



OpenShift Container Platform 4.3

节点

在 OpenShift Container Platform 中配置和管理节点

OpenShift Container Platform 4.3 节点

在 OpenShift Container Platform 中配置和管理节点

法律通告

Copyright © 2020 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文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 使用 POD	4
1.1. 使用 POD	4
1.2. 查看 POD	6
1.3. 为 POD 配置 OPENSIFT CONTAINER PLATFORM 集群	8
1.4. 自动缩放 POD	12
1.5. 为 POD 提供敏感数据	23
1.6. 使用设备插件来利用 POD 访问外部资源	30
1.7. 在 POD 调度决策中纳入 POD 优先级	33
1.8. 使用节点选择器将 POD 放置到特定节点	37
第 2 章 控制节点上的 POD 放置 (调度)	41
2.1. 使用调度程序控制 POD 放置	41
2.2. 配置默认调度程序以控制 POD 放置	42
2.3. 使用关联性和反关联性规则相对于其他 POD 放置 POD	57
2.4. 使用节点关联性规则控制节点上的 POD 放置	64
2.5. 将 POD 放置到过量使用的节点	69
2.6. 使用节点污点控制 POD 放置	70
2.7. 使用节点选择器将 POD 放置到特定节点	77
第 3 章 使用作业和 DAEMONSET	89
3.1. 使用 DAEMONSET 在节点上自动运行后台任务	89
3.2. 使用作业在 POD 中运行任务	91
第 4 章 操作节点	98
4.1. 查看和列出 OPENSIFT CONTAINER PLATFORM 集群中的节点	98
4.2. 操作节点	102
4.3. 管理节点	109
4.4. 管理每个节点的 POD 数量上限	111
4.5. 使用 NODE TUNING OPERATOR	113
4.6. 了解节点重新引导	119
4.7. 使用垃圾回收释放节点资源	120
4.8. 为 OPENSIFT CONTAINER PLATFORM 集群中的节点分配资源	124
4.9. 查看节点审计日志	127
4.10. 机器配置守护进程指标	131
第 5 章 操作容器	135
5.1. 了解容器	135
5.2. 在部署 POD 前使用初始容器来执行任务	135
5.3. 使用卷来持久保留容器数据	137
5.4. 使用投射卷来映射卷	143
5.5. 允许容器消耗 API 对象	149
5.6. 将文件复制到 OPENSIFT CONTAINER PLATFORM 容器或从中复制	157
5.7. 在 OPENSIFT CONTAINER PLATFORM 容器中执行远程命令	159
5.8. 使用端口转发访问容器中的应用程序	160
5.9. 在容器中使用 SYSCTL	162
第 6 章 操作集群	167
6.1. 查看 OPENSIFT CONTAINER PLATFORM 集群中的系统事件信息	167
6.2. 估算 OPENSIFT CONTAINER PLATFORM 节点可以容纳的 POD 数量	176
6.3. 设置限制范围	180
6.4. 配置集群内存以满足容器内存和风险要求	187
6.5. 配置集群以将 POD 放置到过量使用的节点上	193

第 1 章 使用 POD

1.1. 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。

1.1.1. 了解 pod

对容器而言，Pod 大致相当于一个机器实例（物理或虚拟）。每个 pod 分配有自己的内部 IP 地址，因此拥有完整的端口空间，并且 pod 内的容器可以共享其本地存储和网络。

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

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



注意

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



警告

不受复制控制器管理的裸机 pod 不能在节点中断时重新调度。

1.1.2. pod 配置示例

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

以下示例定义中的 pod 提供一个长时间运行的服务，该服务实际上是 OpenShift Container Platform 基础架构的一部分，即集成的容器镜像 registry。它展示了 pod 的许多特性，其中大多数已在其他主题中阐述，因此这里仅简略提及：

Pod 对象定义 (YAML)

```
kind: Pod
apiVersion: v1
metadata:
  name: example
  namespace: default
  selfLink: /api/v1/namespaces/default/pods/example
  uid: 5cc30063-0265780783bc
  resourceVersion: '165032'
  creationTimestamp: '2019-02-13T20:31:37Z'
```



```
labels: 1
  app: hello-openshift
annotations:
  openshift.io/scc: anyuid
spec:
  restartPolicy: Always 2
  serviceAccountName: default
  imagePullSecrets:
    - name: default-dockercfg-5zrhb
  priority: 0
  schedulerName: default-scheduler
  terminationGracePeriodSeconds: 30
  nodeName: ip-10-0-140-16.us-east-2.compute.internal
  securityContext: 3
    seLinuxOptions:
      level: 's0:c11,c10'
  containers: 4
    - resources: {}
      terminationMessagePath: /dev/termination-log
      name: hello-openshift
      securityContext:
        capabilities:
          drop:
            - MKNOD
        procMount: Default
      ports:
        - containerPort: 8080
          protocol: TCP
      imagePullPolicy: Always
      volumeMounts: 5
        - name: default-token-wbqsl
          readOnly: true
          mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      terminationMessagePolicy: File
      image: registry.redhat.io/openshift4/ose-ogging-eventrouter:v4.3 6
  serviceAccount: default 7
  volumes: 8
    - name: default-token-wbqsl
      secret:
        secretName: default-token-wbqsl
        defaultMode: 420
  dnsPolicy: ClusterFirst
status:
  phase: Pending
  conditions:
    - type: Initialized
      status: 'True'
      lastProbeTime: null
      lastTransitionTime: '2019-02-13T20:31:37Z'
    - type: Ready
      status: 'False'
      lastProbeTime: null
      lastTransitionTime: '2019-02-13T20:31:37Z'
      reason: ContainersNotReady
      message: 'containers with unready status: [hello-openshift]'
```

```

- type: ContainersReady
  status: 'False'
  lastProbeTime: null
  lastTransitionTime: '2019-02-13T20:31:37Z'
  reason: ContainersNotReady
  message: 'containers with unready status: [hello-openshift]'
- type: PodScheduled
  status: 'True'
  lastProbeTime: null
  lastTransitionTime: '2019-02-13T20:31:37Z'
hostIP: 10.0.140.16
startTime: '2019-02-13T20:31:37Z'
containerStatuses:
- name: hello-openshift
  state:
    waiting:
      reason: ContainerCreating
  lastState: {}
  ready: false
  restartCount: 0
  image: openshift/hello-openshift
  imageID: "
qosClass: BestEffort

```

- 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。本例中的一个标签是 **registry=default**。
- 2 pod 重启策略，可能的值有 **Always**、**OnFailure** 和 **Never**。默认值为 **Always**。
- 3 OpenShift Container Platform 为容器定义了一个安全上下文，指定是否允许其作为特权容器来运行，或者以所选用户身份运行，等等。默认上下文的限制性比较强，但管理员可以根据需要进行修改。
- 4 **containers** 指定包括一个或多个容器定义的数组。
- 5 容器指定在容器中挂载外部存储卷的位置。在本例中，有一个卷可用来存储对凭证的访问，该卷是根据 registry 对 OpenShift Container Platform API 发出请求所需的。
- 6 pod 中的每个容器使用自己的容器镜像进行实例化。
- 7 pod 对 OpenShift Container Platform API 发出请求是一种比较常见的模式，利用一个 **serviceAccount** 字段指定 pod 在发出请求时使用哪个服务帐户用户来进行身份验证。这可以为自定义基础架构组件提供精细的访问控制。
- 8 pod 定义了可供其容器使用的存储卷。在本例中，它提供了一个用于存储 registry 的临时卷，以及一个包含服务帐户凭证的 **secret** 卷。



注意

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

1.2. 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。

1.2.1. 关于 pod

OpenShift Container Platform 使用 Kubernetes 的 *pod* 概念，它是共同部署在同一主机上的一个或多个容器，也是可被定义、部署和管理的最小计算单元。对容器而言，Pod 大致相当于机器实例（物理或虚拟）。

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

1.2.2. 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 -n openshift-console
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

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>
```

1.2.3. 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
NAME                CPU(cores)  MEMORY(bytes)
console-7f58c69899-q8c8k  0m          22Mi
console-7f58c69899-xhbgg  0m          25Mi
downloads-594fccf94-bcxk8  3m          18Mi
downloads-594fccf94-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 `!=`。

1.3. 为 POD 配置 OPENSIFT CONTAINER PLATFORM 集群

作为管理员，您可以为 pod 创建和维护高效的集群。

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

1.3.1. 配置 pod 重启后的行为

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

可能的值有：

- **Always** - 在 pod 被重启之前，按规定的延时值（10s，20s，40s）不断尝试重启 pod 中成功退出的容器。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

状况	控制器类型	重启策略
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	Daemonset	任意

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

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



注意

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

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

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

1.3.2. 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>
```

1.3.3. 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 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。它们允许在操作过程中指定 pod 的安全约束，比如为维护而清空节点。

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

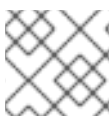
允许 **maxUnavailable** 为 **0%** 或 **0**，**minAvailable** 为 **100%** 或等于副本数，但这样设置可能会阻止节点排空操作。

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
```

NAMESPACE	NAME	MIN-AVAILABLE	SELECTOR
another-project	another-pdb	4	bar=foo
test-project	my-pdb	2	foo=bar

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。

1.3.3.1. 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/v1beta1 ❶
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2 ❷
  selector: ❸
    matchLabels:
      foo: bar
```

- ❶ **PodDisruptionBudget** 是 **policy/v1beta1** API 组的一部分。
- ❷ 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%**）。
- ❸ 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。

或者：

```
apiVersion: policy/v1beta1 ❶
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  maxUnavailable: 25% ❷
  selector: ❸
    matchLabels:
      foo: bar
```

- ❶ **PodDisruptionBudget** 是 **policy/v1beta1** API 组的一部分。
- ❷ 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%**）。
- ❸ 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。

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>
```

1.3.4. 使用关键 pod 防止删除 pod

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

标记为关键 (critical) 的 Pod 不允许被驱逐。

流程

使 pod 成为关键 pod :

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

```
spec:
  template:
    metadata:
      name: critical-pod
      priorityClassName: system-cluster-critical 1
```

- 1 绝不可从节点驱逐的 pod 的默认优先级类。

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

1. 创建 pod :

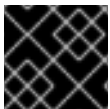
```
$ oc create -f <file-name>.yaml
```

1.4. 自动缩放 POD

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

1.4.1. 了解 pod 横向自动扩展

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



重要

根据内存使用率自动缩放仅是一项技术预览功能。

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

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

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

要使用 pod 横向自动扩展, 您的集群管理员必须已经正确配置了集群指标。

1.4.1.1. 支持的指标

pod 横向自动扩展支持以下指标 :

表 1.1. 指标

指标	描述	API 版本
CPU 使用率	已用的 CPU 内核数。可以用来计算 pod 的已请求 CPU 百分比。	autoscaling/v1、autoscaling/v2beta2
内存使用率	已用内存量。可以用来计算 pod 的已请求内存百分比。	autoscaling/v2beta2

重要

对于基于内存的自动缩放，内存用量必须与副本数呈正比增大和减小。平均而言：

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

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

1.4.2. 基于 CPU 使用率创建 pod 横向自动扩展

您可以为现有的 DeploymentConfig 或 ReplicationController 对象创建一个 pod 横向自动扩展 (HPA)，用于自动扩展与该对象关联的 Pod，以维护您指定的 CPU 用量。

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

为 CPU 使用率自动扩展时，您可以使用 **oc autoscale** 命令，并指定要在任意给定时间运行的 Pod 的最小和最大数量，以及 Pod 的目标平均 CPU 使用率。如果未指定最小值，则 OpenShift Container Platform 服务器会为 Pod 赋予一个默认值。要自动缩放特定 CPU 值，创建一个带有目标 CPU 和 Pod 限制的 **HorizontalPodAutoscaler** 对象。

先决条件

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

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

```
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-
kubernetes-scheduler-ip-10-0-135-131.ec2.internal
  Timestamp:         2019-05-23T18:47:56Z
  Window:            1m0s
  Events:            <none>

```

流程

为 CPU 使用率创建 pod 横向自动扩展

1. 执行以下步骤之一：

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

```

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

```

- 1 指定 DeploymentConfig 的名称。对象必须存在。
- 2 另外，还可以指定缩减时的最小副本数量。
- 3 指定扩展时的最大副本数量。
- 4 指定所有 pod 的目标平均 CPU 使用率（以请求 CPU 的百分比表示）。如果未指定或为负数，则会使用默认的自动缩放策略。

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

```

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

```

- 1 指定 ReplicationController 的名称。对象必须存在。
- 2 指定缩减时的最小副本数量。
- 3 指定扩展时的最大副本数量。
- 4 指定所有 pod 的目标平均 CPU 使用率（以请求 CPU 的百分比表示）。如果未指定或为负数，则会使用默认的自动缩放策略。

- 要扩展特定 CPU 值，请为现有 DeploymentConfig 或 ReplicationController 创建类似如下的一个 YAML 文件：

a. 创建一个类似以下示例的 YAML 文件：

```

apiVersion: autoscaling/v2beta2 ❶
kind: HorizontalPodAutoscaler
metadata:
  name: cpu-autoscale ❷
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: v1 ❸
    kind: ReplicationController ❹
    name: example ❺
  minReplicas: 1 ❻
  maxReplicas: 10 ❼
  metrics: ❽
  - type: Resource
    resource:
      name: cpu ❾
      target:
        type: Utilization ❿
        averageValue: 500m ⓫

```

- ❶ 使用 **autoscaling/v2beta2** API。
- ❷ 指定此 pod 横向自动扩展对象的名称。
- ❸ 指定要缩放对象的 API 版本。
 - 对于 ReplicationController，使用 **v1**，
 - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
- ❹ 指定要缩放的对象类型，可以是 **ReplicationController** 或 **DeploymentConfig**。
- ❺ 指定要缩放的对象名称。对象必须存在。
- ❻ 指定缩减时的最小副本数量。
- ❼ 指定扩展时的最大副本数量。
- ❽ 对于内存使用率，使用 **metrics** 参数。
- ❾ 为 CPU 使用率指定 **cpu**。
- ❿ 设置 **Utilization**。
- ⓫ 将类型设置为 **averageValue**。

b. 创建 Pod 横向自动扩展：

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

2. 验证 pod 横向自动扩展是否已创建：

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

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

例如，以下命令会创建一个 Pod 横向自动扩展，维护由 **image-registry** DeploymentConfig 控制的 3 到 7 个 Pod 副本，以便确保所有 Pod 的平均 CPU 使用率为 75%。

```
$ oc autoscale dc/image-registry --min 3 --max 7 --cpu-percent=75
deploymentconfig "image-registry" autoscaled
```

该命令创建带有以下定义的 pod 横向自动扩展：

```
$ oc edit hpa frontend -n openshift-image-registry
```

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  creationTimestamp: "2020-02-21T20:19:28Z"
  name: image-registry
  namespace: default
  resourceVersion: "32452"
  selfLink: /apis/autoscaling/v1/namespaces/default/horizontalpodautoscalers/frontend
  uid: 1a934a22-925d-431e-813a-d00461ad7521
spec:
  maxReplicas: 7
  minReplicas: 3
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1
    kind: DeploymentConfig
    name: image-registry
    targetCPUUtilizationPercentage: 75
status:
  currentReplicas: 5
  desiredReplicas: 0
```

下列演示了 **image-registry** DeploymentConfig 的自动缩放。初始部署需要 3 个 Pod。HPA 对象将最小值增加到 5，如果 pod 的 CPU 用量达到 75%，会将 pod 数最高增加到 7：

```
$ oc get dc image-registry
NAME          REVISION  DESIRED  CURRENT  TRIGGERED BY
image-registry 1          3        3        config
```

```
$ oc autoscale dc/image-registry --min=5 --max=7 --cpu-percent=75
horizontalpodautoscaler.autoscaling/image-registry autoscaled
```

```
$ oc get dc image-registry
NAME          REVISION  DESIRED  CURRENT  TRIGGERED BY
image-registry 1          5        5        config
```

1.4.3. 基于内存使用率创建 pod 横向自动扩展对象

您可以为现有 DeploymentConfig 或 ReplicationController 对象创建一个 pod 横向自动扩展 (HPA)，用于自动扩展与该对象关联的 Pod，以便维护您指定的平均内存使用率（可以是一个直接的值，也可以是请求的内存百分比）。

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

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



重要

根据内存使用率自动缩放仅是一项技术预览功能。技术预览功能不包括在红帽生产服务级别协议 (SLA) 中，且其功能可能并不完善。因此，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

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

先决条件

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

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

```
Name:      openshift-kube-scheduler-ip-10-0-129-223.compute.internal
Namespace: openshift-kube-scheduler
Labels:    <none>
Annotations: <none>
API Version: metrics.k8s.io/v1beta1
Containers:
  Name: scheduler
  Usage:
    Cpu: 2m
    Memory: 41056Ki
  Name: wait-for-host-port
  Usage:
    Memory: 0
Kind:      PodMetrics
Metadata:
  Creation Timestamp: 2020-02-14T22:21:14Z
  Self Link:         /apis/metrics.k8s.io/v1beta1/namespaces/openshift-kube-scheduler/pods/openshift-kube-scheduler-ip-10-0-129-223.compute.internal
  Timestamp:        2020-02-14T22:21:14Z
  Window:           5m0s
  Events:           <none>
```

流程

根据内存使用率创建 pod 横向自动扩展：

1. 为以下之一创建一个 YAML 文件：

- 要扩展特定内存值，请为现有 DeploymentConfig 或 ReplicationController 创建类似如下的 **HorizontalPodAutoscaler** 对象：

```

apiVersion: autoscaling/v2beta2 ❶
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory ❷
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: v1 ❸
    kind: ReplicationController ❹
    name: example ❺
  minReplicas: 1 ❻
  maxReplicas: 10 ❼
  metrics: ❽
  - type: Resource
    resource:
      name: memory ❾
      target:
        type: AverageValue ❿
        averageValue: 500Mi ⓫

```

- ❶ 使用 **autoscaling/v2beta2** API。
- ❷ 指定此 pod 横向自动扩展对象的名称。
- ❸ 指定要缩放对象的 API 版本。
 - 对于 ReplicationController，使用 **v1**，
 - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
- ❹ 指定要缩放的对象类型，可以是 **ReplicationController** 或 **DeploymentConfig**。
- ❺ 指定要缩放的对象名称。对象必须存在。
- ❻ 指定缩减时的最小副本数量。
- ❼ 指定扩展时的最大副本数量。
- ❽ 对于内存使用率，使用 **metrics** 参数。
- ❾ 为内存使用率指定 **memory**。
- ❿ 将类型设置为 **AverageValue**。
- ⓫ 指定 **averageValue** 和一个特定的内存值。

- 要缩放一个百分比，创建一个与以下类似的 **HorizontalPodAutoscaler** 对象：

```

apiVersion: autoscaling/v2beta2 ❶
kind: HorizontalPodAutoscaler
metadata:

```

```

name: memory-autoscale ❷
namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps.openshift.io/v1 ❸
    kind: DeploymentConfig ❹
    name: example ❺
  minReplicas: 1 ❻
  maxReplicas: 10 ❼
  metrics: ❽
  - type: Resource
    resource:
      name: memory ❾
      target:
        type: Utilization ❿
        averageUtilization: 50 ⓫

```

- ❶ 使用 **autoscaling/v2beta2** API。
- ❷ 指定此 pod 横向自动扩展对象的名称。
- ❸ 指定要缩放对象的 API 版本。
 - 对于 ReplicationController，使用 **v1**，
 - 对于 DeploymentConfig，使用 **apps.openshift.io/v1**。
- ❹ 指定要缩放的对象类型，可以是 **ReplicationController** 或 **DeploymentConfig**。
- ❺ 指定要缩放的对象名称。对象必须存在。
- ❻ 指定缩减时的最小副本数量。
- ❼ 指定扩展时的最大副本数量。
- ❽ 对于内存使用率，使用 **metrics** 参数。
- ❾ 为内存使用率指定 **memory**。
- ❿ 设置 **Utilization**。
- ⓫ 为所有 Pod 指定 **averageUtilization** 和一个目标平均内存利用率，以请求内存的百分比表示。目标 pod 必须配置内存请求。

2. 创建 Pod 横向自动扩展：

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

例如：

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

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

3. 验证 pod 横向自动扩展是否已创建：

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

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS
REPLICAS AGE				
1 20m	ReplicationController/example	2441216/500Mi	1	10

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

```
Name: hpa-resource-metrics-memory
Namespace: default
Labels: <none>
Annotations: <none>
CreationTimestamp: Wed, 04 Mar 2020 16:31:37 +0530
Reference: ReplicationController/example
Metrics: ( current / target )
  resource memory on pods: 2441216 / 500Mi
Min replicas: 1
Max replicas: 10
ReplicationController pods: 1 current / 1 desired
Conditions:
  Type          Status Reason          Message
  ----          -
  AbleToScale   True   ReadyForNewScale   recommended size matches current size
  ScalingActive True   ValidMetricFound   the HPA was able to successfully calculate a
  replica count from memory resource
  ScalingLimited False  DesiredWithinRange the desired count is within the acceptable
  range
Events:
  Type    Reason          Age          From          Message
  ----    -
  Normal  SuccessfulRescale 6m34s       horizontal-pod-autoscaler New size: 1;
  reason: All metrics below target
```

1.4.4. 了解 pod 横向自动扩展状态条件

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

HPA 状态条件可通过 **v2beta1** 版的自动扩展 API 使用。

HPA 可以通过下列状态条件给予响应：

- **AbleToScale** 条件指示 HPA 是否能够获取和更新指标，以及是否有任何与退避相关的条件阻碍了缩放。
 - **True** 条件表示允许缩放。
 - **False** 条件表示因为指定原因不允许缩放。
- **ScalingActive** 条件指示 HPA 是否已启用（例如，目标的副本数不为零），并且可以计算所需的指标。
 - **True** 条件表示指标工作正常。

- **False** 条件通常表示获取指标时出现问题。
- **ScalingLimited** 条件表示所需的规模由 pod 横向自动扩展限定最大或最小限制。
 - **True** 条件表示您需要提高或降低最小或最大副本数才能进行缩放。
 - **False** 条件表示允许请求的缩放。

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

1 pod 横向自动扩展状态消息。

下列中是一个无法缩放的 pod :

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

下列中是一个无法获得缩放所需指标的 pod :

```
Conditions:
  Type           Status Reason           Message
  ----           -
  AbleToScale    True   SucceededGetScale the HPA controller was able to get the target's
```

current scale

```
ScalingActive      False  FailedGetResourceMetric  the HPA was unable to compute the replica
count: unable to get metrics for resource cpu: no metrics returned from heapster
```

下例中是一个请求的自动缩放低于所需下限的 pod :

Conditions:

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

1.4.4.1. 查看 pod 横向自动扩展状态条件

您可以查看 pod 横向自动扩展 (HPA) 对 pod 设置的状态条件。



注意

pod 横向自动扩展状态条件可通过 **v2beta1** 版的自动扩展 API 使用。

先决条件

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

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

```
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-
kubernetes-scheduler-ip-10-0-135-131.ec2.internal
Timestamp:         2019-05-23T18:47:56Z
Window:            1m0s
Events:            <none>
```

流程

要查看 pod 上的状态条件，请使用以下命令并提供 pod 的名称：

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

例如：

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

这些条件会出现在输出中的 **Conditions** 字段里。

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

1.4.5. 其他资源

如需有关复制控制器和部署控制器的更多信息，请参见[了解 Deployment](#) 和 [DeploymentConfig](#)。

1.5. 为 POD 提供敏感数据

有些应用程序需要密码和用户名等敏感信息，但您不希望开发人员持有这些信息。

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

1.5.1. 了解 secret

Secret 对象类型提供了一种机制来保存敏感信息，如密码、OpenShift Container Platform 客户端配置文件和私有源存储库凭证等。Secret 将敏感内容与 pod 分离。您可以使用卷插件将 secret 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: dmFsdWUtMQ0K ③
  password: dmFsdWUtMg0KDQo=
stringData: ④
  hostname: myapp.mydomain.com ⑤

```

- ① ① 指示 secret 密钥名称和值的结构。
- 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。

1.5.1.1. 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/service-account-token**。使用服务帐户令牌。
- **kubernetes.io/basic-auth**。搭配基本身份验证使用。
- **kubernetes.io/ssh-auth**。搭配 SSH 密钥身份验证使用。
- **kubernetes.io/tls**。搭配 TLS 证书颁发机构使用。

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。

如需不同 secret 类型的示例，请参阅 *使用 secret* 中的代码示例。

1.5.1.2. secret 配置示例

以下是几个 secret 配置文件示例。

创建四个文件的 Secret YAML

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: dmFsdWUtMQ0K 1
  password: dmFsdWUtMQ0KDQo= 2
stringData:
  hostname: myapp.mydomain.com 3
secret.properties: |- 4
  property1=valueA
  property2=valueB
```

- 1** 文件包含已解码的值。
- 2** 文件包含已解码的值。
- 3** 文件包含提供的字符串。
- 4** 文件包含提供的数据。

一个 Pod 的 YAML 定义，使用卷中的 secret 数据。

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  volumes:
    - name: secret-volume
```

```
secret:
  secretName: test-secret
restartPolicy: Never
```

一个 Pod 的 YAML 定义，在环境变量中使用 secret 数据

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
  - name: secret-test-container
    image: busybox
    command: [ "/bin/sh", "-c", "export" ]
    env:
    - name: TEST_SECRET_USERNAME_ENV_VAR
      valueFrom:
        secretKeyRef:
          name: test-secret
          key: username
  restartPolicy: Never
```

一个 Build Config 的 YAML 定义，在环境变量中使用 secret 数据。

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: TEST_SECRET_USERNAME_ENV_VAR
        valueFrom:
          secretKeyRef:
            name: test-secret
            key: username
```

1.5.1.3. Secret 数据密钥

Secret 密钥必须在 DNS 子域中。

1.5.2. 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：

- 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。

1.5.2.1. 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：

- 容器的环境变量。
- 作为挂载到一个或多个容器上的卷中的文件。
- 在拉取 pod 的镜像时由 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 也可能耗尽内存。

1.5.2.2. 创建不透明 secret

作为管理员，您可以创建不透明 secret，从而能使用包含任意值的无结构 **key:value** 对。

流程

1. 在主机上的 YAML 文件中创建 secret 对象。

例如：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque ❶
data:
  username: dXNlci1uYW1l
  password: cGFzc3dvcmQ=
```

- ❶ 指定不透明 secret。

2. 使用以下命令来创建 secret 对象：

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

然后：

1. 为您要使用 secret 来允许引用该 secret 的 pod 更新服务帐户。
2. 创建以环境变量或文件（使用 **secret** 卷）形式消耗 secret 的 pod。

1.5.3. 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。

1.5.4. 关于将签名证书与 secret 搭配使用

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

服务用证书 secret 旨在支持需要开箱即用证书的复杂中间件应用程序。它的设置与管理员工具为节点和主控机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.alpha.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。

1.5.4.1. 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.alpha.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 规格。
2. 使用您要用于 secret 的名称，添加 **service.alpha.openshift.io/serving-cert-secret-name** 注解。

```
kind: Service
```



```

apiVersion: v1
metadata:
  name: my-service
  annotations:
    service.alpha.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：

```

$ oc get secrets

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

$ oc describe secret my-service-pod
Name:         my-service-pod
Namespace:    openshift-console
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: builder
              kubernetes.io/service-account.uid: ab-11e9-988a-0eb4e1b4a396

Type: kubernetes.io/service-account-token

Data

ca.crt:      5802 bytes
namespace:   17 bytes
token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50li
wia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9uYW1lc3BhY2UiOiJvcGVuc2hpdjY29uc
29sZSIsImt1YmVyb
cnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC51aWQiOiJhYmE4Y2UyZC00MzVILTEyZTkt
OTg4YS0wZWl0ZTFiNGEz
OTYiLCJzdWliOiJzeXN0ZW06c2VydmljZWJjY291bnQ6b3BlbnNoaWZ

```

5. 编辑与该 secret 搭配的 pod 规格。

```

apiVersion: v1
kind: Pod
metadata:
  name: my-service-pod
spec:
  containers:

```

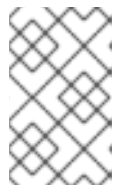
```

- name: mypod
  image: redis
  volumeMounts:
  - name: foo
    mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: my-cert
      items:
      - key: username
        path: my-group/my-username
        mode: 511

```

当它可用时，您的 pod 将会运行。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.alpha.openshift.io/expiry** 注解中查看到期日期，其采用 RFC3339 格式。



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。

1.5.5. 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.alpha.openshift.io/serving-cert-generation-error** 注解包含以下信息）：

```
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**。您必须移除旧 secret 并清除服务注解 **service.alpha.openshift.io/serving-cert-generation-error** 和 **service.alpha.openshift.io/serving-cert-generation-error-num**，以强制重新生成证书：

```

$ oc delete secret <secret_name>
$ oc annotate service <service_name> service.alpha.openshift.io/serving-cert-generation-error-
$ oc annotate service <service_name> service.alpha.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 -。

1.6. 使用设备插件来利用 POD 访问外部资源

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

1.6.1. 了解设备插件

设备插件提供一致并可移植的解决方案，以便跨集群消耗硬件设备。设备插件通过一种扩展机制提供对这些设备的支持，从而将设备提供给容器，提供设备健康检查，并且安全地共享设备。



重要

OpenShift Container Platform 支持设备插件 API，但设备插件容器则由各家供应商提供支持。

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

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

  // ListAndWatch returns a stream of List of Devices
  // Whenever a Device state change or a Device disappears, ListAndWatch
  // returns the new list
  rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

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

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

设备插件示例

- [适用于 COS 型操作系统的 Nvidia GPU 设备插件](#)
- [Nvidia 官方 GPU 设备插件](#)
- [Solarflare 设备插件](#)
- [KubeVirt 设备插件：vfio 和 kvm](#)



注意

对于简单设备插件参考实现，设备管理器代码中提供了一个存根设备插件：[vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go](https://github.com/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go)。

1.6.1.1. 设备插件部署方法

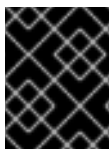
- **Daemonset** 是部署设备插件的推荐方法。
- 在启动时，设备插件会尝试在节点上的 `/var/lib/kubelet/device-plugin/` 创建一个 UNIX 域套接字，以服务来自于设备管理器的 RPC。

- 由于设备插件必须管理硬件资源、主机文件系统的访问权以及套接字创建，它们必须在一个特权安全上下文中运行。
- 各种设备插件实现中提供了有关部署步骤的更多细节。

1.6.2. 了解设备管理器

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

您可以公告专用的硬件，而不必修改任何上游代码。



重要

OpenShift Container Platform 支持设备插件 API，但设备插件容器则由各家供应商提供支持。

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

在启动时，设备插件通过在 `/var/lib/kubelet/device-plugins/kubelet.sock` 上调用 **Register** 将自身注册到设备管理器，再启动位于 `/var/lib/kubelet/device-plugins/<plugin>.sock` 的 gRPC 服务来服务设备管理器请求。

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

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

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

1.6.3. 启用设备管理器

启用设备管理器来实现设备插件，在不更改上游代码的前提下公告专用硬件。

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

1. 为您要配置的节点类型获取与静态机器配置池 CRD 关联的标签。执行以下步骤之一：
 - a. 查看机器配置：

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

例如：

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

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

```
Name:      00-worker
```

```
Namespace:
```

```
Labels:    machineconfiguration.openshift.io/role=worker 1
```

-

1 1 设备管理器所需的标签。

流程

1. 为配置更改创建自定义资源 (CR)。

设备管理器 CR 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: devicemgr 1
spec:
  machineConfigPoolSelector:
    matchLabels:
      machineconfiguration.openshift.io: devicemgr 2
  kubeletConfig:
    feature-gates:
      - DevicePlugins=true 3
```

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

2. 创建设备管理器：

```
$ oc create -f devicemgr.yaml

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

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

1.7. 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 优先级指明 pod 相对于其他 pod 的重要程度，并根据这个优先级对 pod 进行排队。Pod 抢占则允许集群驱除低优先级 pod 或与之争抢，从而在合适的节点上没有可用空间时能够调度优先级较高的 pod。pod 优先级也会影响 pod 的调度顺序以及节点上资源不足驱除顺序。

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

抢占通过调度程序配置文件中的 `disablePreemption` 参数来进行控制，该参数默认设置为 `false`。

1.7.1. 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。

1.7.1.1. 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。对于不应被抢占或驱除的关键 pod，可保留大于十亿的数值。默认情况下，OpenShift Container Platform 有两个保留优先级类，用于需要保证调度的关键系统 pod。

```
$ oc get priorityclasses
NAME                CREATED AT
cluster-logging     2019-03-13T14:45:12Z
system-cluster-critical 2019-03-13T14:01:10Z
system-node-critical  2019-03-13T14:01:10Z
```

- **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
- **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。



注意

如果升级现有的集群，则现有 pod 的优先级相当于为零。不过，带有 **scheduler.alpha.kubernetes.io/critical-pod** 注解的现有 pod 会自动转换为 **system-cluster-critical** 类。带有该注解的 Fluentd 集群日志 pod 转换为 **cluster-logging** 优先级类。

1.7.1.2. 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。

1.7.2. 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节点时，较高优先级 pod 规格的 **nominatedNodeName** 字段将设为该节点的名称，**nodename** 字段同样如此。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 规格的 **nominatedNodeName** 字段和 **nodeName** 字段可能会不同。

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。

1.7.2.1. Pod 抢占和其他调度程序设置

如果启用 pod 优先级与抢占功能，请考虑其他的调度程序设置：

pod 优先级和 pod 中断预算

pod 中断预算指定某一时间必须保持在线的副本的最小数量或百分比。如果您指定了 pod 中断预算，OpenShift Container Platform 会在抢占 pod 时尽力尊重这些预算。调度程序会尝试在不违反 pod 中断预算的前提下抢占 pod。如果找不到这样的 pod，则可能会无视 pod 中断预算要求而抢占较低优先级 pod。

pod 优先级和 pod 关联性

pod 关联性要求将新 pod 调度到与具有同样标签的其他 pod 相同的节点上。

如果待处理 pod 与节点上的一个或多个低优先级 pod 具有 pod 间关联性，调度程序就不能在不违反关联要求的前提下抢占较低优先级 pod。这时，调度程序会寻找其他节点来调度待处理 pod。但是，不能保证调度程序能够找到合适的节点，因此可能无法调度待处理 pod。

要防止这种情况，请仔细配置优先级相同的 pod 的 pod 关联性。

1.7.2.2. 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。

1.7.3. 配置优先级和抢占

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，可以应用 pod 优先级与抢占。

优先级类对象示例

```
apiVersion: scheduling.k8s.io/v1beta1
kind: PriorityClass
metadata:
  name: high-priority ①
value: 1000000 ②
globalDefault: false ③
description: "This priority class should be used for XYZ service pods only." ④
```

- ① 优先级类对象的名称。
- ② 对象的优先级值。
- ③ 此可选字段指定是否应该将这个优先级类用于 pod，而不指定优先级类名。此字段默认为 **false**。集群中只能存在一个 **globalDefault** 设为 **true** 的优先级类。如果没有 **globalDefault:true** 的优先级类，则无优先级类名称的 pod 的优先级为零。添加具有 **globalDefault:true** 的优先级类只会影响在添加优先级类后创建的 pod，不会更改现有 pod 的优先级。
- ④ 此可选任意文本字符串用于描述开发人员应对哪些 pod 使用这个优先级类。

流程

配置集群以使用优先级与抢占功能：

1. 创建一个或多个优先级类：
 - a. 指定优先级的名称和值。
 - b. （可选）指定优先级类的 **globalDefault** 字段和描述。
2. 创建 pod 规格或编辑现有的 pod 使其包含优先级类的名称，如下所示：

带有优先级类名称的 pod 规格示例

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
    priorityClassName: high-priority ①
```

- ① 指定要用于此 pod 的优先级类。

3. 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。

1.7.4. 禁用优先级与抢占

您可以禁用 pod 优先级与抢占功能。

禁用该功能后，现有 pod 会保留其优先级字段，但会禁用抢占功能，而且也会忽略优先级字段。如果禁用该功能，您便无法在新 pod 中设置优先级类名称。



重要

在集群面临资源压力时，关键 pod 依赖于调度程序抢占功能来进行调度。因此，红帽建议您不要禁用抢占。DaemonSet pod 由 DaemonSet 控制器调度，不受禁用抢占的影响。

流程

为集群禁用抢占：

1. 编辑调度程序 Operator 自定义资源以添加 **disablePreemption: true** 参数：

```
oc edit scheduler cluster

apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  creationTimestamp: '2019-03-12T01:45:02Z'
  generation: 1
  name: example
  resourceVersion: '1882034'
  selfLink: /apis/config.openshift.io/v1/schedulers/example
  uid: 743701e9-4468-11e9-bd34-02a7fe1bf828
spec:
  disablePreemption: true
```

1.8. 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。

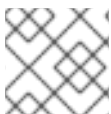
1.8.1. 使用节点选择器控制 pod 放置

您可以使用 pod 上的节点选择器标签来控制 pod 的调度位置。

使用节点选择器时，OpenShift Container Platform 会将 pod 调度到包含匹配标签的节点。

要向现有的 pod 添加节点选择器，请将节点选择器添加到该节点的控制对象，如 ReplicaSet、Daemonset 或 StatefulSet。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。

您可以向节点或 MachineConfig 添加标签，但标签不会在节点或机器停机后保留。将标签添加到 MachineSet 可确保新的节点或机器具有该标签。



注意

您不能直接将节点选择器添加到现有调度的 pod 中。

前提条件

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** ReplicaSet 控制：

```
$ oc describe pod router-default-66d5cf9464-7pwkc

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：

```
ownerReferences:
- apiVersion: apps/v1
  kind: ReplicaSet
  name: router-default-66d5cf9464
  uid: d81dd094-da26-11e9-a48a-128e7edf0312
  controller: true
  blockOwnerDeletion: true
```

流程

1. 通过使用 MachineSet 或直接编辑节点，为节点添加标签：
 - 在创建节点时，使用 MachineSet 将标签添加到由 MachineSet 管理的节点上：
 - a. 运行以下命令，将节点选择器添加到 MachineSet 中：

```
$ oc patch MachineSet <name> --type='json' -
p=[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>","<key>="<value>"}]}] -n openshift-machine-api
```

例如：

```
$ oc patch MachineSet abc612-msrtw-worker-us-east-1c --type='json' -
p=[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}] -n openshift-machine-api
```

- b. 使用 **oc edit** 命令验证标签是否已添加到 MachineSet:

例如：

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

MachineSet 对象示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
...
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
...

```

- 直接向节点添加标签：

- a. 为节点编辑 **Node** 对象：

```
$ oc label nodes <name> <key>=<value>
```

例如，若要为以下节点添加标签：

```
$ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
```

- b. 验证标签是否已添加到节点：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS ROLES  AGE  VERSION
ip-10-0-142-25.ec2.internal  Ready  worker  17m  v1.18.3+002a51f
```

2. 将匹配的节点选择器添加到 pod：

- 要将节点选择器添加到现有和未来的 pod，请向 pod 的控制对象添加节点选择器：

ReplicaSet 对象示例

```
kind: ReplicaSet
```

```

....
spec:
....
template:
  metadata:
    creationTimestamp: null
  labels:
    ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
    pod-template-hash: 66d5cf9464
  spec:
    nodeSelector:
      beta.kubernetes.io/os: linux
      node-role.kubernetes.io/worker: "
      type: user-node ❶

```

❶ 添加节点选择器。

- 要将节点选择器添加到特定的 pod，直接将选择器添加到 **Pod** 对象中：

Pod 对象示例

```

apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    <key>: <value>
...

```

例如：

使用节点选择器的 Pod 对象示例

```

apiVersion: v1
kind: Pod
....
spec:
  nodeSelector:
    region: east
    type: user-node

```

第 2 章 控制节点上的 POD 放置 (调度)

2.1. 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。

调度程序代码具有明确隔离，会监测创建的新 pod 并确定最适合托管它们的节点。然后，它会利用主 API 为 pod 创建 pod 至节点的绑定。

默认 pod 调度

OpenShift Container Platform 附带一个默认调度程序，能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 Container Platform 高级调度功能来配置 pod，从而使 pod 能够根据要求或偏好在特定的节点上运行，或者与特定的 pod 一起运行。

- 使用 [pod 关联性和反关联性规则](#)。
- 使用 [pod 关联性](#)控制 pod 放置。
- 使用 [节点关联性](#)控制 pod 放置。
- 将 pod 放置到[过量使用的节点](#)。
- 使用 [节点选择器](#)控制 pod 放置。
- 使用 [污点和容限](#)控制 pod 位置。

2.1.1. 调度程序用例

在 OpenShift Container Platform 中调度的一个重要用例是支持灵活的关联性和反关联性策略。

2.1.1.1. 基础架构拓扑级别

管理员可以通过在节点上指定标签，为基础架构（节点）定义多个拓扑级别。例如，**region=r1**、**zone=z1**、**rack=s1**。

这些标签名称没有特别的含义，管理员可以自由为其基础架构级别命名，比如城市/楼宇/房间。另外，管理员可以为其基础架构拓扑定义任意数量的级别，通常三个级别比较适当（例如：**regions** → **zone** → **racks**）。管理员可以在各个级别上以任何组合指定关联性和反关联性规则。

2.1.1.2. 关联性

管理员应能够配置调度程序，在任何一个甚至多个拓扑级别上指定关联性。特定级别上的关联性指示所有属于同一服务的 pod 调度到属于同一级别的节点。这会让管理员确保对等 pod 在地理上不会过于分散，以此处理应用程序对延迟的要求。如果同一关联性组中没有节点可用于托管 pod，则不调度该 pod。

如果需要更多地控制 pod 的调度位置，请参阅“使用节点关联性”和“使用 Pod 关联性和反关联性”。

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。

2.1.1.3. 反关联性

管理员应能够配置调度程序，在任何一个甚至多个拓扑级别上指定反关联性。特定级别上的反关联性（或分散）指示属于同一服务的所有 pod 分散到属于该级别的不同节点上。这样可确保应用程序合理分布，以实现高可用性目的。调度程序尝试在所有适用的节点之间尽可能均匀地平衡服务 pod。

如果需要更多地控制 pod 的调度位置，请参阅“使用节点关联性”和“使用 Pod 关联性和反关联性”。

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。

2.2. 配置默认调度程序以控制 POD 放置

OpenShift Container Platform 的默认 pod 调度程序负责确定在集群的节点中放置新的 pod。它从 pod 读取数据，并尝试根据配置的策略寻找最适合的节点。它完全独立存在，作为单机或可插拔的解决方案。它不会修改 pod，只是为 pod 创建将 pod 与特定节点衔接起来的绑定。

调度程序的策略由一组 predicates 和 priorities 来定义。如需 predicates 和 priorities 的列表，请参阅[修改调度程序策略](#)。

默认调度程序对象示例

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: 2019-05-20T15:39:01Z
  generation: 1
  name: cluster
  resourceVersion: "1491"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: 6435dd99-7b15-11e9-bd48-0aec821b8e34
spec:
  policy: ❶
    name: scheduler-policy
  defaultNodeSelector: type=user-node,region=east ❷
```

❶ 您可以指定自定义调度程序策略文件的名称。

❷ （可选）指定默认节点选择器以限制将 pod 放置到特定的节点。

2.2.1. 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。这可以通过名为 *predicates* 的过滤函数列表筛选每个节点来完成。

排列过滤后节点列表的优先顺序

实现方式是让每个节点通过一系列优先级函数，为其分配从 0 到 10 的一个分数。0 代表不适合的节点，10 则代表最适合托管该 pod。调度程序配置还可以为每个优先级函数使用简单的权重（正数值）。每个优先级函数提供的节点分数乘以权重（大多数优先级的默认权重为 1），然后将每个节点从

所有优先级获得的分数相加。管理员可以使用这个权重属性，为一些优先级赋予更高的重要性。

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。

2.2.1.1. 了解调度程序策略

调度程序的策略由一组 predicates 和 priorities 来定义。

调度程序配置文件是一个 JSON 文件，必须命名为 **policy.cfg**，用于指定调度程序将要考量的 predicates 和 priorities。

如果没有调度程序策略文件，则使用默认的调度程序行为。



重要

调度程序配置文件中定义的 predicates 和 priorities 会完全覆盖默认的调度程序策略。如果需要任何默认的 predicates 和 priorities，必须在策略配置中明确指定对应的函数。

调度程序 ConfigMap 示例

```
apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"},
        {"name": "CheckVolumeBinding"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "PodFitsResources"},
        {"name": "MatchInterPodAffinity"},
        {"name": "CheckNodeUnschedulable"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "HostName"},
        {"name": "PodToleratesNodeTaints"}
      ],
      "priorities": [
        {"name": "LeastRequestedPriority", "weight": 1},
        {"name": "BalancedResourceAllocation", "weight": 1},
        {"name": "ServiceSpreadingPriority", "weight": 1},
        {"name": "NodePreferAvoidPodsPriority", "weight": 1},
        {"name": "NodeAffinityPriority", "weight": 1},
        {"name": "TaintTolerationPriority", "weight": 1},
        {"name": "ImageLocalityPriority", "weight": 1},
        {"name": "SelectorSpreadPriority", "weight": 1},
        {"name": "InterPodAffinityPriority", "weight": 1},
        {"name": "EqualPriority", "weight": 1}
      ]
    }

```

```

    ]
  }
  kind: ConfigMap
  metadata:
    creationTimestamp: "2019-09-17T08:42:33Z"
    name: scheduler-policy
    namespace: openshift-config
    resourceVersion: "59500"
    selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
    uid: 17ee8865-d927-11e9-b213-02d1e1709840`

```

2.2.2. 创建调度程序策略文件

您可以通过创建 JSON 文件并使用所需的 predicates 和 priorities 来控制或更改默认的调度行为。然后，您可以通过 JSON 文件生成 ConfigMap，并将 **cluster** 调度程序对象指定为使用该 ConfigMap。

流程

配置调度程序策略：

1. 使用所需的 predicates 和 priorities，创建一个名为 **policy.cfg** 的 JSON 文件。

调度程序 JSON 文件示例

```

{
  "kind" : "Policy",
  "apiVersion" : "v1",
  "predicates" : [ ❶
    {"name" : "PodFitsHostPorts"},
    {"name" : "PodFitsResources"},
    {"name" : "NoDiskConflict"},
    {"name" : "NoVolumeZoneConflict"},
    {"name" : "MatchNodeSelector"},
    {"name" : "MaxEBSVolumeCount"},
    {"name" : "MaxAzureDiskVolumeCount"},
    {"name" : "checkServiceAffinity"},
    {"name" : "PodToleratesNodeNoExecuteTaints"},
    {"name" : "MaxGCEPDVolumeCount"},
    {"name" : "MatchInterPodAffinity"},
    {"name" : "PodToleratesNodeTaints"},
    {"name" : "HostName"}
  ],
  "priorities" : [ ❷
    {"name" : "LeastRequestedPriority", "weight" : 1},
    {"name" : "BalancedResourceAllocation", "weight" : 1},
    {"name" : "ServiceSpreadingPriority", "weight" : 1},
    {"name" : "EqualPriority", "weight" : 1}
  ]
}

```

❶ 根据需要添加 predicates。

❷ 根据需要添加 priorities。

2. 根据调度程序 JSON 文件创建 ConfigMap :

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> 1
```

1 输入 ConfigMap 的名称。

例如 :

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

```
configmap/scheduler-policy created
```

```
apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"},
        {"name": "CheckVolumeBinding"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "PodFitsResources"},
        {"name": "MatchInterPodAffinity"},
        {"name": "CheckNodeUnschedulable"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "HostName"},
        {"name": "PodToleratesNodeTaints"}
      ],
      "priorities": [
        {"name": "LeastRequestedPriority", "weight": 1},
        {"name": "BalancedResourceAllocation", "weight": 1},
        {"name": "ServiceSpreadingPriority", "weight": 1},
        {"name": "NodePreferAvoidPodsPriority", "weight": 1},
        {"name": "NodeAffinityPriority", "weight": 1},
        {"name": "TaintTolerationPriority", "weight": 1},
        {"name": "ImageLocalityPriority", "weight": 1},
        {"name": "SelectorSpreadPriority", "weight": 1},
        {"name": "InterPodAffinityPriority", "weight": 1},
        {"name": "EqualPriority", "weight": 1}
      ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T08:42:33Z"
  name: scheduler-policy
  namespace: openshift-config
```

```
resourceVersion: "59500"
selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy
uid: 17ee8865-d927-11e9-b213-02d1e1709840`
```

3. 编辑调度程序 Operator 自定义资源以添加 ConfigMap :

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"<configmap-name>"}}}' --type=merge 1
```

- 1 指定 ConfigMap 的名称。

例如 :

```
$ oc patch Scheduler cluster --type='merge' -p '{"spec":{"policy":{"name":"scheduler-policy"}}}' --type=merge
```

在修改了调度程序配置资源后，等待 **openshift-kube-apiserver** pod 重新部署。这可能需要几分钟。只有重新部署 pod 后，新的调度程序才会生效。

4. 通过查看 **openshift-kube-scheduler** 命名空间中调度程序 pod 的日志，来验证已配置调度程序策略。以下命令检查由调度程序注册的 predicates 和 priorities :

```
$ oc logs <scheduler-pod> | grep predicates
```

例如 :

```
$ oc logs openshift-kube-scheduler-ip-10-0-141-29.ec2.internal | grep predicates
```

```
Creating scheduler with fit predicates 'map[MaxGCEPDVolumeCount:{}
MaxAzureDiskVolumeCount:{} CheckNodeUnschedulable:{} NoDiskConflict:{}
NoVolumeZoneConflict:{} MatchNodeSelector:{} GeneralPredicates:{}
MaxCSIVolumeCountPred:{} CheckVolumeBinding:{} MaxEBSVolumeCount:{}
PodFitsResources:{} MatchInterPodAffinity:{} HostName:{} PodToleratesNodeTaints:}]' and
priority functions 'map[InterPodAffinityPriority:{} LeastRequestedPriority:{}
ServiceSpreadingPriority:{} ImageLocalityPriority:{} SelectorSpreadPriority:{} EqualPriority:{}
BalancedResourceAllocation:{} NodePreferAvoidPodsPriority:{} NodeAffinityPriority:{}
TaintTolerationPriority:}]'
```

2.2.3. 修改调度程序策略

您可以通过在 **openshift-config** 项目中创建或编辑调度程序策略 ConfigMap 来更改调度行为。在 ConfigMap 中添加和移除 predicates 和 priorities，以创建 *调度程序策略*。

流程

要修改当前的自定义调度，请使用以下方法之一：

- 编辑调度程序策略 ConfigMap :

```
$ oc edit configmap <configmap-name> -n openshift-config
```

例如 :

```

$ oc edit configmap scheduler-policy -n openshift-config

apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [ 1
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "GeneralPredicates"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "MaxCSIVolumeCountPred"},
        {"name": "CheckVolumeBinding"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "PodFitsResources"},
        {"name": "MatchInterPodAffinity"},
        {"name": "CheckNodeUnschedulable"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "HostName"},
        {"name": "PodToleratesNodeTaints"}
      ],
      "priorities": [ 2
        {"name": "LeastRequestedPriority", "weight": 1},
        {"name": "BalancedResourceAllocation", "weight": 1},
        {"name": "ServiceSpreadingPriority", "weight": 1},
        {"name": "NodePreferAvoidPodsPriority", "weight": 1},
        {"name": "NodeAffinityPriority", "weight": 1},
        {"name": "TaintTolerationPriority", "weight": 1},
        {"name": "ImageLocalityPriority", "weight": 1},
        {"name": "SelectorSpreadPriority", "weight": 1},
        {"name": "InterPodAffinityPriority", "weight": 1},
        {"name": "EqualPriority", "weight": 1}
      ]
    }
kind: ConfigMap
metadata:
  creationTimestamp: "2019-09-17T17:44:19Z"
  name: scheduler-policy
  namespace: openshift-config
  resourceVersion: "15370"
  selfLink: /api/v1/namespaces/openshift-config/configmaps/scheduler-policy

```

- 1 根据需要添加或移除 predicates。
- 2 根据需要，添加、移除或更改 predicates 的权重。

调度程序需要几分钟时间来使用更新后的策略重启 pod。

- 更改所用的 predicates 和 priorities :
 1. 移除调度程序策略 CongifMap :

```
$ oc delete configmap -n openshift-config <name>
```

例如：

```
$ oc delete configmap -n openshift-config scheduler-policy
```

2. 根据需要，编辑 **policy.cfg** 文件来添加和移除 policies 和 predicates。

例如：

```
$ vi policy.cfg
```

```
apiVersion: v1
data:
  policy.cfg: |
    {
      "kind": "Policy",
      "apiVersion": "v1",
      "predicates": [
        {"name": "PodFitsHostPorts"},
        {"name": "PodFitsResources"},
        {"name": "NoDiskConflict"},
        {"name": "NoVolumeZoneConflict"},
        {"name": "MatchNodeSelector"},
        {"name": "MaxEBSVolumeCount"},
        {"name": "MaxAzureDiskVolumeCount"},
        {"name": "CheckVolumeBinding"},
        {"name": "CheckServiceAffinity"},
        {"name": "PodToleratesNodeNoExecuteTaints"},
        {"name": "MaxGCEPDVolumeCount"},
        {"name": "MatchInterPodAffinity"},
        {"name": "PodToleratesNodeTaints"},
        {"name": "HostName"}
      ],
      "priorities": [
        {"name": "LeastRequestedPriority", "weight": 2},
        {"name": "BalancedResourceAllocation", "weight": 2},
        {"name": "ServiceSpreadingPriority", "weight": 2},
        {"name": "EqualPriority", "weight": 2}
      ]
    }
  }
```

3. 根据调度程序 JSON 文件重新创建调度程序策略 ConfigMap：

```
$ oc create configmap -n openshift-config --from-file=policy.cfg <configmap-name> 1
```

1 输入 ConfigMap 的名称。

例如：

```
$ oc create configmap -n openshift-config --from-file=policy.cfg scheduler-policy
```

```
configmap/scheduler-policy created
```

2.2.3.1. 了解调度程序 predicates

predicates 是用于过滤掉不合格节点的规则。

OpenShift Container Platform 中默认提供一些 predicates。其中的一些 predicates 可以通过提供特定参数来自定义。可以组合多个 predicates 来提供更多节点过滤。

2.2.3.1.1. 静态 predicates

此类 predicates 不接受任何来自于用户的配置参数或输入。它们通过其确切的名称在调度程序配置中指定。

2.2.3.1.1.1. 默认 predicates

默认的调度程序策略包括以下 predicates :

NoVolumeZoneConflict 检查区域中是否有 pod 请求的卷。

```
{"name" : "NoVolumeZoneConflict"}
```

MaxEBSVolumeCount 检查可附加到 AWS 实例的最大卷数量。

```
{"name" : "MaxEBSVolumeCount"}
```

MaxAzureDiskVolumeCount 检查 Azure 磁盘卷的最大数量。

```
{"name" : "MaxAzureDiskVolumeCount"}
```

PodToleratesNodeTaints 检查 pod 是否可以容忍节点污点。

```
{"name" : "PodToleratesNodeTaints"}
```

CheckNodeUnschedulable 检查 pod 是否能够调度到指定有 **Unschedulable** 的节点。

```
{"name" : "CheckNodeUnschedulable"}
```

CheckVolumeBinding 根据 pod 请求的卷, 评估 pod 是否同时适合绑定和未绑定 PVC。* 对于绑定的 PVC, 该 predicate 检查给定节点是否满足对应 PV 的节点关联性。* 对于未绑定 PVC, 该 predicate 会搜索可满足 PVC 要求的可用 PV, 并且检查给定节点是否满足 PV 的节点关联性。

如果所有绑定 PVC 都有与节点兼容的 PV, 且所有未绑定 PVC 都可与可用并兼容节点的 PV 匹配, 该 predicate 会返回 true。

```
{"name" : "CheckVolumeBinding"}
```

NoDiskConflict 检查 pod 请求的卷是否可用。

```
{"name" : "NoDiskConflict"}
```

MaxGCEPDVolumeCount 检查 Google Compute Engine (GCE) 持久磁盘 (PD) 的最大数量。

```
{"name" : "MaxGCEPDVolumeCount"}
```

MaxCSIVolumeCountPred

MatchInterPodAffinity 检查 pod 关联性/反关联性规则是否允许该 pod。

```
{"name" : "MatchInterPodAffinity"}
```

2.2.3.1.1.2. 其他静态 predicates

OpenShift Container Platform 还支持下列 predicates :



注意

如果启用了 **Taint Nodes By Condition** 功能，则无法使用 **CheckNode-*** predicates。 **Taint Nodes By Condition** 功能默认启用。

CheckNodeCondition 检查 pod 是否可以调度到报告 **磁盘不足**、**网络不可用**或**未就绪**状况的节点。

```
{"name" : "CheckNodeCondition"}
```

CheckNodeLabelPresence 检查节点上是否存在所有指定的标签，不论它们的值是什么。

```
{"name" : "CheckNodeLabelPresence"}
```

checkServiceAffinity 检查 ServiceAffinity 标签是否对于节点上调度的 pod 而言一致。

```
{"name" : "checkServiceAffinity"}
```

PodToleratesNodeNoExecuteTaints 检查 pod 容限是否容忍节点的 **NoExecute** 污点。

```
{"name" : "PodToleratesNodeNoExecuteTaints"}
```

2.2.3.1.2. 常规 predicates

下列常规 predicates 检查是否通过非关键 predicates 和必要 predicates 的检查。非关键 predicates 是指只有非关键 pod 必须通过检查的 predicates，而必要 predicates 是指所有 pod 都必须通过检查的 predicates。

默认调度程序策略包含常规 predicates。

非关键常规 predicates

PodFitsResources 根据资源可用性（CPU、内存、GPU 等）来确定适合性。节点可以声明其资源容量，然后 pod 可以指定它们所需要的资源。适合性基于请求的资源，而非使用的资源。

```
{"name" : "PodFitsResources"}
```

必要常规 predicates

PodFitsHostPorts 确定节点是否有空闲端口可用于请求的 pod 端口（不存在端口冲突）。

```
{"name" : "PodFitsHostPorts"}
```

HostName 基于存在 Host 参数以及与主机名称匹配的字符串来确定适合性。

■

```
{"name" : "HostName"}
```

MatchNodeSelector 基于 pod 中定义的节点选择器 (nodeSelector) 查询来确定适合性。

```
{"name" : "MatchNodeSelector"}
```

2.2.3.2. 了解调度程序优先级

优先级是根据偏好对节点进行排序的规则。

可以指定一组自定义优先级来配置调度程序。OpenShift Container Platform 中默认提供一些优先级。也可通过提供某些参数来自定义其他优先级。可将多个优先级合并，并为每个优先级赋予不同的权重来影响优先顺序。

2.2.3.2.1. 静态优先级

静态优先级不使用用户提供的配置参数，但权重除外。权重必须指定，且不能为 0 或负数。

这些是在 **openshift-config** 项目中的调度程序策略 Configmap 中指定的。

2.2.3.2.1.1. 默认优先级

默认调度程序策略包括以下优先级。每个优先级函数的权重为 **1**，但 **NodePreferAvoidPodsPriority** 除外，它的权重是 **10000**。

NodeAffinityPriority 根据节点关联性调度偏好来排列节点的优先顺序

```
{"name" : "NodeAffinityPriority", "weight" : 1}
```

TaintTolerationPriority 为 pod 优先考虑那些具有较少 **不可容忍的污点** 的节点。不可容忍的污点是具有 **PreferNoSchedule** 键的污点。

```
{"name" : "TaintTolerationPriority", "weight" : 1}
```

ImageLocalityPriority 优先选择已请求了 pod 容器镜像的节点。

```
{"name" : "ImageLocalityPriority", "weight" : 1}
```

SelectorSpreadPriority 查找与 pod 匹配的服务、复制控制器 (RC)、复制集 (RS) 和有状态集，然后查找与这些选择器匹配的现有 pod。调度程序优先选择具有较少现有匹配 pod 的节点。然后，它会将 pod 调度到具有与所调度 pod 的选择器匹配的 pod 数量最少的节点上。

```
{"name" : "SelectorSpreadPriority", "weight" : 1}
```

InterPodAffinityPriority 通过迭代 **weightedPodAffinityTerm** 的元素并在节点满足对应的 PodAffinityTerm 时加上 **权重** 来计算总和。总和最高的节点是优先级最高的节点。

```
{"name" : "InterPodAffinityPriority", "weight" : 1}
```

LeastRequestedPriority 优先选择请求资源较少的节点。它计算节点上调度的 pod 所请求的内存和 CPU 百分比，并优先选择可用/剩余容量最高的节点。

```
{"name": "LeastRequestedPriority", "weight": 1}
```

BalancedResourceAllocation 优先选择资源使用率均衡的节点。它以占容量比形式计算 CPU 和内存已使用量的差值，并基于两个指标相互接近的程度来优先选择节点。这应该始终与 **LeastRequestedPriority** 一同使用。

```
{"name": "BalancedResourceAllocation", "weight": 1}
```

NodePreferAvoidPodsPriority 忽略除复制控制器以外的控制器拥有的 pod。

```
{"name": "NodePreferAvoidPodsPriority", "weight": 10000}
```

2.2.3.2.1.2. 其他静态优先级

OpenShift Container Platform 还支持下列优先级：

若不提供任何优先级配置，则 **EqualPriority** 为所有节点赋予相等的权重 **1**。建议您仅在测试环境中使用此优先级。

```
{"name": "EqualPriority", "weight": 1}
```

MostRequestedPriority 优先选择具有最多所请求资源的节点。它计算节点上调度的 pod 所请求的内存与 CPU 百分比，并根据请求量对容量的平均占比的最大值来排列优先级。

```
{"name": "MostRequestedPriority", "weight": 1}
```

ServiceSpreadingPriority 通过尽量减少将属于同一服务的 pod 分配到同一台机器的数量来分散 pod。

```
{"name": "ServiceSpreadingPriority", "weight": 1}
```

2.2.3.2.2. 可配置优先级

您可以在 **openshift-config** 项目的调度程序策略的 Configmap 中配置这些优先级，以添加可影响优先级的标记 (label)。

优先级函数的类型由它们所使用的参数来标识。由于它们是可配置的，因此可以组合类型相同（但配置参数不同）的多个优先级，但前提是它们的用户定义名称不同。

有关使用这些优先级的详情，请参考“修改调度程序策略”。

ServiceAntiAffinity 接受一个标签，确保将属于同一服务的 pod 正常地分散到基于标签值的一组节点。它为指定标签值相同的所有节点赋予相同的分数。它将较高的分数给予组内 pod 密度最低的节点。

```
{
  "kind": "Policy",
  "apiVersion": "v1",

  "priorities": [
    {
      "name": "<name>", 1
      "weight": 1 2
      "argument": {
```



```

    "serviceAntiAffinity":{
      "label": "<label>" ❸
    }
  }
}
]
}

```

- ❶ 指定优先级的名称。
- ❷ 指定权重。输入非零正数值。
- ❸ 指定要匹配的标签。

例如：

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "priorities": [
    {
      "name": "RackSpread",
      "weight": 1,
      "argument": {
        "serviceAntiAffinity": {
          "label": "rack"
        }
      }
    }
  ]
}

```



注意

在某些情况下，基于自定义标签的 **ServiceAntiAffinity** 不能按预期分散 pod。请参考 [此红帽解决方案](#)。

* **labelPreference** 参数根据指定的标签赋予优先级。如果节点上存在该标签，则该节点被赋予优先级。如果未指定标签，则为没有标签的节点赋予优先级。

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "priorities": [
    {
      "name": "<name>", ❶
      "weight": 1 ❷
      "argument": {
        "labelPreference": {
          "label": "<label>", ❸
          "presence": true ❹
        }
      }
    }
  ]
}

```

```

    }
  ]
}

```

- ❶ 指定优先级的名称。
- ❷ 指定权重。输入非零正数值。
- ❸ 指定要匹配的标签。
- ❹ 指定是否需要该标签，可以是 **true** 或 **false**。

2.2.4. 策略配置示例

如果使用调度程序策略文件进行了指定，则如下配置会指定默认的调度程序配置。

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {
      "name": "RegionZoneAffinity", ❶
      "argument": {
        "serviceAffinity": { ❷
          "labels": ["region, zone"] ❸
        }
      }
    }
  ],
  "priorities": [
    {
      "name": "RackSpread", ❹
      "weight": 1,
      "argument": {
        "serviceAntiAffinity": { ❺
          "label": "rack" ❻
        }
      }
    }
  ]
}

```

- ❶ predicate 的名称。
- ❷ predicate 的类型。
- ❸ predicate 的标签。
- ❹ 优先级的名称。
- ❺ 优先级的类型。
- ❻ 优先级的标签。

在下方的所有示例配置中，predicates 和 priorities 函数的列表都已截断，仅包含与指定用例相关的内容。在实践中，完整/有意义的调度程序策略应当包含前文所述的大部分（若非全部）默认 predicates 和 priorities。

以下示例定义了三个拓扑级别，即 region（关联性）-> zone（关联性）-> rack（反关联性）：

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {
      "name": "RegionZoneAffinity",
      "argument": {
        "serviceAffinity": {
          "labels": ["region, zone"]
        }
      }
    }
  ],
  "priorities": [
    {
      "name": "RackSpread",
      "weight": 1,
      "argument": {
        "serviceAntiAffinity": {
          "label": "rack"
        }
      }
    }
  ]
}
```

以下示例定义了三个拓扑级别，即 city（关联性）-> building（反关联性）-> room（反关联性）：

```
{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {
      "name": "CityAffinity",
      "argument": {
        "serviceAffinity": {
          "label": "city"
        }
      }
    }
  ],
  "priorities": [
    {
      "name": "BuildingSpread",
      "weight": 1,
      "argument": {
        "serviceAntiAffinity": {
          "label": "building"
        }
      }
    }
  ]
}
```

```

    }
  },
  {
    "name": "RoomSpread",
    "weight" : 1,
    "argument": {
      "serviceAntiAffinity": {
        "label": "room"
      }
    }
  }
]
}

```

以下示例定义了一个策略，以仅使用定义了“region”标签的节点，并且优先选择定义有“zone”标签的节点：

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {
      "name": "RequireRegion",
      "argument": {
        "labelPreference": {
          "labels": ["region"],
          "presence": true
        }
      }
    }
  ],
  "priorities": [
    {
      "name": "ZonePreferred",
      "weight" : 1,
      "argument": {
        "labelPreference": {
          "label": "zone",
          "presence": true
        }
      }
    }
  ]
}

```

以下示例组合使用静态和可配置的 predicates 和 priorities：

```

{
  "kind": "Policy",
  "apiVersion": "v1",
  "predicates": [
    {
      "name": "RegionAffinity",
      "argument": {
        "serviceAffinity": {

```

```

        "labels": ["region"]
      }
    },
    {
      "name": "RequireRegion",
      "argument": {
        "labelsPresence": {
          "labels": ["region"],
          "presence": true
        }
      }
    },
    {
      "name": "BuildingNodesAvoid",
      "argument": {
        "labelsPresence": {
          "label": "building",
          "presence": false
        }
      }
    },
    {"name": "PodFitsPorts"},
    {"name": "MatchNodeSelector"}
  ],
  "priorities": [
    {
      "name": "ZoneSpread",
      "weight": 2,
      "argument": {
        "serviceAntiAffinity": {
          "label": "zone"
        }
      }
    },
    {
      "name": "ZonePreferred",
      "weight": 1,
      "argument": {
        "labelPreference": {
          "label": "zone",
          "presence": true
        }
      }
    },
    {"name": "ServiceSpreadingPriority", "weight": 1}
  ]
}

```

2.3. 使用关联性和反关联性规则相对于其他 POD 放置 POD

关联性是 pod 的一个属性，用于控制它们希望调度到的节点。反关联性是 pod 的一个属性，用于阻止 pod 调度到某个节点上。

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

2.3.1. 了解 pod 关联性

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

- 如果新 pod 上的标签选择器与当前 pod 上的标签匹配，pod 关联性可以命令调度程序将新 pod 放置到与其他 pod 相同的节点上。
- 如果新 pod 上的标签选择器与当前 pod 上的标签匹配，pod 反关联性可以阻止调度程序将新 pod 放置到与具有相同标签的 pod 相同的节点上。

例如，您可以使用关联性规则，在服务内或相对于其他服务中的 pod 来分散或聚拢 pod。如果特定服务的 pod 的性能已知会受到另一服务的 pod 影响，那么您可以利用反关联性规则，防止前一服务的 pod 调度到与后一服务的 pod 相同的节点上。或者，您可以将服务的 pod 分散到不同的节点或可用区间，以减少关联的故障。

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 规格文件来配置 pod 的关联性/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。

设有 pod 关联性的 pod 配置文件示例

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2
        - labelSelector:
            matchExpressions:
              - key: security 3
                operator: In 4
                values:
```

```

- S1 5
  topologyKey: failure-domain.beta.kubernetes.io/zone
containers:
- name: with-pod-affinity
  image: docker.io/ocpqe/hello-pod

```

- 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**。

设有 pod 反关联性的 pod 配置文件示例

```

apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  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

```

- 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。

2.3.2. 配置 pod 关联性规则

以下步骤演示了一个简单的双 pod 配置，它创建一个带有某标签的 pod，以及一个使用关联性来允许随着该 pod 一起调度的 pod。

流程

1. 创建 pod 规格中具有特定标签的 pod：

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-s1
  labels:
    security: S1
spec:
  containers:
  - name: security-s1
    image: docker.io/ocpqe/hello-pod
```

2. 在创建其他 pod 时，请按下方所示编辑 pod 规格：

- a. 使用 **podAntiAffinity** 小节配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数或 **preferredDuringSchedulingIgnoredDuringExecution** 参数：
- b. 指定必须满足的键和值。如果您希望新 pod 与另一个 pod 一起调度，请使用与第一个 pod 上标签相同的 **key** 和 **value** 参数。

```
podAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchExpressions:
      - key: security
        operator: In
        values:
        - S1
    topologyKey: failure-domain.beta.kubernetes.io/zone
```

- c. 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。
 - d. 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。
3. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

2.3.3. 配置 pod 反关联性规则

以下步骤演示了一个简单的双 pod 配置，它创建一个带有某标签的 pod，以及一个使用反关联性偏好规则来尝试阻止随着该 pod 一起调度的 pod。

流程

1. 创建 pod 规格中具有特定标签的 pod :

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-s2
  labels:
    security: S2
spec:
  containers:
  - name: security-s2
    image: docker.io/ocpqe/hello-pod
```

2. 在创建其他 pod 时, 请编辑 pod 规格来设置以下参数 :
3. 使用 **podAntiAffinity** 小节配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数或 **preferredDuringSchedulingIgnoredDuringExecution** 参数 :
 - a. 为节点指定一个 1 到 100 的权重。优先选择权重最高的节点。
 - b. 指定必须满足的键和值。如果您希望新 pod 不与另一个 pod 一起调度, 请使用与第一个 pod 上标签相同的 **key** 和 **value** 参数。

```
podAntiAffinity:
  preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 100
  podAffinityTerm:
    labelSelector:
      matchExpressions:
      - key: security
        operator: In
        values:
        - S2
    topologyKey: kubernetes.io/hostname
```

- c. 为偏好规则指定一个 1 到 100 的权重。
 - d. 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。
4. 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。
 5. 创建 pod。

```
$ oc create -f <pod-spec>.yaml
```

2.3.4. pod 关联性和反关联性规则示例

以下示例演示了 pod 关联性和 pod 反关联性。

2.3.4.1. Pod 关联性

以下示例演示了具有匹配标签和标签选择器的 pod 的 pod 关联性。

- pod **team4** 具有标签 **team:4**。

```
$ cat team4.yaml
apiVersion: v1
kind: Pod
metadata:
  name: team4
  labels:
    team: "4"
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- pod **team4a** 在 **podAffinity** 下具有标签选择器 **team:4**。

```
$ cat pod-team4a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: team4a
spec:
  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
```

- **team4a** pod 调度到与 **team4** pod 相同的节点上。

2.3.4.2. Pod 反关联性

以下示例演示了具有匹配标签和标签选择器的 pod 的 pod 反关联性。

- pod **pod-s1** 具有标签 **security:s1**。

```
cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
  - name: ocp
    image: docker.io/ocpqe/hello-pod
```

- pod **pod-s2** 在 **podAntiAffinity** 下具有标签选择器 **security:s1**。

```
cat pod-s2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  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
```

- pod **pod-s2** 无法调度到与 **pod-s1** 相同的节点上。

2.3.4.3. 无匹配标签的 Pod 反关联性

以下示例演示了在没有匹配标签和标签选择器时的 pod 的 pod 关联性。

- pod **pod-s1** 具有标签 **security:s1**。

```
$ cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
  labels:
    security: s1
spec:
  containers:
    - name: ocp
      image: docker.io/ocpqe/hello-pod
```

- pod **pod-s2** 具有标签选择器 **security:s2**。

```
$ cat pod-s2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s2
spec:
  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

```

- 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：

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
pod-s2	0/1	Pending	0	32s	<none>	

2.4. 使用节点关联性规则控制节点上的 POD 放置

关联性是 pod 的一个属性，用于控制它们希望调度到的节点。

在 OpenShift Container Platform 中，节点关联性是由调度程序用来确定 pod 的可放置位置的一组规则。规则是使用节点中的自定义标签和 pod 中指定的选择器进行定义的。

2.4.1. 了解节点关联性

节点关联性允许 pod 指定与可以放置该 pod 的一组节点的关联性。节点对放置没有控制权。

例如，您可以将 pod 配置为仅在具有特定 CPU 或位于特定可用区的节点上运行。

节点关联性规则有两种，即**必要规则**和**偏好规则**。

必须满足必要规则，pod 才能调度到节点上。偏好规则指定在满足规则时调度程序会尝试强制执行规则，但不保证一定能强制执行成功。



注意

如果节点标签在运行时改变，使得不再满足 pod 上的节点关联性规则，该 pod 将继续在这个节点上运行。

您可以通过 pod 规格文件配置节点关联性。您可以指定必要规则或偏好规则，或同时指定这两种规则。如果您同时指定，节点必须首先满足必要规则，然后尝试满足偏好规则。

下例中的 pod 规格包含一条规则，要求 pod 放置到具有某一标签（键为 **e2e-az-NorthSouth** 且值为 **e2e-az-North** 或 **e2e-az-South**）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:
  affinity:
    nodeAffinity: 1
      requiredDuringSchedulingIgnoredDuringExecution: 2

```

```

nodeSelectorTerms:
- matchExpressions:
  - key: e2e-az-NorthSouth ❸
    operator: In ❹
    values:
  - e2e-az-North ❺
  - e2e-az-South ❻
containers:
- name: with-node-affinity
  image: docker.io/ocpqe/hello-pod

```

- ❶ 用于配置节点关联性的小节。
- ❷ 定义必要规则。
- ❸❺❻ 必须匹配键/值对 (标签) 才会应用该规则。
- ❹ 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:
  affinity:
    nodeAffinity: ❶
      preferredDuringSchedulingIgnoredDuringExecution: ❷
        - weight: 1 ❸
          preference:
            matchExpressions:
              - key: e2e-az-EastWest ❹
                operator: In ❺
                values:
              - e2e-az-East ❻
              - e2e-az-West ❼
  containers:
  - name: with-node-affinity
    image: docker.io/ocpqe/hello-pod

```

- ❶ 用于配置节点关联性的小节。
- ❷ 定义偏好规则。
- ❸ 为偏好规则指定权重。优先选择权重最高的节点。
- ❹❺❻ 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。

2.4.2. 配置节点关联性必要规则

必须满足必要规则，pod 才能调度到节点上。

流程

以下步骤演示了一个简单的配置，此配置会创建一个节点，以及调度程序要放置到该节点上的 pod。

1. 使用 **oc label node** 命令给节点添加标签：

```
$ oc label node node1 e2e-az-name=e2e-az1
```

2. 在 pod 规格中，使用 **nodeAffinity** 小节来配置 **requiredDuringSchedulingIgnoredDuringExecution** 参数：

- a. 指定必须满足的键和值。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 **value** 参数。
- b. 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**、**Lt** 或 **Gt**。例如，使用运算符 **In** 来要求节点上存在该标签：

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
```

3. 创建 pod：

```
$ oc create -f e2e-az2.yaml
```

2.4.3. 配置节点关联性偏好规则

偏好规则指定在满足规则时调度程序会尝试强制执行规则，但不保证一定能强制执行成功。

流程

以下步骤演示了一个简单的配置，此配置会创建一个节点，以及调度程序尝试放置到该节点上的 pod。

1. 使用 **oc label node** 命令给节点添加标签：

```
$ oc label node node1 e2e-az-name=e2e-az3
```

2. 在 pod 规格中，使用 **nodeAffinity** 小节来配置 **preferredDuringSchedulingIgnoredDuringExecution** 参数：

- a. 为节点指定一个权重，值为 1 到 100 的数字。优先选择权重最高的节点。
- b. 指定必须满足的键和值。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 **value** 参数：

```
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: e2e-az-name
            operator: In
            values:
            - e2e-az3
```

3. 指定一个 **operator**。运算符可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**、**Lt** 或 **Gt**。例如，使用运算符 **In** 来要求节点上存在该标签。
4. 创建 pod。

```
$ oc create -f e2e-az3.yaml
```

2.4.4. 节点关联性规则示例

以下示例演示了节点关联性。

2.4.4.1. 具有匹配标签的节点关联性

以下示例演示了具有匹配标签的节点与 pod 的节点关联性：

- Node1 节点具有标签 **zone:us**：

```
$ oc label node node1 zone=us
```

- pod **pod-s1** 在节点关联性必要规则下具有 **zone** 和 **us** 键/值对：

```
$ cat pod-s1.yaml
apiVersion: v1
```

```

kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "zone"
            operator: In
            values:
            - us

```

- pod **pod-s1** 可以调度到 **Node1** 上 :

```

$ oc get pod -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP      NODE
pod-s1 1/1     Running   0           4m   IP1    node1

```

2.4.4.2. 无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** 节点有标签 **zone:emea** :

```

$ oc label node node1 zone=emea

```

- pod **pod-s1** 在节点关联性必要规则下具有 **zone** 和 **us** 键/值对 :

```

$ cat pod-s1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-s1
spec:
  containers:
  - image: "docker.io/ocpqe/hello-pod"
    name: hello-pod
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: "zone"
            operator: In
            values:
            - us

```

- pod **pod-s1** 无法调度到 **Node1** 上 :


```
$ oc describe pod pod-s1
<---snip-->
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).
```

2.4.5. 其他资源

如需有关更改节点标签的信息，请参阅[了解如何更新节点上的标签](#)。

2.5. 将 POD 放置到过量使用的节点

处于*过量使用* (*overcommitted*) 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，并提供最低服务保障。限制约束节点上可以消耗的计算资源数量。

2.5.1. 了解过量使用

请求和限制可让管理员允许和管理节点上资源的过量使用。调度程序使用请求来调度容器，并提供最低服务保障。限制约束节点上可以消耗的计算资源数量。

OpenShift Container Platform 管理员可以通过配置主控机 (master) 来覆盖开发人员容器上设置的请求和限制之间的比率，来控制过量使用的程度并管理节点上的容器密度。与项目一级上的用于指定限制和默认值的 LimitRange 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。

2.5.2. 了解节点过量使用

在过量使用的环境中，务必要正确配置节点，以提供最佳的系统行为。

当节点启动时，它会确保为内存管理正确设置内核可微调标识。除非物理内存不足，否则内核应该永不会在内存分配时失败。

在过量使用的环境中，务必要正确配置节点，以提供最佳的系统行为。

当节点启动时，它会确保为内存管理正确设置内核可微调标识。除非物理内存不足，否则内核应该永不会在内存分配时失败。

为确保这一行为，OpenShift Container Platform 通过将 `vm.overcommit_memory` 参数设置为 `1` 来覆盖默认操作系统设置，从而将内核配置为始终过量使用内存。

OpenShift Container Platform 还通过将 `vm.panic_on_oom` 参数设置为 `0`，将内核配置为不会在内存不足时崩溃。设置为 `0` 可告知内核在内存不足 (OOM) 情况下调用 `oom_killer`，以根据优先级终止进程

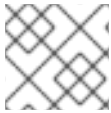
您可以通过对节点运行以下命令来查看当前的设置：

```
$ sysctl -a |grep commit
```

```
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
```

```
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

2.6. 使用节点污点控制 POD 放置

通过污点和容限，节点可以控制哪些 Pod 应该（或不应该）调度到节点上。

2.6.1. 了解污点和容限

通过使用污点 (*taint*)，节点可以拒绝调度 pod，除非 pod 具有匹配的容限 (*toleration*)。

您可以通过节点规格 (**NodeSpec**) 将污点应用到节点，并通过 pod 规格 (**PodSpec**) 将容限应用到 pod。节点上的污点指示节点排斥所有不容许该污点的 Pod。

污点与容限由 key、value 和 effect 组成。运算符允许您将其中一个参数留空。

表 2.1. 污点和容限组件

参数	描述
key	key 是任意字符串，最多 253 个字符。key 必须以字母或数字开头，可以包含字母、数字、连字符、句点和下划线。
value	value 是任意字符串，最多 63 个字符。value 必须以字母或数字开头，可以包含字母、数字、连字符、句点和下划线。

参数	描述						
effect	<p>effect 的值包括：</p> <table border="1"> <tr> <td>NoSchedule</td> <td> <ul style="list-style-type: none"> 与污点不匹配的新 pod 不会调度到该节点上。 该节点上现有的 pod 会保留。 </td> </tr> <tr> <td>PreferNoSchedule</td> <td> <ul style="list-style-type: none"> 与污点不匹配的新 pod 可以调度到该节点上，但调度程序会尽量不这样调度。 该节点上现有的 pod 会保留。 </td> </tr> <tr> <td>NoExecute</td> <td> <ul style="list-style-type: none"> 与污点不匹配的新 pod 无法调度到该节点上。 节点上没有匹配容限的现有 pod 将被移除。 </td> </tr> </table>	NoSchedule	<ul style="list-style-type: none"> 与污点不匹配的新 pod 不会调度到该节点上。 该节点上现有的 pod 会保留。 	PreferNoSchedule	<ul style="list-style-type: none"> 与污点不匹配的新 pod 可以调度到该节点上，但调度程序会尽量不这样调度。 该节点上现有的 pod 会保留。 	NoExecute	<ul style="list-style-type: none"> 与污点不匹配的新 pod 无法调度到该节点上。 节点上没有匹配容限的现有 pod 将被移除。
NoSchedule	<ul style="list-style-type: none"> 与污点不匹配的新 pod 不会调度到该节点上。 该节点上现有的 pod 会保留。 						
PreferNoSchedule	<ul style="list-style-type: none"> 与污点不匹配的新 pod 可以调度到该节点上，但调度程序会尽量不这样调度。 该节点上现有的 pod 会保留。 						
NoExecute	<ul style="list-style-type: none"> 与污点不匹配的新 pod 无法调度到该节点上。 节点上没有匹配容限的现有 pod 将被移除。 						
operator	<table border="1"> <tr> <td>Equal</td> <td>key/value/effect 参数必须匹配。这是默认值。</td> </tr> <tr> <td>Exists</td> <td>key/effect 参数必须匹配。您必须保留一个空的 value 参数，这将匹配任何值。</td> </tr> </table>	Equal	key/value/effect 参数必须匹配。这是默认值。	Exists	key/effect 参数必须匹配。您必须保留一个空的 value 参数，这将匹配任何值。		
Equal	key/value/effect 参数必须匹配。这是默认值。						
Exists	key/effect 参数必须匹配。您必须保留一个空的 value 参数，这将匹配任何值。						

容限与污点匹配：

- 如果 **operator** 参数设为 **Equal**：
 - **key** 参数相同；
 - **value** 参数相同；
 - **effect** 参数相同。
- 如果 **operator** 参数设为 **Exists**：
 - **key** 参数相同；
 - **effect** 参数相同。

Kubernetes 中内置了以下污点：

- **node.kubernetes.io/not-ready** : 节点未就绪。这与节点状况 **Ready=False** 对应。
- **node.kubernetes.io/unreachable** : 节点无法从节点控制器访问。这与节点状况 **Ready=Unknown** 对应。
- **node.kubernetes.io/out-of-disk** : 节点上的可用空间不足，无法添加新 pod。这与节点状况 **OutOfDisk=True** 对应。
- **node.kubernetes.io/memory-pressure** : 节点存在内存压力问题。这与节点状况 **MemoryPressure=True** 对应。
- **node.kubernetes.io/disk-pressure** : 节点存在磁盘压力问题。这与节点状况 **DiskPressure=True** 对应。
- **node.kubernetes.io/network-unavailable** : 节点网络不可用。
- **node.kubernetes.io/unschedulable** : 节点不可调度。
- **node.cloudprovider.kubernetes.io/uninitialized** : 当节点控制器通过外部云提供商启动时，在节点上设置这个污点来将其标记为不可用。在云控制器管理器中的某个控制器初始化这个节点后，kubelet 会移除此污点。

2.6.1.1. 了解如何使用容限秒数来延迟 pod 驱除

您可以通过在 pod 规格中指定 **tolerationSeconds** 参数，指定 pod 在被驱除前可以保持与节点绑定的时长。如果将具有 **NoExecute** effect 的污点添加到某个节点，则所有不容许该污点的 pod 都被立即驱除（容许该污点的 pod 不会被驱除）。但是，如果要被驱除的 pod 具有 **tolerationSeconds** 参数，则只有该时限到期后 pod 才会被驱除。

例如：

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

在这里，如果此 pod 正在运行但没有匹配的污点，pod 保持与节点绑定 3600 秒，然后被驱除。如果污点在这个时间之前移除，pod 就不会被驱除。

2.6.1.2. 了解如何使用多个污点

您可以在同一个节点中放入多个污点，并在同一 pod 中放入多个容限。OpenShift Container Platform 按照如下所述处理多个污点和容限：

1. 处理 pod 具有匹配容限的污点。
2. 其余的不匹配污点在 pod 上有指示的 effect：
 - 如果至少有一个不匹配污点具有 **NoSchedule** effect，则 OpenShift Container Platform 无法将 pod 调度到该节点上。
 - 如果没有不匹配污点具有 **NoSchedule** effect，但至少有一个不匹配污点具有 **PreferNoSchedule** effect，则 OpenShift Container Platform 尝试不将 pod 调度到该节点上。

- 如果至少有一个未匹配污点具有 **NoExecute** effect, OpenShift Container Platform 会将 pod 从该节点驱除 (如果它已在该节点上运行), 或者不将 pod 调度到该节点上 (如果还没有在该节点上运行)。
 - 不容许污点的 Pod 会立即被驱除。
 - 如果 Pod 容许污点, 且没有在容限规格中指定 **tolerationSeconds**, 则会永久保持绑定。
 - 如果 Pod 容许污点, 且指定了 **tolerationSeconds**, 则会在指定的时间里保持绑定。

例如 :

- 节点具有以下污点 :

```
$ oc adm taint nodes node1 key1=value1:NoSchedule
$ oc adm taint nodes node1 key1=value1:NoExecute
$ oc adm taint nodes node1 key2=value2:NoSchedule
```

- pod 具有以下容限 :

```
tolerations:
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoSchedule"
- key: "key1"
  operator: "Equal"
  value: "value1"
  effect: "NoExecute"
```

在本例中, pod 无法调度到节点上, 因为没有与第三个污点匹配的容限。如果在添加污点时 pod 已在节点上运行, pod 会继续运行, 因为第三个污点是三个污点中 pod 唯一不容许的污点。

2.6.1.3. 防止因为节点问题而发生 pod 驱除

Taint-Based Evictions 功能默认启用, 为未就绪或无法访问的节点添加一个带有 **NoExecute** 效果的污点。这样, 就可以对每个 pod 设置在节点变得不可访问或未就绪时保持与节点绑定的时长, 而不是使用默认的五分钟。例如, 当网络问题解决时如果工作负载安全可以继续运行, 您可能希望允许不可访问节点上的 Pod。

如果节点进入未就绪状态, 节点控制器会将 **node.kubernetes.io/not-ready:NoExecute** 污点添加到节点。如果节点进入不可访问状态, 节点控制器会将 **node.kubernetes.io/unreachable:NoExecute** 污点添加到节点。

NoExecute 污点会影响节点上已在运行的 Pod, 如下所示 :

- 不容许污点的 Pod 会立即被驱除。
- 如果 Pod 容许污点, 且没有在容限规格中指定 **tolerationSeconds**, 则会永久保持绑定。
- 如果 Pod 容许污点, 且指定了 **tolerationSeconds**, 则会在指定的时间里保持绑定。

2.6.1.4. 了解 pod 调度和节点状况 (根据条件设置污点节点)

OpenShift Container Platform 会自动将报告内存压力和磁盘压力等状况的节点变成污点。如果某个节点报告一个状况，则添加一个污点，直到状况被清除为止。这些污点具有 **NoSchedule** effect；即，Pod 无法调度到该节点上，除非 Pod 有匹配的容限。此功能称为 **根据条件设置污点节点 (Taint Nodes By Condition)**，默认为启用。

在调度 Pod 前，调度程序会检查节点上是否有这些污点。如果污点存在，则将 Pod 调度到另一个节点。由于调度程序检查的是污点而非实际的节点状况，因此您可以通过添加适当的 Pod 容限，将调度程序配置为忽略其中一些节点状况。

DaemonSet 控制器会自动将下列容限添加到所有守护进程，以保证向后兼容性：

- node.kubernetes.io/memory-pressure
- node.kubernetes.io/disk-pressure
- node.kubernetes.io/out-of-disk（仅限关键 Pod）
- node.kubernetes.io/unschedulable（1.10 或更高版本）
- node.kubernetes.io/network-unavailable（仅限主机网络）

您还可以在 DaemonSet 中添加任意容限。

2.6.1.5. 了解根据状况驱除 pod (基于污点的驱除)

基于污点的驱除 (Taint-Based Evictions) 功能默认为启用，可以从遇到特定状况（如未就绪和无法访问）的节点驱除 pod。当节点遇到其中一个状况时，OpenShift Container Platform 会自动给节点添加污点，并开始驱除 pod 以及将 pod 重新调度到其他节点。

Taint Based Evictions 具有 **NoExecute** effect，不容许污点的 pod 都会被立即驱除，而容许污点的 pod 则永不会被驱除。



注意

OpenShift Container Platform 会以限速方式驱除 pod，从而防止在主机从节点分离等情形中发生大量 pod 驱除。

此功能与 **tolerationSeconds** 相结合，允许您指定 pod 应当在多久时间内保持与具有某一节点状况的节点的绑定。如果在 **tolerationSeconds** 到期后状况仍然存在，则污点会保持在节点上，并且 pod 会以限速方式被驱除。如果状况在 **tolerationSeconds** 到期前清除，则 pod 不会被移除。

OpenShift Container Platform 会自动为 **node.kubernetes.io/not-ready** 和 **node.kubernetes.io/unreachable** 添加容限并设置 **tolerationSeconds=300**，除非 pod 配置中已指定了其中任一种容限。

```
spec
  tolerations:
  - key: node.kubernetes.io/not-ready
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
  - key: node.kubernetes.io/unreachable
    operator: Exists
    effect: NoExecute
    tolerationSeconds: 300
```

这些容限确保了在默认情况下，pod 在检测到这些节点条件问题中的任何一个时，会保持绑定 5 分钟。

您可以根据需要配置这些容限。例如，如果您有一个具有许多本地状态的应用程序，您可能希望在发生网络分区时让 pod 与节点保持绑定更久一些，以等待分区恢复并避免 pod 驱除行为的发生。

DaemonSet pod 创建时具有以下污点的 NoExecute 容限，但没有设置 tolerationSeconds：

- **node.kubernetes.io/unreachable**
- **node.kubernetes.io/not-ready**

这可以确保 DaemonSet pod 不会因为节点状况而被驱除，即使 **DefaultTolerationSeconds** 准入控制器已被禁用。

2.6.2. 添加污点和容限

您可以给节点添加污点并给 pod 添加容限，让节点能够控制哪些 pod 应该（或不应该）调度到节点上。

流程

1. 在以下命令中使用污点和容限组件表中描述的参数：

```
$ oc adm taint nodes <node-name> <key>=<value>:<effect>
```

例如：

```
$ oc adm taint nodes node1 key1=value1:NoExecute
```

本例在 **node1** 上放置一个键为 **key1** 且值为 **value1** 的污点，污点效果是 **NoExecute**。

2. 通过编辑 pod 规格以使其包含一个 **tolerations** 部分，给 pod 添加容限：

使用 Equal 运算符的 pod 配置文件示例

```
tolerations:
- key: "key1" ①
  operator: "Equal" ②
  value: "value1" ③
  effect: "NoExecute" ④
  tolerationSeconds: 3600 ⑤
```

① ② ③ ④ 容限参数，如污点和容限组件表中所述。

⑤ **tolerationSeconds** 参数指定 pod 在被驱除前可以保持与节点绑定的时长。

例如：

使用 Exists 运算符的 pod 配置文件示例

```
tolerations:
- key: "key1"
  operator: "Exists"
```

```
effect: "NoExecute"
tolerationSeconds: 3600
```

这两个容限都与上述 `oc adm taint` 命令创建的污点匹配。具有任一容限的 pod 可以调度到 `node1`。

2.6.2.1. 使用污点和容限为用户指定专用节点

您可以指定一组节点供一组特定用户独占使用。

流程

指定专用节点：

1. 给这些节点添加污点：
例如：

```
$ oc adm taint nodes node1 dedicated=groupName:NoSchedule
```

2. 通过编写自定义准入控制器，给 pod 添加对应的容限。
只有具有这些容限的 pod 才可以使用专用节点。

2.6.2.2. 使用污点和容限将用户绑定到节点

您可以配置节点，使特定用户只能使用专用节点。

流程

配置节点以使用户只能使用该节点：

1. 给这些节点添加污点：
例如：

```
$ oc adm taint nodes node1 dedicated=groupName:NoSchedule
```

2. 通过编写自定义准入控制器，给 pod 添加对应的容限。
准入控制器应该添加一个节点关联性，使得 pod 只能调度到具有 `key:value` 标签 (`dedicated=groupName`) 的节点。

3. 给专用节点添加与污点类似的标签（如 `key:value` 标签）。

2.6.2.3. 使用污点和容限控制具有特殊硬件的节点

如果集群中有少量节点具有特殊的硬件（如 GPU），您可以使用污点和容限让不需要特殊硬件的 pod 与这些节点保持距离，从而将这些节点保留给那些确实需要特殊硬件的 pod。您还可以要求需要特殊硬件的 pod 使用特定的节点。

流程

确保 pod 被禁止使用特殊硬件：

1. 使用以下命令之一，给拥有特殊硬件的节点添加污点：

```
$ oc adm taint nodes <node-name> disktype=ssd:NoSchedule
$ oc adm taint nodes <node-name> disktype=ssd:PreferNoSchedule
```


2. 给利用准入控制器来使用特殊硬件的 pod 添加对应的容限。

例如，准入控制器可以使用 pod 的一些特征来决定是否应该允许 pod 通过添加容限来使用特殊节点。

要确保 pod 只能使用特殊硬件，您需要一些额外的机制。例如，您可以给具有特殊硬件的节点打上标签，并在需要该硬件的 pod 上使用节点关联性。

2.6.3. 删除污点和容限

您可以根据需要，从节点移除污点并从 pod 移除容限。

流程

移除污点和容限：

1. 从节点移除污点：

```
$ oc adm taint nodes <node-name> <key>-
```

例如：

```
$ oc adm taint nodes ip-10-0-132-248.ec2.internal key1-
node/ip-10-0-132-248.ec2.internal untainted
```

2. 要从 pod 移除某一容限，请编辑 pod 规格来移除该容限：

```
tolerations:
- key: "key2"
  operator: "Exists"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

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。您可以使用特定节点选择器将特定的 pod 放置到特定的节点上，项目节点选择器将新 pod 放置到该项目的特定节点上，或默认的集群范围节点选择器，以将新 pod 放置到集群中的任何特定节点上。

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。

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。

2.7.1. 使用节点选择器控制 pod 放置

您可以使用 pod 上的节点选择器标签来控制 pod 的调度位置。

使用节点选择器时，OpenShift Container Platform 会将 pod 调度到包含匹配标签的节点。

要向现有的 pod 添加节点选择器，请将节点选择器添加到该节点的控制对象，如 ReplicaSet、Daemonset 或 StatefulSet。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。

您可以向节点或 MachineConfig 添加标签，但标签不会在节点或机器停机后保留。将标签添加到 MachineSet 可确保新的节点或机器具有该标签。



注意

您不能直接将节点选择器添加到现有调度的 pod 中。

前提条件

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** ReplicaSet 控制：

```
$ oc describe pod router-default-66d5cf9464-7pwkc
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：

```
ownerReferences:
- apiVersion: apps/v1
  kind: ReplicaSet
  name: router-default-66d5cf9464
  uid: d81dd094-da26-11e9-a48a-128e7edf0312
  controller: true
  blockOwnerDeletion: true
```

流程

1. 通过使用 MachineSet 或直接编辑节点，为节点添加标签：
 - 在创建节点时，使用 MachineSet 将标签添加到由 MachineSet 管理的节点上：
 - a. 运行以下命令，将节点选择器添加到 MachineSet 中：

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels","value":{"<key>="
<value>","<key>="<value>"}]}' -n openshift-machine-api
```

例如：

```
$ oc patch MachineSet abc612-msrtw-worker-us-east-1c --type='json' -
p=[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}] -n openshift-machine-api
```

- b. 使用 **oc edit** 命令验证标签是否已添加到 MachineSet:
例如 :

```
$ oc edit MachineSet abc612-msrtw-worker-us-east-1c -n openshift-machine-api
```

MachineSet 对象示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
...
spec:
...
  template:
    metadata:
    ...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
    ...
```

- 直接向节点添加标签 :
 - a. 为节点编辑 **Node** 对象 :

```
$ oc label nodes <name> <key>=<value>
```

例如, 若要为以下节点添加标签 :

```
$ oc label nodes ip-10-0-142-25.ec2.internal type=user-node region=east
```

- b. 验证标签是否已添加到节点 :

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS ROLES  AGE  VERSION
ip-10-0-142-25.ec2.internal Ready  worker  17m  v1.18.3+002a51f
```

2. 将匹配的节点选择器添加到 pod :

- 要将节点选择器添加到现有和未来的 pod，请向 pod 的控制对象添加节点选择器：

ReplicaSet 对象示例

```
kind: ReplicaSet
...
spec:
...
template:
  metadata:
    creationTimestamp: null
    labels:
      ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
      pod-template-hash: 66d5cf9464
  spec:
    nodeSelector:
      beta.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特定的 pod，直接将选择器添加到 **Pod** 对象中：

Pod 对象示例

```
apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    <key>: <value>
...

```

例如：

使用节点选择器的 Pod 对象示例

```
apiVersion: v1
kind: Pod
...
spec:
  nodeSelector:
    region: east
    type: user-node
```

2.7.2. 创建默认的集群范围节点选择器

您可以组合使用 pod 上的默认集群范围节点选择器和节点上的标签，将集群中创建的所有 pod 限制到特定节点。

使用集群范围节点选择器时，如果您在集群中创建 pod, OpenShift Container Platform 会将默认节点选择器添加到 pod，并将该 pod 调度到具有匹配标签的节点。

您可以通过创建调度程序 Operator 自定义资源 (CR) 来配置集群范围节点选择器。您可以通过编辑节点对象、MachineSet 或 MachineConfig 向节点添加标签。将标签添加到 MachineSet 可确保节点或机器停机时，新节点具有标签。如果节点或机器停机，添加到节点或 MachineConfig 的标签不会保留。

例如，调度程序配置集群范围的 **region=east** 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 1
  ...
```

集群中的节点具有 **type=user-node,region=east** 标签：

节点对象示例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
  ...
labels:
  region: east
  type: user-node
  ...
```

如果您在该集群中创建 pod，则该 pod 使用集群范围节点选择器创建，并调度到标记的节点：

Pod 对象示例

```
apiVersion: v1
kind: Pod
  ...
spec:
  nodeSelector:
    region: east
  ...
```

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 spec 中的节点选择器不使用项目节点选择器，则 Pod 不会被创建或调度。

如果 **Pod** 对象包含不是集群范围节点选择器或项目节点选择器，则 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 输出示例

```

apiVersion: v1
kind: Pod
....

spec:
  nodeSelector:
    region: west

```

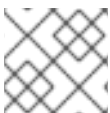
从该 spec 创建 pod 时，您收到类似以下消息的错误：

错误信息示例

```

Error from server (Forbidden): error when creating "pod.yaml": pods "pod-4" is forbidden: pod node label selector conflicts with its project node label selector

```



注意

您可以向 pod 添加额外的键/值对。但是，您无法为默认项目键添加其他值。

流程

添加默认的集群范围节点选择器：

1. 编辑调度程序 Operator 自定义资源以添加集群节点选择器：

```
$ oc edit scheduler cluster
```

```

apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  name: cluster
...

spec:
  defaultNodeSelector: type=user-node,region=east 1

```

```
mastersSchedulable: false
policy:
  name: ""
```

1 1 使用适当的 **<key>:<value>** 对添加节点选择器。

完成此更改后，请等待重新部署 **openshift-kube-apiserver** 项目中的 pod。这可能需要几分钟。只有 pod 重新部署后，默认集群节点选择器才会生效。

2. 通过使用 MachineSet 或直接编辑节点，为节点添加标签：

- 在创建节点时，使用 MachineSet 将标签添加到由 MachineSet 管理的节点上：

a. 运行以下命令，将节点选择器添加到 MachineSet 中：

```
$ oc patch MachineSet <name> --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>",<key>="<value>"}}]' -n openshift-machine-api 1
```

1 为每个节点选择器添加 **<key> /<value>** 对。

例如：

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p='[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}]' -n openshift-machine-api
```

b. 使用 **oc edit** 命令验证标签是否已添加到 MachineSet:

例如：

```
$ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

输出示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
...
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
```

c. 通过缩减至 **0** 并扩展节点来重新部署与该 MachineSet 关联的节点：

例如：

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

- d. 当节点就绪并可用时，使用 **oc get** 命令验证该标签是否已添加到节点：

```
$ oc get nodes -l <key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node
```

输出示例

```
NAME                                STATUS ROLES AGE VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp Ready worker 61s v1.18.3+002a51f
```

- 直接向节点添加标签：

- a. 为节点编辑 **Node** 对象：

```
$ oc label nodes <name> <key>=<value>
```

例如，若要为以下节点添加标签：

```
$ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 type=user-node region=east
```

- b. 使用 **oc get** 命令验证标签是否已添加到节点：

```
$ oc get nodes -l <key>=<value>,<key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS ROLES AGE VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 Ready worker 17m v1.18.3+002a51f
```

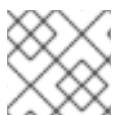
2.7.3. 创建项目范围节点选择器

您可以组合使用项目中的节点选择器和节点上的标签，将该项目中创建的所有 pod 限制到标记的节点。

当您在这个项目中创建 pod 时，OpenShift Container Platform 会将节点选择器添加到项目中 pod，并将 pod 调度到项目中具有匹配标签的节点。如果存在集群范围默认节点选择器，则以项目节点选择器为准。

您可以通过编辑 **Namespace** 对象来向项目添加标签，以添加 **openshift.io/node-selector** 参数，其中包

含标签定义。您可以通过编辑节点对象、MachineSet 或 MachineConfig 向节点添加标签。将标签添加到 MachineSet 可确保节点或机器停机时，新节点具有标签。如果节点或机器停机，添加到节点或 MachineConfig 的标签不会保留。



注意

您可以向 pod 添加额外的键/值对。但是，您无法为默认项目键添加其他值。

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:
  annotations:
    openshift.io/node-selector: "region=east"
...
```

以下节点具有 **type=user-node,region=east** 标签：

节点对象示例

```
apiVersion: v1
kind: Node
metadata:
  name: ci-ln-qg1il3k-f76d1-hlmhl-worker-b-df2s4
...
labels:
  region: east
  type: user-node
...
```

如果您在这个示例项目中创建 pod，则 pod 会被创建并带有项目节点选择器，并被调度到标记的节点：

Pod 对象示例

```
apiVersion: v1
kind: Pod
metadata:
...
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
...

spec:
  nodeSelector:
    region: west

....
```

在创建 pod 时，您会收到类似以下消息的错误：

错误信息示例

```
Error from server (Forbidden): error when creating "pod.yaml": pods "pod-4" is forbidden: pod node label selector conflicts with its project node label selector
```

流程

添加默认项目节点选择器：

1. 创建项目或编辑现有项目以添加 **openshift.io/node-selector** 参数：

```
$ oc edit project <name>
```

```
apiVersion: v1
kind: Project
metadata:
  annotations:
    openshift.io/node-selector: "type=user-node,region=east" ❶
    openshift.io/sa.scc.mcs: s0:c17,c14
    openshift.io/sa.scc.supplemental-groups: 1000300000/10000
    openshift.io/sa.scc.uid-range: 1000300000/10000
  creationTimestamp: 2019-06-10T14:39:45Z
  labels:
    openshift.io/run-level: "0"
  name: demo
  resourceVersion: "401885"
  selfLink: /api/v1/namespaces/openshift-kube-apiserver
  uid: 96ecc54b-8b8d-11e9-9f54-0a9ae641edd0
spec:
  finalizers:
    - kubernetes
status:
  phase: Active
```

- ❶ 使用适当的 **<key>:<value>** 对添加 **openshift.io/node-selector**。

2. 通过使用 MachineSet 或直接编辑节点，为节点添加标签：

- 在创建节点时, 使用 MachineSet 将标签添加到由 MachineSet 管理的节点上 :

- a. 运行以下命令, 将节点选择器添加到 MachineSet 中 :

```
$ oc patch MachineSet <name> --type='json' -
p=[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"<key>="
<value>",<key>="<value>"}}} -n openshift-machine-api
```

例如 :

```
$ oc patch MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c --type='json' -
p=[{"op":"add","path":"/spec/template/spec/metadata/labels", "value":{"type":"user-
node","region":"east"}}] -n openshift-machine-api
```

- b. 使用 **oc edit** 命令验证标签是否已添加到 MachineSet:

例如 :

```
$ oc edit MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-machine-api
```

输出示例

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
...
spec:
...
  template:
    metadata:
...
    spec:
      metadata:
        labels:
          region: east
          type: user-node
```

- c. 重新部署与该 MachineSet 关联的节点 :

例如 :

```
$ oc scale --replicas=0 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-
machine-api
```

```
$ oc scale --replicas=1 MachineSet ci-ln-l8nry52-f76d1-hl7m7-worker-c -n openshift-
machine-api
```

- d. 在节点就绪并可用时, 使用 **oc get** 命令验证该标签是否已添加到节点 :

```
$ oc label MachineSet abc612-msrtw-worker-us-east-1c type=user-node region=east
```

例如 :

```
$ oc get nodes -l type=user-node
```

输出示例

```
NAME                                STATUS ROLES  AGE  VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-c-vmqzp Ready  worker  61s  v1.18.3+002a51f
```

- 直接向节点添加标签：

- a. 编辑 **Node** 对象以添加标签：

```
$ oc label <resource> <name> <key>=<value>
```

例如，若要为以下节点添加标签：

```
$ oc label nodes ci-ln-l8nry52-f76d1-hl7m7-worker-c-tgq49 type=user-node
region=east
```

- b. 使用 **oc get** 命令验证标签是否已添加到节点：

```
$ oc get nodes -l <key>=<value>
```

例如：

```
$ oc get nodes -l type=user-node,region=east
```

输出示例

```
NAME                                STATUS ROLES  AGE  VERSION
ci-ln-l8nry52-f76d1-hl7m7-worker-b-tgq49 Ready  worker  17m  v1.18.3+002a51f
```

第 3 章 使用作业和 DAEMONSET

3.1. 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 DaemonSet 在 OpenShift Container Platform 集群的特定节点或所有节点上运行 pod 副本。

DaemonSet 可确保所有（或某些）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 DaemonSet 会清理它所创建的 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，只有集群管理员才能创建 daemonset。

如需有关 daemonset 的更多信息，请参阅 [Kubernetes 文档](#)。



重要

Daemonset 调度与项目的默认节点选择器不相兼容。如果您没有成功禁用，daemonset 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。

3.1.1. 通过默认调度程序调度

DaemonSet 可确保符合条件的所有节点都运行 Pod 的副本。通常，Kubernetes 调度程序会选择要在其上运行 Pod 的节点。然而，DaemonSet 控制器创建并调度了前面的 daemonSet pod。这会引发以下问题：

- pod 行为不一致：等待调度的普通 pod 已创建好并处于待处理状态，但没有以待处理状态创建 DaemonSet pod。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，DaemonSet 控制器将在不考虑 pod 优先级和抢占的前提下做出调度决策。

OpenShift Container Platform 中默认启用 ScheduleDaemonSetPods，以便使用默认调度程序而非 DaemonSet 控制器来调度 DaemonSet，具体方法是添加 NodeAffinity 术语到 DaemonSet pod，而不是添加 .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 DaemonSet pod 的节点关联性已经存在，它会被替换掉。DaemonSet 控制器仅在创建或修改 DaemonSet pod 时执行这些操作，且不会对 DaemonSet 的 spec.template 进行任何更改。

```
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 DaemonSet Pod 中。默认调度程序在调度 DaemonSet Pod 时会忽略不可调度的节点。

3.1.2. 创建 daemonset

在创建 daemonset 时，请使用 **nodeSelector** 字段来指示 daemonset 应该在其上部署副本的节点。

先决条件

- 在开始使用 daemonset 之前，请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": ""}}}'
```

- 如果要创建新项目，请使用 **oc adm new-project <name> --node-selector=""** 覆盖默认节点选择器。

流程

创建 daemonset：

1. 定义 daemonset yaml 文件：

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
    matchLabels:
      name: hello-daemonset ❶
  template:
    metadata:
      labels:
        name: hello-daemonset ❷
    spec:
      nodeSelector: ❸
        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
```

- ❶ 决定哪些 pod 属于 daemonset 的标签选择器。
- ❷ pod 模板的标签选择器。必须与上述标签选择器匹配。
- ❸ 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 daemonset 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
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
Node:    openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9|grep Node
Node:    openshift-node02.hostname.com/10.14.20.137
```

重要

- 如果更新 daemonset 的 pod 模板，现有的 pod 副本不受影响。
- 如果您删除了 daemonSet，然后在创建新 daemonset 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，daemonset 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 daemonset，请通过删除旧副本或旧节点来强制创建新的 pod 副本。

3.2. 使用作业在 POD 中运行任务

作业 (job) 在 OpenShift Container Platform 集群中执行某项任务。

作业会跟踪任务的整体进度，并使用活跃、成功和失败 pod 的相关信息来更新其状态。删除作业会清理它创建的所有 pod 副本。作业是 Kubernetes API 的一部分，可以像其他对象类型一样通过 **oc** 命令进行管理。

作业规格示例

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1 1
  completions: 1 2
  activeDeadlineSeconds: 1800 3
  backoffLimit: 6 4
  template: 5
    metadata:
      name: pi
    spec:
      containers:
        - name: pi
          image: perl
          command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure 6
```

1. 作业应并行运行的 pod 副本。
2. pod 成功完成后需要标记为作业也完成。
3. 作业可以运行的最长时间。
4. 作业的重试次数。
5. 控制器创建的 pod 模板。
6. pod 的重启策略。

如需有关作业的更多信息，请参阅 [Kubernetes 文档](#)。

3.2.1. 了解作业和 CronJob

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。删除作业会清除它创建的所有 pod。作业是 Kubernetes API 的一部分，可以像其他对象类型一样通过 `oc` 命令进行管理。

OpenShift Container Platform 中有两种资源类型可以创建只运行一次的对象：

作业

常规作业是一种只运行一次的对象，它会创建一个任务并确保作业完成。

有三种适合作为作业运行的任务类型：

- 非并行作业：
 - 仅启动一个 Pod 的作业，除非 Pod 失败。
 - 一旦 Pod 成功终止，作业就会马上完成。
- 带有固定完成计数的并行作业：
 - 启动多个 pod 的作业。
 - Job 代表整个任务，并在 1 到 **completions** 范围内的每个值都有一个成功 Pod 时完成。
- 带有工作队列的并行作业：
 - 在一个给定 pod 中具有多个并行 worker 进程的作业。
 - OpenShift Container Platform 协调 pod，以确定每个 pod 都应该使用什么作业，或使用一个外部队列服务。
 - 每个 Pod 都可以独立决定是否所有对等 pod 都已完成（整个作业完成）。
 - 当所有来自作业的 Pod 都成功终止时，不会创建新的 Pod。
 - 当至少有一个 Pod 成功终止并且所有 Pod 都终止时，作业成功完成。
 - 当任何 Pod 成功退出时，其他 Pod 都不应该为这个任务做任何工作或写任何输出。Pod 都应该处于退出过程中。

如需有关如何使用不同类型的作业的更多信息，请参阅 Kubernetes 文档中的 [作业模式](#)。

CronJob

通过使用 CronJob，可以调度一个作业运行多次。

CronJob 基于常规作业构建，允许您指定作业的运行方式。CronJob 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。

CronJob 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。CronJob 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。



警告

CronJob 大致会在调度的每个执行时间创建一个作业对象，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。

3.2.2. 了解如何创建作业

两种资源类型都需要一个由以下关键部分组成的作业配置：

- pod 模板，用于描述 OpenShift Container Platform 创建的 pod。
- **parallelism** 参数，用于指定在任意时间点上应并行运行多少个 pod 来执行某个作业。
 - 对于非重复作业，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。
 - 对于非重复作业，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，请保留未设置。当取消设置默认为 **parallelism** 值。

3.2.2.1. 了解如何为作业设置最长持续时间

在定义作业时，您可以通过设置 **activeDeadlineSeconds** 字段来定义其最长持续时间。以秒为单位指定，默认情况下不设置。若未设置，则不强制执行最长持续时间。

最长持续时间从系统中调度第一个 pod 的时间开始计算，并且定义作业在多久时间内处于活跃状态。它将跟踪整个执行时间。达到指定的超时后，OpenShift Container Platform 将终止作业。

3.2.2.2. 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。

3.2.2.3. 了解如何配置 CronJob 以移除工件

CronJob 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。CronJob 规格内有两个字段负责这一事务：

- **.spec.successfulJobsHistoryLimit**。要保留的成功完成作业数（默认为 3）。
- **.spec.failedJobsHistoryLimit**。要保留的失败完成作业数（默认为 1）。

提示

- 删除您不再需要的 CronJob：

```
$ oc delete cronjob/<cron_job_name>
```

这样可防止生成不必要的工件。

- 您可以通过将 **spec.suspend** 设置为 true 来挂起后续执行。所有后续执行都会挂起，直到重置为 **false**。

3.2.3. 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。策略仅设定由哪一个子系统执行重启。

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。

3.2.4. 创建作业

您可以通过创建作业对象在 OpenShift Container Platform 中创建作业。

流程

创建作业：

1. 创建一个类似以下示例的 YAML 文件：

```
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. 可选值，定义一个作业应并行运行多少个 pod 副本；默认与 **completions** 相同。
 - 对于非并行作业，请保留未设置。当取消设置时，默认为 **1**。
2. 另外，还可指定标记作业完成时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. 可选值，定义作业可以运行的最长持续时间。
4. 另外，还可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。

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

1. 创建作业：

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



注意

您还可以使用 **oc run**，在一个命令中创建并启动作业。以下命令会创建并启动与上一示例中指定的相同的作业：

```
$ oc run pi --image=perl --replicas=1 --restart=OnFailure \
  --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

3.2.5. 创建 CronJob

您可以通过创建作业对象在 OpenShift Container Platform 中创建 CronJob。

流程

创建 CronJob：

1. 创建一个类似以下示例的 YAML 文件：

```

apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/* * * * *" 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 1 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 2 2 可选的并发策略，指定如何对待 CronJob 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 CronJob 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 3 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 4 4 可选标志，允许挂起 CronJob。若设为 **true**，则会挂起所有后续执行。
- 5 5 5 要保留的成功完成作业数（默认为 3）。
- 6 6 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 CronJob 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 CronJob :

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



注意

您还可以使用 **oc run**，在一个命令中创建并启动 CronJob。以下命令会创建并启动与上一示例中指定的相同的 CronJob :

```
$ oc run pi --image=perl --schedule='*/1 * * * *' \
  --restart=OnFailure --labels parent="cronjobpi" \
  --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

使用 **oc run** 时，**--schedule** 选项接受采用 [cron 格式](#) 的调度计划。

在创建 CronJob 时，**oc run** 仅支持 **Never** 或 **OnFailure** 重启策略 (**--restart**)。

第 4 章 操作节点

4.1. 查看和列出 OPENSIFT CONTAINER PLATFORM 集群中的节点

您可以列出集群中的所有节点，以获取节点的相关信息，如状态、年龄、内存用量和其他详情。

在执行节点管理操作时，CLI 与代表实际节点主机的节点对象交互。主控机（master）使用来自节点对象的信息执行健康检查，以此验证节点。

4.1.1. 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
```

NAME	STATUS	ROLES	AGE	VERSION
master.example.com	Ready	master	7h	v1.16.2
node1.example.com	Ready	worker	7h	v1.16.2
node2.example.com	Ready	worker	7h	v1.16.2

- **-wide** 选项可提供所有节点的附加信息。

```
$ oc get nodes -o wide
```

- 以下命令列出一个节点的相关信息：

```
$ oc get node <node>
```

这些命令输出中的 **STATUS** 列可显示节点的以下状况：

表 4.1. 节点状况

状况	描述
Ready	节点通过返回 True 来报告其自身对 apiserver 的就绪度。
NotReady	其中一个底层组件（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。

- 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
```

```
Name:          node1.example.com 1
```

```

Roles:          worker 2
Labels:         beta.kubernetes.io/arch=amd64 3
                beta.kubernetes.io/instance-type=m4.large
                beta.kubernetes.io/os=linux
                failure-domain.beta.kubernetes.io/region=us-east-2
                failure-domain.beta.kubernetes.io/zone=us-east-2a
                kubernetes.io/hostname=ip-10-0-140-16
                node-role.kubernetes.io/worker=
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://1.16.0-0.6.dev.rhaos4.3.git9ad059b.el8-rc2
Kubelet Version:       v1.16.2
Kube-Proxy Version:    v1.16.2
PodCIDR:               10.128.4.0/24
ProviderID:            aws:///us-east-2a/i-04e87b31dc6b3e171
Non-terminated Pods:   (13 in total) 10

```

Namespace	Name	CPU Requests	CPU Limits
openshift-cluster-node-tuning-operator	tuned-hdl5q	0 (0%)	0 (0%)
openshift-dns	dns-default-l69zr	0 (0%)	0 (0%)
openshift-image-registry	node-ca-9hmcg	0 (0%)	0 (0%)
openshift-ingress	router-default-76455c45c-c5ptv	0 (0%)	0 (0%)
openshift-machine-config-operator	machine-config-daemon-cvqw9	20m (1%)	0
openshift-marketplace	community-operators-f67fh	0 (0%)	0 (0%)
openshift-monitoring	alertmanager-main-0	50m (3%)	50m (3%)
openshift-monitoring	grafana-78765ddcc7-hnjmm	100m (6%)	200m
openshift-monitoring	node-exporter-l7q8d	10m (0%)	20m (1%)
openshift-monitoring	prometheus-adapter-75d769c874-hvb85	0 (0%)	0
openshift-multus	multus-kw8w5	0 (0%)	0 (0%)
openshift-sdn	ovs-t4dsn	100m (6%)	0 (0%)
openshift-sdn	sdn-g79hg	100m (6%)	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-efs	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 节点状况。
- 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。

4.1.2. 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 --selector=<node_selector>
$ oc describe -l=<pod_selector>
```

例如：

```
$ oc describe node --selector=beta.kubernetes.io/os
$ oc describe node -l node-role.kubernetes.io/worker
```

4.1.3. 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
```

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 `!=`。

4.2. 操作节点

作为管理员，您可以执行若干任务来提高集群的效率。

4.2.1. 了解如何撤离节点上的 pod

通过撤离 pod，您可以迁移给定的一个或多个节点上的所有或选定 pod。

您只能撤离由复制控制器支持的 pod。复制控制器在其他节点上创建新 pod，并从指定节点移除现有的 pod。

裸机 pod（即不由复制控制器支持的 pod）默认情况下不受影响。您可以通过指定 pod 选择器来撤离一小部分 pod。pod 选择器基于标签，因此带有指定标签的所有 pod 都将被撤离。

注意

节点必须首先标记为不可调度，然后才能执行 pod 撤离。

```
$ oc adm cordon <node1>
node/<node1> cordoned
```

```
$ oc get node <node1>
NAME          STATUS          ROLES    AGE    VERSION
<node1>      NotReady,SchedulingDisabled  worker  1d     v1.16.2
```

完成后，使用 **oc adm uncordon** 将节点标记为可以调度。

```
$ oc adm uncordon <node1>
```

- 以下命令撤离一个或多个节点上的所有或选定 pod：

```
$ oc adm drain <node1> <node2> [--pod-selector=<pod_selector>]
```

- 以下命令使用 **--force** 选项来强制删除裸机 pod。设为 **true** 时，即使存在不由复制控制器、ReplicaSet、作业、daemonset 或 StatefulSet 管理的 pod，也会继续执行删除：

```
$ oc adm drain <node1> <node2> --force=true
```

- 以下命令使用 **--grace-period** 以秒为单位设置一个期限，以便各个 pod 能够安全地终止。如果为负，则使用 pod 中指定的默认值：

```
$ oc adm drain <node1> <node2> --grace-period=-1
```

- 以下命令使用了 **--ignore-daemonsets** 标记（设为 **true**），以忽略由 DaemonSet 管理的 pod：

```
$ oc adm drain <node1> <node2> --ignore-daemonsets=true
```

- 以下命令使用 **--timeout** 标记来设置在放弃前要等待的时长。值为 **0** 时设定无限时长：

```
$ oc adm drain <node1> <node2> --timeout=5s
```

- 以下命令使用了 **--delete-local-data** 标记（设为 **true**），在存在使用 emptyDir 的 pod 时也仍然删除 pod。节点排空时会删除本地数据：

```
$ oc adm drain <node1> <node2> --delete-local-data=true
```

- 以下命令使用了 **--dry-run** 选项（设为 **true**），它会列出将要迁移的对象而不实际执行撤离：

```
$ oc adm drain <node1> <node2> --dry-run=true
```

您可以使用 **--selector=<node_selector>** 选项来撤离选定节点上的 pod，而不指定具体的节点名称（如 **<node1> <node2>**）。

4.2.2. 了解如何更新节点上的标签

您可以更新节点上的任何标签。

节点标签不会在节点删除后保留，即使机器备份了节点也是如此。



注意

对 MachineSet 的任何更改都不会应用到 MachineSet 拥有的现有机器。例如，对现有 MachineSet 编辑或添加的标签不会传播到与该 MachineSet 关联的现有机器和节点。

- 以下命令在节点上添加或更新标签：

```
$ oc label node <node> <key_1>=<value_1> ... <key_n>=<value_n>
```

例如：

```
$ oc label nodes webconsole-7f7f6 unhealthy=true
```

- 以下命令更新命名空间中的所有 pod：

```
$ oc label pods --all <key_1>=<value_1>
```

例如：

```
$ oc label pods --all status=unhealthy
```

4.2.3. 了解如何将节点标记为不可调度或可以调度

默认情况下，具有 **Ready** 状态的健康节点被标记为可以调度，即允许在该节点上放置新的 pod。如果手动将节点标记为不可调度，则会阻止在该节点上调度任何新的 pod。节点上的现有 pod 不受影响。

- 以下命令将一个或多个节点标记为不可调度：

```
$ oc adm cordon <node>
```

例如：

```
$ oc adm cordon node1.example.com
node/node1.example.com cordoned
```

NAME	LABELS	STATUS
node1.example.com	kubernetes.io/hostname=node1.example.com	Ready,SchedulingDisabled

- 以下命令将当前不可调度的一个或多个节点标记为可以调度：

```
$ oc adm uncordon <node1>
```

另外，您也可以使用 **--selector=<node_selector>** 选项将选定的节点标记为可以调度或不可调度，而不指定具体的节点名称（如 **<node>**）。

4.2.4. 将 master 节点配置为可以调度

自 OpenShift Container Platform 4.2 起，您可以将 master 节点配置为可以调度，即允许在 master 节点上放置新的 Pod。默认情况下，master 节点不可调度。但是，如果集群不包含任何 worker 节点，那么 master 节点会默认标记为可以调度。



重要

在版本 4.3 中，创建没有 worker 节点的集群只能在裸机上部署，且为一个技术预览功能。对于所有其他集群类型，您可以将 master 设置为可调度，但必须保留 worker 节点。

您可以通过配置 **mastersSchedulable** 字段，来允许或禁止调度 master 节点。

流程

1. 编辑 **schedulers.config.openshift.io** 资源。

```
$ oc edit schedulers.config.openshift.io cluster
```

2. 配置 **mastersSchedulable** 字段。

```
apiVersion: config.openshift.io/v1
kind: Scheduler
metadata:
  creationTimestamp: "2019-09-10T03:04:05Z"
  generation: 1
  name: cluster
  resourceVersion: "433"
  selfLink: /apis/config.openshift.io/v1/schedulers/cluster
  uid: a636d30a-d377-11e9-88d4-0a60097bee62
spec:
  mastersSchedulable: false 1
  policy:
    name: ""
  status: {}
```

- 1** 设置为 **true** 以允许调度 master 节点，或设置为 **false** 以禁止调度 master 节点。

3. 保存文件以使改变生效。

4.2.5. 删除节点

4.2.5.1. 从集群中删除节点

当您使用 CLI 删除节点时，节点对象会从 Kubernetes 中删除，但该节点上存在的 Pod 不会被删除。任何未由复制控制器支持的裸机 Pod 都无法从 OpenShift Container Platform 访问。由复制控制器支持的 Pod 会重新调度到其他可用的节点。您必须删除本地清单 Pod。

流程

要从 OpenShift Container Platform 集群中删除节点，请编辑相应的 MachineSet：



注意

如果您在裸机上运行集群，则无法通过编辑 MachineSet 来删除节点。只有在集群与云供应商集成时才会使用 MachineSet。相反，您必须在手动删除前取消调度并排空节点。

1. 查看集群中的 MachineSet :

```
$ oc get machinesets -n openshift-machine-api
```

MachineSet 以 <clusterid>-worker-<aws-region-az> 的形式列出。

2. 缩放 MachineSet :

```
$ oc scale --replicas=2 machineset <machineset> -n openshift-machine-api
```

如需有关使用 MachineSet 缩放集群的更多信息，请参见“手动缩放 MachineSet”。

4.2.5.2. 从裸机集群中删除节点

当您使用 CLI 删除节点时，节点对象会从 Kubernetes 中删除，但该节点上存在的 Pod 不会被删除。任何未由复制控制器支持的裸机 Pod 都无法从 OpenShift Container Platform 访问。由复制控制器支持的 Pod 会重新调度到其他可用的节点。您必须删除本地清单 Pod。

流程

通过完成以下步骤，从裸机上运行的 OpenShift Container Platform 集群中删除节点：

1. 将节点标记为不可调度。

```
$ oc adm cordon <node_name>
```

2. 排空节点上的所有 Pod:

```
$ oc adm drain <node_name> --force=true
```

3. 从集群中删除您的节点：

```
$ oc delete node <node_name>
```

虽然节点对象现已从集群中删除，但它仍然可在重启后或 kubelet 服务重启后重新加入集群。要永久删除该节点及其所有数据，您必须[弃用该节点](#)。

4.2.6. 为节点添加内核参数

在一些特殊情况下，您可能需要为集群中的一组节点添加内核参数。进行此操作时应小心谨慎，而且您必须先清楚了解所设参数的影响。



警告

不当使用内核参数会导致系统变得无法引导。

您可以设置的内核参数示例包括：

- **selinux=0**：禁用 Security Enhanced Linux (SELinux)。虽然不建议在生产环境中这样操作，但禁用 SELinux 可将性能提高 2%- 3%。
- **nosmt**：在内核中禁用对称多线程 (SMT)。多线程允许每个 CPU 有多个逻辑线程。您可以在多租户环境中考虑使用 **nosmt**，以减少潜在的跨线程攻击风险。禁用 SMT 在本质上相当于选择安全性而非性能。

如需内核参数的列表和描述，请参阅 [Kernel.org](https://kernel.org) 内核参数。

在下列步骤中，您要创建一个用于标识以下对象的 MachineConfig：

- 您要添加内核参数的一组机器。本例中为具有 worker 角色的机器。
- 附加到现有内核参数末尾的内核参数。
- 指示 MachineConfig 列表中应用更改的位置的标签。

先决条件

- 具有正常运行的 OpenShift Container Platform 集群的管理特权。

流程

1. 列出 OpenShift Container Platform 集群的现有 MachineConfig，以确定如何标记您的 MachineConfig：

```
$ oc get MachineConfig
NAME                                     GENERATEDBYCONTROLLER
IGNITIONVERSION  CREATED
00-master                                     577c2d527b09cd7a481a162c50592139caa15e20
2.2.0      30m
00-worker                                     577c2d527b09cd7a481a162c50592139caa15e20
2.2.0      30m
01-master-container-runtime
577c2d527b09cd7a481a162c50592139caa15e20  2.2.0      30m
01-master-kubelet
577c2d527b09cd7a481a162c50592139caa15e20  2.2.0      30m
01-worker-container-runtime
577c2d527b09cd7a481a162c50592139caa15e20  2.2.0      30m
01-worker-kubelet
577c2d527b09cd7a481a162c50592139caa15e20  2.2.0      30m
99-master-1131169f-dae9-11e9-b5dd-12a845e8ffd8-registries
577c2d527b09cd7a481a162c50592139caa15e20  2.2.0      30m
99-master-ssh                                     2.2.0      30m
99-worker-114e8ac7-dae9-11e9-b5dd-12a845e8ffd8-registries
```

```

577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      30m
99-worker-ssh                               2.2.0      30m
rendered-master-b3729e5f6124ca3678188071343115d0
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      30m
rendered-worker-18ff9506c718be1e8bd0a066850065b7
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      30m

```

2. 创建一个用来标识内核参数的 MachineConfig 文件（如 **05-worker-kernelarg-selinuxoff.yaml**）

```

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker 1
  name: 05-worker-kernelarg-selinuxoff 2
spec:
  config:
    ignition:
      version: 2.2.0
  kernelArguments:
    - selinux=0 3

```

- 1** 仅将新内核参数应用到 worker 节点。
- 2** 用于标识它插入到 MachineConfig 中的什么位置 (05) 以及发挥什么作用（添加一个内核参数来关闭 SELinux）。
- 3** 将确切的内核参数标识为 **selinux=0**。

3. 创建新的 MachineConfig：

```
$ oc create -f 05-worker-kernelarg-selinuxoff.yaml
```

4. 检查 MachineConfig 以查看是否添加了新配置：

```

$ oc get MachineConfig
NAME                                     GENERATEDBYCONTROLLER
IGNITIONVERSION  CREATED
00-master                577c2d527b09cd7a481a162c50592139caa15e20
2.2.0                    31m
00-worker                577c2d527b09cd7a481a162c50592139caa15e20
2.2.0                    31m
01-master-container-runtime
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
01-master-kubelet
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
01-worker-container-runtime
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
01-worker-kubelet
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m

05-worker-kernelarg-selinuxoff          2.2.0
105s

```



```

99-master-1131169f-dae9-11e9-b5dd-12a845e8ffd8-registries
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
99-master-ssh                                     2.2.0      30m
99-worker-114e8ac7-dae9-11e9-b5dd-12a845e8ffd8-registries
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
99-worker-ssh                                     2.2.0      31m
rendered-master-b3729e5f6124ca3678188071343115d0
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m
rendered-worker-18ff9506c718be1e8bd0a066850065b7
577c2d527b09cd7a481a162c50592139caa15e20 2.2.0      31m

```

5. 检查节点：

```

$ oc get nodes
NAME                                STATUS              ROLES    AGE  VERSION
ip-10-0-136-161.ec2.internal        Ready              worker   28m  v1.16.2
ip-10-0-136-243.ec2.internal        Ready              master   34m  v1.16.2
ip-10-0-141-105.ec2.internal        Ready,SchedulingDisabled worker   28m  v1.16.2
ip-10-0-142-249.ec2.internal        Ready              master   34m  v1.16.2
ip-10-0-153-11.ec2.internal         Ready              worker   28m  v1.16.2
ip-10-0-153-150.ec2.internal        Ready              master   34m  v1.16.2

```

您可以发现，在应用更改时每个 worker 节点上的调度都会被禁用。

6. 前往其中一个 worker 节点并列出内核命令行参数（主机上的 `/proc/cmdline` 中），以检查内核参数确实已发挥作用：

```

$ oc debug node/ip-10-0-141-105.ec2.internal
Starting pod/ip-10-0-141-105ec2internal-debug ...
To use host binaries, run `chroot /host`

sh-4.3# cat /host/proc/cmdline
BOOT_IMAGE=/ostree/rhcos-... console=tty0 console=ttyS0,115200n8
rootflags=defaults,prjquota rw root=UUID=fd0... ostree=/ostree/boot.0/rhcos/16...
coreos.oem.id=qemu coreos.oem.id=ec2 ignition.platform.id=ec2 selinux=0

sh-4.3# exit

```

您应该会看到 `selinux=0` 参数已添加至其他内核参数。

4.2.7. 其他资源

如需有关使用 MachineSet 缩放集群的更多信息，请参见[手动缩放 MachineSet](#)。

4.3. 管理节点

OpenShift Container Platform 使用 KubeletConfig 自定义资源来管理节点配置。通过创建 KubeletConfig 实例来创建受管 MachineConfig，从而覆盖节点上的设置。



注意

不支持为更改配置而登录远程机器。

4.3.1. 修改节点

要对集群或 MachinePool 进行配置更改，您必须创建自定义资源定义或 KubeletConfig 实例。OpenShift Container Platform 使用 Machine Config Controller 来监控是否通过 CRD 进行了任何更改，并将更改应用于集群。

流程

1. 为您要配置节点类型，获取与静态 CRD (Machine Config Pool) 关联的标签。执行以下步骤之一：

- a. 检查所需 machineconfigpool 的当前标签。

例如：

```
$ oc get machineconfigpool --show-labels
NAME CONFIG UPDATED UPDATING DEGRADED
LABELS
master rendered-master-e05b81f5ca4db1d249a1bf32f9ec24fd True False
False operator.machineconfiguration.openshift.io/required-for-upgrade=
worker rendered-worker-f50e78e1bc06d8e82327763145bfcf62 True False
False
```

- b. 为所需的 machineconfigpool 添加自定义标签。

例如：

```
$ oc label machineconfigpool worker custom-kubelet=enabled
```

2. 为您的配置更改创建 KubeletConfig 自定义资源 (CR)。

例如：

custom-config CR 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-config ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: enabled ❷
  kubeletConfig: ❸
    podsPerCore: 10
    maxPods: 250
    systemReserved:
      cpu: 1000m
      memory: 500Mi
    kubeReserved:
      cpu: 1000m
      memory: 500Mi
```

❶ 为 CR 分配一个名称。

❷ 指定要应用配置更改的标签，这是您添加到 machineconfigpool 中的标签。

3 指定要更改的新值。

3. 创建 CR 对象。

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

例如：

```
$ oc create -f master-kube-config.yaml
```

大多数 `KubeletConfig` 选项可由用户设置。不允许覆盖下列选项：

- CgroupDriver
- ClusterDNS
- ClusterDomain
- RuntimeRequestTimeout
- StaticPodPath

4.4. 管理每个节点的 POD 数量上限

在 OpenShift Container Platform 中，您可以根据节点上的处理器内核数和/或硬限制，来配置可在节点上运行的 pod 数量。如果您同时使用这两个选项，则取两者中较小的限制来限制节点上的 pod 数。

超过这些值可能会导致：

- OpenShift Container Platform CPU 使用率提高。
- pod 调度缓慢。
- 潜在的内存不足情形，具体取决于节点中的内存量。
- IP 地址池耗尽。
- 资源过量使用，导致用户应用程序性能变差。



注意

包含单个容器的一个 pod 实际上会使用两个容器。第二个容器在容器实际启动前先设置了网络。因此，运行 10 个 pod 的节点实际上运行有 20 个容器。

`PodsPerCore` 参数根据节点的处理器内核数限制节点上可运行的 pod 数量。例如，如果将一个有 4 个处理器内核的节点上的 `PodsPerCore` 设置为 10，则该节点上允许的 pod 数量上限为 40。

`maxPods` 参数将节点上可运行的 pod 数量限制为一个固定值，不考虑节点的属性。

4.4.1. 配置每个节点的 Pod 数量上限

有两个参数控制可调度到节点的 pod 数量上限，分别为 `PodsPerCore` 和 `maxPods`。如果您同时使用这两个选项，则取两者中较小的限制来限制节点上的 pod 数。

例如，如果将一个有 4 个处理器内核的节点上的 `PodsPerCore` 设置为 **10**，则该节点上允许的 pod 数量上限为 40。

前提条件

1. 为您要配置的节点类型获取与静态机器配置池 CRD 关联的标签。执行以下步骤之一：

- a. 查看 Machine Config Pool：

```
$ oc describe machineconfigpool <name>
```

例如：

```
$ oc describe machineconfigpool worker

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods 1
```

- 1** 如果添加了标签，它会出现在 **labels** 下。

- b. 如果标签不存在，则添加一个键/值对：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

max-pods CR 配置示例

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-max-pods 1
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods 2
  kubeletConfig:
    podsPerCore: 10 3
    maxPods: 250 4
```

- 1** 为 CR 分配一个名称。
- 2** 指定要应用配置更改的标签。
- 3** 根据节点的处理器内核数限制节点上可运行的 pod 数量。

- 4 将节点上可运行的 pod 数量指定为一个固定值，而不考虑节点的属性。



注意

将 `PodsPerCore` 设置为 0 可禁用此限制。

在上例中，`PodsPerCore` 的默认值为 10，`maxPods` 的默认值则为 250。这意味着，除非节点有 25 个以上的内核，否则 `PodsPerCore` 就是默认的限制因素。

- 列出 Machine Config Pool CRD 来查看是否应用了更改。如果 Machine Config Controller 抓取到更改，则 **UPDATING** 列会报告 **True**：

```
$ oc get machineconfigpools
NAME CONFIG UPDATED UPDATING DEGRADED
master master-9cc2c72f205e103bb534 False False False
worker worker-8cecd1236b33ee3f8a5e False True False
```

更改完成后，**UPDATED** 列会报告 **True**。

```
$ oc get machineconfigpools
NAME CONFIG UPDATED UPDATING DEGRADED
master master-9cc2c72f205e103bb534 False True False
worker worker-8cecd1236b33ee3f8a5e True False False
```

4.5. 使用 NODE TUNING OPERATOR

了解 Node Tuning Operator，以及如何使用它通过编排 tuned 守护进程以管理节点级别的性能优化。

4.5.1. 关于 Node Tuning Operator

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。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。Node Tuning Operator 把为 OpenShift Container Platform 容器化的 Tuned 守护进程作为一个 Kubernetes DaemonSet 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。

在版本 4.1 及更高版本中，OpenShift Container Platform 标准安装中包含了 Node Tuning Operator。

4.5.2. 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：

```
$ oc get Tuned/default -o yaml -n openshift-cluster-node-tuning-operator
```

默认 CR 旨在为 OpenShift Container Platform 平台提供标准的节点级性能优化，它只能被修改来设置 Operator Management 状态。Operator 将覆盖对默认 CR 的任何其他自定义更改。若进行自定义性能优化，请创建自己的 Tuned CR。新创建的 CR 将与默认的 CR 合并，并基于节点或 pod 标识和配置文件优先级对节点应用自定义调整。

4.5.3. 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。

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 plug-ins 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:
- match:                # optional; if omitted, profile match is assumed unless a profile with a
  higher matches first
  <match>                # an optional array
  priority: <priority>   # profile ordering priority, lower numbers mean higher priority (0 is the
  highest priority)
  profile: <tuned_profile_name> # e.g. tuned_profile_1

# ...

- match:
```

```
<match>
priority: <priority>
profile: <tuned_profile_name>    # e.g. tuned_profile_n
```

如果省略 **<match>**，则假定配置集匹配（如 **true**）。

<match> 是一个递归定义的可选数组，如下所示：

```
- label: <label_name>    # node or Pod label name
  value: <label_value>  # optional node or Pod label value; if omitted, the presence of <label_name>
  is enough to match
  type: <label_type>    # optional node or Pod type (`node` or `pod`); if omitted, `node` is assumed
  <match>              # an optional <match> array
```

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>** 数组中任何一项，整个 **<match>** 数组评估为 **true**。因此，该数组以逻辑 OR 运算来运作。

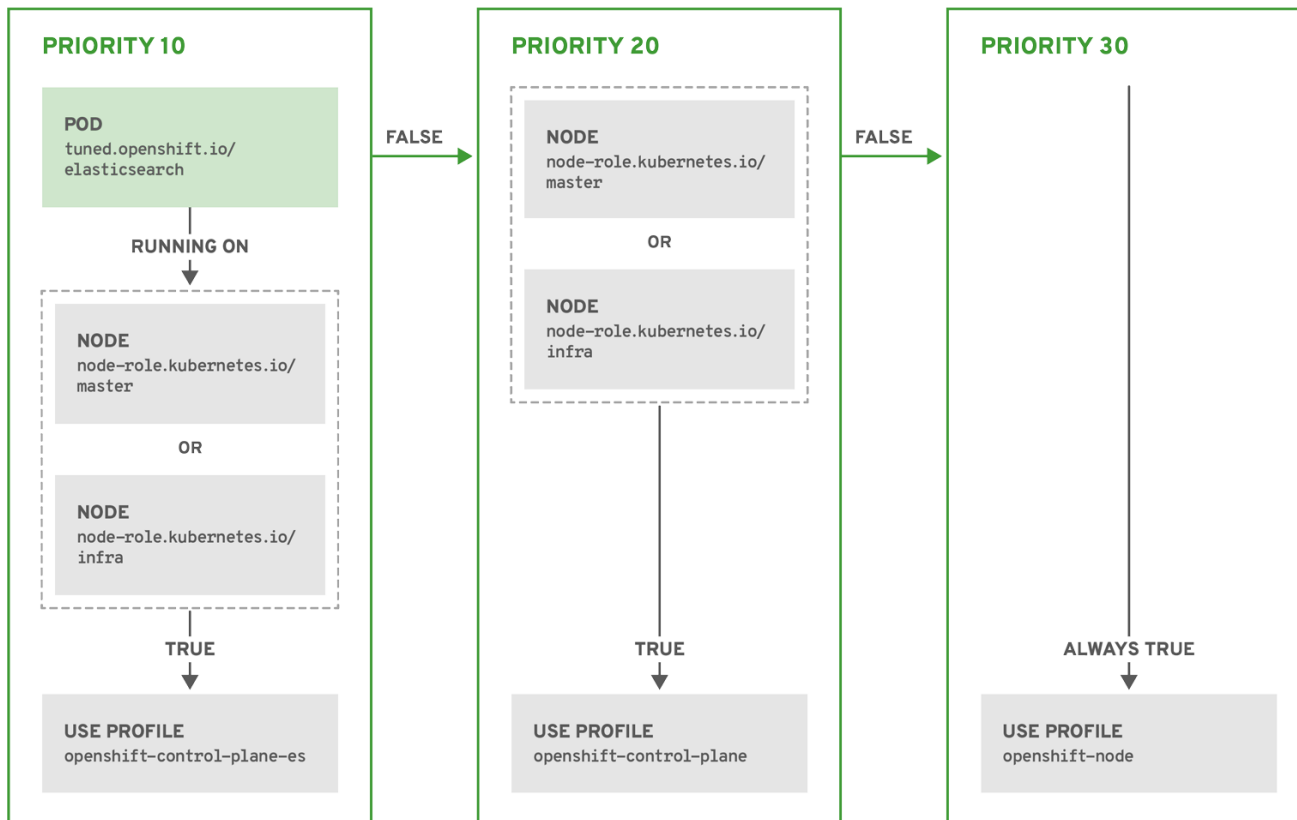
示例

```
- 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。



OPENSIFT_10_0319

4.5.4. 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/v1alpha1
kind: Tuned
metadata:
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - name: "openshift"
    data: |
      [main]
      summary=Optimize systems running OpenShift (parent profile)
      include=${f:virt_check:virtual-guest:throughput-performance}
      [selinux]
      avc_cache_threshold=8192
      [net]
      nf_conntrack_hashsize=131072
      [sysctl]
      net.ipv4.ip_forward=1
      kernel.pid_max=>131072
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.neigh.default.gc_thresh1=8192
      net.ipv4.neigh.default.gc_thresh2=32768
      net.ipv4.neigh.default.gc_thresh3=65536
      net.ipv6.neigh.default.gc_thresh1=8192
      net.ipv6.neigh.default.gc_thresh2=32768
  
```



```

net.ipv6.neigh.default.gc_thresh3=65536
[sysfs]
/sys/module/nvme_core/parameters/io_timeout=4294967295
/sys/module/nvme_core/parameters/max_retries=10
- name: "openshift-control-plane"
data: |
[main]
summary=Optimize systems running OpenShift control plane
include=openshift
[sysctl]
# ktune sysctl settings, maximizing i/o throughput
#
# Minimal preemption granularity for CPU-bound tasks:
# (default: 1 msec# (1 + ilog(ncpus)), units: nanoseconds)
kernel.sched_min_granularity_ns=10000000
# The total time the scheduler will consider a migrated process
# "cache hot" and thus less likely to be re-migrated
# (system default is 500000, i.e. 0.5 ms)
kernel.sched_migration_cost_ns=5000000
# SCHED_OTHER wake-up granularity.
#
# Preemption granularity when tasks wake up. Lower the value to
# improve wake-up latency and throughput for latency critical tasks.
kernel.sched_wakeup_granularity_ns=4000000
- name: "openshift-node"
data: |
[main]
summary=Optimize systems running OpenShift nodes
include=openshift
[sysctl]
net.ipv4.tcp_fastopen=3
fs.inotify.max_user_watches=65536
- name: "openshift-control-plane-es"
data: |
[main]
summary=Optimize systems running ES on OpenShift control-plane
include=openshift-control-plane
[sysctl]
vm.max_map_count=262144
- name: "openshift-node-es"
data: |
[main]
summary=Optimize systems running ES on OpenShift nodes
include=openshift-node
[sysctl]
vm.max_map_count=262144
recommend:
- profile: "openshift-control-plane-es"
priority: 10
match:
- label: "tuned.openshift.io/elasticsearch"
type: "pod"
match:
- label: "node-role.kubernetes.io/master"
- label: "node-role.kubernetes.io/infra"

```

```
- profile: "openshift-node-es"
  priority: 20
  match:
  - label: "tuned.openshift.io/elasticsearch"
    type: "pod"

- profile: "openshift-control-plane"
  priority: 30
  match:
  - label: "node-role.kubernetes.io/master"
  - label: "node-role.kubernetes.io/infra"

- profile: "openshift-node"
  priority: 40
```

4.5.5. 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

其中一些插件提供了不受支持的动态性能优化功能。以下 Tuned 插件目前还不支持：

- bootloader
- script
- systemd

如需更多信息，请参阅[可用的 Tuned 插件](#)和[Tuned 入门](#)。

4.6. 了解节点重新引导

若要在重新引导节点时不造成平台上应用程序中断运行，务必要先撤离（evacuate）相关的 pod。对于由路由层提供高可用性的 pod，不需要执行其他操作。对于需要存储的其他 pod（通常是数据库），务必要确保它们能够在 pod 临时下线时仍然保持运作。尽管为有状态 pod 实现可持续运行的方法会视不同应用程序而异，但在所有情形中，都要将调度程序配置为使用节点反关联性，从而确保 pod 在可用节点之间合理分布。

另一个挑战是如何处理运行关键基础架构的节点，比如路由器或 registry。相同的节点撤离过程同样适用于这类节点，但必须要了解某些边缘情况。

4.6.1. 了解基础架构节点重新引导

基础架构节点是标记为运行 OpenShift Container Platform 环境组成部分的节点。目前，管理节点重新引导的最简单方法是确保至少有三个节点可用于运行基础架构。运行基础结构的节点称为 **master** 节点。

以下情景演示了一个常见的错误，只有两个节点可用时，它可能会导致 OpenShift Container Platform 上运行的应用程序发生服务中断。

- 节点 A 标记为不可调度，所有 pod 都被撤离。
- 该节点上运行的 registry pod 现在重新部署到节点 B 上。也就是说，节点 B 现在同时运行两个 registry pod。
- 节点 B 现在标记为不可调度，并且被撤离。
- 在节点 B 上公开两个 pod 端点的服务在短时间内失去所有端点，直到它们被重新部署到节点 A。

如果使用三个 master 节点用于基础架构，则在出现同样的情况时不会导致服务中断。但由于 pod 调度的原因，最后一个节点在撤离并返回到轮转后，会保留为运行零个 registry。其他两个节点将分别运行两个 registry 和一个 registry。最好的解决方法是借助 pod 反关联性。

4.6.2. 使用 pod 反关联性重新引导节点

Pod 反关联性和节点反关联性稍有不同。如果没有其他适当的位置来部署 pod，则可以违反节点反关联性。Pod 反关联性可以设置为必要的或偏好的。

进行了这样的设置后，如果只有两个基础架构节点可用，并且一个节点发生重新引导，容器镜像 registry pod 会被禁止在另一节点上运行。**oc get pods** 会报告 pod 未就绪，直到有合适的节点可用为止。一旦某个节点可用，并且所有 pod 恢复到就绪状态，下一个节点就可以重启。

流程

使用 pod 反关联性重新引导节点：

1. 编辑节点规格以配置 pod 反关联性：

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-antiaffinity
spec:
  affinity:
    podAntiAffinity: 1
```

```

preferredDuringSchedulingIgnoredDuringExecution: 2
- weight: 100 3
podAffinityTerm:
  labelSelector:
    matchExpressions:
      - key: registry 4
        operator: In 5
        values:
          - default
    topologyKey: kubernetes.io/hostname

```

- 1 用于配置 pod 反关联性的小节。
- 2 定义偏好规则。
- 3 为偏好规则指定权重。优先选择权重最高的节点。
- 4 描述用来决定何时应用反关联性规则的 pod 标签。指定标签的键和值。
- 5 运算符表示现有 pod 上的标签和新 pod 规格中 **matchExpression** 参数的值集合之间的关系。可以是 **In**、**NotIn**、**Exists** 或 **DoesNotExist**。

本例假定容器镜像 registry pod 具有标签 **registry=default**。Pod 反关联性可以使用任何 Kubernetes 匹配表达式。

2. 在调度策略文件中启用 **MatchInterPodAffinity** 调度程序 predicate。

4.6.3. 了解如何重新引导运行路由器的节点

在大多数情况下，运行 OpenShift Container Platform 路由器的 pod 会公开一个主机端口。

PodFitsPorts 调度程序 predicate 确保没有使用相同端口的其他路由器 pod 在同一节点上运行，并且达成 pod 反关联性。如果路由器依赖 IP 故障转移来实现高可用性，则不需要任何其他操作。

如果路由器 pod 依赖 AWS Elastic Load Balancing 等外部服务来实现高可用性，则由该服务负责响应路由器 pod 重启。

在极少见的情形中，路由器 pod 可能没有配置主机端口。这时，务必要按照推荐的基础架构节点重启流程来进行操作。

4.7. 使用垃圾回收释放节点资源

作为管理员，您可以通过垃圾回收来释放资源，从而使用 OpenShift Container Platform 确保节点高效运行。

OpenShift Container Platform 节点执行两种类型的垃圾回收：

- 容器垃圾回收：移除已终止的容器。
- 镜像垃圾回收：移除没有被任何正在运行的 pod 引用的镜像。

4.7.1. 了解如何通过垃圾回收移除已终止的容器

可使用驱除阈值来执行容器垃圾回收。

为垃圾回收设定了驱除阈值时，节点会尝试为任何可从 API 访问的 pod 保留容器。如果 pod 已被删除，则容器也会被删除。只要 pod 没有被删除且没有达到驱除阈值，容器就会保留。如果节点遭遇磁盘压力，它会移除容器，并且无法再通过 **oc logs** 访问其日志。

- **eviction-soft** - 软驱除阈值将驱除阈值与一个由管理员指定的必要宽限期配对。
- **removal-hard** - 硬驱除阈值没有宽限期，如果观察到，OpenShift Container Platform 就会立即采取行动。

如果节点在软驱除阈值上下浮动，但没有超过其关联的宽限期，则对应的节点将持续在 **true** 和 **false** 之间振荡。因此，调度程序可能会做出不当的调度决策。

要防止这种情况的出现，请使用 **remove-pressure-transition-period** 标记来控制 OpenShift Container Platform 在摆脱压力状况前必须等待的时间。OpenShift Container Platform 不会设置在状况切换回 false 前，在指定期限内针对指定压力状况满足的驱除阈值。

4.7.2. 了解如何通过垃圾回收移除镜像

镜像垃圾回收依靠节点上 **cAdvisor** 报告的磁盘用量来决定从节点中移除哪些镜像。

镜像垃圾收集策略基于两个条件：

- 触发镜像垃圾回收的磁盘用量百分比（以整数表示）。默认值为 **85**。
- 镜像垃圾回收试尝试释放的磁盘用量百分比（以整数表示）。默认值为 **80**。

对于镜像垃圾回收，您可以使用自定义资源修改以下任意变量。

表 4.2. 用于配置镜像垃圾回收的变量

设置	描述
imageMinimumGCAge	在通过垃圾收集移除镜像前，未用镜像的最小年龄。默认值为 2m 。
imageGCHighThresholdPercent	触发镜像垃圾回收的磁盘用量百分比，以整数表示。默认值为 85 。
imageGCLowThresholdPercent	镜像垃圾回收试尝试释放的磁盘用量百分比，以整数表示。默认值为 80 。

每次运行垃圾收集器都会检索两个镜像列表：

1. 目前在至少一个 pod 中运行的镜像的列表。
2. 主机上可用镜像的列表。

随着新容器运行，新镜像即会出现。所有镜像都标有时间戳。如果镜像正在运行（上方第一个列表）或者刚被检测到（上方第二个列表），它将标上当前的时间。其余镜像的标记来自于以前的运行。然后，所有镜像都根据时间戳进行排序。

一旦开始回收，首先删除最旧的镜像，直到满足停止条件。

4.7.3. 为容器和镜像配置垃圾回收

作为管理员，您可以通过为各个 Machine Config Pool 创建 **kubeletConfig** 对象来配置 OpenShift Container Platform 执行垃圾回收的方式。



注意

OpenShift Container Platform 仅支持每个 Machine Config Pool 具有一个 **kubeletConfig** 对象。

您可以配置以下几项的任意组合：

- 容器软驱除
- 容器硬驱除
- 镜像驱除

对于容器软驱除，您还可以配置驱除前宽限期。

先决条件

1. 为您要配置节点类型获取与静态机器配置池 CRD 关联的标签。执行以下步骤之一：
 - a. 查看 Machine Config Pool：

```
$ oc describe machineconfigpool <name>
```

例如：

```
$ oc describe machineconfigpool worker
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods 1
```

- 1 如果添加了标签，它会出现在 **labels** 下。

- b. 如果标签不存在，则添加一个键/值对：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

容器垃圾回收 CR 的配置示例：

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

```

name: worker-kubeconfig ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods ❷
  kubeletConfig:
    evictionSoft: ❸
      memory.available: "500Mi" ❹
      nodefs.available: "10%"
      nodefs.inodesFree: "5%"
      imagefs.available: "15%"
      imagefs.inodesFree: "10%"
    evictionSoftGracePeriod: ❺
      memory.available: "1m30s"
      nodefs.available: "1m30s"
      nodefs.inodesFree: "1m30s"
      imagefs.available: "1m30s"
      imagefs.inodesFree: "1m30s"
    evictionHard:
      memory.available: "200Mi"
      nodefs.available: "5%"
      nodefs.inodesFree: "4%"
      imagefs.available: "10%"
      imagefs.inodesFree: "5%"
    evictionPressureTransitionPeriod: 0s ❻
    imageMinimumGCAGE: 5m ❼
    imageGCHighThresholdPercent: 80 ❽
    imageGCLowThresholdPercent: 75 ❾

```

- ❶ 对象的名称。
- ❷ 选择器标签。
- ❸ 驱除类型：**EvictionSoft** 和 **EvictionHard**。
- ❹ 基于特定驱除触发信号的驱除阈值。
- ❺ 软驱除宽限期。此参数不适用于 **eviction-hard**。
- ❻ 摆脱驱除压力状况前的等待时间
- ❼ 在通过垃圾收集移除镜像前，未用镜像的最小年龄。
- ❽ 触发镜像垃圾回收的磁盘用量百分比（以整数表示）。
- ❾ 镜像垃圾回收试尝试释放的磁盘用量百分比（以整数表示）。

2. 创建对象：

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

例如：

```
oc create -f gc-container.yaml
```

```
kubeletconfig.machineconfiguration.openshift.io/gc-container created
```

- 验证垃圾回收是否活跃。您在自定义资源中指定的 Machine Config Pool 会将 **UPDATING** 显示为“true”，直到更改完全实施为止：

```
$ oc get machineconfigpool
```

```
NAME    CONFIG                                UPDATED  UPDATING
master  rendered-master-546383f80705bd5aeaba93  True     False
worker  rendered-worker-b4c51bb33ccaae6fc4a6a5  False    True
```

4.8. 为 OPENSIFT CONTAINER PLATFORM 集群中的节点分配资源

为提供更加可靠的调度并且最大程度减少资源过量使用，每个节点都可保留一部分资源供主机上的所有底层节点组件（如 kubelet 和 kube-proxy）和其余系统组件（如 sshd 和 NetworkManager）使用。指定之后，调度程序就能获得更多关于节点为 pod 分配的资源（如内存和 CPU）的信息。

4.8.1. 了解如何为节点分配资源

OpenShift Container Platform 中为节点组件保留的 CPU 和内存资源基于两个节点设置：

设置	描述
kube-reserved	为节点组件保留的资源。默认为 none。
system-reserved	为其余系统组件保留的资源。默认为 none。

如果未设置标记，则默认为 **0**。如果未设置任何标记，则分配的资源设置为引入可分配资源前的节点容量。

4.8.1.1. OpenShift Container Platform 如何计算分配的资源

分配的资源数量根据以下公式来计算：

$$[\text{Allocatable}] = [\text{Node Capacity}] - [\text{kube-reserved}] - [\text{system-reserved}] - [\text{Hard-Eviction-Thresholds}]$$


注意

其中，可分配量中扣除了 **Hard-Eviction-Thresholds** 是一个新的改变，旨在提高系统可靠性。现在，对最终用户 pod 的可分配量在节点一级被强制实施。用户可使用 **experimental-allocatable-ignore-eviction** 设置来保留传统的行为，但这个设置会在未来发行版中弃用。

如果 **[Allocatable]** 为负，它会被设为 **0**。

每个节点报告由容器运行时和 kubelet 使用的系统资源。为更方便地配置 **--system-reserved** 和 **--kube-reserved**，您可以使用节点概述 API 内省相应节点的资源使用情况，该 API 可通过 **/api/v1/nodes/<node>/proxy/stats/summary** 访问。

4.8.1.2. 节点如何强制实施资源限制

节点可以根据配置的可分配值限制 pod 可消耗的资源总量。此功能可以防止 pod 造成系统服务（如容器运行时和节点代理等等）缺少资源，因而能显著提高节点可靠性。强烈建议管理员根据所需的节点使用率目标保留资源，以便提高节点可靠性。

节点使用一个新的 **cgroup** 分级结构来强制实施对资源的约束。它可以强制实现对服务质量的要求。所有 pod 都在专用的 cgroup 层次结构中启动，与系统守护进程隔离。

也可以选择 `enforcing-node-allocatable` 标记中指定相应令牌来强制节点实施 `kube-reserved` 和 `system-reserved`。如果指定，则需要提供对应的 `--kube-reserved-cgroup` 或 `--system-reserved-cgroup`。在未来的发行版中，节点和容器运行时将会被打包在一个与 `system.slice` 分开的通用 cgroup 中。在此之前，建议用户不要更改 `enforcing-node-allocatable` 标记的默认值。

管理员应该如对待有保障 pod 一般对待系统守护进程。系统守护进程可能会在其限定控制组中爆发，此行为需要作为集群部署的一个部分进行管理。强制实施 `system-reserved` 限制可导致节点上关键系统服务缺少 CPU 资源或面临 OOM 终止。我们的建议是，只在操作员为其节点进行了详尽性能测试，确定了精确的估计值并且有信心能够在该组中任何进程发生 OOM 终止时都能复原，才可强制实施 `system-reserved`。

因此，强烈建议用户默认只为 **pod** 强制实施节点可分配量，并为系统守护进程设置适当的保留量，以保持整个节点的可靠性。

4.8.1.3. 了解驱除阈值

如果某个节点面临内存压力，这可能会影响整个节点以及该节点上运行的所有 pod。如果系统守护进程使用的内存超过为其保留的内存量，可能会发生 OOM 事件，影响整个节点以及该节点上运行的所有 pod。为避免系统 OOM（或降低其发生概率），节点会提供资源不足处理。

您可以使用 `--eviction-hard` 标记保留一些内存。每当节点上的内存可用量低于该绝对值或百分比时，节点会尝试驱除 pod。如果节点上没有系统守护进程，pod 的内存会被限制在 `capacity - eviction-hard` 内。因此，pod 不能使用作为达到内存不足状态前驱除缓冲量而预留的资源。

下例演示了节点内存可分配量的影响：

- 节点容量为 **32Gi**
- `--kube-reserved` 为 **2Gi**
- `--system-reserved` 为 **1Gi**
- `--eviction-hard` 设置为 **100Mi**。

对于这个节点，有效节点可分配量的值是 **28.9Gi**。如果节点和系统组件用尽其所有保留量，则 pod 的可内存为 **28.9Gi**，并且 kubelet 会在超过此用量时驱除 pod。

如果您通过顶级 cgroup 强制实施节点可分配量 (**28.9Gi**)，那么 pod 永不会超过 **28.9Gi**。除非系统守护进程消耗的内存超过 **3.1Gi**，否则不会执行驱除。

如果系统守护进程没有用尽其所有保留量，那么在上例中，pod 会在节点开始驱除前面临被其限定 cgroup 执行 `memcg` OOM 终止的问题。为了在这种情况下更好地强制实施 QoS，节点会对所有 pod 的顶级 cgroup 应用硬驱除阈值，即 **Node Allocatable + Eviction Hard Thresholds**。

如果系统守护进程没有用尽所有保留量，每当 pod 消耗的内存超过 **28.9Gi** 时，节点就会驱除 pod。如果不及驱逐，消耗的内存超过 **29Gi** 时就会对 pod 执行 OOM 终止。

4.8.1.4. 调度程序如何确定资源可用性

调度程序使用 `node.Status.Allocatable`（而非 `node.Status.Capacity`）的值来决定节点是否成为 pod 调度的候选者。

在默认情况下，节点会将其机器容量报告为可完全被集群调度。

4.8.2. 为节点配置分配的资源

OpenShift Container Platform 支持对 CPU 和内存资源类型执行分配。如果管理员启用了临时存储技术预览功能，则也支持 `ephemeral-resource` 资源类型。对于 `cpu` 类型，资源数量以内核数为单位来指定，例如 `200m`、`0.5` 或 `1`。对于 `memory` 和 `ephemeral-storage`，则以字节数为单位来指定，例如 `200Ki`、`50Mi` 或 `5Gi`。

作为管理员，您可以通过一组 `<resource_type>=<resource_quantity>` 对（如 `cpu=200m,memory=512Mi`）来利用自定义资源 (CR) 进行设置。

先决条件

- 若要帮助自己确定 `--system-reserved` 和 `--kube-reserved` 的设置，您可以使用节点概述 API 内省相应节点的资源使用情况，该 API 可通过 `/api/v1/nodes/<node>/proxy/stats/summary` 访问。对节点运行以下命令：

```
$ oc get --raw /api/v1/nodes/<node>/proxy/stats/summary
```

例如，若要访问 `cluster.node22` 节点的资源，您可以运行：

```
$ oc get --raw /api/v1/nodes/cluster.node22/proxy/stats/summary
{
  "node": {
    "nodeName": "cluster.node22",
    "systemContainers": [
      {
        "cpu": {
          "usageCoreNanoSeconds": 929684480915,
          "usageNanoCores": 190998084
        },
        "memory": {
          "rssBytes": 176726016,
          "usageBytes": 1397895168,
          "workingSetBytes": 1050509312
        },
        "name": "kubelet"
      },
      {
        "cpu": {
          "usageCoreNanoSeconds": 128521955903,
          "usageNanoCores": 5928600
        },
        "memory": {
          "rssBytes": 35958784,
          "usageBytes": 129671168,
          "workingSetBytes": 102416384
        },
        "name": "runtime"
      }
    ]
  }
}
```

```

    }
  ]
}
}

```

2. 为您要配置的节点类型获取与静态机器配置池 CRD 关联的标签。执行以下步骤之一：

a. 查看 Machine Config Pool：

```
$ oc describe machineconfigpool <name>
```

例如：

```
$ oc describe machineconfigpool worker

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods ❶
```

❶ 如果添加了标签，它会出现在 **labels** 下。

b. 如果标签不存在，则添加一个键/值对：

```
$ oc label machineconfigpool worker custom-kubelet=small-pods
```

流程

1. 为配置更改创建自定义资源 (CR)。

资源分配 CR 的示例配置

```

apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-allocatable ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods ❷
  kubeletConfig:
    systemReserved:
      cpu: 500m
      memory: 512Mi
    kubeReserved:
      cpu: 500m
      memory: 512Mi

```

4.9. 查看节点审计日志

审计提供一组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。

4.9.1. 关于 API 审计日志

审计在 API 服务器级别运作，记录所有传入到服务器的请求。每个审计日志包含以下信息：

表 4.3. 审计日志字段

字段	描述
level	生成事件的审计级别。
auditID	为每个请求生成的唯一审计 ID。
stage	生成此事件实例时请求处理的阶段。
requestURI	客户端向服务器发送的请求 URI。
verb	与请求相关联的 Kubernetes 操作动词。对于非资源请求，这是小写 HTTP 方法。
user	经过身份验证的用户信息。
impersonatedUser	可选。如果请求模拟了另一个用户，则为被模拟的用户信息。
sourceIPs	可选。源 IP，请求发起的源和任何中间代理。
userAgent	可选。客户端报告的用户代理字符串。请注意，用户代理由客户端提供，且必须不可信任。
objectRef	可选。这个请求的目标对象引用。这不适用于 List 类型请求，或者非资源请求。
responseStatus	可选。响应的状态，即使 responseObject 不是 Status 类型也会生成。对于成功的响应，这只会包括代码。对于非状态类型错误响应，这将自动生成出错信息。
requestObject	可选。请求中的 API 对象，采用 JSON 格式。在进行 version conversion、defaulting、admission 或 merging 之前，在请求中的 RequestObject 记录（可能会被转换为 JSON 格式）。这是一个外部版本化的对象类型，可能自身并不是一个有效的对象。对于非资源请求，这会被忽略，且只在 Request 级别或更高级别中被记录。
responseObject	可选。响应中返回的 API 对象，使用 JSON 格式。在转换为外部类型后， responseObject 被记录，并被序列化为 JSON 数据。在非资源请求中会省略它，且仅在 Response 级别中记录。
requestReceivedTimestamp	请求到达 API 服务器的时间。

字段	描述
stageTimestamp	请求到达当前审计阶段的时间。
annotations	可选。一个无结构的键值映射，它存储在一个审计事件中，可以通过在请求服务链中调用的插件来设置它，包括认证、授权和准入插件。请注意，这些注解用于审计事件，且与所提交对象的 metadata.annotations 没有关联。标识信息组件的键应该是唯一的以避免名称冲突，例如 podsecuritypolicy.admission.k8s.io/policy 。值应该较短。注解包含在 Metadata 级别中。

Kubernetes API 服务器的输出示例：

```
{
  "kind": "Event",
  "apiVersion": "audit.k8s.io/v1",
  "level": "Metadata",
  "auditID": "ad209ce1-fec7-4130-8192-c4cc63f1d8cd",
  "stage": "ResponseComplete",
  "requestURI": "/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s",
  "verb": "update",
  "user": {
    "username": "system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client",
    "uid": "dd4997e3-d565-4e37-80f8-7fc122ccd785",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:openshift-kube-controller-manager",
      "system:authenticated"
    ],
    "sourceIPs": [":1"],
    "userAgent": "cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64) kubernetes/$Format",
    "objectRef": {
      "resource": "configmaps",
      "namespace": "openshift-kube-controller-manager",
      "name": "cert-recovery-controller-lock",
      "uid": "5c57190b-6993-425d-8101-8337e48c7548",
      "apiVersion": "v1",
      "resourceVersion": "574307"
    },
    "responseStatus": {
      "metadata": {},
      "code": 200,
      "requestReceivedTimestamp": "2020-04-02T08:27:20.200962Z",
      "stageTimestamp": "2020-04-02T08:27:20.206710Z",
      "annotations": {
        "authorization.k8s.io/decision": "allow",
        "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding 'system:openshift:operator:kube-controller-manager-recovery' of ClusterRole 'cluster-admin' to ServiceAccount 'localhost-recovery-client/openshift-kube-controller-manager'"
      }
    }
  }
}
```

4.9.2. 查看审计日志

您可以查看 OpenShift Container Platform API 服务器或每个 master 节点的 Kubernetes API 服务器的日志。

流程

查看审计日志：

1. 查看 OpenShift Container Platform API 服务器日志
 - a. 如果需要，获取您要查看日志的节点名称：

```
$ oc adm node-logs --role=master --path=openshift-apiserver/
ip-10-0-140-97.ec2.internal audit-2019-04-09T00-12-19.834.log
ip-10-0-140-97.ec2.internal audit-2019-04-09T11-13-00.469.log
ip-10-0-140-97.ec2.internal audit.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T00-11-49.835.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T11-08-30.469.log
ip-10-0-153-35.ec2.internal audit.log
```

```
ip-10-0-170-165.ec2.internal audit-2019-04-09T00-13-00.128.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T11-10-04.082.log
ip-10-0-170-165.ec2.internal audit.log
```

- b. 查看特定 master 节点和时间戳的 OpenShift Container Platform API 服务器日志，或者查看该 master 的所有日志：

```
$ oc adm node-logs <node-name> --path=openshift-apiserver/<log-name>
```

例如：

```
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit-2019-04-08T13-09-01.227.log
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=openshift-apiserver/audit.log
```

输出内容类似以下示例：

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"ad209ce1-fec7-4130-8192-c4cc63f1d8cd","stage":"ResponseComplete","requestURI":"/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s","verb":"update","user":{"username":"system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client","uid":"dd4997e3-d565-4e37-80f8-7fc122ccd785","groups":["system:serviceaccounts","system:serviceaccounts:openshift-kube-controller-manager","system:authenticated"],"sourceIPs":["::1"],"userAgent":"cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64) kubernetes/$Format","objectRef":{"resource":"configmaps","namespace":"openshift-kube-controller-manager","name":"cert-recovery-controller-lock","uid":"5c57190b-6993-425d-8101-8337e48c7548","apiVersion":"v1","resourceVersion":"574307"},"responseStatus":{"metadata":{"code":200},"requestReceivedTimestamp":"2020-04-02T08:27:20.200962Z","stageTimestamp":"2020-04-02T08:27:20.206710Z"},"annotations":{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"system:openshift:operator:kube-controller-manager-recovery\" of ClusterRole \"cluster-admin\" to ServiceAccount \"localhost-recovery-client/openshift-kube-controller-manager\"}}
```

2. 查看 Kubernetes API 服务器日志：

- a. 如果需要，获取您要查看日志的节点名称：

```
$ oc adm node-logs --role=master --path=kube-apiserver/

ip-10-0-140-97.ec2.internal audit-2019-04-09T14-07-27.129.log
ip-10-0-140-97.ec2.internal audit-2019-04-09T19-18-32.542.log
ip-10-0-140-97.ec2.internal audit.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T19-24-22.620.log
ip-10-0-153-35.ec2.internal audit-2019-04-09T19-51-30.905.log
ip-10-0-153-35.ec2.internal audit.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T18-37-07.511.log
ip-10-0-170-165.ec2.internal audit-2019-04-09T19-21-14.371.log
ip-10-0-170-165.ec2.internal audit.log
```

- b. 查看特定 master 节点和时间戳的 Kubernetes API 服务器日志，或者查看该 master 的所有日志：

```
$ oc adm node-logs <node-name> --path=kube-apiserver/<log-name>
```

例如：

```
$ oc adm node-logs ip-10-0-140-97.ec2.internal --path=kube-apiserver/audit-2019-04-09T14-07-27.129.log
$ oc adm node-logs ip-10-0-170-165.ec2.internal --path=kube-apiserver/audit.log
```

输出内容类似以下示例：

```
{"kind":"Event","apiVersion":"audit.k8s.io/v1","level":"Metadata","auditID":"ad209ce1-fec7-4130-8192-c4cc63f1d8cd","stage":"ResponseComplete","requestURI":"/api/v1/namespaces/openshift-kube-controller-manager/configmaps/cert-recovery-controller-lock?timeout=35s","verb":"update","user":{"username":"system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client","uid":"dd4997e3-d565-4e37-80f8-7fc122ccd785","groups":["system:serviceaccounts","system:serviceaccounts:openshift-kube-controller-manager","system:authenticated"]},"sourceIPs":["::1"],"userAgent":"cluster-kube-controller-manager-operator/v0.0.0 (linux/amd64) kubernetes/$Format","objectRef":{"resource":"configmaps","namespace":"openshift-kube-controller-manager","name":"cert-recovery-controller-lock","uid":"5c57190b-6993-425d-8101-8337e48c7548","apiVersion":"v1","resourceVersion":"574307"},"responseStatus":{"metadata":{},"code":200},"requestReceivedTimestamp":"2020-04-02T08:27:20.200962Z","stageTimestamp":"2020-04-02T08:27:20.206710Z","annotations":{"authorization.k8s.io/decision":"allow","authorization.k8s.io/reason":"RBAC: allowed by ClusterRoleBinding \"system:openshift:operator:kube-controller-manager-recovery\" of ClusterRole \"cluster-admin\" to ServiceAccount \"localhost-recovery-client/openshift-kube-controller-manager\"}}
```

4.10. 机器配置守护进程指标

Machine Config Daemon 是 Machine Config Operator 的一部分。它可在集群的每个节点中运行。Machine Config Daemon 的目的是管理每个节点上的配置更改和更新。

4.10.1. 机器配置守护进程指标

从 OpenShift Container Platform 4.3 开始，Machine Config Daemon 提供了一组指标。这些指标可以使用 Prometheus Cluster Monitoring 来访问。

下表介绍了这些指标。



注意

在 **Name** 和 **Description** 栏中带有 * 标记的指标数据代表了可能会造成性能问题的严重错误。这些问题可能会阻止更新和升级操作。



注意

虽然有些条目包含获取特定日志的命令，但最完整的日志数据可以通过 **oc adm must-gather** 命令获得。

表 4.4. MCO 指标

名称	格式	描述	备注
mcd_host_os_and_version	[[]string{"os", "version"}]	显示运行 MCD 的操作系统，如 RHCOS 或 RHEL。如果是 RHCOS，则会提供版本信息。	
ssh_accessed	计数	显示在节点中成功进行 SSH 验证的次数。	非零值显示可能已经有人手动更改了节点。由于磁盘中的状态和机器配置中定义的状态的不同，这种更改可能会导致不可协调的错误。
mcd_drain*	{"drain_time", "err"}	在排空失败时出现的错误。*	<p>虽然排空可能需要多次尝试方可成功，但最终失败的排空会操作会阻止更新进行。drain_time 指标显示排空操作所用的时间，这可帮助进行故障排除。</p> <p>如需进一步调查，请运行以下命令查看日志：</p> <pre>\$ oc logs -f -n openshift-machine-config-operator machine-config-daemon- <hash> -c machine-config-daemon</pre>
mcd_pivot_err*	[[]string{"pivot_target", "err"}]	pivot 过程中遇到的日志错误。*	<p>pivot 错误可能会导致 OS 升级无法进行。</p> <p>要进行进一步调查，请运行这个命令访问该节点并查看其所有日志：</p> <pre>\$ oc debug node/<node> — chroot /host journalctl -u pivot.service</pre> <p>或者，可以运行这个命令只查看来自 machine-config-daemon 容器中的日志：</p> <pre>\$ oc logs -f -n openshift-machine-config-operator machine-config-daemon- <hash> -c machine-config-daemon</pre>

名称	格式	描述	备注
mcd_state	<code>[]string{"state", "reason"}</code>	指定节点的 Machine Config Daemon 状态。可能的状态是 "Done"、"Working" 和 "Degraded"。如果是 "Degraded"，则会包括原因。	如需进一步调查，请运行以下命令查看日志： \$ oc logs -f -n openshift-machine-config-operator machine-config-daemon- <hash> -c machine-config-daemon
mcd_kubelet_state*	<code>[]string{"err"}</code>	日志 kubelet 健康失败。*	这应该为空，故障计数为 0。如果失败数超过 2，则代表超过了阈值。这表示 kubelet 健康可能存在问题。 要进行进一步调查，请运行这个命令访问该节点并查看其所有日志： \$ oc debug node/<node> — chroot /host journalctl -u kubelet
mcd_reboot_err*	<code>[]string{"message", "err"}</code>	重启失败以及相应错误的日志。*	这应该为空，代表重启成功。 如需进一步调查，请运行以下命令查看日志： \$ oc logs -f -n openshift-machine-config-operator machine-config-daemon- <hash> -c machine-config-daemon
mcd_update_state	<code>[]string{"config", "err"}</code>	记录配置更新的成功或失败以及相应的错误。	预期的值为 rendered-master/rendered-worker-XXXX 。如果更新失败，则会出现错误。 如需进一步调查，请运行以下命令查看日志： \$ oc logs -f -n openshift-machine-config-operator machine-config-daemon- <hash> -c machine-config-daemon

其他资源

- 请参阅 [Prometheus Cluster Monitoring 堆栈的文档](#)。

- 请参阅 [有关收集集群数据的文档](#)。

第 5 章 操作容器

5.1. 了解容器

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

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

多年来，Linux 内核一直在整合容器技术的能力。OpenShift Container Platform 和 Kubernetes 增加了在多主机安装之间编排容器的功能。

5.2. 在部署 POD 前使用初始容器来执行任务

OpenShift Container Platform 提供了一组 *初始容器 (Init Containers)*，它们是在应用程序容器之前运行的专用容器，可以包含不出现在应用程序镜像中的实用程序或设置脚本。

5.2.1. 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](#)。

5.2.2. 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**。两个容器都完成后，该 Pod 就会启动。

流程

1. 为初始容器创建 YAML 文件：

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

2. 为 **myservice** 服务创建 YAML 文件。

```
kind: Service
apiVersion: v1
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

3. 为 **mydb** 服务创建 YAML 文件。

```
kind: Service
apiVersion: v1
metadata:
  name: mydb
spec:
  ports:
```

```
- protocol: TCP
  port: 80
  targetPort: 9377
```

4. 运行以下命令来创建 **myapp-pod** :

```
$ oc create -f myapp.yaml

pod/myapp-pod created
```

5. 查看 pod 的状态 :

```
$ oc get pods
NAME                READY   STATUS    RESTARTS   AGE
myapp-pod           0/1     Init:0/2   0           5s
```

注意 pod 状态指出它正在等待

6. 运行以下命令来创建上述服务 :

```
$ oc create -f mydb.yaml
$ oc create -f myservice.yaml
```

7. 查看 pod 的状态 :

```
$ oc get pods
NAME                READY   STATUS    RESTARTS   AGE
myapp-pod           1/1     Running   0           2m
```

5.3. 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。

5.3.1. 了解卷

卷是挂载的文件系统，供 pod 及其容器使用，可以通过多个主机上本地或网络附加存储端点来支持。默认情况下，容器不具持久性；重启之后，其中的内容会被清除。

为确保卷上的文件系统不包含任何错误，并在出现错误时尽可能进行修复，OpenShift Container Platform 在调用 **mount** 实用程序之前会先调用 **fsck**。在添加卷或更新现有卷时会出现这种情况。

最简单的卷类型是 **emptyDir**，这是单一机器上的一个临时目录。管理员也可以允许您请求自动附加到 pod 的持久性卷。



注意

如果集群管理员启用了 **FSGroup** 参数，则 **emptyDir** 卷存储可能会受到基于 pod **FSGroup** 的配额的限制。

5.3.2. 使用 OpenShift Container Platform CLI 操作卷

您可以使用 CLI 命令 **oc set volume**，为任何使用 pod 模板的对象（如复制控制器或 DeploymentConfig）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** 指定以下值之一：

表 5.1. 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** 指定 **--add**、**--remove** 或 **--list**。

必要参数

所有 **<mandatory_parameters>** 都特定于所选操作，并在后续小节中阐述。

选项

所有 **<options>** 都特定于所选操作，并在后续小节阐述。

5.3.3. 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> --list [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 --list
```

- 列出在所有 DeploymentConfig 中定义的卷 v1：

```
$ oc set volume dc --all --name=v1
```

5.3.4. 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 5.2. 添加卷时支持的选项

选项	描述	默认
--name	卷的名称。	若未指定，则自动生成。
-t, --type	卷源的名称。支持的值有 emptyDir 、 hostPath 、 secret 、 configmap 、 persistentVolumeClaim 或 projected 。	emptyDir
-c, --containers	按名称选择容器。它还可以使用通配符 '*' 来匹配任意字符。	'*'
-m, --mount-path	所选容器内的挂载路径。	
--path	主机路径。 --type=hostPath 的必要参数。	
--secret-name	secret 的名称。 --type=secret 的必要参数。	

选项	描述	默认
--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 DeploymentConfig **注册表** 中：

```
$ oc set volume dc/registry --add
```

- 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
```

- 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
```

- 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\"  
}}"
```

5.3.5. 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
```

- 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
```

5.3.6. 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 5.3. 移除卷时支持的选项

选项	描述	默认
<code>--name</code>	卷的名称。	
<code>-c, --containers</code>	按名称选择容器。它还可以使用通配符 <code>**</code> 来匹配任意字符。	<code>**</code>
<code>--confirm</code>	指定您想要一次性移除多个卷。	
<code>-o, --output</code>	显示修改后的对象，而不在服务器上更新它们。支持的值有 <code>json</code> 和 <code>yaml</code> 。	
<code>--output-version</code>	输出给定版本的修改后对象。	<code>api-version</code>

例如：

- 从 DeploymentConfig `d1` 中移除卷 `v1`：

```
$ oc set volume dc/d1 --remove --name=v1
```

- 从 DeploymentConfig **d1** 的容器 **c1** 卸载卷 **v1**，并在卷 **v1** 未被 **d1** 上的任何容器引用时将它移除：

```
$ 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
```

5.3.7. 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。

流程

1. 运行 **oc rsh** 命令来查看卷中的文件列表：

```
$ oc rsh <pod>
sh-4.3$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

2. 指定 **subPath**：

subPath 用法示例

```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
  containers:
    - name: mysql
      image: mysql
      volumeMounts:
        - mountPath: /var/lib/mysql
          name: site-data
          subPath: mysql 1
    - name: php
      image: php
      volumeMounts:
        - mountPath: /var/www/html
          name: site-data
          subPath: html 2
  volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

1 1 数据库存储在 **mysql** 文件夹中。

2 2 HTML 内容存储在 **html** 文件夹中。

5.4. 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。

5.4.1. 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、configmap 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、configmap 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。

以下一般情景演示了如何使用投射卷。

ConfigMap、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。

ConfigMap + 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 configmap，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）。这个示例允许操作员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。

5.4.1.1. Pod 规格示例

以下是用于创建投射卷的 pod 规格示例。

带有 secret、downward API 和 configmap 的 Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
```

```

spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts: ❶
    - name: all-in-one
      mountPath: "/projected-volume" ❷
      readOnly: true ❸
  volumes: ❹
  - name: all-in-one ❺
    projected:
      defaultMode: 0400 ❻
      sources:
      - secret:
          name: mysecret ❼
          items:
          - key: username
            path: my-group/my-username ❽
      - downwardAPI: ❾
          items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: container-test
              resource: limits.cpu
      - configMap: ❿
          name: myconfigmap
          items:
          - key: config
            path: my-group/my-config
            mode: 0777 ⓫

```

- ❶ 为每个需要 secret 的容器添加 **volumeMounts** 部分。
- ❷ 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。
- ❸ 将 **readOnly** 设为 **true**。
- ❹ 添加一个 **volumes** 块，以列出每个投射卷源。
- ❺ 为卷指定任意名称。
- ❻ 设置文件的执行权限。
- ❼ 添加 secret。输入 secret 对象的名称。必须列出您要使用的每个 secret。
- ❽ 指定 **mountPath** 下 secret 文件的路径。此处，secret 文件位于 `/projected-volume/my-group/my-config`。
- ❾ 添加 Downward API 源。
- ❿ 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:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  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**。

5.4.1.2. 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:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  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

5.4.2. 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 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。

流程

使用投射卷挂载现有的 Secret 卷源。

1. 输入以下内容并相应地替换密码和用户信息，创建包含这些 secret 的文件：

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  pass: MWYyZDFIMmU2N2Rm
  user: YWRtaW4=
```

user 和 **pass** 值可以是采用 **base64** 编码的任意有效字符串。此处的示例使用了 **base64** 编码值 **user: admin** 和 **pass:1f2d1e2e67df**。

```
$ echo -n "admin" | base64
YWRtaW4=
$ echo -n "1f2d1e2e67df" | base64
MWYyZDFIMmU2N2Rm
```

2. 使用以下命令来创建 **secret** :

```
$ oc create -f <secrets-filename>
```

例如 :

```
$ oc create -f secret.yaml
secret "mysecret" created
```

3. 您可以使用以下命令来检查是否创建了 **secret** :

```
$ oc get secret <secret-name>
$ oc get secret <secret-name> -o yaml
```

例如 :

```
$ oc get secret mysecret
NAME      TYPE      DATA   AGE
mysecret  Opaque    2       17h
```

```
$ oc get secret mysecret -o yaml
```

```
apiVersion: v1
data:
  pass: MWYyZDFIMmU2N2Rm
  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
```

4. 创建类似以下示例的 **pod** 配置文件，使其包含 **volumes** 部分 :

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
    - name: test-projected-volume
```

```

image: busybox
args:
- sleep
- "86400"
volumeMounts:
- name: all-in-one
  mountPath: "/projected-volume"
  readOnly: true
volumes:
- name: all-in-one
  projected:
    sources:
    - secret: ❶
      name: user
    - secret: ❷
      name: pass

```

❶ ❷ 您创建的 secret 的名称。

5. 从配置文件创建 pod :

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

例如 :

```
$ oc create -f secret-pod.yaml
pod "test-projected-volume" created
```

6. 验证 pod 容器是否在运行, 然后留意 Pod 的更改 :

```
$ oc get pod <name>
```

输出结果应该类似以下示例 :

```
$ oc get pod test-projected-volume
NAME                READY   STATUS    RESTARTS   AGE
test-projected-volume 1/1     Running  0           14s
```

7. 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
```

8. 在 shell 中, 验证 **projected-volumes** 目录是否包含您的投射源 :

```
/ # ls
bin          home        root        tmp
dev          proc        run         usr
etc          projected-volume sys         var
```


5.5. 允许容器消耗 API 对象

Downward API 是一种允许容器消耗 API 对象的相关信息且不与 OpenShift Container Platform 耦合的机制。此类信息包括 pod 的名称、命名空间和资源值。容器可以使用环境变量或卷插件来消耗来自 Downward API 的信息。

5.5.1. 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。

5.5.2. 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。根据您的选择的方法，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。

5.5.2.1. 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

使用环境变量

1. 创建 **pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  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
    restartPolicy: Never
```

2. 从 **pod.yaml** 文件创建 pod：

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

3. 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
```

5.5.2.2. 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 **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:
  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
  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
```

2. 从 **volume-pod.yaml** 文件创建 pod：

```
$ oc create -f volume-pod.yaml
```

3. 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
cluster=downward-api-test-cluster1
rack=rack-123
```

```

zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api

```

5.5.3. 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。

5.5.3.1. 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。

流程

使用环境变量：

1. 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：

```

...
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
    ...

```

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。

2. 从 `pod.yaml` 文件创建 pod :

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

5.5.3.2. 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。

流程

使用卷插件：

1. 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：

```
....
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
....
```

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。

2. 从 **volume-pod.yaml** 文件创建 pod :

```
$ oc create -f volume-pod.yaml
```

5.5.4. 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 **secret.yaml** 文件 :

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  password: cGFzc3dvcmQ=
  username: ZGV2ZWxvcGVy
type: kubernetes.io/basic-auth
```

2. 从 secret.yaml 文件创建 **Secret** :

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

3. 创建 **pod.yaml** 文件来引用上述 **Secret** 中的 **username** 字段 :

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  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
    restartPolicy: Never
```

4. 从 **pod.yaml** 文件创建 pod :

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

5. 检查容器日志中的 **MY_SECRET_USERNAME** 值 :

```
$ oc logs -p dapi-env-test-pod
```

5.5.5. 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. 创建 **configmap.yaml** 文件：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

2. 从 **configmap.yaml** 文件创建 **ConfigMap**：

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

3. 创建 **pod.yaml** 文件来引用上述 **ConfigMap**：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  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
      restartPolicy: Always
```

4. 从 **pod.yaml** 文件创建 pod：

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

5. 检查容器日志中的 **MY_CONFIGMAP_VALUE** 值：

```
$ oc logs -p dapi-env-test-pod
```

5.5.6. 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 `pod.yaml` 文件来引用现有的环境变量：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  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)
  restartPolicy: Never
```

2. 从 `pod.yaml` 文件创建 pod：

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

3. 检查容器日志中的 `MY_ENV_VAR_REF_ENV` 值：

```
$ oc logs -p dapi-env-test-pod
```

5.5.7. 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 `pod.yaml` 文件来引用现有的环境变量：

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  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)
  restartPolicy: Never
```


2. 从 **pod.yaml** 文件创建 pod :

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

3. 检查容器日志中的 **MY_NEW_ENV** 值 :

```
$ oc logs -p dapi-env-test-pod
```

5.6. 将文件复制到 OPENSIFT CONTAINER PLATFORM 容器或从中复制

您可以使用 **rsync** 命令，通过 CLI 将本地文件复制到容器中的远程目录，或从中复制文件。

5.6.1. 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>]
```

5.6.1.1. 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**。

5.6.2. 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>
```

例如：

```
$ oc rsync /home/user/source devpod1234:/src
```

```
WARNING: cannot use rsync: rsync not available in container
status.txt
```

- 将 pod 目录复制到本地目录：

```
$ oc rsync devpod1234:/src /home/user/source
```

```
oc rsync devpod1234:/src/status.txt /home/user/
WARNING: cannot use rsync: rsync not available in container
status.txt
```

5.6.3. 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 SRC POD:DEST
```

或者：

```
$ export RSYNC_RSH='oc rsh'
$ rsync --exclude-from=FILE SRC POD:DEST
```

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。

5.7. 在 OPENSIFT CONTAINER PLATFORM 容器中执行远程命令

您可以使用 CLI 在 OpenShift Container Platform 容器中执行远程命令。

5.7.1. 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
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。

5.7.2. 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 `SPDY`。

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。

5.8. 使用端口转发访问容器中的应用程序

OpenShift Container Platform 支持向 pod 转发端口。

5.8.1. 了解端口转发

您可以使用 CLI 将一个或多个本地端口转发到 pod。这样，您可以在本地侦听一个指定或随机端口，并且与 pod 中的指定端口来回转发数据。

CLI 中内置了端口转发支持：

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

CLI 侦听用户指定的本地端口，并通过以下协议进行转发。

可使用以下格式来指定端口：

5000	客户端在本地侦听端口 5000，并转发到 pod 中的 5000。
6000:5000	客户端在本地侦听端口 6000，并转发到 pod 中的 5000。
:5000 或 0:5000	客户端选择本地的一个空闲端口，并转发到 pod 中的 5000。

OpenShift Container Platform 处理来自客户端的端口转发请求。在收到请求后，OpenShift Container Platform 会升级响应并等待客户端创建端口转发流。当 OpenShift Container Platform 收到新流时，它会在流和 pod 端口之间复制数据。

从架构上看，有不同的选项可用于转发到 pod 端口。支持的 OpenShift Container Platform 实施会直接调用节点主机上的 `nsenter` 来进入 pod 的网络命名空间，然后调用 `socat` 在流和 pod 端口之间复制数据。不过，自定义实施中可能会包括运行一个 `helper pod`，然后运行 `nsenter` 和 `socat`，从而不需要在主机上安装这些二进制代码。

5.8.2. 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

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

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

Forwarding from 127.0.0.1:42390 -> 5000
Forwarding from [::1]:42390 -> 5000
```

或者：

```
$ oc port-forward <pod> 0:5000
```

5.8.3. 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的流；当前使用 **SPDY**。

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。

5.9. 在容器中使用 SYSCTL

Sysctl 设置可以通过 Kubernetes 来公开，允许用户在运行时为容器内的命名空间修改某些内核参数。只有拥有命名空间的 sysctl 才能独立于 pod 进行设置。如果 sysctl 没有命名空间，则称为 *节点级别*，无法在 OpenShift Container Platform 中设置。此外，只有被认为是安全的 sysctl 才会默认列在白名单中；您可以在节点上手动启用其他不安全 sysctl 来供用户使用。

5.9.1. 关于 sysctl

在 Linux 中，管理员可通过 sysctl 接口在运行时修改内核参数。参数可通过 `/proc/sys/` 虚拟进程文件系统提供。这些参数涵盖了各种不同的子系统，例如：

- 内核（通用前缀：*kernel.*）
- 网络（通用前缀：*net.*）
- 虚拟内存（通用前缀：*vm.*）
- MDADM（通用前缀：*dev.*）

如需了解更多子系统，请参阅 [Kernel 文档](#)。要获取所有参数的列表，请运行：

```
$ sudo sysctl -a
```

5.9.1.1. 命名空间和节点级 sysctl

许多 sysctl 在 Linux 内核中是有 *命名空间* 的。这意味着您可以针对节点上的每个 pod 单独设置它们。sysctl 必须拥有命名空间，才能在 Kubernetes 内的 pod 上下文中访问它们。

以下 sysctl 已知是拥有命名空间的：

- *kernel.shm**
- *kernel.msg**
- *kernel.sem*
- *fs.mqueue.**

另外，*net.** 组中的大多数 sysctl 都已知是拥有命名空间的。其命名空间的采用根据内核版本和发行方而有所不同。

无命名空间的 sysctl 被视为 *节点级别*，且必须由集群管理员手动设置，或者通过使用节点的底层 Linux 发行版，如修改 `/etc/sysctls.conf` 文件，或者通过使用带有特权容器的 DaemonSet。



注意

可以考虑将带有特殊 sysctl 节点标记为污点。仅将 pod 调度到需要这些 sysctl 设置的节点。使用污点和容限功能来标记节点。

5.9.1.2. 安全与不安全 sysctl

sysctl 划分为 *安全* 和 *不安全* sysctl。

sysctl 若要被视为安全，必须使用正确的命名空间，且必须在同一节点的不同 pod 之间进行适当的隔离。也就是说，如果您为一个 pod 设置了 sysctl，它不得：

- 影响节点上的其他任何 pod
- 危害节点的健康
- 获取超过 pod 资源限制的 CPU 或内存资源

OpenShift Container Platform 支持（或列入白名单）安全集合中的以下 sysctl：

- *kernel.shm_rmid_forced*
- *net.ipv4.ip_local_port_range*
- *net.ipv4.tcp_syncookies*

所有安全 sysctl 都默认启用。您可以通过修改 pod 规格在 pod 中使用 sysctl。

任何未在 OpenShift Container Platform 白名单中列出的 sysctl 都被视为对 OpenShift Container Platform 而言不安全。请注意，仅拥有命名空间还不足以使 sysctl 被视为安全。

所有不安全 sysctl 都默认禁用，集群管理员必须逐个节点手动启用它们。禁用了不安全 sysctl 的 Pod 会被调度，但不会启动。

```
$ oc get pod
```

NAME	READY	STATUS	RESTARTS	AGE
hello-pod	0/1	SysctlForbidden	0	14s

5.9.2. 为 pod 设置 sysctl

您可以使用 pod 的 **securityContext** 在 pod 上设置 sysctl。**securityContext** 适用于同一 pod 中的所有容器。

默认允许安全 sysctl。带有不安全 sysctl 的 pod 无法在任何节点上启动，除非集群管理员在某个节点上明确启用了不安全 sysctl。与节点级 sysctl 一样，使用污点和容限功能或节点上的标签将这些 pod 调度到正确的节点。

以下示例使用 pod **securityContext** 设置一个安全 sysctl **kernel.shm_rmid_forced**，以及两个不安全 sysctl **net.ipv4.route.min_pmtu** 和 **kernel.msgmax**。在规格中，安全和不安全 sysctl 并无区别。



警告

为了避免让操作系统变得不稳定，只有在了解了参数的作用后才修改 sysctl 参数。

流程

使用安全和不安全 sysctl：

1. 修改定义 pod 的 YAML 文件并添加 **securityContext** 规格，如下例所示：

-

```

apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
      - name: kernel.shm_rmid_forced
        value: "0"
      - name: net.ipv4.route.min_pmtu
        value: "552"
      - name: kernel.msgmax
        value: "65536"
  ...

```

2. 创建 pod :

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

如果节点不允许不安全 sysctl，则 pod 会被调度，但不会部署：

```
$ oc get pod
```

NAME	READY	STATUS	RESTARTS	AGE
hello-pod	0/1	SysctlForbidden	0	14s

5.9.3. 启用不安全 sysctl

集群管理员可在非常特殊的情况下允许某些不安全 sysctl，比如高性能或实时应用程序性能优化。

如果要使用不安全 sysctl，集群管理员必须为特定类型的节点单独启用它们。sysctl 必须拥有命名空间。



警告

由于其不安全特性，使用不安全 sysctl 的风险由您自行承担，而且可能会造成严重问题，例如容器行为不当、资源短缺或节点受损。

流程

1. 在要运行带有不安全 sysctl 的容器的 MachineConfigPool 中添加标签：

```
$ oc edit machineconfigpool worker
```

```

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z

```



```
generation: 1
labels:
  custom-kubelet: sysctl ❶
```

- ❶ 添加 **key: pair** 标签。

2. 创建 KubeletConfig 自定义资源 (CR) :

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: custom-kubelet
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: sysctl ❶
  kubeletConfig:
    allowedUnsafeSysctls: ❷
    - "kernel.msg*"
    - "net.ipv4.route.min_pmtu"
```

- ❶ 指定来自 MachineConfigPool 的标签。
- ❷ 列出您想要允许的不安全 sysctl。

3. 创建对象 :

```
$ oc apply -f set-sysctl-worker.yaml
```

创建了名称采用 **99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet** 格式的 MachineConfig。

4. 使用 **machineconfigpool** 对象 **status** 字段等待集群重启 :

例如 :

```
status:
  conditions:
    - lastTransitionTime: '2019-08-11T15:32:00Z'
      message: >-
        All nodes are updating to
        rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
      reason: ""
      status: 'True'
      type: Updating
```

集群就绪时会显示类似如下的消息 :

```
- lastTransitionTime: '2019-08-11T16:00:00Z'
  message: >-
    All nodes are updated with
    rendered-worker-ccbfb5d2838d65013ab36300b7b3dc13
  reason: ""
  status: 'True'
  type: Updated
```

5. 集群就绪后，在新的 MachineConfig 中检查合并的 **KubeletConfig** :

```
$ oc get machineconfig 99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet -o json | grep ownerReference -A7
```

```
"ownerReferences": [  
  {  
    "apiVersion": "machineconfiguration.openshift.io/v1",  
    "blockOwnerDeletion": true,  
    "controller": true,  
    "kind": "KubeletConfig",  
    "name": "custom-kubelet",  
    "uid": "3f64a766-bae8-11e9-abe8-0a1a2a4813f2"
```

现在，您可以根据需要为 pod 添加不安全 sysctl。

第 6 章 操作集群

6.1. 查看 OPENSIFT CONTAINER PLATFORM 集群中的系统事件信息

OpenShift Container Platform 中的事件根据 OpenShift Container Platform 集群中 API 对象的事件进行建模。

6.1.1. 了解事件

事件允许 OpenShift Container Platform 以无关资源的方式记录实际事件的信息。它们还允许开发人员和管理员以统一的方式消耗系统组件的信息。

6.1.2. 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
```

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" iface to "eth0": no netns: failed to Statfs "/proc/33366/ns/net": no such file or directory
8m31s	Normal	Scheduled	pod/dapi-volume-test-pod	Successfully assigned openshift-config/dapi-volume-test-pod to ip-10-0-171-202.ec2.internal

- 从 OpenShift Container Platform 控制台查看项目中的事件。
 - 启动 OpenShift Container Platform 控制台。
 - 点击 **Home** → **Events**，再选择您的项目。
 - 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。

6.1.3. 事件列表

本节介绍 OpenShift Container Platform 的事件。

表 6.1. 配置事件

名称	描述
FailedValidation	pod 配置验证失败。

表 6.2. 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。
ExceededGrace Period	在指定宽限期内，容器运行时没有停止 pod。

表 6.3. 健康事件

名称	描述
Unhealthy	容器不健康。

表 6.4. 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。
Failed	拉取镜像失败。

名称	描述
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 6.5. 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 6.6. 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。
InsufficientFreeCPU	可用 CPU 不足。
InsufficientFreeMemory	可用内存不足。
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。

名称	描述
NodeSchedulable	节点可以调度。
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。

名称	描述
SuccessfulUnmountVolume	卷已被成功卸载。
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 6.7. Pod Worker 事件

名称	描述
FailedSync	Pod 同步失败。

表 6.8. 系统事件

名称	描述
SystemOOM	集群遇到 OOM（内存不足）状况。

表 6.9. Pod 事件

名称	描述
FailedKillPod	停止 pod 失败。

名称	描述
FailedCreatePodContainer	创建 pod 容器失败。
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 6.10. 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。

名称	描述
FailedRescale	新大小 : <size> ; 原因 : <msg> ; 错误 : <error-msg> 。
SuccessfulRescale	新大小 : <size> ; 原因 : <msg> 。
FailedUpdateStatus	未能更新状态。

表 6.11. 网络事件 (openshift-sdn)

名称	描述
Starting	正在启动 OpenShift-SDN。
NetworkFailed	pod 的网络接口已经丢失, pod 也将被停止。

表 6.12. 网络事件 (kube-proxy)

名称	描述
NeedPods	服务端口 <serviceName>:<port> 需要 pod。

表 6.13. 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。

名称	描述
ExternalProvisioning	在手动或通过外部软件置备声明的卷时发生。
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 6.14. 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 6.15. 部署

名称	描述
DeploymentCancellationFailed	未能取消部署。
DeploymentCancelled	已取消部署。
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 6.16. 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 6.17. DaemonSet 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 6.18. LoadBalancer 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。

6.2. 估算 OPENSHIFT CONTAINER PLATFORM 节点可以容纳的 POD 数量

作为集群管理员，您可以使用集群容量工具来查看可以调度的 pod 数，以便在资源耗尽前增加当前的资源，并确保以后的 pod 能被调度。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。

6.2.1. 了解 OpenShift Container Platform 集群容量工具

集群容量工具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，从而提供更加准确的估算。



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。

您可以从命令行中以单机实用程序的方式来运行集群容量分析工具，也可以作为 OpenShift Container Platform 集群里 pod 中的一个作业来运行。以 pod 内作业的方式运行时，可以将它运行多次而无需干预。

6.2.2. 在命令行中运行集群容量工具

您可以从命令行运行 OpenShift Container Platform 集群容量工具，以估算可以调度到集群的 pod 数量。

先决条件

- 下载并安装 [cluster-capacity 工具](#)。
- 创建一个示例 pod 规格文件，工具将使用该文件来估算资源用量。**podspec** 以 **limits** 或 **requests** 的形式指定资源要求。集群容量工具在估算分析时会考虑 pod 的资源要求。pod 规格输入的示例如下：

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
  labels:
    app: guestbook
    tier: frontend
spec:
  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
```

流程

在命令行中运行该工具：

1. 运行以下命令：

```
$ ./cluster-capacity --kubeconfig <path-to-kubeconfig> \ 1
--podspec <path-to-pod-spec> 2
```

1 指定 Kubernetes 配置文件的路径。

2 指定示例 pod 规格文件的路径

您还可以添加 **--verbose** 选项，以输出集群的各个节点上可以调度多少个 pod 的详细描述：

```
$ ./cluster-capacity --kubeconfig <path-to-kubeconfig> \
--podspec <path-to-pod-spec> --verbose
```

2. 查看输出结果，其类似于以下示例：

```
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)
```

在上例中，集群中估计可以调度的 pod 数量是 52。

6.2.3. 以 pod 中作业的方式运行集群容量工具

若以 pod 中作业的方式运行集群容量工具，其优点是可以在无需用户干预的前提下多次运行。以作业方式运行集群容量工具需要使用 **ConfigMap**。

先决条件

下载并安装 [cluster-capacity 工具](#)。

流程

运行集群容量工具：

1. 创建集群角色：

```
$ cat << EOF | oc create -f -
kind: ClusterRole
apiVersion: v1
metadata:
  name: cluster-capacity-role
rules:
- apiGroups: [""]
  resources: ["pods", "nodes", "persistentvolumeclaims", "persistentvolumes", "services"]
  verbs: ["get", "watch", "list"]
EOF
```

2. 创建服务帐户：

```
$ oc create sa cluster-capacity-sa
```

3. 将角色添加到服务帐户：

```
$ oc adm policy add-cluster-role-to-user cluster-capacity-role \
system:serviceaccount:default:cluster-capacity-sa
```

4. 定义并创建 pod 规格：

```
apiVersion: v1
kind: Pod
metadata:
  name: small-pod
```

```

labels:
  app: guestbook
  tier: frontend
spec:
  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

```

5. 集群容量分析使用名为 **cluster-capacity-configmap** 的 **ConfigMap** 挂载到卷中，从而将输入 pod 规格文件 **pod.yaml** 挂载到卷 **test-volume** 的路径 **/test-pod** 上。如果还没有创建 **ConfigMap**，请在创建作业前创建：

```

$ oc create configmap cluster-capacity-configmap \
  --from-file=pod.yaml=pod.yaml

```

6. 使用以下作业规格文件示例创建作业：

```

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 的 **pod.yaml** 键与 pod 规格文件相同，但这不是强制要求。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。

7. 以 pod 中作业的方式运行集群容量镜像：

```
$ oc create -f cluster-capacity-job.yaml
```

8. 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
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)
```

6.3. 设置限制范围

默认情况下，容器在 OpenShift Container Platform 集群上使用无限的计算资源运行。通过限制范围，您可以限制项目中特定对象的资源消耗：

- Pod 和容器：您可以为 Pod 及其容器设置 CPU 和内存的最小和最大要求。
- 镜像流：您可以设置 ImageStream 对象中的镜像和标签数量的限制。
- 镜像：您可以限制可推送到内部 registry 的镜像大小。
- 持久性卷声明(PVC):您可以限制请求的 PVC 的大小。

如果 Pod 未满足限制范围强制的限制，则无法在命名空间中创建 Pod。

6.3.1. 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一个 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"

```

6.3.1.1. 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限制范围对象。

6.3.1.1.1. 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限制范围对象中设置的值。如果没有，则 Pod 不会被创建。

- 对于在限制范围对象中指定的容器，容器 CPU 或内存请求和限制必须大于或等于 **min** 资源约束。
- 在限制范围对象中指定的容器，容器 CPU 或内存请求必须小于或等于最大 **资源** 约束。如果限值范围定义了 **max** CPU，则不需要在 Pod spec 中定义 CPU **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限制范围对象中指定的容器的 **maxLimitRequestRatio** 值。如果限制范围定义了 **maxLimitRequestRatio** 约束，则任何新容器都必须具有 **request** 和 **limit** 值。OpenShift Container Platform 通过 **limit** 除以 **request** 来计算限制与请求的比率。这个值应该是大于 1 的非负整数。

例如，如果容器的 **limit** 值中包括 **cpu: 500**，**request** 值中包括 **cpu: 100**，则 **cpu** 的限制与请求的比率是 **5**。这个比例必须小于或等于 **maxLimitRequestRatio**。

如果 Pod spec 没有指定容器资源内存或限制，则将限制范围对象中指定的容器的 **default** 或 **defaultRequestCPU** 和内存值分配给容器。

容器 LimitRange 对象定义

```

apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" ❶
spec:
  limits:
    - type: "Container"
      max:
        cpu: "2" ❷
        memory: "1Gi" ❸
      min:
        cpu: "100m" ❹
        memory: "4Mi" ❺
      default:
        cpu: "300m" ❻
        memory: "200Mi" ❼
      defaultRequest:
        cpu: "200m" ❽
        memory: "100Mi" ❾
      maxLimitRequestRatio:
        cpu: "10" ❿

```

- ❶ LimitRange 对象的名称。
- ❷ Pod 中单个容器可以请求的最大 CPU 量。
- ❸ Pod 中单个容器可以请求的最大内存量。
- ❹ Pod 中单个容器可以请求的最小 CPU 量。不设置 **min** 值或设置 **0** 代表没有限制，允许 Pod 使用超过 **max** CPU 的值。
- ❺ Pod 中单个容器可以请求的最小内存量。不设置 **min** 值或设置 **0** 代表没有限制，允许 Pod 使用超过 **max** 内存的值。
- ❻ 如果未在 Pod spec 中指定，容器可以使用的默认 CPU 量。
- ❼ 如果未在 Pod spec 中指定，容器可以使用的默认内存量。
- ❽ 如果未在 Pod spec 中指定，容器可以请求的默认 CPU 数量。
- ❾ 如果未在 Pod spec 中指定，容器可以请求的默认内存量。
- ❿ 容器最大的限制与请求的比率。

6.3.1.1.2. 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限制范围对象中设置的值。如果没有，则 Pod 不会被创建。

在 Pod 中的所有容器中，需要满足以下条件：

- 对于在限制范围对象中指定的 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限制在限制范围对象中指定的 Pod 的 **max** 资源限制。
- 容器限制与请求的比例必须小于或等于限制范围对象中指定的 **maxLimitRequestRatio** 约束。

Pod LimitRange 对象定义

```

apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" ❶
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" ❷
        memory: "1Gi" ❸
      min:
        cpu: "200m" ❹
        memory: "6Mi" ❺
      maxLimitRequestRatio:
        cpu: "10" ❻

```

- ❶ 限制范围对象的名称。
- ❷ Pod 可在所有容器间请求的最大 CPU 量。
- ❸ Pod 可在所有容器间请求的最大内存量。
- ❹ Pod 可在所有容器间请求的最小 CPU 量。不设置 **min** 值或设置 **0** 代表没有限制，允许 Pod 使用超过 **max** CPU 的值。
- ❺ Pod 可在所有容器间请求的最小内存量。不设置 **min** 值或设置 **0** 代表没有限制，允许 Pod 使用超过 **max** 内存的值。
- ❻ 容器最大的限制与请求的比率。

6.3.1.1.3. 镜像限制

限制范围允许您指定可推送到内部 registry 的镜像的最大大小。

将镜像推送到内部 registry 时，需要满足以下条件：

- 镜像大小必须小于或等于限制范围对象中指定的镜像的 **max** 值。

镜像 LimitRange 对象定义

```

apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "resource-limits" ❶
spec:
  limits:

```

```
- type: openshift.io/Image
  max:
    storage: 1Gi 2
```

- 1 限制范围对象的名称。
- 2 可以推送到内部 registry 的最大镜像大小。



注意

要防止超过限制的 Blob 上传到 registry，则必须将 registry 配置为强制实施配额。



警告

在上传的镜像清单中，镜像大小并非始终可用。这对使用 Docker 1.10 或更高版本构建并推送到 v2 registry 的镜像来说尤为如此。如果这样的镜像使用旧的 Docker 守护进程拉取，由 registry 将镜像清单转换为 schema v1 时缺少了所有与大小相关的信息。镜像没有设置存储限制会阻止镜像被上传。

[这个问题正在被决。](#)

6.3.1.1.4. 镜像流限值

您可以通过一个限制范围来为镜像流指定限值。

对于每个镜像流，需要满足以下条件：

- 镜像流规格中的镜像标签数量必须小于或等于限制范围对象中的 **openshift.io/image-tags** 约束。
- 镜像流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 限制范围对象的名称。
- 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，镜像流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内部 registry 的大量镜像进行限制。内部和外部引用无法区分。

6.3.1.1.5. 持久性卷声明 (PVC) 限制

限制范围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限制范围对象中指定的 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限制范围对象中指定的 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
```

- 限制范围对象的名称。
- 持久性卷声明中可请求的最小存储量。
- 在持久性卷声明中请求的最大存储量。

6.3.2. 创建限制范围

将限制范围应用到一个项目：

- 使用您的所需规格创建限制范围对象：

```
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" ❸
max:
  cpu: "2"
  memory: "1Gi"
min:
  cpu: "100m"
  memory: "4Mi"
default: ❹
  cpu: "300m"
  memory: "200Mi"
defaultRequest: ❺
  cpu: "200m"
  memory: "100Mi"
maxLimitRequestRatio: ❻
  cpu: "10"
- type: openshift.io/Image ❼
max:
  storage: 1Gi
- type: openshift.io/ImageStream ❽
max:
  openshift.io/image-tags: 20
  openshift.io/images: 30
- type: "PersistentVolumeClaim" ❾
min:
  storage: "2Gi"
max:
  storage: "50Gi"

```

- ❶ 为 LimitRange 对象指定一个名称。
- ❷ 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。
- ❸ 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。
- ❹ 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。
- ❺ 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。
- ❻ 可选。对于容器，指定 Pod 规格中可以指定的最大限制到请求比例。
- ❼ 要为镜像对象设置限制，请设置可推送到内部 registry 的镜像的最大值。
- ❽ 要为镜像流设置限值，请根据需要设置镜像流对象文件中的最大镜像标签和引用数。
- ❾ 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> ❶
```

- ❶ 指定您创建的 YAML 文件的名称以及要应用限制的项目。

6.3.3. 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. 获取项目中定义的限值范围列表。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. 限制范围的信息，例如 **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 -     -     -
```

6.3.4. 删除限制范围

要删除一个项目中的所有活跃的限制范围使它不再起作用：

1. 运行以下命令：

```
$ oc delete limits <limit_name>
```

6.4. 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。

6.4.1. 了解管理应用程序内存

在继续操作前，建议您先通篇阅读有关 OpenShift Container Platform 如何管理计算资源的概述。

对于每种资源（内存、CPU 或存储），OpenShift Container Platform 允许在 pod 中的各个容器上设置可选的 **request** 和 **limit** 值。

注意以下关于内存请求和内存限制的信息：

- **内存请求**
 - 如果指定，内存请求值会影响 OpenShift Container Platform 调度程序。将容器调度到节点时，调度程序会考虑内存请求，然后在所选节点上隔离出请求的内存供该容器使用。
 - 如果节点的内存已用尽，OpenShift Container Platform 将优先驱除其内存用量超出内存请求最多的容器。在严重的内存耗尽情形中，节点 OOM 终止程序可以根据类似的指标选择并终止容器中的一个进程。
 - 集群管理员可以分配配额，或者分配内存请求值的默认值。
 - 集群管理员可以覆盖开发者指定的内存请求值，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 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。

6.4.1.1. 管理应用程序内存策略

如下是 OpenShift Container Platform 上调整应用程序内存大小的步骤：

1. 确定预期的容器内存用量

从经验判断（例如，通过独立的负载测试），根据需要确定容器内存用量的预期平均值和峰值。需要考虑容器中有可能并行运行的所有进程：例如，主应用程序是否生成任何辅助脚本？

2. 确定风险嗜好

确定用于驱除的风险嗜好。如果风险嗜好较低，则容器应根据预期的峰值用量加上一个安全裕度百分比来请求内存。如果风险嗜好较高，那么根据预期的平均用量请求内存可能更为妥当。

3. 设定容器内存请求

根据以上所述设定容器内存请求。请求越能准确表示应用程序内存用量越好。如果请求过高，集群和配额用量效率低下。如果请求过低，应用程序驱除的几率就会提高。

4. 根据需要设定容器内存限制

在必要时，设定容器内存限制。如果容器中所有进程的总内存用量超过限制，那么设置限制会立即终止容器进程，所以这既有利也有弊。一方面，可能会导致过早出现意料之外的过量内存使用（“快速失败”）；另一方面，也会突然终止进程。

需要注意的是，有些 OpenShift Container Platform 集群可能要求设置限制；有些集群可能会根据限制覆盖请求；而且有些应用程序镜像会依赖于设置的限制，因为这比请求值更容易检测。

如果设置内存限制，其大小不应小于预期峰值容器内存用量加上安全裕度百分比。

5. 确保应用程序经过性能优化

在适当时，确保应用程序已根据配置的请求和限制进行了性能优化。对于池化内存的应用程序（如 JVM），这一步尤为相关。本页的其余部分将介绍这方面的内容。

6.4.2. 了解 OpenShift Container Platform 的 OpenJDK 设置

默认的 OpenJDK 设置在容器化环境中效果不佳。因此在容器中运行 OpenJDK 时，务必要提供一些额外的 Java 内存设置。

JVM 内存布局比较复杂，并且视版本而异，因此本文不做详细讨论。但作为在容器中运行 OpenJDK 的起点，至少以下三个于内存相关的任务非常重要：

1. 覆盖 JVM 最大堆大小。
2. 在可能的情况下，促使 JVM 向操作系统释放未使用的内存。
3. 确保正确配置了容器中的所有 JVM 进程。

优化容器中运行的 JVM 工作负载已超出本文讨论范畴，并且可能涉及设置多个额外的 JVM 选项。

6.4.2.1. 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：

1. 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**。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）。**
2. 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。

6.4.2.2. 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。

OpenShift Container Platform Jenkins maven slave 镜像使用以下 JVM 参数来促使 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。

6.4.2.3. 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** 和 **MAVEN_OPTS** 等) 来配置它们的 JVM，或许难以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，为确保在 slave 镜像中运行的所有 JVM 工作负载都默认使用这些选项，OpenShift Container Platform Jenkins maven slave 镜像将进行以下设置：

```
JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"
```

这不能保证不需要额外选项，只是用作一个实用的起点。

6.4.3. 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：

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  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
```

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。

1. 创建 pod :

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

2. 使用远程 shell 访问 pod :

```
$ oc rsh test
```

3. 检查是否应用了请求的值 :

```
$ env | grep MEMORY | sort
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。

6.4.4. 了解 OOM 终止策略

如果容器中所有进程的内存总用量超过内存限制，或者在严重的节点内存耗尽情形下，OpenShift Container Platform 可以终止容器中的某个进程。

当一个进程被 OOM 终止时，这有可能会造成容器立即退出，但也不一定。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. `/sys/fs/cgroup/memory/memory.oom_control` 中的 `oom_kill` 计数器递增

```
$ grep '^oom_kill' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 0
$ sed -e " </dev/zero # provoke an OOM kill
Killed
$ echo $?
137
$ grep '^oom_kill' /sys/fs/cgroup/memory/memory.oom_control
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。OOM 终止的 pod 可以根据 **restartPolicy** 的值来重新启动。如果不重启，ReplicationController 等控制器会看到 pod 的失败状态，并创建一个新 pod 来取代旧 pod。

如果不重启，pod 状态如下：

```
$ oc get pod test
NAME    READY   STATUS    RESTARTS   AGE
test    0/1     OOMKilled 0           1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: false
    restartCount: 0
  state:
    terminated:
      exitCode: 137
      reason: OOMKilled
  phase: Failed
```

如果重启，其状态如下：

```
$ oc get pod test
NAME    READY   STATUS    RESTARTS   AGE
test    1/1     Running   1           1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
  lastState:
    terminated:
      exitCode: 137
      reason: OOMKilled
  state:
    running:
  phase: Running
```

6.4.5. 了解 pod 驱除

OpenShift Container Platform 可在节点内存耗尽时从节点上驱除 pod。根据内存耗尽的程度，驱除可能是安全操作，但也不一定。安全驱除表示，各个容器的主进程 (PID 1) 收到 SIGTERM 信号，稍等片刻后，如果进程还未退出，则会收到一个 SIGKILL 信号。非安全驱除暗示着各个容器的主进程会立即收到 SIGKILL 信号。

被驱除的 pod 将具有原因为 **Evicted** 的 **Failed** 阶段。无论 **restartPolicy** 的值是什么，该 pod 都不会重启。但是，ReplicationController 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
NAME    READY   STATUS    RESTARTS   AGE
test    0/1     Evicted   0           1m

$ oc get pod test -o yaml
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
```

6.5. 配置集群以将 POD 放置到过量使用的节点上

处于*过量使用* (*overcommitted*) 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。



注意

OpenShift Container Platform 中默认启用过量使用。请参阅[禁用节点过量使用](#)。

6.5.1. 了解过量使用

请求和限制可让管理员允许和管理节点上资源的过量使用。调度程序使用请求来调度容器，并提供最低服务保障。限制约束节点上可以消耗的计算资源数量。

OpenShift Container Platform 管理员可以通过配置主控机 (master) 来覆盖开发人员容器上设置的请求和限制之间的比率，来控制过量使用的程度并管理节点上的容器密度。与项目一级上的用于指定限制和默认值的 LimitRange 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。

6.5.2. 了解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。

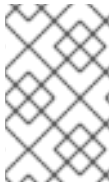
调度基于请求的资源，而配额和硬限制指的是资源限制，它们可以设置为高于请求的资源。请求和限制的差值决定了过量使用程度；例如，如果为容器赋予 1Gi 内存请求和 2Gi 内存限制，则根据 1Gi 请求将容器调度到节点上，但最多可使用 2Gi；因此过量使用为 200%。

6.5.2.1. 为 Fluentd 配置缓冲数据块限制

如果 Fluentd 日志记录器保存大量日志，Fluentd 会执行文件缓冲来降低内存用量并防止数据丢失。

Fluentd 文件缓冲以 *数据块* 的形式存储记录。数据块存储在 *缓冲区* 中。

您可以通过编辑 Fluentd Daemonset 中的环境变量来调整集群中的文件缓冲：



注意

修改 Fluentd Daemonset 中的 **FILE_BUFFER_LIMIT** 或 **BUFFER_SIZE_LIMIT** 参数，您需要把集群日志系统设置为未管理（unmanaged）状态。处于非受管状态的 Operator 不被正式支持，集群管理员需要完全掌控各个组件的配置和升级。

- **BUFFER_SIZE_LIMIT**。此参数决定 Fluentd 创建新块之前每个块文件的最大大小。默认值为 **8M**。此参数设置 Fluentd **chunk_limit_size** 变量。
高 **BUFFER_SIZE_LIMIT** 可以使每个块文件收集更多记录。但较大的记录需要更长的时间才会被发送到日志存储中。
- **FILE_BUFFER_LIMIT**。这个参数决定每个日志输出的文件缓冲区大小。此值只是基于调度 Fluentd pod 的节点中的可用空间的请求。OpenShift Container Platform 不允许 Fluentd 超过节点容量。默认值为 **256Mi**。
高 **FILE_BUFFER_LIMIT** 意味着基于输出数量的更高的 **BUFFER_QUEUE_LIMIT**。但是，如果节点的空间面临压力，Fluentd 可能会失败。

默认情况下，如果所有日志都发送到单个资源，则 **number_of_outputs** 为 **1**，否则每多一个资源就会递增 **1**。如果您使用 Log Forwarding API、Fluentd **Forward** 协议或 syslog 协议将日志转发到外部位置，则可能会有多个输出。

持久性卷大小必须大于 **FILE_BUFFER_LIMIT** 与输出相乘的结果。

- **BUFFER_QUEUE_LIMIT**。这个参数是允许的最大缓冲块数量。**BUFFER_QUEUE_LIMIT** 参数无法直接调整。OpenShift Container Platform 根据日志记录输出数量、块大小以及可用文件系统空间来计算这个值。默认是 **32** 块。要修改 **BUFFER_QUEUE_LIMIT**，需要修改 **FILE_BUFFER_LIMIT** 的值。**BUFFER_QUEUE_LIMIT** 参数设置 Fluentd **queue_limit_length** 参数。
OpenShift Container Platform 计算 **BUFFER_QUEUE_LIMIT** 的公式是 $(\text{FILE_BUFFER_LIMIT} / (\text{number_of_outputs} * \text{BUFFER_SIZE_LIMIT}))$ 。

使用默认设置的值，**BUFFER_QUEUE_LIMIT** 的值是 **32**：

- **FILE_BUFFER_LIMIT = 256Mi**
- **number_of_outputs = 1**
- **BUFFER_SIZE_LIMIT = 8Mi**

OpenShift Container Platform 使用 Fluentd [file 缓冲插件](#) 来配置存储块的方式。您可以使用以下命令查看缓冲区文件的位置：

```
$ oc get cm fluentd -o json | jq -r '.data."fluent.conf"'
```

```
<buffer>
  @type file 1
  path '/var/lib/flunetd/retry-elasticsearch' 2
```

1 Fluentd **file** 缓冲插件。不要更改这个值。

2 保存缓冲块的路径。

先决条件

- 将集群日志记录设置为非受管状态。处于非受管状态的 Operator 不被正式支持，集群管理员需要完全掌控各个组件的配置和升级。

流程

配置缓冲区块限制：

1. 编辑 **fluentd** Daemonset 中的以下任一参数。

```
spec:
  template:
    spec:
      containers:
        env:
          - name: FILE_BUFFER_LIMIT 1
            value: "256"
          - name: BUFFER_SIZE_LIMIT 2
            value: 8Mi
```

1 指定每个输出的 Fluentd 文件缓冲大小。

2 指定每个 Fluentd 缓冲块的最大大小。

6.5.3. 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。

6.5.3.1. 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。

6.5.3.2. 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。

6.5.4. 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。

对于每个计算资源，容器划分到三个 QoS 类中的一个，它们按照优先级降序排列：

表 6.19. 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，则容器被归类为 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，则容器被归类为 Burstable 。
3 (最低)	BestEffort	如果没有为任何资源设置请求和限制，则容器被归类为 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。

6.5.4.1. 了解如何为不同的服务质量层级保留内存

您可以使用 **qos-reserved** 参数指定在特定 QoS 级别上 pod 要保留的内存百分比。此功能尝试保留请求的资源，阻止较低 QoS 类中的 pod 使用较高 QoS 类中 pod 所请求的资源。

OpenShift Container Platform 按照如下所示使用 **qos-reserved** 参数：

- 值为 **qos-reserved=memory=100%** 时，阻止 **Burstable** 和 **BestEffort** QoS 类消耗较高 QoS 类所请求的内存。这会增加 **BestEffort** 和 **Burstable** 工作负载上为了提高 **Guaranteed** 和 **Burstable** 工作负载的内存资源保障而遭遇 OOM 的风险。
- 值为 **qos-reserved=memory=50%** 时，允许 **Burstable** 和 **BestEffort** QoS 类消耗较高 QoS 类所请求的内存的一半。
- 值为 **qos-reserved=memory=0%** 时，允许 **Burstable** 和 **BestEffort** QoS 类最多消耗节点的所有可分配数量（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。

6.5.5. 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工作。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。

6.5.6. 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。

在过量使用的环境中，务必要正确配置节点，以提供最佳的系统行为。

当节点启动时，它会确保为内存管理正确设置内核可微调标识。除非物理内存不足，否则内核应该永不会在内存分配时失败。

为确保这一行为，OpenShift Container Platform 通过将 **vm.overcommit_memory** 参数设置为 **1** 来覆盖默认操作系统设置，从而将内核配置为始终过量使用内存。

OpenShift Container Platform 还通过将 **vm.panic_on_oom** 参数设置为 **0**，将内核配置为不会在内存不足时崩溃。设置为 0 可告知内核在内存不足 (OOM) 情况下调用 oom_killer，以根据优先级终止进程

您可以通过对节点运行以下命令来查看当前的设置：

```
$ sysctl -a |grep commit
vm.overcommit_memory = 1
```

```
$ sysctl -a |grep panic
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

6.5.6.1. 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。

先决条件

1. 为您要配置的节点类型获取与静态机器配置池 CRD 关联的标签。执行以下步骤之一：

- a. 查看 Machine Config Pool：

```
$ oc describe machineconfigpool <name>
```

例如：

```
$ oc describe machineconfigpool worker

apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  creationTimestamp: 2019-02-08T14:52:39Z
  generation: 1
  labels:
    custom-kubelet: small-pods ❶
```

- ❶ 如果添加了标签，它会出现在 **labels** 下。

- b. 如果标签不存在，则添加一个键/值对：

```
$ 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 ❶
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: small-pods ❷
  kubeletConfig:
    cpu-cfs-quota: ❸
      - "false"
```

- ❶ 为 CR 分配一个名称。
- ❷ 指定要应用配置更改的标签。
- ❸ 将 **cpu-cfs-quota** 参数设置为 **false**。

如果需要禁用 CPU 限制强制实施，则务必要了解其对您的节点可能造成的影响：

- 如果容器发出 CPU 请求，它将继续由 Linux 内核中的 CFS 份额来实施。
- 如果容器没有明确发出 CPU 请求，但它确实指定了限制，则请求将默认为指定的限制，并且由 Linux 内核中的 CFS 份额来实施。
- 如果容器同时指定 CPU 请求和限制，则由 Linux 内核中的 CFS 份额来实施请求，并且限制不会对节点产生影响。

6.5.6.2. 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让集群正常工作，如 `sshd` 等）。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”。

6.5.7. 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
```

6.5.8. 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. 编辑项目对象文件
2. 添加以下注解：

```
quota.openshift.io/cluster-resource-override-enabled: "false"
```

3. 创建项目对象：

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

6.6. 使用 FEATUREGATE 启用 OPENSIFT CONTAINER PLATFORM 功能

作为管理员，您可以开启技术预览功能。

6.6.1. 了解 FeatureGate 和技术预览功能

您可以使用 FeatureGate 自定义资源在集群中开启和关闭技术预览功能。

例如，这可让您确保为生产集群关闭技术预览功能，同时为测试集群保留这些功能，从而能完整地进行测试。

6.6.2. 受 FeatureGate 影响的功能

以下功能受到 FeatureGate 的影响：

FeatureGate	描述	默认
RotateKubeletServerCertificate	在集群中启用服务器 TLS 证书轮转。	True
SupportPodPidsLimit	支持限制 Pod 中的运行进程 (PID) 数。	True
MachineHealthCheck	启用自动修复机器池中不健康的机器。	True
LocalStorageCapacityIsolation	启用本地临时存储消耗，以及 emptyDir 卷的 sizeLimit 属性。	False

您可以通过编辑 Feature Gate 自定义资源启用这些功能。这些功能一旦开启便无法撤消，而且会妨碍集群升级。

6.6.3. 使用 FeatureGate 启用技术预览功能

您可以通过编辑 `openshift-config` 项目中的 FeatureGate 自定义资源（名为 `cluster`），为集群中的所有节点开启和关闭技术预览功能。

FeatureGate 可以启用以下技术预览功能：

- **RotateKubeletServerCertificate**
- **SupportPodPidsLimit**



重要

开启技术预览功能后无法撤消操作，而且会妨碍升级。

流程

为整个集群开启技术预览功能：

1. 创建 FeatureGate 实例：
 - a. 切换到 **Administration** → **Custom Resource Definitions** 页面。
 - b. 在 **Custom Resource Definitions** 页面中，点击 **FeatureGate**。

- c. 在 Custom Resource Definitions 页面中，点击 Actions Menu 并选择 View Instances。
- d. 在 Feature Gates 页面中，点击 Create Feature Gates。
- e. 用以下示例替换代码：

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster
spec: {}
```

- f. 点击 Create。

2. 要打开技术预览功能，请将 spec 参数更改为：

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  name: cluster
spec:
  featureSet: TechPreviewNoUpgrade 1
```

- 1** 添加 **featureSet: TechPreviewNoUpgrade**，以启用受 FeatureGate 影响的技术预览功能。



注意

开启技术预览功能后无法撤消操作，而且会妨碍升级。