OpenShift Container Platform 3.11

架构

OpenShift Container Platform 3.11 架构信息

Last Updated: 2023-03-25

OpenShift Container Platform 3.11 架构

OpenShift Container Platform 3.11 架构信息

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

了解 OpenShift Container Platform 3.11 的架构，包括基础架构和核心组件。这些主题还包括身份验证、网络和源代码管理。

目录

第 1 章 概述

OpenShift v3 是一个分层的系统，它用于公开底层 Docker 格式的容器镜像和 Kubernetes 概念，其目的是使开发人员可以轻松地开发和部署应用程序。例如，安装 Ruby、推送代码和添加 MySQL。

与 OpenShift v2 相比，它在创建模型的所有方面后，提供了更加灵活的配置。作为单独的对象，应用程序的概念被移除以更灵活的"服务"组成，允许两个 web 容器重复使用数据库或将数据库直接公开给网络边缘。

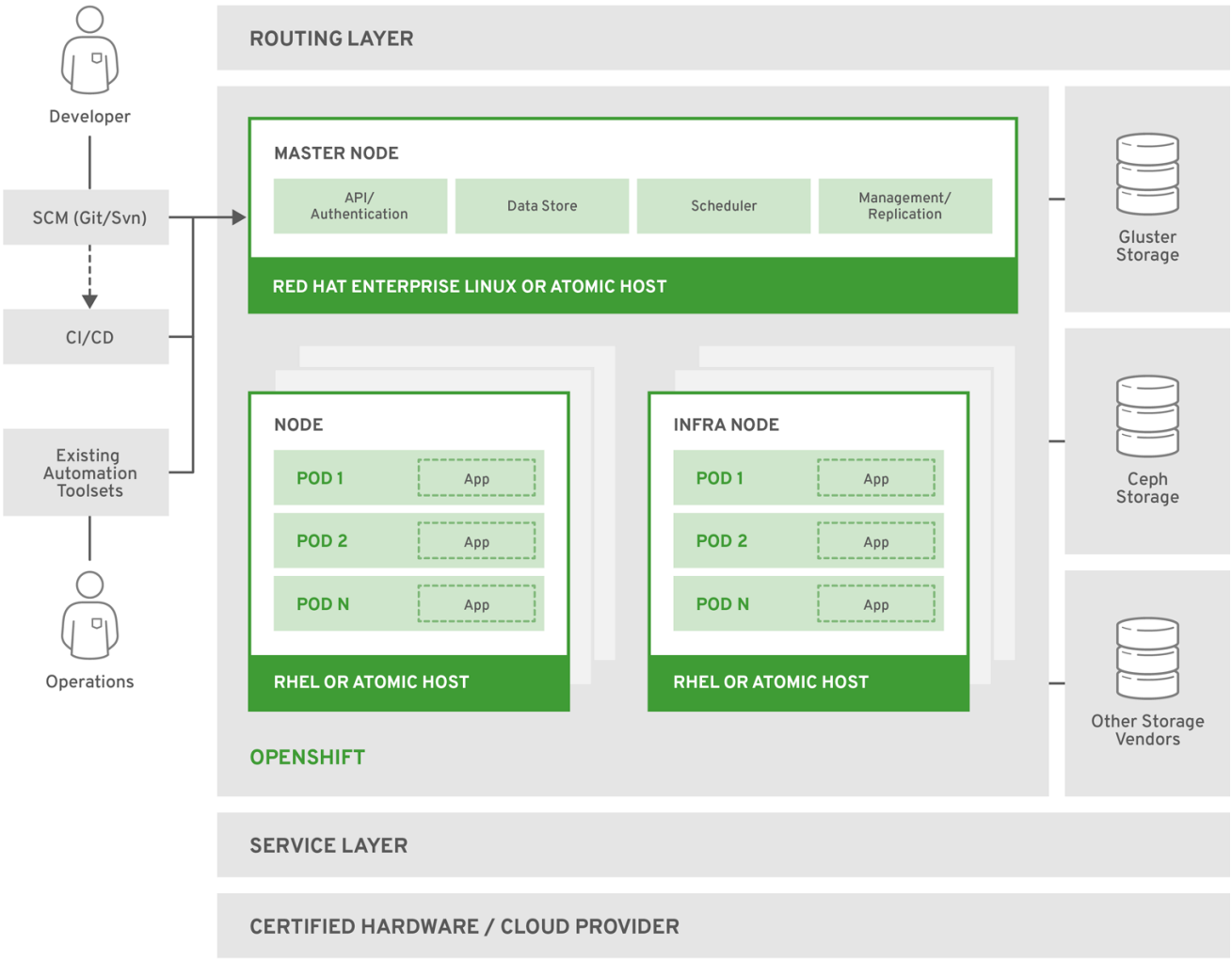
1.1. 什么是层？

Docker 服务为打包和创建基于 Linux 的轻量级容器镜像提供了一个抽象层。Kubernetes 提供 集群管理功能，并可以在多个主机上编配容器。

OpenShift Container Platform 增加了：

- 面向开发人员的源代码管理、构建和部署
- 在系统流时，以扩展的方式管理和提升镜像
- 大规模管理应用程序
- 用于组织大型开发人员组织的团队和用户跟踪
- 支持集群的网络基础架构

图 1.1. OpenShift Container Platform 架构概述



如需有关架构概述中的节点类型的更多信息，请参阅 Kubernetes 基础架构。

1.2. OPENSHIFT CONTAINER PLATFORM 架构是什么？

OpenShift Container Platform 有一个基于微服务的架构，具有可一起工作的较小且分离的单元。它运行在 Kubernetes 集群 之上，其中包含有关 etcd 中存储的对象的数据，它是一个可靠的集群键值存储。这些服务按功能划分：

- REST API，用于公开核心对象。
- 控制器读取这些 API，对其他对象应用更改，以及报告状态或写入对象。

用户调用 REST API 以更改系统的状态。控制器使用 REST API 来读取用户所需状态，然后尝试让系统的其他部分保持同步。例如，当用户请求创建 "build" 对象的构建时。构建控制器会看到已创建了新构建，并在集群上运行一个进程来执行该构建。构建完成后，控制器通过 REST API 更新构建对象，用户会看到其构建已经完成。

控制器模式意味着 OpenShift Container Platform 中的大部分功能是可扩展的。构建运行和启动的方式可以独立自定义管理镜像的方式，或者如何部署。控制器正在执行系统的"业务逻辑"，采取用户操作并将其转换为现实。通过自定义这些控制器或将其替换为您自己的逻辑，可以实施不同的行为。从系统管理的角度来看，这也意味着 API 可用于根据重复的计划编写常见的管理操作。这些脚本也是监视更改和采取行动的控制器。OpenShift Container Platform 使可以采用第一类行为来自定义集群。

要实现这个目标，控制器利用系统的一个可靠的更改流，将其系统视图与用户执行的操作同步。此事件流

会从 etcd 推送到 REST API，然后在更改后马上推送到控制器，因此更改可以快速高效地通过系统进行传输。但是，由于任何时候都可能会出现故障，控制器还必须能够在启动时获得系统的最新状态，并确认所有内容都处于正确的状态。这个重新同步（resynchronization）功能很重要，因为这意味着即使出现问题，Operator 也可以重启受影响的组件，系统会在继续前双重检查所有信息。系统最终应聚合到用户的意图，因为控制器总是可以使系统进行同步。

1.3. OPENSHIFT CONTAINER PLATFORM 如何安全？

OpenShift Container Platform 和 Kubernetes API 会验证带有凭证的用户的身份，然后根据其角色授权它们。开发人员和管理员可以通过多种方法进行身份验证，主要通过 OAuth 令牌和 X.509 客户端证书进行身份验证。OAuth 令牌使用 JSON Web Algorithm *RS256* 签名，这是带有 SHA-256 的 RSA 签名算法 PKCS#1 v1.5。

开发人员（系统客户端）通常从 **oc** 或 web 控制台 等 客户端程序发出 REST API 调用，并使用 OAuth bearer 令牌进行大部分通信。基础架构组件（如节点）使用由系统中包含其身份的客户端证书。在容器中运行的基础架构组件使用与其 服务帐户关联的 令牌来连接到 API。

授权在 OpenShift Container Platform 策略引擎中处理，它定义诸如 "create pod" 或 "list services" 的操作，并将它们分组到策略文档中的角色。角色通过用户或组标识符绑定到用户或组。当用户或服务帐户尝试操作时，策略引擎会检查一个或多个分配给该用户的角色（例如，集群管理员或当前项目的管理员），然后再继续。

由于集群中运行的每个容器都与服务帐户关联，因此也可以将 secret 与这些服务帐户关联，并将其自动传送到容器中。这可让基础架构管理用于拉取和推送镜像、构建和部署组件的 secret，同时还允许应用程序代码轻松利用这些 secret。

1.3.1. TLS 支持

所有与 REST API 的通信频道以及 etcd 和 API 服务器等 master 组件 之间都使用 TLS 保护。TLS 提供具有 X.509 服务器证书和公钥基础架构的强大加密、数据完整性和验证服务器。默认情况下，会为每个 OpenShift Container Platform 部署创建一个新的内部 PKI。内部 PKI 使用 2048 位 RSA 密钥和 SHA-256 签名。也支持公共主机的 自定义证书。

OpenShift Container Platform 使用 Golang 的 **crypto/tls** 的标准库实现，且不依赖于任何外部加密和 TLS 库。另外，客户端依赖于用于 GSSAPI 验证和 OpenPGP 签名的外部库。GSSAPI 通常由 MIT Kerberos 或 Heimdal Kerberos 提供，它们都使用 OpenSSL 的 libcrypto。OpenPGP 签名验证由 libgpgme 和 GnuPG 处理。

不安全的版本 SSL 2.0 和 SSL 3.0 不受支持，且不可用。OpenShift Container Platform 服务器和 **oc** 客户端默认只提供 TLS 1.2。TLS 1.0 和 TLS 1.1 可以在服务器配置中启用。服务器和客户端都首选使用经过身份验证的加密算法和完美转发保密性实现现代加密套件。禁用使用已弃用和不安全的算法（如 RC4、3DES 和 MD5）的密码套件。有些内部客户端（例如，LDAP 身份验证）限制了启用了 1.2 和更多密码套件的 TLS 1.0 的设置。

表 1.1. 支持的 TLS 版本

TLS 版本	OpenShift Container Platform Server	oc Client	其他客户端
SSL 2.0	不支持	不支持	不支持
SSL 3.0	不支持	不支持	不支持
TLS 1.0	没有 [1]	没有 [1]	或许 [2]

TLS 版本	OpenShift Container Platform Server	oc Client	其他客户端
TLS 1.1	没有 [1]	没有 [1]	或许 [2]
TLS 1.2	**是**	**是**	**是**
TLS 1.3	N/A [3]	N/A [3]	N/A [3]

1. 默认禁用，但在服务器配置中可以启用。
2. 一些内部客户端，如 LDAP 客户端。
3. TLS 1.3 仍在开发中。

以下列出的 OpenShift Container Platform 服务器和 **oc** 客户端的启用密码套件按首选顺序排序：

- **TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305**
- **TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305**
- **TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256**
- **TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256**
- **TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384**
- **TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384**
- **TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256**
- **TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256**
- **TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA**
- **TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA**
- **TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA**
- **TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA**
- **TLS_RSA_WITH_AES_128_GCM_SHA256**
- **TLS_RSA_WITH_AES_256_GCM_SHA384**
- **TLS_RSA_WITH_AES_128_CBC_SHA**
- **TLS_RSA_WITH_AES_256_CBC_SHA**

第 2 章 基础架构组件

2.1. KUBERNETES 基础架构

2.1.1. 概述

在 OpenShift Container Platform 中，Kubernetes 在一组容器或主机间管理容器化应用，并提供部署、维护和应用程序的扩展机制。容器运行时软件包、实例化和运行容器化应用。Kubernetes 集群由一个或多个 master 和一组节点组成。

您可以选择将 master 配置为高可用性 (HA)，以确保集群没有单点故障。

注意

OpenShift Container Platform 使用 Kubernetes 1.11 和 Docker 1.13.1。

2.1.2. Master

master 是包含 control plane 组件的主机或主机，包括 API 服务器、控制器管理器服务器和 etcd。master 用于管理其 Kubernetes 集群中的节点，并调度 pod 以便在这些节点上运行。

表 2.1. 主组件

组件	描述
API Server	Kubernetes API 服务器验证并配置 pod、服务和复制控制器的数据。它还将 pod 分配给节点，并将 pod 信息与服务配置同步。
etcd	etcd 存储持久 master 状态，而其他组件会监视 etcd 的更改，以自行进入所需状态。etcd 可以配置为高可用性，通常使用 2n+1 对等服务部署。
Controller Manager Server	控制器管理器服务器监视 etcd 是否有复制控制器对象的更改，然后使用 API 强制执行所需的状态。多个这样的过程会创建在某个时间点上有一个活跃群首的集群。
HAProxy	可选，用于配置高可用性 master，使用 **native（原生）** 方法在 API 主端点之间平衡负载。集群安装过程可以使用 **native** 方法为您配置 HAProxy。或者，您可以使用 **原生** 方法，但预先配置您自己的负载均衡器选择。

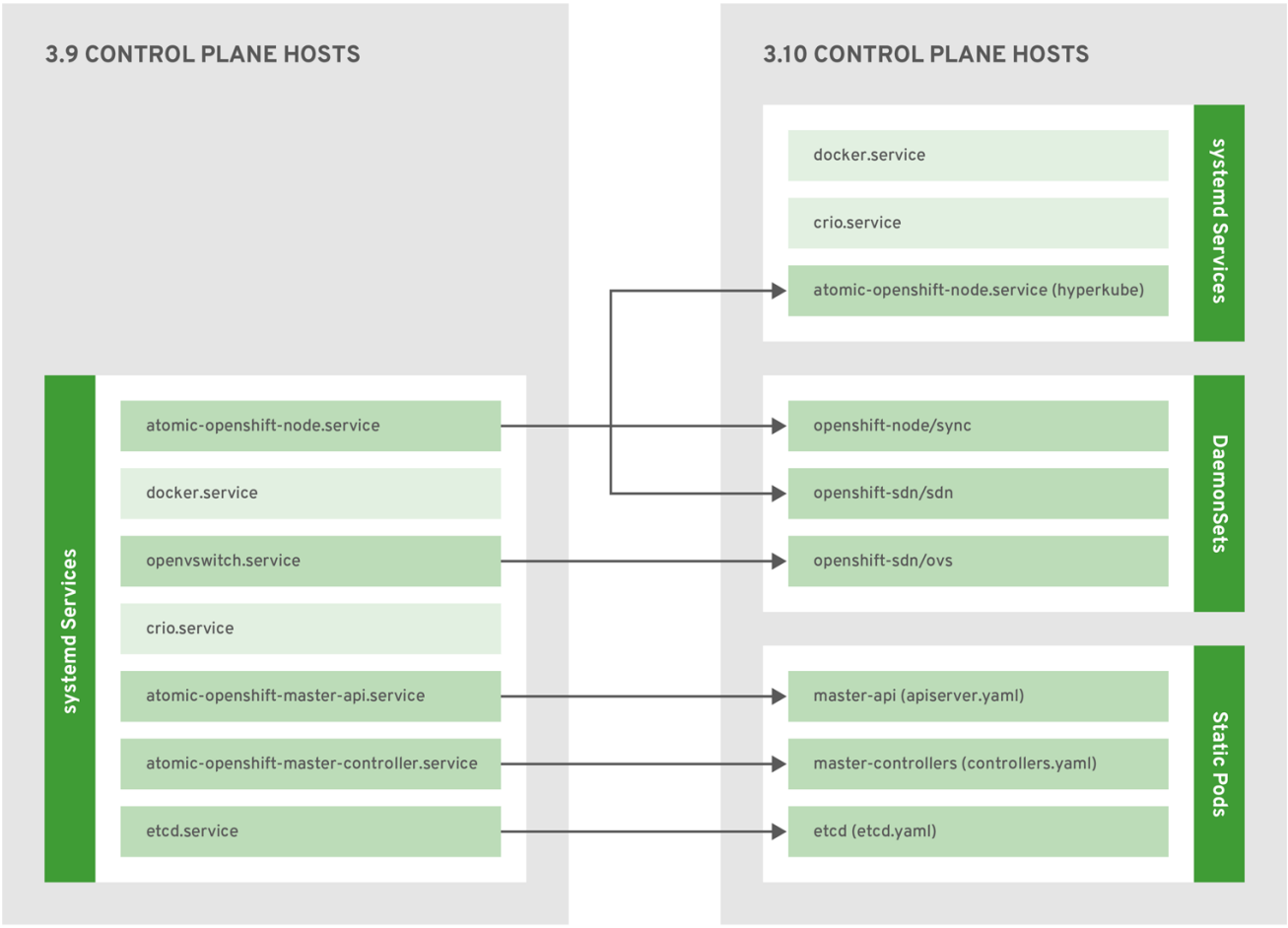
2.1.2.1. Control Plane 静态 Pod

核心 control plane 组件、API 服务器和控制器管理器组件，作为 kubelet 操作 *的静态 pod* 运行。

对于在同一主机上还存在 etcd 的 master，etcd 也会变为静态 pod。在不是 master 的 etcd 主机上仍然支持基于 RPM 的 etcd。

另外，节点组件 **openshift-sdn** 和 **openvswitch** 现在可以使用 DaemonSet 而不是 **systemd** 服务运行。

图 2.1. control plane 主机架构更改



即使作为静态 pod 运行 control plane 组件，master 主机仍然会从 */etc/origin/master/master-config.yaml* 文件中提供其配置，如 Master 和 Node Configuration 所述。

启动序列概述

Hyperkube 是一个包含所有 Kubernetes 的二进制文件（**kube-apiserver**, **controller-manager**, **scheduler**, **proxy**, 和 **kubelet**）。在启动时，**kubelet** 创建 **kubepods.slice**。接下来，**kubelet** 在 **kubepods.slice** 中创建 QoS 级别片段 **burstable.slice** 和 **best-effort.slice**。当 pod 启动时，**kubelet** 使用格式为 pod< **UUID-of-pod>.slice** 的 pod 级别片段，并将该路径传递给容器运行时接口(CRI)的另一端的运行时。然后，Docker 或 CRI-O 会在 pod 级别片段中创建容器级别的片段。

镜像 Pod

master 节点上的 kubelet 会自动在 API 服务器上为每个 control plane 静态 *pod 创建镜像* pod，以便 **kube-system** 项目中的集群可见。默认情况下，**openshift-ansible** 安装程序会安装这些静态 pod 的清单，位于 master 主机上的 */etc/origin/node/pods* 目录中。

这些 pod 定义了以下 **hostPath** 卷：

/etc/origin/master	包含所有证书、配置文件和 *admin.kubeconfig* 文件。
/var/lib/origin	包含二进制文件的卷和潜在的内核转储。

/etc/origin/cloudprovider	包含特定于云供应商的配置（AWS、Azure 等）。
/usr/libexec/kubernetes/kubelet-plugins	包含其他第三方卷插件。
/etc/origin/kubelet-plugins	包含系统容器的其他第三方卷插件。

您可以在静态 pod 上执行的操作集合有限。例如：

```
$ oc logs master-api-<hostname> -n kube-system
```

返回 API 服务器的标准输出。然而：

```
$ oc delete pod master-api-<hostname> -n kube-system
```

不会实际删除 pod。

再如，集群管理员可能需要执行一个常见操作，如增加 API 服务器的**日志级别**，以便在出现问题时提供更详细的数据。您必须编辑 */etc/origin/master/master.env* 文件，其中 **OPTIONS** 变量中的 **--loglevel** 参数可以被修改，因为此值被传递给容器中运行的进程。更改需要重启容器中运行的进程。

重启主服务
要重启在 control plane 静态 pod 中运行的 control plane 服务，请在 master 主机上使用 **master-restart** 命令。

重启 master API：

```
# master-restart api
```

重启控制器：

```
# master-restart controllers
```

重启 etcd：

```
# master-restart etcd
```

查看主服务日志
要查看在 control plane 静态 pod 中运行的 control plane 服务的日志，请为对应的组件使用 **master-logs** 命令：

```
# master-logs api api
```

```
# master-logs controllers controllers
```

```
# master-logs etcd etcd
```

2.1.2.2. 高可用性 Master

您可以选择将 master 配置为高可用性(HA)，以确保集群没有单点故障。

要减少有关 master 可用性的问题，建议进行两个操作：

1. 在重新创建 master 是应创建一个 runbook 条目。runbook 条目是为任何高可用服务需要重新停止。其它解决方案只会控制运行手册必须得到修正的频率。例如，冷备用 master 主机可以正确处理满足 SLA，该 SLA 在创建新应用程序或故障应用程序组件的恢复需要几分钟以上。

2. 使用高可用性解决方案配置 master，并确保集群没有单点故障。集群安装文档 **使用原生** HA 方法和配置 HAProxy 提供了具体示例。您还可以采用概念，并使用 **原生** 方法而不是 HAProxy 将它们应用于现有的 HA 解决方案。

注意

在生产环境的 OpenShift Container Platform 集群中，您必须维护 API 服务器负载均衡器的高可用性。如果 API 服务器负载均衡器不可用，节点无法报告其状态，则所有 pod 都会标记为 dead，pod 的端点也会从服务中删除。

除了为 OpenShift Container Platform 配置 HA 之外，还必须为 API 服务器负载均衡器单独配置 HA。要配置 HA，最好集成一个企业负载均衡器(LB)，例如 F5 Big-IP™ 或 Citrix Netscaler™ 设备。如果此类解决方案不可用，可以运行多个 HAProxy 负载均衡器，并使用 Keepalived 为 HA 提供浮动虚拟 IP 地址。但是，不建议在生产环境中使用这个解决方案。

当在 HAProxy 中 **使用原生** HA 方法时，master 组件有以下可用性：

表 2.2. HAProxy 的可用性列表

角色	样式	备注
etcd	Active-active	具有负载平衡的完全冗余部署。可以安装在单独的主机上，也可以将 collocated 安装到 master 主机上。
API Server	Active-active	由 HAProxy 管理.
Controller Manager Server	Active-passive	一个实例被选为集群领导。
HAProxy	Active-passive	在 API 主（master）端点之间进行负载平衡。

虽然集群 etcd 需要奇数个主机的仲裁数，但 master 服务没有奇数个主机的仲裁数或要求。但是，因为您需要至少两个 master 服务用于 HA，因此在找到 master 服务和 etcd 时，通常要维护一个统一奇数的主机。

2.1.3. 节点

节点为容器提供运行时环境。Kubernetes 群集中的每个节点都需要由主控机管理的服务。节点也具有运行 pod 所需的服务，包括容器运行时、kubelet 和服务代理。

OpenShift Container Platform 从云供应商、物理系统或虚拟系统创建节点。Kubernetes 与代表这些节点的节点对象进行交互。主控机（master）使用来自节点对象的信息执行健康检查，以此验证节点。在通过健康检查前，节点会被忽略，master 会继续检查节点，直到节点有效。Kubernetes 文档包含有关节点状态和管理的更多信息。

管理员可以使用 CLI 在 OpenShift Container Platform 实例中管理节点。要在启动节点服务器时定义完整的配置和安全选项，请使用 专用节点配置文件。

重要

有关推荐的节点数，请参阅集群限制部分。

2.1.3.1. kubelet

每个节点都有一个 kubelet，它更新由容器清单指定的节点，它是一个描述 pod 的 YAML 文件。kubelet 使用一组清单来确保其容器已启动并且它们继续运行。

容器清单可以通过以下方法提供给 kubelet：

- 命令行中每 20 秒检查一次的文件路径。
- 一个 HTTP 端点会传递到命令行，它每 20 秒检查一次。
- kubelet 监视 etcd 服务器，如 ***/registry/hosts/$(hostname -f)***，并作用于任何更改。
- kubelet 侦听 HTTP 并响应简单的 API 来提交新清单。

2.1.3.2. 服务代理

每个节点还运行一个简单的网络代理，它反映了节点上 API 中定义的服务。这允许节点在一组后端上执行简单的 TCP 和 UDP 流转发。

2.1.3.3. 节点对象定义

以下是 Kubernetes 中的节点对象定义示例：

```
apiVersion: v1 (1)
kind: Node (2)
metadata:
  creationTimestamp: null
  labels: (3)
    kubernetes.io/hostname: node1.example.com
  name: node1.example.com (4)
spec:
  externalID: node1.example.com (5)
status:
  nodeInfo:
    bootID: ""
    containerRuntimeVersion: ""
    kernelVersion: ""
    kubeProxyVersion: ""
    kubeletVersion: ""
```

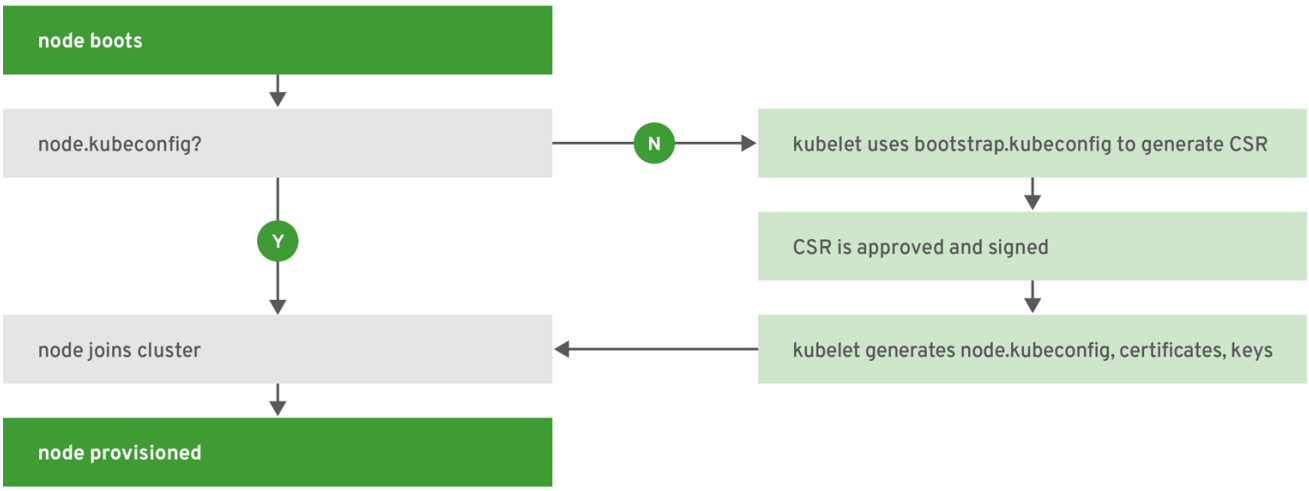
```
machineID: ""
osImage: ""
systemUUID: ""
```

1. **apiVersion** 定义要使用的 API 版本。
2. 将 **kind** 设置为 **Node** 代表这是一个节点对象的定义。
3. **metadata. labels** 列出添加到节点的任何标签。
4. **metadata.name** 是定义节点对象名称的必要值。在运行 **oc get nodes** 命令时，这个值显示在 **NAME** 列中。
5. **spec.externalID** 定义了用于访问节点的完全限定域名。如果为空，默认为 **metadata.name** 值。

2.1.3.4. 节点引导

节点的配置从 master 启动，这意味着节点会从 master 拉取其预定义配置和服务器证书。这可通过减少节点之间的区别以及集中更多配置并让集群在所需状态进行聚合，从而加快节点启动速度。证书轮转和集中式证书管理会被默认启用。

图 2.2. 节点 bootstrap 工作流概述



当节点服务启动时，节点会在加入集群前检查 */etc/origin/node/node.kubeconfig* 文件和其他节点配置文件是否存在。如果不存在，节点将从 master 中拉取配置，然后加入集群。

ConfigMap 用于在集群中存储节点配置，这会在节点主机上的 */etc/origin/node/node-config.yaml* 填充配置文件。有关默认节点组及其 ConfigMap 集合的定义，请参阅在安装集群中定义节点组和主机映射。

节点 Bootstrap 工作流
自动节点 bootstrap 过程使用以下工作流：

1. 默认情况下，会在集群安装过程中创建一组 **clusterrole**、**clusterrolebinding** 和 **serviceaccount** 对象以用于节点引导：
   - **system:node-bootstrapper** 集群角色用于在节点引导期间创建证书签名请求(CSR)：

     ```
     $ oc describe clusterrole.authorization.openshift.io/system:node-bootstrapper
     ```

输出示例

```
Name:   system:node-bootstrapper
Created:  17 hours ago
Labels:   kubernetes.io/bootstrapping=rbac-defaults
Annotations:  authorization.openshift.io/system-only=true
  openshift.io/reconcile-protect=false
Verbs   Non-Resource URLs Resource Names API Groups  Resources
[create get list watch] []   []  [certificates.k8s.io] [certificatesigningrequests]
```

- 以下 **node-bootstrapper** 服务帐户是在 **openshift-infra** 项目中创建的：

```
$ oc describe sa node-bootstrapper -n openshift-infra
```

输出示例

```
Name:              node-bootstrapper
Namespace:         openshift-infra
Labels:            <none>
Annotations:       <none>
Image pull secrets:  node-bootstrapper-dockercfg-f2n8r
Mountable secrets:   node-bootstrapper-token-79htp
                   node-bootstrapper-dockercfg-f2n8r
Tokens:            node-bootstrapper-token-79htp
                   node-bootstrapper-token-mqn2q
Events:            <none>
```

- 以下 **system:node-bootstrapper** 集群角色绑定用于节点 bootstrapper 集群角色和服务帐户：

```
$ oc describe clusterrolebindings system:node-bootstrapper
```

输出示例

```
Name:   system:node-bootstrapper
Created:  17 hours ago
Labels:   <none>
Annotations:  openshift.io/reconcile-protect=false
Role:   /system:node-bootstrapper
Users:   <none>
Groups:   <none>
ServiceAccounts: openshift-infra/node-bootstrapper
Subjects:  <none>
Verbs   Non-Resource URLs Resource Names API Groups  Resources
[create get list watch] []   []  [certificates.k8s.io] [certificatesigningrequests]
```

2. 另外，在集群安装过程中，**openshift-ansible** 安装程序会在 */etc/origin/master* 目录中创建 OpenShift Container Platform 证书颁发机构和各种其他证书、密钥和 ***kubeconfig*** 文件。注意的两个文件有：

/etc/origin/master/admin.kubeconfig	使用 **system:admin** 用户。
/etc/origin/master/bootstrap.kubeconfig	用于不是 master 节点的 bootstrap 节点。

a. 在安装程序使用 **node-bootstrapper** 服务帐户时，会创建 */etc/origin/master/bootstrap.kubeconfig*，如下所示：

```
$ oc --config=/etc/origin/master/admin.kubeconfig \
    serviceaccounts create-kubeconfig node-bootstrapper \
    -n openshift-infra
```

b. 在 master 节点上，*/etc/origin/master/admin.kubeconfig* 用作引导文件，并复制到 */etc/origin/node/boostrap.kubeconfig*。在另外一个非 master 节点上，*/etc/origin/master/bootstrap.kubeconfig* 文件会复制到每个节点主机上的 */etc/origin/node/boostrap.kubeconfig* 的所有其他节点上。

c. 然后，使用 **--bootstrap-kubeconfig** 标记将 */etc/origin/master/bootstrap.kubeconfig* 传递给 kubelet，如下所示：

```
--bootstrap-kubeconfig=/etc/origin/node/bootstrap.kubeconfig
```

3. kubelet 首先使用提供的 */etc/origin/node/bootstrap.kubeconfig* 文件启动。在初始内部连接后，kubelet 创建证书签名请求(CSR)并将其发送到 master。

4. CSR 通过控制器管理器（特别是证书签名请求）进行验证和批准。如果批准，kubelet 客户端和服务器证书会在 */etc/origin/node/ceritificates* 目录中创建。例如：

```
# ls -al /etc/origin/node/certificates/
```

输出示例

```
total 12
drwxr-xr-x. 2 root root  212 Jun 18 21:56 .
drwx------. 4 root root  213 Jun 19 15:18 ..
-rw-------. 1 root root 2826 Jun 18 21:53 kubelet-client-2018-06-18-21-53-15.pem
-rw-------. 1 root root 1167 Jun 18 21:53 kubelet-client-2018-06-18-21-53-45.pem
lrwxrwxrwx. 1 root root   68 Jun 18 21:53 kubelet-client-current.pem ->
/etc/origin/node/certificates/kubelet-client-2018-06-18-21-53-45.pem
-rw-------. 1 root root 1447 Jun 18 21:56 kubelet-server-2018-06-18-21-56-52.pem
lrwxrwxrwx. 1 root root   68 Jun 18 21:56 kubelet-server-current.pem ->
/etc/origin/node/certificates/kubelet-server-2018-06-18-21-56-52.pem
```

5. 在 CSR 批准后，*node.kubeconfig* 文件会在 */etc/origin/node/node.kubeconfig* 中创建。

6. kubelet 使用 */etc/origin/node/node.kubeconfig* 文件以及 */etc/origin/node/certificates/* 目录中的证书重启，直到它准备好加入集群。

节点配置工作流
为节点配置提供使用以下工作流：

1. 最初，节点的 kubelet 是在节点置备时创建的 */etc/origin/node/* 目录中的 ***bootstrap-node-config.yaml*** 启动。

2. 在每个节点上，节点服务文件使用 */usr/local/bin/* 目录中的本地脚本 ***openshift-node*** 来通过提供的 ***bootstrap-node-config.yaml*** 启动 kubelet。

3. 在每个 master 上，目录 */etc/origin/node/pods* 包含 **apiserver** 的 pod 清单，**控制器和 etcd** 在 master 上作为静态 pod 创建。

4. 在集群安装过程中，会创建一个同步 DaemonSet，在每个节点上创建一个同步 pod。sync（同步）pod 会监控文件 */etc/sysconfig/atomic-openshift-node* 中的更改。它专门用于监视要设置的 **BOOTSTRAP_CONFIG_NAME**。**BOOTSTRAP_CONFIG_NAME** 由 **openshift-ansible** 安装程序设置，它是根据节点所属节点配置组的 ConfigMap 名称。
   默认情况下，安装程序会创建以下节点配置组：

   - **node-config-master**
   - **node-config-infra**
   - **node-config-compute**
   - **node-config-all-in-one**
   - **node-config-master-infra**

   在 **openshift-node** 项目中为每个组创建一个 ConfigMap。

5. 同步 pod 根据 **BOOTSTRAP_CONFIG_NAME** 中设置的值提取适当的 ConfigMap。

6. 同步 pod 将 ConfigMap 数据转换为 kubelet 配置，并为该节点主机创建一个 */etc/origin/node/node-config.yaml*。如果对此文件进行了更改（或者是文件的初始创建），则 kubelet 会重启。

修改节点配置
通过编辑 **openshift-node** 项目中的相应 ConfigMap 来修改节点的配置。不要直接修改 */etc/origin/node/node-config.yaml*。

例如，对于 **node-config-compute** 组中的节点，使用以下内容编辑 ConfigMap：

```
$ oc edit cm node-config-compute -n openshift-node
```

2.2. 容器 REGISTRY

2.2.1. 概述

OpenShift Container Platform 可以使用实施容器镜像 registry API 的任何服务器作为镜像源，包括 Docker Hub、由第三方运行的私有 registry 和集成的 OpenShift Container Platform registry。

2.2.2. 集成的 OpenShift Container Registry

OpenShift Container Platform 提供了一个集成的容器镜像 registry，称为 *OpenShift Container Registry* (OCR)，它可按需自动置备新的镜像存储库。这为用户提供一个内置位置，供其应用构建用于推送生成的镜像。

每当一个新镜像推送到 OCR 时，registry 就会通知 OpenShift Container Platform，同时传递有关新镜像的所有信息，如命名空间、名称和镜像元数据。OpenShift Container Platform 的不同部分对新镜像做出反应，创建新的构建和部署。

OCR 也可以作为独立组件部署，它只作为容器镜像 registry，而无需构建和部署集成。详情请参阅安装独立 OpenShift Container Registry 部署的内容。

2.2.3. 第三方 registry

OpenShift Container Platform可以使用由第三方 registry 提供的镜像创建容器，但是这些registry 可能不会象OpenShift Container Platform中集成的registry一样提供相同的镜像通知支持。在这种情况下，OpenShift Container Platform将在创建镜像流时从远程 registry 中获取 tag。**oc import-image <stream>**就可以更新获取的 tag。当检测到新的镜像时，以前的构建和部署将会被重新创建。

2.2.3.1. 身份验证

OpenShift Container Platform 使用用户提供的凭证与 registry 进行联系来访问私有镜像仓库。这样，OpenShift Container Platform就可以对私有仓库进行镜像的 push 和 pull 操作。身份验证包含更多信息。

2.2.4. Red Hat Quay registry

Red Hat Quay 为您提供了一个企业级的容器镜像 registry。它可以作为一个托管的服务，也可以在您自己的数据中心或环境中安装它。Red Hat Quay中的高级 registry 功能包括跨区域复制、镜像扫描及镜像回滚（roll back）功能。

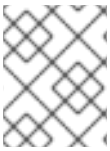
请通过 Quay.io 网站设置您自己的托管 Quay registry 帐户。之后，按照 Quay 教程登录到 Quay registry，并开始管理镜像。另外，请参阅 Red Hat Quay 入门，以了解有关设置您自己的 Red Hat Quay registry 的信息。

您可以象访问其他远程镜像 registry 一样，通过 OpenShift Container Platform 访问您的 Red Hat Quay registry。要了解如何设置将凭证作为安全 registry 访问 Red Hat Quay，请参阅 允许 Pod 从其他安全 registry 中引用镜像。

2.2.5. 启用身份验证的 Red Hat Registry

Red Hat Container Catalog （**registry.access.redhat.com**)是一个托管的镜像 registry，通过它可以获得所需的容器镜像。OpenShift Container Platform 3.11 Red Hat Container Catalog 从 **registry.access.redhat.com** 移到 **registry.redhat.io**。

新的 registry（**Registry.redhat.io**）需要进行身份验证才能访问OpenShift Container Platform上的镜像及内容。当迁移到新registry后，现有的registry仍将在一段时间内可用。



注意

OpenShift Container Platform从**Registry.redhat.io**中提取（pull）镜像，因此需要配置集群以使用它。

新registry使用标准的OAuth机制进行身份验证：

- **身份验证令牌。**令牌（token）是服务帐户，由管理员生成。系统可以使用它们与容器镜像registry进行身份验证。服务帐户不受用户帐户更改的影响，因此使用令牌进行身份验证是一个可靠且具有弹性的方法。这是生产环境集群中唯一受支持的身份验证选项。

- **Web用户名和密码。**这是用于登录到诸如**access.redhat.com**之类的资源的标准凭据集。虽然可以在OpenShift Container Platform上使用此身份验证方法，但在生产环境部署中不支持此方法。此身份验证方法应该只限于在OpenShift Container Platform之外的独立项目中使用。

您可以在 **docker login** 中使用您的凭证（用户名和密码，或身份验证令牌）来访问新 registry 中的内容。

所有镜像流均指向新的registry。由于新 registry 需要进行身份验证才能访问，因此 OpenShift 命名空间中有一个名为 **imagestreamsecret** 的新机密。

您需要将凭据放在两个位置：

- **OpenShift 命名空间** 。您的凭据必须存在于 OpenShift 命名空间中，以便 OpenShift 命名空间中的镜像流可以导入。

- **您的主机。**您的凭据必须存在于主机上，因为在抓取（pull）镜像时，Kubernetes会使用主机中的凭据。

访问新 registry:

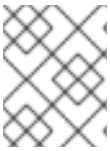
- 验证镜像导入 secret(**imagestreamsecret**)是否位于 OpenShift 命名空间中。该 secret 具有允许您访问新 registry 的凭证。

- 验证所有集群节点都有一个 **/var/lib/origin/.docker/config.json**，可以从 master 中复制，供您访问红帽 registry。

2.3. WEB 控制台

2.3.1. 概述

OpenShift Container Platform Web控制台是可从Web浏览器访问的用户界面。开发人员可以使用 Web 控制台来视觉化、浏览和管理 项目 的内容。



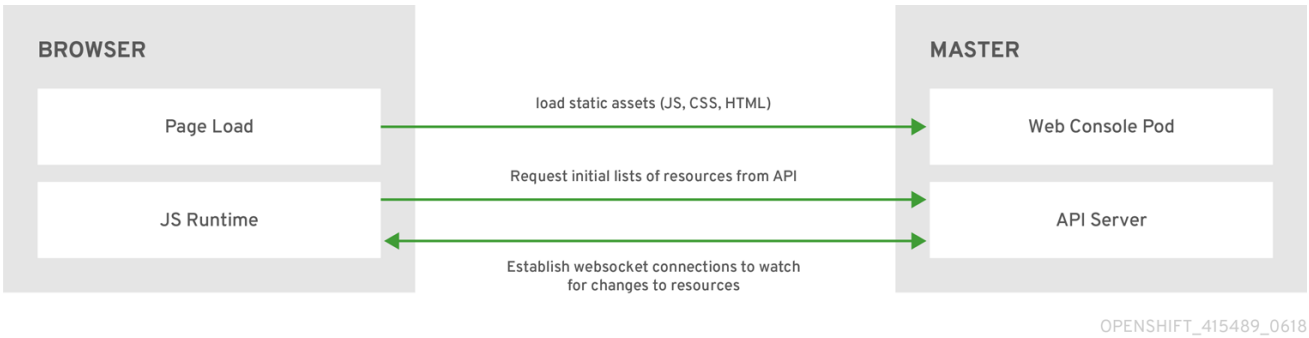
注意

必须启用JavaScript才能使用Web控制台。为获得最佳体验，请使用支持WebSockets的Web浏览器。

Web 控制台作为 pod 在 master 上运行。这个 pod 提供了运行Web控制台所需的静态环境。管理员也可以使用扩展 自定义 Web 控制台，您可以在 web控制台加载时运行脚本和加载自定义样式表。

当您从浏览器访问 Web 控制台时，它会首先加载所有所需的静态资产。然后，它使用 **openshift start** 选项 **--public-master** 或在 **openshift-console-config** 配置映射中定义的 **webconsole-config** 配置映射中的相关参数 **masterPublicURL** 向 OpenShift Container Platform API 发出请求。Web 控制台使用 WebSockets 来保持与 API 服务器的持久连接，并在可用后立即收到更新的信息。

图 2.3. Web 控制台请求架构



为 web 控制台配置的主机名和 IP 地址列入白名单，以便安全地访问 API 服务器，即使浏览器将请求视为跨原始状态。要使用其他主机名从 web 应用程序访问 API 服务器，您必须通过将 **--cors-allowed-origins** 选项指定 **openshift start** 或从相关的 master 配置文件参数 **corsAllowedOrigins** 指定主机名来列入白名单。

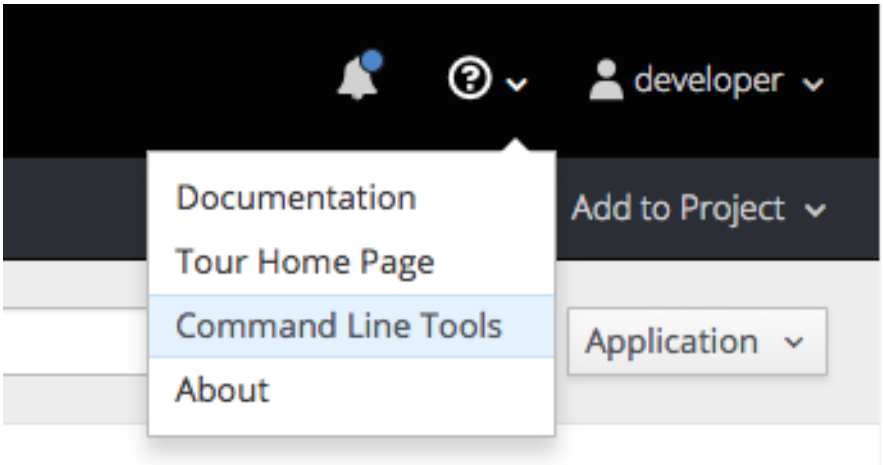
corsAllowedOrigins 参数由配置字段控制。不会对值进行固定或转义。以下是如何获得主机名和转义点的示例：

```
corsAllowedOrigins:
- (?i)//my\.subdomain\.domain\.com(:|\z)
```

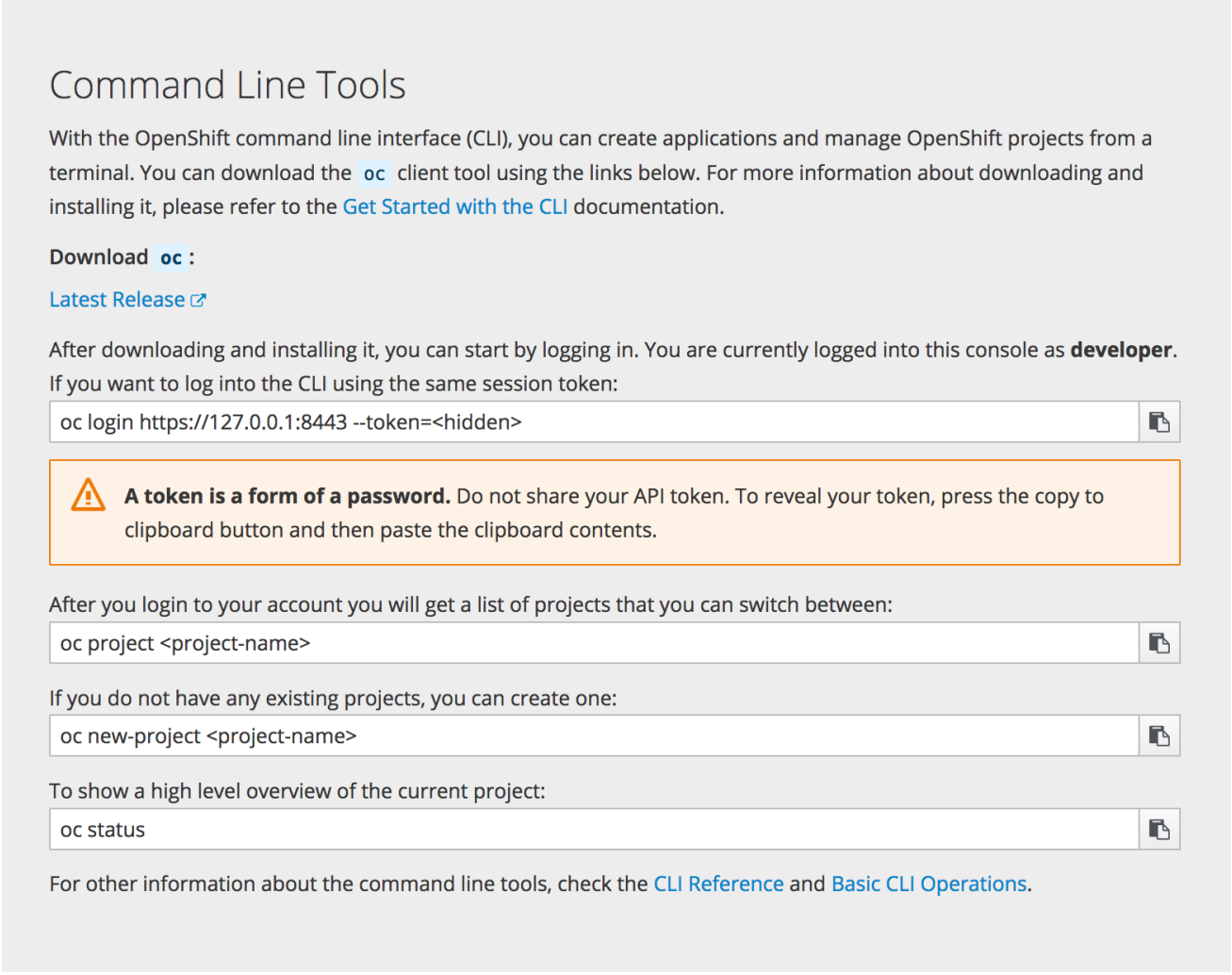
- **(?i)** 使它不区分大小写。
- // 固定到域的开头（并匹配 **http:** 或 **https:** 后面的双斜杠）。
- **\.** 对域名中的点进行转义。
- **(:|\z)** 匹配域名末尾 **(\z)** 或端口分隔符 **(:)**。

2.3.2. CLI 下载

您可从 web 控制台中的 Help 图标访问 CLI 下载：



集群管理员可以 进一步自定义这些链接。



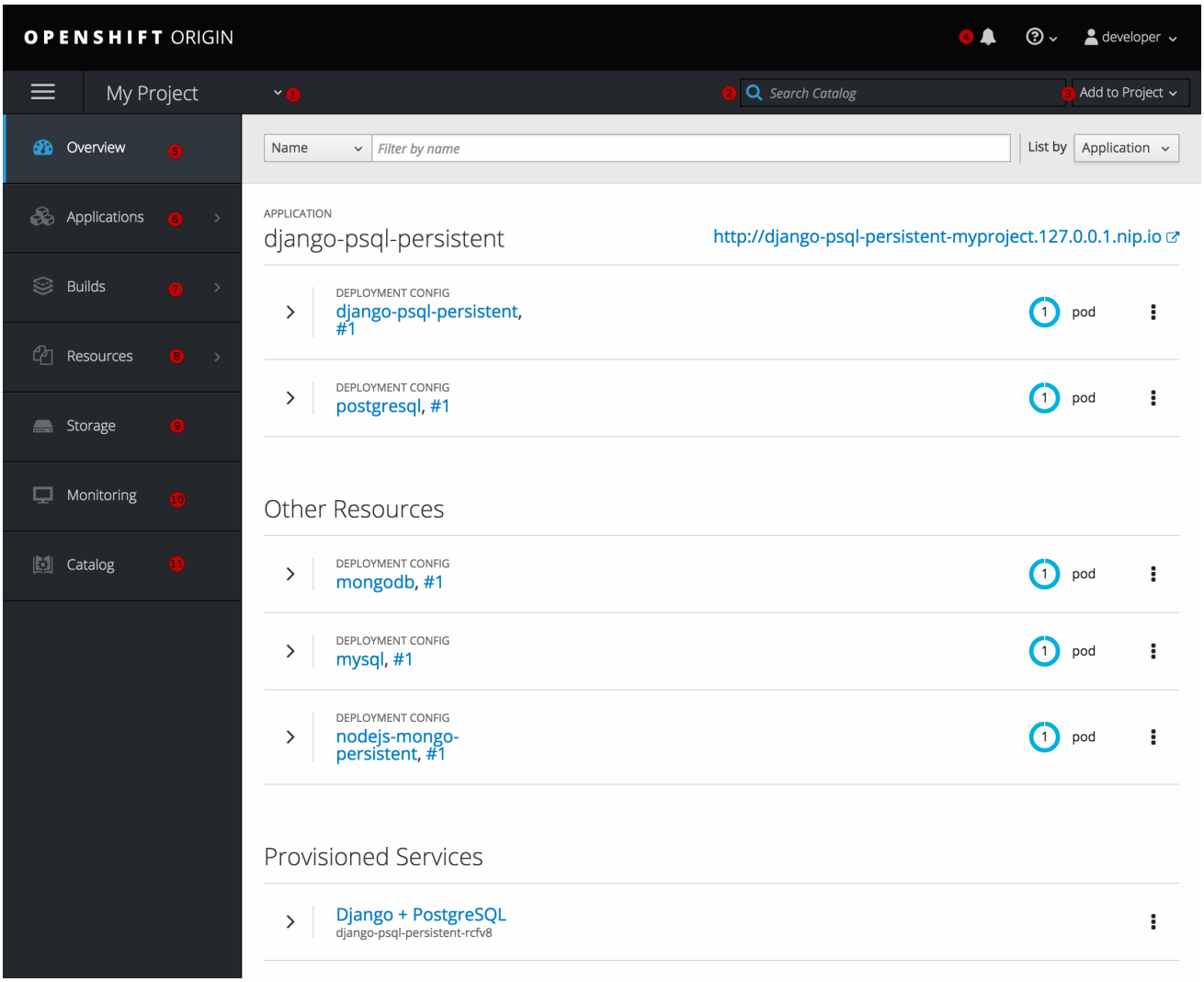
2.3.3. 浏览器要求

查看 OpenShift Container Platform 的已测试的集成。

2.3.4. 项目概述

登录后，Web 控制台为开发人员提供当前所选项目的概览 ：

图 2.4. Web 控制台项目概述



通过项目选择器，您可以 切换有权访问的项目。

要从项目视图中快速查找服务，请输入您的搜索条件

使用源存储库或来自服务目录中的服务创建新应用程序。

与项目相关的通知。

Overview 选项卡（当前已选中）会通过各个组件的高级视图可视化项目的内容。

应用程序 标签页：浏览并针对您的部署、Pod、服务和路由执行操作。

builds 选项卡：浏览并针对您的构建和镜像流执行操作。

resources 选项卡：查看您当前的配额消耗和其他资源。

Storage 选项卡：查看应用程序的持久性卷声明和请求存储。

监控 标签页：查看构建、Pod和部署的日志，以及项目中的所有对象的事件通知。

Catalog 选项卡：从项目内部快速获取目录。

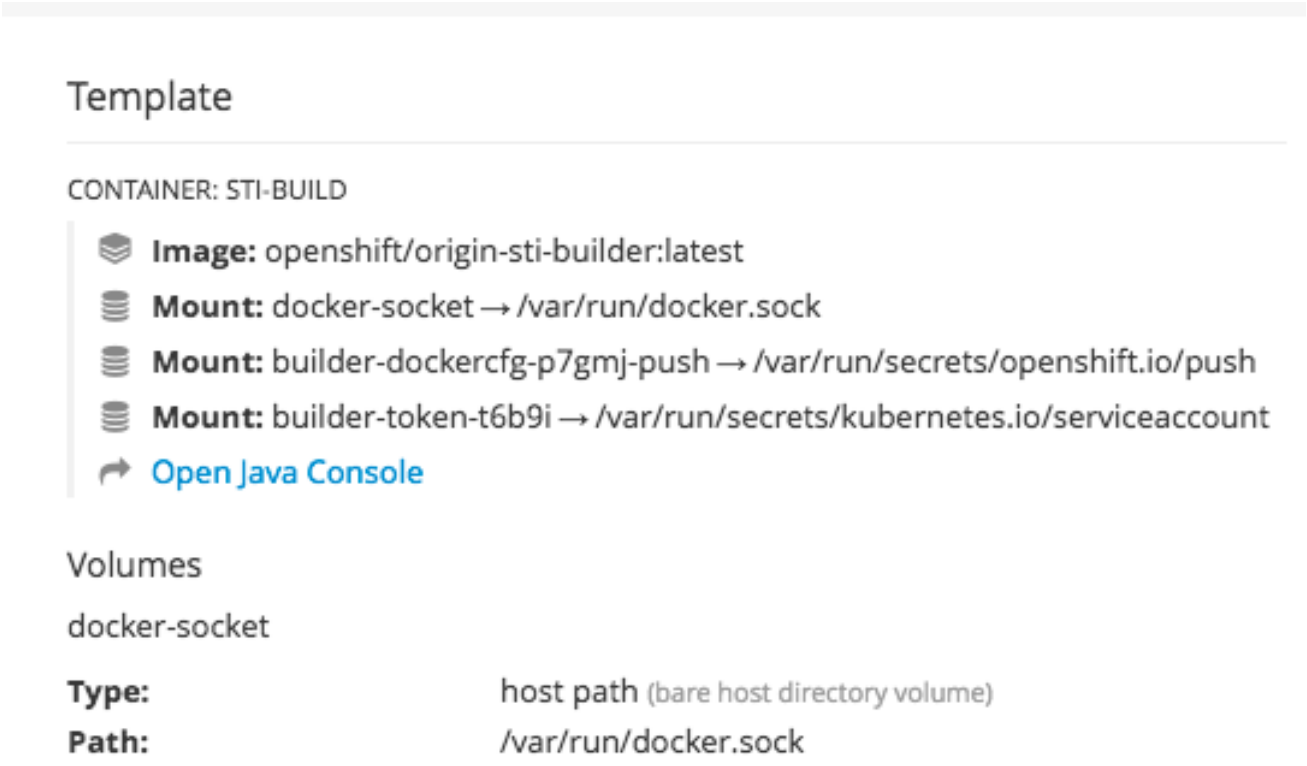
注意

Cockpit 会被自动安装并启用，可帮助您监控开发环境。Red Hat Enterprise Linux Atomic Host:Cockpit 入门 提供了更多有关使用 Cockpit 的信息。

2.3.5. JVM 控制台

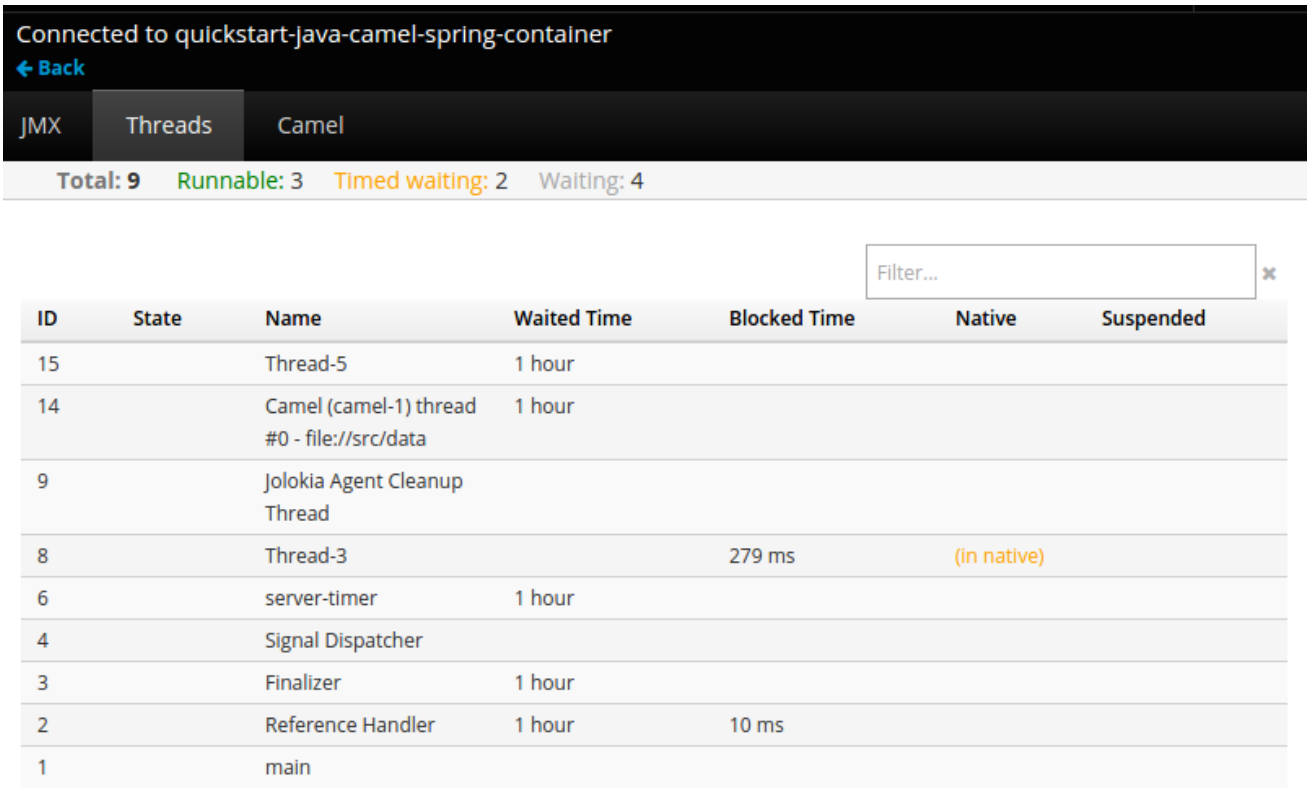
对于基于 Java 镜像的 pod，Web 控制台还会提供对基于 hawt.io的 JVM 控制台的访问，用于查看和管理所有相关集成组件。在 *Browse* → *Pods* 页面中的 pod 详情中会显示 **Connect** 链接，提供带有名为 **jolokia** 的端口的容器。

图 2.5. 带有到 JVM 控制台的链接的 Pod



连接 JVM 控制台后，会显示不同的页面，具体取决于哪些组件与连接的 pod 相关。

图 2.6. JVM 控制台



ID	State	Name	Waited Time	Blocked Time	Native	Suspended
15		Thread-5	1 hour			
14		Camel (camel-1) thread #0 - file://src/data	1 hour			
9		Jolokia Agent Cleanup Thread				
8		Thread-3		279 ms	(in native)	
6		server-timer	1 hour			
4		Signal Dispatcher				
3		Finalizer	1 hour			
2		Reference Handler	1 hour	10 ms		
1		main				

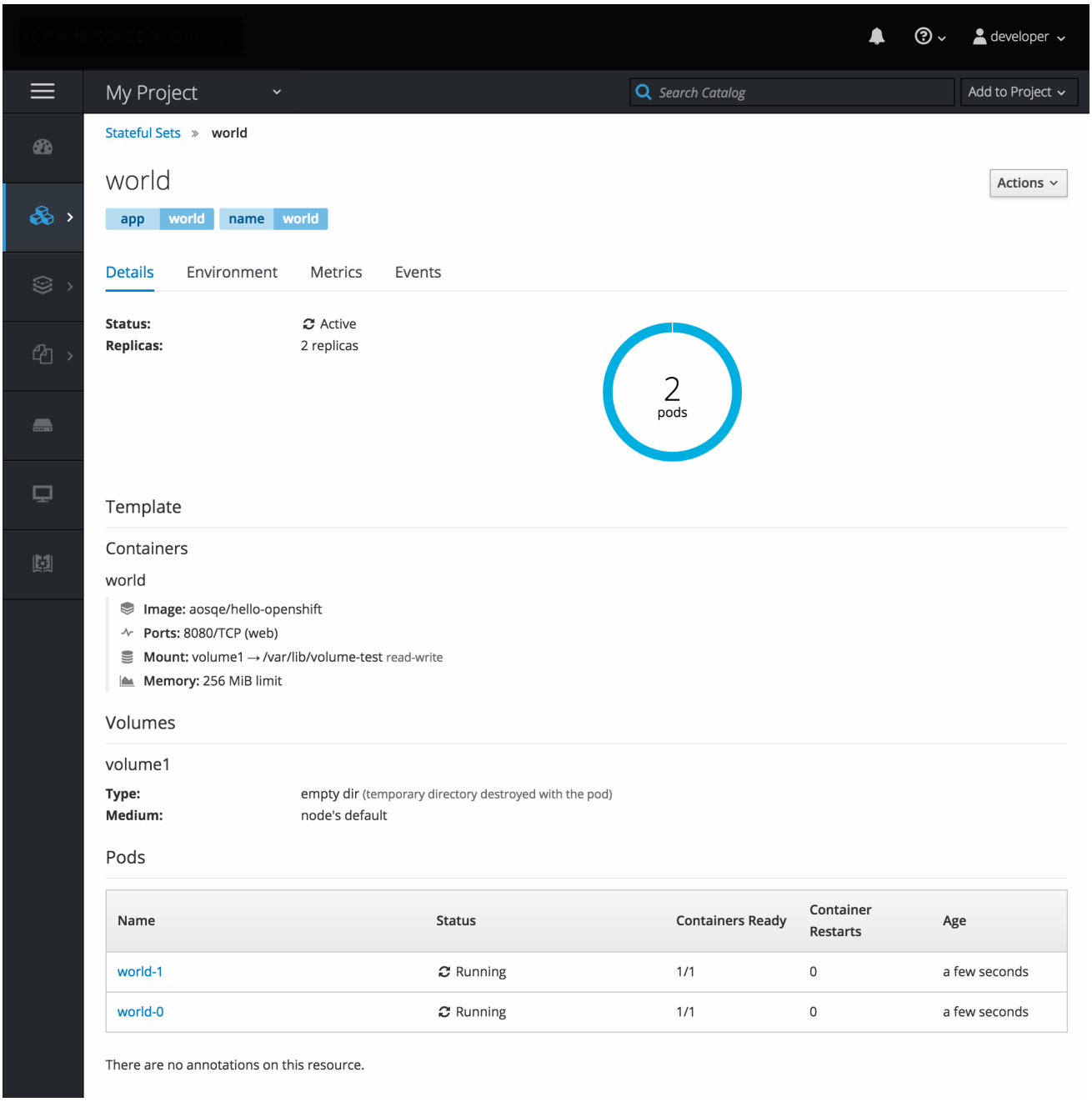
可用的页面如下：

页面	描述
JMX	查看和管理 JMX 域及 mbeans.
线程	查看并监控线程状态。
ActiveMQ	查看和管理 Apache ActiveMQ 代理。
Camel	查看并管理 Apache Camel 路由和依赖项。
OSGi	查看并管理 JBoss Fuse OSGi 环境。

2.3.6. StatefulSets

StatefulSet 控制器为其 pod 提供唯一的身份，并确定部署和扩展顺序。**StatefulSet** 对于唯一标识符、持久性存储、安全部署和扩展以及安全删除和终止很有用。

图 2.7. OpenShift Container Platform 中的 StatefulSet



第 3 章 核心概念

3.1. 概述

以下主题提供有关在使用 OpenShift Container Platform 时将遇到的核心概念和对象的高级架构信息。其中许多对象来自 Kubernetes，由 OpenShift Container Platform 扩展，以提供功能丰富的开发生命周期平台。

- 容器和镜像 是部署应用的构建块。
- Pod 和服务 允许容器相互通信。
- 项目和用户 提供了空间，供社区组织和管理其内容。
- 构建和镜像流 允许您构建工作镜像并对新镜像做出反应。
- 部署 增加了对软件开发和部署生命周期的支持。
- 路由向世界宣布您的服务。
- 模板 允许基于自定义参数一次性创建许多对象。

3.2. 容器和镜像

3.2.1. 容器

OpenShift Container Platform 应用程序的基本单元称为 *容器*。Linux 容器技术是一种轻量型机制，用于隔离运行中的进程，使它们只能跟指定的资源交互。

许多应用程序实例可以在单一主机上的容器中运行，而且相互之间看不到对方的进程、文件和网络等。通常，每个容器都提供单一服务（通常称为"微服务"），如 Web 服务器或数据库，但容器可用于任意工作负载。

多年来，Linux 内核一直在整合容器技术的能力。当前，Docker 项目为主机上的 Linux 容器开发了便捷的管理接口。OpenShift Container Platform 和 Kubernetes 添加了在多主机安装间编配 Docker 格式的容器的功能。

虽然在使用 OpenShift Container Platform 时不直接与 Docker CLI 或服务交互，但了解它们的功能和术语对于了解它们在 OpenShift Container Platform 中的角色以及您的应用程序在容器内如何工作非常重要。**docker** RPM 作为 RHEL 7 的一部分 7 以及 CentOS 和 Fedora 提供，因此您可以与 OpenShift Container Platform 分开进行试验。有关指南，请参阅 Red Hat 系统上的 Docker 格式化容器镜像 入门部分。

3.2.1.1. Init 容器

pod 除了应用程序容器外，pod 还可以包含 init 容器。借助初始容器，您可以重新整理设置脚本和绑定代码。init 容器与常规容器不同，容器始终运行完。每个 init 容器必须在启动下一个容器前成功完成。

如需更多信息，请参阅 Pod 和服务。

3.2.2. 镜像

OpenShift Container Platform 中的容器基于 Docker *格式的容器镜像*。镜像是一种二进制文件，包含运行单一容器的所有要求以及描述其需求和功能的元数据。

您可以将其视为一种打包技术。容器只能访问其镜像中定义的资源，除非创建时授予容器其他访问权限。通过将同一镜像部署到跨越多个主机的多个容器内，并在它们之间进行负载平衡，OpenShift 容器平台可以为镜像中打包的服务提供冗余和横向扩展。

您可以直接使用 Docker CLI 构建镜像，但 OpenShift Container Platform 还提供构建器镜像，通过向现有镜像添加代码或配置来协助创建新的镜像。

由于应用程序会随时间推移开发，因此单个镜像名称实际上可以指代许多不同版本的"same"镜像。每个不同的镜像都由它的唯一哈希值引用（一个较长的十六进制数，如 **fd44297e2ddb050ec4f...**），通常被缩短为 12 个字符（如 **fd44297e2ddb**）。

镜像版本标签策略
除了版本号外，Docker 服务还允许应用标签（如 **v1**、 **v2.1**、**GA** 或默认 **最新**）来进一步指定所需的镜像，因此您可以看到同一镜像被称为 **centos**（表示 **latest** 标签）、**centos:centos7** 或 **fd44297e2ddb**。

警告

对于任何官方 OpenShift Container Platform 镜像，不要使用 **latest** 标签。这些是以 **openshift3/** 开头的镜像。**latest** 可以引用多个版本，如 **3.10** 或 **3.11**。

如何标记镜像指示更新策略。具体来说，镜像更新频率较低。使用以下项来确定您选择的 OpenShift Container Platform 镜像策略：

vX.Y

vX.Y 标签指向 X.Y.Z-<number>。例如，如果 **registry-console** 镜像更新至 v3.11，它指向最新的 3.11.Z-<number> 标签，如 3.11.1-8。

X.Y.Z

与上面的 vX.Y 示例类似，X.Y.Z 标签指向最新的 X.Y.Z-<number>。例如： 3.11.1 将指向 3.11.1-8

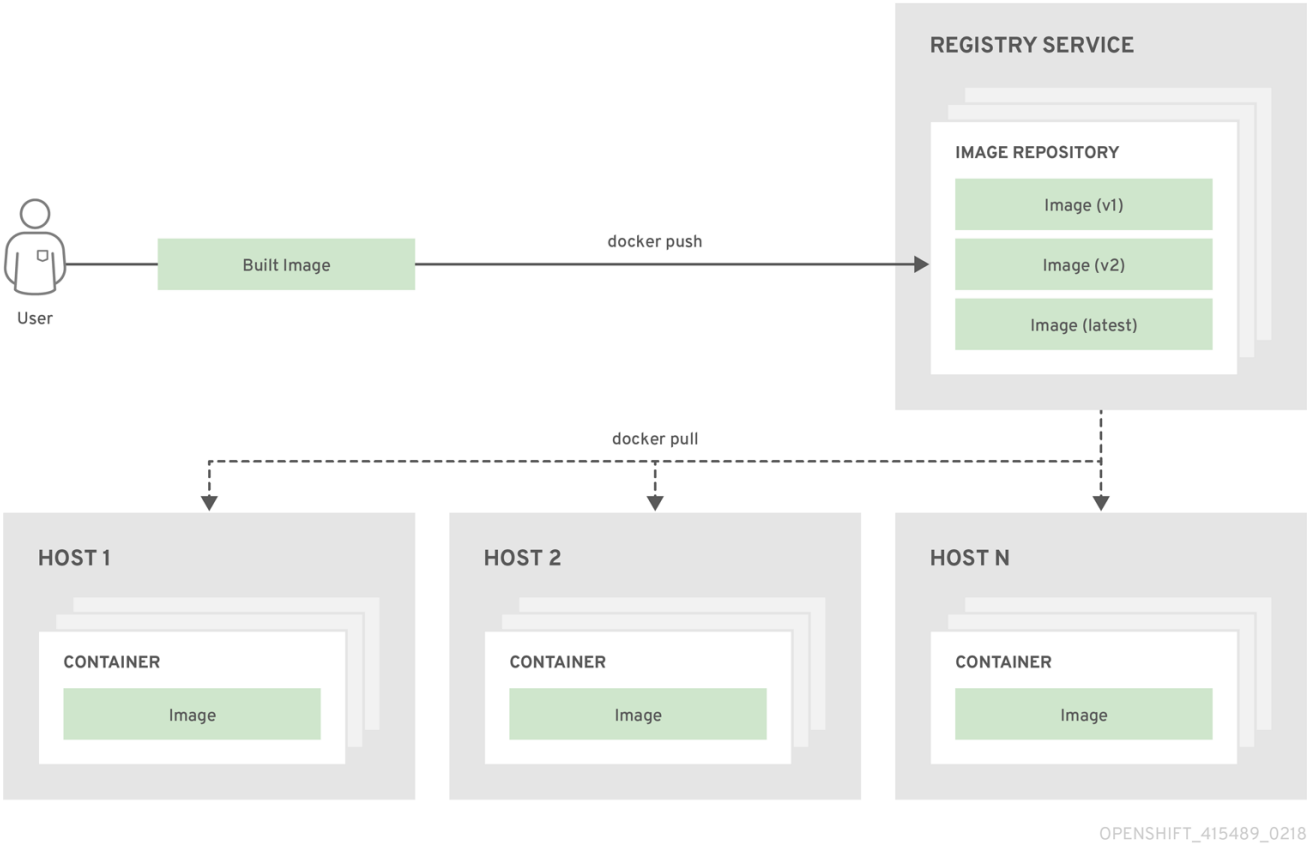
X.Y.Z-<number>

标签是唯一的，不更改。使用此标签时，如果镜像被更新，则镜像不会更新。例如，3.11.1-8 始终会指向 3.11.1-8，即使镜像被更新也是如此。

3.2.3. 容器镜像 registry

容器镜像注册表是用于存储和检索 Docker 格式的容器镜像的服务。registry 包含一个或多个镜像存储库的集合。每个镜像存储库包含一个或多个标记的镜像。Docker 提供自己的注册表，Docker Hub，您也可以使用私有或第三方注册表。红帽在 **registry.redhat.io** 上为订阅者提供了一个 registry。OpenShift Container Platform 还可提供其内部 registry，用于管理自定义容器镜像。

下图中描述了容器、镜像和 registry 之间的关系：



3.3. POD 和服务

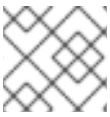
3.3.1. Pods

OpenShift Container Platform 利用 Kubernetes 的 *pod* 概念，它是共同部署在同一主机上的一个或多个容器，也是可被定义、部署和管理的最小计算单元。

对容器而言，Pod 大致相当于机器实例（物理或虚拟）。每个 pod 分配有自己的内部 IP 地址，因此拥有完整的端口空间，并且 pod 内的容器可以共享其本地存储和网络。

Pod 有生命周期，它们经过定义后，被分配到某一节点上运行，然后持续运行，直到容器退出或它们因为其他原因被删除为止。根据策略和退出代码，Pod 可在退出后删除，或被保留下来以启用对容器日志的访问。

OpenShift Container Platform 将 pod 基本上视为不可变；在运行期间无法更改 pod 定义。OpenShift Container Platform 通过终止现有的 pod，再利用修改后的配置和/或基础镜像重新创建 pod，从而实现更改。Pod 也被视为是可抛弃的，不会在重新创建时保持原来的状态。因此，pod 通常应由更高级别的控制器 管理，而不是直接由用户管理。



注意

有关每个 OpenShift Container Platform 节点主机的最大 pod 数量，请参阅集群 最大。

警告

不受复制控制器管理的裸机 pod 不会在节点中断时重新调度。

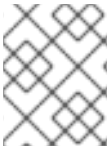
以下是一个提供长时间运行的服务的 pod 示例定义，该服务实际上是 OpenShift Container Platform 基础架构的一部分，即集成的容器镜像 registry。它展示了 pod 的许多特性，其中大多数已在其他主题中阐述，因此这里仅简略提及：

Pod 对象定义 (YAML)

```
apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }
  labels:                                   (1)
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-          (2)
spec:
  containers:                               (3)
  - env:                                    (4)
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
      value: ...
    - name: OPENSHIFT_MASTER
      value: https://master.example.com:8443
    image: openshift/origin-docker-registry:v0.6.2  (5)
    imagePullPolicy: IfNotPresent
    name: registry
    ports:                                  (6)
    - containerPort: 5000
      protocol: TCP
    resources: {}
    securityContext: { ... }                (7)
    volumeMounts:                           (8)
    - mountPath: /registry
      name: registry-storage
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-br6yz
      readOnly: true
  dnsPolicy: ClusterFirst
  imagePullSecrets:
  - name: default-dockercfg-at06w
  restartPolicy: Always                     (9)
```

```
serviceAccount: default                    10
volumes:                                   11
- emptyDir: {}
  name: registry-storage
- name: default-token-br6yz
  secret:
    secretName: default-token-br6yz
```

1. pod 可以被"标上(tag)"一个或多个标签（label），然后使用这些标签在一个操作中选择和管理多组 pod。标签以键/值格式保存在 **metadata** 散列中。本例中的一个标签是 **docker-registry=default**。
2. Pod 在其命名空间内需要具有唯一名称。一个 pod 定义可以使用 **generateName** 属性指定名称的基础，并且会自动添加随机字符来生成唯一名称。
3. **containers** 指定一组容器定义；本例中只有一个，与大多时候相同。
4. 可以指定环境变量以将必要的值传递给每个容器。
5. pod 中的每个容器使用自己的Docker 格式的容器镜像进行安装。
6. 容器可绑定至在 pod 的 IP 上提供的端口。
7. OpenShift Container Platform 为容器定义了一个安全上下文，指定是否允许其作为特权容器来运行，或者以所选用户身份运行，等等。默认上下文的限制性比较强，但管理员可以根据需要进行修改。
8. 容器指定在容器中挂载外部存储卷的位置。在本例中，一个卷用于存储 registry 的数据，另一个卷则提供凭证的访问途径，registry 需要这些凭证来向 OpenShift Container Platform API 发出请求。
9. pod 重启策略，可能的值为 **Always**、**OnFailure** 和 **Never**。默认值为 **Always**。
10. Pod 对 OpenShift Container Platform API 发出请求是一种比较常见的模式，它有一个 **serviceAccount** 字段，用于指定 pod 在发出请求时使用哪个服务帐户用户进行身份验证。这可以为自定义基础架构组件提供精细的访问控制。
11. pod 定义了可供其容器使用的存储卷。在本例中，它提供了一个用于存储 registry 的临时卷，以及一个包含服务帐户凭证的 **secret** 卷。



注意

此 pod 定义不包括 OpenShift Container Platform 在 pod 创建并开始其生命周期后自动填充的属性。Kubernetes pod 文档详细介绍了 pod 的功能和用途。

3.3.1.1. Pod 重启策略

Pod 重启策略决定了 OpenShift Container Platform 在该 pod 中的容器退出时如何响应。策略应用到该 Pod 中的所有容器。

可能的值有：

- **始终 - 在** pod 重启前，按指数避退延时(10s，20s，40s)持续重启 pod 上成功退出的容器。默认值为 **Always**。
- **OnFailure** - 按规定的延时值（10s，20s，40s）不断尝试重启 pod 中失败的容器，上限为 5 分钟。

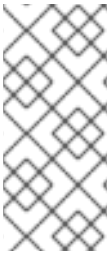
- **Never** - 不尝试重启 pod 中已退出或失败的容器。Pod 立即失败并退出。

绑定到节点后，pod 永远不会绑定到另一个节点。这意味着，需要一个控制器才能使 pod 在节点失败后存活：

状况	控制器类型	重启策略
应该终止的 Pod（例如，批量计算）	作业	**OnFailure** 或 **Never**
不应该终止的 Pod（例如，Web 服务器）	复制控制器	**Always**。
需要在每台机器上运行一个的 Pod	Daemonset	任意

如果 pod 上的容器失败，并且重启策略被设置为 **OnFailure**，则 pod 会保留在该节点上并重新启动容器。如果您不希望容器重启，请使用 **Never** 的重启策略。

如果整个 pod 失败，OpenShift Container Platform 会启动一个新 pod。开发人员需要解决应用程序可能会在新 pod 中重启的问题。特别是，应用程序需要处理由之前运行导致的临时文件、锁定、不完整的输出等。



注意

Kubernetes 架构需要来自云提供商的可靠端点。当云提供商停机时，kubelet 会防止 OpenShift Container Platform 重启。

如果底层云提供商端点不可靠，请不要使用云提供商集成来安装集群。应像在非云环境中一样安装集群。不建议在已安装的集群中打开或关闭云提供商集成。

如需详细了解 OpenShift Container Platform 如何使用与失败容器相关的重启策略，请参阅 Kubernetes 文档中的示例状态。

3.3.2. Init 容器

init 容器 是在 pod 应用程序容器启动前启动的 pod 中的容器。Init 容器可以共享卷，执行网络操作，并在剩余的容器启动前执行计算。Init 容器也可以阻止或延迟应用程序容器的启动，直到满足一些前提条件为止。

当 pod 启动时，在网络和卷初始化后，初始容器会按顺序启动。每个 init 容器都必须成功退出，然后调用下一个容器。如果 init 容器启动（因为运行时无法启动）或退出失败，则会根据 Pod 重启策略 来重试它。

直到所有 init 容器都成功前，pod 无法就绪。

有关一些 init 容器使用示例，请参阅 Kubernetes 文档。

以下示例概述了一个包含两个 init 容器的简单 pod。第一个 init 容器会等待 **myservice**，第二个则等待 **mydb**。两个容器都成功后，Pod 就会启动。

Init 容器 Pod 对象定义(YAML)示例

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice 1
    image: busybox
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb 2
    image: busybox
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

1 指定 **myservice** 容器。

2 指定 **mydb** 容器。

每个 init 容器都具有 app 容器的所有字段， 但 **readinessProbe** 除外。init 容器必须退出才能继续，且无法定义除完成之外的就绪度。

init 容器可以在 pod 上包含 **activeDeadlineSeconds** 和 **livenessProbe**，以防止 init 容器永远失败。活动截止时间包括 init 容器。

3.3.3. 服务

Kubernetes 服务 充当内部负载均衡器。它标识一组复制的 pod，以代理其接收的连接。可以在服务中任意添加或删除支持 pod，同时服务始终保持可用，从而使依赖于服务的任何对象能够以一致的地址引用它。默认服务 clusterIP 地址来自 OpenShift Container Platform 内部网络，它们用于允许 pod 相互访问。

要允许从外部访问该服务，可以将集群外部的 **externalIP** 和 **ingressIP** 地址分配给该服务。这些 **externalIP** 地址也可以是虚拟 IP 地址，提供对该服务 的高可用性 访问权限。

服务被分配一个 IP 地址和端口对，访问时代理到适当的后备 pod。服务使用标签选择器来查找运行的所有容器，这些容器在特定端口上提供特定的网络服务。

与 pod 一样，服务是 REST 对象。以下示例显示了上方定义的 pod 的服务定义：

服务对象定义(YAML)

```
apiVersion: v1
kind: Service
metadata:
  name: docker-registry      1
spec:
  selector:                  2
    docker-registry: default
  clusterIP: 172.30.136.123  3
```

```
ports:
- nodePort: 0
  port: 5000          4
  protocol: TCP
  targetPort: 5000    5
```

1. 服务名称 **docker-registry** 也用于在命名空间中使用插入到其他 pod 中的服务 IP 的环境变量。名称长度最多为 63 个字符。
2. 标签选择器识别所有带有作为后备 pod 附加的 **docker-registry=default** 标签的 pod。
3. 服务的虚拟 IP，从内部 IP 池自动创建。
4. 服务侦听的端口。
5. 服务将连接转发到的后备 pod 上的端口。

Kubernetes 文档包含更多与服务相关的信息。

3.3.3.1. Service externalIPs

除了集群的内部 IP 地址外，用户还可配置集群外部的 IP 地址。管理员负责确保流量通过此 IP 到达节点。

集群管理员必须从 *master-config.yaml* 文件中配置的 **externalIPNetworkCIDRs** 范围内选择 externalIPs。当 *master-config.yaml* 发生更改时，必须重启 master 服务。

externalIPNetworkCIDR /etc/origin/master/master-config.yaml 示例

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.0.1.0.0/24
```

服务 externalIPs 定义(JSON)

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "name": "http",
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376
            }
        ],
        "externalIPs" : [
```

```
        "192.0.1.1"      1
      ]
    }
}
```

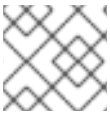
1 用于在其上公开**端口**的外部 IP 地址列表。此列表除了内部 IP 地址列表之外。

3.3.3.2. Service ingressIPs

在非云集群中，可以从地址池中自动分配 externalIP 地址。这可免除管理员手动分配它们的需求。

该池在 */etc/origin/master/master-config.yaml* 文件中进行配置。更改此文件后，重启 master 服务。

ingressIPNetworkCIDR 被默认设置为 **172.29.0.0/16**。如果集群环境还没有使用这个私有范围，请使用默认范围或设置自定义范围。



注意

如果您使用高可用性，则此范围必须小于 256 个地址。

ingressIPNetworkCIDR /etc/origin/master/master-config.yaml 示例

```
networkConfig:
  ingressIPNetworkCIDR: 172.29.0.0/16
```

3.3.3.3. Service NodePort

设置服务 **type=NodePort** 将从错误配置的范围分配一个端口（默认）：30000-32767，每个节点会将端口（每个节点中的相同端口号）代理到您的服务中。

所选端口将在服务配置中报告，位于 **spec.ports[*].nodePort** 下。

要指定一个自定义端口，可将端口号放在 nodePort 字段中。自定义端口号必须在为 nodePorts 配置范围内。当 '**master-config.yaml**' 被改变时，必须重启 master 服务。

Sample servicesNodePortRange /etc/origin/master/master-config.yaml

```
kubernetesMasterConfig:
  servicesNodePortRange: ""
```

该服务将同时作为 < **NodeIP>:spec.ports[].nodePort** 和 **spec.clusterIp:spec.ports[].port**的形式可见

注意

设置 nodePort 是一个特权操作。

3.3.3.4. 服务代理模式

OpenShift Container Platform 有两个不同实现的服务路由基础架构。默认的实现完全基于 **iptables**，并使用 probabilistic **iptables** 重新编写规则以在端点 Pod 之间分发传入的服务连接。较早的实施使用用户空间进程接受进入的连接，然后在客户端和其中一个端点 Pod 之间代理流量。

基于 iptables的实施效率更高，但它要求所有端点始终能够接受连接；用户空间实施的速度较慢，但可以依次尝试多个端点，直到找到可正常工作。如果您有良好的 就绪度检查 （或通常可靠的节点和 pod），则基于 iptables的服务代理是最佳选择。否则，您可以在安装时或通过编辑节点配置文件在部署集群时启用基于用户空间的代理。

3.3.3.5. 无头服务

如果您的应用程序不需要负载均衡或单一服务 IP 地址，您可以创建一个无头服务。当您创建无头服务时，不会执行负载均衡或代理，也不会为该服务分配集群 IP。对于此类服务，DNS 会根据服务是否定义了选择器来自动配置。

使用选择器的服务 ：对于定义选择器的无头服务，端点控制器会在 API 中创建 **Endpoints** 记录，并修改 DNS 配置来返回指向支持该服务的 pod 的 **A** 记录（地址）。

没有选择器的服务 ：对于不定义选择器的无头服务，端点控制器不会创建 **端点** 记录。但是，DNS 系统会查找并配置以下记录：

- 对于 **ExternalName** 类型服务，为 **CNAME** 记录。
- 对于所有其他服务类型，对于与服务共享名称的任何端点的 记录。

3.3.3.5.1. 创建无头服务

创建无头服务与创建标准服务类似，但不会声明 **ClusterIP** 地址。要创建无头服务，请添加 **clusterIP : none** 参数值为服务 YAML 定义。

例如，对于您要成为同一集群或服务一部分的 pod 组。

Pod 列表

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

输出示例

```
NAME              READY  STATUS    RESTARTS   AGE    IP          NODE
frontend-1-287hw  1/1    Running   0          7m     172.17.0.3  node_1
frontend-1-68km5  1/1    Running   0          7m     172.17.0.6  node_1
```

您可以将无头服务定义为：

无头服务定义

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ruby-helloworld-sample
    template: application-template-stibuild
  name: frontend-headless (1)
spec:
  clusterIP: None (2)
  ports:
  - name: web
    port: 5432
```

```
    protocol: TCP
    targetPort: 8080
  selector:
    name: frontend 3
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

1 无头服务的名称。

2 将 **clusterIP** 变量设置为 **None** 将声明一个无头服务。

3 选择具有 **frontend** 标签的所有 pod。

另外，无头服务本身没有任何 IP 地址。

```
$ oc get svc
```

输出示例

```
NAME                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
frontend            ClusterIP   172.30.232.77    <none>        5432/TCP   12m
frontend-headless   ClusterIP   None             <none>        5432/TCP   10m
```

3.3.3.5.2. 使用无头服务进行端点发现

使用无头服务的好处是您可以直接发现 pod 的 IP 地址。标准服务充当负载均衡器或代理，并使用服务名称来提供对工作负载对象的访问。使用无头服务时，服务名称将解析为服务分组的 pod 的 IP 地址集合。

当您查找标准服务的 DNS **A** 记录时，您将获得服务的负载均衡 IP。

```
$ dig frontend.test A +search +short
```

输出示例

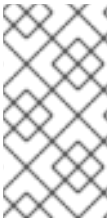
```
172.30.232.77
```

但是，对于无头服务，您可以获取单个 pod 的 IP 列表。

```
$ dig frontend-headless.test A +search +short
```

输出示例

```
172.17.0.3
172.17.0.6
```



注意

对于使用带有 StatefulSet 的无头服务以及相关的用例，您需要在初始化和终止期间解析 pod 的 DNS，将 **publishNotReadyAddresses** 设置为 **true**（默认值为 **false**）。当 **publishNotReadyAddresses** 设为 **true** 时，这表示 DNS 实现必须发布与该服务关联的 Endpoints 子集的 **notReadyAddresses**。

3.3.4. 标签

标签用于组织、组或选择 API 对象。例如，pod 使用标签进行"标记"，服务使用标签选择器标识其代理的 pod。这使得服务能够引用 pod 组，甚至可以将带有不同容器的 pod 视为相关的实体。

大多数对象都可以在其元数据中包含标签。因此标签可用于对特定应用程序的不同对象进行分组，例如，所有 pod、服务、复制控制器 和 部署配置 都可以分组。

标签是简单的键/值对，如下例所示：

```
labels:
  key1: value1
  key2: value2
```

考虑：

- 由 **nginx** 容器组成的 Pod，标签为 **role=webserver**。
- 由 **Apache httpd** 容器组成的 pod，其标签为 **role=webserver**。

定义为使用 **role=webserver** 标签的 pod 的服务或复制控制器会将这两个 pod 视为同一组的一部分。

Kubernetes 文档包含有关标签的更多信息。

3.3.5. Endpoints

后端服务的服务器称为其端点，并由名称与服务同名的 **Endpoints** 类型的对象指定。当服务由 pod 支持时，这些 pod 通常由服务规格中的标签选择器指定，OpenShift Container Platform 会自动创建指向这些 pod 的 Endpoints 对象。

在某些情况下，您可能想要创建服务，但它由外部主机支持，而不是由 OpenShift Container Platform 集群中的 pod 支持。在这种情况下，您可以取消服务中的 **selector** 字段，并 手动创建 Endpoints 对象。

请注意，OpenShift Container Platform 不允许大多数用户手动创建端点对象，指向为 pod 和服务 IP 保留的网络块中的 IP 地址。只有集群管理员或其他有权在 **端点/限制 下创建 资源的用户才能** 创建此类 Endpoint 对象。

3.4. 项目和用户

3.4.1. 用户

与 OpenShift Container Platform 的交互与用户相关联。OpenShift Container Platform 用户对象代表一个执行者，可以通过向 它们或组 添加角色来授予 系统中的权限。

可能存在的用户类型有几种：

常规用户	这是大多数交互式 OpenShift Container Platform 用户表示的方式。常规用户在第一次登录时在系统中自动创建，或者通过 API 创建。常规用户通过 **User** 对象表示。例如，**joe alice**
系统用户	许多系统用户在基础架构定义时自动创建，主要用于使基础架构与 API 安全地交互。这包括集群管理员（有权访问一切资源）、特定于一个节点的用户、供路由器和 registry 使用的用户，以及一些其他用户。最后，有一个**匿名系统用户**，默认供未经身份验证的请求使用。例如： **system:admin system:node:node1.example.com**
服务帐户	这些是与项目关联的特殊系统用户；其中一些在项目首次创建时自动创建，而项目管理员可以创建更多服务帐户来定义对各个项目的内容的访问权限。服务帐户通过 **ServiceAccount** 对象表示。例如：**system:serviceaccount:default:deployer system:serviceaccount:foo:builder**

每一用户必须通过 某种形式的身份验证才能访问 OpenShift Container Platform。无身份验证或身份验证无效的 API 请求会被验证为由 **匿名系统用户** 发出的请求。经过身份验证后，策略决定用户 被授权执行的操作。

3.4.2. 命名空间

Kubernetes 命名空间提供范围集群中资源的机制。在 OpenShift Container Platform 中，项目 是一个带有额外注解的 Kubernetes 命名空间。

命名空间为以下对象提供唯一范围：

- 指定名称的资源，以避免基本命名冲突。
- 委派给可信用户的管理授权。
- 限制社区资源消耗的能力。

系统中的大多数对象都通过命名空间来设定范围，但一些对象不在此列且没有命名空间，如节点和用户。

Kubernetes 文档中提供有关命名空间的更多信息。

3.4.3. 项目

项目是附带额外注解的 Kubernetes 命名空间，是管理常规用户资源访问权限的中央载体。通过项目，一个社区的用户可以在与其他社区隔离的前提下组织和管理其内容。用户必须由管理员授予对项目的访问权限；或者，如果用户有权创建项目，则自动具有自己创建项目的访问权限。

项目可以有单独的 **name**, **displayName**, 和 **description**。

- 必需的 **name** 是项目的唯一标识符，在使用 CLI 工具或 API 时是做常用的。名称长度最多为 63 个字符。
- 可选的 **displayName** 是项目在 web 控制台中的显示形式（默认为 **name**）。
- **可选的描述** 可以是项目更为详细的描述，它也在 Web 控制台中可见。

每个项目限制了自己的一组：

对象（object）	Pod、服务和复制控制器等。
策略（policy）	用户能否对对象执行操作的规则。
约束（constraint）	对各种对象进行限制的配额。
服务帐户	服务帐户自动使用项目中对象的指定访问权限进行操作。

集群管理员可以 创建项目，并将项目 的管理权限委派给 用户社区的任何成员。集群管理员也可以允许开发人员 创建自己的项目。

开发人员和管理员可以使用 CLI 或 Web 控制台 与项目交互。

3.4.3.1. 安装时提供的项目

OpenShift Container Platform 附带大量项目，以 **openshift-** 开头的项目对用户来说最为重要。这些项目托管作为 Pod 运行的主要组件和其他基础架构组件。在这些命名空间中创建的带有 关键 Pod 注解的 pod 被视为关键，它们会保证被 kubelet 准入。在这些命名空间中为主要组件创建的 Pod 已标记为“critical”。

3.5. 构建和镜像流

3.5.1. 构建（build）

构建 (build) 是将输入参数转换为结果对象的过程。此过程最常用于将输入参数或源代码转换为可运行的镜像。**BuildConfig** 对象是整个构建过程的定义。

OpenShift Container Platform 通过从构建镜像创建 Docker 格式的容器，并将它们推送到 容器镜像 registry 中。

构建对象共享共同特征：构建的输入、完成构建过程需要、记录构建过程、从成功构建中发布资源并发布构建的最终状态。构建会使用资源限制，具体是指定资源限值，如 CPU 使用量、内存使用量，以及构建或 Pod 执行时间。

OpenShift Container Platform 构建系统提供对 *构建策略* 的可扩展支持，它们基于构建 API 中指定的可选择类型。可用的构建策略主要有三种：

- Docker 构建
- Source-to-Image (S2I) 构建
- Custom 构建

默认情况下，支持 Docker 构建和 S2I 构建。

构建生成的对象取决于用于创建它的构建器（builder）。对于 Docker 和 S2I 构建，生成的对象为可运行的镜像。对于 Custom 构建，生成的对象是构建器镜像作者指定的任何事物。

另外，也可利用 Pipeline 构建策略来实现复杂的工作流：

- 持续集成
- 持续部署

有关构建命令列表，请参阅开发人员指南。

如需有关 OpenShift Container Platform 如何使用 Docker 进行构建的更多信息，请参阅上游文档。

3.5.1.1. Docker 构建

Docker 构建策略调用 docker build 命令，因此需要一个含有 ***Dockerfile*** 的存储库并且其中包含所有必要的工件，从而能生成可运行的镜像。

3.5.1.2. Source-to-Image(S2I)构建

Source-to-Image(S2I) 是一种用于构建可重复生成的 Docker 格式容器镜像的工具。它通过将应用程序源代码注入容器镜像并汇编新镜像来生成可随时运行的镜像。新镜像融合了基础镜像（构建器）和构建的源代码，并可搭配 **docker run** 命令使用。S2I 支持递增构建，可重复利用以前下载的依赖项和过去构建的工件等。

S2I 的优点包括：

镜像灵活性	可以编写 S2I 脚本，将应用程序代码注入到几乎所有现有的 Docker 格式容器镜像，以此利用现有的生态系统。请注意，S2I 目前依靠 **tar** 来注入应用程序源代码，因此镜像需要能够处理 tar 压缩的内容。
速度	使用 S2I 时，汇编过程可以执行大量复杂操作，无需在每一步创建新层，进而能实现快速的流程。此外，可以编写 S2I 脚本来重复利用应用程序镜像的旧版本，而不必在每次运行构建时下载或构建它们。
可修补性	如果基础镜像因为安全问题而需要补丁，则 S2I 允许基于新的基础镜像重新构建应用程序。
操作效率	通过限制构建操作而不许随意进行 ***Dockerfile*** 允许的操作，PaaS 运维人员可以避免意外或故意滥用构建系统。
操作安全性	构建任意 ***Dockerfile*** 会将主机系统暴露于 root 特权提升。因为整个 Docker 构建过程都通过具备 Docker 特权的用户运行，这可能被恶意用户利用。S2I 限制以 root 用户执行操作，而能够以非 root 用户运行脚本。
用户效率	S2I 禁止开发人员在应用程序构建期间执行任意 **yum install** 类型的操作。因为这类操作可能会减慢开发迭代速度。
生态系统	S2I 倡导共享镜像生态系统，您可以将其中的最佳实践运用于自己的应用程序。
可重复生成性	生成的镜像可以包含所有输入，包括构建工具和依赖项的特定版本。这可确保精确地重新生成镜像。

3.5.1.3. 自定义构建

采用 Custom 构建策略时，开发人员可以定义负责整个构建过程的特定构建器镜像。通过利用自己的构建器镜像，可以自定义构建流程。

自定义构建器（Custom builder）镜像是嵌入了构建过程逻辑的普通 Docker 格式容器镜像，例如用于构建 RPM 或基础镜像。**openshift/origin-custom-docker-builder** 镜像在 Docker Hub registry 上作为自定义构建器镜像的示例实施提供。

3.5.1.4. Pipeline 构建

采用 Pipeline 构建策略时，开发人员可以定义 *Jenkins Pipeline* 由 Jenkins Pipeline 插件执行。构建可以由 OpenShift Container Platform 启动、监控和管理，其方式与任何其他构建类型相同。

Pipeline 工作流在 Jenkinsfile 中定义，或直接嵌入在构建配置中，或者在 Git 存储库中提供并由构建配置引用。

项目第一次使用 Pipeline 策略定义构建配置时，OpenShift Container Platform 会实例化 Jenkins 服务器来执行管道。项目中的后续管道构建配置将共享这个 Jenkins 服务器。

有关如何部署 Jenkins 服务器以及如何配置或禁用自动置备行为的详情，请参阅配置管道执行。

注意

即使所有 Pipeline 构建配置都被删除，Jenkins 服务器也不会自动删除。用户必须手动删除它。

有关 Jenkins Pipelines 的更多信息，请参阅 Jenkins 文档。

3.5.2. 镜像流

镜像流及其关联标签提供了一个用于从 OpenShift Container Platform 中引用容器镜像的抽象 集。镜像流及其标签用于查看可用镜像，确保您使用所需的特定镜像，即使存储库中的镜像发生变化也是如此。

镜像流不含实际镜像数据，它提供了相关镜像的一个单独的虚拟视图，类似于镜像存储库。

您可以在添加新镜像时，配置构建和部署来监控镜像流的通知，并通过分别执行构建或部署来做出反应。

例如，如果部署正在使用某个镜像并且创建了该镜像的新版本，则会自动执行部署以获取镜像的新版本。

但是，如果没有更新 Deployment 或 Build 使用的镜像流标签，则即使更新了容器镜像 registry 中的容器镜像，构建或部署仍会继续使用之前的（已知良好）镜像。

源镜像可存储在以下任一位置：

- OpenShift Container Platform 集成的 registry
- 一个外部 registry，如 **registry.redhat.io** 或 **hub.docker.com**
- OpenShift Container Platform 集群中的其他镜像流

当您定义引用镜像流标签（如构建或部署配置）的对象时，您指向镜像流标签，而不是 Docker 存储库。当您构建或部署应用程序时，OpenShift Container Platform 会使用镜像流标签查询 Docker 存储库，以查找与镜像关联的 ID，并使用正确的镜像。

镜像流元数据会与其他集群信息一起存储在 etcd 实例中。

以下镜像流包含两个标签：**34** 个指向 Python v3.4 镜像和 **35** （指向 Python v3.5 镜像）的 34:

```
$ oc describe is python
```

输出示例

```
Name:   python
```

```
Namespace:  imagestream
Created:  25 hours ago
Labels:   app=python
Annotations:  openshift.io/generated-by=OpenShiftWebConsole
  openshift.io/image.dockerRepositoryCheck=2017-10-03T19:48:00Z
Docker Pull Spec: docker-registry.default.svc:5000/imagestream/python
Image Lookup:  local=false
Unique Images:  2
Tags:   2

34
  tagged from centos/python-34-centos7

  * centos/python-34-
centos7@sha256:28178e2352d31f240de1af1370be855db33ae9782de737bb005247d8791a54d0
      14 seconds ago

35
  tagged from centos/python-35-centos7

  * centos/python-35-
centos7@sha256:2efb79ca3ac9c9145a63675fb0c09220ab3b8d4005d35e0644417ee552548b10
      7 seconds ago
```

使用镜像流有以下几大优势：

- 您可以添加标签、回滚标签和快速处理镜像，而无需使用命令行重新执行 push 操作。
- 当一个新镜像被推送（push）到 registry 时，可触发构建和部署。另外，OpenShift Container Platform 还针对其他资源（如 Kubernetes 对象）具有通用触发器。
- 您可以为定期重新导入标记标签。如果源镜像已更改，则这个更改会被发现并反映在镜像流中，这会触发构建和/或部署流程，具体取决于构建或部署配置。
- 您可使用细粒度访问控制来共享镜像，快速向整个团队分发镜像。
- 如果源更改，镜像流标签仍将指向已知良好的镜像版本，以确保您的应用程序不会意外中断。
- 您可以通过 镜像流对象 的权限配置安全性，以了解谁可以查看和使用镜像。
- 在集群级别上缺少读取或列出镜像权限的用户仍可使用镜像流来检索项目中标记的镜像。

有关策展的镜像流集合，请参阅 OpenShift 镜像流和模板库。

使用镜像流时，务必要了解镜像流标签指向什么以及如何对标签和镜像的更改产生影响。例如：

- 如果您的镜像流标签指向容器镜像标签，则需要了解如何更新容器镜像标签。例如，容器镜像标签 **docker.io/ruby:2.5** 指向 v2.5 ruby 镜像，但容器镜像标签 **docker.io/ruby:latest** 更改有主要版本。因此，镜像流标签所指向的容器镜像标签可以告诉您镜像流标签的稳定程度。
- 如果您的镜像流标签跟随另一个镜像流标签，而不是直接指向容器镜像标签，那么镜像流标签可能会更新，以便在以后遵循不同的镜像流标签。这个更改可能会导致获取不兼容的版本更改。

3.5.2.1. 重要术语

Docker 软件仓库

相关容器镜像和标识它们的标签的集合。例如，OpenShift Jenkins 镜像位于 Docker 存储库中：

```
docker.io/openshift/jenkins-2-centos7
```

容器 registry

可以从 Docker 存储库存储和服务镜像的内容服务器。例如：

```
registry.redhat.io
```

容器镜像

可以作为容器运行的特定一组内容。通常与 Docker 存储库中的特定标签关联。

容器镜像标签

应用到存储库中容器镜像的标签，用于区分特定镜像。例如，这里 3.6.0 是一个标签：

```
docker.io/openshift/jenkins-2-centos7:3.6.0
```

注意

可以更新容器镜像标签，以便可以随时指向新的容器镜像内容。

容器镜像 ID

可用于拉取镜像的 SHA（安全哈希算法）代码。例如：

```
docker.io/openshift/jenkins-2-centos7@sha256:ab312bda324
```

注意

SHA 镜像 ID 不能更改。特定 SHA 标识符会始终引用完全相同的容器镜像内容。

镜像流

一个 OpenShift Container Platform 对象，其中包含指向由标签识别的任意数量的 Docker 格式容器镜像的指针。您可以将镜像流视为等同于 Docker 存储库。

镜像流标签

指向镜像流中镜像的命名指针。镜像流标签与容器镜像标签类似。请参阅以下镜像流标签。

镜像流镜像

允许您从标记了特定容器镜像的特定镜像流中检索该镜像。镜像流镜像是一个 API 资源对象，用于收集一些有关特定镜像 SHA 标识符的元数据。请参阅以下镜像流镜像。

镜像流触发器

当镜像流标签更改时引发特定操作的触发器。例如，导入可导致标签值变化。当有部署、构建或其他资源监听这些信息时，就会启动触发器。请参阅以下镜像流触发器。

3.5.2.2. 配置镜像流

镜像流对象文件包含以下元素。

注意

有关管理镜像和镜像流的详细信息，请参阅 开发人员指南。

镜像流对象定义

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/generated-by: OpenShiftNewApp
  creationTimestamp: 2017-09-29T13:33:49Z
  generation: 1
  labels:
    app: ruby-sample-build
    template: application-template-stibuild
  name: origin-ruby-sample (1)
  namespace: test
  resourceVersion: "633"
  selflink: /oapi/v1/namespaces/test/imagestreams/origin-ruby-sample
  uid: ee2b9405-c68c-11e5-8a99-525400f25e34
spec: {}
status:
  dockerImageRepository: 172.30.56.218:5000/test/origin-ruby-sample (2)
  tags:
  - items:
    - created: 2017-09-02T10:15:09Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d (3)
      generation: 2
      image: sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5 (4)
    - created: 2017-09-29T13:40:11Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
      generation: 1
      image: sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest (5)
```

(1) 镜像流的名称。

(2) Docker 存储库路径，在此处可推送(push)新镜像，以在此镜像流中添加/更新镜像。

(3) 此镜像流标签当前引用的 SHA 标识符。引用此镜像流标签的资源使用此标识符。

(4) 此镜像流标签之前引用的 SHA 标识符。可用于回滚至旧镜像。

(5) 镜像流标签名称。

有关引用镜像流的示例构建配置，请参阅配置的 **Strategy** 小节中的 BuildConfig? 是什么。

有关引用镜像流的示例部署配置，请参阅在配置的 **Strategy** 部分中创建部署配置。

3.5.2.3. 镜像流镜像

一个*镜像流镜像*会从一个镜像流内指向一个特定的镜像 ID。

镜像流镜像允许您从标记了镜像的特定镜像流中检索有关镜像的元数据。

每当您将镜像导入或标记到镜像流时，OpenShift Container Platform 中会自动创建镜像流镜像对象。您不必在用于创建镜像流的任何镜像流定义中显式定义镜像流镜像对象。

镜像流镜像包含来自存储库的镜像流名称和镜像 ID，用 @ 符号分隔：

```
<image-stream-name>@<image-id>
```

要引用 上面的镜像流对象示例中的镜像，镜像流镜像类似如下：

```
origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
```

3.5.2.4. 镜像流标签

*镜像流标签*是指向*镜像流*中的一个镜像的命名指针。通常缩写为 *istag*。镜像流标签用于引用或检索给定镜像流和标签的镜像。

镜像流标签可引用任何本地管理或外部管理的镜像。它包含镜像历史记录，表示为标签曾指向的所有镜像的堆栈。每当特定镜像流标签下标记了新的或现有镜像时，该镜像将置于历史记录堆栈的第一位置。之前占据第一位置的镜像将移至第二位置，以此类推。这样便于回滚，从而让标签再次指向历史镜像。

以下镜像流标签来自 上面的 镜像流标签 ：

历史记录中带有两个镜像的镜像流标签

```
  tags:
  - items:
    - created: 2017-09-02T10:15:09Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
      generation: 2
      image: sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
    - created: 2017-09-29T13:40:11Z
      dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:909de62d1f609a717ec433cc25ca5cf00941545c83a01fb31527771e1fab3fc5
      generation: 1
      image: sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
    tag: latest
```

镜像流标签可以是 *持久性标签*，也可以是 *跟踪标签*。

- *持久性标签*是特定于版本的标签，指向镜像的特定版本，如 Python 3.5。
- *跟踪标签*是*引用* 标签，跟随另一个镜像流标签，并在以后更新以更改它们跟随的镜像，这与符号链接非常相似。请注意，这些新等级无法保证向后兼容。
  例如，OpenShift Container Platform 附带的 **latest** 镜像流标签是跟踪标签。这意味着，当有新级别可用时，**latest** 镜像流标签的用户会更新为镜像提供的最新框架级别。指向 **v3.10** 的 **latest** 镜像流标签可以随时更改为 **v3.11**。请务必注意，这些 **latest** 镜像流标签的行为与 Docker **latest** 标签不同。在本例中，**latest** 镜像流标签不指向 Docker 存储库中的最新镜像。它指向另一个镜像

流标签，可能并非镜像的最新版本。例如，如果 **latest** 镜像流标签指向 **v3.10** 镜像，则当发布了 **3.11** 版时，**latest** 标签不会自动更新至 **v3.11**，并保持 **v3.10**，直到手动更新为指向 **v3.11** 镜像流标签。

注意

跟踪标签仅限于单个镜像流，无法引用其他镜像流。

您可以根据自己的需要创建自己的镜像流标签。请参阅 推荐标记。

镜像流标签由镜像流名称和一个标签组成，用冒号隔开：

```
<image stream name>:<tag>
```

例如，要引用 上面的 镜像流对象示例中的 **sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d** 镜像，镜像流标签将是：

```
origin-ruby-sample:latest
```

3.5.2.5. 镜像流更改触发器

当有新版本上游镜像时，镜像流触发器支持自动调用构建和部署。

例如，修改镜像流标签时，构建和部署可以自动启动。实现方法是通过监控特定镜像流标签并在检测到变化时通知构建或部署。

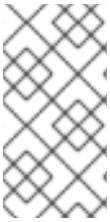
ImageChange 触发器会在镜像流标签内容更改时（ 推送镜像的新版本时 ）产生新的复制控制器。

ImageChange 触发器

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

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

在上例中，当 **origin-ruby-sample** 镜像流的 **latest** 标签值更改并且新镜像值与部署配置的 **helloworld** 容器中指定的当前镜像不同时，会使用 **helloworld** 容器的新镜像创建新的复制控制器。

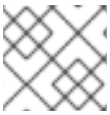


注意

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

3.5.2.6. 镜像流映射

当 集成的 registry 收到新镜像时，它会创建镜像流映射并将其发送到 OpenShift Container Platform，从而提供镜像的项目、名称、标签和镜像元数据。



注意

配置镜像流映射是高级功能。

此信息用于创建新镜像（如果尚未存在）并将此镜像标记到镜像流中。OpenShift Container Platform 存储每个镜像的完整元数据，如命令、入口点和环境变量。OpenShift Container Platform 中的镜像不可变，名称最长 63 个字符。

注意

有关手动标记镜像的详情，请参阅 Developer Guide。

以下镜像流映射示例结果是将镜像标记为 **test/origin-ruby-sample:latest**：

镜像流映射对象定义

```
apiVersion: v1
kind: ImageStreamMapping
metadata:
  creationTimestamp: null
  name: origin-ruby-sample
  namespace: test
tag: latest
image:
  dockerImageLayers:
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c29
    size: 196634330
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef
    size: 0
  - name: sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea092
    size: 177723024
  - name: sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b6e
    size: 55679776
  - name: sha256:92114219a04977b5563d7dff71ec4caa3a37a15b266ce42ee8f43dba9798c966
    size: 11939149
  dockerImageMetadata:
    Architecture: amd64
    Config:
```

```
      Cmd:
      - /usr/libexec/s2i/run
      Entrypoint:
      - container-entrypoint
      Env:
      - RACK_ENV=production
      - OPENSHIFT_BUILD_NAMESPACE=test
      - OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
      - EXAMPLE=sample-app
      - OPENSHIFT_BUILD_NAME=ruby-sample-build-1
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
      ExposedPorts:
        8080/tcp: {}
      Labels:
        build-date: 2015-12-23
        io.k8s.description: Platform for building and running Ruby 2.2 applications
        io.k8s.display-name: 172.30.56.218:5000/test/origin-ruby-sample:latest
        io.openshift.build.commit.author: Ben Parees <bparees@users.noreply.github.com>
        io.openshift.build.commit.date: Wed Jan 20 10:14:27 2016 -0500
        io.openshift.build.commit.id: 00cadc392d39d5ef9117cbc8a31db0889eedd442
        io.openshift.build.commit.message: 'Merge pull request #51 from php-coder/fix_url_and_sti'
        io.openshift.build.commit.ref: master
        io.openshift.build.image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
        io.openshift.build.source-location: https://github.com/openshift/ruby-hello-world.git
        io.openshift.builder-base-version: 8d95148
        io.openshift.builder-version: 8847438ba06307f86ac877465eadc835201241df
        io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
        io.openshift.tags: builder,ruby,ruby22
        io.s2i.scripts-url: image:///usr/libexec/s2i
        license: GPLv2
        name: CentOS Base Image
        vendor: CentOS
      User: "1001"
      WorkingDir: /opt/app-root/src
    Container: 86e9a4a3c760271671ab913616c51c9f3cea846ca524bf07c04a6f6c9e103a76
    ContainerConfig:
      AttachStdout: true
      Cmd:
      - /bin/sh
      - -c
      - tar -C /tmp -xf - && /usr/libexec/s2i/assemble
      Entrypoint:
      - container-entrypoint
      Env:
      - RACK_ENV=production
      - OPENSHIFT_BUILD_NAME=ruby-sample-build-1
      - OPENSHIFT_BUILD_NAMESPACE=test
```

```
      - OPENSHIFT_BUILD_SOURCE=https://github.com/openshift/ruby-hello-world.git
      - EXAMPLE=sample-app
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
      ExposedPorts:
        8080/tcp: {}
      Hostname: ruby-sample-build-1-build
      Image: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
      OpenStdin: true
      StdinOnce: true
      User: "1001"
      WorkingDir: /opt/app-root/src
    Created: 2016-01-29T13:40:00Z
    DockerVersion: 1.8.2.fc21
    Id: 9d7fd5e2d15495802028c569d544329f4286dcd1c9c085ff5699218dbaa69b43
    Parent: 57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
    Size: 441976279
    apiVersion: "1.0"
    kind: DockerImage
  dockerImageMetadataVersion: "1.0"
  dockerImageReference: 172.30.56.218:5000/test/origin-ruby-
sample@sha256:47463d94eb5c049b2d23b03a9530bf944f8f967a0fe79147dd6b9135bf7dd13d
```

3.5.2.7. 使用镜像流

以下小节介绍了如何使用镜像流和镜像流标签。有关使用镜像流的更多信息，请参阅管理镜像。

3.5.2.7.1. 获取有关镜像流的信息

要获取有关镜像流的常规信息及其指向的所有标签的详细信息，请使用以下命令：

```
$ oc describe is/<image-name>
```

例如：

```
$ oc describe is/python
```

输出示例

```
Name:   python
Namespace:  default
Created:  About a minute ago
Labels:   <none>
Annotations:  openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
Docker Pull Spec: docker-registry.default.svc:5000/default/python
Image Lookup:  local=false
```

```
Unique Images:  1
Tags:   1

3.5
  tagged from centos/python-35-centos7

  * centos/python-35-centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
      About a minute ago
```

要获取有关特定镜像流标签的所有可用信息：

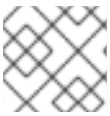
```
$ oc describe istag/<image-stream>:<tag-name>
```

例如：

```
$ oc describe istag/python:latest
```

输出示例

```
Image Name: sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Docker Image: centos/python-35-
centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Name:  sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
Created: 2 minutes ago
Image Size: 251.2 MB (first layer 2.898 MB, last binary layer 72.26 MB)
Image Created: 2 weeks ago
Author:  <none>
Arch:  amd64
Entrypoint: container-entrypoint
Command: /bin/sh -c $STI_SCRIPTS_PATH/usage
Working Dir: /opt/app-root/src
User:  1001
Exposes Ports: 8080/tcp
Docker Labels: build-date=20170801
```



注意

输出的信息多于显示的信息。

3.5.2.7.2. 在镜像流中添加额外标签

要添加指向其中一个现有标签的标签，您可以使用 **oc tag** 命令：

```
oc tag <image-name:tag> <image-name:tag>
```

例如：

```
$ oc tag python:3.5 python:latest
```

输出示例

```
Tag python:latest set to
python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25.
```

使用 **oc describe** 命令确认镜像流有两个标签，一个(**3.5**)指向外部容器镜像，另一个标签(**latest**)指向同一镜像，因为它基于第一个标签创建。

```
$ oc describe is/python
```

输出示例

```
Name:   python
Namespace:  default
Created:  5 minutes ago
Labels:   <none>
Annotations:  openshift.io/image.dockerRepositoryCheck=2017-10-02T17:05:11Z
Docker Pull Spec: docker-registry.default.svc:5000/default/python
Image Lookup:  local=false
Unique Images:  1
Tags:   2

latest
  tagged from python@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25

  * centos/python-35-centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
      About a minute ago

3.5
  tagged from centos/python-35-centos7

  * centos/python-35-centos7@sha256:49c18358df82f4577386404991c51a9559f243e0b1bdc366df25
      5 minutes ago
```

3.5.2.7.3. 为外部镜像添加标签

对所有标签相关的操作使用 **oc tag** 命令，如添加标签指向内部或外部镜像：

```
$ oc tag <repositiory/image> <image-name:tag>
```

例如，该命令可将 **docker.io/python:3.6.0** 镜像映射到 **python** 镜像流中的 **3.6** 标签。

```
$ oc tag docker.io/python:3.6.0 python:3.6
```

输出示例

```
Tag python:3.6 set to docker.io/python:3.6.0.
```

如果外部镜像安全，则需要创建带有用于访问该 registry 的凭证的 secret。如需了解更多详细信息，请参阅从私有 registry 中导入镜像。

3.5.2.7.4. 更新镜像流标签

要更新标签以反映镜像流中的另一标签：

```
$ oc tag <image-name:tag> <image-name:latest>
```

例如，以下命令更新了 **latest** 标签，以反映镜像流中的 **3.6** 标签：

```
$ oc tag python:3.6 python:latest
```

输出示例

```
Tag python:latest set to
python@sha256:438208801c4806548460b27bd1fbcb7bb188273d13871ab43f.
```

3.5.2.7.5. 从镜像流中删除镜像流标签

从镜像流中删除旧标签：

```
$ oc tag -d <image-name:tag>
```

例如：

```
$ oc tag -d python:3.5
```

输出示例

```
Deleted tag default/python:3.5.
```

3.5.2.7.6. 配置标签定期导入

使用外部容器镜像 registry 时，要定期重新导入镜像（例如，获取最新的安全更新），请使用 **--scheduled** 标志：

```
$ oc tag <repositiory/image> <image-name:tag> --scheduled
```

例如：

```
$ oc tag docker.io/python:3.6.0 python:3.6 --scheduled
```

输出示例

```
Tag python:3.6 set to import docker.io/python:3.6.0 periodically.
```

该命令可使 OpenShift Container Platform 定期更新该特定镜像流标签。此周期是集群范围的设置，默认设为 15 分钟。

要删除定期检查，请重新运行上述命令，但省略 **--scheduled** 标志。这会将其行为重置为默认值。

```
$ oc tag <repositiory/image> <image-name:tag>
```

3.6. DEPLOYMENTS

3.6.1. 复制控制器

复制控制器确保任何时候都运行指定数量的 pod 副本。如果 pod 退出或被删除，复制控制器会做出反应，实例化更多 pod 来达到定义的数量。同样，如果运行中的数量超过所需的数目，它会根据需要删除相应数量的 Pod，使其与定义的数量相符。

复制控制器配置包括：

1. 所需的副本数（可在运行时调整）。
2. 创建复制容器集时要使用的容器集定义。
3. 用于标识受管 pod 的选择器。

选择器是分配给由复制控制器管理的 pod 的一组标签。这些标签包含在复制控制器实例化的容器集定义中。复制控制器使用选择器来决定已在运行的 pod 实例数量，以便根据需要进行调整。

复制控制器不会基于负载或流量执行自动扩展，因为复制控制器不会跟踪它们。相反，这需要外部自动缩放器调整其副本数。

复制控制器是名为 **ReplicationController** 的核心 Kubernetes 对象。

以下是 **ReplicationController** 定义示例：

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1 1
  selector:  2
    name: frontend
  template:  3
    metadata:
      labels:  4
        name: frontend 5
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1. 要运行的 pod 的副本数。
2. 要运行的 pod 的标签选择器。
3. 控制器创建的 pod 模板。
4. pod 上的标签应该包括标签选择器中的标签。
5. 扩展任何参数后的最大名称长度为 63 个字符。

3.6.2. 副本集

与 复制控制器 类似，副本集确保在任何给定时间运行指定数量的 pod 副本。副本集与复制控制器之间的区别在于，副本集支持基于集合的选择器要求，而复制控制器只支持基于相等的选择器要求。

注意

只有在您需要自定义更新编配或根本不需要更新时，才使用副本集，否则使用 Deployment。副本集可以独立使用，但由部署使用用来编配 pod 创建、删除和更新。部署会自动管理其副本集，为 pod 提供声明性更新，且不需要手动管理它们创建的副本集。

副本集是一个被称为 **ReplicaSet** 的核心 Kubernetes 对象。

以下是 **ReplicaSet** 定义示例：

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector: (1)
    matchLabels: (2)
      tier: frontend
    matchExpressions: (3)
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

(1) 对一组资源进行的标签查询。**matchLabels** 和 **matchExpressions** 的结果在逻辑上是组合在一起的。

(2) 基于相等的选择器，使用与选择器匹配的标签指定资源。

(3) 基于集合的选择器，用于过滤键。这将选择键等于 **tier** 并且值等于 **frontend** 的所有资源。

3.6.3. Jobs

作业类似于复制控制器，其目的在于出于指定的原因创建 pod。不同之处在于复制控制器是为持续运行的容器集设计的，而作业则用于一次性容器集。作业跟踪所有成功完成，并且达到指定数量的完成时，作业本身将完成。

以下示例计算 π 到 2000 个位置，输出它，然后完成：

```
apiVersion: extensions/v1
kind: Job
metadata:
  name: pi
spec:
  selector:
    matchLabels:
      app: pi
  template:
    metadata:
      name: pi
      labels:
        app: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

有关如何使用作业（job）的更多信息，请参阅作业。

3.6.4. 部署和部署配置

在复制控制器的基础上，OpenShift Container Platform 添加了对软件开发和部署生命周期的支持以及部署的概念。在最简单的情形中，部署仅创建一个新复制控制器，并允许它启动 pod。但是，OpenShift Container Platform 部署也提供从镜像的现有部署过渡到新部署的功能，还可定义要在创建复制控制器之前或之后运行的 hook。

OpenShift Container Platform **DeploymentConfig** 对象定义以下部署详情：

1. **ReplicationController** 定义的元素。
2. 自动创建新部署的触发器。
3. 在部署之间过渡的策略。
4. 生命周期 Hook.

每次触发部署时，无论是手动还是自动，部署器 Pod 均管理部署（包括缩减旧复制控制器、扩展新复制控制器以及运行 hook）。部署 pod 在完成部署后会无限期保留，以便保留其部署日志。当部署被另一个部署替换时，以前的复制控制器会被保留，以便在需要时轻松回滚。

有关如何创建和与部署交互的详细信息，请参阅 Deployment。

以下是含有一些省略和标注的 **DeploymentConfig** 定义示例：

```
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
```

```
    name: frontend
  template: { ... }
  triggers:
  - type: ConfigChange (1)
  - imageChangeParams:
      automatic: true
      containerNames:
      - helloworld
      from:
        kind: ImageStreamTag
        name: hello-openshift:latest
    type: ImageChange (2)
  strategy:
    type: Rolling (3)
```

1. **ConfigChange** 触发器会在复制控制器模板更改时触发新部署的创建。
2. 每当指定镜像流中有新版本的后备镜像可用时，ImageChange **触发器** 会导致创建新部署。
3. 默认的 **Rolling** 策略会在部署之间实现无停机过渡。

3.7. 模板

3.7.1. 概述

模板描述了一组可参数化和处理的对象，用于生成对象列表，供 OpenShift Container Platform 用于创建。要创建的对象可以包含用户有权在项目内创建的任何对象，如 服务、构建配置和 部署配置。模板还可定义一系列标签（label），以应用到该模板中定义的每个对象。

有关创建和使用 模板的详细信息，请参阅模板指南。

第 4 章 其他概念

4.1. 身份验证

4.1.1. 概述

身份验证层识别与 OpenShift Container Platform API 请求关联的用户。然后，授权层使用与请求用户相关的信息来确定是否应允许该请求。

作为管理员，您可以使用 主（master）配置文件 配置身份验证。

4.1.2. 用户和组

OpenShift Container Platform 中的*用户*是可以向 OpenShift Container Platform API 发出请求的实体。通常，这代表与 OpenShift Container Platform 交互的开发人员或管理员的帐户。

用户可以分配到一个或多个*组*中，每个组代表特定的用户集合。在 管理授权策略 时，可以同时为多个用户授予权限，例如允许访问 项目内 的对象，而不必逐一向用户授予权限。

除了显式定义的组外，还有 OpenShift 自动调配的 *系统组或虚拟组*。在 查看集群绑定 时可以看到它们。

在默认虚拟组集合中，特别注意以下内容：

虚拟组	描述
system:authenticated	自动与所有经过身份验证的用户关联。
system:authenticated:oauth	自动与所有使用 OAuth 访问令牌经过身份验证的用户关联。
system:unauthenticated	自动与所有未经身份验证的用户关联。

4.1.3. API 身份验证

对 OpenShift Container Platform API 的请求通过以下方式进行身份验证：

OAuth 访问令牌

- 使用 <master> ***/oauth/authorize 和 <master>/ oauth/*token** 端点从 OpenShift Container Platform OAuth 服务器获取。
- 作为 **Authorization 发送：Bearer...** 标头发送。
- 以 **base64url.bearer.authorization.k8s.io.<base64url-encoded-token>** 形式，作为 websocket 请求的 websocket 子协议标头发送。

X.509 客户端证书

- 需要与 API 服务器的 HTTPS 连接。
- 由 API 服务器针对可信证书颁发机构捆绑包进行验证。

- API 服务器创建证书并分发到控制器，以对自身进行身份验证。

任何具有无效访问令牌或无效证书的请求都会被身份验证层以 401 错误形式拒绝。

如果没有出示访问令牌或证书，身份验证层会将 **system:anonymous** 虚拟用户和 **system:unauthenticated** 虚拟组分配给请求。这使得授权层能够决定匿名用户可以发出哪些（如有）请求。

4.1.3.1. 模拟（Impersonation）

对 OpenShift Container Platform API 的请求可以包含 **Impersonate-User** 标头，这表示请求者希望像来自指定用户一样处理请求。您可以通过在请求中添加 **--as=<user>** 标志来模拟用户。

在用户 A 可以模拟用户 B 之前，用户 A 通过了身份验证。然后，进行授权检查以确保 User A 可以模仿名为 User B 的用户。如果用户 A 请求模拟服务帐户 **system:serviceaccount:namespace:name**，OpenShift Container Platform 会确认 User A 可以模仿名为 **name** 的 **serviceaccount** 在 **命名空间中**。如果检查失败，请求会失败，并显示 403(Forbidden)错误代码。

默认情况下，项目管理员和编辑器可以模拟其命名空间中的服务帐户。**sudoers** 角色允许用户模拟 **system:admin**，后者具有集群管理员权限。通过模拟 **system:admin**，可以在用户管理集群时提供一定的拼写错误（而非安全性）的保护。例如，运行 **oc delete nodes --all** 会失败，但运行 **oc delete nodes --all --as=system:admin** 会失败。您可以运行以下命令来授予该权限的用户：

```
$ oc create clusterrolebinding <any_valid_name> --clusterrole=sudoer --user=<username>
```

如果您需要代表用户创建项目请求，请在命令中包含 **--as=<user> --as-group=<group1> --as-group=<group2>** 标志。因为 **system:authenticated:oauth** 是唯一可以创建项目请求的 bootstrap 组，所以您必须模拟该组，如下例所示：

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

4.1.4. OAuth

OpenShift Container Platform master 包含内置的 OAuth 服务器。用户获取 OAuth 访问令牌来对自身进行 API 身份验证。

当用户请求新的 OAuth 令牌时，OAuth 服务器使用配置的 身份提供程序 来确定发出请求的人员的身份。

然后，它会确定该身份所映射到的用户，为该用户创建一个访问令牌，再返回要使用的令牌。

4.1.4.1. OAuth 客户端

每个对 OAuth 令牌的请求都必须指定要接收和使用令牌的 OAuth 客户端。启动 OpenShift Container Platform API 时会自动创建以下 OAuth 客户端：

OAuth 客户端	使用方法
openshift-web-console	为 Web 控制台请求令牌。

OAuth 客户端	使用方法
openshift-browser-client	使用可处理交互式登录的用户代理，在 ***<master>/oauth/token/request*** 请求令牌。
openshift-challenging-client	使用可处理 **WWW-Authenticate** 质询的用户代理来请求令牌。

注册其他客户端：

```
$ oc create -f <(echo '
kind: OAuthClient
apiVersion: oauth.openshift.io/v1
metadata:
 name: demo (1)
secret: "..." (2)
redirectURIs:
 - "http://www.example.com/" (3)
grantMethod: prompt (4)
')
```

1. OAuth 客户端 **的名称** 用作提交对 <master> ***/oauth/authorize 和 <master>/ oauth/*token** 的请求时的 **client_id** 参数。
2. 在提出对 ***<master>*/oauth/token** 的请求时， **secret** 将作为 **client_secret** 参数使用。
3. 在对 ***<master>*/oauth/authorize** 和 ***<master>*/oauth/token** 的请求中指定的 **redirect_uri** 参数必须等于 **redirectURIs** 中的某一 URI（或使用它作为前缀）。
4. **grantMethod** 用于确定当此客户端请求了令牌且还未被用户授予访问权限时应采取的操作。使用 Grant 选项中看到的相同值。

4.1.4.2. 服务帐户作为 OAuth 客户端

服务帐户可用作受约束形式的 OAuth 客户端。服务帐户只能请求范围的子集，允许访问服务帐户本身的命名空间内的一些基本用户信息和基于角色的功能：

- **user:info**
- **user:check-access**
- **role:<any_role>:<serviceaccount_namespace>**
- **role:<any_role>:<serviceaccount_namespace>:!**

在将服务帐户用作 OAuth 客户端时：

- **client_id** 是 **system:serviceaccount:<serviceaccount_namespace>:<serviceaccount_name>**。
- **client_secret** 可以是该服务帐户的任何 API 令牌。例如：

```
$ oc sa get-token <serviceaccount_name>
```

- 要获取 **WWW-Authenticate** 质询，请将服务帐户上的 **serviceaccounts.openshift.io/oauth-want-challenges** 注解设置为 **true**。
- **redirect_uri** 必须与服务帐户上的注解匹配。服务帐户的重定向 URI 作为 OAuth 客户端提供了更多信息。

4.1.4.3. 重定向作为 OAuth 客户端的服务帐户的 URI

注解键必须具有前缀 **serviceaccounts.openshift.io/oauth-redirecturi.** 或 **serviceaccounts.openshift.io/oauth-redirectreference.**，例如：

```
serviceaccounts.openshift.io/oauth-redirecturi.<name>
```

采用最简单形式时，注解可用于直接指定有效的重定向 URI。例如：

```
"serviceaccounts.openshift.io/oauth-redirecturi.first":  "https://example.com"
"serviceaccounts.openshift.io/oauth-redirecturi.second": "https://other.com"
```

上例中的 **first** 和 **second** 后缀用于分隔两个有效的重定向 URI。

在更复杂的配置中，静态重定向 URI 可能还不够。例如，您可能希望路由的所有入口都被视为有效。这时可使用通过 **serviceaccounts.openshift.io/oauth-redirectreference.** 前缀的动态重定向 URI。

例如：

```
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{\"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{\"kind\":\"Route\",\"name\":\"jenkins\"}}"
```

由于此注解的值包含序列化 JSON 数据，因此在扩展格式中可以更轻松地查看：

```
{
  "kind": "OAuthRedirectReference",
  "apiVersion": "v1",
  "reference": {
    "kind": "Route",
    "name": "jenkins"
  }
}
```

您现在可以看到，**OAuthRedirectReference** 允许引用名为 **jenkins** 的路由。因此，该路由的所有入口现在都被视为有效。**OAuthRedirectReference** 的完整规格是：

```
{
  "kind": "OAuthRedirectReference",
  "apiVersion": "v1",
  "reference": {
    "kind": ..., (1)
    "name": ..., (2)
```

```
    "group": ... 3
  }
}
```

1. **kind** 指的是被引用对象的类型。目前，只支持 **route**。
2. **name** 指的是项目的名称。对象必须与服务帐户位于同一命名空间中。
3. **group** 指的是对象所属的组。此项留空，因为路由的组是空字符串。

可以合并这两个注解前缀，来覆盖引用对象提供的数据。例如：

```
"serviceaccounts.openshift.io/oauth-redirecturi.first": "custompath"
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{\"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{\"kind\":\"Route\",\"name\":\"jenkins\"}}"
```

first 后缀用于将注解绑在一起。假设 **jenkins** 路由的入口为 *https://example.com*，现在 *https://example.com/custompath* 被视为有效，但 *https://example.com* 视为无效。部分提供覆盖数据的格式如下：

类型	语法
Scheme	"https://"
主机名	"//website.com"
端口	"//:8000"
路径	"examplepath"

注意

指定主机名覆盖将替换被引用对象的主机名数据，这不一定是需要的行为。

以上语法的任何组合都可以使用以下格式进行合并：

<scheme:>//<hostname><:port>/<path>

同一对象可以被多次引用，以获得更大的灵活性：

```
"serviceaccounts.openshift.io/oauth-redirecturi.first": "custompath"
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{\"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{\"kind\":\"Route\",\"name\":\"jenkins\"}}"
"serviceaccounts.openshift.io/oauth-redirecturi.second": "//:8000"
"serviceaccounts.openshift.io/oauth-redirectreference.second": "
{\"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{\"kind\":\"Route\",\"name\":\"jenkins\"}}"
```

假设名为 **jenkins** 的路由具有入口 *https://example.com*，则 *https://example.com:8000* 和 *https://example.com/custompath* 都被视为有效。

可以同时使用静态和动态注解，以实现所需的行为：

```
"serviceaccounts.openshift.io/oauth-redirectreference.first": "
{\"kind\":\"OAuthRedirectReference\",\"apiVersion\":\"v1\",\"reference\":
{\"kind\":\"Route\",\"name\":\"jenkins\"}}"
"serviceaccounts.openshift.io/oauth-redirecturi.second": "https://other.com"
```

4.1.4.3.1. OAuth 的 API 事件

在有些情况下，API 服务器会返回一个 **unexpected condition** 错误消息；若不直接访问 API 主日志，很难对此消息进行调试。该错误的基本原因被有意遮挡，以避免向未经身份验证的用户提供有关服务器状态的信息。

这些错误的子集与服务帐户 OAuth 配置问题有关。这些问题在非管理员用户可查看的事件中捕获。OAuth 期间遇到 **unexpected condition** 服务器错误时，可运行 **oc get events** 在 **ServiceAccount** 下查看这些事件。

以下示例警告缺少正确 OAuth 重定向 URI 的服务帐户：

```
$ oc get events | grep ServiceAccount
```

输出示例

```
1m 1m 1 proxy ServiceAccount Warning
NoSAOAuthRedirectURIs service-account-oauth-client-getter
system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-
redirecturi.<some-value>=<redirect> or create a dynamic URI using
serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>
```

运行 **oc describe sa/<service-account-name>** 可以报告与给定服务帐户名称相关的任何 OAuth 事件。

```
$ oc describe sa/proxy | grep -A5 Events
```

输出示例

```
Events:
  FirstSeen     LastSeen        Count   From                            SubObjectPath   Type          Reason
Message
  ---------     --------        -----   ----                            -------------   --------      ------                -------
  3m            3m              1       service-account-oauth-client-getter                     Warning
NoSAOAuthRedirectURIs   system:serviceaccount:myproject:proxy has no redirectURIs; set
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic URI
using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>
```

下方列出可能的事件错误：

无重定向 URI 注解或指定了无效的 URI

```
Reason                  Message
NoSAOAuthRedirectURIs   system:serviceaccount:myproject:proxy has no redirectURIs; set
```

```
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic URI
using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>
```

指定了无效的路由

```
Reason                  Message
NoSAOAuthRedirectURIs   [routes.route.openshift.io "<name>" not found,
system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-
redirecturi.<some-value>=<redirect> or create a dynamic URI using
serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>]
```

指定了无效的引用类型

```
Reason                  Message
NoSAOAuthRedirectURIs   [no kind "<name>" is registered for version "v1",
system:serviceaccount:myproject:proxy has no redirectURIs; set serviceaccounts.openshift.io/oauth-
redirecturi.<some-value>=<redirect> or create a dynamic URI using
serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>]
```

缺少 SA 令牌

```
Reason                  Message
NoSAOAuthTokens         system:serviceaccount:myproject:proxy has no tokens
```

4.1.4.3.1.1. 因为可能的错误配置导致的示例 API 事件

以下步骤代表用户可以进入出现问题状态的一种方法，以及如何调试或修复问题：

1. 将服务帐户用作 OAuth 客户端，创建项目。

   a. 为代理服务帐户对象创建 YAML，并确保它使用路由**代理** ：

      ```
      $ vi serviceaccount.yaml
      ```

      添加以下示例代码：

      ```
      apiVersion: v1
      kind: ServiceAccount
      metadata:
        name: proxy
        annotations:
          serviceaccounts.openshift.io/oauth-redirectreference.primary:
      '{"kind":"OAuthRedirectReference","apiVersion":"v1","reference":
      {"kind":"Route","name":"proxy"}}'
      ```

   b. 为路由对象创建 YAML，以创建与代理的安全连接：

      ```
      $ vi route.yaml
      ```

      添加以下示例代码：

      ```
      apiVersion: route.openshift.io/v1
      kind: Route
      ```

```
metadata:
  name: proxy
spec:
  to:
    name: proxy
  tls:
    termination: Reencrypt
apiVersion: v1
kind: Service
metadata:
  name: proxy
  annotations:
    service.alpha.openshift.io/serving-cert-secret-name: proxy-tls
spec:
  ports:
  - name: proxy
    port: 443
    targetPort: 8443
  selector:
    app: proxy
```

c. 为部署配置创建 YAML 以作为 sidecar 启动代理：

```
$ vi proxysidecar.yaml
```

添加以下示例代码：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: proxy
  template:
    metadata:
      labels:
        app: proxy
    spec:
      serviceAccountName: proxy
      containers:
      - name: oauth-proxy
        image: openshift3/oauth-proxy
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8443
          name: public
        args:
        - --https-address=:8443
        - --provider=openshift
        - --openshift-service-account=proxy
        - --upstream=http://localhost:8080
        - --tls-cert=/etc/tls/private/tls.crt
```

```
    - --tls-key=/etc/tls/private/tls.key
    - --cookie-secret=SECRET
    volumeMounts:
    - mountPath: /etc/tls/private
      name: proxy-tls

  - name: app
    image: openshift/hello-openshift:latest
  volumes:
  - name: proxy-tls
    secret:
      secretName: proxy-tls
```

d. 创建三个对象：

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

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

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

2. 运行 **oc edit sa/proxy** 以编辑服务帐户，并将 **serviceaccounts.openshift.io/oauth-redirectreference** 注解更改为指向不存在的路由。

```
apiVersion: v1
imagePullSecrets:
- name: proxy-dockercfg-08d5n
kind: ServiceAccount
metadata:
  annotations:
    serviceaccounts.openshift.io/oauth-redirectreference.primary:
'{"kind":"OAuthRedirectReference","apiVersion":"v1","reference":
{"kind":"Route","name":"notexist"}}'
...
```

3. 查看服务的 OAuth 日志以查找服务器错误：

```
The authorization server encountered an unexpected condition that prevented it from fulfilling
the request.
```

4. 运行 **oc get events** 来查看 **ServiceAccount** 事件：

```
$ oc get events | grep ServiceAccount
```

输出示例

```
23m 23m 1 proxy ServiceAccount Warning
NoSAOAuthRedirectURIs service-account-oauth-client-getter [routes.route.openshift.io
"notexist" not found, system:serviceaccount:myproject:proxy has no redirectURIs; set
serviceaccounts.openshift.io/oauth-redirecturi.<some-value>=<redirect> or create a dynamic
URI using serviceaccounts.openshift.io/oauth-redirectreference.<some-value>=<reference>]
```

4.1.4.4. 集成

所有对 OAuth 令牌的请求都包括对 ***<master>*/oauth/authorize** 的请求。大多数身份验证集成在此端点前放置身份验证代理，或者将 OpenShift Container Platform 配置为针对后备 身份提供程序 验证凭证。对 ***<master>*/oauth/authorize** 的请求可能来自不能显示交互式登录页面的用户代理，如 CLI。因此，除了交互式登录流程外，OpenShift Container Platform 也支持使用 **WWW-Authenticate** 质询进行验证。

如果在 ***<master>*/oauth/authorize** 端点前面放置身份验证代理，它会向未经身份验证的非浏览器用户代理发送 **WWW-Authenticate** 质询，而不是显示交互式登录页面或重定向到交互式登录流程。

注意

为防止浏览器客户端遭受跨站点请求伪造(CSRF)攻击，只有在请求中存在 **X-CSRF-Token** 标头时，才应发送基本身份验证质询。希望接收基本 **WWW-Authenticate** 质询的客户端应将此标头设置为非空值。

如果身份验证代理不支持 **WWW-Authenticate** 质询，或者如果 OpenShift Container Platform 配置为使用不支持 WWW-Authenticate 质询的身份提供程序，用户可以使用浏览器访问 ***<master>* /oauth/token/request** 来手动获取访问令牌。

4.1.4.5. OAuth Server Metadata

在 OpenShift Container Platform 中运行的应用可能需要发现有关内置 OAuth 服务器的信息。例如，他们可能需要发现没有手工配置的 **<master>** 服务器的地址。为此，OpenShift Container Platform 实施了 IETF OAuth 2.0 授权服务器元数据草案规范。

因此，集群中运行的任何应用程序都可以向 ***https://openshift.default.svc/.well-known/oauth-authorization-server*** 发出 **GET** 请求来获取以下信息：

```
{
  "issuer": "https://<master>", (1)
  "authorization_endpoint": "https://<master>/oauth/authorize", (2)
  "token_endpoint": "https://<master>/oauth/token", (3)
  "scopes_supported": [ (4)
    "user:full",
    "user:info",
    "user:check-access",
    "user:list-scoped-projects",
    "user:list-projects"
  ],
  "response_types_supported": [ (5)
    "code",
    "token"
  ],
  "grant_types_supported": [ (6)
    "authorization_code",
    "implicit"
  ],
  "code_challenge_methods_supported": [ (7)
    "plain",
    "S256"
  ]
}
```

1. 授权服务器的签发者标识符，它是使用 **https** 方案且没有查询或分段组件的 URL。这是包含授权服务器有关信息的 **.well-known** RFC 5785 资源的发布位置。
2. 授权服务器的授权端点的 URL。参见 RFC 6749。
3. 授权服务器的令牌端点的 URL。参见 RFC 6749。
4. 包含此授权服务器支持的 OAuth 2.0 RFC 6749 范围值列表的 JSON 数组。请注意，并非所有支持的范围值都会公告。
5. 包含此授权服务器支持的 OAuth 2.0 **response_type** 值列表的 JSON 数组。使用的数组值与 **response_types** 参数（根据 RFC 7591 中"OAuth 2.0 Dynamic Client Registration Protocol"定义）使用的数组值相同。
6. 包含此授权服务器支持的 OAuth 2.0 授权类型值列表的 JSON 数组。使用的数组值与通过 **grant_types** 参数（根据 RFC 7591 中"**OAuth 2.0 Dynamic Client Registration Protocol**"定义）使用的数组值相同。
7. 包含此授权服务器支持的 PKCE RFC 7636 代码质询方法列表的 JSON 数组。**code_challenge_method** 参数中使用的代码质询方法值在 RFC 7636 第 4.3 节 中定义。有效的代码质询方法值是在 IANA **PKCE Code Challenge Methods** 注册表中注册的值。请参阅 IANA OAuth 参数。

4.1.4.6. 获取 OAuth 令牌

OAuth 服务器支持标准的授权代码授权和隐式授权 OAuth 授权流。

运行以下命令，使用授权代码授权方法请求 OAuth 令牌：

```
$ curl -H "X-Remote-User: <username>" \
    --cacert /etc/origin/master/ca.crt \
    --cert /etc/origin/master/admin.crt \
    --key /etc/origin/master/admin.key \
    -I https://<master-address>/oauth/authorize?response_type=token\&client_id=openshift-challenging-client | grep -oP "access_token=\K[^&]*"
```

在使用隐式授权流 (**response_type=token**) 以及配置为请求 **WWW-Authenticate** 质询（如 **openshift-challenging-client**）的 client_id 以请求 OAuth 令牌时，可能来自 **/oauth/authorize** 的服务器响应及它们的处理方式如下方所列：

状态	内容	客户端响应
302	**Location** 标头含有 URL 片段中的 **access_token** 参数 (RFC 4.2.2)	使用 **access_token** 值作为 OAuth 令牌
302	**Location** 标头包含 **error** 查询参数 (RFC 4.1.2.1)	失败，或向用户显示 **error**（使用可选的 **error_description**）查询值
302	其他 **Location** 标头	接续重定向操作，并使用这些规则处理结果

状态	内容	客户端响应
401	**WWW-Authenticate** 标头存在	在识别了类型时（如 **Basic** 和**Negotiate** 等）响应质询，重新提交请求，再使用这些规则处理结果
401	没有 **WWW-Authenticate** 标头	无法进行质询身份验证。失败，并显示响应正文（可能包含用于获取 OAuth 令牌的链接或备用方法详情）
其他	其他	失败，或可向用户显示响应正文

使用隐式授权流请求 OAuth 令牌：

```
$ curl -u <username>:<password>
'https://<master-address>:8443/oauth/authorize?client_id=openshift-challenging-
client&response_type=token' -skv / 1
/ -H "X-CSRF-Token: xxx" 2
```

输出示例

```
*   Trying 10.64.33.43...
* Connected to 10.64.33.43 (10.64.33.43) port 8443 (#0)
* found 148 certificates in /etc/ssl/certs/ca-certificates.crt
* found 592 certificates in /etc/ssl/certs
* ALPN, offering http/1.1
* SSL connection using TLS1.2 / ECDHE_RSA_AES_128_GCM_SHA256
*       server certificate verification SKIPPED
*       server certificate status verification SKIPPED
*       common name: 10.64.33.43 (matched)
*       server certificate expiration date OK
*       server certificate activation date OK
*       certificate public key: RSA
*       certificate version: #3
*       subject: CN=10.64.33.43
*       start date: Thu, 09 Aug 2018 04:00:39 GMT
*       expire date: Sat, 08 Aug 2020 04:00:40 GMT
*       issuer: CN=openshift-signer@1531109367
*       compression: NULL
* ALPN, server accepted to use http/1.1
* Server auth using Basic with user 'developer'
> GET /oauth/authorize?client_id=openshift-challenging-client&response_type=token HTTP/1.1
> Host: 10.64.33.43:8443
> Authorization: Basic ZGV2ZWxvcGVyOmRzc2Zkcw==
> User-Agent: curl/7.47.0
> Accept: */*
> X-CSRF-Token: xxx
>
< HTTP/1.1 302 Found
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Expires: Fri, 01 Jan 1990 00:00:00 GMT
< Location:
```

```
https://10.64.33.43:8443/oauth/token/implicit#access_token=gzTwOq_mVJ7ovHliHBTgRQEEXa1aCZD
9lnj7lSw3ekQ&expires_in=86400&scope=user%3Afull&token_type=Bearer (1)
< Pragma: no-cache
< Set-Cookie:
ssn=MTUzNTk0OTc1MnxIckVfNW5vNFlLSlF5MF9GWEF6Zm55Vl95bi1ZNE41S1NCbFJMYnN1TWV
wR1hwZmlLMzFQRklzVXRkc0RnUGEzdnBEa0NZZndXV2ZUVzN1dmFPM2dHSUlzUmVXakQ3Q09rV
XpxNlRoVmVkQU5DYmdLTE9SUWlyNkJJTm1mSDQ0N2pCV09La3gzMkMzckwxc1V1QXpybFlXT2ZY
SmI2R2FTVEZsdDBzRjJ8vk6zrQPjQUmoJCqb8Dt5j5s0b4wZlITgKlho9wlKAZI=; Path=/; HttpOnly;
Secure
< Date: Mon, 03 Sep 2018 04:42:32 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host 10.64.33.43 left intact
```

(1) **client-id** 设置为 **openshift-challenging-client**，**response-type** 设置为 **token**。

(2) 将 **X-CSRF-Token** 标头设置为非空值。

(1) 该令牌在 **302** 响应的 **Location** 标头中返回为 **access_token=gzTwOq_mVJ7ovHliHBTgRQEEXa1aCZD9lnj7lSw3ekQ**。

要只查看 OAuth 令牌值，请运行以下命令：

```
$ curl -u <username>:<password> /
'https://<master-address>:8443/oauth/authorize?client_id=openshift-challenging-
client&response_type=token' / (1)
-skv -H "X-CSRF-Token: xxx" --stderr - | grep -oP "access_token=\K[^&]*" (2)
```

(1) **client-id** 设置为 **openshift-challenging-client**，**response-type** 设置为 **token**。

(2) 将 **X-CSRF-Token** 标头设置为非空值。

输出示例

```
hvqxe5aMlAzvbqfM2WWw3D6tR0R2jCQGKx0viZBxwmc
```

您还可以使用 **Code Grant** 方法请求令牌。

4.1.4.7. Prometheus 身份验证指标

OpenShift Container Platform 在身份验证尝试过程中捕获以下 Prometheus 系统指标：

- **openshift_auth_basic_password_count** 统计 **oc login** 用户名和密码的尝试次数。
- **openshift_auth_basic_password_count_result** 按照结果（success 或 error）统计 **oc login** 用户名和密码尝试次数。
- **openshift_auth_form_password_count** 统计 web 控制台登录尝试次数。
- **openshift_auth_form_password_count_result** 统计 web 控制台登录尝试次数（包括成功和失败的登陆尝试）。

- **openshift_auth_password_total** 统计 **oc login** 和 web 控制台登录尝试总次数。

4.2. 授权

4.2.1. 概述

基于角色的访问控制(RBAC)对象确定是否允许用户在项目内执行给定的 操作。

这允许平台管理员使用 集群角色和绑定 来控制谁对 OpenShift Container Platform 平台本身和所有项目具有各种访问权限级别。

开发人员可以 利用本地角色和绑定来控制 谁有权访问 其项目。请注意，授权是独立于 身份验证 的一个步骤，身份验证更在于确定执行操作的人员的身份。

授权通过使用以下几项来管理：

Rules	组 对象 上允许 的操作 集合。例如，某容能否 **创建** pod。
Roles	规则的集合。用户和组 可以同时关联或*绑定*到多个角色。
Bindings	用户和/组与 角色 之间的关联。

集群管理员可以使用 CLI 视觉化规则、角色和绑定。

例如，请考虑以下摘录，其中显示了 **admin** 和 **basic-user** 默认集群角色 的规则集：

```
$ oc describe clusterrole.rbac admin basic-user
```

输出示例

```
Name:  admin
Labels:  <none>
Annotations: openshift.io/description=A user that has edit rights within the project and can change the
project's membership.
  rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources       Non-Resource URLs Resource Names Verbs
  ---------       ----------------- -------------- -----
  appliedclusterresourcequotas     []   []  [get list watch]
  appliedclusterresourcequotas.quota.openshift.io  []   []  [get list watch]
  bindings       []   []  [get list watch]
  buildconfigs       []   []  [create delete deletecollection get list patch update watch]
  buildconfigs.build.openshift.io    []   []  [create delete deletecollection get list patch update watch]
  buildconfigs/instantiate     []   []  [create]
  buildconfigs.build.openshift.io/instantiate   []   []  [create]
  buildconfigs/instantiatebinary     []   []  [create]
  buildconfigs.build.openshift.io/instantiatebinary  []   []  [create]
  buildconfigs/webhooks      []   []  [create delete deletecollection get list patch update watch]
  buildconfigs.build.openshift.io/webhooks   []   []  [create delete deletecollection get list patch update
watch]
  buildlogs       []   []  [create delete deletecollection get list patch update watch]
  buildlogs.build.openshift.io     []   []  [create delete deletecollection get list patch update watch]
```

```
  builds       []   []  [create delete deletecollection get list patch update watch]
  builds.build.openshift.io     []   []  [create delete deletecollection get list patch update watch]
  builds/clone       []   []  [create]
  builds.build.openshift.io/clone     []   []  [create]
  builds/details      []   []  [update]
  builds.build.openshift.io/details     []   []  [update]
  builds/log       []   []  [get list watch]
  builds.build.openshift.io/log     []   []  [get list watch]
  configmaps       []   []  [create delete deletecollection get list patch update watch]
  cronjobs.batch      []   []  [create delete deletecollection get list patch update watch]
  daemonsets.extensions      []   []  [get list watch]
  deploymentconfigrollbacks     []   []  [create]
  deploymentconfigrollbacks.apps.openshift.io   []   []  [create]
  deploymentconfigs      []   []  [create delete deletecollection get list patch update watch]
  deploymentconfigs.apps.openshift.io     []   []  [create delete deletecollection get list patch update
watch]
  deploymentconfigs/instantiate     []   []  [create]
  deploymentconfigs.apps.openshift.io/instantiate  []   []  [create]
  deploymentconfigs/log      []   []  [get list watch]
  deploymentconfigs.apps.openshift.io/log   []   []  [get list watch]
  deploymentconfigs/rollback     []   []  [create]
  deploymentconfigs.apps.openshift.io/rollback   []   []  [create]
  deploymentconfigs/scale     []   []  [create delete deletecollection get list patch update watch]
  deploymentconfigs.apps.openshift.io/scale   []   []  [create delete deletecollection get list patch
update watch]
  deploymentconfigs/status     []   []  [get list watch]
  deploymentconfigs.apps.openshift.io/status   []   []  [get list watch]
  deployments.apps      []   []  [create delete deletecollection get list patch update watch]
  deployments.extensions     []   []  [create delete deletecollection get list patch update watch]
  deployments.extensions/rollback    []   []  [create delete deletecollection get list patch update watch]
  deployments.apps/scale     []   []  [create delete deletecollection get list patch update watch]
  deployments.extensions/scale     []   []  [create delete deletecollection get list patch update watch]
  deployments.apps/status     []   []  [create delete deletecollection get list patch update watch]
  endpoints       []   []  [create delete deletecollection get list patch update watch]
  events       []   []  [get list watch]
  horizontalpodautoscalers.autoscaling    []   []  [create delete deletecollection get list patch update
watch]
  horizontalpodautoscalers.extensions    []   []  [create delete deletecollection get list patch update
watch]
  imagestreamimages      []   []  [create delete deletecollection get list patch update watch]
  imagestreamimages.image.openshift.io    []   []  [create delete deletecollection get list patch update
watch]
  imagestreamimports      []   []  [create]
  imagestreamimports.image.openshift.io    []   []  [create]
  imagestreammappings      []   []  [create delete deletecollection get list patch update watch]
  imagestreammappings.image.openshift.io   []   []  [create delete deletecollection get list patch update
watch]
  imagestreams       []   []  [create delete deletecollection get list patch update watch]
  imagestreams.image.openshift.io    []   []  [create delete deletecollection get list patch update watch]
  imagestreams/layers      []   []  [get update]
  imagestreams.image.openshift.io/layers   []   []  [get update]
  imagestreams/secrets      []   []  [create delete deletecollection get list patch update watch]
  imagestreams.image.openshift.io/secrets   []   []  [create delete deletecollection get list patch update
watch]
  imagestreams/status      []   []  [get list watch]
  imagestreams.image.openshift.io/status   []   []  [get list watch]
```

```
  imagestreamtags      []   []  [create delete deletecollection get list patch update watch]
  imagestreamtags.image.openshift.io    []   []  [create delete deletecollection get list patch update
watch]
  jenkins.build.openshift.io     []   []  [admin edit view]
  jobs.batch       []   []  [create delete deletecollection get list patch update watch]
  limitranges      []   []  [get list watch]
  localresourceaccessreviews     []   []  [create]
  localresourceaccessreviews.authorization.openshift.io  []   []  [create]
  localsubjectaccessreviews      []   []  [create]
  localsubjectaccessreviews.authorization.k8s.io  []   []  [create]
  localsubjectaccessreviews.authorization.openshift.io  []   []  [create]
  namespaces       []   []  [get list watch]
  namespaces/status      []   []  [get list watch]
  networkpolicies.extensions     []   []  [create delete deletecollection get list patch update watch]
  persistentvolumeclaims     []   []  [create delete deletecollection get list patch update watch]
  pods       []   []  [create delete deletecollection get list patch update watch]
  pods/attach      []   []  [create delete deletecollection get list patch update watch]
  pods/exec      []   []  [create delete deletecollection get list patch update watch]
  pods/log       []   []  [get list watch]
  pods/portforward      []   []  [create delete deletecollection get list patch update watch]
  pods/proxy       []   []  [create delete deletecollection get list patch update watch]
  pods/status      []   []  [get list watch]
  podsecuritypolicyreviews     []   []  [create]
  podsecuritypolicyreviews.security.openshift.io  []   []  [create]
  podsecuritypolicyselfsubjectreviews    []   []  [create]
  podsecuritypolicyselfsubjectreviews.security.openshift.io []   []  [create]
  podsecuritypolicysubjectreviews    []   []  [create]
  podsecuritypolicysubjectreviews.security.openshift.io  []   []  [create]
  processedtemplates      []   []  [create delete deletecollection get list patch update watch]
  processedtemplates.template.openshift.io   []   []  [create delete deletecollection get list patch update
watch]
  projects       []   []  [delete get patch update]
  projects.project.openshift.io     []   []  [delete get patch update]
  replicasets.extensions     []   []  [create delete deletecollection get list patch update watch]
  replicasets.extensions/scale     []   []  [create delete deletecollection get list patch update watch]
  replicationcontrollers     []   []  [create delete deletecollection get list patch update watch]
  replicationcontrollers/scale     []   []  [create delete deletecollection get list patch update watch]
  replicationcontrollers.extensions/scale   []   []  [create delete deletecollection get list patch update
watch]
  replicationcontrollers/status     []   []  [get list watch]
  resourceaccessreviews      []   []  [create]
  resourceaccessreviews.authorization.openshift.io  []   []  [create]
  resourcequotas      []   []  [get list watch]
  resourcequotas/status      []   []  [get list watch]
  resourcequotausages      []   []  [get list watch]
  rolebindingrestrictions     []   []  [get list watch]
  rolebindingrestrictions.authorization.openshift.io  []   []  [get list watch]
  rolebindings       []   []  [create delete deletecollection get list patch update watch]
  rolebindings.authorization.openshift.io   []   []  [create delete deletecollection get list patch update
watch]
  rolebindings.rbac.authorization.k8s.io   []   []  [create delete deletecollection get list patch update
watch]
  roles       []   []  [create delete deletecollection get list patch update watch]
  roles.authorization.openshift.io    []   []  [create delete deletecollection get list patch update watch]
  roles.rbac.authorization.k8s.io    []   []  [create delete deletecollection get list patch update watch]
  routes       []   []  [create delete deletecollection get list patch update watch]
```

```
  routes.route.openshift.io     []   []  [create delete deletecollection get list patch update watch]
  routes/custom-host      []   []  [create]
  routes.route.openshift.io/custom-host    []   []  [create]
  routes/status       []   []  [get list watch update]
  routes.route.openshift.io/status    []   []  [get list watch update]
  scheduledjobs.batch      []   []  [create delete deletecollection get list patch update watch]
  secrets       []   []  [create delete deletecollection get list patch update watch]
  serviceaccounts      []   []  [create delete deletecollection get list patch update watch impersonate]
  services       []   []  [create delete deletecollection get list patch update watch]
  services/proxy      []   []  [create delete deletecollection get list patch update watch]
  statefulsets.apps      []   []  [create delete deletecollection get list patch update watch]
  subjectaccessreviews      []   []  [create]
  subjectaccessreviews.authorization.openshift.io  []   []  [create]
  subjectrulesreviews      []   []  [create]
  subjectrulesreviews.authorization.openshift.io  []   []  [create]
  templateconfigs      []   []  [create delete deletecollection get list patch update watch]
  templateconfigs.template.openshift.io    []   []  [create delete deletecollection get list patch update
watch]
  templateinstances      []   []  [create delete deletecollection get list patch update watch]
  templateinstances.template.openshift.io   []   []  [create delete deletecollection get list patch update
watch]
  templates       []   []  [create delete deletecollection get list patch update watch]
  templates.template.openshift.io    []   []  [create delete deletecollection get list patch update watch]


Name:  basic-user
Labels:  <none>
Annotations: openshift.io/description=A user that can get basic information about projects.
  rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources      Non-Resource URLs Resource Names Verbs
  ---------      ----------------- -------------- -----
  clusterroles      []   []  [get list]
  clusterroles.authorization.openshift.io  []   []  [get list]
  clusterroles.rbac.authorization.k8s.io  []   []  [get list watch]
  projectrequests     []   []  [list]
  projectrequests.project.openshift.io   []   []  [list]
  projects      []   []  [list watch]
  projects.project.openshift.io    []   []  [list watch]
  selfsubjectaccessreviews.authorization.k8s.io  []   []  [create]
  selfsubjectrulesreviews    []   []  [create]
  selfsubjectrulesreviews.authorization.openshift.io []   []  [create]
  storageclasses.storage.k8s.io    []   []  [get list]
  users       []   [~]  [get]
  users.user.openshift.io    []   [~]  [get]
```

以下摘录查看本地角色绑定，显示了绑定到不同用户和组的以上角色：

```
$ oc describe rolebinding.rbac admin basic-user -n alice-project
```

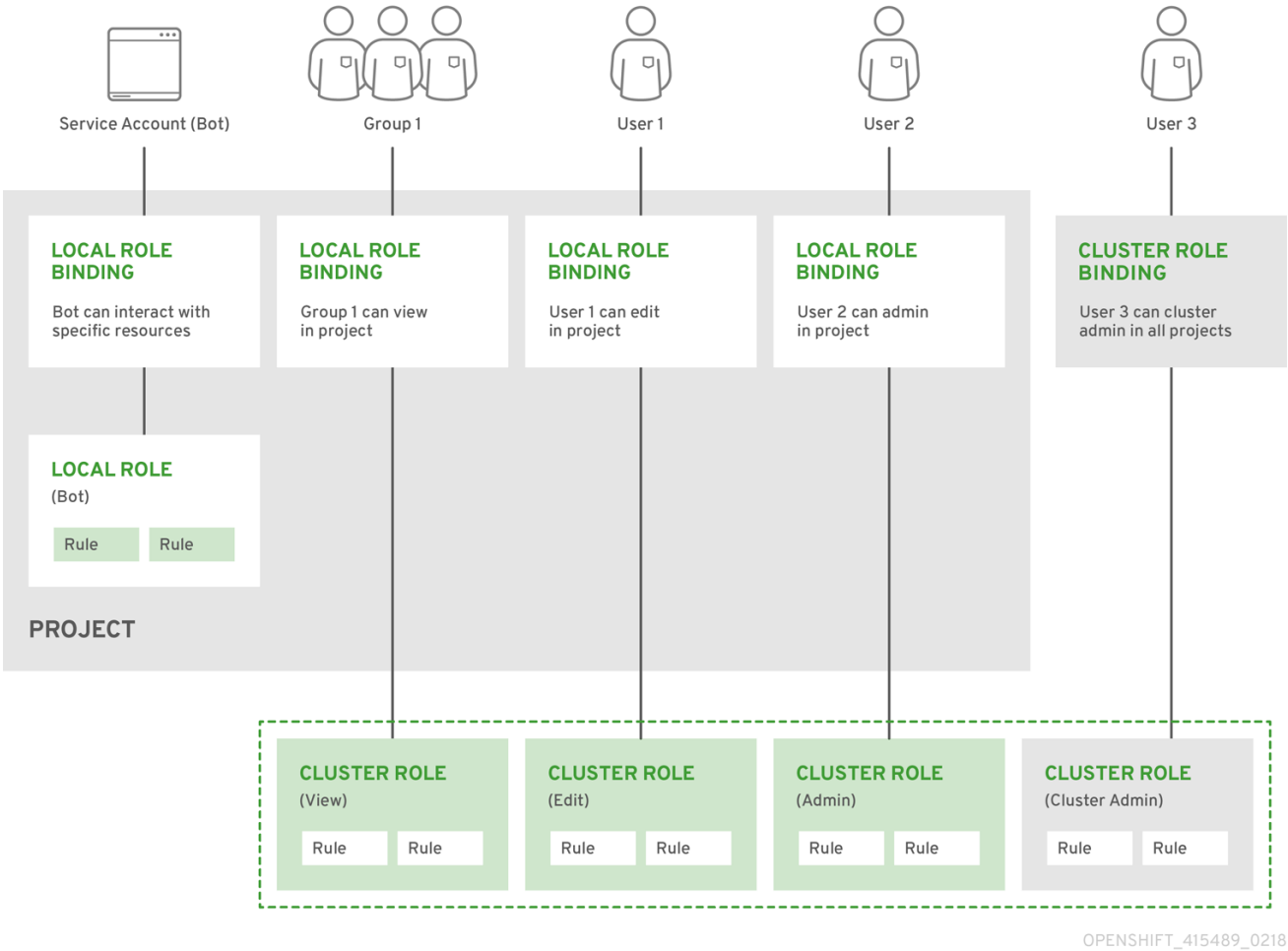
输出示例

```
Name:  admin
Labels:  <none>
Annotations: <none>
```

```
Role:
  Kind: ClusterRole
  Name: admin
Subjects:
  Kind Name  Namespace
  ---- ----  ---------
  User system:admin
  User alice


Name:  basic-user
Labels:  <none>
Annotations: <none>
Role:
  Kind: ClusterRole
  Name: basic-user
Subjects:
  Kind Name Namespace
  ---- ---- ---------
  User joe
  Group devel
```

下方展示了集群角色、本地角色、集群角色绑定、本地角色绑定、用户、组和服务帐户之间的关系。



4.2.2. 评估授权

OpenShift Container Platform 在评估授权时结合了几个因素：

Identity	在授权上下文中，用户名和用户所属组的列表。
操作	正在执行的操作。在大多数情况下，这由以下几项组成： project：正在访问 的项目。 Verb：可以是 **get**、list、**create**、**update**、**patch**、delete、**delete**、**deletecollection** 或 **watch**。 资源名称：正在访问的 API 端点。
绑定	绑定 的完整列表。

OpenShift Container Platform 通过以下步骤评估授权：

1. 使用身份和项目范围操作来查找应用到用户或所属组的所有绑定。
2. 使用绑定来查找应用的所有角色。
3. 使用角色来查找应用的所有规则。
4. 针对每一规则检查操作，以查找匹配项。
5. 如果未找到匹配的规则，则默认拒绝该操作。

4.2.3. 集群和本地 RBAC

控制授权的 RBAC 角色和绑定有两个级别：

集群 RBAC	适用于所有项目 的角色和 绑定。集群范围内存在的角色被视为 *集群角色*。集群角色绑定只能引用集群角色。
本地 RBAC	作用于给定项目 的角色和 绑定。仅存在于项目中的角色被视为 *本地角色*。本地角色绑定可以引用集群和本地角色。

这种双级层次结构允许通过集群角色重新利用多个项目，同时允许通过本地角色在单个项目中自定义。

在评估过程中，同时使用集群角色绑定和本地角色绑定。例如：

1. 选中集群范围的“allow”规则。
2. 选中本地绑定的“allow”规则。
3. 默认为拒绝。

4.2.4. 集群角色和本地角色

角色是策略 规则 的集合，这是一组可以对一组资源执行的允许动词。OpenShift Container Platform 包含一组默认的集群角色，可以绑定到集群范围 或 本地的 用户和组。

默认集群角色	描述
admin	项目管理者。如果 在本地绑定 中使用，admin 用户将有权查看项目中的任何资源，并且修改项目中除配额外的任何资源。
basic-user	此用户可以获取有关项目和用户的基本信息。
cluster-admin	此超级用户可以在任意项目中执行任何操作。与具有本地绑定的用户绑定时，他们可以完全控制项目中每个资源的配额和每个操作。
cluster-status	此用户可以获取基本的集群状态信息。
edit	此用户可以修改项目中的大多数对象，但没有权限查看或修改角色或绑定。
self-provisioner	此用户可以创建自己的项目。
view	此用户无法进行任何修改，但可以查看项目中的大多数对象。不能查看或修改角色或绑定。
cluster-reader	此用户可以读取（但不能查看）集群中的对象。

提示

请记住，用户和组可以同时关联或*绑定到*多个角色。

项目管理员可以视觉化角色，包括使用 CLI 关联的操作动词和资源列表 来查看本地角色和绑定。

重要

通过 本地绑定 绑定到项目管理员的集群角色在项目中受到限制。它不会像授权给 cluster-admin 或 system : **admin 的 集群角色那样在集群范围** 绑定。

集群角色是在集群级别上定义的角色，但可以在集群级别或项目级别绑定。

了解如何为项目创建本地角色。

4.2.4.1. 更新集群角色

任何 OpenShift Container Platform 集群升级后，会在服务器启动时更新默认角色并自动协调。在协调过程中，会添加默认角色中缺少的任何权限。如果您向角色添加了更多权限，则不会删除它们。

如果您自定义默认角色并配置了这些角色以防止自动角色协调，则必须在升级 OpenShift Container Platform 时 手动更新策略定义。

4.2.4.2. 应用自定义角色和权限

要更新自定义角色和权限，强烈建议您使用以下命令：

```
$ oc auth reconcile -f <file> 1
```

1 **<file> 是要应用 的角色和权限的 绝对路径。**

您还可以使用此命令来添加新的自定义角色和权限。如果新自定义角色的名称与现有角色的名称相同，则会更新现有的角色。集群管理员不会收到名称相同的自定义角色通知。

此命令可确保以不会破坏其他客户端的方式正确应用新权限。这通过计算逻辑在规则集之间的操作在内部完成，这是您无法通过 RBAC 资源上的 JSON 合并执行的操作。

4.2.4.3. 集群角色聚合

默认的 **admin**、**edit**、**view** 和 **cluster-reader** 集群角色支持集群角色聚合，其中每个角色的集群规则会在创建新规则时动态更新。只有在您通过创建自定义资源 扩展 Kubernetes API 时，此功能才有意义。

了解如何使用集群角色聚合。

4.2.5. 安全性上下文约束

除了控制用户可执行的操作的 RBAC 资源 外，OpenShift Container Platform 还提供 *安全性上下文约束* (SCC)来控制 Pod 可以执行的操作以及它有权访问的内容。管理员可以使用 CLI 管理 SCC。

SCC 在管理对持久存储的访问时也非常有用。

SCC 是定义 Pod 运行必须满足的一组条件的对象，以便其能被系统接受。它们允许管理员控制以下内容：

1. 运行特权容器。
2. 容器可请求添加的功能
3. 将主机目录用作卷。
4. 容器的 SELinux 上下文。
5. 用户 ID。
6. 使用主机命名空间和联网。
7. 分配拥有 pod 卷的 **FSGroup**
8. 配置允许的补充组
9. 要求使用只读根文件系统
10. 控制卷类型的使用
11. 配置允许的 seccomp 配置集

默认情况下，在集群中添加七个 SCC，集群管理员可使用 CLI 查看它们：

```
$ oc get scc
```

输出示例

```
NAME              PRIV    CAPS    SELINUX    RUNASUSER          FSGROUP    SUPGROUP
PRIORITY   READONLYROOTFS   VOLUMES
```

```
anyuid          false   []      MustRunAs   RunAsAny        RunAsAny    RunAsAny    10      false
[configMap downwardAPI emptyDir persistentVolumeClaim secret]
hostaccess      false   []      MustRunAs   MustRunAsRange  MustRunAs   RunAsAny    <none>
false           [configMap downwardAPI emptyDir hostPath persistentVolumeClaim secret]
hostmount-anyuid false  []      MustRunAs   RunAsAny        RunAsAny    RunAsAny    <none>
false           [configMap downwardAPI emptyDir hostPath nfs persistentVolumeClaim secret]
hostnetwork     false   []      MustRunAs   MustRunAsRange  MustRunAs   MustRunAs   <none>
false           [configMap downwardAPI emptyDir persistentVolumeClaim secret]
nonroot         false   []      MustRunAs   MustRunAsNonRoot RunAsAny   RunAsAny    <none>
false           [configMap downwardAPI emptyDir persistentVolumeClaim secret]
privileged      true    [*]     RunAsAny    RunAsAny        RunAsAny    RunAsAny    <none>
false           [*]
restricted      false   []      MustRunAs   MustRunAsRange  MustRunAs   RunAsAny    <none>
false           [configMap downwardAPI emptyDir persistentVolumeClaim secret]
```

重要

不要修改默认 SCC。修改默认 SCC 可导致升级 OpenShift Container Platform 时出现问题。而是 创建新的 SCC。

集群管理员也可以使用 CLI 查看各个 SCC 的定义。例如，对于特权 SCC：

```
$ oc get -o yaml --export scc/privileged
```

输出示例

```
allowHostDirVolumePlugin: true
allowHostIPC: true
allowHostNetwork: true
allowHostPID: true
allowHostPorts: true
allowPrivilegedContainer: true
allowedCapabilities: 1
- '*'
apiVersion: v1
defaultAddCapabilities: [] 2
fsGroup: 3
  type: RunAsAny
groups: 4
- system:cluster-admins
- system:nodes
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: 'privileged allows access to all privileged and host
      features and the ability to run as any user, any group, any fsGroup, and with
      any SELinux context.  WARNING: this is the most relaxed SCC and should be used
      only for cluster administration. Grant with caution.'
  creationTimestamp: null
  name: privileged
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities: [] 5
```

```
runAsUser: 6
  type: RunAsAny
seLinuxContext: 7
  type: RunAsAny
seccompProfiles:
- '*'
supplementalGroups: 8
  type: RunAsAny
users: 9
- system:serviceaccount:default:registry
- system:serviceaccount:default:router
- system:serviceaccount:openshift-infra:build-controller
volumes:
- '*'
```

1. Pod 可以请求的功能列表。空列表表示不允许请求任何功能，而特殊符号 * 则允许任何功能。
2. 将添加到任何 pod 的附加功能列表。
3. **FSGroup** 策略，指明安全性上下文的允许值。
4. 有权访问此 SCC 的组。
5. 将从 pod 中丢弃的功能列表。
6. 作为用户策略类型运行，指明安全性上下文的允许值。
7. SELinux 上下文策略类型，指明安全性上下文的允许值。
8. 补充组策略，指明安全上下文的允许补充组。
9. 有权访问此 SCC 的用户。

SCC 上的 **users** 和 **groups** 字段控制可以使用哪些 SCC。默认情况下，集群管理员、节点和构建控制器被授予特权 SCC 的访问权限。所有经过身份验证的用户被授予受限 SCC 的访问权限。

Docker 具有允许用于 Pod 的每个容器的默认功能列表。容器使用此默认列表中的功能，但 Pod 清单作者可以通过请求额外的功能或丢弃一些默认功能来修改它。**allowedCapabilities**、**defaultAddCapabilities** 和 **requiredDropCapabilities** 字段用于控制来自 Pod 的此类请求，并指定可以请求哪些功能、必须为每个容器添加哪些功能，以及必须禁止哪些功能。

特权 SCC：

- 允许特权 pod。
- 允许将主机目录挂载为卷.
- 允许容器集以任何用户身份运行。
- 允许 Pod 使用任何 MCS 标签运行。
- 允许 pod 使用主机的 IPC 命名空间。
- 允许容器集使用主机的 PID 命名空间。
- 允许容器集使用任何 FSGroup。

- 允许容器集使用任何补充组。
- 允许容器集使用任何 seccomp 配置集。
- 允许容器集请求任何功能。

受限 SCC：

- 确保 Pod 无法以特权方式运行。
- 确保容器集无法使用主机目录卷。
- 要求容器集以预先分配的 UID 范围内的用户运行。
- 要求 Pod 使用预先分配的 MCS 标签运行。
- 允许容器集使用任何 FSGroup。
- 允许容器集使用任何补充组。

注意

如需有关各个 SCC 的更多信息，请参阅 SCC 的 **kubernetes.io/description** 注解。

SCC 由设置和策略组成，它们控制容器集可以访问的安全功能。这些设置分为三个类别：

由布尔值控制	此类型的字段默认为限制性最强的值。例如，**AllowPrivilegedContainer** 若未指定，则始终设为 **false**。
由允许的集合控制	针对集合检查此类型的字段，以确保其值被允许。
由策略控制	具有生成某个值的策略的条目提供以下功能： ● 生成值的机制，以及 ● 确保指定值属于允许值集合的机制。

4.2.5.1. SCC 策略

4.2.5.1.1. RunAsUser

1. **MustRunAs** - 需要配置 **runAsUser**。使用配置的 **runAsUser** 作为默认值。针对配置的 **runAsUser** 进行验证。
2. **MustRunAsRange** - 如果不使用预分配值，则需要定义最小值和最大值。使用最小值作为默认值。针对整个允许范围进行验证。
3. **MustRunAsNonRoot** - 需要 Pod 提交为具有非零 **runAsUser** 或具有镜像中定义的 **USER** 指令。不提供默认值。
4. **RunAsAny** - 不提供默认值。允许指定任何 **runAsUser**。

4.2.5.1.2. SELinuxContext

1. **MustRunAs** - 如果不使用预分配的值，则需要配置 **seLinuxOptions**。使用 **seLinuxOptions** 作为默认值。针对使用 **seLinuxOptions** 进行验证。

2. **RunAsAny** - 不提供默认值。允许指定任何 **seLinuxOptions**。

4.2.5.1.3. SupplementalGroups

1. **MustRunAs** - 如果不使用预分配值，则需要至少指定一个范围。使用第一个范围内的最小值作为默认值。针对所有范围进行验证。

2. **RunAsAny** - 不提供默认值。允许指定任何 **supplementalGroups**。

4.2.5.1.4. FSGroup

1. **MustRunAs** - 如果不使用预分配值，则需要至少指定一个范围。使用第一个范围内的最小值作为默认值。针对第一个范围内的第一个 ID 进行验证。

2. **RunAsAny** - 不提供默认值。允许指定任何 **fsGroup** ID。

4.2.5.2. 控制卷

通过设置 SCC 的 **volumes** 字段，控制特定卷类型的使用。此字段的允许值与创建卷时定义的卷来源对应：

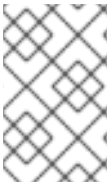
- azureFile
- azureDisk
- flocker
- flexVolume
- hostPath
- emptyDir
- gcePersistentDisk
- awsElasticBlockStore
- secret
- nfs
- iscsi
- glusterfs
- persistentVolumeClaim
- rbd
- cinder
- cephFS
- downwardAPI

- fc
- configMap
- vsphereVolume
- quobyte
- photonPersistentDisk
- projected
- portworxVolume
- scaleIO
- storageos
- *（允许使用所有卷类型的一个特殊值）
- none（禁止使用所有卷类型的一个特殊值。仅为向后兼容而存在）

为新 SCC 推荐的允许卷最小集合是
configMap、downAPI、emptyDir、persistentVolumeClaim、secret 和 projected。

注意

允许卷类型列表并不完整，因为每次发布新版 OpenShift Container Platform 时都会添加新的类型。



注意

为向后兼容，使用 **allowHostDirVolumePlugin** 将覆盖 **volumes** 字段中的设置。例如，如果 **allowHostDirVolumePlugin** 设为 false，但在 **volumes** 字段中是允许，则将移除 **volumes** 中的 **hostPath** 值。

4.2.5.3. 限制对 FlexVolumes 的访问

OpenShift Container Platform 基于其驱动程序提供 FlexVolumes 的额外控制。当 SCC 允许 FlexVolumes 时，pod 可以请求任何 FlexVolumes。但是，当集群管理员在 **AllowedFlexVolumes** 字段中指定驱动程序名称时，pod 只能使用这些驱动程序的 FlexVolume。

只允许两个 FlexVolume 的限制示例

```
volumes:
- flexVolume
allowedFlexVolumes:
- driver: example/lvm
- driver: example/cifs
```

4.2.5.4. seccomp

SeccompProfiles 列出了可为 pod 或容器的 seccomp 注解设置的允许的配置集。unset(nil)或空值表示 pod 或容器没有指定配置集。使用通配符 * 允许所有配置集。当用于生成 pod 的值时，第一个非通配符配置集被用作默认值。

有关配置和使用自定义配置集的更多信息，请参阅 seccomp 文档。

4.2.5.5. 准入

利用 SCC 的*准入控制*可以根据授予用户的能力来控制资源的创建。

就 SCC 而言，这意味着准入控制器可以检查上下文中提供的用户信息以检索一组合适的 SCC。这样做可确保 Pod 具有相应的授权，能够提出与其操作环境相关的请求或生成一组要应用到 Pod 的约束。

准入用于授权 Pod 的 SCC 集合由用户身份和用户所属的组来决定。另外，如果 Pod 指定了服务帐户，则允许的 SCC 集合包括服务帐户可访问的所有约束。

准入使用以下方法来创建 Pod 的最终安全性上下文：

1. 检索所有可用的 SCC。
2. 为请求上未指定的安全性上下文设置生成字段值。
3. 针对可用约束来验证最终设置。

如果找到了匹配的约束集合，则接受 Pod。如果请求不能与 SCC 匹配，则拒绝 Pod。

Pod 必须针对 SCC 验证每一个字段。以下示例中只有其中两个字段必须验证：

注意

这些示例是在使用预分配值的策略的上下文中。

FSGroup SCC 策略为 MustRunAs

如果 Pod 定义了 **fsGroup** ID，该 ID 必须等于默认的 **fsGroup** ID。否则，Pod 不会由该 SCC 验证，而会评估下一个 SCC。

如果 **SecurityContextConstraints.fsGroup** 字段的值为 **RunAsAny**，并且 Pod 规格省略了 **Pod.spec.securityContext.fsGroup**，则此字段被视为有效。注意在验证过程中，其他 SCC 设置可能会拒绝其他 Pod 字段，从而导致 Pod 失败。

SupplementalGroups SCC 策略为 MustRunAs

如果 Pod 规格定义了一个或多个 **supplementalGroups** ID，则 Pod 的 ID 必须等于命名空间的 **openshift.io/sa.scc.supplemental-groups** 注解中的某一个 ID。否则，Pod 不会由该 SCC 验证，而会评估下一个 SCC。

如果 **SecurityContextConstraints.supplementalGroups** 字段的值为 **RunAsAny**，并且 Pod 规格省略了 **Pod.spec.securityContext.supplementalGroups**，则此字段被视为有效。注意在验证过程中，其他 SCC 设置可能会拒绝其他 Pod 字段，从而导致 Pod 失败。

4.2.5.5.1. SCC 优先级

SCC 有一个优先级字段，它会影响准入控制器尝试验证请求时的排序。在排序时，高优先级 SCC 移到集合的前面。确定了可用 SCC 的完整集合后，按照以下方式排序：

1. 优先级最高的在前，nil 视为 0 优先级
2. 如果优先级相等，则 SCC 按照限制性最强到最弱排序

3. 如果优先级和限制性都相等，则 SCC 按照名称排序

默认情况下，授予集群管理员的 anyuid SCC 在 SCC 集合中具有优先权。这使得集群管理员能够以任意用户运行 Pod，而不必在 Pod 的 **SecurityContext** 中指定 **RunAsUser**。若有需要，管理员仍然可以指定 **RunAsUser**。

4.2.5.5.2. 对 SCC 的基于角色的访问控制

从 OpenShift Container Platform 3.11 开始，您可以将 SCC 指定为由 RBAC 处理的资源。这样，您可以将对 SCC 访问的范围限定为某一项目或整个集群。直接将用户、组或服务帐户分配给 SCC 可保留集群范围的范围。

要为角色包含对 SCC 的访问，您可以在 role 的定义中指定以下规则： .Role-Based Access to SCC

```
rules:
- apiGroups:
  - security.openshift.io 1
  resources:
  - securitycontextconstraints 2
  verbs:
  - create
  - delete
  - deletecollection
  - get
  - list
  - patch
  - update
  - watch
  resourceNames:
  - myPermittingSCC 3
```

1. 包含 **Securitycontextconstraints** 资源的 API 组
2. 允许用户在 **resourceNames** 字段中指定 SCC 名称的资源组名称
3. 您要授予访问权限的 SCC 的示例名称

具有这样的规则的本地或集群角色允许以 rolebinding 或 clusterrolebinding 与其绑定的主体使用用户定义的 SCC，名为 **myPermittingSCC**。

注意

由于 RBAC 旨在防止升级，因此即使项目管理员也无法授予 SCC 访问权限，因为它们默认不被允许，**使用** 动词对 SCC 资源（包括 **restricted** SCC）。

4.2.5.5.3. 了解预分配值和安全性上下文约束

准入控制器清楚安全性上下文约束中的某些条件，这些条件会触发它从命名空间中查找预分配值并在处理 pod 前填充安全性上下文约束。每个 SCC 策略都独立于其他策略进行评估，每个策略的预分配值（允许）与 Pod 规格值聚合，为运行的 Pod 中定义的不同 ID 生成最终值。

以下 SCC 导致准入控制器在 Pod 规格中没有定义范围时查找预分配值：

1. **RunAsUser** 策略为 MustRunAsRange 且未设置最小或最大值。准入查找 **openshift.io/sa.scc.uid-range** 注解来填充范围字段。

2. **SELinuxContext** 策略为 MustRunAs 且未设定级别。准入查找 **openshift.io/sa.scc.mcs** 注解来填充级别。

3. **FSGroup** 策略为 MustRunAs。准入查找 **openshift.io/sa.scc.supplemental-groups** 注解。

4. **SupplementalGroups** 策略为 MustRunAs。准入查找 **openshift.io/sa.scc.supplemental-groups** 注解。

在生成阶段，安全性上下文提供程序将默认使用 pod 中未具体设置的任何值。默认基于所使用的策略：

1. **RunAsAny** 和 **MustRunAsNonRoot** 策略不提供默认值。因此，如果 pod 需要定义有一个字段（如组 ID），必须在 pod 规格中定义此字段。

2. **MustRunAs** （单值）策略提供始终使用的默认值。例如，对于组 ID：即使 pod 规格定义了自己的 ID 值，命名空间的默认字段也会出现在 pod 的组中。

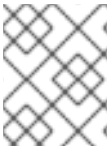
3. **MustRunAsRange** 和 **MustRunAs** （基于范围）策略提供范围的最小值。与单值 **MustRunAs** 策略一样，命名空间的默认值将出现在运行的 Pod 中。如果基于范围的策略可以配置多个范围，它会提供第一个配置范围内的最小值。

注意

如果命名空间上不存在 **openshift.io/sa.scc.supplemental-groups** 注解，则 **FSGroup** 和 **SupplementalGroups** 策略回退到 **openshift.io/sa.scc.uid-range** 注解。如果两者都不存在，则 SCC 将无法创建。

注意

默认情况下，基于注解的 **FSGroup** 策略使用基于注解的最小值的单个范围来配置其自身。例如，如果您的注解显示为 **1/3**，则 **FSGroup** 策略会将其自身配置为最小和最大 **1**。如果要允许 **FSGroup** 字段接受多个组，可以配置不使用注解的自定义 SCC。



注意

openshift.io/sa.scc.supplemental-groups 注解接受以逗号分隔的块列表，格式为 **<start>/<length 或 < start>-<end>**。**openshift.io/sa.scc.uid-range** 注解只接受一个块。

4.2.6. 确定您可以作为经过身份验证的用户执行什么操作

在 OpenShift Container Platform 项目中，您可以确定您可以对所有命名空间范围的资源（包括第三方资源）执行的操作动词。运行：

```
$ oc policy can-i --list --loglevel=8
```

输出将帮助您确定收集信息的 API 请求。

要以用户可读格式接收信息，请运行：

```
$ oc policy can-i --list
```

输出中会提供一个完整的列表。

要确定您可以执行特定的动词， 请运行：

```
$ oc policy can-i <verb> <resource>
```

用户范围 可以提供有关给定范围的更多信息。例如：

```
$ oc policy can-i <verb> <resource> --scopes=user:info
```

4.3. 持久性存储

4.3.1. 概述

管理存储与管理计算资源不同。OpenShift Container Platform 使用 Kubernetes 持久性卷 (PV) 框架来允许集群管理员为集群提供持久性存储。开发者可以使用持久性卷声明 (PVC) 来请求 PV 资源而无需具体了解底层存储基础架构。

PVC 特定于 某个项目，开发人员会创建和使用 PV。PV 资源本身并不特定于某一个项目; 它们可以在整个 OpenShift Container Platform 集群间共享，并可以被任何项目使用。当 PV 绑定到 PVC 后，但该 PV 不能绑定到额外的 PVC。这会使绑定的 PV 限定为 一个命名空间 （绑定项目是什么）。

PV 由一个 **persistentVolume** API 对象定义，它代表了集群管理员置备的已存在的网络存储。它与一个节点一样，是一个集群资源。PV 是卷插件，与 **Volumes** 资源类似，但PV 的生命周期独立于任何使用它的 pod。PV 对象获取具体存储（NFS、iSCSI 或者特定 cloud-provider 的存储系统）的实现详情。

重要

存储的高可用性功能由底层的存储架构提供。

PVC 由 **PersistentVolumeClaim** API 项定义，它代表了开发人员对存储的一个请求。它与一个 pod 类似，pod 会消耗节点资源， PVC 消耗 PV 资源。例如，pod 可以请求特定级别的资源（如 CPU 和内存），而 PVC 可以请求特定 的存储容量 和访问模式 （例如，可以在读/写或多次只读时挂载它们）。

4.3.2. 卷和声明的生命周期

PV 是集群中的资源。PVC 是对这些资源的请求，也是对该资源的声明检查。PV 和 PVC 之间的交互有以下生命周期。

4.3.2.1. 置备存储

根据 PVC 中定义的开发人员的请求，集群管理员配置一个或者多个动态置备程序用来置备存储及一个匹配的 PV。

另外，集群管理员也可以预先创建多个 PV，它们包含了可用存储的详情。PV 存在于 API 中，且可以被使用。

4.3.2.2. 绑定声明

当您创建 PVC 时，您会要求特定的存储量，指定所需的访问模式，并创建一个存储类来描述和分类存储。master 中的控制循环会随时检查是否有新的 PVC，并把新的 PVC 与一个适当的 PV 进行绑定。如果没有适当的 PV，则存储类的置备程序会生成一个适当的 PV。

PV 卷可能会超过您请求的卷。这在手动置备 PV 时尤为如此。要最小化超额，OpenShift Container Platform 将会把 PVC 绑定到匹配所有其他标准的最小 PV。

如果匹配的卷不存在，或者无法使用任何可用的置备程序提供存储类来创建，则声明将无限期保留。当出现了匹配的卷时，相应的声明就会与其绑定。例如：在一个集群中有多个手动置备的 50Gi 卷。它们无法和一个请求 100Gi 的 PVC 相匹配。当在这个集群中添加了一个 100Gi PV 时， PVC 就可以和这个 PV 绑定。

4.3.2.3. 使用 pod 和声明的 PV

pod 使用声明（claim）作为卷。集群通过检查声明来找到绑定的卷，并为 pod 挂载相应的卷。对于那些支持多个访问模式的卷，您必须指定作为 pod 中的卷需要使用哪种模式。

在声明且该声明被绑定后，绑定的 PV 就会属于您，只要您需要它。您可以通过在 pod 的 volumes 定义中包括 **persistentVolumeClaim** 来调度 pod 并访问声明的 PV。请参见以下 语法详情。

4.3.2.4. PVC 保护

PVC 保护会被默认启用。

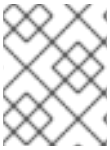
4.3.2.5. 释放卷

当不再需要使用一个卷时，您可以从 API 中删除 PVC 对象，这样相应的资源就可以被重新声明。当声明被删除时，该卷被视为"released"，但它尚不可用于其他声明。这是因为之前声明者的数据仍然还保留在卷中，这些数据必须根据相关政策进行处理。

4.3.2.6. 重新声明卷

PersistentVolume 的重新声明（reclaim）政策指定了在卷被释放后集群可以如何使用它。PV 的重新声明策略可以是 **Retain** 或 **Delete**。

- **Retain** 策略可为那些支持它的卷插件手动重新声明资源。
- **删除** 重新声明(reclaim)策略从 OpenShift Container Platform 以及外部基础架构（如 AWS EBS、GCE PD 或 Cinder 卷）中的相关存储资产中删除 **PersistentVolume** 对象。



注意

动态置备的卷具有默认的 **ReclaimPolicy** 值 **Delete**。手动置备的卷具有默认的 **ReclaimPolicy** 值 **Retain**。

4.3.2.7. 手动重新声明 PersistentVolume

删除 PersistentVolumeClaim 时，PersistentVolume 仍然存在，并被视为 "released"。但是，由于之前声明的数据保留在卷中，所以无法再使用 PV。

要以集群管理员的身份手动重新声明 PV:

1. 删除 PV：

```
$ oc delete <pv-name>
```

外部基础架构（如 AWS EBS、GCE PD、Azure Disk 或 Cinder 卷）中的关联的存储资产在 PV 被删除后仍然存在。

2. 清理相关存储资产中的数据。

3. 删除关联的存储资产。另外，若要重复使用同一存储资产，请使用存储资产定义创建新 PV。

重新声明的 PV 现在可供另一个 PVC 使用。

4.3.2.8. 更改重新声明策略

更改 PV 的重新声明策略：

1. 列出集群中的 PV：

```
$ oc get pv
```

输出示例

```
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
CLAIM            STORAGECLASS     REASON   AGE
 pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
default/claim1   manual                    10s
 pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
default/claim2   manual                    6s
 pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
default/claim3   manual                    3s
```

2. 选择一个 PV 并更改其重新声明策略：

```
$ oc patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

3. 验证您选择的 PV 是否有正确的策略：

```
$ oc get pv
```

输出示例

```
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
CLAIM            STORAGECLASS     REASON   AGE
 pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
default/claim1   manual                    10s
 pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
default/claim2   manual                    6s
 pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Retain          Bound
default/claim3   manual                    3s
```

在前面的输出中，绑定到声明 **default/claim3** 的 PV 现在具有 **Retain** 重新声明策略。当用户删除声明 **default/claim3** 时，不会自动删除 PV。

4.3.3. 持久性卷（PV）

每个 PV 都会包括一个 **spec** 和 **status**，它们分别代表卷的规格和状态，例如：

PV 对象定义示例

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /tmp
    server: 172.17.0.2
```

4.3.3.1. PV 类型

OpenShift Container Platform 支持 **以下** persistentVolume 插件：

- NFS
- hostPath
- GlusterFS
- gluster-block
- OpenShift Container Storage(OCS)文件
- OpenShift Container Storage(OCS)Block
- Ceph RBD
- OpenStack Cinder
- AWS Elastic Block Store (EBS)
- GCE Persistent Disk
- iSCSI
- Fibre Channel
- Azure Disk
- Azure File
- VMWare vSphere
- 本地

4.3.3.2. 容量

通常 PV 有特定的存储容量。这可以通过使用 PV 的 **capacity** 属性来设置。

目前，存储容量是唯一可以设置或请求的资源。以后可能会包括 IOPS 、 throughput 等属性。

4.3.3.3. 访问模式

一个 **PersistentVolume** 可以以资源供应商支持的任何方式挂载到一个主机上。提供商将具有不同的功能，每个 PV 的访问模式设置为该特定卷支持的特定模式。例如：NFS 可以支持多个读/写客户端，但一个特定的 NFS PV 可能会以只读方式导出。每个 PV 都有自己一组访问模式来描述指定的 PV 功能。

声明会与有类似访问模式的卷匹配。用来进行匹配的标准只包括访问模式和大小。声明的访问模式代表一个请求。比声明要求的条件更多的资源可能会匹配，而比要求的条件更少的资源则不会被匹配。例如：如果一个声明请求 RWO，但唯一可用卷是一个 NFS PV（RWO+ROX+RWX），则该声明与这个 NFS 相匹配，因为它支持 RWO。

系统会首先尝试直接匹配。卷的模式必须与您的请求匹配，或包含更多模式。大小必须大于或等于预期值。如果两种类型的卷（例如，NFS 和 iSCSI）具有相同的访问模式，则其中任何一个模式都可与这些模式匹配。不同的卷类型之间没有匹配顺序，在同时匹配时也无法选择特定的一个卷类型。

所有具有相同模式的卷都被分组，然后按大小排序（从小到大）。绑定器获取具有匹配模式的组，并逐一（按大小顺序）进行迭代，直至一个大小匹配。

下表列出了访问模式：

表 4.1. 访问模式

访问模式	CLI 缩写	描述
ReadWriteOnce	**RWO**	卷只可以被一个节点以读写模式挂载。
ReadOnlyMany	**ROX**	卷可以被多个节点以只读形式挂载。
ReadWriteMany	**RWX**	卷可以被多个节点以读写模式挂载。

重要

卷的 **AccessModes** 只是卷功能的一个描述符。它们不会被强制限制。存储供应商会最终负责处理由于资源使用无效导致的运行时错误。

例如，Ceph 提供 **ReadWriteOnce** 访问模式。如果您需要卷的访问模式为 ROX，则需要在声明中指定 **read-only**。供应商中的错误会在运行时作为挂载错误显示。

iSCSI 和 Fibre Channel（光纤通道）卷目前没有隔离机制。您必须保证在同一时间点上只在一个节点使用这些卷。在某些情况下，比如对节点进行 drain 操作时，卷可以被两个节点同时使用。在对节点进行 drain 操作前，需要首先确定使用这些卷的 pod 已被删除。

下表列出了不同的 PV 支持的访问模式：

表 4.2. 支持的 PV 访问模式

卷插件	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
AWS EBS	▮	-	-
Azure File	▮	▮	▮

卷插件	ReadWriteOnce	ReadOnlyMany	ReadWriteMany
Azure Disk	▮	-	-
Ceph RBD	▮	▮	-
Fibre Channel	▮	▮	-
GCE Persistent Disk	▮	-	-
GlusterFS	▮	▮	▮
gluster-block	▮	-	-
HostPath	▮	-	-
iSCSI	▮	▮	-
NFS	▮	▮	▮
OpenStack Cinder	▮	-	-
VMWare vSphere	▮	-	-
本地	▮	-	-

注意

为依赖 AWS EBS、GCE Persistent Disks 或 Openstack Cinder PV 的 pod 使用重新创建的部署策略。

4.3.3.4. 重新声明策略

下表列出了当前的重新声明策略：

表 4.3. 当前重新声明策略

重新声明策略	描述
Retain	允许手动回收.
删除	删除 PV 和关联的外部存储资产。

警告

如果您不想保留所有 pod，请使用动态置备。

4.3.3.5. 阶段

卷可以处于以下几个阶段：

表 4.4. 卷阶段

阶段	描述
Available	可用资源，还未绑定到任何声明
Bound	卷已绑定到一个声明。
Released	以前使用这个卷的声明已被删除，但该资源还没有被集群重新声明。
Failed	卷的自动重新声明失败。

CLI 显示与 PV 绑定的 PVC 名称。

4.3.3.6. 挂载选项

您可以使用注解 **volume.beta.kubernetes.io/mount-options** 指定在挂载持久性卷时的挂载选项。

例如：

挂载选项示例

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
  annotations:
    volume.beta.kubernetes.io/mount-options: rw,nfsvers=4,noexec 1
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  nfs:
    path: /tmp
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Retain
  claimRef:
    name: claim1
    namespace: default
```

1 在将 PV 挂载到磁盘时使用指定的挂载选项。

以下持久性卷类型支持挂载选项：

- NFS
- GlusterFS
- Ceph RBD
- OpenStack Cinder
- AWS Elastic Block Store (EBS)
- GCE Persistent Disk
- iSCSI
- Azure Disk
- Azure File
- VMWare vSphere

注意

Fibre Channel 和 HostPath 持久性卷不支持挂载选项。

4.3.3.7. 递归 chown

每次 PV 挂载到 Pod 时，或者在 Pod 重启时，卷的所有权和权限都会递归更改为与 Pod 那些匹配。当对 PV 中的所有文件和目录执行 **chown** 时，将组读取/写入访问权限添加到卷中。这可让 Pod 内运行的进程访问 PV 文件系统。用户可以通过在其 Pod 和安全上下文约束(SCC)中指定 **fsGroup** 来防止这些递归更改的所有权。

SELinux 标签 也会递归更改。用户不能阻止对 SELinux 标签进行递归更改。

注意

如果用户有大量文件（如 > 100,000），则在将 PV 挂载到 Pod 之前可能会发生大量延迟。

4.3.4. 持久性卷声明 (PVC)

每个 PVC 都会包括一个 **spec** 和 **status**，它们分别代表了声明的规格和状态，例如：

PVC 对象定义示例

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
```

```
  resources:
    requests:
      storage: 8Gi
  storageClassName: gold
```

4.3.4.1. 存储类

另外，通过在 **storageClassName** 属性中指定存储类的名称，声明可以请求一个特定的存储类。只有具有请求的类的 PV（ **storageClassName**的值与 PVC 中的值相同）才会与 PVC 绑定。集群管理员可配置动态置备程序为一个或多个存储类提供服务。集群管理员可根据需要创建与 PVC 的规格匹配的 PV。

集群管理员也可以为所有 PVC 设置默认存储类。当配置了默认存储类时， PVC 必须明确要求将存储类 **StorageClass** 或 **storageClassName**设为 **""**， 以便绑定到没有存储类的 PV。

4.3.4.2. 访问模式

声明在请求带有特定访问权限的存储时，使用与卷相同的格式。

4.3.4.3. Resources

象 pod 一样，声明可以请求具体数量的资源。在这种情况下，请求用于存储。同样的资源模型适用于卷和声明。

4.3.4.4. 声明作为卷

pod 通过将声明作为卷来访问存储。在使用声明时，声明需要和 pod 位于同一个命名空间。集群在 pod 的命名空间中找到声明，并使用它来使用这个声明后台的**PersistentVolume**。卷被挂载到主机和 pod 中，例如：

挂载卷到主机和 pod 示例

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

4.3.5. 块卷支持

重要

块卷支持是一个技术预览功能，它仅适用于手动置备的 PV。

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。

有关红帽技术预览功能支持范围的更多信息，请参阅 https://access.redhat.com/support/offerings/techpreview/。

您可以通过在 PV 和 PVC 规格中包含 API 字段来静态置备原始块卷。

要使用块卷，您必须首先启用 **BlockVolume** 功能门。要为 master 启用功能门，请将 **feature-gates** 添加到 **apiServerArguments** 和 **controllerArguments** 中。要为节点启用功能门，在 **kubeletArguments** 中添加 **功能门**。例如：

```
kubeletArguments:
  feature-gates:
    - BlockVolume=true
```

PV 示例

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block (1)
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

(1) **volumeMode** 项代表这个 PV 是原始块卷。

PVC 示例

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block (1)
```

```
  resources:
    requests:
      storage: 10Gi
```

1 **volumeMode** 项代表请求了一个原始块持久性卷。

pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: [ "tail -f /dev/null" ]
      volumeDevices:  1
        - name: data
          devicePath: /dev/xvda  2
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: block-pvc  3
```

1 **VolumeDevices** （与 **volumeMounts**类似）用于块设备，只能与 **PersistentVolumeClaim** 源一起使用。

2 **Device** Path（与 **mountPath**相似）代表到物理设备的路径。

3 卷源必须是 **persistentVolumeClaim** 类型，且必须与期望的 PVC 的名称匹配。

表 4.5. VolumeMode 可以使用的值

值	默认
Filesystem	是
Block	否

表 4.6. 块卷的绑定方案

PV VolumeMode	PVC VolumeMode	绑定结果
Filesystem	Filesystem	绑定
Unspecified	Unspecified	绑定

PV VolumeMode	PVC VolumeMode	绑定结果
Filesystem	Unspecified	绑定
Unspecified	Filesystem	绑定
Block	Block	绑定
Unspecified	Block	无绑定
Block	Unspecified	无绑定
Filesystem	Block	无绑定
Block	Filesystem	无绑定

重要

未指定值时将使用默认值 **Filesystem**。

4.4. 临时本地存储

4.4.1. 概述

注意

只有启用了临时存储技术预览功能时，才会应用此主题。此功能默认为禁用。如果启用，OpenShift Container Platform 集群将使用临时存储来存储在集群销毁后不需要保留的信息。要启用此功能，请参阅为临时存储配置。

除了持久性存储外，Pod 和容器还需要临时或临时本地存储才能进行操作。此临时存储的生命周期不会超过每个 pod 的生命周期，且此临时存储无法在 pod 间共享。

在 OpenShift Container Platform 3.10 之前，使用容器的可写层、日志目录和 EmptyDir 卷向 pod 公开临时本地存储。Pod 使用临时本地存储进行涂销空间、缓存和日志。与缺少本地存储相关的问题包括：

- Pod 不知道有多少可用的本地存储。
- Pod 无法请求保证的本地存储。
- 本地存储无法保证可以满足需求。
- Pod 可能会因为其他 pod 已使用完本地存储而被驱除。只有在足够的存储重新可用后，新的 pod 才可以使用。

与持久性卷不同，临时存储是无结构和共享、空间而不是实际的数据，除了系统、容器运行时和 OpenShift Container Platform 的其他使用之外，临时存储也不例外。临时存储框架允许 pod 指定其临时本地存储需求，OpenShift Container Platform 会在适当的位置调度 pod，并防止节点过度使用本地存储。

虽然临时存储框架允许管理员和开发人员更好地管理这个本地存储，但它不提供任何与 I/O 吞吐量和延迟有关的内容。

4.4.2. 临时存储的类型

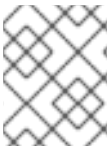
主分区中始终提供临时本地存储。创建主分区（root 和运行时）有两种基本方法。

4.4.2.1. root

默认情况下，这个分区包含 kubelet 的根目录、**/var/lib/origin/** 和 **/var/log/** 目录。此分区可以在用户 Pod、OS 和 Kubernetes 系统守护进程间共享。通过 EmptyDir 卷、容器日志、镜像层和容器可写入层，pod 可以消耗此分区。kubelet 管理这个分区的共享访问和隔离。这个分区是临时的，应用程序无法期望从这个分区中的任何性能 SLA、磁盘 IOPS。

4.4.2.2. Runtime

这是一个可选分区，可用于 overlay 文件系统。OpenShift Container Platform 会尝试识别并提供共享访问以及这个分区的隔离。容器镜像层和可写入层存储在此处。如果 runtime 分区存在，则 **root** 分区不包含任何镜像层或者其它可写入的存储。



注意

当您使用 DeviceMapper 提供运行时存储时，容器的写时复制层在临时存储中不会考虑。使用覆盖存储来监控此临时存储。

4.4.3. 管理临时存储

集群管理员可以通过 设置配额 在非终端状态的所有 Pod 中定义临时存储的限制范围和临时存储请求数量来管理项目中的临时存储。开发人员也可以在 Pod 和容器级别上设置此计算资源的 请求和限值。

4.4.4. 监控临时存储

您可以使用 **/bin/df** 作为监控临时容器数据所在卷的临时存储使用情况的工具，即 **/var/lib/origin** 和 **/var/lib/docker**。如果集群管理员将 **/var/lib/docker** 放置在单独的磁盘中，则使用 **df** 命令，则仅显示 **/var/lib/origin** 的可用空间。

使用 **df -h** 命令显示 **/var/lib** 中已用和可用空间的人类可读值：

```
$ df -h /var/lib
```

输出示例

```
Filesystem  Size  Used Avail Use% Mounted on
/dev/sda1    69G   32G   34G  49% /
```

4.5. 源控制管理

OpenShift Container Platform 利用托管内部（如内部 Git 服务器）或外部（如 GitHub、Bitbucket 等）的已存在的源控制管理(SCM)系统。目前，OpenShift Container Platform 仅支持 Git 解决方案。

SCM 集成与 构建 紧密耦合，两个点是：

- 使用存储库创建 **BuildConfig**，它允许在 OpenShift Container Platform 中构建应用程序。您可以通过检查您的存储库来手动创建 **BuildConfig**，或让 OpenShift Container Platform 自动创建它。
- 在存储库更改时触发构建。

4.6. ADMISSION CONTROLLER

4.6.1. 概述

准入(Admission)插件会在资源持久性前截获对 master API 的请求，但在请求被验证并被授权后。

在请求被接受至集群中之前，每个准入插件会按顺序运行。如果序列中的任何插件拒绝了请求，则整个请求将立即被拒绝，并且将返回一个错误给最终用户。

准入插件可能会在一些情况下修改传入对象，以应用系统配置的默认值。另外，准入插件可能会修改相关资源，作为请求处理的一部分来执行各种操作，如递增配额使用量。

警告

OpenShift Container Platform master 有一个默认插件列表，用于每种类型的资源（Kubernetes 和 OpenShift Container Platform）都默认启用。这些是 master 正常工作所需的。除非您严格了解自己正在做什么，否则不建议修改这些列表。以后版本的产品可能会使用不同的插件集合，并可能会改变它们的排序。如果您在 master 配置文件中覆盖默认插件列表，您需要更新它以反应 OpenShift Container Platform master 的较新版本的要求。

4.6.2. General Admission Rules

OpenShift Container Platform 为 Kubernetes 和 OpenShift Container Platform 资源使用单一准入链。这意味着顶级 **admissionConfig.pluginConfig** 元素现在可以包含插件配置，它们被包括在 **kubernetesMasterConfig.admissionConfig.pluginConfig** 中。

kubernetesMasterConfig.admissionConfig.pluginConfig 应该被移到 **admissionConfig.pluginConfig** 中。

所有支持的准入插件都在单一链中排序。您没有设置 **admissionConfig.pluginOrderOverride** 或 **kubernetesMasterConfig.admissionConfig.pluginOrderOverride**。反之，启用默认为关闭的插件，可以添加其插件相关的配置，或者添加 **DefaultAdmissionConfig** 小节，如下所示：

```
admissionConfig:
  pluginConfig:
    AlwaysPullImages: (1)
      configuration:
        kind: DefaultAdmissionConfig
        apiVersion: v1
        disable: false (2)
```

(1) 准入插件名称。

2 表示应启用插件。它是可选的，此处仅显示参考。

将 **disable** 设置为 **true** 将禁用默认为 on 的准入插件。

警告

准入(Admission)插件通常用于在 API 服务器上实施安全性。在禁用时要小心。

注意

如果您之前使用的是无法安全整合到单一准入链中的 **admissionConfig** 元素，您将在 API 服务器日志中收到警告，您的 API 服务器将会以两个单独的准入链启动。更新 **admissionConfig** 以解决警告。

4.6.3. 自定义准入插件

集群管理员可以配置一些准入插件来控制某些行为，例如：

- 每个用户限制自助项目数量
- 配置全局构建默认值和覆盖
- 控制 Pod 放置
- 管理角色绑定

4.6.4. 使用容器的准入控制器

使用容器的准入控制器也支持 init 容器。

4.7. 自定义 ADMISSION CONTROLLER

4.7.1. 概述

除了默认的准入控制器外，您还可以使用 *准入 webhook* 作为准入链的一部分。

准入 webhook 会在创建时调用 webhook 服务器来模拟 pod，如注入标签，或在准入过程中验证 pod 配置的特定方面。

准入 webhook 在资源持久性前截获到 master API 的请求，但在请求被身份验证和授权后。

4.7.2. Admission Webhooks

在 OpenShift Container Platform 中，您可以使用准入 webhook 对象，在 API 准入链中调用 Webhook 服务器。

您可以配置两类准入 Webhook 对象：

- 变异准入 Webhook 允许使用变异 Webhook 在资源内容被保留前修改资源内容。
- 验证准入 webhook 允许使用验证 Webhook 来实施自定义准入策略。

配置 webhook 和外部 Webhook 服务器超出了本文档的范围。但是，webhook 必须与一个接口关联才能与 OpenShift Container Platform 正常工作。

重要

Admission webhook 只是一个技术预览功能。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。

如需红帽技术预览功能支持范围的更多信息，请参阅 https://access.redhat.com/support/offerings/techpreview/。

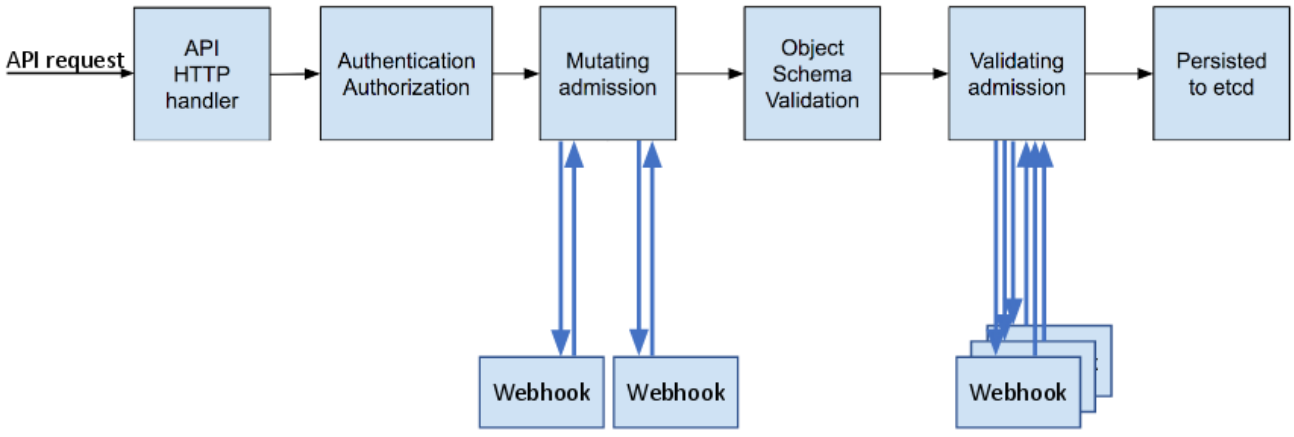
当对象实例化时，OpenShift Container Platform 会发出一个 API 调用来接受该对象。在准入过程中，*变异准入控制器* 可以调用 Webhook 来执行任务，如注入关联性标签。在准入过程结束时，*验证准入控制器* 可以调用 webhook 来确保正确配置了对象，如验证关联性标签。如果验证通过，OpenShift Container Platform 会按照配置来调度对象。

当 API 请求进入时，变异或验证准入控制器将使用配置中的外部 webhook 列表，并并行调用它们：

- **如果所有** webhook 都批准请求，准入链将继续。
- **如果有任何** webhook 拒绝了请求，则准入请求将被拒绝，且其实现的原因是 *第一个* webhook 拒绝原因。
  如果多个 webhook 拒绝准入请求，则只有第一个 webhook 才会返回给用户。
- 如果在调用 Webhook 时遇到错误，该请求将被拒绝，或忽略 Webhook。

准入控制器和 webhook 服务器之间的通信需要使用 TLS 保护。生成 CA 证书并使用证书为 webhook 服务器使用的服务器证书签名。PEM 格式的 CA 证书使用某种机制（如 Service Serving 证书 Secret ）提供给准入控制器。

下图说明了此过程有两个准入 Webhook，它们调用多个 webhook。



准入 Webhook 的一个简单示例用例是资源的各种验证。例如，您有一个基础架构要求所有 pod 都有一组通用的标签，如果 pod 没有这些标签，则您不想保留任何 pod。您可以编写 webhook 来注入这些标签和另一个 Webhook，以验证标签是否存在。然后，OpenShift Container Platform 将调度具有标签的 pod，并传递验证并拒绝因为缺少标签而未通过的 pod。

一些常见用例包括：

- 变异资源，将 side-car 容器注入 pod。
- 限制项目以阻止项目中的一些资源。
- 自定义资源验证，以对依赖字段执行复杂验证。

4.7.2.1. Admission Webhook 的类型

集群管理员可以在 API 服务器的 *准入链中包括 变异准入 webhook 或验证准入* webhook。

变异准入 Webhook 会在准入过程的变异阶段调用，这允许在保留资源内容前修改资源内容。一个变异准入 Webhook 示例是 Pod Node Selector 功能，它使用命名空间上的注解来查找标签选择器并将其添加到 pod 规格中。

Mutating Admission Webhook 配置示例

```
apiVersion: admissionregistration.k8s.io/v1beta1
 kind: MutatingWebhookConfiguration (1)
 metadata:
   name: <controller_name> (2)
 webhooks:
 - name: <webhook_name> (3)
   clientConfig: (4)
     service:
       namespace: (5)
       name: (6)
      path: <webhook_url> (7)
     caBundle: <cert> (8)
   rules: (9)
   - operations: (10)
     - <operation>
     apiGroups:
     - ""
     apiVersions:
     - "*"
     resources:
     - <resource>
   failurePolicy: <policy> (11)
```

(1) 指定变异准入 Webhook 配置。

(2) 准入 webhook 对象的名称。

(3) 要调用的 webhook 的名称。

(4) 如何连接、信任和将数据发送到 webhook 服务器的信息。

(5) 创建前端服务的项目。

(6) 前端服务的名称。

(7) 用于准入请求的 webhook URL。

8 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。

9 定义 API 服务器何时应使用此控制器的规则。

10 触发 API 服务器调用此控制器的操作：

- create
- update
- delete
- connect

11 指定如果 webhook 准入服务器不可用时策略应如何操作。**Ignore** （允许/失败打开）或 **Fail** （块/失败关闭）。

在准入过程 的验证阶段会调用验证准入 Webhook。在此阶段，可以在特定的 API 资源强制不能改变，以确保资源不会再次更改。Pod Node Selector 也是验证准入的一个示例，通过确保所有 **nodeSelector** 字段都受到项目的节点选择器限制。

Validating Admission Webhook 配置示例

```
apiVersion: admissionregistration.k8s.io/v1beta1
 kind: ValidatingWebhookConfiguration 1
 metadata:
   name: <controller_name> 2
 webhooks:
 - name: <webhook_name> 3
   clientConfig: 4
     service:
       namespace: default 5
       name: kubernetes 6
      path: <webhook_url> 7
     caBundle: <cert> 8
   rules: 9
   - operations: 10
     - <operation>
     apiGroups:
     - ""
     apiVersions:
     - "*"
     resources:
     - <resource>
   failurePolicy: <policy> 11
```

1 指定验证准入 Webhook 配置。

2 webhook 准入对象的名称。

3 要调用的 webhook 的名称。

4 如何连接、信任和将数据发送到 webhook 服务器的信息。

5 创建前端服务的项目。

6 前端服务的名称。

7 用于准入请求的 webhook URL。

8 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。

9 定义 API 服务器何时应使用此控制器的规则。

10 触发 API 服务器调用此控制器的操作。

- create
- update
- delete
- connect

11 指定如果 webhook 准入服务器不可用时策略应如何操作。**Ignore** （允许/失败打开）或 **Fail** （块/失败关闭）。

注意

失败打开可能会导致所有客户端的行为无法预计。

4.7.2.2. 创建 Admission Webhook

首先部署外部 webhook 服务器，并确保它正常工作。否则，根据 webhook 是否被配置为 **fail open** 或 **fail closed**，否则操作将会被无条件接受或拒绝。

1. 在 YAML 文件中配置一个 mutating 或 validating 准入 Webhook 对象。
2. 运行以下命令来创建对象：

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

创建准入 webhook 对象后，OpenShift Container Platform 需要几秒钟才能遵守新配置。

3. 为准入 Webhook 创建前端服务：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    role: webhook 1
  name: <name>
spec:
  selector:
    role: webhook 2
```

1 2 触发 Webhook 的任意格式的标签。

4. 运行以下命令来创建对象：

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

5. 将准入 webhook 名称添加到您要由 Webhook 控制的 pod：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    role: webhook (1)
  name: <name>
spec:
  containers:
    - name: <name>
      image: myrepo/myimage:latest
      imagePullPolicy: <policy>
      ports:
       - containerPort: 8000
```

(1) 触发 Webhook 的标签。

注意

有关如何为库构建您自己的安全且可移植的 webhook 准入服务器和通用的-admission-apiserver 的示例，请参阅 kubernetes-namespace- reservation 项目。

4.7.2.3. Admission Webhook 示例

以下是一个准入 Webhook 示例，它将不允许在命名空间被保留时创建命名空间 ：

```
apiVersion: admissionregistration.k8s.io/v1beta1
  kind: ValidatingWebhookConfiguration
  metadata:
    name: namespacereservations.admission.online.openshift.io
  webhooks:
  - name: namespacereservations.admission.online.openshift.io
    clientConfig:
      service:
        namespace: default
        name: webhooks
       path: /apis/admission.online.openshift.io/v1beta1/namespacereservations
      caBundle: KUBE_CA_HERE
    rules:
    - operations:
      - CREATE
      apiGroups:
      - ""
      apiVersions:
      - "b1"
      resources:
      - namespaces
    failurePolicy: Ignore
```

以下是一个由名为 *webhook* 的准入 webhook 评估的 pod 示例：

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    role: webhook
  name: webhook
spec:
  containers:
    - name: webhook
      image: myrepo/myimage:latest
      imagePullPolicy: IfNotPresent
      ports:
- containerPort: 8000
```

以下是 webhook 的前端服务：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    role: webhook
  name: webhook
spec:
  ports:
    - port: 443
      targetPort: 8000
  selector:
role: webhook
```

4.8. 其他 API 对象

4.8.1. LimitRange

限制范围提供了一种机制，可以强制对 Kubernetes 命名空间 中的资源实施 min/max 限制。

通过在命名空间中添加限制范围，您可以强制单个 pod 或容器消耗的最小和最大 CPU 和内存量。

对于 CPU 和内存限值，如果您指定了 **max** 值，但没有在 LimitRange 对象中指定 **min** 限制，则资源可能会消耗超过 **max** 值的 CPU/内存资源。

4.8.2. ResourceQuota

Kubernetes 可以限制在命名空间 中创建的对象数量，以及 命名空间中 跨对象请求的资源总量。这可通过多个团队（一个命名空间中的每个团队）共享一个 Kubernetes 集群，作为防止一个团队利用另一个集群资源团队的机制。

有关 **ResourceQuota** 的更多信息，请参阅集群管理。

4.8.3. 资源

Kubernetes 资源是 pod 或容器可以请求的、分配给或消耗的 Kubernetes **资源**。示例包括内存(RAM)、CPU、磁盘时间和网络带宽。

如需更多信息，请参阅开发人员指南。

4.8.4. Secret

Secret 用于存储敏感信息，如密钥、密码和证书。它们可以被相关的 pod 访问，但与它们的定义分开。

4.8.5. PersistentVolume

持久性卷是由集群管理员置备的基础架构中的对象(**PersistentVolume**)。持久性卷为有状态应用程序提供持久的存储。

4.8.6. PersistentVolumeClaim

PersistentVolumeClaim 对象是 Pod 作者对存储的请求。Kubernetes 将声明与可用卷池匹配，并将它们绑定到一起。然后，声明被 pod 用作卷。Kubernetes 确保卷与需要它的 pod 位于同一个节点上。

4.8.6.1. 自定义资源

自定义资源 是 Kubernetes API 的扩展，它扩展了 API，或者允许您将自己的 API 引入到项目或集群中。

请参阅使用自定义资源扩展 Kubernetes API。

4.8.7. OAuth 对象

4.8.7.1. OAuthClient

OAuthClient 代表一个 OAuth 客户端，如 RFC 6749, section 2 所述。

以下 **OAuthClient** 对象会自动创建：

openshift-web-console	用于为 Web 控制台请求令牌的客户端
openshift-browser-client	用来使用可处理交互式登录的用户代理在 /oauth/token/request 中请求令牌的客户端
openshift-challenging-client	用来使用可处理 WWW-Authenticate 质询的 user-agent 来请求令牌的客户端

OAuthClient 对象定义

```
kind: "OAuthClient"
accessTokenMaxAgeSeconds: null (1)
apiVersion: "oauth.openshift.io/v1"
metadata:
```

```
  name: "openshift-web-console" 2
  selflink: "/oapi/v1/oAuthClients/openshift-web-console"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01Z"
respondWithChallenges: false 3
secret: "45e27750-a8aa-11e4-b2ea-3c970e4b7ffe" 4
redirectURIs:
  - "https://localhost:8443" 5
```

1 访问令牌的生命周期（以秒为单位）（请参见下面的描述）。

2 **name** 用作 OAuth 请求中的 **client_id** 参数。

3 当 **respondWithChallenges** 设为 **true** 时，对 **/oauth/authorize** 的未经身份验证的请求将导致 **WWW-Authenticate** 质询（如果由配置的身份验证方法支持）。

4 **secret** 参数的值用作授权代码流中的 **client_secret** 参数。

5 一个或多个绝对 URI 可以放在 **redirectURIs** 部分中。带有授权请求发送的 **redirect_uri** 参数必须以指定的 **redirectURIs** 之一作为前缀。

accessTokenMaxAgeSeconds 值覆盖单个 OAuth 客户端的 master 配置文件中的默认 **accessTokenMaxAgeSeconds** 值。为客户端设置这个值时，允许在不影响其他客户端生命周期的情况下为那个客户端提供长期的访问令牌。

- 如果为 **null**，则使用 master 配置文件中的默认值。
- 如果设置为 **0**，则令牌不会过期。
- 如果设置为大于 **0** 的值，则为该客户端发出的令牌将获得指定过期时间。例如：**accessTokenMaxAgeSeconds:172800** 会导致令牌在发布后 48 小时过期。

4.8.7.2. OAuthClientAuthorization

OAuthClientAuthorization 代表一个特定 **OAuthClient** 的用户的批准，为用户提供一个具有特定范围的 **OAuthAccessToken**。

OAuthClientAuthorization 对象的创建是在对 **OAuth** 服务器的授权请求期间完成的。

OAuthClientAuthorization Object Definition

```
kind: "OAuthClientAuthorization"
apiVersion: "oauth.openshift.io/v1"
metadata:
  name: "bob:openshift-web-console"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
clientName: "openshift-web-console"
userName: "bob"
userUID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
scopes: []
```

4.8.7.3. OAuthAuthorizeToken

OAuthAuthorizeToken 代表 **OAuth** 授权代码，如 RFC 6749, 第 1.3.1 中所述。

OAuthAuthorizeToken 由对 /oauth/authorize 端点的请求创建，如 RFC 6749、4.1.1 部分 所述。

然后，可以使用 **OAuthAuthorizeToken** 获取 **OAuthAccessToken**，请求指向 **/oauth/token** 端点，如 RFC 6749, 4.1.3 中所述。

OAuthAuthorizeToken Object Definition

```
kind: "OAuthAuthorizeToken"
apiVersion: "oauth.openshift.io/v1"
metadata:
  name: "MDAwYjM5YjMtMzM1MC00NDY4LTkxODItOTA2OTE2YzE0M2Fj" 1
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
clientName: "openshift-web-console" 2
expiresIn: 300 3
scopes: []
redirectURI: "https://localhost:8443/console/oauth" 4
userName: "bob" 5
userUID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 6
```

1. **Name** 代表令牌名称，用作用于 OAuthAccessToken 的授权代码。
2. **clientName** 值是请求此令牌的 OAuthClient。
3. **expiresIn** 值是 creationTimestamp 的过期时间（以秒为单位）。
4. **redirectURI** 值是生成此令牌的授权流期间用户重定向到的位置。
5. **username** 代表令牌的名称，允许获取 OAuthAccessToken。
6. **userUID** 代表用户此令牌的 UID，允许获取 OAuthAccessToken。

4.8.7.4. OAuthAccessToken

OAuthAccessToken 代表 **OAuth** 访问令牌，如 RFC 6749、第 1.4 一节中所述 。

OAuthAccessToken 由对 **/oauth/token** 端点的请求创建，如 RFC 6749,第 4.1.3 部分中所述。

访问令牌用作 bearer 令牌来对 API 进行身份验证。

OAuthAccessToken Object Definition

```
kind: "OAuthAccessToken"
apiVersion: "oauth.openshift.io/v1"
metadata:
  name: "ODliOGE5ZmMtYzczYi00Nzk1LTg4MGEtNzQyZmUxZmUwY2Vh" 1
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:02-00:00"
clientName: "openshift-web-console" 2
expiresIn: 86400 3
scopes: []
```

```
redirectURI: "https://localhost:8443/console/oauth" 4
userName: "bob" 5
userUID: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 6
authorizeToken: "MDAwYjM5YjMtMzM1MC00NDY4LTkxODItOTA2OTE2YzE0M2Fj" 7
```

1. **name** 是令牌名称，用作向 API 进行身份验证的令牌。
2. **clientName** 值是请求此令牌的 OAuthClient。
3. **expiresIn** 值是 creationTimestamp 的过期时间（以秒为单位）。
4. **redirectURI** 是生成此令牌的授权流期间重定向到的用户的位置。
5. **userName** 代表此令牌许进行身份验证的用户。
6. **userUID** 代表此令牌许进行身份验证的用户。
7. **authorizeToken** 是用来获取此令牌的 OAuthAuthorizationToken 的名称（若有）。

4.8.8. 用户对象

4.8.8.1. 身份

当用户登录到 OpenShift Container Platform 时，使用已 配置的身份提供程序 进行修改。这决定了用户的身份，并向 OpenShift Container Platform 提供这些信息。

然后，OpenShift Container Platform 会为该身份查找 **User Identity Mapping** ：

注意

如果使用 **lookup** 映射方法（例如，如果您使用外部 LDAP 系统）配置了身份提供程序，则不会执行这种自动映射。您必须手动创建映射。如需更多信息，请参阅 查找映射方法。

- 如果**身份**已存在，但没有映射到一个**用户**，则登录会失败。
- 如果 **Identity** 已存在，它被映射到一个**用户**，这个用户被提供了一个映射到 **User** 的 **OAuthAccessToken**。
- 如果**身份**不存在，则创建一个 **Identity**、 **User** 和 **UserIdentityMapping**，用户被指定为映射用户的 **OAuthAccessToken**。

身份 对象定义

```
kind: "Identity"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "anypassword:bob" 1
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
providerName: "anypassword" 2
providerUserName: "bob" 3
```

```
user:
  name: "bob" 4
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe" 5
```

1. 身份名称必须采用 providerName:providerUserName 格式。
2. **providerName** 是身份提供程序的名称。
3. **providerUserName** 是在身份提供程序范围内唯一代表此身份的名称。
4. **user** 参数中 **的名称是** 此身份映射到的用户名。
5. **uid** 代表此身份映射到的用户的 UID。

4.8.8.2. User

User 代表系统中的一个用户。通过将角色添加到用户或其组，即可 授予用户权限。

用户对象在首次登录时自动创建，或者可以通过 API 创建。

注意

OpenShift Container Platform 用户名不能包括 /、**:** 和 **%**。

用户 对象定义

```
kind: "User"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "bob" 1
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
  creationTimestamp: "2015-01-01T01:01:01-00:00"
identities:
  - "anypassword:bob" 2
fullName: "Bob User" 3
```

1. **name** 是向用户添加角色时使用的用户名。
2. **身份** 中的值是映射到此用户的身份对象。对于无法登录的用户，可以是 **null** 或 empty。
3. **fullName** 值是一个可选的用户显示名称。

4.8.8.3. UserIdentityMapping

UserIdentityMapping 将一个 **Identity** 映射到一个 **User**.

创建、更新或删除一个 **UserIdentityMapping**，会修改 **Identity** 和 **User** 项中的相关字段。

一个 **Identity** 只能映射到一个单一的**用户**，因此使用一个特定的身份进行登陆会决定**用户**。

一个**用户**可以有多个映射到它的身份。这允许使用多种登录方法来代表同一**用户**。

UserIdentityMapping Object Definition

```
kind: "UserIdentityMapping"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "anypassword:bob" (1)
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
  resourceVersion: "1"
identity:
  name: "anypassword:bob"
  uid: "9316ebad-0fde-11e5-97a1-3c970e4b7ffe"
user:
  name: "bob"
  uid: "9311ac33-0fde-11e5-97a1-3c970e4b7ffe"
```

(1) **useridentitymapping** name 与映射的 **Identity** name 匹配

4.8.8.4. 组

Group 表示系统中的一组用户。组可以通过为用户或他们的组添加角色来对权限进行分组。

组 对象定义

```
kind: "Group"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "developers" (1)
  creationTimestamp: "2015-01-01T01:01:01-00:00"
users:
  - "bob" (2)
```

(1) **name** 是向组添加角色时使用的组名称。

(2) **用户** 中的值是此组成员的用户对象的名称。

第 5 章 网络

5.1. 网络

5.1.1. 概述

Kubernetes 可确保 pod 能够相互联网，并从内部网络为每个 pod 分配一个 IP 地址。这样可保证 pod 中所有容器的行为如同它们在同一主机上一样。为每个 pod 指定专属的 IP 地址，意味着在端口分配、联网、命名、服务发现、负载均衡、应用程序配置和迁移方面，可以像物理主机或虚拟机一样来对待 pod。

不需要在 pod 间创建链接，我们不推荐使用 IP 地址直接与 pod 进行通信。相反，建议您创建一个服务（service），然后使用服务进行交互。

5.1.2. OpenShift Container Platform DNS

如果您 运行多个服务 （如用于多个 pod 的前端和后端服务），以使 frontend pod 与后端服务通信，则要为用户名、服务 IP 等创建环境变量。如果删除并重新创建服务，可以为该服务分配一个新的 IP 地址，并且需要重新创建 frontend pod 来获取服务 IP 环境变量的更新值。另外，必须在任何 frontend pod 之前创建后端服务，以确保正确生成服务 IP，并将它作为环境变量提供给 frontend pod。

因此，OpenShift Container Platform 具有一个内置 DNS，以便服务 DNS 以及服务 IP/端口能够访问这些服务。OpenShift Container Platform 支持在回答服务的 DNS 查询的 master 上运行 SkyDNS 来拆分 DNS。默认情况下，主设备侦听端口 53。

当节点启动时，以下消息表示 Kubelet 被正确地解析到 master：

```
0308 19:51:03.118430    4484 node.go:197] Started Kubelet for node
openshiftdev.local, server at 0.0.0.0:10250
I0308 19:51:03.118459    4484 node.go:199]   Kubelet is setting 10.0.2.15 as a
DNS nameserver for domain "local"
```

如果没有显示第二条消息，则 Kubernetes 服务可能不可用。

在节点主机上，每个容器的名称服务器都有添加到前端的 master 名称，容器的默认搜索域将是 .<***pod_namespace&*** **gt; .cluster.local**。容器随后会在节点上的任何其他名称服务器之前将任何名称服务器查询定向到主名称服务器，这是 Docker 格式的容器的默认行为。主控机将在具有以下格式的 **.cluster.local** 域中应答查询：

表 5.1. DNS 示例名称

对象类型	示例
默认	<pod_namespace>.cluster.local
服务	<service>.<pod_namespace>.svc.cluster.local
Endpoints	<name>.<namespace>.endpoints.cluster.local

这可以防止重启 frontend pod 以便获取新服务，这将为服务创建新 IP。这也无需使用环境变量，因为 pod 可以使用服务 DNS。另外，由于 DNS 不会改变，您也可以将数据库服务引用为 **db.local**。也支持通配符查找，因为查找会解析到服务 IP，并且无需在任何 frontend pod 之前创建后端服务，因为服务名称（以及 DNS）已提前建立。

此 DNS 结构还包括无头服务，其中门户 IP 没有分配给服务，kube-proxy 不会为其端点提供负载均衡或提供路由。服务 DNS 仍可被使用，并响应多个 A 记录，一个用于服务的每个 pod，允许客户端在每个 Pod 间进行循环。

5.2. OPENSHIFT SDN

5.2.1. 概述

OpenShift Container Platform 使用软件定义网络 (SDN) 方法来提供一个统一的集群网络，它允许 OpenShift Container Platform 集群中的不同 pod 相互间进行通信。此 pod 网络是由 OpenShift SDN 建立和维护的，它使用 Open vSwitch (OVS) 配置覆盖网络。

OpenShift SDN 提供三个 SDN 插件来配置 pod 网络：

- **ovs-subnet** 插件是原始插件，它提供了一个 "flat" pod 网络，每个 pod 可以与所有其他 pod 和服务通信。
- **ovs-multitenant** 插件为 pod 和服务提供项目级别的隔离。每个项目收到一个唯一的虚拟网络 ID(VNID)，用于标识分配给项目的 pod 的流量。来自不同项目的 Pod 不能与不同项目的 Pod 和服务互相发送或接收数据包。
  但是，接收 VNID 0 的项目更为特权，它们被允许与所有其他 pod 通信，所有其他 pod 可以与它们通信。在 OpenShift Container Platform 集群中，**default** 项目具有 VNID 0。这有助于某些服务（如负载均衡器）与集群中的所有其他 pod 通信，反之亦然。
- **ovs-networkpolicy** 插件允许项目管理员使用 NetworkPolicy 对象配置自己的隔离策略。

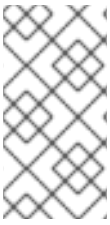
注意

有关在主控机和节点上配置 SDN 的信息，请参见配置 SDN。

5.2.2. 在 Master 上设计

在 OpenShift Container Platform master 上，OpenShift SDN 维护一个节点 registry，存储在 **etcd** 中。当系统管理员注册节点时，OpenShift SDN 从集群网络分配未使用的子网，并将该子网存储在 registry 中。删除节点时，OpenShift SDN 将从 registry 中删除子网，并考虑重新分配的子网。

在默认配置中，集群网络是 **10.128.0.0/14** 网络(i.e. **10.128.0.0 - 10.131.255.255**)，节点将被分配 **/23** 子网（如 **10.128.0.0/23**、**10.128.2.0/23**、**10.128.4.0/23** 等）。这意味着集群网络有 512 个子网可用于分配给节点，给定节点分配 510 地址，可分配给其上运行的容器。集群网络的大小和地址范围可以配置，因为 是主机子网大小。



注意

如果子网扩展到下一个较高的八进制数，它将轮转，以便首先为共享八进制数中带有 0 的子网位。例如，如果网络为 10.1.0.0/16（带有 **hostsubnetlength=6**，则子网为 10.1.0.0/26 和 10.1.1.0/26）到 10.1.255.0/26，则在填充 10.1.0.64/26 之前分配 10.0.64/26、10.1.1.64/26。这样可保证子网更容易被使用。

请注意，master 上的 OpenShift SDN 不会配置本地(master)主机来访问任何集群网络。因此，master 主机无法通过集群网络访问 pod，除非它也作为节点运行。

当使用 **ovs-multitenant** 插件时，OpenShift SDN master 还监视项目的创建和删除，并为它们分配 VXLAN VNID，稍后节点会正确隔离流量。

5.2.3. 在节点上设计

在节点上，OpenShift SDN 首先在上述 registry 中注册带有 SDN master 的本地主机，以便主节点将子网分配给节点。

接下来，OpenShift SDN 会创建和配置三个网络设备：

- **br0** ：将 pod 容器附加到的 OVS 网桥设备。OpenShift SDN 还在这个网桥上配置一组特定于非子网的流规则。
- **Tun0** ：OVS 内部端口（ **br0**上的端口 2）。这会分配集群子网网关地址，用于外部网络访问。OpenShift SDN 配置 **netfilter** 和路由规则，以启用通过 NAT 从集群子网到外部网络的访问。
- **vxlan_sys_4789**:OVS VXLAN 设备（ **br0**上的端口 1），提供对远程节点上容器的访问。在 OVS 规则中称为 **vxlan0**。

每次在主机上启动 pod 时，OpenShift SDN：

1. 从节点的集群子网分配 pod 的一个可用 IP 地址。
2. 将 pod 的 veth 接口对的主机一侧连接到 OVS 网桥 **br0**。
3. 为 OVS 数据库添加 OpenFlow 规则，将寻址到新 pod 的流量路由到正确的 OVS 端口。
4. 对于 **ovs-multitenant** 插件，添加 OpenFlow 规则来标记来自带有 pod 的 VNID 的 pod 的流量，并允许流量与 pod 的 VNID 匹配（或是特权 VNID 0）。非匹配流量通过通用规则过滤。

OpenShift SDN 节点也从 SDN 主控机监控子网更新。当添加新子网时，节点在 **br0** 上添加 OpenFlow 规则，以便远程子网中具有目的地 IP 地址的数据包进入 **vxlan0** （ **br0**上的端口 1），然后传输到网络上。**ovs-subnet** 插件通过 VNID 0 在 VXLAN 间发送所有数据包，但是 **ovs-multitenant** 插件将适当的 VNID 用于源容器。

5.2.4. 数据包流

假设您有两个容器：A 和 B。其中容器 A 的 **eth0** 的对等虚拟以太网设备名为 **vethA**，而容器 B 的 **eth0** 的对等设备名为 **vethB**。

注意

如果您尚未熟悉 Docker 服务使用对等虚拟以太网设备，请参阅 Docker 的高级网络文档。

现在假定容器 A 存在于本地主机上，容器 B 也在本地主机上。然后，从容器 A 到容器 B 的数据包流如下：

eth0 *(A'的netns)* → ***vethA*** → ***br0*** → ***vethB*** → ***eth0*** *(B'的netns)*

接下来，假设容器 A 位于本地主机，容器 B 位于集群网络上的远程主机上。然后，从容器 A 到容器 B 的数据包流如下：

eth0 *(in A's netns)*→ ***vethA*** → ***br0*** → ***vxlan0*** → *network* [1] → ***vxlan0*** → ***br0*** → ***vethB*** → ***eth0*** *（在 B 的 netns）*

最后，如果容器 A 连接到外部主机，流量类似如下：

eth0 *（在 A 的netns）* → ***vethA*** → ***br0*** → ***tun0*** →*(NAT)*→ ***eth0*** *（物理设备）* → *Internet*

几乎所有数据包交付决策都通过 OVS 网桥 **br0** 中的 OpenFlow 规则来执行，这简化了插件网络架构并提供灵活的路由。对于 **ovs-multitenant** 插件，这还提供强制的 网络隔离。

5.2.5. 网络隔离

您可以使用 **ovs-multitenant** 插件来实现网络隔离。当数据包退出分配给非默认项目的 pod 时，OVS 网桥 **br0** 标签与项目分配的 VNID 的数据包。如果数据包定向到节点集群子网中的另一 IP 地址，OVS 网桥仅允许在 VNID 匹配时将数据包传送到目的地 pod。

如果通过 VXLAN 隧道从另一节点收到数据包，则 Tunnel ID 用作 VNID，并且 OVS 网桥仅允许将数据包传送到本地 pod（如果隧道 ID 与目标 pod 的 VNID 匹配）。

目标为其他集群子网的数据包使用其 VNID 标记，并传送到 VXLAN 隧道上，后者具有拥有集群子网的节点的隧道目标地址。

如前文所述，VNID 0 在那个流量中具有特权，但允许任何 VNID 0 的流量输入任何分配了 VNID 0 的容器集，并且允许 VNID 0 的流量进入任何 pod。只有 **default** OpenShift Container Platform 项目被分配一个 VNID 0；所有其他项目都被分配了一个唯一的、启用了隔离的 VNID。集群管理员可以选择使用管理员 CLI 控制项目的 pod 网络。

5.3. 可用的 SDN 插件

OpenShift Container Platform 支持 Kubernetes Container Network Interface(CNI) 作为 OpenShift Container Platform 和 Kubernetes 之间的接口。软件定义型网络(SDN)插件与您的网络需求匹配。可以根据需要添加支持 CNI 接口的附加插件。

5.3.1. OpenShift SDN

OpenShift SDN 会被默认安装并配置，作为基于 Ansible 的安装过程的一部分。如需更多信息，请参阅 OpenShift SDN 部分。

5.3.2. 第三方 SDN 插件

5.3.2.1. Cisco ACI SDN

用于 OpenShift Container Platform 的 Cisco ACI CNI 插件提供了 Cisco Application Policy Infrastructure Controller(Cisco APIC)控制器和一个或多个连接到 Cisco ACI 光纤的 OpenShift Container Platform 集群之间的集成。

这个集成在两个主要功能区域实施：

1. Cisco ACI CNI 插件将 ACI fabric 功能扩展到 OpenShift Container Platform 集群，以便为 OpenShift Container Platform 工作负载提供 IP 地址管理、网络、负载均衡和安全功能。Cisco ACI CNI 插件将所有 OpenShift Container Platform Pod 连接到 Cisco ACI 提供的集成 VXLAN 覆盖。

2. Cisco ACI CNI 插件将整个 OpenShift Container Platform 集群建模为 Cisco APIC 上的 VMM 域。这为 APIC 提供访问 OpenShift Container Platform 集群资源清单的访问权限，包括 OpenShift Container Platform 节点、OpenShift Container Platform 命名空间、服务、部署、其 IP 和 MAC 地址、它们所使用的接口等等。APIC 使用此信息来自动关联物理和虚拟资源，以简化操作。

Cisco ACI CNI 插件旨在为 OpenShift Container Platform 开发人员和管理员透明地集成，并可从操作角度无缝集成。

如需更多信息，请参阅 Red Hat OpenShift Container Platform 架构和设计指南的 Cisco ACI CNI 插件 。

5.3.2.2. Flannel SDN

flannel 是专为容器而设计的虚拟网络层。OpenShift Container Platform 可以将其用于网络容器，而不是默认的软件定义型网络(SDN)组件。这在也依赖 SDN（如 OpenStack）的云供应商平台中运行 OpenShift Container Platform 非常有用，并且您希望避免将数据包封装两个平台两次。

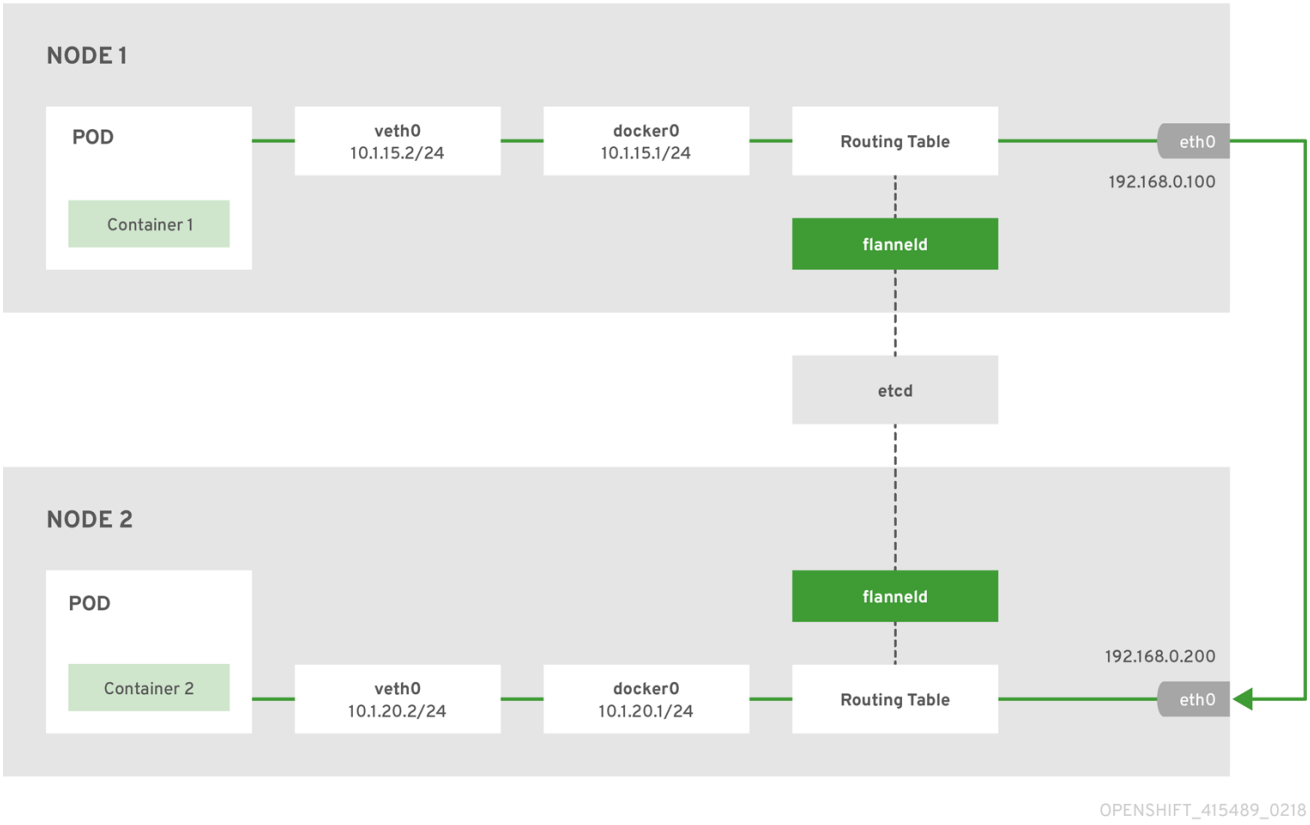
架构

OpenShift Container Platform 在 **host-gw** 模式下运行 **flannel**，它将容器的路由映射到容器。网络中的每个主机都运行一个名为 **flanneld** 的代理，该代理负责：

- 管理每个主机上的唯一子网
- 将 IP 地址分发到其主机上的每个容器
- 将路由从一个容器映射到另一个容器，即使在不同的主机上

每个 **flanneld** 代理都向集中式 **etcd** 存储提供此信息，因此主机上的其他代理可以将数据包路由到 **flannel** 网络中的其他容器。

下图演示了使用 **flannel** 网络从一个容器到另一个容器的架构和数据流：



节点 1 将包含以下路由：

```
default via 192.168.0.100 dev eth0 proto static metric 100
10.1.15.0/24 dev docker0 proto kernel scope link src 10.1.15.1
10.1.20.0/24 via 192.168.0.200 dev eth0
```

节点 2 将包含以下路由：

```
default via 192.168.0.200 dev eth0 proto static metric 100
10.1.20.0/24 dev docker0 proto kernel scope link src 10.1.20.1
10.1.15.0/24 via 192.168.0.100 dev eth0
```

5.3.2.3. NSX-T SDN

VMware NSX-T ™ Data Center 提供了一个基于策略的覆盖网络，通过从第 2 层到第 7 层网络服务（如交换、路由、访问控制、触发、触发和 QoS）为原生 OpenShift Container Platform 网络功能提供基于策略的覆盖网络。

NSX-T 组件可以安装和配置为 Ansible 安装流程的一部分，该流程将 OpenShift Container Platform SDN 集成到数据中心范围内的 NSX-T 虚拟网络中，连接了裸机、虚拟机和 OpenShift Container Platform pod。如需有关如何使用 VMware NSX-T 安装和更新 OpenShift Container Platform 的信息，请参阅安装部分。

NSX-T Container Plug-In(NCP)将 OpenShift Container Platform 集成到一个 NSX-T Manager 中，后者通常为整个数据中心配置。

有关 NSX-T Data Center 构架和管理的信息，请参阅 VMware NSX-T Data Center v2.4 文档 和 NSX-T NCP 配置指南。

5.3.2.4. Nuage SDN

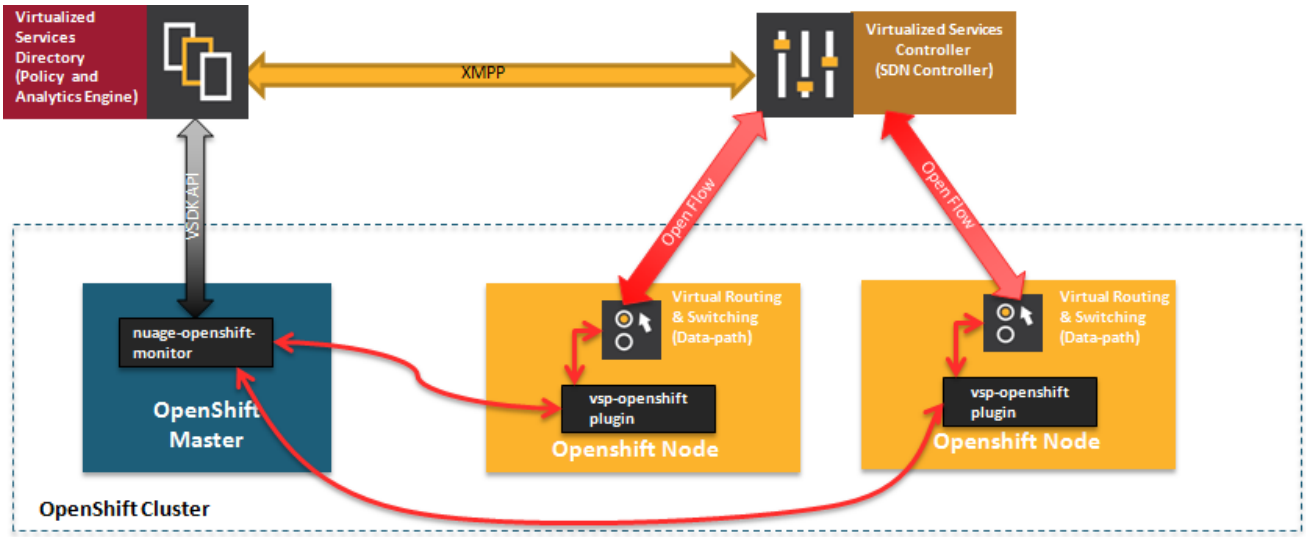
Nuage Networks 的 SDN 解决方案为 OpenShift Container Platform 集群中的 pod 提供高度可扩展且基于策略的覆盖网络。Nuage SDN 可以安装和配置为基于 Ansible 的安装过程的一部分。如需有关如何使用 Nuage SDN 安装和更新 OpenShift Container Platform 的信息，请参阅 高级安装 部分。

Nuage Networks 提供了一个高度可扩展的、基于策略的 SDN 平台，名为 Virtualized Services Platform(VSP)。Nuage VSP 使用 SDN 控制器，以及用于数据平面的开源 Open vSwitch。

Nuage 使用覆盖在 OpenShift Container Platform 和其它由虚拟机和裸机服务器组成的环境之间提供基于策略的网络。平台的实时分析引擎为 OpenShift Container Platform 应用程序启用可见性和安全监控。

Nuage VSP 与 OpenShift Container Platform 集成，通过删除 DevOps 团队面临的网络布局，快速启用和更新业务应用程序。

图 5.1. Nuage VSP 与 OpenShift Container Platform 集成



负责集成的两个具体组件。

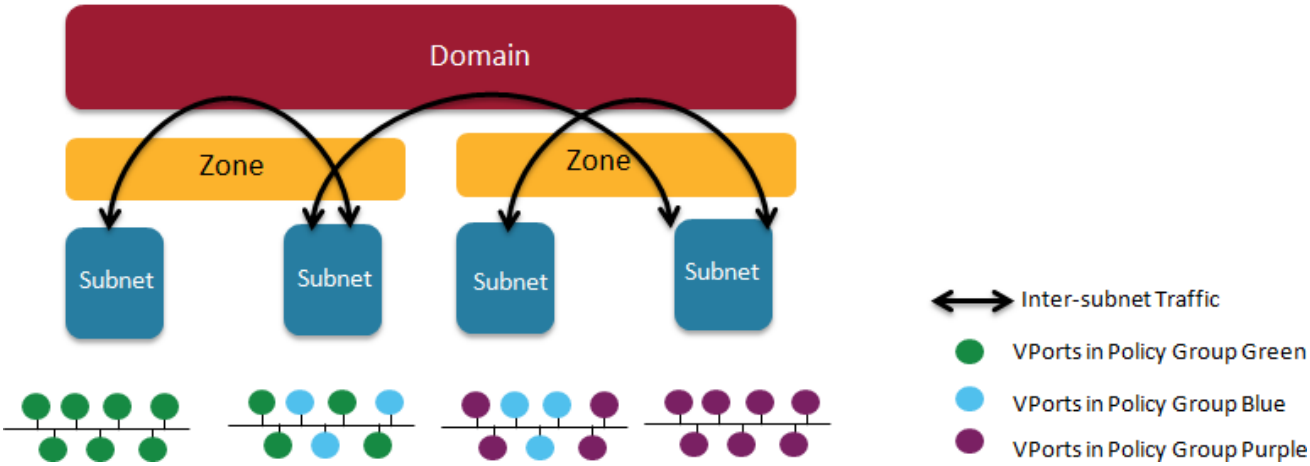
1. **nuage-openshift-monitor** 服务，作为 OpenShift Container Platform master 节点上的单独服务运行。

2. **vsp-openshift** 插件，由 OpenShift Container Platform 运行时在集群的每个节点上调用。

Nuage 虚拟路由和交换机软件(VRS)基于开源 Open vSwitch，负责数据路径转发。VRS 在每个节点上运行，并从控制器获取策略配置。

Nuage VSP 术语

图 5.2. Nuage VSP 构建块



1. 域：一个机构包含一个或多个域。域是单个"Layer 3"空间。在标准网络术语中，域映射到 VRF 实例。

2. zones：zones 在域下定义。区域不直接映射到网络上的任何内容，而是充当与哪些策略关联的对象，以便区域中的所有端点都遵循相同的策略集合。

3. 子网：子网在区域中定义。子网是域实例中的特定第 2 层子网。子网在域中是唯一的，也就是说，域中的子网不允许重叠或根据标准 IP 子网定义包含其他子网。

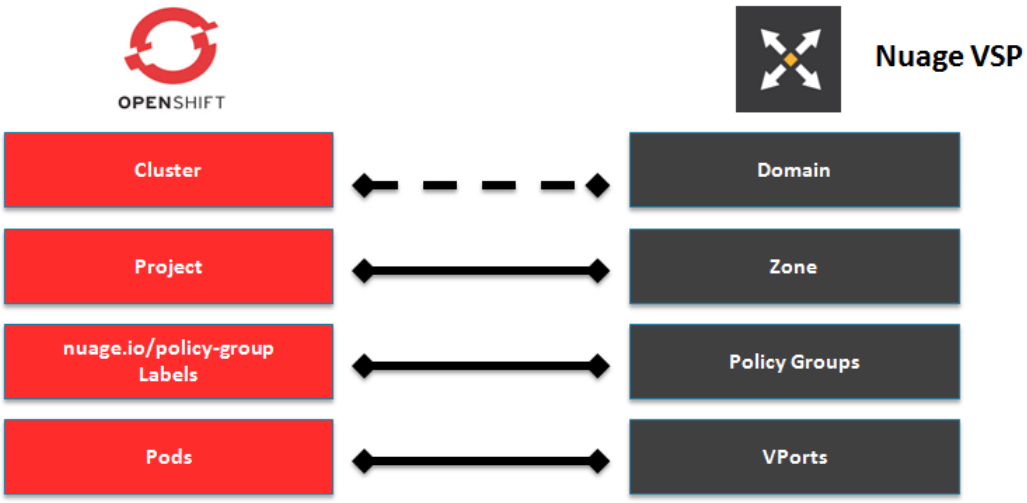
4. VPorts：VPort 是域层次结构中的一个新级别，旨在提供更精细的配置。除了容器和虚拟机外，VPort 也用于附加主机和网桥接口，它们可提供到裸机服务器、设备和传统 VLAN 的连接。

5. 策略组：策略组是 VPorts 的集合。

结构映射

许多 OpenShift Container Platform 概念 直接映射到 Nuage VSP 构造：

图 5.3. Nuage VSP 和 OpenShift Container Platform 映射



Nuage 子网不映射到 OpenShift Container Platform 节点，但特定项目的子网可以跨越 OpenShift Container Platform 中的多个节点。

在 OpenShift Container Platform 中生成的 pod 转换为在 VSP 中创建的虚拟端口。**vsp-openshift** 插件与 VRS 交互，并通过 VSC 从 VSD 获取该虚拟端口的策略。策略组支持将多个 pod 分组在一起，它们必须应用相同的策略集合。目前，pod 只能利用在 VSD 中管理用户创建策略组的操作工作流分配到策略组。pod 是策略组的一部分，方法是使用 Pod 规格中的 **nuage.io/policy-group** 标签来指定。

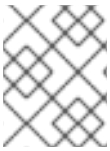
集成组件

Nuage VSP 使用两个主要组件与 OpenShift Container Platform 集成：

1. **nuage-openshift-monitor**
2. **vsp-openshift 插件**

nuage-openshift-monitor

nuage-openshift-monitor 是一个服务，它监控 OpenShift Container Platform API 服务器以创建项目、服务、用户、用户组等。



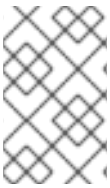
注意

如果是具有多个 master 的高可用性(HA)OpenShift Container Platform 集群，则 **nuage-openshift-monitor** 进程可在所有 master 上独立运行，而无需更改功能。

对于开发人员工作流，n **uage-openshift-monitor** 还通过简化 VSD REST API 来自动创建 VSD 对象，以将 OpenShift Container Platform 构造映射到 VSP 结构。每个集群实例映射到 Nuage VSP 中的单个域。这允许给定企业可能有多个集群安装 - 每个用于该企业的域实例在 Nuage 中。每个 OpenShift Container Platform 项目都映射到集群域中的区(Nuage VSP)。每当 **nuage-openshift-monitor** 可以看到项目的添加或删除时，它会使用与该项目对应的 VSDK API 实例化区，并为该区分配一个子网块。另外，**nuage-openshift-monitor** 还为该项目创建一个网络宏组。同样，当 **nuage-openshift-monitor** 看到服务添加或提取时，它会创建与服务 IP 对应的网络宏，并为该项目（用户提供网络宏组）分配到该项目的网络宏组（也支持使用标签实现与该服务的通信）。

对于开发人员工作流，在区中创建的所有 pod 都从该子网池获取 IP。子网池分配和管理由 **nuage-openshift-monitor** 根据 master-config 文件中的几个插件特定参数来完成。但是，实际 IP 地址解析和 vport 策略解析仍由 VSD 根据项目创建时实例化的域/区完成。如果初始子网池已耗尽，**nuage-openshift-monitor** 会从集群 CIDR 中分离一个额外的子网，以分配给给定项目。

对于操作工作流，用户在其应用程序或 pod 规格中指定 Nuage 识别标签，以便将 pod 解析为特定用户定义的区域和子网。但是，这无法用于解析通过 **nuage-openshift-monitor** 通过开发人员工作流创建的区或子网中的 pod。



注意

在操作工作流中，管理员负责预重新创建 VSD 结构，将 pod 映射到特定区域/子网，以及允许 OpenShift 实体（ACL 规则、策略组、网络宏和网络宏组）之间的通信。有关如何使用 Nuage 标签的详细描述，请参阅 Nuage VSP Openshift 集成指南。

vsp-openshift 插件

vsp-openshift 网络插件由 OpenShift Container Platform 运行时在每个 OpenShift Container Platform 节点上调用。它实现了网络插件 init 和 pod 设置、teardown 和 status hook。vsp-openshift 插件也负责为 pod 分配 IP 地址。特别是，它会与 VRS（转发引擎）通信，并在 pod 上配置 IP 信息。

5.3.3. OpenShift Container Platform 的 Kuryr SDN

Kuryr （或更具体为 Kuryr-Kubernetes）是一个使用 CNI 和 OpenStack Neutron 构建的 SDN 解决方案。其优点包括能够使用多种 Neutron SDN 后端，并在 Kubernetes pod 和 OpenStack 虚拟机(VM)之间提供互联性。

Kuryr-Kubernetes 和 OpenShift Container Platform 集成主要针对在 OpenStack 虚拟机上运行的 OpenShift Container Platform 集群设计。

5.3.3.1. OpenStack 部署要求

Kuryr SDN 对将要使用的 OpenStack 配置有一些要求。特别是：

- 最小服务集是 Keystone 和 Neutron。
- 它适用于 Octavia。
- 必须启用中继端口扩展。
- Neutron 必须使用 Open vSwitch 防火墙驱动程序。

5.3.3.2. kuryr-controller

Kuryr-controller 是一个服务，负责监视正在为新 pod 生成并创建 Neutron 资源的 OpenShift Container Platform API。例如，当创建 pod 时，kuryr-controller 会注意到并调用 OpenStack Neutron 来创建新端口。然后，有关该端口（或 VIF）的信息会被保存到 pod 的注解中。kuryr-controller 也能够使用预先创建的端口池来更快地创建 pod。

目前，kuryr-controller 必须作为单一服务实例运行，其在 OpenShift Container Platform 中的 **Deployment** 为 **replicas=1**。它需要访问底层的 OpenStack 服务 API。

5.3.3.3. kuryr-cni

Kuryr-cni 容器在 Kuryr-Kubernetes 部署中提供两个角色。它负责在 OpenShift Container Platform 节点上安装和配置 Kuryr CNI 脚本，并运行在主机上联网 **Pod** 的 kuryr-daemon 服务。这意味着 kuryr-cni 容器需要在每个 OpenShift Container Platform 节点上运行，因此它被建模为 **DaemonSet**。

每当从 OpenShift Container Platform 主机生成或删除新 pod 时，OpenShift Container Platform CNI 都会调用 Kuryr CNI 脚本。该脚本从 Docker API 获取本地 kuryr-cni 的容器 ID，并通过 docker exec 传递所有 CNI 调用参数来执行 Kuryr CNI 插件二进制代码。然后，该插件通过本地 HTTP 套接字调用 kuryr-daemon，再次传递所有参数。

Kuryr-daemon 服务负责监控为它们创建的 Neutron VIFs 的 **Pod** 的注解。当收到给定 **Pod** 的 CNI 请求时，守护进程在内存中有 VIF 信息，或等待注解出现在 **Pod** 定义中。已知所有联网操作中的 VIF 信息发生后。

5.4. 可用的路由器插件

路由器可以分配给节点，以控制 OpenShift Container Platform 集群中的流量。OpenShift Container Platform 使用 HAProxy 作为默认路由器，但选项可用。

5.4.1. HAProxy 模板路由器

HAProxy 模板路由器实施是模板路由器插件的参考实施。它使用 **openshift3/ose-haproxy-router** 存储库运行 HAProxy 实例以及模板路由器插件。

模板路由器有两个组件：

- 监视端点和路由的打包程序，并基于更改导致 HAProxy 重新加载
- 基于路由和端点构建 HAProxy 配置文件的控制器

注意

HAProxy 路由器使用版本 1.8.1。

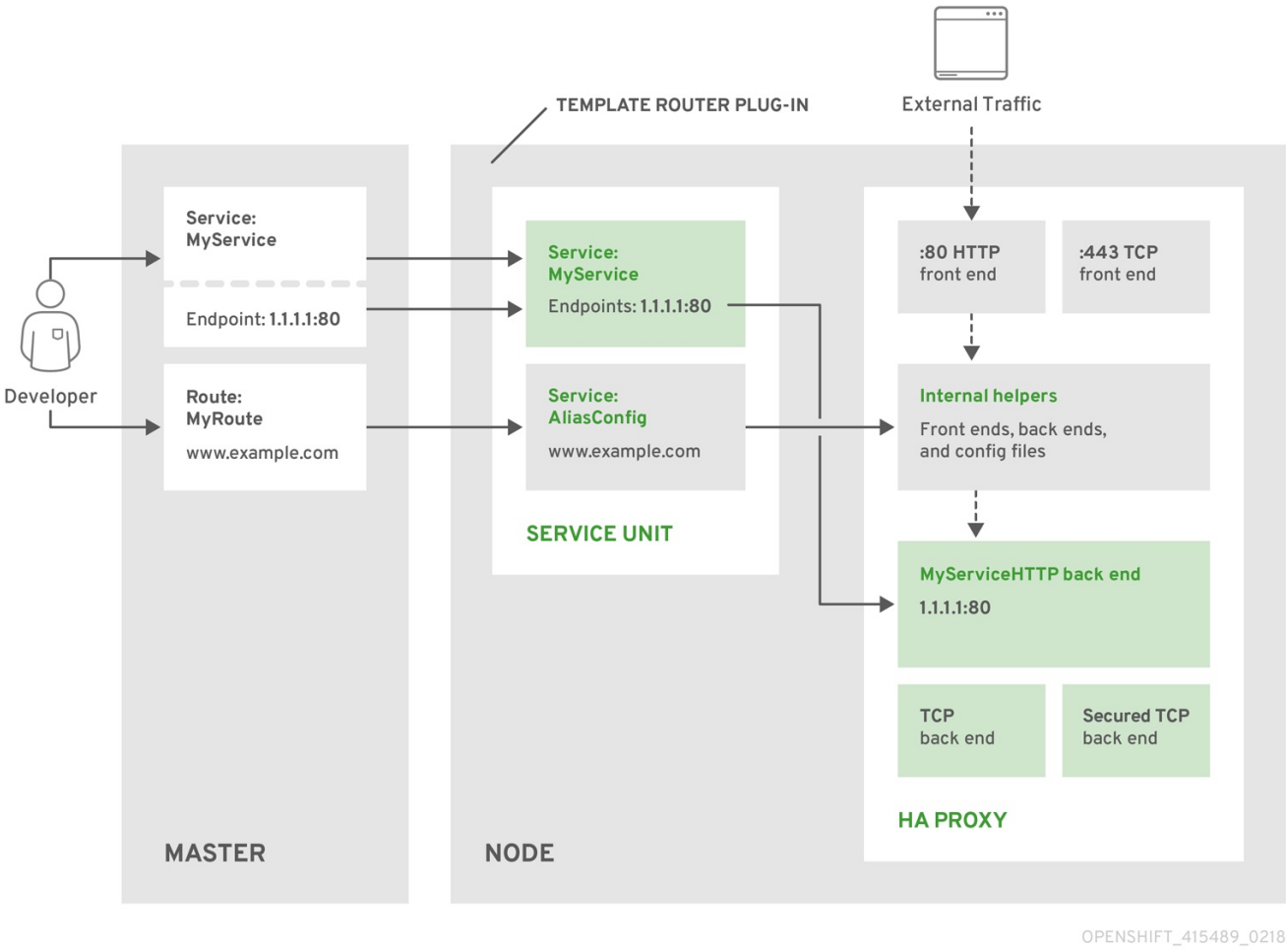
控制器和 HAProxy 存储在由部署配置管理的 pod 中。**oc adm router** 命令自动设置路由器的过程。

控制器监控路由和端点是否有变化，以及 HAProxy 的健康状况。当检测到更改时，它会构建一个新的 haproxy-config 文件并重启 HAProxy。haproxy-config 文件基于路由器的模板文件和 OpenShift Container Platform 的信息构建。

HAProxy 模板文件可以根据需要自定义，以支持 OpenShift Container Platform 当前不支持的功能。HAProxy 手册 描述了 HAProxy 支持的所有功能。

下图显示了如何通过插件处理 master 的数据，最后进入 HAProxy 配置中：

图 5.4. HAProxy 路由器数据流



HAProxy 模板路由器指标

HAProxy 路由器以 Prometheus 格式 公开或发布指标，供外部指标收集和聚合系统（如 Prometheus, statsd）消耗。路由器可以被配置为提供 HAProxy CSV 格式 指标，或者根本不提供路由器指标。

指标从路由器控制器以及 HAProxy 每 5 秒从 HAProxy 收集。当路由器部署并随着时间增加时，路由器指标计数器从零开始。每次重新加载 haproxy 时，HAProxy 指标计数器都会重新设置为零。路由器收集每个 frontend、后端和服务器的 HAProxy 统计信息。为减少资源使用超过 500 台服务器时，将报告后端而非服务器，因为后端可以有多个服务器。

统计信息是 可用 HAProxy 统计 的子集。

以下 HAProxy 指标会定期收集，并转换为 Prometheus 格式。对于每个前端，都会收集"F"计数器。当为每个后端收集计数器并为每个服务器收集"S"服务器计数器时。否则，会为每个后端收集"B"计数器，并且不收集服务器计数器。

如需更多信息，请参阅 路由器环境变量。

在下表中：

栏 1 - HAProxy CSV 统计中的索引

列 2

F	前端指标

b	如果因为 Server Threshold 没有显示服务器指标时的后端指标，
B	显示服务器指标时的后端指标
S	服务器指标.

列 3 - 计数器

列 4 - Counter description

索引	使用方法	计数	描述
2	bBS	current_queue	当前未分配给任何服务器的已排队请求数量。
4	FbS	current_sessions	当前活跃会话数量。
5	FbS	max_sessions	观察到的活跃会话数量上限。
7	FbBS	connections_total	连接总数。
8	FbS	bytes_in_total	当前传入字节总数.
9	FbS	bytes_out_total	传出字节的当前总数。
13	bS	connection_errors_total	连接错误总数。
14	bS	response_errors_total	响应错误总数。
17	bBS	up	后端的当前状态(1 = UP，0 = DOWN)。
21	S	check_failures_total	失败健康检查的总数。
24	S	downtime_seconds_total	总停机时间（以秒为单位）, nil）,
33	FbS	current_session_rate	当前每秒的会话数量已超过秒。
35	FbS	max_session_rate	每秒观察到的会话数量上限。
40	FbS	http_responses_total	HTTP 响应总数, 代码 2xx

43	FbS	http_responses_total	HTTP 响应总数, 代码 5xx
60	bS	http_average_response_latency_milliseconds	最新的 1024 请求（以毫秒为单位）。

路由器控制器提取下列项目：它们只能通过 Prometheus 格式指标使用。

名称	描述
template_router_reload_seconds	测量重新加载路由器的时间（以秒为单位）。
template_router_write_config_seconds	测量将路由器配置写入磁盘的时间（以秒为单位）。
haproxy_exporter_up	是 haproxy 成功的最后一个提取。
haproxy_exporter_csv_parse_failures	解析 CSV 时的错误数。
haproxy_exporter_scrape_interval	允许其他提取前的时间（以秒为单位）与数据大小成比例。
haproxy_exporter_server_threshold	跟踪的服务器数量和当前的阈值。
haproxy_exporter_total_scrapes	当前的 HAProxy 提取总数。
http_request_duration_microseconds	HTTP 请求延迟（微秒）。
http_request_size_bytes	HTTP 请求大小（以字节为单位）。
http_response_size_bytes	HTTP 响应大小（以字节为单位）。
openshift_build_info	带有由 OpenShift 构建的 main、min、git commit & git version 标签的恒定"1"值的指标。
ssh_tunnel_open_count	SSH 隧道总数打开的尝试计数器
ssh_tunnel_open_fail_count	SSH 隧道失败的打开尝试计数器

5.5. 端口转发

5.5.1. 概述

OpenShift Container Platform 利用内置到 Kubernetes 的功能 来支持转发到 pod 的端口转发。这通过使用 HTTP 和多路流协议（如 SPDY 或 HTTP/2）来实施。

开发人员可以使用 CLI 端口转发至容器集。CLI 侦听用户指定的本地端口，并通过上述协议进行转发。

5.5.2. 服务器操作

Kubelet 处理来自客户端的端口转发请求。在收到请求后，它会升级响应并等待客户端创建端口转发流。当它收到新流时，它会在流和 pod 端口之间复制数据。

从架构上看，有不同的选项可用于转发到 pod 端口。OpenShift Container Platform 中支持的实现当前直接调用节点主机上的 **nsenter** 来进入 pod 的网络命名空间，然后调用 **socat** 在流和 pod 端口之间复制数据。但是，自定义实现可能会包括运行一个 "helper" pod，然后运行 **nsenter** 和 **socat**，因此不需要在主机上安装这些二进制文件。

5.6. 远程命令

5.6.1. 概述

OpenShift Container Platform 利用 Kubernetes 中内置的功能来支持在容器中执行命令。这通过使用 HTTP 和多路流协议（如 SPDY 或 HTTP/2）来实施。

开发人员可以使用 CLI 在容器中执行远程命令。

5.6.2. 服务器操作

Kubelet 处理来自客户端的远程执行请求。在收到请求后，它会升级响应，评估请求标头来确定要接收哪些流（**stdin**、**stdout** 和/或 **stderr**）来预期接收并等待客户端创建流。

在 Kubelet 收到所有流后，它会执行容器中的命令，根据情况在流和命令 stdin、stdout 和 stderr 之间复制。当命令终止时，Kubelet 关闭升级的连接以及底层的连接。

从架构上讲，容器中运行命令有不同的选项。在 OpenShift Container Platform 中当前支持的实现人员直接在节点主机上调用 **nsenter**，以在执行 命令前输入容器的命名空间。但是，自定义实现可能会包括使用 **docker exec**，或运行一个 "helper" 容器，然后运行 **nsenter**，以便 **nsenter** 不是必须在主机上安装的所需二进制文件。

5.7. ROUTES

5.7.1. 概述

OpenShift Container Platform 路由以主机名（如 *www.example.com* ）公开一个服务，以便外部客户端可根据名称访问该服务。

主机名的 DNS 解析是独立于路由处理的。您的管理员可能已配置了 DNS 通配符条目，该条目 将解析到运行 OpenShift Container Platform 路由器的 OpenShift Container Platform 节点。如果您使用不同的主机名，您可能需要单独修改其 DNS 记录，以解析为运行路由器的节点。

每个路由都由一个名称（最多 63 个字符）、服务选择器和可选安全配置组成。

5.7.2. 路由器

OpenShift Container Platform 管理员可以将 *路由器* 部署到 OpenShift Container Platform 集群中的节点，它 允许开发人员创建的路由 供外部客户端使用。OpenShift Container Platform 中的路由层是可插拔的，默认提供了几个路由器插件。

注意

有关配置路由器的详情，请参阅配置集群 指南。

路由器使用服务选择器来查找服务，以及支持该服务的端点。当路由器和服务提供负载均衡时，OpenShift Container Platform 会使用路由器负载均衡。路由器在其服务的 IP 地址中检测到相关的更改，并相应地调整其配置。这可用于自定义路由器，将 API 对象的修改传递给外部路由解决方案。

请求的路径以 DNS 解析到一个或多个路由器的主机名开始。建议的方法定义一个指向多个路由器实例支持的一个或多个虚拟 IP(VIP)地址的通配符 DNS 条目的云域。使用云域之外的名称和地址的路由需要配置单独的 DNS 条目。

当 VIP 地址少于路由器时，与地址数量对应的路由器 *处于活跃状态*，其余是 *被动* 的。被动路由器也被称为 *热子路由器*。例如，有两个 VIP 地址和三个路由器，您有一个 "active-active-passive" 配置。有关路由器 VIP 配置的更多信息，请参阅 高可用性。

路由可以在 一组 路由器间分片。管理员可以基于集群范围设置分片，用户可在其项目中为命名空间设置分片。分片(sharding)使得操作员能够定义多个路由器组。组中的每个路由器仅提供流量子集。

OpenShift Container Platform 路由器通过协议提供外部主机名映射和服务的负载平衡，这些协议可直接区分信息到路由器；路由器必须存在于协议中以确定要发送的位置。

路由器插件假设它们默认可以绑定到主机端口 80(HTTP)和 443(HTTPS)。这意味着路由器必须放在不其他使用这些端口的节点上。或者，可通过设置 ROUTER_SERVICE_**HTTP_PORT 和 ROUTER_SERVICE_HTTPS_PORT** 环境变量将路由器配置为侦听其他端口。

由于路由器绑定到主机节点上的端口，因此如果路由器使用主机网络（默认值），则每个节点上仅能侦听这些端口的路由器。配置了集群网络，以便所有路由器可以访问集群中的所有 pod。

路由器支持以下协议：

- HTTP
- HTTPS（使用 SNI）
- Websockets
- 使用 SNI 的 TLS

注意

Websocket 流量使用相同的路由惯例，并且支持与其他流量相同的 TLS 终止类型。

要建立安全连接，必须协商与客户端和服务器通用的 密码。随着时间推移，新的安全的密码将变得可用，并集成到客户端软件中。当旧的客户端已过时时，可以丢弃旧的安全密码。默认情况下，路由器支持广泛的可用客户端。路由器可以配置为使用支持所需客户端的所选密码集合，不包括不太安全的密码。

5.7.2.1. 模板路由器

*模板路由器*是一个路由器类型，它为底层路由器实施提供某些基础架构信息，例如：

- 监视端点和路由的打包程序。
- 端点和路由数据，这些数据被保存到可使用的形式。
- 将内部状态传递给可配置的模板并执行模板。

- 调用重新加载脚本.

5.7.3. 可用路由器插件

如需 已验证的可用路由器插件，请参阅可用的路由器插件部分。

部署路由器时提供了有关部署这些 路由器的说明。

5.7.4. 粘性会话

实施粘性会话取决于底层路由器配置。默认 HAProxy 模板使用 **balance source** 指令实施粘性会话，该指令根据源 IP 平衡。另外，模板路由器插件为底层实施提供服务名称和命名空间。这可用于更高级的配置，例如实施在同级服务器集合之间同步的记忆程序。

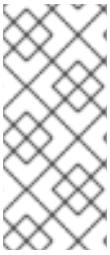
粘性会话可确保来自用户会话的所有流量都进入同一 pod，从而创建更好的用户体验。虽然满足用户的请求，但 pod 会缓存数据，它们可用于后续请求。例如，对于具有五个后端 pod 和两个负载均衡的路由器的集群，您可以确保同一 pod 从同一 Web 浏览器接收 Web 流量，而不考虑处理它的路由器。

虽然需要将路由流量返回到同一 pod，但无法保证它。但是，您可以使用 HTTP 标头设置 Cookie 来确定上一次连接中使用的 pod。当用户向应用发送另一请求时，浏览器会重新发送 Cookie 和路由器知道将流量发送到何处。

集群管理员可以关闭 passthrough 路由与其他连接分开的粘性，或者完全关闭粘性。

默认情况下，直通路由的粘性会话使用 **source** 负载均衡策略来实施。可以修改所有透传路由的默认，使用 **ROUTER_TCP_BALANCE_SCHEME**环境变量，对于独立的路由，使用 **haproxy.router.openshift.io/balance** 特定于路由的注解。

其他类型的路由默认使用 **leastconn** 负载均衡策略，可以使用 **ROUTER_LOAD_BALANCE_ALGORITHM** 环境变量 进行修改。可以使用 **haproxy.router.openshift.io/balance** 特定于路由的注解，可以修改单个路由。



注意

无法在 passthrough 路由上设置 Cookie，因为无法看到 HTTP 流量。相反，数字根据源 IP 地址计算，它决定后端。

如果后端有变化，则流量可能会头向错误的服务器，使其更小，而且当您使用负载均衡器（这会隐藏源 IP）时，则会为所有连接和流量发送到同一 pod 设置相同的数字。

另外，模板路由器插件为底层实施提供服务名称和命名空间。这可用于更高级的配置，例如实施在同级服务器集合之间同步的粘性程序。

此路由器实施的具体配置存储在路由器容器的 */var/lib/haproxy/conf* 目录中的 *haproxy-config.template* 文件中。文件 可以自定义。

注意

源 负载平衡策略 不区分外部客户端 IP 地址；因为 NAT 配置，原始 IP 地址（HAProxy 远程）是相同的。除非 HAProxy 路由器以 **hostNetwork: true** 运行，否则所有外部客户端都将路由到单个 pod。

5.7.5. 路由器环境变量

对于本节中列出的所有项目，您可以在路由器**部署配置**中设置环境变量以更改其配置，或使用 **oc set env** 命令：

```
$ oc set env <object_type>/<object_name> KEY1=VALUE1 KEY2=VALUE2
```

例如：

```
$ oc set env dc/router ROUTER_SYSLOG_ADDRESS=127.0.0.1 ROUTER_LOG_LEVEL=debug
```

表 5.2. 路由器环境变量

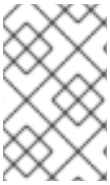
变量	默认	描述
DEFAULT_CERTIFICATE		用于不公开 TLS 服务器证书的路由的默认证书的内容；采用 PEM 格式。
DEFAULT_CERTIFICATE_DIR		包含名为 ***tls.crt*** 的文件的目录路径。如果 ***tls.crt*** 不是包含私钥的 PEM 文件，则首先会将名为 tls.key 的文件组合到同一目录中。然后会使用 PEM-format 内容作为默认证书。只有在没有指定 **DEFAULT_CERTIFICATE** 或 **DEFAULT_CERTIFICATE_PATH** 时使用。
DEFAULT_CERTIFICATE_PATH		用于不公开 TLS 服务器证书的路由的默认证书的路径；采用 PEM 格式。仅在没有指定 **DEFAULT_CERTIFICATE** 时使用。
EXTENDED_VALIDATION	**true**	如果为 **true**，路由器确认证书具有结构性是否正确。它不会针对任何 CA 验证证书。将 **false** 设置为关闭测试。
NAMESPACE_LABELS		要应用到命名空间的标签选择器来监视，为空表示所有。
PROJECT_LABELS		适用于项目的标签选择器来监视，如果为空则代表所有。
RELOAD_SCRIPT		reload 脚本的路径，用于重新加载路由器。
ROUTER_ALLOWED_DOMAINS		以逗号分隔的、路由中主机名只能从属于的域列表。可以使用域中的任何子域。选项 **ROUTER_DENIED_DOMAINS** 覆盖此选项中指定的任何值。如果设置，允许域之外的所有内容都将被拒绝。
ROUTER_BACKEND_PROCESS_ENDPOINTS		字符串，用于指定在使用模板函数 processEndpointsForAlias 时应如何处理端点。有效值为 ["shuffle", ""]。"shuffle" 将在每次调用时随机化元素。默认行为以预先确定的顺序返回。
ROUTER_BIND_PORTS_AFTER_SYNC	**false**	如果设置为 **true** 或 **TRUE**，则路由器不会绑定到任何端口，直到它完全同步状态。如果没有设置为 'true' 或 'TRUE'，路由器将绑定到端口并立即开始处理请求，但可能存在没有加载的路由。

变量	默认	描述
ROUTER_COOKIE_NAME		指定 Cookie 名称来覆盖内部生成的默认名称。名称只能包含大写字母和小写字母、数字、"_" 和 "-"。默认为路由的内部密钥进行哈希处理。
ROUTER_COMPRESSION_MIME	"文本/html 文本/纯文本文本/css"	要压缩空格分开的 mime 类型列表。
ROUTER_DENIED_DOMAINS		在路由中主机名不能属于的、以逗号分隔的域列表。域中没有可以使用子域。覆盖选项 **ROUTER_ALLOWED_DOMAINS**。
ROUTER_ENABLE_COMPRESSION		如果为 **true** 或 **TRUE**，则在可能的情况下会压缩响应。
ROUTER_LISTEN_ADDR	0.0.0.0:1936	设置路由器指标的侦听地址。
ROUTER_LOG_LEVEL	warning	发送到 syslog 服务器的日志级别。
ROUTER_MAX_CONNECTIONS	20000	最大并发连接数。
ROUTER_METRICS_HAPROXY_SERVER_THRESHOLD	500	
ROUTER_METRICS_HAPROXY_EXPORTED		以 CSV 格式收集的指标。例如，**默认SelectedMetrics** = []int{2, 4, 5, 5, 7, 8, 9, 13, 14, 17, 21, 24, 33, 35, 40, 43, 60}
ROUTER_METRICS_HAPROXY_BASE_SCRAPE_INTERVAL	5s	
ROUTER_METRICS_HAPROXY_TIMEOUT	5s	
ROUTER_METRICS_TYPE	hapoxy	为 HAProxy 路由器生成指标（haproxy 是唯一支持的值）
ROUTER_OVERRIDE_DOMAINS		以逗号分隔的域名列表。如果路由的域名与路由中的主机匹配，则主机名将被忽略，并且使用 **ROUTER_SUBDOMAIN** 中定义的模式。

变量	默认	描述
ROUTER_OVERRIDE_HOSTNAME		如果设置 **true**，请使用 **ROUTER_SUBDOMAIN** 中的模板覆盖路由的 spec.host 值。
ROUTER_SERVICE_HTTPS_PORT	443	用于侦听 HTTPS 请求的端口。
ROUTER_SERVICE_HTTP_PORT	80	用于侦听 HTTP 请求的端口。
ROUTER_SERVICE_NAME	public	路由器在路由状态中标识自己的名称。
ROUTER_CANONICAL_HOSTNAME		（可选）路由状态中显示的路由器的主机名。
ROUTER_SERVICE_NAMESPACE		路由器在路由状态中标识自己的命名空间。如果使用 **ROUTER_SERVICE_NAME**，则需要此项。
ROUTER_SERVICE_NO_SNI_PORT	10443	用于后端通信的一些前端的内部端口（请参阅以下注释）。
ROUTER_SERVICE_SNI_PORT	10444	用于后端通信的一些前端的内部端口（请参阅以下注释）。
ROUTER_SUBDOMAIN		应该用来在没有 spec.host （例如 ${name}-${namespace}.myapps.myapps.mycompany.com）为路由生成主机名的模板。
ROUTER_SYSLOG_ADDRESS		发送日志消息的地址。如果为空，则禁用。
ROUTER_SYSLOG_FORMAT		如果设置，请覆盖底层路由器实施使用的默认日志格式。其值应符合底层路由器实施的规范。
ROUTER_TCP_BALANCE_SCHEME	source	负载平衡策略. 用于多个端点，用于直通路由。可用选项包括 **source**、**roundrobin** 或 **leastconn**。
ROUTER_THREADS		指定 haproxy 路由器的线程数量。

变量	默认	描述
ROUTER_LOAD_BALANCE_ALGORITHM	leastconn	负载平衡策略。用于使用多个端点的路由。可用选项包括 **source**、**roundrobin** 和 **leastconn**。
ROUTE_LABELS		要监视的路由的标签选择器为空。
STATS_PASSWORD		访问路由器统计（如果路由器实施支持）需要的密码。
STATS_PORT		用于公开统计信息的端口（如果路由器实施支持）。如果没有设置，则 stats 不会被公开。
STATS_USERNAME		访问路由器统计（如果路由器实施支持）所需要的用户名。
TEMPLATE_FILE	**/var/lib/haproxy/conf/custom/haproxy-config-custom.template**	HAProxy 模板文件的路径（在容器镜像中）。
ROUTER_USE_PROXY_PROTOCOL		当设置为 **true** 或 **TRUE** 时，HAProxy 需要在端口 80 或端口 443 上使用 **PROXY** 协议。如果负载均衡器支持协议，源 IP 地址可以通过负载均衡器传递，如 Amazon ELB。
ROUTER_ALLOW_WILDCARD_ROUTES		当设置为 **true** 或 **TRUE** 时，任何 **通过路由器** 准入检查的通配符策略的路由都会被 HAProxy 路由器提供服务。
ROUTER_DISABLE_NAMESPACE_OWNERSHIP_CHECK		设置为 **true** 以放松命名空间所有权策略。
ROUTER_STRICT_SNI		strict-sni
ROUTER_CIPHERS	intermediate	指定绑定 支持的密码 集合。
ROUTER_HAPROXY_CONFIG_MANAGER		当设置为 **true** 或 **TRUE** 时，启用 HAproxy 的动态配置管理器，该管理器可以管理特定类型的路由，并减少 HAproxy 路由器重新加载的数量。如需更多信息，请参阅使用 动态配置管理器。

变量	默认	描述
COMMIT_INTERVAL	3600	指定提交使用动态配置管理器所做的更改的频率。这会导致底层模板路由器实施重新载入配置。
ROUTER_BLUEPRINT_ROUTE_NAMESPACE		设置为包含作为动态配置管理器蓝图的路由的命名空间。这允许动态配置管理器支持带有任何自定义注解、证书或配置文件的自定义路由。
ROUTER_BLUEPRINT_ROUTE_LABELS		设置为标签选择器，以应用到蓝图路由命名空间中的路由。这可让您指定可用作动态配置管理器蓝图的命名空间中的路由。
ROUTER_BLUEPRINT_ROUTE_POOL_SIZE	10	指定由动态配置管理器管理的每个路由蓝图的预分配池的大小。这可以通过在任何蓝图路由中使用 **router.openshift.io/pool-size** 注解来基于单独的路由覆盖。
ROUTER_MAX_DYNAMIC_SERVERS	5	指定添加到每个路由的最大动态服务器数量，供动态配置管理器使用。



注意

如果要在同一机器上运行多个路由器，您必须更改路由器侦听的端口，**ROUTER_SERVICE_SNI_PORT** 和 **ROUTER_SERVICE_NO_SNI_PORT**。只要在机器上是唯一的，这些端口就可能是您想要的任何端口。这些端口不会在外部公开。

路由器超时变量

TimeUnits 由一个数字及一个时间单位表示：**us** *(microseconds), **ms**（毫秒，默认）、**s**（秒）、**m**（分钟）、**h** *(小时)、**d** （天）。

正则表达式为： [1-9][0-9]*(us\|ms\|s\|m\|h\|d)

ROUTER_BACKEND_CHECK_INTERVAL	5000ms	后端上后续存活度检查之间的时间长度。
ROUTER_CLIENT_FIN_TIMEOUT	1s	控制连接到路由的客户端的 TCP FIN 超时周期。如果给定时间内没有回答关闭连接的 FIN，则 HAProxy 将关闭连接。如果设置为较低值，并且在路由器上使用较少的资源，则这不会产生任何损害。
ROUTER_DEFAULT_CLIENT_TIMEOUT	30s	客户端必须确认或发送数据的时长。
ROUTER_DEFAULT_CONNECT_TIMEOUT	5s	连接最大时间。

ROUTER_DEFAULT_SERVER_FIN_TIMEOUT	1s	控制路由器到支持路由的 pod 的 TCP FIN 超时。
ROUTER_DEFAULT_SERVER_TIMEOUT	30s	服务器必须确认或发送数据的时长。
ROUTER_DEFAULT_TUNNEL_TIMEOUT	1h	TCP 或 WebSocket 连接保持打开的时长。如果您有 websocket/tcp 连接（以及重新加载 HAProxy 的时间），旧的 HAProxy 进程将保留在该期间。
ROUTER_SLOWLORIS_HTTP_KEEPALIVE	300s	设置等待出现新 HTTP 请求的最长时间。如果设置得太低，可能会导致浏览器和应用程序无法期望较小的 **keepalive** 值。附加性. 有关更多信息，请参见下面的框。
ROUTER_SLOWLORIS_TIMEOUT	10s	HTTP 请求传输可以花费的时间长度。
RELOAD_INTERVAL	5s	路由器允许重新加载的最短频率来接受新的更改。
ROUTER_METRICS_HAPROXY_TIMEOUT	5s	收集 HAProxy 指标的超时时间。

注意

某些有效的超时值可以是某些变量的总和，而不是特定的预期超时。

例如：**ROUTER_SLOWLORIS_HTTP_KEEPALIVE** 调整 **超时 http-keep-alive**，并且默认设置为 **300s**，但 haproxy 还会等待 **tcp-request inspect-delay**，它设置为 **5s**。在这种情况下，整个超时时间将是 **300s** 加 **5s**。

5.7.6. 负载均衡策略

当路由具有多个端点时，HAProxy 根据所选的负载均衡策略将请求分发到端点之间的路由。当没有可用的持久性信息时，会出现这种情况，如会话中的第一个请求。

策略可以是以下之一：

- **roundrobin**:每个端点依次使用，取决于它的权重。当服务器的处理时间保持平等时，这是最平滑和最公平的算法。
- **leastconn**:连接数量最低的端点接收请求。当多个端点具有相同最少的连接数时，将执行循环循环。期望较长的会话时使用此算法，如 LDAP、SQL、TSE 或其他算法。不可与通常使用短会话（如 HTTP）的协议一起使用。
- **源** ：源 IP 地址的散列化，并划分到运行的服务器的总权重，以指定为哪些服务器接收请求。这可确保同一客户端 IP 地址始终到达同一服务器，只要没有服务器停机或启动。如果由于运行的服务器数量而变化的哈希结果，多个客户端将定向到不同的服务器。这个算法通常与 passthrough

路由一起使用。

ROUTER_TCP_BALANCE_SCHEME 环境变量 设置 passthrough 路由的默认策略。**ROUTER_LOAD_BALANCE_ALGORITHM** 环境变量 为剩余的路由设置路由器的默认策略。路由特定的注解 (**haproxy.router.openshift.io/balance**)可用于控制特定的路由。

5.7.7. HAProxy Strict SNI

默认情况下，当主机没有解析为 HTTPS 或 TLS SNI 请求的路由时，默认证书会作为 503 响应的一部分返回到调用者。这会公开默认证书，并可能会造成安全问题，因为网站提供了错误的证书。用于绑定的 HAProxy **strict-sni** 选项会阻止使用默认证书。

ROUTER_STRICT_SNI 环境变量控制绑定处理。当设置为 **true** 或 **TRUE** 时，在 HAProxy 绑定中添加 **strict-sni**。默认设置为 **false**。

可以在路由器创建或稍后添加时设置 选项。

```
$ oc adm router --strict-sni
```

This sets **ROUTER_STRICT_SNI=true**.

5.7.8. 路由器加密套件

每个客户端（如 Chrome 30 或 Java8）都包括用来安全地连接到路由器的密码套件。路由器必须至少具有一个加密系统以使连接可以完成：

表 5.3. 路由器类别配置文件

profile	最旧的兼容客户端
Modern	Firefox 27, Chrome 30, Windows 7, IE 11, Edge, Opera 17, Safari 9, Android 5.0, Java 8
intermediate	Firefox 1, Chrome 1, IE 7, Opera 5, Safari 1, Windows XP IE8, Android 2.3, Java 7
old	Windows XP IE6, Java 6

如需更多信息，请参阅 Security/Server Side TLS 参考指南。

默认情况下，路由器选择中间配置集并根据这个配置集设置密码。选择了配置集时，只设定密码。TLS 版本不受配置集的管控。

在创建路由器时，您可以使用 **--ciphers** 选项选择不同的配置集。对于一个已存在的路由器，则可以使用 **modern**, **intermediate**, 或 **old** 值更改 **ROUTER_CIPHERS** 环境变量。或者，可以提供一组":"分隔的密码。密码必须来自以下集合：

```
openssl ciphers
```

5.7.9. 路由主机名

为了在外部公开服务，OpenShift Container Platform 路由允许您将服务与外部可访问的主机名相关联。然后，可以使用这个边缘主机名将流量路由到该服务。

当来自不同命名空间的多个路由声明同一主机时，最旧的路由会胜出并声明命名空间。如果同一命名空间中定义了具有不同路径字段的其他路由，则会添加这些路径。如果使用相同路径的多个路由，则最旧的采用优先级。

造成这个问题的一个后果是，如果您主机名有两个路由：较早的路由以及较新的名称。如果其他人有指向您在创建其他两个路由时创建的同一主机名的路由，那么当您删除旧的路由时，您的声明将不再有效。现在，其他命名空间会声明主机名，并且您的声明会丢失。

具有指定主机的路由：

```
apiVersion: v1
kind: Route
metadata:
  name: host-route
spec:
  host: www.example.com (1)
  to:
    kind: Service
    name: service-name
```

(1) 指定用于公开服务的外部可访问主机名。

没有主机的路由：

```
apiVersion: v1
kind: Route
metadata:
  name: no-route-hostname
spec:
  to:
    kind: Service
    name: service-name
```

如果没有提供主机名作为路由定义的一部分，OpenShift Container Platform 会自动为您生成主机名。生成的主机名的格式如下：

```
<route-name>[-<namespace>].<suffix>
```

以下示例显示了 OpenShift Container Platform 生成的主机名，用于上述路由配置的主机名，而无需将主机添加到命名空间 **mynamespace** 中：

生成的主机名

```
no-route-hostname-mynamespace.router.default.svc.cluster.local (1)
```

(1) 生成的主机名后缀是默认路由子域 **router.default.svc.cluster.local**。

集群管理员也可以 自定义用作其环境的默认路由子域的后缀。

5.7.10. 路由类型

路由可以被保护或不受保护。安全路由提供以下几种 TLS 终止功能来为客户端提供证书。路由器支持 edge, passthrough, and re-encryption 终止。

取消安全的路由对象 YAML 定义

```
apiVersion: v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name
```

无安全的路由是最简单的配置，因为它们不需要密钥或证书，但安全路由为保持私有的连接提供了安全性。

安全路由是指定路由的 TLS 终止的路由。下文介绍了可用的终止 类型。

5.7.10.1. 基于路径的路由

基于路径的路由指定一个路径组件，可以与 URL 进行比较（要求基于 HTTP 的流量），这样便可使用相同的主机名来提供多个路由，每个路由都有不同的路径。路由器应该匹配基于最具体路径的路由；但是，这取决于路由器的实施。主机名和路径被传递给后端服务器，以便它可以成功回答它们的请求。例如：访问路由器的 http://example.com/foo/ 的请求会导致 pod 看到对 http://example.com/foo/ 的请求。

下表显示了路由及其可访问性示例：

表 5.4. 路由可用性

Route	当比较到	可访问
www.example.com/test	*www.example.com/test*	是
	www.example.com	否
www.example.com/test 和 *www.example.com*	*www.example.com/test*	是
	www.example.com	是
www.example.com	*www.example.com/test*	yes（由主机匹配，而不是路由）
	www.example.com	是

带有路径的未安全路由：

```
apiVersion: v1
kind: Route
metadata:
  name: route-unsecured
spec:
```

```
host: www.example.com
path: "/test" (1)
to:
  kind: Service
  name: service-name
```

(1) 该路径是基于路径的路由的唯一添加属性。

注意

使用 passthrough TLS 时，基于路径的路由不可用，因为路由器不会在这种情况下终止 TLS，且无法读取请求的内容。

5.7.10.2. 安全路由

安全路由指定路由的 TLS 终止，并可选择性地提供密钥和证书。

注意

OpenShift Container Platform 中的 TLS 终止依赖于 SNI 来提供自定义证书。任何端口 443 上接收的非SNI 流量均使用 TLS 终止和默认证书处理（这可能与请求的主机名不匹配，从而导致验证错误）。

安全路由可以使用以下三种安全 TLS 终止类型中的任何一种：

边缘终止

使用边缘终止时，TLS 终止在将流量代理到目的地之前发生在路由器上。TLS 证书由路由器的前端提供，因此它们必须配置为路由，否则将 路由器的默认证书 用于 TLS 终止。

使用 Edge 终止的安全路由

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured (1)
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name (2)
  tls:
    termination: edge (3)
    key: |- (4)
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |- (5)
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |- (6)
```

```
    -----BEGIN CERTIFICATE-----
    [...]
    -----END CERTIFICATE-----
```

1 2 对象的名称，长度限于 63 个字符。

3 **termination** 字段是边缘终止的边缘。

4 **key** 字段是 PEM 格式密钥文件的内容。

5 **certificate** 字段是 PEM 格式证书文件的内容。

6 可能需要一个可选的 CA 证书来建立用于验证的证书链。

由于 TLS 在路由器终止，因此不会加密从路由器到端点的连接。

边缘终止的路由可以指定 **insecureEdgeTerminationPolicy**，它允许禁用或重定向在不安全的方案 (**HTTP**)上的流量。**insecureEdgeTerminationPolicy** 允许的值有：**无** 或为空（代表禁用），**允许** 或**重定向**。默认的 **insecureEdgeTerminationPolicy** 是禁用不安全方案中的流量。常见用例是允许通过安全方案提供内容，但通过不安全的方案提供资产（示例图像、样式表和 javascript）。

使用 Edge 终止的安全路由允许 HTTP 流量

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured-allow-insecure 1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name 2
  tls:
    termination:                edge 3
    insecureEdgeTerminationPolicy: Allow 4
    [ ... ]
```

1 2 对象的名称，长度限于 63 个字符。

3 **termination** 字段是边缘终止的边缘。

4 允许在不安全的方案 **HTTP** 发送的请求的不安全策略。

使用 Edge 终止的安全路由将 HTTP 流量重定向到 HTTPS

```
apiVersion: v1
kind: Route
metadata:
  name: route-edge-secured-redirect-insecure 1
spec:
  host: www.example.com
  to:
    kind: Service
```

```
    name: service-name 2
  tls:
    termination:                edge      3
    insecureEdgeTerminationPolicy: Redirect 4
    [ ... ]
```

1 2 对象的名称，长度限于 63 个字符。

3 **termination** 字段是边缘终止的边缘。

4 将不安全方案 **HTTP** 发送的请求重定向到安全方案 **HTTPS** 的不安全策略。

passthrough 终止

如果 passthrough 终止，加密的流量会直接发送到目的地，而路由器不会提供 TLS 终止。因此不需要密钥或证书。

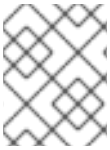
使用 Passthrough 终止的安全路由

```
apiVersion: v1
kind: Route
metadata:
  name: route-passthrough-secured 1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name 2
  tls:
    termination: passthrough      3
```

1 2 对象的名称，长度限于 63 个字符。

3 **termination** 字段设置为 **passthrough**。不需要其他加密字段。

目标 pod 负责为端点上的流量提供证书。这是目前唯一可以支持要求客户端证书的方法（也称为双向验证）。



注意

passthrough 路由也可以具有 **insecureEdgeTerminationPolicy**。唯一有效的值为 **None**（或为空，代表禁用） **或重定向**。

再加密终止

再加密是边缘终止的一种变体，即路由器通过证书终止 TLS，然后再加密它与端点的连接，这可能有不同的证书。因此，连接的完整路径已被加密，即使在内部网络上。路由器使用健康检查来确定主机的真实性。

使用 Re-Encrypt 终止的安全路由

```
apiVersion: v1
```

```
kind: Route
metadata:
  name: route-pt-secured (1)
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name (2)
  tls:
    termination: reencrypt (3)
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |- (4)
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

(1) (2) 对象的名称，长度限于 63 个字符。

(3) **termination** 字段设置为 **reencrypt**。其他字段如边缘终止中所示。

(4) re-encryption. **destinationCACertificate** 指定一个 CA 证书来验证端点证书，保护从路由器到目标 Pod 的连接。如果服务使用服务签名证书，或者管理员为路由器指定默认 CA 证书，且服务有由该 CA 签名的证书，则可以省略此字段。

如果 **destinationCACertificate** 字段留空，路由器会自动利用为服务用证书生成的证书颁发机构，并注入每个 pod 作为 **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt**。这允许使用端到端加密的新路由，而无需为路由生成证书。这适用于自定义路由器或 F5 路由器，这可能不允许 **destinationCACertificate**，除非管理员已允许它。

注意

重新加密路由可以有一个 **insecureEdgeTerminationPolicy**，其所有值都与 edge-terminated 路由相同。

5.7.11. 路由器分片

在 OpenShift Container Platform 中，每个路由都可以在其 **metadata** 字段中拥有任意数量的 标签。路由器使用 *选择器*（也称为 *选择表达式*）来选择要服务整个路由池的路由子集。选择表达式还涉及路由命名空间中的标签。所选路由形成 *路由器分片*。您可以独立于路由本身创建和修改路由器分片。

这种设计 *支持传统的* 分片，以及 *重叠的分片*。在传统的分片中，选择不会有重叠集，一个路由只属于一个分片。在重叠的分片中，选择会导致重叠集，路由可以属于多个不同的分片。例如，单个路由可以属于 **SLA=high** 分片（但不适用于 **SLA=medium** 或 **SLA=low** 分片），以及一个 **geo=west** 分片（而不是 **geo=east** 分片）。

重叠分片的另一个示例是根据路由的命名空间选择的一组路由器：

路由器	选择	命名空间
router-1	**A*** – **J***	**A***, B*, **C***, D*, **E***, **F***, **G***, **H***, **I***, **J***

路由器	选择	命名空间
router-2	**K* – T***	**K*、L*、M*、N*、O***、P*、Q*、**Q*、R*、S*、T***
router-3	**Q* – Z***	**Q*, R*, S*** , **T***, **U***, **V*** , **W*** , X*, **Y** *, **Z***

router-2 和 **router-3** 都提供位于命名空间 **Q***、**R***、**S***、**T*** 中的路由。要将这个示例从重叠到传统的分片，我们可以将 **router-2** 的选择改为 **K* – P***，从而消除重叠。

当路由器分片时，给定路由将绑定到组中的零个或多个路由器。路由绑定确保分片之间路由的唯一性。唯一性允许单个分片内存在同一路由的安全版本。这意味着，路由现在有一个可见的生命周期，从创建改为活跃的生命周期。

在分片环境中，第一个路由可以保留分片可以无限期地保留存在的权利，即使在重启时也是如此。

在绿色/蓝色部署期间，可以在多个路由器中选择路由。OpenShift Container Platform 应用程序管理员可能希望将流量从一个版本的应用程序迁移到另一个版本，然后关闭旧版本。

分片可以由集群管理员在集群级别以及项目/命名空间级别完成。当使用命名空间标签时，路由器的服务帐户必须具有 **cluster-reader** 权限，以允许路由器访问命名空间中的标签。

注意

对于声明相同主机名的两个或更多路由，解析顺序基于路由的期限，最旧的路由将胜出到该主机的声明。如果是分片的路由器，则根据与路由器选择标准匹配的标签选择路由。无法确定何时将标签添加到路由中。因此，如果声明现有主机名的旧路由为"re-labelled"以与路由器的选择条件匹配，则它将根据上述解析顺序替换现有的路由（最旧的路由胜出）。

5.7.12. 备用后端和 Weights

一个路由通常与一个服务相关联，使用 **to:** 令牌，带有 **kind:service**。对路由的所有请求都由服务中的端点根据 负载平衡策略 处理。

支持路由可以有多达四项服务。每个服务处理的请求部分由服务 **weight** 进行管理。

第一项服务使用之前使用 **to:** 令牌输入，而且可以使用 **alternateBackend:** 令牌输入 3 个额外服务。每个服务都必须是 **kind：service**，这是默认设置。

每一服务关联有一个 **权重**。服务处理的请求部分是 **weight / sum_of_all_weights**。当服务具有多个端点时，服务的权重会分布到端点中，每个端点至少达到 1。如果服务 **weight** 是 0，则服务端点将获得 0。

weight 必须介于 0-256 之间。默认值为 100。当 **weight** 为 0 时，服务不参与负载均衡，但继续为现有的持久连接服务。

使用 **alternateBackends** 也可以使用 **roundrobin** 负载均衡策略，确保请求根据 **weight** 按预期分发给服务。**roundrobin** 可使用路由 注解，或使用环境变量为路由设置路由。

以下是使用备用后端进行 A/B 部署的示例路由配置。

带有 alternateBackends 和 weight 的路由：

```
apiVersion: v1
kind: Route
```

```
metadata:
  name: route-alternate-service
  annotations:
    haproxy.router.openshift.io/balance: roundrobin 1
spec:
  host: www.example.com
  to:
    kind: Service
    name: service-name 2
    weight: 20 3
  alternateBackends:
  - kind: Service
    name: service-name2 4
    weight: 10 5
  - kind: Service
    name: service-name3 6
    weight: 10 7
```

1 此路由使用 **roundrobin** 负载均衡策略。

2 第一个服务名称是 **service-name**，它可能具有 0 个或更多 pod

4 6 alternateBackend 服务也可以有 0 个或更多 pod

3 5 7 **权重** 总数为 40。**service-name** 将获得请求的 20/40 或 1/2，**service-name2** 和 **service-name3** 将每个请求获得 1/4，假设每个服务都有 1 个或更多端点。

5.7.13. 特定于路由的注解

通过使用环境变量，路由器可以为它公开的所有路由设置默认选项。单个路由可以通过在其注解中提供特定配置来覆盖这些默认设置。

路由注解

对于本节中列出的所有项目，您可以在 **路由定义** 上设置注解，以便路由更改其配置

表 5.5. 路由注解

变量	描述	默认的环境变量
haproxy.router.openshift.io/balance	设置负载平衡算法。可用选项包括 **source**、**roundrobin** 和 **leastconn**。	passthrough 路由 使用 **ROUTER_TCP_BALANCE_SCHEME** 。否则，使用 **ROUTER_LOAD_BALANCE_algorithm**。
haproxy.router.openshift.io/disable_cookies	禁用使用 cookie 来跟踪相关连接。如果设置为 **true** 或 **TRUE**，则使用均衡算法来选择每个传入 HTTP 请求的后端服务连接。	

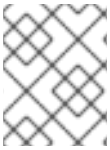
变量	描述	默认的环境变量
router.openshift.io/cookie_name	指定一个可选的、用于此路由的 cookie。名称只能包含大写字母和小写字母、数字、"_" 和 "-"。默认为路由的内部密钥进行哈希处理。	
haproxy.router.openshift.io/pod-concurrent-connections	设置路由器支持的 pod 允许的最大连接数。注意：如果存在多个 pod，则每个 pod 都可允许这里设置的连接数量。但是，如果有多个路由器，它们之间没有协调关系，每个路由器都可能会多次连接。如果没有设置，或者将其设定为 0，则没有限制。	
haproxy.router.openshift.io/rate-limit-connections	设置 **true** 或 **TRUE** 来启用速率限制功能。	
haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp	限制一个 IP 地址共享的并行 TCP 连接数。	
haproxy.router.openshift.io/rate-limit-connections.rate-http	限制 IP 地址可以发出 HTTP 请求的速率。	
haproxy.router.openshift.io/rate-limit-connections.rate-tcp	限制 IP 地址可以进行 TCP 连接的速率。	
haproxy.router.openshift.io/timeout	为路由设定服务器端超时。(TimeUnits)	**ROUTER_DEFAULT_SERVER_TIMEOUT**
router.openshift.io/haproxy.health.check.interval	为后端健康检查设定间隔。(TimeUnits)	**ROUTER_BACKEND_CHECK_INTERVAL**
haproxy.router.openshift.io/ip_whitelist	为路由设置 白名单。	
haproxy.router.openshift.io/hsts_header	为 edge terminated 或 re-encrypt 路由设置 Strict-Transport-Security 标头。	

变量	描述	默认的环境变量
router.openshift.io/cookie-same-site	设置一个值来限制 cookies。数值是： **Lax**：cookies 在访问的站点和第三方站点间进行传输。 **Strict**：cookies 仅限于访问的站点。 **None**：cookies 仅限于指定的站点。 这个值仅适用于重新加密和边缘路由。如需更多信息，请参阅 SameSite cookies 文档。	

设置自定义超时的路由

```
apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms (1)
[...]
```

(1) 使用 HAProxy 支持的单元（us、ms、s、m、h、d）指定新的超时时间。如果没有提供单位，ms 会被默认使用。



注意

如果为 passthrough 路由设置的服务器端的超时值太低，则会导致 WebSocket 连接在那个路由上经常出现超时的情况。

5.7.14. 特定于路由的 IP 白名单

您可以通过在路由中添加 **haproxy.router.openshift.io/ip_whitelist** 注解来限制对路由集合的访问。白名单是空格分隔的 IP 地址和/或 CIDR 列表，用于批准的源地址。来自白名单以外的 IP 地址的请求会被丢弃。

一些示例：

在编辑路由时，添加以下注解来定义所需的源 IP。或者，使用 **oc annotate route <name>**。

只允许一个特定的 IP 地址：

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10
```

允许多个 IP 地址：

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.10 192.168.1.11 192.168.1.12
```

允许 IP CIDR 网络：

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 192.168.1.0/24
```

允许混合 IP 地址和 IP CIDR 网络：

```
metadata:
  annotations:
    haproxy.router.openshift.io/ip_whitelist: 180.5.61.153 192.168.1.0/24 10.0.0.0/8
```

5.7.15. 创建路由指定通配符子域策略

通配符策略允许用户定义涵盖一个域内所有主机的路由（当路由器配置为允许时）。通过 **wildcardPolicy** 字段，路由可以指定通配符策略作为其配置的一部分。任何路由器使用策略运行，允许通配符路由将根据通配符策略正确公开路由。

了解如何配置 HAProxy 路由器以允许通配符路由。

Route 指定 Subdomain WildcardPolicy

```
apiVersion: v1
kind: Route
spec:
  host: wildcard.example.com (1)
  wildcardPolicy: Subdomain (2)
  to:
    kind: Service
    name: service-name
```

1. 指定用于公开服务的外部可访问主机名。
2. 指定外部访问的主机名应允许子域 **example.com** 中的所有主机。***.example.com** 是主机名 **wildcard.example.com** 的子域来访问公开的服务。

5.7.16. 路由状态

route status 字段仅由路由器设置。如果对路由进行了更改，以便路由器不再提供特定路由，则状态将变为过时的。路由器没有清除 **路由 status** 字段。要移除路由状态中的过时的条目，请使用 clear-route-status 脚本。

5.7.17. 在路由中拒绝或允许证书域

可以利用 **ROUTER_DENIED_DOMAINS** 和 **ROUTER_ALLOWED_DOMAINS** 环境变量，将路由器配置为拒绝或允许来自路由中的主机名的特定域子集。

ROUTER_DENIED_DOMAINS	任何指定路由中都不允许列出的域。
ROUTER_ALLOWED_DOMAINS	任何指定路由中只允许列出的域。

拒绝域列表中的域优先于允许的域列表。这意味着 OpenShift Container Platform 首先检查 deny 列表（如果适用），以及主机名不在被拒绝的域列表中，并检查允许的域列表。但是，允许的域列表更为严格，并确保路由器只接受含有属于该列表的主机的路由。

例如，要拒绝 **myrouter** 路由的 **[*.]open.header.test**, **[*.]openshift.org** 和 **[*.]block.it** 路由，请运行以下命令：

```
$ oc adm router myrouter ...
```

```
$ oc set env dc/myrouter ROUTER_DENIED_DOMAINS="open.header.test, openshift.org, block.it"
```

这意味着 **myrouter** 会根据路由的名称接受以下内容：

```
$ oc expose service/<name> --hostname="foo.header.test"
```

```
$ oc expose service/<name> --hostname="www.allow.it"
```

```
$ oc expose service/<name> --hostname="www.openshift.test"
```

但是，**myrouter** 将拒绝以下内容：

```
$ oc expose service/<name> --hostname="open.header.test"
```

```
$ oc expose service/<name> --hostname="www.open.header.test"
```

```
$ oc expose service/<name> --hostname="block.it"
```

```
$ oc expose service/<name> --hostname="franco.baresi.block.it"
```

```
$ oc expose service/<name> --hostname="openshift.org"
```

```
$ oc expose service/<name> --hostname="api.openshift.org"
```

另外，要阻止任何主机名*没有*设置为 **[*.]stickshift.org** 或 **[*.]kates.net** 的路由，请运行以下命令：

```
$ oc adm router myrouter ...
```

```
$ oc set env dc/myrouter ROUTER_ALLOWED_DOMAINS="stickshift.org, kates.net"
```

这意味着 **myrouter** 路由器将接受：

```
$ oc expose service/<name> --hostname="stickshift.org"
```

```
$ oc expose service/<name> --hostname="www.stickshift.org"
```

```
$ oc expose service/<name> --hostname="kates.net"
```

```
$ oc expose service/<name> --hostname="api.kates.net"
```

```
$ oc expose service/<name> --hostname="erno.r.kube.kates.net"
```

但是，**myrouter** 将拒绝以下内容：

```
$ oc expose service/<name> --hostname="www.open.header.test"
```

```
$ oc expose service/<name> --hostname="drive.ottomatic.org"
```

```
$ oc expose service/<name> --hostname="www.wayless.com"
```

```
$ oc expose service/<name> --hostname="www.deny.it"
```

要实现这两个情况，请运行以下命令：

```
$ oc adm router adrouter ...
```

```
$ oc set env dc/adrouter ROUTER_ALLOWED_DOMAINS="okd.io, kates.net" \
    ROUTER_DENIED_DOMAINS="ops.openshift.org, metrics.kates.net"
```

这将允许主机名设置为 **[*.]openshift.org** 或 **[*.]kates.net** 的任何路由，且不允许主机名设置为 **[*.]ops.openshift.org** 或 **[*.metrics.kates.net** 的任何路由。

因此，以下内容会被拒绝：

```
$ oc expose service/<name> --hostname="www.open.header.test"
```

```
$ oc expose service/<name> --hostname="ops.openshift.org"
```

```
$ oc expose service/<name> --hostname="log.ops.openshift.org"
```

```
$ oc expose service/<name> --hostname="www.block.it"
```

```
$ oc expose service/<name> --hostname="metrics.kates.net"
```

```
$ oc expose service/<name> --hostname="int.metrics.kates.net"
```

但是，允许以下内容：

```
$ oc expose service/<name> --hostname="openshift.org"
```

```
$ oc expose service/<name> --hostname="api.openshift.org"
```

```
$ oc expose service/<name> --hostname="m.api.openshift.org"
```

```
$ oc expose service/<name> --hostname="kates.net"
```

```
$ oc expose service/<name> --hostname="api.kates.net"
```

5.7.18. 支持 Kubernetes ingress 对象

Kubernetes ingress 对象是一个配置对象，决定入站连接如何到达内部服务。OpenShift Container Platform 支持使用入口控制器配置文件在这些对象。

此控制器监视入口对象，并创建一个或多个路由来满足入口对象的条件。控制器还负责保持入口对象和生成的路由对象同步。这包括为与 ingress 对象关联的 secret 提供生成的路由权限。

例如，将 ingress 对象配置为：

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: test
spec:
  rules:
  - host: test.com
    http:
     paths:
     - path: /test
       backend:
        serviceName: test-1
        servicePort: 80
```

生成以下路由对象：

```
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: test-a34th (1)
  ownerReferences:
  - apiVersion: extensions/v1beta1
    kind: Ingress
    name: test
    controller: true
spec:
  host: test.com
  path: /test
  to:
    name: test-1
  port:
     targetPort: 80
```

(1) 名称由路由对象生成，入口名称用作前缀。

注意

要创建路由，入口对象必须具有主机、服务和路径。

5.7.19. 禁用命名空间所有权检查

主机和子域由首先发出声明的路由的命名空间所有。在命名空间中创建的其他路由可以在子域上声明。所有其他命名空间都无法对声明的主机和子域进行声明。拥有主机的命名空间还拥有与主机关联的所有路径，例如 ***www.abc.xyz/path1***。

例如，如果主机 ***www.abc.xyz*** 没有被任何路由声明。在命名空间 **ns1** 中使用主机 ***www.abc.xyz*** 创建路由 **r1** 会使命名空间 **ns1** 的所有者为主机 ***www.abc.xyz***，对通配符路由的子域 **abc.xyz**。如果另一个命名空间 **ns2**，试图创建带有路径 ***www.abc.xyz/path1/path2*** 的路由，它将因为另一个命名空间中的路由（本例中为**ns1**） 而失败。

使用通配符路由 拥有子域的命名空间，该命名空间拥有子域中的所有主机。如果命名空间拥有以上示例中的子域 **abc.xyz**，另一个命名空间无法声明 **z.abc.xyz**。

禁用命名空间所有权规则，您可以禁用这些限制，并允许在命名空间间声明主机（和子域）。

警告

如果您决定在路由器中禁用命名空间所有权检查，请注意，最终用户可在命名空间间声明主机的所有权。虽然这种变化可以在某些开发环境中取用，但在生产环境中要谨慎使用此功能，并确保您的集群策略锁定了不受信任的最终用户创建路由。

例如，如果命名空间 **ns1** 创建最旧的路由 **r1** ***www.abc.xyz***，它只具有 **ROUTER_DISABLE_NAMESPACE_OWNERSHIP_CHECK=true**。另一个命名空间可以创建一个通配符路由，即使该子域中没有最旧的路由（**一个bc.xyz**），我们也可能有其他命名空间声明其他非通配符重叠主机（例如 **foo.abc.xyz**、**bar.abc.xyz**、**baz.abc.xyz**）及其声明。

任何其它命名空间（例如 **ns**2）现在可以创建一个路由 **r2** ***www.abc.xyz/p1/p2***，它会被接受。同样，另一个命名空间(**ns3**)也可以创建一个带有子域通配符策略的路由通配符。abc **.xyz**，它可以拥有通配符。

在本示例中，策略 **ROUTER_DISABLE_NAMESPACE_OWNERSHIP_CHECK=true** 更为 lax，并允许在命名空间间声明。如果已声明 host+path，则路由器才会拒绝禁用命名空间所有权的路由。

例如，如果新路由 **rx** 尝试声明 ***www.abc.xyz/p1/p2***，它将被拒绝，因为路由 **r2** 拥有该主机+path 组合。无论路由 **rx** 存在于同一命名空间或其他命名空间中，都会声明正确的 host+path。

此功能可以在路由器创建期间或通过在路由器部署配置中设置环境变量来设置。

设置在路由器创建过程中设置

```
$ oc adm router ... --disable-namespace-ownership-check=true
```

在路由器部署配置中设置环境变量

```
$ oc set env dc/router ROUTER_DISABLE_NAMESPACE_OWNERSHIP_CHECK=true
```

[1] 这时，设备名称引用容器 B 主机上的设备。

第 6 章 SERVICE CATALOG 组件

6.1. SERVICE CATALOG

6.1.1. 概述

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

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

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

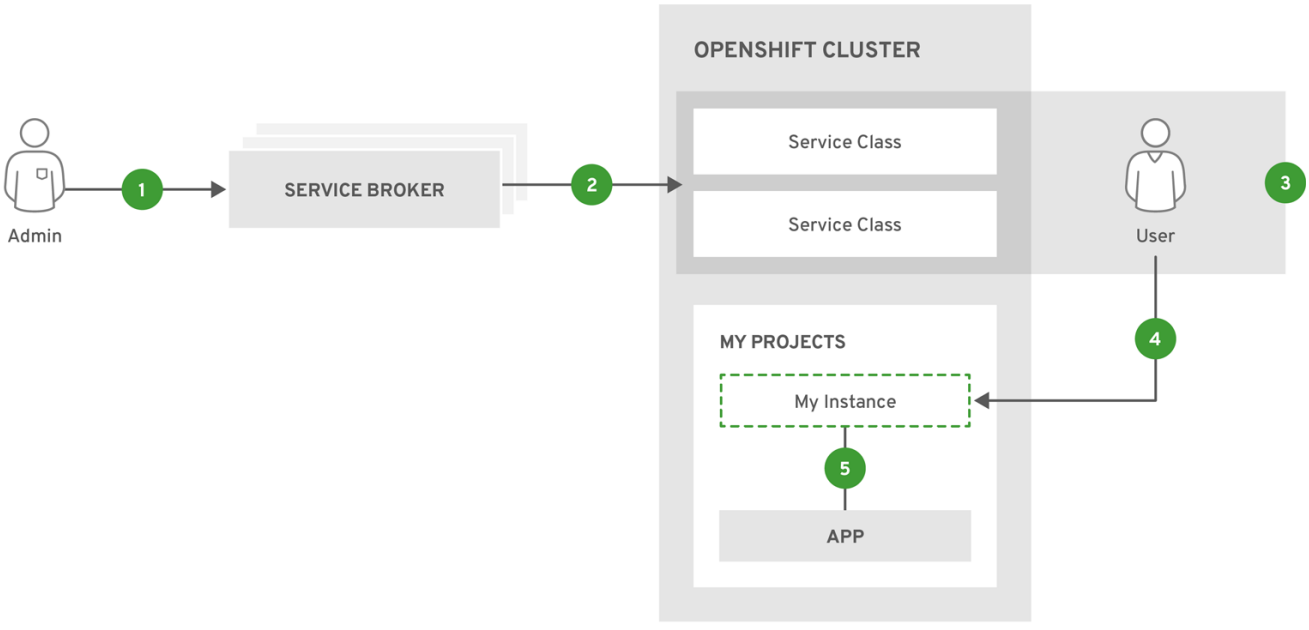
因此，服务用户可从使用不同供应商的不同类型的服务简易性和一致性中受益，而服务供应商则可得益于通过一个集成点来访问多个平台。

6.1.2. 设计

服务目录的设计遵循以下基本工作流：

注意

以下新的术语在概念和术语中进一步定义。



OPENSHIFT_415489_0218

集群管理员在其 OpenShift Container Platform 集群中注册一个或多个 *集群服务代理*。在安装过程中可以为一些默认的服务代理或手动完成。

每个服务代理指定一组 *集群服务类*和这些服务（*服务计划*）到 OpenShift Container Platform，它们应该提供给用户使用。

使用 OpenShift Container Platform Web 控制台或 CLI，用户可以发现可用的服务。例如，集群服务版本类可以是名为 BestDataBase 的数据库即服务。

用户选择集群服务类，并请求自己的 *新实例*。例如，服务实例可以是名为 **my_db** 的 BestDataBase 实例。

用户链接或将、其服务实例 *绑定到* 一组容器集（即应用）。例如，**my_db** 服务实例可以绑定到名为 **my_app** 的用户的应用。

当用户向置备或取消置备资源时，请求将向服务目录发出请求，然后向适当的集群服务代理发送请求。对于某些服务，应该花费一些时间完成 **置备、取消置备 和更新** 等操作。如果集群服务代理不可用，服务目录将继续重试操作。

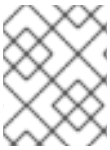
此基础架构允许在 OpenShift Container Platform 上运行的应用程序与它们使用的服务间进行松散耦合。这样，使用这些服务的应用程序可以专注于自己的业务逻辑，同时将这些服务的管理留给提供商。

6.1.2.1. 删除资源

当用户使用服务时（或者可能不再需要计费）时，可以删除服务实例。要删除服务实例，必须先删除服务绑定。删除服务绑定称为 *unbinding*。删除过程的一部分包括删除引用要删除的服务绑定的 secret。

删除所有服务绑定后，可以删除服务实例。删除服务实例称为 *取消置备*。

如果删除了包含服务绑定和服务实例的项目或命名空间，服务目录必须首先请求集群服务代理来删除关联的实例和绑定。这应该会延迟项目或命名空间的实际删除，因为服务目录必须与集群服务代理通信并等待它们执行取消置备工作。在正常情况下，这可能需要几分钟或更长时间，具体取决于该服务。



注意

如果删除部署所使用的服务绑定，还必须从部署中删除对绑定 secret 的任何引用。否则，下次推出部署将失败。

6.1.3. 概念和术语

Cluster Service Broker

集群服务版本代理 是一个符合 OSB API 规格的服务器，用于管理由一个或多个服务组成的集合。软件可以托管在您自己的 OpenShift Container Platform 集群中，或其他位置。
集群管理员可以创建代表集群服务版本代理的 **ClusterServiceBroker** API 资源，并将它们注册到 OpenShift Container Platform 集群。这允许集群管理员使用集群中可用的集群服务代理提供新类型的受管服务。

ClusterServiceBroker 资源指定集群服务代理的连接详情，以及这些服务组（以及这些服务的变体），供用户使用。特殊备注是 **authInfo** 部分，其中包含用于与集群服务代理进行身份验证的数据。

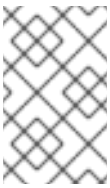
ClusterServiceBroker 资源示例

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceBroker
metadata:
  name: BestCompanySaaS
spec:
  url: http://bestdatabase.example.com
  authInfo:
    basic:
```

```
  secretRef:
    namespace: test-ns
    name: secret-name
```

Cluster Service Class

在服务目录的上下文中也与"service"同步，*集群服务类* 是由特定集群服务代理提供的托管服务类型。每次一个新集群服务版本代理资源被添加到集群中时，服务目录控制器会连接到对应的集群服务代理来获取服务产品列表。自动为每个 **ClusterServiceClass** 资源创建新的 ClusterServiceClass 资源。



注意

OpenShift Container Platform 还具有一个名为 services 的核心概念，它是与内部负载平衡相关的 Kubernetes 资源。这些资源不一定与服务目录和 OSB API 的上下文中使用术语混淆。

ClusterServiceClass 资源示例

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ClusterServiceClass
metadata:
  name: smallDB
  brokerName: BestDataBase
  plans: [...]
```

Cluster Service Plan

集群服务计划 代表集群服务类的层。例如，群集服务类可以公开一组提供不同服务质量(QoS)程度的计划，每个计划都有不同的成本。

服务实例

服务实例 是集群服务版本类的调配实例。当用户希望使用服务类提供的功能时，他们可以创建新的服务实例。
创建新的 **ServiceInstance** 资源时，服务目录控制器连接到适当的集群服务代理，并指示它置备服务实例。

ServiceInstance 资源示例

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceInstance
metadata:
  name: my_db
  namespace: test-ns
spec:
  externalClusterServiceClassName: smallDB
  externalClusterServicePlanName: default
```

Application（应用程序）

术语 *应用程序* 指的是 OpenShift Container Platform 部署工件（如在用户项目中运行的 pod），它将使用一个 *服务实例*。

凭证

凭据是 应用与服务实例通信所需的信息。

服务绑定

*服务绑定*是服务实例和应用程序间的链接。它们由希望应用程序引用和使用服务实例的集群用户创建。
创建后，服务目录控制器会创建一个 Kubernetes secret，其中包含服务实例的连接详情和凭证。这种 secret 通常会挂载到 pod 中。

ServiceBinding 资源示例

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: myBinding
  namespace: test-ns
spec:
  instanceRef:
    name: my_db
  parameters:
    securityLevel: confidential
  secretName: mySecret
```

注意

用户不应使用 Web 控制台更改实例化实例的环境变量的前缀，因为它可能导致应用程序路由无法访问。

参数

*参数是*一个特殊字段，可用于在使用服务绑定或服务实例时将其他数据传递给集群服务代理。唯一格式化要求是有效的 YAML（或 JSON）参数。在上例中，安全级别参数会传递给服务绑定请求中的集群服务代理。对于需要更多安全性的参数，将其放入机密并使用 **参数 From** 来引用它们。

Service Binding Resource 引用 Secret 示例

```
apiVersion: servicecatalog.k8s.io/v1beta1
kind: ServiceBinding
metadata:
  name: myBinding
  namespace: test-ns
spec:
  instanceRef:
    name: my_db
  parametersFrom:
    - secretKeyRef:
        name: securityLevel
        key: myKey
  secretName: mySecret
```

6.1.4. 提供的 Cluster Service Broker

OpenShift Container Platform 提供以下集群服务代理用于服务目录。

- Template Service Broker

- OpenShift Ansible Broker

6.2. 服务目录命令行界面(CLI)

6.2.1. 概述

与服务目录交互 的基本工作流 是：

- 集群管理员安装并注册代理服务器使其可用。
- 用户通过在 OpenShift 项目中实例化这些服务，并将这些服务实例链接到其容器集。

名为 **svcat** 的 Service Catalog 命令行界面(CLI)实用程序可用于处理这些用户相关的任务。虽然 **oc** 命令可以执行相同的任务，但您可以使用 **svcat** 更轻松地与服务目录资源交互。**svcat** 使用 OpenShift 集群上的聚合 API 端点与服务目录 API 通信。

6.2.2. 安装 svcat

如果您的红帽帐户中有活跃的 OpenShift Enterprise 订阅，您可以使用 Red Hat Subscription Management(RHSM)安装 **svcat** ：

```
# yum install atomic-enterprise-service-catalog-svcat
```

6.2.2.1. 云供应商的注意事项

Google Compute Engine for Google Cloud Platform，运行以下命令设置防火墙规则以允许传入的流量：

```
$ gcloud compute firewall-rules create allow-service-catalog-secure --allow tcp:30443 --description
"Allow incoming traffic on 30443 port."
```

6.2.3. 使用 svcat

本节包含用于处理 服务目录工作流中 列出的用户关联的任务的常用命令。使用 **svcat --help** 命令获取更多信息并查看其他可用的命令行选项。本节中的示例输出假定集群中已安装了 Ansible Service Broker。

6.2.3.1. 获取代理详情

您可以查看可用代理列表，同步代理目录，以及获取有关服务目录中部署的代理的详情。

6.2.3.1.1. 查找代理

查看集群中安装的所有代理：

```
$ svcat get brokers
```

输出示例

```
      NAME                                URL                                 STATUS
+-------------------------+-----------------------------------------------------------------------------+--
------+
```

```
  ansible-service-broker    https://asb.openshift-ansible-service-broker.svc:1338/ansible-service-broker
Ready
  template-service-broker   https://apiserver.openshift-template-service-
broker.svc:443/brokers/template.openshift.io   Ready
```

6.2.3.1.2. 同步代理目录

从代理中刷新目录元数据：

```
$ svcat sync broker ansible-service-broker
```

输出示例

```
Synchronization requested for broker: ansible-service-broker
```

6.2.3.1.3. 查看代理详情

查看代理详情：

```
$ svcat describe broker ansible-service-broker
```

输出示例

```
  Name:     ansible-service-broker
  URL:      https://openshift-automation-service-broker.openshift-automation-service-
broker.svc:1338/openshift-automation-service-broker/
  Status:   Ready - Successfully fetched catalog entries from broker @ 2018-06-07 00:32:59 +0000
UTC
```

6.2.3.2. 查看服务类和服务计划

当您创建 **ClusterServiceBroker** 资源时，服务目录控制器会查询代理服务器以查找它所提供的所有服务，并为这些服务创建服务类(**ClusterServiceClass**)。另外，它还为每个代理的服务创建服务计划(**ClusterServicePlan**)。

6.2.3.2.1. 查看服务类

查看可用的 ClusterServiceClass 资源：

```
$ svcat get classes
```

输出示例

```
      NAME                DESCRIPTION
+------------------+-------------------------------+
  rh-mediawiki-apb    Mediawiki apb implementation

  ...

  rh-mariadb-apb      Mariadb apb implementation
```

```
  rh-mysql-apb        Software Collections MySQL APB
  rh-postgresql-apb   SCL PostgreSQL apb
                      implementation
```

查看服务类的详情：

```
$ svcat describe class rh-postgresql-apb
```

输出示例

```
  Name:          rh-postgresql-apb
  Description:   SCL PostgreSQL apb implementation
  UUID:          d5915e05b253df421efe6e41fb6a66ba
  Status:        Active
  Tags:          database, postgresql
  Broker:        ansible-service-broker

 Plans:
  NAME            DESCRIPTION
+------+---------------------------------+
  prod   A single DB server with
         persistent storage
  dev    A single DB server with no
         storage
```

6.2.3.2.2. 查看服务计划

查看集群中可用的 ClusterServicePlan 资源：

```
$ svcat get plans
```

输出示例

```
   NAME          CLASS                 DESCRIPTION
+---------+------------------+--------------------------------+
  default   rh-mediawiki-apb    An APB that deploys MediaWiki

  ...

  prod      rh-mariadb-apb      This plan deploys a single
                                MariaDB instance with 10 GiB
                                of persistent storage
  dev       rh-mariadb-apb      This plan deploys a single
                                MariaDB instance with
                                ephemeral storage
  prod      rh-mysql-apb        A MySQL server with persistent
                                storage
  dev       rh-mysql-apb        A MySQL server with ephemeral
                                storage
  prod      rh-postgresql-apb   A single DB server with
                                persistent storage
  dev       rh-postgresql-apb   A single DB server with no
                                storage
```

查看计划详情：

```
$ svcat describe plan rh-postgresql-apb/dev
```

输出示例

```
  Name:          dev
  Description:   A single DB server with no storage
  UUID:          9783fc2e859f9179833a7dd003baa841
  Status:        Active
  Free:          true
  Class:         rh-postgresql-apb

Instances:
No instances defined

Instance Create Parameter Schema:
  $schema: http://json-schema.org/draft-04/schema
  additionalProperties: false
  properties:
    postgresql_database:
      default: admin
      pattern: ^[a-zA-Z_][a-zA-Z0-9_]*$
      title: PostgreSQL Database Name
      type: string
    postgresql_password:
      pattern: ^[a-zA-Z0-9_~!@#$%^&*()-=<>,.?;:|]+$
      title: PostgreSQL Password
      type: string
    postgresql_user:
      default: admin
      maxLength: 63
      pattern: ^[a-zA-Z_][a-zA-Z0-9_]*$
      title: PostgreSQL User
      type: string
    postgresql_version:
      default: "9.6"
      enum:
      - "9.6"
      - "9.5"
      - "9.4"
      title: PostgreSQL Version
      type: string
  required:
  - postgresql_database
  - postgresql_user
  - postgresql_password
  - postgresql_version
  type: object

Instance Update Parameter Schema:
  $schema: http://json-schema.org/draft-04/schema
  additionalProperties: false
  properties:
    postgresql_version:
```

```
      default: "9.6"
      enum:
      - "9.6"
      - "9.5"
      - "9.4"
      title: PostgreSQL Version
      type: string
    required:
    - postgresql_version
    type: object

  Binding Create Parameter Schema:
    $schema: http://json-schema.org/draft-04/schema
    additionalProperties: false
    type: object
```

6.2.3.3. 置备服务

调配意味着使服务可供使用。若要调配服务，您需要创建服务实例，然后绑定到该服务。

6.2.3.3.1. 创建 ServiceInstance

注意

服务实例必须在 OpenShift 命名空间中创建。

1. 创建新项目。

```
$ oc new-project <project-name> 1
```

1 将 **<project-name>** 替换为项目的名称。

2. 使用以下命令创建服务实例：

```
$ svcat provision postgresql-instance --class rh-postgresql-apb --plan dev --params-json
'{"postgresql_database":"admin","postgresql_password":"admin","postgresql_user":"admin","po
stgresql_version":"9.6"}' -n szh-project
```

输出示例

```
  Name:       postgresql-instance
  Namespace:  szh-project
  Status:
  Class:      rh-postgresql-apb
  Plan:       dev

Parameters:
  postgresql_database: admin
  postgresql_password: admin
  postgresql_user: admin
  postgresql_version: "9.6"
```

6.2.3.3.1.1. 查看服务实例详情

查看服务实例详情：

```
$ svcat get instance
```

输出示例

```
      NAME          NAMESPACE        CLASS        PLAN   STATUS
+--------------------+-------------+------------------+------+--------+
  postgresql-instance   szh-project   rh-postgresql-apb   dev    Ready
```

6.2.3.3.2. 创建 ServiceBinding

创建 **ServiceBinding** 资源：

1. 服务目录控制器与代理服务器通信，以启动绑定。
2. 代理服务器创建凭证并将其签发给服务目录控制器。
3. 服务目录控制器将这些凭据作为 secret 添加到项目中。

使用以下命令创建服务绑定：

```
$ svcat bind postgresql-instance --name mediawiki-postgresql-binding
```

输出示例

```
 Name:       mediawiki-postgresql-binding
 Namespace:  szh-project
 Status:
 Instance:   postgresql-instance

Parameters:
 {}
```

6.2.3.3.2.1. 查看服务绑定详情

1. 查看服务绑定详情：

   ```
   $ svcat get bindings
   ```

   输出示例

   ```
            NAME              NAMESPACE        INSTANCE        STATUS
   +-----------------------------+-------------+--------------------+--------+
     mediawiki-postgresql-binding   szh-project   postgresql-instance   Ready
   ```

2. 在绑定服务后验证实例详情：

   ```
   $ svcat describe instance postgresql-instance
   ```

输出示例

```
  Name:       postgresql-instance
  Namespace:  szh-project
  Status:     Ready - The instance was provisioned successfully @ 2018-06-05 08:42:55
+0000 UTC
  Class:      rh-postgresql-apb
  Plan:       dev

Parameters:
  postgresql_database: admin
  postgresql_password: admin
  postgresql_user: admin
  postgresql_version: "9.6"

Bindings:
             NAME              STATUS
+------------------------------+--------+
  mediawiki-postgresql-binding   Ready
```

6.2.4. 删除资源

要删除服务目录相关资源，您需要取消绑定服务绑定并取消置备服务实例。

6.2.4.1. 删除服务绑定

1. 删除与服务实例关联的所有服务绑定：

```
$ svcat unbind -n <project-name> 1
  \ <instance-name> 2
```

1 包含服务实例的项目名称。

2 与绑定关联的服务实例名称。

例如：

```
$ svcat unbind -n szh-project postgresql-instance
```

输出示例

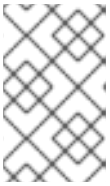
```
deleted mediawiki-postgresql-binding
```

2. 验证所有服务绑定是否已删除：

```
$ svcat get bindings
```

输出示例

```
  NAME   NAMESPACE   INSTANCE   STATUS
+------+-----------+----------+--------+
```



注意

运行此命令可删除实例的所有服务绑定。要从实例中删除单个绑定，请运行命令 **svcat unbind -n <project-name> --name <binding-name>**。例如，**svcat unbind -n szh-project --name mediawiki-postgresql-binding**。

3. 验证关联的 secret 已被删除。

```
$ oc get secret -n szh-project
```

输出示例

```
NAME                       TYPE                                  DATA      AGE
builder-dockercfg-jxk48    kubernetes.io/dockercfg               1         9m
builder-token-92jrf        kubernetes.io/service-account-token   4         9m
builder-token-b4sm6        kubernetes.io/service-account-token   4         9m
default-dockercfg-cggcr    kubernetes.io/dockercfg               1         9m
default-token-g4sg7        kubernetes.io/service-account-token   4         9m
default-token-hvdpq        kubernetes.io/service-account-token   4         9m
deployer-dockercfg-wm8th   kubernetes.io/dockercfg               1         9m
deployer-token-hnk5w       kubernetes.io/service-account-token   4         9m
deployer-token-xfr7c       kubernetes.io/service-account-token   4         9m
```

6.2.4.2. 删除服务实例

1. 取消置备服务实例：

```
$ svcat deprovision postgresql-instance
```

输出示例

```
deleted postgresql-instance
```

2. 验证实例已被删除：

```
$ svcat get instance
```

输出示例

```
  NAME   NAMESPACE   CLASS   PLAN   STATUS
+------+-----------+-------+------+--------+
```

6.2.4.3. 删除服务代理

1. 要删除服务目录的代理服务，请删除 **ClusterServiceBroker** 资源：

```
$ oc delete clusterservicebrokers template-service-broker
```

输出示例

```
clusterservicebroker "template-service-broker" deleted
```

2. 查看集群中安装的所有代理：

```
$ svcat get brokers
```

输出示例

```
          NAME                                  URL                                  STATUS
+-------------------------+-------------------------------------------------------------------------------------
------+--------+
  ansible-service-broker    https://asb.openshift-ansible-service-broker.svc:1338/ansible-
service-broker              Ready
```

3. 查看代理的 **ClusterServiceClass** 资源，以验证代理是否已移除：

```
$ svcat get classes
```

输出示例

```
  NAME   DESCRIPTION
+------+-------------+
```

6.3. TEMPLATE SERVICE BROKER

模板服务代理 (TSB)将服务目录可见性引入到 OpenShift Container Platform 初始发行版本起附带的 默认 Instant App 和 Quickstart 模板。TSB 也可以将 OpenShift Container Platform 模板 编写的任何内容提供给服务，无论是由红帽、集群管理员或用户提供，还是第三方供应商。

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

6.4. OPENSHIFT ANSIBLE BROKER

6.4.1. 概述

OpenShift Ansible 代理(OAB)是一种 Open Service Broker(OSB)API 的实现，用于管理由 *Ansible playbook 捆绑包(APB)* 定义的应用程序。APB 提供了在 OpenShift Container Platform 中定义和发布容器应用程序的新方法，它包括构建到带有 Ansible 运行时的容器镜像中的 Ansible playbook 捆绑包。APB 利用 Ansible 创建可自动化复杂部署的标准机制。

OAB 的设计遵循以下基本工作流：

1. 用户使用 OpenShift Container Platform Web 控制台从服务目录获取可用应用程序列表。
2. 服务目录请求 OAB 可用应用程序。
3. OAB 与定义的容器镜像 registry 通信，以了解有哪些 APB 可用。
4. 用户发出置备特定 APB 的请求。
5. provision 请求通过调用 APB 上的置备方法来满足用户的请求。

6.4.2. Ansible Playbook 捆绑包

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

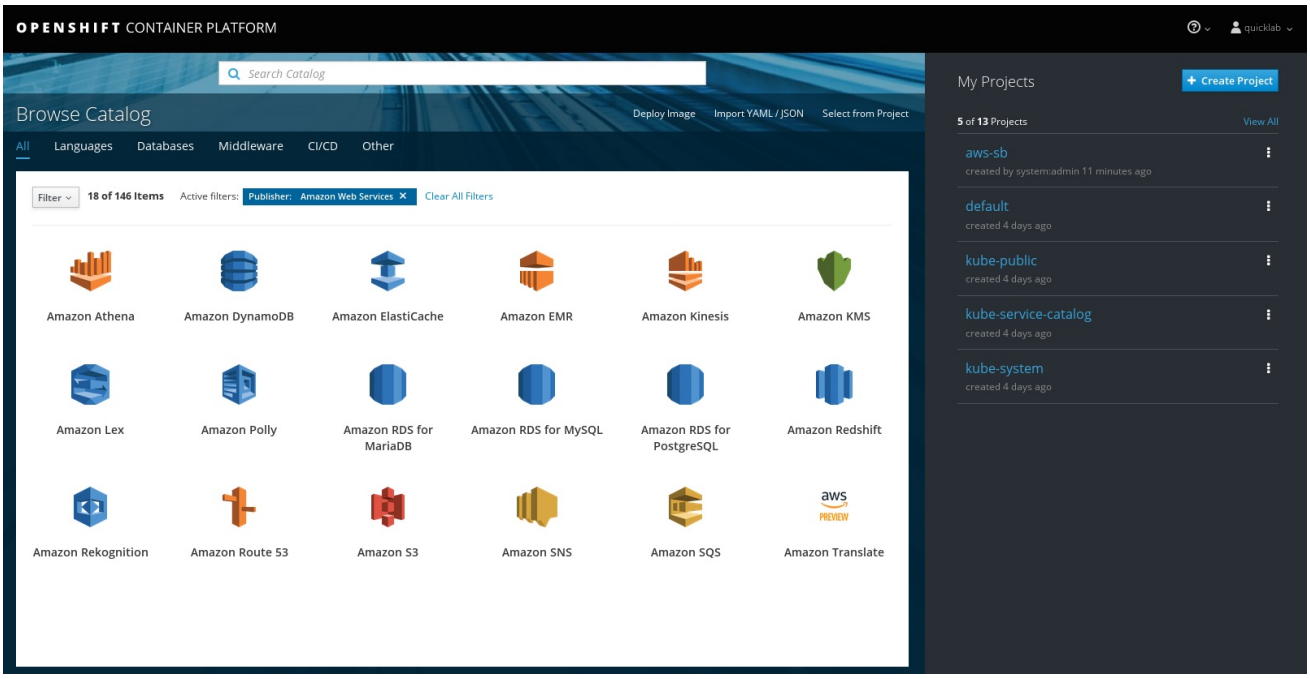
APB 使用含有指定 playbook 的简单目录来执行 OSB API 操作，比如置备和绑定。***apb.yml*** spec 文件中定义的元数据包含部署过程中要使用的一系列必要和可选参数。

如需了解有关整体设计以及如何编写 APB 的详细信息，请参阅 APB 开发指南。

6.5. AWS SERVICE BROKER

AWS Service Broker 通过 OpenShift Container Platform 服务目录提供对 Amazon Web Services(AWS)的访问。可以在 OpenShift Container Platform Web 控制台和 AWS 仪表板中配置和查看 AWS 服务和组件。

图 6.1. OpenShift Container Platform 服务目录中的 AWS 服务示例



有关安装 AWS Service Broker 的详情，请参考 **Amazon Web Services - Labs** 文档中的 AWS Service Broker 文档。

注意

AWS Service Broker 在 OpenShift Container Platform 上被支持并授权。对于最新的两个版本，在新的 OpenShift Container Platform 版本后，Amazon 可直接提供 Amazon 服务代理解决方案中的许多组件。红帽提供对 OpenShift 集群和服务目录问题的安装和故障排除的支持。