OpenShift Container Platform 4.7

CLI 工具

如何使用 OpenShift Container Platform 的命令行工具

Last Updated: 2021-03-30

OpenShift Container Platform 4.7 CLI 工具

如何使用 OpenShift Container Platform 的命令行工具

法律通告

Copyright © 2021 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文档提供有关安装、配置和使用 OpenShift Container Platform 命令行工具的信息。它还包含 CLI 命令的参考信息，以及如何使用它们的示例。

目录

第 1 章 OPENSHIFT CLI (OC) **3**
1.1. OPENSHIFT CLI 入门 3
1.2. 配置 OPENSHIFT CLI 10
1.3. 使用插件扩展 OPENSHIFT CLI 10
1.4. OPENSHIFT CLI 开发人员命令 12
1.5. OPENSHIFT CLI 管理员命令 24
1.6. OC 和 KUBECTL 命令的使用方法 29

第 2 章 开发人员 CLI (ODO) **31**
2.1. 了解 ODO 31
2.2. 安装 ODO 33
2.3. 使用 ODO 创建和部署应用程序 36
2.4. 在受限环境中使用 ODO 68
2.5. 创建 OPERATOR 管理的服务实例 76
2.6. 管理环境变量 78
2.7. 配置 ODO CLI 79
2.8. ODO CLI 参考指南 80
2.9. ODO 架构 90
2.10. ODO 发行注记 94

第 3 章 HELM CLI **96**
3.1. 在 OPENSHIFT CONTAINER PLATFORM 中使用 HELM 3 96
3.2. 配置自定义 HELM CHART 仓库 99
3.3. 禁用 HELM HART 仓库 102

第 4 章 用于 OPENSHIFT SERVERLESS 的 KNATIVE CLI（KN） **104**
4.1. 主要特性 104
4.2. 安装 KNATIVE CLI 104

第 5 章 PIPELINES CLI (TKN) **105**
5.1. 安装 TKN 105
5.2. 配置 OPENSHIFT PIPELINES TKN CLI 107
5.3. OPENSHIFT PIPELINES TKN 参考 107

第 6 章 OPM CLI **118**
6.1. 关于 OPM 118
6.2. 安装 OPM 118
6.3. 其他资源 119

第 7 章 OPERATOR SDK **120**
7.1. 安装 OPERATOR SDK CLI 120
7.2. OPERATOR SDK CLI 参考 121

第 1 章 OPENSHIFT CLI (OC)

1.1. OPENSHIFT CLI 入门

1.1.1. 关于 OpenShift CLI

使用 OpenShift 命令行界面（CLI），**oc**命令，您可以通过终端创建应用程序并管理 OpenShift Container Platform 项目。OpenShift CLI 在以下情况下是理想的选择：

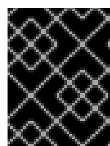
- 直接使用项目源代码
- 编写 OpenShift Container Platform 操作脚本
- 在管理项目时，受带宽资源的限制，Web 控制台无法使用

1.1.2. 安装 OpenShift CLI

您可以通过下载二进制文件或使用 RPM 来安装 OpenShift CLI（**oc**）。

1.1.2.1. 通过下载二进制文件安装 OpenShift CLI

您需要安装 CLI（**oc**） 来使用命令行界面与 OpenShift Container Platform 进行交互。您可在 Linux 、Windows 或 macOS 上安装 **oc**。



重要

如果安装了旧版本的 **oc**，则无法使用 OpenShift Container Platform 4.7 中的所有命令。下载并安装新版本的 **oc**。

1.1.2.1.1. 在 Linux 上安装 OpenShift CLI

您可以按照以下流程在 Linux 上安装 OpenShift CLI（**oc**）二进制文件。

流程

1. 访问 Red Hat OpenShift Cluster Manager 网站的 Infrastructure Provider 页面。
2. 选择您的基础架构供应商及安装类型。
3. 在 **Command-line interface** 部分，从下拉菜单中选择 **Linux**，并点 **Download command-line tools**。
4. 解包存档：

```
$ tar xvzf <file>
```

5. 把 **oc** 二进制代码放到 **PATH** 中的目录中。
执行以下命令可以查看当前的 **PATH** 设置：

```
$ echo $PATH
```

安装 CLI 后，就可以使用**oc**命令：

```
$ oc <command>
```

1.1.2.1.2. 在 Windows 上安装 OpenShift CLI

您可以按照以下流程在 Windows 上安装 OpenShift CLI（**oc**）二进制代码。

流程

1. 访问 Red Hat OpenShift Cluster Manager 网站的 Infrastructure Provider 页面。
2. 选择您的基础架构供应商及安装类型。
3. 在 **Command-line interface** 部分，从下拉菜单中选择 **Windows**，点 **Download command-line tools**。
4. 使用 ZIP 程序解压存档。
5. 把 **oc** 二进制代码放到 **PATH** 中的目录中。
   要查看您的 **PATH**，请打开命令提示窗口并执行以下命令：

   ```
   C:\> path
   ```

安装 CLI 后，就可以使用**oc**命令：

```
C:\> oc <command>
```

1.1.2.1.3. 在 macOS 上安装 OpenShift CLI

您可以按照以下流程在 macOS 上安装 OpenShift CLI（**oc**）二进制代码。

流程

1. 访问 Red Hat OpenShift Cluster Manager 网站的 Infrastructure Provider 页面。
2. 选择您的基础架构供应商及安装类型。
3. 在 **Command-line interface** 部分，从下拉菜单中选择 **MacOS**，并点 **Download command-line tools**。
4. 解包和解压存档。
5. 将 **oc** 二进制文件移到 PATH 的目录中。
   要查看您的 **PATH**，打开一个终端窗口并执行以下命令：

   ```
   $ echo $PATH
   ```

安装 CLI 后，就可以使用**oc**命令：

```
$ oc <command>
```

1.1.2.2. 使用 RPM 安装 OpenShift CLI

对于 Red Hat Enterprise Linux (RHEL)，如果您的红帽帐户中包括有效的 OpenShift Container Platform 订阅，则可将通过 RPM 安装 OpenShift CLI (**oc**)。

先决条件

- 必须具有 root 或 sudo 权限。

流程

1. 使用 Red Hat Subscription Manager 注册：

```
# subscription-manager register
```

2. 获取最新的订阅数据：

```
# subscription-manager refresh
```

3. 列出可用的订阅：

```
# subscription-manager list --available --matches '*OpenShift*'
```

4. 在上一命令的输出中，找到 OpenShift Container Platform 订阅的池 ID，并把订阅附加到注册的系统：

```
# subscription-manager attach --pool=<pool_id>
```

5. 启用 OpenShift Container Platform 4.7 所需的仓库。

   - Red Hat Enterprise Linux 8：

   ```
   # subscription-manager repos --enable="rhocp-4.7-for-rhel-8-x86_64-rpms"
   ```

   - Red Hat Enterprise Linux 7：

   ```
   # subscription-manager repos --enable="rhel-7-server-ose-4.7-rpms"
   ```

6. 安装 **openshift-clients** 软件包：

```
# yum install openshift-clients
```

安装 CLI 后，就可以使用**oc**命令：

```
$ oc <command>
```

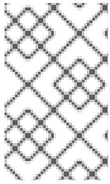
1.1.3. 登录到 OpenShift CLI

您可以登录到**oc** CLI 以访问和管理您的群集。

先决条件

- 有访问 OpenShift Container Platform 集群的权限。

- 已安装CLI。



注意

要访问只能通过 HTTP 代理服务器访问的集群，可以设置 **HTTP_PROXY**、**HTTPS_PROXY** 和 **NO_PROXY** 变量。**oc** CLI 会使用这些环境变量以便所有与集群的通信都通过 HTTP 代理进行。

流程

- 使用**oc login**命令登录到 CLI，根据提示输入所需信息。

```
$ oc login
```

输出示例

```
Server [https://localhost:8443]: https://openshift.example.com:6443 (1)
The server uses a certificate signed by an unknown authority.
You can bypass the certificate check, but any data you send to the server could be
intercepted by others.
Use insecure connections? (y/n): y (2)

Authentication required for https://openshift.example.com:6443 (openshift)
Username: user1 (3)
Password: (4)
Login successful.

You don't have any projects. You can try to create a new project, by running

    oc new-project <projectname>

Welcome! See 'oc help' to get started.
```

1. 输入 OpenShift Container Platform 服务器的 URL。
2. 输入是否使用不安全的连接。
3. 输入要登录的用户名。
4. 输入用户密码。

您现在可以创建项目或执行其他命令来管理集群。

1.1.4. 使用 OpenShift CLI

参阅以下部分以了解如何使用 CLI 完成常见任务。

1.1.4.1. 创建一个项目

使用**oc new-project**命令创建新项目。

```
$ oc new-project my-project
```

输出示例

```
Now using project "my-project" on server "https://openshift.example.com:6443".
```

1.1.4.2. 创建一个新的应用程序

使用**oc new-app**命令创建新应用程序。

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

输出示例

```
--> Found image 40de956 (9 days old) in imagestream "openshift/php" under tag "7.2" for "php"

...

    Run 'oc status' to view your app.
```

1.1.4.3. 查看 pod

使用**oc get pods**命令查看当前项目的 pod。

```
$ oc get pods -o wide
```

输出示例

```
NAME                  READY   STATUS      RESTARTS   AGE     IP            NODE
NOMINATED NODE
cakephp-ex-1-build    0/1     Completed   0          5m45s   10.131.0.10   ip-10-0-141-74.ec2.internal
<none>
cakephp-ex-1-deploy   0/1     Completed   0          3m44s   10.129.2.9    ip-10-0-147-65.ec2.internal
<none>
cakephp-ex-1-ktz97    1/1     Running     0          3m33s   10.128.2.11   ip-10-0-168-105.ec2.internal
<none>
```

1.1.4.4. 查看 pod 日志

使用**oc logs**命令查看特定 pod 的日志。

```
$ oc logs cakephp-ex-1-deploy
```

输出示例

```
--> Scaling cakephp-ex-1 to 1
--> Success
```

1.1.4.5. 查看当前项目

使用**oc project**命令查看当前项目。

```
$ oc project
```

输出示例

```
Using project "my-project" on server "https://openshift.example.com:6443".
```

1.1.4.6. 查看当前项目的状态

使用 **oc status** 命令查看有关当前项目的信息，如服务、部署和构建配置。

```
$ oc status
```

输出示例

```
In project my-project on server https://openshift.example.com:6443

svc/cakephp-ex - 172.30.236.80 ports 8080, 8443
  dc/cakephp-ex deploys istag/cakephp-ex:latest <-
    bc/cakephp-ex source builds https://github.com/sclorg/cakephp-ex on openshift/php:7.2
    deployment #1 deployed 2 minutes ago - 1 pod

3 infos identified, use 'oc status --suggest' to see details.
```

1.1.4.7. 列出支持的 API 资源

使用**oc api-resources**命令查看服务器上支持的 API 资源列表。

```
$ oc api-resources
```

输出示例

```
NAME                    SHORTNAMES   APIGROUP                 NAMESPACED   KIND
bindings                                                      true         Binding
componentstatuses       cs                                    false        ComponentStatus
configmaps              cm                                    true         ConfigMap
...
```

1.1.5. 获得帮助

您可以通过以下方式获得有关 CLI 命令和 OpenShift Container Platform 资源的帮助信息。

- 使用**oc help**获取所有可用 CLI 命令的列表和描述：

  示例：获取 CLI 的常规帮助信息

  ```
  $ oc help
  ```

  输出示例

  ```
  OpenShift Client
  ```

```
This client helps you develop, build, deploy, and run your applications on any OpenShift or
Kubernetes compatible
platform. It also includes the administrative commands for managing a cluster under the 'adm'
subcommand.

Usage:
  oc [flags]

Basic Commands:
  login           Log in to a server
  new-project     Request a new project
  new-app         Create a new application

...
```

- 使用**--help**标志获取有关特定CLI命令的帮助信息：

示例：获取oc create命令的帮助信息

```
$ oc create --help
```

输出示例

```
Create a resource by filename or stdin

JSON and YAML formats are accepted.

Usage:
  oc create -f FILENAME [flags]

...
```

- 使用**oc explain**命令查看特定资源的描述信息和项信息：

示例：查看 Pod 资源的文档

```
$ oc explain pods
```

输出示例

```
KIND:     Pod
VERSION:  v1

DESCRIPTION:
     Pod is a collection of containers that can run on a host. This resource is
     created by clients and scheduled onto hosts.

FIELDS:
   apiVersion <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
```

```
https://git.k8s.io/community/contributors/devel/api-conventions.md#resources
...
```

1.1.6. 注销 OpenShift CLI

您可以注销 OpenShift CLI 以结束当前会话。

- 使用**oc logout**命令。

```
$ oc logout
```

输**出示例**

```
Logged "user1" out on "https://openshift.example.com"
```

这将从服务器中删除已保存的身份验证令牌，并将其从配置文件中删除。

1.2. 配置 OPENSHIFT CLI

1.2.1. 启用 tab 自动完成功能

在安装**oc** CLI工具后，可以启用 tab 自动完成功能，以便在按 Tab 键时自动完成 **oc**命令或显示建议选项。

先决条件

- 已安装**oc** CLI工具。

流程

以下过程为 Bash 启用 tab 自动完成功能。

1. 将Bash完成代码保存到一个文件中。

```
$ oc completion bash > oc_bash_completion
```

2. 将文件复制到**/etc/bash_completion.d/** 。

```
$ sudo cp oc_bash_completion /etc/bash_completion.d/
```

您也可以将文件保存到一个本地目录，并从您的**.bashrc**文件中 source 这个文件。

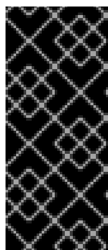
开新终端时 tab 自动完成功能将被启用。

1.3. 使用插件扩展 OPENSHIFT CLI

您可以针对默认的**oc**命令编写并安装插件，从而可以使用OpenShift Container Platform CLI执行新的及更复杂的任务。

1.3.1. 编写CLI插件

您可以使用任何可以编写命令行命令的编程语言或脚本为OpenShift Container Platform CLI编写插件。请注意，您不能使用插件来覆盖现有的**oc**命令。



重要

OpenShift CLI插件目前是技术预览功能。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关详细信息，请参阅红帽技术预览功能支持范围 。

流程

此过程创建一个简单的Bash插件，它的功能是在执行**oc foo**命令时将消息输出到终端。

1. 创建一个名为**oc-foo**的文件。
   在命名插件文件时，请记住以下几点：

   - 文件必须以**oc-**或**kubectl-**开始。
   - 文件名决定了调用这个插件的命令。例如，**oc foo bar**命令将会调用文件名为**oc-foo-bar**的插件。如果希望命令中包含破折号，也可以使用下划线。例如，可以通过**oc foo-bar**命令调用文件名为**oc-foo_bar**的插件。

2. 将以下内容添加到该文件中。

```
#!/bin/bash

# optional argument handling
if [[ "$1" == "version" ]]
then
    echo "1.0.0"
    exit 0
fi

# optional argument handling
if [[ "$1" == "config" ]]
then
    echo $KUBECONFIG
    exit 0
fi

echo "I am a plugin named kubectl-foo"
```

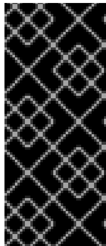
为OpenShift Container Platform CLI安装此插件后，可以使用**oc foo**命令调用它。

其它资源

- 查看 Sample plug-in repository 中使用 Go 编写的插件示例。
- 查看 CLI runtime repository 中的一组用来帮助使用 Go 编写插件的工具程序。

1.3.2. 安装和使用CLI插件

为OpenShift Container Platform CLI编写自定义插件后，必须安装它以使用它提供的功能。



重要

OpenShift CLI插件目前是技术预览功能。技术预览功能不包括在红帽生产服务级别协议（SLA）中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关详细信息，请参阅红帽技术预览功能支持范围 。

先决条件

- 已安装**oc** CLI工具。
- 有一个 CLI 插件文件，其文件名以 **oc-** 或 **kubectl-**开始。

流程

1. 将插件文件设置为可执行文件。

```
$ chmod +x <plugin_file>
```

2. 将文件放在**PATH**中的任何位置，例如/**usr/local/bin/** 。

```
$ sudo mv <plugin_file> /usr/local/bin/.
```

3. 运行 **oc plugin list** 以确认插件可以被列出。

```
$ oc plugin list
```

输出示例

```
The following compatible plugins are available:

/usr/local/bin/<plugin_file>
```

如果您的插件没有被列出，检查文件是否以**OC-**或**kubectl-**开始，是否可执行，并在您的**PATH**中。

4. 调用插件引入的新命令或选项。
例如，如果您从 Sample plug-in repository 构建并安装了 **kubectl-ns** 插件，则可以使用以下命令查看当前命名空间。

```
$ oc ns
```

请注意，调用插件的命令取决于插件的文件名。例如，文件名为 **oc-foo-bar**的插件会被 **oc foo bar** 命令调用。

1.4. OPENSHIFT CLI 开发人员命令

1.4.1. 基本CLI命令

1.4.1.1. explain

显示特定资源的文档。

示例：显示 pod 的文档

```
$ oc explain pods
```

1.4.1.2. login

登录OpenShift Container Platform服务器并保存登录信息以供后续使用。

示例：交互式登录

```
$ oc login
```

示例：指定用户名进行登陆

```
$ oc login -u user1
```

1.4.1.3. new-app

通过指定源代码，模板或镜像来创建新应用程序。

示例：从本地Git仓库创建新应用程序

```
$ oc new-app .
```

示例：从远程Git仓库创建新应用程序

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

示例：从远程的一个私有Git仓库创建新应用程序

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

1.4.1.4. new-project

创建一个新项目，并切换到这个项目作为配置中的默认项目。

示例：创建一个新项目

```
$ oc new-project myproject
```

1.4.1.5. project

切换到另一个项目，并使其成为配置中的默认项目。

示例：切换到另外一个项目

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

1.4.1.6. projects

显示服务器上当前活动项目和现有项目的信息。

示例：列出所有项目

```
$ oc projects
```

1.4.1.7. status

显示当前项目的概况信息。

示例：显示当前项目的状态

```
$ oc status
```

1.4.2. 构建和部署CLI命令

1.4.2.1. cancel-build

取消正在运行，待处理或新的构建。

示例：取消一个构建

```
$ oc cancel-build python-1
```

示例：从 python 构建配置中取消所有待处理的构建

```
$ oc cancel-build buildconfig/python --state=pending
```

1.4.2.2. import-image

从镜像仓库中导入最新的 tag 和镜像信息。

示例：导入最新的镜像信息

```
$ oc import-image my-ruby
```

1.4.2.3. new-build

从源代码创建新构建配置。

示例：从本地 Git 仓库创建构建配置

```
$ oc new-build .
```

示例：从远程 Git 仓库创建构建配置

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

1.4.2.4. rollback

将应用程序还原回以前的部署。

示例：回滚到上次成功部署

```
$ oc rollback php
```

示例：回滚到一个特定版本

```
$ oc rollback php --to-version=3
```

1.4.2.5. rollout

开始一个新的 rollout 操作，查看它的状态或历史信息，或回滚到应用程序的一个以前的版本。

示例：回滚到上次成功部署

```
$ oc rollout undo deploymentconfig/php
```

示例：使用最新状态启动一个新的部署 rollout

```
$ oc rollout latest deploymentconfig/php
```

1.4.2.6. start-build

从构建配置启动构建或复制现有构建。

示例：从指定的构建配置启动构建

```
$ oc start-build python
```

示例：从以前的一个构建版本开始进行构建

```
$ oc start-build --from-build=python-1
```

示例：为当前构建设置要使用的环境变量

```
$ oc start-build python --env=mykey=myvalue
```

1.4.2.7. tag

将现有镜像标记为镜像流。

示例：配置ruby镜像的latest tag 指向2.0 tag

```
$ oc tag ruby:latest ruby:2.0
```

1.4.3. 应用程序管理CLI命令

1.4.3.1. annotate

更新一个或多个资源上的注解。

示例：向路由添加注解

```
$ oc annotate route/test-route haproxy.router.openshift.io/ip_whitelist="192.168.1.10"
```

示例：从路由中删除注解

```
$ oc annotate route/test-route haproxy.router.openshift.io/ip_whitelist-
```

1.4.3.2. apply

通过文件名或标准输入（stdin）以JSON或YAML格式将配置应用于资源。

示例：将 pod.json 中的配置应用到 pod

```
$ oc apply -f pod.json
```

1.4.3.3. autoscale

自动缩放部署或复制控制器。

示例：自动缩放至最少两个最多五个 pod

```
$ oc autoscale deploymentconfig/parksmap-katacoda --min=2 --max=5
```

1.4.3.4. create

通过文件名或标准输入（stdin）使用 JSON 或 YAML 格式创建一个资源。

示例：使用 pod.json 中的内容创建一个 pod

```
$ oc create -f pod.json
```

1.4.3.5. delete

删除一个资源。

示例：删除名为 parksmap-katacoda-1-qfqz4 的 pod

```
$ oc delete pod/parksmap-katacoda-1-qfqz4
```

示例：删除所有带有 app=parksmap-katacoda 标签的 pod

```
$ oc delete pods -l app=parksmap-katacoda
```

1.4.3.6. describe

获得有关一个特定对象的详细信息。

示例：描述名为 example 的部署

```
$ oc describe deployment/example
```

示例：描述所有 pod

```
$ oc describe pods
```

1.4.3.7. edit

编辑一个资源。

示例：使用默认编辑器编辑部署

```
$ oc edit deploymentconfig/parksmap-katacoda
```

示例：使用不同的编辑器编辑部署

```
$ OC_EDITOR="nano" oc edit deploymentconfig/parksmap-katacoda
```

示例：编辑 JSON 格式的部署

```
$ oc edit deploymentconfig/parksmap-katacoda -o json
```

1.4.3.8. expose

以外部方式公开服务作为路由。

示例：开放一个服务

```
$ oc expose service/parksmap-katacoda
```

示例：开放服务并指定主机名

```
$ oc expose service/parksmap-katacoda --hostname=www.my-host.com
```

1.4.3.9. get

显示一个或多个资源。

示例：列出 default 命名空间中的 pod

```
$ oc get pods -n default
```

示例：获取 JSON 格式的 python 部署详情

```
$ oc get deploymentconfig/python -o json
```

1.4.3.10. label

更新一个或多个资源上的标签。

示例：更新 python-1-mz2rf pod，标签 status 设置为 unhealthy

```
$ oc label pod/python-1-mz2rf status=unhealthy
```

1.4.3.11. scale

设置复制控制器或部署所需的副本数。

示例：将 ruby-app 部署扩展为三个 pod

```
$ oc scale deploymentconfig/ruby-app --replicas=3
```

1.4.3.12. secrets

管理项目中的 secret

示例：default 服务账户(service account)使用 my-pull-secret作为 image pull 操作的 secret

```
$ oc secrets link default my-pull-secret --for=pull
```

1.4.3.13. serviceaccounts

获取分配给服务帐户的令牌或，或为服务帐户创建新令牌或**kubeconfig**文件。

示例：获取分配给default服务帐户的令牌

```
$ oc serviceaccounts get-token default
```

1.4.3.14. set

配置现有应用资源。

示例：设置构建配置上的 secret 的名称

```
$ oc set build-secret --source buildconfig/mybc mysecret
```

1.4.4. 调试CLI命令

1.4.4.1. attach

为正在运行的容器附加一个 shell。

示例：从 pod python-1-mz2rf 获取python 容器的输出信息

```
$ oc attach python-1-mz2rf -c python
```

1.4.4.2. cp

将文件和目录复制到容器或从容器中复制。

示例：将文件从 python-1-mz2rf pod 复制到本地文件系统

```
$ oc cp default/python-1-mz2rf:/opt/app-root/src/README.md ~/mydirectory/.
```

1.4.4.3. debug

启动一个 shell 以调试正在运行的应用程序。

示例：调试 python 部署

```
$ oc debug deploymentconfig/python
```

1.4.4.4. exec

在容器中执行命令。

示例：从 pod python-1-mz2rf 的python 容器中执行ls命令

```
$ oc exec python-1-mz2rf -c python ls
```

1.4.4.5. logs

检索特定构建、构建配置、部署或 Pod 的日志输出。

示例：从 python 部署中获得最新的日志

```
$ oc logs -f deploymentconfig/python
```

1.4.4.6. port-forward

将一个或多个本地端口转发到一个 pod。

示例：在本地侦听端口 8888 并将其数据转发到 pod 的端口 5000

```
$ oc port-forward python-1-mz2rf 8888:5000
```

1.4.4.7. proxy

运行到Kubernetes API服务器的代理。

示例：在端口8011上运行到API服务器的代理，由./local/www/提供静态内容

```
$ oc proxy --port=8011 --www=./local/www/
```

1.4.4.8. rsh

打开到容器的远程shell会话。

示例：在 python-1-mz2rf pod 中的第一个容器上打开一个 shell 会话

```
$ oc rsh python-1-mz2rf
```

1.4.4.9. rsync

将目录的内容复制到正在运行的 pod 容器或从容器中复制。**rsync**命令只复制您的操作系统中已更改的文件。

示例：将本地目录中的文件与 pod 目录同步

```
$ oc rsync ~/mydirectory/ python-1-mz2rf:/opt/app-root/src/
```

1.4.4.10. run

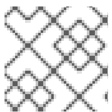
创建运行特定镜像的 pod。

示例：启动运行 perl 镜像的 pod

```
$ oc run my-test --image=perl
```

1.4.4.11. wait

等待一个或多个资源上的特定条件。



注意

此命令是实验性的，如有变更，恕不另行通知。

示例：等待 python-1-mz2rf pod 被删除

```
$ oc wait --for=delete pod/python-1-mz2rf
```

1.4.5. 高级开发人员CLI命令

1.4.5.1. api-resources

显示服务器支持的完整API资源列表。

示例：列出支持的API资源

```
$ oc api-resources
```

1.4.5.2. api-versions

显示服务器支持的完整API版本列表。

示例：列出支持的API版本

```
$ oc api-versions
```

1.4.5.3. auth

检查权限并协调RBAC角色。

示例：检查当前用户是否可以读取 pod 日志

```
$ oc auth can-i get pods --subresource=log
```

示例：从一个文件协调RBAC角色和权限

```
$ oc auth reconcile -f policy.json
```

1.4.5.4. cluster-info

显示 master 和集群服务的地址。

示例：显示集群信息

```
$ oc cluster-info
```

1.4.5.5. convert

将YAML或JSON配置文件转换为一个不同的API版本并打印到标准输出（stdout）。

示例：将pod.yaml转换到最新版本

```
$ oc convert -f pod.yaml
```

1.4.5.6. extract

提取配置映射或 secret 的内容。配置映射或 secret 中的每个密钥都被创建为拥有密钥名称的独立文件。

示例：将 ruby-1-ca 配置映射的内容下载到当前目录

```
$ oc extract configmap/ruby-1-ca
```

示例：将 ruby-1-ca 配置映射的内容输出到 stdout

```
$ oc extract configmap/ruby-1-ca --to=-
```

1.4.5.7. idle

把可扩展资源设置为空闲。一个空闲的服务在接收到数据时将自动变为非空闲状态，或使用**oc scale**命令手动把空闲服务变为非空闲。

示例：把 ruby-app 服务变为空闲状态

```
$ oc idle ruby-app
```

1.4.5.8. image

管理OpenShift Container Platform集群中的镜像。

示例：把一个镜像复制到另外一个 tag

```
$ oc image mirror myregistry.com/myimage:latest myregistry.com/myimage:stable
```

1.4.5.9. observe

观察资源的变化并对其采取措施。

示例：观察服务的更改

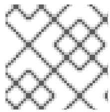
```
$ oc observe services
```

1.4.5.10. patch

使用JSON或YAML格式的策略合并补丁更新对象的一个或多个字段。

示例：将节点node1的spec.unschedulable字段更新为true

```
$ oc patch node/node1 -p '{"spec":{"unschedulable":true}}'
```



注意

如果需要使用自定义资源定义，则必须在命令中包含**--type merge**选项。

1.4.5.11. policy

管理授权策略。

示例：将edit角色添加给当前项目的user1用户

```
$ oc policy add-role-to-user edit user1
```

1.4.5.12. process

将模板处理为资源列表。

示例：将template.json转换为资源列表并传递给oc create

```
$ oc process -f template.json | oc create -f -
```

1.4.5.13. registry

管理OpenShift Container Platform中集成的 registry。

示例：显示集成的 registry 的信息

```
$ oc registry info
```

1.4.5.14. replace

根据指定配置文件的内容修改现有对象。

示例：使用pod.json的内容更新 Pod

```
$ oc replace -f pod.json
```

1.4.6. 设置CLI命令

1.4.6.1. completion

输出指定shell的shell完成代码。

示例：显示Bash的完成代码

```
$ oc completion bash
```

1.4.6.2. config

管理客户端配置文件。

示例：显示当前配置

```
$ oc config view
```

示例：切换到另外一个上下文

```
$ oc config use-context test-context
```

1.4.6.3. logout

退出当前会话。

示例：结束当前会话

```
$ oc logout
```

1.4.6.4. whoami

显示有关当前会话的信息。

示例：显示当前已验证的用户

```
$ oc whoami
```

1.4.7. 其他开发人员CLI命令

1.4.7.1. help

显示CLI的常规帮助信息和可用命令列表。

示例：显示可用命令

```
$ oc help
```

示例：显示new-project命令的帮助信息

```
$ oc help new-project
```

1.4.7.2. plugin

列出用户**PATH**中的可用插件。

示例：列出可用的插件

```
$ oc plugin list
```

1.4.7.3. version

显示**oc**客户端和服务器版本。

示例：显示版本信息

```
$ oc version
```

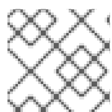
对于集群管理员，还会显示OpenShift Container Platform服务器版本。

1.5. OPENSHIFT CLI 管理员命令

1.5.1. 集群管理CLI命令

1.5.1.1. inspect

为特定资源收集调试信息。



注意

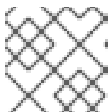
此命令是实验性的，如有变更，恕不另行通知。

示例：为 OpenShift API 服务器集群 Operator 收集调试数据

```
$ oc adm inspect clusteroperator/openshift-apiserver
```

1.5.1.2. must-gather

批量收集有关集群当前状态的数据以供进行问题调试。



注意

此命令是实验性的，如有变更，恕不另行通知。

示例：收集调试信息

```
$ oc adm must-gather
```

1.5.1.3. top

显示服务器上资源的使用情况统计信息。

示例：显示 pod 的 CPU 和内存使用情况

```
$ oc adm top pods
```

示例：显示镜像的使用情况统计信息

```
$ oc adm top images
```

1.5.2. 集群管理 CLI 命令

1.5.2.1. cordon

将节点标记为不可调度。手动将节点标记为不可调度将会阻止在此节点上调度任何新的pod，但不会影响节点上已存在的pod。

示例：将node1标记为不可调度

```
$ oc adm cordon node1
```

1.5.2.2. drain

排空节点以准备进行维护。

示例：排空node1

```
$ oc adm drain node1
```

1.5.2.3. node-logs

显示并过滤节点日志。

示例：获取 NetworkManager的日志

```
$ oc adm node-logs --role master -u NetworkManager.service
```

1.5.2.4. taint

更新一个或多个节点上的污点。

示例：添加污点以为一组用户指定一个节点

```
$ oc adm taint nodes node1 dedicated=groupName:NoSchedule
```

示例：使用 dedicated 从节点 node1上删除污点

```
$ oc adm taint nodes node1 dedicated-
```

1.5.2.5. uncordon

将节点标记为可调度。

示例：将node1标记为可调度

```
$ oc adm uncordon node1
```

1.5.3. 安全和策略CLI命令

1.5.3.1. certificate

批准或拒绝证书签名请求（CSR）。

示例：批准一个 CSR

```
$ oc adm certificate approve csr-sqgzp
```

1.5.3.2. groups

管理集群中的组。

示例：创建一个新组

```
$ oc adm groups new my-group
```

1.5.3.3. new-project

创建一个新项目并指定管理选项。

示例：使用节点选择器创建新项目

```
$ oc adm new-project myproject --node-selector='type=user-node,region=east'
```

1.5.3.4. pod-network

管理集群中的 Pod 网络。

示例：将 project1 和 project2 与其他非全局项目隔离

```
$ oc adm pod-network isolate-projects project1 project2
```

1.5.3.5. policy

管理集群上的角色和策略。

示例：为user1用户添加所有项目的edit角色

```
$ oc adm policy add-cluster-role-to-user edit user1
```

示例：把 privileged安全上下文限制（scc）添加给一个服务账户

```
$ oc adm policy add-scc-to-user privileged -z myserviceaccount
```

1.5.4. 维护CLI命令

1.5.4.1. migrate

根据使用的子命令，将集群上的资源迁移到新版本或格式。

示例：执行所有存储对象的更新

```
$ oc adm migrate storage
```

示例：仅执行 pod 的更新

```
$ oc adm migrate storage --include=pods
```

1.5.4.2. prune

从服务器中删除旧版本的资源。

示例：删除旧版本的构建，包括那些构建配置已不存在的版本

```
$ oc adm prune builds --orphans
```

1.5.5. 配置CLI命令

1.5.5.1. create-bootstrap-policy-file

创建默认引导策略。

示例：创建一个带有默认引导策略的名为policy.json的文件

```
$ oc adm create-bootstrap-policy-file --filename=policy.json
```

1.5.5.2. create-bootstrap-project-template

创建引导程序项目模板。

示例：将YAML格式的引导项目模板输出到stdout

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

1.5.5.3. create-error-template

创建用于自定义错误页面的模板。

示例：创建输出到stdout的错误页模板

```
$ oc adm create-error-template
```

1.5.5.4. create-kubeconfig

基于客户端证书创建基本的**.kubeconfig**文件。

示例：使用提供的客户端证书创建.kubeconfig文件

```
$ oc adm create-kubeconfig \
  --client-certificate=/path/to/client.crt \
  --client-key=/path/to/client.key \
  --certificate-authority=/path/to/ca.crt
```

1.5.5.5. create-login-template

创建用于自定义登陆页面的模板。

示例：创建输出到stdout的登陆页模板

```
$ oc adm create-login-template
```

1.5.5.6. create-provider-selection-template

创建用于自定义供应商选择页面的模板。

示例：创建输出到stdout的供应商选择页模板

```
$ oc adm create-provider-selection-template
```

1.5.6. 其他管理员CLI命令

1.5.6.1. build-chain

输出构建的输入和依赖项。

示例： 输出perl镜像流的依赖项

```
$ oc adm build-chain perl
```

1.5.6.2. completion

输出指定 shell 的 **oc adm** 命令的 shell 完成代码。

示例：显示Bash的oc adm完成代码

```
$ oc adm completion bash
```

1.5.6.3. config

管理客户端配置文件。此命令与**oc config**命令具有相同的作用。

示例：显示当前配置

```
$ oc adm config view
```

示例：切换到另外一个上下文

```
$ oc adm config use-context test-context
```

1.5.6.4. release

管理OpenShift Container Platform发布过程的各个方面，例如查看有关发布的信息或检查发布的内容。

示例：生成两个版本之间的 changelog 信息，并保存到changelog.md

```
$ oc adm release info --changelog=/tmp/git \
    quay.io/openshift-release-dev/ocp-release:4.7.0-x86_64 \
    quay.io/openshift-release-dev/ocp-release:4.7.1-x86_64 \
    > changelog.md
```

1.5.6.5. verify-image-signature

使用本地公共GPG密钥验证导入到内部 registry 的一个镜像的镜像签名。

示例：验证nodejs镜像签名

```
$ oc adm verify-image-signature \
    sha256:2bba968aedb7dd2aafe5fa8c7453f5ac36a0b9639f1bf5b03f95de325238b288 \
    --expected-identity 172.30.1.1:5000/openshift/nodejs:latest \
    --public-key /etc/pki/rpm-gpg/RPM-GPG-KEY-redhat-release \
    --save
```

1.6. OC 和 KUBECTL 命令的使用方法

Kubernetes 命令行界面（CLI）**kubectl** 可以用来对 Kubernetes 集群运行命令。由于 OpenShift Container Platform 是经过认证的 Kubernetes 发行版本，因此您可以使用 OpenShift Container Platform 附带的受支持的 **kubectl** 二进制文件，或者使用 **oc** 二进制文件来获得扩展的功能。

1.6.1. oc 二进制文件

oc 二进制文件提供与 **kubectl** 二进制文件相同的功能，但它经过扩展，可原生支持额外的 OpenShift Container Platform 功能，包括：

- **对 OpenShift Container Platform 资源的完整支持**

DeploymentConfig、**BuildConfig**、**Route**、**ImageStream** 和 **ImageStreamTag** 对象等资源特定于 OpenShift Container Platform 发行版本，并基于标准 Kubernetes 原语构建。

- **身份验证**
  oc 二进制文件提供了一个内置 **login** 命令，此命令可进行身份验证，并让您处理 OpenShift Container Platform 项目，这会将 Kubernetes 命名空间映射到经过身份验证的用户。如需更多信息，请参阅了解身份验证。

- **附加命令**
  例如，借助附加命令 **oc new-app** 可以更轻松地使用现有源代码或预构建镜像来启动新的应用程序。同样，附加命令 **oc new-project** 让您可以更轻松地启动一个项目并切换到该项目作为您的默认项目。

1.6.2. kubectl 二进制文件

提供 **kubectl** 二进制文件的目的是为来自标准 Kubernetes 环境的新 OpenShift Container Platform 用户或者希望使用 **kubectl** CLI 的用户支持现有工作流和脚本。**kubectl** 的现有用户可以继续使用二进制文件与 Kubernetes 原语交互，而不需要对 OpenShift Container Platform 集群进行任何更改。

您可以按照安装 OpenShift CLI 的步骤安装受支持的 **kubectl** 二进制文件。如果您下载二进制文件，或者在使用 RPM 安装 CLI 时安装，则 **kubectl** 二进制文件会包括在存档中。

如需更多信息，请参阅 kubectl 文档。

第 2 章 开发人员 CLI (ODO)

2.1. 了解 ODO

odo 是一个在 OpenShift Container Platform 和 Kubernetes 上创建应用程序的 CLI 工具。通过 **odo**，您可以在集群中编写、构建和调试应用程序，而无需对集群本身进行管理。**odo** 会自动创建部署配置、构建配置、服务路由和其他 OpenShift Container Platform 或 Kubernetes 元素。

现有工具如 **oc** 注重于操作，需要对 Kubernetes 和 OpenShift Container Platform 概念有深入了解。 **odo** 会处理与 Kubernetes 和 OpenShift Container Platform 相关的复杂概念，从而使开发人员可以把主要精力专注于最重要的内容：代码。

2.1.1. 主要特性

odo 的设计是简单而简洁的，其主要特性如下：

- 简单的语法，围绕开发人员熟悉的概念（比如项目、应用程序和组件）进行设计。
- 完全基于客户端。部署不需要 OpenShift Container Platform 以外的服务器。
- 对 Node.js 和 Java 组件的官方支持。
- 与各种语言和框架兼容，比如 Ruby 、Perl 、PHP 和 Python 部分兼容。
- 它会检测本地代码的更改，并自动将其部署到集群中，为验证更改提供即时反馈。
- 列出集群中的所有可用组件和服务。

2.1.2. 核心概念

Project（项目）

项目是一个用来对源代码、测试和库进行管理的独立空间。

Application（应用程序）

应用程序是为最终用户设计的程序。应用程序由多个微服务或组件组成，它们单独用来构建整个应用程序。应用程序示例：视频游戏、媒体播放器、网页浏览器。

组件

组件就是一组 Kubernetes 资源，用于托管代码或数据。每个组件可以单独运行和部署。组件示例：Node.js、Perl、PHP、Python、Ruby。

Service

服务是您的组件链接到或依赖的软件。服务示例：MariaDB、Jenkins、MySQL。在 **odo** 中，服务从 OpenShift Service Catalog 置备，且必须在集群中启用。

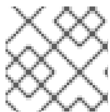
2.1.2.1. 官方支持的语言和相应的容器镜像

表 2.1. 支持的语言、容器镜像、软件包管理器和平台

语言	容器镜像	软件包管理器	平台
Node.js	rhscl/nodejs-10-rhel7	NPM	amd64、s390x、ppc64le

语言	容器镜像	软件包管理器	平台
	rhscl/nodejs-12-rhel7	NPM	amd64、s390x、ppc64le
Java	redhat-openjdk-18/openjdk18-openshift	Maven，Gradle	amd64、s390x、ppc64le
	openjdk/openjdk-11-rhel8	Maven，Gradle	amd64、s390x、ppc64le
	openjdk/openjdk-11-rhel7	Maven，Gradle	amd64、s390x、ppc64le

2.1.2.1.1. 列出可用的容器镜像



注意

可用的容器镜像列表从集群的内部容器 registry 以及与集群关联的外部 registry 提供。

要列出集群的可用组件和相关容器镜像，请执行以下操作：

1. 使用 **odo** 登陆到集群：

```
$ odo login -u developer -p developer
```

2. 要列出可用的 **odo** 支持和不支持的组件以及相应的容器镜像，请执行以下操作：

```
$ odo catalog list components
```

输出示例

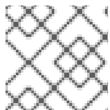
```
Odo Devfile Components:
NAME                 DESCRIPTION                            REGISTRY
java-maven           Upstream Maven and OpenJDK 11          DefaultDevfileRegistry
java-openliberty     Open Liberty microservice in Java      DefaultDevfileRegistry
java-quarkus         Upstream Quarkus with Java+GraalVM     DefaultDevfileRegistry
java-springboot      Spring Boot® using Java                DefaultDevfileRegistry
nodejs               Stack with NodeJS 12                   DefaultDevfileRegistry

Odo OpenShift Components:
NAME      PROJECT      TAGS                                                              SUPPORTED
java      openshift    11,8,latest                                                       YES
dotnet    openshift    2.1,3.1,latest                                                    NO
golang    openshift    1.13.4-ubi7,1.13.4-ubi8,latest                                    NO
httpd     openshift    2.4-el7,2.4-el8,latest                                            NO
nginx     openshift    1.14-el7,1.14-el8,1.16-el7,1.16-el8,latest                        NO
nodejs    openshift    10-ubi7,10-ubi8,12-ubi7,12-ubi8,latest                            NO
perl      openshift    5.26-el7,5.26-ubi8,5.30-el7,latest                                NO
php       openshift    7.2-ubi7,7.2-ubi8,7.3-ubi7,7.3-ubi8,latest                        NO
python    openshift    2.7-ubi7,2.7-ubi8,3.6-ubi7,3.6-ubi8,3.8-ubi7,3.8-ubi8,latest
```

```
NO
ruby        openshift     2.5-ubi7,2.5-ubi8,2.6-ubi7,2.6-ubi8,2.7-ubi7,latest                NO
wildfly     openshift
10.0,10.1,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,8.1,9.0,latest     NO
```

TAGS 列代表可用镜像版本，例如：**10** 代表 **rhoar-nodejs/nodejs-10** 容器镜像。要了解更多有关 CLI 命令的信息，请参阅 odo CLI 参考。

2.2. 安装 ODO

下面的部分论述了如何使用 CLI 或 Visual Studio Code（VS Code）IDE 在不同的平台中安装 **odo**。



注意

目前，**odo** 不支持在限制的网络环境中安装。

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

2.2.1. 在 Linux 中安装 odo

2.2.1.1. 二进制安装

流程

1. 获取二进制文件：

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

2. 更改该文件的权限：

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

2.2.1.2. tarball 安装

流程

1. 获取 tarball:

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-linux-
amd64.tar.gz | gzip -d > /usr/local/bin/odo'
```

2. 更改该文件的权限：

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

2.2.2. 在 IBM Power 的 Linux 上安装 odo

2.2.2.1. 二进制安装

流程

1. 获取二进制文件：

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

2. 更改该文件的权限：

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

2.2.2.2. tarball 安装

流程

1. 获取 tarball:

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-linux-ppc64le.tar.gz | gzip -d > /usr/local/bin/odo'
```

2. 更改该文件的权限：

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

2.2.3. 在 IBM Z 和 LinuxONE 的 Linux 中安装 odo

2.2.3.1. 二进制安装

流程

1. 获取二进制文件：

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

2. 更改该文件的权限：

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

2.2.3.2. tarball 安装

流程

1. 获取 tarball:

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-linux-s390x.tar.gz | gzip -d > /usr/local/bin/odo'
```

2. 更改该文件的权限：

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

2.2.4. 在 Windows 中安装 odo

2.2.4.1. 二进制安装

1. 下载最新的 odo.exe 文件。
2. 将 **odo.exe** 所在位置添加到 **GOPATH/bin** 目录中。

为 Windows 7/8 设置PATH 变量
以下示例演示了如何设置路径变量。您的二进制文件可以位于任何位置，在本例中使用 **C:\go-bin**。

1. 在 **C:\go-bin**创建一个文件夹。
2. 右键点击 Start 并点击 Control Panel。
3. 选择 **系统和安全性**， 然后点击 **系统**。
4. 在左侧的菜单中选择**高级系统设置**并点击底部的**环境变量**按钮。
5. 在变量部分选择**路径**并点**编辑**。
6. 点**新建**并输入**C:\go-bin**， 或者点击 **浏览** 并选择目录，然后点击 **确定**。

为 Windows 10 设置PATH 变量
使用搜索编辑 **环境变量** ：

1. 点击 **搜索** 并输入 **env** 或者 **environment**。
2. 选择为**您的帐户编辑环境变量**。
3. 在变量部分选择**路径**并点**编辑**。
4. 点**新建**并输入**C:\go-bin**， 或者点击 **浏览** 并选择目录，然后点击 **确定**。

2.2.5. 在 macOS 中安装 odo

2.2.5.1. 二进制安装

流程

1. 获取二进制文件：

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

2. 更改该文件的权限：

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

2.2.5.2. tarball 安装

流程

1. 获取 tarball:

```
# sh -c 'curl -L https://mirror.openshift.com/pub/openshift-v4/clients/odo/latest/odo-darwin-amd64.tar.gz | gzip -d > /usr/local/bin/odo'
```

2. 更改该文件的权限：

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

2.2.6. 在 VS Code 上安装 odo

OpenShift VS Code 扩展 使用 **odo** 和 **oc** 二进制文件来与 OpenShift Container Platform 集群交互。要使用这些功能，在 VS Code 中安装 OpenShift VS Code 扩展。

先决条件

- 您已安装了 VS Code。

流程

1. 打开 VS Code.
2. 使用 **Ctrl**+**P** 启动 VS Code Quick Open。
3. 使用以下命令：

```
$ ext install redhat.vscode-openshift-connector
```

2.3. 使用 ODO 创建和部署应用程序

2.3.1. 处理项目

项目将源代码、测试和库保存在一个单独的单元中。

2.3.1.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓ Project 'myproject' is ready for use
✓ New project created and now using project : myproject
```

2.3.2. 使用 odo 创建单组件应用程序

使用 **odo**，您可以在集群中从创建和部署应用程序。

先决条件

- 已安装了 **odo**。
- 有一个正在运行的集群。您可以使用 CodeReady Containers (CRC) 来快速部署一个本地的集群。

2.3.2.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓ Project 'myproject' is ready for use
✓ New project created and now using project : myproject
```

2.3.2.2. 使用 odo 创建 Node.js 应用程序

要创建一个 Node.js 组件，请下载 Node.js 应用程序并使用 **odo**将源代码推送到您的集群中。

流程

1. 为您的组件创建一个新目录：

```
$ mkdir my_components && cd my_components
```

2. 下载示例 Node.js 应用程序：

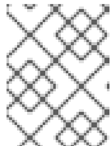
```
$ git clone https://github.com/openshift/nodejs-ex
```

3. 将当前目录变为您的应用程序的目录：

```
$ cd <directory_name>
```

4. 将类型 Node.js 的组件添加到应用程序中：

```
$ odo create nodejs
```



注意

默认情况下使用最新的镜像。也可以使用 **odo create openshift/nodejs:8** 明确指定一个镜像版本。

5. 将初始源代码推送到组件中：

```
$ odo push
```

现在，您的组件已被部署到 OpenShift Container Platform 中。

6. 创建一个 URL，按以下方法在本地配置文件中添加条目：

```
$ odo url create --port 8080
```

7. 推送更改。这会在集群中创建一个 URL。

```
$ odo push
```

8. 列出用于检查组件所需 URL 的 URL。

```
$ odo url list
```

9. 使用生成的 URL 查看部署的应用程序。

```
$ curl <url>
```

2.3.2.3. 修改应用程序代码

您可以修改应用程序代码，并将更改应用于 OpenShift Container Platform 上的应用程序。

1. 使用文本编辑器编辑 Node.js 目录中的一个布局文件。
2. 更新您的组件：

```
$ odo push
```

3. 刷新浏览器中的应用程序查看更改。

2.3.2.4. 在应用程序组件中添加存储

使用 **odo storage** 命令为应用程序添加持久性数据。必须持久化的数据示例包括数据库文件、依赖项和构建工件，如 **.m2** Maven 目录。

流程

1. 在组件中添加存储：

```
$ odo storage create <storage_name> --path=<path_to_the_directory> --size=<size>
```

2. 将存储推送到集群：

```
$ odo push
```

3. 通过列出组件中的所有存储来验证存储现在附加到您的组件中：

```
$ odo storage list
```

```
The component 'nodejs' has the following storage attached:
NAME        SIZE    PATH     STATE
mystorage    1Gi    /data    Pushed
```

4. 从您的组件中删除存储：

```
$ odo storage delete <storage_name>
```

5. 列出所有存储以验证存储状态是否**本地删除**:

```
$ odo storage list
```

```
The component 'nodejs' has the following storage attached:
NAME        SIZE    PATH     STATE
mystorage    1Gi    /data    Locally Deleted
```

6. 将更改推送到集群：

```
$ odo push
```

2.3.2.5. 添加自定义构建者来指定构建镜像

在 OpenShift Container Platform 中，您可以添加自定义镜像来缩小创建自定义镜像间的差距。

以下示例显示 **redhat-openjdk-18** 镜像已被成功导入并使用：

先决条件

- 安装了 OpenShift CLI (oc) 。

流程

1. 将镜像导入 OpenShift Container Platform：

```
$ oc import-image openjdk18 \
--from=registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift \
--confirm
```

2. 标记(tag)镜像使其可以被 odo 访问：

```
$ oc annotate istag/openjdk18:latest tags=builder
```

3. 使用 odo 部署镜像：

```
$ odo create openjdk18 --git \
https://github.com/openshift-evangelists/Wild-West-Backend
```

2.3.2.6. 使用 OpenShift Service Catalog 将应用程序连接到多个服务

OpenShift Service Catalog 是 Kubernetes 的 Open Service Broker API (OSB API) 的一个实现。您可以使用它将部署在 OpenShift Container Platform 中的应用程序连接到各种服务。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 在集群中安装并启用该 service catalog。

流程

- 要列出服务，请使用：

```
$ odo catalog list services
```

- 要使用 service catalog 相关的操作：

```
$ odo service <verb> <service_name>
```

2.3.2.7. 删除应用程序

使用 **odo app delete** 命令删除应用程序。

流程

1. 列出当前项目中的应用程序：

```
$ odo app list
```

输出示例

```
The project '<project_name>' has the following applications:
NAME
app
```

2. 列出与应用程序关联的组件。这些组件将随应用程序一起删除：

```
$ odo component list
```

输出示例

```
APP    NAME                      TYPE      SOURCE      STATE
app    nodejs-nodejs-ex-elyf     nodejs    file://./   Pushed
```

3. 删除应用程序：

```
$ odo app delete <application_name>
```

输出示例

```
? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

4. 使用 **Y** 确认删除。您可以使用 **-f** 标记来阻止确认提示。

2.3.3. 使用 odo 创建多组件应用程序

通过使用 **odo**，可以创建多组件应用程序，修改该应用程序，并以方便和自动的方式链接其组件。

这个例子描述了如何部署多组件应用程序 - 一个射击游戏。应用程序由一个前端 Node.js 组件和一个后端 Java 组件组成。

先决条件

- 已安装了 **odo**。
- 有一个正在运行的集群。开发人员可以使用 CodeReady Containers (CRC) 来快速部署一个本地的集群。
- 已安装 Maven。

2.3.3.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓  Project 'myproject' is ready for use
✓  New project created and now using project : myproject
```

2.3.3.2. 部署后端组件

要创建一个 Java 组件，请导入 Java 构建器镜像（builder image），下载 Java 应用程序并使用 **odo**将源代码推送到您的集群中。

流程

1. 将 **openjdk18** 导入集群：

```
$ oc import-image openjdk18 \
--from=registry.access.redhat.com/redhat-openjdk-18/openjdk18-openshift --confirm
```

2. 将镜像标记为 **builder**，使它可以被 odo 访问：

```
$ oc annotate istag/openjdk18:latest tags=builder
```

3. 运行 **odo catalog list components** 查看创建的镜像：

```
$ odo catalog list components
```

输出示例

```
Odo Devfile Components:
NAME                DESCRIPTION                          REGISTRY
java-maven          Upstream Maven and OpenJDK 11        DefaultDevfileRegistry
java-openliberty    Open Liberty microservice in Java    DefaultDevfileRegistry
java-quarkus        Upstream Quarkus with Java+GraalVM   DefaultDevfileRegistry
java-springboot     Spring Boot® using Java              DefaultDevfileRegistry
nodejs              Stack with NodeJS 12                 DefaultDevfileRegistry

Odo OpenShift Components:
NAME       PROJECT      TAGS                                                            SUPPORTED
java       openshift    11,8,latest                                                     YES
dotnet     openshift    2.1,3.1,latest                                                  NO
golang     openshift    1.13.4-ubi7,1.13.4-ubi8,latest                                  NO
httpd      openshift    2.4-el7,2.4-el8,latest                                          NO
nginx      openshift    1.14-el7,1.14-el8,1.16-el7,1.16-el8,latest                      NO
nodejs     openshift    10-ubi7,10-ubi8,12-ubi7,12-ubi8,latest                          NO
perl       openshift    5.26-el7,5.26-ubi8,5.30-el7,latest                              NO
php        openshift    7.2-ubi7,7.2-ubi8,7.3-ubi7,7.3-ubi8,latest                      NO
python     openshift    2.7-ubi7,2.7-ubi8,3.6-ubi7,3.6-ubi8,3.8-ubi7,3.8-ubi8,latest
NO
ruby       openshift    2.5-ubi7,2.5-ubi8,2.6-ubi7,2.6-ubi8,2.7-ubi7,latest             NO
wildfly    openshift
10.0,10.1,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,8.1,9.0,latest     NO
```

4. 为您的组件创建一个新目录：

```
$ mkdir my_components && cd my_components
```

5. 下载后端应用程序示例：

```
$ git clone https://github.com/openshift-evangelists/Wild-West-Backend backend
```

6. 进入后端源目录：

```
$ cd backend
```

7. 检查目录中有正确的文件：

```
$ ls
```

输出示例

```
debug.sh  pom.xml  src
```

8. 使用 Maven 构建后端源文件以创建一个 JAR 文件：

```
$ mvn package
```

输出示例

```
...
[INFO] -------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------
[INFO] Total time: 2.635 s
[INFO] Finished at: 2019-09-30T16:11:11-04:00
[INFO] Final Memory: 30M/91M
[INFO] -------------------------------------
```

9. 创建名为 **backend**的 Java 组件类型组件配置：

```
$ odo create --s2i openjdk18 backend --binary target/wildwest-1.0.jar
```

输出示例

```
 ✓  Validating component [1ms]
 Please use `odo push` command to create the component with source deployed
```

现在，配置文件 **config.yaml** 位于后端组件的本地目录中，其中包含用于部署的组件信息。

10. 使用以下方法检查 **config.yaml** 文件中后端组件的配置设置：

```
$ odo config view
```

输出示例

```
COMPONENT SETTINGS
------------------------------------------------
PARAMETER         CURRENT_VALUE
Type              openjdk18
Application       app
Project           myproject
SourceType        binary
Ref
SourceLocation    target/wildwest-1.0.jar
Ports             8080/TCP,8443/TCP,8778/TCP
Name              backend
MinMemory
MaxMemory
DebugPort
```

```
Ignore
MinCPU
MaxCPU
```

11. 将组件推送到 OpenShift Container Platform 集群。

```
$ odo push
```

输出示例

```
Validation
 ✓  Checking component [6ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [124ms]

Pushing to component backend of type binary
 ✓  Checking files for pushing [1ms]
 ✓  Waiting for component to start [48s]
 ✓  Syncing files to the component [811ms]
 ✓  Building component [3s]
```

使用 **odo push**，OpenShift Container Platform 创建一个容器来托管后端组件，将容器部署到运行在 OpenShift Container Platform 集群上的 pod 中，并启动 **backend** 组件。

12. 验证：

- odo 中操作的状态：

```
$ odo log -f
```

输出示例

```
2019-09-30 20:14:19.738  INFO 444 --- [           main] c.o.wildwest.WildWestApplication
: Starting WildWestApplication v1.0 onbackend-app-1-9tnhc with PID 444
(/deployments/wildwest-1.0.jar started by jboss in /deployments)
```

- 后端组件的状态：

```
$ odo list
```

输出示例

```
APP     NAME        TYPE         SOURCE                             STATE
app     backend     openjdk18    file://target/wildwest-1.0.jar     Pushed
```

2.3.3.3. 部署前端组件

要创建并部署前端组件，请下载 Node.js 应用程序并使用 **odo**将源代码推送到集群中。

流程

1. 下载前端应用程序示例：

```
$ git clone https://github.com/openshift/nodejs-ex frontend
```

2. 将当前目录改为前端目录：

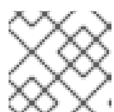
```
$ cd frontend
```

3. 列出目录的内容，以查看前端是一个 Node.js 应用程序。

```
$ ls
```

输出示例

```
README.md      openshift      server.js      views
helm           package.json   tests
```



注意

前端组件使用解释语言 (Node.js) 编写，不需要构建它。

4. 创建名为 **frontend**的 Node.js 组件类型组件配置：

```
$ odo create --s2i nodejs frontend
```

输出示例

```
 ✓  Validating component [5ms]
Please use `odo push` command to create the component with source deployed
```

5. 将组件推送到正在运行的容器中。

```
$ odo push
```

输出示例

```
Validation
 ✓  Checking component [8ms]

Configuration changes
 ✓  Initializing component
 ✓  Creating component [83ms]

Pushing to component frontend of type local
 ✓  Checking files for pushing [2ms]
 ✓  Waiting for component to start [45s]
 ✓  Syncing files to the component [3s]
 ✓  Building component [18s]
 ✓  Changes successfully pushed to component
```

2.3.3.4. 连接两个组件

集群中运行的组件需要连接才能进行交互。OpenShift Container Platform 提供了链接机制以发布一个程序到其客户端的通信绑定。

流程

1. 列出在集群中运行的所有组件：

```
$ odo list
```

输出示例

```
OpenShift Components:
APP     NAME        PROJECT     TYPE          SOURCETYPE     STATE
app     backend     testpro     openjdk18     binary         Pushed
app     frontend    testpro     nodejs        local          Pushed
```

2. 将当前的前端组件链接到后端：

```
$ odo link backend --port 8080
```

输出示例

```
 ✓  Component backend has been successfully linked from the component frontend

Following environment variables were added to frontend component:
- COMPONENT_BACKEND_HOST
- COMPONENT_BACKEND_PORT
```

后端组件的配置信息被添加到前端组件中，前端组件重新启动。

2.3.3.5. 公开组件

流程

1. 进入 **frontend** 目录：

```
$ cd frontend
```

2. 为应用程序创建一个外部 URL：

```
$ odo url create frontend --port 8080
```

输出示例

```
 ✓  URL frontend created for component: frontend

To create URL on the OpenShift  cluster, use `odo push`
```

3. 应用更改：

```
$ odo push
```

输出示例

```
Validation
 ✓  Checking component [21ms]

Configuration changes
 ✓  Retrieving component data [35ms]
 ✓  Applying configuration [29ms]

Applying URL changes
 ✓  URL frontend: http://frontend-app-myproject.192.168.42.79.nip.io created

Pushing to component frontend of type local
 ✓  Checking file changes for pushing [1ms]
 ✓  No file changes detected, skipping build. Use the '-f' flag to force the build.
```

4. 在一个浏览器中使用 URL 来查看应用程序。

> **注意**
>
> 如果一个应用程序需要一个有效的服务账户来访问 OpenShift Container Platform 命名空间并删除活跃的 pod 时，后端组件的 **odo log** 中可能会出现以下错误：
>
> **Message: Forbidden!Configured service account doesn’t have access.Service account may have been revoked**
>
> 要解决这个错误，请服务帐户角色添加权限：
>
> ```
> $ oc policy add-role-to-group view system:serviceaccounts -n <project>
> ```
>
> ```
> $ oc policy add-role-to-group edit system:serviceaccounts -n <project>
> ```
>
> 不要在生产环境集群中使用它。

2.3.3.6. 修改正在运行的应用程序

流程

1. 将本地目录改为前端目录：

```
$ cd frontend
```

2. 使用以下方法监控文件系统中的更改：

```
$ odo watch
```

3. 编辑 **index.html** 文件，为游戏更改显示的名称。

> **注意**
>
> 在 odo 可以识别更改前可能会有一些延迟。

odo 将更改推送到前端组件，并将其状态输出到终端：

```
File /root/frontend/index.html changed
File  changed
Pushing files...
 ✓  Waiting for component to start
 ✓  Copying files to component
 ✓  Building component
```

4. 在浏览器中刷新应用程序页面。现在会显示新名称。

2.3.3.7. 删除应用程序

使用 **odo app delete** 命令删除应用程序。

流程

1. 列出当前项目中的应用程序：

```
$ odo app list
```

输出示例

```
    The project '<project_name>' has the following applications:
    NAME
    app
```

2. 列出与应用程序关联的组件。这些组件将随应用程序一起删除：

```
$ odo component list
```

输出示例

```
    APP     NAME                      TYPE       SOURCE        STATE
    app     nodejs-nodejs-ex-elyf     nodejs     file://./     Pushed
```

3. 删除应用程序：

```
$ odo app delete <application_name>
```

输出示例

```
    ? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

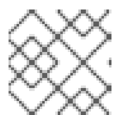
4. 使用 **Y** 确认删除。您可以使用 **-f** 标记来阻止确认提示。

2.3.4. 创建带有数据库的应用程序

这个示例描述了如何部署和连接数据库到前端应用程序。

先决条件

- 已安装了 **odo**。
- 已安装**oc** 客户端。
- 有一个正在运行的集群。开发人员可以使用 CodeReady Containers (CRC) 来快速部署一个本地的集群。
- 在集群中安装并启用该 Service Catalog。



注意

OpenShift Container Platform 4 及以后的版本弃用了 Service Catalog。

2.3.4.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓ Project 'myproject' is ready for use
✓ New project created and now using project : myproject
```

2.3.4.2. 部署前端组件

要创建并部署前端组件，请下载 Node.js 应用程序并使用 **odo**将源代码推送到集群中。

流程

1. 下载前端应用程序示例：

```
$ git clone https://github.com/openshift/nodejs-ex frontend
```

2. 将当前目录改为前端目录：

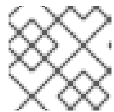
```
$ cd frontend
```

3. 列出目录的内容，以查看前端是一个 Node.js 应用程序。

```
$ ls
```

输出示例

```
README.md     openshift     server.js     views
helm          package.json    tests
```



注意

前端组件使用解释语言 (Node.js) 编写，不需要构建它。

4. 创建名为 **frontend**的 Node.js 组件类型组件配置：

```
$ odo create --s2i nodejs frontend
```

输出示例

```
 ✓  Validating component [5ms]
Please use `odo push` command to create the component with source deployed
```

5. 创建一个 URL 来访问前台接口。

```
$ odo url create myurl
```

输出示例

```
 ✓  URL myurl created for component: nodejs-nodejs-ex-pmdp
```

6. 将组件推送到 OpenShift Container Platform 集群。

```
$ odo push
```

输出示例

```
Validation
 ✓  Checking component [7ms]

 Configuration changes
 ✓  Initializing component
 ✓  Creating component [134ms]

 Applying URL changes
 ✓  URL myurl: http://myurl-app-myproject.192.168.42.79.nip.io created

 Pushing to component nodejs-nodejs-ex-mhbb of type local
 ✓  Checking files for pushing [657850ns]
 ✓  Waiting for component to start [6s]
 ✓  Syncing files to the component [408ms]
 ✓  Building component [7s]
 ✓  Changes successfully pushed to component
```

2.3.4.3. 以互动模式部署数据库

odo 提供了一个简化部署的命令行互动模式。

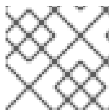
流程

- 运行互动模式并回答提示：

```
$ odo service create
```

输出示例

```
? Which kind of service do you wish to create database
? Which database service class should we use mongodb-persistent
? Enter a value for string property DATABASE_SERVICE_NAME (Database Service Name):
mongodb
? Enter a value for string property MEMORY_LIMIT (Memory Limit): 512Mi
? Enter a value for string property MONGODB_DATABASE (MongoDB Database Name):
sampledb
? Enter a value for string property MONGODB_VERSION (Version of MongoDB Image): 3.2
? Enter a value for string property VOLUME_CAPACITY (Volume Capacity): 1Gi
? Provide values for non-required properties No
? How should we name your service  mongodb-persistent
? Output the non-interactive version of the selected options No
? Wait for the service to be ready No
 ✓  Creating service [32ms]
 ✓  Service 'mongodb-persistent' was created
Progress of the provisioning will not be reported and might take a long time.
You can see the current status by executing 'odo service list'
```



注意

您的密码或用户名将作为环境变量传递给前端的应用程序。

2.3.4.4. 手动部署数据库

1. 列出可用服务：

```
$ odo catalog list services
```

输出示例

```
NAME                         PLANS
django-psql-persistent       default
jenkins-ephemeral            default
jenkins-pipeline-example     default
mariadb-persistent           default
mongodb-persistent           default
mysql-persistent             default
nodejs-mongo-persistent      default
postgresql-persistent        default
rails-pgsql-persistent       default
```

2. 选择 **mongodb-persistent** 服务类型，并查看所需参数：

```
$ odo catalog describe service mongodb-persistent
```

输出示例

```
***********************        | *****************************************************
Name                           | default
-----------------              | -----------------
Display Name                   |
-----------------              | -----------------
Short Description              | Default plan
-----------------              | -----------------
Required Params without a      |
default value                  |
-----------------              | -----------------
Required Params with a default | DATABASE_SERVICE_NAME
value                          | (default: 'mongodb'),
                               | MEMORY_LIMIT (default:
                               | '512Mi'), MONGODB_VERSION
                               | (default: '3.2'),
                               | MONGODB_DATABASE (default:
                               | 'sampledb'), VOLUME_CAPACITY
                               | (default: '1Gi')
-----------------              | -----------------
Optional Params                | MONGODB_ADMIN_PASSWORD,
                               | NAMESPACE, MONGODB_PASSWORD,
                               | MONGODB_USER
```

3. 指定所需参数，并等待数据库部署：

```
$ odo service create mongodb-persistent --plan default --wait -p
DATABASE_SERVICE_NAME=mongodb -p MEMORY_LIMIT=512Mi -p
MONGODB_DATABASE=sampledb -p VOLUME_CAPACITY=1Gi
```

2.3.4.5. 将数据库连接到前端应用程序

1. 将数据库连接到前端服务中：

```
$ odo link mongodb-persistent
```

输出示例

```
 ✓  Service mongodb-persistent has been successfully linked from the component nodejs-
nodejs-ex-mhbb

Following environment variables were added to nodejs-nodejs-ex-mhbb component:
- database_name
- password
- uri
- username
- admin_password
```

2. 查看 pod 中的应用程序和数据库的环境变量：

a. 获取 pod 名称：

```
$ oc get pods
```

输出示例

```
NAME                             READY   STATUS    RESTARTS   AGE
mongodb-1-gsznc                     1/1     Running   0          28m
nodejs-nodejs-ex-mhbb-app-4-vkn9l   1/1     Running   0          1m
```

b. 连接到 pod:

```
$ oc rsh nodejs-nodejs-ex-mhbb-app-4-vkn9l
```

c. 检查环境变量：

```
sh-4.2$ env
```

输出示例

```
uri=mongodb://172.30.126.3:27017
password=dHIOpYneSkX3rTLn
database_name=sampledb
username=user43U
admin_password=NCn41tqmx7RIqmfv
```

3. 在浏览器中打开 URL， 并在右下角记录数据库配置：

```
$ odo url list
```

输出示例

```
Request information
Page view count: 24

DB Connection Info:
Type: MongoDB
URL: mongodb://172.30.126.3:27017/sampledb
```

2.3.5. 创建带有数据库的 Java 应用程序

这个示例描述了如何使用 devfile 部署 Java 应用程序并将其连接到数据库服务。

先决条件

- 一个正在运行的集群。
- 已安装了 **odo**。
- 在集群中安装了 Service Binding Operator。了解如何安装 Operator，联络您的集群管理员，或参阅从 OperatorHub 安装 Operator。

- 在集群中安装了 Dev4Devs PostgreSQL Operator Operator。了解如何安装 Operator，联络您的集群管理员，或参阅从 OperatorHub 安装 Operator。

2.3.5.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓  Project 'myproject' is ready for use
✓  New project created and now using project : myproject
```

2.3.5.2. 使用 odo创建数据库

要创建数据库，您必须有权访问数据库 Operator。本例中使用了 Dev4Devs PostgreSQL Operator。

流程

1. 查看项目中的服务列表：

```
$ odo catalog list services
```

输出示例

```
Operators available in the cluster
NAME                              CRDs
postgresql-operator.v0.1.1        Backup, Database
```

2. 将服务的 YAML 存储到一个文件中：

```
$ odo service create postgresql-operator.v0.1.1/Database --dry-run > db.yaml
```

3. 在 **db.yaml** 文件中的 **metadata:** 部分中添加以下值：

```
  name: sampledatabase
  annotations:
    service.binding/db.name: 'path={.spec.databaseName}'
    service.binding/db.password: 'path={.spec.databasePassword}'
    service.binding/db.user: 'path={.spec.databaseUser}'
```

此配置可确保启动数据库服务时，将适当的注解添加到其中。注解可帮助 Service Binding Operator 将 **databaseName**、**databasePassword** 和 **databaseUser** 的值注入应用程序中。

4. 在 YAML 文件的 **spec:** 部分下更改以下值：

```
databaseName: "sampledb"
databasePassword: "samplepwd"
databaseUser: "sampleuser"
```

5. 从 YAML 文件创建数据库：

```
$ odo service create --from-file db.yaml
```

现在，项目中存在一个数据库实例。

2.3.5.3. 创建 Java MicroServices JPA 应用程序

使用 **odo**，您可以创建和管理一个 Java MicroServices JPA 应用程序示例。

流程

1. 克隆示例应用程序：

```
$ git clone https://github.com/redhat-developer/application-stack-samples
```

2. 进入到应用程序目录：

```
$ cd ./application-stack-samples/jpa
```

3. 初始化项目：

```
$ odo create java-openliberty java-application
```

4. 将应用程序推送到集群：

```
$ odo push
```

现在，应用程序已被部署到集群中。

5. 通过将 OpenShift 日志流传输到终端来查看集群的状态：

```
$ odo log
```

请注意，测试失败并有 **UnknownDatabaseHostException** 错误。这是因为您的应用程序还没有数据库：

```
[INFO] [err] java.net.UnknownHostException: ${DATABASE_CLUSTERIP}
[INFO] [err]     at
java.base/java.net.AbstractPlainSocketImpl.connect(AbstractPlainSocketImpl.java:220)
[INFO] [err]     at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:403)
[INFO] [err]     at java.base/java.net.Socket.connect(Socket.java:609)
[INFO] [err]     at org.postgresql.core.PGStream.<init>(PGStream.java:68)
[INFO] [err]     at
org.postgresql.core.v3.ConnectionFactoryImpl.openConnectionImpl(ConnectionFactoryImpl.jav
a:144)
[INFO] [err]     ... 86 more
```

```
[ERROR] Tests run: 2, Failures: 1, Errors: 1, Skipped: 0, Time elapsed: 0.706 s <<<
FAILURE! - in org.example.app.it.DatabaseIT
[ERROR] testGetAllPeople  Time elapsed: 0.33 s  <<< FAILURE!
org.opentest4j.AssertionFailedError: Expected at least 2 people to be registered, but there
were only: [] ==> expected: <true> but was: <false>
        at org.example.app.it.DatabaseIT.testGetAllPeople(DatabaseIT.java:57)

[ERROR] testGetPerson  Time elapsed: 0.047 s  <<< ERROR!
java.lang.NullPointerException
        at org.example.app.it.DatabaseIT.testGetPerson(DatabaseIT.java:41)

[INFO]
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   DatabaseIT.testGetAllPeople:57 Expected at least 2 people to be registered, but
there were only: [] ==> expected: <true> but was: <false>
[ERROR] Errors:
[ERROR]   DatabaseIT.testGetPerson:41 NullPointer
[INFO]
[ERROR] Tests run: 2, Failures: 1, Errors: 1, Skipped: 0
[INFO]
[ERROR] Integration tests failed: There are test failures.
```

6. 创建入口 URL 以访问应用程序：

```
$ odo url create --port 8080
```

7. 将更改推送到集群：

```
$ odo push
```

8. 显示创建的 URL:

```
$ odo url list
```

输出示例

```
Found the following URLs for component mysboproj
NAME               STATE      URL                                         PORT     SECURE     KIND
java-application-8080     Pushed     http://java-application-8080.apps-crc.testing     8080
false      ingress
```

现在，应用程序已部署到集群中，您可以使用创建的 URL 访问它。

9. 使用 URL 导航到 **CreatePerson.xhtml** 数据条目页面，并使用表单输入用户名和年龄。点 **Save**。
请注意：由于应用程序还没有连接数据库，您将无法通过点 **View Persons Record List** 链接来查看数据。

2.3.5.4. 将 Java 应用程序连接到数据库

要将 Java 应用程序连接到数据库，使用 **odo link** 命令。

流程

1. 显示服务列表：

```
$ odo service list
```

输出示例

```
NAME                      AGE
Database/sampledatabase   6m31s
```

2. 将数据库连接到您的应用程序：

```
$ odo link Database/sampledatabase
```

3. 将更改推送到集群：

```
$ odo push
```

创建并推送链接后，会创建一个包含数据库连接数据的 secret。

4. 检查组件中的从数据库服务注入的值：

```
$ odo exec -- bash -c 'export | grep DATABASE'
declare -x DATABASE_CLUSTERIP="10.106.182.173"
declare -x DATABASE_DB_NAME="sampledb"
declare -x DATABASE_DB_PASSWORD="samplepwd"
declare -x DATABASE_DB_USER="sampleuser"
```

5. 打开 Java 应用程序的 URL 并浏览到 **CreatePerson.xhtml** 数据输入页面。使用表单输入用户名和年龄。点 **Save**。
请注意：现在您可以通过点 **View Persons Record List** 链接来查看数据库中的数据。

您还可以使用CLI 工具（如 **psql** 等）对数据库进行操作。

2.3.6. 在 odo 中使用 devfile

2.3.6.1. 关于 odo 中的 devfile

devfile 是一个可移植的文件，它描述了您的开发环境。使用 devfile，您可以定义一个可移植的开发环境而无需重新配置。

使用 devfile，您可以描述开发环境，如源代码、IDE 工具、应用程序运行时和预定义的命令。要了解更多关于 devfile 的信息，请参阅 devfile 文档。

使用 **odo**，您可以从 devfiles 创建组件。当使用 devfile 创建组件时，**odo** 会将 devfile 转换为一个由 OpenShift Container Platform、Kubernetes 或 Docker 上运行的多个容器组成的工作区。**odo** 自动使用默认的 devfile registry，但用户可以添加自己的 registry。

2.3.6.2. 使用 devfile 创建 Java 应用程序

先决条件

- 已安装了 **odo**。
- 必须知道您的 ingress 域集群名称。如果不知道，请联络您的集群管理员。例如, **apps-crc.testing** 是 Red Hat CodeReady Containers 的集群域名。

2.3.6.2.1. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓ Project 'myproject' is ready for use
✓ New project created and now using project : myproject
```

2.3.6.2.2. 列出可用的 devfile 组件

使用 **odo**，可以显示集群中的所有可用组件。可用的组件取决于集群的配置。

流程

1. 要列出集群中可用的 devfile 组件，请运行：

```
$ odo catalog list components
```

输出列出了可用的 **odo** 组件：

```
Odo Devfile Components:
NAME               DESCRIPTION                            REGISTRY
java-maven         Upstream Maven and OpenJDK 11          DefaultDevfileRegistry
java-openliberty   Open Liberty microservice in Java      DefaultDevfileRegistry
java-quarkus       Upstream Quarkus with Java+GraalVM     DefaultDevfileRegistry
java-springboot    Spring Boot® using Java                DefaultDevfileRegistry
nodejs             Stack with NodeJS 12                   DefaultDevfileRegistry

Odo OpenShift Components:
NAME       PROJECT      TAGS                                                    SUPPORTED
java       openshift    11,8,latest                                             YES
dotnet     openshift    2.1,3.1,latest                                          NO
golang     openshift    1.13.4-ubi7,1.13.4-ubi8,latest                          NO
httpd      openshift    2.4-el7,2.4-el8,latest                                  NO
nginx      openshift    1.14-el7,1.14-el8,1.16-el7,1.16-el8,latest              NO
nodejs     openshift    10-ubi7,10-ubi8,12-ubi7,12-ubi8,latest                  NO
perl       openshift    5.26-el7,5.26-ubi8,5.30-el7,latest                      NO
```

```
php        openshift     7.2-ubi7,7.2-ubi8,7.3-ubi7,7.3-ubi8,latest                          NO
python     openshift     2.7-ubi7,2.7-ubi8,3.6-ubi7,3.6-ubi8,3.8-ubi7,3.8-ubi8,latest
NO
ruby       openshift     2.5-ubi7,2.5-ubi8,2.6-ubi7,2.6-ubi8,2.7-ubi7,latest                   NO
wildfly    openshift
10.0,10.1,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0,8.1,9.0,latest    NO
```

2.3.6.2.3. 使用 devfile 部署 Java 应用程序

在这一部分，您将了解如何使用 devfile 部署使用 Maven 和 Java 8 JDK 的 Java 示例项目。

流程

1. 创建一个目录来存储组件的源代码：

```
$ mkdir <directory-name>
```

2. 创建名为 **myspring** 的 Spring Boot 组件类型的组件配置并下载其示例项目：

```
$ odo create java-spring-boot myspring --starter
```

上面的命令会产生以下输出：

```
Validation
 ✓  Checking devfile compatibility [195728ns]
 ✓  Creating a devfile component from registry: DefaultDevfileRegistry [170275ns]
 ✓  Validating devfile component [281940ns]

Please use `odo push` command to create the component with source deployed
```

odo create 命令从记录的 devfile registry 中下载相关的 **devfile.yaml** 文件。

3. 列出目录的内容以确认下载了 devfile 和示例 Java 应用程序：

```
$ ls
```

上面的命令会产生以下输出：

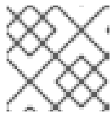
```
README.md    devfile.yaml    pom.xml       src
```

4. 创建一个 URL 以访问部署的组件：

```
$ odo url create --host apps-crc.testing
```

上面的命令会产生以下输出：

```
 ✓  URL myspring-8080.apps-crc.testing created for component: myspring

To apply the URL configuration changes, please use odo push
```



注意

创建 URL 时必须使用集群主机名。

5. 将组件推送到集群：

```
$ odo push
```

上面的命令会产生以下输出：

```
Validation
 ✓  Validating the devfile [81808ns]

Creating Kubernetes resources for component myspring
 ✓  Waiting for component to start [5s]

Applying URL changes
 ✓  URL myspring-8080: http://myspring-8080.apps-crc.testing created

Syncing to component myspring
 ✓  Checking files for pushing [2ms]
 ✓  Syncing files to the component [1s]

Executing devfile commands for component myspring
 ✓  Executing devbuild command "/artifacts/bin/build-container-full.sh" [1m]
 ✓  Executing devrun command "/artifacts/bin/start-server.sh" [2s]

Pushing devfile component myspring
 ✓  Changes successfully pushed to component
```

6. 列出组件的 URL 以验证组件已被成功推送：

```
$ odo url list
```

上面的命令会产生以下输出：

```
Found the following URLs for component myspring
NAME             URL                                    PORT     SECURE
myspring-8080    http://myspring-8080.apps-crc.testing    8080     false
```

7. 使用生成的 URL 查看部署的应用程序。

```
$ curl http://myspring-8080.apps-crc.testing
```

2.3.6.3. 将 S2I 组件转换为 devfile 组件

使用 **odo**，您可以同时创建 Source-to-Image（S2I）和 devfile 组件。如果您有一个现有的 S2I 组件，可以使用 **odo utils** 命令将其转换为 devfile 组件。

流程

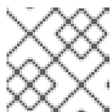
从 S2I 组件目录中运行所有命令。

1. 运行 **odo utils convert-to-devfile** 命令，它会基于您的组件创建 **devfile.yaml** 和 **env.yaml**：

```
$ odo utils convert-to-devfile
```

2. 将组件推送到集群：

```
$ odo push
```



注意

如果 devfile 组件部署失败，请运行以下命令删除它： **odo delete -a**

3. 验证 devfile 组件已成功部署：

```
$ odo list
```

4. 删除 S2I 组件：

```
$ odo delete --s2i
```

2.3.7. 使用存储

持久性存储会在 **odo** 重启时保留数据。

2.3.7.1. 在应用程序组件中添加存储

使用 **odo storage** 命令为应用程序添加持久性数据。必须持久化的数据示例包括数据库文件、依赖项和构建工件，如 **.m2** Maven 目录。

流程

1. 在组件中添加存储：

```
$ odo storage create <storage_name> --path=<path_to_the_directory> --size=<size>
```

2. 将存储推送到集群：

```
$ odo push
```

3. 通过列出组件中的所有存储来验证存储现在附加到您的组件中：

```
$ odo storage list
```

```
The component 'nodejs' has the following storage attached:
NAME        SIZE    PATH     STATE
mystorage    1Gi    /data    Pushed
```

4. 从您的组件中删除存储：

```
$ odo storage delete <storage_name>
```

5. 列出所有存储以验证存储状态是否**本地删除**:

```
$ odo storage list
```

```
The component 'nodejs' has the following storage attached:
NAME        SIZE    PATH    STATE
mystorage   1Gi     /data   Locally Deleted
```

6. 将更改推送到集群：

```
$ odo push
```

2.3.7.2. 在临时存储和持久性存储间切换

您可以使用 **odo preference** 命令在项目中的临时存储和持久性存储间切换。**odo preference** 可以修改集群中的全局首选项。

启用持久性存储后，集群会在重启之间保存信息。

启用临时存储后，集群不会在重启之间保存信息。

默认启用临时存储。

流程

1. 请参阅项目中当前设置的首选项：

```
$ odo preference view
```

```
PARAMETER          CURRENT_VALUE
UpdateNotification
NamePrefix
Timeout
BuildTimeout
PushTimeout
Experimental
PushTarget
Ephemeral          true
```

2. 要取消设置临时存储并设置持久性存储：

```
$ odo preference set Ephemeral false
```

3. 再次设置临时存储：

```
$ odo preference set Ephemeral true
```

odo preference 命令更改您目前部署的所有组件的全局设置，以及您要将来部署的组件。

4. 运行 **odo push** 使 **odo** 为您的组件创建指定的存储：

```
$ odo push
```

其他资源

- 了解临时存储。
- 了解持久性存储

2.3.8. 取消应用程序

您可以删除项目中应用程序和与应用程序关联的所有组件。

2.3.8.1. 删除应用程序

使用 **odo app delete** 命令删除应用程序。

流程

1. 列出当前项目中的应用程序：

```
$ odo app list
```

输出示例

```
The project '<project_name>' has the following applications:
NAME
app
```

2. 列出与应用程序关联的组件。这些组件将随应用程序一起删除：

```
$ odo component list
```

输出示例

```
APP    NAME                   TYPE      SOURCE     STATE
app    nodejs-nodejs-ex-elyf  nodejs    file://./  Pushed
```

3. 删除应用程序：

```
$ odo app delete <application_name>
```

输出示例

```
? Are you sure you want to delete the application: <application_name> from project:
<project_name>
```

4. 使用 **Y** 确认删除。您可以使用 **-f** 标记来阻止确认提示。

2.3.9. 在 odo 中调试应用程序

使用 **odo**，您可以附加一个 debugger 来远程调试应用程序。这个功能只支持 NodeJS 和 Java 组件。

默认情况下，使用 **odo** 创建的组件以 debug 模式运行。debugger 代理在组件上运行，并使用特定端口。要开始调试您的应用程序，必须启动端口转发功能，并在集成开发环境中 (IDE) 附加本地 debugger。

2.3.9.1. 调试应用程序

您可以使用 **odo debug** 命令在 **odo** 中调试应用程序。

流程

1. 下载包含 devfile 中必要的 **debugrun** 步骤的示例应用程序：

```
$ odo create nodejs --starter
```

输出示例

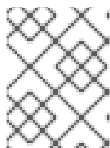
```
Validation
 ✓  Checking devfile existence [11498ns]
 ✓  Checking devfile compatibility [15714ns]
 ✓  Creating a devfile component from registry: DefaultDevfileRegistry [17565ns]
 ✓  Validating devfile component [113876ns]

Starter Project
 ✓  Downloading starter project nodejs-starter from https://github.com/odo-devfiles/nodejs-ex.git [428ms]

Please use `odo push` command to create the component with source deployed
```

2. 使用 **--debug** 标志推送应用程序，在所有调试部署中都需要它：

```
$ odo push --debug
```

输出示例

```
Validation
 ✓  Validating the devfile [29916ns]

Creating Kubernetes resources for component nodejs
 ✓  Waiting for component to start [38ms]

Applying URL changes
 ✓  URLs are synced with the cluster, no changes are required.

Syncing to component nodejs
 ✓  Checking file changes for pushing [1ms]
 ✓  Syncing files to the component [778ms]

Executing devfile commands for component nodejs
 ✓  Executing install command "npm install" [2s]
 ✓  Executing debug command "npm run debug" [1s]

Pushing devfile component nodejs
 ✓  Changes successfully pushed to component
```



注意

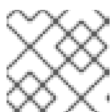
您可以使用 **--debug-command="custom-step"** 标志来指定一个自定义调试命令。

3. 端口转发到本地端口来访问调试接口：

```
$ odo debug port-forward
```

输出示例

```
Started port forwarding at ports - 5858:5858
```



注意

您可以使用 **--local-port** 标志指定端口。

4. 检查 debug 会话是否在一个单独的终端窗口中运行：

```
$ odo debug info
```

输出示例

```
Debug is running for the component on the local port : 5858
```

5. 附加在您选择的 IDE 中捆绑的 debugger。具体步骤根据 IDE 的不同而有所不同，例如：VSCode 调试接口。

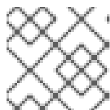
2.3.9.2. 配置调试参数

您可以使用 **odo config** 命令指定远程端口，并使用 **odo debug** 命令指定本地端口 。

流程

- 要设置调试代理应运行的远程端口，请运行：

```
$ odo config set DebugPort 9292
```

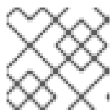


注意

您必须重新部署您的组件，才能在组件中反映此值。

- 要在本地端口中启用转发端口功能，请运行：

```
$ odo debug port-forward --local-port 9292
```



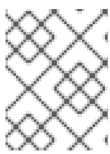
注意

本地端口值不具有持久性。您必须在每次需要更改端口时都提供该端口值。

2.3.10. 示例应用程序

odo 提供与 OpenShift 组件类型目录中列出的语言或运行时的部分兼容性。例如：

```
NAME      PROJECT     TAGS
dotnet    openshift   2.0,latest
httpd     openshift   2.4,latest
java      openshift   8,latest
nginx     openshift   1.10,1.12,1.8,latest
nodejs    openshift   0.10,4,6,8,latest
perl      openshift   5.16,5.20,5.24,latest
php       openshift   5.5,5.6,7.0,7.1,latest
python    openshift   2.7,3.3,3.4,3.5,3.6,latest
ruby      openshift   2.0,2.2,2.3,2.4,latest
wildfly   openshift   10.0,10.1,8.1,9.0,latest
```



注意

对于 **odo**，Java 和 Node.js 是官方支持的组件类型。运行 **odo catalog list components** 来验证官方支持的组件类型。

要通过 web 访问该组件，请使用 **odo url create** 创建一个 URL。

2.3.10.1. Git 软件仓库示例

2.3.10.1.1. httpd

这个示例在 CentOS 7 上使用 httpd 构建并提供静态内容。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请查看 Apache HTTP 服务器容器镜像存储库。

```
$ odo create httpd --git https://github.com/openshift/httpd-ex.git
```

2.3.10.1.2. java

这个示例在 CentOS 7 上构建并运行 fat JAR Java 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请参见 Java S2I Builder image 。

```
$ odo create java --git https://github.com/spring-projects/spring-petclinic.git
```

2.3.10.1.3. nodejs

在 CentOS 7 中构建并运行 Node.js 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请参见 Node.js 8 container image。

```
$ odo create nodejs --git https://github.com/openshift/nodejs-ex.git
```

2.3.10.1.4. Perl

这个示例在 CentOS 7 中构建并运行 Perl 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请查看 Perl 5.26 容器镜像。

```
$ odo create perl --git https://github.com/openshift/dancer-ex.git
```

2.3.10.1.5. php

这个示例在 CentOS 7 中构建并运行 PHP 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请参见 PHP 7.1 Docker image。

```
$ odo create php --git https://github.com/openshift/cakephp-ex.git
```

2.3.10.1.6. Python

这个示例在 CentOS 7 中构建并运行 Python 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请查看 Python 3.6 容器镜像。

```
$ odo create python --git https://github.com/openshift/django-ex.git
```

2.3.10.1.7. Ruby

这个示例在 CentOS 7 中构建并运行 Ruby 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请参见 Ruby 2.5 container image。

```
$ odo create ruby --git https://github.com/openshift/ruby-ex.git
```

2.3.10.1.8. WildFly

这个示例在 CentOS 7 中构建并运行 WildFly 应用程序。有关使用这个构建器镜像的更多信息，包括 OpenShift Container Platform 的注意事项，请 查看 Wilfly - OpenShift 的 CentOS Docker 镜像 。

```
$ odo create wildfly --git https://github.com/openshift/openshift-jee-sample.git
```

2.3.10.2. 二进制示例

2.3.10.2.1. java

Java 可以用来部署二进制工件，如下：

```
$ git clone https://github.com/spring-projects/spring-petclinic.git
$ cd spring-petclinic
$ mvn package
$ odo create java test3 --binary target/*.jar
$ odo push
```

2.3.10.2.2. WildFly

WildFly 可以用来部署二进制应用程序，如下：

```
$ git clone https://github.com/openshiftdemos/os-sample-java-web.git
$ cd os-sample-java-web
$ mvn package
$ cd ..
```

```
$ mkdir example && cd example
$ mv ../os-sample-java-web/target/ROOT.war example.war
$ odo create wildfly --binary example.war
```

2.4. 在受限环境中使用 ODO

2.4.1. 受限环境中的 odo

要在断开连接的集群或受限环境中置备的集群中运行 **odo**， 集群管理员需要创建带有registry 镜像（mirror）的集群。

要在断开连接的集群中工作，必须首先将 将 **odo init** 镜像推送到集群的 registry 中， 然后使用 **ODO_BOOTSTRAPPER_IMAGE** 环境变量覆盖 **odo init** 镜像路径。

在推送 **odo init**镜像后，您需要为 registry 中的支持的构建程序镜像（builder image）在本地创建一个镜像（mirror）， 覆盖 registry 镜像（mirror）， 然后创建您的应用程序。构建程序镜像是为应用程序配置运行时环境所必需的，它还包含构建应用程序所需的构建工具，例如：用于 Node.js 的 npm 或用于 Java 的 Maven。一个 registry 镜像（mirror）会包含应用程序所需的所有依赖项。

其它资源

- 创建用于在受限网络中安装的镜像 registry
- 访问 registry

2.4.2. 将 odo init 镜像推送到受限集群的 registry 中

根据集群和操作系统的配置，您可以将 **odo** init 镜像推送到一个 registry 镜像（mirror）中，或直接推送到内部 registry。

2.4.2.1. 先决条件

- 在客户端操作系统上安装 **oc** 。
- 在客户端操作系统中安装 **odo**。
- 访问已配置的内部 registry 或 registry 镜像（mirror）的受限集群。

2.4.2.2. 将 odox init 镜像推送到镜像的容器镜像仓库

根据您的操作系统，您可以将 **odo** init 镜像推送到带有 registry 镜像（mirror）的集群中，如下所示：

2.4.2.2.1. 将 init 镜像推送到 Linux 上的镜像 registry

流程

1. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

   ```
   $ echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
   ```

2. 将编码的 root CA 证书复制到适当的位置：

   ```
   $ sudo cp ./disconnect-ca.crt /etc/pki/ca-trust/source/anchors/<mirror-registry>.crt
   ```

3. 信任客户端平台中的 CA，并登录 OpenShift Container Platform 镜像 registry：

```
$ sudo update-ca-trust enable && sudo systemctl daemon-reload && sudo systemctl restart /
docker && docker login <mirror-registry>:5000 -u <username> -p <password>
```

4. 对 **odox** init 镜像进行镜像（mirror）：

```
$ oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
```

5. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

```
$ export ODO_BOOTSTRAPPER_IMAGE=<mirror-registry>:5000/openshiftdo/odo-init-
image-rhel7:<tag>
```

2.4.2.2.2. 将 init 镜像推送到 MacOS 上的镜像 registry

流程

1. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

```
$ echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
```

2. 将编码的 root CA 证书复制到适当的位置：
   a. 使用 Docker UI 重启 Docker。
   b. 运行以下命令：

   ```
   $ docker login <mirror-registry>:5000 -u <username> -p <password>
   ```

3. 对 **odox** init 镜像进行镜像（mirror）：

```
$ oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
```

4. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

```
$ export ODO_BOOTSTRAPPER_IMAGE=<mirror-registry>:5000/openshiftdo/odo-init-
image-rhel7:<tag>
```

2.4.2.2.3. 将 init 镜像推送到 Windows 上的镜像 registry

流程

1. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

```
PS C:\> echo <content_of_additional_ca> | base64 -d > disconnect-ca.crt
```

2. 作为管理员，请执行以下命令将编码的 root CA 证书复制到适当的位置：

```
PS C:\WINDOWS\system32> certutil -addstore -f "ROOT" disconnect-ca.crt
```

3. 信任客户端平台中的 CA，并登录 OpenShift Container Platform 镜像 registry：
   a. 使用 Docker UI 重启 Docker。
   b. 运行以下命令：

   ```
   PS C:\WINDOWS\system32> docker login <mirror-registry>:5000 -u <username> -p
   <password>
   ```

4. 对 **odox** init 镜像进行镜像（mirror）：

   ```
   PS C:\> oc image mirror registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
   <mirror-registry>:5000/openshiftdo/odo-init-image-rhel7:<tag>
   ```

5. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

   ```
   PS C:\> $env:ODO_BOOTSTRAPPER_IMAGE="<mirror-registry>:5000/openshiftdo/odo-
   init-image-rhel7:<tag>"
   ```

2.4.2.3. 将 odo init 镜像直接推送到内部 registry

如果集群允许镜像直接推送到内部 registry，请将 **odo** init 镜像推送到 registry：

2.4.2.3.1. 在 Linux 中直接推送init 镜像

流程

1. 启用默认路由：

   ```
   $ oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
   {"defaultRoute":true}}' --type='merge' -n openshift-image-registry
   ```

2. 获取通配符路由 CA：

   ```
   $ oc get secret router-certs-default -n openshift-ingress -o yaml
   ```

   输出示例

   ```
   apiVersion: v1
   data:
     tls.crt: **************************
     tls.key: ##################
   kind: Secret
   metadata:
     [...]
   type: kubernetes.io/tls
   ```

3. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

   ```
   $ echo <tls.crt> | base64 -d > ca.crt
   ```

4. 在客户端平台中信任 CA：

```
$ sudo cp ca.crt  /etc/pki/ca-trust/source/anchors/externalroute.crt && sudo update-ca-trust
enable && sudo systemctl daemon-reload && sudo systemctl restart docker
```

5. 登录到内部 registry：

```
$ oc get route -n openshift-image-registry
NAME      HOST/PORT    PATH   SERVICES    PORT  TERMINATION   WILDCARD
default-route   <registry_path>         image-registry   <all>   reencrypt     None

$ docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. 推送 **odo** init 镜像：

```
$ docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

$ docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

$ docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

```
$ export ODO_BOOTSTRAPPER_IMAGE=<registry_path>/openshiftdo/odo-init-image-
rhel7:1.0.1
```

2.4.2.3.2. 在 MacOS 上直接推送 init 镜像

流程

1. 启用默认路由：

```
$ oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
{"defaultRoute":true}}' --type='merge' -n openshift-image-registry
```

2. 获取通配符路由 CA：

```
$ oc get secret router-certs-default -n openshift-ingress -o yaml
```

输出示例

```
apiVersion: v1
data:
  tls.crt: **************************
  tls.key: ##################
kind: Secret
metadata:
  [...]
type: kubernetes.io/tls
```

3. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

```
$ echo <tls.crt> | base64 -d > ca.crt
```

4. 在客户端平台中信任 CA：

```
$ sudo security add-trusted-cert -d -r trustRoot -k /Library/Keychains/System.keychain ca.crt
```

5. 登录到内部 registry：

```
$ oc get route -n openshift-image-registry
NAME        HOST/PORT          PATH   SERVICES          PORT   TERMINATION   WILDCARD
default-route   <registry_path>          image-registry   <all>   reencrypt     None

$ docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. 推送 **odo** init 镜像：

```
$ docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

$ docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

$ docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

```
$ export ODO_BOOTSTRAPPER_IMAGE=<registry_path>/openshiftdo/odo-init-image-rhel7:1.0.1
```

2.4.2.3.3. 在 Windows 上直接推送 init 镜像

流程

1. 启用默认路由：

```
PS C:\> oc patch configs.imageregistry.operator.openshift.io cluster -p '{"spec":
{"defaultRoute":true}}' --type='merge' -n openshift-image-registry
```

2. 获取通配符路由 CA：

```
PS C:\> oc get secret router-certs-default -n openshift-ingress -o yaml
```

输出示例

```
apiVersion: v1
data:
  tls.crt: **************************
  tls.key: ##################
kind: Secret
metadata:
  [...]
type: kubernetes.io/tls
```

3. 使用 **base64** 对您的镜像 registry 的 root 认证授权 (CA) 内容进行编码：

```
PS C:\> echo <tls.crt> | base64 -d > ca.crt
```

4. 作为管理员，请执行以下命令在客户端平台中信任 CA：

```
PS C:\WINDOWS\system32> certutil -addstore -f "ROOT" ca.crt
```

5. 登录到内部 registry：

```
PS C:\> oc get route -n openshift-image-registry
NAME        HOST/PORT    PATH   SERVICES     PORT  TERMINATION   WILDCARD
default-route   <registry_path>         image-registry   <all>   reencrypt     None

PS C:\> docker login <registry_path> -u kubeadmin -p $(oc whoami -t)
```

6. 推送 **odo** init 镜像：

```
PS C:\> docker pull registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>

PS C:\> docker tag registry.access.redhat.com/openshiftdo/odo-init-image-rhel7:<tag>
<registry_path>/openshiftdo/odo-init-image-rhel7:<tag>

PS C:\> docker push <registry_path>/openshiftdo/odo-init-image-rhel7:<tag>
```

7. 设置 **ODO_BOOTSTRAPPER_IMAGE** 环境变量来覆盖默认的 **odo** init 镜像路径：

```
PS C:\> $env:ODO_BOOTSTRAPPER_IMAGE="<registry_path>/openshiftdo/odo-init-
image-rhel7:<tag>"
```

2.4.3. 在断开连接的集群中创建和部署组件

将 **init** 镜像推送到具有镜像（mirror） registry 的集群中后，您必须使用 **oc** 工具为应用程序所需的构建程序镜像创建一个镜像（mirror），使用环境变量覆盖镜像 registry，然后创建组件。

2.4.3.1. 先决条件

- 在客户端操作系统上安装 **oc** 。
- 在客户端操作系统中安装 **odo**。
- 访问已配置的内部容器镜像仓库或 registry 镜像（mirror）的受限集群。
- 将 **odo** init 镜像推送到集群 registry。

2.4.3.2. 为支持的构建器镜像创建一个镜像（mirror）

要使用 Node.js 依赖项的 npm 软件包及 Java 依赖项的 Maven 软件包，并为应用程序配置运行时环境，您必须从镜像 registry 中镜像相应的构建器镜像。

流程

1. 验证所需镜像标签没有导入：

```
$ oc describe is nodejs -n openshift
```

输出示例

```
Name:                   nodejs
Namespace:              openshift
[...]

10
  tagged from <mirror-registry>:<port>/rhoar-nodejs/nodejs-10
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs, hidden
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io "<mirror-registry>:
<port>/rhoar-nodejs/nodejs-10:latest" not found
      About an hour ago

10-SCL (latest)
  tagged from <mirror-registry>:<port>/rhscl/nodejs-10-rhel7
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  ! error: Import failed (NotFound): dockerimage.image.openshift.io "<mirror-registry>:
<port>/rhscl/nodejs-10-rhel7:latest" not found
      About an hour ago

[...]
```

2. 将支持的镜像标签镜像到私有 registry：

```
$ oc image mirror registry.access.redhat.com/rhscl/nodejs-10-rhel7:<tag>
<private_registry>/rhscl/nodejs-10-rhel7:<tag>
```

3. 导入镜像：

```
$ oc tag <mirror-registry>:<port>/rhscl/nodejs-10-rhel7:<tag> nodejs-10-rhel7:latest --
scheduled
```

您必须定期重新导入镜像。**--scheduled** 标志启用镜像自动重新导入。

4. 验证带有指定标签的镜像已被导入：

```
$ oc describe is nodejs -n openshift
```

输出示例

```
Name:                nodejs
[...]
10-SCL (latest)
  tagged from <mirror-registry>:<port>/rhscl/nodejs-10-rhel7
    prefer registry pullthrough when referencing this tag

  Build and run Node.js 10 applications on RHEL 7. For more information about using this
builder image, including OpenShift considerations, see https://github.com/nodeshift/centos7-
s2i-nodejs.
  Tags: builder, nodejs
  Example Repo: https://github.com/sclorg/nodejs-ex.git

  * <mirror-registry>:<port>/rhscl/nodejs-10-
rhel7@sha256:d669ecbc11ac88293de50219dae8619832c6a0f5b04883b480e073590fab7c54

      3 minutes ago

[...]
```

2.4.3.3. mirror registry 介绍

要从私有 mirror registry 中为 Node.js 依赖项下载 npm 软件包，及为 Java 依赖项下载 Maven 软件包，您必须在集群中创建并配置 mirror npm 或 Maven registry。然后您可以覆盖现有组件上的 mirror registry，或覆盖创建新组件时的 mirror registry。

流程

- 覆盖现有组件上的 mirror registry：

```
$ odo config set --env NPM_MIRROR=<npm_mirror_registry>
```

- 覆盖创建组件时的 mirror registry：

```
$ odo component create nodejs --env NPM_MIRROR=<npm_mirror_registry>
```

2.4.3.4. 使用 odo 创建 Node.js 应用程序

要创建一个 Node.js 组件，请下载 Node.js 应用程序并使用 **odo**将源代码推送到您的集群中。

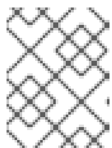
流程

1. 将当前目录变为您的应用程序的目录：

```
$ cd <directory_name>
```

2. 将类型 Node.js 的组件添加到应用程序中：

```
$ odo create nodejs
```



注意

默认情况下使用最新的镜像。也可以使用 **odo create openshift/nodejs:8** 明确指定一个镜像版本。

3. 将初始源代码推送到组件中：

```
$ odo push
```

现在，您的组件已被部署到 OpenShift Container Platform 中。

4. 创建一个 URL，按以下方法在本地配置文件中添加条目：

```
$ odo url create --port 8080
```

5. 推送更改。这会在集群中创建一个 URL。

```
$ odo push
```

6. 列出用于检查组件所需 URL 的 URL。

```
$ odo url list
```

7. 使用生成的 URL 查看部署的应用程序。

```
$ curl <url>
```

2.5. 创建 OPERATOR 管理的服务实例

Operators 是打包、部署和管理 Kubernetes 应用程序的一个方法。通过 **odo**，您可以从 Operator 提供的自定义资源定义（CRD）创建服务实例。然后，您可以在项目中使用这些实例，并将其链接到您的组件。

要从 Operator 创建服务，您必须确保 Operator 在它的 **metadata** 中定义了有效值才能启动请求的服务。**odo** 使用 Operator 的 **metadata.annotations.alm-examples** YAML 文件启动该服务。如果此 YAML 具有占位符值或示例值，服务将无法启动。您可以修改 YAML 文件，使用修改后的值启动服务。要了解如何修改 YAML 文件并启动服务，请参阅通过 YAML 文件创建服务。

2.5.1. 先决条件

- 安装 **oc** CLI 并登录到集群。
    - 请注意，集群的配置决定了可用的服务。要访问 Operator 服务，集群管理员必须首先在集群上安装相应的 Operator。如需了解更多相关信息，请参阅 在集群中添加 Operator。
- 安装 **odo** CLI。

2.5.2. 创建一个项目

创建一个项目来在一个独立的空间内保存源代码、测试并对库进行管理。

流程

1. 登陆到一个OpenShift Container Platform集群。

```
$ odo login -u developer -p developer
```

2. 创建一个项目：

```
$ odo project create myproject
```

输出示例

```
✓ Project 'myproject' is ready for use
✓ New project created and now using project : myproject
```

2.5.3. 列出来自于集群中安装的 Operator 的可用服务

使用 **odo**，可以显示集群中安装的 Operator 列表及其提供的服务。

- 要列出当前项目中安装的 Operator，请运行：

```
$ odo catalog list services
```

命令列出 Operators 和 CRD。该命令的输出显示集群中安装的 Operator。例如：

```
Operators available in the cluster
NAME                          CRDs
etcdoperator.v0.9.4           EtcdCluster, EtcdBackup, EtcdRestore
mongodb-enterprise.v1.4.5     MongoDB, MongoDBUser, MongoDBOpsManager
```

etcdoperator.v0.9.4 是 Operator，**EtcdCluster**、**EtcdBackup** 和 **EtcdRestore** 是 Operator 提供的 CRD。

2.5.4. 从 Operator 创建服务

如果 Operator 在 **metadata** 中定义了有效值来启动请求的服务，您可以通过 **odo service create** 来使用服务。

1. 将服务的 YAML 作为本地驱动器上的文件输出：

```
$ oc get csv/etcdoperator.v0.9.4 -o yaml
```

2. 验证服务的值是否有效：

```
apiVersion: etcd.database.coreos.com/v1beta2
kind: EtcdCluster
metadata:
  name: example
spec:
  size: 3
  version: 3.2.13
```

3. 从 **etcdoperator.v0.9.4** Operator 启动 **EtcdCluster** 服务：

```
$ odo service create etcdoperator.v0.9.4 EtcdCluster
```

4. 验证服务是否已启动：

```
$ oc get EtcdCluster
```

2.5.5. 从 YAML 文件创建服务

如果服务或自定义资源（CR）的 YAML 定义无效或有占位符数据，您可以使用 **--dry-run** 标志获取 YAML 定义。指定正确的值后使用更正的 YAML 定义启动服务。通过打印和修改用于启动服务的 YAML，**odo** 可以在实际启动一个服务前输出由 Operator 提供的服务或 CR 的 YAML 定义。

1. 要显示服务的 YAML，请运行：

```
$ odo service create <operator-name> --dry-run
```

例如，要打印由 **etcdoperator.v0.9.4** Operator 提供的 **EtcdCluster** 的 YAML 定义，请运行：

```
$ odo service create etcdoperator.v0.9.4 --dry-run
```

YAML 保存为 **etcd.yaml** 文件。

2. 修改 **etcd.yaml** 文件：

```
apiVersion: etcd.database.coreos.com/v1beta2
kind: EtcdCluster
metadata:
  name: my-etcd-cluster (1)
spec:
  size: 1 (2)
  version: 3.2.13
```

(1) 将名称从 **example** 改为 **my-etcd-cluster**

(2) 将大小从 **3** 减小到 **1**

3. 从 YAML 文件启动服务：

```
$ odo service create --from-file etcd.yaml
```

4. 验证 **EtcdCluster** 服务已使用一个 pod 启动，而不是预先配置的三个 pod:

```
$ oc get pods | grep my-etcd-cluster
```

2.6. 管理环境变量

odo 在 **config** 文件中存储特定于组件的配置和环境变量。您可以使用 **odo config** 命令为组件设置、取消设置和列出环境变量，而无需修改 **config** 文件。

2.6.1. 设置和取消设置环境变量

流程

- 在组件中设置环境变量：

```
$ odo config set --env <variable>=<value>
```

- 在组件中取消设置环境变量：

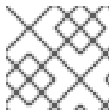
```
$ odo config unset --env <variable>
```

- 列出组件中的所有环境变量：

```
$ odo config view
```

2.7. 配置 ODO CLI

2.7.1. 使用命令完成



注意

目前只有 bash 、zsh 和 shell 支持命令完成。

odo 提供基于用户输入的智能命令参数完成功能。要做到这一点，odo 需要与执行 shell 集成。

流程

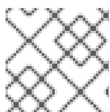
- 要自动安装命令完成：
  1. 运行：

     ```
     $ odo --complete
     ```

  2. 提示安装完成 hook 时按 **y** 键。
- 要手动安装完成 hook，请在 shell 配置文件中添加 **complete -o nospace -C <full_path_to_your_odo_binary> odo**。在修改了 shell 配置文件后，重启 shell。
- 禁用完成：
  1. 运行：

     ```
     $ odo --uncomplete
     ```

  2. 提示后按 **y** 卸载完成 hook。



注意

如果您重命名 odo 可执行文件或将其移动到其他目录中，请重新启用命令完成。

2.7.2. 忽略文件或特征

您可以通过修改应用程序根目录中的 **.odoignore** 文件来配置要忽略的文件或模式列表。这适用于 **odo push** 和 **odo watch**。

如果 **.odoignore 文件** 不存在，则会使用 **.gitignore** 文件来忽略特定的文件和文件夹。

要忽略 **.git** 文件、任意带有 **.js** 扩展名的文件，以及 **tests** 目录，在 **.odoignore** 或 **.gitignore** 文件中添加以下内容：

```
.git
*.js
tests/
```

.odoignore 文件 允许任何 glob 表达式。

2.8. ODO CLI 参考指南

2.8.1. 基本 odo CLI 命令

2.8.1.1. app

执行与 OpenShift Container Platform 项目相关的应用程序操作。

使用 app 的示例

```
# Delete the application
odo app delete myapp

# Describe 'webapp' application,
odo app describe webapp

# List all applications in the current project
odo app list

# List all applications in the specified project
odo app list --project myproject
```

2.8.1.2. catalog

执行与目录相关的操作。

使用 catalog 的示例

```
# Get the supported components
odo catalog list components

# Get the supported services from service catalog
odo catalog list services

# Search for a component
odo catalog search component python

# Search for a service
odo catalog search service mysql

# Describe a service
odo catalog describe service mysql-persistent
```

2.8.1.3. component

管理应用程序的组件。

使用 component 的示例

```
# Create a new component
odo component create

# Create a local configuration and create all objects on the cluster
odo component create --now
```

2.8.1.4. config

在 **config** 文件中修改与 **odo** 相关的设置。

使用 config 的示例

```
# For viewing the current local configuration
odo config view

# Set a configuration value in the local configuration
odo config set Type java
odo config set Name test
odo config set MinMemory 50M
odo config set MaxMemory 500M
odo config set Memory 250M
odo config set Ignore false
odo config set MinCPU 0.5
odo config set MaxCPU 2
odo config set CPU 1

# Set an environment variable in the local configuration
odo config set --env KAFKA_HOST=kafka --env KAFKA_PORT=6639

# Create a local configuration and apply the changes to the cluster immediately
odo config set --now

# Unset a configuration value in the local config
odo config unset Type
odo config unset Name
odo config unset MinMemory
odo config unset MaxMemory
odo config unset Memory
odo config unset Ignore
odo config unset MinCPU
odo config unset MaxCPU
odo config unset CPU

# Unset an env variable in the local config
odo config unset --env KAFKA_HOST --env KAFKA_PORT
```

Application（应用程序）	Application 是应用程序名称，组件需要作为此应用程序的一部分
CPU	组件可消耗的最少和最多 CPU
Ignore	进行 push 和 watch 时使用 .odoignore 文件

表 2.2. 可用的本地参数：

Application（应用程序）	组件需要成为其一部分的应用程序名称
CPU	组件可消耗的最少和最多 CPU
Ignore	进行 push 和 watch 时是否使用 **.odoignore** 文件
MaxCPU	组件可消耗的最多 CPU
MaxMemory	组件可消耗的最大内存
Memory	组件可消耗的最小和最大内存
MinCPU	组件可消耗的最小 CPU
MinMemory	组件提供的最小内存
名称	组件的名称
Ports	要在组件中打开的端口
Project（项目）	组件所属项目的名称
Ref	用于从 git 源创建组件的 git ref
SourceLocation	该路径表示二进制文件或者 git 源的位置
SourceType	组件源的类型 - git/binary/local
存储	组件的存储
类型	组件的类型
URL	访问该组件的 URL

2.8.1.5. create

创建描述在 OpenShift Container Platform 中部署的组件的配置。如果没有提供组件名称，它会被自动生成。

默认情况下，构建者镜像是从当前命名空间中使用的。如果要明确指定一个命名空间，使用：**odo create namespace/name:version**。如果没有指定版本，则默认为 **latest**。

使用 **odo catalog list** 查看可以部署的完整组件类型列表。

使用 create 的示例

```
 # Create new Node.js component with the source in current directory.
 odo create nodejs

 # Create new Node.js component and push it to the cluster immediately.
 odo create nodejs --now

 # A specific image version may also be specified
 odo create nodejs:latest

 # Create new Node.js component named 'frontend' with the source in './frontend' directory
 odo create nodejs frontend --context ./frontend

 # Create a new Node.js component of version 6 from the 'openshift' namespace
 odo create openshift/nodejs:6 --context /nodejs-ex

 # Create new Wildfly component with binary named sample.war in './downloads' directory
 odo create wildfly wildfly --binary ./downloads/sample.war

 # Create new Node.js component with source from remote git repository
 odo create nodejs --git https://github.com/openshift/nodejs-ex.git

 # Create new Node.js git component while specifying a branch, tag or commit ref
 odo create nodejs --git https://github.com/openshift/nodejs-ex.git --ref master

 # Create new Node.js git component while specifying a tag
 odo create nodejs --git https://github.com/openshift/nodejs-ex.git --ref v1.0.1

 # Create new Node.js component with the source in current directory and ports 8080-tcp,8100-tcp
and 9100-udp exposed
 odo create nodejs --port 8080,8100/tcp,9100/udp

 # Create new Node.js component with the source in current directory and env variables key=value
and key1=value1 exposed
 odo create nodejs --env key=value,key1=value1

 # Create a new Python component with the source in a Git repository
 odo create python --git https://github.com/openshift/django-ex.git

 # Passing memory limits
 odo create nodejs --memory 150Mi
 odo create nodejs --min-memory 150Mi --max-memory 300 Mi

 # Passing cpu limits
 odo create nodejs --cpu 2
 odo create nodejs --min-cpu 200m --max-cpu 2
```

2.8.1.6. debug

调试组件。

使用 debug 的示例

```
# Displaying information about the state of debugging
odo debug info

# Starting the port forwarding for a component to debug the application
odo debug port-forward

# Setting a local port to port forward
odo debug port-forward --local-port 9292
```

2.8.1.7. delete

删除一个现有组件。

使用 delete 的示例

```
  # Delete component named 'frontend'.
  odo delete frontend
  odo delete frontend --all-apps
```

2.8.1.8. describe

描述指定组件。

使用 describe 的示例

```
  # Describe nodejs component
  odo describe nodejs
```

2.8.1.9. link

将组件链接到服务或组件。

使用 link 的示例

```
  # Link the current component to the 'my-postgresql' service
  odo link my-postgresql

  # Link component 'nodejs' to the 'my-postgresql' service
  odo link my-postgresql --component nodejs

  # Link current component to the 'backend' component (backend must have a single exposed port)
  odo link backend

  # Link component 'nodejs' to the 'backend' component
  odo link backend --component nodejs

  # Link current component to port 8080 of the 'backend' component (backend must have port 8080
exposed)
  odo link backend --port 8080
```

link 会把适当的 secret 添加到源组件的环境中。然后源组件就可以使用 secret 条目作为环境变量。如果没有提供源组件，则会假定当前活跃的组件。

2.8.1.10. list

列出当前应用程序中的所有组件以及组件的状态。

组件的状态

Pushed

推送到集群的组件。

Not Pushed

不推送到集群的组件。

Unknown

odo 从集群中断开连接。

使用 list 的示例

```
# List all components in the application
odo list

# List all the components in a given path
odo list --path <path_to_your_component>
```

2.8.1.11. log

获取指定组件的日志。

使用 log 的示例

```
# Get the logs for the nodejs component
odo log nodejs
```

2.8.1.12. login

登录到集群。

使用 login 的示例

```
# Log in interactively
odo login

# Log in to the given server with the given certificate authority file
odo login localhost:8443 --certificate-authority=/path/to/cert.crt

# Log in to the given server with the given credentials (basic auth)
odo login localhost:8443 --username=myuser --password=mypass

# Log in to the given server with the given credentials (token)
odo login localhost:8443 --token=xxxxxxxxxxxxxxxxxxxxxxxxx
```

2.8.1.13. logout

登出当前 OpenShift Container Platform 会话。

使用 logout 的示例

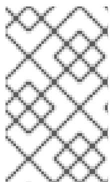
```
# Log out
odo logout
```

2.8.1.14. preference

在全局首选项文件中修改与 **odo** 相关的配置设置。

使用 preference 的示例

```
# For viewing the current preferences
odo preference view

# Set a preference value in the global preference
odo preference set UpdateNotification false
odo preference set NamePrefix "app"
odo preference set Timeout 20

# Enable experimental mode
odo preference set experimental true

# Unset a preference value in the global preference
odo preference unset  UpdateNotification
odo preference unset  NamePrefix
odo preference unset  Timeout

# Disable experimental mode
odo preference set experimental false

# Use persistent volumes in the cluster
odo preference set ephemeral false
```



注意

默认情况下，全局首选项文件的路径为 **~/.odo/preferece.yaml**，它被存储在环境变量 **GLOBALODOCONFIG** 中。您可以通过为环境变量设置一个新的值来指定一个新的自定义的首选项路径，例如: **GLOBALODOCONFIG="new_path/preference.yaml"**

表 2.3. 可用参数：

NamePrefix	默认前缀是当前目录的名称。使用这个值设置默认名称前缀。
Timeout	OpenShift Container Platform 服务器的连接检查超时时间（以秒为单位）。
UpdateNotification	控制是否显示更新通知。

2.8.1.15. project

执行项目操作。

使用 project 示例

```
# Set the active project
odo project set

# Create a new project
odo project create myproject

# List all the projects
odo project list

# Delete a project
odo project delete myproject

# Get the active project
odo project get
```

2.8.1.16. push

将源代码推送到组件中。

使用 push 的示例

```
# Push source code to the current component
odo push

# Push data to the current component from the original source.
odo push

# Push source code in ~/mycode to component called my-component
odo push my-component --context ~/mycode

# Push source code and display event notifications in JSON format.
odo push -o json
```

2.8.1.17. registry

创建和修改自定义 registry。

使用 registry 的示例

```
# Add a registry to the registry list
odo registry add <registry name> <registry URL>

# List a registry in the registry list
odo registry list

# Delete a registry from the registry list
odo registry delete <registry name>
```

```
# Update a registry in the registry list
odo registry update <registry name> <registry URL>

# List a component with a corresponding registry
odo catalog list components

# Create a component that is hosted by a specific registry
odo create <component type> --registry <registry name>
```

2.8.1.18. service

执行服务目录操作。

使用 service 的示例

```
  # Create new postgresql service from service catalog using dev plan and name my-postgresql-db.
  odo service create dh-postgresql-apb my-postgresql-db --plan dev -p postgresql_user=luke -p
postgresql_password=secret

  # Delete the service named 'mysql-persistent'
  odo service delete mysql-persistent

  # List all services in the application
  odo service list
```

2.8.1.19. storage

执行存储操作。

使用 storage 的示例

```
  # Create storage of size 1Gb to a component
  odo storage create mystorage --path=/opt/app-root/src/storage/ --size=1Gi

  # Delete storage mystorage from the currently active component
  odo storage delete mystorage

  # List all storage attached or mounted to the current component and
  # all unattached or unmounted storage in the current application
  odo storage list

  # Set the `-o json` flag to get a JSON formatted output
  odo storage list -o json
```

2.8.1.20. unlink

取消链接组件或服务。

要使这个命令成功，服务或组件需要在使用 **odo link** 前已被链接。

使用 unlink 的示例

```
  # Unlink the 'my-postgresql' service from the current component
```

```
  odo unlink my-postgresql

  # Unlink the 'my-postgresql' service  from the 'nodejs' component
  odo unlink my-postgresql --component nodejs

  # Unlink the 'backend' component from the current component (backend must have a single
exposed port)
  odo unlink backend

  # Unlink the 'backend' service  from the 'nodejs' component
  odo unlink backend --component nodejs

  # Unlink the backend's 8080 port from the current component
  odo unlink backend --port 8080
```

2.8.1.21. update

更新组件的源代码路径

使用 update 的示例

```
  # Change the source code path of a currently active component to local (use the current directory as
a source)
  odo update --local

  # Change the source code path of the frontend component to local with source in ./frontend directory
  odo update frontend --local ./frontend

  # Change the source code path of a currently active component to git
  odo update --git https://github.com/openshift/nodejs-ex.git

  # Change the source code path of the component named node-ex to git
  odo update node-ex --git https://github.com/openshift/nodejs-ex.git

  # Change the source code path of the component named wildfly to a binary named sample.war in
./downloads directory
  odo update wildfly --binary ./downloads/sample.war
```

2.8.1.22. url

向外界开放一个组件。

使用 url 的示例

```
  # Create a URL for the current component with a specific port
  odo url create --port 8080

  # Create a URL with a specific name and port
  odo url create example --port 8080

  # Create a URL with a specific name by automatic detection of port (only for components which
expose only one service port)
  odo url create example
```

```
# Create a URL with a specific name and port for component frontend
odo url create example --port 8080 --component frontend

# Delete a URL to a component
odo url delete myurl

# List the available URLs
odo url list

# Create a URL in the configuration and apply the changes to the cluster
odo url create --now

# Create an HTTPS URL
odo url create --secure
```

使用这个命令生成的 URL 可以从集群外访问部署的组件。

2.8.1.23. utils

终端命令和修改 odo 配置的工具。

使用 utils 的示例

```
# Bash terminal PS1 support
source <(odo utils terminal bash)

# Zsh terminal PS1 support
source <(odo utils terminal zsh)
```

2.8.1.24. version

打印客户端版本信息。

使用 version 的示例

```
# Print the client version of odo
odo version
```

2.8.1.25. watch

odo 启动更改监控，并在出现更改时自动更新组件。

使用 watch 的示例

```
# Watch for changes in directory for current component
odo watch

# Watch for changes in directory for component called frontend
odo watch frontend
```

2.9. ODO 架构

本节论述了 **odo** 架构以及 **odo** 如何在集群中管理资源。

2.9.1. 开发者设置

使用 odo，您可以在 OpenShift Container Platform 集群中从终端创建和部署应用程序。代码编辑器插件使用 odo，让用户可以从其 IDE 终端与 OpenShift Container Platform 集群交互。使用 odo 的插件示例：VS Code OpenShift Connector、OpenShift Connector for Intellij、Codewind for Eclipse Che。

odo 可在 Windows、macOS 和 Linux 操作系统上以及从任意终端运行。odo 为 bash 和 zsh 命令行 shell 提供自动完成功能。

odo 支持 Node.js 和 Java 组件。

2.9.2. OpenShift source-to-image

OpenShift Source-to-Image (S2I) 是一个开源项目，可帮助从源代码构建工件并将其注入容器镜像。S2I 通过构建源代码来生成可随时运行的镜像，无需 Dockerfile。odo 使用 S2I 构建器镜像来执行容器内的开发者源代码。

2.9.3. OpenShift 集群对象

2.9.3.1. Init 容器

Init 容器是应用程序容器启动前运行的专用容器，为应用程序容器的运行配置必要的环境。Init 容器可以包含应用程序镜像没有的文件，例如设置脚本。Init 容器始终需要运行完毕，如果任何 init 容器失败，应用程序容器就不会启动。

由 odo 创建的 pod 执行两个初始 (Init) 容器：

- **copy-supervisord** Init 容器。
- **copy-files-to-volume** Init 容器。

2.9.3.1.1. copy-supervisord

copy-supervisord Init 容器会将必要的文件复制到 **emptyDir** 卷中。主应用程序容器从 **emptyDir** 卷中使用这些文件。

复制到 emptyDir 卷中的文件：

- 二进制文件：
  - **go-init** 是一个最小的 init 系统。它作为应用程序容器中的第一个进程 (PID 1) 运行。go-init 会启动运行开发者代码的 **SupervisorD** 守护进程。处理孤立进程需要用到 go-init。
  - **SupervisorD** 是一个进程控制系统。它监控配置的进程并确保它们正在运行。如果需要，它也会重启服务。对于 odo，**SupervisorD** 会执行并监控开发者代码。
- 配置文件：
  - **supervisor.conf** 是 SupervisorD 守护进程启动时所必需的配置文件。
- 脚本：
  - 在 OpenShift S2I 概念中，**assemble-and-restart** 用于构建和部署用户源代码。assemble-and-restart 脚本首先在应用程序容器中编译用户源代码，然后重启 SupervisorD 来使用户更改生效。

 - 在 OpenShift S2I 概念中，**Run** 用于执行所编译的源代码。**run** 脚本执行 **assemble-and-restart** 脚本创建的编译代码。
 - **s2i-setup** 是一个脚本，它可创建 **assemble-and-restart** 和 run 脚本成功执行所需的文件和目录。每次应用程序容器启动时都会执行此脚本。

- 目录：
 - **language-scripts**：OpenShift S2I 允许使用自定义 **assemble** 和 **run** 脚本。**language-scripts** 目录中有几个特定语言的自定义脚本。自定义脚本提供额外的配置，以使 odo debug 正常工作。

emptyDir 卷挂载到 Init 容器和应用程序容器的 **/opt/odo** 挂载点。

2.9.3.1.2. copy-files-to-volume

copy-files-to-volume Init 容器将位于 S2I 构建器镜像中的 **/opt/app-root** 的文件复制到持久性卷中。然后该卷会挂载于应用程序容器中的同一位置 (**/opt/app-root**)。

如果 **/opt/app-root** 上没有持久性卷，则当持久性卷声明挂载到同一位置时，这个目录中的数据将会丢失。

PVC 挂载于 Init 容器中的 **/mnt** 挂载点。

2.9.3.2. 应用程序容器

应用程序容器是执行用户源代码的主要容器。

应用程序容器挂载了两个卷：

- 挂载于 **/opt/odo** 的 **emptyDir** 卷
- 挂载于 **/opt/app-root** 的持久性卷

go-init 作为应用程序容器内的第一个进程执行。然后，**go-init** 进程启动 **SupervisorD** 守护进程。

SupervisorD 执行并监控用户编译的源代码。如果用户进程崩溃，**SupervisorD** 会重新启动它。

2.9.3.3. 持久性卷和持久性卷声明

持久性卷声明（PVC）是在 Kubernetes 中置备持久性卷的卷类型。持久性卷的生命周期独立于 pod 生命周期。持久性卷上的数据会在 pod 重启后保留。

copy-files-to-volume Init 容器会将必要的文件复制到持久性卷中。主应用程序容器在运行时使用这些文件来执行。

持久性卷的命名规则是 <component_name>-s2idata。

Container	挂载到的 PVC
copy-files-to-volume	**/mnt**
应用程序容器	**/opt/app-root**

2.9.3.4. emptyDir 卷

当 pod 分配给节点时，会创建一个 **emptyDir** 卷，只要该 pod 在节点上运行，该卷即会一直存在。如果容器被重启或移动，则 **emptyDir** 的内容会被删除，Init 容器会将数据重新恢复到 **emptyDir**。**emptyDir** 最初为空。

copy-supervisord Init 容器会将必要的文件复制到 **emptyDir** 卷中。然后，主应用程序容器会在运行时使用这些文件来执行。

Container	挂载到 emptyDir 的卷
copy-supervisord	**/opt/odo**
应用程序容器	**/opt/odo**

2.9.3.5. Service

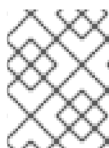
服务是一个 Kubernetes 概念，它抽象地代表了与一组 pod 的通信方式。

odo 为每个应用程序 pod 创建一个服务，使其可访问以进行通信。

2.9.4. odo push 工作流

本节论述了 **odo push** 工作流。odo push 会使用所有必要的 OpenShift Container Platform 资源在 OpenShift Container Platform 集群中部署用户代码。

1. 创建资源
   如果尚未创建，**odo push** 会创建以下 OpenShift Container Platform 资源：
   - **DeploymentConfig** 对象：
     - 执行两个 init 容器：**copy-supervisord** 和 **copy-files-to-volume**。init 容器将文件复制到 **emptyDir** 和 **PersistentVolume** 类型的卷上。
     - 应用程序容器启动。应用程序容器中的第一个进程是 PID=1 的 **go-init** 进程。
     - **go-init** 进程启动 SupervisorD 守护进程。

       注意

       用户应用程序代码尚未复制到应用程序容器中，因此 **SupervisorD** 守护进程没有执行 **run** 脚本。
   - **Service** 对象
   - **Secret** 对象
   - **PersistentVolumeClaim** 对象
2. 文件索引
   - 文件索引器会将源代码目录中的文件编成索引。索引器会以递归方式遍历源代码目录，并找到已创建、删除或重命名的文件。

   - 文件索引器在 **.odo** 目录下的 odo index 文件中维护索引信息。
   - 如果 odo index 文件不存在，这意味着文件索引程序是首次执行，并会创建新的 odo index JSON 文件。odo index JSON 文件包含文件映射 - 已遍历文件的相对文件路径以及已更改和已删除文件的绝对路径。

3. 推送代码
本地代码被复制到应用程序容器中，通常位于 **/tmp/src** 下。

4. 执行 **assemble-and-restart**
源代码成功复制后，会在运行的应用程序容器中执行 **assemble-and-restart** 脚本。

2.10. ODO 发行注记

2.10.1. odo 的主要变化和功能增强

- **odo** 与 devfile 集成不再是一个技术预览功能，现在已被完全支持。如需更多信息，请参阅使用 devfile 创建应用程序。
- **odo** 中的默认部署机制现在是 devfile 部署。
- 添加有关将 S2I 转换为 devfile 的文档。请参阅将 S2I 组件转换为 devfile 组件 。
- 添加命令以显示 devfile 组件的 JSON 输出。
- **odo debug** 不再是一个技术预览功能，现在已被完全支持。如需更多信息，请参阅在 odo 中调试应用程序。
- **odo** 与 Operator 集成不再是一个技术预览功能，现在已被完全支持。如需更多信息，请参阅 创建由 Operator 管理的服务实例。
- **odo** 现在支持 IBM Z。
- 很多命令的输出已改进并增强。

2.10.2. 获取支持

对于文档

如果您在文档中发现错误或者有改进文档的建议，请在 Bugzilla 中提交问题。选择 **OpenShift Container Platform** 产品类型和 **Documentation** 组件类型。

对于产品

如果您发现了错误，遇到问题或者有改进 **odo** 功能的建议，请在 Bugzilla 中提交问题。选择 **Red Hat odo for OpenShift Container Platform** 产品类型。

请在问题描述中提供尽可能多的细节。

2.10.3. 已知问题

- Bug 1760574 已删除的命名空间在 **odo project get** 命令中列出。
- Bug 1760586 **odo delete** 命令在项目被删除及一个组件名称被设置后，会出现一个死循环。

- Bug 1760588 **odo service create** 命令在 Cygwin 中运行时会崩溃。
- Bug 1760590 在 Git BASH for Windows 中， **odo login -u developer** 命令在请求时没有隐藏输入的密码。
- Bug 1783188 在断开连接的集群中，**odo component create** 命令会抛出一个错误 **...tag not found...** 尽管组件列在目录列表中。
- Bug 1761440 在一个项目中不可能创建同一类型的两个服务。
- Bug 1821643 **odo push** 在 .NET 组件标签 2.1+ 上无法工作。
  临时解决方案： 运行以下命令来指定您的 .NET 项目文件：

```
$ odo config set --env DOTNET_STARTUP_PROJECT=<path to your project file>
```

第 3 章 HELM CLI

3.1. 在 OPENSHIFT CONTAINER PLATFORM 中使用 HELM 3

3.1.1. 了解 Helm

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

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

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

在每次安装 chart，或一个版本被升级或回滚时，都会创建增量修订版本。

3.1.1.1. 主要特性

Helm 提供以下功能：

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

3.1.2. 安装 Helm

下面的部分论述了如何使用 CLI 在不同的平台中安装 Helm。

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

先决条件

- 已安装了 Go 版本 1.13 或更高版本。

3.1.2.1. 对于 Linux

1. 下载 Helm 二进制文件并将其添加到您的路径中：

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

2. 使二进制文件可执行：

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

3. 检查已安装的版本：

```
$ helm version
```

输**出示例**

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

3.1.2.2. 对于 Windows 7/8

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

3.1.2.3. 对于 Windows 10

1. 下载最新的 **.exe** 文件并放入您自己选择的目录。
2. 点击 **搜索** 并输入 **env** 或者 **environment**。
3. 选择为**您的帐户编辑环境变量**。
4. 在变量部分选择**路径**并点**编辑**。
5. 点**新建**并输入到 exe 文件所在目录的路径，或者点击 **浏览** 并选择目录，然后点击**确定**。

3.1.2.4. 对于 macOS：

1. 下载 Helm 二进制文件并将其添加到您的路径中：

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

2. 使二进制文件可执行：

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

3. 检查已安装的版本：

```
$ helm version
```

输**出示例**

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

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

先决条件

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

流程

1. 创建一个新项目

```
$ oc new-project mysql
```

2. 将一个 Helm chart 存储库添加到本地 Helm 客户端：

```
$ helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

输出示例

```
"stable" has been added to your repositories
```

3. 更新存储库：

```
$ helm repo update
```

4. 安装示例 MySQL chart：

```
$ helm install example-mysql stable/mysql
```

5. 验证 chart 是否已成功安装：

```
$ helm list
```

输出示例

```
NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION
example-mysql mysql 1 2019-12-05 15:06:51.379134163 -0500 EST deployed mysql-1.5.0
5.7.27
```

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

流程

1. 创建一个新项目

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

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

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

–

3. 进入包含 chart 示例的目录：

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

4. 编辑 **Chart.yaml** 文件并添加 chart 描述：

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

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

(2) chart 的名称。

(3) chart 的描述。

(4) 用作图标的图形的 URL。

5. 验证 chart 格式是否正确：

```
$ helm lint
```

输出示例

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

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

6. 前往上一个目录级别：

```
$ cd ..
```

7. 安装 chart：

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

8. 验证 chart 是否已成功安装：

```
$ helm list
```

输出示例

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

3.2. 配置自定义 HELM CHART 仓库

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

作为集群管理员，您可以添加多个 Helm Chart 仓库（默认仓库除外），并在 **Developer Catalog** 中显示这些仓库中的 Helm chart。

3.2.1. 添加自定义 Helm Chart 仓库

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

流程

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

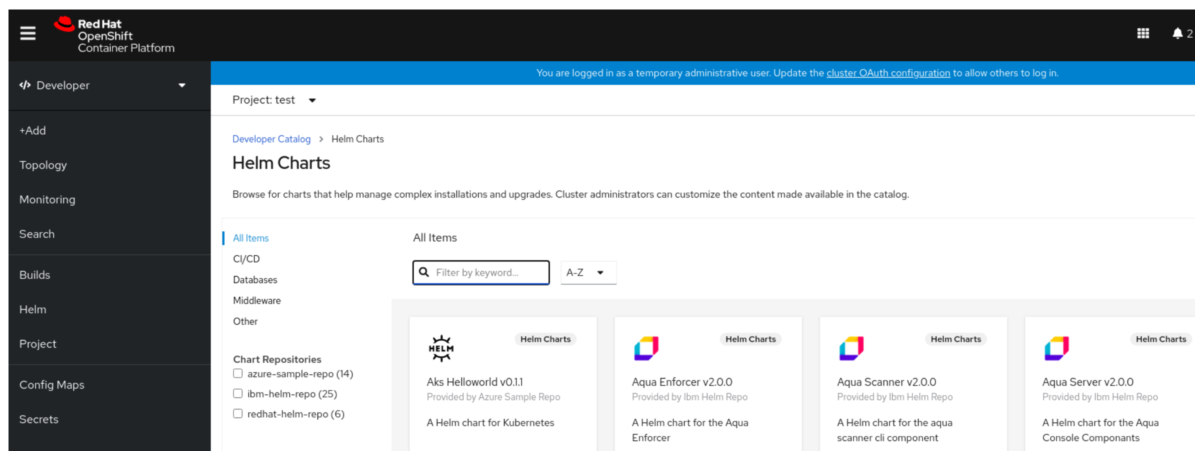
```
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <name>
spec:
 # optional name that might be used by console
 # name: <chart-display-name>
  connectionConfig:
    url: <helm-chart-repository-url>
```

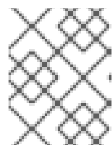
例如，要添加 Azure 示例 chart 存储库，请运行：

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

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

图 3.1. Chart 软件仓库过滤器





注意

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

3.2.2. 创建凭证和 CA 证书以添加 Helm Chart 仓库

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

流程

配置凭证和证书，然后使用 CLI 添加 Helm Chart 仓库：

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

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

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

```
$ oc create secret generic helm-tls-configs \
--from-file=tls.crt=/path/to/certs/client.crt \
--from-file=tls.key=/path/to/certs//client.key \
-n openshift-config
```

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

3. 按如下所示添加 Helm 仓库：

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <helm-repository>
spec:
  name: <helm-repository>
  connectionConfig:
```

```
    url: <URL for the Helm repository>
    tlsConfig:
        name: helm-tls-configs
    ca:
 name: helm-ca-cert
EOF
```

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

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

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

3.3. 禁用 HELM HART 仓库

作为集群管理员，您可以删除集群中的 Helm Chart 仓库，以便在 **Developer Catalog** 中不再显示它们。

3.3.1. 在集群中禁用 Helm Chart 仓库

您可以通过在 **HelmChartRepository** 自定义资源中添加 **disabled** 属性来禁用目录中的 Helm Charts。

流程

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

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

- 使用 Web 控制台禁用最近添加的 Helm Chart 仓库：

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

示例

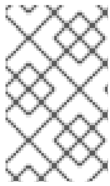
```
spec:
  connectionConfig:
    url: <url-of-the-repositoru-to-be-disabled>
  disabled: true
```

现在，这个仓库已被禁用，并不会出现在目录中。

第 4 章 用于 OPENSHIFT SERVERLESS 的 KNATIVE CLI（KN）

Knative **kn** CLI 在 OpenShift Container Platform 上启用了与 Knative 组件的简单交互。

您可以通过安装 OpenShift Serverless 在 OpenShift Container Platform 上启用 Knative。如需更多信息，请参阅开始使用 OpenShift Serverless。



注意

OpenShift Serverless 不能使用 **kn** CLI 安装。集群管理员必须安装 OpenShift Serverless Operator 并设置 Knative 组件，如 OpenShift Container Platform 的 Serverless 应用程序文档中所述。

4.1. 主要特性

kn CLI 旨在使无服务器计算任务简单而简洁。**kn** CLI 的主要功能包括：

- 从命令行部署无服务器应用程序。
- 管理 Knative Serving 的功能，如服务、修订和流量分割。
- 创建和管理 Knative Eventing 组件，如事件源和触发器。
- 创建 sink 绑定来连接现有的 Kubernetes 应用程序和 Knative 服务。
- 使用灵活的插件架构扩展 **kn** CLI，类似于 **kubectl** CLI。
- 为 Knative 服务配置 autoscaling 参数。
- 脚本化使用，如等待一个操作的结果，或部署自定义推出和回滚策略。

4.2. 安装 KNATIVE CLI

请参阅安装 Knative CLI。

第 5 章 PIPELINES CLI (TKN)

5.1. 安装 TKN

通过一个终端，使用 **tkn** CLI 管理 Red Hat OpenShift Pipelines。下面的部分论述了如何在不同的平台中安装 **tkn**。

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

5.1.1. 在 Linux 上安装 Red Hat OpenShift Pipelines CLI（tkn）

对于 Linux 系统，您可以直接将 CLI 下载为 **tar.gz** 存档。

流程

1. 下载相关的 CLI。
    - Linux (x86_64, amd64)
    - Linux on IBM Z and LinuxONE (s390x)
    - Linux on IBM Power Systems (ppc64le)
2. 解包存档：

```
$ tar xvzf <file>
```

3. 把 **tkn** 二进制代码放到 **PATH** 中的一个目录下。
4. 运行以下命令可以查看 **PATH** 的值：

```
$ echo $PATH
```

5.1.2. 使用 RPM 在 Linux 上安装 Red Hat OpenShift Pipelines CLI（tkn）

对于 Red Hat Enterprise Linux（RHEL）版本 8，您可以使用 RPM 安装 Red Hat OpenShift Pipelines CLI（**tkn**）。

先决条件

- 您的红帽帐户必须具有有效的 OpenShift Container Platform 订阅。
- 您在本地系统中有 root 或者 sudo 权限。

流程

1. 使用 Red Hat Subscription Manager 注册：

```
# subscription-manager register
```

2. 获取最新的订阅数据：

```
# subscription-manager refresh
```

–

3. 列出可用的订阅：

```
# subscription-manager list --available --matches '*pipelines*'
```

4. 在上一命令的输出中，找到 OpenShift Container Platform 订阅的池 ID，并把订阅附加到注册的系统：

```
# subscription-manager attach --pool=<pool_id>
```

5. 启用 Red Hat OpenShift Pipelines 所需的仓库：

```
# subscription-manager repos --enable="pipelines-1.3-for-rhel-8-x86_64-rpms"
```

6. 安装 **openshift-pipelines-client** 软件包：

```
# yum install openshift-pipelines-client
```

安装 CLI 后，就可以使用**tkn**命令：

```
$ tkn version
```

5.1.3. 在 Windows 上安装 Red Hat OpenShift Pipelines CLI（tkn）

对于 Windows，**tkn** CLI 以一个 **zip** 文件的形式提供。

流程

1. 下载 CLI。
2. 使用 ZIP 程序解压存档。
3. 将 **tkn.exe** 文件的位置添加到 **PATH** 环境变量中。
4. 要查看您的 **PATH**，请打开命令窗口并运行以下命令：

```
C:\> path
```

5.1.4. 在 macOS 上安装 Red Hat OpenShift Pipelines CLI（tkn）

对于 macOS，**tkn** CLI 以一个 **tar.gz** 文件的形式提供。

流程

1. 下载 CLI。
2. 解包和解压存档。
3. 将 **tkn** 二进制文件迁移至 PATH 上的目录中。
4. 要查看 **PATH**，打开终端窗口并运行：

```
$ echo $PATH
```

5.2. 配置 OPENSHIFT PIPELINES TKN CLI

配置 Red Hat OpenShift Pipelines **tkn** CLI 以启用 tab 自动完成功能。

5.2.1. 启用 tab 自动完成功能

在安装**tkn** CLI，可以启用 tab 自动完成功能，以便在按 Tab 键时自动完成 **tkn**命令或显示建议选项。

先决条件

- 已安装**tkn** CLI。
- 需要在本地系统中安装了 **bash-completion**。

流程

以下过程为 Bash 启用 tab 自动完成功能。

1. 将 Bash 完成代码保存到一个文件中：

```
$ tkn completion bash > tkn_bash_completion
```

2. 将文件复制到 **/etc/bash_completion.d/**：

```
$ sudo cp tkn_bash_completion /etc/bash_completion.d/
```

   您也可以将文件保存到一个本地目录，并从您的**.bashrc**文件中 source 这个文件。

开新终端时 tab 自动完成功能将被启用。

5.3. OPENSHIFT PIPELINES TKN 参考

本节列出了基本的 **tkn** CLI 命令。

5.3.1. 基本语法

tkn [command or options] [arguments…]

5.3.2. 全局选项

--help, -h

5.3.3. 工具命令

5.3.3.1. tkn

tkn CLI 的主命令。

示例： 显示所有选项

```
$ tkn
```

5.3.3.2. completion [shell]

输出 shell 完成代码，必须经过评估方可提供互动完成。支持的 shell 是 **bash** 和 **zsh**。

示例：bash shell 完成代码

```
$ tkn completion bash
```

5.3.3.3. version

输出 **tkn** CLI 的版本信息。

示例： 检查 tkn 版本

```
$ tkn version
```

5.3.4. Pipelines 管理命令

5.3.4.1. pipeline

管理管道。

示例： 显示帮助信息

```
$ tkn pipeline --help
```

5.3.4.2. pipeline delete

删除 Pipeline

示例： 从命名空间中删除 mypipeline Pipeline

```
$ tkn pipeline delete mypipeline -n myspace
```

5.3.4.3. pipeline describe

描述管道。

示例： 描述 mypipeline Pipeline

```
$ tkn pipeline describe mypipeline
```

5.3.4.4. pipeline list

列出管道。

示例： 显示 Pipelines 列表

```
$ tkn pipeline list
```

5.3.4.5. pipeline logs

显示特定 Pipeline 的 Pipeline 日志。

示例：mypipeline Pipeline 的 Stream live 日志

```
$ tkn pipeline logs -f mypipeline
```

5.3.4.6. pipeline start

运行 Pipeline。

示例： 启动 mypipeline Pipeline

```
$ tkn pipeline start mypipeline
```

5.3.5. PipelineRun 命令

5.3.5.1. pipelinerun

管理 PipelineRuns。

示例： 显示帮助信息

```
$ tkn pipelinerun -h
```

5.3.5.2. pipelinerun cancel

取消 PipelineRun。

示例：从命名空间中取消 mypipelinerun PipelineRun

```
$ tkn pipelinerun cancel mypipelinerun -n myspace
```

5.3.5.3. pipelinerun delete

删除 PipelineRun。

示例： 从命名空间中删除 PipelineRuns

```
$ tkn pipelinerun delete mypipelinerun1 mypipelinerun2 -n myspace
```

5.3.5.4. pipelinerun describe

描述 PipelineRun。

示例：描述命名空间中的 mypipelinerun PipelineRun

```
$ tkn pipelinerun describe mypipelinerun -n myspace
```

5.3.5.5. pipelinerun list

列出 PipelineRuns。

示例： 显示命名空间中的 PipelineRuns 列表

```
$ tkn pipelinerun list -n myspace
```

5.3.5.6. pipelinerun logs

显示一个 PipelineRun 的日志。

示例：显示 mypipelinerun PipelineRun 的日志，包括命名空间中的所有任务和步骤

```
$ tkn pipelinerun logs mypipelinerun -a -n myspace
```

5.3.6. 任务管理命令

5.3.6.1. task

管理任务。

示例： 显示帮助信息

```
$ tkn task -h
```

5.3.6.2. task delete

删除一个任务。

示例：从命令空间中删除 mytask1 和 mytask2 任务

```
$ tkn task delete mytask1 mytask2 -n myspace
```

5.3.6.3. task describe

描述一个任务。

示例：描述一个命名空间中的 mytask 任务

```
$ tkn task describe mytask -n myspace
```

5.3.6.4. task list

列出任务。

示例： 列出命名空间中的所有任务

```
$ tkn task list -n myspace
```

5.3.6.5. task logs

显示任务日志。

示例：显示 mytask 任务的 mytaskrun TaskRun 的日志

```
$ tkn task logs mytask mytaskrun -n myspace
```

5.3.6.6. task start

启动一个任务。

示例： 在命名空间中启动 mytask 任务

```
$ tkn task start mytask -s <ServiceAccountName> -n myspace
```

5.3.7. TaskRun 命令

5.3.7.1. taskrun

管理 TaskRuns。

示例： 显示帮助信息

```
$ tkn taskrun -h
```

5.3.7.2. taskrun cancel

取消 TaskRun。

示例：从一个命名空间中取消 mytaskrun TaskRun

```
$ tkn taskrun cancel mytaskrun -n myspace
```

5.3.7.3. taskrun delete

删除一个 TaskRun。

示例：从一个命名空间中删除 mytaskrun1 和 mytaskrun2 TaskRuns

```
$ tkn taskrun delete mytaskrun1 mytaskrun2 -n myspace
```

5.3.7.4. taskrun describe

描述 TaskRun。

示例：描述命名空间中的 mytaskrun TaskRun

```
$ tkn taskrun describe mytaskrun -n myspace
```

5.3.7.5. taskrun list

列出 TaskRuns。

示例： 列出命名空间中的所有 TaskRuns

```
$ tkn taskrun list -n myspace
```

5.3.7.6. taskrun logs

显示 TaskRun 日志。

示例： 显示命名空间中 mytaskrun TaskRun 的实时日志

```
$ tkn taskrun logs -f mytaskrun -n myspace
```

5.3.8. 条件管理命令

5.3.8.1. 条件

管理条件（Condition）。

示例： 显示帮助信息

```
$ tkn condition --help
```

5.3.8.2. 删除条件

删除一个条件。

示例：从命名空间中删除 mycondition1 Condition

```
$ tkn condition delete mycondition1 -n myspace
```

5.3.8.3. condition describe

描述条件。

示例：在命名空间中描述 mycondition1 Condition

```
$ tkn condition describe mycondition1 -n myspace
```

5.3.8.4. condition list

列出条件。

示例： 列出命名空间中的条件

```
$ tkn condition list -n myspace
```

5.3.9. Pipeline 资源管理命令

5.3.9.1. resource

管理管道资源。

示例：显示帮助信息

```
$ tkn resource -h
```

5.3.9.2. resource create

创建一个 Pipeline 资源。

示例：在命名空间中创建一个 Pipeline 资源

```
$ tkn resource create -n myspace
```

这是一个交互式命令，它要求输入资源名称、资源类型以及基于资源类型的值。

5.3.9.3. resource delete

删除 Pipeline 资源。

示例：从命名空间中删除 myresource Pipeline 资源

```
$ tkn resource delete myresource -n myspace
```

5.3.9.4. resource describe

描述管道资源。

示例：描述 myresource Pipeline 资源

```
$ tkn resource describe myresource -n myspace
```

5.3.9.5. resource list

列出管道资源。

示例：列出命名空间中的所有管道资源

```
$ tkn resource list -n myspace
```

5.3.10. ClusterTask 管理命令

5.3.10.1. clustertask

管理 ClusterTasks。

示例：显示帮助信息

```
$ tkn clustertask --help
```

5.3.10.2. clustertask delete

删除集群中的 ClusterTask 资源。

示例：删除 mytask1 和 mytask2 ClusterTasks

```
$ tkn clustertask delete mytask1 mytask2
```

5.3.10.3. clustertask describe

描述 ClusterTask。

示例：描述 mytask ClusterTask

```
$ tkn clustertask describe mytask1
```

5.3.10.4. clustertask list

列出 ClusterTasks。

示例：列出 ClusterTasks

```
$ tkn clustertask list
```

5.3.10.5. clustertask start

启动 ClusterTasks。

示例：启动 mytask ClusterTask

```
$ tkn clustertask start mytask
```

5.3.11. 触发器管理命令

5.3.11.1. eventlistener

管理 EventListeners。

示例：显示帮助信息

```
$ tkn eventlistener -h
```

5.3.11.2. eventlistener delete

删除一个 EventListener。

示例：删除命令空间中的 mylistener1 和 mylistener2 EventListeners

```
$ tkn eventlistener delete mylistener1 mylistener2 -n myspace
```

5.3.11.3. eventlistener describe

描述 EventListener。

示例：描述命名空间中的 mylistener EventListener

```
$ tkn eventlistener describe mylistener -n myspace
```

5.3.11.4. eventlistener list

列出 EventListeners。

示例： 列出命名空间中的所有 EventListeners

```
$ tkn eventlistener list -n myspace
```

5.3.11.5. triggerbinding

管理 TriggerBindings。

示例： 显示 TriggerBindings 帮助信息

```
$ tkn triggerbinding -h
```

5.3.11.6. triggerbinding delete

删除 TriggerBinding。

示例：删除一个命名空间中的 mybinding1 和 mybinding2 TriggerBindings

```
$ tkn triggerbinding delete mybinding1 mybinding2 -n myspace
```

5.3.11.7. triggerbinding describe

描述 TriggerBinding。

示例：描述命名空间中的 mybinding TriggerBinding

```
$ tkn triggerbinding describe mybinding -n myspace
```

5.3.11.8. triggerbinding list

列出 TriggerBindings。

示例： 列出命名空间中的所有 TriggerBindings

```
$ tkn triggerbinding list -n myspace
```

5.3.11.9. triggertemplate

管理 TriggerTemplates。

示例： 显示 TriggerTemplate 帮助

```
$ tkn triggertemplate -h
```

5.3.11.10. triggertemplate delete

删除 TriggerTemplate。

示例：删除命名空间中的 mytemplate1 和 mytemplate2 TriggerTemplates

```
$ tkn triggertemplate delete mytemplate1 mytemplate2 -n `myspace`
```

5.3.11.11. triggertemplate describe

描述 TriggerTemplate。

示例： 描述命名空间中的 mytemplate TriggerTemplate

```
$ tkn triggertemplate describe mytemplate -n `myspace`
```

5.3.11.12. triggertemplate list

列出 TriggerTemplates。

示例： 列出命名空间中的所有 TriggerTemplates

```
$ tkn triggertemplate list -n myspace
```

5.3.11.13. clustertriggerbinding

管理 ClusterTriggerBindings。

示例： 显示 ClusterTriggerBindings 帮助信息

```
$ tkn clustertriggerbinding -h
```

5.3.11.14. clustertriggerbinding delete

删除 ClusterTriggerBinding。

示例： 删除 myclusterbinding1 和 myclusterbinding2 ClusterTriggerBindings

```
$ tkn clustertriggerbinding delete myclusterbinding1 myclusterbinding2
```

5.3.11.15. clustertriggerbinding describe

描述 ClusterTriggerBinding。

示例： 描述 myclusterbinding ClusterTriggerBinding

```
$ tkn clustertriggerbinding describe myclusterbinding
```

5.3.11.16. clustertriggerbinding list

列出 ClusterTriggerBindings。

示例： 列出所有 ClusterTriggerBindings

```
$ tkn clustertriggerbinding list
```

第 6 章 OPM CLI

6.1. 关于 OPM

opm CLI 工具由 Operator Framework 提供，用于 Operator Bundle Format。您可以通过一个名为 *index* 的捆绑包列表创建和维护 Operator 目录，类似于软件存储库。其结果是一个名为 *index image（索引镜像）* 的容器镜像，它可以存储在容器的 registry 中，然后安装到集群中。

索引包含一个指向 Operator 清单内容的指针数据库，可通过在运行容器镜像时提供的已包含 API 进行查询。在 OpenShift Container Platform 中，Operator Lifecycle Manager（OLM）可通过在 **CatalogSource** 中引用索引镜像来使它作为目录一个（catalog），它会定期轮询镜像，以对集群上安装的 Operator 进行更新。

其他资源

- 如需有关捆绑格式的更多信息，请参阅 Operator Framework 打包格式。
- 要使用 Operator SDK 创建捆绑包镜像，请参阅使用 捆绑包镜像。

6.2. 安装 OPM

您可以在您的 Linux、macOS 或者 Windows 工作站上安装 **opm** CLI 工具。

先决条件

- 对于 Linux，您必须提供以下软件包：
    - **podman** 1.9.3+（推荐版本 2.0+）
    - **glibc** 版本 2.28+

流程

1. 导航到 OpenShift 镜像站点并下载与您的操作系统匹配的 tarball 的最新版本。
2. 解包存档。
    - 对于 Linux 或者 macOS:

    ```
    $ tar xvf <file>
    ```

    - 对于 Windows，使用 ZIP 程序解压存档。
3. 将文件放在 **PATH** 中的任何位置。
    - 对于 Linux 或者 macOS:
        a. 检查 **PATH**:

        ```
        $ echo $PATH
        ```

        b. 移动文件。例如：

        ```
        $ sudo mv ./opm /usr/local/bin/
        ```

- 对于 Windows:

a. 检查 **PATH**:

```
C:\> path
```

b. 移动文件：

```
C:\> move opm.exe <directory>
```

验证

- 安装 **opm** CLI 后，验证是否可用：

```
$ opm version
```

输出示例

```
Version: version.Version{OpmVersion:"v1.15.4-2-g6183dbb3",
GitCommit:"6183dbb3567397e759f25752011834f86f47a3ea", BuildDate:"2021-02-
13T04:16:08Z", GoOs:"linux", GoArch:"amd64"}
```

6.3. 其他资源

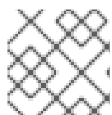
- 请参阅为 **opm** 操作管理自定义目录，包括创建、更新和修剪索引镜像。

第 7 章 OPERATOR SDK

7.1. 安装 OPERATOR SDK CLI

Operator SDK 提供了一个命令行界面（CLI）工具，Operator 开发人员可使用它来构建、测试和部署 Operator。您可以在工作站上安装 Operator SDK CLI，以便准备开始编写自己的 Operator。

如需有关 Operator SDK 的完整文档，请参阅 Operators。



注意

OpenShift Container Platform 4.7 支持 Operator SDK v1.3.0。

7.1.1. 安装 Operator SDK CLI

您可以在 Linux 上安装 OpenShift SDK CLI 工具。

先决条件

- Go v1.13+
- **docker** v17.03+、**podman** v1.9.3+ 或 **buildah** v1.7+

流程

1. 导航到 OpenShift 镜像站点。
2. 从 **latest** 的目录中，下载 Linux 的 tarball 最新版本。
3. 解包存档：

```
$ tar xvf operator-sdk-v1.3.0-ocp-linux-x86_64.tar.gz
```

4. 使文件可执行：

```
$ chmod +x operator-sdk
```

5. 将提取的 **operator-sdk** 二进制文件移到 **PATH** 中的一个目录中。

提示

检查 **PATH**：

```
$ echo $PATH
```

```
$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk
```

验证

- 安装 Operator SDK CLI 后，验证它是否可用：

```
$ operator-sdk version
```

输出示例

```
operator-sdk version: "v1.3.0-ocp", ...
```

7.2. OPERATOR SDK CLI 参考

Operator SDK 命令行界面（CLI）是一个开发组件，旨在更轻松地编写 Operator。

operator SDK CLI 语法

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

具有集群管理员访问权限的 operator 作者（如 OpenShift Container Platform）可以使用 Operator SDK CLI 根据 Go、Ansible 或 Helm 开发自己的 Operator。Kubebuilder 作为基于 Go 的 Operator 的构建解决方案嵌入到 Operator SDK 中，这意味着现有的 Kubebuilder 项目可以象 Operator SDK 一样使用并继续工作。

如需有关 Operator SDK 的完整文档，请参阅 Operators。

7.2.1. bundle

operator-sdk bundle 命令管理 Operator 捆绑包元数据。

7.2.1.1. validate

bundle validate 子命令会验证 Operator 捆绑包。

表 7.1. bundle validate 标记

标记	描述
-h, --help	**bundle validate** 子命令的帮助输出。
--index-builder（字符串）	拉取和解包捆绑包镜像的工具。仅在验证捆绑包镜像时使用。可用选项是 **docker**（默认值）、**podman** 或 **none**。
--list-optional	列出所有可用的可选验证器。设置后，不会运行验证器。
--select-optional（字符串）	选择要运行的可选验证器的标签选择器。当使用 **--list-optional** 标志运行时，会列出可用的可选验证器。

7.2.2. cleanup

operator-sdk cleanup 命令会销毁并删除为通过 **run** 命令部署的 Operator 创建的资源。

表 7.2. cleanup 标记

标记	描述
-h，--help	**run bundle** 子命令的帮助输出。
--kubeconfig (string)	用于 CLI 请求的 **kubeconfig** 文件的路径。
n、--namespace（字符串）	如果存在，代表在其中运行 CLI 请求的命名空间。
--timeout <duration>	失败前，等待命令完成的时间。默认值为 **2m0s**。

7.2.3. completion

operator-sdk completion 命令生成 shell completion，以便更迅速、更轻松地发出 CLI 命令。

表 7.3. **completion** 子命令

子命令	描述
bash	生成 bash completion。
zsh	生成 zsh completion。

表 7.4. **completion** 标记

标记	描述
-h, --help	使用方法帮助输出。

例如：

```
$ operator-sdk completion bash
```

输出示例

```
# bash completion for operator-sdk                         -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

7.2.4. create

operator-sdk create 命令用于创建或 *scaffold* Kubernetes API。

7.2.4.1. api

create api 子命令构建 Kubernetes API。子命令必须在 **init** 命令初始化的项目中运行。

表 7.5. **create api** 标记

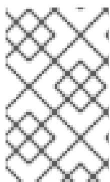
标记	描述
-h，--help	**run bundle** 子命令的帮助输出。

7.2.5. generate

operator-sdk generate 命令调用特定的生成器来生成代码或清单。

7.2.5.1. bundle

generate bundle 子命令为您的 Operator 项目生成一组捆绑包清单、元数据和 **bundle.Dockerfile** 文件。



注意

通常，您首先运行 **generate kustomize manifests** 子命令来生成由 **generate bundle** 子命令使用的输入 Kustomize 基础。但是，您可以使用初始项目中的 **make bundle** 命令按顺序自动运行这些命令。

表 7.6. generate bundle 标记

标记	描述
--channels（字符串）	捆绑包所属频道的以逗号分隔的列表。默认值为 **alpha**。
--crds-dir（字符串）	**CustomResoureDefinition** 清单的根目录。
--default-channel（字符串）	捆绑包的默认频道。
--deploy-dir（字符串）	Operator 清单的根目录，如部署和 RBAC。这个目录与传递给 **--input-dir** 标记的目录不同。
-h，--help	**generate bundle** 的帮助信息
--input-dir（字符串）	从中读取现有捆绑包的目录。这个目录是捆绑包 **manifests** 目录的父目录，它与 **--deploy-dir** 目录不同。
--kustomize-dir（字符串）	包含 Kustomize 基础的目录以及用于捆绑包清单的 **kustomization.yaml** 文件。默认路径为 **config/manifests**。
--manifests	生成捆绑包清单。
--metadata	生成捆绑包元数据和 Dockerfile。
--output-dir（字符串）	将捆绑包写入的目录。
--overwrite	如果捆绑包元数据和 Dockerfile 存在，则覆盖它们。默认值为 **true**。

标记	描述
--package（字符串）	捆绑包的软件包名称。
-q、**--quiet**	在静默模式下运行。
--stdout	将捆绑包清单写入标准输出。
--version（字符串）	生成的捆绑包中的 Operator 语义版本。仅在创建新捆绑包或升级 Operator 时设置。

其他资源

- 如需了解包括使用 **make bundle** 命令来调用 **generate bundle** 子命令的完整流程，请参阅捆绑 Operator 和 Operator Lifecycle Manager 部署。

7.2.5.2. kustomize

generate kustomize 子命令包含为 Operator 生成 Kustomize 数据的子命令。

7.2.5.2.1. 清单

generate kustomize manifests 子命令生成或重新生成 Kustomize 基础以及 **config/manifests** 目录中的 **kustomization.yaml** 文件，用于其他 Operator SDK 命令构建捆绑包清单。在默认情况下，这个命令会以互动方式询问 UI 元数据，即清单基础的重要组件，除非基础已存在或设置了 **--interactive=false** 标志。

表 7.7. generate kustomize manifests 标记

标记	描述
--apis-dir（字符串）	API 类型定义的根目录。
-h，**--help**	**generate kustomize manifests** 的帮助信息。
--input-dir（字符串）	包含现有 Kustomize 文件的目录。
--interactive	当设置为 **false** 时，如果没有 Kustomize 基础，则会出现交互式命令提示符来接受自定义元数据。
--output-dir（字符串）	写入 Kustomize 文件的目录。
--package（字符串）	软件包名称。
-q、**--quiet**	在静默模式下运行。

7.2.6. init

operator-sdk init 命令初始化 Operator 项目，并为给定插件生成或 *scaffolds* 默认项目目录布局。

这个命令会写入以下文件：

- boilerplate 许可证文件
- 带有域和库的**PROJECT** 文件
- 构建项目的**Makefile**
- **go.mod** 文件带有项目依赖项
- 用于自定义清单的 **kustomization.yaml** 文件
- 用于为管理器清单自定义镜像的补丁文件
- 启用 Prometheus 指标的补丁文件
- 运行的 **main.go** 文件

表 7.8. **init** 标记

标记	描述
--help, -h	**init** 命令的帮助输出。
--plugins（字符串）	插件的名称和可选版本，用于初始化项目。可用插件包括 **ansible.sdk.operatorframework.io/v1**、**go.kubebuilder.io/v2**、**go.kubebuilder.io/v3** 和 **helm.sdk.operatorframework.io/v1**。
--project-version	项目版本。可用值为 **2** 和 **3-alpha**（默认值）。

7.2.7. run

operator-sdk run 命令提供可在各种环境中启动 Operator 的选项。

7.2.7.1. bundle

run bundle 子命令使用 Operator Lifecycle Manager（OLM）以捆绑包格式部署 Operator。

表 7.9. **run bundle** 标记

标记	描述
--index-image（字符串）	在其中注入捆绑包的索引镜像。默认镜像为 **quay.io/operator-framework/upstream-opm-builder:latest**。
--install-mode <install_mode_value>	安装 Operator 的集群服务版本（CSV）支持的模式，如 **AllNamespaces** 或 **SingleNamespace**。
--timeout <duration>	安装超时。默认值为 **2m0s**。

标记	描述
--kubeconfig (string)	用于 CLI 请求的 **kubeconfig** 文件的路径。
n、**--namespace**（字符串）	如果存在，代表在其中运行 CLI 请求的命名空间。
-h，**--help**	**run bundle** 子命令的帮助输出。

其他资源

- 如需有关可能安装模式的详细信息，请参阅 Operator 组成员资格。

7.2.7.2. bundle-upgrade

run bundle-upgrade 子命令升级之前使用 Operator Lifecycle Manager（OLM）以捆绑包格式安装的 Operator。

表 7.10. run bundle-upgrade 标记

标记	描述
--timeout <duration>	升级超时。默认值为 **2m0s**。
--kubeconfig (string)	用于 CLI 请求的 **kubeconfig** 文件的路径。
n、**--namespace**（字符串）	如果存在，代表在其中运行 CLI 请求的命名空间。
-h，**--help**	**run bundle** 子命令的帮助输出。

7.2.8. scorecard

operator-sdk scorecard 命令运行 scorecard 工具来验证 Operator 捆绑包并提供改进建议。该命令使用一个参数，可以是捆绑包镜像，也可以是包含清单和元数据的目录。如果参数包含镜像标签，则镜像必须远程存在。

表 7.11. scorecard 标记

标记	描述
-c, **--config**（字符串）	scorecard 配置文件的路径。默认路径为 **bundle/tests/scorecard/config.yaml**。
-h，**--help**	**scorecard** 命令的帮助输出。
--kubeconfig (string)	**kubeconfig** 文件的路径。

标记	描述
-L、--list	列出哪些测试可以运行。
-n、--namespace（字符串）	运行测试镜像的命名空间。
-o, --output（字符串）	结果的输出格式。可用值为 **text** （默认值）和 **json**。
-l、--selector（字符串）	标识选择器以确定要运行哪个测试。
-s、--service-account（字符串）	用于测试的服务帐户。默认值为 **default**。
-x、--skip-cleanup	运行测试后禁用资源清理。
-w、--wait-time <duration>	等待测试完成的时间，如 **35s**。默认值为 **30s**。

其他资源

- 如需有关运行scorecard 工具的详细信息，请参阅使用 scorecard 工具验证 Operator。