
[Alauda Container Platform](#) > [虚拟化](#) > 虚拟化

虚拟化

概览

介绍

容器编排虚拟机解决方案

特性

产品特性

控制台特性与 API 对象的映射

架构和权限

限制与约束

概念与术语表

概念与术语表

工作方式

计算对象

镜像和磁盘

网络

迁移与可用性

备份与恢复

从哪里开始

安装

安装

前提条件

规划和要求

操作步骤

资源配额说明

镜像

介绍

优势

从集群外上传镜像

操作指南

权限

实用指南

虚拟机

介绍

可引导卷

注意事项

操作指南

实用指南

从可引导卷创建虚拟机

故障排除

更新或删除可引导卷

网络

介绍

优势

操作指南

实用指南

存储

介绍

优势

操作指南

备份与恢复

介绍

应用场景

使用限制

操作指南

概览

介绍

容器编排虚拟机解决方案

特性

产品特性

控制台特性与 API 对象的映射

架构和权限

限制与约束

介绍

对于采用基于虚拟机架构的企业，迁移到 Kubernetes 和基于容器的架构必然需要进行应用现代化改造。然而，受限于业务需要持续在线或开发习惯难以改变等因素，企业往往无法在短时间内完全脱离虚拟化架构。

因此，一个能够在同一平台上统一配置、管理和控制容器资源与虚拟机资源的解决方案就显得尤为重要。

目录

容器编排虚拟机解决方案

特性

产品特性

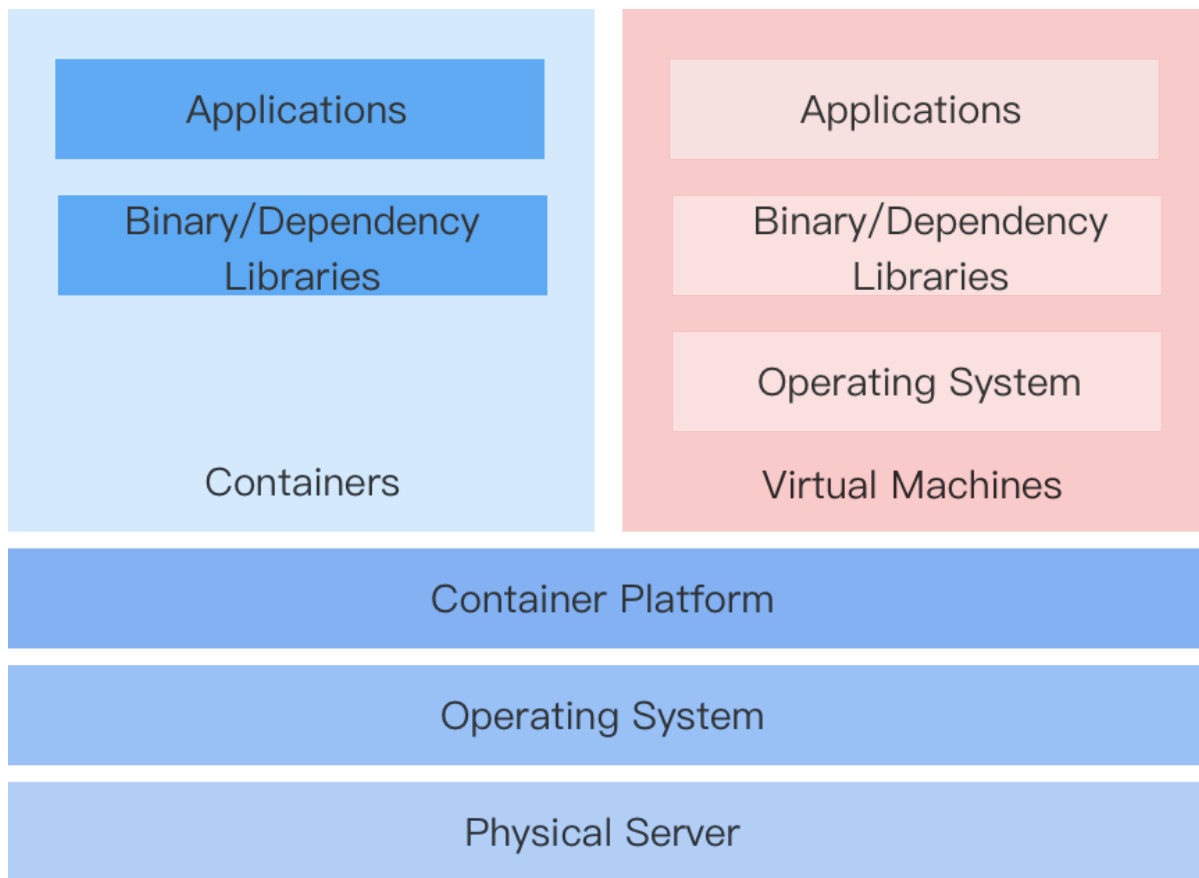
控制台特性与 API 对象的映射

架构和权限

限制与约束

容器编排虚拟机解决方案

本平台基于开源组件 KubeVirt 实现虚拟机（VMI，VirtualMachineInstance）方案，使创建容器编排虚拟机以及运行虚拟化应用更加便捷和高效。



特性

快速转型

无需重写应用或修改镜像。只需将现有应用打包为 qcow2 或 raw 格式的虚拟机镜像，并在平台上使用该镜像创建虚拟机，即可将应用部署到容器平台。

保持使用习惯

容器化虚拟机可以采用与传统虚拟机相似的方式进行管理，而无需关注底层容器实现，包括虚拟机生命周期管理、磁盘和网络以及快照管理。

虚拟化与容器化共存

- 统一平台既支持管理虚拟化服务，也支持基于 Kubernetes 的容器调度与管理。
- 在继续使用虚拟机工作负载的基础上，可逐步推进容器化应用现代化改造。
- 需要与虚拟化应用交互的新容器化应用的开发不受影响。

产品特性

- 虚拟机：支持使用管理员分配的镜像创建虚拟机并进行管理，包括启动和停止虚拟机、管理快照、远程登录虚拟机以及修改虚拟机配置。
- 虚拟磁盘：支持查看和管理当前项目中创建的磁盘信息，包括创建磁盘、查看磁盘名称、存储类、容量以及关联的虚拟机。
- 虚拟机快照：支持查看虚拟机快照的状态、关联的虚拟机以及最近一次回滚时间等详情。
- 虚拟机镜像：支持查看当前项目下的虚拟机镜像信息，包括镜像提供方式和操作系统。
- 密钥对：支持查看和管理当前项目中创建的密钥对，包括创建密钥对以及查看关联的虚拟机列表。
- 可启动卷：跨命名空间管理黄金镜像启动源，支持四种导入来源（HTTP/HTTPS、container registry、clone PVC、clone snapshot）。
- **Live Migration**：在无停机的情况下将运行中的虚拟机在节点之间迁移，并受集群级迁移限制和按命名空间划分的迁移策略约束。
- 存储迁移：在线将运行中虚拟机的磁盘在不同存储类之间迁移。
- 更多生命周期操作：克隆整个虚拟机、暂停和恢复虚拟机，以及打开串口控制台（文本 TTY）或 VNC 控制台。

控制台特性与 API 对象的映射

控制台特性	Kubernetes 资源
虚拟机	<code>VirtualMachine</code> (<code>kubevirt.io/v1</code>)
虚拟磁盘	<code>DataVolume</code> / <code>PVC</code> (<code>cdi.kubevirt.io/v1beta1</code>)
虚拟机快照	<code>VirtualMachineSnapshot</code> (<code>snapshot.kubevirt.io/v1beta1</code>)
可启动卷	<code>DataSource</code> (<code>cdi.kubevirt.io/v1beta1</code>)
克隆	<code>VirtualMachineClone</code> (<code>clone.kubevirt.io/v1beta1</code>)
Live Migration	<code>VirtualMachineInstanceMigration</code> (<code>kubevirt.io/v1</code>)
存储迁移	<code>VirtualMachineStorageMigrationPlan</code> + <code>VirtualMachineStorageMigration</code> (<code>migrations.kubevirt.io/v1alpha1</code>)

控制台特性	Kubernetes 资源
密钥对	<code>Secret</code> (type <code>kubernetes.io/ssh-public-key</code>)
暂停 / 恢复，串口 / VNC 控制台	虚拟机实例子资源 (<code>subresources.kubevirt.io</code>)

架构和权限

平台的虚拟化基于 KubeVirt 和 CDI 构建，由 HyperConverged (HCO) operator 管理：

- **HyperConverged (HCO)** 是唯一事实来源——它负责部署并持续协调 KubeVirt 和 CDI。集群级别的设置（例如 Live Migration 限制）必须配置在 `HyperConverged` 资源上，而不是直接配置在 `KubeVirt` 上。
- **CDI (Containerized Data Importer)** 负责将磁盘镜像导入并克隆到 PVC 中——它是可启动卷和虚拟机磁盘背后的引擎。
- **virt-controller** 负责调度虚拟机，**virt-handler** 运行在每个节点上，**virt-launcher** 则将每个运行中的虚拟机封装在一个 Pod 中。

要委派访问权限，请在命名空间中绑定 KubeVirt 集群角色：

集群角色	授予权限
<code>kubevirt.io:view</code>	对虚拟机和实例的只读访问
<code>kubevirt.io:edit</code>	创建 / 更新 / 删除虚拟机并打开控制台
<code>kubevirt.io:admin</code>	完整的虚拟机管理，包括子资源
<code>kubevirt.io:migrate</code>	触发 Live Migration

CDI 还提供对应的 `cdi.kubevirt.io:view` / `:edit` / `:admin` 角色，用于管理 DataVolumes 和 DataSources。

限制与约束

必须基于物理机集群实现，并且需要在启用虚拟化的情况下将 KubeVirt 组件部署到集群中。平台提供通过 Operator 部署 KubeVirt 组件的能力，以及启用虚拟化的界面，所有相关配置均由平台管理员完成。

概念与术语表

本页面面向熟悉 Kubernetes 但初次接触虚拟化的管理员。手册其余部分都默认使用下列术语；这里会为每个术语定义一次，并给出最接近的 Kubernetes 类比，以便各个操作指南页面保持简洁。

目录

工作方式

计算对象

镜像和磁盘

网络

迁移与可用性

备份与恢复

从哪里开始

工作方式

本平台上的虚拟化由 KubeVirt 和 CDI 提供，并由一个 operator 部署和协调：

- **HyperConverged (HCO)** — [HyperConverged](#) 自定义资源是唯一可信源。operator 会基于它安装并持续协调 KubeVirt 和 CDI，因此集群级设置（例如 live-migration 限制）应配置在 [HyperConverged](#) 上，而不是直接配置在 [KubeVirt](#) 资源上。类比：一个顶层 operator CR，类似于拥有其下所有对象的 [Installation](#) / [Subscription](#)。

- **KubeVirt** — 运行虚拟机。其控制器包括 **virt-controller**（集群级，负责调度 VM）、**virt-handler**（一个 DaemonSet，每个节点一个，类似于 VM 的 kubelet）以及 **virt-launcher**（每个正在运行的 VM 一个 Pod，封装 QEMU 进程）。
- **CDI (Containerized Data Importer)** — 将磁盘镜像导入并克隆到 PVC 中。它是可启动卷和 VM 磁盘背后的引擎。

因此，一个正在运行的虚拟机本质上就是普通节点上的一个普通 **Pod**（virt-launcher Pod），它持有一个普通的 **PVC** 作为磁盘，并获得一个普通的 **Pod IP** —— 这也是为什么你现有的 Kubernetes 工具链（RBAC、NetworkPolicy、调度、监控）仍然适用。

计算对象

术语	它是什么	Kubernetes 类比
VirtualMachine (<code>kubevirt.io/v1</code>)	VM 的期望状态定义（CPU、内存、磁盘、网络，以及是否应处于运行状态）。	<code>Deployment</code> —— 你创建和编辑的规格。
VirtualMachineInstance (VMI)	当 <code>VirtualMachine</code> 启动时生成的 <i>运行中</i> 实例。生命周期子资源（console、VNC、pause、migrate）作用于 VMI。	<code>Pod</code> —— 由规格创建，仅在运行期间存在。
virt-launcher pod	封装某个正在运行的 VM 的 QEMU 进程的 Pod。	实际执行工作负载的 Pod。
<code>spec.running</code>	<code>VirtualMachine</code> 上的布尔电源开关（ <code>true</code> 启动， <code>false</code> 停止）。本平台使用此字段。	<code>replicas: 1</code> 与 <code>0</code> 。
<code>spec.runStrategy</code>	<code>spec.running</code> 的上游替代方案（ <code>Always</code> / <code>RerunOnFailure</code> / <code>Manual</code> / <code>Halted</code> ），用于更细粒度的重启行为。	重启策略。
instancetype	VM 可引用的可复用、集群范围的 CPU/内存配置文件，而不是在资源中内联定义。	尺寸预设 / 类似 <code>LimitRange</code> 的模板。

术语	它是什么	Kubernetes 类比
preference	可复用的 OS 特定设备/固件默认值配置文件（总线类型、时钟、特性）。	应用于 VM 的设备/固件“配置文件”。
QEMU guest agent	安装在 guest OS 内部的 agent。用于启用 IP 上报、优雅关机，以及面向应用一致性快照的文件系统冻结。没有它，平台无法看到 guest IP，且在线快照只能达到崩溃一致性。	运行负载内部的 sidecar/agent，负责报告健康状况并响应信号。参见 安装 Guest Tools 。
vTPM	虚拟 TPM 2.0 设备 (<code>spec.template.spec.domain.devices.tpm</code>)。 Windows 11 和 BitLocker 需要它。	附加到 VM 的虚拟安全设备。

镜像和磁盘

VM 从一个由可复用 **image** 克隆而来的 **disk** 启动。流程如下：

```
remote image (HTTP / registry / S3) —CDI import—► PVC (a DataVolume)
—published as—► DataSource (a "bootable volume") —cloned at VM create—► the VM's boot disk
```

术语	它是什么	Kubernetes 类比
qcow2 / raw	磁盘镜像文件格式。 <code>qcow2</code> 是稀疏/写时复制格式； <code>raw</code> 是扁平镜像。这里指的是你导入的 <i>内容</i> ，不是容器镜像。	一个磁盘镜像制品，例如 OS cloud image。
containerDisk	打包在容器镜像 <i>内部</i> 、并像容器镜像一样拉取的 VM 磁盘。注意：普通应用容器镜像不是 containerDisk，不能直接启动。	其负载是可启动磁盘的 OCI 镜像。
CDI / DataVolume	<code>DataVolume</code> (<code>cdi.kubevirt.io/v1beta1</code>) 声明“将此镜像导入到 PVC 中”；CDI 负责导入并	一个 PVC 加上一个用于填充它的 init job。

术语	它是什么	Kubernetes 类比
	创建底层 PVC。	
DataSource / bootable volume	<code>DataSource</code> (<code>cdi.kubevirt.io/v1beta1</code>) 指向一个 golden 镜像 PVC，也称为 Bootable Volume ；创建 VM 时会将其克隆为 VM 的启动磁盘。	只读的“golden”PVC，作为模板使用。参见 可启动卷 。
volumeMode: Block vs Filesystem	PVC 的呈现方式。 Block 更接近裸金属，且更适合作为 VM 磁盘；Filesystem 也可用。	你已熟悉的 PVC <code>volumeMode</code> 。
StorageProfile / cloneStrategy	CDI 按 StorageClass 发布的对象，用于描述支持的访问/卷模式，以及克隆的实现方式 (<code>csi-clone</code> 、 <code>snapshot</code> 或 host-assisted <code>copy</code>)。	StorageClass 的能力元数据。参见 管理虚拟磁盘 。

新集群在管理员导入某个可启动卷之前，不会有任何可启动卷（或者将 golden 镜像发布到 `kube-public` 中）。请从 [可启动卷](#) 开始。

网络

默认情况下，VM 运行在 virt-launcher **Pod** 中，并使用该 Pod 的网络——因此它会获得一个 Pod IP，并受到与任何 Pod 相同的 CNI 和 `NetworkPolicy` 约束。创建表单中的 **Network Mode** 映射到 KubeVirt 的某种 binding：

UI 标签	YAML binding	行为
Bridged	<code>bridge: {}</code>	VM 接管 Pod 的网络接口，并直接使用 Pod IP 。某些协议需要此模式；默认情况下它不适合 live-migration —— 通过 bridge 进行 live migration 需要 <code>kubevirt.io/allow-pod-bridge-network-live-migration</code> 注解以及受支持的 CNI（参见 Live Migration ）。
NAT	<code>masquerade: {}</code>	VM 获得一个内部私有 IP，并通过 Pod IP 进行 NAT 出口。默认模式；适合 live-migration。

UI 中的“**Container group**”指的是为 VM 提供底层支持的 Kubernetes **Pod**。

如果你需要在物理网络上获得可路由 IP（而不是 Pod IP），请使用 Kube-OVN **Underlay** 子网或辅助 NIC。第二张及后续 NIC 通过 `NetworkAttachmentDefinition`（Multus）挂载。

迁移与可用性

术语	它是什么	Kubernetes
Live migration	通过复制内存，将一个 运行中 的 VM 无停机迁移到另一节点。 Pre-copy 会在 VM 运行时迭代复制内存； post-copy 则在 VM 已到达目标端后再完成剩余复制。	在不重启重新调度常是 Pod 参见 Live Migration
VirtualMachineInstanceMigration	触发一次 live migration 的请求对象；删除它会取消正在进行的迁移。	一个一次迁移 VM
MigrationPolicy	迁移调优的集群范围覆盖项（按 namespace/VM 生效）。集群默认值位于 <code>HyperConverged.spec.liveMigrationConfig</code> 。	通过标签 CR。
evictionStrategy	节点 drain 时的处理方式： <code>LiveMigrate</code> 、 <code>LiveMigrateIfPossible</code> 或 <code>None</code> 。不可迁移的 VM（RWO 存储或设备直通）若设置为 <code>LiveMigrate</code> ，会阻止 drain ，从而阻止集群升级。	类似 <code>PodDisruptionBudget</code> 的约束，不是仅仅可用。
Live migration requires shared storage	只有当 VM 的所有磁盘 PVC 都使用 ReadWriteMany (RWX) 访问模式时，VM 才能进行 live-migrate（例如 Block 模式下的 CephRBD）。	RWX 与相同概念
Storage migration	在线将正在运行的 VM 的磁盘在不同 StorageClass 之间迁移。	在不停机 PVC 重新 StorageClass

备份与恢复

术语	它是什么	Kubernetes 类比
VirtualMachineSnapshot / VirtualMachineRestore	VM 磁盘（以及配置）的某一时刻快照及其恢复对象。由 CSI <code>VolumeSnapshot</code> 提供支持。	作用域限定到整个 VM 的 <code>VolumeSnapshot</code> 。
crash-consistent vs application-consistent	没有 guest agent 时，快照是 <i>crash-consistent</i> （类似直接断电）。有 guest agent 时，会先使文件系统静默，从而得到 <i>application-consistent</i> 快照。	存储备份中常见的 fsfreeze 区别。

从哪里开始

首次阅读手册时可按以下路径进行：

1. [安装](#) — 启用虚拟化并部署 HCO operator。
2. [可启动卷](#) — 导入可用的 OS 镜像（新集群默认没有）。
3. [创建虚拟机](#) — 基于该镜像创建你的第一台 VM。
4. [串行控制台](#) — 登录（cloud image 没有默认密码；请使用你在创建时设置的 SSH key 或密码）。
5. [管理虚拟磁盘](#) — 挂载数据磁盘。

[概述](#) 中还有一张“Console feature → API object”映射表，并且每个操作指南页面都提供一个可直接复制的 **Using the API** 区块。

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > 安装

安装

为了使项目人员能够充分利用容器平台中的虚拟化功能，平台管理员必须执行以下操作来准备虚拟化环境。

目录

前提条件

规划和要求

操作步骤

启用节点虚拟化

操作步骤

部署 Operator

创建 HyperConverged 实例

配置虚拟机超售比（可选）

重要说明

资源配额说明

前提条件

- 下载与您的平台架构对应的 **ACP Virtualization with KubeVirt** 安装包。
 - 使用 上架软件包 机制上架 **ACP Virtualization with KubeVirt** 安装包。
-

- 使用虚拟化功能时，需要提前规划并准备网络和存储环境。

注意：

- 如果需要通过 IP 直接连接虚拟机，集群必须使用 Kube-OVN Underlay 网络模式。您可以参考最佳实践 [Preparing Kube-OVN Underlay Physical Network](#)。
- 建议将 TopoLVM 与 Kubevirt 一起使用，因为它可以提供接近裸机级的性能。如果对性能要求不高，也可以使用 Ceph 分布式存储。

存储产品	说明
TopoLVM	优点：相对轻量，性能较好。 缺点：不能跨节点使用，可靠性较低，无法提供冗余。
Ceph 分布式存储	优点：可跨节点使用，高可用，并且具备冗余能力。 缺点：由于存在磁盘副本，资源利用率较低；性能较差。

- 如果使用 TopoLVM 并配置了多个磁盘，请确保启用虚拟化的节点上的剩余存储容量能够满足多个磁盘的总容量；否则，虚拟机创建将失败。
- 如果使用 Ceph 分布式存储，请确保存储所在网络与虚拟机所在网络之间可以互通。

规划和要求

在启用虚拟化之前，请根据以下要求规划集群：

- CPU** 虚拟化扩展：启用虚拟化的节点必须在固件中开启 Intel VT-x 或 AMD-V。不同节点使用混合 CPU 型号可能会导致无法进行热迁移。
- 热迁移需要共享存储：只有当虚拟机的所有 PVC 都使用 **ReadWriteMany (RWX)** 访问模式时，才能对该虚拟机执行热迁移。对于必须迁移的工作负载，请规划支持 RWX 的存储（例如块模式下的 Ceph RBD）。
- 卷模式：虚拟机磁盘优先使用 **Block** 卷模式，而不是 Filesystem，因为其性能更接近裸金属。
- 快照和克隆：创建快照（以及基于快照的克隆）要求磁盘的 StorageClass 绑定

`VolumeSnapshotClass`。

操作步骤

1 启用节点虚拟化

当自建集群中的节点为物理机时，您可以通过启用或禁用节点虚拟化开关，控制是否允许 Kubernetes 在该节点上调度虚拟机实例（VMIs）。

- 当开关启用时，允许在该物理机节点上调度新的虚拟机；Windows 物理节点不支持启用虚拟化。
- 当开关禁用时，将阻止在该物理机节点上调度新的虚拟机，但不会影响已经在该节点上运行的虚拟机。

操作步骤

1. 进入 管理员。
2. 在左侧导航栏中，点击 集群管理 > 集群。
3. 点击 自建集群名称。
4. 在 节点 选项卡中，点击目标节点右侧的 ⋮，然后选择 启用虚拟化。
5. 点击 确认。

2 部署 Operator

1. 登录，进入 管理员 页面。
2. 点击 **Marketplace > OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **ACP Virtualization with KubeVirt**，点击 安装，进入 安装 **ACP Virtualization with KubeVirt** 页面。

配置参数：

参数	推荐配置
Channel	默认通道为 <code>alpha</code> 。
Installation Mode	<code>Cluster</code> ：集群中的所有命名空间共享一个 Operator 实例进行创建和管理，从而降低资源使用量。

参数	推荐配置
Installation Place	选择 Recommended ，Namespace 仅支持 kubevirt 。
Upgrade Strategy	Manual ：当 Operator Hub 中有新版本时，需要手动确认后才会将 Operator 升级到最新版本。

3 创建 HyperConverged 实例

1. 进入 管理员。
2. 点击 **Marketplace > OperatorHub**。
3. 找到 **ACP Virtualization with KubeVirt**，点击进入 **ACP Virtualization with KubeVirt** 详情页。
4. 点击 所有实例。
5. 在 **HyperConverged** 实例卡片上点击 创建实例。
注意：每个集群只需要创建一个 **HyperConverged** 实例。
6. 切换到 YAML 视图，仅将示例中 `spec.storageImport.insecureRegistries` 字段里指定的 占位符 替换为正确的虚拟机镜像仓库地址，例如：
`192.168.16.214:60080`，其余参数保持默认值。

```
spec:
  storageImport:
    insecureRegistries:
      - placeholder
```

替换结果：

```
spec:
  storageImport:
    insecureRegistries:
      - '192.168.16.214:60080'
```

7. 点击 创建，等待资源列表中自动创建 CDI 和 KubeVirt 类型实例，同时确保 YAML 中显示的 **status.phase** 为 **deployed**，这表示 HyperConverged 实例已成功创建。

另外，您也可以在命令行中应用 `HyperConverged` 资源，并等待其变为

`Available`：

```
kubectl apply -f - <<'EOF'
apiVersion: hco.kubevirt.io/v1beta1
kind: HyperConverged
metadata:
  name: kubevirt-hyperconverged
  namespace: kubevirt
spec: {} # defaults are applied by the operator; tune liveMigrationConfig, storageImport, etc. as needed
EOF

kubectl get hyperconverged kubevirt-hyperconverged -n kubevirt \
  -o jsonpath='{range .status.conditions[?(@.type=="Available")]}{.status}{"\n"}{end}'
```

4 配置虚拟机超售比（可选）

- 在 集群管理 > 集群 中配置虚拟机所在集群的超售比。
- 或者在 项目管理 > 命名空间 中配置虚拟机所在命名空间的超售比。

重要说明

- 虚拟机仅支持 CPU 超售比，推荐配置值为 2 到 4。
- 一旦为虚拟机启用超售比，在创建虚拟机时，容器的请求值（requests）将固定为指定限制值（limits）/ 虚拟机超售比，因此用户通过 YAML 设置的请求值将不再生效。
例如：假设虚拟机的 CPU 资源超售比设置为 4，如果用户在创建虚拟机时指定 CPU 限制值为 4c，则 CPU 请求值为 $4c/4 = 1c$ 。

资源配额说明

虚拟机的内存资源配额受其所在命名空间的内存资源配额限制。由于承载虚拟机的 Pod 的内存通常大于虚拟机实际可用内存，建议预留 20% 的资源。当命名空间中剩余可用资源低于 20% 时，请及时扩容资源。



Alauda Container Platform > 虚拟化 > 虚拟化 > 镜像

镜像

介绍

介绍

优势

从集群外上传镜像

操作指南

添加虚拟机镜像

操作步骤

更新/删除虚拟机镜像

更新/删除镜像

实用指南

上传本地镜像并创建虚拟机

前提条件

操作步骤

使用 KubeVirt 基于 ISO 创建 Windows 虚拟机

前提条件

限制

使用 KubeVirt 创建 Linux 虚拟机

前提条件

限制与约束

创建 Windows 虚拟机

操作步骤

操作步骤

远程访问

导出虚拟机镜像

操作步骤

权限

权限

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [镜像](#) > 介绍

介绍

Alauda Container Platform Virtualization with KubeVirt 利用 Kubernetes 扩展 API 的能力，将虚拟机镜像抽象为 **Custom Resource Definition** (CRD)。它为用户提供了一个用户界面 (UI)，以便将存储在远程仓库中的虚拟机镜像轻松导入 ACP 以供使用。

目录

优势

从集群外上传镜像

优势

- 支持主流操作系统
支持多种常用的 Linux 发行版和 Windows 操作系统。
- 多架构支持
兼容 **X86_64** 和 **ARM64** 架构。
- 多源支持
允许从以下来源导入虚拟机镜像：
 - 镜像仓库
 - 文件服务器
 - 兼容 S3 的对象存储

- 多格式支持
支持 **QCOW2** 和 **RAW** 格式的虚拟机镜像。

从集群外上传镜像

如果要将本地镜像文件上传到平台（而不是从远程源导入），请使用 `virtctl image-upload`，它会将文件流式传输到 CDI 上传代理。`cdi-uploadproxy` Service（位于 `kubevirt` 命名空间中）必须能够从你运行该命令的位置访问——你可以通过 Gateway API、Ingress 或 `NodePort` / `LoadBalancer` Service 将其暴露出来，以便管理员可以从集群外上传镜像。有关完整操作步骤，包括暴露代理以及使用上传的镜像创建虚拟机，请参见 [上传本地镜像并创建虚拟机](#)；有关 `virtctl` 参考，请参见 [使用 virtctl 命令行工具](#)。

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [镜像](#) > 操作指南

操作指南

添加虚拟机镜像

操作步骤

更新/删除虚拟机镜像

更新/删除镜像

添加虚拟机镜像

该平台支持添加 **X86_64** 和 **ARM64 (Alpha)** 架构的虚拟机镜像，便于开发者快速为现有服务创建虚拟机，并促进业务系统迁移。

目录

操作步骤

操作步骤

1. 进入 管理员。
2. 在左侧导航栏中，点击 虚拟化管理 > 虚拟机镜像。
3. 点击 添加虚拟机镜像。
4. 根据以下说明配置相关参数。

参数	说明
Provisioning Method	目前仅支持 公共镜像 方式，表示添加的镜像可用于已分配的项目。

参数	说明
Operating System	支持的操作系统包括： CentOS/Ubuntu/RedHat/Debian/TLinux/Other Linux/Windows (Alpha) 。 支持的系统架构为： X86_64 和 ARM64 (Alpha) 。
Source	• Image Repository：存储在容器镜像仓库中的虚拟机镜像。 • HTTP：通过 HTTP 协议存储在文件服务器上的虚拟机镜像。 • Object Storage (S3)：可通过对象存储协议 (S3) 获取的虚拟机镜像。如果不需要认证，请使用 HTTP 作为来源。
CPU Architecture	标记 CPU 架构信息。对于镜像仓库来源，支持多选；对于其他来源，仅支持单选。
Image Address	支持 KVM 虚拟机镜像，包括 qcow2/raw 格式。 • 如果来源为镜像仓库，请输入 <code>repository_address:image_version</code>，例如 <code>registry.example.com/library/ubuntu:latest</code>。 • 如果来源为 HTTP，请输入镜像文件 URL，必须以 <code>http://</code> 或 <code>https://</code> 开头，例如 <code>http://192.168.0.1/vm_image/centos_7.8.qcow2</code>。 • 如果来源为对象存储 (S3)，请输入可通过对象存储协议 (S3) 获取的镜像地址，例如 <code>https://endpoint/bucket/centos.qcow2</code>。
Authentication	根据镜像仓库是否需要认证，可将开关切换为开启或关闭。启用后，您可以选择现有镜像凭证或点击 添加凭证，目前仅支持 用户名/密码 类型凭证。 注意：当来源为对象存储 (S3) 时，无法关闭认证。
Assigned Project	为该镜像分配项目使用权限。 • All Projects：将该镜像的使用权限分配给所有项目。 • Specific Project：将该镜像的使用权限分配给指定项目。

参数	说明
	No Assignment：暂不分配给任何项目。镜像创建后，可通过 Update Image 操作进行分配。

5. 点击 添加。

更新/删除虚拟机镜像

1. 导航到 管理员。
2. 在左侧边栏中，单击 虚拟化管理 > 虚拟机镜像。
3. 单击 ⋮ > 更新/删除。
4. 确认后，单击 更新/删除。

更新/删除镜像凭据

1. 转到 [管理员](#)。
2. 在左侧导航栏中，单击 [虚拟化管理](#) > [虚拟机镜像](#)。
3. 在 [镜像凭据](#) 选项卡中，单击：[更新/删除](#)。
4. 确认后，单击 [更新/删除](#)。



Alauda Container Platform > 虚拟化 > 虚拟化 > 镜像 > 实用指南

实用指南

上传本地镜像并创建虚拟机

前提条件

操作步骤

创建 Windows 虚拟机

使用 KubeVirt 基于 ISO 创建 Wir 使用 KubeV

前提条件

限制

操作步骤

远程访问

前提条件

限制与约束

操作步骤

导出虚拟机镜像

操作步骤

上传本地镜像并创建虚拟机

当你已经在本地机器上有一个虚拟机磁盘镜像时，请使用此操作步骤（例如从操作系统厂商下载的 `qcow2`、`raw` 或 `img` 文件），并希望基于该镜像运行虚拟机，而不是从远程 HTTP、registry 或 S3 源导入镜像。

在 Alauda Container Platform 上，这是一个基于 KubeVirt `virtctl` 客户端和 CDI upload proxy 的命令行工作流——Web 控制台目前还不支持本地上传来源。上传后的镜像会以 `DataSource`（一个可启动卷）的形式存储，虚拟机可以从中启动，并且可被多个虚拟机复用。

提示：如果你希望改为从 ISO 安装操作系统，请参见 [基于 ISO 创建 Linux 镜像](#) 和 [基于 ISO 创建 Windows 镜像](#)。

目录

前提条件

操作步骤

在集群外暴露 CDI upload proxy

将 `virtctl` 指向 upload proxy URL

上传本地镜像

从已上传镜像创建虚拟机

创建 Windows 虚拟机

前提条件

- 已安装 `virtctl` 命令行客户端，并且其版本与集群的 KubeVirt 版本一致。`virtctl` 是标准的 KubeVirt CLI；请从 KubeVirt releases 下载与你的集群匹配的二进制文件，并将其放到 `PATH` 中。
- 已安装并配置好 `kubectl` 命令行工具，以便访问集群。
- 本地存在一个 `qcow2`、`raw` 或 `img` 格式的磁盘镜像。为了加快上传速度，可先使用 `virt-sparsify`、`xz` 或 `gzip` 压缩镜像。
- 一个支持 `ReadWriteOnce`（RWO）访问模式的 `StorageClass`。
- `cdi-uploadproxy` Service（位于 `kubevirt` 命名空间）可从运行 `virtctl` 的位置访问——见步骤 1。

操作步骤

1 在集群外暴露 CDI upload proxy

`virtctl image-upload` 会将镜像流式传输到 `cdi-uploadproxy` Service。该 Service 是 `kubevirt` 命名空间中的内部 `ClusterIP` Service。你必须让它能从运行 `virtctl` 的位置访问。

注意：在 OpenShift 上，CDI 会自动发现指向 `cdi-uploadproxy` 的 OpenShift `Route`，并为你填充 upload URL。Alauda Container Platform 运行在上游 Kubernetes 上，没有 `Route` 资源，因此需要你自己暴露该 Service（此步骤），并显式设置 upload URL（步骤 2）。

使用 **TLS passthrough**，以便 `virtctl` 直接与 `cdi-uploadproxy` 协商 TLS（该 proxy 提供自签名证书，因此客户端需要传递 `--insecure`）。对于下面的每种方式，后端都应为 `kubevirt` 命名空间中 443 端口上的 `cdi-uploadproxy` Service；请注意生成的外部 URL，以便在步骤 2 中使用。

- **Gateway API (Envoy Gateway)** ——推荐。为 `Gateway` 添加一个处于 `Passthrough` 模式的 `TLS` listener，然后创建一个 `TLSRoute`，其后端为 443 端口上的 `cdi-uploadproxy` Service。Gateway 的外部地址——由 Gateway 的 Service Type 设置的 `LoadBalancer` IP 或 `NodePort` ——将成为 upload URL。完整操作步

骤请参见 [配置 GatewayAPI Gateway](#) 和 [配置 GatewayAPI Route](#)，并参见 [Envoy Gateway Operator](#) 以安装控制器。

- **Ingress (ingress-nginx)**。使用启用 SSL passthrough 的 `Ingress` 暴露 `cdi-uploadproxy`。请参见 [配置 Ingress](#) 和 [Ingress Nginx Operator](#)。旧版 ALB ingress (`cpaas.io/alb2`) 已弃用。
- **LoadBalancer 或 NodePort Service**。如果没有 ingress controller，可以直接暴露该 Service——在配置了 MetalLB 的环境中使用 `LoadBalancer`，或者使用始终可用的 `NodePort`：

```
kubectl expose service cdi-uploadproxy -n kubevirt \
  --name=cdi-uploadproxy-np --type=NodePort --port=443 --target-port=8443
kubectl get service cdi-uploadproxy-np -n kubevirt # note the 443:
3:<nodePort> mapping
```

通过 `https://<node-ip>:<nodePort>` 访问该 proxy。

暴露 Service 后，请验证它是否能通过新的 endpoint 正常响应（证书是自签名的，因此请使用 `-k`）：

```
curl -k https://<upload-proxy-host>/healthz
# -> OK
```

2 将 `virtctl` 指向 upload proxy URL

`virtctl image-upload` 需要知道步骤 1 中的外部 URL。以下两种方式任选其一。

一次性配置（持久化）。设置 `CDIConfig.spec.uploadProxyURLOverride`，这样每次执行 `virtctl image-upload` 时都会自动发现该 URL。由于 CDI 配置由 HyperConverged (HCO) operator 管理——它会将直接修改回滚，并且不会暴露此字段——因此需要通过 HCO 的 jsonpatch annotation 来设置：

```
kubectl annotate hyperconverged kubevirt-hyperconverged -n kubevirt \
  'containerizeddataimporter.kubevirt.io/jsonpatch=[{"op":"add","path":"/spec/config/uploadProxyURLOverride","value":"https://cdi-uploadproxy.example.com:8443"}]' \
  --overwrite

# Verify it propagated to the calculated URL that virtctl reads:
kubectl get cdiconfig config -o jsonpath='{.status.uploadProxyURL}'
```

警告：HCO 将 jsonpatch annotation 说明为一种高级、不受支持的机制——错误的 patch 可能会使虚拟化栈不稳定。请谨慎使用，并在需要回退时删除该 annotation (`kubectl annotate hyperconverged kubevirt-hyperconverged -n kubevirt containerizeddataimporter.kubevirt.io/jsonpatch-`)。

或者按命令传入。跳过集群配置，在每次上传时通过 `--uploadproxy-url` 提供该 URL (见步骤 3)。

3 上传本地镜像

运行 `virtctl image-upload`。 `--datasource` 标志还会创建一个指向已上传卷的 `DataSource`，使其可被选作可启动卷：

```
virtctl image-upload dv uploaded-image \
  --datasource \
  --size=30Gi \
  --storage-class=rbd-1 \
  --image-path=./my-image.qcow2 \
  --namespace=my-namespace
# add --uploadproxy-url=https://cdi-uploadproxy.example.com:8443 --in
secure
# if you did NOT set uploadProxyURLOverride in Step 2.
```

说明：

- `--size` 必须大于镜像的虚拟磁盘大小（不是文件大小）。例如，如果你的镜像有一个 24 GiB 的虚拟磁盘，请至少请求 30 GiB 的卷。CDI 会拒绝小于未压缩磁盘大小的卷。

- 使用 `--no-create`（并省略 `--size`）可以上传到一个已存在的 `DataVolume` 中。
- `--insecure` 会跳过对 proxy 自签名证书的验证。

等待上传和导入完成：

```
kubectl get dv uploaded-image -n my-namespace -w
# PHASE: UploadReady -> ... -> Succeeded

kubectl get datasource uploaded-image -n my-namespace
# the DataSource becomes Ready
```

注意：使用 `virtctl --datasource` 创建的 `DataSource` 不包含用于标记可启动卷的 `virtualization.cpaas.io/*` labels。该镜像可完全通过 API 使用（通过 `sourceRef`，如下一步所示）；如果还希望应用该 label 约定，请参见 [Bootable Volumes](#)。

4 从已上传镜像创建虚拟机

通过 `sourceRef` 在 `dataVolumeTemplates` 条目中引用该 `DataSource`。KubeVirt 会将已上传卷克隆为新虚拟机的一块新磁盘：


```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: example-vm
  namespace: my-namespace
spec:
  runStrategy: RerunOnFailure
  dataVolumeTemplates:
    - metadata:
        name: example-vm-rootdisk
      spec:
        sourceRef:
          kind: DataSource
          name: uploaded-image
          namespace: my-namespace
        storage:
          resources:
            requests:
              storage: 30Gi
          storageClassName: rbd-1
  template:
    metadata:
      labels:
        kubevirt.io/vm: example-vm
    spec:
      domain:
        cpu:
          cores: 1
        resources:
          requests:
            memory: 2Gi
        devices:
          disks:
            - name: rootdisk
              disk:
                bus: virtio
          interfaces:
            - name: default
              masquerade: {}
      networks:
        - name: default
          pod: {}
      volumes:
```

```
- name: rootdisk
  dataVolume:
    name: example-vm-rootdisk
```

```
kubectl apply -f example-vm.yaml
kubectl get vm example-vm -n my-namespace      # STATUS: Running
kubectl get vmi example-vm -n my-namespace     # PHASE: Running,
with an IP
```

打开串行控制台确认操作系统已启动；出现登录提示符表示启动成功：

```
virtctl console example-vm -n my-namespace
# example-vm login:
```

提示：在满足所需 labels 后，也可以在创建虚拟机时将已上传镜像作为可启动卷引用——参见 [创建虚拟机](#) 和 [Bootable Volumes](#)。

创建 Windows 虚拟机

对于 Windows，先将镜像上传到卷中，然后在创建虚拟机时对其进行克隆，并在首次启动期间应用 `autounattend.xml` 应答文件。上传步骤与上面的操作步骤完全相同（使用 `qcow2` / `raw` / `img` 格式的 Windows 磁盘镜像）；关于基于 ISO 的安装流程和 VirtIO 驱动要求，请参见 [基于 ISO 创建 Windows 镜像](#)。

使用 KubeVirt 基于 ISO 创建 Windows 镜像

本文介绍了一个基于开源组件 KubeVirt 的虚拟机方案，使用 KubeVirt 虚拟化技术通过 ISO 镜像文件创建 Windows 操作系统镜像。ISO 文件通过 CDI DataVolume 上传到集群中，无需构建并将容器镜像推送到镜像仓库。

目录

前提条件

限制

操作步骤

将 ISO 文件上传到 DataVolume

创建虚拟机

安装 Windows 操作系统

安装 virtio-win-tools

导出自定义 Windows 镜像

使用 Windows 镜像

添加内部路由

远程访问

前提条件

- 集群中的所有组件均正常运行。

- 请提前准备好 Windows 镜像和[最新的 virtio-win-tools](#)。
- 已安装并配置好可访问集群的 `kubectl` 命令行工具。
- 集群中已提供支持 `ReadWriteMany` (RWX) 访问模式的 StorageClass。热迁移要求附加到虚拟机的每个 PVC 都必须可共享，因此本文中的所有 DataVolume 都使用 RWX。如果不需要热迁移，`ReadWriteOnce` (RWO) 也可以使用。

限制

- 启动 KubeVirt 时，自定义镜像文件系统的大小会影响将镜像写入 PVC 中磁盘的速度。如果文件系统过大，可能会导致创建时间延长。
- 建议将 Windows C 盘保持在 100G 以下，以尽量减小初始大小。Windows 系统在创建完成后，如需扩容必须手动执行。

操作步骤

1 将 ISO 文件上传到 DataVolume

通过 CDI 上传机制将 Windows ISO 和 virtio-win ISO 文件直接上传到集群存储。

配置 **CDI** 上传代理

1. 为 CDI 上传代理设置端口转发。此端口转发会话将用于上传两个 ISO 文件。`cdi-uploadproxy` Service 运行在安装虚拟化组件的命名空间中，默认是 `kubevirt`。

```
kubectl port-forward -n kubevirt svc/cdi-uploadproxy 8443:443 &
```

注意：每次上传都需要各自的认证令牌，该令牌通过 CDI 的 `UploadTokenRequest` 获取。令牌仅授权向一个特定 **PVC** 上传，并且有效期为 **5** 分钟，因此下面每个上传步骤都会在传输开始前立即请求令牌，而不是一次性预先获取。

上传 **Windows ISO**

1. 将以下 YAML 保存为 `dv-win-iso.yaml`，创建一个用于上传 Windows ISO 的 DataVolume。请根据实际 ISO 文件大小调整 `storage` 大小（Windows ISO 通常为

4–6 GB) 。

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: win-iso-dv
  namespace: default
  annotations:
    cdi.kubevirt.io/storage.bind.immediate.requested: "true"
spec:
  source:
    upload: {}
  storage:
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 8Gi
    storageClassName: vm-cephrbd # Replace with your actual StorageClass
    volumeMode: Block
```

2. 执行以下命令创建 DataVolume。

```
kubectl apply -f dv-win-iso.yaml
```

3. 等待 DataVolume 进入 `UploadReady` 阶段。

```
kubectl get dv win-iso-dv -w
# The PHASE column should show UploadReady
```

4. 为 `win-iso-dv` PVC 请求上传令牌。该令牌有效期为 5 分钟，因此请在上传前立即执行。

```
TOKEN=$(kubectl create -o jsonpath='{.status.token}' -f - <<'EOF'
apiVersion: upload.cdi.kubevirt.io/v1beta1
kind: UploadTokenRequest
metadata:
  name: win-iso-dv-upload
  namespace: default
spec:
  pvcName: win-iso-dv
EOF
)
```

5. 使用 curl 上传 Windows ISO 文件。请将文件路径替换为 ISO 的实际路径。

```
curl -v --insecure \
  -H "Authorization: Bearer ${TOKEN}" \
  --data-binary @/path/to/en_windows_server_2019_x64_dvd_4cb967d8.
iso \
  "https://localhost:8443/v1beta1/upload"
```

6. 验证 DataVolume 状态是否变为 **Succeeded**。

```
kubectl get dv win-iso-dv
# The PHASE column should show Succeeded
```

上传 virtio-win ISO

1. 将以下 YAML 保存为 **dv-virtio-iso.yaml**，创建一个用于上传 virtio-win ISO 的 DataVolume。

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: virtio-iso-dv
  namespace: default
  annotations:
    cdi.kubevirt.io/storage.bind.immediate.requested: "true"
spec:
  source:
    upload: {}
  storage:
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 1Gi
    storageClassName: vm-cephrbd # Replace with your actual StorageClass
    volumeMode: Block
```

2. 执行以下命令创建 DataVolume。

```
kubectl apply -f dv-virtio-iso.yaml
```

3. 等待 DataVolume 进入 `UploadReady` 阶段。

```
kubectl get dv virtio-iso-dv -w
# The PHASE column should show UploadReady
```

4. 为 `virtio-iso-dv` PVC 请求上传令牌。由于一个令牌只允许上传到单个 PVC，因此这与 Windows ISO 使用的令牌不同。

```
TOKEN=$(kubectl create -o jsonpath='{.status.token}' -f - <<'EOF'
apiVersion: upload.cdi.kubevirt.io/v1beta1
kind: UploadTokenRequest
metadata:
  name: virtio-iso-dv-upload
  namespace: default
spec:
  pvcName: virtio-iso-dv
EOF
)
```

5. 使用 curl 上传 virtio-win ISO 文件。

```
curl -v --insecure \
  -H "Authorization: Bearer ${TOKEN}" \
  --data-binary @/path/to/virtio-win.iso \
  "https://localhost:8443/v1beta1/upload"
```

6. 验证 DataVolume 状态是否变为 `Succeeded`。

```
kubectl get dv virtio-iso-dv
# The PHASE column should show Succeeded
```

注意：`--insecure` 标志用于跳过自签名证书验证。在生产环境中，请配置正确的证书。大文件上传可能需要较长时间，请确保网络稳定。

2 创建虚拟机

1. 进入容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟机。
3. 点击创建虚拟机。
4. 在表单页面中填写必要参数，例如名称、镜像等。有关详细参数和配置，请参阅[创建虚拟机](#)。
5. 切换到 YAML。
6. 使用以下内容替换 `spec.template.spec.domain.devices` 字段下的配置。USB tablet 输入设备是 VNC 控制台中正确定位鼠标所必需的（参见 [kubevirt#2392](#)、

[kubevirt#3474](#))。如果没有它，鼠标指针会发生偏移，因为 VNC 使用绝对坐标，而默认的 PS/2 鼠标只支持相对坐标。

```
domain:
  devices:
    inputs:
      - type: tablet
        bus: usb
        name: tablet
    disks:
      - disk:
          bus: virtio
          name: cloudinitdisk
      - bootOrder: 1
        cdrom:
          bus: sata
          name: win-iso
      - cdrom:
          bus: sata
          name: virtio-iso
      - disk:
          bus: sata
          name: rootfs
          bootOrder: 10
```

7. 使用以下内容替换 `spec.template.spec.volumes` 字段。两个 ISO 文件都作为 DataVolume 引用，而不是容器镜像。

```

volumes:
  - cloudInitConfigDrive:
      userData: >-
        #cloud-config
        disable_root: false
        ssh_pwauth: true
        users:
          - default
          - name: root
            lock_passwd: false
            hashed_passwd: "<hash>" # Generate with: mkpass
wd --method=SHA-512 --rounds=4096
      name: cloudinitdisk
  - dataVolume:
      name: win-iso-dv
      name: win-iso
  - dataVolume:
      name: virtio-iso-dv
      name: virtio-iso
  - dataVolume:
      name: aa-test-rootfs
      name: rootfs

```

8. 在 `spec.template.metadata.annotations` 下添加以下注解。此虚拟机的主网卡在 pod 网络上使用 `bridge` 绑定，KubeVirt 默认会拒绝对其执行热迁移；该注解声明底层 CNI 能够跟随迁移，从而解除这一限制。平台默认的 CNI Kube-OVN 可以在迁移过程中保持虚拟机 IP 地址不变。如果没有该注解，迁移会因 `InterfaceNotLiveMigratable` 被拒绝。

```

template:
  metadata:
    annotations:
      kubevirt.io/allow-pod-bridge-network-live-migration: "true"

```

注意：KubeVirt 只会报告阻止热迁移的第一个条件，并且接口检查会先于磁盘检查执行。如果仍有任何 PVC 使用 `ReadWriteOnce`，即使添加了该注解，虚拟机仍然不可迁移，此时报告的原因会变为 `DisksNotLiveMigratable`。注解和 RWX 访问模式都必不可少。

9. 检查 YAML 文件。完成配置后的完整 YAML 如下所示。



```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  annotations:
    cpaas.io/creator: test@example.io
    cpaas.io/display-name: ""
    cpaas.io/updated-at: 2024-09-01T14:57:55Z
    kubevirt.io/latest-observed-api-version: v1
    kubevirt.io/storage-observed-api-version: v1
  generation: 16
  labels:
    virtualization.cpaas.io/image-name: debian-2120-x86
    virtualization.cpaas.io/image-os-arch: amd64
    virtualization.cpaas.io/image-os-type: debian
    virtualization.cpaas.io/image-supply-by: public
    vm.cpaas.io/name: aa-test
  name: aa-test
  namespace: default
spec:
  dataVolumeTemplates:
    - metadata:
        creationTimestamp: null
        labels:
          vm.cpaas.io/reclaim-policy: Delete
          vm.cpaas.io/used-by: aa-test
        name: aa-test-rootfs
      spec:
        pvc:
          accessModes:
            - ReadWriteMany
          resources:
            requests:
              storage: 100Gi
          storageClassName: vm-cephrbd
          volumeMode: Block
        source:
          blank: {}
  running: true
  template:
    metadata:
      annotations:
        cpaas.io/creator: test@example.io
        cpaas.io/display-name: ""
```

```
cpaas.io/updated-at: 2024-09-01T14:55:44Z
kubevirt.io/latest-observed-api-version: v1
kubevirt.io/storage-observed-api-version: v1
kubevirt.io/allow-pod-bridge-network-live-migration: "true"

creationTimestamp: null
labels:
  virtualization.cpaas.io/image-name: debian-2120-x86
  virtualization.cpaas.io/image-os-arch: amd64
  virtualization.cpaas.io/image-os-type: debian
  virtualization.cpaas.io/image-supply-by: public
  vm.cpaas.io/name: aa-test
spec:
  affinity:
    nodeAffinity: {}
  architecture: amd64
  domain:
    devices:
      inputs:
        - type: tablet
          bus: usb
          name: tablet
      disks:
        - disk:
            bus: virtio
            name: cloudinitdisk
          bootOrder: 1
          cdrom:
            bus: sata
            name: win-iso
        - cdrom:
            bus: sata
            name: virtio-iso
        - disk:
            bus: sata
            name: rootfs
            bootOrder: 10
      interfaces:
        - bridge: {}
          name: default
  machine:
    type: q35
  resources:
    limits:
```

```

      cpu: "4"
      memory: 8Gi
    requests:
      cpu: "4"
      memory: 8Gi
  networks:
    - name: default
      pod: {}
  nodeSelector:
    kubernetes.io/arch: amd64
    vm.cpaas.io/baremetal: "true"
  volumes:
    - cloudInitConfigDrive:
        userData: >-
          #cloud-config
          disable_root: false
          ssh_pwauth: true
          users:
            - default
            - name: root
              lock_passwd: false
              hashed_passwd: "<hash>" # Generate with: mkpass
wd --method=SHA-512 --rounds=4096
      name: cloudinitdisk
    - dataVolume:
        name: win-iso-dv
        name: win-iso
    - dataVolume:
        name: virtio-iso-dv
        name: virtio-iso
    - dataVolume:
        name: aa-test-rootfs
        name: rootfs

```

10. 点击创建。

11. 点击操作 > **VNC** 登录。

12. 当出现提示 按任意键从 **CD** 或 **DVD** 启动 时，按任意键进入 Windows 安装程序；如果没有看到该提示，请点击页面左上角的发送远程命令，然后在下拉菜单中选择 **Ctrl-Alt-Delete** 以重启服务器。

注意：如果虚拟机详情页面顶部出现消息 当前虚拟机存在需要重启才能生效的配置更改，请重启，可忽略此消息；无需重启。

3 安装 Windows 操作系统

1. 进入安装页面后，按照安装向导完成系统安装。

注意：在分区选择步骤中，磁盘总线必须为 sata，系统才能正确识别磁盘。因此，需要依次选中每个分区并点击删除，清除所有分区，让系统自动处理。

2. 配置管理员账户密码后，点击页面左上角的发送远程命令，然后在下拉菜单中选择 **Ctrl-Alt-Delete**。
3. 当提示 **Ctrl+Alt+Delete** 组合将重启服务器，确认重启 时，点击确定。
4. 输入密码进入 Windows 系统桌面；至此，Windows 操作系统安装完成。

4 安装 virtio-win-tools

该工具主要包含所需驱动程序。

1. 打开文件资源管理器。
2. 双击 **CD Drive(E:) virtio-win-<version>**，运行 **virtio-win-guest-tools** 目录进入安装页面，并按照安装说明进行安装。 `\<version>` 部分请根据实际情况替换。
3. 安装完成后，关闭 Windows 系统电源。

5 导出自定义 Windows 镜像

请参阅[导出虚拟机镜像](#)了解具体操作。

6 使用 Windows 镜像

1. 进入容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟机。
3. 点击创建虚拟机。
4. 在表单页面中填写必要参数。镜像请选择导出的 Windows 镜像。有关详细参数和配置，请参阅[创建虚拟机](#)。
5. （可选）如果使用较新的操作系统，例如 Windows 11，请启用时钟、UEFI、TPM 等功能。切换到 YAML，并使用以下 YAML 文件替换原始 YAML 文件。



```
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstance
metadata:
  labels:
    special: vmi-windows
  name: vmi-windows
spec:
  domain:
    clock:
      timer:
        hpet:
          present: false
        hyperv: {}
        pit:
          tickPolicy: delay
        rtc:
          tickPolicy: catchup
      utc: {}
    cpu:
      cores: 2
    devices:
      inputs:
        - type: tablet
          bus: usb
          name: tablet
      disks:
        - disk:
            bus: sata
            name: pvcdisk
      interfaces:
        - masquerade: {}
          model: e1000
          name: default
      tpm: {}
    features:
      acpi: {}
      apic: {}
      hyperv:
        relaxed: {}
        spinlocks:
          spinlocks: 8191
        vapic: {}
      smm: {}
```

```
firmware:
  bootloader:
    efi:
      secureBoot: true
  uuid: 5d307ca9-b3ef-428c-8861-06e72d69f223
resources:
  requests:
    memory: 4Gi
networks:
- name: default
  pod: {}
terminationGracePeriodSeconds: 0
volumes:
- name: pvcdisk
  persistentVolumeClaim:
    claimName: disk-windows
- name: winiso
  persistentVolumeClaim:
    claimName: win11cd-pvc
```

6. 点击创建。

7 添加内部路由

通过配置主机端口类型的内部路由，为远程桌面连接暴露端口。

1. 进入容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟机。
3. 点击列表中使用 Windows 镜像创建的虚拟机名称，进入详情页。
4. 在登录信息区域，点击内部路由旁边的添加图标。
5. 按照以下说明配置参数。

参数	描述
类型	选择主机端口。
端口	• 协议：选择 TCP。 • 服务端口：使用 3389。 • 虚拟机端口：使用 3389。

参数	描述
	服务端口名称：使用 rdp。

- 6. 点击确定返回详情页。
- 7. 点击登录信息区域中的内部路由链接。
- 8. 保存基本信息区域中的虚拟 IP 信息以及端口区域中的主机端口信息。

远程访问

本文以使用 Windows 操作系统进行远程连接为例。其他操作系统可以使用支持 RDP 协议的软件进行连接。

- 1. 打开远程桌面连接。
- 2. 输入在添加内部路由步骤中保存的虚拟 IP 和主机端口，格式为虚拟 IP:主机端口 ，例如：192.1.1.1:3389 。
- 3. 点击连接。

使用 KubeVirt 基于 ISO 创建 Linux 镜像

本文档介绍了一种基于开源组件 KubeVirt 实现的虚拟机方案。该方案利用 KubeVirt 虚拟化技术，从 ISO 镜像文件创建 Linux 操作系统镜像。ISO 通过 CDI DataVolume 上传到集群，无需再构建并推送容器镜像到镜像仓库。

目录

前提条件

限制与约束

操作步骤

将 Linux ISO 上传到 DataVolume

创建虚拟机

安装 Linux 操作系统

修改 YAML 文件

安装所需软件并修改配置

导出并使用自定义 Linux 镜像

前提条件

- 集群中的所有组件均正常运行。
- 请提前准备好 Linux 镜像。本文档以 [Ubuntu 操作系统](#) 为例。

- 已安装并配置 `kubect1` 命令行工具，以便访问集群。
- 集群中存在支持 `ReadWriteOnce` (RWO) 访问模式的 `StorageClass`。

限制与约束

- 启动 KubeVirt 时，自定义镜像的文件系统大小会影响将镜像写入 PVC 磁盘的速度。如果文件系统过大，可能会导致创建时间较长。
- 建议将 Linux 根分区大小控制在 100G 以下，以减小初始大小。在配置 cloud-init 后，创建虚拟机时为根分区分配更大的存储空间，系统将自动进行扩容。

操作步骤

1 将 Linux ISO 上传到 DataVolume

使用 CDI 的上传机制将 ISO 文件直接上传到集群存储，无需构建容器镜像。

1. 将以下 YAML 保存为 `dv-iso-upload.yaml` 文件，创建一个 upload 类型的 `DataVolume`。请根据实际 ISO 文件大小调整 `storage` 大小。

```

apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: iso-upload-dv
  namespace: default
  annotations:
    cdi.kubevirt.io/storage.bind.immediate.requested: "true"
spec:
  source:
    upload: {}
  storage:
    accessModes:
      - ReadWriteOnce
    resources:
      requests:
        storage: 8Gi
    storageClassName: vm-cephrbd # Replace with your actual StorageClass
    volumeMode: Block

```

2. 执行以下命令创建 DataVolume。

```
kubectl apply -f dv-iso-upload.yaml
```

3. 等待 DataVolume 进入 `UploadReady` 阶段。

```

kubectl get dv iso-upload-dv -w
# The PHASE column should show UploadReady

```

4. 为 CDI 上传代理设置端口转发。`cdi-uploadproxy` Service 运行在安装虚拟化组件的命名空间中，默认是 `kubevirt`。

```
kubectl port-forward -n kubevirt svc/cdi-uploadproxy 8443:443 &
```

5. 使用 `UploadTokenRequest` 向 CDI 请求上传令牌。该令牌用于授权向 `iso-upload-dv` PVC 上传，有效期为 5 分钟，因此请在上传前立即执行此步骤。

```
TOKEN=$(kubectl create -o jsonpath='{.status.token}' -f - <<'EOF'
apiVersion: upload.cdi.kubevirt.io/v1beta1
kind: UploadTokenRequest
metadata:
  name: iso-upload-dv-upload
  namespace: default
spec:
  pvcName: iso-upload-dv
EOF
)
```

6. 使用 curl 上传 ISO 文件。请将文件路径替换为实际 ISO 路径。

```
curl -v --insecure \
  -H "Authorization: Bearer ${TOKEN}" \
  --data-binary @/path/to/ubuntu-24.04-live-server-amd64.iso \
  "https://localhost:8443/v1beta1/upload"
```

说明：`--insecure` 标志用于跳过自签名证书验证。在生产环境中，请配置正确的证书。大文件上传可能需要较长时间，请确保网络稳定。

7. 验证 DataVolume 状态已变为 `Succeeded`。

```
kubectl get dv iso-upload-dv
# The PHASE column should show Succeeded
```

2 创建虚拟机

1. 进入容器平台。
2. 在左侧导航栏中点击 虚拟化 > 虚拟机。
3. 点击 创建虚拟机。
4. 在表单页面按如下方式填写参数。关于具体参数和配置，请参考 [创建虚拟机](#)。

参数	描述
选择镜像	选择虚拟机的模板镜像。

参数	描述
IP 地址	保持默认值，将通过 DHCP 获取。
网络模式	使用 NAT 模式；此处不要使用 bridged 。

5. 切换到 YAML。

6. 将 `spec.template.spec.domain.devices.disks` 字段下的配置替换为以下内容。ISO DataVolume 会以 CDROM 形式挂载，并具有最高启动优先级，`rootfs` 磁盘用作安装目标。

```
domain:
  devices:
    disks:
      - bootOrder: 1
        cdrom:
          bus: sata
          name: iso-disk
      - disk:
          bus: virtio
          name: cloudinitdisk
      - disk:
          bus: virtio
          name: rootfs
          bootOrder: 10
```

7. 将 `spec.template.spec.volumes` 字段替换为以下内容。ISO 作为 DataVolume 引用，而不是容器镜像。

```
volumes:
  - dataVolume:
      name: iso-upload-dv
      name: iso-disk
  - cloudInitConfigDrive:
      userData: |-
        #cloud-config
        disable_root: false
        ssh_pwauth: false
        users:
          - default
          - name: root
            lock_passwd: false
            hashed_passwd: "<hash>" # Generate with: mkpass
wd --method=SHA-512 --rounds=4096
      name: cloudinitdisk
  - dataVolume:
      name: aa-rootfs
      name: rootfs
```

8. 检查 YAML 文件；完成后的完整 YAML 配置如下。



```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  annotations:
    kubevirt.io/latest-observed-api-version: v1
    kubevirt.io/storage-observed-api-version: v1
  labels:
    virtualization.cpaas.io/image-name: debian-2120-x86
    virtualization.cpaas.io/image-os-arch: amd64
    virtualization.cpaas.io/image-os-type: debian
    virtualization.cpaas.io/image-supply-by: public
    vm.cpaas.io/name: aa
  name: aa
spec:
  dataVolumeTemplates:
    - metadata:
        creationTimestamp: null
        labels:
          vm.cpaas.io/reclaim-policy: Delete
          vm.cpaas.io/used-by: aa
        name: aa-rootfs
      spec:
        pvc:
          accessModes:
            - ReadWriteOnce
          resources:
            requests:
              storage: 100Gi
          storageClassName: vm-cephrbd
          volumeMode: Block
        source:
          blank: {}
  running: true
  template:
    metadata:
      annotations:
        cpaas.io/creator: test@example.io
        cpaas.io/display-name: ""
        cpaas.io/updated-at: 2024-09-09T03:49:08Z
        kubevirt.io/latest-observed-api-version: v1
        kubevirt.io/storage-observed-api-version: v1
      creationTimestamp: null
      labels:
```

```
virtualization.cpaas.io/image-name: debian-2120-x86
virtualization.cpaas.io/image-os-arch: amd64
virtualization.cpaas.io/image-os-type: debian
virtualization.cpaas.io/image-supply-by: public
vm.cpaas.io/name: aa
spec:
  accessCredentials:
    - sshPublicKey:
        propagationMethod:
          qemuGuestAgent:
            users:
              - root
        source:
          secret:
            secretName: test-xeon
  affinity:
    nodeAffinity: {}
  architecture: amd64
  domain:
    devices:
      disks:
        - bootOrder: 1
          cdrom:
            bus: sata
            name: iso-disk
        - disk:
            bus: virtio
            name: cloudinitdisk
        - disk:
            bus: virtio
            name: rootfs
            bootOrder: 10
      interfaces:
        - bridge: {}
          name: default
  machine:
    type: q35
  resources:
    limits:
      cpu: "1"
      memory: 2Gi
    requests:
      cpu: "1"
      memory: 2Gi
```

```

networks:
  - name: default
    pod: {}
nodeSelector:
  kubernetes.io/arch: amd64
  vm.cpaas.io/baremetal: "true"
volumes:
  - dataVolume:
      name: iso-upload-dv
      name: iso-disk
  - cloudInitConfigDrive:
      userData: |-
        #cloud-config
        disable_root: false
        ssh_pwauth: false
        users:
          - default
          - name: root
            lock_passwd: false
            hashed_passwd: "<hash>" # Generate with: mkpass
wd --method=SHA-512 --rounds=4096
      name: cloudinitdisk
  - dataVolume:
      name: aa-rootfs
      name: rootfs

```

9. 点击 创建。

10. 点击 操作 > VNC 登录。

11. 当出现 **press any key boot from CD or DVD** 提示时，按任意键进入安装程序；如果没有看到该提示，请点击页面左上角的 发送远程命令，然后在下拉菜单中点击 **Ctrl-Alt-Delete** 以重启服务器。

说明：如果虚拟机详情页顶部出现 **Current virtual machine has configuration changes that require a restart to take effect. Please restart.** 提示，可以忽略该消息；此时无需重启。

3 安装 Linux 操作系统

1. 进入安装页面后，按照安装向导继续操作。本文档以安装 Ubuntu 操作系统为例，不同操作系统在安装过程中的配置项一般类似，因此不再展开说明。以下对部分配置项进行说明。

配置项	描述
Installation Type	建议使用最小化安装，以尽量减小镜像大小。
Storage Configuration	选择自定义存储。将磁盘格式化为 ext4 或 xfs，并挂载到根分区 (/) 。 说明：磁盘分区不要使用 LVM (Create volume group (LVM)) 。
SSH Configuration	选择安装 OpenSSH 工具，以便通过 SSH 访问。

2. 等待安装完成。

4 修改 YAML 文件

1. 进入容器平台。
2. 在左侧导航栏中点击 虚拟化 > 虚拟机。
3. 点击列表中的虚拟机名称进入详情页。
4. 点击 停止。
5. 点击右上角 操作 > 更新。
6. 切换到 YAML。
7. 确认 spec.template.spec.domain.devices.disks 下名为 **rootfs** 的磁盘其 bootOrder 为 1。如果不是 1，请将其修改为 1。
8. 删除 spec.template.spec.domain.devices.disks 下名为 **iso-disk** 的 ISO 磁盘相关内容；需要删除的具体内容如下。

```
- bootOrder: 1
  cdrom:
    bus: sata
    name: iso-disk
```

9. 删除 spec.template.spec.volumes 下名为 **iso-disk** 的卷相关内容；需要删除的具体内容如下。

```
- dataVolume:  
  name: iso-upload-dv  
  name: iso-disk
```

10. 点击 更新。

11. 点击 启动。

5 安装所需软件并修改配置

说明：以下命令和配置文件在不同操作系统之间可能略有差异；请根据实际环境进行调整。

1. 输入用户名和密码，登录操作系统。
2. 切换到 root 用户权限。
3. 安装软件包。

- 对于 CentOS 系列，执行以下命令：

```
yum install cloud-utils cloud-init qemu-guest-agent vim
```

- 对于 Debian 系列，执行以下命令：

```
apt install cloud-init cloud-guest-utils qemu-guest-agent vim
```

4. 编辑 SSHD 配置文件。

1. 执行以下命令编辑 sshd_config 文件。

```
vim /etc/ssh/sshd_config
```

2. 添加以下配置。

```
PermitRootLogin yes # Allow the root user to log in with a pass  
word  
PubkeyAuthentication yes # Allow key-based login
```

3. 保存修改后的配置。

5. 执行以下命令删除 root 用户的默认密码。

```
passwd -d root
```

6. 修改源地址文件。

1. 执行以下命令修改系统源地址文件，并将地址更改为合适的镜像站点地址。

```
vim /etc/apt/sources.list.d/ubuntu.sources
```

2. 修改完成后保存配置。

7. 修改 cloud-init 配置，以自动扩展根目录。

1. 执行以下命令编辑 cloud.cfg 配置文件。

```
vim /etc/cloud/cloud.cfg
```

2. 添加以下配置内容。

```
runcmd:
  - [growpart, /dev/vda, 1] # The growpart command is used to e
    xtend the partition on the disk, which will extend the /dev/vda1
    partition.
  - [xfs_growfs, /dev/vda1] # The xfs_growfs command is used to
    extend the XFS file system to occupy all available space on the
    partition. /dev/vda1 is the partition where the file system to b
    e extended is located. After extending the partition, using xfs_
    growfs ensures that the file system itself is also expanded to t
    he new partition size.
```

3. 修改完成后保存配置。

8. 完成配置后，关闭操作系统。

6 导出并使用自定义 Linux 镜像

具体操作请参考 [导出虚拟机镜像](#)。

导出虚拟机镜像

该功能用于导出虚拟机的系统镜像并上传至对象存储，允许将对象存储中的文件添加为平台虚拟机镜像的来源。

目录

操作步骤

- 停止虚拟机
- 创建 vmexport 资源
- 下载虚拟机镜像文件
- 将虚拟机镜像文件上传至对象存储
- 创建虚拟机镜像

操作步骤

注意：以下所有操作均需在虚拟机所在集群的控制节点上执行。

1 停止虚拟机

1. 进入 管理员。
2. 在左侧导航栏点击 虚拟化管理 > 虚拟机。
3. 点击需要导出系统镜像的虚拟机名称，跳转至容器平台的虚拟机详情页面。

4. 点击 停止。

2 创建 vmexport 资源

1. 打开 CLI 工具。

2. 执行以下命令设置变量。

```
NAMESPACE=<namespace>
VM_NAME=<vm_name>
TTL_DURATION=2h
```

参数说明：

- **NAMESPACE**：虚拟机所在的命名空间名称；将 `<namespace>` 部分替换为该名称。
- **VM_NAME**：需要导出系统镜像的虚拟机名称；将 `<vm_name>` 部分替换为该名称。
- **TTL_DURATION**：导出任务的存活时间，默认为 2 小时，可根据需要延长。

3. 执行以下命令创建 vmexport 资源。

```
cat <<EOF | kubectl create -f -
apiVersion: export.kubevirt.io/v1alpha1
kind: VirtualMachineExport
metadata:
  name: export-$VM_NAME
  namespace: $NAMESPACE
spec:
  ttlDuration: $TTL_DURATION
  source:
    apiGroup: "kubevirt.io"
    kind: VirtualMachine
    name: $VM_NAME
EOF
```

若出现类似回显信息，表示创建成功。

```
virtualmachineexport.export.kubevirt.io/export-k1 created
```

4. 执行以下命令查看 vmexport 资源状态。

```
kubectl -n $NAMESPACE get vmexport export-$VM_NAME -w
```

回显信息：

NAME	SOURCEKIND	SOURCENAME	PHASE
export-k1	VirtualMachine	k1	Ready

5. 当回显信息中的 PHASE 字段变为 Ready 时，按 ctrl (control) + c 停止 watch 操作。

6. 执行以下命令获取 TOKEN。

```
TOKEN=$(kubectl -n $NAMESPACE get secret export-token-export-$VM_NAME -o jsonpath={.data.token} | base64 -d)
```

3 下载虚拟机镜像文件

1. 执行以下命令获取指定命名空间中虚拟机导出 Pod 的 IP 地址，并存入 EXPORT_SERVER_IP 环境变量。

```
EXPORT_SERVER_IP=$(kubectl -n $NAMESPACE get po virt-export-export-$VM_NAME -o jsonpath='{.status.podIP}')
```

2. 执行以下命令设置 URL 环境变量，指向虚拟机的磁盘镜像文件。

```
URL=https://$EXPORT_SERVER_IP:8443/volumes/$VM_NAME-rootfs/disk.img.gz
```

3. 执行以下命令下载镜像文件，下载后的文件名为 disk.img.gz。

```
curl -k -O -H "x-kubevirt-export-token: $TOKEN" $URL
```

4 将虚拟机镜像文件上传至对象存储

将下载的镜像文件上传至对象存储。上传可使用任意 S3 工具，本文以 mc (minio-client) 工具为例介绍。

1. 执行以下命令配置 mc 工具，连接指定的 S3 存储服务。

```
mc alias set minio <ENDPOINT> <ACCESSKEY> <SECRETKEY>
```

参数说明：

- ENDPOINT：S3 存储服务地址；将 `<ENDPOINT>` 部分替换为该地址。
- ACCESSKEY、SECRETKEY：用于认证的 S3 存储服务用户 ak 和 sk；相关信息请参考 [MinIO Object Storage](#)。

2. 执行以下命令创建用于存储虚拟机镜像文件的桶。

```
mc mb minio/vmdisks
```

3. 执行以下命令将导出的虚拟机镜像文件 disk.img.gz 上传至创建的桶中。

```
mc put disk.img.gz minio/vmdisks
```

5 创建虚拟机镜像

1. 进入 管理员。
2. 在左侧导航栏点击 虚拟化管理 > 虚拟机镜像。
3. 点击 添加虚拟机镜像。
4. 在镜像地址中填写 `<ENDPOINT>/vmdisks/disk.img.gz`，将 `<ENDPOINT>` 部分替换为 S3 存储服务地址。其他参数说明请参考 [添加虚拟机镜像](#)。
5. 点击 添加。

权限

功能	操作	平台管理员	平台审计人员	项目
虚拟机镜像 acp- virtualmachineimagetemplates	查看	✓	✓	
	创建	✓	✕	
	更新	✓	✕	
	删除	✓	✕	

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > 虚拟机

虚拟机

介绍

[介绍](#)

可引导卷

[可引导卷](#)[注意事项](#)[查看可引导卷](#)[添加可引导卷](#)[从可引导卷创建虚拟机](#)[更新或删除可引导卷](#)

操作指南

[创建虚拟机](#)[前提条件](#)[虚拟机批量操作](#)[操作步骤](#)[暂停和恢复虚拟机](#)[注意](#)

注意事项

创建虚拟机

使用 API

使用 API

暂停虚拟机

恢复虚拟机

克隆虚拟机

使用 VNC 登录虚拟机

操作步骤

使用 API

虚拟机控制台

控制台

管理虚拟机

重置密码

更新密钥

更新规格

热迁移

更新 NAT 网络配置

更新标签和注解

添加服务

重装操作系统

配置 IP

使用 API

使用 InstanceType 规格导入的虚拟机

监控和告警

监控

告警

使用 API

管理虚拟机

虚拟机的快速定位

前提条件

操作步骤

使用 API

实用指南

使用 virtctl 命令行工具

控制台和访问

上传和导出镜像

配置 USB 主机透传

功能概述

使用场景

安装来宾工具

QEMU 来宾智能

VirtIO 驱动程序

热插拔磁盘
电源和迁移
使用 libguestfs 编辑和封存镜像

前提条件
步骤
操作结果
了解更多

虚拟 TPM (vTPM)

CPU 和内存热插拔

工作原理
前提条件
第 1 步：创建具有热插拔余量的 VM
第 2 步：热插拔 CPU
第 3 步：热插拔内存
向下扩容：哪些可以在线完成，哪些需要重
已验证行为汇总
故障排除

虚拟机恢复

操作步骤

在 KubeVirt 上克隆虚拟机

确保先决条件
快速开始
理解 VirtualMachineClone 对象

迁移

在 StorageClass 上克隆虚拟机

前提条件
准备迁移

从现有虚拟机创建 VM 模板

前提条件
操作步骤

为虚拟机配置高可用性

fencing 和 remediation 过程中的事件流程
操作步骤

故障排除

Pod 迁移及虚拟机节点异常关闭时

问题描述
原因分析

热迁移错误消息与解决方案

启动和镜像引导

读取来宾启动日志
诊断卡住的镜像

解决方案

捕获虚拟机内存

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [虚拟机](#) > 介绍

介绍

KubeVirt 提供了诸如 **VirtualMachine** 和 **VirtualMachineInstance** 等 CRD（自定义资源定义），用于抽象虚拟机（VM）资源。基于这些 CRD，用户可以获得全面的虚拟机管理能力。在此基础上，**ACP Virtualization With KubeVirt** 通过提供一个 **Web Console**，进一步提升了易用性，使用户能够更轻松地执行各种操作。

可引导卷

可引导卷 是一个可直接使用的引导镜像，虚拟机可基于它创建。每个可引导卷都是一个由 CDI 从导入源或克隆源填充的 PVC 所支持的 KubeVirt `DataSource` (`cdi.kubevirt.io/v1beta1`)。可引导卷会跨命名空间聚合——包括存放平台“golden images”的共享 `kube-public` 命名空间——因此可被多个虚拟机重复使用。

当你创建虚拟机并选择 **Provision Method > Image Instance** 时，选择器会列出可用于启动的可引导卷。

目录

注意事项

查看可引导卷

添加可引导卷

从可引导卷创建虚拟机

更新或删除可引导卷

注意事项

- 只有当其 `DataSource` 带有 `virtualization.cpaas.io/*` 标签契约（操作系统、架构、容量、storage class、volume mode、access mode）时，可引导卷才可用于虚拟机创建。没有这些标签的数据源仍会显示在管理列表中，但不会出现在 **Image Instance** 选择器中。
- 在 CDI 填充后端 PVC 时，该卷的 **Status** 为 `Importing`；一旦可供使用，则为 `Ready`。

- 尚不支持将本地 **Upload** 作为源；请使用 HTTP/HTTPS、Registry、Clone PVC 或 Clone snapshot。

查看可引导卷

可引导卷由 `DataSource` 对象提供支持，并且会跨命名空间聚合（包括共享的 `kube-public` 命名空间）。可直接列出它们：

```
kubectl get datasource -A
```

添加可引导卷

可引导卷支持四种源类型：

Source type	What it does	DataVolume source
HTTP/HTTPS (default)	从 HTTP(S) URL 导入 qcow2/raw 镜像	<code>http</code>
Registry	从容器镜像仓库拉取 container disk	<code>registry</code>
Clone PVC	复制现有 PVC	<code>pvc</code>
Clone snapshot	从 VolumeSnapshot 恢复	<code>snapshot</code>

Note：请在你自己的项目命名空间中创建可引导卷。共享的 `kube-public` 命名空间包含由集群管理员发布的平台 **golden images**——普通项目用户可以直接使用（从中克隆并启动）`kube-public` 镜像，但在 `kube-public` 中创建或修改对象需要更高的平台权限。

添加可引导卷会创建两个对象：一个 `DataVolume`（CDI 会将镜像导入到 PVC 中）以及一个指向它并携带该标签契约的 `DataSource`。以下示例通过 HTTP 导入镜像：

1) DataVolume – CDI imports the image into a PVC of the same name.

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: ubuntu-2204
  namespace: demo
spec:
  source:
    http:
      url: https://images.example.com/ubuntu-22.04.qcow2
      # secretRef: my-pull-secret # for an authenticated source
  storage:
    resources:
      requests:
        storage: 20Gi
    storageClassName: sc-topolvm
    volumeMode: Filesystem
    accessModes:
      - ReadWriteOnce
```

2) DataSource – makes the PVC selectable as a bootable volume.

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataSource
metadata:
  name: ubuntu-2204
  namespace: demo
  labels:
    virtualization.cpaas.io/image-os-type: ubuntu
    virtualization.cpaas.io/image-os-arch: amd64
    virtualization.cpaas.io/size: 20Gi
    virtualization.cpaas.io/storage-class: sc-topolvm
    virtualization.cpaas.io/volume-mode: Filesystem
    virtualization.cpaas.io/access-mode: ReadWriteOnce
    virtualization.cpaas.io/source-origin: url # url | registry | pvc |
snapshot
spec:
  source:
    pvc:
      name: ubuntu-2204
      namespace: demo
```

对于其他源类型，请替换 `DataVolume` 上的 `spec.source`：

```
# Registry
source:
  registry:
    url: docker://registry.example.com/library/fedora:40
# Clone an existing PVC
source:
  pvc:
    name: source-pvc
    namespace: demo
# Restore from a VolumeSnapshot
source:
  snapshot:
    name: source-snapshot
    namespace: demo
```

应用配置并观察导入过程：

```
kubectl apply -f bootable-volume.yaml
kubectl get datavolume ubuntu-2204 -n demo -w
```

从可引导卷创建虚拟机

从可引导卷创建的虚拟机默认使用该可引导卷所在的命名空间（同命名空间克隆）。

在底层，虚拟机通过 `dataVolumeTemplates` 条目引用该数据源——CDI 会将可引导卷克隆到新虚拟机的根磁盘中：

```
spec:
  dataVolumeTemplates:
    - metadata:
        name: web-01-rootfs
      spec:
        sourceRef:
          kind: DataSource
          name: ubuntu-2204
          # The bootable volume's namespace. Use kube-public to boot from
an
          # administrator-published golden image.
        namespace: demo
      storage:
        resources:
          requests:
            storage: 20Gi
        storageClassName: sc-topolvm
        volumeMode: Filesystem
        accessModes:
          - ReadWriteOnce
```

克隆源命名空间仅限于当前/目标命名空间以及 `kube-public`。从其他非公开命名空间进行克隆，可能需要额外的 CDI 克隆权限。

更新或删除可引导卷

删除可引导卷会移除其 `DataSource` 和受管理的 `DataVolume`。后端 PVC 由 `DataVolume` 拥有（通过 `ownerReferences`），因此在删除 `DataVolume` 时会自动被垃圾回收——在 `DataVolume` 仍然存在时不要删除 PVC，否则 CDI 会重新为其提供。`DataSource`、`DataVolume` 和 PVC 共享相同的名称：

```
kubectl delete datasource ubuntu-2204 -n demo
kubectl delete datavolume ubuntu-2204 -n demo
```

如果 CDI 在成功导入后已经自动回收了 `DataVolume`，那么 `kubectl delete datavolume` 会返回 `NotFound`，而 PVC 会处于无所有者状态——这种情况下请直接删除它：

```
kubectl delete pvc ubuntu-2204 -n demo
```

操作指南

创建虚拟机

[前提条件](#)[注意事项](#)[创建虚拟机](#)[使用 API](#)

虚拟机批量操作

[操作步骤](#)[使用 API](#)

暂停和恢复虚拟机

[注意](#)[暂停虚拟机](#)[恢复虚拟机](#)

克隆虚拟机

使用 VNC 登录虚拟机

[操作步骤](#)[使用 API](#)

虚拟机控制台

[控制台](#)

管理虚拟机

[重置密码](#)[更新密钥](#)[更新规格](#)[热迁移](#)[更新 NAT 网络配置](#)[更新标签和注解](#)[添加服务](#)[重装操作系统](#)[配置 IP](#)[使用 API](#)

监控和告警

[监控](#)[告警](#)[使用 API](#)

虚拟机的快速定位

[前提条件](#)[操作步骤](#)[使用 API](#)

管理虚拟机

使用 InstanceType 规格导入的虚拟机

创建虚拟机

使用镜像创建虚拟机（`VirtualMachine` 资源），并将其调度到已安装 KubeVirt 组件且已启用虚拟化的物理节点上。`VirtualMachine` 是所需状态对象（类似于 Deployment）；当它启动时，KubeVirt 会生成一个正在运行的 `VirtualMachineInstance`（类似于 Pod）。有关本页面中使用的术语，请参阅 [概念与术语表](#)。

您可以通过 [创建虚拟机](#) 使用表单创建虚拟机，也可以切换到 YAML 格式进行操作。

目录

[前提条件](#)

[注意事项](#)

[创建虚拟机](#)

[操作步骤](#)

[相关操作](#)

[使用 API](#)

前提条件

- 在使用镜像创建虚拟机之前，请先与平台管理员确认以下事项：
 - 目标集群是自建集群，且已部署 Kubevirt 组件。
 - 目标节点必须是已启用虚拟化的物理节点。

- 平台中已添加虚拟机镜像。
- 如果您需要使用虚拟机的物理 GPU 直通功能，请联系平台管理员进行以下配置：
 1. 获取 GPU 直通环境准备方案，并准备所需环境。
 2. 准备所需的物理 GPU，并为虚拟机启用相关的物理 GPU 直通功能。

注意事项

使用 Windows 虚拟机时，仅支持通过虚拟机镜像中设置的 用户名/密码 进行登录。请提前联系平台管理员获取相关信息。

创建虚拟机

操作步骤

注意：以下内容以使用表单创建虚拟机为例，您也可以切换到 YAML 格式进行操作。

1. 进入 容器平台。
2. 在左侧导航栏中，点击 虚拟化 > 虚拟机。
3. 点击 创建虚拟机。
4. 在 基本信息 区域中，填写虚拟机名称和显示名称，并设置标签或注解。

参数	说明
标签	用于选择对象并查找满足特定条件的对象集合。必须是键值对，例如： <code>app.kubernetes.io/name: hello-app</code> 。
注解	用于向开发和运维团队提供任意信息。必须是键值对，例如： <code>cpaas.io/maintainer: kim</code> 。

5. 设置机型并选择虚拟机镜像。

参数	说明
规格	您可以根据需要选择推荐的使用场景或自定义资源限制。
物理 GPU (Alpha)	选择物理 GPU 的型号；每台虚拟机仅可分配一块物理 GPU。 注意：虚拟机的物理 GPU 直通是指在虚拟化环境中将实际的图形处理单元 (GPU) 直接分配给虚拟机，使其能够直接访问并使用物理 GPU，从而获得与直接运行在物理机上相当的图形性能，避免虚拟图形适配器带来的性能瓶颈，并提升整体性能。
镜像	选择由平台管理员分配到平台项目中的公共镜像。 注意：仅支持选择与集群架构相同 CPU 架构 的镜像。

6. 在 存储 区域中，请参考以下说明配置相关信息。

参数	说明
磁盘名称	存储磁盘的名称；系统盘名称不可修改。
类型	- 根磁盘：系统自动创建一个 VirtIO 类型的 rootfs 系统盘，用于存储操作系统和数据。 - 数据盘：点击可添加多个数据盘，用于持久化数据存储。默认使用 VirtIO 设备。 注意：数据盘名称不能与现有磁盘名称重复。
卷模式	- 文件系统：将磁盘作为已挂载的文件目录进行挂载。 - 块设备：将磁盘作为块设备进行挂载。
存储类	平台通过创建和管理持久卷声明来维护虚拟机磁盘。您需要指定一个用于动态创建持久卷声明的存储类。 不同存储类支持的卷模式不同；如果所选卷模式没有可用的存储类，请联系管理员添加。

参数	说明
容量	虚拟机存储所需容量；系统盘最小为 20 G。
随 VM 删除	默认启用，且不可修改，表示删除虚拟机时也会删除磁盘数据。

7. 在 网络 区域中，请参考以下说明配置相关信息。

参数	说明
IP 地址	- 默认 动态（DHCP）；虚拟机启动时会动态分配 IP，虚拟机停止时释放该 IP。 - 如果绑定 静态 IP，即使重启后虚拟机也始终使用该 IP 地址。如果当前项目中没有可用 IP，请先适当释放一个 IP。
网络模式	- 桥接：虚拟机与容器组共享同一个 IP 地址，并通过该 IP 地址对外通信。 - NAT：虚拟机会分配一个内部 IP 地址，但对外通信时会转换为容器组 IP 地址。开放端口表示虚拟机的暴露端口，例如 SSH 服务端口 22；不填写开放端口 表示所有端口均开放。
辅助网卡	按需添加辅助网卡。 注意： - 如果需要辅助网卡功能，或当前没有可用的辅助网卡网络类型，请联系平台管理员进行配置。 - SR-IOV 类型仅支持 x86_64 架构上的 Linux 操作系统。 - 默认通过 DHCP 获取 IP 地址。 - 多次重启后，SR-IOV 虚拟机可能会出现两个不同的 VF，但 MAC 地址相同。

8. 在 初始化设置 区域中，请参考以下说明配置相关信息。

参数	说明
密钥	始终使用 SSH 密钥进行远程登录验证。此方式不需要密码校验，建议使用密钥登录虚拟机。 - 您可以使用平台中已有的密钥，也可以立即创建新密钥；所有相关密钥都可在 虚拟化 > 密钥对 页面查看。 - 只有持有私钥的用户才能通过 SSH 访问虚拟机。如果需要多人共同维护虚拟机，可以关联多个密钥，并将私钥分配给不同用户。如果发生密钥泄露，可及时吊销关联密钥以减少损失。 - SSH 密钥的公钥以机密形式存储在平台中；平台不会存储您的私钥，请自行妥善保管。 - root 用户密码请参考相关操作系统文档。
密码	使用操作系统用户和密码进行登录验证，后续仍可更新为密钥方式。 - 该用户仅为初始账户；虚拟机创建成功后，您也可以在虚拟机中创建其他操作系统用户用于登录。 - 平台会对您的 root 用户密码进行加密存储，且您将无法再次看到其明文密码，请自行妥善保管。
立即启动	默认启用。启用此选项后，虚拟机将在创建完成后立即启动；否则仅创建虚拟机。

9. （可选）在 高级配置 区域中，请参考以下说明配置相关信息。

参数	说明
健康检查	- 存活检查：检查虚拟机是否处于健康状态；如果检测结果异常，将根据健康检查配置判断是否重启实例。 - 可用性检查：检查虚拟机是否已完成启动并处于正常服务状态；如果检测到虚拟机实例的健康状态异常，将更新虚拟机的状态。 相关参数说明。
节点亲和性	- 优选：尽可能将虚拟机调度到满足亲和性要求的节点。系统会结合亲和性权重和其他调度要求（例如计算资源需求）来确定可运行该虚拟机的节点。 - 必需：虚拟机仅会调度到完全满足亲和性要求的节点。

10. 确认信息无误后，点击 **创建**。

等待虚拟机状态从 **Creating** 变为 **Running**。

相关操作

您可以在列表页右侧点击  图标，或在详情页右上角点击 **Actions**，按需更新或删除虚拟机。有关重置密码或更新密钥等其他相关操作，请参阅 [管理虚拟机](#)。

注意：

- 仅当虚拟机处于 **Abnormal**、**Unknown** 或 **Stopped** 状态时，才能执行更新操作。
- 更新时不支持显示在虚拟机创建后单独挂载或创建的磁盘。
- 默认情况下，更新时 **立即启动** 处于禁用状态；您可以按需启用。

使用 API

虚拟机是 `VirtualMachine`（`kubevirt.io/v1`）。下面的示例从可启动卷（`DataSource`）启动，挂载 Pod 网络，并通过 cloud-init 注入 SSH 密钥：


```

apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: web-01
  namespace: demo
spec:
  running: false          # set true to start on creation
  dataVolumeTemplates:
    - metadata:
        name: web-01-rootfs
      spec:
        sourceRef:          # clone an existing bootable volume (DataSource)
          kind: DataSource
          name: centos-stream9
          namespace: kube-public
        storage:
          resources:
            requests:
              storage: 20Gi
          storageClassName: rbd-1
          volumeMode: Block
          accessModes:
            - ReadWriteMany
  template:
    metadata:
      labels:
        kubevirt.io/vm: web-01
    spec:
      domain:
        cpu:
          cores: 1
        resources:          # ACP resource-limit model: requests == limits
          requests:
            memory: 2Gi
          limits:
            memory: 2Gi
      devices:
        disks:
          - name: rootfs
            disk: { bus: virtio }
          - name: cloudinit
            disk: { bus: virtio }

```

```

    disk: { bus: virtio }
  interfaces:
    - name: default
      masquerade: {}
  networks:
    - name: default
      pod: {}
  volumes:
    - name: rootfs
      dataVolume:
        name: web-01-rootfs
    - name: cloudinit
      cloudInitNoCloud:
        userData: |
          #cloud-config
          ssh_authorized_keys:
            - ssh-ed25519 AAAAC3...example user@host

```

```
kubectl apply -f vm.yaml
```

```
# Start it (ACP virtual machines use the spec.running field):
```

```
kubectl patch vm web-01 -n demo --type merge -p '{"spec":{"running":true}}'
```

引导磁盘由 `dataVolumeTemplates` 条目提供，该条目会克隆所引用的可启动卷。也支持导入带有 `spec.instanceType` 引用（而不是内联资源）的虚拟机，并且在您编辑它时会转换为内联 spec——请参阅 [管理虚拟机](#)。

虚拟机批量操作

执行启动、停止、重启和删除虚拟机等批量操作。

目录

操作步骤

使用 API

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中单击 **Virtualization > Virtual Machines**。
3. 定位目标虚拟机，单击：可对单个虚拟机执行操作，或参考下图对虚拟机执行批量操作。

注意：

- 当虚拟机处于已暂停或已停止状态时，可以执行 **Start/Batch Start** 操作；当虚拟机处于 Preparing、Starting、Running、Paused、Unknown 或 Exception 状态时，可以执行 **Stop/Batch Stop** 操作；当虚拟机处于 Running 状态时，可以执行 **Restart/Batch Restart** 操作。
- 对虚拟机执行强制 **Restart/Stop** 操作等同于切断虚拟机电源，可能导致尚未写入磁盘的数据丢失。

4. 按照界面提示完成操作。当虚拟机变为以下状态时，表示操作成功。

操作	状态
Start Virtual Machine	运行中
Stop Virtual Machine	已停止
Restart Virtual Machine	运行中

使用 API

ACP 虚拟机将其电源状态保存在 `spec.running` 字段中。通过 patch 该字段来启动或停止；重启请使用 `virtctl`：

```
# Start
kubectl patch vm web-01 -n demo --type merge -p '{"spec":{"running":true}}' # or: virtctl start web-01 -n demo
# Stop
kubectl patch vm web-01 -n demo --type merge -p '{"spec":{"running":false}}' # or: virtctl stop web-01 -n demo
# Restart (a running VM)
virtctl restart web-01 -n demo
# Delete
kubectl delete vm web-01 -n demo
```

这里没有批量对象——在循环中对每个虚拟机应用相同的命令即可。强制停止或重启（`virtctl stop --force`）等同于切断电源，可能会丢失尚未写入磁盘的数据。

暂停和恢复虚拟机

暂停虚拟机会冻结其 CPU 和内存状态，但不会释放资源或丢失数据——工作负载会一直处于暂停状态，直到你将其恢复。可将其用于临时让出主机调度，或者冻结虚拟机以便进行检查和取证。

暂停和恢复通过 KubeVirt 的 `pause` / `unpause` 子资源（`subresources.kubevirt.io/v1`）作用于虚拟机实例；它们不会添加任何 CRD。

目录

注意

暂停虚拟机

恢复虚拟机

注意

- **Pause** 适用于 **Running** 的虚拟机；**Resume** 适用于 **Paused** 的虚拟机。
- 暂停不会释放 CPU/内存，也不会丢失内存中的状态——来宾工作负载只是被暂停。
- 暂停是一项会中断工作的操作（工作负载会一直中断，直到你恢复它），因此控制台会要求确认。恢复是一项安全操作，会立即生效，无需确认。
- 权限：需要在 `virtualmachineinstances` 上具有 `update` 动词权限。

暂停虚拟机

使用 `virtctl`，它会向虚拟机实例的 `pause` 子资源发出一个空正文的 `PUT` 请求：

```
virtctl pause vm web-01 -n demo
```

底层请求如下：

```
PUT /apis/subresources.kubevirt.io/v1/namespaces/demo/virtualmachineinstances/web-01/pause
```

成功后，虚拟机实例会带有 `Paused=True` 条件（原因 `PausedByUser`），并且 `VirtualMachine.status.printableStatus` 会变为 `Paused`。对已暂停的虚拟机执行暂停操作会返回 `409 Conflict`。

恢复虚拟机

```
virtctl unpause vm web-01 -n demo
```

底层请求如下：

```
PUT /apis/subresources.kubevirt.io/v1/namespaces/demo/virtualmachineinstances/web-01/unpause
```

`VirtualMachine.status.printableStatus` 会恢复为 `Running`。对未暂停的虚拟机执行恢复操作会返回 `409 Conflict`。

克隆虚拟机

克隆会创建现有虚拟机的完整副本——包括其规格、磁盘和网络配置——并将其作为同一命名空间中的新虚拟机。ACP 通过 KubeVirt `VirtualMachineClone` (`clone.kubevirt.io/v1beta1`) 资源实现此功能，该资源通过内部 `snapshot → restore` 异步执行复制。

这与克隆可启动卷（PVC 或 snapshot 数据源）不同：VM 克隆会复制整台机器，而可启动卷克隆只会复制磁盘镜像。

目录

[前提条件](#)[注意事项](#)[克隆虚拟机](#)[相关操作](#)

前提条件

- 支持 **snapshot** 的存储。由于克隆是通过内部 `snapshot → restore` 实现的，因此源虚拟机的磁盘必须位于支持 volume snapshot 的 StorageClass 上（该存储后端存在匹配的 `VolumeSnapshotClass` ）。如果存储不支持 snapshot，克隆将失败——请检查 `VirtualMachineClone` 的 `status.phase` 和 `status.conditions` 以查看错误。完整的前提条件列表请参见 [在 KubeVirt 上克隆虚拟机](#)。

注意事项

- 仅限同一命名空间。`VirtualMachineClone.spec.target` 没有 `namespace` 字段，因此新虚拟机始终会在源对象所在的命名空间中创建。若要将虚拟机复制到其他命名空间，请改为导出后重新导入。
- 新虚拟机是异步出现的。提交克隆请求后会立即返回；目标对象会在 `snapshot → restore` 完成时创建（`status.phase` 会推进到 `Succeeded`）。
- 克隆已停止的虚拟机最可靠，因为此时磁盘状态是静止的。
- **MAC** 地址默认会重置，因此克隆后的虚拟机不会在网络上与源虚拟机冲突。
- **Kube-OVN** 网络：克隆会始终过滤掉源对象的 `ovn.kubernetes.io/*` pod 注解，因此新虚拟机会请求一个新的 IP，而不会继承（并与之冲突）源 IP。
- 权限：克隆操作受 `clone.kubevirt.io` API group 中 `virtualmachineclones` 的 `create` verb 约束。该 group 不属于标准的 **Virtual Machines** 功能权限，因此仅持有 Virtual Machines 角色的用户会看到 **Clone** 操作被禁用。平台管理员或命名空间管理员必须先授予一个额外的 role/binding，允许对 `virtualmachineclones.clone.kubevirt.io` 执行 `create`，该用户才能进行克隆。

克隆虚拟机

在源命名空间中创建一个 `VirtualMachineClone` 对象。下面的示例将 `web-01` 克隆为 `web-01-clone`，重置 MAC 地址（`newMacAddresses: {}`），并从 pod 模板中移除 Kube-OVN 注解，使克隆后的虚拟机获得新的 IP：

```

apiVersion: clone.kubevirt.io/v1beta1
kind: VirtualMachineClone
metadata:
  name: web-01-clone
  namespace: demo
spec:
  source:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: web-01
  target:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: web-01-clone
  # Keep all VM-level labels and annotations. Interface MAC addresses are
  # regenerated by default; set newMacAddresses to pin specific MACs instead.
  labelFilters: ["*"]
  annotationFilters: ["*"]
  # Kube-OVN: keep every pod-template annotation EXCEPT the source's
  # ovn.kubernetes.io/* address bindings, so the clone gets a fresh IP instead of
  # inheriting the source's. Clone filters are whitelist-based: a bare pattern
  # keeps the matching keys (everything else is dropped) and a "!" prefix
  # excludes – so the "*" entry is required (a lone "!ovn.kubernetes.io/*"
  # would
  # whitelist nothing and strip ALL pod-template annotations).
  template:
    annotationFilters:
      - "*"
      - "!ovn.kubernetes.io/*"

```

应用后，观察克隆进度直到 **Succeeded**：

```

kubectl apply -f vmclone.yaml
kubectl get virtualmachineclone web-01-clone -n demo \
  -o jsonpath='{.status.phase}'

```

当 `status.phase` 为 `Succeeded` 时，新的虚拟机 `web-01-clone` 就会存在于 `demo` 命名空间中。`status.restoreName` 和 `status.snapshotName` 分别引用了操作过程中创建的中间 restore 和 snapshot 对象。

如果只想复制部分元数据，请设置顶层的 `labelFilters` / `annotationFilters`（它们过滤的是虚拟机对象自身的 labels/annotations；`template.*` 过滤的是 **pod** 模板元数据）：

```
spec:
  labelFilters:
    - "*"
    - "!some/transient-label"
  annotationFilters:
    - "*"

```

相关操作

当克隆达到 **Running** 状态后，你可以像管理其他虚拟机一样管理它——请参见 [管理虚拟机](#)。由于 MAC 地址（以及在 Kube-OVN 中的 IP）会被重新生成，如果来宾系统内部固定了地址，请确认新虚拟机的网络配置。

串口控制台

串口控制台会将文本终端（TTY）附加到虚拟机的串口。它非常适合无头 Linux 来宾、查看内核启动消息以及捕获 kernel panic 输出——这些场景下图形化的 [VNC 控制台](#) 并不适用。

串口控制台通过 WebSocket 连接到 KubeVirt 虚拟机实例 `console` 子资源，并传输原始 TTY 字节流（子协议 `plain.kubevirt.io`）。

目录

注意事项

打开串口控制台

注意事项

- 只要虚拟机具有运行中的实例（Running、Paused、Starting 或 Stopping），串口控制台就可用。对于已停止的虚拟机，由于不存在实例，因此不可用——该端点会返回 `404`。
- 来宾必须在其控制台上暴露串口 `getty`——Linux cloud image 通常会在内核命令行中设置 `console=ttyS0`。Windows 来宾没有串口控制台；请改用 VNC。
- 权限：需要对 `virtualmachineinstances/console` 授予 `get` 动词权限。

打开串口控制台

```
virtctl console web-01 -n demo
```

这会升级为指向虚拟机实例 `console` 子资源的 WebSocket :

```
GET /apis/subresources.kubevirt.io/v1/namespaces/demo/virtualmachineinstances/web-01/console
Upgrade: websocket
Sec-WebSocket-Protocol: plain.kubevirt.io
```

运行中的实例会返回 `101 Switching Protocols` , 然后持续传输原始 TTY 字节 ; 已停止的虚拟机会返回 `404` (不存在实例) 。

使用 VNC 登录虚拟机

作为应急操作方法，使用 Web Console（VNC）登录虚拟机。

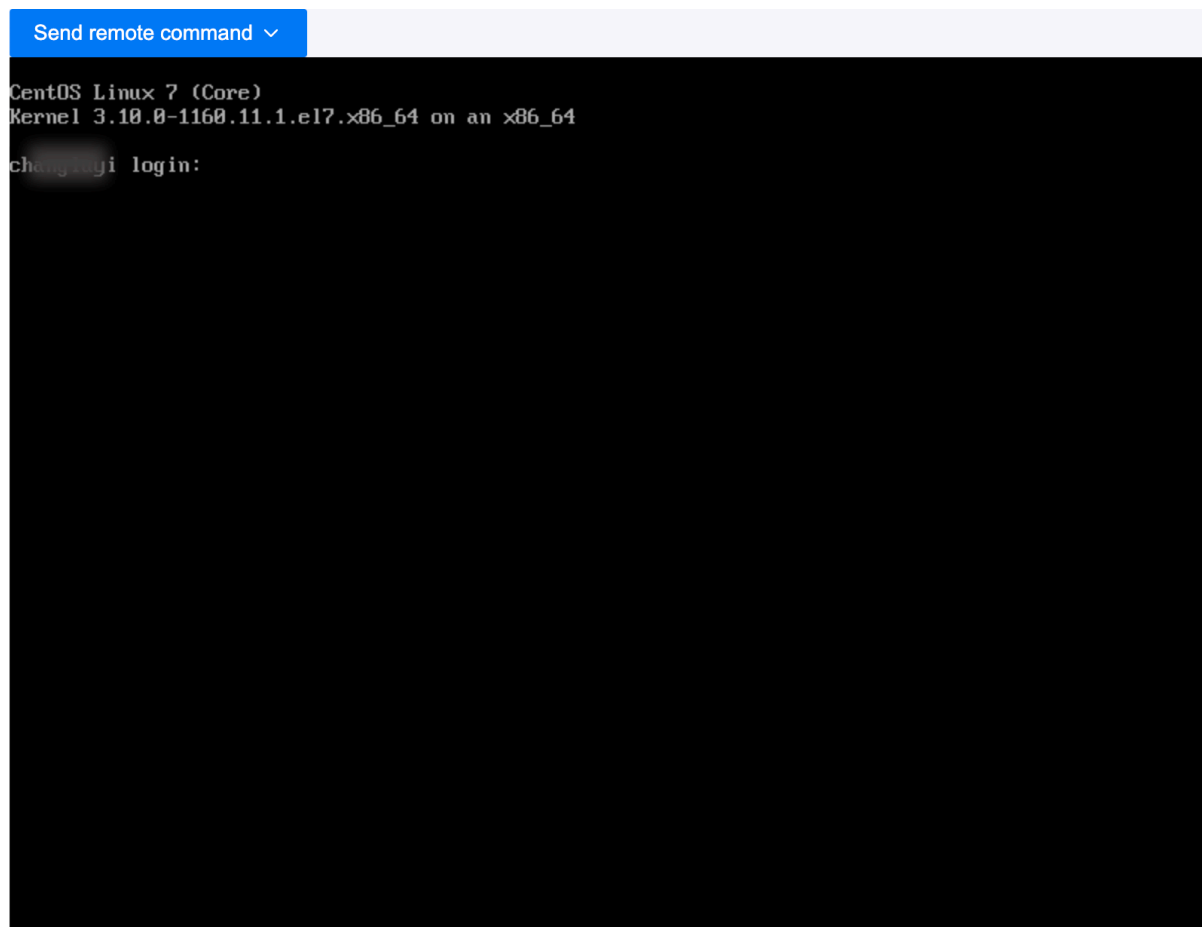
目录

[操作步骤](#)

[使用 API](#)

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 单击 **⋮ > VNC Login**。
4. 控制台窗口将自动打开；您需要输入用户名和密码才能登录。



注意：

- 支持发送常用键盘命令。
- 支持复制和粘贴命令及参数。

使用 API

```
virtctl vnc web-01 -n demo
```

`virtctl vnc` 会启动一个本地 VNC 查看器，通过 WebSocket 连接到虚拟机实例 `vnc` 子资源：

```
GET /apis/subresources.kubevirt.io/v1/namespaces/demo/virtualmachineinstances/web-01/vnc
Upgrade: websocket
```

运行中的实例会返回 `101 Switching Protocols`；该连接需要对 `virtualmachineinstances/vnc` 具有 `get` 动词权限。

管理密钥对

创建、更新或删除密钥对。

密钥对存储为类型为 `kubernetes.io/ssh-public-key` 的 Kubernetes `Secret`；只保留公钥，平台绝不会存储您的私钥。通过 关联 **SSH** 密钥 创建虚拟机时会使用密钥对（通过 cloud-init 注入）。

目录

[创建密钥对](#)

[更新密钥对](#)

[删除密钥对](#)

[使用 API](#)

创建密钥对

1. 导航到 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization > Key Pairs**。
3. 点击 **Create Key Pair**。

当前仅支持 SSH 类型的密钥对。您可以手动导入密钥，也可以让系统自动生成密钥对。使用系统生成的密钥对时，平台支持将私钥自动下载到本地计算机。平台不会保存私钥。

4. 点击 **Create**。

更新密钥对

1. 导航到 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization > Key Pairs**。
3. 找到 密钥对名称，点击 **Update**。
4. 重新导入或让系统生成新的密钥对后，点击 **Update**。

删除密钥对

1. 导航到 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization > Key Pairs**。
3. 找到 密钥对名称，点击 **Delete**，然后确认。

使用 API

密钥对是类型为 `kubernetes.io/ssh-public-key` 的 `Secret`，其 `ssh-publickey` 值为公钥：

```
apiVersion: v1
kind: Secret
metadata:
  name: my-key
  namespace: demo
type: kubernetes.io/ssh-public-key
stringData:
  ssh-publickey: "ssh-ed25519 AAAA...example user@host"
```

```
kubectl apply -f keypair.yaml
kubectl get secret my-key -n demo -o jsonpath='{.type}'{"\n"}' # kubernetes.io/ssh-public-key
kubectl delete secret my-key -n demo
```

在通过 关联 **SSH** 密钥 创建虚拟机时引用该密钥；平台会通过 cloud-init `ssh_authorized_keys` 将公钥注入到 guest 中。

管理虚拟机

以下操作可在虚拟机列表（行内：菜单）和详情页面中执行。以下操作已作为 API/CLI 参考文档记录：[克隆](#)、[暂停和恢复](#)以及[串口控制台](#)。每个操作也都有对应的 API 调用——请参见下方的使用 **API**。

目录

重置密码

操作步骤

更新密钥

操作步骤

更新规格

热迁移

更新 NAT 网络配置

操作步骤

更新标签和注解

添加服务

重装操作系统

操作步骤

配置 IP

操作步骤

使用 API

使用 InstanceType 规格导入的虚拟机

重置密码

重置 **root** 用户密码。使用密码登录虚拟机时，此密码也将作为登录密码。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到该虚拟机，并选择 ⋮ > **Reset Password**。
4. 设置密码。
5. 单击 **Reset**。

注意：请妥善保管密码。为确保环境安全，平台会对您的密码进行加密存储，且您将无法再次查看明文密码。

更新密钥

更新 SSH 密钥。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到该虚拟机，并选择 ⋮ > **Update Key**。
4. 选择一个或多个已关联的密钥，或选择 **Create Key**。
5. 选择是否立即重启；更新密钥需要重启虚拟机后才会生效。
6. 单击 **Update**。

更新规格

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到目标虚拟机，并单击：**> Update Specifications**。
4. 根据平台推荐场景或自定义需求修改相关资源。
5. 选择是否 **Restart Immediately**；配置将在重启后生效。
6. 单击 **Update**。

热迁移

注意：有关前置条件、集群级迁移限制和命名空间级策略，以及 API，请参见[虚拟机热迁移](#)。

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到目标虚拟机，并单击：**> Live Migration**。
4. 单击 **Confirm**。

更新 NAT 网络配置

在使用 NAT 网络模式时，平台默认会为 SSH 服务开放 22 端口，您也可以根据需要开放其他端口。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 单击 **Virtual Machine Name**。
4. 在 **Basic Information** 区域中，单击 **Open Port** 右侧的图标。
5. 输入端口号，并按 Enter 键确认。
6. 选择是否 **Restart Immediately**；配置将在重启后生效。
7. 单击 **Update**。

更新标签和注解

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 单击 **Virtual Machine Name**。
4. 在 **Basic Information** 区域中，单击 **Tags** 或 **Annotations** 右侧的图标。
5. 按需进行配置，然后单击 **Update**。

添加服务

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 单击 **Virtual Machine Name**。
4. 在 **Login Information** 区域中，单击 **Internal Route** 右侧的图标。
5. 请参见 [Create Service](#) 页面，快速为虚拟机添加内部路由。
6. 单击 **Confirm**。

重装操作系统

强烈建议在重装操作系统前先备份数据，以防止数据丢失。

注意：此操作会清除虚拟机 系统盘 中的所有数据以及所有 快照，且不可恢复。请谨慎操作！

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到该虚拟机，并选择：**> Reinstall Operating System**。
4. 在 **Reinstall Operating System** 窗口中，配置以下参数。
 - **Provisioning Method**：当前仅支持公有镜像。

- **Select Image**：默认使用当前操作系统镜像进行重装。如果您希望重装新的操作系统，请先选择虚拟机镜像所属的操作系统，再选择属于该操作系统的虚拟机镜像。

5. 单击 **Reinstall**。

配置 IP

通过动态分配（DHCP）为虚拟机分配 IP，或为虚拟机绑定固定 IP；新的 IP 会在虚拟机重启后生效。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，单击 **Virtualization > Virtual Machines**。
3. 找到目标虚拟机，并单击：**> Configure IP**。
4. 配置 **IP Address**。
 - 填写一个可用 IP：绑定固定 IP 后，即使重启，虚拟机也会始终使用该 IP 地址。
 - 保持该项为空：将使用动态分配（DHCP）获取 IP，在虚拟机启动时分配 IP，停止时释放 IP。
5. 选择是否 **Restart Immediately**；配置将在重启后生效。
6. 单击 **Configure**。

使用 API

控制台操作映射到 `VirtualMachine` 资源上的标准调用：

- 更新规格 —— patch 内联资源（ACP 使用资源限制模型，因此请将 `requests` 和 `limits` 设置为相同的值）；重启虚拟机后更改才会生效：

```
kubectl patch vm web-01 -n demo --type merge -p \
  '{"spec":{"template":{"spec":{"domain":{"resources":{"requests":{"cpu":"2","memory":"4Gi"},"limits":{"cpu":"2","memory":"4Gi"}}}}}}}'
```

- 更新标签和注解：

```
kubectl label vm web-01 -n demo app=billing
kubectl annotate vm web-01 -n demo cpaas.io/maintainer=kim
```

- 热迁移 —— 创建一个 `VirtualMachineInstanceMigration` ；请参见[虚拟机热迁移](#)。

使用 `InstanceType` 规格导入的虚拟机

从其他平台导入的虚拟机（例如通过 Forklift）可能会通过引用的

`VirtualMachineClusterInstancetype`（`specinstancetype`）来定义其 CPU 和内存，而不是在 `domain.resources` 中内联定义。系统会解析所引用的实例类型，以便显示其实际生效的 CPU 和内存。

当您编辑这类虚拟机的规格时，系统会一次性将其转换为自定义的内联规格——写入 `domain.resources`，并移除 `specinstancetype`——且更改会在重启后生效。未带 ACP 标签的原始导入虚拟机只能通过 YAML 进行编辑。

```
# List the cluster instance types a VM can reference:
kubectl get virtualmachineclusterinstancetype
```



[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [虚拟机](#) > [操作指南](#) > [监控和告警](#)

监控和告警

从 CPU、内存、存储和网络等方面对虚拟机进行监控和告警。为便于及时告警，还可以配置通知策略。

直观呈现的监控数据可为运维巡检或性能调优提供决策支持，而完善的告警和通知机制则有助于保障虚拟机的稳定运行。

目录

[监控](#)

[告警](#)

[配置告警策略](#)

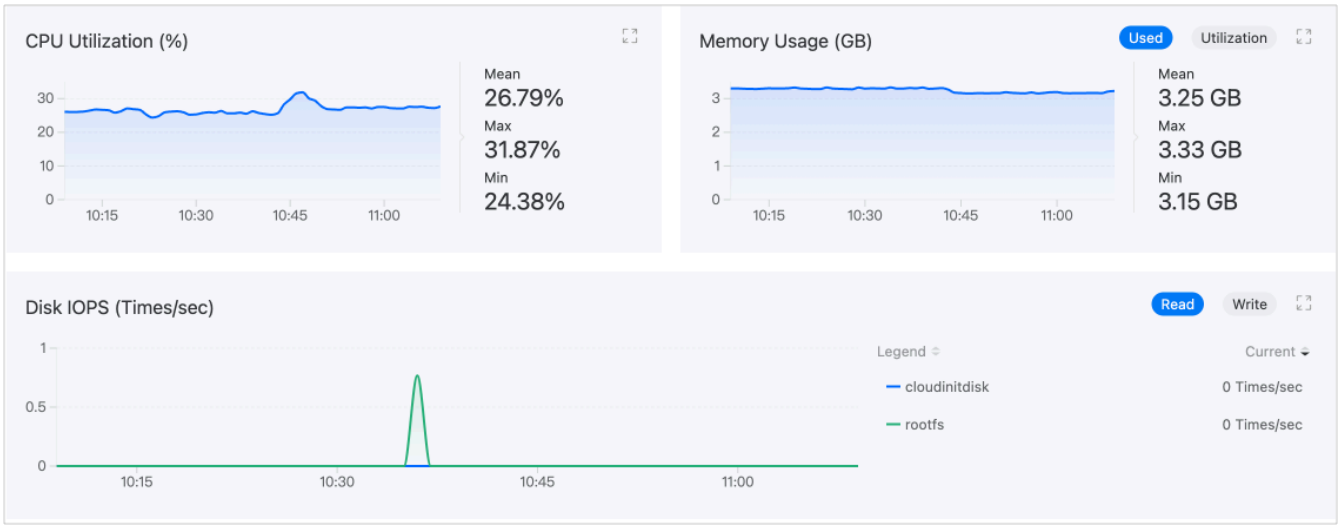
[处理告警](#)

[绑定通知策略](#)

[使用 API](#)

监控

默认情况下，平台会收集虚拟机常用的性能监控指标，包括 CPU、内存、存储和网络。进入 [虚拟化 > 虚拟机](#)，在虚拟机详情的 [监控](#) 选项卡中，可以查看这些指标的实时监控数据。



告警

配置告警策略

要启用告警，必须先创建告警策略。告警策略用于描述你希望监控的对象、希望在什么条件下接收告警，以及如何接收相关告警通知。进入 容器平台 > 虚拟化 > 虚拟机，在虚拟机详情的告警 选项卡中，点击 创建告警策略 完成配置。

参数	描述
告警类型	- 指标告警：被监控对象是平台预定义的指标，例如 <i>内存使用率</i>。 - 事件告警：被监控对象是某个事件的原因，即虚拟机切换到当前状态的原因，例如 BackOff、Pulling、Failed。
触发条件	由比较运算符、告警阈值和持续时间组成。通过将实时监控结果与设置的阈值进行比较，判断是否触发告警。 如果设置了持续时间，平台还会比较被监控对象处于告警状态的持续时长。
告警级别	- 提示：被监控对象存在预期内的问题，暂时不会直接影响业务运行，但存在潜在风险。例如，CPU 使用率连续 3 分钟超过 70%。 - 警告：被监控对象存在运行风险，如不及时处理，可能会影响正常业务运行。例如，CPU 使用率连续 3 分钟超过 80%。 - 严重：被监控对象存在已知问题，可能导致平台功能异常，从而影响正常业务运行。 - 灾难：被监控对象已发生故障，导致平台服务中断、数据丢失，影响重大。

提示：虚拟机告警功能与平台的通用告警功能类似。有关更详细的配置指导，请参阅平台的通用告警文档。

处理告警

进入 告警 选项卡，如果存在告警状态策略提示，请及时处理。

绑定通知策略

除了在 告警 选项卡中进行实时告警外，平台还支持通过 email、SMS 等方式将告警信息发送给相关人员，通知他们采取必要措施解决问题或防止故障。通知策略需要联系管理员进行设置。

使用 API

告警策略基于 Prometheus 规则。你也可以直接将告警定义为 `PrometheusRule` (`monitoring.coreos.com/v1`)：

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: vm-high-memory
  namespace: demo
spec:
  groups:
    - name: vm.rules
      rules:
        - alert: VMHighMemoryUsage
          expr: kubevirt_vmi_memory_used_bytes / kubevirt_vmi_memory_avai
lable_bytes > 0.8
          for: 3m
          labels:
            severity: warning
          annotations:
            summary: "VMI {{ $labels.name }} memory usage above 80%"
```

KubeVirt 会输出按每个 virtual machine instance 划分的指标（前缀为 `kubevirt_vmi_*`，并带有 `namespace` 和 `name` 标签），你可以在 Prometheus 中查询，或用于你自己的

dashboard——例如 `kubevirt_vmi_memory_used_bytes` 、
`kubevirt_vmi_network_receive_bytes_total` 、
`kubevirt_vmi_storage_read_traffic_bytes_total` 和
`kubevirt_vmi_migration_data_processed_bytes` 。准确的指标名称会因 KubeVirt 版本而
异；可通过查询 `{__name__=~"kubevirt_vmi_.+"}` 列出完整集合。

虚拟机的快速定位

平台支持按集群显示虚拟机列表，便于平台管理员快速定位虚拟机的命名空间并执行扩容或故障排查等操作，从而提高运维效率。

目录

[前提条件](#)[操作步骤](#)[使用 API](#)

前提条件

在使用前，请确保当前集群已启用虚拟化功能。请参阅 [安装](#)。

操作步骤

1. 进入 管理员。
2. 在左侧导航栏中，点击 虚拟化管理 > 虚拟机。
3. 选择 **Cluster** 以查看该集群中的虚拟机列表。
4. 您可以通过名称、IP 地址或创建者快速定位虚拟机。
5. 单击虚拟机的 **Name** 链接，进入该虚拟机的详情页，在此可以执行扩容或故障排查等操作。

使用 API

在命令行中列出并定位所有命名空间中的虚拟机：

```
# All virtual machines in the cluster, with status:
```

```
kubectl get vm -A -o wide
```

```
# A specific virtual machine by name across namespaces:
```

```
kubectl get vm -A --field-selector metadata.name=web-01
```

```
# Virtual machine instances show the assigned IP address and node:
```

```
kubectl get vmi -A -o wide
```

实用指南

使用 **virtctl** 命令行工具

控制台和访问

上传和导出镜像

热插拔磁盘

电源和迁移

使用 libguestfs 编辑和封存镜像

配置 **USB** 主机透传

功能概述

使用场景

前提条件

步骤

操作结果

了解更多

安装来宾工具

QEMU 来宾智能

VirtIO 驱动程序

虚拟 TPM (vTPM)

CPU 和内存热插拔

工作原理

前提条件

第 1 步：创建具有热插拔余量的 VM

第 2 步：热插拔 CPU

第 3 步：热插拔内存

向下缩容：哪些可以在线完成，哪些需要重启

已验证行为汇总

故障排除

虚拟机恢复

操作步骤

在 **KubeVirt** 上克隆虚拟机

确保先决条件

快速开始

理解 VirtualMachineClone 对象

在 **StorageClass**

前提条件

准备迁移

为虚拟机配置高可用性

从现有虚拟机创建 **VM** 模板

前提条件

迁移

操作步骤

1. 在 Alauda 控制台上创建一个新的 Kubernetes 集群。

操作步骤

使用 virtctl 命令行工具

`virtctl` 是 KubeVirt 的标准命令行客户端。它通过补充 `kubectrl` 无法直接处理的一些命令来扩展功能，这些命令需要流式连接或子资源调用——例如打开控制台、上传和导出镜像、热插拔磁盘，以及触发迁移。它随 KubeVirt 一起发布；请从 KubeVirt releases 中下载与集群 KubeVirt 版本匹配的二进制文件，并将其放置到你的 `PATH` 中。

```
# Example: download virtctl for the cluster's KubeVirt version
VERSION=$(kubectrl get kubevirt kubevirt -n kubevirt -o jsonpath='{.status.observedKubeVirtVersion}')
curl -L -o virtctl "https://github.com/kubevirt/kubevirt/releases/download/${VERSION}/virtctl-${VERSION}-linux-amd64"
chmod +x virtctl && sudo mv virtctl /usr/local/bin/
```

目录

控制台和访问

上传和导出镜像

热插拔磁盘

电源和迁移

使用 libguestfs 编辑和封存镜像

控制台和访问

```
virtctl console web-01 -n demo      # serial (text) console – see Serial Console
virtctl vnc web-01 -n demo          # graphical VNC console – see VNC
virtctl ssh user@web-01 -n demo     # SSH into the VM (uses the VM's pod network)
virtctl scp ./file user@web-01:/tmp -n demo
```

参见 [Serial Console](#) 和 [Logging into the Virtual Machine using VNC](#)。

上传和导出镜像

```
# Upload a local image into a DataVolume (CDI cdi-uploadproxy must be reachable):
virtctl image-upload dv my-image --size=20Gi --storage-class=rbd-1 \
  --image-path=./my-image.qcow2 -n demo

# Export a VM disk or snapshot for download:
virtctl vmexport download my-export --vm=web-01 --output=disk.img.gz -n demo
```

有关完整的本地镜像上传和虚拟机创建工作流，请参见 [Uploading a Local Image and Creating a Virtual Machine](#)，以及 [Exporting Virtual Machine Images](#)。

热插拔磁盘

```
virtctl addvolume web-01 --volume-name=datadisk-1 --persist -n demo
virtctl removevolume web-01 --volume-name=datadisk-1 -n demo
```

参见 [Managing Virtual Disks](#)。

电源和迁移

```
virtctl start web-01 -n demo
virtctl stop web-01 -n demo
virtctl restart web-01 -n demo
virtctl pause vm web-01 -n demo          # freeze CPU/memory
virtctl unpause vm web-01 -n demo
virtctl migrate web-01 -n demo           # start a live migration
virtctl migrate-cancel web-01 -n demo    # cancel an in-progress live migration
```

参见 [Virtual Machine Live Migration](#) 和 [Pausing and Resuming a Virtual Machine](#)。

使用 libguestfs 编辑和封存镜像

`virtctl guestfs` 会启动一个交互式 Pod，并将 libguestfs 工具附加到一个 PVC 上，因此你可以离线检查或修改磁盘镜像——例如，在将其作为可启动卷发布之前重置 machine ID 或对模板进行泛化：

```
virtctl guestfs my-pvc -n demo
# inside the pod:
virt-customize -a /dev/vda --root-password password:'...'
virt-sysprep -a /dev/vda          # generalize (remove host-specific data)
before templating
```

配置 USB 主机透传

目录

功能概述

使用场景

前提条件

步骤

暴露 USB 设备

将 USB 设备分配给 Virtual Machine

操作结果

了解更多

暴露多个 USB 设备

将 USB 设备分配给 Virtual Machine

功能概述

USB（Universal Serial Bus）透传功能可让您从虚拟机访问和管理 USB 设备。

使用场景

在虚拟机（VM）中运行的某些应用程序具有加密要求，需要与专用 USB 设备交互。在这种情况下，需要将 USB 设备从宿主机透传到 VM。

前提条件

- 平台版本必须至少为 v3.18。

步骤

1 暴露 USB 设备

要将 USB 设备分配给 VM，必须通过 **ResourceName** 暴露该 USB 设备。您可以通过编辑 kubevirt 命名空间下 **HyperConverged CR** 中的

`spec.permittedHostDevices.usbHostDevices` 部分进行配置。

以下是一个 **ResourceName** 为 `kubevirt.io/storage` 的 USB 设备示例配置，其中 vendor 为 `0bda`，product 为 `8812`：

```
spec:
  permittedHostDevices:
    usbHostDevices:
      - resourceName: kubevirt.io/storage
    selectors:
      - vendor: '0bda'
        product: '8812'
```

提示

可以使用 `lsusb` 命令获取 USB 设备的 vendor 和 product 标识符。例如：

```
lsusb
Bus 001 Device 007: ID 0bda:8812 Realtek Semiconductor Corp. RT
L8812AU 802.11a/b/g/n/ac 2T2R DB WLAN Adapter
```

该命令会列出所有已连接的 USB 设备，其中 ID 显示 vendor:product 对。

2 将 USB 设备分配给 Virtual Machine

现在，在 VM 配置中，您可以添加

`spec.template.spec.domain.devices.hostDevices.deviceName`，以引用上一步提供的 `ResourceName` 并为其分配本地名称。例如：

```
spec:
  template:
    spec:
      domain:
        devices:
          hostDevices:
            - deviceName: kubevirt.io/storage
              name: usb-storage
```

提示

请确保在编辑配置之前已停止 VM。

操作结果

完成配置后，在虚拟机内执行 `lsusb` 命令。如果输出中列出了宿主节点的 USB 设备，则表示透传成功。例如：

```
lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 002: ID 0bda:8812 Realtek Semiconductor Corp. RTL8812AU 80
2.11a/b/g/n/ac 2T2R DB WLAN Adapter
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

了解更多

您可能希望将多个 USB 设备透传到虚拟机，例如键盘、鼠标或智能卡设备。我们支持在同一个 resourceName 下分配多个 USB 设备。配置方法如下：

1 暴露多个 USB 设备

```
spec:
  permittedHostDevices:
    usbHostDevices:
      - resourceName: kubevirt.io/peripherals
        selectors:
          - vendor: '0bda'
            product: '8812'
          - vendor: '062a'
            product: '4102'
          - vendor: '072f'
            product: 'b100'
```

提示

注意：所有 USB 设备都必须已物理连接并在宿主机上被检测到，以确保能够成功分配给虚拟机。

2 将 USB 设备分配给 Virtual Machine

```
spec:
  template:
    spec:
      domain:
        devices:
          hostDevices:
            - deviceName: kubevirt.io/peripherals
              name: local-peripherals
```

安装来宾工具 (Guest Agent、VirtIO 驱动程序、vTPM)

在来宾中安装正确的工具，可以解锁平台所依赖的功能——报告 IP 地址、优雅关机、应用一致性快照，以及（对于 Windows）可用的磁盘和网络驱动程序。

目录

[QEMU 来宾智能体](#)

[VirtIO 驱动程序 \(Windows\)](#)

[虚拟 TPM \(vTPM\)](#)

QEMU 来宾智能体

QEMU 来宾智能体允许平台读取来宾 IP、优雅地关闭来宾，并冻结文件系统以生成应用一致性快照。没有它，在线快照只能是崩溃一致性。

- **Linux** — 从发行版软件包中安装并启用该智能体：

```
# RHEL / CentOS / Fedora
sudo dnf install -y qemu-guest-agent && sudo systemctl enable --now qemu-guest-agent

# Debian / Ubuntu
sudo apt-get install -y qemu-guest-agent && sudo systemctl enable --now qemu-guest-agent
```

- **Windows** — 从 virtio-win 介质 (`guest-agent\qemu-ga-x86_64.msi`) 中安装来宾智能体。

当该智能体运行时，`VirtualMachineInstance.status.conditions` 会包含 `AgentConnected=True`，并且 IP 地址会显示在控制台和 `status.interfaces` 中。

VirtIO 驱动程序 (Windows)

Windows 没有内置的 VirtIO 驱动程序，因此新安装的 Windows 来宾在安装驱动程序之前无法看到其 VirtIO 磁盘或 NIC。将 **virtio-win** 容器磁盘作为 CD-ROM 挂载，并安装驱动程序：

- 在 Windows 安装过程中，从 virtio-win CD-ROM 中选择 加载驱动程序（启动磁盘使用 `viostor`）。
- 安装完成后，从同一介质运行 `virtio-win-guest-tools.exe`，以安装所有驱动程序以及来宾智能体。

虚拟 TPM (vTPM)

Windows 11 和 BitLocker 需要 TPM 2.0 设备。通过

`spec.template.spec.domain.devices.tpm` 向虚拟机添加虚拟 TPM：

```
spec:
  template:
    spec:
      domain:
        devices:
          tpm: {}
        firmware:
          bootloader:
            efi:
              secureBoot: true    # Windows 11 also requires UEFI + Secure
Boot
```

请注意，vTPM 的持久状态不包含在虚拟机快照或克隆中。

虚拟机热迁移

目录

概述

Pre-Copy

约束与限制

前提条件

操作步骤

部署 kubevirt-operator

创建 HyperConverged 实例

准备虚拟机

开始热迁移

使用 API

迁移设置

集群默认值

命名空间策略

概述

虚拟机热迁移技术允许在不关闭或不中断虚拟机的情况下，将虚拟机从一台物理服务器迁移到另一台物理服务器。平台的虚拟机方案基于开源组件 KubeVirt 实现，默认使用 **pre-copy** 内存迁移。

Pre-Copy

pre-copy 内存迁移是一种常用的虚拟机迁移技术，它通过预先复制虚拟机的内存数据，确保迁移过程中的服务连续性。具体过程如下：

1. 初始阶段：在迁移开始时，源主机会在虚拟机继续运行的同时，将虚拟机的内存页复制到目标主机。由于虚拟机仍在运行，部分内存页可能会在复制过程中被修改。
2. 迭代复制：源主机反复将已修改的内存页复制到目标主机，直到已修改页的数量下降到可接受的水平。每一轮复制称为一次迭代，未修改的内存页数量会在每次迭代后逐步减少。
3. 停止并复制：当剩余未复制的内存页足够少时，虚拟机会短暂暂停（通常只需几秒到十几秒），在此期间将最后的内存页复制到目标主机，并同步虚拟机的 CPU 和设备状态到目标主机。
4. 恢复运行：虚拟机在目标主机上恢复运行。

约束与限制

建议参与热迁移操作的两台物理机使用相同的硬件配置。如果配置不一致（例如 CPU 型号不同），迁移可能失败。

前提条件

请提前启用相关的虚拟机热迁移功能。

操作步骤

部署 kubevirt-operator

注意：具体步骤和参数说明请参见 [安装](#)。

1. 进入管理员。
2. 在左侧导航栏中，点击应用商店管理 > **Operators**。
3. 点击页面顶部的集群，切换到需要部署 Operator 的集群。

- 在 **OperatorHub** 选项卡中，点击 **KubeVirt HyperConverged Cluster Operator** 卡片上的部署。
- 按需配置参数并点击部署。您可以在已部署选项卡中查看 Operator 的部署状态。

创建 HyperConverged 实例

具体创建步骤请参见 [创建 HyperConverged 实例](#)。

准备虚拟机

注意：建议使用 Kube-OVN Underlay 网络。相关配置请参见 [创建子网 \(Kube-OVN Underlay Network\)](#)。

- 进入容器平台。
- 在左侧导航栏中，点击虚拟化 > 虚拟机。
- 点击创建虚拟机。
- 在基本信息区域点击更多展开更多配置项，然后点击 **Annotations** 对应的 添加，按照以下键值对添加注解。如果网络插件为 Kube-OVN，则无需手动填写该注解。

注意：由于表单限制，请先输入注解的值，再输入注解的键。

Annotation	
Value	true
Key	kubevirt.io/allow-pod-bridge-network-live-migration

- 按需配置其他虚拟机参数。具体参数说明请参见相关产品文档。

参数	说明
Volume Mode	必须使用 Block Mode 。
Storage Class	必须使用 CephRBD block storage type storage class 。
Network Mode	推荐使用 Bridge 。

6. 点击创建。

开始热迁移

注意：仅当虚拟机状态为 **Running** 时，才能启动热迁移。

1. 进入容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟机。
3. 启动热迁移。可通过以下两种方式进行：
 - 在列表中找到需要迁移的虚拟机，点击右侧的 **热迁移**。
 - 在列表中点击需要迁移的虚拟机名称，进入详情页，然后点击 **Actions > 热迁移**。
4. 点击确认。您可以通过虚拟机状态或实时事件查看迁移进度。当状态从 **Migrating** 变为 **Running**，或者实时事件中出现类似 **Migrated: The VirtualMachineInstance migrated to node 10.1.1.1.** 的信息时，表示迁移成功。

使用 API

通过创建一个名为正在运行的虚拟机实例的 `VirtualMachineInstanceMigration` 来启动热迁移：

```
apiVersion: kubevirt.io/v1
kind: VirtualMachineInstanceMigration
metadata:
  name: migrate-web-01
  namespace: demo
spec:
  vmiName: web-01
```

```
kubectl apply -f migration.yaml
# phase progresses Pending -> Scheduling -> Running -> Succeeded
kubectl get virtualmachineinstancemigration migrate-web-01 -n demo -o jsonpath='{.status.phase}'
```

当阶段为 `Succeeded` 时，实例正在目标节点上运行 — 请参见

`VirtualMachineInstance.status.migrationState.targetNode`。要取消正在进行的迁

移，请删除其 `VirtualMachineInstanceMigration` 对象（没有 cancel 字段）：

```
kubectl delete virtualmachineinstancemigration migrate-web-01 -n demo
```

迁移设置

集群范围的热迁移限制以及按命名空间的覆盖项，通过 `HyperConverged` 资源和集群级 `MigrationPolicy` 对象进行配置，如下所述。

集群默认值

集群默认值是 KubeVirt 的 `liveMigrationConfig`，由 HyperConverged operator 管理。请将其设置在 `HyperConverged` 资源上，切勿直接设置在 `KubeVirt` 上，因为 operator 会将该值回写：

```
kubectl get hyperconverged kubevirt-hyperconverged -n kubevirt -o jsonpath  
h='{.spec.liveMigrationConfig}'  
# {"parallelMigrationsPerCluster":5,"parallelOutboundMigrationsPerNode":  
2,  
# "completionTimeoutPerGiB":150,"progressTimeout":150,  
# "allowAutoConverge":false,"allowPostCopy":false}
```

字段	默认值	含义
<code>parallelMigrationsPerCluster</code>	5	集群范围内允许的最大并发迁移数
<code>parallelOutboundMigrationsPerNode</code>	2	每个节点允许的最大并发出站迁移数
<code>completionTimeoutPerGiB</code>	150	迁移在被中止前，每 GiB 对应的秒数
<code>progressTimeout</code>	150	在无进展情况下中止迁移前的秒数

字段	默认值	含义
<code>allowAutoConverge</code>	false	限制 guest CPU 以帮助迁移收敛
<code>allowPostCopy</code>	false	如果 pre-copy 无法收敛，则切换到 post-copy

命名空间策略

集群级 `MigrationPolicy` 会覆盖其 selector 匹配到的命名空间（或虚拟机）的默认值：

```
apiVersion: migrations.kubevirt.io/v1alpha1
kind: MigrationPolicy
metadata:
  name: policy-fast
spec:
  selectors:
    namespaceSelector:
      kubernetes.io/metadata.name: demo
  allowAutoConverge: true
  completionTimeoutPerGiB: 100
```

如果某个虚拟机所属的命名空间未匹配任何策略，则使用集群默认值。

CPU 和内存热插拔

本文档介绍如何在不中断运行的情况下，向正在运行的虚拟机（VM）添加 CPU 和内存，以及同样重要的内容：不同 guest 操作系统对热插拔的响应方式。平台侧对所有 OS 都相同，但 guest 是否真正使用新资源取决于 guest kernel 或 driver。下面标记为 *verified* 的各 OS 结果是在一个运行中的集群上，使用 CentOS 7.9、Ubuntu 20.04、Debian 13 和 Windows Server 2022 guest 测试得出的；关于其他版本（Debian 12、Ubuntu 22.04/24.04、RHEL 8.10/9.4、其他 Windows 版本）的兼容性说明基于上游和厂商文档，而不是这些测试。

目录

工作原理

前提条件

第 1 步：创建具有热插拔余量的 VM

第 2 步：热插拔 CPU

Guest OS 行为 — CPU

第 3 步：热插拔内存

Guest OS 行为 — 内存

向下缩容：哪些可以在线完成，哪些需要重启

已验证行为汇总

故障排除

工作原理

平台的虚拟化基于 KubeVirt，它通过 `LiveUpdate` 滚动更新策略（平台默认值）对运行中的 VM 应用 CPU 和内存更改：

- **CPU 热插拔** 通过增加 CPU **socket** 数量来添加 vCPU。热插拔上限为 `maxSockets`；额外的 socket 会预先配置为可供 QEMU 之后填充的空槽位。
- **内存热插拔** 通过 **virtio-mem** 设备扩展 guest 内存。上限为 `maxGuest`；`guest` 与 `maxGuest` 之间的差值由 virtio-mem 设备提供支持，该设备可在运行时调整大小——双向都可以，只要大小保持在启动时内存之上或等于启动时内存（向下缩减尽力而为，见下文）。
- 这两种更改都通过自动 **live migration** 交付：在你编辑 VM spec 之后，平台会将 VM 迁移到一个新的 `virt-launcher` pod，该 pod 的资源分配与新大小匹配。你不需要自己触发迁移——只需注意，每次热插拔都会产生一次迁移，而且无法进行 live migrate 的 VM 也无法进行热插拔。

进度会在 VirtualMachineInstance (VMI) 上显示：

- Condition `HotVCPUPhase` / `HotMemoryChange` — 正在执行热插拔。
- `status.currentCPUTopology` — domain 当前的 CPU 拓扑。
- `status.memory` — `guestAtBoot` / `guestCurrent` / `guestRequested`；待处理的内存热插拔会显示 `guestRequested > guestCurrent`。
- VM 上的 Condition `RestartRequired` — 请求的更改无法在线应用（例如减少 CPU socket，或者将内存降低到启动时值以下）。

前提条件

- VM 必须能够 **live migrate**：VMI 上的 `LiveMigratable` Condition 必须为 `True`。热插拔依赖 live migration；不可迁移的 VM 不能在线扩容。从存储角度看，这意味着每个基于 **PVC/DataVolume** 的磁盘都需要共享存储，并使用 `ReadWriteMany` 访问模式——包括由 RWO PVC 支持的 CD-ROM。不依赖 PVC 的卷（`containerDisk`、cloud-init、`emptyDisk`）不需要 RWX。非存储因素（host device、某些 CPU 配置）也可能导致 VM 不可迁移；请注意，Condition 消息只会报告它找到的第一个阻塞原因。
- VM 必须有热插拔余量（`maxSockets` / `maxGuest`）。有效上限在 VMI 启动时就固定了——要么来自 VM spec 中显式设置的值，要么在省略时由平台自动计算（见下文）。这些值不能在运行中的 VM 上提高：超过当前上限需要重启。

- 对于 内存 热插拔，guest 必须具备能够协商平台 QEMU 所提供功能集的 virtio-mem driver——尤其是 `UNPLUGGED_INACCESSIBLE` 功能。对于 Linux guest，这是一个 kernel 能力（见下方每个 OS 的结果）；对于 Windows guest，这是 virtio-win package 中的 `viomem` driver，且默认不会安装在现有 VM 上（见下方 Windows 行）。在 Linux guest 中，热插拔尝试后可在 guest 内检查：

```
dmesg | grep virtio_mem
# healthy: virtio_mem virtio1: start address: ..., region size: ...
# too old: virtio_mem virtio1: virtio: device refuses features: 3
```

第 1 步：创建具有热插拔余量的 VM

VM template 中的相关字段：

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
spec:
  template:
    spec:
      domain:
        cpu:
          sockets: 1          # current vCPU count = sockets × cores × threads
          cores: 1
          threads: 1
          maxSockets: 4      # hotplug ceiling: up to 4 sockets without restart
        memory:
          guest: 2Gi         # current guest memory
          maxGuest: 8Gi      # hotplug ceiling: up to 8Gi without restart
```

如果省略 `maxSockets` / `maxGuest`，平台会在 VMI 启动时自动计算这些值（默认情况下是启动值的 4 倍；该比例和绝对上限可通过 `liveUpdateConfiguration` 在集群范围内配置）。计算出的值会落在 VMI 上，而不是 VM spec 上——即使是通过 web console 创建、template 里只有 `resources.requests` / `resources.limits` 的 VM，仍然会获得完整的热插拔余量。尽管如此，仍建议显式设置最大值，这样上限就是一个有意识的选择。

在 guest 内，这个余量在启动时就已经可见：使用上面 spec 创建的 VM 启动后显示 1 个 CPU online，CPU 1–3 为 offline slot，内存为 2Gi。

WARNING

对于一个 VM，如果它的 template 只通过 `resources.requests` / `resources.limits` 表达大小——这正是 web console 生成的内容——那么第一次扩容在不同资源上的行为不同（已针对这类 VM 验证）：

- 内存：向正在运行的 VM 添加 `domain.memory.guest` 会被视为 live update。热插拔会正常进行——不需要重启。
- CPU：template 中没有 `cpu` block，而添加一个（即使只是 `cpu.sockets`）也属于不可在线更新的更改。VM 会获得 `RestartRequired`（`a non-live-updatable field was changed in the template spec`）；因此第一次 CPU 扩容需要重启一次，之后 spec 会携带显式的 `cpu` block，后续每次增加 socket 都会在线应用。
- 不要把它们放在一起：如果一个 patch 同时添加 `cpu` block 和 `memory.guest`，CPU 更改会触发 `RestartRequired`，而该 condition 也会抑制原本可在线应用的内存热插拔。如果你需要在线应用内存，请单独 patch。

第 2 步：热插拔 CPU

增加正在运行的 VM 的 socket 数量：

```
kubectl patch vm <vm-name> -n <namespace> --type=merge \
  -p '{"spec":{"template":{"spec":{"domain":{"cpu":{"sockets":2}}}}}}'
```

接下来会发生什么（通常在约 ~30 秒内完成）：

1. VMI 获得 `HotVCPUChange` Condition。
2. 平台自动将 VM live migrate 到新的 `virt-launcher` pod。
3. `status.currentCPUTopology` 报告新的 socket 数量，QEMU 热添加 vCPU。

观察它：

```
kubectl get vmi <vm-name> -n <namespace> \
-o jsonpath='{.status.currentCPUTopology}'
```

Guest OS 行为 — CPU

对于所有 OS，vCPU 都是在 hypervisor 层面进行热添加的。它是否会立即被使用取决于 guest：

Guest OS	行为（已验证）
CentOS 7.9	新 vCPU 会被自动上线。RHEL 系列带有 <code>40-redhat.rules</code> ，会无条件地将任何热添加的 CPU 上线。迁移完成后几秒内， <code>nproc</code> 就会反映新的数量。
Ubuntu 20.04	新 vCPU 会出现，但保持 offline （ <code>nproc</code> 不变）。Ubuntu 的 <code>40-vm-hotadd.rules</code> 只会在 Hyper-V 和 Xen 上自动上线 CPU——它不匹配 KVM/QEMU guest。
Debian 13	与 Ubuntu 相同——vCPU 已添加，但保持 offline；Debian 根本不带自动上线规则。
Windows Server 2022 (Standard)	新 vCPU 会被自动且立即识别——在 live session 中， <code>%NUMBER_OF_PROCESSORS%</code> 会反映新的数量，无需重启，也无需手动操作。

要在 Ubuntu/Debian（以及其他没有覆盖 KVM 的 udev 规则的发行版）上将 CPU 上线，可以手动上线：

```
echo 1 > /sys/devices/system/cpu/cpu1/online
```

或者安装一条 udev 规则（这正是 RHEL 所提供的做法），以便将来热插拔的 CPU 能自动被识别：

```
cat <<'EOF' > /etc/udev/rules.d/99-kvm-cpu-hotplug.rules
SUBSYSTEM=="cpu", ACTION=="add", TEST=="online", ATTR{online}=="0", ATTR
{online}="1"
EOF
udevadm control --reload
```

第 3 步：热插拔内存

增加正在运行的 VM 的 guest 内存：

```
kubectl patch vm <vm-name> -n <namespace> --type=merge \
  -p '{"spec":{"template":{"spec":{"domain":{"memory":{"guest":"4Gi"}}}}}}'
```

平台会 live migrate VM，并调整 virtio-mem 设备大小。跟踪结果：

```
kubectl get vmi <vm-name> -n <namespace> -o jsonpath='{.status.memory}'
# {"guestAtBoot":"2Gi","guestCurrent":"4Gi","guestRequested":"4Gi"}    <-
success
# {"guestAtBoot":"2Gi","guestCurrent":"2Gi","guestRequested":"4Gi"}    <-
pending: guest cannot take it
```

Guest OS 行为 — 内存

这里，guest driver 支持决定一切。对于无法使用 virtio-mem 的 guest，热插拔不会失败——它会悄然保持 pending：

Guest OS	Kernel / driver	行为（已验证）
Debian 13	6.12	可用。在 patch 之后约 ~30 秒，<code>guestCurrent</code> 达到新大小，所有热插拔的内存都被自动上线，无需任何手动步骤（与 CPU 热插拔不同）。请注意，自动上线是 image 的属性，而不是 virtio-mem driver 本身的属性：该 image 的 kernel 启用了自动 block 上线（<code>/sys/devices/system/memory/auto_online_blocks</code> 返回 <code>online</code>）。在没有此类策略的 guest 上（kernel 默

Guest OS	Kernel / driver	行为（已验证）
		认值、boot 参数或 udev 规则），热插拔的 block 会保持 offline，而 virtio-mem 会暂停后续插入——如果在其他 image 上热插拔卡住，请检查 <code>auto_online_blocks</code> 。
CentOS 7.9	3.10	不生效。这个 kernel 没有 virtio-mem driver。 <code>guestRequested</code> 会无限期地保持在 <code>guestCurrent</code> 之上；不会报错，也不会触发 <code>RestartRequired</code> Condition。
Ubuntu 20.04	5.4 (GA)	与 CentOS 7 相同——该 kernel 早于 virtio-mem 的出现。
Ubuntu 20.04	5.15 (HWE)	仍然不工作，而且更隐蔽： <code>virtio_mem</code> module 会加载，但设备会拒绝它—— <code>dmesg</code> 显示 <code>virtio_mem virtio1: virtio: device refuses features: 3</code> ，因为平台的 QEMU 需要这个 kernel 不具备的 <code>UNPLUGGED_INACCESSIBLE</code> 特性。注意，Ubuntu 22.04 的 GA kernel 也是 5.15，同样不包含该特性——请使用 22.04 HWE kernel 或 Ubuntu 24.04+。
Windows Server 2022	virtio-win <code>viomem</code>	在安装 <code>viomem</code> driver 后可用——已通过 live 验证：8Gi → 10Gi 在线应用，guest 立即报告新的总量，无需重启。如果没有该 driver，virtio-mem PCI device（ <code>PCI\VEN_1AF4&DEV_1058</code> ）会在 Device Manager 中显示错误，且请求会永远保持 pending。该 driver 随平台提供的 virtio-win ISO 一起提供（已通过 virtio-win 0.1.266 验证）：在提升权限的 shell 中安装，然后重试，或等待 pending 请求应用。

从附带的 virtio-win CD-ROM 安装 Windows driver（drive letter 可能不同）：

```
pnputil /add-driver E:\viomem\2k22\amd64\viomem.inf /install
# Get-PnpDevice | ? InstanceId -match 1058 -> Status OK, "VirtIO Viome
m Driver"
```

完整的 virtio-win guest-tools installer（`virtio-win-guest-tools.exe`，位于同一 ISO 上）在当前版本中也会默认安装 `viomem`，因此使用较新 guest-tools 进行部署的 VM 开箱即支持内存热插拔。该 driver 仅提供 x64 版本（较新的 Windows 还提供 ARM64 版本）。

关于 pending 状态的说明：

- pending 请求不会丢失。它会在 VM 下次重启时作为普通启动内存应用（`guestAtBoot` 会变成请求大小——已在 CentOS 7 上验证）。对于没有可用 driver 的 guest，内存“热插拔”会优雅地退化为计划中的冷扩容。或者，也可以通过将 `memory.guest` 重新设置为当前 live 值来取消 pending 请求——该回退会被接受，不会触发 `RestartRequired`，因为实际上并没有缩减到低于启动值。
- 管理员可以从 node 侧确认该机制：在 `virt-launcher` pod 内执行 `virsh dumpxml <domain>`，可以看到一个 `<memory model='virtio-mem'>` 设备，其中 `requested` 大小会跟随每次热插拔变化，而 `current` 反映 guest driver 实际接受的大小（在没有可用 virtio-mem 的 guest 上，`current: 0`）。

向下缩容：哪些可以在线完成，哪些需要重启

这两个资源的行为不同（已在 Linux 和 Windows guest 上验证）：

- 内存可以在线缩减——但只是尽力而为，而且只能缩减到启动时的值为止。将 `memory.guest` 降低到任意大于等于 `guestAtBoot` 的值都会被在线接受：不会触发 `RestartRequired`，会进行一次自动 live migration，并要求 virtio-mem 设备归还内存。guest 是否完全配合不能保证——这取决于 guest 实际能够释放多少热插拔范围内的内存（内存压力、不可移动的 kernel 分配、锁页以及上线策略都会影响结果）。务必通过比较 `guestCurrent` 与 `guestRequested` 来确认结果。在空闲 guest 上，我们的测试中完整缩减在一分钟内完成（Debian 13 上 4Gi → 3Gi，Windows Server 2022 上 10Gi → 9Gi → 8Gi）；在一个受到人为内存压力的 Debian 13 guest 上，同样的请求会停在半路（`guestRequested: 2Gi`，`guestCurrent` 卡在约 ~2.1Gi），没有错误也没有 condition——而且即使压力解除后，仍会有少量剩余无法回收，因为 kernel 分配已经落在热插拔的 block 中。部分完成的缩减并不是失败状态：已释放的部分会被归还，请求保持 pending，你可以随时把 spec 回退到当前 live 值。
- 将内存降低到 `guestAtBoot` 以下无法在线完成：VM 会获得 `RestartRequired`（`memory updated in template spec to a value lower than what the VM started with`），在重启之前不会有任何变化。换句话说，在线缩减只能尝试回收之前热插拔的内存，不能回收启动内存。
- CPU 不能在线缩减。任何对 `sockets` 的减少都会设置 `RestartRequired`（`Reduction of CPU socket count requires a restart`）；在重启之前，拓扑不会发生变化。

将 spec 回退到当前 live 值会清除 pending 的 `RestartRequired` Condition。

已验证行为汇总

操作	平台行为	CentOS 7.9 (3.10)	Ubuntu 20.04 (5.4 / HWE 5.15)	Debian 13 (6.12)
CPU 热插拔 (sockets ↑)	通过自动 live migration 应用	vCPU 自动上线	已添加但处于 offline — 需要手动上线或 udev 规则	已添加但处于 offline — 需要手动上线或 udev 规则
内存热插拔 (guest ↑)	通过自动 live migration 调整 virtio-mem 大小	永远 pending ; 下次重启时应用	永远 pending (5.15 特性协商失败) ; 下次重启时应用	约 ~30 秒内在线完成, block 自动上线
内存向下缩容 (guest ↓, ≥ 启动值)	通过自动 live migration 在线请求; 是否完成取决于 guest 是否释放内存	— (没有可用 driver)	— (没有可用 driver)	空闲时在线完成; 内存压力下会部分停滞 (无错误)
内存向下缩容 (guest ↓, < 启动值)	不支持在线—— RestartRequired	—	—	—
CPU 向下缩容 (sockets ↓)	不支持在线—— RestartRequired	—	—	—

关于内存热插拔的实用建议：关键在于 guest driver 是否支持 `UNPLUGGED_INACCESSIBLE` virtio-mem 特性，而且发行版 kernel 会独立于版本号回移植该特性——所以不要只看 `uname -r`。已知可用的 guest 包括 Debian 12+、Ubuntu 24.04+（或使用 HWE kernel 的 22.04——GA 5.15 kernel 不可用），以及回移植了该特性的 RHEL 系列版本（Red Hat 文档说明 RHEL 9.4+ 和 8.10 支持 virtio-mem guest；但请注意，RHEL 8.10 默认只支持 hot-plug：除非 guest 使用 `memhp_default_state=online_movable` 启动，否则内存 unplug 默认被禁用）；经验证的 Windows Server 2022 guest 在安装平台 virtio-win ISO 中的 `viomem` driver 后即可正常工作。如有疑问，请先测试一次热插拔，并按上文所示检查 `dmesg` / Device Manager。对于 CPU 热插拔，本例中测试的四个 guest 在 hypervisor 侧的热添加都成功了；但 guest 是否无需额外步骤就使用这些 vCPU 则各不相同——RHEL 系列和 Windows Server 2022 可以直接使

用，Debian/Ubuntu 需要上面的 udev 规则，其他 Windows 版本可能不同（Microsoft 将 CPU hot-add 与特定 Server SKU 中的 dynamic-hardware-partitioning 支持关联——在依赖之前，请先在你的版本上验证）。

故障排除

- 热插拔似乎卡住了 —— 检查 VM 是否可以 live migrate（VMI 上的 `LiveMigratable` Condition，命名空间中的 migration 事件）。热插拔是通过 live migration 交付的，因此任何阻止迁移的因素（基于 RWO 的 PVC 磁盘、节点限制）都会阻止热插拔。Condition 消息只显示第一个阻塞原因——修复一个问题后可能会暴露另一个问题。
- `guestRequested` > `guestCurrent`（增长请求卡住了） —— guest 没有接收内存。在 Linux 上运行 `dmesg | grep virtio_mem`：`device refuses features` 表示 driver 缺少 `UNPLUGGED_INACCESSIBLE` 特性；完全没有输出则表示 driver 根本没有绑定——这可能是一个不支持 virtio-mem 的 kernel，也可能只是 module 没有加载，因此在断定 kernel 太旧之前，请先检查 `lsmod | grep virtio_mem`。如果 driver 本身没问题，请检查上线策略（`cat /sys/devices/system/memory/auto_online_blocks`；如果新增 block 保持 offline，virtio-mem 会暂停插入）。在 Windows 上，增长卡住意味着 `viomem` driver 缺失、版本过旧、未绑定，或处于错误状态——在这些情况下，Device Manager 都会将 virtio-mem PCI device（`PCI\VEN_1AF4&DEV_1058`）显示为错误状态。按上文所示，从平台的 virtio-win ISO 安装当前 driver 可以覆盖常见情况；pending 请求随后会在无需重启的情况下应用。无论如何，请求都不会丢失——要么升级 guest，要么计划一次重启来应用 cold memory。
- `guestRequested` < `guestCurrent`（缩减请求卡住了） —— 通常是 guest 无法释放热插拔范围的那部分内存（正在使用的内存、锁页，或落入热插拔 block 的不可移动 kernel 分配）——但首先要确认 guest 确实支持并且已启用内存 unplug：有些能够正常热插拔的 guest 默认禁用了 unplug（例如 RHEL 8.10 需要通过 `memhp_default_state=online_movable` 启动）。请求会在后台持续重试，之后可能完成、部分完成，或永远不能完全完成。降低 guest 内存压力，接受部分结果，或者将 `memory.guest` 回退到当前 live 值以取消剩余部分。
- 新的 vCPU 在 `nproc` 中不可见 —— 这说明它已热添加，但仍处于 offline。检查 `cat /sys/devices/system/cpu/online` 与 `offline`，然后按上文所示手动上线或通过 udev 规则上线。
- VM 上出现 `RestartRequired` —— 请求的更改无法在线应用（CPU 减少、内存低于启动值、首次添加 `cpu` block，或者提高 `maxSockets` / `maxGuest`）。要么回退 spec 以清除

该状态，要么重启 VM 来应用更改。

虚拟机恢复

在某些场景下，例如错误修改 `fstab` 或文件系统错误需要 `fsck`，虚拟机可能无法正常启动。此时，可以利用救援模式修复根文件系统（`rootfs`）或从系统中取回数据。

目录

操作步骤

获取镜像地址

修改虚拟机 YAML 文件

挂载原 `rootfs` 并进行修复

恢复虚拟机 YAML 文件

操作步骤

获取镜像地址

1. 在左侧导航栏点击 虚拟化管理 > 虚拟机镜像。
2. 选择平台提供的 来源 为 镜像仓库，操作系统选择 **CentOS** 或 **Ubuntu**。右侧点击：**> 更新**。
3. 复制并保存 镜像地址。本文以 `192.168.1.1:11443/3rdparty/vmdisks/centos:7.9` 为例。

4. 点击 取消。

修改虚拟机 YAML 文件

1. 进入 容器平台。
2. 在左侧导航栏点击 虚拟化 > 虚拟机。
3. 在需要修复的虚拟机右侧点击 ⋮ > 停止 或 强制停止。
4. 在虚拟机右侧点击 ⋮ > 更新。
5. 切换到 **YAML**，修改以下字段。

- 在 `spec.template.spec.domain.devices.disks` 下添加如下内容。添加 `bootOrder` 参数可以控制虚拟机启动时磁盘的优先级，`bootOrder` 值越小优先级越高。

注意：如果原 `spec.template.spec.domain.devices.disks` 字段中已有 `bootOrder: 1`，请将原值调大，确保新添加的 `bootOrder` 值小于原值。

```
disks:
  - bootOrder: 1
    disk:
      bus: virtio
      name: containerdisk
```

修改后的 YAML 示例：

```

domain:
  devices:
    disks:
      - bootOrder: 1 # 新增字段
        disk:
          bus: virtio
          name: containerdisk
      - disk:
          bus: virtio
          name: cloudinitdisk
      - disk: # 将原 bootOrder: 1 值调大
          bus: virtio
          name: rootfs
          bootOrder: 10
      - disk:
          bus: virtio
          name: "1"

```

- 在 `spec.template.spec.volumes` 下添加如下内容。

注意：请将以下 `image` 字段中的镜像地址替换为从[获取镜像地址](#)中获得的地址。

```

- containerDisk:
    image: 192.168.1.1:11443/3rdparty/vmdisks/centos:7.9
    name: containerdisk

```

修改后的 YAML 示例：

```

volumes:
  - containerDisk: # 新增字段
      image: 192.168.1.1:11443/3rdparty/vmdisks/centos:7.9
      name: containerdisk
  - dataVolume:
      name: k2-rootfs
      name: rootfs
  - dataVolume:
      name: k2-1
      name: "1"

```

6. 点击 更新。

注意：修改 YAML 文件后，不要切换到 表单，直接点击 更新 即可。

7. 在虚拟机右侧点击： 启动。

挂载原 **rootfs** 并进行修复

1. 使用原密码或密钥登录虚拟机，执行命令 `df -h /`，会发现 **rootfs** 文件系统已被替换。此时可以使用挂载相关命令挂载，或使用 `fsck` 相关命令检查并修复原文件系统。
2. 修复完成后，关闭虚拟机。

恢复虚拟机 **YAML** 文件

按照[修改虚拟机 YAML 文件](#)中的步骤，将虚拟机 **YAML** 文件恢复至原始状态。此时虚拟机即可正常启动。

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [虚拟机](#) > [实用指南](#) > 在 KubeVirt 上克隆虚拟机

在 KubeVirt 上克隆虚拟机

本文档提供使用 KubeVirt 的 `VirtualMachineClone` API 克隆虚拟机（VM）的分步指导。

目录

确保先决条件

快速开始

理解 VirtualMachineClone 对象

查看完整的 VirtualMachineClone 示例

理解各字段

检查 Clone 操作阶段

确保先决条件

在开始 VM 克隆操作之前，请确保满足以下要求：

- 支持快照的存储：Clone API 依赖 Snapshot & Restore 功能。虚拟机的存储类必须支持 **volume snapshots**，并且必须为该存储后端显式启用快照功能。

快速开始

按照以下快速步骤克隆 VM：

1. 准备 **Clone** 清单：

创建一个名为 `clone.yaml` 的文件，内容如下：

```
apiVersion: clone.kubevirt.io/v1beta1
kind: VirtualMachineClone
metadata:
  name: example-vm-clone
  namespace: ns-where-vm-run
spec:
  source:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: source-vm
  target:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: target-vm
```

2. 执行 **Clone** 操作：

应用该清单：

```
kubectl create -f clone.yaml
```

3. 监控 **Clone** 状态：

等待克隆成功完成：

```
kubectl wait vmclone example-vm-clone --for=jsonpath='{.status.phase}'=
Succeeded --timeout=300s
```

4. 验证克隆后的 **VM**：

检查克隆后的 VM 配置：

```
kubectl get vm target-vm -o yaml
```

5. 修复 **DataVolume** 标签（UI 元数据）：

平台 UI 通过标签 `vm.cpaas.io/used-by=<vm-name>` 将 VM 与其磁盘关联，该标签会自动添加到每个 DataVolume。在执行克隆操作后，新的 DataVolume 会继承 source VM 的标

签，因此 UI 仍会认为它属于旧 VM。请更新新创建的 DV 上的标签，使关联关系能够正确显示（不影响功能）。

```
# List DataVolumes in the VM's namespace; the cloned DV name usually starts with "restore-"
kubectl get datavolumes -n <ns-where-vm-run>

# Overwrite the label to point to the cloned VM
kubectl label datavolume <new-dv-name> -n <ns-where-vm-run> vm.cpaas.io/used-by=<target-vm> --overwrite
```

理解 VirtualMachineClone 对象

查看完整的 VirtualMachineClone 示例

下面是一个完整的 `VirtualMachineClone` 资源示例，并包含详细的行内注释：



```

apiVersion: clone.kubevirt.io/v1beta1
kind: VirtualMachineClone
metadata:
  name: detailed-vm-clone
  namespace: ns-where-vm-run
spec:
  # Source VM details
  source:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: vm-source

  # Target VM details
  target:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: vm-target

  # Filters for labels and annotations copied from source
  labelFilters:
    - "*"
    - "!exclude-key/*"
  annotationFilters:
    - "include-annotations/*"

  # Template filters to manage network annotations.
  # annotationFilters is a WHITELIST: list "*" first to keep all annotations,
  # then negate the ones to drop. On Kube-OVN, drop the per-VM OVN annotations
  # (fixed IP / pod-bridge live-migration) so the clone gets a fresh identity:
  template:
    labelFilters:
      - "*"
    annotationFilters:
      - "*"
      - "!ovn.kubernetes.io/*"

  # Explicitly set new MAC addresses
  newMacAddresses:
    eth0: "02-00-00-aa-bb-cc"

```

```
# Explicitly set SMBios serial
newSMBiosSerial: "unique-serial-1234"

# JSON patches to further customize the cloned VM
patches:
  - '{"op": "add", "path": "/metadata/labels/new-label", "value": "new-value"}'
  - '{"op": "replace", "path": "/spec/template/metadata/annotations/new-annotation", "value": "updated-value"}'
```

理解各字段

- **Source 和 Target :**
 - 定义原始 VM (`source`) 和克隆后的 VM (`target`) 。
 - 如果省略 `target` 名称，则会自动生成。
 - 两个 VM 必须位于同一命名空间中。
- **Label 和 Annotation 过滤器 :**
 - 使用通配符 (`*`) 和否定 (`!`) 控制从 source VM 复制或排除的 label/annotation。
- **Template Label 和 Annotation 过滤器 :**
 - 适用于管理与网络相关的 annotation，尤其是使用 Kube-OVN 等 CNI 时。
- **newMacAddresses :**
 - 可选地为网络接口指定新的 MAC 地址。
 - 如果省略，则会自动重新生成。
- **newSMBiosSerial :**
 - 可选地指定新的 SMBios serial。
 - 如果未提供，则会基于 VM 名称自动生成。
- **JSON Patches :**
 - 直接应用于 VM spec 的高级自定义。

检查 Clone 操作阶段

`VirtualMachineClone` 对象的 `.status.phase` 会根据克隆过程的进度而变化。下表说明了各个阶段：

阶段	说明
SnapshotInProgress	正在为 source VM 创建快照，这是克隆运行中 VM 时的初始步骤。
CreatingTargetVM	快照已完成；正在为 target VM 创建元数据和规格。
RestoreInProgress	正在创建 DataVolume 和 PersistentVolumeClaim，并从快照恢复数据。
Succeeded	操作成功完成。target VM 和存储已就绪。
Failed	操作失败。请检查 <code>events</code> 和 <code>status.conditions</code> 以获取详细错误信息。
Unknown	无法确定克隆操作状态，可能表示控制器存在问题。

在 StorageClass 之间迁移虚拟机

本文档介绍如何将虚拟机（VM）磁盘从一个 StorageClass 迁移到另一个 StorageClass。当你需要将 VM 存储迁移到另一个后端、存储池或 CSI driver 时，可使用此操作步骤，例如从 Ceph RBD 迁移到 CephFS、在两个 Ceph RBD 池之间迁移，或从平台提供的存储迁移到用户提供的 storage class。

可用的迁移方法有两种：

方法	VM 状态	推荐场景
使用 CDI clone 的冷迁移	已停止	当源存储或目标存储仅支持 <code>ReadWriteOnce</code> 、涉及本地存储或拓扑感知存储，或者相比避免停机而言更重视最大兼容性时使用。
Live storage migration	运行中	仅当 VM、源存储、目标存储、网络以及目标节点调度条件都支持 live migration 时使用。

目录

前提条件

准备迁移

使用 DataVolume Clone 进行冷迁移

停止 VM

创建目标 DataVolume

更新 VM 磁盘引用

启动并验证 VM

Live Storage Migration

确认 Live Migration 就绪状态

创建迁移对象

故障排查

没有可用于 Live Storage Migration 的 VM

Live Migration 卡在调度阶段

目标 DataVolume 未完成

Live Storage Migration 未达到 Completed

Migration 资源在清理期间卡住

前提条件

- 已启用 ACP virtualization，并包含 live storage migration 支持。
- 你拥有在 VM 命名空间中管理 `VirtualMachine`、`VirtualMachineInstance`、`PersistentVolumeClaim` 和 `DataVolume` 资源的权限。
- 源 VM 使用 CDI `DataVolume` 或基于 PVC 的磁盘。
- 目标 StorageClass 具有足够容量，并支持所需的 `accessModes` 和 `volumeMode`。
- 对于冷迁移，请规划维护窗口，因为在克隆磁盘并更新 VM 定义期间，VM 会停止。
- 对于 live storage migration，运行中的 VMI 必须同时报告 `LiveMigratable=True` 和 `StorageLiveMigratable=True`。

准备迁移

为 VM 和目标存储设置变量：

```
NAMESPACE=<vm-namespace>
VM=<vm-name>
ROOT_VOLUME=<root-disk-volume-name>
SRC_PVC=<source-pvc-name>
DST_SC=<destination-storage-class>
```

在创建目标磁盘之前，检查目标 StorageProfile：

```
kubectl get storageprofile "$DST_SC" -o yaml
```

从目标 StorageProfile 支持的 claim property sets 中选择 `accessModes` 和 `volumeMode`。例如，Ceph RBD 通常与 `volumeMode: Block` 一起使用，而 CephFS 通常与 `volumeMode: Filesystem` 一起使用。

对于 `VirtualMachineStorageMigrationPlan`，它用于 live storage migration，并在本文档后面介绍，你也可以将 `accessModes` 或 `volumeMode` 设置为 `Auto`，让 storage migration controller 从 StorageProfile 中查找该值。对于手动创建的 `DataVolume`，请使用显式的 Kubernetes PVC 值，例如 `ReadWriteOnce`、`ReadWriteMany`、`Block` 或 `Filesystem`。

如果你计划使用 live storage migration，请检查运行中的 VMI 条件：

```
kubectl get vmi "$VM" -n "$NAMESPACE" \
  -o jsonpath='{range .status.conditions[*]}{.type}={.status}{ " "}{.reason}{ "\n"}{end}'
```

仅当输出同时包含以下两个条件时，才继续执行 live storage migration：

```
LiveMigratable=True
StorageLiveMigratable=True
```

如果任一条件缺失或设置为 `False`，请使用冷迁移，或先解决所报告的限制，再重试 live migration。

使用 DataVolume Clone 进行冷迁移

冷迁移是跨 CSI driver 最兼容的方法。它会停止 VM，将源 PVC 克隆到使用目标 StorageClass 的新 `DataVolume`，然后更新 VM 以使用新的 `DataVolume`。

停止 VM

停止 VM：

```
kubectl patch vm "$VM" -n "$NAMESPACE" --type=merge \
  -p '{"spec":{"running":false}}'
```

注意

某些 VM API 版本会提示 `spec.running` 已弃用，建议改用 `spec.runStrategy`。如果 VM 已经使用 `spec.running`，在先移除 `spec.running` 之前不要设置 `spec.runStrategy`，因为这两个字段互斥。

等待 VMI 被删除：

```
kubectl wait vmi "$VM" -n "$NAMESPACE" --for=delete --timeout=10m
```

创建目标 DataVolume

创建一个 DataVolume，将源 PVC 克隆到目标 StorageClass：

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: <destination-datavolume-name>
  namespace: <vm-namespace>
spec:
  source:
    pvc:
      namespace: <vm-namespace>
      name: <source-pvc-name>
  pvc:
    accessModes:
      - <destination-access-mode>
    volumeMode: <Block-or-Filesystem>
    resources:
      requests:
        storage: <destination-size>
    storageClassName: <destination-storage-class>
```

应用该清单：

```
kubectl apply -f <destination-datavolume-file>.yaml
```

对于 `volumeBindingMode: Immediate` 的 StorageClass，请等待克隆完成：

```
kubectl wait dv <destination-datavolume-name> -n "$NAMESPACE" \
  --for=jsonpath='{.status.phase}'=Succeeded --timeout=60m
```

对于 `volumeBindingMode: WaitForFirstConsumer` 的 StorageClass，DataVolume 可以保持在 `PendingPopulation` 状态，直到 VM 启动并成为第一个 consumer。这对于本地存储或拓扑感知存储是预期行为。继续更新 VM、启动 VM，并监控 DataVolume 和 VMI。

注意

当从 block 模式迁移到 filesystem 模式时，请申请比源磁盘更大的目标大小。增加 5% 到 10% 是一个实用的起点，因为目标 filesystem 和磁盘映像文件需要额外空间。

更新 VM 磁盘引用

更新 VM，使根磁盘卷指向目标 DataVolume。

在编辑 `spec.dataVolumeTemplates` 和 `spec.template.spec.volumes` 之类的数组时，请保留所有必需条目。如果 VM 还有其他磁盘、cloud-init 卷、secret 或 config map，请将它们包含在最终的 VM 规格中，避免被移除。

示例 patch 内容：

```

spec:
  dataVolumeTemplates:
    - metadata:
        name: <destination-datavolume-name>
      spec:
        source:
          pvc:
            namespace: <vm-namespace>
            name: <source-pvc-name>
        pvc:
          accessModes:
            - <destination-access-mode>
          volumeMode: <Block-or-Filesystem>
          resources:
            requests:
              storage: <destination-size>
            storageClassName: <destination-storage-class>
  template:
    spec:
      volumes:
        - name: <root-disk-volume-name>
          dataVolume:
            name: <destination-datavolume-name>
        - name: <other-volume-name>
          <other-volume-source>: {}

```

应用该 patch :

```

kubectl patch vm "$VM" -n "$NAMESPACE" --type=merge --patch-file=<vm-patch-file>.yaml

```

启动并验证 VM

启动 VM :

```

kubectl patch vm "$VM" -n "$NAMESPACE" --type=merge \
  -p '{"spec":{"running":true}}'

```

检查 VM、VMI、PVC 和 DataVolume 状态：

```
kubectl get vm,vmi,pvc,dv -n "$NAMESPACE"
```

如果目标使用 `WaitForFirstConsumer`，请监控 DataVolume 和 PVC，直到 VM 启动：

```
kubectl get dv <destination-datavolume-name> -n "$NAMESPACE" -w
kubectl describe pvc <destination-pvc-name> -n "$NAMESPACE"
```

在 VM 运行且 guest workload 检查通过后，仅当确认旧 DataVolume 或 PVC 不再被 VM 引用时，再删除它们。

```
kubectl delete dv <old-datavolume-name> -n "$NAMESPACE"
```

Live Storage Migration

仅当 VM 必须保持运行且满足所有 live migration 前提条件时，才使用 live storage migration。

确认 Live Migration 就绪状态

检查 VMI 条件：

```
kubectl get vmi "$VM" -n "$NAMESPACE" \
  -o jsonpath='{range .status.conditions[*]}{.type}={.status}{ " "}{.reason}{ "\n"}{end}'
```

仅当输出包含 `LiveMigratable=True` 和 `StorageLiveMigratable=True` 时，才继续。

同时确认以下要求：

- 目标 StorageClass 可以使用所选的 `accessModes` 和 `volumeMode` 创建 PVC。
- VM 可以调度到多个节点。

- VM 在 `.status.conditions` 中没有报告 `RestartRequired=True`。如果 VM 有需要重启才能生效的待处理更改，请先重启 VM 或先解决这些待处理更改，再开始 live storage migration。
- 如果集群存在混合 CPU model，则 VM 使用所有候选节点都支持的 CPU model。
- 如果 VM 在 pod network 上使用 bridge binding，则 VM 具有以下 annotation：

```
kubevirt.io/allow-pod-bridge-network-live-migration: 'true'
```

创建迁移对象

ACP console 是首选入口，因为它会在创建迁移资源之前验证 VM 和目标存储选项。

如果你使用 Kubernetes manifest，请创建 `VirtualMachineStorageMigrationPlan` 和 `VirtualMachineStorageMigration`。仅创建 plan 不会执行迁移——migration 对象才是触发器：

```

apiVersion: migrations.kubevirt.io/v1alpha1
kind: VirtualMachineStorageMigrationPlan
metadata:
  name: <migration-plan-name>
  namespace: <vm-namespace>
spec:
  # keepSource (default) keeps the original PVCs after a successful migration;
  # the source storage is NOT auto-deleted – remove it manually once verified.
  retentionPolicy: keepSource
  virtualMachines:
    - name: <vm-name>
      targetMigrationPVCs:
        - volumeName: <root-disk-volume-name>
          destinationPVC:
            name: <destination-pvc-name>
            storageClassName: <destination-storage-class>
            accessModes:
              - <destination-access-mode>
            volumeMode: <Block-Filesystem-or-Auto>
  ---
apiVersion: migrations.kubevirt.io/v1alpha1
kind: VirtualMachineStorageMigration
metadata:
  name: <migration-name>
  namespace: <vm-namespace>
spec:
  virtualMachineStorageMigrationPlanRef:
    apiVersion: migrations.kubevirt.io/v1alpha1
    kind: VirtualMachineStorageMigrationPlan
    name: <migration-plan-name>
    namespace: <vm-namespace>

```

应用该 manifest :

```
kubectl create -f <storage-migration-file>.yaml
```

监控进度 :

```
kubectl get virtualmachinestoragemigration,virtualmachinestoragemigration
plan \
  -n "$NAMESPACE" -w

kubectl get vmim,pvc,dv -n "$NAMESPACE"
```

预期流程如下：

1. ACP 创建目标 PVC 和 DataVolume。
2. ACP 更新 VM 磁盘引用，指向目标 DataVolume。
3. KubeVirt 通过 live migration 执行 workload 更新。
4. ACP 清理已完成的 workload 更新资源。
5. `VirtualMachineStorageMigration` 进入 `Completed`。

当迁移达到 `Completed` 时，VM 规格会引用目标 DataVolume，且 VM 会继续在已迁移的磁盘上运行。在删除任何旧存储对象之前，请先验证 VM、VMI、目标 PVC 和目标 DataVolume：

```
kubectl get vm "$VM" -n "$NAMESPACE" \
  -o jsonpath='{range .spec.template.spec.volumes[*]}{.name}{" -> "}{.dataVolume.name}{"\n"}{end}'}

kubectl get vm,vmi,pvc,dv -n "$NAMESPACE"
```

在迁移成功且 guest workload 检查通过后，只有在旧 DataVolume 或 PVC 不再被 VM 或任何回滚流程引用时，才移除它们。

故障排查

没有可用于 Live Storage Migration 的 VM

描述 migration plan：

```
kubectl describe virtualmachinestoragemigrationplan <migration-plan-name>
-n "$NAMESPACE"
```

如果 plan 报告 `No virtual machines are ready for storage migration`，请检查 VM 是否在运行，以及 VMI 是否同时报告 `LiveMigratable=True` 和 `StorageLiveMigratable=True`。

如果任一条件为 false，请使用冷迁移，或先修复底层原因。同时确认目标 `accessModes` 和 `volumeMode` 是否受目标 StorageProfile 支持，以及 VM 是否报告 `RestartRequired=True`。常见原因包括非共享 PVC、本地存储、不受支持的网络绑定、不受支持的目标 PVC 属性、需要重启才能生效的待处理 VM 更改，以及节点调度约束。

Live Migration 卡在调度阶段

描述处于 pending 状态的 launcher pod 或 `VirtualMachineInstanceMigration`：

```
kubectl get pod -n "$NAMESPACE" | grep virt-launcher
kubectl describe vmim <vmim-name> -n "$NAMESPACE"
```

常见原因包括：

- 目标 launcher pod 无法调度到与源节点不同的节点上。
- VM 使用了仅存在于源节点上的 CPU model 或 CPU feature set。
- 目标 PVC 无法在目标节点上附加或挂载。

请使用可移植的 CPU model，并确认目标存储支持所需的访问模式。

目标 DataVolume 未完成

检查 DataVolume 和 PVC events：

```
kubectl describe dv <destination-datavolume-name> -n "$NAMESPACE"
kubectl describe pvc <destination-pvc-name> -n "$NAMESPACE"
```

对于 `WaitForFirstConsumer` 存储，在 consumer pod 被调度之前，`PendingPopulation` 可能是预期状态。对于 block 到 filesystem 的迁移，如果 CDI 报告目标 PVC 过小，请增大申请的目标大小并重试。

Live Storage Migration 未达到 Completed

如果 `VirtualMachineStorageMigration` 没有进入 `Completed`，请同时检查 storage migration 资源和底层 KubeVirt migration：

```
kubectl describe virtualmachinestoragemigration <migration-name> -n "$NAMESPACE"
kubectl get vmim -n "$NAMESPACE"
kubectl describe vmim <vmim-name> -n "$NAMESPACE"
```

如果 `VirtualMachineInstanceMigration` 仍在运行，请将其作为 live migration 问题继续排查。如果 `VirtualMachineInstanceMigration` 已经成功，但 storage migration 资源仍未完成，请在联系支持之前收集 migration 资源、migration plan、VM、VMI、PVC、DataVolume 和 controller 日志。

Migration 资源在清理期间卡住

先删除 migration 资源：

```
kubectl delete virtualmachinestoragemigration <migration-name> -n "$NAMESPACE" --wait=false
kubectl delete virtualmachinestoragemigrationplan <migration-plan-name> -n "$NAMESPACE" --wait=false
```

如果某个 finalizer 阻止删除，请仅在确认没有活动迁移仍在运行后再移除它：

```
kubectl patch virtualmachinestoragemigration <migration-name> -n "$NAMESPACE" --type=json \
  -p='[{"op":"remove","path":"/metadata/finalizers"}]'
```

```
kubectl patch virtualmachinestoragemigrationplan <migration-plan-name> -n "$NAMESPACE" --type=json \
  -p='[{"op":"remove","path":"/metadata/finalizers"}]'
```

在中断的 live storage migration 之后重新启动 VM 之前，请确认 VM 引用的是哪个 DataVolume：

```
kubectl get vm "$VM" -n "$NAMESPACE" \
  -o jsonpath='{range .spec.template.spec.volumes[*]}{.name}{ " -> "}{.dataVolume.name}{ "\n"}{end}'
```

如果 VM 规格已经更新，但迁移未完成，则在启动 VM 之前，或者完成迁移，或者将 VM 磁盘引用恢复为预期的 DataVolume。

为虚拟机配置高可用性

目录

运行策略和驱逐策略

概览

术语表

组件概览

fencing 和 remediation 过程中的事件流程

操作步骤

Operator 列表

部署 Self Node Remediation Operator

配置 Self Node Remediation Operator (可选)

配置 Self Node Remediation Template (可选)

部署 Node Health Check Operator

创建 NodeHealthCheck 实例

验证 (可选)

运行策略和驱逐策略

虚拟机规格中的两个字段决定了虚拟机在故障以及节点维护期间的行为：

- 运行策略（`spec.runStrategy`）控制期望的电源状态和重启行为：`Always`（保持运行，如果停止则重启）、`RerunOnFailure`（仅在故障后重启）、`Manual`（状态仅由 start/stop/restart 控制）以及 `Halted`（保持停止）。ACP 虚拟机使用更简单的 `spec.running` 布尔值（`true` / `false`），其分别等同于 `Always` / `Halted`。
- 驱逐策略（`spec.template.spec.evictionStrategy`）控制节点因维护或升级而被排空时会发生什么：
 - `LiveMigrate` — 将虚拟机实时迁移到另一节点，而不是将其终止。
 - `LiveMigrateIfPossible` — 如果虚拟机可迁移，则执行实时迁移；否则行为与 `None` 相同。
 - `None` — 不采取任何特殊操作（虚拟机随节点一起关闭）。

对于不可迁移的虚拟机（例如使用 RWO 存储或直通设备的虚拟机），如果设置了 `evictionStrategy: LiveMigrate`，将阻塞节点排空——因此也会阻塞集群升级——直到该虚拟机被停止或手动迁移为止。对于必须在维护期间继续运行的虚拟机，请规划 RWX 存储（参见 [在存储类之间迁移虚拟机](#)），或者设置 `LiveMigrateIfPossible` 以避免阻塞。

概览

硬件并不完美，软件也会包含 bug。当发生节点级故障时，例如内核挂起或网络接口控制器（NIC）故障，集群所需完成的工作并不会减少，受影响节点上的工作负载需要在其他位置重新启动。但是，某些工作负载，例如 ReadWriteOnce (RWO) 卷和 StatefulSets，可能需要至多一个语义。

影响这些工作负载的故障可能导致数据丢失、数据损坏，或两者兼有。在启动工作负载恢复（即 remediation）之前，确保节点处于安全状态非常重要，这一过程称为 fencing；理想情况下，也应同时恢复节点本身。

并不总是现实可行地依赖管理员介入来确认节点和工作负载的真实状态。为便于这种介入，Alauda Container Platform 提供了多个组件，用于自动化故障检测、fencing 和 remediation。

术语表

缩写	术语
SNR	Self Node Remediation
NHC	Node Health Check

组件概览

- **Self Node Remediation Operator**

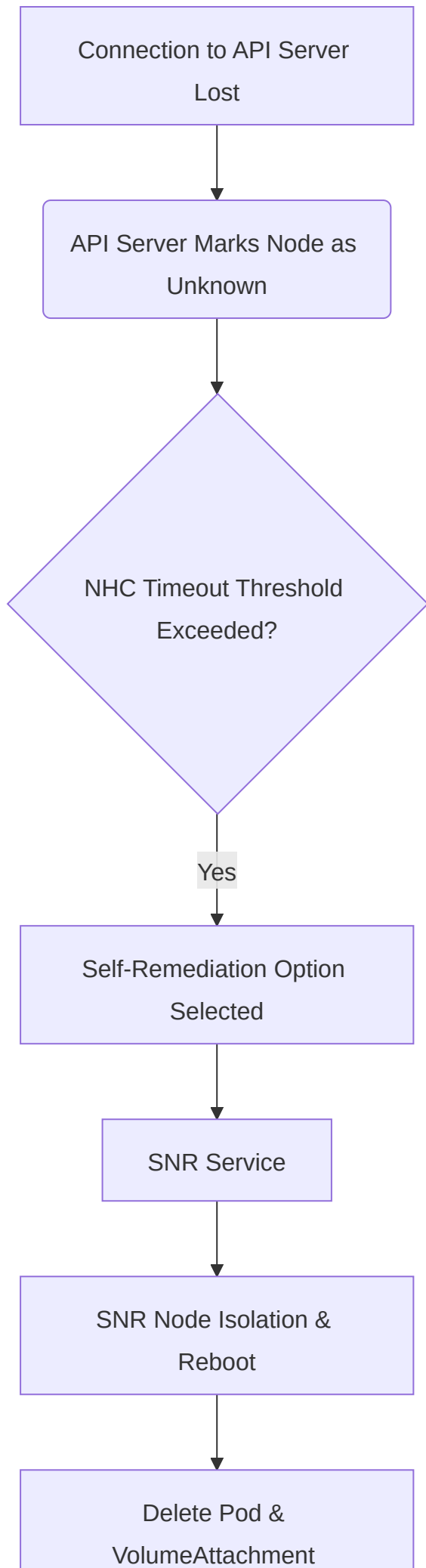
Self Node Remediation Operator 是 Alauda Container Platform 的一个附加 Operator，它实现了一套外部的 fencing 和 remediation 系统，用于重启不健康节点，并删除资源，例如 Pods 和 VolumeAttachments。重启可确保工作负载被隔离，而删除资源可加快受影响工作负载的重新调度。与其他外部系统不同，Self Node Remediation 不需要任何管理接口，例如 Intelligent Platform Management Interface (IPMI) 或用于节点供应的 API。

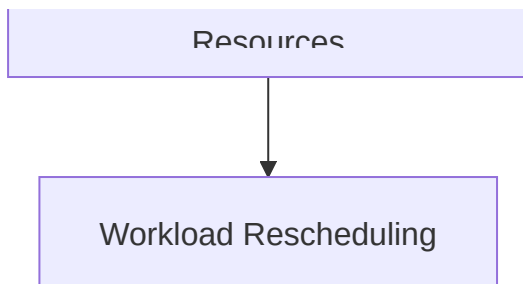
Self Node Remediation 可被故障检测系统使用，例如 Machine Health Check 或 Node Health Check。

- **Node Health Check Operator**

Node Health Check Operator 是 Alauda Container Platform 的一个附加 Operator，它实现了一套监控节点状态的故障检测系统。它不内置 fencing 或 remediation 系统，因此必须与一个提供这些功能的外部系统进行配置。默认情况下，它配置为使用 Self Node Remediation 系统。

fencing 和 remediation 过程中的事件流程





操作步骤

1 Operator 列表

- 下载与您的平台架构对应的 **Alauda Build of SelfNodeRemediation** 安装包。
- 使用上架软件包机制上架 **Alauda Build of SelfNodeRemediation** 安装包。
- 下载与您的平台架构对应的 **Alauda Build of NodeHealthCheck** 安装包。
- 使用上架软件包机制上架 **Alauda Build of NodeHealthCheck** 安装包。

2 部署 Self Node Remediation Operator

1. 登录，进入管理员页面。
2. 单击 **Marketplace > OperatorHub** 进入 **OperatorHub** 页面。
3. 找到 **Alauda Build of SelfNodeRemediation**，单击 **Install**，并进入 **Install Alauda Build of SelfNodeRemediation** 页面。

配置参数：

参数	推荐配置
Channel	默认 channel 为 <code>stable</code> 。
Installation Mode	<code>Cluster</code> ：集群中的所有命名空间共享一个用于创建和管理的 Operator 实例，从而降低资源使用。
Installation Place	选择 <code>Recommended</code> ，Namespace 仅支持 workload-availability 。
Upgrade Strategy	<code>Manual</code> ：当 Operator Hub 中有新版本时，需要手动确认才能将 Operator 升级到最新版本。

3 配置 Self Node Remediation Operator (可选)

Self Node Remediation Operator 会创建名为 `self-node-remediation-config` 的 `SelfNodeRemediationConfig` CR。该 CR 会在 Self Node Remediation Operator 的命名空间中创建。

注意

对 `SelfNodeRemediationConfig` CR 的更改会重新创建 Self Node Remediation daemon set。

`SelfNodeRemediationConfig` CR 的示例如下 YAML 文件所示：

```
apiVersion: self-node-remediation.medik8s.io/v1alpha1
kind: SelfNodeRemediationConfig
metadata:
  name: self-node-remediation-config
  namespace: workload-availability
spec:
  safeTimeToAssumeNodeRebootedSeconds: 180
  watchdogFilePath: /dev/watchdog
  isSoftwareRebootEnabled: true
  apiServerTimeout: 15s
  apiCheckInterval: 5s
  maxApiErrorThreshold: 3
  peerApiServerTimeout: 5s
  peerDialTimeout: 5s
  peerRequestTimeout: 5s
  peerUpdateInterval: 15m
  hostPort: 30001
  customDsTolerations:
    - effect: NoSchedule
      key: node-role.kubernetes.io/infra
      operator: Equal
      value: "value1"
      tolerationSeconds: 3600
```

参数

参数	描述
safeTimeToAssumeNodeRebootedSeconds	指定一个可选时间段，Operator 会在该时间段后再恢复运行于不健康节点上的受影响工作负载。在失败节点上仍然运行时就启动替代 Pod，可能会导致数据损坏并破坏一次运行语义。Operator 会根据 <code>ApiServerTimeout</code>、<code>ApiCheckInterval</code>、<code>MaxApiErrorThreshold</code>、<code>PeerDialTimeout</code> 和 <code>PeerRequestTimeout</code> 字段中的值，以及 watchdog 超时和修复时的集群规模，计算一个最小持续时间。
watchdogFilePath	指定节点中 watchdog 设备的文件路径。如果输入的 watchdog 设备路径不正确，Self Node Remediation Operator 会自动检测 softdog 设备路径。如果 watchdog 设备不可用，<code>SelfNodeRemediationConfig</code> CR 将使用软件重启。
isSoftwareRebootEnabled	指定是否要启用不健康节点的软件重启。默认情况下，<code>isSoftwareRebootEnabled</code> 的值为 <code>true</code>。如需禁用软件重启，请将该参数值设置为 <code>false</code>。
apiServerTimeout	指定检查与每个 API server 连通性的超时时长。当该时长到期后，Operator 会开始修复。超时时长必须大于或等于 10 毫秒。
apiCheckInterval	指定检查与每个 API server 连通性的频率。超时时长必须大于或等于 1 秒。
maxApiErrorThreshold	指定阈值。达到该阈值后，节点会开始联系其 peers。阈值必须大于或等于 1 秒。
peerApiServerTimeout	指定 peer 连接 API server 的超时时长。超时时长必须大于或等于 10 毫秒。

参数	描述
peerDialTimeout	指定与 peer 建立连接的超时时长。超时时长必须大于或等于 10 毫秒。
peerRequestTimeout	指定从 peer 获取响应的超时时长。超时时长必须大于或等于 10 毫秒。
peerUpdateInterval	指定更新 peer 信息（例如 IP 地址）的频率。超时时长必须大于或等于 10 秒。
hostPort	指定一个可选值，用于更改 Self Node Remediation agent 用于内部通信的端口。该值必须大于 0。默认值为端口 30001。
customDsTolerations	指定自定义 toleration，Self Node Remediation agent 运行在 DaemonSets 上，以支持对不同类型节点的修复。

注意

- Self Node Remediation Operator 默认会在部署命名空间中创建该 CR。
- 该 CR 的名称必须为 `self-node-remediation-config`。
- 只能有一个 `SelfNodeRemediationConfig` CR。
- 删除 `SelfNodeRemediationConfig` CR 会禁用 Self Node Remediation。

4 配置 Self Node Remediation Template（可选）

Self Node Remediation Operator 还会创建 SelfNodeRemediationTemplate Custom Resource Definition (CRD)。该 CRD 定义了节点的 remediation 策略，旨在更快恢复工作负载。可用的 remediation 策略如下：

- **Automatic**

该 remediation 策略通过让 Self Node Remediation Operator 决定最适合集群的 remediation 策略，从而简化 remediation 过程。该策略会检查集群中是否可用

`OutOfServiceTaint` 策略。如果 `OutOfServiceTaint` 策略可用，Operator 将选择 `OutOfServiceTaint` 策略；如果 `OutOfServiceTaint` 策略不可用，Operator 将选择 `ResourceDeletion` 策略。`Automatic` 是默认的 remediation 策略。

- **ResourceDeletion**

该 remediation 策略会删除节点上的 Pods，而不是删除节点对象。

- **OutOfServiceTaint**

该 remediation 策略会通过向节点上添加 `OutOfServiceTaint`，隐式地删除节点上的 Pods 及其关联的 volume attachments，而不是删除节点对象。

Self Node Remediation Operator 会为 `self-node-remediation-automatic-strategy-template` 策略创建 SelfNodeRemediationTemplate CR，Automatic remediation 策略会使用该 CR。

SelfNodeRemediationTemplate CR 的示例如下 YAML 文件所示：

```
apiVersion: self-node-remediation.medik8s.io/v1alpha1
kind: SelfNodeRemediationTemplate
metadata:
  creationTimestamp: "2022-03-02T08:02:40Z"
  name: self-node-remediation-<remediation_object>-deletion-template
  namespace: workload-availability
spec:
  template:
    spec:
      remediationStrategy: <remediation_strategy>
```

参数

参数	描述
<code>remediation_strategy</code>	值：Automatic、ResourceDeletion、OutOfServiceTaint

5 部署 Node Health Check Operator

1. 登录，进入管理员页面。
2. 单击 **Marketplace > OperatorHub** 进入 **OperatorHub** 页面。

3. 找到 **Alauda Build of NodeHealthCheck**，单击 **Install**，并进入 **Install Alauda Build of NodeHealthCheck** 页面。

配置参数：

参数	推荐配置
Channel	默认 channel 为 <code>stable</code> 。
Installation Mode	<code>Cluster</code> ：集群中的所有命名空间共享一个用于创建和管理的 Operator 实例，从而降低资源使用。
Installation Place	选择 <code>Recommended</code> ，Namespace 仅支持 workload-availability 。
Upgrade Strategy	<code>Manual</code> ：当 Operator Hub 中有新版本时，需要手动确认才能将 Operator 升级到最新版本。

6 创建 NodeHealthCheck 实例

在集群控制节点上执行以下命令：

命令

```
cat << EOF | kubectl apply -f -
apiVersion: remediation.medik8s.io/v1alpha1
kind: NodeHealthCheck
metadata:
  name: nodehealthcheck-<name>
spec:
  minHealthy: <minHealthy>
  remediationTemplate:
    apiVersion: self-node-remediation.medik8s.io/v1alpha1
    kind: SelfNodeRemediationTemplate
    name: self-node-remediation-automatic-strategy-template
    namespace: workload-availability
  selector: <selector>
  unhealthyConditions:
    - duration: 300s
      status: 'False'
      type: Ready
    - duration: 300s
      status: Unknown
      type: Ready
EOF
```

示例

```

cat << EOF | kubectl apply -f -
apiVersion: remediation.medik8s.io/v1alpha1
kind: NodeHealthCheck
metadata:
  name: nodehealthcheck-worker
spec:
  minHealthy: 51%
  remediationTemplate:
    apiVersion: self-node-remediation.medik8s.io/v1alpha1
    kind: SelfNodeRemediationTemplate
    name: self-node-remediation-automatic-strategy-template
    namespace: workload-availability
  selector:
    matchExpressions:
      - key: node-role.kubernetes.io/control-plane
        operator: DoesNotExist
      - key: node-role.kubernetes.io/master
        operator: DoesNotExist
  unhealthyConditions:
    - duration: 300s
      status: 'False'
      type: Ready
    - duration: 300s
      status: Unknown
      type: Ready
EOF

```

参数：

参数	描述
name	资源名称
minHealthy	指定健康节点的最小比例。只有当健康节点比例大于或等于该值时，故障节点才会被修复。默认值为 51%
selector	指定用于匹配待检查和自修复节点的 LabelSelector。请避免在同一个实例中同时指定 control-plane 和 worker 节点

模拟虚拟机运行所在节点的故障，并确认虚拟机会自动调度到其他节点上运行。

从现有虚拟机创建 VM 模板

本文档概述了如何从现有 VM 创建可复用的虚拟机（VM）模板，以便快速部署新的 VM。

目录

前提条件

操作步骤

步骤 1：在虚拟机上进行基本配置

步骤 2：创建 VM 快照

步骤 3：获取磁盘快照资源名称

步骤 4：创建 DataSource 资源

标签参数说明：

步骤 5：使用模板创建新的 VM

前提条件

- 已正确部署并配置的 KubeVirt 环境。
- 可访问 Web Console 和 kubectl 工具。
- 已配置并安装了所需软件的 VM。

操作步骤

步骤 1：在虚拟机上进行基本配置

在 VM 内执行以下步骤：

- 安装 [cloud-init](#) ↗。
- 安装 `qemu-guest-agent`。
- 安装任何所需软件。

安装完成后，运行以下命令清理 cloud-init 数据并关闭 VM：

```
cloud-init clean  
shutdown -h now
```

步骤 2：创建 VM 快照

使用 KubeVirt Web Console：

1. 导航到 **Virtualization > Virtual Machines**。
2. 选择要用作模板的 VM。
3. 单击 **Actions**，选择 **Create Snapshot**，为快照命名，然后确认。

步骤 3：获取磁盘快照资源名称

可通过以下任一方式获取完整的快照资源名称：

- 通过 **Web Console**：
 - 导航到 **Storage > Volume Snapshots**。
 - 在 "Data Source" 下找到并记录完整的快照资源名称。
- 使用 **kubectl**：

```
kubectl get volumesnapshots -n <NAMESPACE>
```

从输出中记录完整的快照资源名称。

步骤 4：创建 DataSource 资源

在 `kube-public` 命名空间中创建以下 DataSource 资源，并将占位符替换为实际的快照名称和命名空间：

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataSource
metadata:
  annotations:
    cpaas.io/display-name: AlaudaOS-Clone
  labels:
    virtualization.cpaas.io/image-os-arch: amd64
    virtualization.cpaas.io/image-os-type: linux
    virtualization.cpaas.io/storage-class: cephrrbd
    virtualization.cpaas.io/access-mode: ReadWriteMany
    virtualization.cpaas.io/size: 30Gi
    virtualization.cpaas.io/volume-mode: Block
  name: alaudaos-clone
  namespace: kube-public
spec:
  source:
    snapshot:
      name: <Your Snapshot Resource Name>
      namespace: <Your Snapshot Namespace>
```

标签参数说明：

键	可选值	说明
virtualization.cpaas.io/image-os-arch	amd64, arm64	VM 操作系统架构
virtualization.cpaas.io/image-os-type	linux, windows	VM 操作系统类型

键	可选值	说明
virtualization.cpaas.io/storage-class	storage class 名称	默认存储类，可在创建 VM 时调整
virtualization.cpaas.io/access-mode	ReadWriteOnce, ReadWriteMany	磁盘访问模式；VM 热迁移请使用 ReadWriteMany
virtualization.cpaas.io/size	容量（Gi、Ti 等）	默认磁盘大小；请指定合适的大小
virtualization.cpaas.io/volume-mode	Block, Filesystem	磁盘卷模式；建议使用 Block 模式以获得更好的性能

重要：

- 确保命名空间为 `kube-public`。
- 这些与磁盘相关的参数可在创建 VM 时修改，但提供默认值可以简化操作流程。

步骤 5：使用模板创建新的 VM

1. 访问 KubeVirt Web Console，进入 **Container Platform > Virtualization > Virtual Machines**。
2. 单击 **Create Virtual Machine**。
3. 在 **Image** 下，将 **Provision Method** 选择为 **Image Instance**。
4. 从下拉列表中选择新创建的 DataSource。
5. 根据需要配置其他参数，并完成 VM 创建流程。

现在，您已成功使用 VM 模板创建并部署新的 VM。

故障排除

Pod 迁移及虚拟机节点异常关闭时	热迁移错误消息与解决方案	启动和镜像引导失败
问题描述		读取来宾启动日志
原因分析		诊断卡住的镜像
解决方案		捕获虚拟机内存

Pod 迁移及虚拟机节点异常关闭恢复

目录

问题描述

原因分析

解决方案

正常关闭时虚拟机 Pod 迁移

异常关闭恢复

问题描述

无论节点是正常关闭还是发生异常宕机，该节点上运行的虚拟机 Pod 都不会自动迁移到其他健康节点。

原因分析

平台基于开源组件 KubeVirt 实现虚拟机方案，但从 KubeVirt 角度无法区分虚拟机真实宕机和网络等原因导致的连接失败。如果盲目将虚拟机迁移到其他节点，可能导致同一虚拟机存在多个实例。

解决方案

维护虚拟机节点时，需按照本文档进行手动操作。无论是正常关闭还是异常宕机，虚拟机 Pod 都必须手动驱逐或强制删除。

注意：以下命令需在对应集群的主节点上执行。

正常关闭时虚拟机 Pod 迁移

1. 在 CLI 工具中执行以下命令获取节点信息，返回信息中的 `NAME` 字段即为 `Node-Name`。

```
kubectl get nodes
```

输出：

NAME	STATUS	ROLES	AGE	VERSION
1.1.1.211	Ready	control-plane,master	99d	v1.28.8

2. （可选）执行以下命令查看该节点下的虚拟机实例。

```
kubectl get vmis --all-namespaces -o wide | grep <Node-Name> # 将命令中的  
<Node-Name> 替换为步骤 1 中获取的 Node-Name
```

输出：

test-test	vm-t-export-clone	13d	Running	1.1.1.1	1.
1.1.1.211	True	False			

3. 在正常关闭前，执行以下命令驱逐即将关闭节点上的所有虚拟机 Pod，若输出如下则表示驱逐成功。

```
kubectl drain <Node-Name> --delete-local-data --ignore-daemonsets=true  
--force --pod-selector=kubevirt.io=virt-launcher # 将命令中的 <Node-Nam  
e> 替换为待关闭节点的 Node-Name
```

输出：

```
Flag --delete-local-data has been deprecated, This option is deprecated
and will be deleted. Use --delete-emptydir-data.
node/1.1.1.211 cordoned
evicting pod test-test/virt-launcher-vm-t-export-clone-hmnkk
pod/virt-launcher-vm-t-export-clone-hmnkk evicted
node/1.1.1.211 drained
```

4. 在所有虚拟机启动到其他节点后，关闭该节点。
5. 节点关闭并重启后，执行以下命令将节点标记为可调度。

```
kubectl uncordon <Node-Name> # 将命令中的 <Node-Name> 替换为已关闭并重启的节点 Node-Name
```

输出：

```
node/1.1.1.211 uncordoned
```

6. 此时，该节点上原有的虚拟机实例已迁移至其他健康节点，节点重启后可用于新的 Pod 调度。

异常关闭恢复

1. 在 CLI 工具中执行以下命令获取节点信息，返回信息中的 `NAME` 字段即为 `Node-Name`。

```
kubectl get nodes
```

输出：

NAME	STATUS	ROLES	AGE	VERSION
1.1.1.211	Ready	control-plane,master	99d	v1.28.8

2. 执行以下命令强制删除该节点上的所有虚拟机 Pod。

```
kubectl get po -A -l kubevirt.io=virt-launcher -o wide | grep <Node-Name> | awk '{print "kubectl delete pod --force -n " $1, $2}' | bash #
```

将命令中的 <Node-Name> 替换为异常宕机节点的 Node-Name。

3. 执行以下命令删除该节点上的 volume attachments。

```
kubectl get volumeattachments.storage.k8s.io | grep <Node-Name> | awk '{print $1}' | xargs kubectl delete volumeattachments.storage.k8s.io #
```

将命令中的 <Node-Name> 替换为异常宕机节点的 Node-Name。

4. 执行以下命令查询异常宕机节点上是否存在带有标签 kubevirt.io=virt-api 的 Pod。

```
kubectl -n kubevirt get po -l kubevirt.io=virt-api -o wide | grep <Node-Name> # 将命令中的 <Node-Name> 替换为异常宕机节点的 Node-Name。
```

若存在，执行以下命令删除这些 Pod。

```
kubectl -n kubevirt get po -l kubevirt.io=virt-api -o name | xargs kubectl -n kubevirt delete --force --grace-period=0
```

5. 执行以下命令查询异常宕机节点上是否存在带有标签 kubevirt.io=virt-controller 的 Pod。

```
kubectl -n kubevirt get po -l kubevirt.io=virt-controller -o wide | grep <Node-Name> # 将命令中的 <Node-Name> 替换为异常宕机节点的 Node-Name。
```

若存在，执行以下命令删除这些 Pod。

```
kubectl -n kubevirt get po -l kubevirt.io=virt-controller -o name | xargs kubectl -n kubevirt delete --force --grace-period=0
```

6. 此时，虚拟机实例将在节点异常关闭后迁移至其他健康节点。

热迁移错误消息与解决方案

错误消息	原因	解决方案
无法迁移未使用 masquerade、未使用带有 <annotation> VM 注解的 bridge，或未使用可迁移插件连接到 pod network 的 VMI	虚拟机的网络配置不支持热迁移。	请检查以下配置： - 检查当前集群使用的 CNI network plugin；推荐使用 Kube-OVN。 - 检查虚拟机对应 YAML 文件中的 <code>metadata.annotations</code> 和 <code>spec.template.metadata.annotations</code> 字段中是否存在 <code>"kubevirt.io/allow-pod-bridge-network-live-migration": "true"</code> 注解；如果不存在，请手动添加。
- 无法迁移 VMI：无法确定 PVC <pvc name> 是否为共享，热迁移要求所有 PVC 都必须共享（使用 ReadWriteMany 访问模式） - 无法迁移 VMI：PVC <pvc name> 不是共享的，热迁移要求所有 PVC 都必须共享（使用	虚拟机的存储类型不支持多节点读写（RWX）访问模式。	虚拟机创建后，相关参数无法修改。因此，请重新创建虚拟机并选择支持多节点读写（RWX）的存储类型；推荐使用 CephRBD block 存储。如果重新创建后问题仍然存在，请联系相关人员寻求帮助。

错误消息	原因	解决方案
ReadWriteMany 访问模式) • 无法迁移 VMI：后端存储 PVC 不是 RWX • 无法迁移使用非共享 HostDisk 的 VMI		
其他错误消息	虚拟机不支持热迁移。	请联系相关人员寻求帮助。

启动和镜像导入故障排查

目录

读取来宾启动日志

诊断卡住的镜像导入

捕获虚拟机内存转储

读取来宾启动日志

当虚拟机无法启动时（kernel panic、缺少 root disk、卡在 bootloader），图形控制台和串行控制台可能无法显示足够信息。启用串行控制台日志记录，以便将来宾的启动输出捕获到 launcher pod 的日志中：

```
spec:
  template:
    spec:
      domain:
        devices:
          logSerialConsole: true
```

虚拟机重启后，launcher pod 会增加一个 `guest-console-log` 容器。使用以下命令读取捕获到的启动输出：

```
kubectl logs <virt-launcher-pod> -c guest-console-log -n <namespace>
```

诊断卡住的镜像导入

处于 **Importing** 状态且一直未完成的可启动卷或虚拟机磁盘，通常存在 **DataVolume** 问题。检查 DataVolume 的 phase 和条件：

```
kubectl get datavolume <name> -n <ns> \
  -o jsonpath='phase={.status.phase} {range .status.conditions[*]}{.type}
={.status} {end}{"\n"}'
# A healthy completed import: phase=Succeeded Bound=True Ready=True Running=False

kubectl describe datavolume <name> -n <ns> # events show pull / auth /
space errors
```

Bound 条件反映 PVC，**Running** 反映 importer pod，**Ready** 表示磁盘可用。

Running=False 且带有消息时，通常指向 importer pod 失败——例如未找到镜像、registry 身份验证失败，或空间不足。

捕获虚拟机内存转储

要分析挂起或崩溃（但仍在运行）的虚拟机，可以将其来宾 RAM 捕获到 PVC 中，供离线分析：

```
virtctl memory-dump get web-01 --claim-name=web-01-memory -n demo
```

转储会写入指定的 PVC，之后你可以下载并检查。请参阅 [使用 virtctl 命令行工具](#)。



Alauda Container Platform > 虚拟化 > 虚拟化 > 网络

网络

介绍

介绍

优势

操作指南

配置网络

配置 IP

通过 IP 直接连接到虚拟机

添加 Service

实用指南

通过 **Network Policy** 控制虚拟机

操作步骤

配置 **SR-IOV**

术语

配置虚拟机网络

前提条件

使用 API

结果验证

约束与限制

前提条件

操作步骤

结果验证

相关说明

操作步骤

[配置多网卡后](#)

前提条件

操作步骤

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [网络](#) > 介绍

介绍

ACP Virtualization With KubeVirt 与 Kube-OVN 深度集成，扩展了对传统虚拟机（VM）网络需求的支持，并针对特定场景优化了性能。

目录

| [优势](#)

优势

- IPv6 支持
完整支持 IPv6。
- 静态 IP 保留
确保虚拟机重启后保持相同的 IP 地址，符合传统虚拟机的使用习惯。
- 多网络模式支持
支持容器网络、SR-IOV 等多种网络模式，满足多样化的用户场景。

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [网络](#) > 操作指南

操作指南

配置网络

配置 IP

通过 IP 直接连接到虚拟机

添加 Service

[Alauda Container Platform](#) > [虚拟化](#) > [虚拟化](#) > [网络](#) > [操作指南](#) > [配置网络](#)

配置网络

目录

配置 IP

通过 IP 直接连接到虚拟机

添加 Service

配置 IP

参考 [配置 IP](#)

通过 IP 直接连接到虚拟机

参考 [准备 Kube-OVN Underlay 物理网络](#)

添加 Service

参考 [添加 Service](#)

实用指南

通过 Network Policy 控制虚拟机

操作步骤

使用 API

结果验证

配置 SR-IOV

术语

约束与限制

前提条件

操作步骤

结果验证

相关说明

配置虚拟机网络

前提条件

操作步骤

配置多网卡虚拟机

前提条件

操作步骤



通过 Network Policy 控制虚拟机网络请求

平台的虚拟机方案基于开源组件 KubeVirt 实现，实际上运行在 Pod 中。通过使用 Network Policy 的功能，可以控制虚拟机的入站和出站请求。

目录

操作步骤

使用 API

结果验证

第一步：创建允许所有流量通过的虚拟机和网络策略

第二步：更新网络策略，将 `www.example.com` 从白名单中移除

操作步骤

1. 进入 **Container Platform**。
2. 在左侧导航栏中，点击 **Network > Network Policies**。
3. 点击 **Create Network Policy**。
4. 按需配置以下参数。

参数	描述
关联方式	- 计算组件：按需选择目标计算组件；建议选择 All 作为目标计算组件。 - 标签选择器：根据 Pod 的标签进行匹配。
方向	Ingress：从外部发送到 Pod 的请求。 Egress：从 Pod 发送到外部的请求；如果要禁止虚拟机请求某个外部地址，请选择此项。
协议	在 TCP 或 UDP 之间进行选择。 注意： - 当在虚拟机中使用域名请求外部服务时，需要添加 UDP 协议白名单，因为 DNS 协议使用 UDP。 - 表单不支持配置 ICMP 协议；启用白名单规则后，ICMP 协议将被禁用，这将导致无法执行 Ping 操作。
访问端口	指定哪些端口的流量可以入站或出站。如果将此字段留空，则默认允许通过所有端口的流量。 注意：这里需要同时为 UDP 和 TCP 协议放行 1053 和 53 端口，以允许 DNS 流量出站；否则，域名解析将失败。
远端类型	指定允许访问的远端类型。可选项包括：计算组件、namespace 和 IP 段。
排除远端	当远端类型为 IP 段 时，从白名单中移除指定的 IP（即禁止访问）。单个 IP 可按 <code>IP/32</code> 形式移除。 注意：此字段仅支持输入 IP；如果域名对应的 IP 不明确，可以使用命令 <code>curl -vvv <domain></code> 请求该域名，并从返回信息中获取对应的 IP 地址。

5. 点击 **Create**。

使用 API

控制台表单映射到标准 Kubernetes `NetworkPolicy` (`networking.k8s.io/v1`)。下面的示例允许 DNS 出站流量以及所有出站 IP 流量，但排除一个地址——这与下方 结果验证 场景的效果相同：

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: vm-egress-allow
  namespace: demo
spec:
  # Association Method = Label Selector: select the virtual machine's pod
  # S.
  podSelector:
    matchLabels:
      vm.kubevirt.io/name: web-01
  policyTypes:
    - Egress
  egress:
    # Allow DNS (both UDP and TCP, ports 53 and 1053) or name resolution
    # fails.
    - to:
        - namespaceSelector: {}
      ports:
        - { protocol: UDP, port: 53 }
        - { protocol: TCP, port: 53 }
        - { protocol: UDP, port: 1053 }
        - { protocol: TCP, port: 1053 }
    # Remote Type = IP Segment (cidr) with an excluded address (Exclude R
    # emote).
    - to:
        - ipBlock:
            cidr: 0.0.0.0/0
            except:
              - 93.184.215.14/32
```

字段映射：关联方式（标签选择器） → `spec.podSelector`（通过 `vm.kubevirt.io/name` 选择虚拟机的 Pod）；方向 → `policyTypes` 以及 `ingress` / `egress`；协议 / 访问端口 → `ports`；远端类型 = IP 段 → `ipBlock.cidr`；排除远端 → `ipBlock.except`。

结果验证

本文通过虚拟机访问 www.example.com 来验证该配置。

第一步：创建允许所有流量通过的虚拟机和网络策略

- 1. 创建虚拟机，详细步骤请参考 [创建虚拟机](#)。
- 2. 在虚拟机的命名空间中配置网络策略，为 TCP 和 UDP 协议添加白名单规则，参数如下：
 - TCP 协议白名单：

参数	描述
关联方式	选择 计算组件。
目标计算组件	选择 All 。
方向	选择 Egress 。
协议	选择 TCP 。
远端类型	选择 IP 段
远端	输入 0.0.0.0/0 ，表示允许所有流量出站。

- UDP 协议白名单规则：

参数	描述
方向	选择 Egress 。
协议	选择 UDP 。
远端类型	选择 IP 段
远端	输入 0.0.0.0/0 ，表示允许所有流量出站。

3. 网络策略创建后，登录虚拟机并执行以下命令请求 www.example.com ↗。

```
curl www.example.com
```

4. 请求成功。

第二步：更新网络策略，将 www.example.com ↗ 从白名单中移除

1. 执行以下命令以获取 www.example.com ↗ 的 IP 地址，结果为 93.184.215.14。

```
curl -vvv www.example.com
```

2. 更新 [第一步](#) 中创建的网络策略，修改后的参数如下：

参数	描述
排除远端	在 TCP 协议白名单规则中，将排除远端参数填入 93.184.215.14/32，表示将 IP 地址 93.184.215.14 从白名单中移除。

3. 更新网络策略后，登录虚拟机并执行以下命令请求 www.example.com ↗。

```
curl www.example.com
```

4. 请求超时，说明排除远端功能已生效。

配置 SR-IOV

通过配置物理服务器节点支持创建带有 SR-IOV（Single Root I/O Virtualization）网卡的虚拟机，实现虚拟机的更低延迟，同时支持独立 IPv6 以及双栈 IPv4/IPv6 功能。

目录

术语

约束与限制

前提条件

Chart

镜像

操作步骤

在物理机 BIOS 中启用 SR-IOV

启用 IOMMU

加载系统内核的 VFIO 模块

创建 VF 设备

绑定 VFIO 驱动

部署 Multus CNI 插件

部署 sriov-network-operator

为物理节点设置 Node 角色标识标签

检查资源是否创建成功

为物理节点设置 SR-IOV 节点特性标签

检查网卡设备支持情况

配置 IP 地址

结果验证

相关说明

CentOS 虚拟机内核参数配置

术语

术语	定义
Multus CNI	作为其他 CNI 插件的中间件，使 Kubernetes 支持 Pod 的多网卡功能。
SR-IOV	允许对节点上的物理 NIC 进行虚拟化，将其拆分为多个 VF 供 Pod 或虚拟机使用，提供更优的网络性能。
VF	从物理 PCI 设备创建的虚拟设备；VF 可以直接分配给虚拟机或容器，类似独立的物理 PCI 设备，显著提升 I/O 性能。

约束与限制

SR-IOV 功能依赖于 glibc，仅支持 glibc 版本 2.34 及以上。但 Kylin V10 和 CentOS 7.x 操作系统均不支持该版本，因此这两个操作系统无法使用 SR-IOV 功能。

前提条件

获取以下 charts 和镜像，并上传至镜像仓库。本文档以仓库地址 `build-harbor.example.cn` 为例。具体获取 charts 和镜像的方法，请联系相关人员。

Chart

- `build-harbor.example.cn/example/chart-sriov-network-operator:v3.15.0`

镜像

- `build-harbor.example.cn/3rdparty/sriov/sriov-network-operator:4.13`
- `build-harbor.example.cn/3rdparty/sriov/sriov-network-operator-config-daemon:4.13`
- `build-harbor.example.cn/3rdparty/sriov/sriov-cni:4.13`
- `build-harbor.example.cn/3rdparty/sriov/ib-sriov-cni:4.13`
- `build-harbor.example.cn/3rdparty/sriov/sriov-network-device-plugin:4.13`
- `build-harbor.example.cn/3rdparty/sriov/network-resources-injector:4.13`
- `build-harbor.example.cn/3rdparty/sriov/sriov-network-operator-webhook:4.13`
- `build-harbor.example.cn/3rdparty/kubectl:v3.15.1`

操作步骤

注意：以下所有命令均在终端执行。

1 在物理机 BIOS 中启用 SR-IOV

配置前，使用以下命令查看主板信息。

```
dmidecode -t 1
# dmidecode 3.3
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0x0100, DMI type 1, 27 bytes
System Information
    Product Name: PowerEdge R620
    Version: Not Specified
    Serial Number: 7SJNF62
    UUID: 4c4c4544-0053-4a10-804e-b7c04f463632
    Wake-up Type: Power Switch
    SKU Number: SKU=NotProvided;ModelName=PowerEdge R620
    Family: Not Specified
```

不同服务器厂商启用 BIOS 中 SR-IOV 的操作不同，请参考对应厂商文档。一般步骤如下：

1. 重启服务器。
2. BIOS POST 期间显示品牌 Logo 时，按 F2 键进入系统设置。
3. 点击 **Processor Settings > Virtualization Technology**，将 **Virtualization Technology** 设置为 **Enabled**。
4. 点击 **Settings > Integrated devices**，将 **SR-IOV Global Enable** 设置为 **Enabled**。
5. 保存配置并重启服务器。

2 启用 IOMMU

不同操作系统启用 IOMMU 的操作不同，请参考对应操作系统文档。本文档以 CentOS 为例。

1. 编辑 `/etc/default/grub` 文件，在 `GRUB_CMDLINE_LINUX` 配置项中添加 `intel_iommu=on iommu=pt`。

```
GRUB_CMDLINE_LINUX="crashkernel=auto rd.lvm.lv=centos/root rhgb quiet intel_iommu=on iommu=pt"
```

2. 执行以下命令生成 `grub.cfg` 文件。

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

3. 重启服务器。
4. 执行以下命令，若输出包含 `IOMMU enabled`，表示启用成功。

```
dmesg | grep -i iommu
```

3 加载系统内核的 VFIO 模块

1. 执行以下命令加载 `vfio-pci` 模块。

```
modprobe vfio-pci
```

2. 加载后，执行以下命令，若能正常显示配置信息，则 VFIO 内核模块加载成功。

CentOS 下检查 VFIO 加载状态

```
lsmod | grep vfio
```

```
vfio_pci          41993  0
vfio_iommu_type1  22440  0
vfio              32657  2 vfio_iommu_type1, vfio_pci
irqbypass        13503  2 kvm, vfio_pc
```

Ubuntu 下检查 VFIO 加载状态

```
cat /lib/modules/$(uname -r)/modules.builtin | grep vfio
```

```
kernel/drivers/vfio/vfio.ko
kernel/drivers/vfio/vfio_virqfd.ko
kernel/drivers/vfio/vfio_iommu_type1.ko
kernel/drivers/vfio/pci/vfio-pci-core.ko
kernel/drivers/vfio/pci/vfio-pci.ko
```

4 创建 VF 设备

1. 执行以下命令查看当前支持的 VF 设备。

```
find /sys -name *vfs*
```

```
/sys/devices/pci0000:00/0000:00:03.0/0000:05:00.1/sriov_totalvfs
/sys/devices/pci0000:00/0000:00:03.0/0000:05:00.1/sriov_numvfs
/sys/devices/pci0000:00/0000:00:03.0/0000:05:00.0/sriov_totalvfs
/sys/devices/pci0000:00/0000:00:03.0/0000:05:00.0/sriov_numvfs
```

输出信息说明：

- **0000:05:00 .1**：SR-IOV 物理网卡 enp5s0f1 的 PCI 地址。
- **0000:05:00 .0**：SR-IOV 物理网卡 enp5s0f0 的 PCI 地址。
- **sriov_totalvfs**：支持的 VF 数量。
- **sriov_numvfs**：当前 VF 数量。

2. 执行以下命令获取物理机的网卡信息。

```
ifconfig
```

```
enp5s0f0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.66.213 netmask 255.255.255.0 broadcast 192.168.66.255
    inet6 1066::192:168:66:213 prefixlen 112 scopeid 0x0<global>
    inet6 fe80::a236:9fff:fe29:6c00 prefixlen 64 scopeid 0x20<link>
    ether a0:36:9f:29:6c:00 txqueuelen 1000 (Ethernet)
    RX packets 13889 bytes 1075801 (1.0 MB)
    RX errors 0 dropped 1603 overruns 0 frame 0
    TX packets 5057 bytes 440807 (440.8 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

enp5s0f1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet6 fe80::a236:9fff:fe29:6c02 prefixlen 64 scopeid 0x20<link>
    ether a0:36:9f:29:6c:02 txqueuelen 1000 (Ethernet)
    RX packets 1714 bytes 227506 (227.5 KB)
    RX errors 0 dropped 1604 overruns 0 frame 0
    TX packets 70 bytes 19241 (19.2 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

3. 执行命令 `ethtool -i <NIC name>` 获取对应物理网卡的 PCI 地址，如下所示。

```

ethtool -i enp5s0f0
driver: ixgbe
version: 5.15.0-76-generic
firmware-version: 0x8000030d, 14.5.8
expansion-rom-version:
bus-info: 0000:05:00.0    ## enp5s0f0 网卡的 PCI 地址
supports-statistics: yes
supports-test: yes
supports-eeprom-access: yes
supports-register-dump: yes
supports-priv-flags: yes

ethtool -i enp5s0f1
driver: ixgbe
version: 5.15.0-76-generic
firmware-version: 0x8000030d, 14.5.8
expansion-rom-version:
bus-info: 0000:05:00.1    ## enp5s0f1 网卡的 PCI 地址
supports-statistics: yes
supports-test: yes
supports-eeprom-access: yes
supports-register-dump: yes
supports-priv-flags: yes

```

4. 执行以下命令创建 VF。本文档以配置 enp5s0f1 网卡为例，若需虚拟化多个网卡，则全部配置。

```

cat /sys/devices/pci0000:00/0000:00:03.0/0000:05:00.1/sriov_totalv
fs    ## 查看支持的 VF 数量
63

echo 8 > /sys/devices/pci0000:00/0000:00:03.0/0000:05:00.1/sriov_n
umvfs ## 设置当前 VF 数量

cat /sys/devices/pci0000:00/0000:00:03.0/0000:05:00.1/sriov_numvfs
## 查看当前 VF 数量
8

```

5. 执行以下命令检查 VF 是否创建成功。

注意：可见配置的 8 个 VF 地址，如 `05:10.1`。这些 VF 地址需补充 域标识符，最终格式为：`0000:05:10.1`。

```
lspci | grep Virtual
00:11.0 PCI bridge: Intel Corporation C600/X79 series chipset PCI
Express Virtual Root Port (rev 05)
05:10.1 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:10.3 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:10.5 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:10.7 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:11.1 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:11.3 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:11.5 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
05:11.7 Ethernet controller: Intel Corporation 82599 Ethernet Cont
roller Virtual Function (rev 01)
```

5 绑定 VFIO 驱动

1. 下载 [绑定脚本](#)，执行 `python3 dpdk-devbind.py -b vfio-pci <带域标识符的 VF 地址>` 命令，将 enp5s0f1 网卡的 8 个 VF 绑定到 vfio-pci 驱动，示例如下。

```
python3 dpdk-devbind.py -b vfio-pci 0000:05:10.1
python3 dpdk-devbind.py -b vfio-pci 0000:05:10.3
python3 dpdk-devbind.py -b vfio-pci 0000:05:10.5
python3 dpdk-devbind.py -b vfio-pci 0000:05:10.7
python3 dpdk-devbind.py -b vfio-pci 0000:05:11.1
python3 dpdk-devbind.py -b vfio-pci 0000:05:11.3
python3 dpdk-devbind.py -b vfio-pci 0000:05:11.5
python3 dpdk-devbind.py -b vfio-pci 0000:05:11.7
```

2. 绑定成功后，执行以下命令检查绑定结果。在输出结果的 **Network devices using DPDK-compatible driver** 区域查找已绑定的 VF，其中 VF 设备 ID 为 `10ed`。


```
python3 dpdk-devbind.py --status
```

```
Network devices using DPDK-compatible driver
```

```
=====
```

```
0000:05:10.1 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:10.3 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:10.5 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:10.7 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:11.1 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:11.3 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:11.5 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
0000:05:11.7 '82599 Ethernet Controller Virtual Function 10ed' drv  
=vfio-pci unused=ixgbevf
```

```
Network devices using kernel driver
```

```
=====
```

```
0000:01:00.0 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=en  
o1 drv=tg3 unused=vfio-pci
```

```
0000:01:00.1 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=en  
o2 drv=tg3 unused=vfio-pci
```

```
0000:02:00.0 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=en  
o3 drv=tg3 unused=vfio-pci
```

```
0000:02:00.1 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=en  
o4 drv=tg3 unused=vfio-pci
```

```
0000:05:00.0 'Ethernet 10G 2P X520 Adapter 154d' if=enp5s0f0 drv=i  
xgbe unused=vfio-pci *Active*
```

```
0000:05:00.1 'Ethernet 10G 2P X520 Adapter 154d' if=enp5s0f1 drv=i  
xgbe unused=vfio-pci
```

```
No 'Baseband' devices detected
```

```
=====
```

```
No 'Crypto' devices detected
```

```
=====
```

```
No 'DMA' devices detected
```

```

=====

No 'Eventdev' devices detected
=====

No 'Mempool' devices detected
=====

No 'Compress' devices detected
=====

No 'Misc (rawdev)' devices detected
=====

No 'Regex' devices detected
=====

```

6 部署 Multus CNI 插件

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Cluster Management > Clusters**。
3. 点击虚拟机集群名称，切换到 **Plugins** 标签页。
 - 部署 **Multus CNI** 插件。

7 部署 sriov-network-operator

执行以下命令部署 sriov-network-operator。

```

REGISTRY=<$registry> # 将 <$registry> 替换为 sriov-network-operator
镜像所在的仓库地址，例如：REGISTRY=build-harbor.example.cn
NICSELECTOR=["<nics>"] # 将 <nics> 替换为网卡名称，例如：NICSELECTOR=["ens802f1","ens802f2"], 多个用逗号分隔
NUMVFS=<numVfs> # 将 <numVfs> 替换为 VF 数量，例如：NUMVFS=8

cat <<EOF | kubectl create -f -
apiVersion: operator.alauda.io/v1alpha1
kind: AppRelease
metadata:
  annotations:
    auto-recycle: "true"
    interval-sync: "true"
  name: sriov-network-operator
  namespace: cpaas-system
spec:
  destination:
    cluster: ""
    namespace: "kube-system"
  source:
    charts:
      - name: <chartName> # 将 <chartName> 替换为实际 chart 路径，例如：name = example/chart-sriov-network-operator
        releaseName: sriov-network-operator
        targetRevision: v3.15.0
        repoURL: $REGISTRY
    timeout: 120
  values:
    global:
      registry:
        address: $REGISTRY
    networkNodePolicy:
      nicSelector: $NICSELECTOR
      numVfs: $NUMVFS
EOF

```

8 为物理节点设置 Node 角色标识标签

注意：执行此操作前，确保 `sriov-network-operator` 的 Pod 正常运行。

1. 进入 **Administrator**。

2. 在左侧导航栏点击 **Cluster Management > Clusters**。
3. 点击集群名称，切换到 **Nodes** 标签页。
4. 点击支持 SR-IOV 的物理节点：> **Update Node Labels**。
5. 设置节点标签如下：

- `node-role.kubernetes.io/worker: ""`

6. 点击 **Update**。

9 检查资源是否创建成功

在 CLI 工具中执行命令 `kubectl -n cpaas-system get sriovnetworknodestates`，检查 `sriovnetworknodestates` 资源是否创建成功。若看到如下类似输出，表示创建成功。若资源创建失败，请检查 Multus CNI 插件和 sriov-network-operator 是否部署成功。

```
kubectl -n cpaas-system get sriovnetworknodestates
NAME                                SYNC STATUS    AGE
192.168.254.88                     Succeeded     5d22h
```

10 为物理节点设置 SR-IOV 节点特性标签

注意：执行此操作前，确保 `sriovnetworknodestates` 资源已成功创建。

1. 进入 **Administrator**。
2. 在左侧导航栏点击 **Cluster Management > Clusters**。
3. 点击集群名称，切换到 **Nodes** 标签页。
4. 点击支持 SR-IOV 的物理节点：> **Update Node Labels**。
5. 设置节点标签如下：

- `feature.node.kubernetes.io/network-sriov.capable: "true"`

11 检查网卡设备支持情况

1. 执行命令 `lspci -n -s <带域标识符的 VF 地址>`，获取当前网卡设备的厂商 ID 和设备 ID，如下所示。

```
lspci -n -s 0000:05:00.1
05:00.1 0200: 8086:154d (rev 01)
```

输出说明：

- **8086**：厂商 ID。
- **154d**：设备 ID。

2. 执行命令 `lspci -s <带域标识符的 VF 地址> -vvv | grep Ethernet`，获取当前网卡名称，如下所示。

```
lspci -s 0000:05:00.1 -vvv | grep Ethernet
05:00.1 Ethernet controller: Intel Corporation Ethernet 10G 2P X52
0 Adapter (rev 01)
```

3. 在 `cpaas-system` 命名空间下，找到名为 `supported-nic-ids` 的 ConfigMap 配置文件，检查其 `data` 部分是否包含当前网卡的配置信息。

注意：若当前网卡不在支持列表中，需要参考[步骤 4](#)将网卡添加到配置文件中。若已在支持列表中，则跳过[步骤 4](#)。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: supported-nic-ids
  namespace: cpaas-system
data:
  Broadcom_bnxt_BCM57414_2x25G: 14e4 16d7 16dc
  Broadcom_bnxt_BCM75508_2x100G: 14e4 1750 1806
  Intel_i40e_10G_X710_SFP: 8086 1572 154c
  Intel_i40e_25G_SFP28: 8086 158b 154c
  Intel_i40e_40G_XL710_QSFP: 8086 1583 154c
  Intel_i40e_X710_X557_AT_10G: 8086 1589 154c
  Intel_i40e_XXV710: 8086 158a 154c
  Intel_i40e_XXV710_N3000: 8086 0d58 154c
  Intel_ice_Columbiaville_E810: 8086 1591 1889
  Intel_ice_Columbiaville_E810-CQDA2_2CQDA2: 8086 1592 1889
  Intel_ice_Columbiaville_E810-XXVDA2: 8086 159b 1889
  Intel_ice_Columbiaville_E810-XXVDA4: 8086 1593 1889
```

4. 以 `<NIC 名称>: <厂商 ID> <设备 ID> <VF 设备 ID>` 格式，将当前网卡添加到支持列表的 data 部分，如下所示。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: supported-nic-ids
  namespace: cpaas-system
data:
  Broadcom_bnxt_BCM57414_2x25G: 14e4 16d7 16dc
  Broadcom_bnxt_BCM75508_2x100G: 14e4 1750 1806

  Intel_Corporation_X520: 8086 154d 10ed          ## 新增网卡信息

  Intel_i40e_10G_X710_SFP: 8086 1572 154c
  Intel_i40e_25G_SFP28: 8086 158b 154c
  Intel_i40e_40G_XL710_QSFP: 8086 1583 154c
  Intel_i40e_X710_X557_AT_10G: 8086 1589 154c
  Intel_i40e_XXV710: 8086 158a 154c
  Intel_i40e_XXV710_N3000: 8086 0d58 154c
  Intel_ice_Columbiaville_E810: 8086 1591 1889
  Intel_ice_Columbiaville_E810-CQDA2_2CQDA2: 8086 1592 1889
  Intel_ice_Columbiaville_E810-XXVDA2: 8086 159b 1889
  Intel_ice_Columbiaville_E810-XXVDA4: 8086 1593 1889
```

参数配置说明：

- **Intel_Corporation_X520**：网卡名称，可自定义。
- **8086**：厂商 ID。
- **154d**：设备 ID。
- **10ed**：VF 设备 ID，可在[绑定结果](#)中查看。

12 配置 IP 地址

登录交换机配置 DHCP（动态主机配置协议）。

注意：若无法使用 DHCP，请在虚拟机中手动配置 IP 地址。

结果验证

1. 进入 **Container Platform**。
2. 在左侧导航栏点击 **Virtualization > Virtual Machines**。
3. 点击 **Create Virtual Machine**，添加辅助网卡时，选择 **SR-IOV** 作为 **Network Type**。
4. 完成虚拟机创建。
5. 通过 VNC 访问虚拟机，可看到 eth1 成功获取 IP 地址，表示配置成功。

```

[root@sriov-demo ~]#
[root@sriov-demo ~]# dhclient eth1
[root@sriov-demo ~]# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:00:00:0c:8f:c0 brd ff:ff:ff:ff:ff:ff
    inet 10.33.0.44/16 brd 10.33.255.255 scope global dynamic eth0
        valid_lft 86313367sec preferred_lft 86313367sec
    inet6 fe80::200:ff:fe0c:8fc0/64 scope link
        valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    link/ether 06:1e:b5:e1:5f:f7 brd ff:ff:ff:ff:ff:ff
    inet 192.168.39.7/24 brd 192.168.39.255 scope global dynamic eth1
        valid_lft 86398sec preferred_lft 86398sec
    inet6 2002::41e:b5ff:fee1:5ff7/64 scope global mngtmpaddr dynamic
        valid_lft 2591997sec preferred_lft 604797sec
    inet6 fe80::41e:b5ff:fee1:5ff7/64 scope link
        valid_lft forever preferred_lft forever
[root@sriov-demo ~]#
  
```

相关说明

CentOS 虚拟机内核参数配置

CentOS 虚拟机使用 SR-IOV 网卡后，需要修改对应网卡的内核参数，具体步骤如下。

1. 打开终端，执行以下命令修改对应网卡的内核参数。将命令中的 `<NIC Name>` 替换为实际网卡名称。

```

sysctl -w net.ipv4.conf.<NIC Name>.rp_filter=2
echo "net.ipv4.conf.<NIC Name>.rp_filter=2" >> /etc/sysctl.conf
  
```

2. 执行以下命令加载并应用 `/etc/sysctl.conf` 文件中的所有内核参数命令，使内核配置生效。输出信息中值为 2 表示修改成功。

```
sysctl -p
```

输出信息：

```
net.ipv4.conf.<NIC Name>.rp_filter = 2
```

配置虚拟机使用网络绑定模式以支持 IPv6

网络绑定模式是虚拟机网络的插件扩展机制。平台默认使用名为 ManagedTap 的插件来实现虚拟机的 IPv6 支持。该插件允许虚拟机通过 CNI 的 DHCP Server 获取 IP 地址。因此，只要 CNI 的 DHCP Server 支持 IPv6，虚拟机也将具备 IPv6 功能。

目前，我们使用 Kube-OVN 作为 CNI。由于 Kube-OVN 的 DHCP Server 完全支持 IPv6，虚拟机可以通过 ManagedTap 和 Kube-OVN 的结合，实现完善的 IPv6 功能。

目录

前提条件

操作步骤

在虚拟机子网中添加 IPv6 配置

在 Web 控制台中创建使用网络绑定模式的虚拟机

通过 VNC 访问虚拟机并配置网络接口

配置 IPv6 默认路由

前提条件

- ACP 版本必须为 v4.0.0 或更高。
- 使用 Kube-OVN 作为 CNI，且虚拟机子网配置为 Underlay。

操作步骤

1 在虚拟机子网中添加 IPv6 配置

```
kubectl edit subnet <subnet-name>
```

在 `spec` 下添加以下参数：

```
spec:
  enableDHCP: true
  enableIPv6RA: true
  u2oInterconnection: true
```

2 在 Web 控制台中创建使用网络绑定模式的虚拟机

创建虚拟机时，选择 **Network Binding** 作为网络模式。

3 通过 VNC 访问虚拟机并配置网络接口

对于 CentOS 系统，编辑 `/etc/sysconfig/network-scripts/ifcfg-enp1s0` 文件，添加以下配置：

```
IPV6INIT=yes
DHCPV6C=yes
IPV6_AUTOCONF=yes
```

重启网络

```
systemctl restart network
```

4 配置 IPv6 默认路由

如果交换机配置了发送 Router Advertisement (RA) 消息，则无需手动配置路由。默认路由可以通过交换机发送的 RA 消息自动学习。

```
ip r r default via <subnet-v6-gateway>
```

配置多网卡虚拟机

结合 Kube-OVN 和 Multus，为虚拟机提供多网卡支持

目录

前提条件

操作步骤

创建二级网络

创建多网卡虚拟机

配置新网卡的网络

热插拔网络接口

前提条件

- Alauda Container Platform 版本必须为 v4.1.0 或更高。
- 使用 Kube-OVN 作为 CNI。
- 已安装 Alauda Container Platform Networking for Multus。

操作步骤

1 创建二级网络

1. 创建 NetworkAttachmentDefinition

在集群控制节点执行以下命令：

Command

```
cat << EOF | kubectl create -f -
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: <name>
  namespace: <namespace>
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "<provider>"
  }'
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: 'k8s.cni.cncf.io/v1'
kind: NetworkAttachmentDefinition
metadata:
  name: attachnet
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.0",
    "type": "kube-ovn",
    "server_socket": "/run/openvswitch/kube-ovn-daemon.sock",
    "provider": "attachnet.default.ovn"
  }'
EOF
```

参数说明：

- name：NetworkAttachmentDefinition 的名称。
- namespace：NetworkAttachmentDefinition 所在的命名空间，必须与虚拟机使用相同的命名空间。
- provider：当前 NetworkAttachmentDefinition 的 `<name>.<namespace>.ovn`，Kube-OVN 会根据此信息查找对应的 Subnet 资源。注意后缀必须设置为 ovn。

2. 创建 Kube-OVN 子网

如果将 Kube-OVN 用作二级网卡，provider 应设置为对应

NetworkAttachmentDefinition 的 `<name>.<namespace>.ovn`，且必须以 ovn 结尾。

在集群控制节点执行以下命令：

Command

```
cat << EOF | kubectl create -f -
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: <name>
spec:
  protocol: IPv4
  enableDHCP: true
  provider: <provider>
  cidrBlock: <cidrBlock>
  gateway: <gateway>
  excludeIps:
    - <excludeIps>
EOF
```

Example

```
cat << EOF | kubectl create -f -
apiVersion: kubeovn.io/v1
kind: Subnet
metadata:
  name: attachnet
spec:
  protocol: IPv4
  enableDHCP: true
  provider: attachnet.default.ovn
  cidrBlock: 172.17.0.0/16
  gateway: 172.17.0.1
  excludeIps:
    - 172.17.0.0..172.17.0.10
EOF
```

参数说明：

- name：子网名称。
- provider：NetworkAttachmentDefinition 的 provider。
- cidrBlock：子网 CIDR。
- gateway：网关地址。
- excludeIps：保留的 IP 集合，不会被自动分配。例如，可用作计算组件的固定 IP 地址。

2 创建多网卡虚拟机

1. 通过 UI 创建虚拟机

2. 切换到 **YAML** 视图，为虚拟机添加另一张网卡

在 `spec.template.spec.domain.devices.interfaces` 下添加新接口

在 `spec.template.spec.networks` 下添加新网络

```
spec:
  template:
    spec:
      domain:
        devices:
          interfaces:
            - bridge: {}
              name: default
            # 新接口
            - bridge: {}
              name: dyniface1
          networks:
            - name: default
              pod: {}
            # 新网络
            - multus:
                networkName: <networkName>
                name: dyniface1
```

其中 networkName 是 NetworkAttachmentDefinition 的名称。

3 配置新网卡的网络

虚拟机启动后，需要进入虚拟机手动配置新添加的网卡网络。

4 热插拔网络接口

支持在运行中的虚拟机中热插拔网络接口。

热插拔支持使用 virtio 模型且通过桥接绑定或 SR-IOV 绑定连接的接口。

热插拔仅支持通过桥接绑定连接的接口。

1. 向运行中的虚拟机添加接口

使用 `kubecttl edit` 修改虚拟机的 YAML 配置

```
spec:
  template:
    spec:
      domain:
        devices:
          interfaces:
            - bridge: {}
              name: default
            # 新接口
            - bridge: {}
              name: dyniface1
          networks:
            - name: default
              pod: {}
            # 新网络
            - multus:
                networkName: <networkName>
                name: dyniface1
```

2. 从运行中的虚拟机移除接口

使用 `kubectrl edit` 修改虚拟机的 YAML 配置

```
spec:
  template:
    spec:
      domain:
        devices:
          interfaces:
            - bridge: {}
              name: default
            # 将接口状态设置为 absent
            - bridge: {}
              name: dyniface1
              state: absent
          networks:
            - name: default
              pod: {}
            # 新网络
            - multus:
                networkName: <networkName>
                name: dyniface1
```


存储

介绍

介绍

优势

操作指南

管理虚拟磁盘

创建虚拟磁盘

挂载虚拟磁盘

扩展虚拟磁盘

卸载虚拟磁盘

删除虚拟磁盘

StorageProfile 和跨命名空间克隆范围

介绍

ACP Virtualization 结合 KubeVirt Storage 通过无缝集成 Kubernetes 原生存储机制，为虚拟机（VM）提供持久存储能力。它利用 **PersistentVolumeClaim** (PVC) 来存储虚拟机磁盘数据，并通过 **Container Storage Interface** (CSI) 实现与多种存储系统的集成。此外，平台还采用 **Containerized Data Importer** (CDI) 来初始化虚拟机磁盘数据。在此基础上，平台扩展了虚拟机磁盘管理的高级功能，实现了全面的生命周期控制。

目录

| [优势](#)

优势

- 用户友好的操作
大多数虚拟机磁盘操作可以通过 **Web UI** 轻松完成，减少对 CLI 专业知识的依赖。
- 虚拟机磁盘生命周期管理
配置虚拟机磁盘在关联虚拟机终止时是否自动删除。

操作指南

管理虚拟磁盘

[创建虚拟磁盘](#)

[挂载虚拟磁盘](#)

[扩展虚拟磁盘](#)

[卸载虚拟磁盘](#)

[删除虚拟磁盘](#)

[StorageProfile 和跨命名空间克隆范围](#)

管理虚拟磁盘

数据盘可用于满足业务的数据持久化需求。

数据盘由 CDI `DataVolume` 支持（它会创建同名的 PVC）。下面的操作分别对应于：创建一个 `DataVolume`、将其热插拔到虚拟机（`addvolume`）、扩展其 PVC、将其分离（`removevolume`），以及删除该 `DataVolume`。

目录

创建虚拟磁盘

操作步骤

使用 API

挂载虚拟磁盘

操作步骤

使用 API

扩展虚拟磁盘

操作步骤

使用 API

卸载虚拟磁盘

操作步骤

使用 API

删除虚拟磁盘

操作步骤

使用 API

创建虚拟磁盘

为虚拟机创建一个数据盘。一次只能添加一个虚拟磁盘；如果需要多个磁盘，请重复此操作。

注意：当虚拟机处于运行中状态时，可以在线挂载虚拟磁盘。

操作步骤

- 1. 访问容器平台。
- 2. 在左侧导航栏中，单击虚拟化 > 虚拟磁盘。
- 3. 单击创建虚拟磁盘。
- 4. 根据以下说明配置信息。

参数	说明
卷模式	- 文件系统：以挂载文件目录的方式挂载磁盘。 - 块设备：将磁盘作为块设备挂载。
Storage Class	平台通过自动创建和管理 PersistentVolumeClaim 来维护虚拟机磁盘。您需要指定用于动态创建 PersistentVolumeClaim 的 Storage Class。 不同的 Storage Class 支持不同的卷模式。如果所选卷模式没有可用的 Storage Class，请联系管理员进行添加。
删除随 VM	启用后，在删除虚拟机时也会删除磁盘数据。
挂载	- 不挂载：仅创建虚拟磁盘；需要时可在之后再挂载。 - 挂载到 VM：选择需要将虚拟磁盘挂载到的目标虚拟机。

- 5. 单击创建。

使用 API

创建一个空白数据盘作为 `DataVolume`：

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: datadisk-1
  namespace: demo
spec:
  source:
    blank: {}
  storage:
    resources:
      requests:
        storage: 10Gi
    storageClassName: rbd-1
    volumeMode: Block
    accessModes:
      - ReadWriteMany
```

```
kubectl apply -f datadisk.yaml
# wait for the DataVolume to reach Succeeded and its PVC to be Bound
kubectl get datavolume datadisk-1 -n demo -w
```

挂载虚拟磁盘

将数据盘挂载到虚拟机，把已创建的虚拟磁盘附加到目标虚拟机。

注意：当虚拟机处于运行中状态时，可以在线挂载虚拟磁盘。

操作步骤

1. 访问容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟磁盘。
3. 在要挂载的虚拟磁盘旁，单击：> 挂载。

4. 选择目标虚拟机，然后单击挂载。

使用 API

将磁盘热插拔到运行中的虚拟机中。使用 `virtctl` 时，请加上 `--persist`，这样磁盘会被添加到虚拟机规格中，并且在重启后仍然存在：

```
virtctl addvolume web-01 --volume-name=datadisk-1 --persist -n demo
```

这会发起 `addvolume` 子资源请求（`AddVolumeOptions`）。随后磁盘会出现在 `VirtualMachineInstance.status.volumeStatus` 中，并从 `AttachedToNode` 逐步变为 `Ready`。如果不使用 `--persist`，热插拔只会作用于当前运行中的实例，不会在重启后保留。

扩展虚拟磁盘

扩展已经挂载到虚拟机上的系统盘和数据盘。

操作步骤

1. 访问容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟机。
3. 单击虚拟机名称，进入详情页面。
4. 在虚拟磁盘区域中，找到要扩展的磁盘，然后单击：> 扩展。
5. 输入新的容量，然后单击扩展。

使用 API

修改 PVC 请求的大小（StorageClass 必须允许卷扩展）：

```
kubectl patch pvc datadisk-1 -n demo --type merge \
  -p '{"spec":{"resources":{"requests":{"storage":"20Gi"}}}}'
```

`status.capacity.storage` 会增长到新大小；Ceph RBD 等 CSI 驱动支持在线扩展（无需重启）。

卸载虚拟磁盘

将数据盘从虚拟机中卸载；只有处于已停止状态的虚拟机才能卸载磁盘。

操作步骤

1. 访问容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟磁盘。
3. 在要卸载的虚拟磁盘旁，单击 ⋮ > 卸载，然后确认。

使用 API

分离热插拔磁盘：

```
virtctl removevolume web-01 --volume-name=datadisk-1 -n demo
```

这会发起 `removevolume` 子资源请求，并将磁盘分离（对应条目会从 `VirtualMachineInstance.status.volumeStatus` 中移除）。KubeVirt API 支持从运行中的虚拟机分离磁盘，但在移除之前，请确保来宾系统中该磁盘已不再使用（在操作系统层面已卸载），以避免数据丢失。

删除虚拟磁盘

仅当虚拟磁盘处于未挂载状态时，才支持删除。

注意：系统盘不能删除。

操作步骤

1. 访问容器平台。

2. 在左侧导航栏中，单击虚拟化 > 虚拟磁盘。
3. 在要删除的虚拟磁盘旁，单击： > 删除，然后确认。

使用 API

```
kubectl delete datavolume datadisk-1 -n demo
```

底层 PVC 归 `DataVolume` 所有，并会随之被垃圾回收。如果 CDI 已经对该 `DataVolume` 执行了垃圾回收，则直接删除 PVC (`kubectl delete pvc datadisk-1 -n demo`) 。

StorageProfile 和跨命名空间克隆范围

CDI 会为每个 StorageClass 发布一个 `StorageProfile`，其中描述了在该后端上如何进行磁盘的动态供给和克隆。查看它可以了解支持的访问模式/卷模式以及克隆方法：

```
kubectl get storageprofile rbd-1 -o jsonpath='{.status.cloneStrategy} claimPropertySets={.status.claimPropertySets}{"\n"}'
# cloneStrategy=csi-clone claimPropertySets=[{RWX,Block},{RWO,Block},{RWO,Filesystem}]
```

`cloneStrategy` (`csi-clone`、`snapshot` 或 host-assisted `copy`) 决定可启动卷或 VM 克隆如何复制磁盘；`snapshot/clone` 是否可用，取决于该 StorageClass 是否具备这些能力。

跨命名空间克隆范围：在创建虚拟机或可启动卷时，控制台只会提供来自当前命名空间和 `kube-public` 的启动镜像作为克隆源。这个 UI 级别的限制——而不是 RBAC——是该平台支持的边界：请将共享的黄金镜像发布到 `kube-public`，内置的 `os-images.kubevirt.io:view` 角色会使每个已认证用户都能对其进行克隆。

CDI 上游的 `datavolumes/source` 权限 (`datavolume-cloner` ClusterRole) 不能可靠地限制 CSI 路径的克隆，因此将其绑定并不是一种受支持且足够的方式，无法用于从任意非公开命名空间暴露镜像。对于共享镜像，请改用 `kube-public`。

备份与恢复

介绍

介绍

应用场景

使用限制

操作指南

使用快照

前提条件

说明

创建快照

回滚快照

删除快照

使用 Velero

前提条件

操作步骤

介绍

ACP Virtualization With Kubevirt 提供虚拟机快照能力，允许用户通过快照备份和恢复虚拟机。

目录

| [应用场景](#)

[使用限制](#)

应用场景

- 灾难恢复与故障回滚
当虚拟机因硬件故障、人工误操作（例如误删文件）或恶意攻击（例如勒索软件）而发生数据丢失时，快照可作为恢复业务的最后一道防线。

使用限制

- 虚拟机运行时（在线）或停止状态下都可以创建快照。对于应用一致性快照，需要在来宾系统中安装 QEMU guest agent；否则，在线快照将是崩溃一致性快照。要回滚快照，必须先停止虚拟机。

- 虚拟机磁盘必须使用支持卷快照的 storage class（已绑定 `VolumeSnapshotClass`）。

操作指南

使用快照

前提条件

说明

创建快照

回滚快照

删除快照

使用 **Velero**

前提条件

操作步骤

使用快照

虚拟机快照会保存虚拟机的当前状态，并可在发生意外故障时用于将虚拟机恢复到该状态。

目录

前提条件

说明

快照一致性

创建快照

操作步骤

使用 API

回滚快照

说明

操作步骤

使用 API

删除快照

说明

操作步骤

使用 API

前提条件

- 管理员已在平台管理中部署了 **Volume Snapshot**。
- 虚拟机快照基于卷快照。请确保至少有一个磁盘绑定到支持卷快照的存储类，例如 CephFS 内置存储。
- 快照可以在虚拟机运行中（在线）或已停止（离线）时创建。对于应用一致性快照，请在访客系统中安装 QEMU guest agent，以便冻结文件系统；如果没有安装，则在线快照为崩溃一致性（快照的 `status.indications` 会报告 `Online` 和 `NoGuestAgent`）。回滚快照要求虚拟机首先必须处于已停止状态——请参见[停止虚拟机](#)。

说明

如果集群中存在多种相同类型的存储，例如挂载了多个不同来源的 Ceph RBD 存储，那么当虚拟机使用此类存储时，磁盘快照功能可能无法正常工作。

快照一致性

检查快照的 `status.indications` 以了解其可信程度：

- `GuestAgent` — QEMU guest agent 在快照前冻结了访客系统文件系统，因此它是应用一致性的（这是最安全的恢复方式）。
- `NoGuestAgent` — 没有可用的 guest agent，因此在线快照是崩溃一致性的（等同于断电重启）。请安装 [guest agent](#) 以获得应用一致性快照。
- `QuiesceFailed` — 已尝试冻结，但失败了；请将该快照视为崩溃一致性。
- `Online` — 快照是在虚拟机运行时创建的。

虚拟 TPM (vTPM) 的持久状态不会被快照捕获，因此依赖 vTPM 的虚拟机（例如用于 BitLocker）在恢复后可能需要手动重新配置。

创建快照

虚拟机快照包含的内容：虚拟机设置以及支持卷快照的磁盘状态。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 虚拟化 > 虚拟机。
3. 找到目标虚拟机并点击 **创建快照**。
4. 填写快照描述。描述可以帮助你记录虚拟机的当前状态，例如 `Initial Installation`、`Before Application Upgrade`。
5. 点击 **创建**。快照所需时间取决于网络条件和工作负载，请耐心等待。
6. 检查快照状态。
 - 当快照状态变为 `Ready` 时，表示创建成功。
 - 如果快照长时间保持 `Not Ready` 状态，请点击  > 查看原因并排查问题，然后重新创建快照。

使用 API

创建一个引用该虚拟机的 `VirtualMachineSnapshot`：

```
apiVersion: snapshot.kubevirt.io/v1beta1
kind: VirtualMachineSnapshot
metadata:
  name: web-01-snap-1
  namespace: demo
spec:
  source:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: web-01
```

```
kubectl apply -f snapshot.yaml
kubectl get virtualmachinesnapshot web-01-snap-1 -n demo \
  -o jsonpath='{.status.phase} readyToUse={.status.readyToUse} {.status.indications}{"\n"}'
```

当 `status.readyToUse` 为 `true` 时，表示快照已完成。对于正在运行的虚拟机快照，`status.indications` 会包含 `Online`（如果未安装 guest agent，还会包含 `NoGuestAgent`，此时快照为崩溃一致性）。

回滚快照

将虚拟机设置以及支持卷快照的磁盘回滚到创建快照时的状态。例如，快照创建后新增的磁盘将被移除；已修改的磁盘数据将被恢复。

说明

如果存在绑定到支持 LVM 机制的存储类的磁盘（例如 TopoLVM），请先与管理员确认该存储类的回收策略设置为 **Retain**（`reclaimPolicy: Retain`），以便正确使用快照回滚功能。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 虚拟化 > 虚拟机。
3. 点击 *虚拟机名称*。
4. 在 **Snapshots** 选项卡中，找到目标快照并点击：> 回滚。
5. 阅读界面上的提示信息，确认无误后点击 回滚。
注意：回滚操作无法中止或撤销，请谨慎操作。
6. 点击快照名称，在“快照回滚记录”中查看是否已完成回滚。回滚所需时间取决于网络条件和工作负载，请耐心等待。

说明

- 如果回滚失败，虚拟机状态将保持不变。你可以正常启动虚拟机，或者再次尝试回滚该快照。
- 如果虚拟机在回滚过程中启动，它将恢复到停止前的状态；当再次停止虚拟机时，它会继续回滚到快照创建时的状态。
- 为避免操作冲突，请确保最近一次回滚记录已完成，再对该虚拟机执行其他操作。

使用 API

通过创建一个以虚拟机为目标并引用该快照的 `VirtualMachineRestore` 来执行回滚。请先停止目标虚拟机——恢复要求虚拟机处于停止状态：

```
apiVersion: snapshot.kubevirt.io/v1beta1
kind: VirtualMachineRestore
metadata:
  name: web-01-restore-1
  namespace: demo
spec:
  target:
    apiGroup: kubevirt.io
    kind: VirtualMachine
    name: web-01
  virtualMachineSnapshotName: web-01-snap-1
```

```
kubectl apply -f restore.yaml
kubectl get virtualmachinerestore web-01-restore-1 -n demo \
  -o jsonpath='{complete={.status.complete}}{"\n"}'
```

当 `status.complete` 为 `true` 时，回滚完成。

删除快照

删除不再需要的虚拟机快照以释放磁盘资源。

说明

在删除已回滚的虚拟机快照时，如果虚拟机磁盘需要基于快照复制数据（例如 TopoLVM），你必须等到基于回滚版本的虚拟机已启动后再删除，否则虚拟机将无法启动。

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **虚拟化 > 虚拟机**。
3. 点击 **虚拟机名称**。
4. 在 **Snapshots** 选项卡中，找到目标快照并点击：**> 删除**。
5. 阅读提示信息并确认无误后，点击 **删除**。

使用 **API**

```
kubectl delete virtualmachinesnapshot web-01-snap-1 -n demo
```

使用 Velero

Velero 是 VMware 提供的开源 Kubernetes 集群备份和迁移工具。KubeVirt 提供了 Velero 插件以支持虚拟机的备份与恢复。

目录

前提条件

操作步骤

准备工作

备份

恢复

跨集群恢复

恢复到不同命名空间

恢复到不同存储类

前提条件

- **Kubernetes** 版本：1.20 或更高（Velero 要求）
- **S3 存储**：Velero BackupStorageLocation 使用平台提供的 Ceph 或 MinIO 对象存储
- **块存储**：平台提供的 Ceph RBD

操作步骤

准备工作

1. 通过 平台管理 → 集群管理 → 备份与恢复 → 备份仓库 在 ACP 平台部署 Velero。使用对象存储信息创建备份仓库。
2. 启用 CSI 功能并添加 KubeVirt 插件：

```
kubectl edit deploy -n cpaas-system velero
```

在 Velero 部署中添加以下内容：

```
containers:
- args:
  - server
  - --uploader-type=restic
  - --namespace=cpaas-system
  - --features=EnableCSI
command:
- /velero
```

在 initContainers 部分添加：

```
initContainers:
- image: registry.example.org/3rdparty/kubevirt/kubevirt-velero-plugin:
v0.7.0
  imagePullPolicy: IfNotPresent
  name: velero-plugin-kubevirt
  resources: {}
  terminationMessagePath: /dev/termination-log
  terminationMessagePolicy: File
  volumeMounts:
  - mountPath: /target
    name: plugins
```

3. 验证 KubeVirt 插件安装：

```
POD=$(kubectl -n cpaas-system get pod -l app.kubernetes.io/name=velero
-o jsonpath='{.items[0].metadata.name}')
kubectl -n cpaas-system exec -ti "$POD" -- /velero get plugins | grep k
ubevirt
```

示例输出：

```
kubevirt-velero-plugin/backup-datavolume-action      BackupItemAction
kubevirt-velero-plugin/backup-datavolume-action      BackupItemAction
kubevirt-velero-plugin/backup-virtualmachine-action  BackupItemAction
kubevirt-velero-plugin/backup-virtualmachine-action  BackupItemAction
kubevirt-velero-plugin/backup-virtualmachineinstance-action BackupItemAction
kubevirt-velero-plugin/backup-virtualmachineinstance-action BackupItemAction
kubevirt-velero-plugin/restore-pod-action            RestoreItemAction
kubevirt-velero-plugin/restore-pod-action            RestoreItemAction
kubevirt-velero-plugin/restore-pvc-action            RestoreItemAction
kubevirt-velero-plugin/restore-pvc-action            RestoreItemAction
kubevirt-velero-plugin/restore-vm-action            RestoreItemAction
kubevirt-velero-plugin/restore-vm-action            RestoreItemAction
kubevirt-velero-plugin/restore-vmi-action            RestoreItemAction
kubevirt-velero-plugin/restore-vmi-action            RestoreItemAction
```

4. 调整 node-agent 挂载宿主机 kubelet 目录并开启特权模式（数据通过 `/var/lib/kubelet` 迁移时必需）：

```
kubectl edit ds -n cpaas-system node-agent
```

更新为：

```
securityContext:
  privileged: true
volumeMounts:
- mountPath: /host_pods
  mountPropagation: HostToContainer
  name: host-pods
- mountPath: /var/lib/kubelet/plugins
  mountPropagation: HostToContainer
  name: host-plugins
- mountPath: /scratch
  name: scratch
securityContext:
  runAsUser: 0
volumes:
- hostPath:
    path: /var/lib/kubelet/pods
    type: ""
    name: host-pods
- hostPath:
    path: /var/lib/kubelet/plugins
    type: ""
    name: host-plugins
- emptyDir: {}
  name: scratch
```

备份

1. 按需创建带有多个磁盘的虚拟机。
2. 创建备份资源：

```

apiVersion: velero.io/v1
kind: Backup
metadata:
  annotations:
    velero.io/resource-timeout: 10m0s
    velero.io/source-cluster-k8s-gitversion: v1.30.4
    velero.io/source-cluster-k8s-major-version: "1"
    velero.io/source-cluster-k8s-minor-version: "30"
  labels:
    velero.io/storage-location: default
name: example-backup
namespace: cpaas-system
spec:
  csiSnapshotTimeout: 10m0s
  defaultVolumesToFsBackup: false
  hooks: {}
  includedNamespaces:
    - example-namespace
  itemOperationTimeout: 4h0m0s
  metadata: {}
  snapshotMoveData: true
  storageLocation: default
  ttl: 720h0m0s

```

3. 等待备份完成并验证：

```

POD=$(kubectl -n cpaas-system get pod -l app.kubernetes.io/name=velero
-o jsonpath='{.items[0].metadata.name}')
kubectl -n cpaas-system exec -ti "$POD" -- /velero backup describe exam
ple-backup --details
kubectl -n cpaas-system exec -ti "$POD" -- /velero backup get example-b
ackup

```

示例输出：

NAME	STATUS	EXPIRES	STORAGE LOCATION	ERRORS	WARNINGS	CREATED
example-backup	WaitingForPluginOperations	0	default	0	0	2025-01-25 14:14:08 +0000 UTC
		29d			<none>	

4. 查看数据迁移状态：

```
kubectl get dataupload -n cpaas-system
```

示例输出：

NAME	STATUS	STARTED	BYTES DONE	TOTAL BYTES
STORAGE LOCATION AGE NODE				
example-backup-fhc6b	Completed	11h	21474836480	21474836480
default 11h 192.168.254.66				
example-backup-qwwzh	Completed	11h	21474836480	21474836480
default 11h 192.168.254.15				

5. 验证备份完成：

```
POD=$(kubectl -n cpaas-system get pod -l app.kubernetes.io/name=velero
-o jsonpath='{.items[0].metadata.name}')
kubectl -n cpaas-system exec -ti "$POD" -- /velero backup get example-b
ackup
```

示例输出：

NAME	STATUS	ERRORS	WARNINGS	CREATED
EXPIRES STORAGE LOCATION SELECTOR				
example-backup 000 UTC 29d	Completed	0	0	2025-01-25 14:14:08 +0
	default		<none>	

6. 检查备份仓库（如 MinIO）中的目录：

```
mc ls <backup-repository>
```

示例输出：

```
[2025-01-26 09:23:08 CST] 0B backups/
[2025-01-26 09:23:08 CST] 0B kopia/
[2025-01-26 09:23:08 CST] 0B restic/
```

恢复

1. 删除原虚拟机。
2. 通过 平台管理 → 集群管理 → 备份与恢复 → 恢复管理 创建恢复资源，或手动创建：

```
apiVersion: velero.io/v1
kind: Restore
metadata:
  annotations:
    cpaas.io/description: ""
  finalizers:
    - restores.velero.io/external-resources-finalizer
  name: example-restore
  namespace: cpaas-system
spec:
  backupName: example-backup
  excludedResources:
    - nodes
    - events
    - events.events.k8s.io
    - backups.velero.io
    - restores.velero.io
    - resticrepositories.velero.io
    - csinodes.storage.k8s.io
    - volumeattachments.storage.k8s.io
    - backuprepositories.velero.io
  hooks: {}
  includedNamespaces:
    - example-namespace
  itemOperationTimeout: 4h0m0s
  namespaceMapping: {}
```

3. 验证恢复：

```
kubectl exec -ti -n cpaas-system velero-5df7bb7598-ljjrn -- /velero restore get
```

查看 datadownload 状态：

```
kubectl get datadownload -n cpaas-system
```

示例输出：

NAME	STATUS	STARTED	BYTES DONE	TOTAL BYTES	STORAGE LO
CATION	AGE	NODE			
example-restore-7lfc	Completed	11m		21474836480	21474836480
default		15m	192.168.254.66		
example-restore-cjcd	Completed	15m		21474836480	21474836480
default		15m	192.168.254.66		

恢复状态：

NAME	BACKUP	STATUS	STARTED	COM
PLETED		ERRORS	WARNINGS	CREATED
SELECTOR				
aa	example-backup	Completed	2025-01-26 01:53:14 +0000 UTC	202
5-01-26 02:01:24 +0000 UTC	0	2	2025-01-26 01:53:14 +0	
000 UTC	<none>			

4. 确认虚拟机已恢复且正常运行。

跨集群恢复

1. 在新集群部署 Velero 并完成准备工作。
2. 配置与原集群相同的备份仓库（相同桶和目录），备份资源将自动出现。
3. 按上述恢复步骤操作。

恢复到不同命名空间

在恢复 spec 中添加 `namespaceMapping` 字段：

```
spec:
  namespaceMapping:
    example-namespace: test-namespace
```

恢复到不同存储类

1. 在恢复前，在 `cpaas-system` 命名空间创建 ConfigMap：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: change-storage-class-config
  namespace: cpaas-system
  labels:
    velero.io/plugin-config: ""
    velero.io/change-storage-class: RestoreItemAction
data:
  vm-cephrbd: vm-topolvm-a
```

2. 确保存储能力（如跨节点访问、RWX）一致。
3. 注意：当前不支持 RWX 模式的修改。