OpenShift Container Platform 4.16

专用硬件和驱动程序启用

了解 OpenShift Container Platform 中的硬件启用

Last Updated: 2024-07-03

OpenShift Container Platform 4.16 专用硬件和驱动程序启用

了解 OpenShift Container Platform 中的硬件启用

法律通告

摘要

本文档概述 OpenShift Container Platform 中的硬件启用。

目录

第 1 章 关于专用硬件和驱动程序启用

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

驱动程序容器是容器镜像，用于在容器操作系统（如 Red Hat Enterprise Linux CoreOS (RHCOS)）上构建和部署树外内核模块和驱动程序。内核模块和驱动程序是在操作系统内核中具有高级别权限运行的软件库。它们扩展了内核功能，或者提供控制新设备所需的硬件特定代码。例如，硬件设备，如现场可编程阵列 (FPGA) 或图形处理单元(GPU)，以及软件定义的存储解决方案（客户端机器上需要内核模块）。驱动程序容器是用于在 OpenShift Container Platform 部署中启用这些技术的软件堆栈的第一层。

第 2 章 驱动程序工具包

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

2.1. 关于驱动程序工具包

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

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

Driver Toolkit 中的内核软件包列表包括以下内容及其依赖项：

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

另外，Driver Toolkit 还包含相应的实时内核软件包：

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

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

- **elfutils-libelf-devel**
- **kmod**
- **binutilskabi-dw**
- **kernel-abi-whitelists**
- 以上的依赖项

用途
在出现 Driver Toolkit 之前，用户可以在 OpenShift Container Platform 中的一个 pod 中安装内核软件包，或在构建配置中使用 entitled builds，或从主机 **machine-os-content** 的内核 RPM 进行安装。Driver

Toolkit 通过删除授权步骤简化了流程，并避免了访问 pod 中的 machine-os-content 特权操作。Driver Toolkit 也可以由有权访问预发布的 OpenShift Container Platform 版本的合作伙伴使用，用于未来的 OpenShift Container Platform 版本的硬件设备的预构建 driver-containers。

Kernel Module Management (KMM) 也使用 Driver Toolkit，它目前作为 OperatorHub 上的社区 Operator 提供。KMM 支持树外和第三方内核驱动程序以及底层操作系统的支持软件。用户可以为 KMM 创建模块以构建和部署驱动程序容器，并支持设备插件或指标等软件。模块可以包含构建配置，用于在 Driver Toolkit 上构建基于驱动程序容器的驱动程序，或者 KMM 可以部署预构建驱动程序容器。

2.2. 拉取 DRIVER TOOLKIT 容器镜像

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

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

Red Hat Ecosystem Catalog 包括了使用 **podman** 或 OpenShift Container Platform 从 **registry.redhat.io** 中拉取 **driver-toolkit** 镜像的说明。最新次版本的 driver-toolkit 镜像使用 **registry.redhat.io** 上的次版本标记，例如：**registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.16**。

2.2.2. 在有效负载中查找驱动程序工具包镜像 URL

先决条件

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

流程

1. 使用 **oc adm** 命令提取与特定发行版本对应的 **driver-toolkit** 的镜像 URL：
   - 对于 x86 镜像，命令如下：

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.16.z-x86_64 --image-for=driver-toolkit
     ```

   - 对于 ARM 镜像，命令如下：

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.16.z-aarch64 --image-for=driver-toolkit
     ```

   输**出示例**

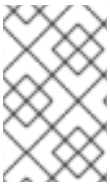
   ```
   quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:b53883ca2bac5925857148c4a1abc300ced96c222498e3bc134fe7ce3a1dd404
   ```

2. 使用有效的 pull secret 获取此镜像，如安装 OpenShift Container Platform 所需的 pull secret：

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

2.3. 使用 DRIVER TOOLKIT

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



注意

Driver Toolkit 包括为内核模块签名所需的依赖项、**openssl**、**mokutil** 和 **keyutils**。但是，在这个示例中，**simple-kmod** 内核模块没有签名，因此无法在启用了安全引导 (**Secure Boot**) 的系统中载入。

2.3.1. 在集群中构建并运行 simple-kmod 驱动程序容器

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 您可以将集群的 Image Registry Operator 状态设置为 **Managed**。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。

流程

创建命名空间。例如：

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

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

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

```
    dockerfile: |
      ARG DTK
      FROM ${DTK} as builder

      ARG KVER

      WORKDIR /build/

      RUN git clone https://github.com/openshift-psap/simple-kmod.git

      WORKDIR /build/simple-kmod

      RUN make all install KVER=${KVER}

      FROM registry.redhat.io/ubi8/ubi-minimal

      ARG KVER

      # Required for installing `modprobe`
      RUN microdnf install kmod

      COPY --from=builder /lib/modules/${KVER}/simple-kmod.ko /lib/modules/${KVER}/
      COPY --from=builder /lib/modules/${KVER}/simple-procfs-kmod.ko
/lib/modules/${KVER}/
      RUN depmod ${KVER}
  strategy:
    dockerStrategy:
      buildArgs:
        - name: KMODVER
          value: DEMO
          # $ oc adm release info quay.io/openshift-release-dev/ocp-release:<cluster version>-
x86_64 --image-for=driver-toolkit
        - name: DTK
          value: quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:34864ccd2f4b6e385705a730864c04a40908e57acede44457a783d739e377cae
        - name: KVER
          value: 4.18.0-372.26.1.el8_6.x86_64
  output:
    to:
      kind: ImageStreamTag
      name: simple-kmod-driver-container:demo
```

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

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

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

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

3. 使用创建镜像流和构建配置

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

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

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

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

```
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/simple-kmod-
demo/simple-kmod-driver-container:demo
        name: simple-kmod-driver-container
        imagePullPolicy: Always
        command: [sleep, infinity]
        lifecycle:
          postStart:
            exec:
              command: ["modprobe", "-v", "-a" , "simple-kmod", "simple-procfs-kmod"]
          preStop:
            exec:
              command: ["modprobe", "-r", "-a" , "simple-kmod", "simple-procfs-kmod"]
        securityContext:
          privileged: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
```

b. 创建 RBAC 规则和守护进程集：

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

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

a. 验证 pod 是否正在运行：

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

输**出示例**

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

b. 在驱动程序容器 pod 中执行 **lsmod** 命令：

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

输**出示例**

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

2.4. 其他资源

- 有关为集群配置 registry 存储的更多信息，请参阅 OpenShift Container Platform 中的 Image Registry Operator。

第 3 章 NODE FEATURE DISCOVERY OPERATOR

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

3.1. 关于 NODE FEATURE DISCOVERY OPERATOR

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

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

3.2. 安装 NODE FEATURE DISCOVERY OPERATOR

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

3.2.1. 使用 CLI 安装 NFD Operator

作为集群管理员，您可以使用 CLI 安装 NFD Operator。

先决条件

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

流程

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

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

   b. 运行以下命令创建命名空间：

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

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

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

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

b. 运行以下命令来创建 **OperatorGroup** CR:

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

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

订阅**示例**

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

d. 运行以下命令来创建订阅对象：

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

e. 进入 **openshift-nfd** 项目：

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

验证

- 要验证 Operator 部署是否成功，请运行：

```
$ oc get pods
```

输**出示例**

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

一个成功的部署会显示 **Running** 状态。

3.2.2. 使用 Web 控制台安装 NFD Operator

作为集群管理员，您可以使用 Web 控制台安装 NFD Operator。

流程

1. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
2. 从可用的 Operator 列表中选择 **Node Feature Discovery**，然后点 **Install**。
3. 在 **Install Operator** 页面中，选择 **A specific namespace on the cluster**，然后点 **Install**。您不需要创建命名空间，因为它已为您创建。

验证

验证 NFD Operator 是否已成功安装：

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

注意

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

故障排除

如果 Operator 没有被安装，请按照以下步骤进行故障排除：

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

3.3. 使用 NODE FEATURE DISCOVERY OPERATOR

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

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

3.3.1. 使用 CLI 创建 NodeFeatureDiscovery 实例

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

先决条件

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

流程

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

```
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: "" # instance is empty by default
  topologyupdater: false # False by default
  operand:
    image: registry.redhat.io/openshift4/ose-node-feature-discovery:v4.16
    imagePullPolicy: Always
  workerConfig:
    configData: |
      core:
      #  labelWhiteList:
      #  noPublish: false
        sleepInterval: 60s
      #  sources: [all]
      #  klog:
      #    addDirHeader: false
      #    alsologtostderr: false
      #    logBacktraceAt:
      #    logtostderr: true
      #    skipHeaders: false
      #    stderrthreshold: 2
      #    v: 0
      #    vmodule:
      ##   NOTE: the following options are not dynamically run-time configurable
      ##         and require a nfd-worker restart to take effect after being changed
      #    logDir:
      #    logFile:
      #    logFileMaxSize: 1800
      #    skipLogHeaders: false
      sources:
        cpu:
          cpuid:
      #     NOTE: whitelist has priority over blacklist
            attributeBlacklist:
              - "BMI1"
              - "BMI2"
              - "CLMUL"
              - "CMOV"
              - "CX16"
              - "ERMS"
              - "F16C"
              - "HTT"
              - "LZCNT"
              - "MMX"
              - "MMXEXT"
              - "NX"
              - "POPCNT"
```

```
            - "RDRAND"
            - "RDSEED"
            - "RDTSCP"
            - "SGX"
            - "SSE"
            - "SSE2"
            - "SSE3"
            - "SSE4.1"
            - "SSE4.2"
            - "SSSE3"
          attributeWhitelist:
      kernel:
        kconfigFile: "/path/to/kconfig"
        configOpts:
          - "NO_HZ"
          - "X86"
          - "DMI"
      pci:
        deviceClassWhitelist:
          - "0200"
          - "03"
          - "12"
        deviceLabelFields:
          - "class"
  customConfig:
    configData: |
        - name: "more.kernel.features"
          matchOn:
          - loadedKMod: ["example_kmod3"]
```

有关如何自定义 NFD worker 的详情，请参考 nfd-worker 的配置文件参考。

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

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

验证

- 要验证是否已创建实例，请运行：

```
$ oc get pods
```

输**出示例**

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

一个成功的部署会显示 **Running** 状态。

3.3.2. 使用 Web 控制台创建 NodeFeatureDiscovery CR

流程

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

3.4. 配置 NODE FEATURE DISCOVERY OPERATOR

3.4.1. core

core 部分包含不特定于任何特定功能源的常见配置设置。

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

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

用法示例

```
core:
  sleepInterval: 60s (1)
```

默认值为 **60s**。

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

如果指定，这个值会被弃用的 **--sources** 命令行标志覆盖。

默认：**[all]**

用法示例

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

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

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

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

默认： **null**

用法示例

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

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

如果指定，**--no-publish** 命令行标志会覆盖这个值。

例如：

用法示例

```
core:
  noPublish: true (1)
```

默认值为 **false**。

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

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

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

默认：**false**

运行时可配置：是

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

默认：**false**

运行时可配置：是

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

默认： **空**

运行时可配置：是

core.klog.logDir
如果非空，在此目录中写入日志文件。

默认： **空**

运行是时配置：否

core.klog.logFile

如果不为空，则使用此日志文件。

默认： **空**

运行是时配置：否

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

默认： **1800**

运行是时配置：否

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

默认： **true**

运行时可配置：是

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

默认：**false**

运行时可配置：是

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

默认：**false**

运行是时配置：否

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

默认： **2**

运行时可配置：是

core.klog.v
core.klog.v 是日志级别详细程度的值。

默认： **0**

运行时可配置：是

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

默认： **空**

运行时可配置：是

3.4.2. sources

sources 部分包含特定于功能源的配置参数。

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

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

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

用法示例

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

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

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

默认： **空**

用法示例

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

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

默认： **空**

用法示例

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

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

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

用法示例

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

sources.pci.deviceClassWhitelist

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

默认： **["03", "0b40", "12"]**

用法示例

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

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

默认： **[class, vendor]**

用法示例

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

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

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

默认： **["0e", "ef", "fe", "ff"]**

用法示例

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

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

默认： **[class、vendor、device]**

用法示例

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

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

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

默认： **空**

用法示例

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

3.5. 关于 NODEFEATURERULE 自定义资源

NodeFeatureRule 对象是一个 **NodeFeatureDiscovery** 自定义资源，专为基于规则的自定义标签节点而设计。有些用例包括特定于应用程序的标记或厂商分布，以便为其设备创建特定标签。

NodeFeatureRule 对象提供了一种创建特定厂商或特定于应用程序的标签和污点的方法。它使用灵活的基于规则的机制来创建标签，并根据节点功能选择性地污点。

3.6. 使用 NODEFEATURERULE 自定义资源

如果一组规则与条件匹配，创建一个 **NodeFeatureRule** 对象来标记节点。

流程

1. 创建名为 **nodefeaturerule.yaml** 的自定义资源文件，其中包含以下文本：

```
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureRule
metadata:
  name: example-rule
spec:
  rules:
    - name: "example rule"
      labels:
        "example-custom-feature": "true"
      # Label is created if all of the rules below match
      matchFeatures:
        # Match if "veth" kernel module is loaded
        - feature: kernel.loadedmodule
          matchExpressions:
            veth: {op: Exists}
        # Match if any PCI device with vendor 8086 exists in the system
        - feature: pci.device
          matchExpressions:
            vendor: {op: In, value: ["8086"]}
```

此自定义资源指定在加载 **veth** 模块且集群中存在厂商代码 **8086** 的任何 PCI 设备时发生标记。

2. 运行以下命令，将 **nodefeaturerule.yaml** 文件应用到集群：

```
$ oc apply -f https://raw.githubusercontent.com/kubernetes-sigs/node-feature-discovery/v0.13.6/examples/nodefeaturerule.yaml
```

这个示例在载入了 **veth** 模块的节点上应用 feature 标签，以及存在厂商代码 **8086** 的任何 PCI 设备。

注意

可能会出现最多 1 分钟的重新标记延迟。

3.7. 使用 NFD TOPOLOGY UPDATER

Node Feature Discovery(NFD)Topology Updater 是一个守护进程，负责检查 worker 节点上分配的资源。它考虑可以为每个区分配给新 pod 的资源，其中区域可以是 Non-Uniform Memory Access(NUMA)节点。NFD Topology Updater 将信息发送到 nfd-master，它会创建一个与集群中的所有 worker 节点对应的 **NodeResourceTopology** 自定义资源(CR)。NFD Topology Updater 其中一个实例在集群的每个节点上运行。

要在 NFD 中启用 Topology Updater worker，将 **NodeFeatureDiscovery** CR 中的 **topologyupdater** 变量设置为 **true**，如使用 **Node Feature Discovery Operator** 一节中所述。

3.7.1. NodeResourceTopology CR

使用 NFD Topology Updater 时，NFD 会创建与节点资源硬件拓扑对应的自定义资源实例，例如：

```
apiVersion: topology.node.k8s.io/v1alpha1
kind: NodeResourceTopology
metadata:
  name: node1
topologyPolicies: ["SingleNUMANodeContainerLevel"]
zones:
  - name: node-0
    type: Node
    resources:
      - name: cpu
        capacity: 20
        allocatable: 16
        available: 10
      - name: vendor/nic1
        capacity: 3
        allocatable: 3
        available: 3
  - name: node-1
    type: Node
    resources:
      - name: cpu
        capacity: 30
        allocatable: 30
        available: 15
      - name: vendor/nic2
        capacity: 6
        allocatable: 6
```

```
      available: 6
  - name: node-2
    type: Node
    resources:
      - name: cpu
        capacity: 30
        allocatable: 30
        available: 15
      - name: vendor/nic1
        capacity: 3
        allocatable: 3
        available: 3
```

3.7.2. NFD Topology Updater 命令行标志

要查看可用的命令行标志，请运行 **nfd-topology-updater -help** 命令。例如，在 podman 容器中，运行以下命令：

```
$ podman run gcr.io/k8s-staging-nfd/node-feature-discovery:master nfd-topology-updater -help
```

-ca-file
-ca-file 标志是用于控制 NFD Topology Updater 上的 mutual TLS 身份验证的三个标记之一，其他两个是 **-cert-file** 和 '-key-file'。此标志指定用于验证 nfd-master 真实性的 TLS root 证书。

默认： 空

重要

-ca-file 标志必须与 **-cert-file** 和 **-key-file** 标志一起指定。

Example

```
$ nfd-topology-updater -ca-file=/opt/nfd/ca.crt -cert-file=/opt/nfd/updater.crt -key-
file=/opt/nfd/updater.key
```

-cert-file
-cert-file 标志是在 NFD Topology Updater 上控制 mutual TLS 身份验证的三个标记之一，其他两个与 **-ca-file** 和 **-key-file flags**。此标志指定为身份验证传出请求的 TLS 证书。

默认： 空

重要

-cert-file 标志必须与 **-ca-file** 和 **-key-file** 标志一起指定。

Example

```
$ nfd-topology-updater -cert-file=/opt/nfd/updater.crt -key-file=/opt/nfd/updater.key -ca-
file=/opt/nfd/ca.crt
```

-h, -help
打印使用方法并退出.

-key-file
key-file 标志是控制 NFD Topology Updater 上的 mutual TLS 身份验证的三个标记之一，其他两个是 **-ca-file** 和 **-cert-file**。此标志指定与给定证书文件或 **-cert-file** 对应的私钥，用于验证传出请求。

默认： 空

重要

key-file 标志必须与 **-ca-file** 和 **-cert-file** 标志一起指定。

Example

```
$ nfd-topology-updater -key-file=/opt/nfd/updater.key -cert-file=/opt/nfd/updater.crt -ca-file=/opt/nfd/ca.crt
```

-kubelet-config-file
-kubelet-config-file 指定到 Kubelet 配置文件的路径。

默认：**/host-var/lib/kubelet/config.yaml**

Example

```
$ nfd-topology-updater -kubelet-config-file=/var/lib/kubelet/config.yaml
```

-no-publish
-no-publish 标志禁用与 nfd-master 的所有通信，使其成为 nfd-topology-updater 的空运行标记。NFD Topology Updater 会正常运行资源硬件拓扑检测，但不会将 CR 请求发送到 nfd-master。

默认：**false**

Example

```
$ nfd-topology-updater -no-publish
```

3.7.2.1. -oneshot

-oneshot 标志会导致 NFD Topology Updater 在传递资源硬件拓扑检测后退出。

默认：**false**

Example

```
$ nfd-topology-updater -oneshot -no-publish
```

-podresources-socket
-podresources-socket 标志指定 Unix 套接字的路径，其中 kubelet 会导出 gRPC 服务来启用使用中的 CPU 和设备的发现，并为它们提供元数据。

默认：**/host-var/liblib/kubelet/pod-resources/kubelet.sock**

Example

```
$ nfd-topology-updater -podresources-socket=/var/lib/kubelet/pod-resources/kubelet.sock
```

-

-server
-server 标志指定要连接到的 nfd-master 端点的地址。

默认：**localhost:8080**

Example

```
$ nfd-topology-updater -server=nfd-master.nfd.svc.cluster.local:443
```

-server-name-override
-server-name-override 标志指定从 nfd-master TLS 证书期望的通用名称(CN)。这个标志主要用于开发和调试目的。

默认： 空

Example

```
$ nfd-topology-updater -server-name-override=localhost
```

-sleep-interval
-sleep-interval 标志指定资源硬件拓扑重新检查和自定义资源更新之间的间隔。非正数值意味着睡眠间隔无限，不会进行重新检测。

默认：**60s**

Example

```
$ nfd-topology-updater -sleep-interval=1h
```

-version
打印版本并退出。

-watch-namespace
watch-namespace 标志指定命名空间，以确保仅在指定命名空间中运行的容器集发生资源硬件拓扑考试。在资源核算过程中不考虑在指定命名空间中运行的 Pod。这对于测试和调试目的特别有用。* 值表示所有命名空间中的所有 pod 在计数过程中都会考虑。

默认：*

Example

```
$ nfd-topology-updater -watch-namespace=rte
```

第 4 章 内核模块管理 OPERATOR

了解内核模块管理(KMM) Operator，以及如何使用它在 OpenShift Container Platform 集群上部署树外内核模块和设备插件。

4.1. 关于内核模块管理 OPERATOR

Kernel Module Management (KMM) Operator 在 OpenShift Container Platform 集群中管理、构建、签名和部署树外内核模块和设备插件。

KMM 添加一个新的 **Module** CRD，它描述了树外内核模块及其关联的设备插件。您可以使用 **Module** 资源配置如何加载模块，为内核版本定义 **ModuleLoader** 镜像，并包含为特定内核版本构建和签名模块的说明。

KMM 旨在针对任何内核模块一次性容纳多个内核版本，允许无缝节点升级并减少应用程序停机时间。

4.2. 安装内核模块管理 OPERATOR

作为集群管理员，您可以使用 OpenShift CLI 或 Web 控制台安装内核模块管理(KMM) Operator。

OpenShift Container Platform 4.12 及更新的版本支持 KMM Operator。在版本 4.11 上安装 KMM 不需要具体附加步骤。有关在版本 4.10 及更早版本上安装 KMM 的详情，请参阅"在早期版本的 OpenShift Container Platform 上安装 Kernel Module Management Operator 部分"。

4.2.1. 使用 Web 控制台安装 Kernel Module Management Operator

作为集群管理员，您可以使用 OpenShift Container Platform Web 控制台安装 Kernel Module Management (KMM) Operator。

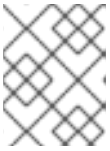
流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 安装内核模块管理 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 **Kernel Module Management Operator**，然后点 **Install**。
   c. 在 **Installed Namespace** 列表中，选择 **openshift-kmm** 命名空间。
   d. 点 **Install**。

验证

验证 KMM Operator 是否已成功安装：

1. 进入到 **Operators → Installed Operators** 页面。
2. 确保 **openshift-kmm** 项目中列出的 **Kernel Module Management Operator** 的 **Status** 为 **InstallSucceeded**。



注意

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

故障排除

1. 排除安装 Operator 的问题：
   a. 导航到 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。
   b. 进入到 **Workloads → Pods** 页面，在 **openshift-kmm** 项目中检查 pod 的日志。

4.2.2. 使用 CLI 安装内核模块管理 Operator

作为集群管理员，您可以使用 OpenShift CLI 安装内核模块管理 (KMM) Operator。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。

流程

1. 在 **openshift-kmm** 命名空间中安装 KMM:
   a. 创建以下 **Namespace** CR 并保存 YAML 文件，如 **kmm-namespace.yaml** ：

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

   b. 创建以下 **OperatorGroup** CR 并保存 YAML 文件，如 **kmm-op-group.yaml** ：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

   c. 创建以下 **Subscription** CR 并保存 YAML 文件，如 **kmm-sub.yaml** ：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
spec:
  channel: release-1.0
  installPlanApproval: Automatic
```

```
name: kernel-module-management
source: redhat-operators
sourceNamespace: openshift-marketplace
startingCSV: kernel-module-management.v1.0.0
```

d. 运行以下命令来创建订阅对象：

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

验证

- 要验证 Operator 部署是否成功，请运行以下命令：

```
$ oc get -n openshift-kmm deployments.apps kmm-operator-controller
```

输出示例

```
NAME                          READY UP-TO-DATE  AVAILABLE AGE
kmm-operator-controller       1/1   1           1         97s
```

Operator 可用。

4.2.3. 在早期版本的 OpenShift Container Platform 上安装 Kernel Module Management Operator

OpenShift Container Platform 4.12 及更新的版本支持 KMM Operator。对于版本 4.10 及更早版本，您必须创建一个新的 **SecurityContextConstraint** 对象，并将其绑定到 Operator 的 **ServiceAccount**。作为集群管理员，您可以使用 OpenShift CLI 安装内核模块管理 (KMM) Operator。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。

流程

1. 在 **openshift-kmm** 命名空间中安装 KMM:

   a. 创建以下 **Namespace** CR 并保存 YAML 文件，如 **kmm-namespace.yaml** 文件：

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

   b. 创建以下 **SecurityContextConstraint** 对象并保存 YAML 文件，如 **kmm-security-constraint.yaml** ：

```
allowHostDirVolumePlugin: false
allowHostIPC: false
```

```
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: false
allowPrivilegedContainer: false
allowedCapabilities:
  - NET_BIND_SERVICE
apiVersion: security.openshift.io/v1
defaultAddCapabilities: null
fsGroup:
  type: MustRunAs
groups: []
kind: SecurityContextConstraints
metadata:
  name: restricted-v2
priority: null
readOnlyRootFilesystem: false
requiredDropCapabilities:
  - ALL
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
seccompProfiles:
  - runtime/default
supplementalGroups:
  type: RunAsAny
users: []
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - persistentVolumeClaim
  - projected
  - secret
```

c. 运行以下命令，将 **SecurityContextConstraint** 对象绑定到 Operator 的 **ServiceAccount** ：

```
$ oc apply -f kmm-security-constraint.yaml
```

```
$ oc adm policy add-scc-to-user kmm-security-constraint -z kmm-operator-controller -n openshift-kmm
```

d. 创建以下 **OperatorGroup** CR 并保存 YAML 文件，如 **kmm-op-group.yaml** ：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

e. 创建以下 **Subscription** CR 并保存 YAML 文件，如 **kmm-sub.yaml** ：

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
spec:
  channel: release-1.0
  installPlanApproval: Automatic
  name: kernel-module-management
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: kernel-module-management.v1.0.0
```

f. 运行以下命令来创建订阅对象：

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

验证

- 要验证 Operator 部署是否成功，请运行以下命令：

```
$ oc get -n openshift-kmm deployments.apps kmm-operator-controller
```

输**出示例**

```
NAME                          READY UP-TO-DATE  AVAILABLE AGE
kmm-operator-controller       1/1   1           1         97s
```

Operator 可用。

4.3. 配置内核模块管理 OPERATOR

在大多数情况下，内核模块管理(KMM) Operator 的默认配置不需要修改。但是，您可以按照以下流程修改 Operator 设置以适应您的环境。

Operator 配置在 Operator 命名空间中的 **kmm-operator-manager-config ConfigMap** 中设置。

流程

1. 要修改设置，请输入以下命令编辑 **ConfigMap** 数据：

```
$ oc edit configmap -n "$namespace" kmm-operator-manager-config
```

输**出示例**

```
healthProbeBindAddress: :8081
job:
  gcDelay: 1h
leaderElection:
  enabled: true
  resourceID: kmm.sigs.x-k8s.io
webhook:
```

```
    disableHTTP2: true  # CVE-2023-44487
    port: 9443
  metrics:
    enableAuthnAuthz: true
    disableHTTP2: true  # CVE-2023-44487
    bindAddress: 0.0.0.0:8443
    secureServing: true
  worker:
    runAsUser: 0
    seLinuxType: spc_t
    setFirmwareClassPath: /var/lib/firmware
```

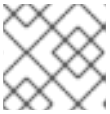
表 4.1. Operator 配置参数

参数	描述
healthProbeBindAddress	定义 Operator 监控 kubelet 健康探测的地址。推荐的值为 **:8081**。
job.gcDelay	定义在删除之前，应保留成功构建 pod 的持续时间。此设置没有推荐的值。有关此设置的有效值的详情，请参考 ParseDuration。
leaderElection.enabled	确定领导选举机制是否被用来确保任何时候都只有一个 KMM Operator 运行的副本。如需更多信息，请参阅 Leases。推荐的值为 **true**。
leaderElection.resourceID	确定领导选举机制用于存放领导锁的资源名称。推荐的值为 **kmm.sigs.x-k8s.io**。
webhook.disableHTTP2	如果为 **true**，则禁用 webhook 服务器的 HTTP/2，作为cve-2023-44487 的缓解方案。推荐的值为 **true**。
webhook.port	定义 Operator 监控 Webhook 请求的端口。推荐的值为 **9443**。
metrics.enableAuthnAuthz	确定指标是否使用 **TokenReviews** 验证，并在 kube-apiserver 中使用 **SubjectAccessReviews** 授权。 对于身份验证和授权，控制器需要一个具有以下规则的 **ClusterRole** ： • **apiGroups: authentication.k8s.io, resources: tokenreviews, verbs: create** • **apiGroups: authorization.k8s.io, resources: subjectaccessreviews, verbs: create** 例如，要提取指标（例如使用 Prometheus），客户端需要具有以下规则的 **ClusterRole** ： • **nonResourceURLs: "/metrics", verbs: get** 推荐的值为 **true**。
metrics.disableHTTP2	如果为 **true**，则禁用 metrics 服务器的 HTTP/2 作为CVE-2023-44487 的缓解方案。推荐的值为 **true**。

参数	描述
metrics.bindAddress	决定指标服务器的绑定地址。如果未指定，则默认为 **:8080**。要禁用指标服务器，设置为 **0**。推荐的值为 **0.0.0.0:8443**。
metrics.secureServing	决定是否通过 HTTPS 而不是 HTTP 提供指标。推荐的值为 **true**。
worker.runAsUser	决定 worker 容器安全上下文的 **runAsUser** 字段的值。如需更多信息，请参阅 SecurityContext。推荐的值为 **9443**。
worker.seLinuxType	决定 worker 容器安全上下文的 **seLinuxOptions.type** 字段的值。如需更多信息，请参阅 SecurityContext。推荐的值为 **spc_t**。
worker.setFirmwareClassPath	将内核的固件搜索路径设置为节点上的 **/sys/module/firmware_class/parameters/path** 文件中。如果您需要通过 worker 应用程序设置该值，则推荐的值为 **/var/lib/firmware**。否则，取消设置。

2. 修改设置后，使用以下命令重启控制器：

```
$ oc delete pod -n "<namespace>" -l app.kubernetes.io/component=kmm
```



注意

<namespace> 的值取决于您的原始安装方法。

其他资源

- 如需更多信息，请参阅 安装内核模块管理 Operator。

4.3.1. 卸载内核模块

在移至较新的版本时，您必须卸载内核模块，或者如果它们在节点上引入一些不必要的副作用。

流程

- 要从节点中卸载使用 KMM 加载的模块，请删除对应的 **Module** 资源。然后，KMM 会根据需要创建 worker pod，以运行 **modprobe -r** 并从节点卸载内核模块。

警告

卸载 worker pod 时，KMM 需要载入内核模块时使用的所有资源。这包括 **模块** 中引用的 **ServiceAccount**，以及定义的任何 RBAC，以允许特权 KMM worker Pod 运行。它还包括 **.spec.imageRepoSecret** 中引用的任何 pull secret。

为了避免 KMM 无法从节点卸载内核模块的情况，请确保在 **Module** 资源仍存在于任何状态时不会删除这些资源，包括 **Terminating**。KMM 包含一个验证准入 Webhook，它拒绝删除至少包含一个 **Module** 资源的命名空间。

4.3.2. 设置内核固件搜索路径

Linux 内核接受 **firmware_class.path** 参数作为固件的搜索路径，如 固件搜索路径 中所述。

在尝试加载 kmods 前，KMM worker pod 可以通过写入 sysfs 来在节点上设置这个值。

流程

- 要定义固件搜索路径，请在 Operator 配置中将 **worker.setFirmwareClassPath** 设置为 **/var/lib/firmware**。

其他资源

- 有关 **worker.setFirmwareClassPath** 路径的更多信息，请参阅配置内核模块管理 Operator。

4.4. 卸载内核模块管理 OPERATOR

根据 KMM Operator 的安装方式，使用以下流程之一卸载内核模块管理(KMM) Operator。

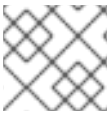
4.4.1. 卸载红帽目录安装

如果从红帽目录中安装了 KMM，请使用这个步骤。

流程

使用以下方法卸载 KMM Operator：

- 在 OpenShift 控制台的 **Operators** -→ **Installed Operators** 下找到并卸载 Operator。



注意

另外，您可以删除 KMM 命名空间中的 **Subscription** 资源。

4.4.2. 卸载 CLI 安装

如果使用 OpenShift CLI 安装 KMM Operator，请使用此命令。

流程

- 运行以下命令来卸载 KMM Operator：

```
$ oc delete -k https://github.com/rh-ecosystem-edge/kernel-module-
management/config/default
```

注意

使用此命令删除集群中的 **Module** CRD 和所有 **Module** 实例。

4.5. 内核模块部署

内核模块管理 (KMM) 监控集群中的 **Node** 和 **Module** 资源，以确定是否应该从节点上载入内核模块。

要有资格获得模块，节点必须包含以下内容：

- 与模块的 **.spec.selector** 字段匹配的标签。
- 与模块的 **.spec.moduleLoader.container.kernelMappings** 字段中某一项匹配的内核版本。
- 如果在模块中配置了排序升级 (**ordered_upgrade.md**)，则标签与 **.spec.moduleLoader.container.version** 字段匹配。

当 KMM 将节点与在 **Module** 资源中配置的所需状态协调时，它会在目标节点上创建 worker pod 以运行必要的操作。KMM Operator 监控 pod 的结果并记录信息。Operator 使用此信息在模块成功加载时标记 **Node** 对象，并在配置后运行设备插件。

worker pod 运行 KMM **worker** 二进制文件，它执行以下任务：

- 拉取 **Module** 资源中配置的 kmod 镜像。kmod 镜像是包含 **.ko** 文件的标准 OCI 镜像。
- 在 pod 文件系统中提取镜像。
- 使用指定的参数运行 **modprobe**，以执行必要的操作。

4.5.1. 模块自定义资源定义

Module 自定义资源定义 (CRD) 代表可通过 kmod 镜像在所有或选择集群中载入的内核模块。 **Module** 自定义资源 (CR) 指定一个或多个兼容它的内核版本，以及一个节点选择器。

Module 资源的兼容版本列在 **.spec.moduleLoader.container.kernelMappings** 下。内核映射可以与 **literal** 版本匹配，也可以使用 **regexp** 同时匹配其中的许多版本。

Module 资源的协调循环运行以下命令：

1. 列出与 **.spec.selector** 匹配的所有节点。
2. 构建在这些节点上运行的所有内核版本。
3. 对于每个内核版本：
    a. 进入 **.spec.moduleLoader.container.kernelMappings**，并找到适当的容器镜像名称。如果内核映射定义了 **build** 或 **sign**，且容器镜像尚不存在，请根据需要运行构建、签名 pod 或两者。
    b. 创建 worker pod 以拉取上一步中确定的容器镜像并运行 **modprobe**。

c. 如果定义了 **.spec.devicePlugin**，请使用 **.spec.devicePlugin.container** 中指定的配置创建一个设备插件守护进程集。

4. 在以下运行 **garbage-collect**：

a. 过时的设备插件 **DaemonSet** 不以任何节点为目标。

b. 成功构建 pod。

c. 成功签名 pod。

4.5.2. 在内核模块之间设置软依赖项

有些配置要求以特定顺序加载多个内核模块才能正常工作，即使模块不会直接依赖于通过符号相互依赖。它们称为软依赖项。**depmod** 通常不知道这些依赖项，它们不会出现在它生成的文件中。例如，如果 **mod_a** 有一个软依赖项 **mod_b**，**modprobe mod_a** 不会加载 **mod_b**。

您可以使用 **modulesLoadingOrder** 字段在 Module 自定义资源定义(CRD)中声明软依赖项来解决这些情况。

```
# ...
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: mod_a
        dirName: /opt
        firmwarePath: /firmware
        parameters:
          - param=1
        modulesLoadingOrder:
          - mod_a
          - mod_b
```

在上面的配置中，worker pod 首先会尝试在从 kmod 镜像加载 **mod_a** 前卸载 in-tree **mod_b**。当 worker pod 终止并卸载 **mod_a** 时，**mod_b** 不会再次加载。

注意

列表中的第一个值（最后加载）必须与 **moduleName** 等效。

4.6. 安全和权限

重要

加载内核模块是一个高度敏感的操作。加载后，内核模块具有在节点上执行任何类型的操作的所有可能权限。

4.6.1. ServiceAccounts 和 SecurityContextConstraints

内核模块管理 (KMM) 创建一个特权工作负载，以在节点上加载内核模块。该工作负载需要 **ServiceAccounts** 被允许来使用 **privileged SecurityContextConstraint** (SCC) 资源。

该工作负载的授权模型取决于 **Module** 资源的命名空间及其 spec。

- 如果设置了 **.spec.moduleLoader.serviceAccountName** 或 **.spec.devicePlugin.serviceAccountName** 字段，则始终使用它们。
- 如果没有设置这些字段，则：
  - 如果在 Operator 命名空间中创建了 **Module** 资源（默认为**openshift-kmm** ），则 KMM 使用其默认的、强大的 **ServiceAccount** 来运行 worker 和设备插件 Pod。
  - 如果在任何其他命名空间中创建了 **Module** 资源，则 KMM 会使用命名空间的 **default ServiceAccount** 运行 pod。**Module** 资源无法运行特权工作负载，除非您手动启用它以使用 **privileged** SCC。

重要

openshift-kmm 是一个可信命名空间。

在设置 RBAC 权限时，请记住在 **openshift-kmm** 命名空间中创建 **Module** 资源的任何用户或 **ServiceAccount** 都会导致 KMM 在集群中的任何节点上运行特权工作负载。

要允许任何 **ServiceAccount** 使用 **特权** SCC 并运行 worker 或设备插件 pod，您可以使用 **oc adm policy** 命令，如下例所示：

```
$ oc adm policy add-scc-to-user privileged -z "${serviceAccountName}" [ -n "${namespace}" ]
```

4.6.2. Pod 安全标准

OpenShift 运行一个同步机制，它根据使用的安全上下文自动设置命名空间 Pod 安全级别。不需要操作。

其他资源

- 了解并管理 pod 安全准入

4.7. 使用树外模块替换树内模块

您可以使用内核模块管理(KMM)构建可按需载入或卸载到内核的内核模块。这些模块可以在不需要重启系统的情况下扩展内核的功能。模块可以配置为内置或动态加载。

动态加载的模块包括树内模块和树外(OOT)模块。in-tree 模块是 Linux 内核树的内部，即它们已经是内核的一部分。树外模块是 Linux 内核树的外部。它们通常是为开发和测试目的编写的，例如测试树级或处理不兼容的内核模块的新版本。

由 KMM 加载的一些模块可能会替换节点上已经载入的树内模块。要在载入模块前卸载树内模块，请将 **.spec.moduleLoader.container.inTreeModulesToRemove** 字段设置为您要卸载的模块。以下示例演示了所有内核映射的模块替换：

```
# ...
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: mod_a

      inTreeModulesToRemove: [mod_a, mod_b]
```

在本例中，**moduleLoader** pod 使用 **inTreeModulesToRemove** 在 **moduleLoader** 镜像加载 **mod_a** 前卸载 in-tree **mod_a** 和 **mod_b**。当 **moduleLoader`pod 被终止，`mod_a** 被卸载，则 **mod_b** 不会被再次加载。

以下是针对特定内核映射的模块替换示例：

```
# ...
spec:
  moduleLoader:
    container:
      kernelMappings:
        - literal: 6.0.15-300.fc37.x86_64
          containerImage: "some.registry/org/my-kmod:${KERNEL_FULL_VERSION}"
          inTreeModulesToRemove: [<module_name>, <module_name>]
```

其他资源

- 构建 linux 内核模块

4.7.1. 模块 CR 示例

以下是一个注解的 **Module** 示例：

```
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: <my_kmod>
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: <my_kmod> (1)
        dirName: /opt (2)
        firmwarePath: /firmware (3)
        parameters: (4)
          - param=1
      kernelMappings: (5)
        - literal: 6.0.15-300.fc37.x86_64
          containerImage: some.registry/org/my-kmod:6.0.15-300.fc37.x86_64
        - regexp: '^.+\fc37\.x86_64$' (6)
          containerImage: "some.other.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
        - regexp: '^.+$' (7)
          containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
          build:
            buildArgs: (8)
              - name: ARG_NAME
                value: <some_value>
            secrets:
              - name: <some_kubernetes_secret> (9)
            baseImageRegistryTLS: (10)
              insecure: false
              insecureSkipTLSVerify: false (11)
            dockerfileConfigMap: (12)
```

```
          name: <my_kmod_dockerfile>
        sign:
          certSecret:
            name: <cert_secret> (13)
          keySecret:
            name: <key_secret> (14)
          filesToSign:
            - /opt/lib/modules/${KERNEL_FULL_VERSION}/<my_kmod>.ko
        registryTLS: (15)
          insecure: false (16)
          insecureSkipTLSVerify: false
    serviceAccountName: <sa_module_loader> (17)
  devicePlugin: (18)
    container:
      image: some.registry/org/device-plugin:latest (19)
      env:
        - name: MY_DEVICE_PLUGIN_ENV_VAR
          value: SOME_VALUE
      volumeMounts: (20)
        - mountPath: /some/mountPath
          name: <device_plugin_volume>
    volumes: (21)
      - name: <device_plugin_volume>
        configMap:
          name: <some_configmap>
    serviceAccountName: <sa_device_plugin> (22)
  imageRepoSecret: (23)
    name: <secret_name>
  selector:
    node-role.kubernetes.io/worker: ""
```

(1) (1) (1) 必需。

(2) 可选。

(3) 可选：在节点上将 **/firmware/*** 复制到 **/var/lib/firmware/**。

(4) 可选。

(5) 至少需要一个内核项。

(6) 对于运行与正则表达式匹配的内核的每个节点，KMM 创建一个 **DaemonSet** 资源，运行 **containerImage** 中指定的镜像，使用 **${KERNEL_FULL_VERSION}** 替换为内核版本。

(7) 对于任何其他内核，使用 **my-kmod** ConfigMap 中的 Dockerfile 构建镜像。

(8) 可选。

(9) 可选： **some-kubernetes-secret** 的值可以从位于 **/run/secrets/some-kubernetes-secret** 的构建环境中获取。

(10) 此字段无效。在 kmod 镜像中构建 kmod 镜像或签名 kmod 时，您可能需要从提供由不可信证书颁发机构 (CA) 签名的证书的 registry 中拉取基础镜像。要让 KMM 信任该 CA，还必须通过替换集群的 CA 捆绑包来信任新的 CA。

请参阅"附加资源"以了解如何替换集群的 CA 捆绑包。

11 可选：避免使用此参数。如果设置为 **true**，构建将在使用普通 HTTP 在 Dockerfile **FROM** 指令中拉取镜像时跳过任何 TLS 服务器证书验证。

12 必需。

13 必需：包含带有密钥"证书"的公钥的 secret。

14 必需：包含带有密钥"密钥"的私有 secureboot 密钥的 secret。

15 可选：避免使用此参数。如果设置为 **true**，则允许 KMM 检查容器镜像是否已使用普通 HTTP。

16 可选：避免使用此参数。如果设置为 **true**，KMM 会在检查容器镜像是否已存在时跳过任何 TLS 服务器证书验证。

17 可选。

18 可选。

19 必需：如果存在设备插件部分。

20 可选。

21 可选。

22 可选。

23 可选：用于拉取模块加载程序和设备插件镜像。

其他资源

- 替换 CA Bundle 证书

4.8. 树内依赖项的符号链接

有些内核模块依赖于节点操作系统附带的其他内核模块。为了避免将这些依赖项复制到 kmod 镜像中，内核模块管理 (KMM) 将 **/usr/lib/modules** 挂载到构建和 worker pod 文件系统中。

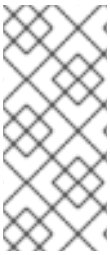
通过创建从 **/opt/usr/lib/modules/<kernel_version>/<symlink_name>** 到 **/usr/lib/modules/<kernel_version>** 的符号链接，**depmod** 可以使用构建节点文件系统上的 in-tree kmods 来解析依赖项。

在运行时，worker pod 会提取整个镜像，包括 **<symlink_name>** 符号链接。该符号链接指向 worker pod 中的 **/usr/lib/modules/<kernel_version>**，该链接从节点的文件系统挂载。**modprobe** 然后可以跟随该链接，并根据需要加载树内依赖项。

在以下示例中，**host** 是 **/opt/usr/lib/modules/<kernel_version>** 下的符号链接名称：

```
ARG DTK_AUTO

FROM ${DTK_AUTO} as builder

#
# Build steps
```

```
#

FROM ubi9/ubi

ARG KERNEL_FULL_VERSION

RUN dnf update && dnf install -y kmod

COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/

# Create the symbolic link
RUN ln -s /lib/modules/${KERNEL_FULL_VERSION}
/opt/lib/modules/${KERNEL_FULL_VERSION}/host

RUN depmod -b /opt ${KERNEL_FULL_VERSION}
```



注意

depmod 根据运行 kmod 镜像构建的节点上存在的内核模块生成依赖项文件。

在 KMM 加载内核模块的节点上，**modprobe** 要求文件存在于 **/usr/lib/modules/<kernel_version>** 下，以及相同的文件系统布局。强烈建议您构建和目标节点共享相同的操作系统和发行版本。

4.9. 创建 KMOD 镜像

内核模块管理 (KMM) 与专用的 kmod 镜像一起工作，它们是包含 **.ko** 文件的标准 OCI 镜像。**.ko** 文件的位置必须与以下模式匹配：**<prefix>/lib/modules/[kernel-version]/**。

在使用 **.ko** 文件时请注意以下几点：

- 在大多数情况下，**<prefix>** 应该等于 **/opt**。这是 **Module** CRD 的默认值。
- **kernel-version** 不能为空，且必须与为构建内核模块的内核版本相同。

4.9.1. 运行 depmod

建议您在构建过程结束时运行 **depmod** 来生成 **modules.dep** 和 **.map** 文件。如果您的 kmod 镜像包含多个内核模块，并且其中一个模块依赖于另一个模块，则这特别有用。

注意

您必须有一个红帽订阅才能下载 **kernel-devel** 软件包。

流程

- 运行以下命令，为特定内核版本生成 **modules.dep** 和 **.map** 文件：

```
$ depmod -b /opt ${KERNEL_FULL_VERSION}+`.
```

4.9.1.1. Dockerfile 示例

如果要在 OpenShift Container Platform 上构建镜像，请考虑使用 Driver Tool Kit (DTK)。

如需更多信息，请参阅使用授权构建。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kmm-ci-dockerfile
data:
  dockerfile: |
    ARG DTK_AUTO
    FROM ${DTK_AUTO} as builder
    ARG KERNEL_FULL_VERSION
    WORKDIR /usr/src
    RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
    WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
    RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_FULL_VERSION}/build make all
    FROM registry.redhat.io/ubi9/ubi-minimal
    ARG KERNEL_FULL_VERSION
    RUN microdnf install kmod
    COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/
    COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/
    RUN depmod -b /opt ${KERNEL_FULL_VERSION}
```

其他资源

- 驱动程序工具包

4.9.2. 在集群中构建

KMM 可以在集群中构建 kmod 镜像。按照以下准则：

- 使用内核映射的 **build** 部分提供构建说明。
- 将容器镜像的 **Dockerfile** 复制到 **dockerfile** 键下的 **ConfigMap** 资源中。
- 确保 **ConfigMap** 位于与 **Module** 相同的命名空间中。

KMM 检查 **containerImage** 字段中指定的镜像名称是否存在。如果存在，则会跳过构建。

否则，KMM 创建一个 **Build** 资源来构建您的镜像。构建镜像后，KMM 继续进行 **Module** 协调。请参见以下示例。

```
# ...
- regexp: '^.+$'
  containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
  build:
    buildArgs: (1)
      - name: ARG_NAME
        value: <some_value>
    secrets: (2)
```

```
    - name: <some_kubernetes_secret> 3
  baseImageRegistryTLS:
    insecure: false 4
    insecureSkipTLSVerify: false 5
  dockerfileConfigMap: 6
    name: <my_kmod_dockerfile>
registryTLS:
  insecure: false 7
  insecureSkipTLSVerify: false 8
```

1. 可选。
2. 可选。
3. 将以 **/run/secrets/some-kubernetes-secret** 的形式挂载到构建 Pod 中。
4. 可选：避免使用此参数。如果设置为 **true**，则允许构建使用普通 HTTP 在 Dockerfile **FROM** 指令中拉取镜像。
5. 可选：避免使用此参数。如果设置为 **true**，构建将在使用普通 HTTP 在 Dockerfile **FROM** 指令中拉取镜像时跳过任何 TLS 服务器证书验证。
6. 必需。
7. 可选：避免使用此参数。如果设置为 **true**，则允许 KMM 检查容器镜像是否已使用普通 HTTP。
8. 可选：避免使用此参数。如果设置为 **true**，KMM 会在检查容器镜像是否已存在时跳过任何 TLS 服务器证书验证。

成功构建 pod 会立即收集垃圾回收，除非 Operator 配置中设置了 **job.gcDelay** 参数。失败的构建 pod 始终会被保留，且必须由管理员手动删除，才能重启构建。

其他资源

- 构建配置资源
- preflight 验证内核模块管理 (KMM) 模块

4.9.3. 使用 Driver Toolkit

Driver Toolkit (DTK) 是一个便捷的基础镜像，用于构建构建 kmod loader 镜像。它包含集群中当前运行的 OpenShift 版本的工具和库。

流程

使用 DTK 作为多阶段 **Dockerfile** 的第一个阶段。

1. 构建内核模块。
2. 将 **.ko** 文件复制到较小的最终用户镜像中，如 **ubi-minimal**。
3. 要在集群内构建中使用 DTK，请使用 **DTK_AUTO** 构建参数。在创建 **Build** 资源时，该值由 KMM 自动设置。请参见以下示例。

```
ARG DTK_AUTO
```

```
FROM ${DTK_AUTO} as builder
ARG KERNEL_FULL_VERSION
WORKDIR /usr/src
RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_FULL_VERSION}/build make all
FROM ubi9/ubi-minimal
ARG KERNEL_FULL_VERSION
RUN microdnf install kmod
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_FULL_VERSION}/
RUN depmod -b /opt ${KERNEL_FULL_VERSION}
```

其他资源

- 驱动程序工具包

4.10. 使用内核模块管理 (KMM) 的签名

在启用了安全引导的系统上，所有内核模块 (kmod) 必须使用注册到 Machine Owner 的密钥 (MOK) 数据库的公钥/私钥对进行签名。作为发布的一部分分发的驱动程序应该已经由发行版的私钥签名，但对于内核模块构建树外，KMM 支持使用内核映射的 **sign** 部分对内核模块进行签名。

有关使用安全引导的详情，请参阅 生成公钥和私钥对

先决条件

- 正确 (DER) 格式的公钥对。
- 至少一个启用了安全引导的节点，在 MOK 数据库中注册了公钥。
- 预构建的驱动程序容器镜像，或构建一个集群所需的源代码和 **Dockerfile**。

4.11. 为 SECUREBOOT 添加密钥

要使用 KMM 内核模块管理 (KMM) 为内核模块签名，需要一个证书和私钥。有关如何创建这些密钥对的详情，请参阅生成公钥和私钥对。

有关如何提取公钥和私钥对的详情，请参阅使用私钥签名内核模块。使用第 1 到 4 步将密钥提取到文件中。

流程

1. 创建包含证书以及包含私钥的 **sb_cert.priv** 文件的 **sb_cert.cer** 文件：

```
$ openssl req -x509 -new -nodes -utf8 -sha256 -days 36500 -batch -config
configuration_file.config -outform DER -out my_signing_key_pub.der -keyout
my_signing_key.priv
```

2. 使用以下方法之一添加文件：
   - 将文件直接添加为 secret ：

```
$ oc create secret generic my-signing-key --from-file=key=<my_signing_key.priv>
```

```
$ oc create secret generic my-signing-key-pub --from-file=cert=
<my_signing_key_pub.der>
```

- 根据 base64 编码添加文件：

```
$ cat sb_cert.priv | base64 -w 0 > my_signing_key2.base64
```

```
$ cat sb_cert.cer | base64 -w 0 > my_signing_key_pub.base64
```

3. 在 YAML 文件中添加编码的文本：

```
apiVersion: v1
kind: Secret
metadata:
  name: my-signing-key-pub
  namespace: default 1
type: Opaque
data:
  cert: <base64_encoded_secureboot_public_key>

---
apiVersion: v1
kind: Secret
metadata:
  name: my-signing-key
  namespace: default 2
type: Opaque
data:
  key: <base64_encoded_secureboot_private_key>
```

1 2 **namespace** - 使用有效的命名空间替换 **default**。

4. 应用 YAML 文件：

```
$ oc apply -f <yaml_filename>
```

4.11.1. 检查密钥

添加密钥后，您必须检查它们以确保正确设置它们。

流程

1. 检查以确保正确设置公钥 secret：

```
$ oc get secret -o yaml <certificate secret name> | awk '/cert/{print $2; exit}' | base64 -d  |
openssl x509 -inform der -text
```

这应该会显示带有 Serial Number, Issuer, Subject 等的证书。

2. 检查以确保正确设置私钥 secret：

```
$ oc get secret -o yaml <private key secret name> | awk '/key/{print $2; exit}' | base64 -d
```

这应该显示包括在 **-----BEGIN PRIVATE KEY-----** 和 **-----END PRIVATE KEY-----** 行中的密钥。

4.12. 在预构建的镜像中签名 KMODS

如果您有预构建的镜像，如由硬件供应商分发的镜像，或者在其他位置构建。

以下 YAML 文件将公钥/私钥对添加为带有所需密钥名称的 secret - **key** 为私钥，**cert** 为公钥。然后，集群会拉取 **unsignedImage** 镜像，打开它，签署 **filesToSign** 中列出的内核模块，将它们添加回来，并将生成的镜像推送到 **containerImage**。

然后，KMM 将签名的 kmods 加载到与选择器匹配的所有节点上。kmods 在其 MOK 数据库中有公钥的任何节点上成功载入，以及所有未启用 secure-boot 的节点，这将忽略签名。

先决条件

- **keySecret** 和 **certSecret** secret 已在与其他资源相同的命名空间中创建。

流程

- 应用 YAML 文件：

```
---
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: example-module
spec:
  moduleLoader:
    serviceAccountName: default
    container:
      modprobe: 1
        moduleName: '<module_name>'
      kernelMappings:
        # the kmods will be deployed on all nodes in the cluster with a kernel that matches the
regexp
        - regexp: '^.*\.x86_64$'
          # the container to produce containing the signed kmods
          containerImage: <image_name> 2
          sign:
            # the image containing the unsigned kmods (we need this because we are not
building the kmods within the cluster)
            unsignedImage: <image_name> 3
            keySecret: # a secret holding the private secureboot key with the key 'key'
              name: <private_key_secret_name>
            certSecret: # a secret holding the public secureboot key with the key 'cert'
              name: <certificate_secret_name>
            filesToSign: # full path within the unsignedImage container to the kmod(s) to sign
              - /opt/lib/modules/4.18.0-348.2.1.el8_5.x86_64/kmm_ci_a.ko
  imageRepoSecret:
    # the name of a secret containing credentials to pull unsignedImage and push
```

```
containerImage to the registry
    name: repo-pull-secret
  selector:
    kubernetes.io/arch: amd64
```

1. 要载入的 kmod 的名称。
2. 容器镜像的名称。例如：**quay.io/myuser/my-driver:<kernelversion>**。
3. 未签名镜像的名称。例如：**quay.io/myuser/my-driver:<kernelversion>**。

4.13. 构建并签名 KMOD 镜像

如果您有源代码且必须首先构建镜像，请使用这个流程。

以下 YAML 文件使用存储库中的源代码构建新容器镜像。生成的镜像将保存到带有临时名称的 registry 中，然后使用 **sign** 部分中的参数来签名此临时镜像。

临时镜像名称基于最终镜像名称，设置为 **<containerImage>:<tag>-<namespace>_<module name>_kmm_unsigned**。

例如，使用以下 YAML 文件，内核模块管理(KMM) 构建名为 **example.org/repository/minimal-driver:final-default_example-module_kmm_unsigned** 的镜像，其中包含带有未签名的 kmods 的构建并将其推送到 registry。然后，它创建一个名为 **example.org/repository/minimal-driver:final** 的第二个镜像，其中包含签名的 kmod。它是 worker pod 拉取的第二个镜像，其中包含要在集群节点上载入的 kmods。

签名后，您可以从 registry 中安全地删除临时镜像。如果需要，它将被重建。

先决条件

- **keySecret** 和 **certSecret** secret 已在与其他资源相同的命名空间中创建。

流程

- 应用 YAML 文件：

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-module-dockerfile
  namespace: <namespace> 1
data:
  Dockerfile: |
    ARG DTK_AUTO
    ARG KERNEL_VERSION
    FROM ${DTK_AUTO} as builder
    WORKDIR /build/
    RUN git clone -b main --single-branch https://github.com/rh-ecosystem-edge/kernel-module-management.git
    WORKDIR kernel-module-management/ci/kmm-kmod/
    RUN make
    FROM registry.access.redhat.com/ubi9/ubi:latest
```

```
    ARG KERNEL_VERSION
    RUN yum -y install kmod && yum clean all
    RUN mkdir -p /opt/lib/modules/${KERNEL_VERSION}
    COPY --from=builder /build/kernel-module-management/ci/kmm-kmod/*.ko
/opt/lib/modules/${KERNEL_VERSION}/
    RUN /usr/sbin/depmod -b /opt
---
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: example-module
  namespace: <namespace> 2
spec:
  moduleLoader:
    serviceAccountName: default 3
    container:
      modprobe:
        moduleName: simple_kmod
      kernelMappings:
        - regexp: '^.*\.x86_64$'
          containerImage: <final_driver_container_name>
          build:
            dockerfileConfigMap:
              name: example-module-dockerfile
          sign:
            keySecret:
              name: <private_key_secret_name>
            certSecret:
              name: <certificate_secret_name>
            filesToSign:
              - /opt/lib/modules/4.18.0-348.2.1.el8_5.x86_64/kmm_ci_a.ko
  imageRepoSecret: 4
    name: repo-pull-secret
  selector: # top-level selector
    kubernetes.io/arch: amd64
```

1 2 使用有效命名空间替换 **default**。

3 默认 **serviceAccountName** 没有运行特权模块所需的权限。有关创建服务帐户的详情，请参考本节的"添加资源"中的"创建服务帐户"。

4 用作 **DaemonSet** 对象中的 **imagePullSecrets**，用于拉取和推送构建和签名功能。

其他资源

- 创建服务帐户.

4.14. KMM HUB 和 SPOKE

在 hub 和 spoke 场景中，许多 spoke 集群连接到一个中央强大的 hub 集群。内核模块管理(KMM)依赖于 Red Hat Advanced Cluster Management (RHACM)在 hub 和 spoke 环境中运行。

KMM 通过分离 KMM 功能与 hub 和 spoke 环境兼容。提供了 **ManagedClusterModule** 自定义资源定义 (CRD) 来打包现有的 **Module** CRD，并将其扩展为选择 Spoke 集群。另外，提供了 KMM-Hub，它是一个新的独立控制器，用于在 hub 集群上构建镜像并签署模块。

在 hub 和 spoke 设置中，spokes 专注于由 hub 集群管理的资源受限集群。spoke 运行 KMM 的单集群版本，并禁用这些资源密集型功能。要将 KMM 适应此环境，您应该将 spoke 上运行的工作负载降低为最小值，而 hub 会处理昂贵的任务。

构建内核模块镜像并签名 **.ko** 文件，应在 hub 上运行。Module Loader 和 Device Plugin **DaemonSet** 的调度只能在 spoke 上进行。

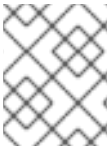
其他资源

- Red Hat Advanced Cluster Management (RHACM)

4.14.1. KMM-Hub

KMM 项目提供 KMM-Hub，它是一个专用于 hub 集群的 KMM 版本。KMM-Hub 监控 spoke 上运行的所有内核版本，并决定集群中应该接收内核模块的节点。

KMM-Hub 运行所有计算密集型任务，如镜像构建和 kmod 签名，并准备裁剪通过 RHACM 传送到 spoke 的 **Module**。



注意

KMM-Hub 无法用于在 hub 集群中加载内核模块。安装常规版本的 KMM 以加载内核模块。

其他资源

- 安装 KMM

4.14.2. 安装 KMM-Hub

您可以使用以下方法之一安装 KMM-Hub：

- 使用 Operator Lifecycle Manager (OLM)
- 创建 KMM 资源

其他资源

- KMM Operator 捆绑包

4.14.2.1. 使用 Operator Lifecycle Manager 安装 KMM-Hub

使用 OpenShift 控制台的 **Operators** 部分安装 KMM-Hub。

4.14.2.2. 通过创建 KMM 资源来安装 KMM-Hub

流程

- 如果要以编程方式安装 KMM-Hub，您可以使用以下资源创建 **Namespace**、**OperatorGroup** 和 **Subscription** 资源：

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-kmm-hub
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: kernel-module-management-hub
  namespace: openshift-kmm-hub
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: kernel-module-management-hub
  namespace: openshift-kmm-hub
spec:
  channel: stable
  installPlanApproval: Automatic
  name: kernel-module-management-hub
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

4.14.3. 使用 ManagedClusterModule CRD

使用 **ManagedClusterModule** 自定义资源定义(CRD)在 spoke 集群上配置内核模块的部署。此 CRD 是集群范围的，嵌套了一个 **Module** spec，并添加以下附加字段：

```
apiVersion: hub.kmm.sigs.x-k8s.io/v1beta1
kind: ManagedClusterModule
metadata:
  name: <my-mcm>
  # No namespace, because this resource is cluster-scoped.
spec:
  moduleSpec: (1)
    selector: (2)
      node-wants-my-mcm: 'true'

  spokeNamespace: <some-namespace> (3)

  selector: (4)
    wants-my-mcm: 'true'
```

1. **moduleSpec**: 包含 **moduleLoader** 和 **devicePlugin** 部分，类似于 **Module** 资源。
2. 选择 **ManagedCluster** 中的节点。
3. 指定应在其中创建 **Module** 的命名空间。
4. 选择 **ManagedCluster** 对象。

如果 **.spec.moduleSpec** 中存在构建或签名指令，则这些 pod 在 Operator 命名空间中的 hub 集群上运行。

当 **.spec.selector** 匹配一个或多个 **ManagedCluster** 资源时，KMM-Hub 在对应的命名空间中创建 **ManifestWork** 资源。**manifestwork** 包含 trimmed-down **Module** 资源，保留了内核映射，但所有 **build** 和 **sign** 子部分都会被删除。包含以标签结尾的镜像名称的 **containerImage** 字段将替换为等效摘要。

4.14.4. 在 spoke 上运行 KMM

在 spoke 上安装内核模块管理 (KMM) 后，不需要进一步操作。从 hub 创建一个 **ManagedClusterModule** 对象，以便在 spoke 集群上部署内核模块。

流程

您可以通过 RHACM **Policy** 对象在 spokes 集群上安装 KMM。除了从 Operator hub 安装 KMM 并以轻量级 spoke 模式运行它外，**Policy** 还会配置 RHACM 代理所需的额外 RBAC 来管理 **Module** 资源。

- 使用以下 RHACM 策略在 spoke 集群上安装 KMM：

```
---
apiVersion: policy.open-cluster-management.io/v1
kind: Policy
metadata:
  name: install-kmm
spec:
  remediationAction: enforce
  disabled: false
  policy-templates:
    - objectDefinition:
        apiVersion: policy.open-cluster-management.io/v1
        kind: ConfigurationPolicy
        metadata:
          name: install-kmm
        spec:
          severity: high
          object-templates:
          - complianceType: mustonlyhave
            objectDefinition:
              apiVersion: v1
              kind: Namespace
              metadata:
                name: openshift-kmm
          - complianceType: mustonlyhave
            objectDefinition:
              apiVersion: operators.coreos.com/v1
              kind: OperatorGroup
              metadata:
                name: kmm
                namespace: openshift-kmm
              spec:
                upgradeStrategy: Default
          - complianceType: mustonlyhave
            objectDefinition:
              apiVersion: operators.coreos.com/v1alpha1
              kind: Subscription
              metadata:
```

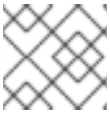
```
            name: kernel-module-management
            namespace: openshift-kmm
          spec:
            channel: stable
            config:
              env:
                - name: KMM_MANAGED 1
                  value: "1"
            installPlanApproval: Automatic
            name: kernel-module-management
            source: redhat-operators
            sourceNamespace: openshift-marketplace
        - complianceType: mustonlyhave
          objectDefinition:
            apiVersion: rbac.authorization.k8s.io/v1
            kind: ClusterRole
            metadata:
              name: kmm-module-manager
            rules:
              - apiGroups: [kmm.sigs.x-k8s.io]
                resources: [modules]
                verbs: [create, delete, get, list, patch, update, watch]
        - complianceType: mustonlyhave
          objectDefinition:
            apiVersion: rbac.authorization.k8s.io/v1
            kind: ClusterRoleBinding
            metadata:
              name: klusterlet-kmm
            subjects:
            - kind: ServiceAccount
              name: klusterlet-work-sa
              namespace: open-cluster-management-agent
            roleRef:
              kind: ClusterRole
              name: kmm-module-manager
              apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: all-managed-clusters
spec:
  clusterSelector: 2
    matchExpressions: []
---
apiVersion: policy.open-cluster-management.io/v1
kind: PlacementBinding
metadata:
  name: install-kmm
placementRef:
  apiGroup: apps.open-cluster-management.io
  kind: PlacementRule
  name: all-managed-clusters
subjects:
```

```
- apiGroup: policy.open-cluster-management.io
  kind: Policy
  name: install-kmm
```

1. 当在 spoke 集群上运行 KMM 时，需要这个环境变量。
2. **spec.clusterSelector** 字段可以自定义为仅目标选择的集群。

4.15. 内核模块的自定义升级

如果需要，使用此流程在节点上运行维护操作时升级内核模块，包括重新引导节点。要最小化对集群中运行的工作负载的影响，请按顺序运行内核升级过程，一次一个节点。



注意

此流程需要了解使用内核模块的工作负载，且必须由集群管理员管理。

先决条件

- 在升级前，在内核模块使用的所有节点上设置 **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>=$moduleVersion** 标签。
- 终止节点上的所有用户应用程序工作负载，或将其移至另一节点。
- 卸载当前载入的内核模块。
- 在内核模块卸载前，确保用户工作负载（在访问内核模块的集群中运行的应用程序）在载入新内核模块版本后不会在节点上运行，且工作负载在载入了新的内核模块版本后在该节点上运行。

流程

1. 确保节点上由 KMM 管理的设备插件已被卸载。
2. 更新 **Module** 自定义资源(CR)中的以下字段：
   - **containerImage** （到适当的内核版本）
   - **version**
     更新应当具有原子性，即 **containerImage** 和 **version** 字段必须同时更新。
3. 使用升级的节点上的内核模块终止任何工作负载。
4. 删除节点上的 **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>** 标签。运行以下命令从节点卸载内核模块：

```
$ oc label node/<node_name> kmm.node.kubernetes.io/version-module.
<module_namespace>.<module_name>-
```

5. 如果需要，作为集群管理员，在内核模块升级的节点上执行任何额外的维护。
   如果没有额外的升级，您可以通过将 **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>** 标签值更新为新的 **$moduleVersion**（如 **Module** 中的设置），并跳过第 3 到 6 步。

6. 运行以下命令，将 **kmm.node.kubernetes.io/version-module.<module_namespace>.<module_name>=$moduleVersion** 标签添加到节点。**$moduleVersion** 必须与 **Module** CR 中的 **version** 字段的新值相等。

```
$ oc label node/<node_name> kmm.node.kubernetes.io/version-module.
<module_namespace>.<module_name>=<desired_version>
```

注意

由于标签名称中的 Kubernetes 限制，**Module** 名称和命名空间的组合长度不能超过 39 个字符。

7. 恢复该节点上利用内核模块的任何工作负载。

8. 重新加载节点上由 KMM 管理的设备插件。

4.16. 第 1 天载入内核模块

内核模块管理(KMM)通常是一个第 2 天 Operator。只有在 Linux (RHCOS) 服务器的完整初始化后才会加载内核模块。但是，在某些情况下，必须在早期阶段载入内核模块。第 1 天功能允许您在 Linux **systemd** 初始化阶段使用 Machine Config Operator (MCO)加载内核模块。

其他资源

- Machine Config Operator

4.16.1. 第 1 天支持的用例

第 1 天功能支持有限数量的用例。主要用例是在 NetworkManager 服务初始化前允许加载树外(OOT)内核模块。它不支持在 **initramfs** 阶段载入内核模块。

以下是第 1 天功能所需的条件：

- 内核模块没有在内核中载入。
- in-tree 内核模块加载到内核中，但可以被卸载，并由 OOT 内核模块替代。这意味着 in-tree 模块没有被任何其他内核模块引用。
- 为了使第 1 天功能正常工作，节点必须具有功能网络接口，即该接口的一个树内内核驱动程序。OOT 内核模块可以是网络驱动程序，它将替换功能网络驱动程序。

4.16.2. OOT 内核模块加载流

加载 out-of-tree (OOT) 内核模块会利用 Machine Config Operator (MCO)。流程序列如下：

流程

1. 将 **MachineConfig** 资源应用到现有的正在运行的集群。要识别需要更新的必要节点，您必须创建一个适当的 **MachineConfigPool** 资源。
2. MCO 通过节点应用重启节点。在任何重启的节点上，部署了两个新的 **systemd** 服务： **pull** 服务和 **load** 服务。

3. **load** 服务被配置为在 **NetworkConfiguration** 服务之前运行。该服务会尝试拉取预定义的内核模块镜像，然后使用该镜像卸载树内模块并加载 OOT 内核模块。

4. **pull** 服务配置为在 NetworkManager 服务后运行。该服务检查预配置的内核模块镜像是否位于节点的文件系统中。如果是，该服务正常存在，服务器将继续引导过程。如果没有，它会将镜像拉取到节点上，并在之后重启该节点。

4.16.3. 内核模块镜像

第 1 天功能使用与第 2 天 KMM 构建相同的 DTK 的镜像。out-of-tree 内核模块应位于 **/opt/lib/modules/${kernelVersion}** 下。

其他资源

- 驱动程序工具包

4.16.4. in-tree 模块替换

第 1 天功能始终尝试将树内内核模块替换为 OOT 版本。如果没有加载 in-tree 内核模块，则流不会受到影响；服务继续进行并加载 OOT 内核模块。

4.16.5. MCO yaml 创建

KMM 提供了一个 API，用于为第 1 天功能创建 MCO YAML 清单：

```
ProduceMachineConfig(machineConfigName, machineConfigPoolRef, kernelModuleImage,
kernelModuleName string) (string, error)
```

返回的输出是要应用的 MCO YAML 清单的字符串表示。客户最多可应用此 YAML。

参数是：

machineConfigName
: MCO YAML 清单的名称。此参数设置为 MCO YAML 清单元数据的 **name** 参数。

machineConfigPoolRef
: 用于识别目标节点的 **MachineConfigPool** 名称。

kernelModuleImage
: 包含 OOT 内核模块的容器镜像名称。

kernelModuleName
: OOT 内核模块的名称。这个参数用于卸载 in-tree 内核模块（如果加载到内核），并加载 OOT 内核模块。

API 位于 KMM 源代码的 **pkg/mcproducer** 软件包下。KMM operator 不需要运行以使用第 1 天功能。您只需要将 **pkg/mcproducer** 软件包导入到其 operator/utility 代码中，调用 API，并将生成的 MCO YAML 应用到集群。

4.16.6. MachineConfigPool

MachineConfigPool 标识受应用的 MCO 影响的节点集合。

```
kind: MachineConfigPool
metadata:
```

```
  name: sfc
spec:
  machineConfigSelector: 1
    matchExpressions:
      - {key: machineconfiguration.openshift.io/role, operator: In, values: [worker, sfc]}
  nodeSelector: 2
    matchLabels:
      node-role.kubernetes.io/sfc: ""
  paused: false
  maxUnavailable: 1
```

1 与 MachineConfig 中的标签匹配。

2 匹配节点上的标签。

OCP 集群中预定义的 **MachineConfigPools** ：

- **Worker** ：将集群中的所有 worker 节点目标
- **Master** ：将集群中的所有 master 节点目标

定义以下 **MachineConfig** 以 master **MachineConfigPool** 为目标：

```
metadata:
  labels:
    machineconfiguration.opensfhit.io/role: master
```

定义以下 **MachineConfig** 以 worker **MachineConfigPool** 为目标：

```
metadata:
  labels:
    machineconfiguration.opensfhit.io/role: worker
```

其他资源

- 关于 MachineConfigPool

4.17. 调试和故障排除

如果您的驱动程序容器中的 kmods 没有签名或使用错误的密钥签名，则容器可能会进入 **PostStartHookError** 或 **CrashLoopBackOff** 状态。您可以在容器上运行 **oc describe** 命令验证，该命令在此场景中显示以下信息：

```
modprobe: ERROR: could not insert '<your_kmod_name>': Required key not available
```

4.18. KMM 固件支持

内核模块有时需要从文件系统中加载固件文件。KMM 支持将固件文件从 kmod 镜像复制到节点的文件系统。

在运行 **modprobe** 命令前，节点上的 **.spec.moduleLoader.container.modprobe.firmwarePath** 的内容会被复制到节点上的 **/var/lib/firmware** 路径中。

在运行 **modprobe -r** 命令之前，所有文件和空目录都会从该位置中删除，以便在 pod 终止时卸载内核模块。

4.18.1. 在节点上配置查找路径

在 OpenShift Container Platform 节点上，固件的默认查找路径集合不包括 **/var/lib/firmware** 路径。

流程

1. 使用 Machine Config Operator 创建包含 **/var/lib/firmware** 路径的 **MachineConfig** 自定义资源 (CR)：

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker (1)
  name: 99-worker-kernel-args-firmware-path
spec:
  kernelArguments:
    - 'firmware_class.path=/var/lib/firmware'
```

(1) 您可以根据您的需要配置标签。对于单节点 OpenShift，请使用 **control-pane** 或 **master** 对象。

2. 通过应用 **MachineConfig** CR，节点会自动重启。

其他资源

- Machine Config Operator。

4.18.2. 构建 kmod 镜像

流程

- 除了构建内核模块本身外，在构建器镜像中包含二进制固件：

```
FROM registry.redhat.io/ubi9/ubi-minimal as builder

# Build the kmod

RUN ["mkdir", "/firmware"]
RUN ["curl", "-o", "/firmware/firmware.bin", "https://artifacts.example.com/firmware.bin"]

FROM registry.redhat.io/ubi9/ubi-minimal

# Copy the kmod, install modprobe, run depmod

COPY --from=builder /firmware /firmware
```

4.18.3. 调整模块资源

流程

流程

- 在 **Module** 自定义资源 (CR) 中设置 **.spec.moduleLoader.container.modprobe.firmwarePath**：

```
apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: my-kmod
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: my-kmod  # Required

        firmwarePath: /firmware 1
```

1 可选：在节点上将 **/firmware/*** 复制到 **/var/lib/firmware/**。

4.19. 第 0 天到第 2 天 KMOD 安装

您可以在第 0 天到第 2 天的操作中，在没有内核模块管理 (KMM) 的情况下安装一些内核模块 (kmods)。这有助于将 kmods 转换为 KMM。

使用以下条件来确定合适的 kmod 安装。

第 0 天

在集群中节点变为 **Ready** 所需的最基本 kmod。这些类型的 kmod 示例包括：

- 作为引导过程的一部分挂载 rootFS 所需的存储驱动程序
- 机器访问 bootstrap 节点上的 **machine-config-server** 所需的网络驱动程序来拉取 ignition 并加入集群

第 1 天

集群中的节点变为 **Ready** 不需要的 kmods，但当节点为 **Ready** 时无法卸载。
此类 kmod 的一个示例是，树外 (OOT) 网络驱动程序，它取代了一个过时的树内驱动程序来充分利用 NIC，**NetworkManager** 依赖于它。当节点为 **Ready** 时，因为是 **NetworkManager** 的依赖项而无法卸载驱动程序。

第 2 天

kmods 可以动态加载到内核或从其中删除，而无需干扰集群基础架构，例如连接性。
这些类型的 kmod 示例包括：

- GPU operator
- 二级网络适配器
- 可现场编程门阵列 (FPGA)

4.19.1. 分层背景

当集群中安装第 0 天 kmod 时，通过 Machine Config Operator (MCO) 和 OpenShift Container Platform 升级应用层不会触发节点升级。

只有在向它添加新的功能时才需要重新编译驱动程序，因为节点的操作系统将保持不变。

4.19.2. 生命周期管理

当驱动程序允许时，您可以在无需重启的情况下，使用 KMM 管理 kmod 的第 0 天到第 2 天的生命周期。

注意

但如果升级需要重新引导节点（例如，当需要重建 **initramfs** 文件时）这将无法实现。

使用以下选项之一进行生命周期管理。

4.19.2.1. 将 kmod 视为树内驱动程序

在您要升级 kmods 时使用此方法。在这种情况下，将 kmod 视为树内驱动程序，并在集群中创建一个带有 **inTreeRemoval** 字段的 **Module** 用来卸载驱动程序的旧版本。

请注意以下将 kmod 视为树状驱动程序的特征：

- 当 KMM 尝试在所有选择的节点上同时卸载和加载 kmod 时可能会出现停机的情况。
- 如果删除驱动会导致丢失到节点的连接，则这个方法可以正常工作。因为 KMM 使用单个 pod 来卸载和加载驱动程序。

4.19.2.2. 使用有特定顺序的升级

您可以使用有特定顺序的升级(ordered_upgrade.md) 在集群中创建一个版本化的 **Module** 来代表没有效果的 kmods，因为 kmods 已被加载。

请注意使用有特定顺序的升级特性：

- 没有集群停机时间。因为您可以控制升级的过程，以及同时升级的节点数量，所以可以实现在没有停机时间的情况下完成升级。
- 如果卸载驱动程序会导致丢失与节点的连接，则这个方法将无法正常工作，因为 KMM 创建两个不同的 worker pod，一个用于卸载，另一个用于加载。这些 pod 不会被调度。

4.20. KMM 故障排除

当对 KMM 安装问题进行故障排除时，您可以监控日志以确定在哪个阶段出现问题。然后，检索与该阶段相关的诊断数据。

4.20.1. 读取 Operator 日志

您可以使用 **oc logs** 命令来读取 Operator 日志，如下例所示。

KMM 控制器的命令示例

```
$ oc logs -fn openshift-kmm deployments/kmm-operator-controller
```

KMM Webhook 服务器的命令示例

```
$ oc logs -fn openshift-kmm deployments/kmm-operator-webhook-server
```

KMM-Hub 控制器的命令示例

```
$ oc logs -fn openshift-kmm-hub deployments/kmm-operator-hub-controller
```

KMM-Hub webhook 服务器的命令示例

```
$ oc logs -fn openshift-kmm deployments/kmm-operator-hub-webhook-server
```

4.20.2. 观察事件

使用以下方法查看 KMM 事件。

build 和 sign
KMM 在启动 kmod 镜像构建或观察其结果时都会发布事件。这些事件附加到 **Module** 对象，并在 **oc describe module** 命令的输出的末尾可用，如下例所示：

```
$ oc describe modules.kmm.sigs.x-k8s.io kmm-ci-a
[...]
Events:
  Type    Reason         Age              From  Message
  ----    ------         ----             ----  -------
  Normal  BuildCreated    2m29s            kmm   Build created for kernel 6.6.2-201.fc39.x86_64
  Normal  BuildSucceeded  63s              kmm   Build job succeeded for kernel 6.6.2-201.fc39.x86_64
  Normal  SignCreated     64s (x2 over 64s)  kmm   Sign created for kernel 6.6.2-201.fc39.x86_64
  Normal  SignSucceeded   57s              kmm   Sign job succeeded for kernel 6.6.2-201.fc39.x86_64
```

模块载入或卸载
KMM 会在节点上成功加载或卸载内核模块时发布事件。这些事件附加到 **Node** 对象，并在 **oc describe node** 命令的输出的末尾可用，如下例所示：

```
$ oc describe node my-node
[...]
Events:
  Type    Reason         Age    From  Message
  ----    ------         ----   ----  -------
[...]
  Normal  ModuleLoaded    4m17s  kmm   Module default/kmm-ci-a loaded into the kernel
  Normal  ModuleUnloaded  2s     kmm   Module default/kmm-ci-a unloaded from the kernel
```

4.20.3. 使用 must-gather 工具

oc adm must-gather 命令是收集支持捆绑包并为红帽支持提供调试信息的首选方法。可以使用适当的参数运行命令来收集具体信息，如以下部分所述：

其他资源

- 关于 must-gather 工具

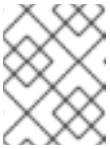
4.20.3.1. 为 KMM 收集数据

流程

1. 为 KMM Operator 控制器管理器收集数据：

   a. 设置 **MUST_GATHER_IMAGE** 变量：

   ```
   $ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm kmm-operator-controller -ojsonpath='{.spec.template.spec.containers[?(@.name=="manager")].env[?(@.name=="RELATED_IMAGE_MUST_GATHER")].value}')
   $ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather
   ```

   注意

   如果在自定义命名空间中安装 KMM，请使用 **-n <namespace>** 开关指定命名空间。

   b. 运行 **must-gather** 工具：

   ```
   $ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather
   ```

2. 查看 Operator 日志：

   ```
   $ oc logs -fn openshift-kmm deployments/kmm-operator-controller
   ```

   例 4.1. 输出示例

   ```
   I0228 09:36:37.352405       1 request.go:682] Waited for 1.001998746s due to client-side throttling, not priority and fairness, request: GET:https://172.30.0.1:443/apis/machine.openshift.io/v1beta1?timeout=32s
   I0228 09:36:40.767060       1 listener.go:44] kmm/controller-runtime/metrics "msg"="Metrics server is starting to listen" "addr"="127.0.0.1:8080"
   I0228 09:36:40.769483       1 main.go:234] kmm/setup "msg"="starting manager"
   I0228 09:36:40.769907       1 internal.go:366] kmm "msg"="Starting server" "addr"={"IP":"127.0.0.1","Port":8080,"Zone":""} "kind"="metrics" "path"="/metrics"
   I0228 09:36:40.770025       1 internal.go:366] kmm "msg"="Starting server" "addr"={"IP":"::","Port":8081,"Zone":""} "kind"="health probe"
   I0228 09:36:40.770128       1 leaderelection.go:248] attempting to acquire leader lease openshift-kmm/kmm.sigs.x-k8s.io...
   I0228 09:36:40.784396       1 leaderelection.go:258] successfully acquired lease openshift-kmm/kmm.sigs.x-k8s.io
   I0228 09:36:40.784876       1 controller.go:185] kmm "msg"="Starting EventSource" "controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module" "source"="kind source: *v1beta1.Module"
   I0228 09:36:40.784925       1 controller.go:185] kmm "msg"="Starting EventSource" "controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module" "source"="kind source: *v1.DaemonSet"
   I0228 09:36:40.784968       1 controller.go:185] kmm "msg"="Starting EventSource" "controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module" "source"="kind source: *v1.Build"
   I0228 09:36:40.785001       1 controller.go:185] kmm "msg"="Starting EventSource" "controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
   ```

```
"source"="kind source: *v1.Job"
I0228 09:36:40.785025       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
"source"="kind source: *v1.Node"
I0228 09:36:40.785039       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="Module" "controllerGroup"="kmm.sigs.x-k8s.io" "controllerKind"="Module"
I0228 09:36:40.785458       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PodNodeModule" "controllerGroup"="" "controllerKind"="Pod" "source"="kind
source: *v1.Pod"
I0228 09:36:40.786947       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1beta1.PreflightValidation"
I0228 09:36:40.787406       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1.Build"
I0228 09:36:40.787474       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1.Job"
I0228 09:36:40.787488       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation" "source"="kind source: *v1beta1.Module"
I0228 09:36:40.787603       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="NodeKernel" "controllerGroup"="" "controllerKind"="Node" "source"="kind
source: *v1.Node"
I0228 09:36:40.787634       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="NodeKernel" "controllerGroup"="" "controllerKind"="Node"
I0228 09:36:40.787680       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="PreflightValidation" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidation"
I0228 09:36:40.785607       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream" "source"="kind source: *v1.ImageStream"
I0228 09:36:40.787822       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP" "source"="kind source:
*v1beta1.PreflightValidationOCP"
I0228 09:36:40.787853       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream"
I0228 09:36:40.787879       1 controller.go:185] kmm "msg"="Starting EventSource"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP" "source"="kind source:
*v1beta1.PreflightValidation"
I0228 09:36:40.787905       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="preflightvalidationocp" "controllerGroup"="kmm.sigs.x-k8s.io"
"controllerKind"="PreflightValidationOCP"
I0228 09:36:40.786489       1 controller.go:193] kmm "msg"="Starting Controller"
"controller"="PodNodeModule" "controllerGroup"="" "controllerKind"="Pod"
```

4.20.3.2. 为 KMM-Hub 收集数据

流程

1. 为 KMM Operator hub 控制器管理器收集数据：

a. 设置 **MUST_GATHER_IMAGE** 变量：

```
$ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm-hub kmm-
operator-hub-controller -ojsonpath='{.spec.template.spec.containers[?
(@.name=="manager")].env[?
(@.name=="RELATED_IMAGE_MUST_GATHER")].value}')
$ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather -u
```

注意

如果在自定义命名空间中安装 KMM，请使用 **-n <namespace>** 开关指定命名空间。

b. 运行 **must-gather** 工具：

```
$ oc adm must-gather --image="${MUST_GATHER_IMAGE}" -- /usr/bin/gather -u
```

2. 查看 Operator 日志：

```
$ oc logs -fn openshift-kmm-hub deployments/kmm-operator-hub-controller
```

例 4.2. 输出示例

```
I0417 11:34:08.807472       1 request.go:682] Waited for 1.023403273s due to client-side
throttling, not priority and fairness, request:
GET:https://172.30.0.1:443/apis/tuned.openshift.io/v1?timeout=32s
I0417 11:34:12.373413       1 listener.go:44] kmm-hub/controller-runtime/metrics
"msg"="Metrics server is starting to listen" "addr"="127.0.0.1:8080"
I0417 11:34:12.376253       1 main.go:150] kmm-hub/setup "msg"="Adding controller"
"name"="ManagedClusterModule"
I0417 11:34:12.376621       1 main.go:186] kmm-hub/setup "msg"="starting manager"
I0417 11:34:12.377690       1 leaderelection.go:248] attempting to acquire leader lease
openshift-kmm-hub/kmm-hub.sigs.x-k8s.io...
I0417 11:34:12.378078       1 internal.go:366] kmm-hub "msg"="Starting server" "addr"=
{"IP":"127.0.0.1","Port":8080,"Zone":""} "kind"="metrics" "path"="/metrics"
I0417 11:34:12.378222       1 internal.go:366] kmm-hub "msg"="Starting server" "addr"=
{"IP":"::","Port":8081,"Zone":""} "kind"="health probe"
I0417 11:34:12.395703       1 leaderelection.go:258] successfully acquired lease
openshift-kmm-hub/kmm-hub.sigs.x-k8s.io
I0417 11:34:12.396334       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "source"="kind source:
*v1beta1.ManagedClusterModule"
I0417 11:34:12.396403       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "source"="kind source: *v1.ManifestWork"
I0417 11:34:12.396430       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "source"="kind source: *v1.Build"
I0417 11:34:12.396469       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "source"="kind source: *v1.Job"
I0417 11:34:12.396522       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
```

```
"controllerKind"="ManagedClusterModule" "source"="kind source: *v1.ManagedCluster"
I0417 11:34:12.396543       1 controller.go:193] kmm-hub "msg"="Starting Controller"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule"
I0417 11:34:12.397175       1 controller.go:185] kmm-hub "msg"="Starting EventSource"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream" "source"="kind source: *v1.ImageStream"
I0417 11:34:12.397221       1 controller.go:193] kmm-hub "msg"="Starting Controller"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream"
I0417 11:34:12.498335       1 filter.go:196] kmm-hub "msg"="Listing all
ManagedClusterModules" "managedcluster"="local-cluster"
I0417 11:34:12.498570       1 filter.go:205] kmm-hub "msg"="Listed
ManagedClusterModules" "count"=0 "managedcluster"="local-cluster"
I0417 11:34:12.498629       1 filter.go:238] kmm-hub "msg"="Adding reconciliation
requests" "count"=0 "managedcluster"="local-cluster"
I0417 11:34:12.498687       1 filter.go:196] kmm-hub "msg"="Listing all
ManagedClusterModules" "managedcluster"="sno1-0"
I0417 11:34:12.498750       1 filter.go:205] kmm-hub "msg"="Listed
ManagedClusterModules" "count"=0 "managedcluster"="sno1-0"
I0417 11:34:12.498801       1 filter.go:238] kmm-hub "msg"="Adding reconciliation
requests" "count"=0 "managedcluster"="sno1-0"
I0417 11:34:12.501947       1 controller.go:227] kmm-hub "msg"="Starting workers"
"controller"="imagestream" "controllerGroup"="image.openshift.io"
"controllerKind"="ImageStream" "worker count"=1
I0417 11:34:12.501948       1 controller.go:227] kmm-hub "msg"="Starting workers"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule" "worker count"=1
I0417 11:34:12.502285       1 imagestream_reconciler.go:50] kmm-hub "msg"="registered
imagestream info mapping" "ImageStream"={"name":"driver-
toolkit","namespace":"openshift"} "controller"="imagestream"
"controllerGroup"="image.openshift.io" "controllerKind"="ImageStream"
"dtkImage"="quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:df42b4785a7a662b30da53bdb0d206120cf4d24b45674227b16051ba4b7c393
4" "name"="driver-toolkit" "namespace"="openshift"
"osImageVersion"="412.86.202302211547-0" "reconcileID"="e709ff0a-5664-4007-8270-
49b5dff8bae9"
```