的失败状态，并创建一个新 pod 来取代旧 pod。