OpenShift Dedicated 4

节点

OpenShift Dedicated 节点

Last Updated: 2024-07-03

OpenShift Dedicated 4 节点

OpenShift Dedicated 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 Dedicated 中，control plane 节点不仅仅包含用于管理 OpenShift Dedicated 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 Dedicated 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

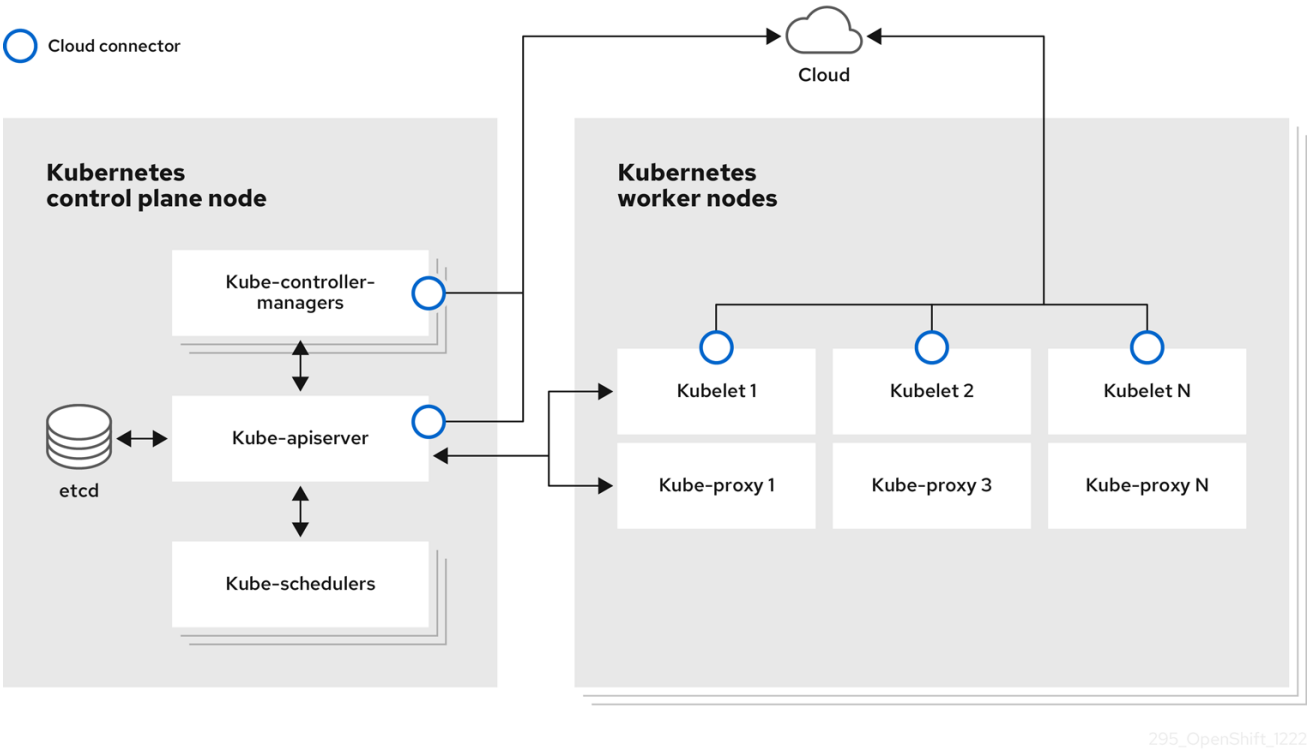
kubelet 在节点上运行并读取容器清单。它确保定义的容器已启动且正在运行。kubelet 进程维护工作和节点服务器的状态。kubelet 管理网络流量和端口转发。kubelet 管理仅由 Kubernetes 创建的容器。

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 Dedicated 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。

增强操作
OpenShift Dedicated 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。
- 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。

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 Dedicated 集群中管理 pod。

- 使用 OpenShift Dedicated 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。
  - Pod 拓扑分布约束。
- 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 Dedicated 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。

- Secrets:有些应用程序需要敏感信息，如密码和用户名。管理员可以使用 **Secret** 对象为 pod 提供机密数据，使用 **Secret** 对象。

1.3. 关于容器

容器是 OpenShift Dedicated 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 Dedicated 提供称为 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 Dedicated 集群来使集群高效和应用程序 pod 具有高可用性。

1.4. OPENSHIFT DEDICATED 节点常用术语表

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 Dedicated 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。

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 Dedicated 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 Dedicated 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 Dedicated 将 pod 视为不可变；在运行时无法更改 pod 定义。OpenShift Dedicated 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。

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

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：

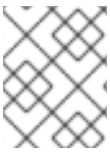
Pod 对象定义（YAML）

```
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 Dedicated 为容器定义了一个安全上下文，指定是否允许它们作为特权容器运行，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 Dedicated 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 Dedicated 利用 Kubernetes 的 *pod* 概念，它是共同部署在同一主机上的一个或多个容器，也是可被定义、部署和管理的最小计算单元。对容器而言，Pod 大致相当于机器实例（物理或虚拟）。

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

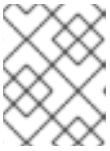
您可以在 OpenShift CLI (**oc**) 和 Web 控制台中查看各种资源的日志。日志从日志的尾部或末尾读取。

先决条件

- 访问 OpenShift CLI（**oc**）。

流程 (UI)

1. 在 OpenShift Dedicated 控制台中，进入到 **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 DEDICATED 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 OpenShift Dedicated 在该 pod 中的容器退出时如何响应。该策略适用于 pod 中的所有容器。

可能的值有：

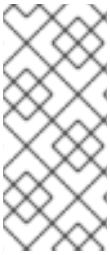
- **Always** - 在 pod 被重启之前，按规定的延时值（10s，20s，40s）不断尝试重启 pod 中成功退出的容器（最长为 5 分钟）。默认值为 **Always**。
- **OnFailure** - 按规定的延时值（10s，20s，40s）不断尝试重启 pod 中失败的容器，上限为 5 分钟。
- **Never** - 不尝试重启 pod 中已退出或失败的容器。Pod 立即失败并退出。

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 Dedicated 会启动新的 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 Dedicated 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。

如需了解 OpenShift Dedicated 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>
```

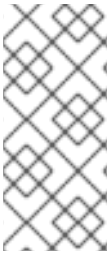
2.3.3. 了解如何使用 pod 中断预算来指定必须在线的 pod 数量

*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 Dedicated 中的所有机器配置池，**maxUnavailable** 的默认设置都是 **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
```

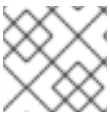
注意

以下示例包含特定于 AWS 上的 OpenShift Dedicated 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>
```

2.4. 使用 SECRET 为 POD 提供敏感数据

其他资源

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。

2.4.1. 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 Dedicated 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。

2.4.1.1. 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** ：将用作镜像 pull secret
- **kubernetes.io/dockerconfigjson**: 使用 作为镜像 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。

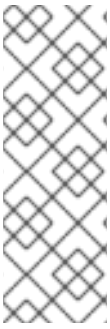
有关创建不同类型的 secret 的示例，请参阅 *了解如何创建 secret*。

2.4.1.2. Secret 数据密钥

Secret 密钥必须在 DNS 子域中。

2.4.1.3. 自动生成的镜像 pull secret

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



注意

在 OpenShift Dedicated 4.16 之前，还为每个创建的服务帐户 API 令牌 secret 生成长期服务帐户令牌 secret。从 OpenShift Dedicated 4.16 开始，不再创建此服务帐户 API 令牌 secret。

升级到 4 后，任何现有的长期服务帐户 API 令牌 secret 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。

2.4.2. 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。

2.4.2.1. 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。

2.4.2.2. 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

2.4.2.3. 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

```
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。

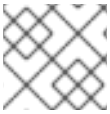
其他资源

- 了解如何创建 secret

2.4.2.4. 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

2.4.2.5. 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

2.4.2.6. 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

2.4.2.7. 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。此字段是只写的；其值仅通过 **data** 字段返回。

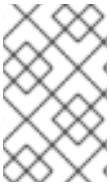
3. 点 **Create**。
4. 点 **Add Secret to workload**。
   a. 从下拉菜单中选择要添加的工作负载。
   b. 点击 **Save**。

2.4.3. 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。

2.4.4. 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。

2.4.5. 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 Dedicated，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。

2.4.5.1. 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
   ```

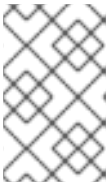
5. 编辑与该 secret 搭配的 **Pod** spec。

```
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。

2.4.6. 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：

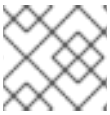
1. 删除 secret：

```
$ 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 **-**。

2.5. 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。

2.5.1. 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 Dedicated 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 Dedicated 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 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 Dedicated 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。

2.5.2. 在 OpenShift Dedicated Web 控制台中创建配置映射

您可以在 OpenShift Dedicated 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**。

2.5.3. 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]
```

2.5.3.1. 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
```

2.5.3.2. 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 Dedicated 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。

2.5.3.3. 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
```

2.5.4. 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。

2.5.4.1. 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**。

2.5.4.2. 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
```

2.5.4.3. 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
```

2.6. 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。

要使用优先级与抢占功能，引用 pod 规格中的优先级类，以应用该权重以进行调度。

2.6.1. 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。

2.6.1.1. 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 Dedicated 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。

2.6.1.2. 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。

2.6.2. 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。

2.6.2.1. 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。

2.6.2.2. 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 Dedicated 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。

2.6.2.3. 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。

2.6.3. 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. 通过创建类似如下的 YAML 文件，定义 pod spec 使其包含优先级类的名称：

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: system-cluster-critical (1)
```

(1) 指定要用于此 pod 的优先级类。

2. 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。

2.7. 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。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。

2.7.1. 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 Dedicated 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

- 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。

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 Dedicated 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 Dedicated 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 Dedicated 版本定义自定义 Metrics Autoscaler Operator 版本。

Version	OpenShift Dedicated 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 Dedicated 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 Dedicated 集群中运行的 Operator 提供了程序错误修正。以下公告可用于 RHBA-2024:2043。

重要

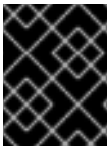
在安装自定义 Metrics Autoscaler Operator 的这个版本前，请删除任何以前安装的技术预览版本或社区支持的 KEDA 版本。

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 Dedicated 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。

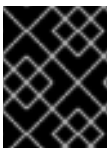
3.1.2.2.1. 程序错误修复

- 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 **指标** 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 Dedicated 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 Dedicated 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 Dedicated 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 Dedicated 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 Dedicated 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 Dedicated **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 Dedicated 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

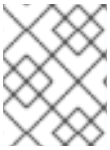
作为开发者，您可以使用 Custom Metrics Autoscaler Operator for Red Hat OpenShift 指定 OpenShift Dedicated 如何根据不基于 CPU 或内存的自定义指标自动增加或减少部署、有状态集、自定义资源或作业的数量。

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 Dedicated 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

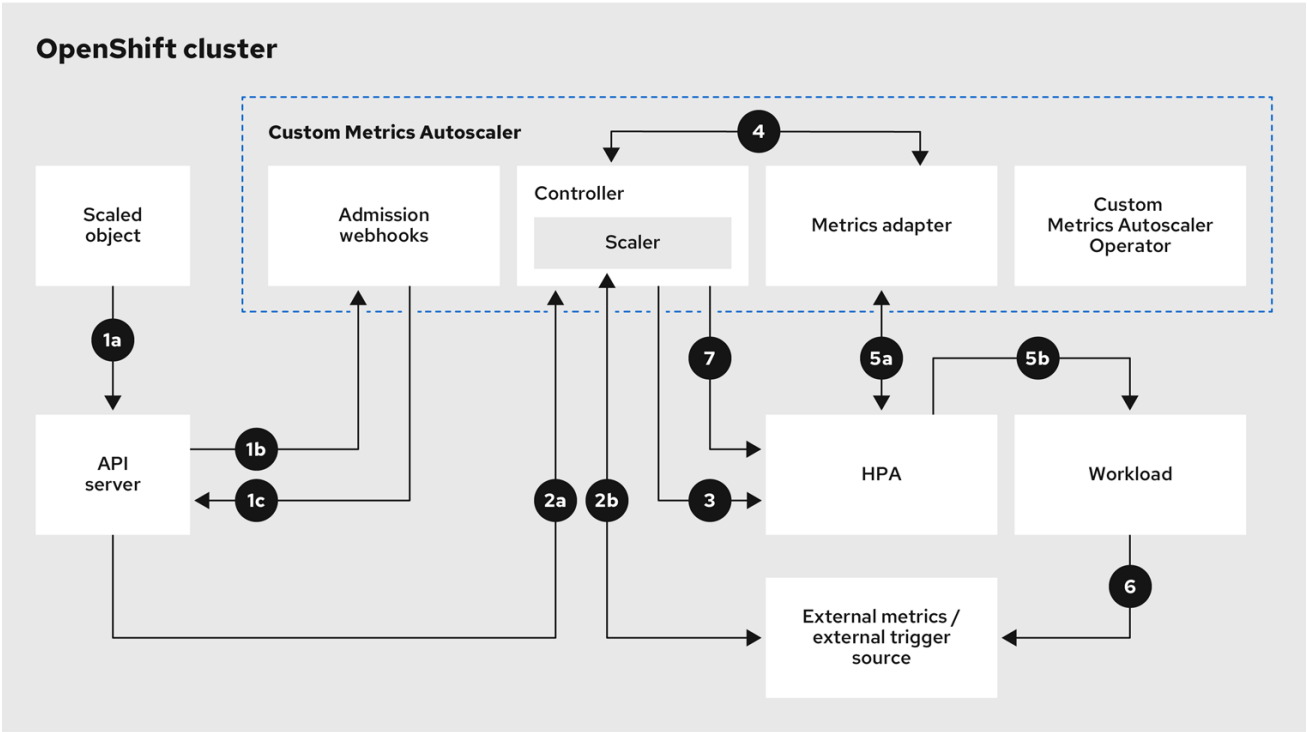
您只能为每个您要扩展的工作负载创建一个扩展对象或扩展作业。另外，您不能在同一工作负载中使用扩展的对象或扩展作业以及 pod 横向自动扩展 (HPA)。

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 Dedicated 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

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。
  如果您的 OpenShift Dedicated 集群位于由红帽所有的云帐户中（非CCS），您必须请求 **cluster-admin** 权限。
- 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
```

- 确保 **keda** 命名空间存在。如果没有，则必须安全地创建 **keda** 命名空间。

流程

1. 在 OpenShift Dedicated Web 控制台中，点 **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** 选择了 **A specific namespace on the cluster** 选项。
4. 对于 **Installed Namespace**，点 **Select a namespace**。
5. 点 **Select Project**:
   - 如果存在 **keda** 命名空间，请从列表中选择 **keda**。
   - 如果 **keda** 命名空间不存在：
     a. 选择 **Create Project** 以打开 **Create Project** 窗口。
     b. 在 **Name** 字段中输入 **keda**。
     c. 在 **Display Name** 字段中输入描述性名称，如 **keda**。
     d. 可选：在 **Display Name** 字段中，为命名空间添加描述。
     e. 点 **Create**。
6. 点 **Install**。
7. 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 **keda** 项目，并验证 **custom-metrics-autoscaler-operator** autoscaler pod 是否正在运行。
   c. 进入到 **Workloads → Deployments** 以验证 **custom-metrics-autoscaler-operator** 部署是否正在运行。
8. 可选：使用以下命令在 OpenShift CLI 中验证安装：

```
$ oc get all -n 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
```

9. 安装 **KedaController** 自定义资源，该资源创建所需的 CRD：

   a. 在 OpenShift Dedicated web 控制台中，点 **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: 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

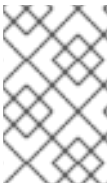
触发器（也称为 scalers）提供自定义 Metrics Autoscaler Operator 用来扩展 pod 的指标。

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

您可以根据 Prometheus 指标扩展 pod，该指标可以使用已安装的 OpenShift Dedicated 监控或外部 Prometheus 服务器作为指标源。有关使用 OpenShift Dedicated 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 Dedicated 监控。

(3) 可选：指定您要缩放的对象的命名空间。如果将 OpenShift Dedicated 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 Dedicated 监控

您可以使用已安装的 OpenShift Dedicated 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 Dedicated 监控。
- OpenShift Dedicated 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 Dedicated 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 Dedicated secret、平台原生 pod 身份验证机制、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**。

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: 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 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 keda
```

例如：

```
$ oc rsh pod/keda-metrics-apiserver-65c7cc44fd-rrl4r -n 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 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：

- **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 Dedicated **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 **dedicated-admin** 角色的用户身份登录到 OpenShift Dedicated。
- 安装了 OpenShift Dedicated CLI (**oc**)。

流程

1. 进入存储 **must-gather** 数据的目录。
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：

```
└── 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
```

```
├── 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 Dedicated Web 控制台访问指标并运行查询。

流程

1. 在 OpenShift Dedicated 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 Dedicated 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。

指标名称	描述
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 Dedicated 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。

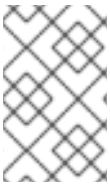
- 如果为 **True**，自定义指标自动扩展会获取指标，因为您创建的一个或多个对象中没有指标或多个问题。

3.10.2. 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 Dedicated 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 Dedicated 会在删除 **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 Dedicated 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 Dedicated web 控制台中，点 **Operators → Installed Operators**。
2. 切换到 **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 keda
```

7. 删除 Cluster Metric Autoscaler Operator：

```
$ oc delete operator/openshift-custom-metrics-autoscaler-operator.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 Dedicated 附带一个默认调度程序，满足大多数用户的需求。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 Dedicated 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：

- Pod 关联性和反关联性规则
- 节点关联性
- 节点选择器
- 节点过量使用

4.1.1. 关于默认调度程序

默认的 OpenShift Dedicated 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 Dedicated 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. 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。

4.1.2.2. 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。

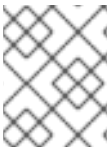
在 OpenShift Dedicated 中，可以借助 *pod 关联性*和 *pod 反关联性*来根据其他 pod 上的键/值标签限制 pod 有资格调度到哪些节点。

4.2.1. 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。

4.2.2. 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
```

4.2.3. 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
```

4.2.4. 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。

4.2.4.1. 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。

4.2.4.2. 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。

4.2.4.3. 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. 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 Dedicated 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。

4.3.1. 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。

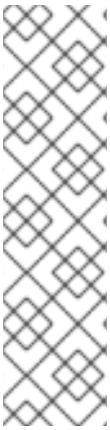
(2) 定义偏好规则。

(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。

4.3.2. 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. 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
```

4.3.3. 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. 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
```

4.3.4. 节点关联性规则示例

以下示例演示了节点关联性。

4.3.4.1. 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
```

4.3.4.2. 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. 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。

4.4.1. 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 Dedicated 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。

4.4.2. 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 Dedicated 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 Dedicated 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

4.5. 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。

4.5.1. 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 Dedicated 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 Dedicated 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 Dedicated 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

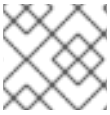
```
apiVersion: v1
kind: Pod
metadata:
  name: west-region
#...
spec:
  nodeSelector:
    region: west
#...
```

4.5.2. 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 Dedicated 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

- 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。

4.6. 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。

4.6.1. 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 Dedicated 管理员可以标记节点以提供拓扑信息，如区域、区域、节点或其他用户定义的域。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。

4.6.2. 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 **dedicated-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
```

4.6.3. 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。

4.6.3.1. 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]
```

4.6.3.2. 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]
```

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 Dedicated 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 Dedicated 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，方法是将 **NodeAffinity** 术语添加到守护进程集 pod 中，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: ''
#...
```

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
   ```

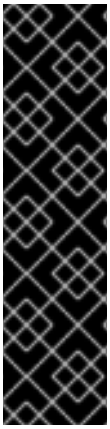
   输出示例

   ```
   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*在 OpenShift Dedicated 集群中执行任务。

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 Dedicated 中有两种资源类型可以创建运行一次对象：

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 Dedicated 协调 pod，以确定每个 pod 都应该使用什么操作或使用外部队列服务。
  - 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 Dedicated 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 Dedicated 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）。

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 Dedicated 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 Dedicated 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 Dedicated 中创建 cron 作业。

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
  concurrencyPolicy: "Replace"             2
  startingDeadlineSeconds: 200             3
  suspend: true                            4
  successfulJobsHistoryLimit: 3            5
  failedJobsHistoryLimit: 1                6
  jobTemplate:                             7
    spec:
      template:
        metadata:
          labels:                          8
            parent: "cronjobpi"
        spec:
```

```
    containers:
    - name: pi
      image: perl
      command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    restartPolicy: OnFailure 9
```

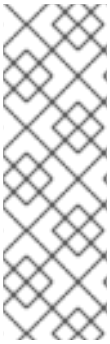
1. 以 cron 格式指定的作业调度计划。在本例中，作业将每分钟运行一次。
2. 可选的并发策略，指定如何对待 Cron Job 中的并发作业。只能指定以下并发策略之一。若未指定，默认为允许并发执行。
   - **Allow**，允许 Cron Job 并发运行。
   - **Forbid**，禁止并发运行。如果上一运行尚未结束，则跳过下一运行。
   - **Replace**，取消当前运行的作业并替换为新作业。
3. 可选期限（秒为单位），如果作业因任何原因而错过预定时间，则在此期限内启动作业。错过的作业执行计为失败的作业。若不指定，则没有期限。
4. 可选标志，允许挂起 Cron Job。若设为 **true**，则会挂起所有后续执行。
5. 要保留的成功完成作业数（默认为 3）。
6. 要保留的失败完成作业数（默认为 1）。
7. 作业模板。类似于作业示例。
8. 为此 Cron Job 生成的作业设置一个标签。
9. pod 的重启策略。这不适用于作业控制器。

注意

.spec.successfulJobsHistoryLimit 和 **.spec.failedJobsHistoryLimit** 字段是可选的。用于指定应保留的已完成作业和已失败作业的数量。默认情况下，分别设置为 **3** 和 **1**。如果将限制设定为 **0**，则对应种类的作业完成后不予保留。

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. 查看并列出 OPENSHIFT DEDICATED 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 Dedicated 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。

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。

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. 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。

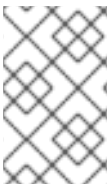
Operator 将为 OpenShift Dedicated 的容器化 TuneD 守护进程作为一个 Kubernetes 守护进程集进行管理。它保证了自定义性能优化设置以可被守护进程支持的格式传递到在集群中运行的所有容器化的 TuneD 守护进程中。相应的守护进程会在集群的所有节点上运行，每个节点上运行一个。

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，以实现 OpenShift Dedicated 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。

Node Tuning Operator 是版本 4.1 及之后的版本中的标准 OpenShift Dedicated 安装的一部分。

注意

在早期版本的 OpenShift Dedicated 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 Dedicated 4.11 及更高版本中，此功能是 Node Tuning Operator 的一部分。

6.2.1. 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 Dedicated 平台提供标准节点级别的性能优化，只能修改它来设置 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，并根据节点或 pod 标签和配置集优先级对 OpenShift Dedicated 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。

6.2.2. 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。

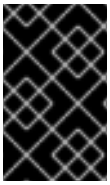
3 可选的对象类型（**node** 或 **pod**）。如果省略，会使用 **node**。

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

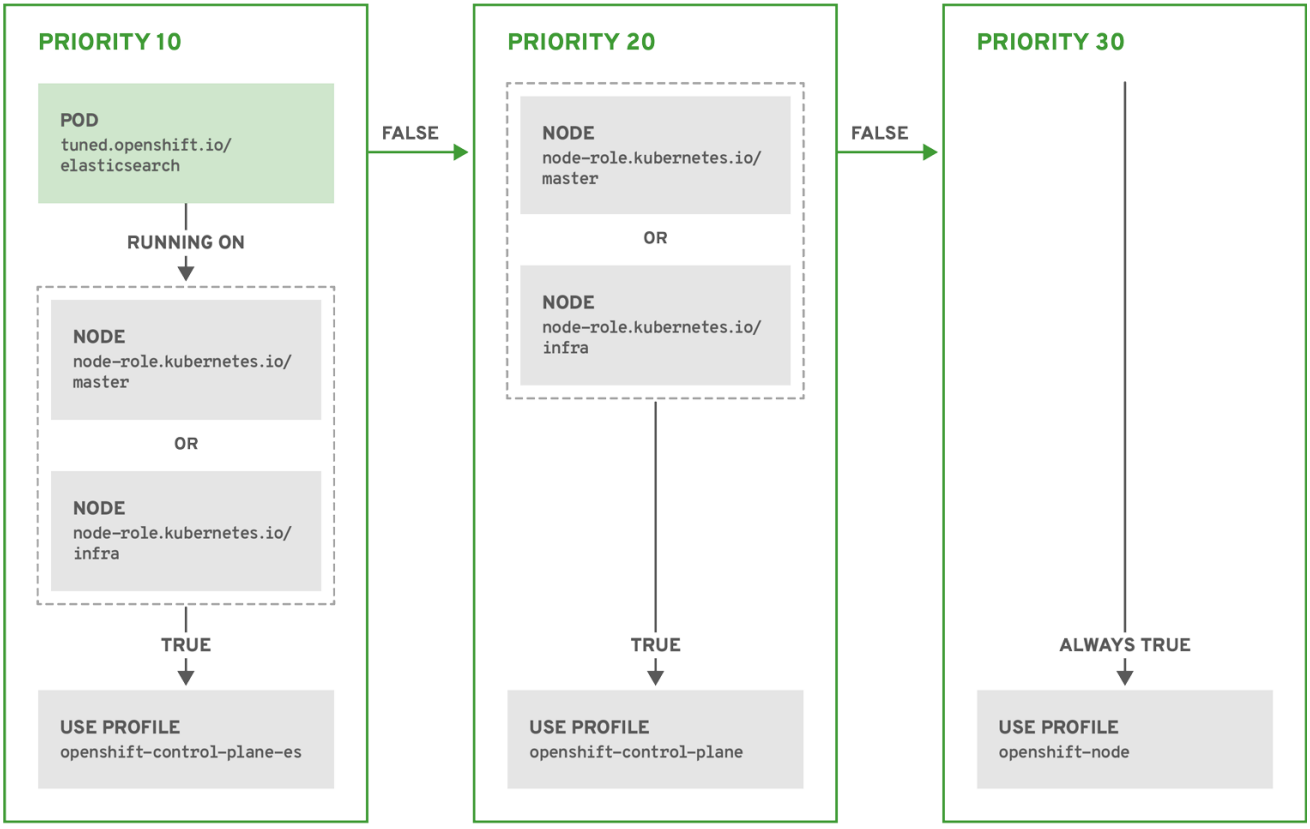
```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra
  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。



OPENSHIFT_10_0319

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

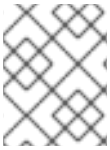
通过此功能，所有针对于 OpenShift Dedicated 集群上的云供应商都可以方便地分配 TuneD 配置集。这可实现，而无需添加额外的节点标签或将节点分组到机器配置池中。

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。

6.2.3. 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 Dedicated 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 ^ {} \;
```

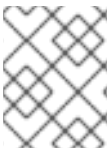
6.2.4. 支持的 TuneD 守护进程插件

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

6.3. 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。

第 7 章 操作容器

7.1. 了解容器

OpenShift Dedicated 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 Dedicated 和 Kubernetes 添加了在多主机安装间编配容器的功能。

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。

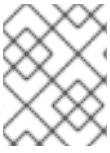
保存在这些缓存中的内存量与系统使用的 CPU 数量成比例。因此，有大量 CPU 会导致更多的内核内存被保存在这些缓存中。这些缓存中有大量内核内存可能会导致 OpenShift Dedicated 容器超过配置的内存限值，从而导致容器被终止。

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 Dedicated 文档使用术语 *容器运行时（container runtime）* 代表低级别容器运行时。其他文档可以将容器引擎引用为容器运行时。

OpenShift Dedicated 使用 CRI-O 作为容器引擎，runC 或 crun 作为容器运行时。默认容器运行时为 runC。

7.2. 在部署 POD 前使用初始容器来执行任务

OpenShift Dedicated 提供 *init* 容器，它们是在应用程序容器之前运行的专用容器，可以包含应用程序镜像中不存在的工具或设置脚本。

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 Dedicated 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. 通过 OpenShift Dedicated CLI 使用卷

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。

选项	描述	默认
-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。	
--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

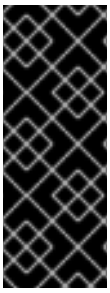
注意

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 Dedicated 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

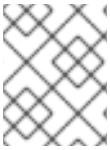
注意

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 对象信息的机制，而无需与 OpenShift Dedicated 耦合。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

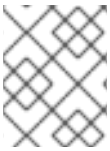
在创建 pod 时，您可以使用 Downward API 注入关于计算资源请求和限制的信息，以便镜像和应用程序作者能够正确地为特定环境创建镜像。

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
# ...
```

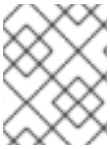
   b. 从 **pod.yaml** 文件创建 pod：

```
$ 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 DEDICATED 容器或从 OPENSHIFT DEDICATED 容器复制

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

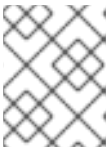
CLI 中内置了将本地文件复制到容器或从容器中复制文件的支持。

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 DEDICATED 容器中执行远程命令

您可以使用 CLI 在 OpenShift Dedicated 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：

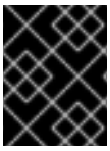
```
$ oc exec <pod> [-c <container>] -- <command> [<arg_1> ... <arg_n>]
```

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 Dedicated 支持向 pod 转发端口转发。

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 Dedicated 处理来自客户端的端口转发请求。在收到请求后，OpenShift Dedicated 会升级响应并等待客户端创建端口转发流。当 OpenShift Dedicated 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 Dedicated 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。

第 8 章 操作集群

8.1. 查看 OPENSHIFT DEDICATED 集群中的系统事件信息

OpenShift Dedicated 中的事件根据 OpenShift Dedicated 集群中 API 对象的事件建模。

8.1.1. 了解事件

事件允许 OpenShift Dedicated 以与资源无关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 Dedicated 控制台查看项目中的事件。

1. 启动 OpenShift Dedicated 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 Dedicated 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>**。

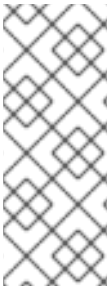
名称	描述
ExternalIP	列出外部 IP 地址。例如，**Added: <external-ip>**。
UID	列出新 UID。例如，**<old-service-uid>** → **<new-service-uid>**。
ExternalTrafficPolicy	列出新 **ExternalTrafficPolicy**。例如，**<old-policy>** → **<new-policy>**。
HealthCheckNodePort	列出新 **HealthCheckNodePort**。例如，**<old-node-port>** → **new-node-port>**。
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 DEDICATED 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 Dedicated 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 Dedicated 集群中使用未绑定的计算资源运行。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 Dedicated 通过将 **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。

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，建议阅读 OpenShift Dedicated 如何管理计算资源的概述。

对于每种资源（内存、CPU、存储），OpenShift Dedicated 允许将可选**请求**和**限制**值放在 pod 中的每个容器上。

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 Dedicated 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 Dedicated 会优先驱除其内存用量超过其内存请求的容器。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

在 OpenShift Dedicated 上调整应用程序内存大小的步骤如下：

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。

   请注意，有些 OpenShift Dedicated 集群可能需要设置限制值；有些集群可能会根据限制覆盖请求；有些应用程序镜像依赖于设置的限制，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。

8.4.2. 了解 OpenShift Dedicated 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。

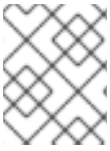
8.4.2.3. 了解如何确保正确配置容器中的所有 JVM 进程

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 Dedicated 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 Dedicated 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

当节点的内存用尽时，OpenShift Dedicated 可以从其节点中驱除 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 Dedicated 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（node overcommit）一起使用，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 Dedicated 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 Dedicated 控制台或 CLI 安装 Cluster Resource Override Operator，如以下部分所示。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：

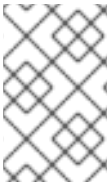
```
apiVersion: operator.autoscaling.openshift.io/v1
kind: ClusterResourceOverride
metadata:
  name: cluster (1)
spec:
  podResourceOverride:
    spec:
      memoryRequestToLimitPercent: 50 (2)
      cpuRequestToLimitPercent: 25 (3)
      limitCPUToMemoryPercent: 200 (4)
# ...
```

(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 Dedicated Web 控制台安装 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 Dedicated Web 控制台安装 Cluster Resource Override Operator：

1. 在 OpenShift Dedicated 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","met
```

```
adata":{"annotations":{},"name":"cluster"},"spec":{"podResourceOverride":{"spec":
{"cpuRequestToLimitPercent":25,"limitCPUToMemoryPercent":200,"memoryRequestToLi
mitPercent":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 Dedicated CLI 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"
  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 Dedicated 使用 **qos-reserved** 参数，如下所示：

- 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 Dedicated 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 Dedicated 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
```

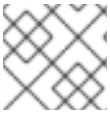
输出示例

```
#...
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. 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。