OpenShift Container Platform 4.3

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: 2020-11-30

OpenShift Container Platform 4.3 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 © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文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 安装大型集群的实践建议 **4**
1.1. 安装大型集群的实践建议 4

第 2 章 推荐的主机实践 **5**
2.1. 推荐的节点主机实践 5
2.2. 创建一个 KUBELETCONFIG CRD 来编辑 KUBELET 参数 5
2.3. 主节点大小 7
2.4. 推荐的 ETCD 实践 8
2.5. 其他资源 8

第 3 章 推荐的集群扩展实践 **9**
3.1. 扩展集群的建议实践 9
3.2. 修改 MACHINESET 9
3.3. 关于 MACHINEHEALTHCHECK 10
3.4. MACHINEHEALTHCHECK 资源示例 10
3.5. 创建 MACHINEHEALTHCHECK 资源 11

第 4 章 使用 NODE TUNING OPERATOR **12**
4.1. 关于 NODE TUNING OPERATOR 12
4.2. 访问 NODE TUNING OPERATOR 示例规格 12
4.3. 在集群中设置默认配置集 12
4.4. 验证是否应用了 TUNED 配置集 14
4.5. 自定义调整规格 15
4.6. 自定义调整示例 18
4.7. 支持的 TUNED 守护进程插件 19

第 5 章 使用 CLUSTER LOADER **20**
5.1. 安装 CLUSTER LOADER 20
5.2. 运行 CLUSTER LOADER 20
5.3. 配置 CLUSTER LOADER 20
5.4. 已知问题 24

第 6 章 使用 CPU MANAGER **26**
6.1. 设置 CPU MANAGER 26

第 7 章 使用拓扑管理器 **30**
7.1. 设置拓扑管理器 30
7.2. 拓扑管理器策略 31
7.3. POD 与拓扑管理器策略的交互 32

第 8 章 扩展 CLUSTER MONITORING OPERATOR **33**
8.1. PROMETHEUS 数据库存储要求 33
8.2. 配置集群监控 33

第 9 章 根据对象限制规划您的环境 **36**
9.1. 经过 OPENSHIFT CONTAINER PLATFORM 主发行版本测试的集群最大值 36
9.2. 经过 OPENSHIFT CONTAINER PLATFORM 测试的集群最大值 37
9.3. 测试集群最大值的 OPENSHIFT CONTAINER PLATFORM 环境和配置 38
9.4. 如何根据经过测试的集群限制规划您的环境 39
9.5. 如何根据应用程序要求规划您的环境 40

第 10 章 优化存储 **41**
10.1. 可用的持久性存储选项 41
10.2. 推荐的可配置存储技术 41

10.3. 数据存储管理 44
第 11 章 优化路由 46
11.1. 基础路由器性能 46
11.2. 路由器性能优化 47
第 12 章 巨页的作用及应用程序如何使用它们 48
12.1. 巨页的作用 48
12.2. 应用程序如何使用巨页 48
12.3. 配置巨页 49

第 1 章 安装大型集群的实践建议

在安装大型集群或把现有集群进行大规模扩展时，请应用以下实践建议。

1.1. 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 clusterNetwork 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。

第 2 章 推荐的主机实践

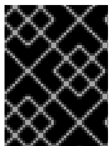
本节为 OpenShift Container Platform 提供推荐的主机实践。

2.1. 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。

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** 的值不能超过 **maxPods** 的值。

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
```

2.2. 创建一个 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 。这可让您创建 KubeletConfig 自定义资源 (CR) 来编辑 kubelet 参数。

流程

1. 运行：

```
$ oc get machineconfig
```

   这个命令显示了可选择的可用机器配置对象列表。默认情况下，与 kubelet 相关的配置为 **01-master-kubelet** 和 **01-worker-kubelet**。

2. 要检查每个节点中最大 Pod 数量的当前设置，请运行：

```
# oc describe node <node-ip> | grep Allocatable -A6
```

找到 **value: pods: <value>**.

例如：

```
# oc describe node ip-172-31-128-158.us-east-2.compute.internal | grep Allocatable -A6
```

输**出示例**

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

3. 要设置 worker 节点上的每个节点的最大 Pod，请创建一个包含 kubelet 配置的自定义资源文件。例如: **change-maxPods-cr.yaml**：

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

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 率：

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

a. 运行：

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

b. 运行：

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

c. 运行：

```
$ oc get kubeletconfig
```

这个命令会返回 **set-max-pods**。

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. 查看 worker 节点的 **maxPods** 的变化 ：

```
$ oc describe node
```

a. 运行以下命令验证更改：

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

这个命令会显示 **True** 状态和 **type:Success**

流程

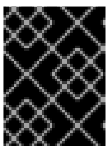
默认情况下，在对可用的 worker 节点应用 kubelet 相关的配置时，只允许一台机器不可用。对于大型集群来说，它可能需要很长时间才可以反映出配置的更改。在任何时候，您可以调整更新的机器数量来加快进程速度。

1. 运行：

```
$ oc edit machineconfigpool worker
```

2. 将 **maxUnavailable** 设为所需值。

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



重要

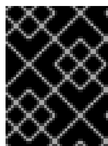
当设置该值时，请考虑无法使用的 worker 节点数量，而不影响在集群中运行的应用程序。

2.3. 主节点大小

主（master）节点对资源的要求取决于集群中的节点数量。以下推荐的主节点大小是基于 control plane 密度测试的结果。

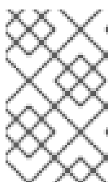
worker 节点数量	CPU 内核	内存 (GB)
25	4	16

worker 节点数量	CPU 内核	内存 (GB)
100	8	32
250	16	64



重要

因为无法修改正在运行的 OpenShift Container Platform 4.3 集群中的主节点大小，所以您必须估计节点总数并在安装过程中使用推荐的主节点大小。



注意

在 OpenShift Container Platform 4.3 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。

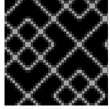
2.4. 推荐的 ETCD 实践

对于大型高密度的集群，如果键空间增长过大，超过空间配额，则 etcd 的性能可能会受到极大影响。因此，需要定期维护 etcd，包括整理碎片以便在数据存储中释放空间。强烈建议您密切监控 Prometheus 中的 etcd 指标数据，并提早进行碎片整理。否则，etcd 可能会引发一个集群范围的警告，使集群进入维护模式（只能对键进行读和删除）。需要密切关注的指标数据是 **etcd_server_quota_backend_bytes**，即当前的配额限制；**etcd_mvcc_db_total_size_in_use_in_in_bytes**，它显示了对历史数据进行压缩后的实际数据库用量；**etcd_debugging_mvcc_db_total_size_in_bytes**，它显示了数据库大小，包括等待进行碎片处理的空闲空间。

2.5. 其他资源

- OpenShift Container Platform 集群限制

第 3 章 推荐的集群扩展实践



重要

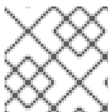
本节中的指导信息仅与使用云供应商集成的安装相关。

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。

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 MachineSet，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。

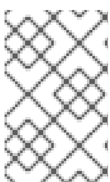
如果同时将 MachineSet 中的副本设置为更高数量，则控制器可能无法创建机器。部署 OpenShift Container Platform 的云平台可以处理的请求数量将会影响该进程。当尝试创建、检查和更新有状态的机器时，控制器会开始进行更多的查询。部署 OpenShift Container Platform 的云平台具有 API 请求限制，如果出现过量查询，则可能会因为云平台的限制而导致机器创建失败。

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。

3.2. 修改 MACHINESET

若要更改 MachineSet，请编辑 MachineSet YAML。然后，通过删除每个机器或将 MachineSet 副本数减为 0 来删除所有与 MachineSet 相关联的机器。然后，将副本数量调回所需的数量。您对 MachineSet 进行的更改不会影响现有的机器。

如果需要扩展 MachineSet 但不进行其他更改，则无需删除机器。



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 MachineSet 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 MachineSet：

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

2. 将 MachineSet 缩减为 **0**：

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

或者：

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

等待机器被删除。

3. 根据需要扩展 MachineSet：

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

或者：

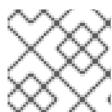
```
$ oc edit machineset <machineset> -n openshift-machine-api
```

等待机器启动。新 Machine 包含您对 MachineSet 所做的更改。

3.3. 关于 MACHINEHEALTHCHECK

MachineHealthCheck 可自动修复特定 MachinePool 中不正常的 Machine。

要监控机器的健康状况，您可以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 15 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

您无法将 MachineHealthCheck 应用到具有主控机（master）角色的机器。

监控 MachineHealthCheck 资源的控制器将检查是否出现了您定义的状态。如果机器不能通过健康检查，会自动被删除并创建新的机器来代替它。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数量大于 **maxUnhealthy** 的值，则补救会停止，以便手动进行干预。

若要停止检查，请删除其资源。

3.4. MACHINEHEALTHCHECK 资源示例

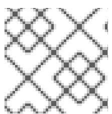
MachineHealthCheck 资源类似于以下 YAML 文件：

MachineHealthCheck

```
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)
```

1. 指定要部署的 MachineHealthCheck 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 MachineSet。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。如果设置的超时时间较长，则可能会导致不健康的机器上的工作负载长时间停机。
7. 指定目标机器池中允许的不健康机器的数量。这可设为一个百分比或一个整数。



注意

matchLabels 只是示例; 您必须根据具体需要映射您的机器组。

3.5. 创建 MACHINEHEALTHCHECK 资源

您可以为集群中除 **master** 池之外的所有 MachinePool 创建 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，其包含 MachineHealthCheck 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。Node Tuning Operator 把为 OpenShift Container Platform 容器化的 Tuned 守护进程作为一个 Kubernetes DaemonSet 进行管理。它保证了自定义性能优化设置以可被守护进程支持的格式传递到在集群中运行的所有容器化的 Tuned 守护进程中。相应的守护进程会在集群的所有节点上运行，每个节点上运行一个。

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

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

4.2. 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。

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/v1alpha1
kind: Tuned
metadata:
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - name: "openshift"
    data: |
      [main]
      summary=Optimize systems running OpenShift (parent profile)
      include=${f:virt_check:virtual-guest:throughput-performance}
      [selinux]
      avc_cache_threshold=8192
      [net]
      nf_conntrack_hashsize=131072
```

```
      [sysctl]
      net.ipv4.ip_forward=1
      kernel.pid_max=>131072
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.neigh.default.gc_thresh1=8192
      net.ipv4.neigh.default.gc_thresh2=32768
      net.ipv4.neigh.default.gc_thresh3=65536
      net.ipv6.neigh.default.gc_thresh1=8192
      net.ipv6.neigh.default.gc_thresh2=32768
      net.ipv6.neigh.default.gc_thresh3=65536
      [sysfs]
      /sys/module/nvme_core/parameters/io_timeout=4294967295
      /sys/module/nvme_core/parameters/max_retries=10
  - name: "openshift-control-plane"
    data: |
      [main]
      summary=Optimize systems running OpenShift control plane
      include=openshift
      [sysctl]
      # ktune sysctl settings, maximizing i/o throughput
      #
      # Minimal preemption granularity for CPU-bound tasks:
      # (default: 1 msec#  (1 + ilog(ncpus)), units: nanoseconds)
      kernel.sched_min_granularity_ns=10000000
      # The total time the scheduler will consider a migrated process
      # "cache hot" and thus less likely to be re-migrated
      # (system default is 500000, i.e. 0.5 ms)
      kernel.sched_migration_cost_ns=5000000
      # SCHED_OTHER wake-up granularity.
      #
      # Preemption granularity when tasks wake up.  Lower the value to
      # improve wake-up latency and throughput for latency critical tasks.
      kernel.sched_wakeup_granularity_ns=4000000
  - name: "openshift-node"
    data: |
      [main]
      summary=Optimize systems running OpenShift nodes
      include=openshift
      [sysctl]
      net.ipv4.tcp_fastopen=3
      fs.inotify.max_user_watches=65536
  - name: "openshift-control-plane-es"
    data: |
      [main]
      summary=Optimize systems running ES on OpenShift control-plane
      include=openshift-control-plane
      [sysctl]
      vm.max_map_count=262144
  - name: "openshift-node-es"
    data: |
      [main]
      summary=Optimize systems running ES on OpenShift nodes
      include=openshift-node
      [sysctl]
      vm.max_map_count=262144
  recommend:
```

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

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

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

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

4.4. 验证是否应用了 TUNED 配置集

使用这个流程检查在每个节点上应用了哪些 Tuned 配置集。

流程

1. 检查每个节点上运行的 Tuned Pod:

```
$ oc get pods -n openshift-cluster-node-tuning-operator -o wide
```

输**出示例**

```
NAME                                   READY   STATUS    RESTARTS   AGE    IP            NODE
NOMINATED NODE   READINESS GATES
cluster-node-tuning-operator-599489d4f7-k4hw4   1/1     Running   0          6d2h   10.129.0.76
ip-10-0-145-113.eu-west-3.compute.internal   <none>           <none>
tuned-2jkzp                            1/1     Running   1          6d3h   10.0.145.113   ip-10-0-145-
113.eu-west-3.compute.internal   <none>           <none>
tuned-g9mkx                            1/1     Running   1          6d3h   10.0.147.108   ip-10-0-
147-108.eu-west-3.compute.internal   <none>           <none>
tuned-kbxsh                            1/1     Running   1          6d3h   10.0.132.143   ip-10-0-132-
143.eu-west-3.compute.internal   <none>           <none>
tuned-kn9x6                            1/1     Running   1          6d3h   10.0.163.177   ip-10-0-163-
177.eu-west-3.compute.internal   <none>           <none>
tuned-vvxwx                            1/1     Running   1          6d3h   10.0.131.87    ip-10-0-131-
87.eu-west-3.compute.internal   <none>           <none>
tuned-zqrwq                            1/1     Running   1          6d3h   10.0.161.51    ip-10-0-161-
51.eu-west-3.compute.internal   <none>           <none>
```

2. 提取从每个 Pod 应用的配置集，并将它们与上一个列表中匹配：

```
$ for p in `oc get pods -n openshift-cluster-node-tuning-operator -l openshift-app=tuned -o=jsonpath='{range .items[*]}{.metadata.name} {end}'`; do printf "\n*** $p ***\n" ; oc logs pod/$p -n openshift-cluster-node-tuning-operator | grep applied; done
```

输出示例

```
*** tuned-2jkzp ***
2020-07-10 13:53:35,368 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-control-plane' applied

*** tuned-g9mkx ***
2020-07-10 14:07:17,089 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node' applied
2020-07-10 15:56:29,005 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node-es' applied
2020-07-10 16:00:19,006 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node' applied
2020-07-10 16:00:48,989 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node-es' applied

*** tuned-kbxsh ***
2020-07-10 13:53:30,565 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node' applied
2020-07-10 15:56:30,199 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node-es' applied

*** tuned-kn9x6 ***
2020-07-10 14:10:57,123 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node' applied
2020-07-10 15:56:28,757 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-node-es' applied

*** tuned-vvxwx ***
2020-07-10 14:11:44,932 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-control-plane' applied

*** tuned-zqrwq ***
2020-07-10 14:07:40,246 INFO     tuned.daemon.daemon: static tuning from profile 'openshift-control-plane' applied
```

重要

自定义性能优化规格的自定义配置集只是一个技术预览功能。红帽产品服务等级协议 (SLA) 不支持技术预览功能，且可能无法完成。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，请参阅 https://access.redhat.com/support/offerings/techpreview/。

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。

配置集数据

profile: 部分列出了 Tuned 配置集及其名称。

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

    [sysctl]
    net.ipv4.ip_forward=1
    # ... other sysctl's or other Tuned daemon plug-ins supported by the containerized Tuned

# ...

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

    # tuned_profile_n profile settings
```

建议的配置集

profile: 选择逻辑通过 CR 的 **recommend:** 部分来定义：

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

# ...

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

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

<match> 是一个递归定义的可选数组，如下所示：

```
- label: <label_name>     # node or Pod label name
  value: <label_value>    # optional node or Pod label value; if omitted, the presence of <label_name>
is enough to match
```

```
type: <label_type>     # optional node or Pod type (`node` or `pod`); if omitted, `node` is assumed
<match>                # an optional <match> array
```

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

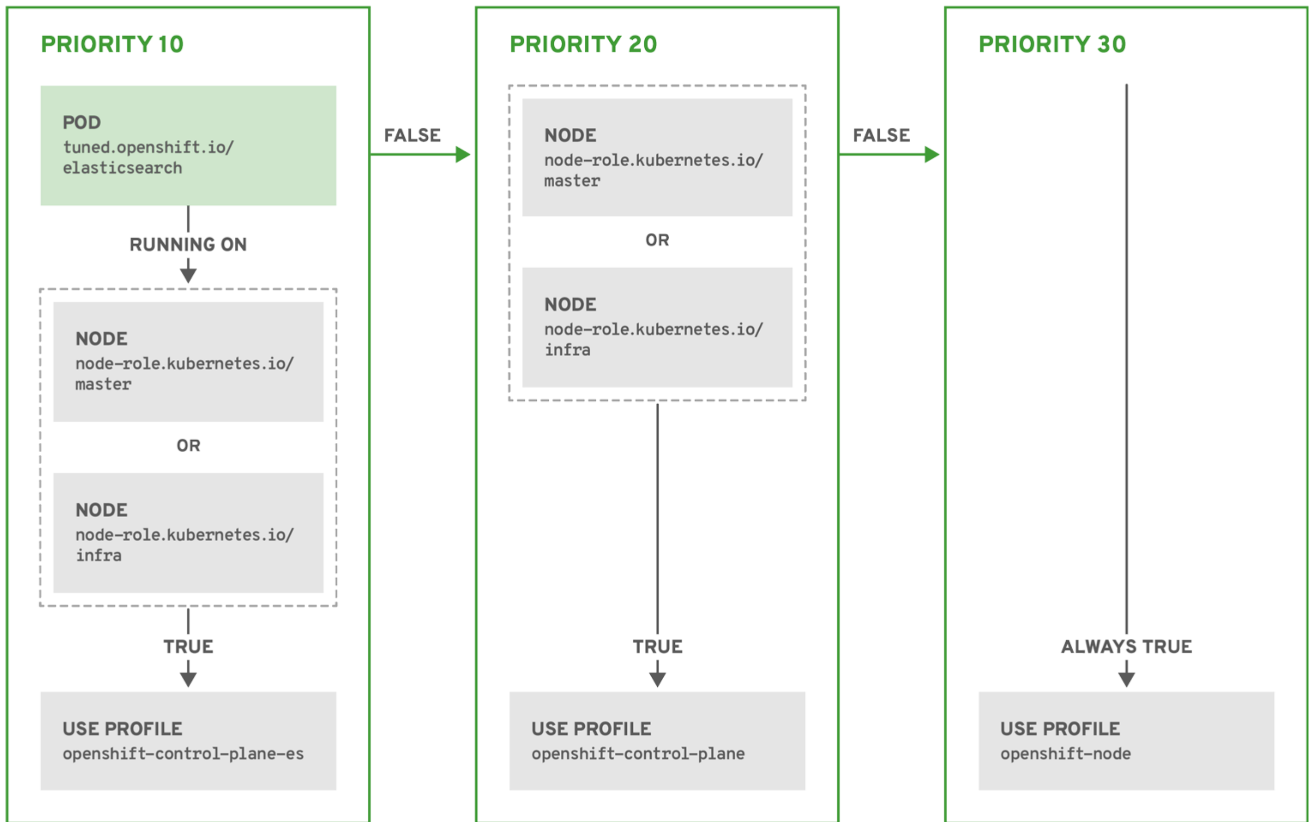
示例

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

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

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

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



4.6. 自定义调整示例

以下 CR 对运行带有标签 **tuned.openshift.io/ingress-pod-label=ingress-pod-label-value** 的一个 ingress pod 的OpenShift Container Platform 节点应用自定义节点级别调整。作为管理员，使用以下命令创建一个自定义 Tuned CR。

示例

```
oc create -f- <<_EOF_
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-pod-label
      value: "ingress-pod-label-value"
```

```
      type: pod
    priority: 10
    profile: openshift-ingress
_EOF_
```

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。以下 Tuned 插件目前还不支持：

- 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。

5.1. 安装 CLUSTER LOADER

流程

1. 要拉取容器镜像，请运行：

```
$ sudo podman pull quay.io/openshift/origin-tests:4.3
```

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：

```
$ sudo podman run -v ${LOCAL_KUBECONFIG}:/root/.kube/config:z -i \
quay.io/openshift/origin-tests:4.3 /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：

```
$ sudo podman run -v ${LOCAL_KUBECONFIG}:/root/.kube/config:z \
-v ${LOCAL_CONFIG_FILE_PATH}:/root/configs/:z \
-i quay.io/openshift/origin-tests:4.3 \
/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 ❶
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: ❷
    - name: default
      pods:
        stepping: ❸
          stepsize: 5
          pause: 0 s
        rate_limit: ❹
          delay: 0 ms
```

❶ 端到端测试的可选设置。设置为 **local** 以避免额外的日志信息。

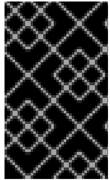
❷

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 podspec 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	每个配置都有的一个定义的子对象. **tuningsets** 允许用户定义一个调整集来为创建项目或对象（Pods 、模板等等）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 ConfigMap 名称，值是一个指向用来创建 ConfigMap 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 Podspec 或模板的本地文件路径。
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	布尔值等待带有匹配标签的 **Pod** 进入 **运行** 状态。
succeeded	布尔值等待带有标签的 **Pod** 进入**完成**状态。
selectors	匹配处于 **运行**或**完成**状态的 Pod 的选择器列表。
timeout	字符串等待处于**运行**或**完成**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. 在 worker **MachineConfigPool** 中添加一个标签：

```
metadata:
  creationTimestamp: 2019-xx-xxx
  generation: 3
  labels:
    custom-kubelet: cpumanager-enabled
```

4. 创建 **KubeletConfig**，**cpumanager-kubeletconfig.yaml**，自定义资源 (CR) 。使用上一步中创建的标签，以新的 **KubeletConfig**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
     cpuManagerReconcilePeriod: 5s
```

5. 创建动态 **KubeletConfig**：

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

这会在 **KubeletConfig** 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 **KubeletConfig**：

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

       "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-4.4# cat /host/etc/kubernetes/kubelet.conf | grep cpuManager
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。这是专用于这个 Pod 的内核数：

```
# cat cpumanager-pod.yaml
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
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
cpuset.cpus 1
tasks 32706
```

12. 检查任务允许的 CPU 列表：

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

```
...
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。将 **kube-reserved** 设定为 500 毫秒，这意味着，一个内核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：

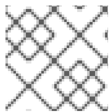
```
NAME                READY   STATUS    RESTARTS   AGE
cpumanager-6cqz7    1/1     Running   0          33m
cpumanager-7qc2t    0/1     Pending   0          11s
```

第 7 章 使用拓扑管理器

Topology Manager（拓扑管理器）是一个 Kubelet 组件，它从 CPU Manager 和设备管理器收集提示信息（hint），以匹配同一非统一内存访问 (NUMA) 节点上的 pod CPU 和设备资源。

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

拓扑管理器是 OpenShift Container Platform 中的一个 alpha 功能。

7.1. 设置拓扑管理器

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

1. 启用 LatencySensitive FeatureGate

```
# oc edit featuregate/cluster
```

2. 添加 Feature Set: LatencySensitive 到 spec：

```
# oc describe featuregate/cluster

Name:         cluster
Namespace:
Labels:       <none>
Annotations:  release.openshift.io/create-only: true
API Version:  config.openshift.io/v1
Kind:         FeatureGate
Metadata:
  Creation Timestamp:  2019-10-30T15:06:41Z
  Generation:          2
  Resource Version:    7773803
  Self Link:           /apis/config.openshift.io/v1/featuregates/cluster
  UID:                 b00204ab-cc5e-4ca5-ad93-b9bdd738c1de
Spec:
  Feature Set:  LatencySensitive
Events:         <none>
```

3. 使用 KubeletConfig 配置 Topology Manager 策略。
   以下 YAML 文件示例显示了一个指定的 **single-numa-node** 策略。

```
apiVersion: machineconfiguration.openshift.io/v1
kind: KubeletConfig
metadata:
  name: cpumanager-enabled
```

```
spec:
  machineConfigPoolSelector:
    matchLabels:
      custom-kubelet: cpumanager-enabled
  kubeletConfig:
     cpuManagerPolicy: static
     cpuManagerReconcilePeriod: 5s
     topologyManagerPolicy: single-numa-node 1
```

1 指定所选的拓扑管理器策略。

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

7.2. 拓扑管理器策略

拓扑管理器可用于满足以下条件的节点和 Pod 中：

- 节点的 CPU Manager 策略配置为 **static**。
- Pod 是 **Guaranteed** QoS 类。

当满足以上条件时，Topology Manager 会对 Pod 的 CPU 和设备请求进行调整。

拓扑管理器支持 4 个分配策略。这些策略通过 Kubelet 标志 **--topology-manager-policy** 进行设置。策略包括：

- **none** （默认）
- **best-effort**
- **restricted**
- **single-numa-node**

7.2.1. none 策略

这是默认策略，不执行任何拓扑对齐调整。

7.2.2. best-effort 策略

对于带有 best-effort 拓扑管理策略的 Guaranteed Pod 中的每个容器，kublet 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。

7.2.3. restricted 策略

对于带有 restricted 拓扑管理策略的 Guaranteed Pod 中的每个容器，kublet 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，则拓扑管理器会使节点拒绝该 pod。这会导致 pod 处于 Terminated 状态，且 pod 准入失败。

7.2.4. single-numa-node

对于带有 single-numa-node 拓扑管理策略的 Guaranteed Pod 中的每个容器，kublet 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。如果可能，pod 将会分配给该节点。如果不可能，则拓扑管理器将使节点拒绝 pod。这会导致 pod 处于 Terminated 状态，且 pod 准入失败。

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

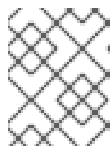
红帽对不同的扩展大小进行了各种测试。

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	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。

实验室环境

在以前的版本中，所有实验都是在 RHOSP 环境中的 OpenShift Container Platform 中进行的：

- infra nodes (VM) - 40 个内核，157 GB RAM。
- CNS 节点 (VM) - 16 个内核、62GB RAM 、nvme 驱动。

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

流程

为 Prometheus 增加存储容量：

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

```
apiVersion: v1
kind: ConfigMap
data:
  config.yaml: |
    prometheusOperator:
      baseImage: quay.io/coreos/prometheus-operator
      prometheusConfigReloaderBaseImage: quay.io/coreos/prometheus-config-reloader
      configReloaderBaseImage: quay.io/coreos/configmap-reload
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    prometheusK8s:
      retention: {{PROMETHEUS_RETENTION_PERIOD}} (1)
      baseImage: openshift/prometheus
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      volumeClaimTemplate:
        spec:
          storageClassName: gp2
          resources:
            requests:
              storage: {{PROMETHEUS_STORAGE_SIZE}} (2)
    alertmanagerMain:
      baseImage: openshift/prometheus-alertmanager
      nodeSelector:
        node-role.kubernetes.io/infra: ""
      volumeClaimTemplate:
        spec:
          storageClassName: gp2
          resources:
            requests:
              storage: {{ALERTMANAGER_STORAGE_SIZE}} (3)
    nodeExporter:
      baseImage: openshift/prometheus-node-exporter
    kubeRbacProxy:
      baseImage: quay.io/coreos/kube-rbac-proxy
    kubeStateMetrics:
      baseImage: quay.io/coreos/kube-state-metrics
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    grafana:
      baseImage: grafana/grafana
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    auth:
      baseImage: openshift/oauth-proxy
    k8sPrometheusAdapter:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
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. 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。

3. 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. 设置值，如保留周期和存储大小。

3. 运行以下命令应用这些更改：

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

第 9 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。

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。

9.1. 经过 OPENSHIFT CONTAINER PLATFORM 主发行版本测试的集群最大值

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
节点数量	2,000	2,000
Pod 数 [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。
命名空间的数量[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
服务数 [5]	10,000	10,000
每个命名空间的服务数	5,000	5,000
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

1. 这里的 pod 数量是测试 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 KubeletConfig 将 **maxPods** 设置为 **500** 来创建集群。如果需要 500 个用户 Pod，您需要一个 **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 v4 (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，包括通过碎片管理释放 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。

9.2. 经过 OPENSHIFT CONTAINER PLATFORM 测试的集群最大值

最大类型	3.11 测试的最大值	4.1 测试的最大值	4.2 测试的最大值	4.3 测试的最大值
节点数量	2,000	2,000	2,000	2,000
Pod 数 [1]	150,000	150,000	150,000	150,000
每个节点的 pod 数量	250	250	250	500
每个内核的 pod 数量	没有默认值。	没有默认值。	没有默认值。	没有默认值。
命名空间的数量 [2]	10,000	10,000	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）- 管道 (Pipeline) 策略	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 [3]	25,000	25,000	25,000	25,000
服务数 [4]	10,000	10,000	10,000	10,000
每个命名空间的服务数	5,000	5,000	5,000	5,000
每个服务中的后端数	5,000	5,000	5,000	5,000
每个命名空间的部署数量 [3]	2,000	2,000	2,000	2,000

1. 这里的 pod 数量是测试 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. 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，包括通过碎片管理释放 etcd 存储。

3. 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。

4. 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。

在 OpenShift Container Platform 4.3 中，与 OpenShift Container Platform 3.11 和之前的版本相比，系统会保留半个 CPU 内核（500 millicore）。

9.3. 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	io1	220 / 3000	3	us-west-2
Infra [2]	m5.12xlarge	48	192	gp2	100	3	us-west-2
Workload [3]	m5.4xlarge	16	64	gp2	500 [4]	1	us-west-2
Worker	m5.2xlarge	8	32	gp2	100	3/25/250/2000 [5]	us-west-2

1. 带有 3000 个 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。

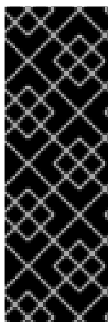
5. 在迭代中扩展了集群，且性能和可扩展性测试是在指定节点数中执行的。

Azure 云平台：

节点	Flavor	vCPU	RAM(GiB)	磁盘类型	磁盘大小 (GiB)/iops	数量	区域
Master/etcd [1]	Standard_D8s_v3	8	32	Premium SSD	1024 (P30)	3	centralus
Infra [2]	Standard_D16s_v3	16	64	Premium SSD	1024 (P30)	3	centralus
Worker	Standard_D4s_v3	4	16	Premium SSD	1024 (P30)	3/25/100/110 [3]	centralus

1. 对于高 iops 且流量会封顶，1024GB 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. 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。

9.4. 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

如果想把集群的规模限制在 2200 个 pod，假设每个节点最多有 250 个 pod，则需要最少 9 个节点：

```
2200 / 250 = 8.8
```

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

9.5. 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。

第 10 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。

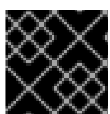
10.1. 可用的持久性存储选项

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

表 10.1. 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 [a]，和厂商 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

[a] NetApp NFS 在使用 Trident 插件时支持动态 PV 置备。



重要

目前，OpenShift Container Platform 4.3 不支持 CNS。

10.2. 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 10.2. 推荐的、可配置的存储技术

存储类型	ROX[1]	RWX[2]	Registry	扩展的 registry	指标[3]	日志记录	Apps
Block	是[4]	否	可配置	无法配置	推荐的	推荐的	推荐的
File	是[4]	是	可配置	可配置	可配置[5]	可配置[6]	推荐的
对象	是	是	推荐的	推荐的	无法配置	无法配置	无法配置[7]

[1] ReadOnlyMany

[2] ReadWriteMany.

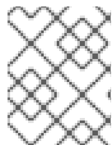
[3] Prometheus 是用于指标数据的底层技术。

[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 PersistentVolumeClaims 上配置 RWX 访问模式。

[6] 要进行日志记录，使用任何共享存储都会是一个反 pattern。每个 elasticsearch 都需要一个卷。

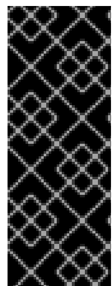
[7] 对象存储不是通过 OpenShift Container Platform 的 PV/持久性卷声明 (PVC) 消耗的。应用程序必须与对象存储 REST API 集成。



注意

扩展的 registry 是指一个 OpenShift Container Platform registry，它有两个或更多个 pod 运行副本。

10.2.1. 特定应用程序存储建议



重要

测试显示，在 RHEL 中使用 NFS 服务器作为核心服务的存储后端可能会出现问题。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，不推荐使用 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。

10.2.1.1. 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。

10.2.1.2. 扩展的 registry

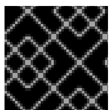
在扩展的/HA OpenShift Container Platform registry 集群部署中：

- 存储技术必须支持 RWX 访问模式，且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。
- 存储应兼容 S3 或 Swift。
- 对于应用于生产环境负载的扩展的/HA OpenShift Container Platform registry 集群部署，不建议使用文件存储。
- 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。

10.2.1.3. 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。



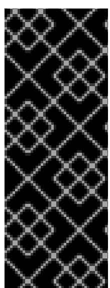
重要

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。

10.2.1.4. 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。
- 对于应用于生产环境负载的扩展的/HA OpenShift Container Platform registry 集群部署，不建议使用文件存储。
- 对象存储是不可配置的。



重要

测试显示，在 RHEL 中使用 NFS 服务器作为核心服务的存储后端可能会出现问题。这包括用于日志存储的 Elasticsearch。因此，不推荐使用 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。

10.2.1.5. 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。

10.2.2. 其他特定的应用程序存储建议

- OpenShift Container Platform 内部 **etcd**：为了获得最好的 **etcd** 可靠性，首选使用具有最低一致性延迟的存储技术。
- 强烈建议您使用带有可快速处理串口写入(fsync)的存储的 **etcd**，比如 NVMe 或者 SSD。不建议使用 Ceph、NFS 和 spinning 磁盘。
- 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。

10.3. 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 10.3. 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/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。**

目录	备注	大小	预期增长
/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 11 章 优化路由

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

11.1. 基础路由器性能

OpenShift Container Platform 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，一个单独的 HAProxy 路由器处理 100 个路由来提供后端的 1kB 静态页面的公共云实例中进行测试，其每秒的交易数如下。

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

使用 **ROUTER_THREADS=4** 默认路由器配置，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。路由器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路由，则应该使用路由器分片（router sharding）以帮助横向扩展路由层。

11.2. 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路由器的部署：**ROUTER_THREADS**、**ROUTER_DEFAULT_TUNNEL_TIMEOUT**、**ROUTER_DEFAULT_CLIENT_TIMEOUT**、**ROUTER_DEFAULT_SERVER_TIMEOUT** 和 **RELOAD_INTERVAL**。

您可以修改路由器部署，但当 Ingress Operator 被启用时，其配置会被覆盖。

第 12 章 巨页的作用及应用程序如何使用它们

12.1. 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。

12.2. 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。

其他资源

- 配置 THG

12.3. 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 Node Tuning Operator 在特定节点中分配巨页。

流程

1. 为相关节点添加标签，以便 Node Tuning Operator 知道要应用 tuned 配置集的节点。tuned 配置集规定了要分配的巨页数量：

```
$ oc label node <node_using_hugepages> hugepages=true
```

2. 创建一个包含以下内容的名为 **hugepages_tuning.yaml** 的文件：

```
apiVersion: tuned.openshift.io/v1
kind: Tuned
metadata:
  name: hugepages (1)
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile: (2)
  - data: |
      [main]
      summary=Configuration for hugepages
      include=openshift-node

      [vm]
      transparent_hugepages=never

      [sysctl]
```

```
      vm.nr_hugepages=1024
    name: node-hugepages
  recommend:
  - match: 3
    - label: hugepages
    priority: 30
    profile: node-hugepages
```

1. 将 **name** 参数的值设置为 **hugepages**。
2. 将 **profile** 部分设置为分配巨页。
3. 将 **match** 部分设置为将配置集与带有 **hugepages** 标签的节点相关联。

3. 使用 **hugepages_tuning.yaml** 文件创建定制的 **hugepages** tuned 配置集：

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

4. 在创建配置集后，Operator 会将新配置集应用到正确的节点中，并分配巨页。在使用巨页的节点中检查 tuned pod 的日志以确认：

```
$ oc logs <tuned_pod_on_node_using_hugepages> \
    -n openshift-cluster-node-tuning-operator | grep 'applied$' | tail -n1

2019-08-08 07:20:41,286 INFO     tuned.daemon.daemon: static tuning from profile 'node-
hugepages' applied
```