OpenShift Container Platform 4.9

专用硬件和驱动程序启用

了解 OpenShift Container Platform 中的硬件启用

Last Updated: 2022-01-15

OpenShift Container Platform 4.9 专用硬件和驱动程序启用

了解 OpenShift Container Platform 中的硬件启用

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.

法律通告

摘要

本文档概述 OpenShift Container Platform 中的硬件启用。

目录

第 1 章 关于专用硬件和驱动程序启用

许多应用程序需要依赖于内核模块或驱动程序的专用硬件或软件。您可以使用驱动程序容器在 Red Hat Enterprise Linux CoreOS (RHCOS) 节点上载入树外内核模块。要在集群安装过程中部署树外驱动程序，请使用 **kmods-via-containers** 框架。为了在现有 OpenShift Container Platform 集群中载入驱动程序或内核模块，OpenShift Container Platform 提供了几个工具：

- Driver Toolkit 是一个容器镜像，是每个 OpenShift Container Platform 版本的一部分。它包含构建驱动程序或内核模块所需的内核软件包和其他常见依赖项。Driver Toolkit 可用作 OpenShift Container Platform 上构建的驱动程序容器镜像的基础镜像。

- 特殊资源 Operator (SRO) 编配驱动程序容器的构建和管理，以便在现有 OpenShift 或 Kubernetes 集群上加载内核模块和驱动程序。

- Node Feature Discovery (NFD) Operator 为 CPU 功能、内核版本、PCIe 设备供应商 ID 等添加节点标签。

第 2 章 驱动程序工具包

了解驱动程序工具包以及如何将其用作驱动程序容器的基础镜像，以便在 Kubernetes 上启用特殊软件和硬件设备。

重要

Driver Toolkit 只是一个技术预览功能。技术预览功能不被红帽产品服务等级协议 (SLA) 支持，且可能在功能方面有缺陷。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的详情，请参阅 https://access.redhat.com/support/offerings/techpreview/。

2.1. 关于驱动程序工具包

背景信息

Driver Toolkit 是 OpenShift Container Platform 有效负载中的一个容器镜像，用作可构建驱动程序容器的基础镜像。Driver Toolkit 镜像包含通常作为构建或安装内核模块的依赖项所需的内核软件包，以及驱动程序容器所需的一些工具。这些软件包的版本将与相应 OpenShift Container Platform 发行版本中的 Red Hat Enterprise Linux CoreOS（RHCOS）节点上运行的内核版本匹配。

驱动程序容器是容器镜像，用于在容器操作系统（如 RHCOS）上构建和部署树外内核模块和驱动程序。内核模块和驱动程序是在操作系统内核中具有高级别权限运行的软件库。它们扩展了内核功能，或者提供控制新设备所需的硬件特定代码。示例包括 Field Programmable Gate Arrays（FPGA）或 GPU 等硬件设备，以及软件定义型存储（SDS）解决方案（如 Lustre parallel 文件系统，它在客户端机器上需要内核模块）。驱动程序容器是用于在 Kubernetes 上启用这些技术的软件堆栈的第一层。

Driver Toolkit 中的内核软件包列表包括以下内容及其依赖项：

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

另外，Driver Toolkit 还包含相应的实时内核软件包：

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

Driver Toolkit 还有几个通常需要的工具来构建和安装内核模块，其中包括：

- **elfutils-libelf-devel**
- **kmod**

- **binutilskabi-dw**
- **kernel-abi-whitelists**
- 以上的依赖项

用途

在出现 Driver Toolkit 之前，您可以在 OpenShift Container Platform 中的一个 pod 中安装内核软件包，或在构建配置中使用 entitled builds，或从主机 **machine-os-content** 的内核 RPM 进行安装。Driver Toolkit 通过删除授权步骤简化了流程，并避免了访问 pod 中的 machine-os-content 特权操作。Driver Toolkit 也可以由有权访问预发布的 OpenShift Container Platform 版本的合作伙伴使用，用于未来的 OpenShift Container Platform 版本的硬件设备的预构建 driver-containers。

特殊资源 Operator（SRO）也使用 Driver Toolkit，目前作为 OperatorHub 上的社区 Operator 提供。SRO 支持树外和第三方内核驱动程序以及底层操作系统的支持软件。用户可以为 SRO 创建*配方*（*recipes*）来构建和部署驱动程序容器，以及支持诸如设备插件或指标的软件。配方可以包含构建配置，以基于 Driver Toolkit 构建驱动程序容器，或者 SRO 可以部署预构建驱动程序容器。

2.2. 拉取 DRIVER TOOLKIT 容器镜像

driver-toolkit 镜像包括在 Red Hat Ecosystem Catalog 的容器镜像部分 和 OpenShift Container Platform 发行版本有效负载中。与 OpenShift Container Platform 最新次要版本对应的镜像将标记为目录中的版本号。具体版本的镜像 URL 可使用 **oc adm** CLI 命令找到。

2.2.1. 从 registry.redhat.io 中拉取 Driver Toolkit 容器镜像

Red Hat Ecosystem Catalog 包括了使用 podman 或 OpenShift Container Platform 从 **registry.redhat.io** 中拉取 **driver-toolkit** 镜像的说明。最新次版本的 driver-toolkit 镜像将标记为 registry.redhat.io 中的次版本，如 **registry.redhat.io/openshift4/driver-toolkit-rhel8:v4.9**。

2.2.2. 在有效负载中查找驱动程序工具包镜像 URL

先决条件

- 从 Red Hat OpenShift Cluster Manager 站点的 Pull Secret 页面获取执行 OpenShift Container Platform 安装所需的镜像 pull secret。
- 已安装 OpenShift CLI（**oc**）。

流程

1. 可以使用 **oc adm** 命令从发行镜像中提取与特定发行版本对应的 **driver-toolkit** 镜像 URL：

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

输出示例

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

2. 可以使用有效的 pull secret 拉取此镜像，如安装 OpenShift Container Platform 所需的 pull secret。

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

2.3. 使用 DRIVER TOOLKIT

例如，Driver Toolkit 可用作基础镜像来构建非常简单的内核模块，名为 simple-kmod。

注意

Driver Toolkit 包含为内核模块签名所需的依赖项、**openssl**、**mokutil** 和 **keyutils**。但是，在这个示例中，simple-kmod 内核模块没有签名，因此无法在启用了**安全引导（Secure Boot）**的系统中载入。

2.3.1. 在集群中构建并运行 simple-kmod 驱动程序容器

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 您可以将集群的 Image Registry Operator 状态设置为 **Managed**。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。

流程

创建命名空间。例如：

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

1. YAML 定义了 **ImageStream**，用于存储 **simple-kmod** 驱动程序容器镜像，以及用于构建容器的 **BuildConfig**。将此 YAML 保存为 **0000-buildconfig.yaml.template**。

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  labels:
    app: simple-kmod-driver-container
  name: simple-kmod-driver-container
  namespace: simple-kmod-demo
spec: {}
---
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  labels:
    app: simple-kmod-driver-build
  name: simple-kmod-driver-build
  namespace: simple-kmod-demo
spec:
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  runPolicy: "Serial"
```

```
  triggers:
    - type: "ConfigChange"
    - type: "ImageChange"
  source:
    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
```

2. 在以下命令中，使用您运行的 OpenShift Container Platform 版本的相关的正确 driver toolki 镜像替换 "DRIVER_TOOLKIT_IMAGE" 部分。

```
$ OCP_VERSION=$(oc get clusterversion/version -ojsonpath={.status.desired.version})
```

```
$ DRIVER_TOOLKIT_IMAGE=$(oc adm release info $OCP_VERSION --image-for=driver-toolkit)
```

```
$ sed "s#DRIVER_TOOLKIT_IMAGE#${DRIVER_TOOLKIT_IMAGE}#" 0000-buildconfig.yaml.template > 0000-buildconfig.yaml
```

3. 使用创建镜像流和构建配置

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

4. 构建器 Pod 成功完成后，将驱动程序容器镜像部署为 **DaemonSet**。

   a. 驱动程序容器必须使用特权安全上下文运行，才能在主机上加载内核模块。以下 YAML 文件包含用于运行驱动程序容器的 RBAC 规则和 **DaemonSet**。将此 YAML 保存为 **1000-drivercontainer.yaml**。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: simple-kmod-driver-container
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: simple-kmod-driver-container
rules:
- apiGroups:
  - security.openshift.io
  resources:
  - securitycontextconstraints
  verbs:
  - use
  resourceNames:
  - privileged
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: simple-kmod-driver-container
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: simple-kmod-driver-container
subjects:
- kind: ServiceAccount
  name: simple-kmod-driver-container
userNames:
- system:serviceaccount:simple-kmod-demo:simple-kmod-driver-container
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
```

```
  name: simple-kmod-driver-container
spec:
  selector:
    matchLabels:
      app: simple-kmod-driver-container
  template:
    metadata:
      labels:
        app: simple-kmod-driver-container
    spec:
      serviceAccount: simple-kmod-driver-container
      serviceAccountName: simple-kmod-driver-container
      containers:
      - image: image-registry.openshift-image-registry.svc:5000/simple-kmod-
demo/simple-kmod-driver-container:demo
        name: simple-kmod-driver-container
        imagePullPolicy: Always
        command: ["/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. 创建 RBAC 规则和守护进程集：

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

5. 当 pod 在 worker 节点上运行后，使用 **lsmod** 验证在主机机器上是否成功载入了 **simple_kmod** 内核模块。

a. 验证 pod 是否正在运行：

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

输出示例

```
NAME                                READY   STATUS      RESTARTS   AGE
simple-kmod-driver-build-1-build    0/1     Completed   0          6m
simple-kmod-driver-container-b22fd   1/1     Running     0          40s
simple-kmod-driver-container-jz9vn   1/1     Running     0          40s
simple-kmod-driver-container-p45cc   1/1     Running     0          40s
```

b. 在驱动程序容器 pod 中执行 **lsmod** 命令：

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

输出示例

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

2.4. 其他资源

- 有关为集群配置 registry 存储的更多信息，请参阅 OpenShift Container Platform 中的 Image Registry Operator。

第 3 章 特殊资源 OPERATOR

了解特殊资源 Operator (SRO) 以及如何使用它来构建和管理驱动程序容器，以便将内核模块和设备驱动程序加载到 OpenShift Container Platform 集群的节点上。

重要

特殊资源 Operator 只是一个技术预览功能。技术预览功能不被红帽产品服务等级协议 (SLA) 支持，且可能在功能方面有缺陷。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的详情，请参阅 https://access.redhat.com/support/offerings/techpreview/。

3.1. 关于特殊资源 OPERATOR

特殊资源 Operator (SRO) 可帮助您管理现有 OpenShift Container Platform 集群上的内核模块和驱动程序的部署。SRO 可用于像构建和加载单个内核模块那样简单的情况，或者像为硬件加速器部署驱动程序、设备插件和监控堆栈这样的复杂情况。

对于载入内核模块，SRO 围绕使用驱动程序容器设计。云原生环境（特别是在纯容器操作系统上运行）中越来越多地使用驱动程序容器向主机提供硬件驱动程序。驱动程序容器将内核堆栈扩展至特定内核的现成软件和硬件功能之外。驱动程序容器在各种支持容器的 Linux 发行版上工作。对于驱动程序容器，主机操作系统保持干净，主机上不同库版本或二进制文件之间没有冲突。

3.2. 安装特殊资源 OPERATOR

作为集群管理员，您可以使用 OpenShift CLI 或 Web 控制台安装特殊 资源 Operator(SRO)。

3.2.1. 使用 CLI 安装特殊资源 Operator

作为集群管理员，您可以使用 OpenShift CLI 安装特殊资源 Operator (SRO)。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。
- 已安装 Node Feature Discovery (NFD) Operator。

流程

1. 为特殊资源 Operator 创建命名空间：

   a. 创建以下用于定义 **openshift-special-resource-operator** 命名空间的 **Namespace** 自定义资源 (CR)，然后在 **sro-namespace.yaml** 文件中保存 YAML：

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

b. 运行以下命令创建命名空间：

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

2. 在您上一步中创建的命名空间中安装 SRO：

a. 创建以下 **OperatorGroup** CR，并在 **sro-operatorgroup.yaml** 文件中保存 YAML：

```
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. 运行以下命令来创建 operator 组：

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

c. 运行以下 **oc get** 命令以获取下一步所需的 **channel** 值：

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

输出示例

```
4.9
```

d. 创建以下 **Subscription** CR，并将 YAML 保存到 **sro-sub.yaml** 文件中：

Subscription CR 示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-special-resource-operator
  namespace: openshift-special-resource-operator
spec:
  channel: "4.9" (1)
  installPlanApproval: Automatic
  name: openshift-special-resource-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

(1) 将 channel 值替换为上一命令的输出。

e. 运行以下命令来创建订阅对象：

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

f. 切换到 **openshift-special-resource-operator** 项目：

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

验证

- 要验证 Operator 部署是否成功，请运行：

```
$ oc get pods
```

输出示例

```
NAME                                              READY   STATUS    RESTARTS   AGE
special-resource-controller-manager-7bfb544d45-xx62r   2/2     Running   0          2m28s
```

一个成功的部署会显示 **Running** 状态。

3.2.2. 使用 Web 控制台安装特殊资源 Operator

作为集群管理员，您可以使用 OpenShift Container Platform Web 控制台安装特殊 Resource Operator (SRO)。

先决条件

- 已安装 Node Feature Discovery (NFD) Operator。

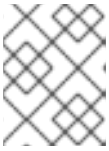
流程

1. 登陆到 OpenShift Container Platform Web 控制台。
2. 为特殊 Resource Operator 创建所需的命名空间：
   a. 进行 **Administration → Namespaces**，点 **Create Namespace**。
   b. 在 **Name** 字段中输入 **openshift-special-resource-operator**，点 **Create**。
3. 安装特殊资源 Operator：
   a. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
   b. 从可用的 Operator 列表中选择 **Special Resource Operator**，然后单击 **Install**。
   c. 在 **Install Operator** 页面中，选择**集群上的一个特定命名空间**，选择上一节中创建的命名空间，然后点 **Install**。

验证

验证特殊 Resource Operator 是否已成功安装：

1. 导航到 **Operators → Installed Operators** 页面。
2. 确保 **openshift-special-resource-operator** 项目中列出的 **Special Resource Operatorr** 的 **Status** 为 **InstallSucceeded**。



注意

在安装过程中，Operator 可能会显示 **Failed** 状态。如果安装过程结束后有 **InstallSucceeded** 信息，您可以忽略这个 **Failed** 信息。

3. 如果 Operator 没有被成功安装，请按照以下步骤进行故障排除：

   a. 导航到 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。

   b. 导航到 **Workloads → Pods** 页面，在 **openshift-special-resource-operator** 项目中检查 pod 的日志。

注意

Node Feature Discovery (NFD) Operator 是特殊 Resource Operator (SRO) 的依赖项。如果在安装 SRO 之前没有安装 NFD Operator，Operator Lifecycle Manager 将自动安装 NFD Operator。但是，所需的节点功能发现操作对象不会被自动部署。Node Feature Discovery Operator 文档提供了有关如何使用 NFD Operator 部署 NFD 的详细信息。

3.3. 使用特殊资源 OPERATOR

特殊资源 Operator (SRO) 用于管理驱动程序容器的构建和部署。构建和部署容器所需的对象可以在 Helm Chart 中定义。

本节中的示例使用 simple-kmod 内核模块演示如何使用 SRO 构建和运行驱动程序容器。在第一个示例中，SRO 镜像包含一个本地 Helm chart 仓库，包括用于部署 simple-kmod 内核模块的模板。在本例中，使用 **SpecialResource** 清单来部署驱动程序容器。在第二个示例中，simple-kmod **SpecialResource** 对象指向为存储 Helm chart 而创建的 **ConfigMap** 对象。

3.3.1. 使用 SRO 镜像中的模板来构建和运行 simple-kmod SpecialResource

SRO 镜像包含 Helm chart 的本地仓库，包括用于部署 simple-kmod 内核模块的模板。在本例中，simple-kmod 内核模块用于显示 SRO 如何管理内部 SRO 存储库中定义的驱动程序容器。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 您可以将集群的 Image Registry Operator 状态设置为 **Managed**。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。
- 已安装 Node Feature Discovery (NFD) Operator。
- 已安装特殊 Resource Operator。

流程

1. 要使用 SRO 镜像的本地 Helm 仓库部署 simple-kmod，请使用以下 **SpecialResource** 清单。将此 YAML 保存为 **simple-kmod-local.yaml**。

```
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. 创建 **SpecialResource**:

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

simple-kmod 资源部署在 **simple-kmod** 命名空间中，如对象清单中指定的。片刻之后，**simple-kmod** 驱动程序容器的构建 pod 开始运行。构建在几分钟后完成，然后驱动程序容器容器集开始运行。

3. 使用 **oc get pods** 命令显示 pod 的状态：

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

输出示例

```
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. 要显示 simple-kmod 驱动程序容器镜像构建的日志，请使用 **oc logs** 命令以及上面获取的构建 pod 名称：

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

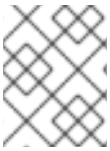
5. 要验证是否载入了 simple-kmod 内核模块，请在上面的 **oc get pods** 命令返回的一个驱动程序容器 pod 中执行 **lsmod** 命令：

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

-

输出示例

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



注意

如果要从节点中删除 simple-kmod 内核模块，请使用 **oc delete** 命令删除 simple-kmod **SpecialResource** API 对象。删除驱动程序容器 pod 时，内核模块会被卸载。

3.3.2. 使用配置映射构建并运行 simple-kmod SpecialResource

在本例中，simple-kmod 内核模块用于显示 SRO 如何管理存储在配置映射中的 Helm Chart 模板中的驱动程序容器。

先决条件

- 有一个正在运行的 OpenShift Container Platform 集群。
- 您可以将集群的 Image Registry Operator 状态设置为 **Managed**。
- 已安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 权限的用户身份登录 OpenShift CLI。
- 已安装 Node Feature Discovery (NFD) Operator。
- 已安装特殊 Resource Operator。
- 已安装 Helm CLI (**helm**)。

流程

1. 要创建 simple-kmod **SpecialResource** 对象，请定义用于构建镜像的镜像流和构建配置，以及用于运行容器的服务帐户、角色、角色绑定和守护进程集。需要服务帐户、角色和角色绑定来运行具有特权安全上下文的守护进程集，以便加载内核模块。

   a. 创建 **templates** 目录，并更改到此目录：

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

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

   b. 将镜像流和构建配置的 YAML 模板保存到 **templates** 目录中的 **0000-buildconfig.yaml** 中：

   ```
   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}}** 等模板由 SRO 填写，具体基于 **SpecialResource** CR 中的字段和 Operator 已知的变量，如 **{{.Values.KernelFullVersion}}**。

c. 将 **templates** 目录中的 RBAC 资源和守护进程设置的以下 YAML 模板保存为 **1000-driver-container.yaml** ：

```
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. 进入 **chart/simple-kmod-0.0.1** 目录：

```
$ cd ..
```

e. 在 **chart/simple-kmod-0.0.1** 目录中，将 Chart 的以下 YAML 保存为 **Chart.yaml** ：

```
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. 在 **Chart** 目录中，使用 **helm package** 命令创建 chart：

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

输出示例

```
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. 创建配置映射以存储 chart 文件：

   a. 为配置映射文件创建目录：

   ```
   $ mkdir cm
   ```

   b. 将 Helm Chart 复制到 **cm** 目录中：

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

   c. 创建一个索引文件，指定包含 Helm Chart 的 Helm 仓库：

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

   d. 为 Helm Chart 中定义的对象创建一个命名空间：

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

   e. 创建配置映射对象：

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

4. 使用以下 **SpecialResource** 清单，使用您在配置映射中创建的 Helm Chart 部署 simple-kmod 对象。将此 YAML 保存为 **simple-kmod-configmap.yaml** ：

```
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. 可选：取消注释 **#debug: true** 行，使 chart 中的 YAML 文件完整显示在 Operator 日志中，并验证日志是否已正确创建并模板化。

2. **spec.chart.repository.url** 字段指示 SRO 在配置映射中查找 chart。

5. 在命令行中创建 **SpecialResource** 文件：

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

simple-kmod 资源部署在 **simple-kmod** 命名空间中，如对象清单中指定的。片刻之后，**simple-kmod** 驱动程序容器的构建 pod 开始运行。构建在几分钟后完成，然后驱动程序容器容器集开始运行。

6. 使用 **oc get pods** 命令显示构建 pod 的状态：

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

输出示例

```
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
```

7. 使用 **oc logs** 命令以及从上述 **oc get pods** 命令获取的构建 pod 名称，以显示 simple-kmod 驱动程序容器镜像构建的日志：

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

8. 要验证是否载入了 simple-kmod 内核模块，请在上面的 **oc get pods** 命令返回的一个驱动程序容器 pod 中执行 **lsmod** 命令：

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

输出示例

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

注意

如果要从节点中删除 simple-kmod 内核模块，请使用 **oc delete** 命令删除 simple-kmod **SpecialResource** API 对象。删除驱动程序容器 pod 时，内核模块会被卸载。

3.4. 其他资源

- 有关在使用特殊 Resource Operator 前恢复 Image Registry Operator 状态的信息，请参阅安装过程中删除的镜像 registry。
- 有关安装 NFD Operator 的详情，请参阅 Node Feature Discovery (NFD) Operator。

第 4 章 NODE FEATURE DISCOVERY OPERATOR

了解 Node Feature Discovery（NFD）Operator 以及如何使用它通过编排节点功能发现（用于检测硬件功能和系统配置的 Kubernetes 附加组件）来公开节点级信息。

4.1. 关于 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery Operator(NFD)通过标记节点使其包含特定于硬件的信息来管理 OpenShift Container Platform 集群中的硬件功能和配置检测。NFD 使用特定于节点的属性标记主机，如 PCI 卡、内核、操作系统版本等。

NFD Operator 可以通过搜索 "Node Feature Discovery" 在 Operator Hub 上找到。

4.2. 安装 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery（NFD）Operator 编排运行 NFD 守护进程集需要的所有资源。作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台安装 NFD Operator。

4.2.1. 使用 CLI 安装 NFD Operator

作为集群管理员，您可以使用 CLI 安装 NFD Operator。

先决条件

- OpenShift Container Platform 集群
- 安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 特权的用户身份登录。

流程

1. 为 NFD Operator 创建命名空间。
   a. 创建定义 **openshift-nfd** 命名空间的以下 **Namespace** 自定义资源（CR），然后在 **nfd-namespace.yaml** 文件中保存 YAML：

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

   b. 运行以下命令创建命名空间：

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

2. 通过创建以下对象，在您上一步中创建的命名空间中安装 NFD Operator：
   a. 创建以下 **OperatorGroup** CR，并在 nfd **-operatorgroup.yaml** 文件中保存 YAML：

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
```

```
  generateName: openshift-nfd-
  name: openshift-nfd
  namespace: openshift-nfd
spec:
  targetNamespaces:
  - openshift-nfd
```

b. 运行以下命令来创建 **OperatorGroup** CR:

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

c. 运行以下命令获取下一步所需的 **channel** 值。

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

输出示例

```
4.9
```

d. 创建以下 **Subscription** CR，并将 YAML 保存到 **nfd-sub.yaml** 文件中：

订阅示例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: nfd
  namespace: openshift-nfd
spec:
  channel: "4.9"
  installPlanApproval: Automatic
  name: nfd
  source: redhat-operators
  sourceNamespace: openshift-marketplace
```

e. 运行以下命令来创建订阅对象：

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

f. 进入 **openshift-nfd** 项目：

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

验证

- 要验证 Operator 部署是否成功，请运行：

```
$ oc get pods
```

输出示例

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

一个成功的部署会显示 **Running** 状态。

4.2.2. 使用 Web 控制台安装 NFD Operator

作为集群管理员，您可以使用 Web 控制台安装 NFD Operator。

注意

建议按照上一节所述创建**命名空间**。

流程

1. 在 OpenShift Container Platform Web 控制台中，点击 **Operators → OperatorHub**。
2. 从可用的 Operator 列表中选择 **Node Feature Discovery**，然后点 **Install**。
3. 在 **Install Operator** 页面中，选择**集群上的一个特定命名空间**，选择上一节中创建的命名空间，然后点 **Install**。

验证

验证 NFD Operator 是否已成功安装：

1. 导航到 **Operators → Installed Operators** 页面。
2. 确保 **openshift-nfd** 项目中列出了 **Node Feature Discovery**，**Status** 为 **InstallSucceeded**。

注意

在安装过程中，Operator 可能会显示 **Failed** 状态。如果安装过程结束后有 **InstallSucceeded** 信息，您可以忽略这个 **Failed** 信息。

故障排除

如果 Operator 没有被安装，请按照以下步骤进行故障排除：

1. 导航到 **Operators → Installed Operators** 页面，检查 **Operator Subscriptions** 和 **Install Plans** 选项卡中的 **Status** 项中是否有任何错误。
2. 导航到 **Workloads → Pods** 页面，在 **openshift-nfd** 项目中检查 pod 的日志。

4.3. 使用 NODE FEATURE DISCOVERY OPERATOR

Node Feature Discovery（NFD）Operator 通过监视 **NodeFeatureDiscovery** CR 来编排运行 Node-Feature-Discovery 守护进程所需的所有资源。根据 **NodeFeatureDiscovery** CR，Operator 将在所需命名空间中创建操作对象（NFD）组件。您可以编辑 CR 来选择另一个 **命名空间**、**镜像**、**imagePullPolicy** 和 **nfd-worker-conf**，以及其他选项。

作为集群管理员，您可以使用 OpenShift Container Platform CLI 或 Web 控制台创建 **NodeFeatureDiscovery** 实例。

4.3.1. 使用 CLI 创建 NodeFeatureDiscovery 实例

作为集群管理员，您可以使用 CLI 创建 **NodeFeatureDiscovery** CR 实例。

先决条件

- OpenShift Container Platform 集群
- 安装 OpenShift CLI（**oc**）。
- 以具有 **cluster-admin** 特权的用户身份登录。
- 安装 NFD Operator。

流程

1. 创建以下 **NodeFeatureDiscovery** 自定义资源（CR），然后在 **NodeFeatureDiscovery.yaml** 文件中保存 YAML：

```
apiVersion: nfd.openshift.io/v1
kind: NodeFeatureDiscovery
metadata:
  name: nfd-instance
  namespace: openshift-nfd
spec:
  instance: "" # instance is empty by default
  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. 运行以下命令来创建 **NodeFeatureDiscovery** CR 实例：

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

验证

- 要验证是否已创建实例，请运行：

```
$ oc get pods
```

输出示例

```
NAME                                  READY   STATUS    RESTARTS   AGE
nfd-controller-manager-7f86ccfb58-vgr4x   2/2     Running   0          11m
nfd-master-hcn64                      1/1     Running   0          60s
nfd-master-lnnxx                      1/1     Running   0          60s
nfd-master-mp6hr                      1/1     Running   0          60s
nfd-worker-vgcz9                      1/1     Running   0          60s
nfd-worker-xqbws                      1/1     Running   0          60s
```

一个成功的部署会显示 **Running** 状态。

4.3.2. 使用 Web 控制台创建 NodeFeatureDiscovery CR

流程

1. 导航到 **Operators → Installed Operators** 页面。
2. 查找 **Node Feature Discovery**，并在 **Provided APIs** 下看到一个方框。
3. 单击 **Create instance**。
4. 编辑 **NodeFeatureDiscovery** CR 的值。
5. 点击 **Create**。

4.4. 配置 NODE FEATURE DISCOVERY OPERATOR

4.4.1. core

core 部分包含不特定于任何特定功能源的常见配置设置。

core.sleepInterval
core.sleepInterval 指定连续通过功能检测或重新检测之间的间隔，还可指定节点重新标记之间的间隔。非正数值意味着睡眠间隔无限；不进行重新检测或重新标记。

如果指定，这个值会被弃用的 **--sleep-interval** 命令行标志覆盖。

示例用法

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

默认值为 **60s**。

core.sources
core.sources 指定启用的功能源列表。特殊值 **all** 可启用所有功能源。

如果指定，这个值会被弃用的 **--sources** 命令行标志覆盖。

默认：**[all]**

示例用法

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

core.labelWhiteList
core.labelWhiteList 根据标签名称指定用于过滤功能标签的正则表达式。不匹配的标签将不会被发布。

正则表达式仅与标签的 basename 部分（"/"后的名称部分）进行匹配。标签前缀或命名空间会被省略。

如果指定，这个值会被弃用的 **--label-whitelist** 命令行标志覆盖。

默认： **null**

示例用法

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

core.noPublish
将 **core.noPublish** 设置为 **true** 可禁用与 **nfd-master** 的所有通信。它实际上是一个空运行标记; **nfd-worker** 会正常运行功能检测，但不会向 **nfd-master** 发送实际的标记请求。

如果指定，**--no-publish** 命令行标志会覆盖这个值。

例如：

示例用法

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

默认值为 **false**。

core.klog
以下选项指定日志记录器配置，其中大多数可以在运行时动态调整。

日志记录器选项也可以使用命令行标志来指定，其优先级高于任何对应的配置文件选项。

core.klog.addDirHeader
如果设置为 **true**，**core.klog.addDirHeader** 将文件目录添加到日志消息的标头中。

默认：**false**

运行时可配置：是

core.klog.alsologtostderr
将日志信息输出到标准错误以及文件。

默认：**false**

运行时可配置：是

core.klog.logBacktraceAt
当日志记录达到行 file:N 时，触发堆栈跟踪功能。

默认： **空**

运行时可配置：是

core.klog.logDir
如果非空，在此目录中写入日志文件。

默认： **空**

运行是时配置：否

core.klog.logFile
如果不为空，则使用此日志文件。

默认： **空**

运行是时配置：否

core.klog.logFileMaxSize
core.klog.logFileMaxSize 定义日志文件可增大的最大大小。单位是 MB。如果值为 **0**，则最大文件大小没有限制。

默认： **1800**

运行是时配置：否

core.klog.logtostderr
将日志信息输出到标准错误而不是文件

默认： **true**

运行时可配置：是

core.klog.skipHeaders
如果 **core.klog.skipHeaders** 设为 **true**，忽略日志消息中的标头前缀。

默认：**false**

运行时可配置：是

core.klog.skipLogHeaders
如果 **core.klog.skipLogHeaders** 设为 **true**，在打开日志文件时忽略标头。

默认：**false**

运行是时配置：否

core.klog.stderrthreshold
处于或超过此阈值的日志输出到 stderr。

默认： **2**

运行时可配置：是

core.klog.v
core.klog.v 是日志级别详细程度的值。

默认： **0**

运行时可配置：是

core.klog.vmodule
core.klog.vmodule 是文件过滤日志的、以逗号分隔的 **pattern=N** 设置列表。

默认： **空**

运行时可配置：是

4.4.2. 源代码

sources 部分包含特定于功能源的配置参数。

sources.cpu.cpuid.attributeBlacklist
防止发布此选项中列出的 **cpuid** 功能。

如果指定，则 **source.cpu.cpuid.attributeWhitelist** 将覆盖这个值。

默认：**[BMI1, BMI2, CLMUL, CMOV, CX16, ERMS, F16C, HTT, LZCNT, MMX, MMXEXT, NX, POPCNT, RDRAND, RDSEED, RDTSCP, SGX, SGXLC, SSE, SSE2, SSE3, SSE4.1, SSE4.2, SSSE3]**

示例用法

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

sources.cpu.cpuid.attributeWhitelist
仅发布在此选项中列出的 **cpuid** 功能。

source.cpu.cpuid.attributeWhitelist 优先于 **source.cpu.cpuid.attributeBlacklist**。

默认： **空**

示例用法

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

sources.kernel.kconfigFile
source.kernel.kconfigFile 是内核配置文件的路径。如果为空，NFD 会在已知的标准位置运行搜索。

默认： **空**

示例用法

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

sources.kernel.configOpts
sources.kernel.configOpts 代表内核配置选项，作为功能标签发布。

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

示例用法

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

sources.pci.deviceClassWhitelist
source.pci.deviceClassWhitelist 是用于发布标签的 PCI 设备类 ID 的列表。它只能指定为主类（例如 **03**）或全类子类组合（例如 **0300**）。前者表示接受所有子类。可以使用 **deviceLabelFields** 进一步配置标签格式。

默认： **["03", "0b40", "12"]**

示例用法

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

sources.pci.deviceLabelFields
sources.pci.deviceLabelFields 是构建功能标签名称时要使用的 PCI ID 字段集。有效字段包括 **class**、**vendor**、**device**、**subsystem_vendor** 和 **subsystem_device**。

默认： **[class, vendor]**

示例用法

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

在上例配置中，NFD 会发布标签，如 **feature.node.kubernetes.io/pci-<class-id>_<vendor-id>_<device-id>.present=true**

sources.usb.deviceClassWhitelist
source.usb.deviceClassWhitelist 是要为其发布功能标签的 USB 设备类 ID 的列表。可以使用 **deviceLabelFields** 进一步配置标签格式。

默认： **["0e", "ef", "fe", "ff"]**

示例用法

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

sources.usb.deviceLabelFields
sources.usb.deviceLabelFields 是 USB ID 字段集合，从中组成功能标签的名称。有效字段包括 **class**、**vendor** 和 **device**。

默认： **[class、vendor、device]**

示例用法

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

使用上面的示例配置，NFD 会发布类似如下标签： **feature.node.kubernetes.io/usb-<class-id>_<vendor-id>.present=true**。

sources.custom
source **.custom** 是自定义功能源中用于创建用户特定标签的规则列表。

默认： **空**

示例用法

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