OpenShift Container Platform 4.5

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: 2021-06-14

OpenShift Container Platform 4.5 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

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

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。

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** 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
```

2.2. 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

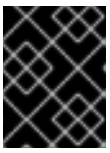
默认情况下，在对可用的 worker 节点应用 kubelet 相关的配置时，只允许一台机器不可用。对于大型集群来说，它可能需要很长时间才可以反映出配置的更改。在任何时候，您可以调整更新的机器数量来加快进程速度。

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. CONTROL PLANE 节点大小

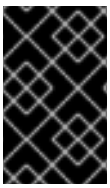
control plane 节点对资源的要求取决于集群中的节点数量。以下推荐的 control plane 节点大小是基于 control plane 密度测试的结果。control plane 测试会根据节点数在每个命名空间中在集群中创建以下对象：

- 12 个镜像流
- 3 个构建配置
- 6 个构建

- 1 个部署,每个 pod 副本挂载两个 secret
- 2 个部署,1 个 pod 副本挂载两个 secret
- 3 个服务指向以前的部署
- 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 节点的集群中，当一个节点停止、重新引导或失败时，CPU 和内存用量会增加，因为剩余的两个节点必须处理负载才能具有高可用性。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。为了避免大型和高密度的集群出现级联失败，请将 master 节点上的总资源使用量至少保留到所有可用容量的一半，以处理资源用量增加。相应地增加 master 节点上的 CPU 和内存。

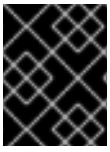


重要

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。

重要

因为无法修改正在运行的 OpenShift Container Platform 4.5 集群中的 control plane 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 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 4.5 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。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。有关 etcd 碎片整理的说明，请参考 **etcd 数据的碎片整理**部分。

etcd 将数据写入磁盘，因此其性能在很大程度上取决于磁盘性能。etcd 在磁盘上持久化。速度较慢的磁盘胡来自其他进程的磁盘活动可能会造成长时间的 fsync 延迟，,从而导致 etcd 丢失心跳信号，从而无法及时向磁盘提交新的操作，这可能导致请求超时并暂时丢失领导（leader）的功能。强烈建议您在由低延迟和高吞吐量的 SSD/NVMe 磁盘支持的机器上运行 etcd。

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。**etcd_disk_wal_fsync_duration_seconds_bucket** 报告 etcd 磁盘 fsync 持续时间，**etcd_server_leader_changes_seen_total** 报告领导更改。要排除一个较慢的磁盘并且确认磁盘的速度合理，**etcd_disk_wal_fsync_duration_seconds_bucket** 的p99 值应该小于 10ms。

fio 是一个 I/O 基准测试工具，可用于在创建 OpenShift 集群之前或之后验证 etcd 的硬件。运行 fio 并分析结果：

假设在接受测试的机器上安装了类似 podman 或 docker 的容器运行时，并且 etcd 写入数据存在于 -/var/lib/etcd，请运行：

流程

如果使用 podman，运行以下命令：

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

另外，如果使用 docker，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 10ms。

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（IO）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])）** 报告 etcd 在成员间复制客户端请求的时间 ; 它应该小于 50 ms。

2.5. 分离 ETCD 数据

在 etcd 历史记录压缩和其他事件后，必须定期执行手动清除碎片以便重新声明磁盘空间。

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。

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 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.us-west-1.compute.internal 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
   ```

2.6. 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
- 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
- OpenShift Container Storage
- Red Hat Advanced Cluster Manager

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。

2.7. 移动监控解决方案

默认情况下，部署包含 Prometheus、Grafana 和 AlertManager 的 Prometheus Cluster Monitoring 堆栈来提供集群监控功能。它由 Cluster Monitoring Operator 进行管理。若要将其组件移到其他机器上，需要创建并应用自定义配置映射。

流程

1. 将以下 **ConfigMap** 定义保存为 **cluster-monitoring-configmap.yaml** 文件：

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

```
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |+
    alertmanagerMain:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    prometheusK8s:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    prometheusOperator:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    grafana:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    k8sPrometheusAdapter:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    kubeStateMetrics:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    telemeterClient:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    openshiftStateMetrics:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
    thanosQuerier:
      nodeSelector:
        node-role.kubernetes.io/infra: ""
```

运行此配置映射会强制将监控堆栈的组件重新部署到基础架构节点。

2. 应用新的配置映射：

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

3. 观察监控 pod 移至新机器：

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

4. 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。

2.8. 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
```

3. 在对象的 **spec** 部分添加以下文本行：

```
  nodeSelector:
    node-role.kubernetes.io/infra: ""
```

4. 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。

2.9. 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
```

在 **spec** 中添加使用 **infra** 标签的 **nodeSelector** 的部分，如下所示：

```
  spec:
    nodePlacement:
```

```
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/infra: ""
```

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   v1.18.3
   ```

   由于角色列表包含 **infra**，因此 Pod 在正确的节点上运行。

2.10. 基础架构节点大小

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	32
100	8	64
250	32	192
500	32	192

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 4.5 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基础架构组件 - Prometheus、Router 和 Registry。日志记录是第二天操作，建议不考虑它。

注意

在 OpenShift Container Platform 4.5 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。

2.11. 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：

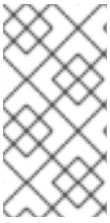
- 将节点分散到所有可用区以获得更高的可用性。
- 同时扩展的机器数量不要超过 25 到 50 个。
- 考虑在每个可用区创建一个具有类似大小的替代实例类型的新机器集，以帮助缓解周期性供应商容量限制。例如，在 AWS 上，使用 m5.large 和 m5d.large。

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
```

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
```

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

您可以使用 **MachineHealthCheck** 资源定义集群中的机器被视为不健康的条件。会自动修复满足条件的机器。

要监控机器健康状况，创建一个 **MachineHealthCheck** 自定义资源（CR），其中包含要监控的机器集合的标签以及要检查的条件，如维持 **NotReady** 状态 15 分钟，或在 node-problem-detector 中显示持久性状况。

监控 **MachineHealthCheck** CR 的控制器会检查您定义的条件。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。

注意

对于具有 master 角色的机器，机器健康检查会报告不健康的节点数量，但不会删除机器。例如：

输出示例

```
$ oc get machinehealthcheck example -n openshift-machine-api
```

```
NAME      MAXUNHEALTHY   EXPECTEDMACHINES   CURRENTHEALTHY
example   40%            3                  1
```

为限制删除机器造成的破坏性影响，控制器一次仅排空并删除一个节点。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，则控制器会停止删除机器，您必须手动进行处理。

要停止检查，请删除自定义资源。

3.3.1. Bare Metal 上的 MachineHealthCheck

在裸机集群上删除机器会触发重新置备裸机主机。通常，裸机重新置备是一个需要较长时间的过程，在这个过程中，集群缺少计算资源，应用程序可能会中断。要将默认补救过程从机器删除到主机的节能周期，请使用 **machine.openshift.io/remediation-strategy: external-baremetal** 注解来注解 MachineHealthCheck 资源。

设置注解后，不健康的机器会使用 BMC 凭证进行节能。

3.3.2. 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

MachineHealthCheck 资源类似以下 YAML 文件之一：

裸机的MachineHealthCheck

```
apiVersion: machine.openshift.io/v1beta1
kind: MachineHealthCheck
metadata:
  name: example 1
  namespace: openshift-machine-api
  annotations:
```

```
    machine.openshift.io/remediation-strategy: external-baremetal (2)
spec:
  selector:
    matchLabels:
      machine.openshift.io/cluster-api-machine-role: <role> (3)
      machine.openshift.io/cluster-api-machine-type: <role> (4)
      machine.openshift.io/cluster-api-machineset: <cluster_name>-<label>-<zone> (5)
  unhealthyConditions:
  - type:    "Ready"
    timeout: "300s" (6)
    status: "False"
  - type:    "Ready"
    timeout: "300s" (7)
    status: "Unknown"
  maxUnhealthy: "40%" (8)
  nodeStartupTimeout: "10m" (9)
```

(1) 指定要部署的机器健康检查的名称。

(2) 对于裸机集群，您必须在 **annotations** 部分中包含 **machine.openshift.io/remediation-strategy: external-baremetal** 注解来启用电源周期补救。采用这种补救策略时，不健康的主机会被重启，而不是从集群中删除。

(3)(4) 为要检查的机器池指定一个标签。

(5) 以 **<cluster_name>-<label>-<zone>** 格式 指定要跟踪的机器集。例如， **prod-node-us-east-1a**。

(6)(7) 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。

(8) 指定目标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。

(9) 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。

所有其他安装类型的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
  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 指定目标池中允许的不健康机器的数量。这可设为一个百分比或一个整数。

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。

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。

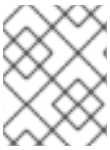
3.4.1.2. 使用百分比设置 maxUnhealthy

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

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
```

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:** 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)
```

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**。

<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，并在与机器配置池的节点选择器匹配的所有节点上设置配置集 **<tuned_profile_name>**。

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

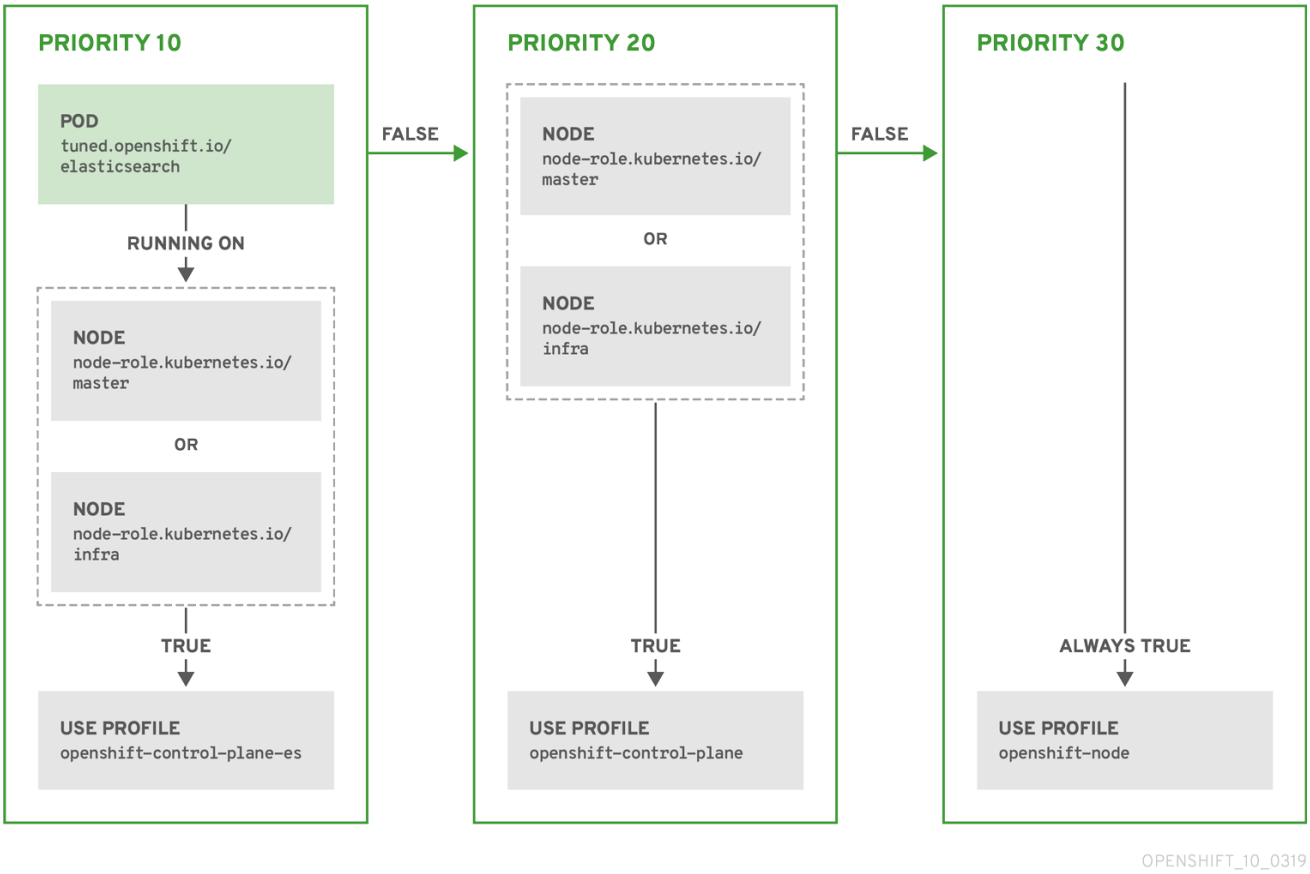
```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra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

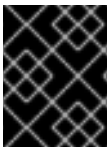
为尽量减少节点的重新引导情况，为目标节点添加机器配置池将匹配的节点选择器标签，然后创建上述 Tuned CR，最后创建自定义机器配置池。

4.6. 自定义调整示例

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。作为管理员，使用以下命令创建一个自定义 Tuned CR。

自定义调整示例

```
$ 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-node-label
    priority: 10
    profile: openshift-ingress
_EOF_
```



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。

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：

```
$ podman pull quay.io/openshift/origin-tests:4.5
```

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:4.5 /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:4.5 \
/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，您必须启用 **LatencySensitive** Feature Gate，并在 **cpumanager-enabled** 子定义资源（CR）中配置拓扑管理器策略。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

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. 编辑 **FeatureGate** 对象以添加 **LatencySensitive** 功能集：

```
$ oc edit featuregate/cluster
```

```
apiVersion: config.openshift.io/v1
kind: FeatureGate
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: 2020-06-05T14:41:09Z
  generation: 2
  managedFields:
  - apiVersion: config.openshift.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .: {}
          f:release.openshift.io/create-only: {}
      f:spec: {}
    manager: cluster-version-operator
    operation: Update
    time: 2020-06-05T14:41:09Z
  - apiVersion: config.openshift.io/v1
    fieldsType: FieldsV1
    fieldsV1:
      f:spec:
        f:featureSet: {}
    manager: oc
    operation: Update
    time: 2020-06-05T15:21:44Z
  name: cluster
  resourceVersion: "28457"
  selfLink: /apis/config.openshift.io/v1/featuregates/cluster
  uid: e802e840-89ee-4137-a7e5-ca15fd2806f8
spec:
  featureSet: LatencySensitive (1)
...
```

(1) 添加以逗号分隔的列表中设定的 **LatencySensitive** 功能。

2. 在 **cpumanager-enabled** 自定义资源（CR）中配置的拓扑管理器。

```
$ 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 指定所选的拓扑管理器策略。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

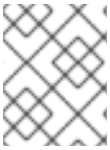
以下 Prometheus 存储要求并不具有规定性。取决于工作负载活动和资源使用情况，集群中可能会观察到更高资源消耗。

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

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. 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
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 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 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	4.1 测试的最大值	4.2 测试的最大值	4.3 测试的最大值	4.4 测试的最大值	4.5 测试的最大值
节点数	2,000	2,000	2,000	250	500
pod 数量 [1]	150,000	150,000	150,000	62,500	62,500
每个节点的 pod 数量	250	250	500	500	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。	没有默认值。
命名空间数量 [2]	10,000	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）- Source-to-Image (S2I) 构建策略	10,000（默认 pod RAM 512 Mi）- Source-to-Image (S2I) 构建策略	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	25,000
服务数 [4]	10,000	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	5,000
每个服务中的后端数	5,000	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	2,000

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. 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。

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	3/25/250/500 [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。

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

如果您计划把集群的规模限制在有 2200 个 pod，则需要至少有 5 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
```

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。

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：

```
 ---
   Kind: Template
   apiVersion: v1
   metadata:
     name: deploymentConfigTemplate
     creationTimestamp:
     annotations:
       description: This template will create a deploymentConfig with 1 replica, 4 env vars and a
service.
       tags: ''
   objects:
   - kind: DeploymentConfig
     apiVersion: v1
     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
- kind: Service
  apiVersion: v1
  metadata:
    name: service${IDENTIFIER}
  spec:
    selector:
      name: replicationcontroller${IDENTIFIER}
    ports:
    - name: serviceport${IDENTIFIER}
      protocol: TCP
      port: 80
      targetPort: 8080
    portalIP: ''
    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: deploymentConfigTemplate
```

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。

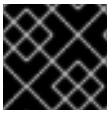
10.1. 可用的持久性存储选项

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 [1],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

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



重要

目前，OpenShift Container Platform 4.5 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**

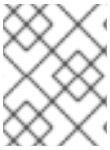
[3] **Prometheus 是用于指标数据的底层技术。**

[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持久性卷声明（PVC）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。**

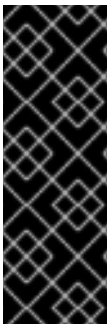
[7] **对象存储不会通过 OpenShift Container Platform 的 PV 或 PVC 使用。应用程序必须与对象存储 REST API 集成。**



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。

10.2.1. 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。
- 对象存储应该兼容 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。

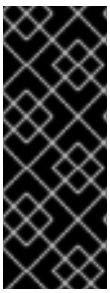
重要

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. 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。

11.2. 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 12 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、网络接口卡（NIC）offload、多队列和 ethtool 设置来调整。

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。

12.1. 为您的网络优化 MTU

有两个重要最大传输单元（MTU）：网卡（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。

12.2. 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。

12.3. 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 13 章 巨页的作用及应用程序如何使用它们

13.1. 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。

13.2. 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

13.3. 配置巨页

节点必须预先分配在 OpenShift Container Platform 集群中使用的巨页。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。

13.3.1. 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** 配置集

```
$ 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。