OpenShift Container Platform 4.8

可伸缩性和性能

扩展 OpenShift Container Platform 集群并调整产品环境的性能

Last Updated: 2023-06-12

OpenShift Container Platform 4.8 可伸缩性和性能

扩展 OpenShift Container Platform 集群并调整产品环境的性能

法律通告

Copyright © 2023 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

本文档提供了扩展集群和优化 OpenShift Container Platform 环境性能的说明。

目录

第 1 章 推荐的主机实践 5
1.1. 推荐的节点主机实践 5
1.2. 创建 KUBELETCONFIG CRD 来编辑 KUBELET 参数 5
1.3. 修改不可用 WORKER 节点的数量 9
1.4. CONTROL PLANE 节点大小 10
1.5. 推荐的 ETCD 实践 12
1.6. 分离 ETCD 数据 14
1.7. OPENSHIFT CONTAINER PLATFORM 基础架构组件 17
1.8. 移动监控解决方案 17
1.9. 移动默认 REGISTRY 20
1.10. 移动路由器 21
1.11. 基础架构节点大小 23
1.12. 其他资源 24

第 2 章 IBM Z 和 LINUXONE 环境的推荐主机实践 25
2.1. 管理 CPU 过量使用 25
2.2. 禁用透明巨页 25
2.3. 使用 RECEIVE FLOW STEERING（RFS）提高网络性能 25
2.4. 选择您的网络设置 27
2.5. 确保 Z/VM 上使用 HYPERPAV 的高磁盘性能 27
2.6. IBM Z 主机上的 RHEL KVM 建议 29

第 3 章 推荐的集群扩展实践 33
3.1. 扩展集群的建议实践 33
3.2. 修改机器集 33
3.3. 关于机器健康检查 35
3.4. MACHINEHEALTHCHECK 资源示例 36
3.5. 创建 MACHINEHEALTHCHECK 资源 37

第 4 章 使用 NODE TUNING OPERATOR 39
4.1. 关于 NODE TUNING OPERATOR 39
4.2. 访问 NODE TUNING OPERATOR 示例规格 39
4.3. 在集群中设置默认配置集 39
4.4. 验证是否应用了 TUNED 配置集 41
4.5. 自定义调整规格 41
4.6. 自定义调整示例 45
4.7. 支持的 TUNED 守护进程插件 47

第 5 章 使用 CLUSTER LOADER 48
5.1. 安装 CLUSTER LOADER 48
5.2. 运行 CLUSTER LOADER 48
5.3. 配置 CLUSTER LOADER 48
5.4. 已知问题 52

第 6 章 使用 CPU MANAGER 54
6.1. 设置 CPU MANAGER 54

第 7 章 使用拓扑管理器 59
7.1. 拓扑管理器策略 59
7.2. 设置拓扑管理器 59
7.3. POD 与拓扑管理器策略的交互 60

第 8 章 扩展 CLUSTER MONITORING OPERATOR 62

8.1. PROMETHEUS 数据库存储要求 62
8.2. 配置集群监控 62

第 9 章 NODE FEATURE DISCOVERY OPERATOR 64
9.1. 关于 NODE FEATURE DISCOVERY OPERATOR 64
9.2. 安装 NODE FEATURE DISCOVERY OPERATOR 64
9.3. 使用 NODE FEATURE DISCOVERY OPERATOR 66
9.4. 配置 NODE FEATURE DISCOVERY OPERATOR 70

第 10 章 驱动程序工具包（DRIVER TOOLKIT） 76
10.1. 关于驱动程序工具包 76
10.2. 拉取 DRIVER TOOLKIT 容器镜像 77
10.3. 使用 DRIVER TOOLKIT 78

第 11 章 根据对象限制规划您的环境 82
11.1. OPENSHIFT CONTAINER PLATFORM 为主发行版本测试了集群最大值 82
11.2. 测试集群最大值的 OPENSHIFT CONTAINER PLATFORM 环境和配置 83
11.3. 如何根据经过测试的集群限制规划您的环境 85
11.4. 如何根据应用程序要求规划您的环境 86

第 12 章 优化存储 89
12.1. 可用的持久性存储选项 89
12.2. 推荐的可配置存储技术 89
12.3. 数据存储管理 92

第 13 章 优化路由 93
13.1. INGRESS CONTROLLER（ROUTER）性能的基线 93
13.2. INGRESS CONTROLLER（路由器）性能优化 94

第 14 章 优化网络 95
14.1. 为您的网络优化 MTU 95
14.2. 安装大型集群的实践建议 95
14.3. IPSEC 的影响 96

第 15 章 管理裸机主机 97
15.1. 关于裸机主机和节点 97
15.2. 维护裸机主机 97

第 16 章 巨页的作用及应用程序如何使用它们 101
16.1. 巨页的作用 101
16.2. 应用程序如何使用巨页 101
16.3. 使用 DOWNWARD API 消耗巨页资源 102
16.4. 配置巨页 104
16.5. 禁用透明巨页 105

第 17 章 低延迟节点的 PERFORMANCE ADDON OPERATOR 107
17.1. 了解低延迟 107
17.2. 安装 PERFORMANCE ADDON OPERATOR 107
17.3. 升级 PERFORMANCE ADDON OPERATOR 110
17.4. 置备实时和低延迟工作负载 112
17.5. 使用性能配置集调整节点以实现低延迟 124
17.6. 使用 PERFORMANCE ADDON OPERATOR 减少 NIC 队列 128
17.7. 调试低延迟 CNF 调整状态 135
17.8. 为红帽支持收集调试数据延迟 137

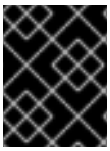
第 18 章 为平台验证执行延迟测试 140

18.1. 运行延迟测试的先决条件 140
18.2. 关于延迟测试的发现模式 140
18.3. 测量延迟 140
18.4. 运行延迟测试 141
18.5. 生成延迟测试失败报告 145
18.6. 生成 JUNIT 延迟测试报告 145
18.7. 在断开连接的集群中运行延迟测试 146
18.8. 对 CNF-TESTS 容器的错误进行故障排除 148

第 19 章 创建性能配置集 149
19.1. 关于性能配置集创建器 149
19.2. 其他资源 160

第 1 章 推荐的主机实践

本节为 OpenShift Container Platform 提供推荐的主机实践。



重要

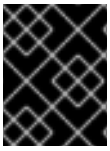
这些指南适用于带有软件定义网络（SDN）而不是开放虚拟网络（OVN）的 OpenShift Container Platform。

1.1. 推荐的节点主机实践

OpenShift Container Platform 节点配置文件包含重要的选项。例如，控制可以为节点调度的最大 pod 数量的两个参数: **podsPerCore** 和 **maxPods**。

当两个参数都被设置时，其中较小的值限制了节点上的 pod 数量。超过这些值可导致：

- CPU 使用率增加。
- 减慢 pod 调度的速度。
- 根据节点中的内存数量，可能出现内存耗尽的问题。
- 耗尽 IP 地址池。
- 资源过量使用，导致用户应用程序性能变差。



重要

在 Kubernetes 中，包含单个容器的 pod 实际使用两个容器。第二个容器用来在实际容器启动前设置联网。因此，运行 10 个 pod 的系统实际上会运行 20 个容器。

注意

云供应商的磁盘 IOPS 节流可能会对 CRI-O 和 kubelet 产生影响。当节点上运行大量 I/O 高负载的 pod 时，可能会出现超载的问题。建议您监控节点上的磁盘 I/O，并使用有足够吞吐量的卷。

podsPerCore 根据节点中的处理器内核数来设置节点可运行的 pod 数量。例如：在一个有 4 个处理器内核的节点上将 **podsPerCore** 设为 **10**，则该节点上允许的最大 pod 数量为 **40**。

```
kubeletConfig:
  podsPerCore: 10
```

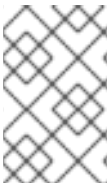
将 **podsPerCore** 设置为 **0** 可禁用这个限制。默认为 **0**。**podsPerCore** 不能超过 **maxPods**。

maxPods 把节点可以运行的 pod 数量设置为一个固定值，而不需要考虑节点的属性。

```
kubeletConfig:
  maxPods: 250
```

1.2. 创建 KUBELETCONFIG CRD 来编辑 KUBELET 参数

kubelet 配置目前被序列化为 Ignition 配置，因此可以直接编辑。但是，在 Machine Config Controller (MCC) 中同时添加了新的 **kubelet-config-controller**。这可让您使用 **KubeletConfig** 自定义资源 (CR) 来编辑 kubelet 参数。



注意

因为 **kubeletConfig** 对象中的字段直接从上游 Kubernetes 传递给 kubelet，kubelet 会直接验证这些值。**kubeletConfig** 对象中的无效值可能会导致集群节点不可用。有关有效值，请参阅 Kubernetes 文档。

请考虑以下指导：

- 为每个机器配置池创建一个 **KubeletConfig** CR，带有该池需要更改的所有配置。如果要将相同的内容应用到所有池，则所有池仅需要一个 **KubeletConfig** CR。
- 编辑现有的 **KubeletConfig** CR 以修改现有设置或添加新设置，而不是为每个更改创建一个 CR。建议您仅创建一个 CR 来修改不同的机器配置池，或用于临时更改，以便您可以恢复更改。
- 根据需要，创建多个 **KubeletConfig** CR，每个集群限制为 10。对于第一个 **KubeletConfig** CR，Machine Config Operator (MCO) 会创建一个机器配置，并附带 **kubelet**。对于每个后续 CR，控制器会创建另一个带有数字后缀的 **kubelet** 机器配置。例如，如果您有一个带有 **-2** 后缀的 **kubelet** 机器配置，则下一个 **kubelet** 机器配置会附加 **-3**。

如果要删除机器配置，以相反的顺序删除它们，以避免超过限制。例如，在删除 **kubelet-2** 机器配置前删除 **kubelet-3** 机器配置。

注意

如果您有一个带有 **kubelet-9** 后缀的机器配置，并且创建了另一个 **KubeletConfig** CR，则不会创建新的机器配置，即使少于 10 个 **kubelet** 机器配置。

KubeletConfig CR 示例

```
$ oc get kubeletconfig
```

```
NAME                AGE
set-max-pods        15m
```

显示 KubeletConfig 机器配置示例

```
$ oc get mc | grep kubelet
```

```
...
99-worker-generated-kubelet-1                  b5c5119de007945b6fe6fb215db3b8e2ceb12511    3.2.0
26m
...
```

以下流程演示了如何配置 worker 节点上的每个节点的最大 pod 数量。

先决条件

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

a. 查看机器配置池：

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

例如：

```
$ oc describe machineconfigpool worker
```

输出示例

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

流程

1. 查看您可以选择的可用机器配置对象：

```
$ oc get machineconfig
```

默认情况下，与 kubelet 相关的配置为 **01-master-kubelet** 和 **01-worker-kubelet**。

2. 检查每个节点的最大 pod 的当前值：

```
$ oc describe node <node_name>
```

例如：

```
$ oc describe node ci-ln-5grqprb-f76d1-ncnqq-worker-a-mdv94
```

在 **Allocatable** 小节中找到 **value: pods: <value>**：

输出示例

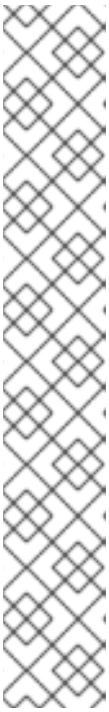
```
Allocatable:
 attachable-volumes-aws-ebs:  25
 cpu:                         3500m
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      15341844Ki
 pods:                        250
```

3. 通过创建一个包含 kubelet 配置的自定义资源文件，设置 worker 节点上的每个节点的最大 pod：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-max-pods
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: set-max-pods (1)
  kubeletConfig:
    maxPods: 500 (2)
```

(1) 输入机器配置池中的标签。

(2) 添加 kubelet 配置。在本例中，使用 **maxPods** 设置每个节点的最大 pod。



注意

kubelet 与 API 服务器进行交互的频率取决于每秒的查询数量 (QPS) 和 burst 值。如果每个节点上运行的 pod 数量有限，使用默认值（**kubeAPIQPS** 为 **50**，**kubeAPIBurst** 为 **100**）就可以。如果节点上有足够 CPU 和内存资源，则建议更新 kubelet QPS 和 burst 速率。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: set-max-pods
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: set-max-pods
  kubeletConfig:
    maxPods: <pod_count>
    kubeAPIBurst: <burst_rate>
    kubeAPIQPS: <QPS>
```

a. 为带有标签的 worker 更新机器配置池：

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

b. 创建 **KubeletConfig** 对象：

```
$ oc create -f change-maxPods-cr.yaml
```

c. 验证 **KubeletConfig** 对象是否已创建：

```
$ oc get kubeletconfig
```

输出示例

```
NAME              AGE
set-max-pods      15m
```

根据集群中的 worker 节点数量，等待每个 worker 节点被逐个重启。对于有 3 个 worker 节点的集群，这个过程可能需要大约 10 到 15 分钟。

4. 验证更改是否已应用到节点：

   a. 在 worker 节点上检查 **maxPods** 值已更改：

```
$ oc describe node <node_name>
```

   b. 找到 **Allocatable** 小节：

```
 ...
Allocatable:
  attachable-volumes-gce-pd:  127
  cpu:                        3500m
  ephemeral-storage:          123201474766
  hugepages-1Gi:              0
  hugepages-2Mi:              0
  memory:                     14225400Ki
  pods:                       500 (1)
 ...
```

(1) 在本例中，**pods** 参数应报告您在 **KubeletConfig** 对象中设置的值。

5. 验证 **KubeletConfig** 对象中的更改：

```
$ oc get kubeletconfigs set-max-pods -o yaml
```

这应该会显示 **status: "True"** 和 **type:Success**：

```
spec:
  kubeletConfig:
    maxPods: 500
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: set-max-pods
status:
  conditions:
  - lastTransitionTime: "2021-06-30T17:04:07Z"
    message: Success
    status: "True"
    type: Success
```

1.3. 修改不可用 WORKER 节点的数量

默认情况下，在对可用的 worker 节点应用 kubelet 相关的配置时，只允许一台机器不可用。对于大型集群来说，它可能需要很长时间才可以反映出配置的更改。在任何时候，您可以调整更新的机器数量来加快进程速度。

流程

1. 编辑 **worker** 机器配置池：

```
$ oc edit machineconfigpool worker
```

2. 将 **maxUnavailable** 设置为您需要的值：

```
spec:
  maxUnavailable: <node_count>
```

重要

当设置该值时，请考虑无法使用的 worker 节点数量，而不影响在集群中运行的应用程序。

1.4. CONTROL PLANE 节点大小

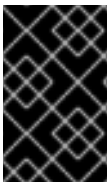
control plane 节点对资源的要求取决于集群中的节点数量。以下推荐的 control plane 节点大小是基于 control plane 密度测试的结果。control plane 测试会根据节点数在每个命名空间中在集群中创建以下对象：

- 12 个镜像流
- 3 个构建配置
- 6 个构建
- 1 个部署，带有 2 个 pod 副本，每个都挂载两个 secret
- 2 个部署，带有 1 个 pod 副本，挂载了两个 secret
- 3 个指向以前部署的服务
- 3 个指向之前部署的路由
- 10 个 secret，其中 2 个由以前的部署挂载
- 10 个配置映射，其中 2 个由以前的部署挂载

worker 节点数量	集群负载（命名空间）	CPU 内核	内存 (GB)
25	500	4	16
100	1000	8	32
250	4000	16	96

在具有三个 master 或 control plane 节点的大型高密度集群中，当其中一个节点停止、重启或失败时，CPU 和内存用量将会激增。故障可能是因为电源、网络或底层基础架构出现意外问题，除了在关闭集群后重启集群以节约成本的情况下。其余两个 control plane 节点必须处理负载才能高度可用，从而增加资源使用量。另外，在升级过程中还会有这个预期，因为 master 被封锁、排空并按顺序重新引导，以应

用操作系统更新以及 control plane Operator 更新。为了避免级联失败，请将 control plane 节点上的总体 CPU 和内存资源使用量保留为最多 60% 的所有可用容量，以处理资源使用量激增。相应地增加 control plane 节点上的 CPU 和内存，以避免因为缺少资源而造成潜在的停机。



重要

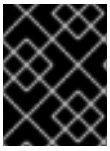
节点大小取决于集群中的节点和对象数量。它还取决于集群上是否正在主动创建这些对象。在创建对象时，control plane 在资源使用量方面与对象处于**运行（running）**阶段的时间相比更活跃。

Operator Lifecycle Manager（OLM）在 control plane 节点上运行，其内存占用量取决于 OLM 在集群中管理的命名空间和用户安装的 operator 的数量。Control plane 节点需要相应地调整大小，以避免 OOM 终止。以下数据基于集群最大测试的结果。

命名空间数量	处于空闲状态的 OLM 内存（GB）	安装了 5 个用户 operator 的 OLM 内存（GB）
500	0.823	1.7
1000	1.2	2.5
1500	1.7	3.2
2000	2	4.4
3000	2.7	5.6
4000	3.8	7.6
5000	4.2	9.02
6000	5.8	11.3
7000	6.6	12.9
8000	6.9	14.8
9000	8	17.7
10,000	9.9	21.6

重要

如果使用安装程序置备的基础架构安装方法，则无法修改正在运行的 OpenShift Container Platform 4.8 集群中的 control plane 节点大小。反之，您必须估计节点总数并在安装过程中使用推荐的 control plane 节点大小。



重要

建议基于在带有 OpenShiftSDN 作为网络插件的 OpenShift Container Platform 集群上捕获的数据点。

注意

在 OpenShift Container Platform 4.8 中，与 OpenShift Container Platform 3.11 及之前的版本相比，系统现在默认保留半个 CPU 内核（500 millicore）。确定大小时应该考虑这一点。

1.4.1. 增加 Amazon Web Services(AWS)master 实例的类别大小

当您在集群中过载 AWS master 节点并且 master 节点需要更多资源时，您可以增加 master 实例的类别文件大小。

注意

建议您在增大 AWS master 实例的类别大小前备份 etcd。

先决条件

- AWS 上有一个 IPI(安装程序置备的基础架构)或 UPI（用户置备的基础架构）集群。

流程

1. 打开 AWS 控制台，获取 master 实例。
2. 停止一个 master 实例。
3. 选择已停止的实例，然后点 **Actions → Instance Settings → Change instance type**。
4. 将实例更改为较大的类型，确保类型与之前选择相同，并应用更改。例如，您可以将 **m5.xlarge** 更改为 **m5.2xlarge** 或 **m5.4xlarge**。
5. 备份实例，再对下一个 master 实例重复上述步骤。

其他资源

- 备份 etcd

1.5. 推荐的 ETCD 实践

因为 etcd 将数据写入磁盘并在磁盘上持久化，所以其性能取决于磁盘性能。虽然 etcd 并不是有非常高的 I/O 负载，但它需要使用一个具有低延迟的块设备才能获得最佳性能和稳定性。因为 etcd 的共识协议依赖于将元数据永久存储到一个日志 (WAL)，所以 etcd 对磁盘的写延迟非常敏感。减慢来自其他进程的磁盘活动和磁盘活动可能会导致长时间的 fsync 延迟。

这些延迟可能会导致 etcd 丢失心跳，不会及时向磁盘提交新的建议，并最终遇到请求超时和临时丢失问题。高写入延迟也会导致 OpenShift API 较慢，这会影响集群性能。因此，需要避免在 control-plane 节点运行其他工作负载。

就延迟而言，应该在一个可最少以 50 IOPS 按顺序写入 8000 字节的块设备上运行。也就是说，当有一个 20ms 的延迟时，使用 fdatasync 来同步 WAL 中的写入操作。对于高负载的集群，建议使用 8000 字节的连续 500 IOPS (2 毫秒)。要测量这些数字，您可以使用基准测试工具，如 fio。

要实现这样的性能，在由低延迟和高吞吐量的 SSD 或 NVMe 磁盘支持的机器上运行 etcd。考虑使用单层单元(SLC)固态驱动器(SSD)（它为每个内存单元提供 1 位），这是可靠的，非常适合于写密集型工作负载。

以下硬盘功能提供最佳的 etcd 性能：

- 支持快速读取操作的低延迟。
- 高带宽写入，实现更快速的压缩和碎片处理。
- 高带宽读取，加快从故障恢复。
- 固态硬盘是最小的选择，NVMe 驱动器是首选的。
- 来自不同制造商的服务器级硬件可以提高可靠性。
- RAID 0 技术以提高性能。
- 专用 etcd 驱动器。不要将日志文件或其他重重工作负载放在 etcd 驱动器中。

避免 NAS 或 SAN 设置，以及旋转驱动器。始终使用相关工具（如 fio）进行基准测试。在增加时，持续监控集群性能。

注意

避免使用网络文件系统 (NFS) 协议或其他基于网络的文件系统。

需要在部署的 OpenShift Container Platform 集群上监控的一些关键指标包括，日志持续时间之前的 etcd 磁盘写入的 p99 值，以及 etcd leader 更改的数量。使用 Prometheus 跟踪这些指标。

要在创建 OpenShift Container Platform 集群之前或之后验证 etcd 的硬件，您可以使用 fio。

先决条件

- 您正在测试的机器上安装了 Podman 或 Docker 等容器运行时。
- 数据被写入 **/var/lib/etcd** 路径。

流程

- 运行 fio 并分析结果：
  - 如果使用 Podman，请运行以下命令：

```
$ sudo podman run --volume /var/lib/etcd:/var/lib/etcd:Z quay.io/openshift-scale/etcd-perf
```

  - 如果使用 Docker，请运行以下命令：

```
$ sudo docker run --volume /var/lib/etcd:/var/lib/etcd:Z quay.io/openshift-scale/etcd-perf
```

输出会报告磁盘是否足够快以运行 etcd，它会检查测试运行中获得的 fsync 指标的 p99 值是否小于 20ms。一些最重要的 etcd 指标可能受到 I/O 性能的影响，如下所示：

- **etcd_disk_wal_fsync_duration_seconds_bucket** 指标报告了 etcd 的 WAL fsync 持续时间。
- **etcd_disk_backend_commit_duration_seconds_bucket** 指标报告 etcd 后端提交延迟持续时间
- **etcd_server_leader_changes_seen_total** 指标报告领导更改

etcd 在所有成员间复制请求，因此其性能会严重依赖于网络输入/输出（I/O）的延迟。大量网络延迟会导致 etcd heartbeat 的时间比选举超时时间更长，这会导致一个可能会对集群造成破坏的领导选举。在部署的 OpenShift Container Platform 集群上监控的一个关键指标是每个 etcd 集群成员上的 etcd 网络对延迟的 p99 百分比。使用 Prometheus 跟踪指标数据。

histogram_quantile（0.99, rate(etcd_network_peer_round_trip_time_seconds_bucket[2m]) 指标报告 etcd 在成员间复制客户端请求的时间。确保它小于 50 ms。

1.6. 分离 ETCD 数据

对于大型、高密度的集群，如果键空间增长过大并超过空间配额，etcd 的性能将会受到影响。定期维护并处理碎片化的 etcd，以释放数据存储中的空间。监控 Prometheus 以了解 etcd 指标数据，并在需要时对其进行碎片处理；否则，etcd 可能会引发一个集群范围的警报，使集群进入维护模式，仅能接受对键的读和删除操作。

监控这些关键指标：

- **etcd_server_quota_backend_bytes**，这是当前配额限制
- **etcd_mvcc_db_total_size_in_use_in_bytes**，表示历史压缩后实际数据库使用量
- **etcd_debugging_mvcc_db_total_size_in_bytes** 会显示数据库大小，包括等待碎片整理的空闲空间

在导致磁盘碎片的事件后（如 etcd 历史记录紧凑）对 etcd 数据进行清理以回收磁盘空间。

历史压缩将自动每五分钟执行一次，并在后端数据库中造成混乱。此碎片空间可供 etcd 使用，但主机文件系统不可用。您必须对碎片 etcd 进行碎片清除，才能使这个空间可供主机文件系统使用。

因为 etcd 将数据写入磁盘，所以其性能主要取决于磁盘性能。根据您的集群的具体情况，考虑每个月清理一次 etcd 碎片，或每个月清理两次。您还可以监控 **etcd_db_total_size_in_bytes** 指标，以确定是否需要进行碎片操作。

您还可以通过检查 etcd 数据库大小（MB）来决定是否需要进行碎片整理。通过 PromQL 表达 **(etcd_mvcc_db_total_size_in_bytes - etcd_mvcc_db_total_size_in_use_in_bytes)/1024/1024** 来释放空间。

警告

分离 etcd 是一个阻止性操作。在进行碎片处理完成前，etcd 成员不会响应。因此，在每个下一个 pod 要进行碎片清理前，至少等待一分钟，以便集群可以恢复正常工作。

按照以下步骤对每个 etcd 成员上的 etcd 数据进行碎片处理。

先决条件

- 您可以使用具有 **cluster-admin** 角色的用户访问集群。

流程

1. 确定哪个 etcd 成员是领导成员，因为领导会进行最后的碎片处理。
   a. 获取 etcd pod 列表：

```
$ oc get pods -n openshift-etcd -o wide | grep -v quorum-guard | grep etcd
```

   输出示例

```
etcd-ip-10-0-159-225.example.redhat.com              3/3     Running     0          175m
10.0.159.225   ip-10-0-159-225.example.redhat.com   <none>           <none>
etcd-ip-10-0-191-37.example.redhat.com               3/3     Running     0          173m
10.0.191.37    ip-10-0-191-37.example.redhat.com    <none>           <none>
etcd-ip-10-0-199-170.example.redhat.com              3/3     Running     0          176m
10.0.199.170   ip-10-0-199-170.example.redhat.com   <none>           <none>
```

   b. 选择 pod 并运行以下命令来确定哪个 etcd 成员是领导：

```
$ oc rsh -n openshift-etcd etcd-ip-10-0-159-225.example.redhat.com etcdctl endpoint
status --cluster -w table
```

   输出示例

```
Defaulting container name to etcdctl.
Use 'oc describe pod/etcd-ip-10-0-159-225.example.redhat.com -n openshift-etcd' to see
all of the containers in this pod.
+---------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
|         ENDPOINT          |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER |
RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+---------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
|  https://10.0.191.37:2379 | 251cd44483d811c3 |   3.4.9 |  104 MB |     false |      false |
7 |      91624 |              91624 |        |
| https://10.0.159.225:2379 | 264c7c58ecbdabee |   3.4.9 |  104 MB |     false |      false |
7 |      91624 |              91624 |        |
| https://10.0.199.170:2379 | 9ac311f93915cc79 |   3.4.9 |  104 MB |      true |      false |
7 |      91624 |              91624 |        |
+---------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
```

   基于此输出的 **IS LEADER** 列，**https://10.0.199.170:2379** 端点是领导。与上一步输出匹配此端点，领导的 pod 名称为 **etcd-ip-10-0-199-170.example.redhat.com**。

2. 清理 etcd 成员。
   a. 连接到正在运行的 etcd 容器，传递 *不是* 领导的 pod 的名称：

```
$ oc rsh -n openshift-etcd etcd-ip-10-0-159-225.example.redhat.com
```

b. 取消设置 **ETCDCTL_ENDPOINTS** 环境变量：

```
sh-4.4# unset ETCDCTL_ENDPOINTS
```

c. 清理 etcd 成员：

```
sh-4.4# etcdctl --command-timeout=30s --endpoints=https://localhost:2379 defrag
```

输出示例

```
Finished defragmenting etcd member[https://localhost:2379]
```

如果发生超时错误，增加 **--command-timeout** 的值，直到命令成功为止。

d. 验证数据库大小是否已缩小：

```
sh-4.4# etcdctl endpoint status -w table --cluster
```

输出示例

```
+--------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
|        ENDPOINT          |        ID        | VERSION | DB SIZE | IS LEADER | IS LEARNER |
RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
+--------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
| https://10.0.191.37:2379 | 251cd44483d811c3 |   3.4.9 |  104 MB |     false |      false |
7 |      91624 |              91624 |        |
| https://10.0.159.225:2379 | 264c7c58ecbdabee |   3.4.9 |   41 MB |     false |      false |
7 |      91624 |              91624 |        | 1
| https://10.0.199.170:2379 | 9ac311f93915cc79 |   3.4.9 |  104 MB |      true |      false |
7 |      91624 |              91624 |        |
+--------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+--------------------+--------+
```

本例显示这个 etcd 成员的数据库大小现在为 41 MB，而起始大小为 104 MB。

e. 重复这些步骤以连接到其他 etcd 成员并进行碎片处理。最后才对领导进行碎片清除。至少要在碎片处理操作之间等待一分钟，以便 etcd pod 可以恢复。在 etcd pod 恢复前，etcd 成员不会响应。

3. 如果因为超过空间配额而触发任何 **NOSPACE** 警告，请清除它们。

a. 检查是否有 **NOSPACE** 警告：

```
sh-4.4# etcdctl alarm list
```

输出示例

```
memberID:12345678912345678912 alarm:NOSPACE
```

b. 清除警告：

```
sh-4.4# etcdctl alarm disarm
```

1.7. OPENSHIFT CONTAINER PLATFORM 基础架构组件

以下基础架构工作负载不会导致 OpenShift Container Platform worker 订阅：

- 在主控机上运行的 Kubernetes 和 OpenShift Container Platform control plane 服务
- 默认路由器
- 集成的容器镜像 registry
- 基于 HAProxy 的 Ingress Controller
- 集群指标集合或监控服务，包括监控用户定义的项目的组件
- 集群聚合日志
- 服务代理
- Red Hat Quay
- Red Hat OpenShift Container Storage
- Red Hat Advanced Cluster Manager
- Red Hat Advanced Cluster Security for Kubernetes
- Red Hat OpenShift GitOps
- Red Hat OpenShift Pipelines

运行任何其他容器、Pod 或组件的所有节点都需要是您的订阅可涵盖的 worker 节点。

其他资源

- 有关基础架构节点以及可在基础架构节点上运行，请参阅 OpenShift sizing and subscription guide for enterprise Kubernetes 文档中的 "Red Hat OpenShift control plane and infrastructure nodes"部分。

1.8. 移动监控解决方案

监控堆栈包含多个组件，包括 Prometheus、Grafana 和 Alertmanager。Cluster Monitoring Operator 管理此堆栈。要将监控堆栈重新部署到基础架构节点，您可以创建并应用自定义配置映射。

流程

1. 编辑 **cluster-monitoring-config** 配置映射，并更改 **nodeSelector** 以使用 **infra** 标签：

```
$ oc edit configmap cluster-monitoring-config -n openshift-monitoring
```

```
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |+
    alertmanagerMain:
      nodeSelector: (1)
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    prometheusK8s:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    prometheusOperator:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    grafana:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    k8sPrometheusAdapter:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
```

```
    kubeStateMetrics:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    telemeterClient:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    openshiftStateMetrics:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
    thanosQuerier:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      tolerations:
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoSchedule
      - key: node-role.kubernetes.io/infra
        value: reserved
        effect: NoExecute
```

1 1 添加 **nodeSelector** 参数，并设为适用于您想要移动的组件的值。您可以根据为节点指定的值，按所示格式使用 **nodeSelector** 或使用 **<key>: <value>** 对。如果您在 infrasructure 节点中添加了污点，还要添加匹配的容限。

2. 观察监控 pod 移至新机器：

```
$ watch 'oc get pod -n openshift-monitoring -o wide'
```

3. 如果组件没有移到 **infra** 节点，请删除带有这个组件的 pod:

```
$ oc delete pod -n openshift-monitoring <pod>
```

已删除 pod 的组件在 **infra** 节点上重新创建。

1.9. 移动默认 REGISTRY

您需要配置 registry Operator，以便将其 Pod 部署到其他节点。

先决条件

- 在 OpenShift Container Platform 集群中配置额外的机器集。

流程

1. 查看 **config/instance** 对象：

```
$ oc get configs.imageregistry.operator.openshift.io/cluster -o yaml
```

输出示例

```
apiVersion: imageregistry.operator.openshift.io/v1
kind: Config
metadata:
  creationTimestamp: 2019-02-05T13:52:05Z
  finalizers:
  - imageregistry.operator.openshift.io/finalizer
  generation: 1
  name: cluster
  resourceVersion: "56174"
  selfLink: /apis/imageregistry.operator.openshift.io/v1/configs/cluster
  uid: 36fd3724-294d-11e9-a524-12ffeee2931b
spec:
  httpSecret: d9a012ccd117b1e6616ceccb2c3bb66a5fed1b5e481623
  logging: 2
  managementState: Managed
  proxy: {}
  replicas: 1
  requests:
    read: {}
    write: {}
  storage:
    s3:
      bucket: image-registry-us-east-1-c92e88cad85b48ec8b312344dff03c82-392c
      region: us-east-1
status:
...
```

2. 编辑 **config/instance** 对象：

```
$ oc edit configs.imageregistry.operator.openshift.io/cluster
```

```
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          namespaces:
          - openshift-image-registry
```

```
        topologyKey: kubernetes.io/hostname
      weight: 100
logLevel: Normal
managementState: Managed
nodeSelector: (1)
  node-role.kubernetes.io/infra: ""
tolerations:
- effect: NoSchedule
  key: node-role.kubernetes.io/infra
  value: reserved
- effect: NoExecute
  key: node-role.kubernetes.io/infra
  value: reserved
```

(1) 添加 **nodeSelector** 参数，并设为适用于您想要移动的组件的值。您可以根据为节点指定的值，按所示格式使用 **nodeSelector** 或使用 **<key>: <value>** 对。如果您在 infrasructure 节点中添加了污点，还要添加匹配的容限。

3. 验证 registry pod 已移至基础架构节点。

   a. 运行以下命令，以识别 registry pod 所在的节点：

   ```
   $ oc get pods -o wide -n openshift-image-registry
   ```

   b. 确认节点具有您指定的标签：

   ```
   $ oc describe node <node_name>
   ```

   查看命令输出，并确认 **node-role.kubernetes.io/infra** 列在 **LABELS** 列表中。

1.10. 移动路由器

您可以将路由器 pod 部署到不同的机器集中。默认情况下，pod 部署到 worker 节点。

先决条件

- 在 OpenShift Container Platform 集群中配置额外的机器集。

流程

1. 查看路由器 Operator 的 **IngressController** 自定义资源：

   ```
   $ oc get ingresscontroller default -n openshift-ingress-operator -o yaml
   ```

   命令输出类似于以下文本：

   ```
   apiVersion: operator.openshift.io/v1
   kind: IngressController
   metadata:
     creationTimestamp: 2019-04-18T12:35:39Z
     finalizers:
     - ingresscontroller.operator.openshift.io/finalizer-ingresscontroller
     generation: 1
   ```

```
  name: default
  namespace: openshift-ingress-operator
  resourceVersion: "11341"
  selfLink: /apis/operator.openshift.io/v1/namespaces/openshift-ingress-
operator/ingresscontrollers/default
  uid: 79509e05-61d6-11e9-bc55-02ce4781844a
spec: {}
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2019-04-18T12:36:15Z
    status: "True"
    type: Available
  domain: apps.<cluster>.example.com
  endpointPublishingStrategy:
    type: LoadBalancerService
  selector: ingresscontroller.operator.openshift.io/deployment-ingresscontroller=default
```

2. 编辑 **ingresscontroller** 资源，并更改 **nodeSelector** 以使用**infra** 标签：

```
$ oc edit ingresscontroller default -n openshift-ingress-operator
```

```
spec:
  nodePlacement:
    nodeSelector: (1)
      matchLabels:
        node-role.kubernetes.io/infra: ""
    tolerations:
    - effect: NoSchedule
      key: node-role.kubernetes.io/infra
      value: reserved
    - effect: NoExecute
      key: node-role.kubernetes.io/infra
      value: reserved
```

(1) 添加 **nodeSelector** 参数，并设为适用于您想要移动的组件的值。您可以根据为节点指定的值，按所示格式使用 **nodeSelector** 或使用 **<key>: <value>** 对。如果您在 infrasructure 节点中添加了污点，还要添加匹配的容限。

3. 确认路由器 Pod 在 **infra** 节点上运行。

   a. 查看路由器 Pod 列表，并记下正在运行的 Pod 的节点名称：

```
$ oc get pod -n openshift-ingress -o wide
```

输出示例

```
NAME                              READY   STATUS        RESTARTS   AGE     IP           NODE
NOMINATED NODE   READINESS GATES
router-default-86798b4b5d-bdlvd   1/1     Running       0          28s     10.130.2.4   ip-10-
0-217-226.ec2.internal   <none>           <none>
router-default-955d875f4-255g8    0/1     Terminating   0          19h     10.129.2.4   ip-10-
0-148-172.ec2.internal   <none>           <none>
```

在本例中，正在运行的 Pod 位于 **ip-10-0-217-226.ec2.internal** 节点上。

b. 查看正在运行的 Pod 的节点状态：

```
$ oc get node <node_name> 1
```

1 指定从 Pod 列表获得的 **<node_name>**。

输出示例

```
NAME                           STATUS  ROLES        AGE   VERSION
ip-10-0-217-226.ec2.internal   Ready   infra,worker 17h   v1.21.0
```

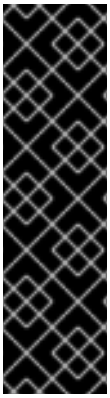
由于角色列表包含 **infra**，因此 Pod 在正确的节点上运行。

1.11. 基础架构节点大小

*基础架构节点*是标记为运行 OpenShift Container Platform 环境组成部分的节点。基础架构节点的资源要求取决于集群中的集群年龄、节点和对象，因为这些因素可能会导致 Prometheus 的指标或时间序列增加。以下推荐的基础架构节点大小是基于集群最大值和 control plane 密度测试的结果。

worker 节点数量	CPU 内核	内存 (GB)
25	4	16
100	8	32
250	16	128
500	32	128

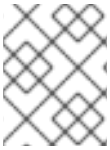
通常，建议每个集群有三个基础架构节点。



重要

这些大小建议基于缩放测试，该测试可在整个集群中创建大量对象。这些测试包括达到一些集群最大值。在 OpenShift Container Platform 4.8 集群中有 250 个和 500 个节点时，这些最大值为 10000 个命名空间，包含 61000 个 pod、10000 个 部署、181000 个 secret、400 个配置映射等。Prometheus 是一个高内存密集型应用程序，资源使用量取决于各种因素，包括节点、对象、Prometheus 指标提取间隔、指标或时间序列以及集群的年龄。磁盘大小还取决于保留周期。您必须考虑以上因素并相应地调整它们的大小。

建议的大小只适用于在集群安装过程中安装的 Prometheus、Router 和 Registry 基础架构组件。日志记录（Logging）是第二天操作，因此没有包含在这些建议中。



注意

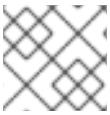
在 OpenShift Container Platform 4.8 中，与 OpenShift Container Platform 3.11 及之前的版本相比，系统现在默认保留半个 CPU 内核（500 millicore）。这会影响缩放建议。

1.12. 其他资源

- OpenShift Container Platform 集群限制
- 创建基础架构机器集

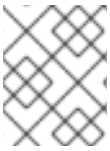
第 2 章 IBM Z 和 LINUXONE 环境的推荐主机实践

本节为 IBM Z 和 LinuxONE 上的 OpenShift Container Platform 提供推荐的主机实践。



注意

s390x 架构在很多方面都是唯一的。因此，此处提出的一些建议可能不适用于其他平台。



注意

除非另有说明，否则这些实践适用于 IBM Z 和 LinuxONE 上的 z/VM 和 Red Hat Enterprise Linux (RHEL) KVM 安装。

2.1. 管理 CPU 过量使用

在高度虚拟化的 IBM Z 环境中，您必须仔细规划基础架构的设置和大小。虚拟化最重要的功能之一是能够进行资源过量使用，从而将更多资源分配给虚拟机，而不是在管理程序级别实际可用。这主要依赖于具体的工作负载，并没有适用于所有环境的“黄金法则”。

根据您的设置，在设计 CPU 过量使用 时请考虑这些最佳实践：

- 在 LPAR 级别 (PR/SM hypervisor)，避免将所有可用物理内核 (IFL) 分配给每个 LPAR。例如，当有四个物理 IFL 可用时，您不应该定义三个 LPAR，每个都带有四个逻辑 IFL。
- 检查并了解 LPAR 共享和权重.
- 过多的虚拟 CPU 会对性能造成负面影响。不要将比逻辑处理器定义为 LPAR 更多的虚拟处理器。
- 为峰值工作负载配置每个客户机的虚拟处理器数量，而不是配置更多.
- 从一个小的数量开始，并监控工作负载。如有必要，逐步增加 vCPU 数量。
- 并非所有工作负载都适合适用高过量使用比率。如果工作负载是 CPU 密集型的，那么您可能无法在不对性能造成影响的情况下使用高的比率。对于高 I/O 密集型工作负载，即便具有较高的过量使用比率，也能保持一致的性能。

其他资源

- Z/VM 通用性能问题和解决方案
- Z/VM 过量使用注意事项
- LPAR CPU 管理

2.2. 禁用透明巨页

Transparent Huge Pages (THP) 会试图自动执行创建、管理和使用巨页的大部分方面。由于 THP 自动管理巨页，因此并不始终对所有类型的工作负载进行最佳处理。THP 可能会导致性能下降，因为许多应用程序都自行处理巨页。因此，请考虑禁用 THP。

2.3. 使用 RECEIVE FLOW STEERING（RFS）提高网络性能

通过进一步减少网络延迟，Receive Flow Steering (RFS) 进一步扩展了 Receive Packet Steering

(RPS)。RFS 在技术上基于 RPS，通过增加 CPU 缓存命中率来提高数据包处理的效率。RFS 通过确定计算最方便的 CPU，以便缓存命中更有可能在 CPU 中发生，增加了对队列长度的考虑。因此，会减少 CPU 缓存无效的频率，从而只需要较少的循环来重建缓存。这有助于缩短数据包处理运行时间。

2.3.1. 使用 Machine Config Operator (MCO) 激活 RFS

流程

1. 将以下 MCO 示例配置集复制到 YAML 文件中。例如，**enable-rfs.yaml**：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 50-enable-rfs
spec:
  config:
    ignition:
      version: 2.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=US-
ASCII,%23%20turn%20on%20Receive%20Flow%20Steering%20%28RFS%29%20for%20all
%20network%20interfaces%0ASUBSYSTEM%3D%3D%22net%22%2C%20ACTION%3D%
3D%22add%22%2C%20RUN%7Bprogram%7D%2B%3D%22/bin/bash%20-
c%20%27for%20x%20in%20/sys/%24DEVPATH/queues/rx-
%2A%3B%20do%20echo%208192%20%3E%20%24x/rps_flow_cnt%3B%20%20done%27
%22%0A
        filesystem: root
        mode: 0644
        path: /etc/udev/rules.d/70-persistent-net.rules
      - contents:
          source: data:text/plain;charset=US-
ASCII,%23%20define%20sock%20flow%20enbtried%20for%20%20Receive%20Flow%20Ste
ering%20%28RFS%29%0Anet.core.rps_sock_flow_entries%3D8192%0A
        filesystem: root
        mode: 0644
        path: /etc/sysctl.d/95-enable-rps.conf
```

2. 创建 MCO 配置集：

```
$ oc create -f enable-rfs.yaml
```

3. 验证是否列出了名为 **50-enable-rfs** 的条目：

```
$ oc get mc
```

4. 要取消激活，请输入：

```
$ oc delete mc 50-enable-rfs
```

其他资源

- IBM Z 上的 OpenShift Container Platform：使用 RFS 调整您的网络性能
- 配置 Receive Flow Steering (RFS)
- 在 Linux 网络堆栈中扩展

2.4. 选择您的网络设置

网络堆栈是 OpenShift Container Platform 等基于 Kubernetes 的产品最重要的组件之一。对于 IBM Z 设置，网络设置取决于您选择的虚拟机监控程序。取决于具体的工作负载和应用，最佳实践通常需要根据用例和流量模式进行更改。

根据您的设置，考虑以下最佳实践：

- 考虑有关网络设备的所有选项，以优化您的流量模式。探索 OSA-Express、RoCE Express、HiperSockets、z/VM VSwitch、Linux 网桥 (KVM) 的优势，以确定哪个选项为您的设置带来最大好处。
- 始终使用最新可用的 NIC 版本。例如，OSA Express 7S 10 GbE 与带有事务工作负载类型的 OSA Express 6S 10 GbE 相比有显著改进，尽管两者都是 10 GbE 适配器。
- 每个虚拟交换机都添加了额外的延迟层。
- 负载平衡器在集群外的网络通信中扮演重要角色。如果这对应用程序至关重要，请考虑使用生产环境级的硬件负载平衡器。
- OpenShift Container Platform SDN 引入了影响网络性能的流程和规则。确保对 pod 关联性和放置进行考虑，以便至关重要的服务会受益于本地通信的优势。
- 平衡性能和功能之间的权衡.

其他资源

- IBM Z 上的 OpenShift Container Platform - 性能体验、小窍门和经验
- IBM Z 网络中的 OpenShift Container Platform 性能
- 使用节点关联性规则控制节点上的 pod 放置

2.5. 确保 Z/VM 上使用 HYPERPAV 的高磁盘性能

DASD 和 ECKD 设备在 IBM Z 环境中通常使用磁盘类型。在 z/VM 环境中的典型 OpenShift Container Platform 设置中，DASD 磁盘通常用于支持节点的本地存储。您可以设置 HyperPAV 别名设备，以便为支持 z/VM 客户机的 DASD 磁盘提供更多吞吐量和总体更好的 I/O 性能。

将 HyperPAV 用于本地存储设备可带来显著的性能优势。但是，您必须考虑吞吐量和 CPU 成本之间有一个权衡。

2.5.1. 使用 Machine Config Operator (MCO) 在使用 z/VM full-pack minidisks 的节点中激活 HyperPAV 别名

对于使用 full-pack minidisk 的基于 z/VM 的 OpenShift Container Platform 设置，您可以通过在所有节点中激活 HyperPAV 别名来利用 MCO 配置集的优势。您必须为 control plane 和计算节点添加 YAML 配置。

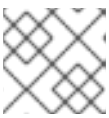
流程

1. 将以下 MCO 示例配置集复制到 control plane 节点的 YAML 文件中。例如，**05-master-kernelarg-hpav.yaml**:

```
$ cat 05-master-kernelarg-hpav.yaml
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: master
  name: 05-master-kernelarg-hpav
spec:
  config:
    ignition:
      version: 3.1.0
  kernelArguments:
    - rd.dasd=800-805
```

2. 将以下 MCO 示例配置集复制到计算节点的 YAML 文件中。例如，**05-worker-kernelarg-hpav.yaml**:

```
$ cat 05-worker-kernelarg-hpav.yaml
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: 05-worker-kernelarg-hpav
spec:
  config:
    ignition:
      version: 3.1.0
  kernelArguments:
    - rd.dasd=800-805
```



注意

您必须修改 **rd.dasd** 参数以适合设备 ID。

3. 创建 MCO 配置集：

```
$ oc create -f 05-master-kernelarg-hpav.yaml
```

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

4. 要取消激活，请输入：

```
$ oc delete -f 05-master-kernelarg-hpav.yaml
```

```
$ oc delete -f 05-worker-kernelarg-hpav.yaml
```

其他资源

- 为 ECKD DASD 使用 HyperPAV
- 在 z/VM 上的 Linux guest 上扩展 HyperPAV 别名设备

2.6. IBM Z 主机上的 RHEL KVM 建议

优化 KVM 虚拟服务器环境很大程度上取决于虚拟服务器的工作负载和可用资源。增强一个环境中性能的相同操作可能会对另一种环境产生负面影响。为特定设置找到最佳平衡可能是一项挑战，通常需要进行各种试验。

下面的部分论述了在 IBM Z 和 LinuxONE 环境中将 OpenShift Container Platform 与 RHEL KVM 搭配使用时的一些最佳实践。

2.6.1. 将多个队列用于您的 VirtIO 网络接口

使用多个虚拟 CPU 时，如果您为传入和传出数据包提供多个队列，则可以并行传输软件包。使用 **driver** 元素的 **queues** 属性来配置多个队列。指定一个最小为 2 的整数，该整数不超过虚拟服务器的虚拟 CPU 数量。

以下示例规格为网络接口配置两个输入和输出队列：

```
<interface type="direct">
   <source network="net01"/>
   <model type="virtio"/>
   <driver ... queues="2"/>
</interface>
```

多个队列旨在为网络接口提供增强的性能，但也使用内存和 CPU 资源。首先为繁忙的接口定义两个队列。接下来，尝试为不太繁忙的接口定义两个队列，为繁忙的接口设置两个以上的队列。

2.6.2. 对虚拟块设备使用 I/O 线程

要使虚拟块设备使用 I/O 线程，您必须为虚拟服务器和每个虚拟块设备配置一个或多个 I/O 线程，以使用其中一个 I/O 线程。

以下示例指定了 **<iothreads>3</iothreads>** 来配置三个 I/O 线程，带有连续十进制线程 ID 1、2 和 3。**iothread="2"** 参数指定要使用 ID 为 2 的 I/O 线程的磁盘设备的驱动程序元素。

I/O 线程规格示例

```
...
<domain>
  <iothreads>3</iothreads> 1
    ...
     <devices>
       ...
          <disk type="block" device="disk"> 2
<driver ... iothread="2"/>
    </disk>
```

```
    ...
  </devices>
  ...
</domain>
```

1 I/O 线程的数量。

2 磁盘设备的驱动程序元素。

线程可以提高磁盘设备的 I/O 操作性能，但也可使用内存和 CPU 资源。您可以将多个设备配置为使用同一线程。线程到设备的最佳映射取决于可用资源和工作负载。

从少量 I/O 线程开始。通常，为所有磁盘设备使用单个 I/O 线程就足够了。不要配置超过虚拟 CPU 数量的线程，也不要配置空闲线程。

您可以使用 **virsh iothreadadd** 命令将具有特定线程 ID 的 I/O 线程添加到正在运行的虚拟服务器。

2.6.3. 避免虚拟 SCSI 设备

仅在需要通过 SCSI 特定的接口解决设备时配置虚拟 SCSI 设备。将磁盘空间配置为虚拟块设备，而非虚拟 SCSI 设备，无论主机上的支持是什么。

但是，您可能需要以下特定于 SCSI 的接口：

- 主机上 SCSI 附加磁带驱动器的 LUN。
- 在主机文件系统中挂载在虚拟 DVD 驱动器中的 DVD ISO 文件。

2.6.4. 为磁盘配置客户机缓存

将磁盘设备配置为由客户机而不是主机执行缓存。

确保磁盘设备的 driver 元素包含 **cache="none"** 和 **io="native"** 参数。

```
<disk type="block" device="disk">
  <driver name="qemu" type="raw" cache="none" io="native" iothread="1"/>
...
</disk>
```

2.6.5. 排除内存气球（Balloon）设备

除非您需要动态内存大小，否则请不要定义内存气球设备，并确保 libvirt 不会为您创建。将 **memballoon** 参数作为设备元素的子项包含在您的域配置 XML 文件中。

- 检查活跃配置集列表：

```
<memballoon model="none"/>
```

2.6.6. 调整主机调度程序的 CPU 迁移算法

重要

除非您非常了解相关的影响，请不要更改调度程序设置。在进行完整的测试并确定相关的影响前，不要对生产系统应用更改。

kernel.sched_migration_cost_ns 参数指定以纳秒为单位的时间间隔。任务最后一次执行后，CPU 缓存被视为具有有用内容，直到此间隔过期为止。增加这个间隔会导致任务迁移减少。默认值为 500000 ns。

如果存在可运行的进程时 CPU 空闲时间高于预期的间隔，请尝试缩短这个间隔。如果任务非常频繁地在 CPU 或节点之间进行转换，请尝试增加它。

要动态将间隔设置为 60000 ns，请输入以下命令：

```
# sysctl kernel.sched_migration_cost_ns=60000
```

要将值永久更改为 60000 ns，在 **/etc/sysctl.conf** 中添加以下条目：

```
kernel.sched_migration_cost_ns=60000
```

2.6.7. 禁用 cpuset cgroup 控制器

注意

此设置仅适用于使用 cgroups 版本 1 的 KVM 主机。要在主机上启用 CPU 热插拔，请禁用 cgroup 控制器。

流程

1. 使用您选择的编辑器打开 **/etc/libvirt/qemu.conf**。
2. 转至 **cgroup_controllers** 行。
3. 复制整行并从副本中删除前导编号符号(#)。
4. 删除 **cpuset** 条目，如下所示：

   ```
   cgroup_controllers = [ "cpu", "devices", "memory", "blkio", "cpuacct" ]
   ```

5. 要使新设置生效，您必须重启 libvirtd 守护进程：
   a. 停止所有虚拟机。
   b. 运行以下命令:

      ```
      # systemctl restart libvirtd
      ```

   c. 重新启动虚拟机。

此设置在主机重新引导后保留。

2.6.8. 为空闲的虚拟 CPU 调整轮询周期

当虚拟 CPU 空闲时，KVM 会轮询虚拟 CPU 的唤醒条件，然后再分配主机资源。您可以指定时间间隔，在间隔期间在 **/sys/module/kvm/parameters/halt_poll_ns** 的 sysfs 中进行轮询。在指定时间内，轮询可减少虚拟 CPU 的唤醒延迟，但会牺牲资源使用量。根据工作负载，更长或更短的轮询时间可能很有用。时间间隔以纳秒为单位指定。默认值为 50000 ns。

- 要针对低 CPU 消耗进行优化，请输入一个小的值或写入 0 来禁用轮询：

```
# echo 0 > /sys/module/kvm/parameters/halt_poll_ns
```

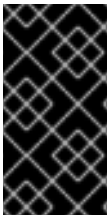
- 要针对低延迟进行优化（例如，用于事务的工作负载），请输入一个大的值：

```
# echo 80000 > /sys/module/kvm/parameters/halt_poll_ns
```

其他资源

- Linux on IBM Z 性能调节 KVM
- 在 IBM Z 中使用虚拟化

第 3 章 推荐的集群扩展实践



重要

本节中的指导信息仅与使用云供应商集成的安装相关。

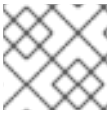
这些指南适用于带有软件定义网络（SDN）而不是开放虚拟网络（OVN）的 OpenShift Container Platform。

应用以下最佳实践来扩展 OpenShift Container Platform 集群中的 worker 机器数量。您可以通过增加或减少 worker MachineSet 中定义的副本数量来扩展 worker 机器集。

3.1. 扩展集群的建议实践

将集群扩展到具有更多节点时：

- 将节点分散到所有可用区以获得更高的可用性。
- 同时扩展的机器数量不要超过 25 到 50 个。
- 考虑在每个可用区创建一个具有类似大小的替代实例类型的新机器集，以帮助缓解周期性供应商容量限制。例如，在 AWS 上，使用 m5.large 和 m5d.large。



注意

云供应商可能会为 API 服务实施配额。因此，需要对集群逐渐进行扩展。

如果同时将机器集中的副本设置为更高数量，则控制器可能无法创建机器。部署 OpenShift Container Platform 的云平台可以处理的请求数量将会影响该进程。当尝试创建、检查和更新有状态的机器时，控制器会开始进行更多的查询。部署 OpenShift Container Platform 的云平台具有 API 请求限制，如果出现过量查询，则可能会因为云平台的限制而导致机器创建失败。

当扩展到具有大量节点时，启用机器健康检查。如果出现故障，健康检查会监控状况并自动修复不健康的机器。

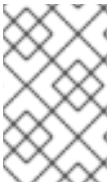
注意

当对大型且高密度的集群减少节点数时，可能需要大量时间，因为这个过程涉及排空或驱除在同时终止的节点上运行的对象。另外，如果要驱除的对象太多，对客户端的请求处理会出现瓶颈。目前将默认的客户端 QPS 和 burst 率分别设定为 **5** 和 **10**，且无法在 OpenShift Container Platform 中进行修改。

3.2. 修改机器集

要更改机器集，编辑 **MachineSet** YAML。然后，通过删除每台机器或将机器设置为 **0** 个副本来删除与机器设置关联的所有机器。然后，将副本数量调回所需的数量。您对机器集所做的更改不会影响现有的机器。

如果您需要在不进行其他更改的情况下扩展机器集，则不需要删除机器。



注意

默认情况下，OpenShift Container Platform 路由器 Pod 部署在 worker 上。由于路由器需要访问某些集群资源（包括 Web 控制台），除非先重新放置了路由器 Pod，否则请不要将 worker 机器集扩展为 **0**。

先决条件

- 安装 OpenShift Container Platform 集群和 **oc** 命令行。
- 以具有 **cluster-admin** 权限的用户身份登录 **oc**。

流程

1. 编辑机器集：

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

2. 将机器缩减为 **0**:

```
$ oc scale --replicas=0 machineset <machineset> -n openshift-machine-api
```

或者：

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

提示

您还可以应用以下 YAML 来扩展机器集：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  replicas: 0
```

等待机器被删除。

3. 根据需要扩展机器设置：

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

或者：

```
$ oc edit machineset <machineset> -n openshift-machine-api
```

提示

您还可以应用以下 YAML 来扩展机器集：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineSet
metadata:
  name: <machineset>
  namespace: openshift-machine-api
spec:
  replicas: 2
```

等待机器启动。新机器包含您对机器集所做的更改。

3.3. 关于机器健康检查

机器健康检查自动修复特定机器池中不健康的机器。

要监控机器的健康状况，创建资源来定义控制器的配置。设置要检查的条件（例如，处于 **NotReady** 状态达到五分钟或 node-problem-detector 中显示了持久性状况），以及用于要监控的机器集合的标签。

注意

您不能对具有 master 角色的机器进行机器健康检查。

监控 **MachineHealthCheck** 资源的控制器会检查定义的条件。如果机器无法进行健康检查，则会自动删除机器并创建一个机器来代替它。删除机器之后，您会看到**机器被删除**事件。

为限制删除机器造成的破坏性影响，控制器一次仅清空并删除一个节点。如果目标机器池中不健康的机器池中不健康的机器数量大于 **maxUnhealthy** 的值，则补救会停止，需要启用手动干预。

注意

请根据工作负载和要求仔细考虑超时。

- 超时时间较长可能会导致不健康的机器上的工作负载长时间停机。
- 超时时间太短可能会导致补救循环。例如，检查 **NotReady** 状态的超时时间必须足够长，以便机器能够完成启动过程。

要停止检查，请删除资源。

例如，您应该在升级过程中停止检查，因为集群中的节点可能会临时不可用。**MachineHealthCheck** 可能会识别不健康的节点，并重新引导它们。为避免重新引导这样的节点，请在更新集群前删除您部署的任何 **MachineHealthCheck** 资源。但是，默认部署的 **MachineHealthCheck** 资源（如 **machine-api-termination-handler**）不能被删除并重新创建。

3.3.1. 部署机器健康检查时的限制

部署机器健康检查前需要考虑以下限制：

- 只有机器集拥有的机器才可以由机器健康检查修复。

- 目前不支持 control plane 机器，如果不健康，则不会被修复。
- 如果机器的节点从集群中移除，机器健康检查会认为机器不健康，并立即修复机器。
- 如果机器对应的节点在 **nodeStartupTimeout** 之后没有加入集群，则会修复机器。
- 如果 **Machine** 资源阶段为 **Failed**，则会立即修复机器。

3.4. MACHINEHEALTHCHECK 资源示例

所有基于云的安装类型的 **MachineHealthCheck** 资源，以及裸机以外的资源，类似以下 YAML 文件：

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineHealthCheck
metadata:
  name: example 1
  namespace: openshift-machine-api
spec:
  selector:
    matchLabels:
      machine.openshift.io/cluster-api-machine-role: <role> 2
      machine.openshift.io/cluster-api-machine-type: <role> 3
      machine.openshift.io/cluster-api-machineset: <cluster_name>-<label>-<zone> 4
  unhealthyConditions:
  - type:    "Ready"
    timeout: "300s" 5
    status: "False"
  - type:    "Ready"
    timeout: "300s" 6
    status: "Unknown"
  maxUnhealthy: "40%" 7
  nodeStartupTimeout: "10m" 8
```

1. 指定要部署的机器健康检查的名称。
2. 3. 为要检查的机器池指定一个标签。
4. 以 **<cluster_name>-<label>-<zone>** 格式 指定要跟踪的机器集。例如，**prod-node-us-east-1a**。
5. 6. 指定节点条件的超时持续时间。如果在超时时间内满足了条件，则会修复机器。超时时间较长可能会导致不健康的机器上的工作负载长时间停机。
7. 指定目标池中允许同时修复的机器数量。这可设为一个百分比或一个整数。如果不健康的机器数量超过 **maxUnhealthy** 设定的限制，则不会执行补救。
8. 指定机器健康检查在决定机器不健康前必须等待节点加入集群的超时持续时间。

注意

matchLabels 只是示例; 您必须根据具体需要映射您的机器组。

3.4.1. 短路机器健康检查补救

短路可确保仅在集群健康时机器健康检查修复机器。通过 **MachineHealthCheck** 资源中的 **maxUnhealthy** 字段配置短路。

如果用户在修复任何机器前为 **maxUnhealthy** 字段定义了一个值，**MachineHealthCheck** 会将 **maxUnhealthy** 的值与它决定不健康的目标池中的机器数量进行比较。如果不健康的机器数量超过 **maxUnhealthy** 限制，则不会执行补救。

重要

如果没有设置 **maxUnhealthy**，则默认值为 **100%**，无论集群状态如何，机器都会被修复。

适当的 **maxUnhealthy** 值取决于您部署的集群规模以及 **MachineHealthCheck** 覆盖的机器数量。例如，您可以使用 **maxUnhealthy** 值覆盖多个可用区间的多个机器集，以便在丢失整个区时，**maxUnhealthy** 设置可以在集群中阻止进一步补救。

maxUnhealthy 字段可以设置为整数或百分比。根据 **maxUnhealthy** 值，有不同的补救实现。

3.4.1.1. 使用绝对值设置 maxUnhealthy

如果将 **maxUnhealthy** 设为 **2**:

- 如果 2 个或更少节点不健康，则可执行补救
- 如果 3 个或更多节点不健康，则不会执行补救

这些值与机器健康检查要检查的机器数量无关。

3.4.1.2. 使用百分比设置 maxUnhealthy

如果 **maxUnhealthy** 被设置为 **40%**，有 25 个机器被检查：

- 如果有 10 个或更少节点处于不健康状态，则可执行补救
- 如果 11 个或多个节点不健康，则不会执行补救

如果 **maxUnhealthy** 被设置为 **40%**，有 6 个机器被检查：

- 如果 2 个或更少节点不健康，则可执行补救
- 如果 3 个或更多节点不健康，则不会执行补救

注意

当被检查的 **maxUnhealthy** 机器的百分比不是一个整数时，允许的机器数量会被舍入到一个小的整数。

3.5. 创建 MACHINEHEALTHCHECK 资源

您可以为集群中的所有 **MachineSet** 创建 **MachineHealthCheck** 资源。您不应该创建针对 control plane 机器的 **MachineHealthCheck** 资源。

先决条件

- 安装 **oc** 命令行界面。

流程

1. 创建一个 **healthcheck.yml** 文件，其中包含您的机器健康检查的定义。
2. 将 **healthcheck.yml** 文件应用到您的集群：

```
$ oc apply -f healthcheck.yml
```

第 4 章 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。

4.1. 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。

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

在发生触发配置集更改的事件时，或通过接收和处理终止信号安全终止容器化 TuneD 守护进程时，容器化 TuneD 守护进程所应用的节点级设置将被回滚。

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。

4.2. 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

1. 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。

警告

虽然在某些情况下，对 pod 标识的支持可以作为自动交付所需调整的一个便捷方式，但我们不鼓励使用这种方法，特别是在大型集群中。默认 Tuned CR 并不带有 pod 标识匹配。如果创建了带有 pod 标识匹配的自定义配置集，则该功能将在此时启用。在以后的 Node Tuning Operator 版本中可能会弃用 pod 标识功能。

4.3. 在集群中设置默认配置集

以下是在集群中设置的默认配置集。

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
```

```
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - 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=>4194304
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.conf.all.arp_announce=2
      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
      vm.max_map_count=262144

      [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
      fs.inotify.max_user_instances=8192

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

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

4.4. 验证是否应用了 TUNED 配置集

验证应用到集群节点的 TuneD 配置集。

```
$ oc get profile -n openshift-cluster-node-tuning-operator
```

输出示例

```
NAME          TUNED                APPLIED   DEGRADED   AGE
master-0      openshift-control-plane   True      False      6h33m
master-1      openshift-control-plane   True      False      6h33m
master-2      openshift-control-plane   True      False      6h33m
worker-a      openshift-node            True      False      6h28m
worker-b      openshift-node            True      False      6h28m
```

- **NAME**：配置集（Profile）对象的名称。每个节点有一个 Profile 对象，其名称相互匹配。
- **TUNED**：要应用的 TuneD 配置集的名称。
- **APPLIED**：如果 TuneD 守护进程应用了所需的配置集，则为 **True**。（**True/False/Unknown**）。
- **DEGRADED**：如果在应用 TuneD 配置集时报告了任何错误则为 **True**（**True/False/Unknown**）。
- **AGE**：创建 Profile 对象后经过的时间。

4.5. 自定义调整规格

Operator 的自定义资源 (CR) 包含两个主要部分。第一部分是 **profile:**，这是 TuneD 配置集及其名称的列表。第二部分是 **recommend:**，用来定义配置集选择逻辑。

多个自定义调优规格可以共存，作为 Operator 命名空间中的多个 CR。Operator 会检测到是否存在新 CR 或删除了旧 CR。所有现有的自定义性能优化设置都会合并，同时更新容器化 TuneD 守护进程的适当对象。

管理状态

通过调整默认的 Tuned CR 来设置 Operator Management 状态。默认情况下，Operator 处于 Managed 状态，默认的 Tuned CR 中没有 **spec.managementState** 字段。Operator Management 状态的有效值如下：

- Managed: Operator 会在配置资源更新时更新其操作对象
- Unmanaged: Operator 将忽略配置资源的更改
- Removed: Operator 将移除 Operator 置备的操作对象和资源

配置集数据

profile: 部分列出了 TuneD 配置集及其名称。

```
profile:
- name: tuned_profile_1
  data: |
    # TuneD profile specification
    [main]
    summary=Description of tuned_profile_1 profile

    [sysctl]
    net.ipv4.ip_forward=1
    # ... other sysctl's or other TuneD daemon plugins supported by the containerized TuneD

# ...

- name: tuned_profile_n
  data: |
    # TuneD profile specification
    [main]
    summary=Description of tuned_profile_n profile

    # tuned_profile_n profile settings
```

建议的配置集

profile: 选择逻辑通过 CR 的 **recommend:** 部分来定义。**recommend:** 部分是根据选择标准推荐配置集的项目列表。

```
recommend:
<recommend-item-1>
# ...
<recommend-item-n>
```

列表中的独立项：

```
- machineConfigLabels: (1)
    <mcLabels> (2)
```

```
    match: 3
      <match> 4
    priority: <priority> 5
    profile: <tuned_profile_name> 6
    operand: 7
      debug: <bool> 8
```

1. 可选。
2. **MachineConfig** 标签的键/值字典。键必须是唯一的。
3. 如果省略，则会假设配置集匹配，除非设置了优先级更高的配置集，或设置了 **machineConfigLabels**。
4. 可选列表。
5. 配置集排序优先级。较低数字表示优先级更高（**0** 是最高优先级）。
6. 在匹配项中应用的 TuneD 配置集。例如 **tuned_profile_1**。
7. 可选操作对象配置。
8. 为 TuneD 守护进程打开或关闭调试。**true** 为打开，**false** 为关闭。默认值为 **false**。

<match> 是一个递归定义的可选数组，如下所示：

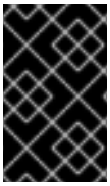
```
- label: <label_name> 1
  value: <label_value> 2
  type: <label_type> 3
    <match> 4
```

1. 节点或 pod 标签名称。
2. 可选的节点或 pod 标签值。如果省略，**<label_name>** 足以匹配。
3. 可选的对象类型（**node** 或 **pod**）。如果省略，会使用 **node**。
4. 可选的 **<match>** 列表。

如果不省略 **<match>**，则所有嵌套的 **<match>** 部分也必须评估为 **true**。否则会假定 **false**，并且不会应用或建议具有对应 **<match>** 部分的配置集。因此，嵌套（子级 **<match>** 部分）会以逻辑 AND 运算来运作。反之，如果匹配 **<match>** 列表中任何一项，整个 **<match>** 列表评估为 **true**。因此，该列表以逻辑 OR 运算来运作。

如果定义了 **machineConfigLabels**，基于机器配置池的匹配会对给定的 **recommend:** 列表项打开。**<mcLabels>** 指定机器配置标签。机器配置会自动创建，以在配置集 **<tuned_profile_name>** 中应用主机设置，如内核引导参数。这包括使用与 **<mcLabels>** 匹配的机器配置选择器查找所有机器配置池，并在分配了找到的机器配置池的所有节点上设置配置集 **<tuned_profile_name>**。要针对同时具有 master 和 worker 角色的节点，您必须使用 master 角色。

列表项 **match** 和 **machineConfigLabels** 由逻辑 OR 操作符连接。**match** 项首先以短电路方式评估。因此，如果它被评估为 **true**，则不考虑 **MachineConfigLabels** 项。



重要

当使用基于机器配置池的匹配时，建议将具有相同硬件配置的节点分组到同一机器配置池中。不遵循这个原则可能会导致在共享同一机器配置池的两个或者多个节点中 TuneD 操作对象导致内核参数冲突。

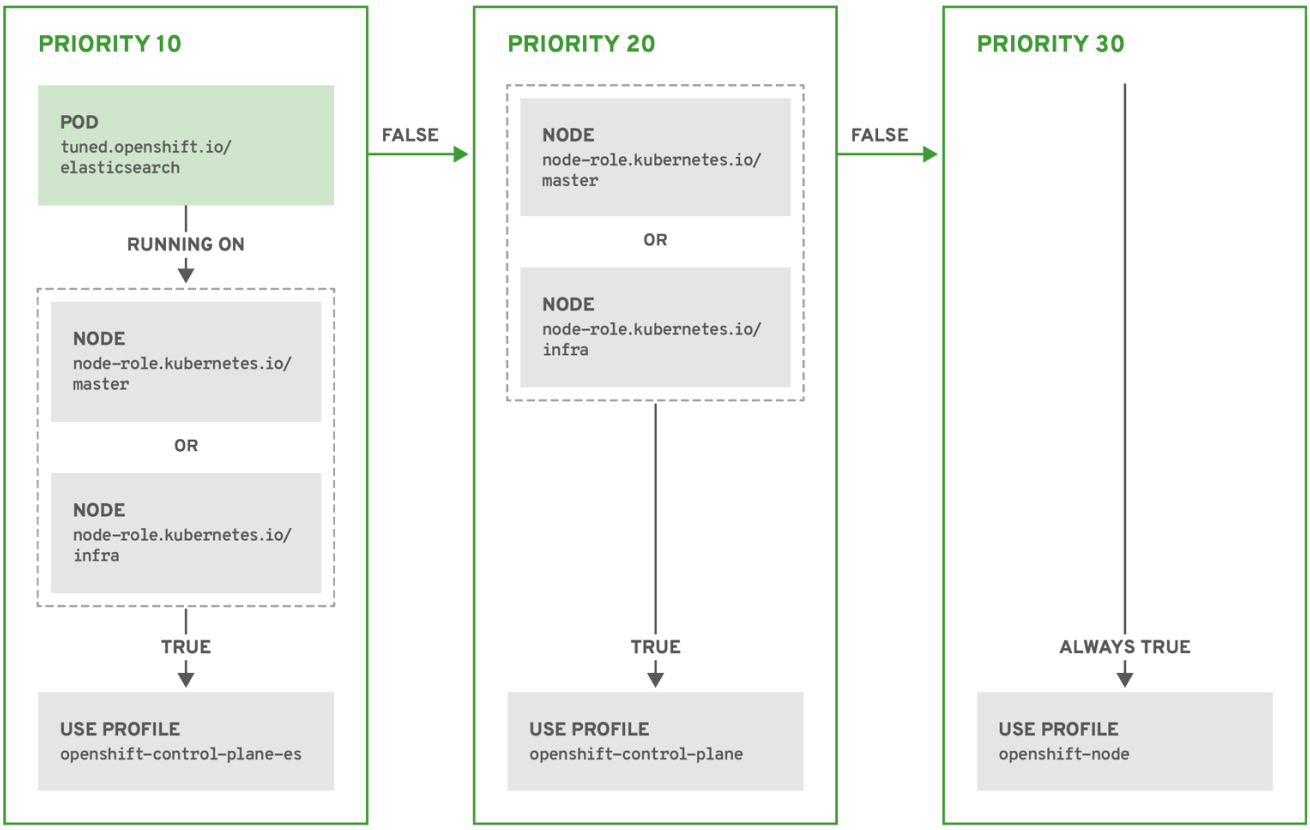
示例：基于节点或 pod 标签的匹配

```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra
  priority: 20
  profile: openshift-control-plane
- priority: 30
  profile: openshift-node
```

根据配置集优先级，以上 CR 针对容器化 TuneD 守护进程转换为 **recommend.conf** 文件。优先级最高 (**10**) 的配置集是 **openshift-control-plane-es**，因此会首先考虑它。在给定节点上运行的容器化 TuneD 守护进程会查看同一节点上是否在运行设有 **tuned.openshift.io/elasticsearch** 标签的 pod。如果没有，则整个 **<match>** 部分评估为 **false**。如果存在具有该标签的 pod，为了让 **<match>** 部分评估为 **true**，节点标签也需要是 **node-role.kubernetes.io/master** 或 **node-role.kubernetes.io/infra**。

如果这些标签对优先级为 **10** 的配置集而言匹配，则应用 **openshift-control-plane-es** 配置集，并且不考虑其他配置集。如果节点/pod 标签组合不匹配，则考虑优先级第二高的配置集 (**openshift-control-plane**)。如果容器化 TuneD Pod 在具有标签 **node-role.kubernetes.io/master** 或 **node-role.kubernetes.io/infra** 的节点上运行，则应用此配置集。

最后，配置集 **openshift-node** 的优先级最低 (**30**)。它没有 **<match>** 部分，因此始终匹配。如果给定节点上不匹配任何优先级更高的配置集，它会作为一个适用于所有节点的配置集来设置 **openshift-node** 配置集。



示例：基于机器配置池的匹配

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: openshift-node-custom
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=Custom OpenShift node profile with an additional kernel parameter
      include=openshift-node
      [bootloader]
      cmdline_openshift_node_custom=+skew_tick=1
    name: openshift-node-custom

  recommend:
  - machineConfigLabels:
      machineconfiguration.openshift.io/role: "worker-custom"
    priority: 20
    profile: openshift-node-custom
```

为尽量减少节点的重新引导情况，为目标节点添加机器配置池将匹配的节点选择器标签，然后创建上述 Tuned CR，最后创建自定义机器配置池。

4.6. 自定义调整示例

从默认 CR 中使用 TuneD 配置集

以下 CR 对带有标签 **tuned.openshift.io/ingress-node-label** 的 OpenShift Container Platform 节点应用节点一级的自定义调整。

示例：使用 openshift-control-plane TuneD 配置集进行自定义性能优化

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: ingress
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
      summary=A custom OpenShift ingress profile
      include=openshift-control-plane
      [sysctl]
      net.ipv4.ip_local_port_range="1024 65535"
      net.ipv4.tcp_tw_reuse=1
    name: openshift-ingress
  recommend:
  - match:
    - label: tuned.openshift.io/ingress-node-label
    priority: 10
    profile: openshift-ingress
```

重要

对于开发自定义配置集的人员。我们强烈建议包括在默认 Tuned CR 中提供的默认 TuneD 守护进程配置集。上面的示例使用默认 **openshift-control-plane** 配置集。

使用内置 TuneD 配置集

由于 NTO 管理的守护进程集已被成功推出，TuneD 操作对象会管理 TuneD 守护进程的同一版本。要列出守护进程支持的内置 TuneD 配置集，请以以下方式查询任何 TuneD pod：

```
$ oc exec $tuned_pod -n openshift-cluster-node-tuning-operator -- find /usr/lib/tuned/ -name
tuned.conf -printf '%h\n' | sed 's|^.*/||'
```

您可以使用自定义调优规格中检索的配置集名称。

示例：使用内置 hpc-compute TuneD 配置集

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: openshift-node-hpc-compute
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - data: |
      [main]
```

```
    summary=Custom OpenShift node profile for HPC compute workloads
    include=openshift-node,hpc-compute
  name: openshift-node-hpc-compute

recommend:
- match:
  - label: tuned.openshift.io/openshift-node-hpc-compute
  priority: 20
  profile: openshift-node-hpc-compute
```

除了内置的 **hpc-compute** 配置集外，上面的示例还包括默认 Tuned CR 中提供的 **openshift-node** TuneD 守护进程配置集，以对计算节点使用特定于 OpenShift 的调优。

4.7. 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。目前不支持以下 TuneD 插件：

- bootloader
- script
- systemd

如需更多信息，请参阅 Available TuneD Plug-ins 和 Getting Started with TuneD 。

第 5 章 使用 CLUSTER LOADER

Cluster Loader 是一个将大量对象部署到集群的工具程序，它可创建用户定义的集群对象。构建、配置并运行 Cluster Loader 以测量处于各种集群状态的 OpenShift Container Platform 部署的性能指标。

重要

Cluster Loader 现已弃用，并将在以后的发行版本中删除。

5.1. 安装 CLUSTER LOADER

流程

1. 要拉取容器镜像，请运行：

```
$ podman pull quay.io/openshift/origin-tests:4.8
```

5.2. 运行 CLUSTER LOADER

先决条件

- 软件仓库会提示您进行验证。registry 凭证允许您访问没有公开的镜像。使用您在安装时产生的现有身份验证凭证。

流程

1. 使用内置的测试配置执行 Cluster Loader，它会部署五个模板构建并等待它们完成：

```
$ podman run -v ${LOCAL_KUBECONFIG}:/root/.kube/config:z -i \
quay.io/openshift/origin-tests:4.8 /bin/bash -c 'export KUBECONFIG=/root/.kube/config && \
openshift-tests run-test "[sig-scalability][Feature:Performance] Load cluster \
should populate the cluster [Slow][Serial] [Suite:openshift]"'
```

或者，通过设置 **VIPERCONFIG** 环境变量来执行带有用户定义的配置的 Cluster Loader：

```
$ podman run -v ${LOCAL_KUBECONFIG}:/root/.kube/config:z \
-v ${LOCAL_CONFIG_FILE_PATH}:/root/configs/:z \
-i quay.io/openshift/origin-tests:4.8 \
/bin/bash -c 'KUBECONFIG=/root/.kube/config VIPERCONFIG=/root/configs/test.yaml \
openshift-tests run-test "[sig-scalability][Feature:Performance] Load cluster \
should populate the cluster [Slow][Serial] [Suite:openshift]"'
```

在这个示例中，**${LOCAL_KUBECONFIG}** 代表 **kubeconfig** 在本地文件系统中的路径。另外，还有一个名为 **${LOCAL_CONFIG_FILE_PATH}**的目录，它被挂载到包含名为 **test.yaml**的配置文件的容器中。另外，如果 **test.yaml** 引用了任何外部模板文件或 podspec 文件，则也应该被挂载到容器中。

5.3. 配置 CLUSTER LOADER

该工具创建多个命名空间（项目），其中包含多个模板或 pod。

5.3.1. Cluster Loader 配置文件示例

Cluster Loader 的配置文件是一个基本的 YAML 文件：

```
provider: local 1
ClusterLoader:
  cleanup: true
  projects:
    - num: 1
      basename: clusterloader-cakephp-mysql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: cakephp-mysql.json

    - num: 1
      basename: clusterloader-dancer-mysql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: dancer-mysql.json

    - num: 1
      basename: clusterloader-django-postgresql
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: django-postgresql.json

    - num: 1
      basename: clusterloader-nodejs-mongodb
      tuning: default
      ifexists: reuse
      templates:
        - num: 1
          file: quickstarts/nodejs-mongodb.json

    - num: 1
      basename: clusterloader-rails-postgresql
      tuning: default
      templates:
        - num: 1
          file: rails-postgresql.json

  tuningsets: 2
    - name: default
      pods:
        stepping: 3
          stepsize: 5
          pause: 0 s
        rate_limit: 4
          delay: 0 ms
```

1. 端到端测试的可选设置。设置为 **local** 以避免额外的日志信息。
2. 调整集允许速率限制和分步，可以生成几批 pod，同时在两组间暂停使用。在继续执行前，Cluster Loader 会监控上一步的完成情况。
3. 为每 **N** 个对象被创建后，会暂停 **M** 秒。
4. 在创建不同对象期间，限制率会等待 **M** 毫秒。

本例假定对任何外部模板文件或 pod spec 文件的引用也会挂载到容器中。

重要

如果您在 Microsoft Azure 上运行 Cluster Loader，则必须将 **AZURE_AUTH_LOCATION** 变量设置为包含 **terraform.azure.auto.tfvars.json** 输出结果的文件，该文件存在于安装程序目录中。

5.3.2. 配置字段

表 5.1. 顶层 Cluster Loader 字段

字段	描述
cleanup	可设置为 **true** 或 **false**。每个配置有一个定义。如果设置为 **true**，**cleanup** 会删除所有由 Cluster Loader 在测试结束时创建的命名空间（项目）。
projects	包含一个或多个定义的子对象。在 **projects** 下，定义了要创建的每个命名空间，**projects** 有几个必需的子标题。
tuningsets	每个配置都有一个定义的子对象。**tuningset** 允许用户定义一个调整集，为创建项目或对象（pods、模板等）添加可配置的计时。
sync	每个配置都有一个定义的可选子对象。在创建对象的过程中添加同步的可能性。

表 5.2. projects 下的字段

字段	描述
num	整数。定义要创建项目的数量。
basename	字符串项目基本名称的一个定义。在**Basename**后面会附加相同命名空间的计数以避免冲突。
tuning	字符串需要应用到在这个命名空间里部署的项目的 tuning 设置。

字段	描述
ifexists	包含 **reuse** 或 **delete** 的字符串。如果发现一个项目或者命名空间的名称与执行期间创建的项目或命名空间的名称相同时，需要进行什么操作。
configmaps	键值对列表。键是配置映射名称，值是指向创建配置映射的文件的路径。
secrets	键值对列表。key 是 secret 名称，值是一个指向用来创建 secret 的文件的路径。
pods	要部署的 pod 的一个或者多个定义的子对象。
templates	要部署模板的一个或者多个定义的子对象。

表 5.3. pods 和 templates 下的字段

字段	描述
num	整数。要部署的 pod 或模板数量。
image	字符串到可以拉取镜像的软件仓库的 docker 镜像 URL。
basename	字符串要创建的模板（或 pod）的基本名称的一个定义。
file	字符串到要创建的 pod 规格或模板的本地文件的路径。
parameters	健值对。在 **parameters** 下，您可以指定一组值在 pod 或模板中进行覆盖。

表 5.4. tuningsets 下的字段

字段	描述
name	字符串tuning 集的名称，该名称将与在一个项目中定义 turning 时指定的名称匹配。
pods	指定应用于 pod 的 **tuningsets** 的子对象。
templates	指定应用于模板的 **tuningsets** 的子对象。

表 5.5. tuningsets pods 或 tuningsets templates 下的字段

字段	描述
stepping	子对象。如果要在步骤创建模式中创建对象，需要使用的步骤配置。
rate_limit	子对象。用来限制对象创建率的频率限制 turning 集。

表 5.6. **tuningsets pods 或 tuningsets templates, stepping 下的字段**

字段	描述
stepsize	整数。在暂停对象创建前要创建的对象数量。
pause	整数。在创建了由 **stepsize** 定义的对象数后需要暂停的秒数。
timeout	整数。如果对象创建失败，在失败前要等待的秒数。
delay	整数。在创建请求间等待多少毫秒 (ms)

表 5.7. **sync 下的字段**

字段	描述
server	带有 **enabled** 和 **port** 字段的子对象。布尔值 **enabled** 定义了是否启动用于 pod 同步的 HTTP 服务器。整数值 **port** 定义了要监听的 HTTP 服务器端口（默认为 **9090**）。
running	布尔值等待带有与**selectors** 匹配的标签的 pod 进入 **Running** 状态。
succeeded	布尔值等待带有与**selectors** 匹配的标签的 pod 进入 **Completed** 状态。
selectors	匹配处于 **Running** 或 **Completed** 状态 的 pod 的选择器列表。
timeout	字符串等待处于 **Running** 或 **Completed** 状态的 pod 的同步超时时间。对于不是 **0** 的值，其时间单位是：[ns|us|ms|s|m|h]

5.4. 已知问题

- 当在没有配置的情况下调用 Cluster Loader 会失败。(BZ#1761925)
- 如果用户模板中没有定义 **IDENTIFIER** 参数，则模板创建失败，错误信息为：**error: unknown parameter name "IDENTIFIER"**。如果部署模板，在模板中添加这个参数以避免出现这个错误：

```
{
  "name": "IDENTIFIER",
  "description": "Number to append to the name of resources",
  "value": "1"
}
```

如果部署 pod，则不需要添加该参数。

第 6 章 使用 CPU MANAGER

CPU Manager 管理 CPU 组并限制特定 CPU 的负载。

CPU Manager 对于有以下属性的负载有用：

- 需要尽可能多的 CPU 时间。
- 对处理器缓存丢失非常敏感。
- 低延迟网络应用程序。
- 需要与其他进程协调，并从共享一个处理器缓存中受益。

6.1. 设置 CPU MANAGER

流程

1. 可选：标记节点：

```
# oc label node perf-node.example.com cpumanager=true
```

2. 编辑启用 CPU Manager 的节点的 **MachineConfigPool** 。在这个示例中，所有 worker 都启用了 CPU Manager：

```
# oc edit machineconfigpool worker
```

3. 为 worker 机器配置池添加标签：

```
metadata:
  creationTimestamp: 2020-xx-xxx
  generation: 3
  labels:
    custom-kubelet: cpumanager-enabled
```

4. 创建 **KubeletConfig**，**cpumanager-kubeletconfig.yaml**，自定义资源 (CR) 。请参阅上一步中创建的标签，以便使用新的 kubelet 配置更新正确的节点。请参见 **MachineConfigPoolSelector** 部分：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
     cpuManagerPolicy: static 1
     cpuManagerReconcilePeriod: 5s 2
```

1 指定一个策略：

- **none**.这个策略明确启用了现有的默认 CPU 关联性方案，从而不会出现超越调度程序自动进行的关联性。
- **static**。此策略允许具有某些资源特征的 pod 获得提高 CPU 关联性和节点上专用的 pod。

2 可选。指定 CPU Manager 协调频率。默认值为 **5s**。

5. 创建动态 kubelet 配置：

```
# oc create -f cpumanager-kubeletconfig.yaml
```

这会在 kubelet 配置中添加 CPU Manager 功能，如果需要，Machine Config Operator（MCO）将重启节点。要启用 CPU Manager，则不需要重启。

6. 检查合并的 kubelet 配置：

```
# oc get machineconfig 99-worker-XXXXXX-XXXXX-XXXX-XXXXX-kubelet -o json | grep ownerReference -A7
```

输出示例

```
    "ownerReferences": [
        {
            "apiVersion": "machineconfiguration.openshift.io/v1",
            "kind": "KubeletConfig",
            "name": "cpumanager-enabled",
            "uid": "7ed5616d-6b72-11e9-aae1-021e1ce18878"
        }
    ]
```

7. 检查 worker 是否有更新的 **kubelet.conf**：

```
# oc debug node/perf-node.example.com
sh-4.2# cat /host/etc/kubernetes/kubelet.conf | grep cpuManager
```

输出示例

```
cpuManagerPolicy: static      1
cpuManagerReconcilePeriod: 5s 2
```

1 2 当创建 **KubeletConfig** CR 时会定义这些设置。

8. 创建请求一个或多个内核的 pod。限制和请求都必须将其 CPU 值设置为一个整数。这是专用于此 pod 的内核数：

```
# cat cpumanager-pod.yaml
```

输出示例

```
apiVersion: v1
```

```
kind: Pod
metadata:
  generateName: cpumanager-
spec:
  containers:
  - name: cpumanager
    image: gcr.io/google_containers/pause-amd64:3.0
    resources:
      requests:
        cpu: 1
        memory: "1G"
      limits:
        cpu: 1
        memory: "1G"
  nodeSelector:
    cpumanager: "true"
```

9. 创建 pod：

```
# oc create -f cpumanager-pod.yaml
```

10. 确定为您标记的节点调度了 pod：

```
# oc describe pod cpumanager
```

输出示例

```
Name:               cpumanager-6cqz7
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:  perf-node.example.com/xxx.xx.xx.xxx
...
 Limits:
      cpu:     1
      memory:  1G
    Requests:
      cpu:        1
      memory:     1G
...
QoS Class:       Guaranteed
Node-Selectors:  cpumanager=true
```

11. 确认正确配置了 **cgroups**。获取 **pause** 进程的进程 ID（PID）：

```
# ├─init.scope
│ └─1 /usr/lib/systemd/systemd --switched-root --system --deserialize 17
└─kubepods.slice
  ├─kubepods-pod69c01f8e_6b74_11e9_ac0f_0a2b62178a22.slice
  │ ├─crio-b5437308f1a574c542bdf08563b865c0345c8f8c0b0a655612c.scope
  │ └─32706 /pause
```

服务质量（QoS）等级为 **Guaranteed** 的 pod 被放置到 **kubepods.slice** 中。其它 QoS 等级的 pod 会位于 **kubepods** 的子 **cgroups** 中：

```
# cd /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-
pod69c01f8e_6b74_11e9_ac0f_0a2b62178a22.slice/crio-
b5437308f1ad1a7db0574c542bdf08563b865c0345c86e9585f8c0b0a655612c.scope
# for i in `ls cpuset.cpus tasks` ; do echo -n "$i "; cat $i ; done
```

输出示例

```
cpuset.cpus 1
tasks 32706
```

12. 检查任务允许的 CPU 列表：

```
# grep ^Cpus_allowed_list /proc/32706/status
```

输出示例

```
 Cpus_allowed_list:    1
```

13. 确认系统中的另一个 pod（在这个示例中，QoS 等级为 **burstable** 的 pod）不能在为等级为**Guaranteed** 的 pod 分配的内核中运行：

```
# cat /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-besteffort.slice/kubepods-besteffort-
podc494a073_6b77_11e9_98c0_06bba5c387ea.slice/crio-
c56982f57b75a2420947f0afc6cafe7534c5734efc34157525fa9abbf99e3849.scope/cpuset.cpus

0
# oc describe node perf-node.example.com
```

输出示例

```
...
Capacity:
 attachable-volumes-aws-ebs:  39
 cpu:                         2
 ephemeral-storage:           124768236Ki
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      8162900Ki
 pods:                        250
Allocatable:
 attachable-volumes-aws-ebs:  39
 cpu:                         1500m
 ephemeral-storage:           124768236Ki
 hugepages-1Gi:               0
 hugepages-2Mi:               0
 memory:                      7548500Ki
 pods:                        250
-------                           ----                        ------------  ----------  ---------------  -------------  --
-
  default                         cpumanager-6cqz7            1 (66%)       1 (66%)     1G (12%)
1G (12%)       29m

Allocated resources:
```

```
(Total limits may be over 100 percent, i.e., overcommitted.)
Resource                    Requests         Limits
--------                    --------         ------
cpu                         1440m (96%)      1 (66%)
```

这个 VM 有两个 CPU 内核。**system-reserved** 设置保留 500 millicores，这代表一个内核中的一半被从节点的总容量中减小，以达到 **Node Allocatable** 的数量。您可以看到 **Allocatable CPU** 是 1500 毫秒。这意味着您可以运行一个 CPU Manager pod，因为每个 pod 需要一个完整的内核。一个完整的内核等于 1000 毫秒。如果您尝试调度第二个 pod，系统将接受该 pod，但不会调度它：

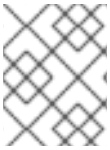
```
NAME                    READY   STATUS    RESTARTS   AGE
cpumanager-6cqz7        1/1     Running   0          33m
cpumanager-7qc2t        0/1     Pending   0          11s
```

第 7 章 使用拓扑管理器

拓扑管理器（Topology Manager）从 CPU Manager、设备管理器和其他 Hint 提供者收集提示信息，以匹配相同非统一内存访问（NUMA）节点上的所有 QoS 类的 pod 资源（如 CPU、SR-IOV VF 和其他设备资源）。

拓扑管理器使用收集来的提示信息中获得的拓扑信息，根据配置的 Topology Manager 策略以及请求的 Pod 资源，决定节点是否被节点接受或拒绝。

拓扑管理器对希望使用硬件加速器来支持对工作延迟有极高要求的操作及高吞吐并发计算的负载很有用。



注意

要使用拓扑管理器，必须使用 **static** 策略的 CPU Manager。有关 CPU Manager 的详情请参考使用 CPU Manager。

7.1. 拓扑管理器策略

拓扑管理器通过从 Hint 提供者（如 CPU Manager 和设备管理器）收集拓扑提示来调整所有级别服务质量（QoS）的 **Pod** 资源，并使用收集的提示来匹配 **Pod** 资源。

注意

要将 CPU 资源与 **Pod** 规格中的其他请求资源匹配，必须使用 **static** CPU Manager 策略启用 CPU Manager。

拓扑管理器支持四个分配策略，这些策略在 **cpumanager-enabled** 自定义资源（CR）中定义：

none 策略

这是默认策略，不执行任何拓扑对齐调整。

best-effort 策略

对于带有 **best-effort** 拓扑管理策略的 pod 中的每个容器，kubelet 会调用每个 Hint 提供者来发现其资源的可用性。使用这些信息，拓扑管理器会保存那个容器的首选 NUMA 节点关联性设置。如果关联性没有被首选设置，则拓扑管理器会保存这个设置，并把 pod 分配给节点。

restricted 策略

对于带有 **restricted** 拓扑管理策略的 pod 中的每个容器，kubelet 会调用每个 Hint 提供者来发现其资源的可用性。使用这些信息，拓扑管理器会保存那个容器的首选 NUMA 节点关联性设置。如果关联性没有被首选，则拓扑管理器会从节点拒绝这个 pod，从而导致 pod 处于 **Terminated** 状态，且 pod 准入失败。

single-numa-node 策略

对于带有 **single-numa-node** 拓扑管理策略的 pod 中的每个容器，kubelet 会调用每个 Hint 提供者来发现其资源的可用性。使用这个信息，拓扑管理器会决定单个 NUMA 节点关联性是否可能。如果是，pod 将会分配给该节点。如果无法使用单一 NUMA 节点关联性，则拓扑管理器会拒绝来自节点的 pod。这会导致 pod 处于 Terminated 状态，且 pod 准入失败。

7.2. 设置拓扑管理器

要使用拓扑管理器，您必须在 **cpumanager-enabled** 自定义资源（CR）中配置分配策略。如果您设置了 CPU Manager，则该文件可能会存在。如果这个文件不存在，您可以创建该文件。

先决条件

- 将 CPU Manager 策略配置为 **static**。请参考扩展和性能文档中的使用 CPU Manager 部分 。

流程

激活 Topolgy Manager:

1. 在 **cpumanager-enabled** 自定义资源（CR）中配置拓扑管理器分配策略。

```
$ oc edit KubeletConfig cpumanager-enabled
```

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
    cpuManagerPolicy: static 1
    cpuManagerReconcilePeriod: 5s
    topologyManagerPolicy: single-numa-node 2
```

1. 此参数必须是 **static**。
2. 指定所选拓扑管理器分配策略。在这里，策略是 **single-numa-node**。有效值为：**default**、**best-effort**、**restricted**、**single-numa-node**。

其他资源

- 有关 CPU Manager 的详情请参考使用 CPU Manager。

7.3. POD 与拓扑管理器策略的交互

以下的 **Pod** specs 示例演示了 Pod 与 Topology Manager 的交互。

因为没有指定资源请求或限制，以下 pod 以 **BestEffort** QoS 类运行。

```
spec:
  containers:
  - name: nginx
    image: nginx
```

因为请求小于限制，下一个 pod 以 **Burstable** QoS 类运行。

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
```

```
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

如果所选策略不是 **none**，则拓扑管理器将不考虑其中任何一个 **Pod** 规格。

因为请求等于限制，最后一个 pod 以 Guaranteed QoS 类运行。

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

拓扑管理器将考虑这个 pod。拓扑管理器会参考 CPU Manager 的静态策略，该策略可返回可用 CPU 的拓扑结构。拓扑管理器还参考设备管理器来发现可用设备的拓扑结构，如 example.com/device。

拓扑管理器将使用此信息存储该容器的最佳拓扑。在本 pod 中，CPU Manager 和设备管理器将在资源分配阶段使用此存储的信息。

第 8 章 扩展 CLUSTER MONITORING OPERATOR

OpenShift Container Platform 会提供 Cluster Monitoring Operator 在基于 Prometheus 的监控堆栈中收集并存储的数据。作为管理员，您可以在一个 dashboard 接口（Grafana）中查看系统资源、容器和组件指标。

8.1. PROMETHEUS 数据库存储要求

红帽对不同的扩展大小进行了各种测试。

注意

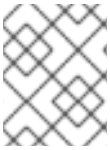
以下 Prometheus 存储要求并不具有规定性。取决于工作负载活动和资源使用情况，集群中可能会观察到更高资源消耗。

表 8.1. Prometheus 数据库的存储要求取决于集群中的节点/pod 数量

节点数量	pod 数量	每天增加的 Prometheus 存储	每 15 天增加的 Prometheus 存储	RAM 空间（每个缩放大小）	网络（每个 tsdb 块）
50	1800	6.3 GB	94 GB	6 GB	16 MB
100	3600	13 GB	195 GB	10 GB	26 MB
150	5400	19 GB	283 GB	12 GB	36 MB
200	7200	25 GB	375 GB	14 GB	46 MB

大约 20%的预期大小被添加为开销，以保证存储要求不会超过计算的值。

上面的计算用于默认的 OpenShift Container Platform Cluster Monitoring Operator。



注意

CPU 利用率会有轻微影响。这个比例为在每 50 个节点和 1800 个 pod 的 40 个内核中大约有 1 个。

针对 OpenShift Container Platform 的建议

- 至少使用三个基础架构（infra）节点。
- 至少使用三个带有 NVMe（non-volatile memory express）驱动的 **openshift-container-storage** 节点。

8.2. 配置集群监控

您可以为集群监控堆栈中的 Prometheus 组件增加存储容量。

流程

为 Prometheus 增加存储容量：

1. 创建 YAML 配置文件 **cluster-monitoring-config.yaml**。例如：

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |
    prometheusK8s:
      retention: {{PROMETHEUS_RETENTION_PERIOD}} 1
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      volumeClaimTemplate:
        spec:
          storageClassName: {{STORAGE_CLASS}} 2
          resources:
            requests:
              storage: {{PROMETHEUS_STORAGE_SIZE}} 3
    alertmanagerMain:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      volumeClaimTemplate:
        spec:
          storageClassName: {{STORAGE_CLASS}} 4
          resources:
            requests:
              storage: {{ALERTMANAGER_STORAGE_SIZE}} 5
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
```

1. 一个典型的值是 **PROMETHEUS_retention_PERIOD=15d**。时间单位使用以下后缀之一：s、m、h、d。

2 4. 集群的存储类。

3. 一个典型的值是 **PROMETHEUS_STORAGE_SIZE=2000Gi**。存储值可以是一个纯整数，也可以是带有以下后缀之一的整数：E、P、T、G、M、K。您也可以使用以下效果相同的后缀：Ei、Pi、Ti、Gi、Mi、Ki。

5. 一个典型的值 **是 alertmanager_STORAGE_SIZE=20Gi**。存储值可以是一个纯整数，也可以是带有以下后缀之一的整数：E、P、T、G、M、K。您也可以使用以下效果相同的后缀：Ei、Pi、Ti、Gi、Mi、Ki。

2. 为保留周期、存储类和存储大小添加值。

3. 保存这个文件。

4. 运行以下命令应用这些更改：

```
$ oc create -f cluster-monitoring-config.yaml
```

第 9 章 NODE FEATURE DISCOVERY OPERATOR

了解 Node Feature Discovery（NFD）Operator 以及如何使用它通过编排节点功能发现（用于检测硬件功能和系统配置的 Kubernetes 附加组件）来公开节点级信息。

9.1. 关于 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery Operator（NFD）通过将节点标记为硬件特定信息来管理 OpenShift Container Platform 集群中硬件功能和配置的检测。NFD 使用特定于节点的属性标记主机，如 PCI 卡、内核、操作系统版本等。

NFD Operator 可以通过搜索 "Node Feature Discovery" 在 Operator Hub 上找到。

9.2. 安装 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery（NFD）Operator 编排运行 NFD 守护进程集需要的所有资源。作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台安装 NFD Operator。

9.2.1. 使用 CLI 安装 NFD Operator

作为集群管理员，您可以使用 CLI 安装 NFD Operator。

先决条件

- OpenShift Container Platform 集群
- 安装 OpenShift CLI (**oc**) 。
- 以具有 **cluster-admin** 特权的用户身份登录。

流程

1. 为 NFD Operator 创建命名空间。
   a. 创建定义 **openshift-nfd** 命名空间的以下 **Namespace** 自定义资源（CR），然后在 **nfd-namespace.yaml** 文件中保存 YAML：

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-nfd
```

   b. 运行以下命令创建命名空间：

```
$ oc create -f nfd-namespace.yaml
```

2. 通过创建以下对象，在您上一步中创建的命名空间中安装 NFD Operator：
   a. 创建以下 **OperatorGroup** CR，并在 nfd **-operatorgroup.yaml** 文件中保存 YAML：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
```

```
  generateName: openshift-nfd-
  name: openshift-nfd
  namespace: openshift-nfd
spec:
  targetNamespaces:
  - openshift-nfd
```

b. 运行以下命令来创建 **OperatorGroup** CR:

```
$ oc create -f nfd-operatorgroup.yaml
```

c. 运行以下命令获取下一步所需的 **channel** 值。

```
$ oc get packagemanifest nfd -n openshift-marketplace -o
jsonpath='{.status.defaultChannel}'
```

输出示例

```
4.8
```

d. 创建以下 **Subscription** CR，并将 YAML 保存到 **nfd-sub.yaml** 文件中：

订阅示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: nfd
  namespace: openshift-nfd
spec:
  channel: "4.8"
  installPlanApproval: Automatic
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

e. 运行以下命令来创建订阅对象：

```
$ oc create -f nfd-sub.yaml
```

f. 进入 **openshift-nfd** 项目：

```
$ oc project openshift-nfd
```

验证

- 要验证 Operator 部署是否成功，请运行：

```
$ oc get pods
```

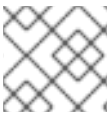
输出示例

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          10m
```

一个成功的部署会显示 **Running** 状态。

9.2.2. 使用 Web 控制台安装 NFD Operator

作为集群管理员，您可以使用 Web 控制台安装 NFD Operator。



注意

建议按照上一节所述创建**命名空间**。

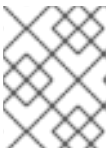
流程

1. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
2. 从可用的 Operator 列表中选择 **Node Feature Discovery**，然后点 **Install**。
3. 在 **Install Operator** 页面中，选择**集群上的一个特定命名空间**，选择上一节中创建的命名空间，然后点 **Install**。

验证

验证 NFD Operator 是否已成功安装：

1. 进入到 **Operators → Installed Operators** 页面。
2. 确保 **openshift-nfd** 项目中列出了 **Node Feature Discovery**，**Status** 为 **InstallSucceeded**。



注意

在安装过程中，Operator 可能会显示 **Failed** 状态。如果安装过程结束后有 **InstallSucceeded** 信息，您可以忽略这个 **Failed** 信息。

故障排除

如果 Operator 没有被安装，请按照以下步骤进行故障排除：

1. 导航到 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。
2. 导航到 **Workloads → Pods** 页面，在 **openshift-nfd** 项目中检查 pod 的日志。

9.3. 使用 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery（NFD）Operator 通过监视 **NodeFeatureDiscovery** CR 来编排运行 Node-Feature-Discovery 守护进程所需的所有资源。根据 **NodeFeatureDiscovery** CR，Operator 将在所需命名空间中创建操作对象（NFD）组件。您可以编辑 CR 来选择另一个 **命名空间**、**镜像**、**imagePullPolicy** 和 **nfd-worker-conf**，以及其他选项。

作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台创建 **NodeFeatureDiscovery** 实例。

9.3.1. 使用 CLI 创建 NodeFeatureDiscovery 实例

作为集群管理员，您可以使用 CLI 创建 **NodeFeatureDiscovery** CR 实例。

先决条件

- OpenShift Container Platform 集群
- 安装 OpenShift CLI (**oc**) 。
- 以具有 **cluster-admin** 权限的用户身份登录。
- 安装 NFD Operator。

流程

1. 创建以下 **NodeFeatureDiscovery** 自定义资源（CR），然后在 **NodeFeatureDiscovery.yaml** 文件中保存 YAML：

```
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: "" # instance is empty by default
  operand:
    namespace: openshift-nfd
    image: quay.io/openshift/origin-node-feature-discovery:4.8
    imagePullPolicy: Always
  workerConfig:
    configData: |
      #core:
      #  labelWhiteList:
      #  noPublish: false
      #  sleepInterval: 60s
      #  sources: [all]
      #  klog:
      #    addDirHeader: false
      #    alsologtostderr: false
      #    logBacktraceAt:
      #    logtostderr: true
      #    skipHeaders: false
      #    stderrthreshold: 2
      #    v: 0
      #    vmodule:
      ##   NOTE: the following options are not dynamically run-time configurable
      ##         and require a nfd-worker restart to take effect after being changed
      #    logDir:
      #    logFile:
      #    logFileMaxSize: 1800
      #    skipLogHeaders: false
      #sources:
      #  cpu:
      #    cpuid:
      ##     NOTE: whitelist has priority over blacklist
```

```
#      attributeBlacklist:
#        - "BMI1"
#        - "BMI2"
#        - "CLMUL"
#        - "CMOV"
#        - "CX16"
#        - "ERMS"
#        - "F16C"
#        - "HTT"
#        - "LZCNT"
#        - "MMX"
#        - "MMXEXT"
#        - "NX"
#        - "POPCNT"
#        - "RDRAND"
#        - "RDSEED"
#        - "RDTSCP"
#        - "SGX"
#        - "SSE"
#        - "SSE2"
#        - "SSE3"
#        - "SSE4.1"
#        - "SSE4.2"
#        - "SSSE3"
#      attributeWhitelist:
#  kernel:
#    kconfigFile: "/path/to/kconfig"
#    configOpts:
#      - "NO_HZ"
#      - "X86"
#      - "DMI"
#  pci:
#    deviceClassWhitelist:
#      - "0200"
#      - "03"
#      - "12"
#    deviceLabelFields:
#      - "class"
#      - "vendor"
#      - "device"
#      - "subsystem_vendor"
#      - "subsystem_device"
#  usb:
#    deviceClassWhitelist:
#      - "0e"
#      - "ef"
#      - "fe"
#      - "ff"
#    deviceLabelFields:
#      - "class"
#      - "vendor"
#      - "device"
#  custom:
#    - name: "my.kernel.feature"
#      matchOn:
#        - loadedKMod: ["example_kmod1", "example_kmod2"]
```

```
    #    - name: "my.pci.feature"
    #      matchOn:
    #        - pciId:
    #            class: ["0200"]
    #            vendor: ["15b3"]
    #            device: ["1014", "1017"]
    #        - pciId :
    #            vendor: ["8086"]
    #            device: ["1000", "1100"]
    #    - name: "my.usb.feature"
    #      matchOn:
    #        - usbId:
    #          class: ["ff"]
    #          vendor: ["03e7"]
    #          device: ["2485"]
    #        - usbId:
    #          class: ["fe"]
    #          vendor: ["1a6e"]
    #          device: ["089a"]
    #    - name: "my.combined.feature"
    #      matchOn:
    #        - pciId:
    #            vendor: ["15b3"]
    #            device: ["1014", "1017"]
    #          loadedKMod : ["vendor_kmod1", "vendor_kmod2"]
  customConfig:
    configData: |
    #    - name: "more.kernel.features"
    #      matchOn:
    #      - loadedKMod: ["example_kmod3"]
    #    - name: "more.features.by.nodename"
    #      value: customValue
    #      matchOn:
    #      - nodename: ["special-.*-node-.*"]
```

2. 运行以下命令来创建 **NodeFeatureDiscovery** CR 实例：

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

验证

- 要验证是否已创建实例，请运行：

```
$ oc get pods
```

输出示例

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          11m
nfd-master-hcn64                      1/1     Running   0          60s
nfd-master-lnnxx                      1/1     Running   0          60s
nfd-master-mp6hr                      1/1     Running   0          60s
nfd-worker-vgcz9                      1/1     Running   0          60s
nfd-worker-xqbws                      1/1     Running   0          60s
```

一个成功的部署会显示 **Running** 状态。

9.3.2. 使用 Web 控制台创建 NodeFeatureDiscovery CR

流程

1. 进入到 **Operators → Installed Operators** 页面。
2. 查找 **Node Feature Discovery**，并在 **Provided APIs** 下看到一个方框。
3. 单击 **Create instance**。
4. 编辑 **NodeFeatureDiscovery** CR 的值。
5. 点 **Create**。

9.4. 配置 NODE FEATURE DISCOVERY OPERATOR

9.4.1. core

core 部分包含不特定于任何特定功能源的常见配置设置。

core.sleepInterval
core.sleepInterval 指定连续通过功能检测或重新检测之间的间隔，还可指定节点重新标记之间的间隔。非正数值意味着睡眠间隔无限；不进行重新检测或重新标记。

如果指定，这个值会被弃用的 **--sleep-interval** 命令行标志覆盖。

用法示例

```
core:
  sleepInterval: 60s 1
```

默认值为 **60s**。

core.sources
core.sources 指定启用的功能源列表。特殊值 **all** 可启用所有功能源。

如果指定，这个值会被弃用的 **--sources** 命令行标志覆盖。

默认：**[all]**

用法示例

```
core:
  sources:
    - system
    - custom
```

core.labelWhiteList
core.labelWhiteList 根据标签名称指定用于过滤功能标签的正则表达式。不匹配的标签将不会被发布。

正则表达式仅与标签的 basename 部分（"/"后的名称部分）进行匹配。标签前缀或命名空间会被省略。

如果指定，这个值会被弃用的 **--label-whitelist** 命令行标志覆盖。

默认： **null**

用法示例

```
core:
  labelWhiteList: '^cpu-cpuid'
```

core.noPublish
将 **core.noPublish** 设置为 **true** 可禁用与 **nfd-master** 的所有通信。它实际上是一个空运行标记; **nfd-worker** 会正常运行功能检测，但不会向 **nfd-master** 发送实际的标记请求。

如果指定，**--no-publish** 命令行标志会覆盖这个值。

例如：

用法示例

```
core:
  noPublish: true 1
```

默认值为 **false**。

core.klog
以下选项指定日志记录器配置，其中大多数可以在运行时动态调整。

日志记录器选项也可以使用命令行标志来指定，其优先级高于任何对应的配置文件选项。

core.klog.addDirHeader
如果设置为 **true**，**core.klog.addDirHeader** 将文件目录添加到日志消息的标头中。

默认：**false**

运行时可配置：是

core.klog.alsologtostderr
将日志信息输出到标准错误以及文件。

默认：**false**

运行时可配置：是

core.klog.logBacktraceAt
当日志记录达到行 file:N 时，触发堆栈跟踪功能。

默认： **空**

运行时可配置：是

core.klog.logDir
如果非空，在此目录中写入日志文件。

默认： **空**

运行是时配置：否

core.klog.logFile
如果不为空，则使用此日志文件。

默认： **空**

运行是时配置：否

core.klog.logFileMaxSize
core.klog.logFileMaxSize 定义日志文件可增大的最大大小。单位是 MB。如果值为 **0**，则最大文件大小没有限制。

默认： **1800**

运行是时配置：否

core.klog.logtostderr
将日志信息输出到标准错误而不是文件

默认： **true**

运行时可配置：是

core.klog.skipHeaders
如果 **core.klog.skipHeaders** 设为 **true**，忽略日志消息中的标头前缀。

默认：**false**

运行时可配置：是

core.klog.skipLogHeaders
如果 **core.klog.skipLogHeaders** 设为 **true**，在打开日志文件时忽略标头。

默认：**false**

运行是时配置：否

core.klog.stderrthreshold
处于或超过此阈值的日志输出到 stderr。

默认： **2**

运行时可配置：是

core.klog.v
core.klog.v 是日志级别详细程度的值。

默认： **0**

运行时可配置：是

core.klog.vmodule
core.klog.vmodule 是文件过滤日志的、以逗号分隔的 **pattern=N** 设置列表。

默认： **空**

运行时可配置：是

9.4.2. sources

sources 部分包含特定于功能源的配置参数。

sources.cpu.cpuid.attributeBlacklist
防止发布此选项中列出的 **cpuid** 功能。

如果指定，则 **source.cpu.cpuid.attributeWhitelist** 将覆盖这个值。

默认：**[BMI1, BMI2, CLMUL, CMOV, CX16, ERMS, F16C, HTT, LZCNT, MMX, MMXEXT, NX, POPCNT, RDRAND, RDSEED, RDTSCP, SGX, SGXLC, SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSSE3]**

用法示例

```
sources:
  cpu:
    cpuid:
      attributeBlacklist: [MMX, MMXEXT]
```

sources.cpu.cpuid.attributeWhitelist
仅发布在此选项中列出的 **cpuid** 功能。

source.cpu.cpuid.attributeWhitelist 优先于 **source.cpu.cpuid.attributeBlacklist**。

默认： **空**

用法示例

```
sources:
  cpu:
    cpuid:
      attributeWhitelist: [AVX512BW, AVX512CD, AVX512DQ, AVX512F, AVX512VL]
```

sources.kernel.kconfigFile
source.kernel.kconfigFile 是内核配置文件的路径。如果为空，NFD 会在已知的标准位置运行搜索。

默认： **空**

用法示例

```
sources:
  kernel:
    kconfigFile: "/path/to/kconfig"
```

sources.kernel.configOpts
sources.kernel.configOpts 代表内核配置选项，作为功能标签发布。

默认：**[NO_HZ, NO_HZ_IDLE, NO_HZ_FULL, PREEMPT]**

用法示例

```
sources:
  kernel:
    configOpts: [NO_HZ, X86, DMI]
```

sources.pci.deviceClassWhitelist
sources.pci.deviceClassWhitelist 是用来发布标签的 PCI 设备类 ID 列表。它只能指定为主类（例如 **03**）或全类子类组合（例如 **0300**）。前者表示接受所有子类。可以使用 **deviceLabelFields** 进一步配置标签格式。

默认： **["03", "0b40", "12"]**

用法示例

```
sources:
  pci:
    deviceClassWhitelist: ["0200", "03"]
```

sources.pci.deviceLabelFields
sources.pci.deviceLabelFields 是构建功能标签名称时要使用的 PCI ID 字段集合。有效字段包括 **class**、**vendor**、**device**、**subsystem_vendor** 和 **subsystem_device**。

默认： **[class, vendor]**

用法示例

```
sources:
  pci:
    deviceLabelFields: [class, vendor, device]
```

在上例配置中，NFD 会发布标签，如 **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true**

sources.usb.deviceClassWhitelist
sources.usb.deviceClassWhitelist 是一个 USB 设备类 ID 列表，用于发布功能标签。可以使用 **deviceLabelFields** 进一步配置标签格式。

默认： **["0e", "ef", "fe", "ff"]**

用法示例

```
sources:
  usb:
    deviceClassWhitelist: ["ef", "ff"]
```

sources.usb.deviceLabelFields
sources.usb.deviceLabelFields 是一组 USB ID 字段，用于编写功能标签的名称。有效字段包括 **class**、**vendor** 和 **device**。

默认： **[class、vendor、device]**

用法示例

```
sources:
  pci:
    deviceLabelFields: [class, vendor]
```

使用上面的示例配置，NFD 会发布类似如下标签： **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true**。

sources.custom
sources.custom 是在自定义功能源中处理的规则列表，用于创建特定于用户的标签。

默认： **空**

用法示例

```
source:
  custom:
  - name: "my.custom.feature"
    matchOn:
    - loadedKMod: ["e1000e"]
    - pciId:
        class: ["0200"]
        vendor: ["8086"]
```

第 10 章 驱动程序工具包（DRIVER TOOLKIT）

了解驱动程序工具包以及如何将其用作驱动程序容器的基础镜像，以便在 Kubernetes 上启用特殊软件和硬件设备。

重要

Driver Toolkit 只是一个技术预览功能。技术预览功能不受红帽产品服务等级协议（SLA）支持，且功能可能并不完整。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的详情，请参考 https://access.redhat.com/support/offerings/techpreview/。

10.1. 关于驱动程序工具包

背景信息

Driver Toolkit 是 OpenShift Container Platform 有效负载中的一个容器镜像，用作可构建驱动程序容器的基础镜像。Driver Toolkit 镜像包含通常作为构建或安装内核模块的依赖项所需的内核软件包，以及驱动程序容器所需的一些工具。这些软件包的版本将与相应 OpenShift Container Platform 发行版本中的 Red Hat Enterprise Linux CoreOS（RHCOS）节点上运行的内核版本匹配。

驱动程序容器是容器镜像，用于在容器操作系统（如 RHCOS）上构建和部署树外内核模块和驱动程序。内核模块和驱动程序是在操作系统内核中具有高级别权限运行的软件库。它们扩展了内核功能，或者提供控制新设备所需的硬件特定代码。示例包括 Field Programmable Gate Arrays（FPGA）或 GPU 等硬件设备，以及软件定义型存储（SDS）解决方案（如 Lustre parallel 文件系统，它在客户端机器上需要内核模块）。驱动程序容器是用于在 Kubernetes 上启用这些技术的软件堆栈的第一层。

Driver Toolkit 中的内核软件包列表包括以下内容及其依赖项：

- **kernel-core**
- **kernel-devel**
- **kernel-headers**
- **kernel-modules**
- **kernel-modules-extra**

另外，Driver Toolkit 还包含相应的实时内核软件包：

- **kernel-rt-core**
- **kernel-rt-devel**
- **kernel-rt-modules**
- **kernel-rt-modules-extra**

Driver Toolkit 还有几个通常需要的工具来构建和安装内核模块，其中包括：

- **elfutils-libelf-devel**
- **kmod**

- **binutilskabi-dw**
- **kernel-abi-whitelists**
- 以上的依赖项

目的

在出现 Driver Toolkit 之前，您可以在 OpenShift Container Platform 中的一个 pod 中安装内核软件包，或在构建配置中使用 entitled builds，或从主机 **machine-os-content** 的内核 RPM 进行安装。Driver Toolkit 通过删除授权步骤简化了流程，并避免了访问 pod 中的 machine-os-content 特权操作。Driver Toolkit 也可以由有权访问预发布的 OpenShift Container Platform 版本的合作伙伴使用，用于未来的 OpenShift Container Platform 版本的硬件设备的预构建 driver-containers。

特殊资源 Operator（SRO）也使用 Driver Toolkit，目前作为 OperatorHub 上的社区 Operator 提供。SRO 支持树外和第三方内核驱动程序以及底层操作系统的支持软件。用户可以为 SRO 创建*配方*（*recipes*）来构建和部署驱动程序容器，以及支持诸如设备插件或指标的软件。配方可以包含构建配置，以基于 Driver Toolkit 构建驱动程序容器，或者 SRO 可以部署预构建驱动程序容器。

10.2. 拉取 DRIVER TOOLKIT 容器镜像

driver-toolkit 镜像包括在 Red Hat Ecosystem Catalog 的容器镜像部分 和 OpenShift Container Platform 发行版本有效负载中。与 OpenShift Container Platform 最新次要版本对应的镜像将标记为目录中的版本号。具体版本的镜像 URL 可使用 **oc adm** CLI 命令找到。

10.2.1. 从 registry.redhat.io 中拉取 Driver Toolkit 容器镜像

Red Hat Ecosystem Catalog 包括了使用 podman 或 OpenShift Container Platform 从 **registry.redhat.io** 中拉取 **driver-toolkit** 镜像的说明。最新次版本的 driver-toolkit 镜像将标记为 registry.redhat.io 中的次版本，如 **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.8**。

10.2.2. 在有效负载中查找驱动程序工具包镜像 URL

先决条件

- 您已从 Red Hat OpenShift Cluster Manager 获取了镜像 pull secret 。
- 已安装 OpenShift CLI（**oc**）。

流程

1. 可以使用 **oc adm** 命令从发行镜像中提取与特定发行版本对应的 **driver-toolkit** 镜像 URL：

```
$ oc adm release info 4.8.0 --image-for=driver-toolkit
```

输出示例

```
quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:0fd84aee79606178b6561ac71f8540f404d518ae5deff45f6d6ac8f02636c7f4
```

2. 可以使用有效的 pull secret 拉取此镜像，如安装 OpenShift Container Platform 所需的 pull secret。

```
$ podman pull --authfile=path/to/pullsecret.json quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:<SHA>
```

10.3. 使用 DRIVER TOOLKIT

例如，Driver Toolkit 可用作基础镜像来构建非常简单的内核模块，名为 simple-kmod。

10.3.1. 在集群中构建并运行 simple-kmod 驱动程序容器

先决条件

- OpenShift Container Platform 集群
- 安装 OpenShift CLI (**oc**) 。
- 以具有 **cluster-admin** 特权的用户身份登录。

流程

创建命名空间。例如：

```
$ oc new-project simple-kmod-demo
```

1. YAML 定义了 **ImageStream**，用于存储 **simple-kmod** 驱动程序容器镜像，以及用于构建容器的 **BuildConfig**。将此 YAML 保存为 **0000-buildconfig.yaml.template**。

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  labels:
    app: simple-kmod-driver-container
  name: simple-kmod-driver-container
  namespace: simple-kmod-demo
spec: {}
---
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: simple-kmod-driver-build
  name: simple-kmod-driver-build
  namespace: simple-kmod-demo
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
  triggers:
    - type: "ConfigChange"
    - type: "ImageChange"
  source:
    git:
      ref: "master"
      uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
    type: Git
```

```
    dockerfile: |
      FROM DRIVER_TOOLKIT_IMAGE

      WORKDIR /build/

      RUN yum -y install git make sudo gcc \
      && yum clean all \
      && rm -rf /var/cache/dnf

      # Expecting kmod software version as an input to the build
      ARG KMODVER

      # Grab the software from upstream
      RUN git clone https://github.com/openshift-psap/simple-kmod.git
      WORKDIR simple-kmod

      # Prep and build the module
      RUN make buildprep KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"
kernel-core) KMODVER=${KMODVER} \
      && make all       KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER} \
      && make install   KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER}

      # Add the helper tools
      WORKDIR /root/kvc-simple-kmod
      ADD Makefile .
      ADD simple-kmod-lib.sh .
      ADD simple-kmod-wrapper.sh .
      ADD simple-kmod.conf .
      RUN mkdir -p /usr/lib/kvc/ \
      && mkdir -p /etc/kvc/ \
      && make install

      RUN systemctl enable kmods-via-containers@simple-kmod
  strategy:
    dockerStrategy:
      buildArgs:
        - name: KMODVER
          value: DEMO
  output:
    to:
      kind: ImageStreamTag
      name: simple-kmod-driver-container:demo
```

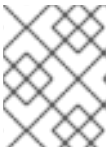
2. 在以下命令中，使用您运行的 OpenShift Container Platform 版本的相关的正确 driver toolki 镜像替换 "DRIVER_TOOLKIT_IMAGE" 部分。

```
$ OCP_VERSION=$(oc get clusterversion/version -ojsonpath={.status.desired.version})
```

```
$ DRIVER_TOOLKIT_IMAGE=$(oc adm release info $OCP_VERSION --image-for=driver-
toolkit)
```

```
$ sed "s#DRIVER_TOOLKIT_IMAGE#${DRIVER_TOOLKIT_IMAGE}#" 0000-
buildconfig.yaml.template > 0000-buildconfig.yaml
```

-



注意

OpenShift Container Platform 4.6 中引入了驱动程序工具包，其版本为 4.6.30，在 4.7 和 4.8 中的版本为 4.7.11。

3. 使用创建镜像流和构建配置

```
$ oc create -f 0000-buildconfig.yaml
```

4. 构建器 Pod 成功完成后，将驱动程序容器镜像部署为 **DaemonSet**。

   a. 驱动程序容器必须使用特权安全上下文运行，才能在主机上加载内核模块。以下 YAML 文件包含用于运行驱动程序容器的 RBAC 规则和 **DaemonSet**。将此 YAML 保存为 **1000-drivercontainer.yaml**。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: simple-kmod-driver-container
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: simple-kmod-driver-container
rules:
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  resourceNames:
  - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: simple-kmod-driver-container
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: simple-kmod-driver-container
subjects:
- kind: ServiceAccount
  name: simple-kmod-driver-container
userNames:
- system:serviceaccount:simple-kmod-demo:simple-kmod-driver-container
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: simple-kmod-driver-container
spec:
  selector:
```

```
    matchLabels:
      app: simple-kmod-driver-container
  template:
    metadata:
      labels:
        app: simple-kmod-driver-container
    spec:
      serviceAccount: simple-kmod-driver-container
      serviceAccountName: simple-kmod-driver-container
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/simple-kmod-
demo/simple-kmod-driver-container:demo
        name: simple-kmod-driver-container
        imagePullPolicy: Always
        command: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-containers@simple-kmod"]
        securityContext:
          privileged: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
```

b. 创建 RBAC 规则和守护进程集：

```
$ oc create -f 1000-drivercontainer.yaml
```

5. 当 pod 在 worker 节点上运行后，使用 **lsmod** 验证在主机机器上是否成功载入了 **simple_kmod** 内核模块。

a. 验证 pod 是否正在运行：

```
$ oc get pod -n simple-kmod-demo
```

输出示例

```
NAME                                READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-1-build    0/1     Completed   0          6m
simple-kmod-driver-container-b22fd  1/1     Running     0          40s
simple-kmod-driver-container-jz9vn  1/1     Running     0          40s
simple-kmod-driver-container-p45cc  1/1     Running     0          40s
```

b. 在驱动程序容器 pod 中执行 **lsmod** 命令：

```
$ oc exec -it pod/simple-kmod-driver-container-p45cc -- lsmod | grep simple
```

输出示例

```
simple_procfs_kmod     16384  0
simple_kmod            16384  0
```

第 11 章 根据对象限制规划您的环境

在规划 OpenShift Container Platform 集群时，请考虑以下对象限制。

这些限制基于最大可能的集群。对于较小的集群，最大值限制会较低。很多因素会影响指定的阈值，包括 etcd 版本或者存储数据格式。

重要

这些指南适用于带有软件定义网络（SDN）而不是开放虚拟网络（OVN）的 OpenShift Container Platform。

在大多数情况下，超过这些限制会降低整体性能。它不一定意味着集群会出现错误。

11.1. OPENSHIFT CONTAINER PLATFORM 为主发行版本测试了集群最大值

为 OpenShift Container Platform 3.x 测试的云平台： Red Hat OpenStack Platform（RHOSP）、Amazon Web Services 和 Microsoft Azure。为 OpenShift Container Platform 4.x 测试的云平台：Amazon Web Services、 Microsoft Azure 和 Google Cloud Platform。

最大类型	3.x 测试的最大值	4.x 测试的最大值
节点数	2,000	2,000
pod 数量 [1]	150,000	150,000
每个节点的 pod 数量	250	500 [2]
每个内核的 pod 数量	没有默认值。	没有默认值。
命名空间数量 [3]	10,000	10,000
构建（build）数	10,000（默认 pod RAM 512 Mi）- 管道 (Pipeline) 策略	10,000（默认 pod RAM 512 Mi）- Source-to-Image (S2I) 构建策略
每个命名空间的 pod 数量 [4]	25,000	25,000
每个 Ingress Controller 的路由和后端数量	每个路由器 2,000 个	每个路由器 2,000 个
secret 的数量	80,000	80,000
配置映射数量	90,000	90,000
服务数量 [5]	10,000	10,000
每个命名空间的服务数	5,000	5,000

最大类型	3.x 测试的最大值	4.x 测试的最大值
每个服务中的后端数	5,000	5,000
每个命名空间的部署数量 [4]	2,000	2,000
构建配置数	12,000	12,000
自定义资源定义 (CRD) 的数量	没有默认值。	512 [6]

1. 这里的 pod 数量是 test pod 的数量。实际的 pod 数量取决于应用程序的内存、CPU 和存储要求。

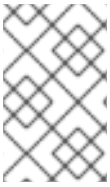
2. 这在一个有 100 个 work 节点，每个 worker 节点有 500 个 pod 的集群中测试。默认 **maxPods** 仍为 250。要获得 500 **maxPods**，则必须使用自定义 kubelet 配置将 **maxPods** 设置为 **500** 来创建集群。如果需要 500 个用户 Pod，则需要 **hostPrefix** 为 **22**，因为节点上已经运行了 10-15 个系统 pod。带有 Persistent VolumeClaim (PVC) 的最大 pod 数量取决于分配 PVC 的后端存储。在我们的测试中，只有 OpenShift Container Storage (OCS v4) 能够满足本文档中提到的每个节点的 pod 数量。

3. 当有大量活跃的项目时，如果键空间增长过大并超过空间配额，etcd 的性能将会受到影响。强烈建议您定期维护 etcd 存储（包括整理碎片）来释放 etcd 存储。

4. 系统中有一些控制循环，它们必须对给定命名空间中的所有对象进行迭代，以作为对一些状态更改的响应。在单一命名空间中有大量给定类型的对象可使这些循环的运行成本变高，并降低对给定状态变化的处理速度。限制假设系统有足够的 CPU 、内存和磁盘来满足应用程序的要求。

5. 每个服务端口和每个服务后端在 iptables 中都有对应条目。给定服务的后端数量会影响端点对象的大小，这会影响到整个系统发送的数据大小。

6. OpenShift Container Platform 的限制是 512 个总自定义资源定义(CRD)，其中包括由 OpenShift Container Platform 安装的产品、与 OpenShift Container Platform 集成并创建了 CRD 的产品。如果创建超过 512 CRD，则 **oc** 命令请求可能会节流。



注意

红帽不提供针对 OpenShift Container Platform 集群大小调整的直接指导。这是因为，判断集群是否在 OpenShift Container Platform 支持的边界内，需要仔细考虑限制集群扩展的所有多维因素。

11.2. 测试集群最大值的 OPENSHIFT CONTAINER PLATFORM 环境和配置

AWS 云平台：

节点	Flavor	vCPU	RAM(GiB)	磁盘类型	磁盘大小(GiB)/IOS	数量	区域
Master/etcd [1]	r5.4xlarge	16	128	gp3	220	3	us-west-2

节点	Flavor	vCPU	RAM(GiB)	磁盘类型	磁盘大小(GiB)/IOS	数量	区域
Infra [2]	m5.12xlarge	48	192	gp3	100	3	us-west-2
Workload [3]	m5.4xlarge	16	64	gp3	500 [4]	1	us-west-2
Worker	m5.2xlarge	8	32	gp3	100	3/25/250/500 [5]	us-west-2

1. 带有基准性能为 3000 IOPS 和 125 MiB 每秒的 gp3 磁盘用于 control plane/etcd 节点，因为 etcd 对延迟敏感。gp3 卷不使用突发性能。
2. Infra 节点用于托管 Monitoring、Ingress 和 Registry 组件，以确保它们有足够资源可大规模运行。
3. 工作负载节点专用于运行性能和可扩展工作负载生成器。
4. 使用更大的磁盘，以便有足够的空间存储在运行性能和可扩展性测试期间收集的大量数据。
5. 在迭代中扩展了集群，且性能和可扩展性测试是在指定节点数中执行的。

IBM Power 系统平台：

节点	vCPU	RAM(GiB)	磁盘类型	磁盘大小(GiB)/IOS	数量
Master/etcd [1]	16	32	io1	每个 GiB 120 / 10 IOPS	3
Infra [2]	16	64	gp2	120	2
Workload [3]	16	256	gp2	120 [4]	1
Worker	16	64	gp2	120	3/25/250/500 [5]

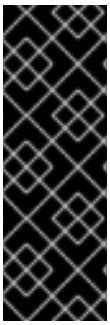
1. 带有 120 / 3 IOPS 的 io1 磁盘用于 master/etcd 节点，因为 etcd 非常大，且敏感延迟。
2. Infra 节点用于托管 Monitoring、Ingress 和 Registry 组件，以确保它们有足够资源可大规模运行。
3. 工作负载节点专用于运行性能和可扩展工作负载生成器。
4. 使用更大的磁盘，以便有足够的空间存储在运行性能和可扩展性测试期间收集的大量数据。
5. 在迭代中扩展了集群，且性能和可扩展性测试是在指定节点数中执行的。

11.2.1. IBM Z 平台

节点	vCPU [4]	RAM(GiB)[5]	磁盘类型	磁盘大小 (GiB)/IOS	数量
Control plane/etcd [1,2]	8	32	ds8k	300 / LCU 1	3
Compute [1,3]	8	32	ds8k	150 / LCU 2	4 节点（每个节点扩展到 100/250/500 pod）

1. 节点在两个逻辑控制单元 (LCU) 之间分发，以优化 control plane/etcd 节点的磁盘 I/O 负载，因为 etcd 非常大，且对延迟敏感。etcd I/O 需求不应干扰其他工作负载。
2. 四个计算节点用于运行同时具有 100/250/500 pod 的多个迭代的测试。首先，使用闲置 pod 来评估 pod 是否可以实例。接下来，使用网络和 CPU 要求客户端/服务器工作负载来评估系统在压力下的稳定性。客户端和服务器 pod 是部署范围，每个对分布在两个计算节点上。
3. 没有单独的工作负载节点。工作负载在两个计算节点之间模拟微服务工作负载。
4. 使用的物理处理器数量是 6 个用于 Linux (IFL)的集成设施。
5. 使用的总物理内存为 512 GiB。

11.3. 如何根据经过测试的集群限制规划您的环境



重要

在节点中过度订阅物理资源会影响在 pod 放置过程中对 Kubernetes 调度程序的资源保证。了解可以采取什么措施避免内存交换。

某些限制只在单一维度中扩展。当很多对象在集群中运行时，它们会有所不同。

本文档中给出的数字基于红帽的测试方法、设置、配置和调整。这些数字会根据您自己的设置和环境而有所不同。

在规划您的环境时，请确定每个节点会运行多少 个 pod ：

```
required pods per cluster / pods per node = total number of nodes needed
```

每个节点上的 Pod 数量做多为 250。而在某个节点中运行的 pod 的具体数量取决于应用程序本身。请参阅 *如何根据应用程序要求规划您的环境* 中的内容来计划应用程序的内存、CPU 和存储要求。

示例情境

如果您计划把集群的规模限制在有 2200 个 pod，则需要至少有五个节点，假设每个节点最多有 500 个 pod：

```
2200 / 500 = 4.4
```

如果将节点数量增加到 20，那么 pod 的分布情况将变为每个节点有 110 个 pod：

```
2200 / 20 = 110
```

其中：

```
required pods per cluster / total number of nodes = expected pods per node
```

11.4. 如何根据应用程序要求规划您的环境

考虑应用程序环境示例：

pod 类型	pod 数量	最大内存	CPU 内核	持久性存储
Apache	100	500 MB	0.5	1 GB
node.js	200	1 GB	1	1 GB
postgresql	100	1 GB	2	10 GB
JBoss EAP	100	1 GB	1	1 GB

推断的要求: 550 个 CPU 内核、450GB RAM 和 1.4TB 存储。

根据您的具体情况，节点的实例大小可以被增大或降低。在节点上通常会使用资源过度分配。在这个部署场景中，您可以选择运行多个额外的较小节点，或数量更少的较大节点来提供同样数量的资源。在做出决定前应考虑一些因素，如操作的灵活性以及每个实例的成本。

节点类型	数量	CPU	RAM (GB)
节点（选择 1）	100	4	16
节点（选择 2）	50	8	32
节点（选择 3）	25	16	64

有些应用程序很适合于过度分配的环境，有些则不适合。大多数 Java 应用程序以及使用巨页的应用程序都不允许使用过度分配功能。它们的内存不能用于其他应用程序。在上面的例子中，环境大约会出现 30% 过度分配的情况，这是一个常见的比例。

应用程序 pod 可以使用环境变量或 DNS 访问服务。如果使用环境变量，当 pod 在节点上运行时，对于每个活跃服务，则 kubelet 的变量都会注入。集群感知 DNS 服务器监视 Kubernetes API 提供了新服务，并为每个服务创建一组 DNS 记录。如果整个集群中启用了 DNS，则所有 pod 都应自动根据其 DNS 名称解析服务。如果您必须超过 5000 服务，可以使用 DNS 进行服务发现。当使用环境变量进行服务发现时，参数列表超过了命名空间中 5000 服务后允许的长度，则 pod 和部署将失败。要解决这个问题，请禁用部署的服务规格文件中的服务链接：

```
---
apiVersion: template.openshift.io/v1
```

```
kind: Template
metadata:
  name: deployment-config-template
  creationTimestamp:
  annotations:
    description: This template will create a deploymentConfig with 1 replica, 4 env vars and a service.
    tags: ''
objects:
- apiVersion: apps.openshift.io/v1
  kind: DeploymentConfig
  metadata:
    name: deploymentconfig${IDENTIFIER}
  spec:
    template:
      metadata:
        labels:
          name: replicationcontroller${IDENTIFIER}
      spec:
        enableServiceLinks: false
        containers:
        - name: pause${IDENTIFIER}
          image: "${IMAGE}"
          ports:
          - containerPort: 8080
            protocol: TCP
          env:
          - name: ENVVAR1_${IDENTIFIER}
            value: "${ENV_VALUE}"
          - name: ENVVAR2_${IDENTIFIER}
            value: "${ENV_VALUE}"
          - name: ENVVAR3_${IDENTIFIER}
            value: "${ENV_VALUE}"
          - name: ENVVAR4_${IDENTIFIER}
            value: "${ENV_VALUE}"
          resources: {}
          imagePullPolicy: IfNotPresent
          capabilities: {}
          securityContext:
            capabilities: {}
            privileged: false
        restartPolicy: Always
        serviceAccount: ''
    replicas: 1
    selector:
      name: replicationcontroller${IDENTIFIER}
    triggers:
    - type: ConfigChange
    strategy:
      type: Rolling
- apiVersion: v1
  kind: Service
  metadata:
    name: service${IDENTIFIER}
  spec:
    selector:
      name: replicationcontroller${IDENTIFIER}
```

```
    ports:
    - name: serviceport${IDENTIFIER}
      protocol: TCP
      port: 80
      targetPort: 8080
    clusterIP: ''
    type: ClusterIP
    sessionAffinity: None
  status:
    loadBalancer: {}
parameters:
- name: IDENTIFIER
  description: Number to append to the name of resources
  value: '1'
  required: true
- name: IMAGE
  description: Image to use for deploymentConfig
  value: gcr.io/google-containers/pause-amd64:3.0
  required: false
- name: ENV_VALUE
  description: Value to use for environment variables
  generate: expression
  from: "[A-Za-z0-9]{255}"
  required: false
labels:
  template: deployment-config-template
```

可在命名空间中运行的应用程序 pod 数量取决于服务数量以及环境变量用于服务发现时的服务名称长度。系统中的**ARG_MAX** 为新进程定义最大参数长度，默认设置为 **2097152 KiB**。Kubelet 将环境变量注入到要在命名空间中运行的每个 pod 中，包括：

- **<SERVICE_NAME>_SERVICE_HOST=<IP>**
- **<SERVICE_NAME>_SERVICE_PORT=<PORT>**
- **<SERVICE_NAME>_PORT=tcp://<IP>:<PORT>**
- **<SERVICE_NAME>_PORT_<PORT>_TCP=tcp://<IP>:<PORT>**
- **<SERVICE_NAME>_PORT_<PORT>_TCP_PROTO=tcp**
- **<SERVICE_NAME>_PORT_<PORT>_TCP_PORT=<PORT>**
- **<SERVICE_NAME>_PORT_<PORT>_TCP_ADDR=<ADDR>**

如果参数长度超过允许的值，服务名称中的字符数会受到影响，命名空间中的 pod 将开始失败。例如，在一个带有 5000 服务的命名空间中，服务名称的限制为 33 个字符，它可让您在命名空间中运行 5000 个 Pod。

第 12 章 优化存储

优化存储有助于最小化所有资源中的存储使用。通过优化存储，管理员可帮助确保现有存储资源以高效的方式工作。

12.1. 可用的持久性存储选项

了解持久性存储选项，以便可以优化 OpenShift Container Platform 环境。

表 12.1. 可用存储选项

存储类型	描述	例子
Block	• 在操作系统 (OS) 中作为块设备 • 适用于需要完全控制存储，并绕过文件系统在低层直接操作文件的应用程序 • 也称为存储区域网络 (SAN) • 不可共享，这意味着，每次只有一个客户端可以挂载这种类型的端点	AWS EBS 和 VMware vSphere 支持在 OpenShift Container Platform 中的原生动态持久性卷 (PV)置备 。
File	• 在 OS 中作为要挂载的文件系统导出 • 也称为网络附加存储（Network Attached Storage，NAS） • 取决于不同的协议、实现、厂商及范围，其并行性、延迟、文件锁定机制和其它功能可能会有很大不同。	RHEL NFS、NetApp NFS [1] 和供应商 NFS
对象	• 通过 REST API 端点访问 • 可配置用于 OpenShift Container Platform Registry • 应用程序必须在应用程序和（/或）容器中构建其驱动程序。	AWS S3

1. NetApp NFS 在使用 Trident 插件时支持动态 PV 置备。

重要

目前，OpenShift Container Platform 4.8 不支持 CNS。

12.2. 推荐的可配置存储技术

下表总结了为给定的 OpenShift Container Platform 集群应用程序推荐的可配置存储技术。

表 12.2. 推荐的、可配置的存储技术

存储类型	ROX[1]	RWX[2]	Registry	扩展的 registry	Metrics[3]	日志记录	Apps
Block	Yes[4]	否	可配置	无法配置	推荐的	推荐的	推荐的
File	Yes[4]	是	可配置	可配置	Configurable[5]	Configurable[6]	推荐的
对象	是	是	推荐的	推荐的	无法配置	无法配置	Not configurable[7]

[1] **ReadOnlyMany**

[2] **ReadWriteMany**

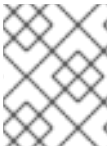
[3] **Prometheus 是用于指标数据的底层技术。**

[4] **这不适用于物理磁盘、虚拟机物理磁盘、VMDK、NFS 回送、AWS EBS 和 Azure 磁盘。**

[5] **对于指标数据，使用 ReadWriteMany (RWX) 访问模式的文件存储是不可靠的。如果使用文件存储，请不要在配置用于指标数据的持久性卷声明（PVC）上配置 RWX 访问模式。**

[6] **用于日志记录，使用任何共享存储都将是一个反模式。每个 elasticsearch 都需要一个卷。**

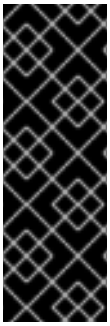
[7] **对象存储不会通过 OpenShift Container Platform 的 PV 或 PVC 使用。应用程序必须与对象存储 REST API 集成。**



注意

扩展的容器镜像仓库（registry）是一个 OpenShift Container Platform 容器镜像仓库，它有两个或更多个 pod 运行副本。

12.2.1. 特定应用程序存储建议



重要

测试显示在 Red Hat Enterprise Linux(RHEL)中使用 NFS 服务器作为核心服务的存储后端的问题。这包括 OpenShift Container Registry 和 Quay，Prometheus 用于监控存储，以及 Elasticsearch 用于日志存储。因此，不建议使用 RHEL NFS 作为 PV 后端用于核心服务。

市场上的其他 NFS 实现可能没有这些问题。如需了解更多与此问题相关的信息，请联络相关的 NFS 厂商。

12.2.1.1. Registry

在一个非扩展的/高可用性 (HA) OpenShift Container Platform registry 集群部署中：

- 存储技术不需要支持 RWX 访问模式。

- 存储技术必须保证读写一致性。
- 首选存储技术是对象存储，然后是块存储。
- 对于应用于生产环境工作负载的 OpenShift Container Platform Registry 集群部署，我们不推荐使用文件存储。

12.2.1.2. 扩展的 registry

在扩展的/HA OpenShift Container Platform registry 集群部署中：

- 存储技术必须支持 RWX 访问模式。
- 存储技术必须保证读写一致性。
- 首选存储技术是对象存储。
- 支持 Amazon Simple Storage Service(Amazon S3)、Google Cloud Storage(GCS)、Microsoft Azure Blob Storage 和 OpenStack Swift。
- 对象存储应该兼容 S3 或 Swift。
- 对于非云平台，如 vSphere 和裸机安装，唯一可配置的技术是文件存储。
- 块存储是不可配置的。

12.2.1.3. 指标

在 OpenShift Container Platform 托管的 metrics 集群部署中：

- 首选存储技术是块存储。
- 对象存储是不可配置的。

重要

在带有生产环境负载的托管 metrics 集群部署中不推荐使用文件存储。

12.2.1.4. 日志记录

在 OpenShift Container Platform 托管的日志集群部署中：

- 首选存储技术是块存储。
- 对象存储是不可配置的。

12.2.1.5. 应用程序

应用程序的用例会根据不同应用程序而不同，如下例所示：

- 支持动态 PV 部署的存储技术的挂载时间延迟较低，且不与节点绑定来支持一个健康的集群。
- 应用程序开发人员需要了解应用程序对存储的要求，以及如何与所需的存储一起工作以确保应用程序扩展或者与存储层交互时不会出现问题。

12.2.2. 其他特定的应用程序存储建议

重要

不建议在 **Write** 密集型工作负载（如 **etcd** ）中使用 RAID 配置。如果您使用 RAID 配置运行 **etcd**，您可能会遇到工作负载性能问题的风险。

- Red Hat OpenStack Platform（RHOSP）Cinder: RHOSP Cinder 倾向于在 ROX 访问模式用例中使用。
- 数据库：数据库（RDBMS 、 nosql DBs 等等）倾向于使用专用块存储来获得最好的性能。
- etcd 数据库必须具有足够的存储和适当的性能容量才能启用大型集群。有关监控和基准测试工具的信息，以建立基本存储和高性能环境，请参阅 *推荐 etcd 实践*。

12.3. 数据存储管理

下表总结了 OpenShift Container Platform 组件写入数据的主要目录。

表 12.3. 用于存储 OpenShift Container Platform 数据的主目录

目录	备注	大小	预期增长
/var/lib/etcd	用于存储数据库的 etcd 存储。	小于 20 GB。 数据库可增大到 8 GB。	随着环境增长会缓慢增长。只存储元数据。 每多加 8 GB 内存需要额外 20-25 GB。
/var/lib/containers	这是 CRI-O 运行时的挂载点。用于活跃容器运行时的存储，包括 Pod 和本地镜像存储。不适用于 registry 存储。	有 16 GB 内存的节点需要 50 GB。请注意，这个大小不应该用于决定最小集群要求。 每多加 8 GB 内存需要额外 20-25 GB。	增长受运行容器容量的限制。
/var/lib/kubelet	pod 的临时卷（Ephemeral volume）存储。这包括在运行时挂载到容器的任何外部存储。包括环境变量、kube secret 和不受持久性卷支持的数据卷。	可变	如果需要存储的 pod 使用持久性卷，则最小。如果使用临时存储，可能会快速增长。
/var/log	**所有组件的日志文件。**	**10 到 30 GB。**	**日志文件可能会快速增长；大小可以通过增加磁盘或使用日志轮转来管理。**

第 13 章 优化路由

OpenShift Container Platform HAProxy 路由器扩展以优化性能。

13.1. INGRESS CONTROLLER（ROUTER）性能的基线

OpenShift Container Platform Ingress Controller，或称为路由器，是所有用于 OpenShift Container Platform 服务的外部流量的入站点。

当根据每秒处理的 HTTP 请求来评估单个 HAProxy 路由器性能时，其性能取决于多个因素。特别是：

- HTTP keep-alive/close 模式
- 路由类型
- 对 TLS 会话恢复客户端的支持
- 每个目标路由的并行连接数
- 目标路由数
- 后端服务器页面大小
- 底层基础结构（网络/SDN 解决方案、CPU 等）

具体环境中的性能会有所不同，红帽实验室在一个有 4 个 vCPU/16GB RAM 的公共云实例中进行测试。一个 HAProxy 路由器处理由后端终止的 100 个路由服务提供 1kB 静态页面，每秒处理以下传输数。

在 HTTP 的 keep-alive 模式下：

Encryption	LoadBalancerService	HostNetwork
none	21515	29622
edge	16743	22913
passthrough	36786	53295
re-encrypt	21583	25198

在 HTTP 关闭（无 keep-alive）情境中：

Encryption	LoadBalancerService	HostNetwork
none	5719	8273
edge	2729	4069
passthrough	4121	5344

Encryption	LoadBalancerService	HostNetwork
re-encrypt	2320	2941

默认 Ingress Controller 配置使用 **ROUTER_THREADS=4**，并测试了两个不同的端点发布策略 (LoadBalancerService/hostnetwork) 。TLS 会话恢复用于加密路由。使用 HTTP keep-alive 设置，单个 HAProxy 路由器可在页面大小小到 8 kB 时充满 1 Gbit NIC。

当在使用现代处理器的裸机中运行时，性能可以期望达到以上公共云实例测试性能的大约两倍。这个开销是由公有云的虚拟化层造成的，基于私有云虚拟化的环境也会有类似的开销。下表是有关在路由器后面的应用程序数量的指导信息：

应用程序数量	应用程序类型
5-10	静态文件/web 服务器或者缓存代理
100-1000	生成动态内容的应用程序

取决于所使用的技术，HAProxy 通常可支持 5 到 1000 个程序的路由。Ingress Controller 性能可能会受其后面的应用程序的能力和性能的限制，如使用的语言，静态内容或动态内容。

如果有多个服务于应用程序的 Ingress 或路由器，则应该使用路由器分片（router sharding）以帮助横向扩展路由层。

如需有关 Ingress 分片的更多信息，请参阅使用路由标签和使用命名空间标签配置 Ingress Controller 分片。

13.2. INGRESS CONTROLLER（路由器）性能优化

OpenShift Container Platform 不再支持通过设置以下环境变量来修改 Ingress Controller 的部署：**ROUTER_THREADS**、**ROUTER_DEFAULT_TUNNEL_TIMEOUT**、**ROUTER_DEFAULT_CLIENT_TIMEOUT**、**ROUTER_DEFAULT_SERVER_TIMEOUT** 和 **RELOAD_INTERVAL**。

您可以修改 Ingress Controller 的部署，但当 Ingress Operator 被启用时，其配置会被覆盖。

第 14 章 优化网络

OpenShift SDN 使用 OpenvSwitch、虚拟可扩展 LAN（VXLAN）隧道、OpenFlow 规则和 iptables。这个网络可以通过使用 jumbo 帧、网络接口控制器 (NIC) 卸载、多队列和 ethtool 设置来调节。

OVN-Kubernetes 使用 Geneve（通用网络虚拟化封装）而不是 VXLAN 作为隧道协议。

VXLAN 提供通过 VLAN 的好处，比如网络从 4096 增加到一千六百万，以及跨物理网络的第 2 层连接。这允许服务后的所有 pod 相互通信，即使它们在不同系统中运行也是如此。

VXLAN 在用户数据报协议（UDP）数据包中封装所有隧道流量。但是，这会导致 CPU 使用率增加。这些外部数据包和内部数据包集都遵循常规的校验规则，以保证在传输过程中不会损坏数据。根据 CPU 性能，这种额外的处理开销可能会降低吞吐量，与传统的非覆盖网络相比会增加延迟。

云、虚拟机和裸机 CPU 性能可以处理很多 Gbps 网络吞吐量。当使用高带宽链接（如 10 或 40 Gbps）时，性能可能会降低。基于 VXLAN 的环境里存在一个已知问题，它并不适用于容器或 OpenShift Container Platform。由于 VXLAN 的实现，任何依赖于 VXLAN 隧道的网络都会有相似的性能。

如果您希望超过 Gbps，可以：

- 试用采用不同路由技术的网络插件，比如边框网关协议（BGP）。
- 使用 VXLAN-offload 功能的网络适配器。VXLAN-offload 将数据包校验和相关的 CPU 开销从系统 CPU 移动到网络适配器的专用硬件中。这会释放 Pod 和应用程序使用的 CPU 周期，并允许用户利用其网络基础架构的全部带宽。

VXLAN-offload 不会降低延迟。但是，即使延迟测试也会降低 CPU 使用率。

14.1. 为您的网络优化 MTU

有两个重要的最大传输单元 (MTU)：网络接口控制器 (NIC) MTU 和集群网络 MTU。

NIC MTU 仅在 OpenShift Container Platform 安装时进行配置。MTU 必须小于或等于您网络 NIC 的最大支持值。如果您要优化吞吐量，请选择最大可能的值。如果您要优化最小延迟，请选择一个较低值。

SDN 覆盖的 MTU 必须至少小于 NIC MTU 50 字节。此帐户用于 SDN overlay 标头。因此，在一个普通以太网网络中，将其设置为 **1450**。在 jumbo 帧以太网网络中，将其设置为 **8950**。

对于 OVN 和 Geneve，MTU 必须至少小于 NIC MTU 100 字节。

注意

这个 50 字节覆 overlay 头与 OpenShift SDN 相关。其他 SDN 解决方案可能需要该值更大或更少。

14.2. 安装大型集群的实践建议

在安装大型集群或将现有的集群扩展到较大规模时，请在安装集群在 **install-config.yaml** 文件中相应地设置集群网络 **cidr** ：

```
networking:
  clusterNetwork:
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  machineCIDR: 10.0.0.0/16
```

```
networkType: OpenShiftSDN
serviceNetwork:
- 172.30.0.0/16
```

如果集群的节点数超过 500 个，则无法使用默认的集群网络 **cidr 10.128.0.0/14**。在这种情况下，必须将其设置为 **10.128.0.0/12** 或 **10.128.0.0/10**，以支持超过 500 个节点的环境。

14.3. IPSEC 的影响

因为加密和解密节点主机使用 CPU 电源，所以启用加密时，无论使用的 IP 安全系统是什么，性能都会影响节点上的吞吐量和 CPU 使用量。

IPsec 在到达 NIC 前，会在 IP 有效负载级别加密流量，以保护用于 NIC 卸载的字段。这意味着，在启用 IPSec 时，一些 NIC 加速功能可能无法使用，并可能导致吞吐量降低并增加 CPU 用量。

其他资源

- 修改高级网络配置参数
- OVN-Kubernetes 网络供应商的配置参数
- OpenShift SDN 网络供应商的配置参数

第 15 章 管理裸机主机

在裸机集群中安装 OpenShift Container Platform 时，您可以使用**机器（machine）**和**机器集（machineset）**自定义资源（CR）为集群中存在的裸机主机置备和管理裸机节点。

15.1. 关于裸机主机和节点

要将 Red Hat Enterprise Linux CoreOS（RHCOS）裸机主机置备为集群中的节点，首先创建一个与裸机主机硬件对应的 **MachineSet** 自定义资源（CR）对象。裸机主机机器集描述了特定于您的配置的基础架构组件。将特定的 Kubernetes 标签应用于这些机器集，然后将基础架构组件更新为仅在那些机器上运行。

当您扩展包含 **metal3.io/autoscale-to-hosts** 注解的相关 **MachineSet** 时，**Machine** CR 会被自动创建。OpenShift Container Platform 使用 **Machine** CR 来置备与 **MachineSet** CR 中指定的主机对应的裸机节点。

15.2. 维护裸机主机

您可从 OpenShift Container Platform Web 控制台维护集群中的裸机主机详情。导航到 **Compute → Bare Metal Hosts**，然后从 **Actions** 下拉菜单中选择一个任务。您可以在此处管理诸如 BMC 详情、主机的引导 MAC 地址、启用电源管理等项目。您还可以查看主机的网络接口和驱动器详情。

您可以将裸机主机移入维护模式。当您将主机移入维护模式时，调度程序会将所有受管工作负载从对应的裸机节点中移出。在处于维护模式时不会调度新的工作负载。

您可以在 web 控制台中取消置备裸机主机。取消置备主机执行以下操作：

1. 使用 **cluster.k8s.io/delete-machine: true**注解裸机主机 CR
2. 缩减相关的机器集

注意

在不先将守护进程集和未管理的静态 pod 移动到另一节点的情况下，关闭主机电源可能会导致服务中断和数据丢失。

其他资源

- 在裸机中添加计算机器

15.2.1. 使用 web 控制台在集群中添加裸机主机

您可以在 web 控制台中在集群中添加裸机主机。

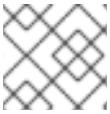
先决条件

- 在裸机上安装 RHCOS 集群。
- 以具有 **cluster-admin** 权限的用户身份登录。

流程

1. 在 web 控制台中，导航到 **Compute → Bare Metal Hosts**。

2. 选择 **Add Host → New with Dialog**。
3. 为新的裸机主机指定唯一名称。
4. 设置 **引导 MAC 地址**。
5. 设置 **基板管理控制台（BMC）地址**.
6. 可选：为主机启用电源管理。这允许 OpenShift Container Platform 控制主机的电源状态。
7. 输入主机的基板管理控制器（BMC）的用户凭据。
8. 选择在创建后打开主机电源，然后选择 **Create**。
9. 向上扩展副本数，以匹配可用的裸机主机数量。导航到 **Compute → MachineSets**，然后从 **Actions** 下拉菜单中选择 **Edit Machine count** 来增加集群中的机器副本数量。



注意

您还可以使用 **oc scale** 命令和适当的裸机机器集来管理裸机节点的数量。

15.2.2. 在 web 控制台中使用 YAML 在集群中添加裸机主机

您可以使用描述裸机主机的 YAML 文件在 web 控制台中在集群中添加裸机主机。

先决条件

- 在裸机基础架构上安装 RHCOS 计算机器，以便在集群中使用。
- 以具有 **cluster-admin** 权限的用户身份登录。
- 为裸机主机创建 **Secret** CR。

流程

1. 在 web 控制台中，导航到 **Compute → Bare Metal Hosts**。
2. 选择 **Add Host → New from YAML**。
3. 复制并粘贴以下 YAML，使用您的主机详情修改相关字段：

```
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: <bare_metal_host_name>
spec:
  online: true
  bmc:
    address: <bmc_address>
    credentialsName: <secret_credentials_name>  (1)
    disableCertificateVerification: True
  bootMACAddress: <host_boot_mac_address>
  hardwareProfile: unknown
```

(1) (1) (1) **credentialsName** 必须引用有效的 **Secret** CR。如果在 **credentialsName** 中没有引用有效的 **Secret**，则 **baremetal-operator** 无法管理裸机主机。如需有关 secret 以及如何创建 secret 的更多信息，请参阅了解 secret。

4. 选择 **Create** 以保存 YAML 并创建新的裸机主机。

5. 向上扩展副本数，以匹配可用的裸机主机数量。导航到 **Compute → MachineSets**，然后从 **Actions** 下拉菜单中选择 **Edit Machine count** 来增加集群中的机器数量。

注意

您还可以使用 **oc scale** 命令和适当的裸机机器集来管理裸机节点的数量。

15.2.3. 自动将机器扩展到可用的裸机主机数量

要自动创建与可用 **BareMetalHost** 对象数量匹配的 **Machine** 对象数量，请在 **MachineSet** 对象中添加 **metal3.io/autoscale-to-hosts** 注解。

先决条件

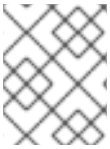
- 安装 RHCOS 裸机计算机器以在集群中使用，并创建对应的 **BareMetalHost** 对象。
- 安装 OpenShift Container Platform CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录。

流程

1. 通过添加 **metal3.io/autoscale-to-hosts** 注解来注解您要配置的用于自动扩展的机器集。将 **<machineset>** 替换为机器集的名称。

```
$ oc annotate machineset <machineset> -n openshift-machine-api 'metal3.io/autoscale-to-hosts=<any_value>'
```

等待新的缩放计算机启动。



注意

当您使用 **BareMetalHost** 对象在集群中创建机器时，**BareMetalHost** 上更改了标签或选择器，**BareMetalHost** 对象仍然会根据创建 **Machine** 对象的 **MachineSet** 进行计数。

15.2.4. 从 provisioner 节点中删除裸机主机

在某些情况下，您可能想要从 provisioner 节点临时删除裸机主机。例如，在使用 OpenShift Container Platform 管理控制台或机器配置池更新触发裸机主机重启时，OpenShift Container Platform 日志会登录到集成的 Dell Remote Access Controller (iDrac)，并发出删除作业队列。

要防止管理与可用 **BareMetalHost** 对象数量匹配的 **Machine** 对象数量，请在 **MachineSet** 对象中添加 **baremetalhost.metal3.io/detached** 注解。

注意

这个注解只适用于处于 **Provisioned**, **ExternallyProvisioned** 或 **Ready/Available** 状态的 **BareMetalHost** 对象。

先决条件

- 安装 RHCOS 裸机计算机器以在集群中使用，并创建对应的 **BareMetalHost** 对象。
- 安装 OpenShift Container Platform CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录。

流程

1. 通过添加 **baremetalhost.metal3.io/detached** 注解来注解您要从 provisioner 节点中删除的计算机器集。

```
$ oc annotate machineset <machineset> -n openshift-machine-api
'baremetalhost.metal3.io/detached'
```

等待新机器启动。

注意

当您使用 **BareMetalHost** 对象在集群中创建机器时，**BareMetalHost** 上更改了标签或选择器，**BareMetalHost** 对象仍然会根据创建 **Machine** 对象的 **MachineSet** 进行计数。

2. 在置备用例中，使用以下命令在重启完成后删除注解：

```
$ oc annotate machineset <machineset> -n openshift-machine-api
'baremetalhost.metal3.io/detached-'
```

其他资源

- 扩展集群
- 裸机上的 MachineHealthCheck

第 16 章 巨页的作用及应用程序如何使用它们

16.1. 巨页的作用

内存在块（称为页）中进行管理。在大多数系统中，页的大小为 4Ki。1Mi 内存相当于 256 个页，1Gi 内存相当于 256,000 个页。CPU 有内置的内存管理单元，可在硬件中管理这些页的列表。Translation Lookaside Buffer (TLB) 是虚拟页到物理页映射的小型硬件缓存。如果在硬件指令中包括的虚拟地址可以在 TLB 中找到，则其映射信息可以被快速获得。如果没有包括在 TLN 中，则称为 TLB miss。系统将会使用基于软件的，速度较慢的地址转换机制，从而出现性能降低的问题。因为 TLB 的大小是固定的，因此降低 TLB miss 的唯一方法是增加页的大小。

巨页指一个大于 4Ki 的内存页。在 x86_64 构架中，有两个常见的巨页大小: 2Mi 和 1Gi。在其它构架上的大小会有所不同。要使用巨页，必须写相应的代码以便应用程序了解它们。Transparent Huge Pages（THP）试图在应用程序不需要了解的情况下自动管理巨页，但这个技术有一定的限制。特别是，它的页大小会被限为 2Mi。当有较高的内存使用率时，THP 可能会导致节点性能下降，或出现大量内存碎片（因为 THP 的碎片处理）导致内存页被锁定。因此，有些应用程序可能更适用于（或推荐）使用预先分配的巨页，而不是 THP。

在 OpenShift Container Platform 中，pod 中的应用程序可以分配并消耗预先分配的巨页。

16.2. 应用程序如何使用巨页

节点必须预先分配巨页以便节点报告其巨页容量。一个节点只能预先分配一个固定大小的巨页。

巨页可以使用名为 **hugepages-<size>** 的容器一级的资源需求被消耗。其中 size 是特定节点上支持的整数值的最精简的二进制标记。例如：如果某个节点支持 2048KiB 页大小，它将会有一个可调度的资源 **hugepages-2Mi**。与 CPU 或者内存不同，巨页不支持过量分配。

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - securityContext:
      privileged: true
    image: rhel7:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /dev/hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 100Mi (1)
        memory: "1Gi"
        cpu: "1"
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

1 为巨页指定要分配的准确内存数量。不要将这个值指定为巨页内存大小乘以页的大小。例如，巨页的大小为 2MB，如果应用程序需要使用由巨页组成的 100MB 的内存，则需要分配 50 个巨页。

分配特定大小的巨页

有些平台支持多个巨页大小。要分配指定大小的巨页，在巨页引导命令参数前使用巨页大小选择参数**hugepagesz=<size>**。**<size>** 的值必须以字节为单位，并可以使用一个可选的后缀 [**kKmMgG**]。默认的巨页大小可使用 **default_hugepagesz=<size>** 引导参数定义。

巨页要求

- 巨页面请求必须等于限制。如果指定了限制，则它是默认的，但请求不是。
- 巨页在 pod 范围内被隔离。容器隔离功能计划在以后的版本中推出。
- 后端为巨页的 **EmptyDir** 卷不能消耗大于 pod 请求的巨页内存。
- 通过带有 **SHM_HUGETLB** 的 **shmget()** 来使用巨页的应用程序，需要运行一个匹配 ***proc/sys/vm/hugetlb_shm_group*** 的 supplemental 组。

16.3. 使用 DOWNWARD API 消耗巨页资源

您可以使用 Downward API 注入容器消耗的巨页资源的信息。

您可以将资源分配作为环境变量、卷插件或两者都注入。您在容器中开发和运行的应用可以通过读取指定卷中的环境变量或文件来确定可用的资源。

流程

1. 创建一个类似以下示例的 **hugepages-volume-pod.yaml** 文件：

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
  labels:
    app: hugepages-example
spec:
  containers:
  - securityContext:
      capabilities:
        add: [ "IPC_LOCK" ]
    image: rhel7:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /dev/hugepages
      name: hugepage
    - mountPath: /etc/podinfo
      name: podinfo
    resources:
      limits:
        hugepages-1Gi: 2Gi
```

```
        memory: "1Gi"
        cpu: "1"
      requests:
        hugepages-1Gi: 2Gi
    env:
    - name: REQUESTS_HUGEPAGES_1GI <.>
      valueFrom:
        resourceFieldRef:
          containerName: example
          resource: requests.hugepages-1Gi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
  - name: podinfo
    downwardAPI:
      items:
        - path: "hugepages_1G_request" <.>
          resourceFieldRef:
            containerName: example
            resource: requests.hugepages-1Gi
            divisor: 1Gi
```

<.> 指定从 **requests.hugepages-1Gi** 读取资源使用，并将值公开为 **REQUESTS_HUGEPAGES_1GI** 环境变量。< .> 指定从 **requests.hugepages-1Gi** 读取资源使用,并将值公开为文件 **/etc/podinfo/hugepages_1G_request**。

2. 从 **hugepages-volume-pod.yaml** 文件创建 pod：

```
$ oc create -f hugepages-volume-pod.yaml
```

验证

1. 检查 **REQUESTS_HUGEPAGES_1GI** 环境变量的值：

```
$ oc exec -it $(oc get pods -l app=hugepages-example -o
jsonpath='{.items[0].metadata.name}') \
    -- env | grep REQUESTS_HUGEPAGES_1GI
```

输出示例

```
REQUESTS_HUGEPAGES_1GI=2147483648
```

2. 检查 **/etc/podinfo/hugepages_1G_request** 文件的值：

```
$ oc exec -it $(oc get pods -l app=hugepages-example -o
jsonpath='{.items[0].metadata.name}') \
    -- cat /etc/podinfo/hugepages_1G_request
```

输出示例

```
2
```

其他资源

- 允许容器消耗 Downward API 对象

16.4. 配置巨页

节点必须预先分配在 OpenShift Container Platform 集群中使用的巨页。保留巨页的方法有两种： 在引导时和在运行时。在引导时进行保留会增加成功的可能性，因为内存还没有很大的碎片。Node Tuning Operator 目前支持在特定节点上分配巨页。

16.4.1. 在引导时

流程

要减少节点重启的情况，请按照以下步骤顺序进行操作：

1. 通过标签标记所有需要相同巨页设置的节点。

```
$ oc label node <node_using_hugepages> node-role.kubernetes.io/worker-hp=
```

2. 创建一个包含以下内容的文件，并把它命名为 **hugepages_tuning.yaml**：

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: hugepages (1)
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile: (2)
  - data: |
      [main]
      summary=Boot time configuration for hugepages
      include=openshift-node
      [bootloader]
      cmdline_openshift_node_hugepages=hugepagesz=2M hugepages=50 (3)
    name: openshift-node-hugepages

  recommend:
  - machineConfigLabels: (4)
      machineconfiguration.openshift.io/role: "worker-hp"
    priority: 30
    profile: openshift-node-hugepages
```

(1) 将 Tuned 资源的 **name** 设置为 **hugepages**。

(2) 将 **profile** 部分设置为分配巨页。

(3) 请注意，参数顺序是非常重要的，因为有些平台支持各种大小的巨页。

(4) 启用基于机器配置池的匹配。

3. 创建 Tuned **hugepages** 对象

```
$ oc create -f hugepages-tuned-boottime.yaml
```

4. 创建一个带有以下内容的文件，并把它命名为 **hugepages-mcp.yaml**：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  name: worker-hp
  labels:
    worker-hp: ""
spec:
  machineConfigSelector:
    matchExpressions:
      - {key: machineconfiguration.openshift.io/role, operator: In, values: [worker,worker-hp]}
  nodeSelector:
    matchLabels:
      node-role.kubernetes.io/worker-hp: ""
```

5. 创建机器配置池：

```
$ oc create -f hugepages-mcp.yaml
```

因为有足够的非碎片内存，**worker-hp** 机器配置池中的所有节点现在都应分配 50 个 2Mi 巨页。

```
$ oc get node <node_using_hugepages> -o jsonpath="{.status.allocatable.hugepages-2Mi}"
100Mi
```

警告

目前，这个功能只在 Red Hat Enterprise Linux CoreOS（RHCOS）8.x worker 节点上被支持。在 Red Hat Enterprise Linux (RHEL) 7.x worker 节点上，目前不支持 TuneD **[bootloader]** 插件。

16.5. 禁用透明巨页

Transparent Huge Pages (THP) 会试图自动执行创建、管理和使用巨页的大部分方面。由于 THP 自动管理巨页，因此并不始终对所有类型的工作负载进行最佳处理。THP 可能会导致性能下降，因为许多应用程序都自行处理巨页。因此，请考虑禁用 THP。以下步骤描述了如何使用 Node Tuning Operator (NTO)禁用 THP。

流程

1. 使用以下内容创建文件，并将其命名为 **thp-disable-tuned.yaml** ：

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: thp-workers-profile
  namespace: openshift-cluster-node-tuning-operator
spec:
```

```
    profile:
    - data: |
        [main]
        summary=Custom tuned profile for OpenShift to turn off THP on worker nodes
        include=openshift-node

        [vm]
        transparent_hugepages=never
      name: openshift-thp-never-worker

    recommend:
    - match:
      - label: node-role.kubernetes.io/worker
      priority: 25
      profile: openshift-thp-never-worker
```

2. 创建 Tuned 对象：

```
$ oc create -f thp-disable-tuned.yaml
```

3. 检查活跃配置集列表：

```
$ oc get profile -n openshift-cluster-node-tuning-operator
```

验证

- 登录到其中一个节点，并执行常规 THP 检查来验证节点是否成功应用了配置集：

```
$ cat /sys/kernel/mm/transparent_hugepage/enabled
```

输出示例

```
always madvise [never]
```

第 17 章 低延迟节点的 PERFORMANCE ADDON OPERATOR

17.1. 了解低延迟

在 Telco / 5G 领域，Edge 计算对于减少延迟和拥塞问题，以及提高应用程序性能方面扮演了关键角色。

简单地说，延迟决定了数据（packets）从发送方到接收方的速度，以及在接收方处理后返回到发送方的速度。显然，维护一个最低延迟速度的网络构架是满足 5G 的网络性能要求的关键。对于 4G 技术，它的平均延迟为 50ms，与之相比，5G 的目标是达到 1ms 或更少的延迟。这个对延迟的降低会将无线网络的吞吐量提高 10 倍。

很多在 Telco 空间部署的应用程序都需要低延迟，它们只能容忍零数据包丢失。针对零数据包丢失进行调节有助于缓解降低网络性能的固有问题。如需更多信息，请参阅 Red Hat OpenStack Platform(RHOSP) 中的 Zero Packet Los 调节。

Edge 计算也可用于降低延迟率。它位于云的边缘，并且更接近用户。这可大大减少用户和远程数据中心之间的距离，从而减少应用程序响应时间和性能延迟。

管理员必须能够集中管理多个 Edge 站点和本地服务，以便所有部署都可以以最低的管理成本运行。它们还需要一个简便的方法来部署和配置其集群的某些节点，以实现实时低延迟和高性能目的。低延迟节点对于如 Cloud-native Network Functions（CNF）和 Data Plane Development Kit（DPDK） 等应用程序非常有用。

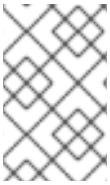
OpenShift Container Platform 目前提供在 OpenShift Container Platform 集群上调整软件的机制，以获取实时运行和低延迟时间（响应时间小于 20 微秒）。这包括调整内核和 OpenShift Container Platform 设置值、安装内核和重新配置机器。但是这个方法需要设置四个不同的 Operator，并执行很多配置，这些配置在手动完成时比较复杂，并容易出错。

OpenShift Container Platform 提供了一个 Performance Addon Operator 来实现自动性能优化，从而实现 OpenShift 应用程序的低延迟性能。集群管理员使用此性能配置集配置，这有助于以更可靠的方式进行更改。管理员可以指定是否要将内核更新至 kernel-rt，为集群和操作系统日常任务保留 CPU（包括 pod infra 容器），以及隔离 CPU，以便应用程序容器运行工作负载。

17.1.1. 关于低延迟和实时应用程序超线程

超线程是一个 Intel 处理器技术，它允许物理 CPU 处理器内核作为两个逻辑内核同时执行两个独立的线程。超线程可以为并行处理很有用的某些工作负载类型的系统吞吐量提供更好的系统吞吐量。默认的 OpenShift Container Platform 配置需要默认启用超线程。

对于电信领域的应用程序，设计您的应用程序架构非常重要，以尽量减小延迟。超线程会降低性能，并严重影响需要低延迟的计算负载的吞吐量。禁用超线程可确保性能的可预测性，并可减少这些工作负载的处理时间。



注意

超线程实现和配置会因运行 OpenShift Container Platform 的硬件而异。如需了解特定于该硬件的超线程实现的更多详情，请参考相关的主机硬件调节信息。禁用超线程可以增加集群的每个内核的成本。

其他资源

- 为集群配置超线程

17.2. 安装 PERFORMANCE ADDON OPERATOR

Performance Addon Operator 提供了在一组节点上启用高级节点性能调整的功能。作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台安装 Performance Addon Operator。

17.2.1. 使用 CLI 安装 Operator

作为集群管理员，您可以使用 CLI 安装 Operator。

先决条件

- 在裸机硬件上安装的集群。
- 安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 特权的用户身份登录。

流程

1. 通过完成以下操作，为 Performance Addon Operator 创建命名空间：
   a. 创建用于定义 **openshift-performance-addon-operator** 命名空间的以下 Namespace 自定义资源（CR），然后在 **pao-namespace.yaml** 文件中保存 YAML：

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-performance-addon-operator
  annotations:
    workload.openshift.io/allowed: management
```

   b. 运行以下命令创建命名空间：

```
$ oc create -f pao-namespace.yaml
```

2. 通过创建以下对象，在您上一步创建的命名空间中安装 Performance Addon Operator:
   a. 创建以下 **OperatorGroup** CR，并在 **pao-operatorgroup.yaml** 文件中保存 YAML:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: openshift-performance-addon-operator
  namespace: openshift-performance-addon-operator
```

   b. 运行以下命令来创建 **OperatorGroup** CR:

```
$ oc create -f pao-operatorgroup.yaml
```

   c. 运行以下命令获取下一步所需的 **channel** 值。

```
$ oc get packagemanifest performance-addon-operator -n openshift-marketplace -o
jsonpath='{.status.defaultChannel}'
```

   输出示例

```
4.8
```

d. 创建以下订阅 CR，并将 YAML 保存到 **pao-sub.yaml** 文件中：

订阅示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-performance-addon-operator-subscription
  namespace: openshift-performance-addon-operator
spec:
  channel: "<channel>" (1)
  name: performance-addon-operator
  source: redhat-operators (2)
  sourceNamespace: openshift-marketplace
```

(1) 使用在前一步中获得的值指定 **.status.defaultChannel** 参数。

(2) 您必须指定 **redhat-operators** 值。

e. 运行以下命令创建订阅对象：

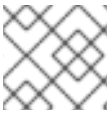
```
$ oc create -f pao-sub.yaml
```

f. 进入 **openshift-performance-addon-operator** 项目：

```
$ oc project openshift-performance-addon-operator
```

17.2.2. 使用 Web 控制台安装 Performance Addon Operator

作为集群管理员，您可以使用 web 控制台安装 Performance Addon Operator。

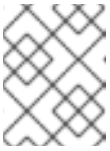


注意

如上一节所述，您必须创建 **Namespace** CR 和 **OperatorGroup** CR。

流程

1. 使用 OpenShift Container Platform Web 控制台安装 Performance Addon Operator:
   a. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
   b. 从可用的 Operator 列表中选择 **Performance Addon Operator**，然后点 **Install**。
   c. 在 **Install Operator** 页面中，选择 **All namespaces on the cluster**。然后点击 **Install**。
2. 可选：验证 performance-addon-operator 是否已成功安装：
   a. 切换到 **Operators → Installed Operators** 页面。
   b. 确保 **openshift-operators** 项目中列出的 **Performance Addon Operator** 的 **Status** 为 **Succeeded**。



注意

在安装过程中，Operator 可能会显示 **Failed** 状态。如果后续安装成功并显示 **Succeeded** 信息，您可以忽略 **Failed** 信息。

如果 Operator 没有被安装，您可以进一步排除故障：

- 进入 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。
- 进入 **Workloads → Pods** 页面，检查 **openshift-operators** 项目中 pod 的日志。

17.3. 升级 PERFORMANCE ADDON OPERATOR

您可以手动升级到 Performance Addon Operator 的下一个次版本，并使用 web 控制台监控更新的状态。

17.3.1. 关于升级 Performance Addon Operator

- 您可以使用 OpenShift Container Platform web 控制台更改 Operator 订阅的频道把 Performance Addon Operator 升级到下一个次要版本。
- 您可在安装 Performance Addon Operator 的过程中启用自动 z-stream 更新功能。
- 更新通过 Marketplace Operator 实现，该 Operator 会在 OpenShift Container Platform 安装过程中部署。Marketplace Operator 使外部 Operator 可供集群使用。
- 更新完成所需时间取决于您的网络连接情况。大部分自动更新可在十五分钟内完成。

17.3.1.1. Performance Addon Operator 升级对您的集群有什么影响

- 低延迟性能和巨页均不会受到影响。
- 更新 Operator 应该不会造成任何意外重启。

17.3.1.2. 将 Performance Addon Operator 升级到下一个次版本

您可以使用 OpenShift Container Platform Web 控制台修改 Operator 订阅的频道，将 Performance Addon Operator 手动升级到下一个次版本 。

先决条件

- 使用具有 cluster-admin 角色的用户访问集群。

流程

1. 访问 Web 控制台，进入 **Operators → Installed Operators**。
2. 点击**Performance Addon Operator** 打开 **Operator Details** 页面。
3. 单击 **Subscription** 选项卡，以打开 **Subscription 详情**页面。
4. 在 **Update 频道**窗格中，点击版本号右侧的铅笔图标打开 **Change Subscription 更新频道**窗口。
5. 选择下一个次要版本。例如，如果要升级到 Performance Addon Operator 4.8，请选择 **4.8**。

6. 点击 **Save**。

7. 通过导航到 **Operators → Installed Operators** 来检查升级的状态。您还可以通过运行以下 **oc** 命令来检查状态：

```
$ oc get csv -n openshift-performance-addon-operator
```

17.3.1.3. 当以前安装到特定命名空间时，升级 Performance Addon Operator

如果您之前已将 Performance Addon Operator 安装到集群中的特定命名空间中，如 **openshift-performance-addon-operator**，修改 **OperatorGroup** 对象以在升级前删除 **targetNamespaces** 条目。

先决条件

- 安装 OpenShift Container Platform CLI（oc）。
- 以具有 cluster-admin 权限的用户身份登录 OpenShift 集群。

流程

1. 运行以下命令，编辑 Performance Addon Operator **OperatorGroup** CR 并删除包含 **targetNamespaces** 条目的 **spec** 元素：

```
$ oc patch operatorgroup -n openshift-performance-addon-operator openshift-performance-addon-operator --type json -p '[{ "op": "remove", "path": "/spec" }]'
```

2. 等待 Operator Lifecycle Manager（OLM）处理更改。

3. 验证 OperatorGroup CR 更改是否已成功应用。检查 **OperatorGroup** CR **spec** 元素是否已删除：

```
$ oc describe -n openshift-performance-addon-operator og openshift-performance-addon-operator
```

4. 执行 Performance Addon Operator 升级。

17.3.2. 监控升级状态

监控 Performance Addon Operator 升级状态的最佳方法是查看 **ClusterServiceVersion**（CSV）**PHASE**。您还可以在 web 控制台中或通过运行 **oc get csv** 命令来监控 CSV 状况。

注意

PHASE 和状况值均是基于可用信息的近似值。

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 安装 OpenShift CLI（**oc**）。

流程

1. 运行以下命令：

```
$ oc get csv
```

2. 查看输出，检查 **PHASE** 字段。例如：

```
VERSION    REPLACES                                PHASE
4.8.0      performance-addon-operator.v4.8.0       Installing
4.7.0                                              Replacing
```

3. 再次运行 **get csv** 以验证输出：

```
# oc get csv
```

输出示例

```
NAME                         DISPLAY                VERSION   REPLACES
PHASE
performance-addon-operator.v4.8.0   Performance Addon Operator   4.8.0     performance-
addon-operator.v4.7.0   Succeeded
```

17.4. 置备实时和低延迟工作负载

很多行业和机构需要非常高的计算性能，并需要低且可预测的延迟，特别是银行和电信业。对于这些行业，它们有其特殊的要求，OpenShift Container Platform 提供了一个 Performance Addon Operator 来实现自动调整，以便为 OpenShift Container Platform 应用程序实现低延迟性能和响应时间。

集群管理员可以使用此性能配置集配置以更可靠的方式进行这些更改。管理员可以指定是否将内核更新至 kernel-rt（实时），为集群和操作系统日常任务保留 CPU（包括 pod infra 容器），以及隔离 CPU，以便应用程序容器运行工作负载。

警告

将执行探测与需要保证 CPU 的应用配合使用可能会导致延迟激增。建议使用其他探测（如正确配置的一组网络探测作为替代方案）。

17.4.1. 已知的实时限制

注意

RT 内核只在 worker 节点上被支持。

要充分利用实时模式，容器必须使用升级的权限运行。如需了解有关授予特权的信息，请参阅为容器设置能力。

OpenShift Container Platform 会限制允许的功能，因此您可能需要创建 **SecurityContext**。

注意

使用 Red Hat Enterprise Linux CoreOS（RHCOS）系统的裸机安装完全支持此步骤。

在确定正确的性能预期时，应该意识到实时内核并不是万能的。它的目的是提供一个持续的、低延迟的确定性机制，从而提供可预测的响应时间。在系统中，会存在与实时内核关联的额外内核开销。这是因为在单独调度的线程中处理硬件中断。某些增加的工作负载开销会导致整个吞吐量下降。实际的影响依赖于特定的负载，范围从 0% 到 30%。然而，这是获得确定性所需要付出的代价。

17.4.2. 使用实时功能置备 worker

1. 在集群上安装 Performance Addon Operator。
2. 可选：在 OpenShift Container Platform 集群中添加节点。请参阅设置 BIOS 参数。
3. 使用 **oc** 命令将标签 **worker-rt** 添加到需要实时功能的 worker 节点。
4. 为实时节点创建新机器配置池：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfigPool
metadata:
  name: worker-rt
  labels:
    machineconfiguration.openshift.io/role: worker-rt
spec:
  machineConfigSelector:
    matchExpressions:
      - {
          key: machineconfiguration.openshift.io/role,
          operator: In,
          values: [worker, worker-rt],
        }
  paused: false
  nodeSelector:
    matchLabels:
      node-role.kubernetes.io/worker-rt: ""
```

请注意，为具有标签 worker-rt 标签的节点组创建一个机器配置池 **worker-rt**。

5. 使用节点角色标签将节点添加到正确的机器配置池。

注意

您必须决定使用实时工作负载配置哪些节点。您可以配置集群中的所有节点，或配置节点的子集。期望所有节点都是专用机器配置池的一部分的 Performance Addon Operator。如果使用所有节点，您必须将 Performance Addon Operator 指向 worker 节点角色标签。如果使用子集，您必须将节点分组到新机器配置池中。

6. 使用正确的 housekeeping 内核和 **realTimeKernel: enabled: true** 创建 **PerformanceProfile**。
7. 您必须在 **PerformanceProfile** 中设置 **MachineConfigPoolSelector** ：

```
apiVersion: performance.openshift.io/v2
```

```
kind: PerformanceProfile
metadata:
 name: example-performanceprofile
spec:
...
  realTimeKernel:
    enabled: true
  nodeSelector:
    node-role.kubernetes.io/worker-rt: ""
  machineConfigPoolSelector:
    machineconfiguration.openshift.io/role: worker-rt
```

8. 验证匹配的机器配置池是否存在一个标签：

```
$ oc describe mcp/worker-rt
```

输出示例

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

9. OpenShift Container Platform 将开始配置节点，这可能涉及多次重启。等待节点处于稳定状态。这个过程所需要的时间取决于您所使用的具体硬件，预计每个节点需要 20 分钟。

10. 验证所有内容是否按预期工作。

17.4.3. 验证实时内核安装

使用这个命令确定安装了实时内核：

```
$ oc get node -o wide
```

记录带有角色 **worker-rt**，包括 **4.18.0-211.rt5.23.el8.x86_64** 字符串的 worker。

```
NAME                         STATUS   ROLES          AGE  VERSION                INTERNAL-IP
EXTERNAL-IP   OS-IMAGE                                KERNEL-VERSION
CONTAINER-RUNTIME
rt-worker-0.example.com              Ready  worker,worker-rt   5d17h   v1.22.1
128.66.135.107   <none>              Red Hat Enterprise Linux CoreOS 46.82.202008252340-0 (Ootpa)
4.18.0-211.rt5.23.el8.x86_64   cri-o://1.21.0-90.rhaos4.8.git4a0ac05.el8-rc.1
[...]
```

17.4.4. 创建一个实时工作负载

使用以下步骤准备一个使用实时功能的工作负载。

流程

1. 创建带有 **Guaranteed** 类 QoS 类的 pod。

2. 可选：禁用 DPDK 的 CPU 负载均衡。

3. 分配正确的节点选择器。

在编写应用程序时，请遵循应用程序调整和部署中的常规建议。

17.4.5. 创建带有 Guaranteed 类 QoS 类的 pod

在创建带有 **Guaranteed** 类的 QoS 类的 pod 时请注意以下几点：

- pod 中的每个容器都必须具有内存限制和内存请求，且它们必须相同。
- pod 中的每个容器都必须具有 CPU 限制和 CPU 请求，且它们必须相同。

以下示例显示了一个容器的 pod 的配置文件。容器设置了内存限制和内存请求，均为 200 MiB。容器具有 CPU 限制和 CPU 请求，均为 1 CPU。

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo
  namespace: qos-example
spec:
  containers:
  - name: qos-demo-ctr
    image: <image-pull-spec>
    resources:
      limits:
        memory: "200Mi"
        cpu: "1"
      requests:
        memory: "200Mi"
        cpu: "1"
```

1. 创建 pod：

```
$ oc  apply -f qos-pod.yaml --namespace=qos-example
```

2. 查看有关 pod 的详细信息：

```
$ oc get pod qos-demo --namespace=qos-example --output=yaml
```

输出示例

```
spec:
  containers:
    ...
status:
  qosClass: Guaranteed
```

注意

如果容器指定了自己的内存限值，但没有指定内存请求，OpenShift Container Platform 会自动分配与限制匹配的内存请求。同样，如果容器指定了自己的 CPU 限值，但没有指定 CPU 请求，OpenShift Container Platform 会自动分配与限制匹配的 CPU 请求。

17.4.6. 可选：禁用 DPDK 的 CPU 负载均衡

禁用或启用 CPU 负载均衡的功能在 CRI-O 级别实现。CRI-O 下的代码仅在满足以下要求时禁用或启用 CPU 负载均衡。

- pod 必须使用 **performance-<profile-name>** 运行时类。您可以通过查看性能配置集的状态来获得正确的名称，如下所示：

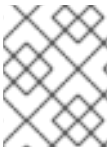
```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
...
status:
  ...
  runtimeClass: performance-manual
```

- pod 必须具有 **cpu-load-balancing.crio.io: true** 注解。

Performance Addon Operator 负责在相关节点下创建高性能运行时处理器配置片断，并在集群下创建高性能运行时类。它具有与默认运行时处理相同的内容，但它启用了 CPU 负载均衡配置功能。

要禁用 pod 的 CPU 负载均衡，**Pod** 规格必须包括以下字段：

```
apiVersion: v1
kind: Pod
metadata:
  ...
  annotations:
    ...
    cpu-load-balancing.crio.io: "disable"
    ...
  ...
spec:
  ...
  runtimeClassName: performance-<profile_name>
  ...
```



注意

仅在启用了 CPU 管理器静态策略，以及带有保证 QoS 使用整个 CPU 的 pod 时，禁用 CPU 负载均衡。否则，禁用 CPU 负载均衡会影响集群中其他容器的性能。

17.4.7. 分配适当的节点选择器

为节点分配 pod 的首选方法是使用与性能配置集相同的节点选择器，如下所示：

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  # ...
  nodeSelector:
    node-role.kubernetes.io/worker-rt: ""
```

如需更多信息，请参阅使用节点选择器将 pod 放置到特定的节点上。

17.4.8. 将工作负载调度到具有实时功能的 worker

使用与附加到机器配置的节点匹配的标签选择器，这些选择器是为低延迟配置的。如需更多信息，请参阅将 pod 分配给节点。

17.4.9. 管理设备中断处理保证 pod 隔离 CPU

Performance Addon Operator 可以通过将主机 CPU 划分为保留的 CPU 来管理主机 CPU，以进行集群和操作系统日常任务（包括 pod infra 容器），以及用于应用程序容器运行工作负载的隔离 CPU。这可让您将低延迟工作负载的 CPU 设置为隔离状态。

设备中断在所有隔离和保留 CPU 之间平衡负载，以避免出现 CPU 超载问题，但运行有保证 pod 的 CPU 除外。当为 pod 设置相关注解时，保证 pod CPU 无法处理设备中断。

在性能配置集中，**globallyDisableIrqLoadBalancing** 用于管理设备中断是否被处理。对于某些工作负载，保留 CPU 并不总是足以处理设备中断，因此不会在隔离的 CPU 上禁用设备中断。默认情况下，Performance Addon Operator 不会禁用隔离 CPU 上的设备中断。

要实现低延迟，有些（而非全部）pod 需要它们运行的 CPU 不处理设备中断。pod 注解 **irq-load-balancing.crio.io** 用于定义是否处理设备中断。配置后，CRI-O 仅在 pod 正在运行时禁用设备中断。

17.4.9.1. 禁用 CPU CFS 配额

要减少单独保证 pod 的 CPU 节流，创建一个带有注解 **cpu-quota.crio.io: "disable"** 的 pod 规格。此注释在 pod 运行时禁用 CPU 完全公平调度程序(CFS)配额。以下 pod 规格包含此注解：

```
apiVersion: performance.openshift.io/v2
kind: Pod
metadata:
  annotations:
      cpu-quota.crio.io: "disable"
spec:
    runtimeClassName: performance-<profile_name>
...
```

注意

仅在启用了 CPU 管理器静态策略，以及带有保证 QoS 使用整个 CPU 的 pod 时禁用 CPU CFS 配额。否则，禁用 CPU CFS 配额可能会影响集群中其他容器的性能。

17.4.9.2. 禁用 Performance Addon Operator 中的全局设备中断处理

要将 Performance Addon Operator 配置为禁用隔离 CPU 集的全局设备中断，将 performance 配置集中的 **globallyDisableIrqLoadBalancing** 字段设置为 **true**。在为 **true** 时，会忽略有冲突的 pod 注解。在为 **false** 时，IRQ 负载会在所有 CPU 之间平衡。

一个性能配置集片段演示了这个设置：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: manual
```

```
spec:
  globallyDisableIrqLoadBalancing: true
...
```

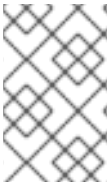
17.4.9.3. 禁用单个 pod 的中断处理

要禁用单个 pod 的中断处理，确保在性能配置集中将 **globallyDisableIrqLoadBalancing** 设置为 **false**。然后，在 pod 规格中，将 **irq-load-balancing.crio.io** pod 注解设置为 **disable**。以下 pod 规格包含此注解：

```
apiVersion: performance.openshift.io/v2
kind: Pod
metadata:
  annotations:
      irq-load-balancing.crio.io: "disable"
spec:
    runtimeClassName: performance-<profile_name>
...
```

17.4.10. 升级性能配置集以使用设备中断处理

当您将 Performance Addon Operator 性能配置集自定义资源定义（CRD）从 v1 或 v1alpha1 升级到 v2 时，现有配置集会将 **globallyDisableIrqLoadBalancing** 设置为 **true**。



注意

globallyDisableIrqLoadBalancing 切换用于 Isolated CPU 集是否禁用了 IRQ 负载均衡。当选项设置为 **true** 时，它会禁用 Isolated CPU 集的 IRQ 负载均衡。将选项设置为 **false** 允许在所有 CPU 之间平衡 IRQ。

17.4.10.1. 支持的 API 版本

Performance Addon Operator 在性能配置集 **apiVersion** 字段中支持 **v2**、**v1** 和 **v1alpha1**。v1 和 v1alpha1 API 相同。v2 API 包括一个可选的布尔值项 **globallyDisableIrqLoadBalancing**，默认值为 **false**。

17.4.10.1.1. 将 Performance Addon Operator API 从 v1alpha1 升级到 v1

当将 Performance Addon Operator API 版本从 v1alpha1 升级到 v1 时，v1alpha1 性能配置集会通过"None" Conversion 策略自行转换，并提供给带有 API 版本 v1 的 Performance Addon Operator。

17.4.10.1.2. 将 Performance Addon Operator API 从 v1alpha1 或 v1 升级到 v2

当从旧的 Performance Addon Operator API 版本升级时，现有的 v1 和 v1alpha1 性能配置集将使用转换 Webhook 转换，它将注入 **globallyDisableIrqLoadBalancing** 字段，值为 **true**。

17.4.11. 为 IRQ 动态负载平衡配置节点

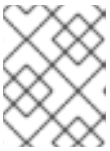
要将集群节点配置为处理 IRQ 动态负载平衡，请执行以下操作：

1. 以具有 cluster-admin 权限的用户身份登录 OpenShift Container Platform 集群。
2. 将性能配置集 **apiVersion** 设置为使用 **performance.openshift.io/v2**。

3. 删除 **globallyDisableIrqLoadBalancing** 字段，或把它设置为 **false**。

4. 设置适当的隔离 CPU 和保留的 CPU。以下片段演示了保留 2 个 CPU 的配置集。对于在 **isolated** CPU 集中运行的 pod，启用 IRQ 负载均衡：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: dynamic-irq-profile
spec:
  cpu:
    isolated: 2-5
    reserved: 0-1
...
```



注意

当您配置保留的和隔离的 CPU 时，pod 中的 infra 容器将使用保留的 CPU,应用程序容器则使用隔离的 CPU。

5. 创建使用独有 CPU 的 pod，并将 **irq-load-balancing.crio.io** 和 **cpu-quota.crio.io** 注解设置为 **disable**。例如：

```
apiVersion: v1
kind: Pod
metadata:
  name: dynamic-irq-pod
  annotations:
     irq-load-balancing.crio.io: "disable"
     cpu-quota.crio.io: "disable"
spec:
  containers:
  - name: dynamic-irq-pod
    image: "quay.io/openshift-kni/cnf-tests:4.8"
    command: ["sleep", "10h"]
    resources:
      requests:
        cpu: 2
        memory: "200M"
      limits:
        cpu: 2
        memory: "200M"
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
  runtimeClassName: performance-dynamic-irq-profile
...
```

6. 以 performance-<profile_name> 格式输入 pod 的 **runtimeClassName**，其中 <profile_name> 是来自 **PerformanceProfile** YAML 的 **name**，在本例中是 **performance-dynamic-irq-profile**。

7. 将节点选择器设置为以 cnf-worker 为目标。

8. 确保 pod 正确运行。状态应该为 **running**，并应正确设置了 cnf-worker 节点：

```
$ oc get pod -o wide
```

预期输出

```
NAME              READY   STATUS    RESTARTS   AGE     IP           NODE
NOMINATED NODE   READINESS GATES
dynamic-irq-pod   1/1     Running   0          5h33m   <ip-address>   <node-name>   <none>
<none>
```

9. 获取为 IRQ 动态负载均衡配置的 pod 运行 CPU：

```
$ oc exec -it dynamic-irq-pod -- /bin/bash -c "grep Cpus_allowed_list /proc/self/status | awk
'{print $2}'"
```

预期输出

```
Cpus_allowed_list:  2-3
```

10. 确保正确应用节点配置。SSH 到节点以验证配置。

```
$ oc debug node/<node-name>
```

预期输出

```
Starting pod/<node-name>-debug ...
To use host binaries, run `chroot /host`

Pod IP: <ip-address>
If you don't see a command prompt, try pressing enter.

sh-4.4#
```

11. 验证可以使用节点文件系统：

```
sh-4.4# chroot /host
```

预期输出

```
sh-4.4#
```

12. 确保默认系统 CPU 关联性掩码不包括 **dynamic-irq-pod** CPU，如 CPU 2 和 3。

```
$ cat /proc/irq/default_smp_affinity
```

输出示例

```
33
```

13. 确定系统 IRQ 没有配置为在 **dynamic-irq-pod** CPU 中运行：

```
find /proc/irq/ -name smp_affinity_list -exec sh -c 'i="$1"; mask=$(cat $i); file=$(echo $i); echo
$file: $mask' _ {} \;
```

输出示例

```
/proc/irq/0/smp_affinity_list: 0-5
/proc/irq/1/smp_affinity_list: 5
/proc/irq/2/smp_affinity_list: 0-5
/proc/irq/3/smp_affinity_list: 0-5
/proc/irq/4/smp_affinity_list: 0
/proc/irq/5/smp_affinity_list: 0-5
/proc/irq/6/smp_affinity_list: 0-5
/proc/irq/7/smp_affinity_list: 0-5
/proc/irq/8/smp_affinity_list: 4
/proc/irq/9/smp_affinity_list: 4
/proc/irq/10/smp_affinity_list: 0-5
/proc/irq/11/smp_affinity_list: 0
/proc/irq/12/smp_affinity_list: 1
/proc/irq/13/smp_affinity_list: 0-5
/proc/irq/14/smp_affinity_list: 1
/proc/irq/15/smp_affinity_list: 0
/proc/irq/24/smp_affinity_list: 1
/proc/irq/25/smp_affinity_list: 1
/proc/irq/26/smp_affinity_list: 1
/proc/irq/27/smp_affinity_list: 5
/proc/irq/28/smp_affinity_list: 1
/proc/irq/29/smp_affinity_list: 0
/proc/irq/30/smp_affinity_list: 0-5
```

一些 IRQ 控制器不支持 IRQ 重新平衡，并将始终将所有在线 CPU 公开为 IRQ 掩码。这些 IRQ 控制器在 CPU 0 上运行。如需了解主机配置的更多信息，请 SSH 到主机并运行以下命令，将 **<irq-num>** 替换为您要查询的 CPU 号码：

```
$ cat /proc/irq/<irq-num>/effective_affinity
```

17.4.12. 为集群配置超线程

要为 OpenShift Container Platform 集群配置超线程，请将性能配置集中的 CPU 线程设置为为保留或隔离的 CPU 池配置的相同内核。

注意

如果您配置了性能配置集，然后更改主机的超线程配置，请确保更新 **PerformanceProfile** YAML 中的 CPU **isolated** 和 **reserved**字段以匹配新配置。

警告

禁用之前启用的主机超线程配置可能会导致 **PerformanceProfile** YAML 中列出的 CPU 内核 ID 错误。此不正确的配置可能会导致节点不可用，因为无法找到列出的 CPU。

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 安装 OpenShift CLI（oc）。

流程

1. 确定在您要配置的主机的 CPU 上运行哪些线程。
   您可以通过登录到集群并运行以下命令来查看在主机 CPU 上运行哪些线程：

```
$ lscpu --all --extended
```

输出示例

```
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE MAXMHZ    MINMHZ
0   0    0      0    0:0:0:0       yes    4800.0000 400.0000
1   0    0      1    1:1:1:0       yes    4800.0000 400.0000
2   0    0      2    2:2:2:0       yes    4800.0000 400.0000
3   0    0      3    3:3:3:0       yes    4800.0000 400.0000
4   0    0      0    0:0:0:0       yes    4800.0000 400.0000
5   0    0      1    1:1:1:0       yes    4800.0000 400.0000
6   0    0      2    2:2:2:0       yes    4800.0000 400.0000
7   0    0      3    3:3:3:0       yes    4800.0000 400.0000
```

在这个示例中，在四个物理 CPU 内核中运行了八个逻辑 CPU 内核。CPU0 和 CPU4 在物理 Core0 中运行，CPU1 和 CPU5 在物理 Core 1 中运行，以此类推。

另外要查看为特定物理 CPU 内核设定的线程（以下示例中的**cpu0** ），打开命令提示符并运行以下命令：

```
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
```

输出示例

```
0-4
```

2. 在 **PerformanceProfile** YAML 中应用隔离和保留的 CPU。例如，您可以将逻辑内核 CPU0 和 CPU4 设置为 **isolated** ；将逻辑内核 CPU1 到 CPU3 以及 CPU5 到 CPU7 设置为 **reserved**。当您配置保留的和隔离的 CPU 时，pod 中的 infra 容器将使用保留的 CPU,应用程序容器则使用隔离的 CPU。

```
...
```

```
  cpu:
    isolated: 0,4
    reserved: 1-3,5-7
...
```

注意

保留和隔离的 CPU 池不得重叠，并且必须一起跨越 worker 节点中的所有可用内核。

重要

大多数 Intel 处理器上默认启用超线程。如果启用超线程，特定内核处理的所有线程都必须被隔离或者在同一个内核中处理。

17.4.12.1. 禁用低延迟应用程序超线程

在为低延迟进程配置集群时，请考虑是否要在部署集群前禁用超线程。要禁用超线程，请执行以下操作：

1. 创建一个适合您的硬件和拓扑的性能配置集。
2. 将 **nosmt** 设为附加内核参数。以下示例的性能配置集演示了此设置：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: example-performanceprofile
spec:
  additionalKernelArgs:
    - nmi_watchdog=0
    - audit=0
    - mce=off
    - processor.max_cstate=1
    - idle=poll
    - intel_idle.max_cstate=0
    - nosmt
  cpu:
    isolated: 2-3
    reserved: 0-1
  hugepages:
    defaultHugepagesSize: 1G
    pages:
      - count: 2
        node: 0
        size: 1G
  nodeSelector:
    node-role.kubernetes.io/performance: ''
  realTimeKernel:
    enabled: true
```

注意

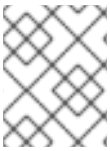
当您配置保留的和隔离的 CPU 时，pod 中的 infra 容器将使用保留的 CPU,应用程序容器则使用隔离的 CPU。

17.5. 使用性能配置集调整节点以实现低延迟

性能配置集可让您控制属于特定机器配置池的节点的延迟调整方面。指定设置后，**PerformanceProfile** 对象将编译为执行实际节点级别调整的多个对象：

- 操作节点的 **MachineConfig** 文件。
- 用于配置拓扑管理器、CPU Manager 和 OpenShift Container Platform 节点的 **KubeletConfig** 文件。
- 配置 Node Tuning Operator 的 Tuned 配置集。

您可以使用性能配置集指定是否将内核更新至 kernel-rt，分配大页面，以及划分 CPU 以执行内务处理或运行工作负载。



注意

您可以手动创建 **PerformanceProfile** 对象，或使用 Performance Profile Creator (PPC) 生成性能配置集。有关 PPC 的更多信息，请参见以下的其他资源。

性能配置集示例

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
 name: performance
spec:
 cpu:
  isolated: "5-15" 1
  reserved: "0-4" 2
 hugepages:
  defaultHugepagesSize: "1G"
  pages:
  - size: "1G"
    count: 16
    node: 0
 realTimeKernel:
  enabled: true 3
 numa: 4
  topologyPolicy: "best-effort"
 nodeSelector:
  node-role.kubernetes.io/worker-cnf: "" 5
```

1. 使用此字段隔离要用于工作负载的应用容器的特定 CPU。
2. 使用此字段保留要用于 infra 容器进行内务的特定 CPU。
3. 使用此字段在节点上安装实时内核。有效值为 **true** 或者 **false**。设置 **true** 值将安装实时内核。
4. 使用此字段配置拓扑管理器策略。有效值为 **none**（默认）、**best-effort**、**restricted** 和 **single-numa-node**。如需更多信息，请参阅拓扑管理器策略。
5. 使用此字段指定节点选择器，将性能配置集应用到特定的节点。

其他资源

- 有关使用 Performance Profile Creator (PPC) 生成性能配置集的详情，请参考创建性能配置集。

17.5.1. 配置巨页

节点必须预先分配在 OpenShift Container Platform 集群中使用的巨页。使用 Performance Addon Operator 在指定节点上分配巨页。

OpenShift Container Platform 提供了创建和分配巨页的方法。Performance Addon Operator 提供了一种更易于使用性能配置集的方法。

例如，在性能配置集的 **hugepages pages** 部分，您可以指定多个块的 **size**、**count** 以及可选的 **node**:

```
hugepages:
  defaultHugepagesSize: "1G"
  pages:
  - size:  "1G"
    count:  4
    node:  0 (1)
```

1 **node** 是分配巨页的 NUMA 节点。如果省略了 **node**，该页面将平均分布在所有 NUMA 节点中。

注意

等待显示更新已完成的相关机器配置池状态。

这些是分配巨页的唯一配置步骤。

验证

- 要验证配置，请查看节点上的 **/proc/meminfo** 文件：

```
$ oc debug node/ip-10-0-141-105.ec2.internal
```

```
# grep -i huge /proc/meminfo
```

输出示例

```
AnonHugePages:    ###### ##
ShmemHugePages:        0 kB
HugePages_Total:       2
HugePages_Free:        2
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       #### ##
Hugetlb:            #### ##
```

- 使用 **oc describe** 报告新大小：

```
$ oc describe node worker-0.ocp4poc.example.com | grep -i huge
```

输出示例

```
                        hugepages-1g=true
hugepages-###:  ###
hugepages-###:  ###
```

17.5.2. 分配多个巨页大小

您可以在同一容器下请求具有不同大小的巨页。这样，您可以定义由具有不同巨页大小的容器组成的更复杂的 pod。

例如，您可以把大小定义为 **1G** 和 **2M**，Performance Addon Operator 会在节点上配置这两个大小，如下所示：

```
spec:
  hugepages:
    defaultHugepagesSize: 1G
    pages:
    - count: 1024
      node: 0
      size: 2M
    - count: 4
      node: 1
      size: 1G
```

17.5.3. 为 infra 和应用程序容器限制 CPU

通用内务处理和工作负载任务使用 CPU 的方式可能会影响对延迟敏感的进程。默认情况下，容器运行时使用所有在线 CPU 一起运行所有容器，这可能导致上下文切换和延迟激增。对 CPU 进行分区可防止无状态进程通过相互分离来干扰对延迟敏感的进程。下表描述了在使用 Performance Add-On Operator 调整节点后在 CPU 上运行的进程：

表 17.1. 进程的 CPU 分配

进程类型	详情
Burstable 和 **BestEffort** pod	在除了运行低延迟工作负载外的任意 CPU 上运行
基础架构 pod	在除了运行低延迟工作负载外的任意 CPU 上运行
中断	重定向到保留的 CPU（OpenShift Container Platform 4.7 及更新的版本中的可选）
内核进程	固定保留的 CPU
对延迟敏感的工作负载 pod	固定到隔离池中的特定专用 CPU
OS 进程/systemd 服务	固定保留的 CPU

在一个节点上的对于所有 QoS 进程类型（ **Burstable**、**BestEffort** 或 **Guaranteed** ）的 pod 的可分配容量等于隔离池的容量。保留池的容量已从节点的总内核容量中删除，供集群和操作系统日常任务使用。

示例 1

节点具有 100 个内核的容量。通过使用性能配置集，集群管理员将 50 个内核分配给隔离池，将 50 个内核分配给保留池。集群管理员为 QoS 为 **BestEffort** 或 **Burstable** 的 pod 分配 25 个内核，为 **Guaranteed** 的 pod 分配 25 个内核。这与隔离池的容量匹配。

示例 2

节点具有 100 个内核的容量。通过使用性能配置集，集群管理员将 50 个内核分配给隔离池，将 50 个内核分配给保留池。集群管理员为 QoS 为 **BestEffort** 或 **Burstable** 的 pod 分配一个内核，为 **Guaranteed** 的 pod 分配 50 个内核。这超过了隔离池容量一个内核。Pod 调度因为 CPU 容量不足而失败。

使用的确切分区模式取决于许多因素，如硬件、工作负载特性和预期的系统负载。以下是一些用例示例：

- 如果对延迟敏感的工作负载使用特定的硬件，如网络接口控制器(NIC)，请确保隔离池中的 CPU 尽可能地与这个硬件接近。至少，您应该将工作负载放在同一个非统一内存访问 (NUMA) 节点中。
- 保留的池用于处理所有中断。根据系统网络，分配一个足够大小的保留池来处理所有传入的数据包中断。在 4.8 及更高版本中，工作负载可以选择性地被标记为敏感版本。

在决定哪些特定 CPU 用于保留和隔离分区时，需要详细分析和测量。设备和内存的 NUMA 紧密度等因素扮演了角色。选择也取决于工作负载架构和具体的用例。

重要

保留和隔离的 CPU 池不得重叠，并且必须一起跨越 worker 节点中的所有可用内核。

为确保内务处理任务和工作负载不会相互干扰，请在性能配置集的 **spec** 部分指定两组 CPU。

- **isolated** - 指定应用程序容器工作负载的 CPU。这些 CPU 的延迟最低。这个组中的进程没有中断，例如，可以达到更高的 DPDK 零数据包丢失带宽。
- **reserved** - 为集群和操作系统日常任务指定 CPU。**reserved** 组中的线程经常会比较繁忙。不要在 **reserved** 组中运行对延迟敏感的应用程序。对延迟敏感的应用程序在 **isolated** 组中运行。

流程

1. 创建适合环境硬件和拓扑的性能配置集。
2. 使用您想要为 infra 和应用程序容器保留和隔离的 CPU 添加 **reserved** 和 **isolated** 参数：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: infra-cpus
spec:
  cpu:
    reserved: "0-4,9" (1)
    isolated: "5-8" (2)
  nodeSelector: (3)
    node-role.kubernetes.io/worker: ""
```

(1) 指定 infra 容器用于执行集群和操作系统日常任务的 CPU。

(2) 指定应用程序容器运行工作负载的 CPU。

3 可选：指定一个节点选择器，以将性能配置集应用到特定的节点。

其他资源

- 管理设备中断处理保证 pod 隔离 CPU
- 创建一个带有 Guaranteed 类 QoS 类的 pod

17.6. 使用 PERFORMANCE ADDON OPERATOR 减少 NIC 队列

Performance Addon Operator 允许您通过配置性能配置集来调整每个网络设备的网络接口控制器 (NIC) 队列计数。设备网络队列允许在不同物理队列之间分布数据包，每个队列获得用于数据包处理的独立线程。

在实时或低延迟系统中，固定到隔离 CPU 的所有不必要的中断请求行（IRQ）必须移到保留或日常 CPU。

在部署需要系统、OpenShift Container Platform 网络或使用 Data Plane Development Kit（DPDK）工作负载的混合部署中，需要多个队列才能实现良好的吞吐量，并且 NIC 队列的数量应该会调整或保持不变。例如，为了实现低延迟，应该将基于 DPDK 的工作负载的 NIC 队列数量减少到仅保留或日常 CPU 的数量。

默认情况下，每个 CPU 会创建太多的队列，在为低延迟调优时，这些队列不适用于托管 CPU 的中断表中。减少队列数量可以进行正确的调优。数量较小的队列意味着在 IRQ 表中包括较少数量的中断。

17.6.1. 使用性能配置集调整 NIC 队列

通过性能配置集，您可以调整每个网络设备的队列计数。

支持的网络设备：

- 非虚拟网络设备
- 支持多个队列的网络设备（通道）

不支持的网络设备：

- 纯软件网络接口
- 块设备
- Intel DPDK 虚拟功能

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 安装 OpenShift CLI（**oc**）。

流程

1. 以具有 **cluster-admin** 权限的用户身份登录运行 Performance Addon Operator 的 OpenShift Container Platform 集群。

2. 创建并应用适合您的硬件和拓扑的性能配置集。有关创建配置集的指南，请参阅"创建性能配置集"部分。

3. 编辑这个创建的性能配置集：

```
$ oc edit -f <your_profile_name>.yaml
```

4. 使用 **net** 对象填充 **spec** 字段。对象列表可以包含两个字段：

   - **userLevelNetworking** 是一个必需字段，指定为布尔值标记。如果 **userLevelNetworking** 为 **true**，则队列数将设置为所有支持设备的保留 CPU 计数。默认值为 **false**。
   - **devices** 是一个可选字段，指定队列设置为保留 CPU 数的设备列表。如果设备列表为空，则配置适用于所有网络设备。配置如下：
     - **interfaceName**：此字段指定接口名称，并支持 shell 样式的通配符，可以是正数或负数。
       - 通配符语法示例如下： **<string> .***
       - 负规则的前缀为感叹号。要将网络队列更改应用到排除列表以外的所有设备，请使用 **!<device>**。例如 **!eno1**。
     - **vendorID**：网络设备供应商 ID，以带有 **0x** 前缀的 16 位十六进制数字代表。
     - **deviceID**：网络设备 ID（model），以带有 **0x** 前缀的 16 位十六进制数字代表。

注意

当指定 **deviceID** 时，还必须定义 **vendorID**。与设备条目 **interfaceName**、**vendorID** 或 **vendorID** 加 **deviceID** 中指定的所有设备标识符相匹配的设备会被视为一个网络设备。然后，此网络设备的 net 队列数设置为保留的 CPU 计数。

当指定了两个或多个设备时，网络队列数将设置为与其中一个设备匹配的任何网络设备。

5. 使用此示例性能配置集将所有设备的队列数设置为保留的 CPU 计数：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: manual
spec:
  cpu:
    isolated: 3-51,54-103
    reserved: 0-2,52-54
  net:
    userLevelNetworking: true
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

6. 使用这个示例性能配置集，将所有与任何定义的设备标识符匹配的保留 CPU 数设置为保留的 CPU 计数：

```
apiVersion: performance.openshift.io/v2
```

```
kind: PerformanceProfile
metadata:
  name: manual
spec:
  cpu:
    isolated: 3-51,54-103
    reserved: 0-2,52-54
  net:
    userLevelNetworking: true
    devices:
    - interfaceName: “eth0”
    - interfaceName: “eth1”
    - vendorID: “0x1af4”
    - deviceID: “0x1000”
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

7. 使用这个示例性能配置集，将所有以接口名称 **eth** 开头的设备的队列数设置为保留的 CPU 计数：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: manual
spec:
  cpu:
    isolated: 3-51,54-103
    reserved: 0-2,52-54
  net:
    userLevelNetworking: true
    devices:
    - interfaceName: “eth*”
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

8. 使用这个示例性能配置集。将所有设备的队列数设置为保留的 CPU 计数，该接口具有 **eno1** 以外的任何接口：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: manual
spec:
  cpu:
    isolated: 3-51,54-103
    reserved: 0-2,52-54
  net:
    userLevelNetworking: true
    devices:
    - interfaceName: “!eno1”
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

9. 使用这个示例性能配置集，将所有具有接口名称 **eth0**，**vendorID** 为 **0x1af4**、**deviceID** 为 **0x1000** 的设备的队列数设置为保留 CPU 数：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: manual
spec:
  cpu:
    isolated: 3-51,54-103
    reserved: 0-2,52-54
  net:
    userLevelNetworking: true
    devices:
    - interfaceName: “eth0”
    - vendorID: “0x1af4”
    - deviceID: “0x1000”
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

10. 应用更新的性能配置集：

```
$ oc apply -f <your_profile_name>.yaml
```

其他资源

- 创建性能配置集.

17.6.2. 验证队列状态

在这一部分中，一些示例演示了不同的性能配置集以及如何验证是否应用了更改。

示例 1

在本例中，网络队列数为*所有*支持的设备设置为保留 CPU 数（2）。

性能配置集中的相关部分是：

```
apiVersion: performance.openshift.io/v2
metadata:
  name: performance
spec:
  kind: PerformanceProfile
  spec:
    cpu:
      reserved: 0-1  #total = 2
      isolated: 2-8
    net:
      userLevelNetworking: true
# ...
```

- 使用以下命令显示与设备关联的队列状态：

注意

在应用了性能配置集的节点中运行这个命令。

```
$ ethtool -l <device>
```

- 在应用配置集前验证队列状态：

```
$ ethtool -l ens4
```

输出示例

```
Channel parameters for ens4:
Pre-set maximums:
RX:         0
TX:         0
Other:      0
Combined:   4
Current hardware settings:
RX:         0
TX:         0
Other:      0
Combined:   4
```

- 应用配置集后验证队列状态：

```
$ ethtool -l ens4
```

输出示例

```
Channel parameters for ens4:
Pre-set maximums:
RX:         0
TX:         0
Other:      0
Combined:   4
Current hardware settings:
RX:         0
TX:         0
Other:      0
Combined:   2 (1)
```

(1) 该组合通道显示为*所有*支持的设备保留 CPU 的总数为 2。这与性能配置集中配置的内容匹配。

示例 2

在本例中，针对具有特定 **vendorID** 的*所有*受支持的网络设备，网络队列数设置为保留 CPU 数（2）。

性能配置集中的相关部分是：

```
apiVersion: performance.openshift.io/v2
metadata:
  name: performance
spec:
  kind: PerformanceProfile
  spec:
    cpu:
```

```
    reserved: 0-1  #total = 2
    isolated: 2-8
  net:
    userLevelNetworking: true
    devices:
    - vendorID = 0x1af4
# ...
```

- 使用以下命令显示与设备关联的队列状态：

注意

在应用了性能配置集的节点中运行这个命令。

```
$ ethtool -l <device>
```

- 应用配置集后验证队列状态：

```
$ ethtool -l ens4
```

输出示例

```
Channel parameters for ens4:
Pre-set maximums:
RX:         0
TX:         0
Other:      0
Combined:   4
Current hardware settings:
RX:         0
TX:         0
Other:      0
Combined:   2 (1)
```

(1) 带有 **vendorID=0x1af4** 的所有支持设备的预留 CPU 总数为 2。例如，如果存在另一个网络设备 **ens2**，其 **vendorID=0x1af4** 也具有总计的网络队列为 2。这与性能配置集中配置的内容匹配。

示例 3

在本例中，针对与任何定义的设备标识符匹配的*所有*受支持网络设备，网络队列数设置为保留 CPU 数（2）。

命令 **udevadm info** 提供了有关设备的详细报告。在这个示例中，设备是：

```
# udevadm info -p /sys/class/net/ens4
...
E: ID_MODEL_ID=0x1000
E: ID_VENDOR_ID=0x1af4
E: INTERFACE=ens4
...
```

```
# udevadm info -p /sys/class/net/eth0
```

```
...
E: ID_MODEL_ID=0x1002
E: ID_VENDOR_ID=0x1001
E: INTERFACE=eth0
...
```

- 对于 **interfaceName** 等于 **eth0** 的设备，以及具有 **vendorID=0x1af4** 的设备，并使用以下性能配置集，将网络队列设置为 2：

```
apiVersion: performance.openshift.io/v2
metadata:
  name: performance
spec:
  kind: PerformanceProfile
    spec:
      cpu:
        reserved: 0-1  #total = 2
        isolated: 2-8
      net:
        userLevelNetworking: true
        devices:
        - interfaceName = eth0
        - vendorID = 0x1af4
...
```

- 应用配置集后验证队列状态：

```
$ ethtool -l ens4
```

输出示例

```
Channel parameters for ens4:
Pre-set maximums:
RX:         0
TX:         0
Other:      0
Combined:   4
Current hardware settings:
RX:         0
TX:         0
Other:      0
Combined:   2 (1)
```

(1) 带有 **vendorID=0x1af4** 的所有支持设备的预留 CPU 总数设置为 2。例如，如果存在另一个带有 **vendorID=0x1af4** 的网络设备 **ens2**，则其总子网队列也将设置为 2。类似地，**interfaceName** 等于 **eth0** 的设备会将总网络队列设置为 2。

17.6.3. 与调整 NIC 队列关联的日志记录

详细说明所分配设备的日志消息记录在相应的 Tuned 守护进程日志中。以下信息可能会记录到 **/var/log/tuned/tuned.log** 文件中：

- 记录了一个 **INFO** 信息，详细描述了成功分配的设备：

```
INFO tuned.plugins.base: instance net_test (net): assigning devices ens1, ens2, ens3
```

- 如果无法分配任何设备，则会记录 **WARNING** 信息：

```
WARNING  tuned.plugins.base: instance net_test: no matching devices available
```

17.7. 调试低延迟 CNF 调整状态

PerformanceProfile 自定义资源（CR）包含报告调整状态和调试延迟降级问题的状态字段。这些字段报告描述 Operator 协调功能状态的条件。

当附加到性能配置集的机器配置池处于降级状态时会出现一个典型的问题，从而导致 **PerformanceProfile** 状态降级。在这种情况下，机器配置池会给出一个失败信息。

Performance Addon Operator 包含 **performanceProfile.spec.status.Conditions status** 字段：

```
Status:
  Conditions:
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                True
    Type:                  Available
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                True
    Type:                  Upgradeable
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                False
    Type:                  Progressing
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                False
    Type:                  Degraded
```

Status 字段包含指定 **Type** 值来指示性能配置集状态的 **Conditions**：

Available

所有机器配置和 Tuned 配置集都已被成功创建，且集群组件可用于处理它们（NTO、MCO、Kubelet）。

Upgradeable

代表 Operator 维护的资源是否处于可安全升级的状态。

Progressing

表示已从性能配置集启动部署过程。

Degraded

如果出现以下情况代表错误：

- 验证性能配置集失败。
- 创建所有相关组件未能成功完成。

每个类型都包括以下字段：

状态

特定类型的状态（**true** 或 **false**）。

Timestamp

事务的时间戳。

Reason string

机器可读的原因。

Message string

描述状态和错误详情的人类可读的原因信息（如果存在）。

17.7.1. 机器配置池

性能配置集及其创建的产品会根据关联的机器配置池（MCP）应用到节点。MCP 包含有关应用由性能附加组件创建的机器配置的有价值的信息，它包括了内核 arg、Kube 配置、巨页分配和 rt-kernel 部署。性能附加控制器监控 MCP 中的更改，并相应地更新性能配置集状态。

MCP 返回到性能配置集状态的唯一条件是 MCP 处于 **Degraded** 状态，这会导致 **performaceProfile.status.condition.Degraded = true**。

示例

以下示例是创建关联机器配置池（**worker-cnf**）的性能配置集：

1. 关联的机器配置池处于降级状态：

```
# oc get mcp
```

输出示例

```
NAME        CONFIG                                    UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master      rendered-master-2ee57a93fa6c9181b546ca46e1571d2d      True      False
False      3              3              3              0              2d21h
worker      rendered-worker-d6b2bdc07d9f5a59a6b68950acf25e5f      True      False
False      2              2              2              0              2d21h
worker-cnf   rendered-worker-cnf-6c838641b8a08fff08dbd8b02fb63f7c   False     True
True       2              1              1              1              2d20h
```

2. MCP 的 **describe** 部分包括了原因：

```
# oc describe mcp worker-cnf
```

输出示例

```
  Message:               Node node-worker-cnf is reporting: "prepping update:
  machineconfig.machineconfiguration.openshift.io \"rendered-worker-cnf-
40b9996919c08e335f3ff230ce1d170\" not
  found"
    Reason:              1 nodes are reporting degraded status on sync
```

3. 降级状态也应该出现在标记为 **degraded = true** 的性能配置集的 **status** 字段中：

```
# oc describe performanceprofiles performance
```

输出示例

```
Message: Machine config pool worker-cnf Degraded Reason: 1 nodes are reporting
degraded status on sync.
Machine config pool worker-cnf Degraded Message: Node yquinn-q8s5v-w-b-
z5lqn.c.openshift-gce-devel.internal is
reporting: "prepping update: machineconfig.machineconfiguration.openshift.io
\"rendered-worker-cnf-40b9996919c08e335f3ff230ce1d170\" not found".    Reason:
MCPDegraded
  Status:  True
  Type:    Degraded
```

17.8. 为红帽支持收集调试数据延迟

在提交问题单时同时提供您的集群信息，可以帮助红帽支持为您进行排除故障。

您可使用 **must-gather** 工具来收集有关 OpenShift Container Platform 集群的诊断信息，包括节点调整、NUMA 拓扑和其他调试延迟设置问题所需的信息。

为了获得快速支持，请提供 OpenShift Container Platform 和低延迟调整的诊断信息。

17.8.1. 关于 must-gather 工具

oc adm must-gather CLI 命令可收集最有助于解决问题的集群信息，如：

- 资源定义
- 审计日志
- 服务日志

您在运行该命令时，可通过包含 **--image** 参数来指定一个或多个镜像。指定镜像后，该工具便会收集有关相应功能或产品的信息。在运行 **oc adm must-gather** 时，集群上会创建一个新 pod。在该 pod 上收集数据，并保存至以 **must-gather.local** 开头的一个新目录中。该目录在当前工作目录中创建。

17.8.2. 关于收集低延迟数据

使用 **oc adm must-gather** CLI 命令来收集有关集群的信息，包括与低延迟性能优化相关的功能和对象，包括：

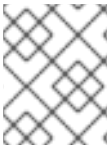
- Performance Addon Operator 命名空间和子对象。
- **MachineConfigPool** 和关联的 **MachineConfig** 对象。
- Node Tuning Operator 和关联的 Tuned 对象。
- Linux 内核命令行选项。
- CPU 和 NUMA 拓扑
- 基本 PCI 设备信息和 NUMA 本地性。

要使用 **must-gather** 来收集 Performance Addon Operator 调试信息，您必须指定 Performance Addon Operator **must-gather** 镜像：

```
--image=registry.redhat.io/openshift4/performance-addon-operator-must-gather-rhel8:v4.8.
```

17.8.3. 收集有关特定功能的数据

您可通过将 **oc adm must-gather** CLI 命令与 **--image** 或 **--image-stream** 参数结合使用来收集有关特定功能的调试信息。**must-gather** 工具支持多个镜像，这样您便可通过运行单个命令收集多个功能的数据。



注意

要收集除特定功能数据外的默认 **must-gather** 数据，请添加 **--image-stream=openshift/must-gather** 参数。

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 安装了 OpenShift Container Platform CLI（oc）。

流程

1. 进入存储 **must-gather** 数据的目录。
2. 使用一个或多个 **--image** 或 **--image-stream** 参数运行 **oc adm must-gather** 命令。例如，使用以下命令可收集默认集群数据和 Performance Addon Operator 特定信息：

```
$ oc adm must-gather \
 --image-stream=openshift/must-gather \ 1

 --image=registry.redhat.io/openshift4/performance-addon-operator-must-gather-rhel8:v4.8
 2
```

1. 默认 OpenShift Container Platform **must-gather** 镜像。
2. 低延迟调整诊断的 **must-gather** 镜像。

3. 从工作目录中创建的 **must-gather** 目录创建一个压缩文件。例如，在使用 Linux 操作系统的计算机上运行以下命令：

```
$ tar cvaf must-gather.tar.gz must-gather.local.5421342344627712289/ 1
```

1. 将 **must-gather-local.5421342344627712289/** 替换为实际目录名称。

4. 在红帽客户门户中为您的问题单附上压缩文件。

其他资源

- 有关 MachineConfig 和 KubeletConfig 的更多信息，请参阅管理节点。
- 如需有关 Node Tuning Operator 的更多信息，请参阅使用 Node Tuning Operator。

- 有关 PerformanceProfile 的更多信息，请参阅配置巨页。
- 如需有关容器消耗巨页的更多信息，请参阅应用程序如何使用巨页。

第 18 章 为平台验证执行延迟测试

您可以使用 Cloud-native Network Function (CNF) 测试镜像在启用了 CNF 的 OpenShift Container Platform 集群上运行延迟测试，其中安装了运行 CNF 工作负载所需的所有组件。运行延迟测试以验证工作负载的节点调整。

cnf-tests 容器镜像位于 **registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8** 中。

重要

cnf-tests 镜像还包含了红帽不支持的多个测试。红帽只支持延迟测试。

18.1. 运行延迟测试的先决条件

运行延迟测试前，集群必须满足以下要求：

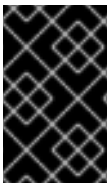
1. 已使用 Performance Addon Operator 配置一个性能配置集。
2. 已在集群中应用了所有所需的 CNF 配置。
3. 已在集群中应用了已存在的 **MachineConfigPool** CR。默认 worker 池为 **worker-cnf**。

其他资源

- 有关创建集群性能配置集的更多信息，请参阅置备实时和低延迟工作负载。

18.2. 关于延迟测试的发现模式

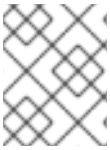
使用发现模式在不更改其配置的情况下验证集群的功能。在测试时使用现有环境配置。测试可以找到所需的配置项目，并使用这些项目来执行测试。如果没有找到运行特定测试所需的资源，则会跳过测试，为用户提供正确的信息。测试完成后，不会清理预配置的配置项目，测试环境可立即用于另一个测试运行。



重要

在运行延迟测试时，始终使用 **-e DISCOVERY_MODE=true** 和 **-ginkgo.focus** 设置为适当的延迟测试。如果您没有以发现模式运行延迟测试，则测试运行修改现有的实时集群性能配置集配置。

限制测试过程中使用的节点
通过指定 **NODES_SELECTOR** 环境变量来限制执行测试的节点，例如 **-e NODES_SELECTOR=node-role.kubernetes.io/worker-cnf**。测试创建的任何资源都仅限于具有匹配标签的节点。



注意

如果要覆盖默认 worker 池，请将 **-e ROLE_WORKER_CNF=<custom_worker_pool>** 变量传递给指定适当标签的命令。

18.3. 测量延迟

cnf-tests 镜像使用 **oslat** 工具来测量系统的延迟。

oslat

行为与 CPU 密集型 DPDK 应用程序类似，并测量模拟 CPU 繁重数据处理的忙碌循环和中断。

测试引入了以下环境变量：

表 18.1. 延迟测试环境变量

环境变量	描述
LATENCY_TEST_DELAY	指定测试开始运行的时间（以秒为单位）。您可以使用变量来允许 CPU 管理器协调循环来更新默认的 CPU 池。默认值为 0。
LATENCY_TEST_CPUS	指定运行延迟测试的 pod 使用的 CPU 数量。如果没有设置变量，则默认配置包含所有隔离的 CPU。
LATENCY_TEST_RUNTIME	指定延迟测试必须运行的时间（以秒为单位）。默认值为 300 秒。
OSLAT_MAXIMUM_LATENCY	指定 **oslat** 测试结果的最大可接受延迟（微秒）。如果没有为 **OSLAT_MAXIMUM_LATENCY** 设置值，该工具会跳过预期和实际最大延迟的比较。
LATENCY_TEST_RUN	指明测试是否应该运行的布尔值参数。**LATENCY_TEST_RUN** 默认设置为 **false**。要运行延迟测试，请将此值设置为 **true**。

18.4. 运行延迟测试

运行集群延迟测试，以验证 Cloud-native Network Function (CNF) 工作负载的节点调整。

重要

始终使用 **DISCOVERY_MODE=true** 设置运行延迟测试。如果没有，测试套件将对正在运行的集群配置进行更改。

注意

当以非 root 用户或非特权用户执行 **podman** 命令时，挂载路径可能会失败，错误为 **permission denied**。要使 **podman** 命令正常工作，请将 **:Z** 附加到卷创建中，例如 **-v $(pwd)/:/kubeconfig:Z**。这允许 **podman** 进行正确的 SELinux 重新标记。

流程

1. 在包含 **kubeconfig** 文件的目录中打开 shell 提示符。
   您可以在当前目录中为测试镜像提供 **kubeconfig** 文件，及其相关的 **$KUBECONFIG** 环境变量（通过卷挂载）。这允许运行的容器使用容器内的 **kubeconfig** 文件。

2. 输入以下命令运行延迟测试：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e LATENCY_TEST_RUN=true -e DISCOVERY_MODE=true
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
/usr/bin/test-run.sh -ginkgo.focus="\[performance\]\ Latency\ Test"
```

3. 可选：附加 **-ginkgo.dryRun** 以空运行模式运行延迟测试。这对于检查测试运行的内容非常有用。

4. 可选： 附加 **-ginkgo.v** 用来运行测试，并增加详细程度。

5. 可选： 要针对特定的性能配置集运行延迟测试，请运行以下命令，替换适当的值：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e LATENCY_TEST_RUN=true -e LATENCY_TEST_RUNTIME=600 -e
OSLAT_MAXIMUM_LATENCY=20 \
-e PERF_TEST_PROFILE=<performance_profile> registry.redhat.io/openshift4/cnf-tests-
rhel8:v4.8 \
/usr/bin/test-run.sh -ginkgo.focus="[performance]\ Latency\ Test"
```

其中：

<performance_profile>
是您要对其运行延迟测试的性能配置集的名称。

重要

如需有效延迟测试结果，请至少运行测试 12 小时。

18.4.1. 运行 oslat

oslat 测试模拟 CPU 密集型 DPDK 应用程序，并测量所有中断和中断来测试集群处理 CPU 大量数据处理的方式。

重要

始终使用 **DISCOVERY_MODE=true** 设置运行延迟测试。如果没有，测试套件将对正在运行的集群配置进行更改。

注意

当以非 root 用户或非特权用户执行 **podman** 命令时，挂载路径可能会失败，错误为 **permission denied**。要使 **podman** 命令正常工作，请将 **:Z** 附加到卷创建中，例如 **-v $(pwd)/:/kubeconfig:Z**。这允许 **podman** 进行正确的 SELinux 重新标记。

先决条件

- 您使用客户门户网站凭证登录到 **registry.redhat.io**。
- 已使用 Performance addon operator 应用集群性能配置集。

流程

- 要执行 **oslat** 测试，请运行以下命令，根据需要替换变量值：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e LATENCY_TEST_RUN=true -e DISCOVERY_MODE=true -e
ROLE_WORKER_CNF=worker-cnf \
-e LATENCY_TEST_CPUS=7 -e LATENCY_TEST_RUNTIME=600 -e
```

```
OSLAT_MAXIMUM_LATENCY=20 \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
/usr/bin/test-run.sh -ginkgo.v -ginkgo.focus="oslat"
```

LATENCY_TEST_CPUS 指定使用 **oslat** 命令测试的 CPU 列表。

命令运行 **oslat** 工具 10 分钟（600 秒）。当观察到的最大延迟低于 **OSLAT_MAXIMUM_LATENCY** (20 μs)时，测试会成功运行。

如果结果超过延迟阈值，测试会失败。

重要

对于有效结果，测试应至少运行 12 小时。

失败输出示例

```
running /usr/bin//validationsuite -ginkgo.v -ginkgo.focus=oslat
I0829 12:36:55.386776       8 request.go:668] Waited for 1.000303471s due to client-side
throttling, not priority and fairness, request:
GET:https://api.cnfdc8.t5g.lab.eng.bos.redhat.com:6443/apis/authentication.k8s.io/v1?
timeout=32s
Running Suite: CNF Features e2e validation
==========================================

Discovery mode enabled, skipping setup
running /usr/bin//cnftests -ginkgo.v -ginkgo.focus=oslat
I0829 12:37:01.219077      20 request.go:668] Waited for 1.050010755s due to client-side
throttling, not priority and fairness, request:
GET:https://api.cnfdc8.t5g.lab.eng.bos.redhat.com:6443/apis/snapshot.storage.k8s.io/v1beta1?
timeout=32s
Running Suite: CNF Features e2e integration tests
================================================
Random Seed: 1630240617
Will run 1 of 142 specs

SSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSSS
------------------------------
[performance] Latency Test with the oslat image
  should succeed
  /go/src/github.com/openshift-kni/cnf-features-deploy/vendor/github.com/openshift-
kni/performance-addon-operators/functests/4_latency/latency.go:134
STEP: Waiting two minutes to download the latencyTest image
STEP: Waiting another two minutes to give enough time for the cluster to move the pod to
Succeeded phase
Aug 29 12:37:59.324: [INFO]: found mcd machine-config-daemon-wf4w8 for node
cnfdc8.clus2.t5g.lab.eng.bos.redhat.com

• Failure [49.246 seconds]
[performance] Latency Test
/go/src/github.com/openshift-kni/cnf-features-deploy/vendor/github.com/openshift-
kni/performance-addon-operators/functests/4_latency/latency.go:59
  with the oslat image
  /go/src/github.com/openshift-kni/cnf-features-deploy/vendor/github.com/openshift-
kni/performance-addon-operators/functests/4_latency/latency.go:112
```

```
    should succeed [It]
    /go/src/github.com/openshift-kni/cnf-features-deploy/vendor/github.com/openshift-
kni/performance-addon-operators/functests/4_latency/latency.go:134

    The current latency 27 is bigger than the expected one 20 [1]
    Expected
        <bool>: false
    to be true
 /go/src/github.com/openshift-kni/cnf-features-deploy/vendor/github.com/openshift-
kni/performance-addon-operators/functests/4_latency/latency.go:168

Log file created at: 2021/08/29 13:25:21
Running on machine: oslat-57c2g
Binary: Built with gc go1.16.6 for linux/amd64
Log line format: [IWEF]mmdd hh:mm:ss.uuuuuu threadid file:line] msg
I0829 13:25:21.569182       1 node.go:37] Environment information: /proc/cmdline:
BOOT_IMAGE=(hd0,gpt3)/ostree/rhcos-
612d89f4519a53ad0b1a132f4add78372661bfb3994f5fe115654971aa58a543/vmlinuz-
4.18.0-305.10.2.rt7.83.el8_4.x86_64 ip=dhcp random.trust_cpu=on console=tty0
console=ttyS0,115200n8
ostree=/ostree/boot.0/rhcos/612d89f4519a53ad0b1a132f4add78372661bfb3994f5fe11565497
1aa58a543/0 ignition.platform.id=openstack root=UUID=5a4ddf16-9372-44d9-ac4e-
3ee329e16ab3 rw rootflags=prjquota skew_tick=1 nohz=on rcu_nocbs=1-3
tuned.non_isolcpus=000000ff,ffffffff,ffffffff,ffffffff1 intel_pstate=disable nosoftlockup
tsc=nowatchdog intel_iommu=on iommu=pt isolcpus=managed_irq,1-3
systemd.cpu_affinity=0,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,
29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,
59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,
89,90,91,92,93,94,95,96,97,98,99,100,101,102,103 default_hugepagesz=1G
hugepagesz=2M hugepages=128 nmi_watchdog=0 audit=0 mce=off
processor.max_cstate=1 idle=poll intel_idle.max_cstate=0
I0829 13:25:21.569345       1 node.go:44] Environment information: kernel version 4.18.0-
305.10.2.rt7.83.el8_4.x86_64
I0829 13:25:21.569367       1 main.go:53] Running the oslat command with arguments \
[--duration 600 --rtprio 1 --cpu-list 4,6,52,54,56,58 --cpu-main-thread 2]
I0829 13:35:22.632263       1 main.go:59] Succeeded to run the oslat command: oslat V 2.00
Total runtime:    600 seconds
Thread priority:  SCHED_FIFO:1
CPU list:     4,6,52,54,56,58
CPU for main thread:  2
Workload:     no
Workload mem:     0 (KiB)
Preheat cores:    6

Pre-heat for 1 seconds...
Test starts...
Test completed.

        Core:  4 6 52 54 56 58
    CPU Freq:  2096 2096 2096 2096 2096 2096 (Mhz)
    001 (us):  19390720316 19141129810 20265099129 20280959461 19391991159
19119877333
    002 (us):  5304 5249 5777 5947 6829 4971
    003 (us):  28 14 434 47 208 21
    004 (us):  1388 853 123568 152817 5576 0
    005 (us):  207850 223544 103827 91812 227236 231563
```

```
  006 (us):  60770 122038 277581 323120 122633 122357
  007 (us):  280023 223992 63016 25896 214194 218395
  008 (us):  40604 25152 24368 4264 24440 25115
  009 (us):  6858 3065 5815 810 3286 2116
  010 (us):  1947 936 1452 151 474 361
...
   Minimum:  1 1 1 1 1 1 (us)
   Average:  1.000 1.000 1.000 1.000 1.000 1.000 (us)
   Maximum:  37 38 49 28 28 19 (us)
   Max-Min:  36 37 48 27 27 18 (us)
  Duration:  599.667 599.667 599.667 599.667 599.667 599.667 (sec)
```

1 在本例中，测量的延迟超出了最大允许的值。

18.5. 生成延迟测试失败报告

使用以下步骤生成 JUnit 延迟测试输出和测试失败报告。

先决条件

- 已安装 OpenShift CLI(**oc**)。
- 您已以具有 **cluster-admin** 权限的用户身份登录。

流程

- 使用集群状态和资源的信息创建测试失败报告，通过传递 **--report** 参数并使用报告转储的路径来进行故障排除：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -v $(pwd)/reportdest:<report_folder_path> \
-e KUBECONFIG=/kubeconfig/kubeconfig  -e DISCOVERY_MODE=true \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
/usr/bin/test-run.sh --report <report_folder_path> \
-ginkgo.focus="\[performance\]\ Latency\ Test"
```

其中：

<report_folder_path>
是生成报告的文件夹的路径。

18.6. 生成 JUNIT 延迟测试报告

使用以下步骤生成 JUnit 延迟测试输出和测试失败报告。

先决条件

- 已安装 OpenShift CLI(**oc**)。
- 您已以具有 **cluster-admin** 权限的用户身份登录。

流程

- 通过传递 **--junit** 参数和转储报告的路径来创建兼容 JUnit 的 XML 报告：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -v $(pwd)/junitdest:<junit_folder_path> \
-e KUBECONFIG=/kubeconfig/kubeconfig  -e DISCOVERY_MODE=true \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
/usr/bin/test-run.sh --junit <junit_folder_path> \
-ginkgo.focus="\[performance\]\ Latency\ Test"
```

其中：

<junit_folder_path>

是生成 junit 报告的文件夹的路径

18.7. 在断开连接的集群中运行延迟测试

CNF 测试镜像可在无法访问外部 registry 的断开连接的集群中运行测试。这需要两个步骤：

1. 将 **cnf-tests** 镜像镜像到自定义断开连接的 registry。
2. 指示测试使用来自自定义断开连接的 registry 的镜像。

将镜像镜像(mirror)到集群可访问的自定义 registry
mirror 中提供了镜像可执行文件，以提供 **oc** 需要的输入来镜像运行测试到本地 registry 所需的镜像。

1. 从可访问集群和 registry.redhat.io 的中间机器运行这个命令：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
/usr/bin/mirror -registry <disconnected_registry> | oc image mirror -f -
```

其中：

<disconnected_registry>

是您配置的断开连接的镜像 registry，如 **my.local.registry:5000/**。

2. 当您将 **cnf-tests** 镜像 mirror 到断开连接的 registry 中时，您必须覆盖用于运行测试时用来获取镜像的原始 registry，例如：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e DISCOVERY_MODE=true -e IMAGE_REGISTRY="<disconnected_registry>" \
-e CNF_TESTS_IMAGE="cnf-tests-rhel8:v4.8" \
/usr/bin/test-run.sh -ginkgo.focus="\[performance\]\ Latency\ Test"
```

配置测试以使用自定义 registry 中的镜像
您可以使用 **CNF_TESTS_IMAGE** 和 **IMAGE_REGISTRY** 变量来使用自定义测试镜像和镜像 registry 运行延迟测试。

- 要将延迟测试配置为使用自定义测试镜像和镜像 registry，请运行以下命令：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e IMAGE_REGISTRY="<custom_image_registry>" \
-e CNF_TESTS_IMAGE="<custom_cnf-tests_image>" \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 /usr/bin/test-run.sh
```

其中：

<custom_image_registry>

是自定义镜像 registry，如 **custom.registry:5000/**。

<custom_cnf-tests_image>

是自定义 cnf-tests 镜像，如 **custom-cnf-tests-image:latest**。

将镜像 mirror 到集群内部 registry

OpenShift Container Platform 提供了一个内建的容器镜像 registry，它作为一个标准的工作负载在集群中运行。

流程

1. 通过使用路由公开到 registry 的外部访问权限：

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster --patch '{"spec":
{"defaultRoute":true}}' --type=merge
```

2. 运行以下命令来获取 registry 端点：

```
$ REGISTRY=$(oc get route default-route -n openshift-image-registry --template='{{
.spec.host }}')
```

3. 创建用于公开镜像的命名空间：

```
$ oc create ns cnftests
```

4. 使镜像流可供用于测试的所有命名空间使用。这需要允许 test 命名空间从 **cnf-tests** 镜像流中获取镜像。运行以下命令：

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:cnf-features-
testing:default --namespace=cnftests
```

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:performance-addon-
operators-testing:default --namespace=cnftests
```

5. 运行以下命令，检索 docker secret 名称和 auth 令牌：

```
$ SECRET=$(oc -n cnftests get secret | grep builder-docker | awk {'print $1'}
```

```
$ TOKEN=$(oc -n cnftests get secret $SECRET -o jsonpath="{.data['\.dockercfg']}" | base64
--decode | jq '.["image-registry.openshift-image-registry.svc:5000"].auth')
```

6. 创建 **dockerauth.json** 文件，例如：

```
$ echo "{\"auths\": { \"$REGISTRY\": { \"auth\": $TOKEN } }}" > dockerauth.json
```

7. 对镜像进行 mirror：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
registry.redhat.io/openshift4/cnf-tests-rhel8:4.8 \
/usr/bin/mirror -registry $REGISTRY/cnftests |  oc image mirror --insecure=true \
-a=$(pwd)/dockerauth.json -f -
```

8. 运行测试：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
-e DISCOVERY_MODE=true -e IMAGE_REGISTRY=image-registry.openshift-image-
registry.svc:5000/cnftests \
cnf-tests-local:latest /usr/bin/test-run.sh -ginkgo.focus="\[performance\]\ Latency\ Test"
```

对不同的测试镜像进行镜像(mirror)

您可以选择更改对延迟测试镜像的默认上游镜像。

流程

1. **mirror** 命令默认尝试对上游镜像进行 mirror。这可以通过向镜像传递带有以下格式的文件来覆盖：

```
[
  {
    "registry": "public.registry.io:5000",
    "image": "imageforcnftests:4.8"
  }
]
```

2. 将文件传递给 **mirror** 命令，例如将其在本地保存为 **images.json**。使用以下命令，本地路径挂载到容器内的 **/kubeconfig** 中，并可传递给 mirror 命令。

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 /usr/bin/mirror \
--registry "my.local.registry:5000/" --images "/kubeconfig/images.json" \
| oc image mirror -f -
```

18.8. 对 CNF-TESTS 容器的错误进行故障排除

要运行延迟测试，集群必须从 **cnf-tests** 容器中访问。

先决条件

- 已安装 OpenShift CLI(**oc**)。
- 您已以具有 **cluster-admin** 权限的用户身份登录。

流程

- 运行以下命令，验证可以从 **cnf-tests** 容器中访问集群：

```
$ podman run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig \
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.8 \
oc get nodes
```

如果这个命令无法正常工作，则可能会出现与跨 DNS、MTU 大小或防火墙访问相关的错误。

第 19 章 创建性能配置集

了解 Performance Profile Creator（PPC），以及如何使用它来创建性能配置集。

19.1. 关于性能配置集创建器

Performance Profile Creator（PPC）是一个命令行工具，附带 Performance Addon Operator，用于创建性能配置集。该工具消耗来自集群的 **must-gather** 数据以及几个用户提供的配置集参数。PPC 生成适合您的硬件和拓扑的性能配置集。

该工具使用以下方法之一运行：

- 调用 **podman**
- 调用一个打包程序脚本

19.1.1. 使用 must-gather 命令收集有关集群的数据

Performance Profile Creator（PPC）工具需要 **must-gather** 数据。作为集群管理员，运行 **must-gather** 命令来捕获集群的信息。

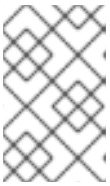
先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 访问 Performance Addon Operator 镜像。
- 已安装 OpenShift CLI（**oc**）。

流程

1. 进入存储 **must-gather** 数据的目录。
2. 在集群中运行 **must-gather**：

```
$ oc adm must-gather --image=<PAO_image> --dest-dir=<dir>
```



注意

must-gather 命令必须使用 **performance-addon-operator-must-gather** 镜像运行。输出可以被压缩（可选）。如果您正在运行性能配置集 Creator wrapper 脚本，则需要压缩输出。

示例

```
$ oc adm must-gather --image=registry.redhat.io/openshift4/performance-addon-operator-must-gather-rhel8:v4.8 --dest-dir=must-gather
```

3. 从 **must-gather** 目录创建一个压缩文件：

```
$ tar cvaf must-gather.tar.gz must-gather/
```

19.1.2. 使用 podman 运行 Performance Profile Creator

作为集群管理员，您可以运行 **podman** 和 Performance Profile Creator 来创建性能配置集。

先决条件

- 使用具有 **cluster-admin** 角色的用户访问集群。
- 在裸机硬件上安装的集群。
- 安装了 **podman** 和 OpenShift CLI（**oc**）的节点。

流程

1. 检查机器配置池：

```
$ oc get mcp
```

输出示例

```
NAME         CONFIG                                             UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master       rendered-master-acd1358917e9f98cbdb599aea622d78b       True      False
False      3              3                 3                   0                   22h
worker-cnf   rendered-worker-cnf-1d871ac76e1951d32b2fe92369879826   False     True
False      2              1                 1                   0                   22h
```

2. 使用 Podman 向 **registry.redhat.io** 进行身份验证：

```
$ podman login registry.redhat.io
```

```
Username: myrhusername
Password: ************
```

3. 可选：显示 PPC 工具的帮助信息：

```
$ podman run --entrypoint performance-profile-creator
registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8 -h
```

输出示例

```
A tool that automates creation of Performance Profiles

Usage:
  performance-profile-creator [flags]

Flags:
      --disable-ht                 Disable Hyperthreading
  -h, --help                       help for performance-profile-creator
      --info string                Show cluster information; requires --must-gather-dir-path,
ignore the other arguments. [Valid values: log, json] (default "log")
      --mcp-name string            MCP name corresponding to the target machines
```

```
(required)
      --must-gather-dir-path string       Must gather directory path (default "must-gather")
      --power-consumption-mode string     The power consumption mode.  [Valid values:
default, low-latency, ultra-low-latency] (default "default")
      --profile-name string               Name of the performance profile to be created (default
"performance")
      --reserved-cpu-count int            Number of reserved CPUs (required)
      --rt-kernel                         Enable Real Time Kernel (required)
      --split-reserved-cpus-across-numa   Split the Reserved CPUs across NUMA nodes
      --topology-manager-policy string    Kubelet Topology Manager Policy of the performance
profile to be created. [Valid values: single-numa-node, best-effort, restricted] (default
"restricted")
      --user-level-networking             Run with User level Networking(DPDK) enabled
```

4. 以发现模式运行 Performance Profile Creator 工具：

注意

发现模式使用 **must-gather** 的输出来检查您的集群。生成的输出包括以下信息：

- 使用分配的 CPU ID 进行 NUMA 单元分区
- 是否启用超线程

使用此信息，您可以为提供给 Performance Profile Creator 工具的部分参数设置适当的值。

```
$ podman run --entrypoint performance-profile-creator -v /must-gather:/must-gather:z
registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8 --info log --must-gather-
dir-path /must-gather
```

注意

此命令使用性能配置集创建器作为 **podman** 的新入口点。它将主机的 **must-gather** 数据映射到容器镜像，并调用所需的用户提供的配置集参数来生成 **my-performance-profile.yaml** 文件。

-v 选项可以是到以下的任一路径：

- **must-gather** 输出目录
- 包含 **must-gather** 解压缩 tarball 的现有目录

info 选项要求值指定输出格式。可能的值有 log 和 JSON。JSON 格式被保留用于调试。

5. 运行 **podman**：

```
$ podman run --entrypoint performance-profile-creator -v /must-gather:/must-gather:z
registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8 --mcp-name=worker-cnf
--reserved-cpu-count=20 --rt-kernel=true --split-reserved-cpus-across-numa=false --topology-
manager-policy=single-numa-node --must-gather-dir-path /must-gather  --power-
consumption-mode=ultra-low-latency > my-performance-profile.yaml
```

注意

Performance Profile Creator 参数显示在 Performance Profile Creator 参数表中。需要以下参数：

- **reserved-cpu-count**
- **mcp-name**
- **rt-kernel**

本例中的 **mcp-name** 参数根据 **oc get mcp** 命令的输出设置为 **worker-cnf**。对于单节点 OpenShift（SNO），使用 **--mcp-name=master**。

6. 查看创建的 YAML 文件：

```
$ cat my-performance-profile.yaml
```

输出示例

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: performance
spec:
  additionalKernelArgs:
  - nmi_watchdog=0
  - audit=0
  - mce=off
  - processor.max_cstate=1
  - intel_idle.max_cstate=0
  - idle=poll
  cpu:
    isolated: 1,3,5,7,9,11,13,15,17,19-39,41,43,45,47,49,51,53,55,57,59-79
    reserved: 0,2,4,6,8,10,12,14,16,18,40,42,44,46,48,50,52,54,56,58
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
  numa:
    topologyPolicy: single-numa-node
  realTimeKernel:
    enabled: true
```

7. 应用生成的配置集：

注意

在应用配置集前安装 Performance Addon Operator。

```
$ oc apply -f my-performance-profile.yaml
```

19.1.2.1. 如何运行 podman 创建性能配置集

以下示例演示了如何运行 **podman** 来创建具有 20 个保留 CPU 的性能配置集，这些 CPU 将在 NUMA 节点之间拆分。

节点硬件配置：

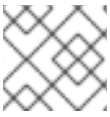
- 80 个 CPU
- 启用超线程
- 两个 NUMA 节点
- 编号为偶数的 CPU 在 NUMA 节点 0 上运行，编号为奇数的 CPU 在 NUMA 节点 1 上运行

运行 **podman** 以创建性能配置集：

```
$ podman run --entrypoint performance-profile-creator -v /must-gather:/must-gather:z
registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8 --mcp-name=worker-cnf --
reserved-cpu-count=20 --rt-kernel=true --split-reserved-cpus-across-numa=true --must-gather-dir-
path /must-gather > my-performance-profile.yaml
```

创建的配置集在以下 YAML 中描述：

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: performance
spec:
  cpu:
    isolated: 10-39,50-79
    reserved: 0-9,40-49
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
  numa:
    topologyPolicy: restricted
  realTimeKernel:
    enabled: true
```



注意

在这种情况下，在 NUMA 节点 0 上保留 10 个 CPU，NUMA 节点 1 上保留 10 个 CPU。

19.1.3. 运行性能配置集 Creator wrapper 脚本

性能配置集打包程序脚本简化了性能配置文件 Creator（PPC）工具的运行。它隐藏了运行 **podman** 的复杂性并指定映射目录，它支持创建性能配置集。

先决条件

- 访问 Performance Addon Operator 镜像。
- 访问 **must-gather** tarball。

流程

1. 在本地机器上创建一个文件，例如 **run-perf-profile-creator.sh**：

```
$ vi run-perf-profile-creator.sh
```

2. 将以下代码粘贴到文件中：

```
#!/bin/bash

readonly CONTAINER_RUNTIME=${CONTAINER_RUNTIME:-podman}
readonly CURRENT_SCRIPT=$(basename "$0")
readonly CMD="${CONTAINER_RUNTIME} run --entrypoint performance-profile-creator"
readonly IMG_EXISTS_CMD="${CONTAINER_RUNTIME} image exists"
readonly IMG_PULL_CMD="${CONTAINER_RUNTIME} image pull"
readonly MUST_GATHER_VOL="/must-gather"

PAO_IMG="registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8"
MG_TARBALL=""
DATA_DIR=""

usage() {
  print "Wrapper usage:"
  print "  ${CURRENT_SCRIPT} [-h] [-p image][-t path] -- [performance-profile-creator flags]"
  print ""
  print "Options:"
  print "   -h              help for ${CURRENT_SCRIPT}"
  print "   -p              Performance Addon Operator image"
  print "   -t              path to a must-gather tarball"

  ${IMG_EXISTS_CMD} "${PAO_IMG}" && ${CMD} "${PAO_IMG}" -h
}

function cleanup {
  [ -d "${DATA_DIR}" ] && rm -rf "${DATA_DIR}"
}
trap cleanup EXIT

exit_error() {
  print "error: $*"
  usage
  exit 1
}

print() {
  echo  "$*" >&2
}

check_requirements() {
  ${IMG_EXISTS_CMD} "${PAO_IMG}" || ${IMG_PULL_CMD} "${PAO_IMG}" || \
     exit_error "Performance Addon Operator image not found"

  [ -n "${MG_TARBALL}" ] || exit_error "Must-gather tarball file path is mandatory"
  [ -f "${MG_TARBALL}" ] || exit_error "Must-gather tarball file not found"

  DATA_DIR=$(mktemp -d -t "${CURRENT_SCRIPT}XXXX") || exit_error "Cannot create the
data directory"
  tar -zxf "${MG_TARBALL}" --directory "${DATA_DIR}" || exit_error "Cannot decompress the
must-gather tarball"
```

```
  chmod a+rx "${DATA_DIR}"

  return 0
}

main() {
  while getopts ':hp:t:' OPT; do
    case "${OPT}" in
      h)
        usage
        exit 0
        ;;
      p)
        PAO_IMG="${OPTARG}"
        ;;
      t)
        MG_TARBALL="${OPTARG}"
        ;;
      ?)
        exit_error "invalid argument: ${OPTARG}"
        ;;
    esac
  done
  shift $((OPTIND - 1))

  check_requirements || exit 1

  ${CMD} -v "${DATA_DIR}:${MUST_GATHER_VOL}:z" "${PAO_IMG}" "$@" --must-gather-
dir-path "${MUST_GATHER_VOL}"
  echo "" 1>&2
}

main "$@"
```

3. 为这个脚本中的每个人添加执行权限：

```
$ chmod a+x run-perf-profile-creator.sh
```

4. 可选：显示 **run-perf-profile-creator.sh** 命令用法：

```
$ ./run-perf-profile-creator.sh -h
```

预期输出

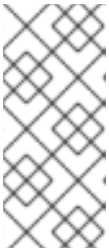
```
Wrapper usage:
  run-perf-profile-creator.sh [-h] [-p image][-t path] -- [performance-profile-creator flags]

Options:
   -h              help for run-perf-profile-creator.sh
   -p              Performance Addon Operator image (1)
   -t             path to a must-gather tarball (2)

A tool that automates creation of Performance Profiles
```

```
  Usage:
    performance-profile-creator [flags]

  Flags:
        --disable-ht                 Disable Hyperthreading
    -h, --help                       help for performance-profile-creator
        --info string                Show cluster information; requires --must-gather-dir-path,
ignore the other arguments. [Valid values: log, json] (default "log")
        --mcp-name string            MCP name corresponding to the target machines
(required)
        --must-gather-dir-path string  Must gather directory path (default "must-gather")
        --power-consumption-mode string  The power consumption mode.  [Valid values:
default, low-latency, ultra-low-latency] (default "default")
        --profile-name string        Name of the performance profile to be created (default
"performance")
        --reserved-cpu-count int     Number of reserved CPUs (required)
        --rt-kernel                  Enable Real Time Kernel (required)
        --split-reserved-cpus-across-numa  Split the Reserved CPUs across NUMA nodes
        --topology-manager-policy string  Kubelet Topology Manager Policy of the
performance profile to be created. [Valid values: single-numa-node, best-effort, restricted]
(default "restricted")
        --user-level-networking      Run with User level Networking(DPDK) enabled
```



注意

有两个参数类型：

- wrapper 参数，即 **-h**、**-p** 和 **-t**
- PPC 参数

1. 可选：指定 Performance Addon Operator 镜像。如果没有设置，则使用默认上游镜像：**registry.redhat.io/openshift4/performance-addon-rhel8-operator:v4.8**。
2. **-t** 是必需的打包程序脚本参数，并指定 **must-gather** tarball 的路径。

5. 以发现模式运行性能配置集创建器工具：

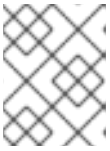
注意

发现模式使用 **must-gather** 的输出来检查您的集群。生成的输出包括以下信息：

- 使用分配的 CPU ID 进行 NUMA 单元分区
- 是否启用超线程

使用此信息，您可以为提供给 Performance Profile Creator 工具的部分参数设置适当的值。

```
$ ./run-perf-profile-creator.sh -t /must-gather/must-gather.tar.gz -- --info=log
```



注意

info 选项要求值指定输出格式。可能的值有 log 和 JSON。JSON 格式被保留用于调试。

6. 检查机器配置池：

```
$ oc get mcp
```

输出示例

```
NAME         CONFIG                                             UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master       rendered-master-acd1358917e9f98cbdb599aea622d78b       True      False
False      3              3                   3                   0                   22h
worker-cnf   rendered-worker-cnf-1d871ac76e1951d32b2fe92369879826   False     True
False      2              1                   1                   0                   22h
```

7. 创建性能配置集：

```
$ ./run-perf-profile-creator.sh -t /must-gather/must-gather.tar.gz -- --mcp-name=worker-cnf --
reserved-cpu-count=2 --rt-kernel=true > my-performance-profile.yaml
```

注意

Performance Profile Creator 参数显示在 Performance Profile Creator 参数表中。需要以下参数：

- **reserved-cpu-count**
- **mcp-name**
- **rt-kernel**

本例中的 **mcp-name** 参数根据 **oc get mcp** 命令的输出设置为 **worker-cnf**。对于单节点 OpenShift（SNO），使用 **--mcp-name=master**。

8. 查看创建的 YAML 文件：

```
$ cat my-performance-profile.yaml
```

输出示例

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: performance
spec:
  cpu:
    isolated: 1-39,41-79
    reserved: 0,40
  nodeSelector:
```

```
    node-role.kubernetes.io/worker-cnf: ""
  numa:
    topologyPolicy: restricted
  realTimeKernel:
    enabled: false
```

9. 应用生成的配置集：

注意

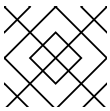
在应用配置集前安装 Performance Addon Operator。

```
$ oc apply -f my-performance-profile.yaml
```

19.1.4. Performance Profile Creator 参数

表 19.1. Performance Profile Creator 参数

参数	描述
disable-ht	禁用超线程。 可能的值： **true** 或 **false**。 默认值： **false**。 **警告** 如果此参数设为 **true**，则不应禁用 BIOS 中的超线程。禁用超线程通过内核命令行参数实现。
info	这会捕获集群信息，仅用于发现模式。发现模式还需要 **must-gather-dir-path** 参数。如果设置了任何其他参数，则忽略它们。 可能的值： • **log** • **JSON** **注意** 这些选项定义输出格式，以保留用于调试的 JSON 格式。 默认： **log**。
mcp-name	MCP 名称（如 **worker-cnf**）与目标机器对应。这个参数是必需的。

参数	描述
must-gather-dir-path	必须收集目录路径。这个参数是必需的。 当用户使用 wrapper 脚本 **must-gather** 运行该工具时，脚本本身会提供该工具，用户不得指定它。
power-consumption-mode	电源功耗模式。 可能的值： • **default** • **low-latency** • **ultra-low-latency** 默认： **default**。
profile-name	要创建的性能配置集的名称。默认：**performance**.
reserved-cpu-count	保留 CPU 的数量。这个参数是必需的。  **注意** 这必须是一个自然数字。不允许使用 0 值。
rt-kernel	启用实时内核。这个参数是必需的。 可能的值： **true** 或 **false**。
split-reserved-cpus-across-numa	将保留的 CPU 划分到 NUMA 节点。 可能的值： **true** 或 **false**。 默认值： **false**。
topology-manager-policy	要创建的性能配置集的 kubelet Topology Manager 策略。 可能的值： • **single-numa-node** • **best-effort** • **restricted** 默认： **restricted**。
user-level-networking	在启用了用户级别网络（DPDK）的情况下运行。 可能的值： **true** 或 **false**。 默认值： **false**。

19.2. 其他资源

- 有关 **must-gather** 工具的更多信息，请参阅收集集群的相关数据。