OpenShift Container Platform 4.9

특수 하드웨어 및 드라이버 활성화

OpenShift Container Platform의 하드웨어 활성화 관련 정보

Last Updated: 2023-04-08

OpenShift Container Platform 4.9 특수 하드웨어 및 드라이버 활성화

OpenShift Container Platform의 하드웨어 활성화 관련 정보

법적 공지

초록

이 문서에서는 OpenShift Container Platform의 하드웨어 활성화에 대한 개요를 설명합니다.

차례

1장. 특수 하드웨어 및 드라이버 활성화 정보

많은 애플리케이션에는 커널 모듈 또는 드라이버에 따라 달라지는 특수 하드웨어 또는 소프트웨어가 필요합니다. 드라이버 컨테이너를 사용하여 RHCOS(Red Hat Enterprise Linux CoreOS) 노드에서 트리 외부 (out-of-tree) 커널 모듈을 로드할 수 있습니다. 클러스터를 설치하는 동안 트리 외부 (out-of-tree) 드라이버를 배포하려면 **kmods-via-containers** 프레임워크를 사용합니다. 기존 OpenShift Container Platform 클러스터에서 드라이버 또는 커널 모듈을 로드하기 위해 OpenShift Container Platform에서는 다음과 같은 몇 가지 툴을 제공합니다.

- 드라이버 툴킷은 모든 OpenShift Container Platform 릴리스의 일부인 컨테이너 이미지입니다. 드라이버 또는 커널 모듈을 빌드하는 데 필요한 커널 패키지 및 기타 공통 종속성이 포함되어 있습니다. 드라이버 툴킷은 OpenShift Container Platform에서 드라이버 컨테이너 이미지 빌드의 기본 이미지로 사용할 수 있습니다.

- SRO(Special Resource Operator)는 기존 OpenShift 또는 Kubernetes 클러스터에서 커널 모듈과 드라이버를 로드하도록 드라이버 컨테이너의 빌드 및 관리를 오케스트레이션합니다.

- NFD(Node Feature Discovery) Operator는 CPU 기능, 커널 버전, PCIe 장치 벤더 ID 등에 대한 노드 레이블을 추가합니다.

2장. 드라이버 툴킷

드라이버 툴킷(Driver Toolkit)과 이 툴킷을 Kubernetes에서 특수 소프트웨어 및 하드웨어 장치를 활성화하기 위한 드라이버 컨테이너의 기본 이미지로 사용하는 방법에 대해 알아 보십시오.

중요

드라이버 툴킷(Driver Toolkit)은 기술 프리뷰 기능으로만 사용할 수 있습니다. 기술 프리뷰 기능은 Red Hat 프로덕션 서비스 수준 계약(SLA)에서 지원되지 않으며 기능적으로 완전하지 않을 수 있습니다. 따라서 프로덕션 환경에서 사용하는 것은 권장하지 않습니다. 이러한 기능을 사용하면 향후 제품 기능을 조기에 이용할 수 있어 개발 과정에서 고객이 기능을 테스트하고 피드백을 제공할 수 있습니다.

Red Hat 기술 프리뷰 기능의 지원 범위에 대한 자세한 내용은 기술 프리뷰 기능 지원 범위를 참조하십시오.

2.1. 드라이버 툴킷 정보

배경

Driver Toolkit은 드라이버 컨테이너를 빌드할 수 있는 기본 이미지로 사용되는 OpenShift Container Platform 페이로드의 컨테이너 이미지입니다. Driver Toolkit 이미지에는 커널 모듈을 빌드하거나 설치하는 데 일반적으로 필요한 커널 패키지와 드라이버 컨테이너에 필요한 몇 가지 툴이 포함되어 있습니다. 이러한 패키지의 버전은 해당 OpenShift Container Platform 릴리스의 RHCOS(Red Hat Enterprise Linux CoreOS) 노드에서 실행되는 커널 버전과 동일합니다.

드라이버 컨테이너는 RHCOS와 같은 컨테이너 운영 체제에서 트리 외부 커널 모듈 및 드라이버를 빌드하고 배포하는 데 사용되는 컨테이너 이미지입니다. 커널 모듈과 드라이버는 운영 체제 커널에서 높은 수준의 권한으로 실행되는 소프트웨어 라이브러리입니다. 커널 기능을 확장하거나 새 장치를 제어하는 데 필요한 하드웨어별 코드를 제공합니다. 예를 들어 Field Programmable Gate Arrays(예: Field Programmable Gate Arrays) 또는 GPU와 같은 하드웨어 장치, Lustre 병렬 파일 시스템과 같은 소프트웨어 정의 스토리지(SDS) 솔루션은 클라이언트 시스템에 커널 모듈이 필요합니다. 드라이버 컨테이너는 Kubernetes에서 이러한 기술을 활성화하는 데 사용되는 소프트웨어 스택의 첫 번째 계층입니다.

Driver Toolkit의 커널 패키지 목록에는 다음과 같은 종속성이 포함되어 있습니다.

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

또한 Driver Toolkit에는 해당 실시간 커널 패키지도 포함되어 있습니다.

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

또한 Driver Toolkit에는 다음을 포함하여 커널 모듈을 빌드하고 설치하는 데 일반적으로 필요한 여러 도구가 있습니다.

- **elfutils-libelf-devel**
- **kmod**
- **binutilskabi-dw**
- **kernel-abi-whitelists**
- 위의 종속 항목

목적
Driver Toolkit이 출시하기 전에는 권한이 부여된 빌드를 사용하거나 호스트 **machine-os-content**의 커널 RPM에서 설치하여 OpenShift Container Platform의 Pod에 커널 패키지를 설치할 수 있었습니다. Driver Toolkit은 인타이틀먼트 단계를 제거하여 프로세스를 간소화하고 Pod에서 machine-os-content에 액세스하는 권한 있는 작업을 피할 수 있습니다. Driver Toolkit은 사전 릴리스된 OpenShift Container Platform 버전에 액세스할 수 있는 파트너가 향후 OpenShift Container Platform 릴리스를 위한 하드웨어 장치용 드라이버 컨테이너를 사전 구축하는 데 사용할 수도 있습니다.

Driver Toolkit은 현재 OperatorHub에서 커뮤니티 Operator로 사용할 수 있는 SAR(Special Resource Operator)에서도 사용됩니다. SRO는 외부 및 타사 커널 드라이버와 기본 운영 체제에 대한 지원 소프트웨어를 지원합니다. SRO에 대한 *레시피*를 생성하여 드라이버 컨테이너를 빌드하고 배포할 수 있으며 장치 플러그인 또는 메트릭과 같은 지원 소프트웨어를 생성할 수 있습니다. 레시피에는 Driver Toolkit을 기반으로 드라이버 컨테이너를 빌드하는 빌드 구성이 포함될 수 있으며, SRO는 사전 빌드된 드라이버 컨테이너를 배포할 수 있습니다.

2.2. DRIVER TOOLKIT 컨테이너 이미지 가져오기

driver-toolkit 이미지는 Red Hat Ecosystem Catalog의 컨테이너 이미지 섹션과 OpenShift Container Platform 릴리스 페이로드에서 사용할 수 있습니다. OpenShift Container Platform의 최신 마이너 릴리스에 해당하는 이미지에는 카탈로그의 버전 번호로 태그가 지정됩니다. 특정 릴리스의 이미지 URL은 **oc adm** CLI 명령을 사용하여 찾을 수 있습니다.

2.2.1. registry.redhat.io에서 Driver Toolkit 컨테이너 이미지 가져오기

registry.redhat.io에서 podman 또는 OpenShift Container Platform을 사용하여 **driver-toolkit** 이미지를 가져오는 방법은 Red Hat Ecosystem Catalog 에서 확인할 수 있습니다. 최신 마이너 릴리스의 driver-toolkit 이미지는 **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.9**의 마이너 릴리스 버전에 태그가 지정됩니다.

2.2.2. 페이로드에서 Driver Toolkit 이미지 URL 검색

사전 요구 사항

- Red Hat OpenShift Cluster Manager에서 이미지 풀 시크릿 을 가져왔습니다.
- OpenShift CLI(**oc**)를 설치합니다.

절차

1. 특정 릴리스에 해당하는 **driver-toolkit**의 이미지 URL은 **oc adm** 명령을 사용하여 릴리스 이미지에서 얻을 수 있습니다.

```
$ oc adm release info 4.9.0 --image-for=driver-toolkit
```

출력 예

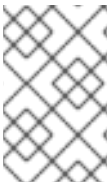
```
quay.io/openshift-release-dev/ocp-v4.0-art-
dev@sha256:0fd84aee79606178b6561ac71f8540f404d518ae5deff45f6d6ac8f02636c7f4
```

2. 이 이미지는 OpenShift Container Platform을 설치하는 데 필요한 pull secret과 같은 유효한 풀 시크릿을 사용하여 가져올 수 있습니다.

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

2.3. DRIVER TOOLKIT 사용

예를 들어 Driver Toolkit은 simple-kmod라는 매우 간단한 커널 모듈을 빌드하기 위한 기본 이미지로 사용할 수 있습니다.



참고

Driver Toolkit에는 커널 모듈에 서명하는 데 필요한 필수 종속성인 **openssl**, **mokutil** 및 **keyutils**가 포함되어 있습니다. 그러나 이 예에서는 simple-kmod 커널 모듈이 서명되지 않았으므로 **Secure Boot**가 활성화된 시스템에서 로드할 수 없습니다.

2.3.1. 클러스터에서 simple-kmod 드라이버 컨테이너를 빌드하고 실행합니다.

사전 요구 사항

- 실행 중인 OpenShift Container Platform 클러스터가 있어야 합니다.
- Image Registry Operator 상태를 클러스터의 **Managed**로 설정합니다.
- OpenShift CLI(**oc**)를 설치합니다.
- **cluster-admin** 권한이 있는 사용자로 OpenShift CLI에 로그인했습니다.

절차

네임스페이스를 생성합니다. 예를 들면 다음과 같습니다.

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

1. YAML은 **simple-kmod** 드라이버 컨테이너 이미지를 저장하기 위한 **ImageStream**과 컨테이너 빌드를 위한 **BuildConfig**를 정의합니다. 이 YAML을 **0000-buildconfig.yaml.template**로 저장합니다.

```
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. "DRIVER_TOOLKIT_IMAGE" 대신 실행 중인 OpenShift Container Platform 버전에 대해 올바른 드라이버 툴킷 이미지를 다음 명령으로 대체하십시오.

```
$ 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. 이미지 스트림 및 빌드 구성을 만듭니다.

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

4. 빌더 Pod가 성공적으로 완료되면 드라이버 컨테이너 이미지를 **DaemonSet**으로 배포합니다.

   a. 호스트에서 커널 모듈을 로드하려면 드라이버 컨테이너를 권한 있는 보안 컨텍스트로 실행해야 합니다. 다음 YAML 파일에는 RBAC 규칙과 드라이버 컨테이너 실행을 위한 **DaemonSet**이 포함되어 있습니다. 이 YAML을 **1000-drivercontainer.yaml**로 저장합니다.

```
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 규칙 및 데몬 세트를 생성합니다.

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

5. 작업자 노드에서 pod가 실행된 후 **lsmod**가 있는 호스트 시스템에 **simple_kmod** 커널 모듈이 제대로 로드되었는지 확인합니다.

a. pod가 실행 중인지 확인합니다.

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

출력 예

```
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. 드라이버 컨테이너 Pod에서 **lsmod** 명령을 실행합니다.

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

출력 예

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

2.4. 추가 리소스

- 클러스터의 레지스트리 스토리지 구성에 대한 자세한 내용은 OpenShift Container Platform의 이미지 레지스트리 Operator를 참조하십시오.

3장. SRO(SPECIAL RESOURCE OPERATOR)

SRO(Special Resource Operator)에 대해 알아보고 이를 사용하여 OpenShift Container Platform 클러스터의 노드에서 커널 모듈 및 장치 드라이버를 로드하기 위한 드라이버 컨테이너를 빌드 및 관리하는 방법에 대해 알아봅니다.

중요

SRO(Special Resource Operator)는 기술 프리뷰 기능입니다. 기술 프리뷰 기능은 Red Hat 프로덕션 서비스 수준 계약(SLA)에서 지원되지 않으며 기능적으로 완전하지 않을 수 있습니다. 따라서 프로덕션 환경에서 사용하는 것은 권장하지 않습니다. 이러한 기능을 사용하면 향후 제품 기능을 조기에 이용할 수 있어 개발 과정에서 고객이 기능을 테스트하고 피드백을 제공할 수 있습니다.

Red Hat 기술 프리뷰 기능의 지원 범위에 대한 자세한 내용은 기술 프리뷰 기능 지원 범위를 참조하십시오.

3.1. SRO(SPECIAL RESOURCE OPERATOR) 정보

SRO(Special Resource Operator)를 사용하면 기존 OpenShift Container Platform 클러스터에서 커널 모듈 및 드라이버 배포를 관리할 수 있습니다. SRO는 단일 커널 모듈을 빌드하고 로드하거나 드라이버, 장치 플러그인 및 모니터링 스택을 하드웨어 가속기를 위한 배포만큼 간단하게 사용할 수 있습니다.

커널 모듈을 로드하기 위해 SRO는 드라이버 컨테이너 사용을 중심으로 설계되었습니다. 드라이버 컨테이너는 특히 순수 컨테이너 운영 체제에서 실행될 때 호스트에 하드웨어 드라이버를 제공하기 위해 클라우드 네이티브 환경에서 점점 더 많이 사용되고 있습니다. 드라이버 컨테이너는 기본적으로 제공되는 소프트웨어 및 특정 커널의 하드웨어 기능을 넘어 커널 스택을 확장합니다. 드라이버 컨테이너는 컨테이너 가능한 다양한 Linux 배포판에서 작동합니다. 드라이버 컨테이너를 사용하면 호스트 운영 체제가 깔끔하게 유지되며 호스트의 여러 라이브러리 버전이나 바이너리 간에 충돌이 발생하지 않습니다.

3.2. SPECIAL RESOURCE OPERATOR 설치

클러스터 관리자는 OpenShift CLI 또는 웹 콘솔을 사용하여 SRO(Special Resource Operator)를 설치할 수 있습니다.

3.2.1. CLI를 사용하여 Special Resource Operator 설치

클러스터 관리자는 OpenShift CLI를 사용하여 SRO(Special Resource Operator)를 설치할 수 있습니다.

사전 요구 사항

- 실행 중인 OpenShift Container Platform 클러스터가 있어야 합니다.
- OpenShift CLI(**oc**)를 설치합니다.
- **cluster-admin** 권한이 있는 사용자로 OpenShift CLI에 로그인했습니다.
- NFD (Node Feature Discovery) Operator를 설치했습니다.

절차

1. Special Resource Operator의 네임스페이스를 생성합니다.

a. **openshift-special-resource-operator** 네임스페이스를 정의하는 다음 **Namespace** CR(사용자 정의 리소스)을 생성한 다음 YAML을 **sro-namespace.yaml** 파일에 저장합니다.

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

b. 다음 명령을 실행하여 네임스페이스를 생성합니다.

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

2. 이전 단계에서 생성한 네임스페이스에 SRO를 설치합니다.

a. 다음 **OperatorGroup** CR을 생성하고 YAML을 **sro-operatorgroup.yaml** 파일에 저장합니다.

```
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. 다음 명령을 실행하여 operator 그룹을 생성합니다.

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

c. 다음 **Subscription** CR을 생성하고 YAML을 **sro-sub.yaml** 파일에 저장합니다.

서브스크립션 CR의 예

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

d. 다음 명령을 실행하여 서브스크립션 오브젝트를 생성합니다.

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

e. **openshift-special-resource-operator** 프로젝트로 전환합니다.

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

검증

- Operator 배포가 완료되었는지 확인하려면 다음을 실행합니다.

```
$ oc get pods
```

출력 예

```
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
```

성공적인 배포에는 **Running** 상태가 표시됩니다.

3.2.2. 웹 콘솔을 사용하여 Special Resource Operator 설치

클러스터 관리자는 OpenShift Container Platform 웹 콘솔을 사용하여 SRO(Special Resource Operator)를 설치할 수 있습니다.

사전 요구 사항

- NFD (Node Feature Discovery) Operator를 설치했습니다.

절차

1. OpenShift Container Platform 웹 콘솔에 로그인합니다.
2. Special Resource Operator에 필요한 네임스페이스를 생성합니다.
   a. **관리 → 네임스페이스**로 이동하여 **네임스페이스 생성**을 클릭합니다.
   b. **이름** 필드에 **openshift-special-resource-operator**를 입력하고 **생성**을 클릭합니다.
3. Special Resource Operator를 설치합니다.
   a. OpenShift Container Platform 웹 콘솔에서 **Operator → OperatorHub**를 클릭합니다.
   b. 사용 가능한 Operator 목록에서 **Special Resource Operator**를 선택한 다음 **설치**를 클릭합니다.
   c. **Operator 설치** 페이지에서 **클러스터에서 특정 네임스페이스**를 선택하고 이전 섹션에서 생성된 네임스페이스를 선택한 다음 **설치**를 클릭합니다.

검증

Special Resource Operator가 성공적으로 설치되었는지 확인하려면 다음을 수행합니다.

1. **Operator → 설치된 Operator** 페이지로 이동합니다.
2. **Special Resource Operator**가 **openshift-special-resource-operator** 프로젝트에 **InstallSucceeded 상태**로 나열되어 있는지 확인합니다.

참고

설치하는 동안 Operator는 **실패** 상태를 표시할 수 있습니다. 나중에 **InstallSucceeded** 메시지와 함께 설치에 성공하면 이 **실패** 메시지를 무시할 수 있습니다.

3. Operator가 설치된 것으로 나타나지 않으면 다음과 같이 추가 문제 해결을 수행합니다.

   a. **Operator → 설치된 Operator** 페이지로 이동하고 **Operator 서브스크립션** 및 **설치 계획** 탭의 **상태**에 장애나 오류가 있는지 검사합니다.

   b. **워크로드 → Pod** 페이지로 이동하여 **openshift-special-resource-operator** 프로젝트에서 Pod 로그를 확인합니다.

참고

NFD(Node Feature Discovery) Operator는 SRO(Special Resource Operator)의 종속성입니다. SRO를 설치하기 전에 NFD Operator가 설치되지 않은 경우 Operator Lifecycle Manager가 NFD Operator를 자동으로 설치합니다. 그러나 필요한 Node Feature Discovery 피연산자가 자동으로 배포되지 않습니다. Node Feature Discovery Operator 설명서에서는 NFD Operator를 사용하여 NFD를 배포하는 방법에 대한 세부 정보를 제공합니다.

3.3. SPECIAL RESOURCE OPERATOR 사용

SRO(Special Resource Operator)는 드라이버 컨테이너의 빌드 및 배포를 관리하는 데 사용됩니다. 컨테이너를 빌드하고 배포하는 데 필요한 오브젝트는 Helm 차트에 정의할 수 있습니다.

이 섹션의 예제에서는 simple-kmod **SpecialResource** 오브젝트를 사용하여 Helm 차트를 저장하기 위해 생성된 **ConfigMap** 오브젝트를 가리킵니다.

3.3.1. 구성 맵을 사용하여 simple-kmod SpecialResource 구축 및 실행

이 예에서 simple-kmod 커널 모듈은 SRO가 구성 맵에 저장된 Helm 차트 템플릿에 정의된 드라이버 컨테이너를 관리하는 방법을 보여줍니다.

사전 요구 사항

- 실행 중인 OpenShift Container Platform 클러스터가 있어야 합니다.
- Image Registry Operator 상태를 클러스터의 **Managed**로 설정합니다.
- OpenShift CLI(**oc**)를 설치합니다.
- **cluster-admin** 권한이 있는 사용자로 OpenShift CLI에 로그인했습니다.
- NFD (Node Feature Discovery) Operator를 설치했습니다.
- Special Resource Operator를 설치했습니다.
- Helm CLI(**helm**)가 설치되어 있어야 합니다.

절차

1. simple-kmod **SpecialResource** 오브젝트를 생성하려면 이미지 스트림과 빌드 구성을 정의하여

이미지를 빌드하고, 컨테이너를 실행하도록 서비스 계정, 역할, 역할 바인딩 및 데몬 세트를 정의합니다. 커널 모듈을 로드할 수 있도록 권한 있는 보안 컨텍스트로 데몬 세트를 실행하려면 서비스 계정, 역할 및 역할 바인딩이 필요합니다.

a. **templates** 디렉터리를 생성하고 이 디렉터리로 변경합니다.

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

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

b. 이미지 스트림 및 빌드 구성에 대한 이 YAML 템플릿을 **templates** 디렉터리에 **0000-buildconfig.yaml**로 저장합니다.

```
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)

(1)(2)(3)(4)(5) **{{.Values.specialresource.metadata.name}}**과 같은 템플릿은 **SpecialResource** CR의 필드와 **{{.Values.KernelFullVersion}}**과 같은 Operator에 알려진 변수를 기반으로 SRO에 의해 채워집니다.

c. **templates** 디렉터리에 설정된 RBAC 리소스 및 데몬에 대한 다음 YAML 템플릿을 **1000-driver-container.yaml**로 저장합니다.

```
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** 디렉터리로 변경합니다.

```
$ cd ..
```

e. 차트에 대한 다음 YAML을 **chart/simple-kmod-0.0.1** 디렉터리에 **Chart.yaml**로 저장합니다.

```
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** 디렉토리에서 **hhelm package** 명령을 사용하여 차트를 생성합니다.

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

출력 예

```
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. 차트 파일을 저장할 구성 맵을 생성합니다.

   a. 구성 맵 파일의 디렉터리를 생성합니다.

```
$ mkdir cm
```

   b. Helm 차트를 **cm** 디렉터리에 복사합니다.

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

   c. Helm 차트가 포함된 Helm 리포지터리를 지정하는 인덱스 파일을 생성합니다.

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

   d. Helm 차트에 정의된 오브젝트의 네임스페이스를 생성합니다.

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

   e. 구성 맵 오브젝트를 생성합니다.

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

4. 구성 맵에서 생성한 Helm 차트를 사용하여 simple-kmod 오브젝트를 배포하려면 다음 **SpecialResource** 매니페스트를 사용합니다. 이 YAML을 **simple-kmod-configmap.yaml**로 저장합니다.

```
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 선택 사항: **#debug: true** 행의 주석을 제거하여 Operator 로그에서 차트에 YAML 파일을 출력하고 로그가 생성되고 올바르게 템플릿되었는지 확인합니다.

2 **spec.chart.repository.url** 필드는 SRO에 구성 맵의 차트를 찾도록 지시합니다.

5. 명령줄에서 **SpecialResource** 파일을 만듭니다.

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

simple-kmod 리소스는 오브젝트 매니페스트에 지정된 대로 **simple-kmod** 네임스페이스에 배포됩니다. 잠시 후 **simple-kmod** 드라이버 컨테이너의 빌드 Pod가 실행되기 시작합니다. 몇 분 후에 빌드가 완료되면 드라이버 컨테이너 pod가 실행됩니다.

6. **oc get pods** 명령을 사용하여 빌드 Pod의 상태를 표시합니다.

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

출력 예

```
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
```

7. **oc logs** 명령과 위의 **oc get pods** 명령에서 얻은 빌드 Pod 이름을 사용하여 simple-kmod 드라이버 컨테이너 이미지 빌드의 로그를 표시합니다.

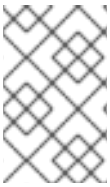
```
$ oc logs pod/simple-kmod-driver-build-12813789169ac0ee-1-build -n simple-kmod
```

8. simple-kmod 커널 모듈이 로드되었는지 확인하려면 위의 **oc get pods** 명령에서 반환된 드라이버 컨테이너 Pod 중 하나에서 **lsmod** 명령을 실행합니다.

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

출력 예

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



참고

노드에서 simple-kmod 커널 모듈을 제거하려면 **oc delete** 명령을 사용하여 simple-kmod **SpecialResource** API 오브젝트를 삭제합니다. 드라이버 컨테이너 Pod가 삭제되면 커널 모듈이 언로드됩니다.

3.4. 추가 리소스

- 특수 리소스 Operator를 사용하기 전에 이미지 레지스트리 Operator 상태를 복원하는 방법에 대한 자세한 내용은 설치 중에 제거된 이미지 레지스트리 를 참조하십시오.
- NFD Operator 설치에 대한 자세한 내용은 NFD(Node Feature Discovery) Operator 를 참조하십시오.

4장. NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery (NFD) Operator 및 이를 사용하여 하드웨어 기능과 시스템 구성을 감지하기 위한 Kubernetes 애드온 기능인 NFD(Node Feature Discovery)을 오케스트레이션하여 노드 수준 정보를 공개하는 방법을 설명합니다.

4.1. NODE FEATURE DISCOVERY OPERATOR 정보

NFD(노드 기능 검색 Operator)는 하드웨어 관련 정보로 노드에 레이블을 지정하여 OpenShift Container Platform 클러스터에서 하드웨어 기능 및 구성의 탐지를 관리합니다. NFD는 PCI 카드, 커널, 운영 체제 버전과 같은 노드별 속성을 사용하여 호스트에 레이블을 지정합니다.

NFD Operator는 "Node Feature Discovery"을 검색하여 Operator Hub에서 확인할 수 있습니다.

4.2. NODE FEATURE DISCOVERY OPERATOR 설치

NFD(Node Feature Discovery) Operator는 NFD 데몬 세트를 실행하는 데 필요한 모든 리소스를 오케스트레이션합니다. 클러스터 관리자는 OpenShift Container Platform CLI 또는 웹 콘솔을 사용하여 NFD Operator를 설치할 수 있습니다.

4.2.1. CLI를 사용하여 NFD Operator 설치

클러스터 관리자는 CLI를 사용하여 NFD Operator를 설치할 수 있습니다.

사전 요구 사항

- OpenShift Container Platform 클러스터
- OpenShift CLI(**oc**)를 설치합니다.
- **cluster-admin** 권한이 있는 사용자로 로그인합니다.

절차

1. NFD Operator의 네임스페이스를 생성합니다.
   a. **openshift-nfd** 네임스페이스를 정의하는 다음 **Namespace** CR(사용자 정의 리소스)을 생성하고 **nfd-namespace.yaml** 파일에 YAML을 저장합니다.

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

   b. 다음 명령을 실행하여 네임스페이스를 생성합니다.

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

2. 다음 오브젝트를 생성하여 이전 단계에서 생성한 네임스페이스에 NFD Operator를 설치합니다.
   a. 다음 **OperatorGroup** CR을 생성하고 해당 YAML을 **nfd-operatorgroup.yaml** 파일에 저장합니다.

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

b. 다음 명령을 실행하여 **OperatorGroup** CR을 생성합니다.

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

c. 다음 **Subscription** CR을 생성하고 해당 YAML을 **nfd-sub.yaml** 파일에 저장합니다.

서브스크립션의 예

```
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. 다음 명령을 실행하여 서브스크립션 오브젝트를 생성합니다.

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

e. **openshift-nfd** 프로젝트로 변경합니다.

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

검증

- Operator 배포가 완료되었는지 확인하려면 다음을 실행합니다.

```
$ oc get pods
```

출력 예

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

성공적인 배포에는 **Running** 상태가 표시됩니다.

4.2.2. 웹 콘솔을 사용하여 NFD Operator 설치

클러스터 관리자는 웹 콘솔을 사용하여 NFD Operator를 설치할 수 있습니다.

절차

1. OpenShift Container Platform 웹 콘솔에서 **Operator → OperatorHub**를 클릭합니다.
2. 사용 가능한 Operator 목록에서 **Node Feature Discovery**를 선택한 다음 **설치**를 클릭합니다.
3. **Operator 설치** 페이지에서 **클러스터의 특정 네임스페이스를** 선택한 다음 **설치를** 클릭합니다. 네임스페이스가 생성되므로 생성할 필요가 없습니다.

검증

다음과 같이 NFD Operator가 설치되었는지 확인합니다.

1. **Operator → 설치된 Operator** 페이지로 이동합니다.
2. **Node Feature Discovery**가 **openshift-nfd** 프로젝트에 **InstallSucceeded 상태**로 나열되어 있는지 확인합니다.

참고

설치 중에 Operator는 **실패** 상태를 표시할 수 있습니다. 나중에 **InstallSucceeded** 메시지와 함께 설치에 성공하면 이 **실패** 메시지를 무시할 수 있습니다.

문제 해결

Operator가 설치된 것으로 나타나지 않으면 다음과 같이 추가 문제 해결을 수행합니다.

1. **Operator → 설치된 Operator** 페이지로 이동하고 **Operator 서브스크립션** 및 **설치 계획** 탭의 **상태**에 장애나 오류가 있는지 검사합니다.
2. **Workloads → Pod** 페이지로 이동하여 **openshift-nfd** 프로젝트에서 Pod 로그를 확인합니다.

4.3. NODE FEATURE DISCOVERY OPERATOR 사용

NFD(Node Feature Discovery) Operator는 **NodeFeatureDiscovery** CR을 확인하여 Node-Versionature-Discovery 데몬 세트를 실행하는 데 필요한 모든 리소스를 오케스트레이션합니다. **NodefeatureatureDiscovery** CR을 기반으로 Operator는 원하는 네임스페이스에 피연산자(NFD) 구성 요소를 생성합니다. CR을 편집하여 다른 옵션 중에서 다른 **namespace**, **image**, **imagePullPolicy**, **nfd-worker-conf**를 선택할 수 있습니다.

클러스터 관리자는 OpenShift Container Platform CLI 또는 웹 콘솔을 사용하여 **NodeFeatureDiscovery** 인스턴스를 만들 수 있습니다.

4.3.1. CLI를 사용하여 NodeEnatureDiscovery 인스턴스 생성

클러스터 관리자는 CLI를 사용하여 **NodefeatureatureDiscovery** CR 인스턴스를 생성할 수 있습니다.

사전 요구 사항

- OpenShift Container Platform 클러스터
- OpenShift CLI(**oc**)를 설치합니다.

- **cluster-admin** 권한이 있는 사용자로 로그인합니다.
- NFD Operator를 설치합니다.

절차

1. 다음 **NodeEnatureDiscovery** 사용자 정의 리소스(CR)를 생성한 다음 YAML을 **NodefeatureatureDiscovery.yaml** 파일에 저장합니다.

```
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:
#         - pciId:
#             class: ["0200"]
#             vendor: ["15b3"]
#             device: ["1014", "1017"]
#         - pciId :
#             vendor: ["8086"]
#             device: ["1000", "1100"]
#     - name: "my.usb.feature"
```

```
      #     matchOn:
      #       - usbId:
      #         class: ["ff"]
      #         vendor: ["03e7"]
      #         device: ["2485"]
      #       - usbId:
      #         class: ["fe"]
      #         vendor: ["1a6e"]
      #         device: ["089a"]
      #   - name: "my.combined.feature"
      #     matchOn:
      #       - pciId:
      #           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. 다음 명령어를 실행하여 **NodeFeatureDiscovery** CR 인스턴스를 만듭니다.

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

검증

- 인스턴스가 생성되었는지 확인하려면 다음을 실행합니다.

```
$ oc get pods
```

출력 예

```
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
```

성공적인 배포에는 **Running** 상태가 표시됩니다.

4.3.2. 웹 콘솔을 사용하여 NodeEnatureDiscovery CR 만들기

절차

1. **Operator → 설치된 Operator** 페이지로 이동합니다.

2. Node Feature Discovery를 찾고 **제공된 APIs**아래에 있는 상자를 확인합니다.
3. **인스턴스 만들기**를 클릭합니다.
4. **NodefeatureatureDiscovery** CR의 값을 편집합니다.
5. **생성**을 클릭합니다.

4.4. NODE FEATURE DISCOVERY OPERATOR 설정

4.4.1. 코어

core 섹션에는 특정 기능 소스와 관련이 없는 일반적인 구성 설정이 포함되어 있습니다.

core.sleepInterval
core.sleepInterval은 기능 검색 또는 재검색의 연속 통과 간격과 노드 레이블 재지정 간격을 지정합니다. 양수가 아닌 값은 무한 절전 상태를 의미합니다. 재검색되거나 레이블이 다시 지정되지 않습니다.

이 값은 지정된 경우 더 이상 사용되지 않는 **--sleep-interval** 명령줄 플래그로 재정의됩니다.

사용 예

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

기본값은 **60s**입니다.

core.sources
core.sources는 활성화된 기능 소스 목록을 지정합니다. 특수한 값 **all**은 모든 기능 소스를 활성화합니다.

이 값은 지정된 경우 더 이상 사용되지 않는 **--sources** 명령줄 플래그로 재정의됩니다.

기본값: **[all]**

사용 예

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

core.labelWhiteList
core.labelWhiteList는 레이블 이름을 기반으로 기능 레이블을 필터링하기 위한 정규식을 지정합니다. 일치하지 않는 레이블은 게시되지 않습니다.

정규 표현식은 레이블의 기반 이름 부분인 '/' 뒤에 있는 이름의 부분과만 일치합니다. 레이블 접두사 또는 네임스페이스가 생략됩니다.

이 값은 지정된 경우 더 이상 사용되지 않는 **--label-whitelist** 명령줄 플래그로 재정의됩니다.

기본값: **null**

사용 예

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

core.noPublish
core.noPublish를 **true**로 설정하면 **nfd-master**와의 모든 통신이 비활성화됩니다. 이것은 실질적으로는 드라이런 플래그입니다. **nfd-worker**는 정상적으로 기능 감지를 실행하지만 레이블 요청은 **nfd-master**로 전송되지 않습니다.

이 값은 지정된 경우 **--no-publish** 명령줄 플래그로 재정의됩니다.

예제:

사용 예

```
core:
  noPublish: true 1
```

기본값은 **false**입니다.

core.klog
다음 옵션은 대부분 런타임에 동적으로 조정할 수 있는 로거 구성을 지정합니다.

로거 옵션은 명령줄 플래그를 사용하여 지정할 수도 있으며, 이러한 옵션은 해당 구성 파일 옵션보다 우선합니다.

core.klog.addDirHeader
true로 설정하면**core.klog.addDirHeader**에서 파일 디렉터리를 로그 메시지의 헤더에 추가합니다.

기본값: **false**

런타임 설정 가능: yes

core.klog.alsologtostderr
표준 오류 및 파일에 기록합니다.

기본값: **false**

런타임 설정 가능: yes

core.klog.logBacktraceAt
로깅이 file:N 행에 도달하면 스택 추적을 출력합니다.

기본값: **empty**

런타임 설정 가능: yes

core.klog.logDir
비어 있지 않은 경우 이 디렉터리에 로그 파일을 작성합니다.

기본값: **empty**

런타임 설정 가능: no

core.klog.logFile
비어 있지 않은 경우 이 로그 파일을 사용합니다.

기본값: **empty**

런타임 설정 가능: no

core.klog.logFileMaxSize
core.klog.logFileMaxSize는 로그 파일의 최대 크기를 정의합니다. 단위는 메가바이트입니다. 값이 **0**인 경우 최대 파일 크기는 무제한입니다.

기본값: **1800**

런타임 설정 가능: no

core.klog.logtostderr
파일 대신 표준 오류에 기록합니다.

기본값: **true**

런타임 설정 가능: yes

core.klog.skipHeaders
core.klog.skipHeaders가 **true**로 설정된 경우 로그 메시지에서 헤더 접두사를 사용하지 않습니다.

기본값: **false**

런타임 설정 가능: yes

core.klog.skipLogHeaders
core.klog.skipLogHeaders가 **true**로 설정된 경우 로그 파일을 열 때 헤더를 사용하지 않습니다.

기본값: **false**

런타임 설정 가능: no

core.klog.stderrthreshold
임계값 이상의 로그는 stderr에 있습니다.

기본값: **2**

런타임 설정 가능: yes

core.klog.v
core.klog.v는 로그 수준 세부 정보 표시의 수치입니다.

기본값: **0**

런타임 설정 가능: yes

core.klog.vmodule
core.klog.vmodule은 파일 필터링된 로깅의 쉼표로 구분된 **pattern=N** 설정 목록입니다.

기본값: **empty**

런타임 설정 가능: yes

4.4.2. 소스

source 섹션에는 기능 소스 관련 구성 매개변수가 포함되어 있습니다.

sources.cpu.cpuid.attributeBlacklist
이 옵션에 나열된 **cpuid** 기능만을 공개합니다.

이 값은 **sources.cpu.cpuid.attributeWhitelist**로에 의해 재정의됩니다.

기본값: **[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]**

사용 예

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

sources.cpu.cpuid.attributeWhitelist
이 옵션에 나열된 **cpuid** 기능만 게시합니다.

sources.cpu.cpuid.attributeWhitelist는 **sources.cpu.cpuid.attributeBlacklist**보다 우선합니다.

기본값: **empty**

사용 예

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

sources.kernel.kconfigFile
sources.kernel.kconfigFile은 커널 구성 파일의 경로입니다. 비어 있는 경우 NFD는 일반적인 표준 위치에서 검색을 실행합니다.

기본값: **empty**

사용 예

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

sources.kernel.configOpts
Source.kernel.configOpts는 기능 레이블로 게시하는 커널 구성 옵션을 나타냅니다.

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

사용 예

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

sources.pci.deviceClassWhitelist

source.pci.deviceClassWhitelist 는 레이블을 게시할 PCI 장치 클래스 ID 목록입니다. 메인 클래스로만 (예: **03**) 또는 전체 클래스-하위 클래스 조합(예: **0300**)으로 지정할 수 있습니다. 전자는 모든 하위 클래스가 허용됨을 의미합니다. 레이블 형식은 **deviceLabelFields**를 사용하여 추가로 구성할 수 있습니다.

기본값: **["03", "0b40", "12"]**

사용 예

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

sources.pci.deviceLabelFields
source.pci.deviceLabelFields 는 기능 레이블의 이름을 구성할 때 사용할 PCI ID 필드의 집합입니다. 유효한 필드는 **class**,**vendor**,**device**,**subsystem_vendor** 및 **subsystem_device**입니다.

기본값: **[class, vendor]**

사용 예

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

위의 설정 예제에서 NFD는 **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true**와 같은 레이블을 게시합니다.

sources.usb.deviceClassWhitelist
source.usb.deviceClassWhitelist 는 기능 레이블을 게시할 USB 장치 클래스 ID 목록입니다. 레이블 형식은 **deviceLabelFields**를 사용하여 추가로 구성할 수 있습니다.

기본값: **["0e", "ef", "fe", "ff"]**

사용 예

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

sources.usb.deviceLabelFields
source.usb.deviceLabelFields 는 기능 레이블의 이름을 구성할 USB ID 필드의 집합입니다. 유효한 필드는 **class**,**vendor**, **device**입니다.

기본값: **[class, vendor, device]**

사용 예

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

위의 config 예제에서 NFD는 **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true**와 같은 레이블을 게시합니다.

sources.custom

source.custom 은 사용자별 레이블을 생성하기 위해 사용자 지정 기능 소스에서 처리할 규칙 목록입니다.

기본값: **empty**

사용 예

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