







## Join a community dedicated to learning open source

The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.



**Network** with tens of thousands of community members



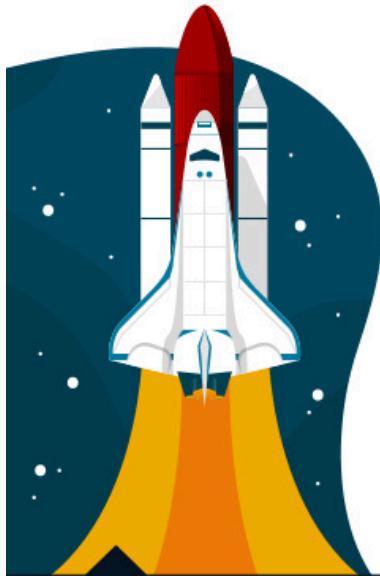
**Engage** in thousands of active conversations and posts



**Join and interact** with hundreds of certified training instructors



**Unlock** badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

**Access** free Red Hat training videos

**Discover** the latest Red Hat Training and Certification news

**Connect** with your instructor - and your classmates - before, after, and during your training course.

**Join** peers as you explore Red Hat products

Join the conversation [learn.redhat.com](https://learn.redhat.com)



Copyright © 2020 Red Hat, Inc. Red Hat, Red Hat Enterprise Linux, the Red Hat logo, and Ansible are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.



# 红帽 OpenShift 开发人员二：构建 Kubernetes 应用



**OCP 4.10 DO288**  
**红帽 OpenShift 开发人员二：构建 Kubernetes 应用**  
**版 4 20220915**  
**出版日期 20220915**

作者: Zach Guterman, Richard Allred, Ricardo Jun,  
Ravishankar Srinivasan, Fernando Lozano, Ivan Chavero,  
Dan Kolepp, Jordi Sola Alaball, Manuel Aude Morales,  
Eduardo Ramírez Martínez, Guy Bianco, Randy Thomas,  
Marek Czernek, Michael Jarrett, Alex Corcoles, Wasim Raja,  
Sourabh Mishra, Christopher Caillouet  
编辑: Sam Ffrench, David O'Brien, Seth Kenlon, Julian Cable

Copyright © 2022 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are  
Copyright © 2022 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but  
not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of  
Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat,  
Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details  
contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send  
email to [training@redhat.com](mailto:training@redhat.com) or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

Red Hat, Red Hat Enterprise Linux, the Red Hat logo, JBoss, OpenShift, Fedora, Hibernate, Ansible, CloudForms, RHCA,  
RHCE, RHCSA, Ceph, and Gluster are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the  
United States and other countries.

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

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

XFS® is a registered trademark of Hewlett Packard Enterprise Development LP or its subsidiaries in the United States  
and/or other countries.

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

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

The OpenStack word mark and the Square O Design, together or apart, are trademarks or registered trademarks of  
OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's  
permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the  
OpenStack community.

All other trademarks are the property of their respective owners.

供稿人: Grega Bremec、Sajith Sugathan、Dave Sacco、Rob Locke、Bowe Strickland、Rudolf  
Kastl、Chris Tusa、Joel Birchler、Chetan Tiwary

<b>文档规范</b>	<b>ix</b>
<b>简介</b>	<b>xi</b>
红帽 OpenShift 开发二：构建 Kubernetes 应用 .....	xi
课堂环境介绍 .....	xii
控制您的系统 .....	xiv
<b>1. 在 OpenShift 集群中部署和管理应用</b>	<b>1</b>
红帽 OpenShift 容器平台 4 简介 .....	2
小测验：OpenShift 4 简介 .....	6
指导练习：配置课堂环境 .....	8
在 OpenShift 集群中部署应用 .....	16
指导练习：在 OpenShift 集群中部署应用 .....	24
使用 Web 控制台管理应用 .....	31
指导练习：使用 Web 控制台管理应用 .....	36
使用 CLI 管理应用 .....	42
指导练习：使用 CLI 管理应用 .....	46
开放研究实验：在 OpenShift 集群中部署和管理应用 .....	55
总结 .....	62
<b>2. 针对 OpenShift 设计容器化应用</b>	<b>63</b>
选择容器化方法 .....	64
小测验：选择容器化方法 .....	67
使用高级 Containerfile 指令构建容器镜像 .....	71
指导练习：使用高级 Containerfile 指令构建容器镜像 .....	78
将配置数据注入应用 .....	86
指导练习：将配置数据注入应用 .....	93
开放研究实验：针对 OpenShift 设计容器化应用 .....	100
总结 .....	110
<b>3. 发布企业容器镜像</b>	<b>111</b>
管理企业注册表中的镜像 .....	112
指导练习：使用企业注册表 .....	119
允许访问 OpenShift 注册表 .....	123
指导练习：允许访问 OpenShift 注册表 .....	126
创建镜像流 .....	130
指导练习：创建镜像流 .....	136
开放研究实验：发布企业容器镜像 .....	140
总结 .....	147
<b>4. 在 OpenShift 上管理构建</b>	<b>149</b>
介绍红帽 OpenShift 构建流程 .....	150
小测验：OpenShift 构建流程 .....	154
管理应用构建 .....	160
指导练习：管理应用构建 .....	163
触发构建 .....	169
指导练习：触发构建 .....	171
实施 Post-commit 构建 hook .....	176
指导练习：实施 Post-Commit 构建 hook .....	178
开放研究实验：面向 OpenShift 构建应用 .....	183
总结 .....	190
<b>5. 自定义源至镜像构建</b>	<b>191</b>
介绍源至镜像架构 .....	192
小测验：介绍源至镜像架构 .....	198
自定义现有 S2I 基础镜像 .....	202
指导练习：自定义 S2I 构建 .....	205
创建 S2I 基础镜像 .....	210

指导练习: 创建 S2I 基础镜像 .....	215
开放研究实验: 自定义源至镜像构建 .....	225
总结 .....	237
<b>6. 部署多容器应用</b>	<b>239</b>
描述 OpenShift 模板 .....	240
小测验: 描述 OpenShift 模板 .....	244
创建 Helm 图表 .....	246
指导练习: 创建 Helm 图表 .....	250
使用 Kustomize 自定义部署 .....	255
指导练习: 使用 Kustomize 自定义部署 .....	258
开放研究实验: 部署多容器应用 .....	264
总结 .....	276
<b>7. 管理应用部署</b>	<b>277</b>
监控应用健康 .....	278
指导练习: 激活探测 .....	283
选择适合的部署策略 .....	290
指导练习: 实施部署策略 .....	294
使用 CLI 命令管理应用部署 .....	301
指导练习: 管理应用部署 .....	306
开放研究实验: 管理应用部署 .....	313
总结 .....	323
<b>8. 面向 OpenShift 构建应用</b>	<b>325</b>
集成外部服务 .....	326
指导练习: 集成外部服务 .....	328
容器化服务 .....	332
指导练习: 将 Nexus 作为服务容器化 .....	339
使用 JKube 部署云原生应用 .....	347
指导练习: 使用 JKube 部署应用 .....	352
开放研究实验: 为 OpenShift 构建云原生应用 .....	362
总结 .....	370
<b>9. 总复习: 红帽 OpenShift 开发人员二: 构建 Kubernetes 应用</b>	<b>371</b>
总复习 .....	372
开放研究实验: 总复习 .....	374
<b>A. 创建 GitHub 帐户</b>	<b>385</b>
创建 GitHub 帐户 .....	386
<b>B. 创建 Quay 帐户</b>	<b>389</b>
创建 Quay 帐户 .....	390
<b>C. 实用的 Git 命令</b>	<b>393</b>
Git 命令 .....	394

# 文档规范



## 参考文献

“引用”部分介绍了查找与主题相关的外部文档的位置。



## 注意

“注意”包括针对执行手头任务的提示、快捷方式或者替代方法。忽略注意事项对结果不会有负面影响，但您可能错失某个可简化操作的技巧。



## 重要

“重要信息”框详细介绍容易忽略的内容：仅适用于当前会话的配置更改，或在应用更新前需要重新启动的服务。忽略标有“重要信息”的框不会导致数据丢失，但可能引起不便和困扰。



## 警告

不能忽略“警告”。忽略警告将很可能导致数据丢失。



# 简介

## 红帽 OpenShift 开发二：构建 Kubernetes 应用

基于容器技术和 Kubernetes 的红帽 OpenShift 容器平台可以为开发人员提供一个企业级解决方案，用于开发和部署容器化软件应用。

红帽 OpenShift 开发二：构建 Kubernetes 应用 (DO288) 是 OpenShift 开发提升课程中的第二门课程，旨在教授学员如何在 OpenShift 集群上设计、构建和部署容器化软件应用。无论是要编写原生容器应用还是要迁移现有应用，本课程都可提供相关的实操培训，以借助红帽 OpenShift 容器平台提升开发人员的生产力。

### 课程目标

- 在 OpenShift 集群上设计、构建和部署容器化应用。

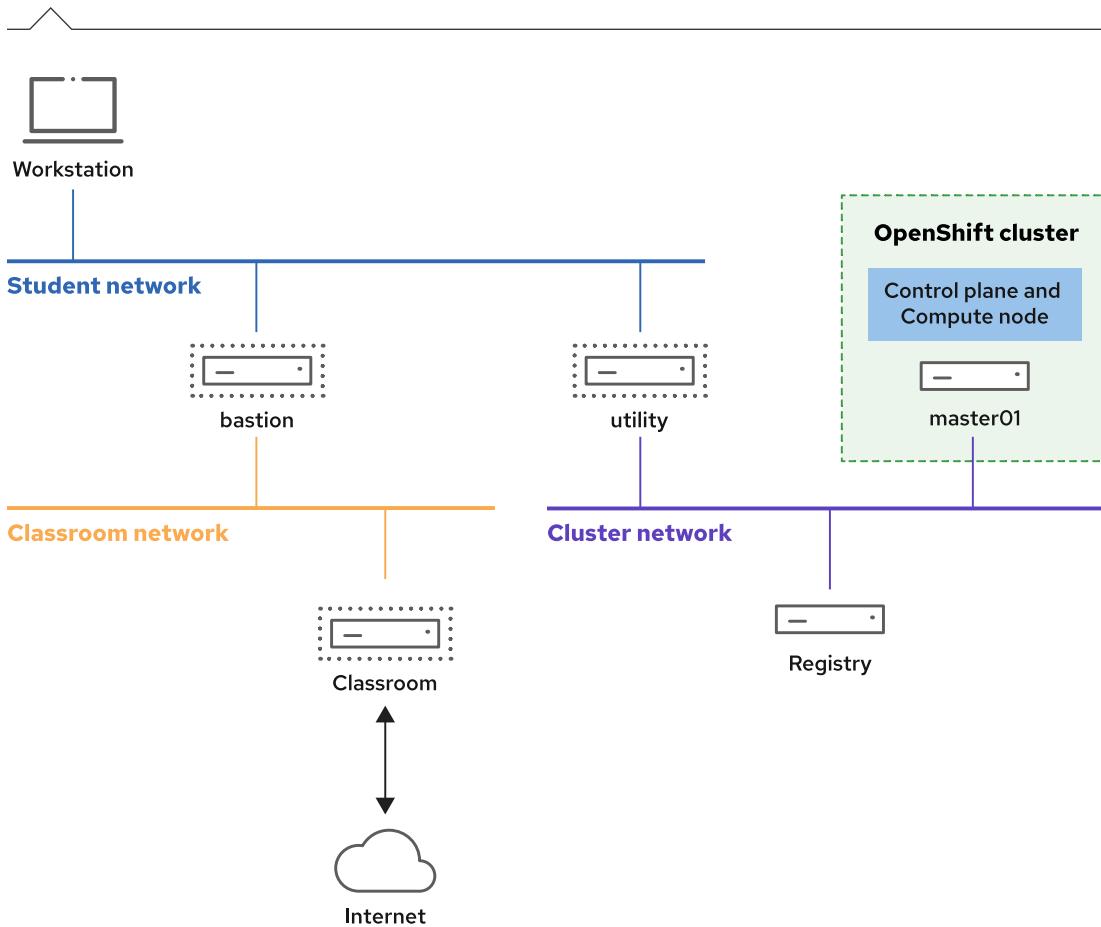
### 培训对象

- 软件开发人员
- 软件架构师

### 前提条件

- 已完成“容器、Kubernetes 和红帽 OpenShift 简介”课程 (DO180)，或具备同等知识。
- 获得 RHCSA 或更高级别的认证有助于导航和使用命令行，但这不是必需的。

# 课堂环境介绍



在本课程中，用于动手实践学习活动的主要计算机系统是 **workstation**。

名为 **bastion** 的系统必须始终处于运行状态。

这两个系统都在 `lab.example.com` DNS 域内。

所有学员计算机系统都有一个标准用户帐户 **student**，其密码为 **student**。所有学员系统的 **root** 密码都是 **redhat**。

## 教室计算机

计算机名称	IP 地址	角色
<code>workstation.lab.example.com</code>	172.25.250.9	学员使用的图形工作站
<code>bastion.lab.example.com</code>	172.25.250.254	将虚拟机链接到中央课程服务的路由器
<code>classroom.lab.example.com</code>	172.25.252.254	用于托管课堂资料的服务器

## 简介

计算机名称	IP 地址	角色
utility.lab.example.com	172.25.250.253	用于提供 RHOCP 集群所需支持服务的服务器，包括 DHCP 和 NFS 以及到 RHOCP 服务器的路由
master01.ocp4.example.com	192.168.50.10	An OpenShift control plane and compute node
registry.ocp4.example.com	192.168.50.50	用于提供镜像注册表服务的服务器

**bastion** 系统充当连接学员计算机的网络和课堂网络之间的路由器。如果 **bastion** 关闭，其他学员计算机可能无法正常工作，甚至可能在启动过程中挂起。

**utility** 系统充当连接 OpenShift 集群计算机网络与学员网络之间的路由器。如果 **utility** 关闭，OpenShift 集群可能无法正常工作，甚至可能在启动过程中挂起。

学员使用 **workstation** 计算机访问专用 OpenShift 集群，学员具有集群管理员特权。

## OpenShift 访问方法

访问方式	端点
Web 控制台	<a href="https://console-openshift-console.apps.ocp4.example.com">https://console-openshift-console.apps.ocp4.example.com</a>
API	<a href="https://api.ocp4.example.com:6443">https://api.ocp4.example.com:6443</a>

OpenShift 集群有一个标准用户帐户 **developer**，其密码为 **developer**。管理帐户 **admin** 的密码为 **redhat**。

# 控制您的系统

在红帽在线学习（ROLE）课堂中，会向您分配远程计算机。自学课程通过托管在 [rol.redhat.com](http://rol.redhat.com) 上的 Web 应用进行访问。您应使用红帽客户门户用户凭据来登录此网站。

## 控制虚拟机

课堂环境中的虚拟机通过网页界面上的控件进行控制。每个课堂虚拟机的状态显示在 **Lab Environment** 标签页上。

The screenshot shows a web-based interface for managing a classroom lab environment. At the top, there are tabs for 'Table of Contents', 'Course', 'Lab Environment' (which is selected), and other icons. Below the tabs, there's a section titled '▶ Lab Controls' with instructions: 'Click CREATE to build all of the virtual machines needed for the classroom lab environment. This may take several minutes to complete. Once created the environment can then be stopped and restarted to pause your experience.' It also states that deleting the lab will remove all virtual machines and lose progress. Below this is a table listing five virtual machines:

VM Name	Status	Action	Open Console
bastion	active	ACTION -	OPEN CONSOLE
classroom	active	ACTION -	OPEN CONSOLE
servera	building	ACTION -	OPEN CONSOLE
serverb	building	ACTION -	OPEN CONSOLE
workstation	active	ACTION -	OPEN CONSOLE

At the bottom left are 'DELETE' and 'STOP' buttons, and at the bottom right is an information icon.

图 0.2: 示例课程 Lab Environment 管理页面

### 计算机状态

虚拟机状态	描述
building	正在创建虚拟机。
active	虚拟机正在运行且可以使用。如果刚刚启动，它或许仍在启动服务。
stopped	虚拟机已彻底关机。启动后，虚拟机将引导至与关闭前相同的状态。磁盘状态保持不变。

### 课堂操作

按钮或操作	描述
CREATE	创建 ROLE 课堂。创建并启动本课堂所需的所有虚拟机。可能需要几分钟来完成创建。
CREATING	正在创建 ROLE 课堂虚拟机。创建并启动本课堂所需的所有虚拟机。可能需要几分钟来完成创建。

## 简介

按钮或操作	描述
DELETE	删除 ROLE 课堂。销毁课堂中的所有虚拟机。 <b>保存在这些系统磁盘上的所有工作都将丢失。</b>
START	启动课堂中的所有虚拟机。
STARTING	正在启动课堂中的所有虚拟机。
STOP	停止课堂中的所有虚拟机。

## 计算机操作

按钮或操作	描述
OPEN CONSOLE	在新的浏览器标签页中连接到虚拟机的系统控制台。必要时，您可以直接登录虚拟机并运行命令。通常情况下，应当仅登录 <b>workstation</b> 虚拟机并使用 <b>ssh</b> 连接到其他虚拟机。
ACTION > Start	启动虚拟机（开机）。
ACTION > Shutdown	正常关闭虚拟机，保存磁盘内容。
ACTION > Power Off	强制关闭虚拟机，仍会保存磁盘内容。该操作等同于从物理机拔除电源。
ACTION > Reset	强制关闭虚拟机，并将关联的存储重置为初始状态。 <b>保存在该系统磁盘上的所有工作都将丢失。</b>

练习开始时，如果按照指引重置单个虚拟机节点，请仅对特定的虚拟机单击 ACTION > Reset。

练习开始时，如果要按照指示重置所有虚拟机，请对列表中的每个虚拟机单击 ACTION > Reset。

如果想要将课堂环境恢复到课程开始时的原始状态，请单击 DELETE 来移除整个课堂环境。实验删除后，单击 CREATE 以调配一套新的课堂系统。



## 自动停止和自动销毁计时器

完成红帽在线学习注册后，您可以获得特定的上机时间。为了帮助您节省分配到的上机时间，ROLE 课堂会使用计时器，当相应的计时器到点时，将关闭或删除课堂环境。

若要调整计时器，请在课程管理页面的底部找到两个 + 按钮。单击自动停止 + 按钮可为自动停止计时器增加一个小时。单击自动销毁 + 按钮可为自动销毁计时器增加一天。自动停止时间最多为 11 小时，自动销毁时间最多为 14 天。在工作时，请小心设置计时器，以免课堂环境意外关闭。请注意不要将计时器设置为不必要的长时间，否则可能会浪费您分配到的订阅时间。

## 进行实验练习

您可能会在本课程中看到以下实验活动类型：

## 简介

- 引导式练习是在演示部分之后的动手实践练习。它将逐步引导您完成要执行的程序。
- 测验通常用于检查知识型学习情况，或在动手实践活动因其他原因而无法进行时使用。
- 章节末实验是一项可评分的实践操作活动，可帮助您检验学习收获。根据该章中的引导式练习，您将完成一组高级步骤，但这些步骤不会引导您完成每一个命令。您会获得一个包含分步演练的解决方案。
- 总复习实验在课程结束时使用。这也是一项可评分的实践操作活动，可能涵盖整个课程的内容。您完成了对活动中要完成的任务的规格说明，但没有收到执行此操作的具体步骤。另外，您也会获得与规格说明相符的分步演练。

若要在每个实践操作活动开始时准备您的实验环境，请使用活动说明中指定的活动名称来运行 **lab start** 命令。类似地，在每个实践操作活动结束时，请使用相同的活动名称运行 **lab finish** 命令，以在活动结束后进行清理。每个实践操作活动在课程中都具有一个唯一名称。

运行练习脚本的语法如下：

```
[student@workstation ~]$ lab action exercise
```

**action** 是 **start**、**grade** 或 **finish** 的一个选项。所有练习都支持 **start** 和 **finish**。仅章节末实验和总复习实验支持 **grade**。

### **start**

**start** 操作验证开始练习所需要的资源。其中可能包括配置设置、创建资源、检查必备服务和验证上一练习中的必要结果。您可以随时进行练习，甚至无需完成前面的练习。

### **grade**

对于可评分的活动，**grade** 操作指示 **lab** 命令评估您的工作，显示一系列评分标准，并为每项标准列出 **PASS** 或 **FAIL** 状态。若要使每项都达到 **PASS** 状态，请纠正问题并重新运行 **grade** 操作。

### **finish**

**finish** 操作将清理练习期间配置的资源。您可以根据需要进行多次练习。

**lab** 命令支持 Tab 键补全。例如，若要列出您可以开始的所有练习，请输入 **lab start**，再按 Tab 键两次。

## 章 1

# 在 OpenShift 集群中部署和管理应用

### 目标

利用各种应用封装方法将应用部署至 OpenShift 集群，并管理应用资源。

### 培训目标

- 描述 OpenShift 4 的架构和新功能。
- 使用 CLI 通过 Dockerfile 将应用部署至集群。
- 通过容器镜像部署应用，并使用 Web 控制台来管理应用资源。
- 通过源代码部署应用，并使用命令行界面管理应用资源。

### 章节

- OpenShift 4 简介（及测验）
- 将应用部署至 OpenShift 集群（及引导式练习）
- 使用 Web 控制台管理应用（及引导式练习）
- 使用 CLI 管理应用（及引导式练习）

### 实验

在 OpenShift 集群中部署和管理应用

# 红帽 OpenShift 容器平台 4 简介

## 培训目标

学完本节后，您应能够描述 OpenShift 容器平台 4 的架构和新功能。

## OpenShift 容器平台 4 架构

红帽 OpenShift 容器平台 4 (RHOC4) 是基于红帽 CoreOS 和 Kubernetes 基础构建的一组模块化组件和服务。OpenShift 为开发人员添加的平台即服务 (PaaS) 功能包括远程管理、安全性增强、监控与审计、应用生命周期管理和自助服务接口。它提供了编排服务，可简化容器化应用的部署、管理和扩展。

OpenShift 集群是可以通过与任何其他 Kubernetes 集群相同的方式来管理，也可以使用 OpenShift 提供的管理工具来管理，如命令行界面或 Web 控制台。这种额外的工具可实现更高效的工作流，并使日常任务更加容易管理。

使用 OpenShift 的一个主要优点是它使用多个节点来确保其托管应用的弹性和可扩展性。OpenShift 组成一个节点服务器集群，这些服务器运行容器，并由一组主控机服务器集中管理。单个主机可以同时充当主控机和节点，但通常应分隔这两种角色以增强稳定性和高可用性。

下图演示了 OpenShift 容器平台 4 架构的高层次逻辑概述。

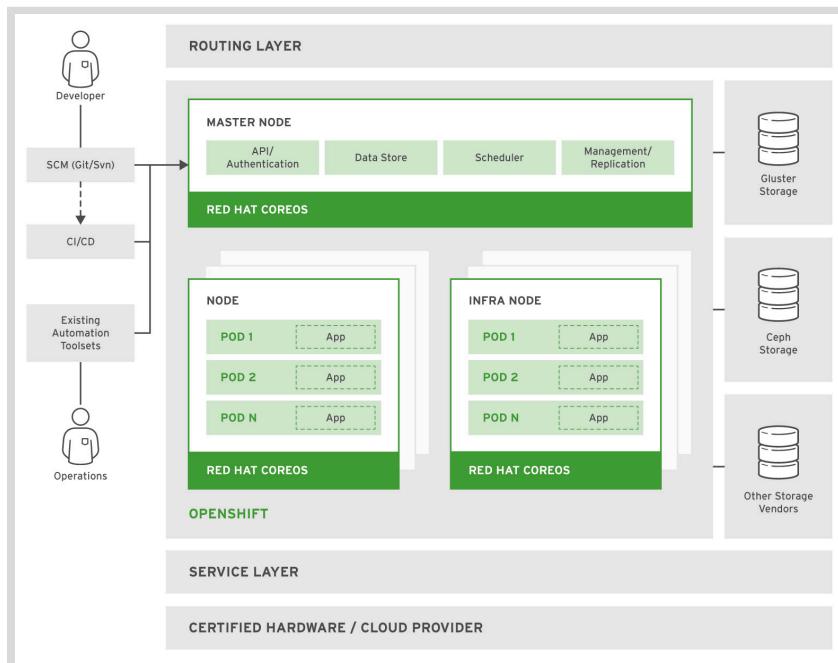


图 1.1: OpenShift 4 架构

以下示意图演示了 OpenShift 容器平台堆栈。

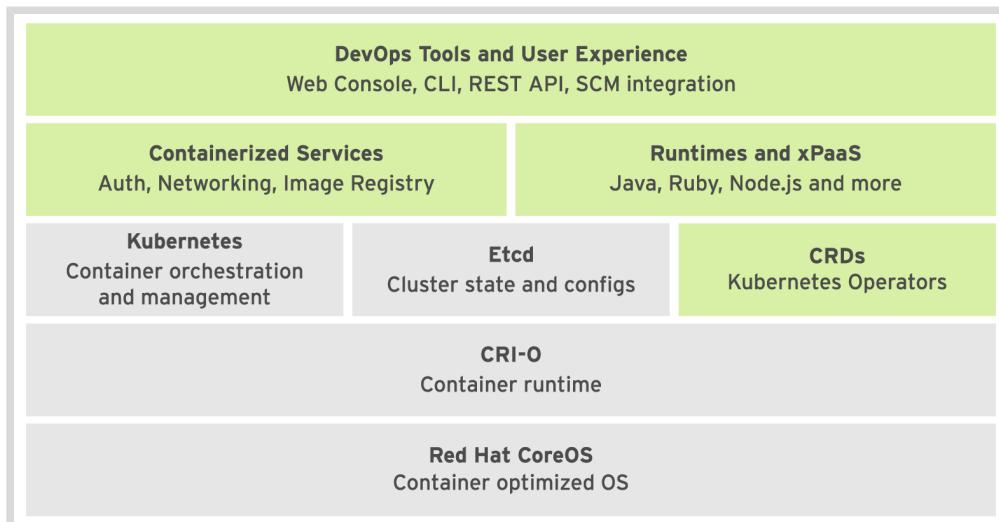


图 1.2: OpenShift 组件堆栈

图中自下而上、从左到右显示了红帽整合并增强的基本容器基础架构：

### 红帽 CoreOS

红帽 CoreOS 是 OpenShift 运行时所基于的基础操作系统。红帽 CoreOS 是一种 Linux 发行版，专注于为容器执行提供不可变的操作系统。

### CRI-O

CRI-O 是一种 Kubernetes 容器运行时接口 (CRI) 实施，以支持使用兼容开放容器项目 (OCI) 的运行时。CRI-O 可以使用符合 CRI 的任何容器运行时，例如：**runc**（由 Docker 服务使用）或 **rkt**（来自于 CoreOS）。

### Kubernetes

Kubernetes 管理运行容器的物理或虚拟主机集群。它使用资源描述由多种资源组成的多容器应用，以及它们的互连方式。

### etcd

etcd 是一种分布式键值存储，供 Kubernetes 用于存储 Kubernetes 集群内容器和其他资源的配置和状态信息。

### 自定义资源定义 (CRD)

自定义资源定义 (CRD) 是存储在 etcd 中并由 Kubernetes 管理的资源类型。这些资源类型构成了 OpenShift 管理的所有资源的状态和配置。

### 容器化服务

容器化服务承担许多 PaaS 基础架构职责，如联网和授权。RHOCP 将 Kubernetes 的基本容器基础架构和底层容器运行时用于大部分内部功能。也就是说，大部分 RHOCP 内部服务作为由 Kubernetes 编排的容器运行。

### 运行时和 xPaaS

运行时和 xPaaS 是可随时供开发人员使用的基础容器镜像，各自预配置了特定的运行时语言或数据库。xPaaS 产品是 JBoss EAP 和 ActiveMQ 等红帽中间件产品的一组基础镜像。红帽 OpenShift 应用运行时 (RHOAR) 是针对 OpenShift 中云原生应用优化的一组运行时。可用的应用运行时有红帽 JBoss EAP、OpenJDK、Thorntail、Eclipse Vert.x、Spring Boot 和 Node.js。

### DevOps 工具和用户体验

DevOps 工具和用户体验：RHOCP 提供 Web UI 和 CLI 管理工具，以用于管理用户应用和 RHOCP 服务。OpenShift Web UI 和 CLI 工具从 REST API 构建，后者可通过 IDE 和 CI 平台等外部工具加以利用。

下表列出了在使用 OpenShift 时最常使用的术语。

## OpenShift 术语

术语	定义
节点	在 OpenShift 集群中托管应用的服务器。
主控节点	管理 OpenShift 集群中控制平面的节点服务器。主控节点提供基本的集群服务，如 API 或控制器。
工作程序节点	也称为计算节点，为集群执行工作负载。应用容器集调度到工作程序节点上。
资源	资源是受 OpenShift 管理的任何种类的组件定义。资源包含受管组件的配置（例如，分配给节点的角色），以及组件的当前状态（例如，节点是否可用）。
控制器	控制器是一种 OpenShift 组件，它监视资源并进行更改，尝试将当前状态移向所需状态。
标签	可分配给任何 OpenShift 资源的键值对。选择器使用标签来筛选合格的资源，以用于调度和其他操作。
命名空间或项目	OpenShift 资源和流程的作用域，以便同名的资源可在不同的上下文中使用。
控制台	OpenShift 提供的 Web UI，供开发人员和管理员用于管理集群资源。



### 注意

最新的 OpenShift 版本将许多控制器实施为操作程序。操作程序是 Kubernetes 插件组件，可以对集群事件做出反应并控制资源状态。操作程序和 Operator 框架不在本课程的讨论范畴。

## RHOCP 4 新功能

RHOCP 4 与先前版本相比有了巨大变化。除了与先前版本保持向后兼容性外，它还包括多种新功能，例如：

- CoreOS 作为所有节点的默认操作系统，提供针对容器优化的不可变基础架构。
- 新的集群安装程序，简化了安装和更新集群中主控机和工作程序节点的过程。
- 一个自我管理平台，能够在不中断服务的前提下自动应用集群更新和恢复。
- 基于“角色”概念而重新设计的 Web 控制台，面向平台管理员和开发人员。
- 用于构建、测试和打包操作程序的 Operator SDK。

## 描述 OpenShift 资源类型

作为开发人员，您将在 OpenShift 中处理许多不同种类的资源类型。可以利用 YAML 或 JSON 文件或通过 OpenShift 管理工具创建和配置这些资源：

### 容器集 (pod)

共享 IP 地址和持久存储卷等资源的容器的集合。容器集是 OpenShift 的基本工作单元。

**服务 (svc)**

特定的 IP/端口组合，提供一组容器集的访问权限。默认情况下，服务以轮询方式将客户端连接到容器集。

**复制控制器 (rc)**

一种 OpenShift 资源，用于定义如何将容器集（水平扩展）复制到不同的节点。复制控制器是一种基本的 OpenShift 服务，可为容器集和容器提供高可用性。

**持久卷 (pv)**

由容器集使用的存储区域。

**持久卷声明 (pvc)**

容器集发出的存储请求。**pvc** 将 **pv** 链接到一个容器集，以便其容器可以使用它，通常是通过将存储挂载到容器的文件系统中。

**配置映射 (cm)**

一组可供其他资源使用的键和值。ConfigMap 和机密通常用于集中由多个资源使用的配置值。机密与 ConfigMap 映射的不同之处在于，机密用于存储敏感数据（通常会加密），并且它们的访问仅限于少数授权用户。

**部署配置 (dc)**

容器集中包含的一组容器，以及要使用的部署策略。**dc** 也提供基本但可扩展的持续交付工作流。

**构建配置 (bc)**

要在 OpenShift 项目中执行的流程。OpenShift 源至镜像 (S2I) 功能使用 BuildConfig 从 Git 存储库中存储的应用源代码构建容器镜像。**bc** 与 **dc** 搭配，提供基本但可扩展的持续集成和持续交付工作流。

**路由**

可被 OpenShift 路由器识别为集群中部署的各种应用和微服务入口点的 DNS 主机名。

**镜像流 (is)**

镜像流及其标记提供了从 OpenShift 容器平台中引用容器镜像的抽象。镜像流及其标记可让您跟踪可用的镜像，并确保即使存储库中的镜像发生变化，您也能使用所需的特定镜像。镜像流不包含实际的镜像数据，而是提供相关镜像的虚拟视图，类似于镜像存储库。

**参考文献****Kubernetes 文档网站**

<https://kubernetes.io/docs/>

**OpenShift 文档网站**

<https://docs.openshift.com/>

**CoreOS 操作程序和 Operator 框架**

<https://coreos.com/operators/>

## ► 小测验

# OpenShift 4 简介

选择以下问题的正确答案：

完成测验后，单击 **CHECK**。如需重试，请单击 **RESET**。单击 **SHOW SOLUTION** 以查看所有正确答案。

### ► 1. 哪一项关于 OpenShift 对 Kubernetes 的增加项的陈述是正确的？

- a. OpenShift 添加了使 Kubernetes 上应用开发和部署变得轻松、高效的功能。
- b. 为 OpenShift 创建的容器镜像无法用于普通的 Kubernetes。
- c. 为启用新的功能，红帽在 RHOC 产品内维护 Kubernetes 的派生版本。
- d. RHOC 没有持续集成和持续部署方面的新功能，但您可以使用外部工具。

### ► 2. 哪一项是关于 OpenShift 中持久存储的正确陈述？

- a. 开发人员创建持久卷声明，以请求项目容器集可用于存储数据的集群存储区域。
- b. 持久卷声明表示容器集可以请求以存储数据的一个存储区域，但它由集群管理员调配。
- c. 持久卷声明表示可分配给节点的内存量，以便开发人员可以声明需要多少内存来运行其应用。
- d. 持久卷声明表示可分配给应用容器集的 CPU 处理单元数量，受集群管理员管理的限值约束。
- e. OpenShift 通过允许管理员直接将节点上可用的存储映射到集群中运行的应用，从而支持持久存储。

### ► 3. 哪两项是关于 OpenShift 资源类型的正确陈述？（请选择两项。）

- a. 容器集负责调配自己的持久存储。
- b. 从同一复制控制器生成的所有容器集必须在同一节点内运行。
- c. 服务负责为容器集外部访问提供 IP 地址。
- d. 路由负责为容器集外部访问提供主机名。
- e. 复制控制器负责监控和维护特定应用的容器集数量。

### ► 4. 哪两项是关于 OpenShift 4 架构的正确陈述？（请选择两项。）

- a. 没有主控机也可以管理 OpenShift 节点。节点形成对等网络。
- b. OpenShift 主控机管理容器集扩展，并且调度容器集以在节点上运行。
- c. 集群中的主节点必须运行红帽 CoreOS。
- d. 集群中的主节点必须运行红帽企业 Linux 8。
- e. 集群中的主节点必须运行红帽企业 Linux 7。

## ► 解决方案

# OpenShift 4 简介

选择以下问题的正确答案：

完成测验后，单击 **CHECK**。如需重试，请单击 **RESET**。单击 **SHOW SOLUTION** 以查看所有正确答案。

### ► 1. 哪一项关于 OpenShift 对 Kubernetes 的增加项的陈述是正确的？

- a. OpenShift 添加了使 Kubernetes 上应用开发和部署变得轻松、高效的功能。
- b. 为 OpenShift 创建的容器镜像无法用于普通的 Kubernetes。
- c. 为启用新的功能，红帽在 RHOCP 产品内维护 Kubernetes 的派生版本。
- d. RHOCP 没有持续集成和持续部署方面的新功能，但您可以使用外部工具。

### ► 2. 哪一项是关于 OpenShift 中持久存储的正确陈述？

- a. 开发人员创建持久卷声明，以请求项目容器集可用于存储数据的集群存储区域。
- b. 持久卷声明表示容器集可以请求以存储数据的一个存储区域，但它由集群管理员调配。
- c. 持久卷声明表示可分配给节点的内存量，以便开发人员可以声明需要多少内存来运行其应用。
- d. 持久卷声明表示可分配给应用容器集的 CPU 处理单元数量，受集群管理员管理的限值约束。
- e. OpenShift 通过允许管理员直接将节点上可用的存储映射到集群中运行的应用，从而支持持久存储。

### ► 3. 哪两项是关于 OpenShift 资源类型的正确陈述？（请选择两项。）

- a. 容器集负责调配自己的持久存储。
- b. 从同一复制控制器生成的所有容器集必须在同一节点内运行。
- c. 服务负责为容器集外部访问提供 IP 地址。
- d. 路由负责为容器集外部访问提供主机名。
- e. 复制控制器负责监控和维护特定应用的容器集数量。

### ► 4. 哪两项是关于 OpenShift 4 架构的正确陈述？（请选择两项。）

- a. 没有主控机也可以管理 OpenShift 节点。节点形成对等网络。
- b. OpenShift 主控机管理容器集扩展，并且调度容器集以在节点上运行。
- c. 集群中的主节点必须运行红帽 CoreOS。
- d. 集群中的主节点必须运行红帽企业 Linux 8。
- e. 集群中的主节点必须运行红帽企业 Linux 7。

## ► 指导练习

# 配置课堂环境

在本练习中，您将配置用于访问本课程使用的所有基础架构的 **workstation**。

## 成果

您应能够：

- 配置 **workstation** 以访问整个课程中要使用的 OpenShift 集群、容器镜像注册表和 Git 存储库。
- 将本课程的示例应用存储库派生到您的个人 GitHub 帐户。
- 从您的个人 GitHub 帐户，将本课程的示例应用存储库克隆到 **workstation** 虚拟机。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 红帽培训在线学习环境中 DO288 课程的访问权限。
- 个人的免费 GitHub 帐户。如果您需要注册 GitHub，请参见中附录 A, 创建 GitHub 帐户的说明。
- 个人的免费 Quay.io 帐户。如果您需要注册 Quay.io，请参见附录 B, 创建 Quay 帐户中的说明。
- 来自 GitHub 的个人访问令牌。

## 说明

开始任何练习之前，必须执行下列所有步骤。

### ► 1. 准备 Github 访问令牌。

- 1.1. 使用 Web 浏览器导航至 <https://github.com> 并进行身份验证。
- 1.2. 在页面顶部，单击配置文件图标，选择 **Settings** 菜单，然后在页面的左侧窗格中选择 **Developer settings**。

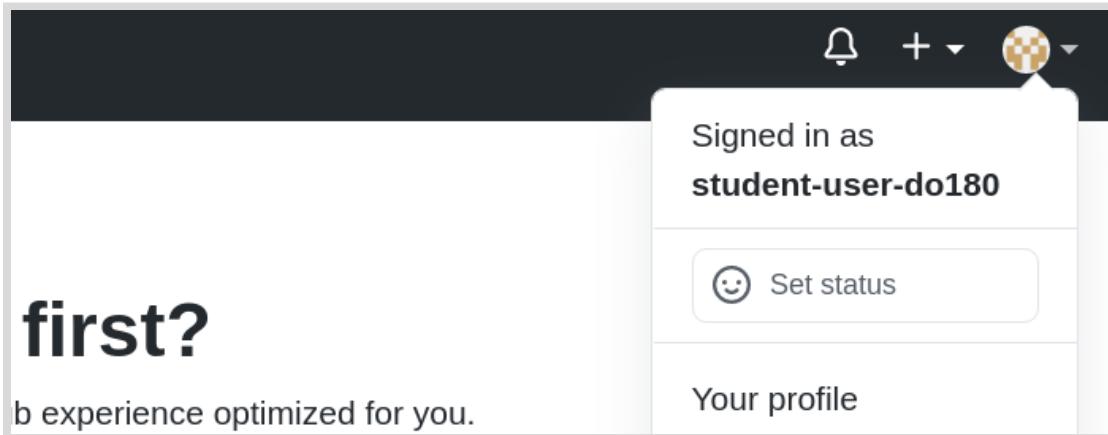


图 1.3: 用户菜单

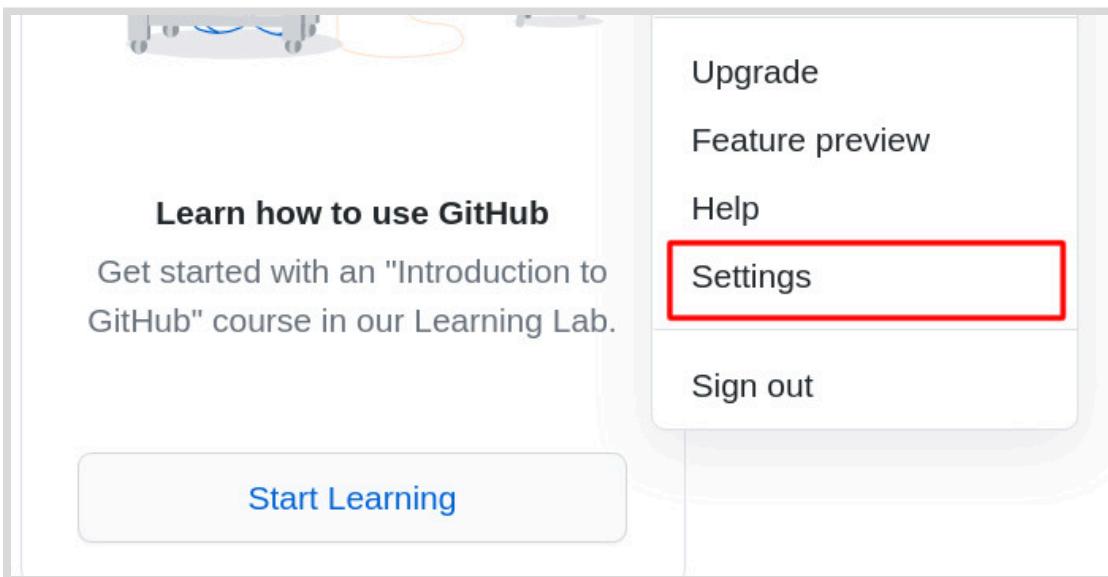


图 1.4: 设置菜单

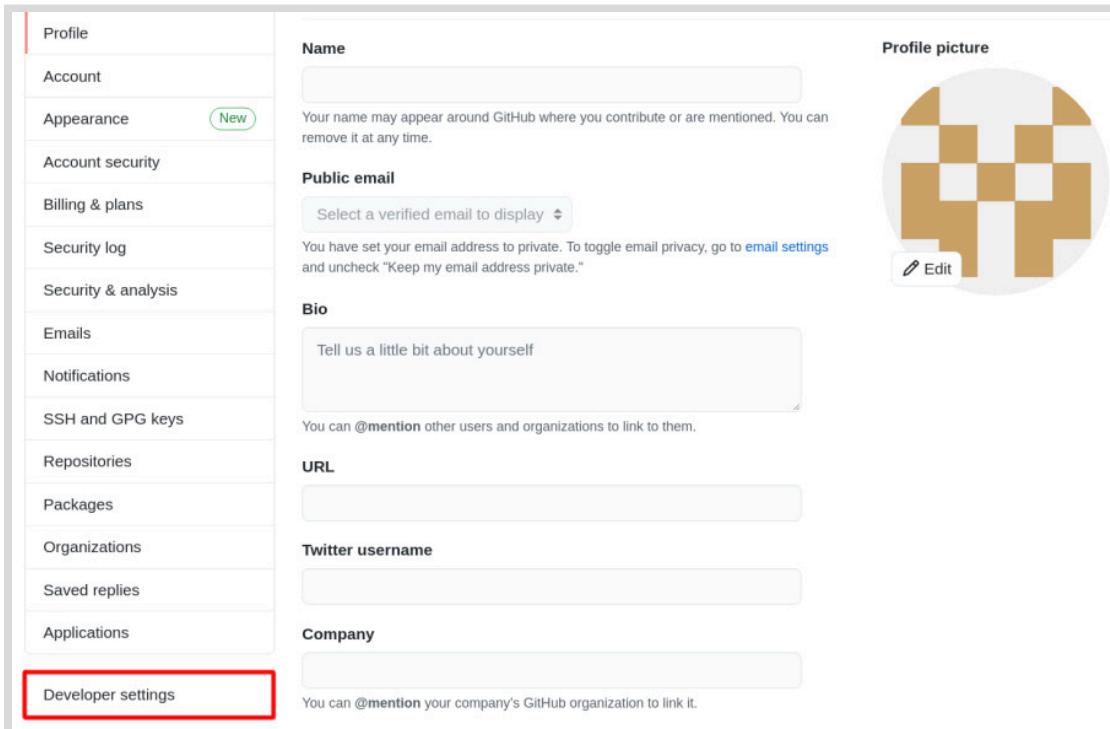


图 1.5: 开发人员设置

- 1.3. 选择左侧窗格中的“个人访问令牌”部分。在下一页，单击 **Generate new token** 以创建新令牌，然后系统会提示您输入密码。

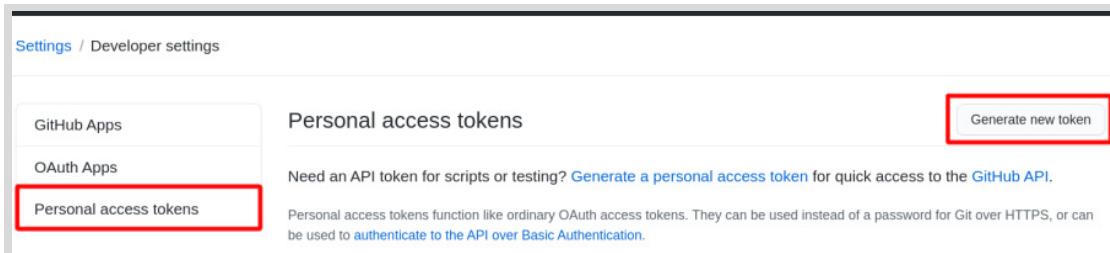


图 1.6: 个人访问令牌窗格

- 1.4. 在 **Note** 字段中，写下关于新访问令牌的简短描述。
- 1.5. 选择 **public\_repo** 选项，并将其他选项保留为未选中状态。通过单击 **Generate token**，创建新的访问令牌。

## New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

**Note**

Course DO280

What's this token for?

**Select scopes**

Scopes define the access for personal tokens. [Read more about OAuth scopes](#).

<input type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events

图 1.7: 个人访问令牌配置

- 1.6. 您的新个人访问令牌显示在输出中。使用首选文本编辑器，在 student 的主目录中创建一个名为 **token** 的新文件，并确保将生成的个人访问令牌粘贴到该文件中。

**重要**

个人访问令牌无法再次显示在 GitHub 中。请务必将令牌保存到安全的位置，否则您再次需要个人访问令牌时必须要重新创建。

## Personal access tokens

Generate new token    Revoke all

Tokens you have generated that can be used to access the [GitHub API](#).

Make sure to copy your new personal access token now. You won't be able to see it again!

✓ ghp_kgYGZwCGE1CrdovkzuzeLTWvYY6eBX2l0vcK	<a href="#">Copy</a>	<a href="#">Delete</a>
--	----------------------	------------------------

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

图 1.8: 生成的访问令牌



## 重要

本课程中，如果在命令行使用 Git 操作时提示您输入密码，请使用访问令牌作为密码。

### ► 2. 您需要配置 **workstation** 虚拟机。

对于以下步骤，请使用红帽培训在线学习环境在您调配在线实验环境时为您提供的值：

The screenshot shows the Red Hat Training Lab Environment interface. At the top, there are tabs for 'Table of Contents', 'Course' (selected), 'Lab Environment', and two icons. Below the tabs, there are two expandable sections: 'SSH Private Key & Instructions' and 'Lab Controls'. The 'Lab Controls' section contains buttons for 'DELETE' (red), 'STOP' (blue), and 'ACTION' (green). To the right of these buttons is a 'Public IP:' label with the value '150.239.52.234'. Below these controls is a table listing three nodes: 'bastion' (active), 'classroom' (active), and 'master01' (active). For each node, there is an 'ACTION' button and an 'OPEN CONSOLE' button. A red 'DOWNLOAD SSH KEY' button is located at the bottom right of the 'SSH Private Key & Instructions' section.

在 **workstation** 虚拟机上打开一个终端，再执行以下命令。回答交互式提示问题以配置您的工作站，然后开始本课程中的任何其他练习。

如果犯了错误，您可以随时使用 **Ctrl+C** 中断命令，并重新开始。

```
[student@workstation ~]$ lab-configure
```

2.1. **lab-configure** 命令首先会显示一系列交互式提示，然后尝试查找其中一些的适当默认值。

```
This script configures the connection parameters to access the OpenShift cluster  
for your lab scripts
```

- Enter the API Endpoint: <https://api.ocp4.example.com:6443> ①
- Enter the GitHub Account Name: `yourgituser` ②
- Enter the Quay.io Account Name: `yourquayuser` ③

```
...output omitted...
```

① 您 OpenShift 集群主 API 的 URL。以单行形式键入 URL，不含空格或换行符。在调配实验环境时，红帽培训会为您提供此信息。您需要此信息来登录集群并且部署容器化的应用。

② ③ 您的个人 GitHub 和 Quay.io 帐户名称。您需要在这些在线服务上拥有有效的免费帐户，才能完成本课程的练习。如果从未使用过这些在线服务中的任何一项，请

参阅附录 A, 创建 GitHub 帐户 和 附录 B, 创建 Quay 帐户 来了解有关如何注册的说明。

2.2. `lab-configure` 命令会显示您输入的所有信息，并尝试连接您的 OpenShift 集群：

```
...output omitted...

You entered:
· API Endpoint:          https://api.ocp4.example.com:6443
· GitHub Account Name:   yourgithubuser
· Quay.io Account Name:  yourquayuser

...output omitted...
```

2.3. 如果到目前为止一切正常，则 `lab-configure` 会尝试访问您的公共 GitHub 和 Quay.io 帐户：

```
...output omitted...

Verifying your GitHub account name...

Verifying your Quay.io account name...

...output omitted...
```

2.4. 如果所有检查都通过，`lab-configure` 命令会保存您的配置：

```
...output omitted...

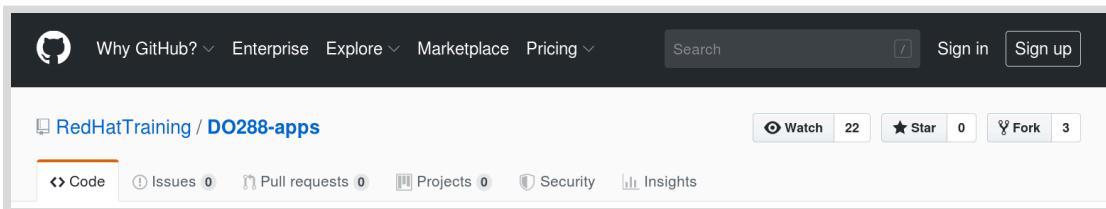
Saving your lab configuration file...

All fine, lab config saved. You can now proceed with your exercises.
```

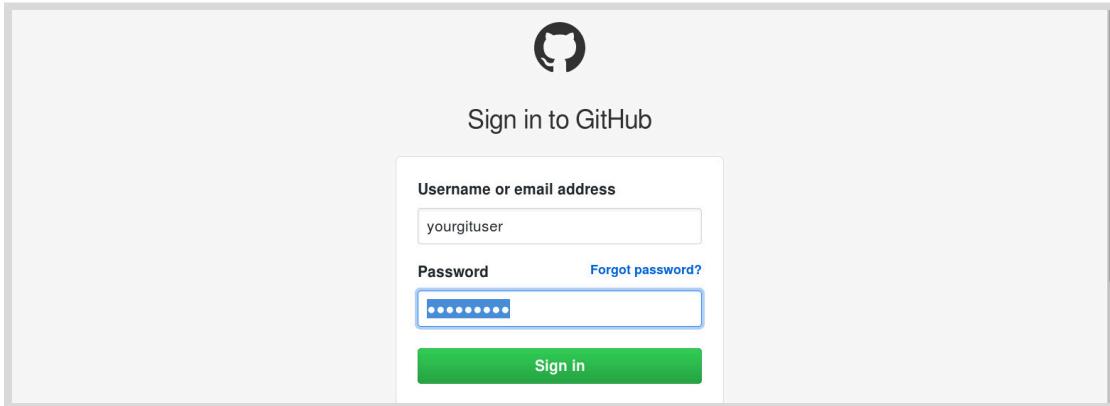
2.5. 如果保存配置时没有出现错误，您差不多可以开始本课程的任何练习。如果出现任何错误，请不要尝试开始任何练习，直到您可以成功执行 `lab-configure` 命令为止。

► 3. 开始任何练习之前，您需要将本课程的示例应用派生到您的个人 GitHub 帐户。执行以下步骤：

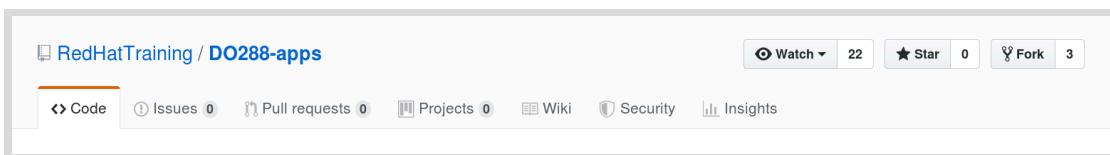
3.1. 打开 Web 浏览器并前往 <https://github.com/RedHatTraining/DO288-apps>。如果您没有登录 GitHub，请单击右上角的 **Sign in**。



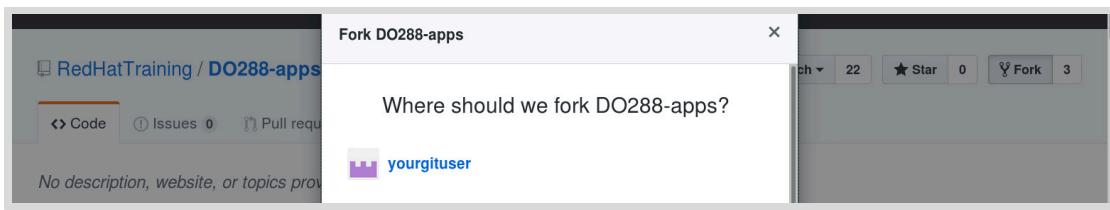
3.2. 使用您的个人用户名和密码来登录 GitHub。



3.3. 返回到 RedHatTraining/DO288-apps 存储库，并单击右上角的 Fork。



3.4. 在 Fork DO288-apps 窗口中，单击 yourgituser 以选择您的个人 GitHub 项目。



### 重要

虽然可以重命名 <https://github.com/RedHatTraining/DO288-apps> 存储库的个人分支，但本课程中的评测脚本、帮助程序脚本和示例输出假定在对存储库派生时保留名称 DO288-apps。

3.5. 几分钟后，GitHub Web 界面将显示您的新存储库 yourgituser/DO288-apps。

▶ 4. 开始任何练习之前，您需要将本课程的示例应用从您的个人 GitHub 帐户克隆到 workstation 虚拟机。执行以下步骤：

4.1. 运行以下命令，以克隆本课程的示例应用存储库。将 yourgithubuser 替换为您的个人 GitHub 帐户的名称：

```
[student@workstation ~]$ git clone https://github.com/yourgithubuser/DO288-apps
Cloning into 'DO288-apps'
...output omitted...
```

4.2. 验证 /home/student/DO288-apps 是否为 Git 存储库：

```
[student@workstation ~]$ cd D0288-apps  
[student@workstation D0288-apps]$ git status  
# On branch main  
  
nothing to commit, working directory clean
```

- 4.3. 验证 `/home/student/D0288-apps` 是否包含本课程的示例应用，再改回到 `student` 用户的主文件夹。

```
[student@workstation D0288-apps]$ head README.md  
# D0288 Containerized Example Applications  
...output omitted...  
[student@workstation D0288-apps]$ cd ~  
[student@workstation ~]$
```

## 完成

您的 `workstation` 虚拟机上现在已拥有 `D0288-apps` 存储库的本地克隆，您也成功执行了 `lab-configure` 命令，这表示您已准备好开始本课程的练习。

在本课程中，从源代码构建应用的所有练习都从 `main` Git 存储库的 `D0288-apps` 分支开始。对源代码进行更改的练习需要您创建新的分支来托管您的更改，从而使 `main` 分支始终包含已知良好的起始点。如果出于某种原因，您需要暂停或重新开始练习，并且需要保存或放弃您在 Git 分支中所做的更改，请参阅附录 C, 实用的 Git 命令。

本引导式练习到此结束。

# 在 OpenShift 集群中部署应用

## 培训目标

学完本节后，您应能够：

- 使用 CLI 通过 Dockerfile 将应用部署至集群。
- 描述使用 `oc new-app` 命令和 Web 控制台在项目中创建的资源。

## 开发路径

红帽 OpenShift 容器平台专为构建和部署容器化应用而设计。OpenShift 支持的主要用例共有两种：

- 使用 OpenShift 工具管理整个应用生命周期，从初始开发到投入生产
- 向 OpenShift 部署在 OpenShift 外部构建的现有容器化应用



### 重要

在 OpenShift 4.6 中，`oc new-app` 命令现在默认生成 Deployment 资源，而不是 DeploymentConfig 资源。此版本的 DO288 仅在需要时使用 DeploymentConfig。要创建 DeploymentConfig 资源，您可以在调用 `oc new-app` 时传递 `--as-deployment-config` 标志。有关更多信息，请参见 [了解 Deployment 和 DeploymentConfig](https://docs.openshift.com/container-platform/4.6/applications/deployments/what-deployments-are.html) [https://docs.openshift.com/container-platform/4.6/applications/deployments/what-deployments-are.html]。

`oc new-app` 命令可以创建构建应用并将其部署至 OpenShift 所需的资源。它会根据所需用例创建不同的资源：

- 如果您希望 OpenShift 管理整个应用生命周期，请使用 `oc new-app` 命令来创建构建配置，以管理用于创建应用容器镜像的构建流程。`oc new-app` 命令还会创建部署配资源，以管理用于在 OpenShift 集群中运行所生成容器镜像的部署流程。在以下示例中，您要将整个生命周期委派到 OpenShift 集群：克隆 Git 存储库，构建容器镜像，并将它部署到 OpenShift 集群。

```
[user@host ~]$ oc new-app \
https://github.com/RedHatTraining/DO288/tree/main/apps/apache-httpd
```

如果 URL 引用 Git 存储库，您可以选择使用井号 (#) 分隔符来指定分支名称。例如，以下 URL 指示使用名为 my-branch 的分支 `my-branch`: `https://github.com/RedHatTraining/DO288-apps#my-branch`。

- 如果您有现有的容器化应用要部署到 OpenShift，请使用 `oc new-app` 命令来创建部署资源，以管理用于在 OpenShift 集群中运行现有容器镜像的部署流程。在以下示例中，您要使用 `--docker-image` 选项来引用容器镜像：

```
[user@host ~]$ oc new-app --docker-image=registry.access.redhat.com/rhel7-mysql57
```

`oc new-app` 命令还会创建一些辅助资源，如服务和镜像流。要支持 OpenShift 的容器化应用管理方式，需要使用这些资源，本课程的后续章节会谈及这些资源。

Web 控制台中的 **Add to Project** 按钮所执行的任务与 `oc new-app` 命令相同。本课程的后续章节将介绍 OpenShift Web 控制台及其用法。

## 介绍 `oc new-app` 命令选项

`oc new-app` 命令可通过单个 URL 参数（最简单的形式）指向 Git 存储库或容器镜像。它会访问这个 URL，以确定如何解释该参数并执行构建或部署。

`oc new-app` 命令做出的决定可能与您的期望不符。例如：

- 如果 Git 存储库同时包含 Dockerfile 和 `index.php` 文件，OpenShift 将无法确定要采取的方式，除非有明确提及。
- 如果 Git 存储库包含以 PHP 为目标的源代码，但 OpenShift 集群支持部署 PHP 版本 5.6 或 7.0，则构建流程将失败，因为它不清楚要使用哪一个版本。

为了适应这些方案和其他方案，`oc new-app` 命令提供了多个选项来进一步指定构建应用的确切方式：

### 支持的选项

选项	描述
<code>--as-deployment-config</code>	配置 <code>oc new-app</code> ，以创建 DeploymentConfig 资源而非 Deployment 资源。
<code>--image-stream</code> <code>-i</code>	提供镜像流以用作 S2I 构建的 S2I 构建器镜像或用来部署容器镜像。
<code>--strategy</code>	<code>docker</code> 或 <code>pipeline</code> 或 <code>source</code>
<code>--code</code>	提供指向要用作 S2I 构建输入的 Git 存储库的 URL。
<code>--docker-image</code>	提供指向要部署的容器镜像的 URL。
<code>--dry-run</code>	设置为 <code>true</code> ，以显示操作结果而不执行该操作。
<code>--context-dir</code>	提供要视为根的目录的路径。

## 使用 OpenShift 管理整个应用生命周期

OpenShift 使用源至镜像 (S2I) 流程来管理应用生命周期。S2I 会从 Git 存储库获取应用源代码，将其与基础容器镜像整合，创建来源，然后使用已运行就绪的应用来创建容器镜像。

`oc new-app` 命令会将 Git 存储库 URL 用作输入参数，并检查应用源代码，以确定要用于创建应用容器镜像的构建器镜像：

```
[user@host ~]$ oc new-app http://gitserver.example.com/mygitrepo
```

`oc new-app` 命令可以选择将构建器镜像流名称用作参数，即将其作为 Git URL 的一部分（添加波形符 (~) 作为前缀）或使用 `--image-stream` 参数（简写形式：`-i`）。

以下两个命令演示了使用 PHP S2I 构建器镜像：

```
[user@host ~]$ oc new-app php~http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app -i php http://gitserver.example.com/mygitrepo
```

可以选择在镜像流名称后面加上一个标记，通常为编程语言运行时的版本号。例如：

```
[user@host ~]$ oc new-app php:7.0~http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app -i php:7.0 http://gitserver.example.com/mygitrepo
```

## 指定镜像流名称

部分开发人员更喜欢使用 **-i** 选项，而非波形符表示法，因为波形符的可读性欠佳，具体取决于屏幕字体。以下三个命令会产生相同的结果：

```
[user@host ~]$ oc new-app \
myis~http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app \
-i myis http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app -i myis --strategy source \
--code http://gitserver.example.com/mygitrepo
```

虽然 **oc new-app** 命令旨在作为交付应用的一种便捷方式，但开发人员需要注意该命令将尝试“猜测”给定 Git 存储库的源语言。

在上一示例中，如果 **myis** 并非 OpenShift 提供的标准 S2I 镜像流，则只有第一个示例能够正常运行。波形符表示法禁用 **oc new-app** 命令的语言检测功能。这允许使用指向 **oc new-app** 命令未知编程语言的构建器的镜像流。

波形符 (~) 和 **--image-stream (-i)** 选项工作方式不同，**-i** 选项要求在本地安装 git 客户端，因为语言检测需要克隆存储库，从而能检查项目，而波形符 (~) 表示法则不会这样。

## 将现有容器化应用部署到 OpenShift

如果您在 OpenShift 外部开发应用，并且应用容器镜像可从 OpenShift 集群可访问的容器镜像注册表中获取，则 **oc new-app** 命令可以将容器镜像 URL 用作输入参数：

```
[user@host ~]$ oc new-app \
registry.example.com/mycontainerimage
```

请注意，根据上述命令，无法确定 URL 引用的是 Git 存储库还是注册表中的某个容器镜像。**oc new-app** 命令会访问输入 URL 以消除这种不确定性。OpenShift 会检查该 URL 的内容，并确定它是源代码还是容器镜像注册表。为避免这种不确定性，请使用 **--code** 或 **--docker-image** 选项。例如：

```
[user@host ~]$ oc new-app \
--code http://gitserver.example.com/mygitrepo
```

```
[user@host ~]$ oc new-app \
--docker-image registry.example.com/mycontainerimage
```

## 使用 OpenShift 部署现有 Dockerfile

在许多情况下，您已拥有通过 Dockerfile 构建的容器镜像。如果 Dockerfile 可通过 Git 存储库访问，则 `oc new-app` 命令可以创建用于在 OpenShift 集群内部执行 Dockerfile 构建的构建配置，然后将所生成的容器镜像提取到内部注册表中：

```
[user@host ~]$ oc new-app \
http://gitserver.example.com/mydockerfileproject
```

OpenShift 会访问源 URL，以确定其是否包含 Dockerfile。如果同一项目中包含编程语言的源文件，则 OpenShift 可能会创建适用于 S2I 构建（而非 Dockerfile 构建）的构建器配置。为避免这种不确定性，请使用 `--strategy` 选项：

```
[user@host ~]$ oc new-app \
--strategy docker http://gitserver.example.com/mydockerfileproject
```

以下示例演示了如何将 `--strategy` 选项用于 S2I 构建：

```
[user@host ~]$ oc new-app \
--strategy source http://gitserver.example.com/user/mygitrepo
```

还可在带有 `--strategy` 的同一个命令中使用其他选项，如 `--image-stream` 和 `--code`。



### 注意

`oc new-app` 命令还可提供一些选项，以从模板创建应用，或者通过 Jenkins 管道构建应用，但这些选项不在本章节的讨论范围内。

## oc new-app 命令创建的资源

`oc new-app` 命令可向当前项目添加以下资源，以支持构建和部署应用：

- 用于从源代码或 Dockerfile 构建应用容器镜像的构建配置。
- 指向内部注册表中的已生成镜像或指向外部注册表中的现有镜像的镜像流。
- 会使用镜像流作为输入来创建应用容器集的部署资源。
- 适用于应用容器镜像公开的所有端口的服务。如果应用容器镜像没有声明任何公开端口，则 `oc new-app` 命令不会创建任何服务。

这些资源会启动一系列进程，进而在项目中创建更多资源，如用于运行容器化应用的应用容器。

以下命令基于镜像 `mysql` 创建应用，其标签设为 `db=mysql`：

```
[user@host ~]$ oc new-app \
mysql -e MYSQL_USER=user -e MYSQL_PASSWORD=pass \
-e MYSQL_DATABASE=testdb -l db=mysql
```

下图显示了在参数是容器镜像时 `oc new-app` 命令创建的 Kubernetes 和 OpenShift 资源：

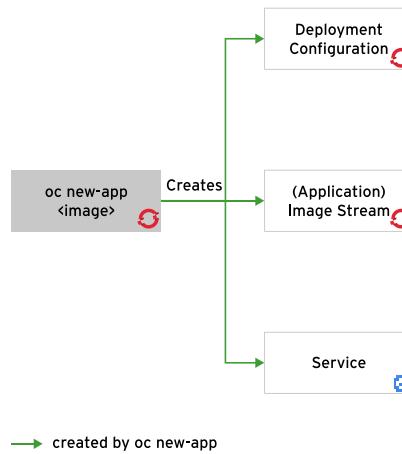


图 1.15: oc new-app 命令创建的资源

以下命令会使用部署配置从 PHP 编程语言的源代码创建应用：

```
[user@host ~]$ oc new-app --as-deployment-config \
--name hello -i php \
--code http://gitserver.example.com/mygitrepo
```

在完成构建和部署流程后，请使用 `oc get all` 命令显示 `test` 项目中的所有资源。除了 `oc new-app` 命令所创建的资源外，输出中还显示一些其他资源：

NAME	TYPE	FROM	LATEST		
bc/hello	Source	Git	3 <span>1</span>		
NAME	TYPE	FROM	STATUS	STARTED	DURATION
builds/hello-1	Source	Git@3a0af02	Complete	About an hour ago	1m16s <span>2</span>
NAME	DOCKER REPO			TAGS	UPDATED
is/hello	docker-registry.default.svc:5000/test/hello			3 <span>3</span>	
NAME	REVISION	DESIRED	CURRENT	TRIGGERED BY	
dc/hello	1	1	1	config,image(hello:latest)	4 <span>4</span>
NAME	DESIRED	CURRENT	READY	AGE	
rc/hello-1	1	1	1	3m <span>5</span>	
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
svc/hello	172.30.2.186	<none>	8080/TCP	2m31s <span>6</span>	

NAME	READY	STATUS	RESTARTS	AGE
po/hello-1-build	0/1	Completed	0	2m11s ⑦
po/hello-1_tmf1	1/1	Running	0	1m23s ⑧

- ① `oc new-app` 命令创建的构建配置。
- ② `oc new-app` 命令触发的第一个构建。
- ③ `oc new-app` 命令创建的镜像流。它会指向 S2I 流程创建的容器镜像。
- ④ `oc new-app` 命令创建的部署配置。
- ⑤ 首次部署时创建的复制控制器配置。后续部署可能也会创建部署程序容器集。
- ⑥ `oc new-app` 命令使用 PHP S2I 构建器镜像公开端口 8080/TCP 创建的服务。
- ⑦ OpenShift 会保留来自最新构建的构建容器集，因为您可能需要检查相关日志。部署程序容器集会在成功终止后删除。
- ⑧ 首次部署时创建的应用容器集。

以下命令会使用部署（而非部署配置）从 PHP 编程语言的源代码创建应用：

```
[user@host ~]$ oc new-app \
--name hello -i php \
--code http://gitserver.example.com/mygitrepo
```

在完成构建和部署流程后，请使用 `oc get all` 命令显示 `test` 项目中的所有资源。除了 `oc new-app` 命令所创建的资源外，输出中还显示一些其他资源：

NAME	READY	STATUS	RESTARTS	AGE
pod/hello-1-build	0/1	Completed	0	2m11s ①
pod/hello-57f548f776-q86pg	1/1	Running	0	1m23s ②
NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/hello	1/1	1	1	62s ③
NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/hello-57f548f776	1	1	1	3m ④
replicaset.apps/hello-875348fa8e	0	0	0	4m
NAME	TYPE	FROM	LATEST	
buildconfig.build.openshift.io/hello	Source	Git	3 ⑤	
NAME	TYPE	FROM	STATUS	STARTED
DURATION				
build.build.openshift.io/hello-1	Source	Git@3a0af02	Complete	About an hour ago ⑥
NAME	IMAGE REPOSITORY			
TAGS	UPDATED			
imagestream.image.openshift.io/hello-hello	latest	46 seconds ago ⑦	docker-registry.default.svc:5000/test/	

- ① OpenShift 会保留来自最新构建的构建容器集，因为您可能需要检查相关日志。部署程序容器集会在成功终止后删除。
- ② 首次部署时创建的应用容器集。
- ③ `oc new-app` 命令创建的部署配置。
- ④ 首次部署时创建的复制控制器配置。后续部署可能也会创建部署程序容器集。
- ⑤ `oc new-app` 命令创建的构建配置。
- ⑥ `oc new-app` 命令触发的第一个构建。
- ⑦ `oc new-app` 命令创建的镜像流。它会指向 S2I 流程创建的容器镜像。

应用可能需要使用某些并非由 `oc new-app` 命令创建的资源，如路由、密码和持久卷声明。您可以在使用 `oc new-app` 命令前后使用其他 `oc` 命令来创建这些资源。

`oc new-app` 命令创建的所有资源都包含 `app` 标签。`app` 标签的值与应用 Git 存储库或现有容器镜像的短名称一致。要为 `app` 标签指定其他值，请使用 `--name` 选项，例如：

```
[user@host ~]$ oc new-app \
--name test http://gitserver.example.com/mygitrepo
```

您可以使用单个 `oc delete` 命令和 `app` 标签来删除 `oc new-app` 命令所创建的资源，而无需删除整个项目，也不会影响项目中可能存在的其他资源。以下命令可以删除上一条 `oc new-app` 命令创建的所有资源：

```
[user@host ~]$ oc delete all -l app=test
```

使用 `--name` 选项的参数来指定由 `oc new-app` 命令创建的资源（如构建配置和服务）的基础名称。

`oc new-app` 命令可在同一个 OpenShift 项目中执行多次，以便每次创建一个多容器应用。例如：

- 使用指向 MongoDB 数据库容器镜像的 URL 来运行 `oc new-app` 命令，以创建数据库容器集和服务。
- 使用指向 Node.js 应用（需要访问数据库）的 Git 存储库的 URL 以及第一次调用时所创建的服务来运行 `oc new-app` 命令。

稍后，您可以将这两个命令创建的所有资源都导出到某个模板文件中。

如果您想要查看资源定义，但不在现有项目中创建资源，请使用 `-o` 选项：

```
[user@host ~]$ oc new-app \
-o json registry.example.com/mycontainerimage
```

资源定义会被发送到标准输出，也可重定向到某个文件。然后，您可以对所生成的文件进行自定义，并将其插入到模板定义中。



### 注意

OpenShift 提供了很多适用于常见场景的预定义模板，如一个数据库加上一个应用。例如，`rails-postgresql` 模板会部署一个 PostgreSQL 数据库容器镜像和一个基于来源所构建的 Ruby on Rails 应用。

若要获取 `oc new-app` 命令支持的选项的完整列表，并查看示例列表，请运行 `oc new-app -h` 命令。

## 使用镜像流和标签引用容器镜像

OpenShift 社区建议使用 image stream 资源来引用容器镜像，而不要直接引用容器镜像。镜像流资源会指向内部注册表或外部注册表中的容器镜像，并存储元数据（如可用的标记和镜像内容校验和）。

将容器镜像元数据存入镜像流后，OpenShift 便可基于此类数据执行各种操作（如镜像缓存），无需每次都访问注册表服务器。这样做还使得 OpenShift 能够使用通知或池策略来响应镜像内容更新。

构建配置和部署配置都会使用镜像流事件来执行各种操作，例如：

- 因构建器镜像有所更新而触发新的 S2I 构建。
- 因应用容器镜像在外部注册表中更新而为应用触发新的容器集部署。

要创建镜像流，最简单的方法就是使用带有 `--confirm` 选项的 `oc import-image` 命令。以下示例会为来自不安全注册表（位于 `registry.acme.example.com`）的 `acme/awesome` 容器镜像创建一个名为 `myis` 的镜像流：

```
[user@host ~]$ oc import-image myis --confirm \
--from registry.acme.example.com:5000/acme/awesome --insecure
```

`openshift` 项目提供多个镜像流，供所有 OpenShift 集群用户使用。您可以使用 `oc new-app` 命令和 OpenShift 模板，在当前项目中创建自己的镜像流。

一个镜像流资源可以定义多个镜像流标记。镜像流标记可以指向另一个容器镜像标记或另一个容器镜像名称。这意味着，您可以为常用镜像（如 S2I 构建器镜像）使用更加简短的名称，并为同一镜像的各个变体使用不同的名称或注册表。例如，`openshift` 项目中的 `ruby` 镜像流定义了以下镜像流标记：

- `ruby:2.5` 是指红帽容器目录中的 `rhel8/ruby-25`。
- `ruby:2.6` 是指红帽容器目录中的 `rhel8/ruby-26`。



### 参考文献

如需更多信息，请参阅红帽 OpenShift 容器平台 4.6 的 CLI reference 中的 Developer CLI commands 章节，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/cli\\_tools/index#cli-developer-commands](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/cli_tools/index#cli-developer-commands)

## ► 指导练习

# 在 OpenShift 集群中部署应用

在本练习中，您将使用 OpenShift 从 Dockerfile 中构建和部署应用。

## 成果

您应能够采用 `docker` 构建策略来创建应用，并在不删除项目的情况下从应用中删除所有资源。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 示例应用的父镜像 (`ubi8/ubi`)。
- Git 存储库中的示例应用 (`ubi-echo`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab docker-build start
```

## 说明

### ► 1. 检查示例应用的 Dockerfile。

1.1. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `main` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

1.2. 创建一个新分支，以保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b docker-build
Switched to a new branch 'docker-build'
[student@workstation D0288-apps]$ git push -u origin docker-build
...output omitted...
* [new branch]      docker-build -> docker-build
Branch docker-build set up to track remote branch docker-build from origin.
```

1.3. 查看 `ubi-echo` 文件内应用的 Dockerfile：

```
[student@workstation D0288-apps]$ cat ubi-echo/Dockerfile
FROM registry.access.redhat.com/ubi8/ubi:8.0 ①
USER 1001 ②
CMD bash -c "while true; do echo test; sleep 5; done" ③
```

- ① 父镜像是红帽企业Linux 8.0 的通用基础镜像 (UBI)，来自于红帽容器目录。
- ② 此容器镜像将以这个用户 ID 的身份运行。此处可以使用任何非零值。目的只是为了使其有别于标准系统用户，如通常在更低的 UID 值范围内的 apache。
- ③ 该应用每五秒钟会运行一次循环以回显 “test”。

## ▶ 2. 使用 OpenShift 集群构建应用容器镜像。

### 2.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

### 2.2. 使用您的开发人员用户名登录 OpenShift:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

### 2.3. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-docker-build
Now using project "developer-docker-build" on server "https://
api.ocp4.example.com:6443".
```

### 2.4. 从 ubi-echo 文件夹中的 Dockerfile 创建一个名为 “echo” 的新应用。使用您在上一步中创建的分支。它会（搭配其他资源）创建构建配置：

```
[student@workstation D0288-apps]$ oc new-app --name echo \
https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#docker-build \
--context-dir ubi-echo
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "ubi" created
  imagestream.image.openshift.io "echo" created
  buildconfig.build.openshift.io "echo" created
  deployment.apps.openshift.io "echo" created
--> Success
...output omitted...
```

忽略关于以 root 身份运行基础镜像的警告。请记住，您的 Dockerfile 已切换到非特权用户。

### 2.5. 根据构建日志，采取相应措施：

```
[student@workstation D0288-apps]$ oc logs -f bc/echo
Cloning "https://github.com/youruser/D0288-apps#docker-build" ...
Replaced Dockerfile FROM image registry.access.redhat.com/ubi8/ubi:8.0
Caching blobs under "/var/cache/blobs".

...output omitted...
Pulling image registry.access.redhat.com/ubi8/ubi@sha256:1a2a...75b5
...output omitted...
STEP 1: FROM registry.access.redhat.com/ubi8/ubi@sha256:1a2a...75b5 ①
STEP 2: USER 1001
STEP 3: CMD bash -c "while true; do echo test; sleep 5; done"
STEP 4: ENV "OPENSHIFT_BUILD_NAME"="echo-1" ... ②
STEP 5: LABEL "io.openshift.build.commit.author"=...
STEP 6: COMMIT temp.builder.openshift.io/developer-docker-build/echo-1:... ③
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/developer-docker-
build/echo:latest ... ④
Push successful
```

- ① `oc new-app` 命令将 Git 存储库正确识别成了 Dockerfile 项目，而且 OpenShift 构建执行了 Dockerfile 构建。
- ② OpenShift 使用 `ENV` 和 `LABEL` 指令向应用容器镜像追加了元数据。
- ③ OpenShift 将应用镜像提交到节点的容器引擎。
- ④ OpenShift 将应用镜像从节点的容器引擎推送到集群的内部注册表。

### ▶ 3. 验证应用能否在 OpenShift 中正常工作。

3.1. 等待部署应用容器镜像。重复 `oc status` 命令，直至输出显示部署成功：

```
[student@workstation D0288-apps]$ oc status
In project youruser-docker-build on server
  https://api.ocp4.example.com:6443

deployment/echo deploys istag/echo:latest <-
  bc/echo docker builds https://github.com/your-GitHub-user/D0288-apps#docker-
build on istag/ubi:8.0
  deployment #2 running for 20 minutes - 1 pod
  deployment #1 deployed 21 minutes ago
...output omitted...
```

第一个部署适用于构建器容器集。第二个部署是运行的应用容器集。

3.2. 等待应用容器集准备就绪并在运行。重复 `oc get pod` 命令，直到显示与下方类似的输出：

```
[student@workstation D0288-apps]$ oc get pod
NAME        READY   STATUS    RESTARTS   AGE
echo-1-build  0/1     Completed  0          1m
echo-555xx    1/1     Running   0          14s
```

3.3. 显示应用容器集日志，以表明应用容器镜像正在 OpenShift 下方输出所期望的内容。使用上一步中获得的应用容器集名称。

```
[student@workstation D0288-apps]$ oc logs echo-555xx | tail -n 3
test
test
test
```

▶ 4. 检查构建和部署配置，以了解它们与镜像流有何关联。

4.1. 查看构建配置：

```
[student@workstation D0288-apps]$ oc describe bc echo
Name:           echo
...output omitted...
Labels:         app=echo
...output omitted...
Strategy:      Docker
URL:           https://github.com/your-GitHub-user/D0288-apps
Ref:            docker-build ①
ContextDir:    ubi-echo ②
From Image:    ImageStreamTag ubi:8.0 ③
Output to:     ImageStreamTag echo:latest ④
...output omitted...
```

- ① 构建从 URL 属性中 Git 存储库的 `docker-build` 分支开始。
- ② 构建仅提取 URL 属性中 Git 存储库的 `ubi-echo` 文件夹。
- ③ 构建提取 Dockerfile 中指向父镜像的镜像流，以便新构建可以由镜像更改触发。
- ④ 构建将生成一个新的容器镜像，并通过镜像流将它推送到内部注册表。

4.2. 查看镜像流：

```
[student@workstation D0288-apps]$ oc describe is echo
Name:           echo
...output omitted...
Labels:         app=echo
...output omitted...
Image Repository: image-registry.openshift-image-registry.svc:5000/developer-
docker-build/echo①
...output omitted...
latest
no spec tag

* image-registry.openshift-image-registry.svc:5000/developer-docker-build/
echo@sha256:5bbf...ef0b ②
...output omitted...
```

- ① 镜像流指向使用服务 DNS 名称的 OpenShift 内部注册表。

- ② SHA256哈希可识别最新镜像。镜像流可以使用这个哈希来检测镜像是否有所变化。

#### 4.3. 检查部署：

```
[student@workstation D0288-apps]$ oc describe deployment echo
Name:           echo
...output omitted...
Labels:         app=echo
Annotations:   deployment.kubernetes.io/revision: 2
                image.openshift.io/triggers: ①
                  [{"from":{"kind":"ImageStreamTag","name":"echo:latest"},
                    "fieldPath":"spec.template.spec.containers[?(@.name==\"echo
                    \")].image"}]
...output omitted...
Pod Template:
  ...output omitted...
Containers:
  echo:
    Image:      image-registry.openshift-image-registry.svc:5000/developer-
               docker-build/echo@sha256:5bbf...ef0b ②
...output omitted...
Deployment #1 (latest):
  ...output omitted...
```

- ① 部署的镜像流中有一个触发器。如果镜像流发生变化，就会执行新的部署。  
② 部署配置中的容器集模板会指定容器镜像的SHA256哈希，以便支持各种部署策略（如滚动升级）。

### ▶ 5. 更改应用。

- 5.1. 编辑位于`~/D0288-apps/ubi-echo/Dockerfile`的Dockerfile中的CMD指令，以显示计数器。最终的Dockerfile内容应该如下所示：

```
FROM registry.access.redhat.com/ubi8/ubi:8.0
USER 1001
CMD bash -c "while true; do (( i++ )); echo test \$i; sleep 5; done"
```

- 5.2. 提交更改并推送到Git服务器。

```
[student@workstation D0288-apps]$ cd ubi-echo
[student@workstation ubi-echo]$ git commit -a -m 'Add a counter'
...output omitted...
[student@workstation ubi-echo]$ git push
...output omitted...
[student@workstation ubi-echo]$ cd ~
[student@workstation ~]$
```

### ▶ 6. 重新构建应用，并验证OpenShift是否部署了新的容器镜像。

- 6.1. 启动新的OpenShift构建：

```
[student@workstation ~]$ oc start-build echo  
build.build.openshift.io/echo-2 started
```

6.2. 按照新的构建日志进行操作，并等待构建完成：

```
[student@workstation ~]$ oc logs -f bc/echo  
...output omitted...  
Push successful
```

6.3. 验证 OpenShift 在构建完成后是否启动了新的部署：

```
[student@workstation ~]$ oc status  
...output omitted...  
dc/echo deploys istag/echo:latest <-  
bc/echo docker builds https://github.com/your-GitHub-user/D0288-apps#docker-  
build on istag/ubi:8.0  
deployment #3 running for 43 seconds - 1 pod  
deployment #2 deployed 26 minutes ago  
deployment #1 deployed 27 minutes ago  
...output omitted...
```

6.4. 等待新应用容器集准备就绪并在运行：

```
[student@workstation ~]$ oc get pod  
NAME        READY   STATUS    RESTARTS   AGE  
echo-1-build 0/1     Completed  0          27m  
echo-2-build 0/1     Completed  0          1m  
echo-pl1hg   1/1     Running   0          1m
```

6.5. 显示应用容器集日志，以表明它正在运行新的容器镜像。使用上一步中获得的容器集名称：

```
[student@workstation ~]$ oc logs echo-pl1hg | head -n 3  
test 1  
test 2  
test 3
```

## ► 7. 比较重新构建应用之前和之后的镜像流状态。

检查镜像流的当前状态：

```
[student@workstation ~]$ oc describe is echo  
Name: echo  
...output omitted...  
Labels: app=echo  
...output omitted...  
latest  
no spec tag  
  
* image-registry.openshift-image-registry.svc:5000/developer-docker-build/  
echo@sha256:025a...542f ①
```

```
2 minutes ago
image-registry.openshift-image-registry.svc:5000/developer-docker-build/
echo@sha256:5bbf...ef0b ②
...output omitted...
```

- ① 这是新的镜像。注意其 SHA256 哈希不同于旧镜像。
- ② 这是旧的镜像。

#### ► 8. 删除所有应用资源。

8.1. 使用带有 `oc new-app` 命令所生成的应用标签的 `oc delete` 命令：

```
[student@workstation ~]$ oc delete all -l app=echo
deployment.apps "echo" deleted
buildconfig.build.openshift.io "echo" deleted
build.build.openshift.io "echo-1" deleted
build.build.openshift.io "echo-2" deleted
imagestream.image.openshift.io "echo" deleted
imagestream.image.openshift.io "ubi" deleted
```

8.2. 验证项目中是否已没有任何资源。应从项目的单个应用中删除所有资源。如果输出显示有容器集处于 `Terminating` 状态，请重复上述命令，直至该容器集消失。

```
[student@workstation ~]$ oc get all
No resources found.
```

## 完成

在 `workstation` 上运行 `lab docker-build finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab docker-build finish
```

本引导式练习到此结束。

# 使用 Web 控制台管理应用

## 培训目标

学完本节后，您应能够使用 Web 控制台完成以下操作：

- 从二进制镜像部署应用，并管理应用资源。
- 查看容器集和构建日志。
- 编辑资源定义。

## OpenShift Web 控制台概述

OpenShift Web 控制台是一个基于浏览器的用户界面，可为大多数常见任务提供图形版替代工具，以满足管理 OpenShift 项目和应用的需要。Web 控制台提供的功能主要侧重于开发人员任务和工作流。Web 控制台不提供完整的集群管理功能。这些功能通常需要使用 `oc` 命令。

若要访问 Web 控制台，请使用 OpenShift API URL。如果集群中具有单一控制平面节点，这通常是指向该节点公共主机名的 HTTPS URL。

Web 控制台的首页会显示一个列表，列有当前用户可以访问的各个项目。在主页中，您可以创建新项目，删除现有项目，以及导航到项目概览页面。

Name	Display Name	Status	Requester
your-project	No display name	Active	youruser

图 1.16: Web 控制台项目列表

您的大部分时间应该都会花费在使用项目概览页面和众多的项目资源页面上。项目概览页面会显示与项目中的应用以及所有应用容器集状态有关的摘要信息。从项目概览页面中，您可以导航到资源详情页面，还可以向项目添加应用。

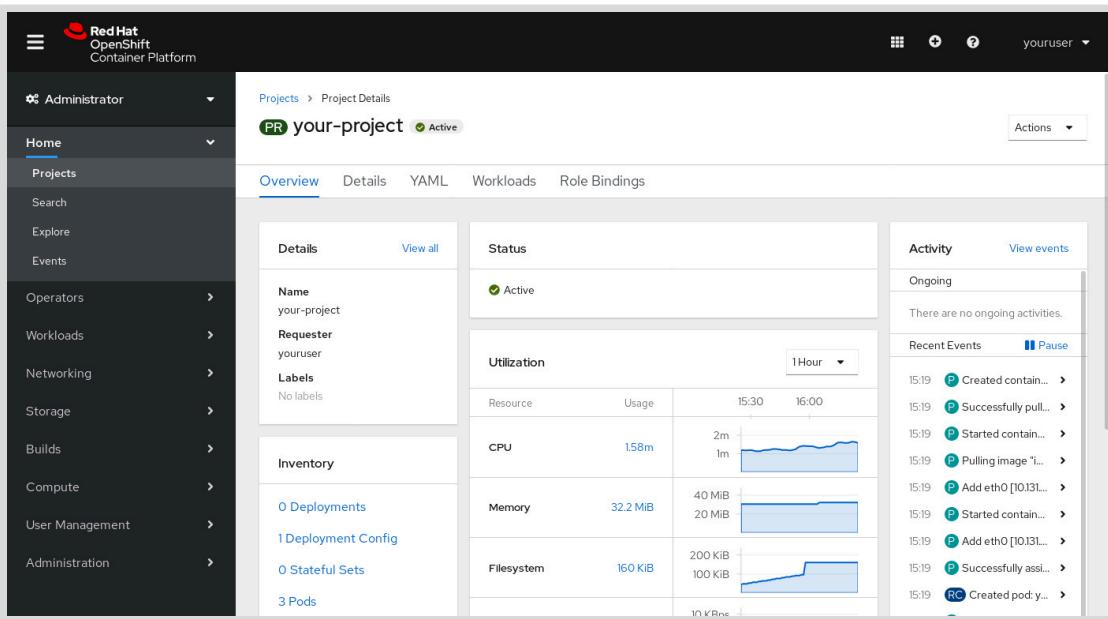


图 1.17: Web 控制台项目概览

## OpenShift 中的应用

OpenShift Web 控制台会将应用定义为一组资源，这些资源的值与 app 标签的值相同。oc new-app 命令可向其创建的所有应用资源添加这一标签。Developer 透视图中的 +Add 部分提供的功能类似于 oc new-app 命令，包括向资源添加 app 标签。

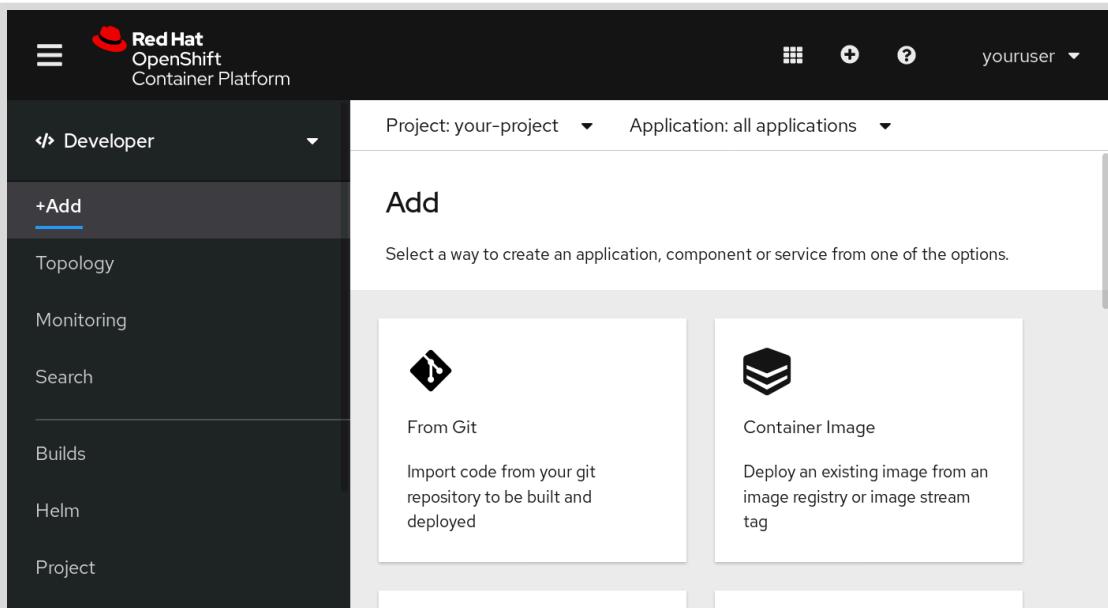


图 1.18: +Add 部分中提供的操作

通过 **Add** 部分，可以访问相应的助理，以协助您选取 S2I 构建器镜像、模板和容器镜像，作为特定项目的一部分向 OpenShift 集群中的项目部署应用。该助理会根据容器镜像中的标签以及模板和镜像流资源中的注释对目录中的镜像和模板进行分类。

## 资源详情页面

大多数资源详情页面都会列出指定类型的所有项目资源。它们会提供相应的链接，以用于删除特定资源并访问各个资源的详情页面。

单个资源的详情页面会通过多个选项卡提供每种资源类型的自定义状态信息。例如：

- 构建详情页面会显示每个构建的历史记录、配置、环境和日志。
- 部署详情页面会显示每个部署的历史记录、配置、环境、事件和日志。
- 服务详情页面会显示通过相应服务实现负载均衡的一组容器集，还会显示指向该项服务的路由（如有）。
- 容器集详情页面会显示每个容器集的状态、配置、环境、日志和事件。用户还可以通过这类详情页面打开终端会话，以运行容器集中任意容器内的 shell。

资源详情页面通常列有与相应资源关联的所有标签。OpenShift 会使用标签来记录资源间的关系。例如，由部署创建的所有容器集都有一个与部署同名的 **deployment** 标签。

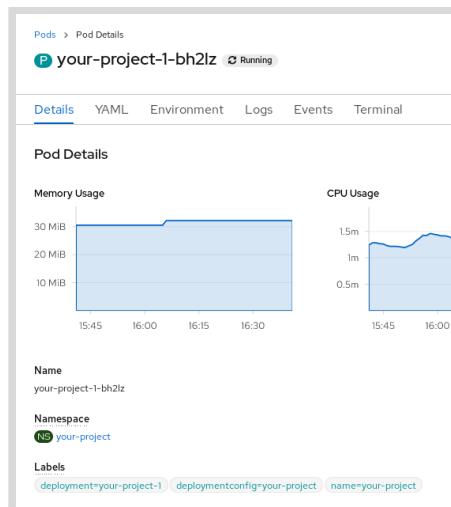


图 1.19: 容器集详情页面顶部的标签

通常，Web 控制台中显示的资源名称可链接到相应资源的详情页面。用户可通过 Web 控制台左侧的导航栏来访问所有受支持的资源类型的详情页面。该导航栏顶部的 home 图标则可用来访问项目列表页面。

## 使用 Web 控制台访问日志

Web 控制台中容器集详情页面包含一个 **Logs** 选项卡，用于显示来自容器集的日志。容器运行时会收集容器内容器的标准输出，并将这些内容存储为容器集日志。



### 注意

只有写入到容器内标准输出的日志可以通过 `oc logs` 命令或在 Web 控制台中查看。如果容器化应用将其日志事件保存在日志文件内，不论是临时容器存储还是持久卷上，OpenShift Web 控制台都不会显示这些日志，也无法通过 `oc logs` 命令查看。

**Logs** 选项卡会自动更新，并提供最新的日志条目。此选项卡还提供以下操作：

- 使用 **Download** 链接，下载日志并将其保存到本地文件中。

## 章 1 | 在 OpenShift 集群中部署和管理应用

- 使用 **Expand** 链接，全屏显示容器集日志以方便查看。

构建和部署操作由 OpenShift 使用构建器和部署程序容器集来执行。Build Details 页面捕获并存储构建器容器集的日志。使用此页面可查看这些构建器容器集日志。OpenShift 不会存储部署日志，除非部署期间发生错误。

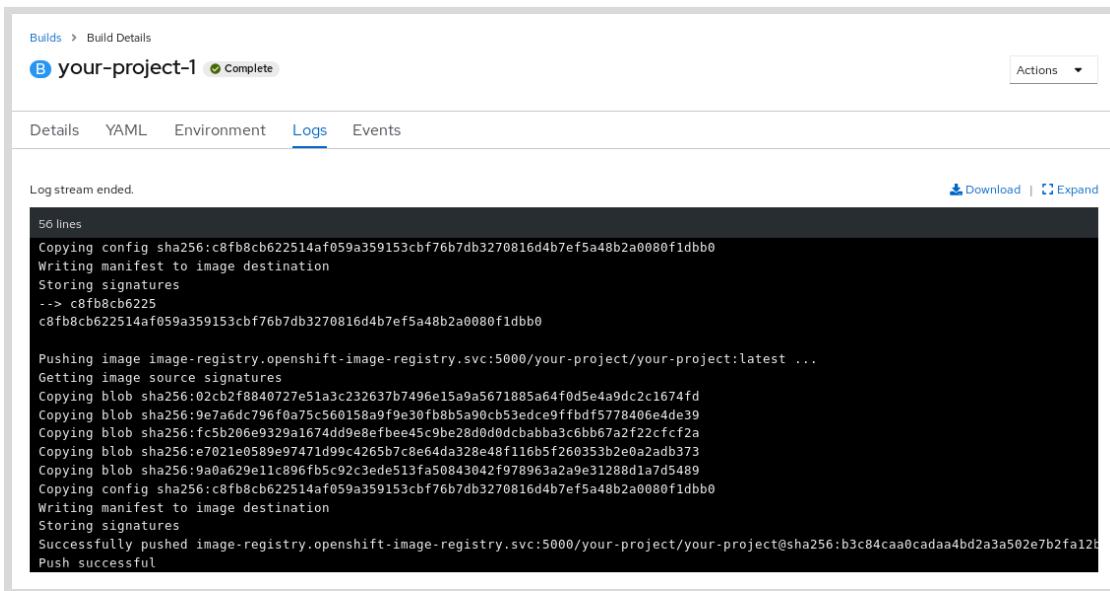


图 1.20: 构建详情页面的 Logs 选项卡

## 使用 Web 控制台管理构建和部署

OpenShift Web 控制台提供相应的功能，可用于管理构建和部署，以及各个配置所触发的所有构建和部署。其中的大部分配置都在针对特定于您要更新的构建配置或部署的详情页面上完成。

构建配置的详情页面提供 **Details**、**YAML**、**Builds**、**Environment** 和 **Events** 选项卡，以及含有 **Start Build** 操作的 **Actions** 按钮。该操作的功能与 `oc start-build` 命令相同。

如果应用源代码存储库未配置为使用 OpenShift Web hook，那么在将更新推送至应用源代码后，您需要使用 Web 控制台或 CLI 来触发新构建。

部署的详情页面上提供的功能丰富多样，包括可用于对部署的不同方面进行自定义的操作，如所需的容器集数或容器集存储。

部署的详情页面也提供 **Action** 按钮，此按钮含有的操作与构建配置详情页面中的不同。提供 **Start Rollout** 操作，其功能与 `oc rollout latest` 命令相同。您需要使用 CLI 来执行更为具体的 `oc rollout` 操作。

## 编辑 OpenShift 资源

OpenShift Web 控制台中的大部分资源详情页面都会提供一个 **Actions** 按钮，用于显示一个菜单。此菜单可能包含以下一些选项：

- Edit resource**: 在基于浏览器且具备语法高亮显示功能的文本编辑器中，使用原始 YAML 语法编辑现有资源。此操作等同于使用 `oc edit -o yaml` 命令。
- Delete resource**: 删除现有资源。此操作等同于使用 `oc delete` 命令。
- Edit Labels**: 打开模式对话框以编辑资源标签。
- Edit Annotations**: 打开模式对话框以编辑注释的键和值。

并非所有资源管理操作都可通过 Web 控制台来执行。尤其是集群管理员操作，此类操作通常需要使用 CLI。



### 参考文献

有关 Web 控制台整理、导航和使用的更多信息，请参阅红帽 OpenShift 容器平台 4.6 的 Web Console 指南，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/web\\_console/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/web_console/index)

## ► 指导练习

# 使用 Web 控制台管理应用

在本练习中，您将使用 OpenShift Web 控制台来部署 Apache HTTP 服务器容器镜像。

## 成果

您应能够使用 OpenShift Web 控制台进行以下操作：

- 创建新项目，并添加新应用来部署容器镜像。
- 执行常规故障排除任务，如查看日志、检查资源定义和删除资源。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 示例应用的容器镜像 (`redhattraining/php-hello-dockerfile`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件：

```
[student@workstation ~]$ lab deploy-image start
```

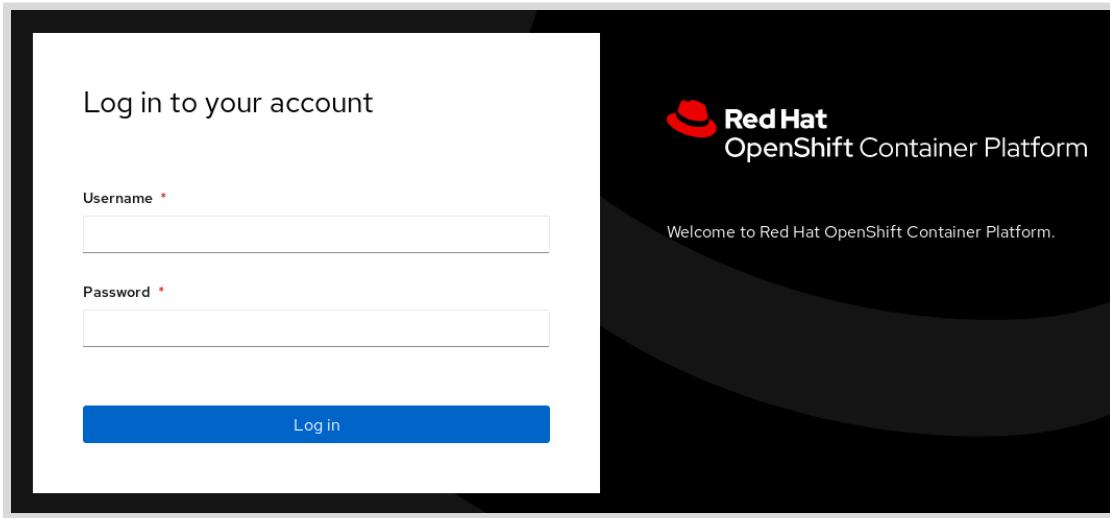
## 说明

- 1. 打开 Web 浏览器，访问 OpenShift Web 控制台。登录并创建名为 `developer-deploy-image` 的项目。

1.1. 查找您的 OpenShift Web 控制台 URL。登录 OpenShift，再使用 `oc` 命令查找 URL。

```
[student@workstation ~]$ oc login -u developer -p developer \
https://api.ocp4.example.com:6443
...output omitted...
[student@workstation ~]$ oc whoami --show-console
https://console.openshift-console.apps.ocp4.example.com
```

- 1.2. 打开 Web 浏览器并导航到 `https://console.openshift-console.apps.ocp4.example.com`，以访问 OpenShift Web 控制台。在 `Log in with` 提示符处，选择 `htpasswd_provider`。
- 1.3. 以 `developer` 用户身份并使用密码 `developer` 进行登录。



- 1.4. 导航到管理员透视图，然后从左侧菜单中选择 Home > Projects。用户首次进入时所处的页面。单击 Create Project。在 Create Project 对话框中的 Name 字段输入 **deploy-image**。无需填写显示名称和描述字段。

单击 **Create** 可创建新项目。

- 2. 从预构建的容器镜像创建一个新应用，该镜像含有使用 PHP 编写的“Hello, World”应用，然后创建一个公开应用的路由。

- 2.1. 在 Workloads 选项卡中，单击 Add page。

The screenshot shows the 'Project details' page for the 'deploy-image' project. The 'Workloads' tab is active. A central message says 'No resources found'. Below it, a link 'Start building your application or visit the [Add page](#) for more details.' is highlighted with a red box.

- 2.2. 单击 Container images。

The screenshot shows the 'Add' page in the OpenShift web interface. At the top, it says 'Project: deploy-image'. Below that, there's a 'Getting started resources' section with three cards: 'Create applications using samples', 'Build with guided documentation', and 'Explore new developer features'. Under 'Create applications using samples', there are links for 'Basic Quarkus' and 'Basic Spring Boot'. Under 'Build with guided documentation', there are links for 'Get started with Quarkus using s2i' and 'Get started with Spring'. Under 'Explore new developer features', there are links for 'Discover certified Helm Charts' and 'Start building your application quickly in topology'. Below this, there are three cards in the 'Developer Catalog': 'Developer Catalog' (with 'All services' link), 'Git Repository' (with 'Import from Git' link), and 'Container images' (which is highlighted with a red box). The 'Container images' card has a sub-description: 'Deploy an existing Image from an Image registry or Image stream tag'.

- 2.3. 在 Deploy Image 页面上的 Image Name 字段输入 `quay.io/redhattraining/php-hello-dockerfile`。OpenShift Web 控制台连接到红帽 Quay.io 公共注册表，并检索容器镜像的相关信息。
- 2.4. 向下滚动以查看镜像信息，并在 Application name 和 Name 字段将 `php-hello-dockerfile` 替换成 `hello`。  
取消勾选页面底部 Advanced options 部分的 Create a route to the Application 复选框。
- 2.5. 单击 Create 可创建应用。Web 控制台将创建部署容器镜像所需的所有 OpenShift 资源，并切换到 Topology 页面。  
Web 控制台显示部署列表。`hello` 部署具有 **1 of 1 Pod** 文本，用于描述部署容器集的状态。
- 2.6. 在导航栏中，单击 Networking > Routes。  
在 Routes 页面，单击 Create Route 按钮。  
在 Name 字段中输入 `hello-route`。在 Hostname 字段中输入 `hello.apps.ocp4.example.com`。  
向下滚动，并从 Service 列表中选择 `hello` 服务。从 Target Port 列表中选择 `8080 → 8080 (TCP)`。不要改动其他字段。向下滚动并单击 Create。
- 2.7. 路由详情页面会更改为显示一个可通过新路由访问应用的 URL。

Project: deploy-image ▾

Routes > Route details

**RT hello-route** ✓ Accepted

**Details** Metrics YAML

**Route details**

Name	hello-route	Location	http://hello.apps.ocp4.example.com
Namespace	NS deploy-image	Status	✓ Accepted
Labels	ann=hello ann.kubernetes.io/component=hello	Host	hello.apps.ocp4.example.com

单击 <http://hello.apps.ocp4.example.com> 可打开一个新浏览器标签页，其中显示 PHP 应用会返回的默认页面。它是一条采用 PHP 版本的简单“Hello, World”消息。

### ▶ 3. 探索 Web 控制台的故障排除功能。

#### 3.1. 查看应用容器集的日志。

在导航栏中，单击 Workloads > Pods。

Red Hat OpenShift Container Platform

Administrator ▾

Home >

Operators >

Workloads ▾

Pods

Project: your-user-deploy-image ▾

**Pods**

Name	Status	Ready	Restarts	Owner	Memory	CPU	Created
hello-7798f56475-vlw9s	Running	1/1	0	hello-7798f56475	613 MB	0.001 cores	Jun 8, 2:07 pm

Create Pod

单击应用容器集名称，如 `hello-7798f56475-vlw9s`，以显示 Pod Details 页面。单击 Logs 选项卡，以查看容器集日志。系统会显示与服务器全限定域名有关的警告消息，但可安全地忽略。

Red Hat OpenShift Container Platform

Administrator ▾

Home >

Projects

Search

Explore

Events

Operators >

Workloads ▾

Pods

Project: your-user-deploy-image ▾

**Pods > Pod Details**

**hello-7798f56475-vlw9s** ✓ Running

Actions ▾

Details YAML Environment Logs Events Terminal

Log streaming... hello

4 lines

```
[08-Jun-2021 18:07:23] NOTICE: [pool www] "user" directive is ignored when FPM is not running as root
[08-Jun-2021 18:07:23] NOTICE: [pool www] "group" directive is ignored when FPM is not running as root
AH00558: httpd: Could not reliably determine the server's fully qualified domain name, using 10.131.0.245. Set the 'ServerName' directive globally to suppress this message
```

Raw | Download | Expand

#### 3.2. 启动正在运行的容器中的某个 shell 会话。

继续在 Pod Details 页面单击 Terminal 选项卡以打开一个连接至应用容器集中单个容器的远程 shell。如果终端窗口太小，请单击 Expand 以隐藏 Web 控制台导航面板。终端只能执行应用的容器镜像内存在的命令。`ps` 命令不可用，但您可以查看 Apache HTTP 服务器访问日志。

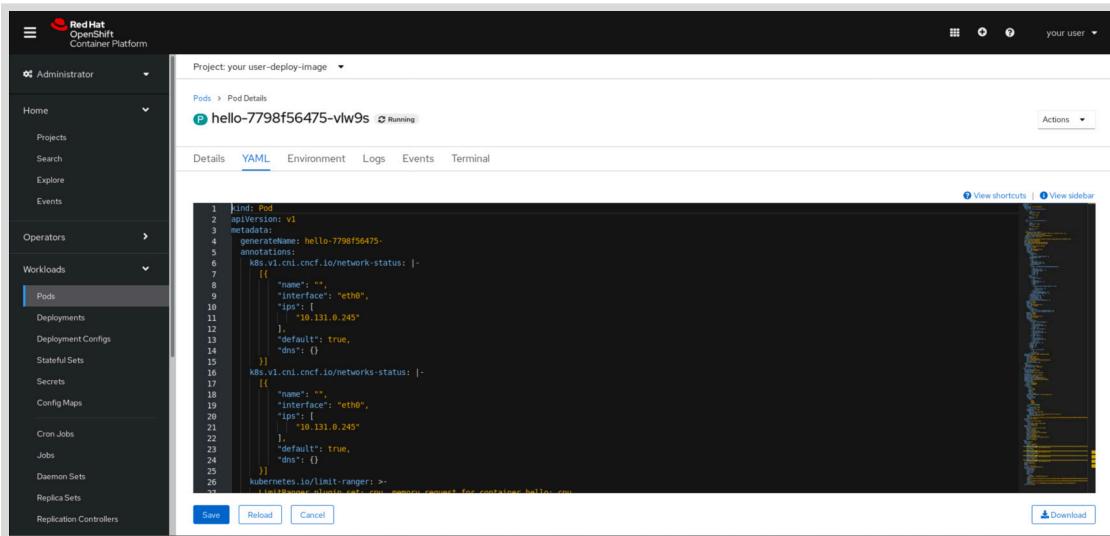
## 章 1 | 在 OpenShift 集群中部署和管理应用

```
sh-4.4$ ps ax
sh: ps: command not found
sh-4.4$
sh-4.4$ cat /var/log/httpd/access_log
10.131.0.1 - - [04/Aug/2020:08:39:57 +0000] "GET / HTTP/1.1" 200 36 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0"
10.131.0.1 - - [04/Aug/2020:08:39:57 +0000] "GET /favicon.ico HTTP/1.1" 404 209 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:68.0) Gecko/20100101 Firefox/68.0"
sh-4.4$
sh-4.4$ cat /var/log/php-fpm/error.log
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'user' directive is ignored when FPM is not running as root
[04-Aug-2020 08:10:59] NOTICE: [pool www] 'group' directive is ignored when FPM is not running as root
[04-Aug-2020 08:10:59] NOTICE: fpm is running, pid 9
[04-Aug-2020 08:10:59] NOTICE: ready to handle connections
[04-Aug-2020 08:10:59] NOTICE: systemd monitor interval set to 10000ms
sh-4.4$ 
```

如有需要，可单击 **Collapse** 重新显示 Web 控制台导航面板。

### 3.3. 查看资源定义。

继续在 Pod Details 页面中，单击 YAML 选项卡。



选项卡中包含适用于 YAML 语法的 Web 型富文本编辑器。不要进行任何更改，单击 **Cancel** 以退出编辑器。

## ▶ 4. 删除项目中的资源。

- 在 Pod Details 页面，单击 Actions > Delete Pod。在确认对话框中，单击 **Delete** 以删除正在运行的容器集。Web 控制台会显示 Pods 页面。

等到 Pods 页面显示部署配置所创建的新容器集，以取代已删除的容器集：

- 在导航栏中，单击 Workloads > Deployment 可查看 Deployment 页面。单击 hello 可查看部署详情页面。

单击右上角的 Actions > Delete Deployment。在确认对话框中，保持选中复选框，单击 **Delete** 可删除部署。

- 在导航栏上，单击 Networking > Services 查看项目中是否仍存在服务。

单击 hello 可访问 Service Details 页面。单击右上角的 Actions > Delete Service。在确认对话框中，单击 **Delete** 可删除该服务。

- 在导航栏中，单击 Networking > Routes 并注意项目中是否仍存在路由。

单击 hello-route 可访问 Route Details 页面。单击右上角的 Actions > Delete Route。在确认对话框中，单击 **Delete** 可删除该路由。

## ▶ 5. 删除项目。

单击左上角的 Home > Projects 可查看 Projects 页面。单击 **deploy-image** 项目右侧的菜单图标，然后单击 Delete Project。在确认对话框中输入 **deploy-image**，然后单击 Delete 可删除该项目。

等待 **deploy-image** 项目从 Projects 页面消失。您无需像上一步一样逐一删除各个应用资源。如果删除某个项目，则会删除该项目中的所有资源。

## 完成

在 **workstation** 上运行 **lab deploy-image finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab deploy-image finish
```

本引导式练习到此结束。

# 使用 CLI 管理应用

---

## 培训目标

学完本节后，您应能够：

- 通过源代码部署应用，并使用命令行界面管理应用资源。
- 描述管理项目和集群所需的 OpenShift 角色。
- 描述源至镜像如何为应用确定构建器镜像。

## OpenShift 集群和项目管理特权

OpenShift 集群管理涉及的许多任务都需要特殊的管理特权。如果您是 OpenShift 集群的集群管理员，则您有权访问该集群，但您通常没有这一级别的访问权限。

更为常见的是，OpenShift 集群为许多不同的用户服务，这些用户可能来自多个组织。这样的多节点集群为用户提供不同的访问权限级别：集群管理员、项目管理员和开发人员。

OpenShift 集群的默认配置允许任意用户创建新项目。用户会自动成为自己所创建项目的项目管理员。集群管理员可以更改 OpenShift 集群的权限，以禁止用户创建项目。

在这种情况下，只有集群管理员可以创建新项目。然后，集群管理员会为其他用户分配项目管理员和开发人员特权。

以下列表概述了各个访问级别的用户可以执行的任务：

### 集群管理员

管理项目、添加节点、创建持久卷、分配项目配额，以及执行其他集群范围的管理任务。

### 项目管理员

管理项目中的资源、分配资源限额，并且向其他用户授予查看和管理项目资源的权限。

### 开发人员

管理项目资源的一个子集。该子集包括构建和部署应用所需的各种资源，如构建和部署配置、持久卷声明、服务、机密和路由。开发人员无法针对这些资源向其他用户授予任何权限，也无法管理大多数的项目级资源（如资源限制）。

本课程中的大部分实践活动中都会以 **developer** 用户的身份来开展，开发人员会获得其所创建项目的项目管理员权限。如果某项活动需要使用集群管理员特权，则该活动要么旨在介绍如何作为拥有集群管理员特权的用户来登录，以按要求执行管理任务，要么预配置了管理员资源。



### 注意

OpenShift Enterprise Administration I (DO280) 课程介绍了如何执行管理任务，以及如何向用户分配集群和项目管理员权限。

## 使用 CLI 对构建、部署和容器集进行故障排除

OpenShift 提供了三种主要机制，用于获取有关项目及其资源的故障排除信息：

## 状态信息

`oc status` 和 `oc get` 等命令可以提供与项目中的资源相关的摘要信息。使用这些命令来获取关键信息，比如：构建是否以失败告终，或者，容器集是否已准备就绪并在运行。

## 资源描述

`oc describe` 命令可以显示资源的相关详细信息，包括其当前状态、配置和最近发生的事情。同时使用 `-o` 选项和 `oc get` 命令则可显示与资源相关的完整低级别配置和状态信息。请使用这些命令来检查资源，并确定 OpenShift 能否检测到与资源相关的任何特定错误条件。

## 资源日志

可运行的资源，如容器集和构建，可以使用 `oc logs` 命令来存储可供查看的日志。这些日志由容器集中运行的应用或由构建流程来生成。请使用这些命令来检索特定于应用的任意错误消息，并获取构建错误的相关详细信息。

如果以上机制无法提供足够的信息，您可以使用 `oc cp` 和 `oc rsh` 命令来直接与容器化应用进行交互。

## 比较可用于描述 OpenShift 资源的两个命令

`oc describe` 命令可以跟踪资源间的关系。例如，通过描述构建配置，可以显示最新构建的相关信息。`oc get -o` 命令只会显示与所请求资源相关的信息。例如，针对构建配置运行 `oc get -o` 命令后，不会显示与最近的构建相关的信息。

使用 `oc edit` 命令可以更改 OpenShift 资源。`oc edit` 命令会检索资源描述、运行 `oc get -o` 命令，使用文本编辑器打开输出文件，然后通过 `oc apply` 命令实施更改。

## 改进容器化应用日志

OpenShift 存储的应用日志的可用性取决于应用容器镜像的设计。正常情况下，容器化应用会将所有日志输出发送到标准输出。如果这类应用像非容器化应用一样将日志输出发送到日志文件，那么这些日志会被保留在容器临时存储中，并且会在应用容器集终止后丢失。

OpenShift 还会基于 EFK 堆栈（Elasticsearch、Fluentd 和 Kibana）提供可选的日志记录子系统。这个日志记录子系统可以为 OpenShift 集群节点和应用日志提供长期存储和搜索功能。应用可能会被设计为充分利用 OpenShift 日志记录子系统，或被设计为将其日志输出发送到标准输出并让 EFK 堆栈收集和处理其日志。

安装和配置 OpenShift 日志记录子系统不在本课程的讨论范围内。

## 读取构建日志

特定构建的构建日志可通过两种方式来检索：您可以引用构建配置或构建资源，也可以引用构建容器集。

以下示例使用了一个名为 `myapp` 的构建配置：

```
[user@host ~]$ oc logs bc/myapp
```

构建配置的日志就是最新构建（无论成功与否）的日志。

这个示例使用了同一构建配置中的第二个构建：

```
[user@host ~]$ oc logs build/myapp-2
```

这个示例使用了为执行同一构建而创建的构建容器集：

```
[user@host ~]$ oc logs myapp-build-2
```

## 获取容器化应用直接访问权限

如果应用将其日志存储在临时的容器存储中，请使用 `oc cp` 和 `oc rsync` 命令来检索这些日志文件。这些命令可用于检索正在运行的容器文件系统中的任意文件，如容器化应用的配置文件。

对于 `oc cp` 和 `oc rsync` 命令，您都必须使用容器文件系统上的远程文件路径。您可以将这些文件存储在临时的容器存储中，或者存储在容器挂载的持久卷中。

例如，要检索存储在名为 `frontend` 的应用容器集中的 Apache HTTP 服务器错误日志，请使用以下命令：

```
[user@host ~]$ oc cp frontend-1-zvjhb:/var/log/httpd/error_log \
/tmp/frontend-server.log
```

在默认情况下，`oc cp` 命令会复制整个文件夹。如果来源参数是单个文件，那么目的地参数也必须是单个文件。不同于 UNIX `cp` 命令，`oc cp` 命令无法将源文件复制至目标文件夹。

`oc cp` 命令需要基础应用容器镜像提供 `tar` 命令，以便能正常发挥作用。如果应用容器内没有安装 `tar`，那么 `oc cp` 会失败。

相同的命令还可用于将文件复制到容器文件系统。请使用这一功能在正在运行的容器中执行快速测试。请勿使用这一功能来永久修复问题。要想修复容器中的问题，建议您为容器镜像和应用资源应用修复程序，然后部署新的应用容器集。

`oc rsync` 命令可使本地文件夹与正在运行的容器中的远程文件夹同步。它会使用本地 `rsync` 命令来减小带宽使用量，但无需容器镜像提供 `rsync` 或 `ssh` 命令。

如果通过检索文件无法为正在运行的容器排除故障，`oc rsh` 命令便会创建远程 shell 以执行容器中的命令。该命令会使用 OpenShift 主控机 API 来创建通向远程容器集的安全隧道，但不会使用 `ssh` 或 `rsh` UNIX 命令。

以下示例演示了如何运行名为 `frontend` 的容器集中的 `ps ax` 命令：

```
[user@host ~]$ oc rsh frontend-1-zvjhb ps ax
```



### 注意

很多容器镜像都不包含常用的 UNIX 故障排除命令，如 `ps` 和 `ping`。`oc rsh` 命令只能运行远程容器提供的命令。

通过在 `oc rsh` 命令中添加 `-t` 选项，可以在容器内启动交互式 shell 会话：

```
[user@host ~]$ oc rsh -t frontend-1-zvjhb
```



### 注意

`oc rsh` 命令显示的 shell 提示取决于容器镜像提供的 shell。

## 构建和部署环境变量

很多容器镜像都希望用户定义环境变量，以提供配置信息。例如，红帽容器目录中的 MySQL 数据库镜像需要使用 `MYSQL_DATABASE` 变量来提供数据库名称。

请在 `oc new-app` 命令中添加 `-e` 选项，以提供环境变量的值。这些值会存储在部署配置中，并添加到部署创建的所有容器集中。

源至镜像 (S2I) 构建器镜像还可接受来自环境变量的配置参数。例如，Node.js 应用通常需要一个 npm 存储库，即 Node.js 的主要软件包管理器，以下载应用所需的 Node.js 依赖项。因此，容器目录中的 Node.js S2I 构建器接受通过 `npm_config_registry` 或 `NPM_MIRROR` 变量提供 URL，S2I 构建器可以在其中找到所需的 npm 模块存储库，以检索所需的 Node.js 依赖项。



### 注意

`npm` 命令由 Node.js S2I 构建器镜像执行，需要您提供 `npm_config_registry` 环境变量的值。

Node.js S2I 构建器镜像中的 `assemble` 脚本（调用 `npm` 命令）要求提供 `NPM_MIRROR` 环境变量的值。

S2I 构建器镜像变量可用于避免将配置信息与应用源存储在 Git 存储库中。不同的环境可能需要不同的配置，例如：

- 开发环境会使用一个 npm 存储库服务器，以供开发人员安装新模块。
- QA 环境会使用另一个 npm 存储库服务器，以便安全团队在模块提升到更高的环境之前审查这些模块。

您可以使用 `oc new-app` 命令的 `-e` 选项，为应用容器集定义环境变量。对于构建器容器集，您可以使用 `oc new-app` 命令的 `--build-env` 选项来定义环境变量。

请注意，部署配置会存储应用容器集的环境变量，构建配置则会存储构建器容器集的环境变量。请参见各个构建器镜像的文档，以查找有关其构建变量和变量默认值的信息。



### 参考文献

如需更多信息，请参阅红帽 OpenShift 容器平台 4.6 的 Images 指南中的 Understanding Containers, Images, and Imagestreams 章节，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/images/index#understanding-images](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/images/index#understanding-images)

## ► 指导练习

# 使用 CLI 管理应用

在本练习中，您将基于模板部署一个由多个容器集组成的容器化应用。您还将针对一个部署错误进行故障排除，并修复该错误。

## 成果

您应能够：

- 从自定义模板创建应用。模板可基于 PHP 源代码部署应用容器集，还可基于 MySQL 服务器容器镜像部署数据库容器集。
- 使用 OpenShift CLI 确定部署错误的根本原因。
- 使用 OpenShift CLI 修复错误。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 自定义模板所需的 S2I 构建器镜像和数据库镜像（PHP 7.2 和 MySQL 5.7）。
- Git 存储库中的示例应用 ([quotes](#))。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab build-template start
```

## 说明

### ► 1. 查看 Quotes 应用源代码。

- 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `main` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 该应用由两个 PHP 页面组成：

```
[student@workstation D0288-apps]$ ls ~/D0288-apps/quotes
get.php  index.php
```

欢迎页面 (`index.php`) 会显示应用的简介。它会链接到另一个页面 (`get.php`)，以随机引用 MySQL 数据库中的内容。

- 查看用于访问该数据库的 PHP 代码：

```
[student@workstation D0288-apps]$ less ~/D0288-apps/quotes/get.php
<?php
    $link = mysqli_connect($_ENV["DATABASE_SERVICE_NAME"], $_ENV["DATABASE_USER"],
    $_ENV["DATABASE_PASSWORD"], $_ENV["DATABASE_NAME"]);
    if (!$link) {
        http_response_code(500);
        error_log("Error: unable to connect to database\n");
        die();
    }
...output omitted...
```

按 q 退出。

示例应用仅使用了标准 PHP 函数，没有使用框架。它使用环境变量来检索数据库连接参数，并会在出现错误时返回标准 HTTP 状态代码。

- ▶ 2. 检查位于 `~/D0288/labs/build-template/php-mysql-ephemeral.json` 的自定义模板。为节省空间，不列出自定义模板的所有内容。您无需对模板进行任何更改即可使用。以下列表回顾了模板中最最重要的部分：

- 2.1. 模板首先定义一个机密和一个路由：

```
{
  "kind": "Template",
  "apiVersion": "'template.openshift.io/v1",
  "metadata": {
    "name": "php-mysql-ephemeral", ①
  ...output omitted...
  "objects": [
    {
      "apiVersion": "v1",
      "kind": "Secret", ②
  ...output omitted...
    },
    "stringData": {
      "database-password": "${DATABASE_PASSWORD}",
      "database-user": "${DATABASE_USER}"
  ...output omitted...
    {
      "apiVersion": "route.openshift.io/v1",
      "kind": "Route", ③
  ...output omitted...
    "spec": {
      "host": "${APPLICATION_DOMAIN}",
      "to": {
        "kind": "Service",
        "name": "${NAME}"
  ...output omitted...
```

- ① 这类模板是标准 `cakephp-mysql-example` 模板的副本，包含特定于所删除框架的资源和参数。自定义模板适用于使用 MySQL 数据库的任何简单 PHP 应用。
- ② secret 可存储数据库登录凭据，并填充应用和数据库容器集中的环境变量。本书后续章节会介绍 OpenShift `secrets`。

- ③ 路由可用于从外部访问应用。

2.2. 模板定义 PHP 应用的资源。以下列表侧重于构建配置，省略了服务和镜像流资源：

```
...output omitted...
{
    "apiVersion": "build.openshift.io/v1",
    "kind": "BuildConfig", ①
...output omitted...
    "source": {
        "contextDir": "${CONTEXT_DIR}",
        "git": {
            "ref": "${SOURCE_REPOSITORY_REF}",
            "uri": "${SOURCE_REPOSITORY_URL}"
        },
        "type": "Git"
    },
    "strategy": {
        "sourceStrategy": {
            "from": {
                "kind": "ImageStreamTag", ②
                "name": "php:7.4-ubi8",
...output omitted...
```

- ① 构建配置可使用 S2I 流程从源代码构建和部署 PHP 应用。构建配置和相关资源与对 Git 存储库执行 `oc new-app` 命令创建的那些相同。
- ② 标准的 OpenShift 镜像流提供 PHP 运行时构建器镜像。

2.3. 模板定义 MySQL 数据库的资源。以下列表侧重于部署，省略了服务和镜像流资源：

```
...output omitted...
{
    "apiVersion": "apps/v1",
    "kind": "Deployment", ①
...output omitted...
    "containers": [
...output omitted...
        "name": "mysql",
        "ports": [
            {
                "containerPort": 3306
...output omitted...
        "volumes": [
            {
                "emptyDir": {}, ②
                "name": "data"
...output omitted...
        "triggers": [
...output omitted...
    "env": {
```

```
...output omitted...
      "image": "image-registry.openshift-image-
registry.svc:5000/openshift/mysql:8.0-el8", 3
...output omitted...
```

- ①** 部署可部署 MySQL 数据库容器。部署配置和相关资源与对数据库镜像执行 `oc new-app` 命令创建的那些相同。
- ②** 永久存储不支持该数据库。如果数据库容器集重启，所有数据都可能会丢失。
- ③** 该部署会使用内部镜像注册表中的镜像。

2.4. 最后，模板定义了几个参数。其中一些参数如下所列。您会用到更多的参数。

```
...output omitted...
  "parameters": [
    {
      "name": "NAME",
      "displayName": "Name",
      "description": "The name assigned to all of the app objects defined in
this template.",
      ...output omitted...
    {
      "name": "SOURCE_REPOSITORY_URL", 1
      "displayName": "Git Repository URL",
      "description": "The URL of the repository with your application source
code.",
      ...output omitted...
    {
      "name": "DATABASE_USER", 2
      "displayName": "Database User",
      ...output omitted...
```

- ①** 参数提供特定于应用的数据，如 Git 存储库 URL。
- ②** 模板的参数还包括数据库配置和连接凭据。

### ▶ 3. 安装自定义模板。

3.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

3.2. 使用您的开发人员用户名登录 OpenShift：

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

3.3. 搜索使用 PHP 和 MySQL 的模板。OpenShift 提供了一个基于 CakePHP 框架的模板。

## 章 1 | 在 OpenShift 集群中部署和管理应用

这个模板不适合用于创建 Quotes 应用，因为它会添加框架所需的资源和依赖项。本练习中提供了一个更简单的模板，即您在上一步中检查的模板。

```
[student@workstation D0288-apps]$ oc get templates -n openshift | grep php \
| grep mysql
cakephp-mysql-example      An example CakePHP application ...
cakephp-mysql-persistent    An example CakePHP application ...
```

3.4. 创建新项目来托管自定义模板：

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "developer-common" on server
"https://api.ocp4.example.com:6443".
...output omitted...
[student@workstation D0288-apps]$ oc create -f \
~/D0288/labs/build-template/php-mysql-ephemeral.json
template.template.openshift.io/php-mysql-ephemeral created
```

红帽建议您在共享项目中创建可重复使用的 OpenShift 资源，如镜像流和模板。

## ▶ 4. 使用自定义模板部署应用。

4.1. 创建新项目来托管应用：

```
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-build-template
Now using project "developer-build-template" on server
"https://api.ocp4.example.com:6443".
...output omitted...
```

4.2. 查看模板参数，以确定可能需要使用哪些参数以部署 Quotes 应用。阅读模板参数的描述：

```
[student@workstation D0288-apps]$ oc describe template php-mysql-ephemeral \
-n ${RHT_OCP4_DEV_USER}-common
Name:  php-mysql-ephemeral
...output omitted...
Parameters:
  Name:          NAME
  Display Name: Name
  Description:   The name assigned to all of the app objects defined in this
template.
  Required:      true
  Value:         php-app
...output omitted...
```

4.3. 检查 `create-app.sh` 脚本。它提供了 `oc new-app` 命令，该命令会使用自定义模板并提供部署 Quotes 应用所需的所有参数，所以您不必输入很长的命令：

```
[student@workstation D0288-apps]$ cat ~/D0288/labs/build-template/create-app.sh
...output omitted...
oc new-app --template ${RHT_OCP4_DEV_USER}-common/php-mysql-ephemeral \
```

```
-p NAME=quotesapi \
-p APPLICATION_DOMAIN=quote-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN} \
-p SOURCE_REPOSITORY_URL=https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
-p CONTEXT_DIR=quotes \
-p DATABASE_SERVICE_NAME=quotesdb \
-p DATABASE_USER=user1 \
-p DATABASE_PASSWORD=mypa55 \
--name quotes
```

#### 4.4. 运行 `create-app.sh` 脚本：

```
[student@workstation D0288-apps]$ ~/D0288/labs/build-template/create-app.sh
--> Deploying template "developer-common/php-mysql-ephemeral" for "developer-
common/php-mysql-ephemeral" to project developer-build-template
...output omitted...
--> Creating resources ...
secret "quotesapi" created
service "quotesapi" created
route.route.openshift.io "quotesapi" created
imagestream.image.openshift.io "quotesapi" created
buildconfig.build.openshift.io "quotesapi" created
deploymentconfig.apps.openshift.io "quotesapi" created
service "quotesdb" created
deploymentconfig.apps.openshift.io "quotesdb" created
--> Success
...output omitted...
```

#### 4.5. 根据应用构建日志采取相应措施：

```
[student@workstation D0288-apps]$ oc logs -f bc/quotesapi
Cloning "https://github.com/your-GitHub-user/D0288-apps" ...
...output omitted...
Push successful
```

#### 4.6. 等待应用和数据库容器集准备就绪并在运行。请注意，您的容器集的确切名称可能与以下输出中显示的不同：

```
[student@workstation D0288-apps]$ oc get pod
NAME                  READY   STATUS    RESTARTS   AGE
quotesapi-1-build     0/1     Completed  0          89s
quotesapi-7d76ff58f8-6j2gx   1/1     Running   0          28s
quotesdb-6b7ffcc649-ds1pq   1/1     Running   0          80s
```

记录应用和数据库容器集的名称（在输出示例中为 `quotesapi-7d76ff58f8-6j2gx` 和 `quotesdb-6b7ffcc649-ds1pq`）。您需要它们来执行后续步骤。

#### 4.7. 使用模板创建的路由来测试应用的 `/get.php` 端点。系统会返回一个 HTTP 错误代码：

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT      ...
quotesapi quote-developer.apps.ocp4.example.com   ...
[student@workstation D0288-apps]$ curl -si \
http://quote-$RHT_OCP4_DEV_USER.$RHT_OCP4_WILDCARD_DOMAIN/get.php
HTTP/1.1 500 Internal Server Error
...output omitted...
```

#### ▶ 5. 对应用与数据库容器集之间的连接性进行故障排除。

应用无法正常工作，但构建和部署流程都已成功完成。应用出错可能是由应用漏洞、缺少前提条件或配置错误而导致的。

作为故障排除的第一步，请验证应用容器集是否已连接到正确的数据库容器集。

- 5.1. 验证数据库服务是否已找到正确的数据库容器集。使用您从第 4.6 步获得的数据库容器集名称：

```
[student@workstation D0288-apps]$ oc describe svc quotesdb | grep Endpoints
Endpoints: 10.8.0.71:3306
[student@workstation ~]$ oc describe pod quotesdb-6b7ffcc649-dslpq | grep IP
IP: 10.8.0.71
...output omitted...
```

- 5.2. 验证数据库容器集登录凭据：

```
[student@workstation D0288-apps]$ oc describe pod quotesdb-6b7ffcc649-dslpq \
| grep -A 4 Environment
Environment:
  MYSQL_USER: <set to the key 'database-user' in secret 'quotesapi'>
  MYSQL_ROOT_PASSWORD: <set to the key 'database-password' in secret
'quotesapi'>
  MYSQL_PASSWORD: <set to the key 'database-password' in secret 'quotesapi'>
  MYSQL_DATABASE: phpapp
```

- 5.3. 验证应用容器集中的数据库连接参数。使用您从第 4.6 步获得的应用容器集名称：

```
[student@workstation D0288-apps]$ oc describe pod quotesapi-7d76ff58f8-6j2gx \
| grep -A 5 Environment
Environment:
  DATABASE_SERVICE_NAME: quotesdb
  DATABASE_NAME: phpapp
  DATABASE_USER: <set to the key 'database-user' in secret
'quotesapi'>
  DATABASE_PASSWORD: <set to the key 'database-password' in secret
'quotesapi'>
  Mounts:
```

请注意，两个容器集中用于提供数据库登录凭据的环境变量应该相互匹配：

- DATABASE\_NAME 等于的 MYSQL\_DATABASE 值。
- DATABASE\_USER 设置为 quotesapi 机密中 database-user 键的值。

- `DATABASE_PASSWORD` 设置为 `quotesapi` 机密中 `database-user` 键的值。
  - `DATABASE_SERVICE_NAME` 等于数据库服务名称，即 `quotesdb`。
- 5.4. 验证应用容器集能否连接到数据库容器集。PHP S2I 构建器镜像不提供常用的联网实用程序（如 `ping` 命令），但在这个案例中 `echo` 命令提供了有用的输出：

```
[student@workstation D0288-apps]$ oc rsh quotesapi-7d76ff58f8-6j2gx bash -c \
'echo > /dev/tcp/$DATABASE_SERVICE_NAME/3306 && echo OK || echo FAIL'
OK
```

上一命令的输出证明问题不在于网络连接。

## ▶ 6. 查看应用日志，以找出错误的根本原因并加以修复。

### 6.1. 查看应用日志。

```
[student@workstation D0288-apps]$ oc logs quotesapi-7d76ff58f8-6j2gx
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.129.0.143. Set the 'ServerName' directive globally to suppress
this message
...output omitted...
[Mon May 27 14:54:56.187516 2019] [php7:notice] [pid 52] [client
10.128.2.3:43952] SQL error: Table 'phpapp.quote' doesn't exist\n
10.128.2.3 - - [27/May/2019:14:54:51 +0000] "GET /get.php HTTP/1.1" 500 - "-"
"curl/7.29.0"
...output omitted...
```

可以安全地忽略与服务器名称有关的消息。如果日志条目前面带有 HTTP 错误代码，即表示存在 SQL 错误。存在 SQL 错误意味着，应用无法查询 `phpapp` 数据库中的 `quote` 表。

上一步骤表明这个数据库名称是正确的。所以，合理的结论就是：该表并未创建。本练习的文件（在 `~/D0288/labs/build-template` 文件夹内）中提供了用于填充数据库的 SQL 脚本。

### 6.2. 将 SQL 脚本复制到数据库容器集：

```
[student@workstation D0288-apps]$ oc cp ~/D0288/labs/build-template/quote.sql \
quotesdb-6b7ffcc649-ds1pq:/tmp/quote.sql
```

### 6.3. 在数据库容器集中运行 SQL 脚本：

```
[student@workstation D0288-apps]$ oc rsh -t quotesdb-6b7ffcc649-ds1pq
sh-4.2$ mysql -u$MYSQL_USER -p$MYSQL_PASSWORD $MYSQL_DATABASE < /tmp/quote.sql
...output omitted...
sh-4.2$ exit
[student@workstation D0288-apps]$
```

### 6.4. 访问应用，以验证它现在能否正常工作。您将会获得一条随机随机名言。记得使用来自第 4.6 步的路由主机名：

```
[student@workstation D0288-apps]$ curl -si \
http://quote-$RHT_OCP4_DEV_USER.$RHT_OCP4_WILDCARD_DOMAIN/get.php
HTTP/1.1 200 OK
...output omitted...
Always remember that you are absolutely unique. Just like everyone else.
```

您可能会看到随机产生的其他消息，但是，收到名言表明应用现在能正常工作。

#### ► 7. 清理。更改到您的主文件夹，并删除在本练习中创建的项目。

```
[student@workstation D0288-apps]$ cd ~
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-template
project.project.openshift.io "developer-build-template" deleted
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "developer-common" deleted
```

## 完成

在 workstation 上运行 `lab build-template finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab build-template finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 在 OpenShift 集群中部署和管理应用

在本实验中，您将基于源代码将一个应用部署到 OpenShift 集群。应用构建配置文件存在错误，您要进行故障排除并修复错误。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 使用源构建策略创建应用。
- 查看构建日志，以查找与构建错误有关的信息。
- 更新应用构建工具配置，以修复构建错误。
- 重新构建应用，并验证其是否已成功部署。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Node.js 16 应用的 S2I 构建器镜像
- Git 存储库中的应用 (**nodejs-helloworld**)。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还会下载用于检查实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab source-build start
```

## 要求

提供的应用以 JavaScript 编写，且使用 Node.js 运行时。它是一个基于 Express 框架的 hello, world 应用。根据以下要求，构建应用并将其部署至 OpenShift 集群：

- 应用代码从名为 **source-build** 的新分支部署。
- OpenShift 的项目名称为 **youruser-source-build**。
- OpenShift 的应用名称为 **greet**。
- 该应用应当可从默认路由访问：  
`greet-_youruser_-source-build.apps.cluster.domain.example.com`
- 包含应用目录源的 Git 存储库为：

## 章 1 | 在 OpenShift 集群中部署和管理应用

[https://github.com/\\_yourgithubuser\\_/DO288-apps/nodejs-helloworld。](https://github.com/_yourgithubuser_/DO288-apps/nodejs-helloworld)

- 构建应用所需的 Npm 模块可从以下位置获取：

<http://nexus-common.apps.cluster.domain.example.com/repository/nodejs>

使用 **npm\_config\_registry** 构建环境变量将这一信息传输至 Node.js 的 S2I 构建器镜像。

- 您可以使用 **python -m json.tool filename.json** 命令来识别 JSON 文件中的语法错误。

## 说明

- 导航到 **DO288-apps** Git 存储库的本地克隆，再从 **main** 分支创建一个名为 **source-build** 的新分支。将 **nodejs-helloworld** 文件夹中的应用部署到 OpenShift 集群中的 **youruser-source-build** 项目。
- 显示构建日志以识别构建错误，并检查应用源以确定根本原因。  
记住您可以使用 **python3 -m json.tool filename** 命令来验证 JSON 文件。
- 修复构建工具配置文件中的错误，并将所做的更改推送到 Git 存储库。
- 为应用启动新的构建，并验证应用是否已成功部署。
- 验证应用日志是否没有显示任何错误，是否将应用公开给外部访问，并且验证应用是否返回了 **Hello, world** 消息。

## 评估

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab source-build grade
```

## 完成

在 **workstation** 上运行 **lab source-build finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab source-build finish
```

本实验到此结束。

## ▶ 解决方案

# 在 OpenShift 集群中部署和管理应用

在本实验中，您将基于源代码将一个应用部署到 OpenShift 集群。应用构建配置文件存在错误，您要进行故障排除并修复错误。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 使用源构建策略创建应用。
- 查看构建日志，以查找与构建错误有关的信息。
- 更新应用构建工具配置，以修复构建错误。
- 重新构建应用，并验证其是否已成功部署。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Node.js 16 应用的 S2I 构建器镜像
- Git 存储库中的应用 (**nodejs-helloworld**)。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还会下载用于检查实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab source-build start
```

## 要求

提供的应用以 JavaScript 编写，且使用 Node.js 运行时。它是一个基于 Express 框架的 hello, world 应用。根据以下要求，构建应用并将其部署至 OpenShift 集群：

- 应用代码从名为 **source-build** 的新分支部署。
- OpenShift 的项目名称为 **youruser-source-build**。
- OpenShift 的应用名称为 **greet**。
- 该应用应当可从默认路由访问：  
`greet-_youruser_-source-build.apps.cluster.domain.example.com`
- 包含应用目录源的 Git 存储库为：

## 章 1 | 在 OpenShift 集群中部署和管理应用

[https://github.com/\\_yourgithubuser\\_/DO288-apps/nodejs-helloworld](https://github.com/_yourgithubuser_/DO288-apps/nodejs-helloworld)。

- 构建应用所需的 Npm 模块可从以下位置获取：

<http://nexus-common.apps.cluster.domain.example.com/repository/nodejs>

使用 **npm\_config\_registry** 构建环境变量将这一信息传输至 Node.js 的 S2I 构建器镜像。

- 您可以使用 **python -m json.tool filename.json** 命令来识别 JSON 文件中的语法错误。

## 说明

- 导航到 DO288-apps Git 存储库的本地克隆，再从 main 分支创建一个名为 **source-build** 的新分支。将 nodejs-helloworld 文件夹中的应用部署到 OpenShift 集群中的 **youruser-source-build** 项目。

1.1. 准备实验环境。

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 登录 OpenShift 并创建项目：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-source-build
...output omitted...
```

1.3. 进入 DO288-apps Git 存储库的本地克隆，并签出课程存储库的 **main** 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd DO288-apps
[student@workstation DO288-apps]$ git checkout main
...output omitted...
```

1.4. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation DO288-apps]$ git checkout -b source-build
Switched to a new branch 'source-build'
[student@workstation DO288-apps]$ git push -u origin source-build
...output omitted...
* [new branch]      source-build -> source-build
Branch source-build set up to track remote branch source-build from origin.
```

1.5. 从 Git 存储库中的源文件，创建一个新的应用。将应用命名为 **greet**。通过 **--build-env** 选项和 **oc new-app** 命令使用 npm 模块 URL 定义构建环境变量。

从 **/home/student/DO288/labs/source-build** 文件夹中的 **oc-new-app.sh** 脚本复制或执行命令：

```
[student@workstation D0288-apps]$ oc new-app --name greet \
--build-env npm_config_registry=\
http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
nodejs:16-ubi8~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#source-build \
\
--context-dir nodejs-helloworld
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "greet" created
buildconfig.build.openshift.io "greet" created
deployment.apps "greet" created
service "greet" created
--> Success
...output omitted...
```



### 注意

`npm_config_registry` 后面的等号 (=) 前后没有空格。这个构建环境变量的完整 `key=value` 对对于页面宽度而言过长。

2. 显示构建日志以识别构建错误，并检查应用源以确定根本原因。

记住您可以使用 `python3 -m json.tool filename` 命令来验证 JSON 文件。

2.1. 根据构建日志，采取相应措施：

```
[student@workstation D0288-apps]$ oc logs -f bc/greet
```

您应该会看到一条 JSON 解析错误消息：

```
...output omitted...
--> Installing all dependencies
npm ERR! code EJSONPARSE
npm ERR! file /opt/app-root/src/package.json
npm ERR! JSON.parse Failed to parse json
npm ERR! JSON.parse Unexpected string in JSON at position 241 while parsing '{'
npm ERR! JSON.parse   "name": "nodejs-helloworld",
npm ERR! JSON.parse   "vers"
npm ERR! JSON.parse Failed to parse package.json data.
npm ERR! JSON.parse package.json must be actual JSON, not just JavaScript.
...output omitted...
```

Node.js 构建器镜像不会指出 `package.json` 构建工具配置文件中的具体错误位置。

2.2. 使用 `python3 -m json.tool` 命令验证 JSON 文件：

```
[student@workstation D0288-apps]$ python3 -m json.tool \
nodejs-helloworld/package.json
```

您应该会看到以下错误消息：

```
Expecting : delimiter: line 12 column 15 (char 241)
```

- 2.3. 使用文本编辑器打开 ~/D0288-apps/nodejs-helloworld/package.json 构建工具配置文件，并查找语法错误。下方列出的部分内容显示 `express` 键后面缺少冒号 (:):

```
"dependencies": {  
    "express" "4.14.x"  
}
```

3. 修复构建工具配置文件中的错误，并将所做的更改推送到 Git 存储库。

- 3.1. 编辑 ~/D0288-apps/nodejs-helloworld/package.json 源文件，以便在 `express` 键后面添加一个冒号 (:)。

最终的文件内容应该如下所示：

```
"dependencies": {  
    "express": "4.14.x"  
}
```

- 3.2. 提交修复并推送到 Git 存储库：

```
[student@workstation D0288-apps]$ cd nodejs-helloworld  
[student@workstation nodejs-helloworld]$ git commit -a -m 'Fixed JSON syntax'  
...output omitted...  
[student@workstation nodejs-helloworld]$ git push  
...output omitted...  
[student@workstation nodejs-helloworld]$ cd ~  
[student@workstation ~]$
```

4. 为应用启动新的构建，并验证应用是否已成功部署。

- 4.1. 为 `greet` 应用启动新的构建，并根据其日志采取相应措施。等待构建正确完成：

```
[student@workstation ~]$ oc start-build --follow bc/greet  
build "greet-2" started  
...output omitted...  
Push successful
```

- 4.2. 验证新部署是否已启动：

```
[student@workstation ~]$ oc status  
...output omitted...  
svc/greet - 172.30.160.185:8080  
deployment/greet deploys istag/greet:latest <-  
    bc/greet source builds https://github.com/yourgithubuser/D0288-apps#source-  
    build on openshift/nodejs:16-ubi8  
    deployment #2 running for 23 seconds - 1 pod  
...output omitted...
```

- 4.3. 等待应用容器集准备就绪并在运行：

```
[student@workstation ~]$ oc get pod
NAME             READY   STATUS    RESTARTS   AGE
greet-1-build   0/1     Error     0          4m
greet-2-build   0/1     Completed  0          2m
greet-594d667bc4-2b9ch 1/1     Running   0          59s
```

5. 验证应用日志是否没有显示任何错误，是否将应用公开给外部访问，并且验证应用是否返回了 hello, world 消息。

5.1. 访问应用日志。您应该不会看到应用返回任何错误消息。

```
[student@workstation ~]$ oc logs greet-1-gf59d
...
Example app listening on port 8080!
```

5.2. 公开应用供外部访问：

```
[student@workstation ~]$ oc expose svc/greet
route.route.openshift.io/greet exposed
```

5.3. 获取 OpenShift 为新路由生成的主机名：

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
greet     greet-youruser-source-build.apps.cluster.domain.example.com ...
```

5.4. 使用 `curl` 命令以及上一步中的主机名，向应用发送 HTTP 请求。这将返回 hello, world 消息：

```
[student@workstation ~]$ curl \
http://greet-{RHT_OCP4_DEV_USER}-source-build.{RHT_OCP4_WILDCARD_DOMAIN}
Hello, World!
```

## 评估

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab source-build grade
```

## 完成

在 `workstation` 上运行 `lab source-build finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab source-build finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- RHOCP 提供了在红帽 CoreOS 和 Kubernetes 上运行的 PaaS 工具。
- OpenShift 支持利用 S2I 流程从应用源代码或直接从 Dockerfile 构建容器镜像。
- 构建和部署配置资源可自动化构建和部署流程，并能自动响应应用源代码的更改或对容器镜像的更新。
- **oc new-app** 命令可以自动检测 Git 存储库中的应用所用的源编程语言。它还提供了许多消除歧义的选项。
- 有助于对构建和部署进行故障排除的 **oc** 子命令有 **get**、**describe**、**edit**、**logs**、**cp** 和 **rsh**。
- 开发人员可能没有其开发环境的集群管理员特权，或者只拥有针对 OpenShift 集群中所有项目的子集的项目管理或编辑特权。

## 章 2

# 针对 OpenShift 设计容器化应用

### 目标

为应用选择应用容器化方法，并封装应用以在 OpenShift 集群上运行。

### 培训目标

- 选择适合的应用容器化方法。
- 使用高级 Dockerfile 指令构建容器镜像。
- 选择将配置数据注入应用的方法，并创建完成注入操作所需的资源。

### 章节

- 选择容器化方法（及测验）
- 使用高级 Containerfile 指令构建容器镜像（及引导式练习）
- 将配置数据注入应用（及引导式练习）

### 实验

针对 OpenShift 设计容器化应用

# 选择容器化方法

## 培训目标

学完本节后，您应能够选择适合的应用容器化方法。

## 选择构建方法

OpenShift 中的主要部署单元是容器镜像，也简单地称为镜像。容器镜像由应用及其所有依赖项构成，如共享库、运行时环境、解释器，等等。根据您计划在 OpenShift 集群上部署和运行的应用类型来创建容器镜像的多种方法：

### 容器镜像

在 OpenShift 外部构建的容器镜像可以直接部署在 OpenShift 集群上。在您已将应用打包为容器镜像的情况下，此方法很有用。如果第三方供应商已向您提供了经过认证且受支持的容器镜像，则也可使用此方法。您可以将第三方供应商构建的镜像部署至 OpenShift 集群。

### Dockerfile

在某些情况下，您会获得用于构建应用容器镜像的 Dockerfile。在这种情况下，您还可考虑使用其他选项：

- 您可以对该 Dockerfile 进行自定义，构建新的镜像以满足您的应用需求。如果更改幅度不大，而且您不想在镜像中添加过多的层，可使用此选项。
- 您可以使用提供的容器镜像作为父级创建 Dockerfile，并自定义基本镜像以满足您的应用需求。如果要创建具有更多自定义项的新子镜像，并且从父镜像继承图层，可使用此选项。

### 源至镜像 (S2I) 构建器镜像

S2I 构建器镜像包含基础操作系统库、编译器和解释器、运行时、框架和源至镜像工具。使用此方法构建应用时，OpenShift 将应用源代码和构建器镜像结合，以创建 OpenShift 随后可部署到集群上的可立即运行的容器镜像。对于开发人员而言，此方法具有多个优点，这也是构建新应用以部署到 OpenShift 集群上的最快方式。

根据您的应用需求，您可以通过多种方式来使用 S2I 构建器镜像：

- 红帽提供了多种受支持的 S2I 构建器镜像，以用于构建各类应用。红帽建议您尽量使用标准 S2I 构建器镜像。
- S2I 构建器镜像与普通容器镜像类似，但前者包含额外的元数据、脚本和工具。您可以使用 Dockerfile 基于红帽提供的父构建器镜像来创建子镜像。
- 如果红帽提供的标准 S2I 构建器镜像都不符合您的应用需求，则您可以构建自己的自定义 S2I 构建器镜像。

## S2I 构建器镜像

S2I 构建器镜像是一种特殊形式的容器镜像，它会生成并输出应用容器镜像。构建器镜像包含应用所基于的基础操作系统库、语言运行时、框架和库，以及各种源至镜像工具和实用程序。

例如，如果想要部署到 OpenShift 的应用是用 PHP 编写的，则可使用 PHP 构建器镜像来生成应用容器镜像。您要提供保存应用源代码的 Git 存储库的位置，然后 OpenShift 会将该源代码与基础构建器镜像进行整合，以生成 OpenShift 可部署到集群的容器镜像。所生成的应用容器镜像包含某一版本

的红帽企业 Linux、一个 PHP 运行时和相应的应用。构建器镜像是一种非常便捷的机制。采用该机制时，无需创建 Dockerfile，即可轻松快速地从代码生成可运行的容器。

## 使用容器镜像

虽然源至镜像构建是构建应用并将其部署至 OpenShift 的首选方式，但在某些情况下您需要部署第三方提供给您的预构建的应用。例如，某些供应商会提供已经过全面认证且可直接运行的受支持容器镜像。在这种情况下，OpenShift 支持部署预构建的容器镜像。

`oc new-app` 命令可以提供多种灵活方式，以向 OpenShift 集群部署容器镜像。最简单的方法就是，通过公共注册表（如 docker.io 或 quay.io）或私有注册表（您所在企业内部托管的注册表）获取预构建的镜像，然后向 `oc new-app` 命令提供该镜像的位置。接着，OpenShift 会提取该镜像并将其部署至 OpenShift 集群，就像部署在 OpenShift 中构建的任何其他镜像一样。



### 注意

您可从以下位置获取演示如何向 OpenShift 集群部署预构建容器镜像的示例教程：OpenShift 二进制部署 [https://blog.openshift.com/binary-deployments-openshift-3/]。

## 使用红帽容器目录

红帽容器目录是一个可信来源，您可从中获取经过认证的最新安全容器镜像。该目录中包含普通容器镜像和 S2I 构建器镜像。任务关键型应用需要使用可信容器。红帽通过经过红帽内部安全团队审查且针对安全漏洞进行过强化的 RPM 资源构建容器目录镜像。

位于 <https://registry.redhat.io> 的红帽容器目录门户可提供与基于各版红帽企业 Linux (RHEL) 和相关系统构建而成的众多容器镜像有关的信息。它还提供了很多可直接使用的容器镜像，您可以由此入手开始针对 OpenShift 开发应用。

红帽会使用 Container Health Index 来为通过红帽容器目录提供的容器镜像进行安全风险评估。有关红帽在 Container Health Index 中所用评分系统的更多详情，请参见 <https://access.redhat.com/articles/2803031>。

有关红帽容器目录的更多详情，请参见 常见问题 (FAQ) [<https://access.redhat.com/containers/#/faq>]

## 为应用选择容器镜像

应用容器镜像的选择取决于多个要素。如果您想构建自定义容器，请从基础操作系统镜像（如 `rhel7`）入手。为了构建和运行需要使用特定开发工具和运行时库的应用，红帽提供了包含 Node.js (`rhsc1/nodejs-8-rhel7`)、Ruby on Rails (`rhsc1/ror-50-rhel7`) 和 Python (`rhsc1/python-36-rhel7`) 等工具的容器镜像。

红帽 Software Collections Library (RHSC1)（或简写为 Software Collections）是红帽面向需要使用最新开发工具（但这些工具通常又不适合标准红帽企业Linux(RHEL)发行计划）的开发人员提供的解决方案。红帽将通过红帽容器目录提供的很多容器镜像作为 RHSC1 的一部分来进行维护。

红帽还提供红帽 OpenShift 应用运行时 (RHOAR)，这是红帽用于云原生和微服务应用的开发平台。RHOAR 为开发以 OpenShift 为目标部署平台的微服务应用提供红帽优化和支持的方法。

RHOAR 支持多个运行时、语言、框架和架构。它提供了选择和灵活性，可为正确的作业选择正确的框架和运行时。使用 RHOAR 开发的应用可以在红帽 OpenShift 容器平台可以运行的任何基础架构上运行，因此可以摆脱供应商锁定。

RHOAR 提供：

- 针对红帽为所选微服务开发框架和运行时构建和支持的二进制文件的访问权限。
- 针对红帽为集成模块构建和支持的二进制文件的访问权限，这些模块可替换或增强框架的微服务模式实施以使用 OpenShift 功能。
- 开发人员支持：使用选定的微服务开发框架、运行时、集成模块以及与选定外部服务（如数据库服务器）的集成编写应用。
- 生产支持：使用选定的微服务开发框架、运行时、集成模块以及集成将应用部署到支持的 OpenShift 集群上。

## 创建 S2I 构建器镜像

如果您希望您的应用将自定义 S2I 构建器镜像与您自己的自定义运行时、脚本、框架和库集合一起使用，则可以构建自己的 S2I 构建器镜像。S2I 构建器镜像可通过多种选项来创建：

- 从头开始创建您自己的 S2I 构建器镜像。如果您的应用无法使用容器目录按原样提供的 S2I 构建器镜像，则可构建自定义 S2I 构建器镜像，以通过自定义构建流程来满足您的应用需求。

OpenShift 提供了 **s2i** 命令行工具，帮助您启动用于创建自定义 S2I 构建器镜像的构建环境。该工具包含在 RHSCL Yum 存储库 (**rhel-server-rhscl-7-rpms**) 的 **source-to-image** 软件包中。

- 对现有 S2I 构建器镜像进行派生。无需从头开始创建。您可以使用适用于容器目录中的现有构建器镜像的 Dockerfile（可从 <https://github.com/sclorg/?q=s2i> 获取），然后再对其进行自定义以满足您的需求。
- 对现有 S2I 构建器镜像进行扩展。您还可以创建子镜像，然后向现有构建器镜像添加内容或替换其中的内容，从而对现有的构建器镜像进行扩展。

有关如何构建自定义 S2I 构建器镜像的逐步说明，请参见位于 <https://blog.openshift.com/create-s2i-builder-image/> 的教程。



### 参考文献

#### 红帽容器目录

<https://access.redhat.com/containers>

适用于红帽 Software Collections Library 所含镜像的 Dockerfile 可从 <https://github.com/sclorg?q=-container> 获取

如需有关红帽支持与 OpenShift 搭配使用的容器镜像的信息，请参见红帽 OpenShift 容器平台 4.10 Images 指南的 Creating Images 章节，网址为 [https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/index#creating-images](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/index#creating-images)

## ► 小测验

# 选择容器化方法

选择以下问题的正确答案：

完成测验后，单击 **CHECK**。如需重试，请单击 **RESET**。单击 **SHOW SOLUTION** 以查看所有正确答案。

- 1. 您需要将一个基于 .NET 的第三方商业应用部署至 OpenShift 集群，该应用被供应商封装成了容器镜像。您会使用以下哪个选项来部署该应用？
- a. 源至镜像构建。
  - b. 自定义源至镜像构建器。
  - c. 将容器镜像暂存在私有容器注册表中，然后使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。
  - d. 以上皆不是。您无法在 OpenShift 集群上部署基于.NET 的应用。
- 2. 您的任务是将内部开发的某个基于 RHEL 7 的自定义 C++ 应用部署至 OpenShift 集群。您会获得该应用的完整源代码。基于最佳实践来看，以下哪两个选项可用于构建要在 OpenShift 集群上进行部署的容器镜像？（请选择两项。）
- a. 使用红帽容器目录中的 **rhel7** 容器镜像作为该应用的基础来创建 Dockerfile，并通过 Git 存储库将其提供给 OpenShift。OpenShift 可从所提供的 Dockerfile 创建容器镜像。
  - b. 从公共注册表（如 docker.io）下载基于 CentOS 7/C++ 的容器镜像，然后创建一个 Dockerfile 以编译和封装该应用。由于基于 CentOS 7 的二进制文件都与 RHEL 7 二进制兼容，所以该应用在部署到 OpenShift 集群后将能正常工作。
  - c. 使用红帽容器目录中的 RHEL 7 容器镜像作为基础来创建 Dockerfile，然后基于该 Dockerfile 构建容器镜像。使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。
  - d. 使用公共注册表中任意基于 Linux 的容器镜像作为基础来创建 Dockerfile，然后基于该 Dockerfile 构建容器镜像。使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。

- 3. 您需将两个应用（分别基于 Ruby on Rails 和 Node.js）迁移并部署至 OpenShift 集群。这两个应用先前是在使用虚拟机的环境中运行的。您已经获得了这两个应用的应用源代码所在在的 Git 存储库的位置。如果需要进一步增强这两个应用的功能，并继续在 OpenShift 环境中进行开发，您会建议使用以下哪个选项将这些应用部署至 OpenShift 集群？
- a. 使用 docker.io 中的 Ruby on Rails 和 Node.js 容器镜像作为基础。为这两个应用分别创建自定义 Dockerfile，然后从所创建的 Dockerfile 构建容器镜像。按照标准二进制文件部署流程将镜像部署至 OpenShift 集群。
  - b. 创建基于 S2I 的自定义构建器镜像，因为 OpenShift 中没有适用于 Ruby on Rails 和 Node.js 的构建器镜像。
  - c. 使用红帽容器目录中的 Ruby on Rails 和 Node.js S2I 构建器镜像，按照标准 S2I 构建流程将应用部署至 OpenShift 集群。
  - d. 使用红帽容器目录中基于 Ruby 和 Node.js 的普通容器镜像，为这两个应用分别创建自定义 Dockerfile。构建由此得到的容器镜像，并将它们部署至 OpenShift 集群。
- 4. 您需将某个使用 Go 编程语言编写的 Web 应用部署至 OpenShift 集群。您所在企业的安全团队要求所有应用都要通过静态源代码分析系统来运行，并要求在部署至生产环境前先进行一系列自动单元和集成测试。该安全团队还提供了一个自定义 Dockerfile，以确保所有应用都部署在基于 RHEL 7 的操作系统（基于红帽容器目录中的标准 RHEL 7 镜像）中。其环境包括用于操作系统中的核心服务的一组精心挑选的软件包、用户和自定义配置。此外，应用架构师还坚持要明确区分源代码级更改与基础架构更改（操作系统、Go 编译器和 Go 工具）。当针对操作系统或 Go 运行时层做出更改和安装补丁时，都应自动触发应用的重新构建和重新部署。您会使用以下哪个选项来实现上述目标？
- a. 创建自定义 Dockerfile，以构建一个由 RHEL 7 操作系统基础、Go 运行时和分析工具组成的应用容器镜像。按照二进制文件部署流程将所形成的镜像部署至 OpenShift 集群。
  - b. 为 RHEL 7 操作系统基础、Go 运行时和分析工具分别创建单独的 Dockerfile。OpenShift 可以自动合并这些 Dockerfile，以形成单个可运行的应用容器镜像。
  - c. 为这个应用创建自定义 S2I 构建器镜像，以基于 Dockerfile 嵌入静态分析工具、Go 编译器和运行时以及 RHEL 7 操作系统镜像。
  - d. 为 RHEL 7 操作系统基础、Go 运行时和分析工具分别创建单独的容器镜像。在将这些镜像暂存到私有或公共容器镜像注册表后，OpenShift 可以自动连接各个镜像中的层，以创建最终的可运行应用容器镜像。
  - e. 以上皆不是。无法满足这一要求，且无法在 OpenShift 集群上部署此类应用。

## ► 解决方案

# 选择容器化方法

选择以下问题的正确答案：

完成测验后，单击 **CHECK**。如需重试，请单击 **RESET**。单击 **SHOW SOLUTION** 以查看所有正确答案。

- 1. 您需要将一个基于 .NET 的第三方商业应用部署至 OpenShift 集群，该应用被供应商封装成了容器镜像。您会使用以下哪个选项来部署该应用？
- a. 源至镜像构建。
  - b. 自定义源至镜像构建器。
  - c. 将容器镜像暂存在私有容器注册表中，然后使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。
  - d. 以上皆不是。您无法在 OpenShift 集群上部署基于.NET 的应用。
- 2. 您的任务是将内部开发的某个基于 RHEL 7 的自定义 C++ 应用部署至 OpenShift 集群。您会获得该应用的完整源代码。基于最佳实践来看，以下哪两个选项可用于构建要在 OpenShift 集群上进行部署的容器镜像？（请选择两项。）
- a. 使用红帽容器目录中的 **rhel7** 容器镜像作为该应用的基础来创建 Dockerfile，并通过 Git 存储库将其提供给 OpenShift。OpenShift 可从所提供的 Dockerfile 创建容器镜像。
  - b. 从公共注册表（如 docker.io）下载基于 CentOS 7/C++ 的容器镜像，然后创建一个 Dockerfile 以编译和封装该应用。由于基于 CentOS 7 的二进制文件都与 RHEL 7 二进制兼容，所以该应用在部署到 OpenShift 集群后将能正常工作。
  - c. 使用红帽容器目录中的 RHEL 7 容器镜像作为基础来创建 Dockerfile，然后基于该 Dockerfile 构建容器镜像。使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。
  - d. 使用公共注册表中任意基于 Linux 的容器镜像作为基础来创建 Dockerfile，然后基于该 Dockerfile 构建容器镜像。使用 OpenShift **oc** 命令行工具将该容器镜像部署至 OpenShift 集群。

- 3. 您需将两个应用（分别基于 Ruby on Rails 和 Node.js）迁移并部署至 OpenShift 集群。这两个应用先前是在使用虚拟机的环境中运行的。您已经获得了这两个应用的应用源代码所在在的 Git 存储库的位置。如果需要进一步增强这两个应用的功能，并继续在 OpenShift 环境中进行开发，您会建议使用以下哪个选项将这些应用部署至 OpenShift 集群？
- a. 使用 docker.io 中的 Ruby on Rails 和 Node.js 容器镜像作为基础。为这两个应用分别创建自定义 Dockerfile，然后从所创建的 Dockerfile 构建容器镜像。按照标准二进制文件部署流程将镜像部署至 OpenShift 集群。
  - b. 创建基于 S2I 的自定义构建器镜像，因为 OpenShift 中没有适用于 Ruby on Rails 和 Node.js 的构建器镜像。
  - c. 使用红帽容器目录中的 Ruby on Rails 和 Node.js S2I 构建器镜像，按照标准 S2I 构建流程将应用部署至 OpenShift 集群。
  - d. 使用红帽容器目录中基于 Ruby 和 Node.js 的普通容器镜像，为这两个应用分别创建自定义 Dockerfile。构建由此得到的容器镜像，并将它们部署至 OpenShift 集群。
- 4. 您需将某个使用 Go 编程语言编写的 Web 应用部署至 OpenShift 集群。您所在企业的安全团队要求所有应用都要通过静态源代码分析系统来运行，并要求在部署至生产环境前先进行一系列自动单元和集成测试。该安全团队还提供了一个自定义 Dockerfile，以确保所有应用都部署在基于 RHEL 7 的操作系统（基于红帽容器目录中的标准 RHEL 7 镜像）中。其环境包括用于操作系统中的核心服务的一组精心挑选的软件包、用户和自定义配置。此外，应用架构师还坚持要明确区分源代码级更改与基础架构更改（操作系统、Go 编译器和 Go 工具）。当针对操作系统或 Go 运行时层做出更改和安装补丁时，都应自动触发应用的重新构建和重新部署。您会使用以下哪个选项来实现上述目标？
- a. 创建自定义 Dockerfile，以构建一个由 RHEL 7 操作系统基础、Go 运行时和分析工具组成的应用容器镜像。按照二进制文件部署流程将所形成的镜像部署至 OpenShift 集群。
  - b. 为 RHEL 7 操作系统基础、Go 运行时和分析工具分别创建单独的 Dockerfile。OpenShift 可以自动合并这些 Dockerfile，以形成单个可运行的应用容器镜像。
  - c. 为这个应用创建自定义 S2I 构建器镜像，以基于 Dockerfile 嵌入静态分析工具、Go 编译器和运行时以及 RHEL 7 操作系统镜像。
  - d. 为 RHEL 7 操作系统基础、Go 运行时和分析工具分别创建单独的容器镜像。在将这些镜像暂存到私有或公共容器镜像注册表后，OpenShift 可以自动连接各个镜像中的层，以创建最终的可运行应用容器镜像。
  - e. 以上皆不是。无法满足这一要求，且无法在 OpenShift 集群上部署此类应用。

# 使用高级 Containerfile 指令构建容器镜像

## 培训目标

学完本节后，您应能够：

- 使用红帽通用基本镜像构建容器化应用。
- 使用高级 Containerfile 指令构建容器镜像。

## 红帽通用基本镜像简介

红帽通用基础镜像 (UBI) 旨在作为高质量、灵活的基础容器镜像，用于构建容器化应用。红帽通用基础镜像的目标是允许用户使用高度可支持的轻量级和高性能的企业级容器基本镜像构建和部署容器化应用。您可以在红帽平台和非红帽平台上运行使用通用基础镜像构建的容器。

红帽从红帽企业 Linux (RHEL) 派生出红帽通用基础镜像。它确实不同于现有的 RHEL 7 基础镜像，最显著的是，它可以根据红帽通用基础镜像最终用户许可协议 (EULA) 的条款重新分发，该协议允许红帽的合作伙伴、客户和社区成员对经过精心设计的企业软件和工具进行标准化，通过添加第三方内容来交付价值。

支持计划是让通用基础镜像遵循与基础 RHEL 内容相同的生命周期和支持日期。在订阅的 RHEL 或 OpenShift 节点上运行时，它遵循与基础 RHEL 内容相同的支持策略。红帽维护适用于 RHEL 7 的通用基础镜像（映射到 RHEL 7 内容），以及另一个适用于 RHEL 8 的 UBI（映射到 RHEL 8 内容）。

红帽建议将通用基础镜像用作新应用的基础容器镜像。红帽承诺在 RHEL 版本的支持生命周期内继续支持较早的 RHEL 基础镜像。

红帽通用基础镜像包括：

- 一组三个基础镜像 (**ubi**、**ubi-minimal** 和 **ubi-init**)。它们镜像是为使用 RHEL 7 基础镜像构建容器提供的内容。
- 一组语言运行时镜像 (**java**、**php**、**python**、**ruby**、**nodejs**)。这些运行时镜像使开发人员能够借助红帽构建和支持的容器镜像放心地开始开发应用。
- 一组关联的 Yum 存储库和通道，包括 RPM 软件包和更新。它们允许您添加应用依赖项并根据需要重新构建容器镜像。

## 通用基础镜像的类型

红帽通用基础镜像提供三个主要基础镜像：

### **ubi**

基于 RHEL 企业级软件包构建的标准基础镜像。适用于大多数应用用例。

### **ubi-minimal**

使用 **microdnf** 构建的一个最小基础镜像，这是 **dnf** 实用程序的缩减版。这提供了最小的容器镜像。

### **ubi-init**

此镜像可让您在单个容器中轻松运行多个服务，如 Web 服务器、应用服务器和数据库等。它允许您使用内置于 **systemd** 单元文件中的知识，而无需确定如何启动服务。

## 红帽通用基础镜像的优势

使用红帽通用基础镜像作为容器化应用的基础镜像具有以下几个优点：

- **最小尺寸：**通用基础镜像是一个相对最小（约 90-200 MB）的基本容器镜像，启动时间很快。
- **安全性：**使用容器基础镜像时，来源是一个很大的问题。您必须使用来自受信任源的受信任镜像。语言运行时、Web 服务器和核心库（如 OpenSSL 等）在投入生产时对安全性有影响。通用基础镜像会及时收到红帽安全团队的安全更新。
- **性能：**基础镜像由红帽内部性能工程团队进行测试、调优和认证。这些是经过验证的容器镜像，广泛用于世界上大部分计算密集型、I/O 密集型和故障敏感型工作负载。
- **ISV、供应商认证和合作伙伴支持：**通用基础镜像继承了支持数千个应用的 RHEL 合作伙伴、ISV 和第三方供应商的广泛生态系统。通用基础镜像使这些合作伙伴可以轻松构建、部署和认证其应用，并允许他们在红帽平台（如 RHEL 和 OpenShift）以及非红帽容器平台上部署生成的容器化应用。
- **构建一次，部署到许多不同的主机上：**红帽通用基础镜像可以在任何地方构建和部署：在 OpenShift/RHEL 或任何其他容器主机（Fedora、Debian、Ubuntu 等）上。

## 高级 Containerfile 指令

Containerfile 会自动构建容器镜像。Containerfile 是一个文本文件，内含一组有关如何构建容器镜像的指令。这些指令会按序逐个执行。本节将介绍一些基本的 Containerfile 指令，然后讨论一些更高级的指令，包括构建容器镜像以在 OpenShift 中部署时使用它们的实用方法。

### RUN 指令

RUN 指令在当前镜像顶部的新层中执行命令，然后提交结果。容器构建进程会将所提交的结果用于 Containerfile 中的下一步。容器构建进程使用 `/bin/sh` 来执行命令。

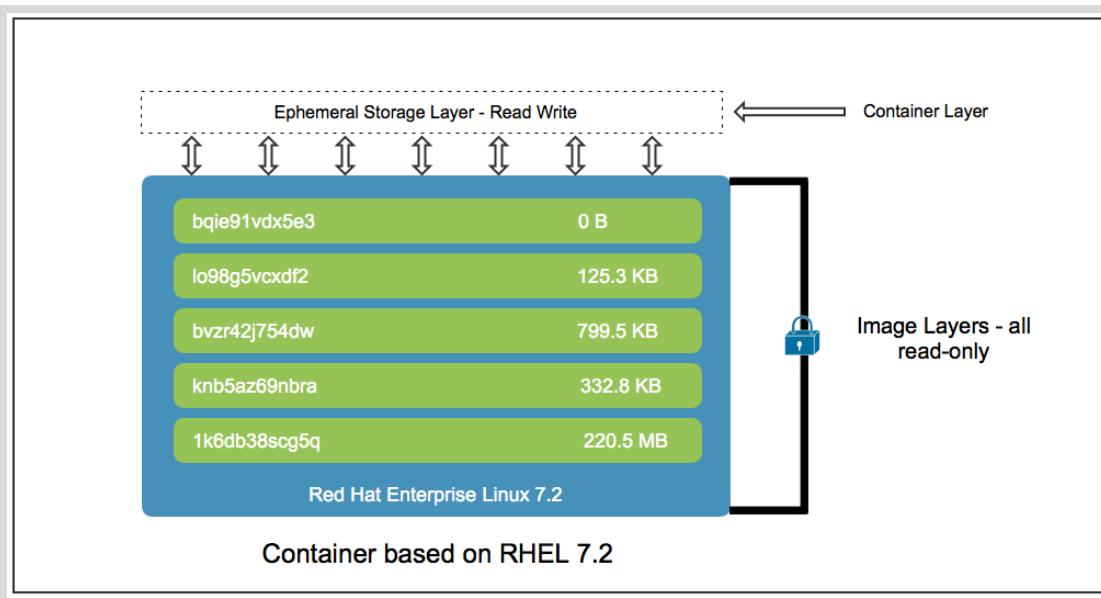


图 2.1: 容器镜像中的分层

Containerfile 中的每个指令都会在最终容器镜像中新增一层。因此，在 Containerfile 中包含太多指令会导致层数过多，进而生成过大的镜像。例如，请思考 Containerfile 中的以下 RUN 指令：

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms"
RUN yum update
RUN yum install -y httpd
RUN yum clean all -y
```

以上例子并非创建容器镜像时的良好做法，因为它为一个目的创建了四个层。创建 Containerfile 时，红帽建议您尽可能减少层数。您可以使用 `&&` 命令分隔符在单个 `RUN` 指令中执行多个命令，以实现相同的目的：

```
RUN yum --disablerepo=* --enablerepo="rhel-7-server-rpms" && \
    yum update && \
    yum install -y httpd && \
    yum clean all -y
```

更新后的示例只创建了一个层，而且也没有破坏可读性。



### 注意

在 Podman 中，只有 `RUN`、`COPY` 和 `ADD` 指令会创建层。其他指令会创建临时中间镜像，不会直接增加镜像的大小。

## LABEL 指令

`LABEL` 指令可定义镜像元数据。标签是与镜像关联的键值对。`LABEL` 指令通常用来为镜像添加描述性元数据（如版本、简短描述）以及添加详情，以向镜像用户提供相关信息。

为 OpenShift 构建镜像时，为标签名称加上前缀 `io.openshift`，以区分 OpenShift 和 Kubernetes 相关元数据。OpenShift 工具可以解析特定标签，并基于这些标签的存在性来执行特定操作。下表列出了部分最常用的标记：

### OpenShift 支持的标签

标签名称	描述
<code>io.openshift.tags</code>	该标签包含一个以逗号分隔的标记列表。标记会将容器镜像归类为众多的功能领域。标记可在应用创建过程中帮助 UI 和生成工具推荐相关的容器镜像。
<code>io.k8s.description</code>	该标签可向容器镜像消费者提供与该镜像所提供的服务或功能有关的更多详细信息。
<code>io.openshift.expose-services</code>	该标签包含一个服务端口列表，这些端口与 Containerfile 中的 <code>EXPOSE</code> 指令相对应，该标签还可提供与向消费者提供的实际服务有关的更多描述性信息。 格式为 <code>PORT[/PROTO]:NAME</code> ，其中的 <code>[/PROTO]</code> 为可选项，其默认值为 <code>tcp</code> （如果未指定）。

如需列有所有特定于 OpenShift 的标签及其描述和使用示例的完整列表，请参考本节结尾处的参考资料。

## WORKDIR 指令

**WORKDIR** 指令可为 Containerfile 中的任意以下 **RUN**、**CMD**、**ENTRYPOINT**、**COPY** 或 **ADD** 指令设置工作目录。

红帽建议在 **WORKDIR** 指令中使用绝对路径。请使用 **WORKDIR**（而非多个 **RUN** 指令）来更改目录，然后运行一些命令。此方法可确保提升长耗时运行的易维护性，并能更加轻松地进行故障排除。

## ENV 指令

**ENV** 指令用于定义可供容器使用的环境变量。您可以在 Containerfile 内声明多个 **ENV** 指令。您可以在容器内使用 **env** 命令来查看各个环境变量。

这是使用 **ENV** 指令来定义文件和文件夹路径（而非在 Containerfile 指令中对其进行硬编码）的良好做法。在存储软件版本号等信息以及为 **PATH** 环境变量附加目录时，这个指令非常有用。

## USER 指令

出于安全原因，红帽建议以非根用户的身份来运行镜像。为减少层数，请避免在 Containerfile 中过多地使用 **USER** 指令。本节稍后会讨论特定于以非根用户身份运行容器的安全隐患。



### 警告

默认情况下，OpenShift 不会遵从容器镜像设置的 **USER** 指令。出于安全原因，OpenShift 会使用随机的 userid（而非根 userid (0)）来运行容器。

## VOLUME 指令

**VOLUME** 指令会在容器内创建一个挂载点，并向消费者表明可以为该挂载点绑定从外部的主机或其他容器挂载的卷。

对于永久数据，红帽建议使用 **VOLUME** 指令。OpenShift 可以为正在运行容器的节点挂载网络附加存储，如果容器移至新节点，则该存储会重新附加到相应的节点。如果使用卷来满足所有永久存储需求，那么即使容器重新启动或移动，您也可以保留相应的内容。

此外，通过在 Containerfile 中显式定义卷，镜像消费者可以轻松知晓他们在运行您的镜像时可以定义哪些卷。

## 使用 ONBUILD 指令构建镜像

**ONBUILD** 指令可在容器镜像中注册 triggers。Containerfile 可使用 **ONBUILD** 来声明只有在构建子镜像时会执行的指令。

**ONBUILD** 指令很适合用来针对常见用例（如向应用预加载数据或提供自定义配置）提供支持，以便轻松自定义容器镜像。父镜像可提供对所有下游子镜像通用的命令。子镜像只会提供数据和配置文件。适用于子镜像的 Containerfile 可以像引用父镜像的 **FROM** 指令一样简单。



### 注意

**ONBUILD** 指令不包括在 OCI 规范中，因此在使用 Podman 或 Buildah 构建容器时，默认情况下不受支持。使用 **--format docker** 选项以启用对 **ONBUILD** 指令的支持。

例如，假设您正在构建一个 Node.js 父镜像，并希望所在企业中的所有开发人员都将其用作基础，以构建满足以下要求的应用：

- 强制实施某些标准，例如：将 JavaScript 源复制到应用文件夹中，以便 Node.js 引擎对其进行解读。
- 执行 `npm install` 命令，以获取 `package.json` 文件中描述的所有依赖关系。

您无法以指令的形式将这些要求嵌入到父 Containerfile 中，因为您没有应用源代码，而且各个应用在其 `package.json` 文件中可能列有不同的依赖项。

在父 Containerfile 中声明 `ONBUILD` 指令。父 Containerfile 如下所示：

```
FROM registry.access.redhat.com/rhscl/nodejs-6-rhel7
EXPOSE 3000
# Make all Node.js apps use /opt/app-root as the main folder (APP_ROOT).
RUN mkdir -p /opt/app-root/
WORKDIR /opt/app-root

# Copy the package.json to APP_ROOT
ONBUILD COPY package.json /opt/app-root

# Install the dependencies
ONBUILD RUN npm install

# Copy the app source code to APP_ROOT
ONBUILD COPY src /opt/app-root

# Start node server on port 3000
CMD [ "npm", "start" ]
```

假设您构建了名为 `mynodejs-base` 的父容器镜像，则使用该父镜像的应用子 Containerfile 如下所示：

```
FROM mynodejs-base
RUN echo "Started Node.js server..."
```

当子镜像的构建流程启动时，它会先触发执行父镜像中定义的三个 `ONBUILD` 指令，然后再调用子 Containerfile 中的 `RUN` 指令。

## 有关 USER 指令的 OpenShift 注意事项

默认情况下，OpenShift 会使用随意分配的 userid 来运行容器。这种方法可以避免因容器引擎存在安全漏洞而导致在容器中运行的进程获得经过升级的主机特权的风险。

## 针对 OpenShift 调整 Containerfile

当您编写或更改用于构建在 OpenShift 集群上运行的镜像的 Containerfile 时，您需要解决以下问题：

- 由容器中的进程读取或写入的目录和文件应归 `root` 组所有，并具有组读取或组写入权限。
- 可执行文件应具有组执行权限。
- 容器中运行的进程不得侦听特权端口（即 1024 以下的端口），因为它们不作为特权用户运行。

通过在 Containerfile 中添加以下 RUN 指令，可以设置目录和文件权限，以允许 **root** 组中的用户访问容器中的进程：

```
RUN chgrp -R 0 directory && \
    chmod -R g=u directory
```

运行容器的用户帐户始终是 **root** 组中的一员，因此容器可以读取或写入到这个目录。**root** 组没有任何特殊权限（不同于 **root** 用户），因而无法最大限度地降低这个配置存在的安全风险。

**chmod** 命令中的 **g=u** 参数可以赋予与所有者用户权限等效的组权限，默认情况下为读取和写入权限。您可以使用 **g+rwx** 参数实现相同的结果。

## 使用安全性上下文约束 (SCC) 以 root 身份运行容器

在某些情况下，您可能无权访问部分镜像的 Containerfile。您可能需要以 **root** 用户的身份来运行此类镜像。在这种情况下，您需要配置 OpenShift，以允许容器以 **root** 身份来运行。

OpenShift 提供安全性上下文约束 (SCC)，它可以控制容器集能够执行的操作以及有权访问的资源。OpenShift 随附了多种内建 SCC。默认情况下，OpenShift 创建的所有容器都会使用名为 **restricted** 的 SCC，它会忽略容器镜像设置的 userid 并为容器随机分配一个 userid。

要使容器使用固定 userid，如 0（即 **root** 用户），您需要使用 **anyuid** SCC。为此，您要先创建一个 service account。服务帐户是容器集的 OpenShift 标识。项目中的所有容器集都在默认服务帐户下运行，除非以其他方式配置容器集或其部署配置。

如果您的应用需要未通过受限 SCC 授予的功能，您可以创建一个新的特定服务帐户，将它添加到适当的 SCC 中，并更改创建应用容器集的部署配置以使用新的服务帐户。

以下步骤详细说明了如何允许容器作为 OpenShift 项目中的 **root** 用户运行：

- 创建新的服务帐户：

```
[user@host ~]$ oc create serviceaccount myserviceaccount
```

- 修改应用的部署配置以使用新的服务帐户。使用 **oc patch** 命令执行以下操作：

```
[user@host ~]$ oc patch dc/demo-app --patch \
'{"spec": {"template": {"spec": {"serviceAccountName": "myserviceaccount"}}}}'
```



### 注意

有关如何使用 **oc patch** 命令的详细信息，请参阅 **oc patch** 的 OpenShift 管理指南 [<https://access.redhat.com/articles/3319751>]。运行 **oc patch -h** 命令以显示用法。

- 将 **myserviceaccount** 服务帐户添加到 **anyuid** SCC 以使用容器中的固定 userid 运行：

```
[user@host ~]$ oc adm policy add-scc-to-user anyuid -z myserviceaccount
```



## 参考文献

### 编写 Dockerfile 的最佳实践

[https://docs.docker.com/engine/userguide/eng-image/dockerfile\\_best-practices](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices)

如需有关创建镜像的更多信息，请参阅 OpenShift 容器平台产品文档的 Images 指南的 Creating Images 一章，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/index#creating-images](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/index#creating-images)

## ▶ 指导练习

# 使用高级 Containerfile 指令构建容器镜像

在本练习中，您将使用红帽 OpenShift 从 Containerfile 中构建和部署 Apache HTTP 服务器容器。

## 成果

您应能够：

- 使用 Containerfile 创建 Apache HTTP 服务器容器镜像，并将其部署在 OpenShift 集群上。
- 通过扩展父 Apache HTTP 服务器镜像创建子容器镜像。
- 更改子容器镜像的 Containerfile，使其在具有随机用户 ID 的 OpenShift 集群上运行。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Apache HTTP 服务器 (`quay.io/redhattraining/httpd-parent`) 的父镜像
- Git 存储库 (container-build) 中的子容器镜像的 Containerfile。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab container-build start
```

## 说明

### ▶ 1. 查看 Apache HTTP 服务器父 Containerfile。

在 `quay.io/redhattraining/httpd-parent` 的 Quay.io 公共注册表中提供预构建 Apache HTTP 服务器父容器镜像。简要查看 `~/D0288/labs/container-build/httpd-parent/Containerfile` 处该父镜像的 Containerfile：

```
FROM registry.access.redhat.com/ubi8/ubi:8.0 ①

MAINTAINER Red Hat Training <training@redhat.com>

# DocumentRoot for Apache
ENV DOCROOT=/var/www/html ②

RUN yum install -y --no-docs --disableplugin=subscription-manager httpd && \
    yum clean all --disableplugin=subscription-manager -y && \
    echo "Hello from the httpd-parent container!" > ${DOCROOT}/index.html ③

# Allows child images to inject their own content into DocumentRoot
ONBUILD COPY src/ ${DOCROOT}/ ④
```

```
EXPOSE 80

# This stuff is needed to ensure a clean start
RUN rm -rf /run/httpd && mkdir /run/httpd

# Run as the root user
USER root ⑤

# Launch httpd
CMD /usr/sbin/httpd -DFOREGROUND
```

- ① 基础镜像是红帽企业 Linux 8.0 的通用基础镜像 (UBI)，来自于红帽容器目录。
- ② 该容器镜像的环境变量。
- ③ RUN 指令包含安装 Apache HTTP 服务器的几个命令，并为 Web 服务器创建默认主页。
- ④ ONBUILD 指令允许子镜像在构建从父镜像扩展而来的镜像时，提供自己定制的 Web 服务器内容。
- ⑤ USER 指令以 root 用户身份运行 Apache HTTP 服务器进程。



### 注意

请注意 RUN 行如何尽可能将多个命令组合成单个指令，以减少镜像中的层数。这会形成更小的镜像，部署更快。

## ▶ 2. 查看 Apache HTTP 服务器子 Containerfile。

您可以使用 Apache HTTP 服务器容器镜像 (`redhattraining/httpd-parent`) 作为基础，来扩展并自定义镜像以适合您的应用。子容器镜像的 Containerfile 存储在课堂 Git 存储库服务器中。要查看 Containerfile，请执行以下步骤：

### 2.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

### 2.2. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `main` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

### 2.3. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b container-build
Switched to a new branch 'container-build'
[student@workstation D0288-apps]$ git push -u origin container-build
...output omitted...
* [new branch]      container-build -> container-build
Branch container-build set up to track remote branch container-build from origin.
```

- 2.4. 检查 ~/D0288-apps/container-build/Containerfile 文件。Containerfile 有一条指令 FROM，它使用 redhattraining/httpd-parent 镜像：

```
FROM quay.io/redhattraining/http-parent
```

- 2.5. 子容器在 ~/D0288-apps/container-build/src 文件夹中提供自己的 index.html 文件，该文件将覆盖父 index.html 文件。子容器镜像的 index.html 文件的内容如下：

```
<!DOCTYPE html>
<html>
<body>
  Hello from the Apache child container!
</body>
</html>
```

### ▶ 3. 使用 Apache HTTP 服务器子 Containerfile 构建和部署容器至 OpenShift 集群。

- 3.1. 使用您的开发人员用户名登录 OpenShift：

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 3.2. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-container-build
Now using project "yourdevuser-container-build" on server "https://
api.cluster.domain.example.com:6443".
```

- 3.3. 构建 Apache HTTP 服务器子镜像：

```
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-apache ./container-build
STEP 1: FROM quay.io/redhattraining/httpd-parent ①
...output omitted...
STEP 2: COPY src/ ${DOCROOT}/ ②
STEP 3: COMMIT do288-apache
...output omitted...
Storing signatures
...output omitted...
```

① Containerfile 会被自动识别出，并且 podman 提取父镜像。

② 父 Containerfile 中的 ONBUILD 指令会触发子 index.html 文件的复制，其会覆盖父索引页。



## 重要

构建进程可能需要稍等片刻才会开始。如果看到以下输出，则再次运行该命令。

```
Error from server (BadRequest): unable to wait for build hola-1 to run: timed out waiting for the condition
```

3.4. 查看镜像：

```
[student@workstation D0288-apps]$ podman images
localhost/do288-apache          latest fc114a884288  9 minutes ago  236 MB
quay.io/redhattraining/httpd-parent latest 4346d3cace25  2 years ago   236 MB
```

3.5. 标记镜像并推送到 Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-apache \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
[student@workstation D0288-apps]$ podman login quay.io -u ${RHT_OCP4_QUAY_USER}
Password:
Login Succeeded!
[student@workstation D0288-apps]$ podman push \
--format v2s1 quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
...output omitted...
Storing signatures
[student@workstation D0288-apps]$
```

3.6. 登录 Quay.io [https://quay.io]，使新镜像变为公共镜像

3.7. 部署 Apache HTTP 服务器子镜像：

```
[student@workstation D0288-apps]$ oc new-app --name hola \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
--> Found Container image 1d6f8d3 (11 minutes old) from quay.io for "quay.io/
yourquayuser/do288-apache"

Red Hat Universal Base Image 8
-----
The Universal Base Image is designed and engineered to be the base layer
for all of your containerized applications, middleware and utilities. This base
image is freely redistributable, but Red Hat only supports Red Hat
technologies through subscriptions for Red Hat products. This image is
maintained by Red Hat and updated regularly.

Tags: base rhel8

...output omitted...
--> Success
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose svc/hola'
Run 'oc status' to view your app.
```

- ▶ 4. 验证应用容器集是否无法启动。容器集将处于 **Error** 状态，但是如果您等待时间太长，容器集将变为 **CrashLoopBackOff** 状态。

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS           RESTARTS   AGE
hola-58554c88b9-qxc7n   0/1    CrashLoopBackOff   0          12s
```

- ▶ 5. 检查容器的日志，查看容器集无法启动的原因：

```
[student@workstation D0288-apps]$ oc logs hola-13p75f5
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name...
(13)Permission denied: AH00072: make_sock: could not bind to address [::]:80 ①
(13)Permission denied: AH00072: make_sock: could not bind to address 0.0.0.0:80 ②
no listening sockets available, shutting down
AH00015: Unable to open logs ③
```

- ① 由于 OpenShift 使用随机 userid 运行容器，低于 1024 的端口是特权端口，只能以 **root** 身份运行。
- ② OpenShift 用于运行容器的随机 userid 不具有在 **/var/log/httpd** (RHEL 7 上的 Apache HTTP 服务器的默认日志文件位置) 中读写日志文件的权限。



### 警告

无法启动的应用容器集很快会被删除。删除容器集之前，确保检查应用容器集日志文件。

- ▶ 6. 删除 OpenShift 项目中的所有资源。下一步是更改 Containerfile 以遵循红帽的 OpenShift 建议。

更新子 Apache HTTP 服务器 Containerfile 之前，删除至今已经创建的项目中的所有资源：

```
[student@workstation D0288-apps]$ oc delete all -l app=hola
service "hola" deleted
deployment.apps "hola" deleted
imagestream.image.openshift.io "hola" deleted
```

- ▶ 7. 通过更新 Apache HTTP 服务器进程以作为随机、非特权用户运行，更改子容器的 Containerfile 以便在 OpenShift 集群上运行。

7.1. 编辑 **~/D0288-apps/container-build/Containerfile** 文件并执行以下步骤。  
您也可以从提供的 **~/D0288/solutions/container-build/Containerfile** 文档中复制指令。

7.2. 覆盖父镜像的 **EXPOSE** 指令并将端口更改为 8080。

```
EXPOSE 8080
```

7.3. 包括 **io.openshift.expose-service** 标签以指示 Web 服务器运行的已更改端口：

```
LABEL io.openshift.expose-services="8080:http"
```

更新标签列表以包括 `io.k8s.description`、`io.k8s.display-name` 和 `io.openshift.tags` 标签，OpenShift 使用它们来提供有关容器镜像的有用元数据：

```
LABEL io.k8s.description="A basic Apache HTTP Server child image, uses ONBUILD" \
      io.k8s.display-name="Apache HTTP Server" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="apache, httpd"
```

7.4. 您需要在非特权端口（即大于 1024）上运行 Web 服务器。使用 `RUN` 指令将 Apache HTTP 服务器配置文件中的端口号从默认端口 80 更改为 8080：

```
RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf \
RUN sed -i "s/#ServerName www.example.com:80/ServerName 0.0.0.0:8080/g" \
/etc/httpd/conf/httpd.conf
```

7.5. 更改 Web 服务器进程读写文件的文件夹的组 ID 和权限：

```
RUN chgrp -R 0 /var/log/httpd /var/run/httpd && \
    chmod -R g=u /var/log/httpd /var/run/httpd
```

7.6. 为非特权用户添加 `USER` 指令。红帽惯例是使用 userid 1001：

```
USER 1001
```

7.7. 保存 Containerfile 并将更改从 `~/D0288-apps/container-build` 文件夹提交到 Git 存储库中：

```
[student@workstation D0288-apps]$ cd container-build
[student@workstation container-build]$ git commit -a -m \
"Changed the Containerfile to enable running as a random uid on OpenShift"
...output omitted...
[student@workstation container-build]$ git push
...output omitted...
[student@workstation container-build]$ cd ..
```

► 8. 重新构建并重新部署 Apache HTTP 服务器子容器镜像。

8.1. 移除旧镜像

```
[student@workstation D0288-apps]$ podman rmi -a --force
...output omitted...
```

8.2. 使用新的 Containerfile 重新创建应用：

```
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-apache ./container-build
STEP 1: FROM quay.io/redhattraining/httpd-parent
...output omitted...
STEP 2: COPY src/ ${DOCROOT}/
STEP 3: EXPOSE 8080
STEP 4: LABEL io.k8s.description="A basic Apache HTTP Server child image,
uses ONBUILD"          io.k8s.display-name="Apache HTTP Server"
io.openshift.expose-services="8080:http"          io.openshift.tags="apache, httpd"
STEP 5: RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf
STEP 6: RUN chgrp -R 0 /var/log/httpd /var/run/httpd &&      chmod -R g=u /var/log/
httpd /var/run/httpd
STEP 7: USER 1001
STEP 8: COMMIT do288-apache
...output omitted...
```

### 8.3. 标记新镜像并替换 Quay.io 中的镜像

```
[student@workstation D0288-apps]$ podman tag do288-apache \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
[student@workstation D0288-apps]$ podman push \
--format v2s1 quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
...output omitted...
Storing signatures
[student@workstation D0288-apps]$
```

### 8.4. 重新部署 Apache HTTP 服务器子容器镜像。

```
[student@workstation ~]$ oc new-app --name hola \
quay.io/${RHT_OCP4_QUAY_USER}/do288-apache
--> Found container image fe746d5 (7 minutes old) from quay.io for "quay.io/
yourquayuser/do288-apache"
...output omitted...
--> Success
Application is not exposed. You can expose services to the outside world by
executing one or more of the commands below:
'oc expose service/hola'
Run 'oc status' to view your app.
```

### 8.5. 等待容器集就绪并在运行。查看应用容器集的状态：

```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
hola-58554c88b9-qxc7n   1/1     Running   0          5s
```

应用容器集现在将成功启动并处于 **Running** 状态。

## ▶ 9. 创建 OpenShift 路由器以公开应用供外部访问：

```
[student@workstation ~]$ oc expose --port='8080' svc/hola
route.route.openshift.io/hola exposed
```

- ▶ 10. 使用 `oc get route` 命令获取路由 URL:

```
[student@workstation ~]$ oc get route  
NAME      HOST/PORT  
hola      hola-yourquayuser-container-build.cluster.domain.example.com ...
```

- ▶ 11. 使用您在上一步中获取的路由 URL 测试应用:

```
[student@workstation ~]$ curl \  
http://hola-${RHT_OCP4_DEV_USER}-container-build.${RHT_OCP4_WILDCARD_DOMAIN}  
...output omitted...  
Hello from the Apache child container!  
...output omitted...
```

- ▶ 12. 清理。删除项目:

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-container-build
```

## 完成

在 `workstation` 上运行 `lab container-build finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab container-build finish
```

本引导式练习到此结束。

# 将配置数据注入应用

## 培训目标

学完本节后，您应能够选择将配置数据注入应用的方法，并创建完成注入操作所需的资源。

## 使 OpenShift 中的应用配置外部化

通常，开发人员都会通过组合环境变量、命令行参数和配置文件来配置其应用。在将应用部署至 OpenShift 时，容器具备的不可变本质会导致配置管理面临挑战。不同于传统的非容器化部署，我们不建议在运行容器化应用时将应用和配置耦合在一起。

对于容器化应用，建议将静态应用二进制文件与动态配置数据分隔开，并使配置外部化。这种分隔可确保应用可在多个环境间移植。

例如，假设您想将部署在 OpenShift 集群上的应用以及测试和用户接受度等中间阶段从开发环境推广至生产环境。您应该在所有阶段使用同一个应用容器镜像，并将特定于各个环境的配置详情置于这个容器镜像的外部。

## 使用机密和配置映射资源

OpenShift 提供了机密和配置映射资源类型，以用于实现应用配置的外部化并管理这些配置。

机密资源用于存储密码、密钥和令牌等敏感信息。作为开发人员，创建机密以避免泄露应用中的凭据和其他敏感信息非常重要。以下几个不同的机密类型可用于强制在机密对象中使用用户名和密钥：`service-account-token`、`basic-auth`、`ssh-auth`、`tls` 和 `opaque`。默认类型为 `opaque`。`opaque` 类型不执行任何验证，并允许包含任意值的非结构化键:值对。

配置映射资源类似于机密资源，但存储非敏感数据。配置映射资源可用于存储细粒度的信息（如单个属性）或粗粒度的信息（如整个配置文件和 JSON 数据）。

您可以使用 OpenShift CLI 或 Web 控制台来创建配置映射和机密资源。然后，您可以在容器集规格中引用这些资源，OpenShift 会将资源数据作为环境变量或作为通过应用容器卷挂载的文件注入到容器中。

您还可以为正在运行的应用更改部署配置，以引用配置映射和机密资源。然后，OpenShift 会自动重新部署应用，并使数据可用于容器。

使用 base64 编码将数据存储在机密资源中。当将来自机密的数据注入容器时，数据将被解码并作为文件挂载，或作为环境变量注入容器内。

## 机密和配置映射的功能

请注意与机密和配置映射相关的以下事项：

- 它们可以独立于其定义被引用。
- 出于安全原因，已为这些资源挂载的卷由临时文件存储设备 (`tmpfs`) 来支持，而且从不存储在节点上。
- 它们会被纳入到命名空间中。

## 创建和管理机密和配置映射

必须先创建机密和配置映射，然后再创建基于其上的容器集。您可以使用 CLI 或 Web 控制台创建这些资源。

### 使用命令行

使用 `oc create` 命令可创建机密和配置映射资源。

要创建新的配置映射以存储字符串文字，请使用以下命令：

```
[user@host ~]$ oc create configmap config_map_name \
--from-literal key1=value1 \
--from-literal key2=value2
```

要创建新的机密以存储字符串文字，请使用以下命令：

```
[user@host ~]$ oc create secret generic secret_name \
--from-literal username=user1 \
--from-literal password=mypassword
```

要创建新的配置映射以存储文件内容或存储包含一组文件的目录，请使用以下命令：

```
[user@host ~]$ oc create configmap config_map_name \
--from-file /home/demo/conf.txt
```

在从文件创建配置映射时，默认情况下，密钥名称为文件的名称，密钥值为文件所含的内容。

在基于目录创建配置映射资源时，名称为目录中有效密钥的各个文件都会存储在该配置映射中。子目录、符号链接、设备文件和管道则会被忽略。

请运行 `oc create configmap --help` 命令以了解更多信息。



#### 注意

在 `oc` 命令行界面中，您还可以将 `configmap` 资源类型参数简写为 `cm`。例如：

```
[user@host ~]$ oc create cm myconf --from-literal key1=value1
[user@host ~]$ oc get cm myconf
```

要创建新的机密以存储文件内容或存储包含一组文件的目录，请使用以下命令：

```
[user@host ~]$ oc create secret generic secret_name \
--from-file /home/demo/mysecret.txt
```

在从文件或目录创建机密时，其密钥名称的设置方式与配置映射相同。

如需了解更多详情（包括如何在机密中存储 TLS 证书和密钥），请运行 `oc create secret --help` 和 `oc secret` 命令。

## 使用 OpenShift Web 控制台

您还可以使用 OpenShift Web 控制台来创建配置映射和机密。要从 Web 控制台创建和管理机密，请登录 OpenShift Web 控制台，然后导航到 **Workloads > Secrets** 页面。

Name	Namespace	Type	Size	Created	⋮
builder-dockercfg-tbf7z	NS your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	⋮
builder-token-cs2r4	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮
builder-token-hxj2h	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮
default-dockercfg-97qm	NS your-project	kubernetes.io/dockercfg	1	Aug 3, 12:05 am	⋮
default-token-c892t	NS your-project	kubernetes.io/service-account-token	4	Aug 3, 12:05 am	⋮

图 2.2: 从 Web 控制台管理机密

要从 Web 控制台创建和管理配置映射，请导航到 **Workloads > Config Maps** 页面。

Name	Namespace	Size	Created	⋮
your-project-l-ca	NS your-project	1	Aug 3, 3:18 pm	⋮
your-project-l-global-ca	NS your-project	1	Aug 3, 3:18 pm	⋮
your-project-l-sys-config	NS your-project	0	Aug 3, 3:18 pm	⋮

图 2.3: 从 Web 控制台管理配置映射

您可以使用 Web 控制台提供的 YAML 编辑器来编辑已分配给配置映射中各个密钥的值，以及已分配给机密中各个密钥的编码值。但是，对于机密，您要先以 base64 格式对数据进行编码，然后再将其插入到机密资源定义中。

## 配置映射和机密资源定义

由于配置映射和机密是常规 OpenShift 资源，所以您可以使用 `oc create` 命令或 Web 控制台以 YAML 或 JSON 格式导入这些资源定义文件。

采用 YAML 格式的配置映射资源示例如下所示：

```
apiVersion: v1
data:
  key1: value1 ①②
  key2: value2 ③④
kind: ConfigMap ⑤
metadata:
  name: myconf ⑥
```

- ① 第一个密钥的名称。默认情况下，与该密钥同名的环境变量或文件会被注入到容器中，具体取决于配置映射资源是作为环境变量还是作为文件注入。
- ② 为配置映射的第一个密钥所存储的值。
- ③ 第二个密钥的名称。
- ④ 为配置映射的第二个密钥所存储的值。
- ⑤ OpenShift 资源类型，在这个案例中，资源类型为配置映射。
- ⑥ 该配置映射在项目中的唯一名称。

采用 YAML 格式的机密资源示例如下所示：

```
apiVersion: v1
data:
  username: cm9vdAo= ①②
  password: c2VjcmV0Cg== ③④
kind: Secret ⑤
metadata:
  name: mysecret ⑥
  type: Opaque
```

- ① 第一个密钥的名称。这会为容器集中的环境变量或文件提供默认名称，方式与设置配置映射的密钥名称相同。
- ② 为第一个密钥存储的值，采用以 base64 编码的格式。
- ③ 第二个密钥的名称。
- ④ 为第二个密钥存储的值，采用以 base64 编码的格式。
- ⑤ OpenShift 资源类型，在这个案例中，资源类型为机密。
- ⑥ 该机密资源在项目中的唯一名称。

## 机密资源定义的替代语法

模板无法采用标准语法来定义机密，因为所有密钥的值都会经过编码。OpenShift 针对此类情况提供了相应的替代语法，以使用 **stringData** 属性替代 **data** 属性，且不会对密钥值进行编码。

如果采用替代语法，上一示例会变为：

```

apiVersion: v1
stringData:
  username: user1
  password: pass1
kind: Secret
metadata:
  name: mysecret
  type: Opaque

```

替代语法从不会保存在 OpenShift 主控机 etcd 数据库中。OpenShift 会将使用替代语法定义的机密资源转换为标准的存储表示法。如果使用以替代语法创建的机密来运行 `oc get`，则会获得一个使用标准语法的资源。

## 用于操控配置映射的命令

要以 JSON 格式查看配置映射详情，或将配置映射资源定义导出到 JSON 文件以进行离线创建，请使用以下命令：

```
[user@host ~]$ oc get configmap/myconf -o json
```

要删除配置映射，请使用以下命令：

```
[user@host ~]$ oc delete configmap/myconf
```

要编辑配置映射，请使用 `oc edit` 命令。默认情况下，该命令会打开一个与 Vim 类似的缓冲区，该缓冲区中包含采用 YAML 格式的配置映射资源定义：

```
[user@host ~]$ oc edit configmap/myconf
```

使用 `oc patch` 命令以编辑配置映射资源。此方法不是交互式的，当您需要通过编写脚本的方式来更改资源时，此方法非常有用。

```
[user@host ~]$ oc patch configmap/myconf --patch '{"data":{"key1":"newValue1"}}'
```

## 用于操控机密的命令

用于操控机密资源的命令与用于配置映射资源的命令类似。

要查看或导出机密详情，请使用以下命令：

```
[user@host ~]$ oc get secret/mysecret -o json
```

要删除机密，请使用以下命令：

```
[user@host ~]$ oc delete secret/mysecret
```

要编辑机密，请先采用 base64 格式对数据进行编码，例如：

```
[user@host ~]$ echo 'newpassword' | base64  
bmV3cGFzc3dvcmQK
```

通过 `oc edit` 命令使用编码值来更新机密资源：

```
[user@host ~]$ oc edit secret/mysecret
```

您还可以使用 `oc patch` 命令来编辑机密资源：

```
[user@host ~]$ oc patch secret/mysecret --patch \  
'{"data":{"password":"bmV3cGFzc3dvcmQK"}}'
```

还可使用 OpenShift Web 控制台来更改和删除配置映射和机密。

## 将机密和配置映射的数据注入应用

配置映射和机密可在应用容器内作为数据卷来挂载，或作为环境变量予以公开。

要将配置映射中存储的所有值注入到从部署创建的容器集的环境变量中，请使用 `oc set env` 命令：

```
[user@host ~]$ oc set env deployment/mydcname \  
--from configmap/myconf
```

要将配置映射中的所有密钥挂载为从部署创建的容器集内的卷中的文件，请使用 `oc set volume` 命令：

```
[user@host ~]$ oc set volume deployment/mydcname --add \  
-t configmap -m /path/to/mount/volume \  
--name myvol --configmap-name myconf
```

要将机密中的数据注入到从部署创建的容器集中，请使用 `oc set env` 命令：

```
[user@host ~]$ oc set env deployment/mydcname \  
--from secret/mysecret
```

要将机密资源中的数据挂载为从部署创建的容器集内的卷，请使用 `oc set volume` 命令：

```
[user@host ~]$ oc set volume deployment/mydcname --add \  
-t secret -m /path/to/mount/volume \  
--name myvol --secret-name mysecret
```

## 应用配置选项

如果并非敏感信息，则请使用配置映射以纯文本形式来存储配置数据。如果要存储敏感信息，则请使用机密。

如果您的应用只包含少量可从环境变量读取或通过命令行传递的简单配置变量，则请使用环境变量来注入配置映射和机密中的数据。环境变量是优于在容器内装载卷的首选方法。

## 章2 | 针对 OpenShift 设计容器化应用

另一方面，如果您的应用包含大量配置变量，或是所迁移的传统应用大量使用了配置文件，请使用卷挂载方法，而不是为每个配置变量创建环境变量。例如，如果应用从文件系统上的特定位置预期一个或多个配置文件，则应从配置文件创建机密或配置映射，并在容器临时文件系统内将它们挂载到应用预期的位置。

要达到此目标，通过指向 `/home/student/configuration.properties` 文件的机密，使用以下命令：

```
[user@host ~]$ oc create secret generic security \
--from-file /home/student/configuration.properties
```

要将机密注入应用，请配置引用在上一命令中创建的机密的卷。卷必须指向应用内存储机密文件的实际目录。

在下面的示例中，`configuration.properties` 文件存储在 `/opt/app-root/secure` 目录中。要将文件绑定到应用，请从应用配置部署配置 (`dc/application`)：

```
[user@host ~]$ oc set volume deployment/application --add \
-t secret -m /opt/app-root/secure \
--name myappsec-vol --secret-name security
```

要创建配置映射，请使用以下命令：

```
[user@host ~]$ oc create configmap properties \
--from-file /home/student/configuration.properties
```

要将应用绑定到配置映射，请从该应用更新部署配置以使用配置映射：

```
[user@host ~]$ oc set env deployment/application \
--from configmap/properties
```



### 参考文献

有关机密的更多信息，请参阅红帽 OpenShift 容器平台 4.10 的 Nodes 指南中的 Providing Sensitive Data to Pods 章节，网址为：  
[https://access.redhat.com/documentation/en-us/redshift\\_container\\_platform/4.10/html-single/nodes/index#nodes-pods-secrets](https://access.redhat.com/documentation/en-us/redshift_container_platform/4.10/html-single/nodes/index#nodes-pods-secrets)

## ▶ 指导练习

# 将配置数据注入应用

在本练习中，您将使用配置映射和机密使容器化应用的配置实现外部化。

## 成果

您应能够：

- 部署基于 Node.js 的简单应用，以打印环境变量和文件中的配置详情。
- 使用配置映射和机密将配置数据注入到容器中。
- 更改配置映射中的数据，并验证应用是否已使用更改后的值。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Node.js 12 的 S2I 构建器镜像。
- Git 存储库中的示例应用 (**app-config**)。

在 **workstation** 虚拟机上运行以下命令，以验证练习前提条件并下载实验和答案文件：

```
[student@workstation ~]$ lab app-config start
```

## 说明

### ▶ 1. 查看应用源代码。

- 1.1. 进入 **D0288-apps** Git 存储库的本地克隆，并签出课程存储库的 **main** 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 1.2. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b app-config
Switched to a new branch 'app-config'
[student@workstation D0288-apps]$ git push -u origin app-config
...output omitted...
* [new branch]      app-config -> app-config
Branch app-config set up to track remote branch app-config from origin.
```

- 1.3. 检查 `/home/student/D0288-apps/app-config/app.js` 文件。

应用读取 APP\_MSG 环境变量的值，并打印 /opt/app-root/secure/myapp.sec 文件的内容：

```
// read in the APP_MSG env var
var msg = process.env.APP_MSG;
...output omitted...
// Read in the secret file
fs.readFile('/opt/app-root/secure/myapp.sec', 'utf8', function (secerr, seadata) {
...output omitted...
```

## ▶ 2. 构建并部署应用。

### 2.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

### 2.2. 使用您的开发人员用户帐户登录 OpenShift:

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

### 2.3. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀：

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-app-config
```

### 2.4. 基于 Git 中的来源创建一个名为 myapp 新应用。使用您在上一步中创建的分支。

您可以从 /home/student/D0288/labs/app-config 文件夹中的 oc-new-app.sh 脚本复制或执行命令：

```
[student@workstation D0288-apps]$ oc new-app --name myapp --build-env \
npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
nodejs:16-ubi8-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#app-config \
--context-dir app-config
...output omitted...
--> Creating resources ...output omitted...
imagestream.image.openshift.io "myapp" created
buildconfig.build.openshift.io "myapp" created
deployment.apps.openshift.io "myapp" created
service "myapp" created
--> Success
...output omitted...
```

注意 npm\_config\_registry 后面的等号 (=) 前后没有空格。

### 2.5. 查看构建日志。等待构建完成并将应用容器镜像推送到 OpenShift 内部注册表：

```
[student@workstation D0288-apps]$ oc logs -f bc/myapp
Cloning "https://github.com/yourgithubuser/D0288-apps#app-config" ...
...output omitted...
---> Installing application source ...
---> Building your Node application from source
...output omitted...
Pushing image image-registry.openshift-image-registry.svc:5000/yourgithubuser-app-
config/myapp:latest ...
...output omitted...
Push successful
```

### ► 3. 测试应用。

3.1. 等待应用部署好。查看应用容器集的状态。应用容器集应处于 **Running** 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
myapp-1-build  0/1     Completed  0          65s
myapp-597fdb8cc9-z6t86  1/1     Running   0          29s
```

3.2. 使用路由以公开应用供外部访问：

```
[student@workstation D0288-apps]$ oc expose svc myapp
route.route.openshift.io/myapp exposed
```

3.3. 标识公开应用 API 的路由 URL：

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT
myapp    myapp-yourdevuser-app-config.apps.cluster.domain.example.com ...
```

3.4. 使用 `curl` 命令调用上一步中标识的路由 URL：

```
[student@workstation D0288-apps]$ curl \
http://myapp-$[RHT_OCP4_DEV_USER]-app-config.$[RHT_OCP4_WILDCARD_DOMAIN]
Value in the APP_MSG env var is => undefined
Error: ENOENT: no such file or directory, open '/opt/app-root/secure/myapp.sec'
```

您会看到环境变量的 `undefined` 值以及 `ENOENT: no such file or directory` 错误，因为容器中不存在此类环境变量或文件。

### ► 4. 创建配置映射和机密资源。

4.1. 创建配置映射资源，以保存存储纯文本数据的配置变量。

创建名为 `myappconf` 的新配置映射资源。在此配置映射中存储名为 `APP_MSG` 的密钥，其值为 `Test Message`：

```
[student@workstation D0288-apps]$ oc create configmap myappconf \
--from-literal APP_MSG="Test Message"
configmap/myappconf created
```

4.2. 验证配置映射是否包含配置数据：

```
[student@workstation D0288-apps]$ oc describe cm/myappconf
Name: myappconf
...output omitted...
Data
=====
APP_MSG:
---
Test Message
...output omitted...
```

4.3. 检查 /home/student/D0288-apps/app-config/myapp.sec 文件的内容：

```
username=user1
password=pass1
salt=xyz123
```

4.4. 创建新的机密，以存储 myapp.sec 文件的内容。

```
[student@workstation D0288-apps]$ oc create secret generic myappfilesec \
--from-file /home/student/D0288-apps/app-config/myapp.sec
secret/myappfilesec created
```

4.5. 验证机密的内容。请注意，这些内容会以 base64 编码的格式来存储：

```
[student@workstation D0288-apps]$ oc get secret/myappfilesec -o json
{
  "apiVersion": "v1",
  "data": {
    "myapp.sec": "dXNlcm5hbWU9dXNlcjEKcGFzc3dvcmQ9cGFyczEKc2...
  },
  "kind": "Secret",
  "metadata": {
    ...output omitted...
    "name": "myappfilesec",
    ...output omitted...
  },
  "type": "Opaque"
}
```

▶ 5. 将配置映射和机密注入到应用容器。

5.1. 使用 `oc set env` 命令将配置映射添加到部署配置：

```
[student@workstation ~]$ oc set env deployment/myapp \
--from configmap/myappconf
deployment.apps.openshift.io/myapp updated
```

5.2. 使用 `oc set volume` 命令将机密添加到部署配置：

## 章 2 | 针对 OpenShift 设计容器化应用

您可以从 `/home/student/D0288/labs/app-config` 文件夹中的 `inject-secret-file.sh` 脚本复制或执行命令：

```
[student@workstation D0288-apps]$ oc set volume deployment/myapp --add \
-t secret -m /opt/app-root/secure \
--name myappsec-vol --secret-name myappfilesec
deployment.apps/myapp volume updated
```

### ▶ 6. 验证应用是否已重新部署并使用了配置映射和机密中的数据。

6.1. 验证应用是否因前几步中对部署所做的更改而重新进行了部署：

```
[student@workstation D0288-apps]$ oc status
In project yourdevuser-app-config on server ...output omitted...

http://myapp-yourdevuser-app-config.apps.cluster.domain.example.com to pod port
8080-tcp (svc/myapp)
deployment/myapp deploys istag/myapp:latest <-
bc/myapp source builds https://github.com/yourgithubuser/D0288-apps#app-config
on openshift/nodejs:16-ubi8
deployment #4 running for 10 seconds - 1 pod
deployment #3 deployed 44 seconds ago
deployment #2 deployed 3 minutes ago
deployment #1 deployed 4 minutes ago
```



#### 注意

您应该会看到应用重新部署了两次，因为有两个 `oc set env` 命令更改了部署。

您还可以安全地忽略与以下内容类似的错误消息：

```
deployment #2 failed 59 seconds ago: newer deployment was found running
```

6.2. 等待应用容器集准备就绪并处于 Running 状态。使用 `oc get pods` 命令获取应用容器集的名称，

```
[student@workstation D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
myapp-1-build  0/1     Completed  0          8m12s
myapp-ddffbc7f9-ntsjq  1/1     Running   0          3m53s
```

6.3. 使用 `oc rsh` 命令检查容器中的环境变量：

```
[student@workstation D0288-apps]$ oc rsh myapp-ddffbc7f9-ntsjq env | grep APP_MSG
APP_MSG=Test Message
```

6.4. 验证容器映射和机密是否已注入到容器。使用路由 URL 重新测试应用：

```
[student@workstation D0288-apps]$ curl \
http://myapp-${RHT_OCP4_DEV_USER}-app-config.${RHT_OCP4_WILDCARD_DOMAIN}
Value in the APP_MSG env var is => Test Message
The secret is => username=user1
password=pass1
salt=xyz123
```

OpenShift 会以环境变量的形式将配置映射注入到容器中，并以文件的形式将机密挂载到容器中。应用会读取环境变量和文件，并显示其数据。

#### ► 7. 更改存储在配置映射中的信息，并重新测试应用。

- 7.1. 使用 `oc edit configmap` 命令更改 `APP_MSG` 密钥的值：

```
[student@workstation D0288-apps]$ oc edit cm/myappconf
```

上述命令将打开一个与 Vim 类似的缓冲区，该缓冲区中包含采用 YAML 格式的配置映射属性。按照如下所示编辑数据部分下与 `APP_MSG` 密钥关联的值并更改值：

```
...output omitted...
apiVersion: v1
data:
  APP_MSG: Changed Test Message
kind: ConfigMap
...output omitted...
```

保存并关闭该文件。

- 7.2. 验证 `APP_MSG` 密钥中的值是否已更新：

```
[student@workstation D0288-apps]$ oc describe cm/myappconf
Name: myappconf
...output omitted...
Data
=====
APP_MSG:
---
Changed Test Message
...output omitted...
```

- 7.3. 使用 `oc delete pod` 命令触发新的部署：删除容器集后，复制控制器会自动部署新的容器集。这可确保应用使用配置映射中已更改的值：

```
[student@workstation D0288-apps]$ oc delete pod myapp-ddffbc7f9-ntsjq
pod "myapp-ddffbc7f9-ntsjq" deleted
```

- 7.4. 等待应用容器集重新部署好并处于 `Running` 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
myapp-1-build 0/1     Completed  0          16m
myapp-ddffbc7f9-5wls6 1/1     Running   0          2m37s
```

7.5. 测试应用，并验证配置映射中显示的是否是更改后的值：

```
[student@workstation D0288-apps]$ curl \
http://myapp-${RHT_OCP4_DEV_USER}-app-config.${RHT_OCP4_WILDCARD_DOMAIN}
Value in the APP_MSG env var is => Changed Test Message
The secret is => username=user1
password=pass1
salt=xyz123
```

► 8. 清理。在 OpenShift 中删除 yourdevuser-app-config 项目：

```
[student@workstation D0288-apps]$ oc delete project \
${RHT_OCP4_DEV_USER}-app-config
project.project.openshift.io "yourdevuser-app-config" deleted
```

## 完成

在 workstation 上运行 `lab app-config finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation D0288-apps]$ lab app-config finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 针对 OpenShift 设计容器化应用

在本实验中，您将修复基于 Thorntail 框架的应用的 Containerfile，以便在 OpenShift 集群上运行。您还将使用配置映射来配置应用。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应该能够为基于 Thorntail 框架的应用修复 Containerfile，以作为随机用户运行，并在 OpenShift 集群上部署该应用。您还应该能够使用配置映射来存储用于配置应用的简单文本字符串。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 应用的可运行 fat JAR。
- 应用 Git 存储库。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还可以下载实验室的帮助文件和答案文件：

```
[student@workstation ~]$ lab design-container start
```

## 要求

该应用使用 Thorntail 框架以 Java 编写。提供了包含应用和 Thorntail 运行时的预构建、可运行 JAR 文件 (fat JAR)。该应用提供了简单的 REST API，它根据作为环境变量注入容器的配置来响应请求。根据以下要求，构建应用并将其部署至 OpenShift 集群：

- OpenShift 的应用名称为 **elvis**。应用的配置应存储在名为 **appconfig** 的配置映射中。
- 将应用部署到 **yourdevuser-design-container** 项目。
- 应用的 REST API 应可通过以下 URL 访问：

**elvis-yourdevuser-design-container.apps.cluster.domain.example.com/api/hello**

- 包含应用源的 Git 存储库和文件夹为：

<https://github.com/yourgithubuser/D0288-apps/hello-java>

- 预构建的应用 JAR 文件位于：

```
https://github.com/RedHatTraining/DO288-apps/releases/download/OCP-4.1-1/hello-java.jar
```

## 说明

1. 导航到 **DO288-apps** Git 存储库的本地克隆，再从 **main** 分支创建一个名为 **design-container** 的新分支。简要查看 **/home/student/DO288-apps/hello-java/** 目录中应用的 Containerfile。
2. 使用应用的 **design-container** 分支在 **DO288-apps** Git 存储库的 **hello-java** 文件夹中部署应用。在 OpenShift 中将应用部署到 **yourdevuser-design-container** 项目，而无需进行任何更改。  
在登录到 OpenShift 集群之前，不要忘记 source **/usr/local/etc/ocp4.config** 文件中的变量。
3. 查看应用容器集的部署状态。容器集将处于 **CrashLoopBackoff** 或 **Error** 状态。查看应用日志以了解应用未正确启动的原因。
4. 编辑应用的 Containerfile 以确保成功部署到 OpenShift 集群。容器应使用随机用户 ID 而不是当前配置的 **wildfly** 用户运行。
5. 提交您对 Containerfile 所做的更改并将更改推送到课堂 Git 存储库。
6. 启动应用新构建。按照新构建的构建日志进行操作。验证应用容器集是否成功启动。
7. 公开服务供外部访问并测试应用。使用 **/api/hello** 上下文路径访问应用的 API。
8. 创建名为 **appconfig** 的新配置映射。在此配置映射中存储名为 **APP\_MSG** 的密钥，其值为 **Elvis lives**。将该键作为环境变量添加到应用的部署中。
9. 验证是否触发了新部署，再等待新应用容器集就绪并在运行。验证 **APP\_MSG** 密钥是否作为环境变量注入容器。
10. 通过调用其 REST API URL (**http://elvis-yourdevuser-design-container.apps.cluster.domain.example.com/api/hello**) 来测试应用，并验证 **APP\_MSG** 密钥值是否出现在响应中。

## 评估

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab design-container grade
```

## 完成

在 **workstation** 上运行 **lab design-container finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab design-container finish
```

本实验到此结束。

## ► 解决方案

# 针对 OpenShift 设计容器化应用

在本实验中，您将修复基于 Thorntail 框架的应用的 Containerfile，以便在 OpenShift 集群上运行。您还将使用配置映射来配置应用。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应该能够为基于 Thorntail 框架的应用修复 Containerfile，以作为随机用户运行，并在 OpenShift 集群上部署该应用。您还应该能够使用配置映射来存储用于配置应用的简单文本字符串。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 应用的可运行 fat JAR。
- 应用 Git 存储库。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还可以下载实验室的帮助文件和答案文件：

```
[student@workstation ~]$ lab design-container start
```

## 要求

该应用使用 Thorntail 框架以 Java 编写。提供了包含应用和 Thorntail 运行时的预构建、可运行 JAR 文件 (fat JAR)。该应用提供了简单的 REST API，它根据作为环境变量注入容器的配置来响应请求。根据以下要求，构建应用并将其部署至 OpenShift 集群：

- OpenShift 的应用名称为 **elvis**。应用的配置应存储在名为 **appconfig** 的配置映射中。
- 将应用部署到 **yourdevuser-design-container** 项目。
- 应用的 REST API 应可通过以下 URL 访问：

**elvis-yourdevuser-design-container.apps.cluster.domain.example.com/api/hello**

- 包含应用源的 Git 存储库和文件夹为：

<https://github.com/yourgithubuser/D0288-apps/hello-java>

- 预构建的应用 JAR 文件位于：

```
https://github.com/RedHatTraining/D0288-apps/releases/download/OCP-4.1-1/hello-java.jar
```

## 说明

1. 导航到 D0288-apps Git 存储库的本地克隆，再从 main 分支创建一个名为 design-container 的新分支。简要查看 /home/student/D0288-apps/hello-java/ 目录中应用的 Containerfile。

- 1.1. 查看 Git 存储库的 main 分支。

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 1.2. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b design-container
Switched to a new branch 'design-container'
[student@workstation D0288-apps]$ git push -u origin design-container
...output omitted...
* [new branch]      design-container -> design-container
Branch design-container set up to track remote branch design-container from
origin.
```

- 1.3. 检查 /home/student/D0288-apps/hello-java/Containerfile 文件。暂时不要对它进行任何更改。

2. 使用应用的 design-container 分支在 D0288-apps Git 存储库的 hello-java 文件夹中部署应用。在 OpenShift 中将应用部署到 yourdevuser-design-container 项目，而无需进行任何更改。

在登录到 OpenShift 集群之前，不要忘记 source /usr/local/etc/ocp4.config 文件中的变量。

- 2.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的配置变量：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- 2.2. 使用您的开发人员用户帐户登录 OpenShift：

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.3. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀：

```
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-design-container
```

2.4. 从 Containerfile 创建新镜像。

```
[student@workstation D0288-apps]$ podman build --layers=false \
-t do288-hello-java ./hello-java
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
STEP 11: COMMIT do288-hello-java
...output omitted...
```

2.5. 标记新镜像并推送到 Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-hello-java \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
[student@workstation D0288-apps]$ podman login quay.io -u ${RHT_OCP4_QUAY_USER}
Password:
Login Succeeded!
[student@workstation D0288-apps]$ podman push \
--format v2s1 quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
Storing signatures
```

2.6. 登录 Quay.io [http://quay.io]，使新添加的镜像变为公共镜像。如果没有这一步，new-app 命令将失败。

2.7. 使用存储库中的镜像，在 OpenShift 中新建一个名为 elvis 的应用

```
[student@workstation D0288-apps]$ oc new-app --name elvis \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "elvis" created
deployment.apps "elvis" created
service "elvis" created
--> Success
...output omitted...
```

3. 查看应用容器集的部署状态。容器集将处于 CrashLoopBackOff 或 Error 状态。查看应用日志以了解应用未正确启动的原因。

3.1. 等待应用容器集部署好。应用未达到 Ready 状态。一段时间后，它将保持在 CrashLoopBackOff 或 Error 状态。

```
[student@workstation D0288-apps]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
elvis-799f8df69b-vh6fr   0/1     Error      2          34s
```

3.2. 查看应用容器集的日志并调查应用容器集无法启动的原因。

```
[student@workstation D0288-apps]$ oc logs elvis-799f8df69b-vh6fr  
/bin/sh: /opt/app-root/bin/run-app.sh: Permission denied
```

由于文件系统中的“权限被拒绝”错误，应用无法启动，因为 OpenShift 不使用 Containerfile 中指定的用户运行容器集。

4. 编辑应用的 Containerfile 以确保成功部署到 OpenShift 集群。容器应使用随机用户 ID 而不是当前配置的 **wildfly** 用户运行。

- 4.1. 使用文本编辑器在 `/home/student/D0288-apps/hello-java/Containerfile` 处编辑 Containerfile。进行以下步骤中列出的更改。您还可以从 `/home/student/D0288/solutions/design-container/Containerfile` 提供的答案文件中复制指令和命令。

移除第一个 RUN 指令中 `useradd` 命令（第 12 行）。

```
useradd wildfly && \
```

- 4.2. 找到位于第 19 行和第 20 行的 `chown` 和 `chmod` 命令：

```
RUN chown -R wildfly:wildfly /opt/app-root && \  
chmod -R 700 /opt/app-root
```

使用以下内容进行替换：

```
RUN chgrp -R 0 /opt/app-root && \  
chmod -R g=u /opt/app-root
```

- 4.3. 将第 24 行的 `USER` 指令中的 **wildfly** 用户替换为 1001 的通用 userid，以避免从父 RHEL 镜像继承用户。OpenShift 会忽略此通用 userid，并遵循红帽构建镜像的建议和约定：

```
USER 1001
```

5. 提交您对 Containerfile 所做的更改并将更改推送到课堂 Git 存储库。

```
[student@workstation D0288-apps]$ cd hello-java  
[student@workstation hello-java]$ git commit -a -m \  
"Fixed Containerfile to run with random user id on OpenShift"  
[student@workstation hello-java]$ git push  
[student@workstation hello-java]$ cd ..  
[student@workstation D0288-apps]$
```

6. 启动应用新构建。按照新构建的构建日志进行操作。验证应用容器集是否成功启动。

- 6.1. 启动应用新构建：

```
[student@workstation D0288-apps]$ podman rmi -a --force  
...output omitted...  
[student@workstation D0288-apps]$ podman build --layers=false \  
-t do288-hello-java ./hello-java
```

## 章2 | 针对 OpenShift 设计容器化应用

```
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
STEP 7: RUN chgrp -R 0 /opt/app-root && chmod -R g=u /opt/app-root
...output omitted...
STEP 9: USER 1001
...output omitted...
```

### 6.2. 标记新镜像并推送到 Quay.io

```
[student@workstation D0288-apps]$ podman tag do288-hello-java \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
[student@workstation D0288-apps]$ podman push \
--format v2s1 quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
Storing signatures
```

### 6.3. 删除旧项目，再重新创建这个项目。

```
[student@workstation D0288-apps]$ oc delete project \
${RHT_OCP4_DEV_USER}-design-container
...output omitted...
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-design-container
```

### 6.4. 使用存储库中更新的镜像，在 OpenShift 集群中新建一个名为 elvis 的应用

```
[student@workstation D0288-apps]$ oc new-app --name elvis \
quay.io/${RHT_OCP4_QUAY_USER}/do288-hello-java
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "elvis" created
deployment.apps "elvis" created
service "elvis" created
--> Success
...output omitted...
```

### 6.5. 等待应用容器集准备就绪并在运行。

```
[student@workstation D0288-apps]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
elvis-6d4fc74867-b6z2h   1/1     Running   0          29s
```

### 6.6. 查看应用容器集的日志，并确认启动期间没有错误：

```
[student@workstation ~]$ oc logs elvis-6d4fc74867-b6z2h
Starting hello-java app...
JVM options => -Xmx512m
...output omitted...
2021-06-22 17:43:56,211 INFO [org.wildfly.extension.undertow] (MSC service thread
1-2) WFLYUT0018: Host default-host starting
...output omitted...
2021-06-22 17:43:56,484 INFO [org.wildfly.swarm] (main) THORN99999: Thorntail is
Ready
```

7. 公开服务供外部访问并测试应用。使用 `/api/hello` 上下文路径访问应用的 API。

7.1. 公开应用供外部访问。

```
[student@workstation ~]$ oc expose svc/elvis
route.route.openshift.io/elvis exposed
```

7.2. 标识公开应用 API 的主机名：

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
elvis     elvis-yourdevuser-design-container.apps.cluster.domain.example.com
```

7.3. 使用上一步中标识的主机名，通过调用 API URL (`http://elvis-yourdevuser-design-container.apps.cluster.domain.example.com/api/hello`) 来测试应用，并验证应用容器集名称是否出现在响应中：

```
[student@workstation ~]$ curl \
http://elvis-${RHT_OCP4_DEV_USER}-design-container.${RHT_OCP4_WILDCARD_DOMAIN}\ \
/api/hello
Hello world from host elvis-6d4fc74867-b6z2h
```

8. 创建名为 `appconfig` 的新配置映射。在此配置映射中存储名为 `APP_MSG` 的密钥，其值为 `Elvis lives`。将该键作为环境变量添加到应用的部署中。

8.1. 创建配置映射：

```
[student@workstation ~]$ oc create cm appconfig \
--from-literal APP_MSG="Elvis lives"
configmap/appconfig created
```

8.2. 查看配置映射的详细信息：

```
[student@workstation ~]$ oc describe cm/appconfig
Name:  appconfig
...output omitted...

Data
=====
APP_MSG:
```

```
---  
Elvis lives  
Events: <none>
```

8.3. 使用 `oc set env` 命令将配置映射添加到部署配置：

```
[student@workstation ~]$ oc set env deployment/elvis --from cm/appconfig  
deployment.apps/elvis updated
```

9. 验证是否触发了新部署，再等待新应用容器集就绪并在运行。验证 `APP_MSG` 密钥是否作为环境变量注入容器。

9.1. 验证是否已触发新部署：

```
[student@workstation ~]$ oc status  
...output omitted...  
deployment/elvis deploys istag/elvis:latest  
  deployment #3 running for 21 seconds - 1 pod  
  deployment #2 deployed 16 minutes ago  
  deployment #1 deployed 16 minutes ago  
...output omitted...
```

9.2. 等待重新部署应用容器集。验证新应用容器集准备就绪并在运行：

```
[student@workstation D0288-apps]$ oc get pods  
NAME           READY   STATUS    RESTARTS   AGE  
elvis-66c7f6d47f-ll2jq   1/1     Running   0          2m36s
```

9.3. 验证 `APP_MSG` 密钥是否作为环境变量注入容器：

```
[student@workstation ~]$ oc rsh elvis-66c7f6d47f-ll2jq env | grep APP_MSG  
APP_MSG=Elvis lives
```

10. 通过调用其 REST API URL (`http://elvis-yourdevuser-design-container.apps.cluster.domain.example.com/api/hello`) 来测试应用，并验证 `APP_MSG` 密钥值是否出现在响应中。

```
[student@workstation ~]$ curl \  
http://elvis-${RHT_OCP4_DEV_USER}-design-container.${RHT_OCP4_WILDCARD_DOMAIN}\ \  
/api/hello  
Hello world from host [elvis-66c7f6d47f-ll2jq]. Message received = Elvis lives
```

## 评估

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab design-container grade
```

## 完成

在 `workstation` 上运行 `lab design-container finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab design-container finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- RHOCP 容器部署选项包括：
  - 直接在 OpenShift 集群上部署预构建的容器镜像。
  - 使用基础镜像创建 Dockerfile 并根据您的需求进行自定义。
  - 使用源至镜像 (S2I) 构建，其中 RHOCP 会将源代码与构建器镜像相结合。
- 在 RHOCP 上运行容器所需的对 Dockerfile 的常见更改：
  - 对容器中的进程读取或写入的文件的根组权限。
  - 可执行文件必须具有组执行权限。
  - 在容器中运行的进程不得侦听特权端口（低于 1024 的端口）。
- 使用机密来存储敏感信息并从您的容器集中访问。
- 使用配置映射资源存储非敏感环境特定数据。

## 章 3

# 发布企业容器镜像

### 目标

与企业注册表交互，并向其发布容器镜像。

### 培训目标

- 使用 Linux 容器工具管理注册表中的容器镜像。
- 使用 Linux 容器工具访问 OpenShift 内部注册表。
- 为外部注册表中的容器镜像创建镜像流。

### 章节

- 管理企业注册表中的镜像（及引导式练习）
- 允许访问 OpenShift 注册表（及引导式练习）
- 创建镜像流（及引导式练习）

### 实验

发布企业容器镜像

# 管理企业注册表中的镜像

---

## 培训目标

学完本节后，您应能够使用 Linux 容器工具管理注册表中的容器镜像。

## 回顾容器注册表

容器镜像注册表、容器注册表或注册表服务器存储作为容器部署的镜像，并提供拉取、推送、更新、搜索和删除容器镜像的机制。它使用由开放容器项目 (OCI) 定义的标准 REST API，该 API 基于 Docker 注册表 HTTP API v2。从运行 OpenShift 集群的组织的角度来看，有许多种容器注册表：

### 公共注册表

允许任何人直接从互联网使用容器镜像而无需任何身份验证的注册表。Docker Hub、Quay.io 和红帽注册表是公共容器注册表的示例。

### 私有注册表

仅对选定的使用者可用的注册表，通常需要身份验证。基于红帽术语的注册表是私有容器注册表的示例。

### 外部注册表

不受您组织控制的注册表。它们通常由云提供商或软件供应商管理。Quay.io 是外部容器注册表的示例。

### 企业注册表

您的组织管理的注册表服务器。它们通常仅对组织的员工和承包商可用。

### OpenShift 内部注册表

由 OpenShift 集群在内部管理的注册表服务器，用于存储容器镜像。使用 OpenShift 的构建配置和 S2I 进程或 Containerfile 创建这些镜像，或者从其他注册表中导入它们。

这些类型的注册表不是相互排斥的：注册表可以同时是公共和私有注册表。通常，公共注册表也是外部注册表，因为您的组织可以通过互联网访问它，无需身份验证，并且您的组织不控制它。

如果组织与注册表提供商有一个计划，允许您托管私有镜像，并且您的组织还可以控制其他哪些人可以访问这些私有镜像，则同一注册表也可以是私有注册表。

对于某些用户，Quay.io 同时充当公共注册表和私有注册表。同一开发人员可以使用来自 Quay.io 的一些公共容器镜像，以及来自供应商的一些需要身份验证的容器镜像。

## 红帽管理的注册表

红帽管理一组公共和私有容器注册表，为不同的受众提供不同类型的容器镜像。红帽或其合作伙伴以生产级 SLA 支持的容器镜像可通过红帽容器目录进行访问。社区和不受支持的容器镜像可通过 Quay.io 访问。

位于 <https://access.redhat.com/containers> 的红帽容器目录是一个 Web 用户界面，您可通过它浏览和搜索这些注册表，并获取有关基于红帽企业 Linux 的镜像的详细信息。

红帽提供的镜像得益于红帽管理红帽企业 Linux 及其他产品中安全漏洞和缺陷的长期经验。红帽安全团队强化并控制这些优质镜像，然后对这些镜像进行签名以防止篡改。每当发现新漏洞时，红帽也会重建这些镜像并执行质量保证流程。

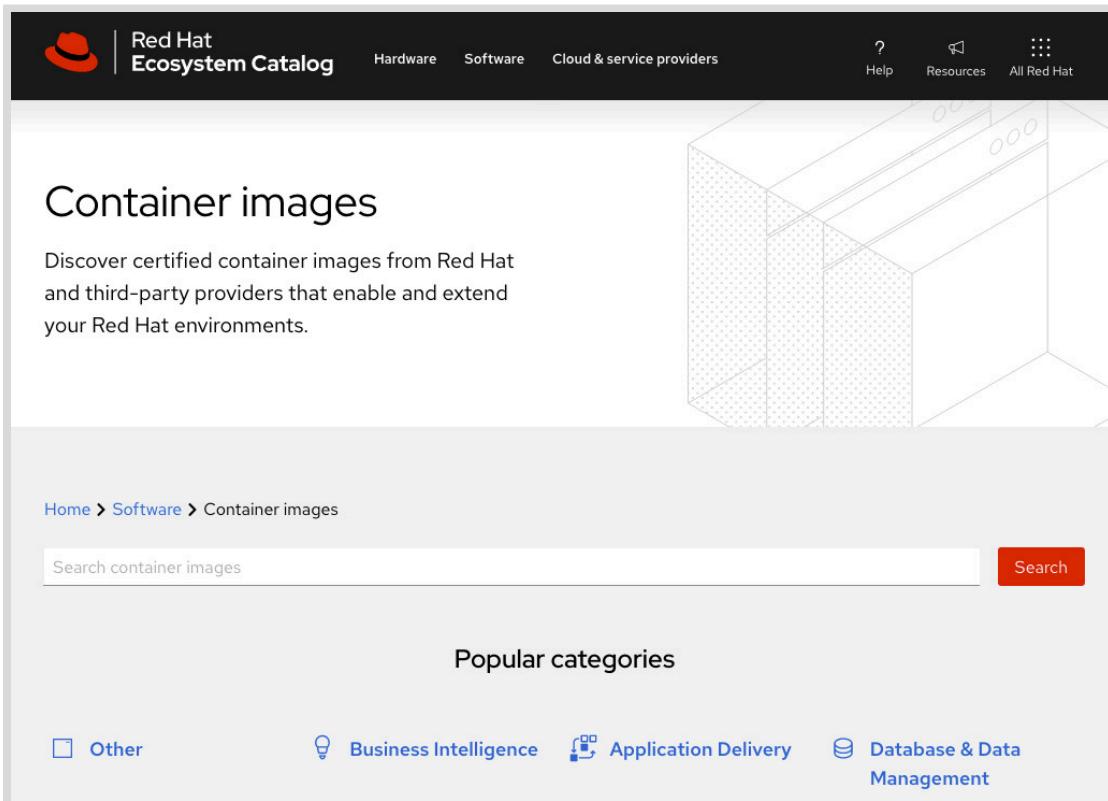


图 3.1: 红帽容器目录

业务关键型应用应尽可能依赖这些受到信任和支持的镜像，而不要使用来自其他一些公共注册表的镜像。那些镜像可能没有得到充分测试或维护，也不会在检测到新安全问题时及时更新。

红帽容器目录为以下三个底层容器注册表提供了一个统一视图：

#### 位于 `registry.access.redhat.com` 的红帽容器注册表

这是一个公共注册表，托管红帽产品的镜像且无需身份验证。请注意，虽然此容器注册表是公共的，但红帽提供的大多数容器镜像都要求用户具有有效的红帽产品订阅，并且它们遵循产品的最终用户协议 (EUA)。只有红帽公共注册表中可用的镜像子集可以自由再分发。这些镜像基于红帽企业 Linux 通用基础镜像 (UBI)。

#### 位于 `registry.redhat.io` 的基于红帽术语的注册表

这是一个私有注册表，托管红帽产品的镜像且需要身份验证。若要从中拉取镜像，您需要利用红帽客户门户凭据进行身份验证。对于共享环境，如 OpenShift 或 CI/CD 管道，您可以创建服务帐户或身份验证令牌，以避免暴露您的个人凭据。

#### 位于 `registry.connect.redhat.com` 的红帽合作伙伴注册表

这是一个私有注册表，托管来自认证合作伙伴的第三方产品的镜像。它也需要您使用红帽客户门户凭据进行身份验证。根据合作伙伴的决定，它们可能需要订阅或许可。

## Quay.io 注册表

红帽还管理 Quay.io 容器注册表，任何人都可以注册免费帐户并发布自己的容器镜像。

红帽不对托管在 Quay.io 的任何容器镜像提供保证。它们的范围可能涵盖完全未经维护的一次性实验；来自无服务级别协议 (SLA) 的开源社区的良好、稳定且维护妥当的容器镜像；以及供应商完全支持的产品，可能提供免费的、未经身份验证的容器镜像访问权限，以便进行产品试用。

### 章 3 | 发布企业容器镜像

大多数用户使用 Quay.io 作为公共注册表，但组织也可以购买允许将 Quay.io 用作私有注册表的计划。

## 部署企业容器注册表

通过互联网访问外部、公共或私有注册表非常方便，但许多组织不允许开发人员拉取和运行外部容器镜像。这些企业限制开发人员只能使用一组符合安全性、质量和一致性标准的容器镜像。

依靠外部注册表为您的生产主机拉取和推送镜像并非毫无风险。例如，注册表由于故障或提供商计划内维护而停机时，您无法部署新的容器。发生故障时，OpenShift 自动扩展也会失败，因为 OpenShift 无法拉取所需的镜像来启动额外的容器集。根据您的带宽，在互联网上拉取和推送镜像也可能是缓慢的过程。

在某些组织中，容器镜像仅供内部使用，不能公开。设置企业注册表可解决此问题。建立企业注册后，在组织内配置容器主机，以便仅从此注册表中拉取镜像，而不是默认的外部注册表。

通过在组织中运行注册表服务器，可以降低风险并实现其他功能。例如，您可以创建不同的环境，并控制谁可以在其中推送或拉取镜像。您可以定义一个审批工作流，将经过验证的镜像从开发环境移动到生产环境。您可以实施漏洞扫描，并在扫描程序检测到生产中的镜像存在缺陷时发送通知。

## 注册表服务器软件

可用的注册表服务器软件包括红帽 Quay Enterprise、开源 Docker 分发服务器，以及诸如 JFrog 和 Nexus 等产品。OpenShift 可以从这些注册表服务器中的任何一个部署容器。

红帽 Quay Enterprise 是一个包含诸多高级功能的容器镜像注册表，如镜像安全扫描、基于角色的访问、组织和团队管理、镜像构建自动化、审计、异地复制和高可用性等。

红帽 Quay Enterprise 提供 Web 界面和 REST API。它可以在本地、选定的云提供商以及红帽 OpenShift 容器平台上作为容器进行部署。它也是 Quay.io 背后的服务器软件。

如果在 OpenShift 上部署 Quay Enterprise，它不会替换集群的内部注册表。对于所有实际用途，在 OpenShift 集群上运行的 Quay Enterprise 实例与任何其他 OpenShift 应用一样，通常可用于组织中的其他 OpenShift 集群和任何其他容器主机。

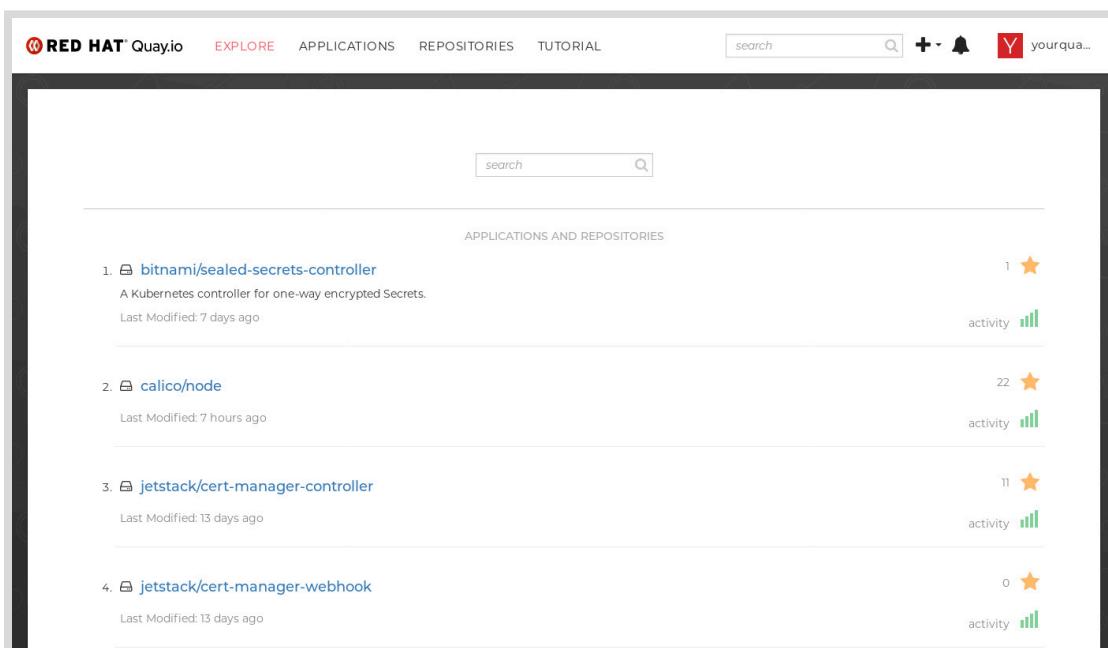


图 3.2: Quay.io 网页

## 访问容器注册表

从 OpenShift 访问公共注册表通常不需要配置，因为公共注册表应该显示由受信任的证书颁发机构 (CA) 提供的 TLS 证书。

从 OpenShift 访问私有注册表通常需要其他配置来管理身份验证凭据和令牌。企业注册表可能还需要配置内部 CA。

要访问容器注册表，请使用 OCI 分发 API，该 API 基于较旧的 Docker 注册表 API。要使用 API，红帽建议您使用红帽企业 Linux (RHEL) 容器工具：Podman、Buildah 和 Skopeo。RHEL 容器工具可以基于 OCI 标准自定义、部署和调试容器镜像。

开放容器项目 (OCI) 组织定义了容器运行时、容器镜像格式和相关 REST API 的开放标准。OCI 容器镜像格式是文件系统文件夹，包含存储容器镜像清单、元数据和图层的离散文件。

## 使用 Podman 管理容器

Podman 是一种工具，用于管理容器集和容器，而无需容器守护进程，从而减少攻击面并提高性能。容器集和容器进程是作为 `podman` 命令的子进程来创建的。

Podman 可以重新启动已停止的容器，还可以推送、提交、配置、构建和创建容器镜像。`podman` 命令通常遵循 `docker` 命令语法并提供其他功能，如管理容器集。Podman 不支持 `docker` 与容器引擎无关的命令，例如群集式模式。

## 向注册表进行身份验证

要访问私有注册表，通常需要进行身份验证。Podman 提供生成访问令牌并将其存储以供后续重用的 `login` 子命令。

```
[user@host ~]$ podman login quay.io
Username: developer1
Password: Mys3cret!
Login Succeeded!
```

身份验证成功后，Podman 会将访问令牌存储在 `/run/user/UID/containers/auth.json` 文件中。`/run/user/UID` 路径前缀不是固定的，并且来自 `XDG_RUNTIME_DIR` 环境变量。

您可以使用 Podman 同时登录多个注册表。每个新登录名都会在同一文件中添加或更新访问令牌。每个访问令牌都由注册表服务器 FQDN 编制索引。

若要从某一注册表注销，请使用 `logout` 子命令：

```
[user@host ~]$ podman logout quay.io
Remove login credentials for quay.io
```

要注销所有注册表，丢弃所有已存储供重用的访问令牌，请使用 `--all` 选项：

```
[user@host ~]$ podman logout --all
Remove login credentials for all registries
```

Skopeo 和 Buildah 也可以使用 Podman 存储的身份验证令牌，但它们无法显示交互式密码提示。

### 章3 | 发布企业容器镜像

默认情况下，Podman 需要 TLS 和远程证书验证。如果您的注册表服务器未配置为使用 TLS，或者配置为使用自签名 TLS 证书或由未知 CA 签名的 TLS 证书，则可以将 `--tls-verify=false` 选项添加到 `login` 和 `pull` 子命令中。

## 使用 Skopeo 管理容器注册表

红帽支持使用 `skopeo` 命令管理容器镜像注册表中的镜像。Skopeo 不使用容器引擎，因此它比使用 Podman 中的 `tag`、`pull`、和 `push` 子命令更高效。

Skopeo 还提供 Podman 中没有的其他功能，例如签署和删除注册表服务器中的容器镜像。

`skopeo` 命令采用子命令、选项和参数：

```
[user@host ~]$ skopeo subcommand [options] location...
```

### 主要子命令

- `copy` 将镜像从一个位置复制到另一个位置。
- `delete` 从注册表删除镜像。
- `inspect` 查看有关镜像的元数据。

### 主要选项

- `--creds username:password`  
向注册表提供登录凭据或身份验证令牌。
- `--[src-|dest-]tls-verify=false`  
禁用 TLS 证书验证。

对于私有注册表的身份验证，Skopeo 也可以使用由 `podman login` 命令创建的同一 `auth.json` 文件。或者，您可以在命令行上传递凭据，如下所示。

```
[user@host ~]$ skopeo inspect --creds developer1:MyS3cret! \
docker://registry.redhat.io/rhscl/postgresql-96-rhel7
```



#### 警告

尽管可以为命令行工具提供凭据，但这会在命令历史记录中创建一个条目并带来其他安全问题。使用技术避免将纯文本凭据传递给命令：

```
[user@host ~]$ read -p "PASSWORD: " -s password
PASSWORD:
[user@host ~]$ skopeo inspect --creds developer1:$password \
docker://registry.redhat.io/rhscl/postgresql-96-rhel7
```

Skopeo 使用 URI 表示容器镜像位置，使用 URI 架构表示容器镜像格式和注册表 API。下面的列表显示了最常见的 URI 架构：

#### oci

表示存储在本地 OCI 格式文件夹中的容器镜像。

### 章 3 | 发布企业容器镜像

#### docker

表示存储在注册表服务器中的远程容器镜像。

#### containers-storage

表示存储在本地容器引擎缓存中的容器镜像。

## 在注册表服务器中推送和标记镜像

Skopeo 中的 `copy` 子命令可以直接在注册表之间复制容器镜像，而无需将镜像层保存在本地容器存储中。它还可以将容器镜像从本地容器引擎复制到注册表服务器，并在单个操作中标记这些镜像。

要将名为 `myimage` 的容器镜像从本地容器引擎复制到 `myorg` 组织或用户帐户下位于 `registry.example.com` 的不安全公共注册表：

```
[user@host ~]$ skopeo copy --dest-tls-verify=false \
containers-storage:myimage \
docker://registry.example.com/myorg/myimage
```

要将容器镜像从 `/home/user/myimage` OCI 格式的文件夹复制到 `myorg` 组织或用户帐户下位于 `registry.example.com` 的不安全公共注册表：

```
[user@host ~]$ skopeo copy --dest-tls-verify=false \
oci:/home/user/myimage \
docker://registry.example.com/myorg/myimage
```

在私有注册表之间复制容器镜像时，可以在调用 `copy` 子命令之前使用 Podman 对两个注册表进行身份验证，或者使用 `--src-creds` 和 `--dest-creds` 选项指定身份验证凭据，如下所示：

```
[user@host ~]$ skopeo copy --src-creds=testuser:testpassword \
--dest-creds=testuser1:testpassword \
docker://srcregistry.domain.com/org1/private \
docker://dstregistry.domain2.com/org2/private
```

`skopeo` 命令的参数始终是完整的镜像名称。以下示例是无效命令，因为它仅提供注册表服务器名称作为目标参数：

```
[user@host ~]$ skopeo copy oci:myimage \
docker://registry.example.com/
```

Skopeo `copy` 命令还可以标记远程存储库中的镜像。下例将标签为 `1.0` 的现有镜像标记为 `latest`：

```
[user@host ~]$ skopeo copy docker://registry.example.com/myorg/myimage:1.0 \
docker://registry.example.com/myorg/myimage:latest
```

为提高效率，Skopeo 不会读取或发送目标中已存在的镜像层。它首先读取源镜像清单，然后确定哪些层已存在于目标中，然后仅复制缺少的层。如果复制从同一父级构建的多个镜像，Skopeo 不会多次复制父层。

## 从注册表删除镜像

要从位于 `registry.example.com` 的注册表中删除 `myorg/myimage` 容器镜像，请运行以下命令：

```
[user@host ~]$ skopeo delete docker://registry.example.com/myorg/myimage
```

delete 子命令可以选择采用 `--creds` 和 `--tls-verify=false` 选项。

## 为 OpenShift 进行私有注册表身份验证

OpenShift 还需要凭据才能访问私有注册表中的容器镜像。这些凭据存储为机密。

您可以直接向 `oc create secret` 命令提供私有注册表凭据：

```
[user@host ~]$ oc create secret docker-registry registrycreds \
--docker-server registry.example.com \
--docker-username youruser \
--docker-password yourpassword
```

创建机密的另一种方法是使用 `podman login` 命令中的身份验证令牌：

```
[user@host ~]$ oc create secret generic registrycreds \
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
```

然后将该机密从您的项目链接到 `default` 服务帐户：

```
[user@host ~]$ oc secrets link default registrycreds --for pull
```

要使用该机密访问 S2I 构建器镜像，请将该机密从您的项目中链接到 `builder` 服务帐户：

```
[user@host ~]$ oc secrets link builder registrycreds
```



### 参考文献

#### 开放容器项目 (OCI)

<https://www.opencontainers.org/>

#### 容器术语实用简介

<https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction>

#### 红帽容器 Registry 身份验证

<https://access.redhat.com/RegistryAuthentication>

有关 RHEL 容器工具的详细信息，请访问红帽企业 Linux 8 的 Building, running, and managing containers 指南，网址为：

[https://access.redhat.com/documentation/en-us/red\\_hat\\_enterprise\\_linux/8/html-single/building\\_running\\_and\\_managing\\_containers/index](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html-single/building_running_and_managing_containers/index)

## ▶ 指导练习

# 使用企业注册表

在本练习中，您将与容器镜像注册表服务器交互。

## 成果

您应能够：

- 将镜像推送到经过身份验证的外部容器注册表。
- 使用经过身份验证的外部容器注册表作为输入，将容器化应用部署到 OpenShift。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- podman 和 skopeo 命令。
- 示例 ubi-sleep 容器镜像的 OCI 兼容文件。

在 workstation 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab external-registry start
```

## 说明

▶ 1. 登录到外部注册表，并从磁盘上的 OCI 兼容文件夹推送镜像。

- 1.1. 检查本地磁盘中的 ubi-sleep 容器 OCI 镜像层。OCI 镜像存储为包含多个文件的文件系统文件夹：

```
[student@workstation ~]$ ls ~/D0288/labs/external-registry/ubi-sleep
blobs  index.json  oci-layout
```

- 1.2. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.3. 使用 Podman 登录您的个人 Quay.io 帐户。Podman 会提示您输入 Quay.io 密码。

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 1.4. 使用 Skopeo 将 OCI 镜像复制到位于 Quay.io 的外部注册表并将其标记为 1.0。

### 3 | 发布企业容器镜像

您也可以从 `/home/student/D0288/labs/external-registry` 文件夹中的 `push-image.sh` 脚本执行或剪切和粘贴以下 `skopeo copy` 命令。不要公开这个存储库。此存储库必须保持私有，仅供本实验使用。

```
[student@workstation ~]$ skopeo copy --format v2s1 \
oci:/home/student/D0288/labs/external-registry/ubi-sleep \
docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

1.5. 使用 Skopeo 检查外部注册表中的镜像：

```
[student@workstation ~]$ skopeo inspect \
docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
{
  "Name": "quay.io/yourquayuser/ubi-sleep",
  "Tag": "1.0",
  ...output omitted...
```

## ▶ 2. 验证 Podman 是否可以从外部注册表运行镜像。

2.1. 从外部注册表中的镜像启动测试容器。

```
[student@workstation ~]$ podman run -d --name sleep \
quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
Trying to pull quay.io/youruser/ubi-sleep:1.0...Getting image source signatures
...output omitted...
```

2.2. 验证新容器是否正在运行：

```
[student@workstation ~]$ podman ps
CONTAINER ID        IMAGE               ...   NAMES
63c5167376e5      quay.io/youruser/ubi-sleep:1.0 ...   sleep
```

2.3. 验证新容器是否生成日志输出：

```
[student@workstation ~]$ podman logs sleep
...output omitted...
sleeping
sleeping
```

2.4. 停止并删除测试容器：

```
[student@workstation ~]$ podman stop sleep
...output omitted...
[student@workstation ~]$ podman rm sleep
...output omitted...
```

## ▶ 3. 基于外部注册表中的镜像将应用部署到 OpenShift:

## 章3 | 发布企业容器镜像

3.1. 登录 OpenShift 并创建新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-external-registry
Now using project "youruser-docker-build" on server "https://
api.cluster.domain.example.com:6443".
```

3.2. 尝试从外部注册表中的容器镜像部署应用。它将失败，因为 OpenShift 需要凭据才能访问外部注册表。

```
[student@workstation ~]$ oc new-app --name sleep \
--image quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
...output omitted...
error: unable to locate any local docker images with name "quay.io/yourquayuser/
ubi-sleep:1.0"
...output omitted...
```

3.3. 从 Podman 存储的容器注册表 API 访问令牌创建机密。

您也可以从 /home/student/D0288/labs/external-registry 文件夹中的 create-secret.sh 脚本执行或剪切和粘贴以下 oc create secret 命令。

```
[student@workstation ~]$ oc create secret generic quayio \
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
secret/quayio created
```

3.4. 将新机密链接到 default 服务帐户。

```
[student@workstation ~]$ oc secrets link default quayio --for pull
```

3.5. 从外部注册表中的容器镜像部署应用。这一次，OpenShift 可以访问外部注册表。

```
[student@workstation ~]$ oc new-app --name sleep \
--image quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "sleep" created
deployment.apps "sleep" created
--> Success
...output omitted...
```

3.6. 等待应用容器集准备就绪并在运行：

```
[student@workstation ~]$ oc get pod
NAME                  READY   STATUS    RESTARTS   AGE
sleep-7bf77b7596-ldrs 1/1     Running   0          95s
```

3.7. 验证容器集是否生成日志输出：

```
[student@workstation ~]$ oc logs sleep-7bf77b7596-ldrsl
...output omitted...
sleeping
sleeping
```

- 4. 删 除 OpenShift 中的项目以及外部容器注册表中的容器和镜像。由于 Quay.io 允许恢复旧的容器镜像，因此还需要在 Quay.io 上删除存储库。

4.1. 删 除 OpenShift 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-registry
project.project.openshift.io "external-registry" deleted
```

4.2. 从外部注册表中删除容器镜像：

```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/ubi-sleep:1.0
```

4.3. 使用您的个人免费帐户登录 Quay.io。

导航到 <http://quay.io> 并单击 **Sign In** 以提供用户凭据。登录 Quay.io。

- 4.4. 在 Quay.io 主菜单上，单击 **Repositories** 并查找 **ubi-sleep**。单击 **ubi-sleep** 以显示 **Repository Activity** 页面。
- 4.5. 在 **ubi-sleep** 存储库的 **Repository Activity** 页面，向下滚动并单击齿轮图标以显示 **Settings** 选项卡。向下滚动并单击 **Delete Repository**。
- 4.6. 在 **Delete** 对话框中，输入存储库名称，单击 **Delete** 以确认要删除 **ubi-sleep** 存储库。几分钟后，您将返回到 **Repositories** 页面。您现在可以注销 Quay.io。

## 完成

在 **workstation** 虚拟机上，运行 **lab external-registry finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab external-registry finish
```

本引导式练习到此结束。

# 允许访问 OpenShift 注册表

## 培训目标

学完本节后，您应能够通过 Linux 容器工具访问 OpenShift 内部注册表。

## 查看内部注册表

OpenShift 运行内部注册表服务器，以支持基于源至镜像 (S2I) 的开发人员工作流。开发人员可使用 `oc new-app` 命令和其他方法创建构建配置，以使用 S2I 创建新的容器镜像。构建配置从源代码或 Containerfile 创建容器镜像，并将它们存储在内部注册表中。

开发人员实际上不需要使用内部注册表。他们可以使用红帽企业 Linux 容器工具在本地构建容器镜像，并将它们推送到 OpenShift 可以访问的外部公共或私有注册表。它们还可以设置其构建配置，将最终镜像直接推送到外部公共或私有注册表。

默认情况下，OpenShift 安装程序部署内部注册表，以便开发人员可以在 OpenShift 集群可供他们使用后立即开始开发工作。如果没有内部注册表，开发人员需要等待，直到有人部署注册表服务器并为他们提供访问凭据。在能够执行任何开发工作之前，他们还必须了解如何配置机密以访问私有注册表。

对于在 S2I 进程之外访问 OpenShift 内部注册表，存在一些用例。例如：

- 组织已在本地构建容器镜像，尚未准备好更改其开发工作流。这些组织可能具有功能有限的私有注册表，并希望将其替换为 OpenShift 内部注册表。
- 组织维护多个 OpenShift 集群，需要将容器镜像从开发复制到生产集群。这些组织可能具有 CI/CD 工具，用于将镜像从内部注册表提升为外部注册表或其他内部注册表。
- 独立软件供应商 (ISV) 为其客户创建容器镜像，并将其发布到由云服务提供商（如 Quay.io）维护的私有注册表或 ISV 维护的企业注册表。

OpenShift 内部注册表可以提供客户需要的所有功能，例如基于 OpenShift 用户、组和角色的细粒度访问控制。例如，如果内部注册表基于 Docker 分发注册表服务器，与组织当前的企业注册表相比，它可以提供更好的功能或更高的易用性。这些组织可以逐步淘汰其当前注册表，并使用 OpenShift 内部注册表作为新的企业注册表。

其他客户可能需要高级功能，如镜像安全扫描和异地复制，并采用更强大的企业注册服务器，如红帽 Quay Enterprise。采用更强大企业注册表的客户可能仍然需要公开内部注册表，以便能够将镜像复制到组织的企业注册表。

## 镜像注册表操作程序

OpenShift 安装程序将内部注册表配置为仅可从其 OpenShift 集群内部访问。公开内部注册表以进行外部访问是一个简单的过程，但需要集群管理特权。

OpenShift 镜像注册表操作程序管理内部注册表。镜像注册表操作程序中的所有配置设置都在 `openshift-image-registry` 项目下的 `cluster` 配置资源中。将 `spec.defaultRoute` 属性更改为 `true`，镜像注册表运算符会创建路由以公开内部注册表。执行此更改的一种方法使用以下 `oc patch` 命令：

```
[user@host ~] oc patch config.imageregistry cluster -n openshift-image-registry \
--type merge -p '[{"spec":{"defaultRoute":true}}]'
```

**default-route** 路由使用部署到集群的应用的默认通配符域名：

```
[user@host ~] oc get route -n openshift-image-registry
NAME          HOST/PORT
default-route  default-route-openshift-image-registry.domain.example.com ...
```

## 向内部注册表进行身份验证

要使用 Linux 容器工具登录内部注册表，您需要获取用户的 OpenShift 身份验证令牌。

使用 `oc whoami -t` 命令获取令牌。令牌是随机长字符串。如果将令牌另存为 shell 变量，则更容易键入命令：

```
[user@host ~] TOKEN=$(oc whoami -t)
```

将令牌用作来自 Podman 的 `login` 子命令的一部分：

```
[user@host ~] podman login -u myuser -p ${TOKEN} \
default-route-openshift-image-registry.domain.example.com
```

您还可以使用令牌作为 Skopeo `--[src|dst]creds` 选项中的值。

```
[user@host ~] skopeo inspect --creds=myuser:${TOKEN} \
docker://default-route-openshift-image-registry.domain.example.com/...
```

## 将内部注册表作为安全或不安全注册表访问

如果您的 OpenShift 集群配置了用于其通配符域的有效 TLS 证书，则可以使用 Linux 容器工具处理您有权访问的任何项目内的镜像。

下面的示例使用 Skopeo 检查 `myproj` 项目中的 `myapp` 应用的容器镜像。它假定之前的 `podman login` 已成功。

```
[user@host ~] skopeo inspect \
docker://default-route-openshift-image-registry.domain.example.com/myproj/myapp
```

如果 OpenShift 集群使用 OpenShift 安装程序默认生成的证书颁发机构 (CA)，则需要将内部注册表作为不安全的注册表访问：

```
[user@host ~] skopeo inspect --tls-verify=false \
docker://default-route-openshift-image-registry.domain.example.com/myproj/myapp
```

集群管理员可以通过不同的方式配置内部注册表的路由，例如使用组织维护的内部 CA。在这种情况下，您的开发人员工作站可能已配置为信任内部注册表的 TLS 证书，也可能尚未配置。

您的组织还可能检索其 OpenShift 集群的内部 CA 的公共证书，并声明它在组织内部受信任。集群管理员可能会设置具有较短主机名的替代路由，以公开内部注册表。

这些场景不在本课程的范围之内。有关为 OpenShift 和本地容器引擎配置 TLS 证书的详细信息，请参阅有关 OpenShift 管理跟踪的红帽培训课程，如 Red Hat OpenShift Administration I (DO280) 和 Red Hat Security: Securing Containers and OpenShift (DO425)。

## 授予对内部注册表中镜像的访问权限

有权访问 OpenShift 项目的任何用户都可以根据其访问级别在该项目中推送和拉取镜像。如果用户具有项目的 **admin** 或 **edit** 角色，他们可以在该项目中拉取和推送镜像。如果他们只有项目的 **view** 角色，则他们只能从该项目中拉取镜像。

OpenShift 还提供了一些专门角色，用于仅授予对项目内镜像的访问权限，并且不授予执行其他开发任务（如在项目内构建和部署应用）的访问权限。最常见的角色有：

### **registry-viewer** 和 **system:image-puller**

这些角色允许用户从内部注册表拉取和检查镜像。

### **registry-editor** 和 **system:image-pusher**

这些角色允许用户将镜像推送和标记到内部注册表。

**system:\*** 角色提供将镜像拉取和推送到内部注册表所需的最低功能。如前所述，已在项目中具有 **admin** 或 **edit** 角色的 OpenShift 用户不需要这些 **system:\*** 角色。

对于希望将内部注册表用作企业注册表的组织，**registry-\*** 角色提供了有关注册表管理的更全面的功能。这些角色授予诸如创建新项目等附加权限，但不授予其他的权限，如构建和部署应用。OCI 标准没有指定如何管理镜像注册表，因此无论谁管理 OpenShift 内部注册表，都需要了解 OpenShift 管理概念和 **oc** 命令。这使得 **registry-\*** 用处不那么大。

以下示例允许用户从给定项目中的内部注册表拉取镜像。您需要具有项目或集群范围的管理员访问权限，才能使用 **oc policy** 命令。

```
[user@host ~] oc policy add-role-to-user system:image-puller \
user_name -n project_name
```

一般 OpenShift 身份验证和授权概念不在本课程的讨论范围之内。有关为 OpenShift 和本地容器引擎配置 TLS 证书的详细信息，请参阅有关 OpenShift 管理跟踪的红帽培训课程，如 Red Hat OpenShift Administration I (DO280) 和 Red Hat Security: Securing Containers and OpenShift (DO425)。



### 参考文献

有关公开内部注册表的更多信息，请参阅红帽 OpenShift 容器平台 4.10 的 Registry 指南中的 Image Registry Operator in OpenShift Container Platform 章节，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/registry/index#configuring-registry-operator](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/registry/index#configuring-registry-operator)

有关授予对内部注册表镜像的访问权限的详细信息，请访问红帽 OpenShift 容器平台 4.10 Registry 指南的 Accessing the registry 章节，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/registry/index#accessing-the-registry](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/registry/index#accessing-the-registry)

## ▶ 指导练习

# 允许访问 OpenShift 注册表

在本练习中，您将使用 Linux 容器工具访问 OpenShift 内部注册表。

## 成果

您应能够：

- 使用 Linux 容器工具将镜像推送到内部注册表。
- 使用 Linux 容器工具从内部注册表创建容器。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的 OpenShift 集群，其内部注册表已公开。
- `podman` 和 `skopeo` 命令。
- 示例 `ubi-info` 容器镜像的 OCI 兼容文件。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab expose-registry start
```

## 说明

### ▶ 1. 验证 OpenShift 集群的内部注册表是否已公开。

1.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 为了便于键入，将内部注册表路由的主机名保存到 shell 变量中。

系统管理员必须提供此路由的值。

```
[student@workstation ~]$ INTERNAL_REGISTRY=\
default-route-openshift-image-registry.apps.ocp4.example.com
```

**注意**

RHOCP 内部注册表的默认路由默认为禁用。系统管理员明确启用实验室环境的路由。

有关启用内部注册表默认路由的更多信息，请参阅红帽 OpenShift 容器平台 4.10 文档中的公开注册表一章，网址为：[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html/registry/securing-exposing-registry](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html/registry/securing-exposing-registry)

- 1.3. 通过登录注册表并使用开发人员的身份验证令牌和注册表主机名，验证通向 RHOCP 内部注册表的路由是否可用。如果公开了通向 RHOCP 内部注册表的默认路由，则开发人员用户可使用 **podman** 登录。

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_DEV_USER} \
-p ${oc whoami -t} ${INTERNAL_REGISTRY}
Login Succeeded!
```

- ▶ 2. 使用开发人员用户帐户将镜像推送到 OpenShift 内部注册表。

- 2.1. 使用您的开发人员用户帐户登录 OpenShift。

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 2.2. 创建项目来托管管理您将推送到 OpenShift 内部注册表的容器镜像的镜像流：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

- 2.3. 检索开发人员用户帐户的 OpenShift 身份验证令牌以用于以后的命令：

```
[student@workstation ~]$ TOKEN=$(oc whoami -t)
```

- 2.4. 验证 **ubi-info** 文件夹是否包含 OCI 格式的容器镜像：

```
[student@workstation ~]$ ls ~/D0288/labs/expose-registry/ubi-info
blobs  index.json  oci-layout
```

- 2.5. 使用 Skopeo 将 OCI 镜像复制到课堂集群的内部注册表并将其标记为 **1.0**。使用您在前面的步骤中检索的主机名和令牌。

您可以从 **/home/student/D0288/labs/expose-registry** 文件夹中的 **push-image.sh** 脚本剪切和粘贴以下 **skopeo copy** 命令。

### 章3 | 发布企业容器镜像

```
[student@workstation ~]$ skopeo copy --format v2s1 \
--dest-creds=${RHT_OCP4_DEV_USER}:${TOKEN} \
oci:/home/student/D0288/labs/expose-registry/ubi-info \
docker://${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

- 2.6. 验证是否已创建镜像流来管理新的容器镜像。为了便于阅读，对 `oc get is` 命令的输出进行了编辑，以在一行上显示每一列，因为镜像存储库名称应该太长，无法适应纸张宽度。

```
[student@workstation ~]$ oc get is
NAME      IMAGE REPOSITORY
ubi-info  default-route-openshift-image-registry.apps...
...output omitted...
```

#### ▶ 3. 从 OpenShift 内部注册表中的镜像创建本地容器。

- 3.1. 将 `ubi-info:1.0` 容器镜像下载到本地容器引擎中。

```
[student@workstation ~]$ podman pull \
${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
...output omitted...
```

- 3.2. 从 `ubi-info:1.0` 容器镜像启动新容器。容器显示系统信息，如主机名和可用内存，然后退出。以下输出中省略了特定于正在运行的容器的信息：

```
[student@workstation ~]$ podman run --name info \
${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0
...output omitted...
--- Host name:
...output omitted...
--- Free memory
...output omitted...
--- Mounted file systems (partial)
...output omitted...
```

#### ▶ 4. 清理。删除在此练习中创建的所有资源。

- 4.1. 通过删除镜像流，从课堂集群的内部注册表中删除容器镜像：

```
[student@workstation ~]$ oc delete is ubi-info
imagestream.image.openshift.io "ubi-info" deleted
```

- 4.2. 删除 OpenShift 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common  
project.project.openshift.io "youruser-common" deleted
```

4.3. 从本地容器引擎中删除测试容器和镜像：

```
[student@workstation ~]$ podman rm info  
...output omitted...  
[student@workstation ~]$ podman rmi -f \  
${INTERNAL_REGISTRY}/${RHT_OCP4_DEV_USER}-common/ubi-info:1.0  
...output omitted...
```

## 完成

在 workstation 虚拟机上，运行 `lab expose-registry finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab expose-registry finish
```

本引导式练习到此结束。

# 创建镜像流

## 培训目标

学完本节后，您应能够为外部注册表中的容器镜像创建镜像流。

## 描述镜像流

镜像流是 OpenShift 和上游 Kubernetes 之间的主要区别之一。Kubernetes 资源直接引用容器镜像，但 OpenShift 资源（如部署配置和构建配置）引用镜像流。OpenShift 还扩展了 Kubernetes 资源（如 StatefulSet 和 CronJob 资源）以及使它们与 OpenShift 镜像流搭配使用的注释。尽管也可以结合使用镜像流和 Kubernetes 部署，但通常无此必要，而且也不在本课程的讨论范畴。

镜像流允许 OpenShift 确保可重复、稳定地部署容器化应用，并将部署回滚到其最新的已知良好状态。

镜像流提供了一个稳定的短名称来引用独立于任何注册表服务器和容器运行时配置的容器镜像。

例如，组织可以先直接从红帽公共注册表下载容器镜像，然后设置企业注册表作为这些镜像的镜像，以节省带宽。OpenShift 用户不会注意到任何更改，因为他们仍然使用相同的镜像流名称引用这些镜像。RHEL 容器工具的用户会注意到更改，因为他们需要更改其命令中的注册表名称或更改其容器引擎配置以首先搜索本地镜像。

在其他情况下，镜像流提供的间接层被证明是有帮助的。假设您从具有安全问题的数据库容器镜像开始，并且供应商使用修补程序更新镜像花费的时间过长。后来，您找到一个为同一数据库提供替代容器镜像的替代供应商，这些安全问题已经修复，甚至具有及时提供其更新的跟踪记录。如果这些容器镜像在环境变量和卷的配置方面兼容，则只需更改镜像流以指向替代供应商的镜像即可。

红帽提供经过强化的受支持的容器镜像，这些镜像主要用于直接替换某些常见开源项目（如 MariaDB 数据库）中的容器镜像。

## 描述镜像流标签

镜像流表示一组或多组容器镜像。每个组或流都由镜像流标签标识。注册表服务器中的容器镜像具有来自同一镜像存储库（或用户或组织）的多个标签，与之不同的是，镜像流可以具有多个镜像流标签，这些标签引用来自不同注册表服务器和不同镜像存储库的容器镜像。

镜像流为一组镜像流标签提供默认配置。每个镜像流标签引用一个容器镜像流，并可以覆盖其关联镜像流中的大多数配置。

镜像流标签存储有关其当前容器镜像的元数据的副本，并可以选择存储其当前和过去的容器镜像层的副本。存储元数据可以更快地搜索和检查容器镜像，因为您不需要访问其源注册表服务器。

存储镜像层允许镜像流标签充当本地镜像缓存，从而避免需要从其源注册表服务器获取这些层。镜像流标签将其缓存的镜像层存储在 OpenShift 内部注册表中。缓存镜像（如容器集和部署配置）的使用者只需将内部注册表作为镜像的源注册表来引用。

还有其他一些与镜像流相关的 OpenShift 资源类型，但开发人员通常可以将它们作为内部注册表的实施详细信息加以忽略，只关注镜像流和镜像流标签。

为了更好地可视化镜像流和镜像流标签之间的关系，可以浏览在所有 OpenShift 集群中预先创建的 **openshift** 项目。您可以看到该项目中有许多镜像流，包括 **php** 镜像流：

```
[user@host ~]$ oc get is -n openshift -o name
...output omitted...
imagestream.image.openshift.io/nodejs
imagestream.image.openshift.io/perl
imagestream.image.openshift.io/php
imagestream.image.openshift.io/postgresql
imagestream.image.openshift.io/python
...output omitted...
```

php 镜像流存在许多标签，并且每个标签都存在镜像流资源：

```
[user@host ~]$ oc get istag -n openshift | grep php
php:7.2      image-registry ...    6 days ago
php:7.3      image-registry ...    6 days ago
php:latest   image-registry ...    6 days ago
```

如果在镜像流上使用 `oc describe` 命令，它将显示来自镜像流及其镜像流标签的信息：

```
[user@host ~]$ oc describe is php -n openshift
Name:                  php
Namespace:             openshift
...output omitted...
Tags:                  3

7.3 (latest)
tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
...output omitted...

7.2
tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
...output omitted...
```

在前面的示例中，每个 `php` 镜像流标签都引用不同的镜像名称。

## 描述镜像名称、标签和 ID

容器镜像的文本名称只是一个字符串。此名称有时被解释为由多个组件组成，例如 `registry-host-name/repository-or-organization-or-user-name/image-name:tag-name`，但将镜像名称拆分为其组件只是一个约定问题，不是结构问题。

镜像 ID 使用 SHA-256 哈希来唯一标识不可变容器镜像。请记住，您无法修改容器镜像。取而代之，您可以创建一个具有新 ID 的新容器镜像。将新容器镜像推送到注册表服务器时，服务器会将现有文本名称与新镜像 ID 关联。

从镜像名称启动容器时，将下载当前与该镜像名称关联的镜像。该名称背后的实际镜像 ID 可能随时更改，您启动的下一个容器可能具有不同的镜像 ID。如果与镜像名称关联的镜像有任何问题，并且您只知道镜像名称，则不能回滚到更早的镜像。

OpenShift 镜像流标签保留它们从注册表服务器获取的最新镜像 ID 的历史记录。镜像 ID 的历史记录是来自镜像流标签的镜像流。例如，如果新容器镜像导致部署错误，则可以使用镜像流标签中的历史记录回滚到上一个镜像。

### 章 3 | 发布企业容器镜像

在外部注册表中更新容器镜像不会自动更新镜像流标签。镜像流标签保留对它获取的最后一个镜像 ID 的引用。此行为对于扩展应用至关重要，因为它将 OpenShift 与注册表服务器上发生的更改隔离开来。

假设您从外部注册表部署应用，在使用几个用户进行几天测试后，您决定扩展其部署以启用更大的用户群。同时，您的供应商更新外部注册表上的容器镜像。如果 OpenShift 没有镜像流标签，则新容器集将获得新的容器镜像，这与原始容器集上的镜像不同。根据更改，这可能会导致应用失败。由于 OpenShift 将原始镜像的镜像 ID 存储在镜像流标签中，因此它可以使用相同的镜像 ID 创建新容器集，并避免原始镜像和更新镜像之间的任何不兼容。

OpenShift 保留用于第一个容器集的镜像 ID，并确保新容器集使用相同的镜像 ID。OpenShift 确保所有容器集使用完全相同的镜像。

为了更好地显示镜像流、镜像流标签、镜像名称和镜像 ID 之间的关系，请参阅以下 `oc describe is` 命令，该命令显示每个镜像流标签的源镜像和当前镜像 ID：

```
[user@host ~]$ oc describe is php -n openshift
Name:          php
Namespace:     openshift
...output omitted...
7.3 (latest)
  tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
  ...output omitted...
  * registry.redhat.io/rhscl/php-73-rhel7@sha256:22ba...09b5
  ...output omitted...
7.2
  tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
  ...output omitted...
  * registry.redhat.io/rhscl/php-72-rhel7@sha256:e8d6...e615
  ...output omitted...
```

如果 OpenShift 集群管理员已更新 `php:7.3` 镜像流标签，则 `oc describe is` 命令将显示该标签的多个镜像 ID：

```
[user@host ~]$ oc describe is php -n openshift
Name:          php
Namespace:     openshift
...output omitted...
7.3 (latest)
  tagged from registry.redhat.io/rhscl/php-73-rhel7:latest
  ...output omitted...
  * registry.redhat.io/rhscl/php-73-rhel7@sha256:22ba...09b5
  ...output omitted...
  registry.redhat.io/rhscl/php-73-rhel7@sha256:bc61...1e91
  ...output omitted...
7.2
  tagged from registry.redhat.io/rhscl/php-72-rhel7:latest
  ...output omitted...
  * registry.redhat.io/rhscl/php-72-rhel7@sha256:e8d6...e615
  ...output omitted...
```

在前面的示例中，星号 (\*) 显示每个镜像流标签的当前镜像 ID。它通常是最后导入的，是第一个列出的。

### 章 3 | 发布企业容器镜像

当 OpenShift 镜像流标签引用外部注册表中的容器镜像时，需要显式更新镜像流标签，以便从外部注册表获取新的镜像 ID。默认情况下，OpenShift 不会监控外部注册表中与镜像名称关联的镜像 ID 的更改。

您可以配置镜像流标签，以按定义的计划检查外部注册表中的更新。默认情况下，新的镜像流标签不检查更新的镜像。

构建配置会自动更新用作输出镜像的镜像流标签。这将强制使用该镜像流标签重新部署应用容器集。

## 管理镜像流和标签

若要为外部注册表上托管的容器镜像创建镜像流标签资源，请使用 `oc import-image` 命令以及 `--confirm` 和 `--from` 选项。以下命令更新镜像流标签；或者，如果不存在镜像流标签，则命令会创建一个：

```
[user@host ~]$ oc import-image myimagestream[:tag] --confirm \
--from registry/myorg/myimage[:tag]
```

如果不指定标签名称，则默认使用 `latest` 标签。在本例中，镜像流标签引用 `myimagestream` 镜像流。如果相应的镜像流尚不存在，OpenShift 将创建它。



### 注意

要为托管在未使用受信任的 TLS 证书设置的注册表服务器上的容器镜像创建镜像流，请向 `oc import-image` 命令添加 `--insecure` 选项。

镜像流标签的标签名称可以不同于源注册表服务器上的容器镜像标签。下面的示例从源注册表服务器 `latest` 标签创建 `1.0` 镜像流标签（通过省略）：

```
[user@host ~]$ oc import-image myimagestream:1.0 --confirm \
--from registry/myorg/myimage
```

要为源注册表服务器中存在的每个容器镜像标签创建一个镜像流标签资源，请向 `oc import-image` 命令添加 `--all` 选项：

```
[user@host ~]$ oc import-image myimagestream --confirm --all \
--from registry/myorg/myimage
```

您可以在现有镜像流上运行 `oc import-image` 命令，以将其当前镜像流标签之一更新为源注册表服务器上的当前镜像 ID。

```
[user@host ~]$ oc import-image myimagestream[:tag] --confirm
```

使用 `--all` 选项更新镜像流将更新所有镜像流标签，并为它在源注册表服务器上找到的新标签创建新的镜像流标签。

您可以使用 `oc tag` 命令对镜像流标签施加更精细的控制。它允许更改，例如：

- 将镜像流标签关联到与其镜像流所关联的服务器不同的注册表服务器。
- 将镜像流标记关联到其他容器镜像名称和标签。

### 章 3 | 发布企业容器镜像

- 将镜像流标签关联到给定的镜像 ID，该 ID 可能不是当前与注册表服务器上的该镜像标签关联的镜像 ID。
- 将镜像流与其他镜像流标签相关联。例如，前面的 `oc describe is` 命令示例显示 `php:latest` 镜像流标签跟在 `php:7.3` 镜像流标签后面。这是一种为镜像流标签创建别名的方法。

教授镜像流管理的所有方面超出了本课程的范围，但在后面的章节中探讨了其中一些方面（如镜像改事件）。

## 将镜像流与私有注册表一起使用

要创建引用私有注册表的镜像流和镜像流标签，OpenShift 需要该注册表服务器的访问令牌。

您将访问令牌作为机密提供，与从私有注册表部署应用的方式相同，并且不需要将该机密链接到任何服务帐户。`oc import-image` 命令在当前项目的机密中搜索与注册表主机名匹配的一项。

下面的示例使用 Podman 登录到私有注册表，创建用于存储访问令牌的机密，然后创建指向私有注册表的镜像流：

```
[user@host ~]$ podman login -u myuser registry.example.com
[user@host ~]$ oc create secret generic regtoken \
--from-file dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
[user@host ~]$ oc import-image myis --confirm \
--from registry.example.com/myorg/myimage
```

创建镜像流后，您可以使用它来通过 `oc new-app -i myis` 命令部署应用。您还可以在 `oc new-app myis~giturl` 命令中使用该镜像流作为构建器镜像。

默认情况下，镜像流资源只能用于在同一项目中创建应用或构建。

## 在多个项目之间共享镜像流

在 OpenShift 中创建项目以存储资源并在多个用户和开发团队之间共享是一种常见做法。这些共享项目存储镜像流和模板等资源，开发人员在将应用部署到其项目时会引用这些资源。

OpenShift 附带一个名为 `openshift` 的共享项目，它为常见编程语言（如 Python 和 Ruby）的 S2I 构建器提供快速启动的应用模板和镜像流。某些组织为其团队创建类似的项目，而不是向 `openshift` 项目添加资源，以避免在将集群升级到新版本的 OpenShift 时出现问题。



### 重要

您可以选择在早期版本中向 `openshift` 项目添加新资源，但自红帽 OpenShift 容器平台 4 起，示例操作程序负责管理 `openshift` 项目，并可能随时删除手动添加到其中的资源。

要使用在另一个项目中定义的镜像流构建和部署应用，有两个选项：

- 在使用镜像流的每个项目上，使用对私有注册表的访问令牌创建机密，并将该机密链接到每个项目的服务帐户。
- 仅在创建镜像流的项目上，使用对私有注册表的访问令牌创建机密，并使用本地引用策略配置该镜像流。向服务帐户授予从使用镜像流的每个项目中使用镜像流的权限。

### 章 3 | 发布企业容器镜像

第一个选项类似于在私有注册表中从容器镜像部署应用时要执行的操作。它抵消了使用镜像流的一些好处，因为如果您引用的镜像流标签改为引用另一个注册表服务器，则需要在所有项目中为新注册表服务器创建一个新机密。

第二个选项允许引用镜像流的项目与它们使用的镜像流标签中的更改保持隔离。为服务帐户分配权限的额外工作归因于服务帐户具有的权限比创建它们的用户帐户更受限制。

下面的示例演示第二个选项。它在 **shared** 项目中创建镜像流，并使用该镜像流在 **myapp** 项目中部署应用。

```
[user@host ~]$ podman login -u myuser registry.example.com
[user@host ~]$ oc project shared
[user@host ~]$ oc create secret generic regtoken \
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
[user@host ~]$ oc import-image myis --confirm \
--reference-policy local \ ①
--from registry.example.com/myorg/myimage
[user@host ~]$ oc policy add-role-to-group system:image-puller \ ②
system:serviceaccounts:myapp ③
[user@host ~]$ oc project myapp
[user@host ~]$ oc new-app -i shared/myis
```

- ① `oc import-image` 命令的 `--reference-policy local` 选项。它将镜像流配置为在内部注册表中缓存镜像层，因此引用镜像流的项目不需要外部私有注册表的访问令牌。
- ② `system:image-puller` 角色允许服务帐户拉取镜像流缓存在内部注册表中的镜像层。
- ③ `system:serviceaccounts:myapp` 组。此组包括 `myapp` 项目中的所有服务帐户。`oc policy` 命令可以引用尚不存在的用户和组。

`oc policy` 命令不需要集群管理员权限；它只需要项目管理员权限。

上一个示例中的镜像流还可以为 `oc new-app shared/myis~giturl` 命令提供构建器镜像。



#### 参考文献

如需关于镜像流、镜像流标签和镜像 ID 的更多信息，请参阅红帽 OpenShift 容器平台 4.10 Images 指南的 Understanding Containers, Images, and Imagestreams 一章，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/index#understanding-images](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/index#understanding-images)

## ▶ 指导练习

# 创建镜像流

在本练习中，您将使用镜像流部署基于 Nginx 的“hello, world”应用。

## 成果

您应能够：

- 在 OpenShift 中将外部注册表中的镜像作为镜像流发布。
- 使用镜像流部署应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- “你好，世界”应用容器镜像 (`redhattraining/hello-world-nginx`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件：

```
[student@workstation ~]$ lab image-stream start
```

## 说明

▶ 1. 登录到 OpenShift 并创建一个项目来托管 Nginx 容器镜像的镜像流。

1.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户帐户登录 OpenShift。

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

1.3. 创建一个项目来托管可能在多个项目之间共享的镜像流：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

▶ 2. 创建指向来自外部注册表的 Nginx 镜像的镜像流。

## 章3 | 发布企业容器镜像

- 2.1. 验证来自 Quay.io 的 `redhattraining/hello-world-nginx` 镜像具有名为 `latest` 的单个标签。

```
[student@workstation ~]$ skopeo inspect \
docker://quay.io/redhattraining/hello-world-nginx
{
  "Name": "quay.io/redhattraining/hello-world-nginx",
  "Tag": "latest",
  "Digest": "sha256:4f4f...acc1",
  "RepoTags": [
    "latest"
  ],
  ...output omitted...
```

- 2.2. 创建指向 Quay.io 中 `redhattraining/hello-world-nginx` 容器镜像的 `hello-world` 镜像流。

```
[student@workstation ~]$ oc import-image hello-world --confirm \
--from quay.io/redhattraining/hello-world-nginx
imagestream.image.openshift.io/hello-world imported
...output omitted...
Name:          hello-world
Namespace:     youruser-common
...output omitted...
Unique Images: 1
Tags:          1

latest
tagged from quay.io/redhattraining/hello-world-nginx
...output omitted...
```

- 2.3. 验证 `hello-world:latest` 镜像流标签是否已创建:

```
[student@workstation ~]$ oc get istag
NAME           IMAGE REF
hello-world:latest  quay.io/redhattraining/hello-world-nginx@sha256:4f4f...acc1 ...
```

- 2.4. 验证镜像流及其标签是否包含有关 Nginx 容器镜像的元数据:

```
[student@workstation ~]$ oc describe is hello-world
Name:          hello-world
Namespace:     youruser-common
...output omitted...
Tags:          1

latest
tagged from quay.io/redhattraining/hello-world-nginx ①
* quay.io/redhattraining/hello-world-nginx@sha256:4f4f...acc1 ②
```

## 章3 | 发布企业容器镜像

```
2 minutes ago
```

```
...output omitted...
```

- ❶ 镜像流标签 `hello-world:latest` 引用来自 Quay.io 的镜像。
- ❷ 星号 (\*) 表示 `latest` 镜像流标签仅引用具有给定 SHA-256 标识符的特定镜像。

► 3. 使用来自 `youruser-common` 项目的 `hello-world` 镜像流创建新项目和部署应用。

3.1. 创建一个项目来托管测试应用：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-image-stream
Now using project "youruser-image-stream" on server
"https://api.cluster.domain.example.com:6443".
```

3.2. 从镜像流部署应用：

```
[student@workstation ~]$ oc new-app --name hello \
-i ${RHT_OCP4_DEV_USER}-common/hello-world
--> Found image 44eaa13 (20 hours old) in image stream "youruser-common/hello-
world" under tag "latest" for "youruser-common/hello-world"
...output omitted...
--> Creating resources ...
imagestreamtag.image.openshift.io "hello:latest" created
deployment.apps "hello" created
service "hello" created
--> Success
...output omitted...
```

3.3. 等待应用容器集准备就绪并在运行：

```
[student@workstation ~]$ oc get pod
NAME           READY   STATUS    RESTARTS   AGE
hello-6599bb7b9c-zk58m   1/1     Running   0          40s
```

3.4. 创建公开应用的路由：

```
[student@workstation ~]$ oc expose svc hello
route.route.openshift.io/hello exposed
```

3.5. 获取路由的主机名：

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
hello    hello-youruser-image-stream.apps.cluster.domain.example.com ...
```

3.6. 使用 `curl` 命令以及上一步中的主机名测试应用。

```
[student@workstation ~]$ curl \
http://hello-${RHT_OCP4_DEV_USER}-image-stream.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
<h1>Hello, world from nginx!</h1>
...output omitted...
```

▶ 4. 删除 youruser-commons 和 youruser-image-stream 项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-image-stream
project.project.openshift.io "youruser-image-stream" deleted
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common
project.project.openshift.io "youruser-image-common" deleted
```

## 完成

在 workstation 虚拟机上，运行 `lab image-stream finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab image-stream finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 发布企业容器镜像

在本实验中，您将使用镜像流向外部注册表发布 OCI 格式的容器镜像，并从该镜像部署应用。



### 注意

各个章节实验结尾处使用的 `grade` 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应该能够从存储在外部注册表中的镜像创建应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 示例容器镜像的 OCI 兼容文件 (`php-info`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab expose-image start
```

## 要求

应用镜像包含显示 Web 服务器环境和 PHP 解释器配置的 PHP 应用。按照如下所示部署应用：

- 在名为 `youruser-common` 的项目中托管在 OpenShift 外部构建的镜像的镜像流。

容器镜像名称是 `php-info`。OCI 格式的镜像层和清单位于 `/home/student/D0288/labs/expose-image/php-info` 文件夹中。将该镜像推送到您的 Quay.io 帐户中的私有镜像存储库。

- 将应用部署到名为 `youruser-expose-image` 的项目中。

应用的资源称为 `info`。使用 OpenShift 分配的默认主机名访问应用。

- 使用 `/usr/local/etc/ocp4.config` 配置文件获取教室配置数据，如 OpenShift 集群的主 API URL。

## 说明

- 将 `php-info` OCI 格式的容器镜像推送到 Quay.io。
- 作为开发人员用户，创建 `youruser-common` 项目以托管指向外部注册表中镜像的镜像流。使用 Quay.io 的登录凭据创建机密并创建 `php-info` 镜像流。

### 章 3 | 发布企业容器镜像

使用 `--reference-policy local` 选项创建镜像流，以便使用该镜像流的其他项目也可以使用存储在 `youruser-common` 项目中的机密。

3. 新建一个名为 `youruser-expose-image` 的项目。然后通过从镜像流部署容器镜像来创建新应用。应用名称使用 `info`。  
为 `youruser-expose-image` 项目的所有服务帐户授予 `youruser-common` 项目的 `system:image-puller` 角色。此角色允许一个项目中的容器集使用另一个项目中的镜像流。文件夹 `/home/student/D0288/labs/expose-image` 中的 `grant-puller-role.sh` 脚本包含执行此操作的 `oc policy` 命令。
4. 公开并测试应用。验证应用是否返回 PHP 解释器配置页面。
5. 对您的作业进行评分。

在 `workstation` 虚拟机上运行以下命令，以验证是否已完成所有任务：

```
[student@workstation ~]$ lab expose-image grade
```

6. 删除 OpenShift 项目并从 Quay.io 中删除镜像存储库。

## 完成

在 `workstation` 虚拟机上，运行 `lab expose-image finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab expose-image finish
```

本实验到此结束。

## ► 解决方案

# 发布企业容器镜像

在本实验中，您将使用镜像流向外部注册表发布 OCI 格式的容器镜像，并从该镜像部署应用。



### 注意

各个章节实验结尾处使用的 `grade` 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应该能够从存储在外部注册表中的镜像创建应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 示例容器镜像的 OCI 兼容文件 (`php-info`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载解决方案文件：

```
[student@workstation ~]$ lab expose-image start
```

## 要求

应用镜像包含显示 Web 服务器环境和 PHP 解释器配置的 PHP 应用。按照如下所示部署应用：

- 在名为 `youruser-common` 的项目中托管在 OpenShift 外部构建的镜像的镜像流。

容器镜像名称是 `php-info`。OCI 格式的镜像层和清单位于 `/home/student/D0288/labs/expose-image/php-info` 文件夹中。将该镜像推送到您的 Quay.io 帐户中的私有镜像存储库。

- 将应用部署到名为 `youruser-expose-image` 的项目中。

应用的资源称为 `info`。使用 OpenShift 分配的默认主机名访问应用。

- 使用 `/usr/local/etc/ocp4.config` 配置文件获取教室配置数据，如 OpenShift 集群的主 API URL。

## 说明

### 1. 将 `php-info` OCI 格式的容器镜像推送到 Quay.io。

- 1.1. 验证 `php-info` 文件夹是否包含 OCI 格式的容器镜像：

```
[student@workstation ~]$ ls ~/D0288/labs/expose-image/php-info  
blobs index.json oci-layout
```

- 1.2. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的配置变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.3. 使用 **podman** 命令来登录 Quay.io

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io  
Password:  
Login Succeeded!
```

- 1.4. 使用 **skopeo** 命令将 OCI 格式的容器镜像推送到 Quay.io 您可以从 **/home/student/D0288/labs/expose-image** 文件夹中的 **push-image.sh** 脚本复制或运行 **skopeo** 命令：

```
[student@workstation ~]$ skopeo copy --format v2s1 \  
oci:/home/student/D0288/labs/expose-image/php-info \  
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info  
...output omitted...  
Writing manifest to image destination  
Storing signatures
```

- 1.5. 使用 Skopeo 检查外部注册表中的镜像以验证它是否标记为 **latest**。

```
[student@workstation ~]$ skopeo inspect \  
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info  
{  
  "Name": "quay.io/yourquayuser/php-info",  
  "Tag": "latest",  
  ...output omitted...
```

2. 作为开发人员用户，创建 **youruser-common** 项目以托管指向外部注册表中镜像的镜像流。使用 Quay.io 的登录凭据创建机密并创建 **php-info** 镜像流。  
使用 **--reference-policy local** 选项创建镜像流，以便使用该镜像流的其他项目也可以使用存储在 **youruser-common** 项目中的机密。

- 2.1. 使用您的开发人员用户帐户登录 OpenShift：

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \  
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}  
Login successful  
...output omitted...
```

- 2.2. 创建托管镜像流的项目。

### 章 3 | 发布企业容器镜像

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-common
Now using project "youruser-common" on server
"https://api.cluster.domain.example.com:6443".
```

2.3. 从 Podman 存储的容器注册表 API 访问令牌创建机密。

您可以从 /home/student/D0288/labs/expose-image 文件夹中的 `create-secret.sh` 脚本复制或运行 `oc create secret` 命令：

```
[student@workstation ~]$ oc create secret generic quayio \
--from-file dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
secret/quayio created
```

2.4. 使用 `oc import-image` 命令导入新的容器镜像：

```
[student@workstation ~]$ oc import-image php-info --confirm \
--reference-policy local \
--from quay.io/${RHT_OCP4_QUAY_USER}/php-info
imagestream.image.openshift.io/php-info imported
...output omitted...
latest
tagged from quay.io/youruser/php-info
will use insecure HTTPS or HTTP connections

* quay.io/youruser/php-info@sha256:4366...f937
  Less than a second ago
...output omitted...
```

2.5. 验证镜像流标签是否已创建并包含有关 `php-info` 容器镜像的元数据：

```
[student@workstation ~]$ oc get istag
NAME          IMAGE REF   ...
php-info:latest  image-registry.openshift-image-registry.svc:5000/youruser-
common/php-info@sha256:4366...f937 ...
```

3. 新建一个名为 `youruser-expose-image` 的项目。然后通过从镜像流部署容器镜像来创建新应用。应用名称使用 `info`。

为 `youruser-expose-image` 项目的所有服务帐户授予 `youruser-common` 项目的 `system:image-puller` 角色。此角色允许一个项目中的容器集使用另一个项目中的镜像流。文件夹 /home/student/D0288/labs/expose-image 中的 `grant-puller-role.sh` 脚本包含执行此操作的 `oc policy` 命令。

3.1. 创建一个项目来托管应用：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-expose-image
```

3.2. 为新 `youruser-expose-image` 项目中的服务帐户授予 `youruser-common` 项目中镜像流的访问权限。您可以从 /home/student/D0288/labs/expose-image 文件夹中的 `grant-puller-role.sh` 脚本复制或运行以下 `oc policy` 命令。

## 章3 | 发布企业容器镜像

```
[student@workstation ~]$ oc policy add-role-to-group \
-n ${RHT_OCP4_DEV_USER}-common system:image-puller \
system:serviceaccounts:${RHT_OCP4_DEV_USER}-expose-image
clusterrole.rbac.authorization.k8s.io/system:image-puller added:
"system:serviceaccounts:youruser-expose-image"
```

3.3. 从 common 项目中的镜像流部署测试应用：

```
[student@workstation ~]$ oc new-app --name info \
-i ${RHT_OCP4_DEV_USER}-common/php-info
...output omitted...
--> Creating resources ...
imagestreamtag.image.openshift.io "info:latest" created
deployment.apps "info" created
service "info" created
--> Success
...output omitted...
```

3.4. 等待应用容器集准备就绪并在运行：

```
[student@workstation ~]$ oc get pod
NAME           READY   STATUS    RESTARTS   AGE
info-5c687bc4bc-j4sdz   1/1     Running   0          26s
```

4. 公开并测试应用。验证应用是否返回 PHP 解释器配置页面。

4.1. 公开 info 应用。

```
[student@workstation ~]$ oc expose svc info
route.route.openshift.io/info exposed
```

4.2. 获取 OpenShift 分配给路由的主机名：

```
[student@workstation ~]$ oc get route info
NAME      HOST/PORT
info      info-youruser-expose-image.apps.cluster.domain.example.com
```

4.3. 使用 curl 命令以及上一步中的路由测试应用。或者，您可以使用 Web 浏览器。

```
[student@workstation ~]$ curl \
http://info-${RHT_OCP4_DEV_USER}-expose-image.${RHT_OCP4_WILDCARD_DOMAIN}
...output omitted...
<title>phpinfo()</title><meta name="ROBOTS" content="NOINDEX, NOFOLLOW, NOARCHIVE" />
</head>
...output omitted...
<tr><td class="e">Server API </td><td class="v">FPM/FastCGI </td></tr>
...output omitted...
<tr><td class="e">PHP API </td><td class="v">20170718 </td></tr>
...output omitted...
```

5. 对您的作业进行评分。

### 章3 | 发布企业容器镜像

在 workstation 虚拟机上运行以下命令，以验证是否已完成所有任务：

```
[student@workstation ~]$ lab expose-image grade
```

6. 删除 OpenShift 项目并从 Quay.io 中删除镜像存储库。

6.1. 删除 youruser-expose-image 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-expose-image  
project.project.openshift.io "youruser-expose-image" deleted
```

6.2. 删除 youruser-common 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-common  
project.project.openshift.io "youruser-common" deleted
```

6.3. 从外部注册表中删除容器镜像：

```
[student@workstation ~]$ skopeo delete \  
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-info:latest
```

6.4. 使用您的个人免费帐户登录 Quay.io。

导航到 <http://quay.io> 并单击 **Sign In** 以提供用户凭据。登录 Quay.io。

6.5. 在 Quay.io 主菜单上，单击 **Repositories** 并查找 **php-info**。单击 **php-info** 可打开 **Repository Activity** 页面。

6.6. 在 **php-info** 存储库的 **Repository Activity** 页面，向下滚动并单击齿轮图标以显示 **Settings** 选项卡。向下滚动并单击 **Delete Repository**。

6.7. 在 **Delete** 对话框中，输入存储库名称，单击 **Delete** 以确认要删除 **php-info** 存储库。几分钟后，您将返回到 **Repositories** 页面。您现在可以注销 Quay.io。

## 完成

在 workstation 虚拟机上，运行 **lab expose-image finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab expose-image finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- OpenShift 开发人员与多种可能需要或不需要身份验证的注册表服务器（包括 OpenShift 内部注册表）进行交互。
- OpenShift 要求开发人员创建机密，将访问令牌存储到私有的外部注册表。
- 就像任何其他注册表服务器一样，Linux 容器工具（Skopeo、Podman 和 Buildah）可以使用 OpenShift 用户访问令牌访问 OpenShift 内部注册表，作为安全或不安全的注册表。
- OpenShift 镜像流和镜像流标签资源提供对容器镜像的稳定引用，并将开发人员与注册表服务器地址隔离开来。



## 章 4

# 在 OpenShift 上管理构建

### 目标

介绍 OpenShift 构建流程以及如何触发和管理构建。

### 培训目标

- 介绍 OpenShift 构建流程。
- 使用 BuildConfig 资源和 CLI 命令来管理应用的构建。
- 使用受支持的方法触发构建流程。
- 使用 post-commit 构建 hook 处理构建后逻辑。

### 章节

- 介绍 OpenShift 构建流程（及测验）
- 管理应用构建（及引导式练习）
- 触发构建（及引导式练习）
- 实施 Post-commit 构建 hook（及引导式练习）

### 实验

在 OpenShift 上管理构建

# 介绍红帽 OpenShift 构建流程

## 培训目标

学完本节后，您应能够介绍红帽 OpenShift 构建流程。

## 构建流程

红帽 OpenShift 容器平台构建过程将源代码或二进制文件和其他输入参数转换为容器镜像，以供在平台上部署。要构建容器镜像，红帽 OpenShift 需要 **BuildConfig** 资源。对于这类资源，需要配置一个构建策略以及一个或多个输入源，如 git、二进制或内联定义。

## 构建策略

以下是红帽 OpenShift 中可供使用的构建策略：

- 源至镜像 (S2I) 构建
- Docker 构建
- 自定义构建

每种策略都需要使用相应的容器镜像。

### 源至镜像 (S2I) 构建

**source-to-image** 策略会基于应用源代码或应用二进制文件创建新的容器镜像。OpenShift 会克隆应用源代码，或将应用二进制文件复制到兼容构建器镜像中，然后通过汇编方式构建一个可以即刻在该平台中进行部署的新容器镜像。

该策略可以简化开发人员构建容器镜像的方式，因为该策略会使用与容器镜像类似的工具，而不是使用低级别的 OS 命令，如 **Containerfiles** 中的 **yum**。

红帽 OpenShift 的 **source-to-image** 策略基于源至镜像 (S2I) 流程。

### Docker 构建

**docker build** 策略会基于 **Containerfile** 文件使用 **buildah** 命令构建新的容器镜像。Docker 策略可以检索 **Containerfile** 和项目，以便从 Git 存储库构建容器镜像，也可以使用构建配置内联提供的 **Containerfile** 作为构建源。

Docker 构建会作为 OpenShift 集群内部的一个容器集来运行。开发人员不需要在其工作站上具有 Docker 工具。



#### 注意

Docker 构建需要使用提升的特权，而且红帽 OpenShift 集群管理员可能会拒绝向部分或所有用户授予 Docker 构建的启动权限。

## 自定义构建

`custom build` 策略会指定负责实施构建流程的构建器镜像。此策略允许开发人员自定义构建流程。请参见本单元中的“参考资料”部分，以查找有关如何创建自定义构建器镜像的更多信息。

## 构建输入源

构建输入源可为构建提供源内容。红帽 OpenShift 支持以下六种类型的输入源，按优先级顺序列出：

- **Containerfile**: 指定用于构建镜像的内联 Containerfile。
- **Image**: 从镜像构建时可为构建流程提供其他文件。
- **Git**: OpenShift 从 Git 存储库克隆输入应用源代码。可以将该存储库中的某个位置配置为默认位置，以便在构建时从中查找应用源代码。
- **Binary**: 允许将本地文件系统中的二进制内容流化至构建器。
- **Input secrets**: 您可以使用输入机密以允许为构建创建在最终应用镜像中不可用的凭据。
- **External artifacts**: 允许将二进制文件复制到构建流程中。

您可以在单个构建中组合多个输入。但是，由于内联 Containerfile 文件优先，它可覆盖由另一个输入提供的任何其他 Containerfile 文件。此外，二进制输入和 Git 存储库是互斥的输入。



### 注意

虽然红帽 OpenShift 提供了多种策略和输入源，但最常见的场景是使用 **Source** 或 **Docker** 策略并将 Git 存储库作为输入源。

## BuildConfig 资源

构建配置可以定义构建流程的实施方式。**BuildConfig** 资源可以定义一个构建配置和一组触发器，以便在红帽 OpenShift 必须创建全新构建时使用。

OpenShift 会使用 `oc new-app` 命令生成 **BuildConfig**。它还可使用 Web 控制台中的 **Add to Project** 按钮或通过从模板创建应用来生成。

以下示例使用 **Source** 策略和 **Git** 输入源构建了一个 PHP 应用：

```
{
  "kind": "BuildConfig",
  "apiVersion": "v1",
  "metadata": {
    "name": "php-example", ①
    ...output omitted...
  },
  "spec": {
    "triggers": [ ②
      {
        "type": "GitHub",
        "github": {
          "secret": "gukAWHzq1On4AJlMjvjS" ③
        }
      },
    ],
    "strategy": {
      "type": "Source"
    }
  }
}
```

```

    ...output omitted...
],
  "runPolicy": "Serial", ④
  "source": { ⑤
    "type": "Git",
    "git": {
      "uri": "http://services.lab.example.com/php-helloworld"
    }
  },
  "strategy": { ⑥
    "type": "Source",
    "sourceStrategy": {
      "from": {
        "kind": "ImageStreamTag",
        "namespace": "openshift",
        "name": "php:7.0"
      }
    }
  },
  "output": { ⑦
    "to": {
      "kind": "ImageStreamTag",
      "name": "php-example:latest"
    }
  },
  ...output omitted...
},
...output omitted...
}

```

- ①** 定义名为 `php-example` 的新 `BuildConfig`。
- ②** 定义可以启动新构建的触发器。
- ③** 用于 webhook 的授权字符串，由红帽 OpenShift 生成。外部应用会将该字符串作为 webhook URL 的一部分进行发送，以触发新构建。
- ④** `runPolicy` 属性可定义能否同时启动构建。值 `Serial` 表示不能同时进行构建。
- ⑤** `source` 属性负责定义构建的输入源。在本示例中，所用的输入源为 Git 存储库。
- ⑥** 定义用于构建最终容器镜像的构建策略。在本示例中，使用的是 `Source` 策略。
- ⑦** `output` 属性可定义在成功构建后推送新容器镜像的位置。



## 参考文献

有关自定义构建镜像和构建输入源的更多信息，请参见红帽 OpenShift 容器平台 4.6 Builds 指南中的 Creating Build Inputs 章节，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/builds/creating-build-inputs](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/creating-build-inputs)

源与二进制 S2I 构建在

<https://developers.redhat.com/blog/2018/09/26/source-versus-binary-s2i-workflows-with-red-hat-openshift-application-runtimes/>  
中进行了更详细的说明

## ► 小测验

# OpenShift 构建流程

选择以下问题的正确答案：

► 1. 以下哪三项是可用来构建容器镜像的有效策略？（请选择三项。）

- a. Docker
- b. Git
- c. 源至镜像
- d. 自定义

► 2. 以下哪两项是可用来构建容器镜像的有效输入源类型？（请选择两项。）

- a. Git
- b. SVN
- c. Filesystem
- d. Containerfile

## ► 3. 根据上述 BuildConfig，以下哪三项陈述是正确的？（请选择三项。）

```
{  
    "kind": "BuildConfig",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "php-example",  
    },  
    "spec": {  
        ...  
        "runPolicy": "Serial",  
        "source": {  
            "type": "Git",  
            "git": {  
                "uri": "http://services.lab.example.com/php-helloworld"  
            }  
        },  
        "strategy": {  
            "type": "Source",  
            "sourceStrategy": {  
                "from": {  
                    "kind": "ImageStreamTag",  
                    "namespace": "openshift",  
                    "name": "php:7.0"  
                }  
            }  
        },  
        "output": {  
            "to": {  
                "kind": "ImageStreamTag",  
                "name": "php-example:latest"  
            }  
        },  
        ...  
    },  
}
```

- a. 多个构建操作可以同时运行。
- b. 成功构建后，**php-example:latest** 容器镜像便可加以部署。
- c. 构建操作使用 Git 输入源来创建最终容器镜像。
- d. **BuildConfig** 使用 **Docker** 策略。
- e. **BuildConfig** 使用 **Source** 策略。

► 4. 对于 Docker 策略，`oc new-app` 命令生成的构建配置会声明以下哪个输入源？

- a. Containerfile
- b. Binary
- c. Git
- d. 以上皆不是

► 5. 开发人员想要采用 Source 策略构建一个 PHP 应用。以下哪个选项可以配置构建配置，以使用 S2I 构建器镜像构建 PHP 应用？

- a. 为 Source 策略使用 `from` 属性。
- b. 使用 Image 输入源的 `from` 属性。
- c. 使用 Containerfile 输入源的 `from` 属性。
- d. 不一定要配置 S2I 构建器镜像。在构建期间，源至镜像流程将识别源语言，并自动选择 PHP 构建器镜像。
- e. 以上皆不是。

## ► 解决方案

# OpenShift 构建流程

选择以下问题的正确答案：

► 1. 以下哪三项是可用来构建容器镜像的有效策略？（请选择三项。）

- a. Docker
- b. Git
- c. 源至镜像
- d. 自定义

► 2. 以下哪两项是可用来构建容器镜像的有效输入源类型？（请选择两项。）

- a. Git
- b. SVN
- c. Filesystem
- d. Containerfile

## ► 3. 根据上述 BuildConfig，以下哪三项陈述是正确的？（请选择三项。）

```
{  
    "kind": "BuildConfig",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "php-example",  
    },  
    "spec": {  
        ...  
        "runPolicy": "Serial",  
        "source": {  
            "type": "Git",  
            "git": {  
                "uri": "http://services.lab.example.com/php-helloworld"  
            }  
        },  
        "strategy": {  
            "type": "Source",  
            "sourceStrategy": {  
                "from": {  
                    "kind": "ImageStreamTag",  
                    "namespace": "openshift",  
                    "name": "php:7.0"  
                }  
            }  
        },  
        "output": {  
            "to": {  
                "kind": "ImageStreamTag",  
                "name": "php-example:latest"  
            }  
        },  
        ...  
    },  
}
```

- a. 多个构建操作可以同时运行。
- b. 成功构建后，`php-example:latest` 容器镜像便可加以部署。
- c. 构建操作使用 Git 输入源来创建最终容器镜像。
- d. BuildConfig 使用 Docker 策略。
- e. BuildConfig 使用 Source 策略。

► 4. 对于 Docker 策略，`oc new-app` 命令生成的构建配置会声明以下哪个输入源？

- a. Containerfile
- b. Binary
- c. Git
- d. 以上皆不是

► 5. 开发人员想要采用 Source 策略构建一个 PHP 应用。以下哪个选项可以配置构建配置，以使用 S2I 构建器镜像构建 PHP 应用？

- a. 为 Source 策略使用 `from` 属性。
- b. 使用 Image 输入源的 `from` 属性。
- c. 使用 Containerfile 输入源的 `from` 属性。
- d. 不一定要配置 S2I 构建器镜像。在构建期间，源至镜像流程将识别源语言，并自动选择 PHP 构建器镜像。
- e. 以上皆不是。

# 管理应用构建

## 培训目标

学完本节后，您应能够使用构建配置资源和 CLI 命令来管理应用构建。

## 创建构建配置

红帽 OpenShift 容器平台通过构建配置资源来管理构建。构建配置可通过两种方式来创建：

### 使用 `oc new-app` 命令

该命令可根据指定的参数来创建构建配置。例如，如果已定义 Git 存储库，则会创建一个采用 Source 策略的构建配置。此外，如果该参数的值是一个模板，且该模板定义了构建配置，则会基于模板参数创建一个构建配置。

### 使用自定义构建配置

使用 YAML 或 JSON 语法创建一个自定义构建配置，然后使用 `oc create -f` 命令导入至红帽 OpenShift。

## 使用 CLI 管理应用构建

红帽 OpenShift 提供多个选项，允许开发人员使用 CLI 来管理应用构建和构建配置。这些命令用于管理应用的生命周期，尤其是在开发期间。请注意，与部署相比，构建配置是一个单独的阶段。红帽 OpenShift 中的成功构建会产生新的容器镜像，从而触发新的部署。

### `oc start-build`

手动启动新构建。构建配置资源的名称是启动新构建所需的唯一参数。

```
[user@host ~]$ oc start-build name
```

成功构建会在输出的镜像流标签中创建一个新的容器镜像。如果部署配置在该镜像流标签上定义了触发器，则部署过程将会启动。

### `oc cancel-build`

取消构建。例如：如果启动构建时使用了错误的应用版本，应用无法部署，您可能需要在失败之前取消构建。

只能取消处于正在运行或待处理状态的构建。取消构建意味着构建容器集会终止，所以不会推送新的容器镜像。因此，不会触发部署。

请提供构建配置资源名称，以取消构建：

```
[user@host ~]$ oc cancel-build name
```

### `oc delete`

删除构建配置。一般来说，当您需要从某个文件导入新的构建配置时，您会先删除一个构建配置。回想一下，`bc` 是构建配置的简写表示法。

```
[user@host ~]$ oc delete bc/name
```

## 章 4 | 在 OpenShift 上管理构建

以下命令将删除构建（而不是构建配置），以收回构建所占用的空间。构建配置可以有多个构建。请提供构建名称，以删除构建：

```
[user@host ~]$ oc delete build/name-1
```

### oc describe

介绍构建配置资源和相关构建的详情，提供标签、策略、webhook 等信息。

```
[user@host ~]$ oc describe bc name
```

描述构建并提供构建名称：

```
[user@host ~]$ oc describe build name-1
```

### oc logs

显示构建日志。您可以检查自己的应用是否在正确构建。这个命令还可以显示已完成的构建的相关日志。不能查看已删除或已修剪的构建的相关日志。构建日志可通过两种方式来显示：

- 显示最近一次构建的构建日志。

```
[user@host ~]$ oc logs -f bc/name
```

-f 选项在日志后显示，直至使用 **Ctrl+C** 终止命令

- 显示特定构建的构建日志：

```
[user@host ~]$ oc logs build/name-1
```

## 修剪构建

默认情况下，将保留最近完成的五个构建。您可以使用 **successfulBuildsHistoryLimit** 属性和 **failedBuildsHistoryLimit** 属性限制保留的先前构建数，如构建配置的以下 YAML 代码段所示：

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

失败的构建包括状态为 **failed**、**canceled** 或 **error** 的构建。构建按其创建时间排序，其中最旧的构建首先被修剪。



### 注意

红帽 OpenShift 管理员可以使用 **oc adm** 对象修剪命令手动修剪构建。

## 日志详细程度

红帽 OpenShift 提供了两种不同的机制来配置构建日志的详细程度。第一种是全局配置，即管理员可以针对整个集群定义构建日志详细程度配置。第二种仅影响来自特定构建配置的构建日志详细程度。这意味着在开发人员需要不同级别的日志详细程度时，他们仍然可以覆盖全局配置以进行更具体的构建配置。

要覆盖管理员的默认设置，开发人员可以编辑构建配置资源，并将 **BUILD\_LOGLEVEL** 环境变量添加为 Source 策略或 Docker 策略的一部分，以配置具体的日志级别：

```
[user@host ~]$ oc set env bc/name BUILD_LOGLEVEL="4"
```

该变量的值必须是零到五之间的数字。零是默认值，其所显示的日志比五少。增加数量时，日志记录消息的详细程度会更大，并且日志包含更多详细信息。

## 修剪构建

“修剪”构建意味着删除已完成或已失败的构建。红帽 OpenShift 可以根据配置自动执行这一操作，或者，您也可以使用 **oc delete** 或 **oc adm prune** 命令手动删减构建。

管理员可以修剪构建并释放磁盘空间，以免 **etcd** 数据存储量不断累积。某些配置可用于修剪构建，如：修剪孤立项、基于成功构建的数量来进行修剪、基于失败构建的数量来进行修剪，等等。

有时，您可能要请管理员更改修剪配置，从而使构建日志的保留时间不长于默认值，以便您针对构建问题进行故障排除。这些主题超出了当前课程的范围。



### 参考文献

有关管理应用构建的更多信息，请参见红帽 OpenShift 容器平台 4.10 Builds 中的

Performing Basic Builds 章节，网址为：

<https://access.redhat.com/documentation/en-us/>

openshift\_container\_platform/4.10/html-single/cicd/index#basic-build-operations

## ► 指导练习

# 管理应用构建

在本练习中，您将通过 OpenShift 采用 Source 策略和 Git 输入源来管理应用构建。

## 成果

您应该能够使用 CLI 来管理 OpenShift 上的构建。

## 在你开始之前

要进行此练习，您需要获得以下资源的访问权限：

- 运行中的 OpenShift 集群。
- OpenJDK 1.8 S2I 构建器镜像和镜像流 (`redhat-openjdk18-openshift:1.8`)。
- Git 存储库中的示例应用，位于 `java-serverhost` 目录中。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab manage-builds start
```

## 说明

### ► 1. 检查示例应用的 Java 源代码。

- 1.1. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `main` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 1.2. 创建一个新分支，以保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b manage-builds
Switched to a new branch 'manage-builds'
[student@workstation D0288-apps]$ git push -u origin manage-builds
...output omitted...
* [new branch]      manage-builds -> manage-builds
Branch manage-builds set up to track remote branch manage-builds from origin.
```

- 1.3. 查看 `java-serverhost` 文件夹内应用的 Java 源代码。

检查 `/home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/jaserverhost/rest/ServerHostEndPoint.java` 文件：

```
package com.redhat.training.example.javaserverhost.rest;

import javax.ws.rs.Path;
import javax.ws.rs.core.Response;
import javax.ws.rs.GET;
import javax.ws.rs.Produces;
import java.net.InetAddress;

@Path("/")
public class ServerHostEndPoint {

    @GET
    @Produces("text/plain")
    public Response doGet() {
        String host = "";
        try {
            host = InetAddress.getLocalHost().getHostName();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        String msg = "I am running on server " + host + " Version 1.0 \n";
        return Response.ok(msg).build();
    }
}
```

应用实施一个简单的 REST 服务，该服务返回其运行时所在的服务器的主机名。

## ▶ 2. 创建新项目。

### 2.1. 提供您的课堂环境配置：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

### 2.2. 使用您的开发人员用户登录 OpenShift：

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

### 2.3. 创建新项目来托管应用：

```
[student@workstation D0288-apps]$ oc new-project \
${RHT_OCP4_DEV_USER}-manage-builds
Now using project "developer-manage-builds" on server "https://
api.ocp4.example.com:6443".
...output-omitted...
```

## ▶ 3. 创建新应用。

### 3.1. 基于 Git 中的来源创建新应用。将下列参数用于构建：

- 应用名称：jhost

## 章 4 | 在 OpenShift 上管理构建

- 分支: `manage-builds`
- 构建环境变量: `MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java`
- 镜像流: `redhat-openjdk18-openshift:1.8`
- 构建目录: `java-serverhost`

您可以执行 `/home/student/D0288/labs/manage-builds/oc-new-app.sh` 脚本, 也可以手动执行命令:

```
[student@workstation D0288-apps]$ oc new-app --name jhost \
--build-env MAVEN_MIRROR_URL=http://${RHT_OCP4_NEXUS_SERVER}/repository/java \
-i redhat-openjdk18-openshift:1.8 \
https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#manage-builds \
--context-dir java-serverhost
...output omitted...
imagestream.image.openshift.io "jhost" created
buildconfig.build.openshift.io "jhost" created
deployment.apps "jhost" created
service "jhost" created
...output omitted...
```

**注意**

环境变量 `MAVEN_MIRROR_URL=` 后不要加空格。环境变量采用 `NAME=VALUE` 格式。

3.2. 使用 `oc logs` 命令检查 `jhost` 构建的构建日志:

```
[student@workstation D0288-apps]$ oc logs -f bc/jhost
...output omitted...
Writing manifest to image destination
Storing signatures
...output omitted...
Push successful
```

## 3.3. 等待应用准备就绪并在运行:

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
jhost-1-build  0/1     Completed  0          32m
jhost-7d9b748448-qwtqs  1/1     Running   0          30m
```

## 3.4. 公开应用供外部访问:

```
[student@workstation D0288-apps]$ oc expose svc/jhost
route.route.openshift.io/jhost exposed
```

## 3.5. 获取新路由的主机名:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT
jhost     jhost-yourdevuser-manage-builds.apps.ocp4.example.com  ...
```

3.6. 测试应用：

```
[student@workstation D0288-apps]$ curl \
http://jhost-${RHT_OCP4_DEV_USER}-manage-builds.${RHT_OCP4_WILDCARD_DOMAIN}
I am running on server jhost-6c8b694c58-2sxdn Version 1.0
```



### 注意

如果上一命令返回 HTML 页面，并显示 `Application is not available` 字样，请等待几秒钟，再试一次。

► 4. 列出构建配置和构建。

4.1. 列出项目中的所有构建配置：

```
[student@workstation D0288-apps]$ oc get bc
NAME      TYPE      FROM          LATEST
jhost     Source    Git@manage-builds  1
```

`oc new-app` 命令使用 `Source` 策略和 `Git` 输入源创建了 `jhost` 构建配置资源。

4.2. `oc new-app` 命令创建的构建配置启动了一个构建。列出项目中所有可用的构建：

```
[student@workstation D0288-apps]$ oc get builds
NAME      TYPE      FROM          STATUS      STARTED      DURATION
jhost-1   Source    Git@cb73a3d  Complete    18 minutes ago  2m4s
```

► 5. 将应用更新至 2.0 版。

5.1. 编辑 `/home/student/D0288-apps/java-serverhost/src/main/java/com/redhat/training/example/javaserverhost/rest/ServerHostEndPoint.java` 文件，并更新至 2.0 版：

```
... code omitted ...
String msg = "I am running on server \"host\" Version 2.0 \n";
return Response.ok(msg).build();
... code omitted ...
```

5.2. 将更改提交到 Git 服务器。

```
[student@workstation D0288-apps]$ cd java-serverhost
[student@workstation java-serverhost]$ git commit -a -m 'Update the version'
...output omitted...
1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation java-serverhost]$ cd ..
```

5.3. 使用 `oc start-build` 命令启动新构建：

```
[student@workstation D0288-apps]$ oc start-build bc/jhost
build.build.openshift.io/jhost-2 started
```

5.4. 应用不包含您提交至本地 Git 存储库的更改。应用使用远程 Git 存储库 URL 作为来源。您必须先将更改推送到远程 Git 存储库，然后才能启动 OpenShift 构建。

取消构建，以免新部署使用较旧版本：

```
[student@workstation D0288-apps]$ oc cancel-build bc/jhost
build.build.openshift.io/jhost-2 marked for cancellation, waiting to be cancelled
build.build.openshift.io/jhost-2 cancelled
```

5.5. 检查构建是否已取消：

```
[student@workstation D0288-apps]$ oc get builds
NAME      TYPE      FROM          STATUS     ...
jhost-1   Source    Git@cb73a3d  Complete   ...
jhost-2   Source    Git@cb73a3d  Cancelled (CancelledBuild) ...
```

如果没有在构建完成之前取消构建，您会看到以下输出：

```
[student@workstation D0288-apps]$ oc get builds
NAME      TYPE      FROM          STATUS     ...
jhost-1   Source    Git@cb73a3d  Complete   ...
jhost-2   Source    Git@20d7733  Complete   ...
```

继续以下步骤。

5.6. 将更新的源代码推送到 Git 服务器：

```
[student@workstation D0288-apps]$ git push
...output omitted...
2aa4b3a..f70977b manage-builds -> manage-builds
[student@workstation java-serverhost]$ cd
```

5.7. 启动新构建，以部署应用的更新版本：

```
[student@workstation ~]$ oc start-build bc/jhost
build.build.openshift.io/jhost-3 started
```

5.8. 列出所有构建：

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM          STATUS     ...
jhost-1   Source    Git@cb73a3d  Complete   ...
jhost-2   Source    Git@cb73a3d  Cancelled (CancelledBuild) ...
jhost-3   Source    Git          Running   ...
```

通过观察发现，共有三个构建可用。第一个由 `oc new-app` 命令创建，第二个已被您取消，第三个可构建经过更新的应用。

## 章 4 | 在 OpenShift 上管理构建

5.9. 根据应用构建日志采取相应措施：

```
[student@workstation ~]$ oc logs -f build/jhost-3  
...output omitted...  
Push successful
```

5.10. 等待应用准备就绪并在运行：

```
[student@workstation ~]$ oc get pods  
NAME          READY   STATUS    RESTARTS   AGE  
jhost-1-build  0/1     Completed  0          53m  
jhost-597888u-kqrwd  1/1     Running   0          4m40s  
jhost-3-build  0/1     Completed  0          6m21s
```

5.11. 测试经过更新的应用：

```
[student@workstation ~]$ curl \  
http://jhost-${RHT_OCP4_DEV_USER}-manage-builds.${RHT_OCP4_WILDCARD_DOMAIN}  
I am running on server jhost-76f7cf9d9c-84547 Version 2.0
```



### 注意

如果上一命令返回 HTML 页面，并显示 `Application is not available` 字样，请等待几秒钟，再试一次。

▶ 6. 清理并删除项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-manage-builds  
project.project.openshift.io "yourdevuser-manage-builds" deleted
```

## 完成

在 `workstation` 上，执行 `lab manage-builds finish` 脚本以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab manage-builds finish
```

本引导式练习到此结束。

# 触发构建

## 培训目标

学完本节后，您应能够使用支持的方法触发构建流程。

## 定义构建触发器

在红帽 OpenShift 中，您可以定义构建触发器，以允许平台根据某些事件自动启动新构建。您可以使用这些构建触发器以使应用容器一直通过任何影响应用的新容器镜像或代码更改来进行更新。红帽 OpenShift 可以定义两类构建触发器：

### 镜像更改触发器

镜像更改触发器会重新构建应用容器镜像，以体现其父镜像所做的更改。

### Webhook 触发器

红帽 OpenShift webhook 触发器是可启动新构建的 HTTP API 端点。使用 Webhook 将红帽 OpenShift 与版本控制系统 (VCS)（如 Github 或 BitBucket）集成，以基于代码更改触发新构建。

镜像更改触发器使得开发人员无需密切留意应用父镜像中的各种变化。当红帽 OpenShift 内部注册表检测到新镜像时，可以自动激活镜像更改触发器。如果镜像来自外部注册表，则必须定期运行 `oc import-image` 命令以验证容器镜像在注册表服务器中是否已更改，以便保持最新。

`oc new-app` 命令会自动采用 Source 或 Docker 构建策略为应用创建镜像更改触发器。

- 采用 Source 策略时，父镜像就是应用编程语言的 S2I 构建器镜像。
- 采用 Docker 策略时，父镜像就是应用 Containerfile 中 `FROM` 指令所引用的镜像。

要查看与构建配置关联的触发器，请使用 `oc describe bc` 命令，如以下示例所示：

```
[user@host ~]$ oc describe bc/name
```

要为构建配置添加镜像更改触发器，请使用 `oc set triggers` 命令：

```
[user@host ~]$ oc set triggers bc/name --from-image=project/image:tag
```

单个构建配置不能包含多个镜像更改触发器。

要从构建配置中删除镜像更改触发器，请使用包含 `--remove` 选项的 `oc set triggers` 命令：

```
[user@host ~]$ oc set triggers bc/name --from-image=project/image:tag --remove
```

使用 `oc set triggers --help` 命令可查看用于添加和移除配置更改触发器的选项。

## 使用 Webhook 触发器启动新构建

红帽 OpenShift webhook 触发器是可启动新构建的 HTTP API 端点。如果使用 webhook 以更改应用源代码的方式将红帽 OpenShift 与版本控制系统 (VCS) (如 Git) 进行整合，则会在红帽 OpenShift 中使用最新代码触发新的构建。

即使软件并非 VCS，也可以使用这些 API 端点，但是红帽 OpenShift 构建只能从 Git 服务器获取源代码。

红帽 OpenShift 容器平台提供了专门的 webhook 类型，用于支持与以下 VCS 服务兼容的 API 端点：

- GitLab
- GitHub
- Bitbucket

红帽 OpenShift 容器平台还提供了一种通用 webhook 类型，用于获取红帽 OpenShift 定义的载荷。通用 webhook 可供任意软件用于启动红帽 OpenShift 构建。请参见本节末尾的产品文档参考资料，以了解通用 webhook 载荷的语法以及可针对各类 webhook 提出的 HTTP API 请求。

对于所有 webhook，必须使用名为 **WebHookSecretKey** 的密钥定义机密，并且值是调用 webhook 时要提供的值。然后，webhook 定义必须引用该机密。该机密可确保 URL 的唯一性，防止其他人触发构建。密钥的值将与 webhook 调用期间提供的机密进行比较。每当您创建触发器或红帽 OpenShift 自动创建触发器时，默认情况下也会创建机密。

**oc new-app** 命令可以创建通用 webhook 和 Git webhook。要向构建配置添加其他类型的 webhook，请使用 **oc set triggers** 命令。例如，要向构建配置添加 GitLab webhook，请使用以下命令：

```
[user@host ~]$ oc set triggers bc/name --from-gitlab
```

如果构建配置已包含 GitLab webhook，则先前的命令会重置嵌在 URL 中的身份验证机密。您必须更新自己的 GitLab 项目，以使用新的 webhook URL。

要从构建配置中移除现有 webhook，请使用包含 **--remove** 选项的 **oc set triggers** 命令。例如，要从构建配置中移除 GitLab webhook，请使用以下命令：

```
[user@host ~]$ oc set triggers bc/name --from-gitlab --remove
```

**oc set triggers bc** 命令还支持 **--from-github** 和 **--from-bitbucket** 选项，以创建特定于每个 VCS 平台的触发器。

要检索 webhook URL 和机密，请使用 **oc describe** 命令并查找您所需的特定 webhook 类型。



### 参考文献

有关构建触发器的更多信息，请参阅红帽 OpenShift 容器平台 4.10 的 CICD 指南中的 Triggering and Modifying Builds 章节，网址为：  
[https://access.redhat.com/documentation/en-us/redshift\\_container\\_platform/4.10/html-single/cicd/index#triggering-build-hooks](https://access.redhat.com/documentation/en-us/redshift_container_platform/4.10/html-single/cicd/index#triggering-build-hooks)

## ► 指导练习

# 触发构建

在本练习中，您将在 OpenShift 中触发新的应用构建，以纳入对应用 S2I 构建器镜像进行的更新。

## 成果

您应能够：

- 创建基于 S2I 构建器镜像的应用。
- 推送 S2I 构建器镜像的新版本。
- 更新镜像流中的元数据，以响应镜像的更新。
- 验证应用是否已重新构建，以使用新的 S2I 构建器镜像。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- PHP S2I 构建器镜像的原始版本 (`php-73-ubi8-original.tar.gz`)
- PHP S2I 构建器镜像的新版本 (`php-73-ubi8-newer.tar.gz`)
- Git 存储库中的示例应用 (`trigger-builds`)。

在 `workstation` 虚拟机上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab trigger-builds start
```

## 说明

### ► 1. 准备实验环境。

1.1. 提供课堂环境配置：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户登录红帽 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

1.3. 为应用创建新项目：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-trigger-builds
```

► 2. 将镜像流添加到要与新应用一起使用的项目。

2.1. 使用 Podman 登录您的个人 Quay.io 帐户。

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io  
Password:  
Login Succeeded!
```

2.2. 将原始 PHP 7.3 构建器镜像推送到您的 Quay.io 注册表。

您可以使用 `/home/student/D0288/labs/trigger-builds/push-original.sh` 脚本复制或执行命令：

```
[student@workstation ~]$ cd /home/student/D0288/labs/trigger-builds  
[student@workstation trigger-builds]$ skopeo copy --format v2s1 \  
oci-archive:php-73-ubi8-original.tar.gz \  
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8:latest  
Getting image source signatures  
...output omitted...  
Writing manifest to image destination  
Storing signatures
```

2.3. Quay.io 注册表默认为私有镜像，因此您必须向构建器服务帐户添加机密才能访问它。

从 Podman 存储的容器注册表 API 访问令牌创建机密。

```
[student@workstation trigger-builds]$ oc create secret generic quay-registry \  
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \  
--type kubernetes.io/dockerconfigjson  
secret/quay-registry created
```

将 Quay.io 注册表机密添加到构建器服务帐户。

```
[student@workstation trigger-builds]$ oc secrets link builder quay-registry
```

2.4. 更新 `php` 镜像流，以从新的容器镜像中获取元数据。外部注册表会使用 `docker-distribution` 软件包，而且不会向红帽 OpenShift 发送镜像更改通知。

```
[student@workstation trigger-builds]$ oc import-image php \  
--from quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8 --confirm  
imagestream.image.openshift.io/php-73-ubi8 imported  
  
Name:          php  
...output omitted...  
latest  
tagged from quay.io/yourquayuser/php-73-ubi8  
  
* quay.io/yourquayuser/php-73-ubi8@sha256:c919...8cb2  
  Less than a second ago  
...output omitted...
```

► 3. 创建新应用。

3.1. 使用下列参数创建新的应用构建：

## 章 4 | 在 OpenShift 上管理构建

- 名称: `trigger`
- 构建器镜像: `php`
- 应用目录: `trigger-builds`

您可以使用 `/home/student/D0288/labs/trigger-builds/oc-new-app.sh` 脚本复制或执行完整命令：

```
[student@workstation trigger-builds]$ oc new-app \
--name trigger \
php~http://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
--context-dir trigger-builds
--> Found image ... "yourdevuser-trigger-builds/php" ... for "php"

...output omitted...

--> Success
Build scheduled, use 'oc logs -f buildconfig/trigger' to track its progress.
...output omitted...
```

### 3.2. 等待构建完成。

```
[student@workstation trigger-builds]$ oc logs -f bc/trigger
...output omitted...
Push successful
```

### 3.3. 等待应用准备就绪并在运行：

```
[student@workstation trigger-builds]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
trigger-1-build  0/1     Completed  0          10m
trigger-1-q6bln  1/1     Running   0          9m2s
```

### 3.4. 验证是否已在构建配置中定义镜像更改触发器：

```
[student@workstation trigger-builds]$ oc describe bc/trigger | grep Triggered
Triggered by: Config, ImageChange
```

如果镜像流检测到其基础镜像有所变化，则最后一个触发器（即 `ImageChange` 触发器）会启动新构建。

## ▶ 4. 更新镜像流以启动新构建。

### 4.1. 将新版本的 PHP S2I 构建器镜像上传到 Quay.io 注册表

您可以使用 `/home/student/D0288/labs/trigger-builds/push-newer.sh` 脚本复制或执行完整命令：

## 第4章 | 在 OpenShift 上管理构建

```
[student@workstation trigger-builds]$ skopeo copy --format v2s1 \
oci-archive:php-73-ubi8-newer.tar.gz \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8:latest
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

4.2. 更新 php 镜像流，以从新的容器镜像中获取元数据。

```
[student@workstation trigger-builds]$ oc import-image php
imagestream.image.openshift.io/php-73-ubi8 imported

Name:          php
...output omitted...
latest
tagged from quay.io/yourquayuser/php-73-ubi8

* quay.io/yourquayuser/php-73-ubi8@sha256:3f5e...6ab7
  Less than a second ago
quay.io/yourquayuser/php-73-ubi8@sha256:c919...8cb2
  2 minutes ago
...output omitted...
```

## ▶ 5. 检查新构建。

5.1. 更新镜像流操作触发了新的构建。列出所有构建，以验证第二个构建是否已启动：

```
[student@workstation trigger-builds]$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED      DURATION
trigger-1  Source    Git@da010df  Complete   13 minutes ago  58s
trigger-2  Source    Git@da010df  Complete   28 seconds ago  15s
```

5.2. 确认 trigger-2 构建是否因镜像流更新而启动：

```
[student@workstation trigger-builds]$ oc describe build trigger-2 | grep cause
Build trigger cause: Image change
```

## ▶ 6. 完成。

6.1. 该命令还可以更改到主目录。

```
[student@workstation trigger-builds]$ cd
```

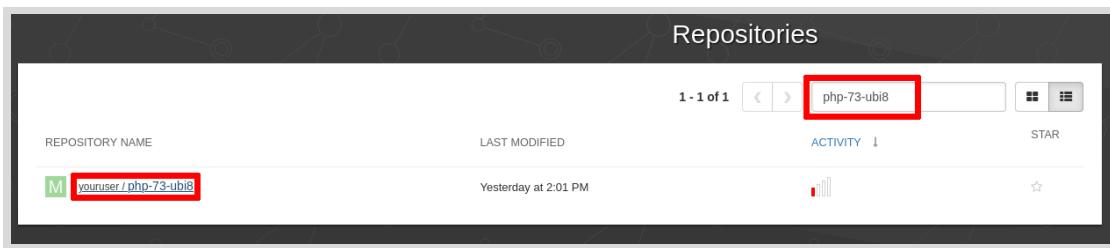
6.2. 从外部注册表中删除容器镜像：

```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/php-73-ubi8
```

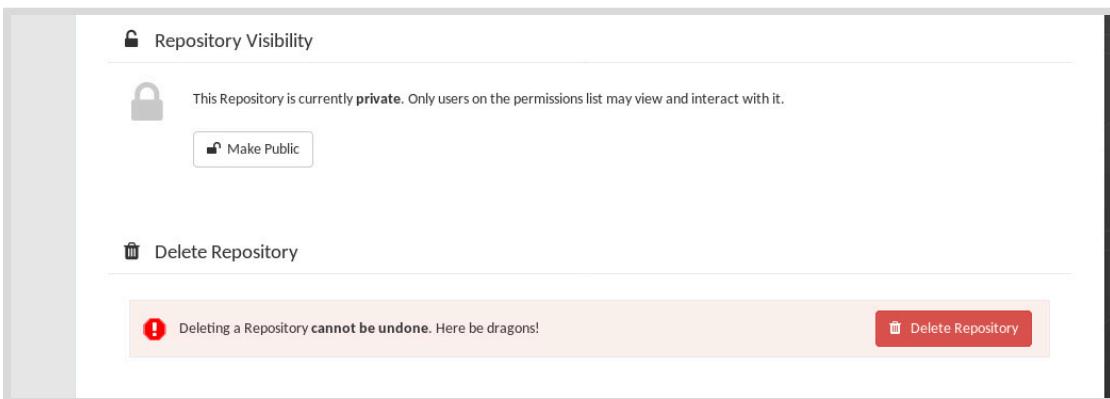
6.3. 使用您的个人免费帐户登录 Quay.io。

## 章 4 | 在 OpenShift 上管理构建

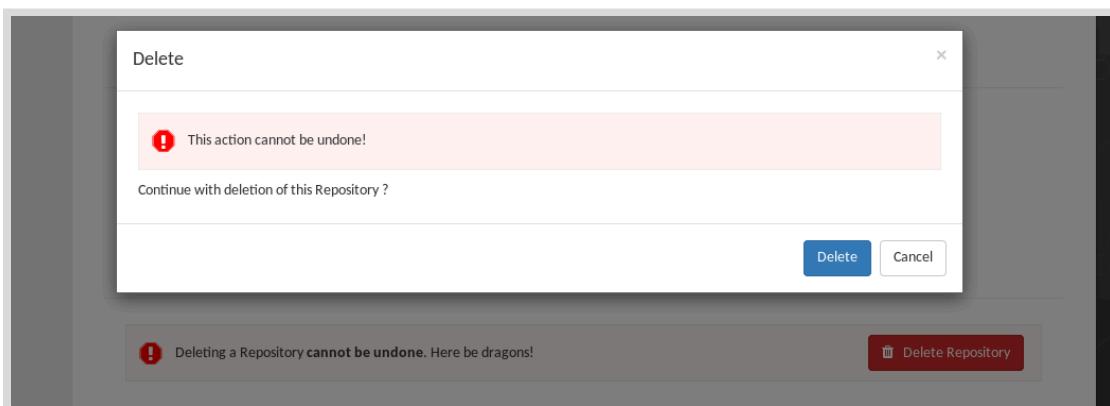
- 6.4. 在 Quay.io 的顶级菜单上，单击 **Repositories**，再使用名称 **php-73-ubi8** 过滤存储库。然后，单击 **php-73-ubi8** 存储库。



- 6.5. 在 **php-73-ubi8** 存储库的 **Repository Activity** 页面，单击齿轮图标。向下滚动，直到找到 **Delete Repository** 按钮并单击它。



- 6.6. 在 Delete 窗口，单击 **Delete** 以确认删除 **php-73-ubi8** 存储库。



## 完成

在 **workstation** 上，执行 **lab trigger-builds finish** 脚本来完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab trigger-builds finish
```

本引导式练习到此结束。

# 实施 Post-commit 构建 hook

## 培训目标

学完本节后，您应能够使用 post-commit 构建 hook 处理构建后逻辑。

## 描述 Post-commit 构建 hook

红帽 OpenShift 容器平台 (RHOCP) 提供了 **post-commit** 构建 hook 功能，可用于在构建期间执行验证任务。**post-commit** 构建 hook 会先在临时容器中运行命令，然后再将构建操作所生成的新容器镜像推送至注册表。该 hook 会使用构建创建的容器镜像来启动一个临时容器。

RHOCP 将根据在这个临时容器中执行命令后的退出代码来确定是否要将镜像推送至注册表。如果命令返回的是非零退出代码（即表示运行失败），则 RHOCP 不会推送镜像并会将构建标记为失败。如果命令运行成功，则 RHOCP 会将镜像推送至注册表。

通过使用 `oc logs` 命令检查构建日志，可以验证构建是否由于 post-commit hook 而失败：

```
[user@host ~]$ oc logs bc/name
...
Running post commit hook ...
...
error: build error: container "openshift_s2i-build_hook-2_post-commit_post-
commit_45ec0816" returned non-zero exit code: 35
```



### 注意

Post-commit hook 仅用于验证镜像，绝不会修改它。

## 查看 Post-commit 构建 Hook 的用例

使用 post-commit 构建 hook 的典型场景是在应用中执行某些测试。这样，在 RHOCP 将镜像推送至注册表并启动新部署之前，可通过测试来检查应用能否正常工作。如果测试失败，则构建失败，并且不会继续部署。

还有其他一些常见的用例，在利用 post-commit 构建 hook 时非常有用。例如：

- 通过 HTTP API 将构建与外部应用进行整合。
- 验证非功能性要求，如应用性能、安全性、可用性或兼容性。
- 向开发人员团队发送电子邮件，通知他们有新构建。

## 配置 Post-commit 构建 hook

您可以配置两类 post-commit 构建 hook：

### 命令

命令会通过 `exec` 系统调用来执行。使用以下命令中所示的 `--command` 选项创建命令 post-commit 构建 hook：

```
[user@host ~]$ oc set build-hook bc/name --post-commit \
--command -- bundle exec rake test --verbose
```



### 注意

-- 参数后面的空格并非误输入。-- 参数可将用于配置 post-commit hook 的 RHOCP 命令行与该 hook 所执行的命令分隔开。

## Shell 脚本

使用 `/bin/sh -ic` 命令运行构建 hook。这类 hook 更为简便，因为其具备 shell 可以提供的所有功能，如参数扩展、重定向等。但其仅在基础镜像包含 `sh` shell 时才可用。使用以下命令中所示的 `--script` 创建 shell 脚本 post-commit 构建 hook：

```
[user@host ~]$ oc set build-hook bc/name --post-commit \
--script="curl http://api.com/user/${USER}"
```



### 参考文献

有关 post-commit 构建 hook 的更多信息，请参见红帽 OpenShift 容器平台 4.6 Builds 指南中的 Triggering and Modifying Builds 章节，网址为  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.6/html-single/builds/triggering-builds-build-hooks](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/builds/triggering-builds-build-hooks)

## ► 指导练习

# 实施 Post-Commit 构建 hook

在本练习中，您将设置一个 post-commit 构建 hook，以便将构建与外部应用进行整合。

## 成果

您应该能将 post-commit 构建 hook 添加到构建配置资源中。该 post-commit 构建 hook 会向 **builds-for-managers** 应用发送 HTTP API 请求。

## 在你开始之前

**builds-for-managers** 应用可供经理用于跟踪开发人员团队所执行的应用构建。该应用会显示项目、Git URL、启动构建的开发人员等信息。您需将自己的构建与该应用整合，以便经理知晓您所做的工作。

要进行此练习，您需要获得以下资源的访问权限：

- 正在运行的红帽 OpenShift 集群。
- PHP S2I 构建器镜像 (**php-73-rhel7:7.3**)。
- builds-for-managers** 容器镜像 (**quay.io/redhattraining/builds-for-managers**)
- 应用 Git 存储库 (**post-commit**)。

在 **workstation** 虚拟机上运行以下命令，以验证前提条件、创建 **post-commit** 项目、启动 **builds-for-managers** 应用并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab post-commit start
```

## 说明

### ► 1. 准备实验环境。

1.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 登录红帽 OpenShift 集群。

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

1.3. 确保您正在处理 **yourdevuser-post-commit** 项目：

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-post-commit  
Already on project "yourdevuser-post-commit_" on server "https://...".
```

由于 `lab post-commit start` 命令创建了此项目，因此您应该已在此项目中。如果不在，您的输出可能与上述不同。

- 1.4. 验证 `builds-for-managers` 正在 `yourdevuser-post-commit` 项目中运行：

```
[student@workstation ~]$ oc status  
In project yourdevuser-post-commit on server https://api.ocp4.example.com:6443...  
  
http://builds-for-managers... to pod port 8080-tcp (svc/builds-for-managers)  
deployment/builds-for-managers deploys istag/builds-for-managers:latest  
deployment #1 deployed 5 minutes ago - 1 pod  
  
...output omitted...
```

▶ 2. 创建新应用。

- 2.1. 基于 Git 中的来源创建新应用。将该应用命名为 `hook`，并使用波形符 (~) 将 `php:7.3` 镜像流添加到 Git 存储库 URL 中。  
可直接通过 `/home/student/D0288/labs/post-commit` 文件夹中的 `oc-new-app.sh` 脚本来复制或执行完整的命令：

```
[student@workstation ~]$ oc new-app --name hook --context-dir post-commit \  
php:7.3~http://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps  
...output omitted...
```

- 2.2. 等待构建完成：

```
[student@workstation ~]$ oc logs -f bc/hook  
...output omitted...  
Push successful
```

- 2.3. 等待应用准备就绪并在运行：

```
[student@workstation ~]$ oc get pods  
NAME                      READY   STATUS    RESTARTS   AGE  
builds-for-managers-f{...}-rb62w  1/1     Running   0          8m  
hook-1-build                0/1     Completed  0          1m  
hook-5c4dcd4998-mqf5w        1/1     Running   0          11s
```

当前正在运行两个不同的应用。第一个是经理所用的 `builds-for-managers` 应用，第二个则是您的 PHP 应用。

▶ 3. 将 PHP 应用构建与 `builds-for-managers` 应用进行整合。

- 3.1. 检查 `/home/student/D0288/labs/post-commit` 文件夹中提供的 `create-hook.sh` 脚本。该脚本会创建一个构建 hook，以使用 `curl` 命令将您的 PHP 应用构建与 `builds-for-managers` 应用进行整合。

```
[student@workstation ~]$ cat ~/DO288/labs/post-commit/create-hook.sh
...output omitted...
oc set build-hook bc/hook --post-commit --command -- \
  bash -c "curl -s -S -i -X POST http://builds-for-managers-
${RHT_OCP4_DEV_USER}-post-commit.${RHT_OCP4_WILDCARD_DOMAIN}/api/builds
-f -d 'developer=${DEVELOPER}&git=${OPENSHIFT_BUILD_SOURCE}&project=\
${OPENSHIFT_BUILD_NAMESPACE}'"
```

curl 命令会向 builds-for-managers 应用发送三个环境变量：

- DEVELOPER：包含开发人员的姓名。
- OPENSHIFT\_BUILD\_SOURCE：包含项目 Git URL。
- OPENSHIFT\_BUILD\_NAMESPACE：包含项目名称。

### 3.2. 运行 create-hook.sh 脚本：

```
[student@workstation ~]$ ~/DO288/labs/post-commit/create-hook.sh
buildconfig.build.openshift.io/hook hooks updated
```

### 3.3. 验证 post-commit 构建 hook 是否已创建：

```
[student@workstation ~]$ oc describe bc/hook | grep Post
Post Commit Hook: ["bash", "-c", "\"curl -s -S -i -X POST http://builds-..."
```

### 3.4. 使用启用了 -F 选项的 oc start-build 命令启动新的构建，以显示日志并验证 HTTP API 响应代码。您将看到 post-commit hook 执行的 curl 命令所返回的 HTTP 400 状态代码：

```
[student@workstation ~]$ oc start-build bc/hook -F
build.build.openshift.io/hook-2 started
...output omitted...
[2/3] STEP 1/2: FROM 28cd...e4d9
[2/3] STEP 2/2: RUN bash -c "curl -s -S -i -X POST http://builds-for-managers-
developer-post-commit.apps.ocp4.example.com/api/builds -f -d 'developer=
${DEVELOPER}&git=${OPENSHIFT_BUILD_SOURCE}&project=${OPENSHIFT_BUILD_N
AMESPACE}'"
curl: (22) The requested URL returned error: 400 Bad Request
error: build error: error building at STEP "RUN bash -c "curl -s -S -i -X POST
http://builds-for-managers-developer-post-commit.apps.ocp4.example.com/api/
builds -f -d 'developer=${DEVELOPER}&git=${OPENSHIFT_BUILD_SOURCE}&project=
${OPENSHIFT_BUILD_N
AMESPACE}'"": error while running runtime: exit status 22
```

builds-for-managers 应用拒绝了 HTTP API 请求，因为 DEVELOPER 环境变量未定义。

### 3.5. 列出各个构建，并验证是否存在因 post-commit hook 失败而失败的构建：

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM          STATUS    ...output omitted...
hook-1    Source    Git@c2166cc  Complete
hook-2    Source    Git@c2166cc  Failed   (GenericBuildFailed)
```

▶ 4. 解决环境变量缺失问题。

- 4.1. 使用您的姓名和 `oc set env` 命令创建 `DEVELOPER` 构建环境变量：

```
[student@workstation ~]$ oc set env bc/hook DEVELOPER="Your Name"
buildconfig.build.openshift.io/hook updated
```

- 4.2. 验证 `hook` 构建配置中是否包含 `DEVELOPER` 环境变量：

```
[student@workstation ~]$ oc set env bc/hook --list
# buildconfigs hook
DEVELOPER=Your Name
```

- 4.3. 启动新的构建，并显示日志以验证 HTTP API 响应代码。您将看到 HTTP 200 状态代码：

```
[student@workstation ~]$ oc start-build bc/hook -F
build.build.openshift.io/hook-3 started
...output omitted...
[2/3] STEP 2/2: RUN bash -c "curl -s -S -i -X POST http://builds-for-managers-
developer-post-commit.apps.ocp4.example.com/api/builds -f -d 'developer=
${DEVELOPER}&git=${OPENSHIFT_BUILD_SOURCE}&project=${OPENSHIFT_BUILD_N
AMESPACE}'"
HTTP/1.1 200 OK
content-type: application/octet-stream
content-length: 15
date: Fri, 09 Sep 2022 13:58:34 GMT
set-cookie: 9bf95dc4d1ba28115289f03e97e86a81=34e8729534f17fa311f0ae8a01fb7f4a;
path=/; HttpOnly

Build persisted[3/3] STEP 1/1: FROM 199b...8665
[3/3] COMMIT temp.builder.openshift.io/developer-post-commit/hook-3:eb674866
--> 199bb23d2bf
Successfully tagged temp.builder.openshift.io/developer-post-commit/
hook-3:eb674866
199b...8665
...output omitted...
Push successful
```

以上 HTTP 200 状态代码代表 `builds-for-managers` 应用已接受 HTTP API 请求。

- 4.4. 打开 Web 浏览器，访问 `http://builds-for-managers-yourdevuser-post-commit.apps.ocp4.example.com`。

```
[student@workstation ~]$ firefox $(oc get route/builds-for-managers \
-o jsonpath='{.spec.host}') &
```

该页面会显示所有的构建以及启动各个构建的开发人员。

## Builds for Managers

Date	Developer	Project	Git Project
2019-07-10 07:08:43	Your Name	youruser-post-commit	<a href="http://github.com/youruser/DO288-apps">http://github.com/youruser/DO288-apps</a>

- ▶ 5. 清理：删除红帽 OpenShift 中的 `yourdevuser-post-commit` 项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-post-commit  
projects.project.openshift.io "yourdevuser-post-commit" deleted
```

## 完成

在 `workstation` 上，执行 `lab post-commit finish` 脚本来完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab post-commit finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 面向 OpenShift 构建应用

在本实验中，您将使用红帽 OpenShift 管理应用构建、对应用进行故障排除、使用 webhook 触发新构建。



### 注意

各个章节实验结尾处的评测脚本需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 通过管理构建的生命周期，对应用进行故障排除。
- 使用 webhook 触发新构建。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- Node.js 应用的 S2I 构建器镜像和镜像流 (**nodejs**)。
- Git 存储库中的应用 (**build-app**)。

在 **workstation** 虚拟机上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab build-app start
```

## 要求

提供的应用以 JavaScript 编写，且使用 Node.js 运行时。它是一个基于 Express 框架的简单应用。您应根据以下要求构建应用并将其部署至红帽 OpenShift 集群：

- 项目名称为 **yourdevuser-build-app**。
- 应用名称为 **simple**。使用实验室目录中的 **oc-new-app.sh** 脚本创建和部署应用。此脚本包含一个有意的错误，您会在后面的步骤中修复该错误。不要修改或编辑 **oc-new-app.sh** 脚本。
- 部署的应用是从 Git 存储库的 **build-app** 子目录中的源代码创建的：  
<https://github.com/yourgithubuser/DO288-apps>.
- 构建应用所需的 NPM 模块可从以下 Nexus 服务器 URL 获取：  
[http://\\${RHT\\_OCP4\\_NEXUS\\_SERVER}/repository/nodejs](http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs).

使用 **npm\_config\_registry** 环境变量将这一信息传输至 Node.js 的 S2I 构建器镜像。

## 章 4 | 在 OpenShift 上管理构建

- 该应用可从默认路由访问：

simple-yourdevuser-build-apps.ocp4.example.com.

## 说明

- 创建 yourdevuser-build-app 项目。
- 执行 /home/student/D0288/labs/build-app/oc-new-app.sh 脚本以创建应用。



### 重要

oc-new-app.sh 脚本中存在一个有意的错误，您会在后面的步骤中修复该错误。不要修改或编辑 oc-new-app.sh 脚本。

- 验证应用构建是否失败，再修复问题。
- 公开应用服务供外部访问并获取路由 URL。
- 启动新构建，并验证应用是否已准备就绪并在运行。使用您在上一步中获取的路由 URL 测试应用是否可访问。
- 使用构建配置的通用 webhook 启动新的应用构建。
- 对您的作业进行评分：

```
[student@workstation ~]$ lab build-app grade
```

- 清理并删除项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-app
```

## 完成

在 workstation 上，执行 lab build-app finish 脚本来完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab build-app finish
```

本实验到此结束。

## ▶ 解决方案

# 面向 OpenShift 构建应用

在本实验中，您将使用红帽 OpenShift 管理应用构建、对应用进行故障排除、使用 webhook 触发新构建。



### 注意

各个章节实验结尾处的评测脚本需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 通过管理构建的生命周期，对应用进行故障排除。
- 使用 webhook 触发新构建。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- Node.js 应用的 S2I 构建器镜像和镜像流 (**nodejs**)。
- Git 存储库中的应用 (**build-app**)。

在 **workstation** 虚拟机上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab build-app start
```

## 要求

提供的应用以 JavaScript 编写，且使用 Node.js 运行时。它是一个基于 Express 框架的简单应用。您应根据以下要求构建应用并将其部署至红帽 OpenShift 集群：

- 项目名称为 **yourdevuser-build-app**。
- 应用名称为 **simple**。使用实验室目录中的 **oc-new-app.sh** 脚本创建和部署应用。此脚本包含一个有意的错误，您会在后面的步骤中修复该错误。不要修改或编辑 **oc-new-app.sh** 脚本。
- 部署的应用是从 Git 存储库的 **build-app** 子目录中的源代码创建的：  
<https://github.com/yourgithubuser/DO288-apps>.
- 构建应用所需的 NPM 模块可从以下 Nexus 服务器 URL 获取：  
[http://\\${RHT\\_OCP4\\_NEXUS\\_SERVER}/repository/nodejs](http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs).

使用 **npm\_config\_registry** 环境变量将这一信息传输至 Node.js 的 S2I 构建器镜像。

## 章 4 | 在 OpenShift 上管理构建

- 该应用可从默认路由访问：

```
simple-yourdevuser-build-apps.ocp4.example.com.
```

## 说明

### 1. 创建 yourdevuser-build-app 项目。

- 加载您的课堂环境配置。

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 使用您的开发人员用户帐户登录红帽 OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

- 创建新项目来托管应用：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-build-app
```

### 2. 执行 /home/student/D0288/labs/build-app/oc-new-app.sh 脚本以创建应用。



#### 重要

oc-new-app.sh 脚本中存在一个有意的错误，您会在后面的步骤中修复该错误。不要修改或编辑 oc-new-app.sh 脚本。

- 查看用于创建该应用的脚本：

```
[student@workstation ~]$ cat ~/D0288/labs/build-app/oc-new-app.sh
...output omitted...
oc new-app --name simple --build-env \
  npm_config_registry=http://invalid-server:8081/nexus/content/groups/nodejs \
  https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
  --context-dir build-app
```

- 执行 oc-new-app.sh 脚本以创建应用：

```
[student@workstation ~]$ ~/D0288/labs/build-app/oc-new-app.sh
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

### 3. 验证应用构建是否失败，再修复问题。

- 查看构建日志，以识别构建错误。错误消息可能需要一些时间才会显示。

```
[student@workstation ~]$ oc logs -f bc/simple
...output omitted...
error: build error: error building at STEP "RUN /usr/libexec/s2i/assemble": error
while running runtime: exit status 1
...output omitted...
```



### 注意

命令可能会显示：

```
[student@workstation ~]$ oc logs -f bc/simple
...output omitted...
error: unexpected EOF
```

发生这种情况时，请重试该步骤。

3.2. 此次构建失败是由无效 Nexus 服务器 URL 导致的。确认 Nexus 服务器 URL 是否有误：

```
[student@workstation ~]$ oc set env bc simple --list
# buildconfigs simple
npm_config_registry=http://invalid-server:8081/nexus/content/groups/nodejs
```

3.3. 修复变量以使用正确的 Nexus 服务器 URL：

```
[student@workstation ~]$ oc set env bc simple \
npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs
buildconfig.build.openshift.io/simple updated
```

注意 `npm_config_registry` 后面的等号 (=) 前后没有空格。

3.4. 验证 `npm_config_registry` 变量的值是否正确：

```
[student@workstation ~]$ oc set env bc simple --list
# buildconfigs simple
npm_config_registry=http://nexus-common.../repository/nodejs
```

注意 `npm_config_registry` 后面的等号 (=) 前后没有空格。对于纸张宽度而言，这个构建环境变量的完整“键=值”对过长。

4. 公开应用服务供外部访问并获取路由 URL。

4.1. 公开服务：

```
[student@workstation ~]$ oc expose svc simple
route.route.openshift.io/simple exposed
```

4.2. 获取路由 URL：

```
[student@workstation ~]$ oc get route/simple \
-o jsonpath='{.spec.host}{"\n"}'
simple-yourdevuser-build-app.apps.ocp4.example.com
```

## 章 4 | 在 OpenShift 上管理构建

5. 启动新构建，并验证应用是否已准备就绪并在运行。使用您在上一步中获取的路由 URL 测试应用是否可访问。

5.1. 启动新构建并持续关注相关日志：

```
[student@workstation ~]$ oc start-build simple
build.build.openshift.io/simple-2 started
...output omitted...
Push successful
```

5.2. 等待应用准备就绪并在运行：

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
simple-1-build 0/1     Error      0          20m
simple-2-build 0/1     Completed   0          4m5s
simple-964487d7b-fs9gr 1/1     Running    0          5m41s
```

5.3. 测试应用：

```
[student@workstation ~]$ curl \
simple-${RHT_OCP4_DEV_USER}-build-app.${RHT_OCP4_WILDCARD_DOMAIN}
Simple app for the Building Applications Lab!
```

6. 使用构建配置的通用 webhook 启动新的应用构建。

6.1. 获取可使用 `oc describe` 命令启动新构建的通用 webhook URL。

```
[student@workstation ~]$ oc describe bc simple
Name: simple
...output omitted...
Webhook Generic:
URL: https://apps.ocp4.example.com/apis/build.openshift.io/v1/
namespaces/yourdevuser-build-app/buildconfigs/simple/webhooks/<secret>/generic
```

6.2. 通过运行 `oc get bc` 命令获取 webhook 的机密，并传递 `-o json` 选项以在 JSON 中转储构建配置详细信息。

```
[student@workstation ~]$ SECRET=$(oc get bc simple \
-o jsonpath=".spec.triggers[*].generic.secret\{\n\}")
```

6.3. 使用从上一步骤的输出结果中找到的 webhook URL 和机密来启动新构建。有关“invalid Content-Type on payload”的错误消息可以安全地忽略。

```
[student@workstation ~]$ curl -X POST -k \
${RHT_OCP4_MASTER_API}\
apis/build.openshift.io/v1/namespaces/${RHT_OCP4_DEV_USER}-build-app\
/buildconfigs/simple/webhooks/$SECRET/generic
...output omitted...
"status": "Success",
...output omitted...
```



### 注意

前面命令的第一行后不包含空格。如果 curl 命令失败，请验证：

- 您的 URL 不包含空格。
- 您的 **\$SECRET** 变量包含正确的机密。

6.4. 列出所有构建，并验证新构建是否已启动：

```
[student@workstation ~]$ oc get builds
NAME      TYPE      FROM      STATUS      STARTED ...
simple-1  Source    Git@3e14daf  Failed (AssembleFailed)  About an hour ago
simple-2  Source    Git@3e14daf  Complete    20 minutes ago
simple-3  Source    Git@3e14daf  Complete    32 seconds ago
```

6.5. 等待新构建完成：

```
[student@workstation ~]$ oc logs -f bc/simple
...output omitted...
Push successful
```

6.6. 等待应用准备就绪并在运行：

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
simple-1-build 0/1     Error     0          94m
simple-2-build 0/1     Completed  0          36m
simple-3-build 0/1     Completed  0          19m
simple-7c6995cdcf-phvk5 1/1     Running   0          16m
```

7. 对您的作业进行评分：

```
[student@workstation ~]$ lab build-app grade
```

8. 清理并删除项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-build-app
```

## 完成

在 workstation 上，执行 **lab build-app finish** 脚本来完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab build-app finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- **BuildConfig** 资源包含一个策略以及一个或多个输入源。
- 构建策略共有四种：**Source**、**Pipeline**、**Docker** 和 **Custom**。
- 按优先级顺序排列的六个构建输入源是：Dockerfile、Git、镜像、二进制、输入机密和外部文件。
- 使用 **oc** CLI 命令（如 **oc start-build**、**oc cancel-build**、**oc delete**、**oc describe** 和 **oc logs**）来管理构建生命周期。
- 构建可通过构建触发器（如镜像更改触发器和 webhook）自动启动。
- 可在构建期间使用 **post-commit** 构建 hook 来执行验证任务。

## 章 5

# 自定义源至镜像构建

### 目标

自定义现有 S2I 构建器镜像，并创建一个新镜像。

### 培训目标

- 介绍在源至镜像构建过程中必须要执行以及可选择性执行的步骤。
- 通过编写脚本，自定义现有 S2I 构建器镜像。
- 使用 S2I 工具新建 S2I 构建器镜像。

### 章节

- 介绍源至镜像架构（及测验）
- 自定义现有 S2I 构建器镜像（及引导式练习）
- 创建 S2I 基础镜像（及引导式练习）

### 实验

自定义源至镜像构建

# 介绍源至镜像架构

---

## 培训目标

学完本节后，您应能够介绍源至镜像构建过程中必须要执行以及可选择性执行的步骤。

## 源至镜像 (S2I) 语言检测

红帽 OpenShift (RHOCP) 可以直接基于 Git 存储库中存储的源代码来创建应用。`oc new-app` 命令的语法更简洁，它只会将 Git 存储库 URL 用作参数，然后，红帽 OpenShift 会尝试自动检测应用所用的编程语言，并选择兼容的构建器镜像。

自动检测逻辑并不完美。`oc new-app` 命令可以使用各种命令行选项，以强制做出特定选择。本课程会在前面几章介绍这些命令行选项。

编程语言检测功能会在 Git 存储库的根目录中查找特定的文件名称。下表显示了一些更常见的选项，但它并不是所有源至镜像兼容语言的广泛列表。请参见产品文档，以查看各个 RHOCP 版本的所有规则：

文件	语言构建器	编程语言
<code>Dockerfile</code>	不适用	Dockerfile 构建（非 S2I）
<code>pom.xml</code>	<code>jee</code>	Java（使用 JBoss EAP）
<code>app.json, package.json</code>	<code>nodejs</code>	Node.js (JavaScript)
<code>composer.json, index.php</code>	<code>php</code>	PHP

RHOCP 会采用多步算法来确定 URL 是否指向源代码存储库，如果是，则还会采用该算法来确定应由哪个构建器镜像来执行构建。以下是该算法的简单介绍：

1. RHOCP 会将 URL 当作容器注册表 URL 来访问。如能成功访问，则需创建构建配置。RHOCP 会创建部署容器镜像所需的部署和其他资源。
2. 如果 URL 指向某个 Git 存储库，则 RHOCP 会检索该存储库中的文件列表并搜索 `Dockerfile` 文件。如果找到了，则构建配置会使用 `docker` 策略。否则，构建配置会使用 `source` 策略，该策略需要使用 S2I 构建器镜像。
3. RHOCP 会搜索语言构建器名称与 `supports` 注释值相匹配的镜像流。搜索到的第一个匹配项会成为 S2I 构建器镜像。
4. 如果没有匹配的注释，则 RHOCP 会搜索名称与语言构建器名称相匹配的镜像流。搜索到的第一个匹配项会成为 S2I 构建器镜像。

只需执行第 3 步和第 4 步，即可轻松向 RHOCP 集群添加新的构建器镜像。这还意味着可能存在多个匹配的构建器镜像。本章前文中已介绍过使用 `oc new-app` 命令行选项可以避免这种不确定性，并确保 RHOCP 选择正确的镜像流作为 S2I 构建器镜像。

例如，当您运行以下命令时，`oc` 命令会识别出您正在引用注册表 URL，并启动容器部署：

```
[user@host ~]$ oc new-app registry.access.redhat.com/ubi8/ubi:8.0
```

或者，当您运行以下命令时，`oc` 命令会识别出您正在使用 Git 存储库，并且它将克隆存储库以查找 `Dockerfile` 文件。如果 RHOCP 集群在存储库的根目录中找到 `Dockerfile` 文件，则它会触发新的容器构建流程。

```
[user@host ~]$ oc new-app https://github.com/RedHatTraining/D0288-apps/ubi-echo
```

如果 RHOCP 集群在存储库的根目录中找到上表中提及的文件之一，则 RHOCP 集群将使用其相应的镜像构建器启动 S2I 进程。

为强制使用特定镜像流，可以使用 PHP 7.3 应用的 `-i` 选项：

```
[user@host ~]$ oc new-app -i php:7.3 \
https://github.com/RedHatTraining/D0288-apps/php-helloworld
```

RHOCP 集群查找名为 `php` 的镜像流，并查找要调用构建器的 7.3 版本。

## 源至镜像 (S2I) 构建流程

S2I 构建流程涉及三个基础组件；通过组合使用这些组件，可以创建最终容器镜像：

### 应用源代码

这是应用的源代码。

### S2I 脚本

S2I 脚本是一组由 RHOCP 构建流程执行以自定义 S2I 构建器镜像的脚本。S2I 脚本可以使用任意编程语言来编写，这些脚本可在 S2I 构建器镜像内执行。

### S2I 构建器镜像

这是包含应用所需运行时环境的容器镜像。该镜像通常包含运行应用所需的编译器、解释器、脚本和其他依赖项。

S2I 构建进程依赖于一些 S2I 脚本，它会在构建工作流的不同阶段执行这些脚本。下表列出了这些脚本，并简单介绍了这些脚本所执行的操作：

脚本	必需？	描述
<code>assemble</code>	是	<code>assemble</code> 脚本会基于来源构建应用，再将应用放入镜像中的相应目录。
<code>run</code>	是	<code>run</code> 脚本会执行您的应用。建议在 <code>run</code> 脚本中运行任何容器进程时使用 <code>exec</code> 命令。这可确保信号传播以及由 <code>run</code> 脚本启动的任何进程的正常关闭。
<code>save-artifacts</code>	否	<code>save-artifacts</code> 脚本会收集应用所需的所有依赖关系，并将这些依赖关系保存到 tar 文件中以加快后续构建的执行速度。例如，Ruby 中由 Bundler 安装的 gems，或者 Java 的 .m2 内容。这意味着构建不必重新下载这些内容，从而可以缩短构建时间。这些依赖关系被收集到一个 tar 文件中，并流化到标准输出。
<code>usage</code>	否	<code>usage</code> 脚本可以提供有关如何正确使用镜像的说明。

脚本	必需?	描述
test/run	否	通过 <code>test/run</code> 脚本，您可以创建简单的流程来验证镜像是否正常工作。

## S2I 构建工作流

S2I 构建流程如下所示：

1. RHOCP 会将基于 S2I 构建器镜像的容器实例化，然后创建一个包含源代码和 S2I 脚本的 tar 文件。接着，RHOCP 会将这个 tar 文件流化到容器中。
2. 在运行 `assemble` 脚本之前，RHOCP 会先提取上一步创建的 tar 文件，并将内容保存到 `--destination` 选项或构建器镜像的 `io.openshift.s2i.destination` 标签所指定的位置。默认位置为 `/tmp` 目录。
3. 如果正在进行的是逐步构建，则 `assemble` 脚本会恢复之前由 `save-artifacts` 脚本保存在 tar 文件中的构建构件。
4. `assemble` 脚本会基于来源构建应用，再将二进制文件放入相应的目录。
5. 如果正在进行的是逐步构建，则 `save-artifacts` 脚本会执行，并将存在依赖关系的所有构建构件保存到 tar 文件中。
6. 在 `assemble` 脚本完成运行后，容器会被提交以创建最终镜像，且 RHOCP 会将 `run` 脚本设置为最终镜像的 `CMD` 指令。

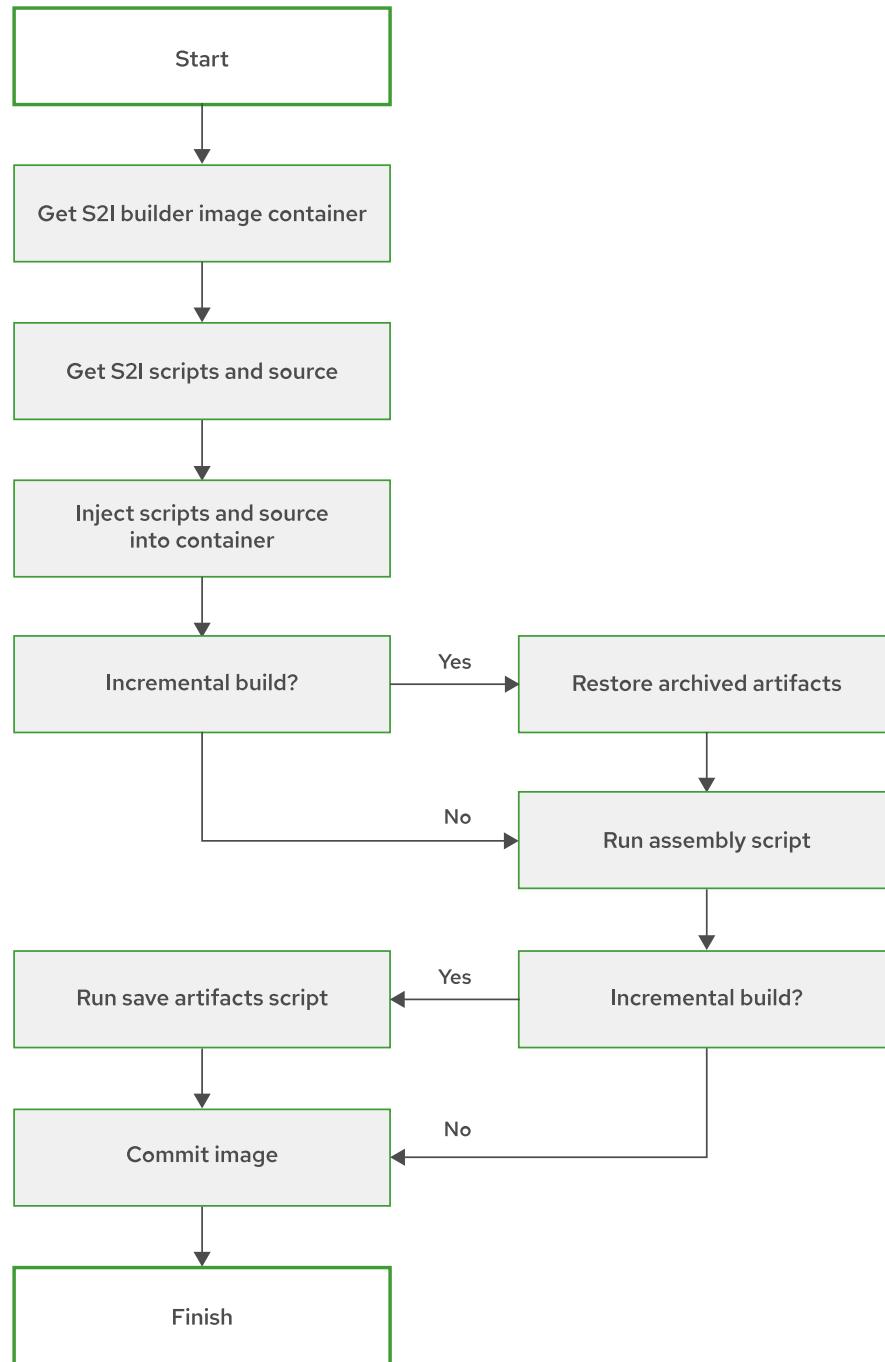


图 5.1: S2I 构建工作流

要创建构建器镜像，请使用创建应用所需的工具（如编译器、构建工具和前面提到的所有脚本文件）声明 `Dockerfile` 文件。以下 `Dockerfile` 构建器文件定义了 NGINX 服务器构建器：

```

FROM registry.access.redhat.com/ubi8/ubi:8.0

LABEL io.k8s.description="My custom Builder" \
      io.k8s.display-name="Nginx 1.6.3" \
      io.openshift.expose-services="8080:http" \
      io.openshift.tags="builder,webserver,html,nginx" \
  
```

## 章 5 | 自定义源至镜像构建

```
io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" 2

RUN yum install -y epel-release && \
    yum install -y --nodocs nginx && \
    yum clean all

EXPOSE 8080 4

COPY ./s2i/bin/ /usr/libexec/s2i 5
```

- 1** 设置用于 RHOCP 的标签来描述构建器镜像。
- 2** 告诉 S2I 在哪里可以找到其必需的脚本（**run**、**assemble**）。
- 3** 安装 NGINX Web 服务器软件包并清理 Yum 缓存。
- 4** 设置使用此镜像生成的应用的默认端口。
- 5** 将 S2I 脚本复制到 **/usr/libexec/s2i** 目录。

汇编脚本可以定义如下：

```
#!/bin/bash -e

if [ "$(ls -A /tmp/src)" ]; then
    mv /tmp/src/* /usr/share/nginx/html/ 1
fi
```

- 1** 覆盖默认 NGINX **index.html** 文件。

运行脚本可以定义如下：

```
#!/bin/bash -e
/usr/sbin/nginx -g "daemon off;" 1
```

- 1** 防止 NGINX 进程作为守护进程运行，以便容器在进程运行 **exec** 脚本后不会退出。

## 覆盖 S2I 构建器脚本

S2I 构建器镜像会提供默认的 S2I 脚本。您可以覆盖这个默认 S2I 脚本，以更改应用的构建和执行方式。您无需通过为原始 S2I 构建器派生源代码来创建新的 S2I 构建器镜像，即可覆盖默认构建行为。

要覆盖应用的默认 S2I 脚本，最简单的方式就是将您的 S2I 脚本纳入到应用的源代码存储库中。您可以在应用源代码存储库的 **.s2i/bin** 文件夹中提供 S2I 脚本。

当 RHOCP 启动 S2I 进程时，它将检查源代码文件夹、自定义 S2I 脚本和应用源代码。RHOCP 包括注入到 S2I 构建器镜像中的 tar 文件中的所有这些文件。RHOCP 随后便会执行自定义 **assemble** 脚本（而不是同 S2I 构建器一并纳入的默认 **assemble** 脚本），接着还会执行其他覆盖自定义脚本（若存在）。



## 参考文献

### 如何覆盖 S2I 构建器脚本

<https://blog.openshift.com/override-s2i-builder-scripts/>

### 如何创建 S2I 构建器镜像

<https://blog.openshift.com/create-s2i-builder-image/>

### 源至镜像 (S2I) 工具

<https://github.com/openshift/source-to-image>

### s2i 命令行界面

<https://github.com/openshift/source-to-image/blob/master/docs/cli.md>

如需有关标准 RHOCP S2I 构建器镜像的构建环境变量的更多信息，请参阅 RHOCP 4.10 文档中的 Images 指南，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/)

## ► 小测验

# 介绍源至镜像架构

选择以下问题的正确答案：

► 1. 以下哪个 S2I 脚本负责在 S2I 构建中构建应用二进制文件？

- a. `run`
- b. `assemble`
- c. `save-artifacts`
- d. `test/run`

► 2. 关于 S2I 构建流程，以下哪两项陈述是正确的？（请选择两项。）

- a. 会先执行 `assemble` 脚本，然后再提取包含应用源代码和 S2I 脚本的 tar 文件。
- b. 会先提取包含应用源代码和 S2I 脚本的 tar 文件，然后再执行 `assemble` 脚本。
- c. 会先提取包含应用源代码和 S2I 脚本的 tar 文件，然后再执行 `run` 脚本。
- d. `run` 脚本会被设置为最终容器镜像的 CMD 指令。
- e. `assemble` 脚本会被设置为最终容器镜像的 CMD 指令。

► 3. 您将在以下哪个目录中（相对于源代码的根目录）中提供您自己的自定义 S2I 脚本？

- a. `.scripts/bin`
- b. `.s2i/bin`
- c. `etc/bin`
- d. `usr/local/bin`

► 4. S2I 构建必须使用以下哪两个脚本？（请选择两项。）

- a. `usage`
- b. `test/run`
- c. `assemble`
- d. `run`
- e. `save-artifacts`

► 5. 对于逐步 S2I 构建而言，以下哪两项是可行的候选方案？（请选择两项。）

- a. 存在大量 JAR 依赖项的 Java EE 应用，这些依赖项由 Apache Maven 来管理。
- b. 基于 SQL 数据库转储文件的应用，该文件会在应用启动时被调用。
- c. 拥有大量静态资产（如镜像、CSS 和 HTML 文件）的 Ruby web 应用。
- d. 存在依赖关系的 Node.js 应用，这些依赖关系由 `npm` 来管理。

► 6. 以下哪个标签向 S2I 构建器镜像指示存储脚本的目录?

- a. s2i-url (openshift)
- b. s2i-dir openshift
- c. s2i-openshift-directory
- d. s2i-URL (openshift)

## ► 解决方案

# 介绍源至镜像架构

选择以下问题的正确答案：

► 1. 以下哪个 S2I 脚本负责在 S2I 构建中构建应用二进制文件？

- a. run
- b. **assemble**
- c. save-artifacts
- d. test/run

► 2. 关于 S2I 构建流程，以下哪两项陈述是正确的？（请选择两项。）

- a. 会先执行 assemble 脚本，然后再提取包含应用源代码和 S2I 脚本的 tar 文件。
- b. 会先提取包含应用源代码和 S2I 脚本的 tar 文件，然后再执行 **assemble** 脚本。
- c. 会先提取包含应用源代码和 S2I 脚本的 tar 文件，然后再执行 run 脚本。
- d. run 脚本会被设置为最终容器镜像的 CMD 指令。
- e. assemble 脚本会被设置为最终容器镜像的 CMD 指令。

► 3. 您将在以下哪个目录中（相对于源代码的根目录）中提供您自己的自定义 S2I 脚本？

- a. .scripts/bin
- b. .s2i/bin
- c. etc/bin
- d. usr/local/bin

► 4. S2I 构建必须使用以下哪两个脚本？（请选择两项。）

- a. usage
- b. test/run
- c. **assemble**
- d. **run**
- e. save-artifacts

► 5. 对于逐步 S2I 构建而言，以下哪两项是可行的候选方案？（请选择两项。）

- a. 存在大量 JAR 依赖项的 Java EE 应用，这些依赖项由 Apache Maven 来管理。
- b. 基于 SQL 数据库转储文件的应用，该文件会在应用启动时被调用。
- c. 拥有大量静态资产（如镜像、CSS 和 HTML 文件）的 Ruby web 应用。
- d. 存在依赖关系的 Node.js 应用，这些依赖关系由 npm 来管理。

► 6. 以下哪个标签向 S2I 构建器镜像指示存储脚本的目录?

- a. s2i-url (openshift)
- b. s2i-dir openshift
- c. s2i-openshift-directory
- d. s2i-URL (openshift)

# 自定义现有 S2I 基础镜像

## 培训目标

学完本节后，您应能够自定义现有 S2I 构建器镜像的 S2I 脚本。

## 自定义 S2I 构建器镜像的脚本

在默认情况下，S2I 脚本会封装在 S2I 构建器镜像中。在某些情况下，您可能会想要自定义这些脚本以更改应用的构建和运行方式，但不重新构建镜像。

S2I 构建流程提供了一种可覆盖默认 S2I 脚本的方法。您可以在应用源代码的 `.s2i/bin` 文件夹中提供自己的 S2I 脚本。构建流程会自动检测并运行的自定义 S2I 脚本，而不是封装在构建器镜像中的默认 S2I 脚本。

根据要对所覆盖脚本执行的自定义数量，您可以选择使用自己的版本彻底替代默认 S2I 脚本。或者，您可以创建调用默认脚本的 wrapper 脚本，然后在调用之前或之后添加必要的自定义。

例如，如果想为 `rhscl/php-73-rhel7` S2I 构建器镜像自定义 S2I 脚本，并更改应用的构建和运行方式，请按照以下步骤操作。可以使用以下过程自定义此构建器镜像中提供的 S2I 脚本：

- 使用 `podman pull` 命令，将容器镜像从容器注册表拉入本地系统。使用 `podman inspect` 命令获取 `io.openshift.s2i.scripts-url` 标签的值，以确定镜像中 S2I 脚本的默认位置。

```
[user@host ~]$ podman pull \
myregistry.example.com/rhscl/php-73-rhel7
...output omitted...
Digest: sha256:...
[user@host ~]$ podman inspect \
--format='{{ index .Config.Labels "io.openshift.s2i.scripts-url"}}' \
rhscl/php-73-rhel7
image:///usr/libexec/s2i
```

- 您还可以使用 `skopeo inspect` 命令，直接从远程注册表中检索相同的信息：

```
[user@host ~]$ skopeo inspect \
docker://myregistry.example.com/rhscl/php-73-rhel7 \
| grep io.openshift.s2i.scripts-url
"io.openshift.s2i.scripts-url": "image:///usr/libexec/s2i",
```

- 在 `.s2i/bin` 文件夹中为 `assemble` 脚本创建打包程序：

```
#!/bin/bash
echo "Making pre-invocation changes..."

/usr/libexec/s2i/assemble
rc=$?

if [ $rc -eq 0 ]; then
```

```
echo "Making post-invocation changes..."  
else  
    echo "Error: assemble failed!"  
fi  
  
exit $rc
```

- 同样地，在 `.s2i/bin` 文件夹中为 `run` 脚本创建打包程序：

```
#!/bin/bash  
echo "Before calling run..."  
exec /usr/libexec/s2i/run
```



### 注意

在打包 `run` 脚本时，您必须使用 `exec` 来调用该脚本。这可以确保默认 `run` 脚本仍以进程 ID “1” 来运行。如果不这样，便会导致关机期间出现信号传播错误，而且您的应用可能会无法正常工作。这也意味着，在调用默认 `run` 脚本后，您便无法运行打包程序脚本中的其他命令。

## S2I 中的渐进构建

在红帽 OpenShift (RHOCP) 集群上使用 S2I 构建应用时，针对应用构建、部署并测试小幅逐步更改是常见的做法。某些企业还会采用持续集成 (CI) 和持续交付 (CD) 技术，即以快速迭代周期多次构建并部署应用，通常无需任何人工干预。

在使用多个依赖组件和库以模块化方式构建应用时，容器不可变的本质会导致 S2I 构建需要很长长时间才能完成。构建流程必须获取这些依赖关系，而且每次源代码发生变化时都要构建并部署应用。

S2I 构建流程提供了一种机制，可通过在 S2I 生命周期内依次调用 `assemble` 和 `save-artifacts` 脚本来缩短构建时间。`save-artifacts` 脚本可确保保存在依赖关系的构件（应用所需的库和组件）已保存，以供日后构建使用。

下次构建时，`assemble` 脚本会先恢复缓存的构件，然后从源代码构建应用。请注意，`save-artifacts` 脚本负责将 `stdout` 的依赖关系流化到 `tar` 文件中。



### 重要

`save-artifacts` 脚本输出应该只包含 `tar` 流化输出，没有其他内容。您应该将该脚本中的其他命令的输出重定向至 `/dev/null`。

例如，假设您正使用由 Apache Maven 管理的多个依赖项来开发基于 Java EE 的应用。如果您已经构建了 S2I 构建器镜像以使 `assemble` 脚本编译和封装应用 JAR 文件，那么可以重复使用先前下载的 JAR 文件的逐步构建可以大幅缩短构建时间。Apache Maven 会将 JAR 依赖关系存储在 `$HOME/.m2` 文件夹中。

用于缓存 Maven JAR 文件的 `save-artifacts` 脚本可以按照如下方式定义：

```
#!/bin/sh -e

# Stream the .m2 folder tar archive to stdout
if [ -d ${HOME}/.m2 ]; then
    pushd ${HOME} > /dev/null
    tar cf - .m2
    popd > /dev/null
fi
```

可在 **assemble** 脚本中进行构建前先恢复构件的对应代码：

```
# Restore the .m2 folder
...output omitted...
if [ -d /tmp/artifacts/.m2 ]; then
    echo "---> Restoring maven artifacts..."
    mv /tmp/artifacts/.m2 ${HOME}/
fi
...output omitted...
```



## 参考文献

如需更多信息，请参阅 RHOCP 4.10 Images 指南的 Creating Images 章节，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/index#creating-images](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/index#creating-images)

## 如何覆盖 S2I 构建器脚本

<https://blog.openshift.com/override-s2i-builder-scripts>

## ► 指导练习

# 自定义 S2I 构建

在本练习中，您将自定义现有 S2I 构建器镜像的 S2I 脚本，以向应用添加信息页面。

## 成果

您应该能够自定义 Apache HTTP 服务器构建器镜像的 `assemble` 和 `run` 脚本。您将使用自己的自定义版本来覆盖默认的内置脚本。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift (RHOCP) 集群。
- `rhscl/httpd-24-rhel7` Apache HTTP 服务器 S2I 构建器镜像。
- 包含 `s2i-scripts` 应用源代码的 Git 存储库分叉。

在 `workstation` 虚拟机上运行以下命令，以验证练习前提条件并下载实验和答案文件：

```
[student@workstation ~]$ lab s2i-scripts start
```

## 说明

### ► 1. 探索封装在 `rhscl/httpd-24-rhel7` 构建器镜像中的 S2I 脚本。

- 在 `workstation` 虚拟机上，从终端窗口运行 `rhscl/httpd-24-rhel7` 镜像，然后覆盖容器入口点以运行 shell：

```
[student@workstation ~]$ podman run --name test -it rhscl/httpd-24-rhel7 bash
Trying to pull registry.access.redhat.com/rhscl/httpd-24-rhel7:latest...
Getting image source signatures
...output omitted...
bash-4.2$
```

- 检查封装在构建器镜像中的 S2I 脚本。这些 S2I 脚本位于 `/usr/libexec/s2i` 文件夹中：

```
bash-4.2$ cat /usr/libexec/s2i/assemble
...output omitted...
bash-4.2$ cat /usr/libexec/s2i/run
...output omitted...
bash-4.2$ cat /usr/libexec/s2i/usage
...output omitted...
```

从容器中退出：

```
bash-4.2$ exit
```

▶ 2. 查看包含自定义 S2I 脚本的应用源代码。

- 2.1. 进入 **D0288-apps** Git 存储库的本地克隆，并签出课程存储库的 **main** 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 2.2. 检查 **/home/student/D0288-apps/s2i-scripts/index.html** 文件。

该 HTML 文件包含一条简单的消息：

```
Hello Class! D0288 rocks!!!
```

- 2.3. 自定义 S2I 脚本位于 **/home/student/D0288-apps/s2i-scripts/.s2i/bin** 文件夹中。**.s2i/bin/assemble** 脚本会将应用源中的 **index.html** 文件复制到位于 **/opt/app-root/src** 的 Web 服务器文档中。它还会创建一个包含页面构建时间和环境信息的 **info.html** 文件：

```
...output omitted...
CUSTOMIZATION STARTS HERE

echo "----> Installing application source"
cp -Rf /tmp/src/*.html ./

DATE=date "+%b %d, %Y @ %H:%M %p"

echo "----> Creating info page"
echo "Page built on $DATE" >> ./info.html
echo "Proudly served by Apache HTTP Server version $HTTPD_VERSION" >> ./info.html

CUSTOMIZATION ENDS HERE
...output omitted...
```

- 2.4. **.s2i/bin/run** 脚本会将 Web 服务器中的启动消息默认日志级别更改为 **debug**：

```
# Make Apache show 'debug' level logs during startup
exec run-httpd -e debug $@
```

▶ 3. 将应用部署到 RHOCP 集群。验证执行的是否是自定义 S2I 脚本。

- 3.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

- 3.2. 使用您的开发人员用户帐户登录 RHOCP：

## 章 5 | 自定义源至镜像构建

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

3.3. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation D0288-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-s2i-scripts
Now using project "youruser-s2i-scripts" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

3.4. 基于 Git 中的来源创建一个名为 **bonjour** 新应用。您需要使用波形符表示法 (~)，将 **httpd:2.4** 镜像流作为前缀添加到 Git URL 中，以确保应用使用 **rhscl/httpd24-rhel7** 构建器镜像。

```
[student@workstation D0288-apps]$ oc new-app \
--name bonjour \
httpd:2.4~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
--context-dir s2i-scripts
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "bonjour" created
buildconfig.build.openshift.io "bonjour" created
deployment.apps "bonjour" created
service "bonjour" created
--> Success
...output omitted...
```

3.5. 查看构建日志。等待构建完成并将应用容器镜像推送到 RHOCP 注册表：

```
[student@workstation D0288-apps]$ oc logs -f bc/bonjour
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Enabling s2i support in httpd24 image
AllowOverride All
--> Installing application source
--> Creating info page
Pushing image image-registry.openshift-image-registry.svc:5000/youruser-s2i-
scripts/bonjour:latest ... ...
...output omitted...
Push successful
```

观察执行的是否是应用提供的自定义 S2I 脚本，而不是构建器镜像的内置 S2I 脚本。

#### ▶ 4. 测试应用。

4.1. 等待应用部署好，然后查看应用容器集的状态。应用容器集应处于 **Running** 状态：

## 章 5 | 自定义源至镜像构建

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
bonjour-1-build   0/1     Completed   0          105s
bonjour-7bc86dd97-wjvhx  1/1     Running    0          72s
```

4.2. 使用路由公开应用供外部访问。

```
[student@workstation D0288-apps]$ oc expose svc bonjour
route.route.openshift.io/bonjour exposed
```

4.3. 使用 `oc get route` 命令获取路由 URL:

```
[student@workstation D0288-apps]$ oc get route
NAME      HOST/PORT
bonjour   bonjour-youruser-s2i-scripts.apps.cluster.domain.example.com ...
```

4.4. 使用 `curl` 命令以及之前命令中的路由 URL 调用应用的索引页面:

```
[student@workstation D0288-apps]$ curl \
http://bonjour-${RHT_OCP4_DEV_USER}-s2i-scripts.${RHT_OCP4_WILDCARD_DOMAIN}
Hello Class! D0288 rocks!!!
```

您应该会看到应用源中的 `index.html` 文件的内容。

4.5. 使用 `curl` 命令调用应用的信息页面:

```
[student@workstation D0288-apps]$ curl \
http://bonjour-${RHT_OCP4_DEV_USER}-s2i-scripts.${RHT_OCP4_WILDCARD_DOMAIN} \
/info.html
Page built on Jun 11, 2021 @ 16:12 PM
Proudly served by Apache HTTP Server version 2.4
```

您应该会看到 `info.html` 文件的内容，以及与页面构建时间和 Apache HTTP 服务器版本有关的详细信息。

4.6. 检查应用容器集的日志。请记住，`run` 脚本中的启动日志级别已更改为 `debug`。您应该会在启动时看到 `debug` 级别的日志消息：

```
[student@workstation D0288-apps]$ oc logs deployment/bonjour
...output omitted...
[Fri Nov 03 16:12:21.690941 2021] [so:debug] [pid 9] mod_so.c(266): AH01575:
 loaded module systemd_module from /opt/rh/httpd24/root/etc/httpd/modules/
mod_systemd.so
[Fri Nov 03 16:12:21.691050 2021] [so:debug] [pid 9] mod_so.c(266): AH01575:
 loaded module cgi_module from /opt/rh/httpd24/root/etc/httpd/modules/mod_cgi.so
...output omitted...
[Fri Nov 03 16:12:21.742471 2021] [ssl:debug] [pid 9] ssl_engine_init.c(270):
 AH01886: SSL FIPS mode disabled
...output omitted...
[Fri Nov 03 16:12:21.745520 2021] [mpm_prefork:notice] [pid 9] AH00163:
 Apache/2.4.25 (RedHat) OpenSSL/1.0.1e-fips configured -- resuming normal
operations
```

## 章 5 | 自定义源至镜像构建

```
[Fri Nov 03 16:12:21.745530 2021] [core:notice] [pid 9] AH00094: Command line:  
'httpd -D FOREGROUND -e debug'  
...output omitted...  
10.131.0.16 - - [03/Nov/2021:16:28:53 +0000] "GET / HTTP/1.1" 200 28 "-"  
"curl/7.29.0"  
10.128.2.216 - - [03/Nov/2021:16:29:03 +0000] "GET /info.html HTTP/1.1" 200 87 "-"  
"curl/7.29.0"
```

### ▶ 5. 清理。删除项目：

```
[student@workstation D0288-apps]$ cd ~  
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-s2i-scripts  
project.project.openshift.io "youruser-redhat-com-s2i-scripts" deleted
```

### ▶ 6. 删除先前为了查看默认 S2I 脚本而创建的 test 容器。

```
[student@workstation ~]$ podman rm test  
925b932059df9e327bffffe1964c7a1a5a45d13d872b2fb67e333b63166923ce3
```

您的 ID 值将与上述不同。

## 完成

在 workstation 虚拟机上，执行 `lab s2i-scripts finish` 脚本以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab s2i-scripts finish  
  
Completing Guided Exercise: Customizing S2I Builds  
  
· Log in on OpenShift..... SUCCESS  
· Git repo '/home/student/D0288-apps' has no pending changes.. SUCCESS  
  
Please use start if you wish to do the exercise again.
```

本引导式练习到此结束。

# 创建 S2I 基础镜像

## 培训目标

学习完本节后，您应能够使用 `s2i` 命令行工具创建 S2I 构建器镜像。

## 构建并发布自定义 S2I 构建器镜像

S2I 构建流程会将应用源代码与相应的 S2I 构建器镜像整合，以生成部署在红帽 OpenShift 容器平台 (RHOCP) 集群上的最终应用容器镜像。

在将 S2I 构建器镜像部署到其他开发人员可用来构建应用的 RHOCP 集群上之前，使用 `s2i` 命令行工具来构建和测试镜像至关重要。在本地计算机上安装 `s2i` 工具，以在 RHOCP 集群之外生成和测试 S2I 构建器镜像。

## 安装 S2I 工具

从 RHEL 7 开始，`s2i` 工具位于 `source-to-image` 软件包中并可使用 Yum 来安装。请确保您的系统已订阅并启用 `rhel-server-rhscl-7-rpms` 或 `rhel-server-rhscl-8-rpms` 存储库，具体取决于您的 RHEL 版本。

对于其他操作系统，`s2i` 工具可从上游 `source-to-image` 项目页面下载，网址为：<https://github.com/openshift/source-to-image/releases>

## 使用 S2I 工具

请使用 `s2i create` 命令来创建所需的模板文件，以创建新的 S2I 构建器镜像：

```
[user@host ~]$ s2i create image_name directory
```

以上命令会创建一个名为 `directory` 的文件夹，并在其中填充您可以根据需要更新的以下模板文件：

```
directory
├── Dockerfile ①
├── Makefile
├── README.md
└── s2i ②
    └── bin
        ├── assemble
        ├── run
        ├── save-artifacts
        └── usage
    └── test
        └── run
            └── test-app ③
                └── index.html
```

① S2I 构建器镜像的 Dockerfile

- ② S2I 脚本目录
- ③ 用于复制应用源代码以进行本地测试的文件夹

**s2i create** 命令会创建一个带有注释的模板 Dockerfile，以使用随机用户 ID 和特定于 OpenShift 的标签在 RHOCP 集群上运行。它还会为 S2I 脚本创建存根，您可以自定义这些存根以满足您的应用需求。在根据需要更新 Dockerfile 和 S2I 脚本后，您可以使用 **podman build** 命令来构建构建器镜像。

```
[user@host ~]$ podman build -t builder_image .
```

待构建器镜像准备就绪后，您可以使用 **s2i build** 命令来构建应用容器镜像。这样便可在本地测试 S2I 构建器镜像，而无需将其推送至注册表服务器并使用构建器镜像将应用部署到 RHOCP 集群：

```
[user@host ~]$ s2i build src builder_image tag_name
```



### 注意

如果包含不属于 OCI 规范的任何指令（如构建器镜像 Dockerfile 文件中的 **ONBUILD** 指令），请确保在构建 S2I 构建器镜像时将 **--format docker** 选项与 **podman build** 命令一起使用。此选项将覆盖 Podman 使用的默认 **oci** 格式。

**s2i build** 命令会将 **src** 选项中定义的应用源代码与 **builder\_image** 容器镜像进行整合，以生成带有标签 **tag\_name** 的应用容器镜像。此命令通过自动将提供的源代码注入构建器镜像来模拟 RHOCP 集群使用的 S2I 进程，但它使用本地 Docker 服务构建测试容器镜像，而不是将镜像部署到 RHOCP 集群。

通过使用 **podman run** 命令运行镜像来测试 **s2i build** 命令生成的应用容器镜像。请注意，如果该应用容器镜像将被部署至 RHOCP，则您需为 **podman run** 命令使用 **-u** 标记并利用随机用户 ID 来模拟运行容器。



### 重要

**s2i build** 命令需要使用本地 Docker 服务，因为它直接通过套接字使用 Docker API 来构建 S2I 容器镜像。RHEL 8 和 RHOCP 4 环境中不包括 Docker，且这不起作用。

为了支持没有 Docker 可用的环境，**s2i build** 命令现在包括 **--as-dockerfile path/to/Dockerfile** 选项。此选项将 **s2i build** 命令配置为生成 Dockerfile 和两个支持目录，供您使用 **podman build** 命令从源存储库和构建器镜像构建测试容器镜像。通过使用此选项，无需本地 Docker 守护进程即可运行 **s2i build** 命令。

您可以提供包含应用来源的目录位置或 Git 存储库 URL 作为 **src** 选项。

对于渐进式构建，务必要创建 **save-artifacts** 脚本，并向 **s2i build** 命令传输 **--incremental** 标记。如果 **save-artifacts** 脚本存在，以前的镜像也已存在，并且您使用 **--incremental=true** 选项，则工作流如下所示：

1. S2I 从以前的构建镜像创建新的容器镜像。
2. S2I 在此容器中运行 **save-artifacts** 脚本。此脚本负责将工件的 tar 文件流化到 stdout。
3. S2I 构建新的输出镜像：

## 章 5 | 自定义源至镜像构建

- 上一个构建中的工件位于传递到该构建的 tar 文件的 **artifacts** 目录中。
- S2I 构建器镜像的 **assemble** 脚本负责检测和使用构建工件。

请运行 **s2i --help** 和 **s2i subcommand --help** 命令，以了解各个子命令及其选项和使用示例。

在完成应用容器镜像的本地测试后，您可以复制 S2I 构建器镜像，以供注册表使用。在使用构建器镜像在 RHOCP 集群上部署应用之前，您必须使用 **oc import-image** 命令基于构建器镜像创建镜像流。然后，您可以使用该镜像流在 RHOCP 中创建应用。

## 构建 Nginx S2I 构建器镜像

要基于 RHEL 8 创建 Nginx S2I 构建器镜像，请执行以下步骤：

### 创建并填充 Dockerfile 项目

使用 **s2i** 命令创建 S2I 构建器镜像项目目录，并根据需要编辑文件：

- 运行 **s2i create** 命令，以便为 S2I 构建器镜像构件创建框架目录结构：

```
[user@host ~]$ s2i create s2i-do288-nginx s2i-do288-nginx
```

- 编辑 Dockerfile 以纳入用于安装 Nginx 和 S2I 脚本的指令。Nginx S2I 构建器镜像的以下 Dockerfile 会使用 RHEL 8 基础镜像：

```
FROM registry.access.redhat.com/ubi8/ubi:8.0 1

ENV X_SCLS="rh-nginx18" \
    PATH="/opt/rh/rh-nginx18/root/usr/sbin:$PATH" \
    NGINX_DOCROOT="/opt/rh/rh-nginx18/root/usr/share/nginx/html"

LABEL io.k8s.description="A Nginx S2I builder image" 2 \
      io.k8s.display-name="Nginx 1.8 S2I builder image for DO288" \
      io.openshift.expose-services="8080:http" \
      io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" \
      io.openshift.tags="builder,webserver,nginx,nginx18,html"

ADD nginxconf.sed /tmp/
COPY ./s2i/bin/ /usr/libexec/s2i 3

RUN yum install -y --nodoscs rh-nginx18 4 \
    && yum clean all \
    && sed -i -f /tmp/nginxconf.sed /etc/opt/rh/rh-nginx18/nginx/nginx.conf \
    && chgrp -R 0 /var/opt/rh/rh-nginx18 /opt/rh/rh-nginx18 5 \
    && chmod -R g=u /var/opt/rh/rh-nginx18 /opt/rh/rh-nginx18 6 \
    && echo 'Hello from the Nginx S2I builder image' > ${NGINX_DOCROOT}/index.html

EXPOSE 8080

USER 1001

CMD ["/usr/libexec/s2i/usage"]
```

## 章 5 | 自定义源至镜像构建

- ① 使用 RHEL 8 通用基础镜像作为 S2I 构建器镜像的基础。
  - ② S2I 构建器镜像使用者的标签元数据。
  - ③ 将 S2I 脚本复制到 `io.openshift.s2i.scripts-url` 标签所示的位置。
  - ④ 安装 Nginx。
  - ⑤ ⑥ 设置权限，以作为随机用户 ID 在 RHOCP 集群上运行。
- 在包含以下内容的应用来源的 `.s2i/bin` 目录中创建 `assemble` 脚本，该脚本会将 HTML 源文件复制到 Nginx web 服务器文档根目录中：

```
#!/bin/bash -e

echo "---> Copying source HTML files to web server root..."
cp -Rf /tmp/src/. /opt/rh/rh-nginx18/root/usr/share/nginx/html/
```

- 在包含以下内容的应用来源的 `.s2i/bin` 目录中创建 `run` 脚本，该脚本会在前台运行 Nginx web 服务器：

```
#!/bin/bash -e

exec nginx -g "daemon off;"
```

## 构建并测试 S2I 构建器镜像

使用 Podman 和 `s2i` 命令来构建 S2I 构建器镜像和测试应用容器镜像：

- 构建 S2I 构建器镜像：

```
[user@host ~]$ podman build -t s2i-do288-nginx .
```

- 运行 `s2i build` 命令以构建测试应用容器镜像。要覆盖构建器镜像中包含的默认 `index.html` 文件，请在 `test/test-app` 目录下创建一个 `index.html` 文件，以将此新文件注入测试 Nginx 容器：

```
[user@host ~]$ s2i build test/test-app s2i-do288-nginx nginx-test \
--as-docker-file /path/to/Dockerfile
```

- 要生成测试容器，请使用 `podman build` 命令，这一次使用由 `s2i build` 命令生成的 Dockerfile：

```
[user@host ~]$ podman build -t nginx-test /path/to/Dockerfile
```

- 要测试容器镜像，请使用 `podman run` 命令以及不同于 Dockerfile 所提供 ID 的用户 ID，以确保可在 RHOCP 集群上使用随机生成的用户 ID 来运行容器：

```
[user@host ~]$ podman run -u 1234 -d -p 8080:8080 nginx-test
```

## 章 5 | 自定义源至镜像构建

在容器完成运行后，如果没有返回任何错误，则请验证在测试 Nginx 容器时，**test** 目录中有没有呈现您所提供的测试 **index.html** 文件。

## 使 S2I 构建器镜像可供 RHOCP 使用

使用 **skopeo** 命令将 S2I 构建器镜像推送至容器注册表，并在 RHOCP 中创建指向该镜像的镜像流。

- 在完成容器的本地测试后，请将 S2I 构建器镜像推送至企业注册表。例如，假设您有一个 Quay.io 帐户，并且已使用 **podman login** 命令登录：

```
[user@host ~]$ skopeo copy containers-storage:localhost/s2i-do288-httppd \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httppd
```

- 要为 Nginx S2I 构建器镜像创建镜像流，请创建一个新项目，然后运行 **oc import-image** 命令：

```
[user@host ~]$ oc import-image s2i-do288-nginx \
--from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-nginx \
--confirm
```



### 注意

回想一下第 3 章 发布企业容器镜像，如果 S2I 构建器镜像不公开，则可能需要为发布该镜像的远程注册表创建包含凭据的拉取机密。您还需要将该机密链接到用于拉取镜像的默认服务帐户，以便使用 **oc import-image** 命令创建镜像流。

- 镜像流创建好后，您便可用它和 Nginx S2I 构建器镜像来创建应用。请注意，除非使用 S2I 构建器镜像来替代 RHOCP **oc new-app** 命令所支持的语言，否则您必须使用波形符表示法 (~)：

```
[user@host ~]$ oc new-app --name nginx-test \
s2i-do288-nginx~git_repository
```



### 参考文献

如需更多信息，请参阅 RedHat RHOCP 4.10 Images 指南的 Creating Images 章节，网址为：

[https://access.redhat.com/documentation/en-us/red\\_hat\\_openshift\\_container\\_platform/4.10/html-single/images/creating-images#creating-images](https://access.redhat.com/documentation/en-us/red_hat_openshift_container_platform/4.10/html-single/images/creating-images#creating-images)

#### 如何创建 S2I 构建器镜像

<https://blog.openshift.com/create-s2i-builder-image>

#### 源至镜像 (S2I) 工具

<https://github.com/openshift/source-to-image>

#### s2i 工具子命令参考

<https://github.com/openshift/source-to-image/blob/master/docs/cli.md>

## ▶ 指导练习

# 创建 S2I 基础镜像

在本练习中，您将构建并测试 Apache HTTP 服务器 S2I 构建器镜像，然后使用该镜像构建并部署应用。

## 成果

您应能够：

- 使用 **s2i** 工具构建 Apache HTTP 服务器 S2I 构建器镜像。
- 使用一个简单应用在本地测试 S2I 构建器镜像。
- 将构建器镜像发布至容器注册表。
- 使用 S2I 构建器镜像在红帽 OpenShift 容器平台 (RHOCP) 集群上部署并测试应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的 RHOCP 集群。
- 示例应用的父镜像 (**ubi8/ubi**)。
- DO288-apps Git 存储库中的应用 (**html-helloworld**)。

在 **workstation** 虚拟机上运行以下命令，以验证练习前提条件并下载实验和答案文件：

```
[student@workstation ~]$ lab apache-s2i start
```

## 说明

- ▶ 1. 在 **workstation** 虚拟机上，验证已安装了 **source-to-image** 软件包，该软件包提供 **s2i** 命令行工具：

```
[student@workstation ~]$ s2i version  
s2i v1.3.1
```

- ▶ 2. 使用 **s2i** 命令创建 S2I 构建器镜像所需的模板文件和目录。

- 2.1. 在 **workstation** 虚拟机上，使用 **s2i create** 命令为 **/home/student/** 目录中的构建器镜像创建模板文件：

```
[student@workstation ~]$ s2i create s2i-do288-httpd s2i-do288-httpd
```

- 2.2. 验证模板文件是否已创建好。**s2i create** 命令会创建以下目录结构：

```
[student@workstation ~]$ tree s2i-do288-httdp
s2i-do288-httdp
├── Dockerfile
├── Makefile
├── README.md
└── s2i
    ├── bin
    │   ├── assemble
    │   ├── run
    │   ├── save-artifacts
    │   └── usage
    └── test
        └── run
            └── test-app
                └── index.html
4 directories, 9 files
```



### 注意

s2i命令会生成一个Containerfile，其旧名称为Dockerfile。Containerfile是首选的名称。

## ▶ 3. 创建Apache HTTP 服务器 S2I 构建器镜像。

3.1. Apache HTTP 服务器构建器镜像的示例 Containerfile 位于 ~/D0288/labs/apache-s2i/Containerfile 中。粗略检查此文件：

```
[student@workstation ~]$ cat ~/D0288/labs/apache-s2i/Containerfile
FROM registry.access.redhat.com/ubi8/ubi:8.4 ①

# Generic labels
LABEL Component="httdp" \
      Name="s2i-do288-httdp" \
      Version="1.0" \
      Release="1"

# Labels consumed by RHOC
LABEL io.k8s.description="A basic Apache HTTP Server S2I builder image" \
      io.k8s.display-name="Apache HTTP Server S2I builder image for D0288" \
      io.openshift.expose-services="8080:http" \
      io.openshift.s2i.scripts-url="image:///usr/libexec/s2i" ③

# This label is used to categorize this image as a builder image in the
# RHOC web console.
LABEL io.openshift.tags="builder, httdp, httdp24"

# Apache HTTP Server DocRoot
ENV DOCROOT /var/www/html

RUN yum install -y --nodos --disableplugin=subscription-manager httdp && \
    yum clean all --disableplugin=subscription-manager -y && \
    ④
```

## 章 5 | 自定义源至镜像构建

```

echo "This is the default index page from the s2i-do288-httpd S2I builder
image." > ${DOCROOT}/index.html 5

# Change web server port to 8080
RUN sed -i "s/Listen 80/Listen 8080/g" /etc/httpd/conf/httpd.conf

# Copy the S2I scripts to the default location indicated by the
# io.openshift.s2i.scripts-url LABEL (default is /usr/libexec/s2i)
COPY ./s2i/bin/ /usr/libexec/s2i 6
...output omitted...

```

- ①** 使用 RHEL 8 通用基础镜像作为此容器的基础镜像。
- ②** 设置用于 RHOCOP 的标签来描述构建器镜像。
- ③** 配置必需 S2I 脚本 (**run**、**assemble**) 的位置。
- ④** 安装 httpd Web 服务器软件包并清理 Yum 缓存。
- ⑤** 设置构建器镜像的默认 **index.html** 文件的内容。
- ⑥** 将 S2I 脚本复制到 **/usr/libexec/s2i** 目录。

然后，将此 Containerfile 复制到 **~/s2i-do288-httpd** 目录并覆盖生成的模板 Containerfile：

```
[student@workstation ~]$ rm ~/s2i-do288-httpd/Dockerfile
[student@workstation ~]$ cp ~/D0288/labs/apache-s2i/Containerfile \
~/s2i-do288-httpd/
```

- 3.2. 同样地，请在 **~/D0288/labs/apache-s2i/s2i/bin** 目录中查看为这个构建器镜像提供的 S2I 脚本，然后将其复制到 **~/s2i-do288-httpd/s2i/bin** 目录并覆盖生成的脚本：

```
[student@workstation ~]$ cp -Rv ~/D0288/labs/apache-s2i/s2i ~/s2i-do288-httpd/
'D0288/labs/apache-s2i/s2i/bin/assemble' -> 's2i-do288-httpd/s2i/bin/assemble'
'D0288/labs/apache-s2i/s2i/bin/usage' -> 's2i-do288-httpd/s2i/bin/usage'
'D0288/labs/apache-s2i/s2i/bin/run' -> 's2i-do288-httpd/s2i/bin/run'
```

- 3.3. 本练习不实施 **save-artifacts** 脚本。从 **~/s2i-do288-httpd/s2i/bin** 目录中目录它：

```
[student@workstation ~]$ rm -f ~/s2i-do288-httpd/s2i/bin/save-artifacts
```

- 3.4. 创建 Apache HTTP 服务器 S2I 构建器镜像。请不要忘记在 **podman build** 命令的末尾加上一个句点。这表示 Podman 应该使用当前目录中的 Containerfile 来构建镜像：

```
[student@workstation ~]$ cd s2i-do288-httpd
[student@workstation s2i-do288-httpd]$ podman build -t s2i-do288-httpd .
STEP 1: FROM registry.access.redhat.com/ubi8/ubi:8.4
...output omitted...
STEP 14: COMMIT s2i-do288-httpd
...output omitted...
```

3.5. 验证构建器镜像是否已创建好：

```
[student@workstation s2i-do288-httdp]$ podman images
REPOSITORY                      TAG      IMAGE ID      CREATED
localhost/s2i-do288-httdp        latest   82beb27428b7  9 seconds ago
registry.access.redhat.com/ubi8/ubi 8.4     7ae69d957d8b  2 weeks ago
...output omitted...
```

► 4. 构建并测试可将 Apache HTTP 服务器 S2I 构建器镜像与应用源代码进行整合的应用容器镜像。

4.1. 待构建器镜像准备就绪后，您可以使用 **s2i build** 命令来构建应用容器镜像。在构建应用容器镜像之前，请查看示例 `~/D0288/labs/apache-s2i/index.html` HTML 文件：

```
[student@workstation s2i-do288-httdp]$ cat ~/D0288/labs/apache-s2i/index.html
This is the index page from the app
```

然后，将此文件复制到 `~/s2i-do288-httdp/test/test-app` 目录以覆盖打包在构建器镜像中的 `index.html` 文件：

```
[student@workstation s2i-do288-httdp]$ cp ~/D0288/labs/apache-s2i/index.html \
~/s2i-do288-httdp/test/test-app/
```

4.2. 为 **s2i build** 命令生成的 Containerfile 创建一个新目录。

```
[student@workstation s2i-do288-httdp]$ mkdir ~/s2i-sample-app
```

4.3. 构建应用容器镜像：

```
[student@workstation s2i-do288-httdp]$ s2i build test/test-app/ \
s2i-do288-httdp s2i-sample-app \
--as-dockerfile ~/s2i-sample-app/Containerfile
Application dockerfile generated in /home/student/s2i-sample-app/Containerfile
```

4.4. 检查生成的应用目录。

```
[student@workstation s2i-do288-httdp]$ cd ~/s2i-sample-app
[student@workstation s2i-sample-app]$ tree .
.
├── Containerfile
├── downloads
│   ├── defaultScripts
│   └── scripts
└── upload
    ├── scripts
    └── src
        └── index.html

6 directories, 2 files
```

## 章 5 | 自定义源至镜像构建

观察位于 `upload/src` 目录中的 `index.html` 文件。这是您在上一步中复制到 `test/test-app` 目录中的相同 `index.html` 文件。

## 4.5. 检查生成的 Containerfile。

```
[student@workstation s2i-sample-app]$ cat Containerfile
FROM s2i-do288-httdp ①
LABEL "io.k8s.display-name=""s2i-sample-app" \
      "io.openshift.s2i.build.image""="s2i-do288-httdp" \
      "io.openshift.s2i.build.source-location""="test/test-app/"

USER root
# Copying in source code
COPY upload/src /tmp/src ③
# Change file ownership to the assemble user. Builder image must support chown
command.
RUN chown -R 1001:0 /tmp/src
USER 1001
# Assemble script sourced from builder image based on user input or image
metadata.
# If this file does not exist in the image, the build will fail.
RUN /usr/libexec/s2i/assemble
# Run script sourced from builder image based on user input or image metadata.
# If this file does not exist in the image, the build will fail.
CMD /usr/libexec/s2i/run
```

- ① 使用 `s2i-do288-httdp` 构建器镜像作为此容器镜像的父级。
- ② 设置用于 RHOCP 的标签来描述应用。
- ③ 复制包含在 `upload/src` 目录中的源代码中。

## 4.6. 从生成的 Containerfile 构建测试容器镜像。

```
[student@workstation s2i-sample-app]$ podman build \
-t s2i-sample-app .
STEP 1: FROM s2i-do288-httdp
STEP 2: LABEL "io.k8s.display-name=""s2i-sample-app"...
...output omitted...
STEP 9: COMMIT s2i-sample-app
...output omitted...
```

## 4.7. 验证应用容器镜像是否已创建好：

```
[student@workstation s2i-sample-app]$ podman images
REPOSITORY                      TAG      IMAGE ID      CREATED
localhost/s2i-sample-app          latest   3c8637c4372d  About an hour ago
localhost/s2i-do288-httdp         latest   d06040a9eeea  About an hour ago
...output omitted...
```

4.8. 在 `workstation` 虚拟机上对应用容器镜像进行本地测试。以随机用户的身份使用 `-u` 标记运行容器，以便以随机用户的身份在 RHOCP 集群上进行模拟运行。您可以从 `~/DO288/labs/apache-s2i/local-test.sh` 脚本复制或直接运行命令：

## 章 5 | 自定义源至镜像构建

```
[student@workstation s2i-sample-app]$ podman run --name test -u 1234 \
-p 8080:8080 -d s2i-sample-app
a5f4b3e5bf ...output omitted...
```

4.9. 验证容器是否已成功启动：

```
[student@workstation s2i-sample-app]$ podman ps
CONTAINER ID   IMAGE                  COMMAND   ...
a5f4b3e5bfaa   localhost/s2i-sample-app:latest   /bin/sh -c /usr/lib...
...output omitted...
```

4.10. 使用 curl 命令测试应用：

```
[student@workstation s2i-sample-app]$ curl http://localhost:8080
This is the index page from the app
```

4.11. 停止测试应用容器：

```
[student@workstation s2i-sample-app]$ podman stop test
a5f4b3e5bf ...output omitted...
```

## ▶ 5. 将 Apache HTTP 服务器 S2I 构建器镜像推送到 Quay.io 帐户。

5.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation s2i-sample-app]$ source /usr/local/etc/ocp4.config
```

5.2. 使用 podman 命令登录您的 Quay.io 帐户。系统将提示您输入密码。

```
[student@workstation s2i-sample-app]$ podman login \
-u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

5.3. 使用 skopeo copy 命令将 S2I 构建器镜像发布到 Quay.io 帐户。

```
[student@workstation s2i-sample-app]$ skopeo copy --format v2s1 \
containers-storage:localhost/s2i-do288-httd \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd
...output omitted...
Writing manifest to image destination
Storing signatures
```

## ▶ 6. 为 Apache HTTP 服务器 S2I 构建器镜像创建镜像流。

6.1. 登录 RHOCP，再创建新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation s2i-sample-app]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation s2i-sample-app]$ oc new-project \
${RHT_OCP4_DEV_USER}-apache-s2i
Now using project "youruser-apache-s2i" on server "https://
api.cluster.domain.example.com:6443".
...output omitted...
```

6.2. 从 Podman 存储的容器注册表 API 访问令牌创建机密。

您也可以从 /home/student/D0288/labs/apache-s2i 目录中的 `create-secret.sh` 脚本执行或剪切和粘贴以下 `oc create secret` 命令。

```
[student@workstation s2i-sample-app]$ oc create secret generic quayio \
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type=kubernetes.io/dockerconfigjson
secret/quayio created
```

6.3. 将新机密链接到 `builder` 服务帐户。

```
[student@workstation s2i-sample-app]$ oc secrets link builder quayio
```

6.4. 通过从您的 Quay.io 容器注册表导入 S2I 构建器镜像来创建镜像流：

```
[student@workstation s2i-sample-app]$ oc import-image s2i-do288-httd \
--from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httd --confirm
imagestream.image.openshift.io/s2i-do288-httd imported

Name: s2i-do288-httd
Namespace: youruser-apache-s2i
Created: Less than a second ago
Labels: <none>
Annotations: openshift.io/image.dockerRepositoryCheck=2019-06-26T02:30:59Z
Image Repository: image-registry.openshift-image-registry.svc:5000/youruser-
apache-s2i/s2i-do288-httd
Image Lookup: local=false
Unique Images: 1
Tags: 1

latest
tagged from quay.io/youruser/s2i-do288-httd

* quay.io/youruser/s2i-do288-httd@sha256:fe0cd09432...
    Less than a second ago

...output omitted...
```

6.5. 验证镜像流是否已创建好：

## 章 5 | 自定义源至镜像构建

```
[student@workstation s2i-sample-app]$ oc get is
NAME           IMAGE REPOSITORY      ...output omitted...
s2i-do288-httpd  ...youruser-apache-s2i/s2i-do288-httpd
```

- 7. 在 RHOCP 集群上部署课堂 Git 存储库中的 `html-helloworld` 应用，并进行测试。该应用由用于显示消息的单个 HTML 文件组成。

- 7.1. 基于 Git 中的来源创建一个名为 `hello` 新应用。您需要将 `s2i-do288-httpd` 镜像流作为前缀添加到 Git URL 中，以确保应用使用您之前创建的 Apache HTTP 服务器构建器镜像：

```
[student@workstation s2i-sample-app]$ oc new-app --name hello-s2i \
s2i-do288-httpd~https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps \
--context-dir=html-helloworld
--> Found image c7a496d (2 hours old) in image stream "youruser-apache-s2i/s2i-
do288-httpd" under tag "latest" for "s2i-do288-httpd"

Apache HTTP Server S2I builder image for D0288
-----
A basic Apache HTTP Server S2I builder image
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

- 7.2. 查看构建日志。等待构建完成并将应用容器镜像推送到 RHOCP 注册表：

```
[student@workstation s2i-sample-app]$ oc logs -f bc/hello-s2i
Cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Copying source files to web server directory...
Pushing image-registry.openshift-image-registry.svc:5000/youruser-apache-s2i/
hello-s2i:latest ...
...output omitted...
Push successful
```

- 7.3. 等待应用部署好。查看应用容器集的状态。应用容器集应处于 `Running` 状态：

```
[student@workstation s2i-sample-app]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
hello-s2i-1-build   0/1     Completed   0          71s
hello-s2i-57cb8dc96c-rnkgt  1/1     Running    0          50s
```

- 7.4. 使用路由公开应用供外部访问：

```
[student@workstation s2i-sample-app]$ oc expose svc hello-s2i
route.route.openshift.io/hello-s2i exposed
```

- 7.5. 使用 `oc get route` 命令获取路由 URL：

## 章 5 | 自定义源至镜像构建

```
[student@workstation s2i-sample-app]$ export APP_URL=$( `oc get route/hello-s2i \
-o jsonpath='{.spec.host}{"\n"}'` )
[student@workstation s2i-sample-app]$ echo ${APP_URL}
hello-s2i-youruser-apache-s2i.apps.cluster.domain.example.com
```

7.6. 使用您在上一步中获取的路由 URL 测试应用：

```
[student@workstation s2i-sample-app]$ curl ${APP_URL}
<html>
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

## ► 8. 清理。

8.1. 删除 RHOCP 中的 **apache-s2i** 项目：

```
[student@workstation s2i-sample-app]$ oc delete project \
${RHT_OCP4_DEV_USER}-apache-s2i
```

8.2. 删除先前在对应用进行本地测试时创建的 **test** 容器：

```
[student@workstation s2i-sample-app]$ podman rm test
a5f4b3e5bf ...output omitted...
```

8.3. 从 **workstation** 虚拟机中删除容器镜像：

```
[student@workstation s2i-sample-app]$ podman rmi -f \
localhost/s2i-sample-app \
localhost/s2i-do288-httdp \
registry.access.redhat.com/ubi8/ubi:8.4
...output omitted...
Untagged: localhost/s2i-sample-app:latest
...output omitted...
Untagged: localhost/s2i-do288-httdp:latest
...output omitted...
Untagged: registry.access.redhat.com/ubi8/ubi:8.4
```

8.4. 从外部注册表中删除 **s2i-do288-httdp** 镜像：

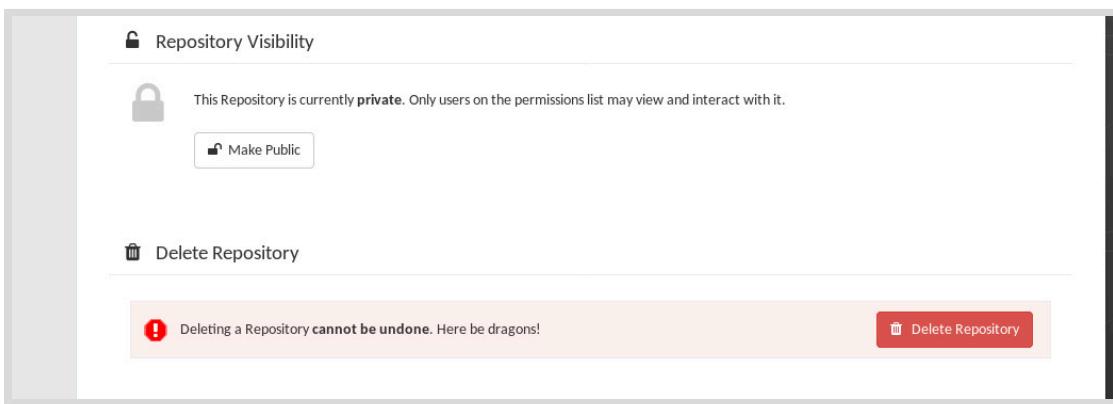
```
[student@workstation s2i-sample-app]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-httdp:latest
```

8.5. 使用您的个人免费帐户登录 Quay.io。

导航到 <http://quay.io> 并单击 **Sign In** 以提供用户凭据。

8.6. 在 Quay.io 主菜单上，单击 **Repositories** 并查找 **s2i-do288-httdp**。单击 **s2i-do288-httdp** 以显示 **Repository Activity** 页面。

- 8.7. 在 **s2i-do288-`httpd`** 存储库的 **Repository Activity** 页面，向下滚动并单击齿轮图标以显示“设置”选项卡。向下滚动并单击 **Delete Repository**。



- 8.8. 在 **Delete** 对话框中，输入存储库名称，单击 **Delete** 以确认要删除 **s2i-do288-`httpd`** 存储库。几分钟后，您将返回到 **Repositories** 页面。您现在可以注销 Quay.io。

## 完成

在 **workstation** 虚拟机上，执行 **lab apache-s2i finish** 脚本以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab apache-s2i finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 自定义源至镜像构建

在本实验中，您将创建一个 S2I 构建器镜像，以运行基于 Go 编程语言的应用。

## 成果

您应能够：

- 基于 RHEL 8 的通用基础镜像构建 Go 编程语言 S2I 构建器镜像。
- 使用 **s2i** 工具创建 Dockerfile，然后使用 Podman 构建和运行生成的容器镜像，从而在本地测试 S2I 构建器镜像。
- 将构建器镜像发布到您的个人 Quay.io 帐户。
- 使用 S2I 构建器镜像在红帽 OpenShift 容器平台 (RHOCP) 集群上部署并测试应用。
- 自定义 **run** 脚本以更改应用的行为，并重新部署应用以测试更改。

## 在你开始之前

要进行此实验，请确保您具有以下资源的访问权限：

- 正在运行的 RHOCP 集群。
- **s2i** 命令行工具。
- 包含 **go-hello** 应用源代码的 Git 存储库分叉。

在 **workstation** 虚拟机上运行以下命令，以验证前提条件。该命令还会下载用于复习实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab custom-s2i start
```

设置脚本会在 `/home/student/D0288/labs` 目录中创建一个名为 **custom-s2i** 的目录。该目录包含 **s2i create** 命令所创建的模板目录结构和文件。

## 要求

在本实验中，您需要部署一个用 Go 编程语言编写的应用。

提供了一个小示例 Go 应用，用于测试 S2I 构建器镜像。该示例 Go 应用会提供一个简单的 HTTP API，以基于 HTTP 请求所请求的资源来回复请求。

要完成实验，请为基于 Go 的应用构建 S2I 构建器镜像，然后按照以下要求在课堂 RHOCP 集群上测试并部署示例 Go 应用：

- 该 S2I 构建器镜像必须命名为 **s2i-do288-go**。构建器镜像必须位于您的个人 Quay.io 帐户中的以下位置：  
`quay.io/youruser/s2i-do288-go`
- 该 S2I 构建器镜像的镜像流命名为 **s2i-do288-go**。

## 章 5 | 自定义源至镜像构建

- 用于 RHOCP 的应用名称为 **greet**。
- 该镜像流和该应用都必须在名为 **youruser-custom-s2i** 的项目中进行创建。
- 应用的 HTTP API 必须可通过以下 URL 访问：  
`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com`。
- Go 应用源代码位于 **go-hello** 目录中的 **D0288-apps** Git 存储库内。
- 在将构建器镜像推送到 Quay.io 帐户之前，请先在 **workstation** 虚拟机上对应用进行本地测试。在测试构建器镜像时，请注意以下事项：
  - 应用的源代码位于 `~/D0288/labs/custom-s2i/test/test-app` 目录中。
  - 将测试应用容器镜像命名为 **s2i-go-app**。
  - 将测试容器命名为 **go-test**。
  - 对容器进行测试时，请务必使用随机用户 ID（如 1234），以便在 RHOCP 集群上进行模拟运行。
  - 将容器端口 8080 绑定到本地端口 8080。
  - 应用基于发出请求的 URL 返回问候语。例如：
- `http://localhost:8080/user1`，则会返回以下回复：

Hello user1!. Welcome!

- 测试完成后，请先删除 **go-test** 容器，然后再继续执行下一步。
- 使用您的 **s2i-do288-go** 构建器镜像测试从源构建 **go-hello** 应用。
- 测试在 RHOCP 上运行的 **go-hello** 应用后，通过在 **go-hello** 应用源代码中覆盖 **s2i-do288-go** 构建器镜像的 **run** 脚本对其自定义。
- 提交自定义 **run** 脚本并将其推送到 Github 存储库，该存储库用作应用构建的源。然后，启动一个新构建并验证 **go-hello** 应用的新版本返回西班牙语的问候语，如下所示：

Hola user1!. Bienvenido!

## 说明

1. 构建器镜像的 S2I 脚本在 `/home/student/D0288/labs/custom-s2i/s2i/bin` 目录中提供。请查看 **assemble**、**run** 和 **usage** 脚本。
2. 编辑构建器镜像的 Dockerfile 以包括一条指令，用于将 S2I 脚本复制到构建器镜像中的相应位置。紧接在文件中已存在的 **TODO** 注释之后添加该新指令。
3. 构建 S2I 构建器镜像。将映像命名为 **s2i-do288-go**。
4. 在本地的 **workstation** 虚拟机上的 `/home/student/D0288/labs/custom-s2i/test/test-app` 目录中构建可将 S2I 构建器镜像与应用源代码进行整合的应用容器镜像的 Dockerfile。在新的 `/home/student/s2i-go-app` 目录中构建 Dockerfile。

## 章 5 | 自定义源至镜像构建

5. 将 **s2i-do288-go** S2I 构建器镜像推送到您的个人 Quay.io 帐户。
6. 为 **s2i-do288-go** S2I 构建器镜像创建名为 **s2i-do288-go** 的镜像流。在名为 **youruser-custom-s2i** 的项目中创建镜像流。
7. 进入 **D0288-apps** Git 存储库的本地克隆，并签出课程存储库的 **master** 分支，以确保从已知良好的状态开始本练习：
8. 创建一个分支，以用于保存您在本练习中所做的任何更改，并将其推送到 GitHub。使用 **custom-s2i** 作为分支名称：
9. 将 **D0288-apps** 存储库的个人 GitHub 分叉中的 **go-hello** 应用部署到课堂 RHOCP 集群，并进行测试。部署应用时，请确保引用在上一步中创建的 **custom-s2i** 分支。应用会向 HTTP 请求所请求的资源回显问候语。  
例如，使用以下 URL (<http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1>) 调用应用会返回以下响应：

```
Hello user1!. Welcome!
```

10. 在应用源中为 **s2i-do288-go** 构建器镜像自定义 **run** 脚本。通过在启动时添加 **--lang es** 参数来改变应用的启动方式。这会改变应用所用的默认语言。  
将更改提交并推送到用作应用构建输入源的分支。
11. 重新构建并测试应用。现在，应用应该会以西班牙语回复请求。  
例如，如果使用以下 URL 调用应用：  
<http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1>，则会返回以下回复：

```
Hola user1!. Bienvenido!
```

12. 对您的作业进行评分。  
在 **workstation** 虚拟机上运行以下命令，以验证是否已完成所有任务：

```
[student@workstation ~]$ lab custom-s2i grade
```

13. 清理。执行以下步骤：

## 完成

在 **workstation** 虚拟机上，运行 **lab custom-s2i finish** 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab custom-s2i finish
```

本实验到此结束。

## ► 解决方案

# 自定义源至镜像构建

在本实验中，您将创建一个 S2I 构建器镜像，以运行基于 Go 编程语言的应用。

## 成果

您应能够：

- 基于 RHEL 8 的通用基础镜像构建 Go 编程语言 S2I 构建器镜像。
- 使用 **s2i** 工具创建 Dockerfile，然后使用 Podman 构建和运行生成的容器镜像，从而在本地测试 S2I 构建器镜像。
- 将构建器镜像发布到您的个人 Quay.io 帐户。
- 使用 S2I 构建器镜像在红帽 OpenShift 容器平台 (RHOCP) 集群上部署并测试应用。
- 自定义 **run** 脚本以更改应用的行为，并重新部署应用以测试更改。

## 在你开始之前

要进行此实验，请确保您具有以下资源的访问权限：

- 正在运行的 RHOCP 集群。
- **s2i** 命令行工具。
- 包含 **go-hello** 应用源代码的 Git 存储库分叉。

在 **workstation** 虚拟机上运行以下命令，以验证前提条件。该命令还会下载用于复习实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab custom-s2i start
```

设置脚本会在 `/home/student/D0288/labs` 目录中创建一个名为 **custom-s2i** 的目录。该目录包含 **s2i create** 命令所创建的模板目录结构和文件。

## 要求

在本实验中，您需要部署一个用 Go 编程语言编写的应用。

提供了一个小示例 Go 应用，用于测试 S2I 构建器镜像。该示例 Go 应用会提供一个简单的 HTTP API，以基于 HTTP 请求所请求的资源来回复请求。

要完成实验，请为基于 Go 的应用构建 S2I 构建器镜像，然后按照以下要求在课堂 RHOCP 集群上测试并部署示例 Go 应用：

- 该 S2I 构建器镜像必须命名为 **s2i-do288-go**。构建器镜像必须位于您的个人 Quay.io 帐户中的以下位置：  
`quay.io/youruser/s2i-do288-go`
- 该 S2I 构建器镜像的镜像流命名为 **s2i-do288-go**。
- 用于 RHOCP 的应用名称为 **greet**。

## 章 5 | 自定义源至镜像构建

- 该镜像流和该应用都必须在名为 `youruser-custom-s2i` 的项目中进行创建。
- 应用的 HTTP API 必须可通过以下 URL 访问：  
`http://greet-youruser-custom-s2i.apps.cluster.domain.example.com`。
- Go 应用源代码位于 `go-hello` 目录中的 `D0288-apps` Git 存储库内。
- 在将构建器镜像推送到 Quay.io 帐户之前，请先在 `workstation` 虚拟机上对应用进行本地测试。在测试构建器镜像时，请注意以下事项：
  - 应用的源代码位于 `~/D0288/labs/custom-s2i/test/test-app` 目录中。
  - 将测试应用容器镜像命名为 `s2i-go-app`。
  - 将测试容器命名为 `go-test`。
  - 对容器进行测试时，请务必使用随机用户 ID（如 1234），以便在 RHOCP 集群上进行模拟运行。
  - 将容器端口 8080 绑定到本地端口 8080。
  - 应用基于发出请求的 URL 返回问候语。例如：  
`http://localhost:8080/user1`，则会返回以下回复：

```
Hello user1!. Welcome!
```

- 测试完成后，请先删除 `go-test` 容器，然后再继续执行下一步。
- 使用您的 `s2i-d0288-go` 构建器镜像测试从源构建 `go-hello` 应用。
- 测试在 RHOCP 上运行的 `go-hello` 应用后，通过在 `go-hello` 应用源代码中覆盖 `s2i-d0288-go` 构建器镜像的 `run` 脚本对其自定义。
- 提交自定义 `run` 脚本并将其推送到 Github 存储库，该存储库用作应用构建的源。然后，启动一个新构建并验证 `go-hello` 应用的新版本返回西班牙语的问候语，如下所示：

```
Hola user1!. Bienvenido!
```

## 说明

- 构建器镜像的 S2I 脚本在 `/home/student/D0288/labs/custom-s2i/s2i/bin` 目录中提供。请查看 `assemble`、`run` 和 `usage` 脚本。
- 编辑构建器镜像的 Dockerfile 以包括一条指令，用于将 S2I 脚本复制到构建器镜像中的相应位置。紧接在文件中已存在的 `TODO` 注释之后添加该新指令。
  - 编辑 `~/D0288/labs/custom-s2i/Dockerfile` 处的 Dockerfile，并在 `TODO` 注释之后添加以下 `COPY` 指令。您还可以从 `~/D0288/solutions/custom-s2i/Dockerfile` 处提供的解决方案 Dockerfile 中复制该指令：

```
COPY ./s2i/bin/ /usr/libexec/s2i
```

## 章 5 | 自定义源至镜像构建

3. 构建 S2I 构建器镜像。将映像命名为 s2i-do288-go。

3.1. 创建 S2I 构建器镜像：

```
[student@workstation ~]$ cd ~/D0288/labs/custom-s2i
[student@workstation custom-s2i]$ podman build -t s2i-do288-go .
STEP 1/12: FROM registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
COMMIT s2i-do288-go
...output omitted...
```

3.2. 验证构建器镜像是否已创建好：

```
[student@workstation custom-s2i]$ podman images
REPOSITORY                      TAG      IMAGE ID      ...
localhost/s2i-do288-go          latest   d1f856d10fa7  ...
...output omitted...
```

4. 在本地的 workstation 虚拟机上的 /home/student/D0288/labs/custom-s2i/test/test-app 目录中构建可将 S2I 构建器镜像与应用源代码进行整合的应用容器镜像的 Dockerfile。在新的 /home/student/s2i-go-app 目录中构建 Dockerfile。

4.1. 创建一个名为的 s2i-go-app 目录，其中 s2i 命令可以保存 Dockerfile。

```
[student@workstation custom-s2i]$ mkdir /home/student/s2i-go-app
```

4.2. 使用 s2i build 命令为应用容器镜像生成 Dockerfile：

```
[student@workstation custom-s2i]$ s2i build test/test-app/ \
s2i-do288-go s2i-go-app \
--as-dockerfile /home/student/s2i-go-app/Dockerfile
Application dockerfile generated in /home/student/s2i-go-app/Dockerfile
```

4.3. 从生成的 Dockerfile 构建测试容器镜像。

```
[student@workstation custom-s2i]$ cd ~/s2i-go-app
[student@workstation s2i-go-app]$ podman build -t s2i-go-app .
STEP 1: FROM s2i-do288-go
STEP 2: LABEL "io.k8s.display-name"="s2i-go-app"
...output omitted...
COMMIT s2i-go-app
...output omitted...
```

4.4. 验证应用容器镜像是否已创建好：

```
[student@workstation s2i-go-app]$ podman images
REPOSITORY                      TAG      IMAGE ID      ...
localhost/s2i-go-app          latest   7d3d8f894f2f  ...
...output omitted...
```

## 章 5 | 自定义源至镜像构建

- 4.5. 在 **workstation** 虚拟机上对应用容器镜像进行本地测试。使用 **-u** 标记运行容器，以便以随机用户的身份在 RHOCP 集群上进行模拟运行。您可以从 **~/DO288/labs/custom-s2i/local-test.sh** 脚本复制或直接运行命令：

```
[student@workstation s2i-go-app]$ podman run --name go-test -u 1234 \
-p 8080:8080 -d s2i-go-app
18b903cfaae4...
```

- 4.6. 验证容器是否已成功启动：

```
[student@workstation s2i-go-app]$ podman ps
CONTAINER ID IMAGE COMMAND
18b903cfaae4 localhost/s2i-go-app:latest /bin/sh -c /usr/lib...
```

- 4.7. 使用 **curl** 命令测试应用：

```
[student@workstation s2i-go-app]$ curl http://localhost:8080/user1
Hello user1!. Welcome!
```

- 4.8. 停止 **go-test** 应用容器：

```
[student@workstation s2i-go-app]$ podman stop go-test
18b903cfaae4...
[student@workstation s2i-go-app]$ cd ~
```

## 5. 将 **s2i-do288-go** S2I 构建器镜像推送到您的个人 Quay.io 帐户。

- 5.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 5.2. 使用 **podman** 命令登录您的 Quay.io 帐户。在系统提示您时输入您的 Quay.io 密码。

```
[student@workstation ~]$ podman login \
-u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 5.3. 使用 **skopeo copy** 命令将 S2I 构建器镜像发布到 Quay.io 帐户。

```
[student@workstation ~]$ skopeo copy --format v2s1 \
containers-storage:localhost/s2i-do288-go \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go
...output omitted...
Writing manifest to image destination
Storing signatures
```

## 6. 为 **s2i-do288-go** S2I 构建器镜像创建名为 **s2i-do288-go** 的镜像流。在名为 **youruser-custom-s2i** 的项目中创建镜像流。

## 章 5 | 自定义源至镜像构建

- 6.1. 登录 RHOCP，再创建新项目。在项目名称中添加开发人员用户名作为前缀。

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-custom-s2i
Now using project "youruser-custom-s2i" on server "https://
api.cluster.domain.example.com:6443".
```

- 6.2. 如果您尚未登录，请使用 Podman 登录您的个人 Quay.io 帐户，以便您可以导出 **auth.json** 文件以用于下一步中的拉取机密。

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

- 6.3. 从 Podman 存储的容器注册表 API 访问令牌创建机密。

您也可以从 `/home/student/D0288/labs/custom-s2i` 目录中的 `create-secret.sh` 脚本执行或剪切和粘贴以下 `oc create secret` 命令。

```
[student@workstation ~]$ oc create secret generic quayio \
--from-file .dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type=kubernetes.io/dockerconfigjson
secret/quayio created
```

- 6.4. 将新机密链接到 **builder** 服务帐户。

```
[student@workstation ~]$ oc secrets link builder quayio
```

- 6.5. 通过从私有课堂注册表导入 S2I 构建器镜像来创建镜像流：

```
[student@workstation ~]$ oc import-image s2i-do288-go \
--from quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go \
--confirm
imagestream.image.openshift.io/s2i-do288-go imported
...output omitted...
```

- 6.6. 验证镜像流是否已创建好：

```
[student@workstation ~]$ oc get is
NAME           IMAGE REPOSITORY          ...output omitted...
s2i-do288-go   ...youruser-custom-s2i/s2i-do288-go ...output omitted...
```

7. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `master` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

## 章 5 | 自定义源至镜像构建

8. 创建一个分支，以用于保存您在本练习中所做的任何更改，并将其推送到 GitHub。使用 **custom-s2i** 作为分支名称：

```
[student@workstation D0288-apps]$ git checkout -b custom-s2i
Switched to a new branch 'custom-s2i'
[student@workstation D0288-apps]$ git push -u origin custom-s2i
...output omitted...
* [new branch]      custom-s2i -> custom-s2i
Branch custom-s2i set up to track remote branch custom-s2i from origin.
[student@workstation D0288-apps]$ cd ~
```

9. 将 **D0288-apps** 存储库的个人 GitHub 分叉中的 **go-hello** 应用部署到课堂 RHOCP 集群，并进行测试。部署应用时，请确保引用在上一步中创建的 **custom-s2i** 分支。应用会向 HTTP 请求所请求的资源回显问候语。

例如，使用以下 URL (<http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1>) 调用应用会返回以下响应：

```
Hello user1!. Welcome!
```

- 9.1. 使用 **s2i-do288-go** 镜像流基于 GitHub 中的源代码创建新应用：

```
[student@workstation ~]$ oc new-app --name greet \
s2i-do288-go-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#custom-s2i \
--context-dir=go-hello
--> Found image a29d3e7 (About an hour old) in image stream "youruser-custom-s2i/
s2i-do288-go" under tag "latest" for "s2i-do288-go"

Go programming language S2I builder image for D0288
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "greet" created
  buildconfig.build.openshift.io "greet" created
  deployment.apps "greet" created
  service "greet" created
--> Success
...output omitted...
```

- 9.2. 查看构建日志。等待构建完成并将应用容器镜像推送到 RHOCP 注册表：

```
[student@workstation ~]$ oc logs -f bc/greet
cloning "https://github.com/youruser/D0288-apps" ...
...output omitted...
--> Installing application source...
--> Building application from source...
...output omitted...
Push successful
```

- 9.3. 等待应用部署好。查看应用容器集的状态。应用容器集必须处于 **Running** 状态：

## 章 5 | 自定义源至镜像构建

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
greet-1-build  0/1     Completed  0          53s
greet-6986b8fcb-fn4rq  1/1     Running   0          14s
```

9.4. 使用路由公开应用供外部访问：

```
[student@workstation ~]$ oc expose svc greet
route.route.openshift.io/greet exposed
```

9.5. 使用 `oc get route` 命令获取路由 URL：

```
[student@workstation ~]$ oc get route/greet -o jsonpath='{.spec.host}{"\n"}'
greet-youruser-custom-s2i.apps.cluster.domain.example.com
```

9.6. 使用您在上一步中获取的路由 URL 测试应用：

```
[student@workstation ~]$ curl \
http://greet-${RHT_OCP4_DEV_USER}-custom-s2i.${RHT_OCP4_WILDCARD_DOMAIN}/user1
Hello user1!. Welcome!
```

10. 在应用源中为 `s2i-do288-go` 构建器镜像自定义 `run` 脚本。通过在启动时添加 `--lang es` 参数来改变应用的启动方式。这会改变应用所用的默认语言。

将更改提交并推送到用作应用构建输入源的分支。

10.1. S2I 脚本可以在位于 `D0288-apps/go-hello` 目录中的应用源的 `.s2i/bin` 目录中进行自定义。创建目录：

```
[student@workstation ~]$ mkdir -p ~/D0288-apps/go-hello/.s2i/bin
```

10.2. 从 `~/D0288/labs/custom-s2i/s2i/bin/run` 复制 S2I 构建器镜像中的 `run` 脚本：

```
[student@workstation ~]$ cp ~/D0288/labs/custom-s2i/s2i/bin/run \
~/D0288-apps/go-hello/.s2i/bin/
```

10.3. 自定义 `~/D0288-apps/go-hello/.s2i/bin/run` 脚本，并添加 `--lang es` 选项至应用启动。您还可从 `~/D0288/solutions/custom-s2i/s2i/bin/run.es` 文件复制完整的脚本：

```
...output omitted...
echo "Starting app with lang option 'es'..."
exec /opt/app-root/app --lang es
```

10.4. 将更改提交到 Git：

```
[student@workstation ~]$ cd ~/D0288-apps/go-hello
[student@workstation go-hello]$ git add .
[student@workstation go-hello]$ git commit -m "Customized run script"
...output omitted...
[student@workstation go-hello]$ git push
...output omitted...
[student@workstation go-hello]$ cd ~
```

- 11.** 重新构建并测试应用。现在，应用应该会以西班牙语回复请求。

例如，如果使用以下 URL 调用应用：

<http://greet-youruser-custom-s2i.apps.cluster.domain.example.com/user1>，则会返回以下回复：

```
Hola user1!. Bienvenido!
```

11.1. 启动应用新构建：

```
[student@workstation ~]$ oc start-build greet
build.build.openshift.io/greet-2 started
```

11.2. 按照构建日志操作并验证是否已经创建新的容器镜像并推送到 RHOCP 内部注册表：

```
[student@workstation ~]$ oc logs -f bc/greet
Cloning "https://github.com/youruser/D0288-apps" ...
Commit: 0023a1b02342b633aad49e58a2eeba11dff33c3d (customized run script)
...output omitted...
Push successful
```

11.3. 等待应用容器集部署好。容器集必须处于 **Running** 状态。验证应用容器集的状态：

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
greet-1-build 0/1     Completed  0          36m
greet-2-build 0/1     Completed  0          50s
greet-6986b8fcb-fn4rq 1/1     Running   0          36m
...output omitted...
```

11.4. 使用您在 Step 9.5 中获取的路由 URL 测试应用：

```
[student@workstation ~]$ curl \
http://greet-${RHT_OCP4_DEV_USER}-custom-s2i.${RHT_OCP4_WILDCARD_DOMAIN}/user1
Hola user1!. Bienvenido!
```

- 12.** 对您的作业进行评分。

在 **workstation** 虚拟机上运行以下命令，以验证是否已完成所有任务：

```
[student@workstation ~]$ lab custom-s2i grade
```

- 13.** 清理。执行以下步骤：

## 章 5 | 自定义源至镜像构建

13.1. 删除 RHOCP 中的 `youruser-custom-s2i` 项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-custom-s2i
```

13.2. 删除先前创建的用于对应用进行本地测试的所有测试容器。

```
[student@workstation ~]$ podman rm go-test
```

13.3. 删除在进行本实验的过程中在 `workstation` 虚拟机中构建的所有容器镜像。

```
[student@workstation ~]$ podman rmi -f \
localhost/s2i-go-app \
localhost/s2i-do288-go \
registry.access.redhat.com/ubi8/ubi:8.0
...output omitted...
```

13.4. 从外部注册表中删除 `s2i-do288-go` 镜像：

```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/s2i-do288-go:latest
```

13.5. 使用您的个人免费帐户登录 Quay.io。

导航到 <http://quay.io> 并单击 **Sign In** 以提供用户凭据。单击 **Sign in to Quay Container Registry** 以登录 Quay.io。

13.6. 在 Quay.io 主菜单上，单击 **Repositories** 并查找 `s2i-do288-go`。`lock` 图标指示它是一个私有存储库，需要对拉取和推送进行身份验证。单击 `s2i-do288-go` 以显示 **Repository Activity** 页面。

13.7. 在 `s2i-do288-go` 存储库的 **Repository Activity** 页面，向下滚动并单击齿轮图标以显示 **Settings** 选项卡。向下滚动并单击 **Delete Repository**。

13.8. 在 **Delete** 对话框口，单击 **Delete** 以确认删除 `s2i-do288-go` 存储库。几分钟后，您将返回到 **Repositories** 页面。您现在可以注销 Quay.io。

## 完成

在 `workstation` 虚拟机上，运行 `lab custom-s2i finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab custom-s2i finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- S2I 构建器镜像是一种特殊的容器镜像，开发人员会用它生成应用容器镜像。构建器镜像包含基础操作系统库、语言运行时、框架和应用依赖项，以及各种源至镜像工具和实用程序。
- S2I 构建器镜像会默认提供 S2I 脚本。您可以向 `.s2i/bin` 目录添加自己的 S2I 脚本，以在应用源代码中覆盖这些 S2I 脚本。
- `s2i` 命令行工具用于在 OpenShift 外部构建和测试 S2I 构建器镜像。
- 在 RHEL 8 或 OpenShift 4 环境中，（默认情况下 Docker 不可用），请使用 `s2i build` 命令的 `--as-dockerfile` 选项。这将导致该命令生成 Dockerfile 和支持目录，您可以使用 Podman 构建这些目录来测试 S2I 构建器镜像。



## 章 6

# 部署多容器应用

### 目标

使用 Helm 图表和 Kustomize 部署多容器应用。

### 培训目标

- 介绍 OpenShift 模板中的各个元素。
- 使用 Helm 图表构建多容器应用。
- 自定义 OpenShift 部署。

### 章节

- 描述 OpenShift 模板（及测验）
- 创建 Helm 图表（及引导式练习）
- 使用 Kustomize 自定义部署（及引导式练习）

### 实验

部署多容器应用

# 描述 OpenShift 模板

## 培训目标

学完本节后，您应能够将红帽 OpenShift (RHOCP) 资源转换为 RHOCP 模板。

## 描述模板

RHOCP 模板是由一组 RHOCP 资源构成的 YAML 或 JSON 文件。模板定义用于自定义资源配置的参数。RHOCP 通过用值替换参数引用并创建一组自定义资源来处理模板。

如果要将一组资源部署为单个单元，而不是单独部署，则模板非常有用。使用模板的示例用例包括：

- 独立软件供应商 (ISV) 提供了一个模板，用于在 RHOCP 上部署其产品。模板包含与产品所包含容器的配置详细信息。也包含部署配置，如副本、服务和路由的数量、持久存储配置、运行状况检查以及用于计算资源的资源限制，如 CPU、内存和 I/O。
- 多层应用由许多独立的组件构成，如 Web 服务器、应用服务器和数据库。您应该将这些组件作为一个单元一起部署到 RHOCP 上，以简化在暂存环境中部署应用的流程。这样，您的 QA 和验收测试团队能够快速调配应用以进行测试。



### 注意

RHOCP 模板已被弃用，将在未来 RHOCP 版本中剔除。尽管 RHOCP 模板仍然受到支持，但对它的探讨是为了完整性，红帽建议您不要使用。

## 模板语法

RHOCP 模板语法遵循 RHOCP 资源的一般语法，但使用 **objects** 属性代替 **spec** 属性。模板通常包括 **parameters** 和 **labels** 属性，以及元数据中的注释。

以下列表显示 YAML 格式的示例模板定义，并说明了主要语法元素。与任何 RHOCP 资源一样，您也可以使用 JSON 语法定义模板：

```
apiVersion: template.openshift.io/v1
kind: Template ①
metadata:
  name: mytemplate
  annotations:
    description: "Description" ②
objects: ③
- apiVersion: v1
  kind: Pod
  metadata:
    name: myapp
  spec:
    containers:
      - env:
          - name: MYAPP_CONFIGURATION
```

```

    value: ${MYPARAMETER} ④
  image: myorganization/myapplication
  name: myapp
  ports:
    - containerPort: 80
      protocol: TCP
  parameters: ⑤
    - description: Myapp configuration data
      name: MYPARAMETER
      required: true
  labels: ⑥
    mylabel: myapp

```

- ① 模板资源类型
- ② 供 RHOCP 工具使用的可选注释
- ③ 资源列表
- ④ 引用模板参数
- ⑤ 参数列表
- ⑥ 标签列表

模板的资源列表通常包括其他资源，如构建配置、部署配置、持久卷声明 (PVC)、服务和路由。

资源列表中的任何属性都可以引用任何参数的值。

您可以为模板定义自定义标签。RHOCP 将这些标签添加到模板创建的所有资源中。

模板定义多个资源时，重要的是要考虑这些资源的定义顺序以适应资源之间的依赖关系。如果资源引用不存在的依赖资源，则 RHOCP 不会报告错误。

如果在依赖资源之前启动，则由资源触发的进程可能会失败。示例之一就是引用某个镜像流作为输出镜像的构建配置，您的模板将在构建配置之后定义该镜像流。在这种情况下，构建配置可能会在镜像流存在之前启动构建。

## 定义模板参数

前面的示例模板包含必填参数的定义。可选参数和必填参数都可以提供默认值。例如：

```

parameters:
- description: Myapp configuration data
  name: MYPARAMETER
  value: /etc/myapp/config.ini

```

RHOCP 可以为参数生成随机默认值。这对机密和密码很有用：

```

parameters:
- description: ACME cloud provider API key
  name: APIKEY
  generate: expression
  from:"[a-zA-Z0-9]{12}"

```

## 章 6 | 部署多容器应用

生成值的语法是 Perl 正则表达式语法的子集。有关完整模板参数表达式语法，请参见本节末尾的参考资料。

## 将模板添加到 OpenShift

要向 RHOCP 添加模板，请使用 `oc create` 命令或 Web 控制台（使用导入 YAML 页面）从模板定义文件创建模板资源，方法与任何其他类型的资源相同。

创建仅包含模板的项目是一种常见做法，因为 RHOCP 可让用户轻松在多个用户和项目之间共享模板。这些项目通常包括其他可能共享的资源，例如镜像流。

红帽 RHOCP 容器平台的默认安装在 `openshift` 项目中提供了多个模板。所有 RHOCP 集群用户都具有对 `openshift` 项目的读取权限，但只有集群管理员才有在此项目中创建或删除模板的权限。

## 从模板创建应用

您可以直接从模板资源定义文件部署应用。`oc new-app` 命令和 `oc process` 命令将模板文件作为输入并进行处理，以应用参数并创建资源。

您还可以将模板文件传递给 `oc create` 命令以在 RHOCP 中创建模板资源。如果模板要由多个开发人员重复利用，最好在共享项目中创建模板资源。如果模板旨在由单个部署使用，则最好将其保存在文件中。

`oc new-app` 命令从模板创建资源，而 `oc process` 命令从模板创建资源列表。您必须将 `oc process` 命令生成的资源列表保存到文件中，或者将其作为输入传递给 `oc create` 命令以从列表中创建资源。

对于 `oc new-app` 命令和 `oc process` 命令，必须使用不同的 `-p` 选项提供每个参数值。两个命令都接受的另一个选项是 `-l` 选项，它为从模板创建的所有资源添加标签。

以下是从模板文件部署应用的示例 `oc new-app` 命令：

```
[user@host ~]$ oc new-app --file mytemplate.yaml -p PARAM1=value1 \
-p PARAM2=value2
```

下例中的 `oc process` 命令将值应用到模板，并将结果存储在本地文件中：

```
[user@host ~]$ oc process -f mytemplate.yaml -p PARAM1=value1 \
-p PARAM2=value2 > myresourcelist.yaml
```

然后将上一个示例生成的文件提供给 `oc create` 命令：

```
[user@host ~]$ oc create -f myresourcelist.yaml
```

您可以使用管道组合前两个示例：

```
[user@host ~]$ oc process -f mytemplate.yaml -p PARAM1=value1 \
-p PARAM2=value2 | oc create -f -
```

红帽建议使用 `oc new-app` 命令而不是 `oc process` 命令。可以使用 `oc process` 命令与 `-f` 选项以仅列出指定模板定义的参数：

```
[user@host ~]$ oc process -f mytemplate.yaml --parameters
```



### 注意

RHOCP 4.10 Web 控制台中没有可列出或查看模板的部分。不过，您可以在 Administrator > Home > Search 和 Developer > Search 页面中列出可用的模板。

默认情况下，您不能从 RHOCP 4.10 Web 控制台中的自定义模板创建应用。集群管理员可以将其他模板安装到 Developer 目录中。Developer 目录中的模板可用于创建新应用。



### 参考文献

有关更多信息，请参阅红帽 OpenShift 容器平台 4.10 文档中的 Using Templates 章节，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/images/using-templates#using-templates](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/images/using-templates#using-templates)

## ► 小测验

# 描述 OpenShift 模板

选择以下问题的正确答案：

### ► 1. OpenShift 模板的目的是什么？

- a. 描述单个容器的资源配置。
- b. 封装一组 OpenShift 资源以供重复利用。
- c. 为 OpenShift 集群提供自定义配置。
- d. 自定义 Web 控制台的外观。

### ► 2. 以下哪两种方法将模板参数定义为可选项？（请选择两项。）

- a. 将 `required` 属性设置为 `false`。
- b. 将 `required` 属性设置为 `not`。
- c. 省略 `required` 属性。
- d. 将 `required` 属性设置为 `expression`。
- e. 将 `from` 属性设置为正则表达式。

### ► 3. 若要从模板创建资源，建议使用以下哪一个命令？

- a. `oc export`。
- b. `oc new-app`。
- c. `oc create`。
- d. `oc process`。
- e. 以上都不对。

### ► 4. 哪三个语句描述了插入模板参数引用的有效位置？（请选择三项。）

- a. 作为容器集资源中的环境变量的名称。
- b. 作为模板创建的所有资源中的标签的密钥。
- c. 作为路由资源中 `host` 属性的值。
- d. 作为模板中注释的值。
- e. 作为容器集资源内的环境变量的值。

## ► 解决方案

# 描述 OpenShift 模板

选择以下问题的正确答案：

► 1. **OpenShift 模板的目的是什么？**

- a. 描述单个容器的资源配置。
- b. 封装一组 OpenShift 资源以供重复利用。
- c. 为 OpenShift 集群提供自定义配置。
- d. 自定义 Web 控制台的外观。

► 2. **以下哪两种方法将模板参数定义为可选项？（请选择两项。）**

- a. 将 `required` 属性设置为 `false`。
- b. 将 `required` 属性设置为 `not`。
- c. 省略 `required` 属性。
- d. 将 `required` 属性设置为 `expression`。
- e. 将 `from` 属性设置为正则表达式。

► 3. **若要从模板创建资源，建议使用以下哪一个命令？**

- a. `oc export`。
- b. `oc new-app`。
- c. `oc create`。
- d. `oc process`。
- e. 以上都不对。

► 4. **哪三个语句描述了插入模板参数引用的有效位置？（请选择三项。）**

- a. 作为容器集资源中的环境变量的名称。
- b. 作为模板创建的所有资源中的标签的密钥。
- c. 作为路由资源中 `host` 属性的值。
- d. 作为模板中注释的值。
- e. 作为容器集资源内的环境变量的值。

# 创建 Helm 图表

---

## Helm 图表

Helm 是 Kubernetes 应用的开源软件包管理器。它提供了一种途径，用来打包、共享和管理 Kubernetes 应用的生命周期。

Helm 图表是定义 Helm 应用的文件和模板的集合。这些文件稍后进行打包，以利用 Helm 存储库进行分发。

## Helm 生命周期

用于管理 Helm 图表生命周期的 Helm CLI 命令的基本命令有：

### Helm 命令

命令	描述
dependency	管理图表的依赖项
install	安装图表
list	列出已安装的发行版
pull	从存储库下载图表
rollback	将发行版回滚到以前的版本
search	搜索图表中的关键字
show	显示图表的信息
status	显示指定发行版的状态
uninstall	卸载发行版
upgrade	升级发行版

## Helm 图表结构

`helm create` 命令可以正确的结构创建所需的文件。此命令创建的文件是 Helm 图表所需的基本文件，其中包括应用正常运行不可或缺的模板。

Helm 图表主要由两个 YAML 文件和一个模板列表构成。

YAML 文件有：

- **Chart.yaml**: 保存图表定义信息。
- **values.yaml**: 保存 Helm 在默认模板和用户创建的模板中使用的值。

除这两个文件外，Helm 图表还保存若干模板文件，它们是构成应用的 Kubernetes 资源的基础。

Chart.yaml 文件的基本结构为：

```
apiVersion: v2 ①
name: mychart ②
description: A Helm chart for Kubernetes ③
type: application ④
version: 0.1.0 ⑤
appVersion: "1.0.0" ⑥
```

- ① 要使用 Helm API 的版本
- ② Helm 图表的名称
- ③ Helm 图表的描述
- ④ Helm 图表类型：应用或库
- ⑤ Helm 图表的版本
- ⑥ 此图表软件包的应用版本

Chart.yaml 文件可以保存其他可选值，其中一个最重要的值是 **dependencies** 部分。dependencies 部分中保存让此 Helm 图表正常工作的其他 Helm 图表的列表。

dependencies 部分的结构为：

```
dependencies: ①
  - name: dependency1 ②
    version: 1.2.3 ③
    repository: https://examplerrepo.com/charts ④
  - name: dependency2
    version: 3.2.1
    repository: https://helmrepo.example.com/charts
```

- ① 依赖项的列表
- ② 第一个必要 Helm 图表的名称
- ③ 要依赖的图表版本
- ④ 保存依赖 Helm 图表的存储库

## 图表值

Helm 处理图表中的模板文件，并在处理图表期间将占位符替换为实际的值。您可以使用 **values.yaml** 文件以静态方式提供这些值，或在打包或安装期间使用 **helm** 命令行工具的 **--set** 标志动态提供这些值。

最常见的做法是使用 **values.yaml** 文件提供大部分值，并且仅将动态值保留在安装时必须提供的值。

以下是此文件内容的摘录：

```

...
image:
  repository: container.repo/name ①
  pullPolicy: IfNotPresent ②
  tag: "2.1" ③

...
serviceAccount:
  create: true ④
  annotations: {}
  name: "" ⑤

...
service:
  type: ClusterIP ⑥
  port: 80 ⑦

```

- ① 应用容器镜像的链接
- ② Kubernetes 集群中的容器拉取策略
- ③ 要使用的容器标签，默认值是图表的 `appVersion`
- ④ 是否为应用创建服务帐户
- ⑤ 服务帐户的名称；若为空，则自动生成
- ⑥ Kubernetes 服务的类型
- ⑦ 外部应用端口

## 模板

Helm 使用模板在运行时创建在 Kubernetes 集群中部署应用所需的资源。Helm CLI 工具会创建其中一些模板，但您可以修改它们或创建新的模板来满足自己的需要。

Helm 使用 Go Template 语言定义 `templates` 目录中的模板以及一些其他函数。

借助一个通用 `deployment.yaml` 模板文件的一个小节，您可以查看占位符和条件部分的用法：

```

metadata:
  name: {{ include "mychart.fullname" . }} ①
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }} ②
  replicas: {{ .Values.replicaCount }} ③
  {{- end }} ④
  template:

```

```
spec:  
  containers:  
    - name: {{ .Chart.Name }} ⑤
```

- ① `Chart.yaml` 文件中值的前缀是图表的名称
- ② 如果值为 `false`，则输出这个块
- ③ `values.yaml` 文件中值的前缀为 `Values`
- ④ 结束条件块
- ⑤ 图表名称的前缀是 `Chart`



### 参考文献

#### Helm

<https://helm.sh/>

#### Go Template 语言

<https://golang.org/pkg/text/template/>

## ► 指导练习

# 创建 Helm 图表

在本练习中，您将为一个多容器应用创建 Helm 图表，并部署到红帽 OpenShift (RHOCP) 集群中。

## 成果

您应当能够为 Famous Quotes 应用及其依赖项创建 Helm 图表并部署到 RHOCP 集群。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 已配置并在运行的 RHOCP 集群。
- Helm CLI 工具

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令来验证此练习的前提条件。

```
[student@workstation ~]$ lab multicontainer-helm start
```

## 说明

### ► 1. 创建新的 Helm 图表。

#### 1.1. 创建 `famouschart` Helm 图表。

使用 `helm create` 命令创建一个全新的 Helm 图表，并命名为 `famouschart`。

```
[student@workstation ~]$ cd ~/DO288/labs/multicontainer-helm
[student@workstation multicontainer-helm]$ helm create famouschart
Creating famouschart
[student@workstation multicontainer-helm]$ cd famouschart
```

#### 1.2. 验证文件结构。

```
[student@workstation famouschart]$ tree .
```

```
.
├── charts
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   ├── _helpers.tpl
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── serviceaccount.yaml
│   └── service.yaml
```

```
|   └── tests  
|     └── test-connection.yaml  
└── values.yaml
```

## ▶ 2. 配置应用部署。

使用 `values.yaml` 文件，为应用配置生成的部署。

### 2.1. 配置镜像和版本值。

更新 `values.yaml` 文件，将 `image` 部分的 `repository` 属性设置为 `quay.io/redhattraining/famous-quotes` 镜像，然后将同一部分的 `tag` 属性值设置为 `2.1` 以选择应用的适当版本。

`values.yaml` 文件的对应部分应当如下所示：

```
image:  
  repository: quay.io/redhattraining/famous-quotes  
  pullPolicy: IfNotPresent  
  tag: "2.1"
```

### 2.2. 确保容器使用正确的端口来连接应用。

在 `templates/deployment.yaml` 文件中，更改 `containers` 部分中包含的 `containerPort` 属性，以便它使用值 `8000`。

`templates/deployment.yaml` 文件的对应部分应当如下所示：

```
ports:  
  - name: http  
    containerPort: 8000  
    protocol: TCP
```

## ▶ 3. 添加数据库依赖项。

Famous Quotes 应用使用数据库来存储名言，因此您必须随同应用提供一个数据库，以便它能正常工作。为此，您必须提供应用图表的依赖项并配置数据库图表。

### 3.1. 将 `mariadb` 依赖项添加到应用图表。

为此，可将以下代码片段添加到 `Chart.yaml` 文件的末尾：

```
dependencies:  
  - name: mariadb  
    version: 11.0.13  
    repository: https://charts.bitnami.com/bitnami
```

您可通过将 `~/DO288/labs/multicontainer-helm/dependencies.yaml` 的内容附加到 `Chart.yaml` 文件来添加此选项：

```
[student@workstation famouschart]$ cat ../dependencies.yaml >> Chart.yaml
```

### 3.2. 更新图表的依赖项。

这会下载添加为依赖项的图表，并且锁定其版本。

```
[student@workstation famouschart]$ helm dependency update
Getting updates for unmanaged Helm repositories...
...Successfully got an update from the "https://charts.bitnami.com/bitnami" chart
repository
Saving 1 charts
Downloading mariadb from repo https://charts.bitnami.com/bitnami
Deleting outdated charts
```

### 3.3. 设置数据库，以便将自定义值来用于身份验证和安全性。

要将相同的值传递给部署的应用，您必须控制它的值，而不是让数据库 Helm 图表随机创建它们。

将下列几行添加到 `values.yaml` 文件的末尾。

```
mariadb:
  auth:
    username: quotes
    password: quotespwd
    database: quotesdb
  primary:
    podSecurityContext:
      enabled: false
    containerSecurityContext:
      enabled: false
```

您可通过将 `~/D0288/labs/multicontainer-helm/mariadb.yaml` 的内容附加到 `values.yaml` 文件来添加此选项：

```
[student@workstation famouschart]$ cat ../mariadb.yaml >> values.yaml
```

## ▶ 4. 使用环境变量来配置应用的数据库访问。

4.1. 默认部署模板不会将任何环境变量传递给部署的应用。修改 `templates/deployment.yaml` 模板，以将 `values.yaml` 文件中定义的环境变量传递给应用容器，再将以下代码片段添加到 `containers` 部分中 `imagePullPolicy` 值的后面：

```
imagePullPolicy: {{ .Values.image.pullPolicy }}
env:
{{- range .Values.env }}
- name: {{ .name }}
  value: {{ .value }}
{{- end }}
```



### 警告

处理 YAML 文件时应确保缩进正确：`imagePullPolicy` 和 `env:` 行必须在同一级别，而 `name` 和 `value` 条目必须在同一级别并且比 `env:` 条目深。- `name` 条目中的 - 必须与 `env:` 条目在同一级别，或比它更深。

4.2. 将适当的环境变量添加到 `values.yaml` 文件的末尾：

```
env:
  - name: "QUOTES_HOSTNAME"
    value: "famousapp-mariadb"
  - name: "QUOTES_DATABASE"
    value: "quotesdb"
  - name: "QUOTES_USER"
    value: "quotes"
  - name: "QUOTES_PASSWORD"
    value: "quotespwd"
```

您可通过将 `~/DO288/labs/multicontainer-helm/env.yaml` 的内容附加到 `values.yaml` 文件来添加此选项：

```
[student@workstation famouschart]$ cat ../env.yaml >> values.yaml
```

## ▶ 5. 使用 Helm 图表部署应用。

### 5.1. 在 RHOCP 中创建项目

```
[student@workstation famouschart]$ source /usr/local/etc/ocp4.config
[student@workstation famouschart]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
[student@workstation famouschart]$ oc new-project \
${RHT_OCP4_DEV_USER}-multicontainer-helm
```

### 5.2. 使用 `helm install` 命令在 RHOCP 集群中部署应用：

```
[student@workstation famouschart]$ helm install famousapp .
NAME: famousapp
LAST DEPLOYED: Thu May 20 19:12:09 2021
NAMESPACE: yourdevuser-multicontainer-helm
STATUS: deployed
REVISION: 1
NOTES:
...output omitted...
```

此命令会创建一个 RHOCP 部署，名为 `famousapp`。要验证此部署的状态，可使用 `oc get deployments` 命令：

```
[student@workstation famouschart]$ oc get deployments
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
famousapp-famouschart   0/1       1           1          10s
```

可能需要片刻之后 `mariadb` 容器集才会完全可供应用使用，因此部署并未达到就绪状态。

运行 `oc get pods` 命令几次，以验证应用和数据库已正确部署：

```
[student@workstation famouschart]$ oc get pods
NAME                           READY   STATUS    RESTARTS   AGE
famousapp-famouschart-7bff544d88-f289l   1/1     Running   3          5m
famousapp-mariadb-0            1/1     Running   0          5m
```

## ▶ 6. 测试应用。

### 6.1. 公开应用的服务：

```
[student@workstation famouschart]$ oc expose service famousapp-famouschart  
route.route.openshift.io/famousapp-famouschart exposed
```

### 6.2. 调用部署的应用的 /random 端点：

```
[student@workstation famouschart]$ FAMOUS_URL=$(oc get route \  
-n ${RHT_OCP4_DEV_USER}-multicontainer-helm famousapp-famouschart \  
-o jsonpath='{.spec.host}'/random)  
[student@workstation famouschart]$ curl $FAMOUS_URL  
8: Those who can imagine anything, can create the impossible.  
- Alan Turing
```

此端点会从数据库中返回一条随机名言，因此您很有可能看到另一条名言。再次执行调用，以验证这种随机行为。

```
[student@workstation famouschart]$ curl $FAMOUS_URL  
1: When words fail, music speaks.  
- William Shakespeare
```

## 完成

在 workstation 虚拟机上，运行 `lab multicontainer-helm finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation famouschart]$ cd ~  
[student@workstation ~]$ lab multicontainer-helm finish
```

本引导式练习到此结束。

# 使用 Kustomize 自定义部署

## Kustomize

Kustomize 这款工具可以针对不同环境或需求自定义 Kubernetes 资源。Kustomize 是一种免模板解决方案，可以帮助重复利用配置和轻松修补任何资源。

最常见用例是满足为开发、暂存和生产等不同环境定义不同资源的需求。但好处不限于此。您可以自己决定用例中要定义的环境、您所拥有的环境，以及每种环境需要的不同配置。

Kustomize 将这些配置集划分为两种类型，即基础和覆盖。

基础类型包含所有派生类型共有的配置和资源。覆盖类型表示特定环境中与基础类型的差别。Kustomize 使用目录来代表这些配置集。

Kustomize 布局可能如下例所示：

```
myapp
└── base
└── overlays
    └── production
    └── staging
```

## Kustomization 文件

对于每一个配置集，Kustomize 都需要一个 `kustomization.yaml` 文件，其可能包含以下内容：

- 要包括的资源文件
- 作为起点的基础配置集
- 对基础配置进行修补的资源
- 应用到基于这一集合的任何配置集的通用资源定义

此文件必须驻留在配置集的目录中。

## 资源文件

`kustomization.yaml` 文件的 `resources` 部分是一个文件列表，这些文件一同创建此特定环境需要的所有资源。例如：

```
resources:
- deployment.yaml
- secrets.yaml
- service.yaml
```

使用这三个资源表示的基础定义的布局如下所示：

```
myapp
└── base
    ├── deployment.yaml
    ├── kustomization.yaml
    ├── secrets.yaml
    └── service.yaml
```

若要维护不是来自单一外部来源的基础集合，将 Kubernetes 对象定义分割成若干个小文件会很有益处。

## 基础配置

覆盖类型的 `kustomization.yaml` 文件需要指向作为起点的基础配置集。例如：

```
bases:
- ../../base
```

如果您遵循预期的目录结构，则相对路径是最为便捷的解决方案。

此覆盖从基础配置起步，然后应用其中定义的任何补丁。

## 通用资源

Kustomize 允许在派生自基础集合的所有资源中添加新配置。

您可使用 `kustomization.yaml` 文件的 `commonLabels` 和 `commonAnnotations` 部分设置标签和注释。

例如，如果要将 `origin=kustomize` 标签添加到基础集合以及依赖于该基础的所有覆盖，您必须将此添加到 `base/kustomization.yaml`：

```
commonLabels:
  origin: kustomize
```

## 补丁

每个覆盖可以借助要应用的补丁列表，提供对基础配置的任意次修改。首先，在覆盖的 `kustomization.yaml` 文件中添加对补丁文件的引用：

```
patches:
- replica_count.yaml
```

然后，在提供的补丁文件中，您需要设法标识要更改的资源和新值：

```
apiVersion: apps/v1
kind: Deployment ①
metadata:
  name: myapp ②
spec:
  replicas: 5 ③
```

① ② `Kind` 和 `name` 提供了一种独特的方式来标识资源

③ 此覆盖中的新值

## 应用自定义

您可以借助 **kustomize** 命令行工具将 Kustomize 用作独立工具；或者，自 Kubernetes 1.14 起，它还可用作 **kubectl apply** 或 **oc apply** 命令的一部分。这些工具只需要有待应用的配置集的文件夹位置。

如下是独立使用这一工具的示例：

```
[user@host ~]$ kustomize build myapp/base | oc apply -f -
```

如下是与集群管理工具集成的示例：

```
[user@host ~]$ oc apply -k myapp/base
```

下方是应用覆盖修改的示例：

```
[user@host ~]$ oc apply -k myapp/overlays/staging
```

Kustomize 仅作为 **kubectl apply** 或 **oc apply** 命令的一部分提供，因为它是对 Kubernetes 对象声明式管理的扩展，设计为与源版本控制系统结合使用。声明式管理方式的工作原理是定义对象的预期状态，并根据对象的存在与否来创建或修改对象。

如果您有可供修改的 Kubernetes 对象源文件，不论原因是您犯了错误还是需要更改某些内容，修改这个文件并使用 **oc apply -f myobject.yaml** 要比删除并重新创建它更加干净利落。更改也会持久保留在文件中，以备将来使用。

Kubernetes 对象命令式管理使用 **kubectl** 和 **oc** 命令，如 **create**、**delete**、**edit** 和 **set**；如果无法执行操作，则命令失败。例如，如果您试图创建已存在的对象，则工具会引发错误。这种管理方式适用于手动操作以及与集群实时交互，而不适用于自动化工具。



### 参考文献

#### Kustomize - Kubernetes 原生配置管理

<https://kustomize.io/>

#### Kubernetes 对象管理

<https://kubernetes.io/docs/concepts/overview/working-with-objects/object-management/>

## ▶ 指导练习

# 使用 Kustomize 自定义部署

在本练习中，您将使用 Kustomize 来自定义红帽 OpenShift 部署。

## 成果

您应能够自定义 Famous Quotes 应用部署，并将其部署到 RHOCP 集群。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 已配置并在运行的 RHOCP 集群。

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令来验证此练习的前提条件。

```
[student@workstation ~]$ lab multicontainer-kustomize start
```

## 说明

您将设置三个环境，即 Development、Staging 和 Production。这三个环境在性能和运行的容器集数量上具有不同的需求。

在下表中查找各个环境的要求：

环境	容器集	内存	CPU 限值
Development	1	128Mi	250m
Staging	2	256Mi	500m
Production	5	512Mi	1000m

### ▶ 1. 查看 Famous Quotes 部署文件。

**famous-quotes.yaml** 文件包含部署“创建 Helm 图表”引导式练习中所用 Famous Quotes 应用需要的全部资源。

```
[student@workstation ~]$ cd ~/D0288/labs/multicontainer-kustomize
[student@workstation multicontainer-kustomize]$ cat famous-quotes.yaml
...
containers:
- name: famouschart
  securityContext:
    {}
  image: "quay.io/redhattraining/famous-quotes:2.1"
...
```

### ▶ 2. 创建 Kustomize 文件夹结构。

## 章 6 | 部署多容器应用

要自定义应用的资源文件，您应该设计文件的结构，以方便理解和引用。

2.1. 创建用来保存所有 Kustomize 文件的文件夹。

```
[student@workstation multicontainer-kustomize]$ mkdir famous-kustomize  
[student@workstation multicontainer-kustomize]$ cd famous-kustomize
```

2.2. 创建基础参考。

使用 `famous-quotes.yaml` 文件，作为 Kustomize 结构的基础。

```
[student@workstation famous-kustomize]$ mkdir base  
[student@workstation famous-kustomize]$ cp ../famous-quotes.yaml \  
base/deployment.yaml
```

2.3. 为基础定义创建 Kustomize 描述文件。

在基础文件夹中创建新的 `kustomization.yaml` 文件，以保存基础 Kustomize 定义。将 `deployment.yaml` 文件作为资源添加到 `base` 文件夹的 `kustomization.yaml` 文件中。

```
resources:  
- deployment.yaml
```

### ▶ 3. 测试基础定义。

要自定义部署，您必须首先验证基础定义是否按预期工作。

3.1. 在 RHOCP 中创建项目。

```
[student@workstation famous-kustomize]$ source /usr/local/etc/ocp4.config  
[student@workstation famous-kustomize]$ oc login -u ${RHT_OCP4_DEV_USER} \  
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}  
[student@workstation famous-kustomize]$ oc new-project \  
${RHT_OCP4_DEV_USER}-multicontainer-kustomize
```

3.2. 应用 Kustomize 基础定义。

使用 `oc apply` 命令应用基础定义：

```
[student@workstation famous-kustomize]$ oc apply -k base  
serviceaccount/famous-quotes-service created  
serviceaccount/famousapp-mariadb created  
configmap/famousapp-mariadb created  
secret/famousapp-mariadb created  
service/famousapp-famouschart created  
service/famousapp-mariadb created  
deployment.apps/famousapp-famouschart created  
statefulset.apps/famousapp-mariadb created  
pod/famousapp-famouschart-test-connection created
```

此命令会创建一个 RHOCP 部署，名为 `famousapp`。要验证此部署的状态，可使用 `oc get deployments` 命令：

## 章 6 | 部署多容器应用

```
[student@workstation famous-kustomize]$ oc get deployments
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
famousapp-famouschart   1/1       1           1          10s
```

可能需要片刻之后 **mariadb** 容器集才会完全可供应用使用，因此请等待应用进入就绪状态。

## 3.3. 公开应用的服务：

```
[student@workstation famous-kustomize]$ oc expose service famousapp-famouschart
route.route.openshift.io/famousapp-famouschart exposed
```

## 3.4. 调用部署的应用的 /random 端点：

```
[student@workstation famous-kustomize]$ FAMOUS_URL=$(oc get route \
-n ${RHT_OCP4_DEV_USER}-multicontainer-kustomize famousapp-famouschart \
-o jsonpath='{.spec.host}'/random)
[student@workstation famous-kustomize]$ curl $FAMOUS_URL
5: Imagination is more important than knowledge.
- Albert Einstein
```

您应该会看到针对各个请求显示的随机名言。

## ▶ 4. 创建 Development 环境定义。

## 4.1. 在 Kustomize 文件夹中创建 Development 覆盖。

```
[student@workstation famous-kustomize]$ mkdir -p overlays/dev
```

## 4.2. 调整 Development 环境的策略。

在 **overlays/dev** 目录中创建 **replica\_limits.yaml** 文件，以存放部署定义中的更改：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: famousapp-famouschart
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: famouschart
      resources:
        limits:
          memory: "128Mi"
          cpu: "250m"
```

4.3. 创建 **overlays/dev/kustomization.yaml** 文件，添加基础文件夹作为基础定义，再添加 **replica\_limits.yaml** 文件作为补丁：

```
bases:  
- ../../base  
patches:  
- replica_limits.yaml
```

4.4. 将 Development 覆盖应用到 Famous Quotes 应用正在运行的实例：

```
[student@workstation famous-kustomize]$ oc apply -k overlays/dev  
serviceaccount/famous-quotes-service unchanged  
serviceaccount/famousapp-mariadb unchanged  
configmap/famousapp-mariadb unchanged  
secret/famousapp-mariadb unchanged  
service/famousapp-famouschart unchanged  
service/famousapp-mariadb configured  
deployment.apps/famousapp-famouschart configured  
statefulset.apps/famousapp-mariadb configured  
pod/famousapp-famouschart-test-connection unchanged
```

4.5. 验证应用是否已正常工作。

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL  
4: Nothing that glitters is gold.  
- Mark Twain
```

4.6. 验证 Kustomize 是否应用了内存限值更改：

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \  
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'  
128Mi
```

## ▶ 5. 创建 Staging 环境。

5.1. 通过复制 Development 环境来创建 Staging 环境：

```
[student@workstation famous-kustomize]$ cp -R overlays/dev overlays/stage
```

5.2. 根据需求表中提供的值，修改 `overlays/stage/replica_limits.yaml` 文件中的配置：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: famousapp-famouschart  
spec:  
  replicas: 2  
  template:  
    spec:  
      containers:  
        - name: famouschart  
          resources:
```

```
limits:  
  memory: "256Mi"  
  cpu: "500m"
```

5.3. 将 Staging 覆盖应用到 Famous Quotes 应用正在运行的实例：

```
[student@workstation famous-kustomize]$ oc apply -k overlays/stage  
serviceaccount/famous-quotes-service unchanged  
serviceaccount/famousapp-mariadb unchanged  
configmap/famousapp-mariadb unchanged  
secret/famousapp-mariadb unchanged  
service/famousapp-famouschart unchanged  
service/famousapp-mariadb configured  
deployment.apps/famousapp-famouschart configured  
statefulset.apps/famousapp-mariadb configured  
pod/famousapp-famouschart-test-connection unchanged
```

5.4. 验证应用是否已正常工作。

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL  
2: Happiness depends upon ourselves.  
- Aristotle
```

5.5. 验证 Kustomize 是否应用了内存限值更改：

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \  
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'  
256Mi
```

## ▶ 6. 创建 Production 环境。

6.1. 通过复制 Development 环境来创建 Production 环境：

```
[student@workstation famous-kustomize]$ cp -R overlays/dev overlays/prod
```

6.2. 根据需求表中提供的值，修改 `overlays/prod/replica_limits.yaml` 文件中的配置：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: famousapp-famouschart  
spec:  
  replicas: 5  
  template:  
    spec:  
      containers:  
        - name: famouschart  
      resources:
```

## 章 6 | 部署多容器应用

```
limits:  
  memory: "512Mi"  
  cpu: "1000m"
```

6.3. 将 Production 覆盖应用到 Famous Quotes 应用正在运行的实例：

```
[student@workstation famous-kustomize]$ oc apply -k overlays/prod  
serviceaccount/famous-quotes-service unchanged  
serviceaccount/famousapp-mariadb unchanged  
configmap/famousapp-mariadb unchanged  
secret/famousapp-mariadb unchanged  
service/famousapp-famouschart unchanged  
service/famousapp-mariadb configured  
deployment.apps/famousapp-famouschart configured  
statefulset.apps/famousapp-mariadb configured  
pod/famousapp-famouschart-test-connection unchanged
```

6.4. 验证应用是否已正常工作。

```
[student@workstation famous-kustomize]$ curl $FAMOUS_URL  
8: Those who can imagine anything, can create the impossible.  
- Alan Turing
```

6.5. 验证 Kustomize 是否应用了内存限值更改：

```
[student@workstation famous-kustomize]$ oc get deployments famousapp-famouschart \  
-o jsonpath='{.spec.template.spec.containers[0].resources.limits.memory}'  
512Mi
```

## 完成

在 workstation 虚拟机上，运行 `lab multicontainer-kustomize finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation famous-kustomize]$ cd ~  
[student@workstation ~]$ lab multicontainer-kustomize finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 部署多容器应用

在本实验中，您将为一个应用创建 Helm 图表，并将其部署到 OpenShift 集群。然后，您将使用 Kustomize 自定义部署的应用。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 为应用创建 Helm 图表。
- 将应用部署到 OpenShift 集群。
- 使用 Kustomize 修改应用部署。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- quay.io/redhattraining/exoplanets:v1.0 中的应用容器镜像。
- 存储库 <https://charts.cockroachdb.com/> 中的 CockroachDB Helm 图表。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还会下载用于检查实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab multicontainer-review start
```

## 要求

Exoplanets 应用是一个 Web 应用，可显示关于其他太阳系行星的一些信息。此应用需要一个 PostgreSQL 兼容数据库（如 CockroachDB）来存储信息，并且它在 Quay 中具有一个预打包的容器镜像。若要访问 Web 应用，可使用指向公开服务的 Web 浏览器。

根据以下要求，创建 Exoplanets 应用的 Helm 图表并将其部署至 OpenShift 集群：

- 使用 **exochart** 作为 Helm 图表名称。
- 使用 quay.io/redhattraining/exoplanets:v1.0 中的应用容器镜像。
- 使用存储库 <https://charts.cockroachdb.com/> 中版本 **6.0.4** 的 **cockroachdb** Helm 图表，作为数据库依赖项。
- 将端口 **8080** 用于应用。

- 使用以下环境变量来配置数据库连接：

### 数据库配置环境变量

变量	值
DB_HOST	exoplanets-cockroachdb
DB_PORT	26257
DB_USER	root
DB_NAME	postgres

- 以 `youruser-multicontainer-review` 为 RHOCP 项目名称。
- 以 `exoplanets` 为 Helm 图表安装名称。
- 以 `exokustom` 为 Kustomize 目录。
- 使用 `helm template` 命令从 Helm 图表创建基础 Kustomize 定义。
- 以 `test` 为 Test 覆盖的目录。
- 将以下资源限值用于 Test 覆盖：

环境	容器集	内存	CPU 限值
Test	5	128Mi	250m

## 说明

- 为名为 `exochart` 的 Exoplanets 应用创建 Helm 图表。
  - 添加数据库依赖项，再使用环境变量配置数据库。
  - 创建一个 OpenShift 项目，再使用 Helm CLI 工具将应用部署到项目中。
  - 测试应用部署。
- 公开应用服务，从应用路由获取地址，然后使用浏览器导航到应用的地址。部署正确的应用与以下镜像类似：

## Exoplanets

The planets listed here are a small subset of the known planets found outside of our solar system. Mass and radius are listed in "Jupiter mass" and "Jupiter radius" units. The orbital period is measured in Earth days. The full dataset is available from the [Open Exoplanet Catalogue](#).

### 2M 0746+20 b

Mass	Radius	Period
30	0.97	4640

### 2M 2140+16 b

Mass	Radius	Period
20	0.92	7340

- 创建基础 Kustomize 目录结构和部署文件。

在`~/DO288/labs/multicontainer-review`目录中创建名为`exokustom`的 Kustomize 目录。



#### 注意

您可以使用`helm template`命令，将 Helm 图表中的对象定义提取到基础定义中：

```
[student@workstation ~]$ helm template app-name helm-directory >
base/deployment.yaml
```

- 使用名为`test`的目录，在 Kustomize 目录中创建并应用 Test 覆盖。
- 在浏览器中测试应用，并验证已应用了自定义值。

## 评估

在`workstation`计算机上，以`student`用户身份使用`lab`命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation exokustom]$ lab multicontainer-review grade
```

## 完成

在`workstation`上运行`lab multicontainer-review finish`命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation exokustom]$ cd  
[student@workstation ~]$ lab multicontainer-review finish
```

本实验到此结束。

## ► 解决方案

# 部署多容器应用

在本实验中，您将为一个应用创建 Helm 图表，并将其部署到 OpenShift 集群。然后，您将使用 Kustomize 自定义部署的应用。



### 注意

各个章节实验结尾处的 **grade** 命令需要您按照实验规范所述，使用正确的项目名称和其他标识符。

## 成果

您应能够：

- 为应用创建 Helm 图表。
- 将应用部署到 OpenShift 集群。
- 使用 Kustomize 修改应用部署。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- quay.io/redhattraining/exoplanets:v1.0 中的应用容器镜像。
- 存储库 <https://charts.cockroachdb.com/> 中的 CockroachDB Helm 图表。

在 **workstation** 上运行以下命令，以验证前提条件。该命令还会下载用于检查实验的帮助程序文件和答案文件：

```
[student@workstation ~]$ lab multicontainer-review start
```

## 要求

Exoplanets 应用是一个 Web 应用，可显示关于其他太阳系行星的一些信息。此应用需要一个 PostgreSQL 兼容数据库（如 CockroachDB）来存储信息，并且它在 Quay 中具有一个预打包的容器镜像。若要访问 Web 应用，可使用指向公开服务的 Web 浏览器。

根据以下要求，创建 Exoplanets 应用的 Helm 图表并将其部署至 OpenShift 集群：

- 使用 **exochart** 作为 Helm 图表名称。
- 使用 quay.io/redhattraining/exoplanets:v1.0 中的应用容器镜像。
- 使用存储库 <https://charts.cockroachdb.com/> 中版本 **6.0.4** 的 **cockroachdb** Helm 图表，作为数据库依赖项。
- 将端口 **8080** 用于应用。
- 使用以下环境变量来配置数据库连接：

### 数据库配置环境变量

变量	值
DB_HOST	exoplanets-cockroachdb
DB_PORT	26257
DB_USER	root
DB_NAME	postgres

- 以 `youruser-multicontainer-review` 为 RHOCP 项目名称。
- 以 `exoplanets` 为 Helm 图表安装名称。
- 以 `exokustom` 为 Kustomize 目录。
- 使用 `helm template` 命令从 Helm 图表创建基础 Kustomize 定义。
- 以 `test` 为 Test 覆盖的目录。
- 将以下资源限值用于 Test 覆盖：

环境	容器集	内存	CPU 限值
Test	5	128Mi	250m

## 说明

1. 为名为 `exochart` 的 Exoplanets 应用创建 Helm 图表。

- 1.1. 创建 `exochart` Helm 图表。

使用 `helm create` 命令创建 Helm 图表，并命名为 `exochart`。

```
[student@workstation ~]$ cd ~/DO288/labs/multicontainer-review
[student@workstation multicontainer-review]$ helm create exochart
Creating exochart
[student@workstation multicontainer-review]$ cd exochart
```

- 1.2. 配置应用容器镜像和版本值。

更新 `values.yaml` 文件，将 `image` 部分的 `repository` 属性设置为 `quay.io/redhattraining/exoplanets` 镜像，然后将同一部分的 `tag` 属性值设置为 `v1.0` 以选择应用的适当版本。

`values.yaml` 文件的对应部分应当如下所示：

```
image:
  repository: quay.io/redhattraining/exoplanets
  pullPolicy: IfNotPresent
  tag: "v1.0"
```

2. 添加数据库依赖项，再使用环境变量配置数据库。

## 章 6 | 部署多容器应用

- 2.1. 将 **cockroachdb** 依赖项添加到应用图表。

为此，可将以下代码片段添加到 **Chart.yaml** 文件的末尾：

```
dependencies:  
- name: cockroachdb  
  version: 6.0.4  
  repository: https://charts.cockroachdb.com/
```

- 2.2. 更新图表的依赖项。

此命令会下载添加为依赖项的图表，并且锁定其版本。

```
[student@workstation exochart]$ helm dependency update  
Getting updates for unmanaged Helm repositories...  
...Successfully got an update from the "https://charts.cockroachdb.com/" chart  
repository  
Saving 1 charts  
Downloading cockroachdb from repo https://charts.cockroachdb.com/  
Deleting outdated charts
```

- 2.3. 默认部署模板不会将任何环境变量传递给部署的应用。修改 **templates/deployment.yaml** 模板，以将 **values.yaml** 文件中定义的环境变量传递给应用容器，再将以下代码片段添加到 **containers** 部分中 **imagePullPolicy** 值的后面：

```
apiVersion: apps/v1  
kind: Deployment  
...output omitted...  
spec:  
  ...output omitted...  
  template:  
    ...output omitted...  
    spec:  
      ...output omitted...  
      containers:  
        - name: {{ .Chart.Name }}  
          ...output omitted...  
          imagePullPolicy: {{ .Values.image.pullPolicy }}  
          env:  
            {{- range .Values.env }}  
            - name: "{{ .name }}"  
              value: "{{ .value }}"  
            {{- end }}  
          ...output omitted...
```

- 2.4. 确保容器使用正确的端口来连接应用。

在 **templates/deployment.yaml** 文件中，更改 **containers** 部分中包含的 **containerPort** 属性，以便它使用值 **8080**。

**templates/deployment.yaml** 文件的对应部分应当如下所示：

```
apiVersion: apps/v1  
kind: Deployment  
...output omitted...
```

## 章 6 | 部署多容器应用

```

spec:
  ...output omitted...
template:
  ...output omitted...
spec:
  ...output omitted...
  containers:
    - name: {{ .Chart.Name }}
      ...output omitted...
  ports:
    - name: http
      containerPort: 8080
      protocol: TCP

```

2.5. 将适当的环境变量添加到 `values.yaml` 文件的末尾：

```

env:
  - name: "DB_HOST"
    value: "exoplanets-cockroachdb"
  - name: "DB_NAME"
    value: "postgres"
  - name: "DB_USER"
    value: "root"
  - name: "DB_PORT"
    value: "26257"

```

3. 创建一个 OpenShift 项目，再使用 Helm CLI 工具将应用部署到项目中。

3.1. 在 OpenShift 中创建一个项目。

```

[student@workstation exochart]$ source /usr/local/etc/ocp4.config
[student@workstation exochart]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
[student@workstation exochart]$ oc new-project \
${RHT_OCP4_DEV_USER}-multicontainer-review

```

3.2. 使用 `helm install` 命令，将应用部署到 OpenShift 集群：

```

[student@workstation exochart]$ helm install exoplanets .
NAME: exoplanets
LAST DEPLOYED: Thu May 20 19:12:09 2021
NAMESPACE: yourdevuser-multicontainer-review
STATUS: deployed
REVISION: 1
NOTES:
...output omitted...

```

验证应用部署是否已正确启动。

```
[student@workstation exochart]$ oc get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
exoplanets-exochart   0/1       1           1          10s
```

等待三个 `cockroachdb` 和应用容器集完全可供应用使用。运行 `oc get pods` 命令几次，以验证应用和数据库已正确部署：

```
[student@workstation exochart]$ oc get pods
NAME                           READY   STATUS    RESTARTS   AGE
exoplanets-cockroachdb-0      1/1     Running   0          1m
exoplanets-cockroachdb-1      1/1     Running   0          1m
exoplanets-cockroachdb-2      1/1     Running   0          1m
exoplanets-exochart-7bff544d88-f289l   1/1     Running   3          1m
```

#### 4. 测试应用部署。

公开应用服务，从应用路由获取地址，然后使用浏览器导航到应用的地址。部署正确的应用与以下镜像类似：

## Exoplanets

The planets listed here are a small subset of the known planets found outside of our solar system. Mass and radius are listed in "Jupiter mass" and "Jupiter radius" units. The orbital period is measured in Earth days. The full dataset is available from the [Open Exoplanet Catalogue](#).

### 2M 0746+20 b

Mass	Radius	Period
30	0.97	4640

### 2M 2140+16 b

Mass	Radius	Period
20	0.92	7340

#### 4.1. 公开应用的服务：

```
[student@workstation exochart]$ oc expose service exoplanets-exochart
route.route.openshift.io/exoplanets-exochart exposed
```

#### 4.2. 在 Web 浏览器中打开部署的应用。

```
[student@workstation exochart]$ firefox $(oc get route exoplanets-exochart \
-o jsonpath='{.spec.host}' \
-n ${RHT_OCP4_DEV_USER}-multicontainer-review ) &
```

5. 创建基础 Kustomize 目录结构和部署文件。

在 ~/DO288/labs/multicontainer-review 目录中创建名为 **exokustom** 的 Kustomize 目录。



### 注意

您可以使用 **helm template** 命令，将 Helm 图表中的对象定义提取到基础定义中：

```
[student@workstation ~]$ helm template app-name helm-directory > base/deployment.yaml
```

- 5.1. 创建 **exokustom** 目录，以存放所有 Kustomize 目录。

```
[student@workstation exochart]$ cd ..  
[student@workstation multicontainer-review]$ mkdir exokustom  
[student@workstation exochart]$ cd exokustom
```

- 5.2. 为基础定义创建目录。

```
[student@workstation exokustom]$ mkdir base
```

- 5.3. 使用 **helm template** 命令，将 Helm 图表中的对象定义提取到基础定义中：

```
[student@workstation exokustom]$ helm template exoplanets \  
.../exochart > base/deployment.yaml
```

- 5.4. 为基础定义创建 Kustomize 描述文件。

在 **base** 目录中创建新的 **kustomization.yaml** 文件，以保存基础 Kustomize 定义。将 **deployment.yaml** 文件作为资源添加到 **base** 目录的 **kustomization.yaml** 文件中。

```
resources:  
- deployment.yaml
```

6. 使用名为 **test** 的目录，在 Kustomize 目录中创建并应用 Test 覆盖。

- 6.1. 在 Kustomize 目录中创建 Test 覆盖。

```
[student@workstation exokustom]$ mkdir -p overlays/test
```

- 6.2. 调整 Test 环境的策略。

在 **overlays/test** 目录中创建 **replica\_limits.yaml** 文件，以存放部署定义中的更改。这些更改必须反映实验要求中定义的资源限制：

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: exoplanets-exochart  
spec:
```

## 章 6 | 部署多容器应用

```
replicas: 5
template:
  spec:
    containers:
      - name: exochart
        resources:
          limits:
            memory: "128Mi"
            cpu: "250m"
```

- 6.3. 创建 `overlays/test/kustomization.yaml` 文件，添加基础目录作为基础定义，再添加 `replica_limits.yaml` 文件作为补丁：

```
bases:
- ../../base
patches:
- replica_limits.yaml
```

- 6.4. 将 Test 覆盖应用到 Exoplanets 应用正在运行的实例：

```
[student@workstation exokustom]$ oc apply -k overlays/test
...output omitted...
serviceaccount/exoplanets-exochart configured
service/exoplanets-cockroachdb-public configured
service/exoplanets-cockroachdb configured
service/exoplanets-exochart configured
deployment.apps/exoplanets-exochart configured
statefulset.apps/exoplanets-cockroachdb configured
...output omitted...
```

此形式的任何警告都可安全忽略：

```
Warning: oc apply should be used on resource created by either oc create --save-config or oc apply
```

7. 在浏览器中测试应用，并验证已应用了自定义值。

- 7.1. 验证应用是否仍然正常工作。

刷新浏览器，您应该仍然会看到前面显示的相同应用。

- 7.2. 验证 Kustomize 是否应用了 CPU 和内存限值更改：

```
[student@workstation exokustom]$ oc get deployments exoplanets-exochart \
-o jsonpath='{.spec.template.spec.containers[0].resources.limits}'
{"cpu":"250m", "memory":"128Mi"}
```

## 评估

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation exokustom]$ lab multiclienter-review grade
```

## 完成

在 `workstation` 上运行 `lab multicontainer-review finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation exokustom]$ cd  
[student@workstation ~]$ lab multicontainer-review finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- 如何将一组红帽 OpenShift 资源转换为模板。
- 如何使用 Helm 图表打包 Kubernetes 应用。
- 使用 Kustomize 针对不同的环境自定义 Kubernetes 资源。
- 如何使用 Kustomize 在不更改应用的前提下自定义打包的 Kubernetes 应用。

## 章 7

# 管理应用部署

### 目标

监控应用的健康状况，并为云原生应用实施各种部署方法。

### 培训目标

- 实施存活度和就绪度探测。
- 为云原生应用选择适合的部署策略。
- 使用 CLI 命令管理应用部署。

### 章节

- 监控应用的健康状况（及引导式练习）
- 选择适合的部署策略（及引导式练习）
- 使用 CLI 命令管理应用部署（及引导式练习）

### 实验

管理应用部署

# 监控应用健康

## 培训目标

学完本节后，您应该能够实施启动、就绪度和存活度探测，以监控应用的就绪情况和健康状况。

## 红帽 OpenShift 容器平台就绪度和存活度探测

应用可能会因为各种原因变得不可靠，例如：

- 暂时丢失连接
- 配置错误
- 应用错误

开发人员可以使用探测来监控其应用。探测使开发人员能够知晓应用状态、资源使用情况和错误等事件。

监控此类事件不仅有助于修复问题，也可帮助进行资源规划和管理。

探测是监控应用健康状态的定期检查。开发人员可以使用 `oc` 命令行工具、YAML 部署模板或红帽 OpenShift Web 控制台来配置探测。

OpenShift 中目前有三种类型的探测：

### 启动探测

启动探测验证容器内的应用是否已经启动。启动探测在所有其他探测之前运行；除非成功完成，否则会禁用其他探测。如果容器的启动探测失败，则容器将被停止，并遵循容器集的 `restartPolicy`。

与定期运行的就绪度探测不同，这种类型的探测仅在启动时执行。

启动探测在容器集配置的 `spec.containers.startupprobe` 属性中进行配置。

### 就绪度探测

就绪度探测确定容器是否准备好为请求服务。如果就绪度探测返回失败状态，红帽 OpenShift 将从所有服务的端点中删除该容器的 IP 地址。

开发人员可以使用就绪度探测向红帽 OpenShift 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。这可用于等待应用执行网络连接、加载文件和缓存，或者执行任何可能需要大量时间并且仅临时影响应用的常见初始任务。

就绪度探测在容器集配置的 `spec.containers.readinessprobe` 属性中进行配置。

### 存活度探测

存活度探测确定容器中运行的应用是否处于 `healthy` 状态。如果存活度探测检测到不健康状态，红帽 OpenShift 将终止该容器并尝试重新部署。

存活度探测在容器集配置的 `spec.containers.livenessprobe` 属性中进行配置。

红帽 OpenShift 提供了五个选项来控制这些探测：

名称	必需	描述	默认值
<code>initialDelaySeconds</code>	是	确定在容器启动后等待多长时间再开始探测。	0
<code>timeoutSeconds</code>	是	确定等待多长时间以便探测完成。如果超过这一时间，红帽 OpenShift 会假定探测失败。	1
<code>periodSeconds</code>	否	指定检查的频率。	1
<code>successThreshold</code>	否	指定在探测失败后应至少连续成功多少次后才将探测视为成功。	1
<code>failureThreshold</code>	否	指定在探测成功后应至少连续失败多少次后才将探测视为失败。	3

## 应用健康状况检查方法

启动、就绪度和存活度探测可以通过三种方式检查应用的健康状况：

### HTTP 检查

HTTP 检查是返回 HTTP 状态代码（例如 REST API）的应用的理想选择。

HTTP 探测使用 GET 请求来检查应用的健康状况。如果 HTTP 响应代码介于 200-399 之间，则检查判定为成功。

下例演示了如何通过 HTTP 检查方法实施就绪度探测：

```
...contents omitted...
readinessProbe:
  httpGet:
    path: /health❶
    port: 8080
  initialDelaySeconds: 15❷
  timeoutSeconds: 1❸
...contents omitted...
```

- ❶ 就绪探测端点。
- ❷ 在容器启动后等待多长时间再检查其健康状况。
- ❸ 等待多长时间以便探测完成。

### 容器执行检查

如果必须根据容器中运行的进程或 shell 脚本的退出代码确定容器状态，则非常适合使用容器执行检查。

使用容器执行检查时，红帽 OpenShift 在容器内执行一条命令。退出检查时状态为 0 视为成功。所有其他状态代码都被视为失败。下例演示了如何实施容器执行检查：

```
...contents omitted...
livenessProbe:
  exec:
    command: ①
      - cat
      - /tmp/health
  initialDelaySeconds: 15
  timeoutSeconds: 1
...contents omitted...
```

- ① 要运行的命令及其参数，作为 YAML 数组。

## TCP 套接字检查

TCP 套接字检查非常适合作为守护进程运行的应用，以及打开的 TCP 端口，例如数据库服务器、文件服务器、Web 服务器和应用服务器。

使用 TCP 套接字检查时，红帽 OpenShift 尝试打开连接容器的套接字。如果检查可以成功建立连接，则容器被视为健康。下例演示了如何通过 TCP 套接字检查方法实施存活度探测：

```
...contents omitted...
livenessProbe:
  tcpSocket:
    port: 8080①
  initialDelaySeconds: 15
  timeoutSeconds: 1
...contents omitted...
```

- ① 要检查的 TCP 端口。

## 使用 Web 控制台管理探测

要创建探测，开发人员可以使用 `oc edit` 或红帽 OpenShift Web 控制台来编辑部署 YAML 文件。

您可以直接从 Web 控制台中的 `Workloads > Deployment > <deployment name>` 页面编辑部署 YAML 文件。单击 `Actions` 下拉菜单，然后选择 `Edit Deployment`。

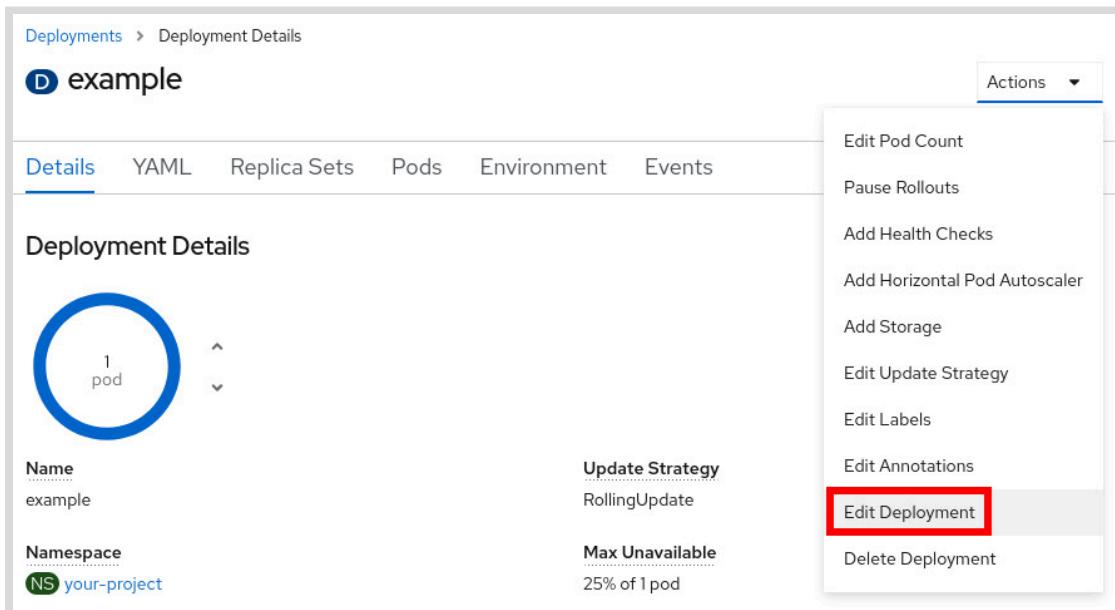


图 7.1: 使用 Web 控制台添加探测

下例显示 Web 控制台中用于部署的 YAML 编辑器。

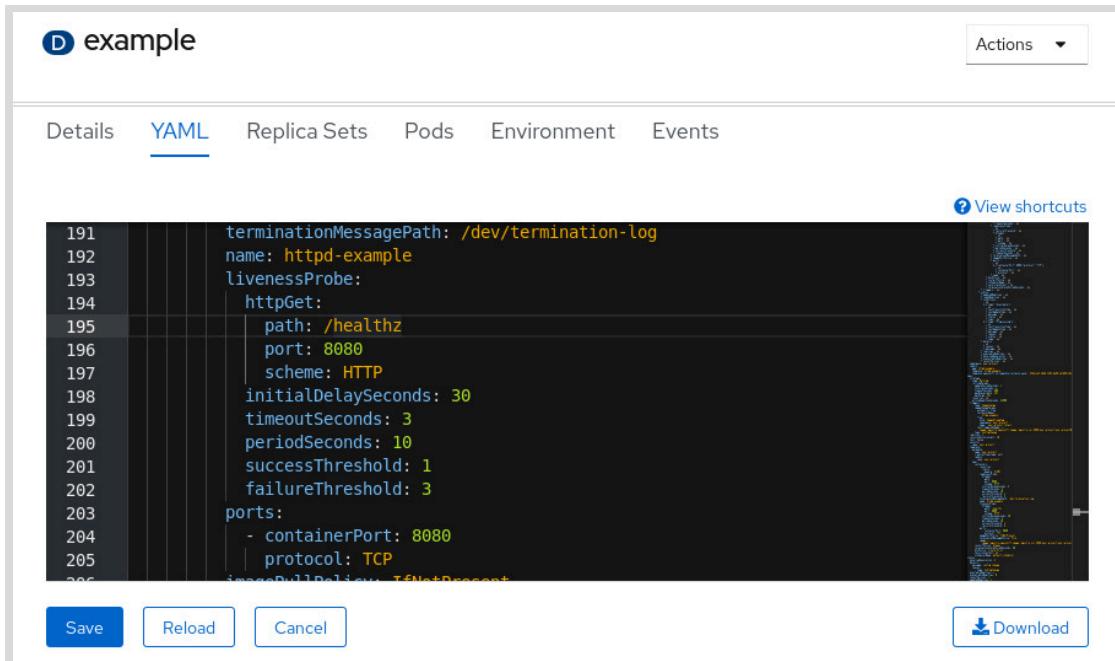


图 7.2: 使用 Web 控制台 YAML 编辑器添加探测

## 使用 CLI 创建探测

`oc set probe` 命令提供了直接编辑部署 YAML 定义的替代方法。在创建用于运行应用的探测时，建议的方法是使用 `oc set probe` 命令，这是因为它可降低您犯错误的几率。此命令提供了许多选项，允许您指定探测类型以及其他必要属性，如端口、URL、超时和期间等。

以下示例演示了使用具有各种选项的 `oc set probe` 命令：

```
[user@host ~]$ oc set probe deployment myapp --readiness \
--get-url=http://:8080/healthz --period=20
```

```
[user@host ~]$ oc set probe deployment myapp --liveness \
--open-tcp=3306 --period=20 \
--timeout-seconds=1
```

```
[user@host ~]$ oc set probe deployment myapp --liveness \
--get-url=http://:8080/healthz --initial-delay-seconds=30 \
--success-threshold=1 --failure-threshold=3
```

使用 `oc set probe --help` 命令查看此命令的所有可用选项。



### 参考文献

如需更多信息，请参阅红帽 OpenShift 容器平台 4.10 Official Documentation 中 Building Applications 一章的 Monitoring application health by using health checks 一节，网址为：  
[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/building\\_applications/index#application-health](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/building_applications/index#application-health)

有关更多信息，请参见 Kubernetes 网站的 Configure Liveness, Readiness and Startup Probes 页面，网址为：  
<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

## ▶ 指导练习

# 激活探测

在本练习中，您将配置存活度和就绪度探测，以监控部署到红帽 OpenShift 集群的应用的健康状况。

在本练习中部署的应用会公开两个 HTTP GET 端点：

- 当应用容器集能够接收请求时，`/healthz` 端点将回复 **200** HTTP 状态代码。

端点指出应用容器集处于健康并可访问的状态。它不指明应用已准备好为请求提供服务。

- 如果整个应用正常工作，则 `/ready` 端点会回复 **200** HTTP 状态代码。

端点会指明应用已准备好为请求提供服务。

在本练习中，`/ready` 端点在应用容器集启动时回复 **200** HTTP 状态代码。`/ready` 端点在部署后前 30 秒回复 **503** HTTP 状态代码，以模拟应用启动缓慢。

您将配置 `/ready` 端点来用于存活度探测，并且配置 `/healthz` 端点用于就绪度探测。

您将模拟红帽 OpenShift 集群中的网络故障，并观察以下情景中的行为：

- 应用不可用。
- 应用可以使用，但无法访问数据库。因此，它无法为请求服务。

## 成果

您应能够：

- 从命令行配置应用的就绪度和存活度探测。
- 在事件日志中查找探测失败消息。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- Node.js S2I 构建器镜像。
- `DO288-apps` Git 存储库中的示例应用 (`probes`)。

在 `workstation` 上运行以下命令，以验证练习前提条件并下载实验文件：

```
[student@workstation ~]$ lab probes start
```

## 说明

- ▶ 1. 创建新项目，然后将 Git 存储库的 `probes` 子目录中的示例应用部署到红帽 OpenShift 集群。

## 第 7 章 | 管理应用部署

1.1. 提供您的课堂环境配置：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户帐户登录红帽 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
```

1.3. 创建新项目：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-probes
```

1.4. 创建一个具有下列参数的新部署：

- 名称：probes
- 构建镜像：nodejs:16-ubi8
- 应用目录：probes
- 构建变量：
  - 名称：npm\_config\_registry，值：http://\${RHT\_OCP4\_NEXUS\_SERVER}/repository/nodejs

您可以使用 `/home/student/D0288/labs/probes/oc-new-app.sh` 脚本复制或执行以下命令：

```
[student@workstation ~]$ oc new-app \
--name probes --context-dir probes --build-env \
npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/nodejs \
nodejs:16-ubi8-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps
--> Success
...output omitted...
```

请注意，`npm_config_registry` 之后的等号 (=) 前后没有空格。

1.5. 查看构建日志。等待构建完成并将应用容器镜像推送到红帽 OpenShift 注册表：

```
[student@workstation ~]$ oc logs -f bc/probes
...output omitted...
Push successful
```

1.6. 等待应用部署好。查看应用容器集的状态。应用容器集应处于 Running 状态：

```
[student@workstation ~]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
probes-1-build        0/1     Completed  0          46s
probes-6cc59f8f98-vxfw9 1/1     Running   0          10s
```

▶ 2. 手动测试应用的 `/ready` 和 `/healthz` 端点。

2.1. 使用路由以公开应用供外部访问:

```
[student@workstation ~]$ oc expose svc probes  
route.route.openshift.io/probes exposed
```

2.2. 测试 /ready 端点:

```
[student@workstation ~]$ curl \  
-i probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}/ready
```

/ready 端点模拟应用的缓慢启动，因此在应用启动后的前 30 秒内，它返回 HTTP 状态代码 503，并返回以下响应：

```
HTTP/1.1 503 Service Unavailable  
...output omitted...  
Error! Service not ready for requests...
```

应用运行 30 秒后，它返回：

```
HTTP/1.1 200 OK  
...output omitted...  
Ready for service requests...
```

2.3. 测试应用的 /healthz 端点:

```
[student@workstation ~]$ curl \  
-i probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}/healthz  
HTTP/1.1 200 OK  
...output omitted...  
OK
```

2.4. 测试应用响应:

```
[student@workstation ~]$ curl \  
probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN}  
Hello! This is the index page for the app.
```

### ▶ 3. 激活应用的就绪度和存活度探测。

3.1. 运行 `oc set` 命令，使用下列参数来配置存活度与就绪度探测：

- 对于存活度探测，请使用 8080 端口上的 /healthz 端点。
- 对于就绪度探测，请使用 8080 端口上的 /ready 端点。
- 对于这两个探测：
  - 将初始延迟配置为 2 秒。
  - 将超时配置为 2 秒。

```
[student@workstation ~]$ oc set probe deployment probes --liveness \
--get-url=http://:8080/healthz \
--initial-delay-seconds=2 --timeout-seconds=2
deployment.apps/probes updated
[student@workstation ~]$ oc set probe deployment probes --readiness \
--get-url=http://:8080/ready \
--initial-delay-seconds=2 --timeout-seconds=2
deployment.apps/probes updated
```

3.2. 验证 livenessProbe 和 readinessProbe 条目中的值：

```
[student@workstation D0288-apps]$ oc describe deployment probes | \
grep -iA 1 liveness
  Liveness: http-get http://:8080/healthz delay=2s timeout=2s period=10s
#success=1 #failure=3
  Readiness: http-get http://:8080/ready delay=2s timeout=2s period=10s
#success=1 #failure=3
```

3.3. 等待应用容器集重新部署好并更改为 READY 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  0/1   Running   0          6s
```

如果 AGE 值不足 30 秒，则 READY 状态将显示 0/1。之后，READY 状态为 1/1：

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  1/1   Running   0          62s
```

3.4. 使用 oc logs 命令查看存活度和就绪度探测的结果：

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc logs -f $POD
...output omitted...
nodejs server running on http://0.0.0.0:8080
ping /healthz => pong [healthy]
ping /ready => pong [notready]
ping /healthz => pong [healthy]
ping /ready => pong [notready]
ping /healthz => pong [healthy]
ping /ready => pong [ready]
...output omitted...
```

观察重新部署后就绪度探测失败约 30 秒，然后成功。回想应用通过在响应就绪状态之前强制设置 30 秒的延迟来模拟应用的缓慢初始化。

不要终止此命令。您将在下一步中继续监视此命令的输出。

#### ► 4. 模拟网络故障。

## 章 7 | 管理应用部署

出现网络故障时，服务会变得不响应。这意味着存活度和就绪度探测都会失败。

红帽 OpenShift 可以在另一个节点上重新创建容器来解决问题。

- 4.1. 在不同的终端窗口或选项卡中，执行 `~/D0288/labs/probes/kill.sh` 脚本以模拟存活度探测失败：

```
[student@workstation ~]$ ~/D0288/labs/probes/kill.sh
Switched app state to unhealthy...
Switched app state to not ready...
```

- 4.2. 返回到监视应用部署的终端：

```
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /healthz => pong [healthy]
Received kill request for health probe.
Received kill request for readiness probe.
ping /healthz => pong [unhealthy]
ping /ready => pong [notready]
ping /ready => pong [notready]
ping /healthz => pong [unhealthy]
ping /healthz => pong [unhealthy]
ping /ready => pong [notready]
npm timing command:run Completed in 118326ms
npm notice
npm notice New minor version of npm available! 8.3.1 -> 8.18.0
npm notice Changelog: https://github.com/npm/cli/releases/tag/v8.18.0
npm notice Run npm install -g npm@8.18.0 to update!
npm notice
npm ERR! path /opt/app-root/src
npm ERR! command failed
npm ERR! signal SIGTERM
npm ERR! command sh -c node app.js
npm timing npm Completed in 118495ms

npm ERR! A complete log of this run can be found in:
npm ERR!     /opt/app-root/src/.npm/_logs/2022-08-31T11_49_50_505Z-debug-0.log
```

存活度探测重复失败后（默认为三个后果性故障），红帽 OpenShift 会重新启动容器集。也就是说，红帽 OpenShift 会在不受网络故障影响的可用节点上重新启动应用。

只有在发出终止请求后立即检查应用日志，您才会看到此日志输出。如果在红帽 OpenShift 重新启动容器集后检查日志，那么日志已被清除，您只能看到下一步中显示的输出。

- 4.3. 验证红帽 OpenShift 是否重新启动了健康状况不佳的容器集。继续检查 `oc get pods` 命令的输出。观察 **RESTARTS** 列并验证计数是否大于零：

```
[student@workstation D0288-apps]$ oc get pods
NAME                  READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  1/1     Running   1          62s
```

等待前面的容器集终止，然后再继续下一步。

4.4. 检查应用日志。存活度探测成功，并且应用报告健康状态。

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /ready => pong [ready]
ping /healthz => pong [healthy]
...output omitted...
```

► 5. 模拟无法访问应用数据库。

如果应用和数据库之间出现网络故障，应用仍然会响应请求，但无法为请求提供服务。也就是说，存活度探测会通过，但就绪度探测会失败。

5.1. 执行 ~/D0288/labs/probes/not-ready.sh 脚本以模拟就绪探测失败：

```
[student@workstation ~]$ ~/D0288/labs/probes/not-ready.sh
Switched app state to not ready...
```

5.2. 检查应用日志。就绪探测返回失败。

```
[student@workstation ~]$ oc logs -f $POD
...output omitted...
ping /ready => pong [notready]
ping /healthz => pong [healthy]
...output omitted...
```

5.3. 检查应用容器集是否不在 READY 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
probes-6cc59f8f98-fsf8x8  0/1     Running   1          78m
```

5.4. 检查服务是否不可用：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
[student@workstation ~]$ curl -is \
probes-${RHT_OCP4_DEV_USER}-probes.${RHT_OCP4_WILDCARD_DOMAIN} \
| grep 'HTTP/1.0'
```

HTTP/1.0 503 Service Unavailable

在生产环境中，红帽 OpenShift 会将请求重新路由到冗余的应用容器集。当应用访问到数据库时，就绪探测会再次通过，容器集则切换到 READY 状态，并且红帽 OpenShift 恢复将流量路由到容器集。

► 6. 验证您是否可以在事件日志中看到探测失败。

对上一步中的容器集使用 `oc describe` 命令：

```
[student@workstation ~]$ POD=$(oc get pods -o name | grep -v build)
[student@workstation ~]$ oc describe $POD
Events:
  Type      Reason     Age   From      Message
  ----      -----     --   --       -----
...output omitted...
  Warning  Unhealthy  ...   ...      Liveness probe failed: ... statuscode: 503
  Normal   Killing    ...   ...      Container probes failed liveness probe, will be
  restarted
...output omitted...
  Warning  Unhealthy  ...   ...      Readiness probe failed: ... statuscode: 503
```

▶ 7. 清理。删除红帽 OpenShift 中的 `probes` 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-probes
```

## 完成

在 `workstation` 上运行 `lab probes finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab probes finish
```

本引导式练习到此结束。

# 选择适合的部署策略

## 培训目标

学完本节后，您应该能够为云原生应用选择适当的部署策略。

## 选择 DeploymentConfig 而非 Deployment

默认情况下，大部分 `oc new-app` 之类红帽 OpenShift 容器平台命令会创建 `Deployment` 资源，这是每个 Kubernetes 分发中都能找到的一流原生 API 资源。不过，红帽 OpenShift 提供了名为 `DeploymentConfig` 的资源。

`DeploymentConfig` 资源使您可以使用其他功能，例如：

- 自定义部署策略
- 生命周期 hook

`Deployment` 资源不支持自定义策略。

您可利用 `Deployment` 资源定义容器生命周期 hook，如 `PostStart` 和 `PreStop`，与 `DeploymentConfig` 资源提供的生命周期 hook 类似。但是，`Deployment` 容器生命周期 hook 不保证会在容器集的 `ENTRYPOINT` 命令之前执行。

因此，这限制了 `Deployment` 容器生命周期 hook 的用例。

如果您不需要 `DeploymentConfig` 资源提供的任何额外功能，或者您的部署必须与 Kubernetes 的其他分发兼容，请使用 `Deployment` 资源。

## 红帽 OpenShift 中的部署策略

部署策略是一种更改或升级应用的方法。目标是以最小的停机时间进行更改或升级，并减少对最终用户的影响。

红帽 OpenShift 提供了若干部署策略。这些策略可以分为两大类别：

- 利用应用部署配置中定义的部署策略。
- 使用红帽 OpenShift 路由器将流量路由到特定的应用容器集。

部署配置中定义的策略会影响使用该应用的所有路由。使用路由器功能的策略会影响单个路由。

下面列出涉及更改部署配置的策略：

### 滚动

滚动策略是默认的策略。

此策略逐步使用新版本应用的实例替换先前版本的应用的实例。此策略执行就绪度探测，以确定新容器集何时就绪。新容器集就绪度探测成功后，部署控制器将缩减旧容器集。

如果发生重大问题，部署控制器会中止滚动部署。开发人员也可使用 `oc rollout cancel` 命令来手动中止滚动部署。

红帽 OpenShift 中的滚动部署是 canary 部署；红帽 OpenShift 在替换所有旧实例之前对新版本 (canary) 进行测试。如果就绪度探测始终不成功，则红帽 OpenShift 会删除 canary 实例，并自动回滚部署配置。

在以下时间使用滚动部署策略：

- 您需要应用更新期间不出现停机。
- 您的应用支持同时运行旧版本和新版本。

### 重新创建

在此策略中，红帽 OpenShift 首先停止当前正在运行的所有容器集，然后仅使用新版本的应用启动容器集。此策略会导致停机，因为在短时间内，您的应用的任何实例都不会运行。

在以下情况下使用重新创建部署策略：

- 您的应用不支持同时运行旧版本和新版本。
- 您的应用使用具有 RWO (ReadWriteOnce) 访问模式的永久卷，不允许从多个容器集进行写入。

### 自定义

如果滚动或重新创建部署策略都不适合您的需求，则您可以使用自定义部署策略来部署应用。有时，要执行的命令需要对系统进行进一步微调（例如，Java 虚拟机的内存），或者您需要将自定义镜像与内部开发的、不面向大众提供的库一起使用。

对于这些类型的用例，请使用自定义策略。您可以在 `DeploymentConfig` 资源的 `spec.strategy.type` 属性中指定该策略。

您可以提供自己的自定义容器镜像，在其中定义部署行为。此自定义镜像在应用的部署配置的 `spec.strategy.customParams.image` 属性中进行定义。您还可以自定义环境变量和要为部署执行的命令。

## 红帽 OpenShift 部署与生命周期 Hook 集成

重新创建和滚动策略支持生命周期 hook。您可以使用这些 hook 在部署过程中的预定义点触发事件。红帽 OpenShift 部署包含三个生命周期 hook：

### 前期生命周期 Hook

启动部署的任何新容器集之前，以及任何旧容器集已关闭之前，红帽 OpenShift 会执行前期生命周期 hook。

### 中期生命周期 Hook

部署中的所有旧容器集已经关闭后，但在启动任何新容器集之前，执行中期生命周期 hook。中期生命周期 Hook 仅适用于 `Recreate` 策略。

### 后期生命周期 Hook

启动部署的所有新容器集之后，并且在所有旧容器集已经关闭之后，执行后期生命周期 hook。

这些生命周期 hook 在 `Strategy` 资源的以下三个属性之一中定义：

- `rollingParams` 用于 `Rolling` 策略。
- `recreateParams` 用于 `Recreate` 策略。
- `customParams` 用于 `Custom` 策略。

您可以添加 `pre`、`mid` 和 `post` 属性，其中包含生命周期 hook。

## 章 7 | 管理应用部署

生命周期 hook 在单独的容器中运行，这些容器是短暂的。红帽 OpenShift 会它们在完成执行后自动进行清理。自动数据库初始化和数据库迁移是生命周期 hook 的良好用例。

每个 hook 都有 `failurePolicy` 属性，定义遇到 hook 故障时要采取的操作。有三个策略：

- 中止：如果 hook 失败，则部署流程被视为失败。
- 重试：重试 hook 执行，直到成功为止。
- 忽略：忽略任何 hook 故障并允许部署继续。

## 使用红帽 OpenShift 路由器实施高级部署策略

下面列出使用红帽 OpenShift 路由器功能的高级部署策略：

### 蓝绿部署

在蓝绿部署中，您有两个同步运行的相同环境，每个环境运行应用的一个不同版本。

红帽 OpenShift 路由器用于将流量从当前的生产中版本（绿色）导向更新的已更新版本（蓝色）。您可以使用路由和两项服务实施此策略。为应用的每个特定版本定义服务。

路由在任何给定时间指向其中一项服务，并且可以在就绪时更改为指向不同服务，或者促进回滚。作为开发人员，您可以在将生产流量路由到新服务之前连接到新服务来测试新版应用。当您的新应用版本生产就绪时，请将生产路由器更改为指向针对您已更新的应用定义的新服务。

### A/B 部署

A/B 部署策略允许您为生产环境中的一组有限用户部署新版本应用。您可以配置红帽 OpenShift，以便它将大多数请求路由到生产环境中当前部署的版本，而有限数量的请求则转到新版本。

通过控制随着测试进行而发送到每个版本的请求部分，可以逐渐增加发送到新版本的请求数。最终，您可以停止将流量路由到以前的版本。在您调整每个版本的请求负载时，可能需要缩放每个服务中的容器集数量以提供预期性能。

此外，也请考虑 N-1 兼容性和安全终止：

## N-1 兼容性

许多部署策略要求两个版本的应用同时运行。当同时运行应用的两个版本时，请确保采用新代码编写的的数据可以由旧版本的代码读取和处理（或正常忽略）；这称为 N-1 兼容性。

应用管理的数据可以采用多种形式：存储在磁盘上、数据库中或临时缓存中的数据。大多数设计良好的无状态 Web 应用都可以支持滚动部署，但测试和设计应用以处理 N-1 兼容性至关重要。

## 正常终止

红帽 OpenShift 允许应用实例在从路由器的负载平衡名单删除之前安全地关闭。应用必须确保它们在退出之前安全地终止用户连接。

红帽 OpenShift 希望关闭它时，向容器中的进程发送 `SIGTERM` 信号。接收 `SIGTERM` 信号时，应用代码应停止接受新连接。停止新连接可确保路由器可以将流量路由到其他活动实例。然后，应用代码应该等所有打开的连接都关闭（或在下次有机会时正常终止单个连接），然后退出。

安全终止期满后，红帽 OpenShift 会向尚未退出的任何进程发送 `SIGKILL` 信号。这会立即结束该进程。容器集或容器集模板的 `terminationGracePeriodSeconds` 属性控制正常终止期（默认为 30 秒），并且可以根据应用进行自定义。



## 参考文献

有关部署策略的更多信息，请参阅红帽 OpenShift 容器平台 4.10 的 Building Applications 指南中的 Deployments 章节，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/building\\_applications/index#deployments](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/building_applications/index#deployments)

如需部署和部署配置资源之间差异的详细说明，请访问

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/building\\_applications/index#what-deployments-are](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/building_applications/index#what-deployments-are)

## 介绍蓝绿、Canary 和滚动部署

<https://opensource.com/article/17/5/colorful-deployments>

如需适用于部署资源的生命周期 hook 的说明，请访问

## 容器生命周期 Hook

<https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/>

## ▶ 指导练习

# 实施部署策略

在本练习中，您将使用部署生命周期 hook 初始化数据库。

## 成果

您应能够：

- 将 MySQL 数据库 **DeploymentConfig** 资源的部署策略更改为 **Recreate**。
- 向 **DeploymentConfig** 资源添加部署后生命周期 hook，以使用来自 SQL 文件的数据初始化 MySQL 数据库。
- 排除故障并修复部署后生命周期执行问题。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- MySQL 8.0 容器镜像 (**rhel8/mysql-80**)。

在 **workstation** 上运行以下命令，以验证练习前提条件并下载实验和答案文件：

```
[student@workstation ~]$ lab strategy start
```

## 说明

▶ 1. 创建新项目，并将基于 **rhel8/mysql-80** 容器镜像的应用部署到红帽 OpenShift 集群。

1.1. 提供课堂环境的配置。

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户名登录红帽 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

1.3. 为应用创建一个新项目。使用您的开发人员用户名，为项目的名称加上前缀。

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-strategy
Now using project "developer-strategy" on server "https://
api.ocp4.example.com:6443".
...output omitted...
```

1.4. 运行 `oc new-app` 命令以使用下列参数创建一个新应用：

- 名称：mysql
- 变量：
  - 名称：MySQL\_USER、value：test
  - 名称：MySQL\_PASSWORD、value：redhat
  - 名称：MySQL\_DATABASE、value：testdb
  - 名称：MySQL\_AI0、value：0
- 容器镜像：registry.redhat.io/rhel8/mysql-80
- 使用 `deploymentconfig` 资源

从 `~/D0288/labs/strategy/oc-new-app.sh` 脚本复制或执行以下命令：

```
[student@workstation ~]$ oc new-app --as-deployment-config \
--name mysql -e MySQL_USER=test -e MySQL_PASSWORD=redhat \
-e MySQL_DATABASE=testdb -e MySQL_AI0=0 \
--image registry.redhat.io/rhel8/mysql-80
--> Found container image ... "registry.redhat.io/rhel8/mysql-80"
...output omitted...
--> Creating resources ...
...output omitted...
--> Success
...output omitted...
```

1.5. 等待 MySQL 容器集部署好。容器集应处于 **READY** 状态：

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
mysql-1-54bhp 1/1     Running   0          11s
mysql-1-deploy 0/1     Completed  0          16s
```

## ▶ 2. 将部署策略更改为 Recreate。

2.1. 验证 MySQL 应用的默认部署策略是否为 **Rolling**：

```
[student@workstation ~]$ oc describe dc/mysql | grep -i strategy:
Strategy: Rolling
```

2.2. 在以下步骤中，您将对部署配置进行多个更改。如需在每次更改后阻止重新部署，请禁用部署的配置更改触发器：

```
[student@workstation ~]$ oc set triggers dc/mysql --from-config --remove
deploymentconfig.apps.openshift.io/mysql triggers updated
```

2.3. 更改默认部署策略。您可以：从 `~/D0288/labs/strategy/recreate.sh` 脚本复制命令、执行脚本，或者键入命令，如下所示：

```
[student@workstation ~]$ oc patch dc/mysql --patch \
'{"spec":{"strategy":{"type":"Recreate"}}}' \
deploymentconfig.apps.openshift.io/mysql patched
```

- 2.4. 从部署配置移除 `rollingParams` 属性。您可以从 `~/D0288/labs/strategy/rm-rolling.sh` 脚本复制或执行命令：

```
[student@workstation ~]$ oc patch dc/mysql --type=json \
-p='[{"op":"remove", "path": "/spec/strategy/rollingParams"}]' \
deploymentconfig.apps.openshift.io/mysql patched
```

► 3. 添加后期生命周期 hook 以初始化 MySQL 数据库的数据。

- 3.1. 检查 `~/D0288/labs/strategy/users.sql` 文件。

此 SQL 脚本会创建名为 `users` 的表，并插入三行数据：

```
CREATE TABLE IF NOT EXISTS users (
    user_id int(10) unsigned NOT NULL AUTO_INCREMENT,
    name varchar(100) NOT NULL,
    email varchar(100) NOT NULL,
    PRIMARY KEY (user_id)) ENGINE=InnoDB DEFAULT CHARSET=utf8;

insert into users(name,email) values ('user1', 'user1@example.com');
insert into users(name,email) values ('user2', 'user2@example.com');
insert into users(name,email) values ('user3', 'user3@example.com');
```

- 3.2. 检查 `~/D0288/labs/strategy/import.sh` 脚本。

该脚本会下载并运行上述 SQL 脚本以初始化数据库。后期生命周期 hook 会下载并执行 `import.sh` 脚本：

```
#!/bin/bash
...output omitted...

echo 'Downloading SQL script that initializes the database...'
curl -s -O https://github.com/RedHatTraining/D0288-apps/releases/download/
OCP-4.1-1/users.sql

echo "Trying $HOOK_RETRIES times, sleeping $HOOK_SLEEP sec between tries:"
while [ "$HOOK_RETRIES" != 0 ]; do

    echo -n 'Checking if MySQL is up...'
    if mysqlshow -h$MYSQL_SERVICE_HOST -u$MYSQL_USER -p$MYSQL_PASSWORD -P3306
$MYSQL_DATABASE &>/dev/null
    then
        echo 'Database is up'
        break
    else
        echo 'Database is down'

        # Sleep to wait for the MySQL pod to be ready
        sleep $HOOK_SLEEP
```

```

fi

let HOOK_RETRIES=$HOOK_RETRIES-1
done

if [ "$HOOK_RETRIES" = 0 ]; then
    echo 'Too many tries, giving up'
    exit 1
fi

# Run the SQL script
if mysql -h$MYSQL_SERVICE_HOST -u$MYSQL_USER -p$MYSQL_PASSWORD -P3306
$MYSQL_DATABASE < /tmp/users.sql
then
    echo 'Database initialized successfully'
else
    echo 'Failed to initialize database'
    exit 2
fi

```

该脚本最多尝试连接数据库 `HOOK_RETRIES` 次，并在尝试间隔期间休眠 `HOOK_SLEEP` 秒。

该脚本将实施重试，因为 hook 容器集与数据库容器集同时启动。因此，数据库容器集必须就绪后，容器集 hook 才能执行 SQL 脚本。

- 3.3. 检查 `~/D0288/labs/strategy/post-hook.sh` 脚本。脚本会向 `DeploymentConfig` 资源添加新的后期生命周期 hook：

```
[student@workstation ~]$ cat ~/D0288/labs/strategy/post-hook.sh
...
oc patch dc/mysql --patch \
'{"spec": {"strategy": {"recreateParams": {"post": {"failurePolicy": "Abort", "execNewPod": {"containerName": "mysql", "command": ["/bin/sh", "-c", "curl -L -s https://github.com/RedHatTraining/D0288-apps/releases/download/OCP-4.1-1/import.sh -o /tmp/import.sh&&chmod 755 /tmp/import.sh&&/tmp/import.sh"]}}}}}'
```

提供的 `import.sh` 和 `users.sql` 本地副本供您参考。Hook 在启动过程中下载和执行文件。

- 3.4. 执行 `~/D0288/labs/strategy/post-hook.sh` 脚本：

```
[student@workstation ~]$ ~/D0288/labs/strategy/post-hook.sh
deploymentconfig.apps.openshift.io/mysql patched
```

#### ▶ 4. 验证修补的部署配置并推出新的部署配置。

- 4.1. 验证部署策略现在是否正处于 `Recreate` 状态，并且具有执行 `import.sh` 脚本的后生命周期 hook：

```
[student@workstation ~]$ oc describe dc/mysql | grep -iA 3 'strategy:'
Strategy: Recreate
Post-deployment hook (pod type, failure policy: Abort):
Container: mysql
Command: /bin/sh -c curl -L -s ...
```

4.2. 强制新部署以测试对策略和新的后期生命周期 hook 的更改：

```
[student@workstation ~]$ oc rollout latest dc/mysql
deploymentconfig.apps.openshift.io/mysql rolled out
```

4.3. 验证新的 MySQL 容器集是否达到 **Running** 状态。然后，**mysql-2-deploy** 和 **mysql-2-hook-post** 容器集达到 **Running** 状态：

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME           READY   STATUS    RESTARTS   ...   NODE
mysql-2-deploy 1/1     Running   0          ...
mysql-2-hook-post 1/1     Running   0          ...
mysql-2-kbnpr   1/1     Running   0          ...
```

几秒钟后，后期生命周期 hook 容器集和第二个部署容器集失败：

NAME	READY	STATUS	RESTARTS	AGE
mysql-1-deploy	0/1	Completed	0	13m
<b>mysql-1-vnq68</b>	<b>1/1</b>	<b>Running</b>	0	114s
mysql-2-deploy	0/1	Error	0	2m11s
mysql-2-hook-post	0/1	Error	0	119s

当 hook 和部署容器集都处于错误状态时，按 **Ctrl+C** 以退出 **watch** 命令。

**mysql-1-vnq68** 容器集是一个新容器集。第二次部署终止了来自第一次部署的原始容器集。第二次部署失败后，使用来自第一次部署的原始部署配置创建一个新容器集。

## ▶ 5. 排除故障并修复后期生命周期 hook。

5.1. 显示来自故障容器集的日志。日志显示脚本只尝试连接了数据库一次，然后返回错误状态：

```
[student@workstation ~]$ oc logs mysql-2-hook-post
Downloading SQL script that initializes the database...
Trying 0 times, sleeping 2 sec between tries:
Too many tries, giving up
```

5.2. 请注意，该脚本的 **HOOK\_RETRIES** 变量值为 0，这不正确，会导致它永远不会连接到数据库。增加脚本尝试连接数据库服务器的次数。将部署配置中的 **HOOK\_RETRIES** 环境变量值设置为 5。

```
[student@workstation ~]$ oc set env dc/mysql HOOK_RETRIES=5
deploymentconfig.apps.openshift.io/mysql updated
```

5.3. 启动第三个部署，以使用环境变量的新值再次运行 hook：

```
[student@workstation ~]$ oc rollout latest dc/mysql
deploymentconfig.apps.openshift.io/mysql rolled out
```

5.4. 一直等到新的后期生命周期 hook 容器集处于 **Completed** 状态：

```
[student@workstation ~]$ watch -n 2 oc get pods
NAME          READY   STATUS    RESTARTS   ...
mysql-1-deploy 0/1     Completed  0          5m26s
mysql-2-deploy 0/1     Error     0          3m30s
mysql-2-hook-post 0/1     Error     0          3m8s
mysql-3-29jwt 1/1     Running   0          57s
mysql-3-deploy 0/1     Completed  0          79s
mysql-3-hook-post 0/1     Completed  0          48s
```

5.5. 打开新终端以显示来自正在运行的后期生命周期 hook 容器集的日志。它们表明脚本可以连接几次数据库，然后返回成功状态：

```
[student@workstation ~]$ oc logs -f mysql-3-hook-post
Downloading SQL script that initializes the database...
Trying 5 times, sleeping 2 sec between tries:
Checking if MySQL is up...Database is up
mysql: [Warning] Using a password on the command line interface can be insecure.
Database initialized successfully
```

尝试次数取决于您的硬件和每个容器集计划由红帽 OpenShift 运行的节点。

如果首次连接了 hook，则当您尝试查看其日志时，容器集将被终止。您可以继续下一步。

5.6. 几秒后，新的数据库容器集是使用对部署配置的最新更改创建的容器集：

NAME	READY	STATUS	RESTARTS	AGE
mysql-1-deploy	0/1	Completed	0	4m29s
mysql-2-deploy	0/1	Error	0	2m2s
mysql-2-hook-post	0/1	Error	0	113s
mysql-3-deploy	0/1	Completed	0	62s
mysql-3-29jwt	1/1	Running	0	49s
mysql-3-hook-post	0/1	Completed	0	45s

**mysql-3-hook-post** 容器集运行并退出后，按 **Ctrl+C** 可退出 **watch** 命令。

请注意，红帽 OpenShift 会保留在之前部署尝试期间创建的故障容器集，以便您可以显示其日志以进行故障排除。

## ▶ 6. 验证新的 MySQL 数据库容器集是否包含来自 SQL 文件的数据。

6.1. 打开 MySQL 容器容器集的 shell 会话。使用 **oc get pods** 命令获取当前 MySQL 容器集的名称：

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
...output omitted...
mysql-3-3p4m1 1/1     Running   0          8m
[student@workstation ~]$ oc rsh mysql-3-3p4m1
sh-4.2$
```

6.2. 验证是否已创建 `users` 表并使用来自 SQL 文件的数据进行填充：

```
sh-4.2$ mysql -u$MYSQL_USER -p$MYSQL_PASSWORD $MYSQL_DATABASE -e "select * from users;"
...output omitted...
+-----+-----+-----+
| user_id | name   | email        |
+-----+-----+-----+
|      1  | user1  | user1@example.com |
|      2  | user2  | user2@example.com |
|      3  | user3  | user3@example.com |
+-----+-----+-----+
```

6.3. 退出 MySQL 会话和容器 shell：

```
sh-4.2$ exit
exit
```

## ▶ 7. 清理。

删除 `strategy` 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-strategy
```

## 完成

在 `workstation` 上运行 `lab strategy finish` 命令以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab strategy finish
```

本引导式练习到此结束。

# 使用 CLI 命令管理应用部署

## 培训目标

学完本节后，您应能够使用 CLI 命令管理应用部署。

## 部署配置

部署配置定义容器集的模板，并且管理新镜像的部署或者属性变化时发生的配置更改。部署配置可以支持许多不同的部署模式，包括完整重启、可自定义滚动更新，以及前期和后期生命周期 hook。

当您创建部署配置时，红帽 OpenShift 会自动创建代表部署配置容器集模板的复制控制器。

部署配置更改时，红帽 OpenShift 将使用最新的容器集模板创建新的复制控制器，并且运行部署流程来向下扩展旧复制控制器并向上扩展新复制控制器。在启动或停止服务负载平衡器和路由器时，红帽 OpenShift 会在其中自动添加和删除应用的实例。

部署配置在资源文件的 **DeploymentConfig** 属性中进行声明，该属性可以是 YAML 或 JSON 格式。使用 **oc** 命令可以像管理任何其他红帽 OpenShift 资源一样管理部署配置。以下模板显示 YAML 格式的部署配置：

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend" ①
spec:
  ...
  replicas: 5 ②
  selector:
    name: "frontend"
  triggers:
    - type: "ConfigChange" ③
    - type: "ImageChange" ④
      imageChangeParams:
        ...
  strategy:
    type: "Rolling"
  ...
```

- ① 部署配置名称。
- ② 要运行的副本数。
- ③ 部署配置发生更改时，配置更改触发器会创建新的复制控制器。
- ④ 镜像更改触发器会在每次有新版本的镜像可用时创建新的复制控制器。

## 使用 CLI 命令管理部署

多个命令行选项可用于管理部署。下表介绍了可用的选项：

## 章 7 | 管理应用部署

- 若要开始部署，可使用 `oc rollout` 命令。`latest` 选项指明必须使用的最新版模板：

```
[user@host ~]$ oc rollout latest dc/name
```

此选项通常用于启动新部署或将应用升级到最新版本。

- 要查看特定部署配置的部署历史记录，请使用 `oc rollout history` 命令：

```
[user@host ~]$ oc rollout history dc/name
```

- 若要访问特定部署的详细信息，可将 `--revision` 参数附加到 `oc rollout history` 命令：

```
[user@host ~]$ oc rollout history dc/name --revision=1
```

- 若要访问部署配置及其最新版本的详细信息，可使用 `oc describe dc` 命令。

```
[user@host ~]$ oc describe dc name
```

- 若要取消部署，可使用 `cancel` 选项运行 `oc rollout` 命令。

```
[user@host ~]$ oc rollout cancel dc/name
```

如果启动时间过长、日志文件中存在不一致或部署正在影响系统中其他资源的行为，则可能需要取消部署。



### 警告

取消后，部署配置将自动回滚到先前运行的复制控制器。

- 若要重试失败的部署配置，可运行 `oc rollout` 命令并附上 `retry` 选项。

```
[user@host ~]$ oc rollout retry dc/name
```

如果要保留相同的修订版本，则可以在以前取消该部署后，或者查找导致部署失败的错误后重试部署配置。



### 注意

重试部署配置时，它会重新启动部署流程，但不会创建新的部署版本。红帽 OpenShift 还会使用与失败时相同的配置重启复制控制器。

- 要使用应用的以前版本，可以使用 `oc rollback` 命令将回滚部署：

```
[user@host ~]$ oc rollback dc/name
```

如果最新部署配置出现问题（例如用户抱怨新功能无法按预期工作），则可以使用 `oc rollback` 命令还原到应用的之前已知工作版本。

**注意**

如果没有通过 `--to-version` 参数指定版本，则使用上一次成功部署的版本。

**注意**

部署配置支持最新部署流程失败时自动回滚到配置的最近一次成功版本。在这种情况下，未能部署的最新模板保持完好并可供检查。

此功能暂时不适用于 **Deployment** 资源，您需要通过设置标签来重新部署构建。处理 **Deployment** 资源时，您可以通过使用 `oc describe imagestream name` 来查找哈希，然后继续使用 `oc tag imagename:latest hash-from-older-build` 命令将哈希设置为标签。

- 为了防止回滚完成后意外启动新的部署流程，镜像更改触发器在回滚过程中被禁用。

但在回滚后，您可以通过 `oc set triggers` 命令重新启用镜像更改触发器：

```
[user@host ~]$ oc set triggers dc/name --auto
```

- 若要查看部署日志，请使用 `oc logs` 命令。

```
[user@host ~]$ oc logs -f dc/name
```

如果最新版本正在运行或失败，则 `oc logs` 命令将返回负责部署容器集的进程的日志。如果成功，它将从应用的容器集中返回日志。

您还可以查看较旧的失败部署进程中的日志，前提是它们尚未被手动修剪或删除：

```
[user@host ~]$ oc logs --version=1 dc/name
```

- 您可以使用 `oc scale` 命令扩展部署中的容器集数量：

```
[user@host ~]$ oc scale dc/name --replicas=3
```

副本的数量最终会传播到部署配置所配置的所需和当前状态。

## 部署触发器

部署配置可以包含触发器，它们会驱动创建新部署以响应红帽 OpenShift 内部和外部的事件。触发部署的事件有两种：

- 配置更改
- 镜像更改

## 配置更改触发器

**ConfigChange** 触发器会在检测到部署配置控制器模板更改时触发新部署。您可以依赖此触发器在更改副本大小、更改镜像以用于应用或对部署配置进行其他更改后激活。

**ConfigChange** 触发器的示例如下所示：

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

## 镜像更改触发器

**ImageChange** 触发器会在镜像流标签值更改时触发新部署。这在出于安全原因或库更新的原因独立于应用代码更新镜像的环境中非常有用。

**ImageChange** 触发器的示例如下所示：

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true①
      containerNames:
        - "helloworld"
    from:
      kind: "ImageStreamTag"
      name: "origin-ruby-sample:latest"
```

- ① 如果 **automatic** 属性设置为 `false`，则触发器将禁用。

在上例中，如果 `origin-ruby-sample` 镜像流的 `latest` 标签值有变化，则使用容器的新标签值创建新的部署。

使用 `oc set triggers` 命令为部署配置设置部署触发器。例如，若要设置 **ImageChange** 触发器，可以运行以下命令：

```
[user@host ~]$ oc set triggers dc/name \
--from-image=myproject/origin-ruby-sample:latest -c helloworld
```

## 设置部署资源限制

部署由节点上消耗资源（内存和 CPU）的容器集完成。默认情况下，容器集使用无限的节点资源。但是，如果项目指定了默认资源限制，则容器集最多仅消耗达到这些限制的资源。

您还可以将资源限制指定为部署策略的一部分，以限制资源使用。这些资源限制适用于部署创建的应用容器集，但不适用于部署程序容器集。您可以将部署资源与重新创建、滚动或自定义部署策略一起使用。

在下例中，部署所需的资源在部署配置的 **resources** 属性下进行声明：

```
type: "Recreate"
resources:
  limits:
    cpu: "100m" ①
    memory: "256Mi" ②
```

- ① CPU 装置中的 CPU 资源。`100m` 等于 0.1 CPU 单位。

- ② 内存资源以字节为单位。`256Mi` 等于  $268435456$  字节 ( $256 * 2^{20}$ )。



### 参考文献

有关部署的更多信息，请参阅红帽 OpenShift 容器平台 4.10 的 Building Applications 指南中的 Deployments 章节，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/building\\_applications/index#deployments](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/building_applications/index#deployments)

## ▶ 指导练习

# 管理应用部署

在本练习中，您将管理在红帽 OpenShift 集群上运行的应用的部署。

## 成果

您应能够：

- 在红帽 OpenShift 集群中部署基于 Thorntail 的应用。
- 更新正在运行的应用的部署配置，以包括存活度探测。
- 对应用源进行更改并重新部署应用。
- 将应用回滚到先前部署的版本。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 正在运行的红帽 OpenShift 集群。
- `redhat-openjdk-18/openjdk18-openshift` OpenJDK S2I 构建器镜像
- Git 存储库中的 `quip` 应用。

在 `workstation` 上运行以下命令，以验证练习前提条件并下载实验和答案文件：

```
[student@workstation ~]$ lab app-deploy start
```

## 说明

### ▶ 1. 查看应用源代码。

- 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `master` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
```

- 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b app-deploy
Switched to a new branch 'app-deploy'
[student@workstation D0288-apps]$ git push -u origin app-deploy
...output omitted...
* [new branch]      app-deploy -> app-deploy
Branch app-deploy set up to track remote branch app-deploy from origin.
[student@workstation D0288-apps]$ cd
```

- 1.3. 检查 ~/DO288-apps/quip/src/main/java/com/redhat/training/example/Quip.java 文件。

quip 应用是一个 Java JAX-RS REST 服务实施，具有两个端点：

```
...output omitted...
@Path("/")
public class Quip {

    @GET
    @Produces("text/plain")
    public Response index() throws Exception {
        String host = InetAddress.getLocalHost().getHostName();
        return Response.ok("Veni, vidi, vici...\n").build();①
    }

    @GET
    @Path("/ready")
    @Produces("text/plain")
    public Response ready() throws Exception {
        return Response.ok("OK\n").build();②
    }
...output omitted...
```

① 对 / 端点的请求返回一条引言。

② 对 /ready 端点的请求返回一条 OK 消息。

## ▶ 2. 基于源代码创建应用。

- 2.1. 提供您的课堂环境配置：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 2.2. 使用您的开发人员用户帐户登录红帽 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.3. 为应用创建一个新项目。使用您的开发人员用户名，为项目的名称加上前缀：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-app-deploy
```

- 2.4. 使用下列参数创建一个应用：

- 名称：quip
- 构建环境变量：
  - 名称：MAVEN\_MIRROR\_URL，值：`http://${RHT_OCP4_NEXUS_SERVER}/repository/java`

## 章 7 | 管理应用部署

- 镜像流: `redhat-openjdk18-openshift:1.8`
- 应用目录: `/quip`

您可以执行 `/home/student/D0288/labs/app-deploy/oc-new-app.sh` 脚本，  
也可以执行以下命令：

```
[student@workstation ~]$ oc new-app --as-deployment-config --name quip \
--build-env MAVEN_MIRROR_URL=http:// ${RHT_OCP4_NEXUS_SERVER}/repository/java \
-i redhat-openjdk18-openshift:1.8 --context-dir quip \
https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#app-deploy
--> Found image c1bf724 (9 months old) in image stream "openshift/redhat-
openjdk18-...
--> Creating resources ...
imagestream.image.openshift.io "quip" created
buildconfig.build.openshift.io "quip" created
deploymentconfig.apps.openshift.io "quip" created
service "quip" created
--> Success
...output omitted...
```

2.5. 查看应用构建日志。构建应用容器镜像并将其推送到红帽 OpenShift 内部注册表将需要一些时间：

```
[student@workstation ~]$ oc logs -f bc/quip
...output omitted...
[INFO] Repackaged .war: /tmp/src/target/quip.war
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
...output omitted...
Pushing image ...-registry.svc:5000/youruser-app-deploy/quip:latest ...
...output omitted...
Push successful
```

2.6. 等待应用部署好。应用容器集必须处于 **READY** 状态：

```
[student@workstation ~]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
quip-1-59j8q   1/1     Running   0          10s
quip-1-build   0/1     Completed  0          89s
quip-1-deploy  0/1     Completed  0          12s
```

### ► 3. 测试应用以验证它是否服务来自客户端的请求。

3.1. 检查应用日志以查看启动过程中是否存在任何错误：

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
quip-1-59j8q  1/1     Running   0          50s
...output omitted...
[student@workstation ~]$ oc logs quip-1-59j8q
...output omitted...
... INFO [org.jboss.as.server] (main) WFLYSRV0010: Deployed "quip.war" (...)
... INFO [org.wildfly.swarm] (main) WFSWARM99999: Thorntail is Ready
```

日志显示应用启动时没有任何错误。

### 3.2. 验证应用的 Service 资源具有路由传入请求的已注册端点：

```
[student@workstation ~]$ oc describe svc/quip
Name:           quip
...output omitted...
IP:             172.30.37.127
Port:           8080-tcp  8080/TCP
TargetPort:     8080/TCP
Endpoints:     10.128.2.111:8080
...output omitted...
```

### 3.3. 使用路由公开应用供外部访问：

```
[student@workstation ~]$ oc expose svc quip
route.route.openshift.io/quip exposed
```

### 3.4. 使用您在上一步中获取的路由 URL 测试应用：

```
[student@workstation ~]$ curl \
http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
Veni, vidi, vici...
```

## ▶ 4. 激活应用的就绪度和存活度探测。

### 4.1. 使用 oc set 命令添加存活度和就绪度探测至 DeploymentConfig 资源，两个探测的参数如下所示：

- 端点： /ready
- 端口： 8080
- 初始延迟： 30 秒
- 超时： 2 秒

```
[student@workstation ~]$ oc set probe dc/quip \
--liveness --readiness --get-url=http://:8080/ready \
--initial-delay-seconds=30 --timeout-seconds=2
deploymentconfig.apps.openshift.io/quip probes updated
```

### 4.2. 验证 livenessProbe 和 readinessProbe 条目中的值：

## 第 7 章 | 管理应用部署

```
[student@workstation ~]$ oc describe dc/quip | grep http-get
  Liveness:      http-get http://:8080/ready delay=30s timeout=2s period=10s
#success=1 #failure=3
  Readiness:     http-get http://:8080/ready delay=30s timeout=2s period=10s
#success=1 #failure=3
```

4.3. 等待应用容器集重新部署好并显示为 READY 状态：

```
[student@workstation ~]$ oc get pods
...output omitted...
quip-2-n6nzw    1/1      Running     0          26s
```

4.4. 使用 `oc describe` 命令检查正在运行的容器集并确保探测处于激活状态：

```
[student@workstation ~]$ oc describe pod quip-2-n6nzw | grep http-get
  Liveness:      http-get http://:8080/ready delay=30s timeout=2s...
  Readiness:     http-get http://:8080/ready delay=30s timeout=2s...
```

应用的就绪度和存活度探测现在处于激活状态。

4.5. 使用您从前一步获取的路由 URL 测试应用：

```
[student@workstation ~]$ curl \
http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
Veni, vidi, vici...
```

## ▶ 5. 对应用源进行更改并重新部署应用。

验证测试应用时您是否可以看到更改。

5.1. 检查 `~/D0288/labs/app-deploy/app-change.sh` 脚本。

脚本会更改用来打印英文消息的源代码，然后提交更改并推送到课堂 Git 存储库。

```
[student@workstation ~]$ cat ~/D0288/labs/app-deploy/app-change.sh
#!/bin/bash

echo "Changing quip to english..."
sed -i 's/Veni, vidi, vici/I came, I saw, I conquered/g' \
/home/student/D0288-apps/quip/src/main/java/com/redhat/training/example/Quip.java

echo "Committing the changes..."
cd /home/student/D0288-apps/quip
git commit -a -m "Changed quip lang to english"

echo "Pushing changes to classroom Git repository..."
git push
cd
```

5.2. 执行 `~/D0288/labs/app-deploy/app-change.sh` 脚本：

```
[student@workstation ~]$ ~/D0288/labs/app-deploy/app-change.sh  
Changing quip to english...  
Committing the changes...  
[app-deploy afdf7c3] Changed quip lang to english  
...output omitted...  
To https://github.com/youruser/D0288-apps  
 dfe07f7..0aa1ac1 app-deploy -> app-deploy
```

5.3. 启动应用新构建并按照构建日志进行操作：

```
[student@workstation ~]$ oc start-build quip -F  
build.build.openshift.io/quip-2 started  
...output omitted...  
Push successful
```

5.4. 等待新的应用容器集部署好。容器集必须处于 **READY** 状态：

```
[student@workstation ~]$ oc get pods  
NAME      READY   STATUS    RESTARTS   AGE  
quip-1-build  0/1     Completed  0          12m  
quip-1-deploy 0/1     Completed  0          11m  
quip-2-build  0/1     Completed  0          2m11s  
quip-2-deploy 0/1     Completed  0          4m59s  
quip-3-deploy 0/1     Completed  0          60s  
quip-3-gvzs5  1/1     Running   0          56s
```

5.5. 更改后重新测试应用，并验证是否已打印英文消息：

```
[student@workstation ~]$ curl \  
http://quip-${RHT_OCP4_DEV_USER}-app-deploy.${RHT_OCP4_WILDCARD_DOMAIN}  
I came, I saw, I conquered...
```

## ▶ 6. 回滚到上一个部署。

验证是否看到前面的 **Veni, vidi, vici...** 消息。

6.1. 回滚到应用以前版本的部署。

您将看到一条警告消息，指出 **oc rollback** 命令禁用了镜像更改触发器。

```
[student@workstation ~]$ oc rollback dc/quip  
deploymentconfig.apps.openshift.io/quip deployment #4 rolled back to quip-2  
Warning: the following images triggers were disabled: quip:latest  
You can re-enable them with: oc set triggers dc/quip --auto
```

6.2. 等待新的应用容器集部署。容器集必须处于 **READY** 状态：

```
[student@workstation ~]$ oc get pods  
NAME      READY   STATUS    RESTARTS   AGE  
quip-1-build  0/1     Completed  0          17m  
quip-1-deploy 0/1     Completed  0          16m
```

quip-2-build	0/1	Completed	0	7m1s
quip-2-deploy	0/1	Completed	0	9m49s
quip-3-deploy	0/1	Completed	0	5m50s
quip-4-9h6ql	1/1	Running	0	89s
quip-4-deploy	0/1	Completed	0	95s

6.3. 回滚应用后，重新进行测试并验证是否已打印拉丁语消息：

```
[student@workstation ~]$ curl \
http://quip-$\{RHT_OCP4_DEV_USER\}-app-deploy.\$\{RHT_OCP4_WILDCARD_DOMAIN\}
Veni, vidi, vici...
```

#### ► 7. 清理。删除项目：

```
[student@workstation ~]$ oc delete project $\{RHT_OCP4_DEV_USER\}-app-deploy
project.project.openshift.io "youruser-app-deploy" deleted
```

## 完成

在 `workstation` 上运行 `lab app-deploy finish` 命令以完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab app-deploy finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 管理应用部署

在本实验中，您将管理应用的部署，并在红帽 OpenShift 容器平台 (RHOC) 集群上进行扩展。

## 成果

您应能够：

- 将基于 PHP 的应用部署到红帽 OpenShift 集群。
- 将应用缩放为在多个容器集中运行。
- 修改应用源、重新部署应用，并验证更改是否已反映。
- 回滚更改并验证是否已部署先前版本的应用。

## 在你开始之前

要进行此实验，请确保您具有以下资源的访问权限：

- 正在运行的红帽 OpenShift 集群。
- DO288-apps Git 存储库中的 `php-scale` 应用源代码。

在 `workstation` 上运行以下命令，以验证前提条件：

```
[student@workstation ~]$ lab manage-deploy start
```

## 要求

本实验使用基于 PHP 的应用来打印以下信息：

- 应用的版本
- 应用容器集的名称
- 应用容器集的 IP 地址

根据以下说明在红帽 OpenShift 集群上部署并测试应用：

- 使用 `php:7.3` 镜像流来部署应用。
- 确保红帽 OpenShift 的应用名称为 `scale`。
- 在名为 `youruser-manage-deploy` 的项目中创建应用。
- 确保可以通过 URL 访问应用：  
`http://scale-youruser-manage-deploy.apps.cluster.domain.example.com`。
- 确保应用在以下位置使用 Git 存储库：  
`https://github.com/youruser/DO288-apps`。

## 章 7 | 管理应用部署

- Git 存储库中的 `php-scale` 目录包含应用的源代码。
- 确保应用使用 `DeploymentConfig` 资源。

## 说明

1. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `master` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps  
[student@workstation D0288-apps]$ git checkout main  
...output omitted...
```

2. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b manage-deploy  
Switched to a new branch 'manage-deploy'  
[student@workstation D0288-apps]$ git push -u origin manage-deploy  
...output omitted...  
 * [new branch]      manage-deploy -> manage-deploy  
Branch manage-deploy set up to track remote branch manage-deploy from origin.
```

3. 使用您的个人开发人员用户名登录红帽 OpenShift 集群。

新建一个名为 `youruser-manage-deploy` 的项目。

使用 `php:7.3` 镜像流，在 `php-scale` 目录中部署应用。将应用命名为 `scale`，并使用 `DeploymentConfig` 资源管理该应用。

部署应用时，请确保引用在上一步中创建的 `manage-deploy` 分支。

使用生成的路由 URL 公开并测试应用。验证您是否可以在输出中看到 `version 1` 和容器集名称。

4. 验证 `Rolling` 策略是否为默认部署策略。

5. 将应用缩放为两个容器集：

使用 `curl` 命令重新测试应用，并验证请求是否在两个容器集之间进行循环负载平衡。

6. 将 `index.php` 文件的版本号更改为 `2`。提交更改并推送到远程 Git 存储库。

不要对源代码进行任何其他更改。

7. 启动应用新构建。

验证红帽 OpenShift 是否缩小运行旧版本的容器集，并使用最新版本的应用扩展两个新容器集。

使用 `curl` 命令重新测试应用。验证输出中是否显示 `version 2`。

8. 回滚到上一个部署。

使用 `curl` 该命令重新测试应用。验证输出中是否显示 `version 1`。

9. 对您的作业进行评分：

```
[student@workstation D0288-apps]$ lab manage-deploy grade
```

10. 清理并删除项目。

```
[student@workstation D0288-apps]$ oc delete project \
${RHT_OCP4_DEV_USER}-manage-deploy
```

## 完成

在 **workstation** 上，执行 **lab manage-deploy finish** 脚本来完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab manage-deploy finish
```

本复习实验到此结束。

## ► 解决方案

# 管理应用部署

在本实验中，您将管理应用的部署，并在红帽 OpenShift 容器平台 (RHOCP) 集群上进行扩展。

## 成果

您应能够：

- 将基于 PHP 的应用部署到红帽 OpenShift 集群。
- 将应用缩放为在多个容器集中运行。
- 修改应用源、重新部署应用，并验证更改是否已反映。
- 回滚更改并验证是否已部署先前版本的应用。

## 在你开始之前

要进行此实验，请确保您具有以下资源的访问权限：

- 正在运行的红帽 OpenShift 集群。
- DO288-apps Git 存储库中的 **php-scale** 应用源代码。

在 **workstation** 上运行以下命令，以验证前提条件：

```
[student@workstation ~]$ lab manage-deploy start
```

## 要求

本实验使用基于 PHP 的应用来打印以下信息：

- 应用的版本
- 应用容器集的名称
- 应用容器集的 IP 地址

根据以下说明在红帽 OpenShift 集群上部署并测试应用：

- 使用 **php:7.3** 镜像流来部署应用。
- 确保红帽 OpenShift 的应用名称为 **scale**。
- 在名为 **youruser-manage-deploy** 的项目中创建应用。
- 确保可以通过 URL 访问应用：  
<http://scale-youruser-manage-deploy.apps.cluster.domain.example.com>。
- 确保应用在以下位置使用 Git 存储库：  
<https://github.com/youruser/DO288-apps>。

## 章 7 | 管理应用部署

- Git 存储库中的 `php-scale` 目录包含应用的源代码。
- 确保应用使用 `DeploymentConfig` 资源。

## 说明

1. 进入 `D0288-apps` Git 存储库的本地克隆，并签出课程存储库的 `master` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd D0288-apps  
[student@workstation D0288-apps]$ git checkout main  
...output omitted...
```

2. 创建一个新分支，以用于保存您在本练习中所做的任何更改：

```
[student@workstation D0288-apps]$ git checkout -b manage-deploy  
Switched to a new branch 'manage-deploy'  
[student@workstation D0288-apps]$ git push -u origin manage-deploy  
...output omitted...  
* [new branch]      manage-deploy -> manage-deploy  
Branch manage-deploy set up to track remote branch manage-deploy from origin.
```

3. 使用您的个人开发人员用户名登录红帽 OpenShift 集群。

新建一个名为 `youruser-manage-deploy` 的项目。

使用 `php:7.3` 镜像流，在 `php-scale` 目录中部署应用。将应用命名为 `scale`，并使用 `DeploymentConfig` 资源管理该应用。

部署应用时，请确保引用在上一步中创建的 `manage-deploy` 分支。

使用生成的路由 URL 公开并测试应用。验证您是否可以在输出中看到 `version 1` 和容器集名称。

3.1. 提供您的课堂环境配置：

```
[student@workstation D0288-apps]$ source /usr/local/etc/ocp4.config
```

3.2. 登录红帽 OpenShift 并创建一个新项目。使用您的开发人员用户名，为项目的名称加上前缀。

```
[student@workstation D0288-apps]$ oc login -u ${RHT_OCP4_DEV_USER} \  
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}  
Login successful.  
...output omitted...  
[student@workstation D0288-apps]$ oc new-project \  
${RHT_OCP4_DEV_USER}-manage-deploy  
Now using project "yourname-manage-deploy" on server "https://  
api.cluster.domain.example.com:6443".  
...output omitted...
```

3.3. 创建新应用：

## 章 7 | 管理应用部署

```
[student@workstation D0288-apps]$ oc new-app --as-deployment-config --name scale \
php:7.3-https://github.com/${RHT_OCP4_GITHUB_USER}/D0288-apps#manage-deploy \
--context-dir php-scale
--> Found image f10275b (3 weeks old) in image stream "openshift/php" under...
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "scale" created
buildconfig.build.openshift.io "scale" created
deploymentconfig.apps.openshift.io "scale" created
service "scale" created
--> Success
...output omitted...
```

3.4. 查看构建日志。等待构建完成并将应用容器镜像推送到红帽 OpenShift 注册表：

```
[student@workstation D0288-apps]$ oc logs -f bc/scale
...output omitted...
Push successful
```

3.5. 等待应用部署好。应用容器集应处于 READY 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
scale-1-build  0/1     Completed  0          5m14s
scale-1-deploy  0/1     Completed  0          82s
scale-1-w48nd  1/1     Running   0          74s
```

3.6. 使用路由以公开应用供外部访问：

```
[student@workstation D0288-apps]$ oc expose svc scale
route.route.openshift.io/scale exposed
```

3.7. 使用 curl 命令测试应用：

```
[student@workstation D0288-apps]$ curl \
http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-w48nd (10.128.1.21)
```

4. 验证 Rolling 策略是否为默认部署策略。

```
[student@workstation D0288-apps]$ oc describe dc/scale | grep -i strategy
Strategy: Rolling
```

5. 将应用缩放为两个容器集：

使用 curl 命令重新测试应用，并验证请求是否在两个容器集之间进行循环负载平衡。

5.1. 在 Web 浏览器中打开红帽 OpenShift 控制台 URL：

```
[student@workstation D0288-apps]$ oc whoami --show-console
https://console.openshift-console.apps.ocp4.example.com
```

## 章 7 | 管理应用部署

- 5.2. 以 developer 用户身份并使用密码 developer 进行登录。
- 5.3. 如果您不在管理员视图中，请单击 Developer > Administrator 切换到管理员视图。  
单击 Projects 页面上的 developer-manage-deploy 可打开项目的 Overview 页面。  
单击 Workloads 选项卡，以显示项目可使用的部署。
- 5.4. 选择 scale 部署配置条目。  
单击 Details 部分。然后，单击蓝色圆圈右侧的向上箭头，将容器集数量增加为两个。

The screenshot shows the OpenShift Project details interface. The 'Workloads' tab is selected. A DeploymentConfig named 'scale' is listed with '1 of 1 Pod'. A modal window for 'DC scale' is open, showing a 'Health checks' section with a note about container scale not having health checks to ensure application correctness, and a 'Details' tab showing 1 Pod.

观察红帽 OpenShift 创建另一个容器集。这可能需要几分钟时间。

- 5.5. 返回到终端窗口，并使用 curl 命令对路由 URL 发出多个 HTTP 请求以测试应用：

```
[student@workstation D0288-apps]$ curl \
http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-gp3w0 (10.128.1.21)
[student@workstation D0288-apps]$ curl \
http://scale-${RHT_OCP4_DEV_USER}-manage-deploy.${RHT_OCP4_WILDCARD_DOMAIN}
This is version 1 of the app. I am running on host -> scale-1-567x7 (10.129.1.101)
```

您应该看到请求在两个容器集之间进行循环负载均衡。



### 注意

您无法使用 Web 浏览器来测试路由 URL，因为红帽 OpenShift 路由器默认启用会话粘性。因此，您无法验证负载平衡行为。使用 curl 命令测试应用。

6. 将 `index.php` 文件的版本号更改为 2。提交更改并推送到远程 Git 存储库。

不要对源代码进行任何其他更改。

- 6.1. 编辑 `~/D0288-apps/php-scale/index.php` 文件并将版本号（在第二行）更改为 2，如下所示。不要更改此文件中的任何其他内容：

```
<?php  
print "This is version 2 of the app. I am running on host...  
?>
```

- 6.2. 提交更改并推送到 Git 存储库：

```
[student@workstation D0288-apps]$ git commit -a -m "Updated app to version 2"  
[manage-deploy 3633f74] updating version  
 1 file changed, 1 insertion(+), 1 deletion(-)  
[student@workstation D0288-apps]$ git push  
...output omitted...
```

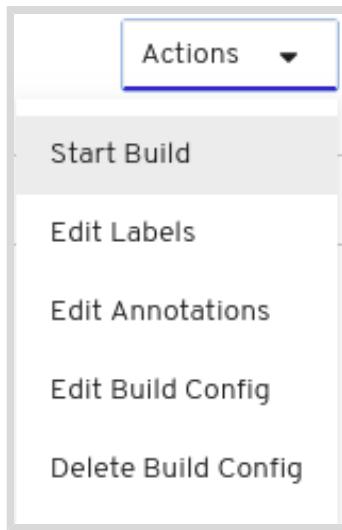
7. 启动应用新构建。

验证红帽 OpenShift 是否缩小运行旧版本的容器集，并使用最新版本的应用扩展两个新容器集。

使用 `curl` 命令重新测试应用。验证输出中是否显示 `version 2`。

- 7.1. 单击红帽 OpenShift Web 控制台左侧菜单中的 **Builds > Build Configs** 条目。

选择 `scale` 构建配置，然后单击 **Actions > Start Build**。



- 7.2. 控制台会将您重定向到有关新构建的摘要页。单击 **Logs** 选项卡，可观察构建日志。

- 7.3. 构建完成后，红帽 OpenShift 会扩展新版本应用的两个新容器集。

当新容器集准备好接收流量时，红帽 OpenShift 会缩减两个较旧的容器集。

单击 **Workloads > Pods** 可查看新的应用容器集。

- 7.4. 返回到终端窗口，并使用 `curl` 命令对路由 URL 发出多个 HTTP 请求以测试应用：

```
[student@workstation D0288-apps]$ curl \
http://scale-$[RHT_OCP4_DEV_USER]-manage-deploy.$[RHT_OCP4_WILDCARD_DOMAIN]
This is version 2 of the app. I am running on host -> scale-2-w7nfz (10.128.1.27)
[student@workstation D0288-apps]$ curl \
http://scale-$[RHT_OCP4_DEV_USER]-manage-deploy.$[RHT_OCP4_WILDCARD_DOMAIN]
This is version 2 of the app. I am running on host -> scale-2-dxswv (10.129.1.119)
```

输出中应显示 **version 2**。您还应该看到请求在两个容器集之间进行循环负载均衡。  
请注意，容器集名称和 IP 地址与旧部署不同。



### 注意

如果没有看到新版本，请验证：

- 您已将更改推送到远程 Git 存储库。
- 新构建已成功完成。

## 8. 回滚到上一个部署。

使用 **curl** 该命令重新测试应用。验证输出中是否显示 **version 1**。

### 8.1. 回滚到应用以前版本的部署。

您将收到一条警告消息，指出 **oc rollback** 命令禁用了镜像更改触发器。

```
[student@workstation D0288-apps]$ oc rollback dc/scale
deploymentconfig.apps.openshift.io/scale deployment #3 rolled back to scale-1
Warning: the following images triggers were disabled: scale:latest
You can re-enable them with: oc set triggers dc/scale --auto
```

8.2. 等待新的应用容器集部署。容器集必须处于 **READY** 状态：

```
[student@workstation D0288-apps]$ oc get pods
NAME        READY   STATUS    RESTARTS   AGE
scale-1-build  0/1    Completed  0          40m
scale-1-deploy 0/1    Completed  0          36m
scale-2-build  0/1    Completed  0          10m
scale-2-deploy 0/1    Completed  0          6m59s
scale-3-bcpxpg  1/1    Running   0          58s
scale-3-deploy  0/1    Completed  0          77s
scale-3-lnp46   1/1    Running   0          68s
```

8.3. 使用 **curl** 命令可验证应用响应：

```
[student@workstation D0288-apps]$ curl \
http://scale-$[RHT_OCP4_DEV_USER]-manage-deploy.$[RHT_OCP4_WILDCARD_DOMAIN]
This is version 1 of the app. I am running on host -> scale-3-bcpxpg (10.128.2.133)
[student@workstation D0288-apps]$ curl \
http://scale-$[RHT_OCP4_DEV_USER]-manage-deploy.$[RHT_OCP4_WILDCARD_DOMAIN]
This is version 1 of the app. I am running on host -> scale-3-lnp46 (10.131.1.206)
```

输出中应显示 **version 1**。您还应该看到请求在两个容器集之间进行循环负载均衡。  
请注意，容器集名称和 IP 地址与之前的部署不同。

9. 对您的作业进行评分：

```
[student@workstation DO288-apps]$ lab manage-deploy grade
```

10. 清理并删除项目。

```
[student@workstation DO288-apps]$ oc delete project \
${RHT_OCP4_DEV_USER}-manage-deploy
```

## 完成

在 workstation 上，执行 `lab manage-deploy finish` 脚本来完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab manage-deploy finish
```

本复习实验到此结束。

# 总结

---

在本章中，您学到了：

- 就绪度和存活度探测可监视应用的健康状况。
- 当您可以同时运行两个版本的应用以在不停机的情况下进行升级时，请使用 **Rolling** 策略。此策略首先使用新版本扩展其他容器集，然后在准备就绪后，使用旧版本缩小容器集。
- 当您无法同时运行两个版本的应用时，请使用 **Recreate** 策略。此策略使用先前版本关闭所有容器集，然后使用较新版本启动其他容器集。
- 当 OpenShift 提供的两个策略不适合您的需求时，请使用 **Custom** 策略自定义部署过程。
- **Recreate** 和 **Rolling** 策略都支持生命周期 hook，允许您在部署过程中的各个点自定义部署。
- 您可以将资源限制指定为部署策略的一部分，以限制应用部署的资源使用。



## 章 8

# 面向 OpenShift 构建应用

### 目标

在 OpenShift 中创建和部署应用。

### 培训目标

- 将容器化应用与非容器化服务进行整合。
- 按照 OpenShift 推荐做法部署容器化的第三方应用。
- 使用红帽 OpenShift 应用运行时部署应用。

### 章节

- 整合外部服务（及引导式练习）
- 使服务容器化（及引导式练习）
- 使用 Eclipse JKube 部署云原生应用（及引导式练习）

### 实验

面向 OpenShift 构建云原生应用

# 集成外部服务

## 培训目标

学完本节后，您应能够将容器化应用与非容器化服务进行集成。

## 回顾红帽 OpenShift 服务

OpenShift 中的典型服务同时具有一个名称和一个选择器。服务使用其选择器来识别应接收发送到服务的应用请求的容器集。OpenShift 应用使用服务名称来连接到服务端点。

同样，FQDN 允许应用使用名称来访问公共服务的端点。不过，OpenShift 服务允许应用在无需公开服务的前提下访问服务端点。

应用使用环境变量或 OpenShift 内部 DNS 服务器发现服务。使用环境变量要求在创建应用容器集之前定义服务，否则应用将不会接收环境变量。

使用 OpenShift 内部 DNS 服务器更加灵活，因为它允许应用动态发现服务。服务名称成为包含该服务的同一 OpenShift 集群内所有容器集的本地 DNS 主机名。OpenShift 将 `svc.cluster.local` 域名后缀添加到所有容器的 DNS 解析器搜索路径中。OpenShift 还会为每个服务分配 `service-name.project-name.svc.cluster.local` 主机名。

例如，如果 `myproject` 项目中存在 `myapi` 服务，则同一 OpenShift 集群内的所有容器集都可以解析 `myapi.myproject.svc.cluster.local` 主机名以获取服务 IP 地址。还提供以下短主机名：

- 来自同一项目的容器集可以使用 `myapi` 服务名称作为短主机名，不带任何域后缀。
- 来自不同项目的容器集可以使用服务名称和 `myapi.myproject` 项目名称作为短主机名，而不使用 `svc.cluster.local` 域后缀。

## 定义外部服务

OpenShift 支持多种方法来定义不包含选择器的服务。这允许 OpenShift 服务指向 OpenShift 集群外部的一个或多个主机。

假设您要容器化的某一应用依赖于并非 OpenShift 集群内即刻可用的现有数据库服务。在容器化应用之前，您不需要将数据库服务迁移到 OpenShift。取而代之，您可开始设计应用以与 OpenShift 服务（包括数据库服务）进行交互。只需创建引用外部数据库服务端点的 OpenShift 服务。

为外部服务的端点创建 OpenShift 服务时，应用能够发现内部和外部服务。此外，如果外部服务的端点有变化，您无需重新配置受影响的应用，而是只需更新相应 OpenShift 服务的端点便。

## 创建外部服务

若要创建外部服务，最简单的方法是使用带有 `--external-name` 选项的 `oc create service externalname` 命令：

```
[user@host ~]$ oc create service externalname myservice \
--external-name myhost.example.com
```

前面的示例也可以接受 IP 地址而不是 DNS 名称。

然后，在 OpenShift 集群内运行的应用使用服务名称的方式与使用常规服务的方式相同，可以是环境变量，也可以是本地主机名。

## 定义服务的端点

典型服务根据服务的选择器属性动态创建 endpoint 资源。`oc status` 和 `oc get all` 命令不显示这些资源。您可以使用 `oc get endpoints` 命令来显示它们。

如果您使用 `oc create service externalname --external-name` 命令来创建服务，该命令还会创建指向作为参数给出的主机名或 IP 地址的端点资源。

如果您不使用 `--external-name` 选项，则不会创建端点资源。在这种情况下，可以使用 `oc create -f` 命令和资源定义文件来显式创建端点资源。

如果从文件创建端点，则可以为同一外部服务定义多个 IP 地址，并依赖 OpenShift 服务负载平衡功能。在这种情况下，OpenShift 不会添加或移除地址以说明每个实例的可用性。外部应用必须更新端点资源中的 IP 地址列表。

有关端点资源定义文件的详细信息，请参阅本节末尾的参考资料。



### 参考文献

查看 Kubernetes 的 Service 概念文档中的 Type ExternalName 部分，网址为：  
<https://kubernetes.io/docs/concepts/services-networking/service/#externalname>

请参阅红帽 OpenShift 容器平台 4.10 Architecture Guide 中 Networking 章节的 OpenShift DNS 命名约定部分，网址为：

[https://access.redhat.com/documentation/en-us/openshift\\_container\\_platform/4.10/html-single/networking/index](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.10/html-single/networking/index)

Authors note: The following resource does not exist yet in the 4.x version of the documentation for OpenShift. However, the information linked below is still relevant to the current 4.10 release of the product.

请参阅红帽 OpenShift 容器平台 3.11 Developer Guide 中的 Integrating External Services 章节，网址为：

[https://docs.openshift.com/container-platform/3.11/dev\\_guide/integrating\\_external\\_services.html](https://docs.openshift.com/container-platform/3.11/dev_guide/integrating_external_services.html)

## ► 指导练习

# 集成外部服务

在本练习中，您将在 OpenShift 上部署一个应用，该应用与在 OpenShift 集群外部运行的数据 库进行通信。

## 成果

您应能够：

- 从源代码部署 **To Do List** 应用。
- 为应用创建指向 OpenShift 集群外部的 MariaDB 数据库服务器的数据服务。
- 验证应用是否使用预先加载到数据库中的数据。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Git 存储库 (**todo-single**) 中的 **To Do List** 应用。
- 提供应用所需的 npm 依赖项 (**restify**、**sequelize** 和 **mysql**) 的 Nexus 服务器。
- Node.js 12 S2I 构建器镜像。
- 在 OpenShift 集群外部运行的预填充 MariaDB 数据库服务器。

在 **workstation** 上，运行以下命令来验证前提条件并部署 To Do List 应用：

```
[student@workstation ~]$ lab external-service start
```

上一个命令运行 `~/D0288/labs/external-service` 文件夹中的 `oc-new-app.sh` 脚本以部署应用。如果需要有关本练习中使用的 **To Do List** 应用资源的更多信息，您可以查看此脚本。

## 说明

### ► 1. 检查 **To Do List** 应用资源。

1.1. 加载您的课堂环境配置。

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户帐户登录 OpenShift。

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

## 章 8 | 面向 OpenShift 构建应用

1.3. 进入托管 To Do List 应用的 youruser-external-service 项目：

```
[student@workstation ~]$ oc project ${RHT_OCP4_DEV_USER}-external-service
Already on project "developer-external-service" on server "https://
api.ocp4.example.com:6443".
```

1.4. 验证 OpenShift 项目是否具有名为 todoapp 的单个应用，该应用是从以下源构建：

```
[student@workstation ~]$ oc status
...output omitted...
http://todoapp-developer-external-service.apps.ocp4.example.com to pod port 8080-
tcp (svc/todoapp)
dc/todoapp deploys istag/todoapp:latest <-
bc/todoapp source builds https://github.com/yourgithubuser/D0288-apps on
openshift/nodejs:16-ubi8
    deployment #1 deployed 26 seconds ago - 1 pod
...output omitted...
```

1.5. 验证应用容器准备就绪并在运行：

```
[student@workstation ~]$ oc get pod
NAME          READY   STATUS    RESTARTS   AGE
todoapp-1-6z6qg  1/1     Running   0          1m
todoapp-1-build  0/1     Completed  0          4m
todoapp-1-deploy 0/1     Completed  0          1m
```

1.6. 检查应用容器集中的环境变量以获取数据库连接参数：

```
[student@workstation ~]$ oc rsh todoapp-1-6z6qg env | grep DATABASE
DATABASE_PASSWORD=redhat123
DATABASE_SVC=tododb
DATABASE_USER=todoapp
DATABASE_INIT=false
DATABASE_NAME=todo
```

▶ 2. 验证 To Do List 应用是否无法访问 DATABASE\_SVC 环境变量指定的数据库服务器，即 tododb。

2.1. 获取公开应用的主机名。由于主机名很长，因此将其保存到 shell 变量中。

```
[student@workstation ~]$ HOSTNAME=$(oc get route todoapp \
-o jsonpath='{.spec.host}')
[student@workstation ~]$ echo ${HOSTNAME}
todoapp-developer-external-service.apps.ocp4.example.com
```

2.2. 使用 curl 命令、上一步中的主机名，以及 API 资源路径 /todo/api/items/6 从数据库中获取项目。应用发出的错误消息指示其无法解析 tododb 服务主机名。

```
[student@workstation ~]$ curl -si http://${HOSTNAME}/todo/api/items/6
HTTP/1.1 500 Internal Server Error
...
{"message":"getaddrinfo ENOTFOUND tododb"}
```

► 3. 检查外部数据库。

- 3.1. 使用 `mysqlshow` 命令连接到外部 MariaDB 服务器中的 `todo` 数据库，再验证它是否包含 `Item` 表。

`RHT_OCP4_MYSQL_SERVER` 变量包含数据库服务器的主机名。以 `todoapp` 为用户，以 `redhat123` 为密码：

```
[student@workstation ~]$ mysqlshow -h${RHT_OCP4_MYSQL_SERVER} \
-utodoapp -predhat123 todo
Database: todo
+-----+
| Tables |
+-----+
| Item   |
+-----+
```

► 4. 创建连接到外部数据库实例的 OpenShift 服务，并验证应用现在是否从外部数据库获取数据。

- 4.1. 使用 `oc create svc` 命令，基于外部名称和上一步中的数据库服务器主机名来创建服务。

```
[student@workstation ~]$ oc create svc externalname tododb \
--external-name ${RHT_OCP4_MYSQL_SERVER}
service/tododb created
```

- 4.2. 验证 `tododb` 服务是否存在，并且显示外部 IP 而非集群 IP：

```
[student@workstation ~]$ oc get svc
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP      ...
todoapp   ClusterIP   172.30.140.201  <none>          ...
tododb    ExternalName <none>          mysql.ocp-eu46a.prod.ole.redhat.com ...
```

- 4.3. 验证应用现在是否能够从外部数据库获取数据。

再次使用 `curl` 命令：

```
[student@workstation ~]$ curl -si http://${HOSTNAME}/todo/api/items/6
HTTP/1.1 200 OK
...
{"id":6,"description":"Verify that the To Do List application works","done":false}
```

► 5. 清理。从 OpenShift 删除 `youruser-external-service` 项目。

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-external-service
```

## 完成

在 `workstation` 上运行 `lab external-service finish` 命令以完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab external-service finish
```

本引导式练习到此结束。

# 容器化服务

---

## 培训目标

学完本节后，学员应能够按照 OpenShift 的建议实践方法来检查和部署容器化的第三方应用。

## 检查要部署到 OpenShift 的容器化应用

有时，您必须在 OpenShift 中部署应用，其供应商要么提供 Dockerfile 来构建其容器镜像，要么在注册表服务器中提供预构建的容器镜像。此应用可能提供自定义应用所需的后端服务，例如数据库或消息传递系统，或开发过程所需的服务（如托管容器注册表、构件存储库或协作工具）。

虽然应用已被其供应商容器化，这并不一定表示它在 OpenShift 上运行良好。鉴于几个可能的原因，首次尝试使用 `oc new-app` 命令部署该应用可能会失败。解决这些问题可能需要自定义应用的部署或更改其 Dockerfile。

以下是常见问题的非详尽列表：

- 应用无法在由 **restricted** 安全上下文约束 (SCC) 定义的默认 OpenShift 安全策略下运行。应用的容器镜像可能期望以 **root** 或其他固定用户身份运行。它可能还需要自定义 SELinux 策略和其他特权设置。

通常，您可以更改 Dockerfile，以便应用遵循默认的 OpenShift 安全策略。否则，请仔细评估在更宽松安全策略下运行应用的风险。

- 应用需要针对资源请求和资源限制进行自定义设置。这发生在传统的单体式应用中，这些应用不是面向基于微服务的架构而设计的。OpenShift 集群管理员设置默认限制范围，为资源请求和项目限制提供默认值，这些值对于传统应用来说可能太小。

您可以对部署指定更多的资源请求和资源限制，以覆盖集群管理员设置的默认值。如果应用要求超过 OpenShift 集群管理员设置的资源配置，则管理员必须增大用户的资源配置，或者为应用提供具有更大资源配置的服务帐户。

- 应用镜像定义适用于独立容器运行时，但不适用于 OpenShift 或通用 Kubernetes 集群的配置设置。应用通常从环境变量读取配置设置，并且通过使用 Dockerfile 中的 **ENV** 指令来定义默认值。

无需更改 Dockerfile 来改变其环境变量。您可以覆盖应用部署中的任何环境变量，包括未由 Dockerfile 中的 **ENV** 指令显式设置的环境变量。

- 应用需要持久存储。应用通常使用 Dockerfile 中的 **VOLUME** 指令定义存储要求。有时，这些应用希望对其卷使用主机文件夹，而 OpenShift 集群管理员会禁止这样做。这些应用可能期望直接访问网络存储，从安全角度来说，这也并不是建议的做法。

若要为 OpenShift 中的应用提供持久存储，正确的方法是定义持久性卷声明 (PVC) 资源，并通过配置应用部署将这些 PVC 附加到应用容器中的卷。OpenShift 代表应用管理网络存储，集群管理员则管理存储的安全策略和性能级别。

您的供应商可能会提供 Kubernetes 资源文件来部署应用，这些资源文件可能需要自定义才能在 OpenShift 下工作。与上游 Kubernetes（以及 Kubernetes 的其他供应商分发）一起工作的部署资源可能无法使用 OpenShift，因为标准 Kubernetes 不附带安全的默认设置。

有时，供应商提供的 Kubernetes 部署资源会假定用户是集群管理员。这些供应商可能没有考虑到 OpenShift 集群通常由多个组织和团队共享，而为了自身安全，这些组织和团队不会被授予集群管理员特权。

## 检查 Nexus 应用的 Dockerfile

由 Sonatype 开发的 Nexus 应用是一个存储库管理器，通常由开发人员用来存储应用所需的构件，例如依赖项。此应用可以存储多种技术的构件，包括 Java、.NET、Docker、Node、Python 和 Ruby。

Sonatype 提供了一个 Dockerfile，用于使用红帽 Linux 基础容器镜像构建 Nexus 应用。生成的镜像可从红帽容器目录中获得，该目录证明它符合红帽的一套核心准则。Dockerfile 已配置为支持 OpenShift 功能，例如允许容器镜像以随机用户帐户身份运行。

### 基础镜像和容器元数据

Dockerfile 项目可从 <https://github.com/sonatype/docker-nexus3/tree/3.18.0> 找到。Dockerfile.rhel.el 文件中提供了以下内容：

```
...output omitted...
FROM registry.access.redhat.com/rhel7/rhel ①

MAINTAINER Sonatype <cloud-ops@sonatype.com>
...output omitted...

LABEL name="Nexus Repository Manager" \
      ...output omitted...
      io.k8s.description="The Nexus Repository Manager server" \
      with universal support for popular component formats." \
      io.k8s.display-name="Nexus Repository Manager" \
      io.openshift.expose-services="8081:8081" \
      io.openshift.tags="Sonatype,Nexus,Repository Manager"

...output omitted...
```

- ① 指定红帽 Linux 7 容器镜像作为基础镜像。还有一个备选的 Dockerfile 文件，该文件使用 GitHub 项目中的红帽通用基础镜像 8。
- ② 定义 Kubernetes 和 OpenShift 的镜像元数据。

Nexus Docker 文件还使用 Dockerfile ARG 指令。ARG 指令定义仅在镜像构建过程中存在的变量。与 ENV 指令不同，这些变量不是生成的容器镜像的一部分：

```
...output omitted...
ARG NEXUS_VERSION=3.18.0-01
ARG NEXUS_DOWNLOAD_URL=https://.../nexus/3/nexus-${NEXUS_VERSION}-unix.tar.gz
ARG NEXUS_DOWNLOAD_SHA256_HASH=e1d9...c99e
...output omitted...
```

构建过程使用这三个变量来控制和验证容器镜像中安装的 Nexus 版本。

Dockerfile 还定义构建的容器镜像中存在的环境变量：

```
...output omitted...
# configure nexus runtime
ENV SONATYPE_DIR=/opt/sonatype
ENV NEXUS_HOME=${SONATYPE_DIR}/nexus \
    NEXUS_DATA=/nexus-data \
    NEXUS_CONTEXT='' \
    SONATYPE_WORK=${SONATYPE_DIR}/sonatype-work \
DOCKER_TYPE='rh-docker'
...output omitted...
```

安装过程使用 `DOCKER_TYPE` 环境变量来自定义安装和配置。`rh-docker` 的值调用红帽认证容器镜像所需的任何配置更改。

## Nexus 安装过程

Sonatype 维护用于标准化 Nexus 应用安装的 Chef 说明书。Chef 是类似于 Puppet 和 Ansible 的一种开源配置管理技术，旨在简化服务器配置和管理过程。Chef 说明书是 Chef 配方的集合，每个配方定义一组特定系统资源的配置。

Nexus Dockerfile 使用 Chef 安装过程来构建 Nexus 容器镜像：

```
...output omitted...
ARG NEXUS_REPOSITORY_MANAGER_COOKBOOK_VERSION="release-0.5.20190212-..."①
ARG NEXUS_REPOSITORY_MANAGER_COOKBOOK_URL="https://github.com/sonatype/..."

ADD solo.json.erb /var/chef/solo.json.erb②

# Install using chef-solo
RUN curl -L ...output omitted.../install.sh | bash \ ③
    && /opt/chef/embedded/bin/erb /var/chef/solo.json.erb > /var/chef/solo.json \
    && chef-solo \
        --node_name nexus_repository_red_hat_docker_build \
        --recipe-url ${NEXUS_REPOSITORY_MANAGER_COOKBOOK_URL} \
        --json-attributes /var/chef/solo.json \
    && rpm -qa chef | xargs rpm -e \
    && rpm --rebuilddb \
    && rm -rf /etc/chef \
    && rm -rf /opt/chefdk \
    && rm -rf /var/cache/yum \
    && rm -rf /var/chef
...output omitted...
```

- ① Dockerfile 使用变量来控制用于安装 Nexus 的 Chef 说明书的版本。在后续的 `RUN` 指令中，说明书 URL 作为 Chef 命令的参数提供。
- ② 在与 Dockerfile 相同的目录中，`solo.json.erb` 文件包含 Chef 配置详细信息。Dockerfile 会将此文件添加到容器镜像以启用 Chef 安装过程。

- ③ Nexus 的安装通过单个 **RUN** 指令完成，该指令的功能如下：

- 下载并安装 Chef。
- 通过 **chef-solo** 命令执行 Chef 说明书以安装 Nexus。
- 删除下载的文件、Chef RPM 和其他无关文件。

Chef 说明书还配置文件权限，以便数据和日志文件夹可以通过 `gui_0` 写入，从而符合 OpenShift 默认安全策略。

## 容器执行环境

Dockerfile 的末尾提供元数据，使容器运行时能够从镜像创建容器：

```
...output omitted...
VOLUME ${NEXUS_DATA}①

EXPOSE 8081②
USER nexus③

ENV INSTALL4J_ADD_VM_PARAMS="-Xms1200m -Xmx1200m ..."④

ENTRYPOINT ["/uid_entrypoint.sh"]⑤
CMD ["sh", "-c", "${SONATYPE_DIR}/start-nexus-repository-manager.sh"]⑥
```

- ① 将 `/nexus-data` 目录配置为卷，因为 `NEXUS_DATA` 变量的值为 `/nexus-data`。Nexus 应用在此目录中存储应用数据和资源。
- ② Nexus 应用通过端口 8081 进行通信。
- ③ Nexus 应用以 `nexus` 用户身份运行。Chef 安装过程会创建并配置此用户。
- ④ Nexus 应用在 Java 虚拟机 (JVM) 上运行。在独立容器运行时中运行时，JVM 需要选项来调整其堆的大小。在 OpenShift 中，您必须覆盖这些选项，并让 JVM 服从 OpenShift 资源限制和资源请求。否则，可能会导致调度和稳定性问题。
- ⑤ 在安装过程中，Chef 说明书会创建 `/uid_entrypoint.sh` 脚本。`/uid_entrypoint.sh` 脚本的目标是识别正在运行应用的用户 ID 并将其定义到 `nexus` 用户。这允许应用使用默认 `restricted` 安全上下文约束正常执行。
- ⑥ 启动 Nexus 应用的命令。入口点先调整权限以考虑随机分配的进程 UID，然后执行此命令。

## 为 Nexus 镜像自定义 OpenShift 资源

如果从官方的 Dockerfile 构建 Nexus 容器镜像，并使用 `oc new-app --docker-image` 等命令进行部署，则可能无法可靠地工作。为确保容器镜像正常工作，您必须对其部署进行一些自定义。

### 设置资源请求和限制

红帽建议部署到 OpenShift 的应用定义资源请求和资源限制。如果不使用非常旧的 Java 虚拟机 (JVM) 版本，则无需定义 JVM 堆选项。

资源请求会指出应用需要运行的最小资源量。OpenShift 调度程序在集群中查找具有足以运行应用的可用 CPU 和内存的工作程序节点，以免因为内存不足错误而无法启动。

## 章 8 | 面向 OpenShift 构建应用

资源限制声明应用的 CPU 和内存使用量随时间可能会增加多少。OpenShift 为应用容器设置 Linux 内核 cgroups，防止它违反这些限制。Linux 内核会终止违反这些限制的容器，然后 OpenShift 启动替代容器，从而缓解内存泄漏、无限循环和死锁引起的问题。JVM 的最新版本会自动调整堆和其他内部数据结构的大小，从而不违反当前的 cgroups 设置。

在部署的容器集模板中定义资源请求和资源限制：

```
...output omitted...
- apiVersion: apps/v1
  kind: Deployment
...output omitted...
  spec:
    ...output omitted...
      template:
        ...output omitted...
          spec:
            containers:
              ...output omitted...
                resources:
                  limits:
                    cpu: "1"
                    memory: 2Gi
                  requests:
                    cpu: 500m
                    memory: 256Mi
...output omitted...
```

在 Nexus 情景中，还必须覆盖 `INSTALL4J_ADD_VM_PARAMS` 环境变量，以删除与内存大小调整相关的任何 JVM 选项：

```
...output omitted...
- apiVersion: apps/v1
  kind: Deployment
...output omitted...
  spec:
    ...output omitted...
      template:
        ...output omitted...
          spec:
            containers:
              - env:
                  - name: INSTALL4J_ADD_VM_PARAMS
                    value: -Djava.util.prefs.userRoot=/nexus-data/javaprefs
...output omitted...
```

前面的示例假定您没有通过覆盖 `NEXUS_DATA` 环境变量来为容器的持久存储指定不同的卷路径。

## 实施探测

红帽建议部署到 OpenShift 的应用定义健康探测。Nexus 提供了一个 API 来确定服务是否处于活动状态以及是否存在任何死锁。但是，您不能使用简单的 HTTP 探测，因为 Nexus API 需要身份验证，即使 Nexus 不正常，API 也会返回 HTTP 200 状态代码。

## 章 8 | 面向 OpenShift 构建应用

您可以创建脚本，以使用 Nexus API 进行身份验证并分析来自 API 的响应。以下脚本验证 Nexus 服务是否已准备好为请求提供服务：

```
#!/bin/sh
curl -si -u admin:$(cat /nexus-data/admin.password) \
      http://localhost:8081/service/metrics/ping | grep pong
```

- ① -u 选项在 HTTP GET 请求中传递用户名和密码。
- ② Nexus 提供的 `/service/metrics/ping` 端点必须在服务就绪时返回 `pong` 值。如果 `pong` 不在响应中，则 `grep` 命令将退出并返回非零状态代码。

`admin` 用户有权向健康检查端点发出请求。当 Nexus 应用首次初始化时，它将为 `admin` 用户生成随机密码。Nexus 将随机密码存储到 `/nexus-data/admin.password` 文件中。

类似地，以下脚本确定 Nexus 服务是否处于活动状态：

```
#!/bin/sh
curl -si -u admin:$(cat /nexus-data/admin.password) \
      http://localhost:8081/service/metrics/healthcheck | grep healthy | \
      grep true
```

由于这些探测脚本很简单，因此可以将每个脚本的内容放在 Nexus 容器规范的相应探测节中：

```
...output omitted...
- apiVersion: apps/v1
  kind: Deployment
...output omitted...
  spec:
    ...output omitted...
    template:
    ...output omitted...
    spec:
      containers:
        ...output omitted...
        livenessProbe:
          exec:
            command:
              - /bin/sh ①
              - "-c" ②
              - > ③
                curl -si -u admin:$(cat /nexus-data/admin.password)
                  http://localhost:8081/service/metrics/healthcheck |
                    grep healthy | grep true
          failureThreshold: 3
          initialDelaySeconds: 10
          periodSeconds: 10
        ...output omitted...
```

- ① 使用 `/bin/sh` shell。
- ② `-c` 选项指示在 shell 中执行单个命令。

- ③ YAML 多行延续指示符。这允许您将一个较长的字符串拆分为几行。

对于大型探测脚本，请考虑修改 Dockerfile 并将探测脚本添加到容器镜像。



## 参考文献

有关 Nexus 服务的更多信息，请参阅 Nexus 存储库管理器文档，网址为：  
<https://help.sonatype.com/repomanager3>

有关 Nexus 应用健康和指标的详细信息，请参阅  
**Nexus 3 服务器指标和健康信息 – Sonatype 支持**

<https://support.sonatype.com/hc/en-us/articles/226254487-Nexus-3-Server-Metrics-and-Health-Information>

有关容器上 JVM 内存设置的详细信息，请参阅  
**Docker 中的 Java：防失败须知**

<https://developers.redhat.com/blog/2017/03/14/java-inside-docker/>

## ▶ 指导练习

# 将 Nexus 作为服务容器化

在本练习中，您将按照 OpenShift 建议做法将 Nexus 服务器部署为容器化应用。

## 成果

您应能够：

- 从 Dockerfile 构建容器镜像。
- 验证 Helm 图表是否使用镜像流，该镜像流是否使用来自私有注册表的容器镜像。
- 验证 Helm 图表是否定义了 Java 应用的资源请求和资源限制，并覆盖将设置堆大小的环境变量。
- 配置 Helm 图表以使用持久卷声明。
- 配置 Helm 图表以使用存活度和就绪度探测。
- 使用 Helm 图表部署 Nexus 实例。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 红帽通用基础镜像 8 (**ubi8/ubi**)。
- Git 存储库中用来构建 Nexus 容器镜像的源文件 (**nexus3**)。

在 **workstation** 上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab nexus-service start
```

## 说明

### ▶ 1. 查看并构建 Nexus **Dockerfile** 文件。

- 1.1. 进入 **DO288-apps** Git 存储库本地克隆的 **nexus3** 子目录，并签出课程存储库的 **main** 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd ~/DO288-apps/nexus3
[student@workstation nexus3]$ git checkout main
...output omitted...
```

- 1.2. 检查 **Dockerfile** 文件，并观察 Nexus 应用将其文件保留在 **/nexus-data** 文件夹中：

```
[student@workstation nexus3]$ grep VOLUME Dockerfile
VOLUME ${NEXUS_DATA}
[student@workstation nexus3]$ grep NEXUS_DATA Dockerfile
  NEXUS_DATA=/nexus-data \
...output omitted...
```

Dockerfile 使用 `/nexus-data` 值定义 `NEXUS_DATA` 环境变量。Dockerfile 使用此变量来定义 Nexus 存储应用数据的卷。

- 1.3. 检查 `Dockerfile` 文件，并观察它定义了设置 Java 虚拟机 (JVM) 堆大小的环境变量。稍后，您必须覆盖此环境变量，以便 OpenShift 部署配置控制容器集内存设置。

```
[student@workstation nexus3]$ grep ENV Dockerfile
...output omitted...
ENV INSTALL4J_ADD_VM_PARAMS="-Xms2703m -Xmx2703m -XX:MaxDirectMemorySize=2703m" -
Djava.util.prefs.userRoot=${NEXUS_DATA}/javaprefs"
```

- 1.4. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation nexus3]$ source /usr/local/etc/ocp4.config
```

- 1.5. 构建容器镜像，并将它推送到您在 Quay.io 上的个人帐户。您可以从 `~/D0288/labs/nexus-service` 处的 `build-nexus-image.sh` 脚本运行或复制并粘贴命令。

在构建过程中，请忽略显示“HOSTNAME is not supported for OCI image format”的警告。

```
[student@workstation nexus3]$ podman build -t nexus3 .
...output omitted...
STEP 33: COMMIT nexus3
[student@workstation nexus3]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
[student@workstation nexus3]$ skopeo copy \
--format v2s1 containers-storage:localhost/nexus3 \
docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
...output omitted...
Writing manifest to image destination
Storing signatures
```



## 重要

完成构建过程应该不会超过 5 分钟。在测试此练习时，报告的构建时间要长得多。

如果构建在五分钟后没有完成，请按 **Ctrl+C** 退出构建程序。使用以下命令，将预构建的 Nexus 镜像直接复制到您的个人 Quay.io 帐户，并跳过构建过程：

```
[student@workstation nexus3]$ skopeo copy --format v2s1 \
docker://quay.io/redhattraining/nexus3:latest \
docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

▶ 2. 完成 Helm 图表以将 Nexus 部署为服务。

提供了一个可部署 Nexus 的部分图表。请更新它以将 Nexus 部署为服务。

2.1. 创建初学者图表的副本以进行编辑，再退出 ~/D0288-apps/nexus3 文件夹。

```
[student@workstation nexus3]$ cp -r ~/D0288/labs/nexus-service/nexus-chart ~/  
[student@workstation nexus3]$ cd ~
```

2.2. 检查 ~/nexus-chart/templates/imagestream.yaml 文件，以验证其是否使用了从配置值设置的容器镜像。

```
[student@workstation ~]$ grep -A1 "kind: DockerImage" ~/nexus-chart/templates/  
imagestream.yaml  
    kind: DockerImage  
    name: {{ .Values.imageName }}
```

2.3. 使用文本编辑器打开 ~/nexus-chart/values.yaml 文件，并进行更新，以将 imageName 值指向您构建的容器镜像。在编辑器中将此文件保持打开。

```
...output omitted...  
imageName: quay.io/youruser/nexus3:latest  
...output omitted...
```

2.4. 检查 ~/nexus-chart/templates/deployment.yaml 文件，以验证它是否为应用容器集定义了资源请求和资源限制：

```
[student@workstation ~]$ grep -B1 -A5 limits: ~/nexus-chart/templates/  
deployment.yaml  
  resources:  
    limits:  
      cpu: "1"  
      memory: {{ .Values.memoryLimit }}  
    requests:  
      cpu: 500m  
      memory: 256Mi
```

2.5. 使用文本编辑器打开 ~/nexus-chart/templates/deployment.yaml 文件，然后覆盖 INSTALL4J\_ADD\_VM\_PARAMS 环境变量，使其不包含任何 JVM 堆大小调整配置，并且仅定义应用所需的 Java 系统属性。在编辑器中将该文件保持打开。

```
apiVersion: v1  
kind: Deployment  
...output omitted...  
  - env:  
    - name: INSTALL4J_ADD_VM_PARAMS  
      value: -Djava.util.prefs.userRoot=/nexus-data/javaprefs  
...output omitted...
```

2.6. 使用文本编辑器更新 ~/nexus-chart/values.yaml 文件，将 memoryLimit 值设置为 2703Mi

- 2.7. Nexus 应用包含一个 `/service/metrics/healthcheck` 端点，用于验证应用是否处于活动状态。访问该端点需要进行身份验证。应用启动后，其 `admin` 用户密码存储在 `/nexus-data/admin.password` 文件中。

检查部署资源中的存活度探测。使用文本编辑器打开 `~/nexus-chart/templates/deployment.yaml` 文件，并将探测配置为在容器启动后 120 秒启动。另外，将探测的最长等待执行时间配置为 30 秒。

```
apiVersion: v1
kind: Deployment
...output omitted...
  livenessProbe:
    exec:
      command:
        - /bin/sh
        - "-c"
        - '>
          curl -siu admin:$(cat /nexus-data/admin.password)
          http://localhost:8081/service/metrics/healthcheck |
          grep healthy | grep true
        ...output omitted...
      initialDelaySeconds: 120
      ...output omitted...
      timeoutSeconds: 30
...output omitted...
```

- 2.8. Nexus 应用包含一个 `/service/metrics/ping` 端点，用于验证应用是否已就绪。

访问此端点需要进行身份验证。

检查部署资源中的就绪度探测。在 `~/nexus-chart/templates/deployment.yaml` 文件中，将探测配置为在容器启动后 120 秒启动。另外，将探测的最长等待执行时间配置为 30 秒。

```
apiVersion: v1
kind: Deployment
...output omitted...
  readinessProbe:
    exec:
      command:
        - /bin/sh
        - "-c"
        - '>
          curl -siu admin:$(cat /nexus-data/admin.password)
          http://localhost:8081/service/metrics/ping |
          grep pong
        ...output omitted...
      initialDelaySeconds: 120
      ...output omitted...
      timeoutSeconds: 30
...output omitted...
```

- 2.9. 在部署资源中，将卷挂载配置为使用 `/nexus-data` 挂载路径处的 `nexus-data` 卷：

```
apiVersion: v1
kind: Deployment
...output omitted...
  volumeMounts:
    - mountPath: /nexus-data
      name: nexus-data
...output omitted...
```

- 2.10. 在部署资源中，将卷命名为 **nexus-data**，然后将卷配置为使用 **nexus-data-pvc** 持久卷声明：

```
apiVersion: v1
kind: Deployment
...output omitted...
  volumes:
    - name: nexus-data
      persistentVolumeClaim:
        claimName: nexus-data-pvc
...output omitted...
```

- 2.11. 永久保存 Nexus 数据需要持久卷声明。在文本编辑器中打开 `~/nexus-chart/templates/pvc.yaml` 文件。在持久卷声明资源中，通过更新 `metadata.name` 属性将持久卷声明命名为 **nexus-data-pvc**：

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  labels:
    app: {{ .Values.nexusServiceName }}
  name: nexus-data-pvc
...output omitted...
```

- 2.12. Helm 图表已完成。保存您的修改。

要验证此步骤中所做的更改，请将您的 Helm 图表文件与 `~/DO288/solutions/nexus-service/nexus-chart/` 中的答案文件进行比较。如果不确定您的编辑，您可以复制答案文件并继续下一步。

### ► 3. 创建新项目。添加一个机密，以允许任何项目用户从 Quay.io 中拉取 Nexus 容器镜像。

- 3.1. 使用您的开发人员用户帐户登录 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 3.2. 为应用创建一个新项目

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-nexus-service
Now using project "yourname-nexus-service" on server
"https://api.ocp4.example.com:6443".
...output omitted...
```

3.3. 使用 Podman (不带 sudo 命令) 登录 Quay.io。

```
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

3.4. 创建一个机密，以访问您的 Quay.io 个人帐户。

```
[student@workstation ~]$ oc create secret generic quayio \
--from-file dockerconfigjson=${XDG_RUNTIME_DIR}/containers/auth.json \
--type kubernetes.io/dockerconfigjson
secret/quayio created
```

3.5. 将机密链接到 default 服务帐户。

```
[student@workstation ~]$ oc secrets link default quayio --for pull
```

3.6. 在 ~/nexus-chart/values.yaml 中设置正确的主机名。复制以下命令的输出，并粘贴到值文件以设置 hostname 值。

```
[student@workstation ~]$ echo "nexus3-${RHT_OCP4_DEV_USER}.${RHT_OCP4_WILDCARD_DOMAIN}"
```

#### ► 4. 创建新应用。

4.1. 从 Helm 图表中创建一个名为 nexus3 的新应用。

您可以复制完整命令，或直接从 /home/student/D0288/labs/nexus-service/oc-new-app.sh 执行。

```
[student@workstation ~]$ helm install nexus3 ~/nexus-chart
NAME: nexus3
LAST DEPLOYED: Mon May 31 12:05:20 2021
NAMESPACE: youruser-nexus-service
STATUS: deployed
REVISION: 1
NOTES:
..output omitted...
```

4.2. 等待应用容器集运行，但未准备就绪。跟踪容器集日志，以观察冗长的 Nexus 服务器初始化过程。看到“Started”消息时，您可以停止跟踪日志。

```
[student@workstation ~]$ oc get pods
NAME          READY   STATUS    RESTARTS   AGE
nexus3-1-kfwwh  0/1     Running   0          1m25s
```

## 章 8 | 面向 OpenShift 构建应用

```
[student@workstation ~]$ oc logs -f nexus3-1-kfwwh
...output omitted...
-----
Started Sonatype Nexus OSS 3.30.1-01
-----
...output omitted...
```

按 **Ctrl+C** 退出 **oc logs** 命令。

- 4.3. 等待应用准备就绪并在运行。OpenShift 可能需要过几秒后才能从应用健康探测获得健康状态。

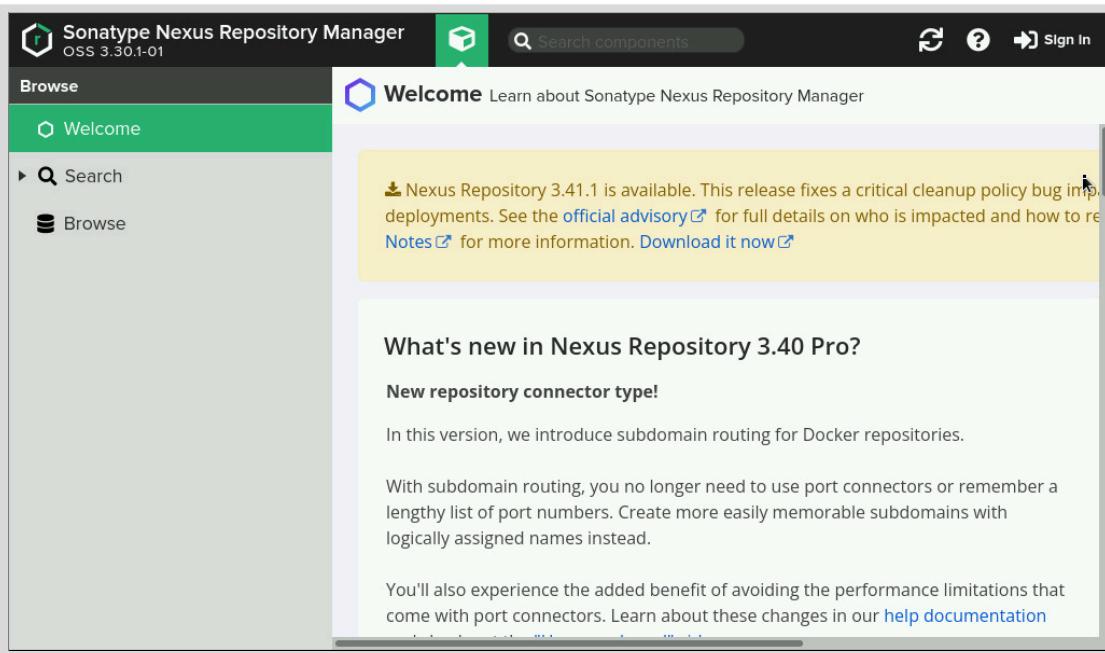
```
[student@workstation ~]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
nexus3-a1b2c3d4e5f6-kfwwh   1/1     Running   0          4m25s
```

## ▶ 5. 测试 Nexus 应用。

- 5.1. 检索 Nexus 应用的路由：

```
[student@workstation ~]$ oc get route
NAME      HOST/PORT
nexus3   nexus3-yourname.apps.ocp4.example.com ...
```

- 5.2. 打开 Web 浏览器并导航到应用路由。该页面显示 Nexus 应用。



如果要以 **admin** 用户身份登录，您必须从容器的 **/nexus-data/admin.password** 文件中获取密码。此测试过程中不需要登录。

## ▶ 6. 删除 OpenShift 中的项目，以及外部容器注册表中的容器和镜像。由于 Quay.io 允许恢复旧的容器镜像，因此还必须在 Quay.io 上删除存储库。

- 6.1. 删 除 OpenShift 项目：

```
[student@workstation ~]$ oc delete project ${RHT_OCP4_DEV_USER}-nexus-service
project.project.openshift.io "youruser.nexus-service" deleted
```

- 6.2. 从外部注册表中删除容器镜像：

```
[student@workstation ~]$ skopeo delete \
docker://quay.io/${RHT_OCP4_QUAY_USER}/nexus3
```

- 6.3. 使用您的个人免费帐户登录 Quay.io。

导航到 <https://quay.io> 并单击 Sign In 以提供用户凭据。登录 Quay.io。

- 6.4. 在 Quay.io 主菜单上，单击 **Repositories** 并查找 **nexus3**。单击 **nexus3** 以显示 **Repository Activity** 页面。

- 6.5. 在 **nexus3** 存储库的 **Repository Activity** 页面，向下滚动并单击齿轮图标以显示 **Settings** 选项卡。向下滚动并单击 **Delete Repository**。

- 6.6. 在 **Delete** 对话框中，输入存储库名称，单击 **Delete** 以确认要删除 **nexus3** 存储库。几分钟后，您将返回到 **Repositories** 页面。您现在可以注销 Quay.io。

## 完成

在 **workstation** 上运行 **lab nexus-service finish** 命令以完成练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab nexus-service finish
```

本引导式练习到此结束。

# 使用 JKube 部署云原生应用

## 培训目标

学完本节后，学员应能够使用 Eclipse JKube 部署云原生应用。

## 原生云应用

云原生是一个非常宽泛的技术描述词汇。这些技术的设计宗旨是在公共云、私有云和混合云中构建和运行可扩展的应用，例如容器、微服务、Kubernetes 及其他现代技术等。

云原生技术可实现弹性、可维护性和可观察性，其流程可以自动化，以实现频繁的更新和部署。例如，开发人员使用 Quarkus 或 JKube 等云原生工具就能创建容器镜像，无需手动创建 Dockerfile。

当应用部署在 OpenShift 中并设计为使用平台提供的服务时，即被视为云原生应用。

## Eclipse JKube

Eclipse JKube 是一组开源插件和库，可以通过不同的策略构建容器镜像，生成 Java 应用并部署到 Kubernetes 和 OpenShift 中，几乎不需配置就能让您的应用成为云原生应用。

JKube 是重构和重塑 Fabric8 的结晶，它导致 Fabric8 生态系统分离，现在是一个面向 Kubernetes 和 OpenShift 的应用的更通用插件。

JKube 由 **JKube Kit**（用于构建镜像和生成部署清单的工具集）、Kubernetes Maven 插件以及 OpenShift Maven 插件构成。这些插件用于部署到对应的集群类型。

JKube 通过其 Maven 插件支持多种 Java 框架，例如：

### Quarkus

一款现代全堆栈云原生框架，具有占用内存少且启动时间短的特点。

### Spring Boot

基于流行的 Spring 框架和自动配置的云原生开发框架。

### Vert.x

基于异步 I/O 和事件流的被动、低延迟开发框架。

### Micronaut

一个现代全堆栈工具包，用于构建易于测试的模块化微服务和无服务器应用。

### Open Liberty

适用于 Java 应用程序的灵活服务器运行时。

## 使用 JKube 的开发工作流

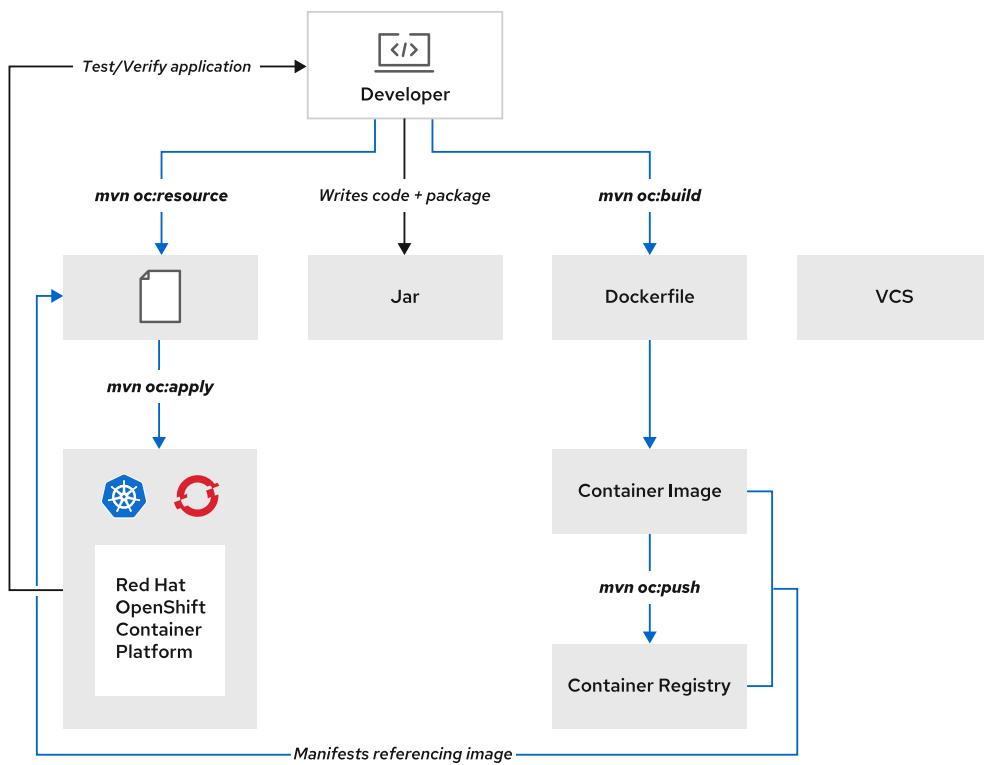


图 8.2: Eclipse JKube 开发人员工作流程。

使用 JKube 已成为开发人员工作流程的一部分。不同的目标有助于开发人员创建资源描述符，将它们部署到 OpenShift 集群，构建容器镜像，并且推送到所需的注册表。

JKube 的 Maven 插件还具有不在本节讨论范围内的其他功能，这些功能可辅助开发人员改进工作流。

通过利用此工作流，开发人员可以在开发环境中为实际部署描述符执行零配置部署，以及内联配置和外部配置部署。

## Kubernetes Maven 插件

借助 JKube 的 Kubernetes Maven 插件，开发人员可以生成容器镜像、部署描述符和配置部署。

## OpenShift Maven 插件

JKube OpenShift Maven 插件基于 Kubernetes Maven 插件构建，是开源 Eclipse JKube 项目的核心组件之一。

您可以使用这个插件来使用二进制构建输入源将 Java 应用部署到 OpenShift 集群。这意味着插件编译和打包应用，使用打包的工件来构建容器镜像，并创建 OpenShift 资源以将应用部署到 OpenShift。

## OpenShift Maven 插件配置

要在项目中使用 JKube OpenShift Maven 插件，请更新项目的 `pom.xml` 文件以启用和配置插件。您必须在 `pom.xml` 的 `build` 部分中，向 `plugins` 列表中添加一个 `plugin` 条目。以下配置是为 Java 应用启用 JKube OpenShift Maven 插件的最小配置：

```
...output omitted...
<build>
  <plugins>
    <plugin>①
      <groupId>org.eclipse.jkube</groupId>
      <artifactId>openshift-maven-plugin</artifactId>
      <version>1.2.0</version>
      <executions>②
        <execution>
          <goals>
            <goal>resource</goal>
            <goal>build</goal>
            <goal>apply</goal>
          </goals>
        </execution>
      </executions>
      <configuration>③
        <!-- additional configuration here -->
      </configuration>
    </plugin>
  </plugins>
</build>
```

- ① 为 Maven 提供查找、下载和使用插件所需的信息。
- ② 为 Maven 配置标准的 `install` 目标。这完全可选。
- ③ 可以更为具体的其他配置，允许您为副本集配置镜像、资源、卷，以及用于精细粒度详细信息的服务和配置映射。

将此 XML 段添加到项目的 `pom.xml` 文件中，以添加 JKube OpenShift Maven 插件。有关 JKube OpenShift Maven 配置的更多详细信息，请参阅本讲义末尾的参考资料。

还支持修改几个不同目标的属性。您可通过这些属性修改默认行为，例如每当运行 `oc:apply` 时强制编辑资源、强制使用 `Deployment` 对象而不使用 `DeploymentConfig`，以及改变其他默认行为。您可以在本节末尾的参考资料页面上阅读有关这些属性的更多内容。

## OpenShift Maven 插件目标

Maven 插件目标代表软件开发生命周期过程中一个定义明确的任务。您可以使用 `mvn` 命令来执行 Maven 插件目标：

```
[user@host sample-app]$ mvn <plug-in goal name>
```

OpenShift Maven 插件提供了一组目标来处理云原生 Java 应用的开发：

**oc:resource**

创建 Kubernetes 和 OpenShift 资源描述符。插件将生成的所有描述符存储在项目的 `target/classes/META-INF/openshift` 子目录。

**oc:build**

编译并打包 Java 应用以创建二进制工件，然后使用该工件来构建关联的应用容器镜像。

插件使用生成器来自动检测编译和打包应用所需的构建参数。特别重要的是，生成器标识了用于应用的相应构建器镜像。对于通用 Java 应用，默认的构建器镜像是 `quay.io/jkube/jkube-java-binary-s2i`。

如果插件检测到对 OpenShift 集群的访问，则将启动 OpenShift 二进制构建。该插件同时支持源至镜像和 Docker 二进制构建。默认情况下，插件会执行源至镜像构建策略。

对于 OpenShift 构建，插件会在 OpenShift 集群中创建应用构建配置和镜像流资源。对于源至镜像和 Docker 构建策略，插件都使用新的容器镜像更新镜像流。

**oc:apply**

将生成的资源应用到连接的 OpenShift 集群。

**oc:deploy**

与 `oc:apply` 类似，区别在于其在后台运行。

**oc:undeploy**

从 OpenShift 集群删除资源。

**oc:watch**

监视文件的更改，然后触发重新构建和重新部署。这在开发过程中很有用处。

JKube OpenShift Maven 插件还支持其他目标，但这已超出本课程讨论范围。如需这些目标的更多详细信息，请参阅本讲义末尾的参考资料。

**重要**

大多数情形中仍然需要运行 `package` 目标，运行次序在上文提及的 `oc` 目标之前。这意味着您需要一个对应的构建插件来支持您的框架或运行时，以便构建 `package`，然后您可以使用 JKube 构建镜像、资源和部署。

## 自定义 OpenShift 资源

JKube OpenShift Maven 插件只需极少的项目配置，即可生成一组 OpenShift 资源来支持在 OpenShift 中部署应用。在某些场景中，生成的 OpenShift 资源可能不足以用于 OpenShift 部署。

从以下两种方式中选择一种来自定义生成的资源：将 OpenShift 资源 YAML 片段添加到项目的 `src/main/jkube` 子目录，或在项目的 `pom.xml` 文件中添加其他配置。在本课程中，您将使用 YAML 资源片段为项目自定义 OpenShift 配置。

如果只向项目添加少许 JKube 配置，则插件将为您的应用创建服务和部署配置资源。如果关联的容器镜像公开端口，则插件还会创建路由资源以公开服务。插件将每个资源定义写入 `target/classes/META-INF/jkube/openshift` 目录中的文件。所有这些资源定义都合并到一个 `openshift.yml` 文件中，保存在项目的 `target/classes/META-INF/jkube` 子目录中。

插件也会处理 `src/main/jkube` 目录中的 YAML 片段文件，并使用每个片段中的内容来覆盖对应的默认资源定义。各个文件的内容模仿 OpenShift 资源的结构，但省略任何未更改的信息。这些文件以小而紧凑为目标，仅表示必须从默认资源配置更改的配置。

## 章 8 | 面向 OpenShift 构建应用

每个片段文件遵循文件命名约定。插件使用片段文件名来标识要覆盖的 OpenShift 资源。片段文件名遵循 **[name] -type.yml** 模式。

**name** 选填，表示资源的名称。如果未提供名称，则插件将使用应用名称来作为资源名称。

**type** 对于此片段修改的 OpenShift 资源的类型。类型值必须为以下值之一：

类型	文件名
服务	svc、service
路由	route
部署	deployment
DeploymentConfig	dc、deploymentconfig
ConfigMap	cm、configmap
机密	secret

例如，请考虑如下 **src/main/jkube/route.yml** 的内容：

```
spec:  
  host: app.alternate.com
```

此片段将应用路由的默认主机名更改为 app.alternate.com。

您也可以使用 **src/main/jkube/cm.yml** 文件创建 ConfigMap 对象。例如：

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: example-data  
data:  
  MSG_TEXT: This is a text value
```



### 参考文献

JKube OpenShift Maven 插件的开源文档可从以下位置找到：  
<https://www.eclipse.org/jkube/docs/openshift-maven-plugin>

## ► 指导练习

# 使用 JKube 部署应用

在本练习中，您将使用 Eclipse JKube Maven 插件向 OpenShift 集群部署微服务。

## 成果

您应能够：

- 使用 Eclipse JKube Maven 插件配置自定义 OpenShift 部署资源。
- 使用 Eclipse JKube Maven 插件配置 OpenShift 配置映射资源。
- 使用 Eclipse JKube Maven 插件部署微服务。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Git 存储库中的微服务应用 (**micro-java**)。

在 **workstation** 上运行以下命令，以验证前提条件并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab micro-java start
```

## 说明

### ► 1. 准备您的课堂环境。

- 1.1. 加载您的课堂环境配置。运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation micro-java]$ source /usr/local/etc/ocp4.config
```

- 1.2. 使用您的开发人员用户名登录 OpenShift。

```
[student@workstation micro-java]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful.
...output omitted...
```

- 1.3. 为应用创建一个新项目。使用您的开发人员用户名为项目的名称加上前缀。

```
[student@workstation micro-java]$ oc new-project ${RHT_OCP4_DEV_USER}-micro-java
Now using project "youruser-micro-java" on server ...
...output omitted...
```

- 2. 检查 `micro-java` 示例应用的 Java 源代码。这个应用是一种 Java Quarkus 微服务，可以部署到非 Kubernetes 环境中。您将添加所需的配置到应用，以便部署到 OpenShift 容器平台集群。

- 2.1. 进入 `DO288-apps` Git 存储库本地克隆的 `micro-java` 子目录。签出课程存储库的 `main` 分支，以确保从已知良好的状态开始本练习：

```
[student@workstation ~]$ cd ~/DO288-apps/micro-java  
[student@workstation micro-java]$ git checkout main  
...output omitted...
```

- 2.2. 创建一个新分支，以保存您在本练习中所做的任何更改：

```
[student@workstation micro-java]$ git checkout -b micro-config  
Switched to a new branch 'micro-config'  
[student@workstation micro-java]$ git push -u origin micro-config  
...output omitted...  
* [new branch]      micro-config -> micro-config  
Branch micro-config set up to track remote branch micro-config from origin.
```

- 2.3. 检查 `HelloResource.java` 源代码的 `src/main/java/com/redhat/training/openshift/hello` 子目录中的 `micro-java` 源代码文件：

```
package com.redhat.training.openshift.hello;  
  
import java.util.Optional;  
  
import javax.ws.rs.Consumes;  
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.Produces;  
import javax.ws.rs.core.MediaType;  
  
import org.eclipse.microprofile.config.inject.ConfigProperty;  
  
@Path("/api")  
public class HelloResource {①  
  
    @ConfigProperty(name = "HOSTNAME", defaultValue = "unknown")  
    String hostname;  
    @ConfigProperty(name = "APP_MSG")②  
    Optional<String> message;  
  
    @GET  
    @Path("/hello")③  
    @Produces("text/plain")  
    public String hello() {  
        String response = "";  
  
        if (!message.isPresent()) {  
            response = "Hello world from host " + hostname + "\n";④  
        } else {  
            response = "Hello world from host [" + hostname + "].\n";  
        }  
    }  
}
```

```

        response += "Message received = " + message + "\n"; ⑤
    }
    return response;
}
}

```

- ① 此类为应用定义 REST 资源。
- ② 应用使用 `HOSTNAME` 和 `APP_MSG` 环境变量，这些变量通过 `ConfigProperty` 注释注入。
- ③ 由于您是在 `/api` 处访问资源的，因此需要在 `/api/hello` 处访问此端点。
- ④ 如果未定义 `APP_MSG` 环境变量，则应用资源会响应一条消息，其中包含运行应用所在服务器的主机名。
- ⑤ 如果定义了 `APP_MSG` 环境变量，则响应消息会包含 `HOSTNAME` 和 `APP_MSG` 环境变量的值。

► 3. 使用 JKube 插件配置应用以部署到 OpenShift。检查项目 `pom.xml` 文件中 JKube Maven 插件的配置：

- 3.1. 更新靠近顶层的 `pom.xml` 文件的项目级属性，以及 JKube 配置文件。将 `jkube.build.switchToDeployment` 项目属性设为 `true`。必须执行此项操作，因为 JKube 默认会对 OpenShift 使用 `DeploymentConfig` 对象，但我们要改为使用 `Deployment` 对象。

```

<?xml version="1.0" encoding="UTF-8"?>
...output omitted...
<groupId>com.redhat.training.openshift</groupId>
<artifactId>micro-java</artifactId>①
<version>1.0</version>②
...output omitted...
<properties>
...output omitted...
<jkube.build.switchToDeployment>true</jkube.build.switchToDeployment>
</properties>
...output omitted...

```

- ① 应用的名称和版本。JKube Maven 插件从这些值创建镜像流标签资源，`micro-java:1.0`。
  - ② 这些版本属性设置 Quarkus 应用的版本。
- 3.2. 查看 `pom.xml` 文件末尾附近 `build` 部分的插件列表，再将以下插件添加到构建的末尾：

```

...output omitted...
<build>
  <plugins>
...output omitted...
  <!-- JKube Maven plugin -->
  <plugin>
    <groupId>org.eclipse.jkube</groupId>①

```

```
<artifactId>openshift-maven-plugin</artifactId>
<version>1.2.0</version>
</plugin>
...output omitted...
```

**①** 我们使用 `openshift-maven-plugin` 插件执行 OpenShift 的所有构建和部署。

- 3.3. 打开 `src/main/jkube/cm.yml` 文件。此文件描述了部署期间在 OpenShift 项目中创建的名为 `configmap-hello` 的 ConfigMap。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-hello
data:
```

- 3.4. 打开 `src/main/jkube/deployment.yml` 文件。此文件覆盖 JKube 使用的一些默认值。为容器添加 `envFrom` 属性，您可以从 `configmap-hello` ConfigMap 使用此属性来设置环境变量。通过添加对 ConfigMap 的这一引用，定义的条目可以作为环境变量提供给应用。

```
...output omitted...
spec:
  containers:
    - env:
        - name: KUBERNETES_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      envFrom:
        - configMapRef:
            name: configmap-hello
      image: micro-java:1.0
      imagePullPolicy: IfNotPresent
      name: quarkus
  ...output omitted...
```

#### ▶ 4. 为应用生成 OpenShift 资源。

- 4.1. 从项目目录，执行 `mvn package oc:build oc:resource`。`package` 会创建一个 JAR 文件供 JKube 用于部署。`oc:build` 会在 OpenShift 上执行 s2i 构建，从而创建名为 `micro-java` 的镜像并供 OpenShift 项目使用。`oc:resource` 会创建多个 YAML 格式的资源文件，为应用做好部署准备：

```
[student@workstation micro-java]$ mvn -DskipTests package oc:build oc:resource
...output omitted...
[INFO] -----
[INFO] Building jar: /home/student/D0288-apps/micro-java/target/micro-
java-1.0.jar①
[INFO] -----
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:build (default-cli) @ micro-java - ②
[INFO] oc: Using OpenShift build with strategy S2I
```

```
[INFO] oc: Running in OpenShift mode
[INFO] oc: Running generator quarkus
[INFO] oc: quarkus: Using Docker image quay.io/jkube/jkube-java-binary-s2i:0.0.9
as base / builder
[INFO] oc: [micro-java:latest] "quarkus": Created docker source tar /home/student/
D0288-apps/micro-java/target/docker/micro-java/latest/tmp/docker-build.tar
[INFO] oc: Updating BuildServiceConfig micro-java-s2i for Source strategy
[INFO] oc: Adding to ImageStream micro-java
[INFO] oc: Starting Build micro-java-s2i
[INFO] oc: Waiting for build micro-java-s2i-2 to complete...
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:resource (default-cli) @ micro-java -③
[INFO] oc: Using docker image name of namespace: your-project
[INFO] oc: Running generator quarkus
[INFO] oc: quarkus: Using Docker image quay.io/jkube/jkube-java-binary-s2i:0.0.9
as base / builder
[INFO] oc: Using resource templates from /home/student/D0288-apps/micro-java/src/
main/jkube
[INFO] oc: jkube-service: Adding a default service 'micro-java' with ports [8080]
...output omitted...
```

- ① **package** 目标会生成一个 jar 文件，该文件在其余步骤中都会用到。您也可以具有原生构建，生成原生的可执行文件。
- ② **oc:build** 目标通过构建镜像来容器化应用。
- ③ **oc:resource** Maven 目标创建三个 YAML 文件。这些文件一起定义示例应用的服务、部署和路由资源。这些文件位于项目 **target/classes/META-INF/jkube/openshift** 子目录中。

#### 4.2. 检查 `target/classes/META-INF/jkube/openshift` 目录中生成的 `micro-java-deployment.yml` 文件。

```
apiVersion: apps.openshift.io/v1
kind: Deployment
metadata:
...output omitted...
  name: micro-java
spec:
  replicas: 1
...output omitted...
  template:
    spec:
      containers:
        - env:
            - name: KUBERNETES_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            envFrom:
              - configMapRef:
                  name: configmap-hello
        image: micro-java:1.0
...output omitted...
```

该文件定义名为 **micro-java** 的部署，此部署使用 **micro-java:1.0** 镜像流标签中的镜像部署单个容器。此外还会使用 **src/main/jkube/deployment.yaml** 文件中所指定的配置映射。

- 4.3. 检查 **target/classes/META-INF/jkube/openshift** 目录中生成的 **micro-java-service.yml** 文件。

```
---  
apiVersion: v1  
kind: Service  
metadata:  
...output omitted...  
  name: micro-java  
spec:  
  ports:  
    - name: http  
      port: 8080  
      protocol: TCP  
      targetPort: 8080  
  selector:  
    app: micro-java  
    provider: jkube  
    group: com.redhat.training.openshift
```

此文件为 **micro-java** 部署中定义的应用容器集定义一个名为 **micro-java** 的服务。

- 4.4. 检查 **target/classes/META-INF/jkube/openshift** 目录中生成的 **micro-java-route.yml** 文件。

```
apiVersion: route.openshift.io/v1  
kind: Route  
metadata:  
...output omitted...  
  name: micro-java  
spec:  
  port:  
    targetPort: 8080  
  to:  
    kind: Service  
    name: micro-java
```

此文件定义名为 **micro-java** 的路由资源，该资源公开 **micro-java** 服务。

- 4.5. **oc:resource** Maven 目标还生成所有资源的组合列表。检查项目 **target/classes/META-INF/jkube** 子目录中生成的 **openshift.yml** 文件。

```
...output omitted...  
kind: "List"  
...output omitted...  
kind: "Service"  
...output omitted...  
kind: "Deployment"
```

```
...output omitted...
kind: "Route"
...output omitted...
```

## ▶ 5. 构建并部署应用。

- 5.1. 使用 JKube Maven 插件构建和部署应用：监控输出以验证构建是否在 OpenShift 模式下运行，并且验证是否创建了 OpenShift 资源。

```
[student@workstation micro-java]$ mvn -DskipTests oc:deploy
...output omitted...
[INFO] >>> openshift-maven-plugin:1.2.0:deploy (default-cli) > install @ micro-
java >>
[INFO]
...output omitted...
[INFO] --- openshift-maven-plugin:1.2.0:deploy (default-cli) @ micro-java -①
[INFO] oc: Using OpenShift at https://api.ocp4.example.com:6443 in namespace your-
project with manifest /home/student/D0288-apps/micro-java/target/classes/META-INF/
jkube/openshift.yml ②
[INFO] oc: OpenShift platform detected
[INFO] oc: Creating a Service from openshift.yml namespace your-project name
micro-java ③
[INFO] oc: Created Service: target/jkube/applyJson/your-project/service-micro-
java.json
[INFO] oc: Creating a ConfigMap from openshift.yml namespace your-project name
configmap-hello
[INFO] oc: Created ConfigMap: target/jkube/applyJson/your-project/configmap-
configmap-hello.json
[INFO] oc: Creating a Deployment from openshift.yml namespace your-project name
micro-java
[INFO] oc: Created Deployment: target/jkube/applyJson/your-project/deployment-
micro-java.json
[INFO] oc: Creating Route your-project:micro-java host: null
[INFO] oc: HINT: Use the command oc get pods -w to watch your pods start up
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 20.921 s④
...output omitted...
```

- ① 部署阶段开始。
- ② 插件可检测活动的 OpenShift 项目。此项目中创建所有资源。
- ③ 插件使用 `openshift.yml` 创建 OpenShift 服务、部署和路由资源。
- ④ 构建时间应当不到一分钟。

如果命令成功完成，您应当能够继续完成活动。

## ▶ 6. 检查 JKube Maven 插件创建的资源。

```
[student@workstation micro-java]$ oc status
In project youruser-micro-java on server ...

http://micro-java-youruser... to pod port 8080 (svc/micro-java)
  deployment/micro-java deploys ...
  deployment #1 deployed 2 minutes ago - 1 pod
bc/micro-java-s2i source builds uploaded code on quay.io/jkube/jkube-java-binary-
s2i:0.0.9
  -> istag/micro-java:1.0
...output omitted...
```

项目中存在路由、服务和部署资源，均命名为 **micro-java**。项目中还存在构建配置和镜像流标签资源。

#### ▶ 7. 测试应用。

- 等待部署新的应用容器集。当 `oc get pods -w` 的输出指示新部署已准备就绪并运行时，请继续执行下一步。

```
[student@workstation micro-java]$ oc get pods -w
NAME                      READY   STATUS    RESTARTS   AGE
micro-java-a1b2c3d4e5f6-5pw6q   1/1     Running   1          14m
micro-java-s2i-1-build        0/1     Completed  0          16m
```

- 测试从外部访问应用。记得在路由 URL 的末尾添加 `/api/hello`，以访问应用的 `HelloResource` REST 资源。

```
[student@workstation D0288-apps]$ ROUTE_URL=$(oc get route \
micro-java --template='{{.spec.host}}')
[student@workstation D0288-apps]$ curl ${ROUTE_URL}/api/hello
Hello world from host micro-java-1-5pw6q
```

由于应用部署没有定义 `APP_MSG` 环境变量的值，因此输出仅包含容器主机名。

#### ▶ 8. 更新项目，以使用 `APP_MSG` 环境变量的新值部署应用容器集。

- 更新 `src/main/jkube/cm.yaml` 片段文件，为其提供新的 `APP_MSG` 值。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-hello
data:
  APP_MSG: this is a new value
```

前面的 YAML 片段定义了名为 `configmap-hello` 的配置映射资源。此配置映射定义值为 `sample external configuration` 的 `APP_MSG` 变量。

- 提交 YAML 片段并推送到 `micro-config` 分支：

```
[student@workstation micro-java]$ git add src/main/jkube/*.yml  
[student@workstation micro-java]$ git commit -am "Add YAML fragments."  
[student@workstation micro-java]$ git push  
...output omitted...
```

- ▶ 9. 重新部署应用。验证 JKube Maven 插件是否创建了配置映射资源。验证应用是否使用配置映射中的 APP\_MSG 变量值来响应。

9.1. 使用 JKube Maven 插件重新部署应用：

```
[student@workstation micro-java]$ mvn -DskipTests oc:build oc:resource oc:apply  
...output omitted...  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
...output omitted...
```

9.2. 验证是否存在 configmap-hello 配置映射：

```
[student@workstation micro-java]$ oc get cm/configmap-hello  
NAME          DATA   AGE  
configmap-hello    1     84m
```

9.3. 等待部署新的应用容器集。如上一步中那样，使用 oc get pods -w 命令等待。

```
[student@workstation micro-java]$ oc get pods -w  
NAME        READY   STATUS    RESTARTS   AGE  
micro-java-1a2b3c4d5e6f-n2rn2   1/1     Running   0          108s  
micro-java-s2i-1-build   0/1     Completed   0          15m
```

9.4. 验证应用响应是否包括 APP\_MSG 变量的新值：

```
[student@workstation micro-java]$ curl ${ROUTE_URL}/api/hello  
Hello world from host [micro-java-3-m9k4j].  
Message received = this is a new value
```



### 注意

如果前一个 curl 返回错误 HTML 页面，指出该应用不可用，则表示应用容器仍未就绪，无法为请求提供服务。等待几秒钟，然后再次尝试同一个命令。

- ▶ 10. 清理：删除在此练习中创建的所有资源。

```
[student@workstation micro-java]$ oc delete project \${RHT_OCP4_DEV_USER}-micro-java
```

## 完成

在 `workstation` 上，执行 `lab micro-java finish` 脚本来完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。此完成操作将释放此项目及其资源。

```
[student@workstation ~]$ lab micro-java finish
```

本引导式练习到此结束。

## ▶ 开放研究实验

# 为 OpenShift 构建云原生应用

在本实验中，您将使用 Eclipse JKube Maven 插件将应用部署到 OpenShift，该应用与在 OpenShift 集群外部运行的数据库进行通信。

## 成果

您应能够：

- 为应用创建指向 OpenShift 集群外部的 MariaDB 数据库服务器的数据库服务。
- 使用 JKube YAML 片段配置自定义 OpenShift 资源。
- 使用 JKube Maven 插件部署 **To Do List** 后端应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Git 存储库中的 **todo-api** 示例应用。
- 外部 MariaDB 数据库服务器。

在 **workstation** 上运行以下命令，以验证前提条件、填充数据库并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab todo-migrate start
```

## 要求

**To Do List** 后端开发团队正在将 Thorntail 应用迁移到 OpenShift。应用源代码位于所克隆的 Git 存储库的 **todo-api** 子目录中。

您必须根据以下要求，配置应用并将其部署到 OpenShift 集群：

- 项目名称为 **youruser-todo-migrate**。您的开发人员用户必须拥有该项目。
- 连接到外部数据库的 OpenShift 项目中存在一个 **tododb** 服务。外部数据库的 FQDN 为 mysql.ocp-eu46a.prod.ole.redhat.com。
- OpenShift 配置映射资源作为部署的一部分创建。

配置映射定义访问外部数据库所需的变量：

使用 JKube YAML 片段创建配置映射资源。

- DATABASE\_USER**: **todoapp**
- DATABASE\_PASSWORD**: **redhat123**
- DATABASE\_SVC\_HOSTNAME**: **tododb**

- **DATABASE\_NAME: todo**

- 自定义 OpenShift 部署资源已作为部署的一部分创建。

使用 JKube YAML 片段将配置映射资源中的所有变量作为环境变量添加到应用容器中。

- 将所有代码更改提交到远程 Git 存储库。

要测试部署是否成功，应用 `todo/api/items/6` 端点应返回 JSON 数据。

## 说明

1. 验证与外部数据库服务器的连接。
2. 创建一个名为 `tododb` 的 OpenShift 服务，该服务连接到外部数据库实例。服务必须在 `yourdevuser-todo-migrate` 项目中创建。
3. 从 `D0288-apps` 存储库本地克隆中的 `master` 分支创建 `todo-migrate` 分支。更改到 `todo-api` 项目子目录。
4. 构建并部署应用。验证应用容器集是否因为未定义所需的环境变量而无法部署。
5. 创建 JKube 用于生成自定义配置映射资源的 YAML 片段，该资源定义数据库环境变量。您可以选择在自定义配置映射资源中定义所需的环境变量，或直接在部署配置中定义。
6. 创建 JKube 用于生成自定义部署配置资源的 YAML 片段。
7. 将自定义配置映射和部署配置资源应用到项目。验证应用是否部署且无任何问题。使用外部路由测试对应用 `todo/api/items/6` 资源的访问。成功的响应将返回 JSON 数据。
8. 应用测试成功后，提交代码更改并将其推送到远程存储库。

## 评估

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab todo-migrate grade
```

## 完成

在 `workstation` 上运行 `lab todo-migrate finish` 命令以完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab todo-migrate finish
```

本实验到此结束。

## ► 解决方案

# 为 OpenShift 构建云原生应用

在本实验中，您将使用 Eclipse JKube Maven 插件将应用部署到 OpenShift，该应用与在 OpenShift 集群外部运行的数据库进行通信。

## 成果

您应能够：

- 为应用创建指向 OpenShift 集群外部的 MariaDB 数据库服务器的数据服务。
- 使用 JKube YAML 片段配置自定义 OpenShift 资源。
- 使用 JKube Maven 插件部署 **To Do List** 后端应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- Git 存储库中的 **todo-api** 示例应用。
- 外部 MariaDB 数据库服务器。

在 **workstation** 上运行以下命令，以验证前提条件、填充数据库并下载完成此练习所需的文件：

```
[student@workstation ~]$ lab todo-migrate start
```

## 要求

**To Do List** 后端开发团队正在将 Thorntail 应用迁移到 OpenShift。应用源代码位于所克隆的 Git 存储库的 **todo-api** 子目录中。

您必须根据以下要求，配置应用并将其部署到 OpenShift 集群：

- 项目名称为 **youruser-todo-migrate**。您的开发人员用户必须拥有该项目。
- 连接到外部数据库的 OpenShift 项目中存在一个 **tododb** 服务。外部数据库的 FQDN 为 mysql.ocp-eu46a.prod.ole.redhat.com。
- OpenShift 配置映射资源作为部署的一部分创建。

配置映射定义访问外部数据库所需的变量：

使用 JKube YAML 片段创建配置映射资源。

- DATABASE\_USER**: todoapp
- DATABASE\_PASSWORD**: redhat123
- DATABASE\_SVC\_HOSTNAME**: tododb
- DATABASE\_NAME**: todo

## 章 8 | 面向 OpenShift 构建应用

- 自定义 OpenShift 部署资源已作为部署的一部分创建。
- 使用 JKube YAML 片段将配置映射资源中的所有变量作为环境变量添加到应用容器中。
- 将所有代码更改提交到远程 Git 存储库。
- 要测试部署是否成功，应用 `todo/api/items/6` 端点应返回 JSON 数据。

## 说明

1. 验证与外部数据库服务器的连接。

- 1.1. 加载您的课堂环境配置。

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

- 1.2. 连接到外部 MySQL 数据库。`RHT_OCP4_MYSQL_SERVER` 变量包含数据库服务器的主机名。

```
[student@workstation ~]$ mysql -h${RHT_OCP4_MYSQL_SERVER} -utodoapp -predhat123
      todo
Reading table information for completion of table and column names
...output omitted...
mysql>
```

- 1.3. 退出 MySQL 客户端以返回 shell 提示符。

```
mysql> exit
Bye
[student@workstation ~]$
```

2. 创建一个名为 `tododb` 的 OpenShift 服务，该服务连接到外部数据库实例。服务必须在 `yourdevuser-todo-migrate` 项目中创建。

- 2.1. 使用您的开发人员用户帐户登录 OpenShift:

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

- 2.2. 创建新项目来托管应用：

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-todo-migrate
```

- 2.3. 基于外部名称创建服务：

```
[student@workstation ~]$ oc create service externalname tododb \
--external-name ${RHT_OCP4_MYSQL_SERVER}
service "tododb" created
```

2.4. 验证 `tododb` 服务是否存在，并且显示外部 IP 而非集群 IP：

```
[student@workstation ~]$ oc get svc
NAME      TYPE      ...      EXTERNAL-IP      PORT(S)      AGE
tododb    ExternalName  ...  mysql.ocp-eu46a.prod.ole.redhat.com  <none>      6s
```

- 从 `D0288-apps` 存储库本地克隆中的 `master` 分支创建 `todo-migrate` 分支。更改到 `todo-api` 项目子目录。

```
[student@workstation ~]$ cd ~/D0288-apps
[student@workstation D0288-apps]$ git checkout main
...output omitted...
[student@workstation D0288-apps]$ git checkout -b todo-migrate
Switched to a new branch 'todo-migrate'
[student@workstation D0288-apps]$ git push -u origin todo-migrate
...output omitted...
[student@workstation D0288-apps]$ cd todo-api
[student@workstation todo-api]$
```

- 构建并部署应用。验证应用容器集是否因为未定义所需的环境变量而无法部署。

4.1. 编译、构建并部署应用。

```
[student@workstation todo-api]$ mvn clean compile package \
oc:build oc:resource oc:apply
```

4.2. 应用部署后，监控应用容器集的日志。

```
[student@workstation todo-api]$ oc get pods
NAME          READY   STATUS      RESTARTS   AGE
todo-api-558555647-mpg9j  0/1     CrashLoopBackOff  3          101s
todo-api-s2i-1-build       0/1     Completed   0          2m34s
```

容器集的确切 `STATUS` 可能会变，但 `RESTARTS` 的数量会增加。



### 注意

容器集 `STATUS` 可能先是 `Running`，然后再变为 `CrashLoopBackOff` 状态。

```
[student@workstation todo-api]$ oc logs -f todo-api-558555647-mpg9j
...output omitted...
One or more configuration errors have prevented the application from starting. The
errors are:
  - SRCFG00011: Could not expand value DATABASE_USER in property
    quarkus.datasource.username
  - SRCFG00011: Could not expand value DATABASE_PASSWORD in property
    quarkus.datasource.password
  - SRCFG00011: Could not expand value DATABASE_SVC_HOSTNAME in property
    quarkus.datasource.jdbc.url
```

失败原因可能是没有定义 `DATABASE_SVC_HOSTNAME` 和 `DATABASE_NAME` 等变量。

5. 创建 JKube 用于生成自定义配置映射资源的 YAML 片段，该资源定义数据库环境变量。您可以选择在自定义配置映射资源中定义所需的环境变量，或直接在部署配置中定义。

5.1. 创建包含以下内容的 `src/main/jkube/cm.yml` 文件：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  DATABASE_USER: todoapp
  DATABASE_PASSWORD: redhat123
  DATABASE_SVC_HOSTNAME: tododb
  DATABASE_NAME: todo
```

此外，`/home/student/D0288/solutions/todo-migrate` 目录中也提供了答案 YAML 片段文件。您可以将它复制到 `src/main/jkube` 子目录：

```
[student@workstation todo-api]$ cp \
~/D0288/solutions/todo-migrate/cm.yml src/main/jkube
```

6. 创建 JKube 用于生成自定义部署配置资源的 YAML 片段。

6.1. 创建包含以下内容的 `src/main/jkube/deployment.yml` 文件：

```
spec:
  template:
    spec:
      containers:
        - envFrom:
          - configMapRef:
            name: db-config
```

配置映射引用名称必须与配置映射资源的名称匹配。

此外，`/home/student/D0288/solutions/todo-migrate` 目录中也提供了答案 YAML 片段文件。您可以将它复制到 `src/main/jkube` 子目录：

```
[student@workstation todo-api]$ cp \
~/D0288/solutions/todo-migrate/deployment.yml src/main/jkube
```

7. 将自定义配置映射和部署配置资源应用到项目。验证应用是否部署且无任何问题。

使用外部路由测试对应用 `todo/api/items/6` 资源的访问。成功的响应将返回 JSON 数据。

7.1. 将新的 OpenShift 资源应用到您的项目。

```
[student@workstation todo-api]$ mvn oc:resource oc:apply
```

7.2. 检查 JKube Maven 插件创建的资源。

```
[student@workstation todo-api]$ oc describe deployment/todo-api \
| grep -A1 "Environment Variables"
Environment Variables from:
  db-config ConfigMap Optional: false
```

配置映射资源名称必须与含有数据库变量的配置映射资源的名称匹配。

```
[student@workstation todo-api]$ oc get configmap
NAME          DATA   AGE
db-config      4      5m16s
...output omitted...
```

验证应用容器集是否处于 Running 状态。

```
[student@workstation todo-api]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
...output omitted...
todo-api-58fc84655-gs9nv     1/1     Running   0          35s
...output omitted...
```

7.3. 等待应用准备就绪并在运行：

```
[student@workstation todo-api]$ oc logs -f todo-api-58fc84655-gs9nv
...output omitted...
INFO: todo-api 1.0.0-SNAPSHOT on JVM (powered by Quarkus 1.11.7.Final-redhat-00009) started in 3.183s. Listening on: http://0.0.0.0:8080
```

7.4. 验证应用是否从外部数据库获取数据。

使用 curl 命令测试应用 todo/api/items/6 资源是否返回 JSON 数据。

```
[student@workstation todo-api]$ ROUTE_URL=$(oc get route todo-api \
--template={{.spec.host}})
[student@workstation todo-api]$ curl -s ${ROUTE_URL}/todo/api/items/6 \
| jq
{
  "id": 6,
  "description": "Verify that the To Do List application works",
  "done": false
}
```

8. 应用测试成功后，提交代码更改并将其推送到远程存储库。

```
[student@workstation todo-api]$ git add src/main/jkube/*
[student@workstation todo-api]$ git commit -m "add YAML fragments"
...output omitted...
[student@workstation todo-api]$ git push origin todo-migrate
...output omitted...
```

## 评估

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab todo-migrate grade
```

## 完成

在 **workstation** 上运行 **lab todo-migrate finish** 命令以完成本练习。此步骤很重要，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab todo-migrate finish
```

本实验到此结束。

# 总结

---

在本章中，您学到了：

- 服务名称成为 OpenShift 集群内所有容器集的本地 DNS 主机名。
- 外部服务通过使用 `external-name` 选项运行 `oc create service externalname` 命令来创建。
- 红帽建议生产部署定义健康探测。
- 红帽提供了一组中间件容器镜像，用于在 OpenShift 中部署应用，包括打包为可运行的 JAR 文件的应用。
- JKube Maven 插件提供了生成 OpenShift 资源和触发 OpenShift 流程（如构建和部署）的功能。

## 章 9

# 总复习：红帽 OpenShift 开发人员二：构建 Kubernetes 应用

### 目标

回顾 红帽 OpenShift 开发人员二：构建 Kubernetes 应用 中的任务

### 培训目标

回顾 红帽 OpenShift 开发人员二：构建 Kubernetes 应用 中的任务

### 章节

总复习

### 实验

- 实验：在 OpenShift 中构建和部署多容器应用

# 总复习

---

## 培训目标

学完本节后，您应能够回顾并温习 红帽 OpenShift 开发人员二：构建 Kubernetes 应用 中学到的知识和技能。

## 复习 红帽 OpenShift 开发人员二：构建 Kubernetes 应用

在开始进行本课程总复习前，学员应当熟悉各章节涉及的主题。

学员可以参考教材前面的章节，进行额外学习。

### 第 1 章 在 OpenShift 集群中部署和管理应用

利用各种应用封装方法将应用部署至 OpenShift 集群，并管理应用资源。

- 描述 OpenShift 4 的架构和新功能。
- 使用 CLI 通过 Dockerfile 将应用部署至集群。
- 通过容器镜像部署应用，并使用 Web 控制台来管理应用资源。
- 通过源代码部署应用，并使用命令行界面管理应用资源。

### 第 2 章 针对 OpenShift 设计容器化应用

为应用选择应用容器化方法，并封装应用以在 OpenShift 集群上运行。

- 选择适合的应用容器化方法。
- 使用高级 Dockerfile 指令构建容器镜像。
- 选择将配置数据注入应用的方法，并创建完成注入操作所需的资源。

### 第 3 章 发布企业容器镜像

与企业注册表交互，并向其发布容器镜像。

- 使用 Linux 容器工具管理注册表中的容器镜像。
- 使用 Linux 容器工具访问 OpenShift 内部注册表。
- 为外部注册表中的容器镜像创建镜像流。

### 第 4 章 在 OpenShift 上管理构建

介绍 OpenShift 构建流程以及如何触发和管理构建。

- 介绍 OpenShift 构建流程。
- 使用 BuildConfig 资源和 CLI 命令来管理应用的构建。
- 使用受支持的方法触发构建流程。

- 使用 post-commit 构建 hook 处理构建后逻辑。

## 第 5 章 自定义源至镜像构建

自定义现有 S2I 构建器镜像，并创建一个新镜像。

- 介绍在源至镜像构建过程中必须要执行以及可选择性执行的步骤。
- 通过编写脚本，自定义现有 S2I 构建器镜像。
- 使用 S2I 工具新建 S2I 构建器镜像。

## 第 6 章 部署多容器应用

使用 Helm 图表和 Kustomize 部署多容器应用。

- 介绍 OpenShift 模板中的各个元素。
- 使用 Helm 图表构建多容器应用。
- 自定义 OpenShift 部署。

## 第 7 章 管理应用部署

监控应用的健康状况，并为云原生应用实施各种部署方法。

- 实施存活度和就绪度探测。
- 为云原生应用选择适合的部署策略。
- 使用 CLI 命令管理应用部署。

本章的目标不包括在总复习实验中。

## 第 8 章 面向 OpenShift 构建应用

在 OpenShift 中创建和部署应用。

- 将容器化应用与非容器化服务进行整合。
- 按照 OpenShift 推荐做法部署容器化的第三方应用。
- 使用红帽 OpenShift 应用运行时部署应用。

## ▶ 开放研究实验

# 总复习

在本复习中，您将在 OpenShift 中部署一个容器 To Do List 应用。此应用由四个组件组成：

- MySQL (MariaDB) 数据库服务器。
- 基于 Node.js 的 HTTP API 后端。
- 基于 React 和 Nginx 的单页面 Web 前端。
- 用来显示列表中任务状态的任务导出服务。

## 成果

您应能够：

- 通过利用各种构建和部署策略在 OpenShift 上部署多环境应用。
- 优化 Containerfile 以减少生成的容器镜像中的层数。
- 构建容器镜像并发布到外部注册表。
- 创建 Helm 图表以封装可重复利用的配置。
- 传递构建环境变量，从 Git 存储库中的源代码部署 Node.js 应用。
- 创建和使用配置映射来存储应用配置参数。
- 使用源至镜像 (S2I) 来部署应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 由 Bitnami 提供的 MariaDB 10.3 Helm 图表。
- Node.js 12 构建器镜像。
- 个人 GitHub 分叉和 D0288-apps 存储库本地克隆，其 `todo-frontend`、`todo-backend` 和 `todo-ssr` 目录中包含应用的源代码。

在 `workstation` 上运行以下命令，以验证前提条件。该命令还可以下载实验室的帮助文件和答案文件：

```
[student@workstation ~]$ lab review-todo start
```

## 规格

- 将应用的四个组件容器化并部署到名为  `${RHT_OCP4_DEV_USER} -review-todo` 的新 OpenShift 项目。

确保按照红帽的建议部署，以便将部署到 OpenShift 的应用的配置外部化。

- 后端要求：

使用 Helm 图表部署后端和数据库组件。

图表应当从 Quay.io 部署 `quay.io/redhattraining/todo-backend:release-46` 标签。

图表应始终拉取新部署上的镜像。

此外还应当将 **11.0.13** 版 Bitnami 图表用作依赖项来部署 MariaDB 10.3 数据库。MariaDB Bitnami 图表托管在 <https://charts.bitnami.com/bitnami> 存储库中。MariaDB Bitnami 图表需要下列参数：

```
mariadb:  
  auth:  
    username: username  
    password: password  
    database: database  
  primary:  
    podSecurityContext:  
      enabled: false  
    containerSecurityContext:  
      enabled: false
```

API 容器集所需的环境变量为 `DATABASE_USER`、`DATABASE_PASSWORD`、`DATABASE_NAME` 和 `DATABASE_SVC`。

服务应当绑定到端口 3000。

公开公共路由以访问 API。

- 前端 (SPA) 要求：

从 DO288-apps Git 存储库个人分叉的本地副本，检索 `todo-frontend` 源代码和 Containerfile。

提供的 Containerfile 会生成符合开放容器计划 (OCI) 容器引擎的镜像，但可能需要更改以符合红帽对 OpenShift 的建议的要求。

不要更改 HTML 和 TypeScript 源代码。

指定名为 `nginx` 的用户和端口 8080，以修复容器镜像。

尽可能减少容器镜像中的层数。

构建 `todo-frontend` 容器镜像并发布到您的个人 Quay.io 帐户，名称和标签设为 `quay.io/yourquayuser/frontend:latest`。

在  `${RHT_OCP4_DEV_USER} - review - todo` 项目中部署 To Do List UI。

公开公共路由以访问 UI。

使用公开的路由测试 To Do List 前端。

- 前端（静态）要求：

使用源至镜像 (S2I) 策略，在 `DO288-apps` 存储库的 `todo-ssr` 目录中构建并部署应用。

它应当使用由 OpenShift 提供的红帽 Node.js 12 S2I 构建器。

将应用名称 **todo-ssr** 提供给 S2I 构建器。

创建并使用名为 **todo-ssr-host** 的配置映射，将环境变量 **API\_HOST** 设为指向使用 HTTP 协议的端口 3000 上的 **todo-list** 服务。

公开公共路由以访问 UI。

- 提示：

对于后端服务，**/api/items** 端点返回待办事项的当前列表，如果不存在任何项目，则返回 **[]**。

在数据库初始化期间，应用 API 容器集发生失败并重新启动是正常现象。

如果需要卸载 Helm 图表，您还必须删除与 MariaDB 数据库服务器关联的持久卷声明 (PVC)（删除项目也会删除对应 PVC）。

如果您能够创建新的待办事项并可在刷新页面后保留，则 SPA 前端正常工作。

如果要仅删除 UI 资源，请使用 **app=todo-frontend** 选择器。

如果您能够查看列出了使用前端的 SPA 版本创建的待办事项的页面，则静态前端正常工作。

## 评估

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab review-todo grade
```

## 完成

在 **workstation** 计算机上，以 **student** 用户身份使用 **lab** 命令，以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab review-todo finish
```

本总复习到此结束。

## ▶ 解决方案

# 总复习

在本复习中，您将在 OpenShift 中部署一个容器 To Do List 应用。此应用由四个组件组成：

- MySQL (MariaDB) 数据库服务器。
- 基于 Node.js 的 HTTP API 后端。
- 基于 React 和 Nginx 的单页面 Web 前端。
- 用来显示列表中任务状态的任务导出服务。

## 成果

您应能够：

- 通过利用各种构建和部署策略在 OpenShift 上部署多环境应用。
- 优化 Containerfile 以减少生成的容器镜像中的层数。
- 构建容器镜像并发布到外部注册表。
- 创建 Helm 图表以封装可重复利用的配置。
- 传递构建环境变量，从 Git 存储库中的源代码部署 Node.js 应用。
- 创建和使用配置映射来存储应用配置参数。
- 使用源至镜像 (S2I) 来部署应用。

## 在你开始之前

要进行此练习，请确保您有权访问以下资源：

- 运行中的 OpenShift 集群。
- 由 Bitnami 提供的 MariaDB 10.3 Helm 图表。
- Node.js 12 构建器镜像。
- 个人 GitHub 分叉和 D0288-apps 存储库本地克隆，其 `todo-frontend`、`todo-backend` 和 `todo-ssr` 目录中包含应用的源代码。

在 `workstation` 上运行以下命令，以验证前提条件。该命令还可以下载实验室的帮助文件和答案文件：

```
[student@workstation ~]$ lab review-todo start
```

1. 在名为  `${RHT_OCP4_DEV_USER} -review-todo` 的新 OpenShift 项目中，创建一个 Helm 图表来部署 Node.js 后端。新图表应当依赖于 Bitnami MariaDB Helm 图表。创建 API 的新路由，使它公开可用。

1.1. 加载您的课堂环境配置。

运行以下命令，以加载在第一个引导式练习中创建的环境变量：

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

1.2. 使用您的开发人员用户帐户登录 OpenShift：

```
[student@workstation ~]$ oc login -u ${RHT_OCP4_DEV_USER} \
-p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_MASTER_API}
Login successful
...output omitted...
```

1.3. 创建名为 \${RHT\_OCP4\_DEV\_USER}-review-todo 的 OpenShift 项目。

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-review-todo
Now using project "youruser-review-todo" on server ...output omitted...
...output omitted...
```

1.4. 在实验目录中，初始化一个名为 todo-list 的新 Helm 图表，并更改到新创建的目录下。

```
[student@workstation ~]$ cd ~/D0288/labs/review-todo
[student@workstation review-todo]$ helm create todo-list
Creating todo-list
[student@workstation review-todo]$ cd todo-list
[student@workstation todo-list]$
```



### 注意

~/D0288/solutions/review-todo/todo-list 中提供了 Helm 图表的已完成版本。如果对自己的答案不确定，您可以参考或复制提供的答案。

1.5. 将以下内容附加到 Chart.yaml 文件，将 MariaDB 图表声明为依赖项。

```
dependencies:
  - name: mariadb
    version: 11.0.13
    repository: https://charts.bitnami.com/bitnami
```

1.6. 拉取这些依赖项。

```
[student@workstation todo-list]$ helm dependency update
Getting updates for unmanaged Helm repositories...
...Successfully got an update from the "https://charts.bitnami.com/bitnami" chart
repository
Saving 1 charts
Downloading mariadb from repo https://charts.bitnami.com/bitnami
Deleting outdated charts
```

1.7. 在 values.yaml 文件中，更新 image 部分的值。

```
image:
  repository: quay.io/redhattraining/todo-backend
  pullPolicy: Always
  # Overrides the image tag whose default is the chart appVersion.
  tag: "release-46"
```

1.8. 将以下部分附加到 `values.yaml` 的末尾：

```
mariadb:
  auth:
    username: todouser
    password: todopwd
    database: tododb
  primary:
    podSecurityContext:
      enabled: false
    containerSecurityContext:
      enabled: false

env:
  - name: DATABASE_NAME
    value: tododb
  - name: DATABASE_USER
    value: todouser
  - name: DATABASE_PASSWORD
    value: todopwd
  - name: DATABASE_SVC
    value: todo-list-mariadb
```

1.9. 另外在 `values.yaml` 中，将 `service` 中的 `port` 值更新为 3000。

```
service:
  type: ClusterIP
  port: 3000
```

1.10. 更新 `templates/deployment.yaml`，在 `containers` 部分添加 `env` 部分，然后将 `containerPort` 设置为 3000。确保缩进与以下内容相符。

```
apiVersion: apps/v1
...
spec:
  template:
    spec:
      containers:
        - name: {{ .Chart.Name }}
          securityContext:
            {{- toYaml .Values.securityContext | nindent 12 --}}
            image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
```

```
imagePullPolicy: {{ .Values.image.pullPolicy }}
```

```
env:
```

```
  {{- range .Values.env }}
```

```
    - name: {{ .name }}
```

```
      value: {{ .value }}
```

```
  {{- end }}
```

```
ports:
```

```
  - name: http
```

```
    containerPort: 3000
```

```
    protocol: TCP
```

1.11. 安装图表，并命名为 `todo-list`:

```
[student@workstation todo-list]$ helm install todo-list .
```

```
NAME: todo-list
```

```
LAST DEPLOYED: ...output omitted...
```

```
NAMESPACE: youruser-review-todo
```

```
STATUS: deployed
```

```
REVISION: 1
```

```
...output omitted...
```

这将在当前选择的项目中创建 Helm 图表定义的所有资源。

1.12. 验证应用是否成功启动:

```
[student@workstation todo-list]$ oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
todo-list-7b6dbb8ccb-bqvz	1/1	Running	2	5m57s
todo-list-mariadb-0	1/1	Running	0	5m57s



### 注意

如果您的 API 容器集依旧崩溃并重新启动，您必须卸载 Helm 图表，再修复并重试。

1.13. 使用 `oc expose` 命令创建通向 API 的路由。

```
[student@workstation todo-list]$ oc expose svc/todo-list
```

```
route.route.openshift.io/todo-list exposed
```

1.14. 检索服务的公开 URL。

```
[student@workstation todo-list]$ export URL_TO_APPLICATION=$(oc get \
```

```
route/todo-list -o jsonpath='{.spec.host}'')
```

```
...output omitted...
```

1.15. 使用 URL 连接 API，以验证其是否已启动。

```
[student@workstation todo-list]$ curl ${URL_TO_APPLICATION}
```

```
OK
```

1.16. 使用 URL 验证服务是否已连接至数据库。

```
[student@workstation todo-list]$ curl ${URL_TO_APPLICATION}/api/items
[]
```

2. 将一个镜像推送到 Quay.io 以便使用 `new-app` 在 OpenShift 中构建并部署 React UI。修复并构建 `todo-frontend` 镜像，以便其使用 `nginx` 用户，公开端口 8080 并尽可能减少层数。创建 UI 的新路由，使它公开可用。

- 2.1. 从您的 `D0288-apps` 分叉中，打开 `todo-frontend/Containerfile` 文件，将单独的 `RUN` 命令合并到一条指令中。

```
RUN cd /tmp/todo-frontend && \
    npm install && \
    npm run build
```



### 注意

`~/D0288/solutions/review-todo/Containerfile-frontend-solution` 中提供的 `Containerfile` 的固定版本。如果对自己的答案不确定，您可以参考或复制提供的答案。

- 2.2. 另外在 `todo-frontend/Containerfile` 文件中，添加 `EXPOSE` 和 `USER` 命令：

```
COPY --from=appbuild /tmp/todo-frontend/build /usr/share/nginx/html

EXPOSE 8080

USER nginx

CMD nginx -g "daemon off;"
```

- 2.3. 使用 Podman 在本地构建镜像。

```
[student@workstation todo-list]$ cd ~/D0288-apps/todo-frontend/
[student@workstation todo-frontend]$ podman build . \
-t quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend:latest
[1/2] STEP 1/7: FROM registry.access.redhat.com/ubi8/nodejs-14 AS appbuild
Trying to pull registry.access.redhat.com/ubi8/nodejs-14:latest...
Getting image source signatures
...output omitted...
[2/2] COMMIT quay.io/yourquayuser/todo-frontend:latest
--> c3a6b1ed1bd
Successfully tagged quay.io/yourquayuser/todo-frontend:latest
c3a6b1ed1bdc36906775cc234e0368de6f95bbfec0c66c5149474a6b128abfc7
```

请注意，完成此步骤可能需要几分钟。

- 2.4. 在浏览器中，导航到 Quay.io 并新建名为 `todo-frontend` 的空存储库。在名称字段下，将存储库设置为公共存储库。否则，OpenShift 将无法访问它。

- 2.5. 使用 Podman 登录您的个人 Quay.io 帐户。

```
[student@workstation todo-list]$ cd
[student@workstation ~]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password:
Login Succeeded!
```

2.6. 使用 Podman 将镜像推送到 Quay.io。

```
[student@workstation ~]$ podman push --format v2s1 \
quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
Getting image source signatures
...output omitted...
Writing manifest to image destination
Storing signatures
```

请注意，您必须通过 Quay.io 的身份验证。

2.7. 通过 `oc new-app` 使用镜像流来部署该镜像。

```
[student@workstation ~]$ oc new-app quay.io/${RHT_OCP4_QUAY_USER}/todo-frontend
--> Found container image ...output omitted...
...output omitted...
--> Creating resources ...
  imagestream.image.openshift.io "todo-frontend" created
  deployment.apps "todo-frontend" created
  service "todo-frontend" created
--> Success
...output omitted...
```

2.8. 验证应用是否成功启动：

```
[student@workstation ~]$ oc get pods
NAME                      READY   STATUS    RESTARTS   AGE
...output omitted...
todo-frontend-7b4d77b4f8-lsrpm   1/1     Running   0          1m20s
```

2.9. 使用 `oc expose` 命令创建通向服务的路由。

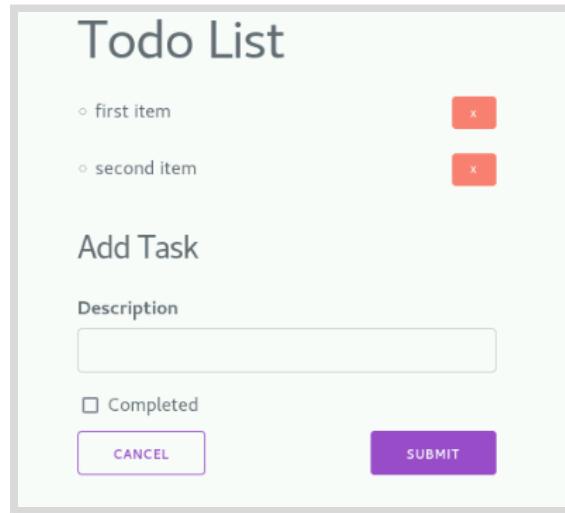
```
[student@workstation ~]$ oc expose svc/todo-frontend
route.route.openshift.io/todo-frontend exposed
```

2.10. 通过检查路由来检索公共 URL。

```
[student@workstation ~]$ oc get route todo-frontend -o jsonpath='{.spec.host}'
todo-frontend-developer-review-todo.apps.ocp4.example.com
```

2.11. 在浏览器中打开检索到的 URL，以验证 UI 是否正常工作。UI 中应包含一个 SPA 待办事项应用。

2.12. 验证您是否能够创建待办事项并可在刷新页面后保留。



3. 使用 S2I 部署服务器端呈现的 Node.js 应用，命名为 **todo-ssr**。其源代码位于 **todo-ssr** 目录中的 DO288-apps 存储库中。确保服务使用端口 3000 运行。创建 UI 的新路由，使它公开可用。

- 3.1. 使用 **new-app** 来利用 S2I 初始化 **todo-ssr** 应用。

```
[student@workstation ~]$ oc new-app \
https://github.com/RedHatTraining/DO288-apps \
--name todo-ssr --context-dir=todo-ssr --build-env \
npm_config_registry="${RHT_OCP4_NEXUS_SERVER}/repository/nodejs"
--> Found image 9350f28 (5 months old) in image stream "openshift/nodejs" under
tag "12-ubi8" for "nodejs"
...output omitted...
--> Creating resources ...
imagestream.image.openshift.io "todo-ssr" created
buildconfig.build.openshift.io "todo-ssr" created
deployment.apps "todo-ssr" created
service "todo-ssr" created
--> Success
...output omitted...
```

- 3.2. 创建名为 **todo-ssr-host** 的配置映射，其中包含 **API\_HOST** 环境变量。

```
[student@workstation ~]$ oc create configmap todo-ssr-host \
--from-literal API_HOST="http://todo-list:3000"
configmap/todo-ssr-host created
```

- 3.3. 将配置映射连接到 **todo-ssr** 部署。

```
[student@workstation ~]$ oc set env deployment/todo-ssr \
--from cm/todo-ssr-host
deployment.apps/todo-ssr updated
```

请注意，连接配置映射后可能需要等待几分钟以便应用重新启动。

- 3.4. 验证应用是否成功启动：

```
[student@workstation ~]$ oc get pods
NAME                      READY   STATUS    RESTARTS   AGE
...output omitted...
todo-ssr-1-build           0/1     Completed   0          6m30s
todo-ssr-64bc5f8987-7654f   1/1     Running    0          1m12s
```

3.5. 使用 `oc expose` 命令创建通向服务的路由。

```
[student@workstation ~]$ oc expose svc/todo-ssr
route.route.openshift.io/todo-ssr exposed
```

3.6. 通过检查路由来检索公共 URL。

```
[student@workstation ~]$ oc get route todo-ssr -o jsonpath='{.spec.host}'
todo-ssr-developer-review-todo.apps.ocp4.example.com
```

3.7. 在浏览器中打开检索到的 URL，验证静态 UI 是否正常工作并单击 **To Do List**。UI 中应包含一个静态页面，其含有您在 SPA 版 UI 中创建的创建的 To Do list 项目。



## 评估

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令为您的成果打分。更正报告的所有错误并重新运行命令，直到成功为止。

```
[student@workstation ~]$ lab review-todo grade
```

## 完成

在 `workstation` 计算机上，以 `student` 用户身份使用 `lab` 命令，以完成本练习。这是重要的一步，可确保前面练习中的资源不会影响后续练习。

```
[student@workstation ~]$ lab review-todo finish
```

本总复习到此结束。

## 附录 A

# 创建 GitHub 帐户

### 目标

介绍如何创建课程中实验所需的 GitHub 帐户。

# 创建 GitHub 帐户

## 培训目标

学完本节后，您应能够创建 GitHub 帐户，并且能够创建公共 Git 存储库。

## 创建 GitHub 帐户

您需要一个 GitHub 帐户，以便为本课程中的实验创建一个或多个公共 Git 存储库。如果您已拥有 GitHub 帐户，您可以跳过此附录中所列的步骤。



### 重要

确保只为本课程中的实验创建 public Git 存储库。实验评分脚本和说明要求通过无需身份验证的访问来克隆存储库。

要创建新的 GitHub 帐户，请前往 <https://github.com>，单击 **Sign up**，然后按照提示操作。您将收到一封电子邮件，其中包含有关如何激活 GitHub 帐户的说明。验证您的电子邮件地址，然后使用您在帐户创建过程中提供的用户名和密码登录 GitHub 网站。

## 创建 GitHub 存储库

通过身份验证后，单击 GitHub 主页左侧 **Repositories** 窗格中的 **New** 来创建新的 Git 存储库。

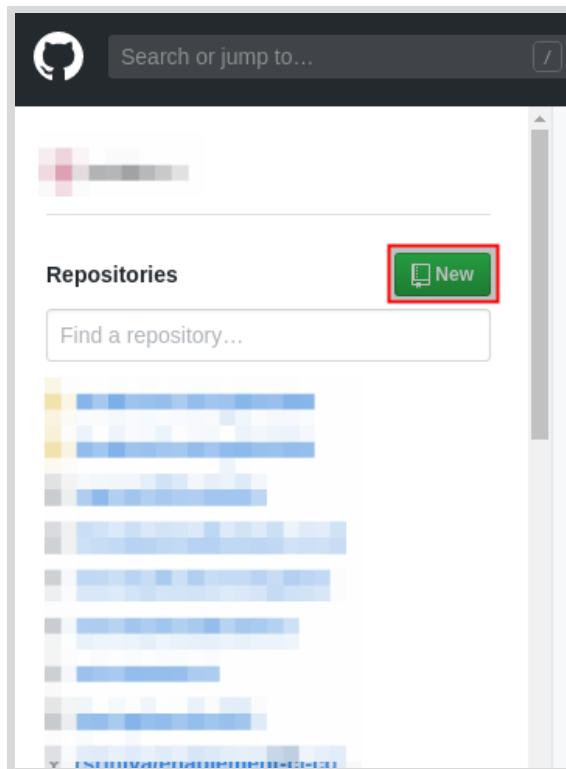


图 A.1: 创建新的 Git 存储库

## 附录 A | 创建 GitHub 帐户

或者，单击右上角（小钟图标右侧）的加号图标（+），然后单击 **New repository**。

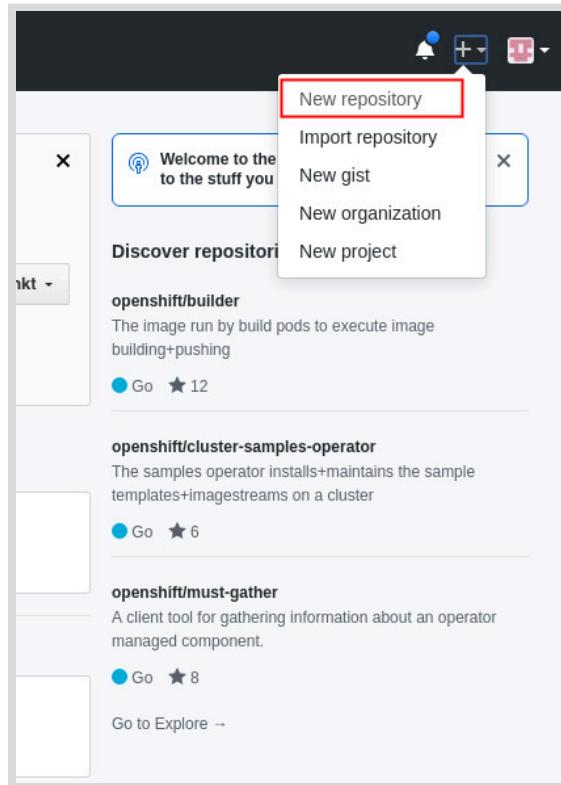


图 A.2: 创建新的 Git 存储库

## 分叉 GitHub 存储库

若要分叉 GitHub 上的存储库，请导航到该存储库，然后单击右上方的 **Fork**。务必单击 **Fork**，而不是按钮旁边的数字。

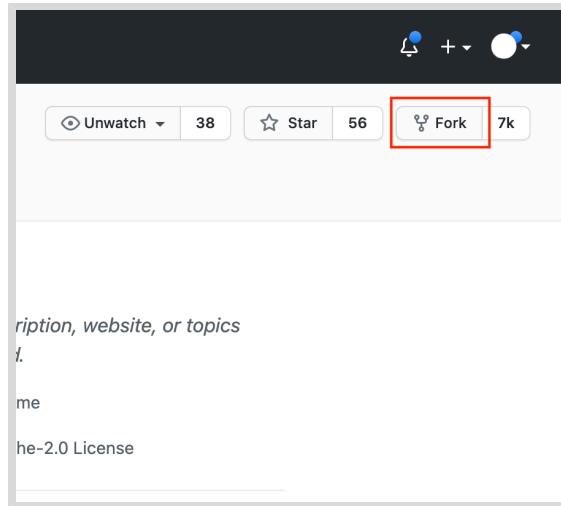


图 A.3: 分叉 Git 存储库

选择您的用户名作为目标，再等待完成相关过程。

在将存储库克隆到工作站时，请使用单击 **Code** 找到的分叉 URL。

## 附录 A | 创建 GitHub 帐户

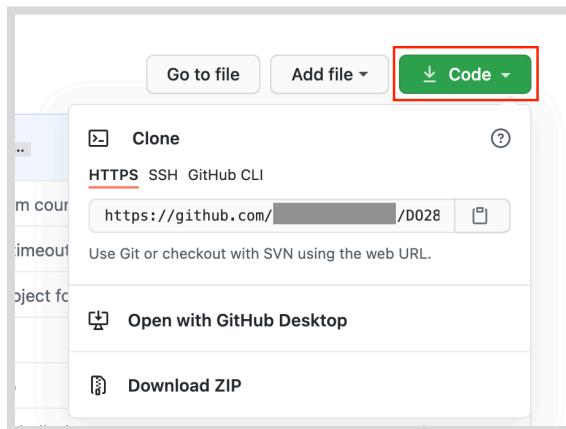


图 A.4: 克隆分叉存储库

## 更新您的分叉

如果存储库的 **RedHatTraining** 副本已更新，则需要更新您的分叉。若要更新您的分叉，请添加存储库本地副本的远程对象，拉取更改，然后将更改推送到您的分叉。

```
[student@workstation D0288-apps]$ git remote add upstream \
https://github.com/RedHatTraining/D0288-apps.git
[student@workstation D0288-apps]$ git pull upstream main
...output omitted...
[student@workstation D0288-apps]$ git push origin main
...output omitted...
```

## 创建 GitHub 个人访问令牌

GitHub 已弃用基于密码的身份验证。也就是说，您必须使用个人访问令牌来运行需要身份验证的 Git 命令，如克隆私有存储库或将更改推送到 GitHub。

如果还没有个人访问令牌，请按照 [创建个人访问令牌](https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token) [https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token] 指南新建令牌。

本课程中的一些练习需要您通过 GitHub 的身份验证。在要求输入密码时，请输入您的个人访问令牌。



### 参考文献

#### 注册新的 GitHub 帐户

<https://help.github.com/en/articles/signing-up-for-a-new-github-account>

#### 创建个人访问令牌

<https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

## 附录 B

# 创建 Quay 帐户

### 目标

介绍如何创建课程中实验所需的 Quay 帐户。

# 创建 Quay 帐户

## 培训目标

学完本节后，您应能够创建 Quay 帐户，并且为课程中需要具有 Quay.io 访问权限的实验使用加密密码。

## 创建 Quay 帐户

您需要一个 Quay 帐户，以便为本课程中的实验创建一个或多个容器镜像存储库。如果您已拥有 Quay 帐户，您可以跳过此附录中列出的创建新帐户的步骤。

若要创建新的 Quay 帐户，请执行以下步骤：

1. 使用 Web 浏览器导航至 <https://quay.io>。
2. 单击右上角搜索栏旁边的 **Sign in**。
3. 在 **Sign in** 页面，使用您的红帽凭据登录。使用红帽凭据进行登录。 

### 警告

2021 年 7 月 31 日后，红帽登录将是唯一可用的 Web 登录选项。

4. 使用红帽 SSO 登录凭据来登录 Quay.io。如果您没有红帽 SSO 帐户，请先使用登录对话框底部的 **Create one now** 链接创建免费的红帽 SSO 帐户。

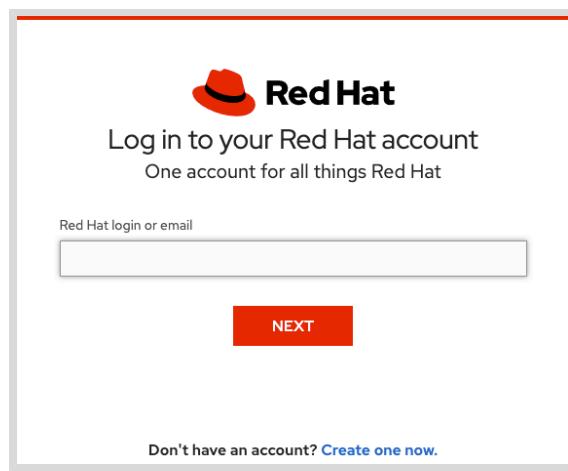


图 B.1: 创建新帐户

## 创建镜像存储库

在实验室中，镜像存储库是从命令行工具创建的。不过，您也可以在 Web 界面中创建它们。

1. 登录 Quay.io 后，您可以通过单击 Repositories 页面中的 **Create New Repository** 来创建新的镜像存储库。

## 附录 B | 创建 Quay 帐户

2. 或者，单击右上角（小钟图标左侧）的加号图标（+），然后单击 **Create New Repository**。

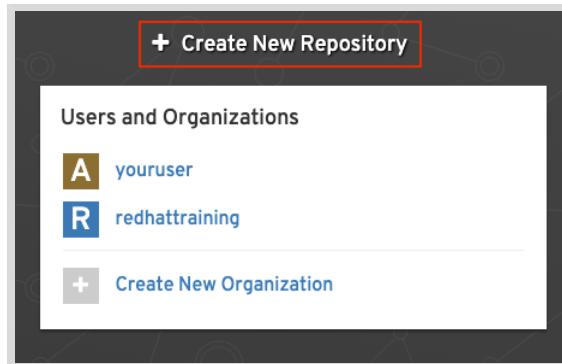


图 B.2: 创建新镜像存储库

## 使存储库变为公共存储库

默认情况下，创建的新存储库是 private 存储库。虽然这足以用于使用登录机密的实验，但一些实验需要 public 存储库。私有存储库的存储库名称旁边有一个小锁图标。

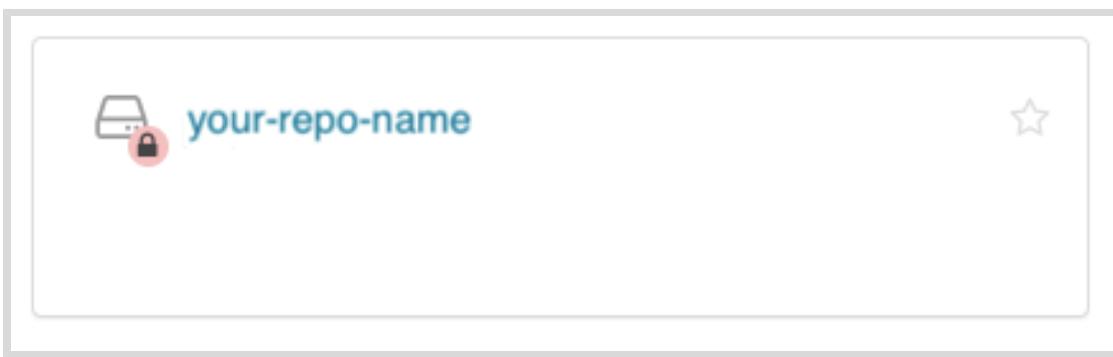


图 B.3: 私有存储库

使用以下步骤，使存储库成为公共存储库。

1. 选择存储库的链接。
2. 单击左侧菜单中的 Settings 图标。

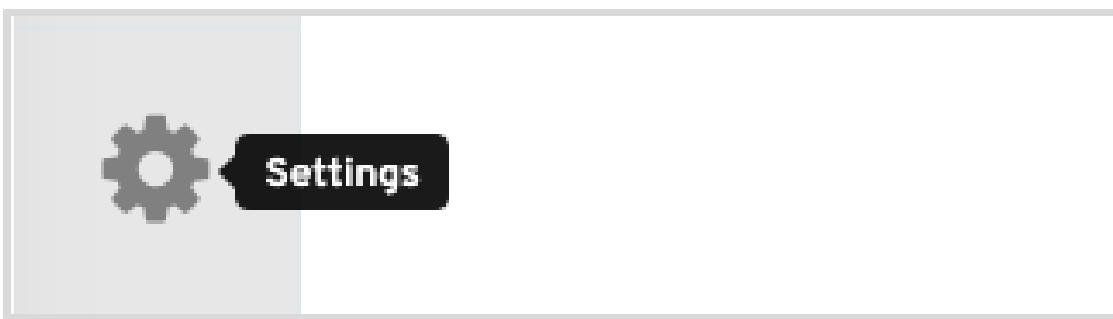


图 B.4: Quay 存储库设置

3. 单击设置页面 **Repository Visibility** 部分中的 **Make Public**。

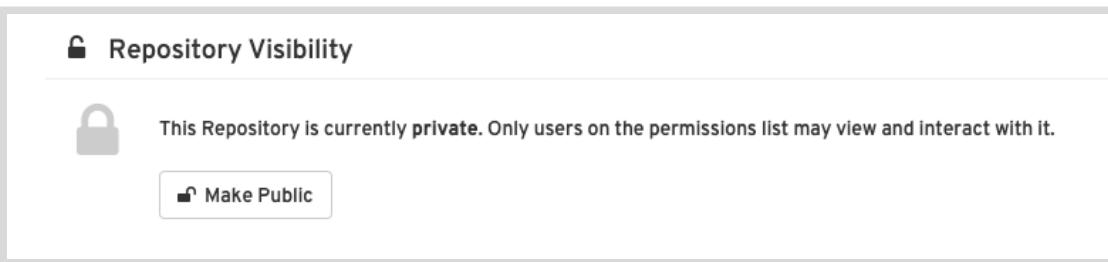


图 B.5: 使存储库变为公共存储库

## 使用 CLI 工具

使用红帽 SSO 创建了帐户后，您需要设置帐户密码以使用 Podman 等 CLI 工具。

1. 单击右上角的名称。
2. 单击 **Account Settings**。
3. 单击 **Change password**。
4. 单击 **Account Settings** 页面顶部的 **Generate Encrypted Password**。
5. 在提示处输入您的帐户密码，并生成加密的等效密码。

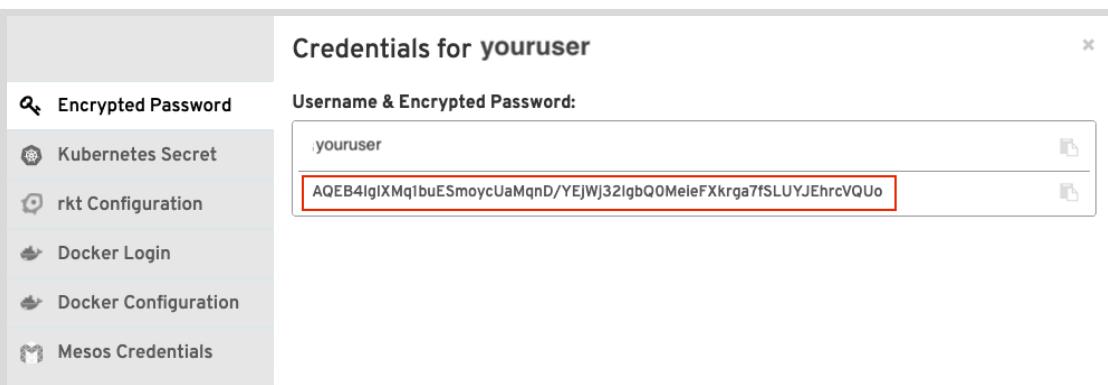


图 B.6: 加密的 Quay 密码

6. 在实验练习中，当系统提示您输入 Quay.io 密码，请使用加密的密码。

```
[student@workstation]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
Password: <use your encrypted password here>
Login Succeeded!
```



### 参考文献

#### Quay.io 入门

<https://docs.quay.io/solution/getting-started.html>

## 附录 C

# 实用的 Git 命令

### 目标

描述本课程中用于实验的实用 Git 命令。

# Git 命令

## 培训目标

学完本节后，您应能够在本课程中重启和重做练习。您还应能够从一个未完成的练习切换到另一个练习，然后再从您离开的位置继续之前的练习。

## 使用 Git 分支

本课程使用 GitHub 托管的 Git 存储库来存储应用课程代码源代码。在本课程开头，您要创建此存储库的派生存储库，它也托管在 GitHub 上。在本课程过程中，您将操作克隆到 **workstation** 虚拟机的本地派生副本。

**origin** 一词指从中克隆本地存储库的远程存储库。

在进行本课程中的练习时，您要为每个练习使用单独的 Git 分支。您对源代码所做的更改都会发生在您为该练习创建的新分支中。不得提交 **main** 分支的任何更改。

下方列出了一系列场景和您可使用的相应 Git 命令。

## 彻底放弃练习并重新开始

若要在完成练习后从头开始重做练习，请执行下列步骤：

1. 在进行练习的过程中，提交并推送本地分支中的所有更改。在练习结束时，通过运行 **finish** 子命令来清理所有资源：

```
[student@workstation ~]$ lab your-exercise finish
```

2. 更改到 **D0288-apps** 存储库的本地克隆，再切换到 **main** 分支：

```
[student@workstation ~]$ cd ~/D0288-apps  
[student@workstation D0288-apps]$ git checkout main
```

3. 删除您的本地分支：

```
[student@workstation D0288-apps]$ git branch -d your-branch
```

4. 删除您的个人 GitHub 帐户上的远程分支：

```
[student@workstation D0288-apps]$ git push origin --delete your-branch
```

5. 使用 **git reset --hard** 丢弃任何待定的更改。

```
[student@workstation D0288-apps]$ git reset --hard
```

6. 如果您有稍后要使用的更改，但需要暂时丢弃，请使用 **git stash -u**：

## 附录 C | 实用的 Git 命令

```
[student@workstation D0288-apps]$ git stash -u
```

7. 使用 **start** 子命令重新开始练习：

```
[student@workstation D0288-apps]$ cd ~  
[student@workstation ~]$ lab your-exercise start
```

## 从未完成的练习切换到其他练习

您可能会遇到这样的情形：已完成了练习中的部分步骤，但希望切换到另一个练习，以后再重新访问当前的练习。

请不要将过多的未完成练习留待以后重新访问。这些练习预留共享资源，您可能会耗尽云提供商和 RHOC 集群的配额。如果您认为可能要过段时间再返回到当前的练习，不妨放弃这个练习，以后再从头重新开始。

如果您选择暂停当前练习并且处理下一个练习，请执行下列步骤：

1. 验证您修改的文件，以查看要添加到提交中的内容。
2. 添加含有您想保存的更改的文件。
3. 提交对本地存储库的任何更改，并将它们推送到您的个人 GitHub 帐户。您可能要记录您停止练习的具体步骤：

```
[student@workstation ~]$ cd ~/D0288-apps  
[student@workstation D0288-apps]$ git status  
...output omitted...  
[student@workstation D0288-apps]$ git add some/file/you/changed  
[student@workstation D0288-apps]$ git commit -m 'Paused at step X.Y'  
[student@workstation D0288-apps]$ git push origin branch-name
```

4. 不要运行原先练习的 **finish** 命令。这非常重要，可以让您使现有的 OpenShift 项目保持不变，以后再继续处理。
5. 通过运行 **start** 子命令来开始下一个练习：

```
[student@workstation ~]$ lab your-exercise start
```

6. 下一个练习将切换到 **main** 分支，可以选择为其更改创建新的分支。这意味着对原先分支中原先练习所做的更改将保持不变。

```
[student@workstation D0288-apps]$ git checkout main
```

7. 之后，完成下一个练习后，如果想要返回到原先的练习，请切换回其分支：

```
[student@workstation ~]$ git checkout branch-name
```

然后，您可以从离开时的步骤继续进行原先的练习。



### 参考文献

#### Git 分支 man page

<https://git-scm.com/docs/git-branch>

#### 什么是 Git 分支?

<https://git-scm.com/book/en/v1/Git-Branching-What-a-Branch-Is>

#### Git 工具 - Stashing

<https://git-scm.com/book/en/v1/Git-Tools-Stashing>