边缘云原生虚拟化研究报告

2024 年 1 月

可信边缘计算推进计划

目　　次

前　言

参与单位（排名不分先后）：中国联合网络通信有限公司研究院，中国联合网络通信有限公司智网创新中心、可信边缘计算推进计划

编写人：黄蓉 庞博 黄倩 陈丹 肖羽 蔡超 侯迎龙 隗英英 高沛 李晓旭 隋佳良 王蕴婷 李昂

1 技术与需求概述

随着网络技术和云技术的发展，边缘计算得到了广泛的应用。边缘计算可以解决高可靠低延迟的设备接入和海量数据的实时计算问题，云技术有力的保障和推动了边缘计算的应用。

1.1 虚拟机和容器

虚拟机和容器是云计算中最常用到的应用部署和运行方式。虚拟机是伴随着虚拟化的技术出现的，容器则云原生技术的典型特征之一，他们的架构对比如下图所示：

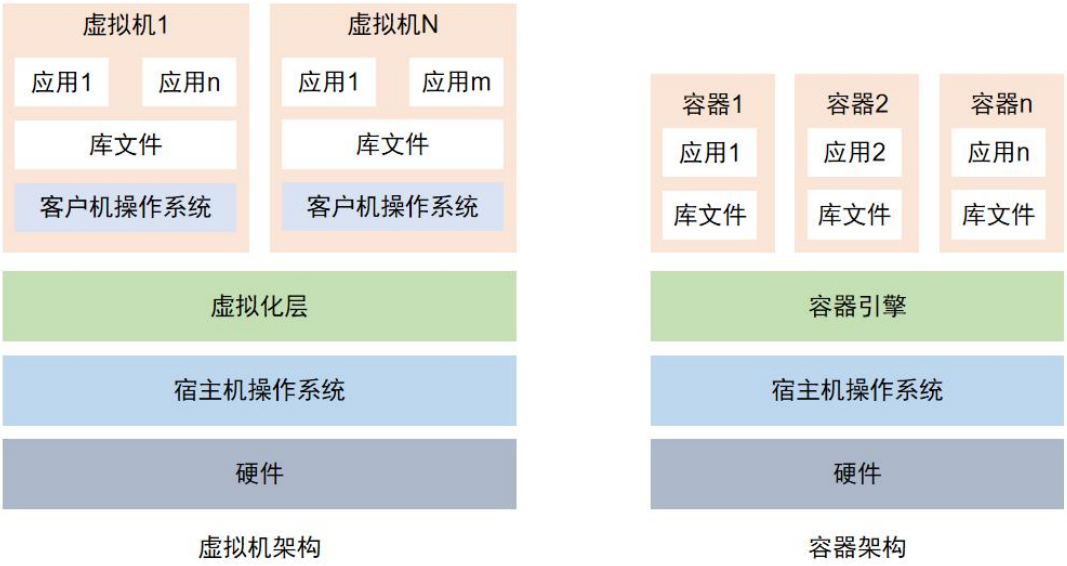


图 1：虚拟机与容器架构对比图

如上图所示，虚拟化技术一般通过虚拟化层（hypervisor）来实现，通过虚拟化技术，虚拟机可以共享物理机的 CPU、内存、IO 等硬件资源，并在逻辑上实现相互隔离。每一个虚拟机都拥有独立的操作系统（客户机操作系统），所以虚拟机不依赖于宿主机操作系统，其安全性和隔离性更强。但是虚拟机占用的资源较多，而且由于要模拟硬件，虚拟化层本身也会占用部分资源，对宿主机性能有一定的消耗。比较而言，容器则是使用宿主机的内核系统加上自身的文件系统。运行容器时是在使用宿主机的内核情况下加载文件系统，一般可以将容器看作是在内核上运行的独立进程。而精简的文件系统可以小到 100MB 以内，因此容器比虚拟机占用资源的更少、启动速度更快。容器缺点是隔离性不如虚拟机，而且由于依赖宿主机内核，所以容器的操作系统选择一般会受到限制。

两种技术的特点对比如下表：

表 1：虚拟机与容器技术特点对比

对比项	虚拟机技术	容器技术
安全隔离性	强，操作系统级别	弱，进程级别
对宿主机操作系统依赖	无	有，需要相同操作系统内核
启动时间	慢，分钟级	快，秒级
磁盘占用	大（GB）	小（MB）
虚拟化性能损耗	大[1]	小

1.2 OpenStack 与 Kubernetes

从运行和管理平台来看，OpenStack[2] 与 Kubernetes（K8s）[3] 分别是对虚拟机和容器进行运行和管理的典型开源项目。

OpenStack 是开源的云计算平台，利用虚拟化技术和底层存储服务，提供了可扩展、灵活、适应性

强的云计算服务。OpenStack 的服务分为核心功能和非核心功能。核心功能是指运行 OpenStack 系统必须的功能的组件，包括：Keystone（身份识别服务）、Glance（镜像服务）、Nova（计算机服务）、Neutron（网络服务）、Cinder（块存储服务）、Swift（对象存储服务）、Horizon（控制面板服务）。非核心功能指的是实现附加功能的组件，如 Ceilometer（测量功能）、Heat（部署编排）、Trove（数据库服务）等。OpenStack 的各个组件（服务）之间使用标准的 API 接口调用，减少了服务之间的依赖。

下图是 OpenStack 的逻辑架构图。

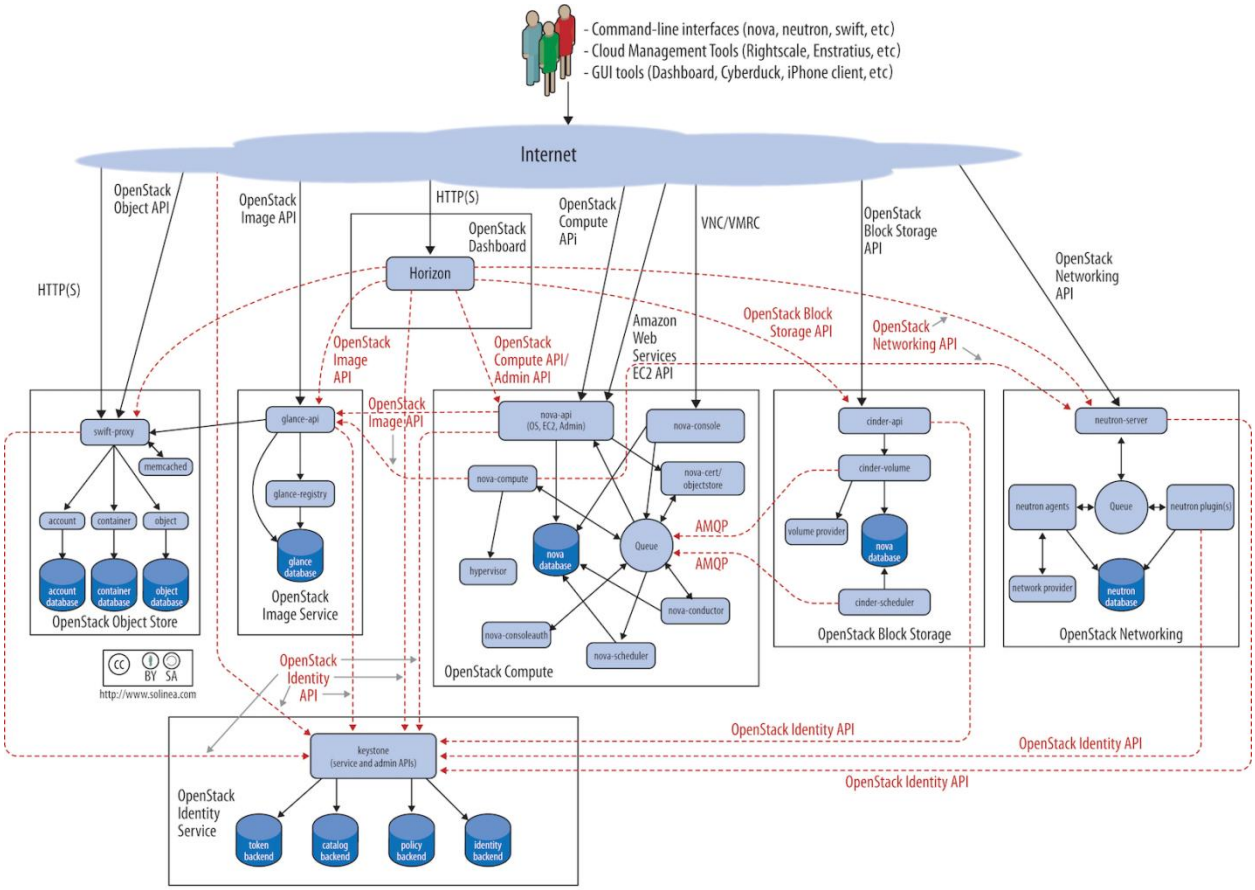


图 2：OpenStack 逻辑架构图

Kubernetes 是容器管理编排引擎，可以自动完成容器的部署、管理和扩展等操作，部署 Kubernetes 的设备环境通常被成为 Kubernetes 集群。Kubernetes 集群逻辑上可以分为两个部分：控制平面与计算设备（或称为节点）。控制平面的包含：kube-apiserver（接口程序，用于处理内部和外部请求）、kube-scheduler（调度程序）、kube-controller-manager（集群控制管理程序）、etcd（数据库）。计算设备包含容器运行时引擎、kubelet（节点代理程序）、kube-proxy（网络代理程序）。Kubernetes 的设计原则包括：安全、易于使用和可扩展。Kubernetes 同样遵循标准化 API 接口，而且 Kubernetes 实现了 CNI（容器网络接口）、CRI（容器运行时接口）、CSI（容器存储接口）等接口和 CRD（用户自定义资源）等，便于实现功能的扩展。

下图是 Kubernetes 的逻辑架构图。

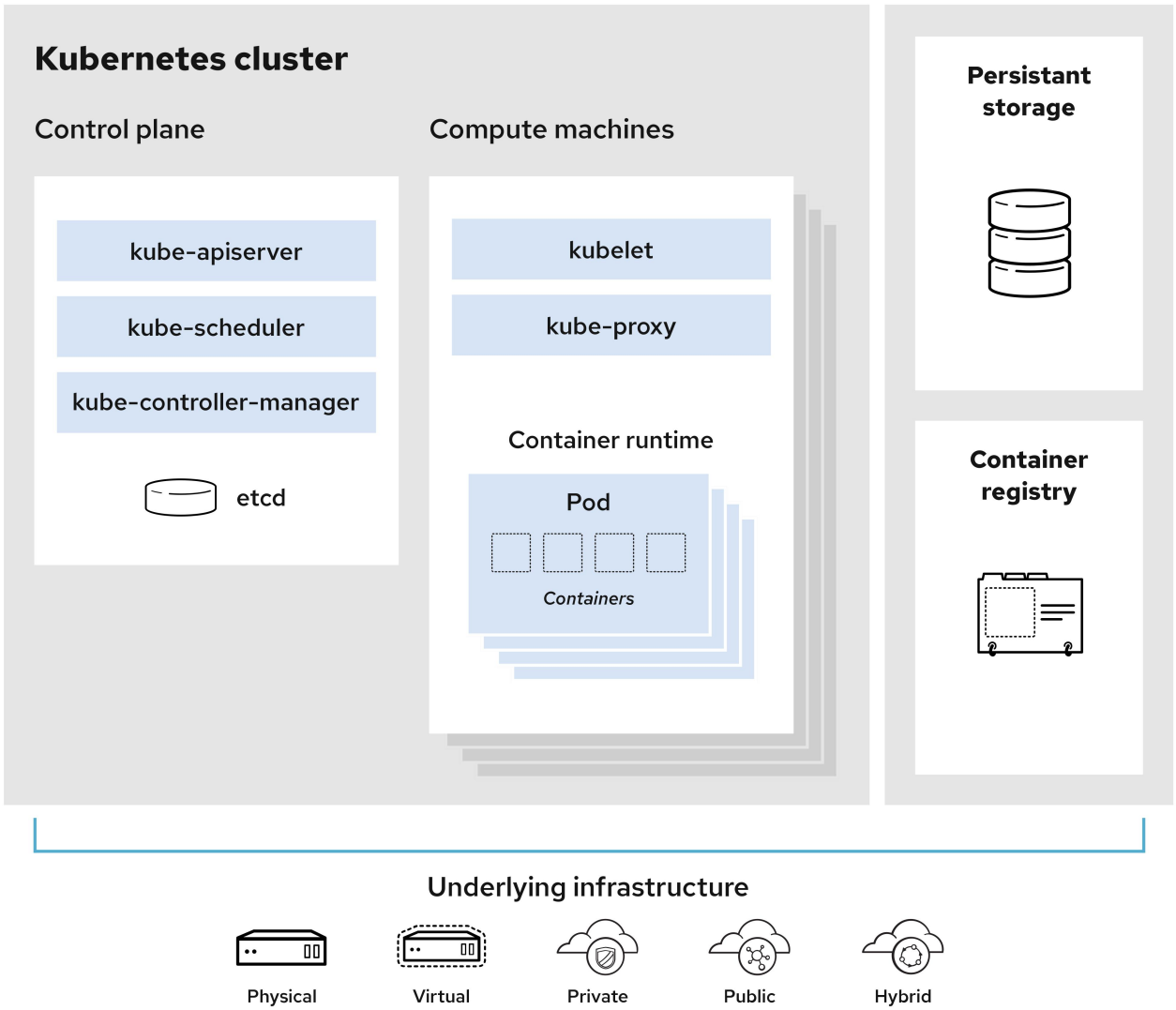


图 3：Kubernetes 逻辑架构图

OpenStack 的设计比较全面，组件众多，部署相对复杂，难于运维，使用成本较高，更适合作为大规模云的管理系统。相对而言，Kubernetes 的设计更加简洁，其核心组件少，便于运维，同时 K8s 的生态很庞大，可以很方便地对其进行扩展或者定制，更适用于资源受限的边缘环境。

1.3 融合管理的演进：K8s 环境下运行虚拟机

当前，通过容器部署的应用越来越广泛。但是，通过虚拟机部署的应用也会存在相当长的时间。首先，有不少现存的应用程序是运行在虚拟机上的，其中一些程序无法轻松地进行容器化重构。其次，即便对程序进行容器化改造，之后的系统调试和问题定位又会带来很大的挑战，尤其是对于通信行业来说，多代通信设备并存，对设备和应用程序的稳定性要求又非常高，对原有的应用程序进行容器化改造的成本和风险都是较大的。最后，一些应用或者场景更加适合使用虚拟机来进行部署。比如下面这些场景更适合使用虚拟机来运行而不是容器：

* NFV（network function virtualization）网络功能虚拟化的场景：将传统的网元虚拟化，使用虚拟机会比使用容器更方便，因为容器在接入多网卡方面比起虚拟机的能力来说还有一定的差距；
* 大模型的研发测试：大模型在研发测试阶段进场需要使用多张GPU协同配合，同时要安装很多多软件依赖包来进行调试和使用，这时直接将多张GPU挂载到一个虚拟机里，然后在虚拟机里来实验开发要比在容器里方便很多；
* 数据库：不是所有的数据库都适合放在容器里运行，比如部分数据库的特定算法需要限制IP的变化，在虚拟机里部署可以有一个固定的IP，会更加方便；

* 很多进程的应用：在容器使用上，有个核心概念就是部署任务单一的进程，比如一个简单的api服务进程加一个日志收集的进程组合成为了一个容器，有些多进程的应用就不适合放在容器中运行了。

于是，随着时间推移，企业会遇到这样的情况，有些应用还是只能运行在虚拟机上，有些应用已经完成了容器化，企业管理员不得不同时运维管理多套平台，这大大增加了运维的难度：

* 计算资源：从管理角度来说计算资源的管理不同的平台的管理方法也是截然不同的，比如OpenStack是通过project quota来管理，而K8s则通过request/limit来管理，管理人员必须完全了解2套机制才能完全很好的管理起来；

* 网络资源：同样，对于网络管理来说，不同的平台也是完全不同的， K8s使用CNI来管理网络，同时OpenStack通过neutron来管理网络，他们的使用理念上也是截然不同的，这样很大程度上增加了学习成本；

* 监控/日志：各种平台都有自己的完整的监控/日志收集系统，它们会占用大量的计算、存储和网络资源，同时部署这样2套平台，从资源使用的角度上来说也是一种很大的浪费；

* 存储资源：相同的存储资源对接K8s和OpenStack方式都是截然不同的，出现问题后找根因的思路和角度也都是不一样的，这样也大大加大了运维的成本和难度。

* 安全风险：软件是由不同工程师编写的代码，运行在不同的操作系统上，每个环境都会遇到安全漏洞，越多组件则面临更多的安全漏洞，同时运维2套平台就意味着面临安全漏洞也会更多，企业面临的安全风险也就更大。

从各个方面来看，企业内部的虚拟机平台和容器平台合并成为同一套平台来进行管理是一个趋势。那么是将K8s合并到OpenStack呢？还是反过来呢？

业内也在研究虚拟机和容器的共平台的部署和管理，从OpenStack和K8s各自的发展来看，两个平台也在进行虚拟机和容器共同管理的探索，比如OpenStack的zun服务将容器作为一种OpenStack资源来进行管理，并通过集成OpenStack的其他服务，为用户呈现统一的、简化的API接口，用户可以通过这些接口来创建、管理容器；K8s也有多个相关的开源项目在研究如何实现对虚拟机的管理（见下文）。

从云技术的现状和发展来看，容器的应用越来越广泛，而且K8s在容器编排领域成为了业内事实上的标准。在边缘环境下，K8s的适用范围也更加广泛，因此，本文将进一步探讨在K8s环境下运行虚拟机的技术和实践范例。

1.4 开源项目简介

本节介绍在K8s环境下运行虚拟机的相关开源项目。当前这些项目还在发展之中，功能还在不断地迭代和完善。

1.4.1 KubeVirt

KubeVirt[4]是一个K8s插件，由Redhat开源，云原生计算基金会（CNCF）赞助的开源项目。KubeVirt插件可以在K8s平台上调度传统的虚拟机。KubeVirt使用自定义资源（CRD）将VM管理接口接入到K8s中，通过一个Pod去使用libvirtd管理VM的方式，实现Pod与VM的一一对应，做到如同容器一般去管理虚拟机，并且做到与容器一样的资源管理、调度规划。KubeVirt的架构图如下。

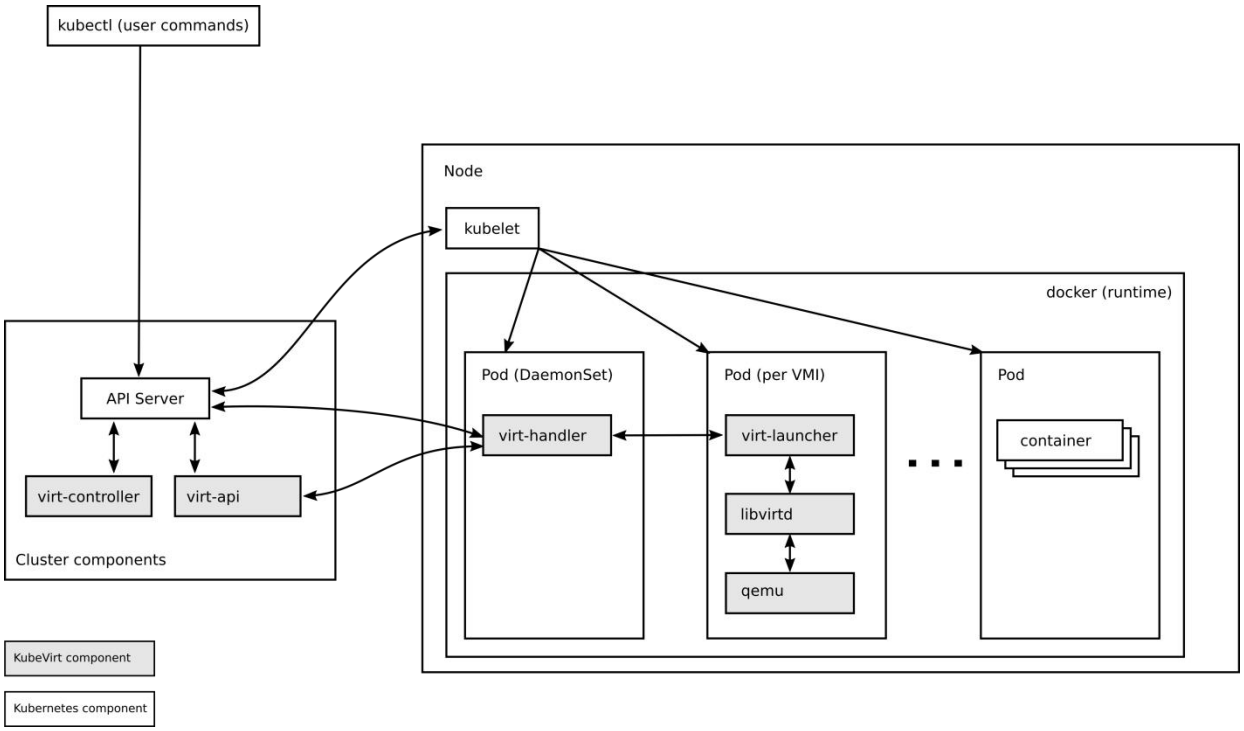


图4：KubeVirt架构图

KubeVirt主要由下列五个部分组成：

virt-api：kubevirt是以CRD形式去管理VM Pod，virt-api就是所有虚拟化操作的入口，这里面包括常规的CDR更新验证、以及console、vm start、stop等操作。

virt-controller：virt-controller会根据VMI CRD，生成对应的virt-launcher Pod，并且维护CRD的状态。与K8s的api-server通讯监控VMI资源的创建删除等状态。

virt-handler：virt-handler会以deamonset形式部署在每一个节点上，负责监控节点上的每个虚拟机实例状态变化，一旦检测到状态的变化，会进行响应并且确保相应的操作能够达到所需（理想）的状态。virt-handler还会保持集群级别VMI Spec与相应libvirt域之间的同步；报告libvirt域状态和集群Spec的变化；调用以节点为中心的插件以满足VMI Spec定义的网络和存储要求。

virt-launcher：每个virt-launcher Pod对应着一个VMI，kubelet只负责virt-launcher Pod运行状态，不会去关心VMI创建情况。virt-handler会根据CRD参数配置去通知virt-launcher去使用本地的libvirtd实例来启动VMI，随着Pod的生命周期结束，virt-lanuncher也会去通知VMI去执行终止操作；其次在每个virt-launcher Pod中还对应着一个libvirtd，virt-launcher通过libvirtd去管理VM的生命周期，不再是以前的虚拟机架构那样一个libvirtd去管理多个VM。

virtctl：virtctl是kubevirt自带类似kubectl的命令行工具，它是越过virt-launcher Pod这一层去直接管理VM虚拟机，可以控制VM的start、stop、restart。

KubeVirt利用CRD的功能定义了若干种资源对象。

VirtualMachineInstance（VMI）：类似于 Kubernetes Pod，是管理虚拟机的最小资源。一个 VirtualMachineInstance 对象即表示一台正在运行的虚拟机实例，包含一个虚拟机所需要的各种配置。

VirtualMachine（VM）：为群集内的 VirtualMachineInstance 提供管理功能，例如开机/关机/重启虚拟机，确保虚拟机实例的启动状态，与虚拟机实例是 1:1 的关系，类似与 spec.replica 为 1 的 StatefulSet。

VirtualMachineInstanceMigrations：提供虚拟机迁移的能力，虽然并不会指定具体迁移的目的节点，但要求提供的存储支持 RWX 读写模式。

VirtualMachineInstanceReplicaSet：类似ReplicaSet，可以启动指定数量的 VirtualMachineInstance，并且保证指定数量的 VirtualMachineInstance 运行，可以配置 HPA。

KubeVirt虚拟机生命周期管理主要分为以下几种状态：

1.虚拟机创建：创建VM对象，并同步创建DataVolume/PVC，从Harbor镜像仓库中拉取系统模板镜像拷贝至目标调度主机，通过调度、IP分配后生成VMI以及管理VM的Launcher Pod从而启动供业务使用的VM。

2.虚拟机运行：运行状态下的VM可以进行控制台管理、快照备份/恢复、热迁移、磁盘热挂载/热删除等操作，此外还可以进行重启、下电操作，提高VM安全的同时解决业务存储空间需求和主机异常Hung等问题。

3.虚拟机关机：关机状态下的VM可以进行快照备份/恢复、冷迁移、CPU/MEM规格变更、重命名以及磁盘挂载等操作，同时可通过重新启动进入运行状态，也可删除进行资源回收。

4.虚拟机删除：对虚机资源进行回收，但VM所属的磁盘数据仍将保留、具备恢复条件。

1.4.2 Kata Container

Kata Container[5]社区由 OpenStack Foundation（OSF）领导，Kata Container 是一个开放源代码的容器，运行时可以构建无缝插入容器生态系统的轻量级虚拟机，通过轻量级虚拟机来构建安全的容器，这些虚拟机的运行方式和性能类似于容器，但是使用硬件虚拟化技术作为第二程防御层，可以提供更强的工作负载隔离。

相较于普通的容器技术，Kata Container 的优点如下：

- ✓ 安全：在专用的内核中运行，提供网络，I/O 和内存的隔离，并可以通过虚拟化扩展利用硬件强制隔离。
- ✓ 兼容性：支持行业标准，包括开放容器格式、Kubernetes CRI 等。
- ✓ 性能：提供与标准 Liunx 容器一致的性能。
- ✓ 简单：易于集成和使用。

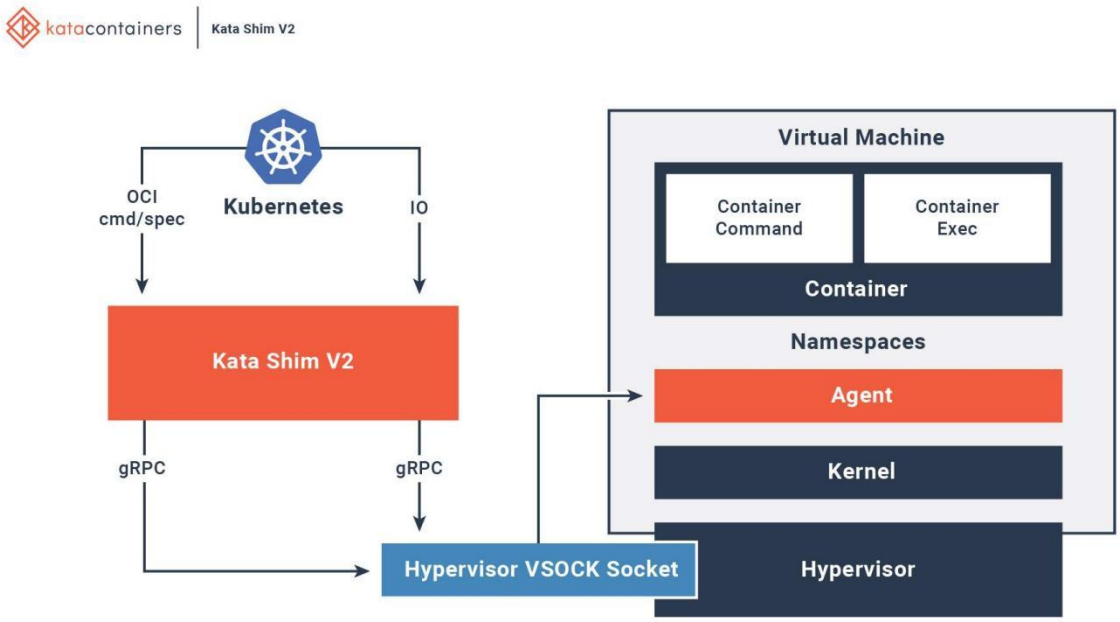


图5：Kata Container架构图

Kata Container主要由由如下几部分组成：

kata-agent：在虚拟机内kata-agent作为一个daemon进程运行，并拉起一个或多个容器的进程。kata-agent使用VIRTIO或VSOCK接口（QEMU在主机上暴露的socket文件）在guest虚拟机中运行gRPC服务器。kata-runtime通过grpc协议与kata-agent通信，向kata-agent发送管理容器的命令。该协议还用于容器和管理引擎（例如Docker Engine）之间传送I/O流（stdout，stderr，stdin）。容器内所有的执行命令和相关的IO流都需要通过QEMU在宿主机暴露的virtio-serial或vsock接口，当使用VIRTIO的情况下，每个虚拟机会创建一个Kata Containers proxy (kata-proxy) 来处理命令和IO流。

kata-runtime：Kata Containers runtime (kata-runtime)通过QEMU/KVM技术创建了一种轻量型的虚拟机，兼容 OCI runtime specification 标准，支持Kubernetes的Container Runtime Interface (CRI)接口，可替换CRI shim runtime (runc) 通过K8s来创建Pod或容器。

kata-proxy：kata-proxy提供了 kata-shim 和 kata-runtime 与VM中的kata-agent通信的方式，其中通信方式是使用virtio-serial或vsock，默认是使用virtio-serial。

Shim：kata-shim类似Docker的 containerd-shim 或CRI-O的 conmon，主要用来监控和回收容器的进程，kata-shim需要处理所有的容器的IO流（stdout, stdin and stderr）和转发相关信号。当前Kata Container发展到了Kata Shim V2（containerd-shim-kata-v2）版本，实现了Containerd Runtime V2 (Shim API)，集成了kata-runtime、kata-shim、kata-proxy的功能，所以架构图中不再包含这几部分。其演进方式如下图所示。

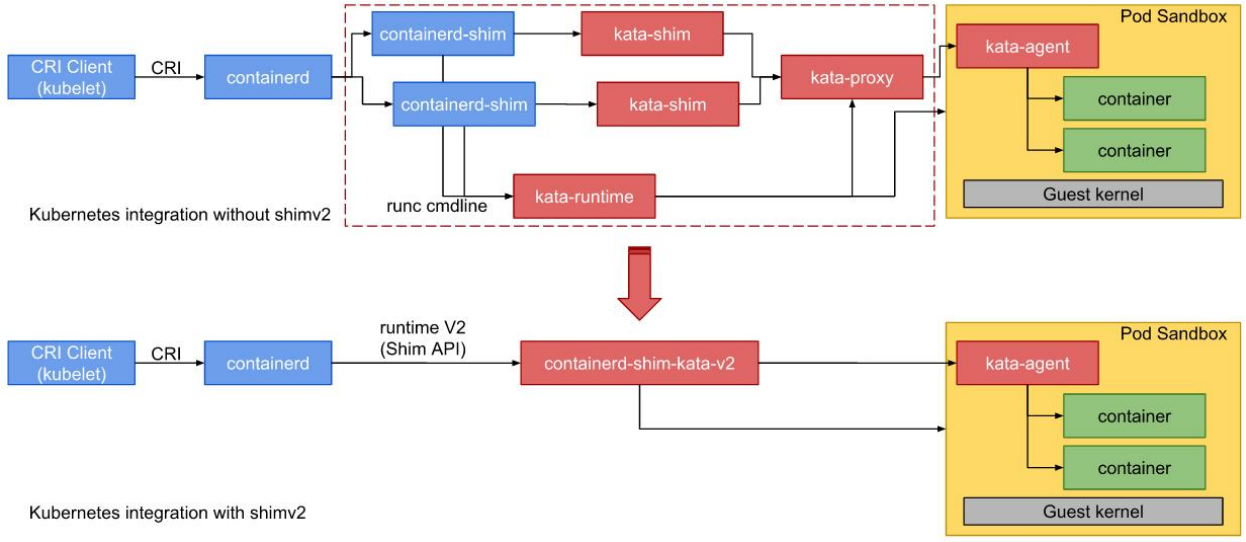


图6：Kata Shim V2演进图

Hypervisor：kata-container通过QEMU/KVM来创建虚拟机给容器运行，可以支持多种hypervisors。

1.4.3 Kube-OVN

Kube-OVN[6]是一款CNCF旗下的企业级云原生网络编排系统，将SDN的能力和云原生结合，提供丰富的功能，极致的性能以及良好的可运维性。Kube-OVN可提供跨云网络管理、传统网络架构与基础设施的互联互通、边缘集群落地等复杂应用场景的能力支持，解除Kubernetes网络面临的性能和安全监控的掣肘，为基于Kubernetes架构原生设计的系统提供成熟的网络底座，提升用户对Kubernetes生态Runtime的稳定性和易用性。

Kube-OVN的设计原则和思路是，平移OpenStack网络的概念和功能到Kubernetes。OpenStack的网络已经发展了很多年，很多设计和概念也基本成了SDN的标准。Kube-OVN通过引入高级功能和成熟的网络概念，从整体上增强Kubernetes网络的能力，并通过OVN实现网络的数据平面，简化维护工作。

Kube-OVN 的组件可以大致分为三类：

- 上游OVN/OVS组件。
- 核心控制器和Agent。
- 监控，运维工具和扩展组件。

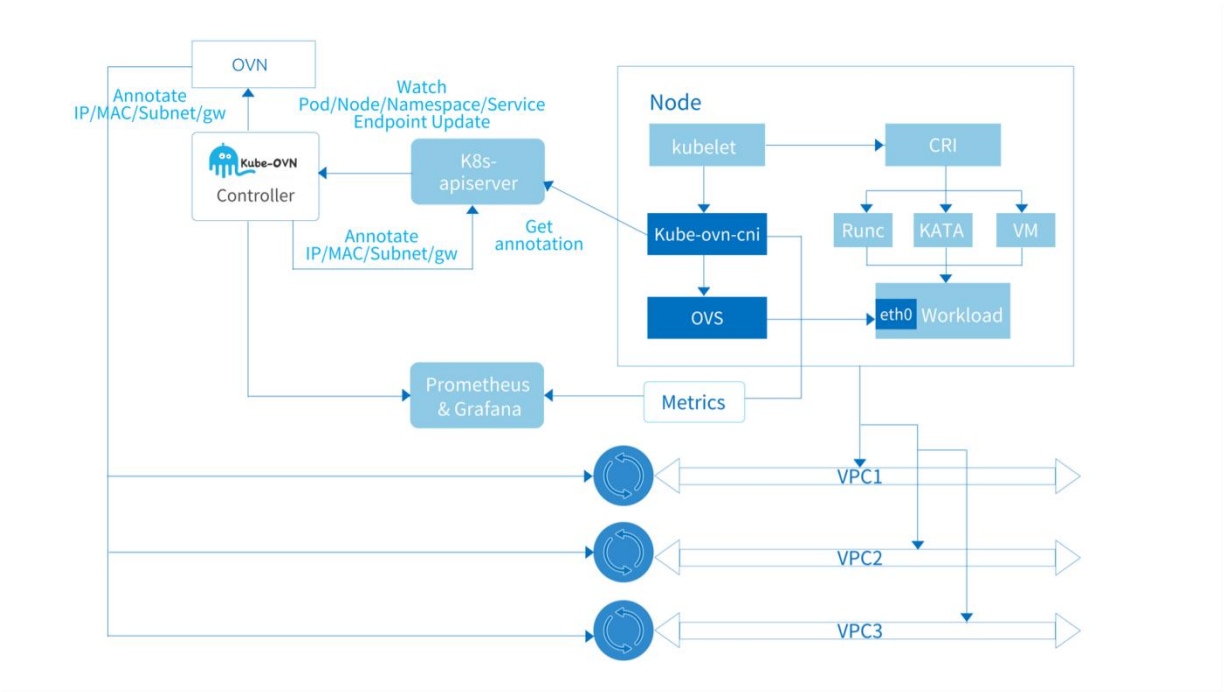


图7：Kube-OVN架构图

上游OVN/OVS组件

该类型组件来自OVN/OVS社区，并针对Kube-OVN的使用场景做了特定修改。OVN/OVS本身是一套成熟的管理虚机和容器的SDN系统。Kube-OVN使用OVN的北向接口创建和调整虚拟网络，并将其中的网络概念映射到Kubernetes之内。

ovn-central：Deployment运行OVN的管理平面组件，包括ovn-nb、ovn-sb和ovn-northd。多个ovn-central实例通过Raft协议同步数据保证高可用。

• ovn-nb：保存虚拟网络配置，并提供 API 进行虚拟网络管理。kube-ovn-controller 将会主要和 ovn-nb 进行交互配置虚拟网络。

• ovn-sb：保存从 ovn-nb 的逻辑网络生成的逻辑流表，以及各个节点的实际物理网络状态。

• ovn-northd：将 ovn-nb 的虚拟网络翻译成 ovn-sb 中的逻辑流表。

ovs-ovn：ovs-ovn 以 DaemonSet 形式运行在每个节点，在 Pod 内运行了 openvswitch、ovsdb 和 ovn-controller。这些组件作为ovn-central的Agent将逻辑流表翻译成真实的网络配置。

核心控制器和 Agent

该部分为Kube-OVN的核心组件，作为OVN和Kubernetes之间的一个桥梁，将两个系统打通并将网络概念进行相互转换。

kube-ovn-controller：该组件为一个Deployment执行所有Kubernetes内资源到OVN资源的翻译工作，其作用相当于整个Kube-OVN系统的控制平面。 kube-ovn-controller监听了所有和网络功能相关资源的事件，并根据资源变化情况更新OVN内的逻辑网络。主要监听的资源包括：Pod、Service、Endpoint、Node、NetworkPolicy、VPC、Subnet、Vlan、ProviderNetwork。

kube-ovn-cni：该组件为一个DaemonSet运行在每个节点上，实现CNI接口，并操作本地的OVS配置单机网络。kube-ovn-cni会配置具体的网络来执行相应流量操作。

监控，运维工具和扩展组件

该部分组件主要提供监控，诊断，运维操作以及和外部进行对接，对Kube-OVN的核心网络能力进行扩展，并简化日常运维操作。

kube-ovn-speaker：该组件为一个DaemonSet运行在特定标签的节点上，对外发布容器网络的路由，使得外部可以直接通过Pod IP访问容器。

kube-ovn-pinger：该组件为一个DaemonSet运行在每个节点上收集OVS运行信息，节点网络质量，网络延迟等信息，收集的监控指标可参考Kube-OVN监控指标。

kube-ovn-monitor：该组件为一个Deployment收集OVN的运行信息，收集的监控指标可参考Kube-OVN监控指标。

kubectl-ko：该组件为kubectl插件，可以快速运行常见运维操作。

2 技术实践

本章通过一些典型地范例介绍对于在K8s环境下运行虚拟机的功能增强的技术实践。

2.1 生命周期管理

2.1.1 在 K8s 环境下实现虚拟机热调整资源

在K8s中启动的虚拟机都是在一个Pod里面运行着libvirtd和qemu等依赖组件，这样kube-scheduler不需要感知Pod里是一个虚拟机还是一个容器，都按照统一的方式进行管理。

既然虚拟机运行在了K8s平台上，那么我们管理虚拟有可以通过kubectl进行管理。

创建虚拟机

通过kubectl create -f vm1.yaml直接通过一个yaml文件来创建一个虚拟机。

更新虚拟机

通过kubectl edit vm -n namespace1即会打开一个vim编辑器，让用户直接可以修改虚拟机的yaml文件。

删除虚拟机

通过kubectl delete vm vm1 -n namespace1来删除在namespace1下的一个虚拟机vm1。

虚拟机热调整资源

由于K8s最近的版本已经支持Pod原地扩容了，可以利用了这个功能，实现kubevirt的虚拟机的cpu和memory热添加的功能，社区目前只支持cpu的热插。

* 社区的热扩容的实现：社区目前之实现了通过了live migration（热迁移）功能来实现的，这样的实现依赖虚拟机是否可以进行热迁移，比如虚拟机如果有gpu挂载的话，就不能实现热迁移，也就不能热扩容。

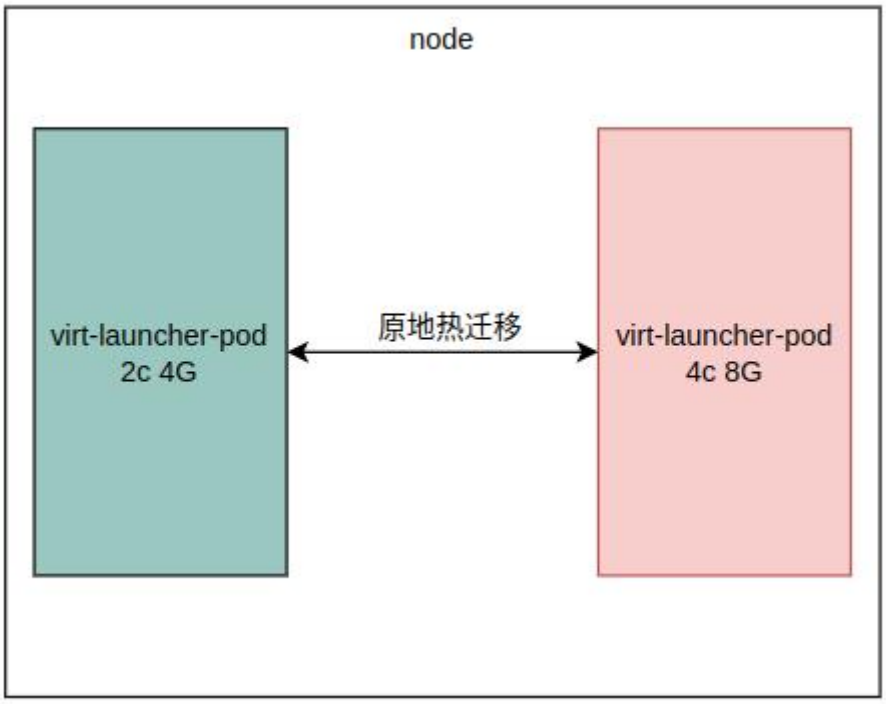


图8：社区版热扩容

* 改进的实现：首先使用了1.27.x版本的特性Pod原地扩容的特性(Pod in-place VPA)先将外部的virt-launcher Pod的limit调整到期望的大小，然后再调用libvirt api去扩容虚拟机到期望的大小。

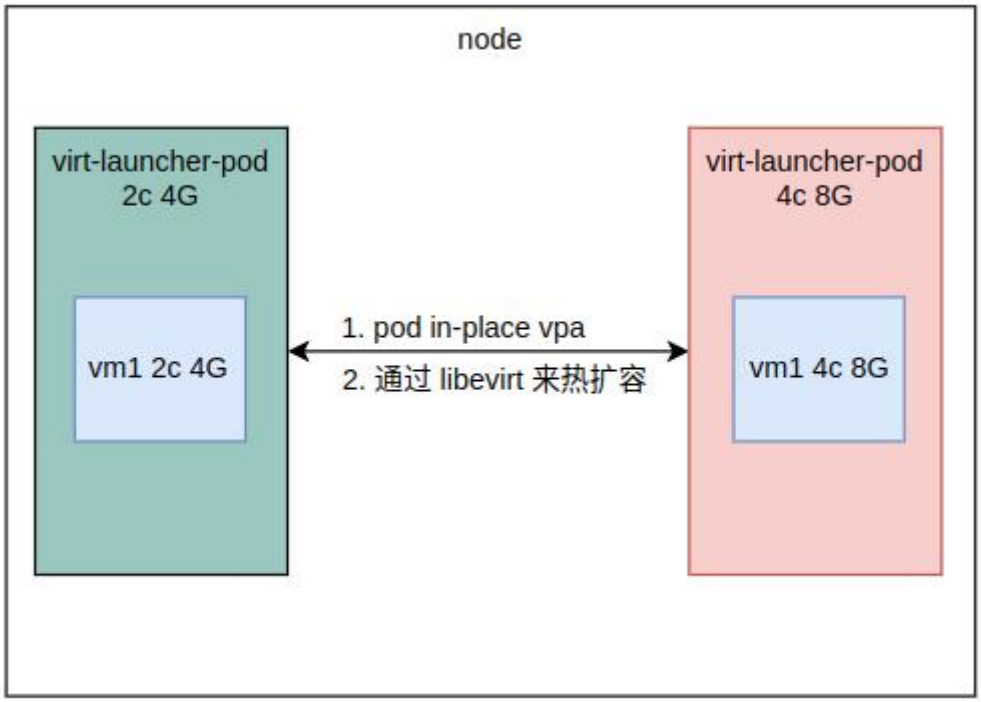


图9：改进版虚拟机热扩容

2.2 镜像管理

2.2.1 从 Harbor 镜像仓库拉取镜像

在容器云平台启动虚拟机，那么虚拟机镜像最好是能和容器镜像一起管理，避免增加不必要的管理成本，所以可以将制作好的虚拟机镜像伪装成为了一个容器镜像存放在Harbor中，这样就不用单独管理虚拟机的镜像了。

Harbor[7]是由VMware公司开源的企业级的Docker Registry管理项目，它包括权限管理(RBAC)、LDAP、日志审核、管理界面、自我注册、镜像复制和中文支持等功能。

Harbor镜像仓库地址为：harbor.mec.local，首先通过kubectl命令进行配置：

```
# kubectl get cm -n mec-images hci-controller-config -o yaml
# kubectl edit cm -n mec-images hci-controller-config -o yaml
apiVersion: v1
data:
  config.yaml: |
    healthzPort: 8080
    resyncPeriod: 10m
    leaderElection:
      leaderElect: true
      leaseDuration: 30s
      renewDeadline: 15s
      resyncPeriod: 5s
```

```
      resourceName: hci-controller
      resourceLock: endpointsleases
      resourceNamespace: mec-images
  controllerConfig:
    baseImageNamespace: mec-images
    snapshotClass: csi-rbdplugin-snapclass # name of VolumeSnapshotClass
    glanceBaseURL: https://172.18.22.100:9292
    registryAddr: harbor.mec.local
kind: ConfigMap
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"config.yaml":"healthzPort: 8080\nresyncPeriod:
  creationTimestamp: "2022-07-06T14:44:34Z"
  name: hci-controller-config
  namespace: mec-images
  resourceVersion: "18229389"
  uid: 3de8bcfc-f87d-4be5-9c85-d84198866133
```

在Harbor仓库中的镜像有 kubevirt/fedora36，创建EVM时采用该镜像：

```
# kubectl create -f evm-registry-fedora.yaml
# cat evm-registry-fedora.yaml
apiVersion: mececs.io/v1beta1
kind: EnhancedVirtualMachine
metadata:
  name: ecs-registry-fedora
  namespace: wq-test
spec:
  template:
    spec:
      running: true
      template:
        metadata:
          labels:
            kubevirt.io/vm: ecs-registry-fedora
          annotations:
            ovn.kubernetes.io/allow_live_migration: 'true'
            K8s.v1.cni.cncf.io/networks: mec-nets/attachnet1
            attachnet1.mec-nets.ovn.kubernetes.io/logical_switch: subnet-ipv4
            attachnet1.mec-nets.ovn.kubernetes.io/default_route: 'true'
            attachnet1.mec-nets.ovn.kubernetes.io/allow_live_migration: 'true'
      spec:
        domain:
          cpu:
            sockets: 4
            cores: 1
            threads: 1
```

```
      memory:
        guest: "8192Mi"
      clock:
        timezone: "Asia/Shanghai"
      timer:
        rtc:
          present: true
      devices:
        disks:
        - disk:
            bus: virtio
            name: cloudinitdisk
        interfaces:
        - bridge: {}
          name: attachnet1
      resources:
        requests:
          cpu: 2
          memory: 2048Mi
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 114.114.114.114
    options:
      - name: ndots
        value: "5"
  hostname: "ecs-registry-fedora"
  networks:
  - name: attachnet1
    multus:
      networkName: mec-nets/attachnet1
  volumes:
  - name: cloudinitdisk
    cloudInitNoCloud:
      userData: |-
        #cloud-config
        password: fedora
        ssh_pwauth: True
        chpasswd: { expire: False }
  source:
    registryImageURL: kubevirt/fedora36:latest
  bootVolume:
    resources:
      requests:
        storage: 10Gi
```

创建成功，EVM可正常访问。

```
# kubectl get evm ecs-registry-fedora -n wq-test
```

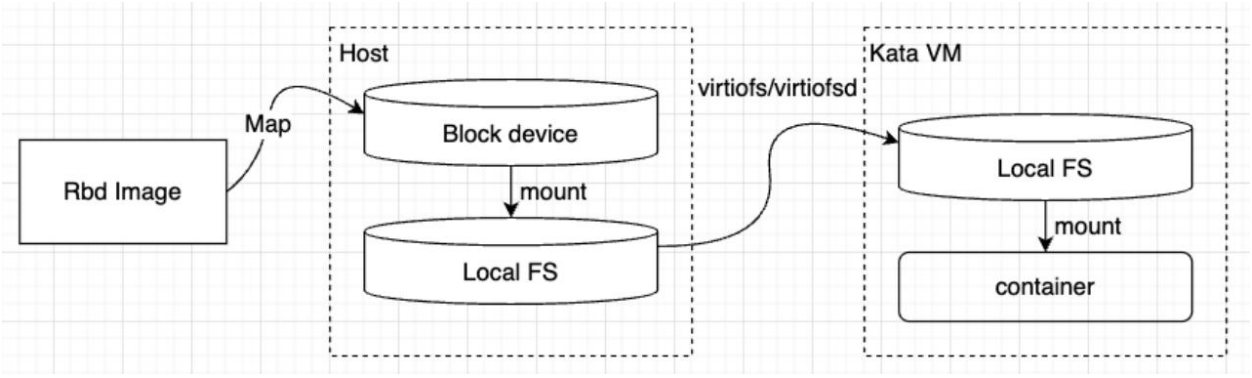
```
NAME                          AGE
ecs-registry-fedora 3h1m
# kubectl get vm ecs-registry-fedora -n wq-test
NAME                          AGE    STATUS    READY
ecs-registry-fedora 3h1m Running True
# kubectl get vmi ecs-registry-fedora -n wq-test
NAME                          AGE    PHASE     IP                NODENAME READY
ecs-registry-fedora 3h1m Running 192.168.100.26 ecs8         True
```

2.3 存储管理

2.3.1 Kata Container 使用卷的存储直通模式

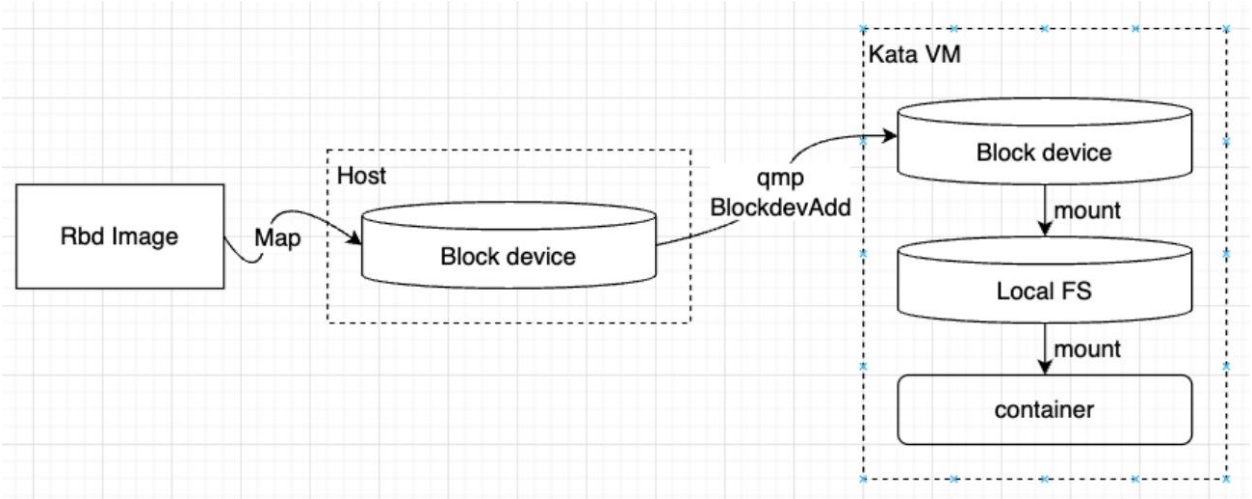
本项优化是为了提高使用Kata作为容器运行时提高Ceph rbd作为磁盘时的IO性能。

原有挂载是储存卷（RBD image）挂载到宿主机的目录中，然后通过virtio-fs协议，将存储卷的内容共享到Kata Container中。如下图所示：

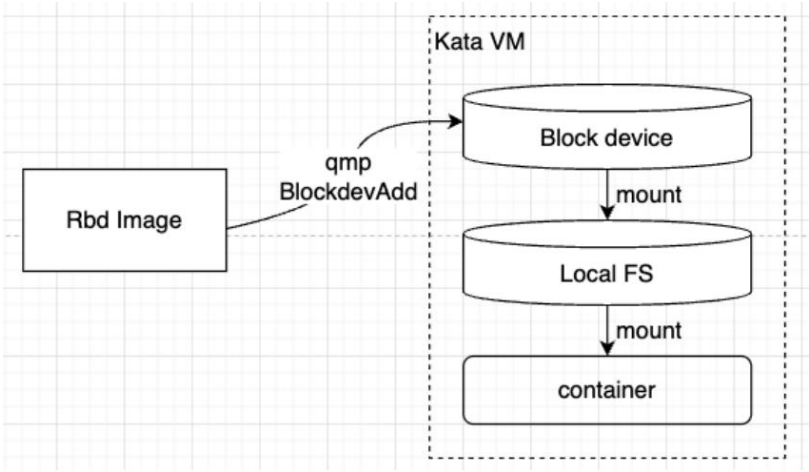


可以添加卷直通功能，分为半直通（块直通）和全直通（rbd 直通）两个模式。

其中半直通如下图所示，去掉了中间的协议virtio-fs，将映射到宿主机上面的块设备通过qmp命令直接添加到了Kata Container中，Kata Container再通过mount块设备进行内容的读写。



全直通如下图所示，进一步的去掉了挂载到宿主机上面的动作，将rbd image直接通过qmp指令作为块设备添加到了Kata Container中， Kata Container再通过mount块设备进行内容的读写。



卷直通功能是否开启通过PVC的annotations进行控制。在PVC的annotations中添加了volume.katacontainers.io/direct-mode字段。

1. 当volume.katacontainers.io/direct-mode值为“block”时，为半直通模式；
2. 当volume.katacontainers.io/direct-mode值为“rbd”时，为全直通模式；
3. 在字段的值不为上述两个或者不添加该字段时为原有模式。

直通模式，通过kubectl exec -it centos-blk-test -- bash进入到对应容器后可以看到对应的块设备且对应的Filesystem不为 none：

```
[root@deployer simple-test]# kubectl exec -it centos-blk-test -- bash
[root@centos-blk-test /]# df -h
Filesystem Size Used Avail Use% Mounted on
/dev/sdb 4.9G 265M 4.4G 6% /
tmpfs 64M 0 64M 0% /dev
tmpfs 998M 0 998M 0% /sys/fs/cgroup
/dev/sdc 2.0G 6.0M 1.9G 1% /data-rbd
/dev/sdd 2.0G 6.0M 1.9G 1% /data-block
shm 998M 0 998M 0% /dev/shm
[root@centos-blk-test /]# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 5G 0 disk
sdb 8:16 0 5G 0 disk /
sdc 8:32 0 2G 0 disk /data-rbd
sdd 8:48 0 2G 0 disk /data-block
pmem0 259:0 0 382M 1 disk
`-pmem0p1 259:1 0 380M 1 part
```

半直通模式，创建Pod之前通过lsblk查看host的块设备信息，如果为半直通则创建完Pod之后host会多出来一个rbd的块设备

```
[root@host1 ~]# lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sr0 11:0 1 486K 0 rom
rbd1 251:16 0 2G 0 disk
vda 252:0 0 50G 0 disk
└─vda1 252:1 0 50G 0 part /
```

2.3.2 Kata Container 和 OpenEBS 适配优化

本项优化是为了提高使用Kata作为容器运行时提高OpenEBS作为磁盘时的IO性能。

OpenEBS[8]是一种开源云原生存储解决方案，托管于CNCF基金会。OpenEBS是K8s本地超融合存储解决方案，它管理节点可用的本地存储，并为有状态工作负载提供本地或高可用的分布式持久卷。OpenEBS支持两大类卷，本地卷和复制卷。管理员和开发人员可以使用kubectl、Helm、Prometheus、Grafana、Weave Scope等K8s可用的工具来交互和管理OpenEBS。

为了实现Kata Container使用OpenEBS的本地卷直通方式，修改OpenEBS的lvm-localpv直通的实现。

OpenEBS的CSI nodeDriver主要功能包括挂载、卸载、扩容和状态获取四项：

1. NodePublishVolume：格式化块设备并将块设备挂载到targetPath。
2. NodeUnPublishVolume：将块设备从targetPath卸载。
3. NodeExpandVolume：对volume进行扩容。
4. NodeGetVolumeStats：获取卷的使用情况。

修改方案依赖PVC annotations实现是否为直通卷的判断，在CSI中针对上述四项功能分别对 kata进行适配：

1. NodePublishVolume：
    a. 通过annotations判断是否为直通卷；
    b. 通过targetPath目录下的文件判断该直通卷是否已经挂载，已经挂载则直接重新调用kata-runtime direct-volume add 即可（重启后kata-runtime add创建的文件会消失）；
    c. 如果未挂载则先对块设备进行格式化（OpenEBS通过调用K8s的库实现格式化并挂载）；
    d. 格式化完成后调用kata-runtime direct-volume add命令；
    e. 在targetPath创建一个文件用于判断是否为直通卷（因为annotations只会在stageVolume、publishVolume阶段可以获取到，其他阶段无法获取），并在文件中写入挂载状态；
2. NodeUnPublishVolume：
    a. 通过targetPath目录下的文件判断是否为直通卷；
    b. 如果为直通卷则调用kata-runtime direct-volume delete命令进行删除；
3. NodeExpandVolume：
    a. 通过targetPath目录下的文件判断是否为直通卷；
    b. 如果为直通卷则在使用lvextend对文件系统进行扩容；
    c. 待块设备扩容完成后调用kata-runtime direct-volume resize命令调整大小；
4. NodeGetVolumeStats：
    a. 通过targetPath目录下的文件判断是否为直通卷；
    b. 如果为直通卷则调用kata-runtime direct-volume stats命令获取volume状态。

2.4 网络能力

2.4.1 DPDK 加速的 OVS

DPDK是X86平台报文快速处理的库和驱动的集合，不是网络协议栈，不提供二层，三层转发功能，不具备防火墙ACL功能，但通过DPDK可以轻松的开发出上述功能。DPDK的优势在于，可以将用户态的数据，不经过内核直接转发到网卡，实现加速目的。DPDK加速的OVS与原始OVS的区别在于，从OVS连接的某个网络端口接收到的报文不需要openvswitch.ko内核态的处理，报文通过DPDK PMD驱动直接到达用户态ovs-vswitchd里。

社区方案里，虚拟机/容器不能利用DPDK技术来提高网络的流量，本节描述的解决方案中加入使用DPDK技术来让虚拟机网络包可以直接通过user space来交互通讯，大大提高的了可以处理的网络流量。

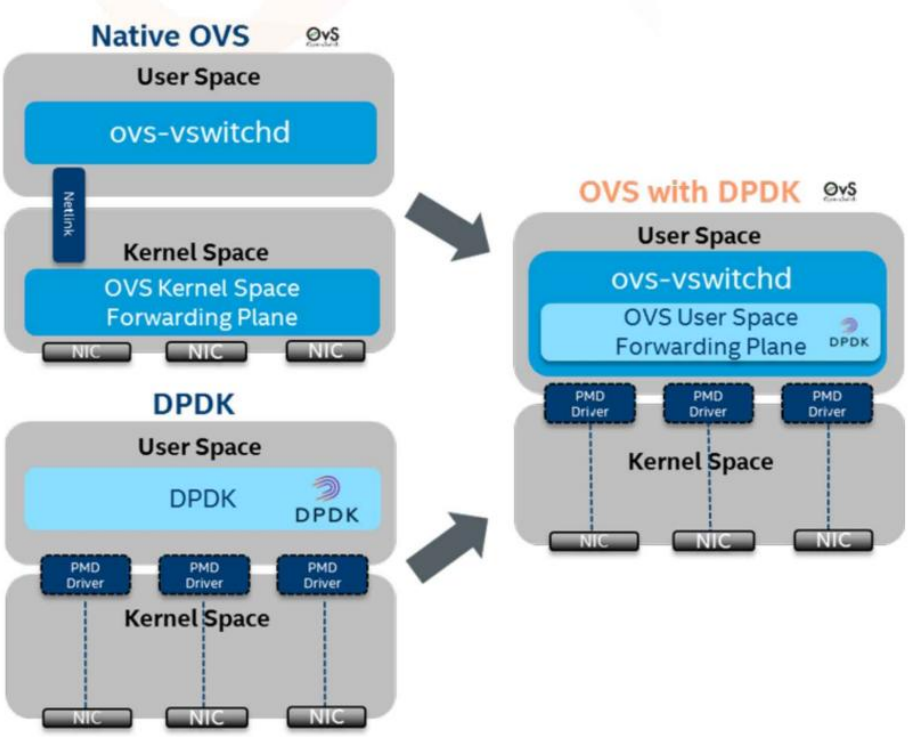


图10：DPDK加速的OVS

DPDK加速的OVS数据流转发的大致流程如下：

1.OVS的ovs-vswitchd接收到从OVS连接的某个网络端口发来的数据包，从数据包中提取源/目的IP、源/目的MAC、端口等信息。

2.OVS在用户态查看精确流表和模糊流表，如果命中，则直接转发。

3.如果不命中，在SDN控制器接入的情况下，经过OpenFlow协议，通告给控制器，由控制器处理。

4.控制器下发新的流表，该数据包重新发起选路、匹配和报文转发。

总结：主要区别在于流表的处理，普通OVS流表转发在内核态，而OVS-DPDK流表转发在用户态。

查看Pod, vmi可正常启动：

```
# kubectl get Pod
NAME READY STATUS RESTARTS AGE
virt-launcher-vm-dpdk-6nrqt 2/2 Running 0 3m13s
# kubectl get vmi -A
NAME        AGE   PHASE     IP                NODENAME READY
vm-dpdk   3m9s Running 172.18.22.173   node173   True
```

查看OVS-DPDK创建了vmi对应的vhostuserclient port（ede503e0_net2_h）：

```
# kubectl ko vsctl node173 show
340aee64-1e86-486a-a99e-a62481e9d67a
    Bridge br-int
        fail_mode: secure
        datapath_type: netdev
        Port "1e9da8d5d0d7_h"
            Interface "1e9da8d5d0d7_h"
        Port ovn0
            Interface ovn0
                type: internal
        Port "43d5339cda9a_h"
            Interface "43d5339cda9a_h"
```

```
        Port "99e85593750e_h"
            Interface "99e85593750e_h"
        Port d6653a5bbdc2_h
            Interface d6653a5bbdc2_h
        Port ede503e0_net2_h
            Interface ede503e0_net2_h
                type: dpdkvhostuserclient
                options:
{vhost-serverpath="/var/run/openvswitch/vhost_sockets/578dd327-f9ba-4a1b-8bd3-1a55351e07ab/vhostuser-sockets/net2"}
```

2.4.2 Kata Container 使用 SR-IOV 设备并运行 DPDK 应用

Kata Container默认采用QEMU作为hypervisor，而QEMU不支持veth，所以一般默认方案是采用TAP来为VM内外打通网络。本节展示在K8s环境下Kata Container使用Mellanox的设备并运行DPDK应用的能力。

本项优化同样是为了提高网络流量，同DPDK技术一样，SRIOV（single root IO virtualization）技术同样也为了提高虚拟机/容器的网络性能，SRIOV技术可以将一张物理网卡变为若干个虚拟的物理网卡，并直接接入虚拟机/容器从而提高网络性能。

内核编译

通常linux启动时先加载kernel，再加载initrd.img，initrd.img通常被用来加载驱动模块。但是在Kata Container中，不能通过启动时加载驱动模块，所以需要在编译kernel时将需要的驱动模块通过配置编译到内核文件中。

Kata提供编译脚本和内核配置模板，可以修改配置文件或者通过make menuconfig命令启动图形界面进行修改。因为模块间的依赖和互斥关系相当复杂，建议通过make menuconfig启动图形界面来开启下列模块：

- CONFIG_DCB networking options -> Data Center Bridging support
- CONFIG_INFINIBAND device Drivers -> InfiniBand support
- CONFIG_DYNAMIC_MEMORY_LAYOUT Processor type and featres -> Randomize the kernel memory sections
- CONFIG_COMPAT Binary Emulations -> IA32 Emulation
- CONFIG_NUMA Processor type and features -> NUMA Memory Allocation and Scheduler Support
- CONFIG_PAGE_POOL General setup-> Page allocator randomization
- CONFIG_MODULES enable_loadable module support
- CONFIG_PCI device driver -> Userspace I/O drivers
- CONFIG_MLX5 device driver -> network device support -> Ethernet driver support -> Mellanox devices

Kernel-header包制作

Mellanox驱动安装需要依赖kernel-header模块，所以需要提前准备和镜像制作系统相同内核的Kernel-header安装包。

在第一步的内核文件夹使用下列命令可以生成.deb的包。

```
$ make deb-pkg
```

编译Mellanox驱动

以Dockerfile为例，编辑镜像步骤如下：

```
FROM ***
WORKDIR /
```

```
# 加入Kernel-header模块包
ADD *.deb ./
# 安装Kernel-header模块包
RUN dpkg -i *.deb
# 安装编译需要的软件和库
RUN apt-get update && apt-get install -y \
    gcc \
    ......
# 下载Mellanox驱动编译包
ADD MLNX_OFED_LINUX-5.7-1.0.2.0-ubuntu20.04-x86_64 ./MLNX_OFED_LINUX-5.7-1.0.
2.0-ubuntu20.04-x86_64
# 编译安装Mellanox驱动
RUN ./mlnxofedinstall --upstream-libs --dpdk

# 运行DPDK相关应用
......
```

之后就可以在Kata Container中使用SR-IOV设备和DPDK的应用了。

2.4.3 开启 IPv4v6 双栈功能

开启双栈可以让K8s平台上的虚拟机/容器能够同时支持ipv4/ipv6（可选）双协议栈。虚拟机开启IPv4v6双栈需要K8s和Kube-ovn都开启双栈。

K8s启用双栈

要启用IPv4v6双协议栈，需要为集群的相关组件启用IPv6DualStack特性，并且设置双协议栈的集群网络分配。K8s采用kubeadm方式部署，需要修改相关组件的yaml文件。

kube-apiserver：启用IPv4v6双栈特性。

```
# vim /etc/kubernetes/manifests/kube-apiserver.yaml
- --feature-gates=IPv6DualStack=true
- --service-cluster-ip-range=10.233.0.0/18,fd00:10:96::/112
```

kube-controller-manager：启用IPv4v6双栈特性，并增加Pod/service IPv6 CIDR。

```
# vim /etc/kubernetes/manifests/kube-controller-manager.yaml
- --feature-gates=IPv6DualStack=true
- --service-cluster-ip-range=10.233.0.0/18,fd00:10:96::/112
- --cluster-cidr=10.244.0.0/16,fc00::/48
- --node-cidr-mask-size-ipv4=24
- --node-cidr-mask-size-ipv6=64
```

Kubelet：启用IPv4v6双栈特性。

```
# vim /etc/sysconfig/kubelet
# vim /var/lib/kubelet/config.yaml
KUBELET_EXTRA_ARGS="--feature-gates=IPv6DualStack=true"
```

kube-proxy：启用IPv4v6双栈特性，并增加Pod IPv6 CIDR。

```
# kubectl -n kube-system edit cm kube-proxy
data:
  config.conf: |-
    ......
    featureGates:
      IPv6DualStack: true
    clusterCIDR: 10.244.0.0/16,fc00::/48
```

Kube-ovn启用双栈

在Kube-ovn的安装脚本中打开对IPv4v6双栈的支持。

```
# vim install.sh
DUAL_STACK=${DUAL_STACK:-true}
```

开启双栈部署好Koube-ovn后，在配置子网双栈时，需要设置子网CIDR格式为cidr=<IPv4 CIDR>,<IPv6 CIDR>，CIDR顺序要求IPv4在前，IPv6在后。

```
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: subnet1-bj1
  namespace: sgbj
spec:
  vpc: vpc-bj1-sg
  protocol: IPv4
  default: false
  cidrBlock: 192.168.100.0/24,fd00:10:18::/64
  excludeIps:
  - 192.168.100.1
  - fd00:10:18::1
  gateway: 192.168.100.1,fd00:10:18::1
  gatewayNode: ""
  disableGatewayCheck: true
  gatewayType: distributed
  natOutgoing: true
  private: false
```

正常指定子网启动Pod。

```
# cat Pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: Pod
  namespace: default
  annotations:
    ovn.kubernetes.io/logical_switch: ovn-default
    K8s.v1.cni.cncf.io/networks: mec-nets/attachnetsg, mec-nets/attachnet1sg
    attachnetsg.mec-nets.ovn.kubernetes.io/logical_switch: subnet1-bj1
    attachnet1sg.mec-nets.ovn.kubernetes.io/logical_switch: subnet2-bj1
spec:
  containers:
    - name: spec-subnet4
      command: ["/bin/ash", "-c", "trap : TERM INT; sleep 36000 & wait"]
      image: rancher/curl
......
```

Pod可以正常获取IPv4和IPv6 地址，并且路由正常。

```
# kubectl exec -it Pod -- ip a
......
52: eth0@if53: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc noqueue state
```

```
UP
        link/ether 00:00:00:5d:bd:30 brd ff:ff:ff:ff:ff:ff
        inet 10.16.0.28/16 brd 10.16.255.255 scope global eth0
            valid_lft forever preferred_lft forever
        inet6 fd00:10:16::1c/64 scope global
            valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fe5d:bd30/64 scope link
            valid_lft forever preferred_lft forever
    54: net1@if55: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc noqueue state
UP
        link/ether 00:00:00:2f:af:e4 brd ff:ff:ff:ff:ff:ff
        inet 192.168.100.5/24 brd 192.168.100.255 scope global net1
            valid_lft forever preferred_lft forever
        inet6 fd00:10:18::5/64 scope global
            valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fe2f:afe4/64 scope link
            valid_lft forever preferred_lft forever
    56: net2@if57: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1400 qdisc noqueue state
UP
        link/ether 00:00:00:09:8a:5d brd ff:ff:ff:ff:ff:ff
        inet 192.168.200.10/24 brd 192.168.200.255 scope global net2
            valid_lft forever preferred_lft forever
        inet6 fd00:10:17::a/64 scope global
            valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fe09:8a5d/64 scope link
            valid_lft forever preferred_lft forever
    ......
    # kubectl exec -it Pod -- ip -6 route show
    fd00:10:16::/64 dev eth0 metric 256
    fd00:10:17::/64 dev net2 metric 256
    fd00:10:18::/64 dev net1 metric 256
    ......
```

OVN逻辑网关和逻辑路由配置正常。

```
    # kubectl ko nbctl show
    switch ........
        port ........
            addresses: ["00:00:00:E6:21:AE 192.168.100.3 fd00:10:18::3"]
        port ........
            addresses: ["00:00:00:2F:AF:E4 192.168.100.5 fd00:10:18::5"]
        port ........
            type: router
            router-port: ........
    ........
    # kubectl ko nbctl lr-route-list ovn-cluster
    IPv4 Routes
                    10.16.0.4               100.64.0.3 src-ip
                    ......
```

```
IPv6 Routes
                fd00:10:16::4           fd00:100:64::3 src-ip
                ......
```

在宿主机上查看ovn0网络正常。

```
# ip a
........
8: ovn0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ether 00:00:00:f0:ac:c6 brd ff:ff:ff:ff:ff:ff
    inet 100.64.0.2/16 brd 100.64.255.255 scope global ovn0
       valid_lft forever preferred_lft forever
    inet6 fd00:100:64::2/64 scope global
       valid_lft forever preferred_lft forever
    inet6 fe80::200:ff:fef0:acc6/64 scope link
       valid_lft forever preferred_lft forever
.......
```

Kata Container使用IPv4v6双栈

ECI指定子网启动Kata Container，可以正常获取双栈地址。

```
# cat Pod10.yaml
apiVersion: v1
kind: Pod
metadata:
  name: Pod10
  namespace: sgbj
  annotations:
    ovn.kubernetes.io/logical_switch: ovn-default
    K8s.v1.cni.cncf.io/networks: mec-nets/attachnetsg, mec-nets/attachnet1sg
    attachnetsg.mec-nets.ovn.kubernetes.io/logical_switch: subnet1-bj1
    attachnet1sg.mec-nets.ovn.kubernetes.io/logical_switch: subnet2-bj1
spec:
  runtimeClassName: kata-qemu
  nodeName: master
  containers:
  - name: Pod7
    command: ["/bin/ash", "-c", "trap : TERM INT; sleep 36000 & wait"]
    image: rancher/curl
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
......
```

在Kata Container中查看本地网卡，可以正常获取IPv4和IPv6地址。

```
# ip a
......
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq state UP qlen 1000
    link/ether 00:00:00:75:b3:f2 brd ff:ff:ff:ff:ff:ff
    inet 10.16.0.4/16 brd 10.16.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

```
        inet6 fd00:10:16::4/64 scope global
           valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fe75:b3f2/64 scope link
           valid_lft forever preferred_lft forever
    3: net1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq state UP qlen 1000
        link/ether 00:00:00:a8:1a:bb brd ff:ff:ff:ff:ff:ff
        inet 192.168.100.2/24 brd 192.168.100.255 scope global net1
           valid_lft forever preferred_lft forever
        inet6 fd00:10:18::2/64 scope global
           valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fea8:1abb/64 scope link
           valid_lft forever preferred_lft forever
    4: net2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq state UP qlen 1000
        link/ether 00:00:00:c7:37:2e brd ff:ff:ff:ff:ff:ff
        inet 192.168.200.2/24 brd 192.168.200.255 scope global net2
           valid_lft forever preferred_lft forever
        inet6 fd00:10:17::2/64 scope global
           valid_lft forever preferred_lft forever
        inet6 fe80::200:ff:fec7:372e/64 scope link
           valid_lft forever preferred_lft forever
    ......
```

VM使用IPv4v6双栈

需要修改Kubevirt支持bridge类型IPv4v6双栈。

```
    # kubectl exec -it Pod10 -n sgbj -- sh
    [fedora@vm1 ~]$ ip a
    ......
    2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq_codel state UP group default
qlen 1000
        link/ether 00:00:00:72:cc:56 brd ff:ff:ff:ff:ff:ff
                                        altname enp1s0
        inet 192.168.100.9/24 brd 192.168.100.255 scope global dynamic noprefixroute eth0
           valid_lft 86313494sec preferred_lft 86313494sec
        inet6 fd00:10:18::9/128 scope global dynamic noprefixroute
           valid_lft 86313495sec preferred_lft 86313495sec
        inet6 fe80::200:ff:fe72:cc56/64 scope link noprefixroute
           valid_lft forever preferred_lft forever
    3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc fq_codel state UP group default
qlen 1000
        link/ether 00:00:00:86:e6:ba brd ff:ff:ff:ff:ff:ff
        altname enp2s0
        inet 192.168.200.9/24 brd 192.168.200.255 scope global dynamic noprefixroute eth1
           valid_lft 86313494sec preferred_lft 86313494sec
        inet6 fd00:10:17::9/128 scope global dynamic noprefixroute
           valid_lft 86313495sec preferred_lft 86313495sec
        inet6 fe80::200:ff:fe86:e6ba/64 scope link noprefixroute
           valid_lft forever preferred_lft forever
    .......
```

```
[fedora@vm1 ~]$ ip -6 r
::1 dev lo proto kernel metric 256 pref medium
fd00:10:17::9 dev eth1 proto kernel metric 101 pref medium
fd00:10:17::/64 dev eth1 proto ra metric 101 pref medium
fd00:10:17::/64 dev eth1 proto ra metric 101 pref medium
fd00:10:18::9 dev eth0 proto kernel metric 100 pref medium
fd00:10:18::/64 dev eth0 proto ra metric 100 pref medium
fe80::/64 dev eth0 proto kernel metric 100 pref medium
fe80::/64 dev eth1 proto kernel metric 101 pref medium
default via fe80::200:ff:fed6:6258 dev eth0 proto ra metric 100 pref medium
default via fe80::200:ff:fe2b:192d dev eth1 proto ra metric 101 pref medium
```

2.4.4 NicPort 网卡热插拔使用

社区实现的虚拟机网卡热插拔的方式不符合用户的使用习惯,本节描述的方法可以给用户更好的界面体验。本节新开发了一个CRD叫做NicPort，该CRD的每个实列即代表了一个IP地址，可以将这个实列单独创建，然后由管理员决定将创建好的实列分配给需要的虚拟机，从而达到良好的用户体验。

创建NicPort分为两种情况：

- ✓ 创建NicPort时选择vm，则直接创建绑定；
- ✓ 创建NicPort时不选择vm，只创建网卡。

创建NicPort

创建NicPort需要选择子网，可以通过kubectl get subnet查看子网信息和CIDR。创建时可以选择是否指定IP，指定IP时要满足IP在所选子网的CIDR范围内，且不冲突。如果不指定IP，创建时会根据子网自动分配。当前CNI类型只支持kube-ovn，network_type类型只支持bridge。

查看子网：

```
# kubectl get subnet
NAME                    PROVIDER VPC            PROTOCOL CIDR
PRIVATE NAT    DEFAULT GATEWAYTYPE V4USED V4AVAILABLE V6USED V6AVAILABLE
EXCLUDEIPS
......
subnet1-bj1             ovn         vpc-bj1-sg   IPv4        192.168.100.0/24
false     true false     distributed 2          251             0          0
["192.168.100.1"]
```

方式一：使用API

登录到ecs1节点，例如创建NicPort时不绑定vm，选择subnet subnet1-bj1，IP为192.168.100.200，MAC为9e:13:e7:31:56:1d，示例如下：

```
curl -v -XPOST -H "Accept: application/json" -H "Content-Type:
application/json" 'https://ecsvip.lab.ecs.io:6443/apis/kubevirt.chinaunicom.co
m/v1/namespaces/default/nicports' \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key \
--data
'{"apiVersion":"kubevirt.chinaunicom.com/v1","kind":"NicPort","metadata":
{"name":"nicport","namespaces":"default"},"spec":{"subnet":"subnet1-
bj1","ip":"192.168.100.200","cni":"kubeovn","mac":"9e:13:e7:31:56:1d","network_type":"bridge","vmi":""}
```

```
}'
```

创建成功后，可以通过 GET NicPort 中的方法查看。

如果创建NicPort时绑定vm，需要将curl data中的vmi字段选择指定的vmi。

方式二：使用 kubectl

创建 nicport1.yaml，示例如下：

```
apiVersion: kubevirt.chinaunicom.com/v1
kind: NicPort
metadata:
    name: nicport1
spec:
    subnet: "subnet1-bj1"
    ip: "192.168.100.221"
    cni: "kube-ovn"
    mac: "9e:13:e7:31:56:1f"
    network_type: "bridge"
    vmi: "" // 选填
```

然后使用kubectl apply -f nicport.yaml命令创建网卡。

获取NicPort

方式一： 使用API

登录到ecs1节点，url的最后要指定NicPort的名字，可以返回NicPort的信息。

```
curl -v -XGET -H "Accept: application/json" -H "Content-Type:
application/json" ‘https://ecsvip.lab.ecs.io:6443/apis/kubevirt.chinaunicom.co
m/v1/namespaces/default/nicports/nicport1’ \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key
```

方式二：使用 kubectl

通过kubectl describe nicport <nicport_name>或者kubectl get nicport <nicport_name> -o yaml查看NicPort信息。

NicPort有多个状态。状态为 phase0.1表示创建了NicPort，正确分配 IP MAC，但未绑定虚拟机；状态为phase1表示绑定了虚拟机，示例如下：

```
Spec:
    Cni: kube-ovn
    Ip: 192.168.100.200
    Mac: 9e:13:e7:31:56:1d
    network_type: bridge
    Subnet: subnet1-bj1
    Vmi:
Status:
    dhcp_advertising_ip:
    Stat: phase0.1    // NicPort状态
Events: <none>
```

列出NicPort

方式一：使用 API

登录到ecs1节点，使用下列命令，可以看到NicPort列表：

```
curl -v -XGET -H "Accept: application/json" -H "Content-Type:
application/json" 'https://ecsvip.lab.ecs.io:6443/apis/kubevirt.chinaunicom.co
m/v1/namespaces/default/nicports?limit=500' \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key
```

方式二：使用 kubectl

使用kubectl get nicports命令可以看到NicPort列表。

更新NicPort

可以更新NicPort的vmi字段，实现将网卡挂载到虚拟机上和从虚拟机卸载网卡：绑定到虚拟机时，设置vmi-target字段为非空；解绑时设置vmi-target字段为空（""）。

方式一：使用 API

挂载NicPort的示例如下：

```
curl -v -XPATCH -H "Accept: application/json" -H "Content-Type:
application/merge-patch+json" 'https://ecsvip.lab.ecs.io:6443/apis/kubevirt.ch
inaunicom.com/v1/namespaces/default/nicports/nicport' \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key \
--data '{"metadata":{"labels":{"vmi-target":"test"}}, "spec":{"vmi":"test"}}'
```

绑定后通过查看可以看到NicPort的状态变为phase1，且有networks字段。还可以查看vm的 annotations 可以看到基于该networks字段的信息。需要注意，vm有关联的Pod存在，所以查看vm时需要查看该Pod，如通过kubectl describe Pod virt-launcher-testvm-lltgc可以看到如下信息：

```
Annotations:
    attachnet10.mec-nets.ovn.kubernetes.io/allocated: true
    attachnet10.mec-nets.ovn.kubernetes.io/cidr: 192.168.100.0/24
    attachnet10.mec-nets.ovn.kubernetes.io/gateway: 192.168.100.1
    attachnet10.mec-nets.ovn.kubernetes.io/ip_address: 192.168.100.200
    attachnet10.mec-nets.ovn.kubernetes.io/logical_router: vpc-bj1-sg
    attachnet10.mec-nets.ovn.kubernetes.io/logical_switch: subnet1-bj1
    attachnet10.mec-nets.ovn.kubernetes.io/mac_address: 9e:13:e7:31:56:1d
    attachnet10.mec-nets.ovn.kubernetes.io/Pod_nic_type: veth-pair
    attachnet10.mec-nets.ovn.kubernetes.io/routed: true
    K8s.v1.cni.cncf.io/networks:
    [{"interface":"net1","name":"attachnet1","namespace":"mecnets"},{"interface":"","name":"attachnet10"
,"namespace":"mec-nets"}]
```

卸载后NicPort的状态将改回phase0.1。卸载NicPort的示例如下：

```
curl -v -XPATCH -H "Accept: application/json" -H "Content-Type:
application/merge-patch+json" 'https://ecsvip.lab.ecs.io:6443/apis/kubevirt.ch
inaunicom.com/v1/namespaces/default/nicports/nicport' \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key \
```

```
--data '{"metadata":{"labels":{"vmi-target":""}}, "spec":{"vmi":"test"}}'
```

方式二：使用 kubectl

绑定和解绑NicPort可以使用Kubectl edit nicport <nicport_name>命令。绑定时，修改spec里vmi-target字段，添加虚拟机名称，为NicPort增加label，保存后即可。解绑时，修改spec里vmi-target字段，删除虚拟机名称，删除NicPort的label，保存后即可。检查点和方式一相同。

删除NicPort

删除前需要先从虚拟机卸载。卸载方法见更新NicPort。

方式一：使用 API

挂载NicPort的示例如下：

```
curl -v -XDELETE -H "Accept: application/json" -H "Content-Type:
application/json" 'https://ecsvip.lab.ecs.io:6443/apis/kubevirt.chinaunicom.co
m/v1/namespaces/default/nicports/nicport1' \
--cacert /etc/kubernetes/pki/ca.crt \
--cert /etc/kubernetes/pki/apiserver-kubelet-client.crt \
--key /etc/kubernetes/pki/apiserver-kubelet-client.key
```

方式二：使用 kubectl

通过创建NicPort时使用的yaml文件删除，使用Kubectl delete -f nicport1.yaml命令即可。

2.4.5 Macvtap 对接的实现

本节描述的的 macvtap 网卡直通方案比社区原生 bridge 方案，网络吞吐性能可以提升约 50%。

Kubevirt + kube-OVN bridge方案网络连接如下图所示：

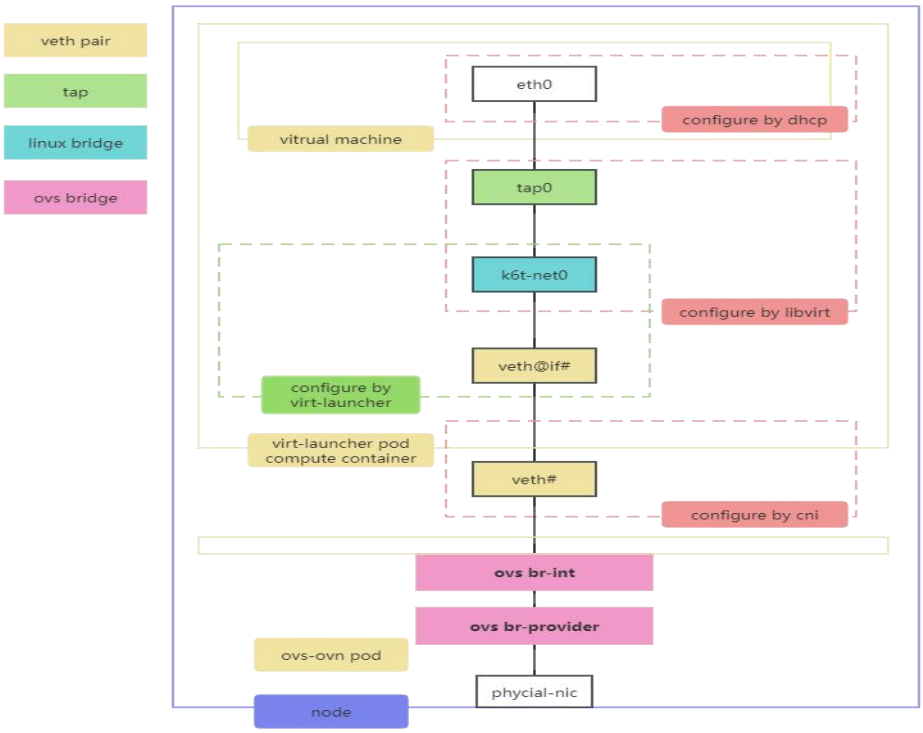


图11：Kubevirt + kube-OVN bridge方案网络连接

1、Libvirt运行在virt-launcher Pod computer容器中；

2、kube-OVN CNI创建veth pair对，一个veth设备挂接到OVS网桥上，另一个veth设备绑定到virt-launcher网络命名空间中；

3、在computer容器中会创建linux网桥 k6t-net0, VM的tap设备挂接到网桥上, 同时也将veth挂接到linux 网桥上，从而打通了VM到OVS网桥的数据通路。

从架构图上可以清晰的看出，VM的流量到容器网络的链路较长，中间需要通过多次内核协议栈，性能有很大的不必要的损耗。

为优化网络性能，可以使用macvtap网卡直通方案，即通过VM使用macvtap网卡，直接缩短VM到容器网络的数据链路，实现架构如下图所示：

1、Libvirt运行在virt-launcher Pod computer容器中；

2、kube-ovn CNI创建OVS internal-port，并以该internal-port创建macvtap网卡，将该macvtap绑定到virt-launcher网络命名空间中；

3、在computer容器中直接只用该macvtap网卡启动VM，从而打通了VM到OVS网桥的数据通路。

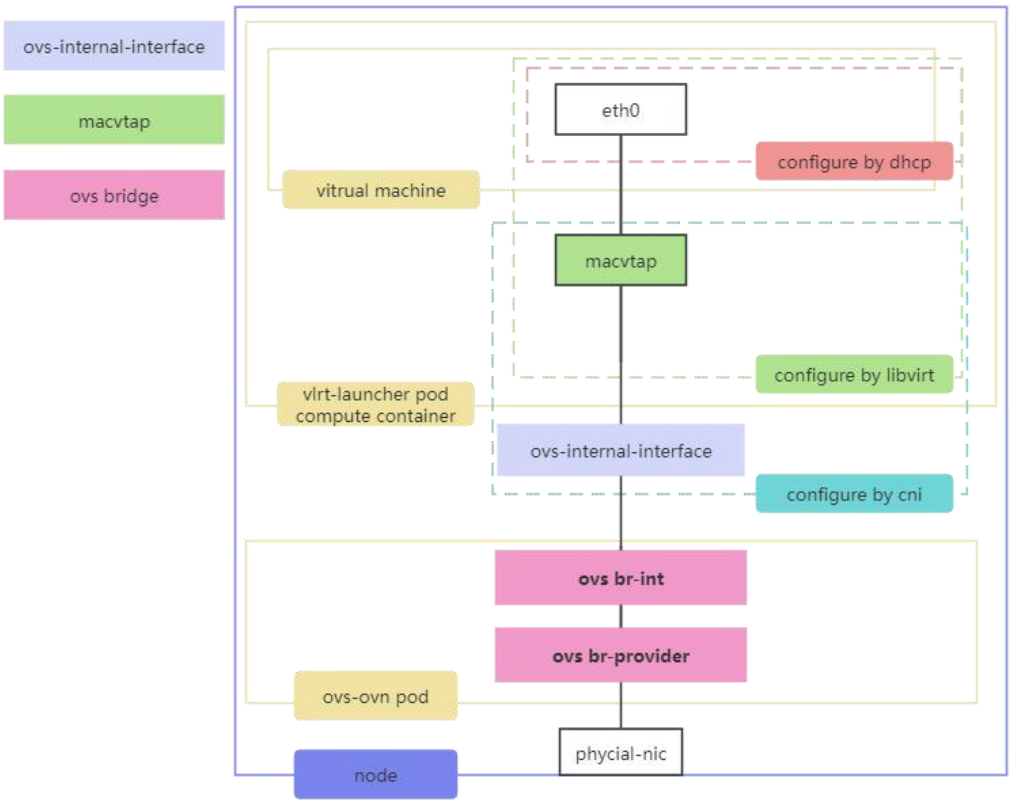


图11：macvtap网卡直通方案网络连接

参 考 文 献

[1] An updated performance comparison of virtual machines and Linux containers https://dominoweb.draco.res.ibm.com/reports/rc25482.pdf
[2] OpenStack: https://www.OpenStack.org
[3] Kubernetes: https://kubernetes.io
[4] Kubevirt: https://kubevirt.io
[5] Kata Container: https://katacontainers.io
[6] Kube-OVN: https://www.kube-ovn.io
[7] Harbor: https://goharbor.io
[8] OpenEbs: https://openebs.io