OpenShift Container Platform 3.11

扩展和性能指南

OpenShift Container Platform 3.11 扩展和性能指南

Last Updated: 2023-03-25

OpenShift Container Platform 3.11 扩展和性能指南

OpenShift Container Platform 3.11 扩展和性能指南

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律通告

摘要

扩展集群并调整生产环境中的性能

目录

第 1 章 概述 4

第 2 章 推荐的安装实践 5
2.1. 预安装依赖项 5
2.2. ANSIBLE 安装优化 5
2.3. 网络注意事项 6

第 3 章 推荐的主机实践 7
3.1. OPENSHIFT CONTAINER PLATFORM MASTER 主机的推荐做法 7
3.2. OPENSHIFT CONTAINER PLATFORM 节点主机的推荐做法 7
3.3. OPENSHIFT CONTAINER PLATFORM ETCD 主机的建议实践 9
3.3.1. 通过 OpenStack 使用 PCI 透传向 etcd 节点提供存储 13
3.4. 使用 TUNED 配置集扩展主机 14

第 4 章 优化计算资源 16
4.1. 过量使用 16
4.2. 镜像注意事项 16
4.2.1. 使用预部署的镜像提高效率 16
4.2.2. 预拉取镜像 16
4.3. 使用 RHEL 工具容器镜像进行调试 17
4.4. 使用基于 ANSIBLE 的健康检查进行调试 17

第 5 章 优化持久性存储 18
5.1. 概述 18
5.2. 常规存储指南 18
5.3. 存储建议 19
5.3.1. 特定应用程序存储建议 19
5.3.1.1. Registry 20
5.3.1.2. 扩展的 registry 20
5.3.1.3. 监控 20
5.3.1.4. 日志记录 20
5.3.1.5. 应用程序 21
5.3.2. 其他特定的应用程序存储建议 21
5.4. 选择图形驱动程序 21
5.4.1. 在 SELinux 中使用 OverlayFS 或 DeviceMapper 的好处 25
5.4.2. 比较 Overlay 和 Overlay2 图形驱动程序 25

第 6 章 优化临时存储 26
6.1. 概述 26
6.2. 常规存储指南 26

第 7 章 网络优化 28
7.1. 优化网络性能 28
7.1.1. 为您的网络优化 MTU 28
7.2. 配置网络子网 29
7.3. 优化 IPSEC 29

第 8 章 路由优化 30
8.1. 扩展 OPENSHIFT CONTAINER PLATFORM HAPROXY 路由器 30
8.1.1. 基准性能 30
8.1.2. 性能优化 31
8.1.2.1. 设置连接的最大数量 31
8.1.2.2. CPU 和中断关联性 31

8.1.2.3. 增加线程数 32
8.1.2.4. 增加缓冲的影响 32
8.1.2.5. HAProxy Reloads 的优化 32

第 9 章 扩展集群指标 33
9.1. 概述 33
9.2. 针对 OPENSHIFT CONTAINER PLATFORM 的建议 33
9.3. 集群指标的容量规划 33
9.4. 扩展 OPENSHIFT CONTAINER PLATFORM 指标 POD 34
9.4.1. 先决条件 34
9.4.2. 扩展 Cassandra 组件 34

第 10 章 扩展 CLUSTER MONITORING OPERATOR 36
10.1. 概述 36
10.2. 针对 OPENSHIFT CONTAINER PLATFORM 的建议 36
10.3. CLUSTER MONITORING OPERATOR 的容量规划 36
10.3.1. 实验室环境 37
10.3.2. 先决条件 37

第 11 章 测试每个集群的最大值 38
11.1. OPENSHIFT CONTAINER PLATFORM 为主版本测试的集群最大限制 38
11.2. OPENSHIFT CONTAINER PLATFORM 测试的集群最大限制 39
11.2.1. 路由最大限制 39
11.3. 测试 OPENSHIFT CONTAINER PLATFORM 集群最大值的环境和配置 40
11.4. 规划环境以延长集群最大值 40
11.5. 规划环境符合应用程序要求 41

第 12 章 使用 CLUSTER LOADER 42
12.1. CLUSTER LOADER 执行的操作 42
12.2. 安装 CLUSTER LOADER 42
12.3. 运行 CLUSTER LOADER 42
12.4. 配置 CLUSTER LOADER 42
12.4.1. 配置字段 42
12.4.2. Cluster Loader 配置文件示例 45
12.5. 已知问题 46

第 13 章 使用 CPU MANAGER 47
13.1. CPU MANAGER 的作用 47
13.2. 设置 CPU MANAGER 47

第 14 章 管理大页面 51
14.1. 巨页的作用 51
14.2. 先决条件 51
14.3. 消耗大页面 51

第 15 章 在 GLUSTERFS 存储上进行优化 53
15.1. 数据库聚合模式指南 53
15.2. 测试的应用程序 53
15.3. 支持列表 53
15.4. 测试结果 54

第 1 章 概述

本指南提供了如何增强 OpenShift Container Platform 集群性能和 OpenShift Container Platform 产品堆栈不同级别的扩展的步骤和示例。它包括构建、扩展和调整 OpenShift Container Platform 集群的建议实践。

调优注意事项可能会因集群设置而异，并建议本指南中的任何性能建议都有利弊。

第 2 章 推荐的安装实践

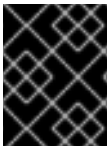
2.1. 预安装依赖项

节点主机将访问网络来安装任何 RPM 依赖项，如 **atomic-openshift-***、**iptables** 和 **CRI-O** 或 **Docker**。预安装这些依赖关系，创建更高效的安装，因为仅在需要时访问 RPM，而不是在安装过程中每个主机执行多次。

对于无法访问 registry 以进行安全目的的计算机也很有用。

2.2. ANSIBLE 安装优化

OpenShift Container Platform 安装方法使用 Ansible。Ansible 对于运行并行操作非常有用，这意味着快速高效的安装。但是，这些可通过额外的调整选项加以改进。如需可用 Ansible 配置选项列表，请参阅 Configuring Ansible 部分。



重要

并行行为可能会认为内容源，如您的镜像 registry 或 Red Hat Satellite 服务器。准备服务器的基础架构 pod 和操作系统补丁可帮助防止出现这个问题。

从最低延迟控制节点（LAN 速度）运行安装程序。不建议在广域网络(WAN)上运行，因此不会因为丢失的网络连接运行安装。

Ansible 为 性能和扩展提供了自己的指导，包括使用 RHEL 6.6 或更高版本来确保 OpenSSH 支持 **ControlPersist** 的版本，并从与集群相同的 LAN 中运行安装程序，*但不*从集群中的机器运行。

以下是用于大型集群安装和管理的 Ansible 示例，其中包含了 Ansible 记录的推荐：

```
# cat /etc/ansible/ansible.cfg
```

输出示例

```
# config file for ansible -- http://ansible.com/
# ==============================================
[defaults]
forks = 20 1
host_key_checking = False
remote_user = root
roles_path = roles/
gathering = smart
fact_caching = jsonfile
fact_caching_connection = $HOME/ansible/facts
fact_caching_timeout = 600
log_path = $HOME/ansible.log
nocows = 1
callback_whitelist = profile_tasks

[privilege_escalation]
become = False

[ssh_connection]
ssh_args = -o ControlMaster=auto -o ControlPersist=600s -o ServerAliveInterval=60
```

```
control_path = %(directory)s/%%h-%%r
pipelining = True 2
timeout = 10
```

1 20 个分叉非常理想，因为大型分叉可能会导致安装失败。

2 pipelining 可减少控制和目标节点之间的连接数量，有助于提高安装程序的性能。

2.3. 网络注意事项

在安装后可能会更改网络子网，但难度比较困难。安装前可以更容易地考虑网络子网大小，因为尽可能提高大小可能会给增长的集群造成问题。

有关推荐的网络子网实践，请参阅网络优化主题。

第 3 章 推荐的主机实践

3.1. OPENSHIFT CONTAINER PLATFORM MASTER 主机的推荐做法

除了 pod 流量外，OpenShift Container Platform 基础架构中最常用的数据路径也介于 OpenShift Container Platform master 主机和 etcd 之间。OpenShift Container Platform API 服务器（master 二进制的一部分）在节点状态、网络配置、secret 等方面咨询 etcd。

通过以下方法优化此流量路径：

- 在 master 主机上运行 etcd。默认情况下，etcd 在所有 master 主机上的静态 pod 中运行。
- 确保 master 主机之间有一个不协调、低延迟 LAN 通信链接。

OpenShift Container Platform master 会积极缓存资源的反序列化版本，以简化 CPU 负载。但是，如果较小的 pod 集群小于 1000 个 pod，这个缓存可能会浪费大量内存用于微小的 CPU 负载。默认缓存大小为 50,000 个条目，它根据资源的大小，可以将 cupy 1 增加到 2 GB 内存。使用 ***/etc/origin/master/master-config.yaml*** 中的以下设置可以减少这个缓存大小：

```
kubernetesMasterConfig:
  apiServerArguments:
    deserialization-cache-size:
    - "1000"
```

发送到 API 服务器的客户端请求或 API 调用数量由每秒的 Queries(QPS)值和 API 服务器处理的并发请求数决定。客户端可能会过量 QPS 速率发出的请求数量取决于突发值，这对具有极限的应用程序来说非常有用，并可执行不监管的请求数量。当 API 服务器处理大量并发请求时，对请求的响应时间，特别是大型和/或高密度的集群。建议您监控 Prometheus 中的 **apiserver_request_count** 速率指标，并相应地调整 **maxRequestsInFlight** 和 **QPS**。

更改默认值时，需要有一个很好的平衡，因为 API 服务器的 CPU 和内存消耗，etcd IOPS 会在并行处理更多请求时增加。另请注意，大量非watch 请求可能会在固定 60 秒超时后取消 API 服务器过载，客户端开始重试。

API 服务器系统中提供了足够的 CPU 和内存资源，API 服务器请求过载问题可安全地缓解这个问题。通过考虑以上提到的因素并浏览了 maxRequestsInFlight、API qps 和 burst 值 *_/etc/origin/master/master-config.yaml

```
masterClients:
  openshiftLoopbackClientConnectionOverrides:
    burst: 600
    qps: 300
servingInfo:
  maxRequestsInFlight: 500
```

注意

以上 maxRequestsInFlight、qps 和 burst 值是 OpenShift Container Platform 的默认值。如果请求的用时小于秒，则 qps 可以大于 maxRequestsInFlight 值。如果 'maxRequestsInFlight' 设为零，则服务器可以处理的并发请求数没有限制。

3.2. OPENSHIFT CONTAINER PLATFORM 节点主机的推荐做法

OpenShift Container Platform 节点配置文件包含重要的选项，如 iptables 同步周期、SDN 网络的最大传输单元(MTU)和 proxy-mode。要配置节点，请修改适当的节点配置映射。

注意

不要直接编辑 **node-config.yaml** 文件。

节点配置文件允许您将参数传递给 kubelet（节点）过程。您可以通过运行 **kubelet --help** 来查看可能的选项列表。

注意

并非所有 kubelet 选项都由 OpenShift Container Platform 支持，并在上游 Kubernetes 中使用。这意味着特定的选项处于有限的支持。

注意

如需了解每个 OpenShift Container Platform 版本的最大支持限制，请参阅集群最大限制。

在 */etc/origin/node/node-config.yaml* 文件中，两个参数控制可以调度到节点的 pod 的最大数量：**pod-per-core** 和 **max-pods**。当两个选项都被使用时，这两个选项中的较小的限制为节点上的 pod 数量。超过这些值可导致：

- OpenShift Container Platform 和 Docker 的 CPU 使用率增加。
- 减慢 pod 调度的速度。
- 潜在的内存不足情况（取决于节点中的内存量）。
- 耗尽 IP 地址池。
- 资源过量使用，导致用户应用程序性能变差。

注意

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。

pods-per-core 根据节点上的处理器内核数来设置节点可运行的 pod 数量。例如，如果将一个有 4 个处理器内核的节点上的 **pod-per-core** 设置为 **10**，则该节点上允许的最大 pod 数量为 40。

```
kubeletArguments:
  pods-per-core:
    - "10"
```

注意

将 **pods-per-core** 设置为 0 可禁用这个限制。

max-pods 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。集群限制 记录 **max-pods** 的最大支持值。

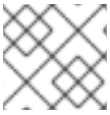
```
kubeletArguments:
  max-pods:
    - "250"
```

使用上例时，**pod-per-core** 的默认值为 **10**，**max-pods** 的默认值为 **250**。这意味着，除非节点有 25 个或更多内核，否则 **pod-per-core** 默认是限制因素。

如需了解 OpenShift Container Platform 集群的建议限制，请参阅安装文档中的 大小注意事项 部分。OpenShift Container Platform 和容器引擎协调容器状态更新的建议大小帐户。这种协调会对主机和容器引擎进程造成 CPU 压力，这些压力可包括编写大量日志数据。

kubelet 与 API 服务器进行交互的频率取决于 qps 和 burst 值。如果每个节点上运行的 pod 有限，默认值就足够了。如果节点上有足够 CPU 和内存资源，可以在 */etc/origin/node/node-config.yaml* 文件中调整 qps 和 burst 值：

```
kubeletArguments:
  kube-api-qps:
  - "20"
  kube-api-burst:
  - "40"
```



注意

以上 qps 和 burst 值是 OpenShift Container Platform 的默认值。

3.3. OPENSHIFT CONTAINER PLATFORM ETCD 主机的建议实践

etcd 是一个分布式键值存储，OpenShift Container Platform 用于配置它。

OpenShift Container Platform 版本	etcd 版本	存储 schema 版本
3.3 及更早版本	2.x	v2
3.4 和 3.5	3.x	v2
3.6	3.x	v2（升级）
3.6	3.x	v3（新安装）
3.7 及更新的版本	3.x	v3

etcd 3.x 引入了重要的可伸缩性和性能改进，用于减少任意大小集群的 CPU、内存、网络和磁盘要求。etcd 3.x 还会实施后向兼容的存储 API，促进磁盘 etcd 数据库的两个步骤迁移。出于迁移目的，OpenShift Container Platform 3.5 中 etcd 3.x 使用的存储模式保留在 v2 模式中。自 OpenShift Container Platform 3.6 起，新安装使用存储模式 v3。从以前的 OpenShift Container Platform 版本升级 *不会* 自动将数据从 v2 迁移到 v3。您必须使用提供的 playbook，并遵循记录的流程来迁移数据。

etcd 版本 3 实现了向后兼容的存储 API，有助于对磁盘 etcd 数据库进行两步迁移。出于迁移目的，OpenShift Container Platform 3.5 中 etcd 3.x 使用的存储模式保留在 v2 模式中。自 OpenShift Container Platform 3.6 起，新安装使用存储模式 v3。作为升级到 OpenShift Container Platform 3.7 的

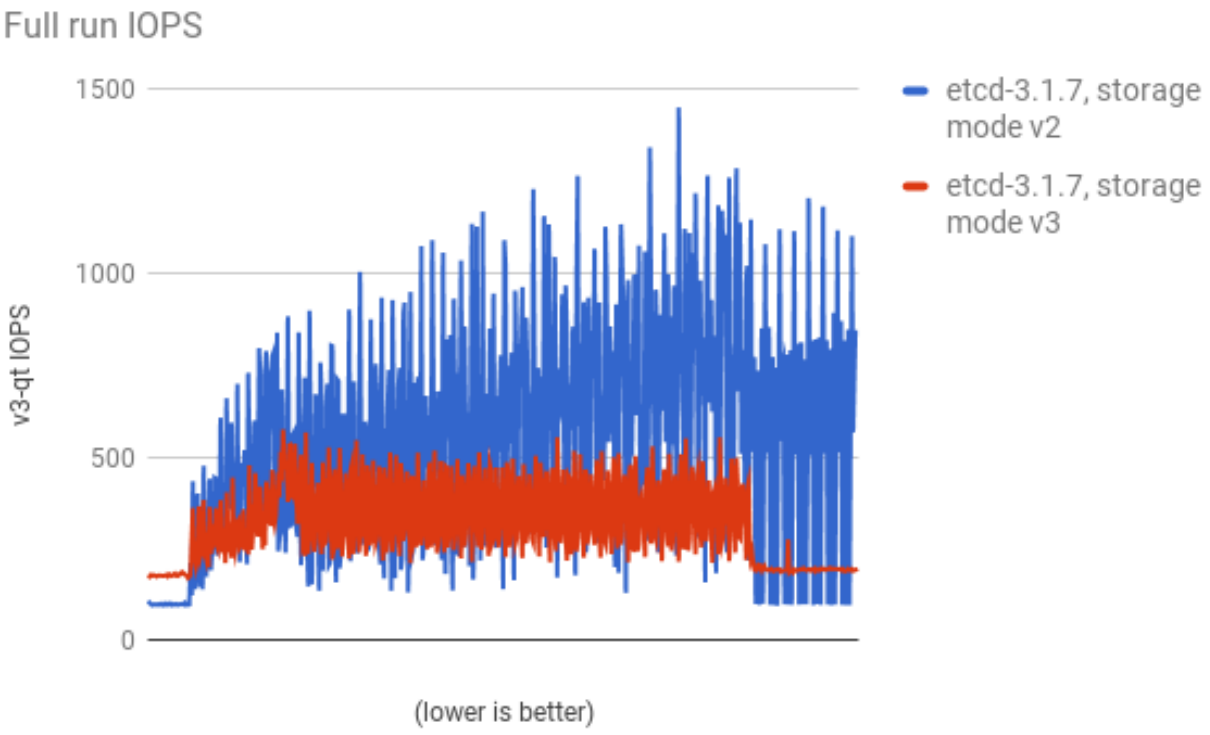
过程的一部分，如果需要，您需要将 etcd 存储 API 升级到 v3。在版本 3.7 及更新的版本中，必须使用 v3 API。

除了更改 v3 的新安装模式外，OpenShift Container Platform 3.6 也会开始对所有 OpenShift Container Platform 类型进行 enforcing *仲裁读取*。这是为了确保对 etcd 的查询不会返回过时的数据。在单节点 etcd 集群中，过时的数据不是问题。在高可用的 etcd 部署中，通常在生产集群中找到，仲裁读取确保有效的查询结果。在数据库术语中，仲裁读是 *线性的* - 每个客户端都会看到集群的最新更新状态，所有客户端会看到相同的读写序列。有关性能改进的更多信息，请参阅 etcd 3.1 公告。

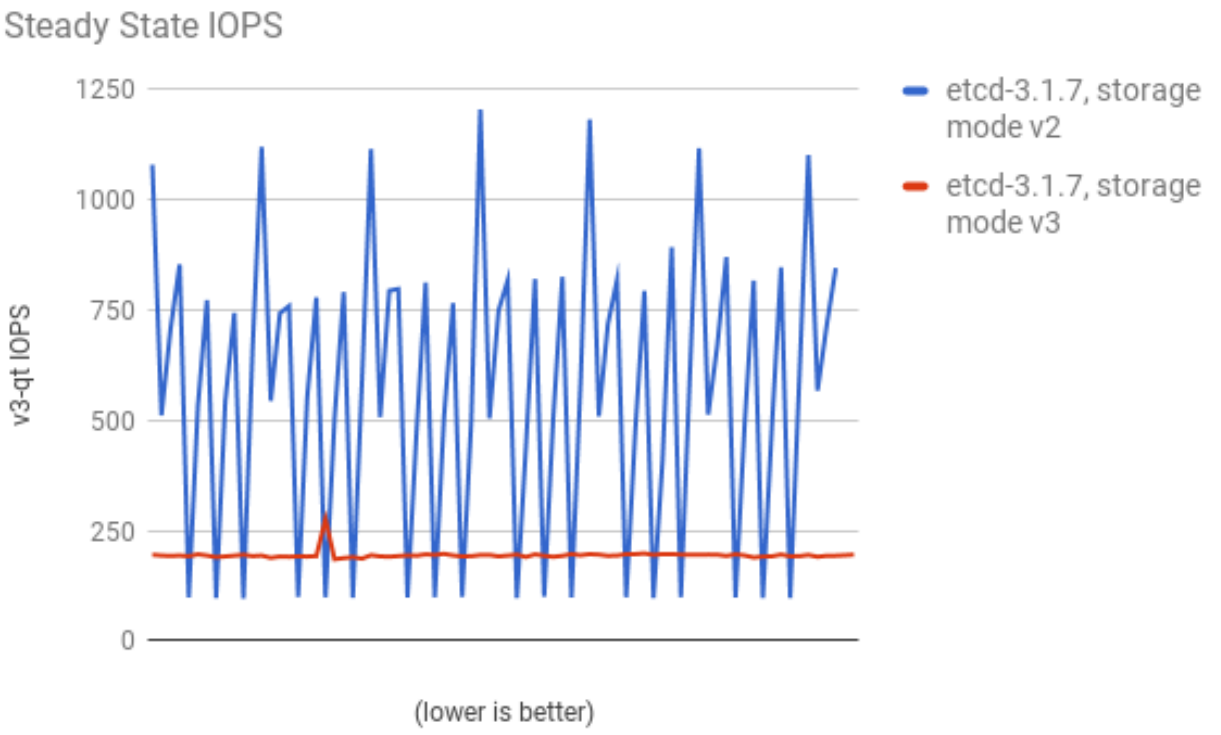
请注意，OpenShift Container Platform 使用 etcd 来存储 Kubernetes 本身所需的其他信息。例如，OpenShift Container Platform 在 etcd 中存储有关镜像、构建和其他组件的信息，如 OpenShift Container Platform 在 Kubernetes 之上添加的功能需要。最后，这意味着 etcd 主机性能和大小调整的信息将与 Kubernetes 和其他建议有所不同。红帽使用 OpenShift Container Platform 用例和参数测试 etcd 可扩展性和性能，以生成最准确的建议。

使用 cluster-loader 实用程序使用 300 个节点 OpenShift Container Platform 3.6 集群来量化性能。etcd 3.x（存储模式 v2）与 etcd 3.x（存储模式 v3）的比较，会在下面的图表中识别出清晰的信息。

在负载下显著降低存储 IOPS：

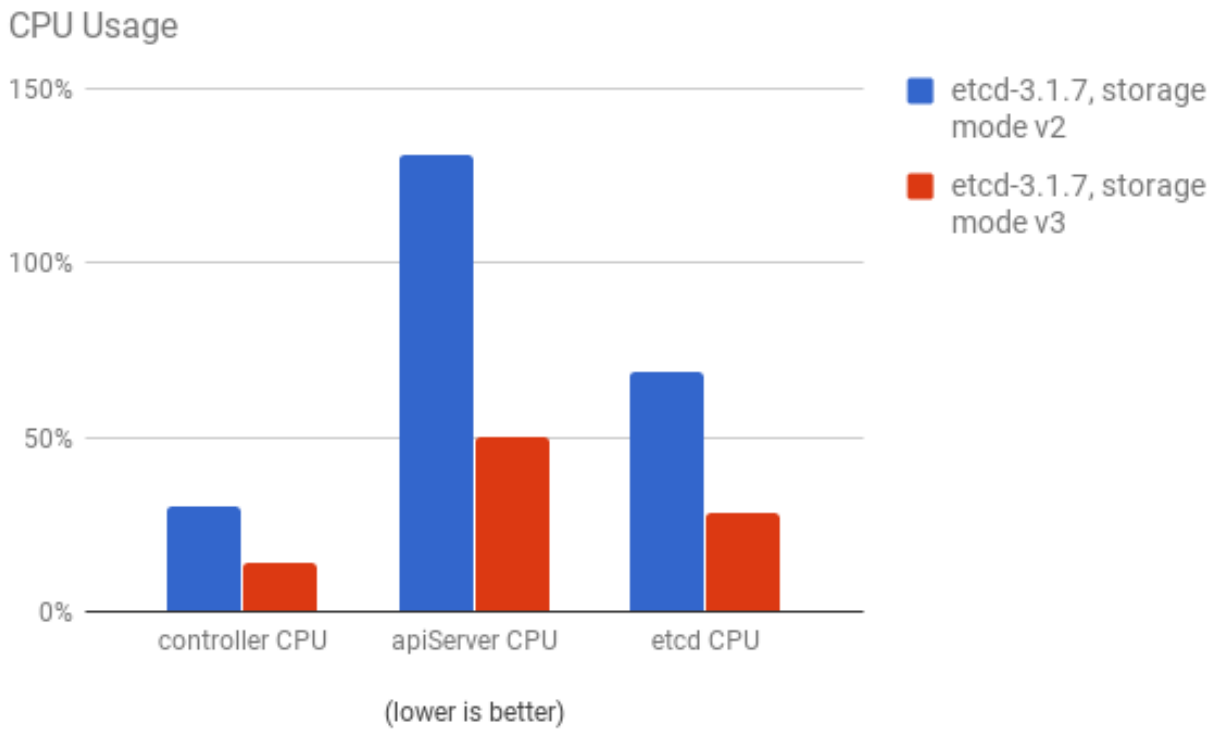


稳定状态的存储 IOPS 也显著降低：

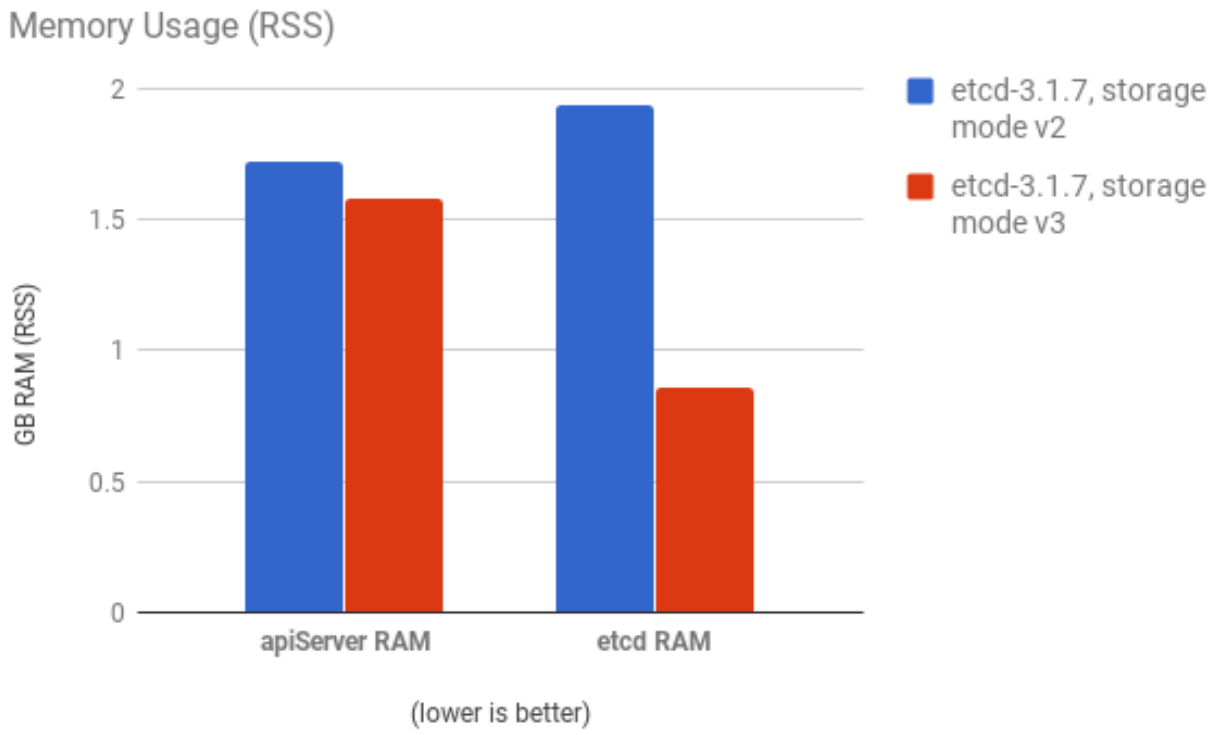


查看相同的 I/O 数据，在两个模式中绘制平均 IOPS：

API 服务器(master)和 etcd 进程的 CPU 使用率会减少：



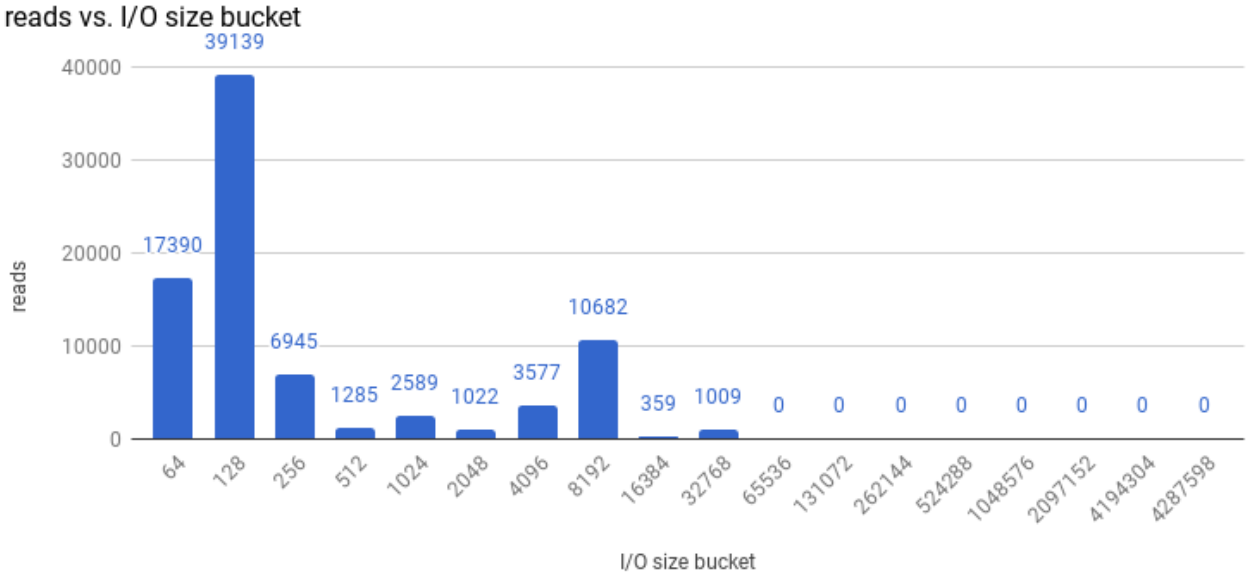
API 服务器(master)和 etcd 进程的内存使用率也会减少：



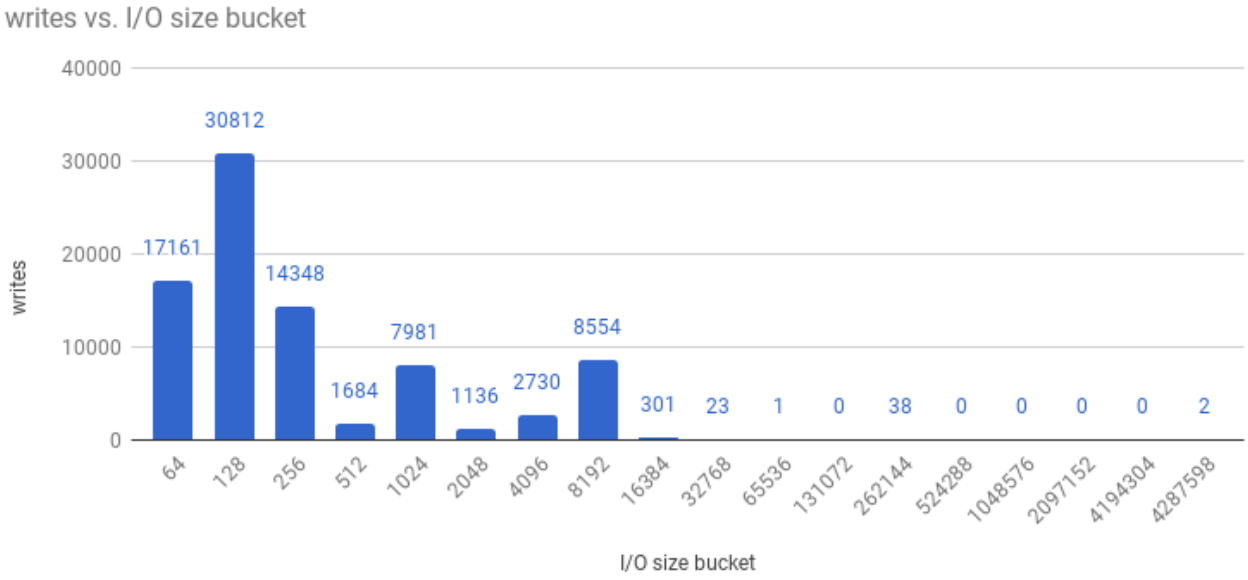
重要

在 OpenShift Container Platform 下分析 etcd 后，etcd 通常会执行少量存储输入和输出。强烈建议您使用带有可快速处理较小读写操作的存储的 etcd。

查看 etcd 3.1 的 3 节点集群（使用存储 v3 模式和仲裁读取强制）执行的大小 I/O 操作，如下所示：



和写入：



注意

etcd 进程通常是内存密集型。Master/API 服务器进程是 CPU 密集型。这使得他们在一台虚拟机或虚拟机(VM)内合理地定位对。通过在同一个主机上定位或提供专用网络，以此优化 etcd 和 master 主机之间的通信。

3.3.1. 通过 OpenStack 使用 PCI 透传向 etcd 节点提供存储

要为 etcd 节点提供快速存储以便 etcd 在大规模稳定，使用 PCI 透传将非易失性内存表达(NVMe)设备直接传递给 etcd 节点。要使用 Red Hat OpenStack 11 或更高版本进行设置，请在存在 PCI 设备的 OpenStack 节点上完成以下内容。

1. 确定在 BIOS 中启用了 Intel Vt-x。
2. 启用 input-output 内存管理单元(IOMMU)。在 */etc/sysconfig/grub* 文件中，将 **intel_iommu=on iommu=pt** 添加到 **GRUB_CMDLINX_LINUX** 行的末尾（在括号内）。

3. 运行以下命令重新生成 */etc/grub2.cfg* ：

```
$ grub2-mkconfig -o /etc/grub2.cfg
```

4. 重启系统：

5. 在 */etc/nova.conf* 中的控制器上：

```
[filter_scheduler]

enabled_filters=RetryFilter,AvailabilityZoneFilter,RamFilter,DiskFilter,ComputeFilter,ComputeC
apabilitiesFilter,ImagePropertiesFilter,ServerGroupAntiAffinityFilter,ServerGroupAffinityFilter,Pci
PassthroughFilter

available_filters=nova.scheduler.filters.all_filters

[pci]

alias = { "vendor_id":"144d", "product_id":"a820",
"device_type":"type-PCI", "name":"nvme" }
```

6. 在控制器上重启 **nova-api** 和 **nova-scheduler**。

7. 在 */etc/nova/nova.conf* 中的计算节点上：

```
[pci]

passthrough_whitelist = { "address": "0000:06:00.0" }

alias = { "vendor_id":"144d", "product_id":"a820",
"device_type":"type-PCI", "name":"nvme" }
```

要检索您想要透传的 NVMe 设备的 **address**、**vendor_id** 和 **product_id** 值，请运行：

```
# lspci -nn | grep devicename
```

8. 重启计算节点上的 **nova-compute**。

9. 将您正在运行的 OpenStack 版本配置为使用 NVMe 并启动 etcd 节点。

3.4. 使用 TUNED 配置集扩展主机

tuned 是 Red Hat Enterprise Linux(RHEL)和其他红帽产品默认启用的调优配置文件交付机制。 **tuned** 自定义 Linux 设置，如 sysctl、电源管理和内核命令行参数，以针对不同的工作负载性能和可扩展性要求优化操作系统。

OpenShift Container Platform 利用 **tuned** 守护进程，包括名为 **openshift**、**openshift-node** 和 **openshift-control-plane** 的 **Tuned** 配置集。这些配置集可以安全地增加内核中存在的一些垂直扩展限制，并在安装过程中自动应用到您的系统。

Tuned 配置集支持在配置集间继承。它们也支持一个自动扩展功能，它根据虚拟环境中是否使用了配置集来选择父配置集。**openshift** 配置集使用这些功能，它是 **openshift-node** 和 **openshift-control-plane** 配置集的父项。它包含与 OpenShift Container Platform 应用程序节点和 control plane 节点相关的调

整。**openshift-node** 和 **openshift-control-plane** 配置集分别在 application 和 control plane 节点上设置。

将 **openshift** 配置集用作父级的配置集层次结构可确保传送到 OpenShift Container Platform 系统的调优是裸机主机的 **throughput-performance**（RHEL 默认）和适用于 RHEL Atomic 主机节点的 **virtual-guest**，以及适用于 RHEL Atomic Host 节点的 **atomic-guest**。

要查看在您的系统中启用了哪个 **Tuned** 配置集，请运行：

```
# tuned-adm active
```

输出示例

```
Current active profile: openshift-node
```

有关 **Tuned** 的详情，请查看 Red Hat Enterprise Linux 性能调优指南 。

第 4 章 优化计算资源

4.1. 过量使用

您可以使用过量使用过程，以便 CPU 和内存等资源可以被需要它们的集群部分访问。

重要

为了避免因为虚拟机监控程序和 Kubernetes 之间的调度冲突而出现错误集群行为，请不要在虚拟机监控程序级别上过量使用。

请注意，当您过量使用时，另一个应用程序可能无权访问需要的资源，从而降低了性能。但是，这可能是因为密度增加和降低成本而是一个可接受的权衡。例如，开发、质量保证(QA)或测试环境可能被过量使用，而生产环境可能并非如此。

OpenShift Container Platform 通过计算资源模型和配额系统实施资源管理。如需有关 OpenShift 资源模型 的更多信息，请参阅文档。

有关过量使用的更多信息和策略，请参阅集群管理指南中的过量使用文档。

4.2. 镜像注意事项

4.2.1. 使用预部署的镜像提高效率

您可以使用内置的多个任务创建基本的 OpenShift Container Platform 镜像，以提高效率、维护所有节点主机上的配置一致性，并减少重复性任务。这称为预先部署的镜像。

例如，因为每个节点都需要 **ose-pod** 镜像来运行 pod，每个节点都必须定期连接容器镜像 registry 以拉取最新的镜像。当您有 100 个节点同时尝试此操作时，这可能会导致镜像 registry 上的资源争用、浪费网络带宽并增加 pod 启动时间，这可能会出现问题。

构建预部署的镜像：

- 创建所需类型和大小的实例。
- 确保专用的存储设备可用于 CRI-O 或 Docker 本地镜像或容器存储，并独立于容器的任何持久性卷。
- 完全更新系统，并确保安装了 CRI-O 或 Docker。
- 确保主机有权访问所有 yum 存储库。
- 设置精简配置的 LVM 存储。
- 将常用的镜像（如 **rhel7** 基础镜像）和 OpenShift Container Platform 基础架构容器镜像（**ose-pod**、**ose-deployer** 等）预部署前镜像。

确保为任何适当的集群配置配置了预部署的镜像，如能够在 OpenStack 或 AWS 上运行，以及任何其他集群配置。

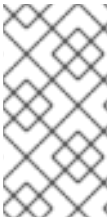
4.2.2. 预拉取镜像

为了有效地生成镜像，您可以将任何必要的容器镜像拉取到所有节点主机。这意味着，镜像不需要初始拉取，这会降低连接速度和性能，特别是用于镜像（如 S2I、指标和日志记录）的时间和性能，这可能会非常大。

对于无法访问 registry 以进行安全目的的计算机也很有用。

另外，您可以使用本地镜像而不是指定 registry 的默认镜像。要做到这一点：

1. 通过将 pod 配置的 **imagePullPolicy** 参数设置为 **IfNotPresent** 或 **Never**，从本地镜像拉取。
2. 确保集群中的所有节点都在本地保存了相同的镜像。



注意

如果可以控制节点配置，则从本地 registry 拉取(pull)适合使用。但是，它不会在不自动替换节点的云供应商（如 GCE）上可靠工作。如果您在 Google Container Engine(GKE)上运行，则每个带有 Google Container Registry 凭证的 ***.dockercfg*** 文件都会有一个 .dockercfg 文件。

4.3. 使用 RHEL 工具容器镜像进行调试

红帽分发 **rhel-tools** 容器镜像，打包工具有助于调试扩展或性能问题。此容器镜像：

- 通过将软件包从基本分发中移到基础分发中，并集成到这个支持容器中，允许用户部署最少的占用空间容器主机。
- 为 Red Hat Enterprise Linux 7 Atomic Host 提供调试功能，它具有不可变的软件包树。**rhel-tools** 包括 tcpdump、sosreport、git、GDB、perf 等实用程序，以及更常见的系统管理实用程序。

使用以下命令使用 **rhel-tools** 容器：

```
# atomic run rhel7/rhel-tools
```

如需更多信息，请参阅 RHEL 工具容器文档。

4.4. 使用基于 ANSIBLE 的健康检查进行调试

通过用于安装和管理 OpenShift Container Platform 集群的 基于 Ansible 的工具提供了额外的诊断健康检查。它们可以为当前 OpenShift Container Platform 安装报告常见部署问题。

这些检查可以使用 **ansible-playbook** 命令（集群安装中使用的相同方法）或作为 **openshift-ansible** 的容器化版本来运行。对于 **ansible-playbook** 方法，检查由 **openshift-ansible** RPM 软件包提供。对于容器化方法，**openshift3/ose-ansible** 容器镜像通过 Red Hat Container Registry 分发。

如需有关可用健康检查和示例用法的信息，请参阅集群管理指南中的 基于 Ansible 的健康检查。

第 5 章 优化持久性存储

5.1. 概述

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。

注意

本指南主要侧重于优化持久性存储。pod 生命周期中使用的数据的本地临时存储会减少选项。只有在启用了临时存储技术预览功能时，才会使用临时存储。此功能默认为禁用。如需更多信息，请参阅为临时存储配置。

5.2. 常规存储指南

下表列出了 OpenShift Container Platform 可用的持久性存储技术。

表 5.1. 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	聚合模式/独立于模式 GlusterFS [1]、iSCSI、光纤通道、Ceph RBD、OpenStack Cinder、AWS EBS [1]、Dell/EMC Scale.IO、VMware vSphere 卷、GCE Persistent Disk [1]、Azure 磁盘
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。	聚合模式/独立于模式 GlusterFS [1]，RHEL NFS, NetApp NFS [2], Azure File, Vendor NFS, Vendor GlusterFS [3], Azure File, AWS EFS
Object	• 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。	聚合模式/独立于模式 GlusterFS [1]、Ceph Object Storage（RADOS 网关）、OpenStack Swift、Ayun OSS、AWS S3、Google Cloud Storage、Azure Blob 存储、Vendor S3 [3], Vendor Swift [3]

1. 聚合模式/独立于模式 GlusterFS、Ceph RBD、OpenStack Cinder、AWS EBS、Azure Disk、GCE 持久磁盘和 VMware vSphere 支持在 OpenShift Container Platform 中原生动态持久性卷 (PV)置备。

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

3. 供应商 GlusterFS、供应商 S3 和供应商 Swift 支持性和可配置性可能有所不同。

您可以使用聚合模式 GlusterFS（超融合或集群托管存储解决方案）或 GlusterFS（外部托管的存储解决方案）用于 OpenShift Container Platform registry、日志记录和监控的对象存储。

5.3. 存储建议

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 5.2. 推荐的、可配置的存储技术

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

1. ReadWriteOnce

2. ReadOnlyMany

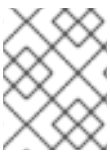
3. ReadWriteMany

4. 这不适用于物理磁盘、虚拟机物理磁盘、VMDK、NFS 回送、AWS EBS、Azure 磁盘和 Cinder（块后者）。

5. 对于监控组件，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，请不要在配置用于监控的持久性卷声明(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。每个日志记录都需要一个卷。

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 registry 是指一个 OpenShift Container Platform registry，它有三个或更多个 pod 运行副本。

5.3.1. 特定应用程序存储建议

重要

测试显示，使用 RHEL NFS 服务器作为容器镜像 registry 的存储后端可能会出现问题。这包括 OpenShift Container Registry 和 Quay、Prometheus for metrics 存储，以及 ElasticSearch for logging 存储。因此，不建议使用 RHEL NFS 服务器来备份核心服务使用的 PV。

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。

5.3.1.1. Registry

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

- 首选存储技术是对象存储，然后是块存储。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。所有 NAS 存储（包含聚合模式/独立模式 GlusterFS），不建议在带有生产环境工作负载的 OpenShift Container Platform Registry 集群部署中使用对象存储接口。
- 虽然 **hostPath** 卷对于一个非扩展的/HA OpenShift Container Platform Registry 是可配置的，但不推荐用于集群部署。

5.3.1.2. 扩展的 registry

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

- 首选存储技术是对象存储。存储技术必须支持 RWX 访问模式，且必须保证读写一致性。
- 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和块存储。
- 所有 NAS 存储（包含聚合模式/独立模式 GlusterFS），不建议在带有生产环境工作负载的 OpenShift Container Platform Registry 集群部署中使用对象存储接口。

5.3.1.3. 监控

在 OpenShift Container Platform 托管监控集群部署中：

- 首选存储技术是块存储。
- 如果您决定配置文件存储，请确保它遵循 POSIX 标准。

重要

测试显示使用 NFS 的显著不可恢复损坏，因此不建议使用。

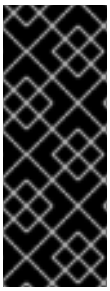
市场上的其他 NFS 实现可能没有这些问题。如需了解更多与此问题相关的信息，请联络相关的 NFS 厂商。

5.3.1.4. 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。

- 不建议使用 NAS 存储（包含聚合模式/独立模式 GlusterFS），因为它对带有生产环境负载的托管 metrics 集群部署使用 iSCSI 中的块存储接口。



重要

测试显示，在 RHEL 中使用 NFS 服务器作为容器镜像 registry 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 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。

5.3.1.5. 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。

5.3.2. 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。
- 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。

5.4. 选择图形驱动程序

容器运行时将镜像和容器存储在图形驱动程序中（可插拔式存储技术），如 DeviceMapper 和 OverlayFS。它们都有各自的优缺点。

有关 OverlayFS 的详情请参考 Red Hat Enterprise Linux(RHEL)7 发行注记 。

表 5.3. 图形驱动程序比较

名称	描述	优点	限制：
OverlayFS • overlay • overlay2	组合一个较低（父）和上层（子上）文件系统和工作目录（位于与子进程相同的文件系统上）。较低文件系统是基础镜像，当您创建新容器时，会创建一个包含 deltas 的新文件系统。	• 在启动和停止容器时比设备映射器更快。设备映射器和 Overlay 之间的启动时间差通常小于一秒。 • 允许页面缓存共享。	没有兼容 POSIX。

名称	描述	优点	限制：
设备映射器精简调配	使用 LVM、设备映射器和 dm-thinp 内核模块。它与一个原始分区（没有文件系统）删除回环设备不同。	• 负载和高密度方面具有可测量的性能优势。 • 它为每个容器提供容量限制（默认为 10G）。	• 您必须有一个专用的分区。 • 默认情况下，在 Red Hat Enterprise Linux(RHEL)中不会设置它。 • 所有容器和镜像共享相同的容量池。在销毁和重新创建池的情况下无法调整它的大小。
设备映射器 loop-lvm	使用设备映射器精简配置模块(dm-thin-pool)实施写时复制(CoW)快照。对于每个设备映射器图形位置，基于两个块设备创建精简池，一个用于数据，另一个用于元数据。默认情况下，这些块设备是使用自动创建稀疏文件的环回挂载自动创建。	它开箱即用，因此对于原型制作和开发目的非常有用。	• 不是 Unix(POSIX)功能的所有可端口操作系统接口的工作方式（例如，**O_DIRECT**）。最重要的是，不支持在生产环境中使用这个模式。 • 所有容器和镜像共享相同的容量池。在销毁和重新创建池的情况下无法调整它的大小。

为了提高性能，红帽建议在设备映射器中使用 overlayFS 存储驱动程序。但是，如果您已在生产环境中使用设备映射器，红帽强烈建议您对容器镜像和容器根文件系统使用精简配置。否则，始终将 overlayfs2 用于 Docker 引擎，或将 overlayFS 用于 CRI-O。

使用循环设备可能会影响性能。虽然您仍可以继续使用它，但会记录以下警告信息：

```
devmapper: Usage of loopback devices is strongly discouraged for production use.
Please use `--storage-opt dm.thinpooldev` or use `man docker` to refer to
dm.thinpooldev section.
```

要简化存储配置，请使用 **docker-storage-setup** 工具，它可以自动完成大部分配置详情：

对于 Overlay

1. 编辑 */etc/sysconfig/docker-storage-setup* 文件以指定设备驱动程序：

```
STORAGE_DRIVER=overlay2
```

注意

如果使用 CRI-O，请指定 **STORAGE_DRIVER=overlay**。

使用 CRI-O 时，默认的 **overlay** 存储驱动程序使用 **overlay2** 优化。

使用 OverlayFS，如果要在不同的逻辑卷上具有 **imagefs**，那么您必须设置 **CONTAINER_ROOT_LV_NAME** 和 **CONTAINER _ROOT_LV_MOUNT_PATH**。设置 **CONTAINER_ROOT_LV_MOUNT_PATH** 需要设置 **CONTAINER_ROOT_LV_NAME**。例如，**CONTAINER_ROOT_LV_NAME="container-root-lv"**。如需更多信息，请参阅使用 Overlay Graph Driver。

2. 如果您的独立磁盘驱动器有一个专用于 docker 存储（例如 */dev/xvdb*）的磁盘驱动器，请将以下内容添加到 */etc/sysconfig/docker-storage-setup* 文件中：

```
DEVS=/dev/xvdb
VG=docker_vg
```

3. 重启 **docker-storage-setup** 服务：

```
# systemctl restart docker-storage-setup
```

4. 要验证 docker 是否使用 overlay2，并监控磁盘空间使用情况，请运行 **docker info** 命令：

```
# docker info | egrep -i 'storage|pool|space|filesystem'
```

输出示例

```
Storage Driver: overlay2 1
 Backing Filesystem: extfs
```

1 使用 **overlay2** 时的 **docker info** 输出。

从 Red Hat Enterprise Linux 7.2 开始，OverlayFS 也支持容器运行时用例，并提供更快的启动时间和页面缓存共享，这可以通过降低总体内存使用率来更好地提高密度。

对于 Thinpool

1. 编辑 */etc/sysconfig/docker-storage-setup* 文件以指定设备驱动程序：

```
STORAGE_DRIVER=devicemapper
```

2. 如果您的独立磁盘驱动器有一个专用于 docker 存储（例如 */dev/xvdb*）的磁盘驱动器，请将以下内容添加到 */etc/sysconfig/docker-storage-setup* 文件中：

```
DEVS=/dev/xvdb
VG=docker_vg
```

3. 重启 **docker-storage-setup** 服务：

```
# systemctl restart docker-storage-setup
```

重启后，**docker-storage-setup** 会设置名为 **docker_vg** 的卷组，并创建一个 thin-pool 逻辑卷。有关 RHEL 的精简配置的文档，请参考 LVM 管理员指南。使用 **lsblk** 命令查看新创建的卷：

```
# lsblk /dev/xvdb
```

输出示例

```
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvdb 202:16 0 20G 0 disk
└─xvdb1 202:17 0 10G 0 part
  ├─docker_vg-docker--pool_tmeta 253:0 0 12M 0 lvm
  │ └─docker_vg-docker--pool 253:2 0 6.9G 0 lvm
  └─docker_vg-docker--pool_tdata 253:1 0 6.9G 0 lvm
    └─docker_vg-docker--pool 253:2 0 6.9G 0 lvm
```

注意

精简配置的卷未挂载，也没有文件系统（individual 容器具有 XFS 文件系统），因此不会在 **df** 输出中出现。

4. 要验证 docker 是否使用 LVM thinpool，并用于监控磁盘空间使用情况，请运行 **docker info** 命令：

```
# docker info | egrep -i 'storage|pool|space|filesystem'
```

输出示例

```
Storage Driver: devicemapper (1)
 Pool Name: docker_vg-docker--pool (2)
Pool Blocksize: 524.3 kB
Backing Filesystem: xfs
Data Space Used: 62.39 MB
Data Space Total: 6.434 GB
Data Space Available: 6.372 GB
Metadata Space Used: 40.96 kB
Metadata Space Total: 16.78 MB
Metadata Space Available: 16.74 MB
```

(1) 使用 **devicemapper** 时 **docker info** 输出。

(2) 对应于您在 */etc/sysconfig/docker-storage-setup* 中指定的 **VG**。

默认情况下，精简池配置为使用基础块设备的 40%。当您使用存储时，LVM 会自动将精简池扩展到 100%。这就是为什么 **Data Space Total** 值与基础 LVM 设备的大小不匹配。

在开发中，红帽分发中的 docker 默认为环回挂载的稀疏文件。查看您的系统是否使用回环模式：

```
# docker info|grep loop0
```

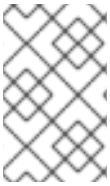
输出示例

```
Data file: /dev/loop0
```

5.4.1. 在 SELinux 中使用 OverlayFS 或 DeviceMapper 的好处

OverlayFS 图形的主要优点是 Linux 页面缓存在共享同一节点上的镜像的容器之间共享。OverlayFS 的此属性会导致在容器启动期间减少输入/输出(I/O)，从而加快容器启动时间，同时减少类似镜像在节点上运行时的内存用量。这些结果在许多环境中都很有用，特别是那些针对密度优化且具有高容器数量率（如构建场）或镜像内容具有重大重叠的目标。

DeviceMapper 无法进行页面缓存共享，因为精简配置的设备会基于每个容器分配。



注意

OverlayFS 是 Red Hat Enterprise Linux(RHEL)7.5 的默认 Docker 存储驱动程序，在 7.3 及更新的版本中被支持。将 OverlayFS 设置为 RHEL 上的默认 Docker 存储配置，以提高性能。有关配置 OverlayFS 以便与 Docker 容器运行时搭配使用的说明。

5.4.2. 比较 Overlay 和 Overlay2 图形驱动程序

OverlayFS 是一种联合文件系统。它允许您在一个文件系统上使用另外一个文件系统。更改记录在上面的文件系统中，而较小的文件系统则未修改。这允许多个用户共享文件系统镜像，如容器或 DVD-ROM，基础镜像使用只读介质。

OverlayFS 在一个 Linux 主机上对两个目录进行分层，并将其呈现为一个目录。这些目录称为层，卸载过程指代为联合挂载。

OverlayFS 使用两个图形驱动程序之一，即 **overlay** 或 **overlay2**。从 Red Hat Enterprise Linux 7.2 开始， **overlay** 成为受支持的图形驱动程序。从 Red Hat Enterprise Linux 7.4 开始， **overlay2** 开始被支持。Red Hat Enterprise Linux 7.4 中支持 docker 守护进程中的 SELinux。有关在您的 RHEL 版本中使用 OverlayFS 的详情，包括支持性和用法注意事项，请查看 Red Hat Enterprise Linux 发行注记 。

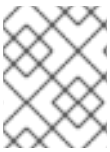
overlay2 驱动程序原生支持多达 128 个低 OverlayFS 层，但 **overlay** 驱动程序只能处理一个较低 OverlayFS 层。因此，**overlay2** 驱动程序为与层相关的 Docker 命令（如 **docker build** ）提供了更好的性能，并在后备文件系统中消耗较少的内节点。

因为 **overlay** 驱动程序与一个较低 OverlayFS 层工作，所以您无法将多层镜像作为多个 OverlayFS 层实现。相反，每个镜像层都以自己的目录的形式在 */var/lib/docker/overlay* 下实施。然后，硬链接将用作一个空间效率的方式来引用与较低层共享的数据。

Docker 建议使用带有 OverlayFS 而不是 **overlay** 驱动程序的 **overlay2** 驱动程序，因为它在内节点使用率方面效率更高。

第 6 章 优化临时存储

6.1. 概述



注意

只有在您启用了临时存储技术预览功能时，才会应用这个主题。此功能默认为禁用。要启用此功能，请参阅为临时存储配置。

注意

红帽产品服务等级协议(SLA)不支持技术预览版本，且其功能可能并不完善，红帽不建议在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。如需更多信息，请参阅红帽技术预览功能支持范围。

Pod 将临时存储用于其内部操作，如保存临时文件。此临时存储的生命周期不会超过每个 pod 的生命周期，且此临时存储无法在 pod 间共享。

在 OpenShift Container Platform 3.10 之前，临时本地存储通过容器的可写层、日志目录和 EmptyDir 卷公开给 pod。与缺少本地存储相关的问题包括：

- Pod 不知道有多少可用的本地存储。
- Pod 无法请求保证的本地存储。
- 本地存储无法保证可以满足需求。
- Pod 可能会因为其他 pod 填充本地存储而被驱除。在足够的存储被重新声明前，不会接受新的 pod。

临时存储仍以同样的方式公开给 pod，但有新方法在临时存储消耗的 pod 上实施请求和限制。

注意

只有在将 CRI-O 用作容器运行时和基于文件的日志进行日志记录时才应用容器日志的管理。

务必要了解，临时存储在系统中的所有 pod 中共享，而且 OpenShift Container Platform 不提供任何级别的服务保证在管理员和用户建立的请求和限值之外。例如，临时存储不提供任何吞吐量、每秒 I/O 操作或存储性能的其他测量结果。

6.2. 常规存储指南

节点的本地存储可分为主分区和从属分区。主分区是唯一可用于临时本地存储的主分区。有两个支持的主分区，即 root 和 runtime。

- root
  根分区默认存放 kubelet 的根目录、**/var/lib/kubelet/** 和 **/var/log/** 目录。您可以在 pod、操作系统和 OpenShift Container Platform 系统守护进程间共享这个分区。Pod 可以通过使用 EmptyDir 卷、容器日志、镜像层和容器可写入层来访问此分区。OpenShift Container Platform 管理此分区的共享访问和隔离。
- Runtime

运行时分区是可用于覆盖文件系统的可选分区。OpenShift Container Platform 会尝试识别并提供共享访问以及这个分区的隔离。此分区包含容器镜像层和可写入层。如果运行时分区存在，则 **root** 分区不包含任何镜像层或可写入层。

第 7 章 网络优化

7.1. 优化网络性能

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。

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：

- 使用原生容器路由。此选项在使用 OpenShift SDN 时有一个重要的操作注意事项，如更新路由器上的路由表。
- 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。

7.1.1. 为您的网络优化 MTU

有两个重要的最大传输单元(MTU)：网卡(NIC)MTU 和 SDN 覆盖的 MTU。

NIC 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。在巨型帧以太网网络中，将其设置为 8950。

注意

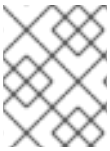
这个 50 字节覆 overlay 头与 OpenShift SDN 相关。其他 SDN 解决方案可能需要该值更大或更少。

要配置 MTU，请编辑适当的节点配置映射 并修改以下部分：

```
networkConfig:
  mtu: 1450 1
  networkPluginName: "redhat/openshift-ovs-subnet" 2
```

1 pod 覆盖网络的最大传输单元(MTU)。

2 为 **ovs-subnet** 插件设置为 **redhat/openshift-ovs-subnet**，为 **ovs-multitenant** 插件设置为 **redhat/openshift-ovs-multitenant**，为 **ovs-networkpolicy** 插件设置为 **redhat/openshift-ovs-**



注意

您必须在所有 master 和作为 OpenShift Container Platform SDN 一部分的节点上更改 MTU 大小。另外，tun0 接口的 MTU 大小必须在属于集群的所有节点中相同。

7.2. 配置网络子网

OpenShift Container Platform 为 pod 和服务提供 IP 地址管理。允许使用的默认值：

- 最大集群大小为 1024 个节点
- 每个 1024 节点都分配有 /23（用于 pod 的 510 可用 IP）
- 大约 65,536 IP 地址用于服务

在大多数情形中，部署之后无法更改这些网络。因此，规划增长非常重要。

在 配置 SDN 文档 中对网络重新定义大小的限制。

要计划较大的环境，可以考虑将下列值添加到 Ansible 清单文件中的 **[OSE3:vars]** 部分：

```
[OSE3:vars]
osm_cluster_network_cidr=10.128.0.0/10
```

这将允许 8192 个节点，每个节点有 510 可用 IP 地址。

如需了解您要安装的软件版本，请参阅 OpenShift Container Platform 文档中的对节点/pod 中的支持性限制。

7.3. 优化 IPSEC

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。

第 8 章 路由优化

8.1. 扩展 OPENSHIFT CONTAINER PLATFORM HAPROXY 路由器

8.1.1. 基准性能

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，我们的实验室在一个有 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	ROUTER_THREADS unset	ROUTER_THREADS=4
none	23681	24327
edge	14981	22768
passthrough	34358	34331
re-encrypt	13288	24605

在 HTTP 关闭（无 keep-alive）情境中：

Encryption	ROUTER_THREADS unset	ROUTER_THREADS=4
none	3245	4527
edge	1910	3043
passthrough	3408	3922

Encryption	ROUTER_THREADS unset	ROUTER_THREADS=4
re-encrypt	1333	2239

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。

应该使用 路由器分片 为应用程序提供更多路由，并帮助水平扩展路由层。

8.1.2. 性能优化

8.1.2.1. 设置连接的最大数量

HAProxy 可扩展性最重要的可调整参数之一是 **maxconn** 参数，该参数将每个进程的最大并发连接数设置为给定数字。通过编辑 OpenShift Container Platform HAProxy 路由器部署配置文件中的 **ROUTER_MAX_CONNECTIONS** 环境变量来调整此参数。

注意

连接包括 frontend 和 internal 后端。这表示两个连接。务必将 **ROUTER_MAX_CONNECTIONS** 设置为两倍，超过您要创建的连接数。

8.1.2.2. CPU 和中断关联性

在 OpenShift Container Platform 中，HAProxy 路由器以单一进程的形式运行。OpenShift Container Platform HAProxy 路由器通常在具有较少但频率较高的内核的系统上更好地执行，而不是在大量频率较高的对称多进程(SMP)系统中执行。

将 HAProxy 进程固定到一个 CPU 内核，而对另一个 CPU 内核的网络中断往往会提高网络性能。在相同的非统一内存访问(NUMA)节点上具有进程和中断，有助于通过确保共享的 L3 缓存来避免内存访问。但是，公共云环境中通常无法进行这种级别的控制。在裸机主机上，**irqbalance** 会自动处理外围组件互连(PCI)本地化，以及用于中断请求行(IRQ)的 NUMA 关联性。在云环境中，此类信息通常不提供给操作系统。

CPU 固定通过 **taskset** 或使用 HAProxy 的 **cpu-map** 参数来执行。此指令采用两个参数：进程 ID 和 CPU 内核 ID。例如，要将 HAProxy 进程 **1** 固定到 CPU 内核 **0**，请将以下行添加到 HAProxy 配置文件的全局部分：

-

```
cpu-map 1 0
```

要修改 HAProxy 配置文件，请参阅部署 Customized HAProxy Router 。

8.1.2.3. 增加线程数

HAProxy 路由器在 OpenShift Container Platform 中支持多线程。在多个 CPU 内核系统上，增加线程数量可帮助性能，特别是在路由器终止 SSL 时。

要指定 HAProxy 路由器的线程数量，请参阅启用 HAProxy 线程和路由器环境变量。

8.1.2.4. 增加缓冲的影响

OpenShift Container Platform HAProxy 路由器请求缓冲区配置限制了传入的请求和来自应用程序响应中的标头大小。可以增加 HAProxy 参数 **tune.bufsize** 以允许处理更大的标头，并允许具有非常大 Cookie 的应用程序正常工作，如许多公共云提供商提供的负载均衡器接受的应用程序。但是，这会影响内存使用总量，特别是在打开大量连接时。使用大量打开连接时，内存用量就与这个可调参数增长几乎成比例。

8.1.2.5. HAProxy Reloads 的优化

较长的连接（如 WebSocket 连接）与较长的客户端/服务器 HAProxy 超时和短暂的 HAProxy 重新加载间隔相结合，可能会导致许多 HAProxy 进程实例化。这些进程必须处理旧的连接，这些连接在 HAProxy 配置重新加载之前启动。大量这些进程不可取，因为它会对系统造成不必要的负载，并可能导致问题，如内存不足的情况。

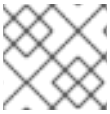
影响此行为的路由器环境变量是 **ROUTER_DEFAULT_TUNNEL_TIMEOUT**、**ROUTER_DEFAULT_CLIENT_TIMEOUT**、**ROUTER_DEFAULT_SERVER_TIMEOUT** 以及 **RELOAD_INTERVAL**。

第 9 章 扩展集群指标

9.1. 概述

OpenShift Container Platform 会公开指标，这些指标数据可由 Heapster 收集并存储在后端。作为 OpenShift Container Platform 管理员，您可以查看一个用户界面中的容器和组件指标。pod 横向自动扩展 也会使用这些指标来确定何时和如何扩展。

本节提供了扩展指标数据组件的信息。



注意

OpenShift Container Platform 不支持自动扩展指标组件，如 Hawkular 和 Heapster。

9.2. 针对 OPENSHIFT CONTAINER PLATFORM 的建议

- 在专用 OpenShift Container Platform 基础架构节点上运行 指标 pod。
- 在配置指标时使用持久性存储。Set **USE_PERSISTENT_STORAGE=true**.
- 在 OpenShift Container Platform 指标部署中保留 **METRICS_RESOLUTION=30** 参数。不建议为 **METRICS_RESOLUTION** 使用低于默认值 **30** 的值。在使用 Ansible 指标安装过程时，这是 **openshift_metrics_resolution** 参数。
- 使用主机 metrics pod 密切监控 OpenShift Container Platform 节点，以检测主机系统的早期容量短（CPU 和内存）。这些容量不足可能会导致指标 pod 出现问题。
- 在 OpenShift Container Platform 版本 3.7 测试中，在 OpenShift Container Platform 集群中监控最多 25,000 个 pod 的测试情况。

9.3. 集群指标的容量规划

如果测试中使用了 210 和 990 OpenShift Container Platform 节点，其中 10500 个 pod 和 11000 个 pod 分别被监控，Cassandra 数据库会在下表中所示的速度增长：

表 9.1. Cassandra 基于集群中的节点/pod 数量的数据库存储要求

节点数量	Pod 数	Cassandra 存储增长速度	Cassandra 每天增长存储	每周增长的 Cassandra 存储
210	10500	每小时 500 MB	15 GB	75 GB
990	11000	每小时 1 GB	30 GB	210 GB

在上个计算中，预计大小大约有 20% 增加了开销，以确保存储要求不会超过计算的值。

如果 **METRICS_DURATION** 和 **METRICS_RESOLUTION** 值保持在默认值（**7** 天和 **15** 秒），则安全地规划星期的 Cassandra 存储要求。

警告

由于 OpenShift Container Platform 指标使用 Cassandra 数据库作为指标数据的数据存储，如果在指标设置过程中设置了 **USE_PERSISTENT_STORAGE=true**，则 **PV** 将位于网络存储的顶部，且 NFS 用作默认数据。但不建议将网络存储与 Cassandra 结合使用。

如果您使用 Cassandra 数据库作为指标数据的数据存储，请参阅 Cassandra 文档 了解其建议。

9.4. 扩展 OPENSHIFT CONTAINER PLATFORM 指标 POD

一组指标 pod(Cassandra/Hawkular/Heapster)能够监控至少 25,000 个 pod。

小心

请注意运行 OpenShift Container Platform 指标 Pod 的节点上的系统负载。使用这些信息来确定是否需要横向扩展多个 OpenShift Container Platform 指标 pod，并将负载分散到多个 OpenShift Container Platform 节点上。不建议扩展 OpenShift Container Platform 指标 heapster pod。

9.4.1. 先决条件

如果使用持久性存储来部署 OpenShift Container Platform 指标，则必须为新的 Cassandra pod 创建一个持久性卷(PV)， 然后才能扩展 OpenShift Container Platform metrics Cassandra Pod 的数量。但是，如果 Cassandra 部署有动态置备的 PV，则不需要这一步。

9.4.2. 扩展 Cassandra 组件

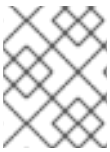
Cassandra 节点使用持久性存储。因此，无法使用复制控制器进行缩放或缩减。

扩展 Cassandra 集群需要修改 **openshift_metrics_cassandra_replicas** 变量，然后重新运行 部署。默认情况下，Cassandra 集群是一个单节点集群。

要将 OpenShift Container Platform 指标的 pod 数量扩展到两个副本，请运行：

```
# oc scale -n openshift-infra --replicas=2 rc hawkular-metrics
```

或者，更新清单文件并重新运行 deployment。



注意

如果您在 Cassandra 集群中添加新节点或删除现有节点，则存储在集群中的数据会在集群中重新平衡。

缩减：

1. 如果远程访问容器，请对您要删除的 Cassandra 节点运行以下命令：

```
$ oc exec -it <hawkular-cassandra-pod> nodetool decommission
```

如果本地访问容器，请运行以下命令：

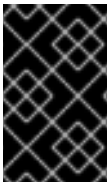
```
$ oc rsh <hawkular-cassandra-pod> nodetool decommission
```

此命令可能需要稍等片刻，因为它在集群中复制数据。您可以使用 **nodetool netstats -H** 监控未完成的进度。

2. 上一命令成功后，将 Cassandra 实例的 **rc** 缩减为 **0**。

```
# oc scale -n openshift-infra --replicas=0 rc <hawkular-cassandra-rc>
```

这将删除 Cassandra 容器集。



重要

如果完成缩减流程并且现有 Cassandra 节点按预期运行，您也可以删除此 Cassandra 实例的 **rc** 及其对应的持久卷声明(PVC)。删除 PVC 可能会永久删除与此 Cassandra 实例关联的任何数据，因此如果缩减没有完全且成功完成，您将无法恢复丢失的数据。

第 10 章 扩展 CLUSTER MONITORING OPERATOR

10.1. 概述

OpenShift Container Platform 会公开可由 cluster-monitoring-operator 收集并存储在后端的指标。作为 OpenShift Container Platform 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。

本节提供有关扩展集群监控操作器的信息。

如果要将 Prometheus 与持久性存储搭配使用，您必须将 Ansible 清单文件中的 **openshift_cluster_monitoring_operator_prometheus_storage_enabled** 变量设置为 **true**。

10.2. 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。
- 使用持久块存储，如 OpenShift Container Storage(OCS)Block。

10.3. CLUSTER MONITORING OPERATOR 的容量规划

对不同的扩展大小执行各种测试。Prometheus 数据库会增加，如下表中所示。

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 10.1. 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

在上个计算中，预计大小大约有 20% 增加了开销，以确保存储要求不会超过计算的值。

以上计算是为默认的 OpenShift Container Platform **cluster-monitoring-operator** 开发的。为实现更高的扩展，编辑 Ansible 清单文件中的 **openshift_cluster_monitoring_operator_prometheus_storage_capacity** 变量，默认值为 **50Gi**。

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。

10.3.1. 实验室环境

所有实验都是在 OpenStack 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。

10.3.2. 先决条件

根据您的缩放目的地，计算并为 Prometheus 数据存储设置相关 PV 大小。由于默认的 Prometheus pod 副本为 2，因此对于具有 3600 个 pod 的 100 个节点，您将需要 188 GB。

例如：

```
195 GB (space per 15 days ) * 2 (pods) = 390 GB free
```

根据此公式，设置 **openshift_cluster_monitoring_operator_prometheus_storage_capacity=195Gi**。

第 11 章 测试每个集群的最大值

在规划 OpenShift Container Platform 集群时，请考虑以下测试的集群对象限制。

这些限制基于最大可能的集群。对于较小的集群，最大值会按比例减小。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。

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

11.1. OPENSHIFT CONTAINER PLATFORM 为主版本测试的集群最大限制

最大类型	3.x 测试的最大值
节点数量	2,000
Pod 数 [1]	150,000
每个节点的 pod 数量	250
每个内核的 pod 数量	没有默认值。
命名空间的数量	10,000
构建(build)数：管道策略	10,000（默认 pod RAM 512Mi）
每个命名空间的 pod 数量 [2]	25,000
服务数 [3]	10,000
每个命名空间的服务数	5,000
每个服务中的后端数	5,000
每个命名空间的部署数量 [2]	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. 系统中有一些控制循环需要迭代给定命名空间中的所有对象，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。最大值假设系统有足够的 CPU、内存和磁盘来满足应用程序要求。
3. 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。

11.2. OPENSHIFT CONTAINER PLATFORM 测试的集群最大限制

最大类型	3.7 测试的最大值	3.9 测试的最大值	3.10 测试的最大值	3.11 测试的最大值
节点数量	2,000	2,000	2,000	2,000
Pod 数 [1]	120,000	120,000	150,000	150,000
每个节点的 pod 数量	250	250	250	250
每个内核的 pod 数量	10 是默认值。	10 是默认值。	没有默认值。	没有默认值。
命名空间的数量	10,000	10,000	10,000	10,000
构建(build)数：管道策略	N/A	10,000（默认 pod RAM 512Mi）	10,000（默认 pod RAM 512Mi）	10,000（默认 pod RAM 512Mi）
每个命名空间的 pod 数量 [2]	3,000	3,000	3,000	25,000
服务数 [3]	10,000	10,000	10,000	10,000
每个命名空间的服务数	N/A	N/A	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 [2]	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. 系统中有一些控制循环需要迭代给定命名空间中的所有对象，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。最大值假设系统有足够的 CPU、内存和磁盘来满足应用程序要求。

3. 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。

11.2.1. 路由最大限制

在 OpenShift Container Platform 3.11.53 中，路由器测试在 Amazon Web Services(AWS)上的 3 节点环境中完成。有 100 个 HTTP 路由，特别是 100 个后端 Nginx pod，**keepalive** 设为 **100**。结果为：

- 每个目标路的 1 个连接 = 每秒 24,327 个请求

- 每个目标路的 40 个连接 = 每秒 20,729 个请求
- 每个目标路由 200 个连接 = 每秒 17,253 个请求

11.3. 测试 OPENSHIFT CONTAINER PLATFORM 集群最大值的环境和配置

基础架构即服务：OpenStack

节点	vCPU	RAM(MiB)	磁盘大小(GiB)	透传磁盘	数量
Master/Etcd [1]	16	124672	128	是，NVMe	3
Infra [2]	40	163584	256	是，NVMe	3
集群 DNS	1	1740	71	否	1
Load Balancer	4	16128	96	否	1
原生存储 [3]	16	65280	200	是，NVMe	3
堡垒 [4]	16	65280	200	否	1
Worker	2	7936	96	否	2000

1. master/etcd 节点由 NVMe 磁盘支持，因为 etcd 非常大，且敏感延迟。
2. Infra 节点托管路由器、Registry、日志记录和监控，并由 NVMe 磁盘支持。
3. 容器原生存储或 Ceph 存储节点由 NVMe 磁盘支持。
4. Bastion 节点是 OpenShift Container Platform 网络的一部分，用于编配性能和扩展测试。

11.4. 规划环境以延长集群最大值

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出现内存交换问题。

某些测试的最大值仅在单一维度中扩展，因此当很多对象在集群中运行时，它们可能会有所不同。

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。

在规划环境时，请确定每个节点应该有多少个 pod:

> Maximum Pods per Cluster / Expected Pods per Node = Total Number of Nodes

在某个节点中运行的 pod 数量取决于应用程序本身。考虑应用程序的内存、CPU 和存储要求。

使用情况示例

如果想把集群的规模限制在没有集群可以有 2200 个 pod，则需要至少有九个节点，假设每个节点最多有 250 个 pod：

> 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

11.5. 规划环境符合应用程序要求

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	500MB	0.5	1GB
node.js	200	1GB	1	1GB
postgresql	100	1GB	2	10GB
JBoss EAP	100	1GB	1	1GB

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，有些应用程序则不同。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 12 章 使用 CLUSTER LOADER

12.1. CLUSTER LOADER 执行的操作

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。

12.2. 安装 CLUSTER LOADER

Cluster Loader 包含在 **atomic-openshift-tests** 软件包中。要安装它，请运行：

```
$ yum install atomic-openshift-tests
```

安装后，测试可执行文件 ***extended.test*** 位于 ***/usr/libexec/atomic-openshift/extended.test*** 中。

12.3. 运行 CLUSTER LOADER

1. 将 **KUBECONFIG** 变量设置为管理员 **kubeconfig** 的位置：

   ```
   $ export KUBECONFIG=${KUBECONFIG-$HOME/.kube/config}
   ```

2. 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：

   ```
   $ cd /usr/libexec/atomic-openshift/
   ```

   ```
   $ ./extended.test --ginkgo.focus="Load cluster"
   ```

   或者，通过为 **--viper-config** 添加标记来执行带有用户定义的配置的 Cluster Loader：

   ```
   $ ./extended.test --ginkgo.focus="Load cluster" --viper-config=config/test (1)
   ```

   (1) 在本例中，有一个名为 ***config/*** 的子目录，其配置文件名为 ***test.yml***。在命令行中，排除配置文件的扩展，因为工具会自动决定文件类型和扩展。

12.4. 配置 CLUSTER LOADER

创建多个命名空间（项目），其中包含多个模板或 pod。

在 ***config/*** 子目录中找到 Cluster Loader 的配置文件。这些配置示例中引用的 pod 文件和模板文件可在 ***content/*** 子目录中找到。

12.4.1. 配置字段

表 12.1. 顶层 Cluster Loader 字段

字段	描述

字段	描述
cleanup	可设置为 **true** 或 **false**。每个配置有一个定义。如果设为 **true**，则 **cleanup** 将删除由 Cluster Loader 在测试结束时创建的所有命名空间（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 12.2. **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。
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 12.3. **pods** 和 **templates** 下的字段

字段	描述
total	没有使用此字段。

字段	描述
num	整数。要部署的 pod 或模板数量。
image	字符串到可以拉取的存储库的容器镜像 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 12.4. **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 12.5. **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 12.6. **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 12.7. 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]

12.4.2. 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: ./examples/quickstarts/cakephp-mysql.json

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

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

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

  - num: 1
    basename: clusterloader-rails-postgresql
    tuning: default
    templates:
      - num: 1
        file: ./examples/quickstarts/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。

12.5. 已知问题

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 13 章 使用 CPU MANAGER

13.1. CPU MANAGER 的作用

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。

13.2. 设置 CPU MANAGER

设置 CPU Manager：

1. 另外，还可为节点添加标签：

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

2. 在目标节点上启用 CPU Manager 支持：

```
# oc edit configmap <name> -n openshift-node
```

例如：

```
# oc edit cm node-config-compute -n openshift-node
```

输出示例

```
...
kubeletArguments:
...
  feature-gates:
  - CPUManager=true
  cpu-manager-policy:
  - static
  cpu-manager-reconcile-period:
  - 5s
  system-reserved: 1
  - cpu=500m
```

```
# systemctl restart atomic-openshift-node
```

1 **system-reserved** 是一个必需的设置。可能需要根据您的环境调整值。

3. 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.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"
```

4. 创建 pod：

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

5. 确定为您标记的节点调度了 pod：

```
# oc describe pod cpumanager
```

输出示例

```
Name:         cpumanager-4gdtn
Namespace:    test
Node:         perf-node.example.com/172.31.62.105
...
    Limits:
      cpu:     1
      memory:  1G
    Requests:
      cpu:        1
      memory:     1G
...
QoS Class:       Guaranteed
Node-Selectors:  cpumanager=true
                 region=primary
```

6. 确认正确配置了 **cgroups**。获取暂停进程的 PID：

```
# systemd-cgls -l
```

输出示例

```
├─1 /usr/lib/systemd/systemd --system --deserialize 20
├─kubepods.slice
│ ├─kubepods-pod0ec1ab8b_e1c4_11e7_bb22_027b30990a24.slice
│ │ ├─docker-
b24e29bc4021064057f941dc5f3538595c317d294f2c8e448b5e61a29c026d1c.scope
│ │ │ └─44216 /pause
```

QoS 等级为 **Guaranteed** 的 pod 放置在 **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-
pod0ec1ab8b_e1c4_11e7_bb22_027b30990a24.slice/docker-
b24e29bc4021064057f941dc5f3538595c317d294f2c8e448b5e61a29c026d1c.scope
# for i in `ls cpuset.cpus tasks` ; do echo -n "$i "; cat $i ; done
```

输出示例

```
cpuset.cpus 2
tasks 44216
```

7. 检查任务允许的 CPU 列表：

```
# grep ^Cpus_allowed_list /proc/44216/status
```

输出示例

```
Cpus_allowed_list:     2
```

8. 验证系统中的另一个 pod（在这个示例中，**burstable** QoS 等级为 burstable 的 pod）无法在为 **Guaranteed** pod 分配的内核中运行：

```
# cat /sys/fs/cgroup/cpuset/kubepods.slice/kubepods-burstable.slice/kubepods-burstable-
podbe76ff22_dead_11e7_b99e_027b30990a24.slice/docker-
da621bea7569704fc39f84385a179923309ab9d832f6360cccbff102e73f9557.scope/cpuset.cpus

0-1,3
```

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

输出示例

```
...
Capacity:
 cpu:     4
 memory:  16266720Ki
 pods:    40
Allocatable:
 cpu:     3500m
 memory:  16164320Ki
 pods:    40
```

```
---
  Namespace                  Name                CPU Requests  CPU Limits  Memory Requests
Memory Limits
  ---------                  ----                ------------  ----------  ---------------
  test                       cpumanager-4gdtn    1 (28%)       1 (28%)     1G (6%)
  test                       cpumanager-hczts    1 (28%)       1 (28%)     1G (6%)
  test                       cpumanager-r9wrq    1 (28%)       1 (28%)     1G (6%)
...
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  CPU Requests  CPU Limits  Memory Requests  Memory Limits
  ------------  ----------  ---------------  -------------
  3 (85%)       3 (85%)     5437500k (32%)   9250M (55%)
```

此虚拟机有四个 CPU 内核。将 **system-reserved** 设置为 500 毫秒，即从节点的总容量中减去一个内核的一半，以达到 **Node Allocatable** 的数量。

您可以看到，**Allocatable CPU** 为 3500 millicore。这意味着，我们可以运行三个 CPU Manager pod，因为每个 pod 都需要一个完整的内核。一个完整的内核等于 1000 毫秒。

如果您尝试调度第四个 pod，系统将接受该 pod，但不会调度它：

```
# oc get pods --all-namespaces |grep test
```

输出示例

```
test          cpumanager-4gdtn          1/1       Running       0          8m
test          cpumanager-hczts          1/1       Running       0          8m
test          cpumanager-nb9d5          0/1       Pending       0          8m
test          cpumanager-r9wrq          1/1       Running       0          8m
```

第 14 章 管理大页面

14.1. 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 内存为 262,144 页等。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。本主题介绍了如何进行。

14.2. 先决条件

1. 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。

14.3. 消耗大页面

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，如果节点支持 2048KiB 页面大小，它将提供一个可调度的资源 **hugepages-2Mi**。与 CPU 或内存不同，巨页不支持过量使用。

```
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - securityContext:
      privileged: true
    image: rhel7:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-2Mi: 100Mi (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。OpenShift Container Platform 会进行相应的计算。如上例所示，您可以直接指定 **100MB** 。

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。如需更多信息，请参阅配置透明巨页。

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。

第 15 章 在 GLUSTERFS 存储上进行优化

15.1. 数据库聚合模式指南

当您将聚合模式用于应用程序时，请遵循本主题中提供的指导和最佳实践，以便您可以根据工作负载类型在 gluster-block 和 GlusterFS 模式之间进行明智选择。

15.2. 测试的应用程序

在 OpenShift Container Platform 3.10 中，使用这些 (no)SQL 数据库进行了大量测试：

- Postgresql SQL v9.6
- MongoDB noSQL v3.2

这些数据库的存储源自聚合模式存储集群。

用于 Postgresql SQL 基准测试 pgbench，用于数据库基准测试。对于 MongoDB noSQL 基准测试 YCSB Yahoo!Cloud Serving Benchmark 用于基准测试， workloada,workloadb,workloadf 被测试

15.3. 支持列表

表 15.1. 表标题 - GlusterFS

数据库	存储后端： **GlusterFS**	**关闭** Performance Translators	**打开** Performance Translators
Postgresql SQL	是	• performance.stat-prefetch • performance.read-ahead • performance.write-behind • performance.readdir-ahead • performance.io-cache • performance.quick-read • performance.open-behind	• performance.strict-o-direct

MongoDB noSQL	是	• performance.stat-prefetch • performance.read-ahead • performance.write-behind • performance.readdir-ahead • performance.io-cache • performance.quick-read • performance.open-behind	• performance.strict-o-direct

表 15.2. 表标题 - gluster-block

数据库	存储后端： **gluster-block**
Postgresql	是
MongoDB	是

如上所述，GlusterFS 的性能转换器已经是附带最新聚合模式镜像的数据库配置集的一部分。

15.4. 测试结果

对于 Postgresql SQL 数据库，GlusterFS 和 gluster-block 会显示大约相同的性能结果。对于 MongoDB noSQL 数据库，gluster-block 更好地执行。因此，对 MongoDB noSQL 数据库使用基于 gluster-block 的存储。