OpenShift Container Platform 4.7

CLI 工具

如何使用 OpenShift Container Platform 的命令行工具

Last Updated: 2023-03-24

OpenShift Container Platform 4.7 CLI 工具

如何使用 OpenShift Container Platform 的命令行工具

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

法律通告

摘要

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 CONTAINER PLATFORM CLI 工具概述

用户在操作 OpenShift Container Platform 时执行一系列操作，例如：

- 管理集群
- 构建、部署和管理应用程序
- 管理部署过程
- 开发 Operator
- 创建和维护 Operator 目录

OpenShift Container Platform 提供了一组命令行界面 (CLI) 工具，通过允许用户从终端执行各种管理和开发操作来简化这些任务。这些工具提供简单的命令来管理应用，并与系统的每个组件交互。

1.1. CLI 工具列表

OpenShift Container Platform 中提供了以下一组 CLI 工具：

- OpenShift CLI (oc)：这是 OpenShift Container Platform 用户最常用的 CLI 工具。它帮助集群管理员和开发人员使用终端在 OpenShift Container Platform 间执行端到端操作。与 Web 控制台不同，它允许用户使用命令脚本直接处理项目源代码。
- 开发人员 CLI(odo) ： **odo** CLI 工具使开发人员能够专注于通过处理与 Kubernetes 和 OpenShift Container Platform 相关的复杂概念来在 OpenShift Container Platform 上创建和维护应用程序的主要目标。它可帮助开发人员从终端在终端中编写、构建和调试应用程序，而无需管理集群。
- Helm CLI: Helm 是一个 Kubernetes 应用程序的软件包管理器，它允许定义、安装和升级打包为 Helm chart 的应用程序。Helm CLI 可帮助用户从终端使用简单命令将应用程序和服务部署到 OpenShift Container Platform 集群。
- Knative CLI(kn): Knative(**kn**)CLI 工具提供简单直观的终端命令，可用于与 OpenShift Serverless 组件交互，如 Knative Serving 和 Eventing。
- Pipelines CLI(tkn) ：OpenShift Pipelines 是 OpenShift Container Platform 中的持续集成和持续交付 (CI/CD) 解决方案，内部使用 Tekton。**tkn** CLI 工具提供简单直观的命令，以便使用终端与 OpenShift Pipelines 进行交互。
- opm CLI ： **opm** CLI 工具可帮助 Operator 开发人员和集群管理员从终端创建和维护 Operator 目录。
- Operator SDK ：Operator SDK 是 Operator Framework 的一个组件，它提供了一个 CLI 工具，可供 Operator 开发人员用于从终端构建、测试和部署 Operator。它简化了 Kubernetes 原生应用程序的构建流程，这些应用程序需要深入掌握特定于应用程序的操作知识。

第 2 章 OPENSHIFT CLI (OC)

2.1. OPENSHIFT CLI 入门

2.1.1. 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：

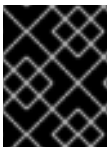
- 直接使用项目源代码
- 编写 OpenShift Container Platform 操作脚本
- 在管理项目时，受带宽资源的限制，Web 控制台无法使用

2.1.2. 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**）。

2.1.2.1. 通过下载二进制文件安装 OpenShift CLI

您可以安装 OpenShift CLI(**oc**)来使用命令行界面与 OpenShift Container Platform 进行交互。您可以在 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**。

2.1.2.1.1. 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. 导航到红帽客户门户网站上的 OpenShift Container Platform 下载页面。
2. 在 **Version** 下拉菜单中选择相应的版本。
3. 单击 **OpenShift v4.7 Linux 客户端**条目旁边的 **Download Now**，再保存文件。
4. 解包存档：

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

5. 将 **oc** 二进制文件放到 **PATH 中的目录中**。
   要查看您的 **PATH**，请执行以下命令：

```
$ echo $PATH
```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

```
$ oc <command>
```

2.1.2.1.2. 在 Windows 上安装 OpenShift CLI

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

流程

1. 导航到红帽客户门户网站上的 OpenShift Container Platform 下载页面。
2. 在 **Version** 下拉菜单中选择相应的版本。
3. 单击 **OpenShift v4.7 Windows 客户端**条目旁边的 **Download Now**，再保存文件。
4. 使用 ZIP 程序解压存档。
5. 将 **oc** 二进制文件移到 **PATH 中的目录中**。
   要查看您的 **PATH**，请打开命令提示并执行以下命令：

   ```
   C:\> path
   ```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

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

2.1.2.1.3. 在 macOS 上安装 OpenShift CLI

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

流程

1. 导航到红帽客户门户网站上的 OpenShift Container Platform 下载页面。
2. 在 **Version** 下拉菜单中选择相应的版本。
3. 单击 **OpenShift v4.7 MacOSX 客户端**条目旁边的 **Download Now**，再保存文件。
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**，请打开终端并执行以下命令：

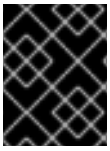
   ```
   $ echo $PATH
   ```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

```
$ oc <command>
```

2.1.2.2. 使用 Web 控制台安装 OpenShift CLI

您可以安装 OpenShift CLI(**oc**)来通过 Web 控制台与 OpenShift Container Platform 进行交互。您可以在 Linux、Windows 或 macOS 上安装 **oc**。



重要

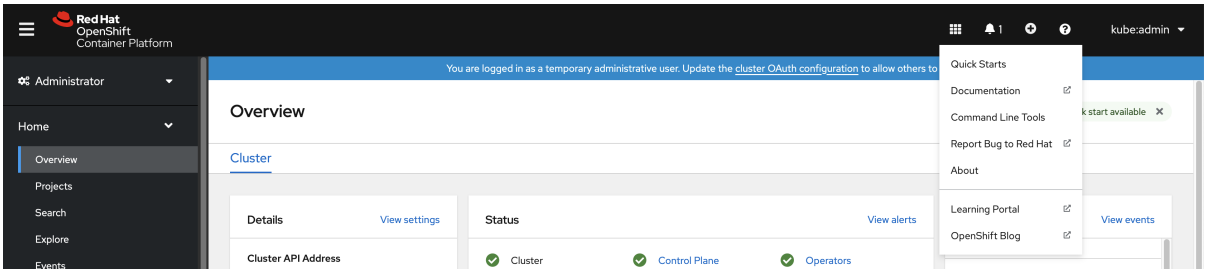
如果安装了旧版本的 **oc**，则无法使用 OpenShift Container Platform 4.7 中的所有命令。下载并安装新版本的 **oc**。

2.1.2.2.1. 使用 Web 控制台在 Linux 上安装 OpenShift CLI

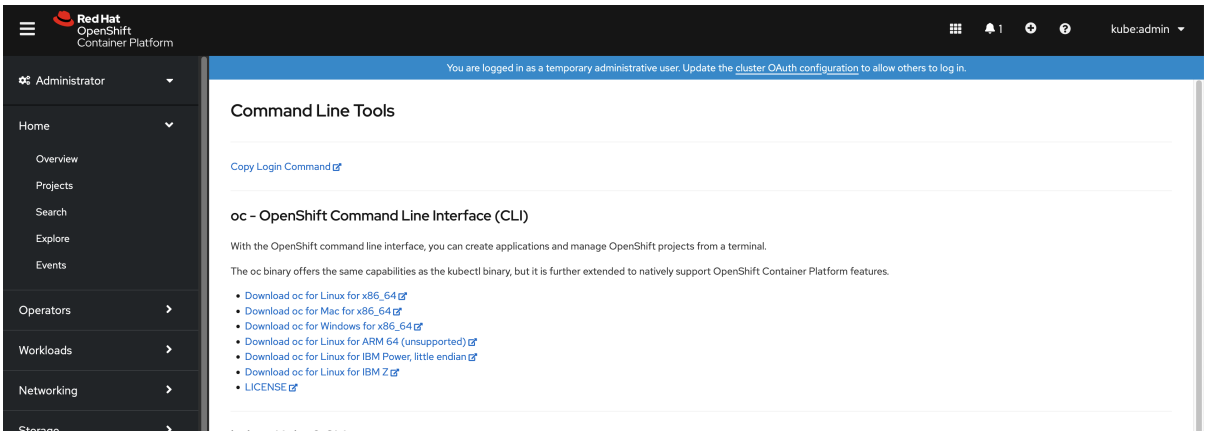
您可以按照以下流程在 Linux 上安装 OpenShift CLI(**oc**)二进制文件。

流程

1. 从 Web 控制台，单击 **?**。



2. 单击 **Command Line Tools**。



3. 为您的 Linux 平台选择适当的 **oc** 二进制文件，然后点 **Download oc for Linux**。
4. 保存该文件。
5. 解包存档。

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

6. 将 **oc** 二进制文件移到 **PATH** 中的目录中。
要查看您的 **PATH**，请执行以下命令：

```
$ echo $PATH
```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

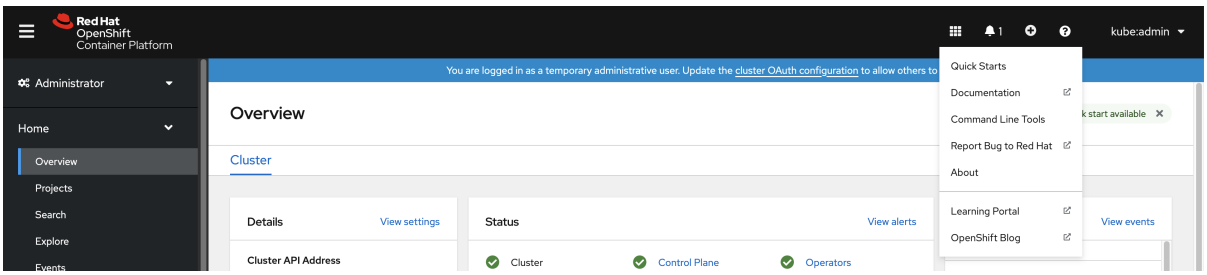
```
$ oc <command>
```

2.1.2.2.2. 使用 Web 控制台在 Windows 上安装 OpenShift CLI

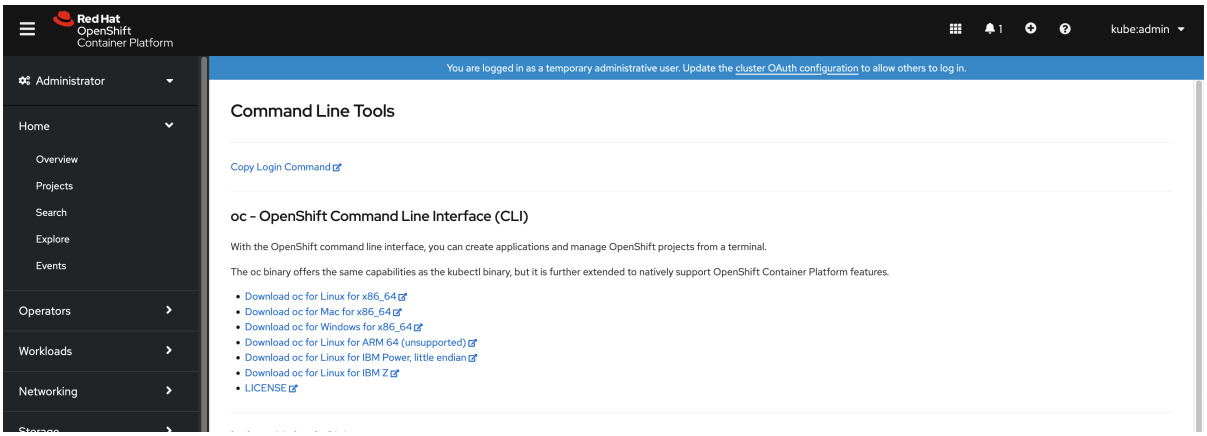
您可以按照以下流程在 Winndows 上安装 OpenShift CLI(**oc**)二进制文件。

流程

1. 从 Web 控制台，单击 **?**。



2. 单击 **Command Line Tools**。



3. 为 Windows 平台选择 **oc** 二进制文件，然后单击 **Download oc for Windows for x86_64**。
4. 保存该文件。
5. 使用 ZIP 程序解压存档。
6. 将 **oc** 二进制文件移到 **PATH 中的目录中**。
   要查看您的 **PATH**，请打开命令提示并执行以下命令：

```
C:\> path
```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

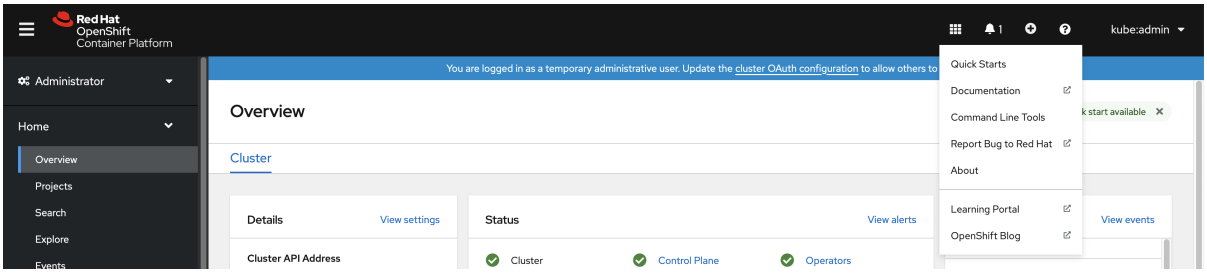
```
C:\> oc <command>
```

2.1.2.2.3. 使用 Web 控制台在 macOS 上安装 OpenShift CLI

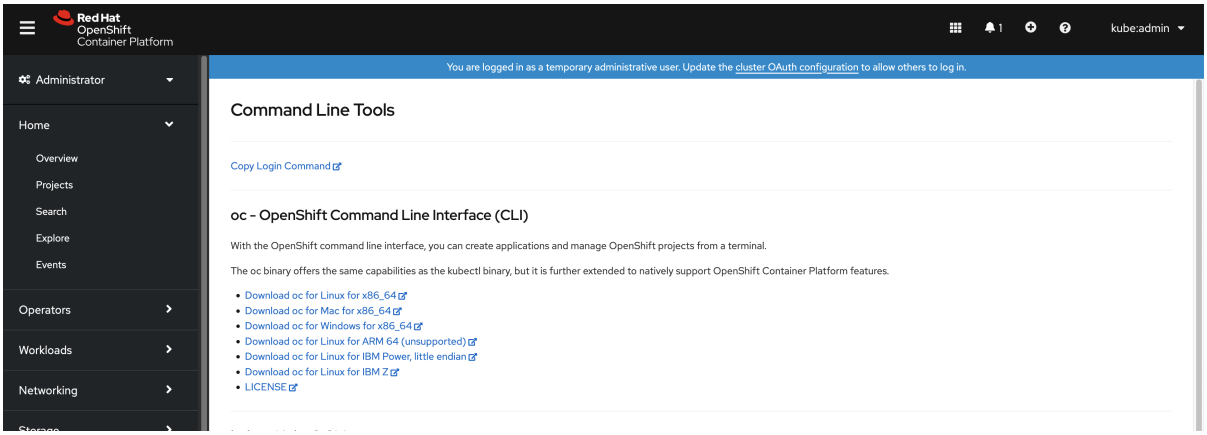
您可以按照以下流程在 macOS 上安装 OpenShift CLI(**oc**)二进制文件。

流程

1. 从 Web 控制台，单击 **?**。



2. 单击 **Command Line Tools**。



3. 为 macOS 平台选择 **oc** 二进制文件，然后单击 **Download oc for Mac for x86_64**。
4. 保存该文件。
5. 解包和解压存档。
6. 将 **oc** 二进制文件移到 PATH 的目录中。
   要查看您的 **PATH**，请打开终端并执行以下命令：

```
$ echo $PATH
```

安装 OpenShift CLI 后，可以使用 **oc** 命令：

```
$ oc <command>
```

2.1.2.3. 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>
```

2.1.2.4. 使用 Homebrew 安装 OpenShift CLI

对于 macOS，您可以使用 Homebrew 软件包管理器安装 OpenShift CLI(**oc**)。

先决条件

- 已安装 Homebrew(**brew**)。

流程

- 运行以下命令来安装 openshift-cli 软件包：

```
$ brew install openshift-cli
```

2.1.3. 登录到 OpenShift CLI

您可以登录到 OpenShift CLI（**oc**）以访问和管理集群。

先决条件

- 有访问 OpenShift Container Platform 集群的权限。

- 已安装 OpenShift CLI（**oc**）。

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。

只有在使用 HTTPS 传输时才会发送身份验证标头。

流程

1. 输入 **oc login** 命令并传递用户名：

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

2. 提示时，请输入所需信息：

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
Password: (3)
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) 输入用户密码。

注意

如果登录到 web 控制台，您可以生成包含令牌和服务器信息的 **oc login** 命令。您可以使用命令来登录 OpenShift Container Platform CLI，而无需交互式的提示。要生成 命令，请从 web 控制台右上角的用户名下拉菜单中选择 **Copy login command**。

您现在可以创建项目或执行其他命令来管理集群。

2.1.4. 使用 OpenShift CLI

参阅以下部分以了解如何使用 CLI 完成常见任务。

2.1.4.1. 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".
```

2.1.4.2. 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

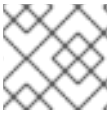
```
--> Found image 40de956 (9 days old) in imagestream "openshift/php" under tag "7.2" for "php"

...

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

2.1.4.3. 查看 pod

使用**oc get pods**命令查看当前项目的 pod。



注意

当您在 pod 中运行 **oc** 且没有指定命名空间时，默认使用 pod 的命名空间。

```
$ 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>
```

2.1.4.4. 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
```

2.1.4.5. 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".
```

2.1.4.6. 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.
```

2.1.4.7. 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
...
```

2.1.5. 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

...
```

2.1.6. 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。

2.2. 配置 OPENSHIFT CLI

2.2.1. 启用 tab 自动完成功能

您可以为 Bash 或 Zsh shell 启用 tab 自动完成功能。

2.2.1.1. 为 Bash 启用 tab 自动完成功能

安装 OpenShift CLI(**oc**)后，您可以启用 tab 自动完成功能，以便在按 Tab 键时自动完成 **oc** 命令或建议选项。以下过程为 Bash shell 启用 tab 自动完成功能。

先决条件

- 已安装 OpenShift CLI (**oc**)。
- 已安装软件包 **bash-completion**。

流程

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。

2.2.1.2. 为 Zsh 启用 tab 自动完成

安装 OpenShift CLI(**oc**)后，您可以启用 tab 自动完成功能，以便在按 Tab 键时自动完成 **oc** 命令或建议选项。以下流程为 Zsh shell 启用 tab 自动完成功能。

先决条件

- 已安装 OpenShift CLI (**oc**)。

流程

- 要将 **oc** 的 tab 自动完成添加到您的 **.zshrc** 文件中，请运行以下命令：

```
$ cat >>~/.zshrc<<EOF
if [ $commands[oc] ]; then
  source <(oc completion zsh)
  compdef _oc oc
fi
EOF
```

开新终端时 tab 自动完成功能将被启用。

2.3. 管理 CLI 配置集

CLI 配置文件允许您配置不同的配置文件或上下文，以用于 CLI 工具概述。上下文由与 *nickname* 关联的用户身份验证 和 OpenShift Container Platform 服务器信息组成。

2.3.1. 关于 CLI 配置集间的切换

通过上下文，您可以在多个 OpenShift Container Platform 服务器或使用 CLI 操作时轻松地切换多个用户。nicknames 通过提供对上下文、用户凭证和集群详情的简短参考来更轻松地管理 CLI 配置。第一次使用 CLI 登录后，OpenShift Container Platform 会创建一个 **~/.kube/config** 文件（如果不存在）。随着更多身份验证和连接详情被提供给 CLI，可以在 **oc login** 操作或手动配置 CLI 配置集过程中自动提供，更新的信息会存储在配置文件中：

CLI 配置文件

```
apiVersion: v1
clusters: (1)
- cluster:
    insecure-skip-tls-verify: true
    server: https://openshift1.example.com:8443
  name: openshift1.example.com:8443
- cluster:
    insecure-skip-tls-verify: true
```

```
    server: https://openshift2.example.com:8443
  name: openshift2.example.com:8443
contexts: 2
- context:
    cluster: openshift1.example.com:8443
    namespace: alice-project
    user: alice/openshift1.example.com:8443
  name: alice-project/openshift1.example.com:8443/alice
- context:
    cluster: openshift1.example.com:8443
    namespace: joe-project
    user: alice/openshift1.example.com:8443
  name: joe-project/openshift1/alice
current-context: joe-project/openshift1.example.com:8443/alice 3
kind: Config
preferences: {}
users: 4
- name: alice/openshift1.example.com:8443
  user:
    token: xZHd2piv5_9vQrg-SKXRJ2Dsl9SceNJdhNTljEKTb8k
```

1 **clusters** 部分定义 OpenShift Container Platform 集群的连接详情，包括其 master 服务器的地址。在本例中，一个集群的别名为 **openshift1.example.com:8443**，另一个别名是 **openshift2.example.com:8443**。

2 这个 **contexts** 项定义了两个上下文：一个别名是 **alice-project/openshift1.example.com:8443/alice**，使用 **alice-project** 项目，**openshift1.example.com:8443** 集群以及 **alice** 用户，另外一个别名是 **joe-project/openshift1.example.com:8443/alice**，使用 **joe-project** 项目，**openshift1.example.com:8443** 集群以及 **alice** 用户。

3 **current-context** 参数显示 **joe-project/openshift1.example.com:8443/alice** 上下文当前正在使用中，允许 **alice** 用户在 **openshift1.example.com:8443** 集群上的 **joe-project** 项目中工作。

4 **users** 部分定义用户凭据。在本例中，用户别名 **alice/openshift1.example.com:8443** 使用访问令牌。

CLI 可以支持多个在运行时加载的配置文件，并合并在一起，以及从命令行指定的覆盖选项。登录后，您可以使用 **oc status** 或 **oc project** 命令验证您当前的环境：

验证当前工作环境

```
$ oc status
```

输出示例

```
oc status
In project Joe's Project (joe-project)

service database (172.30.43.12:5434 -> 3306)
  database deploys docker.io/openshift/mysql-55-centos7:latest
    #1 deployed 25 minutes ago - 1 pod

service frontend (172.30.159.137:5432 -> 8080)
```

```
  frontend deploys origin-ruby-sample:latest <-
    builds https://github.com/openshift/ruby-hello-world with joe-project/ruby-20-centos7:latest
    #1 deployed 22 minutes ago - 2 pods

To see more information about a service or deployment, use 'oc describe service <name>' or 'oc
describe dc <name>'.
You can use 'oc get all' to see lists of each of the types described in this example.
```

列出当前项目

```
$ oc project
```

输出示例

```
Using project "joe-project" from context named "joe-project/openshift1.example.com:8443/alice" on
server "https://openshift1.example.com:8443".
```

您可以再次运行 **oc login** 命令，并在互动过程中提供所需的信息，使用用户凭证和集群详情的任何其他组合登录。基于提供的信息构建上下文（如果尚不存在）。如果您已经登录，并希望切换到当前用户已有权访问的另一个项目，请使用 **oc project** 命令并输入项目名称：

```
$ oc project alice-project
```

输出示例

```
Now using project "alice-project" on server "https://openshift1.example.com:8443".
```

在任何时候，您可以使用 **oc config view** 命令查看当前的 CLI 配置，如输出中所示。其他 CLI 配置命令也可用于更高级的用法。

注意

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

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

2.3.2. 手动配置 CLI 配置集

注意

本节介绍 CLI 配置的更多高级用法。在大多数情况下，您可以使用 **oc login** 和 **oc project** 命令登录并在上下文和项目间切换。

如果要手动配置 CLI 配置文件，您可以使用 **oc config** 命令，而不是直接修改这些文件。**oc config** 命令包括很多有用的子命令来实现这一目的：

表 2.1. CLI 配置子命令

子命令	使用方法
set-cluster	在 CLI 配置文件中设置集群条目。如果引用的 cluster nickname 已存在，则指定的信息将合并到其中。 $ oc config set-cluster <cluster_nickname> [--server=<master_ip_or_fqdn>] [--certificate-authority=<path/to/certificate/authority>] [--api-version=<apiversion>] [--insecure-skip-tls-verify=true]
set-context	在 CLI 配置文件中设置上下文条目。如果引用的上下文 nickname 已存在，则指定的信息将合并在. $ oc config set-context <context_nickname> [--cluster=<cluster_nickname>] [--user=<user_nickname>] [--namespace=<namespace>]
use-context	使用指定上下文 nickname 设置当前上下文。 $ oc config use-context <context_nickname>
set	在 CLI 配置文件中设置单个值。 $ oc config set <property_name> <property_value> **<property_name>** 是一个以点分隔的名称，每个令牌代表属性名称或映射键。**<property_value>** 是要设置的新值。
unset	在 CLI 配置文件中取消设置单个值。 $ oc config unset <property_name> **<property_name>** 是一个以点分隔的名称，每个令牌代表属性名称或映射键。

view	显示当前正在使用的合并 CLI 配置。
	
	$ oc config view
	
	显示指定 CLI 配置文件的结果。
	
	$ oc config view --config=<specific_filename>

用法示例

- 以使用访问令牌的用户身份登录。**alice** 用户使用此令牌：

```
$ oc login https://openshift1.example.com --
token=ns7yVhuRNpDM9cgzfhhxQ7bM5s7N2ZVrkZepSRf4LC0
```

- 查看自动创建的集群条目：

-

```
$ oc config view
```

输出示例

```
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: https://openshift1.example.com
  name: openshift1-example-com
contexts:
- context:
    cluster: openshift1-example-com
    namespace: default
    user: alice/openshift1-example-com
  name: default/openshift1-example-com/alice
current-context: default/openshift1-example-com/alice
kind: Config
preferences: {}
users:
- name: alice/openshift1.example.com
  user:
    token: ns7yVhuRNpDM9cgzfhhxQ7bM5s7N2ZVrkZepSRf4LC0
```

- 更新当前上下文以便用户登录到所需的命名空间：

```
$ oc config set-context `oc config current-context` --namespace=<project_name>
```

- 检查当前上下文，确认是否实施了更改：

```
$ oc whoami -c
```

所有后续 CLI 操作都使用新的上下文，除非通过覆盖 CLI 选项或直至上下文切换为止。

2.3.3. 载入和合并规则

您可以在为 CLI 配置发出加载和合并顺序的 CLI 操作时遵循这些规则：

- 使用以下层次结构和合并规则从工作站检索 CLI 配置文件：
  - 如果设置了 **--config** 选项，则只加载该文件。标志会被设置一次，且不会发生合并。
  - 如果设置了 **$KUBECONFIG** 环境变量，则会使用它。变量可以是路径列表，如果将路径合并在一起。修改值后，会在定义该节的文件中对其进行修改。创建值时，会在存在的第一个文件中创建它。如果链中不存在任何文件，则会在列表中创建最后一个文件。
  - 否则，将使用 ***~/.kube/config*** 文件，且不会发生合并。
- 使用的上下文根据以下流程中的第一个匹配项决定：
  - **--context** 选项的值。
  - CLI 配置文件中的 **current-context** 值。
  - 此阶段允许一个空值。

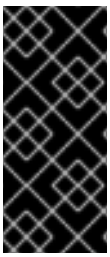
- 要使用的用户和集群是决定的。此时，您可能也可能没有上下文；它们基于以下流程中的第一个匹配项构建，该流中为用户运行一次，一次用于集群：
  - 用于用户名的 **--user** 的值，以及集群名称的 **--cluster** 选项。
  - 如果存在 **--context** 选项，则使用上下文的值。
  - 此阶段允许一个空值。
- 要使用的实际集群信息决定。此时，您可能没有集群信息。集群信息的每个信息根据以下流程中的第一个匹配项构建：
  - 以下命令行选项中的任何值：
    - **--server**,
    - **--api-version**
    - **--certificate-authority**
    - **--insecure-skip-tls-verify**
  - 如果集群信息和属性的值存在，则使用它。
  - 如果您没有服务器位置，则出现错误。
- 要使用的实际用户信息是确定的。用户使用与集群相同的规则构建，但每个用户只能有一个身份验证技术；冲突的技术会导致操作失败。命令行选项优先于配置文件值。有效命令行选项包括：
  - **--auth-path**
  - **--client-certificate**
  - **--client-key**
  - **--token**
- 对于仍缺失的任何信息，将使用默认值，并提示提供其他信息。

2.4. 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。

2.4.1. 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。

2.4.2. 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。

2.5. OPENSHIFT CLI 开发人员命令

2.5.1. 基本CLI命令

2.5.1.1. explain

显示特定资源的文档。

示例：显示 pod 的文档

```
$ oc explain pods
```

2.5.1.2. 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 -u user1
```

2.5.1.3. 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
```

2.5.1.4. 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
```

2.5.1.5. 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
```

2.5.1.6. 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
```

2.5.1.7. 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
```

2.5.2. 构建和部署CLI命令

2.5.2.1. 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
```

2.5.2.2. 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
```

2.5.2.3. 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
```

2.5.2.4. 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
```

2.5.2.5. 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
```

2.5.2.6. 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
```

2.5.2.7. 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
```

2.5.3. 应用程序管理CLI命令

2.5.3.1. 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-
```

2.5.3.2. 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
```

2.5.3.3. 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
```

2.5.3.4. 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
```

2.5.3.5. 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
```

2.5.3.6. 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
```

2.5.3.7. 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
```

2.5.3.8. 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
```

2.5.3.9. 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
```

2.5.3.10. 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
```

2.5.3.11. 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
```

2.5.3.12. 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
```

2.5.3.13. 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
```

2.5.3.14. 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
```

2.5.4. 调试CLI命令

2.5.4.1. 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
```

2.5.4.2. 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/.
```

2.5.4.3. 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
```

2.5.4.4. 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
```

2.5.4.5. 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
```

2.5.4.6. 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
```

2.5.4.7. 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/
```

2.5.4.8. 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
```

2.5.4.9. 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/
```

2.5.4.10. 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
```

2.5.4.11. 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
```

2.5.5. 高级开发人员CLI命令

2.5.5.1. 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
```

2.5.5.2. 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
```

2.5.5.3. 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
```

2.5.5.4. cluster-info

显示 master 和集群服务的地址。

示例：显示集群信息

```
$ oc cluster-info
```

2.5.5.5. 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=-
```

2.5.5.6. 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
```

2.5.5.7. 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
```

2.5.5.8. 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
```

2.5.5.9. 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

如果需要对自定义资源定义进行补丁，则必须在命令中包含 **--type merge** 或 **--type json** 选项。

2.5.5.10. 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
```

2.5.5.11. 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 -
```

2.5.5.12. 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
```

2.5.5.13. 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
```

2.5.6. 设置CLI命令

2.5.6.1. 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
```

2.5.6.2. 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
```

2.5.6.3. logout

退出当前会话。

示例：结束当前会话

```
$ oc logout
```

2.5.6.4. 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
```

2.5.7. 其他开发人员CLI命令

2.5.7.1. 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
```

2.5.7.2. plugin

列出用户**PATH**中的可用插件。

示例：列出可用的插件

```
$ oc plugin list
```

2.5.7.3. version

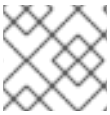
显示**oc**客户端和服务器版本。

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。

2.6. OPENSHIFT CLI 管理员命令



注意

您必须具有 **cluster-admin** 或同等权限才能使用这些管理员命令。

2.6.1. 集群管理CLI命令

2.6.1.1. 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
```

2.6.1.2. 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
```

2.6.1.3. 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
```

2.6.2. 集群管理 CLI 命令

2.6.2.1. 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
```

2.6.2.2. drain

排空节点以准备进行维护。

示例：排空node1

```
$ oc adm drain node1
```

2.6.2.3. 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
```

2.6.2.4. 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-
```

2.6.2.5. uncordon

将节点标记为可调度。

示例：将node1标记为可调度

```
$ oc adm uncordon node1
```

2.6.3. 安全和策略CLI命令

2.6.3.1. 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
```

2.6.3.2. groups

管理集群中的组。

示例：创建一个新组

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

2.6.3.3. 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'
```

2.6.3.4. 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
```

2.6.3.5. 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
```

2.6.4. 维护CLI命令

2.6.4.1. 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
```

2.6.4.2. 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
```

2.6.5. 配置CLI命令

2.6.5.1. 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
```

2.6.5.2. 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
```

2.6.5.3. 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
```

2.6.5.4. 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
```

2.6.5.5. 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
```

2.6.6. 其他管理员CLI命令

2.6.6.1. 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
```

2.6.6.2. 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
```

2.6.6.3. 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
```

2.6.6.4. 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
```

2.6.6.5. 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
```

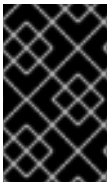
2.7. OC 和 KUBECTL 命令的使用方法

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。

2.7.1. 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。



重要

如果安装了旧版本的 **oc** 二进制文件，则无法使用 OpenShift Container Platform 4.7 中的所有命令。如果要使用最新的功能，您必须下载并安装与 OpenShift Container Platform 服务器版本对应的 **oc** 二进制文件的最新版本。

非安全 API 更改至少涉及两个次发行版本（例如，4.1 到 4.2 到 4.3）来更新旧的 **oc** 二进制文件。使用新功能可能需要较新的 **oc** 二进制文件。一个 4.3 服务器可能会带有版本 4.2 **oc** 二进制文件无法使用的功能，而一个 4.3 **oc** 二进制文件可能会带有 4.2 服务器不支持的功能。

表 2.2. 兼容性列表

	X.Y (**oc** Client)	X.Y+N [a] (**oc** Client)
X.Y (Server)	1	3
X.Y+N [a] (Server)	2	1

[a] 其中 N 是一个大于或等于 1 的数值。

1 完全兼容。

2 **oc** 客户端可能无法访问服务器的功能。

3 **oc** 客户端可能会提供与要访问的服务器不兼任的选项和功能。

2.7.2. 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 3 章 开发人员 CLI (ODO)

3.1. ODO 发行注记

3.1.1. odo 版本 2.5.0 中的显著变化和改进

- 使用 **adler32** 哈希来为每个组件创建唯一路由
- 支持 devfile 中的其他字段来分配资源：
  - cpuRequest
  - cpuLimit
  - memoryRequest
  - memoryLimit
- 在 **odo delete** 命令中添加 **--deploy** 标志，删除使用 **odo deploy** 命令部署的组件：

```
$ odo delete --deploy
```

- 在 **odo link** 命令中添加映射支持
- 支持临时卷，使用 **volume** 组件中的 **ephemeral** 字段。
- 在请求遥测选择时，将默认回答设置为 **yes**
- 通过将额外的遥测数据发送到 devfile registry 来提高指标
- 将 bootstrap 镜像更新至 **registry.access.redhat.com/ocp-tools-4/odo-init-container-rhel8:1.1.11**
- 上游存储库位于 https://github.com/redhat-developer/odo

3.1.2. 程序错误修复

- 在以前的版本中，如果 **.odo/env** 文件不存在，**odo deploy** 将失败。现在，如果需要，请创建 **.odo/env** 文件。
- 在以前的版本中，如果断开与集群的连接，使用 **odo create** 命令创建交互式组件会失败。此问题已在最新的发行版本中解决。

3.1.3. 获取支持

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 **OpenShift Developer Tools and Services** 作为产品类型，**odo** 作为组件。

请在问题描述中提供尽可能多的细节。

对于文档

如果您发现了错误或有改进文档的建议，请提交一个与相关文档组件的 JIRA 问题。

3.2. 了解 ODO

Red Hat OpenShift Developer CLI(**odo**)是在 OpenShift Container Platform 和 Kubernetes 上创建应用程序的工具。使用 **odo**，您可以在 Kubernetes 集群中开发、测试、调试和部署基于微服务的应用，而无需深入了解平台。

odo 遵循 *创建和推送* 工作流。作为用户，当您 *创建*时，信息（或清单）存储在配置文件中。*推送* 时，会在 Kubernetes 集群中创建对应的资源。所有这些配置都存储在 Kubernetes API 中，以实现无缝访问和功能。

odo 使用 *service* 和 *link* 命令将组件和服务链接在一起。**odo** 通过使用集群中的 Kubernetes Operator 创建和部署服务来实现这个目标。可使用 Operator Hub 上可用的任何 Operator 创建服务。在链接服务后，**odo** 会将服务配置注入组件。然后，应用程序就可以使用此配置与 Operator 支持的服务通信。

3.2.1. odo 的主要功能

odo 的设计目的是为 Kubernetes 的开发人员提供一个友好的 Kubernetes 接口，能够：

- 通过创建新清单或使用现有清单，在 Kubernetes 集群上快速部署应用程序
- 使用命令轻松创建和更新清单，而无需理解和维护 Kubernetes 配置文件
- 提供对在 Kubernetes 集群上运行的应用程序的安全访问
- 为 Kubernetes 集群上的应用程序添加和删除额外存储
- 创建 Operator 支持的服务，并将应用程序链接到它们
- 在作为 **odo** 组件部署的多个微服务间创建一个链接
- 在 IDE 中使用 **odo** 进行远程调试应用程序
- 使用 **odo** 轻松测试 Kubernetes 上部署的应用程序

3.2.2. odo 核心概念

odo 将 Kubernetes 概念抽象化为开发人员熟悉的术语：

Application（应用程序）

使用 云原生方法 开发的典型应用，用于执行特定的任务。
*应用程序*示例包括在线视频流、在线购物和酒店预订系统。

组件

一组可单独运行和部署的 Kubernetes 资源。云原生应用是一系列小、独立、松散耦合的*组件*。
*组件*示例包括 API 后端、Web 界面和支付后端。

project

包含源代码、测试和库的单一单元。

Context

包含单一组件的源代码、测试、库和 **odo** 配置文件的目录。

URL

公开组件的机制可从集群外部访问。

存储

集群中的持久性存储。它会在重启和组件重建过程中保留数据。

服务

为组件提供额外的功能的外部应用程序。
服务 示例包括 PostgreSQL、MySQL、Redis 和 RabbitMQ。

在 **odo** 中，服务从 OpenShift Service Catalog 置备，且必须在集群中启用。

devfile

用于定义容器化开发环境的开放式标准，使开发人员工具可以简化和加速工作流。如需更多信息，请参阅文档 https://devfile.io。
您可以连接到公开可用的 *devfile* registry，也可以安装安全 registry。

3.2.3. 列出 odo 中的组件

odo 使用可移植 *devfile* 格式来描述组件及其相关 URL、存储和服务。**odo** 可以连接到各种 devfile registry，以下载用于不同语言和框架的 devfile。有关如何管理 **odo registry** 用来检索 devfile 信息的更多信息，请参阅 **odo** registry 命令的文档。

您可以使用 **odo catalog list components** 命令列出不同 registry 的所有 *devfile*。

流程

1. 使用 **odo** 登陆到集群：

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

2. 列出可用的 **odo** 组件：

```
$ odo catalog list components
```

输出示例

```
Odo Devfile Components:
NAME                        DESCRIPTION                                     REGISTRY
dotnet50                    Stack with .NET 5.0
DefaultDevfileRegistry
dotnet60                    Stack with .NET 6.0
DefaultDevfileRegistry
dotnetcore31                Stack with .NET Core 3.1
DefaultDevfileRegistry
go                          Stack with the latest Go version
DefaultDevfileRegistry
java-maven                  Upstream Maven and OpenJDK 11
DefaultDevfileRegistry
java-openliberty            Java application Maven-built stack using the Open Liberty ru...
DefaultDevfileRegistry
java-openliberty-gradle     Java application Gradle-built stack using the Open Liberty r...
DefaultDevfileRegistry
java-quarkus                Quarkus with Java
DefaultDevfileRegistry
java-springboot             Spring Boot® using Java
```

```
DefaultDevfileRegistry
java-vertx                  Upstream Vert.x using Java
DefaultDevfileRegistry
java-websphereliberty       Java application Maven-built stack using the WebSphere
Liber...    DefaultDevfileRegistry
java-websphereliberty-gradle    Java application Gradle-built stack using the WebSphere
Libe...    DefaultDevfileRegistry
java-wildfly                Upstream WildFly
DefaultDevfileRegistry
java-wildfly-bootable-jar       Java stack with WildFly in bootable Jar mode, OpenJDK 11
and...    DefaultDevfileRegistry
nodejs                      Stack with Node.js 14
DefaultDevfileRegistry
nodejs-angular              Stack with Angular 12
DefaultDevfileRegistry
nodejs-nextjs               Stack with Next.js 11
DefaultDevfileRegistry
nodejs-nuxtjs               Stack with Nuxt.js 2
DefaultDevfileRegistry
nodejs-react                Stack with React 17
DefaultDevfileRegistry
nodejs-svelte               Stack with Svelte 3
DefaultDevfileRegistry
nodejs-vue                  Stack with Vue 3
DefaultDevfileRegistry
php-laravel                 Stack with Laravel 8
DefaultDevfileRegistry
python                      Python Stack with Python 3.7
DefaultDevfileRegistry
python-django               Python3.7 with Django
DefaultDevfileRegistry
```

3.2.4. odo 中的 Telemetry

odo 会收集有关如何使用它的信息，包括操作系统、RAM、CPU、内核数量、**odo** 版本、错误、成功/失败以及 **odo** 命令完成所需的时间。

您可以使用 **odo preference** 命令修改遥测同意：

- **odo preference set ConsentTelemetry true** 代表同意遥测。
- **odo preference unset ConsentTelemetry** 会禁用 Telemetry。
- **odo preference view** 显示当前的首选项。

3.3. 安装 ODO

您可以通过下载二进制文件，在 Linux、Windows 或 macOS 上安装 **odo** CLI。您还可以安装 OpenShift VS Code 扩展，它使用 **odo** 和 **oc** 二进制文件与 OpenShift Container Platform 集群交互。对于 Red Hat Enterprise Linux(RHEL)，您可以使用 RPM 安装 **odo** CLI。

注意

目前，**odo** 不支持在限制的网络环境中安装。

3.3.1. 在 Linux 中安装 odo

odo CLI 可作为二进制文件下载，并为多个操作系统和架构提供 tarball，其中包括：

操作系统	二进制	Tarball
Linux	odo-linux-amd64	odo-linux-amd64.tar.gz
Linux on IBM Power	odo-linux-ppc64le	odo-linux-ppc64le.tar.gz
Linux on IBM Z 和 LinuxONE	odo-linux-s390x	odo-linux-s390x.tar.gz

流程

1. 进入内容网关，再下载适用于您的操作系统和架构的适当文件。
   - 如果下载二进制文件，请将其重命名为 **odo** ：

     ```
     $ curl -L https://developers.redhat.com/content-gateway/rest/mirror/pub/openshift-v4/clients/odo/latest/odo-linux-amd64 -o odo
     ```

   - 如果下载 tarball，解压二进制文件：

     ```
     $ curl -L https://developers.redhat.com/content-gateway/rest/mirror/pub/openshift-v4/clients/odo/latest/odo-linux-amd64.tar.gz -o odo.tar.gz
     $ tar xvzf odo.tar.gz
     ```

2. 更改二进制的权限：

   ```
   $ chmod +x <filename>
   ```

3. 将 **odo** 二进制文件放到 **PATH** 中的目录中。
   要查看您的 **PATH**，请执行以下命令：

   ```
   $ echo $PATH
   ```

4. 验证您的系统中现在可用的 **odo**：

   ```
   $ odo version
   ```

3.3.2. 在 Windows 中安装 odo

用于 Windows 的 **odo** CLI 可作为二进制文件下载，并作为一个存档。

操作系统	二进制	Tarball
Windows	odo-windows-amd64.exe	odo-windows-amd64.exe.zip

流程

1. 进入 内容网关 并下载相应的文件：
   - 如果您下载了二进制文件，请将其重命名为 **odo.exe**。
   - 如果您下载了一个存档包，使用 ZIP 程序解压二进制文件，然后将其重命名为 **odo.exe**。
2. 将 **odo.exe** 二进制文件移到 **PATH** 中的一个目录中。
   要查看您的 **PATH**，请打开命令提示并执行以下命令：

```
C:\> path
```

3. 验证您的系统中现在可用的 **odo**：

```
C:\> odo version
```

3.3.3. 在 macOS 中安装 odo

macOS 的 **odo** CLI 可用于下载作为一个二进制文件，并作为 tarball 进行下载。

操作系统	二进制	Tarball
macOS	odo-darwin-amd64	odo-darwin-amd64.tar.gz

流程

1. 进入 内容网关 并下载相应的文件：
   - 如果下载二进制文件，请将其重命名为 **odo** ：

```
$ curl -L https://developers.redhat.com/content-gateway/rest/mirror/pub/openshift-v4/clients/odo/latest/odo-darwin-amd64 -o odo
```

   - 如果下载 tarball，解压二进制文件：

```
$ curl -L https://developers.redhat.com/content-gateway/rest/mirror/pub/openshift-v4/clients/odo/latest/odo-darwin-amd64.tar.gz -o odo.tar.gz
$ tar xvzf odo.tar.gz
```

2. 更改二进制的权限：

```
# chmod +x odo
```

3. 将 **odo** 二进制文件放到 **PATH** 中的目录中。
   要查看您的 **PATH**，请执行以下命令：

```
$ echo $PATH
```

4. 验证您的系统中现在可用的 **odo**：

```
$ odo version
```

3.3.4. 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
```

3.3.5. 使用 RPM 在 Red Hat Enterprise Linux(RHEL)中安装 odo

对于 Red Hat Enterprise Linux(RHEL)，您可以使用 RPM 安装 **odo** CLI。

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 Developer Tools and Services*'
```

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

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

5. 启用 **odo** 所需的存储库：

```
# subscription-manager repos --enable="ocp-tools-4.9-for-rhel-8-x86_64-rpms"
```

6. 安装 **odo** 软件包：

```
# yum install odo
```

7. 验证您的系统中现在可用的 **odo**：

```
$ odo version
```

3.4. 配置 ODO CLI

您可以在 **preference.yaml** 文件中找到 **odo** 的全局设置，该文件位于 **$HOME/.odo** 目录中。

您可以通过导出 **GLOBALODOCONFIG** 变量来为 **preference.yaml** 文件设置不同的位置。

3.4.1. 查看当前配置

您可以使用以下命令查看当前的 **odo** CLI 配置：

```
$ odo preference view
```

输出示例

```
PARAMETER           CURRENT_VALUE
UpdateNotification
NamePrefix
Timeout
BuildTimeout
PushTimeout
Ephemeral
ConsentTelemetry    true
```

3.4.2. 设置值

您可以使用以下命令为首选键设置值：

```
$ odo preference set <key> <value>
```

注意

首选项键不区分大小写。

示例命令

```
$ odo preference set updatenotification false
```

输出示例

```
Global preference was successfully updated
```

3.4.3. 取消设置值

您可以使用以下命令为首选键取消设置值：

```
$ odo preference unset <key>
```

-

注意

您可以使用 **-f** 标志跳过确认。

示例命令

```
$ odo preference unset updatenotification
? Do you want to unset updatenotification in the preference (y/N) y
```

输出示例

```
Global preference was successfully updated
```

3.4.4. 首选键盘表

下表显示了为 **odo** CLI 设置首选项键的可用选项：

首选键	描述	默认值
UpdateNotification	控制是否显示更新 **odo** 的通知。	true
NamePrefix	为 **odo** 资源设置默认名称前缀。例如，**组件**或**存储**。	当前目录名称
Timeout（超时）	Kubernetes 服务器连接检查的超时。	1 秒
BuildTimeout	等待 git 组件的构建完成超时。	300 秒
PushTimeout	等待组件启动超时。	240 秒
Ephemeral	控制 **odo** 是否应该创建一个 **emptyDir** 卷来存储源代码。	true
ConsentTelemetry	控制 **odo** 是否可以为用户的 **odo** 使用收集遥测功能。	False

3.4.5. 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。

3.5. ODO CLI 参考指南

3.5.1. odo build-images

odo 可根据 Dockerfile 构建容器镜像，并将这些镜像推送到 registry。

在运行 **odo build-images** 命令时，**odo** 会使用 **镜像** 类型搜索 **devfile.yaml** 中的所有组件，例如：

```
components:
- image:
    imageName: quay.io/myusername/myimage
    dockerfile:
      uri: ./Dockerfile (1)
      buildContext: ${PROJECTS_ROOT} (2)
  name: component-built-from-dockerfile
```

(1) **uri** 字段指示要使用的 Dockerfile 的相对路径，相对于包含 **devfile.yaml** 的目录。devfile 规格表示 **uri** 也可以是 HTTP URL，但 odo 尚不支持此 URL。

(2) **buildContext** 指示用作构建上下文的目录。默认值为 **${PROJECTS_ROOT}**。

对于每个镜像组件，odo 执行 **podman** 或 **docker**（按此顺序找到的第一个），以使用指定的 Dockerfile、构建上下文和参数构建镜像。

如果将 **--push** 标志传递给命令，则镜像会在构建后推送到其 registry。

3.5.2. odo catalog

odo 使用不同的 *目录* 来部署 *组件* 和 *服务*。

3.5.2.1. 组件

odo 使用可移植 *devfile* 格式来描述组件。它可以连接到各种 devfile registry，以便为不同的语言和框架下载 devfile。如需更多信息，请参阅 **odo registry**。

3.5.2.1.1. 列出组件

要列出不同 registry 中可用的所有 *devfile*，请运行以下命令：

```
$ odo catalog list components
```

输出示例

```
NAME           DESCRIPTION                      REGISTRY
go             Stack with the latest Go version     DefaultDevfileRegistry
java-maven     Upstream Maven and OpenJDK 11        DefaultDevfileRegistry
```

```
nodejs          Stack with Node.js 14              DefaultDevfileRegistry
php-laravel     Stack with Laravel 8               DefaultDevfileRegistry
python          Python Stack with Python 3.7       DefaultDevfileRegistry
[...]
```

3.5.2.1.2. 获取有关组件的信息

要获得有关特定组件的更多信息，请运行以下命令：

```
$ odo catalog describe component
```

例如，运行以下命令：

```
$ odo catalog describe component nodejs
```

输出示例

```
* Registry: DefaultDevfileRegistry (1)

Starter Projects: (2)
---
name: nodejs-starter
attributes: {}
description: ""
subdir: ""
projectsource:
  sourcetype: ""
  git:
    gitlikeprojectsource:
      commonprojectsource: {}
      checkoutfrom: null
      remotes:
        origin: https://github.com/odo-devfiles/nodejs-ex.git
  zip: null
  custom: null
```

1. *registry* 是从中检索 devfile 的 registry。
2. *Starter 项目*是 devfile 的同一语言和框架的示例项目，可帮助您启动一个新项目。

如需有关从入门项目创建项目的更多信息，请参阅 **odo create**。

3.5.2.2. 服务

odo 可使用 *Operator* 帮助部署*服务*。

odo 仅支持 *Operator Lifecycle Manager* 帮助部署的 Operator。

3.5.2.2.1. 列出服务

要列出可用的 Operator 及其关联的服务，请运行以下命令：

```
$ odo catalog list services
```

输出示例

```
Services available through Operators
NAME                            CRDs
postgresql-operator.v0.1.1          Backup, Database
redis-operator.v0.8.0               RedisCluster, Redis
```

在本例中，集群中安装两个 Operator。**postgresql-operator.v0.1.1** Operator 部署与 PostgreSQL 相关的服务： **Backup** 和 **Database**。**redis-operator.v0.8.0** Operator 将部署与 Redis 相关的服务：**RedisCluster** 和 **Redis**。

注意

要获取所有可用 Operator 的列表， **odo** 会获取当前处于 *Succeeded* 阶段的当前命名空间的 ClusterServiceVersion(CSV)资源。对于支持集群范围的访问权限的 Operator，当创建新命名空间时，这些资源会自动添加到其中。但是，在 *Succeeded* 阶段前可能需要一些时间，**odo** 可能会返回空列表，直到资源就绪为止。

3.5.2.2.2. 搜索服务

要通过关键字搜索特定服务，请运行以下命令：

```
$ odo catalog search service
```

例如，要检索 PostgreSQL 服务，请运行以下命令：

```
$ odo catalog search service postgres
```

输出示例

```
Services available through Operators
NAME                        CRDs
postgresql-operator.v0.1.1      Backup, Database
```

您将看到在其名称中包含 search 关键字的 Operator 列表。

3.5.2.2.3. 获取有关服务的信息

要获取有关特定服务的更多信息，请运行以下命令：

```
$ odo catalog describe service
```

例如：

```
$ odo catalog describe service postgresql-operator.v0.1.1/Database
```

输出示例

```
KIND:    Database
```

```
VERSION: v1alpha1

DESCRIPTION:
    Database is the Schema for the the Database Database API

FIELDS:
  awsAccessKeyId (string)
    AWS S3 accessKey/token ID

    Key ID of AWS S3 storage. Default Value: nil Required to create the Secret
    with the data to allow send the backup files to AWS S3 storage.
[...]
```

服务通过 CustomResourceDefinition(CRD)资源表示在集群中。上一命令显示 CRD 的详细信息，如 **kind**、**version**，以及用于定义此自定义资源实例的字段列表。

从 CRD 中包含的 *OpenAPI schema* 中提取字段列表。此信息在 CRD 中是可选的，如果不存在，它将从代表该服务的 ClusterServiceVersion(CSV)资源中提取。

也可以请求 Operator 支持的服务的描述，而无需提供 CRD 类型信息。要描述没有 CRD 的集群中的 Redis Operator，请运行以下命令：

```
$ odo catalog describe service redis-operator.v0.8.0
```

输出示例

```
NAME: redis-operator.v0.8.0
DESCRIPTION:

 A Golang based redis operator that will make/oversee Redis
 standalone/cluster mode setup on top of the Kubernetes. It can create a
 redis cluster setup with best practices on Cloud as well as the Bare metal
 environment. Also, it provides an in-built monitoring capability using

... (cut short for beverity)

 Logging Operator is licensed under [Apache License, Version
 2.0](https://github.com/OT-CONTAINER-KIT/redis-operator/blob/master/LICENSE)


CRDs:
 NAME          DESCRIPTION
 RedisCluster  Redis Cluster
 Redis         Redis
```

3.5.3. odo create

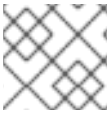
odo 使用 *devfile* 来存储组件的配置，并描述组件的资源，如存储和服务。*odo create* 命令生成这个文件。

3.5.3.1. 创建组件

要为现有项目创建 *devfile*，请运行 **odo create** 命令，使用组件的名称和类型（例如 **nodejs** 或 **go**）：

```
odo create nodejs mynodejs
```

在示例中，**nodejs** 是组件的类型，**mynodejs** 是 **odo** 为您创建的组件的名称。



注意

如需所有支持的组件类型列表，请运行 **odo catalog list components** 命令。

如果您的源代码存在于当前目录之外，可以使用 **--context** 标志来指定路径。例如，如果 nodejs 组件的源位于相对于当前工作目录的名为 **node-backend** 的文件夹，则运行以下命令：

```
odo create nodejs mynodejs --context ./node-backend
```

--context 标志支持相对和绝对路径。

要指定部署组件的项目或应用程序，请使用 **--project** 和 **--app** 标志。例如，要在 **backend** 项目中创建一个部分 **myapp** 应用程序的组件，请运行以下命令：

```
odo create nodejs --app myapp --project backend
```

注意

如果没有指定这些标志，它们将默认为活跃的应用和项目。

3.5.3.2. Starter （初学者）项目

如果您没有现有的源代码，但希望快速设置并运行，请使用初学者项目来试验 devfile 和组件。要使用初学者项目，在 **odo create** 命令中添加 **--starter** 标志。

要获取组件类型的可用启动程序项目列表，请运行 **odo catalog describe component** 命令。例如，若要获取 nodejs 组件类型的所有可用入门项目，请运行以下命令：

```
odo catalog describe component nodejs
```

然后，在 **odo create** 命令中使用 **--starter** 标志指定所需的项目：

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

这将下载与所选组件类型对应的示例模板，在本例中为 **nodejs**。模板已下载到您的当前目录中，或附加到 **--context** 标志指定的位置。如果初学者项目有自己的 devfile，则会保留此 devfile。

3.5.3.3. 使用现有的 devfile

如果要从现有 devfile 创建新组件，您可以使用 **--devfile** 标志指定 devfile 的路径。例如，要基于 GitHub 中的 devfile 创建名为 **mynodejs** 的组件，请使用以下命令：

```
odo create mynodejs --devfile https://raw.githubusercontent.com/odo-
devfiles/registry/master/devfiles/nodejs/devfile.yaml
```

3.5.3.4. 互动创建

您还可以以互动方式运行 **odo create** 命令，以引导您按照创建组件所需的步骤进行指导：

```
$ odo create

? Which devfile component type do you wish to create go
? What do you wish to name the new devfile component go-api
? What project do you want the devfile component to be created in default
Devfile Object Validation
 ✓  Checking devfile existence [164258ns]
 ✓  Creating a devfile component from registry: DefaultDevfileRegistry [246051ns]
Validation
 ✓  Validating if devfile name is correct [92255ns]
? Do you want to download a starter project Yes

Starter Project
 ✓  Downloading starter project go-starter from https://github.com/devfile-samples/devfile-stack-go.git
[429ms]

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

系统将提示您选择组件类型、名称和项目。您还可以选择是否下载初学者项目。完成后，在工作目录中创建一个新的 **devfile.yaml** 文件。

要将这些资源部署到集群中，请运行 **odo push** 命令。

3.5.4. odo delete

odo delete 命令对删除由 **odo** 管理的资源很有用。

3.5.4.1. 删除组件

要删除 *devfile* 组件，请运行 **odo delete** 命令：

```
$ odo delete
```

如果组件已推送到集群，则组件将从集群中删除，以及其依赖存储、URL、secret 和其他资源。如果组件还没有推送，则命令退出并显示一个错误，表示它无法找到集群中的资源。

使用 **-f** 或 **--force** 标志以避免出现确认问题。

3.5.4.2. 取消部署 devfile Kubernetes 组件

要取消部署 devfile Kubernetes 组件（已使用 **odo deploy** 部署 ）,请使用 **--deploy** 标志执行 **odo delete** 命令：

```
$ odo delete --deploy
```

使用 **-f** 或 **--force** 标志以避免出现确认问题。

3.5.4.3. 全部删除

要删除包括以下项目的所有工件，请运行带有 **--all** 标记的 **odo delete** 命令：

- *devfile* 组件

- 使用 **odo deploy** 命令部署的 devfile Kubernetes 组件
- devfile
- 本地配置

```
$ odo delete --all
```

3.5.4.4. 可用标记

-f,--force

使用此标志以避免出现确认问题。

-w,--wait

使用此标志等待组件删除和任何依赖项。取消部署时，此标志不起作用。

关于 *Common Flags* 的文档提供有关可用于命令的标记的更多信息。

3.5.5. odo deploy

odo 可用于部署组件的方式类似于如何使用 CI/CD 系统进行部署。首先， **odo** 构建容器镜像，然后部署部署组件所需的 Kubernetes 资源。

在运行命令 **odo deploy** 时， **odo** 在 devfile 中搜索 kind **deploy** 的默认命令，并执行这个命令。从 2.2.0 版本开始的 devfile 格式支持 kind **部署**。

deploy 命令通常是一个 *复合* 命令，由多个 *应用命令组成* ：

- 引用 **镜像** 组件（应用时）的命令将构建要部署的容器的镜像，然后将其推送到注册表。
- 引用 Kubernetes 组件的命令 （应用时）将在集群中创建 Kubernetes 资源。

使用以下示例 **devfile.yaml** 文件，会使用 目录中存在的 **Dockerfile** 来构建容器镜像。镜像被推送到其 registry，然后使用这个全新的构建镜像在集群中创建 Kubernetes Deployment 资源。

```
schemaVersion: 2.2.0
[...]
variables:
  CONTAINER_IMAGE: quay.io/phmartin/myimage
commands:
  - id: build-image
    apply:
      component: outerloop-build
  - id: deployk8s
    apply:
      component: outerloop-deploy
  - id: deploy
    composite:
      commands:
        - build-image
        - deployk8s
      group:
        kind: deploy
        isDefault: true
components:
```

```
- name: outerloop-build
  image:
    imageName: "{{CONTAINER_IMAGE}}"
    dockerfile:
      uri: ./Dockerfile
      buildContext: ${PROJECTS_ROOT}
- name: outerloop-deploy
  kubernetes:
    inlined: |
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: my-component
      spec:
        replicas: 1
        selector:
          matchLabels:
            app: node-app
        template:
          metadata:
            labels:
              app: node-app
          spec:
            containers:
              - name: main
                image: {{CONTAINER_IMAGE}}
```

3.5.6. odo link

odo link 命令帮助将 **odo** 组件链接到由 Operator 支持的服务或另一个 **odo** 组件。它通过使用 Service Binding Operator 来达到此目的。目前，**odo** 使用 Service Binding 库，而不是 Operator 本身来实现所需的功能。

3.5.6.1. 各种链接选项

odo 提供了不同的选项，用来将组件链接到 Operator 支持的服务或另一个 **odo** 组件。无论您将组件链接到服务还是另一个组件，都可以使用这些选项（或标志）。

3.5.6.1.1. 默认行为

默认情况下，**odo link** 命令在组件目录中创建一个名为 **kubernetes/** 的目录，并在其中存储有关服务和链接的信息（YAML 清单）。当使用 **odo push** 时，**odo** 会将这些清单与 Kubernetes 集群上的资源状态进行比较，并决定是否需要创建、修改或销毁资源以匹配用户指定的内容。

3.5.6.1.2. --inlined 标记

如果在 **odo link** 命令中指定 **--inlined** 标志，**odo** 会将 **devfile.yaml** 中的链接信息存储在组件目录中，而不是在 **kubernetes/** 目录下创建一个文件。**--inlined** 标记的行为与 **odo link** 和 **odo service create** 命令相似。如果您希望在一个 **devfile.yaml** 中存储的所有内容，则此标志很有用。您必须记住在每个 **odo link** 和 **odo service create** 命令中使用 **--inlined** 标志。

3.5.6.1.3. --map 标志

有时，除了默认可用的组件外，您可能还想向组件添加更多绑定信息。例如，如果您将组件链接到服务，并希望从服务的 spec 中绑定一些信息（用于规格的缩写），您可以使用 **--map** 标志。请注意，**odo** 不会

针对所链接的服务或组件的 spec 进行任何验证。只有在您熟悉 Kubernetes YAML 清单时，才建议使用这个标志。

3.5.6.1.4. --bind-as-files 标志

对于目前讨论的所有链接选项，**odo** 会将绑定信息作为环境变量注入组件。如果您想要将这些信息挂载为文件，您可以使用 **--bind-as-files** 标志。这可让 **odo** 将绑定信息作为文件注入到组件的 Pod 中的 **/bindings** 位置。与环境变量方案相比，当您使用 **--bind-as-files** 时，文件会以键命名，并且这些键的值存储为这些文件的内容。

3.5.6.2. 例子

3.5.6.2.1. 默认 odo 链接

在以下示例中，后端组件使用默认的 **odo link** 命令与 PostgreSQL 服务相关联。对于后端组件，请确定您的组件和服务被推送到集群：

```
$ odo list
```

输出示例

```
APP     NAME        PROJECT       TYPE       STATE      MANAGED BY ODO
app     backend     myproject     spring     Pushed     Yes
```

```
$ odo service list
```

输出示例

```
NAME                    MANAGED BY ODO     STATE      AGE
PostgresCluster/hippo     Yes (backend)      Pushed     59m41s
```

现在，运行 **odo link** 将后端组件与 PostgreSQL 服务链接：

```
$ odo link PostgresCluster/hippo
```

输出示例

```
 ✓  Successfully created link between component "backend" and service "PostgresCluster/hippo"

To apply the link, please use `odo push`
```

然后，运行 **odo push** 在 Kubernetes 集群中实际创建链接。

在 **odo push** 成功后，您会看到几个结果：

1. 当您打开由 backend 组件部署的应用程序的 URL 时，它会显示数据库中的 **todo** 列表。例如，在 **odo url list** 命令的输出中，会列出 **todos** 的路径：

   ```
   $ odo url list
   ```

   输出示例

```
Found the following URLs for component backend
NAME        STATE     URL                                  PORT     SECURE     KIND
8080-tcp    Pushed    http://8080-tcp.192.168.39.112.nip.io    8080     false      ingress
```

URL 的正确路径为 http://8080-tcp.192.168.39.112.nip.io/api/v1/todos。确切的 URL 取决于您的设置。另请注意，除非添加一些，否则数据库中也没有 **todos**，因此 URL 可能会只显示空的 JSON 对象。

2. 您可以查看与 Postgres 服务注入后端组件相关的绑定信息。默认情况下，此绑定信息作为环境变量注入。您可以在后端组件的目录中使用 **odo describe** 命令检查它：

```
$ odo describe
```

输出示例：

```
Component Name: backend
Type: spring
Environment Variables:
 · PROJECTS_ROOT=/projects
 · PROJECT_SOURCE=/projects
 · DEBUG_PORT=5858
Storage:
 · m2 of size 3Gi mounted to /home/user/.m2
URLs:
 · http://8080-tcp.192.168.39.112.nip.io exposed via 8080
Linked Services:
 · PostgresCluster/hippo
   Environment Variables:
   · POSTGRESCLUSTER_PGBOUNCER-EMPTY
   · POSTGRESCLUSTER_PGBOUNCER.INI
   · POSTGRESCLUSTER_ROOT.CRT
   · POSTGRESCLUSTER_VERIFIER
   · POSTGRESCLUSTER_ID_ECDSA
   · POSTGRESCLUSTER_PGBOUNCER-VERIFIER
   · POSTGRESCLUSTER_TLS.CRT
   · POSTGRESCLUSTER_PGBOUNCER-URI
   · POSTGRESCLUSTER_PATRONI.CRT-COMBINED
   · POSTGRESCLUSTER_USER
   · pgImage
   · pgVersion
   · POSTGRESCLUSTER_CLUSTERIP
   · POSTGRESCLUSTER_HOST
   · POSTGRESCLUSTER_PGBACKREST_REPO.CONF
   · POSTGRESCLUSTER_PGBOUNCER-USERS.TXT
   · POSTGRESCLUSTER_SSH_CONFIG
   · POSTGRESCLUSTER_TLS.KEY
   · POSTGRESCLUSTER_CONFIG-HASH
   · POSTGRESCLUSTER_PASSWORD
   · POSTGRESCLUSTER_PATRONI.CA-ROOTS
   · POSTGRESCLUSTER_DBNAME
   · POSTGRESCLUSTER_PGBOUNCER-PASSWORD
   · POSTGRESCLUSTER_SSHD_CONFIG
   · POSTGRESCLUSTER_PGBOUNCER-FRONTEND.KEY
   · POSTGRESCLUSTER_PGBACKREST_INSTANCE.CONF
   · POSTGRESCLUSTER_PGBOUNCER-FRONTEND.CA-ROOTS
```

```
· POSTGRESCLUSTER_PGBOUNCER-HOST
· POSTGRESCLUSTER_PORT
· POSTGRESCLUSTER_ROOT.KEY
· POSTGRESCLUSTER_SSH_KNOWN_HOSTS
· POSTGRESCLUSTER_URI
· POSTGRESCLUSTER_PATRONI.YAML
· POSTGRESCLUSTER_DNS.CRT
· POSTGRESCLUSTER_DNS.KEY
· POSTGRESCLUSTER_ID_ECDSA.PUB
· POSTGRESCLUSTER_PGBOUNCER-FRONTEND.CRT
· POSTGRESCLUSTER_PGBOUNCER-PORT
· POSTGRESCLUSTER_CA.CRT
```

其中一些变量在后端组件的 **src/main/resources/application.properties** 文件中使用，以便 Java Spring Boot 应用程序可以连接到 PostgreSQL 数据库服务。

3. 最后，**odo** 在后端组件的目录中创建一个名为 **kubernetes/** 的目录，其中包含以下文件：

```
$ ls kubernetes
odo-service-backend-postgrescluster-hippo.yaml  odo-service-hippo.yaml
```

这些文件包含两个资源的信息（YAML 清单）：

a. **odo-service-hippo.yaml** - 使用 **odo service create --from-file ../postgrescluster.yaml** 命令创建的 Postgres *服务*。

b. **odo-service-backend-postgrescluster-hippo.yaml** - 使用 **odo link** 命令创建的 *链接*。

3.5.6.2.2. 使用带有 --inlined 标记的 odo 链接

在 **odo link** 命令中使用 **--inlined** 标志与没有标志的 **odo link** 命令的效果相同，在注入绑定信息中。但是，通常的差异是，在上述情况下，**kubernetes/** 目录下有两个清单文件，一个用于 Postgres 服务，另一个用于后端组件和该服务之间的链接。但是，当您传递 **--inlined** 标志时，**odo** 不会在 **kubernetes/** 目录下创建一个文件来存储 YAML 清单，而是将其内联存储在 **devfile.yaml** 文件中。

要查看此信息，请首先从 PostgreSQL 服务中取消链接组件：

```
$ odo unlink PostgresCluster/hippo
```

输出示例：

```
 ✓  Successfully unlinked component "backend" from service "PostgresCluster/hippo"

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

要在集群中取消链接它们，请运行 **odo push**。现在，如果您检查 **kubernetes/** 目录，则只看到一个文件：

```
$ ls kubernetes
odo-service-hippo.yaml
```

接下来，使用 **--inlined** 标志来创建链接：

```
$ odo link PostgresCluster/hippo --inlined
```

-

输出示例：

```
 ✓  Successfully created link between component "backend" and service "PostgresCluster/hippo"

To apply the link, please use `odo push`
```

您需要运行 **odo push** 以便在集群中创建链接，如省略 **--inlined** 标志的步骤。**odo** 将配置存储在 **devfile.yaml** 中。在这个文件中，您可以看到类似如下的条目：

```
kubernetes:
  inlined: |
    apiVersion: binding.operators.coreos.com/v1alpha1
    kind: ServiceBinding
    metadata:
      creationTimestamp: null
      name: backend-postgrescluster-hippo
    spec:
      application:
        group: apps
        name: backend-app
        resource: deployments
        version: v1
      bindAsFiles: false
      detectBindingResources: true
      services:
      - group: postgres-operator.crunchydata.com
        id: hippo
        kind: PostgresCluster
        name: hippo
        version: v1beta1
    status:
      secret: ""
  name: backend-postgrescluster-hippo
```

现在，如果您运行 **odo unlink PostgresCluster/hippo**，**odo** 会首先从 **devfile.yaml** 中删除链接信息，然后后续 **odo push** 将从集群中删除链接。

3.5.6.2.3. 自定义绑定

odo link 接受标记 **--map**，它可以将自定义绑定信息注入组件。此类绑定信息将从您链接到您的组件的资源清单中获取。例如，在后端组件和 PostgreSQL 服务的上下文中，您可以将 PostgreSQL 服务的清单 **postgrescluster.yaml** 文件中的信息注入后端组件。

如果 **PostgresCluster** 服务的名称是 **hippo** （或者 **odo service list** 的输出，如果您的 PostgresCluster 服务被命名），当您需要将 YAML 定义中的 **postgresVersion** 值注入后端组件时，请运行以下命令：

```
$ odo link PostgresCluster/hippo --map pgVersion='{{ .hippo.spec.postgresVersion }}'
```

请注意，如果 Postgres 服务的名称与 **hippo** 不同，则必须在上述命令中指定在 **pgVersion** 的值代替 **.hippo** 的位置。

在链接操作后，照常运行 **odo push**。在成功完成推送操作后，您可以从后端组件目录中运行以下命令，以验证是否正确注入自定义映射：

-

```
$ odo exec -- env | grep pgVersion
```

输出示例：

```
pgVersion=13
```

因为您可能希望注入多个自定义绑定信息，**odo link** 接受映射的多个键值对。唯一约束应将它们指定为 **--map <key>=<value>**。例如，如果还想将 PostgreSQL 镜像信息与版本一起注入，您可以运行：

```
$ odo link PostgresCluster/hippo --map pgVersion='{{ .hippo.spec.postgresVersion }}' --map
pgImage='{{ .hippo.spec.image }}'
```

然后运行 **odo push**。要验证两个映射是否已正确注入的映射，请运行以下命令：

```
$ odo exec -- env | grep -e "pgVersion\|pgImage"
```

输出示例：

```
pgVersion=13
pgImage=registry.developers.crunchydata.com/crunchydata/crunchy-postgres-ha:centos8-13.4-0
```

3.5.6.2.3.1. 使用内联还是不使用？

您可以接受默认行为，**odo link** 为 **kubernetes/** 目录下的链接生成清单文件。另外，如果您想将所有内容存储在单个 **devfile.yaml** 文件中，您可以使用 **--inlined** 标志。

3.5.6.3. 将绑定作为文件绑定

odo link 提供的另一个有用标志是 **--bind-as-files**。当传递此标记时，绑定信息不会作为环境变量注入组件的 Pod 中，而是作为文件系统挂载。

确保后端组件和 PostgreSQL 服务之间没有现有链接。您可以通过在后端组件的目录中运行 **odo describe** 来检查输出是否类似以下内容：

```
Linked Services:
・PostgresCluster/hippo
```

使用以下命令从组件中取消链接该服务：

```
$ odo unlink PostgresCluster/hippo
$ odo push
```

3.5.6.4. --bind-as-files 示例

3.5.6.4.1. 使用默认 odo 链接

默认情况下，**odo** 在 **kubernetes/** 目录下创建清单文件来存储链接信息。使用以下命令链接后端组件和 PostgreSQL 服务：

```
$ odo link PostgresCluster/hippo --bind-as-files
$ odo push
```

odo describe 输出示例：

```
$ odo describe

Component Name: backend
Type: spring
Environment Variables:
 · PROJECTS_ROOT=/projects
 · PROJECT_SOURCE=/projects
 · DEBUG_PORT=5858
 · SERVICE_BINDING_ROOT=/bindings
 · SERVICE_BINDING_ROOT=/bindings
Storage:
 · m2 of size 3Gi mounted to /home/user/.m2
URLs:
 · http://8080-tcp.192.168.39.112.nip.io exposed via 8080
Linked Services:
 · PostgresCluster/hippo
   Files:
    · /bindings/backend-postgrescluster-hippo/pgbackrest_instance.conf
    · /bindings/backend-postgrescluster-hippo/user
    · /bindings/backend-postgrescluster-hippo/ssh_known_hosts
    · /bindings/backend-postgrescluster-hippo/clusterIP
    · /bindings/backend-postgrescluster-hippo/password
    · /bindings/backend-postgrescluster-hippo/patroni.yaml
    · /bindings/backend-postgrescluster-hippo/pgbouncer-frontend.crt
    · /bindings/backend-postgrescluster-hippo/pgbouncer-host
    · /bindings/backend-postgrescluster-hippo/root.key
    · /bindings/backend-postgrescluster-hippo/pgbouncer-frontend.key
    · /bindings/backend-postgrescluster-hippo/pgbouncer.ini
    · /bindings/backend-postgrescluster-hippo/uri
    · /bindings/backend-postgrescluster-hippo/config-hash
    · /bindings/backend-postgrescluster-hippo/pgbouncer-empty
    · /bindings/backend-postgrescluster-hippo/port
    · /bindings/backend-postgrescluster-hippo/dns.crt
    · /bindings/backend-postgrescluster-hippo/pgbouncer-uri
    · /bindings/backend-postgrescluster-hippo/root.crt
    · /bindings/backend-postgrescluster-hippo/ssh_config
    · /bindings/backend-postgrescluster-hippo/dns.key
    · /bindings/backend-postgrescluster-hippo/host
    · /bindings/backend-postgrescluster-hippo/patroni.crt-combined
    · /bindings/backend-postgrescluster-hippo/pgbouncer-frontend.ca-roots
    · /bindings/backend-postgrescluster-hippo/tls.key
    · /bindings/backend-postgrescluster-hippo/verifier
    · /bindings/backend-postgrescluster-hippo/ca.crt
    · /bindings/backend-postgrescluster-hippo/dbname
    · /bindings/backend-postgrescluster-hippo/patroni.ca-roots
    · /bindings/backend-postgrescluster-hippo/pgbackrest_repo.conf
    · /bindings/backend-postgrescluster-hippo/pgbouncer-port
    · /bindings/backend-postgrescluster-hippo/pgbouncer-verifier
    · /bindings/backend-postgrescluster-hippo/id_ecdsa
    · /bindings/backend-postgrescluster-hippo/id_ecdsa.pub
    · /bindings/backend-postgrescluster-hippo/pgbouncer-password
    · /bindings/backend-postgrescluster-hippo/pgbouncer-users.txt
    · /bindings/backend-postgrescluster-hippo/sshd_config
    · /bindings/backend-postgrescluster-hippo/tls.crt
```

之前的 **odo describe** 输出中的 **key=value** 格式是一个环境变量，现在作为一个文件被挂载。使用 **cat** 命令查看其中的一些文件的内容：

示例命令：

```
$ odo exec -- cat /bindings/backend-postgrescluster-hippo/password
```

输出示例：

```
q({JC:jn^mm/Bw}eu+j.GX{k
```

示例命令：

```
$ odo exec -- cat /bindings/backend-postgrescluster-hippo/user
```

输出示例：

```
hippo
```

示例命令：

```
$ odo exec -- cat /bindings/backend-postgrescluster-hippo/clusterIP
```

输出示例：

```
10.101.78.56
```

3.5.6.4.2. 使用 --inlined

使用 **--bind-as-files** 和 **--inlined** 的结果与使用 **odo link --inlined** 类似。链接的清单存储在 **devfile.yaml** 中，而不是存储在 **kubernetes/** 目录中的单独文件中。除此之外，**odo describe** 输出的内容与之前的输出相同。

3.5.6.4.3. 自定义绑定

当在将后端组件与 PostgreSQL 服务链接时传递自定义绑定时，这些自定义绑定不会作为环境变量注入，而是作为文件挂载。例如：

```
$ odo link PostgresCluster/hippo --map pgVersion='{{ .hippo.spec.postgresVersion }}' --map
pgImage='{{ .hippo.spec.image }}' --bind-as-files
$ odo push
```

这些自定义绑定作为文件挂载，而不是作为环境变量注入。要验证是否可以正常工作，请运行以下命令：

示例命令：

```
$ odo exec -- cat /bindings/backend-postgrescluster-hippo/pgVersion
```

输出示例：

```
13
```

示例命令：

```
$ odo exec -- cat /bindings/backend-postgrescluster-hippo/pgImage
```

输出示例：

```
registry.developers.crunchydata.com/crunchydata/crunchy-postgres-ha:centos8-13.4-0
```

3.5.7. odo registry

odo 使用可移植 *devfile* 格式来描述组件。**odo** 可以连接到各种 devfile registry，以下载用于不同语言和框架的 devfile。

您可以连接到公开可用的 devfile registry，也可以安装自己的 *安全 Registry*。

您可以使用 **odo registry** 命令管理 **odo** 使用的 registry 来检索 devfile 信息。

3.5.7.1. 列出 registry

要列出 **odo** 当前联系的 registry，请运行以下命令：

```
$ odo registry list
```

输出示例：

```
NAME                    URL                         SECURE
DefaultDevfileRegistry  https://registry.devfile.io  No
```

DefaultDevfileRegistry 是 odo 使用的默认 registry，它由 devfile.io 项目提供。

3.5.7.2. 添加 registry

要添加 registry，请运行以下命令：

```
$ odo registry add
```

输出示例：

```
$ odo registry add StageRegistry https://registry.stage.devfile.io
New registry successfully added
```

如果要部署自己的安全 Registry，您可以指定个人访问令牌来使用 **--token** 标志向安全 registry 进行身份验证：

```
$ odo registry add MyRegistry https://myregistry.example.com --token <access_token>
New registry successfully added
```

3.5.7.3. 删除 registry

要删除 registry，请运行以下命令：

```
$ odo registry delete
```

输出示例：

```
$ odo registry delete StageRegistry
? Are you sure you want to delete registry "StageRegistry" Yes
Successfully deleted registry
```

使用 **--force** （或 **-f**）标记强制删除 registry，而无需确认。

3.5.7.4. 更新 registry

要更新已注册的 registry 的 URL 或已经注册的个人访问令牌，请运行以下命令：

```
$ odo registry update
```

输出示例：

```
$ odo registry update MyRegistry https://otherregistry.example.com --token <other_access_token>
? Are you sure you want to update registry "MyRegistry" Yes
Successfully updated registry
```

使用 **--force** （或 **-f**）标记强制更新 registry，而无需确认。

3.5.8. odo service

odo 可使用 *Operator* 帮助部署*服务*。

可使用 **odo catalog** 命令找到可用的 Operator 和服务列表。

服务在 *组件* 上下文中创建，因此在部署服务前运行 **odo create** 命令。

服务通过以下两个步骤进行部署：

1. 定义服务并将其定义存储在 devfile 中。
2. 使用 **odo push** 命令将定义的服务部署到集群。

3.5.8.1. 创建新服务

要创建新服务，请运行以下命令：

```
$ odo service create
```

例如，要创建一个名为 **my-redis-service** 的 Redis 服务实例，您可以运行以下命令：

输出示例

```
$ odo catalog list services
Services available through Operators
```

```
NAME                    CRDs
redis-operator.v0.8.0   RedisCluster, Redis

$ odo service create redis-operator.v0.8.0/Redis my-redis-service
Successfully added service to the configuration; do 'odo push' to create service on the cluster
```

此命令在 **kubernetes/** 目录中创建 Kubernetes 清单，其中包含服务的定义，此文件从 **devfile.yaml** 文件引用。

```
$ cat kubernetes/odo-service-my-redis-service.yaml
```

输出示例

```
apiVersion: redis.redis.opstreelabs.in/v1beta1
kind: Redis
metadata:
  name: my-redis-service
spec:
  kubernetesConfig:
    image: quay.io/opstree/redis:v6.2.5
    imagePullPolicy: IfNotPresent
    resources:
      limits:
        cpu: 101m
        memory: 128Mi
      requests:
        cpu: 101m
        memory: 128Mi
    serviceType: ClusterIP
  redisExporter:
    enabled: false
    image: quay.io/opstree/redis-exporter:1.0
  storage:
    volumeClaimTemplate:
      spec:
        accessModes:
        - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
```

示例命令

```
$ cat devfile.yaml
```

输出示例

```
[...]
components:
- kubernetes:
    uri: kubernetes/odo-service-my-redis-service.yaml
  name: my-redis-service
[...]
```

请注意，所创建的实例的名称是可选的。如果您不提供名称，它将是服务的小写名称。例如，以下命令创建一个名为 **redis** 的 Redis 服务实例：

```
$ odo service create redis-operator.v0.8.0/Redis
```

3.5.8.1.1. 显示清单

默认情况下，在 **kubernetes/** 目录中创建一个新清单，从 **devfile.yaml** 文件引用。可以使用 **--inlined** 标志在 **devfile.yaml** 文件中内联清单：

```
$ odo service create redis-operator.v0.8.0/Redis my-redis-service --inlined
Successfully added service to the configuration; do 'odo push' to create service on the cluster
```

示例命令

```
$ cat devfile.yaml
```

输出示例

```
[...]
components:
- kubernetes:
    inlined: |
      apiVersion: redis.redis.opstreelabs.in/v1beta1
      kind: Redis
      metadata:
        name: my-redis-service
      spec:
        kubernetesConfig:
          image: quay.io/opstree/redis:v6.2.5
          imagePullPolicy: IfNotPresent
          resources:
            limits:
              cpu: 101m
              memory: 128Mi
            requests:
              cpu: 101m
              memory: 128Mi
          serviceType: ClusterIP
        redisExporter:
          enabled: false
          image: quay.io/opstree/redis-exporter:1.0
        storage:
          volumeClaimTemplate:
            spec:
              accessModes:
              - ReadWriteOnce
              resources:
                requests:
                  storage: 1Gi
  name: my-redis-service
[...]
```

3.5.8.1.2. 配置服务

如果没有特定的自定义，将使用默认配置创建该服务。您可以使用命令行参数或文件来指定您自己的配置。

3.5.8.1.2.1. 使用命令行参数

使用 **--parameters** （或 **-p**）指定您自己的配置。

以下示例使用三个参数配置 Redis 服务：

```
$ odo service create redis-operator.v0.8.0/Redis my-redis-service \
    -p kubernetesConfig.image=quay.io/opstree/redis:v6.2.5 \
    -p kubernetesConfig.serviceType=ClusterIP \
    -p redisExporter.image=quay.io/opstree/redis-exporter:1.0
Successfully added service to the configuration; do 'odo push' to create service on the cluster
```

示例命令

```
$ cat kubernetes/odo-service-my-redis-service.yaml
```

输出示例

```
apiVersion: redis.redis.opstreelabs.in/v1beta1
kind: Redis
metadata:
  name: my-redis-service
spec:
  kubernetesConfig:
    image: quay.io/opstree/redis:v6.2.5
    serviceType: ClusterIP
  redisExporter:
    image: quay.io/opstree/redis-exporter:1.0
```

您可以使用 **odo catalog describe service** 命令获取特定服务的可能参数。

3.5.8.1.2.2. 使用文件

使用 YAML 清单来配置您自己的规格。在以下示例中，Red Hat Redis 服务配置了三个参数。

1. 创建清单：

```
$ cat > my-redis.yaml <<EOF
apiVersion: redis.redis.opstreelabs.in/v1beta1
kind: Redis
metadata:
  name: my-redis-service
spec:
  kubernetesConfig:
    image: quay.io/opstree/redis:v6.2.5
    serviceType: ClusterIP
  redisExporter:
    image: quay.io/opstree/redis-exporter:1.0
EOF
```

2. 在清单中创建服务：

```
$ odo service create --from-file my-redis.yaml
Successfully added service to the configuration; do 'odo push' to create service on the cluster
```

3.5.8.2. 删除服务

要删除服务，请运行以下命令：

```
$ odo service delete
```

输出示例

```
$ odo service list
NAME                    MANAGED BY ODO     STATE            AGE
Redis/my-redis-service     Yes (api)          Deleted locally     5m39s
```

```
$ odo service delete Redis/my-redis-service
? Are you sure you want to delete Redis/my-redis-service Yes
Service "Redis/my-redis-service" has been successfully deleted; do 'odo push' to delete service from
the cluster
```

使用 **--force** （或 **-f**）标记强制删除该服务而无需确认。

3.5.8.3. 列出服务

要列出为组件创建的服务，请运行以下命令：

```
$ odo service list
```

输出示例

```
$ odo service list
NAME                    MANAGED BY ODO     STATE            AGE
Redis/my-redis-service-1   Yes (api)          Not pushed
Redis/my-redis-service-2   Yes (api)          Pushed           52s
Redis/my-redis-service-3   Yes (api)          Deleted locally   1m22s
```

对于每个服务，**STATE** 表示该服务是否使用 **odo push** 命令推送到集群，或者该服务仍然在集群中运行，但使用 **odo service delete** 命令在本地移除 devfile。

3.5.8.4. 获取有关服务的信息

要获取服务的详情，如其类型、版本、名称和配置参数列表，请运行以下命令：

```
$ odo service describe
```

输出示例

```
$ odo service describe Redis/my-redis-service
Version: redis.redis.opstreelabs.in/v1beta1
```

```
Kind: Redis
Name: my-redis-service
Parameters:
NAME                        VALUE
kubernetesConfig.image      quay.io/opstree/redis:v6.2.5
kubernetesConfig.serviceType  ClusterIP
redisExporter.image         quay.io/opstree/redis-exporter:1.0
```

3.5.9. odo storage

odo 允许用户管理附加到组件的存储卷。存储卷可以是使用 **emptyDir** Kubernetes 卷或 持久性卷声明 (PVC)的临时卷。PVC 允许用户声明持久性卷（如 GCE PersistentDisk 或 iSCSI 卷），而无需了解特定的云环境的详情。持久性存储卷可用于在重启后保留数据，并重新构建组件。

3.5.9.1. 添加存储卷

要在集群中添加存储卷，请运行以下命令：

```
$ odo storage create
```

输出示例：

```
$ odo storage create store --path /data --size 1Gi
✓  Added storage store to nodejs-project-ufyy

$ odo storage create tempdir --path /tmp --size 2Gi --ephemeral
✓  Added storage tempdir to nodejs-project-ufyy

Please use `odo push` command to make the storage accessible to the component
```

在上例中，第一个存储卷已挂载到 **/data** 路径中，大小为 **1Gi**，第二个卷已挂载到 **/tmp**，并且是临时卷。

3.5.9.2. 列出存储卷

要检查组件当前使用的存储卷，请运行以下命令：

```
$ odo storage list
```

输出示例：

```
$ odo storage list
The component 'nodejs-project-ufyy' has the following storage attached:
NAME      SIZE     PATH      STATE
store     1Gi      /data     Not Pushed
tempdir   2Gi      /tmp      Not Pushed
```

3.5.9.3. 删除存储卷

要删除存储卷，请运行以下命令：

```
$ odo storage delete
```

输出示例：

```
$ odo storage delete store -f
Deleted storage store from nodejs-project-ufyy

Please use `odo push` command to delete the storage from the cluster
```

在上例中，使用 **-f** 标志强制删除存储而无需询问用户权限。

3.5.9.4. 在特定容器中添加存储

如果您的 devfile 有多个容器，您可以使用 **odo storage create** 命令中的 **--container** 标志指定您要将存储附加到的容器。

以下示例是带有多个容器的 devfile 摘录：

```
components:
  - name: nodejs1
    container:
      image: registry.access.redhat.com/ubi8/nodejs-12:1-36
      memoryLimit: 1024Mi
      endpoints:
        - name: "3000-tcp"
          targetPort: 3000
      mountSources: true
  - name: nodejs2
    container:
      image: registry.access.redhat.com/ubi8/nodejs-12:1-36
      memoryLimit: 1024Mi
```

在示例中，有两个容器**nodejs1** 和 **nodejs2**。要将存储附加到 **nodejs2** 容器，请使用以下命令：

```
$ odo storage create --container
```

输出示例：

```
$ odo storage create store --path /data --size 1Gi --container nodejs2
✓  Added storage store to nodejs-testing-xnfg

Please use `odo push` command to make the storage accessible to the component
```

您可以使用 **odo storage list** 命令列出存储资源：

```
$ odo storage list
```

输出示例：

```
The component 'nodejs-testing-xnfg' has the following storage attached:
NAME      SIZE     PATH      CONTAINER     STATE
store     1Gi      /data     nodejs2       Not Pushed
```

3.5.10. common 标记

多数 **odo** 命令提供了以下标记：

表 3.1. odo 标记

命令	描述
--context	设置定义组件的上下文目录。
--project	设置组件的项目。默认为本地配置中定义的项目。如果没有可用的，则集群上的当前项目。
--app	设置组件的应用程序。默认为本地配置中定义的应用程序。如果没有可用的，则为 *app*。
--kubeconfig	如果不使用默认配置，请将 path 设置为 **kubeconfig** 值。
--show-log	使用此标志查看日志。
-f,--force	使用这个标志代表命令不会提示用户进行确认。
-v,--v	设置详细程度。如需更多信息，请参阅 odo 中的日志记录。
-h,--help	输出命令的帮助信息。

注意

某些命令可能不能使用一些标志。使用 **--help** 标志运行命令以获取所有可用标志的列表。

3.5.11. JSON 输出

输出内容的 **odo** 命令通常会接受 **-o json** 标志以 JSON 格式输出此内容，适用于其他程序来更轻松地解析此输出。

输出结构与 Kubernetes 资源类似，带有 **kind**, **apiVersion**, **metadata**, **spec**, 和**status** 字段。

List 命令会返回 **List** 资源，其中包含一个项列表的 **items**（或类似）字段，每个项目也与 Kubernetes 资源类似。

Delete 命令会返回一个 **Status** 资源；请参阅 Status Kubernetes 资源。

其他命令会返回与命令关联的资源，如 **Application**, **Storage**, **URL** 等。

当前接受 **-o json** 标记的命令的完整列表是：

命令	kind（版本）	列表项目的 kind（版本）	完整内容？
odo application describe	Application (odo.dev/v1alpha1)	*不适用*	否

命令	kind（版本）	列表项目的 kind（版本）	完整内容？
odo application list	List (odo.dev/v1alpha1)	Application (odo.dev/v1alpha1)	?
odo catalog list components	List (odo.dev/v1alpha1)	*缺少*	是
odo catalog list services	List (odo.dev/v1alpha1)	ClusterServiceVersion (operators.coreos.com/v1alpha1)	?
odo catalog describe component	*缺少*	*不适用*	是
odo catalog describe service	CRDDescription (odo.dev/v1alpha1)	*不适用*	是
odo component create	Component (odo.dev/v1alpha1)	*不适用*	是
odo component describe	Component (odo.dev/v1alpha1)	*不适用*	是
odo component list	List (odo.dev/v1alpha1)	Component (odo.dev/v1alpha1)	是
odo config view	DevfileConfiguration (odo.dev/v1alpha1)	*不适用*	是
odo debug info	OdoDebugInfo (odo.dev/v1alpha1)	*不适用*	是
odo env view	EnvInfo (odo.dev/v1alpha1)	*不适用*	是
odo preference view	PreferenceList (odo.dev/v1alpha1)	*不适用*	是
odo project create	Project (odo.dev/v1alpha1)	*不适用*	是
odo project delete	Status (v1)	*不适用*	是
odo project get	Project (odo.dev/v1alpha1)	*不适用*	是

命令	kind（版本）	列表项目的 kind（版本）	完整内容？
odo project list	List (odo.dev/v1alpha1)	Project (odo.dev/v1alpha1)	是
odo registry list	List (odo.dev/v1alpha1)	*缺少*	是
odo service create	服务	*不适用*	是
odo service describe	服务	*不适用*	是
odo service list	List (odo.dev/v1alpha1)	服务	是
odo storage create	Storage (odo.dev/v1alpha1)	*不适用*	是
odo storage delete	Status (v1)	*不适用*	是
odo storage list	List (odo.dev/v1alpha1)	Storage (odo.dev/v1alpha1)	是
odo url list	List (odo.dev/v1alpha1)	URL (odo.dev/v1alpha1)	是

第 4 章 HELM CLI

4.1. HELM 3 入门

4.1.1. 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。

4.1.1.1. 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。

4.1.2. 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。

4.1.2.1. 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"}
```

4.1.2.2. 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**。

4.1.2.3. 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**。

4.1.2.4. 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"}
```

4.1.3. 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
```

4.1.4. 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
```

4.2. 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。

4.2.1. 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。

   Helm Chart 仓库 CR 示例

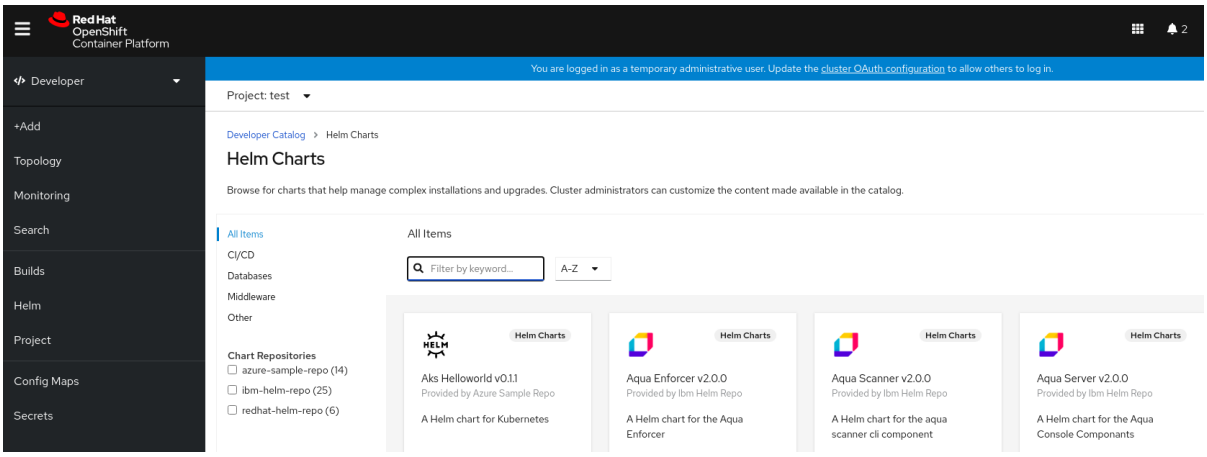
   ```
   apiVersion: helm.openshift.io/v1beta1
   kind: HelmChartRepository
   metadata:
     name: <name>
   spec:
    # optional name that might be used by console
    # name: <chart-display-name>
     connectionConfig:
       url: <helm-chart-repository-url>
   ```

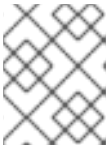
   例如，要添加 Azure 示例 chart 存储库，请运行：

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

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

图 4.1. 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。

4.2.2. 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
```

4.3. 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。

4.3.1. 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 5 章 用于 OPENSHIFT SERVERLESS 的 KNATIVE CLI

Knative(**kn**)CLI 在 OpenShift Container Platform 上启用了与 Knative 组件的简单交互。

5.1. 主要特性

Knative(**kn**)CLI 旨在使无服务器计算任务简单而简洁。Knative 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 Knative 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。

5.2. 安装 KNATIVE CLI

请参阅安装 Knative CLI。

第 6 章 PIPELINES CLI (TKN)

6.1. 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**。

6.1.1. 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
```

6.1.2. 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：

- Linux (x86_64, amd64)

```
# subscription-manager repos --enable="pipelines-1.4-for-rhel-8-x86_64-rpms"
```

- Linux on IBM Z and LinuxONE (s390x)

```
# subscription-manager repos --enable="pipelines-1.4-for-rhel-8-s390x-rpms"
```

- Linux on IBM Power Systems (ppc64le)

```
# subscription-manager repos --enable="pipelines-1.4-for-rhel-8-ppc64le-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
```

6.1.3. 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
```

6.1.4. 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
```

6.2. 配置 OPENSHIFT PIPELINES TKN CLI

配置 Red Hat OpenShift Pipelines **tkn** CLI 以启用 tab 自动完成功能。

6.2.1. 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。

6.3. OPENSHIFT PIPELINES TKN 参考

本节列出了基本的 **tkn** CLI 命令。

6.3.1. 基本语法

tkn [command or options] [arguments…]

6.3.2. 全局选项

--help, -h

6.3.3. 工具命令

6.3.3.1. tkn

tkn CLI 的主命令。

示例：显示所有选项

```
$ tkn
```

6.3.3.2. 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
```

6.3.3.3. version

输出 **tkn** CLI 的版本信息。

示例：检查 tkn 版本

```
$ tkn version
```

6.3.4. Pipelines 管理命令

6.3.4.1. pipeline

管理管道。

示例：显示帮助信息

```
$ tkn pipeline --help
```

6.3.4.2. pipeline delete

删除 Pipeline

示例：从命名空间中删除 mypipeline Pipeline

```
$ tkn pipeline delete mypipeline -n myspace
```

6.3.4.3. pipeline describe

描述管道。

示例：描述 mypipeline Pipeline

```
$ tkn pipeline describe mypipeline
```

6.3.4.4. pipeline list

列出管道。

示例： 显示 Pipelines 列表

```
$ tkn pipeline list
```

6.3.4.5. 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
```

6.3.4.6. pipeline start

运行 Pipeline。

示例： 启动 mypipeline Pipeline

```
$ tkn pipeline start mypipeline
```

6.3.5. PipelineRun 命令

6.3.5.1. pipelinerun

管理 PipelineRuns。

示例： 显示帮助信息

```
$ tkn pipelinerun -h
```

6.3.5.2. pipelinerun cancel

取消 PipelineRun。

示例：从命名空间中取消 mypipelinerun PipelineRun

```
$ tkn pipelinerun cancel mypipelinerun -n myspace
```

6.3.5.3. pipelinerun delete

删除 PipelineRun。

示例： 从命名空间中删除 PipelineRuns

```
$ tkn pipelinerun delete mypipelinerun1 mypipelinerun2 -n myspace
```

6.3.5.4. pipelinerun describe

描述 PipelineRun。

示例：描述命名空间中的 mypipelinerun PipelineRun

```
$ tkn pipelinerun describe mypipelinerun -n myspace
```

6.3.5.5. pipelinerun list

列出 PipelineRuns。

示例： 显示命名空间中的 PipelineRuns 列表

```
$ tkn pipelinerun list -n myspace
```

6.3.5.6. 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
```

6.3.6. 任务管理命令

6.3.6.1. task

管理任务。

示例： 显示帮助信息

```
$ tkn task -h
```

6.3.6.2. 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
```

6.3.6.3. 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
```

6.3.6.4. 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
```

6.3.6.5. 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
```

6.3.6.6. 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
```

6.3.7. TaskRun 命令

6.3.7.1. taskrun

管理 TaskRuns。

示例： 显示帮助信息

```
$ tkn taskrun -h
```

6.3.7.2. 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
```

6.3.7.3. 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
```

6.3.7.4. taskrun describe

描述 TaskRun。

示例：描述命名空间中的 mytaskrun TaskRun

```
$ tkn taskrun describe mytaskrun -n myspace
```

6.3.7.5. taskrun list

列出 TaskRuns。

示例： 列出命名空间中的所有 TaskRuns

```
$ tkn taskrun list -n myspace
```

6.3.7.6. 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
```

6.3.8. 条件管理命令

6.3.8.1. 条件

管理条件（Condition）。

示例： 显示帮助信息

```
$ tkn condition --help
```

6.3.8.2. 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
```

6.3.8.3. condition describe

描述条件。

示例：在命名空间中描述 mycondition1 Condition

```
$ tkn condition describe mycondition1 -n myspace
```

6.3.8.4. condition list

列出条件。

示例： 列出命名空间中的条件

```
$ tkn condition list -n myspace
```

6.3.9. Pipeline 资源管理命令

6.3.9.1. resource

管理管道资源。

示例： 显示帮助信息

```
$ tkn resource -h
```

6.3.9.2. 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。

6.3.9.3. 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
```

6.3.9.4. resource describe

描述管道资源。

示例：描述 myresource Pipeline 资源

```
$ tkn resource describe myresource -n myspace
```

6.3.9.5. 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
```

6.3.10. ClusterTask 管理命令

6.3.10.1. clustertask

管理 ClusterTasks。

示例： 显示帮助信息

```
$ tkn clustertask --help
```

6.3.10.2. clustertask delete

删除集群中的 ClusterTask 资源。

示例： 删除 mytask1 和 mytask2 ClusterTasks

```
$ tkn clustertask delete mytask1 mytask2
```

6.3.10.3. clustertask describe

描述 ClusterTask。

示例： 描述 mytask ClusterTask

```
$ tkn clustertask describe mytask1
```

6.3.10.4. clustertask list

列出 ClusterTasks。

示例： 列出 ClusterTasks

```
$ tkn clustertask list
```

6.3.10.5. clustertask start

启动 ClusterTasks。

示例： 启动 mytask ClusterTask

```
$ tkn clustertask start mytask
```

6.3.11. 触发器管理命令

6.3.11.1. eventlistener

管理 EventListeners。

示例： 显示帮助信息

```
$ tkn eventlistener -h
```

6.3.11.2. 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
```

6.3.11.3. eventlistener describe

描述 EventListener。

示例：描述命名空间中的 mylistener EventListener

```
$ tkn eventlistener describe mylistener -n myspace
```

6.3.11.4. eventlistener list

列出 EventListeners。

示例： 列出命名空间中的所有 EventListeners

```
$ tkn eventlistener list -n myspace
```

6.3.11.5. eventListener 日志

显示 EventListener 的日志。

示例： 在一个命名空间中显示 mylistener EventListener 的日志

```
$ tkn eventlistener logs mylistener -n myspace
```

6.3.11.6. triggerbinding

管理 TriggerBindings。

示例： 显示 TriggerBindings 帮助信息

```
$ tkn triggerbinding -h
```

6.3.11.7. 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
```

6.3.11.8. triggerbinding describe

描述 TriggerBinding。

示例：描述命名空间中的 mybinding TriggerBinding

```
$ tkn triggerbinding describe mybinding -n myspace
```

6.3.11.9. triggerbinding list

列出 TriggerBindings。

示例： 列出命名空间中的所有 TriggerBindings

```
$ tkn triggerbinding list -n myspace
```

6.3.11.10. triggertemplate

管理 TriggerTemplates。

示例： 显示 TriggerTemplate 帮助

```
$ tkn triggertemplate -h
```

6.3.11.11. 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`
```

6.3.11.12. triggertemplate describe

描述 TriggerTemplate。

示例： 描述命名空间中的 mytemplate TriggerTemplate

```
$ tkn triggertemplate describe mytemplate -n `myspace`
```

6.3.11.13. triggertemplate list

列出 TriggerTemplates。

示例： 列出命名空间中的所有 TriggerTemplates

```
$ tkn triggertemplate list -n myspace
```

6.3.11.14. clustertriggerbinding

管理 ClusterTriggerBindings。

示例： 显示 ClusterTriggerBindings 帮助信息

```
$ tkn clustertriggerbinding -h
```

6.3.11.15. clustertriggerbinding delete

删除 ClusterTriggerBinding。

示例： 删除 myclusterbinding1 和 myclusterbinding2 ClusterTriggerBindings

```
$ tkn clustertriggerbinding delete myclusterbinding1 myclusterbinding2
```

6.3.11.16. clustertriggerbinding describe

描述 ClusterTriggerBinding。

示例： 描述 myclusterbinding ClusterTriggerBinding

```
$ tkn clustertriggerbinding describe myclusterbinding
```

6.3.11.17. clustertriggerbinding list

列出 ClusterTriggerBindings。

示例： 列出所有 ClusterTriggerBindings

```
$ tkn clustertriggerbinding list
```

6.3.12. hub 互动命令

与 Tekton Hub 交互，以获取任务和管道等资源。

6.3.12.1. hub

与 hub 交互。

示例： 显示帮助信息

```
$ tkn hub -h
```

示例：与 hub API 服务器交互

```
$ tkn hub --api-server https://api.hub.tekton.dev
```

注意

对于每个示例，若要获取对应的子命令和标记，请运行 **tkn hub <command> --help**。

6.3.12.2. hub downgrade

对一个安装的资源进行降级。

示例：将 mynamespace 命名空间中的 mytask 任务降级到它的较旧版本

```
$ tkn hub downgrade task mytask --to version -n mynamespace
```

6.3.12.3. hub get

按名称、类型、目录和版本获取资源清单。

示例：从 tekton 目录中获取 myresource 管道或任务的特定版本的清单

```
$ tkn hub get [pipeline | task] myresource --from tekton --version version
```

6.3.12.4. hub info

按名称、类型、目录和版本显示资源的信息。

示例：显示 tekton 目录中有关 mytask 任务的特定版本的信息

```
$ tkn hub info task mytask --from tekton --version version
```

6.3.12.5. hub install

按类型、名称和版本从目录安装资源。

示例：从 mynamespace 命名空间中的 tekton 目录安装 mytask 任务的特定版本

```
$ tkn hub install task mytask --from tekton --version version -n mynamespace
```

6.3.12.6. hub reinstall

按类型和名称重新安装资源。

示例：从 mynamespace 命名空间中的 tekton 目录重新安装 mytask 任务的特定版本

```
$ tkn hub reinstall task mytask --from tekton --version version -n mynamespace
```

6.3.12.7. hub search

按名称、类型和标签组合搜索资源。

示例：搜索带有标签 cli的资源

```
$ tkn hub search --tags cli
```

6.3.12.8. hub upgrade

升级已安装的资源。

示例：将 mynamespace 命名空间中安装的 mytask 任务升级到新版本

```
$ tkn hub upgrade task mytask --to version -n mynamespace
```

第 7 章 OPM CLI

7.1. 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。

7.2. 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：RHEL 8 满足以下要求：
  - **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"}
```

7.3. 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 8 章 OPERATOR SDK

8.1. 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。

8.1.1. 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. 在 **4.7.23** 目录中下载 Linux 的 tarball 的最新版本。
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", ...
```

8.2. 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。

8.2.1. bundle

operator-sdk bundle 命令管理 Operator 捆绑包元数据。

8.2.1.1. validate

bundle validate 子命令会验证 Operator 捆绑包。

表 8.1. 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。

8.2.2. 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 8.2. 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**。

8.2.3. 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 8.3. completion 子命令

子命令	描述
bash	生成 bash completion。
zsh	生成 zsh completion。

表 8.4. 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
```

8.2.4. create

operator-sdk create 命令用于创建或 *scaffold* Kubernetes API。

8.2.4.1. 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 8.5. create api 标记

标记	描述
-h，--help	**run bundle** 子命令的帮助输出。

8.2.5. generate

operator-sdk generate 命令调用特定的生成器来生成代码或清单。

8.2.5.1. 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 8.6. 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。

8.2.5.2. kustomize

generate kustomize 子命令包含为 Operator 生成 Kustomize 数据的子命令。

8.2.5.2.1. 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 8.7. 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。

8.2.6. 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 8.8. 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）。

8.2.7. run

operator-sdk run 命令提供可在各种环境中启动 Operator 的选项。

8.2.7.1. bundle

run bundle 子命令使用 Operator Lifecycle Manager（OLM）以捆绑包格式部署 Operator。

表 8.9. 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。

8.2.7.2. 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 8.10. 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。

8.2.8. 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 8.11. 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。