OpenShift Container Platform 4.2

应用程序

在 OpenShift Container Platform 4.2 中创建和管理应用程序

Last Updated: 2020-08-21

OpenShift Container Platform 4.2 应用程序

在 OpenShift Container Platform 4.2 中创建和管理应用程序

法律通告

摘要

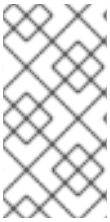
本文档说明如何通过不同方式创建和管理在 OpenShift Container Platform 上运行的用户置备应用程序实例。这包括处理项目以及使用 Open Service Broker API 置备应用程序。

目录

第 1 章 项目

1.1. 处理项目

通过*项目*，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-** 开始的项目是默认项目。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。

1.1.1. 使用 Web 控制台创建项目

如果集群管理员允许，您可以创建新项目。

注意

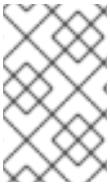
OpenShift Container Platform 认为以 **openshift-** 和 **kube-** 开头的项目是重要的。因此，OpenShift Container Platform 不允许使用 web 控制台创建以 **openshift-** 开头的项目。

流程

1. 浏览至 **Home → Project**。
2. 点击 **Create Project**。
3. 输入项目详情。
4. 点击 **Create**。

1.1.2. 在 Web 控制台中使用 Developer 视角创建项目

您可以使用 OpenShift Container Platform Web 控制台中的 **Developer** 视角在命名空间中创建项目。



注意

以 **openshift-** 和 **kube** 开头的项目用来托管以 Pod 形式运行的集群组件及其他基础架构组件。因此，OpenShift Container Platform 不允许使用 CLI 创建以 **openshift-** 或 **kube-** 开头的项目。集群管理员可以使用 **oc adm new-project** 命令创建这些项目。

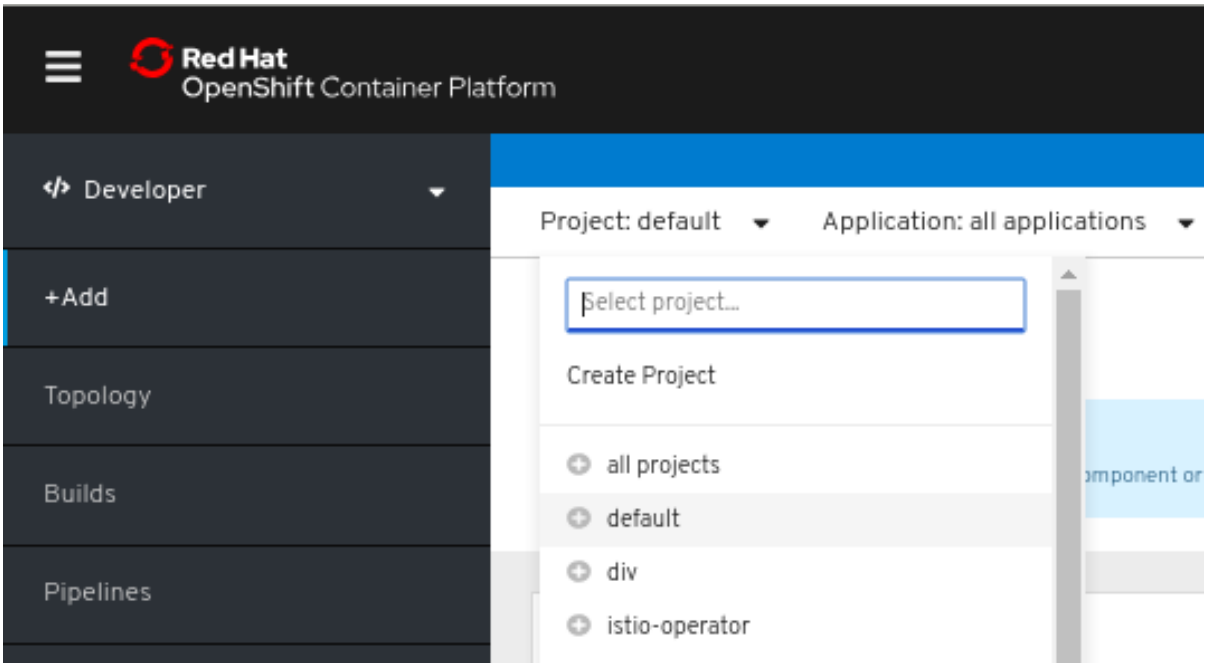
先决条件

- 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。

流程

您可以使用 **Developer** 视角创建项目，如下所示：

1. 在 **Add** 视图中，点击 **Project** 下拉菜单以查看所有可用的项目。选择 **Create Project**。



2. 在 **Create Project** 对话框中，在 **Name** 字段中输入一个唯一名称。例如，在 **Name** 字段中输入 **myproject**，作为项目的名称。
3. 可选：为项目添加 **Display Name** 和 **Description** 详情。
4. 点击 **Create**。

您的项目已创建好，您可以在这个项目中添加应用程序和其他工作负载。

1.1.3. 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。

流程

1. 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。

1.1.4. 使用 Web 控制台查看项目

流程

1. 浏览至 **Home → Project**。
2. 选择要查看的项目。
   在此页面上，点击 **Workloads** 按钮来查看项目中的工作负载。

1.1.5. 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>
```

1.1.6. 添加到项目

流程

1. 在 Web 控制台导航菜单顶部的上下文选择器中，选择 **Developer**。
2. 点击 **+ Add**
3. 在页面顶部，选择要添加到的项目的名称。
4. 点击添加到项目的方法，然后按照工作流操作。

1.1.7. 使用 Web 控制台检查项目状态

流程

1. 浏览至 **Home → Project**。
2. 选择一个项目来查看其状态。

1.1.8. 使用 CLI 检查项目状态

流程

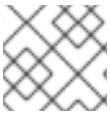
1. 运行：

```
$ oc status
```

此命令提供当前项目的高级概述，以及它的组件和关系。

1.1.9. 使用 web 控制台删除项目

您可以通过 OpenShift Container Platform Web 控制台删除一个项目。



注意

如果您没有删除项目的权限，**Delete Project** 选项将无法使用。

流程

1. 浏览至 **Home → Project**。
2. 找到您要从项目列表中删除的项目。
3. 在项目列表的最右侧，从 Options 菜单中选择 **Delete Project**。  .
4. 打开 **Delete Project** 界面时，在字段中输入要删除的项目名称。
5. 点击 **Delete**。

1.1.10. 使用 CLI 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。

流程

1. 运行：

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

1.2. 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。

1.2.1. 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。

1.2.2. 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
```

1.3. 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。

1.3.1. 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 1.1. 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。

1.3.2. 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. 点击 **Global Configuration**，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。

1.3.3. 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

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

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。

1.3.4. 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. 点击 **Global Configuration**，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:
   ```

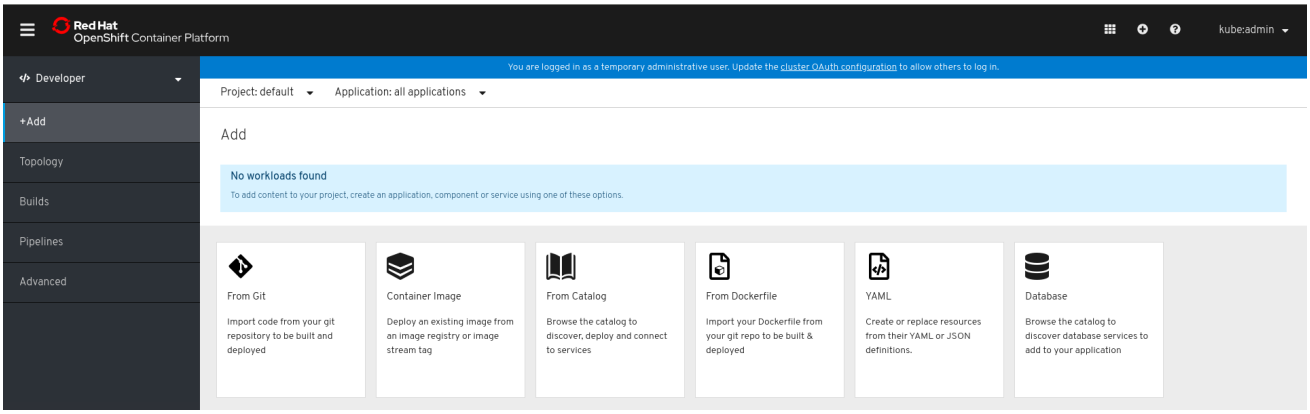
```
  ...
spec:
  projectRequestMessage: To request a project, contact your system administrator at
projectname@example.com.
```

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 2 章 应用程序生命周期管理

2.1. 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：



- **From Git**：使用这个选项时，可从 Git 存储库中导入现有代码库，在 OpenShift Container Platform 上创建、构建和部署应用程序。
- **Container Image**：使用镜像流或 registry 中的现有镜像，将其部署到 OpenShift Container Platform 中。
- **From Catalog**：浏览 **Developer Catalog** 来选择所需的应用程序、服务或 Source-to-Image 构建器，并将其添加到您的项目中。
- **From Dockerfile**：从 Git 存储库导入 Dockerfile，以构建和部署应用程序。
- **YAML**：使用编辑器添加 YAML 或 JSON 定义，以创建和修改资源。
- **Add Database**：查看 **Developer Catalog**，以选择所需的数据库服务并将其添加到应用程序中。

注意

只有集群中安装了 Serverless Operator 时，才会显示上述选项中的无服务器部分。OpenShift Serverless 是一个技术预览功能。

先决条件

要使用 **Developer** 视角创建应用程序，请确认以下几项：

- 已登录 Web 控制台。
- 处于 **Developer** 视角。
- 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。

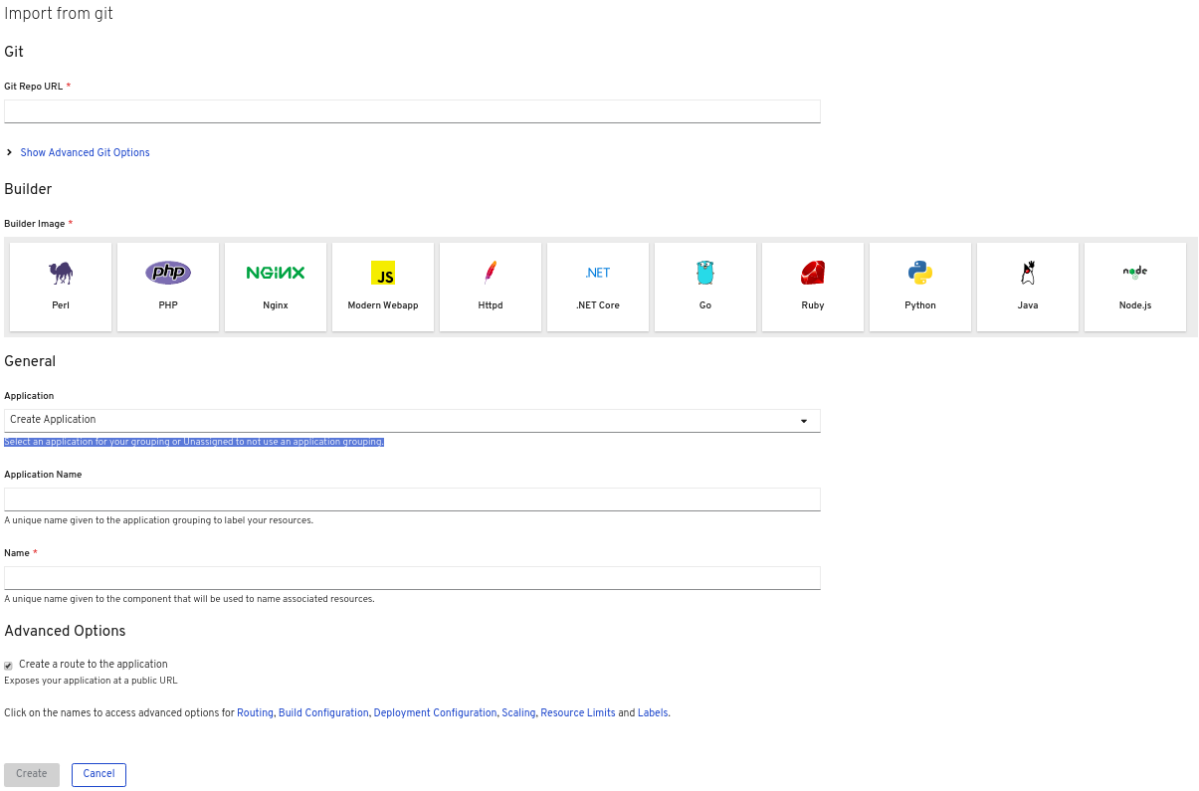
2.1.1. 从 Git 导入代码库来创建应用程序

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

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，如下所示：

流程

1. 在 **Add** 视图中，点击 **From Git** 以查看 **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 nodejs 应用程序的 URL **https://github.com/sclorg/nodejs-ex**。

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. 在 **Builder** 部分中，选择所需的构建器镜像来查看构建器镜像的详情。如果需要，您可以使用 **Builder Image Version** 下拉列表来更改版本。例如，选择 **Node.js** 构建器镜像。

5. 在 **General** 部分中：

   a. 在 **Application Name** 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. 系统会自动填充 **Name** 字段，以标识为此应用程序创建的资源。

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。

6. 在 **Serverless** 部分中选择 **Enable scaling to zero when idle**，以创建可自动将 Pod 数量缩放为零（以防止在空闲时消耗资源）的无服务器应用程序。

注意

只有集群中安装了 Serverless Operator 时，Import from git 表单中才会显示 Serverless 部分。如需进一步了解详细信息，请参阅有关安装 OpenShift Serverless 的文档。

7. Advanced Options 部分中默认选中 Create a route to the application，以便您可以使用公开的 URL 访问应用程序。如果不想通过公共路由公开您的应用程序，可以清除此复选框。

8. 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，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。
     对于无服务器应用程序，Knative Service 管理以上所有路由选项。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**。

   构建和部署配置

   点击 Build Configuration 和 Deployment Configuration 链接，以查看对应的配置选项。其中一些选项默认选中；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 Configuration 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 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。
   对于无服务器应用程序，可以：

   - autoscaler 可设置的 Pod 数设定上限和下限。如果不指定下限，则默认为零。
   - 定义给定时间上每个应用程序实例所需的并发请求数的软限值。它是自动扩展的推荐配置。如果不指定，它将使用集群配置中指定的值。
   - 定义给定时间上每个应用程序实例的并发请求数的硬限值。这在修订模板中进行配置。如果不指定，则默认为集群配置中指定的值。

   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。

9. 点击 Create，以创建应用程序并在 Topology 视图中查看其构建状态。

2.2. 从安装的 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

- 参阅 Operators 指南来进一步了解 Operators 如何工作以及如何将 Operator Lifecycle Manager 集成到 OpenShift Container Platform 中。

2.2.1. 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 4.2 集群
- 管理员已在全集群安装 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。本示例所用项目名为 **my-etcd**。
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 ClusterServiceVersions (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。

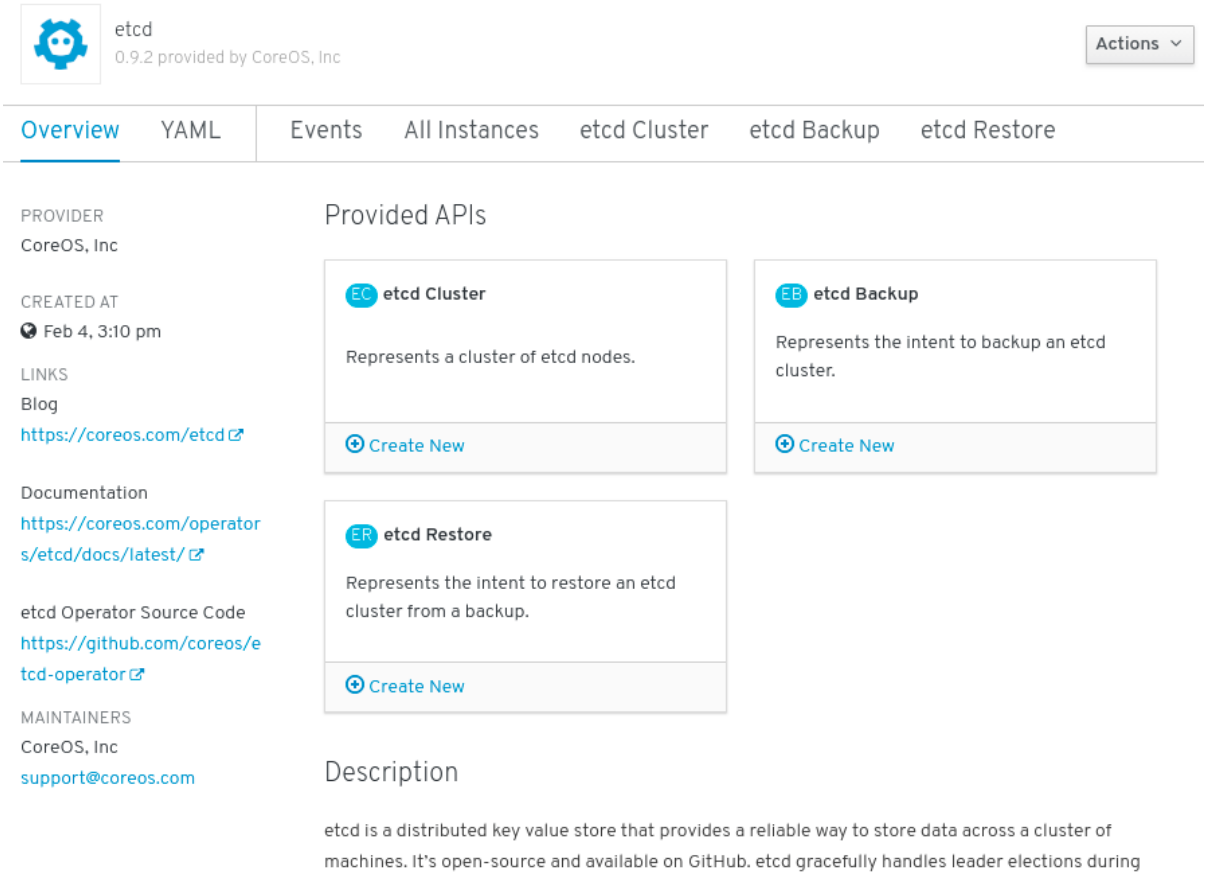
   提示

   使用以下命令从 CLI 获得该列表：

   ```
   $ oc get csv
   ```

3. 进入 **Installed Operators** 页面，点击 **Copied**，然后点击 etcd Operator 查看更多详情和可用操作：

图 2.1. etcd Operator 概述



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类似于内置的原生 Kubernetes 对象（如 **Deployments** 或 **ReplicaSets**），但包含特定于管理 etcd 的逻辑。

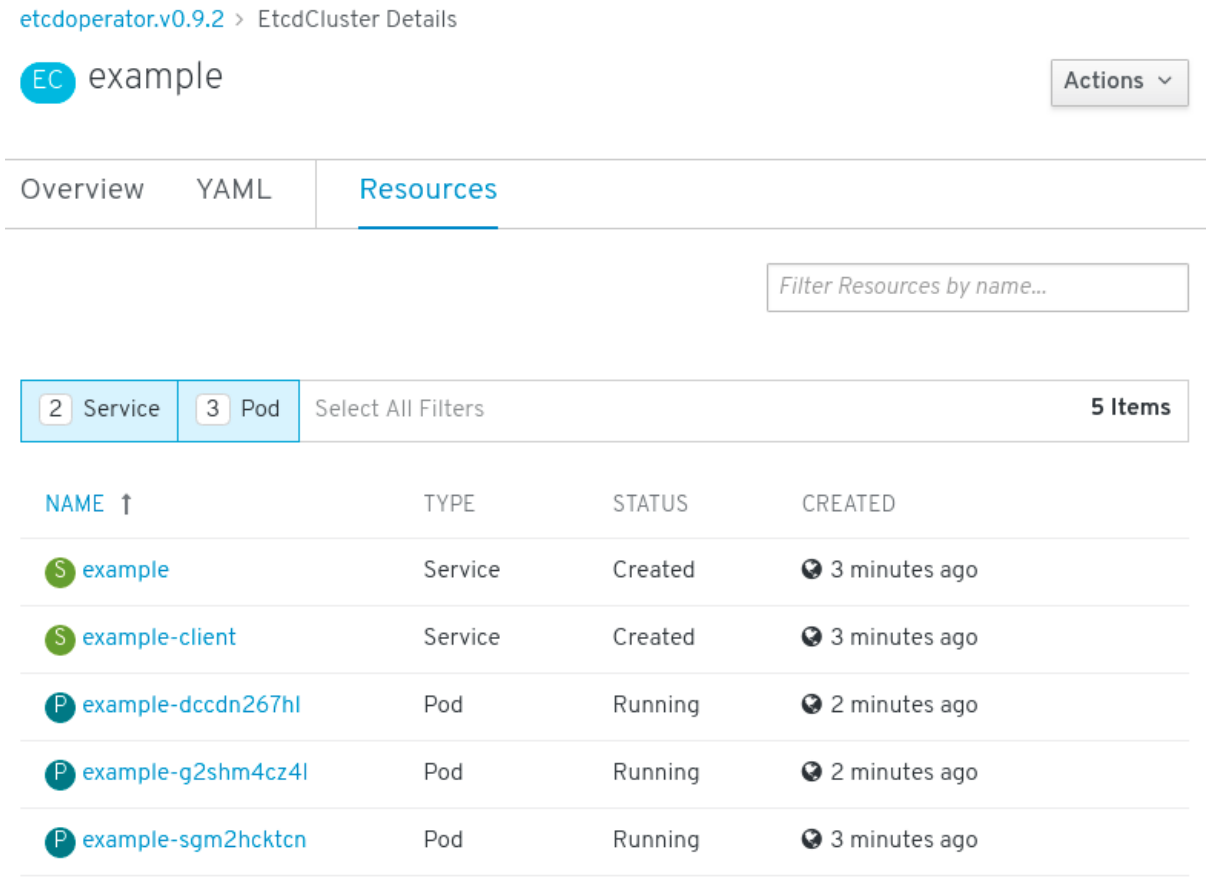
4. 新建 etcd 集群：

   a. 在 **etcd Cluster** API 方框中，点击 **Create New**。

   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、Services 和新 etcd 集群的其他组件。

5. 单击 **Resources** 选项卡，可以看到您的项目现在包含很多由 Operator 自动创建和配置的资源。

图 2.2. etcd Operator 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以轻松地通过其应用程序使用数据库。

2.3. 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。

2.3.1. 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** 命令通常还会创建用来部署新镜像的部署配置，以及为运行您的镜像的部署提供负载均衡访问的服务。

OpenShift Container Platform 会自动检测要使用 **Pipeline** 还是 **Source** 构建策略，如果进行 **Source** 构建，则还检测适当的语言构建器镜像。

2.3.1.1. 本地

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。

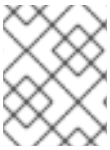
2.3.1.2. 远程

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 **BuildConfig** 中以访问存储库。

您可以通过指定 **--context-dir** 标志来使用源代码存储库的子目录。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
```

2.3.1.3. 构建策略检测

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

否则，它会生成 Source 构建策略。

通过将 **--strategy** 标志设为 **pipeline** 或 **source**，即可覆盖构建策略。

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

注意

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

2.3.1.4. 语言检测

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

表 2.1. new-app 检测的语言

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

检测了语言后，**new-app** 会在 OpenShift Container Platform 服务器上搜索具有与所检测语言匹配的 **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存储库中的源：

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

注意

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

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

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

2.3.2. 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** 或 **-i|--image** 参数明确告知 **new-app** 镜像是容器镜像或镜像流。

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。

2.3.2.1. DockerHub MySQL 镜像

从 Dockerhub MySQL 镜像创建应用程序，例如：

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

2.3.2.2. 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
```

2.3.2.3. 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
```

2.3.3. 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。

从存储的模板创建应用程序，例如：

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

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

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

2.3.3.1. 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=-**：

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

2.3.4. 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 2.2. new-app 输出对象

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

2.3.4.1. 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：

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

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

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

注意

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

2.3.4.2. 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：

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

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

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

2.3.4.3. 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
```

2.3.4.4. 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。

将 **new-app** 工件输出到文件中，编辑工件，再创建工件：

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

2.3.4.5. 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
```

2.3.4.6. 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：

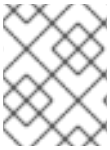
```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

2.3.4.7. 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。

2.3.4.8. 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。要指定哪些镜像要分组在一起，请使用 **+** 分隔符。也可使用 **--group** 命令行参数来指定应分组在一起的镜像。要将源存储库中构建的镜像与其他镜像一起分组，请在组中指定其构建器镜像：

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

将通过源构建的镜像和外部镜像一起部署：

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

```
--group=ruby+mysql
```

2.3.4.9. 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
```

2.4. 使用 TOPOLOGY 视图查看应用程序组成情况

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。

先决条件

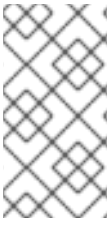
要在 **Topology** 视图中查看应用程序并与之交互，请确保：

- 已登录 Web 控制台。
- 处于 **Developer** 视角。
- 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。 。

2.4.1. 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。创建应用程序后，您会自动定向到 **Topology** 视图，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。

无服务器应用程序显示有 Knative 符号 () 表示。

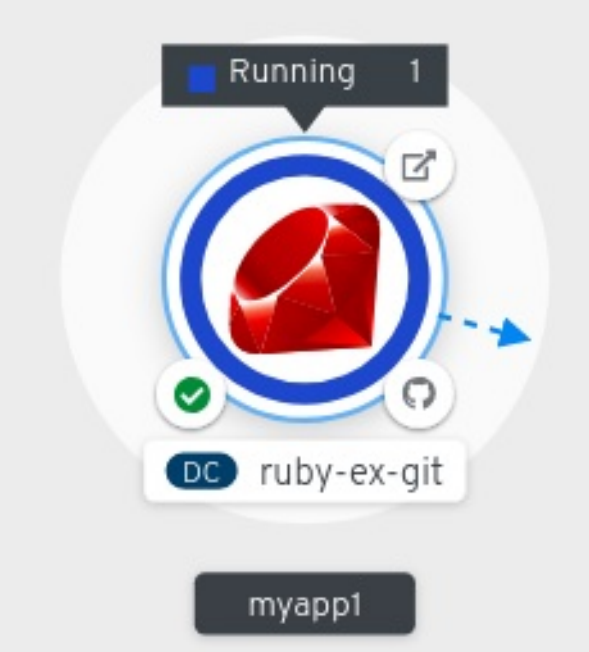


注意

无服务器应用程序需要一些时间才能加载并显示在 **Topology** 视图中。创建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，进行部署并将其显示在 Topology 视图中。如果它是唯一的工作负载，可能会重定向到 **Add** 页面。部署了修订后，无服务器应用程序就会显示在 **Topology** 视图中。

Pod 的状态或阶段由不同的颜色和工具提示来表示：**Running** ( )、**Not Ready** ( )、**Warning**( )、**Failed**( )、**Pending**( )、**Succeeded**( )、**Terminating**( ) 或 **Unknown**( ) 表示。如需有关 Pod 状态的更多信息，请参阅 Kubernetes 文档。

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**。



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：

- **DC**：DeploymentConfigs
- **D**：Deployment
- **SS**：StatefulSet
- **DS**：Daemonset

2.4.2. 与应用程序交互

Web 控制台的 **Developer** 视角中的 **Topology** 视图提供了如下可与应用程序交互的选项：

- 点击 **Open URL** ( )，可查看通过公共 URL 上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。

  如果集群中安装了 **Eclipse Che** Operator，则会创建 Che 工作区 ( )，您也会定向到此工作区来编辑源代码。如果没有安装，您会被定向到 Git 存储库 ( )，即源代码的托管位置。

- 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表示为：**New** ( )、**Pending** ( )、**Running** ( )、**Completed** ( )、**Failed** ( ) 和 **Canceled** ( ) 表示。

2.4.3. 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 **Resources** 选项卡来缩放应用程序 Pod，以及检查构建状态、服务和路由等，如下所示：

- 点击组件节点，以查看右侧的 **Overview** 面板。使用 **Overview** 选项卡可以：
  - 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。

2.4.4. 对应用程序中的多个组件进行分组

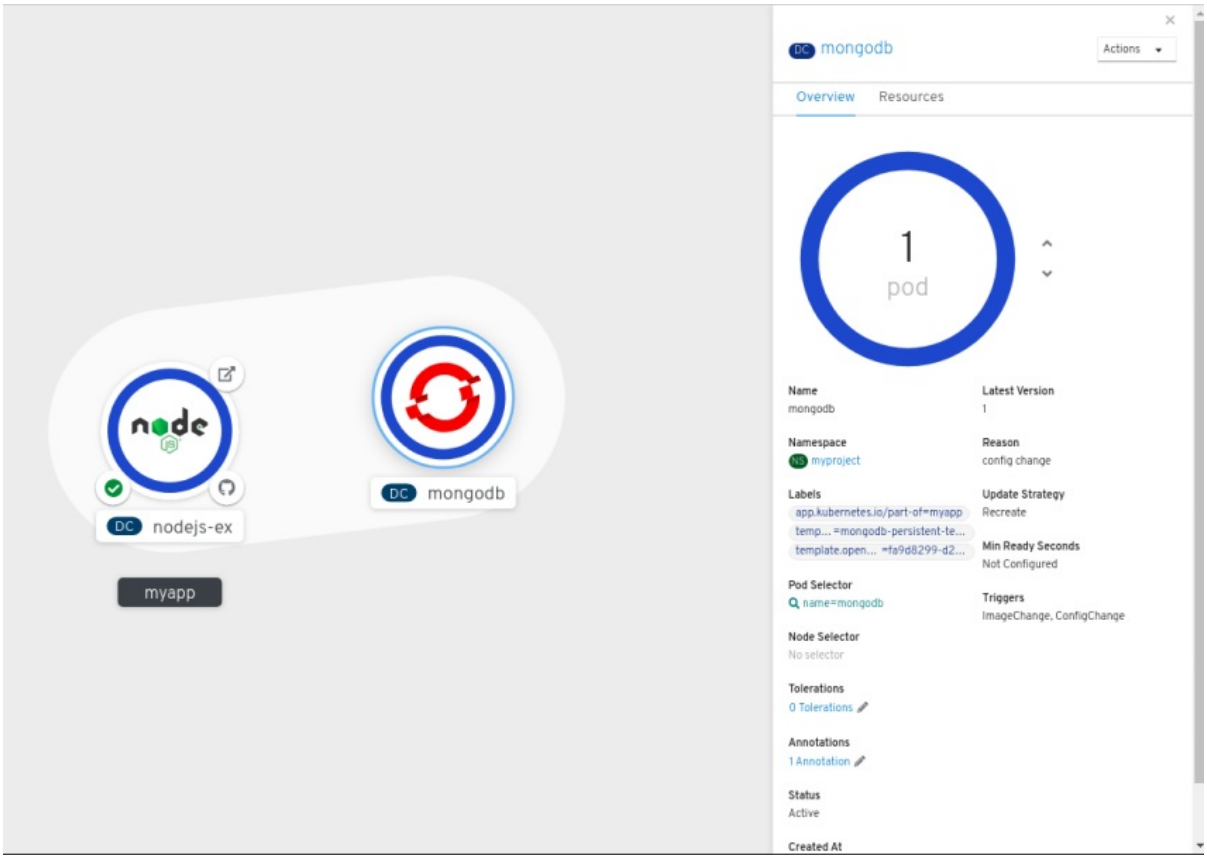
您可以使用 **Add** 页面在项目中添加多个组件或服务，还可使用 **Topology** 页面对应用程序组中的应用程序和资源进行分组。以下流程将 MongoDB 数据库服务添加到具有 Node.js 组件的现有应用程序中。

先决条件

- 确保您已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了 Node.js 应用程序。

流程

1. 在您的项目中创建并部署 MongoDB 服务，如下所示：
   a. 在 **Developer** 视角中，导航到 **Add** 视图，再选择 **Database** 选项来查看 **Developer Catalog**，其包含的多个选项可作为组件或服务添加到应用程序中。
   b. 点击 **MongoDB** 选项以查看该服务的详情。
   c. 点击 **Instantiate Template** 查看使用 MongoDB 服务的详情自动填充的模板，然后点击 **Create** 来创建服务。
2. 在左侧导航面板中，点击 **Topology** 以查看项目中部署的 MongoDB 服务。
3. 要将 MongoDB 服务添加到现有应用程序组中，请选择 **mongodb** Pod 并将其拖到应用程序中；MongoDB 服务会添加到现有应用程序组中。
4. 拖动组件并将其添加到应用程序组中时，会自动将所需的标签添加到组件。点击 MongoDB 服务节点，可看到标签 **app.kubernetes.io/part-of=myapp** 已添加到 **Overview** 面板中的 **Labels** 部分。



另外，您还可以在应用程序中添加组件，如下所示：

1. 要将 MongoDB 服务添加到应用程序中，请点击 **mongodb** Pod 以查看右侧的 **Overview** 面板。
2. 点击面板右上角的 **Actions** 下拉菜单，再选择 **Edit Application Grouping**。
3. 在 **Edit Application Grouping** 对话框中，点击 **Select an Application** 下拉列表，再选择适当的应用程序组。
4. 点击 **Save** 以查看添加到应用程序组中的 MongoDB 服务。

2.4.5. 在应用程序内和应用程序间连接组件

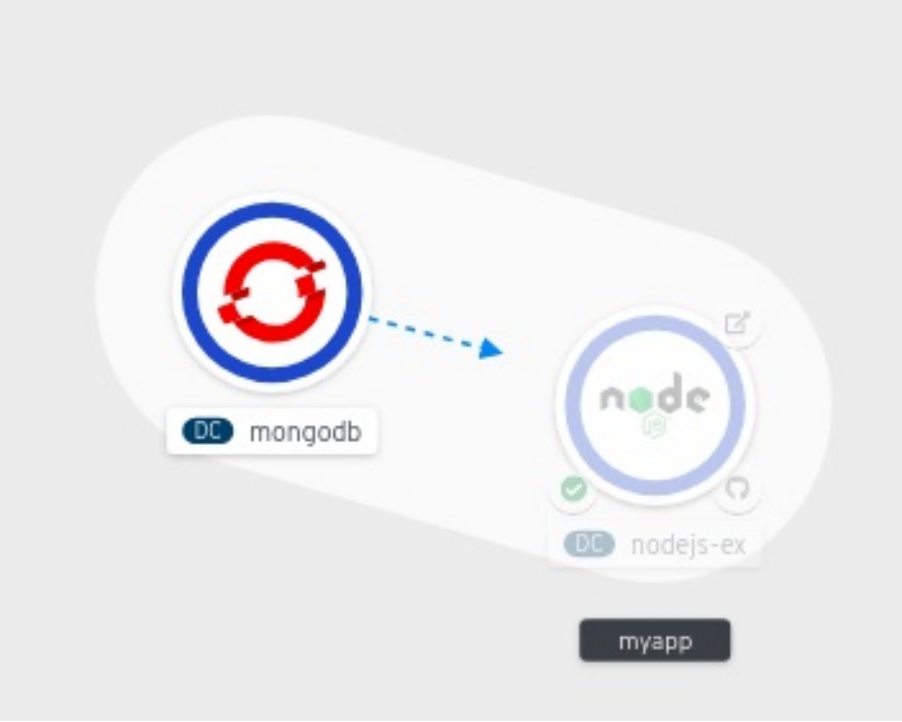
除了对一个应用程序中的多个组件进行分组外，还可以使用 **Topology** 视图来相互连接组件。您可以将 MongoDB 服务与 Node.js 应用程序连接，如下所示：

先决条件

- 确保您已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了 Node.js 应用程序。
- 确保已使用 **Developer** 视角在 OpenShift Container Platform 上创建并部署了 MongoDB 服务。

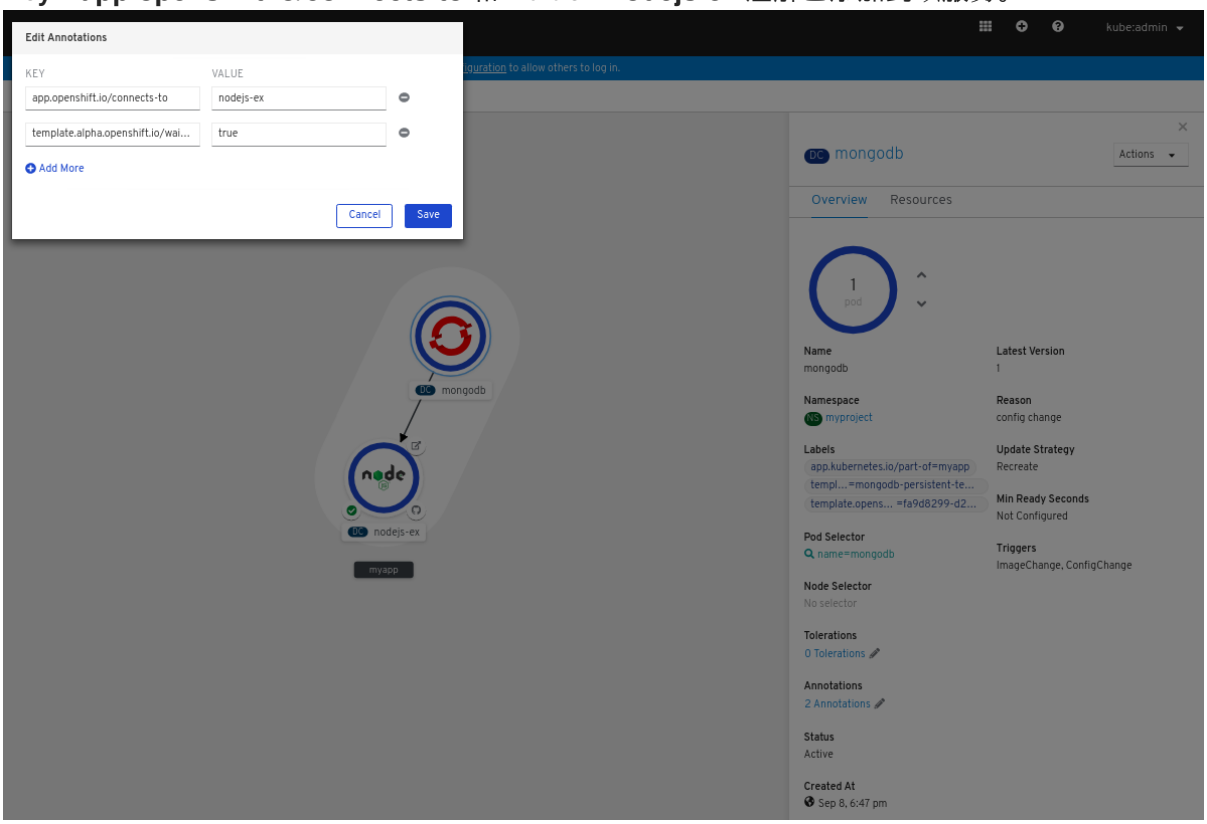
流程

1. 光标悬停到 MongoDB 服务上，节点上出现悬浮的箭头。

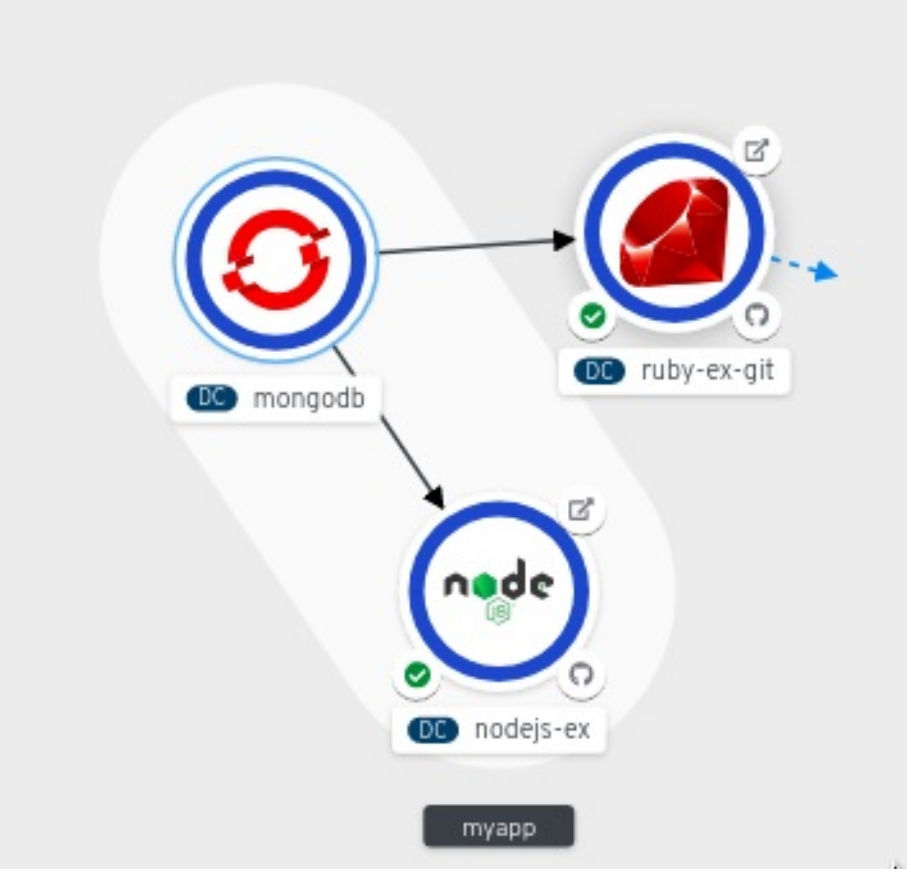


2. 点击箭头并将它拖向 Node.js 组件，使它与 MongoDB 服务连接。

3. 点击 MongoDB 服务以查看 **Overview** 面板。在 **Annotations** 部分，点击编辑图标，即可看到 **Key = app.openshift.io/connects-to** 和 **Value = nodejs-ex** 注解已添加到该服务。



同样，您可以创建其他应用程序和组件，并在它们之间建立连接。



2.4.6. 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。

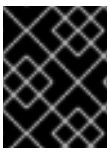
应用程序分组

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。

第 3 章 服务代理

3.1. 安装服务目录



重要

OpenShift Container Platform 4 中已弃用服务目录。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

3.1.1. 关于服务目录

在开发基于微服务的应用程序以在云原生平台中运行时，可以通过许多方式置备不同的资源并共享其协调、凭证和配置，具体取决于服务供应商和平台。

为了给开发人员提供更加顺畅的体验，OpenShift Container Platform 包含*服务目录（service catalog）*，这是 Kubernetes 的 Open Service Broker API (OSB API) 实施。用户可以将部署在 OpenShift Container Platform 中的应用程序与广泛的服务代理连接。

服务目录允许集群管理员使用单一 API 规格集成多个平台。OpenShift Container Platform Web 控制台显示服务目录中由服务代理提供的集群服务类，让用户能够发现并实例化服务以用于其应用程序。

因此，服务用户可从使用不同供应商的不同类型的服务简易性和一致性中受益，而服务供应商则可得益于通过一个集成点来访问多个平台。

默认情况下，OpenShift Container Platform 4 中不安装服务目录。

3.1.2. 安装服务目录

如果计划使用 OpenShift Ansible Broker 或 Template Service Broker 中的任何服务，您必须完成以下步骤来安装服务目录。

OpenShift Container Platform 中默认为服务目录的 API 服务器和控制器管理器创建自定义资源，但最初的 **managementState** 为 **Removed**。要安装服务目录，必须将这些资源的 **managementState** 更改为 **Managed**。

流程

1. 启用服务目录 API 服务器。
   a. 使用以下命令来编辑服务目录 API 服务器资源。

   ```
   $ oc edit servicecatalogapiservers
   ```

   b. 在 **spec** 下，将 **managementState** 字段设置为 **Managed**：

   ```
   spec:
     logLevel: Normal
     managementState: Managed
   ```

   c. 保存文件以应用更改。
   Operator 会安装服务目录 API 服务器组件。自 OpenShift Container Platform 4 起，此组件安装到 **openshift-service-catalog-apiserver** 命名空间中。

2. 启用服务目录控制器管理器。

   a. 使用以下命令来编辑服务目录控制器管理器资源。

   ```
   $ oc edit servicecatalogcontrollermanagers
   ```

   b. 在 **spec**下，将 **managementState** 字段设置为 **Managed**：

   ```
   spec:
     logLevel: Normal
     managementState: Managed
   ```

   c. 保存文件以应用更改。
   Operator 会安装服务目录控制器管理器组件。自 OpenShift Container Platform 4 起，此组件安装到 **openshift-service-catalog-controller-manager** 命名空间中。

3.2. 安装 TEMPLATE SERVICE BROKER

您可以安装 Template Service Broker 来访问它提供的模板应用程序。

重要

OpenShift Container Platform 4 中已弃用 Template Service Broker。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

先决条件

- 安装服务目录

3.2.1. 关于 Template Service Broker

Template Service Broker 将服务目录可见性引入到 OpenShift Container Platform 初始发行版本起附带的默认 Instant App 和 Quickstart 模板Template Service Broker 还能以服务形式提供针对其编写 OpenShift Container Platform 模板的任何对象，无论是来自红帽、集群管理员或用户，还是第三方供应商。

默认情况下，Template Service Broker 显示 **openshift** 项目中全局可用的对象。也可以将其配置为观察集群管理员选择的任何其他项目。

默认情况下，OpenShift Container Platform 4 中不安装 Template Service Broker。

3.2.2. 安装 Template Service Broker Operator

先决条件

- 已安装服务目录。

流程

以下流程使用 Web 控制台安装 Template Service Broker Operator。

1. 创建命名空间。

   a. 在 Web 控制台中导航至 **Administration → Namespaces**，再点击 **Create Namespace**。

b. 在 Name 字段中，输入 **openshift-template-service-broker** 并点击 Create。

注意

命名空间必须以 **openshift-** 开头。

2. 导航到 **Operators → OperatorHub** 页面。验证是否已选中 **openshift-template-service-broker** 项目。

3. 选择 **Template Service Broker Operator**。

4. 阅读 Operator 的信息并点击 **Install**。

5. 检查默认选择并点击 **Subscribe**。

接下来，需要启动 Template Service Broker，以便访问它提供的模板应用程序。

3.2.3. 启动 Template Service Broker

在安装 Template Service Broker Operator 后，可以按照以下流程启动 Template Service Broker。

先决条件

- 已安装服务目录。
- 已安装 Template Service Broker Operator。

流程

1. 在 Web 控制台中导航到 **Operators → Installed Operators**，再选择 **openshift-template-service-broker** 项目。

2. 选择 **Template Service Broker Operator**。

3. 在 **Provided APIs** 下，点击 **Template Service Broker** 的 **Create New**。

4. 检查默认 YAML 并点击 **Create**。

5. 验证是否已启动 Template Service Broker。
   在 Template Service Broker 启动后，可以通过导航到 **Catalog → Developer Catalog** 并选中 **Service Class** 复选框来查看可用的模板应用程序。请注意，可能需要等几分钟后模板服务代理才会启动，并且模板应用程序可供使用。

   如果还没有看到这些服务类，您可以检查以下项目的状态：

   - Template Service Broker Pod 状态
     - 在 **openshift-template-service-broker** 项目的 **Workloads → Pods** 页面中，验证名称开头为 **apiserver-** 的 Pod 的状态是否为 **Running** 并且就绪状态是否为 **Ready**。
   - 集群服务代理状态
     - 在 **Catalog → Broker Management → Service Brokers** 页面中，验证 **template-service-broker** 服务代理的状态是否为 **Ready**。
   - 服务目录控制器管理器 Pod 日志

- 在 **openshift-service-catalog-controller-manager** 项目的 **Workloads → Pods** 页面中，检查各个 Pod 的日志，并验证是否看到一个含有 **Successfully fetched catalog entries from broker** 消息的日志条目。

3.3. 置备模板应用程序

3.3.1. 置备模板应用程序

以下流程置备一个由 Template Service Broker 提供的 PostgreSQL 模板应用程序示例。

先决条件

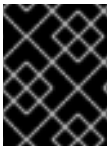
- 已安装服务目录。
- 已安装 Template Service Broker。

流程

1. 创建一个项目。
   a. 在 Web 控制台中，导航到 **Home → Project** 并点击 **Create Project**。
   b. 在 **Name** 字段中输入 **test-postgresql**，再点击 **Create**。
2. 创建服务实例。
   a. 导航到 **Catalog → Developer Catalog** 页面。
   b. 选择 **PostgreSQL (Ephemeral)** 模板应用程序，再点击 **Create Service Instance**。
   c. 检查默认选择项并设置任何其他必要的字段，然后点击 **Create**。
   d. 前往 **Catalog → Provisioned Services**，再验证 **postgresql-ephemeral** 服务实例是否已安装并具有 **Ready** 状态。
   您可以在 **Home → Events** 页面中查看进度。片刻之后，您应该会看到 **postgresql-ephemeral** 的一个事件，其含有"The instance was provisioned successfully"消息。
3. 创建服务绑定。
   a. 在 **Provisioned Services** 页面中，依次点击 **postgresql-ephemeral** 和 **Create Service Binding**。
   b. 检查默认服务绑定名称，再点击 **Create**。
   这会使用提供的名称为绑定创建一个新 secret。
4. 检查创建的 secret。
   a. 导航到 **Workloads → Secrets**，再验证是否创建了一个名为 **postgresql-ephemeral** 的 secret。
   b. 点击 **postgresql-ephemeral**，再查看 **Data** 部分中用于绑定到其他应用程序的键值对。

3.4. 卸载 TEMPLATE SERVICE BROKER

如果您不再需要访问 Template Service Broker 提供的模板应用程序，可以卸载该代理程序。



重要

OpenShift Container Platform 4 中已弃用 Template Service Broker。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

3.4.1. 卸载 Template Service Broker

以下流程使用 Web 控制台卸载 Template Service Broker 及其 Operator。

警告

如果集群中存在从 Template Service Broker 置备的服务，请不要卸载此代理程序，否则您在尝试管理该服务时可能会遇到错误。

先决条件

- 已安装 Template Service Broker。

流程

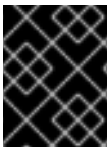
此流程假定您在 **openshift-template-service-broker** 项目中安装了 Template Service Broker。

1. 卸载 Template Service Broker。
   a. 导航到 **Operators → Installed Operators**，再从下拉菜单中选择 **openshift-template-service-broker** 项目。
   b. 点击 **Template Service Broker Operator**。
   c. 选择 **Template Service Broker** 选项卡。
   d. 点击 **template-service-broker**。
   e. 在 **Actions** 下拉菜单中，选择 **Delete Template Service Broker**。
   f. 点击确认弹出窗口中的 **Delete** 按钮。
   Template Service Broker 现已卸载，稍后模板应用程序也会从 Developer Catalog 中移除。
2. 卸载 Template Service Broker Operator。
   a. 在 **Operators → Installed Operators** 页面中，滚动页面或在 **Filter by name** 中输入关键字以查找 Template Service Broker Operator，然后点击它。
   b. 在 **Operator Details** 页面的右侧，从 **Actions** 下拉菜单中选择 **Uninstall Operator**。
   c. 如果要删除所有安装相关组件，则在看到 **Remove Operator Subscription** 窗口提示时，勾选 **Also completely remove the Operator from the selected namespace** 复选框。该操作将删除 CSV，继而删除与 Operator 相关的 Pod、Deployment、CRD 和 CR。
   d. 选择 **Remove**。此 Operator 将停止运行，并且不再接收更新。集群中不再安装 Template Service Broker Operator。

卸载 Template Service Broker 后，用户将不再能够访问 Template Service Broker 提供的模板应用程序。

3.5. 安装 OPENSHIFT ANSIBLE BROKER

您可以安装 OpenShift Ansible Broker，以访问它所提供的服务捆绑包。



重要

OpenShift Container Platform 4 中已弃用 OpenShift Ansible Broker。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

先决条件

- 安装服务目录

3.5.1. 关于 OpenShift Ansible Broker

OpenShift Ansible Broker 是 Open Service Broker (OSB) API 的一种实施，可以管理 *Ansible playbook 捆绑包* (APB) 定义的应用程序。APB 提供了一个在 OpenShift Container Platform 中定义和发布容器应用程序的方法，它包括构建到带有 Ansible 运行时的容器镜像中的一系列 Ansible playbook。APB 利用 Ansible 创建可自动完成复杂部署的标准机制。

OpenShift Ansible Broker 遵循以下基本工作流：

1. 用户使用 OpenShift Container Platform Web 控制台从服务目录请求可用应用程序列表。
2. 服务目录从 OpenShift Ansible Broker 请求可用应用程序列表。
3. OpenShift Ansible Broker 与定义的容器镜像 registry 通信，以了解有哪些 APB 可用。
4. 用户发出置备特定 APB 的请求。
5. OpenShift Ansible Broker 通过调用 APB 上的置备方法来履行用户的置备请求。

默认情况下，OpenShift Container Platform 4 中不安装 OpenShift Ansible Service Broker。

3.5.1.1. Ansible playbook 捆绑包

Ansible playbook 捆绑包 (APB) 是一种轻量级应用程序定义，可让您利用 Ansible 角色和 playbook 中的现有投入。

APB 使用含有指定 playbook 的简单目录来执行 OSB API 操作，比如置备和绑定。**apb.yml** 文件中定义的元数据包含部署过程中要使用的一系列必要和可选参数。

其他资源

- Ansible playbook 捆绑包存储库

3.5.2. 安装 OpenShift Ansible Service Broker Operator

先决条件

- 已安装服务目录。

流程

以下流程使用 Web 控制台安装 OpenShift Ansible Service Broker Operator。

1. 创建命名空间。
   a. 在 Web 控制台中导航至 **Administration → Namespaces**，再点击 **Create Namespace**。
   b. 在 **Name** 字段中输入 **openshift-ansible-service-broker**，在 **Labels** 字段中输入 **openshift.io/cluster-monitoring=true**，然后点击 **Create**。

注意

命名空间必须以 **openshift-** 开头。

2. 创建集群角色绑定。
   a. 导航到 **Administration → Role Bindings**，再点击 **Create Binding**。
   b. 对于 **Binding Type**，选择 **Cluster-wide Role Binding (ClusterRoleBinding)**。
   c. 对于 **Role Binding**，在 **Name** 字段中输入 **ansible-service-broker**。
   d. 对于 **Role**，选择 **admin**。
   e. 对于 **Subject**，选择 **Service Account** 选项，再选择 **openshift-ansible-service-broker** 命名空间，然后在 **Subject Name** 字段中输入 **openshift-ansible-service-broker-operator**。
   f. 点击 **Create**。
3. 创建用于连接 Red Hat Container Catalog 的 secret。
   a. 导航到 **Workloads → Secrets**。验证是否已选中 **openshift-ansible-service-broker** 项目。
   b. 点击 **Create → Key/Value Secret**。
   c. 输入 **asb-registry-auth** 作为 **Secret Name**。
   d. 添加名为 **username** 的 **Key** 和值为 Red Hat Container Catalog 用户名的 **Value**。
   e. 点击 **Add Key/Value**，再添加名为 **password** 的 **Key** 和值为 Red Hat Container Catalog 密码的 **Value**。
   f. 点击 **Create**。
4. 导航到 **Operators → OperatorHub** 页面。验证是否已选中 **openshift-ansible-service-broker** 项目。
5. 选择 **OpenShift Ansible Service Broker Operator**。
6. 阅读 Operator 的信息并点击 **Install**。
7. 检查默认选择并点击 **Subscribe**。

接下来，必须启动 OpenShift Ansible Broker，以便能访问它所提供的服务捆绑包。

3.5.3. 启动 OpenShift Ansible Broker

安装 OpenShift Ansible Service Broker Operator 后，可以按照以下流程启动 OpenShift Ansible Broker。

先决条件

- 已安装服务目录。
- 已安装 OpenShift Ansible Service Broker Operator。

流程

1. 在 Web 控制台中导航到 **Operators → Installed Operators**，再选择 **openshift-ansible-service-broker** 项目。
2. 选择 **OpenShift Ansible Service Broker Operator**。
3. 在 **Provided APIs** 下，点击 **Automation Broker** 的 **Create New**。
4. 在提供的默认 YAML 里的 **spec** 字段中添加以下内容：

```
registry:
  - name: rhcc
    type: rhcc
    url: https://registry.redhat.io
    auth_type: secret
    auth_name: asb-registry-auth
```

这将引用安装 OpenShift Ansible Service Broker Operator 时创建的 secret，该 secret 允许您连接到 Red Hat Container Catalog。

5. 设置其他 OpenShift Ansible Broker 配置选项，再点击 **Create**。
6. 验证 OpenShift Ansible Broker 是否已启动。
在 OpenShift Ansible Broker 启动后，您可以通过导航到 **Catalog → Developer Catalog** 并选中 **Service Class** 复选框来查看可用的服务捆绑包。请注意，可能需要等几分钟后 OpenShift Ansible Broker 才会启动，并且服务捆绑包可供使用。

   如果还没有看到这些服务类，您可以检查以下项目的状态：

   - OpenShift Ansible Broker Pod 状态
     - 在 **openshift-ansible-service-broker** 项目的 **Workloads → Pods** 页面中，验证名称开头为 **asb-** 的 Pod 的状态是否为 **Running** 并且就绪状态是否为 **Ready**。
   - 集群服务代理状态
     - 在 **Catalog → Broker Management → Service Brokers** 页面中，验证 **ansible-service-broker** 服务代理的状态是否为 **Ready**。
   - 服务目录控制器管理器 Pod 日志
     - 在 **openshift-service-catalog-controller-manager** 项目的 **Workloads → Pods** 页面中，检查各个 Pod 的日志，并验证是否看到一个含有 **Successfully fetched catalog entries from broker** 消息的日志条目。

3.5.3.1. OpenShift Ansible Broker 配置选项

您可以为 OpenShift Ansible Broker 设置以下选项。

表 3.1. OpenShift Ansible Broker 配置选项

YAML 键	描述	默认值
brokerName	用于标识代理实例的名称。	**ansible-service-broker**
brokerNamespace	代理所处的命名空间。	**openshift-ansible-service-broker**
brokerImage	用于代理的完全限定镜像。	**docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0**
brokerImagePullPolicy	用于代理镜像本身的拉取策略。	**IfNotPresent**
brokerNodeSelector	用于代理部署的节点选择器字符串。	**''**
registries	代理 registry 的 yaml 列表，指定允许用户配置的、代理用于从中发现并提供 APB 镜像的 registry。	请参阅默认 registries 数组。
logLevel	代理日志的日志级别。	**info**
apbPullPolicy	APB Pod 的拉取策略。	**IfNotPresent**
sandboxRole	赋予用于执行 APB 的服务帐户的角色。	**edit**
keepNamespace	是否在 APB 完成后（不论结果如何）删除为运行 APB 而创建的临时命名空间。	**false**
keepNamespaceOnError	是否在 APB 完成后（仅当结果为错误时）删除为运行 APB 而创建的临时命名空间。	**false**
bootstrapOnStartup	代理是否应该在启动时运行它的 bootstrap 例程。	**true**
refreshInterval	代理 bootstrap 刷新其 APB 清单的时间间隔。	**600s**
launchApbOnBind	*试验性：*切换对绑定操作执行 APB 的代理。	**false**
autoEscalate	代理是否应该在运行 APB 时提升用户的权限。这通常保留为 **false**，因为代理执行原始用户授权来确保该用户具有授予 APB 沙盒的权限。	**false**
outputRequest	是否输出代理接收的低级 HTTP 请求。	**false**

registries 的默认数组

```
- type: rhcc
  name: rhcc
  url: https://registry.redhat.io
  white_list:
  - ".*-apb$"
  auth_type: secret
  auth_name: asb-registry-auth
```

3.6. 配置 OPENSHIFT ANSIBLE BROKER

重要

OpenShift Container Platform 4 中已弃用 OpenShift Ansible Broker。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

3.6.1. 配置 OpenShift Ansible Broker

以下流程自定义 OpenShift Ansible Broker 的设置。

先决条件

- 已安装并启动 OpenShift Ansible Broker。

流程

此流程假定您已将 **ansible-service-broker** 用作 OpenShift Ansible Broker 的名称以及它所安装到的项目的名称。

1. 在 Web 控制台中导航到 **Operators → Installed Operators**，再选择 **ansible-service-broker** 项目。
2. 选择 **OpenShift Ansible Service Broker Operator**。
3. 在 **Automation Broker** 选项卡中，选择 **ansible-service-broker**。
4. 在 **YAML** 选项卡中，在 **spec** 字段中添加或更新所有 OpenShift Ansible Broker 配置选项。例如：

```
spec:
  keepNamespace: true
  sandboxRole: edit
```

5. 点击 **Save** 以应用这些更改。

3.6.1.1. OpenShift Ansible Broker 配置选项

您可以为 OpenShift Ansible Broker 设置以下选项。

表 3.2. OpenShift Ansible Broker 配置选项

YAML 键	描述	默认值
brokerName	用于标识代理实例的名称。	**ansible-service-broker**
brokerNamespace	代理所处的命名空间。	**openshift-ansible-service-broker**
brokerImage	用于代理的完全限定镜像。	**docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0**
brokerImagePullPolicy	用于代理镜像本身的拉取策略。	**IfNotPresent**
brokerNodeSelector	用于代理部署的节点选择器字符串。	**''**
registries	代理 registry 的 yaml 列表，指定允许用户配置的、代理用于从中发现并提供 APB 镜像的 registry。	请参阅 default registries array。
logLevel	代理日志的日志级别。	**info**
apbPullPolicy	APB Pod 的拉取策略。	**IfNotPresent**
sandboxRole	赋予用于执行 APB 的服务帐户的角色。	**edit**
keepNamespace	是否在 APB 完成后（不论结果如何）删除为运行 APB 而创建的临时命名空间。	**false**
keepNamespaceOnError	是否在 APB 完成后（仅当结果为错误时）删除为运行 APB 而创建的临时命名空间。	**false**
bootstrapOnStartup	代理是否应该在启动时运行它的 bootstrap 例程。	**true**
refreshInterval	代理 bootstrap 刷新其 APB 清单的时间间隔。	**600s**
launchApbOnBind	*试验性：*切换对绑定操作执行 APB 的代理。	**false**
autoEscalate	代理是否应该在运行 APB 时提升用户的权限。这通常保留为 **false**，因为代理执行原始用户授权来确保该用户具有授予 APB 沙盒的权限。	**false**
outputRequest	是否输出代理接收的低级 HTTP 请求。	**false**

registries 的默认数组

```
- type: rhcc
```

```
name: rhcc
url: https://registry.redhat.io
white_list:
- ".*-apb$"
auth_type: secret
auth_name: asb-registry-auth
```

3.6.2. 为 OpenShift Ansible Broker 配置监控

若要让 Prometheus 监控 OpenShift Ansible Broker，您必须创建以下资源来为 Prometheus 授予权限，以便其访问安装有 OpenShift Ansible Broker 的命名空间。

先决条件

- 已安装 OpenShift Ansible Broker。

注意

此流程假定您已将 OpenShift Ansible Broker 安装到 **openshift-ansible-service-broker** 命名空间中。

流程

1. 创建角色。
   a. 导航到 **Administration → Roles**，再点击 **Create Role**。
   b. 在编辑器中，用以下内容替换 YAML：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: prometheus-k8s
  namespace: openshift-ansible-service-broker
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
```

   c. 点击 **Create**。
2. 创建角色绑定。
   a. 导航到 **Administration → Role Bindings**，再点击 **Create Binding**。
   b. 对于 **Binding Type**，选择 **Namespace Role Binding (RoleBinding)**。

c. 对于 **Role Binding**，在 **Name** 字段中输入 **prometheus-k8s**，并在 **Namespace** 字段中输入 **openshift-ansible-service-broker**。

d. 对于 **Role**，选择 **prometheus-k8s**。

e. 对于 **Subject**，选择 **Service Account** 选项，选择 **openshift-monitoring** 命名空间，然后在 **Subject Name** 字段中输入 **prometheus-k8s**。

f. 点击 **Create**。

Prometheus 现在有权访问 OpenShift Ansible Broker 指标。

3.7. 置备服务捆绑包

3.7.1. 置备服务捆绑包

以下流程置备一个由 OpenShift Ansible Broker 提供的 PostgreSQL 服务捆绑包 (APB) 示例。

先决条件

- 已安装服务目录。
- 已安装 OpenShift Ansible Broker。

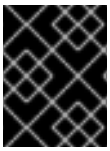
流程

1. 创建一个项目。
   a. 在 Web 控制台中，导航到 **Home → Project** 并点击 **Create Project**。
   b. 在 **Name** 字段中输入 **test-postgresql-apb**，再点击 **Create**。
2. 创建服务实例。
   a. 导航到 **Catalog → Developer Catalog** 页面。
   b. 选择 **PostgreSQL (APB)** 服务捆绑包，再点击 **Create Service Instance**。
   c. 检查默认选择项并设置任何其他必要的字段，然后点击 **Create**。
   d. 前往 **Catalog → Provisioned Services**，再验证 **dh-postgresql-apb** 服务实例是否已安装并具有 **Ready** 状态。
   您可以在 **Home → Events** 页面中查看进度。片刻之后，您应该会看到 **dh-postgresql-apb** 的一个事件，其含有“The instance was provisioned successfully”消息。
3. 创建服务绑定。
   a. 在 **Provisioned Services** 页面中，依次点击 **dh-postgresql-apb** 和 **Create Service Binding**。
   b. 检查默认服务绑定名称，再点击 **Create**。
   这会使用提供的名称为绑定创建一个新 secret。
4. 检查创建的 secret。
   a. 导航到 **Workloads → Secrets**，再验证是否创建了一个名为 **dh-postgresql-apb** 的 secret。

b. 点击 **dh-postgresql-apb**，再查看 **Data** 部分中用于绑定到其他应用程序的键值对。

3.8. 卸载 OPENSHIFT ANSIBLE BROKER

如果不再需要访问 OpenShift Ansible Broker 所提供的服务捆绑包，可以卸载该代理程序。



重要

OpenShift Container Platform 4 中已弃用 OpenShift Ansible Broker。Operator Framework 和 Operator Lifecycle Manager (OLM) 提供了等效和更好的功能。

3.8.1. 卸载 OpenShift Ansible Broker

以下流程使用 Web 控制台卸载 OpenShift Ansible Broker 及其 Operator。

警告

如果集群中存在从 OpenShift Ansible Broker 置备的服务，请不要卸载此代理程序，否则您在尝试管理该服务时可能会遇到错误。

先决条件

- 已安装 OpenShift Ansible Broker。

流程

此流程假定您已将 OpenShift Ansible Broker 安装到 **openshift-ansible-service-broker** 项目中。

1. 卸载 OpenShift Ansible Broker。
   a. 导航到 **Operators → Installed Operators**，再从下拉菜单中选择 **openshift-ansible-service-broker** 项目。
   b. 点击 **OpenShift Ansible Service Broker Operator**。
   c. 选择 **Automation Broker** 选项卡。
   d. 点击 **ansible-service-broker**。
   e. 在 **Actions** 下拉菜单中，选择 **Delete Automation Broker**。
   f. 点击确认弹出窗口中的 **Delete** 按钮。
   OpenShift Ansible Broker 现已卸载，稍后服务捆绑包也会从 Developer Catalog 中移除。
2. 卸载 OpenShift Ansible Service Broker Operator。
   a. 在 **Operators → Installed Operators** 页面中，滚动页面或在 **Filter by name** 中输入关键字以查找 OpenShift Ansible Service Broker Operator，然后点击它。
   b. 在 **Operator Details** 页面的右侧，从 **Actions** 下拉菜单中选择 **Uninstall Operator**。

c. 如果要删除所有安装相关组件，则在看到 **Remove Operator Subscription** 窗口提示时，勾选 **Also completely remove the Operator from the selected namespace** 复选框。该操作将删除 CSV，继而删除与 Operator 相关的 Pod、Deployment、CRD 和 CR。

d. 选择 **Remove**。此 Operator 将停止运行，并且不再接收更新。

集群中不再安装 OpenShift Ansible Service Broker Operator。

卸载 OpenShift Ansible Broker 后，用户将无法再访问 OpenShift Ansible Broker 提供的服务捆绑包。

第 4 章 部署

4.1. 了解 DEPLOYMENT 和 DEPLOYMENTCONFIG

OpenShift Container Platform 中的 *Deployment* 和 *DeploymentConfig* 是 API 对象，提供两种既相似又不同的方法来细致地管理常见的用户应用程序。由以下独立 API 对象组成：

- DeploymentConfig 或 Deployment，各自将应用程序特定组件的所需状态描述为 Pod 模板。
- DeploymentConfig 涉及一个或多个 *ReplicationController*，其包含 DeploymentConfig 状态的时间点记录，作为 Pod 模板。同样，Deployment 涉及一个或多个 *ReplicaSet*，即 ReplicationController 的后续。
- 一个或多个 Pod，代表应用程序某一特定版本的实例。

4.1.1. 部署构建块

Deployment 和 DeploymentConfig 各自通过使用原生 Kubernetes API 对象 ReplicationController 和 ReplicaSet 来启用，作为构建块。

用户不必操控 ReplicationController、ReplicaSet，或者 DeploymentConfig 或 Deployment 拥有的 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。

4.1.1.1. ReplicationController

ReplicationController 确保任何时候都运行指定数量的 Pod 副本。如果 Pod 退出或被删除，ReplicationController 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。

ReplicationController 配置包括：

- 所需的副本数（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 ReplicationController 管理的 Pod 的一组标签。这些标签包含在 ReplicationController 实例化的 Pod 定义中。ReplicationController 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。

ReplicationController 不负责根据负载或流量执行自动扩展，也不进行跟踪。相反，这需要由外部自动缩放器调整其副本数。

以下是 ReplicationController 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 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。

4.1.1.2. ReplicaSet

与 ReplicationController 类似，ReplicaSet 也是一个原生 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。ReplicaSet 和 ReplicationController 之间的区别在于，ReplicaSet 支持基于集合的选择器要求，而 ReplicationController 只支持基于相等的选择器要求。

注意

只有您需要自定义更新编配或根本不需要更新时才可使用 ReplicaSet。否则，请使用 Deployment。ReplicaSet 可以独立使用，但由部署用于编配 Pod 创建、删除和更新。部署自动管理其 ReplicaSet，为 Pod 提供声明性更新，而且不需要手动管理它们创建的 ReplicaSet。

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。

4.1.2. DeploymentConfig

在 ReplicationController 的基础上，OpenShift Container Platform 在 *DeploymentConfig* 概念中添加了对软件开发和部署生命周期的支持。在最简单的情形中，DeploymentConfig 会创建一个新的 ReplicationController，并允许它启动 Pod。

但是，从 DeploymentConfig 部署 OpenShift Container Platform 也支持从镜像的现有部署过渡到新部署，还支持定义可在创建 ReplicationController 之前或之后运行的 hook。

DeploymentConfig 部署系统提供以下功能：

- DeploymentConfig，它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。策略在 Pod 内运行，通常称为部署流程。
- 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 时，会创建一个 ReplicationController 来代表 DeploymentConfig 的 Pod 模板。如果 DeploymentConfig 发生变化，则会使用最新的 Pod 模板创建一个新的 ReplicationController，同时运行部署流程来缩减旧 ReplicationController 并扩展新的 ReplicationController。

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，不论是手动还是自动，部署器 Pod 会管理部署，包括缩减旧的 ReplicationController、扩展新的 ReplicationController，以及运行 hook。部署 Pod 在完成 Deployment 后无限期保留，以保存其 Deployment 日志。当部署被另一个部署替换时，上一 ReplicationController 会保留下来，以便在需要时轻松回滚。

DeploymentConfig 定义示例

```
apiVersion: 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. 每当 ReplicationController 模板更改时，**ConfigChange** 触发器会引发创建新的 Deployment。
2. 每当指定镜像流中有新版本的后备镜像可用时，**ImageChange** 触发器引发创建新的 Deployment。
3. 默认的 **Rolling** 策略会在 Deployment 之间实现无停机过渡。

4.1.3. 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 充当 OpenShift Container Platform 专用 DeploymentConfig 的后代。

与 DeploymentConfig 一样，Deployment 将应用程序特定组件的所需状态描述为 Pod 模板。Deployment 创建 ReplicaSet，后者负责编配 Pod 生命周期。

例如，以下 Deployment 定义创建用于调出一个 **hello-openshift** Pod 的 ReplicaSet：

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
```

4.1.4. 比较 Deployment 和 DeploymentConfig

Kubernetes Deployment 和 OpenShift Container Platform 提供的 DeploymentConfig 均在 OpenShift Container Platform 中受到支持；不过，建议您使用 Deployment，除非需要 DeploymentConfig 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。

4.1.4.1. 设计

Deployment 和 DeploymentConfig 之间的一大区别在于每种设计都为推出部署过程而选择的 CAP 原理。DeploymentConfig 以一致性为先，而 Deployment 更重视可用性。

对于 DeploymentConfig，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，Deployment 推出部署由控制器管理器推进。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 Deployment 做出反应，但这个问题会在故障发生后很快进行调节。

4.1.4.2. DeploymentConfig 相关功能

自动回滚
目前，在出现故障时，Deployment 不支持自动回滚到上次成功部署的 ReplicaSet。

触发器
Deployment 有一个隐式 **ConfigChange** 触发器 (trigger)，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
Deployment 尚不支持用户指定的自定义部署策略。

4.1.4.3. Deployment 相关功能

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 可以拥有尽可能多的活跃 ReplicaSet，最终部署控制器将缩减所有旧 ReplicaSet，并扩展最新的 ReplicaSet。

DeploymentConfig 最多可以运行一个部署器 Pod，否则多个部署器之间会产生冲突，试图扩展它们认为应该是最新的 ReplicationController。因此，任意时间点上只能有两个 ReplicationController 处于活跃状态。最终，这会转化为 Deployment 能够更快地进行推出部署。

按比例扩展
因为 Deployment 控制器是提供 Deployment 所拥有的新旧 ReplicaSet 的大小的唯一来源，所以它能够对正在推出（rollout）的部署进行扩展。额外的副本会根据每个 ReplicaSet 的大小按比例分发。

当一个推出部署（rollout）正在进行时无法扩展 DeploymentConfig，因为 DeploymentConfig 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。另一方面，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 DeploymentConfig，则部署器进程不受影响，它会继续运行直到完成为止。

4.2. 管理部署过程

4.2.1. 管理 DeploymentConfig

使用 OpenShift Container Platform Web 控制台的 **Workloads** 页面或 **oc** CLI 可以管理 DeploymentConfig。以下流程演示了 CLI 的用法（除非另有说明）。

4.2.1.1. 启动部署

您可以启动一个*推出部署*来启动应用程序的部署过程。

流程

1. 要从现有的 DeploymentConfig 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 ReplicationController。

4.2.1.2. 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 创建的 ReplicationController 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. 如需关于部署配置和最新修订的详细信息，可使用 **oc describe** 命令：

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

4.2.1.3. 重试部署

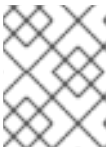
如果无法部署 DeploymentConfig 的当前修订，您可以重启部署过程。

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，命令会显示一条消息，而且不重试部署过程。



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 ReplicationController 会拥有与失败时相同的配置。

4.2.1.4. 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 的模板以匹配 undo 命令中指定的部署修订，并且启动新的 ReplicationController。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 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

DeploymentConfig 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。

4.2.1.5. 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 的 **spec** 字段中添加 **command** 参数。您也可以添加 **args** 字段来修改 **command**（如果 **command** 不存在，则修改 **ENTRYPOINT**）。

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

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

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

4.2.1.6. 查看部署日志

流程

1. 获得指定 DeploymentConfig 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 ReplicationController 及其部署器 Pod）并且没有手动清理或删除：

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

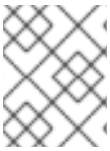
4.2.1.7. 部署触发器

DeploymentConfig 可以包含触发器，推动创建新部署过程来响应集群内的事件。

警告

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

ConfigChange 部署触发器
当 DeploymentConfig 的 Pod 模板中检测到配置更改时，**ConfigChange** 触发器会产生新的 ReplicationController。



注意

如果 DeploymentConfig 上定义了 **ConfigChange** 触发器，则在 DeploymentConfig 本身创建后会自动创建第一个 ReplicationController，而且不会暂停。

ConfigChange 部署触发器

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

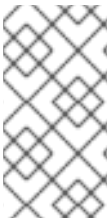
imageChange 部署触发器
ImageChange 触发器会在镜像流标签的内容发生改变时（推送镜像的新版本时）产生新的 ReplicationController。

imageChange 部署触发器

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

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

在上例中，当 **origin-ruby-sample** 镜像流的 **latest** 标签值更改并且新镜像值与 DeploymentConfig 的 **helloworld** 容器中指定的当前镜像不同时，系统会使用新镜像为 **helloworld** 创建新的 ReplicationController。



注意

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

4.2.1.7.1. 设置部署触发器

流程

1. 您可以使用 **oc set triggers** 命令为 DeploymentConfig 设置部署触发器。例如，若要设置 **ImageChangeTrigger**，请使用以下命令：

```
$ oc set triggers dc/<dc_name> \
    --from-image=<project>/<image>:<tag> -c <container_name>
```

4.2.1.8. 设置部署资源

注意

只有在集群管理员启用了临时存储技术预览功能时，这个资源才可用。此功能默认为禁用。

部署由在节点上消耗资源（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。

您还可以在部署策略中指定资源限值来限制资源使用。部署资源可以用在 Recreate、Rolling 或 Custom 部署策略中。

流程

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

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

(1) **cpu** 以 CPU 单元数为单位：**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)。只有集群管理员启用了临时存储技术预览功能时才适用。

不过，如果您的项目定义了配额，则需要以下两项之一：

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

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

(1) **requests** 对象包含与配额中资源列表对应的资源列表。

- 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。

4.2.1.9. 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，请使用 **oc scale** 命令。例如，以下命令将 **frontend** DeploymentConfig 中的副本数设置为 **3**。

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

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

4.2.1.10. 从 DeploymentConfig 访问私有存储库

您可以在 DeploymentConfig 中添加 secret，让它能够访问私有存储库中的镜像。此流程演示了 OpenShift Container Platform Web 控制台方法。

流程

1. 创建新项目。
2. 在 **Workloads** 页面中，创建一个含有用于访问私有镜像存储库的凭证的 secret。
3. 创建 DeploymentConfig。
4. 在 DeploymentConfig 编辑器页面中，设置 **Pull Secret** 并保存您的更改。

4.2.1.11. 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
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。

4.2.1.12. 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：

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

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

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

4.3. 使用 DEPLOYMENTCONFIG 策略

*部署策略*是更改或升级应用程序的一种方法。其目的是在无需停机的前提下进行修改，从而使用户几乎不会注意到这些变化。

因为最终用户通常通过由路由器控制的路由访问应用程序，所以部署策略侧重于 DeploymentConfig 功能或路由功能。侧重于 DeploymentConfig 的策略会影响到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。

许多部署策略通过 DeploymentConfig 支持，一些额外的策略则通过路由器功能支持。本节将讨论 DeploymentConfig 策略。

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。

部署策略使用就绪状态检查来确定新 Pod 是否准备就绪。如果未通过就绪状态检查，DeploymentConfig 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。

4.3.1. 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 中没有指定任何策略，则 Rolling 策略就是默认的部署策略。

在缩减旧组件前，滚动部署通常会借助 **readiness check** 等待新 Pod 变为 **ready**。如果发生严重问题，可以中止 Rolling 部署。

何时使用 Rolling 部署：

- 希望在应用程序更新过程中不需要停机时。
- 应用程序同时支持运行旧代码和新代码时。

Rolling 部署意味着您可以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。

Rolling 策略定义示例

```
strategy:
  type: Rolling
  rollingParams:
    updatePeriodSeconds: 1 (1)
    intervalSeconds: 1 (2)
    timeoutSeconds: 120 (3)
    maxSurge: "20%" (4)
    maxUnavailable: "10%" (5)
    pre: {} (6)
    post: {}
```

1. 在各个 Pod 更新之间等待的时间。如果未指定，则默认值为 **1**。
2. 更新后轮询部署状态之间等待的时间。如果未指定，则默认值为 **1**。
3. 放弃前等待扩展事件的时间。可选，默认值为 **600**。这里的*放弃*表示自动回滚到以前的完整部署。

4 **maxSurge** 是可选的；如果未指定，则默认为 **25%**。请参考以下流程中的信息。

5 **maxUnavailable** 是可选的；如果未指定，则默认为 **25%** 。请参考以下流程中的信息。

6 **pre** 和 **post** 都是生命周期 hook。

Rolling 策略：

1. 执行任何 **pre** 生命周期 hook。
2. 根据激增数扩展新的 ReplicationController。
3. 根据最大不可用数，缩减旧的 ReplicationController。
4. 重复此扩展，直到新的 ReplicationController 达到所需的副本数，并且旧的 ReplicationController 已缩减到零。
5. 执行任何 **post** 生命周期 hook。

重要

在缩减时，Rolling 策略会等待 Pod 准备就绪，以便它能决定进一步缩放是否会影响到可用性。如果扩展 Pod 永不就绪，部署过程将最终超时并导致部署失败。

maxUnavailable 参数是在更新过程中不可用的 Pod 的最大数量。**maxSurge** 参数是原始 Pod 数之上最多可以调度的 Pod 数。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**。

4.3.1.1. Canary 部署

OpenShift Container Platform 中的所有 Rolling 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，则移除该 Canary 实例，并且自动回滚 DeploymentConfig。

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。

4.3.1.2. 创建 Rolling 部署

在 OpenShift Container Platform 中，Rolling 部署是默认类型。您可以使用 CLI 创建 Rolling 部署。

流程

1. 根据 DockerHub 中找到的示例部署镜像创建一个应用程序：

```
$ oc new-app openshift/deployment-example
```

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 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 ReplicationController 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，该过程中止，并且 DeploymentConfig 回滚到之前的版本。

4.3.2. Recreate 策略

Recreate 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

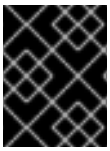
```
strategy:
  type: Recreate
  recreateParams: 1
    pre: {} 2
    mid: {}
    post: {}
```

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。

何时使用 Recreate 部署：

- 需要在新代码启动前进行迁移或进行其他数据转换时。
- 不支持同时运行应用程序代码的新旧版本时。
- 当使用 RWO 卷时，不支持在多个副本间共享该卷。

Recreate 部署会出现停机的情况，这是因为在短时间内会出现没有运行您的应用程序实例的情况。然而，旧代码和新代码不会被同时运行。

4.3.3. Custom 策略

Custom 策略允许您提供自己的部署行为。

Custom 策略定义示例

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
      - name: ENV_1
        value: VALUE_1
```

在上例中，**organization/strategy** 容器镜像提供部署行为。可选的 **command** 数组覆盖镜像的 **Dockerfile** 中指定的任何 **CMD** 指令。提供的可选环境变量添加到策略过程的执行环境中。

另外，OpenShift Container Platform 为部署过程提供以下环境变量：

环境变量	描述
OPENSHIFT_DEPLOYMENT_NAME	新部署 (ReplicationController) 的名称。
OPENSHIFT_DEPLOYMENT_NAMESPACE	新部署的命名空间。

新部署的副本数最初为零。该策略负责使新部署积极使用最能满足用户需求的逻辑。

另外，也可使用 **customParams** 将自定义部署逻辑注入现有的部署策略中。提供自定义 shell 脚本逻辑并调用 **openshift-deploy** 二进制文件。用户不必提供自定义的部署器容器镜像；本例中使用默认的 OpenShift Container Platform 部署器镜像：

```
strategy:
  type: Rolling
  customParams:
    command:
    - /bin/sh
    - -c
    - |
      set -e
      openshift-deploy --until=50%
      echo Halfway there
      openshift-deploy
      echo Complete
```

这会产生以下部署：

```
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。

4.3.4. 生命周期 hook

Rolling 和 Recreate 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 **failurePolicy**，定义在遇到 hook 失败时策略应执行的操作：

Abort	如果 hook 失败，部署过程将被视为失败。

Retry	应重试 hook 执行过程，直到成功为止。
Ignore	所有 hook 失败都应忽略，部署应继续进行。

Hook 具有特定类型的字段，用于描述如何执行 Hook。目前，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 模板的新 Pod 中执行 hook 代码。

以下简化 DeploymentConfig 示例使用了 Rolling 策略。为简明起见，省略了触发器和其他一些次要的细节：

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld 1
          command: [ "/usr/bin/command", "arg1", "arg2" ] 2
          env: 3
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data 4
```

1. **helloworld** 名称指代 **spec.template.spec.containers[0].name**。
2. 此 **command** 覆盖 **openshift/origin-ruby-sample** 镜像中定义的任何 **ENTRYPOINT**。
3. **env** 是 hook 容器的一组可选环境变量。
4. **volumes** 是 hook 容器的一组可选的卷引用。

在本例中，将使用 **helloworld** 容器中的 **openshift/origin-ruby-sample** 镜像在新 Pod 中执行 **pre** hook。hook Pod 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 Pod 继承 **data** 卷。

4.3.4.1. 设置生命周期 hook

您可以使用 CLI 为 DeploymentConfig 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 \
    -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

4.4. 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 DeploymentConfig 进行对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 DeploymentConfig 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（蓝色版本）上线进行测试和评估，同时用户仍然使用稳定版本（绿色版本）。准备就绪后，用户切换到蓝色版本。如果出现问题，您可以切回到绿色版本。

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。

4.4.1. 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。

4.4.2. 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。

4.4.3. 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** 属性可控制恰当终止的时间（默认为 30 秒），并可根据需要对单个应用程序进行独立设置。

4.4.4. 蓝绿部署

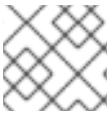
蓝绿部署涉及同时运行应用程序的两个版本，并将流量从生产版本（绿色版本）移动到更新版本（蓝色版本）。您可以使用 Rolling 策略或切换路由中的服务。

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。

4.4.4.1. 设置蓝绿部署

蓝绿部署使用两个 DeploymentConfig。这两者都在运行，生产环境中的 DeploymentConfig 依赖于路由指定的服务，每个 DeploymentConfig 都公开给不同的服务。



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（蓝色）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（绿色）。

流程

1. 创建示例应用程序的两个副本：

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

这会创建两个独立的应用程序组件：一个在 **example-green** 服务下运行 **v1** 镜像，另一个使用 **example-blue** 服务下的 **v2** 镜像。

2. 创建指向旧服务的路由：

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. 通过 **example-green.<project>.<router_domain>** 访问应用程序，验证您能否看到 **v1** 镜像。

4. 编辑路由并将服务名称改为 **example-blue**：

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

5. 要验证路由是否已改变，请刷新浏览器直到您看到 **v2** 镜像。

4.4.5. 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。

4.4.5.1. 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，拥有大量端点的服务会得到高于必要值的 **weight**。这时，可以减少 Pod 数量来获得所需的负载均衡 **weight**。

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。各自创建一个 DeploymentConfig。应用程序是同一程序的不同版本；一个是当前生产版本，另一个是提议的新版本：

```
$ oc new-app openshift/deployment-example --name=ab-example-a
$ oc new-app openshift/deployment-example --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
```

通过 **ab-example-<project>.<router_domain>** 访问应用程序，以验证您能否看到所需的版本。

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。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 DeploymentConfig 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>
...
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
...
```

4.4.5.1.1. 使用 Web 控制台管理权重

流程

1. 导航到 Route 详情页面 (Applications/Routes)。
2. 从 Actions 菜单中选择 **Edit**。
3. 选中 **Split traffic across multiple services**。
4. **Service Weights** 滑块设置发送到各个服务的流量的百分比。

如果在超过两个服务之间进行流量分割，各个服务的相对权重通过 0 到 256 范围内的整数来指定。

在分割流量的应用程序的展开行中 Overview 上会显示流量加权情况。

4.4.5.1.2. 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 DeploymentConfig。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
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。
例如：

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
$ oc set route-backends ab-example --adjust ab-example-b=5%
$ 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。

4.4.5.1.3. 一个服务，多个 DeploymentConfig

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
```

   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（或直接使用服务 IP）提供应用程序：

```
$ oc expose svc/ab-example-a --name=ab-example
```

3. 通过 **ab-example-<project>.<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"
```

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 DeploymentConfig 中更改 **SUBTITLE** 环境变量来触发部署：

```
$ oc edit dc/ab-example-a
```

   或

```
$ oc edit dc/ab-example-b
```

第 5 章 配额

5.1. 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累计资源消耗。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。

5.1.1. 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 5.1. 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。
ephemeral-storage	非终端状态的所有 Pod 的本地临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。
requests.cpu	非终端状态的所有 Pod 的 CPU 请求总和不能超过这个值。**cpu** 和 **requests.cpu** 的值相同，并可互换使用。
requests.memory	非终端状态的所有 Pod 的内存请求总和不能超过这个值。**memory** 和 **requests.memory** 的值相同，并可互换使用。
requests.ephemeral-storage	非终端状态的所有 Pod 的临时存储请求总和不能超过这个值。**ephemeral-storage** 和 **requests.ephemeral-storage** 的值相同，并可互换使用。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。
limits.cpu	非终端状态的所有 Pod 的 CPU 限值总和不能超过这个值。
limits.memory	非终端状态的所有 Pod 的内存限值总和不能超过这个值。
limits.ephemeral-storage	非终端状态的所有 Pod 的临时存储限值总和不能超过这个值。只有在您启用了临时存储技术预览时，此资源才可用。此功能默认为禁用。

表 5.2. 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。
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	在处于任何状态且具有匹配存储类的所有持久性卷声明中，存储请求总和不能超过这个值。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	项目中可以存在的具有匹配存储类的持久性卷声明的总数。

表 5.3. 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。

5.1.2. 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
Terminating	匹配 **spec.activeDeadlineSeconds >= 0** 的 Pod。
NotTerminating	匹配 **spec.activeDeadlineSeconds** 为 **nil** 的 Pod。
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**

Terminating、**NotTerminating** 和 **NotBestEffort** 范围将配额仅限为跟踪以下资源：

- **pods**
- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**
- **ephemeral-storage**
- **requests.ephemeral-storage**
- **limits.ephemeral-storage**

注意

只有在启用了临时存储技术预览功能时，才会应用临时存储请求和限值。此功能默认为禁用。

5.1.3. 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。

5.1.4. 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。

5.1.5. 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 ReplicationController 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)
    requests.ephemeral-storage: 2Gi (4)
    limits.cpu: "2" (5)
    limits.memory: 2Gi (6)
    limits.ephemeral-storage: 4Gi (7)
```

1. 项目中可以存在的处于非终端状态的 Pod 总数。
2. 在非终端状态的所有 Pod 中，CPU 请求总和不能超过 1 个内核。
3. 在非终端状态的所有 Pod 中，内存请求总和不能超过 1Gi。
4. 在非终端状态的所有 Pod 中，临时存储请求总和不能超过 2Gi。
5. 在非终端状态的所有 Pod 中，CPU 限值总和不能超过 2 个内核。
6. 在非终端状态的所有 Pod 中，内存限值总和不能超过 2Gi。
7. 在非终端状态的所有 Pod 中，临时存储限值总和不能超过 4Gi。

besteffort.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" (1)
  scopes:
  - BestEffort (2)
```

1. 项目中可以存在的具有 **BestEffort** 服务质量的非终端状态 Pod 的总数。
2. 将配额仅限为在内存或 CPU 方面具有 **BestEffort** 服务质量的匹配 Pod。

compute-resources-long-running.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
```

```
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
    limits.ephemeral-storage: "4Gi" 4
  scopes:
  - NotTerminating 5
```

1 处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 限值总和不能超过这个值。

3 在非终端状态的所有 Pod 中，内存限值总和不能超过这个值。

4 在非终端状态的所有 Pod 中，临时存储限值总和不能超过这个值。

5 将配额仅限为 **spec.activeDeadlineSeconds** 设为 **nil** 的匹配 Pod。构建 Pod 会归入 **NotTerminating** 下，除非应用了 **RestartNever** 策略。

compute-resources-time-bound.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
    limits.ephemeral-storage: "1Gi" 4
  scopes:
  - Terminating 5
```

1 处于非终端状态的 Pod 总数。

2 在非终端状态的所有 Pod 中，CPU 限值总和不能超过这个值。

3 在非终端状态的所有 Pod 中，内存限值总和不能超过这个值。

4 在非终端状态的所有 Pod 中，临时存储限值总和不能超过这个值。

5 将配额仅限为 **spec.activeDeadlineSeconds >=0** 的匹配 Pod。例如，此配额适用于构建或部署器 Pod，而非 Web 服务器或数据库等长时间运行的 Pod。

storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
```

```
spec:
  hard:
    persistentvolumeclaims: "10" (1)
    requests.storage: "50Gi" (2)
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi" (3)
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" (4)
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" (5)
    bronze.storageclass.storage.k8s.io/requests.storage: "0" (6)
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" (7)
```

(1) 项目中的持久性卷声明总数

(2) 在一个项目中的所有持久性卷声明中，请求的存储总和不能超过这个值。

(3) 在一个项目中的所有持久性卷声明中，金级存储类中请求的存储总和不能超过这个值。

(4) 在一个项目中的所有持久性卷声明中，银级存储类中请求的存储总和不能超过这个值。

(5) 在一个项目中的所有持久性卷声明中，银级存储类中声明总数不能超过这个值。

(6) 在一个项目中的所有持久性卷声明中，铜级存储类中请求的存储总和不能超过这个值。如果此值设为 **0**，则表示铜级存储类无法请求存储。

(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。

5.1.6. 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
```

5.1.6.1. 创建对象数配额

您可以针对所有 OpenShift Container Platform 标准命名空间资源类型创建对象数配额，如 **BuildConfig**和 **DeploymentConfig**。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，如果服务器存储中存在某一对象，则从其配额中扣减。这些类型的配额对防止耗尽存储资源很有用处。

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>** 是资源名称，**<group>** 则是 API 组（若适用）。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
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
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
```

5.1.6.2. 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'
                    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. 在命名空间 **nvidia** 中设置配额。本例中配额为 **1**：

```
# cat gpu-quota.yaml
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
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
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  0     1
```

5. 运行一个请求单个 GPU 的 Pod：

```
# oc create -f 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. 验证 Pod 是否在运行：

```
# oc get pods
NAME            READY    STATUS     RESTARTS   AGE
gpu-pod-s46h7   1/1      Running    0          1m
```

7. 验证配额计数器 **Used** 是否正确：

```
# oc describe quota gpu-quota -n nvidia
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  1     1
```

8. 尝试在 **nvidia** 命名空间中创建第二个 GPU Pod。从技术上讲，该节点支持 2 个 GPU，因为它有 2 个 GPU：

```
# oc create -f gpu-pod.yaml
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。

5.1.7. 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
NAME              AGE
besteffort        11m
compute-resources   2m
core-object-counts  29m
```

2. 描述您关注的配额，如 **core-object-counts** 配额：

```
$ oc describe quota core-object-counts -n demoproject
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
configmaps  3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

5.1.8. 要求显式配额来消耗资源

如果资源不受配额管理，用户可以消耗的资源量就不会有限制。例如，如果没有与金级存储类相关的存储配额，则项目可以创建的金级存储量没有限制。

对于高成本计算或存储资源，管理员可能要求赋予显式配额方可消耗资源。譬如，如果某个项目没有显式赋予与金级存储类有关的存储配额，则该项目的用户将无法创建该类型的存储。

流程

要求显式配额来消耗特定资源：

1. 将以下小节添加到主控机配置中：

```
admissionConfig:
  pluginConfig:
    ResourceQuota:
      configuration:
        apiVersion: resourcequota.admission.k8s.io/v1alpha1
        kind: Configuration
        limitedResources:
        - resource: persistentvolumeclaims (1)
        matchContains:
        - gold.storageclass.storage.k8s.io/requests.storage (2)
```

(1) 默认限制其消耗的组/资源。

(2) 通过配额跟踪并且与默认限制的组/资源关联的资源名称。

在上例中，配额系统会拦截创建或更新 **persistentVolumeClaim** 的每个操作。它会检查将要消耗的配额理解哪些资源，如果项目中没有配额涵盖这些资源，则请求会被拒绝。

在本例中，如果用户创建的 **PersistentVolumeClaim** 使用与金级存储类关联的存储，并且项目中没有匹配的配额，请求会被拒绝。

5.1.9. 配置配额同步周期

在删除一组资源后，但在恢复配额使用量前，用户在尝试重新使用这些资源时可能会遇到问题。资源同步时间表由 **resource-quota-sync-period** 设置决定，该设置可由集群管理员配置。

在使用自动化时，调整重新生成时间对创建资源和决定资源使用量很有帮助。

注意

resource-quota-sync-period 设置旨在平衡系统性能。缩短同步周期可能会导致 master 遭遇重度负载。

流程

配置配额同步周期：

1. 编辑 Kubernetes 控制器管理器。

```
$ oc edit kubecontrollermanager cluster
```

2. 为 **resource-quota-sync-period** 字段指定时间量（以秒为单位），将 **unsupportedconfigOverrides** 字段更改为具有下列设置：

```
unsupportedConfigOverrides:
  extendedArguments:
    resource-quota-sync-period:
    - 60s
```

5.2. 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。

5.2.1. 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: 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: v1
kind: ClusterResourceQuota
metadata:
  creationTimestamp: null
  name: for-name
spec:
  quota:
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: null
    labels:
      matchLabels:
        name: frontend
```

5.2.2. 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
```

例如：

```
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
```

5.2.3. 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 6 章 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。

6.1. 了解健康检查

探测（probe）是一种 Kubernetes 操作，它会定期对运行中的容器执行诊断。目前有两种类型的探测，分别满足不同的目的。

就绪度（Readiness）探测

就绪度检查确定在其中调度这个操作的容器是否已准备好满足请求。如果某一容器的就绪度探测失败，则端点控制器将确保从所有服务的端点中移除该容器的 IP 地址。就绪度探测也可用于向端点控制器发送信号，即使有容器在运行它也不应从代理接收任何流量。

例如，就绪度检查可以控制要使用哪些 Pod。如果 Pod 尚未就绪，它会被移除。

存活度（Liveness）探测

存活度检查确定在其中调度这个操作的容器是否仍然在运行。如果存活度探测因为死锁等某种状况而失败，kubelet 会终止容器。容器而后根据其重启策略做出响应。

例如，**restartPolicy** 为 **Always** 或 **OnFailure** 的节点上的存活度探测可以终止并重启该节点上的容器。

存活度检查示例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness-http
    image: k8s.gcr.io/liveness (1)
    args:
    - /server
    livenessProbe: (2)
      httpGet: (3)
        # host: my-host
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      initialDelaySeconds: 15 (4)
      timeoutSeconds: 1 (5)
    name: liveness (6)
```

(1) 指定用于存活度探测的镜像。

(2) 指定健康检查的类型。

3 指定存活度检查的类型：

- HTTP 检查。指定 **httpGet**。
- 容器执行检查。指定 **exec**。
- TCP 套接字检查。指定 **tcpSocket**。

4 指定容器启动后执行第一个探测前的秒数。

5 指定探测之间的秒数。

不健康容器存活度检查输出结果示例

```
$ oc describe pod pod1

....

FirstSeen LastSeen    Count   From            SubobjectPath          Type       Reason      Message
--------- --------    -----   ----            -------------          --------   ------      -------
37s       37s     1   {default-scheduler }                       Normal     Scheduled   Successfully assigned
liveness-exec to worker0
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Pulling     pulling image
"k8s.gcr.io/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Pulled      Successfully
pulled image "k8s.gcr.io/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Created     Created
container with docker id 86849c15382e; Security:[seccomp=unconfined]
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Started     Started
container with docker id 86849c15382e
2s        2s      1   {kubelet worker0}   spec.containers{liveness}   Warning    Unhealthy   Liveness
probe failed: cat: can't open '/tmp/healthy': No such file or directory
```

6.1.1. 了解健康检查的类型

存活度检查和就绪度检查可以通过三种方式进行配置：

HTTP 检查

kubelet 使用 Web hook 来确定容器的健康状态。如果 HTTP 响应代码介于 200 和 399 之间，则检查被认定为成功。

对于在完全初始化后返回 HTTP 状态代码的应用程序，HTTP 检查是理想的选择。

容器执行检查

kubelet 在容器内执行一个命令。退出检查时状态为 0 被视为成功。

TCP 套接字检查

kubelet 尝试向容器打开一个套接字。只有检查能够建立连接，容器才被视为健康。对于只有初始化完毕后才开始侦听的应用程序，TCP 套接字检查是理想的选择。

6.2. 配置健康检查

若要配置健康检查，请为您需要的每一种检查创建一个 pod。

流程

创建健康检查：

1. 创建存活度容器执行检查：

   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - args:
    image: k8s.gcr.io/liveness
    livenessProbe:
      exec: (1)
        command: (2)
        - cat
        - /tmp/health
      initialDelaySeconds: 15 (3)
...
```

(1) 指定存活度检查以及存活度检查的类型。

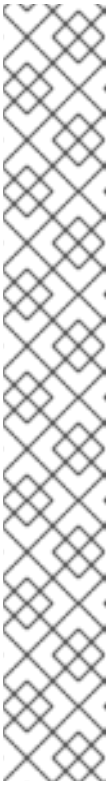
(2) 指定要在容器中使用的命令。

(3) 指定容器启动后执行第一个探测前的秒数。

   b. 验证健康检查 pod 的状态：

```
$ oc describe pod liveness-exec

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
"k8s.gcr.io/liveness"
  Normal  Pulled     1s    kubelet, ip-10-0-143-40.ec2.internal  Successfully pulled image
"k8s.gcr.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
```



注意

timeoutSeconds 参数不影响容器执行检查的就绪度和存活度探测。您可以在探测本身中使用超时机制，因为 OpenShift Container Platform 无法对进入容器的 exec 调用执行超时。在探测中实施超时的一种方法是使用 **timeout** 参数来运行存活度或就绪度探测：

```
spec:
  containers:
    livenessProbe:
      exec:
        command:
          - /bin/bash
          - '-c'
          - timeout 60 /opt/eap/bin/livenessProbe.sh 1
      timeoutSeconds: 1
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 3
```

1 超时值和探测脚本路径。

c. 创建检查：

```
$ oc create -f <file-name>.yaml
```

2. 创建存活度 TCP 套接字检查：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
  - name: contaier1 1
    image: k8s.gcr.io/liveness
    ports:
    - containerPort: 8080 2
    livenessProbe: 3
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15 4
      timeoutSeconds: 1 5
```

1 2 指定检查要连接到的容器名称和端口。

3 指定存活度健康检查以及存活度检查的类型。

4 指定容器启动后执行第一个探测前的秒数。

5 指定探测之间的秒数。

b. 创建检查：

```
$ oc create -f <file-name>.yaml
```

3. 创建就绪度 HTTP 检查：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness-http
spec:
  containers:
  - args:
    image: k8s.gcr.io/readiness 1
    readinessProbe: 2
    httpGet:
    # host: my-host 3
    # scheme: HTTPS 4
      path: /healthz
      port: 8080
    initialDelaySeconds: 15 5
    timeoutSeconds: 1 6
```

1 指定用于存活度探测的镜像。

2 指定就绪度健康检查以及就绪度检查的类型。

3 指定主机 IP 地址。如果未定义 **host**，则使用 **PodIP**。

4 指定 **HTTP** 或 **HTTPS**。如果未定义 **scheme**，则使用 **HTTP** 方案。

5 指定容器启动后执行第一个探测前的秒数。

6 指定探测之间的秒数。

b. 创建检查：

```
$ oc create -f <file-name>.yaml
```

第 7 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共云中部署集群时，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 DeploymentConfig。闲置应用程序的操作涉及闲置所有关联的资源。

7.1. 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。

7.1.1. 闲置一个服务

流程

1. 要闲置一个服务，请运行：

```
$ oc idle <service>
```

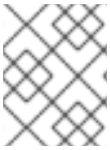
7.1.2. 闲置多个服务

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。

7.2. 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。

第 8 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。

8.1. 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>**。

8.2. 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 8.1. 修剪组 CLI 配置选项

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。

查看 prune 命令删除的组：

```
$ oc adm prune groups --sync-config=ldap-sync-config.yaml
```

执行修剪操作：

```
$ oc adm prune groups --sync-config=ldap-sync-config.yaml --confirm
```

8.3. 修剪部署

要修剪因为时间和状态而不再需要的部署，管理员可运行以下命令：

```
$ oc adm prune deployments [<options>]
```

表 8.2. 修剪部署 CLI 配置选项

选项	描述
--confirm	指明应该执行修剪，而不是空运行。
--orphans	修剪所有不再具有 DeploymentConfig、状态为 **Complete** 或 **Failed** 且副本数为零的部署。
--keep-complete=<N>	对于每个 DeploymentConfig，保留状态为 **Complete** 且副本数为零的最后 **N** 个部署。（默认值 **5**）
--keep-failed=<N>	对于每个 DeploymentConfig，保留状态为 **Failed** 且副本数为零的最后 **N** 个部署。（默认值 **1**）
--keep-younger-than=<duration>	不修剪相对于当前时间年龄不到 **<duration>** 的对象。（默认值 **60m**）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**)。

查看修剪操作要删除的对象：

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

真正执行修剪操作：

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```

8.4. 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 8.3. 修剪构建 CLI 配置选项

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 **5**）。

选项	描述
--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 **1**）。
--keep-younger-than=<duration>	不修剪相对于当前时间年龄不到 **<duration>** 的对象（默认值 **60m**）。

查看修剪操作要删除的对象：

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

真正执行修剪操作：

```
$ 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

8.5. 修剪镜像

要修剪系统因为年龄、状态或超过限值而不再需要的镜像，管理员可运行以下命令：

```
$ oc adm prune images [<options>]
```

目前，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**）。

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，修剪之后需要清除缓存。这可以通过重新部署 registry 来完成：

```
# oc patch deployment image-registry -n openshift-image-registry --type=merge --patch="{\"spec\":
{\"template\":{\"metadata\":{\"annotations\":{\"kubectl.kubernetes.io/restartedAt\": \"$(date '+%Y-%m-
%dT%H:%M:%SZ' -u)\"}}}}}"
```

如果集成 registry 使用 Redis 缓存，您必须手动清理数据库。

oc adm prune images 操作需要 registry 的路由。默认不创建 registry 路由。请参阅 OpenShift Container Platform 中的 Image Registry Operator 来了解有关如何创建 registry 路由的信息，并参阅公开 registry来了解有关如何公开 registry 服务的详细信息。

表 8.4. 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**）。
--keep-younger-than=<duration>	不修剪相对于当前时间年龄不到 **<duration>** 的镜像。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。

8.5.1. 镜像修剪条件

- 对于“由 OenShift Container Platform 管理”的镜像（带有注解 **openshift.io/image.managed** 的镜像），请移除创建时间为至少 **--keep-younger-than** 分钟前，且目前没有被以下对象引用的镜像：
    - 任何 **--keep-younger-than** 分钟前之后创建的 Pod。
    - 任何 **--keep-younger-than** 分钟前之后创建的镜像流。

    - 任何正在运行的 Pod。
    - 任何待处理的 Pod。
    - 任何 ReplicationController。
    - 任何部署。
    - 任何 DeploymentConfig。
    - 任何 ReplicaSets。
    - 任何构建配置。
    - 任何构建。
    - **stream.status.tags[].items** 中 **--keep-tag-revisions** 个最新项。
- 对于"由 OpenShift Container Platform 管理"的镜像（带有注解 **openshift.io/image.managed** 的镜像），请移除超过同一项目中定义的最小限值，并且目前没有被以下对象引用的镜像：
    - 任何正在运行的 Pod。
    - 任何待处理的 Pod。
    - 任何 ReplicationController。
    - 任何部署。
    - 任何 DeploymentConfig。
    - 任何 ReplicaSets。
    - 任何构建配置。
    - 任何构建。
- 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。

若要分开移除 OpenShift Container Platform 镜像 API 对象和 registry 中的数据，可以使用 **--prune-registry=false** 选项再硬修剪 registry；这可以缩减一些计时窗口，与尝试通过一个命令同时修剪这两者相比也更为安全。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。您仍应追踪在修剪操作期间创建的可能会引用镜像的 API 对象，从而避免引用已删除的内容。

另请注意，重做修剪时如果不带 **--prune-registry** 选项或带有 **--prune-registry=true** 选项，不会造成修剪之前通过 **--prune-registry=false** 修剪的镜像的镜像 registry 中关联的存储。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。

8.5.2. 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，并且保留年龄不足 60 分钟的资源（镜像、镜像流和 Pod）：

```
$ 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真正执行修剪操作：

```
$ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
```

```
$ oc adm prune images --prune-over-size-limit --confirm
```

8.5.3. 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。

8.5.4. 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 **sha:abz** 的陈旧镜像。在标记镜像的密码空间 **N** 中运行以下命令，会造成在一个名为 **myapp** 的镜像流对该镜像标记三次：

```
$ image_name="sha:abz"
$ oc get is -n N -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag := $is.status.tags}}'\
  '{{range $ii, $item := $tag.items}}{{if eq $item.image """${image_name}"'\
  $'"}}{{$is.metadata.name}}:{{$tag.tag}} at position {{$ii}} out of {{len $tag.items}}\n'\
  '{{end}}{{end}}{{end}}{{end}}'
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**。

  小心

  此操作实际上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。遗憾的是，这对旧版标签可能并不始终适合。

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，实际上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
```

1. 建议的解决方案是保护 registry 的安全。否则，您需要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

8.6. 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：

```
$ oc -n openshift-image-registry \
    rsh deploy/image-registry \
    /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 \
    rsh deploy/image-registry env REGISTRY_LOG_LEVEL=info \
    /usr/bin/dockerregistry -prune=check
```

已截断的输出示例

```
$ oc exec image-registry-3-vhndw \
    -- /bin/sh -c 'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'

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：

```
$ oc -n openshift-image-registry \
    rsh deploy/image-registry \
    /usr/bin/dockerregistry -prune=delete
```

   输**出示例**

```
$ oc exec image-registry-3-vhndw \
    -- /usr/bin/dockerregistry -prune=delete

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
```

8.7. 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