OpenShift Container Platform 4.16

节点

在 OpenShift Container Platform 中配置和管理节点

Last Updated: 2024-07-03

OpenShift Container Platform 4.16 节点

在 OpenShift Container Platform 中配置和管理节点

法律通告

摘要

本文提供有关在集群中配置和管理节点、Pod 和容器的说明。它还提供有关配置 Pod 调度和放置、使用作业（job）和 DaemonSet 来自动执行操作，以及确保集群保持高效性的其他任务信息。

目录

第 1 章 节点概述

1.1. 关于节点

节点是 Kubernetes 集群中的虚拟机或裸机。Worker 节点托管您的应用程序容器，分组为 pod。control plane 节点运行控制 Kubernetes 集群所需的服务。在 OpenShift Container Platform 中，control plane 节点不仅仅包含用于管理 OpenShift Container Platform 集群的 Kubernetes 服务。

在集群中运行稳定和健康的节点是基本运行托管应用程序的基本操作。在 OpenShift Container Platform 中，您可以通过代表节点的 **Node** 对象访问、管理和监控节点。使用 OpenShift CLI(**oc**)或 Web 控制台，您可以在节点上执行以下操作。

节点的以下组件负责维护运行 pod 并提供 Kubernetes 运行时环境。

容器运行时

容器运行时负责运行容器。Kubernetes 提供多个运行时，如 containerd、cri-o、rktlet 和 Docker。

Kubelet

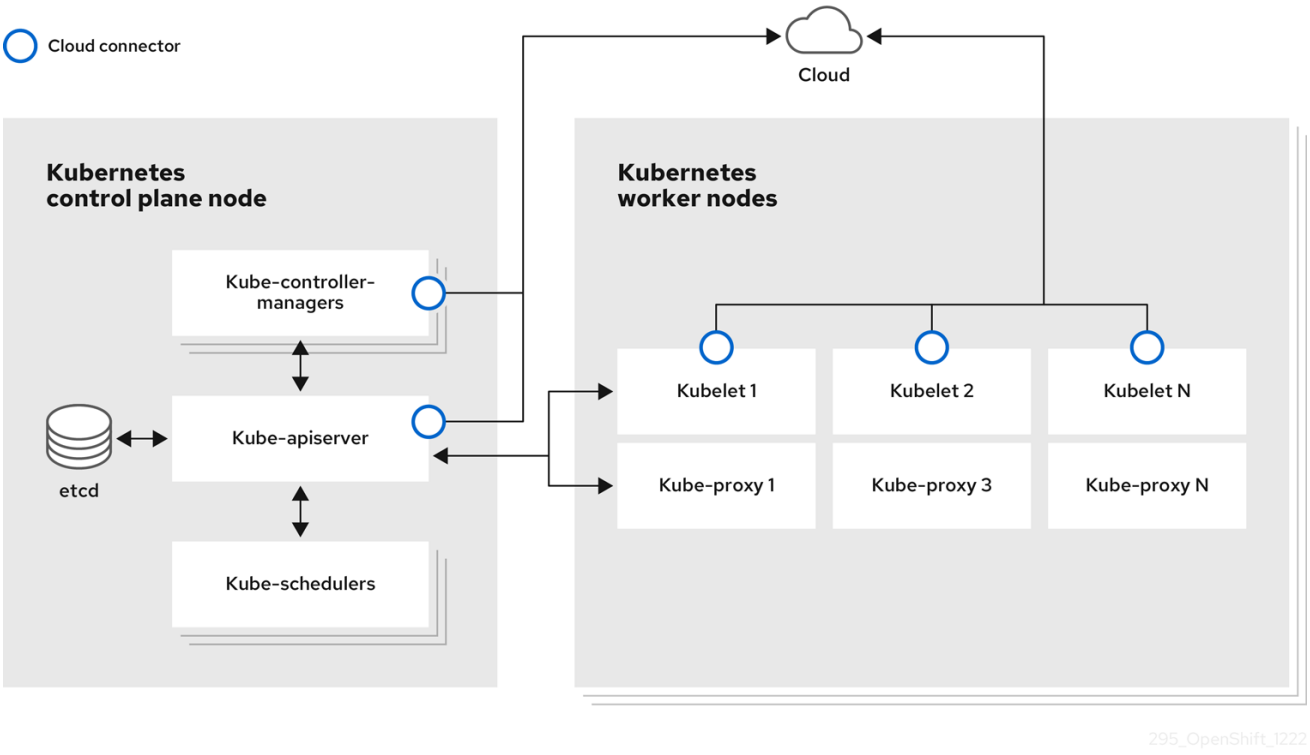
kubelet 在节点上运行并读取容器清单。它确保定义的容器已启动且正在运行。kubelet 进程维护工作和节点服务器的状态。kubelet 管理网络流量和端口转发。kubelet 管理仅由 Kubernetes 创建的容器。

Kube-proxy

kube-proxy 在集群的每个节点上运行，并维护 Kubernetes 资源之间的网络流量。Kube-proxy 可确保网络环境被隔离并可访问。

DNS

集群 DNS 是一个 DNS 服务器，它为 Kubernetes 服务提供 DNS 记录。由 Kubernetes 启动的容器会在其 DNS 搜索中自动包含此 DNS 服务器。



读取操作

通过读操作，管理员可以或开发人员获取 OpenShift Container Platform 集群中节点的信息。

- 列出集群中的所有节点。
- 获取节点的相关信息，如内存和 CPU 使用量、健康、状态和年龄。
- 列出节点上运行的 pod。

管理操作
作为管理员，您可以通过几个任务轻松地在 OpenShift Container Platform 集群中管理节点：

- 添加或更新节点标签。标签是应用于 **Node** 对象的键值对。您可以使用标签来控制 pod 的调度。
- 使用自定义资源定义(CRD)或 **kubeletConfig** 对象更改节点配置。
- 配置节点以允许或禁止调度 pod。具有 **Ready** 状态的健康 worker 节点默认允许 pod 放置，而 control plane 节点没有；您可以通过将 worker 节点配置为不可调度，并将 control plane 节点配置为可以调度。
- 使用 **system-reserved** 设置为节点分配资源。您可以允许 OpenShift Container Platform 自动决定节点的最佳 **system-reserved** CPU 和内存资源，也可以手动决定并为节点设置最佳资源。
- 根据节点上的处理器内核数、硬限制或两者，配置可在节点上运行的 pod 数量。
- 使用 pod 反关联性来安全地重新引导节点。
- 通过使用计算机器集缩减集群，从集群中删除节点。要从裸机集群中删除节点，您必须首先排空节点上的所有 pod，然后手动删除该节点。

增强操作
OpenShift Container Platform 不仅支持访问和管理节点；作为管理员，您可以在节点上执行以下任务，使集群更高效、应用程序友好，并为开发人员提供更好的环境。

- 使用 Node Tuning Operator，为需要一定等级内核调整的高性能应用程序管理节点级别的性能优化。
- 在节点上启用 TLS 安全配置集，以保护 kubelet 和 Kubernetes API 服务器之间的通信。
- 使用守护进程集在节点上自动运行后台任务。您可以创建并使用守护进程集来创建共享存储，在每个节点上运行日志记录 pod，或者在所有节点上部署监控代理。
- 使用垃圾回收释放节点资源。您可以通过删除终止的容器以及任何正在运行的 pod 不引用的镜像来确保节点高效运行。
- 在一组节点中添加内核参数。
- 将 OpenShift Container Platform 集群配置为在网络边缘（远程 worker 节点）具有 worker 节点。如需有关在 OpenShift Container Platform 集群中具有远程 worker 节点的挑战，以及一些在远程 worker 节点上管理 pod 的建议方法，请参阅在网络边缘使用远程 worker 节点。

1.2. 关于 POD

pod 是节点上共同部署的一个或多个容器。作为集群管理员，您可以定义 pod，为它分配在准备好调度和管理的健康节点上运行。只要容器正在运行，pod 就会运行。在 Pod 被定义并运行后，您无法更改它。使用 pod 时，您可以执行的一些操作包括：

读取操作
作为管理员，您可以通过以下任务来获取项目中的 pod 信息：

- 列出与项目关联的 pod，包括副本数、重启、当前状态和年龄等信息。
- 查看 pod 用量统计，如 CPU、内存和存储消耗。

管理操作
以下任务列表概述了管理员如何在 OpenShift Container Platform 集群中管理 pod。

- 使用 OpenShift Container Platform 中可用的高级调度功能控制 pod 调度：
  - 节点到 pod 的绑定规则，如 pod 关联性、节点关联性 和 反关联性。
  - 节点标签和选择器。
  - 污点和容限。
  - Pod 拓扑分布约束。
  - 二级调度。
- 配置 descheduler 以根据特定策略驱除 pod，以便调度程序将 pod 重新调度到更合适的节点。
- 配置 pod 如何使用 pod 控制器重启后的行为，然后重新启动策略。
- 限制 pod 上的出口和入口流量。
- 从具有 pod 模板的任何对象中添加和移除卷。卷是 pod 中所有容器使用的已挂载文件系统。容器存储是临时的；您可以使用卷来持久保留容器数据。

增强操作
您可以使用 OpenShift Container Platform 中提供的各种工具和功能，更轻松地使用 pod。以下操作涉及使用这些工具和功能来更好地管理 pod。

操作	用户	更多信息
创建并使用 pod 横向自动扩展。	开发者	您可以使用 pod 横向自动扩展来指定您要运行的 pod 的最小和最大数量，以及 pod 的目标 CPU 使用率或内存使用率。通过使用 pod 横向自动扩展，您可以自动扩展 pod。
安装和使用垂直 pod 自动缩放器。	管理员和开发人员	作为管理员，通过监控资源和资源要求，使用垂直 pod 自动扩展来更好地利用集群资源。 作为开发人员，使用垂直 pod 自动扩展来确保 pod 在高负载时可以继续工作，方法是将 pod 调度到具有每个 pod 充足资源的节点。
使用设备插件提供对外部资源的访问。	Administrator	设备插件是在节点（kubelet 的外部）上运行的 gRPC 服务，用于管理特定的硬件资源。您可以部署设插件，以提供一致且可移植的解决方案，以便在集群中使用硬件设备。

操作	用户	更多信息
使用 **Secret** 对象 向 pod 提供敏感数据。	Administrator	有些应用程序需要敏感信息，如密码和用户名。您可以使用 **Secret** 对象向应用程序 pod 提供此类信息。

1.3. 关于容器

容器是 OpenShift Container Platform 应用程序的基本单元，它由应用程序代码与其依赖项、库和二进制文件一起打包。容器提供不同环境间的一致性和多个部署目标：物理服务器、虚拟机 (VM) 和私有或公有云。

Linux 容器技术是一种轻量型机制，用于隔离运行中的进程，仅限制对指定的资源的访问。作为管理员，您可以在 Linux 容器上执行各种任务，例如：

- 将文件复制到一个容器中或从容器中复制。
- 允许容器消耗 API 对象。
- 在容器中执行远程命令。
- 使用端口转发来访问容器中的应用程序。

OpenShift Container Platform 提供针对 Init 容器的专用容器。init 容器在应用程序容器之前运行，可以包含应用程序镜像中不存在的工具或设置脚本。您可以在部署 pod 的其余部分之前，使用 Init 容器执行任务。

除了在节点、Pod 和容器上执行特定任务外，您还可使用整个 OpenShift Container Platform 集群来使集群高效和应用程序 pod 具有高可用性。

1.4. 关于节点上的自动扩展 POD

OpenShift Container Platform 提供了三种工具，可用于自动扩展节点上的 pod 数量以及分配给 pod 的资源。

Pod 横向自动扩展

Horizontal Pod Autoscaler (HPA) 可以根据从属于该复制控制器或部署配置的 pod 收集的指标自动增加或减少复制控制器或部署配置的规模。
如需更多信息，请参阅使用 pod 横向自动扩展自动扩展 pod。

自定义 Metrics Autoscaler

自定义 Metrics Autoscaler 可以根据不基于 CPU 或内存的自定义指标自动增加或减少部署、有状态集、自定义资源或作业的 pod 数量。
如需更多信息，请参阅自定义 Metrics Autoscaler Operator 概述。

Vertical Pod Autoscaler

Vertical Pod Autoscaler (VPA) 可以自动查看 pod 中容器的运行状况和当前的 CPU 和内存资源，并根据它所了解的用量值更新资源限值和请求。
如需更多信息，请参阅使用垂直 pod 自动扩展自动调整 pod 资源级别。

1.5. OPENSHIFT CONTAINER PLATFORM 节点的常用术语表

该术语表定义了在 *节点*内容中使用的常用术语。

Container

它是一个轻量级且可执行的镜像，它包括了软件及其所有依赖项。容器虚拟化操作系统，因此您可以在任意位置运行容器，包括数据中心到公共或私有云，甚至在开发人员笔记本电脑中运行。

守护进程集

确保 pod 副本在 OpenShift Container Platform 集群的合格节点上运行。

egress

通过来自 pod 的网络出站流量进行外部数据共享的过程。

垃圾回收

清理集群资源的过程，如终止的容器和未被任何正在运行的 Pod 引用的镜像。

横向 Pod 自动扩展 (HPA)

作为 Kubernetes API 资源和控制器实现。您可以使用 HPA 指定您要运行的 pod 的最小和最大数量。您还可以指定 pod 应该针对的 CPU 或内存使用率。当超过给定 CPU 或内存阈值时，HPA 会扩展或缩放 pod。

入口

到一个 pod 的传入流量。

作业

要完成的进程。作业创建一个或多个 pod 对象，并确保指定的 pod 成功完成。

标签

您可以使用标签（即键值对）来组织并选择对象子集，如 pod。

节点

OpenShift Container Platform 集群中的 worker 机器。节点可以是虚拟机 (VM) 或物理机器。

Node Tuning Operator

您可以使用 Node Tuning Operator，使用 TuneD 守护进程来管理节点级别的性能优化。它保证了自定义性能优化设置以可被守护进程支持的格式传递到在集群中运行的所有容器化的 TuneD 守护进程中。相应的守护进程会在集群的所有节点上运行，每个节点上运行一个。

自助服务修复 Operator

Operator 在集群节点上运行，并检测和重启不健康的节点。

Pod

一个或多个带有共享资源（如卷和 IP 地址）的容器，在 OpenShift Container Platform 集群中运行。pod 是定义、部署和管理的最小计算单元。

容限（toleration）

表示 pod 允许（但不需要）调度到具有匹配污点的节点组。您可以使用容限来启用调度程序来调度具有匹配污点的 pod。

污点（taint）

一个核心对象，由一个键、值和效果组成。污点和容限可以一起工作，以确保 pod 不会调度到不相关的节点上。

第 2 章 使用 POD

2.1. 使用 POD

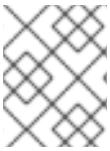
pod 是共同部署在同一主机上的一个或多个容器，也是可被定义、部署和管理的最小计算单元。

2.1.1. 了解 pod

对容器而言，Pod 大致相当于一个机器实例（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 通常应通过更高级别的控制器来管理，而不直接由用户管理。



注意

如需了解每个 OpenShift Container Platform 节点主机的最大 pod 数，请参阅“集群限制”。

警告

不受复制控制器管理的裸机 pod 不能在节点中断时重新调度。

2.1.2. pod 配置示例

OpenShift Container Platform 使用 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。

以下是 pod 的示例定义。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）

```
kind: Pod
apiVersion: v1
metadata:
  name: example
  labels:
    environment: production
    app: abc 1
spec:
  restartPolicy: Always 2
  securityContext: 3
```

```
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers: 4
    - name: abc
      args:
      - sleep
      - "1000000"
      volumeMounts: 5
       - name: cache-volume
         mountPath: /cache 6
      image: registry.access.redhat.com/ubi7/ubi-init:latest 7
      securityContext:
        allowPrivilegeEscalation: false
        runAsNonRoot: true
        capabilities:
          drop: ["ALL"]
      resources:
        limits:
          memory: "100Mi"
          cpu: "1"
        requests:
          memory: "100Mi"
          cpu: "1"
  volumes: 8
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
```

1. pod 可以被"标上"一个或多个标签，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。

2. pod 重启策略，可能的值有 **Always**、**OnFailure** 和 **Never**。默认值为 **Always**。

3. 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。

4. **containers** 指定包括一个或多个容器定义的数组。

5. 容器指定在容器中挂载外部存储卷的位置。

6. 指定要为 pod 提供的卷。卷挂载在指定路径上。不要挂载到容器 root、/ 或主机和容器中相同的任何路径。如果容器有足够权限，可能会损坏您的主机系统（如主机的 **/dev/pts** 文件）。使用 **/host** 挂载主机是安全的。

7. pod 中的每个容器使用自己的容器镜像进行实例化。

8. pod 定义了可供其容器使用的存储卷。

   如果将具有高文件数的持久性卷附加到 pod，则这些 pod 可能会失败，或者可能需要很长时间才能启动。如需更多信息，请参阅在 OpenShift 中使用具有高文件计数的持久性卷时，为什么 pod 无法启动或占用大量时间来实现"Ready"状态？

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。

2.1.3. 其他资源

- 如需有关 pod 和存储的更多信息，请参阅了解持久性存储和了解临时存储。

2.2. 查看 POD

作为管理员，您可以查看集群中的 pod，并确定这些 pod 和整个集群的健康状态。

2.2.1. 关于 pod

OpenShift Container Platform 使用 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）。

您可以查看与特定项目关联的 pod 列表，或者查看 pod 的使用情况统计。

2.2.2. 查看项目中的 pod

您可以查看与当前项目关联的 pod 列表，包括副本数、当前状态、重启次数和 pod 的年龄。

流程

查看项目中的 pod：

1. 切换到对应项目：

```
$ oc project <project-name>
```

2. 运行以下命令：

```
$ oc get pods
```

例如：

```
$ oc get pods
```

输出示例

```
NAME                       READY   STATUS    RESTARTS   AGE
console-698d866b78-bnshf   1/1     Running   2          165m
console-698d866b78-m87pm   1/1     Running   2          165m
```

添加 **-o wide** 标记来查看 pod IP 地址和 pod 所在的节点。

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

输出示例

```
```

```
NAME                    READY   STATUS    RESTARTS   AGE    IP          NODE
NOMINATED NODE
console-698d866b78-bnshf   1/1     Running   2          166m   10.128.0.24   ip-10-0-152-
71.ec2.internal   <none>
console-698d866b78-m87pm   1/1     Running   2          166m   10.129.0.23   ip-10-0-173-
237.ec2.internal   <none>
```

2.2.3. 查看 pod 用量统计

您可以显示 pod 的用量统计，这些统计信息为容器提供了运行时环境。这些用量统计包括 CPU、内存和存储的消耗。

先决条件

- 您必须有 **cluster-reader** 权限才能查看用量统计。
- 必须安装 Metrics 才能查看用量统计。

流程

查看用量统计：

1. 运行以下命令：

```
$ oc adm top pods
```

例如：

```
$ oc adm top pods -n openshift-console
```

输出示例

```
NAME                         CPU(cores)   MEMORY(bytes)
console-7f58c69899-q8c8k     0m           22Mi
console-7f58c69899-xhbgg     0m           25Mi
downloads-594fcccf94-bcxk8   3m           18Mi
downloads-594fcccf94-kv4p6   2m           15Mi
```

2. 运行以下命令，以查看带有标签的 pod 用量统计：

```
$ oc adm top pod --selector=''
```

您必须选择过滤所基于的选择器（标签查询）。支持 **=**、**==** 和 **!=**。

例如：

```
$ oc adm top pod --selector='name=my-pod'
```

2.2.4. 查看资源日志

您可以在 OpenShift CLI (**oc**) 和 Web 控制台中查看各种资源的日志。日志从日志的尾部或末尾读取。

先决条件

- 访问 OpenShift CLI（**oc**）。

流程 (UI)

1. 在 OpenShift Container Platform 控制台中，导航到 **Workloads → Pods**，或通过您要调查的资源导航到 pod。

注意

有些资源（如构建）没有直接查询的 pod。在这种情况下，您可以在资源的 **Details** 页面中找到 **Logs** 链接。

2. 从下拉菜单中选择一个项目。
3. 点您要调查的 pod 的名称。
4. 点 **Logs**。

流程 (CLI)

- 查看特定 pod 的日志：

```
$ oc logs -f <pod_name> -c <container_name>
```

其中：

-f
可选：指定输出是否遵循要写到日志中的内容。

<pod_name>
指定 pod 的名称。

<container_name>
可选：指定容器的名称。当 pod 具有多个容器时，您必须指定容器名称。

例如：

```
$ oc logs ruby-58cd97df55-mww7r
```

```
$ oc logs -f ruby-57f7f4855b-znl92 -c ruby
```

输出的日志文件内容。

- 查看特定资源的日志：

```
$ oc logs <object_type>/<resource_name> 1
```

1 指定资源类型和名称。

例如：

```
$ oc logs deployment/ruby
```

输出的日志文件内容。

2.3. 为 POD 配置 OPENSHIFT CONTAINER PLATFORM 集群

作为管理员，您可以为 pod 创建和维护高效的集群。

通过确保集群高效运行，您可以使用一些工具为开发人员提供更好的环境，例如，pod 退出时的行为，确保始终有所需数量的 pod 在运行，何时重启设计为只运行一次的 pod，限制 pod 可以使用的带宽，以及如何在中断时让 pod 保持运行。

2.3.1. 配置 pod 重启后的行为

pod 重启策略决定了 OpenShift Container Platform 在该 pod 中的容器退出时作出何种响应。该策略适用于 pod 中的所有容器。

可能的值有：

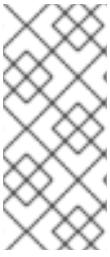
- **Always** - 在 pod 被重启之前，按规定的延时值（10s，20s，40s）不断尝试重启 pod 中成功退出的容器（最长为 5 分钟）。默认值为 **Always**。
- **OnFailure** - 按规定的延时值（10s，20s，40s）不断尝试重启 pod 中失败的容器，上限为 5 分钟。
- **Never** - 不尝试重启 pod 中已退出或失败的容器。Pod 立即失败并退出。

在 pod 绑定到某个节点后，该 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**。
每台机器必须运行一个的 Pod	守护进程集	任意

如果 pod 上的容器失败且重启策略设为 **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新启动，请使用 **Never** 重启策略。

如果整个 pod 失败，OpenShift Container Platform 会启动一个新 pod。开发人员必须解决应用程序可能会在新 pod 中重启的情况。特别是，应用程序必须处理由以往运行产生的临时文件、锁定、不完整输出等结果。



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。

2.3.2. 限制可供 pod 使用的带宽

您可以对 pod 应用服务质量流量控制，有效限制其可用带宽。出口流量（从 pod 传出）按照策略来处理，仅在超出配置的速率时丢弃数据包。入口流量（传入 pod 中）通过控制已排队数据包进行处理，以便有效地处理数据。您对 pod 应用的限制不会影响其他 pod 的带宽。

流程

限制 pod 的带宽：

1. 编写对象定义 JSON 文件，并使用 **kubernetes.io/ingress-bandwidth** 和 **kubernetes.io/egress-bandwidth** 注解指定数据流量速度。例如，将 pod 出口和入口带宽限制为 10M/s：

   受限 Pod 对象定义

```
{
    "kind": "Pod",
    "spec": {
        "containers": [
            {
                "image": "openshift/hello-openshift",
                "name": "hello-openshift"
            }
        ]
    },
    "apiVersion": "v1",
    "metadata": {
        "name": "iperf-slow",
        "annotations": {
            "kubernetes.io/ingress-bandwidth": "10M",
            "kubernetes.io/egress-bandwidth": "10M"
        }
    }
}
```

2. 使用对象定义创建 pod：

```
$ oc create -f <file_or_dir_path>
```

2.3.3. 了解如何使用 pod 中断预算来指定必须在线的 pod 数量

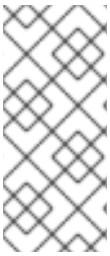
*pod 中断预算*允许在操作过程中指定 pod 的安全限制，如排空节点以进行维护。

PodDisruptionBudget 是一个 API 对象，用于指定在某一时间必须保持在线的副本的最小数量或百分比。在项目中进行这些设置对节点维护（比如缩减集群或升级集群）有益，而且仅在自愿驱除（而非节点失败）时遵从这些设置。

PodDisruptionBudget 对象的配置由以下关键部分组成：

- 标签选择器，即一组 pod 的标签查询。

- 可用性级别，用来指定必须同时可用的最少 pod 的数量：
  - **minAvailable** 是必须始终可用的 pod 的数量，即使在中断期间也是如此。
  - **maxUnavailable** 是中断期间可以无法使用的 pod 的数量。



注意

Available 指的是具有 **Ready=True** 的 pod 数量。**ready=True** 指的是能够服务请求的 pod，并应添加到所有匹配服务的负载平衡池中。

允许 **maxUnavailable** 为 **0%** 或 **0**，**minAvailable** 为 **100%** 或等于副本数，但这样设置可能会阻止节点排空操作。

警告

对于 OpenShift Container Platform 中的所有机器配置池，**maxUnavailable** 的默认设置是 **1**。建议您不要更改这个值，且一次只更新一个 control plane 节点。对于 control plane 池，请不要将这个值改为 **3**。

您可以使用以下命令来检查所有项目的 pod 中断预算：

```
$ oc get poddisruptionbudget --all-namespaces
```

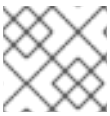
注意

以下示例包含特定于 AWS 上的 OpenShift Container Platform 的一些值。

输出示例

```
NAMESPACE                              NAME                                MIN AVAILABLE   MAX UNAVAILABLE
ALLOWED DISRUPTIONS   AGE
openshift-apiserver                    openshift-apiserver-pdb             N/A             1               1
121m
openshift-cloud-controller-manager     aws-cloud-controller-manager        1               N/A             1
125m
openshift-cloud-credential-operator    pod-identity-webhook                1               N/A             1
117m
openshift-cluster-csi-drivers          aws-ebs-csi-driver-controller-pdb   N/A             1               1
121m
openshift-cluster-storage-operator     csi-snapshot-controller-pdb         N/A             1               1
122m
openshift-cluster-storage-operator     csi-snapshot-webhook-pdb            N/A             1               1
122m
openshift-console                      console                             N/A             1               1
116m
#...
```

如果系统中至少有 **minAvailable** 个 pod 正在运行，则 **PodDisruptionBudget** 被视为是健康的。超过这一限制的每个 pod 都可被驱除。



注意

根据您的 pod 优先级与抢占设置，可能会无视 pod 中断预算要求而移除较低优先级 pod。

2.3.3.1. 使用 pod 中断预算指定必须在线的 pod 数量

您可以使用 **PodDisruptionBudget** 对象来指定某一时间必须保持在线的副本的最小数量或百分比。

流程

配置 pod 中断预算：

1. 使用类似以下示例的对象定义来创建 YAML 文件：

```
apiVersion: policy/v1 (1)
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2 (2)
  selector: (3)
    matchLabels:
      name: my-pod
```

(1) **PodDisruptionBudget** 是 **policy/v1** API 组的一部分。

(2) 必须同时可用的最小 pod 数量。这可以是整数，也可以是指定百分比的字符串（如 **20%**）。

(3) 对一组资源进行的标签查询。**matchLabels** 和 **matchExpressions** 的结果在逻辑上是联合的。要选择项目中的所有 pod，将此参数设置为空，如 **selector {}**。

或者：

```
apiVersion: policy/v1 (1)
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  maxUnavailable: 25% (2)
  selector: (3)
    matchLabels:
      name: my-pod
```

(1) **PodDisruptionBudget** 是 **policy/v1** API 组的一部分。

(2) 同时不能使用的最多的 pod 数量。这可以是整数，也可以是指定百分比的字符串（如 **20%**）。

(3) 对一组资源进行的标签查询。**matchLabels** 和 **matchExpressions** 的结果在逻辑上是联合的。要选择项目中的所有 pod，将此参数设置为空，如 **selector {}**。

2. 运行以下命令，将对象添加到项目中：

```
$ oc create -f </path/to/file> -n <project_name>
```

2.3.3.2. 为不健康的 pod 指定驱除策略

当您使用 pod 中断预算 (PDB) 来指定必须同时有多少 pod 可用时，您还可以定义驱除不健康 pod 的条件。

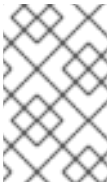
您可以选择以下策略之一：

IfHealthyBudget

只有在保护的应用程序没有被中断时，运行的还没有处于健康状态的 pod 才能被驱除。

AlwaysAllow

无论是否满足 pod 中断预算中的条件，运行的还没有处于健康状态的 pod 都可以被驱除。此策略可帮助驱除出现故障的应用程序，如 pod 处于 **CrashLoopBackOff** 状态或无法报告 **Ready** 状态的应用程序。



注意

建议您在 **PodDisruptionBudget** 对象中将 **unhealthyPodEvictionPolicy** 字段设置为 **AlwaysAllow**，以便在节点排空期间支持收集错误的应用程序。默认行为是等待应用程序 pod 处于健康状态，然后才能排空操作。

流程

1. 创建定义 **PodDisruptionBudget** 对象的 YAML 文件，并指定不健康的 pod 驱除策略：

pod-disruption-budget.yaml 文件示例

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      name: my-pod
  unhealthyPodEvictionPolicy: AlwaysAllow 1
```

1 选择 **IfHealthyBudget** 或 **AlwaysAllow** 作为不健康 pod 的驱除策略。当 **unhealthyPodEvictionPolicy** 字段为空时，默认为 **IfHealthyBudget**。

2. 运行以下命令来创建 **PodDisruptionBudget** 对象：

```
$ oc create -f pod-disruption-budget.yaml
```

现在，设置了 **AlwaysAllow** 不健康 pod 驱除策略的 PDB，您可以排空节点并驱除受此 PDB 保护的应用程序的 pod。

其他资源

- 使用功能门启用功能
- Kubernetes 文档中的不健康 Pod 驱除策略

2.3.4. 使用关键 pod 防止删除 pod

有不少核心组件对于集群完全正常工作而言至关重要，但它们在常规集群节点而非主节点上运行。如果一个关键附加组件被驱除，集群可能会停止正常工作。

标记为关键 (critical) 的 Pod 不允许被驱除。

流程

使 pod 成为关键 pod：

1. 创建 **Pod** spec 或编辑现有的 pod，使其包含 **system-cluster-critical** 优先级类：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pdb
spec:
  template:
    metadata:
      name: critical-pod
    priorityClassName: system-cluster-critical (1)
# ...
```

(1) 绝不可从节点驱除的 pod 的默认优先级类。

此外，对于对集群而言很重要但可在必要时移除的 pod，可以指定 **system-node-critical**。

2. 创建 pod：

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

2.3.5. 当使用带有大量文件的持久性卷时，可以减少 pod 超时的情况

如果存储卷包含多个文件（~1,000,000 或更多），您可能会遇到 pod 超时的问题。

这是因为当挂载卷时，OpenShift Container Platform 会递归更改每个卷内容的所有权和权限，以匹配 pod 的 **securityContext** 中指定的 **fsGroup**。对于大型卷，检查和更改所有权和权限可能会非常耗时，从而导致 pod 启动非常慢。

您可以通过应用以下临时解决方案之一来缩短这个延迟：

- 使用安全性上下文约束 (SCC) 跳过卷的 SELinux 重新标记。
- 使用 SCC 中的 **fsGroupChangePolicy** 字段来控制 OpenShift Container Platform 检查和管理卷的所有权和权限的方式。
- 使用 Cluster Resource Override Operator 自动应用 SCC 来跳过 SELinux 重新标记。

- 使用运行时类跳过卷的 SELinux 重新标记。

如需更多信息，请参阅在 OpenShift 中使用具有高文件计数的持久性卷时，为什么 pod 无法启动或占用大量时间来实现"Ready"状态？

2.4. 使用 POD 横向自动扩展自动扩展 POD

作为开发人员，您可以使用 pod 横向自动扩展 (HPA) 来指定 OpenShift Container Platform 如何根据从属于某复制控制器或部署配置的 pod 收集的指标来自动增加或缩小该复制控制器或部署配置的规模。您可以为部署、部署配置、副本集、复制控制器或有状态集创建 HPA。

有关根据自定义指标缩放 pod 的信息，请参阅基于自定义指标自动扩展 pod。

注意

除非需要特定功能或由其他对象提供的行为，否则建议使用 **Deployment** 对象或 **ReplicaSet** 对象。如需有关这些对象的更多信息，请参阅了解部署。

2.4.1. 了解 pod 横向自动扩展

您可以创建一个 pod 横向自动扩展来指定您要运行的 pod 的最小和最大数量，以及 pod 的目标 CPU 使用率或内存使用率。

在创建了 pod 横向自动扩展后，OpenShift Container Platform 会开始查询 pod 上的 CPU 和/或内存资源指标。当这些指标可用时，pod 横向自动扩展会计算当前指标使用率与所需指标使用率的比率，并相应地扩展或缩减。查询和缩放是定期进行的，但可能需要一到两分钟时间才会有可用指标。

对于复制控制器，这种缩放直接与复制控制器的副本对应。对于部署配置，缩放直接与部署配置的副本计数对应。注意，自动缩放仅应用到 **Complete** 阶段的最新部署。

OpenShift Container Platform 会自动考虑资源情况，并防止在资源激增期间进行不必要的自动缩放，比如在启动过程中。处于 **unready** 状态的 pod 在扩展时具有 **0 CPU** 用量，自动扩展在缩减时会忽略这些 pod。没有已知指标的 Pod 在扩展时具有 **0% CPU** 用量，在缩减时具有 **100% CPU** 用量。这在 HPA 决策过程中提供更高的稳定性。要使用这个功能，您必须配置就绪度检查来确定新 pod 是否准备就绪。

要使用 pod 横向自动扩展，您的集群管理员必须已经正确配置了集群指标。

2.4.1.1. 支持的指标

pod 横向自动扩展支持以下指标：

表 2.1. 指标

指标	描述	API 版本
CPU 使用率	已用的 CPU 内核数。可以用来计算 pod 的已请求 CPU 百分比。	**autoscaling/v1, autoscaling/v2**
内存使用率	已用内存量。可以用来计算 pod 的已请求内存百分比。	**autoscaling/v2**

重要

对于基于内存的自动缩放，内存用量必须与副本数呈正比增大和减小。平均而言：

- 增加副本数一定会导致每个 pod 的内存（工作集）用量总体降低。
- 减少副本数一定会导致每个 pod 的内存用量总体增高。

使用 OpenShift Container Platform Web 控制台检查应用程序的内存行为，并确保应用程序在使用基于内存的自动缩放前满足这些要求。

以下示例显示了 **image-registry Deployment** 对象的自动扩展。初始部署需要 3 个 pod。HPA 对象将最小值增加到 5。如果 pod 的 CPU 用量达到 75%，pod 会增加到 7:

```
$ oc autoscale deployment/image-registry --min=5 --max=7 --cpu-percent=75
```

输出示例

```
horizontalpodautoscaler.autoscaling/image-registry autoscaled
```

image-registry Deployment 对象的 HPA 示例，其中 minReplicas 被设置为 3

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: image-registry
  namespace: default
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: image-registry
  targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

1. 查看部署的新状态：

   ```
   $ oc get deployment image-registry
   ```

   部署中现在有 5 个 pod:

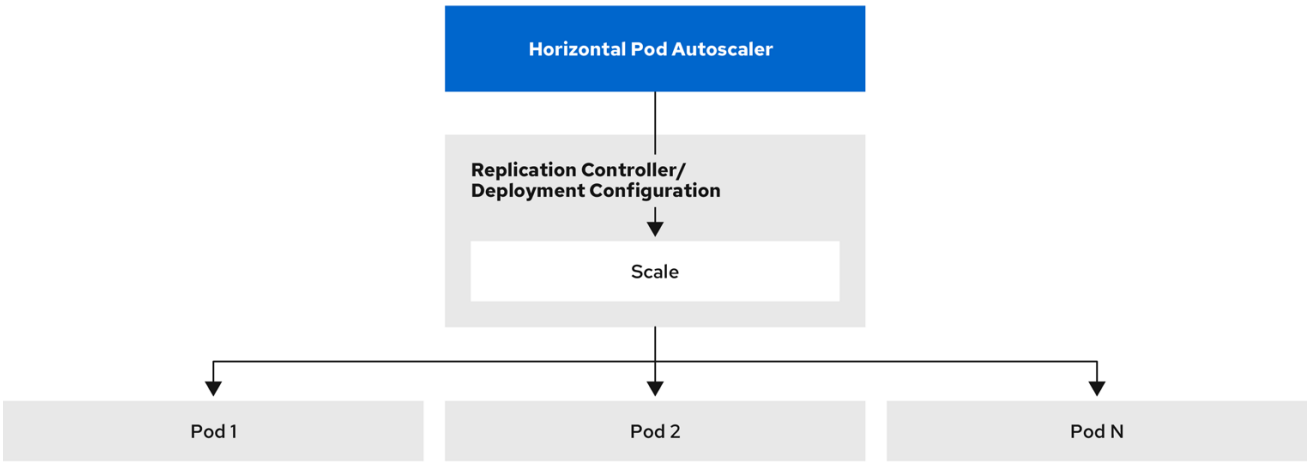
   输出示例

   ```
   NAME             REVISION   DESIRED   CURRENT   TRIGGERED BY
   image-registry   1          5         5         config
   ```

2.4.2. HPA 的工作原理？

pod 横向自动扩展(HPA)扩展了 pod 自动扩展的概念。HPA 允许您创建和管理一组负载均衡的节点。当给定的 CPU 或内存阈值被超过时，HPA 会自动增加或减少 pod 数量。

图 2.1. HPA 的高级别工作流



HPA 是 Kubernetes 自动扩展 API 组中的 API 资源。自动扩展器充当控制循环，在同步周期内默认为 15 秒。在此期间，控制器管理器会根据 HPA 的 YAML 文件中定义的 CPU、内存使用率或两者查询 CPU、内存使用或两者。控制器管理器为 HPA 为目标的每个 pod 来获取来自每个 pod 资源指标（如 CPU 或内存）的资源指标的利用率指标。

如果设置了使用值目标，控制器会将利用率值视为各个 pod 中容器对等资源请求的百分比。然后，控制器需要所有目标 pod 的平均利用率，并生成一个用于缩放所需副本数的比率。HPA 配置为从 **metrics.k8s.io** 获取指标（由 metrics 服务器提供）。由于指标评估的动态性质，副本的数量在扩展一组副本期间会波动。

注意

要实现 HPA，所有目标 pod 都必须在其容器上设置了一个资源请求。

2.4.3. 关于请求和限制

调度程序使用您为 pod 中容器指定的资源请求，来确定要将 pod 放置到哪个节点。kubelet 强制执行您为容器指定的资源限值，以确保容器不允许使用超过指定的限制。kubelet 还保留针对该容器使用的系统资源的请求数量。

如何使用资源指标？

在 pod 规格中，您必须指定资源请求，如 CPU 和内存。HPA 使用此规范来确定资源利用率，然后扩展目标或缩减。

例如，HPA 对象使用以下指标源：

```
type: Resource
resource:
  name: cpu
  target:
    type: Utilization
    averageUtilization: 60
```

在本例中，HPA 将 pod 的平均利用率保持在 scale 目标为 60%。使用率是当前资源使用量与 pod 请求的资源之间的比率。

2.4.4. 最佳实践

所有 pod 都必须配置资源请求

HPA 根据 OpenShift Container Platform 集群中观察的 pod 或内存使用率值做出缩放决定。利用率值计算为各个容器集的资源请求的百分比。缺少资源请求值可能会影响 HPA 的最佳性能。

配置冷却期

在横向 pod 自动扩展过程中，可能会快速扩展事件，而不会造成时间差。配置 cool down 周期，以防止频繁的副本波动。您可以通过配置 **stabilizationWindowSeconds** 字段指定 cool down period。当用于扩展的指标保持波动时，stabilization 窗口用于限制副本数的波动。自动扩展算法使用这个窗口来推断以前的预期状态，并避免对工作负载扩展不需要的更改。

例如，为 **scaleDown** 字段指定了 stabilization 窗口：

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

在上例中，过去 5 分钟的所有所需状态都被视为。此近似滚动的最大值，避免让扩展算法频繁删除 pod，仅在稍后触发同等的 pod 重新创建。

2.4.4.1. 扩展策略

autoscaling/v2 API 允许您为 pod 横向自动扩展添加 *扩展策略*。扩展策略用于控制 OpenShift Container Platform 横向自动扩展（HPA）如何扩展 pod。扩展策略允许您通过设置在指定时间段内扩展的特定数量或特定百分比来限制 HPA 扩展或缩减的速率。您还可以定义一个 *稳定化窗口（stabilization window）*，在指标有较大波动时，使用之前计算出的期望状态来控制扩展。您可以为相同的扩展方向创建多个策略，并根据更改的大小决定使用哪些策略。您还可以通过计时的迭代限制缩放。HPA 在迭代过程中扩展 pod，然后在以后的迭代中执行扩展（如果需要）。

带有扩展策略的 HPA 对象示例

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
  behavior:
    scaleDown: 1
      policies: 2
      - type: Pods 3
        value: 4 4
        periodSeconds: 60 5
      - type: Percent
        value: 10 6
        periodSeconds: 60
      selectPolicy: Min 7
      stabilizationWindowSeconds: 300 8
```

```
    scaleUp: 9
      policies:
      - type: Pods
        value: 5 10
        periodSeconds: 70
      - type: Percent
        value: 12 11
        periodSeconds: 80
      selectPolicy: Max
      stabilizationWindowSeconds: 0
...
```

1. 指定扩展策略的方向，可以是 **scaleDown** 或 **scaleUp**。本例为缩减创建一个策略。
2. 定义扩展策略。
3. 决定策略是否在每次迭代过程中根据特定的 pod 数量或 pod 百分比进行扩展。默认值为 **pod**。
4. 在每次迭代过程中缩放数量（pod 数量或 pod 的百分比）的限制。在按 pod 数量进行缩减时没有默认的值。
5. 决定扩展迭代的长度。默认值为 **15** 秒。
6. 按百分比缩减的默认值为 100%。
7. 如果定义了多个策略，则决定首先使用哪个策略。指定 **Max** 使用允许最多更改的策略，**Min** 使用允许最小更改的策略，或者 **Disabled** 阻止 HPA 在策略方向进行扩展。默认值为 **Max**。
8. 决定 HPA 应该重新查看所需状态的时间周期。默认值为 **0**。
9. 本例为扩展创建了策略。
10. 扩展数量的限制（按 pod 的数量）。扩展 pod 数量的默认值为 4%。
11. 扩展数量的限制（按 pod 的百分比）。按百分比扩展的默认值为 100%。

缩减策略示例

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory
  namespace: default
spec:
...
  minReplicas: 20
...
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Pods
        value: 4
        periodSeconds: 30
      - type: Percent
```

```
      value: 10
      periodSeconds: 60
    selectPolicy: Max
  scaleUp:
    selectPolicy: Disabled
```

在本例中，当 pod 的数量大于 40 时，则使用基于百分比的策略进行缩减。这个策略会产生较大变化，这是 **selectPolicy** 需要的。

如果有 80 个 pod 副本，在第一次迭代时 HPA 会将 pod 减少 8 个，即 80 个 pod 的 10%（根据 **type: Percent** 和 **value: 10** 参数），持续一分钟（**periodSeconds: 60**）。对于下一个迭代，pod 的数量为 72。HPA 计算剩余 pod 的 10% 为 7.2，这个数值被舍入到 8，这会缩减 8 个 pod。在每一后续迭代中，将根据剩余的 pod 数量重新计算要缩放的 pod 数量。当 pod 的数量低于 40 时，基于 pod 的策略会被应用，因为基于 pod 的数值会大于基于百分比的数值。HPA 每次减少 4 个 pod（**type: Pod** 和 **value: 4**），持续 30 秒（**periodSeconds: 30**），直到剩余 20 个副本（**minReplicas**）。

selectPolicy: Disabled 参数可防止 HPA 扩展 pod。如果需要，可以通过调整副本集或部署集中的副本数来手动扩展。

如果设置，您可以使用 **oc edit** 命令查看扩展策略：

```
$ oc edit hpa hpa-resource-metrics-memory
```

输出示例

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  annotations:
    autoscaling.alpha.kubernetes.io/behavior:\
'{"ScaleUp":{"StabilizationWindowSeconds":0,"SelectPolicy":"Max","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":15},{"Type":"Percent","Value":100,"PeriodSeconds":15}]},\
"ScaleDown":{"StabilizationWindowSeconds":300,"SelectPolicy":"Min","Policies":
[{"Type":"Pods","Value":4,"PeriodSeconds":60},{"Type":"Percent","Value":10,"PeriodSeconds":60}]}}'
...
```

2.4.5. 使用 Web 控制台创建 pod 横向自动扩展

在 web 控制台中，您可以创建一个 pod 横向自动扩展(HPA)，用于指定要在 **Deployment** 或 **DeploymentConfig** 对象上运行的 pod 的最小和最大数量。您还可以定义 pod 的目标 CPU 或内存用量。

注意

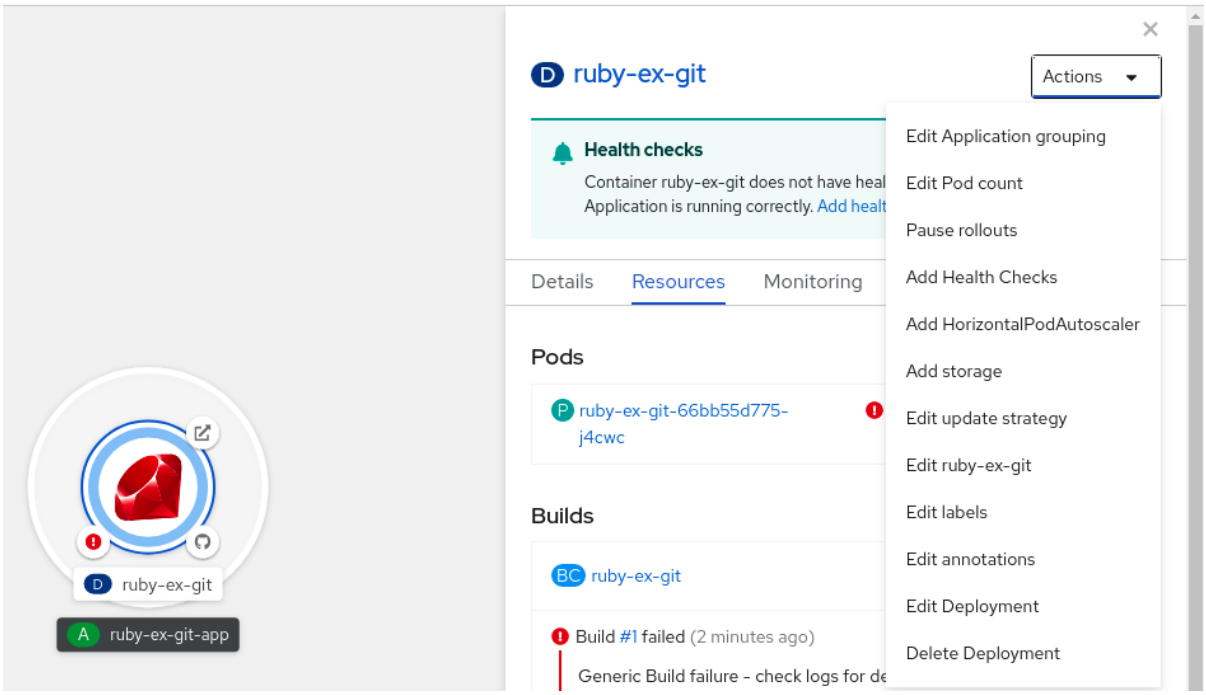
HPA 不能添加到作为 Operator 支持服务、Knative 服务或 Helm chart 一部分的部署中。

流程

在 web 控制台中创建 HPA：

1. 在 **Topology** 视图中，点击节点公开侧面板。
2. 在 **Actions** 下拉列表中，选择 **Add HorizontalPodAutoscaler** 来打开 **Add HorizontalPodAutoscaler** 表单。

图 2.2. 添加 HorizontalPodAutoscaler



3. 在 **Add HorizontalPodAutoscaler** 表单中，定义名称、最小和最大 pod 限值、CPU 和内存用量，并点 **Save**。

注意

如果缺少 CPU 和内存用量的值，则会显示警告。

在 web 控制台中编辑 HPA：

1. 在 **Topology** 视图中，点击节点公开侧面板。
2. 在 **Actions** 下拉列表中，选择 **Edit HorizontalPodAutoscaler** 来打开 **Edit Horizontal Pod Autoscaler** 表单。
3. 在 **Edit Horizontal Pod Autoscaler** 表单中，编辑最小和最大 pod 限值以及 CPU 和内存用量，然后点 **Save**。

注意

在 web 控制台中创建或编辑 pod 横向自动扩展时，您可以从 **Form** 视图切换到 **YAML** 视图。

在 web 控制台中删除 HPA：

1. 在 **Topology** 视图中，点击节点公开侧面板。
2. 在 **Actions** 下拉列表中，选择 **Remove HorizontalPodAutoscaler**。
3. 在确认弹出窗口中点击 **Remove** 删除 HPA。

2.4.6. 使用 CLI 根据 CPU 使用率创建 pod 横向自动扩展

使用 OpenShift Container Platform CLI，您可以创建一个 pod 横向自动扩展(HPA)来自动扩展现有的 **Deployment**、**DeploymentConfig**、**ReplicaSet**、**ReplicationController** 或 **StatefulSet** 对象。HPA 扩展与该对象关联的 pod，以维护您指定的 CPU 用量。

注意

除非需要特定功能或由其他对象提供的行为，否则建议使用 **Deployment** 对象或 **ReplicaSet** 对象。

HPA 会在最小和最大数量之间增加和减少副本数，以保持所有 pod 的指定 CPU 使用率。

为 CPU 使用率自动扩展时，您可以使用 **oc autoscale** 命令，并指定要在任意给定时间运行的 pod 的最小和最大数量，以及 pod 的目标平均 CPU 使用率。如果未指定最小值，则 OpenShift Container Platform 服务器会为 pod 赋予一个默认值。

要自动缩放特定 CPU 值，创建一个带有目标 CPU 和 pod 限制的 **HorizontalPodAutoscaler** 对象。

先决条件

要使用 pod 横向自动扩展，您的集群管理员必须已经正确配置了集群指标。您可以使用 **oc describe PodMetrics <pod-name>** 命令来判断是否已配置了指标。如果配置了指标，输出类似于以下示例，其中 **Usage** 下列出了 **Cpu** 和 **Memory**。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

输出示例

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:             2019-05-23T18:47:56Z
Window:                1m0s
Events:                <none>
```

流程

为 CPU 使用率创建 pod 横向自动扩展：

1. 执行以下之一：

- 要根据 CPU 使用率百分比来缩放，请为现有对象创建一个 **HorizontalPodAutoscaler** 对象：

```
$ oc autoscale <object_type>/<name> \ 1
  --min <number> \ 2
  --max <number> \ 3
  --cpu-percent=<percent> 4
```

1. 指定要自动扩展的对象类型和名称。对象必须存在，并需要是 **Deployment**, **DeploymentConfig/dc**, **ReplicaSet/rs**, **ReplicationController/rc**, 或 **StatefulSet**。
2. 另外，还可以指定缩减时的最小副本数量。
3. 指定扩展时的最大副本数量。
4. 指定所有 pod 的目标平均 CPU 使用率（以请求 CPU 的百分比表示）。如果未指定或为负数，则会使用默认的自动缩放策略。

例如，以下命令显示 **image-registry Deployment** 对象的自动扩展。初始部署需要 3 个 pod。HPA 对象将最小值增加到 5。如果 pod 的 CPU 用量达到 75%，pod 将增加到 7:

```
$ oc autoscale deployment/image-registry --min=5 --max=7 --cpu-percent=75
```

- 要扩展特定 CPU 值，请为现有对象创建类似如下的 YAML 文件：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: autoscaling/v2 1
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-autoscale 2
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 3
    kind: Deployment 4
    name: example 5
  minReplicas: 1 6
  maxReplicas: 10 7
  metrics: 8
  - type: Resource
    resource:
      name: cpu 9
      target:
        type: AverageValue 10
        averageValue: 500m 11
```

1. 使用 **autoscaling/v2** API。
2. 指定此 pod 横向自动扩展对象的名称。
3. 指定要缩放对象的 API 版本：

- 对于 **Deployment**，**ReplicaSet**、**Statefulset** 对象，使用 **apps/v1**。
- 对于 **ReplicationController**，使用 **v1**。
- 对于 **DeploymentConfig**，使用 **apps.openshift.io/v1**。

4 指定对象类型。对象需要是 **Deployment**, **DeploymentConfig/dc**, **ReplicaSet/rs**, **ReplicationController/rc**, 或 **StatefulSet**.

5 指定要缩放的对象名称。对象必须存在。

6 指定缩减时的最小副本数量。

7 指定扩展时的最大副本数量。

8 对于内存使用率，使用 **metrics** 参数。

9 为 CPU 使用率指定 **cpu** 。

10 设置为 **AverageValue**。

11 使用目标 CPU 值设置为 **averageValue**。

b. 创建 Pod 横向自动扩展：

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

2. 验证 pod 横向自动扩展是否已创建：

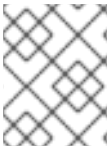
```
$ oc get hpa cpu-autoscale
```

输出示例

```
NAME          REFERENCE            TARGETS       MINPODS   MAXPODS   REPLICAS
AGE
cpu-autoscale   Deployment/example   173m/500m     1         10        1         20m
```

2.4.7. 使用 CLI 根据内存使用率创建 pod 横向自动扩展对象

使用 OpenShift Container Platform CLI，您可以创建一个 pod 横向自动扩展(HPA)来自动扩展现有的 **Deployment**、**DeploymentConfig**、**ReplicaSet**、**ReplicationController** 或 **StatefulSet** 对象。HPA 扩展与该对象关联的 pod，以维护您指定的平均内存使用率（可以是直接值，也可以是请求的内存百分比）。



注意

除非需要特定功能或由其他对象提供的行为，否则建议使用 **Deployment** 对象或 **ReplicaSet** 对象。

HPA 增加和减少最小和最大数量之间的副本数量，以维护所有 pod 的指定内存使用率。

对于内存使用率，您可以指定 pod 的最小和最大数量，以及 pod 的目标平均内存使用率。如果未指定最小值，则 OpenShift Container Platform 服务器会为 pod 赋予一个默认值。

先决条件

要使用 pod 横向自动扩展，您的集群管理员必须已经正确配置了集群指标。您可以使用 **oc describe PodMetrics <pod-name>** 命令来判断是否已配置了指标。如果配置了指标，输出类似于以下示例，其中 **Usage** 下列出了 **Cpu** 和 **Memory**。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-129-223.compute.internal -n openshift-kube-scheduler
```

输出示例

```
Name:         openshift-kube-scheduler-ip-10-0-129-223.compute.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Cpu:     0
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2020-02-14T22:21:14Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-kube-scheduler-ip-10-0-129-223.compute.internal
Timestamp:             2020-02-14T22:21:14Z
Window:                5m0s
Events:                <none>
```

流程

根据内存使用率创建 pod 横向自动扩展：

1. 为以下之一创建一个 YAML 文件：
   - 要扩展特定内存值，请为现有对象创建类似如下的 **HorizontalPodAutoscaler** 对象：

```
apiVersion: autoscaling/v2 (1)
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory (2)
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (3)
    kind: Deployment (4)
    name: example (5)
  minReplicas: 1 (6)
  maxReplicas: 10 (7)
  metrics: (8)
```

```
- type: Resource
  resource:
    name: memory 9
    target:
      type: AverageValue 10
      averageValue: 500Mi 11
behavior: 12
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
    selectPolicy: Max
```

1. 使用 **autoscaling/v2** API。
2. 指定此 pod 横向自动扩展对象的名称。
3. 指定要缩放对象的 API 版本：
   - 对于 **Deployment**、**ReplicaSet** 或 **Statefulset** 对象，请使用 **apps/v1**。
   - 对于 **ReplicationController**，使用 **v1**。
   - 对于 **DeploymentConfig**，使用 **apps.openshift.io/v1**。
4. 指定对象类型。对象必须是 **Deployment**、**DeploymentConfig**、**ReplicaSet**、**ReplicationController** 或 **StatefulSet**。
5. 指定要缩放的对象名称。对象必须存在。
6. 指定缩减时的最小副本数量。
7. 指定扩展时的最大副本数量。
8. 对于内存使用率，使用 **metrics** 参数。
9. 为内存使用率指定 **memory** 。
10. 将类型设置为 **AverageValue**。
11. 指定 **averageValue** 和一个特定的内存值。
12. 可选：指定一个扩展策略来控制扩展或缩减率。

- 要缩放一个百分比，请为现有对象创建一个类似如下的 **HorizontalPodAutoscaler** 对象：

```
apiVersion: autoscaling/v2 1
kind: HorizontalPodAutoscaler
metadata:
```

```
  name: memory-autoscale (2)
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (3)
    kind: Deployment (4)
    name: example (5)
  minReplicas: 1 (6)
  maxReplicas: 10 (7)
  metrics: (8)
  - type: Resource
    resource:
      name: memory (9)
      target:
        type: Utilization (10)
        averageUtilization: 50 (11)
  behavior: (12)
    scaleUp:
      stabilizationWindowSeconds: 180
      policies:
      - type: Pods
        value: 6
        periodSeconds: 120
      - type: Percent
        value: 10
        periodSeconds: 120
      selectPolicy: Max
```

1. 使用 **autoscaling/v2** API。
2. 指定此 pod 横向自动扩展对象的名称。
3. 指定要缩放对象的 API 版本：
   - 对于 ReplicationController，使用 **v1**。
   - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
   - 对于 Deployment、ReplicaSet 和 Statefulset 对象，使用 **apps/v1**。
4. 指定对象类型。对象必须是 **Deployment**、**DeploymentConfig**、**ReplicaSet**、**ReplicationController** 或 **StatefulSet**。
5. 指定要缩放的对象名称。对象必须存在。
6. 指定缩减时的最小副本数量。
7. 指定扩展时的最大副本数量。
8. 对于内存使用率，使用 **metrics** 参数。
9. 为内存使用率指定 **memory** 。
10. 设置 **Utilization**。

11 为所有 pod 指定 **averageUtilization** 和一个目标平均内存利用率，以请求内存的百分比表示。目标 pod 必须配置内存请求。

12 可选：指定一个扩展策略来控制扩展或缩减率。

2. 创建 Pod 横向自动扩展：

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

例如：

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

输出示例

```
horizontalpodautoscaler.autoscaling/hpa-resource-metrics-memory created
```

3. 验证 pod 横向自动扩展是否已创建：

```
$ oc get hpa hpa-resource-metrics-memory
```

输出示例

```
NAME                        REFERENCE           TARGETS        MINPODS   MAXPODS
REPLICAS   AGE
hpa-resource-metrics-memory   Deployment/example   2441216/500Mi   1         10        1
20m
```

```
$ oc describe hpa hpa-resource-metrics-memory
```

输出示例

```
Name:                       hpa-resource-metrics-memory
Namespace:                  default
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Wed, 04 Mar 2020 16:31:37 +0530
Reference:                  Deployment/example
Metrics:                    ( current / target )
  resource memory on pods:   2441216 / 500Mi
Min replicas:               1
Max replicas:               10
ReplicationController pods:  1 current / 1 desired
Conditions:
  Type           Status  Reason              Message
  ----           ------  ------              -------
  AbleToScale    True    ReadyForNewScale    recommended size matches current size
  ScalingActive  True    ValidMetricFound    the HPA was able to successfully calculate a
replica count from memory resource
  ScalingLimited  False   DesiredWithinRange  the desired count is within the acceptable
range
Events:
```

```
Type    Reason              Age               From                    Message
----    ------              ----              ----                    -------
Normal  SuccessfulRescale   6m34s             horizontal-pod-autoscaler  New size: 1;
reason: All metrics below target
```

2.4.8. 使用 CLI 了解 pod 横向自动扩展状态条件

您可以使用设置的状态条件来判断 pod 横向自动扩展 (HPA) 是否能够缩放，以及目前是否受到某种方式的限制。

HPA 状态条件可通过 **v2** 版的自动扩展 API 使用。

HPA 可以通过下列状态条件给予响应：

- **AbleToScale** 条件指示 HPA 是否能够获取和更新指标，以及是否有任何与退避相关的条件阻碍了缩放。
  - **True** 条件表示允许缩放。
  - **False** 条件表示因为指定原因不允许缩放。
- **ScalingActive** 条件指示 HPA 是否已启用（例如，目标的副本数不为零），并且可以计算所需的指标。
  - **True** 条件表示指标工作正常。
  - **False** 条件通常表示获取指标时出现问题。
- **ScalingLimited** 条件表示所需的规模由 pod 横向自动扩展限定最大或最小限制。
  - **True** 条件表示您需要提高或降低最小或最大副本数才能进行缩放。
  - **False** 条件表示允许请求的缩放。

```
$ oc describe hpa cm-test
```

输出示例

```
Name:                       cm-test
Namespace:                  prom
Labels:                     <none>
Annotations:                <none>
CreationTimestamp:          Fri, 16 Jun 2017 18:09:22 +0000
Reference:                  ReplicationController/cm-test
Metrics:                    ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:               1
Max replicas:               4
ReplicationController pods:     1 current / 1 desired
Conditions: 1
  Type            Status    Reason            Message
  ----            ------    ------            -------
  AbleToScale       True      ReadyForNewScale    the last scale time was sufficiently old
as to warrant a new scale
  ScalingActive     True      ValidMetricFound    the HPA was able to successfully
calculate a replica count from pods metric http_request
```

```
  ScalingLimited    False     DesiredWithinRange  the desired replica count is within the
acceptable range
Events:
```

1 pod 横向自动扩展状态消息。

下例中是一个无法缩放的 pod：

输出示例

```
Conditions:
  Type         Status  Reason          Message
  ----         ------  ------          -------
  AbleToScale  False   FailedGetScale  the HPA controller was unable to get the target's current
scale: no matches for kind "ReplicationController" in group "apps"
Events:
  Type     Reason          Age               From                    Message
  ----     ------          ----              ----                    -------
  Warning  FailedGetScale  6s (x3 over 36s)  horizontal-pod-autoscaler  no matches for kind
"ReplicationController" in group "apps"
```

下例中是一个无法获得缩放所需指标的 pod：

输出示例

```
Conditions:
  Type                Status    Reason                  Message
  ----                ------    ------                  -------
  AbleToScale         True      SucceededGetScale       the HPA controller was able to get the target's
current scale
  ScalingActive       False     FailedGetResourceMetric    the HPA was unable to compute the replica
count: failed to get cpu utilization: unable to get metrics for resource cpu: no metrics returned from
resource metrics API
```

下例中是一个请求的自动缩放低于所需下限的 pod：

输出示例

```
Conditions:
  Type            Status    Reason            Message
  ----            ------    ------            -------
  AbleToScale     True      ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive   True      ValidMetricFound    the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited  False     DesiredWithinRange  the desired replica count is within the acceptable
range
```

2.4.8.1. 使用 CLI 查看 pod 横向自动扩展状态条件

您可以查看 pod 横向自动扩展 (HPA) 对 pod 设置的状态条件。

注意

pod 横向自动扩展状态条件可通过 **v2** 版本的自动扩展 API 使用。

先决条件

要使用 pod 横向自动扩展，您的集群管理员必须已经正确配置了集群指标。您可以使用 **oc describe PodMetrics <pod-name>** 命令来判断是否已配置了指标。如果配置了指标，输出类似于以下示例，其中 **Usage** 下列出了 **Cpu** 和 **Memory**。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

输出示例

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:             2019-05-23T18:47:56Z
Window:                1m0s
Events:                <none>
```

流程

要查看 pod 上的状态条件，请使用以下命令并提供 pod 的名称：

```
$ oc describe hpa <pod-name>
```

例如：

```
$ oc describe hpa cm-test
```

这些条件会出现在输出中的 **Conditions** 字段里。

输出示例

```
Name:                         cm-test
Namespace:                    prom
Labels:                       <none>
Annotations:                  <none>
```

```
CreationTimestamp:              Fri, 16 Jun 2017 18:09:22 +0000
Reference:                      ReplicationController/cm-test
Metrics:                        ( current / target )
  "http_requests" on pods:      66m / 500m
Min replicas:                   1
Max replicas:                   4
ReplicationController pods:     1 current / 1 desired
Conditions: 1
  Type              Status    Reason              Message
  ----              ------    ------              -------
  AbleToScale       True      ReadyForNewScale    the last scale time was sufficiently old as to warrant a new scale
  ScalingActive     True      ValidMetricFound    the HPA was able to successfully calculate a replica count from pods metric http_request
  ScalingLimited    False     DesiredWithinRange  the desired replica count is within the acceptable range
```

2.4.9. 其他资源

- 如需有关复制控制器和部署控制器的更多信息，请参阅了解部署和部署配置。
- 有关使用 HPA 的示例，请参阅 基于 Memory Utilization 的 Horizontal Pod 自动扩展 Quarkus 应用。

2.5. 使用垂直 POD 自动扩展自动调整 POD 资源级别

OpenShift Container Platform Vertical Pod Autoscaler Operator（VPA）会自动检查 pod 中容器的运行状况和当前的 CPU 和内存资源，并根据它所了解的用量值更新资源限值和请求。VPA 使用单独的自定义资源（CR）来更新与工作负载对象关联的所有 Pod，如 **Deployment**、**DeploymentConfig**、**StatefulSet**、**Job**、**DaemonSet**、**ReplicaSet** 或 **ReplicationController**。

VPA 可帮助您了解 Pod 的最佳 CPU 和内存使用情况，并可以通过 pod 生命周期自动维护 pod 资源。

2.5.1. 关于 Vertical Pod Autoscaler Operator

Vertical Pod Autoscaler Operator（VPA）作为 API 资源和自定义资源（CR）实现。CR 决定 VPA Operator 对与特定工作负载对象（如守护进程集、复制控制器等）关联的 pod 执行的操作。

VPA Operator 由三个组件组成，每个组件在 VPA 命名空间中都有自己的 pod：

Recommender

VPA recommender 监控当前和过去的资源消耗，并根据这些数据决定关联工作负载对象中的 pod 的最佳 CPU 和内存资源。

Updater

VPA updater 检查相关工作负载对象中的 pod 是否具有正确的资源。如果资源正确，则 updater 不执行任何操作。如果资源不正确，则 updater 会终止 pod，以便它们的控制器可以使用更新的请求重新创建它们。

准入控制器

VPA 准入控制器在关联的工作负载对象中的每个新 pod 上设置正确的资源请求，无论 pod 是新的，还是因为 VPA updater 的操作由它的控制器重新创建的。

您可以使用默认推荐程序，或使用您自己的备选推荐程序根据您自己的算法自动扩展。

默认推荐器会自动计算这些 pod 中容器的流程以及当前的 CPU 和内存使用情况，并使用这些数据来决定优化的资源限制和请求，以确保这些 pod 始终高效操作。例如，默认推荐器会建议，减少请求资源超过使用资源的 pod 的资源，并为没有请求充足资源的 pod 增加资源。

VPA 每次自动删除任何与建议不兼容的 pod，以便您的应用程序可以在不需要停机的情况下继续满足请求。然后，工作负载对象使用原始资源限制和请求重新部署 pod。VPA 使用一个变异准入 webhook 来更新 pod，在 pod 被允许到节点前，具有优化的资源限制和请求。如果您不希望 VPA 删除 pod，可以查看 VPA 资源限制和请求，并根据需要手动更新 pod。

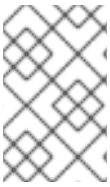
注意

默认情况下，工作负载对象必须至少指定两个副本，以便 VPA 自动删除其 pod。指定了比这个最小值更少的副本数的工作负载对象不会被删除。如果您手动删除这些 pod，当工作负载对象重新部署 pod 时，VPA 会使用其建议更新新的 pod。您可以通过修改 **VerticalPodAutoscalerController** 对象来更改这个最小值，如 *更改 VPA 最小值*所示。

例如，您有一个 pod 使用了 CPU 的 50%，但只请求 10%。VPA 会认定该 pod 消耗的 CPU 多于请求的 CPU，并删除 pod。工作负载对象（如副本集）会重启 pod，VPA 使用推荐的资源更新新 pod。

对于开发人员，您可以使用 VPA 来帮助确保 pod 在高负载时可以继续工作，具体方法是将 pod 调度到每个 pod 具有适当资源的节点上。

管理员可以使用 VPA 来更好地利用集群资源，例如防止 pod 保留比所需的 CPU 资源更多的资源。VPA 监控实际使用的工作负载，并对资源进行调整，以确保可以满足其他工作负载的需要。VPA 还维护初始容器配置中指定的限值和请求之间的比例。



注意

如果您停止在集群中运行 VPA，或删除特定的 VPA CR，则已由 VPA 修改的 pod 的资源请求不会改变。任何新 pod 都会根据工作负载对象中的定义获得资源，而不是之前由 VPA 提供的的建议。

2.5.2. 安装 Vertical Pod Autoscaler Operator

您可以使用 OpenShift Container Platform web 控制台安装 Vertical Pod Autoscaler Operator（VPA）。

流程

1. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
2. 从可用 Operator 列表中选择 **VerticalPodAutoscaler**，点 **Install**。
3. 在 **Install Operator** 页面中，确保选择了 **Operator 推荐的命名空间** 选项。这会在 **openshift-vertical-pod-autoscaler** 命名空间中创建 Operator。如果这个命名空间还没有存在，会自动创建它。
4. 点 **Install**。

验证

1. 列出 VPA Operator 组件来验证安装：
   a. 导航到 **Workloads → Pods**。
   b. 从下拉菜单中选择 **openshift-vertical-pod-autoscaler** 项目，并验证是否运行了四个 pod。

c. 进入 **Workloads → Deployments** 以验证运行了四个部署。

2. 可选：使用以下命令在 OpenShift Container Platform CLI 中验证安装：

```
$ oc get all -n openshift-vertical-pod-autoscaler
```

输出显示四个 pod 和四个部署：

输出示例

```
NAME                                             READY   STATUS    RESTARTS   AGE
pod/vertical-pod-autoscaler-operator-85b4569c47-2gmhc   1/1     Running   0          3m13s
pod/vpa-admission-plugin-default-67644fc87f-xq7k9       1/1     Running   0          2m56s
pod/vpa-recommender-default-7c54764b59-8gckt            1/1     Running   0          2m56s
pod/vpa-updater-default-7f6cc87858-47vw9                1/1     Running   0          2m56s

NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/vpa-webhook   ClusterIP   172.30.53.206   <none>        443/TCP   2m56s

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/vertical-pod-autoscaler-operator   1/1     1            1           3m13s
deployment.apps/vpa-admission-plugin-default       1/1     1            1           2m56s
deployment.apps/vpa-recommender-default            1/1     1            1           2m56s
deployment.apps/vpa-updater-default                1/1     1            1           2m56s

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/vertical-pod-autoscaler-operator-85b4569c47   1         1         1       3m13s
replicaset.apps/vpa-admission-plugin-default-67644fc87f       1         1         1       2m56s
replicaset.apps/vpa-recommender-default-7c54764b59            1         1         1       2m56s
replicaset.apps/vpa-updater-default-7f6cc87858                1         1         1       2m56s
```

2.5.3. 移动 Vertical Pod Autoscaler Operator 组件

Vertical Pod Autoscaler Operator (VPA)，每个组件在 control plane 节点上的 VPA 命名空间中都有自己的 pod。您可以通过在 VPA 订阅和 **VerticalPodAutoscalerController** CR 中添加节点选择器，将 VPA Operator 和组件 pod 移到基础架构节点。

您可以创建并使用基础架构节点来创建仅托管基础架构组件的机器，如默认路由器、集成的容器镜像 registry 以及集群指标和监控的组件。这些基础架构节点不计入运行环境所需的订阅总数中。如需更多信息，请参阅 *创建基础架构机器集*。

您可以根据您的机构，将组件移到同一节点或单独的节点。

以下示例显示了 VPA pod 到 control plane 节点的默认部署。

输出示例

```
NAME                                        READY   STATUS    RESTARTS   AGE     IP           NODE
NOMINATED NODE   READINESS GATES
vertical-pod-autoscaler-operator-6c75fcc9cd-5pb6z   1/1     Running   0          7m59s   10.128.2.24
c416-tfsbj-master-1   <none>           <none>
vpa-admission-plugin-default-6cb78d6f8b-rpcrj       1/1     Running   0          5m37s   10.129.2.22
c416-tfsbj-master-1   <none>           <none>
vpa-recommender-default-66846bd94c-dsmpp            1/1     Running   0          5m37s   10.129.2.20
```

```
c416-tfsbj-master-0   <none>          <none>
vpa-updater-default-db8b58df-2nkvf                1/1     Running   0          5m37s   10.129.2.21   c416-
tfsbj-master-1   <none>          <none>
```

流程

1. 通过将节点选择器添加到 VPA Operator 的 **Subscription** 自定义资源 (CR)中来移动 VPA Operator pod：

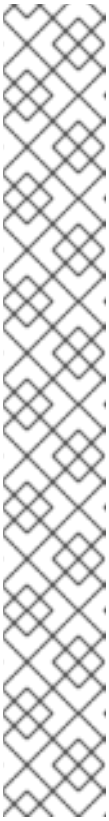
   a. 编辑 CR：

   ```
   $ oc edit Subscription vertical-pod-autoscaler -n openshift-vertical-pod-autoscaler
   ```

   b. 添加节点选择器以匹配您要安装 VPA Operator pod 的节点上的节点角色标签：

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     labels:
       operators.coreos.com/vertical-pod-autoscaler.openshift-vertical-pod-autoscaler: ""
     name: vertical-pod-autoscaler
   # ...
   spec:
     config:
       nodeSelector:
         node-role.kubernetes.io/<node_role>: "" (1)
   ```

   (1) 指定您要移动 VPA Operator pod 节点的节点角色。



注意

如果 infra 节点使用污点，则需要为 **Subscription** CR 添加容限。

例如：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  labels:
    operators.coreos.com/vertical-pod-autoscaler.openshift-vertical-pod-
autoscaler: ""
  name: vertical-pod-autoscaler
# ...
spec:
  config:
    nodeSelector:
      node-role.kubernetes.io/infra: ""
    tolerations: (1)
    - key: "node-role.kubernetes.io/infra"
      operator: "Exists"
      effect: "NoSchedule"
```

1 在要移动 VPA Operator pod 的节点上为污点指定容限。

2. 通过将节点选择器添加到 **VerticalPodAutoscaler** 自定义资源 (CR) 来移动每个 VPA 组件：

   a. 编辑 CR：

```
$ oc edit VerticalPodAutoscalerController default -n openshift-vertical-pod-autoscaler
```

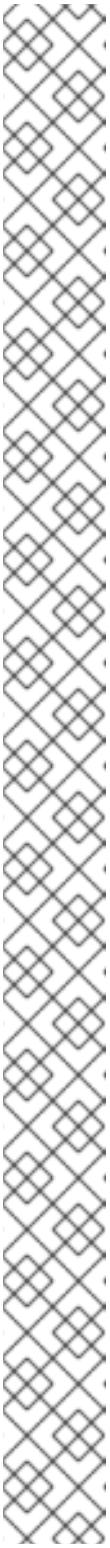
   b. 添加节点选择器以匹配您要安装 VPA 组件的节点上的节点角色标签：

```
apiVersion: autoscaling.openshift.io/v1
kind: VerticalPodAutoscalerController
metadata:
 name: default
  namespace: openshift-vertical-pod-autoscaler
# ...
spec:
  deploymentOverrides:
    admission:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/<node_role>: "" 1
    recommender:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/<node_role>: "" 2
    updater:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/<node_role>: "" 3
```

1 可选：指定 VPA 准入 pod 的节点角色。

2 可选：指定 VPA recommender pod 的节点角色。

3 可选：指定 VPA updater pod 的节点角色。



注意

如果目标节点使用污点，则需要为 **VerticalPodAutoscalerController** CR 添加容限。

例如：

```
apiVersion: autoscaling.openshift.io/v1
kind: VerticalPodAutoscalerController
metadata:
 name: default
  namespace: openshift-vertical-pod-autoscaler
# ...
spec:
  deploymentOverrides:
    admission:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/worker: ""
      tolerations: (1)
      - key: "my-example-node-taint-key"
        operator: "Exists"
        effect: "NoSchedule"
    recommender:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/worker: ""
      tolerations: (2)
      - key: "my-example-node-taint-key"
        operator: "Exists"
        effect: "NoSchedule"
    updater:
      container:
        resources: {}
      nodeSelector:
        node-role.kubernetes.io/worker: ""
      tolerations: (3)
      - key: "my-example-node-taint-key"
        operator: "Exists"
        effect: "NoSchedule"
```

1. 在您要安装 pod 的节点上为污点指定准入控制器 pod 的容限。
2. 在您要安装 pod 的节点上为污点指定推荐 pod 的容限。
3. 在您要安装 pod 的节点上为污点指定更新 pod 的容限。

验证

- 您可以使用以下命令验证 pod 是否已移动：

```
$ oc get pods -n openshift-vertical-pod-autoscaler -o wide
```

-

pod 不再部署到 control plane 节点。

输出示例

```
NAME                                        READY   STATUS    RESTARTS   AGE     IP
NODE                          NOMINATED NODE   READINESS GATES
vertical-pod-autoscaler-operator-6c75fcc9cd-5pb6z   1/1     Running   0          7m59s
10.128.2.24   c416-tfsbj-infra-eastus3-2bndt   <none>           <none>
vpa-admission-plugin-default-6cb78d6f8b-rpcrj       1/1     Running   0          5m37s
10.129.2.22   c416-tfsbj-infra-eastus1-lrgj8   <none>           <none>
vpa-recommender-default-66846bd94c-dsmpp            1/1     Running   0          5m37s
10.129.2.20   c416-tfsbj-infra-eastus1-lrgj8   <none>           <none>
vpa-updater-default-db8b58df-2nkvf                  1/1     Running   0          5m37s   10.129.2.21
c416-tfsbj-infra-eastus1-lrgj8   <none>           <none>
```

其他资源

- 创建基础架构机器集

2.5.4. 关于使用 Vertical Pod Autoscaler Operator

要使用 Vertical Pod Autoscaler Operator（vpa），您需要为集群中的工作负载对象创建 VPA 自定义资源（CR）。VPA 学习并应用与该工作负载对象关联的 pod 的最佳 CPU 和内存资源。您可以使用 VPA 与部署、有状态集、作业、守护进程集、副本集或复制控制器工作负载对象一起使用。VPA CR 必须与您要监控的 pod 位于同一个项目中。

您可以使用 VPA CR 关联一个工作负载对象，并指定 VPA 使用什么模式运行：

- **Auto** 和 **Recreate** 模式会在 pod 生命周期内自动应用 VPA 对 CPU 和内存建议。VPA 会删除项目中任何与建议不兼容的 pod。当由工作负载对象重新部署时，VPA 会在其建议中更新新 pod。
- **Initial** 模式仅在创建 pod 时自动应用 VPA 建议。
- **Off** 模式只提供推荐的资源限制和请求信息，用户可以手动应用其中的建议。**off** 模式不会更新 pod。

您还可以使用 CR 使特定容器不受 VPA 评估和更新的影响。

例如，pod 具有以下限制和请求：

```
resources:
  limits:
    cpu: 1
    memory: 500Mi
  requests:
    cpu: 500m
    memory: 100Mi
```

在创建了一个设置为 **auto** 的 VPA 后，VPA 会了解资源使用情况并删除 pod。重新部署时，pod 会使用新的资源限值和请求：

```
resources:
  limits:
    cpu: 50m
```

```
    memory: 1250Mi
  requests:
    cpu: 25m
    memory: 262144k
```

您可以使用以下命令查看 VPA 建议：

```
$ oc get vpa <vpa-name> --output yaml
```

几分钟后，输出显示 CPU 和内存请求的建议，如下所示：

输出示例

```
...
status:
...
  recommendation:
    containerRecommendations:
    - containerName: frontend
      lowerBound:
        cpu: 25m
        memory: 262144k
      target:
        cpu: 25m
        memory: 262144k
      uncappedTarget:
        cpu: 25m
        memory: 262144k
      upperBound:
        cpu: 262m
        memory: "274357142"
    - containerName: backend
      lowerBound:
        cpu: 12m
        memory: 131072k
      target:
        cpu: 12m
        memory: 131072k
      uncappedTarget:
        cpu: 12m
        memory: 131072k
      upperBound:
        cpu: 476m
        memory: "498558823"
...
```

输出显示推荐的资源、**目标**、最低推荐资源、**lowerBound**、最高推荐资源、**upperBound**、以及最新资源建议和 **uncappedTarget**。

VPA 使用 **lessBound** 和 **upperBound** 值来确定一个 pod 是否需要更新。如果 pod 的资源请求低于 **lowerBound** 值，或高于 **upperBound** 值，则 VPA 会终止 pod，并使用 **target** 值重新创建 pod。

2.5.4.1. 更改 VPA 最小值

默认情况下，工作负载对象必须至少指定两个副本，以便 VPA 自动删除和更新其 pod。因此，VPA 不会

自动执行指定少于两个副本的工作负载对象。如果 pod 由 VPA 外部的一些进程重启，VPA 会从这些工作负载对象更新的新 pod。您可以通过修改 **VerticalPodAutoscalerController** 自定义资源（CR）中的 **minReplicas** 参数来更改此集群范围的最小值。

例如，如果您将 **minReplicas** 设置为 **3**，则 VPA 不会为指定少于三个副本的工作负载对象删除和更新 pod。

注意

如果将 **minReplicas** 设置为 **1**，则 VPA 可以为只指定一个副本的工作负载对象删除唯一的 pod。只有在 VPA 删除 pod 以调整其资源时，您的工作负载可以允许停机时，才应使用此设置来使用一个副本对象。为了避免使用一个副本的对象出现不必要的停机时间，将带有 **podUpdatePolicy** 设置的 VPA CR 配置为 **Initial**，这只有在 VPA 外部的一些进程重启时，或状态为 **Off** 时才重启。这可让您在适合的时间手动更新 pod。

VerticalPodAutoscalerController 对象示例

```
apiVersion: autoscaling.openshift.io/v1
kind: VerticalPodAutoscalerController
metadata:
  creationTimestamp: "2021-04-21T19:29:49Z"
  generation: 2
  name: default
  namespace: openshift-vertical-pod-autoscaler
  resourceVersion: "142172"
  uid: 180e17e9-03cc-427f-9955-3b4d7aeb2d59
spec:
  minReplicas: 3 (1)
  podMinCPUMillicores: 25
  podMinMemoryMb: 250
  recommendationOnly: false
  safetyMarginFraction: 0.15
```

1. 指定 VPA 中要操作的工作负载对象中的最小副本数。VPA 不会自动删除任何小于最小副本的对象。

2.5.4.2. 自动应用 VPA 建议

要使用 VPA 来自动更新 pod，为特定工作负载对象创建一个 VPA CR，并将 **updateMode** 设置为 **Auto** 或 **Recreate**。

当为工作复杂对象创建 pod 时，VPA 会持续监控容器以分析其 CPU 和内存需求。VPA 会删除任何不满足 VPA 对 CPU 和内存的建议的 pod。重新部署后，pod 根据 VPA 建议使用新的资源限值和请求，并遵循您的应用程序的 pod 中断预算。建议被添加到 VPA CR 的 **status** 字段中以进行引用。

注意

默认情况下，工作负载对象必须至少指定两个副本，以便 VPA 自动删除其 pod。指定了比这个最小值更少的副本数的工作负载对象不会被删除。如果您手动删除这些 pod，当工作负载对象重新部署 pod 时，VPA 会使用其建议更新新的 pod。您可以通过修改 **VerticalPodAutoscalerController** 对象来更改这个最小值，如 *更改 VPA 最小值*所示。

Auto 模式的 VPA CR 示例

-

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Auto" (3)
```

1. 您希望此 VPA CR 管理的工作负载对象类型。
2. 您希望此 VPA CR 管理的工作负载对象名称。
3. 将模式设置为 **Auto** 或 **Recreate**:
   - **Auto**.VPA 分配创建 pod 的资源请求，并在请求的资源与新建议有很大不同时终止这些 Pod 来更新现存的 pod。
   - **Recreate**。VPA 分配创建 pod 的资源请求，并在请求的资源与新建议有很大不同时终止这些 Pod 来更新现存的 pod。这个模式应该很少使用，只有在需要确保每当资源请求改变时 pod 就需要重启时才使用。

注意

在 VPA 可以决定资源建议并将推荐的资源应用到新 pod 之前，操作 pod 必须存在并在项目中运行。

如果工作负载的资源使用情况（如 CPU 和内存）一致，VPA 可以在几分钟内决定资源的建议。如果工作负载的资源使用情况不一致，VPA 必须以各种资源使用量间隔收集指标，以便 VPA 做出准确的建议。

2.5.4.3. 在创建 pod 时自动应用 VPA 建议

要仅在 pod 首次部署时使用 VPA 来应用推荐的资源，为特定的工作负载对象创建一个 VPA CR，将 **updateMode** 设置为 **Initial**。

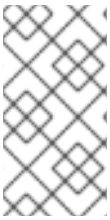
然后，手动删除与您要使用 VPA 建议的工作负载对象关联的 pod。在 **Initial** 模式中，VPA 不会删除 pod，也不会更新 pod，它会学习新的资源建议。

Initial 模式的 VPA CR 示例

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
```

```
    name:       frontend (2)
  updatePolicy:
    updateMode: "Initial" (3)
```

1. 您希望此 VPA CR 管理的工作负载对象类型。
2. 您希望此 VPA CR 管理的工作负载对象名称。
3. 将模式设置为 **Initial**。VPA 在 pod 创建时分配资源，在 pod 生命周期中不会更改资源。



注意

在 VPA 可以决定推荐的资源并对新 pod 应用建议之前，操作 pod 必须存在并在项目中运行。

要从 VPA 获取最准确的建议，请至少等待 8 天，让 pod 运行以及 VPA 稳定。

2.5.4.4. 手动应用 VPA 建议

要使用 VPA 来仅决定推荐的 CPU 和内存值而不进行实际的应用，对特定的工作负载创建一个 VPA CR，把 **updateMode** 设置为 **off**。

当为该工作负载对象创建 pod 时, VPA 会分析容器的 CPU 和内存需求，并在 VPA CR 的 **status** 字段中记录推荐。VPA 会提供新的资源建议，但不会更新 pod。

使用 Off 模式的 VPA CR 示例

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Off" (3)
```

1. 您希望此 VPA CR 管理的工作负载对象类型。
2. 您希望此 VPA CR 管理的工作负载对象名称。
3. 将模式设置为 **Off**。

您可以使用以下命令查看建议。

```
$ oc get vpa <vpa-name> --output yaml
```

根据建议，您可以编辑工作负载对象以添加 CPU 和内存请求，然后删除 pod 并使用推荐的资源重新部署 pod。

注意

在 VPA 可以决定推荐的资源并对新 pod 应用建议之前，操作 pod 必须存在并在项目中运行。

要从 VPA 获取最准确的建议，请至少等待 8 天，让 pod 运行以及 VPA 稳定。

2.5.4.5. 阻止容器特定容器应用 VPA 建议

如果您的工作负载对象有多个容器，且您不希望 VPA 对所有容器进行评估并进行操作，请为特定工作负载对象创建一个 VPA CR，添加一个 **resourcePolicy** 已使特定容器不受 VPA 的影响。

当 VPA 使用推荐的资源更新 pod 时，任何带有 **resourcePolicy** 的容器都不会被更新，且 VPA 不会对这些 pod 中的容器提供建议。

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment (1)
    name:       frontend (2)
  updatePolicy:
    updateMode: "Auto" (3)
  resourcePolicy: (4)
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
```

1. 您希望此 VPA CR 管理的工作负载对象类型。
2. 您希望此 VPA CR 管理的工作负载对象名称。
3. 将模式设置为 **Auto**、**Recreate** 或 **Off**。**Recreate** 模式应该很少使用，只有在需要确保每当资源请求改变时 pod 就需要重启时才使用。
4. 指定不受 VPA 影响的容器，将 **mode** 设置为 **Off**。

例如，一个 pod 有两个容器，它们有相同的资源请求和限值：

```
# ...
spec:
  containers:
  - name: frontend
    resources:
      limits:
        cpu: 1
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
  - name: backend
```

```
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
# ...
```

在启用一个带有 **backend** 排除容器设置的 VPA CR 后，VPA 终止并使用推荐的资源重新创建 pod 的行为只适用于 **frontend** 容器：

```
...
spec:
  containers:
    name: frontend
    resources:
      limits:
        cpu: 50m
        memory: 1250Mi
      requests:
        cpu: 25m
        memory: 262144k
...
    name: backend
    resources:
      limits:
        cpu: "1"
        memory: 500Mi
      requests:
        cpu: 500m
        memory: 100Mi
...
```

2.5.4.6. 性能调优 VPA Operator

作为集群管理员，您可以调整 Vertical Pod Autoscaler Operator (VPA) 的性能，以限制 VPA 对 Kubernetes API 服务器发出请求的速率，并为 VPA recommender, updater 和准入控制器组件 pod 指定 CPU 和内存资源。

另外，您可以将 VPA Operator 配置为仅监控由 VPA 自定义资源 (CR) 管理的工作负载。默认情况下，VPA Operator 会监控集群中的所有工作负载。这允许 VPA Operator 为所有工作负载处理和存储 8 天的历史数据，如果为工作负载创建新的 VPA CR，Operator 可以使用该数据。但是，这会导致 VPA Operator 使用大量 CPU 和内存，这可能会导致 Operator 失败，特别是在大型集群中。通过将 VPA Operator 配置为仅监控 VPA CR 的工作负载，您可以在 CPU 和内存资源上保存。一个权衡方案是，如果您有一个运行的工作负载，并且创建一个 VPA CR 来管理那个工作负载，则 VPA Operator 没有该工作负载的历史数据。因此，在工作负载运行了一段时间后，初始建议并没有这些有用。

这些调整允许您确保 VPA 有足够资源以峰值效率运行，并防止 pod 准入中的节流和可能的延迟。

您可以通过编辑 **VerticalPodAutoscalerController** 自定义资源 (CR) 在 VPA 组件上执行以下调整：

- 要防止节流和 pod 准入延迟，使用 **kube-api-qps** 和 **kube-api-burst** 参数为 Kubernetes API 服务器的 VPA 请求设置 queries-per-second (QPS) 和突发率。

- 为确保足够的 CPU 和内存，请使用标准 **cpu** 和 **memory** 资源请求为 VPA 组件 pod 设置 CPU 和内存请求。
- 要将 VPA Operator 配置为仅监控由 VPA CR 管理的工作负载，请将 recommender 组件的 **memory-saver** 参数设置为 **true**。

以下示例 VPA 控制器 CR 设置 VPA API QPS 和突发（burts）率，配置组件 pod 资源请求，并为 recommender 将 **memory-saver** 设置为 **true** ：

示例 VerticalPodAutoscalerController CR

```
apiVersion: autoscaling.openshift.io/v1
kind: VerticalPodAutoscalerController
metadata:
  name: default
  namespace: openshift-vertical-pod-autoscaler
spec:
  deploymentOverrides:
    admission: 1
      container:
        args: 2
          - '--kube-api-qps=30.0'
          - '--kube-api-burst=40.0'
        resources:
          requests: 3
            cpu: 40m
            memory: 40Mi
    recommender: 4
      container:
        args:
          - '--kube-api-qps=20.0'
          - '--kube-api-burst=60.0'
          - '--memory-saver=true' 5
        resources:
          requests:
            cpu: 60m
            memory: 60Mi
    updater: 6
      container:
        args:
          - '--kube-api-qps=20.0'
          - '--kube-api-burst=80.0'
        resources:
          requests:
            cpu: 80m
            memory: 80Mi
  minReplicas: 2
  podMinCPUMillicores: 25
  podMinMemoryMb: 250
  recommendationOnly: false
  safetyMarginFraction: 0.15
```

1 为 VPA 准入控制器指定调优参数。

2 为 VPA 准入控制器指定 API QPS 和突发率。

- **kube-api-qps**: 指定向 Kubernetes API 服务器发出请求时的每秒查询 (QPS) 限制。默认值为 **5.0**。
- **kube-api-burst** ：指定向 Kubernetes API 服务器发出请求时的突发限制。默认值为 **10.0**。

3 指定 VPA 准入控制器 pod 的 CPU 和内存请求。

4 指定 VPA 建议器的调优参数。

5 指定 VPA Operator 只监控 VPA CR 的工作负载。默认值为 **false**。

6 指定 VPA updater 的调优参数。

您可以验证设置是否已应用到每个 VPA 组件 pod。

updater pod 示例

```
apiVersion: v1
kind: Pod
metadata:
  name: vpa-updater-default-d65ffb9dc-hgw44
  namespace: openshift-vertical-pod-autoscaler
# ...
spec:
  containers:
  - args:
    - --logtostderr
    - --v=1
    - --min-replicas=2
    - --kube-api-qps=20.0
    - --kube-api-burst=80.0
# ...
    resources:
      requests:
        cpu: 80m
        memory: 80Mi
# ...
```

准入控制器 pod 示例

```
apiVersion: v1
kind: Pod
metadata:
  name: vpa-admission-plugin-default-756999448c-l7tsd
  namespace: openshift-vertical-pod-autoscaler
# ...
spec:
  containers:
  - args:
    - --logtostderr
    - --v=1
    - --tls-cert-file=/data/tls-certs/tls.crt
    - --tls-private-key=/data/tls-certs/tls.key
    - --client-ca-file=/data/tls-ca-certs/service-ca.crt
    - --webhook-timeout-seconds=10
```

```
    - --kube-api-qps=30.0
    - --kube-api-burst=40.0
# ...
    resources:
      requests:
        cpu: 40m
        memory: 40Mi
# ...
```

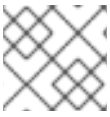
recommender pod 示例

```
apiVersion: v1
kind: Pod
metadata:
  name: vpa-recommender-default-74c979dbbc-znrd2
  namespace: openshift-vertical-pod-autoscaler
# ...
spec:
  containers:
  - args:
    - --logtostderr
    - --v=1
    - --recommendation-margin-fraction=0.15
    - --pod-recommendation-min-cpu-millicores=25
    - --pod-recommendation-min-memory-mb=250
    - --kube-api-qps=20.0
    - --kube-api-burst=60.0
    - --memory-saver=true
# ...
    resources:
      requests:
        cpu: 60m
        memory: 60Mi
# ...
```

2.5.4.7. 使用一个替代推荐器

您可以根据自己的算法使用自己的推荐器来自动扩展。如果您没有指定替代的推荐器，OpenShift Container Platform 会使用默认的推荐器，它会根据历史使用情况推荐 CPU 和内存请求。因为没有适用于所有工作负载的通用推荐策略，您可能需要为特定工作负载创建和部署不同的推荐器。

例如，当容器出现某些资源行为时，默认的推荐器可能无法准确预测将来的资源使用量，例如，在监控应用程序使用的使用量高峰和闲置间交替的模式，或者重复与深度学习应用程序使用的模式。将默认推荐器用于这些使用行为可能会导致应用程序的过度置备和内存不足(OOM)终止。



注意

有关如何创建推荐器的说明超出了本文档的范围，

流程

为 pod 使用替代推荐器：

1. 为替代推荐器创建服务帐户，并将该服务帐户绑定到所需的集群角色：
   -

```
apiVersion: v1 (1)
kind: ServiceAccount
metadata:
  name: alt-vpa-recommender-sa
  namespace: <namespace_name>
---
apiVersion: rbac.authorization.k8s.io/v1 (2)
kind: ClusterRoleBinding
metadata:
  name: system:example-metrics-reader
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:metrics-reader
subjects:
- kind: ServiceAccount
  name: alt-vpa-recommender-sa
  namespace: <namespace_name>
---
apiVersion: rbac.authorization.k8s.io/v1 (3)
kind: ClusterRoleBinding
metadata:
  name: system:example-vpa-actor
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:vpa-actor
subjects:
- kind: ServiceAccount
  name: alt-vpa-recommender-sa
  namespace: <namespace_name>
---
apiVersion: rbac.authorization.k8s.io/v1 (4)
kind: ClusterRoleBinding
metadata:
  name: system:example-vpa-target-reader-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:vpa-target-reader
subjects:
- kind: ServiceAccount
  name: alt-vpa-recommender-sa
  namespace: <namespace_name>
```

1. 在部署了推荐器的命名空间中为推荐器创建一个服务账户。
2. 将推进器服务帐户绑定到 **metrics-reader** 角色。指定要部署推进器的命名空间。
3. 将推进器服务帐户绑定到 **vpa-actor** 角色。指定要部署推进器的命名空间。
4. 将推进器服务帐户绑定到 **vpa-target-reader** 角色。指定要部署推进器的命名空间。

2. 要在集群中添加备选推荐程序，请创建一个类似如下的 Deployment 对象：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: alt-vpa-recommender
  namespace: <namespace_name>
spec:
  replicas: 1
  selector:
    matchLabels:
      app: alt-vpa-recommender
  template:
    metadata:
      labels:
        app: alt-vpa-recommender
    spec:
      containers: 1
      - name: recommender
        image: quay.io/example/alt-recommender:latest 2
        imagePullPolicy: Always
        resources:
          limits:
            cpu: 200m
            memory: 1000Mi
          requests:
            cpu: 50m
            memory: 500Mi
        ports:
        - name: prometheus
          containerPort: 8942
        securityContext:
          allowPrivilegeEscalation: false
          capabilities:
            drop:
              - ALL
          seccompProfile:
            type: RuntimeDefault
      serviceAccountName: alt-vpa-recommender-sa 3
      securityContext:
        runAsNonRoot: true
```

1 为您的备选推荐程序创建容器。

2 指定您的推荐镜像。

3 关联您为推荐器创建的服务帐户。

为同一命名空间中的备选推荐器创建新 pod。

```
$ oc get pods
```

输出示例

```
NAME                                READY   STATUS    RESTARTS   AGE
frontend-845d5478d-558zf            1/1     Running   0          4m25s
```

```
frontend-845d5478d-7z9gx                1/1     Running   0          4m25s
frontend-845d5478d-b7l4j                1/1     Running   0          4m25s
vpa-alt-recommender-55878867f9-6tp5v    1/1     Running   0          9s
```

3. 配置包含替代推荐器 **Deployment** 对象名称的 VPA CR。

VPA CR 示例，使其包含替代的推荐程序

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
  namespace: <namespace_name>
spec:
  recommenders:
    - name: alt-vpa-recommender 1
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment 2
    name:       frontend
```

1 指定替代推荐器部署的名称。

2 指定您希望此 VPA 管理的现有工作负载对象的名称。

2.5.5. 使用 Vertical Pod Autoscaler Operator

您可以通过创建 VPA 自定义资源（CR）来使用 Vertical Pod Autoscaler Operator（VPA）。CR 指明应分析哪些 pod，并决定 VPA 应该对这些 pod 执行的操作。

先决条件

- 要自动扩展的工作负载对象必须存在。
- 如果要使用替代的推荐器，则必须存在包括那个推进器的部署。

流程

为特定工作负载对象创建 VPA CR:

1. 切换到您要缩放的工作负载对象所在的项目。
   a. 创建一个 VPA CR YAML 文件：

```
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: vpa-recommender
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind:       Deployment 1
    name:       frontend 2
  updatePolicy:
```

```
    updateMode: "Auto" 3
  resourcePolicy: 4
    containerPolicies:
    - containerName: my-opt-sidecar
      mode: "Off"
  recommenders: 5
    - name: my-recommender
```

1. 指定您需要这个 VPA 管理的工作负载对象类型：**Deployment**、**StatefulSet**、**Job**、**DaemonSet**、**ReplicaSet** 或 **ReplicationController**。
2. 指定您希望此 VPA 管理的现有工作负载对象的名称。
3. 指定 VPA 模式：
   - **auto** 会在与控制器关联的 pod 上自动应用推荐的资源。VPA 会终止现有的 pod，并使用推荐的资源限制和请求创建新 pod。
   - **recreate** 会在与工作负载对象关联的 pod 上自动应用推荐的资源。VPA 会终止现有的 pod，并使用推荐的资源限制和请求创建新 pod。**recreate** 模式应该很少使用，只有在需要确保每当资源请求改变时 pod 就需要重启时才使用。
   - **Initial** 在创建与工作负载对象关联的 pod 时自动应用推荐的资源。VPA 会学习新的资源建议，但不会更新 pod。
   - **off** 仅为与工作负载对象关联的 pod 生成资源建议。VPA 不会更新 pod，它只会学习新的资源建议，且不会将建议应用到新 pod。
4. 可选。指定不需要受 VPA 影响的容器，将模式设置为 **Off**。
5. 可选。指定替代的推荐器。

b. 创建 VPA CR:

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

在一段短暂的时间后，VPA 会了解与工作负载对象关联的 pod 中容器的资源使用情况。

您可以使用以下命令查看 VPA 建议：

```
$ oc get vpa <vpa-name> --output yaml
```

输出显示 CPU 和内存请求的建议，如下所示：

输出示例

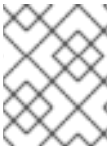
```
...
status:

...

  recommendation:
    containerRecommendations:
```

```
  - containerName: frontend
    lowerBound: 1
      cpu: 25m
      memory: 262144k
    target: 2
      cpu: 25m
      memory: 262144k
    uncappedTarget: 3
      cpu: 25m
      memory: 262144k
    upperBound: 4
      cpu: 262m
      memory: "274357142"
  - containerName: backend
    lowerBound:
      cpu: 12m
      memory: 131072k
    target:
      cpu: 12m
      memory: 131072k
    uncappedTarget:
      cpu: 12m
      memory: 131072k
    upperBound:
      cpu: 476m
      memory: "498558823"

...
```

1. **lowerBound** 是最低的推荐资源级别。
2. **target**是推荐的资源级别。
3. **upperBound** 是最高的推荐资源级别。
4. **uncappedTarget** 是最新资源建议。

2.5.6. 卸载 Vertical Pod Autoscaler Operator

您可以从 OpenShift Container Platform 集群中删除 Vertical Pod Autoscaler Operator（VPA）。卸载后，已由现有 VPA CR 修改的 pod 的资源请求不会改变。任何新 pod 都会根据工作负载对象中的定义获得资源，而不是之前由 VPA 提供的的建议。



注意

您可以使用 **oc delete vpa <vpa-name>** 命令删除特定的 VPA CR。在卸载垂直 pod 自动扩展时，同样的操作适用于资源请求。

删除 VPA Operator 后，建议您删除与 Operator 相关的其他组件，以避免潜在的问题。

先决条件

- 已安装 Vertical Pod Autoscaler Operator。

流程

1. 在 OpenShift Container Platform web 控制台中，点击 **Operators → Installed Operators**。
2. 切换到 **openshift-vertical-pod-autoscaler** 项目。
3. 对于 **VerticalPodAutoscaler** Operator，点 Options 菜单 并选择 **Uninstall Operator**。
4. 可选： 要删除与 Operator 关联的所有操作对象，请在对话框中选择 **Delete all operand instance for this operator** 复选框。
5. 点 **Uninstall**。
6. 可选： 使用 OpenShift CLI 删除 VPA 组件：
   a. 删除 VPA 命名空间：

```
$ oc delete namespace openshift-vertical-pod-autoscaler
```

   b. 删除 VPA 自定义资源定义 (CRD) 对象：

```
$ oc delete crd verticalpodautoscalercheckpoints.autoscaling.k8s.io
```

```
$ oc delete crd verticalpodautoscalercontrollers.autoscaling.openshift.io
```

```
$ oc delete crd verticalpodautoscalers.autoscaling.k8s.io
```

   删除 CRD 会删除关联的角色、集群角色和角色绑定。

注意

此操作会从集群中移除，集群中的所有用户创建的 VPA CR。如果重新安装 VPA，您必须再次创建这些对象。

   c. 运行以下命令来删除 **MutatingWebhookConfiguration** 对象：

```
$ oc delete MutatingWebhookConfiguration vpa-webhook-config
```

   d. 删除 VPA Operator：

```
$ oc delete operator/vertical-pod-autoscaler.openshift-vertical-pod-autoscaler
```

2.6. 使用 SECRET 为 POD 提供敏感数据

有些应用程序需要密码和用户名等敏感信息，但您不希望开发人员持有这些信息。

作为管理员，您可以使用 **Secret** 对象在不以明文方式公开的前提下提供此类信息。

2.6.1. 了解 secret

Secret 对象类型提供了一种机制来保存敏感信息，如密码、OpenShift Container Platform 客户端配置文件和私有源存储库凭证等。secret 将敏感内容与 Pod 分离。您可以使用卷插件将 secret 信息挂载到容器中，系统也可以使用 secret 代表 Pod 执行操作。

主要属性包括：

- Secret 数据可以独立于其定义来引用。
- Secret 数据卷由临时文件工具 (tmpfs) 支持，永远不会停留在节点上。
- secret 数据可以在命名空间内共享。

YAML Secret 对象定义

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque (1)
data: (2)
  username: <username> (3)
  password: <password>
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

1. 指示 secret 的键和值的结构。
2. **data** 字段中允许的键格式必须符合 Kubernetes 标识符术语表 中 **DNS_SUBDOMAIN** 值的规范。
3. 与 **data** 映射中键关联的值必须采用 base64 编码。
4. **stringData** 映射中的条目将转换为 base64，然后该条目将自动移动到 **data** 映射中。此字段是只写的；其值仅通过 **data** 字段返回。
5. 与 **stringData** 映射中键关联的值由纯文本字符串组成。

您必须先创建 secret，然后创建依赖于此 secret 的 Pod。

在创建 secret 时：

- 使用 secret 数据创建 secret 对象。
- 更新 pod 的服务帐户以允许引用该 secret。
- 创建以环境变量或文件（使用 **secret** 卷）形式消耗 secret 的 pod。

2.6.1.1. secret 的类型

type 字段中的值指明 secret 的键名称和值的结构。此类型可用于强制使 secret 对象中存在用户名和密钥。如果您不想进行验证，请使用 **opaque** 类型，这也是默认类型。

指定以下一种类型来触发最小服务器端验证，确保 secret 数据中存在特定的键名称：

- **kubernetes.io/basic-auth**：使用基本身份验证

- **kubernetes.io/dockercfg**：用作镜像 pull secret
- **kubernetes.io/dockerconfigjson**: 用作镜像 pull secret
- **kubernetes.io/service-account-token**：用来获取旧的服务帐户 API 令牌
- **kubernetes.io/ssh-auth**：与 SSH 密钥身份验证一起使用
- **kubernetes.io/tls**：与 TLS 证书颁发机构一起使用

如果您不想要验证，请指定 **type: Opaque**，即 secret 没有声明键名称或值需要符合任何约定。*opaque* secret 允许使用无结构 **key:value** 对，可以包含任意值。

注意

您可以指定其他任意类型，如 **example.com/my-secret-type**。这些类型不是在服务器端强制执行，而是表明 secret 的创建者意在符合该类型的键/值要求。

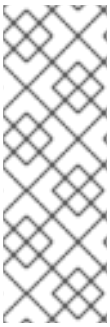
有关创建不同类型的 secret 的示例，请参阅 *了解如何创建 secret*。

2.6.1.2. Secret 数据密钥

Secret 密钥必须在 DNS 子域中。

2.6.1.3. 自动生成的镜像 pull secret

默认情况下，OpenShift Container Platform 为每个服务帐户创建一个镜像 pull secret。



注意

在 OpenShift Container Platform 4.16 之前，还会为创建的每个服务帐户生成长期服务帐户 API 令牌 secret。从 OpenShift Container Platform 4.16 开始，不再创建此服务帐户 API 令牌 secret。

升级到 4.16 后，任何现有的长期服务帐户 API 令牌 secret 都不会被删除，并将继续正常工作。有关检测集群中使用的长期 API 令牌，以及在不需要时删除它们的信息，请参阅红帽知识库文章 OpenShift Container Platform 中的 Long-lived 服务帐户 API 令牌 。

此镜像 pull secret 需要将 OpenShift 镜像 registry 集成到集群的用户身份验证和授权系统中。

但是，如果您不启用 **ImageRegistry** 功能，或者在 Cluster Image Registry Operator 配置中禁用集成的 OpenShift 镜像 registry，则不会为每个服务帐户生成镜像 pull secret。

当在之前启用的集群中禁用集成的 OpenShift 镜像 registry 时，之前生成的镜像 pull secret 会被自动删除。

2.6.2. 了解如何创建 secret

作为管理员，您必须先创建 secret，然后开发人员才能创建依赖于该 secret 的 pod。

在创建 secret 时：

1. 创建包含您要保留 secret 的数据的 secret 对象。在以下部分中取消每个 secret 类型所需的特定数据。

创建不透明 secret 的 YAML 对象示例

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
type: Opaque (1)
data: (2)
  username: <username>
  password: <password>
stringData: (3)
  hostname: myapp.mydomain.com
  secret.properties: |
    property1=valueA
    property2=valueB
```

1. 指定 secret 的类型。
2. 指定编码的字符串和数据。
3. 指定解码的字符串和数据。

使用 **data** 或 **stringdata** 字段，不能同时使用这两个字段。

2. 更新 pod 的服务帐户以引用 secret：

使用 secret 的服务帐户的 YAML

```
apiVersion: v1
kind: ServiceAccount
 ...
secrets:
- name: test-secret
```

3. 创建以环境变量或文件（使用 **secret** 卷）形式消耗 secret 的 pod：

pod 的 YAML 使用 secret 数据填充卷中的文件

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts: (1)
          - name: secret-volume
            mountPath: /etc/secret-volume (2)
```

```
        readOnly: true 3
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret 4
  restartPolicy: Never
```

1 为每个需要 secret 的容器添加 **volumeMounts** 字段。

2 指定您希望显示 secret 的未使用目录名称。secret 数据映射中的每个密钥都将成为 **mountPath** 下的文件名。

3 设置为 **true**。如果为 true，这指示驱动程序提供只读卷。

4 指定 secret 的名称。

pod 的 YAML 使用 secret 数据填充环境变量

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef: 1
              name: test-secret
              key: username
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  restartPolicy: Never
```

1 指定消耗 secret 密钥的环境变量。

构建配置的 YAML 使用 secret 数据填充环境变量

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
```

```
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef: 1
            name: test-secret
            key: username
      from:
        kind: ImageStreamTag
        namespace: openshift
        name: 'cli:latest'
```

1 指定消耗 secret 密钥的环境变量。

2.6.2.1. Secret 创建限制

若要使用 secret，pod 需要引用该 secret。可以通过三种方式将 secret 用于 Pod：

- 为容器产生环境变量。
- 作为挂载到一个或多个容器上的卷中的文件。
- 在拉取 Pod 的镜像时通过 kubelet 使用。

卷类型 secret 使用卷机制将数据作为文件写入到容器中。镜像拉取 secret 使用服务帐户，将 secret 自动注入到命名空间中的所有 pod。

当模板包含 secret 定义时，模板使用提供的 secret 的唯一方法是确保验证 secret 卷源通过验证，并且指定的对象引用实际指向 **Secret** 类型的对象。因此，secret 需要在依赖它的任何 Pod 之前创建。确保这一点的最有效方法是通过使用服务帐户自动注入。

Secret API 对象驻留在命名空间中。它们只能由同一命名空间中的 pod 引用。

每个 secret 的大小限制为 1MB。这是为了防止创建可能会耗尽 apiserver 和 kubelet 内存的大型 secret。不过，创建许多较小的 secret 也可能会耗尽内存。

2.6.2.2. 创建不透明 secret

作为管理员，您可以创建一个不透明 secret，它允许您存储包含任意值的无结构 **key:value** 对。

流程

1. 在控制平面节点上的 YAML 文件中创建 **Secret** 对象。
   例如：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
```

```
data:
  username: <username>
  password: <password>
```

1 指定不透明 secret。

2. 使用以下命令来创建 **Secret** 对象：

```
$ oc create -f <filename>.yaml
```

3. 在 pod 中使用该 secret:

   a. 更新 pod 的服务帐户以引用 secret，如 "Understanding how to create secrets" 部分所示。

   b. 创建以环境变量或文件（使用 secret 卷）形式消耗 **secret** 的 pod，如"创建 secret"部分所示。

其他资源

- 了解如何创建 secret

2.6.2.3. 创建旧的服务帐户令牌 secret

作为管理员，您可以创建一个旧的服务帐户令牌 secret，该 secret 允许您将服务帐户令牌分发到必须通过 API 进行身份验证的应用程序。

警告

建议您使用 TokenRequest API 获取绑定的服务帐户令牌，而不使用旧的服务帐户令牌 secret。只有在无法使用 TokenRequest API 且在可读的 API 对象中存在非过期令牌时，才应创建服务帐户令牌 secret。

绑定服务帐户令牌比服务帐户令牌 secret 更安全，原因如下：

- 绑定服务帐户令牌具有绑定的生命周期。
- 绑定服务帐户令牌包含受众。
- 绑定服务帐户令牌可以绑定到 pod 或 secret，绑定令牌在删除绑定对象时无效。

工作负载自动注入投射卷以获取绑定服务帐户令牌。如果您的工作负载需要额外的服务帐户令牌，请在工作负载清单中添加额外的投射卷。

如需更多信息，请参阅"使用卷投射配置绑定服务帐户令牌"。

流程

1. 在控制平面节点上的 YAML 文件中创建 **Secret** 对象：

Secret 对象示例

Secret 对象示例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name" 1
type: kubernetes.io/service-account-token 2
```

1. 指定一个现有服务帐户名称。如果您要同时创建 **ServiceAccount** 和 **Secret** 对象，请首先创建 **ServiceAccount** 对象。
2. 指定服务帐户令牌 secret。

2. 使用以下命令来创建 **Secret** 对象：

```
$ oc create -f <filename>.yaml
```

3. 在 pod 中使用该 secret:
   a. 更新 pod 的服务帐户以引用 secret，如 "Understanding how to create secrets" 部分所示。
   b. 创建以环境变量或文件（使用 secret 卷）形式消耗 **secret** 的 pod，如"创建 secret"部分所示。

其他资源

- 了解如何创建 secret
- 使用卷投射配置绑定服务帐户令牌
- 了解并创建服务帐户

2.6.2.4. 创建基本身份验证 secret

作为管理员，您可以创建一个基本身份验证 secret，该 secret 允许您存储基本身份验证所需的凭证。在使用此 secret 类型时，**Secret** 对象的 **data** 参数必须包含以下密钥，采用 base64 格式编码：

- **用户名** ：用于身份验证的用户名
- **密码** ：用于身份验证的密码或令牌

注意

您可以使用 **stringData** 参数使用明文内容。

流程

1. 在控制平面节点上的 YAML 文件中创建 **Secret** 对象：

secret 对象示例

```
apiVersion: v1
```

```
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth (1)
data:
stringData: (2)
  username: admin
  password: <password>
```

(1) 指定基本身份验证 secret。

(2) 指定要使用的基本身份验证值。

2. 使用以下命令来创建 **Secret** 对象：

```
$ oc create -f <filename>.yaml
```

3. 在 pod 中使用该 secret:

   a. 更新 pod 的服务帐户以引用 secret，如 "Understanding how to create secrets" 部分所示。

   b. 创建以环境变量或文件（使用 secret 卷）形式消耗 **secret** 的 pod，如"创建 secret"部分所示。

其他资源

- 了解如何创建 secret

2.6.2.5. 创建 SSH 身份验证 secret

作为管理员，您可以创建一个 SSH 验证 secret，该 secret 允许您存储用于 SSH 验证的数据。在使用此 secret 类型时，**Secret** 对象的 **data** 参数必须包含要使用的 SSH 凭证。

流程

1. 在控制平面节点上的 YAML 文件中创建 **Secret** 对象：

   secret 对象示例：

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth (1)
data:
  ssh-privatekey: | (2)
        MIIEpQIBAAKCAQEAulqb/Y ...
```

(1) 指定 SSH 身份验证 secret。

(2) 指定 SSH 密钥/值对，作为要使用的 SSH 凭据。

2. 使用以下命令来创建 **Secret** 对象：

```
$ oc create -f <filename>.yaml
```

3. 在 pod 中使用该 secret:

   a. 更新 pod 的服务帐户以引用 secret，如 "Understanding how to create secrets" 部分所示。

   b. 创建以环境变量或文件（使用 secret 卷）形式消耗 **secret** 的 pod，如"创建 secret"部分所示。

其他资源

- 了解如何创建 secret

2.6.2.6. 创建 Docker 配置 secret

作为管理员，您可以创建一个 Docker 配置 secret，该 secret 允许您存储用于访问容器镜像 registry 的凭证。

- **kubernetes.io/dockercfg**。使用此机密类型存储本地 Docker 配置文件。**secret** 对象的 **data** 参数必须包含以 base64 格式编码的 **.dockercfg** 文件的内容。
- **kubernetes.io/dockerconfigjson**。使用此机密类型存储本地 Docker 配置 JSON 文件。**secret** 对象的 **data** 参数必须包含以 base64 格式编码的 **.docker/config.json** 文件的内容。

流程

1. 在控制平面节点上的 YAML 文件中创建 **Secret** 对象。

Docker 配置 secret 对象示例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-docker-cfg
  namespace: my-project
type: kubernetes.io/dockerconfig (1)
data:

.dockerconfig:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cgYXV
0aCBrZXlzCg== (2)
```

(1) 指定该 secret 使用 Docker 配置文件。

(2) base64 编码的 Docker 配置文件

Docker 配置 JSON secret 对象示例

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-docker-json
```

```
  namespace: my-project
type: kubernetes.io/dockerconfig (1)
data:

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXlzCg== (2)
```

1. 指定该 secret 使用 Docker 配置 JSONfile。
2. base64 编码的 Docker 配置 JSON 文件

2. 使用以下命令来创建 **Secret** 对象

```
$ oc create -f <filename>.yaml
```

3. 在 pod 中使用该 secret:
   a. 更新 pod 的服务帐户以引用 secret，如 "Understanding how to create secrets" 部分所示。
   b. 创建以环境变量或文件（使用 secret 卷）形式消耗 **secret** 的 pod，如"创建 secret"部分所示。

其他资源

- 了解如何创建 secret

2.6.2.7. 使用 Web 控制台创建 secret

您可以使用 Web 控制台创建 secret。

流程

1. 导航到 **Workloads → Secrets**。
2. 点 **Create → From YAML**。
   a. 手动编辑您的规格的 YAML，或者将文件拖放到 YAML 编辑器。例如：

```
apiVersion: v1
kind: Secret
metadata:
  name: example
  namespace: <namespace>
type: Opaque (1)
data:
  username: <base64 encoded username>
  password: <base64 encoded password>
stringData: (2)
  hostname: myapp.mydomain.com
```

1. 本例指定了一个 opaque secret，但您可以看到其他 secret 类型，如服务帐户令牌 secret、基本身份验证 secret、SSH 身份验证 secret 或使用 Docker 配置的 secret。
2. **stringData** 映射中的条目将转换为 base64，然后该条目将自动移动到 **data** 映射中。此

3. 点 **Create**。
4. 点 **Add Secret to workload**。
   a. 从下拉菜单中选择要添加的工作负载。
   b. 点击 **Save**。

2.6.3. 了解如何更新 secret

修改 secret 值时，值（由已在运行的 pod 使用）不会动态更改。若要更改 secret，您必须删除原始 pod 并创建一个新 pod（可能具有相同的 PodSpec）。

更新 secret 遵循与部署新容器镜像相同的工作流程。您可以使用 **kubectl rolling-update** 命令。

secret 中的 **resourceVersion** 值不在引用时指定。因此，如果在 pod 启动的同时更新 secret，则将不能定义用于 pod 的 secret 版本。

注意

目前，无法检查 Pod 创建时使用的 secret 对象的资源版本。按照计划 Pod 将报告此信息，以便控制器可以重启使用旧 **resourceVersion** 的 Pod。在此期间，请勿更新现有 secret 的数据，而应创建具有不同名称的新数据。

2.6.4. 创建和使用 secret

作为管理员，您可以创建一个服务帐户令牌 secret。这可让您将服务帐户令牌分发到必须通过 API 进行身份验证的应用程序。

流程

1. 运行以下命令，在命名空间中创建服务帐户：

```
$ oc create sa <service_account_name> -n <your_namespace>
```

2. 将以下 YAML 示例保存到名为 **service-account-token-secret.yaml** 的文件中。这个示例包括可用于生成服务帐户令牌的 **Secret** 对象配置：

```
apiVersion: v1
kind: Secret
metadata:
  name: <secret_name> (1)
  annotations:
    kubernetes.io/service-account.name: "sa-name" (2)
type: kubernetes.io/service-account-token (3)
```

(1) 将 **<secret_name>** 替换为服务帐户令牌 secret 的名称。

(2) 指定一个现有服务帐户名称。如果您要同时创建 **ServiceAccount** 和 **Secret** 对象，请首先创建 **ServiceAccount** 对象。

(3) 指定服务帐户令牌 secret 类型。

3. 通过应用文件来生成服务帐户令牌：

```
$ oc apply -f service-account-token-secret.yaml
```

4. 运行以下命令，从 secret 获取服务帐户令牌：

```
$ oc get secret <sa_token_secret> -o jsonpath='{.data.token}' | base64 --decode 1
```

输出示例

```
ayJhbGciOiJSUzI1NiIsImtpZCI6IklOb2dtck1qZ3hCSWpoNnh5YnZhSE9QMkk3YnRZMVZoclFf
QTZfRFp1YlUifQ.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5
pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZX
J2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6ImJ1aWxkZXItdG9rZW4tdHZrbnIiLCJrdWJlcm5l
dGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiYnVpbGRlciIsImt1
YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjNmZGU
2MGZmLTA1NGYtNDkyZi04YzhjLTNlZjE0NDk3MmFmNyIsInN1YiI6InN5c3RlbTpzZXJ2aWNl
YWNjb3VudDpkZWZhdWx0OmJ1aWxkZXIifQ.OmqFTDuMHC_lYvvEUrjr1x453hlEEHYcxS9VK
SzmRkP1SiVZWPNPkTWlfNRp6bIUZD3U6aN3N7dMSN0eI5hu36xPgpKTdvuckKLTCnelMx6c
xOdAbrcw1mCmOClNscwjS1KO1kzMtYnnq8rXHiMJELsNlhnRyyIXRTtNBsy4t64T3283s3SLsa
ncyx0gy0ujx-Ch3uKAKdZi5iT-I8jnnQ-ds5THDs2h65RJhgglQEmSxpHrLGZFmyHAQI-
_SjvmHZPXEc482x3SkaQHNLqpmrpJorNqh1M8ZHKzlujhZgVooMvJmWPXTb2vnvi3DGn2XI-
hZxI1yD2yGH1RBpYUHA
```

1 将 <sa_token_secret> 替换为服务帐户令牌 secret 的名称。

5. 使用您的服务帐户令牌与集群的 API 进行身份验证：

```
$ curl -X GET <openshift_cluster_api> --header "Authorization: Bearer <token>" 1 2
```

1 将 **<openshift_cluster_api>** 替换为 OpenShift 集群 API。

2 将 **<token>** 替换为上一命令输出的服务帐户令牌。

2.6.5. 关于将签名证书与 secret 搭配使用

若要与服务进行安全通信，您可以配置 OpenShift Container Platform，以生成一个签名的服务用证书/密钥对，再添加到项目中的 secret 里。

服务用证书 secret 旨在支持需要开箱即用证书的复杂中间件应用程序。它的设置与管理员工具为节点和 master 生成的服务器证书相同。

为服务用证书 secret 配置的服务 Pod 规格。

```
apiVersion: v1
kind: Service
metadata:
  name: registry
  annotations:
    service.beta.openshift.io/serving-cert-secret-name: registry-cert 1
# ...
```

1 指定证书的名称

其他 pod 可以信任集群创建的证书（仅对内部 DNS 名称进行签名），方法是使用 pod 中自动挂载的 */var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt* 文件中的 CA 捆绑。

此功能的签名算法是 **x509.SHA256WithRSA**。要手动轮转，请删除生成的 secret。这会创建新的证书。

2.6.5.1. 生成签名证书以便与 secret 搭配使用

要将签名的服务用证书/密钥对用于 pod，请创建或编辑服务以添加到 **service.beta.openshift.io/serving-cert-secret-name** 注解，然后将 secret 添加到该 pod。

流程

创建*服务用证书 secret*：

1. 编辑服务的 **Pod** spec。
2. 使用您要用于 secret 的名称，添加 **service.beta.openshift.io/serving-cert-secret-name** 注解。

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
  annotations:
     service.beta.openshift.io/serving-cert-secret-name: my-cert 1
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

证书和密钥采用 PEM 格式，分别存储在 **tls.crt** 和 **tls.key** 中。

3. 创建服务：

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

4. 查看 secret 以确保已成功创建：
   a. 查看所有 secret 列表：

```
$ oc get secrets
```

输出示例

```
NAME                  TYPE                                  DATA      AGE
my-cert               kubernetes.io/tls                     2         9m
```

   b. 查看您的 secret 详情：

```
$ oc describe secret my-cert
```

输出示例

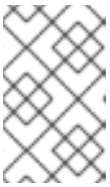
```
Name:         my-cert
Namespace:    openshift-console
Labels:       <none>
Annotations:  service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z
              service.beta.openshift.io/originating-service-name: my-service
              service.beta.openshift.io/originating-service-uid: 640f0ec3-afc2-4380-bf31-
a8c784846a11
              service.beta.openshift.io/expiry: 2023-03-08T23:22:40Z

Type:  kubernetes.io/tls

Data
====
tls.key:  1679 bytes
tls.crt:  2595 bytes
```

5. 编辑与该 secret 搭配的 **Pod** spec。

```
apiVersion: v1
kind: Pod
metadata:
  name: my-service-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: my-container
      mountPath: "/etc/my-path"
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
  - name: my-volume
    secret:
      secretName: my-cert
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

当它可用时，您的 Pod 就可运行。该证书对内部服务 DNS 名称 **<service.name>.<service.namespace>.svc** 有效。

证书/密钥对在接近到期时自动替换。在 secret 的 **service.beta.openshift.io/expiry** 注解中查看过期日期，其格式为 RFC3339。



注意

在大多数情形中，服务 DNS 名称 **<service.name>.<service.namespace>.svc** 不可从外部路由。**<service.name>.<service.namespace>.svc** 的主要用途是集群内或服务内通信，也用于重新加密路由。

2.6.6. secret 故障排除

如果服务证书生成失败并显示以下信息（ 服务的 **service.beta.openshift.io/serving-cert-generation-error** 注解包含）：

```
secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not match 77b6dd80-d716-11e6-9d6f-0e9c0057b60
```

生成证书的服务不再存在，或者具有不同的 **serviceUID** 。您必须删除旧 secret 并清除服务上的以下注解 **service.beta.openshift.io/serving-cert-generation-error**, **service.beta.openshift.io/serving-cert-generation-error-num** 以强制重新生成证书：

1. 删除 secret：

```
$ oc delete secret <secret_name>
```

2. 清除注解：

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-
```

注意

在用于移除注解的命令中，要移除的注解后面有一个 **-**。

2.7. 使用外部 SECRET 存储为 POD 提供敏感数据

有些应用程序需要密码和用户名等敏感信息，但您不希望开发人员持有这些信息。

使用 Kubernetes **Secret** 对象提供敏感信息的一个替代选择是，使用外部 secret 存储来存储敏感信息。您可以使用 Secrets Store CSI Driver Operator 与外部 secret 存储集成，并将 secret 内容挂载为 pod 卷。

重要

Secret Store CSI Driver Operator 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

2.7.1. 关于 Secret Store CSI Driver Operator

Kubernetes secret 以 Base64 编码的形式存储。etcd 为这些 secret 提供加密，但在检索 secret 时，它们会被解密并提供给用户。如果没有在集群中正确配置基于角色的访问控制，则具有 API 或 etcd 访问权限

的任何人都可以检索或修改 secret。另外，有权在命名空间中创建 pod 的任何人都可以使用该命名空间中的任何 secret 来读取该命名空间中的任何 secret。

要安全地存储和管理您的 secret，您可以将 OpenShift Container Platform Secrets Store Container Storage Interface (CSI) Driver Operator 配置为使用供应商插件从外部 secret 管理系统（如 Azure Key Vault）挂载 secret。应用程序可以使用 secret，但 secret 在应用程序 pod 被销毁后不会在系统中保留。

Secret Store CSI Driver Operator（**secrets-store.csi.k8s.io**）允许 OpenShift Container Platform 将存储在企业级外部 secret 中的多个 secret、密钥和证书作为卷挂载到 pod 中。Secrets Store CSI Driver Operator 使用 gRPC 与供应商通信，以从指定的外部 secret 存储获取挂载内容。附加卷后，其中的数据将挂载到容器的文件系统。Secret 存储卷以 in-line 形式挂载。

2.7.1.1. Secret 存储供应商

以下 secret 存储供应商可用于 Secret Store CSI Driver Operator：

- AWS Secrets Manager
- AWS Systems Manager Parameter Store
- Azure Key Vault
- HashiCorp Vault

2.7.1.2. 自动轮转

Secrets Store CSI 驱动程序会定期使用外部 secret 存储中的内容轮转挂载卷中的内容。如果外部 secret 存储中更新了 secret，secret 将在挂载的卷中更新。Secrets Store CSI Driver Operator 每 2 分钟轮询一次更新。

如果启用了将挂载内容作为 Kubernetes secret 同步，则 Kubernetes secret 也会被轮转。

使用 secret 数据的应用程序必须监视是否有对 secret 的更新。

2.7.2. 安装 Secret Store CSI 驱动程序

先决条件

- 访问 OpenShift Container Platform Web 控制台。
- 集群的管理员访问权限。

流程

安装 Secret Store CSI 驱动程序：

1. 安装 Secret Store CSI Driver Operator：
   a. 登录到 web 控制台。
   b. 点 **Operators → OperatorHub**。
   c. 通过在过滤器框中输入 "Secrets Store CSI" 来查找 Secrets Store CSI Driver Operator。
   d. 点 **Secrets Store CSI Driver Operator** 按钮。

e. 在 **Secrets Store CSI Driver Operator** 页面中，点 **Install**。

f. 在 **Install Operator** 页面中，确保：

- 选择 **All namespaces on the cluster (default)**。
- **安装的命名空间** 被设置为 **openshift-cluster-csi-drivers**。

g. 点 **Install**。
安装完成后，Secret Store CSI Driver Operator 会在 web 控制台的 **Installed Operators** 部分列出。

2. 为驱动程序创建 **ClusterCSIDriver** 实例 (**secrets-store.csi.k8s.io**)：

a. 点 **Administration** → **CustomResourceDefinitions** → **ClusterCSIDriver**。

b. 在 **Instances** 选项卡上，单击 **Create ClusterCSIDriver**。
使用以下 YAML 文件：

```
apiVersion: operator.openshift.io/v1
kind: ClusterCSIDriver
metadata:
    name: secrets-store.csi.k8s.io
spec:
  managementState: Managed
```

c. 点 **Create**。

2.7.3. 将 secret 从外部 secret 存储挂载到 CSI 卷

安装 Secret Store CSI Driver Operator 后，您可以将 secret 从以下外部 secret 存储挂载到 CSI 卷：

- AWS Secrets Manager
- AWS Systems Manager Parameter Store
- Azure Key Vault
- HashiCorp Vault

2.7.3.1. 从 AWS Secrets Manager 挂载 secret

您可以使用 Secrets Store CSI Driver Operator 将 secret 从 AWS Secrets Manager 挂载到 OpenShift Container Platform 中的 CSI 卷。要从 AWS Secrets Manager 挂载 secret，您的集群必须安装在 AWS 上，并使用 AWS 安全令牌服务 (STS)。

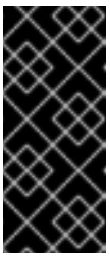
先决条件

- 您的集群安装在 AWS 上，并使用 AWS 安全令牌服务 (STS)。
- 已安装 Secrets Store CSI Driver Operator。具体步骤请参阅 *安装 Secret Store CSI 驱动程序*。
- 您已将 AWS Secrets Manager 配置为存储所需的 secret。
- 您已提取并准备好 **ccoctl** 二进制文件。

- 已安装 **jq** CLI 工具。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 安装 AWS Secrets Manager 供应商：
   a. 使用供应商资源的以下配置创建一个 YAML 文件：



重要

Secret Store CSI 驱动程序的 AWS Secrets Manager 供应商是一个上游供应商。

此配置会根据上游 AWS 文档中提供的配置进行修改，以便它可以与 OpenShift Container Platform 正常工作。对此配置的更改可能会影响功能。

aws-provider.yaml 文件示例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-secrets-store-provider-aws
  namespace: openshift-cluster-csi-drivers
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csi-secrets-store-provider-aws-cluster-role
rules:
- apiGroups: [""]
  resources: ["serviceaccounts/token"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["serviceaccounts"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: csi-secrets-store-provider-aws-cluster-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: csi-secrets-store-provider-aws-cluster-role
subjects:
- kind: ServiceAccount
  name: csi-secrets-store-provider-aws
```

```
  namespace: openshift-cluster-csi-drivers
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: openshift-cluster-csi-drivers
  name: csi-secrets-store-provider-aws
  labels:
    app: csi-secrets-store-provider-aws
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: csi-secrets-store-provider-aws
  template:
    metadata:
      labels:
        app: csi-secrets-store-provider-aws
    spec:
      serviceAccountName: csi-secrets-store-provider-aws
      hostNetwork: false
      containers:
        - name: provider-aws-installer
          image: public.ecr.aws/aws-secrets-manager/secrets-store-csi-driver-provider-
aws:1.0.r2-50-g5b4aca1-2023.06.09.21.19
          imagePullPolicy: Always
          args:
              - --provider-volume=/etc/kubernetes/secrets-store-csi-providers
          resources:
            requests:
              cpu: 50m
              memory: 100Mi
            limits:
              cpu: 50m
              memory: 100Mi
          securityContext:
            privileged: true
          volumeMounts:
            - mountPath: "/etc/kubernetes/secrets-store-csi-providers"
              name: providervol
            - name: mountpoint-dir
              mountPath: /var/lib/kubelet/pods
              mountPropagation: HostToContainer
      tolerations:
      - operator: Exists
      volumes:
        - name: providervol
          hostPath:
            path: "/etc/kubernetes/secrets-store-csi-providers"
        - name: mountpoint-dir
          hostPath:
            path: /var/lib/kubelet/pods
            type: DirectoryOrCreate
      nodeSelector:
        kubernetes.io/os: linux
```

b. 运行以下命令，授予 **csi-secrets-store-provider-aws** 服务帐户的特权访问权限：

```
$ oc adm policy add-scc-to-user privileged -z csi-secrets-store-provider-aws -n openshift-cluster-csi-drivers
```

c. 运行以下命令来创建供应商资源：

```
$ oc apply -f aws-provider.yaml
```

2. 授予服务帐户读取 AWS secret 对象的权限：

a. 运行以下命令，创建一个目录使其包含凭证请求：

```
$ mkdir credentialsrequest-dir-aws
```

b. 使用以下配置为凭证请求创建 YAML 文件：

credentialsrequest.yaml 文件示例

```
apiVersion: cloudcredential.openshift.io/v1
kind: CredentialsRequest
metadata:
  name: aws-provider-test
  namespace: openshift-cloud-credential-operator
spec:
  providerSpec:
    apiVersion: cloudcredential.openshift.io/v1
    kind: AWSProviderSpec
    statementEntries:
    - action:
      - "secretsmanager:GetSecretValue"
      - "secretsmanager:DescribeSecret"
      effect: Allow
      resource: "arn:*:secretsmanager:*:*:secret:testSecret-??????"
  secretRef:
    name: aws-creds
    namespace: my-namespace
  serviceAccountNames:
  - aws-provider
```

c. 运行以下命令来检索 OIDC 供应商：

```
$ oc get --raw=/.well-known/openid-configuration | jq -r '.issuer'
```

输出示例

```
https://<oidc_provider_name>
```

从输出中复制 OIDC 供应商名称 **<oidc_provider_name>**，在下一步中使用。

d. 运行以下命令，使用 **ccoctl** 工具处理凭证请求：

```
$ ccoctl aws create-iam-roles \
```

```
    --name my-role --region=<aws_region> \
    --credentials-requests-dir=credentialsrequest-dir-aws \
    --identity-provider-arn arn:aws:iam::<aws_account>:oidc-
provider/<oidc_provider_name> --output-dir=credrequests-ccoctl-output
```

输出示例

```
2023/05/15 18:10:34 Role arn:aws:iam::<aws_account_id>:role/my-role-my-namespace-
aws-creds created
2023/05/15 18:10:34 Saved credentials configuration to: credrequests-ccoctl-
output/manifests/my-namespace-aws-creds-credentials.yaml
2023/05/15 18:10:35 Updated Role policy for Role my-role-my-namespace-aws-creds
```

从输出中复制 **<aws_role_arn>** 以在下一步中使用。例如，**arn:aws:iam::<aws_account_id>:role/my-role-my-namespace-aws-creds**。

e. 运行以下命令，使用角色 ARN 绑定服务帐户：

```
$ oc annotate -n my-namespace sa/aws-provider eks.amazonaws.com/role-arn="
<aws_role_arn>"
```

3. 创建 secret 供应商类以定义您的 secret 存储供应商：

a. 创建定义 **SecretProviderClass** 对象的 YAML 文件：

secret-provider-class-aws.yaml示例

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-aws-provider                    (1)
  namespace: my-namespace                  (2)
spec:
  provider: aws                            (3)
  parameters:                              (4)
    objects: |
      - objectName: "testSecret"
        objectType: "secretsmanager"
```

(1) 指定 secret 供应商类的名称。

(2) 指定 secret 供应商类的命名空间。

(3) 将供应商指定为 **aws**。

(4) 指定特定于供应商的配置参数。

b. 运行以下命令来创建 **SecretProviderClass** 对象：

```
$ oc create -f secret-provider-class-aws.yaml
```

4. 创建部署以使用此 secret 供应商类：

a. 创建定义 **Deployment** 对象的 YAML 文件：

deployment.yaml 示例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-aws-deployment                    (1)
  namespace: my-namespace                    (2)
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-storage
  template:
    metadata:
      labels:
        app: my-storage
    spec:
      containers:
      - name: busybox
        image: k8s.gcr.io/e2e-test-images/busybox:1.29
        command:
          - "/bin/sleep"
          - "10000"
        volumeMounts:
        - name: secrets-store-inline
          mountPath: "/mnt/secrets-store"
          readOnly: true
      volumes:
        - name: secrets-store-inline
          csi:
            driver: secrets-store.csi.k8s.io
            readOnly: true
            volumeAttributes:
              secretProviderClass: "my-aws-provider"   (3)
```

(1) 指定部署的名称。

(2) 指定部署的命名空间。这必须与 secret 供应商类相同。

(3) 指定 secret 供应商类的名称。

b. 运行以下命令来创建 **Deployment** 对象：

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

验证

- 验证您可以从 pod 卷挂载中的 AWS Secrets Manager 访问 secret：

a. 列出 pod 挂载中的 secret：

```
$ oc exec busybox-<hash> -n my-namespace -- ls /mnt/secrets-store/
```

-

输出示例

```
testSecret
```

b. 查看 pod 挂载中的 secret：

```
$ oc exec busybox-<hash> -n my-namespace -- cat /mnt/secrets-store/testSecret
```

输出示例

```
<secret_value>
```

其他资源

- 配置 Cloud Credential Operator 工具

2.7.3.2. 从 AWS Systems Manager Parameter Store 中挂载 secret

您可以使用 Secrets Store CSI Driver Operator 将 secret 从 AWS Systems Manager Parameter Store 挂载到 OpenShift Container Platform 中的 CSI 卷。要从 AWS Systems Manager Parameter Store 挂载 secret，您的集群必须安装在 AWS 上，并使用 AWS 安全令牌服务(STS)。

先决条件

- 您的集群安装在 AWS 上，并使用 AWS 安全令牌服务 (STS)。
- 已安装 Secrets Store CSI Driver Operator。具体步骤请参阅 *安装 Secret Store CSI 驱动程序*。
- 您已配置了 AWS Systems Manager Parameter Store 以存储所需的 secret。
- 您已提取并准备好 **ccoctl** 二进制文件。
- 已安装 **jq** CLI 工具。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 安装 AWS Systems Manager Parameter Store 供应商：

   a. 使用供应商资源的以下配置创建一个 YAML 文件：

重要

Secret Store CSI 驱动程序的 AWS Systems Manager Parameter Store 供应商是一个上游供应商。

此配置会根据上游 AWS 文档中提供的配置进行修改，以便它可以与 OpenShift Container Platform 正常工作。对此配置的更改可能会影响功能。

aws-provider.yaml 文件示例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-secrets-store-provider-aws
  namespace: openshift-cluster-csi-drivers
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csi-secrets-store-provider-aws-cluster-role
rules:
- apiGroups: [""]
  resources: ["serviceaccounts/token"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["serviceaccounts"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: csi-secrets-store-provider-aws-cluster-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: csi-secrets-store-provider-aws-cluster-role
subjects:
- kind: ServiceAccount
  name: csi-secrets-store-provider-aws
  namespace: openshift-cluster-csi-drivers
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: openshift-cluster-csi-drivers
  name: csi-secrets-store-provider-aws
  labels:
    app: csi-secrets-store-provider-aws
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: csi-secrets-store-provider-aws
  template:
    metadata:
      labels:
        app: csi-secrets-store-provider-aws
    spec:
      serviceAccountName: csi-secrets-store-provider-aws
```

```
      hostNetwork: false
      containers:
        - name: provider-aws-installer
          image: public.ecr.aws/aws-secrets-manager/secrets-store-csi-driver-provider-
aws:1.0.r2-50-g5b4aca1-2023.06.09.21.19
          imagePullPolicy: Always
          args:
            - --provider-volume=/etc/kubernetes/secrets-store-csi-providers
          resources:
            requests:
              cpu: 50m
              memory: 100Mi
            limits:
              cpu: 50m
              memory: 100Mi
          securityContext:
            privileged: true
          volumeMounts:
            - mountPath: "/etc/kubernetes/secrets-store-csi-providers"
              name: providervol
            - name: mountpoint-dir
              mountPath: /var/lib/kubelet/pods
              mountPropagation: HostToContainer
      tolerations:
      - operator: Exists
      volumes:
        - name: providervol
          hostPath:
            path: "/etc/kubernetes/secrets-store-csi-providers"
        - name: mountpoint-dir
          hostPath:
            path: /var/lib/kubelet/pods
            type: DirectoryOrCreate
      nodeSelector:
        kubernetes.io/os: linux
```

b. 运行以下命令，授予 **csi-secrets-store-provider-aws** 服务帐户的特权访问权限：

```
$ oc adm policy add-scc-to-user privileged -z csi-secrets-store-provider-aws -n openshift-
cluster-csi-drivers
```

c. 运行以下命令来创建供应商资源：

```
$ oc apply -f aws-provider.yaml
```

2. 授予服务帐户读取 AWS secret 对象的权限：

a. 运行以下命令，创建一个目录使其包含凭证请求：

```
$ mkdir credentialsrequest-dir-aws
```

b. 使用以下配置为凭证请求创建 YAML 文件：

credentialsrequest.yaml 文件示例

```
apiVersion: cloudcredential.openshift.io/v1
kind: CredentialsRequest
metadata:
  name: aws-provider-test
  namespace: openshift-cloud-credential-operator
spec:
  providerSpec:
    apiVersion: cloudcredential.openshift.io/v1
    kind: AWSProviderSpec
    statementEntries:
    - action:
      - "ssm:GetParameter"
      - "ssm:GetParameters"
      effect: Allow
      resource: "arn:*:ssm:*:*:parameter/testParameter*"
  secretRef:
    name: aws-creds
    namespace: my-namespace
  serviceAccountNames:
  - aws-provider
```

c. 运行以下命令来检索 OIDC 供应商：

```
$ oc get --raw=/.well-known/openid-configuration | jq -r '.issuer'
```

输出示例

```
https://<oidc_provider_name>
```

从输出中复制 OIDC 供应商名称 **<oidc_provider_name>**，在下一步中使用。

d. 运行以下命令，使用 **ccoctl** 工具处理凭证请求：

```
$ ccoctl aws create-iam-roles \
    --name my-role --region=<aws_region> \
    --credentials-requests-dir=credentialsrequest-dir-aws \
    --identity-provider-arn arn:aws:iam::<aws_account>:oidc-
provider/<oidc_provider_name> --output-dir=credrequests-ccoctl-output
```

输出示例

```
2023/05/15 18:10:34 Role arn:aws:iam::<aws_account_id>:role/my-role-my-namespace-
aws-creds created
2023/05/15 18:10:34 Saved credentials configuration to: credrequests-ccoctl-
output/manifests/my-namespace-aws-creds-credentials.yaml
2023/05/15 18:10:35 Updated Role policy for Role my-role-my-namespace-aws-creds
```

从输出中复制 **<aws_role_arn>** 以在下一步中使用。例如，**arn:aws:iam::<aws_account_id>:role/my-role-my-namespace-aws-creds**。

e. 运行以下命令，使用角色 ARN 绑定服务帐户：

```
$ oc annotate -n my-namespace sa/aws-provider eks.amazonaws.com/role-arn="
<aws_role_arn>"
```

3. 创建 secret 供应商类以定义您的 secret 存储供应商：

   a. 创建定义 **SecretProviderClass** 对象的 YAML 文件：

   secret-provider-class-aws.yaml示例

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-aws-provider                    (1)
  namespace: my-namespace                  (2)
spec:
  provider: aws                            (3)
  parameters:                              (4)
    objects: |
      - objectName: "testParameter"
        objectType: "ssmparameter"
```

   (1) 指定 secret 供应商类的名称。

   (2) 指定 secret 供应商类的命名空间。

   (3) 将供应商指定为 **aws**。

   (4) 指定特定于供应商的配置参数。

   b. 运行以下命令来创建 **SecretProviderClass** 对象：

```
$ oc create -f secret-provider-class-aws.yaml
```

4. 创建部署以使用此 secret 供应商类：

   a. 创建定义 **Deployment** 对象的 YAML 文件：

   deployment.yaml 示例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-aws-deployment                  (1)
  namespace: my-namespace                  (2)
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-storage
  template:
    metadata:
      labels:
        app: my-storage
    spec:
      containers:
      - name: busybox
```

```
      image: k8s.gcr.io/e2e-test-images/busybox:1.29
      command:
        - "/bin/sleep"
        - "10000"
      volumeMounts:
      - name: secrets-store-inline
        mountPath: "/mnt/secrets-store"
        readOnly: true
    volumes:
      - name: secrets-store-inline
        csi:
          driver: secrets-store.csi.k8s.io
          readOnly: true
          volumeAttributes:
            secretProviderClass: "my-aws-provider" 3
```

1 指定部署的名称。

2 指定部署的命名空间。这必须与 secret 供应商类相同。

3 指定 secret 供应商类的名称。

b. 运行以下命令来创建 **Deployment** 对象：

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

验证

- 验证您可以从 pod 卷挂载中的 AWS Systems Manager Parameter Store 访问 secret：

  a. 列出 pod 挂载中的 secret：

  ```
  $ oc exec busybox-<hash> -n my-namespace -- ls /mnt/secrets-store/
  ```

  输出示例

  ```
  testParameter
  ```

  b. 查看 pod 挂载中的 secret：

  ```
  $ oc exec busybox-<hash> -n my-namespace -- cat /mnt/secrets-store/testSecret
  ```

  输出示例

  ```
  <secret_value>
  ```

其他资源

- 配置 Cloud Credential Operator 工具

2.7.3.3. 从 Azure Key Vault 挂载 secret

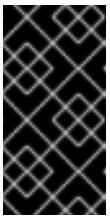
您可以使用 Secrets Store CSI Driver Operator 将 secret 从 Azure Key Vault 挂载到 OpenShift Container Platform 中的 CSI 卷。要从 Azure Key Vault 挂载 secret，您的集群必须安装在 Microsoft Azure 上。

先决条件

- 集群安装在 Azure 上。
- 已安装 Secrets Store CSI Driver Operator。具体步骤请参阅 *安装 Secret Store CSI 驱动程序*。
- 您已将 Azure Key Vault 配置为存储所需的 secret。
- 已安装 Azure CLI (**az**)。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 安装 Azure Key Vault 供应商：
   a. 使用供应商资源的以下配置创建一个 YAML 文件：



重要

Secrets Store CSI 驱动程序的 Azure Key Vault 供应商是一个上游供应商。

此配置会根据上游 Azure 文档中提供的配置进行修改，以便它可以与 OpenShift Container Platform 正常工作。对此配置的更改可能会影响功能。

azure-provider.yaml 文件示例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-secrets-store-provider-azure
  namespace: openshift-cluster-csi-drivers
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csi-secrets-store-provider-azure-cluster-role
rules:
- apiGroups: [""]
  resources: ["serviceaccounts/token"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["serviceaccounts"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: ClusterRoleBinding
metadata:
  name: csi-secrets-store-provider-azure-cluster-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: csi-secrets-store-provider-azure-cluster-role
subjects:
- kind: ServiceAccount
  name: csi-secrets-store-provider-azure
  namespace: openshift-cluster-csi-drivers
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: openshift-cluster-csi-drivers
  name: csi-secrets-store-provider-azure
  labels:
    app: csi-secrets-store-provider-azure
spec:
  updateStrategy:
    type: RollingUpdate
  selector:
    matchLabels:
      app: csi-secrets-store-provider-azure
  template:
    metadata:
      labels:
        app: csi-secrets-store-provider-azure
    spec:
      serviceAccountName: csi-secrets-store-provider-azure
      hostNetwork: true
      containers:
        - name: provider-azure-installer
          image: mcr.microsoft.com/oss/azure/secrets-store/provider-azure:v1.4.1
          imagePullPolicy: IfNotPresent
          args:
            - --endpoint=unix:///provider/azure.sock
            - --construct-pem-chain=true
            - --healthz-port=8989
            - --healthz-path=/healthz
            - --healthz-timeout=5s
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8989
            failureThreshold: 3
            initialDelaySeconds: 5
            timeoutSeconds: 10
            periodSeconds: 30
          resources:
            requests:
              cpu: 50m
              memory: 100Mi
            limits:
              cpu: 50m
```

```
          memory: 100Mi
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        runAsUser: 0
        capabilities:
          drop:
          - ALL
      volumeMounts:
        - mountPath: "/provider"
          name: providervol
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: type
            operator: NotIn
            values:
            - virtual-kubelet
  volumes:
    - name: providervol
      hostPath:
        path: "/var/run/secrets-store-csi-providers"
  tolerations:
  - operator: Exists
  nodeSelector:
    kubernetes.io/os: linux
```

b. 运行以下命令，授予 **csi-secrets-store-provider-azure** 服务帐户的特权访问权限：

```
$ oc adm policy add-scc-to-user privileged -z csi-secrets-store-provider-azure -n
openshift-cluster-csi-drivers
```

c. 运行以下命令来创建供应商资源：

```
$ oc apply -f azure-provider.yaml
```

2. 创建服务主体来访问密钥库：

a. 运行以下命令，将服务主体客户端 secret 设置为环境变量：

```
$ SERVICE_PRINCIPAL_CLIENT_SECRET="$(az ad sp create-for-rbac --name
https://$KEYVAULT_NAME --query 'password' -otsv)"
```

b. 运行以下命令，将服务主体客户端 ID 设置为环境变量：

```
$ SERVICE_PRINCIPAL_CLIENT_ID="$(az ad sp list --display-name
https://$KEYVAULT_NAME --query '[0].appId' -otsv)"
```

c. 运行以下命令，使用服务主体客户端 secret 和 ID 创建通用 secret：

```
$ oc create secret generic secrets-store-creds -n my-namespace --from-literal
clientid=${SERVICE_PRINCIPAL_CLIENT_ID} --from-literal
clientsecret=${SERVICE_PRINCIPAL_CLIENT_SECRET}
```

d. 应用 **secrets-store.csi.k8s.io/used=true** 标签，以允许供应商查找此 **nodePublishSecretRef** secret：

```
$ oc -n my-namespace label secret secrets-store-creds secrets-
store.csi.k8s.io/used=true
```

3. 创建 secret 供应商类以定义您的 secret 存储供应商：

a. 创建定义 **SecretProviderClass** 对象的 YAML 文件：

secret-provider-class-azure.yaml 示例

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-azure-provider                 1
  namespace: my-namespace                 2
spec:
  provider: azure                         3
  parameters:                             4
    usePodIdentity: "false"
    useVMManagedIdentity: "false"
    userAssignedIdentityID: ""
    keyvaultName: "kvname"
    objects: |
      array:
        - |
          objectName: secret1
          objectType: secret
    tenantId: "tid"
```

1. 指定 secret 供应商类的名称。
2. 指定 secret 供应商类的命名空间。
3. 将供应商指定为 **azure**。
4. 指定特定于供应商的配置参数。

b. 运行以下命令来创建 **SecretProviderClass** 对象：

```
$ oc create -f secret-provider-class-azure.yaml
```

4. 创建部署以使用此 secret 供应商类：

a. 创建定义 **Deployment** 对象的 YAML 文件：

deployment.yaml 示例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-azure-deployment                    1
  namespace: my-namespace                      2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-storage
  template:
    metadata:
      labels:
        app: my-storage
    spec:
      containers:
      - name: busybox
        image: k8s.gcr.io/e2e-test-images/busybox:1.29
        command:
          - "/bin/sleep"
          - "10000"
        volumeMounts:
        - name: secrets-store-inline
          mountPath: "/mnt/secrets-store"
          readOnly: true
      volumes:
        - name: secrets-store-inline
          csi:
            driver: secrets-store.csi.k8s.io
            readOnly: true
            volumeAttributes:
              secretProviderClass: "my-azure-provider"   3
            nodePublishSecretRef:
              name: secrets-store-creds                  4
```

1. 指定部署的名称。
2. 指定部署的命名空间。这必须与 secret 供应商类相同。
3. 指定 secret 供应商类的名称。
4. 指定包含用于访问 Azure Key Vault 的服务主体凭证的 Kubernetes secret 名称。

b. 运行以下命令来创建 **Deployment** 对象：

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

验证

- 验证您可以从 pod 卷挂载中的 Azure Key Vault 访问 secret：

  a. 列出 pod 挂载中的 secret：

```
$ oc exec busybox-<hash> -n my-namespace -- ls /mnt/secrets-store/
```

-

输出示例

```
secret1
```

b. 查看 pod 挂载中的 secret：

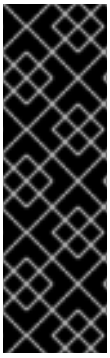
```
$ oc exec busybox-<hash> -n my-namespace -- cat /mnt/secrets-store/secret1
```

输出示例

```
my-secret-value
```

2.7.3.4. 从 HashiCorp Vault 挂载 secret

您可以使用 Secrets Store CSI Driver Operator 将 secret 从 HashiCorp Vault 挂载到 OpenShift Container Platform 中的 CSI 卷。



重要

使用 Secrets Store CSI Driver Operator 从 HashiCorp Vault 挂载 secret 已使用以下云供应商测试：

- Amazon Web Services (AWS)
- Microsoft Azure

其他云供应商可能可以正常工作，但还没有测试。以后可能会测试其他云供应商。

先决条件

- 已安装 Secrets Store CSI Driver Operator。具体步骤请参阅 *安装 Secret Store CSI 驱动程序*。
- 您已安装 Helm。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 运行以下命令来添加 HashiCorp Helm 仓库：

```
$ helm repo add hashicorp https://helm.releases.hashicorp.com
```

2. 运行以下命令，更新所有仓库以确保 Helm 了解最新版本：

```
$ helm repo update
```

3. 安装 HashiCorp Vault 供应商：

a. 运行以下命令，为 Vault 创建一个新项目：

```
$ oc new-project vault
```

b. 运行以下命令，为 pod 安全准入标记 **vault** 命名空间：

```
$ oc label ns vault security.openshift.io/scc.podSecurityLabelSync=false pod-
security.kubernetes.io/enforce=privileged pod-security.kubernetes.io/audit=privileged
pod-security.kubernetes.io/warn=privileged --overwrite
```

c. 运行以下命令，授予 **vault** 服务帐户的特权访问权限：

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

d. 运行以下命令，授予 **vault-csi-provider** 服务帐户的访问权限：

```
$ oc adm policy add-scc-to-user privileged -z vault-csi-provider -n vault
```

e. 运行以下命令来部署 HashiCorp Vault：

```
$ helm install vault hashicorp/vault --namespace=vault \
  --set "server.dev.enabled=true" \
  --set "injector.enabled=false" \
  --set "csi.enabled=true" \
  --set "global.openshift=true" \
  --set "injector.agentImage.repository=docker.io/hashicorp/vault" \
  --set "server.image.repository=docker.io/hashicorp/vault" \
  --set "csi.image.repository=docker.io/hashicorp/vault-csi-provider" \
  --set "csi.agent.image.repository=docker.io/hashicorp/vault" \
  --set "csi.daemonSet.providersDir=/var/run/secrets-store-csi-providers"
```

f. 运行以下命令，修补 **vault-csi-driver** 守护进程，将 **securityContext** 设置为 **privileged** ：

```
$ oc patch daemonset -n vault vault-csi-provider --type='json' -p='[{"op": "add", "path":
"/spec/template/spec/containers/0/securityContext", "value": {"privileged": true} }]'
```

g. 运行以下命令，验证 **vault-csi-provider** pod 是否已正确启动：

```
$ oc get pods -n vault
```

输出示例

```
NAME                         READY   STATUS    RESTARTS   AGE
vault-0                      1/1     Running   0          24m
vault-csi-provider-87rgw     1/2     Running   0          5s
vault-csi-provider-bd6hp     1/2     Running   0          4s
vault-csi-provider-smlv7     1/2     Running   0          5s
```

4. 配置 HashiCorp Vault 以存储所需的 secret：

a. 运行以下命令来创建 secret：

```
$ oc exec vault-0 --namespace=vault -- vault kv put secret/example1 testSecret1=my-
secret-value
```

b. 运行以下命令，验证 **secret/example1** 的路径是否可读：

```
$ oc exec vault-0 --namespace=vault -- vault kv get secret/example1
```

输出示例

```
= Secret Path =
secret/data/example1

======= Metadata =======
Key              Value
---              -----
created_time     2024-04-05T07:05:16.713911211Z
custom_metadata  <nil>
deletion_time    n/a
destroyed        false
version          1

=== Data ===
Key           Value
---           -----
testSecret1   my-secret-value
```

5. 将 Vault 配置为使用 Kubernetes 身份验证：

   a. 运行以下命令来启用 Kubernetes auth 方法：

   ```
   $ oc exec vault-0 --namespace=vault -- vault auth enable kubernetes
   ```

   输出示例

   ```
   Success! Enabled kubernetes auth method at: kubernetes/
   ```

   b. 配置 Kubernetes auth 方法：

      i. 运行以下命令，将令牌查看器设置为环境变量：

      ```
      $ TOKEN_REVIEWER_JWT="$(oc exec vault-0 --namespace=vault -- cat
      /var/run/secrets/kubernetes.io/serviceaccount/token)"
      ```

      ii. 运行以下命令，将 Kubernetes 服务 IP 地址设置为环境变量：

      ```
      $ KUBERNETES_SERVICE_IP="$(oc get svc kubernetes -o go-template="{{
      .spec.clusterIP }}")"
      ```

      iii. 运行以下命令来更新 Kubernetes auth 方法：

      ```
      $ oc exec -i vault-0 --namespace=vault -- vault write auth/kubernetes/config \
        issuer="https://kubernetes.default.svc.cluster.local" \
        token_reviewer_jwt="${TOKEN_REVIEWER_JWT}" \
        kubernetes_host="https://${KUBERNETES_SERVICE_IP}:443" \
        kubernetes_ca_cert=@/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      ```

      输出示例

```
Success! Data written to: auth/kubernetes/config
```

c. 运行以下命令，为应用程序创建一个策略：

```
$ oc exec -i vault-0 --namespace=vault -- vault policy write csi -<<EOF
  path "secret/data/*" {
  capabilities = ["read"]
  }
  EOF
```

输出示例

```
Success! Uploaded policy: csi
```

d. 运行以下命令，创建一个身份验证角色来访问应用程序：

```
$ oc exec -i vault-0 --namespace=vault -- vault write auth/kubernetes/role/csi \
  bound_service_account_names=default \
  bound_service_account_namespaces=default,test-ns,negative-test-ns,my-namespace \
  policies=csi \
  ttl=20m
```

输出示例

```
Success! Data written to: auth/kubernetes/role/csi
```

e. 运行以下命令，验证所有 **vault** pod 是否都正常运行：

```
$ oc get pods -n vault
```

输出示例

```
NAME                    READY   STATUS    RESTARTS   AGE
vault-0                 1/1     Running   0          43m
vault-csi-provider-87rgw   2/2     Running   0          19m
vault-csi-provider-bd6hp   2/2     Running   0          19m
vault-csi-provider-smlv7   2/2     Running   0          19m
```

f. 运行以下命令，验证所有 **secrets-store-csi-driver** pod 是否正常运行：

```
$ oc get pods -n openshift-cluster-csi-drivers | grep -E "secrets"
```

输出示例

```
secrets-store-csi-driver-node-46d2g                3/3     Running   0          45m
secrets-store-csi-driver-node-d2jjn                3/3     Running   0          45m
secrets-store-csi-driver-node-drmt4                3/3     Running   0          45m
secrets-store-csi-driver-node-j2wlt                3/3     Running   0          45m
secrets-store-csi-driver-node-v9xv4                3/3     Running   0          45m
secrets-store-csi-driver-node-vlz28                3/3     Running   0          45m
secrets-store-csi-driver-operator-84bd699478-fpxrw   1/1     Running   0          47m
```

6. 创建 secret 供应商类以定义您的 secret 存储供应商：

   a. 创建定义 **SecretProviderClass** 对象的 YAML 文件：

   secret-provider-class-vault.yaml 示例

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-vault-provider                    (1)
  namespace: my-namespace                    (2)
spec:
  provider: vault                            (3)
  parameters:                                (4)
    roleName: "csi"
    vaultAddress: "http://vault.vault:8200"
    objects:  |
      - secretPath: "secret/data/example1"
        objectName: "testSecret1"
        secretKey: "testSecret1
```

   (1) 指定 secret 供应商类的名称。

   (2) 指定 secret 供应商类的命名空间。

   (3) 将提供程序指定为 **vault**。

   (4) 指定特定于供应商的配置参数。

   b. 运行以下命令来创建 **SecretProviderClass** 对象：

```
$ oc create -f secret-provider-class-vault.yaml
```

7. 创建部署以使用此 secret 供应商类：

   a. 创建定义 **Deployment** 对象的 YAML 文件：

   deployment.yaml 示例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: busybox-deployment                   (1)
  namespace: my-namespace                    (2)
  labels:
    app: busybox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: busybox
  template:
    metadata:
      labels:
```

```
      app: busybox
  spec:
    terminationGracePeriodSeconds: 0
    containers:
    - image: registry.k8s.io/e2e-test-images/busybox:1.29-4
      name: busybox
      imagePullPolicy: IfNotPresent
      command:
      - "/bin/sleep"
      - "10000"
      volumeMounts:
      - name: secrets-store-inline
        mountPath: "/mnt/secrets-store"
        readOnly: true
    volumes:
      - name: secrets-store-inline
        csi:
          driver: secrets-store.csi.k8s.io
          readOnly: true
          volumeAttributes:
            secretProviderClass: "my-vault-provider"          3
```

1. 指定部署的名称。
2. 指定部署的命名空间。这必须与 secret 供应商类相同。
3. 指定 secret 供应商类的名称。

b. 运行以下命令来创建 **Deployment** 对象：

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

验证

- 验证您是否可以从 pod 卷挂载中的 HashiCorp Vault 访问 secret：

a. 运行以下命令，列出 pod 挂载中的 secret：

```
$ oc exec busybox-<hash> -n my-namespace -- ls /mnt/secrets-store/
```

输出示例

```
testSecret1
```

b. 运行以下命令，查看 pod 挂载中的 secret：

```
$ oc exec busybox-<hash> -n my-namespace -- cat /mnt/secrets-store/testSecret1
```

输出示例

```
my-secret-value
```

2.7.4. 启用对作为 Kubernetes secret 挂载的内容进行同步

您可以启用同步，从挂载的卷中的内容创建 Kubernetes secret。您可能要启用同步的示例是使用部署中的环境变量来引用 Kubernetes secret。

警告

如果您不想将 secret 存储在 OpenShift Container Platform 集群和 etcd 中，请不要启用同步。仅在需要它时启用此功能，比如当您想要使用环境变量来引用 secret 时。

如果启用了同步，在启动挂载 secret 的 pod 后，来自挂载卷的 secret 会同步为 Kubernetes secret。

当所有挂载内容的 pod 被删除时，同步的 Kubernetes secret 会被删除。

先决条件

- 已安装 Secrets Store CSI Driver Operator。
- 已安装 secret 存储供应商。
- 您已创建了 secret 供应商类。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 运行以下命令来编辑 **SecretProviderClass** 资源：

```
$ oc edit secretproviderclass my-azure-provider 1
```

1 将 **my-azure-provider** 替换为 secret 供应商类的名称。

2. 使用同步的 Kubernetes secret 配置添加 **secretsObjects** 部分：

```
apiVersion: secrets-store.csi.x-k8s.io/v1
kind: SecretProviderClass
metadata:
  name: my-azure-provider
  namespace: my-namespace
spec:
  provider: azure
  secretObjects:                       1
    - secretName: tlssecret            2
      type: kubernetes.io/tls          3
      labels:
        environment: "test"
      data:
        - objectName: tlskey           4
          key: tls.key                 5
```

```
      - objectName: tlscrt
        key: tls.crt
  parameters:
    usePodIdentity: "false"
    keyvaultName: "kvname"
    objects:  |
      array:
        - |
          objectName: tlskey
          objectType: secret
        - |
          objectName: tlscrt
          objectType: secret
    tenantId: "tid"
```

1. 指定同步的 Kubernetes secret 的配置。
2. 指定要创建的 Kubernetes **Secret** 对象的名称。
3. 指定要创建的 Kubernetes **Secret** 对象的类型。例如：**Opaque** 或 **kubernetes.io/tls**。
4. 指定要同步内容的对象名称或别名。
5. 指定指定 **objectName** 中的 data 字段来填充 Kubernetes secret。

3. 保存文件以使改变生效。

2.7.5. 查看 pod 卷挂载中的 secret 状态

您可以查看 pod 卷挂载中 secret 的详细信息，包括版本。

Secrets Store CSI Driver Operator 在与 pod 相同的命名空间中创建一个 **SecretProviderClassPodStatus** 资源。您可以查看此资源来查看详细信息，包括版本，以及 pod 卷挂载中的 secret。

先决条件

- 已安装 Secrets Store CSI Driver Operator。
- 已安装 secret 存储供应商。
- 您已创建了 secret 供应商类。
- 您已部署了从 Secrets Store CSI Driver Operator 挂载卷的 pod。
- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

- 运行以下命令，查看 pod 卷挂载中 secret 的详细信息：

```
$ oc get secretproviderclasspodstatus <secret_provider_class_pod_status_name> -o yaml
```
1

1 secret 供应商类 pod 状态对象的名称采用 **<pod_name>-<namespace>-<secret_provider_class_name>** 的格式。

输出示例

```
...
status:
  mounted: true
  objects:
  - id: secret/tlscrt
    version: f352293b97da4fa18d96a9528534cb33
  - id: secret/tlskey
    version: 02534bc3d5df481cb138f8b2a13951ef
  podName: busybox-<hash>
  secretProviderClassName: my-azure-provider
  targetPath: /var/lib/kubelet/pods/f0d49c1e-c87a-4beb-888f-
37798456a3e7/volumes/kubernetes.io~csi/secrets-store-inline/mount
```

2.7.6. 卸载 Secret Store CSI Driver Operator

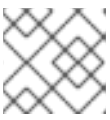
先决条件

- 访问 OpenShift Container Platform Web 控制台。
- 集群的管理员访问权限。

流程

卸载 Secret Store CSI Driver Operator：

1. 停止所有使用 **secrets-store.csi.k8s.io** 供应商的应用程序 pod。
2. 为所选 secret 存储删除任何第三方供应商插件。
3. 删除 Container Storage Interface (CSI) 驱动程序和相关清单：
   a. 点 **Administration → CustomResourceDefinitions → ClusterCSIDriver**。
   b. 在 **Instances** 选项卡上，对于 **secrets-store.csi.k8s.io**，点左侧的下拉菜单，然后点 **Delete ClusterCSIDriver**。
   c. 出现提示时，单击 **Delete**。
4. 验证 CSI 驱动程序 pod 是否不再运行。
5. 卸载 Secret Store CSI Driver Operator：



注意

在卸载 Operator 前，必须先删除 CSI 驱动程序。

   a. 点 **Operators → Installed Operators**。

b. 在 **Installed Operators** 页面中，在 **Search by name** 框中输入 "Secrets Store CSI" 来查找 Operator，然后点击它。

c. 在 **Installed Operators** > **Operator 详情页面** 的右上角，点 **Actions** → **Uninstall Operator**。

d. 当在 **Uninstall Operator** 窗口中提示时，点 **Uninstall** 按钮从命名空间中删除 Operator。Operator 在集群中部署的任何应用程序都需要手动清理。
卸载后，Secret Store CSI Driver Operator 不再列在 web 控制台的 **Installed Operators** 部分。

2.8. 创建和使用配置映射

以下部分定义配置映射以及如何创建和使用它们。

2.8.1. 了解配置映射

许多应用程序需要使用配置文件、命令行参数和环境变量的某些组合来进行配置。在 OpenShift Container Platform 中，这些配置工件与镜像内容分离，以便使容器化应用程序可以移植。

ConfigMap 对象提供了将容器注入到配置数据的机制，同时保持容器与 OpenShift Container Platform 无关。配置映射可用于存储细粒度信息（如个别属性）或粗粒度信息（如完整配置文件或 JSON blob）。

ConfigMap 对象包含配置数据的键值对，这些数据可在 Pod 中消耗或用于存储控制器等系统组件的配置数据。例如：

ConfigMap 对象定义

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: my-namespace
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
binaryData:
  bar: L3Jvb3QvMTAw 2
```

1 包含配置数据。

2 指向含有非 UTF8 数据的文件，如二进制 Java 密钥存储文件。以 Base64 格式输入文件数据。

注意

从二进制文件（如镜像）创建配置映射时，您可以使用 **binaryData** 字段。

可以在 Pod 中以各种方式消耗配置数据。配置映射可用于：

- 在容器中填充环境变量值
- 设置容器中的命令行参数
- 填充卷中的配置文件

用户和系统组件可以在配置映射中存储配置数据。

配置映射与 secret 类似，但设计为能更加便捷地支持与不含敏感信息的字符串配合。

配置映射限制
在 pod 中可以消耗它的内容前，必须创建配置映射。

可以编写控制器来容许缺少的配置数据。根据具体情况使用配置映射来参考各个组件。

ConfigMap 对象驻留在一个项目中。

它们只能被同一项目中的 pod 引用。

Kubelet 只支持为它从 API 服务器获取的 pod 使用配置映射。

这包括使用 CLI 创建或间接从复制控制器创建的 pod。它不包括通过 OpenShift Container Platform 节点的 **--manifest-url** 标记、**--config** 标记，或通过 REST API 创建的 pod，因为这些不是创建 pod 的通用方法。

2.8.2. 在 OpenShift Container Platform Web 控制台中创建配置映射

您可以在 OpenShift Container Platform Web 控制台中创建配置映射。

流程

- 以集群管理员身份创建配置映射：
  1. 在 Administrator 视角中，选择 **Workloads → Config Maps**。
  2. 在该页面右上方选择 **Create Config Map**。
  3. 输入配置映射的内容。
  4. 选择 **Create**。
- 以开发者身份创建配置映射：
  1. 在 Developer 视角中，选择 **Config Maps**。
  2. 在该页面右上方选择 **Create Config Map**。
  3. 输入配置映射的内容。
  4. 选择 **Create**。

2.8.3. 使用 CLI 创建配置映射

您可以使用以下命令从目录、特定文件或文字值创建配置映射。

流程

- 创建配置映射：

```
$ oc create configmap <configmap_name> [options]
```

2.8.3.1. 从目录创建配置映射

您可以使用 **--from-file** 标志从目录创建配置映射。这个方法允许您使用目录中的多个文件来创建配置映射。

目录中的每个文件用于在配置映射中填充键，其中键的名称是文件名，键的值是文件的内容。

例如，以下命令会创建一个带有 **example-files** 目录内容的配置映射：

```
$ oc create configmap game-config --from-file=example-files/
```

查看配置映射中的密钥：

```
$ oc describe configmaps game-config
```

输出示例

```
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data

game.properties:        158 bytes
ui.properties:          83 bytes
```

您可以看到，映射中的两个键都是从命令中指定的目录中的文件名创建的。这些密钥的内容可能非常大，因此 **oc describe** 的输出只显示键的名称及其大小。

前提条件

- 您必须有一个目录，其中包含您要使用填充配置映射的数据的文件。
  以下流程使用这些示例文件：**game.properties** 和 **ui.properties** ：

  ```
  $ cat example-files/game.properties
  ```

  输出示例

  ```
  enemies=aliens
  lives=3
  enemies.cheat=true
  enemies.cheat.level=noGoodRotten
  secret.code.passphrase=UUDDLRLRBABAS
  secret.code.allowed=true
  secret.code.lives=30
  ```

  ```
  $ cat example-files/ui.properties
  ```

输出示例

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

流程

- 输入以下命令，创建包含此目录中每个文件内容的配置映射：

```
$ oc create configmap game-config \
    --from-file=example-files/
```

验证

- 使用带有 **-o** 选项的 **oc get** 命令以查看键的值：

```
$ oc get configmaps game-config -o yaml
```

输出示例

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

2.8.3.2. 从文件创建配置映射

您可以使用 **--from-file** 标志从文件创建配置映射。您可以多次将 **--from-file** 选项传递给 CLI。

您还可以通过将 **key=value** 表达式传递给 **--from-file** 选项，在配置映射中为从文件中导入的内容指定要设置的键。例如：

```
$ oc create configmap game-config-3 --from-file=game-special-key=example-files/game.properties
```

注意

如果从文件创建一个配置映射，您可以在不会破坏非 UTF8 数据的项中包含非 UTF8 的数据。OpenShift Container Platform 检测到二进制文件，并将该文件编码为 **MIME**。在服务器上，**MIME** 有效负载被解码并存储而不会损坏数据。

前提条件

- 您必须有一个目录，其中包含您要使用填充配置映射的数据的文件。以下流程使用这些示例文件：**game.properties** 和 **ui.properties** ：

```
$ cat example-files/game.properties
```

输出示例

```
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

```
$ cat example-files/ui.properties
```

输出示例

```
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

流程

- 通过指定特定文件来创建配置映射：

```
$ oc create configmap game-config-2 \
    --from-file=example-files/game.properties \
    --from-file=example-files/ui.properties
```

- 通过指定键值对来创建配置映射：

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

验证

- 使用 **-o** 选项为对象输入 **oc get** 命令，以查看文件中的键值：

```
$ oc get configmaps game-config-2 -o yaml
```

输出示例

```
apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

- 使用 **-o** 选项为对象输入 **oc get** 命令，以查看键值对中的键值：

```
$ oc get configmaps game-config-3 -o yaml
```

输出示例

```
apiVersion: v1
data:
  game-special-key: |- (1)
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

1 这是您在前面的步骤中设置的密钥。

2.8.3.3. 从字面值创建配置映射

您可以为配置映射提供字面值。

--from-literal 选项采用 **key=value** 语法，它允许直接在命令行中提供字面值。

流程

- 通过指定字面值来创建配置映射：

```
$ oc create configmap special-config \
    --from-literal=special.how=very \
    --from-literal=special.type=charm
```

验证

- 使用带有 **-o** 选项的 **oc get** 命令以查看键的值：

```
$ oc get configmaps special-config -o yaml
```

输出示例

```
apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

2.8.4. 用例： 在 pod 中使用配置映射

以下小节描述了在 pod 中消耗 **ConfigMap** 对象时的一些用例。

2.8.4.1. 使用配置映射在容器中填充环境变量

您可以使用配置映射在容器中填充各个环境变量，或从构成有效环境变量名称的所有键填充容器中的环境变量。

例如，请考虑以下配置映射：

有两个环境变量的 ConfigMap

```
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: special-config 1
  namespace: default 2
data:
  special.how: very 3
  special.type: charm 4
```

1 配置映射的名称。

2 配置映射所在的项目。配置映射只能由同一项目中的 pod 引用。

3 4 要注入的环境变量。

带有一个环境变量的ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config 1
  namespace: default
data:
  log_level: INFO 2
```

1 配置映射的名称。

2 要注入的环境变量。

流程

- 您可以使用 **configMapKeyRef** 部分在 pod 中使用此 **ConfigMap** 的键。

  配置为注入特定环境变量的 Pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: 1
        - name: SPECIAL_LEVEL_KEY 2
          valueFrom:
            configMapKeyRef:
              name: special-config 3
              key: special.how 4
```

```
      - name: SPECIAL_TYPE_KEY
        valueFrom:
          configMapKeyRef:
            name: special-config 5
            key: special.type 6
            optional: true 7
    envFrom: 8
      - configMapRef:
          name: env-config 9
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  restartPolicy: Never
```

1. 从 **ConfigMap** 中拉取指定的环境变量的小节。
2. 要将键值注入到的 pod 环境变量的名称。
3. 5 要从中拉取特定环境变量的 **ConfigMap** 名称。
4. 6 要从 **ConfigMap** 中拉取的环境变量。
7. 使环境变量成为可选。作为可选项，即使指定的 **ConfigMap** 和键不存在，也会启动 pod。
8. 从 **ConfigMap** 中拉取所有环境变量的小节。
9. 要从中拉取所有环境变量的 **ConfigMap** 名称。

当此 pod 运行时，pod 日志包括以下输出：

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

注意

示例输出中没有列出 **SPECIAL_TYPE_KEY=charm**，因为设置了 **optional: true**。

2.8.4.2. 使用配置映射为容器命令设置命令行参数

您可以通过 Kubernetes 替换语法 **$(VAR_NAME)**，使用配置映射来设置容器中的命令或参数的值。

例如，请考虑以下配置映射：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

流程

- 要将值注入到容器中的一个命令中，使用您要用作环境变量的键。然后，您可以使用 **$(VAR_NAME)** 语法在容器的命令中引用它们。

  配置为注入特定环境变量的 pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
1
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  restartPolicy: Never
```

1 使用您要用作环境变量的键将值注入到容器中的命令中。

当此 pod 运行时，test-container 容器中运行的 echo 命令的输出如下：

```
very charm
```

2.8.4.3. 使用配置映射将内容注入卷

您可以使用配置映射将内容注入卷。

ConfigMap 自定义资源(CR)示例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
```

```
  namespace: default
data:
  special.how: very
  special.type: charm
```

流程

您可以使用配置映射将内容注入卷中有两个不同的选项。

- 使用配置映射将内容注入卷的最基本方法是在卷中填充键为文件名称的文件，文件的内容是键值：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  volumes:
    - name: config-volume
      configMap:
        name: special-config 1
  restartPolicy: Never
```

1 包含密钥的文件。

当这个 pod 运行时，cat 命令的输出将是：

```
very
```

- 您还可以控制投射配置映射键的卷中的路径：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
```

```
    type: RuntimeDefault
containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "cat", "/etc/config/path/to/special-key" ]
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
volumes:
  - name: config-volume
    configMap:
      name: special-config
      items:
      - key: special.how
        path: path/to/special-key (1)
restartPolicy: Never
```

(1) 配置映射键的路径。

当这个 pod 运行时，cat 命令的输出将是：

```
very
```

2.9. 使用设备插件来利用 POD 访问外部资源

借助设备插件，您无需编写自定义代码，就能在 OpenShift Container Platform pod 中使用特定的设备类型，如 GPU、InfiniBand 或其他需要供应商专用初始化和设置的类似计算资源。

2.9.1. 了解设备插件

设备插件提供一致并可移植的解决方案，以便跨集群消耗硬件设备。设备插件通过一种扩展机制为这些设备提供支持，从而使这些设备可供容器使用，提供这些设备的健康检查，并安全地共享它们。

重要

OpenShift Container Platform 支持设备插件 API，但设备插件容器由各个供应商提供支持。

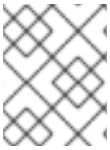
设备插件是在节点（**kubelet** 的外部）上运行的 gRPC 服务，负责管理特定的硬件资源。任何设备插件都必须支持以下远程过程调用 (RPC)：

```
service DevicePlugin {
    // GetDevicePluginOptions returns options to be communicated with Device
    // Manager
    rpc GetDevicePluginOptions(Empty) returns (DevicePluginOptions) {}

    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device disappears, ListAndWatch
    // returns the new list
```

```
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

    // Allocate is called during container creation so that the Device
    // Plug-in can run device specific operations and instruct Kubelet
    // of the steps to make the Device available in the container
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}

    // PreStartcontainer is called, if indicated by Device Plug-in during
    // registration phase, before each container start. Device plug-in
    // can run device specific operations such as resetting the device
    // before making devices available to the container
    rpc PreStartcontainer(PreStartcontainerRequest) returns (PreStartcontainerResponse) {}
}
```

设备插件示例

- 适用于 COS 型操作系统的 Nvidia GPU 设备插件
- Nvidia 官方 GPU 设备插件
- Solarflare 设备插件
- KubeVirt 设备插件：vfio 和 kvm
- 用于 IBM® Crypto Express (CEX) 卡的 Kubernetes 设备插件



注意

对于简单设备插件参考实现，设备管理器代码中有一个 stub 设备插件：*vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go*。

2.9.1.1. 设备插件部署方法

- 守护进程集是设备插件部署的推荐方法。
- 在启动时，设备插件会尝试在节点上 */var/lib/kubelet/device-plugin/* 创建一个 UNIX 域套接字，以便服务来自于设备管理器的 RPC。
- 由于设备插件必须管理硬件资源、主机文件系统的访问权以及套接字创建，它们必须在一个特权安全上下文中运行。
- 各种设备插件实现中提供了有关部署步骤的更多细节。

2.9.2. 了解设备管理器

设备管理器提供了一种机制，可借助称为“设备插件”的插件公告专用节点硬件资源。

您可以公告专用的硬件，而不必修改任何上游代码。

重要

OpenShift Container Platform 支持设备插件 API，但设备插件容器由各个供应商提供支持。

设备管理器将设备公告为**外部资源**。用户 pod 可以利用相同的**限制/请求**机制来使用设备管理器公告的设备，这一机制也用于请求任何其他**扩展资源**。

在启动时，设备插件会在 */var/lib/kubelet/device-plugins/kubelet.sock* 上调用 **Register** 将自身注册到设备管理器，并启动位于 */ var/lib/kubelet/device-plugins/<plugin>.sock* 的 gRPC 服务，以服务设备管理器请求。

在处理新的注册请求时，设备管理器会在设备插件服务中调用 **ListAndWatch** 远程过程调用 (RPC)。作为响应，设备管理器通过 gRPC 流从插件中获取设备对象的列表。设备管理器对流进行持续监控，以确认插件有没有新的更新。在插件一端，插件也会使流保持开放；只要任何设备的状态有所改变，就会通过相同的流传输连接将新设备列表发送到设备管理器。

在处理新的 pod 准入请求时，Kubelet 将请求的**扩展资源**传递给设备管理器以进行设备分配。设备管理器在其数据库中检查，以验证是否存在对应的插件。如果插件存在并且有可分配的设备及本地缓存，则在该特定设备插件上调用 **Allocate** RPC。

此外，设备插件也可以执行其他几个特定于设备的操作，如驱动程序安装、设备初始化和设备重置。这些功能视具体实现而异。

2.9.3. 启用设备管理器

启用设备管理器来实现设备插件，在不更改上游代码的前提下公告专用硬件。

设备管理器提供了一种机制，可借助称为“设备插件”的插件公告专用节点硬件资源。

1. 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** CRD 关联的标签。执行以下步骤之一：

   a. 查看机器配置：

   ```
   # oc describe machineconfig <name>
   ```

   例如：

   ```
   # oc describe machineconfig 00-worker
   ```

   输出示例

   ```
   Name:         00-worker
   Namespace:
   Labels:       machineconfiguration.openshift.io/role=worker 1
   ```

   1 设备管理器所需标签。

流程

1. 为配置更改创建自定义资源 (CR)。

   设备管理器 CR 配置示例

   ```
   apiVersion: machineconfiguration.openshift.io/v1
   kind: KubeletConfig
   metadata:
   ```

```
  name: devicemgr (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      machineconfiguration.openshift.io: devicemgr (2)
  kubeletConfig:
    feature-gates:
      - DevicePlugins=true (3)
```

1. 为 CR 分配一个名称。
2. 输入来自机器配置池的标签。
3. 将 **DevicePlugins** 设为“true”。

2. 创建设备管理器：

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

输出示例

```
kubeletconfig.machineconfiguration.openshift.io/devicemgr created
```

3. 通过确认节点上已创建了 */var/lib/kubelet/device-plugins/kubelet.sock*，确保已启用了设备管理器。这是设备管理器 gRPC 服务器在其上侦听新插件注册的 UNIX 域套接字。只有启用了设备管理器，才会在 Kubelet 启动时创建此 sock 文件。

2.10. 在 POD 调度决策中纳入 POD 优先级

您可以在集群中启用 pod 优先级与抢占功能。pod 优先级代表与其他 pod 相比此 pod 的重要性，并根据优先级进行队列处理。抢占（preemption）则允许集群驱除低优先级 pod 或与之争抢，从而在合适的节点上没有可用空间时能够调度优先级较高的 pod。pod 优先级也会影响 pod 的调度顺序以及节点上资源不足驱除顺序。

要使用优先级和抢占功能，您需要创建优先级类来定义 pod 的相对权重。然后，在 pod 规格中引用优先级类，以应用这个权重来进行调度。

2.10.1. 了解 pod 优先级

当您使用 pod 优先级与抢占功能时，调度程序会根据优先级来调度待处理 pod，而待处理 pod 会放在调度队列中优先级较低的其他待处理 pod 的前面。因此，如果达到调度要求，较高优先级的 pod 可能比低优先级的 pod 更早调度。如果 pod 无法调度，调度程序会继续调度其他较低优先级 pod。

2.10.1.1. Pod 优先级类

您可以为 pod 分配一个优先级类，它是一种非命名空间的对象，用于定义从名称到优先级整数值的映射。数值越大，优先级越高。

优先级类对象可以取小于或等于 1000000000（十亿）的 32 位整数值。对于不得被抢占或被驱除的关键 pod，请保留大于或等于 10 亿的数值。默认情况下，OpenShift Container Platform 有两个保留优先级类，用于需要保证调度的关键系统 pod。

```
$ oc get priorityclasses
```

输出示例

```
NAME                     VALUE        GLOBAL-DEFAULT   AGE
system-node-critical     2000001000   false            72m
system-cluster-critical  2000000000   false            72m
openshift-user-critical  1000000000   false            3d13h
cluster-logging          1000000      false            29s
```

- **system-node-critical** - 此优先级类的值为 2000001000，用于所有不得从节点上驱除的 pod。具有此优先级类的 pod 示例有 **sdn-ovs** 和 **sdn** 等。许多关键组件默认包括 **system-node-critical** 优先级类，例如：
  - master-api
  - master-controller
  - master-etcd
  - sdn
  - sdn-ovs
  - sync
- **system-cluster-critical** - 此优先级类的值是 2000000000（二十亿），用于对集群而言很重要的 pod。在某些情况下，具有此优先级类的 Pod 可以从节点中驱除。例如，配置了 **system-node-critical** 优先级类的 pod 可以拥有优先权。不过，此优先级类确实能够保证调度。具有此优先级类的 pod 示例有 fluentd 以及 descheduler 这样的附加组件等。许多关键组件默认包括 **system-cluster-critical** 优先级类，例如：
  - fluentd
  - metrics-server
  - descheduler
- **openshift-user-critical** - 您可以使用带有重要 pod 的 **priorityClassName** 字段，这些 pod 无法绑定其资源消耗，且没有可预测的资源消耗行为。**openshift-monitoring** 和 **openshift-user-workload-monitoring** 命名空间下的 Prometheus Pod 使用 **openshift-user-critical priorityClassName**。监控工作负载使用 **system-critical** 作为其第一个 **priorityClass**，但在监控使用过量内存时造成问题，且无法驱除它们。因此，监控会丢弃优先级，为调度程序带来灵活性，并围绕移动繁重的工作负载来保持关键节点正常操作。
- **cluster-logging** - 此优先级类供 Fluentd 用于确保 Fluentd pod 优先于其他应用调度到节点上。

2.10.1.2. Pod 优先级名称

拥有一个或多个优先级类后，您可以创建 pod，并在 **Pod** 规格中指定优先级类名称。优先准入控制器使用优先级类名称字段来填充优先级的整数值。如果没有找到给定名称的优先级类，pod 将被拒绝。

2.10.2. 了解 pod 抢占

当开发人员创建 pod 时，pod 会排入某一队列。如果开发人员为 pod 配置了 pod 优先级或抢占，调度程序会从队列中选取 pod，并尝试将 pod 调度到某个节点上。如果调度程序无法在满足 pod 的所有指定要求的适当节点上找到空间，则会为待处理 pod 触发抢占逻辑。

当调度程序在节点上抢占一个或多个 pod 时，较高优先级 **Pod** spec 的 **nominatedNodeName 字段** 将设为该节点的名称，**nodename** 字段也是如此。调度程序使用 **nominatedNodeName** 字段来跟踪为 pod 保留的资源，同时也向用户提供与集群中抢占相关的信息。

在调度程序抢占了某一较低优先级 pod 后，调度程序会尊重该 pod 的安全终止期限。如果在调度程序等待较低优先级 pod 终止过程中另一节点变为可用，调度程序会将较高优先级 pod 调度到该节点上。因此，**Pod** spec 的 **nominatedNodeName** 字段和 **nodeName** 字段可能会有所不同。

另外，如果调度程序在某一节点上抢占 pod 并正在等待终止，这时又有优先级比待处理 pod 高的 pod 需要调度，那么调度程序可以改为调度这个优先级更高的 pod。在这种情况下，调度程序会清除待处理 pod 的 **nominatedNodeName**，使该 pod 有资格调度到其他节点上。

抢占不一定从节点中移除所有较低优先级 pod。调度程序可以通过移除一部分较低优先级 pod 调度待处理 pod。

只有待处理 pod 能够调度到节点时，调度程序才会对这个节点考虑 pod 抢占。

2.10.2.1. 非抢占优先级类

抢占策略设置为 **Never** 的 Pod 会放置在较低优先级 pod 的调度队列中，但无法抢占其他 pod。等待调度的非抢占 pod 会保留在调度队列中，直到资源可用且可以调度。非抢占 pod 与其他 pod 一样，受调度程序后退避的影响。这意味着，如果调度程序尝试调度这些 pod，它们会以较低频率重试，允许在调度前调度其他优先级较低的 pod。

非抢占 pod 仍可被其他高优先级 pod 抢占。

2.10.2.2. Pod 抢占和其他调度程序设置

如果启用 pod 优先级与抢占功能，请考虑其他的调度程序设置：

pod 优先级和 pod 中断预算

pod 中断预算指定某一时间必须保持在线的副本的最小数量或百分比。如果您指定了 pod 中断预算，OpenShift Container Platform 会在抢占 pod 时尽力尊重这些预算。调度程序会尝试在不违反 pod 中断预算的前提下抢占 pod。如果找不到这样的 pod，则可能会无视 pod 中断预算要求而抢占较低优先级 pod。

pod 优先级和 pod 关联性

pod 关联性要求将新 pod 调度到与具有同样标签的其他 pod 相同的节点上。

如果待处理 pod 与节点上的一个或多个低优先级 pod 具有 pod 间关联性，调度程序就不能在不违反关联要求的前提下抢占较低优先级 pod。这时，调度程序会寻找其他节点来调度待处理 pod。但是，不能保证调度程序能够找到合适的节点，因此可能无法调度待处理 pod。

要防止这种情况，请仔细配置优先级相同的 pod 的 pod 关联性。

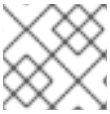
2.10.2.3. 安全终止被抢占的 pod

在抢占 pod 时，调度程序会等待 pod 安全终止期限到期，使 pod 能够完成工作并退出。如果 pod 在到期后没有退出，调度程序会终止该 pod。此安全终止期限会在调度程序抢占该 pod 的时间和待处理 pod 调度到节点的时间之间造成一个时间差。

要尽量缩短这个时间差，可以为较低优先级 pod 配置较短的安全终止期限。

2.10.3. 配置优先级和抢占

您可以通过创建优先级类对象并使用 pod 规格中的 **priorityClassName** 将 pod 与优先级关联来应用 pod 优先级与抢占。



注意

您不能直接将优先级类添加到现有调度的 pod 中。

流程

配置集群以使用优先级与抢占功能：

1. 创建一个或多个优先级类：
   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority (1)
value: 1000000 (2)
preemptionPolicy: PreemptLowerPriority (3)
globalDefault: false (4)
description: "This priority class should be used for XYZ service pods only." (5)
```

   (1) 优先级类对象的名称。

   (2) 对象的优先级值。

   (3) 可选。指定此优先级类是否被抢占或非抢占。抢占策略默认为 **PreemptLowerPriority**，它允许该优先级类中的 pod 抢占较低优先级 pod。如果抢占策略设置为 **Never**，则该优先级类中的 pod 就不会被抢占。

   (4) 可选。指定是否应该将这个优先级类用于没有指定优先级类名称的 pod。此字段默认为 **false**。集群中只能存在一个 **globalDefault** 设为 **true** 的优先级类。如果没有 **globalDefault:true** 的优先级类，则无优先级类名称的 pod 的优先级为零。添加具有 **globalDefault:true** 的优先级类只会影响在添加优先级类后创建的 pod，不会更改现有 pod 的优先级。

   (5) 可选。描述开发人员应该用于此优先级类的 pod。输入任意文本字符串。

   b. 创建优先级类：

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

2. 创建 pod spec 使其包含优先级类的名称：
   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
```

```
    labels:
      env: test
  spec:
    securityContext:
      runAsNonRoot: true
      seccompProfile:
        type: RuntimeDefault
    containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
    priorityClassName: high-priority (1)
```

(1) 指定要用于此 pod 的优先级类。

b. 创建 pod：

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

您可以将优先级名称直接添加到 pod 配置或 pod 模板中。

2.11. 使用节点选择器将 POD 放置到特定节点

*节点选择器*指定一个键值对映射。使用节点中的自定义标签和 pod 中指定的选择器来定义规则。

若要使 pod 有资格在某一节点上运行，pod 必须具有指定为该节点上标签的键值对。

如果您在同一 pod 配置中同时使用节点关联性和节点选择器，请查看下方的重要注意事项。

2.11.1. 使用节点选择器控制 pod 放置

您可以使用节点上的 pod 和标签上的节点选择器来控制 pod 的调度位置。使用节点选择器时，OpenShift Container Platform 会将 pod 调度到包含匹配标签的节点。

您可向节点、计算机器集或机器配置添加标签。将标签添加到计算机器集可确保节点或机器停机时，新节点具有该标签。如果节点或机器停机，添加到节点或机器配置的标签不会保留。

要将节点选择器添加到现有 pod 中，将节点选择器添加到该 pod 的控制对象中，如 **ReplicaSet** 对象、**DaemonSet** 对象、**StatefulSet** 对象、**Deployment** 对象或 **DeploymentConfig** 对象。任何属于该控制对象的现有 pod 都会在具有匹配标签的节点上重新创建。如果要创建新 pod，可以将节点选择器直接添加到 pod 规格中。如果 pod 没有控制对象，您必须删除 pod，编辑 pod 规格并重新创建 pod。

注意

您不能直接将节点选择器添加到现有调度的 pod 中。

先决条件

要将节点选择器添加到现有 pod 中，请确定该 pod 的控制对象。例如，**router-default-66d5cf9464-m2g75** pod 由 **router-default-66d5cf9464** 副本集控制：

```
$ oc describe pod router-default-66d5cf9464-7pwkc
```

输出示例

```
kind: Pod
apiVersion: v1
metadata:
# ...
Name:               router-default-66d5cf9464-7pwkc
Namespace:          openshift-ingress
# ...
Controlled By:      ReplicaSet/router-default-66d5cf9464
# ...
```

Web 控制台在 pod YAML 的 **ownerReferences** 下列出控制对象：

```
apiVersion: v1
kind: Pod
metadata:
  name: router-default-66d5cf9464-7pwkc
# ...
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: router-default-66d5cf9464
      uid: d81dd094-da26-11e9-a48a-128e7edf0312
      controller: true
      blockOwnerDeletion: true
# ...
```

流程

1. 使用计算机器集或直接编辑节点，为节点添加标签：

   - 在创建节点时，使用 **MachineSet** 对象向由计算机器集管理的节点添加标签：

     a. 运行以下命令，将标签添加到 **MachineSet** 对象中：

     ```
     $ oc patch MachineSet <name> --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
     <value>","<key>"="<value>"}}]'  -n openshift-machine-api
     ```

     例如：

     ```
     $ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
     node","region":"east"}}]'  -n openshift-machine-api
     ```

提示

您还可以应用以下 YAML 来向计算机器集中添加标签：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: xf2bd-infra-us-east-2a
  namespace: openshift-machine-api
spec:
  template:
    spec:
      metadata:
        labels:
          region: "east"
          type: "user-node"
# ...
```

b. 使用 **oc edit** 命令验证标签是否已添加到 **MachineSet** 对象中：
例如：

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

MachineSet 对象示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet

# ...

spec:
# ...
  template:
    metadata:
# ...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
# ...
```

- 直接向节点添加标签：

a. 为节点编辑 **Node** 对象：

```
$ oc label nodes <name> <key>=<value>
```

例如，若要为以下节点添加标签：

```
$ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
```

提示

您还可以应用以下 YAML 来向节点添加标签：

```
kind: Node
apiVersion: v1
metadata:
  name: hello-node-6fbccf8d9
  labels:
    type: "user-node"
    region: "east"
# ...
```

b. 验证标签是否已添加到节点：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                          STATUS   ROLES    AGE   VERSION
ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.29.4
```

2. 将匹配的节点选择器添加到 pod：

- 要将节点选择器添加到现有和未来的 pod，请向 pod 的控制对象添加节点选择器：

带有标签的 ReplicaSet 对象示例

```
kind: ReplicaSet
apiVersion: apps/v1
metadata:
  name: hello-node-6fbccf8d9
# ...
spec:
# ...
  template:
    metadata:
      creationTimestamp: null
      labels:
        ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
        pod-template-hash: 66d5cf9464
    spec:
      nodeSelector:
        kubernetes.io/os: linux
        node-role.kubernetes.io/worker: ''
        type: user-node 1
# ...
```

1 添加节点选择器。

- 要将节点选择器添加到一个特定的新 pod，直接将选择器添加到 **Pod** 对象中：

使用节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-node-6fbccf8d9
# ...
spec:
  nodeSelector:
    region: east
    type: user-node
# ...
```

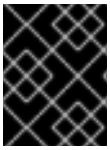
注意

您不能直接将节点选择器添加到现有调度的 pod 中。

2.12. RUN ONCE DURATION OVERRIDE OPERATOR

2.12.1. Run Once Duration Override Operator 概述

您可以使用 Run Once Duration Override Operator 指定运行一次 pod 的最大时间限制。



重要

目前，OpenShift Container Platform 4.16 不提供 Run Once Duration Override Operator。计划在不久的将来发布 Operator。

2.12.1.1. 关于 Run Once Duration Override Operator

OpenShift Container Platform 依赖于运行一次 pod 来执行诸如部署 pod 或执行构建等任务。Run-once pod 是带有 **RestartPolicy** 为 **Never** 或 **OnFailure** 的 pod。

集群管理员可以使用 Run Once Duration Override Operator 来强制限制这些运行一次 pod 处于活跃状态的时间。时间限制过期后，集群将尝试主动终止这些 pod。具有此类限制的主要原因是防止构建等任务运行过长的时间。

要将 Run Once Duration Override Operator 中的运行一次持续时间覆盖应用到运行一次的 pod，您必须在每个适用的命名空间中启用它。

如果运行一次的 pod 和 Run Once Duration Override Operator 都设置了其 **activeDeadlineSeconds** 值，则会使用这两个值中的低值。

2.12.2. Run Once Duration Override Operator 发行注记

集群管理员可以使用 Run Once Duration Override Operator 来强制对运行一次 pod 处于活跃状态的时间强制限制。时间限制过期后，集群会尝试终止运行一次的 pod。具有此类限制的主要原因是防止构建等任务运行过长的时间。

要将 Run Once Duration Override Operator 中的运行一次持续时间覆盖应用到运行一次的 pod，您必须在每个适用的命名空间中启用它。

本发行注记介绍了为 OpenShift Container Platform 的 Run Once Duration Override Operator 的开发。

有关 Run Once Duration Override Operator 的概述，请参阅关于 Run Once Duration Override Operator。

2.12.2.1. Run Once Duration Override Operator 1.1.1

发布日期：24 年 7 月 1 日

以下公告可用于 Run Once Duration Override Operator 1.1.1: RHSA-2024:1616

2.12.2.1.1. 新功能及功能增强

- 您可以在以 FIPS 模式运行的 OpenShift Container Platform 集群中安装和使用 Run Once Duration Override Operator。

重要

要为集群启用 FIPS 模式，您必须从配置为以 FIPS 模式操作的 Red Hat Enterprise Linux (RHEL) 计算机运行安装程序。有关在 RHEL 中配置 FIPS 模式的更多信息，请参阅在 FIPS 模式中安装该系统。

当以 FIPS 模式运行 Red Hat Enterprise Linux (RHEL) 或 Red Hat Enterprise Linux CoreOS (RHCOS)时，OpenShift Container Platform 核心组件使用 RHEL 加密库，在 x86_64、ppc64le 和 s390x 架构上提交到 NIST FIPS 140-2/140-3 Validation。

2.12.2.1.2. 程序错误修复

- 此 Run Once Duration Override Operator 发行版本解决了几个常见漏洞和暴露 (CVE)。

2.12.3. 覆盖运行一次 pod 的活动期限

您可以使用 Run Once Duration Override Operator 指定运行一次 pod 的最大时间限制。通过在命名空间上启用运行一次持续时间覆盖，以后在该命名空间中创建或更新的所有运行一次 pod 将其 **activeDeadlineSeconds** 字段设置为 Run Once Duration Override Operator 指定的值。

重要

目前，OpenShift Container Platform 4.16 不提供 Run Once Duration Override Operator。计划在不久的将来发布 Operator。

2.12.3.1. 安装 Run Once Duration Override Operator

您可以使用 Web 控制台安装 Run Once Duration Override Operator。

先决条件

- 您可以使用 **cluster-admin** 权限访问集群。
- 访问 OpenShift Container Platform web 控制台。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。

2. 为 Run Once Duration Override Operator 创建所需的命名空间。
   a. 进行 **Administration → Namespaces**，点 **Create Namespace**。
   b. 在 **Name** 字段中输入 **openshift-run-once-duration-override-operator**，然后点 **Create**。
3. 安装 Run Once Duration Override Operator。
   a. 导航至 **Operators → OperatorHub**。
   b. 在过滤器框中输入 **Run Once Duration Override Operator**。
   c. 选择 **Run Once Duration Override Operator** 并点 **Install**。
   d. 在 **Install Operator** 页面中：
      i. **Update channel** 设置为 **stable**，它会安装 Run Once Duration Override Operator 的最新稳定版本。
      ii. 选择 **A specific namespace on the cluster**。
      iii. 从 **Installed namespace** 下的下拉菜单中选择 **openshift-run-once-duration-override-operator**。
      iv. 选择一个 **更新批准策略**。
         - **Automatic** 策略允许 Operator Lifecycle Manager（OLM）在有新版本可用时自动更新 Operator。
         - **Manual** 策略需要拥有适当凭证的用户批准 Operator 更新。
      v. 点 **Install**。
4. 创建 **RunOnceDurationOverride** 实例。
   a. 在 **Operators → Installed Operators** 页面中，点 **Run Once Duration Override Operator**。
   b. 选择 **Run Once Duration Override** 选项卡，然后点 **Create RunOnceDurationOverride**。
   c. 根据需要编辑设置。
   在 **runOnceDurationOverride** 部分下，您可以更新 **spec.activeDeadlineSeconds** 值（如果需要）。预定义的值为 **3600** 秒，或 1 小时。
   d. 点 **Create**。

验证

1. 登录到 OpenShift CLI。
2. 验证所有 pod 均已创建并正确运行。

```
$ oc get pods -n openshift-run-once-duration-override-operator
```

输出示例

```
NAME                                             READY   STATUS    RESTARTS   AGE
run-once-duration-override-operator-7b88c676f6-lcxgc   1/1     Running   0          7m46s
```

```
runoncedurationoverride-62blp          1/1    Running   0        41s
runoncedurationoverride-h8h8b          1/1    Running   0        41s
runoncedurationoverride-tdsqk          1/1    Running   0        41s
```

2.12.3.2. 在命名空间中启用运行一次持续时间覆盖

要将 Run Once Duration Override Operator 中的运行一次持续时间覆盖应用到运行一次的 pod，您必须在每个适用的命名空间中启用它。

先决条件

- 已安装 Run Once Duration Override Operator。

流程

1. 登录到 OpenShift CLI。
2. 添加标签，为命名空间启用运行一次持续时间覆盖：

```
$ oc label namespace <namespace> \ 1
    runoncedurationoverrides.admission.runoncedurationoverride.openshift.io/enabled=true
```

1 指定要启用运行一次持续时间覆盖的命名空间。

在此命名空间中启用运行一次持续时间覆盖后，在此命名空间中未来创建的运行一次的 pod 会将其 **activeDeadlineSeconds** 字段设置为 Run Once Duration Override Operator 的覆盖值。此命名空间中的现有 pod 也会在下一次更新时将设置其 **activeDeadlineSeconds** 值。

验证

1. 在启用了运行一次持续时间覆盖的命名空间中创建一个测试运行一次 pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: example
  namespace: <namespace>                1
spec:
  restartPolicy: Never                  2
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: busybox
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
      image: busybox:1.25
      command:
        - /bin/sh
```

```
      - -ec
      - |
        while sleep 5; do date; done
```

1. 将 **<namespace>** 替换为您的命名空间的名称。
2. **restartPolicy** 必须是 **Never** 或 **OnFailure**，才能是一个运行一次的 pod。

2. 验证 pod 是否已设置其 **activeDeadlineSeconds** 字段：

```
$ oc get pods -n <namespace> -o yaml | grep activeDeadlineSeconds
```

输出示例

```
  activeDeadlineSeconds: 3600
```

2.12.3.3. 更新运行一次活跃截止时间覆盖值

您可以自定义 Run Once Duration Override Operator 适用于运行一次的 pod 的覆盖值。预定义的值为 **3600** 秒，或 1 小时。

先决条件

- 您可以使用 **cluster-admin** 权限访问集群。
- 已安装 Run Once Duration Override Operator。

流程

1. 登录到 OpenShift CLI。
2. 编辑 **RunOnceDurationOverride** 资源：

```
$ oc edit runoncedurationoverride cluster
```

3. 更新 **activeDeadlineSeconds** 字段：

```
apiVersion: operator.openshift.io/v1
kind: RunOnceDurationOverride
metadata:
# ...
spec:
  runOnceDurationOverride:
    spec:
      activeDeadlineSeconds: 1800 (1)
# ...
```

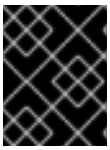
1. 将 **activeDeadlineSeconds** 字段设置为所需的值，以秒为单位。

4. 保存文件以使改变生效。

在启用了运行一次持续时间覆盖的命名空间中创建的任何运行后 pod 都会将其 **activeDeadlineSeconds** 字段设置为这个新值。这些命名空间中的现有运行一次 pod 会在更新时收到这个新值。

2.12.4. 卸载 Run Once Duration Override Operator

您可以通过卸载 Operator 并删除其相关资源，从 OpenShift Container Platform 中删除 Run Once Duration Override Operator。



重要

目前，OpenShift Container Platform 4.16 不提供 Run Once Duration Override Operator。计划在不久的将来发布 Operator。

2.12.4.1. 卸载 Run Once Duration Override Operator

您可以使用 Web 控制台卸载 Run Once Duration Override Operator。卸载 Run Once Duration Override Operator 不会取消设置运行一次的 pod 的 **activeDeadlineSeconds** 字段，但它不再将覆盖值应用到将来的运行一次 Pod。

先决条件

- 您可以使用 **cluster-admin** 权限访问集群。
- 访问 OpenShift Container Platform web 控制台。
- 已安装 Run Once Duration Override Operator。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 导航到 **Operators → Installed Operators**。
3. 从 **Project** 下拉列表中选择 **openshift-run-once-duration-override-operator**。
4. 删除 **RunOnceDurationOverride** 实例。
   a. 点 **Run Once Duration Override Operator** 并选择 **Run Once Duration Override** 选项卡。
   b. 点 **集群** 条目旁的 Options 菜单 并选择 **Delete RunOnceDurationOverride**。
   c. 在确认对话框中，点 **Delete**。
5. 卸载 Run Once Duration Override Operator Operator。
   a. 导航到 **Operators → Installed Operators**。
   b. 点 **Run Once Duration Override Operator** 条目旁边的 Options 菜单 ，并点 **Uninstall Operator**。
   c. 在确认对话框中，点 **Uninstall**。

2.12.4.2. 卸载 Run Once Duration Override Operator 资源

另外，在卸载 Run Once Duration Override Operator 后，您可以从集群中删除其相关资源。

先决条件

- 您可以使用 **cluster-admin** 权限访问集群。
- 访问 OpenShift Container Platform web 控制台。
- 您已卸载了 Run Once Duration Override Operator。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 删除安装 Run Once Duration Override Operator 时创建的 CRD：
   a. 进入到 **Administration** → **CustomResourceDefinitions**。
   b. 在 **Name** 字段中输入 **RunOnceDurationOverride** 来过滤 CRD。
   c. 点 **RunOnceDurationOverride** CRD 旁边的选项菜单 ，选择 **Delete CustomResourceDefinition**。
   d. 在确认对话框中，点 **Delete**。
3. 删除 **openshift-run-once-duration-override-operator** 命名空间。
   a. 导航至 **Administration** → **Namespaces**。
   b. 在过滤器框中输入 **openshift-run-once-duration-override-operator**。
   c. 点 **openshift-run-once-duration-override-operator** 条目旁的 Options 菜单 并选择 **Delete Namespace**。
   d. 在确认对话框中，输入 **openshift-run-once-duration-override-operator** 并点 **Delete**。
4. 从启用的命名空间中删除运行一次持续时间覆盖标签。
   a. 导航至 **Administration** → **Namespaces**。
   b. 选择您的命名空间。
   c. 点 **Labels** 字段旁的 **Edit**。
   d. 删除 **runonceduration**overrides.admission.runonceduration**override.openshift.io/enabled=true** 标签，然后点 **Save**。

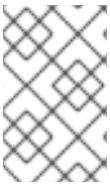
第 3 章 使用自定义 METRICS AUTOSCALER OPERATOR 自动扩展 POD

3.1. 发行注记

3.1.1. 自定义 Metrics Autoscaler Operator 发行注记

Red Hat OpenShift 的自定义 Metrics Autoscaler Operator 发行注记介绍了新的功能和增强功能、已弃用的功能以及已知的问题。

Custom Metrics Autoscaler Operator 使用基于 Kubernetes 的 Event Driven Autoscaler (KEDA)，并基于 OpenShift Container Platform 横向自动扩展(HPA)构建。



注意

Custom Metrics Autoscaler Operator for Red Hat OpenShift 作为可安装的组件提供，它与 OpenShift Container Platform 核心不同。Red Hat OpenShift Container Platform 生命周期政策概述了发行版本兼容性。

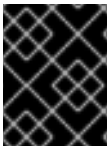
3.1.1.1. 支持的版本

下表为每个 OpenShift Container Platform 版本定义自定义 Metrics Autoscaler Operator 版本。

版本	OpenShift Container Platform 版本	公开发行（GA）
2.12.1	4.15	公开发行（GA）
2.12.1	4.14	公开发行（GA）
2.12.1	4.13	公开发行（GA）
2.12.1	4.12	公开发行（GA）

3.1.1.2. 自定义 Metrics Autoscaler Operator 2.12.1-394 发行注记

此自定义 Metrics Autoscaler Operator 2.12.1-394 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了程序错误修正。以下公告可用于 RHSA-2024:2901。



重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.1.2.1. 程序错误修复

- 在以前的版本中，当对无效 JSON 的特定表单进行 unmarshaling 处理时，**protojson.Unmarshal** 函数会进入一个死循环。当 unmarshaling 到包含 **google.protobuf.Any** 值或设置了 **UnmarshalOptions.DiscardUnknown** 选项时，可能会出现此条件。此发行版本解决了这个问题。(OCPBUGS-30305)

- 在以前的版本中，当解析多部分表单时，可以明确使用 **Request.ParseMultipartForm** 方法明，或使用 **Request.FormValue**、**Request.PostFormValue** 或 **Request.FormFile** 方法隐式应用，解析表单的总大小限制不适用于在读单一表单行时消耗的内存。这可能会允许在恶意设计的输入中包含非常长的行，从而导致分配大量内存，这可能会导致内存耗尽。在这个版本中，解析过程可以正确地限制表单行的最大大小。(OCPBUGS-30360)

- 在以前的版本中，当遵循 HTTP 重定向到不在匹配子域或初始域的完全匹配的域时，HTTP 客户端不会转发敏感标头，如 **Authorization** 或 **Cookie**。例如：从 example.com 到 www.example.com 的重定向会转发 **Authorization** 标头，但重定向到 www.example.org 不会转发标头。恶意精心设计的 HTTP 重定向可能会导致敏感标头被意外转发。此发行版本解决了这个问题。(OCPBUGS-30365)

- 在以前的版本中，验证包含带有未知公钥算法的证书的证书链会导致证书验证过程 panic。此条件会影响将 **Config.ClientAuth** 参数设置为 **VerifyClientCertIfGiven** 或 **RequireAndVerifyClientCert** 值的所有加密和 TLS 客户端和服务器。默认行为是 TLS 服务器无法验证客户端证书。此发行版本解决了这个问题。(OCPBUGS-30370)

- 在以前的版本中，如果从 **MarshalJSON** 方法返回的错误包含用户控制的数据，数据可能会被用来破坏 HTML 模板软件包的上下文自动转义行为。此条件允许后续操作将意外内容注入模板。此发行版本解决了这个问题。(OCPBUGS-30397)

- 在以前的版本中，**net/http** 和 **golang.org/x/net/http2** Go 软件包没有限制为 HTTP/2 请求读取的 **CONTINUATION** 帧的数量。此条件允许攻击者为单个请求提供任意的一组大量标头，这些标头将被读取、解码，然后丢弃。这可能导致 CPU 消耗过量。此发行版本解决了这个问题。(OCPBUGS-30894)

3.1.2. Custom Metrics Autoscaler Operator 的过去发行版本发行注记

以下发行注记适用于以前的自定义 Metrics Autoscaler Operator 版本。

有关当前版本，请参阅自定义 Metrics Autoscaler Operator 发行注记。

3.1.2.1. 自定义 Metrics Autoscaler Operator 2.12.1-384 发行注记

此自定义 Metrics Autoscaler Operator 2.12.1-384 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。以下公告可用于 RHBA-2024:2043。

重要

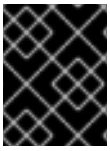
在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.1.1. 程序错误修复

- 在以前的版本中，**custom-metrics-autoscaler** 和 **custom-metrics-autoscaler-adapter** 镜像缺少时区信息。因此，带有 **cron** 触发器的扩展对象无法正常工作，因为控制器无法找到时区信息。在这个版本中，镜像构建被更新为包含时区信息。因此，包含 **cron** 触发器的对象现在可以正常工作。(OCPBUGS-32395)

3.1.2.2. 自定义 Metrics Autoscaler Operator 2.12.1-376 发行注记

此自定义 Metrics Autoscaler Operator 2.12.1-376 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了安全更新和程序错误修复。以下公告可用于 RHSA-2024:1812。



重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.2.1. 程序错误修复

- 在以前的版本中，如果在扩展对象元数据中指定无效值，如不存在的命名空间，则底层 scaler 客户端无法释放或关闭其客户端描述符，从而导致内存泄漏。在这个版本中，当出现错误时可以正确地关闭底层客户端描述符，从而导致内存泄漏。(OCPBUGS-30145)

- 在以前的版本中，**keda-metrics-apiserver** pod 的 **ServiceMonitor** 自定义资源 (CR) 无法正常工作，因为 CR 引用了 **http** 的错误指标端口名称。在这个版本中，**ServiceMonitor** CR 修正了引用 **metrics** 的正确端口名称。因此，Service Monitor 可以正常工作。(OCPBUGS-25806)

3.1.2.3. 自定义 Metrics Autoscaler Operator 2.11.2-322 发行注记

此自定义 Metrics Autoscaler Operator 2.11.2-322 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了安全更新和程序错误修复。以下公告可用于 RHSA-2023:6144。

重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.3.1. 程序错误修复

- 因为自定义 Metrics Autoscaler Operator 版本 3.11.2-311 已被发布，所以在 Operator 部署中不需要卷挂载，所以自定义 Metrics Autoscaler Operator pod 会每 15 分钟重启。在这个版本中，在 Operator 部署中添加了所需的卷挂载。因此，Operator 不再每 15 分钟重启。(OCPBUGS-22361)

3.1.2.4. 自定义 Metrics Autoscaler Operator 2.11.2-311 发行注记

此自定义 Metrics Autoscaler Operator 2.11.2-311 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。自定义 Metrics Autoscaler Operator 2.11.2-311 的组件在 RHBA-2023:5981 中发布。

重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.4.1. 新功能及功能增强

3.1.2.4.1.1. 现在支持 Red Hat OpenShift Service on AWS (ROSA) 和 OpenShift Dedicated

自定义 Metrics Autoscaler Operator 2.11.2-311 可以安装在 OpenShift ROSA 和 OpenShift Dedicated 受管集群上。自定义 Metrics Autoscaler Operator 的早期版本只能安装在 **openshift-keda** 命名空间中。这导致 Operator 无法安装到 OpenShift ROSA 和 OpenShift Dedicated 集群中。此自定义 Metrics Autoscaler 版本允许安装到其他命名空间，如 **openshift-operators** 或 **keda**，从而可以安装到 ROSA 和 Dedicated 集群中。

3.1.2.4.2. 程序错误修复

- 在以前的版本中，如果安装并配置 Custom Metrics Autoscaler Operator，但没有使用，OpenShift CLI 会在任何 **oc** 命令输入后报告 **could not get resource list for external.metrics.k8s.io/v1beta1: Got empty response for: external.metrics.k8s.io/v1beta1** 错误。虽然这个消息并没有什么危害，但可能会造成混淆。在这个版本中，**Got empty response for: external.metrics...** 不再会出现。(OCPBUGS-15779)

- 在以前的版本中，任何注解或标签更改为由自定义 Metrics Autoscaler 管理的对象在修改 Keda Controller 时（例如在配置更改后）会被自定义 Metrics Autoscaler 恢复。这会导致对象中的标签持续更改。自定义 Metrics Autoscaler 现在使用自己的注解来管理标签和注解，注解或标签不再被错误地恢复。(OCPBUGS-15590)

3.1.2.5. 自定义 Metrics Autoscaler Operator 2.10.1-267 发行注记

此自定义 Metrics Autoscaler Operator 2.10.1-267 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。自定义 Metrics Autoscaler Operator 2.10.1-267 组件在 RHBA-2023:4089 中发布。

重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.5.1. 程序错误修复

- 在以前的版本中，**custom-metrics-autoscaler** 和 **custom-metrics-autoscaler-adapter** 镜像不包含时区信息。因此，带有 cron 触发器的扩展对象无法正常工作，因为控制器无法找到时区信息。在这个版本中，镜像构建包含时区信息。因此，包含 cron 触发器的对象现在可以正常工作。(OCPBUGS-15264)

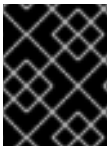
- 在以前的版本中，自定义 Metrics Autoscaler Operator 会尝试拥有所有受管对象，包括其他命名空间中的对象和集群范围的对象。因此，自定义 Metrics Autoscaler Operator 无法创建角色绑定来读取 API 服务器所需的凭证。这会导致 **kube-system** 命名空间中出现错误。在这个版本中，自定义 Metrics Autoscaler Operator 会跳过将 **ownerReference** 字段添加到另一个命名空间中的任何对象或任何集群范围的对象。现在，角色绑定会被创建，且没有任何错误。(OCPBUGS-15038)

- 在以前的版本中，自定义 Metrics Autoscaler Operator 将 **ownerReferences** 字段添加到 **openshift-keda** 命名空间中。虽然这不会造成功能问题，但存在此字段可能会给集群管理员造成混淆。在这个版本中，自定义 Metrics Autoscaler Operator 不会将 **ownerReference** 字段添加到 **openshift-keda** 命名空间中。因此，**openshift-keda** 命名空间不再有一个 superfluous **ownerReference** 字段。(OCPBUGS-15293)

- 在以前的版本中，如果您使用使用 pod 身份以外的身份验证方法配置的 Prometheus 触发器，并且 **podIdentity** 参数设置为 **none**，则触发器将无法扩展。在这个版本中，OpenShift 的自定义 Metrics Autoscaler 可以正确地处理 **none** pod 身份提供程序类型。因此，使用 pod 身份以外的身份验证方法配置的 Prometheus 触发器，其 **podIdentity** 参数设置为 **none** 现在可以正确扩展。(OCPBUGS-15274)

3.1.2.6. 自定义 Metrics Autoscaler Operator 2.10.1 发行注记

此自定义 Metrics Autoscaler Operator 2.10.1 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。自定义 Metrics Autoscaler Operator 2.10.1 的组件在 RHEA-2023:3199 中发布。



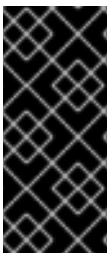
重要

在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

3.1.2.6.1. 新功能及功能增强

3.1.2.6.1.1. 自定义 Metrics Autoscaler Operator 正式发布

现在，自定义 Metrics Autoscaler Operator 从自定义 Metrics Autoscaler Operator 版本 2.10.1 开始正式发布。



重要

使用扩展作业进行扩展只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

3.1.2.6.1.2. 性能指标

现在，您可以使用 Prometheus Query Language (PromQL) 查询自定义 Metrics Autoscaler Operator 的指标。

3.1.2.6.1.3. 暂停扩展对象的自定义指标自动扩展

现在，您可以根据需要暂停扩展对象的自动扩展，并在就绪时恢复自动扩展。

3.1.2.6.1.4. 副本回退到扩展的对象

现在，如果扩展对象无法从源获取指标，您可以指定要回退到的副本数。

3.1.2.6.1.5. 为扩展对象自定义 HPA 命名

现在，您可以在扩展的对象中为 pod 横向自动扩展指定自定义名称。

3.1.2.6.1.6. 激活和扩展阈值

因为 pod 横向自动扩展 (HPA) 无法扩展到 0 个副本或从 0 个副本进行扩展，所以在 HPA 执行缩放后，自定义 Metrics Autoscaler Operator 会进行该扩展。现在，您可以根据副本数指定 HPA 接管自动扩展的时间。这可以提高扩展策略的灵活性。

3.1.2.7. 自定义 Metrics Autoscaler Operator 2.8.2-174 发行注记

此自定义 Metrics Autoscaler Operator 2.8.2-174 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。Custom Metrics Autoscaler Operator 2.8.2-174 组件在 RHEA-2023:1683 中发布。

重要

自定义 Metrics Autoscaler Operator 版本 2.8.2-174 是一个技术预览功能。

3.1.2.7.1. 新功能及功能增强

3.1.2.7.1.1. Operator 升级支持

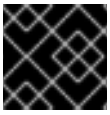
现在，您可以从 Custom Metrics Autoscaler Operator 的早期版本升级。有关升级 Operator 的信息，请参阅"添加资源"中的"删除 Operator 更新频道"。

3.1.2.7.1.2. must-gather 支持

现在，您可以使用 OpenShift Container Platform **must-gather** 工具收集有关自定义 Metrics Autoscaler Operator 及其组件的数据。目前，使用带有自定义 Metrics Autoscaler 的 **must-gather** 工具的过程与其他 Operator 不同。如需更多信息，请参阅"添加资源"中的调试数据。

3.1.2.8. 自定义 Metrics Autoscaler Operator 2.8.2 发行注记

此自定义 Metrics Autoscaler Operator 2.8.2 发行版本为在 OpenShift Container Platform 集群中运行的 Operator 提供了新功能和程序错误修复。自定义 Metrics Autoscaler Operator 2.8.2 组件在 RHSA-2023:1042 中发布。



重要

自定义 Metrics Autoscaler Operator 版本 2.8.2 是一个技术预览功能。

3.1.2.8.1. 新功能及功能增强

3.1.2.8.1.1. 审计日志记录

现在，您可以收集并查看自定义 Metrics Autoscaler Operator 及其相关组件的审计日志。审计日志是安全相关的按时间排序的记录，记录各个用户、管理员或其他系统组件影响系统的一系列活动。

3.1.2.8.1.2. 基于 Apache Kafka 指标扩展应用程序

现在，您可以使用 KEDA Apache kafka 触发器/scaler 根据 Apache Kafka 主题扩展部署。

3.1.2.8.1.3. 根据 CPU 指标扩展应用程序

现在，您可以使用 KEDA CPU 触发器/scaler 根据 CPU 指标扩展部署。

3.1.2.8.1.4. 根据内存指标扩展应用程序

现在，您可以使用 KEDA 内存触发器/scaler 根据内存指标扩展部署。

3.2. 自定义 METRICS AUTOSCALER OPERATOR 概述

作为开发者，您可以使用 Custom Metrics Autoscaler Operator for Red Hat OpenShift 指定 OpenShift Container Platform 如何根据不基于 CPU 或内存的自定义指标自动增加或减少部署、有状态集、自定义资源或作业的数量。

Custom Metrics Autoscaler Operator 是一个基于 Kubernetes Event Driven Autoscaler (KEDA) 的可选 Operator，允许使用 pod 指标以外的其他指标源扩展工作负载。

自定义指标自动扩展目前仅支持 Prometheus、CPU、内存和 Apache Kafka 指标。

Custom Metrics Autoscaler Operator 根据特定应用程序的自定义外部指标扩展 pod。您的其他应用程序继续使用其他扩展方法。您可以配置 *触发器*（也称为 scaler），这是自定义指标自动扩展器用来决定如何扩展的事件和指标的来源。自定义指标自动扩展使用 metrics API 将外部指标转换为 OpenShift Container Platform 可以使用的形式。自定义指标自动扩展会创建一个执行实际缩放的 pod 横向自动扩展 (HPA)。

要使用自定义指标自动扩展，您可以为工作负载创建一个 **ScaledObject** 或 **ScaledJob** 对象，这是定义扩展元数据的自定义资源(CR)。您可以指定要缩放的部署或作业、要缩放的指标源 (trigger) 以及其他参数，如允许的最小和最大副本数。

注意

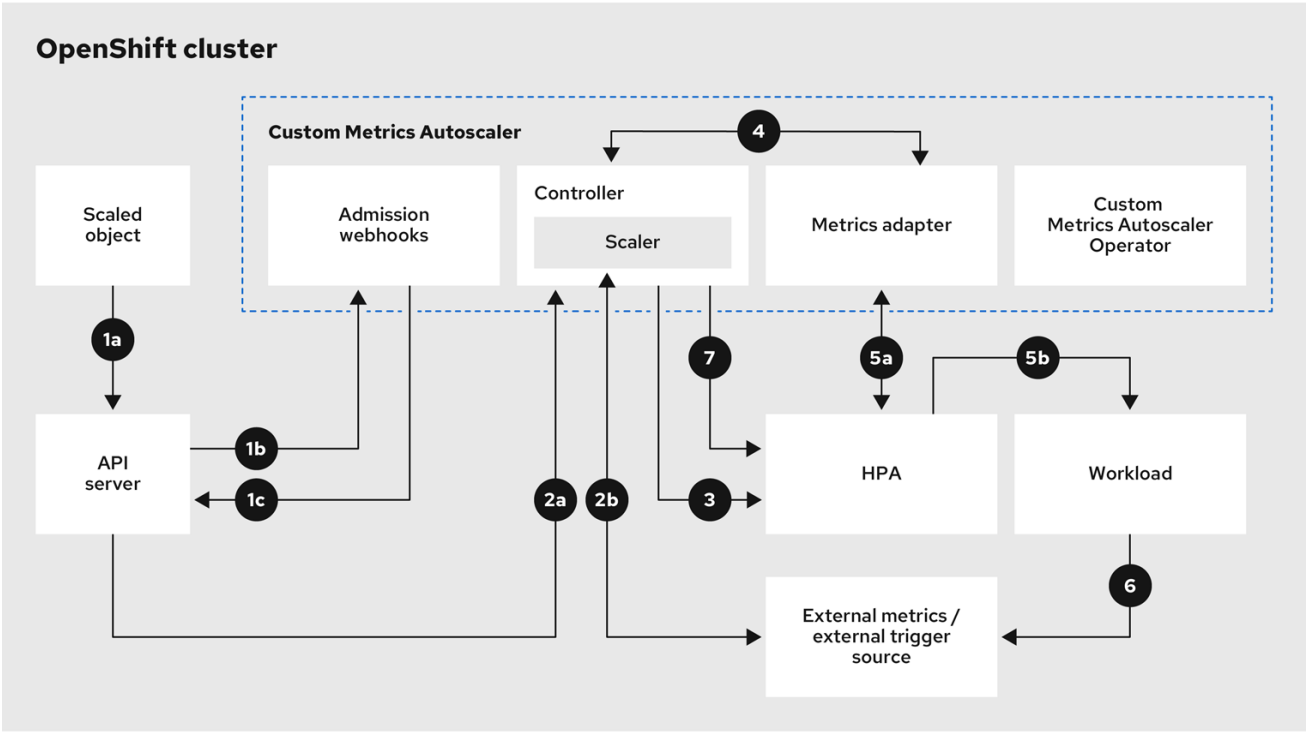
您只能为每个您要扩展的工作负载创建一个扩展对象或扩展作业。另外，您不能在同一工作负载中使用扩展的对象或扩展作业以及 pod 横向自动扩展 (HPA)。

自定义指标自动扩展与 HPA 不同，可以缩减为零。如果将自定义指标自动扩展 CR 中的 **minReplicaCount** 值设置为 0，自定义指标自动扩展会将工作负载从 1 缩减到 **0** 个副本或从 0 个副本扩展到 1。这称为 *激活阶段*。扩展至 1 个副本后，HPA 会控制扩展。这称为 *扩展阶段*。

某些触发器允许您更改由集群指标自动扩展扩展的副本数量。在所有情况下，配置激活阶段的参数始终使用相同的短语，前缀为 *激活*。例如，如果 **threshold** 参数配置缩放，则 **activationThreshold** 将配置激活。通过配置激活和扩展阶段，您可以提高扩展策略的灵活性。例如，您可以配置更高的激活阶段，以便在指标特别低时防止扩展或缩减。

当每个决策不同时，激活值的优先级高于扩展值。例如，如果 **threshold** 被设置为 **10**，并且 **activationThreshold** 为 **50**，如果指标报告 **40**，则缩放器不会激活，并且 pod 缩减为零，即使 HPA 需要 4 个实例。

图 3.1. 自定义指标自动扩展工作流



1. 您可以为集群中的工作负载创建或修改扩展对象自定义资源。对象包含该工作负载的扩展配置。在接受新对象前，OpenShift API 服务器将其发送到自定义指标自动扩展准入 webhook 进程，以确保对象有效。如果验证成功，API 服务器会保留对象。

2. 自定义指标自动扩展控制器监视是否有新的或修改的扩展对象。当 OpenShift API 服务器通知更改控制器时，控制器会监控任何外部触发器源（也称为数据源）在对象中指定以更改指标数据。一个或多个 scalers 请求从外部触发器源扩展数据。例如，对于 Kafka 触发器类型，控制器使用 Kafka scaler 与 Kafka 实例通信来获取触发器请求的数据。

3. 控制器为扩展的对象创建一个 pod 横向自动扩展对象。因此，Horizontal Pod Autoscaler (HPA) Operator 开始监控与触发器关联的扩展数据。HPA 请求从集群 OpenShift API 服务器端点扩展数据。

4. OpenShift API 服务器端点由自定义指标自动扩展指标适配器提供。当 metrics 适配器收到自定义指标的请求时，它使用 GRPC 连接控制器来请求它以获取从 scaler 接收的最新触发器数据。

5. HPA 根据从 metrics adapter 接收的数据做出缩放决策，并通过增加或减少副本来扩展工作负载。

6. 当它运行时，工作负载可能会影响扩展指标。例如，如果扩展工作负载以处理 Kafka 队列中的工作，则队列大小会在工作负载处理所有工作后减小。因此，工作负载会缩减。

7. 如果指标位于 **minReplicaCount** 值指定的范围内，自定义指标自动扩展控制器会禁用所有扩展，并将副本数保留为固定级别。如果指标超过该范围，自定义指标自动扩展控制器将启用扩展并允许 HPA 扩展工作负载。当禁用扩展时，HPA 不会执行任何操作。

3.3. 安装自定义指标自动扩展

您可以使用 OpenShift Container Platform Web 控制台安装自定义 Metrics Autoscaler Operator。

安装会创建以下五个 CRD：

- **ClusterTriggerAuthentication**
- **KedaController**
- **ScaledJob**
- **ScaledObject**
- **TriggerAuthentication**

3.3.1. 安装自定义指标自动扩展

您可以使用以下步骤安装自定义 Metrics Autoscaler Operator。

先决条件

- 删除之前安装的 Cluster Metrics Autoscaler Operator 的技术预览版本。
- 删除基于社区的 KEDA 的任何版本。
  另外，运行以下命令来删除 KEDA 1.x 自定义资源定义：

```
$ oc delete crd scaledobjects.keda.k8s.io
```

```
$ oc delete crd triggerauthentications.keda.k8s.io
```

流程

1. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
2. 从可用的 Operator 列表中选择 **Custom Metrics Autoscaler**，然后点 **Install**。
3. 在 **Install Operator** 页面中，确保为 **Installation Mode** 选择 **All namespaces on the cluster(default)** 选项。这会在所有命名空间中安装 Operator。
4. 确保为 **Installed Namespace** 选择了 **openshift-keda** 命名空间。如果集群中不存在命名空间，OpenShift Container Platform 会创建命名空间。
5. 点 **Install**。
6. 列出自定义 Metrics Autoscaler Operator 组件来验证安装：
   a. 导航到 **Workloads → Pods**。
   b. 从下拉菜单中选择 **openshift-keda** 项目，并验证 **custom-metrics-autoscaler-operator-*** pod 正在运行。
   c. 进入到 **Workloads → Deployments** 以验证 **custom-metrics-autoscaler-operator** 部署是否正在运行。
7. 可选：使用以下命令在 OpenShift CLI 中验证安装：

   ```
   $ oc get all -n openshift-keda
   ```

   输出结果类似如下：

   输出示例

   ```
   NAME                                                READY   STATUS    RESTARTS   AGE
   pod/custom-metrics-autoscaler-operator-5fd8d9ffd8-xt4xp   1/1     Running   0          18m

   NAME                                                READY   UP-TO-DATE   AVAILABLE   AGE
   deployment.apps/custom-metrics-autoscaler-operator   1/1     1            1           18m

   NAME                                                           DESIRED   CURRENT   READY   AGE
   replicaset.apps/custom-metrics-autoscaler-operator-5fd8d9ffd8   1         1         1       18m
   ```

8. 安装 **KedaController** 自定义资源，该资源创建所需的 CRD：
   a. 在 OpenShift Container Platform web 控制台中，点击 **Operators → Installed Operators**。
   b. 点 **Custom Metrics Autoscaler**。
   c. 在 **Operator Details** 页面中，点 **KedaController** 选项卡。
   d. 在 **KedaController** 选项卡中，点 **Create KedaController** 并编辑文件。

   ```
   kind: KedaController
   apiVersion: keda.sh/v1alpha1
   ```

```
metadata:
  name: keda
  namespace: openshift-keda
spec:
  watchNamespace: '' (1)
  operator:
    logLevel: info (2)
    logEncoder: console (3)
  metricsServer:
    logLevel: '0' (4)
    auditConfig: (5)
      logFormat: "json"
      logOutputVolumeClaim: "persistentVolumeClaimName"
      policy:
        rules:
        - level: Metadata
        omitStages: ["RequestReceived"]
        omitManagedFields: false
      lifetime:
        maxAge: "2"
        maxBackup: "1"
        maxSize: "50"
  serviceAccount: {}
```

1. 指定自定义 Metrics Autoscaler Operator 应该在其中扩展应用程序的单个命名空间。将它留空，或将其留空，以便在所有命名空间中扩展应用程序。此字段应具有命名空间或为空。默认值为空。
2. 指定自定义 Metrics Autoscaler Operator 日志消息的详细程度。允许的值有 **debug**、**info** 和**error**。默认为 **info**。
3. 指定 Custom Metrics Autoscaler Operator 日志消息的日志记录格式。允许的值是 **console** 或 **json**。默认为 **console**。
4. 指定自定义 Metrics Autoscaler Metrics 服务器的日志记录级别。允许的值是 **0**（用于 **info**）和 **4**（用于 **debug** ）。默认值为 **0**。
5. 激活自定义 Metrics Autoscaler Operator 的审计日志记录，并指定要使用的审计策略，如"配置审计日志记录"部分中所述。

e. 点 **Create** 创建 KEDA 控制器。

3.4. 了解自定义指标自动扩展触发器

触发器（也称为 scalers）提供自定义 Metrics Autoscaler Operator 用来扩展 pod 的指标。

自定义指标自动扩展目前只支持 Prometheus、CPU、内存和 Apache Kafka 触发器。

您可以使用 **ScaledObject** 或 **ScaledJob** 自定义资源为特定对象配置触发器，如后面的章节中所述。

3.4.1. 了解 Prometheus 触发器

您可以基于 Prometheus 指标扩展 pod，该指标可以使用已安装的 OpenShift Container Platform 监控或外部 Prometheus 服务器作为指标源。有关使用 OpenShift Container Platform 监控作为指标源所需的配置的信息，请参阅"附加资源"。

注意

如果 Prometheus 从自定义指标自动扩展扩展的应用程序收集指标，请不要在自定义资源中将最小副本设置为 **0**。如果没有应用程序 pod，自定义指标自动扩展没有任何要缩放的指标。

带有 Prometheus 目标的扩展对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: prom-scaledobject
  namespace: my-namespace
spec:
# ...
  triggers:
  - type: prometheus (1)
    metadata:
      serverAddress: https://thanos-querier.openshift-monitoring.svc.cluster.local:9092 (2)
      namespace: kedatest (3)
      metricName: http_requests_total (4)
      threshold: '5' (5)
      query: sum(rate(http_requests_total{job="test-app"}[1m])) (6)
      authModes: basic (7)
      cortexOrgID: my-org (8)
      ignoreNullValues: "false" (9)
      unsafeSsl: "false" (10)
```

(1) 指定 Prometheus 作为触发器类型。

(2) 指定 Prometheus 服务器的地址。本例使用 OpenShift Container Platform 监控。

(3) 可选：指定您要缩放的对象的命名空间。如果将 OpenShift Container Platform 监控用作指标的源，则需要此参数。

(4) 指定在 **external.metrics.k8s.io** API 中标识指标的名称。如果您使用的是多个触发器，则所有指标名称都必须是唯一的。

(5) 指定触发扩展的值。必须指定为带引号的字符串值。

(6) 指定要使用的 Prometheus 查询。

(7) 指定要使用的身份验证方法。Prometheus scalers 支持 bearer 身份验证 (**bearer**)、基本身份验证 (**basic**) 或 TLS 身份验证 (**tls**)。您可以在触发器身份验证中配置特定的身份验证参数，如以下部分所述。根据需要，您还可以使用 secret。

(8) 可选：将 **X-Scope-OrgID** 标头传递给多租户 Cortex 或 Prometheus 的 Mimir 存储。这个参数只需要带有多租户 Prometheus 存储，以指示 Prometheus 应该返回哪些数据。

(9) 可选：指定在 Prometheus 目标丢失时触发器应如何进行操作。

- 如果为 **true**，当 Prometheus 目标丢失时触发器将继续操作。这是默认的行为。
- 如果为 **false**，当 Prometheus 目标丢失时触发器会返回错误。

10 可选：指定是否应跳过证书检查。例如，如果在 Prometheus 端点中使用自签名证书，您可以跳过检查。

- 如果为 **true**，则执行证书检查。
- 如果为 **false**，则不会执行证书检查。这是默认的行为。

3.4.1.1. 配置自定义指标自动扩展以使用 OpenShift Container Platform 监控

您可以使用已安装的 OpenShift Container Platform Prometheus 监控作为自定义指标自动扩展使用的指标的来源。但是，需要执行一些额外的配置。

注意

外部 Prometheus 源不需要这些步骤。

您必须执行以下任务，如本节所述：

- 创建服务帐户以获取令牌。
- 创建角色。
- 将该角色添加到服务帐户。
- 在 Prometheus 使用的触发器验证对象中引用令牌。

先决条件

- 必须安装 OpenShift Container Platform 监控。
- OpenShift Container Platform 监控中必须启用对用户定义的工作负载的监控监控，如**创建用户定义的工作负载监控配置映射**部分所述。
- 必须安装 Custom Metrics Autoscaler Operator。

流程

1. 使用您要缩放的对象切换到项目：

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

2. 如果您的集群没有服务帐户，请使用以下命令来创建服务帐户：

```
$ oc create serviceaccount <service_account>
```

其中：

<service_account>
指定服务帐户的名称。

3. 使用以下命令查找分配给服务帐户的令牌：

```
$ oc describe serviceaccount <service_account>
```

其中：

<service_account>

指定服务帐户的名称。

输出示例

```
Name:                thanos
Namespace:           my-project
Labels:              <none>
Annotations:         <none>
Image pull secrets:  thanos-dockercfg-nnwgj
Mountable secrets:   thanos-dockercfg-nnwgj
Tokens:              thanos-token-9g4n5 (1)
Events:              <none>
```

(1) 在触发器身份验证中使用此令牌。

4. 使用服务帐户令牌创建触发器身份验证：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: keda-trigger-auth-prometheus
spec:
  secretTargetRef: (1)
  - parameter: bearerToken (2)
    name: thanos-token-9g4n5 (3)
    key: token (4)
  - parameter: ca
    name: thanos-token-9g4n5
    key: ca.crt
```

(1) 指定此对象使用 secret 进行授权。

(2) 使用令牌指定要提供的身份验证参数。

(3) 指定要使用的令牌名称。

(4) 指定令牌中用于指定参数的密钥。

b. 创建 CR 对象：

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

5. 创建用于读取 Thanos 指标的角色：

a. 使用以下参数创建 YAML 文件：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: thanos-metrics-reader
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
- apiGroups:
  - metrics.k8s.io
  resources:
  - pods
  - nodes
  verbs:
  - get
  - list
  - watch
```

b. 创建 CR 对象：

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

6. 创建用于读取 Thanos 指标的角色绑定：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: thanos-metrics-reader 1
  namespace: my-project 2
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: thanos-metrics-reader
subjects:
- kind: ServiceAccount
  name: thanos 3
  namespace: my-project 4
```

1. 指定您创建的角色的名称。
2. 指定您要缩放的对象的命名空间。
3. 指定要绑定到角色的服务帐户的名称。
4. 指定您要缩放的对象的命名空间。

b. 创建 CR 对象：

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

现在，您可以部署扩展的对象或扩展作业来为应用程序启用自动扩展，如"了解如何添加自定义指标自动扩展"中所述。要将 OpenShift Container Platform 监控用作源，在触发器或 scaler 中，您必须包括以下参数：

- **triggers.type** 必须是 **prometheus**
- **triggers.metadata.serverAddress** 必须是 **https://thanos-querier.openshift-monitoring.svc.cluster.local:9092**
- **triggers.metadata.authModes** 必须是 **bearer**
- **triggers.metadata.namespace** 必须设置为要缩放的对象的命名空间
- **triggers.authenticationRef** 必须指向上一步中指定的触发器身份验证资源

3.4.2. 了解 CPU 触发器

您可以根据 CPU 指标扩展 pod。此触发器使用集群指标作为指标的源。

自定义指标自动扩展扩展与对象关联的 pod，以维护您指定的 CPU 用量。自动缩放器增加或减少最小和最大数量之间的副本数量，以维护所有 pod 的指定 CPU 使用率。内存触发器考虑整个 pod 的内存使用率。如果 pod 有多个容器，则内存触发器会考虑 pod 中所有容器的总内存使用率。

注意

- 此触发器不能与 **ScaledJob** 自定义资源一起使用。
- 当使用内存触发器扩展对象时，对象不会扩展到 **0**，即使您使用多个触发器。

使用 CPU 目标扩展对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: cpu-scaledobject
  namespace: my-namespace
spec:
# ...
  triggers:
  - type: cpu 1
    metricType: Utilization 2
    metadata:
      value: '60' 3
  minReplicaCount: 1 4
```

1. 指定 CPU 作为触发器类型。
2. 指定要使用的指标类型，可以是 **Utilization** 或 **AverageValue**。
3. 指定触发扩展的值。必须指定为带引号的字符串值。

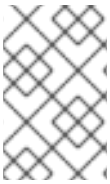
- 在使用 **Utilization** 时，target 值是所有相关 pod 中资源指标的平均值，以 pod 资源请求的值的百分比表示。
- 使用 **AverageValue** 时，target 值是所有相关 Pod 的指标平均值。

4 指定缩减时的最小副本数量。对于 CPU 触发器，输入值 **1** 或更高的值，因为如果您只使用 CPU 指标，HPA 无法缩减为零。

3.4.3. 了解内存触发器

您可以根据内存指标扩展 pod。此触发器使用集群指标作为指标的源。

自定义指标自动扩展扩展与对象关联的 pod，以维护您指定的平均内存用量。自动缩放器会增加和减少最小和最大数量之间的副本数量，以维护所有 pod 的指定内存使用率。内存触发器考虑整个 pod 的内存使用率。如果 pod 有多个容器，则内存使用率是所有容器的总和。



注意

- 此触发器不能与 **ScaledJob** 自定义资源一起使用。
- 当使用内存触发器扩展对象时，对象不会扩展到 **0**，即使您使用多个触发器。

使用内存目标扩展对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: memory-scaledobject
  namespace: my-namespace
spec:
# ...
  triggers:
  - type: memory 1
    metricType: Utilization 2
    metadata:
      value: '60' 3
      containerName: api 4
```

1 将 memory 指定为触发器类型。

2 指定要使用的指标类型，可以是 **Utilization** 或 **AverageValue**。

3 指定触发扩展的值。必须指定为带引号的字符串值。

- 在使用 **Utilization** 时，target 值是所有相关 pod 中资源指标的平均值，以 pod 资源请求的值的百分比表示。
- 使用 **AverageValue** 时，target 值是所有相关 Pod 的指标平均值。

4 可选：根据该容器的内存使用率，而不是整个 pod，指定要缩放的独立容器。在本例中，只有名为 **api** 的容器才会扩展。

3.4.4. 了解 Kafka 触发器

您可以根据 Apache Kafka 主题或支持 Kafka 协议的其他服务扩展 pod。自定义指标自动扩展不会缩放 Kafka 分区数量，除非在扩展的对象或扩展任务中将 **allowIdleConsumers** 参数设置为 **true**。

注意

如果消费者组数量超过主题中的分区数量，则额外的消费者组处于闲置状态。要避免这种情况，默认情况下副本数不会超过：

- 如果指定了主题，则主题上的分区数量
- 如果没有指定主题，则消费者组中的所有主题的分区数量
- 在扩展对象或扩展作业 CR 中指定的 **maxReplicaCount**

您可以使用 **allowIdleConsumers** 参数禁用这些默认行为。

使用 Kafka 目标扩展对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: kafka-scaledobject
  namespace: my-namespace
spec:
# ...
  triggers:
  - type: kafka (1)
    metadata:
      topic: my-topic (2)
      bootstrapServers: my-cluster-kafka-bootstrap.openshift-operators.svc:9092 (3)
      consumerGroup: my-group (4)
      lagThreshold: '10' (5)
      activationLagThreshold: '5' (6)
      offsetResetPolicy: latest (7)
      allowIdleConsumers: true (8)
      scaleToZeroOnInvalidOffset: false (9)
      excludePersistentLag: false (10)
      version: '1.0.0' (11)
      partitionLimitation: '1,2,10-20,31' (12)
```

(1) 指定 Kafka 作为触发器类型。

(2) 指定 Kafka 在处理偏移滞后的 Kafka 主题的名称。

(3) 指定要连接的 Kafka 代理的逗号分隔列表。

(4) 指定用于检查主题上的偏移以及处理相关滞后的 Kafka 消费者组的名称。

(5) 可选：指定触发扩展的平均目标值。必须指定为带引号的字符串值。默认值为 **5**。

(6) 可选：指定激活阶段的目标值。必须指定为带引号的字符串值。

7 可选：为 Kafka 使用者指定 Kafka 偏移重置策略。可用值包括：**latest** 和 **earliest**。默认为 **latest**。

8 可选：指定 Kafka 副本数是否可以超过主题中的分区数量。

- 如果为 **true**，则 Kafka 副本数可能会超过主题上的分区数量。这允许闲置 Kafka 用户。
- 如果为 **false**，则 Kafka 副本数不能超过主题上的分区数量。这是默认值。

9 指定当 Kafka 分区没有有效偏移时触发器的行为方式。

- 如果为 **true**，则该分区的用户将缩减为零。
- 如果为 **false**，则 scaler 为该分区保留单个消费者。这是默认值。

10 可选：指定触发器是否为当前偏移与之前轮询周期的当前偏移量相同或排除分区滞后。

- 如果为 **true**，则扩展程序会排除这些分区中的分区滞后。
- 如果为 **false**，则触发器在所有分区中包含所有消费者滞后。这是默认值。

11 可选：指定 Kafka 代理的版本。必须指定为带引号的字符串值。默认值为 **1.0.0**。

12 可选：指定一个以逗号分隔的分区 ID 列表来限制缩放。如果设置，则仅考虑计算滞后列出的 ID。必须指定为带引号的字符串值。默认为考虑所有分区。

3.5. 了解自定义指标自动扩展触发器身份验证

触发器身份验证允许您在扩展对象或可供关联容器使用的扩展作业中包含身份验证信息。您可以使用触发器身份验证来传递 OpenShift Container Platform secret、平台原生 Pod 验证机制、环境变量等。

您可以在与您要缩放的对象相同的命名空间中定义一个 **TriggerAuthentication** 对象。该触发器身份验证只能由该命名空间中的对象使用。

另外，要在多个命名空间中对象间共享凭证，您可以创建一个可在所有命名空间中使用的 **ClusterTriggerAuthentication** 对象。

触发验证和集群触发器身份验证使用相同的配置。但是，集群触发器身份验证需要在扩展对象的验证引用中有一个额外的 **kind** 参数。

使用 secret 的触发器验证示例

```
kind: TriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata:
  name: secret-triggerauthentication
  namespace: my-namespace 1
spec:
  secretTargetRef: 2
  - parameter: user-name 3
    name: my-secret 4
    key: USER_NAME 5
  - parameter: password
    name: my-secret
    key: USER_PASSWORD
```

1. 指定您要缩放的对象的命名空间。
2. 指定此触发器身份验证使用 secret 进行授权。
3. 使用 secret 指定提供的身份验证参数。
4. 指定要使用的 secret 的名称。
5. 指定 secret 中与指定参数一起使用的密钥。

使用 secret 的集群触发器身份验证示例

```
kind: ClusterTriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata: 1
  name: secret-cluster-triggerauthentication
spec:
  secretTargetRef: 2
  - parameter: user-name 3
    name: secret-name 4
    key: USER_NAME 5
  - parameter: user-password
    name: secret-name
    key: USER_PASSWORD
```

1. 请注意，没有命名空间用于集群触发器身份验证。
2. 指定此触发器身份验证使用 secret 进行授权。
3. 使用 secret 指定提供的身份验证参数。
4. 指定要使用的 secret 的名称。
5. 指定 secret 中与指定参数一起使用的密钥。

使用令牌进行触发器身份验证示例

```
kind: TriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata:
  name: token-triggerauthentication
  namespace: my-namespace 1
spec:
  secretTargetRef: 2
  - parameter: bearerToken 3
    name: my-token-2vzfq 4
    key: token 5
  - parameter: ca
    name: my-token-2vzfq
    key: ca.crt
```

1. 指定您要缩放的对象的命名空间。

2 指定此触发器身份验证使用 secret 进行授权。

3 使用令牌指定要提供的身份验证参数。

4 指定要使用的令牌名称。

5 指定令牌中用于指定参数的密钥。

使用环境变量的触发器身份验证示例

```
kind: TriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata:
  name: env-var-triggerauthentication
  namespace: my-namespace 1
spec:
  env: 2
  - parameter: access_key 3
    name: ACCESS_KEY 4
    containerName: my-container 5
```

1 指定您要缩放的对象的命名空间。

2 指定此触发器身份验证使用环境变量进行授权。

3 指定要使用此变量设置的参数。

4 指定环境变量的名称。

5 可选：指定需要身份验证的容器。容器必须与扩展对象中的 **scaleTargetRef** 引用的资源相同。

使用 pod 验证供应商的触发器身份验证示例

```
kind: TriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata:
  name: pod-id-triggerauthentication
  namespace: my-namespace 1
spec:
  podIdentity: 2
    provider: aws-eks 3
```

1 指定您要缩放的对象的命名空间。

2 指定此触发器身份验证使用平台原生 Pod 验证方法进行授权。

3 指定 pod 身份。支持的值为 **none**,**azure**,**gcp**,**aws-eks**, 或 **aws-kiam**。默认为 **none**。

其他资源

- 如需有关 OpenShift Container Platform secret 的信息，请参阅向 pod 提供敏感数据。

3.5.1. 使用触发器身份验证

您可以使用触发器验证和集群触发器身份验证，方法是使用自定义资源来创建身份验证，然后添加对扩展对象或扩展任务的引用。

先决条件

- 必须安装 Custom Metrics Autoscaler Operator。
- 如果使用 secret，**Secret** 对象必须存在，例如：

  secret 示例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  user-name: <base64_USER_NAME>
  password: <base64_USER_PASSWORD>
```

流程

1. 创建 **TriggerAuthentication** 或 **ClusterTriggerAuthentication** 对象。
   a. 创建定义对象的 YAML 文件：

   使用 secret 的触发器验证示例

```
kind: TriggerAuthentication
apiVersion: keda.sh/v1alpha1
metadata:
  name: prom-triggerauthentication
  namespace: my-namespace
spec:
  secretTargetRef:
  - parameter: user-name
    name: my-secret
    key: USER_NAME
  - parameter: password
    name: my-secret
    key: USER_PASSWORD
```

   b. 创建 **TriggerAuthentication** 对象：

```
$ oc create -f <filename>.yaml
```

2. 创建或编辑使用触发器身份验证的 **ScaledObject** YAML 文件：
   a. 运行以下命令，创建定义对象的 YAML 文件：

   使用触发器身份验证的扩展对象示例

```
apiVersion: keda.sh/v1alpha1
```

```
kind: ScaledObject
metadata:
  name: scaledobject
  namespace: my-namespace
spec:
  scaleTargetRef:
    name: example-deployment
  maxReplicaCount: 100
  minReplicaCount: 0
  pollingInterval: 30
  triggers:
  - type: prometheus
    metadata:
      serverAddress: https://thanos-querier.openshift-monitoring.svc.cluster.local:9092
      namespace: kedatest # replace <NAMESPACE>
      metricName: http_requests_total
      threshold: '5'
      query: sum(rate(http_requests_total{job="test-app"}[1m]))
      authModes: "basic"
    authenticationRef:
      name: prom-triggerauthentication 1
      kind: TriggerAuthentication 2
```

1 指定触发器身份验证对象的名称。

2 指定 **TriggerAuthentication**。**TriggerAuthentication** 是默认值。

使用集群触发器身份验证的扩展对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: scaledobject
  namespace: my-namespace
spec:
  scaleTargetRef:
    name: example-deployment
  maxReplicaCount: 100
  minReplicaCount: 0
  pollingInterval: 30
  triggers:
  - type: prometheus
    metadata:
      serverAddress: https://thanos-querier.openshift-monitoring.svc.cluster.local:9092
      namespace: kedatest # replace <NAMESPACE>
      metricName: http_requests_total
      threshold: '5'
      query: sum(rate(http_requests_total{job="test-app"}[1m]))
      authModes: "basic"
    authenticationRef:
      name: prom-cluster-triggerauthentication 1
      kind: ClusterTriggerAuthentication 2
```

1 指定触发器身份验证对象的名称。

2 指定 **ClusterTriggerAuthentication**。

b. 运行以下命令来创建扩展的对象：

```
$ oc apply -f <filename>
```

3.6. 暂停扩展对象的自定义指标自动扩展

您可以根据需要暂停并重启工作负载的自动扩展。

例如，您可能想要在执行集群维护前暂停自动扩展，或通过删除非传输工作负载来避免资源不足。

3.6.1. 暂停自定义指标自动扩展

您可以通过将 **autoscaling.keda.sh/paused-replicas** 注解添加到扩展对象的自定义指标自动扩展中来暂停扩展对象的自动扩展。自定义指标自动扩展将该工作负载的副本扩展到指定的值，并暂停自动扩展，直到注解被删除为止。

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "4"
# ...
```

流程

1. 使用以下命令编辑工作负载的 **ScaledObject** CR：

```
$ oc edit ScaledObject scaledobject
```

2. 使用任何值添加 **autoscaling.keda.sh/paused-replicas** 注解：

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "4" 1
  creationTimestamp: "2023-02-08T14:41:01Z"
  generation: 1
  name: scaledobject
  namespace: my-project
  resourceVersion: '65729'
  uid: f5aec682-acdf-4232-a783-58b5b82f5dd0
```

1 指定自定义 Metrics Autoscaler Operator 将副本扩展到指定的值，并停止自动扩展。

3.6.2. 为扩展的对象重启自定义指标自动扩展

您可以通过删除该 **ScaledObject** 的 **autoscaling.keda.sh/paused-replicas** 注解来重启暂停的自定义指标自动扩展。

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "4"
# ...
```

流程

1. 使用以下命令编辑工作负载的 **ScaledObject** CR：

```
$ oc edit ScaledObject scaledobject
```

2. 删除 **autoscaling.keda.sh/paused-replicas** 注解。

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "4" (1)
  creationTimestamp: "2023-02-08T14:41:01Z"
  generation: 1
  name: scaledobject
  namespace: my-project
  resourceVersion: '65729'
  uid: f5aec682-acdf-4232-a783-58b5b82f5dd0
```

(1) 删除此注解以重启暂停的自定义指标自动扩展。

3.7. 收集审计日志

您可以收集审计日志，它们是与安全相关的按时间排序的记录，记录各个用户、管理员或其他系统组件影响系统的一系列活动。

例如，审计日志可帮助您了解自动扩展请求来自哪里。当后端因为用户应用程序发出的请求造成过载时，这个信息非常重要，您需要确定哪个是有问题的应用程序。

3.7.1. 配置审计日志记录

您可以通过编辑 **KedaController** 自定义资源来为自定义 Metrics Autoscaler Operator 配置审计。日志通过 **KedaController** CR 中的持久性卷声明发送到卷的审计日志文件。

先决条件

- 必须安装 Custom Metrics Autoscaler Operator。

流程

1. 编辑 **KedaController** 自定义资源以添加 **auditConfig** 小节：

```
kind: KedaController
apiVersion: keda.sh/v1alpha1
```

```
metadata:
  name: keda
  namespace: openshift-keda
spec:
# ...
  metricsServer:
# ...
    auditConfig:
      logFormat: "json" 1
      logOutputVolumeClaim: "pvc-audit-log" 2
      policy:
        rules: 3
        - level: Metadata
        omitStages: "RequestReceived" 4
        omitManagedFields: false 5
      lifetime: 6
        maxAge: "2"
        maxBackup: "1"
        maxSize: "50"
```

1. 指定审计日志的输出格式，可以是 **legacy** 或 **json**。
2. 指定用于存储日志数据的现有持久性卷声明。所有来自 API 服务器的请求都会记录到此持久性卷声明。如果将此字段留空，日志数据将发送到 stdout。
3. 指定应记录哪些事件及其应包含哪些数据：
   - **None** ：不记录事件。
   - **Metadata** ：仅记录请求的元数据，如用户、时间戳等。不要记录请求文本和响应文本。这是默认值。
   - **Request** ：仅记录元数据和请求文本，而不记录响应文本。这个选项不适用于非资源请求。
   - **RequestResponse** ：日志事件元数据、请求文本和响应文本。这个选项不适用于非资源请求。
4. 指定没有创建事件的阶段。
5. 指定是否省略请求的 managed 字段，并从写入 API 审计日志的响应正文，可以是 **true** 来省略字段，或 **false** 包含字段。
6. 指定审计日志的大小和生命周期。
   - **MaxAge** ：根据文件名中编码的时间戳，保留审计日志文件的最大天数。
   - **maxBackup** ：要保留的审计日志文件的最大数量。设置为 **0** 以保留所有审计日志文件。
   - **maxsize** ：在轮转审计日志文件前以 MB 为单位的最大大小。

验证

1. 直接查看审计日志文件：

a. 获取 **keda-metrics-apiserver the pod** 的名称：

```
oc get pod -n openshift-keda
```

输出示例

```
NAME                                              READY   STATUS    RESTARTS   AGE
custom-metrics-autoscaler-operator-5cb44cd75d-9v4lv   1/1     Running   0          8m20s
keda-metrics-apiserver-65c7cc44fd-rrl4r               1/1     Running   0          2m55s
keda-operator-776cbb6768-zpj5b                        1/1     Running   0          2m55s
```

b. 使用类似如下的命令查看日志数据：

```
$ oc logs keda-metrics-apiserver-<hash>|grep -i metadata 1
```

1 可选： 您可以使用 **grep** 命令指定要显示的日志级别： **Metadata**、**Request**、**RequestResponse**。

例如：

```
$ oc logs keda-metrics-apiserver-65c7cc44fd-rrl4r|grep -i metadata
```

输出示例

```
 ...
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"4c81d41b-
3dab-4675-90ce-
20b87ce24013","stage":"ResponseComplete","requestURI":"/healthz","verb":"get","user":
{"username":"system:anonymous","groups":["system:unauthenticated"]},"sourceIPs":
["10.131.0.1"],"userAgent":"kube-probe/1.28","responseStatus":{"metadata":
{},"code":200},"requestReceivedTimestamp":"2023-02-
16T13:00:03.554567Z","stageTimestamp":"2023-02-
16T13:00:03.555032Z","annotations":
{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":""}}
 ...
```

2. 另外，您可以查看特定的日志：

a. 使用类似如下的命令登录到 **keda-metrics-apiserver the** pod:

```
$ oc rsh pod/keda-metrics-apiserver-<hash> -n openshift-keda
```

例如：

```
$ oc rsh pod/keda-metrics-apiserver-65c7cc44fd-rrl4r -n openshift-keda
```

b. 进入 **/var/audit-policy/** 目录：

```
sh-4.4$ cd /var/audit-policy/
```

c. 列出可用的日志：

```
sh-4.4$ ls
```

输出示例

```
log-2023.02.17-14:50  policy.yaml
```

d. 根据需要查看日志：

```
sh-4.4$ cat <log_name>/<pvc_name>|grep -i <log_level> 1
```

1 可选：您可以使用 **grep** 命令指定要显示的日志级别：**Metadata**、**Request**、**RequestResponse**。

例如：

```
sh-4.4$ cat log-2023.02.17-14:50/pvc-audit-log|grep -i Request
```

输出示例

```
 ...
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Request","auditID":"63e7f68c-04ec-4f4d-8749-bf1656572a41","stage":"ResponseComplete","requestURI":"/openapi/v2","verb":"get","user":{"username":"system:aggregator","groups":["system:authenticated"]},"sourceIPs":["10.128.0.1"],"responseStatus":{"metadata":{},"code":304},"requestReceivedTimestamp":"2023-02-17T13:12:55.035478Z","stageTimestamp":"2023-02-17T13:12:55.038346Z","annotations":{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"system:discovery\" of ClusterRole \"system:discovery\" to Group \"system:authenticated\""}}
 ...
```

3.8. 收集调试数据

在提交问题单时同时提供您的集群信息，可以帮助红帽支持为您进行排除故障。

要帮助排除您的问题，请提供以下信息：

- 使用 **must-gather** 工具收集的数据。
- 唯一的集群 ID。

您可以使用 **must-gather** 工具来收集有关自定义 Metrics Autoscaler Operator 及其组件的数据，包括以下项目：

- **openshift-keda** 命名空间及其子对象。
- Custom Metric Autoscaler Operator 安装对象。
- Custom Metric Autoscaler Operator CRD 对象。

3.8.1. 收集调试数据

以下命令为自定义 Metrics Autoscaler Operator 运行 **must-gather** 工具：

```
$ oc adm must-gather --image="$(oc get packagemanifests openshift-custom-metrics-autoscaler-operator \
-n openshift-marketplace \
-o jsonpath='{.status.channels[?(@.name=="stable")].currentCSVDesc.annotations.containerImage}')"
```

注意

标准 OpenShift Container Platform **must-gather** 命令 **oc adm must-gather** 将不会收集自定义 Metrics Autoscaler Operator 数据。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 已安装 OpenShift Container Platform CLI (**oc**)。

流程

1. 进入要存储 **must-gather** 数据的目录。

注意

如果集群使用受限网络，则需要执行额外的步骤。如果您镜像的容器镜像仓库有一个信任的 CA，您必须首先将这个信任的 CA 添加到集群中。对于受限网络中的所有集群，您必须运行以下命令来导入默认的 **must-gather** 镜像作为镜像流。

```
$ oc import-image is/must-gather -n openshift
```

2. 执行以下之一：

- 要只获取自定义 Metrics Autoscaler Operator **must-gather** 数据，请使用以下命令：

```
$ oc adm must-gather --image="$(oc get packagemanifests openshift-custom-metrics-autoscaler-operator \
-n openshift-marketplace \
-o jsonpath='{.status.channels[?(@.name=="stable")].currentCSVDesc.annotations.containerImage}')"
```

must-gather 命令的自定义镜像直接从 Operator 软件包清单中拉取，以便它可用于提供 Custom Metric Autoscaler Operator 的任何集群。

- 除了 Custom Metric Autoscaler Operator 信息外，要收集默认的 **must-gather** 数据：

a. 使用以下命令获取自定义 Metrics Autoscaler Operator 镜像并将其设置为环境变量：

```
$ IMAGE="$(oc get packagemanifests openshift-custom-metrics-autoscaler-operator \
  -n openshift-marketplace \
```

```
  -o jsonpath='{.status.channels[?
(@.name=="stable")].currentCSVDesc.annotations.containerImage}')"
```

b. 使用带有自定义 Metrics Autoscaler Operator 镜像的 **oc adm must-gather** ：

```
$ oc adm must-gather --image-stream=openshift/must-gather --image=${IMAGE}
```

例 3.1. Custom Metric Autoscaler 的 must-gather 输出示例：

```
└── openshift-keda
    ├── apps
    │   ├── daemonsets.yaml
    │   ├── deployments.yaml
    │   ├── replicasets.yaml
    │   └── statefulsets.yaml
    ├── apps.openshift.io
    │   └── deploymentconfigs.yaml
    ├── autoscaling
    │   └── horizontalpodautoscalers.yaml
    ├── batch
    │   ├── cronjobs.yaml
    │   └── jobs.yaml
    ├── build.openshift.io
    │   ├── buildconfigs.yaml
    │   └── builds.yaml
    ├── core
    │   ├── configmaps.yaml
    │   ├── endpoints.yaml
    │   ├── events.yaml
    │   ├── persistentvolumeclaims.yaml
    │   ├── pods.yaml
    │   ├── replicationcontrollers.yaml
    │   ├── secrets.yaml
    │   └── services.yaml
    ├── discovery.k8s.io
    │   └── endpointslices.yaml
    ├── image.openshift.io
    │   └── imagestreams.yaml
    ├── k8s.ovn.org
    │   ├── egressfirewalls.yaml
    │   └── egressqoses.yaml
    ├── keda.sh
    │   ├── kedacontrollers
    │   │   └── keda.yaml
    │   ├── scaledobjects
    │   │   └── example-scaledobject.yaml
    │   └── triggerauthentications
    │       └── example-triggerauthentication.yaml
    ├── monitoring.coreos.com
    │   └── servicemonitors.yaml
    ├── networking.k8s.io
    │   └── networkpolicies.yaml
    ├── openshift-keda.yaml
    ├── pods
    │   ├── custom-metrics-autoscaler-operator-58bd9f458-ptgwx
```

```
        │   ├── custom-metrics-autoscaler-operator
        │   │   └── custom-metrics-autoscaler-operator
        │   │       └── logs
        │   │           ├── current.log
        │   │           ├── previous.insecure.log
        │   │           └── previous.log
        │   └── custom-metrics-autoscaler-operator-58bd9f458-ptgwx.yaml
        ├── custom-metrics-autoscaler-operator-58bd9f458-thbsh
        │   └── custom-metrics-autoscaler-operator
        │       └── custom-metrics-autoscaler-operator
        │           └── logs
        ├── keda-metrics-apiserver-65c7cc44fd-6wq4g
        │   ├── keda-metrics-apiserver
        │   │   └── keda-metrics-apiserver
        │   │       └── logs
        │   │           ├── current.log
        │   │           ├── previous.insecure.log
        │   │           └── previous.log
        │   └── keda-metrics-apiserver-65c7cc44fd-6wq4g.yaml
        └── keda-operator-776cbb6768-fb6m5
            ├── keda-operator
            │   └── keda-operator
            │       └── logs
            │           ├── current.log
            │           ├── previous.insecure.log
            │           └── previous.log
            └── keda-operator-776cbb6768-fb6m5.yaml
    ├── policy
    │   └── poddisruptionbudgets.yaml
    └── route.openshift.io
        └── routes.yaml
```

3. 从工作目录中创建的 **must-gather** 目录创建一个压缩文件。例如，在使用 Linux 操作系统的计算机上运行以下命令：

```
$ tar cvaf must-gather.tar.gz must-gather.local.5421342344627712289/ 1
```

1 将 **must-gather-local.5421342344627712289/** 替换为实际目录名称。

4. 在红帽客户门户中为您的问题单附上压缩文件。

3.9. 查看 OPERATOR 指标

Custom Metrics Autoscaler Operator 会公开从集群监控组件中提取的可随时使用的指标。您可以使用 Prometheus Query Language (PromQL) 来分析和诊断问题来查询指标。控制器 pod 重启时会重置所有指标。

3.9.1. 访问性能指标

您可以使用 OpenShift Container Platform Web 控制台访问指标并运行查询。

流程

1. 在 OpenShift Container Platform Web 控制台中选择 **Administrator** 视角。
2. 选择 **Observe → Metrics**。
3. 要创建自定义查询，请将 PromQL 查询添加到 **Expression** 字段中。
4. 要添加多个查询，选择 **Add Query**。

3.9.1.1. 提供的 Operator 指标

Custom Metrics Autoscaler Operator 会公开以下指标，您可以使用 OpenShift Container Platform Web 控制台查看这些指标。

表 3.1. 自定义 Metric Autoscaler Operator 指标

指标名称	描述
keda_scaler_activity	特定的 scaler 是活跃的还是不活跃的。值 **1** 表示 scaler 处于活跃状态；**0** 表示 scaler 不活跃。
keda_scaler_metrics_value	每个 scaler 的指标的当前值，由计算目标平均值中的 Horizontal Pod Autoscaler (HPA) 使用。
keda_scaler_metrics_latency	从每个 scaler 检索当前指标的延迟。
keda_scaler_errors	每个 scaler 发生的错误数量。
keda_scaler_errors_total	所有 scaler 遇到的错误总数。
keda_scaled_object_errors	每个扩展的对象发生的错误数量。
keda_resource_totals	每个命名空间中的自定义 Metrics Autoscaler 自定义资源总数，每种自定义资源类型。
keda_trigger_totals	根据触发器类型的触发器总数。

自定义 Metrics Autoscaler Admission Webhook 指标

自定义 Metrics Autoscaler Admission Webhook 也会公开以下 Prometheus 指标。

指标名称	描述
keda_scaled_object_validation_total	扩展对象验证的数量。
keda_scaled_object_validation_errors	验证错误的数量。

3.10. 了解如何添加自定义指标自动扩展

要添加自定义指标自动扩展，请为部署、有状态集或自定义资源创建 **ScaledObject** 自定义资源。为作业创建 **ScaledJob** 自定义资源。

您只能为每个您要扩展的工作负载创建一个扩展对象。另外，您不能在同一工作负载中使用扩展的对象和 pod 横向自动扩展(HPA)。

3.10.1. 在工作负载中添加自定义指标自动扩展

您可以为 **Deployment**、**StatefulSet** 或 **custom resource** 对象创建的工作负载创建自定义指标自动扩展。

先决条件

- 必须安装 Custom Metrics Autoscaler Operator。
- 如果您使用自定义指标自动扩展来根据 CPU 或内存进行扩展：
  - 您的集群管理员必须已配置了集群指标。您可以使用 **oc describe PodMetrics <pod-name>** 命令来判断是否已配置了指标。如果配置了指标，输出将类似以下示例，CPU 和 Memory 在 Usage 下显示。

```
$ oc describe PodMetrics openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
```

输出示例

```
Name:         openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Namespace:    openshift-kube-scheduler
Labels:       <none>
Annotations:  <none>
API Version:  metrics.k8s.io/v1beta1
Containers:
  Name:  wait-for-host-port
  Usage:
    Memory:  0
  Name:      scheduler
  Usage:
    Cpu:     8m
    Memory:  45440Ki
Kind:        PodMetrics
Metadata:
  Creation Timestamp:  2019-05-23T18:47:56Z
  Self Link:           /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-
scheduler/pods/openshift-kube-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:             2019-05-23T18:47:56Z
Window:                1m0s
Events:                <none>
```

  - 与您要缩放的对象关联的 pod 必须包含指定的内存和 CPU 限值。例如：

pod 规格示例

```
apiVersion: v1
```

```
kind: Pod
# ...
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      limits:
        memory: "128Mi"
        cpu: "500m"
# ...
```

流程

1. 创建一个类似如下的 YAML 文件：只有名称 **<2>**, 对象名称 **<4>**, 和对象类型 **<5>** 是必需的。

缩放对象示例

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  annotations:
    autoscaling.keda.sh/paused-replicas: "0" 1
  name: scaledobject 2
  namespace: my-namespace
spec:
  scaleTargetRef:
    apiVersion: apps/v1 3
    name: example-deployment 4
    kind: Deployment 5
    envSourceContainerName: .spec.template.spec.containers[0] 6
  cooldownPeriod:  200 7
  maxReplicaCount: 100 8
  minReplicaCount: 0 9
  metricsServer: 10
    auditConfig:
      logFormat: "json"
      logOutputVolumeClaim: "persistentVolumeClaimName"
      policy:
        rules:
        - level: Metadata
        omitStages: "RequestReceived"
        omitManagedFields: false
      lifetime:
        maxAge: "2"
        maxBackup: "1"
        maxSize: "50"
  fallback: 11
    failureThreshold: 3
    replicas: 6
  pollingInterval: 30 12
  advanced:
    restoreToOriginalReplicaCount: false 13
```

```
  horizontalPodAutoscalerConfig:
    name: keda-hpa-scale-down (14)
    behavior: (15)
      scaleDown:
        stabilizationWindowSeconds: 300
        policies:
        - type: Percent
          value: 100
          periodSeconds: 15
  triggers:
  - type: prometheus (16)
    metadata:
      serverAddress: https://thanos-querier.openshift-monitoring.svc.cluster.local:9092
      namespace: kedatest
      metricName: http_requests_total
      threshold: '5'
      query: sum(rate(http_requests_total{job="test-app"}[1m]))
      authModes: basic
    authenticationRef: (17)
      name: prom-triggerauthentication
      kind: TriggerAuthentication
```

1. 可选：指定自定义 Metrics Autoscaler Operator 将副本扩展到指定的值和停止自动扩展，如 "Pausing the custom metrics autoscaler for a workload" 部分所述。
2. 指定此自定义指标自动扩展的名称。
3. 可选：指定目标资源的 API 版本。默认为 **apps/v1**。
4. 指定要缩放的对象名称。
5. 指定 **kind** 为 **Deployment**, **StatefulSet** 或 **CustomResource**。
6. 可选：指定目标资源中的容器的名称，其中的自定义自动扩展器获取包含 secret 的环境变量等。默认为 **.spec.template.spec.containers[0]**。
7. 可选。指定一个在最后的触发器报告后等待的时间（以秒为单位），在经过这个时间后才会将部署缩减为 **0**（如果 **minReplicaCount** 设置为 **0**）。默认值为 **300**。
8. 可选：指定扩展时的最大副本数量。默认值为 **100**。
9. 可选：指定缩减时的最小副本数量。
10. 可选：指定审计日志的参数。如"配置审计日志记录"部分中所述。
11. 可选：指定在扩展程序无法从源中获取由 **failureThreshold** 参数定义的次数时回退到的副本数。有关回退行为的更多信息，请参阅 KEDA 文档。
12. 可选：指定检查每个触发器的时间间隔（以秒为单位）。默认值为 **30**。
13. 可选：指定是否在删除扩展对象后将目标资源扩展为原始副本数。默认为 **false**，这会在删除扩展对象时保留副本数。
14. 可选：指定 pod 横向自动扩展的名称。默认为 **keda-hpa-{scaled-object-name}**。
15. 可选：指定一个扩展策略来控制用来扩展或缩减 pod 的速度，如"扩展策略"部分中所述。

16 指定用作扩展基础的触发器，如"识别自定义指标自动扩展触发器"部分中所述。本例使用 OpenShift Container Platform 监控。

17 可选：指定触发器身份验证或集群触发器身份验证。如需更多信息，请参阅*附加资源*部分中的 *了解自定义指标自动扩展触发器身份验证*。

- 输入 **TriggerAuthentication** 来使用触发器身份验证。这是默认值。
- 输入 **ClusterTriggerAuthentication** 来使用集群触发器身份验证。

2. 运行以下命令来创建自定义指标自动扩展：

```
$ oc create -f <filename>.yaml
```

验证

- 查看命令输出，以验证是否已创建自定义指标自动扩展：

```
$ oc get scaledobject <scaled_object_name>
```

输出示例

```
NAME              SCALETARGETKIND      SCALETARGETNAME      MIN   MAX   TRIGGERS
AUTHENTICATION              READY   ACTIVE   FALLBACK   AGE
scaledobject   apps/v1.Deployment   example-deployment   0   50   prometheus   prom-
triggerauthentication   True   True   True   17s
```

请注意输出中的以下字段：

- **TRIGGERS** ：指示正在使用的触发器或缩放器。
- **AUTHENTICATION** ：指示所使用的任何触发器身份验证的名称。
- **READY** ：指示扩展对象是否准备好启动缩放：
  - 如果为 **True**，则扩展的对象已就绪。
  - 如果 **False**，由于您创建的对象中的一个或多个对象有问题，扩展的对象将不可用。
- **ACTIVE** ：指示扩展是否发生：
  - 如果为 **True**，则会进行缩放。
  - 如果 **False**，则不会发生缩放，因为您创建的一个或多个对象中没有指标或多个问题。
- **FALLBACK** ：指示自定义指标自动扩展是否能够从源获取指标
  - 如果 **False**，自定义指标自动扩展器会获取指标。
  - 如果为 **True**，自定义指标自动扩展会获取指标，因为您创建的一个或多个对象中没有指标或多个问题。

3.10.2. 在作业中添加自定义指标自动扩展

您可以为任何**作业**对象创建自定义指标自动扩展。

重要

使用扩展作业进行扩展只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

先决条件

- 必须安装 Custom Metrics Autoscaler Operator。

流程

1. 创建一个类似以下示例的 YAML 文件：

```
kind: ScaledJob
apiVersion: keda.sh/v1alpha1
metadata:
  name: scaledjob
  namespace: my-namespace
spec:
  failedJobsHistoryLimit: 5
  jobTargetRef:
    activeDeadlineSeconds: 600 (1)
    backoffLimit: 6 (2)
    parallelism: 1 (3)
    completions: 1 (4)
    template: (5)
      metadata:
        name: pi
      spec:
        containers:
        - name: pi
          image: perl
          command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
  maxReplicaCount: 100 (6)
  pollingInterval: 30 (7)
  successfulJobsHistoryLimit: 5 (8)
  failedJobsHistoryLimit: 5 (9)
  envSourceContainerName: (10)
  rolloutStrategy: gradual (11)
  scalingStrategy: (12)
    strategy: "custom"
    customScalingQueueLengthDeduction: 1
    customScalingRunningJobPercentage: "0.5"
    pendingPodConditions:
      - "Ready"
      - "PodScheduled"
      - "AnyOtherCustomPodCondition"
    multipleScalersCalculation : "max"
  triggers:
  - type: prometheus (13)
    metadata:
```

```
    serverAddress: https://thanos-querier.openshift-monitoring.svc.cluster.local:9092
    namespace: kedatest
    metricName: http_requests_total
    threshold: '5'
    query: sum(rate(http_requests_total{job="test-app"}[1m]))
    authModes: "bearer"
  authenticationRef: 14
    name: prom-cluster-triggerauthentication
```

1. 指定作业可以运行的最长持续时间。
2. 指定作业的重试次数。默认值为 **6**。
3. 可选：指定作业应并行运行多少个 pod 副本；默认为 **1**。
   - 对于非并行作业，请保留未设置。如果未设置，则默认值为 **1**。
4. 可选：指定标记作业完成需要成功完成多少个 pod。
   - 对于非并行作业，请保留未设置。如果未设置，则默认值为 **1**。
   - 对于具有固定完成计数的并行作业，请指定完成数。
   - 对于带有工作队列的并行作业，请保留 unset。当取消设置默认值时，默认值为 **parallelism** 参数的值。
5. 指定控制器创建的 pod 模板。
6. 可选：指定扩展时的最大副本数量。默认值为 **100**。
7. 可选：指定检查每个触发器的时间间隔（以秒为单位）。默认值为 **30**。
8. 可选：指定成功完成作业的数量。默认值为 **100**。
9. 可选：指定应保留多少个失败作业。默认值为 **100**。
10. 可选：指定目标资源中的容器的名称，其中的自定义自动扩展器获取包含 secret 的环境变量等。默认为 **.spec.template.spec.containers[0]**。
11. 可选：指定在更新扩展作业时是否被终止现有作业：
    - **default** ：如果关联的扩展作业被更新，则自动扩展器会终止一个现有作业。自动扩展会使用最新的 specs 重新创建作业。
    - **gradual** ：如果关联的扩展作业被更新，则自动扩展不会终止现有的作业。自动缩放器使用最新的 specs 创建新作业。
12. 可选：指定一个扩展策略： **default**、**custom** 或 **accurate**。默认为 **default**。如需更多信息，请参阅下面的"添加资源"部分中的链接。
13. 指定用作扩展基础的触发器，如"识别自定义指标自动扩展触发器"部分中所述。
14. 可选：指定触发器身份验证或集群触发器身份验证。如需更多信息，请参阅*附加资源*部分中的 *了解自定义指标自动扩展触发器身份验证*。
    - 输入 **TriggerAuthentication** 来使用触发器身份验证。这是默认值。

- 输入 **ClusterTriggerAuthentication** 来使用集群触发器身份验证。

2. 运行以下命令来创建自定义指标自动扩展：

```
$ oc create -f <filename>.yaml
```

验证

- 查看命令输出，以验证是否已创建自定义指标自动扩展：

```
$ oc get scaledjob <scaled_job_name>
```

输出示例

```
NAME      MAX   TRIGGERS    AUTHENTICATION             READY   ACTIVE    AGE
scaledjob 100   prometheus  prom-triggerauthentication True    True      8s
```

请注意输出中的以下字段：

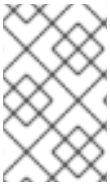
- **TRIGGERS** ：指示正在使用的触发器或缩放器。
- **AUTHENTICATION** ：指示所使用的任何触发器身份验证的名称。
- **READY** ：指示扩展对象是否准备好启动缩放：
  - 如果为 **True**，则扩展的对象已就绪。
  - 如果 **False**，由于您创建的对象中的一个或多个对象有问题，扩展的对象将不可用。
- **ACTIVE** ：指示扩展是否发生：
  - 如果为 **True**，则会进行缩放。
  - 如果 **False**，则不会发生缩放，因为您创建的一个或多个对象中没有指标或多个问题。

3.10.3. 其他资源

- 了解自定义指标自动扩展触发器身份验证

3.11. 删除自定义 METRICS AUTOSCALER OPERATOR

您可以从 OpenShift Container Platform 集群中删除自定义指标自动扩展。删除自定义 Metrics Autoscaler Operator 后，删除与 Operator 相关的其他组件以避免出现潜在的问题。



注意

首先删除 **KedaController** 自定义资源(CR)。如果没有删除 **KedaController** CR，OpenShift Container Platform 会在删除 **openshift-keda** 项目时挂起。如果在删除 CR 前删除了自定义 Metrics Autoscaler Operator，您将无法删除 CR。

3.11.1. 卸载自定义 Metrics Autoscaler Operator

使用以下步骤从 OpenShift Container Platform 集群中删除自定义指标自动扩展。

先决条件

- 必须安装 Custom Metrics Autoscaler Operator。

流程

1. 在 OpenShift Container Platform web 控制台中，点击 **Operators → Installed Operators**。
2. 切换到 **openshift-keda** 项目。
3. 删除 **KedaController** 自定义资源。
   a. 找到 **CustomMetricsAutoscaler** Operator 并点 **KedaController** 选项卡。
   b. 找到自定义资源，然后点 **Delete KedaController**。
   c. 点 **Uninstall**。
4. 删除自定义 Metrics Autoscaler Operator:
   a. 点 **Operators → Installed Operators**。
   b. 找到 **CustomMetricsAutoscaler** Operator 并点 **Options** 菜单  并选择 **Uninstall Operator**。
   c. 点 **Uninstall**。
5. 可选： 使用 OpenShift CLI 删除自定义指标自动扩展组件：
   a. 删除自定义指标自动扩展 CRD：
      - **clustertriggerauthentications.keda.sh**
      - **kedacontrollers.keda.sh**
      - **scaledjobs.keda.sh**
      - **scaledobjects.keda.sh**
      - **triggerauthentications.keda.sh**

      ```
      $ oc delete crd clustertriggerauthentications.keda.sh kedacontrollers.keda.sh
      scaledjobs.keda.sh scaledobjects.keda.sh triggerauthentications.keda.sh
      ```

      删除 CRD 会删除关联的角色、集群角色和角色绑定。但是，可能存在一些必须手动删除的集群角色。
   b. 列出任何自定义指标自动扩展集群角色：

      ```
      $ oc get clusterrole | grep keda.sh
      ```

   c. 删除列出的自定义指标自动扩展集群角色。例如：

      ```
      $ oc delete clusterrole.keda.sh-v1alpha1-admin
      ```

d. 列出任何自定义指标自动扩展集群角色绑定：

```
$ oc get clusterrolebinding | grep keda.sh
```

e. 删除列出的自定义指标自动扩展集群角色绑定。例如：

```
$ oc delete clusterrolebinding.keda.sh-v1alpha1-admin
```

6. 删除自定义指标自动扩展项目：

```
$ oc delete project openshift-keda
```

7. 删除 Cluster Metric Autoscaler Operator：

```
$ oc delete operator/openshift-custom-metrics-autoscaler-operator.openshift-keda
```

第 4 章 控制节点上的 POD 放置（调度）

4.1. 使用调度程序控制 POD 放置

Pod 调度是一个内部过程，决定新 pod 如何放置到集群内的节点上。

调度程度代码具有明确隔离，会监测创建的新 pod 并确定最适合托管它们的节点。然后，它会利用主 API 为 pod 创建 pod 至节点的绑定。

默认 pod 调度

OpenShift Container Platform 附带一个满足大多数用户需求的默认调度程序。默认调度程序使用内置和自定义工具来决定最适合 pod 的调度程序。

高级 pod 调度

如果您想要更多地控制新 pod 的放置位置，可以利用 OpenShift Container Platform 高级调度功能来配置 pod，从而使 pod 能够根据要求或偏好在特定的节点上运行，或者与特定的 pod 一起运行。
您可以使用以下调度功能来控制 pod 放置：

- Scheduler 配置集
- Pod 关联性和反关联性规则
- 节点关联性
- 节点选择器
- 污点和容限
- 节点过量使用

4.1.1. 关于默认调度程序

默认的 OpenShift Container Platform pod 调度程序负责确定新 pod 放置到集群中的节点上。它从 pod 读取数据，并查找最适合配置的配置集的节点。它完全独立存在，作为独立解决方案。它不会修改 pod；它会为将 pod 绑定到特定节点的 pod 创建绑定。

4.1.1.1. 了解默认调度

现有的通用调度程序是平台默认提供的调度程序 *引擎*，它可通过三步操作来选择托管 pod 的节点：

过滤节点

根据指定的约束或要求过滤可用的节点。这可以通过使用名为 *predicates*, 或 *filters* 的过滤器函数列表在每个节点上运行来实现。

排列过滤后节点列表的优先顺序

这可以通过一系列 *priority*, 或 *scoring* 来实现，这些函数为其分配分数介于 0 到 10 之间，0 表示不适合，10 则表示最适合托管该 pod。调度程序配置还可以为每个评分功能使用简单的 *权重*（正数值）。每个评分功能提供的节点分数乘以权重（大多数分数的默认权重为 1），然后将每个节点通过为所有分数提供的分数相加。管理员可以使用这个权重属性为某些分数赋予更高的重要性。

选择最适合的节点

节点按照分数排序，系统选择分数最高的节点来托管该 pod。如果多个节点的分数相同，则随机选择其中一个。

4.1.2. 调度程序用例

在 OpenShift Container Platform 中调度的一个重要用例是支持灵活的关联性和反关联性策略。

4.1.2.1. 基础架构拓扑级别

管理员可以通过在节点上指定标签，为基础架构（节点）定义多个拓扑级别。例如，**region=r1**、**zone=z1**、**rack=s1**。

这些标签名称没有特别的含义，管理员可以自由为其基础架构级别命名，比如城市/楼宇/房间。另外，管理员可以为其基础架构拓扑定义任意数量的级别，通常三个级别比较适当（例如：**regions → zone → racks**）。管理员可以在各个级别上以任何组合指定关联性和反关联性规则。

4.1.2.2. 关联性

管理员应能够配置调度程序，在任何一个甚至多个拓扑级别上指定关联性。特定级别上的关联性指示所有属于同一服务的 pod 调度到属于同一级别的节点。这会让管理员确保对等 pod 在地理上不会过于分散，以此处理应用程序对延迟的要求。如果同一关联性组中没有节点可用于托管 pod，则不调度该 pod。

如果您需要更好地控制 pod 的调度位置，请参阅使用节点关联性规则控制节点上的 pod 放置，以及使用关联性和反关联性规则相对于其他 pod 放置 pod。

管理员可以利用这些高级调度功能，来指定 pod 可以调度到哪些节点，并且相对于其他 pod 来强制或拒绝调度。

4.1.2.3. 反关联性

管理员应能够配置调度程序，在任何一个甚至多个拓扑级别上指定反关联性。特定级别上的反关联性（或分散）指示属于同一服务的所有 pod 分散到属于该级别的不同节点上。这样可确保应用程序合理分布，以实现高可用性目的。调度程序尝试在所有适用的节点之间尽可能均匀地平衡服务 pod。

如果您需要更好地控制 pod 的调度位置，请参阅使用节点关联性规则控制节点上的 pod 放置，以及使用关联性和反关联性规则相对于其他 pod 放置 pod。

管理员可以利用这些高级调度功能，来指定 pod 可以调度到哪些节点，并且相对于其他 pod 来强制或拒绝调度。

4.2. 使用调度程序配置集调度 POD

您可以将 OpenShift Container Platform 配置为使用调度配置集将 pod 调度到集群内的节点上。

4.2.1. 关于调度程序配置集

您可以指定一个调度程序配置集来控制 pod 如何调度到节点上。

可用的调度程序配置集如下：

LowNodeUtilization

此配置集尝试在节点间平均分配 pod，以获得每个节点的资源用量较低。这个配置集提供默认的调度程序行为。

HighNodeUtilization

此配置集会尝试将尽量多的 pod 放置到尽量少的节点。这样可最小化节点数，并且每个节点的资源使用率很高。

NoScoring

这是一个低延迟配置集，通过禁用所有分数（score）插件来实现最快的调度周期。这可能会为更快的调度决策提供更好的要求。

4.2.2. 配置调度程序配置集

您可以将调度程序配置为使用调度程序配置集。

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 编辑 **Scheduler** 对象：

```
$ oc edit scheduler cluster
```

2. 指定在 **spec.profile** 字段中使用的配置集：

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
#...
spec:
  mastersSchedulable: false
  profile: HighNodeUtilization 1
#...
```

1 设置为 **LowNodeUtilization**、**HighNodeUtilization** 或 **NoScoring**。

3. 保存文件以使改变生效。

4.3. 使用关联性和反关联性规则相对于其他 POD 放置 POD

关联性是 pod 的一个属性，用于控制它们希望调度到的节点。反关联性是 pod 的一个属性，用于阻止 pod 调度到某个节点上。

在 OpenShift Container Platform 中，可以借助 *pod 关联性*和 *pod 反关联性*来根据其他 pod 上的键/值标签限制 pod 有资格调度到哪些节点。

4.3.1. 了解 pod 关联性

您可以借助 *pod 关联性*和 *pod 反关联性*来根据其他 pod 上的键/值标签限制 pod 有资格调度到哪些节点。

- 如果新 pod 上的标签选择器与当前 pod 上的标签匹配，pod 关联性可以命令调度程序将新 pod 放置到与其他 pod 相同的节点上。
- 如果新 pod 上的标签选择器与当前 pod 上的标签匹配，pod 反关联性可以阻止调度程序将新 pod 放置到与具有相同标签的 pod 相同的节点上。

例如，您可以使用关联性规则，在服务内或相对于其他服务中的 pod 来分散或聚拢 pod。如果特定服务的 pod 的性能已知会受到另一服务的 pod 影响，那么您可以利用反关联性规则，防止前一服务的 pod 调度到与后一服务的 pod 相同的节点上。或者，您可以将服务的 pod 分散到节点间、可用性区域或可用性集，以减少相关的故障。

注意

标签选择器可能与带有多个 pod 部署的 pod 匹配。在配置反关联性规则时，请使用标签的唯一组合以避免匹配的 pod。

pod 关联性规则有两种，即*必要*规则和*偏好*规则。

必须满足必要规则，pod 才能调度到节点上。偏好规则指定在满足规则时调度程序会尝试强制执行规则，但不保证一定能强制执行成功。

注意

根据 pod 优先级和抢占设置，调度程序可能无法在不违反关联性要求的前提下为 pod 找到适合的节点。若是如此，pod 可能不会被调度。

要防止这种情况，请仔细配置优先级相同的 pod 的 pod 关联性。

您可以通过 **Pod** 规格文件配置 pod 关联性/反关联性。您可以指定必要规则或偏好规则，或同时指定这两种规则。如果您同时指定，节点必须首先满足必要规则，然后尝试满足偏好规则。

以下示例显示了配置了 pod 关联性和反关联性的 **Pod** 规格。

在本例中，pod 关联性规则指明，只有当节点至少有一个已在运行且具有键 **security** 和值 **S1** 的标签的 pod 时，pod 才可以调度到这个节点上。pod 反关联性则表示，如果节点已在运行带有键 **security** 和值 **S2**.的标签的 pod，则 pod 将偏向于不调度到该节点上。

具有 pod 关联性的 Pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    podAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
      - labelSelector:
          matchExpressions:
          - key: security 3
            operator: In 4
            values:
            - S1 5
        topologyKey: topology.kubernetes.io/zone
  containers:
  - name: with-pod-affinity
```

```
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

1. 用于配置 pod 关联性的小节。
2. 定义必要规则。
3. 5. 必须匹配键和值（标签）才会应用该规则。
4. 运算符表示现有 pod 上的标签和新 pod 规格中 **matchExpression** 参数的值集合之间的关系。可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。

具有 pod 反关联性的 Pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    podAntiAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 100 3
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security 4
              operator: In 5
              values:
              - S2
          topologyKey: kubernetes.io/hostname
  containers:
  - name: with-pod-affinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

1. 用于配置 pod 反关联性的小节。
2. 定义偏好规则。
3. 为偏好规则指定权重。优先选择权重最高的节点。
4. 描述用来决定何时应用反关联性规则的 pod 标签。指定标签的键和值。

5 运算符表示现有 pod 上的标签和新 pod 规格中 **matchExpression** 参数的值集合之间的关系。可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。

注意

如果节点标签在运行时改变，使得不再满足 pod 上的关联性规则，pod 会继续在该节点上运行。

4.3.2. 配置 pod 关联性规则

以下步骤演示了一个简单的双 pod 配置，它创建一个带有某标签的 pod，以及一个使用关联性来允许随着该 pod 一起调度的 pod。

注意

您不能直接将关联性添加到调度的 pod 中。

流程

1. 创建 pod 规格中具有特定标签的 pod：
   a. 使用以下内容创建 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: security-s1
  labels:
    security: S1
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: security-s1
    image: docker.io/ocpqe/hello-pod
    securityContext:
      runAsNonRoot: true
      seccompProfile:
        type: RuntimeDefault
```

   b. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

2. 在创建其他 pod 时，配置以下参数以添加关联性：
   a. 使用以下内容创建 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: security-s1-east
# ...
spec:
  affinity: (1)
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: (2)
      - labelSelector:
          matchExpressions:
          - key: security (3)
            values:
            - S1
            operator: In (4)
        topologyKey: topology.kubernetes.io/zone (5)
# ...
```

(1) 添加 pod 关联性。

(2) 配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数或 **preferredDuringSchedulingIgnoredDuringExecution** 参数。

(3) 指定必须满足的 **key** 和 **values**。如果您希望新 pod 与其他 pod 一起调度，请使用与第一个 pod 上标签相同的 **key** 和 **values** 参数。

(4) 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。例如，使用运算符 **In** 来要求节点上存在该标签。

(5) 指定 **topologyKey**，这是一个预填充的 Kubernetes 标签，供系统用于表示这样的拓扑域。

b. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

4.3.3. 配置 pod 反关联性规则

以下步骤演示了一个简单的双 pod 配置，它创建一个带有某标签的 pod，以及一个使用反关联性偏好规则来尝试阻止随着该 pod 一起调度的 pod。

注意

您不能直接将关联性添加到调度的 pod 中。

流程

1. 创建 pod 规格中具有特定标签的 pod：

   a. 使用以下内容创建 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: security-s1
  labels:
```

```
    security: S1
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: security-s1
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

b. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

2. 在创建其他 pod 时，配置以下参数：

a. 使用以下内容创建 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: security-s2-east
# ...
spec:
# ...
  affinity: (1)
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution: (2)
      - weight: 100 (3)
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: security (4)
              values:
              - S1
              operator: In (5)
          topologyKey: kubernetes.io/hostname (6)
# ...
```

(1) 添加 pod 反关联性。

(2) 配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数或 **preferredDuringSchedulingIgnoredDuringExecution** 参数。

(3) 对于一个首选的规则，为节点指定一个 1-100 的权重。优先选择权重最高的节点。

(4) 指定必须满足的 **key** 和 **values**。如果您希望新 pod 不与其他 pod 一起调度，请使用与第一个 pod 上标签相同的 **key** 和 **values** 参数。

(5) 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。例如，使用运算符 **In** 来要求节点上存在该标签。

6 指定 **topologyKey**，它是一个预先填充的 Kubernetes 标签，用于表示这样的拓扑域。

b. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

4.3.4. pod 关联性和反关联性规则示例

以下示例演示了 pod 关联性和 pod 反关联性。

4.3.4.1. Pod 关联性

以下示例演示了具有匹配标签和标签选择器的 pod 的 pod 关联性。

- pod **team4** 具有标签 **team:4**。

```
apiVersion: v1
kind: Pod
metadata:
  name: team4
  labels:
     team: "4"
# ...
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- pod **team4a** 在 **podAffinity** 下具有标签选择器 **team:4**。

```
apiVersion: v1
kind: Pod
metadata:
  name: team4a
# ...
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
```

```
          - key: team
            operator: In
            values:
            - "4"
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-affinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- **team4a** pod 调度到与 **team4** pod 相同的节点上。

4.3.4.2. Pod 反关联性

以下示例演示了具有匹配标签和标签选择器的 pod 的 pod 反关联性。

- pod **pod-s1** 具有标签 **security:s1**。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
# ...
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- pod **pod-s2** 在 **podAntiAffinity** 下具有标签选择器 **security:s1**。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
# ...
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
```

```
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s1
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-antiaffinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- pod **pod-s2** 无法调度到与 **pod-s1** 相同的节点上。

4.3.4.3. 无匹配标签的 Pod 反关联性

以下示例演示了在没有匹配标签和标签选择器时的 pod 的 pod 关联性。

- pod **pod-s1** 具有标签 **security:s1**。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
# ...
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- pod **pod-s2** 具有标签选择器 **security:s2**。

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
# ...
```

```
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values:
            - s2
        topologyKey: kubernetes.io/hostname
  containers:
  - name: pod-affinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

- 除非节点上具有带 **security:s2** 标签的 pod，否则不会调度 **pod-s2**。如果没有具有该标签的其他 pod，新 pod 会保持在待处理状态：

  输出示例

```
NAME     READY    STATUS    RESTARTS   AGE      IP       NODE
pod-s2   0/1      Pending   0          32s      <none>
```

4.3.5. 控制安装 Operator 的位置

4.3.6. 使用 pod 关联性和反关联性来控制安装 Operator 的位置

默认情况下，当安装 Operator 时，OpenShift Container Platform 会随机将 Operator pod 安装到其中一个 worker 节点。然而，在某些情况下，您可能希望该 pod 调度到特定节点或一组节点上。

以下示例描述了您可能希望将 Operator pod 调度到特定节点或一组节点的情况：

- 如果 Operator 需要特定的平台，如 **amd64** 或 **arm64**
- 如果 Operator 需要特定的操作系统，如 Linux 或 Windows
- 如果您希望 Operator 在同一个主机上或位于同一机架的主机上工作
- 如果您希望 Operator 在整个基础架构中分散，以避免因为网络或硬件问题而停机

您可以通过在 Operator 的 **Subscription** 对象中添加节点关联性、pod 关联性或 pod 反关联性限制来控制 Operator pod 的安装位置。节点关联性是由调度程序用来确定 pod 的可放置位置的一组规则。pod 关联性允许您确保将相关的 pod 调度到同一节点。通过 Pod 反关联性，您可以防止 pod 调度到节点上。

您可以通过向 Operator 的 **Subscription** 对象添加 pod 关联性或反关联性来控制 Operator pod 的安装位置。

以下示例演示了如何使用节点关联性或 pod 反关联性将自定义 Metrics Autoscaler Operator 实例安装到集群中的特定节点：

以下示例演示了如何使用 pod 反关联性来防止从具有特定标签的 pod 中安装自定义 Metrics Autoscaler Operator：

将 Operator pod 放置到一个或多个特定节点的 Pod 关联性示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAffinity: 1
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - test
          topologyKey: kubernetes.io/hostname
#...
```

1 将 Operator 的 pod 放置到具有 **app=test** 标签的 pod 的节点上的 pod 关联性。

防止 Operator pod 来自一个或多个特定节点的 Pod 反关联性示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAntiAffinity: 1
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: cpu
              operator: In
              values:
```

```
          - high
      topologyKey: kubernetes.io/hostname
#...
```

1 一个 pod 反关联性，它可防止 Operator 的 pod 调度到具有 **cpu=high** 标签的 pod 的节点上。

流程

要控制 Operator pod 的放置，请完成以下步骤：

1. 照常安装 Operator。
2. 如果需要，请确保您的节点已标记为正确响应关联性。
3. 编辑 Operator **Subscription** 对象以添加关联性：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAntiAffinity: 1
        requiredDuringSchedulingIgnoredDuringExecution:
          podAffinityTerm:
            labelSelector:
              matchExpressions:
              - key: kubernetes.io/hostname
                operator: In
                values:
                - ip-10-0-185-229.ec2.internal
            topologyKey: topology.kubernetes.io/zone
#...
```

1 添加 **podAffinity** 或 **podAntiAffinity**。

验证

- 要确保 pod 部署到特定的节点上，请运行以下命令：

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

输出示例

```
NAME                                      READY   STATUS    RESTARTS   AGE   IP
NODE                          NOMINATED NODE   READINESS GATES
custom-metrics-autoscaler-operator-5dcc45d656-bhshg   1/1     Running   0          50s
10.131.0.20   ip-10-0-185-229.ec2.internal   <none>           <none>
```

4.4. 使用节点关联性规则控制节点上的 POD 放置

关联性是 pod 的一个属性，用于控制它们希望调度到的节点。

在 OpenShift Container Platform 中，节点关联性是由调度程序用来确定 pod 的可放置位置的一组规则。规则是使用节点中的自定义标签和 pod 中指定的选择器进行定义的。

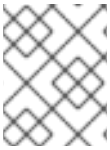
4.4.1. 了解节点关联性

节点关联性允许 pod 指定与可以放置该 pod 的一组节点的关联性。节点对放置没有控制权。

例如，您可以将 pod 配置为仅在具有特定 CPU 或位于特定可用区的节点上运行。

节点关联性规则有两种，即*必要*规则和*偏好*规则。

必须满足必要规则，pod 才能调度到节点上。偏好规则指定在满足规则时调度程序会尝试强制执行规则，但不保证一定能强制执行成功。



注意

如果节点标签在运行时改变，使得不再满足 pod 上的节点关联性规则，该 pod 将继续在这个节点上运行。

您可以通过 **Pod** 规格文件配置节点关联性。您可以指定必要规则或偏好规则，或同时指定这两种规则。如果您同时指定，节点必须首先满足必要规则，然后尝试满足偏好规则。

下例中的 **Pod** spec 包含一条规则，要求 pod 放置到具有键为 **e2e-az-NorthSouth** 且值为 **e2e-az-North** 或 **e2e-az-South** 的标签的节点上：

具有节点关联性必要规则的 pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    nodeAffinity: (1)
      requiredDuringSchedulingIgnoredDuringExecution: (2)
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-NorthSouth (3)
            operator: In (4)
            values:
            - e2e-az-North (5)
            - e2e-az-South (6)
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
```

```
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

1. 用于配置节点关联性的小节。
2. 定义必要规则。
3. 5 6 必须匹配键/值对（标签）才会应用该规则。
4. 运算符表示节点上的标签和 **Pod** 规格中 **matchExpression** 参数的值集合之间的关系。这个值可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**、**Lt** 或 **Gt**。

下例中的节点规格包含一条偏好规则，其规定优先为 pod 选择具有键为 **e2e-az-EastWest** 且值为 **e2e-az-East** 或 **e2e-az-West** 的节点：

具有节点关联性偏好规则的 pod 配置文件示例

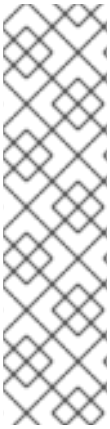
```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  affinity:
    nodeAffinity: 1
      preferredDuringSchedulingIgnoredDuringExecution: 2
      - weight: 1 3
        preference:
          matchExpressions:
          - key: e2e-az-EastWest 4
            operator: In 5
            values:
            - e2e-az-East 6
            - e2e-az-West 7
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
# ...
```

1. 用于配置节点关联性的小节。
2. 定义偏好规则。
3. 为偏好规则指定权重。优先选择权重最高的节点。

4 6 7 必须匹配键/值对（标签）才会应用该规则。

5 运算符表示节点上的标签和 **Pod** 规格中 **matchExpression** 参数的值集合之间的关系。这个值可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**、**Lt** 或 **Gt**。

没有明确的 *节点反关联性*概念，但使用 **NotIn** 或 **DoesNotExist** 运算符就能实现这种行为。



注意

如果您在同一 pod 配置中同时使用节点关联性和节点选择器，请注意以下几点：

- 如果同时配置了 **nodeSelector** 和 **nodeAffinity**，则必须满足这两个条件时 pod 才能调度到候选节点。
- 如果您指定了多个与 **nodeAffinity** 类型关联的 **nodeSelectorTerms**，那么其中一个 **nodeSelectorTerms** 满足时 pod 就能调度到节点上。
- 如果您指定了多个与 **nodeSelectorTerms** 关联的 **matchExpressions**，那么只有所有 **matchExpressions** 都满足时 pod 才能调度到节点上。

4.4.2. 配置节点关联性必要规则

必须满足必要规则，pod 才能调度到节点上。

流程

以下步骤演示了一个简单的配置，此配置会创建一个节点，以及调度程序要放置到该节点上的 pod。

1. 使用 **oc label node** 命令给节点添加标签：

```
$ oc label node node1 e2e-az-name=e2e-az1
```

提示

您还可以应用以下 YAML 来添加标签：

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    e2e-az-name: e2e-az1
#...
```

2. 创建 pod 规格中具有特定标签的 pod：

   a. 使用以下内容创建 YAML 文件：

注意

您不能直接将关联性添加到调度的 pod 中。

输出示例

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
spec:
  affinity: 1
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution: 2
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-name 3
            values:
            - e2e-az1
            - e2e-az2
            operator: In 4
#...
```

1. 添加 pod 关联性。
2. 配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数。
3. 指定必须满足的 **key** 和 **values**。如果希望新 pod 调度到您编辑的节点上，请使用与节点中标签相同的 **key** 和 **values** 参数：
4. 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。例如，使用运算符 **In** 来要求节点上存在该标签。

b. 创建 pod：

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

4.4.3. 配置首选的节点关联性规则

偏好规则指定在满足规则时调度程序会尝试强制执行规则，但不保证一定能强制执行成功。

流程

以下步骤演示了一个简单的配置，此配置会创建一个节点，以及调度程序尝试放置到该节点上的 pod。

1. 使用 **oc label node** 命令给节点添加标签：

```
$ oc label node node1 e2e-az-name=e2e-az3
```

2. 创建具有特定标签的 pod：

a. 使用以下内容创建 YAML 文件：

注意

您不能直接将关联性添加到调度的 pod 中。

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: s1
spec:
  affinity: (1)
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution: (2)
      - weight: (3)
        preference:
          matchExpressions:
          - key: e2e-az-name (4)
            values:
            - e2e-az3
            operator: In (5)
#...
```

1. 添加 pod 关联性。
2. 配置 **preferredDuringSchedulingIgnoredDuringExecution** 参数。
3. 为节点指定一个数字为 1-100 的权重。优先选择权重最高的节点。
4. 指定必须满足的 **key** 和 **values**。如果希望新 pod 调度到您编辑的节点上，请使用与节点中标签相同的 **key** 和 **values** 参数：
5. 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。例如，使用运算符 **In** 来要求节点上存在该标签。

b. 创建 pod。

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

4.4.4. 节点关联性规则示例

以下示例演示了节点关联性。

4.4.4.1. 具有匹配标签的节点关联性

以下示例演示了具有匹配标签的节点与 pod 的节点关联性：

- Node1 节点具有标签 **zone:us**：

```
$ oc label node node1 zone=us
```

提示

您还可以应用以下 YAML 来添加标签：

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    zone: us
#...
```

- pod-s1 pod 在节点关联性必要规则下具有 **zone** 和 **us** 键/值对：

```
$ cat pod-s1.yaml
```

输出示例

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
#...
```

- pod-s1 pod 可以调度到 Node1 上：

```
$ oc get pod -o wide
```

输出示例

```
NAME    READY    STATUS      RESTARTS  AGE     IP      NODE
pod-s1  1/1      Running     0         4m      IP1     node1
```

4.4.4.2. 没有匹配标签的节点关联性

以下示例演示了无匹配标签的节点与 pod 的节点关联性：

- Node1 节点具有标签 **zone:emea**:

```
$ oc label node node1 zone=emea
```

提示

您还可以应用以下 YAML 来添加标签：

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    zone: emea
#...
```

- pod-s1 pod 在节点关联性必要规则下具有 **zone** 和 **us** 键/值对：

```
$ cat pod-s1.yaml
```

输出示例

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - image: "docker.io/ocpqe/hello-pod"
      name: hello-pod
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: "zone"
              operator: In
              values:
              - us
#...
```

- pod-s1 pod 无法调度到 Node1 上：

```
$ oc describe pod pod-s1
```

输出示例

```
...

Events:
 FirstSeen LastSeen Count From              SubObjectPath  Type                Reason
 --------- -------- ----- ----              -------------  --------            ------
 1m        33s      8     default-scheduler Warning        FailedScheduling    No nodes are
available that match all of the following predicates:: MatchNodeSelector (1).
```

4.4.5. 控制安装 Operator 的位置

4.4.6. 使用节点关联性来控制安装 Operator 的位置

默认情况下，当安装 Operator 时，OpenShift Container Platform 会随机将 Operator pod 安装到其中一个 worker 节点。然而，在某些情况下，您可能希望该 pod 调度到特定节点或一组节点上。

以下示例描述了您可能希望将 Operator pod 调度到特定节点或一组节点的情况：

- 如果 Operator 需要特定的平台，如 **amd64** 或 **arm64**
- 如果 Operator 需要特定的操作系统，如 Linux 或 Windows
- 如果您希望 Operator 在同一个主机上或位于同一机架的主机上工作
- 如果您希望 Operator 在整个基础架构中分散，以避免因为网络或硬件问题而停机

您可以通过在 Operator 的 **Subscription** 对象中添加节点关联性、pod 关联性或 pod 反关联性限制来控制 Operator pod 的安装位置。节点关联性是由调度程序用来确定 pod 的可放置位置的一组规则。pod 关联性允许您确保将相关的 pod 调度到同一节点。通过 Pod 反关联性，您可以防止 pod 调度到节点上。

您可以通过在 Operator 的 **Subscription** 对象中添加节点关联性约束来控制 Operator pod 的安装位置。

以下示例演示了如何使用节点关联性或 pod 反关联性将自定义 Metrics Autoscaler Operator 实例安装到集群中的特定节点：以下示例演示如何使用节点关联性将自定义 Metrics Autoscaler Operator 实例安装到集群中的特定节点：

将 Operator pod 放置到特定节点的节点关联性示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
```

```
    nodeAffinity: (1)
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/hostname
            operator: In
            values:
            - ip-10-0-163-94.us-west-2.compute.internal
#...
```

(1) 要求 Operator 的 pod 调度到名为 **ip-10-0-163-94.us-west-2.compute.internal** 的节点关联性。

将 Operator pod 放置到带有特定平台的节点关联性示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      nodeAffinity: (1)
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/arch
              operator: In
              values:
              - arm64
            - key: kubernetes.io/os
              operator: In
              values:
              - linux
#...
```

(1) 要求 Operator 的 pod 调度到具有 **kubernetes.io/arch=arm64** 和 **kubernetes.io/os=linux** 标签的节点上。

流程

要控制 Operator pod 的放置，请完成以下步骤：

1. 照常安装 Operator。
2. 如果需要，请确保您的节点已标记为正确响应关联性。
3. 编辑 Operator **Subscription** 对象以添加关联性：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
```

```
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity: 1
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - ip-10-0-185-229.ec2.internal
#...
```

1 添加 **nodeAffinity**、**podAffinity** 或 **podAntiAffinity**。有关创建关联性的详情，请参考下面的附加资源部分。

添加 **nodeAffinity**。

验证

- 要确保 pod 部署到特定的节点上，请运行以下命令：

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

输出示例

```
NAME                                             READY   STATUS    RESTARTS   AGE   IP
NODE                          NOMINATED NODE   READINESS GATES
custom-metrics-autoscaler-operator-5dcc45d656-bhshg   1/1     Running   0          50s
10.131.0.20   ip-10-0-185-229.ec2.internal   <none>           <none>
```

4.4.7. 其他资源

- 了解如何更新节点上的标签

4.5. 将 POD 放置到过量使用的节点

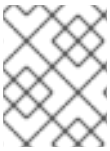
处于*过量使用（overcommited）*状态时，容器计算资源请求和限制的总和超过系统中可用的资源。过量使用常用于开发环境，因为在这种环境中可以接受以牺牲保障性能来换取功能的情况。

请求和限制可让管理员允许和管理节点上资源的过量使用。调度程序使用请求来调度容器，并提供最低服务保证。限制约束节点上可以消耗的计算资源数量。

4.5.1. 了解过量使用

请求和限制可让管理员允许和管理节点上资源的过量使用。调度程序使用请求来调度容器，并提供最低服务保证。限制约束节点上可以消耗的计算资源数量。

OpenShift Container Platform 管理员可以通过配置主控机（master）来覆盖开发人员容器上设置的请求和限制之间的比率，来控制过量使用的程度并管理节点上的容器密度。与项目一级上的用于指定限制和默认值的 **LimitRange** 对象一起使用，可以调整容器限制和请求以达到所需的过量使用程度。



注意

如果没有在容器中设定限制，则这些覆盖无效。创建一个带有默认限制（基于每个独立的项目或在项目模板中）的 **LimitRange** 对象，以确保能够应用覆盖。

在进行这些覆盖后，容器限制和请求必须仍需要满足项目中的 **LimitRange** 对象的要求。这可能会导致 pod 被禁止的情况。例如，开发人员指定了一个接近最小限制的限制，然后其请求被覆盖为低于最小限制。这个问题在以后会加以解决，但目前而言，请小心地配置此功能和 **LimitRange** 对象。

4.5.2. 了解节点过量使用

在过量使用的环境中，务必要正确配置节点，以提供最佳的系统行为。

当节点启动时，它会确保为内存管理正确设置内核可微调标识。除非物理内存不足，否则内核应该永不会在内存分配时失败。

为确保这一行为，OpenShift Container Platform 通过将 **vm.overcommit_memory** 参数设置为 **1** 来覆盖默认操作系统设置，从而将内核配置为始终过量使用内存。

OpenShift Container Platform 还通过将 **vm.panic_on_oom** 参数设置为 **0**，将内核配置为不会在内存不足时崩溃。设置为 0 可告知内核在内存不足 (OOM) 情况下调用 oom_killer，以根据优先级终止进程

您可以通过对节点运行以下命令来查看当前的设置：

```
$ sysctl -a |grep commit
```

输出示例

```
#...
vm.overcommit_memory = 0
#...
```

```
$ sysctl -a |grep panic
```

输出示例

```
#...
vm.panic_on_oom = 0
#...
```

注意

节点上应该已设置了上述标记，不需要进一步操作。

您还可以为每个节点执行以下配置：

- 使用 CPU CFS 配额禁用或强制实施 CPU 限制
- 为系统进程保留资源
- 为不同的服务质量等级保留内存

4.6. 使用节点污点控制 POD 放置

通过污点和容限，节点可以控制哪些 pod 应该（或不应该）调度到节点上。

4.6.1. 了解污点和容限

通过使用*污点（taint）*，节点可以拒绝调度 pod，除非 pod 具有匹配的*容限（toleration）*。

您可以通过节点规格（**NodeSpec**）将污点应用到**节点**，并通过 **Pod** 规格（**PodSpec**）将容限应用到 pod。当您应用污点时，调度程序无法将 pod 放置到该节点上，除非 pod 可以容限该污点。

节点规格中的污点示例

```
apiVersion: v1
kind: Node
metadata:
  name: my-node
#...
spec:
  taints:
  - effect: NoExecute
    key: key1
    value: value1
#...
```

Pod 规格中的容限示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
    tolerationSeconds: 3600
#...
```

污点与容限由 key、value 和 effect 组成。

表 4.1. 污点和容限组件

参数	描述
key	**key** 是任意字符串，最多 253 个字符。key 必须以字母或数字开头，可以包含字母、数字、连字符、句点和下划线。
value	**value** 是任意字符串，最多 63 个字符。value 必须以字母或数字开头，可以包含字母、数字、连字符、句点和下划线。
effect	effect 的值包括： **NoSchedule** [1]： • 与污点不匹配的新 pod 不会调度到该节点上。 • 该节点上现有的 pod 会保留。 **PreferNoSchedule**： • 与污点不匹配的新 pod 可以调度到该节点上，但调度程序会尽量不这样调度。 • 该节点上现有的 pod 会保留。 **NoExecute**： • 与污点不匹配的新 pod 无法调度到该节点上。 • 节点上没有匹配容限的现有 pod 将被移除。
operator	**Equal**：**key**/**value**/**effect** 参数必须匹配。这是默认值。 **Exists**：**key**/**effect** 参数必须匹配。您必须保留一个空的 **value** 参数，这将匹配任何值。

1. 如果向 control plane 节点添加了一个 **NoSchedule** 污点，节点必须具有 **node-role.kubernetes.io/master=:NoSchedule** 污点，这默认会添加。
例如：

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-
cdc1ab7da414629332cc4c3926e6e59c
  name: my-node
#...
spec:
```

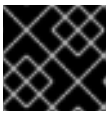
```
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
#...
```

容限与污点匹配：

- 如果 **operator** 参数设为 **Equal**：
  - **key** 参数相同；
  - **value** 参数相同；
  - **effect** 参数相同。
- 如果 **operator** 参数设为 **Exists**：
  - **key** 参数相同；
  - **effect** 参数相同。

OpenShift Container Platform 中内置了以下污点：

- **node.kubernetes.io/not-ready**：节点未就绪。这与节点状况 **Ready=False** 对应。
- **node.kubernetes.io/unreachable**：节点无法从节点控制器访问。这与节点状况 **Ready=Unknown** 对应。
- **node.kubernetes.io/memory-pressure**：节点存在内存压力问题。这与节点状况 **MemoryPressure=True** 对应。
- **node.kubernetes.io/disk-pressure**：节点存在磁盘压力问题。这与节点状况 **DiskPressure=True** 对应。
- **node.kubernetes.io/network-unavailable**：节点网络不可用。
- **node.kubernetes.io/unschedulable**：节点不可调度。
- **node.cloudprovider.kubernetes.io/uninitialized**：当节点控制器通过外部云提供商启动时，在节点上设置这个污点来将其标记为不可用。在云控制器管理器中的某个控制器初始化这个节点后，kubelet 会移除此污点。
- **node.kubernetes.io/pid-pressure** ：节点具有 pid 压力。这与节点状况 **PIDPressure=True** 对应。



重要

OpenShift Container Platform 不设置默认的 pid.available **evictionHard**。

4.6.1.1. 了解如何使用容限秒数来延迟 pod 驱除

您可以通过在 **Pod** 规格或 **MachineSet** 对象中指定 **tolerationSeconds** 参数，指定 pod 在被驱除前可以保持与节点绑定的时长。如果将具有 **NoExecute effect** 的污点添加到节点，则容限污点（包含 **tolerationSeconds** 参数）的 pod，在此期限内 pod 不会被驱除。

输出示例

■

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
    tolerationSeconds: 3600
#...
```

在这里，如果此 pod 正在运行但没有匹配的容限，pod 保持与节点绑定 3600 秒，然后被驱除。如果污点在这个时间之前移除，pod 就不会被驱除。

4.6.1.2. 了解如何使用多个污点

您可以在同一个节点中放入多个污点，并在同一 pod 中放入多个容限。OpenShift Container Platform 按照如下所述处理多个污点和容限：

1. 处理 pod 具有匹配容限的污点。
2. 其余的不匹配污点在 pod 上有指示的 effect：
   - 如果至少有一个不匹配污点具有 **NoSchedule** effect，则 OpenShift Container Platform 无法将 pod 调度到该节点上。
   - 如果没有不匹配污点具有 **NoSchedule** effect，但至少有一个不匹配污点具有 **PreferNoSchedule** effect，则 OpenShift Container Platform 尝试不将 pod 调度到该节点上。
   - 如果至少有一个未匹配污点具有 **NoExecute** effect，OpenShift Container Platform 会将 pod 从该节点驱除（如果它已在该节点上运行），或者不将 pod 调度到该节点上（如果还没有在该节点上运行）。
     - 不容许污点的 Pod 会立即被驱除。
     - 如果 Pod 容许污点而没有在 **Pod** 规格中指定 **tolerationSeconds**，则会永久保持绑定。
     - 如果 Pod 容许污点，且指定了 **tolerationSeconds**，则会在指定的时间里保持绑定。

例如：

- 向节点添加以下污点：

```
$ oc adm taint nodes node1 key1=value1:NoSchedule
```

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

```
$ oc adm taint nodes node1 key2=value2:NoSchedule
```

- pod 具有以下容限：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoSchedule"
  - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoExecute"
#...
```

在本例中，pod 无法调度到节点上，因为没有与第三个污点匹配的容限。如果在添加污点时 pod 已在节点上运行，pod 会继续运行，因为第三个污点是三个污点中 pod 唯一不容许的污点。

4.6.1.3. 了解 pod 调度和节点状况（根据状况保留节点）

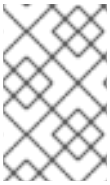
Taint Nodes By Condition （默认启用）可自动污点报告状况的节点，如内存压力和磁盘压力。如果某个节点报告一个状况，则添加一个污点，直到状况被清除为止。这些污点具有 **NoSchedule** effect；即，pod 无法调度到该节点上，除非 pod 有匹配的容限。

在调度 pod 前，调度程序会检查节点上是否有这些污点。如果污点存在，则将 pod 调度到另一个节点。由于调度程序检查的是污点而非实际的节点状况，因此您可以通过添加适当的 pod 容限，将调度程序配置为忽略其中一些节点状况。

为确保向后兼容，守护进程会自动将下列容限添加到所有守护进程中：

- node.kubernetes.io/memory-pressure
- node.kubernetes.io/disk-pressure
- node.kubernetes.io/unschedulable（1.10 或更高版本）
- node.kubernetes.io/network-unavailable（仅限主机网络）

您还可以在守护进程集中添加任意容限。



注意

control plane 还会在具有 QoS 类的 pod 中添加 **node.kubernetes.io/memory-pressure** 容限。这是因为 Kubernetes 在 **Guaranteed** 或 **Burstable** QoS 类中管理 pod。新的 **BestEffort** pod 不会调度到受影响的节点上。

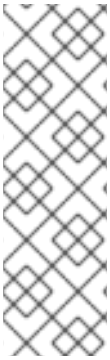
4.6.1.4. 了解根据状况驱除 pod（基于垃圾的驱除）

Taint-Based Evictions 功能默认是启用的，可以从遇到特定状况（如 **not-ready** 和 **unreachable**）的节点驱除 pod。当节点遇到其中一个状况时，OpenShift Container Platform 会自动给节点添加污点，并开始驱除 pod 以及将 pod 重新调度到其他节点。

Taint Based Evictions 具有 **NoExecute** 效果，不容许污点的 pod 都被立即驱除，容许污点的 pod 不会被驱除，除非 pod 使用 **tolerationSeconds** 参数。

tolerationSeconds 参数允许您指定 pod 保持与具有节点状况的节点绑定的时长。如果在 **tolerationSections** 到期后状况仍然存在，则污点会保持在节点上，并且具有匹配容限的 pod 将被驱除。如果状况在 **tolerationSeconds** 到期前清除，则不会删除具有匹配容限的 pod。

如果使用没有值的 **tolerationSeconds** 参数，则 pod 不会因为未就绪和不可访问的节点状况而被驱除。



注意

OpenShift Container Platform 会以限速方式驱除 pod，从而防止在主控机从节点分离等情形中发生大量 pod 驱除。

默认情况下，如果给定区域中的节点超过 55% 的节点不健康，节点生命周期控制器会将该区域的状态改为 **PartialDisruption**，并且 pod 驱除率会减少。对于此状态的小型集群（默认为 50 个节点或更少），这个区中的节点不会污点，驱除会被停止。

如需更多信息，请参阅 Kubernetes 文档中的 有关驱除率限制。

OpenShift Container Platform 会自动为 **node.kubernetes.io/not-ready** 和 **node.kubernetes.io/unreachable** 添加容限并设置 **tolerationSeconds=300**，除非 **Pod** 配置中指定了其中任一种容限。

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300 (1)
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
#...
```

(1) 这些容限确保了在默认情况下，pod 在检测到这些节点条件问题中的任何一个时，会保持绑定五分钟。

您可以根据需要配置这些容限。例如，如果您有一个具有许多本地状态的应用程序，您可能希望在发生网络分区时让 pod 与节点保持绑定更久一些，以等待分区恢复并避免 pod 驱除行为的发生。

由守护进程集生成的 pod 在创建时会带有以下污点的 **NoExecute** 容限，且没有 **tolerationSeconds**:

- **node.kubernetes.io/unreachable**
- **node.kubernetes.io/not-ready**

因此，守护进程集 pod 不会被驱除。

4.6.1.5. 容限所有污点

您可以通过添加 **operator: "Exists"** 容限而无需 **key** 和 **values** 参数，将 pod 配置为容许所有污点。具有此容限的 Pod 不会从具有污点的节点中删除。

用于容忍所有污点的Pod 规格

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - operator: "Exists"
#...
```

4.6.2. 添加污点和容限

您可以为 pod 和污点添加容限，以便节点能够控制哪些 pod 应该或不应该调度到节点上。对于现有的 pod 和节点，您应首先将容限添加到 pod，然后将污点添加到节点，以避免在添加容限前从节点上移除 pod。

流程

1. 通过编辑 **Pod** spec 使其包含 **tolerations** 小节来向 pod 添加容限：

   使用 Equal 运算符的 pod 配置文件示例

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: my-pod
   #...
   spec:
     tolerations:
     - key: "key1" (1)
       value: "value1"
       operator: "Equal"
       effect: "NoExecute"
       tolerationSeconds: 3600 (2)
   #...
   ```

   (1) 容限参数，如 **Taint 和 toleration 组件**表中所述。

   (2) **tolerationSeconds** 参数指定 pod 在被驱除前可以保持与节点绑定的时长。

   例如：

   使用 Exists 运算符的 pod 配置文件示例

   ```
   apiVersion: v1
   kind: Pod
   ```

```
metadata:
  name: my-pod
#...
spec:
   tolerations:
    - key: "key1"
      operator: "Exists" (1)
      effect: "NoExecute"
      tolerationSeconds: 3600
#...
```

(1) **Exists** 运算符不会接受一个 **value**。

本例在 **node1** 上放置一个键为 **key1** 且值为 **value1** 的污点，污点效果是 **NoExecute**。

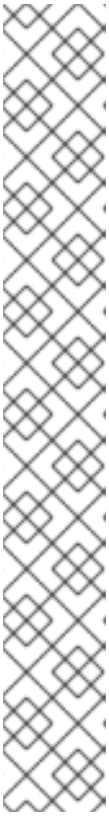
2. 通过以下命令，使用 **Taint 和 toleration 组件**表中描述的参数为节点添加污点：

```
$ oc adm taint nodes <node_name> <key>=<value>:<effect>
```

例如：

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

此命令在 **node1** 上放置一个键为 **key1**，值为 **value1** 的污点，其效果是 **NoExecute**。



注意

如果向 control plane 节点添加了一个 **NoSchedule** 污点，节点必须具有 **node-role.kubernetes.io/master=:NoSchedule** 污点，这默认会添加。

例如：

```
apiVersion: v1
kind: Node
metadata:
  annotations:
    machine.openshift.io/machine: openshift-machine-api/ci-ln-62s7gtb-f76d1-v8jxv-master-0
    machineconfiguration.openshift.io/currentConfig: rendered-master-cdc1ab7da414629332cc4c3926e6e59c
  name: my-node
#...
spec:
  taints:
  - effect: NoSchedule
    key: node-role.kubernetes.io/master
#...
```

pod 上的容限与节点上的污点匹配。具有任一容限的 pod 可以调度到 **node1** 上。

4.6.2.1. 使用计算机器集添加污点和容限

您可以使用计算机器集为节点添加污点。与 **MachineSet** 对象关联的所有节点都会使用污点更新。容限响应由计算机器设置添加的污点，其方式与直接添加到节点的污点相同。

流程

1. 通过编辑 **Pod** spec 使其包含 **tolerations** 小节来向 pod 添加容限：

   使用 Equal 运算符的 pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: "key1" 1
    value: "value1"
    operator: "Equal"
    effect: "NoExecute"
    tolerationSeconds: 3600 2
#...
```

   1 容限参数，如 **Taint 和 toleration 组件**表中所述。

   2 **tolerationSeconds** 参数指定 pod 在被驱除前与节点绑定的时长。

   例如：

   使用 Exists 运算符的 pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
  - key: "key1"
    operator: "Exists"
    effect: "NoExecute"
    tolerationSeconds: 3600
#...
```

2. 将污点添加到 **MachineSet** 对象：

   a. 为您想要污点的节点编辑 **MachineSet** YAML，也可以创建新 **MachineSet** 对象：

```
$ oc edit machineset <machineset>
```

   b. 将污点添加到 **spec.template.spec** 部分：

      计算机器设置规格中的污点示例

      ▪

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: my-machineset
#...
spec:
#...
  template:
#...
    spec:
      taints:
      - effect: NoExecute
        key: key1
        value: value1
#...
```

本例在节点上放置一个键为 **key1**，值为 **value1** 的污点，污点效果是 **NoExecute**。

c. 将计算机器设置为 0:

```
$ oc scale --replicas=0 machineset <machineset> -n openshift-machine-api
```

提示

您还可以应用以下 YAML 来扩展计算机器集：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  replicas: 0
```

等待机器被删除。

d. 根据需要扩展计算机器：

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

或者：

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

等待机器启动。污点添加到与 **MachineSet** 对象关联的节点上。

4.6.2.2. 使用污点和容限将用户绑定到节点

如果要指定一组节点供特定用户独占使用，为 pod 添加容限。然后，在这些节点中添加对应的污点。具有容限的 pod 被允许使用污点节点，或集群中的任何其他节点。

如果您希望确保 pod 只调度到那些污点节点，还要将标签添加到同一组节点，并为 pod 添加节点关联性，以便 pod 只能调度到具有该标签的节点。

流程

配置节点以使用户只能使用该节点：

1. 为这些节点添加对应的污点：
   例如：

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

提示

您还可以应用以下 YAML 来添加污点：

```
kind: Node
apiVersion: v1
metadata:
  name: my-node
#...
spec:
  taints:
    - key: dedicated
      value: groupName
      effect: NoSchedule
#...
```

2. 通过编写自定义准入控制器，为 pod 添加容限。

4.6.2.3. 使用节点选择器和容限创建项目

您可以创建一个使用节点选择器和容限（设为注解）的项目，以控制 pod 放置到特定的节点上。然后，项目中创建的任何后续资源都会调度到与容限匹配的污点节点上。

先决条件

- 通过使用计算机器集或直接编辑节点，已将节点选择的标签添加到一个或多个节点上。
- 通过使用计算机器集或直接编辑节点，已将污点添加到一个或多个节点上。

流程

1. 创建 **Project** 资源定义，在 **metadata.annotations** 部分指定节点选择器和容限：

project.yaml 文件示例

```
kind: Project
apiVersion: project.openshift.io/v1
metadata:
  name: <project_name> 1
  annotations:
    openshift.io/node-selector: '<label>' 2
    scheduler.alpha.kubernetes.io/defaultTolerations: >-
```

```
    [{"operator": "Exists", "effect": "NoSchedule", "key":
    "<key_name>"} 3
    ]
```

1. 项目名称。
2. 默认节点选择器标签。
3. 容限参数，如 **Taint 和 toleration 组件**表中所述。本例使用 **NoSchedule effect**（允许节点上现有的 pod 保留）和 **Exists** 运算符（不使用值）。

2. 使用 **oc apply** 命令来创建项目：

```
$ oc apply -f project.yaml
```

现在，**<project_name>** 命名空间中创建的任何后续资源都应调度到指定的节点上。

其他资源

- 手动向节点或计算机器集添加污点和容限
- 创建项目范围节点选择器
- Operator 工作负载的 Pod 放置

4.6.2.4. 使用污点和容限控制具有特殊硬件的节点

如果集群中有少量节点具有特殊的硬件，您可以使用污点和容限让不需要特殊硬件的 pod 与这些节点保持距离，从而将这些节点保留给那些确实需要特殊硬件的 pod。您还可以要求需要特殊硬件的 pod 使用特定的节点。

您可以将容限添加到需要特殊硬件并污点具有特殊硬件的节点的 pod 中。

流程

确保为特定 pod 保留具有特殊硬件的节点：

1. 为需要特殊硬件的 pod 添加容限。
例如：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
  tolerations:
    - key: "disktype"
      value: "ssd"
      operator: "Equal"
      effect: "NoSchedule"
      tolerationSeconds: 3600
#...
```

2. 使用以下命令之一，给拥有特殊硬件的节点添加污点：

```
$ oc adm taint nodes <node-name> disktype=ssd:NoSchedule
```

或者：

```
$ oc adm taint nodes <node-name> disktype=ssd:PreferNoSchedule
```

提示

您还可以应用以下 YAML 来添加污点：

```
kind: Node
apiVersion: v1
metadata:
  name: my_node
#...
spec:
  taints:
    - key: disktype
      value: ssd
      effect: PreferNoSchedule
#...
```

4.6.3. 删除污点和容限

您可以根据需要，从节点移除污点并从 pod 移除容限。您应首先将容限添加到 pod，然后将污点添加到节点，以避免在添加容限前从节点上移除 pod。

流程

移除污点和容限：

1. 从节点移除污点：

```
$ oc adm taint nodes <node-name> <key>-
```

例如：

```
$ oc adm taint nodes ip-10-0-132-248.ec2.internal key1-
```

输出示例

```
node/ip-10-0-132-248.ec2.internal untainted
```

2. 要从 pod 移除某一容限，请编辑 **Pod** 规格来移除该容限：

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
#...
spec:
```

```
  tolerations:
  - key: "key2"
    operator: "Exists"
    effect: "NoExecute"
    tolerationSeconds: 3600
#...
```

4.7. 使用节点选择器将 POD 放置到特定节点

*节点选择器*指定一个键/值对映射，该映射使用 pod 中指定的自定义标签和选择器定义。

要使 pod 有资格在节点上运行，pod 必须具有与节点上标签相同的键值节点选择器。

4.7.1. 关于节点选择器

您可以使用节点上的 pod 和标签上的节点选择器来控制 pod 的调度位置。使用节点选择器时，OpenShift Container Platform 会将 pod 调度到包含匹配标签的节点。

您可以使用节点选择器将特定的 pod 放置到特定的节点上，集群范围节点选择器将新 pod 放置到集群中的任何特定节点上，以及项目节点选择器，将新 pod 放置到特定的节点上。

例如，作为集群管理员，您可以创建一个基础架构，应用程序开发人员可以通过在创建的每个 pod 中包括节点选择器，将 pod 部署到最接近其地理位置的节点。在本例中，集群由五个数据中心组成，分布在两个区域。在美国，将节点标记为 **us-east**、**us-central** 或 **us-west**。在亚太地区（APAC），将节点标记为 **apac-east** 或 **apac-west**。开发人员可在其创建的 pod 中添加节点选择器，以确保 pod 调度到这些节点上。

如果 **Pod** 对象包含节点选择器，但没有节点具有匹配的标签，则不会调度 pod。

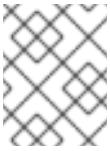
重要

如果您在同一 pod 配置中使用节点选择器和节点关联性，则以下规则控制 pod 放置到节点上：

- 如果同时配置了 **nodeSelector** 和 **nodeAffinity**，则必须满足这两个条件时 pod 才能调度到候选节点。
- 如果您指定了多个与 **nodeAffinity** 类型关联的 **nodeSelectorTerms**，那么其中一个 **nodeSelectorTerms** 满足时 pod 就能调度到节点上。
- 如果您指定了多个与 **nodeSelectorTerms** 关联的 **matchExpressions**，那么只有所有 **matchExpressions** 都满足时 pod 才能调度到节点上。

特定 pod 和节点上的节点选择器

您可以使用节点选择器和标签控制特定 pod 调度到哪些节点上。
要使用节点选择器和标签，首先标记节点以避免 pod 被取消调度，然后将节点选择器添加到 pod。



注意

您不能直接将节点选择器添加到现有调度的 pod 中。您必须标记控制 pod 的对象，如部署配置。

例如，以下 **Node** 对象具有 **region: east** 标签：

带有标识的 Node 对象示例

```
kind: Node
apiVersion: v1
metadata:
  name: ip-10-0-131-14.ec2.internal
  selfLink: /api/v1/nodes/ip-10-0-131-14.ec2.internal
  uid: 7bc2580a-8b8e-11e9-8e01-021ab4174c74
  resourceVersion: '478704'
  creationTimestamp: '2019-06-10T14:46:08Z'
  labels:
    kubernetes.io/os: linux
    topology.kubernetes.io/zone: us-east-1a
    node.openshift.io/os_version: '4.5'
    node-role.kubernetes.io/worker: ''
    topology.kubernetes.io/region: us-east-1
    node.openshift.io/os_id: rhcos
    node.kubernetes.io/instance-type: m4.large
    kubernetes.io/hostname: ip-10-0-131-14
    kubernetes.io/arch: amd64
    region: east (1)
    type: user-node
#...
```

1 与 pod 节点选择器匹配的标签。

pod 具有 **type: user-node,region: east** 节点选择器：

使用节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
#...
spec:
  nodeSelector: (1)
    region: east
    type: user-node
#...
```

1 与节点标签匹配的节点选择器。节点必须具有每个节点选择器的标签。

使用示例 pod 规格创建 pod 时，它可以调度到示例节点上。

默认集群范围节点选择器

使用默认集群范围节点选择器时，如果您在集群中创建 pod,OpenShift Container Platform 会将默认节点选择器添加到 pod，并将该 pod 调度到具有匹配标签的节点。
例如，以下 **Scheduler** 对象具有默认的集群范围的 **region=east** 和 **type=user-node** 节点选择器：

Scheduler Operator 自定义资源示例

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
#...
spec:
  defaultNodeSelector: type=user-node,region=east
#...
```

集群中的节点具有 **type=user-node,region=east** 标签：

Node 对象示例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
#...
  labels:
    region: east
    type: user-node
#...
```

使用节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  name: s1
#...
spec:
  nodeSelector:
    region: east
#...
```

当您使用示例集群中的 pod spec 创建 pod 时，该 pod 会使用集群范围节点选择器创建，并调度到标记的节点：

在标记的节点上带有 pod 的 pod 列表示例

```
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE
NOMINATED NODE   READINESS GATES
pod-s1   1/1     Running   0          20s   10.131.2.6   ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>           <none>
```

注意

如果您在其中创建 pod 的项目具有项目节点选择器，则该选择器优先于集群范围节点选择器。如果 pod 没有项目节点选择器，则 pod 不会被创建或调度。

项目节点选择器

使用项目节点选择器时，如果您在此项目中创建 pod,OpenShift Container Platform 会将节点选择器添加到 pod，并将 pod 调度到具有匹配标签的节点。如果存在集群范围默认节点选择器，则以项目节点选择器为准。
例如，以下项目具有 **region=east** 节点选择器：

Namespace 对象示例

```
apiVersion: v1
kind: Namespace
metadata:
  name: east-region
  annotations:
    openshift.io/node-selector: "region=east"
#...
```

以下节点具有 **type=user-node,region=east** 标签：

Node 对象示例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
#...
  labels:
    region: east
    type: user-node
#...
```

当您使用本例项目中的示例 pod 规格创建 pod 时，pod 会使用项目节点选择器创建，并调度到标记的节点：

Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  namespace: east-region
#...
spec:
  nodeSelector:
    region: east
    type: user-node
#...
```

在标记的节点上带有 pod 的 pod 列表示例

```
NAME    READY   STATUS    RESTARTS   AGE   IP           NODE
NOMINATED NODE   READINESS GATES
pod-s1   1/1    Running   0          20s   10.131.2.6   ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
<none>           <none>
```

如果 pod 包含不同的节点选择器，则项目中的 pod 不会被创建或调度。例如，如果您将以下 Pod 部署到示例项目中，则不会创建它：

带有无效节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  name: west-region
#...
spec:
  nodeSelector:
    region: west
#...
```

4.7.2. 使用节点选择器控制 pod 放置

您可以使用节点上的 pod 和标签上的节点选择器来控制 pod 的调度位置。使用节点选择器时，OpenShift Container Platform 会将 pod 调度到包含匹配标签的节点。

您可向节点、计算机器集或机器配置添加标签。将标签添加到计算机器集可确保节点或机器停机时，新节点具有该标签。如果节点或机器停机，添加到节点或机器配置的标签不会保留。

要将节点选择器添加到现有 pod 中，将节点选择器添加到该 pod 的控制对象中，如 **ReplicaSet** 对象、**DaemonSet** 对象、**StatefulSet** 对象、**Deployment** 对象或 **DeploymentConfig** 对象。任何属于该控制对象的现有 pod 都会在具有匹配标签的节点上重新创建。如果要创建新 pod，可以将节点选择器直接添加到 pod 规格中。如果 pod 没有控制对象，您必须删除 pod，编辑 pod 规格并重新创建 pod。

注意

您不能直接将节点选择器添加到现有调度的 pod 中。

先决条件

要将节点选择器添加到现有 pod 中，请确定该 pod 的控制对象。例如，**router-default-66d5cf9464-m2g75** pod 由 **router-default-66d5cf9464** 副本集控制：

```
$ oc describe pod router-default-66d5cf9464-7pwkc
```

输出示例

```
kind: Pod
apiVersion: v1
metadata:
# ...
Name:               router-default-66d5cf9464-7pwkc
Namespace:          openshift-ingress
# ...
Controlled By:      ReplicaSet/router-default-66d5cf9464
# ...
```

Web 控制台在 pod YAML 的 **ownerReferences** 下列出控制对象：

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: router-default-66d5cf9464-7pwkc
# ...
  ownerReferences:
    - apiVersion: apps/v1
      kind: ReplicaSet
      name: router-default-66d5cf9464
      uid: d81dd094-da26-11e9-a48a-128e7edf0312
      controller: true
      blockOwnerDeletion: true
# ...
```

流程

1. 使用计算机器集或直接编辑节点，为节点添加标签：

   - 在创建节点时，使用 **MachineSet** 对象向由计算机器集管理的节点添加标签：

     a. 运行以下命令，将标签添加到 **MachineSet** 对象中：

     ```
     $ oc patch MachineSet <name> --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
     <value>","<key>"="<value>"}}]'  -n openshift-machine-api
     ```

     例如：

     ```
     $ oc patch MachineSet abc612-msrtw-worker-us-east-1c  --type='json' -
     p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
     node","region":"east"}}]'  -n openshift-machine-api
     ```

     提示

     您还可以应用以下 YAML 来向计算机器集中添加标签：

     ```
     apiVersion: machine.openshift.io/v1beta1
     kind: MachineSet
     metadata:
       name: xf2bd-infra-us-east-2a
       namespace: openshift-machine-api
     spec:
       template:
         spec:
           metadata:
             labels:
               region: "east"
               type: "user-node"
     # ...
     ```

     b. 使用 **oc edit** 命令验证标签是否已添加到 **MachineSet** 对象中：
     例如：

     ```
     $ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
     ```

MachineSet 对象示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet

# ...

spec:
# ...
  template:
    metadata:
# ...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
# ...
```

- 直接向节点添加标签：

  a. 为节点编辑 **Node** 对象：

  ```
  $ oc label nodes <name> <key>=<value>
  ```

  例如，若要为以下节点添加标签：

  ```
  $ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
  ```

  提示

  您还可以应用以下 YAML 来向节点添加标签：

  ```
  kind: Node
  apiVersion: v1
  metadata:
    name: hello-node-6fbccf8d9
    labels:
      type: "user-node"
      region: "east"
  # ...
  ```

  b. 验证标签是否已添加到节点：

  ```
  $ oc get nodes -l type=user-node,region=east
  ```

  输出示例

  ```
  NAME                          STATUS   ROLES    AGE   VERSION
  ip-10-0-142-25.ec2.internal   Ready    worker   17m   v1.29.4
  ```

2. 将匹配的节点选择器添加到 pod：

- 要将节点选择器添加到现有和未来的 pod，请向 pod 的控制对象添加节点选择器：

  带有标签的 ReplicaSet 对象示例

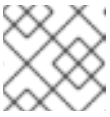
```
kind: ReplicaSet
apiVersion: apps/v1
metadata:
  name: hello-node-6fbccf8d9
# ...
spec:
# ...
  template:
    metadata:
      creationTimestamp: null
      labels:
        ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
        pod-template-hash: 66d5cf9464
    spec:
      nodeSelector:
        kubernetes.io/os: linux
        node-role.kubernetes.io/worker: ''
        type: user-node 1
# ...
```

  1 添加节点选择器。

- 要将节点选择器添加到一个特定的新 pod，直接将选择器添加到 **Pod** 对象中：

  使用节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-node-6fbccf8d9
# ...
spec:
  nodeSelector:
    region: east
    type: user-node
# ...
```



注意

您不能直接将节点选择器添加到现有调度的 pod 中。

4.7.3. 创建默认的集群范围节点选择器

您可以组合使用 pod 上的默认集群范围节点选择器和节点上的标签，将集群中创建的所有 pod 限制到特定节点。

使用集群范围节点选择器时，如果您在集群中创建 pod,OpenShift Container Platform 会将默认节点选择器添加到 pod，并将该 pod 调度到具有匹配标签的节点。

您可以通过编辑调度程序 Operator 自定义资源（CR）来配置集群范围节点选择器。您可向节点、计算机器集或机器配置添加标签。将标签添加到计算机器集可确保节点或机器停机时，新节点具有该标签。如果节点或机器停机，添加到节点或机器配置的标签不会保留。

注意

您可以向 pod 添加额外的键/值对。但是，您无法为一个默认的键添加不同的值。

流程

添加默认的集群范围节点选择器：

1. 编辑调度程序 Operator CR 以添加默认的集群范围节点选择器：

```
$ oc edit scheduler cluster
```

使用节点选择器的调度程序 Operator CR 示例

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
...
spec:
  defaultNodeSelector: type=user-node,region=east (1)
  mastersSchedulable: false
```

(1) 使用适当的 **<key>:<value>** 对添加节点选择器。

完成此更改后，请等待重新部署 **openshift-kube-apiserver** 项目中的 pod。这可能需要几分钟。只有重新部署 pod 后，默认的集群范围节点选择器才会生效。

2. 使用计算机器集或直接编辑节点，为节点添加标签：

- 在创建节点时，使用计算机器集向由计算机器设置管理的节点添加标签：

a. 运行以下命令，将标签添加到 **MachineSet** 对象中：

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>"="
<value>","<key>"="<value>"}}]'  -n openshift-machine-api (1)
```

(1) 为每个标识添加 **<key>/<value>** 对。

例如：

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]'  -n openshift-machine-api
```

提示

您还可以应用以下 YAML 来向计算机器集中添加标签：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  template:
    spec:
      metadata:
        labels:
          region: "east"
          type: "user-node"
```

b. 使用 **oc edit** 命令验证标签是否已添加到 **MachineSet** 对象中：
例如：

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

MachineSet 对象示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
  ...
spec:
  ...
  template:
    metadata:
  ...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
  ...
```

c. 通过缩减至 **0** 并扩展节点来重新部署与该计算机器集关联的节点：
例如：

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. 当节点就绪并可用时，使用 **oc get** 命令验证该标签是否已添加到节点：

```
$ oc get nodes -l <key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node
```

输出示例

```
NAME                                        STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp   Ready    worker   61s   v1.29.4
```

- 直接向节点添加标签：

  a. 为节点编辑 **Node** 对象：

  ```
  $ oc label nodes <name> <key>=<value>
  ```

  例如，若要为以下节点添加标签：

  ```
  $ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 type=user-node
  region=east
  ```

  提示

  您还可以应用以下 YAML 来向节点添加标签：

  ```
  kind: Node
  apiVersion: v1
  metadata:
    name: <node_name>
    labels:
      type: "user-node"
      region: "east"
  ```

  b. 使用 **oc get** 命令验证标签是否已添加到节点：

  ```
  $ oc get nodes -l <key>=<value>,<key>=<value>
  ```

  例如：

  ```
  $ oc get nodes -l type=user-node,region=east
  ```

  输出示例

  ```
  NAME                                        STATUS   ROLES    AGE   VERSION
  ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49   Ready    worker   17m   v1.29.4
  ```

4.7.4. 创建项目范围节点选择器

您可以组合使用项目中的节点选择器和节点上的标签，将该项目中创建的所有 pod 限制到标记的节点。

当您在这个项目中创建 pod 时，OpenShift Container Platform 会将节点选择器添加到项目中 pod，并将 pod 调度到项目中具有匹配标签的节点。如果存在集群范围默认节点选择器，则以项目节点选择器为准。

您可以通过编辑 **Namespace** 对象来向项目添加节点选择器，以添加 **openshift.io/node-selector** 参数。您可向节点、计算机器集或机器配置添加标签。将标签添加到计算机器集可确保节点或机器停机时，新节点具有该标签。如果节点或机器停机，添加到节点或机器配置的标签不会保留。

如果 **Pod** 对象包含节点选择器，则不会调度 pod，但没有项目具有匹配的节点选择器。从该 spec 创建 pod 时，您收到类似以下消息的错误：

错误信息示例

```
Error from server (Forbidden): error when creating "pod.yaml": pods "pod-4" is forbidden: pod node label selector conflicts with its project node label selector
```

注意

您可以向 pod 添加额外的键/值对。但是，您无法为一个项目键添加其他值。

流程

添加默认项目节点选择器：

1. 创建命名空间或编辑现有命名空间，以添加 **openshift.io/node-selector** 参数：

```
$ oc edit namespace <name>
```

输出示例

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    openshift.io/node-selector: "type=user-node,region=east" 1
    openshift.io/description: ""
    openshift.io/display-name: ""
    openshift.io/requester: kube:admin
    openshift.io/sa.scc.mcs: s0:c30,c5
    openshift.io/sa.scc.supplemental-groups: 1000880000/10000
    openshift.io/sa.scc.uid-range: 1000880000/10000
  creationTimestamp: "2021-05-10T12:35:04Z"
  labels:
    kubernetes.io/metadata.name: demo
  name: demo
  resourceVersion: "145537"
  uid: 3f8786e3-1fcb-42e3-a0e3-e2ac54d15001
spec:
  finalizers:
  - kubernetes
```

1 使用适当的 **<key>:<value>** 对添加 **openshift.io/node-selector**。

2. 使用计算机器集或直接编辑节点，为节点添加标签：

- 在创建节点时，使用 **MachineSet** 对象向由计算机器集管理的节点添加标签：

a. 运行以下命令，将标签添加到 **MachineSet** 对象中：

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>","<key>"="<value>"}}]'  -n openshift-machine-api
```

例如：

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]'  -n openshift-machine-api
```

提示

您还可以应用以下 YAML 来向计算机器集中添加标签：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  template:
    spec:
      metadata:
        labels:
          region: "east"
          type: "user-node"
```

b. 使用 **oc edit** 命令验证标签是否已添加到 **MachineSet** 对象中：
例如：

```
$ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

输出示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
...
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
```

c. 重新部署与该计算机器集关联的节点：
例如：

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

d. 当节点就绪并可用时，使用 **oc get** 命令验证该标签是否已添加到节点：

```
$ oc get nodes -l <key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp   Ready    worker   61s   v1.29.4
```

- 直接向节点添加标签：

a. 编辑 **Node** 对象以添加标签：

```
$ oc label <resource> <name> <key>=<value>
```

例如，若要为以下节点添加标签：

```
$ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-c-tgq49 type=user-node region=east
```

提示

您还可以应用以下 YAML 来向节点添加标签：

```
kind: Node
apiVersion: v1
metadata:
  name: <node_name>
  labels:
    type: "user-node"
    region: "east"
```

b. 使用 **oc get** 命令验证标签是否已添加到 **Node** 对象中：

```
$ oc get nodes -l <key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS   ROLES    AGE   VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49   Ready    worker   17m   v1.29.4
```

其他资源

- 使用节点选择器和容限创建项目

4.8. 使用 POD 拓扑分布限制控制 POD 放置

您可以使用 pod 拓扑分布约束来控制 pod 在节点、区、区域或其他用户定义的拓扑域间的放置。

4.8.1. 关于 pod 拓扑分布限制

通过使用 *pod 拓扑分布约束*，您可以对故障域中的 pod 分布提供精细的控制，以帮助实现高可用性和更有效的资源使用。

OpenShift Container Platform 管理员可以标记节点以提供拓扑信息，如区域、区、节点或其他用户定义域。在节点上设置了这些标签后，用户才能定义 pod 拓扑分布约束，以控制 pod 在这些拓扑域中的放置。

您可以指定哪些 pod 要分组在一起，它们分散到哪些拓扑域以及可以接受的基点。只有同一命名空间中的 pod 在因为约束而分散时才会被匹配和分组。

4.8.2. 配置 pod 拓扑分布限制

以下步骤演示了如何配置 pod 拓扑扩展约束，以根据区分配与指定标签匹配的 pod。

您可以指定多个 pod 拓扑分散约束，但您必须确保它们不会相互冲突。必须满足所有 pod 拓扑分布约束才能放置 pod。

先决条件

- 具有 **cluster-admin** 角色的用户已将所需的标签添加到节点。

流程

1. 创建 **Pod** spec 并指定 pod 拓扑分散约束：

   pod-spec.yaml 文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    region: us-east
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
```

```
topologySpreadConstraints:
- maxSkew: 1 (1)
  topologyKey: topology.kubernetes.io/zone (2)
  whenUnsatisfiable: DoNotSchedule (3)
  labelSelector: (4)
    matchLabels:
      region: us-east (5)
  matchLabelKeys:
    - my-pod-label (6)
containers:
- image: "docker.io/ocpqe/hello-pod"
  name: hello-pod
  securityContext:
    allowPrivilegeEscalation: false
    capabilities:
      drop: [ALL]
```

1. 两个拓扑域间的 pod 数量的最大差别。默认为 **1**，您不能指定 **0** 值。
2. 节点标签的密钥。具有此键和相同值的节点被视为在同一拓扑中。
3. 如果不满足分布式约束，如何处理 pod。默认为 **DoNotSchedule**，它会告诉调度程序不要调度 pod。设置为 **ScheduleAnyway**，它仍然会调度 pod，但调度程序会优先考虑 skew 的根据情况以使集群不要出现不平衡的情况。
4. 匹配此标签选择器的 Pod 在分发时被计算并识别为组，以满足约束要求。确保指定标签选择器，否则就无法匹配 pod。
5. 如果您希望以后正确计数此 Pod 规格，请确保此 **Pod** spec 也会设置其标签选择器来匹配这个标签选择器。
6. 用于选择要计算分布的 pod 的 pod 标签键列表。

2. 创建 pod：

```
$ oc create -f pod-spec.yaml
```

4.8.3. pod 拓扑分布限制示例

以下示例演示了 pod 拓扑分散约束配置。

4.8.3.1. 单个 pod 拓扑分布约束示例

此 **Pod** spec 示例定义了一个 pod 拓扑分散约束。它与标记为 **region: us-east** 的 pod 匹配：在区域间分布，指定 skew **1**，并在不满足这些要求时不调度 pod。

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod
  labels:
    region: us-east
spec:
```

```
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        region: us-east
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

4.8.3.2. 多个 pod 拓扑分布约束示例

此 **Pod** spec 示例定义了两个 pod 拓扑分布限制。在标有 **region: us-east** 的 pod 上匹配：指定 skew **1**，并在不满足这些要求时不调度 pod。

第一个限制基于用户定义的标签 **node** 发布 pod，第二个约束根据用户定义的标签 **rack** 分发 pod。调度 pod 必须满足这两个限制。

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pod-2
  labels:
    region: us-east
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: node
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        region: us-east
  - maxSkew: 1
    topologyKey: rack
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        region: us-east
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
    securityContext:
```

```
    allowPrivilegeEscalation: false
    capabilities:
      drop: [ALL]
```

4.8.4. 其他资源

- 了解如何更新节点上的标签

4.9. DESCHEDULER

4.9.1. Descheduler 概述

调度程序（scheduler）被用来决定最适合托管新 pod 的节点，而 descheduler 可以用来驱除正在运行的 pod，从而使 pod 能够重新调度到更合适的节点上。

4.9.1.1. 关于 descheduler

您可以使用 descheduler 根据特定策略驱除 pod，以便可将 pod 重新调度到更合适的节点上。

descheduler 适合于在以下情况下 处理运行的 pod：

- 节点使用不足或过度使用。
- Pod 和节点关联性要求（如污点或标签）已更改，并且原始的调度不再适合于某些节点。
- 节点失败需要移动 pod。
- 集群中添加了新节点。
- Pod 重启的次数太多。

重要

descheduler 不调度被驱除的 pod。调度被驱除 pod 的任务由调度程序（scheduler）执行。

当 descheduler 决定从节点驱除 pod 时，它会使用以下机制：

- **openshift-*** 和 **kube-system** 命名空间中的 Pod 不会被驱除。
- **priorityClassName** 被设置为 **system-cluster-critical** 或 **system-node-critical** 的关键 pod 不会被驱除。
- 不属于复制控制器、副本集、部署或作业一部分的静态、镜像或独立 pod 不会被驱除，因为这些 pod 不会被重新创建。
- 与守护进程集关联的 pod 不会被驱除。
- 具有本地存储的 Pod 不会被驱除。
- BestEffort pod 会在 Burstable 和 Guaranteed pod 之前被驱除。

- 具有 **descheduler.alpha.kubernetes.io/evict** 注解的所有 pod 类型都可以被驱除。此注解用于覆盖防止驱除的检查，用户可以选择驱除哪些 pod。用户应该知道如何创建 pod 以及是否重新创建 pod。

- 对于受 Pod Disruption Budget (PDB) 限制的 pod，如果进行 deschedule 会违反 Pod disruption budget (PDB)，则 pod 不会被驱除。通过使用驱除子资源来处理 PDB 来驱除 pod 。

4.9.1.2. Descheduler 配置集

以下 descheduler 配置集可用：

AffinityAndTaints

此配置集驱除违反了 pod 间的反关联性、节点关联性和节点污点的 pod。
它启用了以下策略：

- **RemovePodsViolatingInterPodAntiAffinity**：删除违反了 pod 间的反关联性的 pod。

- **RemovePodsViolatingNodeAffinity**：移除违反了节点关联性的 pod。

- **RemovePodsViolatingNodeTaints**：移除违反了节点上的 **NoSchedule** 污点的 pod。
  移除具有节点关联性类型 **requiredDuringSchedulingIgnoredDuringExecution**的 pod。

TopologyAndDuplicates

此配置集会驱除 pod 以努力在节点间平均分配类似的 pod 或相同拓扑域的 pod。
它启用了以下策略：

- **RemovePodsViolatingTopologySpreadConstraint**：找到未平衡的拓扑域，并在 **DoNotSchedule** 约束被违反时尝试从较大的 pod 驱除 pod。

- **RemoveDuplicates**：确保只有一个 pod 与同一节点上运行的副本集、复制控制器、部署或作业相关联。如果存在多个重复的 pod，则这些重复的 pod 会被驱除以更好地在集群中的 pod 分布。

LifecycleAndUtilization

此配置集驱除长时间运行的 pod，并平衡节点之间的资源使用情况。
它启用了以下策略：

- **RemovePodsHavingTooManyRestarts** ：删除容器重启了多次的 pod。
  在所有容器（包括初始容器）中被重启的总数超过 100 次的 Pod 。

- **LowNodeUtilization**：查找使用率不足的节点，并在可能的情况下从其他过度使用的节点中驱除 pod，以希望这些被驱除的 pod 可以在使用率低的节点上被重新创建。
  如果节点的用量低于 20%（CPU、内存和 pod 的数量）,则该节点将被视为使用率不足。

  如果节点的用量超过 50%（CPU、内存和 pod 的数量）,则该节点将被视为过量使用。

- **PodLifeTime**：驱除太老的 pod。
  默认情况下，会删除超过 24 小时的 pod。您可以自定义 pod 生命周期值。

SoftTopologyAndDuplicates

这个配置集与 **TopologyAndDuplicates** 相同，不同之处在于具有软拓扑约束的 pod（如 **whenUnsatisfiable: ScheduleAnyway** ）也被视为驱除。

注意

不要同时启用 **SoftTopologyAndDuplicates** 和 **TopologyAndDuplicates**。启用两者会导致冲突。

EvictPodsWithLocalStorage

此配置集允许具有本地存储的 pod 有资格被驱除。

EvictPodsWithPVC

此配置集允许带有持久性卷声明的 pod 有资格驱除。如果使用 **Kubernetes NFS Subdir External Provisioner**，您必须为安装置备程序的命名空间添加排除的命名空间。

4.9.2. Kube Descheduler Operator 发行注记

Kube Descheduler Operator 允许您驱除 pod，以便可以在更合适的节点上重新调度 pod。

本发行注记介绍了 Kube Descheduler Operator 的开发。

如需更多信息，请参阅关于 descheduler。

4.9.2.1. Kube Descheduler Operator 5.0.1 发行注记

发布日期：24 年 7 月 1 日

以下公告可用于 Kube Descheduler Operator 5.0.1 ：

- RHSA-2024:3617

4.9.2.1.1. 新功能及功能增强

- 现在，您可以在以 FIPS 模式运行的 OpenShift Container Platform 集群中安装和使用 Kube Descheduler Operator。

重要

要为集群启用 FIPS 模式，您必须从配置为以 FIPS 模式操作的 Red Hat Enterprise Linux (RHEL) 计算机运行安装程序。有关在 RHEL 中配置 FIPS 模式的更多信息，请参阅在 FIPS 模式中安装该系统。

当以 FIPS 模式运行 Red Hat Enterprise Linux (RHEL) 或 Red Hat Enterprise Linux CoreOS (RHCOS)时，OpenShift Container Platform 核心组件使用 RHEL 加密库，在 x86_64、ppc64le 和 s390x 架构上提交到 NIST FIPS 140-2/140-3 Validation。

4.9.2.1.2. 程序错误修复

- 此 Kube Descheduler Operator 发行版本解决了几个常见漏洞和暴露(CVE)。

4.9.2.2. Kube Descheduler Operator 5.0.0 发行注记

发布日期： 2024 年 3 月 6 日

以下公告可用于 Kube Descheduler Operator 5.0.0 ：

- RHSA-2024:0302

4.9.2.2.1. 主要变化

- 在这个版本中，Kube Descheduler Operator 提供了独立于 OpenShift Container Platform 次版本流的更新。

4.9.2.2.2. 程序错误修复

- 在以前的版本中，descheduler pod 日志显示以下有关 Operator 版本的警告：**failed to convert Descheduler minor version to float**。在这个版本中，不再显示警告。(OCPBUGS-14042)

4.9.3. 使用 descheduler 驱除 pod

您可以通过安装 Kube Descheduler Operator 并设置所需的配置集和其他自定义在 OpenShift Container Platform 中运行 descheduler。

4.9.3.1. 安装 descheduler

在默认情况下，不提供 descheduler。要启用 descheduler，您必须从 OperatorHub 安装 Kube Descheduler Operator，并启用一个或多个 descheduler 配置集。

默认情况下，descheduler 以预测模式运行，这意味着它只模拟 pod 驱除。您必须将 descheduler 的模式更改为自动进行 pod 驱除。

重要

如果您在集群中启用了托管的 control plane，设置自定义优先级阈值，以降低托管 control plane 命名空间中的 pod 被驱除。将优先级阈值类名称设置为 **hypershift-control-plane**，因为它有托管的 control plane 优先级类的最低优先级值（**100000000**）。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform Web 控制台。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 为 Kube Descheduler Operator 创建所需的命名空间。
   a. 进行 **Administration** → **Namespaces**，点 **Create Namespace**。
   b. 在 **Name** 字段中输入 **openshift-kube-descheduler-operator**，在 **Labels** 字段中输入 **openshift.io/cluster-monitoring=true** 来启用 descheduler 指标，然后点击 **Create**。
3. 安装 Kube Descheduler Operator。
   a. 进入 **Operators** → **OperatorHub**。
   b. 在过滤框中输入 **Kube Descheduler Operator**。
   c. 选择 **Kube Descheduler Operator** 并点 **Install**。

d. 在 **Install Operator** 页面中，选择 **A specific namespace on the cluster**。从下拉菜单中选择 **openshift-kube-descheduler-operator**。

e. 将 **Update Channel** 和 **Approval Strategy** 的值调整为所需的值。

f. 点击 **Install**。

4. 创建 descheduler 实例。

a. 在 **Operators → Installed Operators** 页面中，点 **Kube Descheduler Operator**。

b. 选择 **Kube Descheduler** 标签页并点 **Create KubeDescheduler**。

c. 根据需要编辑设置。

i. 要驱除 pod 而不是模拟驱除，请将 **Mode** 字段更改为 **Automatic**。

4.9.3.2. 配置 descheduler 配置集

您可以配置 descheduler 使用哪些配置集来驱除 pod。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。

流程

1. 编辑 **KubeDescheduler** 对象：

```
$ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
```

2. 在 **spec.profiles** 部分指定一个或多个配置集。

```
apiVersion: operator.openshift.io/v1
kind: KubeDescheduler
metadata:
  name: cluster
  namespace: openshift-kube-descheduler-operator
spec:
  deschedulingIntervalSeconds: 3600
  logLevel: Normal
  managementState: Managed
  operatorLogLevel: Normal
  mode: Predictive                          1
  profileCustomizations:
    namespaces:                             2
      excluded:
      - my-namespace
    podLifetime: 48h                        3
    thresholdPriorityClassName: my-priority-class-name 4
  profiles:                                 5
  - AffinityAndTaints
  - TopologyAndDuplicates                   6
```

```
    - LifecycleAndUtilization
    - EvictPodsWithLocalStorage
    - EvictPodsWithPVC
```

1. 可选：默认情况下，descheduler 不会驱除 pod。要驱除 pod，请将 **mode** 设置为 **Automatic**。
2. 可选：设置用户创建命名空间列表，以便从 descheduler 操作中包含或排除。使用 **exclude** 设置要排除的命名空间列表，或者使用 **included** 来设置要包含的命名空间列表。请注意，默认排除受保护的命名空间(**openshift-***、**kube-system**、**hypershift**)。
3. 可选：为 **LifecycleAndUtilization** 配置集启用自定义 pod 生命周期值。有效单位是 **s**、**m** 或 **h**。默认 pod 生命周期为 24 小时。
4. 可选：指定优先级阈值，仅在优先级低于指定级别时才会考虑 pod 进行驱除。使用 **thresholdPriority** 字段设置数字优先级阈值（如 **10000**）或者使用 **thresholdPriorityClassName** 字段指定特定的优先级类名称（如 **my-priority-class-name**）。如果指定优先级类名称，则必须已存在它，否则 descheduler 会抛出错误。不要同时设置 **thresholdPriority** 和 **thresholdPriorityClassName**。
5. 添加一个或多个配置文件以启用。可用配置集：**AffinityAndTaints**、**TopologyAndDuplicates**、**LifecycleAndUtilization**、**SoftTopologyAndDuplicates**、**EvictPodsWithLocalStorage** 和 **EvictPodsWithPVC**。
6. 不要同时启用 **TopologyAndDuplicates** 和 **SoftTopologyAndDuplicates**。启用两者会导致冲突。

您可以启用多个配置集 ; 指定配置集的顺序并不重要。

3. 保存文件以使改变生效。

4.9.3.3. 配置 descheduler 间隔

您可以配置 descheduler 运行之间的时间长度。默认为 3600 秒（一小时）。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。

流程

1. 编辑 **KubeDescheduler** 对象：

```
$ oc edit kubedeschedulers.operator.openshift.io cluster -n openshift-kube-descheduler-operator
```

2. 将 **deschedulingIntervalSeconds** 字段更新为所需的值：

```
apiVersion: operator.openshift.io/v1
kind: KubeDescheduler
metadata:
  name: cluster
  namespace: openshift-kube-descheduler-operator
```

```
spec:
 deschedulingIntervalSeconds: 3600 (1)
...
```

(1) 设置 descheduler 运行间隔的秒数。如果设为 **0**，则 descheduler 会运行一次并退出。

3. 保存文件以使改变生效。

4.9.4. 卸载 Kube Descheduler Operator

您可以通过卸载 Operator 并删除其相关资源，从 OpenShift Container Platform 中删除 Kube Descheduler Operator。

4.9.4.1. 卸载 descheduler

您可以通过删除 descheduler 实例并卸载 Kube Descheduler Operator 从集群中移除 descheduler。此流程还会清理 **KubeDescheduler** CRD 和 **openshift-kube-descheduler-operator** 命名空间。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform Web 控制台。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 删除 descheduler 实例。
   a. 在 **Operators → Installed Operators** 页面中，点 **Kube Descheduler Operator**。
   b. 选择 **Kube Descheduler** 选项卡。
   c. 点 **集群** 条目旁的 Options 菜单 ⋮ 并选择 **Delete KubeDescheduler**。
   d. 在确认对话框中，点 **Delete**。
3. 卸载 Kube Descheduler Operator。
   a. 导航到 **Operators → Installed Operators**。
   b. 点 **Kube Descheduler Operator** 条目 ⋮ 旁边的 Options 菜单，然后选择 **Uninstall Operator**。
   c. 在确认对话框中，点 **Uninstall**。
4. 删除 **openshift-kube-descheduler-operator** 命名空间。
   a. 导航至 **Administration → Namespaces**。
   b. 在过滤器框中输入 **openshift-kube-descheduler-operator**。

c. 点 **openshift-kube-descheduler-operator** 条目旁的 Options 菜单 ，然后选择 **Delete Namespace**。

d. 在确认对话框中，输入 **openshift-kube-descheduler-operator** 并点 **Delete**。

5. 删除 **KubeDescheduler** CRD。

a. 进入 **Administration** → **Custom Resource Definitions**。

b. 在过滤器框中输入 **KubeDescheduler**。

c. 点 **KubeDescheduler** 条目旁的 Options 菜单 ，然后选择 **Delete CustomResourceDefinition**。

d. 在确认对话框中，点 **Delete**。

4.10. 二级调度程序

4.10.1. 二级调度程序概述

您可以安装 Secondary Scheduler Operator 来运行自定义二级调度程序，以及调度 pod 的默认调度程序。

4.10.1.1. 关于 Secondary Scheduler Operator

Red Hat OpenShift 的 Secondary Scheduler Operator 提供了在 OpenShift Container Platform 中部署自定义二级调度程序的方法。二级调度程序与默认调度程序一起运行，以调度 pod。Pod 配置可指定要使用的调度程序。

自定义调度程序必须具有 **/bin/kube-scheduler** 二进制文件，并基于 Kubernetes 调度框架。

重要

您可以使用 Secondary Scheduler Operator 在 OpenShift Container Platform 中部署自定义二级调度程序，但红帽不直接支持自定义二级调度程序的功能。

Secondary Scheduler Operator 创建二级调度程序所需的默认角色和角色绑定。您可以通过为从属调度程序配置 **KubeSchedulerConfiguration** 资源，来指定哪些调度插件来启用或禁用。

4.10.2. Secondary Scheduler Operator for Red Hat OpenShift 发行注记

Red Hat OpenShift 的 Secondary Scheduler Operator 允许您在 OpenShift Container Platform 集群中部署自定义二级调度程序。

本发行注记介绍了针对 Red Hat OpenShift 的 Secondary Scheduler Operator 的开发。

如需更多信息，请参阅关于 Secondary Scheduler Operator 。

4.10.2.1. Red Hat OpenShift 1.3.0 的 Secondary Scheduler Operator 发行注记

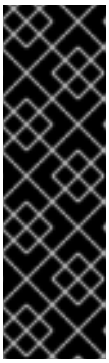
发布日期：24 年 7 月 1 日

以下公告可用于 Red Hat OpenShift 1.3.0 的 Secondary Scheduler Operator：

- RHSA-2024:3637

4.10.2.1.1. 新功能及功能增强

- 现在，您可以在以 FIPS 模式运行的 OpenShift Container Platform 集群中安装和使用 Secondary Scheduler Operator。



重要

要为集群启用 FIPS 模式，您必须从配置为以 FIPS 模式操作的 Red Hat Enterprise Linux (RHEL) 计算机运行安装程序。有关在 RHEL 中配置 FIPS 模式的更多信息，请参阅在 FIPS 模式中安装该系统。

当以 FIPS 模式运行 Red Hat Enterprise Linux (RHEL) 或 Red Hat Enterprise Linux CoreOS (RHCOS)时，OpenShift Container Platform 核心组件使用 RHEL 加密库，在 x86_64、ppc64le 和 s390x 架构上提交到 NIST FIPS 140-2/140-3 Validation。

4.10.2.1.2. 程序错误修复

- 此 Secondary Scheduler Operator 发行版本解决了几个常见漏洞和暴露 (CVE)。

4.10.2.1.3. 已知问题

- 目前，您无法通过 Secondary Scheduler Operator 部署其他资源，如配置映射、CRD 或 RBAC 策略。自定义二级调度程序所需的角色和角色绑定以外的任何资源都必须外部应用。(WRKLDS-645)

4.10.3. 使用二级调度程序调度 pod

您可以通过安装 Secondary Scheduler Operator、部署二级调度程序，并在 pod 定义中设置二级调度程序，在 OpenShift Container Platform 中运行自定义次要调度程序。

4.10.3.1. 安装 Secondary Scheduler Operator

您可以使用 Web 控制台为 Red Hat OpenShift 安装 Secondary Scheduler Operator。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform web 控制台。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 为 Red Hat OpenShift 创建 Secondary Scheduler Operator 所需的命名空间。
   a. 进行 **Administration → Namespaces**，点 **Create Namespace**。
   b. 在 **Name** 字段中输入 **openshift-secondary-scheduler-operator**，再点 **Create**。

3. 为 Red Hat OpenShift 安装 Secondary Scheduler Operator。

   a. 导航至 **Operators → OperatorHub**。

   b. 在过滤器框中输入 **Secondary Scheduler Operator for Red Hat OpenShift**。

   c. 选择 **Secondary Scheduler Operator for Red Hat OpenShift**并点 **Install**。

   d. 在 **Install Operator** 页面中：

      i. **Update channel** 设置为 **stable**，它将为 Red Hat OpenShift 安装 Secondary Scheduler Operator 的最新稳定版本。

      ii. 选择 **A specific namespace on the cluster**，并从下拉菜单中选择 **openshift-secondary-scheduler-operator**。

      iii. 选择一个 **更新批准策略**。

         - **Automatic** 策略允许 Operator Lifecycle Manager（OLM）在有新版本可用时自动更新 Operator。
         - **Manual** 策略需要拥有适当凭证的用户批准 Operator 更新。

      iv. 点 **Install**。

验证

1. 导航到 **Operators → Installed Operators**。
2. 验证 **Secondary Scheduler Operator for Red Hat OpenShift**已列出，**Status** 为 **Succeeded**。

4.10.3.2. 部署二级调度程序

安装 Secondary Scheduler Operator 后，您可以部署二级调度程序。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform web 控制台。
- 安装了 Secondary Scheduler Operator for Red Hat OpenShift。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 创建配置映射来保存二级调度程序的配置。

   a. 进入 **Workloads → ConfigMaps**。

   b. 点 **Create ConfigMap**。

   c. 在 YAML 编辑器中，输入包含必要 **KubeSchedulerConfiguration** 配置的配置映射定义。例如：

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: "secondary-scheduler-config"                    1
  namespace: "openshift-secondary-scheduler-operator"   2
data:
  "config.yaml": |
    apiVersion: kubescheduler.config.k8s.io/v1beta3
    kind: KubeSchedulerConfiguration                    3
    leaderElection:
      leaderElect: false
    profiles:
      - schedulerName: secondary-scheduler              4
        plugins:                                        5
          score:
            disabled:
              - name: NodeResourcesBalancedAllocation
              - name: NodeResourcesLeastAllocated
```

1. 配置映射的名称。这将在创建 **SecondaryScheduler** CR 时，在 **Scheduler Config** 字段中使用。
2. 配置映射必须在 **openshift-secondary-scheduler-operator** 命名空间中创建。
3. 二级调度程序的 **KubeSchedulerConfiguration** 资源。如需更多信息，请参阅 Kubernetes API 文档中的 **KubeSchedulerConfiguration**。
4. 二级调度程序的名称。将其 **spec.schedulerName** 字段设置为此值的 Pod 会使用这个二级调度程序来调度。
5. 为二级调度程序启用或禁用插件。如需列出默认调度插件，请参阅 Kubernetes 文档中的 调度插件。

d. 点 **Create**。

3. 创建 **SecondaryScheduler** CR:

a. 导航到 **Operators → Installed Operators**。

b. 选择 **Secondary Scheduler Operator for Red Hat OpenShift**。

c. 选择 **Secondary Scheduler** 选项卡，然后点 **Create SecondaryScheduler**。

d. **Name** 字段默认为 **cluster**; 不要更改此名称。

e. **Scheduler Config** 字段默认为 **secondary-scheduler-config**。确保这个值与此流程中创建的配置映射的名称匹配。

f. 在 **Scheduler Image** 字段中，输入自定义调度程序的镜像名称。

重要

红帽不直接支持自定义二级调度程序的功能。

g. 点 **Create**。

4.10.3.3. 使用二级调度程序调度 pod

要使用二级调度程序调度 pod，请在 pod 定义中设置 **schedulerName** 字段。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform web 控制台。
- 安装了 Secondary Scheduler Operator for Red Hat OpenShift。
- 配置了二级调度程序。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 导航到 **Workloads → Pods**。
3. 点 **Create Pod**。
4. 在 YAML 编辑器中，输入所需的 pod 配置并添加 **schedulerName** 字段：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  schedulerName: secondary-scheduler (1)
```

(1) 在配置二级调度程序时，**schedulerName** 字段必须与配置映射中定义的名称匹配。

5. 点 **Create**。

验证

1. 登录到 OpenShift CLI。
2. 使用以下命令描述 pod：

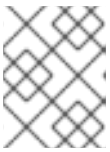
```
$ oc describe pod nginx -n default
```

输出示例

```
Name:         nginx
Namespace:    default
Priority:     0
Node:         ci-ln-t0w4r1k-72292-xkqs4-worker-b-xqkxp/10.0.128.3
...
Events:
  Type    Reason          Age   From                Message
  ----    ------          ----  ----                -------
  Normal  Scheduled       12s   secondary-scheduler  Successfully assigned default/nginx to
ci-ln-t0w4r1k-72292-xkqs4-worker-b-xqkxp
...
```

3. 在事件表中，找到带有与 **Successfully assigned <namespace>/<pod_name> to <node_name>** 类似消息的事件。

4. 在 "From" 列中，验证事件是从二级调度程序生成的，而不是默认调度程序。



注意

您还可以检查 **openshift-secondary-scheduler-namespace** 中的 **secondary-scheduler-*** pod 日志，以验证 pod 是否已由二级调度程序调度。

4.10.4. 卸载 Secondary Scheduler Operator

您可以通过卸载 Operator 并删除其相关资源，从 OpenShift Container Platform 中删除 Red Hat OpenShift 的 Secondary Scheduler Operator。

4.10.4.1. 卸载 Secondary Scheduler Operator

您可以使用 Web 控制台为 Red Hat OpenShift 卸载 Secondary Scheduler Operator。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform web 控制台。
- 安装了 Secondary Scheduler Operator for Red Hat OpenShift。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 为 Red Hat OpenShift Operator 卸载 Secondary Scheduler Operator。
   a. 导航到 **Operators → Installed Operators**。

b. 点 **Secondary Scheduler Operator** 条目旁边的 Options 菜单  ，并点 **Uninstall Operator**。

c. 在确认对话框中，点 **Uninstall**。

4.10.4.2. 删除 Secondary Scheduler Operator 资源

另外，在为 Red Hat OpenShift 卸载 Secondary Scheduler Operator 后，您可以从集群中移除其相关资源。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。
- 访问 OpenShift Container Platform web 控制台。

流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 删除由 Secondary Scheduler Operator 安装的 CRD：
   a. 进入到 **Administration → CustomResourceDefinitions**。
   b. 在 **Name** 字段中输入 **SecondaryScheduler** 以过滤 CRD。
   c. 点 **SecondaryScheduler** CRD 旁边的 Options 菜单  并选择 **Delete Custom Resource Definition**:
3. 删除 **openshift-secondary-scheduler-operator** 命名空间。
   a. 导航至 **Administration → Namespaces**。
   b. 点 **openshift-secondary-scheduler-operator** 旁边的 Options 菜单  并选择 **Delete Namespace**。
   c. 在确认对话框中，在字段中输入 **openshift-secondary-scheduler-operator**，再点 **Delete**。

第 5 章 使用作业和 DAEMONSET

5.1. 使用 DAEMONSET 在节点上自动运行后台任务

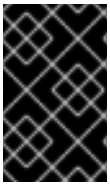
作为管理员，您可以创建并使用守护进程集在 OpenShift Container Platform 集群的特定节点或所有节点上运行 pod 副本。

守护进程集确保所有（或部分）节点都运行 pod 的副本。当节点添加到集群中时，pod 也会添加到集群中。当节点从集群中移除时，这些 pod 也会通过垃圾回收而被移除。删除守护进程集会清理它创建的 pod。

您可以使用 daemonset 创建共享存储，在集群的每一节点上运行日志 pod，或者在每个节点上部署监控代理。

为安全起见，集群管理员和项目管理员可以创建守护进程集。

如需有关守护进程集的更多信息，请参阅 Kubernetes 文档。



重要

守护进程集调度与项目的默认节点选择器不兼容。如果您没有禁用它，守护进程集会与默认节点选择器合并，从而受到限制。这会造成在合并后节点选择器没有选中的节点上频繁地重新创建 pod，进而给集群带来意外的负载。

5.1.1. 通过默认调度程序调度

守护进程集确保所有有资格的节点都运行 pod 的副本。通常，Kubernetes 调度程序会选择要在其上运行 pod 的节点。但是，守护进程集 pod 由守护进程集控制器创建并调度。这会引发以下问题：

- pod 行为不一致：等待调度的普通 pod 被创建好并处于待处理状态，但守护进程集 pod 没有以**待处理**的状态创建。这会给用户造成混淆。
- Pod 抢占由默认调度程序处理。启用抢占后，守护进程集控制器将在不考虑 pod 优先级和抢占的前提下做出调度决策。

OpenShift Container Platform 中默认启用 **ScheduleDaemonSetPods** 功能允许您使用默认调度程序而不是守护进程集控制器来调度守护进程集，具体方法是添加 **NodeAffinity** 术语到守护进程集 pod，而不是 **.spec.nodeName** 术语。然后，默认调度程序用于将 pod 绑定到目标主机。如果守护进程集的节点关联性已经存在，它会被替换掉。守护进程设置控制器仅在创建或修改守护进程集 pod 时执行这些操作，且不会对守护进程集的 **spec.template** 进行任何更改。

```
kind: Pod
apiVersion: v1
metadata:
  name: hello-node-6fbccf8d9-9tmzr
#...
spec:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchFields:
        - key: metadata.name
          operator: In
```

```
        values:
        - target-host-name
#...
```

另外，**node.kubernetes.io/unschedulable:NoSchedule** 容限会自动添加到守护进程设置 Pod 中。在调度守护进程设置 pod 时，默认调度程序会忽略不可调度的节点。

5.1.2. 创建 daemonset

在创建守护进程集时，使用 **nodeSelector** 字段来指示守护进程集应在其上部署副本的节点。

先决条件

- 在开始使用守护进程集之前，通过将命名空间注解 **openshift.io/node-selector** 设置为空字符串来禁用命名空间中的默认项目范围节点选择器：

```
$ oc patch namespace myproject -p \
    '{"metadata": {"annotations": {"openshift.io/node-selector": ""}}}'
```

提示

您还可以应用以下 YAML 来为命名空间禁用默认的项目范围节点选择器：

```
apiVersion: v1
kind: Namespace
metadata:
  name: <namespace>
  annotations:
    openshift.io/node-selector: ''
#...
```

- 如果您要创建新项目，请覆盖默认节点选择器：

```
$ oc adm new-project <name> --node-selector=""
```

流程

创建守护进程集：

1. 定义守护进程集 yaml 文件：

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
     matchLabels:
       name: hello-daemonset 1
  template:
    metadata:
      labels:
        name: hello-daemonset 2
```

```
    spec:
      nodeSelector: 3
        role: worker
      containers:
      - image: openshift/hello-openshift
        imagePullPolicy: Always
        name: registry
        ports:
        - containerPort: 80
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
      serviceAccount: default
      terminationGracePeriodSeconds: 10
#...
```

1 决定哪些 pod 属于守护进程集的标签选择器。

2 pod 模板的标签选择器。必须与上述标签选择器匹配。

3 决定应该在哪些节点上部署 pod 副本的节点选择器。节点上必须存在匹配的标签。

2. 创建守护进程集对象：

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

3. 验证 pod 是否已创建好，并且每个节点都有 pod 副本：

   a. 查找 daemonset pod：

   ```
   $ oc get pods
   ```

   输出示例

   ```
   hello-daemonset-cx6md   1/1       Running   0          2m
   hello-daemonset-e3md9   1/1       Running   0          2m
   ```

   b. 查看 pod 以验证 pod 已放置到节点上：

   ```
   $ oc describe pod/hello-daemonset-cx6md|grep Node
   ```

   输出示例

   ```
   Node:        openshift-node01.hostname.com/10.14.20.134
   ```

   ```
   $ oc describe pod/hello-daemonset-e3md9|grep Node
   ```

   输出示例

   ```
   Node:        openshift-node02.hostname.com/10.14.20.137
   ```

重要

- 如果更新守护进程设置的 pod 模板，现有的 pod 副本不会受到影响。
- 如果您删除了守护进程集，然后在创建新守护进程集时使用不同的模板和相同的标签选择器，它会将现有 pod 副本识别为具有匹配的标签，因而不更新它们，也不会创建新的副本，尽管 pod 模板中存在不匹配。
- 如果您更改了节点标签，守护进程集会把 pod 添加到与新标签匹配的节点，并从不匹配新标签的节点中删除 pod。

要更新守护进程集，请通过删除旧副本或节点来强制创建新的 pod 副本。

5.2. 使用任务在 POD 中运行任务

*作业（job）*在 OpenShift Container Platform 集群中执行某项任务。

作业会跟踪任务的整体进度，并使用活跃、成功和失败 pod 的相关信息来更新其状态。删除作业会清理它创建的所有 pod 副本。作业是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

作业规格示例

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  activeDeadlineSeconds: 1800 (3)
  backoffLimit: 6   (4)
  template:         (5)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (6)
#...
```

1. 作业应并行运行的 pod 副本 。
2. pod 成功完成后需要标记为作业也完成。
3. 作业可以运行的最长时间。
4. 作业的重试次数。
5. 控制器创建的 pod 模板。
6. pod 的重启策略。

其他资源

- Kubernetes 文档中的作业

5.2.1. 了解作业和 cron 作业

作业会跟踪任务的整体进度，并使用活跃、成功和失败 pod 的相关信息来更新其状态。删除作业会清理它创建的所有 pod。作业是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

OpenShift Container Platform 中有两种资源类型可以创建只运行一次的对象：

作业

常规作业是一种只运行一次的对象，它会创建一个任务并确保作业完成。
有三种适合作为作业运行的任务类型：

- 非并行作业：
  - 仅启动一个 pod 的作业，除非 pod 失败。
  - 一旦 pod 成功终止，作业就会马上完成。
- 带有固定完成计数的并行作业：
  - 启动多个 pod 的作业。
  - Job 代表整个任务，并在 **1** 到 **completions** 范围内的每个值都有一个成功 pod 时完成 。
- 带有工作队列的并行作业：
  - 在一个给定 pod 中具有多个并行 worker 进程的作业。
  - OpenShift Container Platform 协调 pod，以确定每个 pod 都应该使用什么作业，或使用一个外部队列服务。
  - 每个 pod 都可以独立决定是否所有对等 pod 都已完成（整个作业完成）。
  - 当所有来自作业的 pod 都成功终止时，不会创建新的 pod。
  - 当至少有一个 pod 成功终止并且所有 pod 都终止时，作业成功完成。
  - 当任何 pod 成功退出时，其他 pod 都不应该为这个任务做任何工作或写任何输出。Pod 都应该处于退出过程中。
    如需有关如何使用不同类型的作业的更多信息，请参阅 Kubernetes 文档中的作业模式 。

Cron job

通过使用 Cron Job，一个作业可以被调度为运行多次。
Cron Job 基于常规作业构建，允许您指定作业的运行方式。Cron job 是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

Cron Job 可用于创建周期性和重复执行的任务，如运行备份或发送电子邮件。Cron Job 也可以将个别任务调度到指定时间执行，例如，将一个作业调度到低活动时段执行。一个 cron 作业会创建一个 **Job** 对象，它基于在运行 cronjob 的 control plane 节点上配置的时区。

警告

Cron Job 大致会在调度的每个执行时间创建一个 **Job** 对象，但在有些情况下，它可能无法创建作业，或者可能会创建两个作业。因此，作业必须具有幂等性，而且您必须配置历史限制。

5.2.1.1. 了解如何创建作业

两种资源类型都需要一个由以下关键部分组成的作业配置：

- pod 模板，用于描述 OpenShift Container Platform 创建的 pod。
- **parallelism** 参数，用于指定在任意时间点上应并行运行多少个 pod 来执行某个作业。
  - 对于非并行作业，请保留未设置。当取消设置时，默认为 **1**。
- **completions** 参数，用于指定需要成功完成多少个 pod 才能完成某个作业。
  - 对于非并行作业，请保留未设置。当取消设置时，默认为 **1**。
  - 对于带有固定完成计数的并行作业，请指定一个值。
  - 对于带有工作队列的并行作业，请保留 unset。当取消设置默认为 **parallelism** 值。

5.2.1.2. 了解如何为作业设置最长持续时间

在定义作业时，您可以通过设置 **activeDeadlineSeconds** 字段来定义其最长持续时间。以秒为单位指定，默认情况下不设置。若未设置，则不强制执行最长持续时间。

最长持续时间从系统中调度第一个 pod 的时间开始计算，并且定义作业在多久时间内处于活跃状态。它将跟踪整个执行时间。达到指定的超时后，OpenShift Container Platform 将终止作业。

5.2.1.3. 了解如何为 pod 失败设置作业避退策略

在因为配置中的逻辑错误或其他类似原因而重试了一定次数后，作业会被视为已经失败。控制器以六分钟为上限，按指数避退延时（**10s**，**20s**，**40s** ...）重新创建与作业关联的失败 pod。如果控制器检查之间没有出现新的失败 pod，则重置这个限制。

使用 **spec.backoffLimit** 参数为作业设置重试次数。

5.2.1.4. 了解如何配置 Cron Job 以移除工件

Cron Job 可能会遗留工件资源，如作业或 pod 等。作为用户，务必要配置一个历史限制，以便能妥善清理旧作业及其 pod。Cron Job 规格内有两个字段负责这一事务：

- **.spec.successfulJobsHistoryLimit**。要保留的成功完成作业数（默认为 3）。
- **.spec.failedJobsHistoryLimit**。要保留的失败完成作业数（默认为 1）。

提示

- 删除您不再需要的 Cron Job：

  ```
  $ oc delete cronjob/<cron_job_name>
  ```

  这样可防止生成不必要的工件。

- 您可以通过将 **spec.suspend** 设置为 true 来挂起后续执行。所有后续执行都会挂起，直到重置为 **false**。

5.2.1.5. 已知限制

作业规格重启策略只适用于 *pod*，不适用于 *作业控制器*。不过，作业控制器被硬编码为可以一直重试直到作业完成为止。

因此，**restartPolicy: Never** 或 **--restart=Never** 会产生与 **restartPolicy: OnFailure** 或 **--restart=OnFailure** 相同的行为。也就是说，作业失败后会自动重启，直到成功（或被手动放弃）为止。策略仅设定由哪一子系统执行重启。

使用 **Never** 策略时，*作业控制器*负责执行重启。在每次尝试时，作业控制器会在作业状态中递增失败次数并创建新的 pod。这意味着，每次尝试失败都会增加 pod 的数量。

使用 **OnFailure** 策略时，*kubelet* 负责执行重启。每次尝试都不会在作业状态中递增失败次数。另外，kubelet 将通过在相同节点上启动 pod 来重试失败的作业。

5.2.2. 创建作业

您可以通过创建作业对象在 OpenShift Container Platform 中创建作业。

流程

创建作业：

1. 创建一个类似以下示例的 YAML 文件：

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  activeDeadlineSeconds: 1800  (3)
  backoffLimit: 6   (4)
  template:         (5)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (6)
#...
```

1. 可选：指定一个作业应并行运行多少个 pod 副本；默认与 **1**。
   - 对于非并行作业，请保留未设置。当取消设置时，默认为 **1**。
2. 可选：指定标记作业完成需要成功完成多少个 pod。
   - 对于非并行作业，请保留未设置。当取消设置时，默认为 **1**。
   - 对于具有固定完成计数的并行作业，请指定完成数。
   - 对于带有工作队列的并行作业，请保留 unset。当取消设置默认为 **parallelism** 值。
3. 可选：指定作业可以运行的最长持续时间。
4. 可选：指定作业的重试次数。此字段默认值为 6。
5. 指定控制器创建的 Pod 模板。
6. 指定 pod 的重启策略：
   - **Never**。不要重启作业。
   - **OnFailure**。仅在失败时重启该任务。
   - **Always**。总是重启该任务。
     如需了解 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。

2. 创建作业：

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

注意

您还可以使用 **oc create job**，在一个命令中创建并启动作业。以下命令会创建并启动一个与上个示例中指定的相似的作业：

```
$ oc create job pi --image=perl -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

5.2.3. 创建 cron job

您可以通过创建作业对象在 OpenShift Container Platform 中创建 Cron Job。

流程

创建 Cron Job：

1. 创建一个类似以下示例的 YAML 文件：

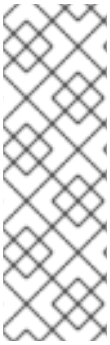
```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: pi
spec:
```

```
  schedule: "*/1 * * * *" 1
  timeZone: Etc/UTC 2
  concurrencyPolicy: "Replace" 3
  startingDeadlineSeconds: 200 4
  suspend: true 5
  successfulJobsHistoryLimit: 3 6
  failedJobsHistoryLimit: 1 7
  jobTemplate: 8
    spec:
      template:
        metadata:
          labels: 9
            parent: "cronjobpi"
        spec:
          containers:
          - name: pi
            image: perl
            command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
          restartPolicy: OnFailure 10
#...
```

1. 以 cron 格式指定的作业调度计划。在本例中，作业将每分钟运行一次。
2. 调度的可选时区。有关有效选项，请参阅 tz 数据库时区列表。如果没有指定，Kubernetes 控制器管理器会解释相对于其本地时区的调度。
3. 可选的并发策略，指定如何对待 Cron Job 中的并发作业。只能指定以下并发策略之一。若未指定，默认为允许并发执行。
   - **Allow**，允许 Cron Job 并发运行。
   - **Forbid**，禁止并发运行。如果上一运行尚未结束，则跳过下一运行。
   - **Replace**，取消当前运行的作业并替换为新作业。
4. 可选期限（秒为单位），如果作业因任何原因而错过预定时间，则在此期限内启动作业。错过的作业执行计为失败的作业。若不指定，则没有期限。
5. 可选标志，允许挂起 Cron Job。若设为 **true**，则会挂起所有后续执行。
6. 要保留的成功完成作业数（默认为 3）。
7. 要保留的失败完成作业数（默认为 1）。
8. 作业模板。类似于作业示例。
9. 为此 Cron Job 生成的作业设置一个标签。
10. pod 的重启策略。这不适用于作业控制器。

2. 创建 cron job：

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



注意

您还可以使用 **oc create cronjob**，在一个命令中创建并启动 Cron Job。以下命令会创建并启动与上一示例中指定的相似的 Cron Job：

```
$ oc create cronjob pi --image=perl --schedule='*/1 * * * *' -- perl -Mbignum=bpi -wle
'print bpi(2000)'
```

使用 **oc create cronjob** 时，**--schedule** 选项接受采用 cron 格式的调度计划。

第 6 章 操作节点

6.1. 查看和列出 OPENSHIFT CONTAINER PLATFORM 集群中的节点

您可以列出集群中的所有节点，以获取节点的相关信息，如状态、年龄、内存用量和其他详情。

在执行节点管理操作时，CLI 与代表实际节点主机的节点对象交互。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。

6.1.1. 关于列出集群中的所有节点

您可以获取集群中节点的详细信息。

- 以下命令列出所有节点：

  ```
  $ oc get nodes
  ```

  以下示例是具有健康节点的集群：

  ```
  $ oc get nodes
  ```

  输出示例

  ```
  NAME                 STATUS    ROLES     AGE       VERSION
  master.example.com   Ready     master    7h        v1.29.4
  node1.example.com    Ready     worker    7h        v1.29.4
  node2.example.com    Ready     worker    7h        v1.29.4
  ```

  以下示例是具有一个不健康节点的集群：

  ```
  $ oc get nodes
  ```

  输出示例

  ```
  NAME                 STATUS                     ROLES     AGE       VERSION
  master.example.com   Ready                      master    7h        v1.29.4
  node1.example.com    NotReady,SchedulingDisabled worker    7h        v1.29.4
  node2.example.com    Ready                      worker    7h        v1.29.4
  ```

  触发 **NotReady** 状态的条件在本节中显示。

- **-o wide** 选项提供有关节点的附加信息。

  ```
  $ oc get nodes -o wide
  ```

  输出示例

  ```
  NAME                 STATUS   ROLES    AGE    VERSION   INTERNAL-IP    EXTERNAL-IP
  OS-IMAGE                                  KERNEL-VERSION            CONTAINER-
  RUNTIME
  master.example.com   Ready    master   171m   v1.29.4   10.0.129.108   <none>        Red Hat
  ```

```
Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-240.15.1.el8_3.x86_64
cri-o://1.29.4-30.rhaos4.10.gitf2f339d.el8-dev
node1.example.com   Ready   worker   72m   v1.29.4   10.0.129.222   <none>        Red Hat
Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-240.15.1.el8_3.x86_64
cri-o://1.29.4-30.rhaos4.10.gitf2f339d.el8-dev
node2.example.com   Ready   worker   164m   v1.29.4   10.0.142.150   <none>        Red Hat
Enterprise Linux CoreOS 48.83.202103210901-0 (Ootpa)   4.18.0-240.15.1.el8_3.x86_64
cri-o://1.29.4-30.rhaos4.10.gitf2f339d.el8-dev
```

- 以下命令列出一个节点的相关信息：

```
$ oc get node <node>
```

例如：

```
$ oc get node node1.example.com
```

输出示例

```
NAME                STATUS   ROLES    AGE     VERSION
node1.example.com   Ready    worker   7h      v1.29.4
```

- 以下命令提供有关特定节点的更多详细信息，包括发生当前状况的原因：

```
$ oc describe node <node>
```

例如：

```
$ oc describe node node1.example.com
```

注意

以下示例包含特定于 AWS 上的 OpenShift Container Platform 的一些值。

输出示例

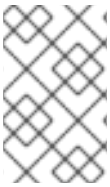
```
Name:               node1.example.com 1
Roles:              worker 2
Labels:             kubernetes.io/os=linux
                    kubernetes.io/hostname=ip-10-0-131-14
                    kubernetes.io/arch=amd64 3
                    node-role.kubernetes.io/worker=
                    node.kubernetes.io/instance-type=m4.large
                    node.openshift.io/os_id=rhcos
                    node.openshift.io/os_version=4.5
                    region=east
                    topology.kubernetes.io/region=us-east-1
                    topology.kubernetes.io/zone=us-east-1a
Annotations:        cluster.k8s.io/machine: openshift-machine-api/ahardin-worker-us-east-2a-q5dzc
4
                    machineconfiguration.openshift.io/currentConfig: worker-
309c228e8b3a92e2235edd544c62fea8
```

```
                machineconfiguration.openshift.io/desiredConfig: worker-
309c228e8b3a92e2235edd544c62fea8
                machineconfiguration.openshift.io/state: Done
                volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Wed, 13 Feb 2019 11:05:57 -0500
Taints:             <none> 5
Unschedulable:      false
Conditions:                6
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason
Message
  ----             ------  -----------------                 ------------------                ------
-------
  OutOfDisk        False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -0500
KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -0500
KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -0500
KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:05:57 -0500
KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Wed, 13 Feb 2019 15:09:42 -0500   Wed, 13 Feb 2019 11:07:09 -0500
KubeletReady                 kubelet is posting ready status
Addresses:  7
  InternalIP:   10.0.140.16
  InternalDNS:  ip-10-0-140-16.us-east-2.compute.internal
  Hostname:     ip-10-0-140-16.us-east-2.compute.internal
Capacity:  8
 attachable-volumes-aws-ebs:  39
 cpu:                         2
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      8172516Ki
 pods:                        250
Allocatable:
 attachable-volumes-aws-ebs:  39
 cpu:                         1500m
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      7558116Ki
 pods:                        250
System Info:  9
 Machine ID:                 63787c9534c24fde9a0cde35c13f1f66
 System UUID:                EC22BF97-A006-4A58-6AF8-0A38DEEA122A
 Boot ID:                    f24ad37d-2594-46b4-8830-7f7555918325
 Kernel Version:             3.10.0-957.5.1.el7.x86_64
 OS Image:                   Red Hat Enterprise Linux CoreOS 410.8.20190520.0 (Ootpa)
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  cri-o://1.29.4-0.6.dev.rhaos4.3.git9ad059b.el8-rc2
 Kubelet Version:            v1.29.4
 Kube-Proxy Version:         v1.29.4
PodCIDR:                     10.128.4.0/24
ProviderID:                  aws:///us-east-2a/i-04e87b31dc6b3e171
Non-terminated Pods:         (12 in total)  10
  Namespace                  Name                                CPU Requests  CPU Limits  Memory
Requests  Memory Limits
```

```
  ---------                          ----                           ------------  ----------  ---------------  -------------
  openshift-cluster-node-tuning-operator  tuned-hdl5q                          0 (0%)        0 (0%)       0 (0%)
0 (0%)
  openshift-dns                       dns-default-l69zr                 0 (0%)        0 (0%)       0 (0%)          0
(0%)
  openshift-image-registry            node-ca-9hmcg                            0 (0%)        0 (0%)       0 (0%)
0 (0%)
  openshift-ingress                   router-default-76455c45c-c5ptv        0 (0%)        0 (0%)       0 (0%)
0 (0%)
  openshift-machine-config-operator      machine-config-daemon-cvqw9          20m (1%)       0 (0%)
50Mi (0%)        0 (0%)
  openshift-marketplace               community-operators-f67fh            0 (0%)        0 (0%)       0 (0%)
0 (0%)
  openshift-monitoring                alertmanager-main-0                  50m (3%)      50m (3%)    210Mi
(2%)       10Mi (0%)
  openshift-monitoring                node-exporter-l7q8d                  10m (0%)      20m (1%)    20Mi
(0%)       40Mi (0%)
  openshift-monitoring                prometheus-adapter-75d769c874-hvb85    0 (0%)        0 (0%)       0
(0%)          0 (0%)
  openshift-multus                    multus-kw8w5                          0 (0%)        0 (0%)       0 (0%)          0
(0%)
  openshift-sdn                       ovs-t4dsn                          100m (6%)     0 (0%)      300Mi (4%)       0
(0%)
  openshift-sdn                       sdn-g79hg                           100m (6%)     0 (0%)      200Mi (2%)
0 (0%)
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource                    Requests     Limits
  --------                    --------     ------
  cpu                         380m (25%)   270m (18%)
  memory                      880Mi (11%)  250Mi (3%)
  attachable-volumes-aws-ebs  0            0
Events:  11
  Type     Reason                   Age              From                       Message
  ----     ------                   ----             ----                       -------
  Normal   NodeHasSufficientPID     6d (x5 over 6d)    kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   NodeAllocatableEnforced  6d                 kubelet, m01.example.com  Updated Node
Allocatable limit across pods
  Normal   NodeHasSufficientMemory  6d (x6 over 6d)    kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientMemory
  Normal   NodeHasNoDiskPressure    6d (x6 over 6d)    kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasNoDiskPressure
  Normal   NodeHasSufficientDisk    6d (x6 over 6d)    kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientDisk
  Normal   NodeHasSufficientPID     6d                 kubelet, m01.example.com  Node
m01.example.com status is now: NodeHasSufficientPID
  Normal   Starting                 6d                 kubelet, m01.example.com  Starting kubelet.
#...
```

1. 节点的名称。
2. 节点的角色，可以是 **master** 或 **worker**。
3. 应用到节点的标签。

4. 应用到节点的注解。
5. 应用到节点的污点。
6. 节点条件和状态。**conditions** 小节列出了 **Ready**、**PIDPressure**、**PIDPressure**、**MemoryPressure**、**DiskPressure** 和 **OutOfDisk** 状态。本节稍后将描述这些条件。
7. 节点的 IP 地址和主机名。
8. pod 资源和可分配的资源。
9. 节点主机的相关信息。
10. 节点上的 pod。
11. 节点报告的事件。



注意

control plane 标签不会自动添加到新创建或更新的 master 节点。如果要为您的节点使用 control plane 标签，可以手动配置标签。如需更多信息，请参阅*附加资源*部分中的*了解如何更新节点上的标签*。

在显示的节点信息中，本节显示的命令输出中会出现以下节点状况：

表 6.1. 节点状况

状况	描述
Ready	如果为 **true**，节点处于健康状态，并可以接受 pod。如果为 **false**，则节点处于不健康的状态，不接受 pod。如果为 **unknown**，代表节点控制器在 **node-monitor-grace-period** 时间内（默认为 40 秒）还没有收到来自节点的心跳信号。
DiskPressure	如果为 **true**，代表磁盘容量较低。
MemoryPressure	如果为 **true**，代表节点内存较低。
PIDPressure	如果为 **true**，代表节点上的进程太多。
OutOfDisk	如果为 **true**，代表节点上的可用空间不足，无法添加新 pod。
NetworkUnavailable	如果为 **true**，代表节点的网络不会被正确配置。
NotReady	如果为**true**，代表一个底层组件（如容器运行时或网络）遇到了问题或尚未配置。
SchedulingDisabled	无法通过调度将 Pod 放置到节点上。

其他资源

- 了解如何更新节点上的标签

6.1.2. 列出集群中某一节点上的 pod

您可以列出特定节点上的所有 pod。

流程

- 列出一个或多个节点上的所有或选定 pod：

```
$ oc describe node <node1> <node2>
```

例如：

```
$ oc describe node ip-10-0-128-218.ec2.internal
```

- 列出选定节点上的所有或选定 pod：

```
$ oc describe node --selector=<node_selector>
```

```
$ oc describe node --selector=kubernetes.io/os
```

或者：

```
$ oc describe node -l=<pod_selector>
```

```
$ oc describe node -l node-role.kubernetes.io/worker
```

- 列出特定节点上的所有 pod，包括终止的 pod：

```
$ oc get pod --all-namespaces --field-selector=spec.nodeName=<nodename>
```

6.1.3. 查看节点上的内存和 CPU 用量统计

您可以显示节点的用量统计，这些统计信息为容器提供了运行时环境。这些用量统计包括 CPU、内存和存储的消耗。

先决条件

- 您必须有 **cluster-reader** 权限才能查看用量统计。
- 必须安装 Metrics 才能查看用量统计。

流程

- 查看用量统计：

```
$ oc adm top nodes
```

输出示例

```
NAME                                        CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
ip-10-0-12-143.ec2.compute.internal     1503m        100%      4533Mi          61%
ip-10-0-132-16.ec2.compute.internal     76m          5%        1391Mi          18%
ip-10-0-140-137.ec2.compute.internal    398m         26%       2473Mi          33%
ip-10-0-142-44.ec2.compute.internal     656m         43%       6119Mi          82%
ip-10-0-146-165.ec2.compute.internal    188m         12%       3367Mi          45%
ip-10-0-19-62.ec2.compute.internal      896m         59%       5754Mi          77%
ip-10-0-44-193.ec2.compute.internal     632m         42%       5349Mi          72%
```

- 查看具有标签的节点的用量统计信息：

```
$ oc adm top node --selector=''
```

您必须选择过滤所基于的选择器（标签查询）。支持 **=**、**==** 和 **!=**。

6.2. 操作节点

作为管理员，您可以执行几个任务来使集群更高效。

6.2.1. 了解如何撤离节点上的 pod

通过撤离 pod，您可以迁移给定的一个或多个节点上的所有或选定 pod。

您只能撤离由复制控制器支持的 pod。复制控制器在其他节点上创建新 pod，并从指定节点移除现有的 pod。

裸机 pod（即不由复制控制器支持的 pod）默认情况下不受影响。您可以通过指定 pod 选择器来撤离一小部分 pod。pod 选择器基于标签，因此带有指定标签的所有 pod 都将被撤离。

流程

1. 在执行 pod 驱除前，标记不可调度的节点。

   a. 将节点标记为不可调度：

   ```
   $ oc adm cordon <node1>
   ```

   输出示例

   ```
   node/<node1> cordoned
   ```

   b. 检查节点状态为 **Ready,SchedulingDisabled**:

   ```
   $ oc get node <node1>
   ```

   输出示例

   ```
   NAME      STATUS                     ROLES    AGE     VERSION
   <node1>   Ready,SchedulingDisabled   worker   1d      v1.29.4
   ```

2. 使用以下方法之一驱除 pod:

   - 在一个或多个节点上驱除所有或选定的 pod：

```
$ oc adm drain <node1> <node2> [--pod-selector=<pod_selector>]
```

- 使用 **--force** 选项强制删除裸机 pod。设为 **true** 时，即使存在不由复制控制器、副本集、作业、守护进程设置或有状态设置管理的 pod，也会继续执行删除：

```
$ oc adm drain <node1> <node2> --force=true
```

- 使用 **--grace-period** 以秒为单位设置一个期限，以便每个 pod 能够安全地终止。如果为负，则使用 pod 中指定的默认值：

```
$ oc adm drain <node1> <node2> --grace-period=-1
```

- 忽略由守护进程集管理的 pod，将 **--ignore-daemonsets** 标记设为 **true**：

```
$ oc adm drain <node1> <node2> --ignore-daemonsets=true
```

- 使用 **--timeout** 标记来设置在放弃前要等待的时长。值为 **0** 时设定无限时长：

```
$ oc adm drain <node1> <node2> --timeout=5s
```

- 即使存在使用 **emptyDir** 卷的 pod，将 **--delete-emptydir-data** 标志设为 **true**，也会删除 pod。节点排空时会删除本地数据：

```
$ oc adm drain <node1> <node2> --delete-emptydir-data=true
```

- 把 **--dry-run** 选项设为 **true**，它会列出将要迁移的对象而不实际执行撤离：

```
$ oc adm drain <node1> <node2>  --dry-run=true
```

  您可以使用 **--selector=<node_selector>** 选项来撤离选定节点上的 pod，而不指定具体的节点名称（如 **<node1> <node2>**）。

3. 完成后将节点标记为可调度。

```
$ oc adm uncordon <node1>
```

6.2.2. 了解如何更新节点上的标签

您可以更新节点上的任何标签。

节点标签不会在节点删除后保留，即使机器备份了节点也是如此。

注意

对 **MachineSet** 对象的任何更改都不会应用到计算机器集拥有的现有机器。例如，对现有 **MachineSet** 对象编辑或添加的标签不会传播到与计算机器集关联的现有机器和节点。

- 以下命令在节点上添加或更新标签：

```
$ oc label node <node> <key_1>=<value_1> ... <key_n>=<value_n>
```

例如：

```
$ oc label nodes webconsole-7f7f6 unhealthy=true
```

提示

您还可以应用以下 YAML 来应用标签：

```
kind: Node
apiVersion: v1
metadata:
  name: webconsole-7f7f6
  labels:
    unhealthy: 'true'
#...
```

- 以下命令更新命名空间中的所有 pod：

```
$ oc label pods --all <key_1>=<value_1>
```

例如：

```
$ oc label pods --all status=unhealthy
```

6.2.3. 了解如何将节点标记为不可调度或可以调度

默认情况下，具有 **Ready** 状态的健康节点被标记为可以调度，这意味着您可以在节点上放置新 pod。如果手动将节点标记为不可调度，则会阻止在该节点上调度任何新的 pod。节点上的现有 pod 不受影响。

- 以下命令将一个或多个节点标记为不可调度：

输出示例

```
$ oc adm cordon <node>
```

例如：

```
$ oc adm cordon node1.example.com
```

输出示例

```
node/node1.example.com cordoned

NAME                LABELS                                        STATUS
node1.example.com   kubernetes.io/hostname=node1.example.com
Ready,SchedulingDisabled
```

- 以下命令将当前不可调度的一个或多个节点标记为可以调度：

```
$ oc adm uncordon <node1>
```

另外，您也可以使用 **--selector=<node_selector>** 选项将选定的节点标记为可以调度或不可调度，而不指定具体的节点名称（如 **<node>**）。

6.2.4. 当节点重启而不排空应用程序 pod 时，处理单节点 OpenShift 集群中的错误

通常，在单节点 OpenShift 集群和 OpenShift Container Platform 集群中，可能会出现节点重启的情况，而无需首先排空节点。这可能会发生请求设备的应用程序 pod 失败，并显示 **UnexpectedAdmissionError** 错误。报告 **Deployment**、**ReplicaSet** 或 **DaemonSet** 错误，因为需要这些设备的应用程序 pod 在 pod 为这些设备提供服务前启动。您无法控制 pod 重启的顺序。

虽然此行为是预期的，但可能会导致 pod 保持在集群中，即使它无法成功部署。pod 继续报告 **UnexpectedAdmissionError**。现在，应用程序 pod 通常包含在 **Deployment**、**ReplicaSet** 或 **DaemonSet** 中可以缓解此问题。如果 pod 处于这个错误状态，则这不太关注，因为另一个实例应该正在运行。属于 **Deployment**、**ReplicaSet** 或 **DaemonSet** 可确保成功创建并执行后续 pod，并确保应用程序成功部署。

上游正在进行工作来确保此类 pod 被安全终止。在解决这一问题前，运行以下命令让单节点 OpenShift 集群删除失败的 pod：

```
$ oc delete pods --field-selector status.phase=Failed -n <POD_NAMESPACE>
```

注意

单节点 OpenShift 集群无法使用排空节点的选项。

其他资源

- 了解如何撤离节点上的 pod

6.2.5. 删除节点

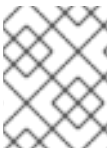
6.2.5.1. 从集群中删除节点

要从 OpenShift Container Platform 集群中删除节点，请缩减适当的 **MachineSet** 对象。

重要

当集群与云供应商集成时，您必须删除对应的机器来删除节点。不要尝试将 **oc delete node** 命令用于此任务。

当您使用 CLI 删除节点时，节点对象会从 Kubernetes 中删除，但该节点上存在的 pod 不会被删除。任何未由复制控制器支持的裸机 pod 都无法从 OpenShift Container Platform 访问。由复制控制器支持的 Pod 会重新调度到其他可用的节点。您必须删除本地清单 pod。



注意

如果您在裸机上运行集群，则无法通过编辑 **MachineSet** 对象来删除节点。计算机器集仅在集群与云供应商集成时才可用。相反，您必须在手动删除前取消调度并排空节点。

流程

1. 运行以下命令，查看集群中的计算机器：

```
$ oc get machinesets -n openshift-machine-api
```

计算机器集以 **<cluster-id>-worker-<aws-region-az>** 的形式列出。

2. 使用以下方法之一缩减计算机器：

- 运行以下命令指定要缩减的副本数：

```
$ oc scale --replicas=2 machineset <machine-set-name> -n openshift-machine-api
```

- 运行以下命令来编辑计算机器设置自定义资源：

```
$ oc edit machineset <machine-set-name> -n openshift-machine-api
```

输出示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  # ...
  name: <machine-set-name>
  namespace: openshift-machine-api
  # ...
spec:
  replicas: 2 (1)
  # ...
```

(1) 指定要缩减的副本数。

其他资源

- 手动扩展计算机器集

6.2.5.2. 从裸机集群中删除节点

当您使用 CLI 删除节点时，节点对象会从 Kubernetes 中删除，但该节点上存在的 pod 不会被删除。任何未由复制控制器支持的裸机 pod 都无法从 OpenShift Container Platform 访问。由复制控制器支持的 Pod 会重新调度到其他可用的节点。您必须删除本地清单 pod。

流程

通过完成以下步骤，从裸机上运行的 OpenShift Container Platform 集群中删除节点：

1. 将节点标记为不可调度：

```
$ oc adm cordon <node_name>
```

2. 排空节点上的所有 pod：

```
$ oc adm drain <node_name> --force=true
```

如果节点离线或者无响应，此步骤可能会失败。即使节点没有响应，它仍然在运行写入共享存储的工作负载。为了避免数据崩溃，请在进行操作前关闭物理硬件。

3. 从集群中删除节点：

```
$ oc delete node <node_name>
```

虽然节点对象现已从集群中删除，但它仍然可在重启后或 kubelet 服务重启后重新加入集群。要永久删除该节点及其所有数据，您必须弃用该节点。

4. 如果您关闭了物理硬件，请重新打开它以便节点可以重新加入集群。

6.3. 管理节点

OpenShift Container Platform 使用 KubeletConfig 自定义资源（CR）来管理节点的配置。通过创建 **KubeletConfig** 对象的实例，会创建一个受管机器配置来覆盖节点上的设置。

注意

不支持为更改配置而登录远程机器。

6.3.1. 修改节点

要对集群或机器池进行配置更改，您必须创建自定义资源定义（CRD）或 **kubeletConfig** 对象。OpenShift Container Platform 使用 Machine Config Controller 来监控是否通过 CRD 进行了更改，以将更改应用到集群。

注意

因为 **kubeletConfig** 对象中的字段直接从上游 Kubernetes 传递给 kubelet，所以对这些字段的验证直接由 kubelet 本身处理。有关这些字段的有效值，请参阅相关的 Kubernetes 文档。**kubeletConfig** 对象中的无效值可能会导致集群节点不可用。

流程

1. 为您要配置的节点类型，获取与静态 CRD (Machine Config Pool) 关联的标签。执行以下步骤之一：

   a. 检查所需机器配置池的当前标签。
   例如：

```
$  oc get machineconfigpool  --show-labels
```

   输出示例

```
NAME      CONFIG                                             UPDATED   UPDATING   DEGRADED
LABELS
master    rendered-master-e05b81f5ca4db1d249a1bf32f9ec24fd   True      False
False     operator.machineconfiguration.openshift.io/required-for-upgrade=
worker    rendered-worker-f50e78e1bc06d8e82327763145bfcf62   True      False
False
```

   b. 为所需的机器配置池添加自定义标签。
   例如：

```
$ oc label machineconfigpool worker custom-kubelet=enabled
```

2. 为您的配置更改创建一个 **kubeletconfig** 自定义资源（CR）。
例如：

custom-config CR 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-config (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: enabled (2)
  kubeletConfig: (3)
    podsPerCore: 10
    maxPods: 250
    systemReserved:
      cpu: 2000m
      memory: 1Gi
#...
```

(1) 为 CR 分配一个名称。

(2) 指定要应用配置更改的标签，这是您添加到机器配置池中的标签。

(3) 指定要更改的新值。

3. 创建 CR 对象。

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

例如：

```
$ oc create -f master-kube-config.yaml
```

大多数 Kubelet 配置选项 可由用户设置。不允许覆盖下列选项：

- CgroupDriver
- ClusterDNS
- ClusterDomain
- StaticPodPath

注意

如果单个节点包含超过 50 个镜像，pod 调度可能会在节点间进行平衡。这是因为节点上的镜像列表默认简写为 50。您可以通过编辑 **KubeletConfig** 对象来禁用镜像限制，并将 **nodeStatusMaxImages** 的值设置为 **-1**。

6.3.2. 更新引导镜像

Machine Config Operator (MCO)使用引导镜像来启动 Red Hat Enterprise Linux CoreOS (RHCOS) 节点。默认情况下，OpenShift Container Platform 不管理引导镜像。

这意味着集群中的引导镜像不会随集群一起更新。例如，如果您的集群最初使用 OpenShift Container Platform 4.12 创建，集群用来创建节点的引导镜像是相同的 4.12 版本，即使集群是更新的版本。如果以后将集群升级到 4.13 或更高版本，新节点将继续使用相同的 4.12 镜像扩展。

这个过程可能会导致以下问题：

- 启动节点的额外时间
- 证书过期问题
- 版本偏移问题

要避免这些问题，您可以将集群配置为在更新集群时更新引导镜像。通过修改 **MachineConfiguration** 对象，您可以启用此功能。目前，更新引导镜像的功能仅适用于 Google Cloud Platform (GCP) 集群，且 Cluster CAPI Operator 受管集群不支持。

重要

更新引导镜像功能只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

要查看集群中使用的当前引导镜像，请检查机器集：

使用引导镜像引用的机器集示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: ci-ln-hmy310k-72292-5f87z-worker-a
  namespace: openshift-machine-api
spec:
# ...
  template:
# ...
    spec:
# ...
      providerSpec:
# ...
        value:
          disks:
          - autoDelete: true
            boot: true
            image: projects/rhcos-cloud/global/images/rhcos-412-85-202203181601-0-gcp-x86-64 (1)
# ...
```

(1) 此引导镜像与最初安装的 OpenShift Container Platform 版本相同，在本例中为 OpenShift Container Platform 4.12，无论集群的当前版本是什么。在机器集中表示引导镜像的方式取决于平台，因为 **providerSpec** 字段的结构与平台不同。

如果您将集群配置为更新引导镜像，机器集中引用的引导镜像与集群的当前版本匹配。

先决条件

- 已使用功能门启用 **TechPreviewNoUpgrade** 功能集。如需更多信息，请参阅附加资源部分中的"使用功能门启用功能"。

流程

1. 运行以下命令，编辑名为 **cluster** 的 **MachineConfiguration** 对象，以启用引导镜像的更新：

```
$ oc edit MachineConfiguration cluster
```

   a. 可选：为所有机器集配置引导镜像更新功能：

```
apiVersion: operator.openshift.io/v1
kind: MachineConfiguration
metadata:
  name: cluster
  namespace: openshift-machine-config-operator
spec:
# ...
  managedBootImages: 1
    machineManagers:
    - resource: machinesets
      apiGroup: machine.openshift.io
      selection:
        mode: All 2
```

   1 激活引导镜像更新功能。

   2 指定集群中的所有机器集都会被更新。

   b. 可选：为特定机器集配置引导镜像更新功能：

```
apiVersion: operator.openshift.io/v1
kind: MachineConfiguration
metadata:
  name: cluster
  namespace: openshift-machine-config-operator
spec:
# ...
  managedBootImages: 1
    machineManagers:
    - resource: machinesets
      apiGroup: machine.openshift.io
      selection:
        mode: Partial
          partial:
            machineResourceSelector:
              matchLabels:
                update-boot-image: "true" 2
```

1 激活引导镜像更新功能。

2 指定具有此标签的任何机器集都会被更新。

提示

如果机器集中没有适当的标签，请运行以下命令来添加键/值对：

```
$ oc label machineset.machine ci-ln-hmy310k-72292-5f87z-worker-a update-boot-image=true -n openshift-machine-api
```

验证

1. 运行以下命令来获取引导镜像版本：

```
$ oc get machinesets <machineset_name> -n openshift-machine-api -o yaml
```

使用引导镜像引用的机器集示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  labels:
    machine.openshift.io/cluster-api-cluster: ci-ln-77hmkpt-72292-d4pxp
    update-boot-image: "true"
  name: ci-ln-77hmkpt-72292-d4pxp-worker-a
  namespace: openshift-machine-api
spec:
# ...
  template:
# ...
    spec:
# ...
      providerSpec:
# ...
        value:
          disks:
          - autoDelete: true
            boot: true
            image: projects/rhcos-cloud/global/images/rhcos-416-92-202402201450-0-gcp-x86-64 1
# ...
```

1 此引导镜像与当前的 OpenShift Container Platform 版本相同。

其他资源

- 使用功能门启用功能

6.3.2.1. 禁用更新的引导镜像

要禁用更新的引导镜像功能，请编辑 **MachineConfiguration** 对象以删除 **managedBootImages** 小节。

如果您在使用新的引导镜像版本创建了某些节点后禁用此功能，则任何现有节点会保留其当前引导镜像。关闭这个功能不会将节点或机器集回滚到原始安装的引导镜像。机器集会保留启用该功能时出现的引导镜像版本，并在将来升级到新的 OpenShift Container Platform 版本时不会再次更新。

流程

1. 通过编辑 **MachineConfiguration** 对象禁用更新的引导镜像：

```
$ oc edit MachineConfiguration cluster
```

2. 删除 **managedBootImages** 小节：

```
apiVersion: operator.openshift.io/v1
kind: MachineConfiguration
metadata:
  name: cluster
  namespace: openshift-machine-config-operator
spec:
# ...
  managedBootImages: 1
    machineManagers:
    - resource: machinesets
      apiGroup: machine.openshift.io
      selection:
        mode: All
```

1 删除整个小节以禁用更新的引导镜像。

6.3.3. 将 control plane 节点配置为可以调度

您可以将 control plane 节点配置为可以调度，这意味着允许在 master 节点上放置新的 pod。默认情况下，control plane 节点不可调度。

您可以将 master 设置为可调度，但必须保留 worker 节点。

注意

您可以在裸机集群中部署没有 worker 节点的 OpenShift Container Platform。在这种情况下，control plane 节点会被标记为可以调度。

您可以通过配置 **mastersSchedulable** 字段来允许或禁止调度 control plane 节点。

重要

当您将 control plane 节点从默认的不可调度配置为可以调度时，需要额外的订阅。这是因为 control plane 节点随后变为 worker 节点。

流程

1. 编辑 **schedulers.config.openshift.io** 资源。

```
$ oc edit schedulers.config.openshift.io cluster
```

2. 配置 **mastersSchedulable** 字段。

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  creationTimestamp: "2019-09-10T03:04:05Z"
  generation: 1
  name: cluster
  resourceVersion: "433"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: a636d30a-d377-11e9-88d4-0a60097bee62
spec:
  mastersSchedulable: false (1)
status: {}
#...
```

(1) 设置为 **true** 以允许调度 control plane 节点，或设置为 **false** 以禁止调度 control plane 节点。

3. 保存文件以使改变生效。

6.3.4. 设置 SELinux 布尔值

OpenShift Container Platform 允许您在 Red Hat Enterprise Linux CoreOS (RHCOS) 节点上启用和禁用 SELinux 布尔值。以下流程解释了如何使用 Machine Config Operator (MCO) 修改节点上的 SELinux 布尔值。此流程使用 **container_manage_cgroup** 作为示例布尔值。您可以将这个值修改为您需要的任何布尔值。

先决条件

- 已安装 OpenShift CLI（oc）。

流程

1. 使用 **MachineConfig** 对象创建新 YAML 文件，如下例所示：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 99-worker-setsebool
spec:
  config:
    ignition:
      version: 3.2.0
    systemd:
      units:
      - contents: |
          [Unit]
          Description=Set SELinux booleans
          Before=kubelet.service
```

```
      [Service]
      Type=oneshot
      ExecStart=/sbin/setsebool container_manage_cgroup=on
      RemainAfterExit=true

      [Install]
      WantedBy=multi-user.target graphical.target
    enabled: true
    name: setsebool.service
#...
```

2. 运行以下命令来创建新的 **MachineConfig** 对象：

```
$ oc create -f 99-worker-setsebool.yaml
```

注意

对 **MachineConfig** 对象应用任何更改将导致所有受影响的节点在应用更改后安全重启。

6.3.5. 为节点添加内核参数

在一些特殊情况下，您可能需要为集群中的一组节点添加内核参数。进行此操作时应小心谨慎，而且您必须先清楚了解所设参数的影响。

警告

不当使用内核参数会导致系统变得无法引导。

您可以设置的内核参数示例包括：

- **nosmt**：在内核中禁用对称多线程 (SMT)。多线程允许每个 CPU 有多个逻辑线程。您可以在多租户环境中考虑使用 **nosmt**，以减少潜在的跨线程攻击风险。禁用 SMT 在本质上相当于选择安全性而非性能。
- **systemd.unified_cgroup_hierarchy**：启用 Linux 控制组版本 2 (cgroup v2)。cgroup v2 是内核控制组的下一个版本，它包括了多个改进。

重要

cgroup v1 是一个已弃用的功能。弃用的功能仍然包含在 OpenShift Container Platform 中，并将继续被支持。但是，这个功能会在以后的发行版本中被删除，且不建议在新的部署中使用。

有关 OpenShift Container Platform 中已弃用或删除的主要功能的最新列表，请参阅 OpenShift Container Platform 发行注记中*已弃用和删除的功能*部分。

- **Enforcing=0**：将 Security Enhanced Linux（SELinux）配置为以 permissive 模式运行。在 permissive 模式中，系统会象 enforcing 模式一样加载安全策略，包括标记对象并在日志中记录

访问拒绝条目，但它并不会拒绝任何操作。虽然不建议在生产环境系统中使用 permissive 模式，但 permissive 模式会有助于调试。

警告

不支持在生产环境中禁用 RHCOS 上的 SELinux。在节点上禁用 SELinux 后，必须在生产集群中重新设置前重新置备它。

如需内核参数的列表和描述，请参阅 Kernel.org 内核参数。

在以下流程中，您要创建一个用于标识以下内容的 **MachineConfig** 对象：

- 您要添加内核参数的一组机器。本例中为具有 worker 角色的机器。
- 附加到现有内核参数末尾的内核参数。
- 指示机器配置列表中应用更改的位置的标签。

先决条件

- 具有正常运行的 OpenShift Container Platform 集群的管理特权。

流程

1. 列出 OpenShift Container Platform 集群的现有 **MachineConfig** 对象，以确定如何标记您的机器配置：

```
$ oc get MachineConfig
```

输出示例

```
NAME                                               GENERATEDBYCONTROLLER
IGNITIONVERSION   AGE
00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
01-master-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-master-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-kubelet                                  52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-ssh                                                                 3.2.0             40m
99-worker-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
```

```
99-worker-ssh                                      3.2.0             40m
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
```

2. 创建一个用于标识内核参数的 **MachineConfig** 对象文件（例如 **05-worker-kernelarg-selinuxpermissive.yaml**）

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker 1
  name: 05-worker-kernelarg-selinuxpermissive 2
spec:
  kernelArguments:
    - enforcing=0 3
```

1. 仅将新内核参数应用到 worker 节点。
2. 用于标识它插入到机器配置中的什么位置（05）以及发挥什么作用（添加一个内核参数来配置 SELinux permissive 模式）。
3. 将确切的内核参数标识为 **enforcing=0**。

3. 创建新机器配置：

```
$ oc create -f 05-worker-kernelarg-selinuxpermissive.yaml
```

4. 检查机器配置以查看是否添加了新配置：

```
$ oc get MachineConfig
```

输出示例

```
NAME                                               GENERATEDBYCONTROLLER
IGNITIONVERSION   AGE
00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
01-master-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-master-kubelet                                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-kubelet                                        52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
05-worker-kernelarg-selinuxpermissive                                                3.2.0             105s
99-master-generated-registries                           52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
```

```
99-master-ssh                                                    3.2.0             40m
99-worker-generated-registries                 52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-worker-ssh                                                    3.2.0             40m
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
```

5. 检查节点：

```
$ oc get nodes
```

输出示例

```
NAME                         STATUS                     ROLES    AGE   VERSION
ip-10-0-136-161.ec2.internal   Ready                      worker   28m   v1.29.4
ip-10-0-136-243.ec2.internal   Ready                      master   34m   v1.29.4
ip-10-0-141-105.ec2.internal   Ready,SchedulingDisabled   worker   28m   v1.29.4
ip-10-0-142-249.ec2.internal   Ready                      master   34m   v1.29.4
ip-10-0-153-11.ec2.internal    Ready                      worker   28m   v1.29.4
ip-10-0-153-150.ec2.internal   Ready                      master   34m   v1.29.4
```

您可以发现，在应用更改时每个 worker 节点上的调度都会被禁用。

6. 前往其中一个 worker 节点并列出内核命令行参数（主机上的 **/proc/cmdline** 中），以检查内核参数确实已发挥作用：

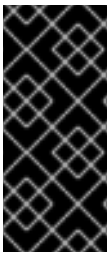
```
$ oc debug node/ip-10-0-141-105.ec2.internal
```

输出示例

```
Starting pod/ip-10-0-141-105ec2internal-debug ...
To use host binaries, run `chroot /host`

sh-4.2# cat /host/proc/cmdline
BOOT_IMAGE=/ostree/rhcos-... console=tty0 console=ttyS0,115200n8
rootflags=defaults,prjquota rw root=UUID=fd0... ostree=/ostree/boot.0/rhcos/16...
coreos.oem.id=qemu coreos.oem.id=ec2 ignition.platform.id=ec2 enforcing=0

sh-4.2# exit
```

您应看到 **enforcing=0** 参数已添加至其他内核参数。

6.3.6. 在节点上启用交换内存使用



重要

在节点上启用交换内存只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

您可以根据节点为 OpenShift Container Platform 工作负载启用交换内存使用。

警告

启用交换内存可能会对工作负载性能和资源不足处理造成负面影响。不要在 control plane 节点上启用交换内存。

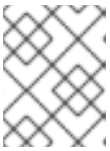
要启用交换内存，请创建一个 **kubeletconfig** 自定义资源(CR)来设置 **swapbehavior** 参数。您可以设置有限或无限的交换内存：

- 有限：使用 **LimitedSwap** 值来限制可以使用的交换内存工作负载量。任何不是由 OpenShift Container Platform 管理的节点上的工作负载都可以使用交换内存。**LimitedSwap** 行为取决于节点是否使用 Linux 控制组 版本 1(cgroups v1) 或 版本 2(cgroup v2) 运行：
  - cgroup v1：OpenShift Container Platform 工作负载可以使用内存和交换的任意组合（如果设置）到 pod 的内存限值。
  - cgroup v2：OpenShift Container Platform 工作负载无法使用交换内存。
- 无限：使用 **UnlimitedSwap** 值来允许工作负载在请求时尽可能多地使用 swap 内存，最多使用系统限制。

由于 kubelet 在没有此配置的情况下不会启动交换内存，因此您必须在 OpenShift Container Platform 中启用交换内存前在节点上启用交换内存。如果节点上没有交换内存，则在 OpenShift Container Platform 中启用交换内存不会起作用。

先决条件

- 您有一个正在运行的 OpenShift Container Platform 集群，它使用版本 4.10 或更高版本。
- 以具有管理特权的用户身份登录集群。
- 您已在集群中启用了 **TechPreviewNoUpgrade** 功能集（请参阅 *Nodes → Working with cluster → Enabling features using feature gates*）。



注意

启用 **TechPreviewNoUpgrade** 功能集将无法撤消，并防止次版本更新。不建议在生产环境集群中使用这些功能集。

- 如果节点上启用了 cgroup v2，则必须通过设置 **swapaccount=1** 内核参数来启用节点上的交换核算。

流程

1. 对要允许交换内存的机器配置池应用自定义标签。

```
$ oc label machineconfigpool worker kubelet-swap=enabled
```

2. 创建自定义资源(CR)来启用和配置 swap 设置。

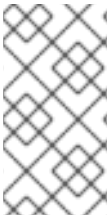
```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: swap-config
spec:
  machineConfigPoolSelector:
    matchLabels:
      kubelet-swap: enabled
  kubeletConfig:
    failSwapOn: false (1)
    memorySwap:
      swapBehavior: LimitedSwap (2)
#...
```

1. 设置为 **false**，以在关联的节点上启用交换内存使用。设置为 **true** 可禁用交换内存使用。
2. 指定交换内存行为。如果未指定，则默认值为 **LimitedSwap**。

3. 在机器上启用交换内存。

6.3.7. 手动将 control plane 节点从一个 RHOSP 主机迁移到另一个 RHOSP 主机

如果没有在集群中启用 control plane 机器集，您可以运行将 control plane 节点从一个 Red Hat OpenStack Platform (RHOSP)节点移动到另一个 Red Hat OpenStack Platform (RHOSP)节点的脚本。



注意

在用户置备的基础架构上运行的集群上不启用 control plane 机器集。

有关 control plane 机器集的详情，请参考"使用 control plane 机器集管理 control plane 机器"。

先决条件

- 环境变量 **OS_CLOUD** 是对在 **clouds.yaml** 文件中具有管理凭证的 **clouds** 条目的引用。
- 环境变量 **KUBECONFIG** 是指包含管理 OpenShift Container Platform 凭证的配置。

流程

- 在命令行中运行以下命令：

```
#!/usr/bin/env bash

set -Eeuo pipefail

if [ $# -lt 1 ]; then
 echo "Usage: '$0 node_name'"
 exit 64
fi

# Check for admin OpenStack credentials
openstack server list --all-projects >/dev/null || { >&2 echo "The script needs OpenStack admin
```

```
credentials. Exiting"; exit 77; }

# Check for admin OpenShift credentials
oc adm top node >/dev/null || { >&2 echo "The script needs OpenShift admin credentials. Exiting"; exit
77; }

set -x

declare -r node_name="$1"
declare server_id
server_id="$(openstack server list --all-projects -f value -c ID -c Name | grep "$node_name" | cut -d' '
-f1)"
readonly server_id

# Drain the node
oc adm cordon "$node_name"
oc adm drain "$node_name" --delete-emptydir-data --ignore-daemonsets --force

# Power off the server
oc debug "node/${node_name}" -- chroot /host shutdown -h 1

# Verify the server is shut off
until openstack server show "$server_id" -f value -c status | grep -q 'SHUTOFF'; do sleep 5; done

# Migrate the node
openstack server migrate --wait "$server_id"

# Resize the VM
openstack server resize confirm "$server_id"

# Wait for the resize confirm to finish
until openstack server show "$server_id" -f value -c status | grep -q 'SHUTOFF'; do sleep 5; done

# Restart the VM
openstack server start "$server_id"

# Wait for the node to show up as Ready:
until oc get node "$node_name" | grep -q "^${node_name}[[:space:]]\+Ready"; do sleep 5; done

# Uncordon the node
oc adm uncordon "$node_name"

# Wait for cluster operators to stabilize
until oc get co -o go-template='statuses: {{ range .items }}{{ range .status.conditions }}{{ if eq .type
"Degraded" }}{{ if ne .status "False" }}DEGRADED{{ end }}{{ else if eq .type "Progressing"}}{{ if ne
.status "False" }}PROGRESSING{{ end }}{{ else if eq .type "Available"}}{{ if ne .status "True"
}}NOTAVAILABLE{{ end }}{{ end }}{{ end }}{{ end }}' | grep -qv '\
(DEGRADED\|PROGRESSING\|NOTAVAILABLE\)'; do sleep 5; done
```

如果脚本完成，control plane 机器将迁移到一个新的 RHOSP 节点。

其他资源

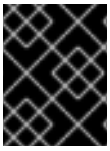
- 使用 control plane 机器集管理 control plane 机器

6.4. 管理每个节点的 POD 数量上限

在 OpenShift Container Platform 中，您可以根据节点上的处理器内核数和/或硬限制，来配置可在节点上运行的 pod 数量。如果您同时使用这两个选项，则取两者中较小的限制来限制节点上的 pod 数。

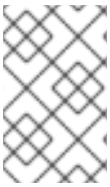
当两个参数都被设置时，其中较小的值限制了节点上的 pod 数量。超过这些值可导致：

- CPU 使用率增加。
- 减慢 pod 调度的速度。
- 根据节点中的内存数量，可能出现内存耗尽的问题。
- 耗尽 IP 地址池。
- 资源过量使用，导致用户应用程序性能变差。



重要

在 Kubernetes 中，包含单个容器的 pod 实际使用两个容器。第二个容器用来在实际容器启动前设置联网。因此，运行 10 个 pod 的系统实际上会运行 20 个容器。



注意

云供应商的磁盘 IOPS 节流可能会对 CRI-O 和 kubelet 产生影响。当节点上运行大量 I/O 高负载的 pod 时，可能会出现超载的问题。建议您监控节点上的磁盘 I/O，并使用有足够吞吐量的卷。

podsPerCore 参数根据节点上的处理器内核数来设置节点可运行的 pod 数量。例如：在一个有 4 个处理器内核的节点上将 **podsPerCore** 设为 **10**，则该节点上允许的最大 pod 数量为 **40**。

```
kubeletConfig:
  podsPerCore: 10
```

将 **podsPerCore** 设置为 **0** 可禁用这个限制。默认值为 **0**。**podsPerCore** 参数的值不能超过 **maxPods** 参数的值。

maxPods 参数将节点可以运行的 pod 数量设置为固定值，而不考虑节点的属性。

```
kubeletConfig:
  maxPods: 250
```

6.4.1. 配置每个节点的最大 pod 数量

有两个参数控制可调度到节点的 pod 数量上限，分别为 **podsPerCore** 和 **maxPods**。如果您同时使用这两个选项，则取两者中较小的限制来限制节点上的 pod 数。

例如，如果将一个有 4 个处理器内核的节点上的 **podsPerCore** 设置为 **10**，则该节点上允许的 pod 数量上限为 40。

先决条件

1. 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** CRD 关联的标签：

```
$ oc edit machineconfigpool <name>
```

例如：

```
$ oc edit machineconfigpool worker
```

输出示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: "" (1)
  name: worker
#...
```

(1) 标签会出现在 Labels 下。

提示

如果标签不存在，请添加键/值对，例如：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

max-pods CR 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-max-pods (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (2)
  kubeletConfig:
    podsPerCore: 10 (3)
    maxPods: 250 (4)
#...
```

(1) 为 CR 分配一个名称。

(2) 指定机器配置池中的标签。

(3) 根据节点的处理器内核数限制节点上可运行的 pod 数量。

4 将节点上可运行的 pod 数量指定为一个固定值，而不考虑节点的属性。

注意

将 **podsPerCore** 设置为 **0** 可禁用这个限制。

在上例中，**podsPerCore** 的默认值为 **10**，**maxPods** 的默认值则为 **250**。这意味着，除非节点有 25 个以上的内核，否则 **podsPerCore** 就是默认的限制因素。

2. 运行以下命令来创建 CR：

```
$ oc create -f <file_name>.yaml
```

验证

1. 列出 **MachineConfigPool** CRD 以查看是否应用了更改。如果 Machine Config Controller 抓取到更改，则 **UPDATING** 列会报告 **True**：

```
$ oc get machineconfigpools
```

输出示例

```
NAME    CONFIG                    UPDATED   UPDATING   DEGRADED
master  master-9cc2c72f205e103bb534  False     False      False
worker  worker-8cecd1236b33ee3f8a5e  False     True       False
```

更改完成后，**UPDATED** 列会报告 **True**。

```
$ oc get machineconfigpools
```

输出示例

```
NAME    CONFIG                    UPDATED   UPDATING   DEGRADED
master  master-9cc2c72f205e103bb534  False     True       False
worker  worker-8cecd1236b33ee3f8a5e  True      False      False
```

6.5. 使用 NODE TUNING OPERATOR

了解 Node Tuning Operator，以及如何使用它通过编排 tuned 守护进程以管理节点级别的性能优化。

用途
Node Tuning Operator 可以帮助您通过编排 TuneD 守护进程来管理节点级别的性能优化，并使用 Performance Profile 控制器获得低延迟性能。大多数高性能应用程序都需要一定程度的内核级性能优化。Node Tuning Operator 为用户提供了一个统一的、节点一级的 sysctl 管理接口，并可以根据具体用户的需要灵活地添加自定义性能优化设置。

Operator 将为 OpenShift Container Platform 容器化 TuneD 守护进程作为一个 Kubernetes 守护进程集进行管理。它保证了自定义性能优化设置以可被守护进程支持的格式传递到在集群中运行的所有容器化的 TuneD 守护进程中。相应的守护进程会在集群的所有节点上运行，每个节点上运行一个。

在发生触发配置集更改的事件时，或通过接收和处理终止信号安全终止容器化 TuneD 守护进程时，容器化 TuneD 守护进程所应用的节点级设置将被回滚。

Node Tuning Operator 使用 Performance Profile 控制器来实现自动性能优化，从而实现 OpenShift Container Platform 应用程序的低延迟性能。

集群管理员配置了性能配置集以定义节点级别的设置，例如：

- 将内核更新至 kernel-rt。
- 为内务选择 CPU。
- 为运行工作负载选择 CPU。

注意

目前，cgroup v2 不支持禁用 CPU 负载均衡。因此，如果您启用了 cgroup v2，则可能无法从性能配置集中获取所需的行为。如果您使用 executeace 配置集，则不建议启用 cgroup v2。

在版本 4.1 及更高版本中，OpenShift Container Platform 标准安装中包含了 Node Tuning Operator。

注意

在早期版本的 OpenShift Container Platform 中，Performance Addon Operator 用来实现自动性能优化，以便为 OpenShift 应用程序实现低延迟性能。在 OpenShift Container Platform 4.11 及更新的版本中，这个功能是 Node Tuning Operator 的一部分。

6.5.1. 访问 Node Tuning Operator 示例规格

使用此流程来访问 Node Tuning Operator 的示例规格。

流程

- 运行以下命令以访问 Node Tuning Operator 示例规格：

```
oc get tuned.tuned.openshift.io/default -o yaml -n openshift-cluster-node-tuning-operator
```

默认 CR 旨在为 OpenShift Container Platform 平台提供标准的节点级性能优化，它只能被修改来设置 Operator Management 状态。Operator 将覆盖对默认 CR 的任何其他自定义更改。若进行自定义性能优化，请创建自己的 Tuned CR。新创建的 CR 将与默认的 CR 合并，并基于节点或 pod 标识和配置文件优先级对节点应用自定义调整。

警告

虽然在某些情况下，对 pod 标识的支持可以作为自动交付所需调整的一个便捷方式，但我们不鼓励使用这种方法，特别是在大型集群中。默认 Tuned CR 并不带有 pod 标识匹配。如果创建了带有 pod 标识匹配的自定义配置集，则该功能将在此时启用。在以后的 Node Tuning Operator 版本中将弃用 pod 标识功能。

6.5.2. 自定义调整规格

Operator 的自定义资源 (CR) 包含两个主要部分。第一部分是 **profile:**，这是 TuneD 配置集及其名称的列表。第二部分是 **recommend:**，用来定义配置集选择逻辑。

多个自定义调优规格可以共存，作为 Operator 命名空间中的多个 CR。Operator 会检测到是否存在新 CR 或删除了旧 CR。所有现有的自定义性能优化设置都会合并，同时更新容器化 TuneD 守护进程的适当对象。

管理状态

通过调整默认的 Tuned CR 来设置 Operator Management 状态。默认情况下，Operator 处于 Managed 状态，默认的 Tuned CR 中没有 **spec.managementState** 字段。Operator Management 状态的有效值如下：

- Managed: Operator 会在配置资源更新时更新其操作对象
- Unmanaged: Operator 将忽略配置资源的更改
- Removed: Operator 将移除 Operator 置备的操作对象和资源

配置集数据

profile: 部分列出了 TuneD 配置集及其名称。

```
profile:
- name: tuned_profile_1
  data: |
    # TuneD profile specification
    [main]
    summary=Description of tuned_profile_1 profile

    [sysctl]
    net.ipv4.ip_forward=1
    # ... other sysctl's or other TuneD daemon plugins supported by the containerized TuneD

# ...

- name: tuned_profile_n
  data: |
    # TuneD profile specification
    [main]
    summary=Description of tuned_profile_n profile

    # tuned_profile_n profile settings
```

建议的配置集

profile: 选择逻辑通过 CR 的 **recommend:** 部分来定义。**recommend:** 部分是根据选择标准推荐配置集的项目列表。

```
recommend:
<recommend-item-1>
# ...
<recommend-item-n>
```

列表中的独立项：

```
- machineConfigLabels: 1
    <mcLabels> 2
  match: 3
    <match> 4
  priority: <priority> 5
  profile: <tuned_profile_name> 6
  operand: 7
    debug: <bool> 8
    tunedConfig:
      reapply_sysctl: <bool> 9
```

1. 可选。
2. **MachineConfig** 标签的键/值字典。键必须是唯一的。
3. 如果省略，则会假设配置集匹配，除非设置了优先级更高的配置集，或设置了 **machineConfigLabels**。
4. 可选列表。
5. 配置集排序优先级。较低数字表示优先级更高（**0** 是最高优先级）。
6. 在匹配项中应用的 TuneD 配置集。例如 **tuned_profile_1**。
7. 可选操作对象配置。
8. 为 TuneD 守护进程打开或关闭调试。**true** 为打开，**false** 为关闭。默认值为 **false**。
9. 为 TuneD 守护进程打开或关闭 **reapply_sysctl** 功能。选择 **true** 代表开启，**false** 代表关闭。

<match> 是一个递归定义的可选数组，如下所示：

```
- label: <label_name> 1
  value: <label_value> 2
  type: <label_type> 3
    <match> 4
```

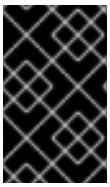
1. 节点或 pod 标签名称。
2. 可选的节点或 pod 标签值。如果省略，**<label_name>** 足以匹配。
3. 可选的对象类型（**node** 或 **pod**）。如果省略，会使用 **node**。
4. 可选的 **<match>** 列表。

如果不省略 **<match>**，则所有嵌套的 **<match>** 部分也必须评估为 **true**。否则会假定 **false**，并且不会应用或建议具有对应 **<match>** 部分的配置集。因此，嵌套（子级 **<match>** 部分）会以逻辑 AND 运算来运作。反之，如果匹配 **<match>** 列表中任何一项，整个 **<match>** 列表评估为 **true**。因此，该列表以逻辑 OR 运算来运作。

如果定义 了 **machineConfigLabels**，基于机器配置池的匹配会对给定的 **recommend:** 列表项打

开。**<mcLabels>** 指定机器配置标签。机器配置会自动创建，以在配置集 **<tuned_profile_name>** 中应用主机设置，如内核引导参数。这包括使用与 **<mcLabels>** 匹配的机器配置选择器查找所有机器配置池，并在分配了找到的机器配置池的所有节点上设置配置集 **<tuned_profile_name>**。要针对同时具有 master 和 worker 角色的节点，您必须使用 master 角色。

列表项 **match** 和 **machineConfigLabels** 由逻辑 OR 操作符连接。 **match** 项首先以短电路方式评估。因此，如果它被评估为 **true**，则不考虑 **MachineConfigLabels** 项。



重要

当使用基于机器配置池的匹配时，建议将具有相同硬件配置的节点分组到同一机器配置池中。不遵循这个原则可能会导致在共享同一机器配置池的两个或者多个节点中 TuneD 操作对象导致内核参数冲突。

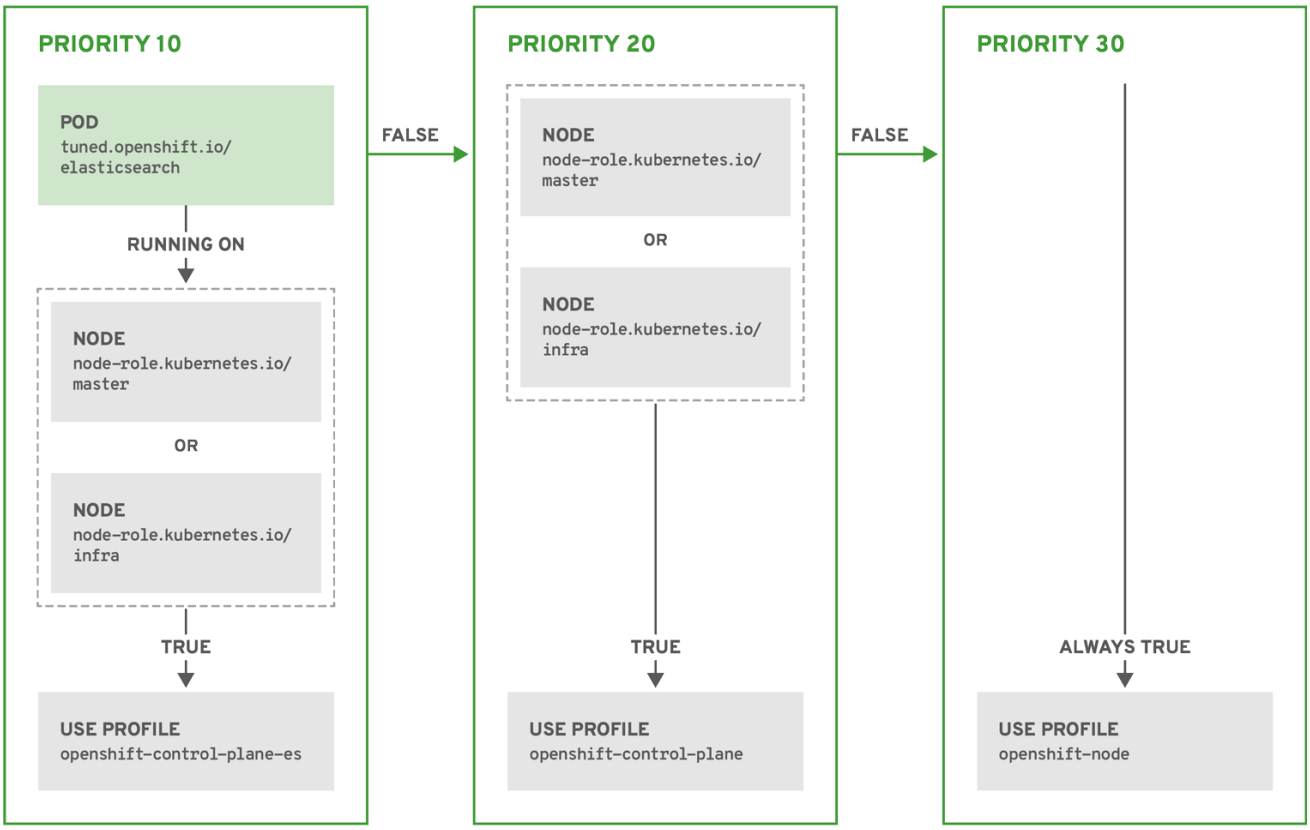
示例：基于节点或 pod 标签的匹配

```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra
  priority: 20
  profile: openshift-control-plane
- priority: 30
  profile: openshift-node
```

根据配置集优先级，以上 CR 针对容器化 TuneD 守护进程转换为 **recommend.conf** 文件。优先级最高 (**10**) 的配置集是 **openshift-control-plane-es**，因此会首先考虑它。在给定节点上运行的容器化 TuneD 守护进程会查看同一节点上是否在运行设有 **tuned.openshift.io/elasticsearch** 标签的 pod。如果没有，则整个 **<match>** 部分评估为 **false**。如果存在具有该标签的 pod，为了让 **<match>** 部分评估为 **true**，节点标签也需要是 **node-role.kubernetes.io/master** 或 **node-role.kubernetes.io/infra**。

如果这些标签对优先级为 **10** 的配置集而言匹配，则应用 **openshift-control-plane-es** 配置集，并且不考虑其他配置集。如果节点/pod 标签组合不匹配，则考虑优先级第二高的配置集 (**openshift-control-plane**)。如果容器化 TuneD Pod 在具有标签 **node-role.kubernetes.io/master** 或 **node-role.kubernetes.io/infra** 的节点上运行，则应用此配置集。

最后，配置集 **openshift-node** 的优先级最低 (**30**)。它没有 **<match>** 部分，因此始终匹配。如果给定节点上不匹配任何优先级更高的配置集，它会作为一个适用于所有节点的配置集来设置 **openshift-node** 配置集。



示例：基于机器配置池的匹配

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: openshift-node-custom
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=Custom OpenShift node profile with an additional kernel parameter
      include=openshift-node
      [bootloader]
      cmdline_openshift_node_custom=+skew_tick=1
    name: openshift-node-custom

  recommend:
  - machineConfigLabels:
      machineconfiguration.openshift.io/role: "worker-custom"
    priority: 20
    profile: openshift-node-custom
```

为尽量减少节点的重新引导情况，为目标节点添加机器配置池将匹配的节点选择器标签，然后创建上述 Tuned CR，最后创建自定义机器配置池。

特定于云供应商的 TuneD 配置集

使用此功能，所有针对于 OpenShift Container Platform 集群上的云供应商都可以方便地分配 TuneD 配置集。这可实现，而无需添加额外的节点标签或将节点分组到机器配置池中。

这个功能会利用 **spec.providerID** 节点对象值（格式为 **<cloud-provider>://<cloud-provider-specific-id>**），并在 NTO operand 容器中写带有 **<cloud-provider>** 值的文件 **/var/lib/ocp-tuned/provider**。然后，TuneD 会使用这个文件的内容来加载 **provider-<cloud-provider>** 配置集（如果这个配置集存在）。

openshift 配置集（**openshift-control-plane** 和 **openshift-node** 配置集都从其中继承设置）现在被更新来使用这个功能（通过使用条件配置集加载）。NTO 或 TuneD 目前不包含任何特定于云供应商的配置集。但是，您可以创建一个自定义配置集 **provider-<cloud-provider>**，它将适用于所有针对于所有云供应商的集群节点。

GCE 云供应商配置集示例

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: provider-gce
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=GCE Cloud provider-specific profile
      # Your tuning for GCE Cloud provider goes here.
    name: provider-gce
```

注意

由于配置集的继承，**provider-<cloud-provider>** 配置集中指定的任何设置都会被 **openshift** 配置集及其子配置集覆盖。

6.5.3. 在集群中设置默认配置集

以下是在集群中设置的默认配置集。

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=Optimize systems running OpenShift (provider specific parent profile)
      include=-provider-${f:exec:cat:/var/lib/ocp-tuned/provider},openshift
    name: openshift
  recommend:
  - profile: openshift-control-plane
    priority: 30
    match:
    - label: node-role.kubernetes.io/master
```

```
    - label: node-role.kubernetes.io/infra
  - profile: openshift-node
    priority: 40
```

从 OpenShift Container Platform 4.9 开始，所有 OpenShift TuneD 配置集都随 TuneD 软件包一起提供。您可以使用 **oc exec** 命令查看这些配置集的内容：

```
$ oc exec $tuned_pod -n openshift-cluster-node-tuning-operator -- find /usr/lib/tuned/openshift{,-control-plane,-node} -name tuned.conf -exec grep -H ^ {} \;
```

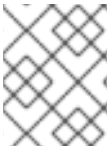
6.5.4. 支持的 TuneD 守护进程插件

在使用 Tuned CR 的 **profile:** 部分中定义的自定义配置集时，以下 TuneD 插件都受到支持，但 **[main]** 部分除外：

- audio
- cpu
- disk
- eeepc_she
- modules
- mounts
- net
- scheduler
- scsi_host
- selinux
- sysctl
- sysfs
- usb
- video
- vm
- bootloader

其中一些插件提供了不受支持的动态性能优化功能。目前不支持以下 TuneD 插件：

- script
- systemd



注意

TuneD bootloader 插件只支持 Red Hat Enterprise Linux CoreOS (RHCOS) worker 节点。

其他资源

- 可用的 TuneD 插件
- TuneD 入门

6.6. 修复、隔离和维护节点

当发生节点级别的故障时，如内核挂起或网络接口控制器 (NIC) 失败，集群所需的工作不会减少，受影响节点的工作负载需要重启一些位置。影响这些工作负载的风险、损坏或两者的故障。在开始恢复工作负载（称为 **remediation（补救）**），以及恢复节点前，需要隔离节点（称为 **fencing** ）。

如需有关补救、隔离和维护节点的更多信息，请参阅 Red Hat OpenShift 文档中的工作负载可用性。

6.7. 了解节点重新引导

若要在重新引导节点时不造成平台上应用程序中断运行，务必要先撤离（evacuate）相关的 pod。对于由路由层提供高可用性的 pod，不需要执行其他操作。对于需要存储的其他 pod（通常是数据库），务必要确保它们能够在一个 pod 临时下线时仍然保持运作。尽管为有状态 pod 实现可持续运行的方法会视不同应用程序而异，但在所有情形中，都要将调度程序配置为使用节点反关联性，从而确保 pod 在可用节点之间合理分布。

另一个挑战是如何处理运行关键基础架构的节点，比如路由器或 registry。相同的节点撤离过程同样适用于这类节点，但必须要了解某些边缘情况。

6.7.1. 关于重新引导运行关键基础架构的节点

在重启托管关键 OpenShift Container Platform 基础架构组件（如路由器 Pod、registry pod 和监控 pod）的节点时，请确保至少有三个节点可用于运行这些组件。

以下场景演示了，在只有两个节点可用时，在 OpenShift Container Platform 上运行的应用程序可能会发生服务中断：

- 节点 A 标记为不可调度，所有 pod 都被撤离。
- 该节点上运行的 registry pod 现在重新部署到节点 B 上。Node B 现在同时运行两个 registry pod。
- 节点 B 现在标记为不可调度，并且被撤离。
- 在节点上公开两个 pod 端点的服务在短时间内丢失所有端点，直到它们被重新部署到节点 A。

当将三个节点用于基础架构组件时，此过程不会造成服务中断。但是，由于 pod 调度，被撤离并返回到轮转的最后一个节点没有 registry pod。其他节点中的一个会有两个 registry pod。要将第三个 registry pod 调度到最后一个节点上，请使用 pod 反关联性以防止将两个 registry pod 放置在同一节点上。

附加信息

- 如需有关 pod 反关联性的更多信息，请参阅使用关联性和反关联性规则相对于其他 pod 放置 pod。

6.7.2. 使用 pod 反关联性重新引导节点

Pod 反关联性和节点反关联性稍有不同。如果没有其他适当的位置来部署 pod，则可以违反节点反关联性。Pod 反关联性可以设置为必要的或偏好的。

在这个版本中，如果只有两个基础架构节点可用，且一个节点被重新引导，容器镜像 registry Pod 将无法在另一个节点上运行。**oc get pods** 将 pod 报告为 unready，直到有合适的节点可用为止。一旦某个节点可用，并且所有 pod 恢复到就绪状态，下一个节点就可以重启。

流程

使用 pod 反关联性重新引导节点：

1. 编辑节点规格以配置 pod 反关联性：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: (1)
      preferredDuringSchedulingIgnoredDuringExecution: (2)
      - weight: 100 (3)
        podAffinityTerm:
          labelSelector:
            matchExpressions:
            - key: registry (4)
              operator: In (5)
              values:
              - default
          topologyKey: kubernetes.io/hostname
#...
```

(1) 用于配置 pod 反关联性的小节。

(2) 定义偏好规则。

(3) 为偏好规则指定权重。优先选择权重最高的节点。

(4) 描述用来决定何时应用反关联性规则的 pod 标签。指定标签的键和值。

(5) 运算符表示现有 pod 上的标签和新 pod 规格中 **matchExpression** 参数的值集合之间的关系。可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。

本例假定容器镜像 registry pod 具有 **registry=default** 标签。Pod 反关联性可以使用任何 Kubernetes 匹配表达式。

2. 在调度策略文件中启用 **MatchInterPodAffinity** 调度程序 predicate。

3. 对节点执行正常重启。

6.7.3. 了解如何重新引导运行路由器的节点

在大多数情况下，运行 OpenShift Container Platform 路由器的 pod 会公开一个主机端口。

PodFitsPorts 调度程序 predicate 确保没有使用相同端口的其他路由器 pod 在同一节点上运行，并且达成 pod 反关联性。如果路由器依赖 IP 故障转移来实现高可用性，则不需要任何其他操作。

如果路由器 pod 依赖 AWS Elastic Load Balancing 等外部服务来实现高可用性，则由该服务负责响应路由器 pod 重启。

在极少见的情形中，路由器 pod 可能没有配置主机端口。这时，务必要按照推荐的基础架构节点重启流程来进行操作。

6.7.4. 正常重新引导节点

在重启节点前，建议备份 etcd 数据以避免该节点上出现数据丢失。

注意

对于需要用户执行 **oc login** 命令而不是 **kubeconfig** 文件中的证书来管理集群的单节点 OpenShift 集群，**oc adm** 命令在封锁并排空节点后可能无法使用。这是因为 **openshift-oauth-apiserver** pod 没有运行，因为 cordon。您可以使用 SSH 访问节点，如以下步骤所示。

在单节点 OpenShift 集群中，在封锁和排空时无法重新调度 Pod。但是，这样做会为 pod（特别是您的工作负载 pod）提供一定的时间来正确停止和释放相关资源。

流程

执行节点正常重启：

1. 将节点标记为不可调度：

```
$ oc adm cordon <node1>
```

2. 排空节点以删除所有正在运行的 pod：

```
$ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data --force
```

您可能会收到与自定义 pod 中断预算(PDB)关联的 pod 无法被驱除的错误。

错误示例

```
error when evicting pods/"rails-postgresql-example-1-72v2w" -n "rails" (will retry after 5s):
Cannot evict pod as it would violate the pod's disruption budget.
```

在这种情况下，再次运行 drain 命令，添加 **disable-eviction** 标记，这将绕过 PDB 检查：

```
$ oc adm drain <node1> --ignore-daemonsets --delete-emptydir-data --force --disable-
eviction
```

3. 以 debug 模式访问节点：

```
$ oc debug node/<node1>
```

4. 将您的根目录改为 **/host** ：

```
$ chroot /host
```

5. 重启节点：

```
$ systemctl reboot
```

目前，节点进入 **NotReady** 状态。

注意

在一些单节点 OpenShift 集群中，**oc** 命令在 cordon 和 drain 节点后可能无法使用，因为 **openshift-oauth-apiserver** pod 没有运行。您可以使用 SSH 连接到节点并执行重启。

```
$ ssh core@<master-node>.<cluster_name>.<base_domain>
```

```
$ sudo systemctl reboot
```

6. 重启完成后，运行以下命令将节点标记为可以调度：

```
$ oc adm uncordon <node1>
```

注意

在一些单节点 OpenShift 集群中，**oc** 命令在 cordon 和 drain 节点后可能无法使用，因为 **openshift-oauth-apiserver** pod 没有运行。您可以使用 SSH 连接到节点并取消封锁它。

```
$ ssh core@<target_node>
```

```
$ sudo oc adm uncordon <node> --kubeconfig /etc/kubernetes/static-pod-resources/kube-apiserver-certs/secrets/node-kubeconfigs/localhost.kubeconfig
```

7. 验证节点是否已就绪：

```
$ oc get node <node1>
```

输出示例

```
NAME    STATUS  ROLES    AGE    VERSION
<node1> Ready   worker   6d22h  v1.18.3+b0068a8
```

附加信息

有关 etcd 数据备份的详情，请参考 备份 etcd 数据。

6.8. 使用垃圾回收释放节点资源

作为管理员，您可以通过垃圾回收来释放资源，从而使用 OpenShift Container Platform 确保节点高效运行。

OpenShift Container Platform 节点执行两种类型的垃圾回收：

- 容器垃圾回收：移除已终止的容器。
- 镜像垃圾回收：移除没有被任何正在运行的 pod 引用的镜像。

6.8.1. 了解如何通过垃圾回收移除已终止的容器

容器垃圾回收使用驱除阈值移除已终止的容器。

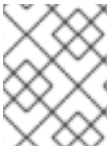
为垃圾回收设定了驱除阈值时，节点会尝试为任何可从 API 访问的 pod 保留容器。如果 pod 已被删除，则容器也会被删除。只要 pod 没有被删除且没有达到驱除阈值，容器就会保留。如果节点遭遇磁盘压力，它会移除容器，并且无法再通过 **oc logs** 访问其日志。

- **eviction-soft** - 软驱除阈值将驱除阈值与一个由管理员指定的必要宽限期配对。
- **removal-hard** - 硬驱除阈值没有宽限期，如果观察到，OpenShift Container Platform 就会立即采取行动。

下表列出了驱除阈值：

表 6.2. 用于配置容器垃圾回收的变量

节点状况	驱除信号	描述
MemoryPressure	**memory.available**	节点上的可用内存。
DiskPressure	• **nodefs.available** • **nodefs.inodesFree** • **imagefs.available** • **imagefs.inodesFree**	节点根文件系统、**nodefs** 或镜像文件系统 **imagefs** 上的可用磁盘空间或索引节点。



注意

对于 **evictionHard**，您必须指定所有这些参数。如果没有指定所有参数，则只应用指定的参数，垃圾回收将无法正常工作。

如果节点在软驱除阈值上下浮动，但没有超过其关联的宽限期，则对应的节点将持续在 **true** 和 **false** 之间振荡。因此，调度程序可能会做出不当的调度决策。

要防止这种情况的出现，请使用 **remove-pressure-transition-period** 标记来控制 OpenShift Container Platform 在摆脱压力状况前必须等待的时间。OpenShift Container Platform 不会设置在状况切换回 false 前，在指定期限内针对指定压力状况满足的驱除阈值。

6.8.2. 了解如何通过垃圾回收移除镜像

镜像垃圾回收会删除未被任何正在运行的 pod 引用的镜像。

OpenShift Container Platform 根据 cAdvisor 报告的磁盘用量决定要从节点中删除哪些镜像。

镜像垃圾收集策略基于两个条件：

- 触发镜像垃圾回收的磁盘用量百分比（以整数表示）。默认值为 85。
- 镜像垃圾回收试尝试释放的磁盘用量百分比（以整数表示）。默认值为 80。

对于镜像垃圾回收，您可以使用自定义资源修改以下任意变量。

表 6.3. 用于配置镜像垃圾回收的变量

设置	描述
imageMinimumGCAge	在通过垃圾收集移除镜像前，未用镜像的最小年龄。默认值为 2m。
imageGCHighThresholdPercent	触发镜像垃圾回收的磁盘用量百分比，以整数表示。默认值为 85。
imageGCLowThresholdPercent	镜像垃圾回收试尝试释放的磁盘用量百分比，以整数表示。默认值为 80。

每次运行垃圾收集器都会检索两个镜像列表：

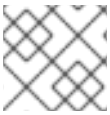
1. 目前在至少一个 pod 中运行的镜像的列表。
2. 主机上可用镜像的列表。

随着新容器运行，新镜像即会出现。所有镜像都标有时间戳。如果镜像正在运行（上方第一个列表）或者刚被检测到（上方第二个列表），它将标上当前的时间。其余镜像的标记来自于以前的运行。然后，所有镜像都根据时间戳进行排序。

一旦开始回收，首先删除最旧的镜像，直到满足停止条件。

6.8.3. 为容器和镜像配置垃圾回收

作为管理员，您可以通过为各个机器配置池创建 **kubeletConfig** 对象来配置 OpenShift Container Platform 执行垃圾回收的方式。



注意

OpenShift Container Platform 只支持每个机器配置池的一个 **kubeletConfig** 对象。

您可以配置以下几项的任意组合：

- 容器软驱除
- 容器硬驱除
- 镜像驱除

容器垃圾回收会移除已终止的容器。镜像垃圾回收会删除未被任何正在运行的 pod 引用的镜像。

先决条件

1. 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** CRD 关联的标签：

```
$ oc edit machineconfigpool <name>
```

例如：

```
$ oc edit machineconfigpool worker
```

输出示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: "" (1)
  name: worker
#...
```

(1) 标签会出现在 Labels 下。

提示

如果标签不存在，请添加键/值对，例如：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

重要

如果只有一个文件系统，或者 **/var/lib/kubelet** 和 **/var/lib/containers/** 位于同一文件系统中，则具有最高值的设置会触发驱除操作，因为它们被首先满足。文件系统会触发驱除。

容器垃圾回收 CR 的配置示例：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: worker-kubeconfig (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (2)
  kubeletConfig:
```

```
    evictionSoft: 3
      memory.available: "500Mi" 4
      nodefs.available: "10%"
      nodefs.inodesFree: "5%"
      imagefs.available: "15%"
      imagefs.inodesFree: "10%"
    evictionSoftGracePeriod: 5
      memory.available: "1m30s"
      nodefs.available: "1m30s"
      nodefs.inodesFree: "1m30s"
      imagefs.available: "1m30s"
      imagefs.inodesFree: "1m30s"
    evictionHard: 6
      memory.available: "200Mi"
      nodefs.available: "5%"
      nodefs.inodesFree: "4%"
      imagefs.available: "10%"
      imagefs.inodesFree: "5%"
    evictionPressureTransitionPeriod: 0s 7
    imageMinimumGCAge: 5m 8
    imageGCHighThresholdPercent: 80 9
    imageGCLowThresholdPercent: 75 10
#...
```

1. 对象的名称。
2. 指定机器配置池中的标签。
3. 对于容器垃圾回收：**eviction Soft** 或 **evictionHard** 的驱除类型。
4. 对于容器垃圾回收：根据特定驱除触发器信号驱除阈值。
5. 对于容器垃圾回收：软驱除周期。此参数不适用于 **eviction-hard**。
6. 对于容器垃圾回收：根据特定驱除触发器信号驱除阈值。对于 **evictionHard**，您必须指定所有这些参数。如果没有指定所有参数，则只应用指定的参数，垃圾回收将无法正常工作。
7. 对于容器垃圾回收： 摆脱驱除压力状况前等待的持续时间。
8. 对于镜像垃圾回收：在镜像被垃圾回收移除前，未使用的镜像的最小期限。
9. 对于镜像垃圾回收：触发镜像垃圾回收的磁盘用量百分比（以整数表示）。
10. 对于镜像垃圾回收：镜像垃圾回收尝试释放的磁盘用量百分比（以整数表示）。

2. 运行以下命令来创建 CR：

```
$ oc create -f <file_name>.yaml
```

例如：

```
$ oc create -f gc-container.yaml
```

输出示例

```
kubeletconfig.machineconfiguration.openshift.io/gc-container created
```

验证

1. 输入以下命令验证垃圾回收是否活跃。您在自定义资源中指定的 Machine Config Pool 会将 **UPDATING** 显示为"true"，直到更改完全实施为止：

```
$ oc get machineconfigpool
```

输出示例

```
NAME     CONFIG                                UPDATED   UPDATING
master   rendered-master-546383f80705bd5aeaba93   True      False
worker   rendered-worker-b4c51bb33ccaae6fc4a6a5   False     True
```

6.9. 为 OPENSHIFT CONTAINER PLATFORM 集群中的节点分配资源

为提供更可靠的调度并最大程度减少节点资源过量使用，请保留一部分 CPU 和内存资源供底层节点组件（如 **kubelet** 和 **kube-proxy**）以及其余系统组件（如 **sshd** 和 **NetworkManager**）使用。通过指定要保留的资源，您可以为调度程序提供有关节点可用于 pod 使用的剩余 CPU 和内存资源的更多信息。您可以允许 OpenShift Container Platform 为您的节点自动决定最佳的 **system-reserved** CPU 和内存资源，也可以为节点手动决定并设置最佳资源。

重要

要手动设置资源值，您必须使用 kubelet 配置 CR。您不能使用机器配置 CR。

6.9.1. 了解如何为节点分配资源

OpenShift Container Platform 中为节点组件保留的 CPU 和内存资源基于两个节点设置：

设置	描述
kube-reserved	此设置不会用于 OpenShift Container Platform。将您要保留的 CPU 和内存资源添加到 **system-reserved** 设置中。
system-reserved	此设置标识要为节点组件和系统组件（如 CRI-O 和 Kubelet）保留的资源。默认设置取决于 OpenShift Container Platform 和 Machine Config Operator 版本。确认 **machine-config-operator** 仓库中的默认 **systemReserved** 参数。

如果没有设置标志，则使用默认值。如果未设置任何标记，则分配的资源设置为引入可分配资源前的节点容量。

注意

任何使用 **reservedSystemCPUs** 参数特别保留的 CPU 都无法使用 **kube-reserved** 或 **system-reserved** 进行分配。

6.9.1.1. OpenShift Container Platform 如何计算分配的资源

分配的资源数量根据以下公式来计算：

```
[Allocatable] = [Node Capacity] - [system-reserved] - [Hard-Eviction-Thresholds]
```

注意

Allocatable 提供的 **Hard-Eviction-Thresholds** 可提高系统可靠性，因为 **Allocatable** 的值在节点级别强制实施。

如果 **Allocatable** 为负值，它会被设为 **0**。

每个节点报告容器运行时和 kubelet 使用的系统资源。为简化配置 **system-reserved** 参数，请使用节点概述 API 查看用于节点的资源。节点概述位于 **/api/v1/nodes/<node\" /proxy/stats/summary**。

6.9.1.2. 节点如何强制实施资源限制

节点可以根据配置的可分配值限制 pod 可消耗的资源总量。此功能可以防止 pod 使用系统服务（如容器运行时和节点代理）所需的 CPU 和内存资源，从而显著提高节点可靠性。为提高节点可靠性，管理员应该根据目标保留资源使用。

节点使用一个新的 cgroup 分级结构来强制实施对资源的约束。它可以强制实现对服务质量的要求。所有 pod 都在专用的 cgroup 层次结构中启动，与系统守护进程隔离。

管理员应该像对待具有保证服务质量的 pod 一样对待系统守护进程。系统守护进程可能会在其限定控制组中爆发，此行为需要作为集群部署的一个部分进行管理。通过在 **system-reserved** 中指定 CPU 和内存资源量，为系统守护进程保留 CPU 和内存资源。

强制实施 **system-reserved** 限制可防止关键系统服务接收 CPU 和内存资源。因此，关键系统服务可能会被内存不足 killer 结束。我们的建议是，只在您为节点进行了详细配置后，强制实施 **system-reserved**，且您可以确定，如果关键系统服务因为该组中的任何进程导致内存不足 killer 终止它时，可以恢复。

6.9.1.3. 了解驱除阈值

如果某个节点面临内存压力，这可能会影响整个节点以及该节点上运行的所有 pod。例如，使用超过保留内存量的系统守护进程可触发内存不足事件。为避免系统耗尽或降低内存不足事件的可能性，节点会提供处理资源不足情况的功能。

您可以使用 **--eviction-hard** 标记保留一些内存。每当节点上的内存可用量低于该绝对值或百分比时，节点会尝试驱除 pod。如果节点上没有系统守护进程，pod 的内存会被限制在 **capacity - eviction-hard** 内。因此，pod 不能使用作为达到内存不足状态前驱除缓冲量而预留的资源。

下例演示了节点内存可分配量的影响：

- 节点容量为 **32Gi**
- --system-reserved 为 **3Gi**
- --eviction-hard 设置为 **100Mi**。

对于这个节点，有效节点可分配量的值是 **28.9Gi**。如果节点和系统组件使用其所有保留量，则 pod 的可用内存为 **28.9Gi**，并且 kubelet 会在超过这个阈值时驱除 pod。

如果您通过顶级 cgroup 强制实施节点可分配量 (**28.9Gi**)，那么 pod 永不会超过 **28.9Gi**。除非系统守护进程消耗的内存超过 **3.1Gi**，否则不会执行驱除。

如果系统守护进程没有用尽其所有保留量，那么在上例中，pod 会在节点开始驱除前面临被其限定 cgroup 执行 memcg OOM 终止的问题。为了在这种情况下更好地强制实施 QoS，节点会对所有 pod 的顶级 cgroup 应用硬驱除阈值，即 **Node Allocatable + Eviction Hard Thresholds**。

如果系统守护进程没有用尽所有保留量，每当 pod 消耗的内存超过 **28.9Gi** 时，节点就会驱除 pod。如果不及时驱除，消耗的内存超过 **29Gi** 时就会对 pod 执行 OOM 终止。

6.9.1.4. 调度程序如何确定资源可用性

调度程序使用 **node.Status.Allocatable**（而非 **node.Status.Capacity**）的值来决定节点是否成为 pod 调度的候选者。

在默认情况下，节点会将其机器容量报告为可完全被集群调度。

6.9.2. 自动为节点分配资源

OpenShift Container Platform 可以自动决定与特定机器配置池关联的节点的最佳 **system-reserved** CPU 和内存资源，并在节点启动时使用这些值更新节点。默认情况下，**system-reserved** CPU 为 **500m**，**system-reserved** 内存为 **1Gi**。

要在节点上自动决定并分配 **system-reserved** 资源，请创建一个 **KubeletConfig** 自定义资源（CR）来设置 **autoSizingReserved: true** 参数。各个节点上的脚本根据每个节点上安装的 CPU 和内存容量，计算相应保留资源的最佳值。该脚本考虑了增加的容量要求保留资源的相应增加。

自动确定最佳的 **system-reserved** 设置可确保集群高效运行，并防止因为 CRI-O 和 kubelet 等系统组件的资源丢失而出现节点故障，而无需手动计算和更新值。

此功能默认为禁用。

先决条件

1. 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** 对象关联的标签：

```
$ oc edit machineconfigpool <name>
```

例如：

```
$ oc edit machineconfigpool worker
```

输出示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: "" (1)
  name: worker
#...
```

(1) 标签会出现在 **Labels** 下。

提示

如果没有适当的标签，请添加键/值对，例如：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源（CR）：

资源分配 CR 的示例配置

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: dynamic-node (1)
spec:
  autoSizingReserved: true (2)
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (3)
#...
```

(1) 为 CR 分配一个名称。

(2) 将 **autoSizingReserved** 参数设置为 **true**，以允许 OpenShift Container Platform 在与指定标签关联的节点上自动决定并分配 **system-reserved** 资源。要在这些节点上禁用自动分配，请将此参数设置为 **false**。

(3) 指定您在"先决条件"部分中配置的机器配置池中的标签。您可以为机器配置池选择任何所需的标签，如 **custom-kubelet: small-pods** 或默认标签 **pools.operator.machineconfiguration.openshift.io/worker: ""**。

前面的示例在所有 worker 节点上启用自动资源分配。OpenShift Container Platform 排空节点、应用 kubelet 配置并重启节点。

2. 运行以下命令来创建 CR：

```
$ oc create -f <file_name>.yaml
```

验证

1. 输入以下命令登录到您配置的节点：

```
$ oc debug node/<node_name>
```

2. 将 **/host** 设置为 debug shell 中的根目录：

```
# chroot /host
```

3. 查看 **/etc/node-sizing.env** 文件：

输出示例

```
SYSTEM_RESERVED_MEMORY=3Gi
SYSTEM_RESERVED_CPU=0.08
```

kubelet 使用 **/etc/node-sizing.env** 文件中的 **system-reserved** 值。在上例中，worker 节点分配了 **0.08** CPU 和 3 Gi 内存。显示最佳值可能需要几分钟时间。

6.9.3. 手动为节点分配资源

OpenShift Container Platform 支持对 CPU 和内存资源类型执行分配。还支持 **ephemeral-resource** 资源类型。对于 **cpu** 类型，您可以以内核数为单位指定资源数量，如 **200m**、**0.5** 或 **1**。对于 **memory** 和 **ephemeral-storage**，您可以指定资源数量（以字节为单位），如 **200Ki**、**50Mi** 或 **5Gi**。默认情况下，**system-reserved** CPU 为 **500m**，**system-reserved** 内存为 **1Gi**。

作为管理员，您可以通过一组 **<resource_type>=<resource_quantity>** 对（如 **cpu=200m,memory=512Mi**）来设置这些值。

重要

您必须使用 kubelet 配置 CR 来手动设置资源值。您不能使用机器配置 CR。

有关推荐的 **system-reserved** 值的详情，请参考推荐的 system-reserved 值。

先决条件

1. 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** CRD 关联的标签：

```
$ oc edit machineconfigpool <name>
```

例如：

```
$ oc edit machineconfigpool worker
```

输出示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: "" 1
  name: worker
#...
```

1 标签会出现在 Labels 下。

提示

如果标签不存在，请添加键/值对，例如：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

资源分配 CR 的示例配置

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-allocatable (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (2)
  kubeletConfig:
    systemReserved: (3)
      cpu: 1000m
      memory: 1Gi
#...
```

(1) 为 CR 分配一个名称。

(2) 指定机器配置池中的标签。

(3) 为节点组件和系统组件指定要保留的资源。

2. 运行以下命令来创建 CR：

```
$ oc create -f <file_name>.yaml
```

6.10. 为集群中的节点分配特定 CPU

使用静态 CPU Manager 策略时，您可以保留特定的 CPU 供集群中的特定节点使用。例如，在具有 24 个 CPU 的系统中，您可以为 control plane 保留编号为 0-3 的 CPU，允许计算节点使用 CPU 4 到 23。

6.10.1. 为节点保留 CPU

要明确定义为特定节点保留的 CPU 列表，请创建一个 **KubeletConfig** 自定义资源（CR）来定义 **reservedSystemCPUs** 参数。此列表替代了使用 **systemReserved** 参数可能保留的 CPU。

流程

1. 为您要配置的节点类型获取与机器配置池（MCP）关联的标签：

```
$ oc describe machineconfigpool <name>
```

例如：

```
$ oc describe machineconfigpool worker
```

输出示例

```
Name:         worker
Namespace:
Labels:       machineconfiguration.openshift.io/mco-built-in=
              pools.operator.machineconfiguration.openshift.io/worker= (1)
Annotations:  <none>
API Version:  machineconfiguration.openshift.io/v1
Kind:         MachineConfigPool
#...
```

(1) 获取 MCP 标签。

2. 为 **KubeletConfig** CR 创建 YAML 文件：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-reserved-cpus (1)
spec:
  kubeletConfig:
    reservedSystemCPUs: "0,1,2,3" (2)
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (3)
#...
```

(1) 为 CR 指定一个名称。

(2) 为与 MCP 关联的节点指定您要保留的 CPU 的内核 ID。

(3) 指定来自 MCP 的标签。

3. 创建 CR 对象：

```
$ oc create -f <file_name>.yaml
```

其他资源

- 如需有关 **systemReserved** 参数的更多信息，请参阅为 OpenShift Container Platform 集群中的节点分配资源。

6.11. 为 KUBELET 启用 TLS 安全配置集

您可以使用 TLS（端口层安全）安全配置集定义 kubelet 在充当 HTTP 服务器时需要哪些 TLS 密码。kubelet 使用其 HTTP/GRPC 服务器与 Kubernetes API 服务器通信，后者向 pod 发送命令，收集日志，并通过 kubelet 对 pod 运行 exec 命令。

TLS 安全配置集定义了 Kubernetes API 服务器连接 kubelet 时必须使用的 TLS 密码，以保护 kubelet 和 Kubernetes API 服务器之间的通信。

注意

默认情况下，当 kubelet 充当 Kubernetes API 服务器的客户端时，它会自动与 API 服务器协商 TLS 参数。

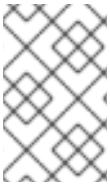
6.11.1. 了解 TLS 安全配置集

您可以使用 TLS（Transport Layer Security）安全配置集来定义各种 OpenShift Container Platform 组件需要哪些 TLS 密码。OpenShift Container Platform TLS 安全配置集基于 Mozilla 推荐的配置。

您可以为每个组件指定以下 TLS 安全配置集之一：

表 6.4. TLS 安全配置集

profile	描述
Old	此配置集用于旧的客户端或库。该配置集基于旧的向后兼容性建议配置。 **Old** 配置集要求最低 TLS 版本 1.0。  **注意** 对于 Ingress Controller，最小 TLS 版本从 1.0 转换为 1.1。
Intermediate	这个配置集是大多数客户端的建议配置。它是 Ingress Controller、kubelet 和 control plane 的默认 TLS 安全配置集。该配置集基于 Intermediate 兼容性推荐的配置。 **Intermediate** 配置集需要最小 TLS 版本 1.2。
Modern	此配置集主要用于不需要向后兼容的现代客户端。这个配置集基于 Modern 兼容性推荐的配置。 **Modern** 配置集需要最低 TLS 版本 1.3。
Custom	此配置集允许您定义要使用的 TLS 版本和密码。  **警告** 使用 **Custom** 配置集时要谨慎，因为无效的配置可能会导致问题。



注意

当使用预定义的配置集类型时，有效的配置集配置可能会在发行版本之间有所改变。例如，使用在版本 X.Y.Z 中部署的 Intermediate 配置集指定了一个规格，升级到版本 X.Y.Z+1 可能会导致应用新的配置集配置，从而导致推出部署。

6.11.2. 为 kubelet 配置 TLS 安全配置集

要在作为 HTTP 服务器时为 kubelet 配置 TLS 安全配置集，请创建一个 **KubeletConfig** 自定义资源（CR）来为特定节点指定预定义或自定义 TLS 安全配置集。如果没有配置 TLS 安全配置集，则默认 TLS 安全配置集为 **Intermediate**。

在 worker 节点上配置 Old TLS 安全配置集的 KubeletConfig CR 示例

```
apiVersion: config.openshift.io/v1
kind: KubeletConfig
 ...
spec:
  tlsSecurityProfile:
    old: {}
    type: Old
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: ""
#...
```

您可以在配置的节点上的 **kubelet.conf** 文件中看到配置 TLS 安全配置集的密码和最小 TLS 版本。

先决条件

- 以具有 **cluster-admin** 角色的用户身份登录到 OpenShift Container Platform。

流程

1. 创建 **KubeletConfig** CR 来配置 TLS 安全配置集：

   Custom 配置集的 KubeletConfig CR 示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-kubelet-tls-security-profile
spec:
  tlsSecurityProfile:
    type: Custom (1)
    custom: (2)
      ciphers: (3)
      - ECDHE-ECDSA-CHACHA20-POLY1305
      - ECDHE-RSA-CHACHA20-POLY1305
      - ECDHE-RSA-AES128-GCM-SHA256
      - ECDHE-ECDSA-AES128-GCM-SHA256
      minTLSVersion: VersionTLS11
  machineConfigPoolSelector:
```

```
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" 4
#...
```

1 指定 TLS 安全配置集类型（**Old**、**Intermediate** 或 **Custom**）。默认值为 **Intermediate**。

2 为所选类型指定适当的字段：

- **old: {}**
- **intermediate: {}**
- **custom:**

3 对于 **custom** 类型，请指定 TLS 密码列表和最低接受的 TLS 版本。

4 可选：为您要应用 TLS 安全配置集的节点指定机器配置池标签。

2. 创建 **KubeletConfig** 对象：

```
$ oc create -f <filename>
```

根据集群中的 worker 节点数量，等待配置的节点被逐个重启。

验证

要验证是否设置了配置集，请在节点处于 **Ready** 状态后执行以下步骤：

1. 为配置的节点启动 debug 会话：

```
$ oc debug node/<node_name>
```

2. 将 **/host** 设置为 debug shell 中的根目录：

```
sh-4.4# chroot /host
```

3. 查看 **kubelet.conf** 文件：

```
sh-4.4# cat /etc/kubernetes/kubelet.conf
```

输出示例

```
  "kind": "KubeletConfiguration",
  "apiVersion": "kubelet.config.k8s.io/v1beta1",
#...
  "tlsCipherSuites": [
    "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_CHACHA20_POLY1305_SHA256",
    "TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256"
```

```
    ],
    "tlsMinVersion": "VersionTLS12",
  #...
```

6.12. 创建基础架构节点

重要

您只能在 Machine API 操作的集群中使用高级机器管理和扩展功能。具有用户置备的基础架构的集群需要额外的验证和配置才能使用 Machine API。

具有基础架构平台类型 **none** 的集群无法使用 Machine API。即使附加到集群的计算机器安装在支持该功能的平台上，也会应用这个限制。在安装后无法更改此参数。

要查看集群的平台类型，请运行以下命令：

```
$ oc get infrastructure cluster -o jsonpath='{.status.platform}'
```

您可以使用基础架构机器集来创建仅托管基础架构组件的机器，如默认路由器、集成的容器镜像 registry 以及集群指标和监控的组件。这些基础架构机器不会被计算为运行环境所需的订阅总数。

在生产部署中，建议您至少部署三个机器集来容纳基础架构组件。OpenShift Logging 和 Red Hat OpenShift Service Mesh 部署 Elasticsearch，这需要三个实例安装到不同的节点上。这些节点都可以部署到不同的可用区以实现高可用性。此配置需要三个不同的机器集，每个可用区都有一个。在没有多个可用区的全局 Azure 区域，您可以使用可用性集来确保高可用性。

注意

在基础架构节点上添加 **NoSchedule** 污点后，在该节点上运行的现有 DNS pod 被标记为 **misscheduled**。您必须删除或在 **misscheduled** DNS pod 中添加容限。

6.12.1. OpenShift Container Platform 基础架构组件

每个自我管理的 Red Hat OpenShift 订阅都包括 OpenShift Container Platform 和其他 OpenShift 相关组件的权利。这些权利包括在运行 OpenShift Container Platform control plane 和基础架构工作负载时，不需要在大小期间考虑这些权利。

要有资格成为基础架构节点并使用包含的权利，只有支持集群的组件，而不是最终用户应用程序的一部分，才能在这些实例上运行。示例包括以下组件：

- Kubernetes 和 OpenShift Container Platform control plane 服务
- 默认路由器
- 集成的容器镜像 registry
- 基于 HAProxy 的 Ingress Controller
- 集群指标集合或监控服务，包括监控用户定义的项目的组件
- 集群聚合日志
- Red Hat Quay

- Red Hat OpenShift Data Foundation
- Red Hat Advanced Cluster Management for Kubernetes
- Red Hat Advanced Cluster Security for Kubernetes
- Red Hat OpenShift GitOps
- Red Hat OpenShift Pipelines
- Red Hat OpenShift Service Mesh

运行任何其他容器、Pod 或组件的所有节点都需要是您的订阅可涵盖的 worker 节点。

有关基础架构节点以及可在基础架构节点上运行，请参阅 OpenShift sizing and subscription guide for enterprise Kubernetes 文档中的 "Red Hat OpenShift control plane and infrastructure nodes"部分。

要创建基础架构节点，您可以使用机器集，标记节点，或使用机器配置池。

6.12.1.1. 创建基础架构节点

重要

请参阅为安装程序置备的基础架构环境创建基础架构机器集，或为其 control plane 节点由机器 API 管理的任何集群创建基础架构机器集。

集群的基础架构系统（也称为 **infra 节点**）的要求已被置备。安装程序只为 control plane 和 worker 节点提供置备。Worker 节点可以通过标记来指定为基础架构节点或应用程序（也称为 **app** ）。

流程

1. 向您要充当应用程序节点的 worker 节点添加标签：

```
$ oc label node <node-name> node-role.kubernetes.io/app=""
```

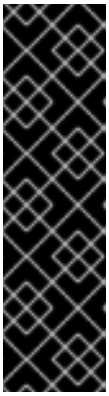
2. 向您要充当基础架构节点的 worker 节点添加标签：

```
$ oc label node <node-name> node-role.kubernetes.io/infra=""
```

3. 检查相关节点现在是否具有 **infra** 角色或 **app** 角色：

```
$ oc get nodes
```

4. 创建默认的集群范围节点选择器。默认节点选择器应用到在所有命名空间中创建的 pod。这会创建一个与 pod 上任何现有节点选择器交集的交集，这会额外限制 pod 的选择器。



重要

如果默认节点选择器键与 pod 标签的键冲突，则不会应用默认节点选择器。

但是，不要设置可能会导致 pod 变得不可调度的默认节点选择器。例如，当 pod 的标签被设置为不同的节点角色（如 **node-role.kubernetes.io/infra=""**）时，将默认节点选择器设置为特定的节点角色（如 **node-role.kubernetes.io/master=""**）可能会导致 pod 无法调度。因此，将默认节点选择器设置为特定节点角色时要小心。

您还可以使用项目节点选择器来避免集群范围节点选择器键冲突。

a. 编辑 **Scheduler** 对象：

```
$ oc edit scheduler cluster
```

b. 使用适当的节点选择器添加 **defaultNodeSelector** 字段：

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
spec:
  defaultNodeSelector: node-role.kubernetes.io/infra="" (1)
# ...
```

(1) 这个示例节点选择器默认在基础架构节点上部署 pod。

c. 保存文件以使改变生效。

现在，您可以将基础架构资源移到新标记的 **infra** 节点。

其他资源

- 将资源移到基础架构机器集

第 7 章 操作容器

7.1. 了解容器

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。OpenShift Container Platform 和 Kubernetes 增加了在多主机安装之间编排容器的功能。

7.1.1. 关于容器和 RHEL 内核内存

由于 Red Hat Enterprise Linux（RHEL）行为，CPU 使用率高的容器可能比预期消耗的内存多。较高的内存消耗可能是由 RHEL 内核中的 **kmem_cache** 造成的。RHEL 内核为每个 cgroup 创建一个 **kmem_cache**。为添加性能，**kmem_cache** 包含 **cpu_cache** 以及任何 NUMA 节点的节点缓存。这些缓存都消耗内核内存。

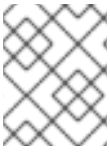
保存在这些缓存中的内存量与系统使用的 CPU 数量成比例。因此，有大量 CPU 会导致更多的内核内存被保存在这些缓存中。这些缓存中有大量内核内存可导致 OpenShift Container Platform 容器超过配置的内存限值，从而导致容器被终止。

为了避免因为内核内存问题而丢失容器，请确保容器请求足够的内存。您可以使用以下公式来估算 **kmem_cache** 所消耗的内存数量，其中 **nproc** 是 **nproc** 命令报告的可用处理单元数。容器请求的下限应该是这个值加上容器内存要求：

```
$(nproc) X 1/2 MiB
```

7.1.2. 关于容器引擎和容器运行时

*容器引擎*是处理用户请求的软件，包括命令行选项和镜像拉取。容器引擎使用 *容器运行时*（也称为 *较低级别的容器运行时*）来运行和管理部署和运行容器所需的组件。您可能需要与容器引擎或容器运行时交互。



注意

OpenShift Container Platform 文档使用 *容器运行时* 来引用低级别容器运行时。其他文档可以将容器引擎引用为容器运行时。

OpenShift Container Platform 使用 CRI-O 作为容器引擎，并运行C 或 crun 作为容器运行时。默认容器运行时为 runC。两个容器运行时遵循 开放容器项目(OCI) 运行时规格。

CRI-O 是 Kubernetes 的原生容器引擎实现，可与操作系统紧密集成来提供高效和优化的 Kubernetes 体验。CRI-O 容器引擎作为 systemd 服务在每个 OpenShift Container Platform 集群节点上运行。

runc 由 Docker 开发并由 Open Container Project 维护，它是一个轻量级、可移植的容器运行时，使用 Go. crun 编写，由红帽开发的快速且低内存容器运行时完全使用 C 编写。从 OpenShift Container Platform 4.16 开始，您可以选择两者。

crun 与 runC 相比有几个改进，包括：

- 较小的二进制文件
- 可以更快地进行处理
- 内存占用较低

runc 与 crun 相比有一些优点，包括：

- 最常用的 OCI 容器运行时。
- 在生产环境更长。
- CRI-O 的默认容器运行时。

您可以根据需要在两个容器运行时之间移动。

有关设置要使用的容器运行时的详情，请参考 创建 **ContainerRuntimeConfig** CR 以编辑 CRI-O 参数。

7.2. 在部署 POD 前使用初始容器来执行任务

OpenShift Container Platform 提供了一组*初始容器（Init Containers）*，它们是在应用程序容器之前运行的专用容器，可以包含不出现在应用程序镜像中的实用程序或设置脚本。

7.2.1. 了解初始容器

您可以在部署 pod 的其余部分之前，使用初始容器资源来执行任务。

pod 可以同时包含初始容器和应用程序容器。借助初始容器，您可以重新整理设置脚本和绑定代码。

初始容器可以：

- 包含并运行出于安全考虑而不应包括在应用容器镜像中的实用程序。
- 包含不出现在应用程序镜像中的设置的实用程序或自定义代码。例如，不需要仅仅为了在设置过程中使用 sed、awk、python 或 dig 等工具而使用 FROM 从其他镜像生成一个镜像。
- 使用 Linux 命名空间，以便使用与应用程序容器不同的文件系统，如访问应用程序容器无法访问的 Secret。

各个初始容器必须成功完成，然后下一个容器才能启动。因此，初始容器提供了一种简单的方法来阻止或延迟应用程序容器的启动，直至满足一定的前提条件。

例如，您可以通过如下一些方式来使用初始容器：

- 通过类似以下示例的 shell 命令，等待创建服务：

```
for i in {1..100}; do sleep 1; if dig myservice; then exit 0; fi; done; exit 1
```

- 通过类似以下示例的命令，从 Downward API 将此 Pod 注册到远程服务器：

```
$ curl -X POST
http://$MANAGEMENT_SERVICE_HOST:$MANAGEMENT_SERVICE_PORT/register -d
‘instance=$()&ip=$()’
```

- 通过类似 **sleep 60** 的命令，等待一段时间后再启动应用程序容器。

- 将一个 git 存储库克隆到卷中。
- 将值放在配置文件中，并且运行模板工具为主应用程序容器动态生成配置文件。例如，将 POD_IP 值放在配置中，并且使用 Jinja 生成主应用程序配置文件。

如需更多信息，请参阅 Kubernetes 文档。

7.2.2. 创建初始容器

下例概述了一个包含两个初始容器的简单 Pod。一个用于等待 **myservice**，另一个用于等待 **mydb**。两个容器完成后，pod 都会启动。

流程

1. 为初始容器创建 pod：
   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: myapp-container
    image: registry.access.redhat.com/ubi9/ubi:latest
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  initContainers:
  - name: init-myservice
    image: registry.access.redhat.com/ubi9/ubi:latest
    command: ['sh', '-c', 'until getent hosts myservice; do echo waiting for myservice; sleep
2; done;']
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  - name: init-mydb
    image: registry.access.redhat.com/ubi9/ubi:latest
    command: ['sh', '-c', 'until getent hosts mydb; do echo waiting for mydb; sleep 2;
done;']
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

b. 创建 pod：

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

c. 查看 pod 的状态：

```
$ oc get pods
```

输出示例

```
NAME                     READY     STATUS            RESTARTS   AGE
myapp-pod                0/1       Init:0/2          0          5s
```

pod 状态 **Init:0/2** 表示它正在等待这两个服务。

2. 创建 **myservice** 服务。

a. 创建一个类似以下示例的 YAML 文件：

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

b. 创建 pod：

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

c. 查看 pod 的状态：

```
$ oc get pods
```

输出示例

```
NAME                     READY     STATUS            RESTARTS   AGE
myapp-pod                0/1       Init:1/2          0          5s
```

pod 状态 **Init:1/2** 表示它正在等待一个服务，本例中为 **mydb** 服务。

3. 创建 **mydb** 服务：

a. 创建一个类似以下示例的 YAML 文件：

```
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
```

```
    ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

b. 创建 pod：

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

c. 查看 pod 的状态：

```
$ oc get pods
```

输出示例

```
NAME                     READY    STATUS           RESTARTS   AGE
myapp-pod                1/1      Running          0          2m
```

pod 状态表示它不再等待服务并运行。

7.3. 使用卷来持久保留容器数据

容器中的文件是临时的。因此，当容器崩溃或停止时，其数据就会丢失。您可以使用*卷*来持久保留 pod 中容器使用的数据。卷是在 pod 的生命周期内保存数据的一个目录，可供 pod 中的容器访问。

7.3.1. 了解卷

卷是挂载的文件系统，供 pod 及其容器使用，可以通过多个主机上本地或网络附加存储端点来支持。默认情况下，容器不具持久性；重启之后，其中的内容会被清除。

为确保卷上的文件系统不包含任何错误，并在出现错误时尽可能进行修复，OpenShift Container Platform 在调用 **mount** 实用程序之前会先调用 **fsck**。在添加卷或更新现有卷时会出现这种情况。

最简单的卷类型是 **emptyDir**，这是单一机器上的一个临时目录。管理员也可以允许您请求自动附加到 pod 的持久性卷。

注意

如果集群管理员启用了 FSGroup 参数，则 **emptyDir** 卷存储可能会受到基于 pod FSGroup 的配额的限制。

7.3.2. 使用 OpenShift Container Platform CLI 操作卷

您可以使用 CLI 命令 **oc set volume**，为任何使用 pod 模板的对象（如复制控制器或部署配置）添加和移除卷和卷挂载。您还可以列出 pod 中的卷，或列出使用 pod 模板的任何对象。

oc set volume 命令使用以下通用语法：

```
$ oc set volume <object_selection> <operation> <mandatory_parameters> <options>
```

对象选择

在 **oc set volume** 命令中为 **object_selection** 参数指定以下内容之一：

表 7.1. 对象选择

语法	描述	示例
<object_type> <name>	选择类型为 ***<object_type>*** 的 ***<name>***。	**deploymentConfig registry**
<object_type>/<name>	选择类型为 ***<object_type>*** 的 ***<name>***。	**deploymentConfig/registry**
***<object_type>*--selector=<object_label_selector>**	选择与给定标签选择器匹配且类型为 ***<object_type>*** 的资源。	**deploymentConfig--selector="name=registry"**
***<object_type>* --all**	选择类型为 ***<object_type>*** 的所有资源。	**deploymentConfig --all**
-f 或 **--filename=*<file_name>***	用于编辑资源的文件名、目录或文件 URL。	**-f registry-deployment-config.json**

操作

为 **oc set volume** 命令中的 **operation** 参数指定 **--add** 或 **--remove**。

必要参数

所有必需的参数都特定于所选操作，并在后续小节中阐述。

选项

所有选项都特定于所选操作，并在后续小节中讨论。

7.3.3. 列出 pod 中的卷和卷挂载

您可以列出 pod 或 pod 模板中的卷和卷挂载：

流程

列出卷：

```
$ oc set volume <object_type>/<name> [options]
```

列出卷支持的选项：

选项	描述	默认
--name	卷的名称。	
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	**'*'**

例如：

- 列出 pod **p1** 的所有卷：

```
$ oc set volume pod/p1
```

- 列出在所有部署配置中定义的卷 **v1**：

```
$ oc set volume dc --all --name=v1
```

7.3.4. 将卷添加到 pod

您可以将卷和卷挂载添加到 pod。

流程

将卷和/或卷挂载添加到 pod 模板中：

```
$ oc set volume <object_type>/<name> --add [options]
```

表 7.2. 添加卷时支持的选项

选项	描述	默认
--name	卷的名称。	若未指定，则自动生成。
-t, --type	卷源的名称。支持的值有 **emptyDir**、**hostPath**、**secret**、**configmap**、**persistentVolumeClaim** 或 **projected**。	**emptyDir**
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	'*'
-m, --mount-path	所选容器内的挂载路径。不要挂载到容器 root、/ 或主机和容器中相同的任何路径。如果容器有足够权限，可能会损坏您的主机系统（如主机的 **/dev/pts** 文件）。使用 **/host** 挂载主机是安全的。	
--path	主机路径。**--type=hostPath** 的必要参数。不要挂载到容器 root、/ 或主机和容器中相同的任何路径。如果容器有足够权限，可能会损坏您的主机系统（如主机的 **/dev/pts** 文件）。使用 **/host** 挂载主机是安全的。	
--secret-name	secret 的名称。**--type=secret** 的必要参数。	
--configmap-name	configmap 的名称。**--type=configmap** 的必要参数。	

选项	描述	默认
--claim-name	持久性卷声明的名称。**--type=persistentVolumeClaim** 的必要参数。	
--source	以 JSON 字符串表示的卷源详情。如果 **--type** 不支持所需的卷源，则建议使用此参数。	
-o, --output	显示修改后的对象，而不在服务器上更新它们。支持的值有 **json** 和 **yaml**。	
--output-version	输出给定版本的修改后对象。	**api-version**

例如：

- 将新卷源 emptyDir 添加到 registry **DeploymentConfig** 对象中：

```
$ oc set volume dc/registry --add
```

提示

您还可以应用以下 YAML 来添加卷：

例 7.1. 带有添加卷的部署配置示例

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: registry
  namespace: registry
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes: 1
        - name: volume-pppsw
          emptyDir: {}
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
```

1 添加卷源 **emptyDir**。

- 为复制控制器 **r1** 添加含有 secret **secret1** 的卷 **v1** 并挂载到容器中的 */data*：

```
$ oc set volume rc/r1 --add --name=v1 --type=secret --secret-name='secret1' --mount-
path=/data
```

提示

您还可以应用以下 YAML 来添加卷：

例 7.2. 带有添加的卷和 secret 的复制控制器示例

```
kind: ReplicationController
apiVersion: v1
metadata:
  name: example-1
  namespace: example
spec:
  replicas: 0
  selector:
    app: httpd
    deployment: example-1
    deploymentconfig: example
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: httpd
        deployment: example-1
        deploymentconfig: example
    spec:
      volumes: 1
        - name: v1
          secret:
            secretName: secret1
            defaultMode: 420
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          volumeMounts: 2
            - name: v1
              mountPath: /data
```

1. 添加卷和 secret。
2. 添加容器挂载路径。

- 使用声明名称 **pvc1** 将现有持久性卷 **v1** 添加到磁盘上的部署配置 ***dc.json***，将该卷挂载到容器 **c1** 中的 ***/data*** 并更新服务器上的 **DeploymentConfig**：

```
$ oc set volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

提示

您还可以应用以下 YAML 来添加卷：

例 7.3. 添加了持久性卷的部署配置示例

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example
  namespace: example
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes:
        - name: volume-pppsw
          emptyDir: {}
        - name: v1 (1)
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts: (2)
            - name: v1
              mountPath: /data
```

(1) 添加名为"pvc1"的持久卷声明。
(2) 添加容器挂载路径。

- 为所有复制控制器添加基于 Git 存储库 **https://github.com/namespace1/project1** 且具有修订 **5125c45f9f563** 的卷 **v1**：

```
$ oc set volume rc --all --add --name=v1 \
  --source='{"gitRepo": {
              "repository": "https://github.com/namespace1/project1",
              "revision": "5125c45f9f563"
            }}'
```

7.3.5. 更新 pod 中的卷和卷挂载

您可以修改 pod 中的卷和卷挂载。

流程

使用 **--overwrite** 选项更新现有卷：

```
$ oc set volume <object_type>/<name> --add --overwrite [options]
```

例如：

- 使用现有持久性卷声明 **pvc1** 替换复制控制器 **r1** 的现有卷 **v1**：

  ```
  $ oc set volume rc/r1 --add --overwrite --name=v1 --type=persistentVolumeClaim --claim-name=pvc1
  ```

  #### 提示

  您还可以应用以下 YAML 来替换卷：

  例 7.4. 使用名为 pvc1的持久性卷声明的复制控制器示例

  ```
  kind: ReplicationController
  apiVersion: v1
  metadata:
    name: example-1
    namespace: example
  spec:
    replicas: 0
    selector:
      app: httpd
      deployment: example-1
      deploymentconfig: example
    template:
      metadata:
        labels:
          app: httpd
          deployment: example-1
          deploymentconfig: example
      spec:
        volumes:
          - name: v1 (1)
            persistentVolumeClaim:
              claimName: pvc1
        containers:
          - name: httpd
            image: >-
              image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
            ports:
              - containerPort: 8080
                protocol: TCP
            volumeMounts:
              - name: v1
                mountPath: /data
  ```

  (1) 将持久卷声明设置为 **pvc1**。

- 将卷 v1 的 **DeploymentConfig** d1 挂载点更改为 */opt*：

```
$ oc set volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

提示

您还可以应用以下 YAML 以更改挂载点：

例 7.5. 将挂载点设置为 opt 的部署配置示例。

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example
  namespace: example
spec:
  replicas: 3
  selector:
    app: httpd
  template:
    metadata:
      labels:
        app: httpd
    spec:
      volumes:
        - name: volume-pppsw
          emptyDir: {}
        - name: v2
          persistentVolumeClaim:
            claimName: pvc1
        - name: v1
          persistentVolumeClaim:
            claimName: pvc1
      containers:
        - name: httpd
          image: >-
            image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest
          ports:
            - containerPort: 8080
              protocol: TCP
          volumeMounts: 1
            - name: v1
              mountPath: /opt
```

1 将挂载点设置为 **/opt**。

7.3.6. 从 pod 中删除卷和卷挂载

您可以从 pod 中移除卷或卷挂载。

流程

从 pod 模板中移除卷：

```
$ oc set volume <object_type>/<name> --remove [options]
```

表 7.3. 移除卷时支持的选项

选项	描述	默认
--name	卷的名称。	
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	'*'
--confirm	指定您想要一次性移除多个卷。	
-o, --output	显示修改后的对象，而不在服务器上更新它们。支持的值有 **json** 和 **yaml**。	
--output-version	输出给定版本的修改后对象。	**api-version**

例如：

- 从 **DeploymentConfig** 对象 **d1** 中删除卷 **v1**：

```
$ oc set volume dc/d1 --remove --name=v1
```

- 为 **DeploymentConfig** 对象从 **d1** 的容器 **c1** 中卸载卷 **v1**，并在 **d1** 上的任何容器都没有引用时删除卷 **v1**：

```
$ oc set volume dc/d1 --remove --name=v1 --containers=c1
```

- 移除复制控制器 **r1** 的所有卷：

```
$ oc set volume rc/r1 --remove --confirm
```

7.3.7. 配置卷以在 pod 中用于多种用途

您可以使用 **volumeMounts.subPath** 属性来指定卷中的 **subPath** 而非卷的根目录，将卷配置为允许在一个 pod 中多处使用这个卷。

注意

您不能将 **subPath** 参数添加到现有调度的 pod 中。

流程

1. 要查看卷中的文件列表，请运行 **oc rsh** 命令：

```
$ oc rsh <pod>
```

输出示例

```
sh-4.2$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

2. 指定 **subPath**：

带有 subPath 参数的 Pod spec 示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
    securityContext:
      runAsNonRoot: true
      seccompProfile:
        type: RuntimeDefault
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql (1)
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html (2)
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

(1) 数据库存储在 **mysql** 文件夹中。

(2) HTML 内容存储在 **html** 文件夹中。

7.4. 使用投射卷来映射卷

*投射卷*会将几个现有的卷源映射到同一个目录中。

可以投射以下类型的卷源：

- Secret

- Config Map
- Downward API

注意

所有源都必须位于与 pod 相同的命名空间中。

7.4.1. 了解投射卷

投射卷可将这些卷源的任何组合映射到一个目录中，让用户能够：

- 使用来自多个 secret、配置映射的密钥和 downward API 信息自动填充单个卷，以便在一个目录中整合不同来源的信息；
- 使用来自多个 secret、配置映射的密钥和 downward API 信息填充单个卷，并且明确指定各个项目的路径，以便能够完全掌控卷中的内容。

重要

当在基于 Linux 的 Pod 的安全上下文中设置 **RunAsUser** 权限时，投射文件具有正确的权限集，包括容器用户所有权。但是，当 Windows pod 中设置了与 Windows 等效的 **RunAsUsername** 权限时，kubelet 将无法正确设置投射卷中的文件的所有权。

因此，在 Windows pod 的安全上下文中设置的 **RunAsUsername** 权限不适用于 OpenShift Container Platform 中运行的 Windows 项目卷。

以下一般情景演示了如何使用投射卷。

配置映射、secret、Downward API。

通过投射卷，使用包含密码的配置数据来部署容器。使用这些资源的应用程序可以在 Kubernetes 上部署 Red Hat OpenStack Platform（RHOSP）。根据服务要用于生产环境还是测试环境，可能需要对配置数据进行不同的编译。如果 pod 标记了生产或测试用途，可以使用 Downward API 选择器 **metadata.labels** 来生成正确的 RHOSP 配置。

配置映射 + secret。

借助投射卷来部署涉及配置数据和密码的容器。例如，您可以执行含有某些敏感加密任务的配置映射，这些任务需要使用保险箱密码文件来解密。

ConfigMap + Downward API。

借助投射卷来生成包含 pod 名称的配置（可通过 **metadata.name** 选择器使用）。然后，此应用程序可以将 pod 名称与请求一起传递，以在不使用 IP 跟踪的前提下轻松地判断来源。

Secret + Downward API。

借助投射卷，将 secret 用作公钥来加密 pod 的命名空间（可通过 **metadata.namespace** 选择器使用）。这个示例允许 Operator 使用应用程序安全地传送命名空间信息，而不必使用加密传输。

7.4.1.1. Pod specs 示例

以下是用于创建投射卷的 **Pod** spec 示例。

带有 secret、Downward API 和配置映射的 Pod

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: volume-test
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: container-test
    image: busybox
    volumeMounts: 1
    - name: all-in-one
      mountPath: "/projected-volume" 2
      readOnly: true 3
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes: 4
  - name: all-in-one 5
    projected:
      defaultMode: 0400 6
      sources:
      - secret:
          name: mysecret 7
          items:
            - key: username
              path: my-group/my-username 8
      - downwardAPI: 9
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap: 10
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
              mode: 0777 11
```

1. 为每个需要 secret 的容器添加 **volumeMounts** 部分。
2. 指定一个到还未使用的目录的路径，secret 将出现在这个目录中。
3. 将 **readOnly** 设为 **true**。
4. 添加一个 **volumes** 块，以列出每个投射卷源。
5. 为卷指定任意名称。

6 设置文件的执行权限。

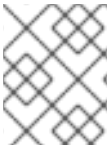
7 添加 secret。输入 secret 对象的名称。必须列出您要使用的每个 secret。

8 指定 **mountPath** 下 secret 文件的路径。此处，secret 文件位于 */projected-volume/my-group/my-username*。

9 添加 Downward API 源。

10 添加 ConfigMap 源。

11 设置具体的投射模式



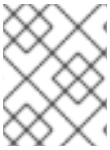
注意

如果 pod 中有多个容器，则每个容器都需要一个 **volumeMounts** 部分，但 **volumes** 部分只需一个即可。

具有设定了非默认权限模式的多个 secret 的 Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
  - name: all-in-one
    projected:
      defaultMode: 0755
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
```

```
    - key: password
      path: my-group/my-password
      mode: 511
```



注意

defaultMode 只能在投射级别上指定，而不针对每个卷源指定。但如上方所示，您可以明确设置每一个投射的 **mode**。

7.4.1.2. 路径注意事项

配置路径相同时发生密钥间冲突

如果您使用同一路径配置多个密钥，则 pod 规格会视其为有效。以下示例中为 **mysecret** 和 **myconfigmap** 指定了相同的路径：

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/data
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/data
```

请考虑以下与卷文件路径相关的情况。

未配置路径的密钥之间发生冲突

只有在创建 pod 时所有路径都已知，才会进行运行时验证，这与上述情景类似。否则发生冲突时，最新指定的资源会覆盖所有之前指定的资源（在 pod 创建后更新的资源也是如此）。

一个路径为显式而另一个路径为自动投射时发生冲突

如果因为用户指定的路径与自动投射的数据匹配，从而发生冲突，则像前文所述一样，后面的资源将覆盖前面的资源

7.4.2. 为 Pod 配置投射卷

在创建投射卷时，请注意 *了解投射卷*中介绍的卷文件路径情况。

以下示例演示了如何使用投射卷挂载现有的 secret 卷源。可以使用这些步骤从本地文件创建用户名和密码 secret。然后，创建一个只运行一个容器的 pod，使用投射卷将 secret 挂载到同一个共享目录中。

用户名和密码值可以是任何经过 **base64** 编码的有效字符串。

以下示例显示 **admin**（base64 编码）：

```
$ echo -n "admin" | base64
```

输出示例

```
YWRtaW4=
```

以下示例显示了 base64 中的 **1f2d1e2e67df** 密码：

```
$ echo -n "1f2d1e2e67df" | base64
```

输出示例

```
MWYyZDFlMmU2N2Rm
```

流程

使用投射卷挂载现有的 secret 卷源。

1. 创建 secret：

   a. 创建一个类似如下的 YAML 文件，根据需要替换密码和用户信息：

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: mysecret
   type: Opaque
   data:
     pass: MWYyZDFlMmU2N2Rm
     user: YWRtaW4=
   ```

   b. 使用以下命令来创建 secret：

   ```
   $ oc create -f <secrets-filename>
   ```

   例如：

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

输出示例

```
secret "mysecret" created
```

c. 您可以使用以下命令来检查是否创建了 secret：

```
$ oc get secret <secret-name>
```

例如：

```
$ oc get secret mysecret
```

输出示例

```
NAME       TYPE      DATA      AGE
mysecret   Opaque    2         17h
```

```
$ oc get secret <secret-name> -o yaml
```

例如：

```
$ oc get secret mysecret -o yaml
```

```
apiVersion: v1
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-05-30T20:21:38Z
  name: mysecret
  namespace: default
  resourceVersion: "2107"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 959e0424-4575-11e7-9f97-fa163e4bd54c
type: Opaque
```

2. 使用投射卷创建 pod。

a. 创建一个类似如下的 YAML 文件，包括 **volumes** 部分：

```
kind: Pod
metadata:
  name: test-projected-volume
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
```

```
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret (1)
```

(1) 您创建的 secret 的名称。

b. 从配置文件创建 pod：

```
$ oc create -f <your_yaml_file>.yaml
```

例如：

```
$ oc create -f secret-pod.yaml
```

输出示例

```
pod "test-projected-volume" created
```

3. 验证 pod 容器是否在运行，然后留意 pod 的更改：

```
$ oc get pod <name>
```

例如：

```
$ oc get pod test-projected-volume
```

输出结果应该类似以下示例：

输出示例

```
NAME                    READY     STATUS    RESTARTS   AGE
test-projected-volume   1/1       Running   0          14s
```

4. 在另一个终端中，使用 **oc exec** 命令来打开连接到运行中容器的 shell：

```
$ oc exec -it <pod> <command>
```

例如：

```
$ oc exec -it test-projected-volume -- /bin/sh
```

5. 在 shell 中，验证 **projected-volumes** 目录是否包含您的投射源：

```
/ # ls
```

输出示例

```
bin              home             root             tmp
dev              proc             run              usr
etc              projected-volume  sys              var
```

7.5. 允许容器消耗 API 对象

Downward API 是一种允许容器消耗 API 对象的相关信息且不与 OpenShift Container Platform 耦合的机制。此类信息包括 pod 的名称、命名空间和资源值。容器可以使用环境变量或卷插件来消耗来自 Downward API 的信息。

7.5.1. 使用 Downward API 向容器公开 Pod 信息

Downward API 包含 pod 的名称、项目和资源值等信息。容器可以使用环境变量或卷插件来消耗来自 Downward API 的信息。

pod 中的字段通过 **FieldRef** API 类型来选择。**FieldRef** 有两个字段：

字段	描述
fieldPath	要选择的字段的路径，这相对于 pod。
apiVersion	要在其中解释 **fieldPath** 选择器的 API 版本。

目前，v1 API 中的有效选择器包括：

选择器	描述
metadata.name	pod 的名称。在环境变量和卷中均受支持。
metadata.namespace	pod 的命名空间。在环境变量和卷中均受支持。
metadata.labels	pod 的标签。仅在卷中支持，环境变量中不支持。
metadata.annotations	pod 的注解。仅在卷中支持，环境变量中不支持。
status.podIP	pod 的 IP。仅在环境变量中支持，卷中不支持。

若未指定 **apiVersion** 字段，则默认为所属 pod 模板的 API 版本。

7.5.2. 了解如何通过 Downward API 消耗容器值

容器可以使用环境变量或卷插件来消耗 API 值。根据您选择的方法，容器可以消耗：

- Pod 名称
- Pod 项目/命名空间
- Pod 注解
- Pod 标签

注解和标签只能通过卷插件来使用。

7.5.2.1. 使用环境变量消耗容器值

在使用容器的环境变量时，请使用 **EnvVar** 类型的 **valueFrom** 字段（类型为 **EnvVarSource**）来指定变量的值应来自 **FieldRef** 源，而非 **value** 字段指定的字面值。

只有 pod 常量属性可以这种方式消耗，因为一旦进程启动并且将变量值已更改的通知发送给进程，就无法更新环境变量。使用环境变量支持的字段包括：

- Pod 名称
- Pod 项目/命名空间

流程

1. 创建一个新的 pod spec，其中包含您希望容器使用的环境变量：
   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
      securityContext:
        allowPrivilegeEscalation: false
```

```
      capabilities:
        drop: [ALL]
  restartPolicy: Never
# ...
```

b. 从 **pod.yaml** 文件创建 pod：

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

验证

- 检查容器的日志，以查看 **MY_POD_NAME** 和 **MY_POD_NAMESPACE** 值：

```
$ oc logs -p dapi-env-test-pod
```

7.5.2.2. 使用卷插件消耗容器值

容器可以使用卷插件来消耗 API 值。

容器可以消耗：

- Pod 名称
- Pod 项目/命名空间
- Pod 注解
- Pod 标签

流程

使用卷插件：

1. 创建一个新的 pod spec，其中包含您希望容器使用的环境变量：
   a. 创建一个类似如下的 **volume-pod.yaml** 文件：

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: "345"
    annotation2: "456"
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: volume-test-container
```

```
    image: gcr.io/google_containers/busybox
    command: ["sh", "-c", "cat /tmp/etc/pod_labels /tmp/etc/pod_annotations"]
    volumeMounts:
      - name: podinfo
        mountPath: /tmp/etc
        readOnly: false
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  volumes:
  - name: podinfo
    downwardAPI:
      defaultMode: 420
      items:
      - fieldRef:
          fieldPath: metadata.name
        path: pod_name
      - fieldRef:
          fieldPath: metadata.namespace
        path: pod_namespace
      - fieldRef:
          fieldPath: metadata.labels
        path: pod_labels
      - fieldRef:
          fieldPath: metadata.annotations
        path: pod_annotations
  restartPolicy: Never
# ...
```

b. 从 **volume-pod.yaml** 文件创建 pod：

```
$ oc create -f volume-pod.yaml
```

验证

- 检查容器的日志，并验证配置的字段是否存在：

```
$ oc logs -p dapi-volume-test-pod
```

输出示例

```
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

7.5.3. 了解如何使用 Downward API 消耗容器资源

在创建 pod 时，您可以使用 Downward API 注入关于计算资源请求和限制的信息，以便镜像和应用程序作者能够正确地为特定环境创建镜像。

您可以使用环境变量或卷插件进行此操作。

7.5.3.1. 使用环境变量消耗容器资源

在创建 pod 时，您可以利用环境变量来使用 Downward API 注入有关计算资源请求和限制的信息。

在创建 pod 配置时，在 **spec.container** 字段中指定与 **resources** 字段的内容对应的环境变量。

注意

如果容器配置中没有包含资源限制，Downward API 会默认使用节点的 CPU 和内存可分配量。

流程

1. 创建一个新的 pod 规格，其中包含您要注入的资源：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.memory
# ...
```

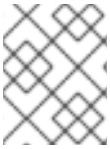
   b. 从 **pod.yaml** 文件创建 pod：

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

7.5.3.2. 使用卷插件消耗容器资源

在创建 pod 时，您可以利用卷插件来使用 Downward API 注入有关计算资源请求和限制的信息。

在创建 pod 配置时，使用 **spec.volumes.downwardAPI.items** 字段来描述与 **spec.resources** 字段对应的所需资源：



注意

如果容器配置中没有包含资源限制，Downward API 会默认使用节点的 CPU 和内存可分配量。

流程

1. 创建一个新的 pod 规格，其中包含您要注入的资源：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c", "while true; do echo; if [[ -e /etc/cpu_limit ]]; then cat
/etc/cpu_limit; fi; if [[ -e /etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e
/etc/mem_limit ]]; then cat /etc/mem_limit; fi; if [[ -e /etc/mem_request ]]; then cat
/etc/mem_request; fi; sleep 5; done"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
```

```
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
# ...
```

b. 从 ***volume-pod.yaml*** 文件创建 pod：

```
$ oc create -f volume-pod.yaml
```

7.5.4. 使用 Downward API 消耗 secret

在创建 pod 时，您可以使用 Downward API 注入 Secret，以便镜像和应用程序作者能够为特定环境创建镜像。

流程

1. 创建要注入的 secret：

   a. 创建一个类似如下的 **secret.yaml** 文件：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  password: <password>
  username: <username>
type: kubernetes.io/basic-auth
```

   b. 从 **secret.yaml** 文件创建 secret 对象：

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

2. 创建引用上述 **Secret** 对象中的 **username** 字段的 pod：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
```

```
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  restartPolicy: Never
# ...
```

b. 从 **pod.yaml** 文件创建 pod：

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

验证

- 检查容器日志中的 **MY_SECRET_USERNAME** 值：

```
$ oc logs -p dapi-env-test-pod
```

7.5.5. 使用 Downward API 消耗配置映射

在创建 pod 时，您可以使用 Downward API 注入配置映射值，以便镜像和应用程序作者能够为特定环境创建镜像。

流程

1. 使用要注入的值创建配置映射：

   a. 创建类似如下的 ***configmap.yaml*** 文件：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

   b. 从 **configmap.yaml** 文件创建配置映射：

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

2. 创建引用上述配置映射的 pod：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
```

```
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_CONFIGMAP_VALUE
          valueFrom:
            configMapKeyRef:
              name: myconfigmap
              key: mykey
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  restartPolicy: Always
# ...
```

b. 从 **pod.yaml** 文件创建 pod：

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

验证

- 检查容器日志中的 **MY_CONFIGMAP_VALUE** 值：

```
$ oc logs -p dapi-env-test-pod
```

7.5.6. 引用环境变量

在创建 pod 时，您可以使用 **$()** 语法引用之前定义的环境变量的值。如果无法解析环境变量引用，则该值将保留为提供的字符串。

流程

1. 创建引用现有环境变量的 pod：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: env-test-container
```

```
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_EXISTING_ENV
          value: my_value
        - name: MY_ENV_VAR_REF_ENV
          value: $(MY_EXISTING_ENV)
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  restartPolicy: Never
# ...
```

b. 从 ***pod.yaml*** 文件创建 pod：

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

验证

- 检查容器日志中的 **MY_ENV_VAR_REF_ENV** 值：

```
$ oc logs -p dapi-env-test-pod
```

7.5.7. 转义环境变量引用

在创建 pod 时，您可以使用双美元符号来转义环境变量引用。然后，其值将设为所提供值的单美元符号版本。

流程

1. 创建引用现有环境变量的 pod：

   a. 创建类似以下示例的 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_NEW_ENV
          value: $$(SOME_OTHER_ENV)
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
```

```
        drop: [ALL]
  restartPolicy: Never
# ...
```

b. 从 ***pod.yaml*** 文件创建 pod：

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

验证

- 检查容器日志中的 **MY_NEW_ENV** 值：

```
$ oc logs -p dapi-env-test-pod
```

7.6. 将文件复制到 OPENSHIFT CONTAINER PLATFORM 容器或从中复制

您可以使用 **rsync** 命令，通过 CLI 将本地文件复制到容器中的远程目录，或从中复制文件。

7.6.1. 了解如何复制文件

oc rsync 命令（或远程同步）是一个实用的工具，能够将数据库存档复制到 pod 中或从 pod 中复制，以满足备份和恢复的需要。当运行的 pod 支持源文件热重载时，您还可以使用 **oc rsync** 将源代码更改复制到运行的 pod，从而进行开发调试。

```
$ oc rsync <source> <destination> [-c <container>]
```

7.6.1.1. 要求

指定复制来源

oc rsync 命令的 source 参数必须指向本地目录或 pod 目录。不支持单个文件。
指定 pod 目录时，目录名称必须加上 pod 名称前缀：

```
<pod name>:<dir>
```

如果目录名以路径分隔符 (/) 结尾，则只有目录的内容会复制到目的地。否则，目录及其内容都会复制到目的地。

指定复制目的地

oc rsync 命令的 destination 参数必须指向某个目录。如果该目录不存在，但使用 **rsync** 进行复制，系统会为您创建这个目录。

删除目的地上的文件

可以使用 **--delete** 标志，在远程目录中删除本地目录中没有的文件。

在文件更改时持续同步

如果使用 **--watch** 选项，命令可以监控源路径上的任何文件系统更改，并在发生更改时同步它们。使用这个参数时，命令会永久运行。
同步会在短暂的静默期后进行，以确保迅速更改的文件系统不会导致持续的同步调用。

使用 **--watch** 选项时，其行为实际上和手动反复调用 **oc rsync** 一致，通常传递给 **oc rsync** 的所有参数也一样。因此，您可以使用与手动调用 **oc rsync** 时相同的标记来控制其行为，比如 **--delete**。

7.6.2. 将文件复制到容器或从容器中复制

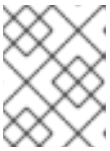
CLI 中内置了将本地文件复制到容器或从容器中复制文件的支持。

先决条件

在使用 **oc rsync** 时，请注意以下几点：

- 必须安装 rsync。**oc rsync** 命令将使用本地的 **rsync** 工具（如果存在于客户端机器和远程容器上）。
  如果本地或远程容器上找不到 **rsync**，则会在本地创建 **tar** 存档并发送到容器（在那里使用 **tar** 实用程序来解压文件）。如果远程容器中没有 **tar**，则复制会失败。

  tar 复制方法不提供与 **oc rsync** 相同的功能。例如，**oc rsync** 会在目的地目录不存在时创建这个目录，而且仅发送来源与目的地上不同的文件。



注意

在 Windows 中，应当安装 **cwRsync** 客户端并添加到 PATH 中，以便与 **oc rsync** 命令搭配使用。

流程

- 将本地目录复制到 pod 目录：

```
$ oc rsync <local-dir> <pod-name>:/<remote-dir> -c <container-name>
```

  例如：

```
$ oc rsync /home/user/source devpod1234:/src -c user-container
```

- 将 pod 目录复制到本地目录：

```
$ oc rsync devpod1234:/src /home/user/source
```

  输出示例

```
$ oc rsync devpod1234:/src/status.txt /home/user/
```

7.6.3. 使用高级 rsync 功能

与标准的 **rsync** 相比，**oc rsync** 命令可用的命令行选项比较少。如果您想要使用某个标准 **rsync** 命令行选项，但 **oc rsync** 中没有这个选项（例如，**--exclude-from=FILE** 选项），您可以使用标准 **rsync** 的 **--rsh** (**-e**) 选项或 **RSYNC_RSH** 变量来作为权宜之计，如下所示：

```
$ rsync --rsh='oc rsh' --exclude-from=<file_name> <local-dir> <pod-name>:/<remote-dir>
```

或：

导出 **RSYNC_RSH** 变量：

```
$ export RSYNC_RSH='oc rsh'
```

然后运行 rsync 命令：

```
$ rsync --exclude-from=<file_name> <local-dir> <pod-name>:/<remote-dir>
```

以上两个示例将标准 **rsync** 配置为使用 **oc rsh** 作为远程 shell 程序，从而连接到远程 pod，它们是运行 **oc rsync** 的替代方法。

7.7. 在 OPENSHIFT CONTAINER PLATFORM 容器中执行远程命令

您可以使用 CLI 在 OpenShift Container Platform 容器中执行远程命令。

7.7.1. 在容器中执行远程命令

CLI 中内置了对执行远程容器命令的支持。

流程

在容器中运行命令：

```
$ oc exec <pod> [-c <container>] -- <command> [<arg_1> ... <arg_n>]
```

例如：

```
$ oc exec mypod date
```

输出示例

```
Thu Apr  9 02:21:53 UTC 2015
```

重要

为了安全起见，**oc exec** 命令在访问特权容器时无法工作，除非该命令由 **cluster-admin** 用户执行。

7.7.2. 用于从客户端发起远程命令的协议

客户端通过向 Kubernetes API 服务器发出请求，来发起在容器中执行远程命令的操作：

```
/proxy/nodes/<node_name>/exec/<namespace>/<pod>/<container>?command=<command>
```

在以上 URL 中：

- **<node_name>** 是节点的 FQDN。
- **<namespace>** 是目标 pod 的项目。
- **<pod>** 是目标 pod 的名称。
- **<container>** 是目标容器的名称。
- **<command>** 是要执行的命令。

例如：

```
/proxy/nodes/node123.openshift.com/exec/myns/mypod/mycontainer?command=date
```

另外，客户端也可以在请求中添加参数来指示是否有以下要求：

- 客户端应向远程容器的命令发送输入 (stdin)。
- 客户端的终端是 TTY。
- 远程容器的命令应该将来自 stdout 的输出发送到客户端。
- 远程容器的命令应该将来自 stderr 的输出发送到客户端。

在向 API 服务器发送 **exec** 请求后，客户端会将连接升级到支持多路复用的流；当前使用 **HTTP/2**。

客户端为 stdin、stdout 和 stderr 分别创建一个流。为了区分流，客户端将流的 **streamType** 标头设置为 **stdin**、**stdout** 或 **stderr** 之一。

在完成远程命令执行请求后，客户端关闭所有流、升级的连接和底层连接。

7.8. 使用端口转发访问容器中的应用程序

OpenShift Container Platform 支持向 pod 转发端口。

7.8.1. 了解端口转发

您可以使用 CLI 将一个或多个本地端口转发到 pod。这样，您可以在本地侦听一个指定或随机端口，并且与 pod 中的指定端口来回转发数据。

CLI 中内置了端口转发支持：

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

CLI 侦听用户指定的本地端口，并通过以下协议进行转发。

可使用以下格式来指定端口：

5000	客户端在本地侦听端口 5000，并转发到 pod 中的 5000。
6000:5000	客户端在本地侦听端口 6000，并转发到 pod 中的 5000。
:5000 或 **0:5000**	客户端选择本地的一个空闲端口，并转发到 pod 中的 5000 。

OpenShift Container Platform 处理来自客户端的端口转发请求。在收到请求后，OpenShift Container Platform 会升级响应并等待客户端创建端口转发流。当 OpenShift Container Platform 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。支持的 OpenShift Container Platform 实施会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。不过，自定义实施中可能会包括运行一个 *helper* pod，然后运行 **nsenter** 和 **socat**，从而不需要在主机上安装这些二进制代码。

7.8.2. 使用端口转发

您可以使用 CLI 将一个或多个本地端口转发到 pod。

流程

使用以下命令侦听 pod 中的指定端口：

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

例如：

- 使用以下命令，侦听本地的 **5000** 和 **6000** 端口，并与 pod 中的 **5000** 和 **6000** 端口来回转发数据：

  ```
  $ oc port-forward <pod> 5000 6000
  ```

  输出示例

  ```
  Forwarding from 127.0.0.1:5000 -> 5000
  Forwarding from [::1]:5000 -> 5000
  Forwarding from 127.0.0.1:6000 -> 6000
  Forwarding from [::1]:6000 -> 6000
  ```

- 使用以下命令，侦听本地的 **8888** 端口并转发到 pod 中的 **5000**：

  ```
  $ oc port-forward <pod> 8888:5000
  ```

  输出示例

  ```
  Forwarding from 127.0.0.1:8888 -> 5000
  Forwarding from [::1]:8888 -> 5000
  ```

- 使用以下命令，侦听本地的一个空闲端口并转发到 pod 中的 **5000**：

  ```
  $ oc port-forward <pod> :5000
  ```

  输出示例

  ```
  Forwarding from 127.0.0.1:42390 -> 5000
  Forwarding from [::1]:42390 -> 5000
  ```

  或者：

  ```
  $ oc port-forward <pod> 0:5000
  ```

7.8.3. 用于从客户端发起端口转发的协议

客户端通过向 Kubernetes API 服务器发出请求，来发起向 pod 转发端口的操作：

```
/proxy/nodes/<node_name>/portForward/<namespace>/<pod>
```

在以上 URL 中：

- **<node_name>** 是节点的 FQDN。
- **<namespace>** 是目标 pod 的命名空间。
- **<pod>** 是目标 pod 的名称。

例如：

```
/proxy/nodes/node123.openshift.com/portForward/myns/mypod
```

向 API 服务器发送端口转发请求后，客户端会将连接升级到支持多路复用流；当前使用 Hyptertext Transfer Protocol Version 2 (HTTP/2)。

客户端创建 **port** 标头中包含 pod 中目标端口的流。写入流的所有数据都通过 Kubelet 传送到目标 pod 和端口。同样，针对被转发连接从 pod 发送的所有数据都会被传回客户端上的同一流。

在完成端口转发请求后，客户端关闭所有流、升级的连接和底层连接。

7.9. 在容器中使用 SYSCTL

sysctl 设置通过 Kubernetes 公开，允许用户在运行时修改某些内核参数。只有拥有命名空间的 sysctl 才能独立于 pod 进行设置。如果 sysctl 没有命名空间（称为 *节点级别*），则必须使用其他方法设置 sysctl，如使用 Node Tuning Operator。

网络 sysctl 是特殊的 sysctl 类别。网络 sysctl 包括：

- 系统范围的 sysctl，如 **net.ipv4.ip_local_port_range**，适用于所有网络。您可以针对节点上的每个 pod 独立设置它们。
- 特定于接口的 sysctl，如 **net.ipv4.conf.IFNAME.accept_local**，它们只适用于给定 pod 的特定额外网络接口。您可以为每个额外网络配置独立设置它们。您可以在网络接口创建后使用 **tuning-cni** 中的配置来设置它们。

此外，只有被认为是 *安全* 的 sysctl 才会默认列在白名单中；您可以在节点上手动启用其他 *不安全* sysctl 来供用户使用。

如果要设置 sysctl 而不是节点级别，您可以在使用 Node Tuning Operator 一节中找到有关此步骤的信息。

7.9.1. 关于 sysctl

在 Linux 中，管理员可通过 sysctl 接口在运行时修改内核参数。参数位于 ***/proc/sys/*** 虚拟进程文件系统中。这些参数涵盖了各种不同的子系统，例如：

- 内核（通用前缀：***kernel.***）
- 网络（通用前缀：***net.***）
- 虚拟内存（通用前缀：***vm.***）
- MDADM（通用前缀：***dev.***）

如需了解更多子系统，请参阅 Kernel 文档。要获取所有参数的列表，请运行：

```
$ sudo sysctl -a
```

7.9.2. 命名空间和节点级 sysctl

许多 sysctl 在 Linux 内核中是有*命名空间*的。这意味着您可以针对节点上的每个 pod 单独设置它们。sysctl 必须拥有命名空间，才能在 Kubernetes 内的 pod 上下文中访问它们。

以下 sysctl 已知是拥有命名空间的：

- ***kernel.shm****
- ***kernel.msg****
- ***kernel.sem***
- ***fs.mqueue.****

另外，**net.*** 组中的大多数 sysctl 都是拥有命名空间的。其命名空间的采用根据内核版本和发行方而有所不同。

无命名空间的 sysctl 被视为*节点级别*，且必须由集群管理员手动设置，或者通过使用节点的底层 Linux 发行版，如修改 ***/etc/sysctls.conf*** 文件，或者通过使用带有特权容器的守护进程集。您可以使用 Node Tuning Operator 来设置*节点级别*的 sysctl。

注意

可以考虑将带有特殊 sysctl 节点标记为污点。仅将 pod 调度到需要这些 sysctl 设置的节点。使用污点和容限功能来标记节点。

7.9.3. 安全和不安全 sysctl

sysctl 划分为*安全*和*不安全* sysctl。

要使系统范围 sysctl 被视为安全，必须具有命名空间。命名空间的 sysctl 可确保命名空间之间有隔离，因此 pod 会被隔离。如果为一个 pod 设置 sysctl，则不能添加以下任一 pod：

- 影响节点上的其他任何 pod
- 危害节点健康状况
- 获取超过 pod 资源限制的 CPU 或内存资源

注意

仅拥有命名空间还不足以使 sysctl 被视为安全。

任何未添加到 OpenShift Container Platform 上允许列表中的 sysctl 都被视为对 OpenShift Container Platform 而言是不安全的。

默认不允许不安全 sysctl。对于系统范围的 sysctl，集群管理员必须基于每个节点手动启用它们。禁用了不安全 sysctl 的 Pod 会被调度，但不会启动。

注意

您不能手动启用特定于接口的不安全 sysctl。

OpenShift Container Platform 将以下系统范围和特定于接口的安全 sysctl 添加到允许的安全列表中：

表 7.4. 系统范围安全 sysctl

sysctl	描述
kernel.shm_rmid_forced	当设置为 **1** 时，当前 IPC 命名空间中的所有共享内存对象会自动强制使用 IPC_RMID。如需更多信息，请参阅 shm_rmid_forced。
net.ipv4.ip_local_port_range	定义 TCP 和 UDP 用于选择本地端口的本地端口范围。第一个数字是第一个端口号，第二个数字是最后一个本地端口号。如果可能，如果这些数字有不同的奇偶校验（甚至一个单数值）。它们必须大于或等于 **ip_unprivileged_port_start**。默认值为 **32768** 和 **60999**。如需更多信息，请参阅 ip_local_port_range。
net.ipv4.tcp_syncookies	当设置了 **net.ipv4.tcp_syncookies** 时，内核通常会处理 TCP SYN 数据包，直至半开连接队列已满（此时SY cookie 功能启动）。这个功能允许系统接受有效连接，即使受拒绝服务攻击也是如此。如需更多信息，请参阅 tcp_syncookies。
net.ipv4.ping_group_range	这将限制 **ICMP_PROTO** 数据报套接字到组范围内的用户。默认值为 **1 0**，代表没有用户可以创建 ping 套接字（即使是 root 也不行）。有关更多信息，请参阅 ping_group_range。
net.ipv4.ip_unprivileged_port_start	这将定义网络命名空间中的第一个无特权端口。要禁用所有特权端口，将其设置为 **0**。特权端口不得与 **ip_local_port_range** 重叠。有关更多信息，请参阅 ip_unprivileged_port_start。

表 7.5. 特定于接口的安全 sysctl

sysctl	描述
net.ipv4.conf.IFNAME.accept_redirects	接受 IPv4 ICMP 重定向消息。
net.ipv4.conf.IFNAME.accept_source_route	接受带有严格的源路由(SRR)选项的 IPv4 数据包。
net.ipv4.conf.IFNAME.arp_accept	使用没有出现在 ARP 表中的 IPv4 地址定义抓取 ARP 帧的行为： • **0** - 不在 ARP 表中创建新条目。 • **1** - 在 ARP 表中创建新条目。
net.ipv4.conf.IFNAME.arp_notify	定义 IPv4 地址和设备更改通知的模式。

sysctl	描述
net.ipv4.conf.IFNAME.disable_policy	禁用这个 IPv4 接口的 IPSEC 策略(SPD)。
net.ipv4.conf.IFNAME.secure_redirects	接受 ICMP 将消息重定向到接口当前网关列表中列出的网关。
net.ipv4.conf.IFNAME.send_redirects	只有当节点充当路由器时，才会启用发送重定向。也就是说，主机不应发送 ICMP 重定向消息。路由器使用它来通知主机与特定目的地可用的更好路由路径。
net.ipv6.conf.IFNAME.accept_ra	接受 IPv6 路由器公告；使用它们自动配置。它还决定是否传输路由器请求。只有功能设置要接受路由器播发时，才会传输路由器请求。
net.ipv6.conf.IFNAME.accept_redirects	接受 IPv6 ICMP 重定向消息。
net.ipv6.conf.IFNAME.accept_source_route	接受带有 SRR 选项的 IPv6 数据包。
net.ipv6.conf.IFNAME.arp_accept	使用没有出现在 ARP 表中的 IPv6 地址定义抓取 ARP 帧的行为： ● **0** - 不在 ARP 表中创建新条目。 ● **1** - 在 ARP 表中创建新条目。
net.ipv6.conf.IFNAME.arp_notify	定义 IPv6 地址和设备更改通知的模式。
net.ipv6.neigh.IFNAME.base_reachable_time_ms	这个参数控制 IPv6 的邻居表中的硬件地址到 IP 映射生命周期。
net.ipv6.neigh.IFNAME.retrans_time_ms	为邻居发现消息设置重新传输计时器。

注意

当使用 **tuning** CNI 插件设置这些值时，请按字面使用值 **IFNAME**。接口名称由 **IFNAME** 令牌表示，并替换为在运行时接口的实际名称。

7.9.4. 更新特定于接口的安全 sysctl 列表

OpenShift Container Platform 包含预定义的安全接口 **sysctl** 列表。您可以通过更新 **openshift-multus** 命名空间中的 **cni-sysctl-allowlist** 来修改此列表。

重要

更新特定于接口的安全 sysctl 列表的支持只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐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，请参阅技术预览功能支持范围。

按照以下步骤修改预定义的安全 **sysctl** 列表。这个步骤描述了如何扩展默认允许列表。

流程

1. 运行以下命令来查看现有的预定义列表：

```
$ oc get cm -n openshift-multus cni-sysctl-allowlist -oyaml
```

预期输出

```
apiVersion: v1
data:
  allowlist.conf: |-
    ^net.ipv4.conf.IFNAME.accept_redirects$
    ^net.ipv4.conf.IFNAME.accept_source_route$
    ^net.ipv4.conf.IFNAME.arp_accept$
    ^net.ipv4.conf.IFNAME.arp_notify$
    ^net.ipv4.conf.IFNAME.disable_policy$
    ^net.ipv4.conf.IFNAME.secure_redirects$
    ^net.ipv4.conf.IFNAME.send_redirects$
    ^net.ipv6.conf.IFNAME.accept_ra$
    ^net.ipv6.conf.IFNAME.accept_redirects$
    ^net.ipv6.conf.IFNAME.accept_source_route$
    ^net.ipv6.conf.IFNAME.arp_accept$
    ^net.ipv6.conf.IFNAME.arp_notify$
    ^net.ipv6.neigh.IFNAME.base_reachable_time_ms$
    ^net.ipv6.neigh.IFNAME.retrans_time_ms$
kind: ConfigMap
metadata:
  annotations:
    kubernetes.io/description: |
      Sysctl allowlist for nodes.
    release.openshift.io/version: 4.16.0-0.nightly-2022-11-16-003434
  creationTimestamp: "2022-11-17T14:09:27Z"
  name: cni-sysctl-allowlist
  namespace: openshift-multus
  resourceVersion: "2422"
  uid: 96d138a3-160e-4943-90ff-6108fa7c50c3
```

2. 使用以下命令编辑列表：

```
$ oc edit cm -n openshift-multus cni-sysctl-allowlist -oyaml
```

例如，要允许您实现更严格的反向路径转发，您需要将 **^net.ipv4.conf.IFNAME.rp_filter$** 和 **^net.ipv6.conf.IFNAME.rp_filter$** 添加到列表中，如下所示：

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will be
# reopened with the relevant failures.
#
apiVersion: v1
data:
  allowlist.conf: |-
    ^net.ipv4.conf.IFNAME.accept_redirects$
    ^net.ipv4.conf.IFNAME.accept_source_route$
    ^net.ipv4.conf.IFNAME.arp_accept$
    ^net.ipv4.conf.IFNAME.arp_notify$
    ^net.ipv4.conf.IFNAME.disable_policy$
    ^net.ipv4.conf.IFNAME.secure_redirects$
    ^net.ipv4.conf.IFNAME.send_redirects$
    ^net.ipv4.conf.IFNAME.rp_filter$
    ^net.ipv6.conf.IFNAME.accept_ra$
    ^net.ipv6.conf.IFNAME.accept_redirects$
    ^net.ipv6.conf.IFNAME.accept_source_route$
    ^net.ipv6.conf.IFNAME.arp_accept$
    ^net.ipv6.conf.IFNAME.arp_notify$
    ^net.ipv6.neigh.IFNAME.base_reachable_time_ms$
    ^net.ipv6.neigh.IFNAME.retrans_time_ms$
    ^net.ipv6.conf.IFNAME.rp_filter$
```

3. 保存对文件的更改并退出。

注意

也支持删除 **sysctl**。编辑该文件，删除 **sysctl** 或 **sysctl**，然后保存更改并退出。

验证

按照以下步骤为 IPv4 强制更严格的反向路径转发。有关反向路径转发的更多信息，请参阅反向路径转发。

1. 使用以下内容创建网络附加定义，如 **reverse-path-fwd-example.yaml** ：

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: tuningnad
  namespace: default
spec:
  config: '{
    "cniVersion": "0.4.0",
    "name": "tuningnad",
    "plugins": [{
      "type": "bridge"
    },
    {
    "type": "tuning",
    "sysctl": {
       "net.ipv4.conf.IFNAME.rp_filter": "1"
     }
```

```
    }
  ]
}'
```

2. 运行以下命令来应用 yaml:

```
$ oc apply -f reverse-path-fwd-example.yaml
```

输出示例

```
networkattachmentdefinition.k8.cni.cncf.io/tuningnad created
```

3. 使用以下 YAML 创建 **pod.yaml** 等 pod：

```
apiVersion: v1
kind: Pod
metadata:
  name: example
  labels:
    app: httpd
  namespace: default
  annotations:
    k8s.v1.cni.cncf.io/networks: tuningnad (1)
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: httpd
      image: 'image-registry.openshift-image-registry.svc:5000/openshift/httpd:latest'
      ports:
        - containerPort: 8080
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
```

(1) 指定配置的 **NetworkAttachmentDefinition** 的名称。

4. 运行以下命令来应用 yaml:

```
$ oc apply -f examplepod.yaml
```

5. 运行以下命令验证 pod 是否已创建：

```
$ oc get pod
```

输出示例

```
NAME     READY   STATUS    RESTARTS   AGE
example  1/1     Running   0          47s
```

6. 运行以下命令登录到 pod：

```
$ oc rsh example
```

7. 验证配置的 sysctl 标记的值。例如，通过运行以下命令查找 **net.ipv4.conf.net1.rp_filter** 的值：

```
sh-4.4# sysctl net.ipv4.conf.net1.rp_filter
```

预期输出

```
net.ipv4.conf.net1.rp_filter = 1
```

其他资源

- Linux 网络文档

7.9.5. 使用安全 sysctl 启动 pod

您可以使用 pod 的 **securityContext** 在 pod 上设置 sysctl。 **securityContext** 适用于同一 pod 中的所有容器。

默认允许安全 sysctl。

这个示例使用 pod **securityContext** 来设置以下安全 sysctl：

- **kernel.shm_rmid_forced**
- **net.ipv4.ip_local_port_range**
- **net.ipv4.tcp_syncookies**
- **net.ipv4.ping_group_range**

警告

为了避免让操作系统变得不稳定，只有在了解了参数的作用后才修改 sysctl 参数。

使用此流程启动带有配置的 sysctl 设置的 pod。

注意

在大多数情况下，您修改现有的 pod 定义并添加 **securityContext** 规格。

流程

1. 创建一个定义示例 pod 的 YAML 文件 **sysctl_pod.yaml** 并添加 **securityContext** 规格，如下例所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  namespace: default
spec:
  containers:
  - name: podexample
    image: centos
    command: ["bin/bash", "-c", "sleep INF"]
    securityContext:
      runAsUser: 2000 (1)
      runAsGroup: 3000 (2)
      allowPrivilegeEscalation: false (3)
      capabilities: (4)
        drop: ["ALL"]
  securityContext:
    runAsNonRoot: true (5)
    seccompProfile: (6)
      type: RuntimeDefault
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "1"
    - name: net.ipv4.ip_local_port_range
      value: "32770       60666"
    - name: net.ipv4.tcp_syncookies
      value: "0"
    - name: net.ipv4.ping_group_range
      value: "0           200000000"
```

(1) **runAsUser** 控制使用哪个用户 ID 运行容器。

(2) **runAsGroup** 控制容器使用哪个主要组 ID。

(3) **allowPrivilegeEscalation** 决定 pod 是否请求允许特权升级。如果未指定，则默认为 true。这个布尔值直接控制在容器进程中是否设置了 **no_new_privs** 标志。

(4) **capabilities** 允许特权操作，而不提供完整的 root 访问权限。此策略可确保从 pod 中丢弃了所有功能。

(5) **runAsNonRoot: true** 要求容器使用 0 以外的任何 UID 运行。

(6) **RuntimeDefault** 为 pod 或容器工作负载启用默认的 seccomp 配置集。

2. 运行以下命令来创建 pod：

```
$ oc apply -f sysctl_pod.yaml
```

3. 运行以下命令验证 pod 是否已创建：

```
$ oc get pod
```

输出示例

```
NAME             READY   STATUS          RESTARTS   AGE
sysctl-example   1/1     Running         0          14s
```

4. 运行以下命令登录到 pod：

```
$ oc rsh sysctl-example
```

5. 验证配置的 sysctl 标记的值。例如，通过运行以下命令找到值 **kernel.shm_rmid_forced** ：

```
sh-4.4# sysctl kernel.shm_rmid_forced
```

预期输出

```
kernel.shm_rmid_forced = 1
```

7.9.6. 使用不安全 sysctl 启动 pod

带有不安全 sysctl 的 pod 无法在任何节点上启动，除非集群管理员在某个节点上明确启用了不安全 sysctl。与节点级 sysctl 一样，使用污点和容限功能或节点上的标签将这些 pod 调度到正确的节点。

以下示例使用 pod **securityContext** 设置安全 sysctl **kernel.shm_rmid_forced**，以及两个不安全 sysctl **net.core.somaxconn** 和 **kernel.msgmax**。在规格中，*安全*和*不安全* sysctl 并无区别。

警告

为了避免让操作系统变得不稳定，只有在了解了参数的作用后才修改 sysctl 参数。

以下示例演示了在 pod 规格中添加安全和不安全 sysctl 时会发生什么：

流程

1. 创建一个 YAML 文件 **sysctl-example-unsafe.yaml**，用于定义示例 pod 并添加 **securityContext** 规格，如下例所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example-unsafe
spec:
  containers:
  - name: podexample
    image: centos
    command: ["bin/bash", "-c", "sleep INF"]
    securityContext:
      runAsUser: 2000
      runAsGroup: 3000
```

```
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
```

2. 使用以下命令创建 pod：

```
$ oc apply -f sysctl-example-unsafe.yaml
```

3. 使用以下命令验证 pod 是否调度但没有部署，但没有部署不安全 sysctl：

```
$ oc get pod
```

输出示例

```
NAME                     READY          STATUS          RESTARTS   AGE
sysctl-example-unsafe    0/1              SysctlForbidden   0          14s
```

7.9.7. 启用不安全 sysctl

集群管理员可在非常特殊的情况下允许某些不安全 sysctl，比如高性能或实时应用程序性能优化。

如果要使用不安全 sysctl，集群管理员必须为特定类型的节点单独启用它们。sysctl 必须拥有命名空间。

您可以通过在 Security Context Constraints 的 **allowedUnsafeSysctls** 字段中指定 sysctl 模式列表来进一步控制 pod 中设置哪些 sysctl。

- **allowedUnsafeSysctls** 选项用来控制特定的需求，如高性能或实时应用程序调整。

警告

由于其不安全特性，使用不安全 sysctl 的风险由您自行承担，而且可能会造成严重问题，例如容器行为不当、资源短缺或节点受损。

流程

1. 运行以下命令，列出 OpenShift Container Platform 集群的现有 MachineConfig 对象，以确定如何标记您的机器配置：

```
$ oc get machineconfigpool
```

输出示例

```
NAME     CONFIG                                         UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master   rendered-master-bfb92f0cd1684e54d8e234ab7423cc96   True      False      False
3              3                 3                   0                      42m
worker   rendered-worker-21b6cb9a0f8919c88caf39db80ac1fce   True      False      False
3              3                 3                   0                      42m
```

2. 运行以下命令，在带有不安全 sysctl 的容器的机器配置池中添加标签：

```
$ oc label machineconfigpool worker custom-kubelet=sysctl
```

3. 创建定义 **KubeletConfig** 自定义资源(CR)的 YAML 文件 **set-sysctl-worker.yaml** ：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-kubelet
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: sysctl 1
  kubeletConfig:
    allowedUnsafeSysctls: 2
      - "kernel.msg*"
      - "net.core.somaxconn"
```

1 指定机器配置池中的标签。

2 列出您想要允许的不安全 sysctl。

4. 运行以下命令来创建对象：

```
$ oc apply -f set-sysctl-worker.yaml
```

5. 运行以下命令，等待 Machine Config Operator 生成新呈现的配置并将其应用到机器：

```
$ oc get machineconfigpool worker -w
```

几分钟后 **UPDATING** 状态从 True 变为 False:

```
NAME     CONFIG                                         UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
worker   rendered-worker-f1704a00fc6f30d3a7de9a15fd68a800   False     True       False
3              2                 2                   0                      71m
worker   rendered-worker-f1704a00fc6f30d3a7de9a15fd68a800   False     True       False
```

```
3          2              3              0              72m
worker   rendered-worker-0188658afe1f3a183ec8c4f14186f4d5   True     False     False
3          3              3              0              72m
```

6. 创建一个 YAML 文件 **sysctl-example-safe-unsafe.yaml**，它定义了一个 pod 示例并添加 **securityContext** 规格，如下例所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example-safe-unsafe
spec:
  containers:
  - name: podexample
    image: centos
    command: ["bin/bash", "-c", "sleep INF"]
    securityContext:
      runAsUser: 2000
      runAsGroup: 3000
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
```

7. 运行以下命令来创建 pod：

```
$ oc apply -f sysctl-example-safe-unsafe.yaml
```

预期输出

```
Warning: would violate PodSecurity "restricted:latest": forbidden sysctls
(net.core.somaxconn, kernel.msgmax)
pod/sysctl-example-safe-unsafe created
```

8. 运行以下命令验证 pod 是否已创建：

```
$ oc get pod
```

输出示例

```
NAME                        READY   STATUS    RESTARTS   AGE
sysctl-example-safe-unsafe   1/1     Running   0          19s
```

9. 运行以下命令登录到 pod：

```
$ oc rsh sysctl-example-safe-unsafe
```

10. 验证配置的 sysctl 标记的值。例如，通过运行以下命令查找 **net.core.somaxconn** 的值：

```
sh-4.4# sysctl net.core.somaxconn
```

预期输出

```
net.core.somaxconn = 1024
```

现在，允许不安全的 sysctl，该值在更新的 pod 规格的 **securityContext** 规格中定义。

7.9.8. 其他资源

- 使用 tuning CNI 配置系统控制
- 使用 Node Tuning Operator

7.10. 使用 /DEV/FUSE 访问更快的构建

您可以使用 **/dev/fuse** 设备配置 pod，以访问更快的构建。

7.10.1. 在非特权 pod 上配置 /dev/fuse

作为虚拟文件系统的替代选择，您可以将 **/dev/fuse** 设备配置为 **io.kubernetes.cri-o.Devices** 注解，以便在非特权 Pod 中访问更快的构建。使用 **/dev/fuse** 是安全、高效且可扩展的，并允许非特权用户像非特权 pod 有特权一样挂载覆盖文件系统。

流程

1. 创建 pod。

```
$ oc exec -ti no-priv -- /bin/bash
```

```
$ cat >> Dockerfile <<EOF
FROM registry.access.redhat.com/ubi9
EOF
```

```
$ podman build .
```

2. 通过将 **/dev/fuse** 设备添加到 **io.kubernetes.cri-o.Devices** 注解来实现 **/dev/fuse**。

```
io.kubernetes.cri-o.Devices: "/dev/fuse"
```

例如：

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: podman-pod
  annotations:
    io.kubernetes.cri-o.Devices: "/dev/fuse"
```

3. 在 pod 规格中配置 **/dev/fuse** 设备。

```
spec:
  containers:
  - name: podman-container
    image: quay.io/podman/stable
    args:
    - sleep
    - "1000000"
    securityContext:
      runAsUser: 1000
```

第 8 章 操作集群

8.1. 查看 OPENSHIFT CONTAINER PLATFORM 集群中的系统事件信息

OpenShift Container Platform 中的事件根据 OpenShift Container Platform 集群中 API 对象的事件进行建模。

8.1.1. 了解事件

事件允许 OpenShift Container Platform 以无关资源的方式记录实际事件的信息。它们还允许开发人员和管理员以统一的方式消耗系统组件的信息。

8.1.2. 使用 CLI 查看事件

您可以使用 CLI，获取给定项目中的事件列表。

流程

- 要查看某一项目中的事件，请使用以下命令：

```
$ oc get events [-n <project>] 1
```

1 项目的名称。

例如：

```
$ oc get events -n openshift-config
```

输出示例

```
LAST SEEN   TYPE      REASON              OBJECT                    MESSAGE
97m         Normal    Scheduled           pod/dapi-env-test-pod     Successfully assigned
openshift-config/dapi-env-test-pod to ip-10-0-171-202.ec2.internal
97m         Normal    Pulling             pod/dapi-env-test-pod     pulling image
"gcr.io/google_containers/busybox"
97m         Normal    Pulled              pod/dapi-env-test-pod     Successfully pulled image
"gcr.io/google_containers/busybox"
97m         Normal    Created             pod/dapi-env-test-pod     Created container
9m5s        Warning   FailedCreatePodSandBox   pod/dapi-volume-test-pod   Failed create
pod sandbox: rpc error: code = Unknown desc = failed to create pod network sandbox
k8s_dapi-volume-test-pod_openshift-config_6bc60c1f-452e-11e9-9140-
0eec59c23068_0(748c7a40db3d08c07fb4f9eba774bd5effe5f0d5090a242432a73eee66ba9e22
): Multus: Err adding pod to network "openshift-sdn": cannot set "openshift-sdn" ifname to
"eth0": no netns: failed to Statfs "/proc/33366/ns/net": no such file or directory
8m31s       Normal    Scheduled           pod/dapi-volume-test-pod  Successfully assigned
openshift-config/dapi-volume-test-pod to ip-10-0-171-202.ec2.internal
```

- 从 OpenShift Container Platform 控制台查看项目中的事件。

1. 启动 OpenShift Container Platform 控制台。
2. 点击 **Home → Events**，再选择您的项目。

3. 移到您想要查看事件的资源。例如，**Home** → Project → <project-name> → <resource-name>。
   pod 和部署等许多对象也具有自己的 **Events** 选项卡，其中显示与该对象相关的事件。

8.1.3. 事件列表

本节介绍 OpenShift Container Platform 的事件。

表 8.1. 配置事件

名称	描述
FailedValidation	pod 配置验证失败。

表 8.2. 容器事件

名称	描述
BackOff	避退重启使容器失败。
Created	已创建容器。
Failed	拉取/创建/启动失败。
Killing	正在终止容器。
Started	容器已启动。
Preempting	正在抢占其他 pod。
ExceededGracePeriod	在指定宽限期内，容器运行时没有停止 pod。

表 8.3. 健康事件

名称	描述
Unhealthy	容器不健康。

表 8.4. 镜像事件

名称	描述
BackOff	避退容器启动，镜像拉取。
ErrImageNeverPull	违反了镜像的 **NeverPull 策略**。

名称	描述
Failed	拉取镜像失败。
InspectFailed	检查镜像失败。
Pulled	成功拉取了镜像，或容器镜像已存在于机器上。
Pulling	正在拉取镜像。

表 8.5. 镜像管理器事件

名称	描述
FreeDiskSpaceFailed	可用磁盘空间失败。
InvalidDiskCapacity	磁盘容量无效。

表 8.6. 节点事件

名称	描述
FailedMount	卷挂载已失败。
HostNetworkNotSupported	主机网络不受支持。
HostPortConflict	主机/端口冲突。
KubeletSetupFailed	kubelet 设置失败。
NilShaper	未定义整形器。
NodeNotReady	节点未就绪。
NodeNotSchedulable	节点不可调度。
NodeReady	节点已就绪。
NodeSchedulable	节点可以调度。

名称	描述
NodeSelectorMismatching	节点选择器不匹配。
OutOfDisk	磁盘空间不足。
Rebooted	节点已重启。
Starting	正在启动 kubelet。
FailedAttachVolume	附加卷失败。
FailedDetachVolume	分离卷失败。
VolumeResizeFailed	扩展/缩减卷失败。
VolumeResizeSuccessful	成功扩展/缩减卷。
FileSystemResizeFailed	扩展/缩减文件系统失败。
FileSystemResizeSuccessful	成功扩展/缩减文件系统。
FailedUnMount	卸载卷失败。
FailedMapVolume	映射卷失败。
FailedUnmapDevice	取消映射设备失败。
AlreadyMountedVolume	卷已经挂载。
SuccessfulDetachVolume	卷已被成功分离。
SuccessfulMountVolume	卷已被成功挂载。
SuccessfulUnMountVolume	卷已被成功卸载。

名称	描述
ContainerGCFailed	容器垃圾回收失败。
ImageGCFailed	镜像垃圾回收失败。
FailedNodeAllocatableEnforcement	未能强制实施系统保留的 Cgroup 限制。
NodeAllocatableEnforced	已强制实施系统保留的 Cgroup 限制。
UnsupportedMountOption	不支持的挂载选项。
SandboxChanged	Pod 沙盒已更改。
FailedCreatePodSandBox	未能创建 pod 沙盒。
FailedPodSandBoxStatus	pod 沙盒状态失败。

表 8.7. Pod worker 事件

名称	描述
FailedSync	Pod 同步失败。

表 8.8. 系统事件

名称	描述
SystemOOM	集群遇到 OOM（内存不足）状况。

表 8.9. Pod 事件

名称	描述
FailedKillPod	停止 pod 失败。
FailedCreatePodContainer	创建 pod 容器失败。

名称	描述
Failed	创建 pod 数据目录失败。
NetworkNotReady	网络未就绪。
FailedCreate	创建时出错：**<error-msg>**。
SuccessfulCreate	已创建 pod：**<pod-name>**。
FailedDelete	删除时出错：**<error-msg>**。
SuccessfulDelete	已删除 pod：**<pod-id>**。

表 8.10. Pod 横向自动扩展事件

名称	描述
SelectorRequired	需要选择器。
InvalidSelector	无法将选择器转换为对应的内部选择器对象。
FailedGetObjectMetric	HPA 无法计算副本数。
InvalidMetricSourceType	未知的指标源类型。
ValidMetricFound	HPA 能够成功计算副本数。
FailedConvertHPA	未能转换给定的 HPA。
FailedGetScale	HPA 控制器无法获取目标的当前规模。
SucceededGetScale	HPA 控制器成功获取了目标的当前规模。
FailedComputeMetricsReplicas	未能根据列出的指标计算所需的副本数。
FailedRescale	新大小：**<size>**；原因：**<msg>**：错误：**<error-msg>**。

名称	描述
SuccessfulRescale	新大小：**<size>**；原因：**<msg>**。
FailedUpdateStatus	未能更新状态。

表 8.11. 网络事件(openshift-sdn)

名称	描述
Starting	启动 OpenShift SDN.
NetworkFailed	pod 的网络接口已经丢失，pod 也将被停止。

表 8.12. 网络事件(kube-proxy)

名称	描述
NeedPods	服务端口 **<serviceName>:<port>** 需要 pod。

表 8.13. 卷事件

名称	描述
FailedBinding	没有可用的持久性卷，而且未设置存储类。
VolumeMismatch	卷大小或类与声明中请求的不同。
VolumeFailedRecycle	创建回收 pod 时出错。
VolumeRecycled	回收卷时发生。
RecyclerPod	回收 pod 时发生。
VolumeDelete	删除卷时发生。
VolumeFailedDelete	删除卷时出错。
ExternalProvisioning	在手动或通过外部软件置备声明的卷时发生。

名称	描述
ProvisioningFailed	未能置备卷。
ProvisioningCleanupFailed	清理置备的卷时出错。
ProvisioningSucceeded	在成功置备了卷时发生。
WaitForFirstConsumer	将绑定延迟到 pod 调度为止。

表 8.14. 生命周期 hook

名称	描述
FailedPostStartHook	处理程序因为 pod 启动而失败。
FailedPreStopHook	处理程序因为预停止而失败。
UnfinishedPreStopHook	预停止 hook 未完成。

表 8.15. 部署

名称	描述
DeploymentCancellationFailed	未能取消部署。
DeploymentCancelled	已取消的部署。
DeploymentCreated	已创建新的复制控制器。
IngressIPRangeFull	没有可用的入口 IP 可分配给服务。

表 8.16. 调度程序事件

名称	描述
FailedScheduling	未能调度 pod：**<pod-namespace>/<pod-name>**。引发此事件有多种原因，如 **AssumePodVolumes** 失败或绑定遭拒等。
Preempted	被节点 **<node-name>** 上的 **<preemptor-namespace>/<preemptor-name>** 抢占。
Scheduled	成功将 **<pod-name>** 分配给 **<node-name>**。

表 8.17. 守护进程集事件

名称	描述
SelectingAll	此 daemon 选择所有 pod。需要非空选择器。
FailedPlacement	未能将 pod 放置到 **<node-name>**。
FailedDaemonPod	在节点 **<node-name>** 上找到了失败的守护进程 pod**<pod-name>**，会尝试将它终止。

表 8.18. 负载均衡器服务事件

名称	描述
CreatingLoadBalancerFailed	创建负载均衡器时出错。
DeletingLoadBalancer	正在删除负载均衡器。
EnsuringLoadBalancer	正在确保负载均衡器。
EnsuredLoadBalancer	已确保负载均衡器。
UnAvailableLoadBalancer	没有可用于 **LoadBalancer** 服务的节点。
LoadBalancerSourceRanges	列出新的 **LoadBalancerSourceRanges**。例如，**<old-source-range>** → **<new-source-range>**。
LoadbalancerIP	列出新 IP 地址。例如，**<old-ip>** → **<new-ip>**。

名称	描述
ExternalIP	列出外部 IP 地址。例如，**Added: <external-ip>**。
UID	列出新 UID。例如，**<old-service-uid>** → **<new-service-uid>**。
ExternalTrafficPolicy	列出新 **ExternalTrafficPolicy**。例如，**<old-policy>** → **<new-policy>**。
HealthCheckNodePort	列出新 **HealthCheckNodePort**。例如，**<old-node-port>** → **new-node-port>**。
UpdatedLoadBalancer	使用新主机更新负载均衡器。
LoadBalancerUpdateFailed	使用新主机更新负载均衡器时出错。
DeletingLoadBalancer	正在删除负载均衡器。
DeletingLoadBalancerFailed	删除负载均衡器时出错。
DeletedLoadBalancer	已删除负载均衡器。

8.2. 估算 OPENSHIFT CONTAINER PLATFORM 节点可以容纳的 POD 数量

作为集群管理员，您可以使用 OpenShift Cluster Capacity Tool 查看可以调度的 pod 数量，以便在资源耗尽前增加当前资源，并确保将来的 pod 可以被调度。此容量来自于集群中的节点主机，包括 CPU、内存和磁盘空间等。

8.2.1. 了解 OpenShift Cluster Capacity Tool

OpenShift Cluster Capacity Tool 模拟一系列调度决策，以确定在资源耗尽前集群中可以调度多少个输入 pod 实例，以提供更准确的估算。

注意

因为它不计算节点间分布的所有资源，所以它所显示的剩余可分配容量是粗略估算值。它只分析剩余的资源，并通过估算集群中可以调度多少个具有给定要求的 pod 实例来估测仍可被消耗的可用容量。

另外，根据选择和关联性条件，可能仅支持将 pod 调度到特定的节点集合。因此，可能很难估算集群还能调度多少个 pod。

您可以从命令行运行 OpenShift Cluster Capacity Tool 作为独立实用程序，或者在 OpenShift Container Platform 集群内的 pod 中作为作业运行。作为 pod 中的作业运行该工具，您可以在不干预的情况下多次运行它。

8.2.2. 在命令行中运行 OpenShift Cluster Capacity Tool

您可以从命令行运行 OpenShift Cluster Capacity Tool，以估算可调度到集群中的 pod 数量。

您可以创建一个示例 pod spec 文件，工具使用它来估算资源使用情况。pod 规格将其资源要求指定为 **limits** 或 **requests**。集群容量工具在估算分析时会考虑 pod 的资源要求。

先决条件

1. 运行 OpenShift Cluster Capacity Tool，它可作为来自红帽生态系统目录中的容器镜像。
2. 创建 pod spec 文件示例：
   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: php-redis
    image: gcr.io/google-samples/gb-frontend:v4
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 150m
        memory: 100Mi
      requests:
        cpu: 150m
        memory: 100Mi
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

   b. 创建集群角色：

```
$ oc create -f <file_name>.yaml
```

   例如：

```
$ oc create -f pod-spec.yaml
```

流程

在命令行中使用集群容量工具：

1. 在终端中登录到 Red Hat Registry：

```
$ podman login registry.redhat.io
```

2. 拉取集群容量工具镜像：

```
$ podman pull registry.redhat.io/openshift4/ose-cluster-capacity
```

3. 运行集群容量工具：

```
$ podman run -v $HOME/.kube:/kube:Z -v $(pwd):/cc:Z  ose-cluster-capacity \
/bin/cluster-capacity --kubeconfig /kube/config --<pod_spec>.yaml /cc/<pod_spec>.yaml \
--verbose
```

其中：

<pod_spec>.yaml

指定要使用的 pod 规格。

详细

输出有关集群中每个节点上可以调度多少个 pod 的详细描述。

输出示例

```
small-pod pod requirements:
 - CPU: 150m
 - Memory: 100Mi

The cluster can schedule 88 instance(s) of the pod small-pod.

Termination reason: Unschedulable: 0/5 nodes are available: 2 Insufficient cpu,
3 node(s) had taint {node-role.kubernetes.io/master: }, that the pod didn't
tolerate.

Pod distribution among nodes:
small-pod
 - 192.168.124.214: 45 instance(s)
 - 192.168.124.120: 43 instance(s)
```

在上例中，集群中预计可以调度的 pod 数量为 88。

8.2.3. 将 OpenShift Cluster Capacity Tool 作为 pod 中的作业运行

通过以 pod 中的作业形式运行 OpenShift Cluster Capacity Tool，您可以多次运行该工具，而无需用户干预。您可以使用 **ConfigMap** 对象以作业的形式运行 OpenShift Cluster Capacity Tool。

先决条件

下载并安装 OpenShift Cluster Capacity Tool。

流程

运行集群容量工具：

1. 创建集群角色：

a. 创建一个类似以下示例的 YAML 文件：

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cluster-capacity-role
rules:
- apiGroups: [""]
  resources: ["pods", "nodes", "persistentvolumeclaims", "persistentvolumes", "services",
"replicationcontrollers"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"]
  resources: ["replicasets", "statefulsets"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["policy"]
  resources: ["poddisruptionbudgets"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["storage.k8s.io"]
  resources: ["storageclasses"]
  verbs: ["get", "watch", "list"]
```

b. 运行以下命令来创建集群角色：

```
$ oc create -f <file_name>.yaml
```

例如：

```
$ oc create sa cluster-capacity-sa
```

2. 创建服务帐户：

```
$ oc create sa cluster-capacity-sa -n default
```

3. 将角色添加到服务帐户：

```
$ oc adm policy add-cluster-role-to-user cluster-capacity-role \
    system:serviceaccount:<namespace>:cluster-capacity-sa
```

其中：

<namespace>
指定 pod 所在的命名空间。

4. 定义并创建 pod 规格：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
```

```
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: php-redis
    image: gcr.io/google-samples/gb-frontend:v4
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 150m
        memory: 100Mi
      requests:
        cpu: 150m
        memory: 100Mi
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

b. 运行以下命令来创建 pod：

```
$ oc create -f <file_name>.yaml
```

例如：

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

5. 运行以下命令来创建配置映射对象：

```
$ oc create configmap cluster-capacity-configmap \
    --from-file=pod.yaml=pod.yaml
```

集群容量分析使用名为 **cluster-capacity-configmap** 的配置映射对象挂载到卷中，将输入 pod 规格文件 **pod.yaml** 挂载到卷 **test-volume** 的路径 **/test-pod**。

6. 使用以下作业规格文件示例创建作业：

a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: batch/v1
kind: Job
metadata:
  name: cluster-capacity-job
spec:
  parallelism: 1
  completions: 1
  template:
    metadata:
      name: cluster-capacity-pod
    spec:
       containers:
       - name: cluster-capacity
         image: openshift/origin-cluster-capacity
```

```
      imagePullPolicy: "Always"
      volumeMounts:
      - mountPath: /test-pod
        name: test-volume
      env:
      - name: CC_INCLUSTER 1
        value: "true"
      command:
      - "/bin/sh"
      - "-ec"
      - |
        /bin/cluster-capacity --podspec=/test-pod/pod.yaml --verbose
    restartPolicy: "Never"
    serviceAccountName: cluster-capacity-sa
    volumes:
    - name: test-volume
      configMap:
        name: cluster-capacity-configmap
```

1 必要的环境变量，使集群容量工具知道它将作为一个 pod 在集群中运行。**ConfigMap** 对象的 **pod.yaml** 键与 **Pod** spec 文件名称相同，但这不是必须的。如果这样做，输入 pod 规格文件可作为 **/test-pod/pod.yaml** 在 pod 中被访问。

b. 运行以下命令，以 pod 中作业的形式运行集群容量镜像：

```
$ oc create -f cluster-capacity-job.yaml
```

验证

1. 检查作业日志，以查找在集群中可调度的 pod 数量：

```
$ oc logs jobs/cluster-capacity-job
```

输出示例

```
small-pod pod requirements:
        - CPU: 150m
        - Memory: 100Mi

The cluster can schedule 52 instance(s) of the pod small-pod.

Termination reason: Unschedulable: No nodes are available that match all of the
following predicates:: Insufficient cpu (2).

Pod distribution among nodes:
small-pod
        - 192.168.124.214: 26 instance(s)
        - 192.168.124.120: 26 instance(s)
```

8.3. 使用限制范围限制资源消耗

默认情况下，容器在 OpenShift Container Platform 集群上使用无限的计算资源运行。通过限制范围，您可以限制项目中特定对象的资源消耗：

- pod 和容器：您可以为 pod 及其容器设置 CPU 和内存的最小和最大要求。
- 镜像流：您可以设置 **ImageStream** 对象中的镜像和标签数量的限制。
- 镜像：您可以限制可推送到内部 registry 的镜像大小。
- 持久性卷声明(PVC):您可以限制请求的 PVC 的大小。

如果 pod 未满足限制范围强制的限制，则无法在命名空间中创建 pod。

8.3.1. 关于限制范围

LimitRange 对象定义的限值范围限制项目中的资源消耗。在项目中，您可以为 pod、容器、镜像、镜像流或持久性卷声明（PVC）设置特定资源限值。

要创建和修改资源的所有请求都会针对项目中的每个 **LimitRange** 对象进行评估。如果资源违反了任何限制，则会拒绝该资源。

以下显示了所有组件的限制范围对象： pod、容器、镜像、镜像流或 PVC。您可以在同一对象中为这些组件的一个或多个组件配置限值。您可以为每个要控制资源的项目创建不同的限制范围对象。

容器的限制范围对象示例

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits"
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
      default:
        cpu: "300m"
        memory: "200Mi"
      defaultRequest:
        cpu: "200m"
        memory: "100Mi"
      maxLimitRequestRatio:
        cpu: "10"
```

8.3.1.1. 关于组件限制

以下示例显示每个组件的限制范围参数。为清楚起见，示例已被分隔。您可以根据需要为任何或所有组件创建一个 **LimitRange** 对象。

8.3.1.1.1. 容器限制

通过限制范围，您可以指定 pod 中每个容器可以请求的特定项目的最小和最大 CPU 和内存。如果在项目中创建容器，则 **Pod** spec 中的容器 CPU 和内存请求必须符合 **LimitRange** 对象中设置的值。如果没有，则 pod 不会被创建。

- 对于在 **LimitRange** 对象中指定的容器，容器 CPU 或内存请求和限制必须大于或等于 **min** 资源约束。
- 容器 CPU 或内存请求和限制必须小于或等于 **LimitRange** 对象中指定的容器的 **max** 资源约束。如果 **LimitRange** 对象定义了 **max** CPU，则不需要在 **Pod** spec 中定义 CPU **请求（request）**值。但您必须指定一个 CPU **limit** 值，它需要满足在限制范围中指定的最大 CPU 限值。
- 容器限制与请求的比例必须小于或等于 **LimitRange** 对象中指定的容器的 **maxLimitRequestRatio** 值。
  如果 **LimitRange** 对象定义了 **maxLimitRequestRatio** 约束，则任何新容器都必须同时具有 **request** 和 **limit** 值。OpenShift Container Platform 通过 **limit** 除以 **request** 来计算限制与请求的比率。这个值应该是大于 1 的非负整数。

  例如，如果容器的 **limit** 值中包括 **cpu: 500**， **request** 值中包括 **cpu: 100**，则 **cpu** 的限制与请求的比率是 **5**。这个比例必须小于或等于 **maxLimitRequestRatio**。

如果 **Pod** spec 没有指定容器资源内存或限制，则将限制范围对象中指定的容器的 **default** 或 **defaultRequest** CPU 和内存值分配给容器。

容器 LimitRange 对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "100m" 4
        memory: "4Mi" 5
      default:
        cpu: "300m" 6
        memory: "200Mi" 7
      defaultRequest:
        cpu: "200m" 8
        memory: "100Mi" 9
      maxLimitRequestRatio:
        cpu: "10" 10
```

1. LimitRange 对象的名称。
2. pod 中单个容器可以请求的最大 CPU 量。
3. pod 中单个容器可以请求的最大内存量。
4. pod 中单个容器可以请求的最小 CPU 量。
5. pod 中单个容器可以请求的最小内存量。
6. 如果未在 **Pod** spec 中指定，容器可以使用的默认 CPU 量。

7 如果未在 **Pod** spec 中指定，容器可以使用的默认内存量。

8 如果没有在 **Pod** spec 中指定，容器可以请求的默认 CPU 量。

9 如果未在 **Pod** spec 中指定，容器可以请求的默认内存量。

10 容器最大的限制与请求的比率。

8.3.1.1.2. Pod 限值

限制范围允许您为给定项目中所有 pod 的容器指定最小和最大 CPU 和内存限值。要在项目中创建容器，**Pod** spec 中的容器 CPU 和内存请求必须符合 **LimitRange** 对象中设置的值。如果没有，则 pod 不会被创建。

如果 **Pod** spec 没有指定容器资源内存或限制，则将限制范围对象中指定的容器的 **default** 或 **defaultRequest** CPU 和内存值分配给容器。

在 pod 中的所有容器中，需要满足以下条件：

- 对于在 **LimitRange** 对象中指定的 pod，容器 CPU 或内存请求和限制必须大于或等于 **min** 资源约束。
- 容器 CPU 或内存请求和限制必须小于或等于 **LimitRange** 对象中指定的 pod 的 **max** 资源约束。
- 容器限制与请求的比例必须小于或等于 **LimitRange** 对象中指定的 **maxLimitRequestRatio** 约束。

Pod LimitRange 对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" 2
        memory: "1Gi" 3
      min:
        cpu: "200m" 4
        memory: "6Mi" 5
      maxLimitRequestRatio:
        cpu: "10" 6
```

1 限制范围对象的名称。

2 pod 可在所有容器间请求的最大 CPU 量。

3 pod 可在所有容器间请求的最大内存量。

4 pod 可在所有容器间请求的最小 CPU 量。

5 pod 可在所有容器间请求的最小内存量。

6 容器最大的限制与请求的比率。

8.3.1.1.3. 镜像限制

LimitRange 对象允许您指定可推送到 OpenShift 镜像 registry 的镜像的最大大小。

将镜像推送到 OpenShift 镜像 registry 时，必须满足以下条件：

- 镜像的大小必须小于或等于 **LimitRange** 对象中指定的镜像的**最大值**。

镜像 LimitRange 对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/Image
      max:
        storage: 1Gi 2
```

1 **LimitRange** 对象的名称。

2 可推送到 OpenShift 镜像 registry 的镜像的最大大小。

注意

要防止超过限制的 Blob 上传到 registry，则必须将 registry 配置为强制实施配额。

警告

在上传的镜像清单中，镜像大小并非始终可用。这对使用 Docker 1.10 或更高版本构建并推送到 v2 registry 的镜像来说尤为如此。如果这样的镜像使用旧的 Docker 守护进程拉取，由 registry 将镜像清单转换为 schema v1 时缺少了所有与大小相关的信息。镜像没有设置存储限制会阻止镜像被上传。

这个问题正在被决。

8.3.1.1.4. 镜像流限值

LimitRange 对象允许您为镜像流指定限值。

对于每个镜像流，需要满足以下条件：

- **ImageStream** 规格中镜像标签的数量必须小于或等于 **LimitRange** 对象中的 **openshift.io/image-tags** 约束。

- **ImageStream** 规格中对镜像的唯一引用数量必须小于或等于限制范围对象中的 **openshift.io/images** 约束。

镜像流 LimitRange 对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: openshift.io/ImageStream
      max:
        openshift.io/image-tags: 20 2
        openshift.io/images: 30 3
```

1. **LimitRange** 对象的名称。
2. 镜像流 spec 中 **imagestream.spec.tags** 参数中唯一镜像标签的最大数量。
3. 镜像流 **spec** 中 **imagestream.status.tags** 参数中唯一镜像引用的最大数量。

openshift.io/image-tags 资源代表唯一镜像引用。可能的引用是 **ImageStreamTag**、**ImageStreamImage** 和 **DockerImage**。可以使用 **oc tag** 和 **oc import-image** 命令创建标签。内部和外部引用之间没有区别。但是，**ImageStream** 规格中标记的每个唯一引用仅计算一次。它不以任何方式限制推送到内部容器镜像 registry，但对标签限制很有用。

openshift.io/images 资源代表镜像流状态中记录的唯一镜像名称。它允许对可以推送到 OpenShift 镜像 registry 的多个镜像进行限制。内部和外部引用无法区分。

8.3.1.1.5. 持久性卷声明（PVC）限制

LimitRange 对象允许您限制持久性卷声明（PVC）中请求的存储。

在一个项目中的所有持久性卷声明中，必须满足以下条件：

- 持久性卷声明（PVC）中的资源请求必须大于或等于 **LimitRange** 对象中指定的 PVC 的 **min** 约束。
- 持久性卷声明（PVC）中的资源请求必须小于或等于 **LimitRange** 对象中指定的 PVC 的 **max** 约束。

PVC LimitRange 对象定义

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "PersistentVolumeClaim"
      min:
```

```
    storage: "2Gi" 2
  max:
    storage: "50Gi" 3
```

1. **LimitRange** 对象的名称。
2. 持久性卷声明中可请求的最小存储量。
3. 在持久性卷声明中请求的最大存储量。

8.3.2. 创建限制范围

将限制范围应用到一个项目：

1. 使用您的所需规格创建 **LimitRange** 对象：

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" 1
spec:
  limits:
    - type: "Pod" 2
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "200m"
        memory: "6Mi"
    - type: "Container" 3
      max:
        cpu: "2"
        memory: "1Gi"
      min:
        cpu: "100m"
        memory: "4Mi"
      default: 4
        cpu: "300m"
        memory: "200Mi"
      defaultRequest: 5
        cpu: "200m"
        memory: "100Mi"
      maxLimitRequestRatio: 6
        cpu: "10"
    - type: openshift.io/Image 7
      max:
        storage: 1Gi
    - type: openshift.io/ImageStream 8
      max:
        openshift.io/image-tags: 20
        openshift.io/images: 30
    - type: "PersistentVolumeClaim" 9
      min:
```

```
      storage: "2Gi"
    max:
      storage: "50Gi"
```

1. 为 **LimitRange** 对象指定一个名称。
2. 要为 pod 设置限值，请根据需要指定最小和最大 CPU 和内存请求。
3. 要为容器设置限值，请根据需要指定最小和最大 CPU 和内存请求。
4. 可选。对于容器，如果没有在 **Pod** spec 中指定，则指定容器可以使用的默认 CPU 或内存量。
5. 可选。对于容器，如果没有在 **Pod** spec 中指定，则指定容器可以请求的默认 CPU 或内存量。
6. 可选。对于容器，指定 **Pod** spec 中可指定的最大限制与请求比例。
7. 要为镜像对象设置限值，请设置可推送到 OpenShift 镜像 registry 的镜像的最大大小。
8. 要为镜像流设置限值，请根据需要设置 **ImageStream** 对象文件中的最大镜像标签和引用数。
9. 要为持久性卷声明设置限制，请设置可请求的最小和最大存储量。

2. 创建对象：

```
$ oc create -f <limit_range_file> -n <project> 1
```

1. 指定您创建的 YAML 文件的名称以及要应用限制的项目。

8.3.3. 查看限制

您可以通过在 web 控制台中导航到项目的 **Quota** 页面来查看项目中定义的任何限制。

您还可以使用 CLI 查看限制范围详情：

1. 获取项目中定义的 **LimitRange** 对象列表。例如，对于名为 **demoproject** 的项目：

```
$ oc get limits -n demoproject
```

```
NAME              CREATED AT
resource-limits   2020-07-15T17:14:23Z
```

2. 描述您感兴趣的 **LimitRange** 对象，如 **resource-limits** 限制范围：

```
$ oc describe limits resource-limits -n demoproject
```

```
Name:                   resource-limits
Namespace:              demoproject
Type                    Resource            Min     Max     Default Request Default Limit   Max
Limit/Request Ratio
----                    --------            ---     ---     --------------- -------------   -----------------------
```

```
Pod                    cpu                    200m    2       -            -            -
Pod                    memory                 6Mi     1Gi     -            -            -
Container              cpu                    100m    2       200m         300m         10
Container              memory                 4Mi     1Gi     100Mi        200Mi        -
openshift.io/Image     storage                -       1Gi     -            -            -
openshift.io/ImageStream  openshift.io/image  -       12      -            -            -
openshift.io/ImageStream  openshift.io/image-tags -    10      -            -            -
PersistentVolumeClaim  storage                -       50Gi    -            -            -
```

8.3.4. 删除限制范围

要删除任何活跃的 **LimitRange** 对象，使其不再在项目中强制实施限制：

- 运行以下命令:

```
$ oc delete limits <limit_name>
```

8.4. 配置集群内存以满足容器内存和风险要求

作为集群管理员，您可以通过以下方式管理应用程序内存，从而帮助集群有效运作：

- 确定容器化应用程序组件的内存和风险要求，并配置容器内存参数以满足这些要求。
- 配置容器化应用程序运行时（如 OpenJDK），以最佳的方式遵守配置的容器内存参数。
- 诊断并解决与在容器中运行相关的内存错误情况。

8.4.1. 了解管理应用程序内存

在继续操作前，建议您仔细阅读 OpenShift Container Platform 如何管理计算资源的概述。

对于每种资源（内存、CPU 或存储），OpenShift Container Platform 允许在 pod 中的各个容器上设置可选的 **request** 和 **limit** 值。

注意以下关于内存请求和内存限制的信息：

- **内存请求**
  - 如果指定，内存请求值会影响 OpenShift Container Platform 调度程序。将容器调度到节点时，调度程序会考虑内存请求，然后在所选节点上隔离出请求的内存供该容器使用。
  - 如果节点的内存已用尽，OpenShift Container Platform 将优先驱除其内存用量超出内存请求最多的容器。在严重的内存耗尽情形中，节点 OOM 终止程序可以根据类似的指标选择并终止容器中的一个进程。
  - 集群管理员可以分配配额，或者分配内存请求值的默认值。
  - 集群管理员可以覆盖开发人员指定的内存请求值，以便管理集群过量使用。
- **内存限制**
  - 如果指定，内存限制值针对可在容器中所有进程间分配的内存提供硬性限制。
  - 如果分配给容器中所有进程的内存超过内存限制，则节点超出内存（OOM）终止程序将立即选择并终止容器中的一个进程。

- 如果同时指定了内存请求和限制，则内存限制必须大于或等于内存请求量。
- 集群管理员可以分配配额，或者分配内存限制值的默认值。
- 最小内存限值为 12MB。如果容器因为一个 **Cannot allocate memory** pod 事件启动失败，这代表内存限制太低。增加或删除内存限制。删除限制可让 pod 消耗无限的节点资源。

8.4.1.1. 管理应用程序内存策略

如下是 OpenShift Container Platform 上调整应用程序内存大小的步骤：

1. **确定预期的容器内存用量**
   从经验判断（例如，通过独立的负载测试），根据需要确定容器内存用量的预期平均值和峰值。需要考虑容器中有可能并行运行的所有进程：例如，主应用程序是否生成任何辅助脚本？

2. **确定风险嗜好**
   确定用于驱除的风险嗜好。如果风险嗜好较低，则容器应根据预期的峰值用量加上一个安全裕度百分比来请求内存。如果风险嗜好较高，那么根据预期的平均用量请求内存可能更为妥当。

3. **设定容器内存请求**
   根据以上所述设定容器内存请求。请求越能准确表示应用程序内存用量越好。如果请求过高，集群和配额用量效率低下。如果请求过低，应用程序驱除的几率就会提高。

4. **根据需要设定容器内存限制**
   在必要时，设定容器内存限制。如果容器中所有进程的总内存用量超过限制，那么设置限制会立即终止容器进程，所以这既有利也有弊。一方面，可能会导致过早出现意料之外的过量内存使用（“快速失败”）；另一方面，也会突然终止进程。

   需要注意的是，有些 OpenShift Container Platform 集群可能要求设置限制；有些集群可能会根据限制覆盖请求；而且有些应用程序镜像会依赖于设置的限制，因为这比请求值更容易检测。

   如果设置内存限制，其大小不应小于预期峰值容器内存用量加上安全裕度百分比。

5. **确保应用程序经过性能优化**
   在适当时，确保应用程序已根据配置的请求和限制进行了性能优化。对于池化内存的应用程序（如 JVM），这一步尤为相关。本页的其余部分将介绍这方面的内容。

其他资源

- 了解计算资源和容器

8.4.2. 了解 OpenShift Container Platform 的 OpenJDK 设置

默认的 OpenJDK 设置在容器化环境中效果不佳。因此在容器中运行 OpenJDK 时，务必要提供一些额外的 Java 内存设置。

JVM 内存布局比较复杂，并且视版本而异，因此本文不做详细讨论。但作为在容器中运行 OpenJDK 的起点，至少以下三个于内存相关的任务非常重要：

1. 覆盖 JVM 最大堆大小。
2. 在可能的情况下，促使 JVM 向操作系统释放未使用的内存。
3. 确保正确配置了容器中的所有 JVM 进程。

优化容器中运行的 JVM 工作负载已超出本文讨论范畴，并且可能涉及设置多个额外的 JVM 选项。

8.4.2.1. 了解如何覆盖 JVM 最大堆大小

对于许多 Java 工作负载，JVM 堆是最大的内存用户。目前，OpenJDK 默认允许将计算节点最多 1/4 (1/**-XX:MaxRAMFraction**) 的内存用于该堆，不论 OpenJDK 是否在容器内运行。因此，务**必要**覆盖此行为，特别是设置了容器内存限制时。

达成以上目标至少有两种方式：

- 如果设置了容器内存限制，并且 JVM 支持那些实验性选项，请设置 **-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap**。

  注意

  UseCGroupMemoryLimitForHeap 选项已在 JDK 11 中删除。使用 **-XX:+UseContainerSupport 替代**。

  这会将 **-XX:MaxRAM** 设置为容器内存限制，并将最大堆大小 (**-XX:MaxHeapSize** / **-Xmx**) 设置为 1/**-XX:MaxRAMFraction**（默认为 1/4）。

- 直接覆盖 **-XX:MaxRAM**、**-XX:MaxHeapSize** 或 **-Xmx**。
  这个选项涉及对值进行硬编码，但也有允许计算安全裕度的好处。

8.4.2.2. 了解如何促使 JVM 向操作系统释放未用的内存

默认情况下，OpenJDK 不会主动向操作系统退还未用的内存。这可能适合许多容器化的 Java 工作负载，但也有明显的例外，例如额外活跃进程与容器内 JVM 共存的工作负载，这些额外进程是原生或附加的 JVM，或者这两者的组合。

基于 Java 的代理可使用以下 JVM 参数来鼓励 JVM 向操作系统释放未使用的内存：

```
-XX:+UseParallelGC
-XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4
-XX:AdaptiveSizePolicyWeight=90.
```

这些参数旨在当分配的内存超过 110% 使用中内存时 (**-XX:MaxHeapFreeRatio**) 将堆内存返还给操作系统，这将在垃圾回收器上最多花费 20% 的 CPU 时间 (**-XX:GCTimeRatio**)。应用程序堆分配一定不会小于初始堆分配（被 **-XX:InitialHeapSize** / **- Xms** 覆盖）。调节 Java 在 OpenShift 中的内存占用（第 1 部分）、调节 Java 在 OpenShift 中的内存占用（第 2 部分） 以及 OpenJDK 和容器提供了其他的详细信息。

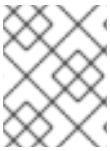
8.4.2.3. 了解如何确保正确配置容器中的所有 JVM 进程

如果多个 JVM 在同一容器中运行，则必须保证它们的配置都正确无误。如果有许多工作负载，需要为每个 JVM 分配一个内存预算百分比，留出较大的额外安全裕度。

许多 Java 工具使用不同的环境变量(**JAVA_OPTS**、**GRADLE_OPTS** 等)来配置其 JVM，并确保将正确的设置传递给正确的 JVM。

OpenJDK 始终尊重 **JAVA_TOOL_OPTIONS** 环境变量，在 **JAVA_TOOL_OPTIONS** 中指定的值会被 JVM 命令行中指定的其他选项覆盖。默认情况下，为了确保这些选项默认用于在基于 Java 的代理镜像中运行的所有 JVM 工作负载，OpenShift Container Platform Jenkins Maven 代理镜像集：

```
JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"
```



注意

UseCGroupMemoryLimitForHeap 选项已在 JDK 11 中删除。使用 **-XX:+UseContainerSupport** 替代。

这不能保证不需要额外选项，只是用作一个实用的起点。

8.4.3. 从 pod 中查找内存请求和限制

希望从 pod 中动态发现内存请求和限制的应用程序应该使用 Downward API。

流程

1. 配置 pod，以添加 **MEMORY_REQUEST** 和 **MEMORY_LIMIT** 小节：
   a. 创建一个类似以下示例的 YAML 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: test
    image: fedora:latest
    command:
    - sleep
    - "3600"
    env:
    - name: MEMORY_REQUEST (1)
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: requests.memory
    - name: MEMORY_LIMIT (2)
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: limits.memory
    resources:
      requests:
        memory: 384Mi
      limits:
        memory: 512Mi
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
```

(1) 添加此小节来发现应用程序内存请求值。

2 添加此小节来发现应用程序内存限制值。

b. 运行以下命令来创建 pod：

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

验证

1. 使用远程 shell 访问 pod：

```
$ oc rsh test
```

2. 检查是否应用了请求的值：

```
$ env | grep MEMORY | sort
```

输出示例

```
MEMORY_LIMIT=536870912
MEMORY_REQUEST=402653184
```

注意

内存限制值也可由 **/sys/fs/cgroup/memory/memory.limit_in_bytes** 文件从容器内部读取。

8.4.4. 了解 OOM 终止策略

如果容器中所有进程的内存总用量超过内存限制，或者在严重的节点内存耗尽情形下，OpenShift Container Platform 可以终止容器中的某个进程。

当进程超出内存（OOM）终止时，这可能会导致容器立即退出。如果容器 PID 1 进程收到 **SIGKILL**，则容器会立即退出。否则，容器行为将取决于其他进程的行为。

例如，某个容器进程以代码 137 退出，这表示它收到了 SIGKILL 信号。

如果容器没有立即退出，则能够检测到 OOM 终止，如下所示：

1. 使用远程 shell 访问 pod：

```
# oc rsh test
```

2. 运行以下命令，查看 **/sys/fs/cgroup/memory/memory.oom_control** 中的当前 OOM 终止计数：

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
```

输出示例

```
oom_kill 0
```

3. 运行以下命令来引发一个 OOM kill：

```
$ sed -e '' </dev/zero
```

输出示例

```
Killed
```

4. 运行以下命令查看 **sed** 命令的退出状态：

```
$ echo $?
```

输出示例

```
137
```

例如，**137** 代表容器进程以代码 137 退出，这表示它收到了 SIGKILL 信号。

5. 运行以下命令，查看 **/sys/fs/cgroup/memory/memory.oom_control** 中的 OOM 终止计数器：

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
```

输出示例

```
oom_kill 1
```

如果 pod 中的一个或多个进程遭遇 OOM 终止，那么当 pod 随后退出时（不论是否立即发生），它都将会具有原因为 **OOMKilled** 的 **Failed** 阶段。被 OOM 终止的 pod 可能会根据 **restartPolicy** 的值重启。如果不重启，复制控制器等控制器会看到 pod 的失败状态，并创建一个新 pod 来替换旧 pod。

使用以下命令获取 pod 状态：

```
$ oc get pod test
```

输出示例

```
NAME      READY     STATUS      RESTARTS   AGE
test      0/1       OOMKilled   0          1m
```

- 如果 pod 没有重启，请运行以下命令来查看 pod:

```
$ oc get pod test -o yaml
```

输出示例

```
...
status:
  containerStatuses:
  - name: test
    ready: false
```

```
    restartCount: 0
    state:
      terminated:
        exitCode: 137
        reason: OOMKilled
  phase: Failed
```

- 如果重启，运行以下命令来查看 pod:

```
$ oc get pod test -o yaml
```

输出示例

```
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
    lastState:
      terminated:
        exitCode: 137
        reason: OOMKilled
    state:
      running:
  phase: Running
```

8.4.5. 了解 pod 驱除

OpenShift Container Platform 可在节点内存耗尽时从节点上驱除 pod。根据内存耗尽的程度，驱除可能是安全操作，但也不一定。安全驱除表示，各个容器的主进程 (PID 1) 收到 SIGTERM 信号，稍等片刻后，如果进程还未退出，则会收到一个 SIGKILL 信号。非安全驱除暗示着各个容器的主进程会立即收到 SIGKILL 信号。

被驱除的 pod 具有 **Failed** 阶段，原因为 **Evicted**。无论 **restartPolicy** 的值是什么，该 pod 都不会重启。但是，复制控制器等控制器会看到 pod 的失败状态，并且创建一个新 pod 来取代旧 pod。

```
$ oc get pod test
```

输出示例

```
NAME      READY     STATUS    RESTARTS   AGE
test      0/1       Evicted   0          1m
```

```
$ oc get pod test -o yaml
```

输出示例

```
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
```

```
phase: Failed
reason: Evicted
```

8.5. 配置集群以将 POD 放置到过量使用的节点上

处于*过量使用（overcommited）*状态时，容器计算资源请求和限制的总和超过系统中可用的资源。例如，您可以在一个开发环境中使用过量使用功能，因为在这种环境中可以接受以牺牲保障性能来换取功能的情况。

容器可以指定计算资源的请求（request）和限值（limit）。请求用于调度容器，以提供最低服务保证。限值用于约束节点上可以消耗的计算资源数量。

调度程序会尝试优化集群中所有节点的计算资源使用。它将 pod 放置到特定的节点上，同时考虑 pod 的计算资源请求和节点的可用容量。

OpenShift Container Platform 管理员可以控制过量使用的程度，并管理节点上的容器密度。您可以使用 ClusterResourceOverride Operator 配置集群一级的过量使用，以覆盖开发人员容器上设置的请求和限值之间的比例。与 节点过量使用、项目内存以及 CPU 限值和默认值 一同使用，您可以调整资源限值和请求，以达到所需的过量使用程度。

注意

在 OpenShift Container Platform 中，您必须启用集群级别的过量使用。节点过量使用功能会被默认启用。请参阅禁用节点过量使用。

8.5.1. 资源请求和过量使用

对于每个计算资源，容器可以指定一个资源请求和限制。根据确保节点有足够可用容量以满足请求值的请求来做出调度决策。如果容器指定了限制，但忽略了请求，则请求会默认采用这些限制。容器无法超过节点上指定的限制。

限制的强制实施取决于计算资源类型。如果容器没有请求或限制，容器会调度到没有资源保障的节点。在实践中，容器可以在最低本地优先级适用的范围内消耗指定的资源。在资源较少的情况下，不指定资源请求的容器将获得最低的服务质量。

调度基于请求的资源，而配额和硬限制指的是资源限制，它们可以设置为高于请求的资源。请求和限制的差值决定了过量使用程度；例如，如果为容器赋予 1Gi 内存请求和 2Gi 内存限制，则根据 1Gi 请求将容器调度到节点上，但最多可使用 2Gi；因此过量使用为 200%。

8.5.2. 使用 Cluster Resource Override Operator 的集群级别的过量使用

Cluster Resource Override Operator 是一个准入 Webhook，可让您控制过量使用的程度，并在集群中的所有节点上管理容器密度。Operator 控制特定项目中节点可以如何超过定义的内存和 CPU 限值。

您必须使用 OpenShift Container Platform 控制台或 CLI 安装 Cluster Resource override Operator，如下所示。在安装过程中，您会创建一个 **ClusterResourceOverride** 自定义资源 (CR)，其中设置过量使用级别，如下例所示：

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster 1
spec:
  podResourceOverride:
```

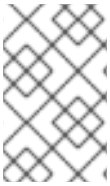
```
  spec:
    memoryRequestToLimitPercent: 50 (2)
    cpuRequestToLimitPercent: 25 (3)
    limitCPUToMemoryPercent: 200 (4)
# ...
```

(1) 名称必须是 **cluster**。

(2) 可选。如果指定或默认指定了容器内存限值，则该内存请求会覆盖到限值的这个百分比，从 1 到 100 之间。默认值为 50.

(3) 可选。如果指定或默认指定了容器 CPU 限值，则将 CPU 请求覆盖到限值的这个百分比，从 1 到 100 之间。默认值为 25。

(4) 可选。如果指定或默认指定了容器内存限值，则 CPU 限值将覆盖的内存限值的百分比（如果指定）。以 100% 扩展 1Gi RAM， 等于 1 个 CPU 内核。这会在覆盖 CPU 请求前进行（如果配置了）。默认值为 200。



注意

如果容器上没有设置限值，则 Cluster Resourceoverride Operator 覆盖无效。创建一个针对单独项目的带有默认限制的 **LimitRange** 对象，或在 **Pod** specs 中配置要应用的覆盖的限制。

配置后，可通过将以下标签应用到每个项目的命名空间对象来启用每个项目的覆盖：

```
apiVersion: v1
kind: Namespace
metadata:

# ...

  labels:
    clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true"

# ...
```

Operator 监视 **ClusterResourceOverride** CR， 并确保 **ClusterResourceOverride** 准入 Webhook 被安装到与 Operator 相同的命名空间。

8.5.2.1. 使用 Web 控制台安装 Cluster Resource Override Operator

您可以使用 OpenShift Container Platform Web 控制台来安装 Cluster Resource Override Operator，以帮助控制集群中的过量使用。

先决条件

- 如果容器上未设置限值，Cluster Resourceoverride Operator 将没有作用。您必须使用一个 **LimitRange** 对象为项目指定默认限值，或在 **Pod** spec 中配置要应用的覆盖的限制。

流程

使用 OpenShift Container Platform web 控制台安装 Cluster Resource Override Operator：

1. 在 OpenShift Container Platform web 控制台中进入 **Home → Projects**
   a. 点击 **Create Project**。
   b. 指定 **clusterresourceoverride-operator** 作为项目的名称。
   c. 点击 **Create**。
2. 进入 **Operators → OperatorHub**。
   a. 从可用 Operator 列表中选择 **ClusterResourceOverride Operator**，再点击 **Install**。
   b. 在 **Install Operator** 页面中，确保为 **Installation Mode** 选择了 **A specific Namespace on the cluster**。
   c. 确保为 **Installed Namespace** 选择了 **clusterresourceoverride-operator**。
   d. 指定**更新频道**和**批准策略**。
   e. 点击 **Install**。
3. 在 **Installed Operators** 页面中，点 **ClusterResourceOverride**。
   a. 在 **ClusterResourceOverride Operator** 详情页面中，点 **Create ClusterResourceOverride**。
   b. 在 **Create ClusterResourceOverride** 页面中，点 **YAML 视图**并编辑 YAML 模板，以根据需要设置过量使用值：

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
# ...
```

(1) 名称必须是 **cluster**。

(2) 可选。指定在 1-100 之间覆盖容器内存限值的百分比（如果使用的话）。默认值为 50.

(3) 可选。指定在 1-100 之间覆盖容器 CPU 限值的百分比（如果使用的话）。默认值为 25。

(4) 可选。如果使用，请指定覆盖容器内存限值的百分比。以 100% 扩展 1Gi RAM， 等于 1 个 CPU 内核。这会在覆盖 CPU 请求前进行处理（如果已配置）。默认值为 200。

   c. 点击 **Create**。
4. 通过检查集群自定义资源的状态来检查准入 Webhook 的当前状态：
   a. 在 **ClusterResourceOverride Operator** 页面中，点击 **cluster**。

b. 在 **ClusterResourceOverride Details** 页中，点 **YAML**。当 webhook 被调用时，**mutatingWebhookConfigurationRef** 项会出现。

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","metadata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLimitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
      memoryRequestToLimitPercent: 50
status:

# ...

    mutatingWebhookConfigurationRef: (1)
      apiVersion: admissionregistration.k8s.io/v1
      kind: MutatingWebhookConfiguration
      name: clusterresourceoverrides.admission.autoscaling.openshift.io
      resourceVersion: "127621"
      uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

# ...
```

(1) 引用 **ClusterResourceOverride** 准入Webhook。

8.5.2.2. 使用 CLI 安装 Cluster Resource Override Operator

您可以使用 OpenShift Container Platform CLI 来安装 Cluster Resource Override Operator，以帮助控制集群中的过量使用。

先决条件

- 如果容器上未设置限值，Cluster Resourceoverride Operator 将没有作用。您必须使用一个 **LimitRange** 对象为项目指定默认限值，或在 **Pod** spec 中配置要应用的覆盖的限制。

流程

使用 CLI 安装 Cluster Resource Override Operator：

1. 为 Cluster Resource Override Operator 创建命名空间：

a. 为 Cluster Resource Override Operator 创建一个 **Namespace** 空间对象 YAML 文件（如 **cro-namespace.yaml**）：

```
apiVersion: v1
kind: Namespace
metadata:
  name: clusterresourceoverride-operator
```

b. 创建命名空间：

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

例如：

```
$ oc create -f cro-namespace.yaml
```

2. 创建一个 Operator 组：

a. 为 Cluster Resource Override Operator 创建一个 **OperatorGroup** 对象 YAML 文件（如 cro-og.yaml）：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: clusterresourceoverride-operator
  namespace: clusterresourceoverride-operator
spec:
  targetNamespaces:
    - clusterresourceoverride-operator
```

b. 创建 Operator 组：

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

例如：

```
$ oc create -f cro-og.yaml
```

3. 创建一个订阅：

a. 为 Cluster Resourceoverride Operator 创建一个 **Subscription** 对象 YAML 文件（如 cro-sub.yaml):

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: clusterresourceoverride
  namespace: clusterresourceoverride-operator
spec:
  channel: "4.16"
  name: clusterresourceoverride
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

b. 创建订阅：

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

例如：

```
$ oc create -f cro-sub.yaml
```

4. 在 **clusterresourceoverride-operator** 命名空间中创建 **ClusterResourceOverride** 自定义资源（CR）对象：

a. 进入 **clusterresourceoverride-operator** 命名空间。

```
$ oc project clusterresourceoverride-operator
```

b. 为 Cluster Resourceoverride Operator 创建 **ClusterResourceOverride** 对象 YAML 文件（如 cro-cr.yaml):

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
   name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
```

(1) 名称必须是 **cluster**。

(2) 可选。指定在 1-100 之间覆盖容器内存限值的百分比（如果使用的话）。默认值为 50.

(3) 可选。指定在 1-100 之间覆盖容器 CPU 限值的百分比（如果使用的话）。默认值为 25。

(4) 可选。如果使用，请指定覆盖容器内存限值的百分比。以 100% 扩展 1Gi RAM， 等于 1 个 CPU 内核。这会在覆盖 CPU 请求前进行处理（如果已配置）。默认值为 200。

c. 创建 **ClusterResourceOverride** 对象：

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

例如：

```
$ oc create -f cro-cr.yaml
```

5. 通过检查集群自定义资源的状态来验证准入 Webhook 的当前状态。

```
$ oc get clusterresourceoverride cluster -n clusterresourceoverride-operator -o yaml
```

当 webhook 被调用时， **mutatingWebhookConfigurationRef** 项会出现。

输出示例

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"operator.autoscaling.openshift.io/v1","kind":"ClusterResourceOverride","metadata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLimitPercent":50}}}}
  creationTimestamp: "2019-12-18T22:35:02Z"
  generation: 1
  name: cluster
  resourceVersion: "127622"
  selfLink: /apis/operator.autoscaling.openshift.io/v1/clusterresourceoverrides/cluster
  uid: 978fc959-1717-4bd1-97d0-ae00ee111e8d
spec:
  podResourceOverride:
    spec:
      cpuRequestToLimitPercent: 25
      limitCPUToMemoryPercent: 200
      memoryRequestToLimitPercent: 50
status:

# ...

    mutatingWebhookConfigurationRef: (1)
      apiVersion: admissionregistration.k8s.io/v1
      kind: MutatingWebhookConfiguration
      name: clusterresourceoverrides.admission.autoscaling.openshift.io
      resourceVersion: "127621"
      uid: 98b3b8ae-d5ce-462b-8ab5-a729ea8f38f3

# ...
```

(1) 引用 **ClusterResourceOverride** 准入Webhook。

8.5.2.3. 配置集群级别的过量使用

Cluster Resource Override Operator 需要一个 **ClusterResourceOverride** 自定义资源 (CR)，以及您希望 Operator 来控制过量使用的每个项目的标识。

先决条件

- 如果容器上未设置限值，Cluster Resourceoverride Operator 将没有作用。您必须使用一个 **LimitRange** 对象为项目指定默认限值，或在 **Pod** spec 中配置要应用的覆盖的限制。

流程

修改集群级别的过量使用：

1. 编辑 **ClusterResourceOverride** CR:

```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
   name: cluster
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (1)
      cpuRequestToLimitPercent: 25 (2)
      limitCPUToMemoryPercent: 200 (3)
# ...
```

(1) 可选。指定在 1-100 之间覆盖容器内存限值的百分比（如果使用的话）。默认值为 50.

(2) 可选。指定在 1-100 之间覆盖容器 CPU 限值的百分比（如果使用的话）。默认值为 25。

(3) 可选。如果使用，请指定覆盖容器内存限值的百分比。以 100% 扩展 1Gi RAM， 等于 1 个 CPU 内核。这会在覆盖 CPU 请求前进行处理（如果已配置）。默认值为 200。

2. 确保在每个您希望 Cluster Resourceoverride Operator 来控制过量使用的项目中都添加了以下标识：

```
apiVersion: v1
kind: Namespace
metadata:

# ...

  labels:
    clusterresourceoverrides.admission.autoscaling.openshift.io/enabled: "true" (1)

# ...
```

(1) 把这个标识添加到每个项目。

8.5.3. 节点级别的过量使用

您可以使用各种方法来控制特定节点上的过量使用，如服务质量 (QOS) 保障、CPU 限值或保留资源。您还可以为特定节点和特定项目禁用过量使用功能。

8.5.3.1. 了解计算资源和容器

计算资源的节点强制行为特定于资源类型。

8.5.3.1.1. 了解容器 CPU 请求

容器可以保证获得其请求的 CPU 量，还可额外消耗节点上提供的超额 CPU，但不会超过容器指定的限制。如果多个容器试图使用超额 CPU，则会根据每个容器请求的 CPU 数量来分配 CPU 时间。

例如，如果一个容器请求了 500m CPU 时间，另一个容器请求了 250m CPU 时间，那么该节点上提供的额外 CPU 时间以 2:1 比例在这两个容器之间分配。如果容器指定了一个限制，它将被限速，无法使用超过指定限制的 CPU。使用 Linux 内核中的 CFS 共享支持强制实施 CPU 请求。默认情况下，使用 Linux 内核中的 CFS 配额支持以 100ms 测量间隔强制实施 CPU 限制，但这可以禁用。

8.5.3.1.2. 了解容器内存请求

容器可以保证获得其请求的内存量。容器可以使用高于请求量的内存，但一旦超过请求量，就有可能在节点上遇到内存不足情形时被终止。如果容器使用的内存少于请求量，它不会被终止，除非系统任务或守护进程需要的内存量超过了节点资源保留考虑在内的内存量。如果容器指定了内存限制，则超过限制数量时会立即被终止。

8.5.3.2. 了解过量使用和服务质量类

当节点上调度了没有发出请求的 pod，或者节点上所有 pod 的限制总和超过了机器可用容量时，该节点处于*过量使用*状态。

在过量使用环境中，节点上的 pod 可能会在任意给定时间点尝试使用超过可用量的计算资源。发生这种情况时，节点必须为 pod 赋予不同的优先级。有助于做出此决策的工具称为服务质量 (QoS) 类。

pod 被指定为三个 QoS 类中的一个，带有降序排列：

表 8.19. 服务质量类

优先级	类名称	描述
1（最高）	**Guaranteed**	如果为所有资源设置了限制和可选请求（不等于 0）并且它们相等，则 pod 被归类为 **Guaranteed**。
2	**Burstable**	如果为所有资源设置了请求和可选限制（不等于 0）并且它们不相等，则 pod 被归类为 **Burstable**。
3（最低）	**BestEffort**	如果没有为任何资源设置请求和限制，则 pod 被归类为 **BestEffort**。

内存是一种不可压缩的资源，因此在内存量较低的情况下，优先级最低的容器首先被终止：

- **Guaranteed** 容器优先级最高，并且保证只有在它们超过限制或者系统遇到内存压力且没有优先级更低的容器可被驱除时，才会被终止。
- 在遇到系统内存压力时，**Burstable** 容器如果超过其请求量并且不存在其他 **BestEffort** 容器，则有较大的可能会被终止。
- **BestEffort** 容器被视为优先级最低。系统内存不足时，这些容器中的进程最先被终止。

8.5.3.2.1. 了解如何为不同的服务质量层级保留内存

您可以使用 **qos-reserved** 参数指定在特定 QoS 级别上 pod 要保留的内存百分比。此功能尝试保留请求的资源，阻止较低 QoS 类中的 pod 使用较高 QoS 类中 pod 所请求的资源。

OpenShift Container Platform 按照如下所示使用 **qos-reserved** 参数：

- 值为 **qos-reserved=memory=100%** 时，阻止 **Burstable** 和 **BestEffort** QoS 类消耗较高 QoS 类所请求的内存。这会增加 **BestEffort** 和 **Burstable** 工作负载上为了提高 **Guaranteed** 和 **Burstable** 工作负载的内存资源保障而遭遇 OOM 的风险。
- 值为 **qos-reserved=memory=50%** 时，允许 **Burstable** 和 **BestEffort** QoS 类消耗较高 QoS 类所请求的内存的一半。
- 值为 **qos-reserved=memory=0%** 时，允许 **Burstable** 和 **BestEffort** QoS 类最多消耗节点的所

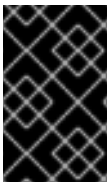
- 值为 **qos-reserved=memory=0%** 时，允许 **Burstable** 和 **BestEffort** QoS 类最多消耗节点的所有可分配数量（若可用），但会增加 **Guaranteed** 工作负载不能访问所请求内存的风险。此条件等同于禁用这项功能。

8.5.3.3. 了解交换内存和 QoS

您可以在节点上默认禁用交换，以便保持服务质量 (QoS) 保障。否则，节点上的物理资源会超额订阅，从而影响 Kubernetes 调度程序在 pod 放置过程中所做的资源保障。

例如，如果两个有保障 pod 达到其内存限制，各个容器可以开始使用交换内存。最终，如果没有足够的交换空间，pod 中的进程可能会因为系统被超额订阅而被终止。

如果不禁用交换，会导致节点无法意识到它们正在经历 **MemoryPressure**，从而造成 pod 无法获得它们在调度请求中索取的内存。这样节点上就会放置更多 pod，进一步增大内存压力，最终增加遭遇系统内存不足 (OOM) 事件的风险。



重要

如果启用了交换，则对于可用内存的资源不足处理驱除阈值将无法正常发挥作用。利用资源不足处理，允许在遇到内存压力时从节点中驱除 pod，并且重新调度到没有此类压力的备选节点上。

8.5.3.4. 了解节点过量使用

在过量使用的环境中，务必要正确配置节点，以提供最佳的系统行为。

当节点启动时，它会确保为内存管理正确设置内核可微调标识。除非物理内存不足，否则内核应该永不会在内存分配时失败。

为确保这一行为，OpenShift Container Platform 通过将 **vm.overcommit_memory** 参数设置为 **1** 来覆盖默认操作系统设置，从而将内核配置为始终过量使用内存。

OpenShift Container Platform 还通过将 **vm.panic_on_oom** 参数设置为 **0**，将内核配置为不会在内存不足时崩溃。设置为 0 可告知内核在内存不足 (OOM) 情况下调用 oom_killer，以根据优先级终止进程

您可以通过对节点运行以下命令来查看当前的设置：

```
$ sysctl -a |grep commit
```

输出示例

```
#...
vm.overcommit_memory = 0
#...
```

```
$ sysctl -a |grep panic
```

输出示例

```
#...
vm.panic_on_oom = 0
#...
```

注意

节点上应该已设置了上述标记，不需要进一步操作。

您还可以为每个节点执行以下配置：

- 使用 CPU CFS 配额禁用或强制实施 CPU 限制
- 为系统进程保留资源
- 为不同的服务质量等级保留内存

8.5.3.5. 使用 CPU CFS 配额禁用或强制实施 CPU 限制

默认情况下，节点使用 Linux 内核中的完全公平调度程序 (CFS) 配额支持来强制实施指定的 CPU 限制。

如果禁用了 CPU 限制强制实施，了解其对节点的影响非常重要：

- 如果容器有 CPU 请求，则请求仍由 Linux 内核中的 CFS 共享来实施。
- 如果容器没有 CPU 请求，但没有 CPU 限制，则 CPU 请求默认为指定的 CPU 限值，并由 Linux 内核中的 CFS 共享强制。
- 如果容器同时具有 CPU 请求和限制，则 CPU 请求由 Linux 内核中的 CFS 共享强制实施，且 CPU 限制不会对节点产生影响。

先决条件

- 输入以下命令为您要配置的节点类型获取与静态 **MachineConfigPool** CRD 关联的标签：

```
$ oc edit machineconfigpool <name>
```

例如：

```
$ oc edit machineconfigpool worker
```

输出示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: "2022-11-16T15:34:25Z"
  generation: 4
  labels:
    pools.operator.machineconfiguration.openshift.io/worker: "" 1
  name: worker
```

1 标签会出现在 Labels 下。

提示

如果标签不存在，请添加键/值对，例如：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

禁用 CPU 限制的示例配置

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: disable-cpu-units (1)
spec:
  machineConfigPoolSelector:
    matchLabels:
      pools.operator.machineconfiguration.openshift.io/worker: "" (2)
  kubeletConfig:
    cpuCfsQuota: false (3)
```

(1) 为 CR 分配一个名称。

(2) 指定机器配置池中的标签。

(3) 将 **cpuCfsQuota** 参数设置为 **false**。

2. 运行以下命令来创建 CR：

```
$ oc create -f <file_name>.yaml
```

8.5.3.6. 为系统进程保留资源

为提供更可靠的调度并且最大程度减少节点资源过量使用，每个节点都可以保留一部分资源供系统守护进程使用（节点上必须运行这些守护进程才能使集群正常工作）。特别是，建议您为内存等不可压缩的资源保留资源。

流程

要明确为非 pod 进程保留资源，请通过指定可用于调度的资源来分配节点资源。如需了解更多详细信息，请参阅“为节点分配资源”。

8.5.3.7. 禁用节点过量使用

启用之后，可以在每个节点上禁用过量使用。

流程

要在节点中禁用过量使用，请在该节点上运行以下命令：

```
$ sysctl -w vm.overcommit_memory=0
```

8.5.4. 项目级别限值

为帮助控制过量使用，您可以设置每个项目的资源限值范围，为过量使用无法超过的项目指定内存和 CPU 限值，以及默认值。

如需有关项目级别资源限值的信息，请参阅附加资源。

另外，您可以为特定项目禁用过量使用。

8.5.4.1. 禁用项目过量使用

启用之后，可以按项目禁用过量使用。例如，您可以允许独立于过量使用配置基础架构组件。

流程

在某个项目中禁用过量使用：

1. 创建或编辑命名空间对象文件。
2. 添加以下注解：

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    quota.openshift.io/cluster-resource-override-enabled: "false" 1
# ...
```

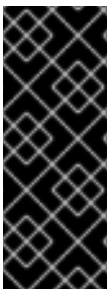
1 将此注解设置为 **false** 可禁用这个命名空间的过量使用。

8.5.5. 其他资源

- 设置部署资源。
- 为节点分配资源。

8.6. 在节点上配置 LINUX CGROUP 版本

自 OpenShift Container Platform 4.14 起，OpenShift Container Platform 在集群中使用 Linux 控制组版本 2 (cgroup v2)。如果您在 OpenShift Container Platform 4.13 或更早版本中使用 cgroup v1，迁移到 OpenShift Container Platform 4.14 或更高版本不会自动将 cgroup 配置更新至版本 2。全新安装 OpenShift Container Platform 4.14 或更高版本默认使用 cgroup v2。但是，您可以在安装时启用 Linux 控制组群版本 1 (cgroup v1)。



重要

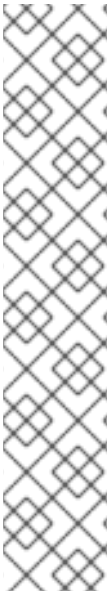
cgroup v1 是一个已弃用的功能。弃用的功能仍然包含在 OpenShift Container Platform 中，并将继续被支持。但是，这个功能会在以后的发行版本中被删除，且不建议在新的部署中使用。

有关 OpenShift Container Platform 中已弃用或删除的主要功能的最新列表，请参阅 OpenShift Container Platform 发行注记中 *已弃用和删除的功能*部分。

cgroup v2 是 Linux cgroup API 的当前版本。cgroup v2 比 cgroup v1 提供多种改进，包括统一层次结构、

更安全的子树委派、新功能，如 Pressure Stall Information，以及增强的资源管理和隔离。但是，cgroup v2 与 cgroup v1 具有不同的 CPU、内存和 I/O 管理特征。因此，在运行 cgroup v2 的集群上，一些工作负载可能会遇到内存或 CPU 用量差异。

您可以根据需要在 cgroup v1 和 cgroup v2 之间更改。在 OpenShift Container Platform 中启用 cgroup v1 禁用集群中的所有 cgroup v2 控制器和层次结构。



注意

- 如果您运行依赖于 cgroup 文件系统的第三方监控和安全代理，请将代理更新为支持 cgroup v2 的版本。
- 如果您配置了 cgroup v2，并将 cAdvisor 作为监控 pod 和容器的独立守护进程集运行，请将 cAdvisor 更新至 v0.43.0 或更高版本。
- 如果您部署 Java 应用程序，请使用完全支持 cgroup v2 的版本，如以下软件包：
  - OpenJDK / HotSpot: jdk8u372, 11.0.16, 15 及更新版本
  - NodeJs 20.3.0 或更高版本
  - IBM Semeru Runtimes: jdk8u345-b01, 11.0.16.0, 17.0.4.0, 18.0.2.0 及更新版本
  - IBM SDK Java 技术版本(IBM Java): 8.0.7.15 及更新版本

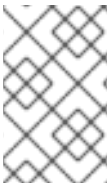
8.6.1. 配置 Linux cgroup

您可以通过编辑 **node.config** 对象来启用 Linux 控制组群版本 1 (cgroup v1) 或 Linux 控制组群版本 2 (cgroup v2)。默认值为 cgroup v2。

重要

cgroup v1 是一个已弃用的功能。弃用的功能仍然包含在 OpenShift Container Platform 中，并将继续被支持。但是，这个功能会在以后的发行版本中被删除，且不建议在新的部署中使用。

有关 OpenShift Container Platform 中已弃用或删除的主要功能的最新列表，请参阅 OpenShift Container Platform 发行注记中 *已弃用和删除的功能*部分。



注意

目前，cgroup v2 不支持禁用 CPU 负载均衡。因此，如果您启用了 cgroup v2，则可能无法从性能配置集中获取所需的行为。如果您使用 executeace 配置集，则不建议启用 cgroup v2。

先决条件

- 您有一个正在运行的 OpenShift Container Platform 集群，它使用版本 4.12 或更高版本。
- 以具有管理特权的用户身份登录集群。

流程

1. 在节点上启用 cgroup v1：

a. 编辑 **node.config** 对象：

```
$ oc edit nodes.config/cluster
```

b. 编辑 **spec.cgroupMode** 参数：

node.config 对象示例

```
apiVersion: config.openshift.io/v2
kind: Node
metadata:
  annotations:
    include.release.openshift.io/ibm-cloud-managed: "true"
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
    release.openshift.io/create-only: "true"
  creationTimestamp: "2022-07-08T16:02:51Z"
  generation: 1
  name: cluster
  ownerReferences:
  - apiVersion: config.openshift.io/v2
    kind: ClusterVersion
    name: version
    uid: 36282574-bf9f-409e-a6cd-3032939293eb
  resourceVersion: "1865"
  uid: 0c0f7a4c-4307-4187-b591-6155695ac85b
spec:
  cgroupMode: "v1" (1)
...
```

(1) 指定 **v1** 启用 cgroup v1，或 **v2** 启用 cgroup v2。

验证

1. 检查机器配置以查看是否添加了新的机器配置：

```
$ oc get mc
```

输出示例

```
NAME                                               GENERATEDBYCONTROLLER
IGNITIONVERSION   AGE
00-master                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
00-worker                                          52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0
33m
01-master-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-master-kubelet                                      52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-container-runtime                              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
01-worker-kubelet                                      52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
```

```
3.2.0             33m
97-master-generated-kubelet                 52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-worker-generated-kubelet                 52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-generated-registries              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-master-ssh                                                    3.2.0             40m
99-worker-generated-registries              52dd3ba6a9a527fc3ab42afac8d12b693534c8c9
3.2.0             33m
99-worker-ssh                                                    3.2.0             40m
rendered-master-23d4317815a5f854bd3553d689cfe2e9
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             10s (1)
rendered-master-23e785de7587df95a4b517e0647e5ab7
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-5d596d9293ca3ea80c896a1191735bb1
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             33m
rendered-worker-dcc7f1b92892d34db74d6832bcc9ccd4
52dd3ba6a9a527fc3ab42afac8d12b693534c8c9   3.2.0             10s
```

(1) 创建新机器配置，如预期一样。

2. 检查新的 **kernelArguments** 是否已添加到新机器配置中：

```
$ oc describe mc <name>
```

cgroup v2 的输出示例

```
apiVersion: machineconfiguration.openshift.io/v2
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 05-worker-kernelarg-selinuxpermissive
spec:
  kernelArguments:
    systemd_unified_cgroup_hierarchy=1 (1)
    cgroup_no_v1="all" (2)
    psi=1 (3)
```

(1) 在 systemd 中启用 cgroup v2。

(2) 禁用 cgroup v1。

(3) 启用 Linux 保证工作站信息 (PSI) 功能。

cgroup v1 的输出示例

```
apiVersion: machineconfiguration.openshift.io/v2
kind: MachineConfig
metadata:
  labels:
```

```
    machineconfiguration.openshift.io/role: worker
  name: 05-worker-kernelarg-selinuxpermissive
spec:
  kernelArguments:
    systemd.unified_cgroup_hierarchy=0 (1)
    systemd.legacy_systemd_cgroup_controller=1 (2)
```

1. 禁用 cgroup v2。
2. 在 systemd 中启用 cgroup v1。

3. 检查节点以查看是否禁用了在节点上调度。这表示要应用更改：

```
$ oc get nodes
```

输出示例

```
NAME                                STATUS                     ROLES    AGE   VERSION
ci-ln-fm1qnwt-72292-99kt6-master-0  Ready,SchedulingDisabled   master   58m
v1.29.4
ci-ln-fm1qnwt-72292-99kt6-master-1  Ready                      master   58m   v1.29.4
ci-ln-fm1qnwt-72292-99kt6-master-2  Ready                      master   58m   v1.29.4
ci-ln-fm1qnwt-72292-99kt6-worker-a-h5gt4   Ready,SchedulingDisabled   worker   48m
v1.29.4
ci-ln-fm1qnwt-72292-99kt6-worker-b-7vtmd   Ready                      worker   48m   v1.29.4
ci-ln-fm1qnwt-72292-99kt6-worker-c-rhzkv   Ready                      worker   48m   v1.29.4
```

4. 节点返回 **Ready** 状态后，为该节点启动 debug 会话：

```
$ oc debug node/<node_name>
```

5. 将 **/host** 设置为 debug shell 中的根目录：

```
sh-4.4# chroot /host
```

6. 检查节点上是否存在 **sys/fs/cgroup/cgroup2fs** 或 **sys/fs/cgroup/tmpfs** 文件：

```
$ stat -c %T -f /sys/fs/cgroup
```

cgroup v2 的输出示例

```
cgroup2fs
```

cgroup v1 的输出示例

```
tmpfs
```

其他资源

- OpenShift Container Platform 安装概述

8.7. 使用功能门启用功能

作为管理员，您可以使用功能门启用不是默认功能集合中的功能。

8.7.1. 了解功能门

您可以使用 **FeatureGate** 自定义资源（CR）在集群中启用特定的功能集。功能集是 OpenShift Container Platform 功能的集合，默认情况下不启用。

您可以使用 **FeatureGate** CR 激活以下功能集：

- **TechPreviewNoUpgrade**.这个功能集是当前技术预览功能的子集。此功能集允许您在测试集群中启用这些技术预览功能，您可以在测试集群中完全测试它们，同时保留生产集群中禁用的功能。

警告

在集群中启用 **TechPreviewNoUpgrade** 功能集无法撤消，并会阻止次版本更新。您不应该在生产环境集群中启用此功能。

此功能集启用了以下技术预览功能：

- 外部云供应商。为 vSphere、AWS、Azure 和 GCP 上的集群启用外部云供应商的支持。对 OpenStack 的支持是 GA。这是一个内部功能，大多数用户不需要与之交互。(**ExternalCloudProvider**)
- OpenShift 构建中的共享资源 CSI 驱动程序。启用 Container Storage Interface (CSI)。(**CSIDriverSharedResource**)
- 节点上的交换内存。根据每个节点为 OpenShift Container Platform 工作负载启用交换内存使用。(**NodeSwap**)
- OpenStack Machine API 提供程序。此最低要求无效，计划在以后的发行版本中从此功能集中删除。(**MachineAPIProviderOpenStack**)
- Insights Operator。启用 **InsightsDataGather** CRD，允许用户配置一些 Insights 数据收集选项。功能集还启用了 **DataGather** CRD，允许用户按需运行 Insights 数据收集。(**InsightsConfigAPI**)
- 递归默认存储类.如果 PVC 创建时没有默认存储类(**RetroactiveDefaultStorageClass**) 启用 OpenShift Container Platform 会主动地将默认存储类分配给 PVC。
- 动态资源分配 API。启用一个新的 API 在 pod 和容器间请求和共享资源。这是一个内部功能，大多数用户不需要与之交互。(**DynamicResourceAllocation**)
- Pod 安全准入强制。为 pod 安全准入启用受限强制模式。如果 pod 违反了 pod 安全标准，它们会被拒绝，而不是仅记录警告信息。(**OpenShiftPodSecurityAdmission**)
- StatefulSet pod 可用性升级限制。允许用户定义在更新过程中不可用的 statefulset pod 的最大数量，这可以减少应用程序停机时间。(**MaxUnavailableStatefulSet**)
- Admin Network 策略和 Baseline Admin Network 策略。在运行 OVN-Kubernetes CNI 插件的

集群中，启用 **AdminNetworkPolicy** 和 **BaselineAdminNetworkPolicy** 资源，它们是 Network Policy V2 API 的一部分。集群管理员可以在创建命名空间前为整个集群应用集群范围的策略和保护。网络管理员可以通过强制无法覆盖的网络流量控制来保护集群。如果需要，网络管理员可以实施可选的基准网络流量控制，这些流量可以被集群中的用户覆盖。目前，这些 API 仅支持集群内流量的策略。(**AdminNetworkPolicy**)

- **MatchConditions** 是一个必须满足的条件列表，只要在满足这些条件时才向此 webhook 发送请求。匹配条件过滤请求，它们已与 rules、namespaceSelector 和 objectSelector 匹配。一个空的 **matchConditions** 列表，匹配所有请求。(**admissionWebhookMatchConditions**)
- **gcpLabelsTags**
- **vSphereStaticIPs**
- **routeExternalCertificate**
- **automatedEtcdBackup**
- **gcpClusterHostedDNS**
- **vSphereControlPlaneMachineset**
- **dnsNameResolver**
- **machineConfigNodes**
- **metricsServer**
- **installAlternateInfrastructureAWS**
- **sdnLiveMigration**
- **mixedCPUsAllocation**
- **managedBootImages**
- **onClusterBuild**
- **signatureStores**
- **DisableKubeletCloudCredentialProviders**
- **BareMetalLoadBalancer**
- **ClusterAPIInstallAWS**
- **ClusterAPIInstallNutanix**
- **ClusterAPIInstallOpenStack**
- **ClusterAPIInstallVSphere**
- **HardwareSpeed**
- **KMSv1**
- **NetworkDiagnosticsConfig**

- **VSphereDriverConfiguration**
- **ExternalOIDC**
- **ChunkSizeMiB**
- **ClusterAPIInstallGCP**
- **ClusterAPIInstallPowerVS**
- **EtcdBackendQuota**
- **Example**
- **ExternalRouteCertificate**
- **ImagePolicy**
- **InsightsConfig**
- **InsightsOnDemandDataGather**
- **MetricsCollectionProfiles**
- **NewOLM**
- **NodeDisruptionPolicy**
- **PinnedImages**
- **PlatformOperators**
- **ServiceAccountTokenNodeBinding**
- **ServiceAccountTokenNodeBindingValidation**
- **ServiceAccountTokenPodNodeInfo**
- **TranslateStreamCloseWebsocketRequests**
- **UpgradeStatus**
- **VSphereMultiVCenters**
- **VolumeGroupSnapshot**

有关 **TechPreviewNoUpgrade** 功能门激活的功能的更多信息，请参阅以下主题：

- 共享资源 CSI 驱动程序和 OpenShift 构建中的 CSI 卷
- CSI inline 临时卷
- 节点上的交换内存
- 使用 Cluster API 管理机器
- 禁用 Insights Operator 收集操作

- 启用 Insights Operator 收集操作
- 运行 Insights Operator 收集操作
- 管理默认存储类
- Pod 安全准入强制。

8.7.2. 在安装时启用功能集

在部署集群前，您可以编辑 **install-config.yaml** 文件来为集群中的所有节点启用功能集。

先决条件

- 您有一个 **install-config.yaml** 文件。

流程

1. 使用 **featureSet** 参数指定您要启用的功能集的名称，如 **TechPreviewNoUpgrade** ：

警告

在集群中启用 **TechPreviewNoUpgrade** 功能集无法撤消，并会阻止次版本更新。您不应该在生产环境集群中启用此功能。

带有启用功能集的 install-config.yaml 文件示例

```
compute:
- hyperthreading: Enabled
  name: worker
  platform:
    aws:
      rootVolume:
        iops: 2000
        size: 500
        type: io1
      metadataService:
        authentication: Optional
      type: c5.4xlarge
      zones:
      - us-west-2c
  replicas: 3
featureSet: TechPreviewNoUpgrade
```

2. 保存文件并在使用安装程序部署集群时引用。

验证

您可以在节点返回就绪状态后查看节点上的 **kubelet.conf** 文件来验证是否启用了功能门。

1. 从 Web 控制台中的 **Administrator** 视角，进入到 **Compute → Nodes**。
2. 选择一个节点。
3. 在 **Node 详情**页面中，点 **Terminal**。
4. 在终端窗口中，将根目录改为 **/host** ：

```
sh-4.2# chroot /host
```

5. 查看 **kubelet.conf** 文件：

```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

输出示例

```
# ...
featureGates:
  InsightsOperatorPullingSCA: true,
  LegacyNodeRoleBehavior: false
# ...
```

在集群中启用列为 **true** 的功能。

注意

列出的功能因 OpenShift Container Platform 版本的不同而有所不同。

8.7.3. 使用 Web 控制台启用功能集

您可以通过编辑 **FeatureGate** 自定义资源（CR）来使用 OpenShift Container Platform Web 控制台为集群中的所有节点启用功能集。

流程

启用功能集：

1. 在 OpenShift Container Platform web 控制台中，切换到 **Administration → Custom Resource Definitions** 页面。
2. 在 **Custom Resource Definitions**页面中，点击 **FeatureGate**。
3. 在 **Custom Resource Definition Details**页面中，点 **Instances** 选项卡。
4. 点 **集群** 功能门，然后点 **YAML** 选项卡。
5. 编辑**集群**实例以添加特定的功能集：

警告

在集群中启用 **TechPreviewNoUpgrade** 功能集无法撤消，并会阻止次版本更新。您不应该在生产环境集群中启用此功能。

功能门自定义资源示例

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster (1)
# ...
spec:
  featureSet: TechPreviewNoUpgrade (2)
```

1. **FeatureGate** CR 的名称必须是 **cluster**。
2. 添加要启用的功能集：
   - **TechPreviewNoUpgrade** 启用了特定的技术预览功能。

保存更改后，会创建新的机器配置，然后更新机器配置池，并在应用更改时在每个节点上调度。

验证

您可以在节点返回就绪状态后查看节点上的 **kubelet.conf** 文件来验证是否启用了功能门。

1. 从 Web 控制台中的 **Administrator** 视角，进入到 **Compute → Nodes**。
2. 选择一个节点。
3. 在 **Node 详情**页面中，点 **Terminal**。
4. 在终端窗口中，将根目录改为 **/host** ：

```
sh-4.2# chroot /host
```

5. 查看 **kubelet.conf** 文件：

```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

输出示例

```
# ...
featureGates:
  InsightsOperatorPullingSCA: true,
  LegacyNodeRoleBehavior: false
# ...
```

在集群中启用列为 **true** 的功能。

注意

列出的功能因 OpenShift Container Platform 版本的不同而有所不同。

8.7.4. 使用 CLI 启用功能集

您可以通过编辑 **FeatureGate** 自定义资源（CR）来使用 OpenShift CLI（**oc**）为集群中的所有节点启用功能集。

先决条件

- 已安装 OpenShift CLI（**oc**）。

流程

启用功能集：

1. 编辑名为 **cluster** 的 **FeatureGate** CR：

```
$ oc edit featuregate cluster
```

警告

在集群中启用 **TechPreviewNoUpgrade** 功能集无法撤消，并会阻止次版本更新。您不应该在生产环境集群中启用此功能。

FeatureGate 自定义资源示例

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster (1)
# ...
spec:
  featureSet: TechPreviewNoUpgrade (2)
```

(1) **FeatureGate** CR 的名称必须是 **cluster**。

(2) 添加要启用的功能集：

- **TechPreviewNoUpgrade** 启用了特定的技术预览功能。

保存更改后，会创建新的机器配置，然后更新机器配置池，并在应用更改时在每个节点上调度。

验证

您可以在节点返回就绪状态后查看节点上的 **kubelet.conf** 文件来验证是否启用了功能门。

1. 从 Web 控制台中的 **Administrator** 视角，进入到 **Compute → Nodes**。
2. 选择一个节点。
3. 在 **Node 详情**页面中，点 **Terminal**。
4. 在终端窗口中，将根目录改为 **/host** ：

```
sh-4.2# chroot /host
```

5. 查看 **kubelet.conf** 文件：

```
sh-4.2# cat /etc/kubernetes/kubelet.conf
```

输出示例

```
# ...
featureGates:
  InsightsOperatorPullingSCA: true,
  LegacyNodeRoleBehavior: false
# ...
```

在集群中启用列为 **true** 的功能。

注意

列出的功能因 OpenShift Container Platform 版本的不同而有所不同。

8.8. 使用 WORKER 延迟配置集提高高延迟环境中的集群稳定性

如果集群管理员为平台验证执行了延迟测试，他们可以发现需要调整集群的操作，以确保高延迟的情况的稳定性。集群管理员只需要更改一个参数，该参数记录在一个文件中，它控制了 Supervisory 进程读取状态并解释集群的运行状况的四个参数。仅更改一个参数可以以方便、可支持的方式提供集群调整。

Kubelet 进程提供监控集群运行状况的起点。**Kubelet** 为 OpenShift Container Platform 集群中的所有节点设置状态值。Kubernetes Controller Manager (**kube controller**) 默认每 10 秒读取状态值。如果 **kube 控制器**无法读取节点状态值，它会在配置的时间后丢失与该节点联系。默认行为是：

1. control plane 上的节点控制器将节点健康状况更新为 **Unhealthy**，并奖节点 **Ready** 的条件标记为 'Unknown'。
2. 因此，调度程序会停止将 pod 调度到该节点。
3. Node Lifecycle Controller 添加了一个 **node.kubernetes.io/unreachable** 污点，对节点具有 **NoExecute** 效果，默认在五分钟后调度节点上的任何 pod 进行驱除。

如果您的网络容易出现延迟问题，尤其是在网络边缘中有节点时，此行为可能会造成问题。在某些情况下，Kubernetes Controller Manager 可能会因为网络延迟而从健康的节点接收更新。**Kubelet** 会从节点中驱除 pod，即使节点处于健康状态。

要避免这个问题，您可以使用 *worker 延迟配置集*调整 **kubelet** 和 Kubernetes Controller Manager 在执行操作前等待状态更新的频率。如果在控制平面和 worker 节点间存在网络延迟，worker 节点没有处于最近状态，这个调整有助于集群可以正常工作。

这些 worker 延迟配置集包含预定义的三组参数，它们带有经过仔细调优的值，以控制集群对增加的延迟进行适当地响应。用户不需要手动进行实验以查找最佳值。

您可在安装集群时配置 worker 延迟配置集，或当您发现集群网络中的延迟增加时。

8.8.1. 了解 worker 延迟配置集

worker 延迟配置集带有四个不同的、包括经过仔细调优的参数的类别。实现这些值的四个参数是 **node-status-update-frequency**、**node-monitor-grace-period**、**default-not-ready-toleration-seconds** 和 **default-unreachable-toleration-seconds**。这些参数可让您使用这些值来控制集群对延迟问题的响应，而无需手动确定最佳值。

重要

不支持手动设置这些参数。参数设置不正确会影响集群的稳定性。

所有 worker 延迟配置集配置以下参数：

node-status-update-frequency

指定 kubelet 将节点状态发布到 API 服务器的频率。

node-monitor-grace-period

指定 Kubernetes Controller Manager 在节点不健康前等待更新的时间（以秒为单位），并将 **node.kubernetes.io/not-ready** 或 **node.kubernetes.io/unreachable** 污点添加到节点。

default-not-ready-toleration-seconds

指定在标记节点不健康后，Kube API Server Operator 在从该节点驱除 pod 前等待的时间（以秒为单位）。

default-unreachable-toleration-seconds

指定在节点无法访问后，Kube API Server Operator 在从该节点驱除 pod 前等待的时间（以秒为单位）。

以下 Operator 监控 worker 延迟配置集的更改并相应地响应：

- Machine Config Operator (MCO) 更新 worker 节点上的 **node-status-update-frequency** 参数。
- Kubernetes Controller Manager 更新 control plane 节点上的 **node-monitor-grace-period** 参数。
- Kubernetes API Server Operator 更新 control plane 节点上的 **default-not-ready-toleration-seconds** 和 **default-unreachable-toleration-seconds** 参数。

虽然默认配置在大多数情况下可以正常工作，但 OpenShift Container Platform 会为网络遇到比通常更高的延迟的情况提供两个其他 worker 延迟配置集。以下部分描述了三个 worker 延迟配置集：

默认 worker 延迟配置集

使用 **Default** 配置集时，每个 **Kubelet** 每 10 秒更新其状态(**node-status-update-frequency**)。**Kube Controller Manager** 每 5 秒检查 **Kubelet** 的状态(**node-monitor-grace-period**)。
在认为 **Kubelet** 不健康前，Kubernetes Controller Manager 会等待 40 秒以获取来自 **Kubelet** 的状态更新。如果没有可用于 Kubernetes Controller Manager 的使用状态，它会使用 **node.kubernetes.io/not-ready** 或 **node.kubernetes.io/unreachable** 污点标记节点，并驱除该节点上的 pod。

如果该节点上的 pod 具有 **NoExecute** 污点，则 pod 会根据 **tolerationSeconds** 运行。如果 pod 没有污点，它将在 300 秒内被驱除(**default-not-ready-toleration-seconds** 和 **Kube API Server**的 **default-unreachable-toleration-seconds** 设置)。

profile	组件	参数	值
Default（默认）	kubelet	**node-status-update-frequency**	10s
	kubelet Controller Manager	**node-monitor-grace-period**	40s
	Kubernetes API Server Operator	**default-not-ready-toleration-seconds**	300s
	Kubernetes API Server Operator	**default-unreachable-toleration-seconds**	300s

中型 worker 延迟配置集

如果网络延迟比通常稍高，则使用 **MediumUpdateAverageReaction** 配置集。**MediumUpdateAverageReaction** 配置集减少了 kubelet 更新频率为 20 秒，并将 Kubernetes Controller Manager 等待这些更新的时间更改为 2 分钟。该节点上的 pod 驱除周期会减少到 60 秒。如果 pod 具有 **tolerationSeconds** 参数，则驱除会等待该参数指定的周期。

Kubernetes Controller Manager 会先等待 2 分钟时间，才会认为节点不健康。另一分钟后，驱除过程会启动。

profile	组件	参数	值
MediumUpdateAverageReaction	kubelet	**node-status-update-frequency**	20s
	kubelet Controller Manager	**node-monitor-grace-period**	2m
	Kubernetes API Server Operator	**default-not-ready-toleration-seconds**	60s
	Kubernetes API Server Operator	**default-unreachable-toleration-seconds**	60s

低 worker 延迟配置集

如果网络延迟非常高，请使用 **LowUpdateSlowReaction** 配置集。

LowUpdateSlowReaction 配置集将 kubelet 更新频率减少为 1 分钟，并将 Kubernetes Controller Manager 等待这些更新的时间更改为 5 分钟。该节点上的 pod 驱除周期会减少到 60 秒。如果 pod 具有 **tolerationSeconds** 参数，则驱除会等待该参数指定的周期。

Kubernetes Controller Manager 在认为节点不健康前会等待 5 分钟。另一分钟后，驱除过程会启动。

profile	组件	参数	值
LowUpdateSlowReaction	kubelet	**node-status-update-frequency**	1m
	kubelet Controller Manager	**node-monitor-grace-period**	5m
	Kubernetes API Server Operator	**default-not-ready-toleration-seconds**	60s
	Kubernetes API Server Operator	**default-unreachable-toleration-seconds**	60s

8.8.2. 使用和更改 worker 延迟配置集

要更改 worker 延迟配置集以处理网络延迟，请编辑 **node.config** 对象以添加配置集的名称。当延迟增加或减少时，您可以随时更改配置集。

您必须一次移动一个 worker 延迟配置集。例如，您无法直接从 **Default** 配置集移到 **LowUpdateSlowReaction** worker 延迟配置集。您必须首先从 **Default** worker 延迟配置集移到 **MediumUpdateAverageReaction** 配置集，然后再移到 **LowUpdateSlowReaction**。同样，当返回到 **Default** 配置集时，您必须首先从低配置集移到中配置集，然后移到 **Default**。

注意

您还可以在安装 OpenShift Container Platform 集群时配置 worker 延迟配置集。

流程

将默认的 worker 延迟配置集改为：

1. 中 worke worker 延迟配置集：
   a. 编辑 **node.config** 对象：

```
$ oc edit nodes.config/cluster
```

   b. 添加 **spec.workerLatencyProfile: MediumUpdateAverageReaction** ：

   node.config 对象示例

```
apiVersion: config.openshift.io/v1
kind: Node
metadata:
  annotations:
    include.release.openshift.io/ibm-cloud-managed: "true"
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
    release.openshift.io/create-only: "true"
  creationTimestamp: "2022-07-08T16:02:51Z"
  generation: 1
  name: cluster
  ownerReferences:
  - apiVersion: config.openshift.io/v1
    kind: ClusterVersion
    name: version
    uid: 36282574-bf9f-409e-a6cd-3032939293eb
  resourceVersion: "1865"
  uid: 0c0f7a4c-4307-4187-b591-6155695ac85b
spec:
  workerLatencyProfile: MediumUpdateAverageReaction (1)

# ...
```

(1) 指定中 worker 延迟策略。

随着更改被应用，每个 worker 节点上的调度都会被禁用。

2. 可选：改为低 worker 延迟配置集：

   a. 编辑 **node.config** 对象：

   ```
   $ oc edit nodes.config/cluster
   ```

   b. 将 **spec.workerLatencyProfile** 值更改为 **LowUpdateSlowReaction** ：

   node.config 对象示例

   ```
   apiVersion: config.openshift.io/v1
   kind: Node
   metadata:
     annotations:
       include.release.openshift.io/ibm-cloud-managed: "true"
       include.release.openshift.io/self-managed-high-availability: "true"
       include.release.openshift.io/single-node-developer: "true"
       release.openshift.io/create-only: "true"
     creationTimestamp: "2022-07-08T16:02:51Z"
     generation: 1
     name: cluster
     ownerReferences:
     - apiVersion: config.openshift.io/v1
       kind: ClusterVersion
       name: version
       uid: 36282574-bf9f-409e-a6cd-3032939293eb
     resourceVersion: "1865"
   ```

```
  uid: 0c0f7a4c-4307-4187-b591-6155695ac85b
spec:
  workerLatencyProfile: LowUpdateSlowReaction (1)

# ...
```

(1) 指定使用低 worker 延迟策略。

随着更改被应用，每个 worker 节点上的调度都会被禁用。

验证

- 当所有节点都返回到 **Ready** 条件时，您可以使用以下命令查看 Kubernetes Controller Manager 以确保应用它：

```
$ oc get KubeControllerManager -o yaml | grep -i workerlatency -A 5 -B 5
```

输出示例

```
# ...
    - lastTransitionTime: "2022-07-11T19:47:10Z"
      reason: ProfileUpdated
      status: "False"
      type: WorkerLatencyProfileProgressing
    - lastTransitionTime: "2022-07-11T19:47:10Z" (1)
      message: all static pod revision(s) have updated latency profile
      reason: ProfileUpdated
      status: "True"
      type: WorkerLatencyProfileComplete
    - lastTransitionTime: "2022-07-11T19:20:11Z"
      reason: AsExpected
      status: "False"
      type: WorkerLatencyProfileDegraded
    - lastTransitionTime: "2022-07-11T19:20:36Z"
      status: "False"
# ...
```

(1) 指定配置集被应用并激活。

要将中配置集改为默认，或将默认改为中，编辑 **node.config** 对象，并将 **spec.workerLatencyProfile** 参数设置为适当的值。

第 9 章 网络边缘上的远程 WORKER 节点

9.1. 在网络边缘使用远程 WORKER 节点

您可以使用位于网络边缘的节点来配置 OpenShift Container Platform 集群。在本主题中，它们称为*远程 worker 节点*。带有远程 worker 节点的典型集群合并了内部 master 和 worker 节点，其他位置的 worker 节点会连接到集群。本主题旨在提供使用远程 worker 节点的最佳实践指导，且不包含特定配置详情。

在使用具有远程 worker 节点的部署模式时，不同行业有不同的用例，如电信、零售、制造企业和政府。例如，您可以通过将远程 worker 节点合并到 Kubernetes 区域来隔离项目和工作负载。

但是，具有远程 worker 节点可能会带来更高的延迟，以及网络连接间丢失的问题。使用带有远程 worker 节点的集群会有一些挑战，包括：

- **网络隔离**：OpenShift Container Platform control plane 和远程 worker 节点必须可以相互通信。由于 control plane 和远程 worker 节点之间很远，网络问题可能会阻止这种通信。如需有关 OpenShift Container Platform 如何响应网络分离以及减少对集群的影响的信息，请参阅使用远程 worker 节点进行网络隔离。

- **电源中断**：因为 control plane 和远程 worker 节点位于不同的位置，因此在远程位置或在两个位置之间的任意点出现停机会给集群造成负面影响。如需了解 OpenShift Container Platform 如何响应节点丢失以及减少对集群的影响的信息，请参阅远程 worker 节点的电源丢失。

- **延迟或临时减少吞吐量**:与任何网络一样，集群和远程 worker 节点之间的网络状况改变都会对集群有负面影响。OpenShift Container Platform 提供多个 *worker 延迟配置集*，供您控制集群的反应延迟问题。

在规划使用远程 worker 节点的集群时，请注意以下限制：

- OpenShift Container Platform 不支持使用与内部集群所使用的不同云供应商的远程 worker 节点。

- 因为系统和环境中的问题（如特定类型的内存不在另一区中可用），将工作负载从一个 Kubernetes 区域移动到不同的 Kubernetes 区可能会有问题。

- 使用代理和防火墙可能会遇到本文档所涉及到的范围以外的限制。有关如何解决这些限制，如 配置防火墙，请参阅相关的 OpenShift Container Platform 文档。

- 您需要配置和维护 control plane 和网络边缘节点之间的 L2/L3 级别网络连接。

9.1.1. 添加远程 worker 节点

在集群中添加远程 worker 节点涉及一些额外的注意事项。

- 您必须确保路由或默认网关被放置，以便在 control plane 和每个远程 worker 节点之间路由流量。

- 您必须将 Ingress VIP 放在 control plane 上。

- 使用用户置备的基础架构添加远程 worker 节点与添加其他 worker 节点相同。

- 要在安装时将远程 worker 节点添加到安装程序置备的集群中，请在安装前在 **install-config.yaml** 文件中指定每个 worker 节点的子网。DHCP 服务器不需要额外的设置。您必须使用虚拟介质，因为远程 worker 节点无法访问本地置备网络。

- 要将远程 worker 节点添加到使用 provisioning 网络部署的安装程序置备的集群中，请确保在

install-config.yaml 文件中将 **virtualMediaViaExternalNetwork** 标志设置为 **true**，以便它将使用虚拟介质添加节点。远程 worker 节点无法访问本地置备网络。它们必须使用虚拟介质而不是 PXE 部署。另外，为每个远程 worker 节点组和 DHCP 服务器中的 control plane 节点指定每个子网。

其他资源

- 在子网间建立通信
- 为子网配置主机网络接口
- 配置要在 control plane 上运行的网络组件

9.1.2. 使用远程 worker 节点进行网络隔离

所有节点每 10 秒向 OpenShift Container Platform 集群中的 Kubernetes Controller Manager Operator（kube 控制器）发送 heartbeat。如果集群没有从节点获得 heartbeat,OpenShift Container Platform 会使用几个默认机制进行响应。

OpenShift Container Platform 旨在可以正确处理网络分区和其他中断问题的出现。您可以缓解一些常见中断的影响，如软件升级中断、网络分割和路由问题。缓解策略包括确保远程 worker 节点上的 pod 请求正确的 CPU 和内存资源量、配置适当的复制策略、使用跨区冗余以及在工作负载中使用 Pod Disruption Tables。

如果 kube 控制器在经过了配置的时间后无法访问节点，则 control plane 上的节点控制器会将节点健康状况更新为 **Unhealthy**，并将节点 **Ready** 条件标记为 **Unknown**。因此，调度程序会停止将 pod 调度到该节点。内部节点控制器添加了 **node.kubernetes.io/unreachable** 污点，对节点具有 **NoExecute** 效果，默认情况下，在五分钟后调度节点上的 pod 进行驱除。

如果工作负载控制器（如 **Deployment** 对象或 **StatefulSet** 对象）将流量定向到不健康节点上的 pod，而其他节点也可以访问集群，OpenShift Container Platform 会从节点上的 pod 路由流量。无法访问集群的节点不会使用新的流量路由进行更新。因此，这些节点上的工作负载可能会继续尝试访问不健康的节点。

您可以通过以下方法降低连接丢失的影响：

- 使用守护进程集创建容许污点的 pod
- 使用在节点停机时自动重启的静态 pod
- 使用 Kubernetes 区域来控制 pod 驱除
- 配置 pod 容限来延迟或避免 pod 驱除
- 配置 kubelet 以控制它在将节点标记为不健康的时间。

有关在带有远程 worker 节点的集群中使用这些对象的更多信息，请参阅关于远程 worker 节点策略。

9.1.3. 远程 worker 节点上的电源丢失

如果远程 worker 节点断电或重启不可用，OpenShift Container Platform 会使用几个默认机制进行响应。

如果 Kubernetes Controller Manager Operator（kube controller）在配置的时间后无法访问节点，control plane 会将节点健康状况更新为 **Unhealthy**，并将节点 **Ready** 条件标记为 **Unknown**。因此，调度程序会停止将 pod 调度到该节点。内部节点控制器添加了 **node.kubernetes.io/unreachable** 污点，对节点具有 **NoExecute** 效果，默认情况下，在五分钟后调度节点上的 pod 进行驱除。

在节点上当节点恢复电源并与 control plane 重新连接时， pod 必须重启。

注意

如果您希望 pod 重启后立即重启，请使用静态 pod。

节点重启后，kubelet 还会重启并尝试重启节点上调度的 pod。如果到 control plane 的连接时间超过默认的五分钟，则 control plane 无法更新节点健康状况并移除 **node.kubernetes.io/unreachable** 污点。在节点上，kubelet 会终止任何正在运行的 pod。当这些条件被清除后，调度程序就可以开始将 pod 调度到该节点。

您可以通过以下方法减轻电源损失的影响：

- 使用守护进程集创建容许污点的 pod
- 使用与节点自动重启的静态 pod
- 配置 pod 容限以延迟或避免 pod 驱除
- 配置 kubelet 以控制节点控制器何时将节点标记为不健康的时间。

有关在带有远程 worker 节点的集群中使用这些对象的更多信息，请参阅关于远程 worker 节点策略。

9.1.4. 远程 worker 吞吐量延迟或临时减少

如果集群管理员为平台验证执行了延迟测试，他们可以发现需要调整集群的操作，以确保高延迟的情况的稳定性。集群管理员只需要更改一个参数，该参数记录在一个文件中，它控制了 Supervisory 进程读取状态并解释集群的运行状况的四个参数。仅更改一个参数可以以方便、可支持的方式提供集群调整。

Kubelet 进程提供监控集群运行状况的起点。**Kubelet** 为 OpenShift Container Platform 集群中的所有节点设置状态值。Kubernetes Controller Manager (**kube controller**) 默认每 10 秒读取状态值。如果 **kube 控制器**无法读取节点状态值，它会在配置的时间后丢失与该节点联系。默认行为是：

1. control plane 上的节点控制器将节点健康状况更新为 **Unhealthy**，并奖节点 **Ready** 的条件标记为 'Unknown'。
2. 因此，调度程序会停止将 pod 调度到该节点。
3. Node Lifecycle Controller 添加了一个 **node.kubernetes.io/unreachable** 污点，对节点具有 **NoExecute** 效果，默认在五分钟后调度节点上的任何 pod 进行驱除。

如果您的网络容易出现延迟问题，尤其是在网络边缘中有节点时，此行为可能会造成问题。在某些情况下，Kubernetes Controller Manager 可能会因为网络延迟而从健康的节点接收更新。**Kubelet** 会从节点中驱除 pod，即使节点处于健康状态。

要避免这个问题，您可以使用 *worker 延迟配置集*调整 **kubelet** 和 Kubernetes Controller Manager 在执行操作前等待状态更新的频率。如果在控制平面和 worker 节点间存在网络延迟，worker 节点没有处于最近状态，这个调整有助于集群可以正常工作。

这些 worker 延迟配置集包含预定义的三组参数，它们带有经过仔细调优的值，以控制集群对增加的延迟进行适当地响应。用户不需要手动进行实验以查找最佳值。

您可在安装集群时配置 worker 延迟配置集，或当您发现集群网络中的延迟增加时。

其他资源

- 使用 worker 延迟配置集提高高延迟环境中的集群稳定性

9.1.5. 远程 worker 节点策略

如果您使用远程 worker 节点，请考虑使用哪个对象来运行应用程序。

建议根据所计划的在出现网络问题或电源丢失时需要进行的行为，使用守护进程集或静态 pod。另外，如果 control plane 无法访问远程 worker 节点，您可以使用 Kubernetes 区和容限来控制或避免 pod 驱除。

守护进程集

守护进程集是管理远程 worker 节点上的 pod 的最佳方法，理由如下：

- 守护进程集通常不需要重新调度。如果节点断开与集群的连接，节点上的 pod 将继续运行。OpenShift Container Platform 不更改守护进程设置 pod 的状态，并使 pod 保留为最新报告的状态。例如，如果守护进程集 pod 处于 **Running** 状态，当节点停止通信时，pod 会继续运行，并假定 OpenShift Container Platform 正在运行。

- 在默认情况下，守护进程集会被创建为带有对没有 **tolerationSeconds** 值的 **node.kubernetes.io/unreachable** 和 **node.kubernetes.io/not-ready** 污点的 **NoExecute** 容限。如果 control plane 无法访问节点，则守护进程集 pod 不会被驱除。例如：

  容限默认添加到守护进程集 pod

  ```
  tolerations:
    - key: node.kubernetes.io/not-ready
      operator: Exists
      effect: NoExecute
    - key: node.kubernetes.io/unreachable
      operator: Exists
      effect: NoExecute
    - key: node.kubernetes.io/disk-pressure
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/memory-pressure
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/pid-pressure
      operator: Exists
      effect: NoSchedule
    - key: node.kubernetes.io/unschedulable
      operator: Exists
      effect: NoSchedule
  ```

- 守护进程集可以使用标签来确保工作负载在匹配的 worker 节点上运行。

- 您可以使用 OpenShift Container Platform 服务端点来加载均衡守护进程设置 pod。

注意

如果 OpenShift Container Platform 无法访问该节点，守护进程集不会在节点重新引导后调度 pod。

静态 pod

如果您希望在一个节点重启后（例如发生了电源中断的情况）重启 pod，考虑使用静态 pod。节点上的 kubelet 会在节点重启时自动重启静态 pod。

注意

静态 pod 无法使用 secret 和配置映射。

Kubernetes 区域

Kubernetes 区域 可能会降低速率，或在某些情况下完全停止 pod 驱除。

当 control plane 无法访问节点时，节点控制器默认应用 **node.kubernetes.io/unreachable** 污点并驱除 pod，驱除率为每秒 0.1 个节点。但是，在使用 Kubernetes 区的集群中，pod 驱除行为会被改变。

如果区被完全破坏，区中的所有节点都具有 **False** 或 **Unknown** 的 **Ready** 条件，control plane 不会将 **node.kubernetes.io/unreachable** 污点应用到那个区的节点。

对于部分受破坏的区，超过 55% 的节点具有 **False** 或 **Unknown** 条件，pod 驱除率会降低为每秒 0.01 个节点。在较小集群（小于 50 个节点）中的节点不具有污点。您的集群必须具有超过三个区域才能使行为生效。

您可以通过应用节点规格中的 **topology.kubernetes.io/region** 标签将节点分配给特定区。

Kubernetes 区节点标签示例

```
kind: Node
apiVersion: v1
metadata:
  labels:
    topology.kubernetes.io/region=east
```

KubeletConfig 对象

您可以调整 kubelet 检查每个节点状态的时间长度。

要设置影响内部节点控制器何时标记具有 **Unhealthy** 或 **Unreachable** 状况的节点的时间间隔，创建一个包含 **node-status-update-frequency** 参数的 **KubeletConfig** 对象，以及 **node-status-report-frequency** 参数。

每个节点上的 kubelet 决定 **node-status-update-frequency** 设置定义的节点状态，并根据 **node-status-report-frequency** 设置向集群报告这个状态。默认情况下，kubelet 每 10 秒决定 pod 状态，并每分钟报告状态。但是，如果节点状态更改，kubelet 会立即报告到集群的更改。只有在启用了 Node Lease 功能门时，OpenShift Container Platform 才会使用 **node-status-report-frequency** 设置，这是 OpenShift Container Platform 集群的默认设置。如果禁用了 Node Lease 功能门，节点会根据 **node-status-update-frequency** 设置报告其状态。

kubelet 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: disable-cpu-units
spec:
  machineConfigPoolSelector:
    matchLabels:
```

```
    machineconfiguration.openshift.io/role: worker (1)
kubeletConfig:
  node-status-update-frequency: (2)
    - "10s"
  node-status-report-frequency: (3)
    - "1m"
```

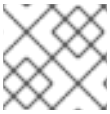
(1) 使用 **MachineConfig** 对象中的标签指定此 **KubeletConfig** 对象应用到的节点类型。

(2) 指定 kubelet 检查与此 **MachineConfig** 对象关联的节点状态的频率。默认值为 **10s**。如果更改此默认值，则 **node-status-report-frequency** 值将更改为相同的值。

(3) 指定 kubelet 报告与此 **MachineConfig** 对象关联的节点状态的频率。默认值为 **1m**。

node-status-update-frequency 参数可用于 **node-monitor-grace-period** 参数。

- **node-monitor-grace-period** 参数指定，如果控制器管理器未接收节点 heartbeat，OpenShift Container Platform 会在与 **MachineConfig** 对象关联的节点标记为 **Unhealthy** 后等待多久。该节点上的工作负载在此时间之后将继续运行。如果在 **node-monitor-grace-period** 过期后远程 worker 节点重新加入集群，pod 会继续运行。新的 pod 可以调度到该节点。**node-monitor-grace-period** 间隔为 **40s**。**node-status-update-frequency** 值必须小于 **node-monitor-grace-period** 值。



注意

不支持修改 **node-monitor-grace-period** 参数。

容限（Tolerations）

如果内部节点控制器添加了一个 **node.kubernetes.io/unreachable** 污点，它在无法访问时对节点有一个 **NoExecute** 的效果，则可以使用 pod 容限来减轻影响。

具有 **NoExecute** 效果的污点会影响节点上运行的 pod：

- 不容许污点的 Pod 会被放入队列进行驱除。
- 如果 Pod 容许污点，且没有在容限规格中指定 **tolerationSeconds** 值，则会永久保持绑定。
- 如果 Pod 容许污点，且指定了 **tolerationSeconds** 值，则会在指定的时间里保持绑定。在这个时间过后， pod 会被放入队列以驱除。

注意

除非明确设置容限，否则 Kubernetes 会自动为 **node.kubernetes.io/not-ready** 和 **node.kubernetes.io/unreachable** 添加带有 **tolerationSeconds=300** 的容限，这意味着如果检测到其中一个污点，pod 会保持绑定 5 分钟。

您可以通过把 pod 配置为使 **node.kubernetes.io/unreachable** 和 **node.kubernetes.io/not-ready** 污点有 **NoExecute** 的效果来延迟或避免 pod 驱除。

pod 规格中的容限示例

```
...
tolerations:
```

```
- key: "node.kubernetes.io/unreachable"
  operator: "Exists"
  effect: "NoExecute" 1
- key: "node.kubernetes.io/not-ready"
  operator: "Exists"
  effect: "NoExecute" 2
  tolerationSeconds: 600 3
...
```

1 没有 **tolerationSeconds** 的 **NoExecute** 效果可在 control plane 无法访问节点时永久保留 pod。

2 带有 **tolerationSeconds**: 600 的 **NoExecute** 效果允许在 control plane 将节点标记为 **Unhealthy** 时让 pod 再保持 10 分钟。

3 您可以指定自己的 **tolerationSeconds** 值。

其他类型的 OpenShift Container Platform 对象

您可以使用副本集、部署和复制控制器。当节点断开连接五分钟后，调度程序可将这些 pod 重新调度到其他节点上。重新调度到其他节点对于某些工作负载（如 REST API）来说是很有帮助的，管理员可以保证特定数量的 pod 正在运行并可以被访问。

注意

在使用远程 worker 节点时，如果远程 worker 节点旨在保留给特定功能，则不同节点上重新调度 pod 可能会是无法接受的。

有状态集不会在停机时重启。pod 处于 **terminating** 状态，直到 control plane 可以确认 pod 已被终止。

为了避免调度到一个无法访问同一类型的持久性存储的节点，OpenShift Container Platform 不允许在网络分离时将需要持久性卷的 pod 迁移到其他区。

其他资源

- 如需有关 Daemonesets 的更多信息，请参阅 DaemonSet。
- 如需有关污点和容限的更多信息，请参阅使用节点污点控制 pod 放置。
- 有关配置 **KubeletConfig** 对象的更多信息，请参阅创建 KubeletConfig CRD。
- 如需有关副本集的更多信息，请参阅 ReplicaSets。
- 如需有关部署的更多信息，请参阅 部署。
- 如需有关复制控制器的更多信息，请参阅 复制控制器。
- 如需有关控制器管理器的更多信息，请参阅 Kubernetes Controller Manager Operator。

第 10 章 单节点 OPENSHIFT 集群的 WORKER 节点

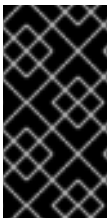
10.1. 将 WORKER 节点添加到单节点 OPENSHIFT 集群

单节点 OpenShift 集群可减少部署到单个主机的主机先决条件。这对在受限环境或网络边缘的环境中部署非常有用。但是，有时您需要为集群添加额外的容量，例如在电信和网络边缘场景中。在这种情况下，您可以将 worker 节点添加到单节点集群中。

注意

与多节点集群不同，默认所有入口流量都路由到单一 control-plane 节点，即使添加了额外的 worker 节点。

您可以通过几种方法将 worker 节点添加到单节点集群中。您可以使用 Red Hat OpenShift Cluster Manager 手动将 worker 节点添加到集群中，或者直接使用 Assisted Installer REST API。



重要

添加 worker 节点不会扩展集群 control plane，它不为集群提供高可用性。对于单节点 OpenShift 集群，通过故障转移到另一站点来处理高可用性。将 worker 节点添加到单节点 OpenShift 集群时，建议最多测试两个 worker 节点。超过推荐的 worker 节点数量可能会导致整体性能较低，包括集群失败。

注意

要添加 worker 节点，您必须有权访问 OpenShift Cluster Manager。当使用基于 Agent 的安装程序在断开连接的环境中安装集群时，不支持此方法。

10.1.1. 安装单节点 OpenShift worker 节点的要求

要安装单节点 OpenShift worker 节点，您必须处理以下要求：

- **管理主机：**您必须有一个计算机准备 ISO 和监控安装。
- **Production-grade server:** 安装单节点 OpenShift worker 节点需要有足够资源的服务器才能运行 OpenShift Container Platform 服务和生产环境工作负载。

表 10.1. 最低资源要求

profile	vCPU	memory	Storage
最小值	2 个 vCPU 内核	8GB RAM	100GB

注意

当未启用并发多线程(SMT)或超线程时，一个 vCPU 相当于一个物理内核。启用后，使用以下公式来计算对应的比率：

（每个内核的线程数 x 内核数）x 插槽数 = vCPU

使用虚拟介质引导时，服务器必须具有基板管理控制器(BMC)。

- **网络：** 如果服务器没有连接到可路由的网络，则 worker 节点服务器必须可以访问互联网或访问本地 registry。worker 节点服务器必须具有 DHCP 保留或静态 IP 地址，并可访问单节点 OpenShift 集群 Kubernetes API、入口路由和集群节点域名。您必须将 DNS 配置为将 IP 地址解析到单节点 OpenShift 集群以下每个完全限定域名 (FQDN)：

  表 10.2. 所需的 DNS 记录

用法	FQDN	描述
Kubernetes API	**api.<cluster_name>.<base_domain>**	添加 DNS A/AAAA 或 CNAME 记录。此记录必须由集群外的客户端解析。
内部 API	**api-int.<cluster_name>.<base_domain>**	在手动创建 ISO 时，添加 DNS A/AAAA 或 CNAME 记录。此记录必须由集群内的节点解析。
Ingress 路由	***.apps.<cluster_name>.<base_domain>**	添加以节点为目标的通配符 DNS A/AAAA 或 CNAME 记录。此记录必须由集群外的客户端解析。

  如果没有持久的 IP 地址，**apiserver** 和 **etcd** 之间的通信可能会失败。

其他资源

- 集群安装的最低资源要求
- 扩展集群的建议实践
- 用户置备的 DNS 要求
- 在 USB 驱动器中创建可引导 ISO 镜像
- 使用 Redfish API 从通过 HTTP 提供的 ISO 镜像引导
- 从集群中删除节点

10.1.2. 使用 Assisted Installer 和 OpenShift Cluster Manager 添加 worker 节点

您可以使用 Assisted Installer 将 worker 节点添加到 Red Hat OpenShift Cluster Manager 上创建的单节点 OpenShift 集群。

重要

只有在运行 OpenShift Container Platform 版本 4.11 并已启动的集群才支持将 worker 节点添加到单节点 OpenShift 集群。

先决条件

- 使用辅助安装程序访问安装的单节点 OpenShift 集群。

- 安装 OpenShift CLI (**oc**) 。
- 以具有 **cluster-admin** 特权的用户身份登录。
- 确保将 worker 节点添加到集群中的集群需要的所有 DNS 记录。

流程

1. 登录到 OpenShift Cluster Manager，并点您要添加 worker 节点的单节点集群。
2. 点 **Add hosts**，再下载新 worker 节点的发现 ISO，添加 SSH 公钥和配置集群范围的代理设置。
3. 使用发现 ISO 引导目标主机，并等待在控制台中发现主机。主机被发现后，开始安装。
4. 当安装继续进行时，安装会为 worker 节点生成待处理的证书签名请求 (CSR)。出现提示时，批准待处理的 CSR 以完成安装。
   当 worker 节点正常运行时，它会作为集群 web 控制台中的 worker 节点列出。

重要

新的 worker 节点将使用与原始集群相同的方法进行加密。

其他资源

- 用户置备的 DNS 要求
- 批准机器的证书签名请求

10.1.3. 使用 Assisted Installer API 添加 worker 节点

您可以使用 Assisted Installer REST API 将 worker 节点添加到单节点 OpenShift 集群。在添加 worker 节点前，您必须登录到 OpenShift Cluster Manager 并针对 API 进行身份验证。

10.1.3.1. 针对 Asssisted Installer REST API 进行身份验证

在使用 Assisted Installer REST API 之前，您必须使用您生成的 JSON Web 令牌 (JWT) 进行身份验证。

先决条件

- 以具有集群创建权限的用户身份登录 OpenShift Cluster Manager。
- 安装 **jq**。

流程

1. 登录到 OpenShift Cluster Manager 并复制您的 API 令牌。
2. 运行以下命令，使用复制的 API 令牌设置 **$OFFLINE_TOKEN** 变量：

```
$ export OFFLINE_TOKEN=<copied_api_token>
```

3. 使用之前设置的 **$OFFLINE_TOKEN** 变量来设置 **$JWT_TOKEN** 变量：

```
$ export JWT_TOKEN=$(
```

```
  curl \
  --silent \
  --header "Accept: application/json" \
  --header "Content-Type: application/x-www-form-urlencoded" \
  --data-urlencode "grant_type=refresh_token" \
  --data-urlencode "client_id=cloud-services" \
  --data-urlencode "refresh_token=${OFFLINE_TOKEN}" \
  "https://sso.redhat.com/auth/realms/redhat-external/protocol/openid-connect/token" \
  | jq --raw-output ".access_token"
)
```

注意

JWT 令牌仅会有效 15 分钟。

验证

- 可选：运行以下命令来检查您可以访问 API：

```
$ curl -s https://api.openshift.com/api/assisted-install/v2/component-versions -H
"Authorization: Bearer ${JWT_TOKEN}" | jq
```

输出示例

```
{
   "release_tag": "v2.5.1",
   "versions":
   {
      "assisted-installer": "registry.redhat.io/rhai-tech-preview/assisted-installer-rhel8:v1.0.0-
175",
      "assisted-installer-controller": "registry.redhat.io/rhai-tech-preview/assisted-installer-
reporter-rhel8:v1.0.0-223",
      "assisted-installer-service": "quay.io/app-sre/assisted-service:ac87f93",
      "discovery-agent": "registry.redhat.io/rhai-tech-preview/assisted-installer-agent-
rhel8:v1.0.0-156"
   }
}
```

10.1.3.2. 使用 Assisted Installer REST API 添加 worker 节点

您可以使用 Assisted Installer REST API 将 worker 节点添加到集群中。

先决条件

- 安装 OpenShift Cluster Manager CLI (**ocm**)。
- 以具有集群创建权限的用户身份登录 OpenShift Cluster Manager。
- 安装 **jq**。
- 确保将 worker 节点添加到集群中的集群需要的所有 DNS 记录。

流程

1. 针对引导安装程序 REST API 进行身份验证，并为会话生成 JSON Web 令牌 (JWT)。生成的 JWT 令牌仅会有效 15 分钟。

2. 运行以下命令设置 **$API_URL** 变量：

```
$ export API_URL=<api_url> 1
```

1 将 **<api_url>** 替换为 Assisted Installer API URL，例如 **https://api.openshift.com**

3. 运行以下命令导入单节点 OpenShift 集群：

   a. 设置 **$OPENSHIFT_CLUSTER_ID** 变量。登录到集群并运行以下命令：

```
$ export OPENSHIFT_CLUSTER_ID=$(oc get clusterversion -o
jsonpath='{.items[].spec.clusterID}')
```

   b. 设置用于导入集群的 **$CLUSTER_REQUEST** 变量：

```
$ export CLUSTER_REQUEST=$(jq --null-input --arg openshift_cluster_id
"$OPENSHIFT_CLUSTER_ID" '{
  "api_vip_dnsname": "<api_vip>", 1
  "openshift_cluster_id": $openshift_cluster_id,
  "name": "<openshift_cluster_name>" 2
}')
```

1 将 **<api_vip>** 替换为集群 API 服务器的主机名。这可以是 API 服务器的 DNS 域，也可以是 worker 节点可访问的单一节点的 IP 地址。例如： **api.compute-1.example.com**。

2 将 **<openshift_cluster_name>** 替换为集群的纯文本名称。集群名称应与在第 1 天集群安装过程中设置的集群名称匹配。

   c. 导入集群并设置 **$CLUSTER_ID** 变量。运行以下命令:

```
$ CLUSTER_ID=$(curl "$API_URL/api/assisted-install/v2/clusters/import" -H
"Authorization: Bearer ${JWT_TOKEN}" -H 'accept: application/json' -H 'Content-Type:
application/json' \
  -d "$CLUSTER_REQUEST" | tee /dev/stderr | jq -r '.id')
```

4. 运行以下命令，为集群生成 **InfraEnv** 资源并设置 **$INFRA_ENV_ID** 变量：

   a. 从位于 console.redhat.com 的 Red Hat OpenShift Cluster Manager 下载 pull secret 文件。

   b. 设置 **$INFRA_ENV_REQUEST** 变量：

```
export INFRA_ENV_REQUEST=$(jq --null-input \
    --slurpfile pull_secret <path_to_pull_secret_file> \ 1
    --arg ssh_pub_key "$(cat <path_to_ssh_pub_key>)" \ 2
    --arg cluster_id "$CLUSTER_ID" '{
  "name": "<infraenv_name>", 3
  "pull_secret": $pull_secret[0] | tojson,
  "cluster_id": $cluster_id,
```

```
  "ssh_authorized_key": $ssh_pub_key,
  "image_type": "<iso_image_type>" 4
}')
```

1. 将 **<path_to_pull_secret_file>** 替换为在 console.redhat.com 上从 Red Hat OpenShift Cluster Manager 下载的 pull secret 的本地文件的路径。
2. 将 **<path_to_ssh_pub_key>** 替换为访问主机所需的公共 SSH 密钥的路径。如果没有设置这个值，则无法在发现模式下访问主机。
3. 将 **<infraenv_name>** 替换为 **InfraEnv** 资源的纯文本名称。
4. 将 **<iso_image_type>** 替换为 ISO 镜像类型，可以是 **full-iso** 或 **minimal-iso**。

c. 将 **$INFRA_ENV_REQUEST** 发布到 /v2/infra-envs API，并设置 **$INFRA_ENV_ID** 变量：

```
$ INFRA_ENV_ID=$(curl "$API_URL/api/assisted-install/v2/infra-envs" -H "Authorization: Bearer ${JWT_TOKEN}" -H 'accept: application/json' -H 'Content-Type: application/json' -d "$INFRA_ENV_REQUEST" | tee /dev/stderr | jq -r '.id')
```

5. 运行以下命令，获取集群 worker 节点的发现 ISO 的 URL：

```
$ curl -s "$API_URL/api/assisted-install/v2/infra-envs/$INFRA_ENV_ID" -H "Authorization: Bearer ${JWT_TOKEN}" | jq -r '.download_url'
```

输出示例

```
https://api.openshift.com/api/assisted-images/images/41b91e72-c33e-42ee-b80f-b5c5bbf6431a?arch=x86_64&image_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2NTYwMjYzNzEsInN1YiI6IjQxYjkxZTcyLWMzM2UtNDJlZS1iODBmLWI1YzViYmY2NDMxYSJ9.1EX_VGaMNejMhrAvVRBS7PDPIQtbOOc8LtG8OukE1a4&type=minimal-iso&version=$VERSION
```

6. 下载 ISO：

```
$ curl -L -s '<iso_url>' --output rhcos-live-minimal.iso 1
```

1. 将 **<iso_url>** 替换为上一步中的 ISO URL。

7. 从下载的 **rhcos-live-minimal.iso** 中引导新的 worker 主机。
8. 获取*没有*安装的集群中的主机列表。继续运行以下命令，直到新主机显示：

```
$ curl -s "$API_URL/api/assisted-install/v2/clusters/$CLUSTER_ID" -H "Authorization: Bearer ${JWT_TOKEN}" | jq -r '.hosts[] | select(.status != "installed").id'
```

输出示例

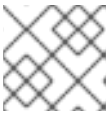
```
2294ba03-c264-4f11-ac08-2f1bb2f8c296
```

9. 为新的 worker 节点设置 **$HOST_ID** 变量，例如：

```
$ HOST_ID=<host_id> 1
```

1 将 **<host_id>** 替换为上一步中的主机 ID。

10. 运行以下命令检查主机是否已就绪：



注意

确保复制整个命令，包括完整的 **jq** 表达式。

```
$ curl -s $API_URL/api/assisted-install/v2/clusters/$CLUSTER_ID -H "Authorization: Bearer
${JWT_TOKEN}" | jq '
def host_name($host):
    if (.suggested_hostname // "") == "" then
        if (.inventory // "") == "" then
            "Unknown hostname, please wait"
        else
            .inventory | fromjson | .hostname
        end
    else
        .suggested_hostname
    end;

def is_notable($validation):
    ["failure", "pending", "error"] | any(. == $validation.status);

def notable_validations($validations_info):
    [
        $validations_info // "{}"
        | fromjson
        | to_entries[].value[]
        | select(is_notable(.))
    ];

{
    "Hosts validations": {
        "Hosts": [
            .hosts[]
            | select(.status != "installed")
            | {
                "id": .id,
                "name": host_name(.),
                "status": .status,
                "notable_validations": notable_validations(.validations_info)
            }
        ]
    },
    "Cluster validations info": {
        "notable_validations": notable_validations(.validations_info)
    }
}
' -r
```

输出示例


```
{
  "Hosts validations": {
    "Hosts": [
      {
        "id": "97ec378c-3568-460c-bc22-df54534ff08f",
        "name": "localhost.localdomain",
        "status": "insufficient",
        "notable_validations": [
          {
            "id": "ntp-synced",
            "status": "failure",
            "message": "Host couldn't synchronize with any NTP server"
          },
          {
            "id": "api-domain-name-resolved-correctly",
            "status": "error",
            "message": "Parse error for domain name resolutions result"
          },
          {
            "id": "api-int-domain-name-resolved-correctly",
            "status": "error",
            "message": "Parse error for domain name resolutions result"
          },
          {
            "id": "apps-domain-name-resolved-correctly",
            "status": "error",
            "message": "Parse error for domain name resolutions result"
          }
        ]
      }
    ]
  },
  "Cluster validations info": {
    "notable_validations": []
  }
}
```


11. 当上一个命令显示主机就绪时，通过运行以下命令来使用 /v2/infra-envs/{infra_env_id}/hosts/{host_id}/actions/install API 开始安装：

```
$ curl -X POST -s "$API_URL/api/assisted-install/v2/infra-envs/$INFRA_ENV_ID/hosts/$HOST_ID/actions/install"  -H "Authorization: Bearer ${JWT_TOKEN}"
```

12. 当安装继续进行时，安装会为 worker 节点生成待处理的证书签名请求 (CSR)。

重要

您必须批准 CSR 才能完成安装。

运行以下 API 调用以监控集群安装：

```
$ curl -s "$API_URL/api/assisted-install/v2/clusters/$CLUSTER_ID" -H "Authorization: Bearer
${JWT_TOKEN}" | jq '{
    "Cluster day-2 hosts":
        [
            .hosts[]
            | select(.status != "installed")
            | {id, requested_hostname, status, status_info, progress, status_updated_at,
updated_at, infra_env_id, cluster_id, created_at}
        ]
}'
```

输出示例

```
{
  "Cluster day-2 hosts": [
    {
      "id": "a1c52dde-3432-4f59-b2ae-0a530c851480",
      "requested_hostname": "control-plane-1",
      "status": "added-to-existing-cluster",
      "status_info": "Host has rebooted and no further updates will be posted. Please check
console for progress and to possibly approve pending CSRs",
      "progress": {
        "current_stage": "Done",
        "installation_percentage": 100,
        "stage_started_at": "2022-07-08T10:56:20.476Z",
        "stage_updated_at": "2022-07-08T10:56:20.476Z"
      },
      "status_updated_at": "2022-07-08T10:56:20.476Z",
      "updated_at": "2022-07-08T10:57:15.306369Z",
      "infra_env_id": "b74ec0c3-d5b5-4717-a866-5b6854791bd3",
      "cluster_id": "8f721322-419d-4eed-aa5b-61b50ea586ae",
      "created_at": "2022-07-06T22:54:57.161614Z"
    }
  ]
}
```

13. 可选：运行以下命令以查看集群的所有事件：

```
$ curl -s "$API_URL/api/assisted-install/v2/events?cluster_id=$CLUSTER_ID" -H
"Authorization: Bearer ${JWT_TOKEN}" | jq -c '.[] | {severity, message, event_time, host_id}'
```

输出示例

```
{"severity":"info","message":"Host compute-0: updated status from insufficient to known (Host
is ready to be installed)","event_time":"2022-07-08T11:21:46.346Z","host_id":"9d7b3b44-
1125-4ad0-9b14-76550087b445"}
{"severity":"info","message":"Host compute-0: updated status from known to installing
(Installation is in progress)","event_time":"2022-07-08T11:28:28.647Z","host_id":"9d7b3b44-
1125-4ad0-9b14-76550087b445"}
{"severity":"info","message":"Host compute-0: updated status from installing to installing-in-
progress (Starting installation)","event_time":"2022-07-
08T11:28:52.068Z","host_id":"9d7b3b44-1125-4ad0-9b14-76550087b445"}
{"severity":"info","message":"Uploaded logs for host compute-0 cluster 8f721322-419d-4eed-
aa5b-61b50ea586ae","event_time":"2022-07-08T11:29:47.802Z","host_id":"9d7b3b44-1125-
```

```
4ad0-9b14-76550087b445"}
{"severity":"info","message":"Host compute-0: updated status from installing-in-progress to added-to-existing-cluster (Host has rebooted and no further updates will be posted. Please check console for progress and to possibly approve pending CSRs)","event_time":"2022-07-08T11:29:48.259Z","host_id":"9d7b3b44-1125-4ad0-9b14-76550087b445"}
{"severity":"info","message":"Host: compute-0, reached installation stage Rebooting","event_time":"2022-07-08T11:29:48.261Z","host_id":"9d7b3b44-1125-4ad0-9b14-76550087b445"}
```

14. 登录到集群并批准待处理的 CSR 以完成安装。

验证

- 检查新 worker 节点是否已成功添加到集群中，状态为 **Ready** ：

```
$ oc get nodes
```

输出示例

```
NAME                         STATUS   ROLES          AGE   VERSION
control-plane-1.example.com  Ready    master,worker  56m   v1.29.4
compute-1.example.com        Ready    worker         11m   v1.29.4
```

其他资源

- 用户置备的 DNS 要求
- 批准机器的证书签名请求

10.1.4. 手动将 worker 节点添加到单节点 OpenShift 集群

您可以通过从 Red Hat Enterprise Linux CoreOS (RHCOS) ISO 启动 worker 节点，并使用集群 **worker.ign** 文件手动将 worker 节点添加到单节点 OpenShift 集群中。

先决条件

- 在裸机上安装单节点 OpenShift 集群。
- 安装 OpenShift CLI (**oc**) 。
- 以具有 **cluster-admin** 特权的用户身份登录。
- 确保将 worker 节点添加到集群中的集群需要的所有 DNS 记录。

流程

1. 设置 OpenShift Container Platform 版本：

```
$ OCP_VERSION=<ocp_version> 1
```

1 将 **<ocp_version>** 替换为当前版本，如 **latest-4.16**

2. 设置主机架构：

```
$ ARCH=<architecture> 1
```

1 将 **<architecture>** 替换为目标主机架构，如 **aarch64** 或 **x86_64**。

3. 运行以下命令，从正在运行的单节点集群获取 **worker.ign** 数据：

```
$ oc extract -n openshift-machine-api secret/worker-user-data-managed --keys=userData --
to=- > worker.ign
```

4. 在 Web 服务器上托管 **worker.ign** 文件，可从您的网络访问。

5. 运行以下命令下载 OpenShift Container Platform 安装程序并使其可用：

```
$ curl -k https://mirror.openshift.com/pub/openshift-
v4/clients/ocp/$OCP_VERSION/openshift-install-linux.tar.gz > openshift-install-linux.tar.gz
```

```
$ tar zxvf openshift-install-linux.tar.gz
```

```
$ chmod +x openshift-install
```

6. 检索 RHCOS ISO URL：

```
$ ISO_URL=$(./openshift-install coreos print-stream-json | grep location | grep $ARCH | grep
iso | cut -d\" -f4)
```

7. 下载 RHCOS ISO：

```
$ curl -L $ISO_URL -o rhcos-live.iso
```

8. 使用 RHCOS ISO 和托管的 **worker.ign** 文件来安装 worker 节点：

   a. 使用 RHCOS ISO 和您首选的安装方法引导目标主机。

   b. 当目标主机从 RHCOS ISO 引导时，打开目标主机上的控制台。

   c. 如果您的本地网络没有启用 DHCP，则需要使用新主机名创建 ignition 文件，并在运行 RHCOS 安装前配置 worker 节点静态 IP 地址。执行以下步骤：

      i. 使用静态 IP 配置 worker 主机网络连接。在目标主机控制台中运行以下命令：

```
$ nmcli con mod <network_interface> ipv4.method manual /
ipv4.addresses <static_ip> ipv4.gateway <network_gateway> ipv4.dns <dns_server>
/
802-3-ethernet.mtu 9000
```

其中：

<static_ip>
是主机静态 IP 地址和 CIDR，如 **10.1.101.50/24**

<network_gateway>
是网络网关，如 **10.1.101.1**

ii. 激活修改的网络接口：

```
$ nmcli con up <network_interface>
```

iii. 创建新的 ignition 文件 **new-worker.ign**，其中包含对原始 **worker.ign** 的引用，以及 **coreos-installer** 程序用来在新 worker 主机上填充 **/etc/hostname** 文件的额外指令。例如：

```
{
  "ignition":{
    "version":"3.2.0",
    "config":{
      "merge":[
        {
          "source":"<hosted_worker_ign_file>" (1)
        }
      ]
    }
  },
  "storage":{
    "files":[
      {
        "path":"/etc/hostname",
        "contents":{
          "source":"data:,<new_fqdn>" (2)
        },
        "mode":420,
        "overwrite":true,
        "path":"/etc/hostname"
      }
    ]
  }
}
```

(1) **<hosted_worker_ign_file>** 是原始 **worker.ign** 文件的本地可访问的 URL。例如：**http://webserver.example.com/worker.ign**

(2) **<new_fqdn>** 是您为 worker 节点设置的新 FQDN。例如，**new-worker.example.com**。

iv. 在 Web 服务器上托管 **new-worker.ign** 文件，可从您的网络访问。

v. 运行以下 **coreos-installer** 命令，传递 **ignition-url** 和硬盘详情：

```
$ sudo coreos-installer install --copy-network /
--ignition-url=<new_worker_ign_file> <hard_disk> --insecure-ignition
```

其中：

<new_worker_ign_file>

是托管 **new-worker.ign** 文件的本地可访问 URL，例如 **http://webserver.example.com/new-worker.ign**

<hard_disk>

是安装 RHCOS 的硬盘，例如 **/dev/sda**

d. 对于启用了 DHCP 的网络，您不需要设置静态 IP。在目标主机控制台中运行以下 **coreos-installer** 命令以安装该系统：

```
$ coreos-installer install --ignition-url=<hosted_worker_ign_file> <hard_disk>
```

e. 要手动启用 DHCP，请将以下 **NMStateConfig** CR 应用到单节点 OpenShift 集群：

```
apiVersion: agent-install.openshift.io/v1
kind: NMStateConfig
metadata:
  name: nmstateconfig-dhcp
  namespace: example-sno
  labels:
    nmstate_config_cluster_name: <nmstate_config_cluster_label>
spec:
  config:
    interfaces:
      - name: eth0
        type: ethernet
        state: up
        ipv4:
          enabled: true
          dhcp: true
        ipv6:
          enabled: false
  interfaces:
    - name: "eth0"
      macAddress: "AA:BB:CC:DD:EE:11"
```

重要

使用静态 IP 地址成功部署 worker 节点需要 **NMStateConfig** CR，如果单节点 OpenShift 使用静态 IP 地址部署，则使用动态 IP 地址添加带有动态 IP 地址的 worker 节点。集群网络 DHCP 不会自动为新 worker 节点设置这些网络设置。

9. 当安装继续进行时，安装会为 worker 节点生成待处理的证书签名请求 (CSR)。出现提示时，批准待处理的 CSR 以完成安装。

10. 安装完成后，重启主机。主机加入集群作为新的 worker 节点。

验证

- 检查新 worker 节点是否已成功添加到集群中，状态为 **Ready** ：

```
$ oc get nodes
```

输出示例

```
NAME                         STATUS   ROLES          AGE   VERSION
control-plane-1.example.com  Ready    master,worker  56m   v1.29.4
compute-1.example.com        Ready    worker         11m   v1.29.4
```

其他资源

- 用户置备的 DNS 要求
- 批准机器的证书签名请求

10.1.5. 批准机器的证书签名请求

当您将机器添加到集群时，会为您添加的每台机器生成两个待处理证书签名请求(CSR)。您必须确认这些 CSR 已获得批准，或根据需要自行批准。必须首先批准客户端请求，然后批准服务器请求。

先决条件

- 您已将机器添加到集群中。

流程

1. 确认集群可以识别这些机器：

```
$ oc get nodes
```

输出示例

```
NAME      STATUS    ROLES   AGE  VERSION
master-0  Ready     master  63m  v1.29.4
master-1  Ready     master  63m  v1.29.4
master-2  Ready     master  64m  v1.29.4
```

输出中列出了您创建的所有机器。

注意

在有些 CSR 被批准前，前面的输出可能不包括计算节点（也称为 worker 节点）。

2. 检查待处理的 CSR，并确保添加到集群中的每台机器都有 **Pending** 或 **Approved** 状态的客户端请求：

```
$ oc get csr
```

输出示例

```
NAME        AGE    REQUESTOR                                                   CONDITION
csr-8b2br   15m    system:serviceaccount:openshift-machine-config-operator:node-
bootstrapper   Pending
csr-8vnps   15m    system:serviceaccount:openshift-machine-config-operator:node-
bootstrapper   Pending
...
```

在本例中，两台机器加入集群。您可能会在列表中看到更多已批准的 CSR。

3. 如果 CSR 没有获得批准，在您添加的机器的所有待处理 CSR 都处于 **Pending 状态** 后，请批准集群机器的 CSR：

注意

由于 CSR 会自动轮转，因此请在将机器添加到集群后一小时内批准您的 CSR。如果没有在一小时内批准它们，证书将会轮转，每个节点会存在多个证书。您必须批准所有这些证书。批准客户端 CSR 后，Kubelet 为服务证书创建一个二级 CSR，这需要手动批准。然后，如果 Kubelet 请求具有相同参数的新证书，则后续提供证书续订请求由 **machine-approver** 自动批准。

注意

对于在未启用机器 API 的平台上运行的集群，如裸机和其他用户置备的基础架构，您必须实施一种方法来自动批准 kubelet 提供证书请求(CSR)。如果没有批准请求，则 **oc exec**、**oc rsh** 和 **oc logs** 命令将无法成功，因为 API 服务器连接到 kubelet 时需要服务证书。与 Kubelet 端点联系的任何操作都需要此证书批准。该方法必须监视新的 CSR，确认 CSR 由 system: **node** 或 **system:admin** 组中的 **node-bootstrapper** 服务帐户提交，并确认节点的身份。

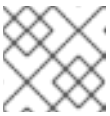
- 要单独批准，请对每个有效的 CSR 运行以下命令：

```
$ oc adm certificate approve <csr_name> 1
```

1 **<csr_name>** 是当前 CSR 列表中 CSR 的名称。

- 要批准所有待处理的 CSR，请运行以下命令：

```
$ oc get csr -o go-template='{{range .items}}{{if not .status}}{{.metadata.name}}{{"\n"}}
{{end}}{{end}}' | xargs --no-run-if-empty oc adm certificate approve
```



注意

在有些 CSR 被批准前，一些 Operator 可能无法使用。

4. 现在，您的客户端请求已被批准，您必须查看添加到集群中的每台机器的服务器请求：

```
$ oc get csr
```

输出示例

```
NAME        AGE    REQUESTOR                                                   CONDITION
csr-bfd72   5m26s  system:node:ip-10-0-50-126.us-east-2.compute.internal
Pending
csr-c57lv   5m26s  system:node:ip-10-0-95-157.us-east-2.compute.internal
Pending
...
```

5. 如果剩余的 CSR 没有被批准，且处于 **Pending** 状态，请批准集群机器的 CSR：

- 要单独批准，请对每个有效的 CSR 运行以下命令：

```
$ oc adm certificate approve <csr_name> 1
```

1 **<csr_name>** 是当前 CSR 列表中 CSR 的名称。

- 要批准所有待处理的 CSR，请运行以下命令：

```
$ oc get csr -o go-template='{{range .items}}{{if not .status}}{{.metadata.name}}{{"\n"}}
{{end}}{{end}}' | xargs oc adm certificate approve
```

6. 批准所有客户端和服务器 CSR 后，机器将 处于 **Ready 状态**。运行以下命令验证：

```
$ oc get nodes
```

输出示例

```
NAME     STATUS   ROLES   AGE  VERSION
master-0 Ready    master  73m  v1.29.4
master-1 Ready    master  73m  v1.29.4
master-2 Ready    master  74m  v1.29.4
worker-0 Ready    worker  11m  v1.29.4
worker-1 Ready    worker  11m  v1.29.4
```

注意

批准服务器 CSR 后可能需要几分钟时间让机器过渡到 **Ready 状态**。

其他信息

- 如需有关 CSR 的更多信息，请参阅 证书签名请求。

第 11 章 节点指标仪表板

节点指标仪表板是一个可视化分析仪表板，可帮助您识别潜在的 pod 扩展问题。

11.1. 关于节点指标仪表板

节点指标仪表板可让管理和支持团队成员监控与 pod 扩展相关的指标，包括用于诊断和排除扩展问题的扩展限制。特别是，您可以使用通过仪表板显示的可视化分析来监控跨节点的工作负载分布。从这些分析中获得的 Insights 可帮助您确定 CRI-O 和 Kubelet 系统组件的运行状况，并识别过度或未平衡的资源消耗和系统不稳定的潜在源。

仪表板显示按以下类别组织的可视化分析小部件：

Critical

包括有助于您识别系统不稳定和低效率的节点问题的视觉化

Outliers

包括视觉化呈现在 95th percentile 之外的运行时持续时间的进程的直方图

平均持续时间

帮助您跟踪系统组件处理操作的时间变化

操作数量

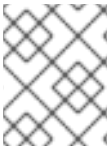
显示视觉化，可帮助您识别所运行操作数量的更改，这有助于确定系统的负载均衡和效率

11.2. 访问节点指标仪表板

您可以从 **Administrator** 视角访问节点指标仪表板。

流程

1. 展开 **Observe** 菜单选项并选择 **Dashboards**。
2. 在 **Dashboard** 过滤器下，选择 **Node cluster**。



注意

如果没有数据出现在 **Critical** 类别下的视觉化中，则不会检测到关键异常情况。仪表板按照预期工作。

11.3. 识别指示最佳节点资源使用情况的指标

节点指标仪表板分为四个类别：**Critical**, **Outliers**, **Average durations**, 和 **Number of Operations**。**Critical** 类别中的指标可帮助您指示最佳节点资源使用情况。这些指标包括：

- 在最后一天中带有最多 OOM 终止的头 3 个容器
- 在最后一小时中镜像拉取的失败率
- 系统保留内存使用率 > 80% 的节点
- Kubelet 系统保留内存使用率 > 50% 的节点
- CRI-O 系统保留内存使用率 > 50% 的节点

- 系统保留 CPU 使用率 > 80% 的节点
- Kubelet 系统保留 CPU 使用率 > 50% 的节点
- CRI-O 系统保留 CPU 使用率 > 50% 的节点

11.3.1. 在最后一天中带有最多 OOM 终止的头 3 个容器

在最后一天中带有最多 OOM 终止的头 3 个容器会获取遇到最多 Out-Of-Memory (OOM) 终止的前三个容器的详细信息。

默认查询示例

```
topk(3, sum(increase(container_runtime_crio_containers_oom_count_total[1d])) by (name))
```

OOM 终止会强制系统因为内存不足而终止某些进程。频繁 OOM 终止可能会妨碍节点的功能，甚至整个 Kubernetes 生态系统。频繁遇到 OOM 终止的容器可能会消耗比预期更多的内存，这会导致系统不稳定。

使用此指标可以发现频繁出现 OOM 终止的容器，并调查这些容器消耗大量内存的原因。如果需要，调整资源分配，并考虑根据容器内存用量重新定义大小。您还可以查看 **Outliers**、**Average durations** 下的指标，以及**操作类别的数量**，以便进一步了解节点的健康状态和稳定性。

11.3.2. 在最后一小时中镜像拉取的失败率

在最后一小时中的镜像拉取故障率会将失败镜像的总数除以成功和失败的镜像拉取的总数，以提供失败率。

默认查询示例

```
rate(container_runtime_crio_image_pulls_failure_total[1h]) /
(rate(container_runtime_crio_image_pulls_success_total[1h]) +
rate(container_runtime_crio_image_pulls_failure_total[1h]))
```

了解镜像拉取失败率对于维护节点健康状况至关重要。高故障率可能表示网络问题、存储问题、错误配置或其他可能破坏 pod 密度和部署新容器的问题。

如果此查询的结果很高，调查可能的原因，如网络连接、远程存储库、节点存储以及镜像引用的准确性。您还可以查看 **Outliers**、**Average durations** 下的指标，以及**操作数量**，以便进一步了解。

11.3.3. 系统保留内存使用率 > 80% 的节点

系统保留内存使用率 > 80% 的节点查询计算每个节点的系统保留内存的百分比。计算方法是，总的 resident set size (RSS) 除以从可分配内存中减去的节点的总内存容量。RSS 是由进程占用的系统内存的部分，它位于主内存 (RAM) 中。如果生成的值等于或超过 80% 阈值，则节点会被标记。

默认查询示例

```
sum by (node) (container_memory_rss{id="/system.slice"}) / sum by (node)
(kube_node_status_capacity{resource="memory"} -
kube_node_status_allocatable{resource="memory"}) * 100 >= 80
```

系统保留内存对于 Kubernetes 节点至关重要，因为它用于运行系统守护进程和 Kubernetes 系统守护进程。超过 80% 的系统保留内存使用率表示系统和 Kubernetes 守护进程消耗太多内存，可能会导致节点不

稳定并可能进一步影响到运行 pod 的性能。过量内存消耗可能会导致内存不足(OOM)终止程序，它可能会终止关键系统进程来释放内存。

如果一个节点被这个指标标记，请找出哪个系统或 Kubernetes 进程消耗了过量内存，并采取适当的操作来缓解这种情况。可能采取的操作包括，减少非关键进程、优化程序配置来减少内存用量，或者将节点系统升级到具有更大内存容量的硬件。您还可以查看 **Outliers**、**Average durations** 下的指标，以及**操作数量**，以便进一步深入了解节点性能。

11.3.4. Kubelet 系统保留内存使用率 > 50% 的节点

Kubelet 系统保留内存使用率 > 50% 的节点查询指示 Kubelet 系统保留内存利用率超过 50% 的节点。查询检查 Kubelet 进程本身在节点上消耗的内存。

默认查询示例

```
sum by (node) (container_memory_rss{id="/system.slice/kubelet.service"}) / sum by (node)
(kube_node_status_capacity{resource="memory"} -
kube_node_status_allocatable{resource="memory"}) * 100 >= 50
```

此查询可帮助您识别节点中可能的内存压力，这可能会影响节点操作稳定性和效率。kubelet 内存使用率持续超过系统保留内存的 50%，表示系统保留设置没有正确配置，且节点不稳定的风险较高。

如果此指标被高亮显示，请检查您的配置策略，并考虑调整系统保留的设置，或调整为 Kubelet 设置的资源限值。另外，如果您的 Kubelet 内存使用率持续超过总保留系统内存的一半，请检查 **Outliers** 下的指标、**平均持续时间**，以及**操作数量**，以便进一步了解更精确的诊断。

11.3.5. CRI-O 系统保留内存使用率 > 50% 的节点

带有 CRI-O 系统保留内存使用率 > 50% 的节点查询计算为 CRI-O 系统保留的内存百分比大于或等于 50% 的所有节点。在这种情况下，内存用量由 resident set size (RSS) 定义，这是 RAM 中持有的 CRI-O 系统内存的一部分。

默认查询示例

```
sum by (node) (container_memory_rss{id="/system.slice/crio.service"}) / sum by (node)
(kube_node_status_capacity{resource="memory"} -
kube_node_status_allocatable{resource="memory"}) * 100 >= 50
```

此查询可帮助您监控每个节点上的 CRI-O 系统保留的内存状态。高利用率可能表示缺少可用资源和潜在性能问题。如果为 CRI-O 系统保留的内存超过推荐的 50% 限制，这表示节点上的 CRI-O 使用了系统保留内存的一半。

检查内存分配和使用，并评估内存资源是否需要转换或增加，以防止可能的节点不稳定。您还可以检查 **Outliers**、**Average durations** 下的指标，以及**操作数量**，以便进一步了解。

11.3.6. 系统保留 CPU 使用率 > 80% 的节点

系统保留 CPU 使用率 > 80% 的节点查询可识别 system-reserved CPU 使用率超过 80% 的节点。查询侧重于 system-reserved 容量，以计算最后 5 分钟内 CPU 用量率，并将其与节点上可用的 CPU 资源进行比较。如果比率超过 80%，节点的结果会显示在指标中。

默认查询示例

```
sum by (node) (rate(container_cpu_usage_seconds_total{id="/system.slice"}[5m]) * 100) / sum by
(node) (kube_node_status_capacity{resource="cpu"} -
kube_node_status_allocatable{resource="cpu"}) >= 80
```

此查询表示 system-reserved CPU 用量的关键级别，这可能会导致资源耗尽。高 system-reserved CPU 使用率可能会导致系统进程（包括 Kubelet 和 CRI-O）无法充分管理节点上的资源。此查询可以指示系统进程过高或配置错误的 CPU 分配。

潜在的修正措施包括将工作负载重新平衡到其他节点，或者增加分配给节点的 CPU 资源。调查高系统 CPU 使用率的原因，并查看 **Outliers** 中对应的指标、**平均持续时间**，以及**操作**类别，以便深入了解节点的行为。

11.3.7. Kubelet 系统保留 CPU 使用率 > 50% 的节点

带有 Kubelet 系统保留的 CPU 使用率 > 50% 查询的节点计算 Kubelet 系统当前使用的 CPU 的百分比。

默认查询示例

```
sum by (node) (rate(container_cpu_usage_seconds_total{id="/system.slice/kubelet.service"}[5m]) *
100) / sum by (node) (kube_node_status_capacity{resource="cpu"} -
kube_node_status_allocatable{resource="cpu"}) >= 50
```

Kubelet 使用系统保留 CPU 进行自己的操作，并运行关键系统服务。对于节点的健康状况，务必要确保系统保留 CPU 使用量不超过 50% 阈值。超过这个限制可能会指示 Kubelet 上的大量利用率或负载，这会影响节点稳定性并可能整个 Kubernetes 集群的性能。

如果此指标中显示任何节点，Kubelet 和系统总体负载非常重。您可以通过在集群中的其他节点之间平衡负载来减少特定节点上的过载。检查 **Outliers**、**Average 持续时间**和**操作**类别下的其他查询指标，以便进一步深入了解并采取必要的纠正措施。

11.3.8. CRI-O 系统保留 CPU 使用率 > 50% 的节点

CRI-O 系统保留 CPU 使用率 > 50% 的节点查询在最后 5 分钟内标识 CRI-O 系统保留 CPU 使用率超过 50% 的节点。查询会基于每个节点监控 CRI-O （容器运行时）的 CPU 资源消耗。

默认查询示例

```
sum by (node) (rate(container_cpu_usage_seconds_total{id="/system.slice/crio.service"}[5m]) * 100)
/ sum by (node) (kube_node_status_capacity{resource="cpu"} -
kube_node_status_allocatable{resource="cpu"}) >= 50
```

此查询可以快速识别可能对 pod 性能造成负面影响的异常启动时间。如果此查询返回高值，您的 pod 启动时间会比通常慢，这意味着 kubelet、pod 配置或资源的潜在问题。

通过检查 pod 配置和分配的资源来进一步调查。确保它们与您的系统功能一致。如果您仍然看到高启动时间，请浏览仪表板中其他类别的指标面板，以确定您的系统组件的状态。

11.4. 自定义仪表板查询

您可以自定义用于构建节点指标仪表板的默认查询。

流程

1. 选择一个指标并点 **Inspect** 进入数据。此页面显示指标详情，包括查询结果的扩展视觉化、用于分析数据的 Prometheus 查询以及查询中使用的数据子集。

2. 对查询参数进行任何更改。

3. 可选：点 **Add query** 来针对数据运行额外的查询。

4. 点 **Run query** 来使用您指定的参数重新运行查询。