



OpenShift Container Platform 4.6

架构

OpenShift Container Platform 架构概述

OpenShift Container Platform 4.6 架构

OpenShift Container Platform 架构概述

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

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Architecture.ent file |.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文概述 OpenShift Container Platform 中的平台和应用架构。

目录

第 1 章 架构概述	4
1.1. 关于安装和更新	4
1.2. 关于 CONTROL PLANE	4
1.3. 关于面向开发人员的容器化应用程序	4
1.4. 关于 RED HAT ENTERPRISE LINUX COREOS(RHCOS)和 IGNITION	4
1.5. 关于 CI/CD 方法	5
1.6. 关于 ARG OCD	5
1.7. 关于准入插件	5
第 2 章 OPENSIFT CONTAINER PLATFORM 架构	6
2.1. OPENSIFT CONTAINER PLATFORM 简介	6
2.1.1. 关于 Kubernetes	6
2.1.2. 容器化应用程序的好处	6
2.1.2.1. 操作系统的好处	6
2.1.2.2. 部署和扩展优势	6
2.1.3. OpenShift Container Platform 概述	7
2.1.3.1. 定制操作系统	7
2.1.3.2. 简化的安装和更新流程	7
2.1.3.3. 其他主要功能	8
2.1.3.4. OpenShift Container Platform 生命周期	8
2.1.4. OpenShift Container Platform 的互联网访问	8
第 3 章 安装和更新	10
3.1. OPENSIFT CONTAINER PLATFORM 安装概述	10
3.1.1. 适用的平台	11
3.1.2. 安装过程	12
采用安装程序置备的基础架构的安装过程	13
采用用户置备的基础架构的安装过程	13
安装过程详细信息	13
安装范围	15
3.2. 关于 OPENSIFT UPDATE 服务	15
3.3. 非受管 OPERATOR 的支持策略	16
3.4. 后续步骤	16
第 4 章 OPENSIFT CONTAINER PLATFORM CONTROL PLANE	17
4.1. 了解 OPENSIFT CONTAINER PLATFORM CONTROL PLANE	17
4.1.1. 使用机器配置池进行节点配置管理	17
4.1.2. OpenShift Container Platform 中的机器角色	17
4.1.2.1. control plane 和节点主机兼容性	18
4.1.2.2. 集群 worker	18
4.1.2.3. 集群 master	18
4.1.3. OpenShift Container Platform 中的 Operator	20
4.1.3.1. OpenShift Container Platform 中的平台 Operator	20
4.1.3.2. 由 OLM 管理的 Operator	21
4.1.3.3. 关于 OpenShift Update 服务	21
4.1.3.4. 了解 Machine Config Operator	22
4.2. 其他资源	22
第 5 章 了解 OPENSIFT CONTAINER PLATFORM 开发	23
5.1. 关于容器化应用程序开发	23
5.2. 构建一个简单容器	23
5.2.1. 容器构建工具选项	24

5.2.2. 基础镜像选项	25
5.2.3. Registry 选项	25
5.3. 为 OPENSIFT CONTAINER PLATFORM 创建 KUBERNETES 清单	26
5.3.1. 关于 Kubernetes pod 和服务	26
5.3.2. 应用程序类型	26
5.3.3. 可用的支持组件	27
5.3.4. 应用清单	27
5.3.5. 后续步骤	27
5.4. 面向 OPERATOR 进行开发	28
第 6 章 RED HAT ENTERPRISE LINUX COREOS (RHCOS)	29
6.1. 关于 RHCOS	29
6.1.1. RHCOS 主要功能	29
6.1.2. 选择如何配置 RHCOS	30
6.1.3. 选择如何配置 RHCOS	31
6.1.4. 关于 Ignition	31
6.1.4.1. Ignition 工作方式	31
6.1.4.2. Ignition 操作序列	32
6.2. 查看 IGNITION 配置文件	32
6.3. 安装后更改 IGNITION 配置	34
第 7 章 CI/CD 方法和实践	36
7.1. 用于集群管理和应用程序配置管理的 CI/CD	36
7.2. GITOPS 方法和实践	36
7.2.1. GitOps 用于单集群和多集群管理	36
7.2.2. 用于应用程序配置管理的 GitOps	36
7.2.3. GitOps 技术供应商和集成商	37
第 8 章 在 OPENSIFT CONTAINER PLATFORM 中使用 ARGOCD	38
8.1. ARGOCD 做什么？	38
8.2. 支持声明	38
8.3. ARGOCD 文档	38
第 9 章 准入插件	39
9.1. 关于准入插件	39
9.2. 默认准入插件	39
9.3. WEBHOOK 准入插件	41
9.4. WEBHOOK 准入插件类型	42
9.4.1. 变异准入插件	43
9.4.2. 验证准入插件	44
9.5. 配置动态准入	45
9.6. 其他资源	52

第 1 章 架构概述

OpenShift Container Platform 是基于云的 Kubernetes 容器平台。OpenShift Container Platform 的基础是基于 Kubernetes，因此共享相同的技术。要了解更多有关 OpenShift Container Platform 和 Kubernetes 的信息，请参阅 [产品架构](#)。

1.1. 关于安装和更新

作为集群管理员，您可以使用 OpenShift Container Platform [安装程序](#) 在以下环境中安装集群：

- 安装程序置备的基础架构集群
- 用户置备的基础架构集群

1.2. 关于 CONTROL PLANE

[control plane](#) 管理集群中的 worker 节点和 pod。您可以使用机器配置池(MCP)配置节点。MCPS 是基于它们处理的资源的机器组，如 control plane 组件或用户工作负载。OpenShift 容器平台为主机分配不同的角色。这些角色定义机器在集群中的功能。集群包含标准 control plane 和 worker 角色类型的定义。

您可以使用 Operator 来打包、部署和管理 control plane 上的服务。Operator 是 OpenShift Container Platform 中的重要组件，因为它们提供以下服务：

- 执行健康检查
- 提供观察应用程序的方法
- 管理无线更新
- 确保应用程序保持在指定状态

1.3. 关于面向开发人员的容器化应用程序

作为开发人员，您可以使用不同的工具、方法和格式 [根据您的独特要求开发容器化应用](#)，例如：

- 使用各种 build-tool、base-image 和 registry 选项来构建简单的容器应用。
- 使用 OperatorHub 和模板等支持组件来开发应用程序。
- 将应用程序打包并部署为 Operator。

您还可以创建 Kubernetes 清单并将其存储在 Git 存储库中。Kubernetes 处理称为 pod 的基本单元。pod 是集群中一个运行中进程的单个实例。Pod 可以包含一个或多个容器。您可以通过对一组 pod 及其访问策略进行分组来创建服务。服务为创建和销毁容器集时的其他应用提供永久内部 IP 地址和主机名。Kubernetes 根据应用程序的类型定义工作负载。

1.4. 关于 RED HAT ENTERPRISE LINUX COREOS(RHCOS)和 IGNITION

作为集群管理员，您可以执行以下 Red Hat Enterprise Linux CoreOS(RHCOS)任务：

- 了解下一代 [单用途容器操作系统技术](#)。
- 选择如何配置 Red Hat Enterprise Linux CoreOS(RHCOS)

- 选择如何部署 Red Hat Enterprise Linux CoreOS(RHCOS) :
 - 安装程序置备的部署
 - 用户置备的基础架构部署

OpenShift Container Platform 安装程序创建部署集群所需的 Ignition 配置文件。Red Hat Enterprise Linux CoreOS(RHCOS)在初始配置过程中使用 Ignition 来执行常见磁盘任务，如分区、格式化、写入文件和配置用户。在第一次启动过程中，Ignition 将从安装介质或您指定的位置读取其配置，并将配置应用到机器。

您可以了解 [Ignition 的工作原理](#)，OpenShift Container Platform 集群中 Red Hat Enterprise Linux CoreOS(RHCOS)机器的过程，查看 Ignition 配置文件，并在安装后更改 Ignition 配置。

1.5. 关于 CI/CD 方法

您可以使用 [持续集成/持续交付\(CI/CD\)](#) 方法自动化应用程序开发的所有阶段。您还可以使用 GitOps 方法来创建可重复且可预测的流程来管理和重新创建 OpenShift Container Platform 集群和应用程序。

1.6. 关于 ARGOCD

您可以使用 [ArgoCD](#)，它是集群资源的声明性 GitOps 连续交付工具。

1.7. 关于准入插件

您可以使用 [准入插件](#) 来规范 OpenShift Container Platform 的功能。在资源请求经过身份验证并授权后，准入插件会截获主 API 的资源请求，以验证资源请求并确保扩展策略遵循。准入插件用于强制实施安全策略、资源限制或配置要求。

第 2 章 OPENSIFT CONTAINER PLATFORM 架构

2.1. OPENSIFT CONTAINER PLATFORM 简介

OpenShift Container Platform 是用于开发和运行容器化应用程序的平台。它旨在允许支持的应用程序和数据中心从少量机器和应用程序扩展到为数百万客户端服务的数千台机器。

OpenShift Container Platform 以 Kubernetes 为基础，为大规模电信、流视频、游戏、银行和其他应用提供引擎技术。借助红帽开放技术中的实现，您可以将容器化应用程序从单一云扩展到内部和多云环境。

2.1.1. 关于 Kubernetes

尽管容器镜像和从中运行的容器是现代应用程序开发的主要构建块，但要大规模运行它们，则需要可靠且灵活的分发系统。Kubernetes 是编配容器的事实标准。

Kubernetes 是一个开源容器编配引擎，用于自动化容器化应用程序的部署、扩展和管理。Kubernetes 的一般概念比较简单：

- 从一个或多个 worker 节点开始，以运行容器工作负载。
- 从一个或多个 control plane 节点（也称为 master 节点）管理这些工作负载的部署。
- 将容器嵌套到名为 pod 的部署单元中。使用 pod 可以为容器提供额外的元数据，并可在单个部署实体中对多个容器进行分组。
- 创建特殊种类的资产。例如，服务由一组 pod 及定义了访问方式的策略来表示。此策略可使容器连接到所需的服务，即便容器没有用于服务的特定 IP 地址。复制控制器（replication controller）是另一种特殊资产，用于指示一次需要运行多少个 pod 副本。您可以使用此功能来自动扩展应用程序，以适应其当前的需求。

短短数年，Kubernetes 已在大量的云和本地环境中被采用。借助开源开发模型，拥护和可以通过为组件（如网络、存储和身份验证）实施不同的技术来扩展 Kubernetes 的功能。

2.1.2. 容器化应用程序的好处

与使用传统部署方法相比，使用容器化应用程序具有许多优势。过去应用程序要安装到包含所有依赖项的操作系统上，容器能让一个应用程序随身携带自己的依赖项。创建容器化应用程序有很多好处。

2.1.2.1. 操作系统的好处

容器使用不含内核的小型专用 Linux 操作系统。它们的文件系统、网络、cgroups、进程表和命名空间与主机 Linux 系统分开，但容器可以在必要时与主机无缝集成。容器以 Linux 为基础，因此可以利用快速创新的开源开发模型带来的所有优势。

因为每个容器都使用专用的操作系统，所以您能够在同一主机上部署需要冲突软件依赖项的不同应用程序。每个容器都带有各自的依赖软件，并且管理自己的接口，如网络和文件系统，因此应用程序无需争用这些资产。

2.1.2.2. 部署和扩展优势

如果您在应用程序的主要版本之间进行滚动升级，则可以持续改进应用程序，既不会造成停机，又能仍然保持与当前版本的兼容性。

您还可以与现有版本一起部署和测试应用程序的新版本。容器通过测试后，只要部署更多新容器并删除旧容器便可。

由于应用程序的所有软件依赖项都在容器本身内解决，因此数据中心的每台主机上都能使用标准的操作系统。您无需逐一为应用主机配置特定的操作系统。当数据中心需要更多容量时，您可以部署另一个通用主机系统。

同样，扩展容器化应用程序也很简单。OpenShift Container Platform 提供了一种简单的、标准方式的容器化服务扩展功能。例如，如果将应用程序构建为一组微服务，而非大型的单体式应用程序，您可以分别扩展各个微服务来满足需求。有了这一能力，您可以只扩展需要的服务，而不是整个应用程序，从而在使用最少资源的前提下满足应用程序需求。

2.1.3. OpenShift Container Platform 概述

OpenShift Container Platform 为 Kubernetes 带来企业级增强，具体包括以下所列：

- 混合云部署。您可以将 OpenShift Container Platform 集群部署到各种公共云平台或数据中心。
- 集成了红帽技术。OpenShift Container Platform 中的主要组件源自 Red Hat Enterprise Linux (RHEL) 和相关的红帽技术。OpenShift Container Platform 得益于红帽企业级优质软件的严格测试和认证计划。
- 开源开发模型。开发以开放方式完成，源代码可从公共软件存储库中获得。这种开放协作促进了快速创新和开发。

虽然 Kubernetes 擅长管理应用程序，但它并未指定或管理平台级要求或部署过程。强大而灵活的平台管理工具和流程是 OpenShift Container Platform 4.6 具备的重要优势。以下各节介绍 OpenShift Container Platform 的一些独特功能和优势。

2.1.3.1. 定制操作系统

OpenShift Container Platform 使用 Red Hat Enterprise Linux CoreOS (RHCOS)，它是一款面向容器的操作系统，专为从 OpenShift Container Platform 运行容器化应用程序而设计，并可使用新工具提供快速安装、基于 Operator 的管理和简化的升级。

RHCOS 包括：

- Ignition，OpenShift Container Platform 将其用作首次启动系统配置来进行机器的初次上线和配置。
- CRI-O，Kubernetes 的原生容器运行时实现，可与操作系统紧密集成来提供高效和优化的 Kubernetes 体验。CRI-O，提供用于运行、停止和重启容器的工具。它完全取代了 OpenShift Container Platform 3 中使用的 Docker Container Engine。
- Kubelet，Kubernetes 的主要节点代理，负责启动和监视容器。

在 OpenShift Container Platform 4.6 中，所有 control plane 都需要使用 RHCOS。但 compute（计算）机器（也被称为 worker）可以使用 Red Hat Enterprise Linux (RHEL) 做为操作系统。如果选择使用 RHEL worker，与将 RHCOS 用于所有集群机器相比，您必须执行更多的系统维护。

2.1.3.2. 简化的安装和更新流程

使用 OpenShift Container Platform 4.6 时，如果您拥有具有正确权限的帐户，通过运行单个命令并提供几个值，就能在支持的云中部署生产集群。如果使用支持的平台，您还可以自定义云安装或在数据中心中安装集群。

对于将 RHCOS 用于所有机器的集群，不论是更新还是升级，OpenShift Container Platform 都是一个简单又高度自动化的流程。由于 OpenShift Container Platform 可以从中央 control plane 全面控制每台机器上运行的系统和服务（包括操作系统本身），因此升级被设计为一个自动事件。如果集群包含 RHEL worker 机器，则 control plane 可从简化的更新过程中受益，但您还需执行更多任务来升级 RHEL 机器。

2.1.3.3. 其他主要功能

Operator 既是 OpenShift Container Platform 4.6 代码库的基本单元，又是部署供应用程序使用的应用程序和软件组件的便捷方式。在 OpenShift Container Platform 中，Operator 可充当平台的基础，不再需要手动升级操作系统和 control plane 应用程序。OpenShift Container Platform Operator（如 Cluster Version Operator 和 Machine Config Operator）允许对这些关键组件进行简化的集群范围内管理。

Operator Lifecycle Manager (OLM) 和 OperatorHub 提供了相应的工具，可用于存储 Operator 并将其分发给开发和部署应用程序的人员。

Red Hat Quay Container Registry 是一个 Quay.io 容器 registry，为 OpenShift Container Platform 集群提供大多数容器镜像和 Operator。Quay.io 是 Red Hat Quay 的一个公共 registry 版本，可存储数百万镜像和标签。

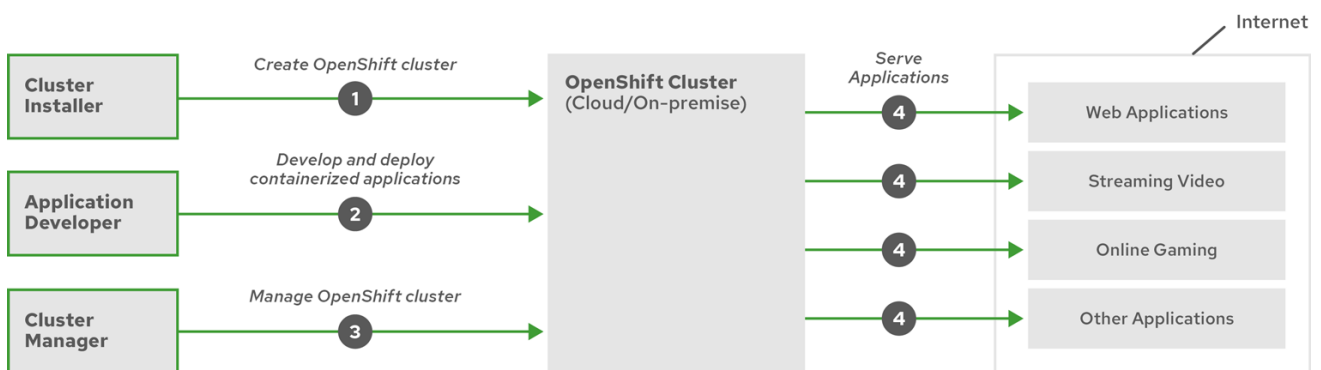
OpenShift Container Platform 中的其他 Kubernetes 增强功能包括软件定义网络 (SDN)、身份验证、日志聚合、监视和路由方面的改进。OpenShift Container Platform 还提供功能齐全的 web 控制台和自定义 OpenShift CLI (**oc**) 界面。

2.1.3.4. OpenShift Container Platform 生命周期

下图显示了 OpenShift Container Platform 的基本生命周期：

- 创建 OpenShift Container Platform 集群
- 管理集群
- 开发和部署应用程序
- 扩展应用程序

图 2.1. OpenShift Container Platform 高级概述



OpenShift_25_0519

2.1.4. OpenShift Container Platform 的互联网访问

在 OpenShift Container Platform 4.6 中，您需要访问互联网来安装集群。

您必须具有以下互联网访问权限：

- 访问 [OpenShift Cluster Manager](#) 以下载安装程序并执行订阅管理。如果集群可以访问互联网，并且没有禁用 Telemetry，该服务会自动授权您的集群。
- 访问 [Quay.io](#)，以获取安装集群所需的软件包。
- 获取执行集群更新所需的软件包。



重要

如果您的集群无法直接访问互联网，则可以在置备的某些类基础架构上执行受限网络安装。在此过程中，您要下载所需的内容，并使用它在镜像 registry (mirror registry) 中填充安装集群并生成安装程序所需的软件包。对于某些安装类型，集群要安装到的环境不需要访问互联网。在更新集群之前，要更新 registry 镜像系统中的内容。

第 3 章 安装和更新

3.1. OPENSIFT CONTAINER PLATFORM 安装概述

OpenShift Container Platform 安装程序为您提供了灵活性。您可以使用安装程序将集群部署到由安装程序置备并由集群维护的基础架构中，也可以将集群部署到您自己准备和维护的基础架构中。

这两种基本类型的 OpenShift Container Platform 集群通常称为安装程序置备的基础架构集群和用户置备的基础架构集群。

两种类型的集群都具有以下特征：

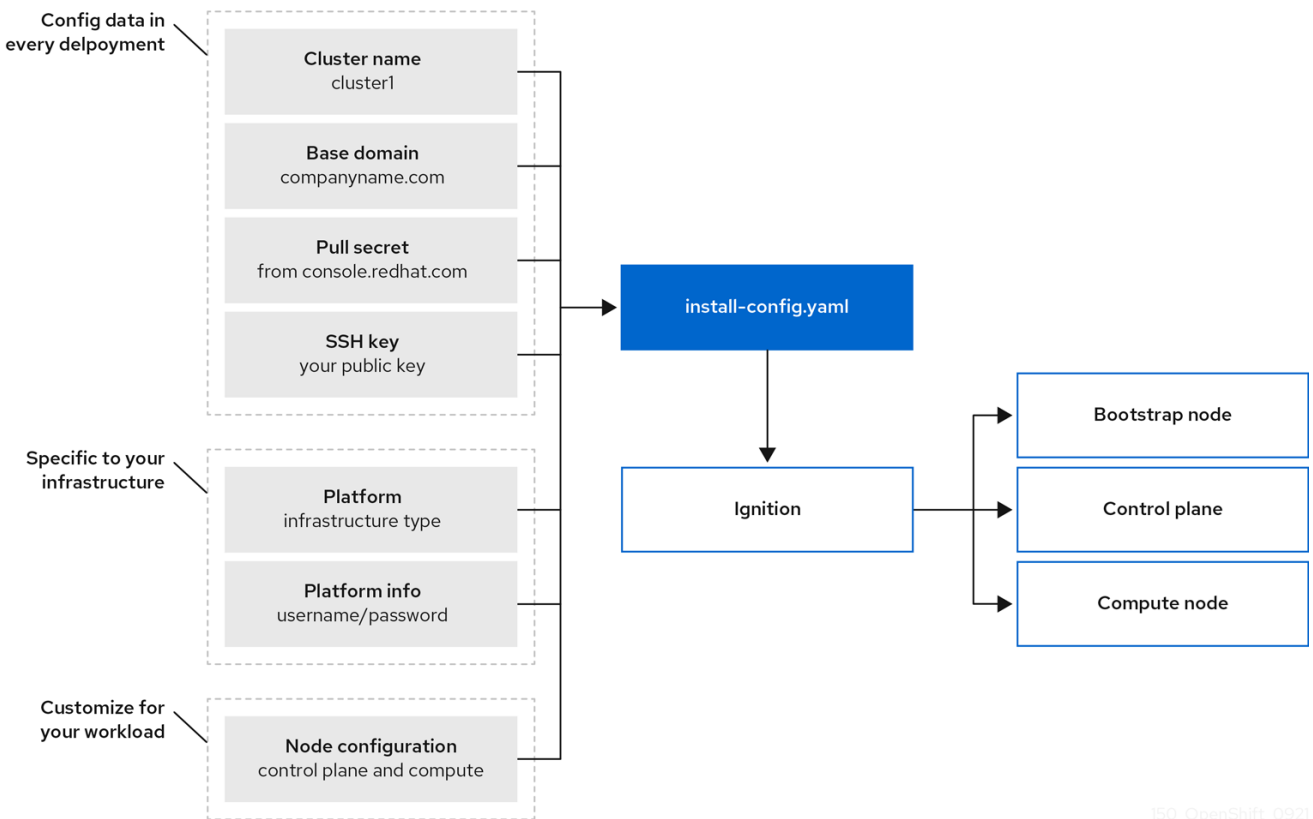
- 默认提供无单点故障的高可用性基础架构
- 管理员可以控制要应用的更新内容和更新的时间

两种类型的集群都使用同一个安装程序来部署。安装程序生成的主要资产是用于 Bootstrap、master 和 worker 机器的 Ignition 配置文件。有了这三个配置和配置得当的基础架构，就能启动 OpenShift Container Platform 集群。

OpenShift Container Platform 安装程序使用一组目标和依赖项来管理集群安装。安装程序具有一组必须实现的目标，并且每个目标都有一组依赖项。因为每个目标仅关注其自己的依赖项，所以安装程序可以采取并行实现多个目标。最终目标是正常运行的集群。通过满足依赖项而不是运行命令，安装程序能够识别和使用现有的组件，而不必运行命令来再次创建它们。

下图显示了安装目标和依赖项的子集：

图 3.1. OpenShift Container Platform 安装目标和依赖项



150_OpenShift_0921

在安装后，每一个集群机器都将使用 Red Hat Enterprise Linux CoreOS (RHCOS) 作为操作系统。RHCOS 是 Red Hat Enterprise Linux (RHEL) 的不可变容器主机版本，具有默认启用 SELinux 的 RHEL 内核。它包括作为 Kubernetes 节点代理的 **kubelet**，以及为 Kubernetes 优化的 CRI-O 容器运行时。

OpenShift Container Platform 4.6 集群中的每一 control plane 机器都必须使用 RHCOS，其中包括一个关键的首次启动置备工具，称为 Ignition。这一工具让集群能够配置机器。操作系统更新作为嵌入在容器镜像中的 Atomic OSTree 存储库交付，该镜像由 Operator 在整个集群中推广。实际的操作系统的更改通过使用 rpm-ostree 在每台机器上作为原子操作原位进行。通过结合使用这些技术，OpenShift Container Platform 可以像管理集群上的任何其他应用程序一样管理操作系统，通过原位升级使整个平台保持最新状态。这些原位更新可以减轻运维团队的负担。

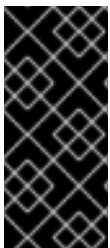
如果将 RHCOS 用作所有集群机器的操作系统，则集群将管理其组件和机器的所有方面，包括操作系统在内。因此，只有安装程序和 Machine Config Operator 才能更改机器。安装程序使用 Ignition 配置文件设置每台机器的确切状态，安装后则由 Machine Config Operator 完成对机器的更多更改，例如应用新证书或密钥等。

3.1.1. 适用的平台

在 OpenShift Container Platform 4.6 中，您可以在以下平台上安装使用安装程序置备的基础架构集群：

- Amazon Web Services (AWS)
- Google Cloud Platform (GCP)
- Microsoft Azure
- Red Hat OpenStack Platform 版本 13 和 16
 - 最新的 OpenShift Container Platform 版本支持最新的 RHOSP 长生命版本和中间版本。如需完整的 RHOSP 发行版本兼容性信息，请参阅 [RHOSP 上的 OpenShift Container Platform 支持列表](#)。
- Red Hat Virtualization (RHV)
- VMware vSphere
- AWS 上的 VMware Cloud (VMC)
- 裸机

对于所有这些集群，包括用来运行安装过程的计算机在内的所有机器都必须可直接访问互联网，以便为平台容器拉取镜像并向红帽提供 telemetry 数据。



重要

安装后，不支持以下更改：

- 混合云供应商平台
- 混合云供应商组件，比如使用与安装集群不同的平台中的持久性存储框架

在 OpenShift Container Platform 4.6 中，您可以在以下平台上安装使用用户置备的基础架构集群：

- AWS
- Azure

- GCP
- RHOSP
- RHV
- VMware vSphere
- AWS 上的 VMware Cloud (VMC)
- 裸机
- IBM Z 或 LinuxONE
- IBM Power 系统

根据平台所支持的情况，在用户置备的基础架构上安装允许您运行可访问互联网的机器，将集群放在代理后面，或者执行 [受限网络安装](#)。在受限网络安装中，您可以下载安装集群所需的镜像 (image)，将它们放在镜像 registry (mirror registry) 中，然后使用那些数据安装集群。虽然您需要访问互联网来为平台容器拉取镜像，但在 vSphere 或裸机基础架构上进行受限网络安装，您的集群机器不需要直接访问互联网。

[OpenShift Container Platform 4.x Tested Integrations](#) 页面中提供了有关针对不同平台进行集成测试的详细信息。

3.1.2. 安装过程

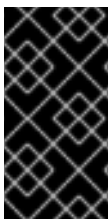
安装 OpenShift Container Platform 集群时，您可以从 Red Hat OpenShift Cluster Manager 站点的适当的 [Infrastructure Provider](#) 页面下载安装程序。此网站管理以下内容：

- 帐户的 REST API
- registry 令牌，这是用于获取所需组件的 pull secret
- 集群注册，它将集群身份信息与您的红帽帐户相关联，以方便收集使用情况指标

在 OpenShift Container Platform 4.6 中，安装程序是对一组资产执行一系列文件转换的 Go 二进制文件。与安装程序交互的方式因您的安装类型而异。

- 对于具有安装程序置备的基础架构集群，您可以将基础架构启动和置备委派给安装程序，而不是亲自执行。安装程序将创建支持集群所需的所有网络、机器和操作系统。
- 如果亲自为集群置备和管理基础架构，则必须提供所有集群基础架构和资源，包括 Bootstrap 机器、网络、负载均衡、存储和独立的集群机器。

安装期间使用三组文件：名为 **install-config.yaml** 的安装配置文件、Kubernetes 清单，以及您的机器类型适用的 Ignition 配置文件。

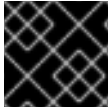


重要

安装期间可以修改控制基础 RHCOS 操作系统的 Kubernetes 和 Ignition 配置文件。但是，没有可用的验证机制来确认您对这些对象所做修改是适当的。如果修改了这些对象，集群可能会无法运行。由于存在这种风险，修改 Kubernetes 和 Ignition 配置文件不受支持，除非您遵循记录的流程或在红帽支持指示下操作。

安装配置文件转换为 Kubernetes 清单，然后清单嵌套到 Ignition 配置文件中。安装程序使用这些 Ignition 配置文件来创建集群。

运行安装程序时，所有配置文件会被修剪，因此请务必备份需要再次使用的所有配置文件。



重要

安装之后，您无法修改在安装过程中设置的参数，但可以修改一些集群属性。

采用安装程序置备的基础架构的安装过程

默认安装类型为使用安装程序置备的基础架构。默认情况下，安装程序充当安装向导，提示您输入它无法自行确定的值，并为其余参数提供合理的默认值。您还可以自定义安装过程来支持高级基础架构场景。安装程序将为集群置备底层基础架构。

您可以安装标准集群或自定义集群。对于标准集群，您要提供安装集群所需的最低限度详细信息。对于自定义集群，您可以指定有关平台的更多详细信息，如 control plane 使用的机器数量、集群部署的虚拟机的类型，或 Kubernetes 服务网络的 CIDR 范围。

若有可能，可以使用此功能来避免置备和维护集群基础架构。在所有其他环境中，可以使用安装程序来生成置备集群基础架构所需的资产。

对于安装程序置备的基础架构的集群，OpenShift Container Platform 可以管理集群的所有方面，包括操作系统本身。每台机器在启动时使用的配置引用其加入的集群中托管的资源。此配置允许集群在应用更新时自行管理。

采用用户置备的基础架构的安装过程

您还可以在自己提供的基础架构上安装 OpenShift Container Platform。您可以使用安装程序来生成置备集群基础架构所需的资产，再创建集群基础架构，然后将集群部署到您提供的基础架构中。

如果不使用安装程序置备的基础架构，您必须自己管理和维护集群资源，包括：

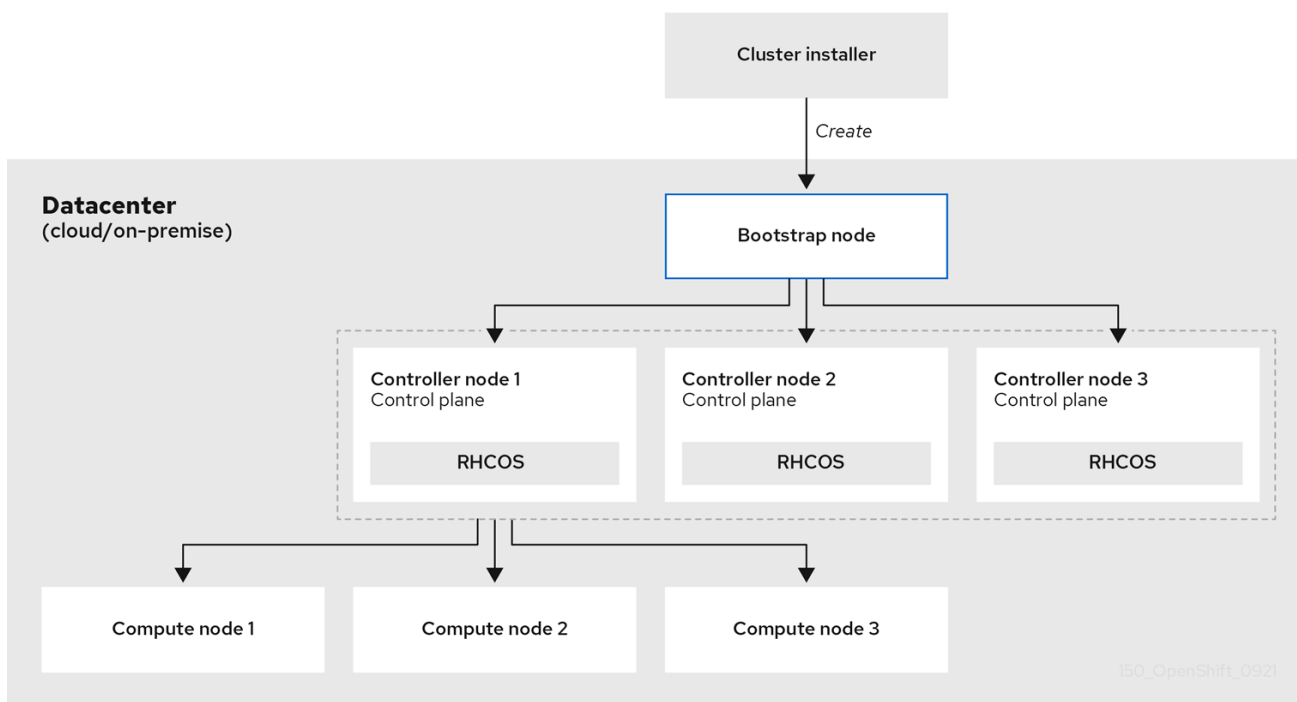
- 组成集群的 control plane 和计算机器的底层基础架构
- 负载均衡器
- 集群网络，包括 DNS 记录和所需的子网
- 集群基础架构和应用程序的存储

如果您的集群使用用户置备的基础架构，您可以选择将 RHEL 计算机器添加到集群中。

安装过程详细信息

由于在置备时集群中的每台机器都需要集群的相关信息，因此 OpenShift Container Platform 在初始配置期间会使用临时 *Bootstrap* 机器将所需的信息提供给持久 control plane。通过使用描述如何创建集群的 Ignition 配置文件进行启动。Bootstrap 机器将创建组成 control plane 的 master 机器。然后，control plane 机器创建计算 (compute) 机器。下图说明了这一过程：

图 3.2. 创建 bootstrap、control plane 和计算机器



集群机器初始化后，Bootstrap 机器将被销毁。所有集群都使用 Bootstrap 过程来初始化集群，但若您自己置备集群的基础架构，则必须手动完成许多步骤。



重要

安装程序生成的 Ignition 配置文件包含在 24 小时后过期的证书，然后在过期时进行续订。如果在更新证书前关闭集群，且集群在 24 小时后重启，集群会自动恢复过期的证书。一个例外情况是，您需要手动批准待处理的 **node-bootstrap** 证书签名请求（CSR）来恢复 kubelet 证书。如需更多信息，请参阅从过期的 *control plane* 证书中恢复的文档。

bootstrapp 集群涉及以下步骤：

1. bootstrap 机器启动并开始托管 control plane 机器引导所需的远程资源。（如果自己配置基础架构，则需要人工干预）
2. bootstrap 机器启动单节点 etcd 集群和一个临时 Kubernetes control plane。
3. control plane 机器从 bootstrap 机器获取远程资源并完成启动。（如果自己配置基础架构，则需要人工干预）
4. 临时 control plane 将生产环境的 control plane 调度到生产环境 control plane 机器。
5. Cluster Version Operator（CVO）在线并安装 etcd Operator。etcd Operator 在所有 control plane 节点上扩展 etcd。
6. 临时 control plane 关机，并将控制权交给生产环境 control plane。
7. bootstrap 机器将 OpenShift Container Platform 组件注入生产环境 control plane。
8. 安装程序关闭 bootstrap 机器。（如果自己配置基础架构，则需要人工干预）
9. control plane 设置计算节点。

10. control plane 以一组 Operator 的形式安装其他服务。

完成此 bootstrap 过程后，将生成一个全面运作的 OpenShift Container Platform 集群。然后，集群下载并配置日常运作所需的其余组件，包括在受支持的环境中创建计算（compute）机器。

安装范围

OpenShift Container Platform 安装程序的作用范围特意设计得比较狭窄。它旨在简化操作并确保成功。安装完成后，您可以完成更多的配置任务。

其他资源

- 如需了解有关 OpenShift Container Platform 配置资源的详细信息，请参阅[可用的集群自定义](#)。

3.2. 关于 OPENSIFT UPDATE 服务

OpenShift 更新服务（OpenShift Update Service，简称 OSUS）为 OpenShift Container Platform（包括 Red Hat Enterprise Linux CoreOS（RHCOS））提供了无线更新（over-the air update）功能。它提供了一个图表，其中包含组件 Operator 的 *顶点*（vertices）和连接它们的 *边*（edges）。图中的边代表了您可以安全更新到的版本。顶点是更新的有效负载，用于指定受管集群组件的预期状态。

集群中的 Cluster Version Operator (CVO) 会检查 OpenShift Container Platform 更新服务，并根据当前组件版本和图中的信息决定有效的更新和更新路径。当您请求更新时，CVO 使用该更新的发行镜像来更新您的集群。发行工件（artifact）作为容器镜像托管在 Quay 中。

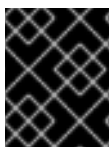
为了让 OpenShift Update Service 仅提供兼容的更新，可以使用一个版本验证管道来驱动自动化过程。每个发行工件都会被验证是否与支持的云平台 and 系统架构以及其他组件包兼容。在管道确认有适用的版本后，OpenShift Update Service 会通知您它可用。



重要

OpenShift Update Service 显示当前集群的所有推荐更新。如果 OpenShift Update Service 不建议升级路径，这可能是由于更新或目标发行版本存在已知问题。

两个控制器在持续更新模式下运行。第一个控制器持续更新有效负载清单，将清单应用到集群，并输出 Operator 的受控推出的状态，以指示它们是否处于可用、升级或失败状态。第二个控制器轮询 OpenShift Update Service，以确定更新是否可用。



重要

仅支持升级到较新版本。不支持将集群还原或回滚到以前的版本。如果您的更新失败，请联系红帽支持。

在更新过程中，Machine Config Operator(MCO)将新配置应用到集群机器。MCO 会处理由 **maxUnavailable** 字段指定的、协调机器配置池中的节点数量，并将它们标记为不可用。在默认情况下，这个值被设置为 **1**。然后，MCO 会应用新配置并重启机器。

如果您将 Red Hat Enterprise Linux (RHEL) 机器用作 worker，MCO 不会在这些机器上更新 kubelet，因为您必须首先在这些机器上更新 OpenShift API。

当新版本规格应用到旧的 kubelet 时，RHEL 机器无法返回 **Ready** 状态。在机器可用前，您无法完成更新。但是，因为已设置了最大不可用节点数，所以可以在一定机器无法使用的情况下，确保正常的集群操作。

OpenShift Update Service 由 Operator 和一个或多个应用程序实例组成。

3.3. 非受管 OPERATOR 的支持策略

Operator 的 **管理状态** 决定了一个 Operator 是否按设计积极管理集群中其相关组件的资源。如果 Operator 设置为 **非受管 (unmanaged)** 状态，它不会响应配置更改，也不会收到更新。

虽然它可以在非生产环境集群或调试过程中使用，但处于非受管状态的 Operator 不被正式支持，集群管理员需要完全掌控各个组件的配置和升级。

可使用以下方法将 Operator 设置为非受管状态：

- **独立 Operator 配置**

独立 Operator 的配置中具有 **managementState** 参数。这可以通过不同的方法来访问，具体取决于 Operator。例如，Cluster Logging Operator 通过修改它管理的自定义资源 (CR) 来达到此目的，而 Cluster Samples Operator 使用了集群范围配置资源。

将 **managementState** 参数更改为 **Unmanaged** 意味着 Operator 不会主动管理它的资源，也不会执行与相关组件相关的操作。一些 Operator 可能不支持此管理状态，因为它可能会损坏集群，需要手动恢复。



警告

将独立 Operator 更改为**非受管**状态会导致不支持该特定组件和功能。报告的问题必须在 **受管 (Managed)** 状态中可以重复出现才能继续获得支持。

- **Cluster Version Operator (CVO) 覆盖**

可将 **spec.overrides** 参数添加到 CVO 配置中，以便管理员提供对组件的 CVO 行为覆盖的列表。将一个组件的 **spec.overrides[].unmanaged** 参数设置为 **true** 会阻止集群升级并在设置 CVO 覆盖后提醒管理员：

```
Disabling ownership via cluster version overrides prevents upgrades. Please remove overrides before continuing.
```



警告

设置 CVO 覆盖会使整个集群处于不受支持状态。在删除所有覆盖后，必须可以重现报告的问题方可获得支持。

3.4. 后续步骤

- [不同平台的安装方法](#)

第 4 章 OPENSIFT CONTAINER PLATFORM CONTROL PLANE

4.1. 了解 OPENSIFT CONTAINER PLATFORM CONTROL PLANE

control plane 由 control plane 机器（也称为 master 机器）组成，用于管理 OpenShift Container Platform 集群。control plane 机器管理计算机器（也被称为 worker）上的工作负载。集群本身通过 Cluster Version Operator、Machine Config Operator 和一组单独 Operator 的操作来管理对机器的所有升级。

4.1.1. 使用机器配置池进行节点配置管理

运行 control plane 组件或用户工作负载的机器会根据其处理的资源类型划分为组。这些机器组称为机器配置池（MCP）。每个 MCP 管理一组节点及其对应的机器配置。节点的角色决定了它所属的 MCP；MCP 会根据其分配的节点角色标签管理节点。MCP 中的节点具有相同的配置；这意味着节点可以扩展并缩减，以适应增加或降低的工作负载。

默认情况下，在安装时集群创建两个 MCP：**master** 和 **worker**。每个默认 MCP 都有一个定义的配置，由 Machine Config Operator（MCO）应用，该配置负责管理 MCP 有助于 MCP 升级。您可以创建额外的 MCP 或自定义池来管理具有超出默认节点类型的自定义用例的节点。

自定义池是从 worker 池中继承其配置的池。它们将任何机器配置用于 worker 池，但添加了仅针对自定义池部署更改的能力。由于自定义池从 worker 池继承其配置，对 worker 池的任何更改都会应用到自定义池。MCO 不支持从 worker 池中继承其配置自定义池。



注意

节点只能包含在一个 MCP 中。如果节点有多个与多个 MCP 对应的标签，如 **worker,infra**，它由 infra 自定义池而不是 worker 池管理。自定义池根据节点标签在选择节点时具有优先权。不属于自定义池的节点由 worker 池管理。

建议您为集群中要管理的每个节点角色创建一个自定义池。例如，如果您创建 infra 节点来处理 infra 工作负载，建议创建一个自定义 infra MCP 将那些节点分组在一起。如果您将 **infra** 角色标签应用到 worker 节点，使其具有 **workerinfra dual** 标签，但没有自定义 infra MCP，则 MCO 认为它是一个 worker 节点。如果您从节点中删除 **worker** 标签，并应用 **infra** 标签而不将其分组到自定义池中，则该节点不可被 MCO 识别，且不受集群管理。



重要

任何使用 **infra** 角色标记的节点如果只运行 infra 工作负载，则不计算到订阅总数中。管理 infra 节点的 MCP 与集群决定订阅的方式相互排斥；为节点添加适当 **infra** 角色的标签，并使用污点以防止用户工作负载调度到该节点上，这是避免为 infra 工作负载添加订阅的唯一要求。

MCO 独立应用池更新。例如，如果有会影响所有池的更新，则每个池更新中的节点会相互并行。如果您添加自定义池，则该池中的节点还会尝试与 master 和 worker 节点同时更新。

4.1.2. OpenShift Container Platform 中的机器角色

OpenShift Container Platform 为主机分配不同的角色。这些角色定义机器在集群内的功能。集群包含标准 master 和 worker 角色类型的定义。



注意

集群还包含 bootstrap 角色的定义。由于 bootstrap 机器仅在集群安装期间使用，因此其功能在集群安装文档中阐述。

4.1.2.1. control plane 和节点主机兼容性

control plane 主机和节点主机之间的 OpenShift Container Platform 版本必须匹配。例如，在一个 4.9 集群中，所有 control plane 主机都必须是 4.9，所有节点也必需是 4.9。

可以接受集群升级过程中的临时不匹配。例如，当从 OpenShift Container Platform 4.8 升级到 4.9 时，一些节点将在其他节点之前升级到 4.9。control plane 主机和节点主机的长期偏移可能会暴露旧计算机器给错误和缺失的功能。用户应该尽快解决偏移的 control plane 主机和节点主机。

kubelet 服务不能更新于 **kube-apiserver**，且可能会有最多两个旧的次版本，具体取决于 OpenShift Container Platform 版本是否奇数甚至更低。下表显示了适当的版本兼容性：

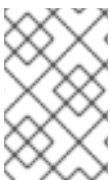
OpenShift Container Platform 版本	支持的 kubelet skew
奇异的 OpenShift Container Platform 次版本 [1]	最多一个版本
甚至 OpenShift Container Platform 次要版本 [2]	最多两个版本

1. 例如，OpenShift Container Platform 4.5、4.7、4.9。
2. 例如，OpenShift Container Platform 4.6、4.8、4.10。

4.1.2.2. 集群 worker

在 Kubernetes 集群中，worker 节点是运行和管理 Kubernetes 用户请求的实际工作负载的地方。worker 节点公告其容量，而作为 master 服务一部分的调度程序决定在哪些节点上启动容器和 pod。重要服务在每个 worker 节点上运行，包括 CRI-O（即容器引擎）、Kubelet（接受并履行运行和停止容器工作负载请求的服务），以及服务代理（管理 worker 之间 Pod 的通信）。

在 OpenShift Container Platform 中，机器集用来控制 worker 机器。具有 worker 角色的机器驱动计算工作负载，这些负载由自动扩展它们的特定机器池管控。因为 OpenShift Container Platform 具有支持多种机器类型的能力，因此 worker 机器被归类为 *计算* (compute) 机器。在本发行版本中，*worker 机器* 和 *计算机器* 是可以被互换使用的术语，因为计算机器的唯一默认类型是 worker 机器。在未来的 OpenShift Container Platform 版本中，默认情况下可能会使用不同类型的计算机器，如基础架构机器。



注意

机器集是 **machine-api** 命名空间中机器资源的定义。机器集是设计在特定云供应商上启动新机器的配置。相反，机器配置池（MCP）是 Machine Config Operator（MCO）命名空间的一部分。MCP 用于将机器分组在一起，以便 MCO 能够管理其配置并便于升级。

4.1.2.3. 集群 master

在 Kubernetes 集群中，control plane 节点（也称为 master 节点）运行控制 Kubernetes 集群所需的服务。在 OpenShift Container Platform 中，control plane 机器是 control plane。它们不仅仅包含用于管理 OpenShift Container Platform 集群的 Kubernetes 服务。因为所有具有 control plane 角色的机器都是

control plane 机器，所以 *master* 和 *control plane* 是可以互换使用的术语。control plane 机器不会被分成机器集，而是由一系列独立的机器 API 资源定义。额外的控件应用到 control plane 机器，以防止您删除所有 control plane 机器并破坏集群。



注意

所有生产部署都必须使用三个 control plane 节点。

master 上属于 Kubernetes 类别的服务包括 Kubernetes API 服务器、etcd、Kubernetes 控制器管理器和 Kubernetes 调度程序。

表 4.1. 在 control plane 上运行的 Kubernetes 服务

组件	描述
Kubernetes API 服务器	Kubernetes API 服务器验证并配置 pod、服务和复制控制器的数据。它还作为集群的共享状态提供了一个焦点。
etcd	etcd 存储持久 master 状态，其他组件则监视 etcd 的更改，以使其自身进入指定状态。
Kubernetes 控制器管理器	Kubernetes 控制器管理器监视 etcd 是否有对象的更改，如复制、命名空间、务帐户控制器对象，然后使用 API 来强制实施指定的状态。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。
Kubernetes 调度程序	Kubernetes 调度程序监视没有分配节点的新创建的 pod，并选择托管该 pod 的最佳节点。

另外，在 control plane 上运行的 OpenShift 服务包括 OpenShift API 服务器、OpenShift 控制器管理器、OpenShift OAuth API 服务器和 OpenShift OAuth 服务器。

表 4.2. 在 control plane 上运行的 OpenShift 服务

组件	描述
OpenShift API 服务器	OpenShift API 服务器验证并配置 OpenShift 资源（如项目、路由和模板）的数据。 OpenShift API 服务器由 OpenShift API Server Operator 管理。
OpenShift 控制器管理器	OpenShift 控制器管理器监视 etcd 是否有 OpenShift 对象的更改，如项目、路由和模板控制器对象，然后使用 API 来强制实施指定的状态。 OpenShift 控制器管理器由 OpenShift Controller Manager Operator 管理。
OpenShift OAuth API 服务器	OpenShift OAuth API 服务器验证并配置数据以向 OpenShift Container Platform 进行身份验证，如用户、组和 OAuth 令牌。 OpenShift OAuth API 服务器由 Cluster Authentication Operator 管理。

组件	描述
OpenShift OAuth 服务器	<p>用户从 OpenShift OAuth 服务器请求令牌，以向 API 进行身份验证。</p> <p>OpenShift OAuth 服务器由 Cluster Authentication Operator 管理。</p>

control plane 机器上的某些服务作为 systemd 服务运行，另一些则作为静态 pod 运行。

systemd 服务适合需要始终在特定系统启动后不久出现的服务。对于 control plane 机器，这包括允许远程登录的 sshd。它还包括以下服务：

- CRI-O 容器引擎 (crio)，用于运行和管理容器。OpenShift Container Platform 4.6 使用 CRI-O，而不是 Docker Container Engine。
- Kubelet (kubelet)，从 master 服务接受管理机器上容器的请求。

CRI-O 和 Kubelet 必须作为 systemd 服务直接在主机上运行，因为它们必须先运行，然后您才能运行其他容器。

installer-* 和 **revision-pruner-*** control plane pod 必须使用 root 权限运行，因为它们需要写入属于 root 用户的 `/etc/kubernetes` 目录。这些 pod 位于以下命名空间中：

- **openshift-etcd**
- **openshift-kube-apiserver**
- **openshift-kube-controller-manager**
- **openshift-kube-scheduler**

4.1.3. OpenShift Container Platform 中的 Operator

在 OpenShift Container Platform 中，Operator 是在 control plane 上打包、部署和管理服务的首选方法。它们还为用户运行的应用程序提供了便利。Operator 与 Kubernetes API 和 CLI 工具（如 **kubectl** 和 **oc** 命令）集成。它们提供了各种方式，以监视应用程序、执行健康检查、管理无线更新，以及确保应用程序保持在指定的状态。

因为 CRI-O 和 Kubelet 在每个节点上运行，所以几乎所有其他集群功能都可以通过使用 Operator 在 control plane 上进行管理。Operator 是 OpenShift Container Platform 4.6 中最重要的组件。使用 Operator 添加到 control plane 的组件包括重要的网络服务和凭证服务。

在 OpenShift Container Platform 集群中管理其他 Operator 的 Operator 是 Cluster Version Operator。

OpenShift Container Platform 4.6 使用不同类型的 Operator 来执行集群操作，并在集群上运行各种服务供您的应用程序使用。

4.1.3.1. OpenShift Container Platform 中的平台 Operator

在 OpenShift Container Platform 4.6 中，所有集群功能都划分到一系列平台 Operator 中。平台 Operator 管理集群功能的特定方面，如集群范围的应用程序日志记录、Kubernetes control plane 管理或机器置备系统。

每个 Operator 都为您提供确定集群功能的简单 API。Operator 将管理组件生命周期的细节隐藏起来。Operator 可以管理一个组件或数十个组件，但最终目标始终是通过自动化常见操作来减轻运维负担。Operator 还提供了更为精细的配置体验。若要配置各个组件，您可以修改 Operator 公开的 API，而不必

修改全局配置文件。

4.1.3.2. 由 OLM 管理的 Operator

Cluster Operator Lifecycle Management (OLM) 组件管理可在应用程序中使用的 Operator。它不管理组成 OpenShift Container Platform 的 Operator。OLM 是一个将 Kubernetes 原生应用程序作为 Operator 进行管理的框架。它不管理 Kubernetes 清单，而是管理 Kubernetes Operator。OLM 管理两种 Operator，即 Red Hat Operator 和经认证的 Operator。

一些 Red Hat Operator 用来提供集群功能，如调度程序和问题检测器。其他 Operator 则可供您自助管理并在应用程序中使用，例如 etcd。OpenShift Container Platform 还提供由社区构建和维护并经过认证的 Operator。这些经过认证的 Operator 为传统应用程序提供 API 层，因此您可以通过 Kubernetes 构造来管理应用程序。

4.1.3.3. 关于 OpenShift Update 服务

OpenShift 更新服务 (OpenShift Update Service, 简称 OSUS) 为 OpenShift Container Platform (包括 Red Hat Enterprise Linux CoreOS (RHCOS)) 提供了无线更新 (over-the air update) 功能。它提供了一个图表，其中包含组件 Operator 的 *顶点 (vertices)* 和连接它们的 *边 (edges)*。图中的边代表了您可以安全更新到的版本。顶点是更新的有效负载，用于指定受管集群组件的预期状态。

集群中的 Cluster Version Operator (CVO) 会检查 OpenShift Container Platform 更新服务，并根据当前组件版本和图中的信息决定有效的更新和更新路径。当您请求更新时，CVO 使用该更新的发行镜像来更新您的集群。发行工件 (artifact) 作为容器镜像托管在 Quay 中。

为了让 OpenShift Update Service 仅提供兼容的更新，可以使用一个版本验证管道来驱动自动化过程。每个发行工件都会被验证是否与支持的云平台 and 系统架构以及其他组件包兼容。在管道确认有适用的版本后，OpenShift Update Service 会通知您它可用。



重要

OpenShift Update Service 显示当前集群的所有推荐更新。如果 OpenShift Update Service 不建议升级路径，这可能是由于更新或目标发行版本存在已知问题。

两个控制器在持续更新模式下运行。第一个控制器持续更新有效负载清单，将清单应用到集群，并输出 Operator 的受控推出的状态，以指示它们是否处于可用、升级或失败状态。第二个控制器轮询 OpenShift Update Service，以确定更新是否可用。



重要

仅支持升级到较新版本。不支持将集群还原或回滚到以前的版本。如果您的更新失败，请联系红帽支持。

在更新过程中，Machine Config Operator (MCO) 将新配置应用到集群机器。MCO 会处理由 **maxUnavailable** 字段指定的、协调机器配置池中的节点数量，并将它们标记为不可用。在默认情况下，这个值被设置为 **1**。然后，MCO 会应用新配置并重启机器。

如果您将 Red Hat Enterprise Linux (RHEL) 机器用作 worker，MCO 不会在这些机器上更新 kubelet，因为您必须首先在这些机器上更新 OpenShift API。

当新版本规格应用到旧的 kubelet 时，RHEL 机器无法返回 **Ready** 状态。在机器可用前，您无法完成更新。但是，因为已设置了最大不可用节点数，所以可以在一定机器无法使用的情况下，确保正常的集群操作。

OpenShift Update Service 由 Operator 和一个或多个应用程序实例组成。

4.1.3.4. 了解 Machine Config Operator

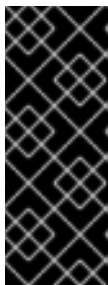
OpenShift Container Platform 4.6 集成了操作系统和集群管理。由于集群管理自己的更新，包括集群节点上 Red Hat Enterprise Linux CoreOS (RHCOS) 的更新，因此 OpenShift Container Platform 提供了可靠的生命周期管理体验，能够简化节点升级的编配。

OpenShift Container Platform 使用三个守护进程集和控制器来简化节点管理。这些守护进程集通过使用标准的 Kubernetes 式构造来编配操作系统更新和主机配置更改。它们包括：

- **machine-config-controller**，协调从 control plane 进行的机器升级。它监控所有集群节点并编配其配置更新。
- **machine-config-daemon** 守护进程集在集群中的每个节点上运行，并根据 MachineConfigController 的指示将机器更新为机器配置定义的配置。当节点检测到更改时，它会排空其 pod，应用更新并重启。这些更改以 Ignition 配置文件的形式出现，这些文件应用指定的机器配置并控制 kubelet 配置。更新本身在容器中交付。此过程是成功管理 OpenShift Container Platform 和 RHCOS 更新的关键。
- **machine-config-server** 守护进程集，在加入集群时为 control plane 节点提供 Ignition 配置文件。

机器配置是 Ignition 配置的子集。**machine-config-daemon** 读取机器配置，以查看是否需要更新 OSTree，或者是否必须应用一系列 systemd kubelet 文件更改、配置更改，或者对操作系统或 OpenShift Container Platform 配置的其他更改。

执行节点管理操作时，您可以创建或修改 **KubeletConfig** 自定义资源(CR)。



重要

当对机器配置进行修改时，Machine Config Operator 会自动重启所有对应的节点，以使更改生效。

要防止节点在机器配置更改后自动重启，在更改之前，必须通过在相应的机器配置池中将 **spec.paused** 字段设置为 **true** 来暂停自动引导过程。暂停后，机器配置更改不会生效，除非将 **spec.paused** 字段设置为 **false**，且节点已重启至新配置。

4.2. 其他资源

- 如需有关在 Machine Config Operator 更改机器配置后防止 control plane 机器的信息，请参阅[禁用 Machine Config Operator 自动重新引导](#)。

第 5 章 了解 OPENSIFT CONTAINER PLATFORM 开发

为了在开发和运行企业级品质应用程序时充分利用容器的功能，请确保您的环境受相应工具的支持，让容器能够：

- 创建为离散的服务，可以连接到其他容器化和非容器化服务。例如，您可能希望将应用程序与数据库衔接，或将监控应用程序附加到数据库。
- 具有弹性，因此在服务器崩溃或需要停机维护或退役时，容器可以在另一台机器上启动。
- 实现自动化，以自动获取代码更改，然后启动和部署自身的新版本。
- 得以扩展或复制，在需求增加时为客户端提供更多实例，在需求下降时缩减为更少的实例。
- 以不同的方式运行，具体由应用程序的类型决定。例如，一个应用程序可能每月运行一次来生成报告，然后退出。另一个应用程序可能需要持续运行，并且必须对客户端高度可用。
- 受到管理，以便您可以监视应用程序的状态并在出现问题时做出反应。

容器得到广泛接受，对能让容器适合企业使用的工具和方法的需求也随之诞生，这使得容器有了丰富的选择。

本节的其余部分介绍在 OpenShift Container Platform 中构建和部署容器化 Kubernetes 应用程序时可以选择创建的资产。它还说明您可以采用哪些方法来满足不同类型的应用程序和开发需求。

5.1. 关于容器化应用程序开发

您可以通过多种方式使用容器来进行应用程序开发，每种方法更适合的使用情景也各不相同。为了说明这种多样性，本文在介绍一系列方法时首先从开发单个容器开始，最后将容器部署为面向大型企业的任务关键型应用程序。这些方法展示了不同的工具、格式和方法，供您用于容器化应用程序开发。本主题描述：

- 构建一个简单容器并将其存储在 registry 中
- 创建 Kubernetes 清单并将其保存到 Git 存储库
- 使 Operator 能够与其他项共享您的应用程序

5.2. 构建一个简单容器

您对应用程序有了一个想法，想要对其进行容器化。

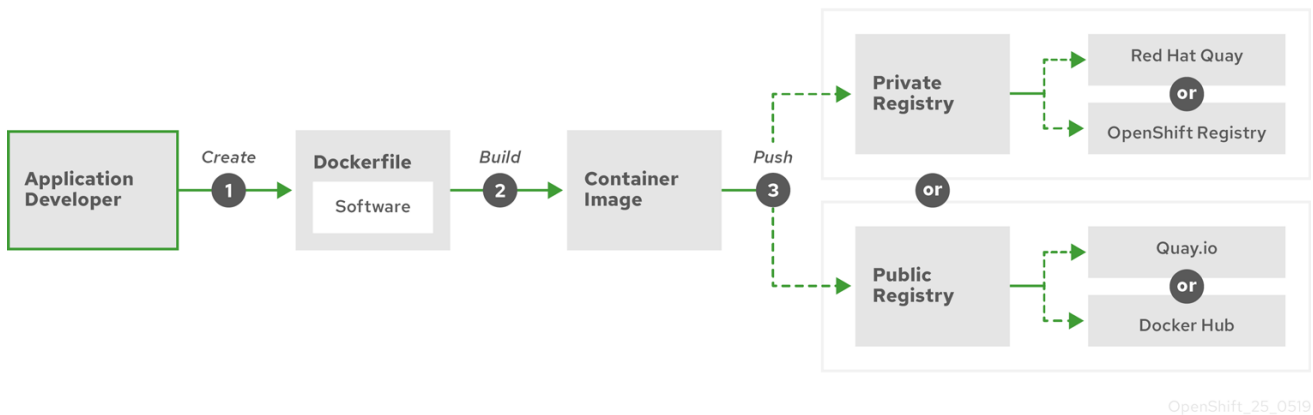
首先，您需要一个用于构建容器的工具，如 buildah 或 docker，还需要一个描述容器中内容的文件，通常是 [Dockerfile](#)。

接下来，您需要一个位置来推送生成的容器镜像，以便可以将它拉取到您想要它运行的位置。这个位置就是容器 registry。

大多数 Linux 操作系统上都会默认安装各个组件的一些示例，但 Dockerfile 除外，需要您自己提供。

下图显示了构建和推送镜像的流程：

图 5.1. 创建简单容器化应用程序并将其推送到 registry



如果您使用运行 Red Hat Enterprise Linux (RHEL) 作为操作系统的计算机，则容器化应用程序创建过程需要以下几个步骤：

1. 安装容器构建工具：RHEL 包含一组工具，其中包括用于构建和管理容器的 podman、buildah 和 skopeo。
2. 创建一个 Dockerfile 来组合基础镜像和软件：有关构建容器的信息存放在名为 **Dockerfile** 的文件中。在这个文件中，您要标识从中构建的基本镜像、要安装的软件包，以及要复制到容器中的软件。您还要标识参数值，如公开到容器外部的网络端口和挂载到容器内的卷。将您的 Dockerfile 和您要容器化的软件放到 RHEL 系统上的目录中。
3. 运行 buildah 或 docker 构建：运行 **buildah build-using-dockerfile** 或 **docker build** 命令，将您选择的基础镜像拉取到本地系统，再创建一个存储在本地的容器镜像。您还可以使用 buildah，在不用 Dockerfile 的前提下构建容器镜像。
4. 标记 (tag) 并推送到 registry：向新容器镜像添加标签 (tag)，以标识要在其中存储和共享容器的 registry 位置。然后，通过运行 **podman push** 或 **docker push** 命令将该镜像推送到 registry。
5. 拉取并运行镜像：从具有 podman 或 docker 等容器客户端工具的任何系统，运行用于标识新镜像的命令。例如，运行 **podman run <image_name>** 或 **docker run <image_name>** 命令。其中，**<image_name>** 是新容器镜像的名称，类似于 **quay.io/myrepo/myapp:latest**。registry 可能需要凭证才能推送和拉取镜像。

如需更多有关构建容器镜像、将其推送到 registry 并运行它们的过程的详细信息，请参阅使用 [Buildah 自定义镜像构建](#)。

5.2.1. 容器构建工具选项

尽管 Docker Container Engine 和 **docker** 命令是操作容器的常用工具，但在 RHEL 和许多其他 Linux 系统上，您也可以选择另一组容器工具，包括 podman、skopeo 和 buildah 等。您仍然可以使用 Docker Container Engine 来创建容器，这些容器将在 OpenShift Container Platform 和任何其他容器平台中运行。

使用 buildah、podman 和 skopeo 构建和管理容器会生成行业标准的容器镜像，其包含经过专门调优的功能，以便于这些容器最终部署到 OpenShift Container Platform 或其他 Kubernetes 环境中。这些工具不是作为守护进程运行的，可在没有 root 权限的情况下运行，因此运行它们的开销比较小。

最终在 OpenShift Container Platform 中运行容器时，您要使用 [CRI-O](#) 容器引擎。CRI-O 在 OpenShift Container Platform 集群中的每一 worker 和 control plane 机器（也称为 master 机器）上运行，但 CRI-O 尚不支持作为 OpenShift Container Platform 外的独立运行时。

5.2.2. 基础镜像选项

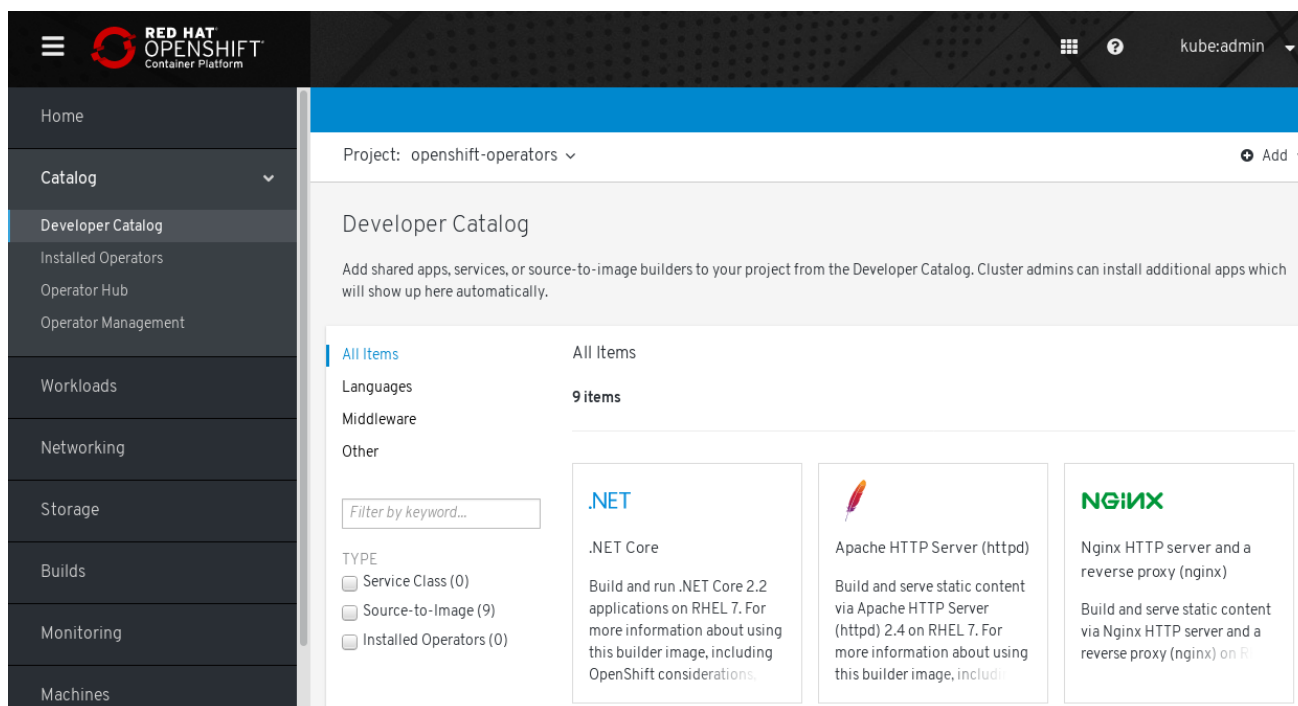
选择用来构建应用程序的基础镜像包含一组软件，这些软件为应用程序提供一个 Linux 系统。在您构建自己的镜像时，您的软件将放置到该文件系统中，可以像对待操作系统一样看待该文件系统。基础镜像的选择会对容器未来的安全性、效率和可升级性产生重大影响。

红帽提供了一组新的基础镜像，称为**红帽通用基础镜像 (UBI)**。这些镜像基于 Red Hat Enterprise Linux，与红帽过去提供的基础镜像相似，但有一个主要区别：无需红帽订阅就能自由重新分发。因此，您可以在 UBI 镜像上构建应用程序，不必担心如何共享它们或需要为不同的环境创建不同的镜像。

这些 UBI 镜像具有标准、初始和最低版本。您还可以将 [Red Hat Software Collections](#) 镜像用作依赖特定运行时环境（如 Node.js、Perl 或 Python）的应用程序的基础。其中一些运行时基础镜像的特殊版本称为 Source-to-Image (S2I) 镜像。使用 S2I 镜像时，您可以将代码插入到可随时运行该代码的基础镜像环境中。

您可以直接从 OpenShift Container Platform Web UI 中使用 S2I 镜像，只需选择 **Catalog → Developer Catalog** 便可，如下图中所示：

图 5.2. 为需要特定运行时的应用选择 S2I 基础镜像



5.2.3. Registry 选项

容器镜像仓库 (registry) 是存储容器镜像的位置，以便您可以与他人共享，并将它们提供给最终运行的平台。您可以选择提供免费帐户的大型公共容器 registry，也可选择提供更多存储和特殊功能的高级版本。您还可以安装自己的 registry，供您的组织专用或有选择地共享给他人。

要获取红帽和认证合作伙伴提供的镜像，您可以从 Red Hat Registry 中提取。Red Hat Registry 存在于两个位置：**registry.access.redhat.com**（无需身份验证，但已弃用）和 **registry.redhat.io**（需要身份验证）。您可以在[红帽生态系统目录的容器镜像部分](#)了解 Red Hat Registry 中的红帽及合作伙伴的镜像。除了列出红帽容器镜像外，它还显示有关这些镜像的内容和质量的广泛信息，包括基于已应用安全更新的健康分数。

大型公共 registry 包括 [Docker Hub](#) 和 [Quay.io](#)。Quay.io registry 由红帽所有和管理。OpenShift Container Platform 中使用的许多组件都存储在 Quay.io 中，包括用于部署 OpenShift Container Platform 本身的容器镜像和 Operator。Quay.io 还提供了存储其他类型内容的方法，包括 Helm Charts。

如果需要私有容器 registry，OpenShift Container Platform 本身包括一个私有容器 registry，随 OpenShift Container Platform 一起安装并在其集群上运行。红帽也提供 Quay.io registry 的私有版本，称为 [Red Hat Quay](#)。Red Hat Quay 包括跨地域复制、Git 构建触发器、Clair 镜像扫描和许多其他功能。

此处提到的所有 registry 都可能需要凭证才能从中下载镜像。这些凭证中有些是通过 OpenShift Container Platform 在整个集群范围内提供的，另一些凭证则可以分配给个人。

5.3. 为 OPENSIFT CONTAINER PLATFORM 创建 KUBERNETES 清单

尽管容器镜像是容器化应用程序的基本构建块，但需要更多信息才能在 Kubernetes 环境（如 OpenShift Container Platform）中管理和部署该应用程序。创建镜像后的典型后续步骤：

- 了解 Kubernetes 清单中使用的不同资源
- 就您将运行的应用程序做出一些决策
- 收集支持组件
- 创建清单并将该清单存储到 Git 存储库中，以便您可以将其存储在源版本控制系统中，对其进行审核、跟踪和升级，并将其部署到下一环境中，在必要时回滚到旧版本，以及与他人共享

5.3.1. 关于 Kubernetes pod 和服务

容器镜像是 docker 的基本单元，而 Kubernetes 使用的基本单元称为 [pods](#)。Pod 代表构建应用程序的下一步。pod 可以包含一个或多个容器。关键之处在于 pod 是您部署、扩展和管理的单个单元。

在决定 pod 中要放入的内容时，可扩展性和命名空间或许是要考虑的主要项目。为便于部署，您可能需要将容器部署到 pod 中，并在 pod 中包含其本身的日志记录和监控容器。以后，在您运行 pod 并需要扩展额外的实例时，其他那些容器也会随之扩展。对于命名空间，pod 中的容器共享相同的网络接口、共享存储卷和资源限制（如内存和 CPU）。这样一来，将 pod 内容作为一个单元进行管理变得更加轻松。pod 中的容器还可以使用标准的进程间通信（如 System V 信号或 POSIX 共享内存）相互通信。

虽然单个 pod 代表 Kubernetes 中的一个可扩展单元，但[服务](#)提供了一个途径，能够将一系列 pod 分组到一起以创造完整且稳定的应用程序，完成诸如负载均衡之类的任务。服务也比 pod 更持久，因为服务可以一直从同一 IP 地址使用，直到您删除为止。在使用服务时，可通过名称来请求服务，OpenShift Container Platform 集群则将该名称解析为您可用于访问构成该服务的 pod 的 IP 地址和端口。

从本质上讲，容器化应用程序与运行它们的操作系统是隔开的，进而与其用户隔开。Kubernetes 清单的一部分描述了如何通过定义[网络策略](#)将应用程序公开给内部和外部网络，对您的容器化应用的通信进行精细的控制。要将来自集群外部的 HTTP、HTTPS 和其他服务的传入请求连接到集群内部的服务，可以使用 [Ingress](#) 资源。

如果容器需要磁盘存储而不是数据库存储（可以通过服务提供），则可以将[卷](#)添加到清单中，使该存储可供 pod 使用。您可以配置清单以创建持久性卷 (PV)，或动态创建添加到 [Pod](#) 定义中的卷。

在定义了组成应用程序的一组 pod 后，您可以在 [Deployment](#) 和 [DeploymentConfig](#) 对象中定义这些 pod。

5.3.2. 应用程序类型

接下来，考虑您的应用程序类型对其运行方式的影响。

Kubernetes 定义了适用于不同类型应用程序的不同工作负载。要确定适合应用程序的工作负载，请考虑该应用程序是否：

- 运行结束后即告完成。例如，启动某个应用程序以生成报告，并在报告完成时退出。之后一个月内可能不会再次运行这个应用程序。对于这些类型的应用程序，合适的 OpenShift Container Platform 对象包括 **Job** 和 **CronJob** 对象。
- 预计将持续运行。对于长时间运行的应用程序，您可以编写一个**部署**。
- 需要高度可用。如果应用程序需要高可用性，那么您需要调整部署的大小，使其包含不止一个实例。**Deployment** 或 **DeploymentConfig** 对象可以包含该类型的应用程序的**副本集**。借助副本集，pod 可以跨越多个节点运行，确保即使在 worker 中断时该应用程序也始终可用。
- 需要在每个节点上运行。某些类型的 Kubernetes 应用程序设计为在集群中的每个 master 节点或 worker 节点上运行。例如，DNS 和监控应用程序需要在每个节点上持续运行。您可以将这类应用程序作为**守护进程集**运行。您还可以根据节点标签在节点的子集上运行守护进程。
- 需要生命周期管理。当您移交应用程序供其他人使用时，请考虑创建 **Operator**。Operator 可帮助您构建智能功能，自动处理备份和升级之类的事务。与 Operator Lifecycle Manager (OLM) 相结合，集群管理器可以将 Operator 公开给选定命名空间，以便集群中的用户可以运行它们。
- 具有标识或编号要求。应用程序可能具有标识或编号要求。例如，您可能需要运行应用程序的三个实例，并将这些实例命名为 **0**、**1** 和 **2**。一个**有状态的集合**适合这个应用程序。有状态的集合对需要独立存储的应用程序（如数据库和 zookeeper 集群）最有用。

5.3.3. 可用的支持组件

您编写的应用程序可能需要支持组件，如数据库或日志记录组件。为满足这一需求，可以从 OpenShift Container Platform Web 控制台中提供的以下目录获取所需的组件：

- OperatorHub，可在每个 OpenShift Container Platform 4.6 集群中使用。借助 OperatorHub，集群操作员可以使用来自红帽、红帽认证合作伙伴和社区成员的 Operator。集群操作员可以在集群中的所有命名空间或选定命名空间中提供这些 Operator，让开发人员能够通过他们的应用程序启动并配置这些 Operator。
- 模板，对于一次性类型的应用程序很有用。在该应用程序中，组件的生命周期在安装后并不重要。模板提供了一种简便方式，可以从最小的开销开始开发 Kubernetes 应用程序。模板可以是资源定义列表，可以是 **Deployment**、**Service**、**Route** 或其他对象。如果要更改名称或资源，可通过参数形式在模板中设置这些值。

您可以根据开发团队的特定需求，配置支持的 Operator 和模板，然后在开发人员开展工作的命名空间中提供它们。许多人将共享模板添加到 **openshift** 命名空间中，因为可以从所有其他命名空间访问这个命名空间。

5.3.4. 应用清单

借助 Kubernetes 清单（manifest），您可以更加全面地了解组成 Kubernetes 应用程序的组件。您可以将这些清单编写为 YAML 文件，并通过应用到集群来部署它们，例如通过运行 **oc apply** 命令。

5.3.5. 后续步骤

此时，请考虑对容器开发过程进行自动化的方法。理想情况下，您可以使用某种 CI 管道来构建镜像并将其推送到 registry。特别是，GitOps 管道可将容器开发与 Git 存储库集成在一起，您将使用 Git 存储库来存储构建应用程序所需的软件。

到目前为止的工作流程可能如下所示：

- 第 1 天：编写一些 YAML。然后，运行 **oc apply** 命令将 YAML 应用于集群并测试其是否正常工作。

- 第 2 天：将 YAML 容器配置文件放进您的 Git 存储库中。想要安装该应用或协助您改进的人可以从那里拉取 YAML，并应用到他们用于运行应用程序的集群中。
- 第 3 天：考虑为应用程序编写 Operator。

5.4. 面向 OPERATOR 进行开发

如果您的应用程序要提供给他人运行，最好将其打包并部署为 Operator。如前文所述，Operator 向您的应用程序添加了一个生命周期组件。使用它可以实现在安装应用程序后需要进行的维护任务。

在将应用程序创建为 Operator 时，您可以纳入自己有关如何运行和维护应用程序的知识。您可以内置用来升级、备份、扩展应用程序或跟踪其状态的功能。正确配置应用程序后，维护任务（如更新 Operator）可以自动发生，并且不为 Operator 用户所见。

例如，设置为在特定时间自动备份数据的 Operator 就非常实用。让 Operator 在设定的时间管理应用程序备份，可以使系统管理员免于记忆这些事务。

传统上手动完成的任何应用程序维护（如备份数据或轮转证书）都可以借助 Operator 自动完成。

第 6 章 RED HAT ENTERPRISE LINUX COREOS (RHCOS)

6.1. 关于 RHCOS

Red Hat Enterprise Linux CoreOS (RHCOS) 通过提供带有自动化远程升级功能的 Red Hat Enterprise Linux (RHEL) 的质量标准来代表下一代单用途容器操作系统技术。

对于所有 OpenShift Container Platform 机器，仅支持将 RHCOS 作为 OpenShift Container Platform 4.6 的组件。RHCOS 是唯一受 OpenShift Container Platform control plane 或 master 机器支持的操作系统。虽然 RHCOS 是所有集群机器的默认操作系统，但您仍可以创建使用 RHEL 作为其操作系统的计算 (compute) 机器 (也称为 worker)。OpenShift Container Platform 4.6 中有两个通用的部署 RHCOS 的方法：

- 如果将集群安装到集群置备的基础架构上，则要在安装过程中将 RHCOS 镜像下载到目标平台，并且使用控制 RHCOS 配置的合适 Ignition 配置文件来部署机器。
- 如果将集群安装到您自己管理的基础架构上，则必须遵循安装文档来获取 RHCOS 镜像，生成 Ignition 配置文件，并且使用 Ignition 配置文件来置备机器。

6.1.1. RHCOS 主要功能

下表描述了 RHCOS 操作系统的主要功能：

- **基于 RHEL**：底层操作系统主要由 RHEL 组件构成。支持 RHEL 的相同质量、安全性和控制措施也支持 RHCOS。例如，RHCOS 软件位于 RPM 软件包中，并且每个 RHCOS 系统都以 RHEL 内核以及由 systemd 初始化系统管理的一组服务启动。
- **控制的不可变性**：尽管 RHCOS 包含 RHEL 组件，但它的管理要比默认的 RHEL 安装更加严格。管理从 OpenShift Container Platform 集群远程执行。设置 RHCOS 机器时，您只能修改一些系统设置。这种受控的不变性使 OpenShift Container Platform 可以存储集群中 RHCOS 系统的最新状态，因而始终都能创建额外的机器并根据最新的 RHCOS 配置执行更新。
- **CRI-O 容器运行时**：尽管 RHCOS 包含运行 Docker 所需的 OCI 和 libcontainer 格式容器的功能，但它融合的是 CRI-O 容器引擎，而非 Docker 容器引擎。通过专注于 Kubernetes 平台（例如 OpenShift Container Platform）所需的功能，CRI-O 可以提供与不同 Kubernetes 版本兼容的特定功能。与提供更大功能集的容器引擎相比，CRI-O 对内存的要求更低，且对安全的攻击面更小。目前，CRI-O 是 OpenShift Container Platform 集群中唯一可用的引擎。
- **容器工具程序组**：对于诸如构建、复制和以其他方式管理容器的任务，RHCOS 用一组兼容的容器工具来代替 Docker CLI 工具。podman CLI 工具支持许多容器运行时功能，例如运行、启动、停止、列举和删除容器及容器镜像。skopeo CLI 工具可以复制、身份认证和签名镜像。您可以使用 **crictl** CLI 工具来处理 CRI-O 容器引擎中的容器和 pod。虽然不建议在 RHCOS 中直接使用这些工具，但可以把它们用于调试目的。
- **rpm-ostree 升级**：RHCOS 具有使用 **rpm-ostree** 系统进行事务升级的功能。更新是通过容器镜像交付的，并且是 OpenShift Container Platform 更新过程的一部分。部署之后，拉取、提取容器镜像并将其写入磁盘，然后修改启动加载程序以启动到新版本。机器将以滚动方式重启并进入更新，确保对集群容量的影响最小。
- **Bootupd 固件和 bootloader 更新**：Package manager 和混合系统（例如 **rpm-ostree**）不会更新固件或者 bootloader。通过 **bootupd**，RHCOS 用户现在可以访问跨系统发行、系统分析操作系统更新工具，该工具管理在现代架构中（如 x86_64、ppc64le 和 aarch64）运行的 UEFI 和旧 BIOS 引导模式中的固件和引导更新。
有关如何安装 **bootupd** 的详情，请参考 [使用 bootupd 更新引导装载程序文档](#)。

- **通过 Machine Config Operator 更新**：在 OpenShift Container Platform 中，Machine Config Operator 处理操作系统升级。**rpm-ostree** 以原子单元形式提供升级，不像 **yum** 升级那样单独升级各个软件包。新的操作系统部署在升级过程中进行，并在下次重启时才会生效。如果升级出现问题，则进行一次回滚并重启就能使系统返回到以前的状态。OpenShift Container Platform 中的 RHCOS 升级是在集群更新期间执行的。

对于 RHCOS 系统，**rpm-ostree** 文件系统的布局具有以下特征：

- **/usr** 是操作系统二进制文件和库的存储位置，并且是只读的。我们不支持更改此设置。
- **/etc**、**/boot** 和 **/var** 在系统上是可写的，但只能由 Machine Config Operator 更改。
- **/var/lib/containers** 是用于存储容器镜像的图形存储位置。

6.1.2. 选择如何配置 RHCOS

RHCOS 的设计思想是，在需要最小用户配置的情况下在 OpenShift Container Platform 集群中进行部署。在它的最基本的形式中包括：

- 从置备的基础架构（如 AWS）开始，或者自行置备基础架构。
- 在运行 **openshift-install** 时，在 **install-config.yaml** 文件中提供少量信息，如凭证和集群名称。

由于 OpenShift Container Platform 中的 RHCOS 系统被设计为通过 OpenShift Container Platform 集群来进行全面管理，因此不建议直接修改 RHCOS 机器。为了调试，您可以对 RHCOS 机器集群进行有限的直接访问，但您不应该直接配置 RHCOS 系统。相反，如果您需要在 OpenShift Container Platform 节点上添加或更改功能，请考虑采用以下方式进行更改：

- **Kubernetes 工作负载对象（DaemonSet、Deployment 等）**：如果要为服务或其他用户级别功能添加到集群中，请考虑将它们添加为 Kubernetes 工作负载对象。为了减少在后续的升级中破坏集群的风险，在特定节点配置之外应用这些功能是最佳方法。
- **第二天自定义配置任务**：如果可能，在不对集群节点进行任何自定义配置的情况下启动集群，并在集群启动后再进行必要的节点更改。这些更改可以更轻松地管理，且不太可能破坏以后的更新。创建机器配置或修改 Operator 自定义资源是进行这些自定义的方法。
- **第一天的自定义配置**：对于集群首次启动时必须执行的自定义配置，可以使用以下方法修改集群以便更改可在首次引导时就生效。第一天自定义配置可通过运行 **openshift-install** 期间的 Ignition 配置和清单文件实现，也可以在由用户置备的 ISO 安装过程中添加引导选项实现。

以下是您可以在第 1 天进行定制的示例：

- **内核参数**：如果在集群首次引导时需要特定的内核功能或调整。
- **磁盘加密**：如果您的安全规则需要对节点上的根文件系统加密，比如 FIPS 支持。
- **内核模块**：如果 Linux 内核没有默认为某个特定的硬件设备（如网卡或视频卡）提供可用的模块。
- **Chronyd**：如果要为节点提供特定的时钟设置，如时间服务器的位置。

要完成这些任务，您可以为 **openshift-install** 提供参数，使其包含额外的对象，如 **MachineConfig**。这些创建机器配置步骤可在集群启动后传递给 Machine Config Operator。



注意

- 安装程序生成的 Ignition 配置文件包含在 24 小时后过期的证书，然后在过期时进行续订。如果在更新证书前关闭集群，且集群在 24 小时后重启，集群会自动恢复过期的证书。一个例外情况是，您需要手动批准待处理的 **node-bootstrapper** 证书签名请求 (CSR) 来恢复 kubelet 证书。如需更多信息，请参阅 *从过期的 control plane 证书中恢复* 的文档。
- 建议您在生成 12 小时后使用 Ignition 配置文件，因为集群安装后 24 小时证书从 16 小时轮转至 22 小时。通过在 12 小时内使用 Ignition 配置文件，您可以避免在安装过程中运行证书更新时避免安装失败。

6.1.3. 选择如何配置 RHCOS

OpenShift Container Platform 的 RHCOS 安装的区别取决于，您是在安装程序置备的基础架构上部署，还是在由用置备的基础架构上部署：

- **安装程序置备**：有些云环境提供预配置的基础架构，以便您使用最小配置来启动 OpenShift Container Platform 集群。对于这些类型的安装，您可以提供 Ignition 配置来在每个节点上放置内容，以便在集群首次启动时可用。
- **用户置备**：如果您置备自己的基础架构，则具有更大的灵活性来向 RHCOS 节点中添加内容。例如：引导 RHCOS ISO 安装程序时您可以添加内核参数来安装每个系统。但是，在多数情况下，如果操作系统本身需要配置，最好通过 Ignition 配置来提供那些配置。

Ignition 工具仅在首次设置 RHCOS 系统时运行。之后，可以使用机器配置来提供 Ignition 配置。

6.1.4. 关于 Ignition

Ignition 是 RHCOS 在初始配置期间用于操作磁盘的实用程序。它可完成常见的磁盘任务，如分区磁盘、格式化分区、写入文件和配置用户等。首次启动时，Ignition 从安装介质或您指定的位置读取其配置，并将配置应用到机器。

无论您要安装集群还是向其中添加机器，Ignition 始终都执行 OpenShift Container Platform 集群机器的初始配置。实际的系统设置大多都在每台机器上进行。对于每台机器，Ignition 都会获取 RHCOS 镜像并启动 RHCOS 内核。内核命令行上的选项标识部署的类型，以及启用了 Ignition 的初始 Ram 磁盘 (initramfs) 的位置。

6.1.4.1. Ignition 工作方式

要使用 Ignition 创建机器，需要 Ignition 配置文件。OpenShift Container Platform 安装程序创建部署集群所需的 Ignition 配置文件。这些文件基于您直接提供给安装程序或通过 **install-config.yaml** 文件提供的信息。

Ignition 配置机器的方式类似于 [cloud-init](#) 或 Linux Anaconda [kickstart](#) 等工具配置系统的方式，但有一些重要的区别：

- Ignition 从一个初始 RAM 磁盘运行，该磁盘与您要安装到的系统相隔离。因此，Ignition 可以重新分区磁盘、设置文件系统，以及对机器的持久文件系统执行其他更改。与之相反，cloud-init 会在系统启动时作为机器的初始系统的一部分运行，因而不易对磁盘分区之类的事项进行基本的更改。使用 cloud-init 时，难以在节点启动期间重新配置启动过程。
- Ignition 旨在初始化系统，而不是更改现有系统。机器完成初始化且内核在安装的系统上运行之后，OpenShift Container Platform 集群中的 Machine Config Operator 将完成所有后续的机器配置。

- Ignition 不是完成一组定义的操作，而是实施声明性配置。它会在新机器启动之前检查所有分区、文件、服务和其他项目是否就位。然后进行更改，例如将必要的文件复制到磁盘，以便新机器符合指定的配置。
- 在 Ignition 完成机器配置之后，内核将继续运行，但会丢弃初始 RAM 磁盘，并转至磁盘上已安装的系统。所有新的系统服务和其他功能都将启动，无需重启系统。
- 因为 Ignition 会确认所有新机器是否都符合声明的配置，所以不会存在配置不全的机器。如果机器设置失败，则初始化过程不会完成，而且 Ignition 也不会启动新机器。您的集群永不会包含配置不全的机器。如果 Ignition 无法完成，机器就不会添加到集群中。您必须添加新的机器。一些失败配置任务的结果可能一直不为人所知，直到后来依赖它的某些事物也失败才被发现。对这类问题的故障排除将会非常困难。而 Ignition 配置的工作方式可以防止出现这个问题。
- 如果 Ignition 配置存在问题，而导致一台机器的设置失败，Ignition 不会尝试使用相同的配置来设置另一台机器。例如，故障可能源自于某一个 Ignition 配置，而构成该配置的父级和子级配置都希望创建同一个文件。在这种情况下，出现的故障将导致 Ignition 配置无法再次用于设置其他机器，直到问题解决为止。
- 如果您有多个 Ignition 配置文件，您可获得该组配置的并集。由于 Ignition 是声明性的，配置之间的冲突可能会导致 Ignition 无法设置机器。这些文件中信息的次序无关紧要。Ignition 将以最有效的方式对每项设置进行分类和实施。例如，如果一个文件需要有多个层级深的目录，而另一个文件需要其路径上的某一目录，则首先创建后一个文件。Ignition 按深度排序并创建所有文件、目录和链接。
- 因为 Ignition 可以从全空的硬盘开始，所以它可以做 cloud-init 不能做的任务：从头开始在裸机上设置系统（使用 PXE 启动等功能）。在裸机情形中，Ignition 配置注入启动分区，以便 Ignition 可以找到它并正确配置系统。

6.1.4.2. Ignition 操作序列

OpenShift Container Platform 集群中 RHCOS 机器的 Ignition 过程包括以下步骤：

- 机器获取其 Ignition 配置文件。control plane 机器（也称为 master 机器）从 bootstrap 机器获取 Ignition 配置文件，worker 机器从一个 control plane 获取 Ignition 配置文件。
- Ignition 在机器上创建磁盘分区、文件系统、目录和链接。它支持 RAID 阵列，但不支持 LVM 卷。
- Ignition 将持久文件系统的根目录挂载到 initramfs 中的 **/sysroot** 目录，然后开始在 **/sysroot** 中工作。
- Ignition 配置所有定义的文件系统，并将它们设置为在运行时进行相应地挂载。
- Ignition 运行 **systemd** 临时文件，将必要的文件填充到 **/var** 目录。
- Ignition 运行 Ignition 配置文件，以设置用户、systemd 单元文件和其他配置文件。
- Ignition 卸载 initramfs 中挂载的持久系统中的所有组件。
- Ignition 启动新机器的初始化过程，该过程再启动系统启动期间在机器上运行的所有其他服务。

然后，机器便已准备好加入集群，不需要重启。

6.2. 查看 IGNITION 配置文件

要查看用于部署 bootstrap 机器的 Ignition 配置文件，请运行以下命令：

```
$ openshift-install create ignition-configs --dir $HOME/testconfig
```

回答几个问题后，您所在的目录中将出现 **bootstrap.ign**、**master.ign** 和 **worker.ign** 文件。

要查看 **bootstrap.ign** 文件的内容，请通过 **jq** 过滤器对其进行管道传递。以下是该文件的片段：

```
$ cat $HOME/testconfig/bootstrap.ign | jq
\{
  "ignition": \{
    "config": \{,
    "storage": \{
      "files": [
        \{
          "filesystem": "root",
          "path": "/etc/motd",
          "user": \{
            "name": "root"
          },
          "append": true,
          "contents": \{
            "source": "data:text/plain;charset=utf-
8;base64,VGhpcyBpcyB0aGUgYm9vdHN0cmFwIG5vZGU7IGl0IHdpbGwgYmUgZGVzdHJveWVklHdo
ZW4gdGhlIG1hc3RlciBpcyBmdWxseSB1cC4KCIRoZSBwcmItYXJ5J5IHNIcnZpY2UgaXMglmJvb3RrdWJl
nNlcnZpY2UiLiBUbyB3YXRjaCBpdHMgc3RhdHVzLCBydW4gZS5nLgoKICBqb3VybmFsY3RslC1iIC1m
IC11IGJvb3RrdWJlLnNlcnZpY2UK",
```

要解码 **bootstrap.ign** 文件中列出的文件内容，请将代表该文件内容的 base64 编码的数据字符串通过管道传递给 **base64 -d** 命令。以下示例使用了上方输出中添加至 bootstrap 机器的 **/etc/motd** 文件的内容：

```
$ echo
VGhpcyBpcyB0aGUgYm9vdHN0cmFwIG5vZGU7IGl0IHdpbGwgYmUgZGVzdHJveWVklHdoZW4gdG
hlIG1hc3RlciBpcyBmdWxseSB1cC4KCIRoZSBwcmItYXJ5J5IHNIcnZpY2UgaXMglmJvb3RrdWJlLnNlcnZp
Y2UiLiBUbyB3YXRjaCBpdHMgc3RhdHVzLCBydW4gZS5nLgoKICBqb3VybmFsY3RslC1iIC1mIC11IGJ
vb3RrdWJlLnNlcnZpY2UK | base64 --decode
```

输出示例

```
This is the bootstrap machine; it will be destroyed when the master is fully up.
```

```
The primary service is "bootkube.service". To watch its status, run, e.g.:
```

```
journalctl -b -f -u bootkube.service
```

对 **master.ign** 和 **worker.ign** 文件重复这些命令，查看每种机器类型的 Ignition 配置文件的来源。对于 **worker.ign**，您应该会看到类似于下面这一行，它确认了如何从 bootstrap 获取 Ignition 配置：

```
"source": "https://api.myign.develcluster.example.com:22623/config/worker",
```

您可以从 **bootstrap.ign** 文件中了解到以下内容：

- 格式：文件的格式在 [Ignition 配置规格](#) 中定义。MCO 稍后使用相同格式的文件，将更改合并到机器的配置中。

- 内容：由于 bootstrap 机器为其他机器提供 Ignition 配置，因此 master 机器和 worker 机器的 Ignition 配置信息都与 bootstrap 机器的配置一起存储在 **bootstrap.ign** 中。
- 大小：文件长度超过 1300 行，包含指向各种资源的路径。
- 要复制到机器的每个文件的内容实际上编码为数据 URL，这往往会使内容读起来有些混乱。（使用前面演示的 jq 和 base64 命令可使内容更易读。）
- 配置：Ignition 配置文件的不同部分通常涵盖刚放入机器文件系统上的文件，而不是用于修改现有文件的命令。例如，不添加与配置该服务的 NFS 相关的一节，是仅添加一个 NFS 配置文件，然后在系统启动时由 init 进程启动该文件。
- 用户：创建一个名为 core 的用户，并将您的 ssh 密钥分配给该用户。这样，您可以使用该用户名和凭证来登录集群。
- 存储：存储部分标识添加到每台机器的文件。一些值得注意的文件包括 **/root/.docker/config.json**（提供集群从容器镜像 registry 表拉取时所需的凭证），以及 **/opt/openshift/manifests** 中用于配置集群的一系列清单文件：
- systemd：systemd 部分包含用于创建 systemd 单元文件的内容。这些文件用于在启动时启动服务，还用于在运行中的系统上管理这些服务。
- 原语：Ignition 还公开低级别原语，其他工具可以此为基础进行构建。

6.3. 安装后更改 IGNITION 配置

机器配置池管理节点集群及其相应机器配置。机器配置包含集群的配置信息。列出所有已知的机器配置池：

```
$ oc get machineconfigpools
```

输出示例

```
NAME CONFIG          UPDATED UPDATING DEGRADED
master master-1638c1aea398413bb918e76632f20799 False False False
worker worker-2feef4f8288936489a5a832ca8efe953 False False False
```

列出所有机器配置：

```
$ oc get machineconfig
```

输出示例

```
NAME                                     GENERATEDBYCONTROLLER IGNITIONVERSION  CREATED
OSIMAGEURL
00-master                               4.0.0-0.150.0.0-dirty 3.1.0            16m
00-master-ssh                           4.0.0-0.150.0.0-dirty 3.1.0            16m
00-worker                               4.0.0-0.150.0.0-dirty 3.1.0            16m
00-worker-ssh                           4.0.0-0.150.0.0-dirty 3.1.0            16m
01-master-kubelet                       4.0.0-0.150.0.0-dirty 3.1.0            16m
01-worker-kubelet                       4.0.0-0.150.0.0-dirty 3.1.0            16m
master-1638c1aea398413bb918e76632f20799 4.0.0-0.150.0.0-dirty 3.1.0            16m
worker-2feef4f8288936489a5a832ca8efe953 4.0.0-0.150.0.0-dirty 3.1.0            16m
```

在涉及到应用这些机器配置时，Machine Config Operator 的行为与 Ignition 有些不同。机器配置按顺序读取（从 00* 到 99*）。机器配置中的标签标识每个所用于的节点类型（master 节点或 worker 节点）。如果同一文件出现在多个机器配置文件中，则以最后一个文件为准。例如，出现在 99* 文件中的任何文件都将替换出现在 00* 文件中的同一文件。输入的 **MachineConfig** 对象将合并为一个“呈现的”**MachineConfig** 对象，该对象将被 Operator 用作目标，也是您可以在 MachineConfigPool 中看到的值。

要查看从机器配置中管理的文件，请查找特定 **MachineConfig** 对象中的"Path:"。例如：

```
$ oc describe machineconfigs 01-worker-container-runtime | grep Path:
```

输出示例

```
Path:      /etc/containers/registries.conf
Path:      /etc/containers/storage.conf
Path:      /etc/crio/crio.conf
```

确保为机器指定一个较新的名称（如 10-worker-container-runtime）。请记住，每个文件的内容都是 URL 样式的数据。然后将新机器配置应用到集群。

第 7 章 CI/CD 方法和实践

使用 *持续的集成/持续提供* (CI/CD) 方法，您可以通过将自动化融入到应用程序开发阶段，从集成和测试阶段到交付和部署阶段，定期向客户提供应用程序。CI/CD 进程通常被称为 "CI/CD pipeline"。CI/CD 的主要概念是持续集成、持续提供和连续部署。

7.1. 用于集群管理和应用程序配置管理的 CI/CD

持续集成 是开发人员自动化的过程。应用程序的代码更改会定期构建、测试并合并到共享的存储库。

持续提供 和 *连续部署* 是有时可互换使用的概念，指的是管道的自动化。持续提供使用自动化，以确保开发人员对应用程序的更改经过测试并发送到软件仓库，操作团队可以将它们部署到产品环境中。持续部署可以实现更改的发行，从存储库开始，并以生产结束。持续部署会加快应用程序的交付使用的速度，并可减少操作团队的工作量。

7.2. GITOPS 方法和实践

GitOps 是一组使用 Git 拉取请求来管理基础架构和应用程序配置的实践。*GitOps* 中的 Git 存储库是系统和应用程序配置的唯一来源。该程序库包含该系统的完整状态，以便可查看并可审核更改到系统状态。*GitOps* 允许您实施 *DevOps* 方法。

您可以使用 *GitOps* 工具来创建可重复和可预测的进程来管理和重新创建 OpenShift Container Platform 集群和应用程序。通过使用 *GitOps*，您可以处理基础架构和应用程序配置 *sprawl* 的问题。它通过将您的基础架构和应用程序定义为“代码”，简化了跨多个群集的基础架构和应用程序配置的传播。为集群配置文件实现 *GitOps* 可方便自动安装，并可让您配置自动化集群自定义。您可以将在 Git 存储库中开发和维护软件的核心原则应用到创建和管理集群和应用程序配置文件。

通过使用 OpenShift Container Platform 自动执行集群配置和容器开发过程，您可以选择在哪里和何时执行 *GitOps* 操作。使用与您的 *GitOps* 策略和执行计划相对应的 CI 管道是最理想的选择。OpenShift Container Platform 提供了选择何时和如何将这个方法整合到您的业务实践和管道中的灵活性。

通过 *GitOps* 集成，可以声明性地配置和存储 OpenShift Container Platform 集群配置

GitOps 可在 OpenShift Container Platform 中正常工作，因为您可以声明性地配置集群并将集群配置状态存储在 Git 中。如需更多信息，请参阅 [可用的集群自定义](#)。

7.2.1. *GitOps* 用于单集群和多集群管理

无论您是使用一个或多个独立或相互合作的 OpenShift Container Platform 集群，都可以使用 *GitOps* 策略来管理以下任务：

- 确保集群具有类似的配置、监控或存储状态。
- 从已知状态恢复或重新创建集群。
- 使用已知状态创建集群。
- 对多个 OpenShift Container Platform 集群应用或恢复配置更改。
- 将模板配置与不同环境关联。

7.2.2. 用于应用程序配置管理的 *GitOps*

您还可以使用 GitOps 实践来管理应用程序配置。当您将应用程序部署到不同环境中的不同集群时（如开发、调试和生产环境），这种实践可确保应用程序的一致性。当您必须在多个集群间部署应用程序（在云或内部部署），为了实现可用性和可伸缩性的目的，使用 GitOps 管理应用程序配置也很有用处。

您可以使用 GitOps 策略来：

- 在集群间（从调试到生产阶段）推广应用程序。
- 对多个 OpenShift Container Platform 集群应用或恢复应用程序更改。

7.2.3. GitOps 技术供应商和集成商

提供了多个与 OpenShift Container Platform 高度集成的社区建议和第三方供应商。

您可以将 GitOps 与以下社区合作伙伴和第三方集成到 OpenShift Container Platform 中：

- [ArgoCD](#)

第 8 章 在 OPENSIFT CONTAINER PLATFORM 中使用 ARGOC CD

8.1. ARGOC CD 做什么？

ArgoCD 是一个声明式连续交付工具，它利用 GitOps 来维护集群资源。ArgoCD 是作为控制器实现的，可持续监控 Git 存储库中定义的应用程序定义和配置，并将这些配置的指定状态与集群中的实时状态进行比较。和 Git 存储库中指定状态不同的配置被归类为 OutOfSync。ArgoCD 报告这些不同之处，并允许管理员自动或者手动将配置重新同步到定义的状态。

ArgoCD 可让您提供全局自定义资源，如用于配置 OpenShift Container Platform 集群的资源。

8.2. 支持声明

红帽不提供对这个工具的支持。要获得对 ArgoCD 的支持，请参阅 ArgoCD 文档的[支持信息](#)。

8.3. ARGOC CD 文档

有关使用 ArgoCD 的更多信息，请参见 [ArgoCD 文档](#)。

第 9 章 准入插件

9.1. 关于准入插件

准入 (Admission) 插件用于帮助规范 OpenShift Container Platform 4.6 的功能。在请求被验证并授权后，准入插件截取到主 API 的请求，验证资源请求以确保符合相关的策略。例如，它们通常会被用来强制执行安全策略、资源限制或配置要求。

准入插件以准入链的形式按序列运行。如果序列中的任何准入插件拒绝某个请求，则整个链将中止并返回错误。

OpenShift Container Platform 为每个资源类型都启用了默认的准入插件。这些是集群正常工作所需要的。准入插件会忽略那些不由它们负责的资源。

除了默认的插件外，准入链还可以通过调用自定义 webhook 服务器的 webhook 准入插件动态进行扩展。webhook 准入插件有两种：变异准入插件和验证准入插件。变异准入插件会首先运行，它可以修改资源并验证请求。验证准入插件会验证请求，并在变异准入插件之后运行，以便由变异准入插件触发的改变也可以被验证。

通过一个变异准入插件调用 webhook 服务器可能会对与目标对象相关的资源产生副作用。在这种情况下，必须执行相应的步骤来验证最终结果是正确的。



警告

应谨慎使用动态准入机制，因为它会影响到集群的 control plane 操作。在 OpenShift Container Platform 4.6 中使用通过 webhook 准入插件调用 webhook 服务器的功能时，请确保已仔细阅读了相关文档，并对变异所带来的副作用进行了测试。需要包括一个步骤，以便在请求没有通过整个准入链时，把资源恢复到其原始状态。

9.2. 默认准入插件

OpenShift Container Platform 4.6 中启用了默认的验证和准入插件。这些默认插件可实现基本的 control plane 功能，如入口策略、集群资源限制覆盖和配额策略。以下列表包含默认准入插件：

例 9.1. 验证准入插件

- **LimitRanger**
- **ServiceAccount**
- **PodNodeSelector**
- 优先级
- **PodTolerationRestriction**
- **OwnerReferencesPermissionEnforcement**
- **PersistentVolumeClaimResize**

- **RuntimeClass**
- **CertificateApproval**
- **CertificateSigning**
- **CertificateSubjectRestriction**
- **autoscaling.openshift.io/ManagementCPUsOverride**
- **authorization.openshift.io/RestrictSubjectBindings**
- **scheduling.openshift.io/OriginPodNodeEnvironment**
- **network.openshift.io/ExternalIPRanger**
- **network.openshift.io/RestrictedEndpointsAdmission**
- **image.openshift.io/ImagePolicy**
- **security.openshift.io/SecurityContextConstraint**
- **security.openshift.io/SCCExecRestrictions**
- **route.openshift.io/IngressAdmission**
- **config.openshift.io/ValidateAPIServer**
- **config.openshift.io/ValidateAuthentication**
- **config.openshift.io/ValidateFeatureGate**
- **config.openshift.io/ValidateConsole**
- **operator.openshift.io/ValidateDNS**
- **config.openshift.io/ValidateImage**
- **config.openshift.io/ValidateOAuth**
- **config.openshift.io/ValidateProject**
- **config.openshift.io/DenyDeleteClusterConfiguration**
- **config.openshift.io/ValidateScheduler**
- **quota.openshift.io/ValidateClusterResourceQuota**
- **security.openshift.io/ValidateSecurityContextConstraints**
- **authorization.openshift.io/ValidateRoleBindingRestriction**
- **config.openshift.io/ValidateNetwork**
- **operator.openshift.io/ValidateKubeControllerManager**
- **ValidatingAdmissionWebhook**

- ResourceQuota
- `quota.openshift.io/ClusterResourceQuota`

例 9.2. 变异准入插件

- NamespaceLifecycle
- LimitRanger
- ServiceAccount
- NodeRestriction
- TaintNodesByCondition
- PodNodeSelector
- 优先级
- DefaultTolerationSeconds
- PodTolerationRestriction
- PersistentVolumeLabel
- DefaultStorageClass
- StorageObjectInUseProtection
- RuntimeClass
- DefaultIngressClass
- `autoscaling.openshift.io/ManagementCPUsOverride`
- `scheduling.openshift.io/OriginPodNodeEnvironment`
- `image.openshift.io/ImagePolicy`
- `security.openshift.io/SecurityContextConstraint`
- `security.openshift.io/DefaultSecurityContextConstraints`
- MutatingAdmissionWebhook

9.3. WEBHOOK 准入插件

除了 OpenShift Container Platform 默认准入插件外，也可以通过称为 webhook 服务器的 webhook 准入插件来实施动态准入，从而扩展准入链的功能。Webhook 服务器通过 HTTP 在定义的端点调用。

OpenShift Container Platform 中有两个类型的 webhook 准入插件：

- 在准入过程中，变异 (*mutating*) 准入插件可以执行任务，如注入关联性标签。

- 在准入过程结束时，验证 (validating) 准入插件会用来确保对象被正确配置。例如，确保关联性标签与预期一样。如果验证通过，OpenShift Container Platform 会按照配置来调度对象。

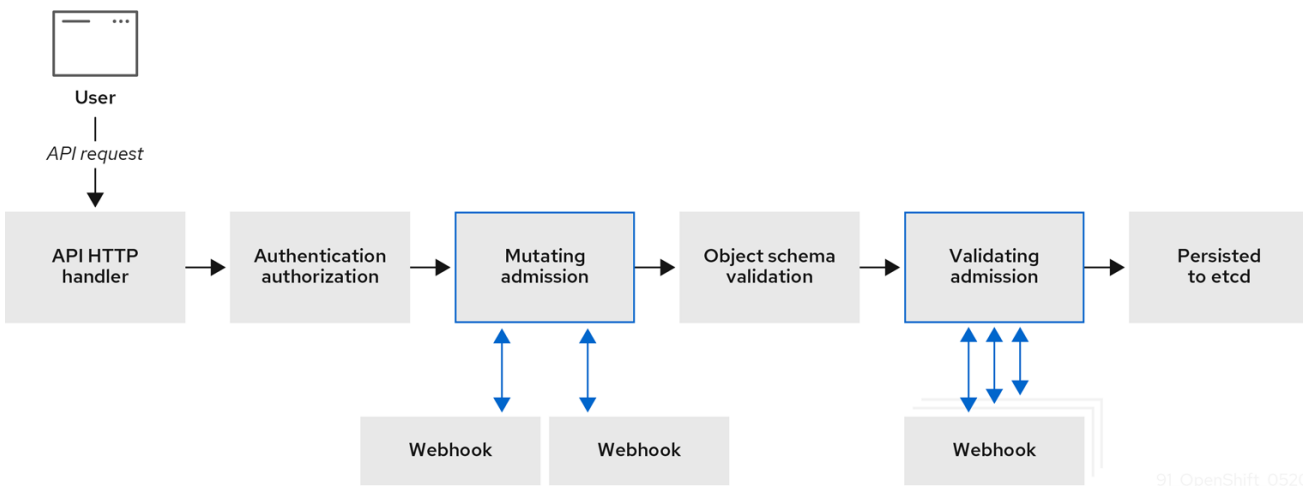
当 API 请求到来时，变异或验证准入插件会使用配置中的外部 webhook 列表,并并行调用它们：

- 如果所有 webhook 都批准请求，准入链将继续。
- 如果任何 webhook 拒绝了请求，则拒绝准入请求。而拒绝的原因是第一个拒绝的原因。
- 如果多个 webhook 拒绝准入请求，则只有第一个拒绝原因返回给用户。
- 如果在调用 webhook 时遇到错误，请求将被拒绝。或者根据错误策略的设置，webhook 会被忽略。如果错误策略被设置为 **Ignore**，则失败时请求将被无条件接受。如果策略被设置为 **Fail**，失败时请求将被拒绝。使用 **Ignore** 可能会为所有客户端造成无法预计的行为。

webhook 准入插件和 webhook 服务器之间的通信必须使用 TLS。生成一个 CA 证书，使用该证书为您的 webhook 准入服务器使用的服务器证书签名。PEM 编码的 CA 证书使用某种机制（如 service serving 证书 secret）提供给 webhook 准入插件。

下图演示了调用多个 webhook 服务器的序列准入链进程。

图 9.1. 带有变异准入插件和验证准入插件的 API 准入链



webhook 准入插件用例示例。这个示例中，所有 pod 都必须具有一组通用标签。在本例中，变异准入插件可以注入标签，验证准入插件可以检查标签是否如预期。之后，OpenShift Container Platform 可以调度包含所需标签的 pod，并拒绝那些没有所需标签的 pod。

Webhook 准入插件常见使用案例包括：

- 命名空间保留。
- 限制由 SR-IOV 网络设备插件管理的自定义网络资源。
- 定义可启用污点以决定哪些 pod 应该调度到节点上的容限。
- Pod 优先级类验证。

9.4. WEBHOOK 准入插件类型

集群管理员可以通过 API 服务器准入链中的变异准入插件或验证准入插件调用 webhook 服务器。

9.4.1. 变异准入插件

变异（mutating）准入插件在准入过程的变异期间被调用，这允许在保留资源内容前修改资源内容。一个可通过变异准入插件调用的 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。

变异准入插件配置示例

```

apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration ❶
metadata:
  name: <webhook_name> ❷
webhooks:
- name: <webhook_name> ❸
  clientConfig: ❹
    service:
      namespace: default ❺
      name: kubernetes ❻
      path: <webhook_url> ❼
    caBundle: <ca_signing_certificate> ❽
  rules: ❾
  - operations: ❿
    - <operation>
    apiGroups:
      - ""
    apiVersions:
      - "*"
    resources:
      - <resource>
  failurePolicy: <policy> ⓫
  sideEffects: None

```

❶ 指定一个变异准入插件配置。

❷ **MutatingWebhookConfiguration** 对象的名称。将 **<webhook_name>** 替换为适当的值。

❸ 要调用的 webhook 的名称。将 **<webhook_name>** 替换为适当的值。

❹ 如何连接、信任和将数据发送到 webhook 服务器的信息。

❺ 创建前端服务的命名空间。

❻ 前端服务的名称。

❼ 用于准入请求的 webhook URL。将 **<webhook_url>** 替换为适当的值。

❽ 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。将 **<ca_signing_certificate>** 替换为采用 base64 格式的适当证书。

❾ 定义 API 服务器何时应使用此 webhook 准入插件的规则。

❿ 一个或多个触发 API 服务器调用此 webhook 准入插件的操作。可能的值包括 **create**、**update**、**delete** 或 **connect**。将 **<operation>** 和 **<resource>** 替换为适当的值。

⓫ 指定如果 webhook 服务器不可用时策略应如何执行。将 **<policy>** 替换为 **Ignore**（在失败时无条件

接受请求) 或 **Fail** (拒绝失败的请求)。使用 **Ignore** 可能会为所有客户端造成无法预计的行为。



重要

在 OpenShift Container Platform 4.6 中，由用户创建或通过变异准入插件的控制循环创建的对象可能会返回非预期的结果，特别是在初始请求中设置的值被覆盖时，我们不建议这样做。

9.4.2. 验证准入插件

在准入过程的验证阶段会调用验证准入插件。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 也是一个 webhook 示例，它由验证准入插件调用，以确保所有 **nodeSelector** 字段均受命名空间的节点选择器限制。

验证准入插件配置示例

```

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

```

- 1** 指定验证准入插件配置。
- 2** **ValidatingWebhookConfiguration** 对象的名称。将 **<webhook_name>** 替换为适当的值。
- 3** 要调用的 webhook 的名称。将 **<webhook_name>** 替换为适当的值。
- 4** 如何连接、信任和将数据发送到 webhook 服务器的信息。
- 5** 创建前端服务的命名空间。
- 6** 前端服务的名称。
- 7** 用于准入请求的 webhook URL。将 **<webhook_url>** 替换为适当的值。

- 8 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。将 `<ca_signing_certificate>` 替换为采用 base64 格式的适当证书。
- 9 定义 API 服务器何时应使用此 webhook 准入插件的规则。
- 10 一个或多个触发 API 服务器调用此 webhook 准入插件的操作。可能的值包括 **create**、**update**、**delete** 或 **connect**。将 `<operation>` 和 `<resource>` 替换为适当的值。
- 11 指定如果 webhook 服务器不可用时策略应如何执行。将 `<policy>` 替换为 **Ignore**（在失败时无条件接受请求）或 **Fail**（拒绝失败的请求）。使用 **Ignore** 可能会为所有客户端造成无法预计的行为。

9.5. 配置动态准入

此流程概述了配置动态准入的高级步骤。通过配置 webhook 准入插件来调用 webhook 服务器以扩展准入链的功能。

webhook 服务器也被配置为一个聚合的 API 服务器。这允许其他 OpenShift Container Platform 组件使用内部凭证与 webhook 通信，并可使用 **oc** 命令进行测试。另外，这还可在 webhook 中启用基于角色的访问控制（RBAC），并防止向 webhook 公开其他 API 服务器的令牌信息。

先决条件

- 一个具有集群管理员权限的 OpenShift Container Platform 账户。
- 已安装 OpenShift Container Platform CLI (**oc**)。
- 公布的 webhook 服务器容器镜像。

流程

1. 使用镜像 registry 构建 webhook 服务器容器镜像，并将其提供给集群。
2. 创建本地 CA 密钥和证书，并使用它们为 webhook 服务器的证书签名请求（CSR）签名。
3. 为 webhook 资源创建新项目：

```
$ oc new-project my-webhook-namespace 1
```

- 1 请注意，webhook 服务器可能会需要一个特定的名称。

4. 在名为 **rbac.yaml** 的文件中为聚合的 API 服务定义 RBAC 规则：

```
apiVersion: v1
kind: List
items:
- apiVersion: rbac.authorization.k8s.io/v1 1
  kind: ClusterRoleBinding
  metadata:
    name: auth-delegator-my-webhook-namespace
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
```

```
  name: system:auth-delegator
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server

- apiVersion: rbac.authorization.k8s.io/v1 2
  kind: ClusterRole
  metadata:
    annotations:
      name: system:openshift:online:my-webhook-server
  rules:
  - apiGroups:
    - online.openshift.io
    resources:
    - namespaces 3
    verbs:
    - get
    - list
    - watch

- apiVersion: rbac.authorization.k8s.io/v1 4
  kind: ClusterRole
  metadata:
    name: system:openshift:online:my-webhook-requester
  rules:
  - apiGroups:
    - admission.online.openshift.io
    resources:
    - namespaces 5
    verbs:
    - create

- apiVersion: rbac.authorization.k8s.io/v1 6
  kind: ClusterRoleBinding
  metadata:
    name: my-webhook-server-my-webhook-namespace
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
    name: system:openshift:online:my-webhook-server
  subjects:
  - kind: ServiceAccount
    namespace: my-webhook-namespace
    name: server

- apiVersion: rbac.authorization.k8s.io/v1 7
  kind: RoleBinding
  metadata:
    namespace: kube-system
    name: extension-server-authentication-reader-my-webhook-namespace
  roleRef:
    kind: Role
    apiGroup: rbac.authorization.k8s.io
    name: extension-apiserver-authentication-reader
```

```

subjects:
- kind: ServiceAccount
  namespace: my-webhook-namespace
  name: server

- apiVersion: rbac.authorization.k8s.io/v1 8
  kind: ClusterRole
  metadata:
    name: my-cluster-role
  rules:
- apiGroups:
  - admissionregistration.k8s.io
  resources:
  - validatingwebhookconfigurations
  - mutatingwebhookconfigurations
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - ""
  resources:
  - namespaces
  verbs:
  - get
  - list
  - watch

- apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRoleBinding
  metadata:
    name: my-cluster-role
  roleRef:
    kind: ClusterRole
    apiGroup: rbac.authorization.k8s.io
    name: my-cluster-role
  subjects:
- kind: ServiceAccount
  namespace: my-webhook-namespace
  name: server

```

- 1 将身份验证和授权委托给 webhook 服务器 API。
- 2 允许 webhook 服务器访问集群资源。
- 3 指向资源。在这个示例中指向 **namespacereservations** 资源。
- 4 启用聚合的 API 服务器创建准入审核。
- 5 指向资源。在这个示例中指向 **namespacereservations** 资源。
- 6 启用 webhook 服务器访问集群资源。
- 7 角色绑定来读取终止身份验证的配置。
- 8 聚合的 API 服务器的默认集群角色和集群角色绑定。

5. 将这些 RBAC 规则应用到集群：

```
$ oc auth reconcile -f rbac.yaml
```

6. 创建名为 **webhook-daemonset.yaml** 的 YAML 文件，用于将 webhook 部署为命名空间中的守护进程设置服务器：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: my-webhook-namespace
  name: server
  labels:
    server: "true"
spec:
  selector:
    matchLabels:
      server: "true"
  template:
    metadata:
      name: server
      labels:
        server: "true"
    spec:
      serviceAccountName: server
      containers:
        - name: my-webhook-container ❶
          image: <image_registry_username>/<image_path>:<tag> ❷
          imagePullPolicy: IfNotPresent
          command:
            - <container_commands> ❸
          ports:
            - containerPort: 8443 ❹
          volumeMounts:
            - mountPath: /var/serving-cert
              name: serving-cert
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8443 ❺
              scheme: HTTPS
          volumes:
            - name: serving-cert
              secret:
                defaultMode: 420
                secretName: server-serving-cert
```

- ❶ 请注意，webhook 服务器可能会预期有一个特定的容器名称。
- ❷ 指向 webhook 服务器容器镜像。将 **<image_registry_username>/<image_path>:<tag>** 替换为适当的值。
- ❸ 指定 webhook 容器运行命令。将 **<container_commands>** 替换为适当的值。
- ❹ 定义 pod 中的目标端口。这个示例使用端口 8443。

- 5 指定就绪探测使用的端口。这个示例使用端口 8443。

7. 部署守护进程集：

```
$ oc apply -f webhook-daemonset.yaml
```

8. 在名为 **webhook-secret.yaml** 的 YAML 文件中为 service serving 证书签名程序定义 secret:

```
apiVersion: v1
kind: Secret
metadata:
  namespace: my-webhook-namespace
  name: server-serving-cert
type: kubernetes.io/tls
data:
  tls.crt: <server_certificate> 1
  tls.key: <server_key> 2
```

- 1 引用签名的 webhook 服务器证书。将 **<server_certificate>** 替换为适当的 base64 格式的证书。
- 2 引用签名的 webhook 服务器密钥。将 **<server_key>** 替换为 base64 格式的适当密钥。

9. 创建 secret：

```
$ oc apply -f webhook-secret.yaml
```

10. 在名为 **webhook-service.yaml** 的 YAML 文件中定义服务帐户和服务：

```
apiVersion: v1
kind: List
items:
- apiVersion: v1
  kind: ServiceAccount
  metadata:
    namespace: my-webhook-namespace
    name: server
- apiVersion: v1
  kind: Service
  metadata:
    namespace: my-webhook-namespace
    name: server
  annotations:
    service.beta.openshift.io/serving-cert-secret-name: server-serving-cert
  spec:
    selector:
      server: "true"
    ports:
      - port: 443 1
        targetPort: 8443 2
```

- 1 定义服务侦听的端口。这个示例使用端口 443。
- 2 定义服务转发连接到的 pod 中的目标端口。这个示例使用端口 8443。

11. 在集群中公开 webhook 服务器：

```
$ oc apply -f webhook-service.yaml
```

12. 在名为 **webhook-crd.yaml** 的文件中为 webhook 服务器定义自定义资源定义：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: namespacereservations.online.openshift.io 1
spec:
  group: online.openshift.io 2
  version: v1alpha1 3
  scope: Cluster 4
  names:
    plural: namespacereservations 5
    singular: namespacereservation 6
    kind: NamespaceReservation 7
```

- 1 反映 **CustomResourceDefinition spec** 值，格式为 **<plural>.<group>**。在这个示例中使用 **namespacereservations** 资源。
- 2 REST API 组名称。
- 3 REST API 版本名称。
- 4 可接受的值是 **Namespaced** 或 **Cluster**。
- 5 URL 中包括的复数名称。
- 6 **oc** 输出中的别名。
- 7 资源清单的引用。

13. 应用自定义资源定义：

```
$ oc apply -f webhook-crd.yaml
```

14. 在名为 **webhook-api-service.yaml** 的文件中配置 webhook 服务器也作为一个聚合的 API 服务器：

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.admission.online.openshift.io
spec:
  caBundle: <ca_signing_certificate> 1
  group: admission.online.openshift.io
```

```

groupPriorityMinimum: 1000
versionPriority: 15
service:
  name: server
  namespace: my-webhook-namespace
version: v1beta1

```

- 1 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。将 `<ca_signing_certificate>` 替换为采用 base64 格式的适当证书。

15. 部署聚合的 API 服务：

```
$ oc apply -f webhook-api-service.yaml
```

16. 在名为 `webhook-config.yaml` 的文件中定义 webhook 准入插件配置。本例使用验证准入插件：

```

apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: namespacesreservations.admission.online.openshift.io 1
webhooks:
- name: namespacesreservations.admission.online.openshift.io 2
  clientConfig:
    service: 3
    namespace: default
    name: kubernetes
    path: /apis/admission.online.openshift.io/v1beta1/namespacereservations 4
    caBundle: <ca_signing_certificate> 5
  rules:
  - operations:
    - CREATE
    apiGroups:
    - project.openshift.io
    apiVersions:
    - "*"
    resources:
    - projectrequests
  - operations:
    - CREATE
    apiGroups:
    - ""
    apiVersions:
    - "*"
    resources:
    - namespaces
failurePolicy: Fail

```

- 1 **ValidatingWebhookConfiguration** 对象的名称。在这个示例中使用 **namespacereservations** 资源。
- 2 要调用的 webhook 的名称。在这个示例中使用 **namespacereservations** 资源。
- 3 通过聚合的 API 启用对 webhook 服务器的访问。

- 4 用于准入请求的 webhook URL。在这个示例中使用 **namespacereservation** 资源。
- 5 为 webhook 服务器使用的服务器证书签名的 PEM 编码的 CA 证书。将 **<ca_signing_certificate>** 替换为采用 base64 格式的适当证书。

17. 部署 webhook :

```
$ oc apply -f webhook-config.yaml
```

18. 验证 webhook 是否如预期运行。例如，如果您配置了动态准入以保留特定的命名空间，请确认创建这些命名空间的请求会被拒绝，并且创建非保留命名空间的请求会成功。

9.6. 其他资源

- [限制由 SR-IOV 网络设备插件管理的自定义网络资源](#)
- [定义可启用污点以决定哪些 pod 应该调度到节点上的容限](#)
- [Pod 优先级类验证](#)