应用说明

英特尔公司
软件定义数据中心解决方案事业部

Kubernetes* 中的多个网络接口

目录

1.0 简介

一段时间以来，通信服务提供商 (CommSP) 行业一直在积极推广软件定义网络 (SDN) 和网络功能虚拟化 (NFV) 领域的技术发展成果，以实现多方面成效，包括改善服务部署敏捷性、提升运营效率和降低基础设施成本等。最初，这一举措需要通过虚拟化将物理网络功能整合至虚拟机 (VM)。最近，他们的相关工作进入新阶段，开始采用容器提升可扩展性与弹性。这些新的网络功能通常被称为云原生虚拟化网络功能 (VNF)。

Kubernetes 是一种领先的容器编排引擎 (COE)，这种开源系统可用于自动部署、扩展和管理容器化应用。Kubernetes 由 Google 开发，是 Linux 基金会 (LF) 旗下 Cloud Native Computing Foundation (CNCF) 的支柱项目。

然而，Kubernetes 缺乏支持 VNF 中多个网络接口的所需功能。传统上，网络功能使用多个网络接口分离控制、管理和数据/用户网络平面。它们还用于支持不同的协议或软件堆栈，满足不同的调整和配置要求。

本文介绍了英特尔的 Kubernetes 解决方案 Multus，该解决方案可满足这一需求。Multus 是一种容器网络接口 (CNI) 插件，可用于为 Kubernetes 中的 pod 创建多个网络接口。pod 是一种可部署的计算单元，由 Kubernetes 创建和管理。Multus 旨在以更简单的方式将当前的 NFV 用例迁移至容器环境。图 1 显示了使用 Multus 前后的 Kubernetes 网络。

长期来看，英特尔认为采用原生 Kubernetes 方法提供多个网络接口势在必行。随着英特尔与合作伙伴继续携手推进云原生进程，Kubernetes 社区正在评估许多选项。

本文旨在帮助使用 Kubernetes 的工程师深入了解如何为 VNF 配置 Multus，同时介绍了

- Multus 如何利用单根 I/O 虚拟化 (SR-IOV) 提升数据/用户平面吞吐量，以实现高数据包吞吐量和低处理延迟。
- 利用 Multus CNI 插件所需的开源软件组件。

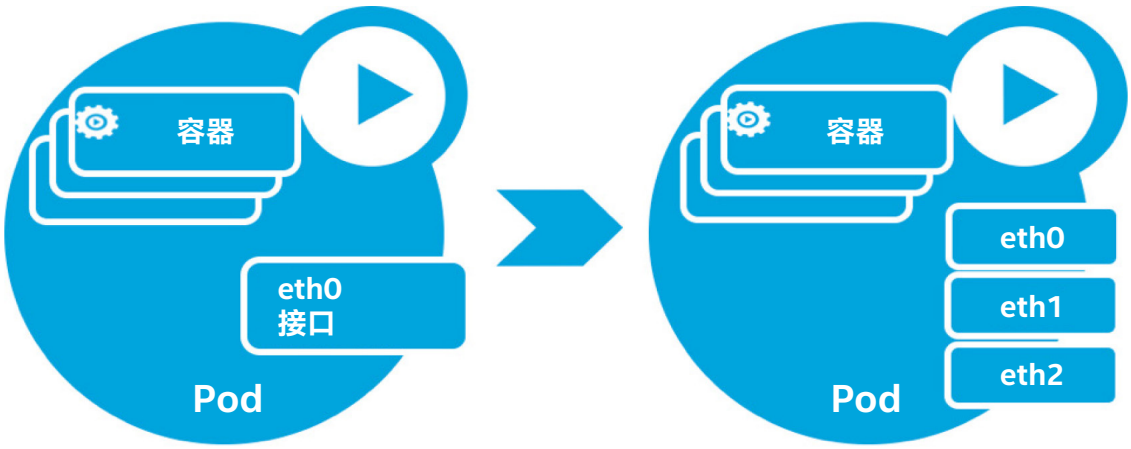

图 1.使用 Multus 前后的 Kubernetes 网络

注：本文未说明如何设置 Kubernetes 集群，并已假设工程师在实施文中步骤之前相关设置已完成。如需查看构建完整系统所需的更多设置和安装指南，请参考附录表格 7.5 中的文档“借助英特尔® 至强® 可扩展处理器为 NFV 用例部署 Kubernetes 和容器裸机平台”。

本文是 Container Experience Kit for Kubernetes Networking 的一部分。该容器体验套件包括一系列用户指南、应用说明和特性简介，以及其他为工程师提供最佳实践文档库的宣传资料，以帮助他们开发基于容器的应用。该体验套件中的其他文档包括：

文档标题	文档类型	位置 URL
Kubernetes 支持多种网络接口	特性简介	https://builders.intel.com/docs/networkbuilders/multiple-network-interfaces-support-in-kubernetes-feature-brief.pdf

2.0 多个网络接口使用 Multus

Multus 是一种 CNI 插件，专用于支持 Kubernetes 环境中的多个网络接口。CNI 是一个 CNCF 项目，该规格和支持框架用于创建相关插件，以在 Linux 容器中创建和配置网络接口。Multus 严格遵守 CNI 规格，您可点击链接 https://github.com/containernetworking/cni/blob/master/SPEC.md 了解相关信息。

在操作上，Multus 相当于其他 CNI 插件的代理和仲裁器，这意味着它在实际创建网络接口时涉及其他 CNI 插件（如 Flannel、Calico、SR-IOV、vHost CNI）。在配置 Multus 时，一个插件必须用作主插件，以配置和管理主网络接口 (eth0)。

在 pod 的所有网络功能配置完成后，只有主网络接口的信息可返回至 Kubernetes。然后，任何数量的其他 CNI 插件可用于创建更多网络接口，但 Kubernetes 并不会获得这些接口的详细信息。这会导致以下结果：尽管 pod 中创建了许多网络接口且这些接口都可被利用，但 Kubernetes 无法支持服务或安全策略，例如在这些网络接口上。

为了解 Multus 的工作原理，您需要回顾 Kubernetes 网络功能的运作方式。Kubernetes 使用网络插件进行网络编排。目前，两种网络插件支持 Kubernetes：

1. **CNI插件：** CNI插件通过实施CNI规格在Linux环境中实现容器网络解决方案的互操作性。
2. **Kubenet：** Kubenet实施具有主机本地CNI插件的基本桥接器 cbr0。另请注意，kubenet 目前被弃用，CNI 是唯一受支持的框架。

图 2 中的序列图显示了 Multus CNI 插件创建多个网络接口的控制流。**Kubelet** 是在 Kubernetes 集群中每个节点上运行的主要代理。其主要功能是在 Kubernetes 控制平面中登记节点，并为后续在该该节点上运行的任何 pod 实施生命周期管理。它还负责为每个 pod 建立网络接口；Kubelet 在实施该操作时序读取 Multus CNI 配置文件，然后使用这些配置设置每个 pod 的网络。在图 2 的设置中，Kubelet 被配置使用 CNI 作为其网络插件。

在被调用来设置 pod 时，Kubelet 会调用其容器运行时（如 Docker 或 CoreOS Rocket (rkt)）设置 pod。Kubelet 还为容器运行时提供一个网络插件包装程序，用于配置其网络。此处指 Multus CNI 插件。Multus 可与配置文件一起使用，支持使用网络对象或二者的组合。在所有这些模式中，Multus 读取配置，将设置网络的实际任务转移至其他被称为代理 (delegate) 的 CNI 插件。第 4.0 部分介绍了网络对象。

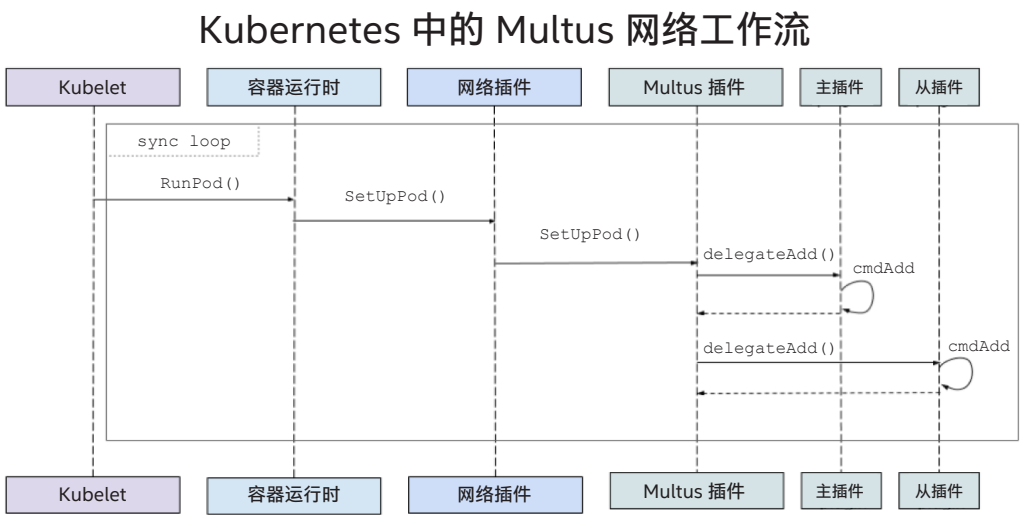

图 2. Kubernetes 中的 Multus 工作流

Multus 然后为每个代理（CNI 插件及其相应配置）调用 **delegateAdd()**。然后，它们调用自己的 **cmdAdd()** 函数，以添加 Pod 的网络接口。这些代理定义需调用的 CNI 插件和它们的参数。这些 CNI 插件的参数可作为 CRD 或 TPR 对象存储在 Kubernetes 中。请参阅第 4.1 部分，了解更多详细信息。

Multus 利用 pod 注释中的网络信息创建更多网络。通过评估 pod 注释，Multus 将确定应调用其他哪些 CNI 插件。

注：插件包括主插件和从插件，不同插件的调用顺序非常重要。

Multus 支持需要多个网络接口的 NFV 用例，同时允许使用在 Kubernetes 中不可使用的接口和软件堆栈，如 SR-IOV 和数据平面开发套件 (DPDK)。

下一部分介绍了使用 SR-IOV 和 DPDK 加速 NFV 数据平面。

3.0 使用 Multus 和 SR-IOV/DPDK 提升网络性能

DPDK 包括一系列开源库和驱动程序，能够围绕操作系统核心发送数据包，最大限度减少数据包收发所需的 CPU 周期数，从而实现快速数据包处理。DPDK 库包括多核框架、大页内存、环形缓冲区和轮询模式驱动程序，可支持联网和其他网络功能。有关更多信息，请访问 www.dpdk.org。

SR-IOV 是一种 PCI-SIG 标准化方法，支持出于可管理性和性能原因隔离 PCI Express (PCIe) 原生硬件资源。实际上，这意味着单个 PCIe 设备 —— 指物理功能 (PF) ——能够以多个单独 PCIe 设备 —— 指虚拟功能 (VF) —— 的形式出现，所需的资源仲裁在设备中进行。

对于 SR-IOV 支持的网络接口卡 (NIC)，每个 VF MAC 和 IP 地址可独立配置，VF 之间的数据包切换在设备硬件中进行。在 Kubernetes pod 中使用 SR-IOV 网络设备的优势包括：

- 与 NIC 设备的直接通信有助于向“裸机”性能靠近。
- 支持用户空间中的多个快速网络数据包处理同步工作负载（例如，基于 DPDK）。
- 在每个工作负载中利用 NIC 加速器和卸载功能。
- 更多信息请阅读英特尔白皮书“面向 NFV 解决方案的 SR-IOV”。更多详情请见附录表 7.5。

英特尔推出了 SR-IOV CNI 插件，支持 Kubernetes pod 在两种模式任意之一的条件下直接连接 SR-IOV 虚拟功能 (VF)。第一个模式在容器主机核心中使用标准 SR-IOV VF 驱动程序。第二个模式支持在用户空间执行 VF 驱动程序和网络协议的 DPDK VNF。

DPDK 支持该应用实现远超核心网络堆栈的数据包处理性能。如需了解性能指标评测结果，请参考“英特尔® 至强® 可扩展平台上的 Kubernetes 和容器裸机功能支持 NFV 用例”。附录中的参考文档表 7.5 提供了本文链接。

让我们仔细查看图 3，图中所示为具有三个网络接口的 pod 设置包括一个具有日志功能的容器化虚拟防火墙。在该图中，eth0 用作 pod 的管理接口，支持该 pod 与 Kubernetes 中的任何其他 pod 通信。eth0 是 Kubernetes 中的主网络接口。

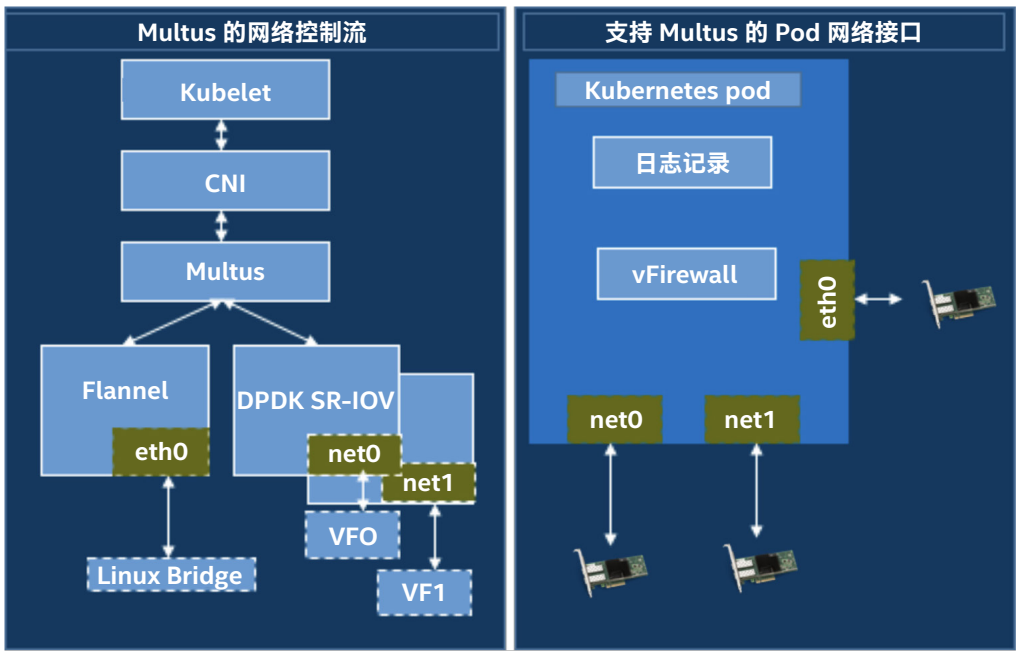

图 3.具有 SR-IOV/ DPDK CNI 的 Multus 网络

此外，图中还显示了两个 SR-IOV VF 接口：net0 和 net1。这些接口用于加速数据平面网络。例如，虚拟防火墙要求使用防火墙规则将两个网络进行相互隔离。VLAN 技术在虚拟防火墙与 802.1Q 交换机和路由器之间实施。该防火墙可识别 VLAN ID，并应用每个 VLAN 的特定防火墙规则，包括进行数据身份验证或应用数据平面网络中建立的相关策略。

该设置的优势包括：

- 网络的逻辑分割
- VLAN 标记的特定颗粒防火墙规则
- 更高的网络吞吐量和低延迟

下面两部分详细介绍了如何使用 SR-IOV/DPDK CNI 插件在 Kubernetes 中配置 Multus。

4.0 在 Kubernetes 中配置 Multus

Multus 提出了网络对象的概念。网络对象表示连接了 pod 接口的网络，它们是具有全局范围的网络的逻辑参考。网络对象存储在 Kubernetes 主注册表中，因而被视为集群级对象。

本部分介绍了在 Kubernetes 中选择网络的两个 Multus 配置选项：

- 使用网络对象和默认网络配置 Multus
- 使用配置文件配置 Multus

使用网络对象配置 Multus 可帮助用户按 pod、而非按节点选择网络。下面列举了用户可能希望设置的一些示例场景：

- Pod A 规格，网络对象注释“flannel”和“SRIOV”连接至 flannel 和 SR-IOV 网络。
- Pod B 规格，网络对象注释“Calico”连接至 Calico 网络。
- Pod C 规格，无任何网络对象注释，但“Weave”为默认网络，连接至 Weave 网络。

注：根据配置使用配置文件的 Multus 可用作任何其他 CNI 插件。

下一部分介绍了如何使用需调用第 3.0 部分（图 3）中三个网络接口的虚拟防火墙用例，借助网络对象配置 Multus，包括 Flannel、SR-IOV 和 SR-IOV（具有虚拟 LAN (VLAN) 标记）。

4.1 使用网络对象配置 Multus

自定义资源是 Kubernetes 中的扩展，可存储特定类型的一系列对象，如网络对象。自定义资源定义 (CRD) 是 Kubernetes 中的内置特性，为创建自定义资源提供了一种简单的方式。该机制为 Kubernetes API 服务器提供了一种描述新 API 实体的工具。从 Kubernetes 版本 1.7 开始，CRD 将取代第三方资源 (TPR)。CRD 提供了稳定的对象，推出了一些新特性，如资源名称的多元化和创建非命名空间 CRD 的能力。

在本部分，“网络”对象是一种自定义资源，使用 CRD 创建。有关使用 TPR 创建对象的更多信息，请查看附录中“如何定义基于 TPR 的网络对象”部分（在第 7.1 部分下方）。Multus 使用这些网络对象在 Kubernetes 中创建网络接口。

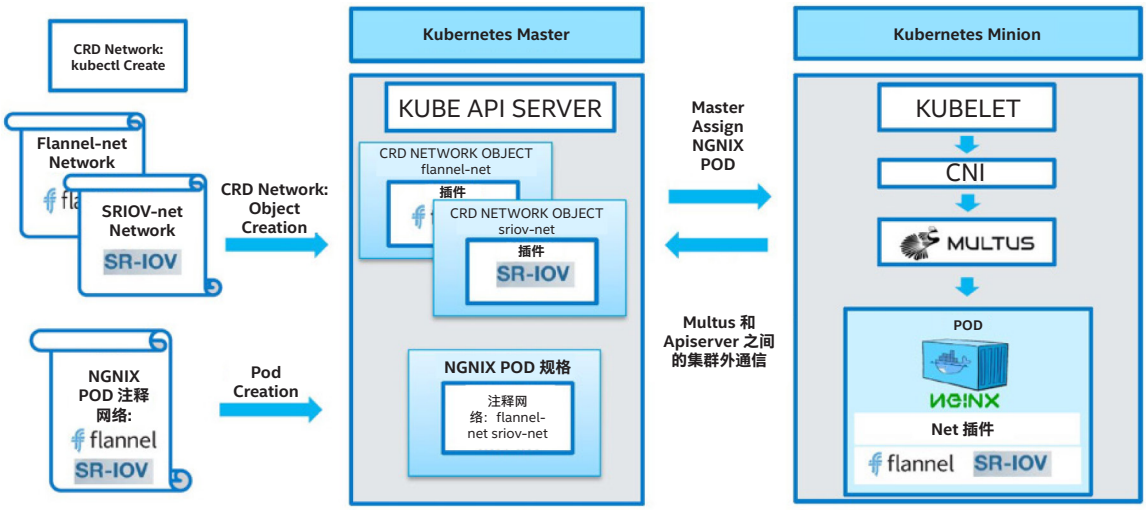

图 4.在 Kubernetes 中创建 Multus 网络接口所需操作的流程图

Kubelet 负责为每个 pod 建立网络接口；它通过调用配置的 CNI 插件来进行这一工作。Multus 被调用后，它恢复与 Multus 相关的 pod 注释，然后使用这些注释恢复 Kubernetes CRD 对象。Kubernetes CRD 对象向 Kubelet 提供应调用哪个插件及需传输配置等方面的信息。主插件的身份和插件调用的顺序非常重要。图 4 所示为在 Kubernetes 中创建 Multus 网络接口所需操作的流程图。

如前所述，Multus 兼容 CRD 和 TPR 扩展对象。然后，对 TPR 的支持最高只到 Kubernetes 版本 1.7。更高版本仅支持基于 CRD 的对象。因此，网络对象需使用 CRD 或 TPR 定义，具体取决于所使用的 Kubernetes 版本。然而，在基于 CRD/TPR 的网络对象中，客户端应用调用相同的 API 自链路（如 Kubelet）。这意味着开发人员或网络编排员可采用标准方式调用 API，无论是 CRD 还是 TPR 网络对象被调用。请参考附录中的第 7.1 部分，了解 TPR 的更多信息。

在设置多个网络接口时，用户需首先定义所需的网络对象，然后创建这些对象。下面描述了如何定义和创建 CRD 网络对象。

4.1.1 定义基于 CRD 的网络对象

1. 首先，创建CRD网络对象规格（请见以下步骤），然后将其保存为“crdnetwork.yaml”文件：

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below,
and be in the form: <plural>.<group>
  name: networks.kubernetes.com
spec:
  # group name to use for REST API: /
apis/<group>/<version>
  group: kubernetes.com
  # version name to use for REST API: /
apis/<group>/<version>
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /
apis/<group>/<version>/<plural>
    plural: networks
    # singular name to be used as an alias
on the CLI and for display
    singular: network
    # kind is normally the CamelCased
singular type. Your resource manifests use
this.
    kind: Network
    # shortNames allow shorter string to
match your resource on the CLI
    shortNames:
    - net
```

2. 然后运行下面的kubectl命令，以创建自定义资源定义：

```
# kubectl create -f ./crdnetwork.yaml
customresourcedefinition "network.kubernetes.
com" created
```

3. 运行下面的kubectl命令，确保网络CRD已创建完成：

```
# kubectl get CustomResourceDefinition
NAME                          KIND
networks.kubernetes.com
CustomResourceDefinition.v1beta1.
apiextensions.k8s.io
```

4. 然后，将下面的 YAML 保存至 flannel-network.yaml：

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: flannel-networkobj
plugin: flannel
args: '[
        {
                "delegate": {
                        "isDefaultGateway":
true
                }
        }
]'
```

5. 现在，创建自定义资源定义：

```
# kubectl create -f customCRD/flannel-network.
yaml
network "flannel-networkobj" created
# kubectl get network
NAME      KIND      ARGS    PLUGIN
flannel-networkobj    Network.v1.kubernetes.com
[ { "delegate": { "isDefaultGateway": true }
} ]    flannel
```

6. 然后，获取自定义网络对象详情：

```
# kubectl get network flannel-networkobj -o
yaml
apiVersion: kubernetes.com/v1
args: '[ { "delegate": { "isDefaultGateway":
true } } ]'
kind: Network
metadata:
  clusterName: ""
  creationTimestamp: 2017-07-11T21:46:52Z
  deletionGracePeriodSeconds: null
  deletionTimestamp: null
  name: flannel-networkobj
  namespace: default
  resourceVersion: "6848829"
  selfLink: /apis/kubernetes.com/v1/
namespaces/default/networks/flannel-networkobj
  uid: 7311c965-6682-11e7-b0b9-408d5c537d27
plugin: flannel
```

4.1.3 如何创建网络对象

在完成上一部分的步骤后，CRD 网络对象定义将添加至 API 服务器。现在可创建网络对象。网络对象应包含 JSON 格式的网络参数。在下面的示例中，插件和参数字段设置为网络对象。网络对象衍生自前述步骤创建的 CRD 对象的 metadata.name。

1. 首先，将下面的YAML保存至文件flannel-network.yaml：

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: flannel-conf
plugin: flannel
args: '[
        {
                "delegate": {
                        "isDefaultGateway":
true
                }
        }
]'
```

2. 现在，运行kubectlcreate命令，以创建flannel-conf网络对象：

```
# kubectl create -f ./flannel-network.yaml
network "flannel-conf" created
```

3. 然后，使用 kubectl 验证网络对象：

```
# kubectl get network
NAME                          KIND
flannel-conf          Network.v1.kubernetes.com
```

4. 用户还可查看网络对象的原始JSON数据。后面的说明显示了 yaml 文件的自定义插件和参数字段，该文件被用于创建网络对象：

```
# kubectl get network flannel-conf -o yaml
apiVersion: kubernetes.com/v1
args: '[ { "delegate": { "isDefaultGateway":
true } } ]'
kind: Network
metadata:
  creationTimestamp: 2017-06-28T14:20:52Z
  name: flannel-conf
  namespace: default
  resourceVersion: "5422876"
  selfLink: /apis/kubernetes.com/v1/
namespaces/default/networks/flannel-conf
  uid: fdcb94a2-5c0c-11e7-bbeb-408d5c537d27
plugin: flannel
```

5. 插件字段应具有 CNI 插件的名称，参数应具有 Flannel 参数，它应该为上述 JSON 格式。Calico、Weave、Romana 和 Cilium 的网络对象也可采用相似方式创建。

6. 将下列 YAML 保存至 sriov-network.yaml：

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: sriov-conf
plugin: sriov
args: '[
        {
                "if0": "enp12s0f1",
                "ipam": {
                    "type": "host-local",
                    "subnet": "10.56.217.0/24",
                    "rangeStart": "10.56.217.171",
                    "rangeEnd": "10.56.217.181",
                    "routes": [
                        { "dst": "0.0.0.0/0" }
                    ],
                    "gateway": "10.56.217.1"
                }
        }
]'
```

7. 然后，将下面的 YAML 保存至文件 sriov-vlanid-l2enable-network.yaml：

```
apiVersion: "kubernetes.com/v1"
kind: Network
metadata:
  name: sriov-vlanid-l2enable-conf
plugin: sriov
args: '[
       {
                "if0": "enp2s0",
                "vlan": 210,
                "if0name": "north",
                "l2enable": true
        }
]'
```

8. 完成下列两步，创建网络对象“sriov-vlanid-l2enable-conf”和“sriov-conf”：

```
# kubectl create -f ./sriov-vlanid-l2enable-
network.yaml
network "sriov-vlanid-l2enable-conf" created

# kubectl create -f ./sriov-network.yaml
network "sriov-conf" created
```

9. 最后，使用 kubectl 验证网络对象：

```
# kubectl get network
NAME                            KIND
flannel-conf          Network.v1.kubernetes.com
sriov-vlanid-l2enable-conf
Network.v1.kubernetes.com
sriov-conf            Network.v1.kubernetes.com

# systemctl restart kubelet
```

此时，网络资源已配置和创建完成。下一步是使用这些网络资源部署 pod。

4.1.4 为 pod 部署多个接口

1. 使用下列内容创建示例pod规格pod-multi-network.yaml 文件。在这一情况下，flannel-conf 网络对象用作主网络：

```
# cat pod-multi-network.yaml

apiVersion: v1
kind: Pod
metadata:
  name: multus-multi-net-pod
  annotations:
    networks: '[
        { "name": "flannel-conf" },
        { "name": "sriov-conf"},
     { "name": "sriov-vlanid-l2enable-conf" }
    ]'
spec:  # specification of the pod's contents
  containers:
  - name: multus-multi-net-pod
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
```

2. 下面，通过主节点创建多个基于网络的 pod：

```
# kubectl create -f ./pod-multi-network.yaml
pod "multus-multi-net-pod" created
```

3. 最后，从主节点中检索运行中 pod 的详情：

```
# kubectl get pods
NAME                  READY
multus-multi-net-pod  1/1
STATUS    RESTARTS    AGE
Running   0           30s
```

4.2 使用配置文件配置 Multus

通过特定配置，Multus 可从其配置文件而非网络对象读取网络配置。在这一情况下，节点中的所有 pod 将具有相同的网络接口。

4.2.1 Multus 配置参数

Multus 接受 JSON 格式的配置参数，如表 1 所述。一些参数必不可少，因此用户在配置 Kubernetes 从节点时需要提供相关参数。

表 1. Multus 配置参数

参数名称	类型	必选	说明
名称	字符串	是	网络名称
类型	字符串	是	“multus”
kubeconfig	字符串	否	kubeconfig 文件用于与 kube-apiserver 的集群外通信
代理	映射	是	代理对象（底层 CNI 插件配置）。添加 kubeconfig 时这点通常被忽略
主插件	bool	是	主插件将 IP 地址和 DNS 报告回容器

下面说明了如何配置 Multus:

4.2.2 使用 CNI 配置文件进行配置

1. 借助从节点的下列内容，创建MultusCNI配置文件/etc/cni/net.d/ multus-cni.conf。仅使用绝对路径指向 kubeconfig 文件（可能有所不同，具体取决于 Kubernetes 集群环境）。假设所有 CNI 二进制文件位于 \opt\ cni\bin 目录下（默认位置）：

```
{
    "name": "minion-cni-network",
    "type": "multus",
    "kubeconfig": "/etc/kubernetes/node-
     kubeconfig.yaml"
}
```

2. 现在，重启 kubelet 服务：

```
 # systemctl restart kubelet
```

4.2.3 对默认网络使用 kubeconfig 进行配置

3. 如果pod规格不包括某些网络对象，自动Kubernetes部署模式或包装程序需要默认的网络特性，如第三方为非 Multus 部署开发的自动化 Ansible 脚本和包装程序。所有 Ansible 脚本应利用 Weave 或其他默认网络功能，与 Multus 协同运行。再比如，在下面的配置中，当 pod 元数据注释不包括网络字段时，Weave 用作默认网络。

```
{
    "name": "minion-cni-network",
    "type": "multus",
    "kubeconfig": "/etc/kubernetes/node-
kubeconfig.yaml",
    "delegates": [{
        "type": "weave-net",
        "hairpinMode": true,
        "masterplugin": true
    }]
}
```

4. 现在，重启 kubelet 服务：

```
 # systemctl restart kubelet
```

4.3 验证 pod 网络

完成配置后，您可检查 pod 网络运行是否符合预期。使用第 4.1.3 部分中规格创建的 pod，应使用提供的配置创建三个接口。为验证这些配置，您需完成以下步骤：

1. 在容器内运行“ifconfig”命令：

```
# kubectl exec -it multus-multi-net-poc -
ifconfig

eth0  Link encap:Ethernet  HWaddr
      06:21:91:2D:74:B9
  inet addr:192.168.42.3  Bcast:0.0.0.0
  Mask:255.255.255.0
  inet6 addr: fe80::421:91ff:fe2d:74b9/64
  Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1450
  Metric:1
  RX packets:0 errors:0 dropped:0
  overruns:0 frame:0
  TX packets:8 errors:0 dropped:0
  overruns:0 carrier:0
  collisions:0 txqueuelen:0
  RX bytes:0 (0.0 B)  TX bytes:648 (648.0 B)

lo    Link encap:Local Loopback
  inet addr:127.0.0.1  Mask:255.0.0.0
  inet6 addr: ::1/128 Scope:Host
  UP LOOPBACK RUNNING  MTU:65536  Metric:1
  RX packets:0 errors:0 dropped:0 overruns:0
  frame:0
  TX packets:0 errors:0 dropped:0 overruns:0
  carrier:0
  collisions:0 txqueuelen:1
  RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

net0  Link encap:Ethernet
     HWaddrD2:94:98:82:00:00
  inet addr:10.56.217.171  Bcast:0.0.0.0
  Mask:255.255.255.0
  inet6 addr: fe80::d094:98ff:fe82:0/64
  Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1500
  Metric:1
  RX packets:2 errors:0 dropped:0 overruns:0
  frame:0
  TX packets:8 errors:0 dropped:0 overruns:0
  carrier:0
  collisions:0 txqueuelen:1000
  RX bytes:120(120.0 B) TX bytes:648(648.0 B)

north  Link encap:Ethernet  HWaddr
       BE:F2:48:42:83:12
  inet6 addr: fe80::bcf2:48ff:fe42:8312/64
  Scope:Link
  UP BROADCAST RUNNING MULTICAST  MTU:1500
  Metric:1
  RX packets:1420 errors:0 dropped:0
  overruns:0 frame:0
  TX packets:1276 errors:0 dropped:0
  overruns:0 carrier:0 collisions:0
  txqueuelen:1000
  RX bytes:95956 (93.7 KiB)  TX bytes:82200
  (80.2 KiB)
```

如上面的输出屏幕所示，三个接口及一个回路接口已创建完成。

有关 pod 接口的说明，请查看下表：

接口名称	说明
lo	回路
eth0@if41	Flannel 网络分流器 (network tap) 接口
net0	SR-IOV CNI 插件将 NIC 1 的 VF0 分配至容器
north	SR-IOV CNI 插件将 NIC 2 的 VF0（带有 VLAN ID 210）分配至容器

2. 用户可检查分配自 SR-IOV NIC 的 VF VLAN ID，是否与第 4.1.2 部分步骤 8 中 sriov-vlanid-l2enable-network.yaml 文件的 VLAN 标记相匹配。用户可使用如下所示的 IPROUTE2 实用程序，检查 SR-IOV NIC 信息，以进一步确认：

```
# ip link show enp2s0
20: enp2s0: <BROADCAST,MULTICAST,UP,LOWER _ UP>
  mtu 1500 qdisc mq state UP mode DEFAULT
  group default qlen 1000
  link/ether 24:8a:07:e8:7d:40 brd
  ff:ff:ff:ff:ff:ff
  vf 0 MAC 00:00:00:00:00:00, vlan 210, spoof
  checking off, link-state auto
  vf 1 MAC 00:00:00:00:00:00, vlan 4095, spoof
  checking off, link-state auto
  vf 2 MAC 00:00:00:00:00:00, vlan 4095, spoof
  checking off, link-state auto
  vf 3 MAC 00:00:00:00:00:00, vlan 4095, spoof
  checking off, link-state auto
```

在完成上述步骤后，Kubernetes pod 应配置三个网络接口："eth0"、"net0"和"north"，以及回路接口"lo"。在上述说明中，Kubernetes 多网络通过 Multus CNI 插件创建网络对象。请点击 GitHub 链接，了解更多信息：https://github.com/Intel-Corp/multus-cni/

5.0 在 Kubernetes 中配置 SR-IOV/DPDK

SR-IOV 支持的 NIC 支持在许多虚拟环境的许多 VNF 中透明共享物理 NIC 端口。每个 VF 可分配至一个容器，并配置单独的 MAC、VLAN 和 IP。SR-IOV CNI 插件支持 Kubernetes pod 连接至 SR-IOV VF。该插件会查找 Multus 配置文件中指定端口上的首个可用 VF。该插件还支持 DPDK 驱动程序（即 vfio-pci）用于这些 VF。DPDK 驱动程序可为 Kubernetes pod 提供高性能网络接口，以帮助容器化 VNF 加速数据平面。

5.1 启用 SR-IOV

以下步骤可在 CentOS、Fedora 或 RHEL 上启用 SR-IOV 插件，以支持英特尔 ixgbe NIC。

1. 首先，使用以下命令启用 SR-IOV：

```
# vi /etc/modprobe.conf
options ixgbe max _ vfs=8,8
```

2. 然后，为 SR-IOV 插件提供更多网络配置参数（请见第 4.1 部分的步骤 7）。sriov-network.yaml 文件中的**"参数"**字段保持 SR-IOV 配置参数值。参数说明请见下：

- **name**（字符串，必选项）：网络名称
- **type**（字符串，必选项）："sriov"
- **if0**（字符串，必选项）：PF 的名称
- **L2enable**（字符串，可选项）：没有（字符串，可选项）：没有 IP 地址的 SR-IOV 接口
- **ipam**（字典，需用于核心模式）：IPAM 配置需用于该网络（核心模式）。更多信息请参考 IPAM。
- **dpdk**（字典，仅需用于用户空间）
  - **kernel_driver**（字符串，需用于 DPDK 模式）：NIC 驱动程序的名称，如 i40evf
  - **dpdk_driver**（字符串，需用于 DPDK 模式）：DPDK 驱动程序的名称，如 vfio-pci
  - **dpdk_tool**（字符串，需用于 DPDK 模式）：dpdk 绑定脚本的绝对路径，如 dpdk-devbind.py Extra 参数
  - **vf**（int，可选项）：VF 索引，默认值为 0
  - **vlan**（int，可选项）：核心模式中用于 VF 设备使用的 VLAN ID，使用 IPAM

3. 在设置完这些参数后，使用SR-IOV插件创建网络对象，请参见第 4.1 部分的步骤 8。

SR-IOV 插件可与 DPDK 或核心一起使用。默认情况下，插件在核心模式下运行。如需为 DPDK 模式下运行，需设置以下网络参数：

- dpdk（字典，仅需用于用户空间）
  - kernel_driver（字符串，需用于 DPDK 模式）：NIC 驱动程序的名称，如 i40evf
  - dpdk_driver（字符串，需用于 DPDK 模式）：DPDK 驱动程序的名称，如 vfio-pci

第 5.1.1 和 5.1.2 部分介绍了这两种模式。

5.1.1 核心模式：

SR-IOV CNI 插件将 SR-IOV VF 接口从主机网络命名空间绑定至容器网络命名空间，并将 IPAM 信息分配至 SR-IOV VF 接口。

5.1.2 DPDK 模式:

SR-IOV CNI 插件将 SR-IOV VF 接口绑定至 DPDK 用户空间。在这一过程中，PCI 地址存储在主机内，且插件会持续实施无状态工作。在删除过程中，SR-IOV CNI 插件会检索 PCI 地址，解除 SR-IOV VF 接口从 DPDK 用户空间到核心空间的绑定。

5.1.3 为 Multus 配置 Flannel 和 DPDK –SR-IOV CNI 插件

如需在 Kubernetes 节点中使用以下内容创建 Multus CNI 配置文件 /etc/cni/net.d/ multus-cni.conf，请确保 DPDK 驱动程序已加载且 SR-IOV VF 已创建。在设置 dpdk_tool 选项时，务必使用绝对路径。

1. 按照这些步骤开始配置:

```
{
   "name": "minion1-multus-demo-network",
   "type": "multus",
   "delegates": [
       {
           "type": "sriov",
           "if0": "enp4s0f3",
           "if0name": "north0",
           "dpdk": {
               "kernel _ driver":"ixgbevf",
               "dpdk _ driver":"igb _ uio",
               "dpdk _ tool":"/root/dpdk/
                tools/dpdk-devbind.py"
           }
       },
       {
           "type": "flannel",
           "masterplugin": true,
           "delegate": {
               "isDefaultGateway": true
           }
       }
    ]
}
```

上文提供了一个 Multus CNI 插件配置文件示例，其同时调用 flannel 和 DPDK-SR-IOV CNI 插件，以提供实际的 CNI 网络功能。

6.0 结论

在 Kubernetes pod 中配置多个网络接口是许多 VNF 应用的基本特性。如今，通过使用 Multus CNI 及本文描述的其他开源软件组件，便可实现该目的。此外，多个网络接口有助于使用面向 SR-IOV 和 DPDK 的接口，为基于容器的应用改进网络吞吐量。为推动虚拟化计算的发展，英特尔致力于开发这些技术。

如欲更详细地了解英特尔在容器领域的进展，请访问https://networkbuilders.intel.com/network-technologies/intel-container-experience-kits。

7.0 附录

附录包括：

- 如何定义基于 TPR 的网络对象的信息
- 包含本文所引用硬件平台和软件的表格。
- 本文所用首字母缩略词概要
- 参考文档链接

7.1 如何定义基于 TPR 的网络对象

1. 首先为网络对象创建第三方资源“tprnetwork.yaml”，如下所示：

```
apiVersion: extensions/v1beta1
kind: ThirdPartyResource
metadata:
  name: network.kubernetes.com
description: "A specification of a Network
obj in the kubernetes"
versions:
- name: v1
```

2. 然后，为第三方资源运行 kubectl create 命令：

```
# kubectl create -f ./tprnetwork.yaml
thirdpartyresource "network.kubernetes.com"
created
```

3. 接着运行 kubectl get 命令，检查网络 TPR 创建：

```
# kubectl get thirdpartyresource
NAME                          DESCRIPTION
network.kubernetes.com    A specification of a
Network obj in the kubernetes
VERSION(S)
v1
```

7.2 硬件

表 7.2 性能测试中使用的硬件组件

项目	说明	注释
平台	英特尔® 服务器主板 S2600WFQ	基于两颗英特尔® 至强® 处理器的服务器主板，具有 2 个 10 GbE 集成式 LAN 端口
处理器	两颗英特尔® 至强® 金牌处理器 6138T	2.0 GHz；125 W；27.5 MB 高速缓存/处理器 20 个内核，40 个超线程内核/处理器
内存	共 192GB；Micron MTA36ASF2G72PZ	12x16GB DDR4 2133MHz 每通道 16GB，每插槽 6 个通道
NIC	英特尔® Ethernet Network Adapter XXV710-DA2 (2x25G)	2 个 1/10/25 GbE 端口，固件版本 5.50
存储	英特尔® DC P3700 SSDPE2MD800G4	SSDPE2MD800G4 800 GB 固态盘 2.5 英寸 NVMe/PCIe
BIOS	英特尔公司 SE5C620.86B0X.01.0007.060920171037 发布日期：2017 年 6 月 9 日	超线程 – 启用 启动性能模式 – 最大性能 节能的睿频加速技术 - 禁用 睿频模式 - 禁用 C State – 禁用 P State – 禁用 英特尔 VT – x 启用 英特尔 VT – d 启用

7.3 软件

表 7.3 性能测试中使用的软件组件

软件组件	说明	参考资料
主机操作系统	Ubuntu 16.04.2 x86_64（服务器） 核心：4.4.0-62-generic	https://www.ubuntu.com/download/server
NIC 核心驱动程序	i40e v2.0.30 i40evf v2.0.30	https://sourceforge.net/projects/e1000/files/i40e%20stable
DPDK	DPDK 17.05（软件下载）	http://fast.dpdk.org/rel/dpdk-17.05.tar.xz
CMK	v1.1.0 和 v1.2.1	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
Ansible	Ansible 2.3.1.0	https://github.com/ansible/ansible/releases
裸机容器设置脚本	包括用于部署 Kubernetes 的 Ansible* 脚本 v1.6.6 和 1.8.4	https://github.com/intel/container-experience-kits
Docker	v1.13.1	https://docs.docker.com/engine/installation/
SR-IOV-CNI	v0.2-alpha. commit ID: a2b6a7e03d8da456f3848a96c6832e6aefc968a6	https://www.ubuntu.com/download/server

7.4 术语

表 7.4 术语

条款	说明
CNCF	云原生计算基金会
CNI	容器网络接口
CNVNF	云原生虚拟化网络功能
COE	容器编排引擎
CommSP	通信服务提供商
CRD	自定义资源定义
DPDK	数据平面开发套件
IPAM	IP 地址管理
IPSec	面向 IP 网络的加密协议
LF	Linux 基金会
NFD	节点特性发现
NFV	网络功能虚拟化
NIC	网络接口卡
PCIe	Peripheral Component Interconnect Express
PF	物理功能
Pod	Kubernetes 中的一个或多个容器
rkt	CoreOS Rocket
SDN	软件定义网络
SLA	服务级别协议
SR-IOV	单根输入/输出虚拟化
STDIN	标准输入
TPR	第三方资源
VETH	虚拟以太网接口
VF	虚拟功能
VLAN	虚拟局域网
VM	虚拟机
VNF	虚拟网络功能
VPP	Vector Packet Processing

7.5 参考文档

表 7.5 参考文档

文档	文档编号/位置
借助英特尔® 至强® 可扩展处理器为 NFV 用例部署 Kubernetes 和容器裸机平台	https://networkbuilders.intel.com/network-technologies/container-experience-kits
英特尔® 至强® 可扩展平台上的 Kubernetes 和容器裸机功能支持 NFV 用例	https://networkbuilders.intel.com/network-technologies/container-experience-kits
启用 Kubernetes 新特性以支持 NFV	https://networkbuilders.intel.com/network-technologies/container-experience-kits
面向容器化 vEPC 应用的 NFV 参考设计	https://builders.intel.com/docs/networkbuilders/nfv-reference-design-for-a-containerized-vepc-application.pdf
了解 CNI（容器网络接口）	http://www.dasblinkenlichten.com/understanding-cni-container-networking-interface/
面向 NFV 解决方案的 SR-IOV（PDF 下载）	https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/sr-iov-nfv-tech-brief.pdf