OpenShift Container Platform 4.10

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

Last Updated: 2023-10-17

OpenShift Container Platform 4.10 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

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

多くのアプリケーションには、カーネルモジュールまたはドライバーに依存する専用のハードウェアまたはソフトウェアが必要です。ドライバーコンテナーを使用して、Red Hat Enterprise Linux CoreOS(RHCOS) ノードで out-of-tree カーネルモジュールを読み込むことができます。クラスターのインストール時に out-of-tree(out oftree) ドライバーをデプロイするには、**kmods-via-containers** フレームワークを使用します。OpenShift Container Platform は、既存の OpenShift Container Platform クラスターでドライバーまたはカーネルモジュールを読み込むためのツールを複数提供します。

- Driver Toolkit は、OpenShift Container Platform の全リリースに同梱されているコンテナーイメージです。これには、ドライバーまたはカーネルモジュールのビルドに必要なカーネルパッケージとその他の共通の依存関係が含まれます。Driver Toolkit は、OpenShift Container Platform でドライバーコンテナーイメージビルドのベースイメージとして使用できます。

- Special Resource Operator(SRO) は、ドライバーコンテナーのビルドおよび管理をオーケストレーションし、既存の OpenShift または Kubernetes クラスターでカーネルモジュールおよびドライバーを読み込みます。

- Node Feature Discovery(NFD)Operator は、CPU 機能、カーネルバージョン、PCIe デバイスベンダー ID などのノードラベルを追加します。

第2章 ドライバーツールキット

Driver Toolkit および、ドライバーコンテナーのベースイメージとして使用して Kubernetes 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を説明します。

重要

Driver Toolkit はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

2.1. DRIVER TOOLKIT について

背景情報
Driver Toolkit は、ドライバーコンテナー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ーイメージです。Driver Toolkit イメージには、カーネルモジュールのビルドまたはインストール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ーで必要なツールが含まれます。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ーは、RHCOS などのコンテナー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ーイメージです。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ーは、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となる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**
- **binutilskabi-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 にインストールすることができていました。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ーを事前にビルドできます。

Driver Toolkit は、現在 OperatorHub のコミュニティー Operator として利用できる Special Resource Operator (SRO) によっても使用されます。SRO 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、SRO の **レシピ** を作成してドライバーコンテナー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。レシピには、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ーをビルドできます。または SRO で事前ビルドされたドライバーコンテナーをデプロイできます。

2.2. DRIVER TOOLKIT コンテナーイメージのプル

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ーイメージ 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 コンテナーイメージのプル

podman または OpenShift Container Platform で **driver-toolkit** イメージを **registry.redhat.io** からプルする手順は、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 driver-toolkit イメージは、**registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.10** のマイナーリリースバージョンでタグ付けされます。

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. 特定のリリースに対応する **driver-toolkit** のイメージ URL は、 **oc adm** コマンドを使用してリリースイメージから取得できます。

```
$ oc adm release info 4.10.0 --image-for=driver-toolkit
```

出力例

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

2. このイメージは、OpenShift Container Platform のインストールに必要なプルシークレットなどの有効なプルシークレットを使用してプルできます。

```
$ 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 **openssl**、**mokutil**、および **keyutils** が含まれます。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** が有効なシステムには読み込めません。

2.3.1. クラスターでの simple-kmod ドライバーコンテナーをビルドし、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ーイメージを保存する **ImageStream** と、コンテナーをビルドする **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:
    git:
      ref: "master"
      uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
    type: Git
    dockerfile: |
      FROM DRIVER_TOOLKIT_IMAGE

      WORKDIR /build/

      # Expecting kmod software version as an input to the build
      ARG KMODVER

      # Grab the software from upstream
      RUN git clone https://github.com/openshift-psap/simple-kmod.git
      WORKDIR simple-kmod

      # Build and install the module
      RUN make all       KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER} \
      && make install   KVER=$(rpm -q --qf "%{VERSION}-%{RELEASE}.%{ARCH}"  kernel-
core) KMODVER=${KMODVER}

      # Add the helper tools
      WORKDIR /root/kvc-simple-kmod
      ADD Makefile .
      ADD simple-kmod-lib.sh .
      ADD simple-kmod-wrapper.sh .
      ADD simple-kmod.conf .
      RUN mkdir -p /usr/lib/kvc/ \
      && mkdir -p /etc/kvc/ \
      && make install

      RUN systemctl enable kmods-via-containers@simple-kmod
  strategy:
    dockerStrategy:
```

```
    buildArgs:
      - name: KMODVER
        value: DEMO
  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ーイメージを **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ーコンテキストでドライバーコンテナー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ーを実行するための 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: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-containers@simple-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ー 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

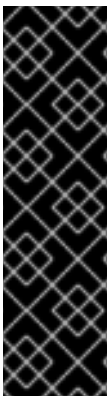
```
simple_procfs_kmod     16384  0
simple_kmod            16384  0
```

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 SPECIAL RESOURCE OPERATOR

Special Resource Operator(SRO) について説明し、これを使用して OpenShift Container Platform クラスターのノードでカーネルモジュールおよびデバイスドライバーを読み込むためのドライバーコンテナーをビルドおよび管理する方法を説明します。



重要

Special Resource Operator はテクノロジープレビュー機能としてのみご利用いただけます。テクノロジープレビュー機能は、Red Hat 製品のサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではないことがあります。Red Hat は実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

3.1. SPECIAL RESOURCE OPERATOR について

Special Resource Operator(SRO) を使用すると、既存の OpenShift Container Platform クラスターでカーネルモジュールとドライバーのデプロイメントの管理が容易になります。SRO は、単一のカーネルモジュールの構築およびロードなどの単純なケースや、ハードウェアアクセラレーター用のドライバー、デバイスプラグイン、およびモニタリングスタックのデプロイなどの複雑なケースに使用できます。

カーネルモジュールを読み込むため、SRO はドライバーコンテナーの使用を想定して設計されています。ドライバーコンテナーは、特にハードウェアドライバーをホストに提供するために純粋なコンテナーオペレーティングシステムで実行する場合など、クラウドネイティブ環境での使用が増えています。ドライバーコンテナーは、特定のカーネルの同梱ソフトウェアおよびハードウェア機能を超えて、カーネルスタックを拡張します。ドライバーコンテナーは、さまざまなコンテナー対応の Linux ディストリビューションで機能します。ドライバーコンテナーを使用すると、ホストオペレーティングシステムはクリーンな状態が保たれ、ライブラリーバージョンが異なる場合や、ホストのバイナリーが異なる場合も衝突しません。

3.2. SPECIAL RESOURCE 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 Special Resource Operator(SRO) をインストールできます。

3.2.1. CLI を使用した Special Resource Operator のインストール

クラスター管理者は、OpenShift CLI を使用して Special Resource Operator(SRO) 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。
- Node Feature Discovery(NFD)Operator がインストールされている。

手順

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

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

   Subscription CR の例

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

   b. 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 sro-sub.yaml
```

   c. **openshift-operators** プロジェクトに切り替えます。

```
$ oc project openshift-operators
```

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-7f4c5f5778-4lvvk               2/2     Running   0          89s
special-resource-controller-manager-6dbf7d4f6f-9kl8h   2/2     Running   0          81s
```

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 コンソールを使用した Special Resource 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 Special Resource Operator(SRO) をインストールできます。

前提条件

- Node Feature Discovery(NFD)Operator がインストールされている。

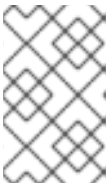
手順

1. OpenShift Container Platform Web コンソールにログインします。
2. Special Resource 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. 利用可能な Operator の一覧から **Special Resource Operator** を選択し、**Install** をクリックします。
   c. **Install Operator** ページで、**クラスターで特定の namespace** を選択し、直前のセクションで作成した namespace を選択してから **Install** をクリックします。

検証

Special Resource Operator が正常にインストールされていることを確認します。

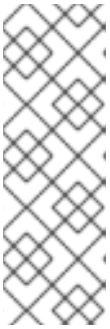
1. **Operators → Installed Operators** ページに移動します。
2. **Status** が **InstallSucceeded** の状態で、**Special Resource Operator** が **openshift-operators** プロジェクトに一覧表示されていることを確認します。



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。

3. 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。
   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-operators** プロジェクトで Pod のログを確認します。



注記

Node Feature Discovery(NFD)Operator は、Special Resource Operator(SRO) の依存関係です。SRO のインストール前に NFD Operator がインストールされていない場合には、Operator Lifecycle Manager で NFD Operator が自動インストールされます。ただし、必要な Node Feature Discovery オペランドは自動的にデプロイされません。Node Feature Discovery Operator ドキュメントでは、NFD Operator を使用して NFD をデプロイする方法の詳細が記載されています。

3.3. SPECIAL RESOURCE OPERATOR の使用

Special Resource Operator(SRO) は、ドライバーコンテナーのビルドおよびデプロイメント管理に使用されます。コンテナーのビルドおよびデプロイに必要なオブジェクトは Helm チャートに定義できます。

このセクションの例では、simple-kmod **SpecialResource** オブジェクトを使用して、Helm チャートを格納するために作成された **ConfigMap** オブジェクトをポイントしています。

3.3.1. 設定マップを使用した simple-kmod SpecialResource のビルドおよび実行

この例では、simple-kmod カーネルモジュールは、Special Resource Operator (SRO) がドライバーコンテナーを管理する方法を示しています。コンテナーは、設定マップに保存されている Helm チャートテンプレートで定義されます。

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。
- Node Feature Discovery(NFD)Operator がインストールされている。
- SRO をインストールされている。
- Helm CLI(**helm**) がインストールされている。

手順

1. simple-kmod **SpecialResource** オブジェクトを作成するには、イメージをビルドするイメージストリームおよびビルド設定を定義し、コンテナーを実行するサービスアカウント、ロール、ロールバインディング、およびデーモンセットを定義します。カーネルモジュールを読み込めるように、特権付きセキュリティーコンテキストでデーモンセットを実行するにはサービスアカウント、ロール、およびロールバインディングが必要です。

   a. **templates** ディレクトリーを作成して、このディレクトリーに移動します。

   ```
   $ mkdir -p chart/simple-kmod-0.0.1/templates
   ```

   ```
   $ cd chart/simple-kmod-0.0.1/templates
   ```

   b. イメージストリームおよびビルド設定の YAML テンプレートを **0000-buildconfig.yaml** として **templates** ディレクトリーに保存します。

   ```
   apiVersion: image.openshift.io/v1
   kind: ImageStream
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-
   {{.Values.groupName.driverContainer}} 1
     name: {{.Values.specialresource.metadata.name}}-
   {{.Values.groupName.driverContainer}} 2
   spec: {}
   ---
   apiVersion: build.openshift.io/v1
   kind: BuildConfig
   metadata:
     labels:
       app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
   3
   ```

```
  name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
④
  annotations:
    specialresource.openshift.io/wait: "true"
    specialresource.openshift.io/driver-container-vendor: simple-kmod
    specialresource.openshift.io/kernel-affine: "true"
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
  triggers:
    - type: "ConfigChange"
    - type: "ImageChange"
  source:
    git:
      ref: {{.Values.specialresource.spec.driverContainer.source.git.ref}}
      uri: {{.Values.specialresource.spec.driverContainer.source.git.uri}}
    type: Git
  strategy:
    dockerStrategy:
      dockerfilePath: Dockerfile.SRO
      buildArgs:
        - name: "IMAGE"
          value: {{ .Values.driverToolkitImage  }}
        {{- range $arg := .Values.buildArgs }}
        - name: {{ $arg.name }}
          value: {{ $arg.value }}
        {{- end }}
        - name: KVER
          value: {{ .Values.kernelFullVersion }}
  output:
    to:
      kind: ImageStreamTag
      name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}} ⑤
```

①②③④⑤ **{{.Values.specialresource.metadata.name}}** などのテンプレートは、**SpecialResource** CR のフィールドおよび **{{.Values.KernelFullVersion}}** などの Operator に認識される変数に基づいて SRO により入力されます。

c. RBAC リソースおよびデーモンセットの以下の YAML テンプレートを **1000-driver-container.yaml** として **templates** ディレクトリーに保存します。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
```

```
rules:
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  resourceNames:
  - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
subjects:
- kind: ServiceAccount
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  namespace: {{.Values.specialresource.spec.namespace}}
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  annotations:
    specialresource.openshift.io/wait: "true"
    specialresource.openshift.io/state: "driver-container"
    specialresource.openshift.io/driver-container-vendor: simple-kmod
    specialresource.openshift.io/kernel-affine: "true"
    specialresource.openshift.io/from-configmap: "true"
spec:
  updateStrategy:
    type: OnDelete
  selector:
    matchLabels:
      app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
  template:
    metadata:
      labels:
        app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
    spec:
      priorityClassName: system-node-critical
      serviceAccount: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
```

```
      serviceAccountName: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
      containers:
      - image: image-registry.openshift-image-
registry.svc:5000/{{.Values.specialresource.spec.namespace}}/{{.Values.specialresource.m
etadata.name}}-{{.Values.groupName.driverContainer}}:v{{.Values.kernelFullVersion}}
        name: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
        imagePullPolicy: Always
        command: ["/sbin/init"]
        lifecycle:
          preStop:
            exec:
              command: ["/bin/sh", "-c", "systemctl stop kmods-via-
containers@{{.Values.specialresource.metadata.name}}"]
        securityContext:
          privileged: true
      nodeSelector:
        node-role.kubernetes.io/worker: ""
        feature.node.kubernetes.io/kernel-version.full: "{{.Values.KernelFullVersion}}"
```

d. **chart/simple-kmod-0.0.1** ディレクトリーに移動します。

```
$ cd ..
```

e. チャートに関する以下の YAML を **Chart.yaml** として **chart/simple-kmod-0.0.1** ディレクトリーに保存します。

```
apiVersion: v2
name: simple-kmod
description: Simple kmod will deploy a simple kmod driver-container
icon: https://avatars.githubusercontent.com/u/55542927
type: application
version: 0.0.1
appVersion: 1.0.0
```

2. **chart** ディレクトリーから、**helm package** コマンドを使用してチャートを作成します。

```
$ helm package simple-kmod-0.0.1/
```

出力例

```
Successfully packaged chart and saved it to:
/data/<username>/git/<github_username>/special-resource-operator/yaml-for-
docs/chart/simple-kmod-0.0.1/simple-kmod-0.0.1.tgz
```

3. 設定マップを作成して、チャートファイルを保存します。

a. 設定マップファイルのディレクトリーを作成します。

```
$ mkdir cm
```

b. Helm チャートを **cm** ディレクトリーにコピーします。

```
$ cp simple-kmod-0.0.1.tgz cm/simple-kmod-0.0.1.tgz
```

c. Helm チャートが含まれる Helm リポジトリーを指定してインデックスファイルを作成します。

```
$ helm repo index cm --url=cm://simple-kmod/simple-kmod-chart
```

d. Helm チャートで定義されるオブジェクトの namespace を作成します。

```
$ oc create namespace simple-kmod
```

e. 設定マップオブジェクトを作成します。

```
$ oc create cm simple-kmod-chart --from-file=cm/index.yaml --from-file=cm/simple-kmod-0.0.1.tgz -n simple-kmod
```

4. 以下の **SpecialResource** マニフェストを使用して、設定マップで作成した Helm チャートにより simple-kmod オブジェクトをデプロイします。この YAML を **simple-kmod-configmap.yaml** として保存します。

```
apiVersion: sro.openshift.io/v1beta1
kind: SpecialResource
metadata:
  name: simple-kmod
spec:
  #debug: true (1)
  namespace: simple-kmod
  chart:
    name: simple-kmod
    version: 0.0.1
    repository:
      name: example
      url: cm://simple-kmod/simple-kmod-chart (2)
  set:
    kind: Values
    apiVersion: sro.openshift.io/v1beta1
    kmodNames: ["simple-kmod", "simple-procfs-kmod"]
    buildArgs:
    - name: "KMODVER"
      value: "SRO"
  driverContainer:
    source:
      git:
        ref: "master"
        uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
```

(1) オプション:**#debug: true** 行のコメントを解除して、チャートに YAML ファイルを Operator ログに完全に出力し、ログが作成され、適切にテンプレート化されていることを確認します。

(2) **spec.chart.repository.url** フィールドは SRO に対して設定マップでチャートを検索するように指示します。

5. コマンドラインで、**SpecialResource** ファイルを作成します。

```
$ oc create -f simple-kmod-configmap.yaml
```

注記

ノードから simple-kmod カーネルモジュールを削除する場合は、**oc delete** コマンドを使用して simple-kmod **SpecialResource** API オブジェクトを削除します。カーネルモジュールは、ドライバーコンテナー Pod が削除されるとアンロードされます。

検証

simple-kmod リソースは、オブジェクトマニフェストで指定された **simple-kmod** namespace にデプロイされます。しばらくすると、**simple-kmod** ドライバーコンテナーのビルド Pod の実行が開始されます。ビルドは数分後に完了し、ドライバーコンテナー Pod の実行が開始されます。

1. **oc get pods** コマンドを使用して、ビルド Pod のステータスを表示します。

```
$ oc get pods -n simple-kmod
```

出力例

```
NAME                                              READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-12813789169ac0ee-1-build     0/1     Completed   0          7m12s
simple-kmod-driver-container-12813789169ac0ee-mjsnh   1/1     Running     0          8m2s
simple-kmod-driver-container-12813789169ac0ee-qtkff   1/1     Running     0          8m2s
```

2. 上記の **oc get pods** コマンドから取得したビルド Pod 名と共に **oc logs** コマンドを使用して、simple-kmod ドライバーコンテナーイメージビルドのログを表示します。

```
$ oc logs pod/simple-kmod-driver-build-12813789169ac0ee-1-build -n simple-kmod
```

3. simple-kmod カーネルモジュールがロードされていることを確認するには、上記の **oc get pods** コマンドから返されたドライバーコンテナー Pod のいずれかで **lsmod** コマンドを実行します。

```
$ oc exec -n simple-kmod -it pod/simple-kmod-driver-container-12813789169ac0ee-mjsnh --
lsmod | grep simple
```

出力例

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

ヒント

sro_kind_completed_info SRO Prometheus メトリックから、デプロイされているさまざまなオブジェクトのステータスに関する情報が分かるので、SROCR インストールのトラブルシューティングに役立ちます。SRO には、環境の正常性監視に使用できる他のタイプのメトリックも含まれます。

3.4. PROMETHEUS SPECIAL RESOURCE OPERATOR メトリクス

Special Resource Operator (SRO) は、**メトリクス**サービスで、次の Prometheus メトリクスを公開します。

メトリクス名	説明
sro_used_nodes	SRO カスタムリソース (CR) によって作成された Pod を実行しているノードを返します。このメトリックは、 **DaemonSet**および**Deployment**オブジェクトでのみ使用できます。
sro_kind_completed_info	SRO CR の Helm チャートで定義されたオブジェクトの**kind**がクラスターに正常にアップロードされたか (値**1**)、そうでないか (値**0**) を表します。オブジェクトの例としては、**DaemonSet**、**Deployment**、**BuildConfig**などがあります。
sro_states_completed_info	SRO が CR の処理を正常に終了したか (値**1**)、または SRO が CR をまだ処理していないか (値**0**) を表します。
sro_managed_resources_total	状態に関係なく、クラスター内の SRO CR の数を返します。

3.5. 関連情報

- Special Resource Operator を使用する前にイメージレジストリー Operator の状態を復元する方法は、インストール時に削除されたイメージレジストリー を参照してください。
- NFD Operator のインストールの詳細については、Node Feature Discovery (NFD) Operator を参照してください。

第4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。

4.1. 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。

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

Node Feature Discovery (NFD) Operator は、NFD デー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。

4.2.1. 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。

4.2.2. 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。

4.3. NODE FEATURE DISCOVERY OPERATOR の使用

Node Feature Discovery (NFD) Operator は、**NodeFeatureDiscovery** CR を監視して Node-Feature-Discovery デー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。

4.3.1. 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:v4.10
    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-vgcz9                         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。

4.3.2. 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。

4.4. NODE FEATURE DISCOVERY OPERATOR の設定

4.4.1. 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-cpuid'
```

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 (1)
```

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、スタックス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

4.4.2. 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"]
    - pciId:
        class: ["0200"]
        vendor: ["8086"]
```

4.5. NFD トポロジーアップデータの使用

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。

4.5.1. 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
```

4.5.2. 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 コンテナーで、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、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
```

4.5.2.1. -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
```