OpenShift Container Platform 4.12

构建应用程序

在 OpenShift Container Platform 中创建和管理应用程序

Last Updated: 2024-07-19

OpenShift Container Platform 4.12 构建应用程序

在 OpenShift Container Platform 中创建和管理应用程序

法律通告

摘要

本文档说明如何通过不同方式创建和管理在 OpenShift Container Platform 上运行的用户置备应用程序实例。这包括处理项目以及使用 Open Service Broker API 置备应用程序。

目录

第 1 章 构建应用程序概述 **4**
1.1. 使用项目 4
1.2. 处理应用程序 4
1.3. 使用 RED HAT MARKETPLACE 4

第 2 章 项目 **6**
2.1. 处理项目 6
2.2. 以其他用户身份创建项目 14
2.3. 配置项目创建 14

第 3 章 创建应用程序 **19**
3.1. 使用 DEVELOPER 视角创建应用程序 19
3.2. 从已安装的 OPERATOR 创建应用程序 26
3.3. 使用 CLI 创建应用程序 27

第 4 章 使用 TOPOLOGY 视图查看应用程序组成 **35**
4.1. 先决条件 35
4.2. 查看应用程序拓扑 35
4.3. 与应用程序和组件交互 36
4.4. 扩展应用程序 POD 以及检查构建和路由 37
4.5. 将组件添加到现有项目 38
4.6. 对应用程序中的多个组件进行分组 39
4.7. 在应用程序中添加服务 40
4.8. 从应用程序中删除服务 41
4.9. 用于 TOPOLOGY 视图的标签和注解 42
4.10. 其他资源 43

第 5 章 导出应用程序 **44**
5.1. 先决条件 44
5.2. 流程 44

第 6 章 将应用程序连接到服务 **45**
6.1. SERVICE BINDING OPERATOR 发行注记 45
6.2. 了解 SERVICE BINDING OPERATOR 54
6.3. 安装 SERVICE BINDING OPERATOR 57
6.4. 服务绑定入门 58
6.5. 在 IBM POWER、IBM Z 和 IBM (R) LINUX 上使用服务绑定 63
6.6. 从服务公开绑定数据 70
6.7. 投射绑定数据 81
6.8. 使用 SERVICE BINDING OPERATOR 绑定工作负载 83
6.9. 使用 DEVELOPER 视角将应用程序连接到服务 94

第 7 章 使用 HELM CHART **101**
7.1. 了解 HELM 101
7.2. 安装 HELM 101
7.3. 配置自定义 HELM CHART 仓库 103
7.4. 使用 HELM 发行版本 113

第 8 章 部署 **115**
8.1. 了解 DEPLOYMENT 和 DEPLOYMENTCONFIG 对象 115
8.2. 管理部署过程 120
8.3. 使用部署策略 127
8.4. 使用基于路由的部署策略 139

第 9 章 配额 **146**
9.1. 项目的资源配额 146
9.2. 跨越多个项目的资源配额 158

第 10 章 将配置映射与应用程序搭配使用 **162**
10.1. 了解配置映射 162
10.2. 用例： 在 POD 中使用配置映射 163

第 11 章 使用 DEVELOPER 视角监控项目和应用程序的指标 **168**
11.1. 先决条件 168
11.2. 监控项目指标数据 168
11.3. 监控应用程序的指标数据 170
11.4. 镜像漏洞分类 171
11.5. 监控应用程序和镜像漏洞指标 172
11.6. 其他资源 173

第 12 章 使用健康检查来监控应用程序的健康状态 **174**
12.1. 了解健康检查 174
12.2. 使用 CLI 配置健康检查 178
12.3. 使用 DEVELOPER 视角监控应用程序的健康状态 181
12.4. 使用 DEVELOPER 视角编辑健康检查 181
12.5. 使用 DEVELOPER 视角监控健康检查失败 182

第 13 章 编辑应用程序 **183**
13.1. 先决条件 183
13.2. 使用 DEVELOPER 视角编辑应用程序的源代码 183
13.3. 使用 DEVELOPER 视角编辑应用程序配置 183

第 14 章 修剪对象以重新声明资源 **186**
14.1. 基本修剪操作 186
14.2. 修剪组 186
14.3. 修剪部署资源 186
14.4. 修剪构建 187
14.5. 自动修剪镜像 188
14.6. 修剪镜像 190
14.7. 硬修剪 REGISTRY 197
14.8. 运行 CRON 任务 199

第 15 章 闲置应用程序 **200**
15.1. 闲置应用程序 200
15.2. 取消闲置应用程序 200

第 16 章 取消应用程序 **202**
16.1. 使用 DEVELOPER 视角删除应用程序 202

第 17 章 使用 RED HAT MARKETPLACE **203**
17.1. RED HAT MARKETPLACE 特性 203

第 1 章 构建应用程序概述

使用 OpenShift Container Platform，您可以使用 Web 控制台或命令行界面 (CLI) 创建、编辑、删除和管理应用程序。

1.1. 使用项目

通过使用项目，您可以以隔离方式组织和管理应用程序。您可以在 OpenShift Container Platform 中管理整个项目生命周期，包括创建、查看和删除项目。

在创建项目后，您可以使用 Developer 视角 授予或撤销对项目的访问权限，并为用户管理集群角色。您还可以在创建用于自动置备新项目的项目模板时编辑项目配置资源。

使用 CLI，您可以通过模拟对 OpenShift Container Platform API 的请求来以不同的用户创建项目。当您请求创建新项目时，OpenShift Container Platform 会使用一个端点来根据自定义模板来置备项目。作为集群管理员，您可以选择阻止经过身份验证的用户组自助置备新项目。

1.2. 处理应用程序

1.2.1. 创建应用程序

要创建应用程序，您必须已创建了一个项目，或者具有适当的角色和权限访问一个项目。您可以通过 web 控制台的开发者视角, 安装的 Operator, 或 the OpenShift CLI (**oc**) 来创建一个应用程序。您可以从 Git、JAR 文件、devfile 或开发人员目录中提供要添加到项目的应用程序。

您还可以使用包含源或二进制代码、镜像和模板的组件，通过 OpenShift CLI (**oc**) 创建应用程序。使用 OpenShift Container Platform Web 控制台，您可以从集群管理员安装的 Operator 创建应用程序。

1.2.2. 维护应用程序

创建应用程序后，您可以使用 Web 控制台监控项目或应用指标。您还可以使用 Web 控制台编辑或删除应用程序。

当应用程序运行时，并非所有应用资源都不会被使用。作为集群管理员，您可以选择闲置这些可扩展资源来减少资源消耗。

1.2.3. 将应用程序连接到服务

应用程序使用后端服务来构建和连接工作负载，这因服务提供商而异。使用 Service Binding Operator 作为开发人员，您可以将工作负载与 Operator 管理的后端服务绑定在一起，而无需手动步骤配置绑定连接。您还可以在 IBM Power、IBM Z 和 IBM® LinuxONE 环境中应用服务绑定。

1.2.4. 部署应用程序

您可以使用 **Deployment** 或 **DeploymentConfig** 对象部署应用程序，并从 Web 控制台管理应用程序。您可以创建部署策略，以帮助减少更改期间或升级到应用程序的停机时间。

您还可以使用 Helm，它是一个软件包管理器，简化了应用程序和服务部署到 OpenShift Container Platform 集群的过程。

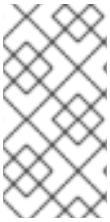
1.3. 使用 RED HAT MARKETPLACE

Red Hat Marketplace 是一个开源云市场，您可以在其中发现并访问在公有云和内部运行的基于容器的环境的认证软件。

第 2 章 项目

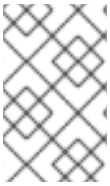
2.1. 处理项目

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



注意

以 **openshift-** 和 **kube-** 开头的项目是 默认项目。这些项目托管作为 pod 运行的主要组件和其他基础架构组件。因此，OpenShift Container Platform 不允许使用 **oc new-project** 命令创建以 **openshift-** 或 **kube-** 开始的项目。集群管理员可以使用 **oc adm new-project** 命令创建这些项目。



注意

您无法将 SCC 分配给在以下某一默认命名空间中创建的 Pod: **default**、**kube-system**、**kube-public**、**openshift-node**、**openshift-infra**、**openshift**。您不能使用这些命名空间用来运行 pod 或服务。

2.1.1. 创建一个项目

您可以使用 OpenShift Container Platform Web 控制台或 OpenShift CLI (**oc**) 在集群中创建项目。

2.1.1.1. 使用 Web 控制台创建项目

您可以使用 OpenShift Container Platform Web 控制台在集群中创建项目。

注意

OpenShift Container Platform 认为以 **openshift-** 和 **kube-** 开头的项目是重要的。因此，OpenShift Container Platform 不允许使用 Web 控制台创建以 **openshift-** 开头的项目。

先决条件

- 在 OpenShift Container Platform 中，确保您有适当的角色和权限来创建项目、应用程序和其他工作负载。

步骤

- 如果使用 **Administrator** 视角：
  a. 浏览至 **Home → Project**。
  b. 点 **Create Project**:
     i. 在 **Create Project** 对话框的 **Name** 项中输入一个唯一的名称，如 **myproject**。
     ii. 可选：为项目添加 **Display Name** 和 **Description** 详情。
     iii. 点 **Create**。
     您的项目的仪表板会显示。

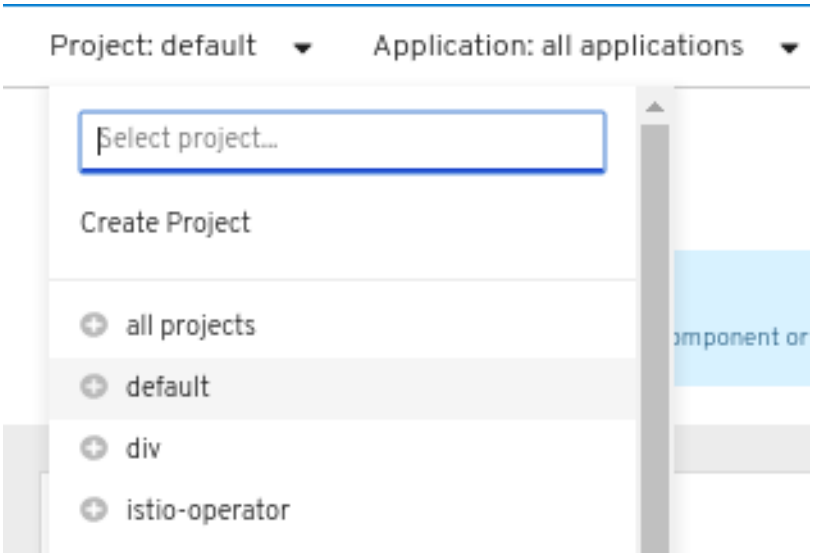
c. 可选： 选择 **Details** 选项卡来查看项目详情。

d. 可选：如果您有针对一个项目的足够权限，您可以使用 **Project Access** 选项卡为项目提供或撤销 admin、edit 和 view 权限。

- 如果使用 **Developer** 视角：

a. 点 **Project** 菜单，再选择 **Create Project**:

图 2.1. Create Project



i. 在 **Create Project** 对话框的 **Name** 项中输入一个唯一的名称，如 **myproject**。

ii. 可选：为项目添加 **Display Name** 和 **Description** 详情。

iii. 点 **Create**。

b. 可选：使用左侧导航面板导航到 **Project** 视图，在仪表板中查看您的项目。

c. 可选：在项目仪表板中，选择 **Details** 选项卡来查看项目详情。

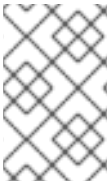
d. 可选：如果您有足够的项目权限，您可以使用项目仪表板的 **Project Access** 选项卡为项目提供或撤销 admin、edit 和 view 权限。

其他资源

- 使用 Web 控制台自定义可用的集群角色

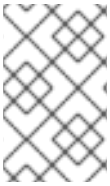
2.1.1.2. 使用 CLI 创建项目

如果集群管理员允许，您可以创建新项目。



注意

OpenShift Container Platform 认为以 **openshift-** 和 **kube-** 开头的项目是重要的。因此，OpenShift Container Platform 不允许使用 **oc new-project** 命令创建以 **openshift-** 或 **kube-** 开始的项目。集群管理员可以使用 **oc adm new-project** 命令创建这些项目。



注意

您无法将 SCC 分配给在以下某一默认命名空间中创建的 Pod: **default**、**kube-system**、**kube-public**、**openshift-node**、**openshift-infra**、**openshift**。您不能使用这些命名空间用来运行 pod 或服务。

流程

- 运行：

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

例如：

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

注意

系统管理员可能会限制允许创建的项目数量。达到限值后，需要删除现有项目才能创建新项目。

2.1.2. 查看项目

您可以使用 OpenShift Container Platform Web 控制台或 OpenShift CLI (**oc**) 查看集群中的项目。

2.1.2.1. 使用 Web 控制台查看项目

您可以使用 OpenShift Container Platform Web 控制台查看您可以访问的项目。

步骤

- 如果使用 **Administrator** 视角：
  a. 在导航菜单中进入到 **Home → Projects**。
  b. 选择要查看的项目。**Overview** 选项卡包含项目的仪表板。
  c. 选择 **Details** 选项卡来查看项目详情。
  d. 选择 **YAML** 选项卡来查看和更新项目资源的 YAML 配置。
  e. 选择 **Workloads** 选项卡来查看项目中的工作负载。
  f. 选择 **RoleBindings** 选项卡来查看和为项目创建角色绑定。
- 如果使用 **Developer** 视角：
  a. 进入到导航菜单中的 **Project** 页面。
  b. 从屏幕顶部的 **Project** 下拉菜单中选择 **All Projects**，以列出集群中的所有项目。
  c. 选择要查看的项目。**Overview** 选项卡包含项目的仪表板。

d. 选择 **Details** 选项卡来查看项目详情。

e. 如果您有足够的项目权限，请选择 **Project access** 选项卡视图并更新项目的特权。

2.1.2.2. 使用 CLI 查看项目

查看项目时，只能看到根据授权策略您有权访问的项目。

流程

1. 要查看项目列表，请运行：

```
$ oc get projects
```

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

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

2.1.3. 使用 Developer 视角为您的项目提供访问权限

您可以使用 **Developer** 视角中的 **Project** 视图来授予或撤销对项目的访问权限。

先决条件

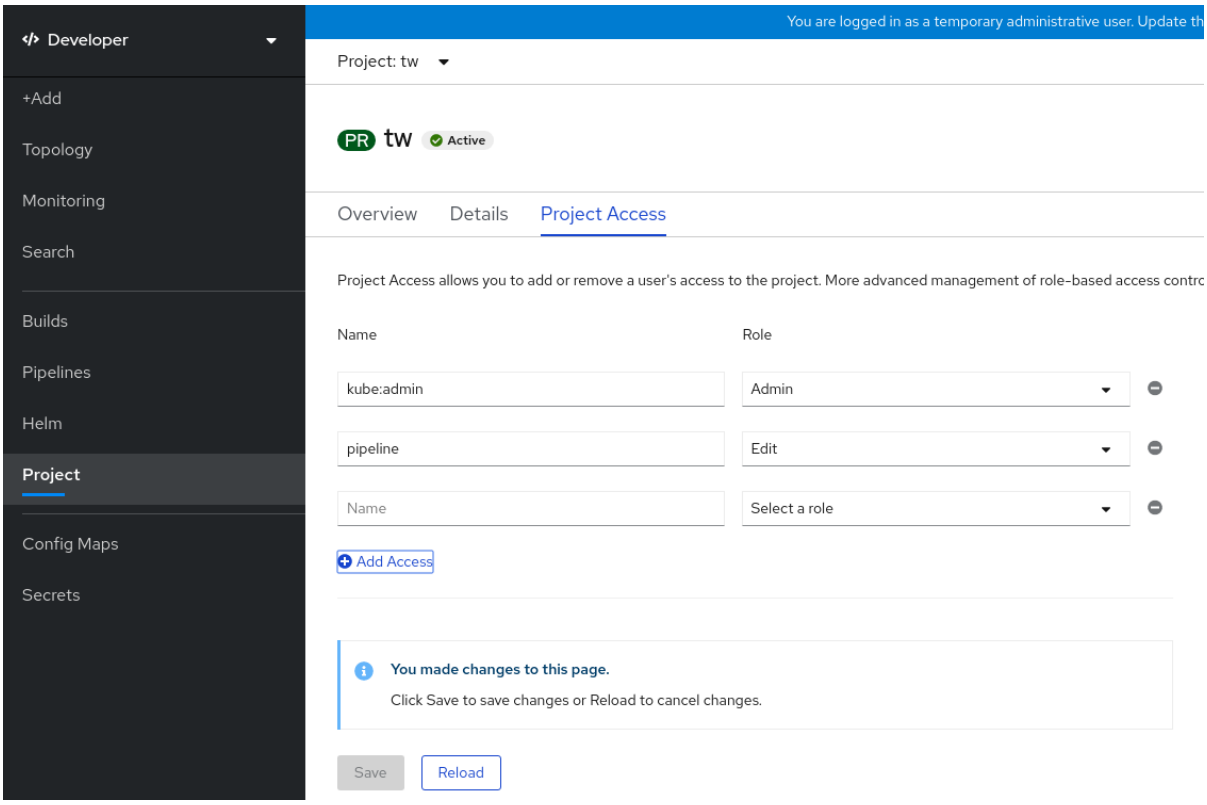
- 您已创建了一个项目。

步骤

将用户添加到项目，并为用户提供 **Admin**、**Edit** 或 **View** 访问权限：

1. 在 **Developer** 视角中，进入到 **Project** 页面。
2. 从 **Project** 菜单中选择您的项目。
3. 选择 **Project Access** 选项卡。
4. 点 **Add access** 为默认权限添加新权限行。

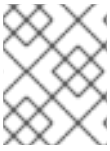
图 2.2. 项目权限



5. 输入用户名，点 **Select a role** 下拉列表，然后选择适当的角色。
6. 点击 **Save** 添加新权限。

您还可以使用：

- **Select a role** 下拉列表修改现有用户的访问权限。
- **Remove Access** 图标以完全删除现有用户对项目的访问权限。



注意

基于角色的高级访问控制是在 **Administrator** 视角的 **Roles** 和 **Roles Binding** 视图中管理的。

2.1.4. 使用 Web 控制台自定义可用的集群角色

在 Web 控制台的 **Developer** 视角中，**Project → Project access** 页面可让项目管理员为项目中的用户授予角色。默认情况下，可以向项目中的用户授予的可用集群角色是 admin、edit 和 view。

作为集群管理员，您可以在集群范围的 **项目访问**页面中定义哪些集群角色可用。您可以通过在 **Console** 配置资源中自定义 **spec.customization.projectAccess.availableClusterRoles** 对象来指定可用的角色。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

步骤

1. 在 **Administrator** 视角中，进入到 **Administration → Cluster settings**。

2. 点 **Configuration** 选项卡。
3. 从 **Configuration resource** 列表中，选择 **Console operator.openshift.io**。
4. 导航到 **YAML** 选项卡以查看和编辑 YAML 代码。
5. 在 **spec** 下的 YAML 代码中，自定义可用于项目访问的集群角色列表。以下示例指定了默认的 **admin**、**edit** 和 **view** 角色：

```
apiVersion: operator.openshift.io/v1
kind: Console
metadata:
  name: cluster
# ...
spec:
  customization:
    projectAccess:
      availableClusterRoles:
      - admin
      - edit
      - view
```

6. 点 **Save** 将更改保存到 **Console** 配置资源。

验证

1. 在 **Developer** 视角中，进入到 **Project** 页面。
2. 从 **Project** 菜单中选择一个项目。
3. 选择 **Project access** 选项卡。
4. 点 **Role** 列中的菜单，并验证可用的角色是否与应用到 **Console** 资源配置的配置匹配。

2.1.5. 添加到项目

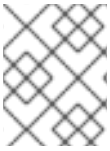
您可以使用 **Developer** 视角中的 **+Add** 页面将项目添加到项目中。

先决条件

- 您已创建了一个项目。

步骤

1. 在 **Developer** 视角中，进入 **+Add** 页面。
2. 从 **Project** 菜单中选择您的项目。
3. 点 **+Add** 页面上的项目，然后按照工作流操作。



注意

您还可以使用 **Add* page to find additional items to add to your project.Click *(在页面顶部的 **Add** 下），并在搜索字段中输入组件名称。

2.1.6. 检查项目状态

您可以使用 OpenShift Container Platform Web 控制台或 OpenShift CLI (**oc**) 查看项目的状态。

2.1.6.1. 使用 Web 控制台检查项目状态

您可以使用 Web 控制台查看项目的状态。

先决条件

- 您已创建了一个项目。

步骤

- 如果使用 **Administrator** 视角：
  a. 浏览至 **Home → Project**。
  b. 从列表中选择一个项目。
  c. 查看 **Overview** 页面中的项目状态。
- 如果使用 **Developer** 视角：
  a. 前往 **Project** 页面。
  b. 从 **Project** 菜单中选择一个项目。
  c. 查看 **Overview** 页面中的项目状态。

2.1.6.2. 使用 CLI 检查项目状态

您可以使用 OpenShift CLI (**oc**)查看项目的状态。

先决条件

- 已安装 OpenShift CLI(**oc**)。
- 您已创建了一个项目。

步骤

1. 切换到项目：

```
$ oc project <project_name> 1
```

1 将 **<project_name>** 替换为项目的名称。

2. 获取项目的高级别概述：

```
$ oc status
```

2.1.7. 删除项目

您可以使用 OpenShift Container Platform Web 控制台或 OpenShift CLI (**oc**) 删除项目。

当您删除项目时，服务器会将项目状态从 **Active** 更新为 **Terminating**。在最终移除项目前，服务器会清除处于 **Terminating** 状态的项目中的所有内容。项目处于 **Terminating** 状态时，您无法将新的内容添加到这个项目中。可以从 CLI 或 Web 控制台删除项目。

2.1.7.1. 使用 Web 控制台删除项目

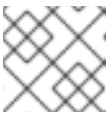
您可以使用 Web 控制台删除项目。

先决条件

- 您已创建了一个项目。
- 有删除项目所需的权限。

步骤

- 如果使用 **Administrator** 视角：
  a. 浏览至 **Home → Project**。
  b. 从列表中选择一个项目。
  c. 点项目的 **Actions** 下拉菜单，再选择 **Delete Project**。



注意

如果您没有删除项目所需的权限，则 **Delete Project** 选项不可用。

  1. 在 **Delete Project?** 窗格中，输入您的项目名称以确认删除。
  2. 点击 **Delete**。

- 如果使用 **Developer** 视角：
  a. 前往 **Project** 页面。
  b. 从 **Project** 菜单中选择您要删除的项目。
  c. 点项目的 **Actions** 下拉菜单，再选择 **Delete Project**。

注意

如果您没有删除项目所需的权限，则 **Delete Project** 选项不可用。

  1. 在 **Delete Project?** 窗格中，输入您的项目名称以确认删除。
  2. 点击 **Delete**。

2.1.7.2. 使用 CLI 删除项目

您可以使用 OpenShift CLI (**oc**) 删除项目。

先决条件

- 已安装 OpenShift CLI(**oc**)。
- 您已创建了一个项目。
- 有删除项目所需的权限。

步骤

1. 删除项目：

```
$ oc delete project <project_name> 1
```

1 将 **<project_name>** 替换为您要删除的项目的名称。

2.2. 以其他用户身份创建项目

通过身份模拟功能，您可以其他用户的身份创建项目。

2.2.1. API 身份模拟（impersonation）

您可以配置对 OpenShift Container Platform API 的请求，使其表现为像是源自于另一用户。如需更多信息，请参阅 Kubernetes 文档中的用户身份模拟。

2.2.2. 在创建项目时模拟用户

您可在创建项目请求时模拟其他用户。由于 **system:authenticated:oauth** 是唯一能够创建项目请求的 bootstrap 组，因此您必须模拟这个组。

流程

- 代表其他用户创建项目请求：

```
$ oc new-project <project> --as=<user> \
    --as-group=system:authenticated --as-group=system:authenticated:oauth
```

2.3. 配置项目创建

在 OpenShift Container Platform 中，*项目*用于对相关对象进行分组和隔离。使用 Web 控制台或 **oc new-project** 命令请求创建新项目时，系统会根据可自定义的模板来使用 OpenShift Container Platform 中的端点置备项目。

作为集群管理员，您可以允许开发人员和服务帐户创建或*自助置备*其自己的项目，并且配置具体的方式。

2.3.1. 关于项目创建

OpenShift Container Platform API 服务器根据项目模板自动置备新的项目，模板通过集群的项目配置资源中的 **projectRequestTemplate** 参数来标识。如果没有定义该参数，API 服务器会创建一个默认模板，该模板将以请求的名称创建项目，并将请求用户分配至该项目的 **admin** 角色。

提交项目请求时，API 会替换模板中的以下参数：

表 2.1. 默认项目模板参数

参数	描述
PROJECT_NAME	项目的名称。必需。
PROJECT_DISPLAYNAME	项目的显示名称。可以为空。
PROJECT_DESCRIPTION	项目的描述。可以为空。
PROJECT_ADMIN_USER	管理用户的用户名。
PROJECT_REQUESTING_USER	请求用户的用户名。

API 访问权限将授予具有 **self-provisioner** 角色和 **self-provisioners** 集群角色绑定的开发人员。默认情况下，所有通过身份验证的开发人员都可获得此角色。

2.3.2. 为新项目修改模板

作为集群管理员，您可以修改默认项目模板，以便使用自定义要求创建新项目。

创建自己的自定义项目模板：

流程

1. 以具有 **cluster-admin** 特权的用户身份登录。
2. 生成默认项目模板：

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

3. 使用文本编辑器，通过添加对象或修改现有对象来修改生成的 **template.yaml** 文件。
4. 项目模板必须创建在 **openshift-config** 命名空间中。加载修改后的模板：

```
$ oc create -f template.yaml -n openshift-config
```

5. 使用 Web 控制台或 CLI 编辑项目配置资源。
   - 使用 Web 控制台：
     i. 导航至 **Administration → Cluster Settings** 页面。
     ii. 单击 **Configuration** 以查看所有配置资源。
     iii. 找到 **Project** 的条目，并点击 **Edit YAML**。
   - 使用 CLI：
     i. 编辑 **project.config.openshift.io/cluster** 资源：

```
$ oc edit project.config.openshift.io/cluster
```

6. 更新 **spec** 部分，使其包含 **projectRequestTemplate** 和 **name** 参数，再设置您上传的项目模板的名称。默认名称为 **project-request**。

   带有自定义项目模板的项目配置资源

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
# ...
spec:
  projectRequestTemplate:
    name: <template_name>
# ...
```

7. 保存更改后，创建一个新项目来验证是否成功应用了您的更改。

2.3.3. 禁用项目自助置备

您可以防止经过身份验证的用户组自助置备新项目。

流程

1. 以具有 **cluster-admin** 特权的用户身份登录。
2. 运行以下命令，以查看 **self-provisioners** 集群角色绑定用法：

```
$ oc describe clusterrolebinding.rbac self-provisioners
```

   输出示例

```
Name:  self-provisioners
Labels:  <none>
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
Role:
  Kind: ClusterRole
  Name: self-provisioner
Subjects:
  Kind Name    Namespace
  ---- ----    ---------
  Group system:authenticated:oauth
```

   检查 **self-provisioners** 部分中的主题。

3. 从 **system:authenticated:oauth** 组中移除 **self-provisioner** 集群角色。

   - 如果 **self-provisioners** 集群角色绑定仅将 **self-provisioner** 角色绑定至 **system:authenticated:oauth** 组，请运行以下命令：

```
$ oc patch clusterrolebinding.rbac self-provisioners -p '{"subjects": null}'
```

   - 如果 **self-provisioners** 集群角色将 **self-provisioner** 角色绑定到 **system:authenticated:oauth** 组以外的多个用户、组或服务帐户，请运行以下命令：

```
$ oc adm policy \
    remove-cluster-role-from-group self-provisioner \
    system:authenticated:oauth
```

4. 编辑 **self-provisioners** 集群角色绑定，以防止自动更新角色。自动更新会使集群角色重置为默认状态。

   - 使用 CLI 更新角色绑定：

     i. 运行以下命令：

     ```
     $ oc edit clusterrolebinding.rbac self-provisioners
     ```

     ii. 在显示的角色绑定中，将 **rbac.authorization.kubernetes.io/autoupdate** 参数值设置为 **false**，如下例所示：

     ```
     apiVersion: authorization.openshift.io/v1
     kind: ClusterRoleBinding
     metadata:
       annotations:
         rbac.authorization.kubernetes.io/autoupdate: "false"
     # ...
     ```

   - 使用单个命令更新角色绑定：

     ```
     $ oc patch clusterrolebinding.rbac self-provisioners -p '{ "metadata": { "annotations": {
     "rbac.authorization.kubernetes.io/autoupdate": "false" } } }'
     ```

5. 以通过身份验证的用户身份登陆，验证是否无法再自助置备项目：

   ```
   $ oc new-project test
   ```

   输出示例

   ```
   Error from server (Forbidden): You may not request a new project via this API.
   ```

   您可以对此项目请求消息进行自定义，以提供特定于您的组织的更多有用说明。

2.3.4. 自定义项目请求消息

当无法自助置备项目的开发人员或服务帐户使用 Web 控制台或 CLI 提出项目创建请求时，默认返回以下错误消息：

```
You may not request a new project via this API.
```

集群管理员可以自定义此消息。您可以对这个消息进行自定义，以提供特定于您的组织的关于如何请求新项目的信息。例如：

- To request a project, contact your system administrator at **projectname@example.com**.
- To request a new project, fill out the project request form located at **https://internal.example.com/openshift-project-request**.

自定义项目请求消息：

流程

1. 使用 Web 控制台或 CLI 编辑项目配置资源。
   - 使用 Web 控制台：
     i. 导航至 **Administration → Cluster Settings** 页面。
     ii. 单击 **Configuration** 以查看所有配置资源。
     iii. 找到 **Project** 的条目，并点击 **Edit YAML**。
   - 使用 CLI：
     i. 以具有 **cluster-admin** 特权的用户身份登录。
     ii. 编辑 **project.config.openshift.io/cluster** 资源：

     ```
     $ oc edit project.config.openshift.io/cluster
     ```

2. 更新 **spec** 部分，使其包含 **projectRequestMessage** 参数，并将值设为您的自定义消息：

   带有自定义项目请求消息的项目配置资源

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   # ...
   spec:
     projectRequestMessage: <message_string>
   # ...
   ```

   例如：

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   # ...
   spec:
     projectRequestMessage: To request a project, contact your system administrator at
   projectname@example.com.
   # ...
   ```

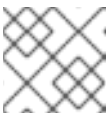
3. 保存更改后，请尝试用无法自助置备项目的开发人员或服务帐户创建一个新项目，以验证是否成功应用了您的更改。

第 3 章 创建应用程序

3.1. 使用 DEVELOPER 视角创建应用程序

Web 控制台中的 **Developer** 视角为您提供了下列选项，以便您从 **+Add** 视图中创建应用程序和相关服务，并将它们部署到 OpenShift Container Platform：

- **入门资源**：使用这些资源帮助您开始使用开发人员控制台。您可以选择使用 **Options** 菜单来隐藏标头。
  - **使用示例创建应用程序**：使用现有代码示例开始在 OpenShift Container Platform 上创建应用程序。
  - **使用引导式练习文档构建**：遵循指导文档构建应用并熟悉关键概念和术语。
  - **探索开发人员新功能**：探索 **Developer** 视角中的新功能和资源。
- **Developer Catalog**：浏览 Developer Catalog 以选择所需的应用、服务或源到镜像构建器，然后将它添加到项目中。
  - **所有服务**：浏览目录以在 OpenShift Container Platform 中发现服务。
  - **Database**：选择所需的数据库服务并将其添加到应用程序中。
  - **Operator Backed**：选择和部署所需的 Operator 管理服务。
  - **Helm Chart**：选择所需的 Helm Chart 来简化应用程序和服务部署。
  - **Devfile**: 从 **Devfile registry** 中选择一个 devfile 来声明性地定义开发环境。
  - **Event Source**：选择一个事件源，从特定系统中注册对一类事件的兴趣。



注意

如果安装了 RHOAS Operator，也可使用 Managed services 选项。

- **Git 存储库**：使用 **From Git**、**From Devfile** 或 **From Dockerfile** 选项分别从您的 Git 存储库中导入一个存在的 codebase、Devfile 或 Dockerfile，以在 OpenShift Container Platform 上构建和部署一个应用程序。
- **Container Image**：使用镜像流或 registry 中的现有镜像，将其部署到 OpenShift Container Platform 中。
- **Pipelines**：使用 Tekton 管道为 OpenShift Container Platform 上的软件交付过程创建 CI/CD 管道。
- **Serverless**：探索 **Serverless** 选项，在 OpenShift Container Platform 中创建、构建和部署无状态和无服务器应用程序。
  - **Channel**：创建一个 Knative 频道以创建一个事件转发，使用内存的持久性层以及可靠的实现
- **示例**：探索可用的示例应用程序，以快速创建、构建和部署应用程序。
- **快速入门** ：了解快速启动选项，使用详细的说明和任务创建、导入并运行应用程序。

- **From Local Machine**：通过 **From Local Machine** 标题导入或上传在您的本地机器中的文件用于更方便地构建并部署应用程序。
  - **导入 YAML**：上传 YAML 文件，以创建并定义用于构建和部署应用程序的资源。
  - **上传 JAR 文件**：上传 JAR 文件以构建和部署 Java 应用。
- **共享我的项目** ：使用此选项向项目添加或删除用户，并提供可访问性选项。
- **Helm Chart 仓库** ： 使用此选项在命名空间中添加 Helm Chart 仓库。
- **对资源重新排序**：使用这些资源重新排序添加到导航窗格中的固定资源。当您将鼠标悬停在导航窗格中时，固定资源左侧会显示拖放图标。拖放的资源只能在它所在的部分中丢弃。

请注意，某些选项（如 **Pipelines**、**Event Source** 和 **Import Virtual Machines**）仅在 OpenShift Pipelines Operator、OpenShift Serverless Operator 和 OpenShift Virtualization Operator 分别安装时才会显示。

3.1.1. 先决条件

要使用 **Developer** 视角创建应用程序，请确认以下几项：

- 已登陆到 web 控制台。
- 您已创建了一个项目，或者具有适当的角色和权限访问项目，以便在 OpenShift Container Platform 中创建应用程序和其他工作负载。

除以上所要求外，要创建无服务器应用程序，请确保：

- 已安装 OpenShift Serverless Operator。
- 您已在 **knative-serving** 命名空间中创建了 **KnativeServing** 资源。

3.1.2. 创建示例应用程序

您可以使用 **Developer** 视角的 **+Add** 流中的示例应用程序来快速创建、构建和部署应用程序。

先决条件

- 已登陆到 OpenShift Container Platform web 控制台，且处于 **Developer** 视角。

步骤

1. 在 **+Add** 视图中，点 **Samples** 标题查看 **Samples** 页面。
2. 在 **Samples** 页面中，选择一个可用的示例应用程序来查看 **Create Sample Application** 表单。
3. 在 **Create Sample Application Form** 中：
   - 在 **Name** 字段中，部署名称会被默认显示。您可以根据需要修改此名称。
   - 在 **Builder Image Version** 中，会默认选择一个构建器镜像。您可以使用 **Builder Image Version** 下拉列表修改此镜像版本。
   - 默认添加 Git 存储库 URL 示例。

4. 点 **Create** 创建示例应用程序。示例应用程序的构建状态显示在 **Topology** 视图中。创建示例应用程序后，您可以看到添加到应用程序的部署。

3.1.3. 使用快速入门创建应用程序

Quick Starts 页面演示了如何在 OpenShift Container Platform 上创建、导入和运行应用程序，以及逐步说明和任务。

先决条件

- 已登陆到 OpenShift Container Platform web 控制台，且处于 **Developer** 视角。

步骤

1. 在 **+Add** 视图中，点 **Getting Started resources→ Build with guided documentation → View all quick start** 链接来查看 **Quick Starts** 页面。
2. 在 **Quick Starts** 页面中，点您要使用的快速启动的标题。
3. 点 **Start** 开始快速启动。
4. 执行显示的步骤。

3.1.4. 从 Git 导入代码库来创建应用程序

您可以在 **Developer** 视角中，使用 GitHub 中的现有代码库，在 OpenShift Container Platform 中创建、构建和部署应用程序。

以下流程逐步指导您在 **Developer** 视角中使用 **Import from Git** 选项来创建应用程序。

流程

1. 在 **+Add** 视图中，点 **Git Repository** 标题中的 **From Git** 来查看 **Import from git** 表单。
2. 在 **Git** 部分中，输入您要用来创建应用程序的代码库的 Git 存储库 URL。例如，输入此示例 Node.js 应用程序的 URL **https://github.com/sclorg/nodejs-ex**。这个 URL 随后会被验证。
3. 可选：点 **Show Advanced Git Options** 来添加详情，例如：
   - **git Reference**，指向特定的分支、标签或提交中的代码，以用于构建应用程序。
   - **Context Dir**，指定要用来构建应用程序的应用程序源代码的子目录。
   - **Source Secret**，创建一个具有用来从私有存储库拉取源代码的凭证的 **Secret Name**。
4. 可选：您可以通过 Git 存储库导入 devfile、Dockerfile 或构建器镜像来进一步自定义部署。
   - 如果您的 Git 存储库包含 devfile、Dockerfile 或构建器镜像，它会被自动检测并填充到相应的路径字段中。如果同一存储库中检测到 devfile、Dockerfile 和构建器镜像，则默认选择 devfile。
   - 若要编辑文件导入类型并选择不同的策略，请单击 **Edit import strategy** 选项。
   - 如果检测到多个 devfile、Dockerfile 或构建器镜像，以导入特定的 devfile、Dockerfile 或构建器镜像，请指定与上下文目录相关的相应路径。

5. 在验证 Git URL 后，会选择建议的构建器镜像并标记为星号。如果构建器镜像没有自动探测到，请选择一个构建器镜像。对于 **https://github.com/sclorg/nodejs-ex** Git URL，默认选择了 Node.js 构建器镜像。

   a. 可选： 使用 **Builder Image Version** 下拉菜单指定版本。

   b. 可选： 使用 **Edit import** 策略来选择不同的策略。

   c. 可选： 对于 Node.js 构建器镜像，请使用 **Run command** 字段覆盖运行应用程序的命令。

6. 在 **General** 部分中：

   a. 在 **Application** 字段中输入应用程序组别的唯一名称，例如 **myapp**。确保应用程序名称在命名空间中具有唯一性。

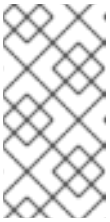
   b. 系统会基于 Git 存储库的 URL 自动填充 **Name** 字段，以标识为此应用程序创建的资源（如果没有存在的应用程序）。如果已有应用程序，可以选择将组件部署到现有应用程序中，创建一个新应用程序，或保持该组件没有被分配。

注意

资源名称必须在命名空间中具有唯一性。如果遇到错误，请修改资源名称。

7. 在 **Resources** 部分，选择：

   - **Deployment**，以纯 Kubernetes 风格方式创建应用程序。
   - **Deployment Config**，创建 OpenShift Container Platform 风格的应用程序。
   - **Serverless Deployment**，创建 Knative 服务。



注意

您可以通过浏览 User preference 页面并点 **Applications → Resource type** 字段来设置 Import 的默认资源首选项。只有集群中安装了 OpenShift Serverless Operator 时，**Import from Git** 表单中才会显示 **Serverless Deployment** 选项。如需了解更多详细信息，请参阅 OpenShift Serverless 文档。

8. 在 **Pipelines** 部分，选择 **Add Pipeline**，然后点 **Show Pipeline Visualization** 来查看应用程序的管道。选择了默认管道，但您可以从应用程序的可用管道列表中选择所需的管道。

9. 可选： 在 **Advanced Options** 部分中，默认选择 **Target port** 和 **Create a route to the application**，以便您可以使用公开的 URL 访问应用程序。
   如果您的应用程序没有在默认公共端口（80）上公开其数据，请清除复选框，并设置您想要公开的目标端口号。

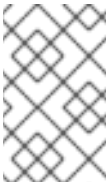
10. 可选：可以使用以下高级选项进一步自定义应用程序：

    路由

    点击 **Routing** 链接，您可以执行以下操作：

    - 自定义路由的主机名。
    - 指定路由器监控的路径。
    - 从下拉列表中选择流量的目标端口。

- 选中 **Secure Route** 复选框来保护您的路由。从相应的下拉列表中，选择所需的 TLS 终止类型，并设置非安全流量的策略。



注意

对于无服务器应用程序，Knative 服务管理上述所有路由选项。但在需要时，您可以自定义流量的目标端口。如果不指定目标端口，则使用默认端口 **8080**。

域映射

如果要创建 **Serverless Deployment**，您可以在创建过程中添加自定义域映射到 Knative 服务。

- 在 **Advanced options** 部分中，点 **Show advanced Routing options**。
  - 如果要映射到该服务的域映射 CR 已存在，您可以从 **Domain mapping** 下拉菜单中选择它。
  - 如果要创建新域映射 CR，在框中输入域名，然后选择 **Create** 选项。例如，如果您在 **example.com 中** 键入，则 **Create** 选项为 **Create "example.com"**。

健康检查

点击 **Health Checks** 链接为您的应用程序添加就绪（Readiness）、存活（Liveness）和启动（Startup）探测。所有探测都预先填充默认数据; 您可以使用默认数据添加探测或根据需要进行自定义。
自定义健康探测：

- 点 **Add Readiness Probe**，在需要的情况下修改参数来检查容器是否准备好处理请求，然后选择要添加的探测。
- 点 **Add Liveness Probe**，在需要的情况下修改参数来检查容器是否仍在运行，选择要添加的探测。
- 点 **Add Startup Probe**，在需要的情况下修改参数来检查容器内的应用程序是否已启动，选择要添加的探测。
  对于每个探测，您可以从下拉列表中指定请求类型 - **HTTP GET**、**Container Command** 或 **TCP Socket**。表单会根据所选请求类型进行更改。然后您可以修改其它参数的默认值，如探测成功和失败的阈值、在容器启动后执行第一个探测前的秒数、探测的频率以及超时值。

构建配置和部署

点 **Build Configuration** 和 **Deployment Configuration** 链接来查看对应的配置选项。一些选项会被默认选中；您可以通过添加必要的触发器和环境变量来进一步自定义。
对于无服务器应用程序，**Deployment** 选项不会显示，因为 Knative 配置资源为您的部署维护所需的状态，而不是由 **DeploymentConfig** 资源来维护。

扩展

点击 **Scaling** 链接，以定义您要初始部署的应用程序的 pod 数或实例数。
如果要创建无服务器部署，也可以配置以下设置：

- **最小 Pod** 决定 Knative 服务在任意给定时间运行的 pod 数量较低限制。这也被称为 **minScale** 设置。

- **最大 Pod** 决定了 Knative 服务可在任意给定时间运行的 pod 数量上限。这也被称为 **maxScale** 设置。
- **并发目标** 决定了给定时间每个应用程序实例所需的并发请求数。
- **并发限制** 决定了给定时间允许每个应用程序的并发请求数的限值。
- **并发利用率** 决定了在 Knative 扩展额外 pod 前必须满足并发请求限制的百分比，以处理额外的流量。
- **自动扩展窗口**定义了平均时间窗口，以便在自动扩展器不处于 panic 模式时提供缩放决策的输入。如果在此窗口中没有收到任何请求，服务将缩减为零。autoscale 窗口的默认持续时间为 **60s**。这也被称为 stable 窗口。

资源限值

点击 **Resource Limit** 链接，设置容器在运行时保证或允许使用的 **CPU** 和 **Memory** 资源的数量。

标签

点击 **Labels** 链接，为您的应用程序添加自定义标签。

11. 单击 **Create** 以创建应用程序，会显示一个成功通知。您可以在 **Topology** 视图中查看应用程序的构建状态。

3.1.5. 通过上传 JAR 文件来部署 Java 应用程序

您可以使用 Web 控制台 **Developer** 视角使用以下选项上传 JAR 文件：

- 导航到 **Developer 视角** 的 **+Add** 视图，再单击 **From Local Machine**标题中的 **Upload JAR 文件**。浏览并选择 JAR 文件，或者拖动 JAR 文件以部署应用程序。
- 进入到 **Topology** 视图并使用 **Upload JAR 文件** 选项，或者拖动 JAR 文件以部署应用程序。
- 使用 **Topology** 视图中的 in-context 菜单，然后使用 **Upload JAR 文件**选项上传 JAR 文件以部署应用程序。

先决条件

- Cluster Samples Operator 必须由集群管理员安装。
- 您可以访问 OpenShift Container Platform Web 控制台，且处于 **Developer** 视角。

流程

1. 在 **Topology** 视图中，右键点任何位置来查看 **Add to Project** 菜单。
2. 将鼠标悬停在 **Add to Project** 菜单上，以查看菜单选项，然后选择 **Upload JAR 文件**选项以查看 **Upload JAR 文件**表单。或者，您可以将 JAR 文件拖到 **Topology** 视图中。
3. 在 **JAR 文件**字段中，浏览本地计算机上所需的 JAR 文件并上传该文件。或者，您可以将 JAR 文件拖到字段。如果将不兼容的文件类型拖到 **Topology** 视图中，则右上角会显示一个警报。如果上传表单的字段中丢弃了不兼容的文件类型，则会显示字段错误。
4. 默认选择运行时图标和构建器镜像。如果构建器镜像没有自动探测到，请选择一个构建器镜像。如果需要，您可以使用 **Builder Image Version** 下拉列表来更改版本。

5. 可选：在 **Application Name** 字段中输入应用程序的唯一名称，用于资源标记。
6. 在 **Name** 字段中输入相关资源的唯一组件名称。
7. 可选： 使用 **Advanced options → Resource type** 下拉列表，从默认资源类型列表中选择不同的资源类型。
8. 在 **Advanced options** 菜单中，点 **Create a Route to the Application**配置您部署的应用程序的公共 URL。
9. 点 **Create** 以部署应用。显示有提示通知，以通知您 JAR 文件正在上传。相关通知还包括用于查看构建日志的链接。

注意

如果您在构建运行时尝试关闭浏览器标签页，则会显示 Web 警报。

上传 JAR 文件并部署应用后，您可以在 **Topology** 视图中查看应用程序。

3.1.6. 使用 Devfile registry 访问 devfile

您可以使用 **Developer** 视角的 **+Add** 流中的 devfile 创建应用程序。 **+Add** 流提供与 devfile 社区 registry 的完整集成。devfile 是一个可移植的 YAML 文件，它描述了您的开发环境，而无需从头开始进行配置。使用 **Devfile registry**，您可以使用预配置的 devfile 创建应用程序。

步骤

1. 进入 **Developer 视角 → +Add → Developer Catalog → All Services**。此时会显示 **Developer Catalog** 中所有可用服务的列表。
2. 在 **Type** 下，点 **Devfiles** 浏览支持特定语言或框架的 devfile。另外，您可以使用 keyword 过滤器使用其名称、标签或描述搜索特定 devfile。
3. 点击您要用来创建应用程序的 devfile。devfile 标题显示 devfile 的详情，包括 devfile 的名称、描述、供应商和 devfile 文档。
4. 点 **Create** 创建一个应用程序，并在 **Topology** 视图中查看应用程序。

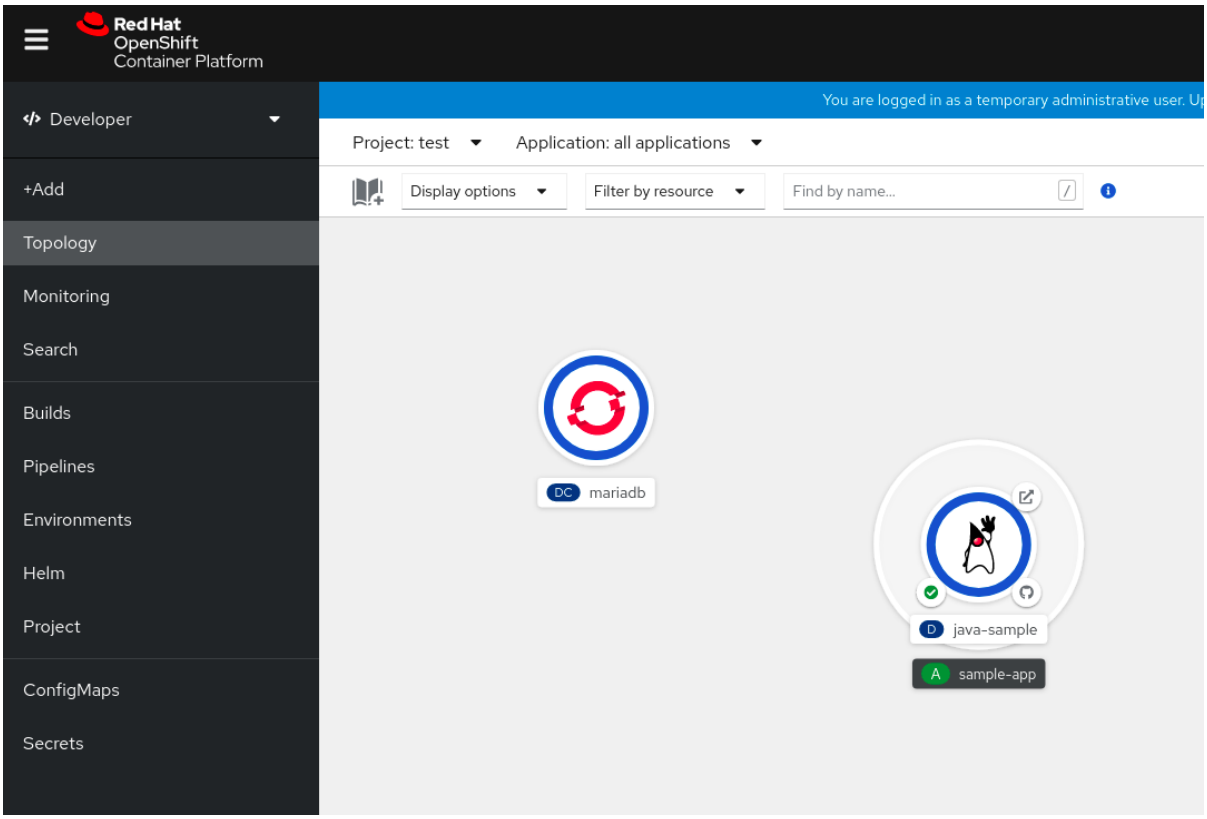
3.1.7. 使用 Developer Catalog 将服务或组件添加到应用程序中

您可以使用 Developer Catalog 根据 Operator 支持的服务（如数据库、构建器镜像和 Helm Charts）部署应用程序和服务。Developer Catalog 包含您可以添加到项目的应用程序组件、服务、事件源或 Source-to-image 构建器的集合。集群管理员可以自定义目录中提供的内容。

流程

1. 在 **Developer** 视角中，导航到 **+Add** 视图，从 **Developer Catalog** 标题中点击 **All Services** 来查看 **Developer Catalog** 中的所有可用服务。
2. 在 **All Services** 下，选择服务类型或您需要添加到项目的组件。在本例中，选择 **Databases** 以列出所有数据库服务，然后点击 **MariaDB** 查看该服务的详情。
3. 点 **Instantiate Template** 查看带有 **MariaDB** 服务详情的自动填充的模板，然后点 **Create** 在 **Topology** 视图中创建并查看 MariaDB 服务的信息。

图 3.1. Topology 中的 MariaDB



3.1.8. 其他资源

- 如需有关 OpenShift Serverless 的 Knative 路由设置的更多信息，请参阅 路由。
- 如需有关 OpenShift Serverless 的域映射设置的更多信息，请参阅为 Knative 服务配置自定义域。
- 如需有关 OpenShift Serverless 的 Knative 自动扩展设置的更多信息，请参阅 自动扩展。
- 有关向项目添加新用户的更多信息，请参阅使用项目。
- 有关创建 Helm Chart 仓库的更多信息，请参阅创建 Helm Chart 仓库。

3.2. 从已安装的 OPERATOR 创建应用程序

Operators 是打包、部署和管理 Kubernetes 应用程序的方法。您可以使用集群管理员安装的 Operator 在 OpenShift Container Platform 上创建应用程序。

本指南为开发人员介绍如何使用 OpenShift Container Platform Web 控制台从已安装的 Operator 中创建应用程序。

其他资源

- 如需有关 Operator 如何工作以及如何将 Operator Lifecycle Manager 集成到 OpenShift Container Platform 中的更多信息，请参阅 Operator 指南。

3.2.1. 使用 Operator 创建 etcd 集群

本流程介绍了如何通过由 Operator Lifecycle Manager (OLM) 管理的 etcd Operator 来新建一个 etcd 集群。

先决条件

- 访问 OpenShift Container Platform 4.12 集群。
- 管理员已在集群范围内安装了 etcd Operator。

流程

1. 针对此流程在 OpenShift Container Platform Web 控制台中新建一个项目。这个示例使用名为 **my-etcd** 的项目。
2. 导航至 **Operators → Installed Operators** 页面。由集群管理员安装到集群且可供使用的 Operator 将以集群服务版本（CSV）列表形式显示在此处。CSV 用于启动和管理由 Operator 提供的软件。

   提示

   使用以下命令从 CLI 获得该列表：

   ```
   $ oc get csv
   ```

3. 在 **Installed Operators** 页面中，点 etcd Operator 查看更多详情和可用操作。
   正如 **Provided API** 下所示，该 Operator 提供了三类新资源，包括一种用于 **etcd Cluster** 的资源（**EtcdCluster** 资源）。这些对象的工作方式与内置的原生 Kubernetes 对象（如 **Deployment** 或 **ReplicaSet**）相似，但包含特定于管理 etcd 的逻辑。
4. 新建 etcd 集群：
   a. 在 **etcd Cluster** API 框中，点 **Create instance**。
   b. 在下一页上，您可对 **EtcdCluster** 对象的最小起始模板进行任何修改，比如集群大小。现在，点击 **Create** 即可完成。点击后即可触发 Operator 启动 pod、服务和新 etcd 集群的其他组件。
5. 点 **example** etcd 集群，然后点 **Resources** 选项卡，您可以看到项目现在包含很多由 Operator 自动创建和配置的资源。
   验证已创建了支持您从项目中的其他 pod 访问数据库的 Kubernetes 服务。
6. 给定项目中具有 **edit** 角色的所有用户均可创建、管理和删除应用程序实例（本例中为 etcd 集群），这些实例由已在项目中创建的 Operator 以自助方式管理，就像云服务一样。如果要赋予其他用户这一权利，项目管理员可使用以下命令添加角色：

   ```
   $ oc policy add-role-to-user edit <user> -n <target_project>
   ```

现在您有了一个 etcd 集群，当 pod 运行不畅，或在集群中的节点之间迁移时，该集群将对故障做出反应并重新平衡数据。最重要的是，具有适当访问权限的集群管理员或开发人员现在可轻松将该数据库用于其应用程序。

3.3. 使用 CLI 创建应用程序

您可以使用 OpenShift Container Platform CLI，从包含源代码或二进制代码、镜像和模板的组件创建 OpenShift Container Platform 应用程序。

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

3.3.1. 从源代码创建应用程序

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

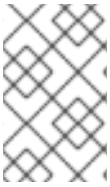
new-app 命令会创建一个构建配置，其本身会从您的源代码中创建一个新的应用程序镜像。**new-app** 命令通常还会创建一个 **Deployment** 对象来部署新镜像，以及为运行您的镜像的部署提供负载均衡访问的服务。

OpenShift Container Platform 会自动检测要使用管道、源或 docker 构建策略，如果进行源构建，则还检测适当的语言构建器镜像。

3.3.1.1. Local

从本地目录中的 Git 存储库创建应用程序：

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



注意

如果使用本地 Git 存储库，该存储库必须具有一个名为 **origin** 的远程源，指向可由 OpenShift Container Platform 集群访问的 URL。如果没有可识别的远程源，运行 **new-app** 命令将创建一个二进制构建。

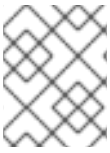
3.3.1.2. 远程

从远程 Git 存储库创建新应用程序：

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

从私有远程 Git 存储库创建应用程序：

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



注意

如果使用私有远程 Git 存储库，您可以使用 **--source-secret** 标志指定一个现有源克隆 secret，此 secret 将注入到构建配置中以访问存储库。

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

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

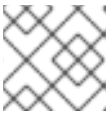
另外，在指定远程 URL 时，您可以通过在 URL 末尾附加 **#<branch_name>** 来指定要使用的 Git 分支：

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

3.3.1.3. 构建策略检测

OpenShift Container Platform 通过检测某些文件自动决定要使用的构建策略：

- 在创建新应用程序时，如果源存储库的根目录或指定上下文目录中存在 Jenkinsfile 文件，则 OpenShift Container Platform 会生成管道构建策略。



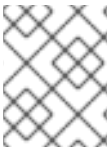
注意

pipeline 构建策略已弃用；请考虑使用 Red Hat OpenShift Pipelines。

- 在创建新应用程序时，如果源存储库的根目录或指定上下文目录中存在 Dockerfile，则 OpenShift Container Platform 会生成 docker 构建策略。
- 如果没有检测到 Jenkins 文件或 Dockerfile，OpenShift Container Platform 会生成源构建策略。

通过将 **--strategy** 标志设置为 **docker**、**pipeline** 或 **source** 来覆盖自动检测到的构建策略。

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



注意

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

3.3.1.4. 语言检测

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

表 3.1. new-app检测到的语言

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

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

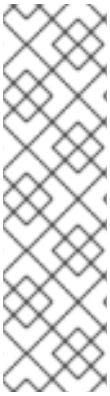
您可以通过指定镜像（镜像流或容器规格）和存储库（以 ~ 作为分隔符），来覆盖构建器用于特定源存储库的镜像。请注意，如果进行这一操作，就不会执行构建策略检测和语言检测。

例如，使用 **myproject/my-ruby** 镜像流以及位于远程存储库中的源：

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

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

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



注意

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

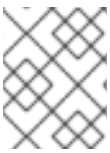
调用 **-i <image> <repository>** 需要 **new-app** 尝试克隆 **repository**，从而判断其工件类型；如果 Git 不可用，此操作会失败。

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

3.3.2. 从镜像创建应用程序

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

new-app 命令尝试确定传递给它的参数中指定的镜像类型。但是，您可以使用 **--docker-image** 参数明确告知 **new-app** 镜像是一个容器镜像，或使用 **-i|--image-stream** 参数明确告知镜像是一个镜像流。



注意

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

3.3.2.1. Docker Hub MySQL 镜像

从 Dockerhub MySQL 镜像创建应用程序，例如：

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

3.3.2.2. 私有 registry 中的镜像

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

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

3.3.2.3. 现有镜像流和可选镜像流标签

从现有镜像流和可选镜像流标签创建应用程序：

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

3.3.3. 从模板创建应用程序

您可以使用之前存储的模板或模板文件创建应用程序，方法是将模板名称指定为参数。例如，您可以存储一个示例应用程序模板，并使用它来创建应用程序。

将应用程序模板上传到当前项目的模板库。以下示例从名为 **example/sample-app/application-template-stibuild.json** 的文件上传一个应用程序模板：

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

然后，通过引用应用程序模板来创建新应用程序。在本例中，模板名称为 **ruby-helloworld-sample**:

```
$ oc new-app ruby-helloworld-sample
```

要通过引用本地文件系统中的模板文件创建新应用程序，而无需首先将其保存到 OpenShift Container Platform 中，使用 **-f|--file** 参数。例如：

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

3.3.3.1. 模板参数

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

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

您可以将参数保存到文件中，然后在实例化模板时通过 **--param-file** 来使用该文件。如果要从标准输入中读取参数，请使用 **--param-file=-**。以下是一个名为 **helloworld.params** 的示例文件：

```
ADMIN_USERNAME=admin
ADMIN_PASSWORD=mypassword
```

在实例化模板时引用文件中的参数：

```
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
```

3.3.4. 修改应用程序创建

new-app 命令生成用于构建、部署和运行所创建应用程序的 OpenShift Container Platform 对象。通常情况下，这些对象是在当前项目中创建的，并分配有从输入源存储库或输入镜像中获得的名称。但是，您可以使用 **new-app** 修改这种行为。

表 3.2. new-app 输出对象

对象	描述
BuildConfig	为命令行中指定的每个源存储库创建一个 **BuildConfig** 对象。**BuildConfig** 对象指定要使用的策略、源位置和构建输出位置。
ImageStreams	对于 **BuildConfig** 对象，通常会创建两个镜像流。其一代表输入镜像。进行源构建时，这是构建器镜像。进行 **Docker** 构建时，这是 **FROM** 镜像。其二代表输出镜像。如果容器镜像指定为 **new-app** 的输入，那么也会为该镜像创建镜像流。
DeploymentConfig	创建一个 **DeploymentConfig** 对象来部署构建的输出或指定的镜像。**new-app** 命令为生成的 **DeploymentConfig** 对象中包含的容器中指定的所有 Docker 卷创建 **emptyDir** 卷。
Service	**new-app** 命令会尝试检测输入镜像中公开的端口。它使用编号最小的已公开端口来生成公开该端口的服务。要公开一个不同的端口，只需在 **new-app** 完成后使用 **oc expose** 命令生成额外服务。
其他	根据模板，可在实例化模板时生成其他对象。

3.3.4.1. 指定环境变量

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

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

这些变量可使用 **--env-file** 参数从文件中读取。以下是一个名为 **postgresql.env** 的示例文件：

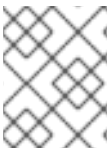
```
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
```

从文件中读取变量：

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

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

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



注意

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

3.3.4.2. 指定构建环境变量

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

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

这些变量可使用 **--build-env-file** 参数从文件中读取。以下是一个名为 **ruby.env** 的示例文件：

```
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
```

从文件中读取变量：

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

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

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

3.3.4.3. 指定标签

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

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

3.3.4.4. 查看输出但不创建

要查看运行 **new-app** 命令的空运行，您可以使用 **-o|--output** 参数及 **yaml** 或 **json** 值。然后，您可以使用输出结果预览创建的对象，或将其重定向到可以编辑的文件。满意之后，您可以使用 **oc create** 创建 OpenShift Container Platform 对象。

要将 **new-app** 工件输出到一个文件，请运行以下命令：

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

编辑该文件：

```
$ vi myapp.yaml
```

通过引用该文件来创建新应用程序：

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

3.3.4.5. 使用其他名称创建对象

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

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

3.3.4.6. 在另一项目中创建对象

通常，**new-app** 会在当前项目中创建对象。不过，您可以使用 **-n|--namespace** 参数在另一项目中创建对象：

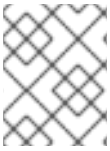
```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

3.3.4.7. 创建多个对象

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

从源存储库和 Docker Hub 镜像创建应用程序：

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



注意

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

3.3.4.8. 在单个 pod 中对镜像和源进行分组

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

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

将通过源构建的镜像和外部镜像一起部署：

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

3.3.4.9. 搜索镜像、模板和其他输入

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

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

第 4 章 使用 TOPOLOGY 视图查看应用程序组成

Web 控制台的 **Developer** 视角中有一个 **Topology** 视图，它以可视化方式展示项目中的所有应用程序、它们的构建状态，以及关联的组件和服务。

4.1. 先决条件

要在 **Topology** 视图中查看应用程序并与之交互，请确保：

- 已登陆到 web 控制台。
- 在项目中拥有适当的角色和权限， 可在 OpenShift Container Platform 中创建应用程序和其他工作负载。
- 您已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了应用程序。 。
- 处于 **Developer** 视角。

4.2. 查看应用程序拓扑

您可以使用 **Developer** 视角中的左侧导航面板进入 **Topology** 视图。部署应用程序后，您会自动定向到 **Graph view**，从中可查看应用程序 pod 状态，快速访问公共 URL 上的应用程序，访问源代码以进行修改，以及查看上一次构建的状态。您可以缩放视图来查看特定应用程序的更多详情。

Topology 视图为您提供了使用 **List** 视图监控应用程序的选项。使用 **List 视图** 图标  查看所有应用程序的列表，并使用 **图形视图** 图标()切回到图形视图。

您可以使用以下命令自定义视图：

- 使用 **Find by name** 字段查找所需组件。搜索结果可能会出现在可见区域之外；点击左侧工具栏中的 **Fit to Screen** 来改变 **Topology** 视图的大小来显示所有组件。
- 使用 **Display Options** 下拉列表配置各种应用程序的 **Topology** 视图。这些选项取决于项目中部署的组件的类型：
  - **展开**组
    - Virtual Machines：显示或隐藏虚拟机。
    - Application Groupings：通过概述应用程序组和与其关联的警报，将应用程序组压缩到卡中。
    - Helm Releases：将部署为 Helm Release 的组件整合到卡中，并概述给定的发行版本。
    - Knative Services：明确将 Knative Service 组件压缩到包含指定组件概述的卡中。
    - Operator Groupings：清除用于将 Operator 部署的组件整合到卡中，并包含给定组的概述。
  - **Show** 项基于 **Pod Count** 或 **Labels**
    - Pod Count：显示组件图标中组件的 pod 数量。

- Labels：显示或隐藏组件标签。

Topology 视图也为您提供了 **Export application** 选项，用于以 ZIP 文件格式下载应用程序。然后，您可以将下载的应用程序导入到另一个项目或集群。如需了解更多详细信息，请参阅*附加资源*部分的*将应用程序导出到另一个项目或集群*。

4.3. 与应用程序和组件交互

在 web 控制台的 **Developer** 视角中的 **Topology** 视图中，**Graph view** 提供了与应用程序和组件交互的选项：

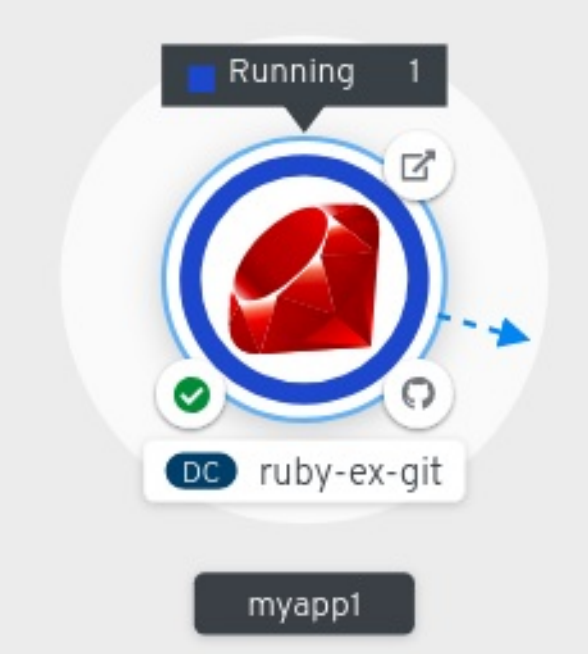
- 点 **Open URL**( )查看通过公共 URL 上的路由公开的应用程序。
- 点击 **Edit Source code** 可访问您的源代码并进行修改。

注意

只有使用 **From Git**、**From Catalog** 和 **From Dockerfile** 选项创建了应用程序时，此功能才可用。

- 光标悬停在 Pod 左下方图标上，可查看最新构建的名称及其状态。应用程序构建的状态显示为 **New** ( )、**Pending** ( )、**Running** ( )、**Complemented** ( )、**Failed** ( )和 **Canceled**( )。
- pod 的状态或阶段由不同的颜色和工具提示来表示：
  - **Running** （ ）：pod 绑定到节点，并创建所有容器。至少一个容器仍在运行，或正在启动或重启过程中。
  - **Not Ready**( )：运行多个容器的 pod，不是所有容器都已就绪。
  - **Warning** （ ）：pod 中的容器被终止，但终止没有成功。有些容器可能是其他状态。
  - **失败**( )：pod 中的所有容器都终止，但至少一个容器出现故障而终止。也代表，容器以非零状态退出，或者被系统终止。
  - **Pending** （ ）：Kubernetes 集群接受 pod，但一个或多个容器尚未设置并准备好运行。这包括 pod 等待调度的时间，以及通过网络下载容器镜像的时间。
  - **Succeeded** （ ）：pod 中的所有容器都成功终止，且不会被重启。
  - **Terminating** （ ）：当 pod 被删除时，一些 kubectl 命令会显示 **Terminating**。**Terminating** 状态不是 pod 的一个阶段。一个 pod 会被赋予一个安全终止期，默认为 30 秒。
  - **Unknown** （ ）：无法获取 pod 状态。此阶段通常是由于与 pod 应该运行的节点通信时出错造成的。
- 创建应用程序并部署镜像后，其状态会显示为 **Pending**。构建应用程序后，它会显示为 **Running**。

图 4.1. 应用程序拓扑



应用程序资源名称附有代表不同类型资源对象的指示符，如下所示：

- CJ: **CronJob**
- D: **Deployment**
- DC: **DeploymentConfig**
- DS: **DaemonSet**
- J: **Job**
- P: **Pod**
- SS: **StatefulSet**
- (Knative):无服务器应用程序

注意

无服务器应用程序需要一些时间才能加载并显示在 Graph 视图中。部署无服务器应用程序时，首先会创建一个服务资源，然后创建一个修订。之后，它会被部署并显示在 Graph 视图中。如果它是唯一的工作负载，可能会重定向到 Add 页面。部署修订后，无服务器应用程序会显示在 Graph 视图中。

4.4. 扩展应用程序 POD 以及检查构建和路由

Topology 视图在 Overview 面板中提供所部署组件的详情。您可以使用 Overview 和 Details 选项卡来扩展应用程序 pod，检查构建状态、服务和路由，如下所示：

- 点击组件节点，以查看右侧的 Overview 面板。使用 Details 选项卡进行：
  - 使用向上和向下箭头缩放 pod，手动增加或减少应用程序的实例数。对于无服务器应用程序，pod 数在空闲时会自动缩减为零，而且能根据频道流量扩展。

- 检查应用程序的 Labels、Annotations 和 Status。

- 点击 Resources 选项卡可以：
  - 查看所有 pod 列表，查看其状态，访问日志，还能点击 pod 来查看 pod 详情。
  - 查看构建及其状态，访问日志，并在需要时启动新的构建。
  - 查看组件所使用的服务和路由。

  对于无服务器应用程序，Resources 选项卡提供用于该组件的版本、路由和配置的有关信息。

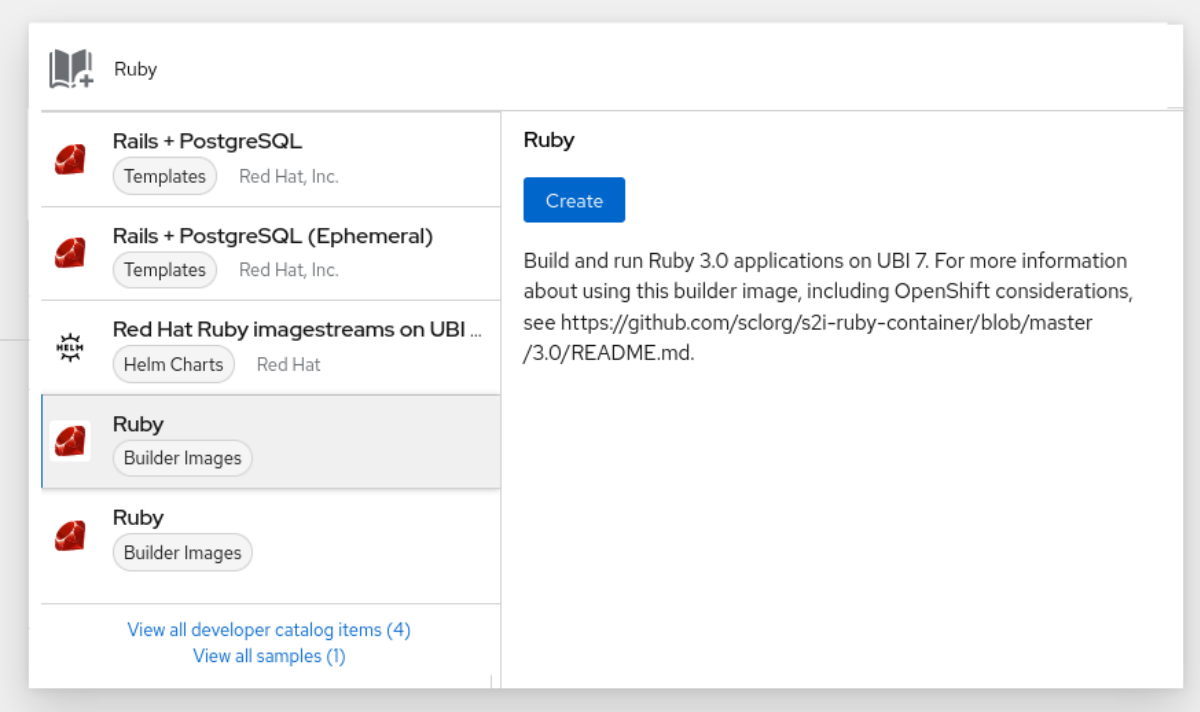
4.5. 将组件添加到现有项目

您可以向项目添加组件。

步骤

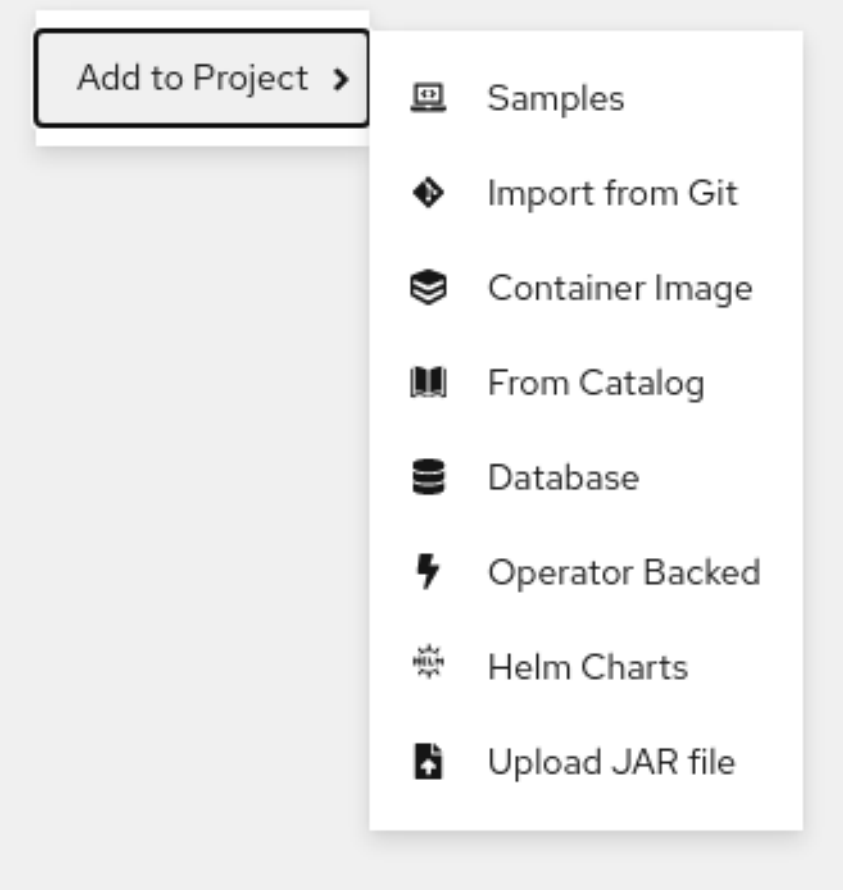
1. 进入 +Add 视图。
2. 点击左侧导航窗格旁的 Add to Project ( ) 或按 Ctrl+Space
3. 搜索组件并点 Start/Create/Install 按钮，或者点 Enter 将组件添加到项目中，并在拓扑 Graph 视图中查看它。

图 4.2. 通过快速搜索添加组件



另外，您还可以在上下文菜单中使用可用选项，如 Import from Git,Container Image,Database,From Catalog,Operator Backed,Helm Charts,Samples, 或 Upload JAR 文件，方法是在拓扑图形视图中添加组件。

图 4.3. 用于添加服务的上下文菜单



4.6. 对应用程序中的多个组件进行分组

您可以使用 **+Add** 视图在项目中添加多个组件或服务，并使用拓扑 **图形视图** 对应用程序组中的应用程序和资源进行分组。

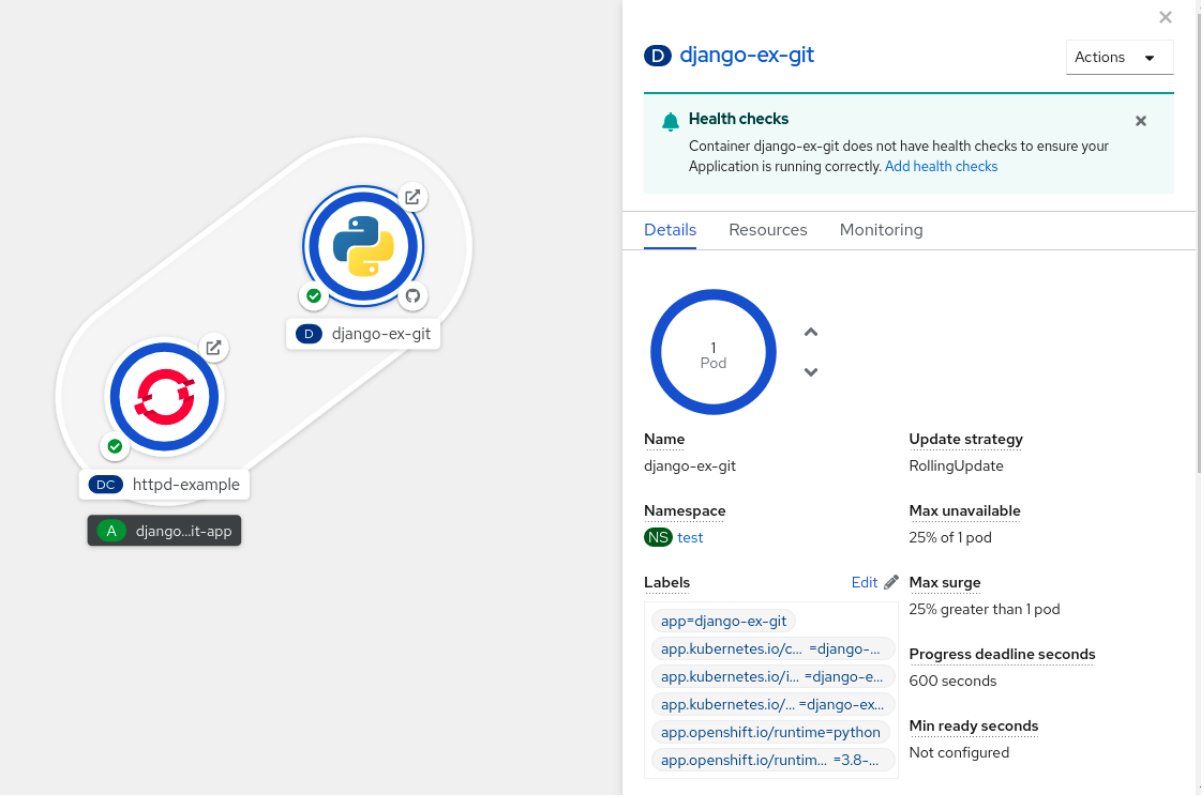
先决条件

- 您已使用 **Developer 视角** 在 OpenShift Container Platform 上创建并部署了最少两个或多个组件。

流程

- 要将服务添加到现有应用程序组中，请按 **Shift**+ 将它拖动到现有应用程序组中。拖动组件并将其添加到应用程序组中时，会将所需的标签添加到组件。

图 4.4. 应用程序分组



另外，您还可以在应用程序中添加组件，如下所示：

1. 点服务 pod 查看右侧的 **Overview** 面板。
2. 单击 **Actions** 下拉菜单，再选择 **Edit Application Grouping**。
3. 在 **Edit Application Grouping** 对话框中，单击 **Application** 下拉列表，然后选择适当的应用程序组。
4. 单击 **Save**，将服务添加到应用组中。

要从应用程序组中删除组件，您可以选择组件并使用 **Shift**+ 拖动操作将组件从应用程序组中拖出。

4.7. 在应用程序中添加服务

要在应用程序中添加服务，请使用使用拓扑 **图形视图中** 的上下文菜单的 **+Add** 操作。

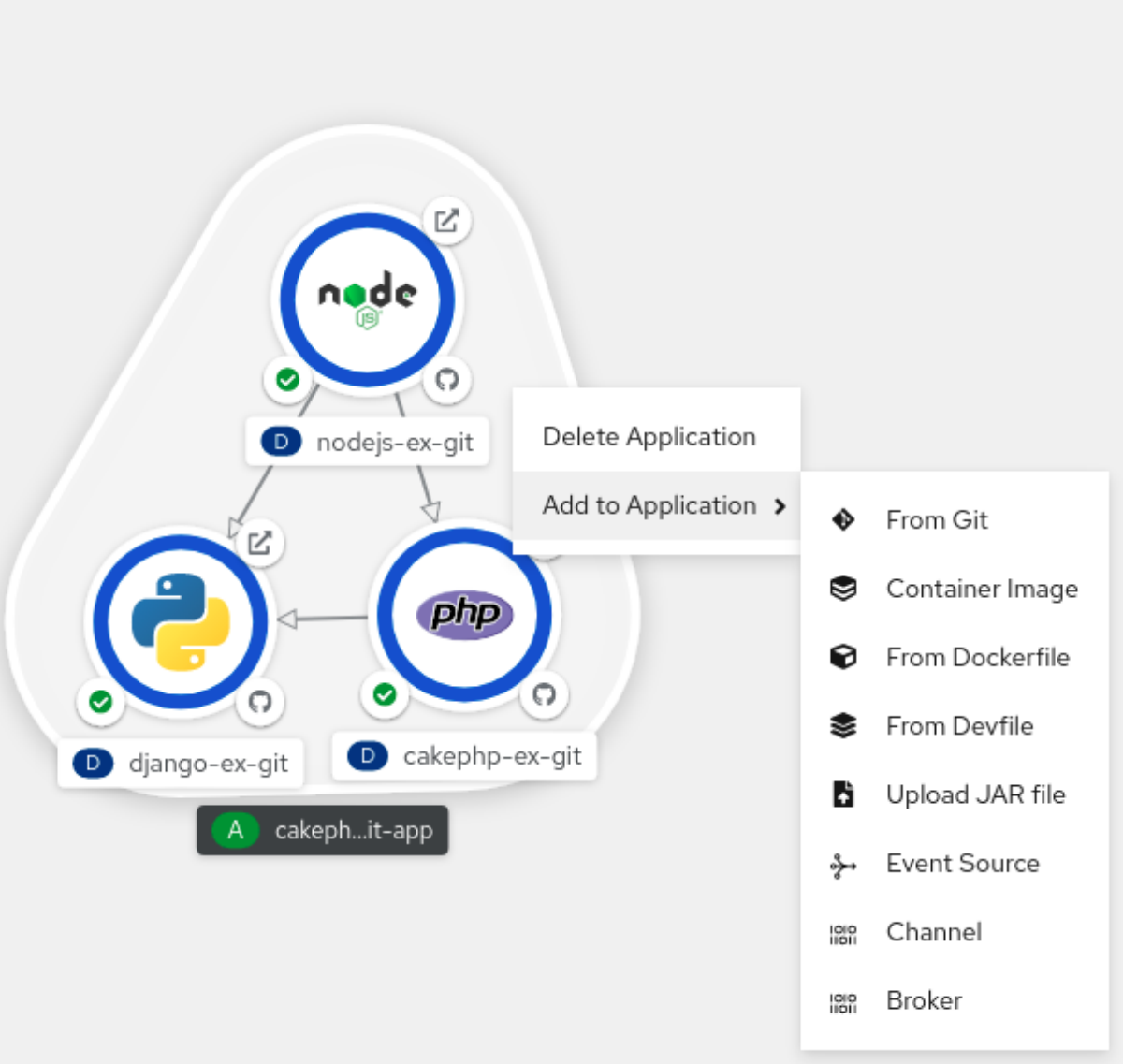
注意

除了上下文菜单外，您还可以使用边栏添加服务，或者将鼠标悬停在应用程序组中并拖动悬挂的箭头。

流程

1. 右键单击拓扑 **图形视图**中的应用程序组，以显示上下文菜单。

图 4.5. 添加资源上下文菜单



2. 使用 **Add to Application** 选择将服务添加到应用程序组中的方法，如 **From Git**、**Container Image**、**From Dockerfile**、**From Devfile**、**Upload JAR 文件**、**Event Source**、**Channel** 或 **Broker**。

3. 完成您选择的方法的表单，再单击 **Create**。例如，若要根据 Git 存储库中的源代码添加服务，请选择 **From Git** 方法，填写 **Import from Git** 表单，然后单击 **Create**。

4.8. 从应用程序中删除服务

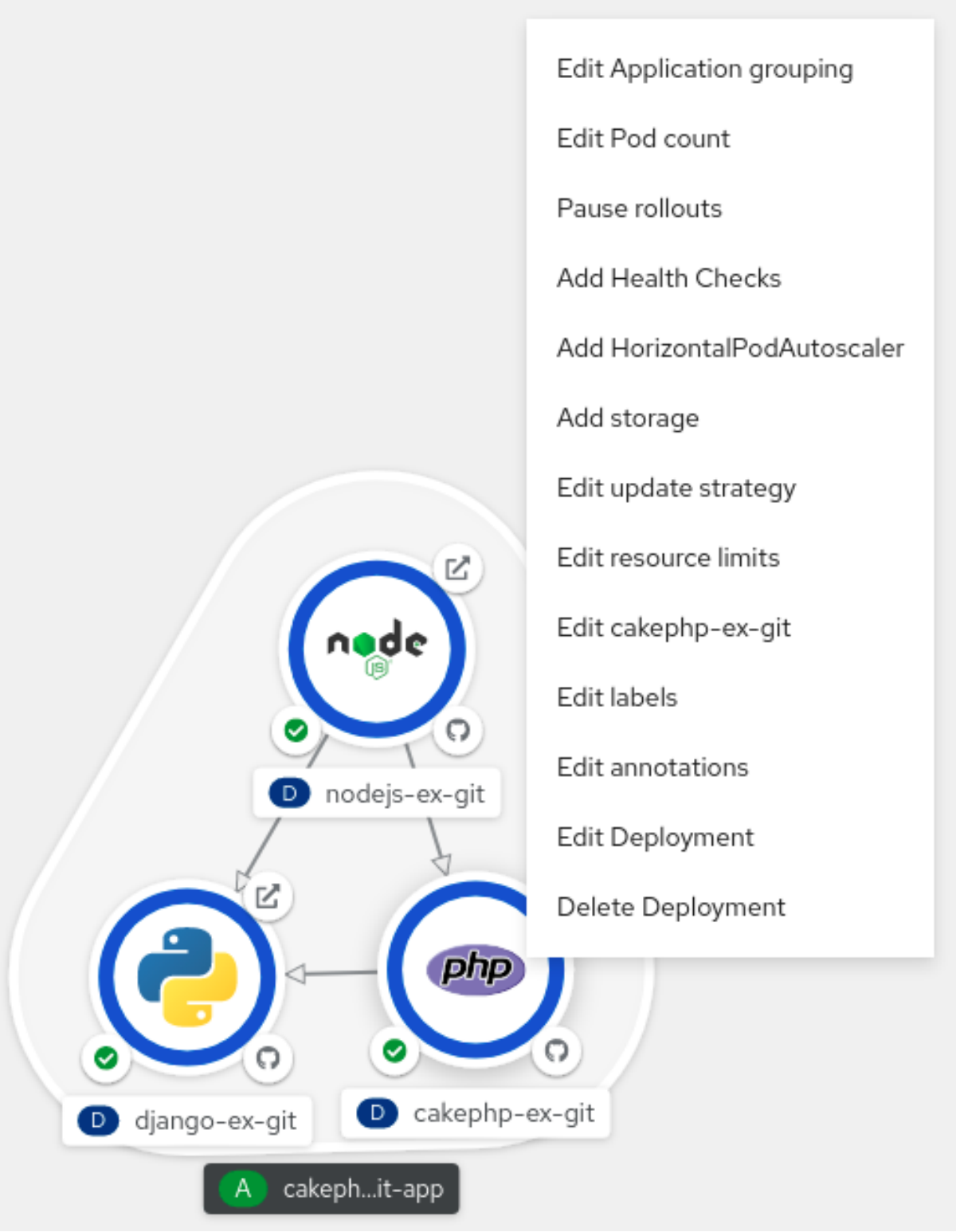
在拓扑 **图形视图**中，使用上下文菜单从应用程序中删除服务。

流程

1. 在拓扑 **图形视图**中的应用程序组中右键单击应用程序组中的服务，以显示上下文菜单。

2. 选择 **Delete Deployment** 以删除服务。

图 4.6. 删除部署选项



4.9. 用于 TOPOLOGY 视图的标签和注解

Topology 使用下列标签和注解：

节点中显示的图标

节点中的图标是通过使用 **app.openshift.io/runtime** 标签（随后是 **app.kubernetes.io/name** 标签）查找匹配图标来定义的。这种匹配是通过预定义的图标集合来完成的。

到源代码编辑器或源的链接

app.openshift.io/vcs-uri 注解用于创建源代码编辑器的链接。

节点连接器

app.openshift.io/connects-to 注解用于连接节点。

应用程序分组

app.kubernetes.io/part-of=<appname> 标签用于对应用程序、服务和组件进行分组。

如需了解 OpenShift Container Platform 应用程序必须使用的标签和注解，请参阅 OpenShift 应用程序的标签和注解指南。

4.10. 其他资源

- 请参阅从 Git 中导入代码库以创建应用程序，了解有关从 Git 创建应用程序的更多信息。
- 请参阅使用 Developer 视角将应用程序连接到服务。
- 请参阅导出应用程序

第 5 章 导出应用程序

作为开发人员，您可以使用 ZIP 文件格式导出应用程序。根据您的需要，使用 +Add 视图中的 Import YAML 选项将导出的应用程序导入到同一集群中的另一个项目或不同的集群。导出应用程序可帮助您重复使用应用程序资源并节省时间。

5.1. 先决条件

- 您已从 OperatorHub 安装了 gitops-primer Operator。

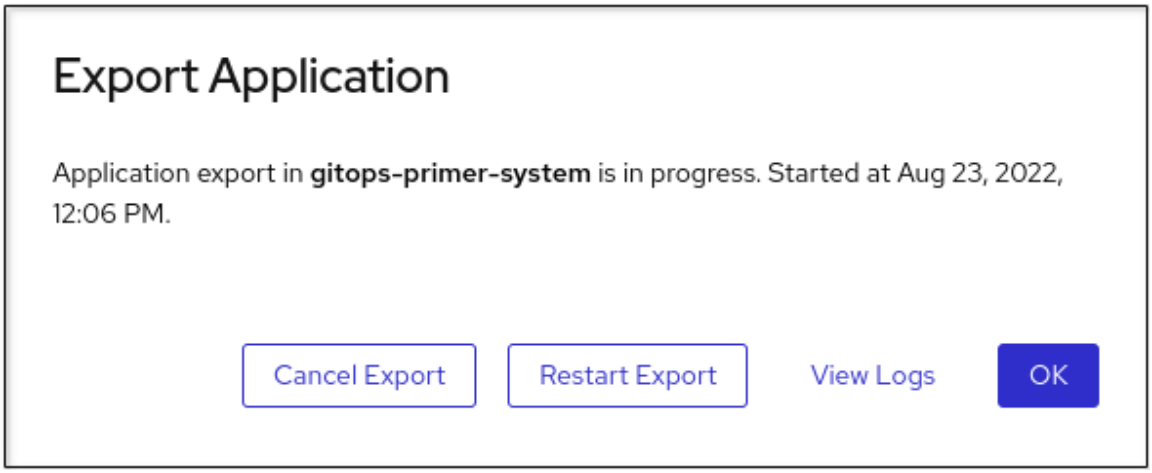
注意

Topology 视图中禁用了 **Export application** 选项，即使安装 gitops-primer Operator 后也是如此。

- 您已在 **Topology** 视图中创建了一个应用程序来启用 **导出应用程序**。

5.2. 流程

1. 在开发者视角中，执行以下步骤之一：
   a. 进入 **+Add** 视图，在 **Application portability** 标题中点 **Export application**。
   b. 进入到 **Topology** 视图，再点 **Export application**。
2. 在 **Export Application** 对话框中，点 **OK**。将打开一个通知，以确认您的项目中的资源导出是否已启动。
3. 在以下情况下可能需要执行的可选步骤：
   - 如果您启动导出不正确的应用程序，点 **Export application → Cancel Export**。
   - 如果您的导出已经进行，并且要启动一个新的导出，请点 **Export application → Restart Export**。
   - 如果要查看与导出应用程序关联的日志，请点 **Export application** 和 **View Logs** 链接。



4. 成功导出后，点对话框中的 **Download** 将 ZIP 格式的应用程序资源下载到您的机器中。

第 6 章 将应用程序连接到服务

6.1. SERVICE BINDING OPERATOR 发行注记

Service Binding Operator 由一个控制器和附带的自定义资源定义 (CRD) 组成，用于服务绑定。它管理工作负载的数据平面并提供支持服务。Service Binding Controller 读取由后备服务的 control plane 提供的数据。然后，它会根据通过 **ServiceBinding** 资源指定的规则将这些数据到工作负载。

使用 Service Binding Operator，您可以：

- 将工作负载与 Operator 管理的后备服务绑定。
- 自动配置绑定数据。
- 为服务提供商提供低接触管理经验，以调配和管理对服务的访问。
- 通过一致、声明性的服务绑定方法增强开发生命周期，消除群集环境中的差异。

Service Binding Operator 的自定义资源定义 (CRD) 支持以下 API：

- 使用 **binding.operators.coreos.com** API 组的**服务绑定**。
- **服务绑定 (Spec API)** 与 **servicebinding.io** API 组。

6.1.1. 支持列表

下表中的一些功能是技术预览。它们并不适用于在生产环境中使用。

在下表中，被标记为以下状态的功能：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 6.1. 支持列表

Service Binding Operator	API 组和支持状态		OpenShift 版本
版本	**binding.operators.coreos.com**	**servicebinding.io**	
1.3.3	GA	GA	4.9-4.12
1.3.1	GA	GA	4.9-4.11
1.3	GA	GA	4.9-4.11
1.2	GA	GA	4.7-4.11
1.1.1	GA	TP	4.7-4.10

Service Binding Operator	API 组和支持状态		OpenShift 版本
1.1	GA	TP	4.7-4.10
1.0.1	GA	TP	4.7-4.9
1.0	GA	TP	4.7-4.9

6.1.2. 使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。这些更改将在即将发行的几个发行本中逐渐实施。详情请查看 Red Hat CTO Chris Wright 信息。

6.1.3. Service Binding Operator 1.3.3 发行注记

Service Binding Operator 1.3.3 现在包括在 OpenShift Container Platform 4.9、4.10、4.11 和 4.12 中。

6.1.3.1. 修复的问题

- 在此次更新之前，为 Service Binding Operator 记录了一个安全漏洞 **CVE-2022-41717** 。在这个版本中修复了 **CVE-2022-41717** 错误，并将 v0.0.0-20220906165146-f3363e06e74c 中的 **golang.org/x/net** 软件包更新为 v0.4.0。 APPSVC-1256
- 在此次更新之前，只有在相应资源在其他置备服务丢失时设置了 "servicebinding.io/provisioned-service: true" 注解时，才会检测到 Provisioned Services。在这个版本中，检测机制会根据 "status.binding.name" 属性正确识别所有 Provisioned Services。APPSVC-1204

6.1.4. Service Binding Operator 1.3.1 发行注记

Service Binding Operator 1.3.1 现在包括在 OpenShift Container Platform 4.9、4.10 和 4.11 中。

6.1.4.1. 修复的问题

- 在此次更新之前，为 Service Binding Operator 记录了一个安全漏洞 **CVE-2022-32149** 。在这个版本中修复了 **CVE-2022-32149** 错误，并将 **golang.org/x/text** 软件包从 v0.3.7 更新至 v0.3.8。APPSVC-1220

6.1.5. Service Binding Operator 1.3 发行注记

Service Binding Operator 1.3 现在包括在 OpenShift Container Platform 4.9、4.10 和 4.11 上。

6.1.5.1. 删除的功能

- 在 Service Binding Operator 1.3 中，Operator Lifecycle Manager (OLM) 描述符功能已被删除，以改进资源利用率。作为 OLM 描述符的替代选择，您可以使用 CRD 注解来声明绑定数据。

6.1.6. Service Binding Operator 1.2 发行注记

Service Binding Operator 1.2 现在包括在 OpenShift Container Platform 4.7、4.8、4.9、4.10 和 4.11 上。

6.1.6.1. 新功能

本节重点介绍 Service Binding Operator 1.2 中的新内容：

- 通过将 **optional** 标志值设为 **true**，启用 Service Binding Operator 以考虑注解中的可选字段。
- 支持 **servicebinding.io/v1beta1** 资源。
- 通过公开相关绑定 secret 而无需存在工作负载，可以改进可绑定服务的可用性。

6.1.6.2. 已知问题

- 目前，当您在 OpenShift Container Platform 4.11 上安装 Service Binding Operator 时，Service Binding Operator 的内存占用量会高于预期限制。但是，使用低的情况下，内存占用量保留在环境或方案的预期范围内。与 OpenShift Container Platform 4.10 相比，在压力下，平均和最大内存占用率都显著提高。这个问题也会在以前的 Service Binding Operator 版本中识别。当前没有解决此问题的方法。APPSVC-1200
- 默认情况下，投射文件的权限设置为 0644。Service Binding Operator 无法因为 Kubernetes 中的一个错误而设置特定权限，这会导致在服务需要特定权限（如 **0600**）时出现问题。作为临时解决方案，您可以修改程序代码或正在工作负载资源中运行的应用程序，将文件复制到 **/tmp** 目录中并设置适当的权限。APPSVC-1127
- 在单一命名空间安装模式中安装 Service Binding Operator 当前存在一个已知问题。因为缺少一些命名空间范围的访问控制 (RBAC) 规则，应用程序可能无法成功绑定到 Service Binding Operator 可自动探测和绑定到的、几个已知的 Operator 支持的服务。当发生这种情况时，它会生成类似以下示例的错误消息：

  错误信息示例

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  临时解决方案 1：在 **all namespaces** 安装模式中安装 Service Binding Operator。这样，会存在适当的集群范围的 RBAC 规则，绑定会成功。

  临时解决方案 2：如果无法在 **all namespaces** 安装模式下安装 Service Binding Operator，在安装了 Service Binding Operator 的命名空间中安装以下角色绑定：

  示例：用于 Crunchy Postgres Operator 的角色绑定

  ```
  kind: RoleBinding
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: service-binding-crunchy-postgres-viewer
  subjects:
    - kind: ServiceAccount
      name: service-binding-operator
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: service-binding-crunchy-postgres-viewer-role
  ```

APPSVC-1062

- 根据规格，当更改 **ClusterWorkloadResourceMapping** 资源时，Service Binding Operator 必须使用早期版本的 **ClusterWorkloadResourceMapping** 资源来删除被项目的绑定数据。目前，当更改 **ClusterWorkloadResourceMapping** 资源时，Service Binding Operator 会使用 **ClusterWorkloadResourceMapping** 资源的最新版本来删除绑定数据。因此，{servicebinding-title} 可能会错误地删除绑定数据。作为临时解决方案，请执行以下步骤：

  1. 删除使用对应 **ClusterWorkloadResourceMapping** 资源的所有 **ServiceBinding** 资源。
  2. 修改 **ClusterWorkloadResourceMapping** 资源。
  3. 重新应用您之前在第 1 步中删除的 **ServiceBinding** 资源。

  APPSVC-1102

6.1.7. Service Binding Operator 1.1.1 发行注记

Service Binding Operator 1.1.1 现在包括在 OpenShift Container Platform 4.7、4.8、4.9 和 4.10 上。

6.1.7.1. 修复的问题

- 在此次更新之前，在 Service Binding Operator Helm chart 中记录了一个安全漏洞 **CVE-2021-38561**。在这个版本中，**CVE-2021-38561** 错误，并将 **golang.org/x/text** 软件包从 v0.3.6 更新至 v0.3.7。APPSVC-1124

- 在此次更新之前，Developer Sandbox 用户没有足够权限来读取 **ClusterWorkloadResourceMapping** 资源。因此，Service Binding Operator 会阻止所有服务绑定成功。在这个版本中，Service Binding Operator 会为任何经过身份验证的主体（包括 Developer Sandbox 用户）包含适当的基于角色的访问控制(RBAC)规则。这些 RBAC 规则允许 Service Binding Operator **get**, **list**, 和 **watch** Developer Sandbox 用户的 **ClusterWorkloadResourceMapping** 资源，并成功处理服务绑定。APPSVC-1135

6.1.7.2. 已知问题

- 在单一命名空间安装模式中安装 Service Binding Operator 当前存在一个已知问题。因为缺少一些命名空间范围的访问控制 (RBAC) 规则，应用程序可能无法成功绑定到 Service Binding Operator 可自动探测和绑定到的、几个已知的 Operator 支持的服务。当发生这种情况时，它会生成类似以下示例的错误消息：

  错误信息示例

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  临时解决方案 1：在 **all namespaces** 安装模式中安装 Service Binding Operator。这样，会存在适当的集群范围的 RBAC 规则，绑定会成功。

  临时解决方案 2：如果无法在 **all namespaces** 安装模式下安装 Service Binding Operator，在安装了 Service Binding Operator 的命名空间中安装以下角色绑定：

  示例：用于 Crunchy Postgres Operator 的角色绑定

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: service-binding-crunchy-postgres-viewer-role
```

APPSVC-1062

- 目前，当您修改 **ClusterWorkloadResourceMapping** 资源时，Service Binding Operator 不会实施正确的行为。作为临时解决方案，请执行以下步骤：

  1. 删除使用对应 **ClusterWorkloadResourceMapping** 资源的所有 **ServiceBinding** 资源。
  2. 修改 **ClusterWorkloadResourceMapping** 资源。
  3. 重新应用您之前在第 1 步中删除的 **ServiceBinding** 资源。

  APPSVC-1102

6.1.8. Service Binding Operator 1.1 发行注记

Service Binding Operator 现在包括在 OpenShift Container Platform 4.7、4.8、4.9 和 4.10 中。

6.1.8.1. 新功能

本节重点介绍 Service Binding Operator 1.1 中的新内容：

- 服务绑定选项
  - 工作负载资源映射：精确定义需要针对辅助工作负载投射绑定数据的位置。
  - 使用标签选择器绑定新工作负载。

6.1.8.2. 修复的问题

- 在此次更新之前，使用标签选择器提取工作负载的服务绑定不会将数据绑定到与给定标签选择器匹配的新工作负载中。因此，Service Binding Operator 无法定期绑定这些新工作负载。在这个版本中，服务绑定将数据绑定到与给定标签选择器匹配的新工作负载。Service Binding Operator 现在定期尝试查找和绑定这些新工作负载。APPSVC-1083

6.1.8.3. 已知问题

- 在单一命名空间安装模式中安装 Service Binding Operator 当前存在一个已知问题。因为缺少一些命名空间范围的访问控制 (RBAC) 规则，应用程序可能无法成功绑定到 Service Binding Operator 可自动探测和绑定到的、几个已知的 Operator 支持的服务。当发生这种情况时，它会生成类似以下示例的错误消息：

  错误信息示例

```
`postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
```

临时解决方案 1：在 **all namespaces** 安装模式中安装 Service Binding Operator。这样，会存在适当的集群范围的 RBAC 规则，绑定会成功。

临时解决方案 2：如果无法在 **all namespaces** 安装模式下安装 Service Binding Operator，在安装了 Service Binding Operator 的命名空间中安装以下角色绑定：

示例：用于 Crunchy Postgres Operator 的角色绑定

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: service-binding-crunchy-postgres-viewer-role
```

APPSVC-1062

- 目前，当您修改 **ClusterWorkloadResourceMapping** 资源时，Service Binding Operator 不会实施正确的行为。作为临时解决方案，请执行以下步骤：

  1. 删除使用对应 **ClusterWorkloadResourceMapping** 资源的所有 **ServiceBinding** 资源。
  2. 修改 **ClusterWorkloadResourceMapping** 资源。
  3. 重新应用您之前在第 1 步中删除的 **ServiceBinding** 资源。

  APPSVC-1102

6.1.9. Service Binding Operator 1.0.1 发行注记

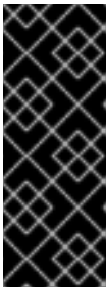
Service Binding Operator 现在包括在 OpenShift Container Platform 4.7、4.8 和 4.9 中。

Service Binding Operator 1.0.1 支持 OpenShift Container Platform 4.9 及之后的版本运行：

- IBM Power 系统
- IBM Z 和 LinuxONE

Service Binding Operator 1.0.1 的自定义资源定义(CRD)支持以下 API：

- 使用 **binding.operators.coreos.com** API 组的**服务绑定**。
- **服务绑定（Spec API 技术预览）**，使用 **servicebinding.io** API 组。



重要

带有 **servicebinding.io** API 组的 **Service Binding (Spec API Tech Preview)**只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

6.1.9.1. 支持列表

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

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 6.2. 支持列表

功能	Service Binding Operator 1.0.1
binding.operators.coreos.com API 组	GA
ServiceBinding.io API 组	TP

6.1.9.2. 修复的问题

- 在此次更新之前，从 CR 的 **postgresql.k8s.enterpriesedb.io/v1** API 的 **Cluster** 自定义资源(CR)中绑定数据值，从 CR 的 **.metadata.name** 字段收集 **主机** 绑定值。收集的绑定值是不正确的主机名，正确的主机名在 **.status.writeService** 字段中可用。在这个版本中，Service Binding Operator 用来公开后备服务 CR 的绑定数据值的注解现在被修改为从 **.status.writeService** 字段收集 **主机** 绑定值。Service Binding Operator 使用这些修改的注解来将正确的主机名注入到**主机**和**供应商**绑定中。APPSVC-1040

- 在这个更新之前，当您绑定 **postgres-operator.crunchydata.com/v1beta1** API 的 **PostgresCluster** CR 时，绑定数据值不包括数据库证书的值。因此，应用程序无法连接到数据库。在这个版本中，对 Service Binding Operator 用来从后备服务 CR 中公开绑定数据的注解的修改现在包含数据库证书。Service Binding Operator 使用以下修改后的注解来生成正确的 **ca.crt**、**tls.crt** 和 **tls.key** 证书文件。APPSVC-1045

- 在此次更新之前，当绑定 **pxc.percona.com** API 的 **PerconaXtraDBCluster** 自定义资源(CR)时，绑定数据值不包括 **端口**和**数据库**值。这些绑定值与其他已注入的绑定值是应用程序成功连接到数据库服务所需的值。在这个版本中，Service Binding Operator 用来公开服务 CR 的绑定数据值的注解现在被修改为项目额外的**端口**和**数据库**绑定值。Service Binding Operator 使用以下修改后的注解来生成完整的、应用程序用于成功连接到数据库服务所需的绑定值。APPSVC-1073

6.1.9.3. 已知问题

- 目前，当您在单一命名空间安装模式下安装 Service Binding Operator 时，因为缺少一些命名空间范围的访问控制(RBAC)规则，所以应用程序可能无法成功绑定到 Service Binding Operator 可自动探测和绑定到的、几个已知的 Operator 支持的服务。另外，还会生成以下出错信息：

  错误信息示例

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  临时解决方案 1：在 **all namespaces** 安装模式中安装 Service Binding Operator。这样，会存在适当的集群范围的 RBAC 规则，绑定会成功。

  临时解决方案 2：如果无法在 **all namespaces** 安装模式下安装 Service Binding Operator，在安装了 Service Binding Operator 的命名空间中安装以下角色绑定：

  示例：用于 Crunchy Postgres Operator 的角色绑定

  ```
  kind: RoleBinding
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: service-binding-crunchy-postgres-viewer
  subjects:
    - kind: ServiceAccount
      name: service-binding-operator
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: service-binding-crunchy-postgres-viewer-role
  ```

  APPSVC-1062

6.1.10. Service Binding Operator 1.0 发行注记

Service Binding Operator 现在包括在 OpenShift Container Platform 4.7、4.8 和 4.9 中。

Service Binding Operator 1.0 的自定义资源定义 (CRD) 支持以下 API：

- 使用 **binding.operators.coreos.com** API 组的**服务绑定**。
- **服务绑定（Spec API 技术预览）**，使用 **servicebinding.io** API 组。

重要

带有 **servicebinding.io** API 组的 **Service Binding (Spec API Tech Preview)**只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

6.1.10.1. 支持列表

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

技术预览功能支持范围

在下表中，功能被标记为以下状态：

- **TP**: *技术预览*
- **GA**: *正式发行*

请参阅红帽门户网站中关于对技术预览功能支持范围的信息：

表 6.3. 支持列表

功能	Service Binding Operator 1.0
binding.operators.coreos.com API 组	GA
ServiceBinding.io API 组	TP

6.1.10.2. 新功能

Service Binding Operator 1.0 支持 OpenShift Container Platform 4.9 及之后的版本运行：

- IBM Power 系统
- IBM Z 和 LinuxONE

本节重点介绍了 Service Binding Operator 1.0 中的新功能：

- 公开服务的绑定数据
  - 根据 CRD、自定义资源 (CR) 或资源中的注解。
  - 基于 Operator Lifecycle Manager (OLM) 描述符中存在的描述符。
  - 支持置备的服务
- 工作负载投射
  - 使用卷挂载以文件形式预测绑定数据。
  - 绑定数据作为环境变量的投射。
- 服务绑定选项
  - 在与工作负载命名空间不同的命名空间中绑定后备服务。
  - 项目绑定数据到特定的容器工作负载。
  - 自动检测来自后备服务 CR 拥有的资源的绑定数据。
  - 编写来自公开绑定数据的自定义绑定数据。
  - 支持非**PodSpec** 兼容工作负载资源。

- 安全性
  - 支持基于角色的访问控制 (RBAC)。

6.1.11. 其他资源

- 了解 Service Binding Operator。

6.2. 了解 SERVICE BINDING OPERATOR

应用程序开发人员需要访问支持的服务，以构建和连接工作负载。连接工作负载以支持服务始终是一个挑战，因为每个服务提供商建议以不同方式访问其机密并在工作负载中使用它们。此外，手动配置和维护这种工作负载和后备服务组合，使得流程变得繁琐、效率低下且容易出错。

Service Binding Operator 可让应用程序开发人员将工作负载与 Operator 管理的后备服务轻松绑定，而无需任何手动步骤来配置绑定连接。

6.2.1. 服务绑定术语

本节总结了服务绑定中使用的基本术语。

服务绑定	提供向工作负载提供服务的信息的操作表示。例如，在 Java 应用程序和它所需的数据库之间建立凭据交换。
后端服务	应用在网络上运行的任何服务或软件作为其正常操作的一部分。示例包括数据库、消息代理、带 REST 端点的应用、事件流、应用程序性能监控器 (APM) 或硬件安全模块 (HSM)。
工作负载（应用程序）	容器内运行的任何进程。示例包括 Spring Boot 应用、NodeJS Express 应用或 Ruby on Rails 应用。
绑定数据	有关用于配置集群中其他资源行为的服务的信息。示例包括凭证、连接详情、卷挂载或 secret。
绑定连接	任何在连接的组件（如可绑定后备服务）和需要支持服务的应用程序之间建立交互的连接。

6.2.2. 关于 Service Binding Operator

Service Binding Operator 由一个控制器和附带的自定义资源定义 (CRD) 组成，用于服务绑定。它管理工作负载的数据平面并提供支持服务。Service Binding Controller 读取由后备服务的 control plane 提供的数据。然后，它会根据通过 **ServiceBinding** 资源指定的规则将这些数据到工作负载。

因此，Service Binding Operator 通过自动收集和与工作负载共享绑定数据，使工作负载能够使用后备服务或外部服务。这个过程包括使后备服务绑定，并将工作负载和服务绑定在一起。

6.2.2.1. 使 Operator 管理的后备服务可绑定

要使服务可绑定，作为 Operator 供应商，您需要公开工作负载所需的绑定数据，以便与 Operator 提供的服务绑定。您可以在管理后备服务的 Operator CRD 中以注解或描述符形式提供绑定数据。

6.2.2.2. 将工作负载与后备服务绑定

通过将 Service Binding Operator 用作应用程序开发人员，您需要声明建立绑定连接的意图。您必须创建一个 **ServiceBinding** CR 来引用后端服务。此操作会触发 Service Binding Operator 将公开的绑定数据项目到工作负载中。Service Binding Operator 接收声明的意图，并将工作负载与后备服务绑定。

Service Binding Operator 的 CRD 支持以下 API：

- 使用 **binding.operators.coreos.com** API 组的**服务绑定**。
- **服务绑定 (Spec API)** 与 **servicebinding.io** API 组。

使用 Service Binding Operator，您可以：

- 将工作负载绑定到 Operator 管理的后备服务。
- 自动配置绑定数据。
- 为服务提供商提供低接触管理经验，以调配和管理对服务的访问。
- 通过一致、声明性的服务绑定方法增强开发生命周期，消除群集环境中的差异。

6.2.3. 主要特性

- 公开服务的绑定数据
  - 根据 CRD、自定义资源 (CR) 或资源中的注解。
- 工作负载投射
  - 使用卷挂载以文件形式预测绑定数据。
  - 绑定数据作为环境变量的投射。
- 服务绑定选项
  - 在与工作负载命名空间不同的命名空间中绑定后备服务。
  - 项目绑定数据到特定的容器工作负载。
  - 自动检测来自后备服务 CR 拥有的资源的绑定数据。
  - 编写来自公开绑定数据的自定义绑定数据。
  - 支持非**PodSpec** 兼容工作负载资源。
- 安全性
  - 支持基于角色的访问控制 (RBAC)。

6.2.4. API 的不同

Service Binding Operator 的 CRD 支持以下 API：

- 使用 **binding.operators.coreos.com** API 组的**服务绑定**。
- **服务绑定 (Spec API)** 与 **servicebinding.io** API 组。

这两个 API 组都有类似的功能，但它们并不完全一致。以下是这些 API 组之间的区别的完整列表：

功能	**binding.operators.coreos.com** API 组支持	由 **servicebinding.io** API 组支持	备注
绑定到置备的服务	是	是	不适用 (N/A)
直接 secret 投射	是	是	不适用 (N/A)
作为文件绑定	是	是	• **servicebinding.io** API 组的服务绑定的默认行为 • 服务绑定 **binding.operators.coreos.com** API 组的参与功能
作为环境变量绑定	是	是	• 服务绑定 **binding.operators.coreos.com** API 组的默认行为。 • 服务绑定 **servicebinding.io** API 组的参与功能：环境变量和文件一起创建。
使用标签选择器选择工作负载	是	是	不适用 (N/A)
发现绑定资源 (**.spec.detectBindingResources**)	是	否	**servicebinding.io** API 组没有对应的功能。
命名策略	是	否	**servicebinding.io** API 组中没有当前的机制来解释命名策略使用的模板。
容器路径	是	部分	因为 **binding.operators.coreos.com** API 组中的服务绑定可以在 **ServiceBinding** 资源中指定映射行为，所以 **servicebinding.io** API 组无法完全支持对等的行为，而无需更多与工作负载相关的信息。

功能	binding.operators.coreos.com API 组支持	由 servicebinding.io API 组支持	备注
容器名称过滤	否	是	**binding.operators.coreos.com** API 组没有等同的功能。
Secret 路径	是	否	**servicebinding.io** API 组没有对应的功能。
备用绑定源（例如，从注解中绑定数据）	是	被 Service Binding Operator 允许	该规范需要支持从置备的服务和 secret 获取数据。但是，对规格的严格读取表示，支持其他绑定数据源。使用这个事实，Service Binding Operator 可以从各种源拉取绑定数据（例如，从注解中拉取绑定数据）。Service Binding Operator 在两个 API 组上支持这些源。

6.2.5. 其他资源

- 服务绑定入门

6.3. 安装 SERVICE BINDING OPERATOR

本指南指导集群管理员完成将 Service Binding Operator 安装到 OpenShift Container Platform 集群的过程。

您可以在 OpenShift Container Platform 4.7 及更新的版本上安装 Service Binding Operator。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 您的集群启用了 Marketplace 功能，或者手动配置 Red Hat Operator 目录源。

6.3.1. 使用 Web 控制台安装 Service Binding Operator

您可以使用 OpenShift Container Platform OperatorHub 安装 Service Binding Operator。安装 Service Binding Operator 时，服务绑定配置所需的自定义资源 (CR) 将与 Operator 一起自动安装。

流程

1. 在控制台的 **Administrator** 视角中，导航到 **Operators → OperatorHub**。
2. 使用 **Filter by keyword** 复选框在目录中搜索 **Service Binding Operator**。点 **Service Binding Operator** 标题。

3. 阅读 **Service Binding Operator** 页面中有关 Operator 的简单描述。点 **Install**。

4. 在 **Install Operator** 页面中：

   a. 为 **Installation Mode** 选择 **All namespaces on the cluster (default)**。选择该项会将 Operator 安装至默认**openshift-operators** 命名空间，这将启用 Operator 以进行监视并在集群中的所有命名空间中可用。

   b. 为 **Approval Strategy** 选择 **Automatic**。这样可确保以后对 Operator 的升级由 Operator Lifecycle Manager (OLM) 自动进行。如果您选择 **Manual** 批准策略，OLM 会创建一个更新请求。作为集群管理员，您必须手动批准 OLM 更新请求，才可将 Operator 更新至新版本。

   c. 选择一个 **Update Channel**。

      - 默认情况下，**stable** 频道启用 Service Binding Operator 最新稳定且受支持的发行版本。

5. 点 **Install**。

注意

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

6. 在 **Installed Operator – ready for use** 窗格中，点 **View Operator**。您会看到 **Installed Operators** 页面中列出的 Operator。

7. 验证 **Status** 是否已设置为 **Succeeded** 来确认已成功安装 Service Binding Operator。

6.3.2. 其它资源

- 服务绑定入门.

6.4. 服务绑定入门

Service Binding Operator 管理工作负载和后备服务的数据平面。本指南提供了一些示例，可帮助您创建数据库实例、部署应用程序，以及使用 Service Binding Operator 在应用程序和数据库服务间创建绑定连接。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已安装 **oc** CLI。
- 已从 OperatorHub 安装了 Service Binding Operator。
- 您已使用 **v5** 更新频道从 OperatorHub 安装 Crunchy Postgres for Kubernetes Operator 的 5.1.2 版本。安装的 Operator 在适当的命名空间中可用，如 **my-petclinic** 命名空间。

注意

您可以使用 **oc create namespace my-petclinic** 命令创建命名空间。

6.4.1. 创建 PostgreSQL 数据库实例

要创建 PostgreSQL 数据库实例，您必须创建一个 **PostgresCluster** 自定义资源 (CR) 并配置数据库。

流程

1. 在 shell 中运行以下命令来在 **my-petclinic** 命名空间中创建 **PostgresCluster** CR：

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.4-0
  postgresVersion: 14
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgbackrest:ubi8-2.38-0
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
EOD
```

此 **PostgresCluster** CR 中的注解启用服务绑定连接并触发 Operator 协调。

输出会验证数据库实例是否已创建：

输出示例

```
postgrescluster.postgres-operator.crunchydata.com/hippo created
```

2. 创建数据库实例后，确保 **my-petclinic** 命名空间中的所有 pod 都在运行：

```
$ oc get pods -n my-petclinic
```

输出（需要几分钟）验证是否创建并配置了数据库：

输出示例

```
NAME                          READY   STATUS      RESTARTS   AGE
hippo-backup-9rxm-88rzq         0/1     Completed   0          2m2s
hippo-instance1-6psd-0          4/4     Running     0          3m28s
hippo-repo-host-0               2/2     Running     0          3m28s
```

配置了数据库后，您可以部署示例应用程序并将其连接到数据库服务。

6.4.2. 部署 Spring PetClinic 示例应用程序

要在 OpenShift Container Platform 集群上部署 Spring PetClinic 示例应用程序，您必须使用部署配置并配置本地环境才能测试应用程序。

流程

1. 在 shell 中运行以下命令，使用 **PostgresCluster** 自定义资源(CR)部署 **spring-petclinic** 应用程序：

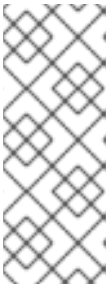
```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-petclinic
  labels:
    app: spring-petclinic
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-petclinic
  template:
    metadata:
      labels:
        app: spring-petclinic
    spec:
      containers:
        - name: app
          image: quay.io/service-binding/spring-petclinic:latest
          imagePullPolicy: Always
          env:
          - name: SPRING_PROFILES_ACTIVE
            value: postgres
          ports:
          - name: http
            containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spring-petclinic
  name: spring-petclinic
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
```

```
  selector:
    app: spring-petclinic
EOD
```

输出会验证 Spring PetClinic 示例应用是否已创建并部署：

输出示例

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```



注意

如果要在 web 控制台的 **Developer** 视角中使用 **容器镜像** 部署应用程序，则必须在**高级选项**的 **Deployment** 部分输入以下环境变量：

- 名称：SPRING_PROFILES_ACTIVE
- 值： postgres

2. 运行以下命令，验证应用程序是否还没有连接到数据库服务：

```
$ oc get pods -n my-petclinic
```

显示 **CrashLoopBackOff** 状态需要几分钟时间 ：

输出示例

```
NAME                              READY   STATUS             RESTARTS      AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago)   2m25s
```

在这个阶段，pod 无法启动。如果您尝试与应用交互，它会返回错误。

3. 公开服务来为应用程序创建路由：

```
$ oc expose service spring-petclinic -n my-petclinic
```

检查输出来验证 **spring-petclinic** 服务是否已公开，同时创建了 Spring PetClinic 示例应用程序的路由：

输出示例

```
route.route.openshift.io/spring-petclinic exposed
```

现在，您可以使用 Service Binding Operator 将应用程序连接到数据库服务。

6.4.3. 将 Spring PetClinic 示例应用程序连接到 PostgreSQL 数据库服务

要将示例应用程序连接到数据库服务，您必须创建一个 **ServiceBinding** 自定义资源 (CR)，该资源会触发 Service Binding Operator 将绑定数据项目到应用程序中。

流程

1. 创建 **ServiceBinding** CR 以项目绑定数据：

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
  services: 1
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster 2
      name: hippo
  application: 3
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
EOD
```

1 指定服务资源列表。

2 数据库的 CR。

3 示例应用程序，指向带有嵌入式 PodSpec 的 Deployment 或任何其他类似资源。

输出会验证是否已创建 **ServiceBinding** CR 以将绑定数据项目到示例应用程序中。

输出示例

```
servicebinding.binding.operators.coreos.com/spring-petclinic created
```

2. 验证服务绑定的请求是否成功：

```
$ oc get servicebindings -n my-petclinic
```

输出示例

```
NAME                      READY   REASON            AGE
spring-petclinic-pgcluster   True    ApplicationsBound   7s
```

默认情况下，来自数据库服务的绑定数据的值作为运行示例应用程序的工作负载容器中的文件进行投射。例如，Secret 资源中的所有值都投射到 **bindings/spring-petclinic-pgcluster** 目录中。

注意

另外，您还可以通过打印出目录内容来验证应用程序中的文件是否包含投射绑定数据：

```
$ for i in username password host port type; do oc exec -it deploy/spring-petclinic -n my-petclinic -- /bin/bash -c 'cd /tmp; find /bindings/*/'$i' -exec echo -n {}:" " \; -exec cat {} \;'; echo; done
```

输出示例：使用来自 secret 资源的所有值

```
/bindings/spring-petclinic-pgcluster/username: <username>
/bindings/spring-petclinic-pgcluster/password: <password>
/bindings/spring-petclinic-pgcluster/host: hippo-primary.my-petclinic.svc
/bindings/spring-petclinic-pgcluster/port: 5432
/bindings/spring-petclinic-pgcluster/type: postgresql
```

3. 设置应用程序端口的端口转发，以便从本地环境访问示例应用程序：

```
$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic
```

输出示例

```
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
```

4. 访问 http://localhost:8080/petclinic。
现在，您可以在 localhost:8080 远程访问 Spring PetClinic 示例应用程序，并查看应用程序现在连接到数据库服务。

6.4.4. 其它资源

- 安装 Service Binding Operator.
- 使用 Developer 视角创建应用程序。
- 管理自定义资源定义中的资源。
- 已知的可绑定 Operator。

6.5. 在 IBM POWER、IBM Z 和 IBM (R) LINUX 上使用服务绑定

Service Binding Operator 管理工作负载和后备服务的数据平面。本指南提供了一些示例，可帮助您创建数据库实例、部署应用程序，以及使用 Service Binding Operator 在应用程序和数据库服务间创建绑定连接。

先决条件

- 可以使用具有 **cluster-admin** 权限的账户访问 OpenShift Container Platform 集群。
- 已安装 **oc** CLI。
- 您已从 OperatorHub 安装 Service Binding Operator。

6.5.1. 部署 PostgreSQL Operator

流程

1. 要在 **my-petclinic** 命名空间中部署 Dev4Devs PostgreSQL Operator，请在 shell 中运行以下命令：

```
$ oc apply -f - << EOD
---
apiVersion: v1
kind: Namespace
metadata:
  name: my-petclinic
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: postgres-operator-group
  namespace: my-petclinic
---
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: ibm-multiarch-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: quay.io/ibm/operator-registry-<architecture> 1
  imagePullPolicy: IfNotPresent
  displayName: ibm-multiarch-catalog
  updateStrategy:
    registryPoll:
      interval: 30m
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: postgresql-operator-dev4devs-com
  namespace: openshift-operators
spec:
  channel: alpha
  installPlanApproval: Automatic
  name: postgresql-operator-dev4devs-com
  source: ibm-multiarch-catalog
  sourceNamespace: openshift-marketplace
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: database-view
  labels:
    servicebinding.io/controller: "true"
rules:
  - apiGroups:
      - postgresql.dev4devs.com
```

```
    resources:
      - databases
    verbs:
      - get
      - list
EOD
```

1 Operator 镜像。

- 对于 IBM Power： **quay.io/ibm/operator-registry-ppc64le:release-4.9**
- 对于 IBM Z 和 IBM® LinuxONE: **quay.io/ibm/operator-registry-s390x:release-4.8**

验证

1. 安装 Operator 后，列出 **openshift-operators** 命名空间中的 Operator 订阅：

```
$ oc get subs -n openshift-operators
```

输出示例

```
NAME                                PACKAGE                             SOURCE                CHANNEL
postgresql-operator-dev4devs-com    postgresql-operator-dev4devs-com    ibm-multiarch-catalog    alpha
rh-service-binding-operator         rh-service-binding-operator         redhat-operators         stable
```

6.5.2. 创建 PostgreSQL 数据库实例

要创建 PostgreSQL 数据库实例，您必须创建一个 **Database** 自定义资源(CR)并配置数据库。

流程

1. 在 shell 中运行以下命令来在 **my-petclinic** 命名空间中创建 **Database** CR：

```
$ oc apply -f - << EOD
apiVersion: postgresql.dev4devs.com/v1alpha1
kind: Database
metadata:
  name: sampledatabase
  namespace: my-petclinic
  annotations:
    host: sampledatabase
    type: postgresql
    port: "5432"
    service.binding/database: 'path={.spec.databaseName}'
    service.binding/port: 'path={.metadata.annotations.port}'
    service.binding/password: 'path={.spec.databasePassword}'
    service.binding/username: 'path={.spec.databaseUser}'
    service.binding/type: 'path={.metadata.annotations.type}'
    service.binding/host: 'path={.metadata.annotations.host}'
spec:
  databaseCpu: 30m
  databaseCpuLimit: 60m
```

```
  databaseMemoryLimit: 512Mi
  databaseMemoryRequest: 128Mi
  databaseName: "sampledb"
  databaseNameKeyEnvVar: POSTGRESQL_DATABASE
  databasePassword: "samplepwd"
  databasePasswordKeyEnvVar: POSTGRESQL_PASSWORD
  databaseStorageRequest: 1Gi
  databaseUser: "sampleuser"
  databaseUserKeyEnvVar: POSTGRESQL_USER
  image: registry.redhat.io/rhel8/postgresql-13:latest
  databaseStorageClassName: nfs-storage-provisioner
  size: 1
EOD
```

此 **Database** CR 中添加的注解可启用服务绑定连接并触发 Operator 协调。

输出会验证数据库实例是否已创建：

输出示例

```
database.postgresql.dev4devs.com/sampledatabase created
```

2. 创建数据库实例后，确保 **my-petclinic** 命名空间中的所有 pod 都在运行：

```
$ oc get pods -n my-petclinic
```

输出（需要几分钟）验证是否创建并配置了数据库：

输出示例

```
NAME                                 READY   STATUS    RESTARTS   AGE
sampledatabase-cbc655488-74kss       0/1     Running   0          32s
```

配置了数据库后，您可以部署示例应用程序并将其连接到数据库服务。

6.5.3. 部署 Spring PetClinic 示例应用程序

要在 OpenShift Container Platform 集群上部署 Spring PetClinic 示例应用程序，您必须使用部署配置并配置本地环境才能测试应用程序。

流程

1. 在 shell 中运行以下命令，使用 **PostgresCluster** 自定义资源(CR)部署 **spring-petclinic** 应用程序：

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-petclinic
  labels:
    app: spring-petclinic
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: spring-petclinic
  template:
    metadata:
      labels:
        app: spring-petclinic
    spec:
      containers:
        - name: app
          image: quay.io/service-binding/spring-petclinic:latest
          imagePullPolicy: Always
          env:
          - name: SPRING_PROFILES_ACTIVE
            value: postgres
          - name: org.springframework.cloud.bindings.boot.enable
            value: "true"
          ports:
          - name: http
            containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spring-petclinic
  name: spring-petclinic
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    app: spring-petclinic
EOD
```

输出会验证 Spring PetClinic 示例应用是否已创建并部署：

输出示例

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```

注意

如果要在 web 控制台的 **Developer** 视角中使用 **容器镜像** 部署应用程序，则必须在**高级选项**的 **Deployment** 部分输入以下环境变量：

- 名称：SPRING_PROFILES_ACTIVE
- 值： postgres

2. 运行以下命令，验证应用程序是否还没有连接到数据库服务：

```
$ oc get pods -n my-petclinic
```

显示 **CrashLoopBackOff** 状态需要几分钟：

输出示例

```
NAME                                READY   STATUS             RESTARTS      AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago)   2m25s
```

在这个阶段，pod 无法启动。如果您尝试与应用交互，它会返回错误。

现在，您可以使用 Service Binding Operator 将应用程序连接到数据库服务。

6.5.4. 将 Spring PetClinic 示例应用程序连接到 PostgreSQL 数据库服务

要将示例应用程序连接到数据库服务，您必须创建一个 **ServiceBinding** 自定义资源 (CR)，该资源会触发 Service Binding Operator 将绑定数据项目到应用程序中。

流程

1. 创建 **ServiceBinding** CR 以项目绑定数据：

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
  services: 1
  - group: postgresql.dev4devs.com
    kind: Database 2
    name: sampledatabase
    version: v1alpha1
  application: 3
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
EOD
```

1. 指定服务资源列表。
2. 数据库的 CR。
3. 示例应用程序，指向带有嵌入式 PodSpec 的 Deployment 或任何其他类似资源。

输出会验证是否已创建 **ServiceBinding** CR 以将绑定数据项目到示例应用程序中。

输出示例

```
servicebinding.binding.operators.coreos.com/spring-petclinic created
```

2. 验证服务绑定的请求是否成功：

```
$ oc get servicebindings -n my-petclinic
```

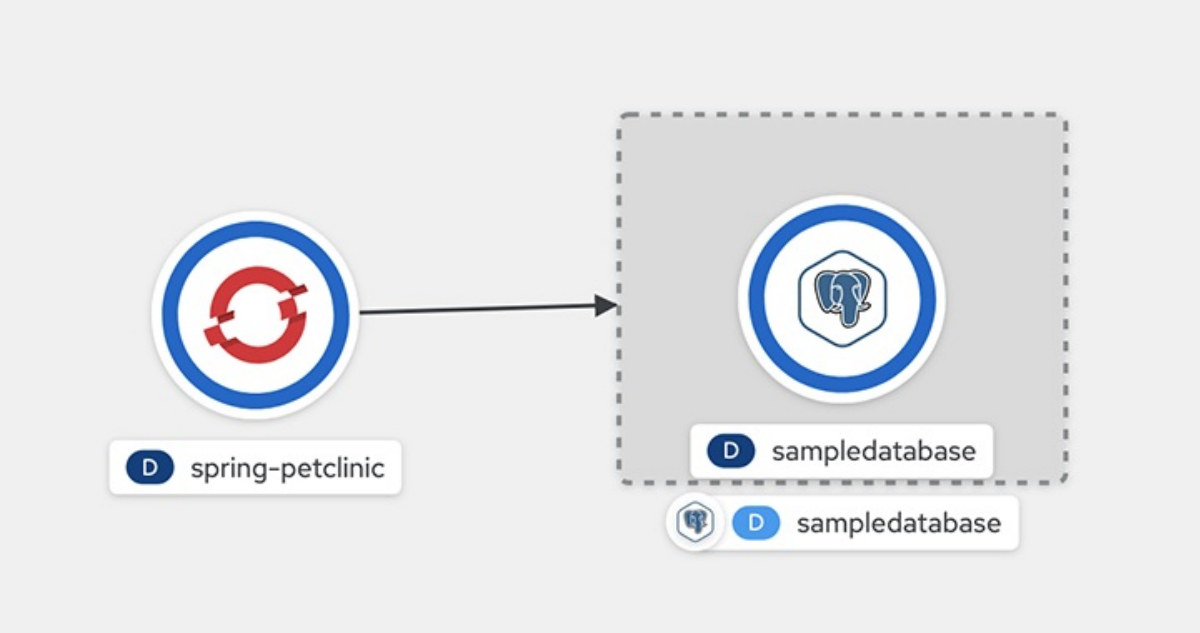
输出示例

```
NAME                        READY   REASON              AGE
spring-petclinic-postgresql   True    ApplicationsBound   47m
```

默认情况下，来自数据库服务的绑定数据的值作为运行示例应用程序的工作负载容器中的文件进行投射。例如，Secret 资源中的所有值都投射到 **bindings/spring-petclinic-pgcluster** 目录中。

3. 创建后，您可以进入拓扑来查看可视连接。

图 6.1. 将 spring-petclinic 连接到示例数据库



4. 设置应用程序端口的端口转发，以便从本地环境访问示例应用程序：

```
$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic
```

输出示例

```
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
```

5. 访问 http://localhost:8080。
现在，您可以在 localhost:8080 远程访问 Spring PetClinic 示例应用程序，并查看应用程序现在连接到数据库服务。

6.5.5. 其他资源

- 安装 Service Binding Operator
- 使用 Developer 视角创建应用程序
- 管理自定义资源定义中的资源

6.6. 从服务公开绑定数据

应用程序开发人员需要访问支持的服务，以构建和连接工作负载。连接工作负载以支持服务始终是一个挑战，因为每个服务提供商需要以不同的方式访问其机密并在工作负载中使用它们。

Service Binding Operator 可让应用程序开发人员将工作负载与 Operator 管理的后备服务轻松绑定，而无需任何手动步骤来配置绑定连接。要使 Service Binding Operator 作为 Operator 供应商或创建后备服务的用户提供绑定数据，您必须公开绑定数据，以便 Service Binding Operator 自动检测到绑定数据。然后，Service Binding Operator 会自动从后备服务收集绑定数据，并将其与工作负载共享，以提供一致且可预测的体验。

6.6.1. 公开绑定数据的方法

本节论述了您可以使用什么方法公开绑定数据。

确保您了解并了解您的工作负载要求和环境，以及如何与所提供的服务配合使用。

在以下情况下公开绑定数据：

- 后备服务作为调配的服务资源提供。
  您要连接到的服务符合 Service Binding 规格。您必须创建一个带有所有所需绑定数据值的 **Secret** 资源，并在后备服务自定义资源 (CR) 中引用它。所有绑定数据值的检测都是自动的。

- 后备服务不能作为调配的服务资源使用。
  您必须从后备服务公开绑定数据。根据工作负载要求和环境，您可以选择以下任一方法公开绑定数据：

  - 直接 secret 引用
  - 通过自定义资源定义(CRD)或 CR 注解声明绑定数据
  - 通过拥有的资源检测绑定数据

6.6.1.1. 置备的服务

置备的服务代表后端服务 CR，引用了放置在后备服务 CR 的 **.status.binding.name** 字段中的 **Secret** 资源。

作为 Operator 供应商或创建后备服务的用户，您可以通过创建 **Secret** 资源并在后备服务 CR 的 **.status.binding.name** 部分中引用它，使用此方法符合 Service Binding 规格。此 **Secret** 资源必须提供工作负载连接到后备服务所需的所有绑定数据值。

以下示例显示一个 **AccountService** CR，它代表一个后备服务和从 CR 引用的 **Secret** 资源。

示例： AccountService CR

```
apiVersion: example.com/v1alpha1
kind: AccountService
name: prod-account-service
spec:
# ...
status:
  binding:
    name: hippo-pguser-hippo
```

示例：引用的 Secret 资源

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
# ...
```

在创建服务绑定资源时，您可以在 **ServiceBinding** 规格中直接提供 **AccountService** 资源的详情，如下所示：

示例： ServiceBinding 资源

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  services:
  - group: "example.com"
    version: v1alpha1
    kind: AccountService
    name: prod-account-service
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

示例： Specification API 中的 ServiceBinding 资源

```
apiVersion: servicebinding.io/v1beta1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  service:
    apiVersion: example.com/v1alpha1
    kind: AccountService
    name: prod-account-service
  workload:
    apiVersion: apps/v1
    kind: Deployment
    name: spring-petclinic
```

此方法公开 **hippo-pguser-hippo** 所引用 **Secret** 资源中的所有键作为绑定数据，以投射到工作负载中。

6.6.1.2. 直接 secret 引用

如果所有必需的绑定数据值都位于 Service Binding 定义中可以引用的 **Secret** 资源中，您可以使用此方法。在这个方法中，**ServiceBinding** 资源直接引用 **Secret** 资源来连接服务。**Secret** 资源中的所有键都公开为绑定数据。

示例：带有 binding.operators.coreos.com API 的规格

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  services:
  - group: ""
    version: v1
    kind: Secret
    name: hippo-pguser-hippo
```

示例：符合 servicebinding.io API 的规格

```
apiVersion: servicebinding.io/v1beta1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  service:
    apiVersion: v1
    kind: Secret
    name: hippo-pguser-hippo
```

6.6.1.3. 通过 CRD 或 CR 注解声明绑定数据

您可以使用此方法注解后备服务的资源，以使用特定注解公开绑定数据。在 **metadata** 部分下添加注解会改变后备服务的 CR 和 CRD。Service Binding Operator 会检测添加到 CR 和 CRD 的注解，然后使用基于注解提取的值创建一个 **Secret** 资源。

以下示例显示了在 **metadata** 部分添加的注解以及从资源引用的 **ConfigMap** 对象：

示例：从 CR 注解中定义的 Secret 对象公开绑定数据

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
# ...
```

前面的示例将 secret 名称放在 **{.metadata.name}-pguser-{.metadata.name}** 模板中，该模板解析为 **hippo-pguser-hippo**。模板可以包含多个 JSONPath 表达式。

示例：从一个资源中参考的 Secret 对象

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
```

示例：从 CR 注解中定义的 ConfigMap 对象公开绑定数据

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-config,objectType=ConfigMap'
# ...
```

前面的示例将配置映射的名称放在解析到 **hippo-config** 的 **{.metadata.name}-config** 模板中。模板可以包含多个 JSONPath 表达式。

示例：从资源引用的 ConfigMap 对象

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

6.6.1.4. 通过拥有的资源检测绑定数据

如果您的后备服务拥有一个或多个 Kubernetes 资源，如路由、服务、配置映射或 secret，您可以使用此方法检测绑定数据。在这个方法中，Service Binding Operator 会检测来自后备服务 CR 拥有的资源的绑定数据。

以下示例显示 **ServiceBinding** CR 中的 **detectBindingResources** API 选项设置为 **true** ：

Example

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-detect-all
  namespace: my-petclinic
spec:
  detectBindingResources: true
  services:
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster
      name: hippo
```

```
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

在上例中，**PostgresCluster** 自定义资源拥有一个或多个 Kubernetes 资源，如路由、服务、配置映射或 secret。

Service Binding Operator 会自动检测每个拥有的资源上公开的绑定数据。

6.6.2. 数据模型

注释中使用的数据模型遵循特定的惯例。

服务绑定注解必须使用以下约定：

```
service.binding(/<NAME>)?:
    "<VALUE>|(path=<JSONPATH_TEMPLATE>(,objectType=<OBJECT_TYPE>)?(,elementType=
<ELEMENT_TYPE>)?(,sourceKey=<SOURCE_KEY>)?(,sourceValue=<SOURCE_VALUE>)?)"
```

其中：

<NAME>	指定要公开的绑定值的名称。只有在将 **objectType** 参数设置为 **Secret** 或 **ConfigMap** 时，才能将其排除。
<VALUE>	指定没有设置 **path** 时公开的常量值。

数据模型详细介绍了 **路径**、**elementType**、**objectType**、**sourceKey** 和 **sourceValue** 参数允许的值和语义。

表 6.4. 参数及其描述

参数	描述	默认值
path	包含以大括号 {} 括起的 JSONPath 表达式的 jsonpath 模板。	N/A
elementType	指定 **path** 参数中引用的元素值是否满足以下类型之一： • **string** • **sliceOfStrings** • **sliceOfMaps**	**string**
objectType	指定 **path** 参数中指示的元素值是否为当前命名空间中的 **ConfigMap**、**Secret** 或纯文本。	**Secret**，如果 **elementType** 为非字符串。

参数	描述	默认值
sourceKey	指定在收集绑定数据时要添加到绑定 secret 的 **ConfigMap** 或 **Secret** 中的键。 备注： ● 当与 **elementType=sliceOfMaps** 一同使用时，**sourceKey** 参数指定映射中的键，其值为在绑定 secret 中用作键的片段。 ● 使用此可选参数在引用的 **Secret** 或 **ConfigMap** 资源中公开特定条目，作为绑定数据。 ● 如果没有指定，则公开来自 **Secret** 或 **ConfigMap** 资源的所有键和值，并添加到绑定 secret 中。	N/A
sourceValue	指定映射片段中的键。 备注： ● 此键的值用作基础，用于生成添加到绑定 secret 的键值对条目值。 ● 另外，**sourceKey** 的值被用作添加到绑定 secret 的 key-value 对条目的键。 ● 只有在 **elementType=sliceOfMaps** 时才必须。	N/A

注意

只有在 **path** 参数中指示的元素引用 **ConfigMap** 或 **Secret** 资源时，**sourceKey** 和 **sourceValue** 参数才适用。

6.6.3. 将注解映射设置为可选

您可以在注解中带有可选字段。例如，如果服务端点不需要身份验证，则凭证的路径可能不存在。在这种情况下，注解的目标路径中会出现一个字段。因此，Service Binding Operator 默认会生成一个错误。

作为服务提供商，要指明是否需要注解映射，您可以在启用服务时为注解中的 **optional** 标记设置值。只有在目标路径可用时，Service Binding Operator 才会提供注解映射。当目标路径不可用时，Service Binding Operator 会跳过可选映射，并继续进行现有映射的预测，而不会抛出任何错误。

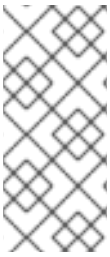
流程

- 要在注解中创建一个字段，将 **optional** 标志值设置为 **true** ：

  Example

```
apiVersion: apps.example.org/v1beta1
kind: Database
```

```
metadata:
  name: my-db
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.spec.name},optional=true
# ...
```



注意

- 如果将 **optional** 标志值设为 **false**，并且 Service Binding Operator 无法找到目标路径，Operator 将无法注解映射。
- 如果 **optional** 标志没有设置值，Service Binding Operator 会将值视为 **false**，并且注解映射会失败。

6.6.4. RBAC 要求

要使用 Service Binding Operator 来公开后备服务绑定数据，您需要特定的基于角色的访问控制(RBAC)权限。在 **ClusterRole** 资源的 **rules** 字段下指定特定的操作动词，以便为后备服务资源授予 RBAC 权限。在定义这些规则时，允许 Service Binding Operator 在整个集群中读取后备服务资源的绑定数据。如果用户没有读取绑定数据或修改应用程序资源的权限，Service Binding Operator 会阻止这样的用户将服务绑定到应用程序。遵循 RBAC 要求避免用户不必要的权限，并防止访问未经授权的服务或应用程序。

Service Binding Operator 使用专用服务帐户对 Kubernetes API 执行请求。默认情况下，此帐户具有将服务绑定到工作负载的权限，它们都由以下标准 Kubernetes 或 OpenShift 对象表示：

- **部署**
- **DaemonSets**
- **ReplicaSet**
- **StatefulSets**
- **DeploymentConfig**

Operator 服务帐户绑定到一个聚合的集群角色，允许 Operator 供应商或集群管理员启用将自定义服务资源绑定到工作负载。要在 **ClusterRole** 中授予所需的权限，请为它标上 **servicebinding.io/controller** 标志，并将标志值设为 **true**。以下示例演示了如何允许 Service Binding Operator **get**、**watch**、**list** Crunchy PostgreSQL Operator 的自定义资源(CR)：

示例：启用到 Crunchy PostgreSQL Operator 置备的 PostgreSQL 数据库实例的绑定

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: postgrescluster-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - postgres-operator.crunchydata.com
  resources:
  - postgresclusters
  verbs:
```

```
    - get
    - watch
    - list
  ...
```

此集群角色可以在安装支持服务 Operator 的过程中部署。

6.6.5. 可公开绑定数据的类别

Service Binding Operator 可让您从后备服务资源和自定义资源定义 (CRD) 中公开绑定数据值。

本节提供了示例，以演示如何使用各种可混合绑定数据类别。您必须修改这些示例，以符合您的工作环境和要求。

6.6.5.1. 从资源公开字符串

以下示例演示了如何将 **PostgresCluster** 自定义资源 (CR) 的 **metadata.name** 字段中的字符串公开为用户名：

Example

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.metadata.name}
# ...
```

6.6.5.2. 将常量值作为绑定项目公开

以下示例演示了如何从 **PostgresCluster** 自定义资源(CR)公开一个常量值：

示例：公开一个常量值

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/type": "postgresql" 1
```

1 绑定 **type** 使用 **postgresql** 值被公开。

6.6.5.3. 公开从资源引用的整个配置映射或 secret

以下示例演示了如何通过注解公开整个 secret：

示例：通过注解公开整个 secret

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
```

示例：从后备服务资源引用的 secret

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
```

6.6.5.4. 从一个配置映射或 secret（从一个资源指代） 中公开一个特定条目

以下示例演示了如何通过注解从配置映射中公开特定条目：

示例：通过注解从配置映射中公开条目

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-config,objectType=ConfigMap,sourceKey=user'
```

示例：从后备服务资源引用的配置映射

绑定数据应具有名称为 **db_timeout** 的键，值为 **10s** ：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

6.6.5.5. 公开资源定义值

以下示例演示了如何通过注解公开资源定义值：

示例：通过注解公开资源定义值

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
```

```
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.metadata.name}
    ...
```

6.6.5.6. 使用每个条目的键和值公开集合条目

以下示例演示了如何通过注解使用每个条目的键和值公开集合条目：

示例：通过注解公开集合条目

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/uri": "path=
{.status.connections},elementType=sliceOfMaps,sourceKey=type,sourceValue=url"
spec:
# ...
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

以下示例演示了注解中集合的先前条目如何投射到绑定应用程序中。

示例：绑定数据文件

```
/bindings/<binding-name>/uri_primary => primary.example.com
/bindings/<binding-name>/uri_secondary => secondary.example.com
/bindings/<binding-name>/uri_404 => black-hole.example.com
```

示例：从后备服务资源配置

```
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

以上示例可帮助您使用键（如 **primary**, **secondary**）将这些值组织为项目。

6.6.5.7. 使用每个项目一个键公开集合的项目

以下示例演示了如何通过注解在各个项目中使用一个键来公开集合项目：

示例：通过注解公开集合项目

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/tags": "path={.spec.tags},elementType=sliceOfStrings"
spec:
    tags:
      - knowledge
      - is
      - power
```

以下示例演示了注解中集合的先前项目如何投射到绑定应用程序中。

示例：绑定数据文件

```
/bindings/<binding-name>/tags_0 => knowledge
/bindings/<binding-name>/tags_1 => is
/bindings/<binding-name>/tags_2 => power
```

示例：从后备服务资源配置

```
spec:
  tags:
  - knowledge
  - is
  - power
```

6.6.5.8. 每个条目使用一个键公开集合条目的值

以下示例演示了如何通过注解使用每个条目值的一个键公开集合条目的值：

示例：通过注解公开集合条目的值

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/url": "path={.spec.connections},elementType=sliceOfStrings,sourceValue=url"
spec:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

-

以下示例演示了注解中集合的先前值如何投射到绑定应用程序中。

示例：绑定数据文件

```
/bindings/<binding-name>/url_0 => primary.example.com
/bindings/<binding-name>/url_1 => secondary.example.com
/bindings/<binding-name>/url_2 => black-hole.example.com
```

6.6.6. 其他资源

- 定义集群服务版本(CSV)
- 项目绑定数据。

6.7. 投射绑定数据

本节提供有关如何使用绑定数据的信息。

6.7.1. 绑定数据的消耗

在后备服务公开绑定数据后，如果工作负载要访问和使用这个数据，您必须将其从后备服务中项目到工作负载中。Service Binding Operator 会自动以以下方法将这一数据集项目到工作负载中：

1. 默认情况下，作为文件。
2. 作为环境变量，在从 **ServiceBinding** 资源配置 **.spec.bindAsFiles** 参数后。

6.7.2. 配置目录路径来项目工作负载容器内绑定数据

默认情况下，Service Binding Operator 将绑定数据作为文件挂载到工作负载资源的特定目录中。您可以使用运行工作负载的容器中的 **SERVICE_BINDING_ROOT** 环境变量设置来配置目录路径。

示例：绑定数据作为文件挂载

```
$SERVICE_BINDING_ROOT 1
├── account-database 2
│   ├── type 3
│   ├── provider 4
│   ├── uri
│   ├── username
│   └── password
└── transaction-event-stream 5
    ├── type
    ├── connection-count
    ├── uri
    ├── certificates
    └── private-key
```

1 根目录。

2 5 存储绑定数据的目录。

3 强制标识符，用于识别投射到对应目录中的绑定数据的类型。

4 可选：标识供应商的标识符，以便应用程序可以识别它可以连接到的后备服务类型。

要将绑定数据用作环境变量，请使用您选择的编程语言的内置语言功能，可以读取环境变量。

示例：Python 客户端使用

```
import os
username = os.getenv("USERNAME")
password = os.getenv("PASSWORD")
```

警告

使用绑定数据目录名称查找绑定数据

Service Binding Operator 使用 **ServiceBinding** 资源名称 (**.metadata.name**) 作为绑定数据目录名称。spec 还提供了一种通过 **.spec.name** 字段覆盖该名称的方法。因此，如果命名空间中有多个 **ServiceBinding** 资源，则绑定数据名称可能会发生冲突。但是，由于 Kubernetes 中卷挂载的性质，绑定数据目录只会包含来自其中一个 **Secret** 资源的值。

6.7.2.1. 计算将绑定数据作为文件投射的最终路径

下表总结了在将文件挂载到特定目录中时如何计算绑定数据投射的最终路径的配置：

表 6.5. 最终路径计算摘要

SERVICE_BINDING_ROOT	最终路径
不可用	**/bindings/<ServiceBinding_ResourceName>**
dir/path/root	**dir/path/root/<ServiceBinding_ResourceName>**

在上表中，**<ServiceBinding_ResourceName>** 条目指定您在自定义资源 (CR) 的 **.metadata.name** 部分中配置的 **ServiceBinding** 资源的名称。

注意

默认情况下，投射文件的权限设置为 0644。Service Binding Operator 无法因为 Kubernetes 中的一个错误而设置特定权限，这会导致在服务需要特定权限（如 **0600**）时出现问题。作为临时解决方案，您可以修改程序代码或正在工作负载资源中运行的应用程序，将文件复制到 **/tmp** 目录中并设置适当的权限。

要在现有 **SERVICE_BINDING_ROOT** 环境变量中访问和使用绑定数据，请使用您选择的编程语言的内置语言功能来读取环境变量。

示例：Python 客户端使用

```
from pyservicebinding import binding
try:
    sb = binding.ServiceBinding()
except binding.ServiceBindingRootMissingError as msg:
    # log the error message and retry/exit
    print("SERVICE_BINDING_ROOT env var not set")
sb = binding.ServiceBinding()
bindings_list = sb.bindings("postgresql")
```

在上例中，**bindings_list** 变量包含 **postgresql** 数据库服务类型的绑定数据。

6.7.3. 投射绑定数据

根据工作负载要求和环境，您可以选择将绑定数据作为文件或环境变量进行项目。

先决条件

- 您了解以下概念：
  - 环境和工作负载要求，以及其如何与所提供的服务配合使用。
  - 在工作负载资源中消耗绑定数据。
  - 配置如何为默认方法计算数据投射的最终路径。
- 绑定数据从后备服务公开。

流程

1. 若要将绑定数据显示为文件，请确保工作负载运行的容器中存在现有的 **SERVICE_BINDING_ROOT** 环境变量，以确定目标文件夹。
2. 要将绑定数据作为环境变量进行项目，请将自定义资源 (CR) 中 **ServiceBinding** 资源中的 **.spec.bindAsFiles** 参数的值设置为 **false**。

6.7.4. 其他资源

- 通过服务开放绑定数据。
- 在应用程序的源代码中使用投射数据。

6.8. 使用 SERVICE BINDING OPERATOR 绑定工作负载

应用程序开发人员必须使用绑定 secret 将工作负载绑定到一个或多个后端服务。生成此 secret 是为了存储工作负载要使用的信息。

例如，假设您要连接的服务已公开绑定数据。在这种情况下，您还需要将工作负载与 **ServiceBinding** 自定义资源(CR)一同使用。通过使用此 **ServiceBinding** CR，工作负载发送带有要绑定的服务详情的绑定请求。

ServiceBinding CR 示例

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
  namespace: my-petclinic
spec:
  services: 1
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
  application: 2
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

1 指定服务资源列表。

2 示例应用程序，指向带有嵌入式 PodSpec 的 Deployment 或任何其他类似资源。

如上例所示，您还可以直接使用 **ConfigMap** 或 **Secret** 本身用作绑定数据源的服务资源。

6.8.1. 命名策略

命名策略仅适用于 **binding.operators.coreos.com** API 组。

命名策略使用 Go 模板来帮助通过服务绑定请求定义自定义绑定名称。命名策略适用于所有属性，包括 **ServiceBinding** 自定义资源(CR)中的映射。

后端服务项目将名称作为文件或环境变量绑定到工作负载。如果工作负载需要特定格式的项目绑定名称，但从后端服务投射绑定名称不能以该格式提供，那么您可以使用命名策略更改绑定名称。

预定义的后处理功能

在使用命名策略时，根据您的工作负载的期望或要求，您可以在任意组合中使用以下预定义的后处理功能来转换字符字符串：

- **大写**：将字符串中的字符转换为大写。
- **小写**：将字符串中的字符转换为小写。
- **标题** ：转字符串中的每个单词的第一个字母大写（某些次要单词除外）。

预定义的命名策略

根据以下预定义的命名策略，处理通过注解声明的绑定名称来更改工作负载：

- **none** ：应用时，绑定名称没有任何更改。

  Example

  在模板编译后，绑定名称采用 **{{ .name }}** 形式。

  ```
  host: hippo-pgbouncer
  port: 5432
  ```

-

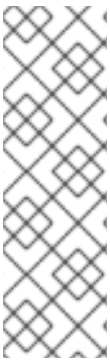
- **upper**: 在没有定义 **namingStrategy** 时应用。应用时，将绑定名称的字符串转换为大写。

  Example

  模板编译后，绑定名称采用 **{{ .service.kind | upper}}_{{ .name | upper }}** 格式。

  ```
  DATABASE_HOST: hippo-pgbouncer
  DATABASE_PORT: 5432
  ```

  如果您的工作负载需要不同的格式，您可以定义自定义命名策略并使用前缀和分隔符更改绑定名称，如 **PORT_DATABASE**。



注意

- 当绑定名称作为文件进行投射时，默认情况下会应用预定义的 **none** 命名策略，绑定名称不会改变。
- 当绑定名称作为环境变量进行投射且没有定义 **namingStrategy** 时，会默认应用预定义的 **uppercase** 命名策略。
- 您可以使用自定义绑定名称和预定义的后处理函数定义自定义命名策略来覆盖预定义的命名策略。

6.8.2. 高级绑定选项

您可以定义 **ServiceBinding** 自定义资源 (CR) 以使用以下高级绑定选项：

- 更改绑定名称：此选项仅适用于 **binding.operators.coreos.com** API 组。
- 编写自定义绑定数据：此选项仅适用于 **binding.operators.coreos.com** API 组。
- 使用标签选择器绑定工作负载：此选项可用于 **binding.operators.coreos.com** 和 **servicebinding.io** API 组。

6.8.2.1. 在将绑定名称改到工作负载前更改绑定名称

您可以指定规则来更改 **ServiceBinding** CR 的 **.spec.namingStrategy** 属性中的绑定名称。例如，假设一个连接到 PostgreSQL 数据库的 Spring PetClinic 示例应用程序。在本例中，PostgreSQL 数据库服务公开数据库的 **host** 和 **port** 字段，以用于绑定。Spring PetClinic 示例应用程序可以通过绑定名称访问此公开绑定数据。

示例： ServiceBinding CR 中的 Spring PetClinic 示例应用程序

```
# ...
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
# ...
```

示例： ServiceBinding CR 中的 PostgreSQL 数据库服务

```
# ...
    services:
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster
      name: hippo
# ...
```

如果未定义 **namingStrategy** 且绑定名称作为环境变量，则后备服务中的 **host: hippo-pgbouncer** 值，并且投射环境变量将如以下示例所示：

Example

```
DATABASE_HOST: hippo-pgbouncer
```

其中：

DATABASE	指定 **kind** 后端服务。
HOST	指定绑定名称。

应用 **POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV** 命名策略后，服务绑定请求准备的自定义绑定名称列表如下所示：

Example

```
POSTGRESQL_DATABASE_HOST_ENV: hippo-pgbouncer
POSTGRESQL_DATABASE_PORT_ENV: 5432
```

以下项目描述了 **POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV** 命名策略中定义的表达式：

- **.name**: 请参阅由支持服务公开的绑定名称。在上例中，绑定名称为 **HOST** 和 **PORT**。
- **.service.kind** ：请参阅其绑定名称通过命名策略更改的服务资源类型。
- **upper** ：用于在编译 Go 模板字符串时处理字符字符串的字符串的字符串。
- **POSTGRESQL** ：自定义绑定名称的前缀。
- **ENV** ：自定义绑定名称的修复。

与前面的示例类似，您可以在 **namingStrategy** 中定义字符串模板，以定义如何由服务绑定请求准备绑定名称的每个键。

6.8.2.2. 编写自定义绑定数据

作为应用程序开发人员，您可以在以下情况下编写自定义绑定数据：

- 后备服务不公开绑定数据。
- 所公开的值不能以所需格式提供，工作负载符合预期。

例如，如果后备服务 CR 将主机、端口和数据库用户公开为绑定数据，但工作负载要求将绑定数据用作连接字符串。您可以使用代表支持服务的 Kubernetes 资源中的属性编写自定义绑定数据。

Example

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
   namespace: my-petclinic
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo (1)
     id: postgresDB (2)
   - group: ""
     version: v1
     kind: Secret
     name: hippo-pguser-hippo
     id: postgresSecret
   application:
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
   mappings:
     ## From the database service
     - name: JDBC_URL
       value: 'jdbc:postgresql://{{ .postgresDB.metadata.annotations.proxy }}:{{ .postgresDB.spec.port }}/{{ .postgresDB.metadata.name }}'
     ## From both the services!
     - name: CREDENTIALS
       value: '{{ .postgresDB.metadata.name }}{{ translationService.postgresSecret.data.password }}'
     ## Generate JSON
     - name: DB_JSON (3)
       value: {{ json .postgresDB.status }} (4)
```

(1) 后端服务资源的名称。

(2) 可选标识符。

(3) Service Binding Operator 生成的 JSON 名称。Service Binding Operator 会将此 JSON 名称作为文件或环境变量的名称。

(4) Service Binding Operator 生成的 JSON 值。Service Binding Operator 将此 JSON 值作为文件或环境变量。JSON 值包含来自后备服务自定义资源的指定字段中的属性。

6.8.2.3. 使用标签选择器绑定工作负载

您可以使用标签选择器指定要绑定的工作负载。如果您使用标签选择器声明服务绑定来获取工作负载，Service Binding Operator 会定期尝试查找和绑定与给定标签选择器匹配的新工作负载。

例如，作为集群管理员，您可以通过在 **ServiceBinding** CR 中设置适当的 **labelSelector** 字段来将服务绑定到带有 **environment: production** 标签的命名空间中的所有 **Deployment**。这可让 Service Binding Operator 将每个工作负载与一个 **ServiceBinding** CR 绑定。

binding.operators.coreos.com/v1alpha1 API 中的 ServiceBinding CR 示例

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: multi-application-binding
  namespace: service-binding-demo
spec:
  application:
    labelSelector: 1
      matchLabels:
        environment: production
    group: apps
    version: v1
    resource: deployments
  services:
    group: ""
    version: v1
    kind: Secret
    name: super-secret-data
```

1 指定正在绑定的工作负载。

servicebinding.io API 中的 ServiceBinding CR 示例

```
apiVersion: servicebindings.io/v1beta1
kind: ServiceBinding
metadata:
  name: multi-application-binding
  namespace: service-binding-demo
spec:
  workload:
    selector: 1
      matchLabels:
        environment: production
    apiVersion: app/v1
    kind: Deployment
  service:
    apiVersion: v1
    kind: Secret
    name: super-secret-data
```

1 指定正在绑定的工作负载。

重要

如果定义以下字段对，Service Binding Operator 会拒绝绑定操作，并生成错误：

- **binding.operators.coreos.com/v1alpha1** API 中的 **name** 和 **labelSelector** 字段。
- **servicebinding.io** API (Spec API) 中的 **name** 和 **selector** 字段。

了解重新绑定（rebinding）行为

例如，在成功绑定后，您可以使用 **name** 字段识别工作负载。如果删除并重新创建那个工作负载，**ServiceBinding** 协调器不会重新绑定工作负载，Operator 无法将绑定数据的项目到工作负载。但是，如果您使用 **labelSelector** 字段识别工作负载，**ServiceBinding** 协调器会重新绑定工作负载，Operator 则项目绑定数据。

6.8.3. 绑定与 PodSpec 不兼容的二级工作负载

服务绑定中的典型场景涉及配置后端服务、工作负载(Deployment)和 Service Binding Operator。考虑涉及与 PodSpec 不兼容且位于主工作负载(Deployment)和 Service Binding Operator 之间的辅助工作负载（也可以是一个应用程序 Operator）的场景。

对于这样的辅助工作负载资源，容器路径的位置是任意的。对于服务绑定，如果 CR 中的辅助工作负载与 PodSpec 不兼容，您必须指定容器路径的位置。这样做可以将数据绑定到 **ServiceBinding** 自定义资源(CR)的二级工作负载中指定的容器路径中，例如，当您不想在一个 pod 中绑定数据时。

在 Service Binding Operator 中，您可以配置容器或 secret 驻留在工作负载中的路径，并在自定义位置绑定这些路径。

6.8.3.1. 配置容器路径的自定义位置

当 Service Binding Operator 项目作为环境变量时，这个自定义位置可用于 **binding.operators.coreos.com** API 组。

考虑辅助工作负载 CR，它不与 PodSpec 兼容，并且具有位于 **spec.containers** 路径的容器：

示例：二级工作负载 CR

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  containers:
  - name: hello-world
    image: quay.io/baijum/secondary-workload:latest
    ports:
    - containerPort: 8080
```

流程

- 通过在 **ServiceBinding** CR 中指定值并将此路径绑定到 **spec.application.bindingPath.containersPath** 自定义位置来配置 **spec.containers** 路径：

  示例：带有自定义位置的 spec.containers 路径的 ServiceBinding CR

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo
     id: postgresDB
   - group: ""
     version: v1
     kind: Secret
     name: hippo-pguser-hippo
     id: postgresSecret
   application: 1
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
   application: 2
     name: secondary-workload
     group: operator.sbo.com
     version: v1
     resource: secondaryworkloads
     bindingPath:
       containersPath: spec.containers 3
```

1 示例应用程序，指向带有嵌入式 PodSpec 的 Deployment 或任何其他类似资源。

2 辅助工作负载，不与 PodSpec 兼容。

3 容器路径的自定义位置。

指定容器路径的位置后，Service Binding Operator 会生成绑定数据，该数据在 **ServiceBinding** CR 的二级工作负载中指定的容器路径中可用。

以下示例显示了带有 **envFrom** 和 **secretRef** 字段的 **spec.containers** 路径：

示例：带有 envFrom 和 secretRef 字段的二级工作负载 CR

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
   name: secondary-workload
spec:
   containers:
   - env: 1
     - name: ServiceBindingOperatorChangeTriggerEnvVar
       value: "31793"
     envFrom:
     - secretRef:
         name: secret-resource-name 2
```

```
    image: quay.io/baijum/secondary-workload:latest
    name: hello-world
    ports:
    - containerPort: 8080
    resources: {}
```

1. 具有 Service Binding Operator 生成的值的唯一容器数组。这些值基于后端服务 CR。
2. Service Binding Operator 生成的 **Secret** 资源的名称。

6.8.3.2. 配置 secret 路径的自定义位置

当 Service Binding Operator 项目作为环境变量时，这个自定义位置可用于 **binding.operators.coreos.com** API 组。

考虑与 PodSpec 不兼容的辅助工作负载 CR，且只有 **spec.secret** 路径中的 secret：

示例：二级工作负载 CR

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  secret: ""
```

流程

- 通过在 **ServiceBinding** CR 中指定值并在 **spec.application.bindingPath.secretPath** 自定义位置上绑定这个路径来配置 **spec.secret** 路径：

  示例：带有自定义位置 spec.secret 路径的 ServiceBinding CR

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
...
  application: (1)
    name: secondary-workload
    group: operator.sbo.com
    version: v1
    resource: secondaryworkloads
    bindingPath:
      secretPath: spec.secret (2)
...
```

  1. 辅助工作负载，不与 PodSpec 兼容。
  2. 包含 **Secret** 资源名称的 secret 路径的自定义位置。

指定 secret 路径的位置后，Service Binding Operator 会生成绑定数据，该数据在 **ServiceBinding** CR 的二级工作负载中指定的 secret 路径中可用。

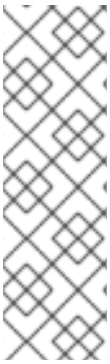
以下示例显示了带有 **binding-request** 值的 **spec.secret** 路径：

示例：使用 binding-request 值进行二级工作负载 CR

```
...
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  secret: binding-request-72ddc0c540ab3a290e138726940591debf14c581 (1)
...
```

(1) Service Binding Operator 生成的 **Secret** 资源的唯一名称。

6.8.3.3. 工作负载资源映射



注意

- 工作负载资源映射可用于 API groups: **binding.operators.coreos.com** 和 **servicebinding.io** 的 **ServiceBinding** 自定义资源 (CR) 的辅助工作负载。
- 您必须仅在 **servicebinding.io** API 组下定义 **ClusterWorkloadResourceMapping** 资源。但是，**ClusterWorkloadResourceMapping** 资源与 **binding.operators.coreos.com** 和 **servicebinding.io** API 组中的 **ServiceBinding** 资源交互。

如果无法使用配置方法配置容器路径，则无法配置自定义路径位置，您可以精确定义需要投射绑定数据的位置。通过在 **servicebinding.io** API 组中定义 **ClusterWorkloadResourceMapping** 资源，指定给定工作负载类型的绑定数据的位置。

以下示例演示了如何为 **CronJob.batch/v1** 资源定义映射。

示例： CronJob.batch/v1 资源的映射

```
apiVersion: servicebinding.io/v1beta1
kind: ClusterWorkloadResourceMapping
metadata:
 name: cronjobs.batch (1)
spec:
  versions:
  - version: "v1" (2)
    annotations: .spec.jobTemplate.spec.template.metadata.annotations (3)
    containers:
    - path: .spec.jobTemplate.spec.template.spec.containers[*] (4)
    - path: .spec.jobTemplate.spec.template.spec.initContainers[*]
      name: .name (5)
```

```
    env: .env 6
    volumeMounts: .volumeMounts 7
  volumes: .spec.jobTemplate.spec.template.spec.volumes 8
```

1. **ClusterWorkloadResourceMapping** 资源的名称，它必须符合映射负载资源的 **plural.group**。
2. 正在映射的资源版本。未指定的任何版本都可以与 "*" 通配符匹配。
3. 可选：一个 pod 中的 **.annotations** 字段标识符，使用固定 JSONPath 指定。默认值为 **.spec.template.spec.annotations**。
4. pod 中的 **.containers** 和 **.initContainers** 字段的标识符，使用 JSONPath 指定。如果没有定义 **containers** 字段下的条目，Service Binding Operator 默认为两个路径：**.spec.template.spec.containers[*]** 和 **.spec.template.spec.initContainers[*]**，所有其他字段都设为默认值。但是，如果您指定了条目，则必须定义 **.path** 字段。
5. 可选：容器中的 **.name** 字段的标识符，使用固定 JSONPath 指定。默认值为 **.name**。
6. 可选：容器中的 **.env** 字段的标识符，使用固定 JSONPath 指定。默认值为 **.env**。
7. 可选：容器中的 **.volumeMounts** 字段的标识符，使用固定 JSONPath 指定。默认值为 **.volumeMounts**。
8. 可选：一个 pod 中的 **.volumes** 字段的标识符，使用固定 JSONPath 指定。默认值为 **.spec.template.spec.volumes**。

重要

- 在这个上下文中，固定 JSONPath 是 JSONPath grammar 的子集，它只接受以下操作：
  - 字段查找：**.spec.template**
  - 数组索引：**.spec['template']**

  所有其他操作都不接受。
- 大多数字段都是可选的。如果没有指定，Service Binding Operator 会假定与 **PodSpec** 资源兼容。
- Service Binding Operator 要求每个字段都结构化地与 pod 部署中的对应字段相同。例如，工作负载资源中的 **.env** 字段的内容必须能够接受 Pod 资源中的 **.env** 字段相同的数据结构。否则，将绑定数据绑定到这样的工作负载可能会导致 Service Binding Operator 意外行为。

特定于 binding.operators.coreos.com API 组的行为

当 **ClusterWorkloadResourceMapping** 资源与 **binding.operators.coreos.com** API 组中的 **ServiceBinding** 资源交互时，您可以预期以下行为：

- 如果将带有 **bindAsFiles: false** 标志值的 **ServiceBinding** 资源与其中一个映射一同创建，那么环境变量将投射到对应 **ClusterWorkloadResourceMapping** 资源中指定的每个 **path** 字段下的 **.envFrom** 字段。

- 作为集群管理员，您可以在 **ServiceBinding.bindings.coreos.com** 资源中指定 **ClusterWorkloadResourceMapping** 资源和 **.spec.application.bindingPath.containersPath** 字段。
  Service Binding Operator 会尝试将数据绑定到 **ClusterWorkloadResourceMapping** 资源和 **.spec.application.bindingPath.containersPath** 字段中指定的位置。这个行为等同于在对应的 **ClusterWorkloadResourceMapping** 资源中添加带有 **path: $containersPath** 属性的容器条目，所有其他值都取取其默认值。

6.8.4. 从后备服务中取消绑定工作负载

您可以使用 **oc** 工具从后端服务中取消绑定工作负载。

- 要从后备服务中取消绑定工作负载，请删除链接到该服务的 **ServiceBinding** 自定义资源(CR)：

```
$ oc delete ServiceBinding <.metadata.name>
```

Example

```
$ oc delete ServiceBinding spring-petclinic-pgcluster
```

其中：

spring-petclinic-pgcluster	指定 **ServiceBinding** CR 的名称。

6.8.5. 其他资源

- 将工作负载与支持服务绑定。
- 将 Spring PetClinic 示例应用程序连接到 PostgreSQL 数据库服务。
- 通过文件创建自定义资源
- ClusterWorkloadResourceMapping 资源的示例模式.

6.9. 使用 DEVELOPER 视角将应用程序连接到服务

使用 **Topology** 视图用于以下目的：

- 对应用程序中的多个组件进行分组。
- 相互连接组件。
- 使用标签将多个资源连接到服务。

您可以使用绑定或可视连接器来连接组件。

只有当目标节点是 Operator 支持的服务时，才可以在组件之间建立绑定连接。为了表示这种情况，当您将箭头拖到这样的目标节点上时，会出现 **Create a binding connector** 工具提示。当应用程序使用绑定连接器连接到服务时，会创建一个 **ServiceBinding** 资源。然后，Service Binding Operator 控制器会将必要的绑定数据项目到应用程序部署中。请求成功后，会重新部署应用程序以在连接的组件间建立交互。

视觉连接器只在组件之间建立视觉连接，描述连接意图。没有建立组件之间的交互。如果目标节点不是一个 Operator 支持的服务，当您将箭头拖到目标节点上时，将会显示 **Create a visual connector** 工具提示。

6.9.1. 发现并识别 Operator 支持的可绑定服务

作为用户，如果要创建可绑定的服务，您必须知道哪些服务可以绑定。可绑定服务是应用程序可轻松使用的服务，因为它们以标准的方式公开其绑定数据，如凭证、连接详情、卷挂载、secret 和其他绑定数据。**Developer** 视角可帮助您发现和识别此类可绑定的服务。

流程

- 要发现并确定由 Operator 支持的可绑定服务，请考虑以下替代方法：
  - 点 **+Add → Developer Catalog → Operator Backed** 查看 Operator 支持的标题。支持服务绑定功能的 Operator 支持的服务在标题上具有 **可绑定**的徽标。
  - 在 **Operator Backed** 页面左侧窗格中，选择 **Bindable**。

    提示

    点 **Service binding** 旁边的 help 图标查看与可绑定服务相关的信息。

  - 点 **+Add → Add** 并搜索 Operator 支持的服务。当您点可绑定服务时，您可以在侧面面板中查看 **Bindable** 徽标。

6.9.2. 在组件之间创建视觉连接

您可以使用可视连接器来描述连接应用程序组件的意图。

此流程介绍了在 PostgreSQL 数据库服务和 Spring PetClinic 示例应用程序之间创建可视连接的示例。

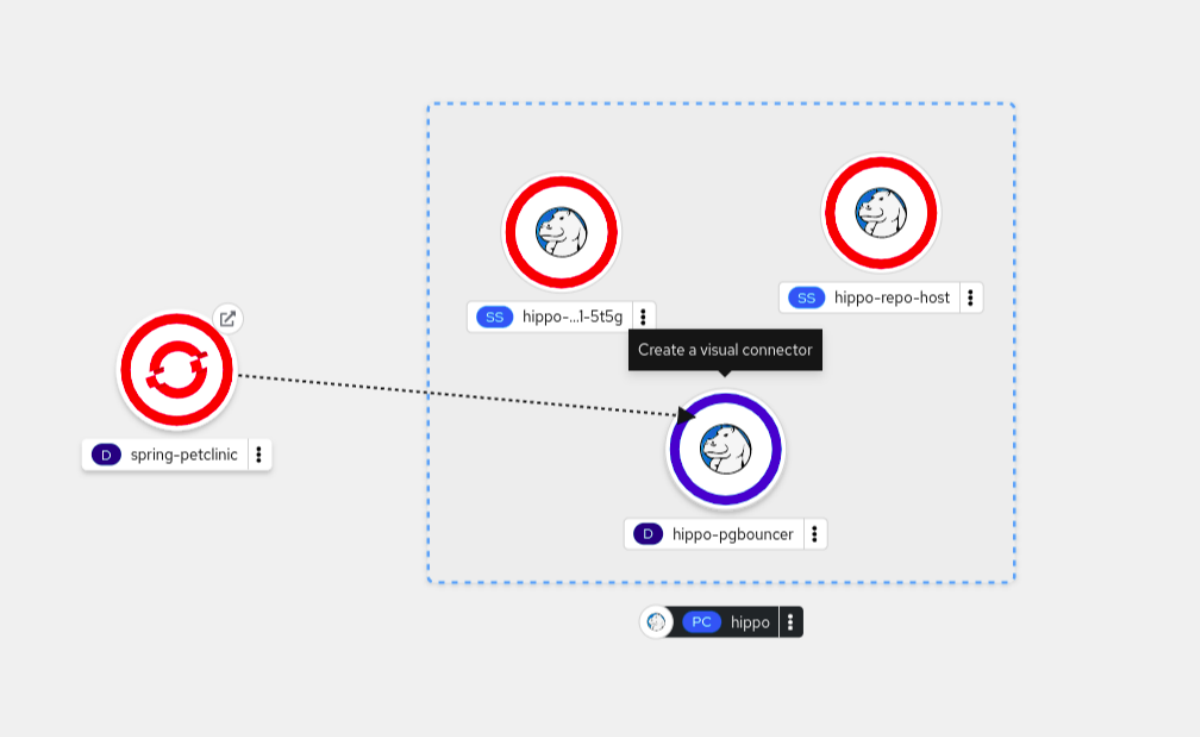
先决条件

- 您已使用 **Developer** 视角创建并部署了 Spring PetClinic 示例应用程序。
- 已使用 **Developer** 视角创建并部署了 Crunchy PostgreSQL 数据库实例。此实例具有以下组件：**hippo-backup**、**hippo-instance**、**hippo-repo-host** 和 **hippo-pgbouncer**。

步骤

1. 在 **Developer** 视角中，切换到相关项目，如 **my-petclinic**。
2. 将鼠标悬停在 Spring PetClinic 示例应用上，以查看节点上的悬挂箭头。

图 6.2. 视觉连接器

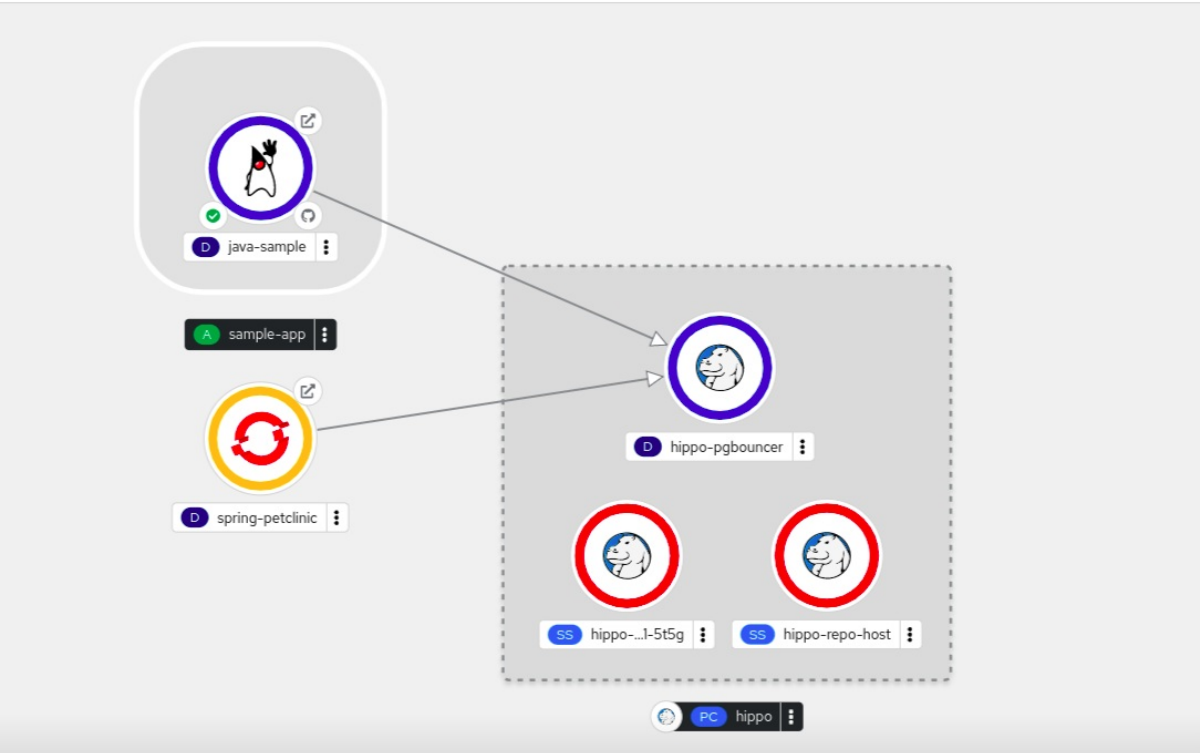


3. 单击箭头并将它拖向 **hippo-pgbouncer** 部署，以将 Spring PetClinic 示例应用与其连接。

4. 点 **spring-petclinic** 部署来查看 Overview 面板。在 Details 选项卡下，点 Annotations 部分中的编辑图标，查看 Key = **app.openshift.io/connects-to** 和 Value = **[{"apiVersion":"apps/v1","kind":"Deployment","name":"hippo-pgbouncer"}]** 注解添加到部署。

5. 可选：您可以重复这些步骤，以在其他应用程序和组件之间建立视觉连接。

图 6.3. 连接多个应用程序



6.9.3. 在组件之间创建绑定连接

您可以创建一个与 Operator 支持的组件的绑定连接，如下例所示，它使用 PostgreSQL Database 服务和 Spring PetClinic 示例应用程序。要创建与 PostgreSQL Database Operator 支持的服务的绑定连接，您必须首先将红帽提供的 PostgreSQL Database Operator 添加到 **OperatorHub**，然后安装 Operator。然后，PostSQL Database Operator 会创建和管理 Database 资源，这会在 secret、配置映射、状态和 spec 属性中公开绑定数据。

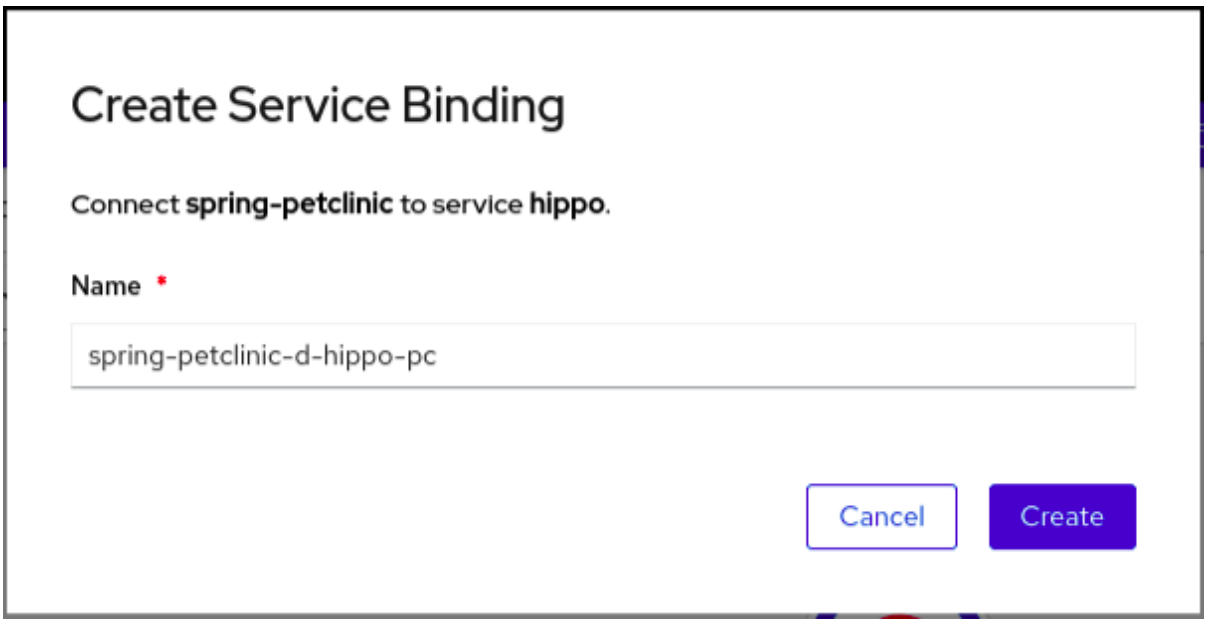
先决条件

- 您在 **Developer** 视角中创建并部署了 Spring PetClinic 示例应用程序。
- 您从 **OperatorHub** 安装 Service Binding Operator。
- 您已使用 **v5 Update** 频道从 OperatorHub 安装了 **Crunchy Postgres for Kubernetes** Operator。
- 您在 **Developer** 视角中创建了一个 **PostgresCluster** 资源，这会导致一个含有以下组件的 Crunchy PostgreSQL 数据库实例：**hippo-backup**、**hippo-instance**、**hippo-repo-host**、**hippo-pgbouncer**。

流程

1. 在 **Developer** 视角中，切换到相关项目，如 **my-petclinic**。
2. 在 **Topology** 视图中，把鼠标移到 Spring PetClinic 示例应用程序上，以查看节点上的悬挂箭头。
3. 将箭头拖放到 Postgres Cluster 的 **hippo** 数据库图标，以使用 Spring PetClinic 示例应用程序进行绑定连接。
4. 在 **Create Service Binding**对话框中，保留默认名称或为服务绑定添加其他名称，然后点 **Create**。

   图 6.4. 服务绑定对话框

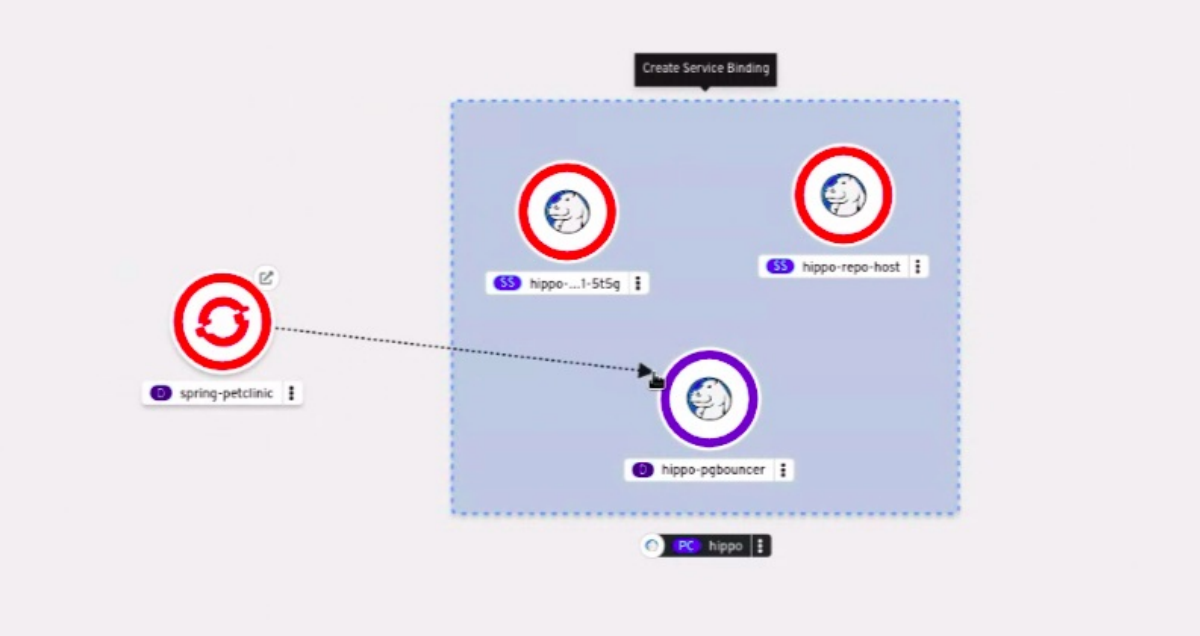


5. 可选：如果使用 Topology 视图进行绑定连接，进入 **+Add → YAML → Import YAML**。
6. 可选：在 YAML 编辑器中，添加 **ServiceBinding** 资源：

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
   namespace: my-petclinic
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo
   application:
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
```

服务绑定请求被创建，并通过 **ServiceBinding** 资源创建绑定连接。当数据库服务连接请求成功后，会重新部署应用程序并建立连接。

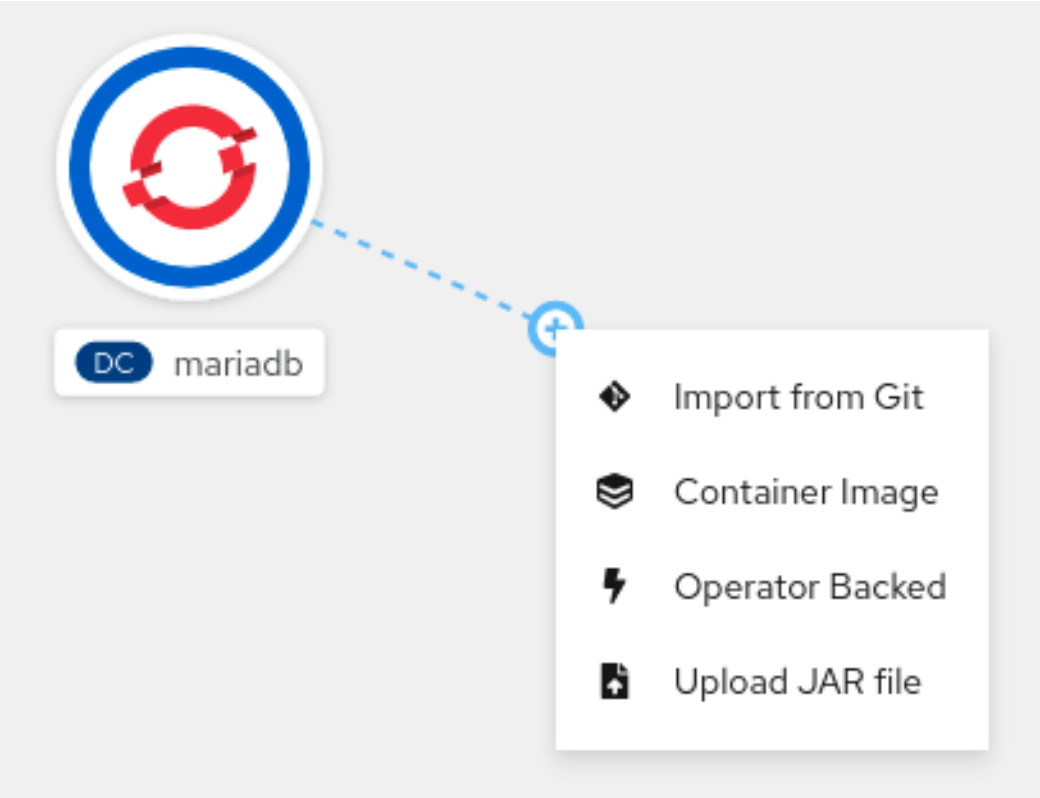
图 6.5. 绑定连接器



提示

您还可以通过拖动悬挂箭头以添加和创建与 Operator 支持的服务的绑定连接来使用上下文菜单。

图 6.6. 创建绑定连接的上下文菜单



7. 在导航菜单中点 **Topology**。Topology 视图中的 spring-petclinic 部署包括 Open URL 链接来查看其网页。

8. 点 **Open URL** 链接。

现在，您可以远程查看 Spring PetClinic 示例应用程序，以确认应用程序现在连接到数据库服务，并且数据已成功投入 Crunchy PostgreSQL 数据库服务中的应用程序。

Service Binding Operator 在应用程序和数据库服务之间成功创建了可正常工作的连接。

6.9.4. 从 Topology 视图验证服务绑定的状态

Developer 视角可帮助您通过 **Topology** 视图验证服务绑定的状态。

流程

1. 如果服务绑定成功，点绑定连接器。侧面板会出现在 **Details** 标签页中显示 **Connected** 状态。另外，您可以从 **Developer** 视角在以下页面中查看 **Connected** 状态：

   - **ServiceBindings** 页面。
   - **ServiceBinding 详情**页面。另外，页面标题显示 **Connected** 图标。

2. 如果服务绑定失败，绑定连接器会显示红色的箭头，并在连接中间有一个红线。点这个连接器，在 **Details** 选项卡的侧面板中查看 **Error** 状态。（可选）点 **Error** 状态查看有关底层问题的特定信息。
   您还可以从 **Developer** 视角查看以下页面中的错误状态和提示信息：

- **ServiceBindings** 页面。
- **ServiceBinding 详情**页面。另外，页面标题会显示**错误**徽标。

提示

在 **ServiceBindings** 页面中，使用 **Filter** 下拉菜单根据服务的状态列出服务绑定。

6.9.5. 其他资源

- 服务绑定入门
- 已知的可绑定 Operator

第 7 章 使用 HELM CHART

7.1. 了解 HELM

Helm 是一个软件包管理程序，它简化了应用程序和服务部署到 OpenShift Container Platform 集群的过程。

Helm 使用名为 *charts* 的打包格式。Helm chart 是描述 OpenShift Container Platform 资源的一个文件集合。

在集群中运行的一个 chart 实例被称为 *release*。当每次一个 chart 在集群中安装时，一个新的 release 会被创建。

在每次安装 chart，或一个版本被升级或回滚时，都会创建增量修订版本。

7.1.1. 主要特性

Helm 提供以下功能：

- 搜索存储在 chart 存储库中的一个大型 chart 集合。
- 修改现有 chart。
- 使用 OpenShift Container Platform 或 Kubernetes 资源创建自己的 chart。
- 将应用程序打包为 chart 并共享。

7.1.2. 红帽 OpenShift Helm chart 认证

您可以选择由红帽为要在 Red Hat OpenShift Container Platform 上部署的所有组件验证并认证 Helm chart。图表采用自动化红帽 OpenShift 认证工作流，确保安全合规，以及与平台的最佳集成和经验。认证可确保 chart 的完整性，并确保 Helm Chart 在 Red Hat OpenShift 集群上无缝工作。

7.1.3. 其他资源

- 有关如何将 Helm chart 认证为红帽合作伙伴的更多信息，请参阅 Red Hat Certification of Helm charts for OpenShift。
- 如需有关红帽合作伙伴 OpenShift 和容器认证指南的更多信息，请参阅 OpenShift 和容器认证合作伙伴指南。
- 如需 chart 列表，请参阅 Red Hat **Helm index** 文件。
- 您可以在 Red Hat Marketplace 查看可用图表。如需更多信息，请参阅使用 Red Hat Marketplace。

7.2. 安装 HELM

下面的部分论述了如何使用 CLI 在不同的平台中安装 Helm。

在 OpenShift Container Platform web 控制台中，点右上角的 ? 图标并选 **Command Line Tools**。

先决条件

- 已安装了 Go 版本 1.13 或更高版本。

7.2.1. 对于 Linux

1. 下载 Helm 二进制文件并将其添加到您的路径中：
   - Linux (x86_64, amd64)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-amd64 -o /usr/local/bin/helm
     ```

   - Linux on IBM Z 和 IBM® LinuxONE (s390x)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-s390x -o /usr/local/bin/helm
     ```

   - Linux on IBM Power (ppc64le)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-ppc64le -o /usr/local/bin/helm
     ```

2. 使二进制文件可执行：

   ```
   # chmod +x /usr/local/bin/helm
   ```

3. 检查已安装的版本：

   ```
   $ helm version
   ```

   输出示例

   ```
   version.BuildInfo{Version:"v3.0",
   GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean",
   GoVersion:"go1.13.4"}
   ```

7.2.2. 对于 Windows 7/8

1. 下载最新的 **.exe** 文件并放入您自己选择的目录 。
2. 右键点击 **Start** 并点击 **Control Panel**。
3. 选择 **系统和安全性** ， 然后点击 **系统**。
4. 在左侧的菜单中选择**高级系统设置**并点击底部的**环境变量**按钮。
5. 在变**量**部分选择**路径**并点**编辑** 。
6. 点**新建**并输入到 **.exe** 文件的路径， 或者点击 **浏览** 并选择目录， 然后点 **确定**。

7.2.3. 对于 Windows 10

1. 下载最新的 **.exe** 文件并放入您自己选择的目录 。

2. 点击 **搜索** 并输入 **env** 或者 **environment**。
3. 选择为**您的帐户**编辑**环境变量**。
4. 在**变量**部分选择**路径**并点**编辑** 。
5. 点**新建**并输入到 exe 文件所在目录的路径，或者点击 **浏览** 并选择目录，然后点击**确定**。

7.2.4. 对于 macOS

1. 下载 Helm 二进制文件并将其添加到您的路径中：

```
# curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-darwin-amd64
-o /usr/local/bin/helm
```

2. 使二进制文件可执行：

```
# chmod +x /usr/local/bin/helm
```

3. 检查已安装的版本：

```
$ helm version
```

输出示例

```
version.BuildInfo{Version:"v3.0",
GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean",
GoVersion:"go1.13.4"}
```

7.3. 配置自定义 HELM CHART 仓库

您可以使用以下方法在 OpenShift Container Platform 集群上安装 Helm chart：

- CLI。
- Web 控制台的 **Developer** 视角。

在 web 控制台的 **Developer** 视角中， **Developer Catalog** 显示集群中可用的 Helm chart。默认情况下，它会从 Red Hat OpenShift Helm Chart 仓库中列出 Helm chart。如需 chart 列表，请参阅 Red Hat **Helm index** 文件。

作为集群管理员，您可以添加多个集群范围的 Helm Chart 仓库，与默认的集群范围 Helm 仓库分开，并在 **Developer Catalog** 中显示这些仓库中的 Helm chart。

作为具有适当基于角色的访问控制(RBAC)权限的普通用户或项目成员，您可以添加多个命名空间范围的 Helm Chart 仓库，除了默认的集群范围的 Helm 仓库，并在 **Developer Catalog** 中显示这些仓库中的 Helm chart。

在 web 控制台的 **Developer** 视角中，您可以使用 **Helm** 页面：

- 使用 **Create** 按钮创建 Helm Releases 和 Repositories。
- 创建、更新或删除集群范围的 Helm Chart 仓库。

- 在 Repositories 选项卡中查看现有 Helm Chart 仓库列表，它也可以作为集群范围或命名空间范围轻松区分。

7.3.1. 在 OpenShift Container Platform 集群中安装 Helm chart

先决条件

- 您有一个正在运行的 OpenShift Container Platform 集群，并已登录该集群。
- 您已安装 Helm。

流程

1. 创建一个新项目：

```
$ oc new-project vault
```

2. 将一个 Helm chart 存储库添加到本地 Helm 客户端：

```
$ helm repo add openshift-helm-charts https://charts.openshift.io/
```

输出示例

```
"openshift-helm-charts" has been added to your repositories
```

3. 更新存储库：

```
$ helm repo update
```

4. 安装 HashiCorp Vault 示例：

```
$ helm install example-vault openshift-helm-charts/hashicorp-vault
```

输出示例

```
NAME: example-vault
LAST DEPLOYED: Fri Mar 11 12:02:12 2022
NAMESPACE: vault
STATUS: deployed
REVISION: 1
NOTES:
Thank you for installing HashiCorp Vault!
```

5. 验证 chart 是否已成功安装：

```
$ helm list
```

输出示例

```
NAME            NAMESPACE REVISION UPDATED                                  STATUS   CHART
APP VERSION
```

```
example-vault vault     1       2022-03-11 12:02:12.296226673 +0530 IST deployed vault-
0.19.0 1.9.2
```

7.3.2. 使用 Developer 视角安装 Helm chart

您可以使用 web 控制台中的 **Developer** 视角或 CLI 从 **Developer Catalog** 中列出的 Helm chart 中选择并安装 chart。您可以通过安装 Helm chart 来创建 Helm 发行版本，并在 web 控制台的 **Developer** 视角中查看它们。

先决条件

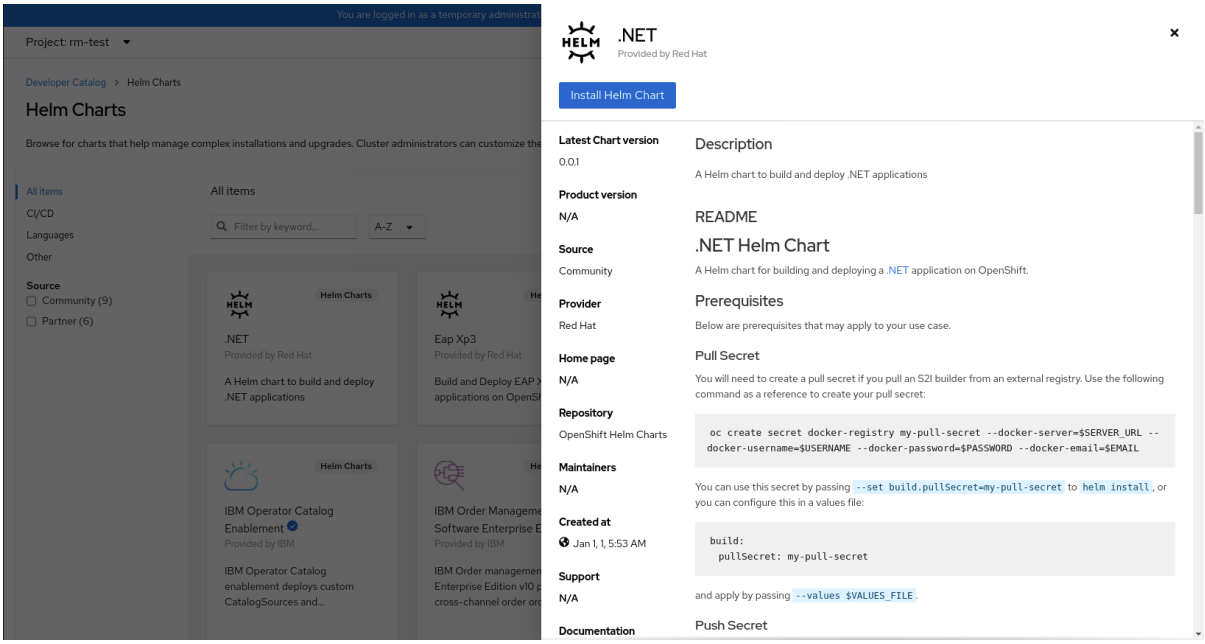
- 已登陆到 web 控制台并切换到 **Developer** 视角。

流程

通过 **Developer Catalog** 提供的 Helm chart 来创建 Helm 发行版本：

1. 在 **Developer** 视角中，进入 **+Add** 视图并选择一个项目。然后点击 **Helm Chart** 选项来查看 **Developer Catalog** 中的所有 Helm Chart。
2. 选择一个 chart，查看它的描述信息、README 和其他与之相关的信息。
3. 点 **Install Helm Chart**。

图 7.1. Developer Catalog 中的 Helm chart



4. 在 **Install Helm Chart** 页面中：
   a. 在 **Release Name** 项中输入 release 的唯一名称。
   b. 从 **Chart Version** 下拉列表中选择所需的 chart 版本。
   c. 使用 **Form View** 或 **YAML View** 配置 Helm Chart。

注意

在可用情况下，您可以在 **YAML View** 和 **Form View** 间切换。在不同视图间切换时数据会被保留。

d. 点击 **Install** 创建 Helm release。**Topology** 视图将会显示，其中包括了发行版本。如果 Helm chart 带有发行注记，则 chart 会被预先选择，右侧面板会显示该发行版本的发行注记。

e. 在 **Helm Releases** 页面中查看新创建的 Helm 发行版本。

您可以使用侧面面板上的 **Actions** 按钮或右键点击 Helm 发行版本来升级、回滚或卸载 Helm 发行版本。

7.3.3. 在 web 终端中使用 Helm

您可以通过在 web 控制台的 **Developer** 视角中访问 web 终端 来使用 Helm。

7.3.4. 在 OpenShift Container Platform 上创建自定义 Helm chart

流程

1. 创建一个新项目：

```
$ oc new-project nodejs-ex-k
```

2. 下载包含 OpenShift Container Platform 对象的示例 Node.js chart：

```
$ git clone https://github.com/redhat-developer/redhat-helm-charts
```

3. 进入包含 chart 示例的目录：

```
$ cd redhat-helm-charts/alpha/nodejs-ex-k/
```

4. 编辑 **Chart.yaml** 文件并添加 chart 描述：

```
apiVersion: v2 (1)
name: nodejs-ex-k (2)
description: A Helm chart for OpenShift (3)
icon: https://static.redhat.com/libs/redhat/brand-assets/latest/corp/logo.svg (4)
version: 0.2.1 (5)
```

(1) Chart API 版本。对于至少需要 Helm 3 的 Helm Chart，它应该是 **v2**。

(2) chart 的名称。

(3) chart 的描述。

(4) 用作图标的图形的 URL。

(5) 根据 Semantic Versioning(SemVer)2.0.0 规范，您的 chart 的版本。

5. 验证 chart 格式是否正确：

```
$ helm lint
```

输出示例

```
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

6. 前往上一个目录级别：

```
$ cd ..
```

7. 安装 chart：

```
$ helm install nodejs-chart nodejs-ex-k
```

8. 验证 chart 是否已成功安装：

```
$ helm list
```

输出示例

```
NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION
nodejs-chart nodejs-ex-k 1 2019-12-05 15:06:51.379134163 -0500 EST deployed nodejs-0.1.0  1.16.0
```

7.3.5. 添加自定义 Helm Chart 仓库

作为集群管理员，您可以将自定义 Helm Chart 存储库添加到集群中，并在 Developer Catalog 中启用从这些仓库中获得 Helm chart 的访问权限。

流程

1. 要添加新的 Helm Chart 仓库，您必须将 Helm Chart 仓库自定义资源（CR）添加到集群中。

Helm Chart 仓库 CR 示例

```
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <name>
spec:
 # optional name that might be used by console
 # name: <chart-display-name>
  connectionConfig:
    url: <helm-chart-repository-url>
```

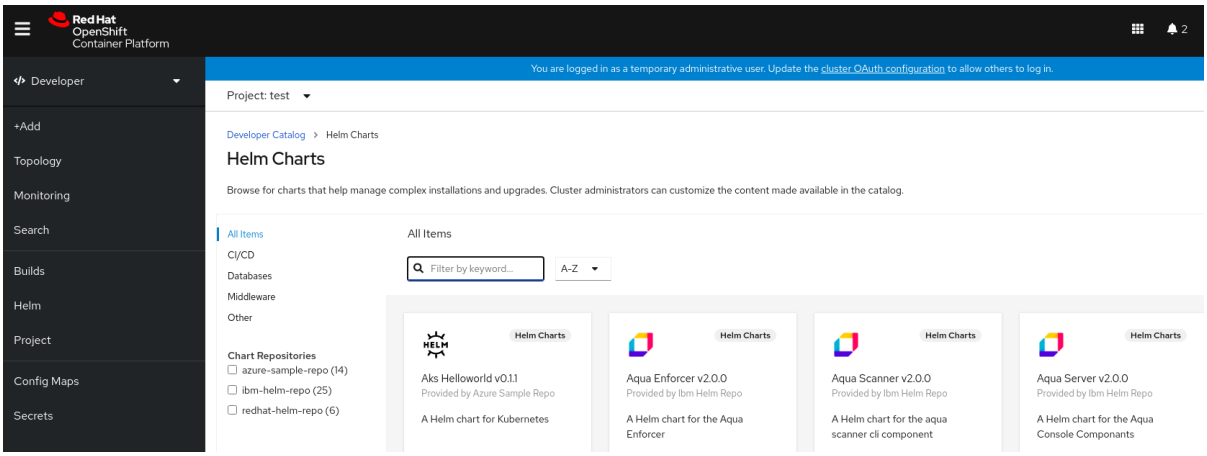
例如，要添加 Azure 示例 chart 存储库，请运行：

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  name: azure-sample-repo
```

```
  connectionConfig:
    url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
EOF
```

2. 导航到 web 控制台中的 **Developer Catalog**，以验证是否已显示 Helm chart 存储库中的 Helm chart。
例如，使用 **Chart 仓库** 过滤器从仓库搜索 Helm chart。

图 7.2. Chart 软件仓库过滤器



注意

如果集群管理员删除了所有 chart 仓库，则无法在 **+Add** 视图、**Developer Catalog** 和左面的导航面板中查看 Helm 选项。

7.3.6. 添加命名空间范围的自定义 Helm Chart 仓库

Helm 仓库的集群范围的 **HelmChartRepository** 自定义资源定义(CRD)可帮助管理员将 Helm 仓库添加为自定义资源。命名空间范围的 **ProjectHelmChartRepository** CRD 允许具有适当基于角色的访问控制(RBAC)权限的项目成员创建其所选 Helm 仓库资源，但仅限于其命名空间。此项目成员可从集群范围和命名空间范围 Helm 仓库资源中看到 chart。

注意

- 管理员可以限制用户创建命名空间范围的 Helm 仓库资源。通过限制用户，管理员具有通过命名空间角色而不是集群角色来控制 RBAC 的灵活性。这可避免用户不必要的权限，并防止访问未经授权的服务或应用程序。
- 添加命名空间范围的 Helm 仓库不会影响现有集群范围的 Helm 仓库的行为。

作为具有适当 RBAC 权限的普通用户或项目成员，您可以在集群中添加自定义命名空间范围的 Helm Chart 仓库，并在 **Developer Catalog** 中启用这些仓库中的 Helm chart 访问 Helm chart。

流程

1. 要添加新的命名空间范围的 Helm Chart 仓库，您必须将 Helm Chart 仓库自定义资源(CR)添加到命名空间中。

命名空间范围的 Helm Chart 仓库 CR 示例

```
apiVersion: helm.openshift.io/v1beta1
```

```
kind: ProjectHelmChartRepository
metadata:
  name: <name>
spec:
  url: https://my.chart-repo.org/stable

  # optional name that might be used by console
  name: <chart-repo-display-name>

  # optional and only needed for UI purposes
  description: <My private chart repo>

  # required: chart repository URL
  connectionConfig:
    url: <helm-chart-repository-url>
```

例如，要将 Azure 示例 chart 存储库添加到 **my-namespace** 命名空间，请运行：

```
$ cat <<EOF | oc apply --namespace my-namespace -f -
apiVersion: helm.openshift.io/v1beta1
kind: ProjectHelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  name: azure-sample-repo
  connectionConfig:
    url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
EOF
```

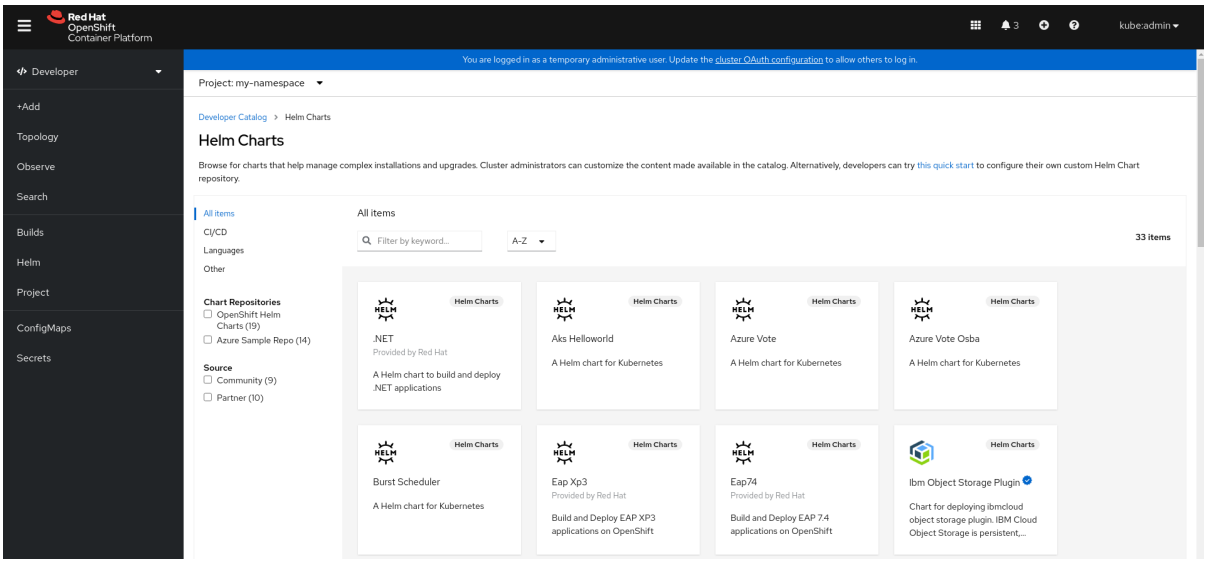
检查输出来验证是否创建了命名空间范围的 Helm Chart Repository CR：

输出示例

```
projecthelmchartrepository.helm.openshift.io/azure-sample-repo created
```

2. 导航到 web 控制台中的 **Developer Catalog**，以验证 **my-namespace** 命名空间中是否显示了来自 Chart 仓库的 Helm chart。
例如，使用 **Chart 仓库** 过滤器从仓库搜索 Helm chart。

图 7.3. 命名空间中的 Chart 软件仓库过滤器

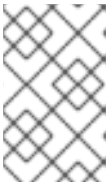


或者，运行：

```
$ oc get projecthelmchartrepositories --namespace my-namespace
```

输出示例

```
NAME                    AGE
azure-sample-repo       1m
```



注意

如果集群管理员或具有适当 RBAC 权限的常规用户删除特定命名空间中的所有 chart 存储库，那么您无法查看 **+Add** 视图、**Developer Catalog** 以及用于该特定命名空间的左侧导航面板中的 Helm 选项。

7.3.7. 创建凭证和 CA 证书以添加 Helm Chart 仓库

有些 Helm Chart 仓库需要凭证和自定义证书颁发机构（CA）证书才能与其连接。您可以使用 Web 控制台和 CLI 添加凭证和证书。

流程

配置凭证和证书，然后使用 CLI 添加 Helm Chart 仓库：

1. 在 **openshift-config** 命名空间中，使用 PEM 编码格式的自定义 CA 证书创建一个 **configmap**，并将它存储在配置映射中的 **ca-bundle.crt** 键下：

```
$ oc create configmap helm-ca-cert \
--from-file=ca-bundle.crt=/path/to/certs/ca.crt \
-n openshift-config
```

2. 在 **openshift-config** 命名空间中，创建一个 **Secret** 对象来添加客户端 TLS 配置：

```
$ oc create secret tls helm-tls-configs \
--cert=/path/to/certs/client.crt \
--key=/path/to/certs/client.key \
```

```
-n openshift-config
```

请注意：客户端证书和密钥必须采用 PEM 编码格式，并分别保存在 **tls.crt** 和 **tls.key** 密钥中。

3. 按如下所示添加 Helm 仓库：

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <helm-repository>
spec:
  name: <helm-repository>
  connectionConfig:
    url: <URL for the Helm repository>
    tlsConfig:
        name: helm-tls-configs
    ca:
 name: helm-ca-cert
EOF
```

ConfigMap 和 **Secret** 使用 **tlsConfig** 和 **ca** 字段在 HelmChartRepository CR 中消耗。这些证书用于连接 Helm 仓库 URL。

4. 默认情况下，所有经过身份验证的用户都可以访问所有配置的 chart。但是，对于需要证书的 Chart 仓库，您必须为用户提供对 **openshift-config** 命名空间中 **helm-ca-cert** 配置映射和 **helm-tls-configs** secret 的读取访问权限，如下所示：

```
$ cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["helm-ca-cert"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["helm-tls-configs"]
  verbs: ["get"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
subjects:
  - kind: Group
    apiGroup: rbac.authorization.k8s.io
    name: 'system:authenticated'
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: Role
  name: helm-chartrepos-tls-conf-viewer
EOF
```

7.3.8. 根据它们的认证级别过滤 Helm Chart

您可以在 **Developer Catalog** 中根据它们的认证级别过滤 Helm chart。

流程

1. 在 **Developer** 视角中，进入 **+Add** 视图并选择一个项目。
2. 在 **Developer Catalog** 标题中，选择 **Helm Chart** 选项来查看 **Developer Catalog** 中的所有 Helm chart。
3. 使用 Helm chart 列表左侧的过滤器过滤所需的 chart:
   - 使用 **Chart Repositories** 过滤器过滤由 **Red Hat Certification Charts** 或 **OpenShift Helm Charts** 提供的 chart。
   - 使用 **Source** 过滤器过滤来自 **合作伙伴**、 **社区** 或 **红帽** 的 chart。认证图表显示为( )图标。

注意

如果只有一个供应商类型，则 **Source** 过滤器不可见。

现在，您可以选择所需的 chart 并安装它。

7.3.9. 禁用 Helm Chart 仓库

您可以通过将 **HelmChartRepository** 自定义资源中的 **disabled** 属性设置为 **true**，从目录中的特定 Helm Chart 仓库禁用 Helm Charts。

流程

- 要通过 CLI 禁用 Helm Chart 仓库，将 **disabled: true** 标志添加到自定义资源中。例如，要删除 Azure 示例 chart 存储库，请运行：

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  connectionConfig:
   url:https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
  disabled: true
EOF
```

- 使用 Web 控制台禁用最近添加的 Helm Chart 仓库：
  1. 进入 **自定义资源定义** 并搜索 **HelmChartRepository** 自定义资源。

2. 进入 **实例**，找到您要禁用的存储库，并点击其名称。
3. 进入 **YAML** 选项卡，在 **spec** 部分添加 **disabled: true** 标志，点 **Save**。

Example

```
spec:
  connectionConfig:
    url: <url-of-the-repositoru-to-be-disabled>
  disabled: true
```

现在，这个仓库已被禁用，并不会出现在目录中。

7.4. 使用 HELM 发行版本

您可以使用 web 控制台中的 **Developer** 视角来更新、回滚或卸载 Helm 发行版本。

7.4.1. 先决条件

- 已登陆到 web 控制台并切换到 **Developer** 视角。

7.4.2. 升级 Helm 发行版本

您可以把一个 Helm 发行版本升级到新的 chart 版本，或更发行版本的配置。

流程

1. 在 **Topology** 视图中，选择 Helm 发行本来查看侧面面板。
2. 点 **Actions → Upgrade Helm Release**。
3. 在 **Upgrade Helm Release** 页面中，选择您要升级到的 **Chart Version**，然后点 **Upgrade** 以创建另一个 Helm 发行版本。**Helm Releases** 页面会显示两个修订版本。

7.4.3. 回滚 Helm 发行版本

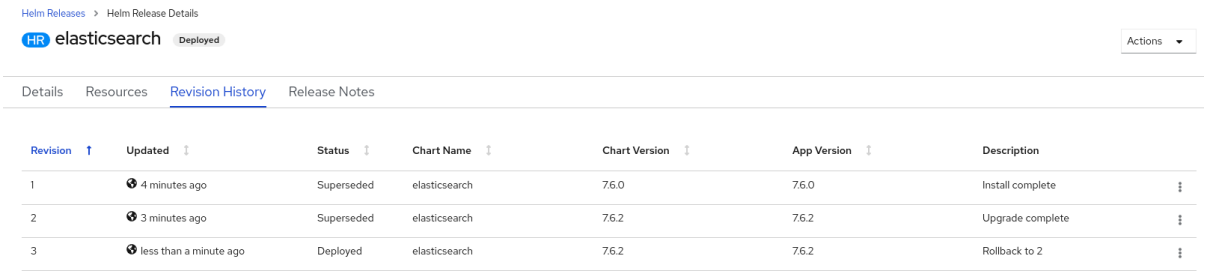
如果一个发行版本有问题，您可以将 Helm 发行版本恢复到上一个版本。

流程

使用 **Helm** 视图回滚发行版本：

1. 在 **Developer** 视角中，导航到 **Helm** 视图以查看命名空间中的 **Helm Release** 版本。
2. 点击列出的发行版本  旁边的 **Options** 菜单，然后选择 **Rollback**。
3. 在 **Rollback Helm Release** 页中，选择要回滚到的 **Revision**，点 **Rollback**。
4. 在 **Helm Releases** 页面中，点 chart 查看该发行版本的详情和资源。
5. 进入 **Revision History** 标签页来查看这个 chart 的所有修订版本。

图 7.4. Helm 修改历史记录



6. 如果需要，您可以进一步使用一个特定修订版本旁的 **Options** 选项 并选择回滚到的修订版本。

7.4.4. 卸载 Helm 发行版本

流程

1. 在 **Topology** 视图中，右键点击 Helm 发行版本并选择 **Uninstall Helm Release**。
2. 在确认提示中，输入 chart 的名称并点击 **Uninstall**。

第 8 章 部署

8.1. 了解 DEPLOYMENT 和 DEPLOYMENTCONFIG 对象

OpenShift Container Platform 中的 **Deployment** 和 **DeploymentConfig** API 对象提供了两个类似但不同的方法来对常见用户应用程序进行精细管理。由以下独立 API 对象组成：

- 一个 **Deployment** 或 **DeploymentConfig** 对象，用于将应用程序特定组件的所需状态描述为 pod 模板。
- **Deployment** 对象涉及一个或多个 *replica sets（复制集）*，其中包含部署状态的一个时间点的记录，作为 pod 模板。同样，**DeploymentConfig** 对象涉及一个或多个 *replication controllers（复制控制器）*，它在副本集之前。
- 一个或多个 pod，表应用程序某一特定版本的实例。

使用 **Deployment** 对象，除非需要由 **DeploymentConfig** 对象提供的特定功能或行为。

8.1.1. 部署构建块

Deployment 和部署配置分别通过使用原生 Kubernetes API 对象 **ReplicaSet** 和 **ReplicationController** 来启用，作为构建块。

用户不必操作由 **Deployment** 或 **DeploymentConfig** 对象拥有的副本集、复制控制器或 pod。部署系统可确保正确传播更改。

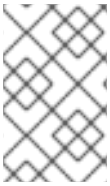
提示

如果现有部署策略不适用于您的用例，而且必须在部署的生命周期内执行手动步骤，那么应考虑创建自定义部署策略。

以下部分详细介绍了这些对象。

8.1.1.1. 副本集（Replica set）

ReplicaSet 是一个原生 Kubernetes API 对象，可以确保在任意给定时间运行指定数量的 Pod 副本。



注意

只有您需要自定义更新编配，或根本不需要更新时，才使用副本集。否则，使用部署。副本集可以独立使用，但由部署使用用来编配 pod 创建、删除和更新。部署会自动管理其副本集，为 pod 提供声明性更新，且不需要手动管理它们创建的副本集。

以下是 **ReplicaSet** 定义示例：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
```

```
selector: 1
  matchLabels: 2
    tier: frontend
  matchExpressions: 3
    - {key: tier, operator: In, values: [frontend]}
template:
  metadata:
    labels:
      tier: frontend
  spec:
    containers:
    - image: openshift/hello-openshift
      name: helloworld
      ports:
      - containerPort: 8080
        protocol: TCP
    restartPolicy: Always
```

1. 对一组资源进行的标签查询。**matchLabels** 和 **matchExpressions** 的结果在逻辑上是组合在一起的。
2. 基于相等的选择器，使用与选择器匹配的标签指定资源。
3. 基于集合的选择器，用于过滤键。这将选择键等于 **tier** 并且值等于 **frontend** 的所有资源。

8.1.1.2. 复制控制器

与副本集类似，复制控制器确保始终运行指定数量的 pod 副本。如果 pod 退出或被删除，复制控制器会做出反应，实例化更多 pod 来达到定义的数量。同样，如果运行中的数量超过所需的数目，它会根据需要删除相应数量的 Pod，使其与定义的数量相符。副本集与复制控制器之间的区别在于，副本集支持基于集合的选择器要求，而复制控制器只支持基于相等的选择器要求。

复制控制器配置包括：

- 需要的副本数量，可在运行时调整。
- 创建复制 **Pod** 时要使用的 Pod 定义。
- 用于标识受管 pod 的选择器。

选择器是分配给由复制控制器管理的 pod 的一组标签。这些标签包含在复制控制器实例化的 **Pod** 定义中。复制控制器使用选择器来决定已在运行的 pod 实例数量，以便根据需要进行调整。

复制控制器不会基于负载或流量执行自动扩展，因为复制控制器不会跟踪它们。相反，这需要由外部自动缩放器调整其副本数。

注意

使用 **DeploymentConfig** 创建复制控制器，而不是直接创建复制控制器。

如果您需要自定义编配或不需要更新，请使用副本集而不是复制控制器。

以下是复制控制器的示例定义：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1 (1)
  selector: (2)
    name: frontend
  template: (3)
    metadata:
      labels: (4)
        name: frontend (5)
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1. 要运行的 pod 的副本数。
2. 要运行的 pod 的标签选择器。
3. 控制器创建的 pod 模板。
4. pod 上的标签应该包括标签选择器中的标签。
5. 扩展任何参数后的最大名称长度为 63 个字符。

8.1.2. 部署

Kubernetes 在 OpenShift Container Platform 中提供了一流的原生 API 对象类型，名为 **Deployment**。**Deployment** 对象用于描述应用程序特定组件的所需状态作为一个 pod 模板。Deployment 创建副本集，用于编配 pod 生命周期。

例如，以下部署定义会创建一个副本集来启动一个 **hello-openshift** pod:

Deployment 定义

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-openshift
  template:
    metadata:
      labels:
        app: hello-openshift
```

```
spec:
  containers:
  - name: hello-openshift
    image: openshift/hello-openshift:latest
    ports:
    - containerPort: 80
```

8.1.3. DeploymentConfig 对象

在复制控制器的基础上，OpenShift Container Platform 增加了对软件开发和部署生命周期的支持，及 **DeploymentConfig** 对象的概念。在最简单的情形中，**DeploymentConfig** 对象会创建一个新的复制控制器，并允许它启动 pod。

但是，从 **DeploymentConfig** 对象部署的 OpenShift Container Platform 也支持从镜像的现有部署过渡到新部署，同时还可以定义在创建复制控制器之前或之后运行的 hook。

DeploymentConfig 部署系统提供以下功能：

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

在创建 **DeploymentConfig** 对象时，会创建一个复制控制器来代表 **DeploymentConfig** 对象的 pod 模板。如果部署被改变，则会使用最新的 pod 模板创建一个新的复制控制器，并运行部署过程来缩减旧复制控制器并扩展新的复制控制器。

在创建时，自动从服务负载均衡器和路由器中添加和移除应用程序的实例。只要应用程序支持接收 **TERM** 信号时安全关机，您可以确保运行的用户连接拥有正常完成的机会。

OpenShift Container Platform **DeploymentConfig** 对象定义以下详细信息：

1. **ReplicationController** 定义的元素。
2. 自动创建新部署的触发器。
3. 在部署之间过渡的策略。
4. 生命周期 hook。

每次触发部署时，无论是手动还是自动，部署器 Pod 均管理部署（包括缩减旧复制控制器、扩展新复制控制器以及运行 hook）。部署 pod 在完成部署后会无限期保留，以便保留其部署日志。当部署被另一个部署替换时，以前的复制控制器会被保留，以便在需要时轻松回滚。

DeploymentConfig 定义示例

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
```

```
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
  - type: ConfigChange (1)
  - imageChangeParams:
      automatic: true
      containerNames:
      - helloworld
      from:
        kind: ImageStreamTag
        name: hello-openshift:latest
    type: ImageChange (2)
  strategy:
    type: Rolling (3)
```

1. 配置更改触发器会在部署配置的 pod 模板中检测到更改时生成新的复制控制器。
2. 镜像更改触发器会在命名镜像流中每次有新版本的后备镜像可用时创建新部署。
3. 默认的 **Rolling** 策略会在部署之间实现无停机过渡。

8.1.4. Deployment 和 DeploymentConfig 对象的比较

OpenShift Container Platform 支持 Kubernetes **Deployment** 对象和 OpenShift Container Platform 提供的 **DeploymentConfig** 对象，但建议您使用 **Deployment** 对象，除非您需要 **DeploymentConfig** 对象提供的特定功能或行为。

以下部分详细阐述两种对象之间的区别，以进一步协助您决定使用哪一种类型。

8.1.4.1. 设计

Deployment 和 **DeploymentConfig** 对象之间的一个重要区别是为推出（rollout）过程所选择的 CAP theorem 属性。**DeploymentConfig** 对象以一致性为先，而 **Deployments** 对象优先于可用性。

对于 **DeploymentConfig** 对象，如果运行一个部署器 pod 的节点停机，它不会被替换掉。流程会等待节点重新在线或被手动删除。手动删除节点也会删除对应的 pod。这意味着您无法删除 pod 来取消推出部署，因为 kubelet 负责删除相关联的 pod。

但是，部署推出由控制器管理器驱动。控制器管理器在 master 上运行高可用性模式，并使用群首选举算法提高可用性与一致性相比的价值。在故障期间，其他 master 有可能同时对同一部署做出反应，但这个问题会在故障发生后很快进行调节。

8.1.4.2. 针对部署的功能

滚动
Deployment 的部署过程是由控制器循环推动的，这与使用部署器 Pod 进行每次新推出部署的 **DeploymentConfig** 相反。这意味着 **Deployment** 对象可以拥有尽可能多的活跃副本集，最终部署控制器将缩减所有旧副本集，并扩展最新的副本集。

DeploymentConfig 对象最多可以有一个部署器 pod 运行，否则多个部署器在试图扩展其认为是最新的复制控制器时会导致冲突。因此，任何时间点上只能有两个复制控制器处于活跃状态。最终，这可以更快地为 **Deployment** 对象推出部署。

按比例扩展
因为部署控制器是适合由 **Deployment** 对象拥有的新和旧副本集的大小的唯一来源，所以它能够扩展持续推出部署。额外副本会根据每个副本集的大小按比例分发。

当推出（rollout）进行的过程中，**DeploymentConfig** 对象无法被扩展，因为控制器会遇到部署器进程中有关新 ReplicationController 大小的问题。

中途暂停推出部署
Deployment 可以在任何时间暂停，这意味着可以暂停正在进行的推出部署。但是，当前还无法暂停部署器 Pod。因此，如果您尝试在推出部署进行期间暂停部署，则部署器进程不受影响，它会继续运行直到完成为止。

8.1.4.3. deploymentConfig 对象相关的功能

自动回滚
目前，在出现故障时，部署不支持自动回滚到上次成功部署的副本集。

触发器
部署有一个隐式配置更改触发器，每次更改部署的 Pod 模板都会自动触发新的推出部署。如果您不想在 Pod 模板更改时进行新的推出部署，请暂停部署：

```
$ oc rollout pause deployments/<name>
```

生命周期 hook
Deployment 尚不支持任何生命周期 hook。

自定义策略
部署不支持用户指定的自定义部署策略。

8.2. 管理部署过程

8.2.1. 管理 DeploymentConfig 对象

DeploymentConfig 对象可以通过 OpenShift Container Platform Web 控制台的 **Workloads** 页面或使用 **oc** CLI 管理。以下流程演示了 CLI 的用法（除非另有说明）。

8.2.1.1. 启动部署

您可以启动一个部署推出（rollout）来开始应用程序的部署过程。

流程

1. 要从现有的 **DeploymentConfig** 对象启动新的部署过程，请运行以下命令：

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

注意

如果部署过程已在进行中，此命令会显示一条消息，不会部署新的复制控制器。

8.2.1.2. 查看部署

您可以查看部署来获取应用程序所有可用修订的基本信息。

流程

1. 要显示所有最近为提供的 **DeploymentConfig** 对象创建的复制控制器的详细信息，包括任何当前运行的部署过程，请运行以下命令：

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

2. 要查看修订的相关细节，请使用 **--revision** 标志：

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

3. 如需有关 **DeploymentConfig** 对象和最新修订的详细信息，请使用 **oc describe** 命令：

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

8.2.1.3. 重试部署

如果 **DeploymentConfig** 对象的当前修订无法部署，您可以重启部署过程。

流程

1. 重启已经失败的部署过程：

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

如果成功部署了最新的修订，该命令会显示一条消息，且不会重试部署过程。

注意

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

8.2.1.4. 回滚部署

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

流程

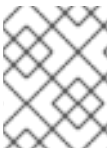
1. 回滚到配置的最近一次部署成功的修订：

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

恢复 **DeploymentConfig** 对象的模板以匹配 undo 命令中指定的部署修订，并且会启动新的复制控制器。如果没有通过 **--to-revision** 指定修订，则使用最近一次成功部署的修订。

2. 部署过程中会禁用 **DeploymentConfig** 对象中的镜像更改触发器，以防止在回滚完成不久后意外启动新的部署过程。
重新启用镜像更改触发器：

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



注意

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

8.2.1.5. 在容器内执行命令

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

流程

1. 在 **DeploymentConfig** 对象的 **spec** 字段中添加 **command** 参数。您也可以添加 **args** 字段来修改 **command**（如果 **command** 不存在，则修改 **ENTRYPOINT**）。

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
  template:
# ...
    spec:
     containers:
     - name: <container_name>
       image: 'image'
       command:
         - '<command>'
       args:
         - '<argument_1>'
         - '<argument_2>'
         - '<argument_3>'
```

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

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
  template:
# ...
    spec:
      containers:
        - name: example-spring-boot
          image: 'image'
          command:
            - java
          args:
```

```
            - '-jar'
            - /opt/app-root/springboots2idemo.jar
# ...
```

8.2.1.6. 查看部署日志

流程

1. 输出给定 **DeploymentConfig** 对象的最新修订的日志：

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

如果最新的修订正在运行或已失败，命令会返回负责部署 Pod 的进程的日志。如果成功，它将从应用程序的 pod 返回日志。

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

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

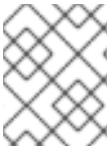
8.2.1.7. 部署触发器

DeploymentConfig 对象可以包含触发器，推动创建新部署过程以响应集群中的事件。

警告

如果 **DeploymentConfig** 对象上没有定义任何触发器，则默认添加配置更改触发器。如果触发器定义为空白字段，则必须手动启动部署。

配置更改部署触发器
当在 **DeploymentConfig** 对象的 pod 模板中发现配置改变时，配置更改触发器会生成一个新的复制控制器。



注意

如果在 **DeploymentConfig** 对象上定义了配置更改触发器，则在 **DeploymentConfig** 对象本身创建后会自动创建第一个复制控制器,且不会暂停。

配置更改部署触发器

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
```

```
# ...
  triggers:
    - type: "ConfigChange"
```

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

镜像更改部署触发器

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  triggers:
    - type: "ImageChange"
      imageChangeParams:
        automatic: true (1)
        from:
          kind: "ImageStreamTag"
          name: "origin-ruby-sample:latest"
          namespace: "myproject"
        containerNames:
          - "helloworld"
```

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

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

注意

如果在 **DeploymentConfig** 对象上定义了镜像更改触发器（带有配置更改触发器和 **automatic=false**，或 **automatic=true**）并且镜像更改触发器指向的镜像流标签尚不存在，则初始部署过程将在镜像导入时或由构建推送到镜像流标签时立即自动开始。

8.2.1.7.1. 设置部署触发器

流程

1. 您可以使用 **oc set triggers** 命令为 **DeploymentConfig** 对象设置部署触发器。例如，要设置镜像更改触发器，请使用以下命令：

```
$ oc set triggers dc/<dc_name> \
    --from-image=<project>/<image>:<tag> -c <container_name>
```

8.2.1.8. 设置部署资源

部署由节点上消耗资源（内存、CPU 和临时存储）的 pod 完成。默认情况下，pod 消耗无限的节点资源。但是，如果某个项目指定了默认容器限值，则 pod 消耗的资源会被限制在这些限值范围内。

注意

部署的最小内存限值为 12MB。如果容器因为一个 **Cannot allocate memory** pod 事件启动失败，这代表内存限制太低。增加或删除内存限制。删除限制可让 pod 消耗无限的节点资源。

您还可以在部署策略中指定资源限值来限制资源使用。部署资源可用于 recreate、rolling 或 custom 部署策略。

流程

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

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  type: "Recreate"
  resources:
    limits:
      cpu: "100m" 1
      memory: "256Mi" 2
      ephemeral-storage: "1Gi" 3
```

1. **cpu** 以 CPU 单元数为单位：**100m** 表示 0.1 个 CPU 单元（100 * 1e-3）。
2. **memory** 以字节为单位：**256Mi** 表示 268435456 字节 (256 * 2 ^ 20)。
3. **ephemeral-storage** 以字节为单位：**1Gi** 表示 1073741824 字节 (2 ^ 30)。

不过，如果您的项目定义了配额，则需要以下两项之一：

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

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  type: "Recreate"
  resources:
    requests: 1
      cpu: "100m"
      memory: "256Mi"
      ephemeral-storage: "1Gi"
```

1. **requests** 对象包含与配额中资源列表对应的资源列表。

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

要设置部署资源，请选择以上选项之一。否则，部署 pod 创建失败，显示无法满足配额要求。

其他资源

- 如需有关资源限值和请求的更多信息，请参阅了解管理应用程序内存。

8.2.1.9. 手动扩展

除了回滚外，您还可以通过手动缩放来对副本数量进行细致的控制。

注意

也可以使用 **oc autoscale** 命令自动扩展 Pod。

流程

1. 要手动扩展 **DeploymentConfig** 对象，请使用 **oc scale** 命令。例如，以下命令将 **frontend DeploymentConfig** 对象中的副本设置为 **3**。

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

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

8.2.1.10. 从 DeploymentConfig 对象访问私有存储库

您可以在 **DeploymentConfig** 对象中添加 secret，以便它可以从私有存储库访问镜像。此流程演示了 OpenShift Container Platform Web 控制台方法。

流程

1. 创建一个新项目。
2. 导航到 **Workloads → Secrets**。
3. 创建一个包含用于访问私有镜像存储库的凭证的 secret。
4. 进入到 **Workloads → DeploymentConfigs**。
5. 创建 **DeploymentConfig** 对象。
6. 在 **DeploymentConfig** 对象编辑器页面中，设置 **Pull Secret** 并保存您的更改。

8.2.1.11. 将 Pod 分配给特定的节点

您可以结合使用节点选择器和带标签的节点来控制 pod 的放置。

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

流程

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

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
# ...
spec:
  nodeSelector:
    disktype: ssd
# ...
```

具有节点选择器时创建的 Pod 会分配给带有指定标签的节点。这里指定的标签将与集群管理员添加的标签结合使用。

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

注意

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

8.2.1.12. 使用其他服务帐户运行 pod

您可以使用非默认服务帐户运行 pod。

流程

1. 编辑 **DeploymentConfig** 对象：

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

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

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: example-dc
# ...
spec:
# ...
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

8.3. 使用部署策略

*部署策略*用于在不停机的情况下更改或升级应用程序，以便用户不会注意到更改。

由于用户通常通过由路由器处理的路由访问应用程序，因此部署策略侧重于 **DeploymentConfig** 对象功能或路由功能。侧重于 **DeploymentConfig** 对象的策略会影响所有使用应用程序的路由。侧重于路由功能的策略则会影响到单个的路由。

多数部署策略通过 **DeploymentConfig** 对象支持，一些额外的策略则通过路由器功能支持。

8.3.1. 选择部署策略

选择部署策略时请考虑以下几点：

- 长时间运行的连接必须被恰当处理。
- 数据库转换可能比较复杂，且必须和应用程序一同执行并回滚。
- 如果应用程序由微服务和传统组件构成，则可能需要停机才能完成转换。
- 您必须拥有进行此操作的基础架构。
- 如果您的测试环境没有被隔离，则可能会破坏到新版本和旧版本。

部署策略使用就绪度检查来确定新 pod 是否准备就绪。如果就绪度检查失败，**DeploymentConfig** 对象会重新尝试运行 pod，直到超时为止。默认超时为 **10m**，其值在 **dc.spec.strategy.*params** 的 **TimeoutSeconds** 中设置。

8.3.2. Rolling 策略

滚动部署会逐渐将应用程序旧版本实例替换为应用程序的新版本实例。如果 **DeploymentConfig** 对象上没有指定任何策略，则滚动策略是默认的部署策略。

在缩减旧组件前，滚动部署通常会 等待新 pod 变为 **ready**。如果发生严重问题，可以中止 Rolling 部署。

使用滚动部署：

- 希望在应用程序更新过程中不需要停机时。
- 应用程序同时支持运行旧代码和新代码时。

滚动部署意味着您同时运行旧版和新版本的代码。这通常需要您的应用程序可以处理 N-1 兼容性。

滚动策略定义示例

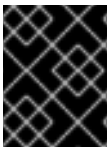
```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  strategy:
    type: Rolling
    rollingParams:
      updatePeriodSeconds: 1 (1)
      intervalSeconds: 1 (2)
      timeoutSeconds: 120 (3)
      maxSurge: "20%" (4)
```

```
    maxUnavailable: "10%" 5
    pre: {} 6
  post: {}
```

1. 各个 pod 更新之间等待的时间。如果未指定，则默认值为 **1**。
2. 更新后轮询部署状态之间等待的时间。如果未指定，则默认值为 **1**。
3. 放弃前等待扩展事件的时间。可选，默认值为 **600**。这里的*放弃*表示自动回滚到以前的完整部署。
4. **maxSurge** 是可选的；如果未指定，则默认为 **25%**。请参考以下流程中的信息。
5. **maxUnavailable** 是可选的；如果未指定，则默认为 **25%** 。请参考以下流程中的信息。
6. **pre** 和 **post** 都是生命周期 hook。

滚动策略：

1. 执行任何 **pre** 生命周期 hook。
2. 根据数量扩展新的复制控制器。
3. 根据最大不可用数，缩减旧的复制控制器。
4. 重复这个扩展，直到新的复制控制器达到所需的副本数，并且旧的复制控制器已缩减为零。
5. 执行任何 **post** 生命周期 hook。



重要

在缩减时，滚动策略会等待 pod 准备就绪，以便它能决定进一步缩放是否会影响到可用性。如果扩展 pod 永不就绪，部署过程将最终超时并导致部署失败。

maxUnavailable 参数是在更新过程中不可用的 pod 的最大数量。**maxSurge** 参数是原始 pod 数量之上最多可以调度的 pod 数量。这两个参数可以设定为百分比（如 **10%**）或绝对值（如 **2**）。两者的默认值都是 **25%**。

这些参数允许对部署的可用性和速度进行调优。例如：

- **maxUnavailable*=0** 和 **maxSurge*=20%** 可确保在更新和快速扩展过程中保持全部容量。
- **maxUnavailable*=10%** 和 **maxSurge*=0** 在执行更新时不使用额外容量（原位更新）。
- **maxUnavailable*=10%** 和 **maxSurge*=10%** 可以快速缩放，可能会有一些容量损失。

一般而言，如果您想要快速推出部署，请使用 **maxSurge**。如果您需要考虑资源配额并可以接受资源部分不可用的情况，则可使用 **maxUnavailable**。

8.3.2.1. Canary 部署

OpenShift Container Platform 中的所有滚动部署都属于 *Canary 部署*；在替换所有旧实例前测试新的版本（Canary）。如果就绪度检查永不成功，则移除 Canary 实例，并且自动回滚 **DeploymentConfig** 对象。

就绪度检查是应用程序代码的一部分，并且可以尽可能的精密，确保新实例就绪可用。如果您必须对应用程序进行更复杂的检查（比如向新实例发送真实用户负载），请考虑实施自定义部署或使用蓝绿部署策略。

8.3.2.2. 创建滚动部署

在 OpenShift Container Platform 中，Rolling 部署是默认类型。您可以使用 CLI 创建滚动部署。

流程

1. 根据 Quay.io 中找到的示例部署镜像创建一个应用程序：

```
$ oc new-app quay.io/openshifttest/deployment-example:latest
```

注意

此镜像不会公开任何端口。如果要通过外部 LoadBalancer 服务公开应用程序，或通过公共互联网访问应用程序，请在完成此步骤后使用 **oc expose dc/deployment-example --port=<port>** 命令创建服务。

2. 如果您安装了路由器，请通过路由（或直接使用服务 IP）提供应用程序。

```
$ oc expose svc/deployment-example
```

3. 通过 **deployment-example.<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

4. 将 **DeploymentConfig** 对象扩展至三个副本：

```
$ oc scale dc/deployment-example --replicas=3
```

5. 通过为示例的新版本标上 **latest** 标签（tag），自动触发新部署：

```
$ oc tag deployment-example:v2 deployment-example:latest
```

6. 在浏览器中刷新页面，直到您看到 **v2** 镜像。

7. 在使用 CLI 时，以下命令显示版本 1 上的 pod 数以及版本 2 上的数量。在 Web 控制台中，pod 逐渐添加到 v2 中并从 v1 中移除：

```
$ oc describe dc deployment-example
```

在部署过程中，新复制控制器以递增方式扩展。新 pod 标记为 **ready**（通过就绪度检查）后，部署过程将继续。

如果 pod 尚未就绪，该过程会中止，部署回滚到之前的版本。

8.3.2.3. 使用 Developer 视角编辑部署

您可以使用 **Developer** 视角编辑部署的部署策略、镜像设置、环境变量和高级选项。

先决条件

- 处于 web 控制台的 **Developer** 视角。
- 您已创建了应用程序。

流程

1. 导航到 **Topology** 视图。
2. 点您的应用程序查看 **Details** 面板。
3. 在 **Actions** 下拉菜单中，选择 **Edit Deployment** 以查看 **Edit Deployment** 页面。
4. 您可以为部署编辑以下**高级选项** ：
   a. 可选：您可以通过点 **Pause rollouts** 暂停推出部署，然后选择 **Pause rollouts for this deployment** 复选框。
   通过暂停推出部署，您可以在不触发推出部署的情况下对应用程序进行更改。您可以随时恢复推出部署。
   b. 可选：通过修改 **Replicas** 的数量，点 **Scaling** 以更改您的镜像实例数量。
5. 点击 **Save**。

8.3.2.4. 使用 Developer 视角启动滚动部署

您可以通过启动滚动部署来升级应用程序。

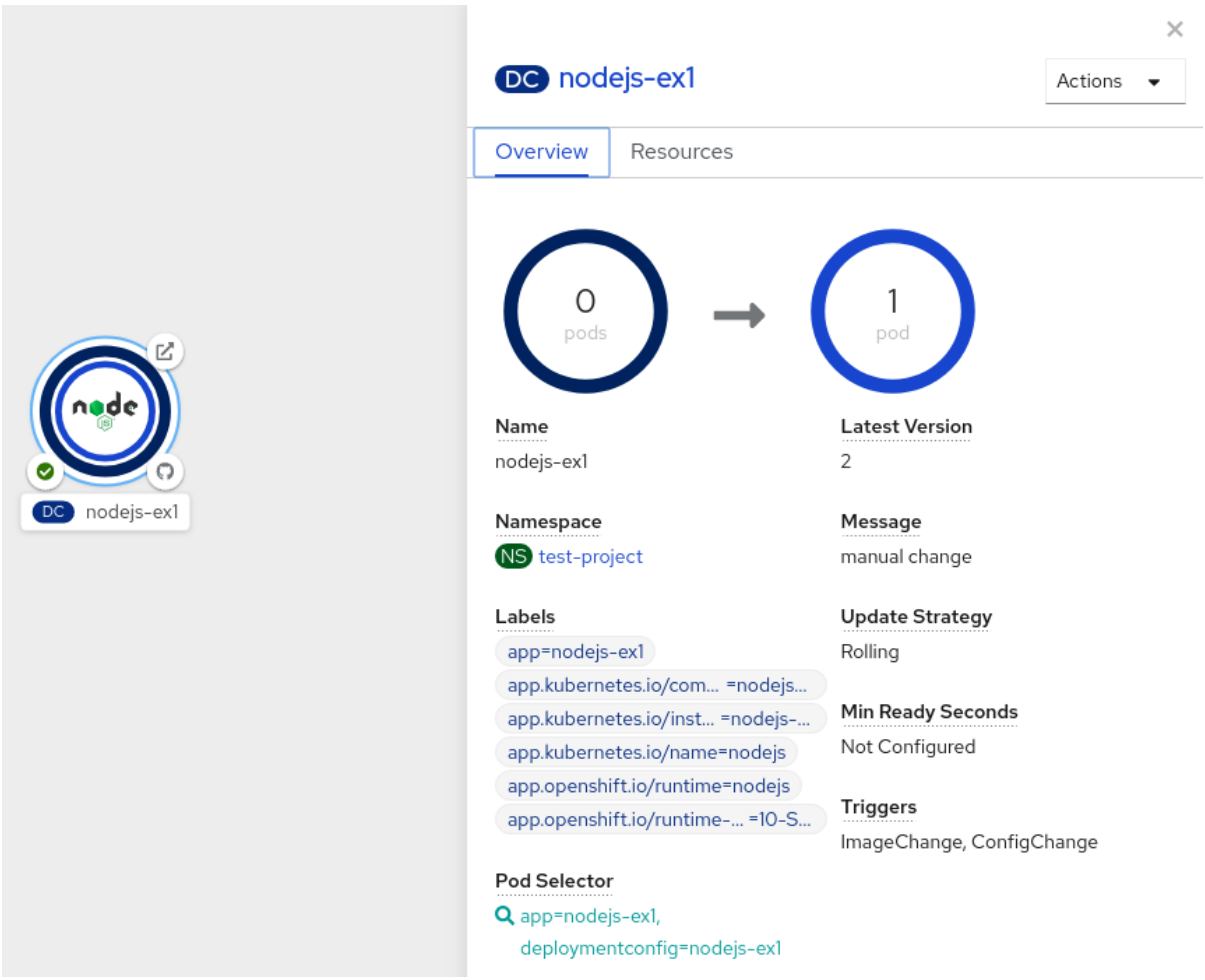
先决条件

- 处于 web 控制台的 **Developer** 视角。
- 您已创建了应用程序。

步骤

1. 在 **Topology** 视图中，点应用程序节点查看侧面面板中的 **Overview** 选项卡。请注意，**Update Strategy** 被设置为默认的 **Rolling** 策略 。
2. 在 **Actions** 下拉菜单中，选择 **Start Rollout** 来启动滚动更新。滚动部署将应用程序更新到新版本，然后终止旧版本。

图 8.1. 滚动更新



其他资源

- 使用 **Developer** 视角在 OpenShift Container Platform 中创建并部署应用程序
- 使用 **Topology** 视图查看项目中的应用程序，验证应用程序的部署状态，并与应用程序进行交互。

8.3.3. Recreate 策略

Recreate（重新创建）策略具有基本的推出部署行为，并支持使用生命周期 hook 将代码注入到部署过程中。

recreate 策略定义示例

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  strategy:
    type: Recreate
    recreateParams: 1
```

```
    pre: {} 2
    mid: {}
    post: {}
```

1 **recreateParams** 可选。

2 **pre**、**mid** 和 **post** 是生命周期 hook。

recreate 策略：

1. 执行任何 **pre** 生命周期 hook。
2. 将上一部署缩减到零。
3. 执行任何 **mid** 生命周期 hook。
4. 向上扩展新的部署。
5. 执行任何 **post** 生命周期 hook。

重要

在扩展过程中，如果部署副本数大于一，则先对部署的第一副本进行就绪状态验证，然后再全面扩展部署。如果第一副本验证失败，部署将被视为失败。

使用重新创建的部署：

- 需要在新代码启动前进行迁移或进行其他数据转换时。
- 不支持同时运行应用程序代码的新旧版本时。
- 当使用 RWO 卷时，不支持在多个副本间共享该卷。

重新创建部署会导致停机，这是因为在短时间内没有运行应用程序实例。然而，旧代码和新代码不会被同时运行。

8.3.3.1. 使用 Developer 视角编辑部署

您可以使用 **Developer** 视角编辑部署的部署策略、镜像设置、环境变量和高级选项。

先决条件

- 处于 web 控制台的 **Developer** 视角。
- 您已创建了应用程序。

流程

1. 导航到 **Topology** 视图。
2. 点您的应用程序查看 **Details** 面板。
3. 在 **Actions** 下拉菜单中，选择 **Edit Deployment** 以查看 **Edit Deployment** 页面。
4. 您可以为部署编辑以下**高级选项** ：

a. 可选：您可以通过点 **Pause rollouts** 暂停推出部署，然后选择 **Pause rollouts for this deployment** 复选框。
通过暂停推出部署，您可以在不触发推出部署的情况下对应用程序进行更改。您可以随时恢复推出部署。

b. 可选：通过修改 **Replicas** 的数量，点 **Scaling** 以更改您的镜像实例数量。

5. 点击 **Save**。

8.3.3.2. 使用 Developer 视角启动重新创建的部署

您可以使用 web 控制台中的 **Developer** 视角将部署策略从默认的滚动模式切换到重新创建模式。

先决条件

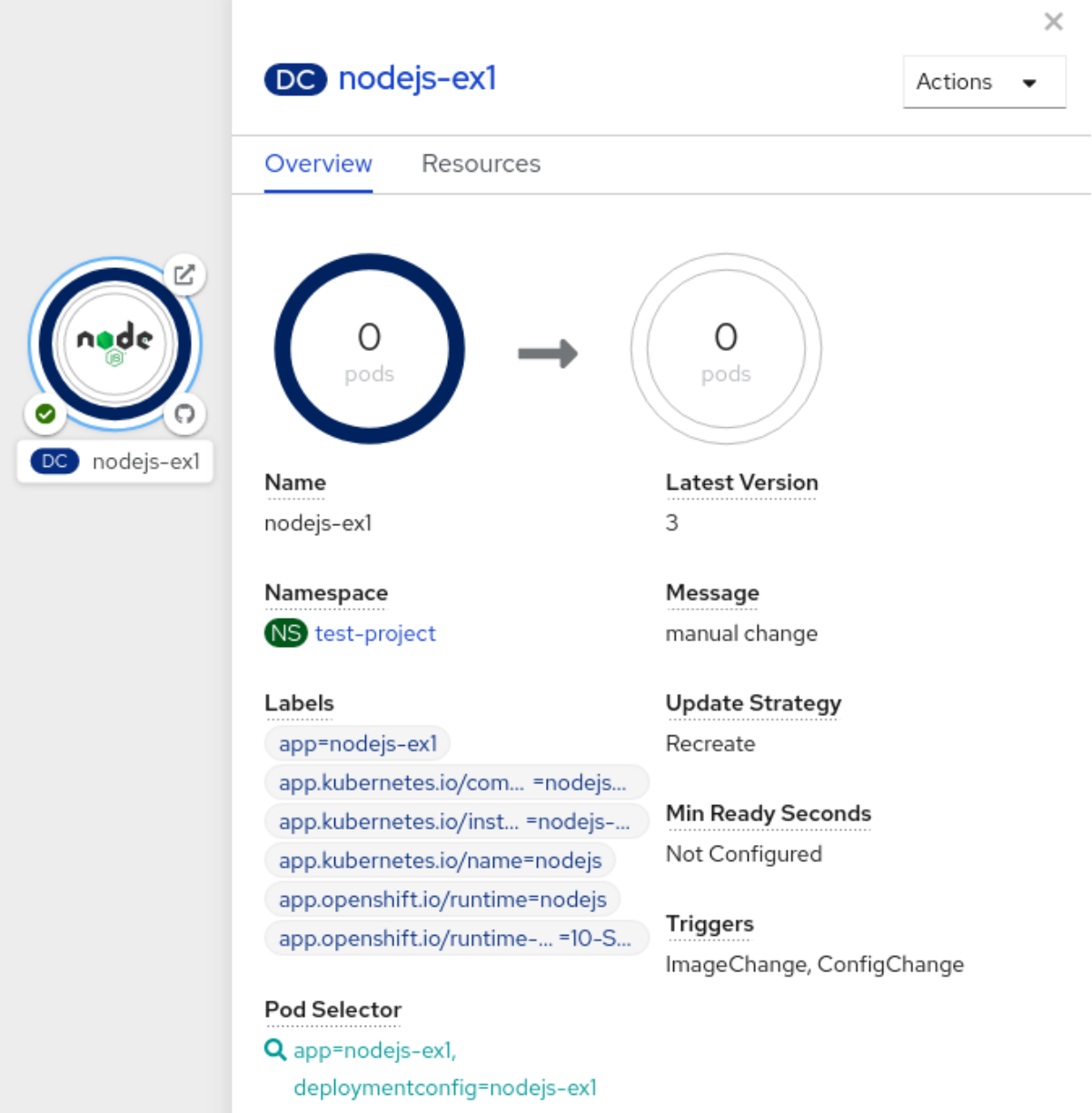
- 确认您使用 web 控制台的 **Developer** 视角。
- 确保您已使用 **Add** 视图创建了一个应用程序，并可以在 **Topology** 视图中查看它。

流程

切换到重新创建的更新策略并升级应用程序：

1. 点您的应用程序查看 **Details** 面板。
2. 在 **Actions** 下拉菜单中，选择 **Edit Deployment Config** 来查看应用程序的部署配置详情。
3. 在 YAML 编辑器中，将 **spec.strategy.type** 更改为 **Recreate**，然后点 **Save**。
4. 在 **Topology** 视图中，选择节点即可看到侧面板中的 **Overview** 选项卡。**Update Strategy** 现在设为 **Recreate**。
5. 使用 **Actions** 下拉菜单选择 **Start Rollout** 以使用 recreate 策略启动更新。recreate 策略首先会终止旧版本应用程序的 pod，然后为新版本启动 pod。

图 8.2. 重新创建更新



其他资源

- 使用 **Developer** 视角在 OpenShift Container Platform 中创建并部署应用程序
- 使用 **Topology** 视图查看项目中的应用程序，验证应用程序的部署状态，并与应用程序进行交互。

8.3.4. Custom 策略

自定义（Custom）策略允许您提供自己的部署行为。

Custom 策略定义示例

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
```

```
# ...
  strategy:
    type: Custom
    customParams:
      image: organization/strategy
      command: [ "command", "arg1" ]
      environment:
        - name: ENV_1
          value: VALUE_1
```

在上例中，**organization/strategy** 容器镜像提供部署行为。可选的 **command** 数组覆盖镜像的 **Dockerfile** 中指定的任何 **CMD** 指令。提供的可选环境变量添加到策略过程的执行环境中。

另外，OpenShift Container Platform 为部署过程提供以下环境变量：

环境变量	描述
OPENSHIFT_DEPLOYMENT_NAME	新部署的名称，即复制控制器。
OPENSHIFT_DEPLOYMENT_NAMESPACE	新部署的命名空间。

新部署的副本数最初为零。该策略负责使新部署积极使用最能满足用户需求的逻辑。

另外，也可使用 **customParams** 对象将自定义部署逻辑注入现有的部署策略中。提供自定义 shell 脚本逻辑并调用 **openshift-deploy** 二进制文件。用户不必提供自定义的部署器容器镜像；本例中使用默认的 OpenShift Container Platform 部署器镜像：

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  strategy:
    type: Rolling
    customParams:
      command:
      - /bin/sh
      - -c
      - |
        set -e
        openshift-deploy --until=50%
        echo Halfway there
        openshift-deploy
        echo Complete
```

这会产生以下部署：

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
```

```
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-1 down to 1
    Scaling custom-deployment-2 up to 2
    Scaling custom-deployment-1 down to 0
--> Success
Complete
```

如果自定义部署策略过程需要访问 OpenShift Container Platform API 或 Kubernetes API，执行该策略的容器可以使用容器中的服务帐户令牌进行身份验证。

8.3.4.1. 使用 Developer 视角编辑部署

您可以使用 **Developer** 视角编辑部署的部署策略、镜像设置、环境变量和高级选项。

先决条件

- 处于 web 控制台的 **Developer** 视角。
- 您已创建了应用程序。

流程

1. 导航到 **Topology** 视图。
2. 点您的应用程序查看 **Details** 面板。
3. 在 **Actions** 下拉菜单中，选择 **Edit Deployment** 以查看 **Edit Deployment** 页面。
4. 您可以为部署编辑以下**高级选项** ：
    a. 可选：您可以通过点 **Pause rollouts** 暂停推出部署，然后选择 **Pause rollouts for this deployment** 复选框。
    通过暂停推出部署，您可以在不触发推出部署的情况下对应用程序进行更改。您可以随时恢复推出部署。
    b. 可选：通过修改 **Replicas** 的数量，点 **Scaling** 以更改您的镜像实例数量。
5. 点击 **Save**。

8.3.5. 生命周期 hook

滚动和重新创建策略支持 *生命周期 hook* 或部署 hook，它允许在策略的预定义点将行为注入到部署过程中：

pre 生命周期 hook 示例

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1 **execNewPod** 是基于 pod 的生命周期 hook。

每个 hook 都有一个 *失败策略*，定义在遇到 hook 失败时策略应执行的操作：

Abort	如果 hook 失败，部署过程将被视为失败。
Retry	应重试 hook 执行过程，直到成功为止。
Ignore	所有 hook 失败都应忽略，部署应继续进行。

Hook 具有特定类型的字段，用于描述如何执行 Hook。目前，基于 pod 的 hook 是唯一受支持的 hook 类型，通过 **execNewPod** 字段指定。

基于 Pod 的生命周期 hook

基于 Pod 的生命周期 hook 在来自 **DeploymentConfig** 对象模板的新 pod 中执行 hook 代码。

以下简化的部署示例使用了滚动策略。为简明起见，省略了触发器和其他一些次要的细节：

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld 1
          command: [ "/usr/bin/command", "arg1", "arg2" ] 2
          env: 3
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data 4
```

1 **helloworld** 名称指代 **spec.template.spec.containers[0].name**。

2 此 **command** 覆盖 **openshift/origin-ruby-sample** 镜像中定义的任何 **ENTRYPOINT**。

3 **env** 是 hook 容器的一组可选环境变量。

4 **volumes** 是 hook 容器的一组可选的卷引用。

在本例中，将使用 **helloworld** 容器中的 **openshift/origin-ruby-sample** 镜像在新 pod 中执行 **pre** hook。hook pod 具有以下属性：

- hook 命令是 **/usr/bin/command arg1 arg2**。
- hook 容器包含 **CUSTOM_VAR1=custom_value1** 环境变量。
- hook 失败策略是 **Abort**；即 hook 失败时部署过程也会失败。
- hook pod 从 **DeploymentConfig** 对象 pod 继承 **data** 卷。

8.3.5.1. 设置生命周期 hook

您可以使用 CLI 为部署设置生命周期 hook 或部署 hook。

流程

1. 使用 **oc set deployment-hook** 命令设定您想要的 hook 类型：**--pre**、**--mid** 或 **--post**。例如，设置部署前 hook：

```
$ oc set deployment-hook dc/frontend \
    --pre -c helloworld -e CUSTOM_VAR1=custom_value1 \
    --volumes data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

8.4. 使用基于路由的部署策略

部署策略为应用程序的演进提供了一个途径。有些策略使用 **Deployment** 对象进行解析到应用程序的所有路由用户可见的更改。其他高级策略，例如本节中描述的策略，结合使用路由器功能和 **Deployment** 对象来影响特定的路由。

最常用的基于路由型策略是使用*蓝绿部署*。新版本（绿色版本）上线进行测试和评估，同时用户仍然使用稳定版本（蓝色版本）。准备就绪后，用户切换到绿色版本。如果出现问题，您可以切回到蓝色版本。

一个常见的替代策略是使用同时活跃的 *A/B 版本*；一些用户使用一个版本，另一些用户使用另一个版本。这可用于试验用户界面变化和其他功能，以获取用户反馈。它还可用来在影响有限用户的生产环境中验证正确的操作。

Canary 部署会测试新版本，但在检测到问题时，迅速回退到上一版本。这可以通过以上两个策略实现。

基于路由的部署策略不会缩放服务中的 pod 数。要保持所需的性能特性，部署配置可能必须要扩展。

8.4.1. 代理分片和流量分割

在生产环境中，您可以精确控制特定分片上的流量分布。在处理大量实例时，可以使用相对比例的独立分片来实现基于百分比的流量分布。这可与*代理分片*良好结合，将接收到的流量转发或分割到在其他位置运行的单独服务或应用程序。

在最简单的配置中，代理会原封不动转发请求。在比较复杂的设置中，可以复制传入的请求，同时将它们发送到独立集群以及应用程序的本地实例，并且比较其结果。其他模式包括使 DR 安装的缓存保持活跃，或抽样传入的流量来满足分析需要。

任何 TCP（或 UDP）代理都可以在所需的分片下运行。使用 **oc scale** 命令更改代理分片下服务请求的相对数量。对于更复杂的流量管理，请考虑使用比例平衡功能自定义 OpenShift Container Platform 路由器。

8.4.2. N-1 兼容性

同时运行新旧代码的应用程序必须要谨慎处理，以确保新代码写入的数据能被旧版代码读取和处理（或恰当忽略）。这有时被称为*架构演进*，而且是一个复杂的问题。

这可采用多种形式：数据存储在磁盘、数据库或临时缓存中，或作为用户浏览器会话的一部分。虽然大多数 Web 应用程序都支持滚动部署，但务必要测试并设计您的应用程序以便能处理它。

在一些应用程序中，同时运行新旧代码的时间是短暂的，因此程序错误或一些用户事务失败是可以接受的。至于其他应用程序，失败模式可能会导致整个应用程序无法运作。

验证 N-1 兼容性的一种方法是使用 A/B 部署：在测试环境中以受控的方式同时运行旧代码和新代码，并验证流向新部署的流量不会导致旧部署失败。

8.4.3. 恰当终止

OpenShift Container Platform 和 Kubernetes 会留出时间，让应用程序实例关机后再从负载均衡轮转中移除。但是，应用程序必须保证在用户退出前彻底终止用户连接。

在关闭时，OpenShift Container Platform 会向容器中的进程发送一个 **TERM** 信号。在接收 **SIGTERM** 时，应用程序代码停止接受新的连接。这样可确保负载均衡器将流量路由到其他活跃实例。然后，应用程序代码会等到所有开启的连接都关闭（或在下次机会出现时恰当终止独立的连接）后再退出。

在恰当终止周期到期后，还未退出的进程会收到 **KILL** 信号，该信号会立即结束此进程。pod 或 pod 模板的 **terminationGracePeriodSeconds** 属性控制恰当终止期限（默认值 30 秒），并可根据需要自定义每个应用程序。

8.4.4. 蓝绿部署

蓝绿部署涉及同时运行应用程序的两个版本，并将流量从生产版本（蓝色版本）移动到更新版本（绿色版本）。您可以使用滚动策略或切换路由中的服务。

由于许多应用程序依赖于持久性数据，您必须有支持 *N-1 兼容性*的应用程序；这意味着，通过创建数据层的两个副本在数据库、存储或磁盘间共享数据并实现实时迁移。

以测试新版本时使用的数据为例。如果是生产数据，新版本中的错误可能会破坏生产版本。

8.4.4.1. 设置蓝绿部署

蓝绿部署使用两个 **Deployment** 对象。这两者都在运行，生产环境中的 Deployment 依赖于路由指定的服务，每个 **Deployment** 对象公开给不同的服务。

注意

路由适用于 Web（HTTP 和 HTTPS）流量，因此这种技术最适合 Web 应用程序。

您可以创建指向新版本的新路由并进行测试。准备就绪后，将生产路径中的服务更改为指向新服务，使新（绿色）版本上线。

如果需要，可以通过将服务切回到之前的版本以回滚到老版本（蓝色）。

流程

1. 创建两个独立的应用程序组件。

   a. 在 **example-blue** 服务下，创建运行 **v1** 镜像的示例应用程序的副本：

   ```
   $ oc new-app openshift/deployment-example:v1 --name=example-blue
   ```

   b. 杂 **example-green** 服务下，创建使用 **v2** 镜像的第二个副本：

   ```
   $ oc new-app openshift/deployment-example:v2 --name=example-green
   ```

2. 创建指向旧服务的路由：

   ```
   $ oc expose svc/example-blue --name=bluegreen-example
   ```

3. 通过 **bluegreen-example-<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

4. 编辑路由并将服务名称改为 **example-green**：

   ```
   $ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-green"}}}'
   ```

5. 要验证路由是否已改变，请刷新浏览器直到您看到 **v2** 镜像。

8.4.5. A/B 部署

A/B 部署策略允许您在生产环境中以有限的方式尝试应用程序的新版本。您可以指定生产版本获得大多数用户请求，同时让有限比例的请求进入新版本。

由于您掌控进入每个版本的请求比例，因此随着测试的推进，您可以增加进入新版本的请求的比例，最终停止使用旧版本。当您调整每个版本的请求负载时，可能需要扩展各个服务中的 pod 数，以提供预期的性能。

除了升级软件外，您还可以使用此功能来试验用户界面的不同版本。由于部分用户会使用旧版本，而另外的一部分用户会使用新版本，因此您可以评估用户对不同版本的反应，以做出明智的设计决策。

若要使此功能凑效，新旧两个版本必须足够相似，让两个版本能够同时运行。这常用于对程序错误修复的发布，也适用于新功能不会影响到旧功能的情况。各个版本需要支持 N-1 兼容性才能正常工作。

OpenShift Container Platform 通过 Web 控制台和 CLI 支持 N-1 兼容性。

8.4.5.1. A/B 测试负载均衡

用户使用多个服务设置路由。每个服务负责应用程序的一个版本。

每个服务分配到一个 **weight**，进入每个服务的请求的比例等于 **service_weight** 除以 **sum_of_weights**。每个服务的 **weight** 分布到该服务的端点，使得端点 **weight** 的总和等于服务 **weight**。

路由最多可有四个服务。服务的 **weight** 可以在 **0** 到 **256** 范围内。当 **weight** 等于 **0** 时，服务不参与负载均衡，但继续为现有的持久连接服务。当服务 **weight** 不为 **0** 时，每个端点的最小 **weight** 为 **1**。因此，具有大量端点的服务会得到高于预期值的 **weight**。在本例中，减少 pod 数量以获得预期的负载均衡 **weight**。

流程

设置 A/B 环境：

1. 创建两个应用程序并使用不同的名称。它们各自创建一个 **Deployment** 对象。应用程序是同一程序的不同版本；一个是当前生产版本，另一个是提议的新版本。

   a. 创建第一个应用程序。以下示例创建了一个名为 **ab-example-a** 的应用程序：

   ```
   $ oc new-app openshift/deployment-example --name=ab-example-a
   ```

   b. 创建第二个应用程序：

   ```
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b
   ```

   两个应用程序都已部署，也创建了服务。

2. 通过路由对外提供应用程序。此时您可以公开其中任一个。先公开当前生产版本，稍后修改路由来添加新版本，这可能比较方便。

   ```
   $ oc expose svc/ab-example-a
   ```

   在 **ab-example-a.<project>.<router_domain>** 查看应用程序，以确保可以看到预期的版本。

3. 当您部署路由时，路由器会根据为服务指定的 **weight** 来均衡流量。此时，存在具有默认 **weight=1** 的单一服务，因此所有请求都会进入该服务。添加其他服务作为 **alternateBackend** 并调整 **weight**，即可激活 A/B 设置。这可通过 **oc set route-backends** 命令或编辑路由来完成。

   注意

   使用 **alternateBackends** 时，也使用 **roundrobin** 负载均衡策略来确保请求按预期分发到服务。也可以使用路由注解为一个路由设置 **roundrobin**。

   如果将 **oc set route-backend** 设为 **0**，则服务不参与负载均衡，但继续为现有的持久连接服务。

   注意

   对路由的更改只会改变流量进入各个服务的比例。您可能需要扩展部署来调整 pod 数量，以处理预期的负载。

   若要编辑路由，请运行：

   ```
   $ oc edit route <route_name>
   ```

   输出示例

   ```
   apiVersion: route.openshift.io/v1
   kind: Route
   metadata:
   metadata:
     name: route-alternate-service
     annotations:
       haproxy.router.openshift.io/balance: roundrobin
   # ...
   spec:
   ```

```
  host: ab-example.my-project.my-domain
  to:
    kind: Service
    name: ab-example-a
    weight: 10
  alternateBackends:
  - kind: Service
    name: ab-example-b
    weight: 15
# ...
```

8.4.5.1.1. 使用 Web 控制台管理现有路由的权重

流程

1. 进入 **Networking → Routes** 页面。
2. 点击您要编辑的路由旁的 Actions 菜单  ，然后选择 **Edit Route**。
3. 编辑 YAML 文件。把 **weight** 更新为 **0** 到 **256** 之间的一个整数，用于指定目标相对于其他目标引用对象的相对权重。值 **0** 会把请求限制到这个后端。默认值为 **100**。运行 **oc explain routes.spec.alternateBackends** 以了解有关选项的更多信息。
4. 点 **Save**。

8.4.5.1.2. 使用 Web 控制台管理新路由的权重

1. 进入 **Networking → Routes** 页面。
2. 点击 **Create Route**。
3. 输入路由 **名称**。
4. 选择 **Service**。
5. 点 **Add Alternate Service**。
6. 为 **Weight** 和 **Alternate Service Weight** 输入一个值。输入一个 **0** 到 **255** 之间的数字，它显示了与其他目标相比的相对权重。默认值为 **100**。
7. 选择 **Target Port**。
8. 点击 **Create**。

8.4.5.1.3. 使用 CLI 管理权重

流程

1. 要管理由路由均衡负载的服务以及对应的权重，请使用 **oc set route-backends** 命令：

```
$ oc set route-backends ROUTENAME \
    [--zero|--equal] [--adjust] SERVICE=WEIGHT[%] [...] [options]
```

例如，以下命令将 **ab-example-a** 设为主服务（**weight=198**）并将 **ab-example-b** 设为第一替代服务（**weight=2**）：

```
$ oc set route-backends ab-example ab-example-a=198 ab-example-b=2
```

这意味着 99% 的流量发送到服务 **ab-example-a**，1% 发送到 **ab-example-b**。

此命令不扩展部署。您可能需要进行此操作，才能有足够的 pod 来处理请求负载。

2. 不带标志运行命令来验证当前配置：

```
$ oc set route-backends ab-example
```

输出示例

```
NAME                KIND     TO          WEIGHT
routes/ab-example       Service  ab-example-a 198 (99%)
routes/ab-example       Service  ab-example-b 2   (1%)
```

3. 要改变个别服务相对于自身或主服务的权重，请使用 **--adjust** 标志。指定百分比来调整服务相对于主服务或第一替代服务（如果指定了主服务）的权重。如果还有其他后端，它们的权重会与更改后的值保持比例。
以下示例会更改 **ab-example-a** 和 **ab-example-b** 服务的权重：

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
```

或者，通过指定百分比来改变服务的权重：

```
$ oc set route-backends ab-example --adjust ab-example-b=5%
```

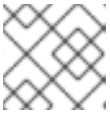
通过在百分比声明前指定 **+**，您可以调整相对于当前设置的权重。例如：

```
$ oc set route-backends ab-example --adjust ab-example-b=+15%
```

--equal 标志将所有服务的 **weight** 设为 **100**：

```
$ oc set route-backends ab-example --equal
```

--zero 标志将所有服务的 **weight** 设为 **0**。之后，所有请求都会返回 503 错误。



注意

并非所有路由器都支持多个后端或加权后端。

8.4.5.1.4. 一个服务，多个 Deployment 对象

流程

1. 创建一个新应用程序，添加对所有分片都通用的 **ab-example=true** 标签：

```
$ oc new-app openshift/deployment-example --name=ab-example-a --as-deployment-
config=true --labels=ab-example=true --env=SUBTITLE\=shardA
```

```
$ oc delete svc/ab-example-a
```

应用程序完成部署，并创建了服务。这是第一个分片。

2. 通过路由提供应用程序，或者直接使用服务 IP:

```
$ oc expose deployment ab-example-a --name=ab-example --selector=ab-example\=true
```

```
$ oc expose service ab-example
```

3. 通过 **ab-example-<project_name>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

4. 创建第二个分片，它基于与第一分片相同的源镜像和标签，但使用不同的标记版本和独特的环境变量：

```
$ oc new-app openshift/deployment-example:v2 \
    --name=ab-example-b --labels=ab-example=true \
    SUBTITLE="shard B" COLOR="red" --as-deployment-config=true
```

```
$ oc delete svc/ab-example-b
```

5. 在这一刻，路由下同时提供了两组 pod。但是，由于两个浏览器（通过使连接保持打开）和路由器（默认借助 Cookie）都试图保留后端服务器的连接，您可能不会看到两个分片都返回给您。使浏览器强制到其中一个分片：

   a. 使用 **oc scale** 命令将 **ab-example-a** 的副本数减少到 **0**。

   ```
   $ oc scale dc/ab-example-a --replicas=0
   ```

   刷新浏览器以显示 **v2** 和 **shard B**（红色）。

   b. 将 **ab-example-a** 扩展为 **1** 个副本，**ab-example-b** 调到 **0**：

   ```
   $ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
   ```

   刷新浏览器以显示 **v1** 和 **shard A**（蓝色）。

6. 如果您对其中任一分片触发部署，那么只有该分片中的 pod 会受到影响。您可以通过在任一 **Deployment** 对象中更改 **SUBTITLE** 环境变量来触发部署：

```
$ oc edit dc/ab-example-a
```

或者

```
$ oc edit dc/ab-example-b
```

第 9 章 配额

9.1. 项目的资源配额

资源配额 由 **ResourceQuota** 对象定义，提供约束来限制各个项目的聚合资源消耗。它可根据类型限制项目中创建的对象数量，以及该项目中资源可以消耗的计算资源和存储的总和。

本指南阐述了资源配额如何工作，集群管理员如何以项目为基础设置和管理资源配额，以及开发人员和集群管理员如何查看配额。

9.1.1. 配额管理的资源

下方描述了可通过配额管理的一系列计算资源和对象类型。

注意

如果 **status.phase in (Failed, Succeeded)** 为 true，则 Pod 处于终端状态。

表 9.1. 配额管理的计算资源

资源名称	描述
cpu	非终端状态的所有 Pod 的 CPU 请求总和不能超过这个值。**CPU** 和 **requests.cpu** 的值相同，并可互换使用。
memory	非终端状态的所有 Pod 的 内存请求总和不能超过这个值。**memory** 和 **requests.memory** 的值相同，并可互换使用。
requests.cpu	非终端状态的所有 Pod 的 CPU 请求总和不能超过这个值。**CPU** 和 **requests.cpu** 的值相同，并可互换使用。
requests.memory	非终端状态的所有 Pod 的 内存请求总和不能超过这个值。**memory** 和 **requests.memory** 的值相同，并可互换使用。
limits.cpu	非终端状态的所有 Pod 的 CPU 限值总和不能超过这个值。
limits.memory	非终端状态的所有 Pod 的内存限值总和不能超过这个值。

表 9.2. 配额管理的存储资源

资源名称	描述
requests.storage	处于任何状态的所有持久性卷声明的存储请求总和不能超过这个值。
persistentvolumeclaims	项目中可以存在的持久性卷声明的总数。

资源名称	描述
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	在处于任何状态且具有匹配存储类的所有持久性卷声明中，存储请求总和不能超过这个值。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	项目中可以存在的具有匹配存储类的持久性卷声明的总数。
ephemeral-storage	非终端状态的所有本地临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。
requests.ephemeral-storage	非终端状态的所有临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。
limits.ephemeral-storage	非终端状态的所有 Pod 的临时存储限值总和不能超过这个值。

表 9.3. 配额管理的对象计数

资源名称	描述
pods	项目中可以存在的处于非终端状态的 Pod 总数。
replicationcontrollers	项目中可以存在的 ReplicationController 的总数。
resourcequotas	项目中可以存在的资源配额总数。
services	项目中可以存在的服务总数。
services.loadbalancers	项目中可以存在的 **LoadBalancer** 类型的服务总数。
services.nodeports	项目中可以存在的 **NodePort** 类型的服务总数。
secrets	项目中可以存在的 secret 的总数。
configmaps	项目中可以存在的 **ConfigMap** 对象的总数。
persistentvolumeclaims	项目中可以存在的持久性卷声明的总数。
openshift.io/imagestreams	项目中可以存在的镜像流的总数。

9.1.2. 配额范围

每个配额都有一组关联的*范围*。配额只在与枚举的范围交集匹配时才会测量资源的使用量。

为配额添加范围会限制该配额可应用的资源集合。指定允许的集合之外的资源会导致验证错误。

影响范围	描述
BestEffort	匹配 **cpu** 或 **memory** 具有最佳服务质量的 Pod。
NotBestEffort	匹配 **cpu** 和 **memory** 没有最佳服务质量的 Pod。

BestEffort 范围将配额仅限为限制以下资源：

- **pods**

NotBestEffort 范围限制配额跟踪以下资源：

- **pods**
- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**

9.1.3. 配额强制

在项目中首次创建资源配额后，项目会限制您创建可能会违反配额约束的新资源，直到它计算了更新后的使用量统计。

在创建了配额并且更新了使用量统计后，项目会接受创建新的内容。当您创建或修改资源时，配额使用量会在请求创建或修改资源时立即递增。

在您删除资源时，配额使用量在下一次完整重新计算项目的配额统计时才会递减。可配置的时间量决定了将配额使用量统计降低到其当前观察到的系统值所需的时间。

如果项目修改超过配额使用量限值，服务器会拒绝该操作，并将对应的错误消息返回给用户，解释违反了配额约束，并说明系统中目前观察到的使用量统计。

9.1.4. 请求与限值

在分配计算资源时，每个容器可能会为 CPU、内存和临时存储各自指定请求和限制值。配额可以限制任何这些值。

如果配额具有为 **requests.cpu** 或 **requests.memory** 指定的值，那么它要求每个传入的容器都明确请求那些资源。如果配额具有为 **limits.cpu** 或 **limits.memory** 指定的值，那么它要求每个传入的容器为那些资源指定一个显性限值。

9.1.5. 资源配额定义示例

core-object-counts.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" (1)
    persistentvolumeclaims: "4" (2)
    replicationcontrollers: "20" (3)
    secrets: "10" (4)
    services: "10" (5)
    services.loadbalancers: "2" (6)
```

1. 项目中可以存在的 **ConfigMap** 对象的总数。
2. 项目中可以存在的持久性卷声明 (PVC) 的总数。
3. 项目中可以存在的复制控制器的总数。
4. 项目中可以存在的 secret 的总数。
5. 项目中可以存在的服务总数。
6. 项目中可以存在的 **LoadBalancer** 类型的服务总数。

openshift-object-counts.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: openshift-object-counts
spec:
  hard:
    openshift.io/imagestreams: "10" (1)
```

1. 项目中可以存在的镜像流的总数。

compute-resources.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" (1)
    requests.cpu: "1" (2)
```

```
    requests.memory: 1Gi 3
    limits.cpu: "2" 4
    limits.memory: 2Gi 5
```

1 项目中可以存在的处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 请求总和不能超过 1 个内核。

3 在非终端状态的所有 Pod 中，内存请求总和不能超过 1Gi。

4 在非终端状态的所有 Pod 中，CPU 限值总和不能超过 2 个内核。

5 在非终端状态的所有 Pod 中，内存限值总和不能超过 2Gi。

besteffort.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
  - BestEffort 2
```

1 项目中可以存在的具有 **BestEffort** 服务质量的非终端状态 Pod 的总数。

2 将配额仅限为在内存或 CPU 方面具有 **BestEffort** 服务质量的匹配 Pod。

compute-resources-long-running.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
  scopes:
  - NotTerminating 4
```

1 处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 限值总和不能超过这个值。

3 在非终端状态的所有 Pod 中，内存限值总和不能超过这个值。

4 将配额仅限为 **spec.activeDeadlineSeconds** 设为 **nil** 的匹配 Pod。构建 pod 不在 **NotTerminating** 下，除非应用了 **RestartNever** 策略。

compute-resources-time-bound.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" (1)
    limits.cpu: "1" (2)
    limits.memory: "1Gi" (3)
  scopes:
  - Terminating (4)
```

1. 处于终止状态的 pod 总数。
2. 在处于终止状态的所有 Pod 中，CPU 限值总和不能超过这个值。
3. 在处于终止状态的所有 Pod 中，内存限值总和不能超过这个值。
4. 将配额仅限为 **spec.activeDeadlineSeconds >=0** 的匹配 Pod。例如，此配额适用于构建或部署器 Pod，而非 Web 服务器或数据库等长时间运行的 Pod。

storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" (1)
    requests.storage: "50Gi" (2)
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi" (3)
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" (4)
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" (5)
    bronze.storageclass.storage.k8s.io/requests.storage: "0" (6)
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" (7)
    requests.ephemeral-storage: 2Gi (8)
    limits.ephemeral-storage: 4Gi (9)
```

1. 项目中的持久性卷声明总数
2. 在一个项目中的所有持久性卷声明中，请求的存储总和不能超过这个值。
3. 在一个项目中的所有持久性卷声明中，金级存储类中请求的存储总和不能超过这个值。
4. 在一个项目中的所有持久性卷声明中，银级存储类中请求的存储总和不能超过这个值。
5. 在一个项目中的所有持久性卷声明中，银级存储类中声明总数不能超过这个值。
6. 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 **0**，则表示铜级存储类无法请求存储。

7 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 **0**，则表示铜级存储类无法创建声明。

8 在非终端状态的所有 Pod 中，临时存储请求总和不能超过 2Gi。

9 在非终端状态的所有 Pod 中，临时存储限值总和不能超过 4Gi。

9.1.6. 创建配额

您可以通过创建配额，来约束给定项目中的资源使用量。

流程

1. 在一个文件中定义配额。
2. 使用该文件创建配额，并将其应用到项目：

```
$ oc create -f <file> [-n <project_name>]
```

例如：

```
$ oc create -f core-object-counts.yaml -n demoproject
```

9.1.6.1. 创建对象数配额

您可以为 OpenShift Container Platform 上的所有标准命名空间资源类型创建对象数配额，如 **BuildConfig** 和 **DeploymentConfig** 对象。对象配额数将定义的配额施加于所有标准命名空间资源类型。

在使用资源配额时，对象会根据创建的配额进行收费。这些类型的配额对防止耗尽资源很有用处。只有在项目中有足够的备用资源时，才能创建配额。

流程

为资源配置对象数配额：

1. 运行以下命令:

```
$ oc create quota <name> \
    --hard=count/<resource>.<group>=<quota>,count/<resource>.<group>=<quota> 1
```

1 **<resource>** 变量是资源名称，**<group>** 则是 API 组（如果适用）。使用 **oc api-resources** 命令可以列出资源及其关联的 API 组。

例如：

```
$ oc create quota test \
    --
hard=count/deployments.extensions=2,count/replicasets.extensions=4,count/pods=3,count/secr
ets=4
```

输出示例

```
resourcequota "test" created
```

本例将列出的资源限制为集群中各个项目的硬限值。

2. 验证是否创建了配额：

```
$ oc describe quota test
```

输出示例

```
Name:                       test
Namespace:                  quota
Resource                    Used  Hard
--------                    ----  ----
count/deployments.extensions  0     2
count/pods                    0     3
count/replicasets.extensions  0     4
count/secrets                 0     4
```

9.1.6.2. 为扩展资源设定资源配额

扩展资源不允许过量使用资源，因此您必须在配额中为相同扩展资源指定 **requests** 和 **limits**。目前，扩展资源只允许使用带有前缀 **requests.** 配额项。以下是如何为 GPU 资源 **nvidia.com/gpu** 设置资源配额的示例场景。

流程

1. 确定集群中某个节点中有多少 GPU 可用。例如：

```
# oc describe node ip-172-31-27-209.us-west-2.compute.internal | egrep
'Capacity|Allocatable|gpu'
```

输出示例

```
                    openshift.com/gpu-accelerator=true
Capacity:
 nvidia.com/gpu:  2
Allocatable:
 nvidia.com/gpu:  2
  nvidia.com/gpu  0           0
```

本例中有 2 个 GPU 可用。

2. 创建一个 **ResourceQuota** 对象，在命名空间 **nvidia** 中设置配额。本例中配额为 **1**：

输出示例

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gpu-quota
  namespace: nvidia
```

```
spec:
  hard:
    requests.nvidia.com/gpu: 1
```

3. 创建配额：

```
# oc create -f gpu-quota.yaml
```

输出示例

```
resourcequota/gpu-quota created
```

4. 验证命名空间是否设置了正确的配额：

```
# oc describe quota gpu-quota -n nvidia
```

输出示例

```
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  0     1
```

5. 定义一个请求单个 GPU 的 Pod。以下示例定义文件名为 **gpu-pod.yaml**：

```
apiVersion: v1
kind: Pod
metadata:
  generateName: gpu-pod-
  namespace: nvidia
spec:
  restartPolicy: OnFailure
  containers:
  - name: rhel7-gpu-pod
    image: rhel7
    env:
      - name: NVIDIA_VISIBLE_DEVICES
        value: all
      - name: NVIDIA_DRIVER_CAPABILITIES
        value: "compute,utility"
      - name: NVIDIA_REQUIRE_CUDA
        value: "cuda>=5.0"
    command: ["sleep"]
    args: ["infinity"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

6. 创建 pod：

```
# oc create -f gpu-pod.yaml
```

7. 验证 Pod 是否在运行：

```
# oc get pods
```

输出示例

```
NAME           READY    STATUS     RESTARTS   AGE
gpu-pod-s46h7  1/1      Running    0          1m
```

8. 验证配额计数器 **Used** 是否正确：

```
# oc describe quota gpu-quota -n nvidia
```

输出示例

```
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  1     1
```

9. 尝试在 **nvidia** 命名空间中创建第二个 GPU Pod。从技术上讲这是可行的，因为它有 2 个 GPU：

```
# oc create -f gpu-pod.yaml
```

输出示例

```
Error from server (Forbidden): error when creating "gpu-pod.yaml": pods "gpu-pod-f7z2w" is
forbidden: exceeded quota: gpu-quota, requested: requests.nvidia.com/gpu=1, used:
requests.nvidia.com/gpu=1, limited: requests.nvidia.com/gpu=1
```

应该会显示此 **Forbidden** 错误消息，因为您有设为 1 个 GPU 的配额，但这一 Pod 试图分配第二个 GPU，而这超过了配额。

9.1.7. 查看配额

您可以在 Web 控制台导航到项目的 **Quota** 页面，查看与项目配额中定义的硬限值相关的使用量统计。

您还可以使用命令行来查看配额详情。

流程

1. 获取项目中定义的配额列表。例如，对于名为 **demoproject** 的项目：

```
$ oc get quota -n demoproject
```

输出示例

```
NAME                         AGE    REQUEST
LIMIT
besteffort                   4s     pods: 1/2
compute-resources-time-bound   10m    pods: 0/2
```

```
limits.cpu: 0/1, limits.memory: 0/1Gi
core-object-counts          109s   configmaps: 2/10, persistentvolumeclaims: 1/4,
replicationcontrollers: 1/20, secrets: 9/10, services: 2/10
```

2. 描述您关注的配额，如 **core-object-counts** 配额：

```
$ oc describe quota core-object-counts -n demoproject
```

输出示例

```
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
configmaps  3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

9.1.8. 配置显式资源配额

在项目请求模板中配置显式资源配额，以便在新项目中应用特定资源配额。

先决条件

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

流程

1. 在项目请求模板中添加资源配额定义：
   - 如果集群中不存在项目请求模板：
     a. 创建 bootstrap 项目模板并将其输出到名为 **template.yaml** 的文件：

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

     b. 在 **template.yaml** 中添加资源配额定义。以下示例定义了名为 'storage-consumption' 的资源配额。定义必须在模板的 **parameter:** 部分前添加：

```
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: storage-consumption
    namespace: ${PROJECT_NAME}
  spec:
    hard:
      persistentvolumeclaims: "10" (1)
      requests.storage: "50Gi" (2)
      gold.storageclass.storage.k8s.io/requests.storage: "10Gi" (3)
```

```
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" 4
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" 5
    bronze.storageclass.storage.k8s.io/requests.storage: "0" 6
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" 7
```

1. 项目中的持久性卷声明总数。
2. 在一个项目中的所有持久性卷声明中，请求的存储总和不能超过这个值。
3. 在一个项目中的所有持久性卷声明中，金级存储类中请求的存储总和不能超过这个值。
4. 在一个项目中的所有持久性卷声明中，银级存储类中请求的存储总和不能超过这个值。
5. 在一个项目中的所有持久性卷声明中，银级存储类中声明总数不能超过这个值。
6. 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 **0**，则 bronze 存储类无法请求存储。
7. 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 **0**，则 bronze 存储类无法创建声明。

c. 通过 **openshift-config** 命名空间中修改的 **template.yaml** 文件创建项目请求模板：

```
$ oc create -f template.yaml -n openshift-config
```

注意

要将配置作为 **kubectl.kubernetes.io/last-applied-configuration**注解包括，将 **--save-config** 选项添加到 **oc create** 命令中。

默认情况下，模板称为 **project-request**。

- 如果项目请求模板已在集群中存在：

注意

如果您使用配置文件以声明性或必要方式管理集群中的对象，请使用这些文件编辑现有项目请求模板。

a. 列出 **openshift-config** 命名空间中的模板：

```
$ oc get templates -n openshift-config
```

b. 编辑现有项目请求模板：

```
$ oc edit template <project_request_template> -n openshift-config
```

c. 将资源配额定义（如前面的 **storage-consumption** 示例）添加到现有模板中。定义必须在模板的 **parameter:** 部分前添加。

2. 如果您创建了项目请求模板，在集群的项目配置资源中引用它：

   a. 访问项目配置资源进行编辑：

      - 使用 web 控制台：

        i. 导航至 **Administration → Cluster Settings** 页面。

        ii. 单击 **Configuration** 以查看所有配置资源。

        iii. 找到 **Project** 的条目，并点击 **Edit YAML**。

      - 使用 CLI：

        i. 编辑 **project.config.openshift.io/cluster** 资源：

        ```
        $ oc edit project.config.openshift.io/cluster
        ```

   b. 更新项目配置资源的 **spec** 部分，使其包含 **projectRequestTemplate** 和 **name** 参数。以下示例引用了默认项目请求模板（名称为 **project-request**）：

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   # ...
   spec:
     projectRequestTemplate:
       name: project-request
   ```

3. 验证在创建项目时是否应用了资源配额：

   a. 创建一个项目：

   ```
   $ oc new-project <project_name>
   ```

   b. 列出项目的资源配额：

   ```
   $ oc get resourcequotas
   ```

   c. 详细描述资源配额：

   ```
   $ oc describe resourcequotas <resource_quota_name>
   ```

9.2. 跨越多个项目的资源配额

多项目配额由 **ClusterResourceQuota** 对象定义，允许在多个项目之间共享配额。对每个选定项目中使用的资源量进行合计，使用合计值来限制所有选定项目中的资源。

本指南介绍了集群管理员如何在多个项目间设置和管理资源配额。

9.2.1. 在创建配额过程中选择多个项目

在创建配额时，您可以根据注解选择和/或标签选择来同时选择多个项目。

流程

1. 要根据注释选择项目，请运行以下命令：

```
$ oc create clusterquota for-user \
    --project-annotation-selector openshift.io/requester=<user_name> \
    --hard pods=10 \
    --hard secrets=20
```

这会创建以下 **ClusterResourceQuota** 对象：

```
apiVersion: quota.openshift.io/v1
kind: ClusterResourceQuota
metadata:
  name: for-user
spec:
  quota: 1
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: 2
      openshift.io/requester: <user_name>
    labels: null 3
status:
  namespaces: 4
  - namespace: ns-one
    status:
      hard:
        pods: "10"
        secrets: "20"
      used:
        pods: "1"
        secrets: "9"
  total: 5
    hard:
      pods: "10"
      secrets: "20"
    used:
      pods: "1"
      secrets: "9"
```

1. 将对所选项目强制执行的 **ResourceQuotaSpec** 对象。
2. 注解的简单键值选择器。
3. 可用来选择项目的标签选择器。
4. 描述每个所选项目中当前配额使用量的各命名空间映射。
5. 所有选定项目的使用量合计。

此多项目配额文档使用默认的项目请求端点控制 **<user_name>** 请求的所有项目。您需要有 10 个 Pod 和 20 个 secret 的限制。

2. 同样，若要根据标签选择项目，请运行以下命令：

```
$ oc create clusterresourcequota for-name \ 1
    --project-label-selector=name=frontend \ 2
    --hard=pods=10 --hard=secrets=20
```

1 **clusterresourcequota** 和 **clusterquota** 是同一命令的别名。**for-name** 是 **ClusterResourceQuota** 对象的名称。

2 要根据标签选择项目，请使用 **--project-label-selector=key=value** 格式提供一个键值对。

这会创建以下 **ClusterResourceQuota** 对象定义：

```
apiVersion: quota.openshift.io/v1
kind: ClusterResourceQuota
metadata:
  creationTimestamp: null
  name: for-name
spec:
  quota:
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: null
    labels:
      matchLabels:
        name: frontend
```

9.2.2. 查看适用的集群资源配额

项目管理员无法创建或修改多项目配额来限制自己的项目，但管理员可以查看应用到自己项目的多项目配额文档。项目管理员可以使用 **AppliedClusterResourceQuota** 资源进行此操作。

流程

1. 要查看应用到某一项目的配额，请运行：

```
$ oc describe AppliedClusterResourceQuota
```

输出示例

```
Name:   for-user
Namespace:  <none>
Created:  19 hours ago
Labels:   <none>
Annotations:  <none>
Label Selector: <null>
AnnotationSelector: map[openshift.io/requester:<user-name>]
Resource  Used  Hard
--------  ----  ----
pods      1     10
secrets   9     20
```

9.2.3. 选择粒度

由于在声明配额分配时会考虑锁定，因此通过多项目配额选择的活跃项目数量是一个重要因素。如果在单个多项目配额下选择超过 100 个项目，这可能会给这些项目中的 API 服务器响应造成不利影响。

第 10 章 将配置映射与应用程序搭配使用

配置映射允许您将配置工件与镜像内容分离，从而使容器化应用程序可以移植。

以下部分定义配置映射以及如何创建和使用它们。

10.1. 了解配置映射

许多应用程序需要使用配置文件、命令行参数和环境变量的某些组合来进行配置。在 OpenShift Container Platform 中，这些配置工件与镜像内容分离，以便使容器化应用程序可以移植。

ConfigMap 对象提供了将容器注入到配置数据的机制，同时保持容器与 OpenShift Container Platform 无关。配置映射可用于存储细粒度信息（如个别属性）或粗粒度信息（如完整配置文件或 JSON blob）。

ConfigMap 对象包含配置数据的键值对，这些数据可在 Pod 中消耗或用于存储控制器等系统组件的配置数据。例如：

ConfigMap 对象定义

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: my-namespace
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData:
  bar: L3Jvb3QvMTAw 2
```

1 包含配置数据。

2 指向含有非 UTF8 数据的文件，如二进制 Java 密钥存储文件。以 Base64 格式输入文件数据。

注意

从二进制文件（如镜像）创建配置映射时，您可以使用 **binaryData** 字段。

可以在 Pod 中以各种方式消耗配置数据。配置映射可用于：

- 在容器中填充环境变量值
- 设置容器中的命令行参数
- 填充卷中的配置文件

用户和系统组件可以在配置映射中存储配置数据。

配置映射与 secret 类似，但设计为能更加便捷地支持与不含敏感信息的字符串配合。

配置映射限制
在 pod 中可以消耗它的内容前，必须创建配置映射。

可以编写控制器来容许缺少的配置数据。根据具体情况使用配置映射来参考各个组件。

ConfigMap 对象驻留在一个项目中。

它们只能被同一项目中的 pod 引用。

Kubelet 只支持为它从 API 服务器获取的 pod 使用配置映射。

这包括使用 CLI 创建或间接从复制控制器创建的 pod。它不包括通过 OpenShift Container Platform 节点的 **--manifest-url** 标记、**--config** 标记，或通过 REST API 创建的 pod，因为这些不是创建 pod 的通用方法。

其他资源

- 创建和使用配置映射

10.2. 用例： 在 POD 中使用配置映射

以下小节描述了在 pod 中消耗 **ConfigMap** 对象时的一些用例。

10.2.1. 使用配置映射在容器中填充环境变量

您可以使用配置映射在容器中填充各个环境变量，或从构成有效环境变量名称的所有键填充容器中的环境变量。

例如，请考虑以下配置映射：

有两个环境变量的 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config 1
  namespace: default 2
data:
  special.how: very 3
  special.type: charm 4
```

1 配置映射的名称。

2 配置映射所在的项目。配置映射只能由同一项目中的 pod 引用。

3 4 要注入的环境变量。

带有一个环境变量的ConfigMap

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: env-config (1)
  namespace: default
data:
  log_level: INFO (2)
```

(1) 配置映射的名称。

(2) 要注入的环境变量。

流程

- 您可以使用 **configMapKeyRef** 部分在 pod 中使用此 **ConfigMap** 的键。

  配置为注入特定环境变量的 Pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: (1)
        - name: SPECIAL_LEVEL_KEY (2)
          valueFrom:
            configMapKeyRef:
              name: special-config (3)
              key: special.how (4)
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config (5)
              key: special.type (6)
              optional: true (7)
      envFrom: (8)
        - configMapRef:
            name: env-config (9)
  restartPolicy: Never
```

(1) 从 **ConfigMap** 中拉取指定的环境变量的小节。

(2) 要将键值注入到的 pod 环境变量的名称。

(3) (5) 要从中拉取特定环境变量的 **ConfigMap** 名称。

(4) (6) 要从 **ConfigMap** 中拉取的环境变量。

(7) 使环境变量成为可选。作为可选项，即使指定的 **ConfigMap** 和键不存在，也会启动 pod。

(8) 从 **ConfigMap** 中拉取所有环境变量的小节。

9 要从中拉取所有环境变量的 **ConfigMap** 名称。

当此 pod 运行时，pod 日志包括以下输出：

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

注意

示例输出中没有列出 **SPECIAL_TYPE_KEY=charm**，因为设置了 **optional: true**。

10.2.2. 使用配置映射为容器命令设置命令行参数

您可以通过 Kubernetes 替换语法 **$(VAR_NAME)**，使用配置映射来设置容器中的命令或参数的值。

例如，请考虑以下配置映射：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

流程

- 要将值注入到容器中的一个命令中，使用您要用作环境变量的键。然后，您可以使用 **$(VAR_NAME)** 语法在容器的命令中引用它们。

  配置为注入特定环境变量的 pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
1
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
```

```
            name: special-config
            key: special.type
  restartPolicy: Never
```

1 使用您要用作环境变量的键将值注入到容器中的命令中。

当此 pod 运行时，test-container 容器中运行的 echo 命令的输出如下：

```
very charm
```

10.2.3. 使用配置映射将内容注入卷

您可以使用配置映射将内容注入卷。

ConfigMap 自定义资源(CR)示例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

流程

您可以使用配置映射将内容注入卷中有两个不同的选项。

- 使用配置映射将内容注入卷的最基本方法是在卷中填充键为文件名称的文件，文件的内容是键值：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config 1
  restartPolicy: Never
```

1 包含密钥的文件。

当这个 pod 运行时，cat 命令的输出将是：

```
very
```

- 您还可以控制投射配置映射键的卷中的路径：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key (1)
  restartPolicy: Never
```

(1) 配置映射键的路径。

当这个 pod 运行时，cat 命令的输出将是：

```
very
```

第 11 章 使用 DEVELOPER 视角监控项目和应用程序的指标

Developer 视角中的 Observe 视图提供了监控项目或应用程序指标的选项，如 CPU、内存和带宽使用情况以及网络相关信息。

11.1. 先决条件

- 您已在 OpenShift Container Platform 上创建并部署了应用程序。
- 已登陆到 web 控制台并切换到 Developer 视角。

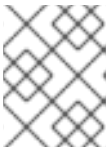
11.2. 监控项目指标数据

在项目中创建应用程序并进行部署后，您可以使用 web 控制台中的 Developer 视角来查看项目的指标数据。

步骤

1. 进入 Observe 以查看项目的 Dashboard, Metrics, Alerts, 和 Events。
2. 可选： 使用 Dashboard 选项卡查看显示以下应用程序指标的图表：
   - CPU 用量
   - 内存用量
   - 带宽消耗
   - 网络相关信息，如传输和接收的数据包率以及丢弃的数据包速率。

   在 Dashboard 选项卡中，您可以访问 Kubernetes 计算资源仪表板。

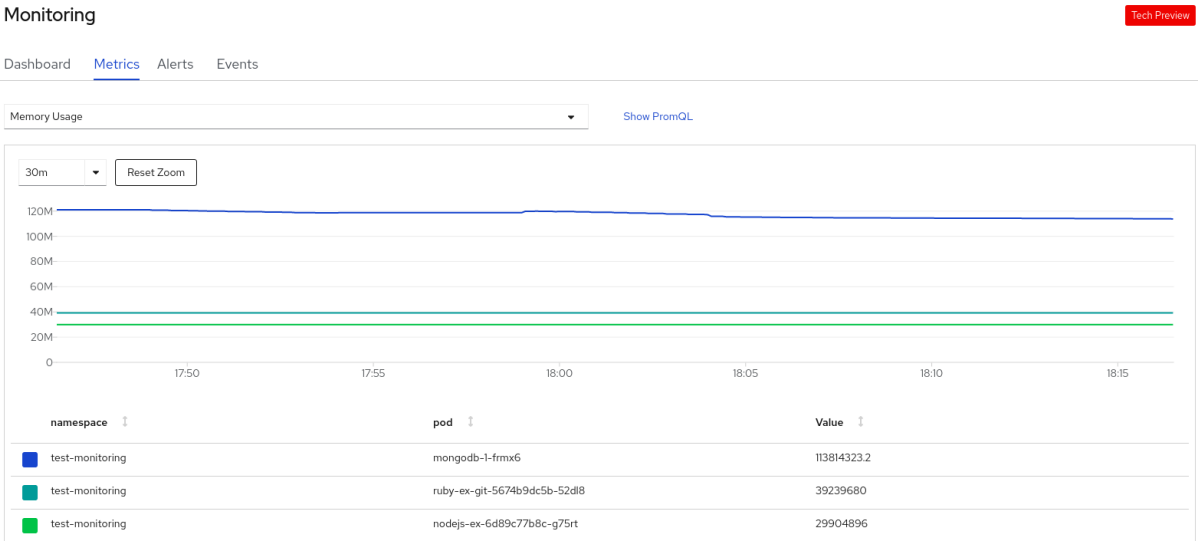
   注意

   在 Dashboard 列表中，默认选择 Kubernetes / Compute Resources / Namespace (Pods) 仪表板。

   使用以下选项查看详情：

   - 从 Dashboard 列表中选择一个仪表板，以查看过滤的指标。选择时，所有仪表板会生成额外的子菜单，但 Kubernetes / Compute Resources / Namespace(Pods)除外。
   - 通过 Time Range 列表选择要捕获数据的时间段。
   - 通过选择 Time Range 列表中的**自定义时间范围**来设置自定义时间范围。您可以输入或选择 From 和 To 的日期和时间。单击 Save 以保存自定义时间范围。
   - 通过 Refresh Interval 列表选择数据刷新的间隔。
   - 将光标锁定在相关图形上，以查看特定 pod 的详细信息。
   - 点每个图形右上角的 Inspect 查看任何特定图形详情。图形详情会出现在 Metrics 选项卡中。
3. 可选：使用 Metrics 选项卡查询所需的项目指标数据。

图 11.1. 监控指标数据



a. 在 **Select Query** 列表中，选择一个选项来过滤项目所需的详细信息。图中显示了经过过滤的、项目中所有应用程序 pod 的指标数据。项目中的 pod 也列在下方。

b. 从 pod 列表中，清除带颜色的方框，删除特定 pod 的指标，进一步过滤查询结果。

c. 点击 **Show PromQL** 查看 Prometheus 查询。您可以使用提示来定制查询并过滤您要查看的命名空间指标来进一步修改查询。

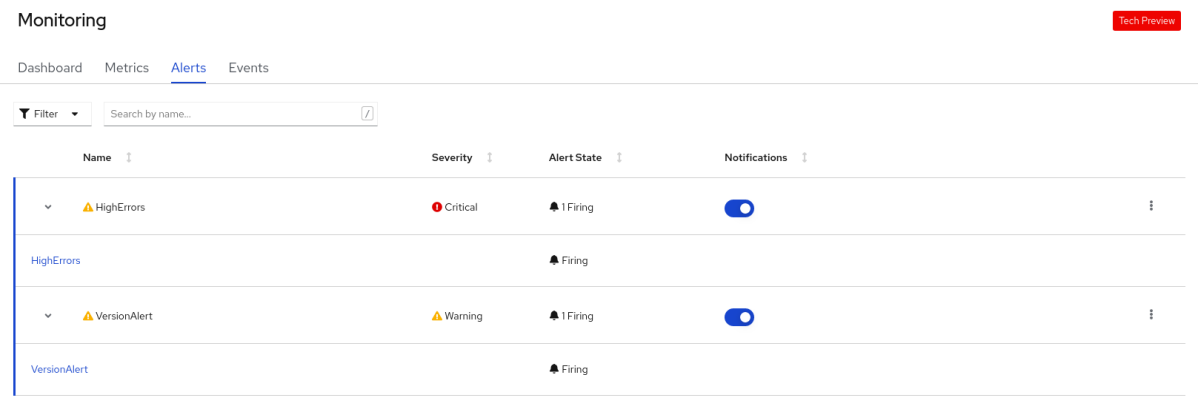
d. 使用下拉列表为要显示的数据设置时间范围。您可以随时点 **Reset Zoom** 把它设置回默认的时间范围。

e. 可选：在 **Select Query** 列表中，选择 **Custom Query** 来创建自定义 Prometheus 查询并过滤相关指标。

4. 可选： 使用 **Alerts** 选项卡执行以下任务：

- 请参阅触发项目中应用程序警报的规则。
- 确定项目中触发警报。
- 如果需要，静默此类警报。

图 11.2. 监控警报



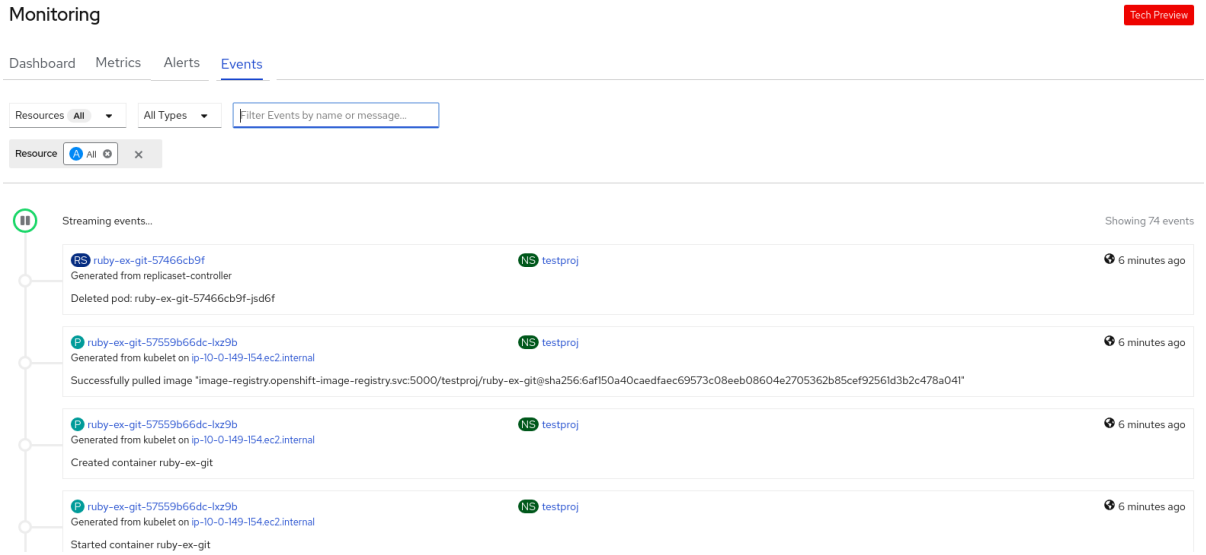
使用以下选项查看详情：

- 使用 **Filter** 列表根据 **Alert State** 和 **Severity** 来过滤警报。

- 点击警报进入该警报的详情页面。在 **Alerts Details** 页面中，您可以点击 **View Metrics** 来查看警报的指标。
- 使用 **Notifications** 切换附加警报规则来静默该规则的所有警报，然后从 **Silence for** 类别中选择静默警报的时间。您必须具有编辑警报的权限才能使用 **Notifications**。
- 使用附加警报  规则的 **Options** 菜单来查看警报规则的详情。

5. 可选： 使用 **Events** 选项卡查看项目的事件。

图 11.3. 监控事件



您可以使用以下选项过滤显示的事件：

- 在 **Resources** 列表中，选择一个资源来查看该资源的事件。
- 在 **All Types** 列表中，选择一个事件类型来查看与该类型相关的事件。
- 使用 **Filter events by name or message**字段搜索特定事件 。

11.3. 监控应用程序的指标数据

在项目中创建应用程序并进行部署后，您可以使用 **Developer** 视角中的 **Topology** 视图来查看应用程序的警报和指标。**Topology** 视图的工作负载节点上会显示应用程序的关键和警告警报。

流程

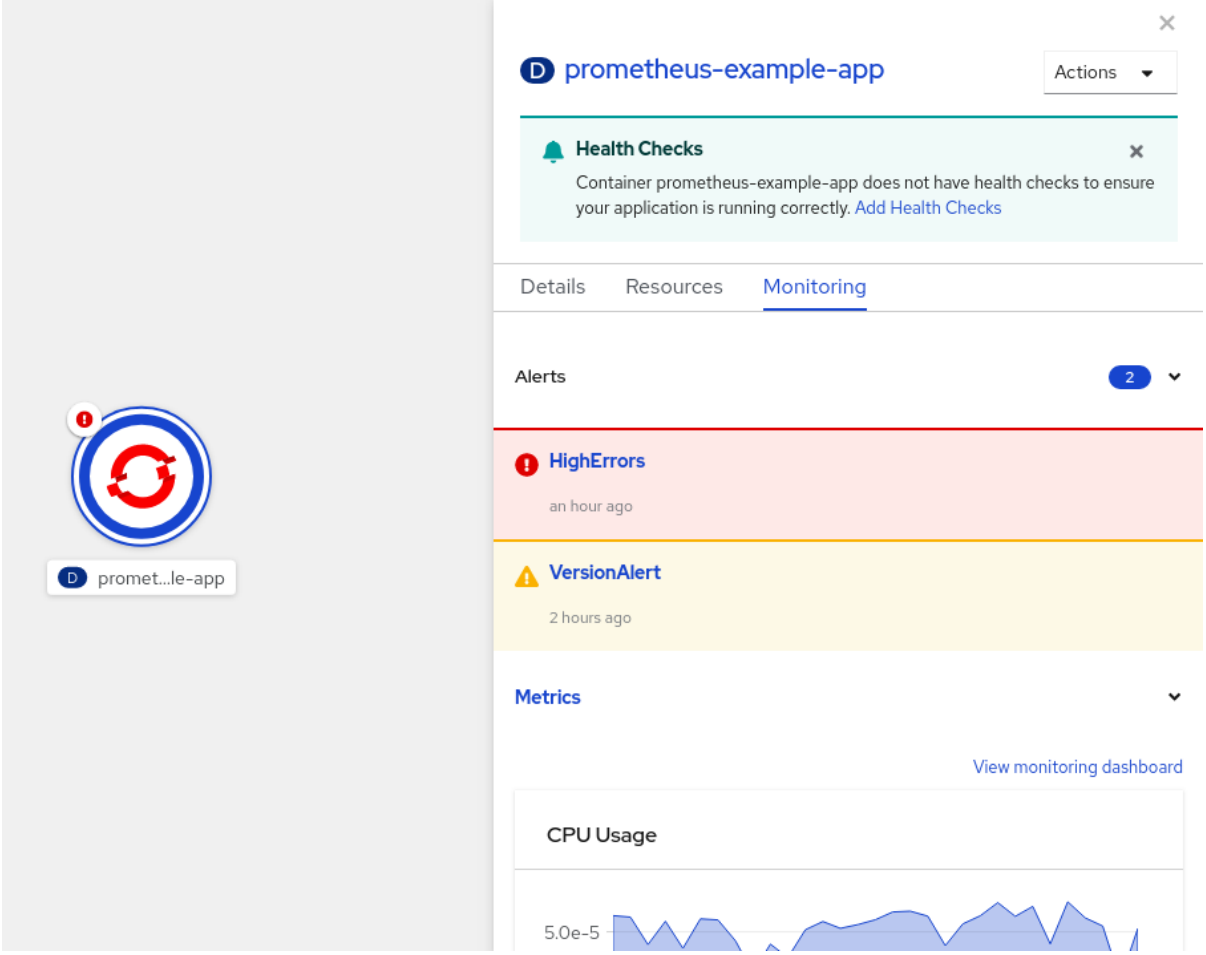
查看工作负载的警报：

1. 在 **Topology** 视图中，点击工作负载查看右侧面板中的工作负载详情。
2. 点 **Observe** 选项卡查看应用程序的关键和警告警告 ; 指标数据图,如 CPU、内存和带宽使用情况 ; 以及应用程序的所有事件。

注意

Topology 视图中仅显示 **Firing** 状态的关键和警告警报。处于 **Silenced**、**Pending** 和 **Not Firing** 状态的警报不会被显示。

图 11.4. 监控应用程序指标数据



a. 点击右侧面板中列出的警报查看 **Alert Details** 页面中的警报详情。

b. 点击任意图表进入 **Metrics** 选项卡，查看应用程序的详细指标。

c. 点击 **View monitoring dashboard** 查看该应用程序的监控仪表板。

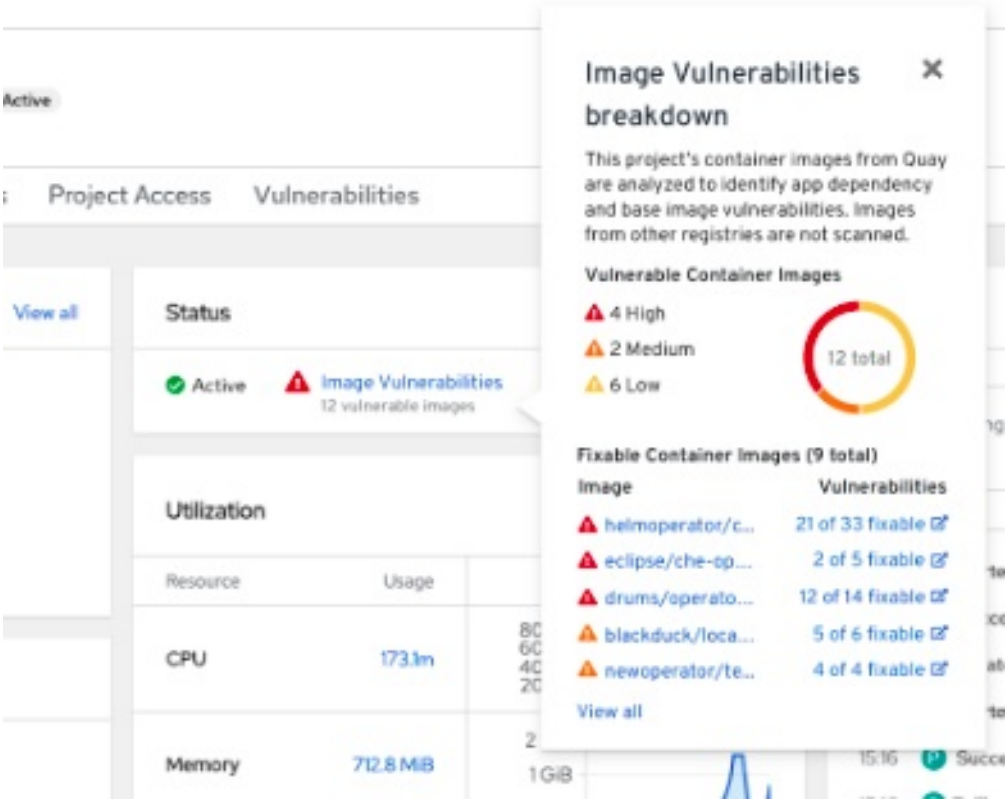
11.4. 镜像漏洞分类

在 **Developer** 视角中，项目仪表板显示 **Status** 部分中的 **Image Vulnerabilities** 链接。使用此链接，您可以查看**镜像安全漏洞分**类窗口，其中包含有关存在安全漏洞的容器镜像的详细信息，以及可修复的容器镜像。图标颜色表示严重性：

- 红色：高优先级。立即修复。
- 橙色：中等优先级。可在高优先级漏洞解决后进行修复。
- 黄色：低优先级。在高优先级和中等优先级漏洞解决后进行修复。

根据严重性级别，您可以对漏洞进行优先级排序，并以有组织的方式修复漏洞。

图 11.5. 查看镜像漏洞



11.5. 监控应用程序和镜像漏洞指标

在项目中创建应用程序并进行部署后，使用 web 控制台中的 **Developer** 视角来查看集群中的应用程序依赖项漏洞的指标。指标可帮助您详细分析以下镜像漏洞：

- 所选项目中存在漏洞镜像的总计数
- 基于所选项目中所有存在安全漏洞镜像的严重性计数
- 分为严重性，以获取漏洞计数、可修复的漏洞计数和每个存在安全漏洞的受影响 pod 数量

先决条件

- 您已从 Operator Hub 安装了 Red Hat Quay Container Security Operator。

注意

Red Hat Quay Container Security operator 通过扫描 quay registry 中的镜像来检测漏洞。

流程

1. 有关镜像漏洞的一般信息，请在 **Developer** 视角的导航面板中点 **Project** 来查看项目仪表板。
2. 在 **Status** 部分中，点 **Image Vulnerabilities**。打开的窗口会显示详细信息，如 **Vulnerable Container Images** 和 **可修复的容器镜像**。
3. 如需详细的漏洞概述，请点项目仪表板上的 **Vulnerabilities** 选项卡。
    a. 要获取有关镜像的更多详细信息，请点其名称。

b. 查看**详情**选项卡中所有漏洞的默认图表。

c. 可选：点切换按钮查看特定类型的漏洞。例如，点 **App dependency** 以查看特定于应用程序依赖项的漏洞。

d. 可选：您可以根据漏洞的**严重性**和类型进行过滤，或根据**严重性**、**软件包**、类型、**源**、**当前版本** 以及**在其中修复的版本**对列表进行排序。

e. 点**漏洞**获取其关联的详情：

- **基础镜像**漏洞显示红帽安全公告 (RHSA) 的信息。
- **应用程序依赖关系**漏洞显示 Snyk 安全应用程序中的信息。

11.6. 其他资源

- 监控概述

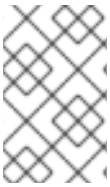
第 12 章 使用健康检查来监控应用程序的健康状态

在软件系统中，组件可能会变得不健康，原因可能源自连接暂时丢失、配置错误或外部依赖项相关问题等临时问题。OpenShift Container Platform 应用程序具有若干选项来探测和处理不健康的容器。

12.1. 了解健康检查

一个健康检查使用就绪度、存活度和启动健康检查的组合来定期对正在运行的容器执行诊断。

您可以在包含要执行健康检查的容器的 pod 规格中包括一个或多个探测。



注意

如果要在现有 pod 中添加或编辑健康检查，您必须编辑 pod **DeploymentConfig** 对象，或者在 web 控制台中使用 **Developer** 视角。您不能使用 CLI 添加或编辑现有 pod 的健康检查。

就绪度探测

*就绪度探测（readiness probe）*决定容器是否准备好接受服务请求。如果容器就绪度探测失败，kubelet 会从可用服务端点列表中移除 pod。
失败后，这个探测将继续检查 pod。如果 pod 可用，kubelet 会将 pod 添加到可用服务端点列表中。

存活度健康检查

*存活度探测（liveness probe）*决定容器是否仍然在运行。如果存活度探测因为死锁等情况而失败，kubelet 会终止容器。pod 会根据其重启策略响应。
例如，在一个 **restartPolicy** 为 **Always** 或 **OnFailure** 的 pod 上的存活度探测会终止并重启容器。

启动探测

*启动探测（startup probe）*指示容器内的应用程序是否启动。其它所有探测在启动成功前被禁用。如果启动探测无法在指定时间内成功，kubelet 会终止容器，容器受 pod **restartPolicy** 影响。
一些应用程序在第一次初始化时可能需要额外时间启动。您可以使用带存活度或就绪度探测的启动探测，延迟这个探测的时长来处理需要长时间启动的系统（使用 **failureThreshold** 和 **periodSeconds** 参数）。

例如，您可以添加一个启动探测，**failureThreshold** 为 30 次失败，**periodSeconds** 为 10 秒，结果为最多 5 分钟（30 * 10 = 300s）。在启动探测第一次成功后，存活度探测会接管。

您可以使用以下测试类型配置存活度、就绪度和启动探测：

- HTTP **GET**：在使用 HTTP **GET** 测试时，测试会使用 web hook 确定容器的健康状态。如果 HTTP 响应代码介于 **200** 与 **399** 之间，则测试可以成功。
  在初始化完成后，可以使用 HTTP **GET** 测试应用程序返回的 HTTP 状态码。

- Container Command：在使用容器命令测试时，探测会在容器内执行命令。如果测试退出的状态为 **0**，代表探测成功。

- TCP socket：当使用 TCP 套接字测试时，探测会尝试为容器打开一个套接字连接。只有在探测可以建立连接时，容器才被视为健康。对于在初始化完成前不会开始监听的应用程序，可以使用 TCP 套接字测试。

您可以配置几个字段来控制探测的行为：

- **initialDelaySeconds**：容器启动后经过这个时间（以秒为单位）后才可以调度探测。默认值为 0。
- **periodSeconds**：执行探测间的延迟时间（以秒为单位）。默认值为 **10**。这个值必须大于 **timeoutSeconds**。
- **timeoutSeconds**：不活跃状态维持了这个时间（以秒为单位）后，探测超时，容器被认为已失败。默认值为 **1**。这个值必须小于 **periodSeconds**。
- **successThreshold**：探测在失败后必须报告成功的次数，才会重置容器状态为成功。对于存活度探测，这个值必须是 **1**。默认值为 **1**。
- **failureThreshold**：这个探测允许失败的次数。默认值为 3。在指定的尝试后：
  - 对于存活度探测，容器被重启
  - 对于就绪度探测，pod 标记为 **Unready**
  - 对于启动探测，容器会终止，并取决于 pod 的 **restartPolicy**

探测示例
以下是在对象规格中显示的不同探测的示例。

就绪度探测示例，它在 pod 规格中带有一个容器命令就绪度探测

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
  - name: goproxy-app (1)
    args:
    image: registry.k8s.io/goproxy:0.1 (2)
    readinessProbe: (3)
      exec: (4)
        command: (5)
        - cat
        - /tmp/healthy
# ...
```

(1) 容器名称。

(2) 要部署的容器镜像。

(3) 就绪度探测。

(4) 容器命令测试。

(5) 在容器上执行的命令。

在 pod 规格中进行容器命令测试的容器命令启动探测和存活度探测示例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
  - name: goproxy-app 1
    args:
    image: registry.k8s.io/goproxy:0.1 2
    livenessProbe: 3
      httpGet: 4
        scheme: HTTPS 5
        path: /healthz
        port: 8080 6
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
    startupProbe: 7
      httpGet: 8
        path: /healthz
        port: 8080 9
      failureThreshold: 30 10
      periodSeconds: 10 11
# ...
```

1. 容器名称。
2. 指定要部署的容器镜像。
3. 存活度探测。
4. HTTP **GET** 测试。
5. 互联网协议： **HTTP** 或 **HTTPS**。默认值为 **HTTP**。
6. 容器正在侦听的端口。
7. 启动探测。
8. HTTP **GET** 测试。
9. 容器正在侦听的端口。
10. 失败后试用探测的次数。
11. 执行探测的秒数。

带有容器命令测试的在 pod 规格中使用超时的存活度探测示例

```
apiVersion: v1
```

```
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
  - name: goproxy-app (1)
    args:
    image: registry.k8s.io/goproxy:0.1 (2)
    livenessProbe: (3)
      exec: (4)
        command: (5)
        - /bin/bash
        - '-c'
        - timeout 60 /opt/eap/bin/livenessProbe.sh
      periodSeconds: 10 (6)
      successThreshold: 1 (7)
      failureThreshold: 3 (8)
# ...
```

(1) 容器名称。

(2) 指定要部署的容器镜像。

(3) 存活度探测。

(4) 探测的类型，这里是一个容器命令探测。

(5) 要在容器内执行的命令行。

(6) 执行探测的频率（以秒为单位）。

(7) 失败后，在显示成功前需要连续成功的次数。

(8) 失败后试用探测的次数。

在部署中使用 TCP 套接字测试的就绪度探测和存活度探测示例

```
kind: Deployment
apiVersion: apps/v1
metadata:
  labels:
    test: health-check
  name: my-application
spec:
# ...
  template:
    spec:
      containers:
        - resources: {}
          readinessProbe: (1)
            tcpSocket:
```

```
        port: 8080
      timeoutSeconds: 1
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 3
    terminationMessagePath: /dev/termination-log
    name: ruby-ex
    livenessProbe: 2
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      timeoutSeconds: 1
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 3
# ...
```

1 就绪度探测。

2 存活度探测。

12.2. 使用 CLI 配置健康检查

要配置就绪度、存活度和启动探测，将一个或多个探测添加到包含要执行健康检查的容器的 pod 规格中

注意

如果要在现有 pod 中添加或编辑健康检查，您必须编辑 pod **DeploymentConfig** 对象，或者在 web 控制台中使用 **Developer** 视角。您不能使用 CLI 添加或编辑现有 pod 的健康检查。

流程

为容器添加探测：

1. 创建 **Pod** 对象来添加一个或多个探测：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
spec:
  containers:
  - name: my-container 1
    args:
    image: registry.k8s.io/goproxy:0.1 2
    livenessProbe: 3
      tcpSocket: 4
        port: 8080 5
      initialDelaySeconds: 15 6
      periodSeconds: 20 7
```

```
    timeoutSeconds: 10 8
  readinessProbe: 9
    httpGet: 10
      host: my-host 11
      scheme: HTTPS 12
      path: /healthz
      port: 8080 13
  startupProbe: 14
    exec: 15
      command: 16
      - cat
      - /tmp/healthy
    failureThreshold: 30 17
    periodSeconds: 20 18
    timeoutSeconds: 10 19
```

1. 指定容器名称。
2. 指定要部署的容器镜像。
3. 可选：创建一个存活度探测。
4. 指定要执行的测试，这里是一个 TCP 套接字测试。
5. 指定容器正在侦听的端口。
6. 指定在调度探测前容器需要已启动的时间（以秒为单位）。
7. 指定执行这个探测的秒数。默认值为 **10**。这个值必须大于 **timeoutSeconds**。
8. 指定一个秒数，在不活跃的时间超过这个值时探测被认为是失败的。默认值为 **1**。这个值必须小于 **periodSeconds**。
9. 可选：创建一个就绪度探测。
10. 指定要执行的测试类型，这里是 HTTP 测试。
11. 指定主机 IP 地址。如果未定义 **host**，则使用 **PodIP**。
12. 指定 **HTTP** 或 **HTTPS**。如果未定义 **scheme**，则使用 **HTTP** 方案。
13. 指定容器正在侦听的端口。
14. 可选：创建一个启动探测。
15. 指定要执行的测试类型，这里是一个容器执行探测。
16. 指定要在容器上执行的命令。
17. 指定失败后测试探测的次数。
18. 指定执行这个探测的秒数。默认值为 **10**。这个值必须大于 **timeoutSeconds**。
19. 指定一个秒数，在不活跃的时间超过这个值时探测被认为是失败的。默认值为 **1**。这个值必须小于 **periodSeconds**。

注意

如果 **initialDelaySeconds** 的值低于 **periodSeconds** 值，则第一个就绪度探测会因计时器的问题在两个阶段间进行。

timeoutSeconds 值必须小于 **periodSeconds** 值。

2. 创建 **Pod** 对象：

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

3. 验证健康检查 pod 的状态：

```
$ oc describe pod my-application
```

输出示例

```
Events:
  Type    Reason     Age   From                                Message
  ----    ------     ----  ----                                -------
  Normal  Scheduled  9s    default-scheduler                   Successfully assigned openshift-
logging/liveness-exec to ip-10-0-143-40.ec2.internal
  Normal  Pulling    2s    kubelet, ip-10-0-143-40.ec2.internal  pulling image
"registry.k8s.io/liveness"
  Normal  Pulled     1s    kubelet, ip-10-0-143-40.ec2.internal  Successfully pulled image
"registry.k8s.io/liveness"
  Normal  Created    1s    kubelet, ip-10-0-143-40.ec2.internal  Created container
  Normal  Started    1s    kubelet, ip-10-0-143-40.ec2.internal  Started container
```

以下是重启容器失败的探测的输出：

不健康容器存活度检查输出示例

```
$ oc describe pod pod1
```

输出示例

```
....

Events:
  Type     Reason          Age                From                                    Message
  ----     ------          ----               ----                                    -------
  Normal   Scheduled       <unknown>                                                  Successfully
assigned aaa/liveness-http to ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
  Normal   AddedInterface  47s                multus                                  Add eth0
[10.129.2.11/23]
  Normal   Pulled          46s                kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 773.406244ms
  Normal   Pulled          28s                kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 233.328564ms
  Normal   Created         10s (x3 over 46s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Created container liveness
  Normal   Started         10s (x3 over 46s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
```

```
Started container liveness
  Warning  Unhealthy       10s (x6 over 34s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-
snzrj  Liveness probe failed: HTTP probe failed with statuscode: 500
  Normal   Killing         10s (x2 over 28s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Container liveness failed liveness probe, will be restarted
  Normal   Pulling         10s (x3 over 47s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Pulling image "registry.k8s.io/liveness"
  Normal   Pulled          10s                kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 244.116568ms
```

12.3. 使用 DEVELOPER 视角监控应用程序的健康状态

您可以使用 **Developer** 视角为容器添加三类健康探测，以确保应用程序健康：

- 使用就绪（Readiness）探测检查容器是否准备好处理请求。
- 使用存活（Liveness）探测检查容器是否在运行。
- 使用启动（Startup）探测检查容器内的应用程序是否已启动。

在创建和部署应用程序时，或部署应用程序后，可以添加健康检查。

12.4. 使用 DEVELOPER 视角编辑健康检查

您可以使用 **Topology** 视图来编辑添加到应用程序中的健康检查、修改它们或添加更多健康检查。

先决条件

- 在 web 控制台中切换到 **Developer** 视角。
- 已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了应用程序。
- 您已将健康检查添加到应用程序中。

流程

1. 在 **Topology** 视图中，右键点击应用程序并选择 **Edit Health Checks**。或者，在侧面面板中点 **Actions** 下拉列表并选择 **Edit Health Checks**。
2. 在 **Edit Health Checks** 页面中：
   - 要删除之前添加的健康探测，请探测旁的减号。
   - 编辑现有探测的参数：
     a. 点以前添加的探测旁的 **Edit Probe** 链接来查看探测的参数。
     b. 根据需要修改参数，并点检查标记保存您的更改。
   - 除了现有健康检查外，要添加新的健康探测，点添加探测链接。例如，要添加一个存活探测来检查容器是否在运行：
     a. 点 **Add Liveness Probe** 会出现包括这个探测的参数的表单。
     b. 根据需要编辑探测参数。

注意

Timeout 值必须小于 **Period** 值。**Timeout** 默认值为 **1**。**Period** 默认值为 **10**。

c. 点表单底部的检查标记。**Liveness Probe Added** 信息会被显示。

3. 点 **Save** 来保存您的修改，并在容器中添加额外的探测。您会进入 **Topology** 视图。
4. 在侧边面板中，点 **Pod** 部分的部署的 pod 来验证是否添加了探测。
5. 在 **Pod Details** 页中，点 **Containers** 部分中列出的容器。
6. 在 **Container Details** 页面中，除了早期存在的探测外，存活探测 - **HTTP Get 10.129.4.65:8080/** 已被添加到容器中。

12.5. 使用 DEVELOPER 视角监控健康检查失败

如果应用程序健康检查失败，您可以使用 **Topology** 视图来监控这些运行状况检查。

先决条件

- 在 web 控制台中切换到 **Developer** 视角。
- 已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了应用程序。
- 您已将健康检查添加到应用程序中。

流程

1. 在 **Topology** 视图中，点应用程序节点来查看侧面板。
2. 点击 **Observe** 选项卡，在 **Events(Warning)** 部分查看健康检查失败。
3. 点 **Events (Warning)** 旁的下箭头来查看与健康检查失败相关的信息。

其他资源

- 有关在 web 控制台中切换到 **Developer** 视角的详情，请参阅关于 **Developer** 视角。
- 如需在创建和部署应用程序时添加健康检查的详细信息，请参阅 使用 Developer 视角创建应用程序中的**高级选项**部分。

第 13 章 编辑应用程序

您可以使用 Topology 视图编辑您创建的应用程序的配置和源代码。

13.1. 先决条件

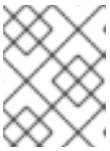
- 在项目中拥有适当的角色和权限，可在 OpenShift Container Platform 中创建和修改应用程序。
- 您已使用 Developer 视角在 OpenShift Container Platform 上创建并部署了应用程序。 。
- 已登陆到 web 控制台并切换到 Developer 视角。

13.2. 使用 DEVELOPER 视角编辑应用程序的源代码

您可以使用 Developer 视角中的 Topology 视图编辑应用程序的源代码。

流程

- 在 Topology 视图中，点击部署的应用程序右下角显示的 Edit Source code 图标，访问源代码并对其进行修改。



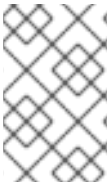
注意

只有使用 From Git、From Catalog 和 From Dockerfile 选项创建了应用程序时，此功能才可用。

如果在集群中安装了 Eclipse Che Operator，则会创建一个 Che 工作区（ ），并定向到工作区编辑源代码。如果没有安装，您将定向到托管您的源代码的 Git 存储库（ ）。

13.3. 使用 DEVELOPER 视角编辑应用程序配置

您可以使用 Developer 视角中的 Topology 视图来编辑应用程序的配置。



注意

当前，只有使用 Developer 视角中的 Add 工作流中的 From Git、Container Image、From Catalog 或 From Dockerfile 选项创建的应用程序配置才可以被编辑。使用 CLI 或 Add 工作流中的 YAML 选项创建的应用程序配置不能被编辑。

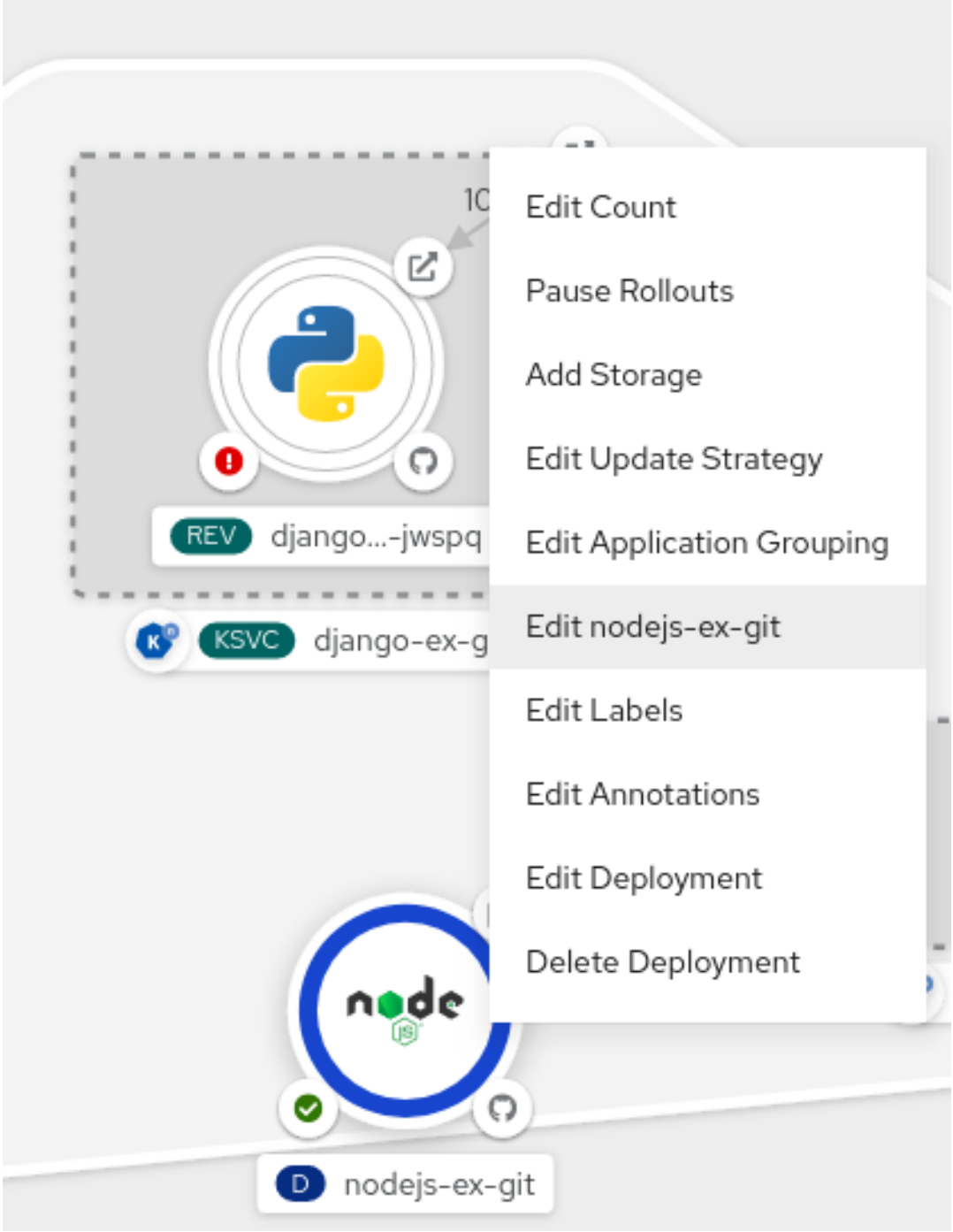
先决条件

确保您已使用 Add 工作流中的 From Git、Container Image、From Catalog或 From Dockerfile 选项创建了应用程序。

流程

1. 当应用程序被创建并显示在 Topology 视图中后，在应用程序上点鼠标右键来查看可用的编辑选项。

图 13.1. 编辑应用程序



2. 点 Edit *application-name* 来查看用来创建应用程序的 Add 工作流。该表单会预先填充创建应用程序时添加的值。

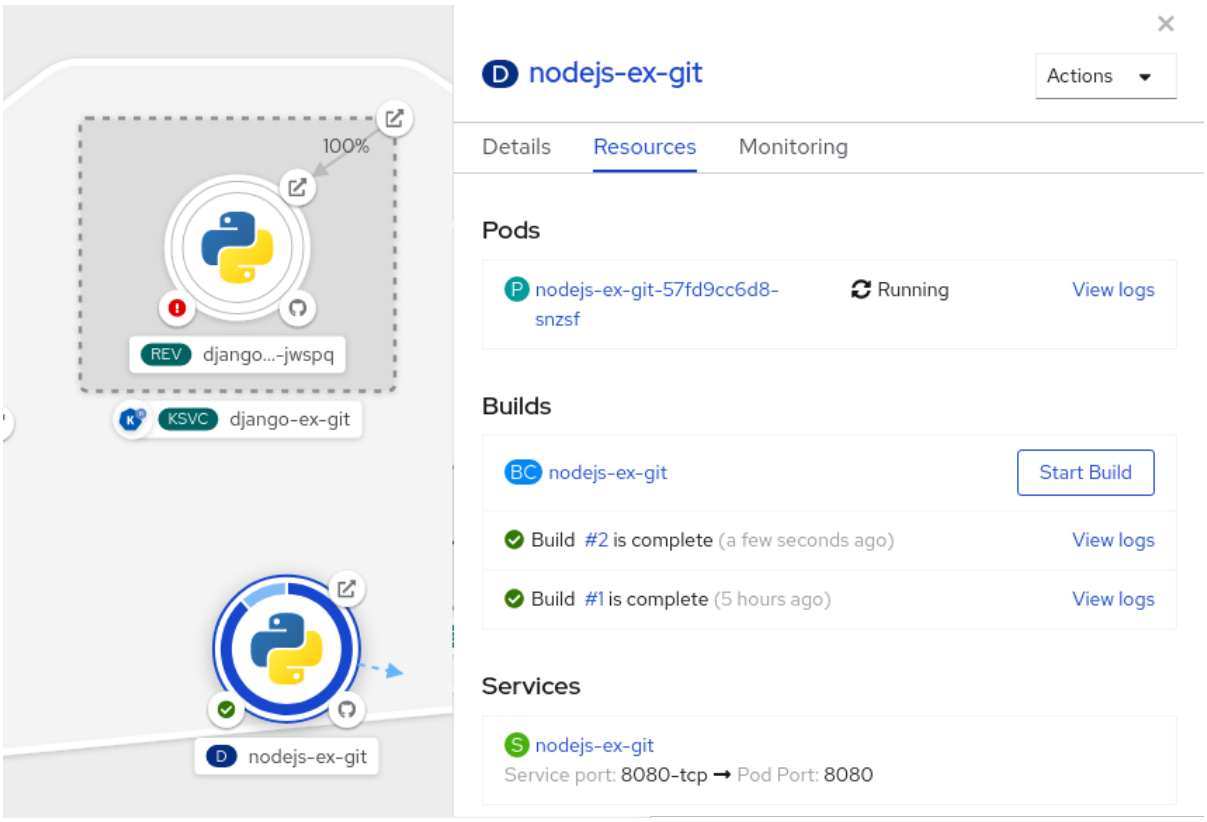
3. 编辑应用程序所需的值。

注意

您不能编辑 General 项中的 Name、CI/CD pipelines 或 Advanced Options 项中的 Create a route to the application项。

4. 点击 Save 重启构建过程并部署新镜像。

图 13.2. 编辑并重新部署应用程序



第 14 章 修剪对象以重新声明资源

随着时间推移，OpenShift Container Platform 中创建的 API 对象可通过常规用户操作（如构建和部署应用程序）积累到集群的 etcd 数据存储中。

集群管理员可以周期性地从集群中修剪不再需要的旧版对象。例如，您可以通过修剪镜像来删除不再使用但仍然占用磁盘空间的旧镜像和旧层。

14.1. 基本修剪操作

CLI 将修剪操作分组到一个通用的父命令下：

```
$ oc adm prune <object_type> <options>
```

这将指定：

- 要对其执行操作的 **<object_type>**，如 **groups**、**builds**、**deployments** 或 **images**。
- 修剪该对象类型所支持的 **<options>**。

14.2. 修剪组

要修剪来自外部提供程序的组记录，管理员可以运行以下命令：

```
$ oc adm prune groups \
    --sync-config=path/to/sync/config [<options>]
```

表 14.1. oc adm prune groups 标记

选项	描述
--confirm	指明应该执行修剪，而不是空运行。
--blacklist	指向组黑名单文件的路径。
--whitelist	指向组白名单文件的路径。
--sync-config	指向同步配置文件的路径。

流程

1. 要查看 prune 命令删除的组，请运行以下命令：

   ```
   $ oc adm prune groups --sync-config=ldap-sync-config.yaml
   ```

2. 要执行修剪操作，请添加 **--confirm** 标志：

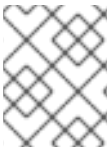
   ```
   $ oc adm prune groups --sync-config=ldap-sync-config.yaml --confirm
   ```

14.3. 修剪部署资源

您可以修剪与系统不再需要的部署关联的资源，因为时间和状态。

以下命令修剪与 **DeploymentConfig** 对象关联的复制控制器 ：

```
$ oc adm prune deployments [<options>]
```



注意

要修剪与 **Deployment** 对象关联的副本集，请使用 **--replica-sets** 标志。这个标志目前还是一个技术预览功能。

表 14.2. oc adm prune deployments 标记

选项	描述
--confirm	指明应该执行修剪，而不是空运行。
--keep-complete=<N>	对于 **DeploymentConfig** 对象，保留状态为 **Complete** 且副本数为零的最后 **N** 个复制控制器。默认值为 **5**。
--keep-failed=<N>	对于 **DeploymentConfig** 对象，保留状态为 **Failed** 的最后 **N** 复制控制器，副本数为零。默认值为 **1**。
--keep-younger-than=<duration>	不修剪存在时间没有超过 **<duration>**（相对于当前时间）的复制控制器。有效度量单位包括纳秒(**ns**)、微秒 (**us**)、毫秒(**ms**)、秒(**s**)、分钟(**m**)和小时(**h**)。默认值为 **60m**。
--orphans	修剪所有不再具有 **DeploymentConfig** 对象、状态为 **Complete** 或 **Failed**、 副本数为零的复制控制器。
--replica-sets=true|false	如果为 **true**，则修剪过程中包含副本集。默认值为 **false**。  **重要** 这个标志是一个技术预览功能。

流程

1. 要查看修剪操作要删除的内容，请运行以下命令：

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

2. 要实际执行修剪操作，请添加 **--confirm** 标志：

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```

14.4. 修剪构建

要修剪系统因为年龄和状态而不再需要的构建，管理员可运行以下命令：

```
$ oc adm prune builds [<options>]
```

表 14.3. oc adm prune builds 标记

选项	描述
--confirm	指明应该执行修剪，而不是空运行。
--orphans	修剪不再有构建配置且状态为 Complete、Failed、Error 或 Canceled 的构建。
--keep-complete=<N>	对于每个构建配置，保留状态为 Complete 的最后 **N** 个构建。默认值为 **5**。
--keep-failed=<N>	对于每个构建配置，保留状态为 failed、error 或 Canceled 的最后 **N** 个构建。默认值为 **1**。
--keep-younger-than=<duration>	不修剪存在时间没有超过 **<duration>**（相对于当前时间）的对象。默认值为 **60m**。

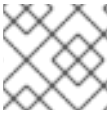
流程

1. 要查看修剪操作要删除的内容，请运行以下命令：

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

2. 要实际执行修剪操作，请添加 **--confirm** 标志：

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```



注意

开发人员可以通过修改其构建配置来启用自动修剪构建。

其他资源

- 执行高级构建 → 修剪构建

14.5. 自动修剪镜像

因为年龄、状态或超过限制，已不再被系统需要的来自 OpenShift 镜像 registry 的镜像会被自动修剪。集群管理员可以配置 Pruning 自定义资源，或挂起它。

先决条件

- 具有 Cluster Administrator 权限。

- 安装 **oc** CLI。

流程

- 验证名为 **imagepruners.imageregistry.operator.openshift.io/cluster** 的项包括以下 **spec** 和 **status** 字段：

```
spec:
  schedule: 0 0 * * * (1)
  suspend: false (2)
  keepTagRevisions: 3 (3)
  keepYoungerThanDuration: 60m (4)
  keepYoungerThan: 3600000000000 (5)
  resources: {} (6)
  affinity: {} (7)
  nodeSelector: {} (8)
  tolerations: [] (9)
  successfulJobsHistoryLimit: 3 (10)
  failedJobsHistoryLimit: 3 (11)
status:
  observedGeneration: 2 (12)
  conditions: (13)
  - type: Available
    status: "True"
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Ready
    message: "Periodic image pruner has been created."
  - type: Scheduled
    status: "True"
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Scheduled
    message: "Image pruner job has been scheduled."
  - type: Failed
    staus: "False"
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Succeeded
    message: "Most recent image pruning job succeeded."
```

(1) **schedule**: **CronJob** 格式的调度。这是可选字段，默认为每日的午夜。

(2) **suspend**: 如果设置为 **true**，**CronJob** 运行的修建操作会被挂起。这是可选字段，默认为 **false**。新集群上的初始值为 **false**。

(3) **keepTagRevisions**：要保留的每个标签的修订版本数量。这是可选字段，默认为 **3**。初始值为 **3**。

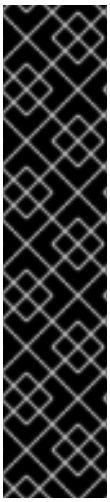
(4) **keepYoungerDuration**：保留比此时间段更早的镜像。这是可选字段。如果没有指定值，则使用 **keepYoungerThan**，或默认值 **60m**（60分钟）。

(5) **keepYoungerThan**：已弃用。与 **keepYoungerThanDuration** 相同，但持续时间被指定为纳秒的整数。这是可选字段。当设置 **keepYoungerThanDuration** 时，会忽略此字段。

(6) **资源**：标准 pod 资源请求和限值。这是可选字段。

(7) **affinity**：标准 pod 关联性。这是可选字段。

8. **nodeSelector** ：标准 pod 节点选择器。这是可选字段。
9. **tolerations**：标准 pod 容限。这是可选字段。
10. **successfulJobsHistoryLimit**：要保留的作业的最大值。必须是 **>= 1** 才能确保报告指标。这是可选字段，默认为 **3**。初始值为 **3**。
11. **failedJobsHistoryLimit**：要保留的最大失败作业数。必须是 **>= 1** 才能确保报告指标。这是可选字段，默认为 **3**。初始值为 **3**。
12. **observedGeneration**: Operator 观察到的生成。
13. **条件**：带有以下类型的标准条件对象：
    - **可用**：指示修剪任务是否已创建。原因可以是 Ready 或 Error。
    - **调度**：指示是否调度了下一个修剪任务。原因可调度、挂起或出错。
    - **失败**：指示最新修剪任务是否失败。



重要

Image Registry Operator 管理修剪器的行为与在 Image Registry Operator 的 **ClusterOperator** 对象上指定的 **managementState** 关联。如果 Image Registry Operator 没有处于 **Managed** 状态，则镜像修剪器仍然可以被 Pruning Custom Resource 配置和管理。

但是，Image Registry Operator 的 **managementState** 会更改部署的镜像修剪器任务的行为：

- **Managed**: 镜像修剪器的 **--prune-registry** 标志被设置为 **true**。
- **Removed**: 镜像修剪器的 **--prune-registry** 标志被设置为 **false**，这意味着它只在 etcd 中修剪镜像元数据。

14.6. 修剪镜像

修剪自定义资源可为 OpenShift 镜像 registry 中的镜像启用自动镜像修剪。管理员可以手工删除因为年龄、状态或超过限值而不再需要的镜像。手动删除镜像的方法有两种：

- 在集群上以一个 **Job** 或 **CronJob** 运行镜像修剪。
- 运行 **oc adm prune images** 命令。

先决条件

- 若要修剪镜像，您必须先以具有访问令牌的用户身份登录到 CLI。用户还必须有集群角色 **system:image-pruner** 或更高级别的角色（如 **cluster-admin**）。
- 公开镜像 registry。

流程

使用以下方法之一可以手工删除因为年龄、状态或超过限值而不再需要的镜像：

- 通过为 **pruner** 服务帐户创建 YAML 文件，在集群中以 **Job** 或 **CronJob** 形式运行镜像修剪，例如：

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

输出示例

```
kind: List
apiVersion: v1
items:
- apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: pruner
    namespace: openshift-image-registry
- apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRoleBinding
  metadata:
    name: openshift-image-registry-pruner
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: system:image-pruner
  subjects:
  - kind: ServiceAccount
    name: pruner
    namespace: openshift-image-registry
- apiVersion: batch/v1
  kind: CronJob
  metadata:
    name: image-pruner
    namespace: openshift-image-registry
  spec:
    schedule: "0 0 * * *"
    concurrencyPolicy: Forbid
    successfulJobsHistoryLimit: 1
    failedJobsHistoryLimit: 3
    jobTemplate:
      spec:
        template:
          spec:
            restartPolicy: OnFailure
            containers:
            - image: "quay.io/openshift/origin-cli:4.1"
              resources:
                requests:
                  cpu: 1
                  memory: 1Gi
              terminationMessagePolicy: FallbackToLogsOnError
              command:
              - oc
              args:
              - adm
              - prune
              - images
```

```
        - --certificate-authority=/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt
        - --keep-tag-revisions=5
        - --keep-younger-than=96h
        - --confirm=true
        name: image-pruner
      serviceAccountName: pruner
```

- 运行 **oc adm prune images [<options>]** 命令：

```
$ oc adm prune images [<options>]
```

除非使用了 **--prune-registry=false**，否则修剪镜像会从集成 registry 中移除数据。

使用 **--namespace** 标志修剪镜像时不移除镜像，只移除镜像流。镜像是没有命名空间的资源。因此，将修剪限制到特定的命名空间会导致无法计算其当前使用量。

默认情况下，集成 registry 会缓存 blob 元数据来减少对存储的请求数量，并提高处理请求的速度。修剪不会更新集成 registry 缓存。在修剪后推送的镜像如果含有修剪的层，它们会被破坏，因为不会推送在缓存中有元数据的已修剪层。因此，您必须重新部署 registry，以便在修剪后清除缓存：

```
$ oc rollout restart deployment/image-registry -n openshift-image-registry
```

如果集成 registry 使用 Redis 缓存，您必须手动清理数据库。

如果无法在修剪后重新部署 registry，那么您必须永久禁用缓存。

oc adm prune images 操作需要 registry 的路由。默认不创建 registry 路由。

Prune images CLI 配置选项表描述了可供 **oc adm prune images <options> 命令使用的选项**。

表 14.4. 修剪镜像 CLI 配置选项

选项	描述
--all	包括没有推送到 registry 但已通过 pullthrough 镜像的镜像。默认为开启。要将修剪限制为已被推送到集成 registry 的镜像，请传递 **--all=false**。
--certificate-authority	与 OpenShift Container Platform 管理的 registry 通信时使用的证书颁发机构文件的路径。默认为来自当前用户配置文件的证书颁发机构数据。如果提供，则发起安全连接。
--confirm	指明应该执行修剪，而不是空运行。这需要具有指向集成容器镜像 registry 的有效路由。如果此命令在集群网络外运行，则必须使用 **--registry-url** 来提供路由。
--force-insecure	谨慎使用这个选项。允许与通过 HTTP 托管或具有无效 HTTPS 证书的容器 registry 进行不安全连接。
--keep-tag-revisions=<N>	对于每个镜像流，每个标签最多保留 **N** 个镜像修订（默认值 **3**）。

选项	描述
--keep-younger-than=<duration>	不修剪相对于当前时间年龄不到 **<duration>** 的镜像。或者，不修剪被相对于当前时间年龄不到 **<duration>** 的其他对象引用的镜像（默认值 **60m**）。
--prune-over-size-limit	修剪超过同一项目中定义的最小限值的每个镜像。此标志不能与 **--keep-tag-revisions** 或 **--keep-younger-than** 结合使用。
--registry-url	联系 registry 时使用的地址。此命令尝试使用由受管镜像和镜像流决定的集群内部 URL。如果失败（registry 无法解析或访问），则需要使用此标志提供一个替代路由。可以在 registry 主机名中加上前缀 **https://** 或 **http://** 来强制执行特定的连接协议。
--prune-registry	此选项与其他选项指定的条件结合，可以控制是否修剪 registry 中与 OpenShift Container Platform 镜像 API 对象对应的数据。默认情况下，镜像修剪同时处理镜像 API 对象和 registry 中对应的数据。 当您只关注移除 etcd 内容时（可能要减少镜像对象的数量，但并不关心清理 registry）或要通过硬修剪 registry 来单独进行操作（可能在 registry 的适当维护窗口期间），此选项很有用处。

14.6.1. 镜像修剪条件

您可以对手动修剪的镜像应用条件。

- 要删除任何由 OpenShift Container Platform 管理的镜像，或删除带有注解 **openshift.io/image.managed**的镜像：
  - 至少在 **--keep-younger-than** 分钟前创建，且当前没有被引用：
    - 在之前 **--keep-younger-than** 分钟内创建的 Pod
    - 在之前 **--keep-younger-than** 分钟内创建的镜像流
    - 运行的 pod
    - 待处理的 pod
    - 复制控制器
    - 部署
    - 部署配置
    - 副本集（Replica set）
    - 构建配置
    - Builds
    - Jobs

        - Cronjobs
        - 有状态的集合
        - **stream.status.tags[].items** 中 **--keep-tag-revisions** 个最新项
    - 超过同一项目中定义的最小限值，且当前没有被引用：
        - 运行的 pod
        - 待处理的 pod
        - 复制控制器
        - 部署
        - 部署配置
        - 副本集（Replica set）
        - 构建配置
        - Builds
        - Jobs
        - Cronjobs
        - 有状态的集合
- 不支持从外部 registry 进行修剪。
- 镜像被修剪后，会从在 **status.tags** 引用了该镜像的所有镜像流中移除对该镜像的所有引用。
- 移除不再被任何镜像引用的镜像层。

注意

--prune-over-size-limit 标志无法与 **--keep-tag-revisions** 或 **--keep-younger-than** 标志结合使用。这样做会返回不允许操作的信息。

与使用一个命令同时进行两个操作相比，把移除 OpenShift Container Platform 镜像 API 对象的操作和从 registry 中删除镜像数据的操作分开进行（使用 **--prune-registry=false** 然后再硬修剪 registry），可以缩减时间窗口且更加安全。但是，计时窗口不会完全剔除。

例如，您仍然可在创建引用某一镜像的 Pod，因为修剪会将该镜像标识为需要修剪。您仍需对在修剪操作期间创建的 API 对象（它可能会引用镜像）加以注意以避免出现引用已删除内容的问题。

重新进行修剪时如果没有使用 **--prune-registry** 选项，或使用 **--prune-registry=true** 选项，则不会修剪之前通过 **--prune-registry=false** 修剪的镜像的镜像 registry 中相关的存储。对于任何使用 **--prune-registry=false** 修剪的镜像，只能通过硬修剪注册表将其从 registry 存储中删除。

14.6.2. 运行镜像修剪操作

流程

1. 查看修剪操作要删除的对象：

   a. 最多保留三个标签修订，并且保证资源（镜像、镜像流和 pod）不长于 60 分钟：

   ```
   $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
   ```

   b. 修剪超过定义的限值的所有镜像：

   ```
   $ oc adm prune images --prune-over-size-limit
   ```

2. 使用上一步中的选项执行修剪操作：

   ```
   $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
   ```

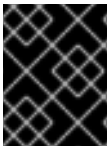
   ```
   $ oc adm prune images --prune-over-size-limit --confirm
   ```

14.6.3. 使用安全或不安全连接

安全连接是首选和推荐的方法。它通过 HTTPS 协议来进行，并且会强制验证证书。若有可能，**prune** 命令始终会尝试使用这种连接。如果不可能，某些情况下会回退到不安全连接，而这存在危险。这时，会跳过证书验证或使用普通 HTTP 协议。

除非指定了 **--certificate-authority**，否则以下情形中允许回退到不安全连接：

1. 使用 **--force-insecure** 选项运行 **prune** 命令。
2. 提供的 **registry-url** 带有 **http://** 架构前缀。
3. 提供的 **registry-url** 是本地链路地址或 **localhost**。
4. 当前用户的配置允许不安全连接。造成的原因可能是用户使用 **--insecure-skip-tls-verify** 登录或在提示时选择不安全连接。



重要

如果 registry 使用有别于 OpenShift Container Platform 所用的证书颁发机构进行保护，则必须通过 **--certificate-authority** 标志来指定。否则，**prune** 命令会出错。

14.6.4. 镜像修剪问题

镜像没有被修剪
如果您的镜像不断积累，且 **prune** 命令只移除您的预期的少许部分，请确保清楚镜像视为修剪候选者时必须要满足的镜像修剪条件。

确保您要移除的镜像在每个标签历史记录中所处的位置高于您选择的标签修订阈值。例如，有一个名为 **sha256:abz** 的陈旧镜像。在您的命名空间中运行以下命令，镜像会在其中标记，它会在一个名为 **myapp** 的镜像流中标记三次：

```
$ oc get is -n <namespace> -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag :=
$is.status.tags}}'\
'{{range $ii, $item := $tag.items}}{{if eq $item.image "sha256:<hash>"}}{{$is.metadata.name}}:
{{$tag.tag}} at position {{$ii}} out of {{len $tag.items}}\n'\
'{{end}}{{end}}{{end}}{{end}}'
```

输出示例

```
myapp:v2 at position 4 out of 5
myapp:v2.1 at position 2 out of 2
myapp:v2.1-may-2016 at position 0 out of 1
```

使用默认选项时，不会修剪该镜像，因为它出现在 **myapp:v2.1-may-2016** 标签历史记录中的位置 **0** 上。要将镜像视为需要修剪，管理员必须：

- 在运行 **oc adm prune images** 命令时指定 **--keep-tag-revisions=0**。

警告

此操作从所有含有基础镜像的命名空间中移除所有标签，除非它们比指定阈值年轻，或者有比指定阈值年轻的对象引用它们。

- 删除所有位置低于修订阈值的 **istag**，即 **myapp:v2.1** 和 **myapp:v2.1-may-2016**。
- 在历史记录中进一步移动镜像，可以通过运行新构建并推送到同一 **istag**，或者标记其他镜像。对于旧版标签，这可能并不是需要的结果。

应该避免在标签的名称中包含某个特定镜像的构建日期或时间，除非镜像必须保留不定的时长。这样的标签通常在历史记录中只有一个镜像，这会永久阻止它们被修剪。

对不安全 registry 使用安全连接
如果您在 **oc adm prune images** 命令的输出中看到类似于如下的消息，这表示您的 registry 未受保护，并且 **oc adm prune images** 客户端尝试使用安全连接：

```
error: error communicating with registry: Get https://172.30.30.30:5000/healthz: http: server gave HTTP response to HTTPS client
```

- 建议的解决方案是保护 registry 的安全。或在，您可以强制客户端使用不安全连接，方法是在命令中附加 **--force-insecure**，但并不建议这样做。

对受保护 registry 使用不安全连接
如果您在 **oc adm prune images** 命令中看到以下错误之一，这表示您的 registry 已设有保护，但签署其证书的证书颁发机构与 **oc adm prune images** 客户端用于连接验证的不同：

```
error: error communicating with registry: Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
error: error communicating with registry: [Get https://172.30.30.30:5000/healthz: x509: certificate signed by unknown authority, Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"]
```

默认情况下，使用存储在用户配置文件中的证书颁发机构数据；与主 API 通信时也是如此。

使用 **--certificate-authority** 选项为容器镜像 registry 服务器提供正确的证书颁发机构。

使用错误的证书颁发机构
以下错误表示，用来为受保护容器镜像 registry 的证书签名的证书颁发机构与客户端使用的不同：

> error: error communicating with registry: Get https://172.30.30.30:5000/: x509: certificate signed by unknown authority

务必通过 **--certificate-authority** 提供正确的证书颁发机构。

作为一种临时解决方案，您可以添加 **--force-insecure** 标签。不过，我们不建议这样做。

其他资源

- 访问 registry
- 公开 registry
- 如需有关如何创建 registry 路由的信息，请参阅 OpenShift Container Platform 中的 Image Registry Operator。

14.7. 硬修剪 REGISTRY

OpenShift Container Registry 可能会积累未被 OpenShift Container Platform 集群的 etcd 引用的 Blob。因此，基本镜像修剪过程对它们无用。它们称为*孤立的 Blob*。

以下情形中可能会出现孤立的 Blob：

- 使用 **oc delete image <sha256:image-id>** 命令手动删除镜像，该命令仅从 etcd 中移除镜像，而不从 registry 存储中移除。
- 守护进程失败引发的推送到 registry 的行为，会造成只上传一些 blob，但不上传其镜像清单（这作为最后一个组件上传）。所有唯一镜像 Blob 变成孤立的 Blob。
- OpenShift Container Platform 因为配额限制而拒绝某一镜像。
- 标准镜像修剪器删除镜像清单，但在删除相关 Blob 前中断。
- registry 修剪器中有一个程序错误，无法移除预定的 Blob，从而导致引用它们的镜像对象被移除，并且 Blob 变成孤立的 Blob。

*硬修剪*registry 是独立于基本镜像修剪的流程，能够让集群管理员移除孤立的 Blob。如果 OpenShift Container registry 的存储空间不足，并且您认为有孤立的 Blob，则应该执行硬修剪。

这应该是罕见的操作，只有在有证据表明创建了大量新的孤立项时才需要。否则，您可以定期执行标准镜像修剪，例如一天一次（取决于要创建的镜像数量）。

流程

从 registry 中硬修剪孤立的 Blob：

1. **登录。**
   使用 CLI，以 **kubeadmin** 或可访问 **openshift-image-registry** 命名空间的其他特权用户身份登录集群 。
2. **运行基本镜像修剪。**
   基本镜像修剪会移除了不再需要的额外镜像。硬修剪不移除自己的镜像。只移除保存在 registry 存储中的 Blob。因此，您应该在硬修剪之前运行此操作。
3. **将 registry 切换成只读模式。**
   如果 registry 不以只读模式运行，任何在修剪的同时发生的推送将会：

- 失败，并导致出现新的孤立项；或者
- 成功，但镜像无法拉取（因为删除了一些引用的 Blob）。

只有 registry 切回到读写模式后，推送才会成功。因此，必须仔细地调度硬修剪。

将 registry 切换成只读模式：

a. 在 **configs.imageregistry.operator.openshift.io/cluster** 中，把 **spec.readOnly** 设置为 **true**：

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster -p '{"spec":
{"readOnly":true}}' --type=merge
```

4. **添加 system:image-pruner 角色。**
用来运行 registry 实例的服务帐户需要额外的权限才能列出某些资源。

a. 获取服务帐户名称：

```
$ service_account=$(oc get -n openshift-image-registry \
    -o jsonpath='{.spec.template.spec.serviceAccountName}' deploy/image-registry)
```

b. 将 **system:image-pruner** 集群角色添加到服务帐户：

```
$ oc adm policy add-cluster-role-to-user \
    system:image-pruner -z \
    ${service_account} -n openshift-image-registry
```

5. **可选：在空运行模式下运行修剪器。**
若要查看会移除多少 Blob，请以空运行模式运行硬修剪器。不会实际进行任何更改。以下示例引用了名为 **image-registry-3-vhndw** 的镜像 registry pod：

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
'/usr/bin/dockerregistry -prune=check'
```

另外，若要获得修剪候选者的准确路径，可提高日志级别：

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

输出示例

```
time="2017-06-22T11:50:25.066156047Z" level=info msg="start prune (dry-run mode)"
distribution_version="v2.4.1+unknown" kubernetes_version=v1.6.1+$Format:%h$
openshift_version=unknown
time="2017-06-22T11:50:25.092257421Z" level=info msg="Would delete blob:
sha256:00043a2a5e384f6b59ab17e2c3d3a3d0a7de01b2cabeb606243e468acc663fa5"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092395621Z" level=info msg="Would delete blob:
sha256:0022d49612807cb348cabc562c072ef34d756adfe0100a61952cbcb87ee6578a"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob:
sha256:0029dd4228961086707e53b881e25eba0564fa80033fbbb2e27847a28d16a37c"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
```

```
time="2017-06-22T11:50:26.673946639Z" level=info msg="Would delete blob:
sha256:ff7664dfc213d6cc60fd5c5f5bb00a7bf4a687e18e1df12d349a1d07b2cf7663"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674024531Z" level=info msg="Would delete blob:
sha256:ff7a933178ccd931f4b5f40f9f19a65be5eeeec207e4fad2a5bafd28afbef57e"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674675469Z" level=info msg="Would delete blob:
sha256:ff9b8956794b426cc80bb49a604a0b24a1553aae96b930c6919a6675db3d5e06"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
...
Would delete 13374 blobs
Would free up 2.835 GiB of disk space
Use -prune=delete to actually delete the data
```

6. **运行硬修剪。**
   在 **docker-registry** pod 的一个正在运行的实例中执行以下命令进行硬修剪。以下示例引用了名为 **image-registry-3-vhndw** 的镜像 registry pod：

   ```
   $ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
   '/usr/bin/dockerregistry -prune=delete'
   ```

   输出示例

   ```
   Deleted 13374 blobs
   Freed up 2.835 GiB of disk space
   ```

7. **将 registry 切回到读写模式。**
   在修剪完成后，registry 可以被切换到读写模式。在 **configs.imageregistry.operator.openshift.io/cluster** 中，把 **spec.readOnly** 设置为 **false**：

   ```
   $ oc patch configs.imageregistry.operator.openshift.io/cluster -p '{"spec":{"readOnly":false}}' -
   -type=merge
   ```

14.8. 运行 CRON 任务

Cron 任务可以修剪成功的任务，但不能正确处理失败的任务。因此，集群管理员应该定期手动清理任务。另外，还应该将 cron 任务的访问权限限制到一小组信任的用户，并且设置适当的配额来阻止 cron 任务创建太多的任务和 Pod。

其他资源

- 使用任务在 Pod 中运行任务
- 跨越多个项目的资源配额
- 使用 RBAC 定义和应用权限

第 15 章 闲置应用程序

集群管理员可以闲置应用程序来减少资源消耗。在成本与资源消耗相关的公有云中部署集群时，这非常有用。

若有任何可扩展的资源不在使用中，OpenShift Container Platform 会发现这些资源并通过将其副本数减少到 **0** 来闲置它们。下一次网络流量定向到这些资源时，通过扩大副本数来取消闲置这些资源，并且继续正常运作。

应用程序由服务以及其他可扩展的资源组成，如部署配置。闲置应用程序的操作涉及闲置所有关联的资源。

15.1. 闲置应用程序

闲置应用程序包括查找与服务关联的可扩展资源（部署配置和复制控制器等）。闲置应用程序时会查找相关的服务，将其标记为空闲，并将资源缩减为零个副本。

您可以使用 **oc idle** 命令来闲置单个服务，或使用 **--resource-names-file** 选项来闲置多个服务。

15.1.1. 闲置一个服务

流程

1. 要闲置一个服务，请运行：

```
$ oc idle <service>
```

15.1.2. 闲置多个服务

如果应用程序横跨一个项目中的一组服务，闲置多个服务会很有用处；或者，可以将闲置多个服务与脚本结合使用，以便批量闲置同一项目中的多个应用程序。

流程

1. 创建一个包含服务列表的文件，每个服务各自列于一行。
2. 使用 **--resource-names-file** 选项闲置这些服务：

```
$ oc idle --resource-names-file <filename>
```

注意

idle 命令仅限于一个项目。若要闲置一个集群中的多个应用程序，可以分别对各个项目运行 **idle** 命令。

15.2. 取消闲置应用程序

当应用程序服务接收网络流量并扩展为之前的状态时，应用程序服务会再次激活。这包括流向服务的流量和通过路由的流量。

也可以通过扩展资源来手动取消闲置应用程序。

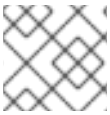
流程

1. 要扩展 DeploymentConfig，请运行：

```
$ oc scale --replicas=1 dc <dc_name>
```

注意

目前，只有默认的 HAProxy 路由器支持通过路由器自动取消闲置。



注意

如果您将 Kuryr-Kubernetes 配置为 SDN，则服务不支持自动取消闲置。

第 16 章 取消应用程序

您可以删除项目中创建的应用程序。

16.1. 使用 DEVELOPER 视角删除应用程序

您可以使用 Developer 视角中的 Topology 视图删除应用程序及其所有关联组件：

1. 点击您要删除的应用程序，即可看到包含应用程序资源详情的侧面板。
2. 点击面板右上角显示的 Actions 下拉菜单，然后选择 Delete Application 即可看到确认对话框。
3. 输入应用程序的名称，并点 Delete 将其删除。

您还可以在要删除的应用程序上点鼠标右键，并点 Delete Application 删除它。

第 17 章 使用 RED HAT MARKETPLACE

Red Hat Marketplace 是一个开源云市场，它可让您轻松发现并访问在公有云和内部运行的基于容器的环境的认证软件。

17.1. RED HAT MARKETPLACE 特性

集群管理员可以使用 Red Hat Marketplace 在 OpenShift Container Platform 上管理软件，授予开发人员部署应用程序实例的自助访问权限，并根据配额与应用程序使用量相关联。

17.1.1. 将 OpenShift Container Platform 集群连接到 Marketplace

集群管理员可以在 OpenShift Container Platform 集群中安装一组连接到 Marketplace 的通用应用程序。它们还可以使用 Marketplace 来跟踪集群针对订阅和配额的使用情况。使用 Marketplace 添加的用户可跟踪其产品的用量，并向相应机构发出帐单。

在 集群连接过程 中，会安装 Marketplace Operator，它用于更新镜像 registry secret、管理目录和报告应用程序用量。

17.1.2. 安装应用程序

集群管理员可以通过 OpenShift Container Platform 的 OperatorHub，或通过 Marketplace web 应用安装 Marketplace 应用程序。

您可以点 web 控制台中的 **Operators => Installed Operators** 来访问已安装的应用程序。

17.1.3. 从不同视角部署应用程序

您可以从 Web 控制台的管理员和 Developer 视角部署 Marketplace 应用程序。

Developer Perspective (开发者视角)
开发人员可以使用 Developer 视角访问新安装的功能。

例如，在安装了数据库 Operator 后，开发人员可从项目中的 catalog 创建实例。数据库用量会被聚合并报告给集群管理员。

此视角不包括 Operator 安装和应用程序使用跟踪。

Administrator perspective（管理员视角）
集群管理员可从管理员的角度来访问 Operator 安装和应用程序使用信息。

它们还可以通过浏览 **Installed Operators** 列表中的自定义资源定义 (CRD) 来启动应用程序实例。