OpenShift Container Platform 4.6

可伸缩性和性能

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

Last Updated: 2023-02-27

OpenShift Container Platform 4.6 可伸缩性和性能

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

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

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 6
1.1. 安装大型集群的实践建议 6

第 2 章 推荐的主机实践 7
2.1. 推荐的节点主机实践 7
2.2. 创建 KUBELETCONFIG CRD 来编辑 KUBELET 参数 7
2.3. CONTROL PLANE 节点大小 10
2.3.1. 增加 Amazon Web Services(AWS)master 实例的类别大小 11
2.4. 推荐的 ETCD 实践 11
2.5. 分离 ETCD 数据 13
2.6. OPENSHIFT CONTAINER PLATFORM 基础架构组件 15
2.7. 移动监控解决方案 16
2.8. 移动默认 REGISTRY 17
2.9. 移动路由器 18
2.10. 基础架构节点大小 20
2.11. 其他资源 21

第 3 章 推荐的集群扩展实践 22
3.1. 扩展集群的建议实践 22
3.2. 修改机器集 22
3.3. 关于机器健康检查 23
3.3.1. Bare Metal 上的 MachineHealthCheck 24
3.3.2. 部署机器健康检查时的限制 24
3.4. MACHINEHEALTHCHECK 资源示例 24
3.4.1. 短路机器健康检查补救 26
3.4.1.1. 使用绝对值设置 maxUnhealthy 26
3.4.1.2. 使用百分比设置 maxUnhealthy 27
3.5. 创建 MACHINEHEALTHCHECK 资源 27

第 4 章 使用 NODE TUNING OPERATOR 28
4.1. 关于 NODE TUNING OPERATOR 28
4.2. 访问 NODE TUNING OPERATOR 示例规格 28
4.3. 在集群中设置默认配置集 28
4.4. 验证是否应用了 TUNED 配置集 30
4.5. 自定义调整规格 31
4.6. 自定义调整示例 35
4.7. 支持的 TUNED 守护进程插件 35

第 5 章 使用 CLUSTER LOADER 37
5.1. 安装 CLUSTER LOADER 37
5.2. 运行 CLUSTER LOADER 37
5.3. 配置 CLUSTER LOADER 37
5.3.1. Cluster Loader 配置文件示例 37
5.3.2. 配置字段 39
5.4. 已知问题 41

第 6 章 使用 CPU MANAGER 43
6.1. 设置 CPU MANAGER 43

第 7 章 使用拓扑管理器 48
7.1. 拓扑管理器策略 48
7.2. 设置拓扑管理器 48
7.3. POD 与拓扑管理器策略的交互 49

第 8 章 扩展 CLUSTER MONITORING OPERATOR 51
8.1. PROMETHEUS 数据库存储要求 51
8.2. 配置集群监控 51

第 9 章 根据对象限制规划您的环境 53
9.1. OPENSHIFT CONTAINER PLATFORM 为主发行版本测试了集群最大值 53
9.2. 测试集群最大值的 OPENSHIFT CONTAINER PLATFORM 环境和配置 54
9.3. 如何根据经过测试的集群限制规划您的环境 55
9.4. 如何根据应用程序要求规划您的环境 56

第 10 章 优化存储 59
10.1. 可用的持久性存储选项 59
10.2. 推荐的可配置存储技术 59
10.2.1. 特定应用程序存储建议 60
10.2.1.1. Registry 60
10.2.1.2. 扩展的 registry 61
10.2.1.3. 指标 61
10.2.1.4. 日志记录 61
10.2.1.5. 应用程序 61
10.2.2. 其他特定的应用程序存储建议 62
10.3. 数据存储管理 62

第 11 章 优化路由 63
11.1. INGRESS CONTROLLER（ROUTER）性能的基线 63
11.2. INGRESS CONTROLLER（路由器）性能优化 64

第 12 章 优化网络 65
12.1. 为您的网络优化 MTU 65
12.2. 安装大型集群的实践建议 65
12.3. IPSEC 的影响 66

第 13 章 管理裸机主机 67
13.1. 关于裸机主机和节点 67
13.2. 维护裸机主机 67
13.2.1. 使用 web 控制台在集群中添加裸机主机 67
13.2.2. 在 web 控制台中使用 YAML 在集群中添加裸机主机 68
13.2.3. 自动将机器扩展到可用的裸机主机数量 69

第 14 章 巨页的作用及应用程序如何使用它们 70
14.1. 巨页的作用 70
14.2. 应用程序如何使用巨页 70
14.3. 配置巨页 71
14.3.1. 在引导时 71

第 15 章 低延迟节点的 PERFORMANCE ADDON OPERATOR 74
15.1. 了解低延迟 74
15.2. 安装 PERFORMANCE ADDON OPERATOR 74
15.2.1. 使用 CLI 安装 Operator 74
15.2.2. 使用 Web 控制台安装 Performance Addon Operator 76
15.3. 升级 PERFORMANCE ADDON OPERATOR 76
15.3.1. 关于升级 Performance Addon Operator 77
15.3.1.1. Performance Addon Operator 升级对您的集群有什么影响 77
15.3.1.2. 将 Performance Addon Operator 升级到下一个次版本 77
15.3.2. 监控升级状态 77
15.4. 置备实时和低延迟工作负载 78

15.4.1. 已知的实时限制 78
15.4.2. 使用实时功能置备 worker 79
15.4.3. 验证实时内核安装 80
15.4.4. 创建一个实时工作负载 80
15.4.5. 创建带有 Guaranteed 类 QoS 类的 pod 81
15.4.6. 可选：禁用 DPDK 的 CPU 负载均衡 82
15.4.7. 分配适当的节点选择器 82
15.4.8. 将工作负载调度到具有实时功能的 worker 83
15.5. 配置巨页 83
15.6. 分配多个巨页大小 84
15.7. 为 INFRA 和应用程序容器限制 CPU 84
15.8. 使用性能配置集调整节点以实现低延迟 86
15.9. 为平台验证进行端到端测试 87
15.9.1. 先决条件 87
15.9.2. 运行测试 88
15.9.3. 镜像参数 88
15.9.3.1. Ginkgo 参数 89
15.9.3.2. 可用功能 89
15.9.4. 空运行 89
15.9.5. 断开连接的模式 89
15.9.5.1. 将镜像镜像(mirror)到集群可访问的自定义 registry 89
15.9.5.2. 指示测试使用来自自定义 registry 中的镜像 90
15.9.5.3. 镜像到集群内部 registry 90
15.9.5.4. 对不同的镜像集进行镜像(mirror) 91
15.9.6. 发现模式 91
15.9.6.1. 所需的环境配置先决条件 92
15.9.6.2. 限制测试过程中使用的节点 93
15.9.6.3. 使用单个性能配置集 93
15.9.6.4. 禁用性能配置集清理 93
15.9.7. 故障排除 94
15.9.8. 测试报告 94
15.9.8.1. JUnit 测试输出 94
15.9.8.2. 测试失败报告 94
15.9.8.3. podman 备注 94
15.9.8.4. 在 OpenShift Container Platform 4.4 中运行 94
15.9.8.5. 使用单个性能配置集 95
15.9.9. 对集群的影响 95
15.9.9.1. SCTP 95
15.9.9.2. SR-IOV 95
15.9.9.3. PTP 96
15.9.9.4. 性能 96
15.9.9.5. DPDK 96
15.9.9.6. 清理 96
15.10. 调试低延迟 CNF 调整状态 96
15.10.1. 机器配置池 97
15.11. 为红帽支持收集调试数据延迟 98
15.11.1. 关于 must-gather 工具 98
15.11.2. 关于收集低延迟数据 99
15.11.3. 收集有关特定功能的数据 99

第 16 章 使用 INTEL FPGA PAC N3000 和 INTEL VRAN DEDICATED 加速器 ACC100 优化数据平面性能 101
16.1. 了解 OPENSHIFT CONTAINER PLATFORM 的 INTEL 硬件加速器卡 101
Intel FPGA PAC N3000 101

vRAN Dedicated 加速器 ACC100 101
16.2. 为 INTEL FPGA PAC N3000 安装 OPENNESS OPERATOR 101
16.2.1. 使用 CLI 安装 Operator 101
16.2.2. 使用 Web 控制台为 Intel FPGA PAC N3000 Operator 安装 OpenNESS Operator 103
16.3. 为 INTEL FPGA PAC N3000 编程 OPENNESS OPERATOR 104
16.3.1. 使用 vRAN 位流编程 N3000 104
16.4. 为 WIRELESS FEC ACCELERATOR 安装 OPENNESS SR-IOV OPERATOR 110
16.4.1. 使用 CLI 为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator 110
16.4.2. 使用 web 控制台为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator 112
16.4.3. 为 Intel FPGA PAC N3000 配置 SR-IOV-FEC Operator 113
16.4.4. 为 Intel vRAN Dedicated Accelerator ACC100 配置 SR-IOV-FEC Operator 120
16.4.5. 在 OpenNESS 上验证应用程序 pod 访问和 FPGA 使用情况 127
16.5. 其它资源 131

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。

重要

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

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
```

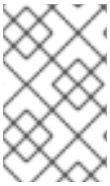
将 **podsPerCore** 设置为 **0** 可禁用这个限制。默认为 **0**。**podsPerCore** 不能超过 **maxPods**。

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。



注意

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

流程

1. 查看您可以选择的可用机器配置对象：

```
$ oc get machineconfig
```

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

2. 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. 为带有标签的 worker 更新机器配置池：

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

b. 创建 **KubeletConfig** 对象：

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

c. 验证 **KubeletConfig** 对象是否已创建：

```
$ oc get kubeletconfig
```

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. 编辑 **worker** 机器配置池：

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

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

在具有三个 master 或 control plane 节点的大型高密度集群中，当其中一个节点停止、重启或失败时，CPU 和内存用量将会激增。故障可能是因为电源、网络或底层基础架构出现意外问题，除了在关闭集群后重启集群以节约成本的情况下。其余两个 control plane 节点必须处理负载才能高度可用，从而增加资源使用量。另外，在升级过程中还会有这个预期，因为 master 被封锁、排空并按顺序重新引导，以应用操作系统更新以及 control plane Operator 更新。为了避免级联失败，请将 control plane 节点上的总体 CPU 和内存资源使用量保留为最多 60% 的所有可用容量，以处理资源使用量激增。相应地增加 control plane 节点上的 CPU 和内存，以避免因为缺少资源而造成潜在的停机。

重要

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

重要

如果使用安装程序置备的基础架构安装方法，则无法修改正在运行的 OpenShift Container Platform 4.6 集群中的 control plane 节点大小。反之，您必须估计节点总数并在安装过程中使用推荐的 control plane 节点大小。

重要

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

注意

在 OpenShift Container Platform 4.6 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。

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

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

注意

建议您在增大 AWS master 实例的类别大小前备份 etcd。

先决条件

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

流程

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

其它资源

- 备份 etcd

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并超过空间配额，etcd 的性能将会受到影响。定期维护和碎片 etcd 释放数据存储中的空间。监控 Prometheus 以了解 etcd 指标数据，并在需要时对其进行碎片处理；否则，etcd 可能会引发一个集群范围的警报，使集群进入维护模式，以只接受键读和删除。

监控这些关键指标：

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

有关 etcd 碎片整理的更多信息，请参阅 "Defragmenting etcd data" 部分。

因为 etcd 将数据写入磁盘并在磁盘上持久化，所以其性能取决于磁盘性能。减慢来自其他进程的磁盘活动和磁盘活动可能会导致长时间的 fsync 延迟。这些延迟可能会导致 etcd 丢失心跳，不会及时向磁盘提交新的建议，并最终遇到请求超时和临时丢失问题。在由低延迟和高吞吐量的 SSD 或 NVMe 磁盘支持的机器上运行 etcd。考虑单层单元(SLC)固态驱动器(SSD)，每个内存单元提供 1 位，是可靠，非常适合于写密集型工作负载。

在部署的 OpenShift Container Platform 集群上监控的一些关键指标是日志持续时间之前的 etcd 磁盘写入的 p99 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** 指标报告 etcd 磁盘 fsync 持续时间。
- **etcd_server_leader_changes_seen_total** 指标报告领导更改。
- 要排除一个较慢的磁盘并确认磁盘的速度合理，请验证 **etcd_disk_wal_fsync_duration_seconds_bucket** 的 99 百分比是否小于 10 ms。

要在创建 OpenShift Container Platform 集群之前或之后验证 etcd 的硬件，您可以使用名为 fio 的 I/O 基准测试工具。

先决条件

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

流程

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

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

  - 如果使用 Docker，请运行以下命令：

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

输出会报告磁盘是否足够快来托管 etcd，它会比较从运行中捕获的 fsync 指标的 p99，以查看它是否小于 10 ms。

因为 etcd 在所有成员间复制请求，所以其性能会严重依赖于网络输入/输出(I/O)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，这会导致对集群造成破坏的领导选举机制。在部署的 OpenShift Container Platform 集群上监控的一个关键指标是每个 etcd 集群成员上的 etcd 网络对延迟的 p99 百分比。使用 Prometheus 跟踪指标数据。

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

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 -v quorum-guard | grep etcd
```

输出示例

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

   b. 选择 pod 并运行以下命令来确定哪个 etcd 成员是领导：

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

输出示例

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

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

2. 清理 etcd 成员。

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

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

   b. 取消设置 **ETCDCTL_ENDPOINTS** 环境变量：

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

   c. 清理 etcd 成员：

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

   输出示例

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

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

   d. 验证数据库大小是否已缩小：

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

   输出示例

   ```
   +--------------------------+-----------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
   |         ENDPOINT         |        ID       | VERSION | DB SIZE | IS LEADER | IS LEARNER | RAFT TERM | RAFT INDEX | RAFT APPLIED INDEX | ERRORS |
   +--------------------------+-----------------+---------+---------+-----------+------------+-----------+------------+--------------------+--------+
   ```

```
| https://10.0.191.37:2379 | 251cd44483d811c3 |   3.4.9 |  104 MB |     false |      false |
7 |     91624 |           91624 |       |
| https://10.0.159.225:2379 | 264c7c58ecbdabee |   3.4.9 |   41 MB |     false |      false |
7 |     91624 |           91624 |       | (1)
| https://10.0.199.170:2379 | 9ac311f93915cc79 |   3.4.9 |  104 MB |      true |      false |
7 |     91624 |           91624 |       |
+--------------------------+------------------+---------+---------+-----------+------------+-----------
+------------+-------------------+--------+
```

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

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

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

a. 检查是否有 **NOSPACE** 警告：

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

输出示例

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

b. 清除警告：

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

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
- 基于 HAProxy 的 Ingress Controller
- 集群指标集合或监控服务，包括监控用户定义的项目的组件
- 集群聚合日志
- 服务代理
- Red Hat Quay
- Red Hat OpenShift Container Storage
- Red Hat Advanced Cluster Manager
- Red Hat Advanced Cluster Security for Kubernetes

- Red Hat OpenShift GitOps
- Red Hat OpenShift Pipelines

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

其它资源

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

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. 修改对象的 **spec** 部分，使其类似以下 YAML：

```
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          namespaces:
          - openshift-image-registry
          topologyKey: kubernetes.io/hostname
        weight: 100
  logLevel: Normal
  managementState: Managed
  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 1 指定从 Pod 列表获得的 **<node_name>**。

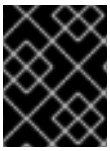
输出示例

```
NAME                          STATUS  ROLES        AGE   VERSION
ip-10-0-217-226.ec2.internal  Ready   infra,worker 17h   v1.19.0
```

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。



重要

以下建议的大小只适用于在集群安装过程中安装的 Prometheus、Router 和 Registry 基础架构组件。logging 是第二天操作，它的大小建议在最后两列中给出。

worker 节点数量	CPU 内核	内存 (GB)	带有日志的 CPU 内核	带有日志的内存 (GB)
25	4	16	4	64
100	8	32	8	128
250	16	128	16	128
500	32	128	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 4.6 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。

注意

在 OpenShift Container Platform 4.6 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。

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

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

3.1. 扩展集群的建议实践

将集群扩展到具有更多节点时：

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

注意

云供应商可能会为 API 服务实施配额。因此，需要对集群逐渐进行扩展。

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

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

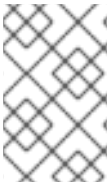
注意

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

3.2. 修改机器集

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

如果您需要在不进行其他更改的情况下扩展机器集，则不需要删除机器。



注意

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

先决条件

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

流程

1. 编辑机器集：

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

2. 将机器缩减为 **0**:

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

或者：

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

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. 指定目标池中允许同时修复的机器数量。这可设为一个百分比或一个整数。如果不健康的机器数量超过 **maxUnhealthy** 设定的限制，则不会执行补救。
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) 指定目标池中允许同时修复的机器数量。这可设为一个百分比或一个整数。如果不健康的机器数量超过 **maxUnhealthy** 设定的限制，则不会执行补救。

(8) 指定机器健康检查在决定机器不健康前必须等待节点加入集群的超时持续时间。

注意

matchLabels 只是示例; 您必须根据具体需要映射您的机器组。

3.4.1. 短路机器健康检查补救

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

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

重要

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

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

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

3.4.1.1. 使用绝对值设置 maxUnhealthy

如果将 **maxUnhealthy** 设为 **2**:

- 如果 2 个或更少节点不健康，则可执行补救

- 如果 3 个或更多节点不健康，则不会执行补救

这些值与机器健康检查要检查的机器数量无关。

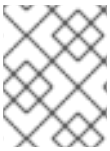
3.4.1.2. 使用百分比设置 maxUnhealthy

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

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

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

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



注意

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

3.5. 创建 MACHINEHEALTHCHECK 资源

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

先决条件

- 安装 **oc** 命令行界面。

流程

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

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

第 4 章 使用 NODE TUNING OPERATOR

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

4.1. 关于 NODE TUNING OPERATOR

Node Tuning Operator 可以帮助您通过编排 Tuned 守护进程来管理节点级别的性能优化。大多数高性能应用程序都需要一定程度的内核级性能优化。Node Tuning Operator 为用户提供了一个统一的、节点一级的 sysctl 管理接口，并可以根据具体用户的需要灵活地添加自定义性能优化设置。

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

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

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

4.2. 访问 NODE TUNING OPERATOR 示例规格

使用此流程来访问 Node Tuning Operator 的示例规格。

流程

1. 运行：

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

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

警告

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

4.3. 在集群中设置默认配置集

以下是在集群中设置的默认配置集。

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

```
  name: default
  namespace: openshift-cluster-node-tuning-operator
spec:
  profile:
  - name: "openshift"
    data: |
      [main]
      summary=Optimize systems running OpenShift (parent profile)
      include=${f:virt_check:virtual-guest:throughput-performance}

      [selinux]
      avc_cache_threshold=8192

      [net]
      nf_conntrack_hashsize=131072

      [sysctl]
      net.ipv4.ip_forward=1
      kernel.pid_max=>4194304
      net.netfilter.nf_conntrack_max=1048576
      net.ipv4.conf.all.arp_announce=2
      net.ipv4.neigh.default.gc_thresh1=8192
      net.ipv4.neigh.default.gc_thresh2=32768
      net.ipv4.neigh.default.gc_thresh3=65536
      net.ipv6.neigh.default.gc_thresh1=8192
      net.ipv6.neigh.default.gc_thresh2=32768
      net.ipv6.neigh.default.gc_thresh3=65536
      vm.max_map_count=262144

      [sysfs]
      /sys/module/nvme_core/parameters/io_timeout=4294967295
      /sys/module/nvme_core/parameters/max_retries=10

  - name: "openshift-control-plane"
    data: |
      [main]
      summary=Optimize systems running OpenShift control plane
      include=openshift

      [sysctl]
      # ktune sysctl settings, maximizing i/o throughput
      #
      # Minimal preemption granularity for CPU-bound tasks:
      # (default: 1 msec#  (1 + ilog(ncpus)), units: nanoseconds)
      kernel.sched_min_granularity_ns=10000000
      # The total time the scheduler will consider a migrated process
      # "cache hot" and thus less likely to be re-migrated
      # (system default is 500000, i.e. 0.5 ms)
      kernel.sched_migration_cost_ns=5000000
      # SCHED_OTHER wake-up granularity.
      #
      # Preemption granularity when tasks wake up.  Lower the value to
      # improve wake-up latency and throughput for latency critical tasks.
      kernel.sched_wakeup_granularity_ns=4000000

  - name: "openshift-node"
```

```
    data: |
      [main]
      summary=Optimize systems running OpenShift nodes
      include=openshift

      [sysctl]
      net.ipv4.tcp_fastopen=3
      fs.inotify.max_user_watches=65536
      fs.inotify.max_user_instances=8192

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

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

4.4. 验证是否应用了 TUNED 配置集

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。

管理状态

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

- Managed: Operator 会在配置资源更新时更新其操作对象
- Unmanaged: Operator 将忽略配置资源的更改

- Removed: Operator 将移除 Operator 置备的操作对象和资源

配置集数据

profile: 部分列出了 Tuned 配置集及其名称。

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

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

# ...

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

    # tuned_profile_n profile settings
```

建议的配置集

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

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

列表中的独立项：

```
- machineConfigLabels: 1
    <mcLabels> 2
  match: 3
    <match> 4
  priority: <priority> 5
  profile: <tuned_profile_name> 6
```

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。

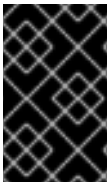
3 可选的对象类型（**node** 或 **pod**）。如果省略，会使用 **node**。

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，并在分配了找到的机器配置池的所有节点上设置配置集 **<tuned_profile_name>**。要针对同时具有 master 和 worker 角色的节点，您必须使用 master 角色。

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



重要

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

示例：基于节点或 pod 标签的匹配

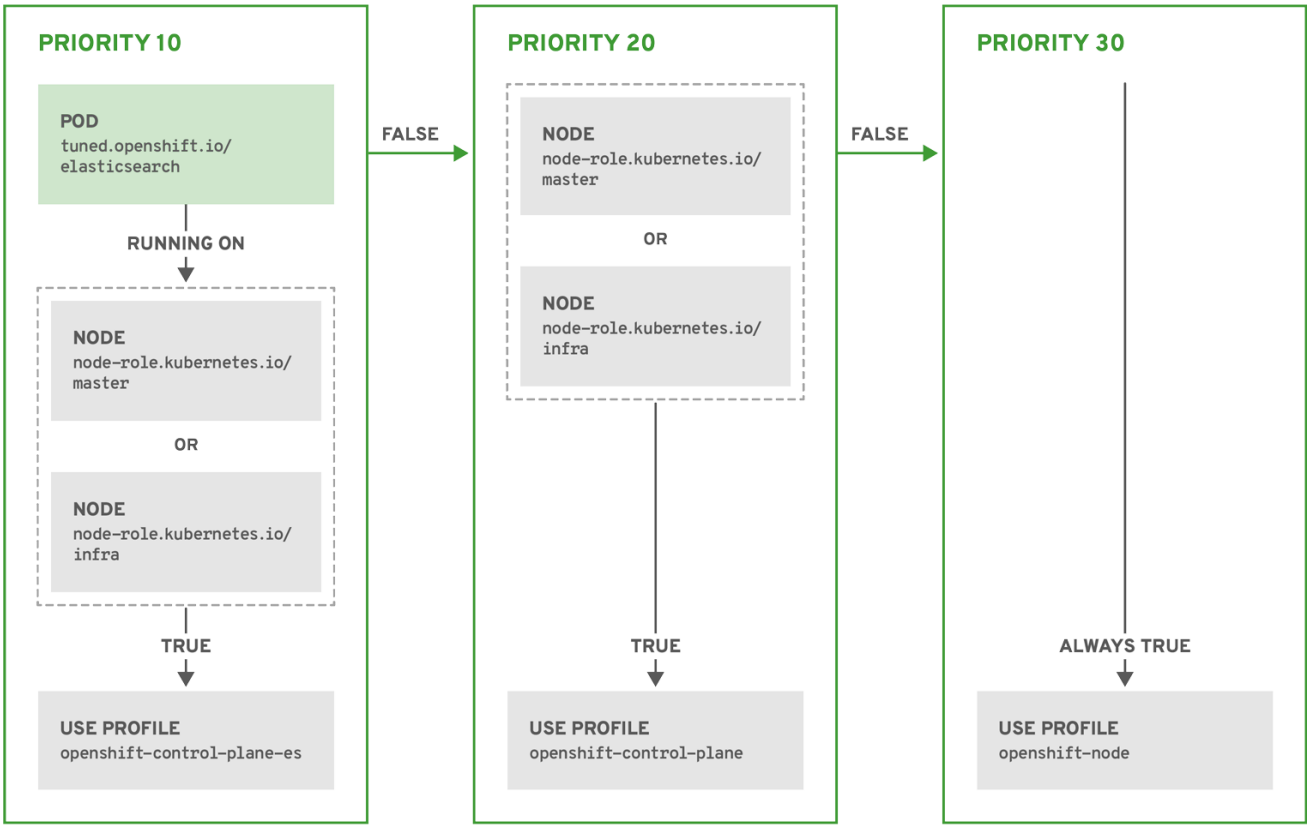
```
- match:
  - label: tuned.openshift.io/elasticsearch
    match:
    - label: node-role.kubernetes.io/master
    - label: node-role.kubernetes.io/infra
    type: pod
  priority: 10
  profile: openshift-control-plane-es
- match:
  - label: node-role.kubernetes.io/master
  - label: node-role.kubernetes.io/infra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

```
    profile: openshift-control-plane
  - priority: 30
    profile: openshift-node
```

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

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

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



OPENSHIFT_10_0319

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
```

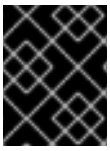
为尽量减少节点的重新引导情况，为目标节点添加机器配置池将匹配的节点选择器标签，然后创建上述 Tuned CR，最后创建自定义机器配置池。

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:4.6
```

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:4.6 /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:4.6 \
/bin/bash -c 'KUBECONFIG=/root/.kube/config VIPERCONFIG=/root/configs/test.yaml \
openshift-tests run-test "[sig-scalability][Feature:Performance] Load cluster \
should populate the cluster [Slow][Serial] [Suite:openshift]"'
```

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

5.3. 配置 CLUSTER LOADER

该工具创建多个命名空间（项目），其中包含多个模板或 pod。

5.3.1. Cluster Loader 配置文件示例

Cluster Loader 的配置文件是一个基本的 YAML 文件：

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

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

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

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

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

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

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

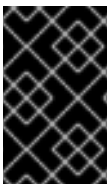
(2)

调整集允许速率限制和分步，可以生成几批 pod，同时在两组间暂停使用。在继续执行前，Cluster Loader 会监控上一步的完成情况。

3 为每 **N** 个对象被创建后，会暂停 **M** 秒。

4 在创建不同对象期间，限制率会等待 **M** 毫秒。

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



重要

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

5.3.2. 配置字段

表 5.1. 顶层 Cluster Loader 字段

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

表 5.2. projects 下的字段

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

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

表 5.3. pods 和 templates 下的字段

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

表 5.4. tuningsets 下的字段

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

表 5.5. tuningsets pods 或 tuningsets templates 下的字段

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

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

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

表 5.7. **sync 下的字段**

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

5.4. 已知问题

- 当在没有配置的情况下调用 Cluster Loader 会失败。(BZ#1761925)

- 如果用户模板中没有定义 **IDENTIFIER** 参数，则模板创建失败，错误信息为：**error: unknown parameter name "IDENTIFIER"**。如果部署模板，在模板中添加这个参数以避免出现这个错误：

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

  如果部署 pod，则不需要添加该参数。

第 6 章 使用 CPU MANAGER

CPU Manager 管理 CPU 组并限制特定 CPU 的负载。

CPU Manager 对于有以下属性的负载有用：

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

6.1. 设置 CPU MANAGER

流程

1. 可选：标记节点：

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

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

```
# oc edit machineconfigpool worker
```

3. 为 worker 机器配置池添加标签：

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

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

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

1 指定一个策略：

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

2 可选。指定 CPU Manager 协调频率。默认值为 **5s**。

5. 创建动态 kubelet 配置：

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

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

6. 检查合并的 kubelet 配置：

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

输出示例

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

7. 检查 worker 是否有更新的 **kubelet.conf**：

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

输出示例

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

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

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

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

输出示例

```
apiVersion: v1
```

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

9. 创建 pod：

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

10. 确定为您标记的节点调度了 pod：

```
# oc describe pod cpumanager
```

输出示例

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

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

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

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

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

输出示例

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

12. 检查任务允许的 CPU 列表：

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

输出示例

```
 Cpus_allowed_list:    1
```

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

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

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

输出示例

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

Allocated resources:
```

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

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

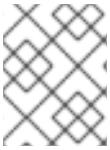
```
NAME                    READY   STATUS    RESTARTS   AGE
cpumanager-6cqz7        1/1     Running   0          33m
cpumanager-7qc2t        0/1     Pending   0          11s
```

第 7 章 使用拓扑管理器

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

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

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



注意

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

7.1. 拓扑管理器策略

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

注意

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

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

none 策略

这是默认策略，不执行任何拓扑对齐调整。

best-effort 策略

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

restricted 策略

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

single-numa-node 策略

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

7.2. 设置拓扑管理器

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

先决条件

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

流程

激活 Topolgy Manager:

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

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

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

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

其他资源

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

7.3. POD 与拓扑管理器策略的交互

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

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

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

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

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

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

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

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

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

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

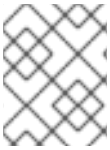
拓扑管理器将使用此信息存储该容器的最佳拓扑。在本 pod 中，CPU Manager 和设备管理器将在资源分配阶段使用此存储的信息。

第 8 章 扩展 CLUSTER MONITORING OPERATOR

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

8.1. PROMETHEUS 数据库存储要求

红帽对不同的扩展大小进行了各种测试。



注意

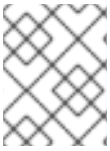
以下 Prometheus 存储要求并不具有规定性。取决于工作负载活动和资源使用情况，集群中可能会观察到更高资源消耗。

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

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

大约 20%的预期大小被添加为开销，以保证存储要求不会超过计算的值。

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



注意

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

针对 OpenShift Container Platform 的建议

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

8.2. 配置集群监控

您可以为集群监控堆栈中的 Prometheus 组件增加存储容量。

流程

为 Prometheus 增加存储容量：

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

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

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

2 4. 集群的存储类。

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

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

2. 为保留周期、存储类和存储大小添加值。

3. 保存该文件。

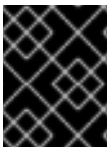
4. 运行以下命令应用这些更改：

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

第 9 章 根据对象限制规划您的环境

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

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



重要

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

在大多数情况下，超过这些限制会降低整体性能。它不一定意味着集群会出现错误。

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
每个 Ingress Controller 的路由和后端数量	每个路由器有 2,000 个	每个路由器有 2,000 个
secret 的数量	80,000	80,000
配置映射数量	90,000	90,000
服务数量 [5]	10,000	10,000
每个命名空间的服务数	5,000	5,000

最大类型	3.x 测试的最大值	4.x 测试的最大值
每个服务中的后端数	5,000	5,000
每个命名空间的部署数量 [4]	2,000	2,000
构建配置数	12,000	12,000
secret 的数量	40,000	40,000
自定义资源定义(CRD)的数量	没有默认值。	512 [6]

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

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

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

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

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

注意

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

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

AWS 云平台：

节点	Flavor	vCPU	RAM(GiB)	磁盘类型	磁盘大小(GiB)/IOS	数量	区域
Master/etcd [1]	r5.4xlarge	16	128	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。

9.3. 如何根据经过测试的集群限制规划您的环境

重要

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

某些限制只在单一维度中扩展。当很多对象在集群中运行时，它们会有所不同。

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

在规划您的环境时，请确定每个节点会运行多少 个 pod ：

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

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

示例情境

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

9.4. 如何根据应用程序要求规划您的环境

考虑应用程序环境示例：

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

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

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

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

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

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

-

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

```
    selector:
      name: replicationcontroller${IDENTIFIER}
    ports:
    - name: serviceport${IDENTIFIER}
      protocol: TCP
      port: 80
      targetPort: 8080
    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: deployment-config-template
```

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

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

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

第 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。

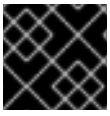
10.1. 可用的持久性存储选项

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] 和供应商 NFS
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。	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 4.6 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	Metrics[3]	日志记录	Apps
Block	Yes[4]	否	可配置	无法配置	推荐的	推荐的	推荐的
File	Yes[4]	是	可配置	可配置	Configurable[5]	Configurable[6]	推荐的
对象	是	是	推荐的	推荐的	无法配置	无法配置	Not configurable[7]

[1] **ReadOnlyMany**

[2] **ReadWriteMany**

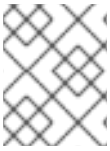
[3] **Prometheus 是用于指标数据的底层技术。**

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

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

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

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



注意

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

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，不建议使用 RHEL NFS 作为 PV 后端用于核心服务。

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

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

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。
- 对象存储是不可配置的。

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

重要

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

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

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/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、网络接口控制器 (NIC) 卸载、多队列和 ethtool 设置来调节。

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

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

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

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

如果您希望超过 Gbps，可以：

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

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

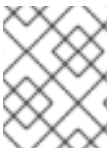
12.1. 为您的网络优化 MTU

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

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

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

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



注意

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

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 管理裸机主机

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

13.1. 关于裸机主机和节点

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

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

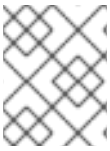
13.2. 维护裸机主机

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

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

您可以在 web 控制台中取消置备裸机主机。取消置备主机执行以下操作：

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



注意

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

其它资源

- 在裸机中添加计算机器

13.2.1. 使用 web 控制台在集群中添加裸机主机

您可以在 web 控制台中在集群中添加裸机主机。

先决条件

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

流程

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

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

注意

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

13.2.2. 在 web 控制台中使用 YAML 在集群中添加裸机主机

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

先决条件

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

流程

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

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

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

4. 选择 **Create** 以保存 YAML 并创建新的裸机主机。

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

注意

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

13.2.3. 自动将机器扩展到可用的裸机主机数量

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

先决条件

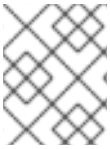
- 安装 RHCOS 裸机计算机器以在集群中使用，并创建对应的 **BareMetalHost** 对象。
- 安装 OpenShift Container Platform CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录。

流程

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

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

等待新的缩放计算机启动。



注意

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

其它资源

- 扩展集群
- 裸机上的 MachineHealthCheck

第 14 章 巨页的作用及应用程序如何使用它们

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。

14.2. 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

14.3. 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。

14.3.1. 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。

第 15 章 低延迟节点的 PERFORMANCE ADDON OPERATOR

15.1. 了解低延迟

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

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

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

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

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

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

OpenShift Container Platform 提供了一个 Performance Addon Operator 来实现自动性能优化，从而实现 OpenShift 应用程序的低延迟性能。集群管理员使用此性能配置集配置，这有助于以更可靠的方式进行更改。管理员可以指定是否要将内核更新至 kernel-rt，保留用于托管的 CPU，以及用于运行工作负载的 CPU。

15.2. 安装 PERFORMANCE ADDON OPERATOR

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

15.2.1. 使用 CLI 安装 Operator

作为集群管理员，您可以使用 CLI 安装 Operator。

先决条件

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

流程

1. 通过完成以下操作，为 Performance Addon Operator 创建命名空间：

a. 创建用于定义 **openshift-performance-addon-operator** 命名空间的以下 Namespace 自定义资源（CR），然后在 **pao-namespace.yaml** 文件中保存 YAML：

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-performance-addon-operator
```

b. 运行以下命令创建命名空间：

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

2. 通过创建以下对象，在您上一步创建的命名空间中安装 Performance Addon Operator:

a. 创建以下 **OperatorGroup** CR，并在 **pao-operatorgroup.yaml** 文件中保存 YAML:

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

b. 运行以下命令来创建 **OperatorGroup** CR:

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

c. 运行以下命令获取下一步所需的 **channel** 值。

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

输出示例

```
4.6
```

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

订阅示例

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

1. 使用在前一步中获得的值指定 **.status.defaultChannel** 参数。
2. 您必须指定 **redhat-operators** 值。

e. 运行以下命令创建订阅对象：

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

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

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

15.2.2. 使用 Web 控制台安装 Performance Addon Operator

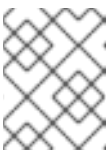
作为集群管理员，您可以使用 web 控制台安装 Performance Addon Operator。

注意

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

流程

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



注意

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

如果 Operator 没有被安装，您可以进一步进行故障排除：

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

15.3. 升级 PERFORMANCE ADDON OPERATOR

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

15.3.1. 关于升级 Performance Addon Operator

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

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

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

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

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

先决条件

- 使用具有 cluster-admin 角色的用户访问集群。

流程

1. 访问 OpenShift Web 控制台，导航到 **Operators → Installed Operators**。
2. 点击**Performance Addon Operator** 打开 **Operator Details** 页面。
3. 点 **Subscription** 标签页打开 **Subscription Overview** 页。
4. 在 **Channel** 窗格中，点击版本号右侧的铅笔图标打开 **Change Subscription Update Channel**窗口。
5. 选择下一个次要版本。例如，如果要升级到 Performance Addon Operator 4.6，选择 **4.6**。
6. 点 **Save**。
7. 通过导航到 **Operators → Installed Operators** 来检查升级的状态。您还可以通过运行以下 **oc** 命令来检查状态：

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

15.3.2. 监控升级状态

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

注意

PHASE 和状况值均是基于可用信息的近似值。

先决条件

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

流程

1. 运行以下命令：

```
$ oc get csv
```

2. 查看输出，检查 **PHASE** 字段。例如：

```
VERSION    REPLACES                                PHASE
4.6.0      performance-addon-operator.v4.5.0       Installing
4.5.0                                              Replacing
```

3. 再次运行 **get csv** 以验证输出：

```
# oc get csv
```

输出示例

```
NAME                         DISPLAY                  VERSION   REPLACES
PHASE
performance-addon-operator.v4.5.0   Performance Addon Operator   4.6.0     performance-
addon-operator.v4.5.0   Succeeded
```

15.4. 置备实时和低延迟工作负载

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

集群管理员使用此性能配置集配置，这有助于以更可靠的方式进行更改。管理员可以指定是否将内核更新至 kernel-rt（实时）、用于处理维护任务的 CPU，以及用于运行工作负载的 CPU。

警告

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

15.4.1. 已知的实时限制

注意

RT 内核只在 worker 节点上被支持。

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

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

注意

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

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

15.4.2. 使用实时功能置备 worker

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

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

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

5. 使用节点角色标签将节点添加到正确的机器配置池。

注意

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

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

```
apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
 name: example-performanceprofile
spec:
...
  realTimeKernel:
    enabled: true
  nodeSelector:
     node-role.kubernetes.io/worker-rt: ""
  machineConfigPoolSelector:
     machineconfiguration.openshift.io/role: worker-rt
```

8. 验证匹配的机器配置池是否存在一个标签：

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

输出示例

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

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

15.4.3. 验证实时内核安装

使用这个命令确定安装了实时内核：

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

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

```
NAME                         STATUS   ROLES          AGE  VERSION                INTERNAL-IP
EXTERNAL-IP   OS-IMAGE                                KERNEL-VERSION
CONTAINER-RUNTIME
rt-worker-0.example.com           Ready  worker,worker-rt   5d17h   v1.22.1
128.66.135.107   <none>           Red Hat Enterprise Linux CoreOS 46.82.202008252340-0 (Ootpa)
4.18.0-211.rt5.23.el8.x86_64   cri-o://1.19.0-90.rhaos4.6.git4a0ac05.el8-rc.1
[...]
```

15.4.4. 创建一个实时工作负载

使用以下步骤准备一个使用实时功能的工作负载。

流程

1. 创建带有 **Guaranteed** 类 QoS 类的 pod。
2. 可选：禁用 DPDK 的 CPU 负载均衡。
3. 分配正确的节点选择器。

在编写应用程序时，请遵循应用程序调整和部署中的常规建议。

15.4.5. 创建带有 Guaranteed 类 QoS 类的 pod

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

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

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

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

1. 创建 pod：

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

2. 查看有关 pod 的详细信息：

```
$ oc get pod qos-demo --namespace=qos-example --output=yaml
```

输出示例

```
spec:
  containers:
    ...
status:
  qosClass: Guaranteed
```

注意

如果容器指定了自己的内存限值，但没有指定内存请求，OpenShift Container Platform 会自动分配与限制匹配的内存请求。同样，如果容器指定了自己的 CPU 限值，但没有指定 CPU 请求，OpenShift Container Platform 会自动分配与限制匹配的 CPU 请求。

15.4.6. 可选：禁用 DPDK 的 CPU 负载均衡

禁用或启用 CPU 负载均衡的功能在 CRI-O 级别实现。CRI-O 下的代码仅在满足以下要求时禁用或启用 CPU 负载均衡。

- pod 必须使用 **performance-<profile-name>** 运行时类。您可以通过查看性能配置集的状态来获得正确的名称，如下所示：

```
apiVersion: performance.openshift.io/v1
kind: PerformanceProfile
...
status:
  ...
  runtimeClass: performance-manual
```

- pod 必须具有 **cpu-load-balancing.crio.io: true** 注解。

Performance Addon Operator 负责在相关节点下创建高性能运行时处理器配置片断，并在集群下创建高性能运行时类。它具有与默认运行时处理相同的内容，但它启用了 CPU 负载均衡配置功能。

要禁用 pod 的 CPU 负载均衡，**Pod** 规格必须包括以下字段：

```
apiVersion: v1
kind: Pod
metadata:
  ...
  annotations:
    ...
    cpu-load-balancing.crio.io: "true"
    ...
  ...
spec:
  ...
  runtimeClassName: performance-<profile_name>
  ...
```

注意

仅在启用了 CPU 管理器静态策略，以及带有保证 QoS 使用整个 CPU 的 pod 时，禁用 CPU 负载均衡。否则，禁用 CPU 负载均衡会影响集群中其他容器的性能。

15.4.7. 分配适当的节点选择器

为节点分配 pod 的首选方法是使用与性能配置集相同的节点选择器，如下所示：

```
apiVersion: v1
kind: Pod
```

```
metadata:
  name: example
spec:
  # ...
  nodeSelector:
    node-role.kubernetes.io/worker-rt: ""
```

如需更多信息，请参阅使用节点选择器将 pod 放置到特定的节点上。

15.4.8. 将工作负载调度到具有实时功能的 worker

使用与附加到机器配置的节点匹配的标签选择器，这些选择器是为低延迟配置的。如需更多信息，请参阅将 pod 分配给节点。

15.5. 配置巨页

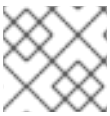
节点必须预先分配在 OpenShift Container Platform 集群中使用的巨页。使用 Performance Addon Operator 在指定节点上分配巨页。

OpenShift Container Platform 提供了创建和分配巨页的方法。Performance Addon Operator 提供了一种更易于使用性能配置集的方法。

例如，在性能配置集的 **hugepages pages** 部分，您可以指定多个块的 **size**、**count** 以及可选的 **node**:

```
hugepages:
  defaultHugepagesSize: "1G"
  pages:
  - size:  "1G"
    count:  4
    node:  0 (1)
```

(1) **node** 是分配巨页的 NUMA 节点。如果省略了 **node**，该页面将平均分布在所有 NUMA 节点中。



注意

等待显示更新已完成的相关机器配置池状态。

这些是分配巨页的唯一配置步骤。

验证

- 要验证配置，请查看节点上的 **/proc/meminfo** 文件：

```
$ oc debug node/ip-10-0-141-105.ec2.internal
```

```
# grep -i huge /proc/meminfo
```

输出示例

```
AnonHugePages:    ###### ##
ShmemHugePages:        0 kB
```

```
HugePages_Total:       2
HugePages_Free:        2
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       #### ##
Hugetlb:            #### ##
```

- 使用 **oc describe** 报告新大小：

```
$ oc describe node worker-0.ocp4poc.example.com | grep -i huge
```

输出示例

```
                            hugepages-1g=true
hugepages-###:  ###
hugepages-###:  ###
```

15.6. 分配多个巨页大小

您可以在同一容器下请求具有不同大小的巨页。这样，您可以定义由具有不同巨页大小的容器组成的更复杂的 pod。

例如，您可以把大小定义为 **1G** 和 **2M**，Performance Addon Operator 会在节点上配置这两个大小，如下所示：

```
spec:
  hugepages:
    defaultHugepagesSize: 1G
    pages:
    - count: 1024
      node: 0
      size: 2M
    - count: 4
      node: 1
      size: 1G
```

15.7. 为 INFRA 和应用程序容器限制 CPU

通用内务处理和工作负载任务使用 CPU 的方式可能会影响对延迟敏感的进程。默认情况下，容器运行时使用所有在线 CPU 一起运行所有容器，这可能导致上下文切换和延迟激增。对 CPU 进行分区可防止无状态进程通过相互分离来干扰对延迟敏感的进程。下表描述了在使用 Performance Addon Operator 调整节点后的进程如何在 CPU 上运行：

表 15.1. 进程的 CPU 分配

进程类型	详情
Burstable 和 **BestEffort** pod	在除了运行低延迟工作负载外的任意 CPU 上运行
基础架构 pod	在除了运行低延迟工作负载外的任意 CPU 上运行

进程类型	详情
中断	重定向至保留的 CPU（在 OpenShift Container Platform 4.6 及更新的版本中可选）
内核进程	固定保留的 CPU
对延迟敏感的工作负载 pod	固定到隔离池中的特定专用 CPU
OS 进程/systemd 服务	固定保留的 CPU

对于所有 QoS 进程类型（**Burstable**、**BestEffort** 或 **Guaranteed**）的 pod 的 pod 的可分配容量等于隔离池的容量。保留池的容量已从节点的总内核容量中删除，供集群和操作系统日常任务使用。

示例 1

节点具有 100 个内核的容量。通过使用性能配置集，集群管理员将 50 个内核分配给隔离池，将 50 个内核分配给保留池。集群管理员为 **BestEffort** 或 **Burstable** pod 为 QoS **Guaranteed** pod 和 25 个内核分配 25 个内核。这与隔离池的容量匹配。

示例 2

节点具有 100 个内核的容量。通过使用性能配置集，集群管理员将 50 个内核分配给隔离池，将 50 个内核分配给保留池。集群管理员为 QoS **Guaranteed** pod 分配 50 个内核，并为 **BestEffort** 或 **Burstable** pod 分配一个核心。这会由一个内核超过隔离池的容量。Pod 调度因为 CPU 容量不足而失败。

使用的确切分区模式取决于许多因素，如硬件、工作负载特性和预期的系统负载。以下是一些用例示例：

- 如果对延迟敏感的工作负载使用特定的硬件，如网络接口卡(NIC)，请确保隔离池中的 CPU 与这个硬件尽可能接近。至少，您应该将工作负载放在同一个非统一内存访问 (NUMA) 节点中。
- 保留的池用于处理所有中断。根据系统网络，分配一个足够大小的保留池来处理所有传入的数据包中断。在 4.6 及更新的版本中，可以选择性地将工作负载标记为敏感。在决定哪些特定 CPU 用于保留和隔离分区时，需要详细分析和测量。设备和内存的 NUMA 紧密度等因素扮演了角色。选择也取决于工作负载架构和具体的用例。

重要

保留和隔离的 CPU 池不得重叠，并且必须一起跨越 worker 节点中的所有可用内核。

为确保内务处理任务和工作负载不会相互干扰，请在性能配置集的 **spec** 部分指定两组 CPU。

- **isolated** - 指定应用程序容器工作负载的 CPU。在这些 CPU 上运行的工作负载遇到最低的延迟，以及零中断，例如达到高零数据包丢失带宽。
- **reserved** - 为集群和操作系统日常任务指定 CPU。 **reserved** 组中的线程经常会比较繁忙。不要在 **reserved** 组中运行对延迟敏感的应用程序。延迟敏感的应用程序 **在隔离组中运行**。
Procedure

1. 创建适合环境硬件和拓扑的性能配置集。
2. 使用您想要为 infra 和应用程序容器保留和隔离的 CPU 添加 **reserved** 和 **isolated** 参数：

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

```
kind: PerformanceProfile
metadata:
  name: infra-cpus
spec:
  cpu:
    reserved: "0-4,9" (1)
    isolated: "5-8" (2)
  nodeSelector: (3)
    node-role.kubernetes.io/worker: ""
```

1. 指定 infra 容器用于执行集群和操作系统日常任务的 CPU。
2. 指定应用程序容器运行工作负载的 CPU。
3. 指定节点选择器，以将性能配置集应用到特定的节点。

15.8. 使用性能配置集调整节点以实现低延迟

性能配置集可让您控制属于特定机器配置池的节点的延迟调整方面。指定设置后，**PerformanceProfile** 对象将编译为执行实际节点级别调整的多个对象：

- 操作节点的 **MachineConfig** 文件。
- 用于配置拓扑管理器、CPU Manager 和 OpenShift Container Platform 节点的 **KubeletConfig** 文件。
- 配置 Node Tuning Operator 的 Tuned 配置集。

流程

1. 准备集群。
2. 创建机器配置池。
3. 安装 Performance Addon Operator。
4. 创建一个适合您的硬件和拓扑的性能配置集。在性能配置集中，您可以指定是否要将内核更新至 kernel-rt,分配巨页，保留用于操作系统日常任务处理进程的 CPU，以及用于运行工作负载的 CPU。
   这是一个典型的性能配置集：

```
apiVersion: performance.openshift.io/v1
kind: PerformanceProfile
metadata:
 name: performance
spec:
 cpu:
  isolated: "5-15"
  reserved: "0-4"
 hugepages:
  defaultHugepagesSize: "1G"
  pages:
  -size: "1G"
   count: 16
```

```
    node: 0
  realTimeKernel:
    enabled: true (1)
  numa: (2)
    topologyPolicy: "best-effort"
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

1. 有效值为 **true** 或者 **false**。设置 **true** 值会在节点上安装实时内核。
2. 使用此字段配置拓扑管理器策略。有效值为 **none**（默认）、**best-effort**、**restricted** 和 **single-numa-node**。如需更多信息，请参阅拓扑管理器策略。

15.9. 为平台验证进行端到端测试

Cloud-native Network Function（CNF）测试镜像是一个容器化测试套件，用于验证运行 CNF 有效负载所需的功能。您可以使用此镜像验证启用了 CNF 的 OpenShift 集群，其中安装了运行 CNF 工作负载所需的所有组件。

由镜像运行的测试被分为三个不同的阶段：

- 简单集群验证
- 设置
- 端到端测试

验证阶段检查集群中是否正确部署了所有需要测试的功能。

验证包括：

- 划分属于要测试的机器的机器配置池
- 在节点上启用 SCTP
- 安装了 Performance Addon Operator
- 安装了 SR-IOV Operator
- 安装了 PTP Operator
- 使用 OVN kubernetes 作为 SDN

每次执行测试时都会执行环境配置。这包括创建 SR-IOV 节点策略、性能配置集或 PTP 配置集等项目。允许测试对已进行了配置的集群进行配置可能会影响集群的功能。另外，对配置项目（如 SR-IOV 节点策略）的更改可能会导致环境临时不可用，直到处理配置更改为止。

15.9.1. 先决条件

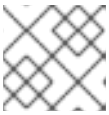
- 测试入口点是 **/usr/bin/test-run.sh**。它运行设置测试集和真实的符合度测试套件。最低要求是为其提供 kubeconfig 文件及其相关的 **$KUBECONFIG** 环境变量，并挂载在一个卷中。
- 测试假设集群中已存在指定功能（以 Operator 的方式）、在集群或机器配置中启用了相关的标记。

- 有些测试需要已存在的机器配置池将其更改附加到其中。这必须在运行测试前在集群中创建。默认 worker 池为 **worker-cnf**，可使用以下清单创建：

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

您可以使用 **ROLE_WORKER_CNF** 变量覆盖 worker 池名称：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig -e
ROLE_WORKER_CNF=custom-worker-pool registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6
/usr/bin/test-run.sh
```



注意

目前，不是所有测试都选择性地在属于该池的节点中运行。

15.9.2. 运行测试

假设该文件位于当前目录中，运行测试套件的命令为：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-run.sh
```

您的 kubeconfig 文件可以从正在运行的容器内被消耗。

15.9.3. 镜像参数

根据具体要求，测试可以使用不同的镜像。测试使用两个镜像，可使用以下环境变量进行修改：

- **CNF_TESTS_IMAGE**
- **DPDK_TESTS_IMAGE**

例如，要更改 **CNF_TESTS_IMAGE** 使用自定义 registry，请运行以下命令：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig -e
CNF_TESTS_IMAGE="custom-cnf-tests-image:latests" registry.redhat.io/openshift4/cnf-tests-
rhel8:v4.6 /usr/bin/test-run.sh
```

15.9.3.1. Ginkgo 参数

测试套件基于 ginkgo BDD 框架。这意味着，它接受可以用于过滤或跳过测试的参数。

您可以使用 **-ginkgo.focus** 参数来过滤一组测试：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-run.sh -ginkgo.focus="performance|sctp"
```

注意

有一个特定的测试需要 SR-IOV 和 SCTP。由于 **focus** 参数的选择性，这个测试会被仅放置 **sriov** 匹配器来触发。如果测试是针对安装了 SR-IOV 但不安装 SCTP 的集群执行，添加 **-ginkgo.skip=SCTP** 参数会导致测试跳过 SCTP 测试。

15.9.3.2. 可用功能

要过滤的可用功能集包括：

- **performance**
- **sriov**
- **ptp**
- **sctp**
- **dpdk**

15.9.4. 空运行

使用此命令以空运行模式运行。这可用于检查测试套件中的内容，并为镜像将要运行的所有测试提供输出。

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-run.sh -ginkgo.dryRun -ginkgo.v
```

15.9.5. 断开连接的模式

CNF 测试镜像支持在断开连接的集群中运行测试，这意味着无法访问外部 registry。这分为两个步骤：

1. 执行镜像。
2. 指示测试以使用来自自定义 registry 的镜像。

15.9.5.1. 将镜像镜像(mirror)到集群可访问的自定义 registry

mirror 中提供了镜像可执行文件，以提供 **oc** 需要的输入来镜像运行测试到本地 registry 所需的镜像。

从可通过互联网访问集群和 registry.redhat.io 的中间机器运行这个命令：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/mirror -registry my.local.registry:5000/ | oc
image mirror -f -
```

然后，按照以下部分中有关覆盖用于获取镜像的 registry 的说明进行操作。

15.9.5.2. 指示测试使用来自自定义 registry 中的镜像

这可以通过设置 **IMAGE_REGISTRY** 环境变量完成：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig -e
IMAGE_REGISTRY="my.local.registry:5000/" -e CNF_TESTS_IMAGE="custom-cnf-tests-
image:latests" registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-run.sh
```

15.9.5.3. 镜像到集群内部 registry

OpenShift Container Platform 提供了一个内建的容器镜像 registry，它作为一个标准的工作负载在集群中运行。

流程

1. 通过使用路由公开到 registry 的外部访问权限：

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster --patch '{"spec":
{"defaultRoute":true}}' --type=merge
```

2. 获取 registry 端点：

```
REGISTRY=$(oc get route default-route -n openshift-image-registry --template='{{ .spec.host
}}')
```

3. 创建用于公开镜像的命名空间：

```
$ oc create ns cnftests
```

4. 将该镜像流供用于测试的所有命名空间使用。这需要允许 test 命名空间从 **cnftests** 镜像流中获取镜像。

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:sctptest:default --
namespace=cnftests
```

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:cnf-features-
testing:default --namespace=cnftests
```

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:performance-addon-
operators-testing:default --namespace=cnftests
```

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:dpdk-testing:default
--namespace=cnftests
```

```
$ oc policy add-role-to-user system:image-puller system:serviceaccount:sriov-conformance-testing:default --namespace=cnftests
```

5. 检索 docker secret 名称和 auth 令牌：

```
SECRET=$(oc -n cnftests get secret | grep builder-docker | awk {'print $1'}
TOKEN=$(oc -n cnftests get secret $SECRET -o jsonpath="{.data['\.dockercfg']}" | base64 --decode | jq '.["image-registry.openshift-image-registry.svc:5000"].auth')
```

6. 编写类似如下的 **dockerauth.json**:

```
echo "{\"auths\": { \"$REGISTRY\": { \"auth\": $TOKEN } }}" > dockerauth.json
```

7. 进行镜像：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/mirror -registry $REGISTRY/cnftests | oc image mirror --insecure=true -a=$(pwd)/dockerauth.json -f -
```

8. 运行测试：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig -e IMAGE_REGISTRY=image-registry.openshift-image-registry.svc:5000/cnftests cnf-tests-local:latest /usr/bin/test-run.sh
```

15.9.5.4. 对不同的镜像集进行镜像(mirror)

流程

1. **mirror** 命令会默认尝试镜像 u/s 镜像。这可以通过向镜像传递带有以下格式的文件来覆盖：

```
[
  {
    "registry": "public.registry.io:5000",
    "image": "imageforcnftests:4.6"
  },
  {
    "registry": "public.registry.io:5000",
    "image": "imagefordpdk:4.6"
  }
]
```

2. 把它传递给 **mirror** 命令，例如将其在本地保存为 **images.json**。使用以下命令，本地路径挂载到容器内的 /**kubeconfig** 中，并可传递给 mirror 命令。

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/mirror --registry "my.local.registry:5000/" --images "/kubeconfig/images.json" | oc image mirror -f -
```

15.9.6. 发现模式

发现模式允许您在不更改其配置的情况下，验证集群的功能。在测试时使用现有环境配置。测试会尝试查

找所需的配置项目，并使用这些项目来执行测试。如果没有找到运行特定测试所需的资源，则会跳过测试，为用户提供正确的信息。测试完成后，不会清理预配置的配置项目，测试环境可立即用于另一个测试运行。

某些配置项目仍由测试创建。这些是运行测试所需的特定项，例如 SR-IOV 网络。这些配置项目会在自定义命名空间中创建，并在执行测试后进行清理。

一个额外的好处是可以减少测试运行时间。因为已经有配置项目，因此不需要时间进行环境配置和分配。

要启用发现模式，必须如下设置 **DISCOVERY_MODE** 环境变量来指示测试：

```
$ docker run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig -e
DISCOVERY_MODE=true registry.redhat.io/openshift-kni/cnf-tests /usr/bin/test-run.sh
```

15.9.6.1. 所需的环境配置先决条件

SR-IOV 测试

大多数 SR-IOV 测试需要以下资源：

- **SriovNetworkNodePolicy**.
- 至少有一个资源由 **SriovNetworkNodePolicy** 指定的资源为可分配的资源 ; 至少 5 个资源数才被视为足够。

有些测试有额外的要求：

- 节点上有可用策略资源的未使用设备，其链接状态为 **DOWN**，而不是桥接 slave。
- 一个 **SriovNetworkNodePolicy**，其 MTU 值为 **9000**。

DPDK 测试

与 DPDK 相关的测试需要：

- 一个性能配置集。
- 一个 SR-IOV 策略。
- 具有可用于 SR-IOV 策略的资源的节点，并使用 **PerformanceProfile** 节点选择器。

PTP 测试

- 一个从 **PtpConfig**（**ptp4lOpts="-s" ,phc2sysOpts="-a -r"**）。
- 具有与 slave **PtpConfig** 匹配标签的节点。

SCTP 测试

- **SriovNetworkNodePolicy**.
- 与 **SriovNetworkNodePolicy** 和启用 SCTP 的 **MachineConfig** 匹配的节点。

Performance Operator 测试

不同的测试有不同的要求。其中一些是：

- 一个性能配置集。
- 带有 **profile.Spec.CPU.Isolated = 1** 的性能配置集。
- 带有 **profile.Spec.RealTimeKernel.Enabled == true** 的性能配置集。
- 没有使用巨页的节点。

15.9.6.2. 限制测试过程中使用的节点

通过指定 **NODES_SELECTOR** 环境变量来限制执行测试的节点。然后，测试创建的任何资源都仅限于指定的节点。

```
$ docker run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig -e
NODES_SELECTOR=node-role.kubernetes.io/worker-cnf registry.redhat.io/openshift-kni/cnf-tests
/usr/bin/test-run.sh
```

15.9.6.3. 使用单个性能配置集

DPDK 测试所需的资源高于性能测试套件所需的资源。为加快执行速度，可使用提供 DPDK 测试套件的测试使用的性能配置集覆盖。

要做到这一点，类似以下内容的配置集可以挂载到容器中，并可指示性能测试来部署它。

```
apiVersion: performance.openshift.io/v1
kind: PerformanceProfile
metadata:
  name: performance
spec:
  cpu:
    isolated: "4-15"
    reserved: "0-3"
  hugepages:
    defaultHugepagesSize: "1G"
    pages:
    - size: "1G"
      count: 16
      node: 0
  realTimeKernel:
    enabled: true
  nodeSelector:
    node-role.kubernetes.io/worker-cnf: ""
```

要覆盖所使用的性能配置集，清单必须挂载到容器中，且必须通过设置 **PERFORMANCE_PROFILE_MANIFEST_OVERRIDE** 参数来指示测试：

```
$ docker run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig -e
PERFORMANCE_PROFILE_MANIFEST_OVERRIDE=/kubeconfig/manifest.yaml
registry.redhat.io/openshift-kni/cnf-tests /usr/bin/test-run.sh
```

15.9.6.4. 禁用性能配置集清理

如果没有在发现模式中运行，该套件会清理所有生成的工件和配置。这包括性能配置集。

删除性能配置集时，机器配置池会被修改并重启节点。在进行了新的迭代后，会生成新的配置集。这会导致在不同测试运行间存在较长的测试周期。

要加快这个过程，设置 **CLEAN_PERFORMANCE_PROFILE="false"** 来指示测试不清理性能配置集。这样，下一个迭代将不需要创建并等待它被应用。

```
$ docker run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig -e
CLEAN_PERFORMANCE_PROFILE="false" registry.redhat.io/openshift-kni/cnf-tests /usr/bin/test-
run.sh
```

15.9.7. 故障排除

集群必须从容器内访问。您可以运行以下命令验证：

```
$ docker run -v $(pwd)/:/kubeconfig -e KUBECONFIG=/kubeconfig/kubeconfig
registry.redhat.io/openshift-kni/cnf-tests oc get nodes
```

如果这不起作用，则可能是跨 DNS、MTU 大小或者防火墙问题导致的。

15.9.8. 测试报告

CNF 端到端测试会产生两个输出：一个 JUnit 测试输出和测试失败报告。

15.9.8.1. JUnit 测试输出

通过传递 **--junit** 参数和转储报告所在路径来生成兼容 JUnit 的 XML:

```
$ docker run -v $(pwd)/:/kubeconfig -v $(pwd)/junitdest:/path/to/junit -e
KUBECONFIG=/kubeconfig/kubeconfig registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-
run.sh --junit /path/to/junit
```

15.9.8.2. 测试失败报告

通过传递 --report 参数来生成包含集群状态信息以及用于故障排除的资源的报告，其方法是通过 **--report** 参数并使用报告转储的路径：

```
$ docker run -v $(pwd)/:/kubeconfig -v $(pwd)/reportdest:/path/to/report -e
KUBECONFIG=/kubeconfig/kubeconfig registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-
run.sh --report /path/to/report
```

15.9.8.3. podman 备注

当以非 root 和非特权身份执行 podman 时，挂载路径可能会因为 "permission denied" 错误而失败。要使它正常工作，在卷创建中附加 **:Z**，例如，使用 **-v $(pwd)/:/kubeconfig:Z** 以允许 podman 进行正确的 SELinux 重新标记。

15.9.8.4. 在 OpenShift Container Platform 4.4 中运行

除以下情况外，CNF 端到端测试与 OpenShift Container Platform 4.4 兼容：

```
[test_id:28466][crit:high][vendor:cnf-qe@redhat.com][level:acceptance] Should contain configuration
injected through openshift-node-performance profile
```

> [test_id:28467][crit:high][vendor:cnf-qe@redhat.com][level:acceptance] Should contain configuration injected through the openshift-node-performance profile

您可以通过添加 **-ginkgo.skip "28466|28467"** 参数来跳过这些测试。

15.9.8.5. 使用单个性能配置集

DPDK 测试需要的资源超过性能测试套件所需要的资源。为加快执行速度，您可以使用提供 DPDK 测试套件的配置集覆盖测试使用的性能配置集。

要做到这一点，类似以下内容的配置集可以挂载到容器中，并可指示性能测试来部署它。

```
apiVersion: performance.openshift.io/v1
kind: PerformanceProfile
metadata:
 name: performance
spec:
 cpu:
  isolated: "5-15"
  reserved: "0-4"
 hugepages:
  defaultHugepagesSize: "1G"
  pages:
  -size: "1G"
   count: 16
   node: 0
 realTimeKernel:
  enabled: true
 numa:
  topologyPolicy: "best-effort"
 nodeSelector:
  node-role.kubernetes.io/worker-cnf: ""
```

要覆盖性能配置集，清单必须挂载到容器中，且必须通过设置 **PERFORMANCE_PROFILE_MANIFEST_OVERRIDE** 来指示测试：

```
$ docker run -v $(pwd)/:/kubeconfig:Z -e KUBECONFIG=/kubeconfig/kubeconfig -e
PERFORMANCE_PROFILE_MANIFEST_OVERRIDE=/kubeconfig/manifest.yaml
registry.redhat.io/openshift4/cnf-tests-rhel8:v4.6 /usr/bin/test-run.sh
```

15.9.9. 对集群的影响

根据功能，运行测试套件可能会对集群造成不同影响。通常，只有 SCTP 测试不会更改集群配置。所有其他特性对配置都会有不同影响。

15.9.9.1. SCTP

SCTP 测试只在不同的节点上运行不同的 pod 以检查是否可以连接性。集群上的影响与在两个节点上运行简单的 pod 相关。

15.9.9.2. SR-IOV

SR-IOV 测试需要更改 SR-IOV 网络配置，其中测试会创建并销毁不同类型的配置。

如果已在集群上安装了现有的 SR-IOV 网络配置，这可能会受到影响，因为可能存在冲突，这取决于这些配置的优先级。

同时，测试的结果可能受到现有配置的影响。

15.9.9.3. PTP

PTP 测试在集群的一组节点中应用 PTP 配置。和 SR-IOV 一样，这可能会与已经存在的任何 PTP 配置冲突，结果无法预计。

15.9.9.4. 性能

性能测试在集群中应用性能配置集。这样做的效果是更改节点配置、保留 CPU、分配内存大页面以及将内核软件包设置为实时。如果集群中已有名为 **performance** 的配置集，则测试不会部署它。

15.9.9.5. DPDK

DPDK 依赖于性能和 SR-IOV 功能，因此测试套件同时配置性能配置集和 SR-IOV 网络，因此影响与 SR-IOV 测试和性能测试中所述的影响相同。

15.9.9.6. 清理

运行测试套件后，清理所有悬停的资源。

15.10. 调试低延迟 CNF 调整状态

PerformanceProfile 自定义资源（CR）包含报告调整状态和调试延迟降级问题的状态字段。这些字段报告描述 Operator 协调功能状态的条件。

当附加到性能配置集的机器配置池处于降级状态时会出现一个典型的问题，从而导致 **PerformanceProfile** 状态降级。在这种情况下，机器配置池会给出一个失败信息。

Performance Addon Operator 包含 **performanceProfile.spec.status.Conditions status** 字段：

```
Status:
  Conditions:
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                True
    Type:                  Available
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                True
    Type:                  Upgradeable
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                False
    Type:                  Progressing
    Last Heartbeat Time:   2020-06-02T10:01:24Z
    Last Transition Time:  2020-06-02T10:01:24Z
    Status:                False
    Type:                  Degraded
```

Status 字段包含指定 **Type** 值来指示性能配置集状态的 **Conditions**：

Available

所有机器配置和 Tuned 配置集都已被成功创建，且集群组件可用于处理它们（NTO、MCO、Kubelet）。

Upgradeable

代表 Operator 维护的资源是否处于可安全升级的状态。

Progressing

表示已从性能配置集启动部署过程。

Degraded

如果出现以下情况代表错误：

- 验证性能配置集失败。
- 创建所有相关组件未能成功完成。

每个类型都包括以下字段：

状态

特定类型的状态（**true** 或 **false**）。

Timestamp

事务的时间戳。

Reason string

机器可读的原因。

Message string

描述状态和错误详情的人类可读的原因信息（如果存在）。

15.10.1. 机器配置池

性能配置集及其创建的产品会根据关联的机器配置池（MCP）应用到节点。MCP 包含有关应用由性能附加组件创建的机器配置的有价值的信息，它包括了内核 arg、Kube 配置、巨页分配和 rt-kernel 部署。性能附加控制器监控 MCP 中的更改，并相应地更新性能配置集状态。

MCP 返回到性能配置集状态的唯一条件是 MCP 处于 **Degraded** 状态，这会导致 **performaceProfile.status.condition.Degraded = true**。

示例

以下示例是创建关联机器配置池（**worker-cnf**）的性能配置集：

1. 关联的机器配置池处于降级状态：

```
# oc get mcp
```

输出示例

```
NAME      CONFIG                                 UPDATED   UPDATING   DEGRADED
MACHINECOUNT   READYMACHINECOUNT   UPDATEDMACHINECOUNT
DEGRADEDMACHINECOUNT   AGE
master    rendered-master-2ee57a93fa6c9181b546ca46e1571d2d    True      False
False     3            3                 3                   0                  2d21h
worker    rendered-worker-d6b2bdc07d9f5a59a6b68950acf25e5f    True      False
```

```
False      2            2                2                0                  2d21h
worker-cnf   rendered-worker-cnf-6c838641b8a08fff08dbd8b02fb63f7c   False      True
True       2            1                1                1                  2d20h
```

2. MCP 的 **describe** 部分包括了原因：

```
# oc describe mcp worker-cnf
```

输出示例

```
  Message:               Node node-worker-cnf is reporting: "prepping update:
  machineconfig.machineconfiguration.openshift.io \"rendered-worker-cnf-
40b9996919c08e335f3ff230ce1d170\" not
  found"
    Reason:              1 nodes are reporting degraded status on sync
```

3. 降级状态也应该出现在标记为 **degraded = true** 的性能配置集的 **status** 字段中：

```
# oc describe performanceprofiles performance
```

输出示例

```
Message: Machine config pool worker-cnf Degraded Reason: 1 nodes are reporting
degraded status on sync.
Machine config pool worker-cnf Degraded Message: Node yquinn-q8s5v-w-b-
z5lqn.c.openshift-gce-devel.internal is
reporting: "prepping update: machineconfig.machineconfiguration.openshift.io
\"rendered-worker-cnf-40b9996919c08e335f3ff230ce1d170\" not found".    Reason:
MCPDegraded
  Status:  True
  Type:    Degraded
```

15.11. 为红帽支持收集调试数据延迟

在提交问题单时同时提供您的集群信息，可以帮助红帽支持为您进行排除故障。

您可使用 **must-gather** 工具来收集有关 OpenShift Container Platform 集群的诊断信息，包括节点调整、NUMA 拓扑和其他调试延迟设置问题所需的信息。

为了获得快速支持，请提供 OpenShift Container Platform 和低延迟调整的诊断信息。

15.11.1. 关于 must-gather 工具

oc adm must-gather CLI 命令可收集最有助于解决问题的集群信息，如：

- 资源定义
- 审计日志
- 服务日志

您在运行该命令时，可通过包含 **--image** 参数来指定一个或多个镜像。指定镜像后，该工具便会收集有关相应功能或产品的信息。在运行 **oc adm must-gather** 时，集群上会创建一个新 pod。在该 pod 上收集数据，并保存至以 **must-gather.local** 开头的一个新目录中。该目录在当前工作目录中创建。

15.11.2. 关于收集低延迟数据

使用 **oc adm must-gather** CLI 命令来收集有关集群的信息，包括与低延迟性能优化相关的功能和对象，包括：

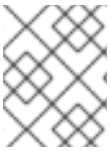
- Performance Addon Operator 命名空间和子对象。
- **MachineConfigPool** 和关联的 **MachineConfig** 对象。
- Node Tuning Operator 和关联的 Tuned 对象。
- Linux 内核命令行选项。
- CPU 和 NUMA 拓扑
- 基本 PCI 设备信息和 NUMA 本地性。

要使用 **must-gather** 来收集 Performance Addon Operator 调试信息，您必须指定 Performance Addon Operator **must-gather** 镜像：

```
--image=registry.redhat.io/openshift4/performance-addon-operator-must-gather-rhel8:v4.6.
```

15.11.3. 收集有关特定功能的数据

您可通过将 **oc adm must-gather** CLI 命令与 **--image** 或 **--image-stream** 参数结合使用来收集有关特定功能的调试信息。**must-gather** 工具支持多个镜像，这样您便可通过运行单个命令收集多个功能的数据。



注意

要收集除特定功能数据外的默认 **must-gather** 数据，请添加 **--image-stream=openshift/must-gather** 参数。

先决条件

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

流程

1. 进入存储 **must-gather** 数据的目录。
2. 使用一个或多个 **--image** 或 **--image-stream** 参数运行 **oc adm must-gather** 命令。例如，使用以下命令可收集默认集群数据和 Performance Addon Operator 特定信息：

```
$ oc adm must-gather \
 --image-stream=openshift/must-gather \ 1

 --image=registry.redhat.io/openshift4/performance-addon-operator-must-gather-rhel8:v4.6
2
```

1. 默认 OpenShift Container Platform **must-gather** 镜像。
2. 低延迟调整诊断的 **must-gather** 镜像。

3. 从工作目录中创建的 **must-gather** 目录创建一个压缩文件。例如，在使用 Linux 操作系统的计算机上运行以下命令：

```
$ tar cvaf must-gather.tar.gz must-gather.local.5421342344627712289/ 1
```

1. 将 **must-gather-local.5421342344627712289/** 替换为实际目录名称。

4. 在红帽客户门户中为您的问题单附上压缩文件。

其他资源

- 有关 MachineConfig 和 KubeletConfig 的更多信息，请参阅管理节点。
- 如需有关 Node Tuning Operator 的更多信息，请参阅使用 Node Tuning Operator。
- 有关 PerformanceProfile 的详情，请参阅配置巨页。
- 有关容器消耗巨页的更多信息，请参阅应用程序如何使用巨页。

第 16 章 使用 INTEL FPGA PAC N3000 和 INTEL VRAN DEDICATED 加速器 ACC100 优化数据平面性能

16.1. 了解 OPENSHIFT CONTAINER PLATFORM 的 INTEL 硬件加速器卡

Intel 提供的硬件加速器卡加速 4G/LTE 和 5G 虚拟化广播访问网络 (vRAN) 工作负载。这会增加一个商用的现成平台的总体计算能力。

Intel FPGA PAC N3000

Intel FPGA PAC N3000 是一个参考 FPGA，它使用 4G/LTE 或 5G 转发错误修正 (FEC) 作为示例工作负载，它加快 5G 或 4G/LTE RAN 第 1 层 (L1) 基本站网络功能。用 4G/LTE 或 5G 位流闪存 Intel FPGA PAC N3000 卡来支持 vRAN 工作负载。

Intel FPGA PAC N3000 是一个全双工，100 Gbps in-system,可重新编程的加速卡，用于多工作负载网络应用程序加速。

当 Intel FPGA PAC N3000 编程为 4G/LTE 或 5G 位stream 时，它会公开用于在 vRAN 工作负载中加速 FEC 的单根 I/O 虚拟化 (SR-IOV) 虚拟功能 (VF) 设备。要利用此功能进行云原生部署，设备的物理功能 (PF) 必须绑定到 **thepf-pci-stub** 驱动程序来创建多个 VF。创建 VF 后，VF 必须绑定到 DPDK 用户空间驱动程序 (vfio)，以将其分配给运行 vRAN 工作负载的特定 pod。

OpenShift Container Platform 上的 Intel FPGA PAC N3000 支持依赖于两个 Operator：

- Intel FPGA PAC N3000 的 Open Operator（编程）
- Wireless FEC Accelerator 的 OpenNESS Operator

vRAN Dedicated 加速器 ACC100

vRAN Dedicated 加速器 ACC100，基于 Intel eASIC 技术卸载，加速了 4G/LTE 和 5G 技术的转发错误更正（FEC）的计算密集型过程，从而释放了处理能力。Intel eASIC 设备是结构化的 ASIC，是 FPGA 和标准应用特定集成电路 (ASIC) 之间的中间技术。

OpenShift Container Platform 上的 Intel vRAN Dedicated 加速器 ACC100 支持使用一个 Operator：

- Wireless FEC Accelerator 的 OpenNESS Operator

16.2. 为 INTEL FPGA PAC N3000 安装 OPENNESS OPERATOR

适用于 Intel FPGA PAC N3000 的 OpenNESS Operator 会编配和管理 OpenShift Container Platform 集群中 Intel FPGA PAC N3000 卡公开的资源或设备。

对于 vRAN 用例，Intel FPGA PAC N3000 的 OpenNESS Operator 与 OpenNESS Operator 用于 Wireless FEC 加速器一同使用。

作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台为 Intel FPGA PAC N3000 安装 OpenNESS Operator。

16.2.1. 使用 CLI 安装 Operator

作为集群管理员，您可以使用 CLI 安装 Operator。

先决条件

- 在裸机硬件上安装的集群。

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

流程

1. 通过完成以下操作，为 N3000 Operator 创建命名空间：
   a. 通过创建名为 **n3000-namespace.yaml** 的文件来定义 **vran-acceleration-operators** 命名空间，如下例所示：

```
apiVersion: v1
kind: Namespace
metadata:
  name: vran-acceleration-operators
  labels:
    openshift.io/cluster-monitoring: "true"
```

   b. 运行以下命令创建命名空间：

```
$ oc create -f n3000-namespace.yaml
```

2. 在您上一步中创建的命名空间中安装 N3000 Operator：
   a. 创建以下 **OperatorGroup** CR，并在 **n3000-operatorgroup.yaml** 文件中保存 YAML：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: n3000-operators
  namespace: vran-acceleration-operators
spec:
  targetNamespaces:
  - vran-acceleration-operators
```

   b. 运行以下命令来创建 **OperatorGroup** CR:

```
$ oc create -f n3000-operatorgroup.yaml
```

   c. 运行以下命令获取下一步所需的 **channel** 值。

```
$ oc get packagemanifest n3000 -n openshift-marketplace -o
jsonpath='{.status.defaultChannel}'
```

   输出示例

```
stable
```

   d. 创建以下 **Subscription** CR，并将 YAML 保存在 **n3000-sub.yaml** 文件中：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
```

```
    name: n3000-subscription
    namespace: vran-acceleration-operators
spec:
    channel: "<channel>"  (1)
    name: n3000
    source: certified-operators  (2)
    sourceNamespace: openshift-marketplace
```

1. 为 **.status.defaultChannel** 参数指定上一步中获取的值的频道值。
2. 您必须指定 **certified-operators** 值。

e. 运行以下命令来创建 **Subscription** CR：

```
$ oc create -f n3000-sub.yaml
```

验证

- 验证已安装了 Operator：

```
$ oc get csv
```

输出示例

```
NAME            DISPLAY                                VERSION  REPLACES    PHASE
n3000.v1.1.0    OpenNESS Operator for Intel® FPGA PAC N3000    1.1.0
Succeeded
```

您现在已成功安装了 Operator。

16.2.2. 使用 Web 控制台为 Intel FPGA PAC N3000 Operator 安装 OpenNESS Operator

作为集群管理员，您可以使用 Web 控制台为 Intel FPGA PAC N3000 安装 OpenNESS Operator。

注意

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

流程

1. 使用 OpenShift Container Platform Web 控制台为 Intel FPGA PAC N3000 安装 OpenNESS Operator：
   a. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
   b. 从可用的 Operator 列表中选择 **OpenNESS Operator for Intel FPGA PAC N3000**，然后点 **Install**。
   c. 在 **Install Operator** 页面中，选择 **All namespaces on the cluster**。然后点击 **Install**。
2. 可选：验证 N3000 Operator 是否已成功安装：

a. 切换到 **Operators → Installed Operators** 页面。

b. 确保 **vran-acceleration-operators** 项目中列出了 **OpenNESS Operator for Intel FPGA PAC N3000**，其 **Status** 为 **InstallSucceeded**。

注意

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

如果控制台没有指示 Operator 已经安装，请执行以下故障排除步骤：

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

16.3. 为 INTEL FPGA PAC N3000 编程 OPENNESS OPERATOR

当 Intel FPGA PAC N3000 使用 vRAN 5G 位流编程时，硬件会使用 vRAN 5G 位流公开 Intel FPGA PAC N3000。此位流公开了用于在 vRAN 工作负载中加速 FEC 的单根 I/O 虚拟化 (SR-IOV) 虚拟功能 (VF) 设备。

作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台为 Intel FPGA PAC N3000 安装 OpenNESS Operator。

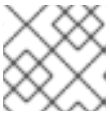
16.3.1. 使用 vRAN 位流编程 N3000

作为集群管理员，您可以使用 vRAN 5G 位流对 Intel FPGA PAC N3000 进行编程。此位流公开了用于在 vRAN 工作负载中加快向前更正 (FEC) 的单根 I/O 虚拟化 (SR-IOV) 虚拟功能 (VF) 设备。

转发错误更正 (FEC) 的角色是更正传输错误，其中消息中的某些位可以丢失或垃圾化。由于传输媒体中的批注、干扰或信号强度较低，消息可能会丢失或被破坏。如果没有 FEC，被破坏的信息需要被重新发送，并添加到网络负载中从而会影响吞吐量和延迟。

先决条件

- Intel FPGA PAC N3000 卡
- 带有 RT 内核配置的 Performance Addon Operator
- 使用 OpenNESS Operator 为 Intel FPGA PAC N3000 安装的节点
- 以具有 **cluster-admin** 权限的用户身份登录



注意

所有命令在 **vran-acceleration-operators** 命名空间中运行。

流程

1. 进入 **vran-acceleration-operators** 项目：

```
$ oc project vran-acceleration-operators
```

2. 验证 pod 是否正在运行：

```
$ oc get pods
```

输出示例

```
NAME                                READY   STATUS    RESTARTS   AGE
fpga-driver-daemonset-8xz4c         1/1     Running   0          15d
fpgainfo-exporter-vhvdq             1/1     Running   1          15d
N3000-controller-manager-b68475c76-gcc6v  2/2   Running   1      15d
N3000-daemonset-5k55l               1/1     Running   1          15d
N3000-discovery-blmjl               1/1     Running   1          15d
N3000-discovery-lblh7               1/1     Running   1          15d
```

以下部分提供有关已安装 pod 的信息：

- **fpga-driver-daemonset** 提供和加载所需的开放编程加速器引擎 (OPAE) 驱动程序
- **fpgainfo-exporter** 为 Prometheus 提供 N3000 遥测数据
- **N3000-controller-manager** 将 N3000Node CR 应用到集群，并管理所有操作对象容器
- **N3000-daemonset** 是主 worker 应用程序。它监控每个节点的 CR 中的更改，并操作更改。在此守护进程中实施的逻辑负责更新卡的 FPGA 用户镜像和 NIC 固件。它还负责排空节点，并在更新需要时将其从委托中删除。
- **N3000-discovery** 发现安装 N3000 加速器设备，并在设备存在时标记 worker 节点

3. 获取包含 Intel FPGA PAC N3000 卡的所有节点：

```
$ oc get n3000node
```

输出示例

```
NAME                FLASH
node1               NotRequested
```

4. 获取有关每个节点卡的信息：

```
$ oc get n3000node node1 -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2020-12-15T17:09:26Z"
    message: Inventory up to date
    observedGeneration: 1
    reason: NotRequested
    status: "False"
    type: Flashed
```

```
  fortville:
  - N3000PCI: 0000:1b:00.0
    NICs:
    - MAC: 64:4c:36:11:1b:a8
      NVMVersion: 7.00 0x800052b0 0.0.0
      PCIAddr: 0000:1a:00.0
      name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
    - MAC: 64:4c:36:11:1b:a9
      NVMVersion: 7.00 0x800052b0 0.0.0
      PCIAddr: 0000:1a:00.1
      name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
    - MAC: 64:4c:36:11:1b:ac
      NVMVersion: 7.00 0x800052b0 0.0.0
      PCIAddr: 0000:1c:00.0
      name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
    - MAC: 64:4c:36:11:1b:ad
      NVMVersion: 7.00 0x800052b0 0.0.0
      PCIAddr: 0000:1c:00.1
      name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
  fpga:
  - PCIAddr: 0000:1b:00.0 (1)
    bitstreamId: "0x23000410010310" (2)
    bitstreamVersion: 0.2.3
    deviceId: "0x0b30"
```

(1) **PCIAddr** 字段指示卡的 PCI 地址。

(2) **bitstreamId** 字段指示当前存储在闪存中的位流。

5. 保存当前的 **bitstreamId**、**PCIAddr**、名称和 **deviceId**（没有 "0x" padding）。

```
$ oc get n3000node -o json
```

6. 更新 Intel FPGA PAC N3000 卡的用户位流：

   a. 通过创建名为 **n3000-cluster.yaml** 的文件，将 N3000 集群资源定义为程序，如下例所示：

```
apiVersion: fpga.intel.com/v1
kind: N3000Cluster
metadata:
  name: n3000 (1)
  namespace: vran-acceleration-operators
spec:
  nodes:
    - nodeName: "node1" (2)
      fpga:
        - userImageURL: "http://10.10.10.122:8000/pkg/20ww27.5-2x2x25G-5GLDPC-
v1.6.1-3.0.0_unsigned.bin" (3)
          PCIAddr: "0000:1b:00.0" (4)
          checksum: "0b0a87b974d35ea16023ceb57f7d5d9c" (5)
```

```
-
```

1. 指定名称。名称必须是 **n3000**。
2. 指定要编程的节点。
3. 指定用户 bitstream 的 URL。这个位流文件必须在 HTTP 或 HTTPS 服务器中访问。
4. 向程序指定卡的 PCI 地址。
5. 指定 **userImageURL** 字段中指定的 bitstream 的 MD5 校验和。

N3000 守护进程使用开放程序加速引擎 (OPAE) 工具更新 FPGA 用户位流并重置 PCI 设备。FPGA 用户位流的更新每个卡最多需要 40 分钟。对于多个节点上的编程卡，在一个时间点上只在一个节点上进行编程。

b. 应用更新以使用位流开始对卡编程：

```
$ oc apply -f n3000-cluster.yaml
```

N3000 守护进程在置备了适当的 5G FEC 用户位流后启动位流编程，如 **20ww27.5-2x2x25G-5GLDPC-v1.6.1-3.0.0_unsigned.bin**，在本例中创建 CR 后。

c. 检查状态：

```
oc get n3000node
```

输出示例

```
NAME          FLASH
node1         InProgress
```

7. 检查日志：

a. 确定 N3000 守护进程的 pod 名称：

```
$ oc get pod -o wide | grep n3000-daemonset | grep node1
```

输出示例

```
n3000-daemonset-5k55l            1/1     Running   0          15d
```

b. 查看日志：

```
$ oc logs n3000-daemonset-5k55l
```

输出示例

```
...
{"level":"info","ts":1608054338.8866854,"logger":"daemon.drainhelper.cordonAndDrain()","
msg":"node drained"}
{"level":"info","ts":1608054338.8867319,"logger":"daemon.drainhelper.Run()","msg":"work
er function - start"}
{"level":"info","ts":1608054338.9003832,"logger":"daemon.fpgaManager.ProgramFPGAs","
```

```
msg":"Start program","PCIAddr":"0000:1b:00.0"}
{"level":"info","ts":1608054338.9004142,"logger":"daemon.fpgaManager.ProgramFPGA","
msg":"Starting","pci":"0000:1b:00.0"}
{"level":"info","ts":1608056309.9367146,"logger":"daemon.fpgaManager.ProgramFPGA","
msg":"Program FPGA completed, start new power cycle N3000 ...","pci":"0000:1b:00.0"}
{"level":"info","ts":1608056333.3528838,"logger":"daemon.drainhelper.Run()","msg":"work
er function - end","performUncordon":true}
...
```

日志文件显示以下事件流：

- 位流会被下载并验证。
- 该节点排空，在此时间内没有工作负载可以运行。
- 启动闪存：
  - 位流会被闪存到卡中。
  - 应用位流。
- 闪存完成后，节点或节点上的 PCI 设备或设备会被重新载入。Wireless FEC 加速器的 OpenNESS SR-IOV Operator 现在可以找到新的闪存设备或设备。

验证

1. 在 FPGA 用户位流更新完成后验证状态：

```
oc get n3000node
```

输出示例

```
NAME          FLASH
node1         Succeeded
```

2. 验证卡的位流 ID 是否已改变：

```
oc get n3000node node1 -o yaml
```

输出示例

```
status:
    conditions:
      - lastTransitionTime: "2020-12-15T18:18:53Z"
        message: Flashed successfully 1
        observedGeneration: 2
        reason: Succeeded
        status: "True"
        type: Flashed
    fortville:
    - N3000PCI: 0000:1b:00.0
      NICs:
      - MAC: 64:4c:36:11:1b:a8
        NVMVersion: 7.00 0x800052b0 0.0.0
```

```
        PCIAddr: 0000:1a:00.0
        name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
      - MAC: 64:4c:36:11:1b:a9
        NVMVersion: 7.00 0x800052b0 0.0.0
        PCIAddr: 0000:1a:00.1
        name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
      - MAC: 64:4c:36:11:1b:ac
        NVMVersion: 7.00 0x800052b0 0.0.0
        PCIAddr: 0000:1c:00.0
        name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
      - MAC: 64:4c:36:11:1b:ad
        NVMVersion: 7.00 0x800052b0 0.0.0
        PCIAddr: 0000:1c:00.1
        name: Ethernet Controller XXV710 Intel(R) FPGA Programmable Acceleration Card
N3000 for Networking
    fpga:
    - PCIAddr: 0000:1b:00.0 (2)
      bitstreamId: "0x2315842A010601" (3)
      bitstreamVersion: 0.2.3
      deviceId: "0x0b30" (4)
```

1. **message** 字段显示设备已成功闪存。
2. **PCIAddr** 字段指示卡的 PCI 地址。
3. **bitstreamId** 字段表示更新的 bitstream ID。
4. **deviceID** 字段表示公开给系统的卡内的位流的设备 ID。

3. 检查节点上的 FEC PCI 设备：

   a. 验证节点配置是否正确应用：

   ```
   $ oc debug node/node1
   ```

   预期输出

   ```
   Starting pod/<node-name>-debug ...
   To use host binaries, run `chroot /host`

   Pod IP: <ip-address>
   If you don't see a command prompt, try pressing enter.

   sh-4.4#
   ```

   b. 验证可以使用节点文件系统：

   ```
   sh-4.4# chroot /host
   ```

   预期输出

   ```
   -
   ```

```
sh-4.4#
```

c. 列出与系统中加速器关联的 PCI 设备：

```
$ lspci | grep accelerators
```

预期输出

```
1b:00.0 Processing accelerators: Intel Corporation Device 0b30
1d:00.0 Processing accelerators: Intel Corporation Device 0d8f (rev 01)
```

属于 FPGA 的设备在输出中报告。设备 ID **0b30** 是 RSU 接口，用于对卡进行编程， **0d8f** 是新编程 5G 设备的物理功能。

16.4. 为 WIRELESS FEC ACCELERATOR 安装 OPENNESS SR-IOV OPERATOR

OpenNESS SR-IOV Operator for Wireless FEC 加速器的角色是编配和管理由 OpenShift Container Platform 集群中一系列 Intel vRAN FEC 加速硬件公开的设备。

计算密集型 4G/LTE 和 5G 工作负载之一是 RAN 第 1 层（L1）转发错误修正（FEC）。FEC 解决了不可靠或不正确的通信通道中的数据传输错误。FEC 技术可检测并更正 4G/LTE 或 5G 数据中的有限数量的错误，而无需重新传输。

FEC 设备由 Intel FPGA PAC N3000 提供，Intel vRAN Dedicated 加速器 ACC100 用于 vRAN 用例。

注意

Intel FPGA PAC N3000 FPGA 需要用 4G/LTE 或 5G 位流进行闪存。

用于连线 FEC 加速器的 OpenNESS SR-IOV Operator 提供了为 FEC 设备创建虚拟功能 (VF) 的功能，将其绑定到适当的驱动程序，并在 4G/LTE 或 5G 部署中配置 VF 队列。

作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator。

16.4.1. 使用 CLI 为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator

作为集群管理员，您可以使用 CLI 为 Wireless FEC 加速器安装 OpenNESS SR-IOV Operator。

先决条件

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

流程

1. 通过完成以下操作，为 Wireless FEC 加速器为 OpenNESS SR-IOV Operator 创建命名空间：

a. 通过创建名为 **sriov-namespace.yaml** 的文件来定义 **vran-acceleration-operators** 命名空间，如下例所示：

```
apiVersion: v1
kind: Namespace
metadata:
  name: vran-acceleration-operators
  labels:
    openshift.io/cluster-monitoring: "true"
```

b. 运行以下命令创建命名空间：

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

2. 通过创建以下对象，在上一步中创建的命名空间中为 Wireless FEC 加速器安装 OpenNESS SR-IOV Operator：

a. 创建以下 **OperatorGroup** CR，并在 **sriov-operatorgroup.yaml** 文件中保存 YAML：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: vran-operators
  namespace: vran-acceleration-operators
spec:
  targetNamespaces:
    - vran-acceleration-operators
```

b. 运行以下命令来创建 **OperatorGroup** CR:

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

c. 运行以下命令获取下一步所需的 **channel** 值。

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

输出示例

```
stable
```

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

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: sriov-fec-subscription
  namespace: vran-acceleration-operators
spec:
  channel: "<channel>" 1
  name: sriov-fec
  source: certified-operators 2
  sourceNamespace: openshift-marketplace
```

1. 为 **.status.defaultChannel** 参数指定上一步中获取的值的频道值。
2. 您必须指定 **certified-operators** 值。

e. 运行以下命令来创建 **Subscription** CR：

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

验证

- 验证是否已安装 Operator：

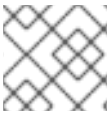
```
$ oc get csv -n vran-acceleration-operators -o custom-
columns=Name:.metadata.name,Phase:.status.phase
```

输出示例

```
Name                         Phase
sriov-fec.v1.1.0             Succeeded
```

16.4.2. 使用 web 控制台为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator

作为集群管理员，您可以使用 web 控制台为 Wireless FEC Accelerator 安装 OpenNESS SR-IOV Operator。

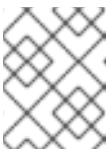


注意

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

流程

1. 使用 OpenShift Container Platform Web 控制台为 Wireless FEC Accelerators 安装 OpenNESS SR-IOV Operator:
   a. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
   b. 从可用的 Operator 列表中选择 **OpenNESS SR-IOV Operator for Wireless FEC Accelerators**，然后点 **Install**。
   c. 在 **Install Operator** 页面中，选择 **All namespaces on the cluster**。然后点击 **Install**。
2. 可选：验证 SRIOV-FEC Operator 是否已成功安装：
   a. 切换到 **Operators → Installed Operators** 页面。
   b. 确保 **OpenNESS SR-IOV Operator for Wireless FEC Accelerators**列在 **vran-acceleration-operators** 项目中，**Status** 为 **InstallSucceeded**。



注意

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

如果控制台没有指示 Operator 已经安装，请执行以下故障排除步骤：

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

16.4.3. 为 Intel FPGA PAC N3000 配置 SR-IOV-FEC Operator

本节论述了如何为 Intel FPGA PAC N3000 编程 SR-IOV-FEC Operator。SR-IOV-FEC Operator 处理用于在 vRAN L1 应用程序中加快 FEC 进程的正向错误修正 (FEC) 设备的管理。

配置 SR-IOV-FEC Operator 涉及以下任务：

- 为 FEC 设备创建所需的虚拟功能（VF）
- 将 VF 绑定到适当的驱动程序
- 为 4G 或 5G 部署中所需的功能配置 VF 队列

转发错误更正 (FEC) 的角色是更正传输错误，其中消息中的某些位可以丢失或垃圾化。由于传输媒体中的批注、干扰或信号强度较低，消息可能会丢失或被破坏。如果没有 FEC，则必须重新发送已垃圾化的消息，以添加到网络负载并影响吞吐量和延迟。

先决条件

- Intel FPGA PAC N3000 卡
- 使用 OpenNESS Operator 为 Intel FPGA PAC N3000 安装的节点（编程）
- 使用 OpenNESS Operator 为 Wireless FEC 加速器安装的节点
- 使用 Performance Addon Operator 配置 RT 内核

流程

1. 进入 **vran-acceleration-operators** 项目：

```
$ oc project vran-acceleration-operators
```

2. 验证 SR-IOV-FEC Operator 是否已安装：

```
$ oc get csv -o custom-columns=Name:.metadata.name,Phase:.status.phase
```

输出示例

```
Name                         Phase
sriov-fec.v1.1.0              Succeeded
n3000.v1.1.0                  Succeeded
```

3. 验证 **N3000** 和 **sriov-fec** pod 是否正在运行：

```
$  oc get pods
```

输出示例

```
NAME                                  READY   STATUS    RESTARTS   AGE
fpga-driver-daemonset-8xz4c           1/1     Running   0          15d
fpgainfo-exporter-vhvdq               1/1     Running   1          15d
N3000-controller-manager-b68475c76-gcc6v  2/2     Running   1          15d
N3000-daemonset-5k55l                 1/1     Running   1          15d
N3000-discovery-blmjl                 1/1     Running   1          15d
N3000-discovery-lblh7                 1/1     Running   1          15d
sriov-device-plugin-j5jlv             1/1     Running   1          15d
sriov-fec-controller-manager-85b6b8f4d4-gd2qg  1/1     Running   1          15d
sriov-fec-daemonset-kqqs6             1/1     Running   1          15d
```

以下部分提供有关已安装 pod 的信息：

- **fpga-driver-daemonset** 提供和加载所需的开放编程加速器引擎 (OPAE) 驱动程序
- **fpgainfo-exporter** 为 Prometheus 提供 N3000 遥测数据
- **N3000-controller-manager** 将 N3000Node CR 应用到集群，并管理所有操作对象容器
- **N3000-daemonset** 是主 worker 应用程序
- **N3000-discovery** 发现安装 N3000 加速器设备，并在设备存在时标记 worker 节点
- **sriov-device-plugin** 将 FEC 虚拟功能公开为节点下的资源
- **sriov-fec-controller-manager** 将 CR 应用到节点，并维护操作对象容器
- **sriov-fec-daemonset** 的责任包括：
  - 发现每个节点上的 SRIOV NIC。
  - 同步第 6 步中定义的自定义资源 (CR) 的状态。
  - 使用 CR 的 spec 作为输入并配置发现的 NIC。

4. 检索包含其中一个支持的 vRAN FEC 加速器设备的所有节点：

```
$ oc get sriovfecnodeconfig
```

输出示例

```
NAME          CONFIGURED
node1         Succeeded
```

5. 查找要配置的 SR-IOV FEC 加速器设备的物理功能 (PF)：

```
$ oc get sriovfecnodeconfig node1 -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2021-03-19T17:19:37Z"
```

```
    message: Configured successfully
    observedGeneration: 1
    reason: ConfigurationSucceeded
    status: "True"
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d5c
      driver: ""
      maxVirtualFunctions: 16
      pciAddress: 0000.1d.00.0 (1)
      vendorID: "8086"
      virtualFunctions: [] (2)
```

1. 此字段显示卡的 PCI 地址。
2. 此字段显示虚拟功能为空。

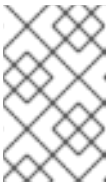
6. 使用所需的设置配置 FEC 设备。

   a. 创建以下自定义资源 (CR)，并将 YAML 保存到 **sriovfec_n3000_cr.yaml** 文件中：

```
apiVersion: sriovfec.intel.com/v1
kind: SriovFecClusterConfig
metadata:
    name: config
    namespace: vran-acceleration-operators
spec:
  nodes:
    - nodeName: node1 (1)
      physicalFunctions:
        - pciAddress: 0000:1d:00.0 (2)
          pfDriver: pci-pf-stub
          vfDriver: vfio-pci
          vfAmount: 2 (3)
          bbDevConfig:
            n3000:
              # Network Type: either "FPGA_5GNR" or "FPGA_LTE"
              networkType: "FPGA_5GNR"
              pfMode: false
              flrTimeout: 610
              downlink:
                bandwidth: 3
                loadBalance: 128
                queues: (4)
                  vf0: 16
                  vf1: 16
                  vf2: 0
                  vf3: 0
                  vf4: 0
                  vf5: 0
                  vf6: 0
                  vf7: 0
              uplink:
```

```
      bandwidth: 3
      loadBalance: 128
      queues: 5
        vf0: 16
        vf1: 16
        vf2: 0
        vf3: 0
        vf4: 0
        vf5: 0
        vf6: 0
        vf7: 0
```

1. 指定节点名称。
2. 指定安装 SR-IOV-FEC Operator 的卡的 PCI 地址。
3. 指定虚拟功能的数量。创建两个虚拟功能：
4. 在 **vf0** 上，使用 16 个总线（downlink 和 uplink）创建一个队列。
5. 在 **vf1** 上，使用 16 个总线（downlink 和 uplink）创建一个队列。



注意

对于 Intel PAC N3000 for vRAN Acceleration，用户可以最多创建 8 个 VF 设备。每个 FEC PF 设备都提供要配置的 64 个队列，32 个队列用于 uplink，32 个队列用于 downlink。队列通常在 VF 之间均匀分布。

b. 应用 CR：

```
$ oc apply -f sriovfec_n3000_cr.yaml
```

应用 CR 后，SR-IOV FEC 守护进程将开始配置 FEC 设备。

验证

1. 检查状态：

```
$ oc get sriovfecclusterconfig config -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2020-12-15T17:19:37Z"
    message: Configured successfully
    observedGeneration: 1
    reason: ConfigurationSucceeded
    status: "True"
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d8f
```

```
    driver: pci-pf-stub
    maxVirtualFunctions: 8
    pciAddress: 0000:1d:00.0
    vendorID: "8086"
    virtualFunctions:
    - deviceID: 0d90
      driver: vfio-pci
      pciAddress: 0000:1d:00.1
    - deviceID: 0d90
      driver: vfio-pci
      pciAddress: 0000:1d:00.2
```

2. 检查日志：

   a. 确定 SR-IOV 守护进程 pod 的名称：

   ```
   $ oc get pod | grep sriov-fec-daemonset
   ```

   输出示例

   ```
   sriov-fec-daemonset-kqqs6                 1/1     Running   0          19h
   ```

   b. 查看日志：

   ```
   $ oc logs sriov-fec-daemonset-kqqs6
   ```

   输出示例

   ```
   2020-12-16T12:46:47.720Z        INFO    daemon.NodeConfigurator.applyConfig
   configuring PF  {"requestedConfig": {"pciAddress":"0000:1d:00.0","pfDriver":"pci-pf-
   stub","vfDriver":"vfio-pci","vfAmount":2,"bbDevConfig":{"n3000":{
   "networkType":"FPGA_5GNR","pfMode":false,"flrTimeout":610,"downlink":
   {"bandwidth":3,"loadBalance":128,"queues":{"vf0":16,"vf1":16}},"uplink":
   {"bandwidth":3,"loadBalance":128,"queues":{"vf0":16,"vf1":16}}}}}
   2020-12-16T12:46:47.720Z        INFO    daemon.NodeConfigurator.loadModule
   executing command       {"cmd": "/usr/sbin/chroot /host/ modprobe pci-pf-stub"}
   2020-12-16T12:46:47.724Z        INFO    daemon.NodeConfigurator.loadModule
   commands output {"output": ""}
   2020-12-16T12:46:47.724Z        INFO    daemon.NodeConfigurator.loadModule
   executing command       {"cmd": "/usr/sbin/chroot /host/ modprobe vfio-pci"}
   2020-12-16T12:46:47.727Z        INFO    daemon.NodeConfigurator.loadModule
   commands output {"output": ""}
   2020-12-16T12:46:47.727Z        INFO    daemon.NodeConfigurator device's
   driver_override path    {"path": "/sys/bus/pci/devices/0000:1d:00.0/driver_override"}
   2020-12-16T12:46:47.727Z        INFO    daemon.NodeConfigurator driver bind path
   {"path": "/sys/bus/pci/drivers/pci-pf-stub/bind"}
   2020-12-16T12:46:47.998Z        INFO    daemon.NodeConfigurator device's
   driver_override path    {"path": "/sys/bus/pci/devices/0000:1d:00.1/driver_override"}
   2020-12-16T12:46:47.998Z        INFO    daemon.NodeConfigurator driver bind path
   {"path": "/sys/bus/pci/drivers/vfio-pci/bind"}
   2020-12-16T12:46:47.998Z        INFO    daemon.NodeConfigurator device's
   driver_override path    {"path": "/sys/bus/pci/devices/0000:1d:00.2/driver_override"}
   2020-12-16T12:46:47.998Z        INFO    daemon.NodeConfigurator driver bind path
   {"path": "/sys/bus/pci/drivers/vfio-pci/bind"}
   ```

```
2020-12-16T12:46:47.999Z        INFO    daemon.NodeConfigurator.applyConfig
executing command       {"cmd": "/sriov_workdir/pf_bb_config FPGA_5GNR -c
/sriov_artifacts/0000:1d:00.0.ini -p 0000:1d:00.0"}
2020-12-16T12:46:48.017Z        INFO    daemon.NodeConfigurator.applyConfig
commands output {"output": "ERROR: Section (FLR) or name (flr_time_out) is not valid.
FEC FPGA RTL v3.0
UL.DL Weights = 3.3
UL.DL Load Balance = 1
28.128
Queue-PF/VF Mapping Table = READY
Ring Descriptor Size = 256 bytes

--------+-----+-----+-----+-----+-----+-----+-----+-----+-----+
       |  PF | VF0 | VF1 | VF2 | VF3 | VF4 | VF5 | VF6 | VF7 |
--------+-----+-----+-----+-----+-----+-----+-----+-----+-----+
UL-Q'00 |     | X |     |     |     |     |     |     |     |
UL-Q'01 |     | X |     |     |     |     |     |     |     |
UL-Q'02 |     | X |     |     |     |     |     |     |     |
UL-Q'03 |     | X |     |     |     |     |     |     |     |
UL-Q'04 |     | X |     |     |     |     |     |     |     |
UL-Q'05 |     | X |     |     |     |     |     |     |     |
UL-Q'06 |     | X |     |     |     |     |     |     |     |
UL-Q'07 |     | X |     |     |     |     |     |     |     |
UL-Q'08 |     | X |     |     |     |     |     |     |     |
UL-Q'09 |     | X |     |     |     |     |     |     |     |
UL-Q'10 |     | X |     |     |     |     |     |     |     |
UL-Q'11 |     | X |     |     |     |     |     |     |     |
UL-Q'12 |     | X |     |     |     |     |     |     |     |
UL-Q'13 |     | X |     |     |     |     |     |     |     |
UL-Q'14 |     | X |     |     |     |     |     |     |     |
UL-Q'15 |     | X |     |     |     |     |     |     |     |
UL-Q'16 |     |     | X |     |     |     |     |     |     |
UL-Q'17 |     |     | X |     |     |     |     |     |     |
UL-Q'18 |     |     | X |     |     |     |     |     |     |
UL-Q'19 |     |     | X |     |     |     |     |     |     |
UL-Q'20 |     |     | X |     |     |     |     |     |     |
UL-Q'21 |     |     | X |     |     |     |     |     |     |
UL-Q'22 |     |     | X |     |     |     |     |     |     |
UL-Q'23 |     |     | X |     |     |     |     |     |     |
UL-Q'24 |     |     | X |     |     |     |     |     |     |
UL-Q'25 |     |     | X |     |     |     |     |     |     |
UL-Q'26 |     |     | X |     |     |     |     |     |     |
UL-Q'27 |     |     | X |     |     |     |     |     |     |
UL-Q'28 |     |     | X |     |     |     |     |     |     |
UL-Q'29 |     |     | X |     |     |     |     |     |     |
UL-Q'30 |     |     | X |     |     |     |     |     |     |
UL-Q'31 |     |     | X |     |     |     |     |     |     |
DL-Q'32 |     | X |     |     |     |     |     |     |     |
DL-Q'33 |     | X |     |     |     |     |     |     |     |
DL-Q'34 |     | X |     |     |     |     |     |     |     |
DL-Q'35 |     | X |     |     |     |     |     |     |     |
DL-Q'36 |     | X |     |     |     |     |     |     |     |
DL-Q'37 |     | X |     |     |     |     |     |     |     |
DL-Q'38 |     | X |     |     |     |     |     |     |     |
DL-Q'39 |     | X |     |     |     |     |     |     |     |
DL-Q'40 |     | X |     |     |     |     |     |     |     |
```

```
DL-Q'41 |    | X |    |    |    |    |    |    |
DL-Q'42 |    | X |    |    |    |    |    |    |
DL-Q'43 |    | X |    |    |    |    |    |    |
DL-Q'44 |    | X |    |    |    |    |    |    |
DL-Q'45 |    | X |    |    |    |    |    |    |
DL-Q'46 |    | X |    |    |    |    |    |    |
DL-Q'47 |    | X |    |    |    |    |    |    |
DL-Q'48 |    |    | X |    |    |    |    |    |
DL-Q'49 |    |    | X |    |    |    |    |    |
DL-Q'50 |    |    | X |    |    |    |    |    |
DL-Q'51 |    |    | X |    |    |    |    |    |
DL-Q'52 |    |    | X |    |    |    |    |    |
DL-Q'53 |    |    | X |    |    |    |    |    |
DL-Q'54 |    |    | X |    |    |    |    |    |
DL-Q'55 |    |    | X |    |    |    |    |    |
DL-Q'56 |    |    | X |    |    |    |    |    |
DL-Q'57 |    |    | X |    |    |    |    |    |
DL-Q'58 |    |    | X |    |    |    |    |    |
DL-Q'59 |    |    | X |    |    |    |    |    |
DL-Q'60 |    |    | X |    |    |    |    |    |
DL-Q'61 |    |    | X |    |    |    |    |    |
DL-Q'62 |    |    | X |    |    |    |    |    |
DL-Q'63 |    |    | X |    |    |    |    |    |
--------+-----+-----+-----+-----+-----+-----+-----+-----+-----+

Mode of operation = VF-mode
FPGA_5GNR PF [0000:1d:00.0] configuration complete!"}
2020-12-16T12:46:48.017Z        INFO    daemon.NodeConfigurator.enableMasterBus
executing command       {"cmd": "/usr/sbin/chroot /host/ setpci -v -s 0000:1d:00.0
COMMAND"}
2020-12-16T12:46:48.037Z        INFO    daemon.NodeConfigurator.enableMasterBus
commands output {"output": "0000:1d:00.0 @04 = 0102\n"}
2020-12-16T12:46:48.037Z        INFO    daemon.NodeConfigurator.enableMasterBus
executing command       {"cmd": "/usr/sbin/chroot /host/ setpci -v -s 0000:1d:00.0
COMMAND=0106"}
2020-12-16T12:46:48.054Z        INFO    daemon.NodeConfigurator.enableMasterBus
commands output {"output": "0000:1d:00.0 @04 0106\n"}
2020-12-16T12:46:48.054Z        INFO    daemon.NodeConfigurator.enableMasterBus
MasterBus set   {"pci": "0000:1d:00.0", "output": "0000:1d:00.0 @04 0106\n"}
2020-12-16T12:46:48.160Z        INFO    daemon.drainhelper.Run()        worker function -
end   {"performUncordon": true}
```

3. 检查卡的 FEC 配置：

```
$ oc get sriovfecnodeconfig node1 -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2020-12-15T17:19:37Z"
    message: Configured successfully
    observedGeneration: 1
    reason: ConfigurationSucceeded
    status: "True"
```

```
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d8f 1
      driver: pci-pf-stub
      maxVirtualFunctions: 8
      pciAddress: 0000:1d:00.0
      vendorID: "8086"
    virtualFunctions:
    - deviceID: 0d90 2
      driver: vfio-pci
      pciAddress: 0000:1d:00.1
    - deviceID: 0d90
      driver: vfio-pci
      pciAddress: 0000:1d:00.2
```

1 **0d8f** 是 FEC 设备的 **deviceID** 物理功能（PF）。

2 **0d90** 是 FEC 设备的 **deviceID** 虚拟功能（VF）。

16.4.4. 为 Intel vRAN Dedicated Accelerator ACC100 配置 SR-IOV-FEC Operator

编程 Intel vRAN Dedicated 加速器 ACC100 会公开用于在 vRAN 工作负载中加速转发错误 (FEC) 的单根 I/O 虚拟化 (SRIOV) 虚拟功能 (VF) 设备。Intel vRAN Dedicated 加速器 ACC100 加速了 4G 和 5G 虚拟化广播访问网络 (vRAN) 工作负载。这会增加一个商用的现成平台的总体计算能力。此设备也称为 Mount Bryce。

SR-IOV-FEC Operator 处理用于在 vRAN L1 应用程序中加快 FEC 进程的正向错误修正 (FEC) 设备的管理。

配置 SR-IOV-FEC Operator 涉及以下任务：

- 为 FEC 设备创建虚拟功能 (VF)
- 将 VF 绑定到适当的驱动程序
- 为 4G 或 5G 部署中所需的功能配置 VF 队列

转发错误更正 (FEC) 的角色是更正传输错误，其中消息中的某些位可以丢失或垃圾化。由于传输媒体中的批注、干扰或信号强度较低，消息可能会丢失或被破坏。如果没有 FEC，则必须重新发送已垃圾化的消息，以添加到网络负载并影响吞吐量和延迟。

先决条件

- Intel FPGA ACC100 5G/4G 卡
- 使用 OpenNESS Operator 为 Wireless FEC 加速器安装的节点
- 在 BIOS 中为节点启用全局 SR-IOV 和 VT-d 设置
- 使用 Performance Addon Operator 配置 RT 内核
- 以具有 **cluster-admin** 权限的用户身份登录

流程

1. 进入 **vran-acceleration-operators** 项目：

```
$ oc project vran-acceleration-operators
```

2. 验证 SR-IOV-FEC Operator 是否已安装：

```
$ oc get csv -o custom-columns=Name:.metadata.name,Phase:.status.phase
```

输出示例

```
Name                                Phase
sriov-fec.v1.1.0                    Succeeded
```

3. 验证 **sriov-fec** pod 是否正在运行：

```
$  oc get pods
```

输出示例

```
NAME                                READY   STATUS    RESTARTS   AGE
sriov-device-plugin-j5jlv           1/1     Running   1          15d
sriov-fec-controller-manager-85b6b8f4d4-gd2qg   1/1     Running   1          15d
sriov-fec-daemonset-kqqs6           1/1     Running   1          15d
```

- **sriov-device-plugin** 将 FEC 虚拟功能公开为节点下的资源
- **sriov-fec-controller-manager** 将 CR 应用到节点，并维护操作对象容器
- **sriov-fec-daemonset** 的责任包括：
  - 发现每个节点上的 SRIOV NIC。
  - 同步第 6 步中定义的自定义资源 (CR) 的状态。
  - 使用 CR 的 spec 作为输入并配置发现的 NIC。

4. 检索包含其中一个支持的 vRAN FEC 加速器设备的所有节点：

```
$ oc get sriovfecnodeconfig
```

输出示例

```
NAME          CONFIGURED
node1         Succeeded
```

5. 查找要配置的 SR-IOV FEC 加速器设备的物理功能 (PF)：

```
$ oc get sriovfecnodeconfig node1 -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2021-03-19T17:19:37Z"
    message: Configured successfully
    observedGeneration: 1
    reason: ConfigurationSucceeded
    status: "True"
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d5c
      driver: ""
      maxVirtualFunctions: 16
      pciAddress: 0000:af:00.0 (1)
      vendorID: "8086"
      virtualFunctions: [] (2)
```

1. 此字段表示卡的 PCI 地址。
2. 此字段显示虚拟功能为空。

6. 在 FEC 设备中配置虚拟功能和队列组的数量：

   a. 创建以下自定义资源(CR)，并将 YAML 保存到 **sriovfec_acc100cr.yaml** 文件中：

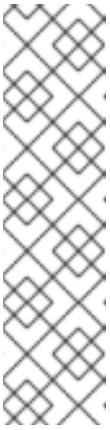
注意

本例为 5G 配置 ACC100 8/8 队列组，为 Uplink 配置 4 个队列组，另一个 4 个队列组用于 Downlink。

```
apiVersion: sriovfec.intel.com/v1
kind: SriovFecClusterConfig
metadata:
  name: config (1)
spec:
  nodes:
  - nodeName: node1 (2)
    physicalFunctions:
    - pciAddress: 0000:af:00.0 (3)
      pfDriver: "pci-pf-stub"
      vfDriver: "vfio-pci"
      vfAmount: 16 (4)
      bbDevConfig:
        acc100:
          # Programming mode: 0 = VF Programming, 1 = PF Programming
          pfMode: false
          numVfBundles: 16
          maxQueueSize: 1024
          uplink4G:
            numQueueGroups: 0
            numAqsPerGroups: 16
            aqDepthLog2: 4
          downlink4G:
```

```
    numQueueGroups: 0
    numAqsPerGroups: 16
    aqDepthLog2: 4
  uplink5G:
    numQueueGroups: 4
    numAqsPerGroups: 16
    aqDepthLog2: 4
  downlink5G:
    numQueueGroups: 4
    numAqsPerGroups: 16
    aqDepthLog2: 4
```

1. 为 CR 对象指定一个名称。可以指定的唯一名称是 **config**。
2. 指定节点名称。
3. 指定 SR-IOV-FEC Operator 将要安装的卡的 PCI 地址。
4. 指定要创建的虚拟功能数量。对于 Intel vRAN Dedicated 加速器 ACC100，创建所有 16 个 VF。



注意

该卡配置为提供最多 8 个队列组，每个组最多有 16 个队列。队列组可以划分到分配给 5G 和 4G 的组，以及 Uplink 和 Downlink。Intel vRAN Dedicated 加速器 ACC100 可以配置为：

- 仅 4G 或 5G
- 同时 4G 和 5G

每个配置的 VF 都可以访问所有队列。每个队列组都具有不同的优先级级别。对给定队列组的请求从应用级别发出，即 vRAN 应用利用 FEC 设备。

b. 应用 CR：

```
$ oc apply -f sriovfec_acc100cr.yaml
```

应用 CR 后，SR-IOV FEC 守护进程将开始配置 FEC 设备。

验证

1. 检查状态：

```
$ oc get sriovfecclusterconfig config -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2021-03-19T11:46:22Z"
    message: Configured successfully
    observedGeneration: 1
```

```
    reason: Succeeded
    status: "True"
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d5c
      driver: pci-pf-stub
      maxVirtualFunctions: 16
      pciAddress: 0000:af:00.0
      vendorID: "8086"
      virtualFunctions:
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.0
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.1
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.2
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.3
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.4
```

2. 检查日志：

   a. 确定 SR-IOV 守护进程的 pod 名称：

   ```
   $ oc get po -o wide | grep sriov-fec-daemonset | grep node1
   ```

   输出示例

   ```
   sriov-fec-daemonset-kqqs6                 1/1     Running   0          19h
   ```

   b. 查看日志：

   ```
   $ oc logs sriov-fec-daemonset-kqqs6
   ```

   输出示例

   ```
   {"level":"Level(-
   2)","ts":1616794345.4786215,"logger":"daemon.drainhelper.cordonAndDrain()","msg":"no
   de drained"}
   {"level":"Level(-
   4)","ts":1616794345.4786265,"logger":"daemon.drainhelper.Run()","msg":"worker
   function - start"}
   {"level":"Level(-
   4)","ts":1616794345.5762916,"logger":"daemon.NodeConfigurator.applyConfig","msg":"cur
   rent node status","inventory":{"sriovAccelerat
   ors":
   [{"vendorID":"8086","deviceID":"0b32","pciAddress":"0000:20:00.0","driver":"","maxVirtualF
   ```

unctions":1,"virtualFunctions":[]},{"vendorID":"8086"
,"deviceID":"0d5c","pciAddress":"0000:af:00.0","driver":"","maxVirtualFunctions":16,"virtualF
unctions":[]}]}}
{"level":"Level(-
4)","ts":1616794345.5763638,"logger":"daemon.NodeConfigurator.applyConfig","msg":"co
nfiguring PF","requestedConfig":{"pciAddress":"
0000:af:00.0","pfDriver":"pci-pf-stub","vfDriver":"vfio-pci","vfAmount":2,"bbDevConfig":
{"acc100":{"pfMode":false,"numVfBundles":16,"maxQueueSize":1
024,"uplink4G":
{"numQueueGroups":4,"numAqsPerGroups":16,"aqDepthLog2":4},"downlink4G":
{"numQueueGroups":4,"numAqsPerGroups":16,"aqDepthLog2":4},"uplink5G":
{"numQueueGroups":0,"numAqsPerGroups":16,"aqDepthLog2":4},"downlink5G":
{"numQueueGroups":0,"numAqsPerGroups":16,"aqDepthLog2":4}}}}}
{"level":"Level(-
4)","ts":1616794345.5774765,"logger":"daemon.NodeConfigurator.loadModule","msg":"ex
ecuting command","cmd":"/usr/sbin/chroot /host/ modprobe pci-pf-stub"}
{"level":"Level(-
4)","ts":1616794345.5842702,"logger":"daemon.NodeConfigurator.loadModule","msg":"co
mmands output","output":""}
{"level":"Level(-
4)","ts":1616794345.5843055,"logger":"daemon.NodeConfigurator.loadModule","msg":"ex
ecuting command","cmd":"/usr/sbin/chroot /host/ modprobe vfio-pci"}
{"level":"Level(-
4)","ts":1616794345.6090655,"logger":"daemon.NodeConfigurator.loadModule","msg":"co
mmands output","output":""}
{"level":"Level(-
2)","ts":1616794345.6091156,"logger":"daemon.NodeConfigurator","msg":"device's
driver_override path","path":"/sys/bus/pci/devices/0000:af:00.0/driver_override"}
{"level":"Level(-
2)","ts":1616794345.6091807,"logger":"daemon.NodeConfigurator","msg":"driver bind
path","path":"/sys/bus/pci/drivers/pci-pf-stub/bind"}
{"level":"Level(-
2)","ts":1616794345.7488534,"logger":"daemon.NodeConfigurator","msg":"device's
driver_override path","path":"/sys/bus/pci/devices/0000:b0:00.0/driver_override"}
{"level":"Level(-
2)","ts":1616794345.748938,"logger":"daemon.NodeConfigurator","msg":"driver bind
path","path":"/sys/bus/pci/drivers/vfio-pci/bind"}
{"level":"Level(-
2)","ts":1616794345.7492096,"logger":"daemon.NodeConfigurator","msg":"device's
driver_override path","path":"/sys/bus/pci/devices/0000:b0:00.1/driver_override"}
{"level":"Level(-
2)","ts":1616794345.7492566,"logger":"daemon.NodeConfigurator","msg":"driver bind
path","path":"/sys/bus/pci/drivers/vfio-pci/bind"}
{"level":"Level(-
4)","ts":1616794345.74968,"logger":"daemon.NodeConfigurator.applyConfig","msg":"exec
uting command","cmd":"/sriov_workdir/pf_bb_config ACC100 -c
/sriov_artifacts/0000:af:00.0.ini -p 0000:af:00.0"}
{"level":"Level(-
4)","ts":1616794346.5203931,"logger":"daemon.NodeConfigurator.applyConfig","msg":"co
mmands output","output":"Queue Groups: 0 5GUL, 0 5GDL, 4 4GUL, 4 4GDL\nNumber
of 5GUL engines 8\nConfiguration in VF mode\nPF ACC100 configuration
complete\nACC100 PF [0000:af:00.0] configuration complete!\n\n"}
{"level":"Level(-
4)","ts":1616794346.520459,"logger":"daemon.NodeConfigurator.enableMasterBus","msg"
:"executing command","cmd":"/usr/sbin/chroot /host/ setpci -v -s 0000:af:00.0
COMMAND"}

{"level":"Level(-4)","ts":1616794346.5458736,"logger":"daemon.NodeConfigurator.enableMasterBus","msg":"commands output","output":"0000:af:00.0 @04 = 0142\n"}
{"level":"Level(-4)","ts":1616794346.5459251,"logger":"daemon.NodeConfigurator.enableMasterBus","msg":"executing command","cmd":"/usr/sbin/chroot /host/ setpci -v -s 0000:af:00.0 COMMAND=0146"}
{"level":"Level(-4)","ts":1616794346.5795262,"logger":"daemon.NodeConfigurator.enableMasterBus","msg":"commands output","output":"0000:af:00.0 @04 0146\n"}
{"level":"Level(-2)","ts":1616794346.5795407,"logger":"daemon.NodeConfigurator.enableMasterBus","msg":"MasterBus set","pci":"0000:af:00.0","output":"0000:af:00.0 @04 0146\n"}
{"level":"Level(-4)","ts":1616794346.6867144,"logger":"daemon.drainhelper.Run()","msg":"worker function - end","performUncordon":true}
{"level":"Level(-4)","ts":1616794346.6867719,"logger":"daemon.drainhelper.Run()","msg":"uncordoning node"}
{"level":"Level(-4)","ts":1616794346.6896322,"logger":"daemon.drainhelper.uncordon()","msg":"starting uncordon attempts"}
{"level":"Level(-2)","ts":1616794346.69735,"logger":"daemon.drainhelper.uncordon()","msg":"node uncordoned"}
{"level":"Level(-4)","ts":1616794346.6973662,"logger":"daemon.drainhelper.Run()","msg":"cancelling the context to finish the leadership"}
{"level":"Level(-4)","ts":1616794346.7029872,"logger":"daemon.drainhelper.Run()","msg":"stopped leading"}
{"level":"Level(-4)","ts":1616794346.7030034,"logger":"daemon.drainhelper","msg":"releasing the lock (bug mitigation)"}
{"level":"Level(-4)","ts":1616794346.8040674,"logger":"daemon.updateInventory","msg":"obtained inventory","inv":{"sriovAccelerators":[{"vendorID":"8086","deviceID":"0b32","pciAddress":"0000:20:00.0","driver":"","maxVirtualFunctions":1,"virtualFunctions":[]},{"vendorID":"8086","deviceID":"0d5c","pciAddress":"0000:af:00.0","driver":"pci-pf-stub","maxVirtualFunctions":16,"virtualFunctions":[{"pciAddress":"0000:b0:00.0","driver":"vfio-pci","deviceID":"0d5d"},{"pciAddress":"0000:b0:00.1","driver":"vfio-pci","deviceID":"0d5d"}]}]}}
{"level":"Level(-4)","ts":1616794346.9058325,"logger":"daemon","msg":"Update ignored, generation unchanged"}
{"level":"Level(-2)","ts":1616794346.9065044,"logger":"daemon.Reconcile","msg":"Reconciled","namespace":"vran-acceleration-operators","name":"pg-itengdvs02r.altera.com"}

3. 检查卡的 FEC 配置：

```
$ oc get sriovfecnodeconfig node1 -o yaml
```

输出示例

```
status:
  conditions:
  - lastTransitionTime: "2021-03-19T11:46:22Z"
    message: Configured successfully
    observedGeneration: 1
    reason: Succeeded
    status: "True"
    type: Configured
  inventory:
    sriovAccelerators:
    - deviceID: 0d5c 1
      driver: pci-pf-stub
      maxVirtualFunctions: 16
      pciAddress: 0000:af:00.0
      vendorID: "8086"
      virtualFunctions:
      - deviceID: 0d5d 2
        driver: vfio-pci
        pciAddress: 0000:b0:00.0
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.1
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.2
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.3
      - deviceID: 0d5d
        driver: vfio-pci
        pciAddress: 0000:b0:00.4
```

1 值 **0d5c** 是 FEC 设备的物理功能的 **deviceID**。

2 值 **0d5d** 是 FEC 设备的虚拟功能的 **deviceID**。

16.4.5. 在 OpenNESS 上验证应用程序 pod 访问和 FPGA 使用情况

OpenNESS 是一种边缘计算软件工具包，可用于在任何类型的网络上添加和管理应用程序和网络功能。

通过构建镜像并为设备运行简单的验证应用程序，验证所有 OpenNESS 功能是否协同工作，包括 SR-IOV 绑定、设备插件、Wire Base Band Device（bbdev）配置和 SR-IOV（FEC）VF 功能。

有关更多信息，请转至 openess.org。

先决条件

- 可选：Intel FPGA PAC N3000 卡
- 带有 n3000-operator 的安装的节点
- 带有 SR-IOV-FEC Operator 的安装的节点
- 使用 Performance Addon Operator 配置的实时内核和巨页

- 以具有 **cluster-admin** 权限的用户身份登录

流程

1. 通过完成以下操作，为测试创建一个命名空间：
   a. 通过创建名为 **test-bbdev-namespace.yaml** 文件的文件来定义 **test-bbdev** 命名空间，如下例所示：

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-bbdev
  labels:
    openshift.io/run-level: "1"
```

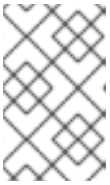
   b. 运行以下命令创建命名空间：

```
$ oc create -f test-bbdev-namespace.yaml
```

2. 创建以下 **Pod** 规格，然后在 **pod-test.yaml** 文件中保存 YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-bbdev-sample-app
  namespace: test-bbdev (1)
spec:
  containers:
  - securityContext:
      privileged: false
      capabilities:
        add:
         - IPC_LOCK
         - SYS_NICE
    name: bbdev-sample-app
    image: bbdev-sample-app:1.0 (2)
    command: [ "sudo", "/bin/bash", "-c", "--" ]
    runAsUser: 0 (3)
    resources:
      requests:
        hugepages-1Gi: 4Gi (4)
        memory: 1Gi
        cpu: "4" (5)
        intel.com/intel_fec_5g: '1' (6)
        #intel.com/intel_fec_acc100: '1'
        #intel.com/intel_fec_lte: '1'
      limits:
        memory: 4Gi
        cpu: "4"
        hugepages-1Gi: 4Gi
        intel.com/intel_fec_5g: '1'
        #intel.com/intel_fec_acc100: '1'
        #intel.com/intel_fec_lte: '1
```

1. 指定在第 1 步中创建的**命名空间**。
2. 这将定义包含已编译的 DPDK 的测试镜像。
3. 使容器在内部以 root 用户身份执行。
4. 指定巨页大小 **hugepages-1Gi** 以及分配给 pod 的巨页数量。需要使用 Performance Addon Operator 配置巨页和隔离的 CPU。
5. 指定 CPU 数量。
6. **intel.com/intel_fec_5g** 支持 N3000 5G FEC 配置测试。



注意

要测试 ACC100 配置，请通过删除 # 符号取消注释 **intel.com/intel_fec_acc100**。要测试 N3000 4G/LTE 配置，请通过删除 # 符号取消注释 **intel.com/intel_fec_lte**。任何时候只能有一个资源处于活动状态。

3. 创建 pod：

```
$ oc apply -f pod-test.yaml
```

4. 检查是否创建了 pod：

```
$ oc get pods -n test-bbdev
```

输出示例

```
NAME                    READY   STATUS    RESTARTS   AGE
pod-bbdev-sample-app    1/1     Running   0          80s
```

5. 使用远程 shell 登录 **pod-bbdev-sample-app**：

```
$ oc rsh pod-bbdev-sample-app
```

输出示例

```
sh-4.4#
```

6. 显示环境变量列表：

```
sh-4.4# env
```

输出示例

```
N3000_CONTROLLER_MANAGER_METRICS_SERVICE_PORT_8443_TCP_ADDR=172.30.133.131
SRIOV_FEC_CONTROLLER_MANAGER_METRICS_SERVICE_PORT_8443_TCP_PROTO=tcp
DPDK_VERSION=20.11
```

```
PCIDEVICE_INTEL_COM_INTEL_FEC_5G=0.0.0.0:1d.00.0 1
~/usr/bin/env
HOSTNAME=fec-pod
```

1 这是虚拟功能的 PCI 地址。根据您在 **pod-test.yaml** 文件中请求的资源，这可以是以下三个 PCI 地址之一：

- PCIDEVICE_INTEL_COM_INTEL_FEC_ACC100
- PCIDEVICE_INTEL_COM_INTEL_FEC_5G
- PCIDEVICE_INTEL_COM_INTEL_FEC_LTE

7. 进入 **test-bbdev** 目录：

```
sh-4.4# cd test/test-bbdev/
```

注意

目录位于 pod 中，而不是本地计算机上。

8. 检查分配给 pod 的 CPU：

```
sh-4.4# export CPU=$(cat /sys/fs/cgroup/cpuset/cpuset.cpus)
sh-4.4# echo ${CPU}
```

这将打印分配给 **fec.pod** 的 CPU。

输出示例

```
24,25,64,65
```

9. 运行 **test-bbdev** 应用程序来测试设备：

```
sh-4.4# ./test-bbdev.py -e="-l ${CPU} -a ${PCIDEVICE_INTEL_COM_INTEL_FEC_5G}" -c
validation \ -n 64 -b 32 -l 1 -v ./test_vectors/*"
```

输出示例

```
Executing: ../../build/app/dpdk-test-bbdev -l 24-25,64-65 0000:1d.00.0 -- -n 64 -l 1 -c
validation -v ./test_vectors/bbdev_null.data -b 32
EAL: Detected 80 lcore(s)
EAL: Detected 2 NUMA nodes
Option -w, --pci-whitelist is deprecated, use -a, --allow option instead
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: Selected IOVA mode 'VA'
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL:   using IOMMU type 1 (Type 1)
EAL: Probe PCI driver: intel_fpga_5ngr_fec_vf (8086:d90) device: 0000:1d.00.0 (socket 1)
EAL: No legacy callbacks, legacy socket not created
```

```
===========================================================
Starting Test Suite : BBdev Validation Tests
Test vector file = ldpc_dec_v7813.data
Device 0 queue 16 setup failed
Allocated all queues (id=16) at prio0 on dev0
Device 0 queue 32 setup failed
Allocated all queues (id=32) at prio1 on dev0
Device 0 queue 48 setup failed
Allocated all queues (id=48) at prio2 on dev0
Device 0 queue 64 setup failed
Allocated all queues (id=64) at prio3 on dev0
Device 0 queue 64 setup failed
All queues on dev 0 allocated: 64
+ ------------------------------------------------------- +
== test: validation
dev:0000:b0:00.0, burst size: 1, num ops: 1, op type: RTE_BBDEV_OP_LDPC_DEC
Operation latency:
        avg: 23092 cycles, 10.0838 us
        min: 23092 cycles, 10.0838 us
        max: 23092 cycles, 10.0838 us
TestCase [ 0] : validation_tc passed
 + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ +
 + Test Suite Summary : BBdev Validation Tests
 + Tests Total :       1
 + Tests Skipped :     0
 + Tests Passed :      1 (1)
 + Tests Failed :      0
 + Tests Lasted :      177.67 ms
 + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ +
```

1 虽然可以跳过某些测试，但请确保向量测试通过。

16.5. 其它资源

- OpenNESS Operator for Intel® FPGA PAC N3000 (Programming)
- Wireless FEC Accelerator 的 OpenNESS Operator