OpenShift Container Platform 4.12

专用硬件和驱动程序启用

了解 OpenShift Container Platform 中的硬件启用

Last Updated: 2023-03-31

OpenShift Container Platform 4.12 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

Copyright © 2023 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

摘要

本文档概述 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 ... 3

第 2 章 驱动程序工具包 ... 4

2.1. 关于驱动程序工具包 4
2.2. 拉取 DRIVER TOOLKIT 容器镜像 5
2.3. 使用 DRIVER TOOLKIT 6
2.4. 其他资源 9

第 3 章 NODE FEATURE DISCOVERY OPERATOR ... 10

3.1. 关于 NODE FEATURE DISCOVERY OPERATOR 10
3.2. 安装 NODE FEATURE DISCOVERY OPERATOR 10
3.3. 使用 NODE FEATURE DISCOVERY OPERATOR 12
3.4. 配置 NODE FEATURE DISCOVERY OPERATOR 15
3.5. 使用 NFD TOPOLOGY UPDATER 20

第 4 章 内核模块管理 OPERATOR ... 24

4.1. 关于内核模块管理 OPERATOR 24
4.2. 安装内核模块管理 OPERATOR 24
4.3. 使用内核模块管理(KMM)的签名 28
4.4. 为 SECUREBOOT 添加密钥 28
4.5. 签名预构建驱动程序容器 30
4.6. 构建并签署 MODULELOADER 容器镜像 31
4.7. 调试和故障排除 33

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 :op-system-first:）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** 中的次版本标记，例如：**registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.12**。

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

- 您已从 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，输入以下命令：

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.12.z-x86_64 --image-for=driver-toolkit
     ```

   - 对于 ARM 镜像，输入以下命令：

     ```
     $ oc adm release info quay.io/openshift-release-dev/ocp-release:4.12.z-aarch64 --image-for=driver-toolkit
     ```

   输出示例

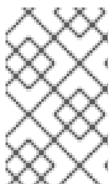
   ```
   quay.io/openshift-release-dev/ocp-v4.0-art-dev@sha256:0fd84aee79606178b6561ac71f8540f404d518ae5deff45f6d6ac8f02636c7f4
   ```

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。

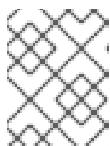
流程

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** 权限的用户身份登录。
- 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:v4.12
    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. 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。

3.5.1. 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
```

3.5.2. NFD Topology Updater 命令行标志

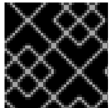
要查看可用的命令行标志，请运行 **nfd-topology-updater -help** 命令。例如，在 podman 容器中，运行以下命令：

```
$ 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。

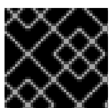
示例

```
$ 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。

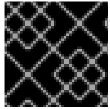
示例

```
$ 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。

示例

```
$ 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**

示例

```
$ 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**

示例

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

3.5.2.1. -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**

示例

```
$ 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**

示例

```
$ 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**

示例

```
$ 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

示例

```
$ 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**

示例

```
$ 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：*

示例

```
$ 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。

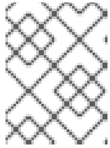
流程

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 **Install Operator** 页面中，选择 **Installation mode** 作为 **A specific namespace on the cluster**。
    d. 在 **Installed Namespace** 列表中，选择 **openshift-kmm** 命名空间。
    e. 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-manager
```

输出示例

```
NAME                              READY UP-TO-DATE  AVAILABLE AGE
kmm-operator-controller-manager   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-manager -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-manager
```

输出示例

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

Operator 可用。

4.3. 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符号部分 签名 内核模块。

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**。

4.4. 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=key=
<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) (1) (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>
```

4.4.1. 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 **-----BEGIN PRIVATE KEY-----** 和 **-----END PRIVATE KEY-----** 行中的密钥。

4.5. 签名预构建驱动程序容器

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/私钥对添加为带有私钥名称 - **密钥** （公钥的证书）的 secret。**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，内核模块管理(KMM)应该部署 DaemonSet，它将签名的 kmods 加载到与选择器匹配的所有节点中。驱动程序容器应在其 MOK 数据库中具有公钥的任何节点上运行，以及所有未启用 secure-boot（忽略签名）的节点。它们应该无法在启用了 secure-boot 的任何上加载，但其 MOK 数据库中没有该密钥。

先决条件

- **keySecret 和 certSecret secret 已创建。**

流程

1. **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: '<your 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 e.g. quay.io/myuser/my-driver:<kernelversion>-
signed>
          sign:
            # the image containing the unsigned kmods (we need this because we are not
building the kmods within the cluster)
            unsignedImage: <image name e.g. quay.io/myuser/my-driver:<kernelversion> >
            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. modprobe - 要载入的 kmod 的名称。

4.6. 构建并签署 MODULELOADER 容器镜像

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)构建一个名为 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 DaemonSet 对象载入的第二个镜像，并将 kmods 部署到集群节点。

签名后，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 已创建。

流程

1. 应用 YAML 文件：

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-module-dockerfile
  namespace: default (1)
data:
  Dockerfile: |
    ARG DTK_AUTO
    ARG KERNEL_VERSION
    FROM ${DTK_AUTO} as builder
    WORKDIR /build/
    RUN git clone -b main --single-branch https://github.com/rh-ecosystem-edge/kernel-
module-management.git
    WORKDIR kernel-module-management/ci/kmm-kmod/
    RUN make
    FROM registry.access.redhat.com/ubi8/ubi:latest
    ARG KERNEL_VERSION
    RUN yum -y install kmod && yum clean all
    RUN mkdir -p /opt/lib/modules/${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: default (2)
spec:
  moduleLoader:
    serviceAccountName: default (3)
    container:
      modprobe:
        moduleName: simple_kmod
```

```
      kernelMappings:
        - regexp: '^.*\.x86_64$'
          containerImage: < the name of the final driver container to produce>
          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

namespace - 使用有效的命名空间替换 default。

3

serviceAccountName - 默认 serviceAccountName 没有运行特权的模块所需的权限。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

imageRepoSecret - 用作 DaemonSet 对象中的 imagePullSecrets，并为构建和签名功能拉取和推送。

其他资源

- 有关创建服务帐户的详情，请参考 创建服务帐户。

4.7. 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
```