



OpenShift Container Platform 4.12

専用のハードウェアおよびドライバーの有効化

OpenShift Container Platform でのハードウェアの有効化に関する説明

OpenShift Container Platform 4.12 専用のハードウェアおよびドライバーの有効化

OpenShift Container Platform でのハードウェアの有効化に関する説明

法律上の通知

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.

概要

本書では、OpenShift Container Platform でのハードウェアの有効化に関して解説します。

目次

第1章 専用のハードウェアおよびドライバーの有効化	3
第2章 ドライバーツールキット	4
2.1. DRIVER TOOLKIT について	4
2.2. DRIVER TOOLKIT コンテナイメージのプル	5
2.3. DRIVER TOOLKIT の使用	6
2.4. 関連情報	10
第3章 NODE FEATURE DISCOVERY OPERATOR	11
3.1. NODE FEATURE DISCOVERY OPERATOR について	11
3.2. NODE FEATURE DISCOVERY OPERATOR のインストール	11
3.3. NODE FEATURE DISCOVERY OPERATOR の使用	13
3.4. NODE FEATURE DISCOVERY OPERATOR の設定	16
3.5. NFD トポロジーアップデートの使用	21
第4章 KERNEL MODULE MANAGEMENT OPERATOR	26
4.1. KERNEL MODULE MANAGEMENT OPERATOR について	26
4.2. KERNEL MODULE MANAGEMENT OPERATOR のインストール	26
4.3. カーネルモジュールのデプロイ	30
4.4. MODULELOADER イメージの使用	35
4.5. カーネルモジュール管理 (KMM) による署名の使用	38
4.6. SECUREBOOT の鍵の追加	38
4.7. ビルド済みのドライバーコンテナに署名する	39
4.8. MODULELOADER コンテナイメージのビルドと署名	41
4.9. デバッグとトラブルシューティング	42
4.10. KMM ファームウェアのサポート	42
4.11. KMM のトラブルシューティング	44

第1章 専用のハードウェアおよびドライバーの有効化

Driver Toolkit (DTK) は、OpenShift Container Platform ペイロードのコンテナイメージであり、ドライバーコンテナを構築するベースイメージとして使用することが目的です。Driver Toolkit イメージには、カーネルモジュールをビルドまたはインストールするための依存関係として一般的に必要なカーネルパッケージと、ドライバーコンテナに必要ないくつかのツールが含まれています。これらのパッケージのバージョンは、対応する OpenShift Container Platform リリースの RHCOS ノードで実行されているカーネルのバージョンと一致します。

ドライバーコンテナは、コンテナオペレーティングシステム (Red Hat Enterprise Linux CoreOS (RHCOS)) でツリー外のカーネルモジュールとドライバーをビルドおよびデプロイメントするために使用されるコンテナイメージです。カーネルモジュールおよびドライバーは、レベルの高い権限で、オペレーティングシステムカーネル内で実行されるソフトウェアライブラリーです。また、カーネル機能の拡張や、新しいデバイスの制御に必要なハードウェア固有のコードを提供します。例としては、field-programmable gate arrays (FPGA) や graphics processing units (GPU) などのハードウェアデバイスや、ソフトウェア定義のストレージソリューションなどがあります。これらはすべて、クライアントマシンでカーネルモジュールを必要とします。ドライバーコンテナは、OpenShift Container Platform デプロイメントでこれらのテクノロジーを有効にするために使用されるソフトウェアスタックの最初の階層です。

第2章 ドライバーツールキット

Driver Toolkit について、およびドライバーコンテナのベースイメージとしてそれを使用して、OpenShift Container Platform デプロイメントで特別なソフトウェアおよびハードウェアデバイスを有効にする方法について説明します。

2.1. DRIVER TOOLKIT について

背景情報

Driver Toolkit は、ドライバーコンテナをビルドできるベースイメージとして使用する OpenShift Container Platform ペイロードのコンテナイメージです。Driver Toolkit イメージには、カーネルモジュールをビルドまたはインストールするための依存関係として一般的に必要なカーネルパッケージと、ドライバーコンテナに必要ないくつかのツールが含まれています。これらのパッケージのバージョンは、対応する OpenShift Container Platform リリースの Red Hat Enterprise Linux CoreOS (RHCOS) ノードで実行されているカーネルバージョンと同じです。

ドライバーコンテナは、RHCOS などのコンテナオペレーティングシステムで out-of-tree カーネルモジュールをビルドしてデプロイするのに使用するコンテナイメージです。カーネルモジュールおよびドライバーは、レベルの高い権限で、オペレーティングシステムカーネル内で実行されるソフトウェアライブラリーです。また、カーネル機能の拡張や、新しいデバイスの制御に必要なハードウェア固有のコードを提供します。例として、Field Programmable Gate Arrays (FPGA) または GPU などのハードウェアデバイスや、クライアントマシンでカーネルモジュールを必要とする Lustre parallel ファイルシステムなどのソフトウェア定義のストレージ (SDS) ソリューションなどがあります。ドライバーコンテナは、Kubernetes でこれらの技術を有効にするために使用されるソフトウェアスタックの最初の層です。

Driver Toolkit のカーネルパッケージの一覧には、以下とその依存関係が含まれます。

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

また、Driver Toolkit には、対応するリアルタイムカーネルパッケージも含まれています。

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

Driver Toolkit には、カーネルモジュールのビルドおよびインストールに一般的に必要なツールが複数あります。たとえば、以下が含まれます。

- **elfutils-libelf-devel**
- **kmod**
- **binutils-kabi-dw**

- **kernel-abi-whitelists**
- 上記の依存関係

目的

Driver Toolkit がリリースされる前は、[エンタイトルメントのあるビルド](#)を使用するか、ホストの **machine-os-content** のカーネル RPM からインストールして、Pod またはビルド設定のカーネルパッケージを OpenShift Container Platform にインストールできていました。Driver Toolkit を使用すると、エンタイトルメントステップがなくなりプロセスが単純化され、Pod で machine-os-content にアクセスする特権操作を回避できます。Driver Toolkit は、プレリリース済みの OpenShift Container Platform バージョンにアクセスできるパートナーも使用でき、今後の OpenShift Container Platform リリース用にハードウェアデバイスのドライバーコンテナを事前にビルドできます。

Driver Toolkit は Kernel Module Management (KMM) でも使用されます。Kernel Module Management (KMM) は、現在 OperatorHub でコミュニティ Operator として利用できます。KMM は、out-of-tree およびサードパーティーのカーネルドライバー、および基礎となるオペレーティングシステムのサポートソフトウェアをサポートします。ユーザーは、KMM のレシピを作成してドライバーコンテナを構築してデプロイしたり、デバイスプラグインやメトリックなどのソフトウェアをサポートしたりできます。モジュールには、ビルド設定を追加して、Driver Toolkit をベースにドライバーコンテナをビルドできます。または KMM で事前ビルドされたドライバーコンテナをデプロイできます。

2.2. DRIVER TOOLKIT コンテナイメージのプル

driver-toolkit イメージは、[Red Hat Ecosystem Catalog](#) および [OpenShift Container Platform リリースペイロードのコンテナイメージ](#) セクションから入手できます。OpenShift Container Platform の最新のマイナーリリースに対応するイメージは、カタログのバージョン番号でタグ付けされます。特定のリリースのイメージ URL は、**oc adm** CLI コマンドを使用して確認できます。

2.2.1. registry.redhat.io からの Driver Toolkit コンテナイメージのプル

podman または OpenShift Container Platform を使用して **registry.redhat.io** から **driver-toolkit** イメージをプルする手順は、[Red Hat Ecosystem Catalog](#) を参照してください。最新のマイナーリリースのドライバーツールキットイメージは、**registry.redhat.io** のマイナーリリースバージョンでタグ付けされます (例: **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.12**)。

2.2.2. ペイロードでの Driver Toolkit イメージ URL の検索

前提条件

- [Red Hat OpenShift Cluster Manager](#) からイメージプルシークレットを取得している。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. **oc adm** コマンドを使用して、特定のリリースに対応する **driver-toolkit** のイメージ URL を抽出します。

- x86 イメージの場合は、次のコマンドを入力します。

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

- ARM イメージの場合は、次のコマンドを入力します。

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

出力例

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

2. OpenShift Container Platform のインストールに必要なプルシークレットなど、有効なプルシークレットを使用してこのイメージを取得します。

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

2.3. DRIVER TOOLKIT の使用

たとえば、Driver Toolkit は **simple-kmod** と呼ばれる単純なカーネルモジュールを構築するベースイメージとして使用できます。



注記

Driver Toolkit には、カーネルモジュールに署名するために必要な依存関係である **openssl**、**mokutil**、および **keyutils** が含まれています。ただし、この例では、**simple-kmod** カーネルモジュールは署名されていないため、**Secure Boot** が有効になっているシステムにはロードできません。

2.3.1. クラスターでの **simple-kmod** ドライバーコンテナをビルドし、実行します。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- クラスターのイメージレジストリー Operator の状態を **Managed** に設定している。
- OpenShift CLI (**oc**) がインストールされている。
- **cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

namespace を作成します。以下はその例です。

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

1. YAML は、**simple-kmod** ドライバーコンテナイメージを保存する **ImageStream** と、コンテナをビルドする **BuildConfig** を定義します。この YAML を **0000-buildconfig.yaml.template** として保存します。

```
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. 以下のコマンドで、DRIVER_TOOLKIT_IMAGE の代わりに、実行中の OpenShift Container Platform バージョンのドライバーツールキットイメージを置き換えます。

```
$ 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. 以下でイメージストリームおよびビルド設定を作成します。

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

4. ビルダー Pod が正常に完了したら、ドライバーコンテナイメージを **DaemonSet** としてデプロイします。

- a. ホスト上でカーネルモジュールを読み込むには、特権付きセキュリティコンテキストでドライバーコンテナを実行する必要があります。以下の YAML ファイルには、ドライバーコンテナを実行するための RBAC ルールおよび **DaemonSet** が含まれます。この YAML を **1000-drivercontainer.yaml** として保存します。

```
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. RBAC ルールおよびデーモンセットを作成します。

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

5. Pod がワーカーノードで実行された後に、**simple_kmod** カーネルモジュールが **lsmod** のホストマシンで正常に読み込まれることを確認します。

- a. Pod が実行されていることを確認します。

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

出力例

```

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. ドライバーコンテナ Pod で **lsmod** コマンドを実行します。

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

出力例

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

2.4. 関連情報

- クラスターのレジストリーストレージの設定に関する詳細は、[OpenShift Container Platform のイメージレジストリー Operator](#) を参照してください。

第3章 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery (NFD) Operator および、これを使用して Node Feature Discovery (ハードウェア機能やシステム設定を検出するための Kubernetes アドオン) をオーケストレーションしてノードレベルの情報を公開する方法を説明します。

3.1. NODE FEATURE DISCOVERY OPERATOR について

Node Feature Discovery Operator (NFD) は、ハードウェア固有の情報でノードにラベルを付け、OpenShift Container Platform クラスターのハードウェア機能と設定の検出を管理します。NFD は、PCI カード、カーネル、オペレーティングシステムのバージョンなど、ノード固有の属性でホストにラベルを付けます。

NFD Operator は、Node Feature Discovery と検索して Operator Hub で確認できます。

3.2. NODE FEATURE DISCOVERY OPERATOR のインストール

Node Feature Discovery (NFD) Operator は、NFD デモンセットの実行に必要なすべてのリソースをオーケストレーションします。クラスター管理者は、OpenShift Container Platform CLI または Web コンソールを使用して NFD Operator をインストールできます。

3.2.1. CLI を使用した NFD Operator のインストール

クラスター管理者は、CLI を使用して NFD Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスター。
- OpenShift CLI (**oc**) をインストールしている。
- **cluster-admin** 権限を持つユーザーとしてログインしている。

手順

1. NFD Operator の namespace を作成します。
 - a. **openshift-nfd** namespace を定義する以下の **Namespace** カスタムリソース (CR) を作成し、YAML を **nfd-namespace.yaml** ファイルに保存します。

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

- b. 以下のコマンドを実行して namespace を作成します。

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

2. 以下のオブジェクトを作成して、直前の手順で作成した namespace に NFD Operator をインストールします。
 - a. 以下の **OperatorGroup** CR を作成し、YAML を **nfd-operatorgroup.yaml** ファイルに保存します。

```

apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  generateName: openshift-nfd-
  name: openshift-nfd
  namespace: openshift-nfd
spec:
  targetNamespaces:
    - openshift-nfd

```

- b. 以下のコマンドを実行して **OperatorGroup** CR を作成します。

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

- c. 以下の **Subscription** CR を作成し、YAML を **nfd-sub.yaml** ファイルに保存します。

Subscription の例

```

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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

- e. **openshift-nfd** プロジェクトに切り替えます。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、以下を実行します。

```
$ oc get pods
```

出力例

```

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

```

正常にデプロイされると、**Running** ステータスが表示されます。

3.2.2. Web コンソールでの NFD Operator のインストール

クラスター管理者は、Web コンソールを使用して NFD Operator をインストールできます。

手順

1. OpenShift Container Platform Web コンソールで、**Operators → OperatorHub** をクリックします。
2. 利用可能な Operator の一覧から **Node Feature Discovery** を選択してから **Install** をクリックします。
3. **Install Operator** ページで **A specific namespace on the cluster** を選択し、**Install** をクリックします。namespace が作成されるため、これを作成する必要はありません。

検証

以下のように、NFD Operator が正常にインストールされていることを確認します。

1. **Operators → Installed Operators** ページに移動します。
2. **Status** が **InstallSucceeded** の **Node Feature Discovery** が **openshift-nfd** プロジェクトに一覧表示されていることを確認します。



注記

インストール時に、Operator は **Failed** ステータスを表示する可能性があります。インストールが後に **InstallSucceeded** メッセージを出して正常に実行される場合は、**Failed** メッセージを無視できます。

トラブルシューティング

Operator がインストール済みとして表示されない場合に、さらにトラブルシューティングを実行します。

1. **Operators → Installed Operators** ページに移動し、**Operator Subscriptions** および **Install Plans** タブで **Status** にエラーがあるかどうかを検査します。
2. **Workloads → Pods** ページに移動し、**openshift-nfd** プロジェクトで Pod のログを確認します。

3.3. NODE FEATURE DISCOVERY OPERATOR の使用

Node Feature Discovery (NFD) Operator は、**NodeFeatureDiscovery** CR を監視して Node-Feature-Discovery デモンセットの実行に必要な全リソースをオーケストレーションします。**NodeFeatureDiscovery** CR に基づいて、Operator は任意の namespace にオペランド (NFD) コンポーネントを作成します。CR を編集して、他にあるオプションの中から、別の **namespace**、**image**、**imagePullPolicy**、および **nfd-worker-conf** を選択できます。

クラスター管理者は、OpenShift Container Platform CLI または Web コンソールを使用して **NodeFeatureDiscovery** を作成できます。

3.3.1. CLI を使用した NodeFeatureDiscovery インスタンスの作成

クラスター管理者は、CLI を使用して **NodeFeatureDiscovery** CR インスタンスを作成できます。

前提条件

- OpenShift Container Platform クラスター。
- OpenShift CLI (**oc**) をインストールしている。
- **cluster-admin** 権限を持つユーザーとしてログインしている。
- NFD Operator をインストールしている。

手順

1. 以下の **NodeFeatureDiscovery** カスタムリソース (CR) を作成し、YAML を NodeFeatureDiscovery **.yaml** ファイルに保存します。

```

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"]

```

NFD ワーカーをカスタマイズする方法は、[nfd-worker の設定ファイルリファレンス](#) を参照してください。

1. 以下のコマンドを実行し、**NodeFeatureDiscovery** CR インスタンスを作成します。

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

検証

- インスタンスが作成されたことを確認するには、以下を実行します。

```
$ oc get pods
```

出力例

```
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-vgc29	1/1	Running	0	60s
nfd-worker-xqbws	1/1	Running	0	60s

正常にデプロイされると、**Running** ステータスが表示されます。

3.3.2. Web コンソールを使用した NodeFeatureDiscovery CR の作成

手順

1. **Operators** → **Installed Operators** ページに移動します。
2. **Node Feature Discovery** を見つけ、**Provided APIs** でボックスを表示します。
3. **Create instance** をクリックします。
4. **NodeFeatureDiscovery** CR の値を編集します。
5. **Create** をクリックします。

3.4. NODE FEATURE DISCOVERY OPERATOR の設定

3.4.1. コア

core セクションには、共通の設定が含まれており、これは特定の機能ソースに固有のものではありません。

core.sleepInterval

core.sleepInterval は、次に機能検出または再検出するまでの間隔を指定するので、ノードの再ラベル付けの間隔も指定します。正の値以外は、無限のスリープ状態を意味するので、再検出や再ラベル付けは行われません。

この値は、指定されている場合は、非推奨の **--sleep-interval** コマンドラインフラグで上書きされます。

使用例

```
core:
  sleepInterval: 60s 1
```

デフォルト値は **60s** です。

core.sources

core.sources は、有効な機能ソースの一覧を指定します。特殊な値 **all** はすべての機能ソースを有効にします。

この値は、指定されている場合は非推奨の **--sources** コマンドラインフラグにより上書きされます。

デフォルト: **[all]**

使用例

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

core.labelWhiteList

core.labelWhiteList は、正規表現を指定してラベル名に基づいて機能ラベルをフィルターします。一致しないラベルは公開されません。

正規表現は、ラベルのベース名 ('/' の後に名前の一部) だけを照合します。ラベルの接頭辞または namespace は省略されます。

この値は、指定されている場合は、非推奨の **--label-whitelist** コマンドラインフラグで上書きされます。

デフォルト: **null**

使用例

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

core.noPublish

core.noPublish を **true** に設定すると、**nfd-master** による全通信が無効になります。これは実質的にはドライランフラグです。**nfd-worker** は通常通り機能検出を実行しますが、ラベル付け要求は **nfd-master** に送信されます。

この値は、指定されている場合には、**--no-publish** コマンドラインフラグにより上書きされます。

例:

使用例

```
core:
  noPublish: true ❶
```

デフォルト値は **false** です。

core.klog

以下のオプションは、実行時にほとんどを動的に調整できるロガー設定を指定します。

ロガーオプションはコマンドラインフラグを使用して指定することもできますが、対応する設定ファイルオプションよりもこちらが優先されます。

core.klog.addDirHeader

true に設定すると、**core.klog.addDirHeader** がファイルディレクトリーをログメッセージのヘッダーに追加します。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.alsologtostderr

標準エラーおよびファイルにロギングします。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.logBacktraceAt

file:N の行にロギングが到達すると、スタックトレースを出力します。

デフォルト: **empty**

ランタイム設定可能: yes

core.klog.logDir

空でない場合は、このディレクトリーにログファイルを書き込みます。

デフォルト: **empty**

ランタイム設定可能: no

core.klog.logFile

空でない場合は、このログファイルを使用します。

デフォルト: **empty**

ランタイム設定可能: no

core.klog.logFileMaxSize

core.klog.logFileMaxSize は、ログファイルの最大サイズを定義します。単位はメガバイトです。値が 0 の場合には、最大ファイルサイズは無制限になります。

デフォルト: **1800**

ランタイム設定可能: no

core.klog.logtostderr

ファイルの代わりに標準エラーにログを記録します。

デフォルト: **true**

ランタイム設定可能: yes

core.klog.skipHeaders

core.klog.skipHeaders が **true** に設定されている場合には、ログメッセージでヘッダー接頭辞を使用しません。

デフォルト: **false**

ランタイム設定可能: yes

core.klog.skipLogHeaders

core.klog.skipLogHeaders が **true** に設定されている場合は、ログファイルを表示する時にヘッダーは使用されません。

デフォルト: **false**

ランタイム設定可能: no

core.klog.stderrthreshold

このしきい値以上のログは stderr になります。

デフォルト: **2**

ランタイム設定可能: yes

core.klog.v

core.klog.v はログレベルの詳細度の数値です。

デフォルト: **0**

ランタイム設定可能: yes

core.klog.vmodule

core.klog.vmodule は、ファイルでフィルターされたログギングの **pattern=N** 設定 (コンマ区切りの一覧) です。

デフォルト: **empty**

ランタイム設定可能: yes

3.4.2. ソース

sources セクションには、機能ソース固有の設定パラメーターが含まれます。

sources.cpu.cpuid.attributeBlacklist

このオプションに記述されている **cpuid** 機能は公開されません。

この値は、指定されている場合は **source.cpu.cpuid.attributeWhitelist** によって上書きされます。

デフォルト: **[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]**

使用例

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

sources.cpu.cpuid.attributeWhitelist

このオプションに記述されている **cpuid** 機能のみを公開します。

sources.cpu.cpuid.attributeWhitelist は **sources.cpu.cpuid.attributeBlacklist** よりも優先されます。

デフォルト: **empty**

使用例

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

sources.kernel.kconfigFile

sources.kernel.kconfigFile は、カーネル設定ファイルのパスです。空の場合には、NFD は一般的な標準場所で検索を実行します。

デフォルト: **empty**

使用例

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

sources.kernel.configOpts

sources.kernel.configOpts は、機能ラベルとして公開するカーネル設定オプションを表します。

デフォルト: **[NO_HZ, NO_HZ_IDLE, NO_HZ_FULL, PREEMPT]**

使用例

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

sources.pci.deviceClassWhitelist

sources.pci.deviceClassWhitelist は、ラベルを公開する [PCI デバイスクラス ID](#) の一覧です。メインクラスとしてののみ (例: **03**) か、完全なクラスサブクラスの組み合わせ (例: **0300**) として指定できます。前者は、すべてのサブクラスが許可されていることを意味します。ラベルの形式は、**deviceLabelFields** でさらに設定できます。

デフォルト: **["03", "0b40", "12"]**

使用例

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

sources.pci.deviceLabelFields

sources.pci.deviceLabelFields は、機能ラベルの名前を構築する時に使用する PCI ID フィールドのセットです。有効なフィールドは **class**、**vendor**、**device**、**subsystem_vendor** および **subsystem_device** です。

デフォルト: **[class, vendor]**

使用例

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

上記の設定例では、NFD は **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true** などのラベルを公開します。

sources.usb.deviceClassWhitelist

sources.usb.deviceClassWhitelist は、機能ラベルを公開する USB [デバイスクラス ID](#) の一覧です。ラベルの形式は、**deviceLabelFields** でさらに設定できます。

デフォルト: **["0e", "ef", "fe", "ff"]**

使用例

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

sources.usb.deviceLabelFields

sources.usb.deviceLabelFields は、機能ラベルの名前を作成する USB ID フィールドのセットです。有効なフィールドは **class**、**vendor**、および **device** です。

デフォルト: **[class, vendor, device]**

使用例

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

上記の設定例では、NFD は **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true** などのラベルを公開します。

sources.custom

sources.custom は、ユーザー固有のラベルを作成するためにカスタム機能ソースで処理するルールの一覧です。

デフォルト: **empty**

使用例

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

3.5. NFD トポロジーアップデートの使用

Node Feature Discovery (NFD) Topology Updater は、ワーカーノードに割り当てられたリソースを調べるデーモンです。これは、ゾーンごとに新規 Pod に割り当てることができるリソースに対応し、ゾーンを Non-Uniform Memory Access (NUMA) ノードにすることができます。NFD Topology Updater は、情報を nfd-master に伝達します。これにより、クラスター内のすべてのワーカーノードに対応する **NodeResourceTopology** カスタムリソース (CR) が作成されます。NFD Topology Updater のインスタンスが 1 台、クラスターの各ノードで実行されます。

NFD で Topology Updater ワーカーを有効にするには **Node Feature Discovery Operator の使用** のセクションで説明されているように、**Node Feature Discovery CR** で **topologyupdater** 変数を **true** に設定します。

3.5.1. NodeResourceTopology CR

NFD Topology Updater を使用して実行すると、NFD は、次のようなノードリソースハードウェアトポロジーに対応するカスタムリソースインスタンスを作成します。

```
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 コマンドラインフラグ

使用可能なコマンドラインフラグを表示するには、**nfd-topology-updater-help** コマンドを実行します。たとえば、podman コンテナで、次のコマンドを実行します。

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

-ca-file

-ca-file フラグは、**-cert-file** フラグおよび `-key-file`` フラグとともに、NFD トポロジーアップデートで相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、nfd-master の信頼性検証に使用する TLS ルート証明書を指定します。

デフォルト: empty



重要

-ca-file フラグは、**-cert-file** と **-key-file** フラグと一緒に指定する必要があります。

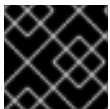
例

```
$ 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 フラグは、**-ca-file** と **-key-file flags** とともに、NFD トポロジーアップデータで相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、送信要求の認証時に提示する TLS 証明書を指定します。

デフォルト: empty



重要

-cert-file フラグは、**-ca-file** と **-key-file** フラグと一緒に指定する必要があります。

例

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

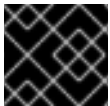
-h, -help

使用法を出力して終了します。

-key-file

-key-file フラグは、**-ca-file** と **-cert-file** フラグとともに、NFD Topology Updater で相互 TLS 認証を制御する 3 つのフラグの 1 つです。このフラグは、指定の証明書ファイルまたは **-cert-file** に対応する秘密鍵 (送信要求の認証に使用) を指定します。

デフォルト: empty



重要

-key-file フラグは、**-ca-file** と **-cert-file** フラグと一緒に指定する必要があります。

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 は、Kubelet の設定ファイルへのパスを指定します。

デフォルト: /host-var/lib/kubelet/config.yaml

例

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

-no-publish

-no-publish フラグは、nfd-master とのすべての通信を無効にし、nfd-topology-updater のドライランフラグにします。NFD Topology Updater は、リソースハードウェアトポロジー検出を正常に実行しますが、CR 要求は nfd-master に送信されません。

デフォルト: **false**

例

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

3.5.2.1. **-oneshot**

-oneshot フラグを使用すると、リソースハードウェアトポロジーの検出が1回行われた後も、NFD Topology Updater が終了します。

デフォルト: **false**

例

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

-podresources-socket

-podresources-socket フラグは、kubelet が gRPC サービスをエクスポートして使用中の CPU とデバイスを検出できるようにし、それらのメタデータを提供する Unix ソケットへのパスを指定します。

デフォルト: **/host-var/liblib/kubelet/pod-resources/kubelet.sock**

例

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

-server

-server フラグは、接続する nfd-master エンドポイントのアドレスを指定します。

デフォルト: **localhost:8080**

例

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

-server-name-override

-server-name-override フラグは、nfd-master TLS 証明書から必要とされるコモンネーム (CN) を指定します。このフラグは、主に開発とデバッグを目的としています。

デフォルト: **empty**

例

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

-sleep-interval

-sleep-interval フラグは、リソースハードウェアトポロジーの再検査とカスタムリソースの更新の間隔を指定します。正でない値は、スリープ間隔が無限であることを意味し、再検出は行われません。

デフォルト: **60s**。

例

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

-version

バージョンを出力して終了します。

-watch-namespace

-watch-namespace フラグは namespace を指定して、指定された namespace で実行されている Pod に対してのみリソースハードウェアトポロジーの検査が行われるようにします。指定された namespace で実行されていない Pod は、リソースアカウンティングでは考慮されません。これは、テストとデバッグの目的で特に役立ちます。* 値は、全 namespace に含まれるすべての Pod がアカウンティングプロセス中に考慮されることを意味します。

デフォルト: *

例

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

第4章 KERNEL MODULE MANAGEMENT OPERATOR

Kernel Module Management (KMM) Operator について、およびそれを使用して out-of-tree のカーネルモジュールとデバイスプラグインを OpenShift Container Platform クラスターにデプロイする方法について説明します。

4.1. KERNEL MODULE MANAGEMENT OPERATOR について

Kernel Module Management (KMM) Operator は、OpenShift Container Platform クラスター上の out-of-tree のカーネルモジュールとデバイスプラグインを管理、ビルド、署名、およびデプロイします。

KMM は、ツリー外のカーネルモジュールとそれに関連するデバイスプラグインを記述する新しい **Module** CRD を追加します。モジュール リソースを使用して、モジュールをロードする方法を設定し、カーネルバージョンの **ModuleLoader** イメージを定義し、特定のカーネルバージョンのモジュールをビルドして署名するための指示を含めることができます。

KMM は、任意のカーネルモジュールに対して一度に複数のカーネルバージョンに対応できるように設計されているため、ノードのシームレスなアップグレードとアプリケーションのダウンタイムの削減が可能になります。

4.2. KERNEL MODULE MANAGEMENT OPERATOR のインストール

クラスター管理者は、OpenShift CLI または Web コンソールを使用して Kernel Module Management (KMM) Operator をインストールできます。

KMM Operator は、OpenShift Container Platform 4.12 以降でサポートされています。バージョン 4.11 に KMM をインストールする場合、特に追加手順は必要ありません。KMM をバージョン 4.10 以前にインストールする方法の詳細は、「以前のバージョンの OpenShift Container Platform への Kernel Module Management Operator のインストール」セクションを参照してください。

4.2.1. Web コンソールを使用した Kernel Module Management Operator のインストール

クラスター管理者は、OpenShift Container Platform Web コンソールを使用して Kernel Module Management (KMM) Operator をインストールできます。

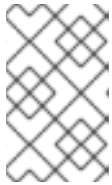
手順

1. OpenShift Container Platform Web コンソールにログインします。
2. Kernel Module Management Operator をインストールします。
 - a. OpenShift Container Platform Web コンソールで、**Operators** → **OperatorHub** をクリックします。
 - b. 使用可能な Operator のリストから **Kernel Module Management Operator** を選択し、**Install** をクリックします。
 - c. **Install Operator** ページで、**Installation mode** を **A specific namespace on the cluster** として選択します。
 - d. **Installed Namespace** リストから、**openshift-kmm** namespace を選択します。
 - e. **Install** をクリックします。

検証

KMM Operator が正常にインストールされたことを確認するには、以下を実行します。

1. **Operators** → **Installed Operators** ページに移動します。
2. **Kernel Module Management Operator** が **openshift-kmm** プロジェクトにリストされ、**Status** が **InstallSucceeded** であることを確認します。



注記

インストール時に、Operator は **Failed** ステータスを表示する可能性があります。インストールが後に **InstallSucceeded** メッセージを出して正常に実行される場合は、**Failed** メッセージを無視できます。

トラブルシューティング

1. Operator のインストールに関する問題をトラブルシューティングするには、以下を実行します。
 - a. **Operators** → **Installed Operators** ページに移動し、**Operator Subscriptions** および **Install Plans** タブで **Status** にエラーがあるかどうかを検査します。
 - b. **Workloads** → **Pods** ページに移動し、**openshift-kmm** プロジェクトで Pod のログを確認します。

4.2.2. CLI を使用した Kernel Module Management Operator のインストール

クラスター管理者は、OpenShift CLI を使用して Kernel Module Management (KMM) Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- OpenShift CLI (**oc**) がインストールされている。
- **cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

1. KMM を **openshift-kmm** namespace にインストールします。
 - a. 次の **Namespace** CR を作成し、YAML ファイル (**kmm-namespace.yaml** など) を保存します。

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

- b. 次の **OperatorGroup** CR を作成し、YAML ファイル (**kmm-op-group.yaml** など) を保存します。

```
apiVersion: operators.coreos.com/v1
```

```
kind: OperatorGroup
metadata:
  name: kernel-module-management
  namespace: openshift-kmm
```

- c. 次の **Subscription** CR を作成し、YAML ファイル (**kmm-sub.yaml** など) を保存します。

```
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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、次のコマンドを実行します。

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

出力例

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

Operator は利用可能です。

4.2.3. 以前のバージョンの OpenShift Container Platform への Kernel Module Management Operator のインストール

KMM Operator は、OpenShift Container Platform 4.12 以降でサポートされています。バージョン 4.10 以前では、新しい **SecurityContextConstraint** オブジェクトを作成し、それを Operator の **ServiceAccount** にバインドする必要があります。クラスター管理者は、OpenShift CLI を使用して Kernel Module Management (KMM) Operator をインストールできます。

前提条件

- OpenShift Container Platform クラスターが実行中である。
- OpenShift CLI (**oc**) がインストールされている。
- cluster-admin** 権限があるユーザーとして OpenShift CLI にログインしている。

手順

1. KMM を **openshift-kmm** namespace にインストールします。

- a. 次の **Namespace** CR を作成し、YAML ファイル (**kmm-namespace.yaml** ファイルなど) を保存します。

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

- b. 次の **SecurityContextConstraint** オブジェクトを作成し、YAML ファイル (**kmm-security-constraint.yaml** など) を保存します。

```
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. 次のコマンドを実行して、**SecurityContextConstraint** オブジェクトを Operator の **ServiceAccount** にバインドします。

```
$ 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. 次の **OperatorGroup** CR を作成し、YAML ファイル (**kmm-op-group.yaml** など) を保存します。

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

- e. 次の **Subscription** CR を作成し、YAML ファイル (**kmm-sub.yaml** など) を保存します。

```
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. 以下のコマンドを実行して Subscription オブジェクトを作成します。

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

検証

- Operator のデプロイメントが正常に行われたことを確認するには、次のコマンドを実行します。

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

出力例

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

Operator は利用可能です。

4.3. カーネルモジュールのデプロイ

Module リソースごとに、カーネルモジュール管理 (KMM) は多数の **DaemonSet** リソースを作成できます。

- クラスターで実行されている互換性のあるカーネルバージョンごとに1つの **ModuleLoader DaemonSet**。
- 1つのデバイスプラグイン **DaemonSet** (設定されている場合)。

モジュールローダーデーモンは、カーネルモジュールをロードするために **ModuleLoader** イメージを実行するリソースを設定します。モジュールローダーイメージは、**.ko** ファイルと **modprobe** および **sleep** バイナリーの両方を含む OCI イメージです。

モジュールローダー Pod が作成されると、Pod は **modprobe** を実行して、指定されたモジュールをカーネルに挿入します。その後、終了するまでスリープ状態になります。その場合、**ExecPreStop** フックは **modprobe -r** を実行してカーネルモジュールをアンロードします。

モジュール リソースで **.spec.devicePlugin** 属性が設定されている場合、KMM はクラスター内に **デバイスプラグイン** デモンセットを作成します。このデモンセットは、以下を対象とします。

- **Module** リソースの **.spec.selector** に一致するノード。
- カーネルモジュールがロードされたノード (モジュールローダー Pod が **Ready** 状態にある場合)。

4.3.1. Module カスタムリソース定義

Module のカスタムリソース定義 (CRD) は、モジュールローダーイメージを介してクラスター内のすべてのノードまたは選択したノードにロードできるカーネルモジュールを表します。**Module** カスタムリソース (CR) は、互換性のある1つ以上のカーネルバージョンとノードセレクターを指定します。

Module リソースの互換性のあるバージョンは、**.spec.moduleLoader.container.kernelMappings** の下にリストされています。カーネルマッピングは、**literal** バージョンと一致するか、**regex** を使用してそれらの多くを同時に一致させることができます。

Module リソースの調整ループでは、次の手順が実行されます。

1. **.spec.selector** に一致するすべてのノードをリスト表示します。
2. それらのノードで実行されているすべてのカーネルバージョンのセットを構築します。
3. 各カーネルバージョンで以下を実行します。
 - a. **.spec.moduleLoader.container.kernelMappings** を調べて、適切なコンテナイメージ名を見つけます。カーネルマッピングに **build** または **sign** が定義されていて、コンテナイメージがまだ存在しない場合は、必要に応じてビルド、署名ジョブ、またはその両方を実行します。
 - b. 前の手順で決定したコンテナイメージを使用して、モジュールローダーデーモンセットを作成します。
 - c. **.spec.devicePlugin** が定義されている場合は、**.spec.devicePlugin.container** で指定された設定を使用して、デバイスプラグインデーモンセットを作成します。
4. 以下で **garbage-collect** を実行します。
 - a. クラスター内のどのノードでも実行されていないカーネルバージョンをターゲットとする既存のデーモンセットリソース。
 - b. 成功したビルドジョブ。

- c. 成功した署名ジョブ。

4.3.2. セキュリティーおよびパーミッション



重要

カーネルモジュールのロードは、非常に機密性の高い操作です。それらがロードされると、カーネルモジュールには、ノード上であらゆる種類の操作を実行するためのすべての可能な権限が付与されます。

4.3.2.1. ServiceAccounts および SecurityContextConstraints

カーネルモジュール管理 (KMM) は、カーネルモジュールをノードにロードするための特権ワークロードを作成します。そのワークロードには、**privileged SecurityContextConstraint** (SCC) リソースの使用を許可された **ServiceAccounts** が必要です。

そのワークロードの承認モデルは、**Module** リソースの namespace とその仕様によって異なります。

- **.spec.moduleLoader.serviceAccountName** または **.spec.devicePlugin.serviceAccountName** フィールドが設定されている場合は常に使用されます。
- これらのフィールドが設定されていない場合:
 - **Module** リソースが Operator の namespace (デフォルトでは **openshift-kmm**) に作成される場合、KMM はそのデフォルトの強力な **ServiceAccounts** を使用してデーモンセットを実行します。
 - **Module** リソースが他の namespace で作成された場合、KMM はデーモンセットを namespace の **default ServiceAccount** として実行します。**Module** リソースは、**privileged** SCC の使用を手動で有効にしない限り、特権ワークロードを実行できません。



重要

openshift-kmm は信頼できる namespace です。

RBAC 権限を設定するときは、ユーザーまたは **ServiceAccount** が **openshift-kmm** namespace で **Module** リソースを作成すると、KMM がクラスター内のすべてのノードで特権ワークロードを自動的に実行することに注意してください。

ServiceAccount が **privileged** SCC を使用できるようにして、モジュールローダーまたはデバイスプラグイン Pod を実行できるようにするには、次のコマンドを使用します。

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

4.3.2.2. Pod のセキュリティー基準

OpenShift は、使用中のセキュリティーコンテキストに基づいて namespace Pod セキュリティーレベルを自動的に設定する同期メカニズムを実行します。アクションは不要です。

関連情報

- [Pod セキュリティー受付の理解と管理](#)

4.3.3. モジュール CR の例

以下は、アノテーション付きの **Module** の例です。

```

apiVersion: kmm.sigs.x-k8s.io/v1beta1
kind: Module
metadata:
  name: <my_kmod>
spec:
  moduleLoader:
    container:
      modprobe:
        moduleName: <my_kmod> ❶
        dirName: /opt ❷
        firmwarePath: /firmware ❸
        parameters: ❹
          - param=1
      kernelMappings: ❺
        - 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$' ❻
          containerImage: "some.other.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
        - regexp: '^.+$.+' ❼
          containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
      build:
        buildArgs: ❽
          - name: ARG_NAME
            value: <some_value>
        secrets:
          - name: <some_kubernetes_secret> ❾
        baseImageRegistryTLS: ❿
          insecure: false
          insecureSkipTLSVerify: false ㉑
        dockerfileConfigMap: ㉒
          name: <my_kmod_dockerfile>
      sign:
        certSecret:
          name: <cert_secret> ㉓
        keySecret:
          name: <key_secret> ㉔
        filesToSign:
          - /opt/lib/modules/${KERNEL_FULL_VERSION}/<my_kmod>.ko
      registryTLS: ㉕
        insecure: false ㉖
        insecureSkipTLSVerify: false
      serviceAccountName: <sa_module_loader> ㉗
    devicePlugin: ㉘
      container:
        image: some.registry/org/device-plugin:latest ㉙
        env:
          - name: MY_DEVICE_PLUGIN_ENV_VAR
            value: SOME_VALUE
        volumeMounts: ㉚

```

```

- 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 必須。
- 2 オプション。
- 3 オプション: **/firmware/*** をノード上の **/var/lib/firmware/** にコピーします。
- 4 オプション。
- 5 少なくとも1つのカーネル項目が必要です。
- 6 正規表現に一致するカーネルを実行しているノードごとに、KMM は **\${KERNEL_FULL_VERSION}** をカーネルバージョンに置き換えて、**containerImage** で指定されたイメージを実行する **DaemonSet** リソースを作成します。
- 7 その他のカーネルの場合は、**my-kmod** ConfigMap の Dockerfile を使用してイメージをビルドします。
- 8 オプション。
- 9 オプション: **some-kubernetes-secret** の値 は、**/run/secrets/some-kubernetes-secret** のビルド環境から取得できます。
- 10 オプション: このパラメーターは使用しないでください。**true** に設定すると、ビルドはプレーン HTTP を使用して Dockerfile **FROM** 命令のイメージをプルできます。
- 11 オプション: このパラメーターは使用しないでください。**true** に設定すると、プレーン HTTP を使用して Dockerfile **FROM** 命令でイメージをプルするときに、ビルドは TLS サーバー証明書の検証をスキップします。
- 12 必須。
- 13 必須: 鍵 **cert** を持つ公開セキュアブート鍵を保持するシークレット。
- 14 必須: 'key' という鍵が含まれるセキュアブート秘密鍵を保持するシークレット。
- 15 オプション: このパラメーターは使用しないでください。**true** に設定すると、KMM はプレーン HTTP を使用してコンテナイメージがすでに存在するかどうかを確認できます。
- 16 オプション: このパラメーターは使用しないでください。**true** に設定すると、コンテナイメージがすでに存在するかどうかを確認するときに、KMM は TLS サーバー証明書の検証をスキップします。
- 17 オプション。
- 18 オプション。

- 19 必須: デバイスプラグインセクションが存在する場合。
- 20 オプション。
- 21 オプション。
- 22 オプション。
- 23 オプション: モジュールローダーとデバイスプラグインイメージをプルするために使用されます。

4.4. MODULELOADER イメージの使用

カーネルモジュール管理 (KMM) は、専用のモジュールローダーイメージで動作します。これらは、次の要件を満たす必要がある標準の OCI イメージです。

- **.ko** ファイルは `/opt/lib/modules/${KERNEL_VERSION}` に配置する必要があります。
- **modprobe** および **sleep** バイナリーは、`$PATH` 変数で定義する必要があります。

4.4.1. depmod の実行

モジュールローダーイメージに複数のカーネルモジュールが含まれており、モジュールの1つが別のモジュールに依存している場合は、ビルドプロセスの最後に **depmod** を実行して、依存関係とマップファイルを生成することを推奨します。



注記

kernel-devel パッケージをダウンロードするには、Red Hat Subscription が必要です。

手順

1. 特定のカーネルバージョンの **modules.dep** および **.map** ファイルを生成するには、**depmod -b/opt \${KERNEL_VERSION}** を実行します。

4.4.1.1. Dockerfile の例

OpenShift Container Platform でイメージをビルドする場合は、Driver Tool Kit (DTK) の使用を検討してください。

詳細は、[資格のあるビルドの使用](#) を参照してください。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kmm-ci-dockerfile
data:
  dockerfile: |
    ARG DTK_AUTO
    FROM ${DTK_AUTO} as builder
    ARG KERNEL_VERSION
    WORKDIR /usr/src
    RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
    WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
    RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_VERSION}/build make all
```

```
FROM registry.redhat.io/ubi8/ubi-minimal
ARG KERNEL_VERSION
RUN microdnf install kmod
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_VERSION}/
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_VERSION}/
RUN depmod -b /opt ${KERNEL_VERSION}
```

関連情報

- [ドライバーツールキット](#)

4.4.2. クラスターでのビルド

KMM は、クラスター内にモジュールローダーイメージを構築できます。次のガイドラインに従ってください。

- カーネルマッピングの build セクションを使用して **build** 命令を提供します。
- コンテナイメージの **Dockerfile** を **ConfigMap** リソースの **dockerfile** キーの下にコピーします。
- **ConfigMap** が **Module** と同じ namespace にあることを確認します。

KMM は、**containerImage** フィールドで指定されたイメージ名が存在するかどうかを確認します。その場合、ビルドはスキップされます。

それ以外の場合、KMM は **Build** リソースを作成してイメージをビルドします。イメージがビルドされると、KMM はモジュールの調整を **Module** します。以下の例を参照してください。

```
# ...
- regexp: '^.+${'
containerImage: "some.registry/org/<my_kmod>:${KERNEL_FULL_VERSION}"
build:
  buildArgs: ❶
    - name: ARG_NAME
      value: <some_value>
  secrets: ❷
    - name: <some_kubernetes_secret> ❸
  baseImageRegistryTLS:
    insecure: false ❹
    insecureSkipTLSVerify: false ❺
  dockerfileConfigMap: ❻
    name: <my_kmod_dockerfile>
  registryTLS:
    insecure: false ❼
    insecureSkipTLSVerify: false ❽
```

- ❶ オプション。
- ❷ オプション。
- ❸ **/run/secrets/some-kubernetes-secret** としてビルド Pod にマウントされます。

- 4 オプション: このパラメーターは使用しないでください。**true** に設定すると、ビルドはプレーン HTTP を使用して Dockerfile **FROM** 命令でイメージをプルできます。
- 5 オプション: このパラメーターは使用しないでください。**true** に設定すると、プレーン HTTP を使用して Dockerfile **FROM** 命令でイメージをプルするときに、ビルドは TLS サーバー証明書の検証をスキップします。
- 6 必須。
- 7 オプション: このパラメーターは使用しないでください。**true** に設定すると、KMM はプレーン HTTP を使用してコンテナイメージがすでに存在するかどうかを確認できます。
- 8 オプション: このパラメーターは使用しないでください。**true** に設定すると、コンテナイメージがすでに存在するかどうかを確認するときに、KMM は TLS サーバー証明書の検証をスキップします。

関連情報

- [Build configuration resources](#)

4.4.3. Driver Toolkit の使用

Driver Toolkit (DTK) は、ビルドモジュールローダーイメージをビルドするための便利なベースイメージです。これには、クラスターで現在実行されている OpenShift バージョンのツールとライブラリーが含まれています。

手順

マルチステージの **Dockerfile** の最初のステージとして DTK を使用します。

1. カーネルモジュールをビルドします。
2. **.ko** ファイルを **ubi-minimal** などの小さなエンドユーザーイメージにコピーします。
3. クラスター内ビルドで DTK を利用するには、**DTK_AUTO** ビルド引数を使用します。この値は、**Build** リソースの作成時に KMM によって自動的に設定されます。以下の例を参照してください。

```
ARG DTK_AUTO
FROM ${DTK_AUTO} as builder
ARG KERNEL_VERSION
WORKDIR /usr/src
RUN ["git", "clone", "https://github.com/rh-ecosystem-edge/kernel-module-management.git"]
WORKDIR /usr/src/kernel-module-management/ci/kmm-kmod
RUN KERNEL_SRC_DIR=/lib/modules/${KERNEL_VERSION}/build make all
FROM registry.redhat.io/ubi8/ubi-minimal
ARG KERNEL_VERSION
RUN microdnf install kmod
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_a.ko
/opt/lib/modules/${KERNEL_VERSION}/
COPY --from=builder /usr/src/kernel-module-management/ci/kmm-kmod/kmm_ci_b.ko
/opt/lib/modules/${KERNEL_VERSION}/
RUN depmod -b /opt ${KERNEL_VERSION}
```

関連情報

- [ドライバーツールキット](#)

4.5. カーネルモジュール管理 (KMM) による署名の使用

セキュアブートが有効なシステムでは、すべてのカーネルモジュール (kmods) は、マシン所有者の鍵 (MOK) データベースに登録された公開/秘密鍵のペアで署名する必要があります。ディストリビューションの一部として配布されるドライバーは、ディストリビューションの秘密鍵によってすでに署名されている必要がありますが、ツリー外でビルドされたカーネルモジュールの場合、KMM はカーネルマッピングの **sign** セクションを使用したカーネルモジュールへの署名をサポートします。

セキュアブートの使用の詳細は、[公開鍵と秘密鍵のペアの生成](#) を参照してください。

前提条件

- 正しい (DER) 形式の公開秘密鍵ペア。
- 公開鍵が MOK データベースに登録されている、少なくとも1つのセキュアブート対応ノード。
- ビルド済みのドライバーコンテナイメージ、またはクラスター内でビルドするために必要なソースコードと **Dockerfile** のいずれか。

4.6. SECUREBOOT の鍵の追加

KMM Kernel Module Management (KMM) を使用してカーネルモジュールに署名するには、証明書と秘密鍵が必要です。これらの作成方法の詳細は、[公開鍵と秘密鍵のペアの生成](#) を参照してください。

公開鍵と秘密鍵のペアを抽出する方法の詳細は、[秘密鍵を使用してカーネルモジュールに署名する](#) を参照してください。手順 1~4 を使用して、キーをファイルに抽出します。

手順

1. 証明書を含む **sb_cert.cer** ファイルと、秘密鍵を含む **sb_cert.priv** ファイルを作成します。

```
$ 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. 次のいずれかの方法を使用して、ファイルを追加します。

- ファイルを [シークレット](#) として直接追加します。

```
$ 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>
```

- base64 エンコーディングでファイルを追加します。

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

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

3. エンコードされたテキストを YAML ファイルに追加します。

```

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

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

```

❶❷ **namespace - default** を 有効な namespace に置き換えます。

4. YAML ファイルを適用します。

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

4.6.1. キーの確認

キーを追加したら、キーが正しく設定されていることを確認する必要があります。

手順

1. 公開鍵シークレットが正しく設定されていることを確認します。

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

これにより、シリアル番号、発行者、サブジェクトなどを含む証明書が表示されます。

2. 秘密鍵シークレットが正しく設定されていることを確認します。

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

これにより、**-----BEGIN PRIVATE KEY-----** および **-----END PRIVATE KEY-----** 行で囲まれたキーが表示されます。

4.7. ビルド済みのドライバーコンテナに署名する

ハードウェアベンダーによって配布されたイメージや別の場所でビルドされたイメージなど、ビルド済みのイメージがある場合は、この手順を使用します。

次の YAML ファイルは、公開鍵と秘密鍵のペアを必要なキー名 (秘密鍵の場合は **key**、公開鍵の場合は **cert**) を持つシークレットとして追加します。次に、クラスターは **unsignedImage** イメージをプルダ

ウンし、これを開いて **filesToSign** に一覧表示されているカーネルモジュールに署名し、それらを再び追加し、作成されたイメージを **containerImage** としてプッシュします。

Kernel Module Management (KMM) は、セクターに一致するすべてのノードに署名済みの kmod をロードする DaemonSet をデプロイする必要があります。ドライバーコンテナは、MOK データベースに公開鍵があるすべてのノード、および署名を無視するセキュアブートが有効になっていないすべてのノードで正常に実行されるはずです。セキュアブートが有効になっているが、MOK データベースにそのキーがない場合は、ロードに失敗するはずです。

前提条件

- **keySecret** および **certSecret** シークレットが作成されている。

手順

1. YAML ファイルを適用します。

```
---
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** - ロードする kmod の名前。

4.8. MODULELOADER コンテナイメージのビルドと署名

ソースコードがあり、最初にイメージをビルドする必要がある場合は、この手順を使用します。

次の YAML ファイルは、リポジトリのソースコードを使用して新しいコンテナイメージをビルドします。生成されたイメージは一時的な名前でレジストリーに保存され、この一時的なイメージは **sign** セクションのパラメーターを使用して署名されます。

一時的なイメージ名は最終的なイメージ名に基づいており、**<containerImage>:<tag>-<namespace>_<module name>_kmm_unsigned** に設定されています。

たとえば、次の YAML ファイルを使用すると、Kernel Module Management (KMM) は、**example.org/repository/minimal-driver:final-default_example-module_kmm_unsigned** という名前のイメージをビルドし、署名されていない kmod を含むビルドを含み、レジストリーにプッシュします。次に、署名された kmod を含む **example.org/repository/minimal-driver:final** という名前の 2 番目のイメージを作成します。これは、**DaemonSet** オブジェクトによって読み込まれ、kmods をクラスターノードにデプロイするこの 2 つ目のイメージです。

署名後、一時イメージはレジストリーから安全に削除できます。必要に応じて再構築されます。

前提条件

- **keySecret** および **certSecret** シークレットが作成されている。

手順

1. YAML ファイルを適用します。

```
---
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 ❷
spec:
  moduleLoader:
    serviceAccountName: default ❸
    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: ❹
      name: repo-pull-secret
  selector: # top-level selector
    kubernetes.io/arch: amd64

```

❶ ❷ **namespace - default** を有効な namespace に置き換えます。

❸ **serviceAccountName**: デフォルトの **serviceAccountName** には、特権付きのモジュールを実行するために必要な権限がありません。サービスアカウントの作成については、このセクションのその他のリソースのサービスアカウントの作成を参照してください。

❹ **imageRepoSecret**: **DaemonSet** オブジェクトの **imagePullSecret** として使用され、ビルドおよび署名機能のプルおよびプッシュに使用されます。

関連情報

サービスアカウントの作成は、[サービスアカウントの作成](#) を参照してください。

4.9. デバッグとトラブルシューティング

ドライバーコンテナ内の kmod が署名されていないか、間違ったキーで署名されている場合、コンテナは **PostStartHookError** または **CrashLoopBackOff** ステータスに入る可能性があります。コンテナで **oc describe** コマンドを実行することで確認できます。このシナリオでは、次のメッセージが表示されます。

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

4.10. KMM ファームウェアのサポート

カーネルモジュールは、ファイルシステムからファームウェアファイルをロードする必要がある場合があります。KMM は、ModuleLoader イメージからノードのファイルシステムへのファームウェアファイルのコピーをサポートしています。

modprobe コマンドを実行してカーネルモジュールを挿入する前に、**.spec.moduleLoader.container.modprobe.firmwarePath** の内容がノードの **/var/lib/firmware** パスにコピーされます。

Pod の終了時に **modprobe -r** コマンドを実行してカーネルモジュールをアンロードする前に、すべてのファイルと空のディレクトリーがその場所から削除されます。

関連情報

- [ModuleLoader イメージの作成。](#)

4.10.1. ノードでのルックアップパスの設定

OpenShift Container Platform ノードでは、ファームウェアのデフォルトのルックアップパスのセットに **/var/lib/firmware** パスが含まれません。

手順

1. Machine Config Operator を使用して、**/var/lib/firmware** パスを含む **MachineConfig** カスタムリソース (CR) を作成します。

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

- ❶ 必要に応じてラベルを設定できます。単一ノードの OpenShift の場合は、**control-plane** または **master** オブジェクトのいずれかを使用します。

2. **MachineConfig** CR を適用すると、ノードが自動的に再起動されます。

関連情報

- [Machine Config Operator.](#)

4.10.2. ModuleLoader イメージのビルド

手順

- カーネルモジュール自体をビルドするだけでなく、バイナリーファームウェアをビルダーイメージに含めます。

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

```
# Copy the kmod, install modprobe, run depmod
```

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

4.10.3. モジュールリソースのチューニング

手順

- **Module** カスタムリソース (CR) で `.spec.moduleLoader.container.modprobe.firmwarePath` を設定します。

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

    firmwarePath: /firmware ❶
```

- ❶ オプション: `/firmware/*` をノード上の `/var/lib/firmware/` にコピーします。

4.11. KMM のトラブルシューティング

KMM インストール問題のトラブルシューティングを行う場合、ログを監視して、どの段階で問題が発生したかを判断できます。次に、その段階に関連する診断データを取得します。

4.11.1. must-gather ツールの使用

`oc adm must-gather` コマンドは、サポートバンドルを収集してデバッグ情報を Red Hat サポートに提供するための推奨される方法です。次のセクションで説明するように、適切な引数を指定してコマンドを実行し、特定の情報を収集します。

関連情報

- [must-gather ツールについて](#)

4.11.1.1. KMM のデータの収集

手順

1. KMM Operator コントローラーマネージャーのデータを収集します。
 - a. **MUST_GATHER_IMAGE** 変数を設定します。

```
$ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm kmm-
```



```
operator-controller-manager -ojsonpath='{.spec.template.spec.containers[?
(@.name=="manager")].env[?
(@.name=="RELATED_IMAGES_MUST_GATHER")].value}')
```



注記

KMM をカスタム namespace にインストールしている場合は、**-n <namespace>** スイッチを使用して namespace を指定します。

- b. **must-gather** ツールを実行します。

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

2. Operator ログを表示します。

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

例4.1 出力例

```
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"

```

4.11.1.2. KMM-Hub のデータ収集

手順

1. KMM Operator ハブコントローラーマネージャーのデータを収集します。
 - a. **MUST_GATHER_IMAGE** 変数を設定します。

```

$ export MUST_GATHER_IMAGE=$(oc get deployment -n openshift-kmm-hub kmm-
operator-hub-controller-manager -ojsonpath='{.spec.template.spec.containers[?
(@.name=="manager")].env[?
(@.name=="RELATED_IMAGES_MUST_GATHER")].value}')

```



注記

KMM をカスタム namespace にインストールしている場合は、**-n <namespace>** スイッチを使用して namespace を指定します。

- b. **must-gather** ツールを実行します。

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

2. Operator ログを表示します。

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

例4.2 出力例

```
10417 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
10417 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"
10417 11:34:12.376253      1 main.go:150] kmm-hub/setup "msg"="Adding controller"
"name"="ManagedClusterModule"
10417 11:34:12.376621      1 main.go:186] kmm-hub/setup "msg"="starting manager"
10417 11:34:12.377690      1 leaderelection.go:248] attempting to acquire leader lease
openshift-kmm-hub/kmm-hub.sigs.x-k8s.io...
10417 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"
10417 11:34:12.378222      1 internal.go:366] kmm-hub "msg"="Starting server" "addr"=
{"IP":"","Port":8081,"Zone":""} "kind"="health probe"
10417 11:34:12.395703      1 leaderelection.go:258] successfully acquired lease
openshift-kmm-hub/kmm-hub.sigs.x-k8s.io
10417 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"
10417 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"
10417 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"
10417 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"
10417 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"
10417 11:34:12.396543      1 controller.go:193] kmm-hub "msg"="Starting Controller"
"controller"="ManagedClusterModule" "controllerGroup"="hub.kmm.sigs.x-k8s.io"
"controllerKind"="ManagedClusterModule"
10417 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"
10417 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"
```