



OpenShift Container Platform 4.9

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ついて確認します。

OpenShift Container Platform 4.9 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ついて確認します。

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Specialized_hardware_and_driver_enablement.ent file |.

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	4
第2章 ドライバツールキット	5
2.1. DRIVER TOOLKIT について	5
背景情報	5
目的	6
2.2. DRIVER TOOLKIT コンテナイメージのプル	6
2.2.1. registry.redhat.io からの Driver Toolkit コンテナイメージのプル	6
2.2.2. ペイロードでの Driver Toolkit イメージ URL の検索	6
2.3. DRIVER TOOLKIT の使用	7
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。	7
2.4. 関連情報	11
第3章 SPECIAL RESOURCE OPERATOR	12
3.1. SPECIAL RESOURCE OPERATOR について	12
3.2. SPECIAL RESOURCE OPERATOR のインストール	12
3.2.1. CLI を使用した Special Resource Operator のインストール	12
3.2.2. Web コンソールを使用した Special Resource Operator のインストール	14
3.3. SPECIAL RESOURCE OPERATOR の使用	15
3.3.1. SRO イメージからのテンプレートを使用した simple-kmod SpecialResource のビルドおよび実行	16
3.3.2. 設定マップを使用した simple-kmod SpecialResource のビルドおよび実行	17
3.4. 関連情報	23
第4章 NODE FEATURE DISCOVERY OPERATOR	24
4.1. NODE FEATURE DISCOVERY OPERATOR について	24
4.2. NODE FEATURE DISCOVERY OPERATOR のインストール	24
4.2.1. CLI を使用した NFD Operator のインストール	24
4.2.2. Web コンソールでの NFD Operator のインストール	26
4.3. NODE FEATURE DISCOVERY OPERATOR の使用	26
4.3.1. CLI を使用した NodeFeatureDiscovery インスタンスの作成	27
4.3.2. Web コンソールを使用した NodeFeatureDiscovery CR の作成	30
4.4. NODE FEATURE DISCOVERY OPERATOR の設定	30
4.4.1. コア	30
core.sleepInterval	30
core.sources	30
core.labelWhiteList	31
core.noPublish	31
core.klog	31
core.klog.addDirHeader	31
core.klog.alsologtostderr	31
core.klog.logBacktraceAt	32
core.klog.logDir	32
core.klog.logFile	32
core.klog.logFileMaxSize	32
core.klog.logtostderr	32
core.klog.skipHeaders	32
core.klog.skipLogHeaders	32
core.klog.stdderrthreshold	33
core.klog.v	33
core.klog.vmodule	33
4.4.2. ソース	33
sources.cpu.cpubid.attributeBlacklist	33

sources.cpu.cpuid.attributeWhitelist	33
sources.kernel.kconfigFile	33
sources.kernel.configOpts	34
sources.pci.deviceClassWhitelist	34
sources.pci.deviceLabelFields	34
sources.usb.deviceClassWhitelist	34
sources.usb.deviceLabelFields	35
sources.custom	35

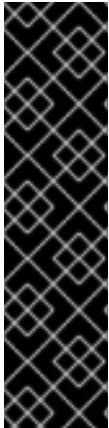
第1章 専用のハードウェアおよびドライバーの有効化

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 ドライバツールキット

Driver Toolkit および、ドライバーコンテナ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。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、<https://access.redhat.com/ja/support/offerings/techpreview/> 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、またはホストの **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を事前にビルドできます。

Driver Toolkit は、現在 OperatorHub のコミュニティ 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 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:v4.9** 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 サイトの「[Pull Secret](#)」ページから、OpenShift Container Platform のインストールに必要な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 4.9.0 --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** に設定します。
- 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。以下は例になります。

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

1. YAML は、**simple-kmod** ドライバ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:
    git:
      ref: "master"
      uri: "https://github.com/openshift-psap/kvc-simple-kmod.git"
    type: Git
  dockerfile: |
    FROM DRIVER_TOOLKIT_IMAGE

    WORKDIR /build/

    RUN yum -y install git make sudo gcc \
      && yum clean all \
      && rm -rf /var/cache/dnf

    # 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

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

- 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
```

- 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
```

- 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。
 - 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
  resourceName:
  - 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 **lsm** 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 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 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。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。

Red Hat のテクノロジープレビュー機能のサポート範囲についての詳細は、<https://access.redhat.com/ja/support/offerings/techpreview/> 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は、さまざまなコンテナ対応の 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. Special Resource Operator の namespace を作成します。
 - a. **openshift-special-resource-operator** 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 **sro-namespace.yaml** ファイルに保存します。

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-special-resource-operator
```

- b. 以下のコマンドを実行して namespace を作成します。

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

2. 直前の手順で作成した namespace に SRO をインストールします。
 - a. 以下の **OperatorGroup** CR を作成し、YAML を **sro-operatorgroup.yaml** ファイルに保存します。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  generateName: openshift-special-resource-operator-
  name: openshift-special-resource-operator
  namespace: openshift-special-resource-operator
spec:
  targetNamespaces:
    - openshift-special-resource-operator
```

- b. 以下のコマンドを実行して Operator グループを作成します。

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

- c. 以下の **oc get** コマンドを実行して、次の手順に必要な **channel** の値を取得します。

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

出力例

```
4.9
```

- d. 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-special-resource-operator
spec:
  channel: "4.9" ❶
  installPlanApproval: Automatic
  name: openshift-special-resource-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace

```

- ❶ channel の値を直前のコマンドの出力に置き換えます。

- e. 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
```

- f. **openshift-special-resource-operator** プロジェクトに切り替えます。

```
$ oc project openshift-special-resource-operator
```

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
special-resource-controller-manager-7bfb544d45-xx62r 2/2   Running 0      2m28s

```

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

- OpenShift Container Platform Web コンソールにログインします。
- Special Resource Operator に必要な namespace を作成します。
 - Administration** → **Namespaces** に移動し、**Create Namespace** をクリックします。
 - Name** フィールドに **openshift-special-resource-operator** を入力し、**Create** をクリックします。

3. 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-special-resource-operator** 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-special-resource-operator** 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せん。

本項の例では、simple-kmod カーネルモジュールを使用して、SRO でドライバーコンテナをビルドし、実行する方法を説明します。最初の例では、SRO イメージには、simple-kmod カーネルモジュールをデプロイするためのテンプレートなど、Helm チャートのローカルリポジトリが含まれます。こ

の場合、**SpecialResource** マニフェストを使用して、ドライバーコンテナをデプロイします。2つ目の例では、simple-kmod **SpecialResource** オブジェクトは、Helm チャートの保存用に作成された **ConfigMap** オブジェクトを参照します。

3.3.1. SRO イメージからのテンプレートを使用した simple-kmod SpecialResource のビルドおよび実行

SRO イメージには、simple-kmod カーネルモジュールをデプロイするためのテンプレートなど、Helm チャートのローカルリポジトリが含まれます。この例では、simple-kmodカーネルモジュールを使用して、SROが内部SROリポジトリで定義されているドライバーコンテナを管理する方法を示しています。

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** に設定します。
- 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。
- Special Resource Operator がインストールされている。

手順

1. SRO イメージのローカル Helm リポジトリを使用して simple-kmod をデプロイするには、以下の **SpecialResource** マニフェストを使用します。この YAML を **simple-kmod-local.yaml** として保存します。

```
apiVersion: sro.openshift.io/v1beta1
kind: SpecialResource
metadata:
  name: simple-kmod
spec:
  namespace: simple-kmod
  chart:
    name: simple-kmod
    version: 0.0.1
    repository:
      name: example
      url: file:///charts/example
  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"
```

2. **SpecialResource** を作成します。

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

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 Pod の実行が開始されます。

3. **oc get pods** コマンドを使用して 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

4. **simple-kmod** ドライバーコンテナイメージビルドのログを表示するには、**oc logs** コマンドを上記で取得したビルド Pod 名と共に使用します。

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

5. **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 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_proofs_kmod    16384 0
simple_kmod           16384 0
```

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 Pod が削除されるとアンロードされます。

3.3.2. 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を使用して、SROが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ているドライバーコンテナを管理する方法を示しています。



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** に設定します。
- 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。
- Special Resource Operator が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むように、特権付きセキュリティ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}} ❶
    name: {{.Values.specialresource.metadata.name}}-
    {{.Values.groupName.driverContainer}} ❷
spec: {}
---
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}}
    name: {{.Values.specialresource.metadata.name}}-{{.Values.groupName.driverBuild}} ❸
    annotations:
      specialresource.openshift.io/wait: "true"
      specialresource.openshift.io/driver-container-vendor: simple-kmod
      specialresource.openshift.io/kernel-affine: "true" ❹
```

```

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

```

- 1 2 3 4 5 **{{.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:
      # Mark this pod as a critical add-on; when enabled, the critical add-on scheduler
      # reserves resources for critical add-on pods so that they can be rescheduled after
      # a failure. This annotation works in tandem with the toleration below.
    annotations:
      scheduler.alpha.kubernetes.io/critical-pod: ""
    labels:
      app: {{.Values.specialresource.metadata.name}}-
{{.Values.groupName.driverContainer}}
    spec:
      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
```

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 Pod の実行が開始されます。

6. **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-qtckff  1/1 Running 0      8m2s
```

7. 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
```

8. 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 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
```



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 Pod が削除されるとアンロードされます。

3.4. 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**) をインストールすること。
- **cluster-admin** 権限を持つユーザーとしてログインすること。

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. 以下のコマンドを実行して、次の手順に必要な **channel** の値を取得します。

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

出力例

```
4.9
```

- d. 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: "4.9"
  installPlanApproval: Automatic
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace

```

- e. 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
```

- f. **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。



注記

先のセクションで説明されているように **Namespace** を作成することが推奨されます。

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、**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

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** を選択することができます。

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**) をインストールすること。
- **cluster-admin** 権限を持つユーザーとしてログインすること。
- NFD Operator をインストールすること。

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
  operand:
    namespace: openshift-nfd
    image: registry.redhat.io/openshift4/ose-node-feature-discovery:v4.9
    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"
#      - "vendor"
#      - "device"
#      - "subsystem_vendor"
#      - "subsystem_device"
#  usb:
#    deviceClassWhitelist:
#      - "0e"
#      - "ef"
#      - "fe"
```



```

#   - "ff"
#   deviceLabelFields:
#   - "class"
#   - "vendor"
#   - "device"
#   custom:
#   - name: "my.kernel.feature"
#     matchOn:
#       - loadedKMod: ["example_kmod1", "example_kmod2"]
#   - name: "my.pci.feature"
#     matchOn:
#       - pcild:
#           class: ["0200"]
#           vendor: ["15b3"]
#           device: ["1014", "1017"]
#       - pcild :
#           vendor: ["8086"]
#           device: ["1000", "1100"]
#   - name: "my.usb.feature"
#     matchOn:
#       - usbld:
#           class: ["ff"]
#           vendor: ["03e7"]
#           device: ["2485"]
#       - usbld:
#           class: ["fe"]
#           vendor: ["1a6e"]
#           device: ["089a"]
#   - name: "my.combined.feature"
#     matchOn:
#       - pcild:
#           vendor: ["15b3"]
#           device: ["1014", "1017"]
#       loadedKMod : ["vendor_kmod1", "vendor_kmod2"]
customConfig:
  configData: |
    #   - name: "more.kernel.features"
    #   matchOn:
    #     - loadedKMod: ["example_kmod3"]
    #   - name: "more.features.by.nodename"
    #     value: customValue
    #   matchOn:
    #     - nodename: ["special-.*-node-.*"]

```

2. 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せん。

使用例

```
core:
  sleepInterval: 60s ❶
```

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。ラベルのプレフィックスまたは 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せん。

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 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プレフィックスを使用しません。

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作成する時に使用する 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 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"]
```