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

概念和术语表

概念和术语表

工作原理

计算对象

镜像和磁盘

网络

迁移和可用性

备份和恢复

从哪里开始

安装

安装

前提条件

规划与要求

操作步骤

资源配额说明

镜像

介绍

优势

镜像源和镜像实例

从集群外部上传镜像

操作指南

权限

实用指南

虚拟机

介绍

可启动卷

注意事项

操作指南

实用指南

故障排除

默认实例类型和首选项

从可启动卷创建虚拟机

更新或删除可启动卷

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。

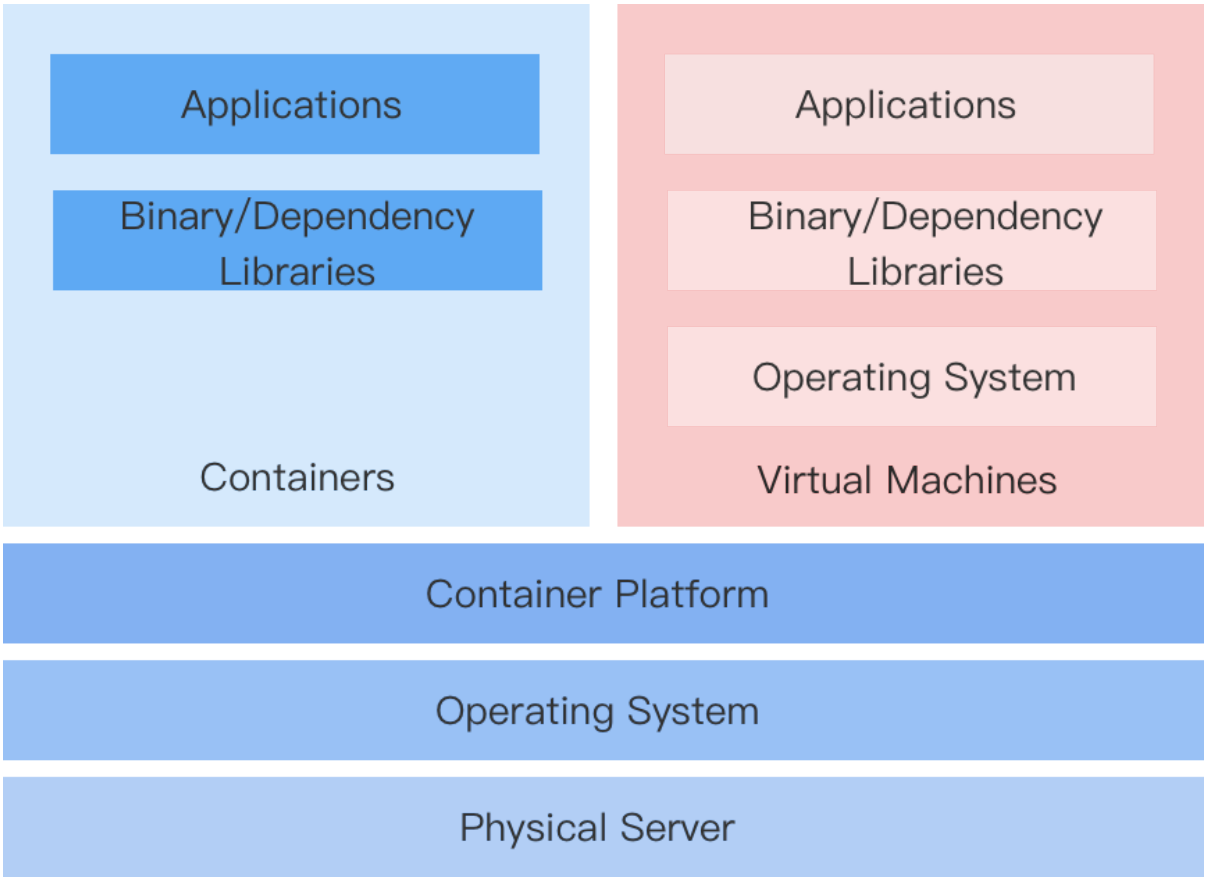
因此，一个能够在同一平台上统一配置、管理和控制容器资源与虚拟机资源的解决方案就显得尤为重要。

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	`VirtualMachine` (`kubevirt.io/v1`)
虚拟磁盘	`DataVolume` / PVC (`cdi.kubevirt.io/v1beta1`)
虚拟机快照	`VirtualMachineSnapshot` (`snapshot.kubevirt.io/v1beta1`)
可启动卷	`DataSource` (`cdi.kubevirt.io/v1beta1`)
克隆	`VirtualMachineClone` (`clone.kubevirt.io/v1beta1`)
Live Migration	`VirtualMachineInstanceMigration` (`kubevirt.io/v1`)
存储迁移	`VirtualMachineStorageMigrationPlan` + `VirtualMachineStorageMigration` (`migrations.kubevirt.io/v1alpha1`)

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

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
`kubevirt.io:view`	对虚拟机和实例的只读访问
`kubevirt.io:edit`	创建 / 更新 / 删除虚拟机并打开控制台
`kubevirt.io:admin`	完整的虚拟机管理，包括子资源
`kubevirt.io:migrate`	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。

概念和术语表

此页面面向熟悉 Kubernetes 但刚接触虚拟化的管理员。本手册的其余部分默认使用以下术语；每个术语都在此处定义一次，并提供最接近的 Kubernetes 类比，以便操作指南保持简洁。

目录

工作原理

计算对象

镜像和磁盘

网络

迁移和可用性

备份和恢复

从哪里开始

工作原理

此平台上的虚拟化基于 KubeVirt 和 CDI，并由一个 operator 进行部署和协调：

- **HyperConverged (HCO)** — `HyperConverged` 自定义资源是唯一事实来源。operator 从中安装 KubeVirt 和 CDI，并持续对其进行协调，因此集群范围的设置（例如实时迁移限制）应设置在 `HyperConverged` 上，而不是直接设置在 `KubeVirt` 资源上。类比：一个顶层 operator CR，例如拥有其下所有资源的 `Installation` / `Subscription`。

- **KubeVirt** — 运行虚拟机。其控制器包括 **virt-controller**（集群级别，用于调度 VM）、**virt-handler**（DaemonSet，每个节点一个，类似于用于 VM 的 kubelet）以及 **virt-launcher**（每个正在运行的 VM 对应一个 pod，用于封装 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，正在运行的虚拟机实际上是普通节点上的普通 **Pod**（virt-launcher pod），使用普通的 **PVC** 作为磁盘，并获取普通的 **Pod IP**——这也是现有 Kubernetes 工具（RBAC、NetworkPolicy、调度、监控）仍然适用的原因。

计算对象

术语	定义	**Kubernetes** 类比
VirtualMachine (`kubevirt.io/v1`)	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）。	一个 `Deployment` — 用于创建和编辑的 spec。
VirtualMachineInstance (VMI)	启动 `VirtualMachine` 后生成的*正在运行的*实例。生命周期子资源（控制台、VNC、暂停、迁移）作用于 VMI。	一个 `Pod` — 根据 spec 创建，仅在运行期间存在。
virt-launcher pod	封装一个正在运行的 VM 的 QEMU 进程的 pod。	实际执行工作负载的 Pod。
`spec.running`	`VirtualMachine` 上的布尔型电源开关（`true` 启动它，`false` 停止它）。此平台使用此字段。	`replicas: 1` 与 `0`。
`spec.runStrategy`	`spec.running` 的上游替代方案（`Always` / `RerunOnFailure` / `Manual` / `Halted`），用于实现更精细的重启行为。	重启策略。
instancetype	可复用的 CPU/内存配置文件，可以是集群范围或命名空间范围，VM 引用它而不是自行声明大小——这是为 VM 设定大小的推荐方式。VM 会根据已解析的	sizing 预设 / 类似 `LimitRange` 的模板。

术语	定义	**Kubernetes** 类比
	副本（修订版本）继续运行。请参阅实例类型和首选项。	
preference	可复用的、特定于 OS 的设备/固件默认设置配置文件（总线类型、时钟、功能），可以是集群范围或命名空间范围；它只填充 VM 未设置的配置。	应用于 VM 的设备/固件“配置文件”。
QEMU guest agent	安装在 guest OS *内部*的智能体。启用 IP 报告、正常关机以及用于生成应用一致性快照的文件系统冻结功能。如果没有它，平台无法看到 guest IP，在线快照只能保证崩溃一致性。	工作负载内部用于报告运行状况并响应信号的 sidecar/智能体。请参阅安装 Guest Tools。
vTPM	虚拟 TPM 2.0 设备（`spec.template.spec.domain.devices.tpm`）。Windows 11 和 BitLocker 所必需。	附加到 VM 的虚拟安全设备。

镜像和磁盘

VM 从可复用 镜像 克隆的 磁盘 启动。其链路如下：

```
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	定义	**Kubernetes** 类比
qcow2 / raw	磁盘镜像文件格式。`qcow2` 是稀疏/写时复制格式；`raw` 是平面镜像。这是要导入的*内容*，而不是容器镜像。	磁盘镜像制品，例如 OS 云镜像。

术语	定义	**Kubernetes** 类比
containerDisk	封装在容器镜像内部、像容器镜像一样拉取的 VM 磁盘。注意：普通的应用容器镜像不是 containerDisk，无法启动。	负载为可启动磁盘的 OCI 镜像。
CDI / DataVolume	`DataVolume`（`cdi.kubevirt.io/v1beta1`）声明“将此镜像导入 PVC”；CDI 执行导入并创建后端 PVC。	PVC 加上用于填充其内容的初始化作业。
DataSource / 可启动卷	`DataSource`（`cdi.kubevirt.io/v1beta1`）指向黄金镜像 PVC，也称为 **Bootable Volume**；创建 VM 时，会将其克隆到 VM 的启动磁盘中。	用作模板的只读“黄金”PVC。请参阅可启动卷。
镜像源 **/** 镜像实例	控制台对两个镜像层级的称呼。镜像源（`VirtualMachineImageTemplate`，由所有集群共享）记录镜像的存储位置以及哪些项目可以使用它；镜像实例是导入到集群某个命名空间中的可启动卷——用于共享黄金镜像的 `kube-public`。	镜像引用与预先拉取的镜像副本。请参阅镜像。
volumeMode：**Block** 与 **Filesystem**	PVC 的呈现方式。**Block** 的性能更接近裸金属，是 VM 磁盘的首选方式；Filesystem 也可以使用。	你已经熟悉的 PVC `volumeMode`。
StorageProfile / cloneStrategy	CDI 为每个 StorageClass 发布的对象，描述支持的访问/卷模式以及克隆方式（`csi-clone`、`snapshot` 或由主机辅助的 `copy`）。	StorageClass 的功能元数据。请参阅管理虚拟磁盘。

新集群中没有可启动卷，除非管理员导入一个（或将黄金镜像发布到 `kube-public`）。请从可启动卷开始。

网络

默认情况下，VM 在 virt-launcher **Pod** 中运行，并使用该 Pod 的网络——因此它会获得一个 Pod IP，并受与任何 Pod 相同的 CNI 和 `NetworkPolicy` 约束。创建表单中的网络模式映射到 KubeVirt 绑定：

UI 标签	YAML 绑定	行为
桥接	`bridge: {}`	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；默认情况下不适合实时迁移——通过桥接进行实时迁移需要 `kubevirt.io/allow-pod-bridge-network-live-migration` 注解以及受支持的 CNI（请参阅实时迁移）。
NAT	`masquerade: {}`	VM 获得一个内部私有 IP，并通过 Pod IP 执行 NAT 出站。默认模式；适合实时迁移。

> UI 中的“容器组”指的是承载 VM 的 Kubernetes **Pod**。

如需在物理网络上使用可路由 IP（而非 Pod IP），请使用 Kube-OVN **Underlay** 子网或辅助 NIC。辅助 NIC 通过 `NetworkAttachmentDefinition`（Multus）进行连接。

迁移和可用性

术语	定义	**Kuberne**
实时迁移	在不停机的情况下，通过复制内存将*运行中的* VM 移动到另一个节点。预复制会在 VM 运行期间迭代复制内存；后复制则在 VM 已位于目标节点后完成复制。	在不重启 重新调度 Pod 通常 点。请参
VirtualMachineInstanceMigration	触发一次实时迁移的请求对象；删除该对象会取消正在进行的迁移。	移动 VM Job。
MigrationPolicy	按命名空间/VM 应用的集群范围迁移调优覆盖配置。集群默认值位于 `HyperConverged.spec.liveMigrationConfig`。	通过标签 CR。
evictionStrategy	节点清空时的处理方式：`LiveMigrate`、`LiveMigrateIfPossible` 或 `None`。具有 `LiveMigrate` 的不可迁移 VM（使用 RWO 存储或设备直通）会阻止节点清空——因此也会阻止集群升级。	类似 `PodDis` 的约束， 移，而不 作。请参

术语	定义	**Kuberne**
实时迁移需要共享存储	只有当 VM 的所有磁盘 PVC 都使用 **ReadWriteMany (RWX)** 访问模式时，VM 才能进行实时迁移（例如 Block 模式的 CephRBD 或 CephFS）。如果存储类的 StorageProfile 针对磁盘的卷模式提供了 **ReadWriteMany**，控制台会为磁盘提供该模式。	RWX 与 相同的模
存储迁移	在线将运行中的 VM 的磁盘在不同 StorageClass 之间移动。	在不停机 PVC 迁移 StorageC

备份和恢复

术语	定义	**Kubernetes** 类比
VirtualMachineSnapshot / VirtualMachineRestore	VM 磁盘（及配置）的时间点快照及其恢复对象。由 CSI `VolumeSnapshot` 提供支持。	作用域覆盖整个 VM 的 `VolumeSnapshot`。
崩溃一致性与应用一致性	如果没有 guest agent，快照为*崩溃一致性*（类似于直接断电）。使用 guest agent 时，会先冻结文件系统，从而生成*应用一致性*快照。	存储备份中常见的 fsfreeze 区别。

从哪里开始

首次使用手册时，可以按以下路径操作：

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（新集群中没有任何镜像）。
3. 创建虚拟机 — 使用该镜像创建第一个 VM。
4. 串行控制台 — 登录（云镜像没有默认密码；请使用创建时设置的 SSH 密钥或密码）。
5. 管理虚拟磁盘 — 挂载数据磁盘。

概览中还提供了“控制台功能 → API 对象”映射表，并且每个操作指南页面都包含可直接复制粘贴的使用 **API** 代码块。

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
规划与要求
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

- 下载与平台架构对应的 **Alauda Virtualization** 安装包。
- 使用上架软件包机制上架 **Alauda Virtualization** 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：
  - 如果需要直接通过 IP 连接虚拟机，则集群必须使用 Kube-OVN Underlay 网络模式。您可以参考最佳实践准备 Kube-OVN Underlay 物理网络。
  - 建议将 TopoLVM 与 Kubevirt 搭配使用，因为它可以提供接近硬件级别的性能。如果性能要求不高，也可以使用 Ceph 分布式存储。

存储产品	描述
TopoLVM	优势：相对轻量且性能良好。 劣势：无法跨节点使用，可靠性较低，且无法提供冗余。
Ceph 分布式存储	优势：可跨节点使用，具有高可用性并支持冗余。 劣势：冗余磁盘副本会导致利用率降低；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，请确保存储所在的网络与虚拟机所在的网络可以相互通信。

规划与要求

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启用 Intel VT-x 或 AMD-V。节点之间的 CPU 型号不一致可能会导致实时迁移失败。
- 实时迁移需要共享存储：只有当虚拟机的所有 PVC 都使用 **ReadWriteMany (RWX)** 访问模式时，虚拟机才能进行实时迁移。对于必须迁移的工作负载，请规划支持 RWX 的存储（例如 Block 模式的 Ceph RBD）。
- 卷模式：对于虚拟机磁盘，建议优先使用 **Block** 卷模式，而不是 Filesystem 卷模式——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 具有绑定的 `VolumeSnapshotClass`。

操作步骤

1 启用节点虚拟化

当平台在非 Alauda OS 上创建的业务集群的节点为物理机时，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 Virtual Machine Instances（VMI）。

- 启用开关后，允许在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。
- 禁用开关后，禁止在物理机节点上调度新的虚拟机，但不会影响该节点上已经运行的虚拟机。

操作步骤

1. 进入管理员。
2. 在左侧导航栏中，单击集群管理 > 集群。
3. 单击类型为 **On-Premises** 的*集群名称*。
4. 在节点页签中，单击要设置虚拟化开关的节点右侧的⋮> 启用虚拟化。
5. 单击确认。

2 部署 **Operator**

1. 登录并进入管理员页面。
2. 单击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Virtualization**，单击安装，进入安装 **Alauda Virtualization** 页面。
   配置参数：

参数	推荐配置
Channel	默认 channel 为 `alpha`。
Installation Mode	`Cluster`：集群中的所有命名空间共享单个 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`：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 创建 HyperConverged 实例

1. 进入管理员。
2. 单击 **Marketplace** > **OperatorHub**。
3. 找到 **Alauda Virtualization**，单击它进入 **Alauda Virtualization** 详情页面。
4. 单击所有实例
5. 在 **HyperConverged** 实例卡片上单击创建实例。
   注意：每个集群中只需创建一个 **HyperConverged** 实例。
6. 切换到 YAML 视图，仅将示例中 `spec.storageImport.insecureRegistries` 字段内指定的*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`，其他参数保持默认值。

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

替换结果：

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

随附的示例还将 `spec.evictionStrategy` 设置为 `LiveMigrateIfPossible`，因此节点排空时，会对可以迁移的虚拟机执行实时迁移，而不是停止这些虚拟机。除非有理由进行更改，否则请保留此值：如果未显式设置，operator 在 ACP 上会回退到

`None`，此时排空节点会关闭其上的虚拟机。有关每种策略的完整行为，请参阅 KubeVirt 节点维护。

```
spec:
  evictionStrategy: LiveMigrateIfPossible
```

7. 单击创建，等待 CDI 和 KubeVirt 类型的实例自动创建到资源列表中，同时确保 YAML 中显示的 **status.phase** 为 `deployed`，表示 HyperConverged 实例已成功创建。
   或者，也可以通过命令行应用 `HyperConverged` 资源，并等待其状态变为 `Available`：

```
kubectl apply -f - <<'EOF'
apiVersion: hco.kubevirt.io/v1beta1
kind: HyperConverged
metadata:
  name: kubevirt-hyperconverged
  namespace: kubevirt
spec:
  # Match the shipped example: without this the operator defaults
to None on
  # ACP and a node drain stops the virtual machines instead of mig
rating them.
  evictionStrategy: LiveMigrateIfPossible
  # Remaining defaults are applied by the operator; tune liveMigra
tionConfig,
  # 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，建议配置值介于 2 和 4 之间。
> - 启用虚拟机超售比后，创建虚拟机时，容器的 request 值（requests）会固定为指定的 **limit** 值（**limits**）**/**虚拟机超售比，因此用户通过 YAML 设置的 request 将不会生效。例如：假设虚拟机的 CPU 资源超售比设置为 4，用户创建虚拟机时指定 CPU limit 值为 4c，则 CPU request 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 内存通常大于虚拟机实际可用的内存，建议预留 20% 的资源。当命名空间中的剩余可用资源低于 20% 时，请及时扩容资源。

KubeVirt™ 是 The Linux Foundation 的商标。Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识目的。

镜像

介绍

介绍

优势

镜像源和镜像实例

从集群外部上传镜像

操作指南

添加镜像源

操作步骤

默认规范

使用 API

更新/删除镜像源

更新/删除镜

添加镜像实例

前提条件

管理镜像实例

状态

称

命名空间中的镜像实例

查看镜像实例

向命名空间添加镜像实例

从镜像实例创建虚拟机

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

从使用自签名证书的 HTTPS 服务

工作原理

约束

先决条件

操作步骤

使用 KubeV

前提条件

限制

操作步骤

远程访问

使用 KubeVirt 基于 ISO 创建 Linux 镜像

前提条件

限制与约束

操作步骤

导出虚拟机

操作步骤

权限

权限

介绍

Alauda Virtualization 利用 Kubernetes 扩展 API 能力，将虚拟机镜像抽象为 **Custom Resource Definition**（CRD）。它为用户提供用户界面（UI），方便用户将存储在远程存储库中的虚拟机镜像导入 ACP 以供使用。

目录

优势

镜像源和镜像实例

从集群外部上传镜像

优势

- 支持主流操作系统
  支持各种常用的 Linux 发行版和 Windows 操作系统。
- 支持多种架构
  兼容 **X86_64** 和 **ARM64** 架构。
- 支持多种来源
  支持从以下来源导入虚拟机镜像：
  - 镜像仓库
  - 文件服务器（HTTP 和 HTTPS，包括使用私有 CA 的服务器）

  - 兼容 S3 的对象存储
  - 集群中已有的磁盘和卷快照
- 支持多种格式
  支持 **QCOW2** 和 **RAW** 格式的虚拟机镜像。

镜像源和镜像实例

控制台分两个层级管理镜像：

	镜像源	镜像实例
定义	镜像的存储位置（仓库、HTTP 或 S3 地址）以及可使用该镜像的项目	导入集群并作为可直接启动的卷的镜像
Kubernetes 资源	`VirtualMachineImageTemplate`，由所有集群共享的平台资源	`DataSource`，其中包含由 CDI 填充的 PersistentVolumeClaim，位于某个集群的一个命名空间中
管理位置	管理员 > 虚拟化 > 镜像 > 镜像源	管理员 > 虚拟化 > 镜像 > 镜像实例（共享，位于 `kube-public`），或 容器平台 > 虚拟化 > 镜像 > 镜像实例（位于某个命名空间中）
从中创建的虚拟机	从该地址导入其系统磁盘	从该卷克隆其系统磁盘 — 速度更快，并且不会再次访问镜像服务器

`kube-public` 中的镜像实例由集群的每个命名空间共享；命名空间中的镜像实例仅供该命名空间使用。镜像实例还可以包含镜像源无法提供的导入设置，例如 HTTPS 服务器的 CA 证书，并且两者都可以声明默认实例类型和偏好设置。

- 添加镜像源
- 添加镜像实例
- 管理镜像实例
- 命名空间中的镜像实例

从集群外部上传镜像

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`，该命令会将文件流式传输到 CDI 上传代理。必须从运行命令的位置访问 `cdi-uploadproxy` Service（位于 `kubevirt` 命名空间中）——通过 Gateway API、Ingress 或 `NodePort` / `LoadBalancer` Service 暴露该 Service，以便管理员从集群外部上传镜像。有关完整的操作步骤（包括暴露代理以及从上传的镜像创建虚拟机），请参阅上传本地镜像并创建虚拟机；有关 `virtctl` 的参考信息，请参阅使用 virtctl 命令行工具。

KubeVirt™ 是 The Linux Foundation 的商标。Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，本文仅出于标识目的使用这些商标。

操作指南

添加镜像源

操作步骤

默认规范

使用 API

更新/删除镜像源

更新/删除镜

添加镜像实例

前提条件

管理镜像实

状态

称

命名空间中的镜像实例

查看镜像实例

向命名空间添加镜像实例

从镜像实例创建虚拟机

添加镜像源

镜像源记录虚拟机镜像的存储位置 — 容器 registry、HTTP 服务器或 S3 存储桶 — 以及哪些项目可以使用该镜像源。它是平台资源（`VirtualMachineImageTemplate`），由所有集群共享。虚拟机可以直接从镜像源导入其系统磁盘，或者管理员可以将其一次性导入集群作为镜像实例，然后从镜像实例克隆虚拟机。

平台支持 **X86_64** 和 **ARM64 (Alpha)** 镜像。

目录

操作步骤

默认规范

使用 API

操作步骤

1. 访问 管理员。
2. 在左侧导航栏中，点击 虚拟化 > 镜像 > 镜像源。
3. 点击 添加镜像源。
4. 参考以下说明配置相关参数。

参数	描述
名称 / 显示名称	镜像源的名称和可选的显示名称。
操作系统	**CentOS**、**Ubuntu**、**RedHat**、**Debian**、**TLinux**、**Other Linux** 或 **Windows (Alpha)**。
源	• **Registry**：存储在容器镜像 registry 中的虚拟机镜像（容器磁盘）。 • **HTTP**：文件服务器上的虚拟机镜像文件，通过 HTTP 或 HTTPS 访问。 • 对象存储 **(S3)**：S3 兼容对象存储中的虚拟机镜像。无需身份验证的 S3 镜像也可以使用 **HTTP** 添加。
CPU 架构	镜像的 CPU 架构。对于 registry 源，可以选择两种架构；对于其他源，只能选择一种架构。
镜像 **URL**	`qcow2` 或 `raw` 格式的 KVM 虚拟机镜像。 • Registry：`repository_address:image_version`，例如 `registry.example.com/library/ubuntu:latest`。 • HTTP：镜像文件的 URL，以 `http://` 或 `https://` 开头，例如 `http://192.168.0.1/vm_image/centos_7.8.qcow2`。 • S3：对象的地址，例如 `https://endpoint/bucket/centos.qcow2`。
授权	如果源需要身份验证，请开启此项，然后选择镜像凭证或点击 添加凭证；仅支持 用户名/密码 凭证。S3 源无法关闭授权。
已分配的项目	• 所有项目：每个项目都可以使用该镜像。 • 指定项目：仅选定的项目可以使用该镜像。 • 无：暂时不分配任何项目；稍后通过更新镜像源来分配项目。

5. 点击 添加。

注意：

- HTTPS 证书为自签名证书或由私有 CA 签发的镜像服务器无法直接从镜像源使用：CDI 需要 CA 证书，而 CA 证书仅由镜像实例携带。从镜像源添加带有 **CA** 证书的镜像实例，然后从该镜像实例创建虚拟机。在创建虚拟机表单中选择此类镜像源时，表单会提醒您这一点。
- 需要身份验证的镜像源或 S3 镜像源只能在命名空间中导入为镜像实例，因为凭证 Secret 位于该命名空间中；请参阅命名空间中的镜像实例。

默认规范

镜像源可以通过四个 `instancetype.kubevirt.io/default-*` 标签，声明从该镜像创建的虚拟机应使用的实例类型和偏好设置。镜像源由所有集群共享，而实例类型和偏好设置属于某个集群，因此控制台不提供这些设置：请在 YAML 中声明，并且只能使用在所有使用该镜像的集群上都存在的名称。在控制台中编辑镜像源时，这些标签会保留。

平台提供的镜像源声明了实例类型 `u1.small`，并在 X86_64 上声明了偏好设置 `rhel.7`（CentOS 7）或 `ubuntu`（Ubuntu 20.04）。

这些标签的使用方式如下：

- 从镜像源添加的镜像实例会继承这些标签。
- 从镜像源创建虚拟机时，实例类型和偏好设置会在模型区域中预先选定。如果集群上不存在该名称，则不会预先选定，并且表单会对此进行提示。

使用 **API**

```yaml
apiVersion: virtualization.alauda.io/v1
kind: VirtualMachineImageTemplate
metadata:
  name: centos-7-9
  labels:
    virtualization.cpaas.io/image-os-type: centos
    virtualization.cpaas.io/image-os-arch: amd64          # amd64 | arm64 | amd64.arm64
    project.cpaas.io/ALL_ALL: "true"                      # allocated to all projects
    # optional default specification
    instancetype.kubevirt.io/default-instancetype: u1.small
    instancetype.kubevirt.io/default-instancetype-kind: VirtualMachineClusterInstancetype
    instancetype.kubevirt.io/default-preference: rhel.7
    instancetype.kubevirt.io/default-preference-kind: VirtualMachineClusterPreference
  annotations:
    cpaas.io/display-name: CentOS 7.9
spec:
  source: http                      # registry | http | s3
  url: https://images.example.com/centos-7.9.qcow2
  # secret:
  #   name: image-credential        # for a source that requires authentication
```

在 `global` 集群上应用：

```yaml
kubectl apply -f centos-7-9.yaml
kubectl get virtualmachineimagetemplates
```

更新/删除镜像源

1. 导航至管理员。
2. 在左侧边栏中，点击虚拟化 > 镜像 > 镜像源。
3. 点击⋮> 更新/删除。
4. 确认后，点击更新/删除。

更新/删除镜像凭据

1. 转到管理员。
2. 在左侧导航栏中，点击虚拟化 > 镜像 > 镜像源。
3. 在镜像凭据选项卡中，点击⋮> 更新/删除。
4. 确认后，点击更新/删除。

添加镜像实例

镜像实例是已导入集群、可作为启动卷使用的操作系统镜像。虚拟机从该镜像克隆其系统磁盘，而不是再次下载镜像，因此从一个镜像创建大量虚拟机的速度很快，也不会给镜像服务器带来负载。

在 Kubernetes 术语中，镜像实例是一个指向 PersistentVolumeClaim 的 `DataSource`（`cdi.kubevirt.io/v1beta1`）。CDI 通过导入 `DataVolume` 填充该声明——从镜像地址导入，或克隆现有磁盘或卷快照。在 API 侧，可启动卷中介绍了相同的对象。

在管理员下添加的镜像实例位于集群的共享 `kube-public` 命名空间中，可供每个命名空间中的虚拟机使用。若要添加仅供一个命名空间使用的镜像实例，包括位于 S3 上的镜像以及需要凭证的镜像，请参阅命名空间中的镜像实例。

目录

前提条件
来源
操作步骤
存储设置
导入设置
CA 证书
请求标头
校验和
拉取方式
使用 API

前提条件

- 集群中已安装 Virtualization。
- 用于镜像卷的存储类。建议使用其 StorageProfile 提供读写多次和块设备的存储类（例如 Ceph RBD 块存储）；请参阅存储设置。
- 若要从镜像源添加，必须已添加该镜像源；请参阅添加镜像源。

来源

来源	导入内容	`DataVolume` 源
镜像源	平台管理员添加的镜像源地址（registry 或 HTTP）。需要凭证的镜像源和 S3 镜像源会列出，但无法在此处选择，因为 `kube-public` 不包含 Secret；请从命名空间中添加。名称会根据镜像源预先填充，其默认实例类型和首选项也会一并继承（请参阅默认规格）。	`registry` 或 `http`
网络	直接输入的镜像地址：**Registry**（`registry.example.com/images/centos:7`；系统会为你添加 `docker://`）或 **HTTP**（`http://` 或 `https://` URL，指向 `qcow2` / `raw` 文件）。此页面无法使用凭证，且此处不提供对象存储（**S3**）——S3 始终需要凭证 Secret，因此请从命名空间中添加 S3 镜像。	`registry` 或 `http`
磁盘	克隆任意命名空间中的现有磁盘：`DataVolume`，或不包含该对象的 PersistentVolumeClaim。正在运行的虚拟机的磁盘会列出，但无法选择（*由正在运行的虚拟机使用……；请先将其停止*）。架构默认为磁盘所属虚拟机的架构。	`pvc`
卷快照	克隆一个已准备就绪、可供使用的 `VolumeSnapshot`。虚拟机快照创建的快照会标记为*来自虚拟机快照……*。	`snapshot`

对于磁盘或卷快照，实例大小默认为来源的大小，且不能小于来源大小。之后删除来源不会影响镜像实例：镜像实例拥有自己的副本。

操作步骤

1. 进入管理员。
2. 在左侧导航栏中，点击虚拟化 > 镜像 > 镜像实例，然后选择集群。
3. 点击添加镜像实例。
4. 配置参数。

参数	说明
来源	镜像源、网络、磁盘或卷快照；请参阅来源。
操作系统 / **CPU** 架构	创建虚拟机时，用于查找镜像实例。带有两种架构标签的镜像源可按任一架构添加。
镜像源 / 来源类型和镜像 **URL** / 磁盘 / 卷快照	根据来源指定要导入的内容。
名称	最多 53 个字符。镜像源会预填充此字段；磁盘或卷快照来源时此字段为空。
显示名称	可选。
实例大小	镜像卷的大小，至少与镜像的虚拟大小相同。默认值为 30 Gi（最小值为 1 Gi）；对于磁盘或卷快照来源，默认值为来源的大小。
StorageClass / 卷模式 / 访问模式	请参阅存储设置。
CA 证书、请求标头、校验和、拉取方式	根据镜像地址的类型显示；请参阅导入设置。磁盘或卷快照来源不使用这些设置。

5. 点击添加。

控制台会先对每个对象执行试运行检查，然后创建 `DataSource`、CA 证书 `ConfigMap`（如果有）和导入 `DataVolume`。如果其中任何一个对象被拒绝，已创建的对象都会被删除，因此

添加失败后不会留下任何对象。CDI 填充卷之前，新镜像实例的状态会显示为正在导入，之后变为可用；请参阅状态。

存储设置

存储字段按以下顺序排列，因为每个字段都会缩小下一个字段的范围：实例大小、**StorageClass**、卷模式、访问模式。

提供的卷模式和访问模式是 CDI 在其 `StorageProfile` 中为该存储类报告的模式。选择存储类后，控制台会选择推荐的组合：

1. 如果有可用选项，则选择带有 **ReadWriteMany** 的 **Block Device** — 卷可以同时挂载到多个节点上，这是实时迁移所需的；
2. 否则选择任何带有 **ReadWriteMany** 的组合（例如 CephFS 上的 **File System**）；
3. 否则选择配置文件中的第一个组合。

提供的访问模式取决于你选择的卷模式。如果在存储类支持 **ReadWriteMany** 的情况下选择了其他访问模式，表单会对此进行提示。如果无法读取存储类的 `StorageProfile`，则模式改为遵循平台的存储类规则。

检查存储类支持的内容：

```
kubectl get storageprofile sc-ceph-cluster-block -o jsonpath='{.status.cl
aimPropertySets}{"\n"}'
# [{"accessModes":["ReadWriteMany"],"volumeMode":"Block"},
#  {"accessModes":["ReadWriteOnce"],"volumeMode":"Block"},
#  {"accessModes":["ReadWriteOnce"],"volumeMode":"Filesystem"}]
```

导入设置

地址类型	设置
HTTP / HTTPS	**CA** 证书、请求标头、校验和
Registry	**CA** 证书、拉取方式

地址类型	设置
S3（仅命名空间）	**CA** 证书、凭证 Secret — 请参阅命名空间中的镜像实例

这些设置会写入导入 `DataVolume`，并与镜像实例一同保留，因此重新导入时会再次使用这些设置。

CA 证书

对于证书为自签名证书或由私有 CA 签发的镜像服务器，请将 CA 证书粘贴为 PEM 文本，或点击加载文件以加载 PEM 或 DER 文件（DER 文件会显示为 PEM）。可以连续粘贴多个证书，例如中间 CA 和根 CA。只接受证书 — 私钥和其他文本会被拒绝。

证书会存储在镜像实例所拥有的 `ConfigMap` `<image instance name>-img-ca` 中，使用键 `tls.crt`，并在删除镜像实例时一同删除；`DataVolume` 会以 `certConfigMap` 的形式引用该证书。没有 CA 证书时，不会创建 `ConfigMap`。镜像地址中的主机必须位于证书的 Subject Alternative Names 中；CDI 不会跳过主机名验证。有关背景信息和故障排除，请参阅从带有自签名证书的 HTTPS 服务器导入虚拟机镜像。

证书仅在你的浏览器中进行检查。表单会列出每个证书 — 角色（根证书、中间证书或叶证书）、主题、颁发者、有效期（临近到期时显示 *expires in N days*）、Subject Alternative Names 和 SHA-256 指纹（与 `openssl x509 -noout -fingerprint -sha256` 相同）。不会访问镜像地址以及证书中的任何 OCSP、CA 颁发者或 CRL 地址；只有在点击添加时，证书才会发送到集群。

由于导入会失败，以下问题会阻止添加镜像实例：

- 证书文本大于 64 KiB；
- 无法读取证书；
- 证书的序列号为负数；
- 证书的 RSA 密钥短于 1024 位；
- 所有证书均已过期或尚未生效。

以下属于警告；你仍然可以添加镜像实例：

- 仅包含服务器证书 — 请添加签发该证书的 CA，否则服务器证书续期后导入会失败；

- 某个证书将在 30 天内到期；
- 某些证书已过期或尚未生效；
- 某个证书使用 SHA-1 或 MD5 签名；
- 某个证书签发了另一个证书，但未标记为 CA；
- 镜像地址使用 `http://`，因此不会使用 CA 证书；
- 叶证书对镜像地址的主机无效（IP 地址必须是 IP SAN；通配符只能匹配一个标签）。

请求标头

仅适用于 HTTP 地址。每行输入一个标头，格式为 `Name: value`；这些标头会随下载一起发送（`extraHeaders`）。标头以纯文本形式存储在 `DataVolume` 中 — 请勿在其中放置机密信息；请改用命名空间中的凭证 Secret。

校验和

仅适用于 HTTP 地址。以 `md5:`、`sha1:`、`sha256:` 或 `sha512:` 后跟十六进制摘要的格式输入预期摘要，例如 `sha256:c199c9b6…875a`。CDI 会据此验证下载的镜像；不匹配会导致导入永久失败，镜像实例会显示为失败，原因是校验和不匹配。

拉取方式

仅适用于 Registry 地址。

- **Pod**（默认）：导入器 Pod 会拉取镜像，并通过临时空间将其复制到卷中。
- **Node**：节点会将镜像作为容器镜像拉取，因此该镜像必须是节点能够拉取的容器磁盘（包括通过节点的 Registry 凭证拉取）；不会使用临时空间。

凭证 Secret 只能由命名空间中的镜像实例使用，并且仅用于 **Pod**；选择凭证后，**Node** 将不可用。请参阅凭证。

平台随附的镜像源默认为 **Node**。

使用 API

对于从带有私有 CA 的 HTTPS 地址导入、名为 `centos-7` 的镜像实例，控制台会在 `kube-public` 中创建以下对象。使用 `kubectl` 应用相同的对象，即可在不使用控制台的情况下添加镜像实例。

```yaml
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataSource
metadata:
  name: centos-7
  namespace: kube-public
  labels:
    virtualization.cpaas.io/image-instance: "true"
    virtualization.cpaas.io/image-os-type: centos        # the operating system
    virtualization.cpaas.io/image-os-arch: amd64         # amd64 | arm64
    virtualization.cpaas.io/source-type: address         # image-source | address | disk | volume-snapshot
    virtualization.cpaas.io/size: 30Gi
    virtualization.cpaas.io/storage-class: sc-ceph-cluster-block
    virtualization.cpaas.io/volume-mode: Block
    virtualization.cpaas.io/access-mode: ReadWriteMany
  annotations:
    cpaas.io/display-name: CentOS 7.9
spec:
  source:
    pvc:
      name: centos-7-img-a1b2c          # the import DataVolume below
      namespace: kube-public
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: centos-7-img-ca
  namespace: kube-public
data:
  tls.crt: |
    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
---
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: centos-7-img-a1b2c              # <image instance>-img-<5 random characters>
  namespace: kube-public
  labels:
    virtualization.cpaas.io/image-instance-name: centos-7
```

```
  annotations:
    cdi.kubevirt.io/storage.bind.immediate.requested: "true"
spec:
  source:
    http:
      url: https://images.example.internal/centos-7.qcow2
      certConfigMap: centos-7-img-ca
      extraHeaders:
        - "X-Request-Source: platform"
      checksum: sha256:<hex digest>
  storage:
    storageClassName: sc-ceph-cluster-block
    volumeMode: Block
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 30Gi
```

控制台还会将镜像实例设为 `DataVolume` 和 `ConfigMap` 的控制器所有者，因此删除 `DataSource` 时也会删除这些对象；手动应用但不包含 `ownerReference` 的对象必须逐个删除。

对于其他地址类型，请将 `DataVolume` 的 `spec.source` 替换为以下选项之一。

Registry，由节点拉取：

```
source:
  registry:
    url: docker://registry.example.com/images/centos:7
    pullMethod: node
    # certConfigMap: centos-7-img-ca    # a registry behind a private CA
```

另一个命名空间中的磁盘（克隆）：

```
source:
  pvc:
    name: web-01-rootfs
    namespace: demo
```

卷快照（克隆）：

```
source:
  snapshot:
    name: web-01-snapshot
    namespace: demo
```

查看导入状态：

```
kubectl get dv,datasource -n kube-public
# NAME                                                PHASE       PROGRESS
# datavolume.cdi.kubevirt.io/centos-7-img-a1b2c       Succeeded   100.0%
```

管理镜像实例

在 管理员 > 虚拟化 > 镜像 > 镜像实例下，列表显示所选集群的镜像实例及其状态，并且每行都提供更新显示名称、更新默认规格、重新导入和删除操作。点击名称可打开其详细信息：导入源、导入 `DataVolume`、声明、阶段、进度以及失败原因。命名空间的镜像实例管理方式相同，请参阅命名空间中的镜像实例。

目录

状态
重新导入
默认规格
更新默认规格
更新显示名称
删除

状态

状态	含义
可用	镜像实例可用于创建虚拟机。

状态	含义
导入中	CDI 正在填充卷；页面会显示进度。*此镜像实例的存储已删除，正在重新导入*表示声明已被删除，但导入 `DataVolume` 仍存在，因此 CDI 会重新创建声明。
导入错误	导入持续失败，CDI 会重试；下方的原因会说明失败原因。
失败	导入已确定失败（例如校验和不匹配）；请重新导入或删除镜像实例。
不完整	导入 `DataVolume` 和声明都已不存在。请重新导入或删除镜像实例。
等待源摘要	由尚未找到要导入镜像的 `DataImportCron` 管理的镜像实例。
不可用	不符合上述任何情况；请检查 `DataSource` 条件。

在重新导入期间，状态旁会显示正在重新导入（含进度）或重新导入错误。

导入错误或失败时，会显示原因和 CDI 的原始消息：

原因	典型原因
校验和不匹配	下载的镜像与校验和不匹配。
找不到镜像	URL 或镜像标签错误。
身份验证失败	凭证错误或缺失。
凭证 **Secret** 不完整	Secret 缺少 `accessKeyId` 或 `secretKey`。
证书不受信任	服务器证书是自签名证书或来自私有 CA，且未提供匹配的 **CA** 证书。
证书与地址不匹配	镜像地址的主机不在证书的主题备用名称中。
大小小于镜像	实例大小小于镜像的虚拟大小。
超出配额	命名空间的存储配额已用尽。
存储类不可用	存储类不存在，或无法置备所请求的模式。

原因	典型原因
导入 **Pod** 无法启动	无法调度或启动导入器 Pod。
源正在使用	磁盘克隆未启动，因为磁盘正在使用中（CDI 报告 `CloneSourceInUse`）；请停止使用该磁盘的虚拟机。
导入失败	其他任何导入失败。

只有可用的镜像实例才会在创建虚拟机时提供；对于由声明支持的镜像实例，其声明还必须已绑定且导入成功。

重新导入

重新导入会使用相同的地址和相同的导入设置（CA 证书、请求标头、校验和、拉取方法、凭证）再次导入镜像，例如服务器上的镜像更新后。磁盘或卷快照也会再次克隆。

1. 在镜像实例上单击⋮> 重新导入。
2. 确认。对话框会显示要导入的地址。

新的导入会在当前项旁边的新 `DataVolume` 中运行。可用的镜像实例会保留当前镜像，直到新的导入成功；成功后，它会切换到新卷，并在没有克隆仍从之前的 `DataVolume` 读取数据后将其删除。如果新的导入失败，镜像实例会保留当前镜像，并显示重新导入错误。

如果镜像实例的 CA 证书 `ConfigMap` 已被删除（*CA ConfigMap … 缺失*），则会拒绝重新导入；请删除镜像实例，然后使用其 CA 证书再次添加。

默认规格

镜像实例可以声明从其创建的虚拟机应使用的实例类型和偏好。这些设置作为其 `DataSource` 上的四个标签存储：

标签	值
`instancetype.kubevirt.io/default-instancetype`	实例类型的名称
`instancetype.kubevirt.io/default-instancetype-kind`	`VirtualMachineClusterInstancetype`
`instancetype.kubevirt.io/default-preference`	偏好的名称
`instancetype.kubevirt.io/default-preference-kind`	`VirtualMachineClusterPreference`

从镜像源添加的镜像实例会从镜像源获取这些标签（请参阅添加镜像源）；从 registry 导入的镜像也可能在镜像本身中携带这些标签。您从镜像实例创建虚拟机时，这些标签会在模型区域中预先选中。

更新默认规格

1. 在镜像实例上单击⋮> 更新默认规格。
2. 选择集群的默认实例类型和默认偏好，或清空字段以移除对应设置。以镜像实例的操作系统命名的偏好会标记为建议并列在最前面。
3. 单击更新。

CDI 会将这些标签复制到 `DataSource` 上，来源包括其背后的对象——声明和导入 `DataVolume`、卷快照或 `DataImportCron`；如果 `DataVolume` 不包含其中任何标签，CDI 还会从克隆它的卷中复制这些标签。因此，控制台也会将您的选择写入这些对象。清空的字段会丢失其名称标签，但会保留其类型标签：虚拟机和控制台只会从名称标签读取默认值，而保留的类型标签会阻止 CDI 将克隆卷的值复制回来。重新导入会保留您的选择，包括已清空的字段；只有当 registry 镜像在镜像本身中声明了默认值时，重新导入才会再次为已清空的字段设置该默认值——之后请再次将其清空。

对于 `kubectl` 也是如此——先为声明和导入 `DataVolume` 添加标签，然后再为 `DataSource` 添加标签：

```
CLAIM=$(kubectl get datasource centos-7 -n kube-public -o jsonpath='{.spec.source.pvc.name}')
for obj in pvc/$CLAIM datavolume/$CLAIM datasource/centos-7; do
  kubectl label -n kube-public "$obj" --overwrite \
    instancetype.kubevirt.io/default-instancetype=u1.medium \
    instancetype.kubevirt.io/default-instancetype-kind=VirtualMachineClusterInstancetype \
    instancetype.kubevirt.io/default-preference=centos.stream9 \
    instancetype.kubevirt.io/default-preference-kind=VirtualMachineClusterPreference
done
```

要清除默认偏好，请在相同的对象上移除其名称标签，并保留其类型标签：

```
for obj in pvc/$CLAIM datavolume/$CLAIM datasource/centos-7; do
  kubectl label -n kube-public "$obj" --overwrite \
    instancetype.kubevirt.io/default-preference- \
    instancetype.kubevirt.io/default-preference-kind=VirtualMachineClusterPreference
done
```

更改默认规格会影响之后创建的虚拟机；现有虚拟机会保留创建时使用的设置。

更新显示名称

单击⋮> 更新显示名称。显示名称存储在 `cpaas.io/display-name` 的 `DataSource` 注解中。

删除

1. 在镜像实例上单击⋮> 删除。
2. 对话框会列出镜像实例的读取方：系统磁盘无法再通过该实例重新创建的虚拟机、无法再创建成员的虚拟机池，以及仍在从该实例读取数据且将无法完成的克隆。已从镜像实例创建的磁盘不受影响。
3. 输入镜像实例的名称并确认。

删除的内容取决于镜像实例的创建方式：

镜像实例	删除内容
在控制台中添加（使用导入 `DataVolume`）	`DataSource`；导入 `DataVolume`、其声明和 CA 证书 `ConfigMap` 会通过其所有者引用一并删除。
由 `DataImportCron` 管理	`DataImportCron`，它会删除其管理的内容。
从现有卷发布	`DataSource`；勾选删除存储也会删除该卷。

对于 `kubectl`，请删除在控制台中添加的镜像实例的 `DataSource`；其余内容会随之删除：

```
kubectl delete datasource centos-7 -n kube-public
```

命名空间中的镜像实例

除了管理员添加到 `kube-public` 的共享镜像实例（参见添加镜像实例）外，每个命名空间也可以拥有自己的镜像实例。一个项目会将不同集群中的命名空间归为一组；镜像实例始终位于一个集群中的一个命名空间——也就是你在容器平台中操作的命名空间——且只有该命名空间中的虚拟机可以使用它。

如果镜像不应与所有命名空间共享，或导入时需要凭证（例如 S3 上的镜像，或需要身份验证的 registry 或 HTTP 服务器），请使用命名空间镜像实例。`kube-public` 不保存 Secret，因此此类镜像只能在命名空间中添加。

目录

查看镜像实例

向命名空间添加镜像实例

凭证

从镜像实例创建虚拟机

查看镜像实例

1. 进入容器平台，然后选择项目、集群和命名空间。
2. 在左侧导航栏中，点击虚拟化 > 镜像 > 镜像实例。

列表显示：

- 该命名空间中各种状态的镜像实例及其操作（更新显示名称、更新默认规格、重新导入、删除和创建虚拟机）——参见管理镜像实例；
- `kube-public` 中可用的镜像实例，标记为 `kube-public`。这些实例在此处为只读：唯一可用的操作是创建虚拟机。

向命名空间添加镜像实例

添加镜像实例需要具备在命名空间中创建 `DataSource` 和 `DataVolume` 对象的权限。表单与管理员页面上的相同（参见添加镜像实例），但有以下区别：

- 所有对象——`DataSource`、CA 证书 `ConfigMap` 和导入 `DataVolume`——都在该命名空间中创建。
- 镜像源列出分配给该项目的镜像源。需要凭证的镜像源以及 S3 镜像源也可以选择；此时必须选择凭证。
- 网络提供**Registry**、**HTTP**和对象存储 **(S3)**。S3 地址的格式为 `https://endpoint/bucket/object`（或 `http://`），并且需要凭证；也可以附带**CA** 证书。
- 磁盘和卷快照仅列出此命名空间和 `kube-public` 中的磁盘与卷快照。

凭证

凭证会列出该命名空间中同时包含 `accessKeyId` 和 `secretKey` 这两个键的 Secret——这是 CDI 用于 registry、HTTP basic 和 S3 身份验证的格式。导入 `DataVolume` 会将该 Secret 引用为 `secretRef`；控制台不会创建或复制 Secret。添加镜像实例前，请先创建 Secret：

```
kubectl create secret generic s3-images -n demo \
  --from-literal=accessKeyId=<access key> \
  --from-literal=secretKey=<secret key>
```

命名空间 `demo` 中 S3 镜像实例的导入 `DataVolume`：

```
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: rocky-9-img-k3x7q
  namespace: demo
spec:
  source:
    s3:
      url: http://rook-ceph-rgw-ceph-object.rook-ceph.svc:7480/images/roc
ky-9.qcow2
      secretRef: s3-images
      # certConfigMap: rocky-9-img-ca   # an endpoint with a private CA
  storage:
    storageClassName: sc-ceph-cluster-block
    volumeMode: Block
    accessModes:
      - ReadWriteMany
    resources:
      requests:
        storage: 30Gi
```

其 `DataSource` 与添加镜像实例中的相同，位于命名空间 `demo` 中，并使用 `virtualization.cpaas.io/source-type: address`。

对于 registry 地址，凭证仅用于 **Pod** 拉取方式：选择凭证后，拉取方式会设为 **Pod**，且节点不可用。选择节点时，节点的容器运行时会使用节点自身对 registry 的访问权限拉取镜像；CDI 只会将 Secret 传递给导入器容器，因此节点不会使用该 Secret。

从镜像实例创建虚拟机

在命名空间或 `kube-public` 的某一行中单击 **Create Virtual Machine**。命名空间的 **Create Virtual Machine** 表单将打开，其中 **Provision Method** > **Image Instance**、架构和镜像实例已预先选择，同时也预先选择了镜像实例的默认实例类型和偏好设置。

创建虚拟机或虚拟机池，或重新安装操作系统时，**Image Instance** 选择器会列出 `kube-public` 的镜像实例以及虚拟机所在命名空间的镜像实例；命名空间中的镜像实例会标记命名空间名称。不提供其他命名空间的镜像实例。系统磁盘会从镜像实例所在的命名空间克隆该镜像实例：

```
spec:
  dataVolumeTemplates:
    - metadata:
        name: web-01-rootfs
      spec:
        sourceRef:
          kind: DataSource
          name: rocky-9
          namespace: demo        # kube-public for a shared image instanc
e
        storage:
          storageClassName: sc-ceph-cluster-block
          volumeMode: Block
          accessModes:
            - ReadWriteMany
          resources:
            requests:
              storage: 30Gi
```

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

从使用自签名证书的 HTTPS 服务

工作原理

约束

先决条件

操作步骤

使用 KubeV

前提条件

限制

操作步骤

远程访问

使用 KubeVirt 基于 ISO 创建 Linux 镜像

前提条件

限制与约束

操作步骤

导出虚拟机

操作步骤

上传本地镜像并创建虚拟机

当您已经在本地计算机上拥有虚拟机磁盘镜像（例如从操作系统供应商处下载的 `qcow2`、`raw` 或 `img` 文件），并希望从该镜像运行虚拟机，而不是从远程 HTTP、registry 或 S3 源导入镜像时，请使用此操作步骤。

在 Alauda Container Platform 上，这是一个基于 KubeVirt `virtctl` 客户端和 CDI 上传代理的命令行工作流 — Web 控制台目前尚不提供本地上传源。上传的镜像将存储为 `DataSource`（可引导卷），虚拟机可以从中引导，并且多个虚拟机可以重复使用该卷。

> 提示：如果您希望从 ISO 安装操作系统，请参阅 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

将 CDI 上传代理暴露到集群外部

将 `virtctl` 指向上传代理 URL

上传本地镜像

从已上传的镜像创建虚拟机

创建 Windows 虚拟机

前提条件

- 已安装 `virtctl` 命令行客户端，且其版本与集群的 KubeVirt 版本匹配。 `virtctl` 是标准的 KubeVirt CLI；请从 KubeVirt releases 下载与集群匹配的二进制文件，并将其放置在您的 `PATH` 上。
- 已安装并配置 `kubectl` 命令行工具，以访问集群。
- 一个采用 `qcow2`、`raw` 或 `img` 格式的本地磁盘镜像。为了加快上传速度，请先使用 `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` namespace 中）可从运行 `virtctl` 的位置访问 — 请参阅步骤 1。

操作步骤

1 将 CDI 上传代理暴露到集群外部

`virtctl image-upload` 会将镜像流式传输到 `cdi-uploadproxy` Service。该 Service 是 `kubevirt` namespace 中的内部 `ClusterIP` Service。您必须使其可从运行 `virtctl` 的位置访问。

> 注意：在 Alauda Container Platform 上，CDI 不会自行暴露上传代理或发现其外部地址，因此您需要自行暴露该 Service（本步骤），并显式设置上传 URL（步骤 2）。

使用 **TLS** 透传，以便 `virtctl` 直接与 `cdi-uploadproxy` 协商 TLS（代理提供自签名证书，因此客户端需传递 `--insecure`）。对于以下每种方法，后端均为 `kubevirt` namespace 中、端口为 `443` 的 `cdi-uploadproxy` Service；请记下生成的外部 URL，以供步骤 2 使用。

- **Gateway API (Envoy Gateway) —** 推荐。 在 `Gateway` 中添加处于 `Passthrough` 模式的 `TLS` listener，然后创建一个 `TLSRoute`，其后端为端口 `443` 上的 `cdi-uploadproxy` Service。Gateway 的外部地址 — 由 Gateway 的 Service Type 设置的 `LoadBalancer` IP 或 `NodePort` — 将成为上传 URL。有关完整操作步骤，请参阅 配置 GatewayAPI Gateway 和 配置 GatewayAPI Route；有关安装控制器的信息，请参阅 Envoy Gateway Operator。

- **Ingress (ingress-nginx)**。 使用 SSL 透传的 `Ingress` 暴露 `cdi-uploadproxy`。请参阅 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，请直接暴露 Service — 在配置了 MetalLB 的环境中使用 `LoadBalancer`，或使用始终有效的 `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:<nodePort> mapping
  ```

  通过 `https://<node-ip>:<nodePort>` 访问代理。

暴露 Service 后，通过新的端点验证其是否能够响应（证书为自签名证书，因此请使用 `-k`）：

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

2 将 `virtctl` 指向上传代理 URL

`virtctl image-upload` 需要知道步骤 1 中的外部 URL。请选择以下方式之一。

配置一次（持久配置）。 设置 `CDIConfig.spec.uploadProxyURLOverride`，使每个 `virtctl image-upload` 自动发现该 URL。由于 CDI 配置由 HyperConverged (HCO) operator 管理 — 该 operator 会撤销直接编辑，且不暴露此字段 — 因此请通过 HCO 的 jsonpatch annotation 进行设置：

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

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

> 警告：HCO 将 jsonpatch annotation 记录为高级的不受支持机制 — 错误的 patch 可能导致虚拟化堆栈不稳定。请谨慎使用，并删除 annotation (`kubectl annotate hyperconverged kubevirt-hyperconverged -n kubevirt containerizeddataimporter.kubevirt.io/jsonpatch-`) 以恢复原状。

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

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`，使其可作为可引导卷进行选择：

```
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.
```

注意：

- `--size` 必须大于镜像的虚拟磁盘大小（而不是文件大小）。例如，如果镜像包含 24 GiB 的虚拟磁盘，请求至少 30 GiB 的卷。CDI 会拒绝小于未压缩磁盘的卷。

- 使用 `--no-create`（并省略 `--size`），将镜像上传到已存在的 `DataVolume` 中。
- `--insecure` 会跳过对代理自签名证书的验证。

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：由 `virtctl --datasource` 创建的 `DataSource` 不包含用于标记可引导卷的 `virtualization.cpaas.io/*` labels。通过 API（使用 `sourceRef`，如下一步所示）可以完全使用该镜像；如需同时应用该标签约定，请参阅可引导卷。

4 从已上传的镜像创建虚拟机

通过 `sourceRef`，在 `dataVolumeTemplates` 条目中引用 `DataSource`。KubeVirt 会将已上传的卷克隆到新虚拟机的全新磁盘中：

```yaml
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，确认操作系统能够引导；出现登录提示即表示引导成功：

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

> 提示：为其添加所需 labels 后，创建虚拟机时也可以将已上传的镜像作为可引导卷引用 — 请参阅 创建虚拟机 和 可引导卷。

创建 **Windows** 虚拟机

对于 Windows，请先将镜像上传到卷，然后在创建虚拟机时将其克隆，并在首次引导期间应用 `autounattend.xml` 应答文件。上传步骤与上述操作步骤相同（使用 `qcow2` / `raw` / `img` Windows 磁盘镜像）；有关基于 ISO 的安装流程和 VirtIO 驱动程序要求，请参阅 创建基于 ISO 的 Windows 镜像。

从使用自签名证书的 HTTPS 服务器导入虚拟机镜像

当虚拟机镜像（`qcow2` 或 `raw`）通过 **HTTPS** 由其证书为自签名证书或由私有 **CA** 签发的服务器提供时，请使用此操作步骤——这是内部文件服务器的常见配置。CDI（Containerized Data Importer）负责下载镜像，会验证服务器的 TLS 证书，并且无法选择跳过对 **HTTP** 源的此验证。不受信任的证书会导致导入失败，并显示：

```
Unable to connect to http data source: HTTP request errored:
Get "https://images.example.internal/centos.qcow2": tls: failed to verify
certificate:
x509: certificate signed by unknown authority
```

使此类导入成功的受支持方法是通过从 `DataVolume` 使用 `spec.source.http.certConfigMap` 引用的 `ConfigMap`，向 **CDI** 提供证书。导入器会将证书添加到其信任存储中，并据此验证服务器——主机名验证仍处于启用状态，不会绕过任何验证。

目录

- 工作原理
- 约束
- 先决条件
- 操作步骤
  - 获取并检查证书

创建证书 ConfigMap

创建直接导入镜像的虚拟机

（可选）作为黄金镜像导入一次以供控制台使用

故障排除

工作原理

- `spec.source.http.certConfigMap` 指定与 `DataVolume` 位于同一命名空间中的 `ConfigMap`。它不能指向其他命名空间。
- CDI 会将 `ConfigMap` 挂载到导入器 Pod 中，并将其中的每个键加载为 PEM 证书（以 `.` 开头的键会被忽略）。请将键命名为 **`tls.crt`**：从 CDI 1.64 开始，导入器还会将证书传递给 `nbdkit`，后者会在下载开始前检查镜像大小，并且只读取 `tls.crt`。使用任何其他键名仍可成功导入，但会跳过下载前的大小检查（超大镜像只会在下载完成后被拒绝），并且导入器日志会显示来自 `nbdkit` 的 `Problem with the SSL CA cert` 行。一个键可以包含多个串联的 PEM 块，因此可以在一个键中提供私有 CA 链（根证书和中间证书）。
- 证书会添加到导入器的系统信任存储中，因此同一个导入器仍可正常处理公共 HTTPS 源。
- 此字段在独立的 `DataVolume` 中以及 `VirtualMachine` 的 `dataVolumeTemplates` 条目中用法相同，并且在 Alauda Container Platform 随附的所有 CDI 版本中都可用（早于 CDI 1.62 即已支持）。

约束

- 镜像源不能携带证书；镜像实例可以。在 管理员 > 虚拟化 > 镜像 > 镜像源 下添加的镜像源，即使使用自签名服务器的 HTTPS URL，也会被接受；但直接从该镜像源创建的虚拟机会获得一个仅包含 `http.url` 的 `dataVolumeTemplates` 条目，并因上述 `x509` 错误而失败（创建虚拟机表单会指出这一点）。应改为添加带有 **CA** 证书的镜像实例——可在控制台中添加（请参阅添加镜像实例），也可使用步骤 4 中的 YAML 添加——然后从该实例创建虚拟机：这些虚拟机是克隆，不会再连接 HTTPS 服务器。
- 没有集群范围的“为所有导入信任此 CA”设置：只有在 `DataVolume` 本身引用了 `ConfigMap` 时，CDI 才会使用额外证书。

- 证书的主题备用名称（SAN）必须包含 URL 中使用的主机名，可以是 DNS 名称或 IP 地址。即使证书受信任，如果 SAN 不匹配，也会被拒绝（请参阅故障排除）。
- `DataVolume` `spec` 是不可变的，因此无法通过修补来添加缺失的 `certConfigMap`。对于独立的 `DataVolume`，请将其删除并重新创建。对于由虚拟机拥有的 `DataVolume`，请先将 `certConfigMap` 添加到虚拟机的 `dataVolumeTemplates`，然后删除 `DataVolume`；虚拟机控制器会根据更新后的模板重新创建它。如果只删除 `DataVolume` 而不更改模板，重新创建的对象将不包含证书。

先决条件

- 服务器的 PEM 格式证书：对于自签名服务器，使用服务器证书本身；对于私有 CA，使用 CA 证书（如有中间证书，也一并提供）。请向服务器所有者索取文件，或从服务器获取（步骤 1）。
- `kubectl` 命令行工具，并已配置为访问运行虚拟化的集群。
- 虚拟机磁盘的 `StorageClass`。
- 集群节点能够访问 HTTPS 服务器（CDI 从导入器 Pod 下载）。

操作步骤

1 获取并检查证书

如果尚未拥有 PEM 文件，请下载服务器提供的证书：

```
openssl s_client -showcerts -connect images.example.internal:443 </dev/null 2>/dev/null \
  | openssl x509 -outform PEM > server.crt
```

对于私有 CA，请使用服务器所有者提供的 CA 证书，而不是服务器证书。然后确认要放入镜像 URL 的主机名已列在 SAN 中：

```
openssl x509 -in server.crt -noout -subject -ext subjectAltName
# subject=CN=images.example.internal
# X509v3 Subject Alternative Name:
#     DNS:images.example.internal, IP Address:192.168.10.20
```

在 URL 中使用该列表中的主机名——本例中为 `https://images.example.internal/...` 或 `https://192.168.10.20/...`。如果缺少所需的地址，请重新签发包含该地址的证书；CDI 不会放宽主机名检查。

2 创建证书 ConfigMap

在 `DataVolume`（或虚拟机）所在的命名空间中创建 `ConfigMap`：

```
kubectl create configmap image-server-ca -n demo --from-file=tls.crt=
server.crt
```

对于 CA 链，请先将 PEM 文件合并为一个文件，再创建 `ConfigMap`（先放置 `cat intermediate.crt root.crt > chain.crt`，再放置 `--from-file=tls.crt=chain.crt`）。

3 创建直接导入镜像的虚拟机

从虚拟机的 `dataVolumeTemplates` 引用 `ConfigMap`。删除虚拟机也会删除导入的磁盘。

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: centos-https
  namespace: demo
spec:
  runStrategy: Always
  dataVolumeTemplates:
    - metadata:
        name: centos-https-rootdisk
      spec:
        source:
          http:
            url: https://images.example.internal/centos.qcow2
            certConfigMap: image-server-ca   # ConfigMap from Step 2, same namespace
            # secretRef: image-server-auth   # only if the server needs basic auth
        storage:
          storageClassName: sc-ceph-cluster-block
          resources:
            requests:
              storage: 20Gi                  # >= the image's virtual disk size
  template:
    spec:
      domain:
        devices:
          disks:
            - name: rootdisk
              disk:
                bus: virtio
        resources:
          requests:
            memory: 2Gi
      volumes:
        - name: rootdisk
          dataVolume:
            name: centos-https-rootdisk
```

```
kubectl apply -f centos-https.yaml
kubectl get dv -n demo -w
# NAME                     PHASE              PROGRESS   RESTARTS
# centos-https-rootdisk    ImportInProgress   37.51%
# centos-https-rootdisk    Succeeded          100.0%
kubectl get vmi -n demo centos-https
# NAME           AGE   PHASE     IP             NODENAME   READY
# centos-https   1m    Running   10.180.0.15    node-1     True
```

单独使用 `DataVolume` 也可以实现相同效果——将相同的 `source.http` 块放在 `spec.source` 对象的 `DataVolume` 下。

4 （可选）作为黄金镜像导入一次以供控制台使用

要允许控制台用户从此镜像创建虚拟机，请将其导入 `kube-public` 一次，并将其发布为 `DataSource`——即 **Image Instance**。

在控制台中：在 管理员 > 虚拟化 > 镜像 > 镜像实例 下，点击 添加镜像实例，选择 源 > 网络 > **HTTP**（或选择带有 HTTPS URL 的镜像源），输入 URL，并将证书粘贴到 **CA** 证书中。控制台会在浏览器中检查证书——主题、有效期、主题备用名称和指纹；如果 URL 的主机名不在 SAN 中，还会显示警告——并为你创建 `ConfigMap` `<name>-img-ca`（键 `tls.crt`）、`DataVolume` 和 `DataSource`。如果要将镜像保留在一个命名空间中，请在该命名空间的 容器平台 > 虚拟化 > 镜像 > 镜像实例 下添加镜像。

使用 **`kubectl`**：创建以下对象。这些标签会使 `DataSource` 显示为 **Image Instance**，并显示在 创建虚拟机表单的 置备方式 > **Image Instance** 选择器中；由于 `DataVolume` 位于 `kube-public` 中，因此会在该命名空间中创建 `ConfigMap`。

```
kubectl create configmap image-server-ca -n kube-public --from-file=t
ls.crt=server.crt
```

```yaml
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataVolume
metadata:
  name: centos-7-9
  namespace: kube-public
spec:
  source:
    http:
      url: https://images.example.internal/centos.qcow2
      certConfigMap: image-server-ca
  storage:
    storageClassName: sc-ceph-cluster-block
    resources:
      requests:
        storage: 20Gi
---
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataSource
metadata:
  name: centos-7-9
  namespace: kube-public
  labels:
    virtualization.cpaas.io/image-os-type: centos   # centos | ubuntu | ...; drives the OS filter
    virtualization.cpaas.io/image-os-arch: amd64    # amd64 | arm64
spec:
  source:
    pvc:
      name: centos-7-9
      namespace: kube-public
```

```shell
kubectl apply -f centos-7-9.yaml
kubectl get dv,datasource -n kube-public centos-7-9
# datavolume.cdi.kubevirt.io/centos-7-9   Succeeded   100.0%
# datasource.cdi.kubevirt.io/centos-7-9   ...  Ready
```

从 Image Instance 创建的虚拟机会克隆 `kube-public` 卷（`dataVolumeTemplates[].spec.sourceRef` 指向 `DataSource`）；HTTPS 服务器及其证书仅参与这一次导入。

故障排除

从 `DataVolume` 条件中查看导入器的错误——在 CDI 重试期间，阶段会保持为 `ImportInProgress`，而 `RESTARTS` 列会持续增长：

```
kubectl get dv <name> -n <namespace> -o jsonpath='{range .status.conditions[*]}{.type}={.status}: {.message}{"\n"}{end}'
```

消息	原因	修复
`x509: certificate signed by unknown authority`	`DataVolume` 上没有 `certConfigMap`，`ConfigMap` 中的证书不正确，或者私有 CA 证书链不完整。	将服务器证书（自签名）或完整的 CA 证书链放入 `ConfigMap`，并在设置 `certConfigMap` 后重新创建 `DataVolume`。
`x509: certificate is valid for a.example.com, not images.example.internal`	URL 中的主机名不在证书的 SAN 中。	使用 SAN 中的主机名，或重新签发证书。
`x509: cannot validate certificate for 192.168.10.20 because it doesn't contain any IP SANs`	URL 使用了 IP 地址，但证书中没有 IP SAN。	使用 SAN 中的 DNS 名称，或重新签发包含 `IP:` SAN 的证书。
导入器 Pod 卡在 `ContainerCreating`，事件为 `configmap "image-server-ca" not found`	缺少 `ConfigMap`，或它位于其他命名空间中。	在 `DataVolume` 所属的命名空间中创建它。
`x509: certificate has expired or is not yet valid`	证书的有效期未涵盖当前时间，或节点时钟不正确。	续订证书，或修正节点时间同步。

由于无法编辑 `DataVolume` `spec`，请通过删除并重新创建 `DataVolume`（或删除并重新创建通过 `dataVolumeTemplates` 拥有它的虚拟机）来应用修复。

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。

```yaml
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: "tru
e"
```

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。

```yaml
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: "tru
e"
      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 `kubectl` 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。

```yaml
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

此功能用于导出虚拟机的系统镜像并将其上传到对象存储，以便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到对象存储

创建虚拟机镜像

操作步骤

注意：以下所有操作都必须在虚拟机所在集群的控制平面节点上执行。

1 停止虚拟机

1. 前往管理员。
2. 在左侧导航栏中，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，页面将跳转到 Container Platform 中的虚拟机详情页。
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**：虚拟机所在命名空间的名称；将 *<namespace>* 替换为此名称。
- **VM_NAME**：需要导出系统镜像的虚拟机名称；将 *<vm_name>* 替换为此名称。
- **TTL_DURATION**：导出任务的生存时间，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
```

如果出现类似的回显信息，则表示创建成功。

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

4. 执行以下命令检查 vmexport 资源的状态。

```
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，并将其存储在 EXPORT_SERVER_IP 环境变量中。

```
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，该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到对象存储

将下载的镜像文件上传到对象存储。可使用任何 S3 工具进行上传，本文以 mc（minio-client）工具为例进行介绍。

1. 执行以下命令配置 mc 工具并连接到指定的 S3 存储服务。

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

参数说明：

- ENDPOINT：S3 存储服务的地址；将 `<ENDPOINT>` 替换为此地址。
- ACCESSKEY、SECRETKEY：用于身份验证的 S3 存储服务用户 ak 和 sk；相关信息请参阅 MinIO 对象存储↗。

2. 执行以下命令创建用于存储虚拟机镜像文件的存储桶。

```
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 上传到已创建的存储桶。

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

5 创建虚拟机镜像

1. 前往管理员。
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. 在镜像地址中填写 *<ENDPOINT>*/vmdisks/disk.img.gz，并将 <ENDPOINT> 替换为 S3 存储服务的地址。其他参数说明请参阅添加虚拟机镜像。
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	✓	×	

虚拟机

介绍

介绍

可启动卷

可启动卷

注意事项

查看可启动卷

添加可启动卷

默认实例类型和首选项

从可启动卷创建虚拟机

更新或删除可启动卷

操作指南

创建虚拟机

实例类型和偏好

虚拟机批量

先决条件
注意
创建虚拟机
使用 API

概念
查看实例类型和偏好
创建实例类型或偏好
更新或删除实例类型或偏好
虚拟机如何显示其实例类型
使用 YAML

操作步骤
使用 API

暂停和恢复虚

克隆虚拟机

前提条件
注意事项
克隆虚拟机
相关操作

串口控制台

注意事项
打开串口控制台

使用 VNC 登

操作步骤

管理密钥对

创建密钥对
更新密钥对
删除密钥对
使用 API

管理虚拟机

重置密码
更新密钥
更新规格
迁移到实例类型
实时迁移

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

导入的虚拟机

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（vTP

CPU 和内存热插拔

工作原理

前提条件

步骤 1：创建具有热插拔余量的 VM

步骤 2：热插拔 CPU

步骤 3：热插拔内存

缩减：哪些操作可在线完成，哪些需要重启

已验证的行为总结

故障排除

KubeVirt 节点维护

KubeVirt 如何处理排空

维护窗口之前

排空节点

完成维护

意外的节点故障

迁移

使用 Desche

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 之间迁移虚拟机

前提条件

准备迁移

使用 DataVolume 克隆进行冷迁移

在线存储迁移

配置虚拟机的高可用性

运行策略和驱逐策略

概述

术语表

组件概述

隔离和修复期间的事件流

从现有虚拟机

前提条件

操作步骤

操作步骤

故障排除

节点故障时的 VM 迁移和恢复

计划内维护

从意外的节点故障中恢复

热迁移错误消息与解决方案

启动和镜像导

读取来宾启动日

诊断卡住的镜像

捕获虚拟机内存

介绍

KubeVirt 提供 **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 VM 管理能力。在此基础上，**Alauda Virtualization** 进一步提升了易用性，通过提供 **Web** 控制台，让用户能够更加便捷地执行各种操作。

KubeVirt™ 是 The Linux Foundation 的商标。Alauda 是独立供应商。本产品与 The Linux Foundation 无关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处仅用于标识。

可启动卷

可启动卷是可直接使用的启动映像，可用于创建虚拟机。每个可启动卷都是由 PVC 提供支持的 KubeVirt `DataSource`（`cdi.kubevirt.io/v1beta1`），CDI 会从导入源或克隆源填充该 PVC。在控制台中，可启动卷作为映像实例进行管理 — 请参阅添加映像实例、管理映像实例和命名空间中的映像实例。本页面从 API 角度介绍可启动卷。

创建虚拟机并选择置备方法 > 映像实例时，选择器会列出共享 `kube-public` 命名空间中的可启动卷（平台“黄金映像”存放于此），以及虚拟机自身命名空间中的可启动卷。

目录

注意事项
查看可启动卷
添加可启动卷
默认实例类型和首选项
从可启动卷创建虚拟机
更新或删除可启动卷

注意事项

- 只有当可启动卷的 `DataSource` 带有下面所述的 `virtualization.cpaas.io/*` 标签契约（至少包括选择器用于筛选的操作系统和架构），其状态为 `Ready`，并且 — 对于 PVC 源

— 其 PVC 已绑定且导入成功时，才会提供该可启动卷用于创建虚拟机。不带这些标签的数据源仍会显示在映像实例列表中，但不会在映像实例选择器中提供。

- 控制台会将导入进度显示为映像实例状态：CDI 填充后端 PVC 时为正在导入，可供使用后为可用；请参阅状态。
- 本地上传不是控制台中的源；请改用 `virtctl` 上传（请参阅上传本地映像并创建虚拟机）。

查看可启动卷

可启动卷由 `DataSource` 对象提供支持，这些对象位于 `kube-public` 以及存放其自身对象的命名空间中。直接列出这些对象：

```
kubectl get datasource -A
```

添加可启动卷

可启动卷支持以下源；控制台中的名称如下：

控制台源	功能	`DataVolume` 源
网络 > **HTTP**（或 HTTP 映像源）	从 HTTP(S) URL 导入 qcow2/raw 映像	`http`
网络 > 注册表（或注册表映像源）	从容器注册表拉取容器磁盘	`registry`
网络 > 对象存储（**S3**）（仅限命名空间）	使用包含凭证的 Secret 从兼容 S3 的对象存储导入映像	`s3`
磁盘	克隆现有 PVC	`pvc`
卷快照	从 VolumeSnapshot 恢复	`snapshot`

> 注意：请在自己的命名空间中创建可启动卷。共享的 `kube-public` 命名空间包含由集群管理员发布的平台黄金映像 — 用户可以直接使用（克隆并从中启动）`kube-public` 映像，但在 `kube-public` 中创建或修改对象需要提升的平台权限。

添加可启动卷会创建两个对象：一个 `DataVolume`（CDI 将映像导入 PVC），以及一个指向该对象并携带标签契约的 `DataSource`。示例通过 HTTP 导入映像；控制台将 `DataVolume` 命名为 `<name>-img-<5 random characters>`，并使 `DataSource` 成为其所有者，而手写的一对对象可以共用一个名称，如下所示：

```yaml
# 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
      # certConfigMap: image-server-ca # for a server with a private CA
  storage:
    resources:
      requests:
        storage: 20Gi
    storageClassName: sc-ceph-cluster-block
    volumeMode: Block
    accessModes:
      - ReadWriteMany
---
# 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-instance: "true"
    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-ceph-cluster-block
    virtualization.cpaas.io/volume-mode: Block
    virtualization.cpaas.io/access-mode: ReadWriteMany
    virtualization.cpaas.io/source-type: address   # image-source | addre
ss | disk | volume-snapshot
spec:
  source:
    pvc:
      name: ubuntu-2204
      namespace: demo
```

选择存储类的 `StorageProfile` 所提供的卷模式和访问模式；在可用时，建议使用具有 **ReadWriteMany** 的 **Block**（请参阅存储设置）。

对于其他源类型，将 `DataVolume` 中的 `spec.source` 替换为以下选项之一。

注册表：

```
source:
  registry:
    url: docker://registry.example.com/library/fedora:40
```

S3，其中 Secret 包含 `accessKeyId` 和 `secretKey`：

```
source:
  s3:
    url: https://s3.example.com/images/fedora-40.qcow2
    secretRef: s3-images
```

克隆现有 PVC：

```
source:
  pvc:
    name: source-pvc
    namespace: demo
```

从 VolumeSnapshot 恢复：

```
source:
  snapshot:
    name: source-snapshot
    namespace: demo
```

应用并监视导入过程：

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

默认实例类型和首选项

可启动卷可以通过其 `DataSource` 上的标签，声明适合它的实例类型和首选项：

标签	值	缺少时
`instancetype.kubevirt.io/default-instancetype`	实例类型的名称	无默认实例类
`instancetype.kubevirt.io/default-instancetype-kind`	`VirtualMachineClusterInstancetype` 或 `VirtualMachineInstancetype`	`VirtualMac`
`instancetype.kubevirt.io/default-preference`	首选项的名称	无默认首选项
`instancetype.kubevirt.io/default-preference-kind`	`VirtualMachineClusterPreference` 或 `VirtualMachinePreference`	`VirtualMac`

创建虚拟机并选择此可启动卷作为镜像实例时，声明的实例类型和首选项会在模型区域中预先选定；你仍可以更改它们。表单还提供跟随镜像选项，该选项会将启动卷写入为引用源；平台会在创建虚拟机时读取一次这些标签（请参阅创建虚拟机）。命名空间范围的实例类型或首选项只能供其所属命名空间中的虚拟机使用。

CDI 会将这些标签从其背后的对象复制到 **`DataSource`** 上：这些对象包括 PVC 和同名的 `DataVolume`，或卷快照；如果 `DataVolume` 没有任何这些标签，则从导入或克隆它所用的卷复制。因此，仅在 `DataSource` 上设置的标签，只要这些对象之一声明了其他值，就会被覆盖。在控制台中，对镜像实例使用更新默认规范，即可将所选值写入所有这些对象（请参阅默认规范）。使用 `kubectl` 时，也要为 PVC、`DataVolume` 和 `DataSource` 添加标签：

```
for obj in pvc/centos-stream9 datavolume/centos-stream9 datasource/centos
-stream9; do
  kubectl label -n kube-public "$obj" --overwrite \
    instancetype.kubevirt.io/default-instancetype=u1.medium \
    instancetype.kubevirt.io/default-preference=centos.stream9
done
```

或者在创建时，在 `DataSource`（以及 `DataVolume`）中声明这些标签：

```yaml
apiVersion: cdi.kubevirt.io/v1beta1
kind: DataSource
metadata:
  name: centos-stream9
  namespace: kube-public
  labels:
    instancetype.kubevirt.io/default-instancetype: u1.medium
    instancetype.kubevirt.io/default-preference: centos.stream9
spec:
  source:
    pvc:
      name: centos-stream9
      namespace: kube-public
```

之后更改标签只会影响随后创建的虚拟机；现有虚拟机仍使用创建时所用的对象。

从可启动卷创建虚拟机

虚拟机将在其自身的命名空间中创建，并从 `kube-public` 或同一命名空间中的可启动卷克隆其根磁盘。在底层，虚拟机通过 `dataVolumeTemplates` 条目引用数据源 — CDI 会将可启动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: kube-public for an
          # administrator-published golden image, or the virtual machin
e's own.
          namespace: demo
        storage:
          resources:
            requests:
              storage: 20Gi
          storageClassName: sc-ceph-cluster-block
          volumeMode: Block
          accessModes:
            - ReadWriteMany   # the console picks it from the StorageProf
ile
```

克隆源命名空间仅限于虚拟机的命名空间和 `kube-public`。从另一个非公开命名空间克隆可能需要额外的 CDI 克隆权限。

更新或删除可启动卷

在控制台中，对镜像实例使用重新导入和删除；请参阅管理镜像实例。删除由控制台添加的镜像实例会移除其 `DataSource`，而导入 `DataVolume`、其 PVC 及其 CA 证书 `ConfigMap` 会根据所有者引用一并删除。

对于手动编写的可启动卷，请删除 `DataSource` 和 `DataVolume`。后备 PVC 由 `DataVolume`（通过 `ownerReferences`）拥有，因此在删除 `DataVolume` 时会自动进行垃圾回收 — 请勿在 `DataVolume` 仍存在时删除 PVC，否则 CDI 会重新预配它：

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

如果 CDI 已在成功导入后对 `DataVolume` 进行了垃圾回收，则 `kubectl delete datavolume` 会返回 `NotFound`，而 PVC 将不再具有所有者 — 此时请直接删除它：

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

操作指南

创建虚拟机

先决条件

注意

创建虚拟机

使用 API

克隆虚拟机

前提条件

注意事项

克隆虚拟机

相关操作

管理密钥对

创建密钥对

更新密钥对

删除密钥对

使用 API

虚拟机的快速定位

实例类型和偏好

概念

查看实例类型和偏好

创建实例类型或偏好

更新或删除实例类型或偏好

虚拟机如何显示其实例类型

串口控制台

注意事项

打开串口控制台

管理虚拟机

重置密码

更新密钥

更新规格

迁移到实例类型

实时迁移

虚拟机批量操

操作步骤

使用 API

暂停和恢复虚

使用 VNC 登

操作步骤

监控和告警

监控

告警

使用 API

前提条件

操作步骤

使用 API

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。`VirtualMachine` 是期望状态对象（类似于 Deployment）；启动后，KubeVirt 会生成一个正在运行的 `VirtualMachineInstance`（类似于 Pod）。有关本页使用的术语，请参阅概念和术语表。

你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。

使用实例类型和首选项为新虚拟机设置规格。实例类型声明一次 CPU 和内存，所有引用该实例类型的虚拟机均可共享；首选项则应用客户机操作系统最适用的设备和固件设置。也可以直接在虚拟机（机器模型）中设置 CPU 和内存。请参阅实例类型和首选项。

目录

先决条件
注意
创建虚拟机
操作步骤
相关操作
使用 API
推荐：引用实例类型和首选项
跟随镜像
替代方案：在虚拟机中设置大小（机器模型）

先决条件

- 使用镜像创建虚拟机之前，请向平台管理员确认以下事项：
  - 目标集群是平台在非 Alauda OS 上创建的业务集群，并且已部署 Kubevirt 组件。
  - 目标节点必须是已启用虚拟化的物理节点。
  - 已向平台添加镜像源并将其分配给项目，或者 `kube-public` 中或命名空间中已有镜像实例（请参阅镜像）。
- 如果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，并准备必要的环境。
  2. 准备所需的物理 GPU，并为虚拟机的物理 GPU 直通启用相关功能。

注意

使用 Windows 虚拟机时，仅支持使用虚拟机镜像中设置的用户名/密码登录。请提前联系平台管理员获取此信息。

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，你也可以切换到 YAML 格式进行操作。

1. 进入容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟机。
3. 单击创建虚拟机。
4. 在基本信息区域中，填写虚拟机的名称和显示名称，并设置标签或注解。

参数	说明
标签	用于选择对象并查找符合特定条件的对象集合。必须是键值对，例如：*app.kubernetes.io/name: hello-app*。
注解	用于向开发和运维团队提供任何信息。必须是键值对，例如：*cpaas.io/maintainer: kim*。

5. 在镜像区域中，选择虚拟机启动所用的镜像。

参数	说明
CPU 架构	镜像的架构；只能选择与集群架构相同的镜像。
置备方式	镜像源：由平台管理员分配给项目的公共镜像；系统盘从其地址导入。镜像实例：镜像实例（可引导卷），虚拟机将从中克隆系统盘；列表中会显示 `kube-public` 的镜像实例以及虚拟机所在命名空间的镜像实例，后者会标注命名空间名称。
选择镜像	操作系统和镜像。镜像实例或镜像源可以声明默认实例类型和偏好设置；此时系统会在型号区域中预选这些设置，你仍可进行修改。如果集群中不存在声明的名称，则不会预选，并会在表单中提示。对于地址为 `https://` 的镜像源，表单会提醒你：由私有 CA 保护的镜像必须从包含 CA 证书的镜像实例创建。

6. 在型号区域中，选择虚拟机的规格配置方式。

参数	说明
规格配置	实例类型（默认，推荐）：虚拟机引用实例类型，本身不包含 CPU 或内存。机器型号：将 CPU 和内存写入虚拟机。两者不能同时使用。
实例类型	在实例类型配置路径下显示。列出当前命名空间（此命名空间）和集群（集群）的实例类型；每个选项会显示其 vCPU 数量、内存，以及需要时的 hugepage 大小，常用类型会标记为产品提供。需要 hugepage 的实例类型必须使用提供相应 hugepage 的节点。

参数	说明
遵循镜像设置	当所选镜像实例声明了默认实例类型（或偏好设置）时提供。虚拟机将使用镜像声明的对象，而不是由你选择；创建虚拟机时，系统会读取该对象并将其写入虚拟机。如果镜像未声明对象，则会拒绝创建。
型号	在机器型号配置路径下显示。选择推荐的使用场景，或自定义 CPU 和内存；请求值和限制值将设置为相同值。
偏好设置	两种配置路径均可选，且建议设置。列出当前命名空间和集群的偏好设置；偏好设置会补充虚拟机未设置的设备、固件和时钟设置。
物理 **GPU**（**Alpha**）	选择物理 GPU 的型号；每台虚拟机只能分配一个物理 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，从而达到与在物理机上直接运行相当的图形性能，避免虚拟图形适配器造成的性能瓶颈并提升整体性能。

7. 在存储区域中，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：系统会自动创建 VirtIO 类型的 rootfs 系统盘，用于存储操作系统和数据。 • 数据盘：单击此项可添加多个数据盘，用于持久化存储数据。默认使用 VirtIO 设备。 注意：数据盘名称不得与现有磁盘名称重复。
卷模式	• 文件系统：将磁盘挂载为文件目录。 • 块设备：将磁盘挂载为块设备。

参数	说明
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。你需要指定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。 每个磁盘的访问模式遵循存储类的 `StorageProfile`：如果配置文件针对相应卷模式提供 **ReadWriteMany**，则使用该模式，例如 Ceph RBD 的块设备或 CephFS 的文件系统，以便虚拟机能够实时迁移；否则使用配置文件提供的访问模式。没有配置文件时，Ceph RBD 块设备使用 **ReadWriteMany**，其他情况使用 **ReadWriteOnce**。
容量	虚拟机存储所需的容量；系统盘的最小容量为 20 G。
随虚拟机删除	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。

8. 在网络区域中，参考以下说明配置相关信息。

参数	说明
IP 地址	• 默认设置为动态 **(DHCP)**；虚拟机启动时会动态分配 IP，停止时会释放 IP。 • 如果绑定静态 **IP**，虚拟机即使重启也会始终使用该 IP 地址。如果当前项目没有可用 IP，请先适当释放 IP。
网络模式	• 桥接：虚拟机与容器组共用同一 IP 地址，并通过此 IP 地址与外部通信。 • **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对外暴露的端口，例如 SSH 服务端口 22；如果不填写开放端口，则表示开放所有端口。
辅助网卡	根据需要添加辅助网卡。 注意：

参数	说明
	• 如果需要辅助网卡功能，或者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。

9. 在初始化设置区域中，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。此方法无需验证密码，建议使用密钥登录虚拟机。 • 你可以使用平台中已有的密钥，也可以立即创建新密钥；所有相关密钥均可在虚拟化 **>** 密钥对页面中查看。 • 只有持有私钥的人员才能通过 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。如果密钥泄露，可以及时撤销关联的密钥以减少损失。 • SSH 密钥的公钥以保密形式存储在平台中；平台不会存储你的私钥，请自行妥善保管。 • 有关 root 用户密码的信息，请参阅相应操作系统文档。
密码	使用操作系统用户名和密码进行登录验证，之后仍可切换为密钥方式。 • 该用户仅为初始账户；虚拟机成功创建后，你也可以在虚拟机中创建其他操作系统用户用于登录。 • 平台会加密并存储你的 root 用户密码，你之后将无法再次查看其明文，请自行妥善保管。
立即启动	默认启用。启用此选项后，虚拟机会在创建完成后立即启动；否则仅创建虚拟机。

10. （可选）在高级配置区域中，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。 有关相关参数的说明。
节点亲和性	• 首选：系统会尽可能将虚拟机调度到满足亲和性要求的节点。系统将结合亲和性权重和其他调度要求（例如计算资源要求），确定能够运行虚拟机的节点。 • 必需：系统只会将虚拟机调度到完全满足亲和性要求的节点。

11. 确认信息无误后，单击创建。
    等待虚拟机状态从创建中变为运行中。

相关操作

你可以根据需要，单击列表页右侧的⋮图标或详情页右上角的操作，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：

- 仅当虚拟机处于异常、未知或已停止状态时，才能执行更新操作。
- 更新时不支持显示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；你可以根据需要启用。

使用 API

虚拟机是一个 `VirtualMachine` （ `kubevirt.io/v1` ）。

推荐：引用实例类型和首选项

下面的虚拟机从可启动卷（`DataSource`）启动，从集群实例类型 `u1.medium` 获取其大小，从集群首选项 `centos.stream9` 获取其设备设置，连接 Pod 网络，并通过 cloud-init 注入 SSH 密钥：

```yaml
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: web-01
  namespace: demo
spec:
  running: false           # set true to start on creation
  instancetype:            # the size: guest CPU and memory
    kind: VirtualMachineClusterInstancetype
    name: u1.medium
  preference:              # optional: preferred devices, firmware and cl
ock
    kind: VirtualMachineClusterPreference
    name: centos.stream9
  dataVolumeTemplates:
    - metadata:
        name: web-01-rootfs
      spec:
        sourceRef:          # clone an existing bootable volume (DataSour
ce)
          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:
        # No resources, cpu or memory.guest here: they come from the inst
ance type.
        devices:
          disks:
            - name: rootfs
              disk: {}      # the preference chooses the disk bus
```

```
        - name: cloudinit
          disk: {}
      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":tru
e}}'
```

- 要引用命名空间范围的对象，请将 `kind` 设置为 `VirtualMachineInstancetype` 或 `VirtualMachinePreference`；该对象必须位于虚拟机所在的命名空间中。不带 `kind` 的引用表示集群范围的 kind。
- 引用实例类型的虚拟机不得设置 `spec.template.spec.domain.resources` CPU 或内存、`domain.cpu.sockets` / `cores` / `threads`、`domain.memory.guest` 或实例类型设置的任何其他配置；此类虚拟机会被拒绝。
- 首选项只会填充虚拟机未设置的配置：例如，显式设置了 `bus` 的磁盘会保留其总线。

跟随镜像

当可启动卷声明了默认实例类型和首选项（请参阅可启动卷）时，虚拟机可以从该卷获取它们，而无需指定对象名称。跟随镜像会写入以下内容：

```
spec:
  instancetype:
    inferFromVolume: rootfs            # the boot volume
    inferFromVolumeFailurePolicy: Reject
  preference:
    inferFromVolume: rootfs
    inferFromVolumeFailurePolicy: Reject
```

平台会在创建虚拟机时从卷中读取一次默认值，并将其指定的对象存储在虚拟机中 — `spec.instancetype.name` 和 `spec.preference.name` 会替代 `inferFromVolume` 字段。之后更改卷的标签不会影响现有虚拟机。使用 `Reject` 策略时，如果卷未声明默认值，则会拒绝创建；使用 `Ignore` 策略时，创建的虚拟机不会包含该实例类型或首选项。

替代方案：在虚拟机中设置大小（机器模型）

不使用实例类型时，大小会写入模板。ACP 使用资源限制模型，因此 `requests` 和 `limits` 包含相同的值。仍然可以引用首选项：

```
spec:
  preference:
    kind: VirtualMachineClusterPreference
    name: centos.stream9
  template:
    spec:
      domain:
        cpu:
          cores: 1
        resources:          # requests == limits
          requests:
            memory: 2Gi
          limits:
            memory: 2Gi
```

启动磁盘由 `dataVolumeTemplates` 条目配置，该条目克隆所引用的可启动卷。要稍后更改虚拟机的大小，或将虚拟机从机器模型迁移到实例类型，请参阅管理虚拟机。

实例类型和偏好

实例类型是声明虚拟机大小的可复用对象：包括客户机 CPU 数量和内存，以及可选的专用 CPU、巨页、热插拔上限、GPU、主机设备和节点放置。偏好是声明客户机操作系统偏好的虚拟硬件配置的可复用对象，例如磁盘总线、网络接口模型、固件和时钟，以及此类客户机所需的最小资源。

引用实例类型和偏好是确定新虚拟机大小的推荐方式：

- 大小只需定义一次即可共享。引用同一实例类型的每个虚拟机都具有相同的大小，而调整虚拟机大小意味着将其指向另一个实例类型。
- 偏好会应用操作系统运行效果最佳的设置，无需编辑每个虚拟机。
- 虚拟机会继续使用其启动时所用的实例类型副本，因此编辑共享实例类型不会意外更改正在运行的工作负载。

仍然可以直接在虚拟机中设置 CPU 和内存（机器模型）；请参阅创建虚拟机。要将现有虚拟机从机器模型迁移到实例类型，请参阅迁移到实例类型。

目录

概念

常见实例类型

查看实例类型和偏好

创建实例类型或偏好

更新或删除实例类型或偏好

虚拟机如何显示其实例类型

使用 YAML

概念

类型	作用域	管理位置
`VirtualMachineClusterInstancetype`	集群	管理员 > 虚拟化 > 规格
`VirtualMachineClusterPreference`	集群	管理员 > 虚拟化 > 规格
`VirtualMachineInstancetype`	命名空间	容器平台 > 虚拟化 > 规格
`VirtualMachinePreference`	命名空间	容器平台 > 虚拟化 > 规格

这四种类型都属于 API 组 `instancetype.kubevirt.io/v1beta1`。

- 作用域：虚拟机可以从任何命名空间引用集群作用域对象，但只能从其自身命名空间引用命名空间对象。省略 `kind` 的引用表示集群作用域类型。
- 修订版本：虚拟机首次解析其实例类型或偏好时，平台会在虚拟机所在的命名空间中存储该对象的副本（一个 `ControllerRevision`），并将其记录在虚拟机的状态中。虚拟机会继续使用该副本：之后更新或删除对象不会更改已经解析该对象的虚拟机。将虚拟机指向其他对象会记录一个新的副本。
- 无法覆盖实例类型设置：引用实例类型的虚拟机不得设置自身的 CPU 数量或内存（`resources` CPU 和内存、`domain.cpu` 套接字 / 核心 / 线程、`domain.memory.guest`），也不得设置实例类型同时设置的任何其他配置。此类虚拟机会被拒绝。
- 由产品提供：平台会在每个虚拟化集群上安装一组常见实例类型和偏好。这些对象标记为由产品提供，由虚拟化 operator 维护，且无法更新或删除。对于其他大小和设置，请创建自己的对象。

常见实例类型

常见实例类型的名称为 `<series>.<size>`，例如 `u1.medium`。名称以 `1gi` 结尾的大小使用 1Gi 巨页，而不是 2Mi 巨页。

系列	适用场景	特征
`u1`	通用工作负载	仅客户机 CPU 和内存
`o1`	使用超额分配内存的通用工作负载	内存超额分配 50%
`d1`	一致且可预测的性能	专用 CPU、隔离的模拟器线程
`cx1`	计算密集型工作负载	专用 CPU、隔离的模拟器线程、映射到主机的客户机 NUMA、巨页
`m1`	内存密集型工作负载	巨页
`n1`	网络密集型（DPDK）工作负载	专用 CPU、隔离的模拟器线程、1Gi 巨页
`rt1`	实时工作负载	专用 CPU、隔离的模拟器线程、映射到主机的客户机 NUMA、实时、1Gi 巨页

常见偏好以客户机操作系统命名，例如 `centos.stream9`、`ubuntu`、`rhel.9` 或 `windows.2k22.virtio`。

WARNING

设置巨页大小的实例类型（`cx1`、`m1`、`n1` 和 `rt1` 系列，或您自己的实例类型）需要节点预先分配该大小的巨页，且单个节点上的巨页至少要满足客户机内存需求。在没有此类巨页的集群上，引用此类实例类型的虚拟机可以创建但无法启动：它会保持异常状态，其 virt-launcher pod 会报告 `Insufficient hugepages-2Mi`（或 `hugepages-1Gi`）。列表的巨页大小列和实例类型选择器会显示哪些实例类型需要巨页。使用以下命令检查节点提供的巨页：

```
kubectl get nodes -o custom-columns='NODE:.metadata.name,HUGEPAGES_2Mi:.status.allocatable.hugepages-2Mi,HUGEPAGES_1Gi:.status.allocatable.hugepages-1Gi'
```

查看实例类型和偏好

1. 进入 **Administrator** 视图以管理集群范围的对象，或进入 **Container Platform** 视图以管理当前命名空间中的对象。
2. 在左侧导航栏中，点击 **Virtualization** > **Specifications**。
3. 选择 **Instance Type** 或 **Preference** 选项卡。

列	描述
Source	对于常见对象为 **Provided by product**，否则为 **Custom**。
Guest CPU / **Guest Memory**	实例类型提供给虚拟机的大小。
Hugepage Size	实例类型所需的巨页大小；不使用巨页时为 `-`。
Preferred Settings	偏好声明的设置数量。

4. 点击名称以打开详细信息页面：
   - **Instance Type Settings** / **Preference Settings** 仅显示对象声明的设置。偏好中的每个首选设备、固件、时钟和功能设置各占一行，例如 **Preferred Disk Bus** 和 **Preferred Interface Model**。
   - **Virtual Machines** 选项卡列出引用该对象的虚拟机。
   - **YAML** 选项卡显示完整对象。

创建实例类型或偏好

1. 在 **Specifications** 页面上，选择 **Instance Type** 或 **Preference** 选项卡，然后点击 **Create Instance Type** 或 **Create Preference**。
2. 填写表单，或切换到 **YAML**。表单中没有的设置可以在 YAML 中设置；请参见使用 YAML。实例类型：

参数	描述
Name	创建后无法更改。
Guest CPU	必需。客户机可见的 vCPU 数量。

参数	描述
Guest Memory	必需。大小可以是 `8Gi` 等数量。
Dedicated CPU Placement	将每个 vCPU 固定到节点的一个物理 CPU。
Isolated Emulator Thread	在其自身的物理 CPU 上运行模拟器线程；与专用 CPU 配合使用。
Hotplug Socket Ceiling	正在运行的虚拟机可以热插拔到的 CPU 插槽数上限。
Hotplug Memory Ceiling	正在运行的虚拟机可以热插拔到的内存上限。
Hugepage Size	使用此大小的巨页作为客户机内存的后备（例如 `2Mi` 或 `1Gi`）；节点必须提供这些巨页。
GPUs / Host Devices	传递给虚拟机的设备，每个设备包含一个名称和一个设备名称。
Node Selector	节点必须具有的标签，虚拟机才能在该节点上运行。
Labels / Annotations	实例类型对象的元数据。

偏好：

参数	描述
Preferred CPU Topology	客户机 CPU 数量的布局方式，例如按插槽或按核心布局。
Preferred Disk Bus / Preferred CD-ROM Bus	例如 `virtio`、`sata` 或 `scsi`。
Preferred Interface Model	例如 `virtio` 或 `e1000e`。
Preferred Input Bus / Preferred Input Type	例如 `usb` 和 `tablet`。

参数	描述
Preferred TPM	附加虚拟 TPM。
Preferred Graphics Device / **Preferred Serial Console**	是否附加图形设备和串行控制台。
Preferred EFI Firmware / **Preferred Secure Boot** / **Preferred BIOS Firmware**	引导固件。安全启动需要 EFI 固件和 SMM 功能。
Preferred SMM Feature	启用 System Management Mode。
Preferred Machine Type	例如 `q35`。
Preferred Storage Class	虚拟机卷的存储类。
Preferred Architecture	例如 `amd64`。
Required Guest CPU / **Required Guest Memory** / **Required Architecture**	引用此偏好的虚拟机必须提供的最小值；较小的虚拟机会被拒绝。

3. 点击 **Create**。

偏好仅在虚拟机未设置相应值的位置应用其设置；虚拟机自行设置的值会保留。

更新或删除实例类型或偏好

- 要更新，请在列表中点击⋮> **Update**，或在详情页点击 **Update**。名称无法更改。如果虚拟机引用了该对象，系统会在保存前列出这些虚拟机：每个虚拟机都会保留其状态中记录的副本，之后解析该对象的虚拟机将应用此更改。
- 要删除，请点击⋮> **Delete**。确认对话框会列出引用该对象的虚拟机。已启动的虚拟机会保留其记录的副本；尚未启动的虚拟机在引用现有对象之前无法启动。
- 标记为 **Provided by product** 的对象仅提供 **View YAML**。

虚拟机如何显示其实例类型

- 在虚拟机列表中，**Resources** 列显示 **Instance Type** 及其名称、**Machine Model** 或 **Sizing undetermined**。
- 在虚拟机详情页中，**Instance Type** 和 **Preference** 行会显示请求的对象、解析到的对象及其记录的修订版本；如果无法解析引用，还会显示原因，例如 `FailedFindInstancetype`。

使用 YAML

集群范围的实例类型：

```yaml
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachineClusterInstancetype
metadata:
  name: standard.4c8g
spec:
  cpu:
    guest: 4          # required: vCPUs of the guest
  memory:
    guest: 8Gi        # required: memory of the guest
```

具有专用 CPU 和 1Gi 大页的命名空间实例类型（节点必须提供 1Gi 大页）：

```yaml
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachineInstancetype
metadata:
  name: db.8c32g
  namespace: demo
spec:
  cpu:
    guest: 8
    dedicatedCPUPlacement: true
    isolateEmulatorThread: true
  memory:
    guest: 32Gi
    hugepages:
      pageSize: 1Gi
```

具有热插拔上限的实例类型，因此运行中的虚拟机可以扩展到 8 个 vCPU 和 32Gi 内存：

```yaml
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachineClusterInstancetype
metadata:
  name: elastic.2c8g
spec:
  cpu:
    guest: 2
    maxSockets: 8
  memory:
    guest: 8Gi
    maxGuest: 32Gi
```

面向使用 virtio 设备的 Linux 客户机的集群范围偏好：

```yaml
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachineClusterPreference
metadata:
  name: linux-virtio
spec:
  devices:
    preferredDiskBus: virtio
    preferredInterfaceModel: virtio
  requirements:       # the smallest virtual machine this guest runs on
    cpu:
      guest: 1
    memory:
      guest: 2Gi
```

面向启用 UEFI 安全启动并具有虚拟 TPM 的 Windows 客户机的命名空间偏好：

```yaml
apiVersion: instancetype.kubevirt.io/v1beta1
kind: VirtualMachinePreference
metadata:
  name: windows-uefi
  namespace: demo
spec:
  cpu:
    preferredCPUTopology: sockets
  devices:
    preferredDiskBus: sata
    preferredInterfaceModel: e1000e
    preferredTPM:
      persistent: true
  firmware:
    preferredUseEfi: true
    preferredUseSecureBoot: true
  features:
    preferredSmm: {}
```

应用并列出这些对象：

```
kubectl apply -f instancetype.yaml
kubectl get virtualmachineclusterinstancetypes
kubectl get virtualmachineinstancetypes -n demo
kubectl get virtualmachineclusterpreferences
kubectl get virtualmachinepreferences -n demo
```

查找引用实例类型的虚拟机，以及虚拟机解析到的副本：

```
kubectl get vm -A -o jsonpath='{range .items[?(@.spec.instancetype.name=
="standard.4c8g")]}{.metadata.namespace}/{.metadata.name}{"\n"}{end}'

kubectl get vm web-01 -n demo -o jsonpath='{.status.instancetypeRef}'
# {"controllerRevisionRef":{"name":"web-01-standard.4c8g-v1beta1-<uid>-
1"},"kind":"VirtualMachineClusterInstancetype","name":"standard.4c8g"}
```

表单涵盖常见设置。以下设置只能在 YAML 中设置：

类型	设置
实例类型	`spec.annotations`、`spec.cpu.model`、`spec.cpu.numa`、`spec.cpu.realtime`、`spec.memory.overcommitPercent`、`spec.ioThreads`、`spec.ioThreadsPolicy`、`spec.launchSecurity`、`spec.schedulerName`
偏好	`spec.annotations`、`spec.clock`、`spec.cpu.preferredCPUFeatures`、`spec.cpu.spreadOptions`、`spec.features`（`preferredSmm` 除外）、`spec.firmware.preferredEfi`、`spec.firmware.preferredUseBiosSerial`、其他 `spec.devices` 设置（例如 `preferredDiskDedicatedIoThread`、`preferredBlockMultiQueue`、`preferredRng`）、`spec.preferSpreadSocketToCoreRatio`、`spec.preferredSubdomain`、`spec.preferredTerminationGracePeriodSeconds`

要在虚拟机中使用实例类型和偏好，请参阅创建虚拟机。

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":tru
e}}'    # or: virtctl start web-01 -n demo
# Stop
kubectl patch vm web-01 -n demo --type merge -p '{"spec":{"running":fals
e}}'   # 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：

```yaml
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 inst
ead.
  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 in
stead of
  # inheriting the source's. Clone filters are whitelist-based: a bare pa
ttern
  # 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`：

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

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/virtualmachineinsta
nces/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。

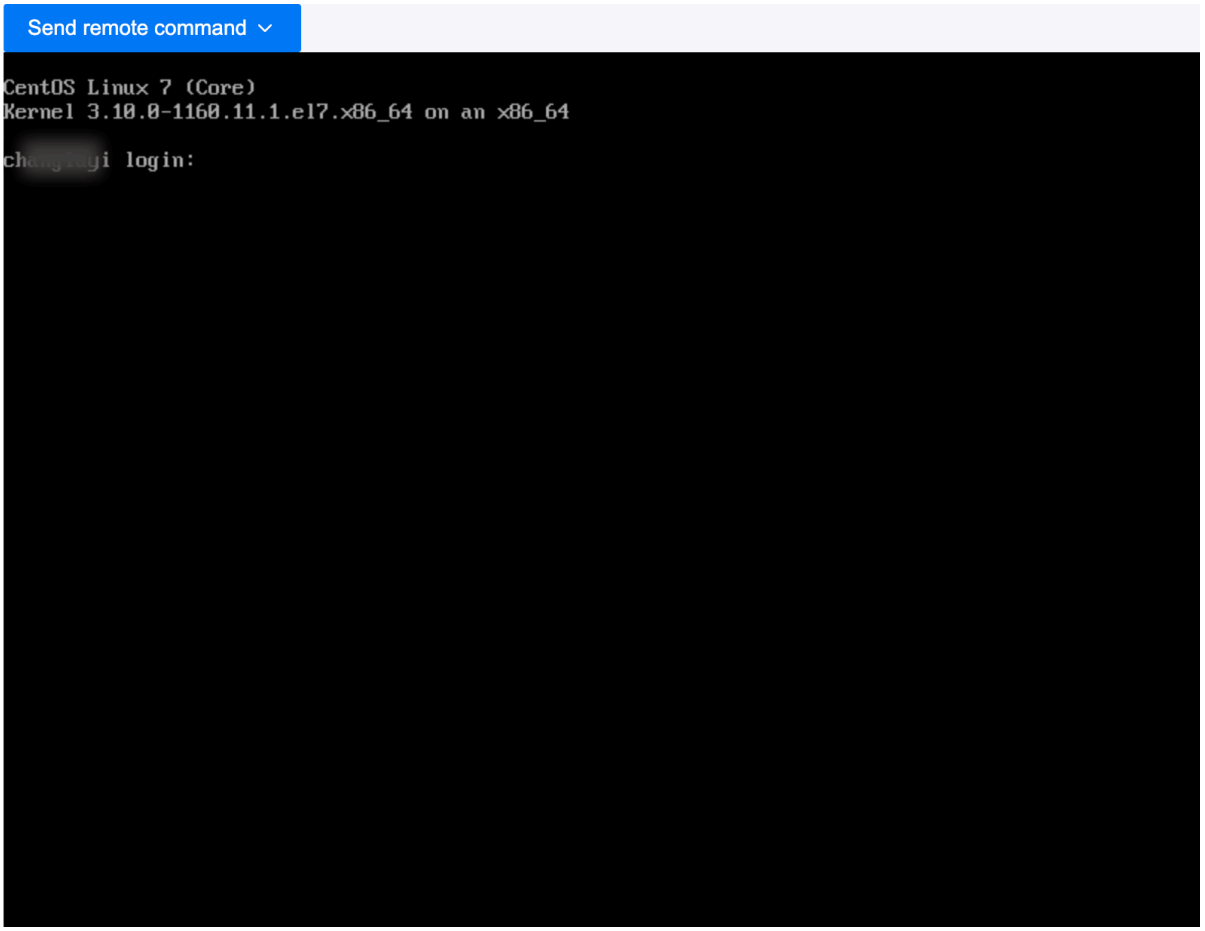
目录

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/virtualmachineinsta
nces/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：

```yaml
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"}'   # kuberne
tes.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

以下操作可从虚拟机列表（行⋮菜单）和详情页面中执行。以下操作作为 API/CLI 参考进行说明：克隆、暂停和恢复以及串行控制台。每项操作也都有对应的 API 调用 — 请参阅下方的使用 **API**。

目录

- 重置密码
  - 操作步骤
- 更新密钥
  - 操作步骤
- 更新规格
- 迁移到实例类型
- 实时迁移
- 更新 NAT 网络配置
  - 操作步骤
- 更新标签和注释
- 添加服务
- 重新安装操作系统
  - 操作步骤
- 配置 IP
  - 操作步骤
- 使用 API
- 导入的虚拟机

重置密码

重置 **root** 用户密码。使用密码登录时，此密码也会作为虚拟机的登录密码。

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. 找到虚拟机，然后选择⋮> 重置密码。
4. 设置密码。
5. 单击重置。
   注意：请妥善保管您的密码。为确保环境安全，平台会加密并存储您的密码，之后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. 访问容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟机。
3. 找到虚拟机，然后选择⋮> 更新密钥。
4. 选择一个或多个关联密钥，或选择创建密钥。
5. 选择是否立即重启；更新密钥需要重启虚拟机才能生效。
6. 单击更新。

更新规格

更改虚拟机的大小，或在两种规格路径之间切换：虚拟机所引用的实例类型，或写入虚拟机的机器模型。

1. 访问容器平台。
2. 在左侧导航栏中，单击虚拟化 > 虚拟机。
3. 找到目标虚拟机，然后单击⋮> 更新规格。对话框会打开虚拟机当前使用的规格路径。
4. 选择大小：

参数	描述
规格路径	实例类型或机器模型。切换路径会在同一次更新中移除之前路径的字段。
实例类型	要引用的实例类型，可以来自当前命名空间或集群。虚拟机的偏好设置会予以保留。
模型	推荐的使用场景，或自定义 CPU 和内存。对于已迁移到实例类型的虚拟机，会预选迁移前的大小，并按原样恢复。

5. 选择是否立即重启。
6. 单击更新。

更改生效时：

- 机器模型：新的 CPU 和内存在虚拟机重启后生效。
- 运行中虚拟机的实例类型：引用会立即更改，平台会尝试在不重启的情况下应用新大小。每次热插拔都通过实时迁移执行，因此仅当满足以下所有条件时，新大小才会实时应用：
  - 集群会实时推出虚拟机更改（KubeVirt `vmRolloutStrategy` 为 `LiveUpdate`）。
  - 虚拟机可以进行实时迁移（其 `LiveMigratable` 条件为 `True`）。
  - 新的 CPU 数量和内存均未超出运行中实例启动时设置的热插拔上限。这些上限是其实例类型的 `maxSockets` 和 `maxGuest`。当实例类型未设置上限时，平台会在实例启动时计算上限 — 默认是初始套接字数量和内存的四倍；其比例和绝对上限在 `HyperConverged` 资源的 `spec.liveUpdateConfiguration`（`maxHotplugRatio`、`maxCpuSockets`、`maxGuest`）中进行集群范围的设置。上限等于大小的实例类型没有增长空间。运行中实例的上限位于其 VMI 上；请参阅下面的命令。
  - CPU 数量只能增加；较小的 CPU 数量始终需要重启。

- 对于内存：虚拟机运行在 x86_64 上，客户机操作系统支持内存热插拔（virtio-mem），并且客户机内存至少为 1Gi。

无法实时应用的更改 — 当前三个条件中的任一条件不满足时，所有更改都无法实时应用 — 会在下次重启时生效，在此之前虚拟机会一直带有 `RestartRequired` 条件。有关客户机侧要求，请参阅CPU 和内存热插拔。

- 检查运行中的虚拟机是否可以进行实时迁移、其运行中实例的热插拔上限，以及是否仍需要重启：

```
kubectl get vmi web-01 -n demo -o jsonpath='{.status.conditions[?(@.type=="LiveMigratable")]}'
kubectl get vmi web-01 -n demo -o jsonpath='{.spec.domain.cpu.maxSockets}{" "}{.spec.domain.memory.maxGuest}{"\n"}'
kubectl get vm web-01 -n demo -o jsonpath='{.status.conditions[?(@.type=="RestartRequired")]}'
```

不带平台标签的虚拟机 — 通常是从其他平台原样导入的虚拟机 — 通过其 YAML 进行更新；请参阅导入的虚拟机。

迁移到实例类型

将现有虚拟机从 Machine Model 迁移到 Instance Type，使其大小与使用实例类型创建的虚拟机一致。虚拟机列表的 **Resources** 列显示每个虚拟机的大小设置路径：**Machine Model**、**Instance Type** 及其名称，或 **Sizing undetermined**。

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization** > **Virtual Machines**。
3. 找到虚拟机并点击⋮> **Migrate to Instance Type**（也可在详情页的 **Actions** 菜单中找到）。
4. 查看 **Current Size** — 即虚拟机当前的客户机 CPU 数量和内存 — 并选择一个 **Instance Type**：
   - 大小完全相同的实例类型会预先选中，并标记为 **Same size**。
   - 如果没有大小相同的实例类型，系统会列出最接近的实例类型及其差异；在你选择一个之前，不会选中任何实例类型。选择实例类型会更改虚拟机的大小。

5. （可选）选择 **Also reference a preference**，然后选择一项 preference。preference 会填充虚拟机未设置的设备设置，这可能会更改客户机看到的内容 — 例如其网络接口的模型。
6. 查看 **What changes for the guest**，然后点击 **Migrate**：

   - 当虚拟机将 CPU 定义为核心或线程且未引用 preference 时，CPU 数量会改为套接字数量；按核心授权的软件可能会检测到此变化。
   - 所选 preference 填充的设置会列出。
   - 运行中的虚拟机会一直保持当前大小，直到重启。

迁移操作只会移除与实例类型冲突的大小字段，并保留其他所有设置。虚拟机原有的大小会记录在虚拟机上（位于注释 `vm.cpaas.io/previous-size` 中）；如果你之后使用 **Update Specifications** 将其迁移回 Machine Model，并保留预先选中的大小，则会精确恢复该大小。

在以下情况下，虚拟机无法迁移 — 对话框会说明原因 —：其模板未声明 CPU 或内存；虚拟机与所选实例类型设置了相同的附加设置；或者虚拟机不带任何平台标签（请参阅导入的虚拟机）。

同时迁移多个虚拟机：在列表中选择虚拟机，然后点击 **Batch Actions** > **Batch Migrate to Instance Type**。选择一个实例类型；每个虚拟机会单独迁移，无法迁移的虚拟机会被跳过，并且每个虚拟机都会获得各自的结果（**Migrated**、带原因的 **Skipped**，或带消息的 **Failed**）。

实时迁移

注意：有关前提条件、集群范围的迁移限制和每个命名空间的策略，以及 API，请参阅虚拟机实时迁移。

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

更新 NAT 网络配置

使用 NAT 网络模式时，平台默认开放端口 22 以提供 SSH 服务，你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，点击 **Virtualization** > **Virtual Machines**。
3. 点击 ***Virtual Machine Name***。
4. 在 **Basic Information** 部分，点击 **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. 点击 **Update**。

更新标签和注释

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization** > **Virtual Machines**。
3. 点击 ***Virtual Machine Name***。
4. 在 **Basic Information** 部分，点击 **Tags** 或 **Annotations** 右侧的图标。
5. 根据需要进行配置，然后点击 **Update**。

添加服务

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization** > **Virtual Machines**。
3. 点击 ***Virtual Machine Name***。
4. 在 **Login Information** 部分，点击 Internal Route 右侧的图标。
5. 请参阅创建服务页面，快速为虚拟机添加内部路由。
6. 点击 **Confirm**。

重新安装操作系统

强烈建议在重新安装操作系统前备份数据，以防止数据丢失。

注意：此操作会清除虚拟机中 **system disk** 的所有数据以及所有 **snapshots**，且无法撤销。请谨慎操作！

操作步骤

1. 访问 **Container Platform**。
2. 在左侧导航栏中，点击 **Virtualization** > **Virtual Machines**。
3. 找到虚拟机并选择⋮> **Reinstall Operating System**。虚拟机处于已停止、状态未知或异常状态时，才能执行此操作；请先停止正在运行的虚拟机。
4. 在 **Reinstall Operating System** 窗口中，配置以下参数。
   - **Provision Method**：选择 **Image Source**（从镜像地址导入系统磁盘）或 **Image Instance**（从镜像实例克隆系统磁盘）。镜像实例选择器会列出 `kube-public` 的镜像实例以及虚拟机所在命名空间的镜像实例；后者会标注命名空间名称。当前镜像会标记为 **Current image**。
   - **Select Image**：默认使用当前操作系统镜像进行重新安装。如果要重新安装新的操作系统，请先选择虚拟机镜像的操作系统，然后选择属于该操作系统的虚拟机镜像。
5. 点击 **Reinstall**。

新系统磁盘的创建方式与新虚拟机的系统磁盘相同：其 `sourceRef` 会标明镜像实例及其所在的命名空间，其访问模式遵循存储类的 `StorageProfile`（请参阅创建虚拟机）。

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；重启虚拟机后，新 IP 才会生效。

操作步骤

1. 访问 **Container Platform**。

2. 在左侧导航栏中，点击 **Virtualization** > **Virtual Machines**。
3. 找到目标虚拟机并点击⋮> **Configure IP**。
4. 配置 **IP Address**。
   - 填写可用的 IP：绑定固定 IP 后，即使重启，虚拟机也会始终使用此 IP 地址。
   - 将选项留空：使用动态分配（DHCP）获取 IP；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. 点击 **Configure**。

使用 API

控制台操作会映射为对 `VirtualMachine` 资源的标准调用：

- **Update Specifications**，实例类型路径 — 将虚拟机指向其他实例类型：

  ```
  kubectl patch vm web-01 -n demo --type merge -p \
    '{"spec":{"instancetype":{"kind":"VirtualMachineClusterInstancetyp
  e","name":"u1.large"}}}'
  # The resolved object and its recorded revision:
  kubectl get vm web-01 -n demo -o jsonpath='{.status.instancetypeRef}'
  ```

  如果引用固定了修订版本（`spec.instancetype.revisionName`），请在同一个 patch 中清除该修订版本 — 使用新名称但保留旧修订版本会被拒绝：

  ```
  kubectl patch vm web-01 -n demo --type merge -p \
    '{"spec":{"instancetype":{"name":"u1.large","revisionName":null}}}'
  ```

- **Update Specifications**，机器模型路径 — patch 内联资源（ACP 使用资源限制模型，因此请将 `requests` 和 `limits` 设为相同值）；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":{"cp
  u":"2","memory":"4Gi"},"limits":{"cpu":"2","memory":"4Gi"}}}}}}}'
  ```

- 从实例类型切换到机器模型 — 在一个请求中移除引用并写入规格（虚拟机不能同时携带两者）：

```
kubectl patch vm web-01 -n demo --type json -p '[
  {"op":"remove","path":"/spec/instancetype"},
  {"op":"add","path":"/spec/template/spec/domain/resources","value":
    {"requests":{"cpu":"2","memory":"4Gi"},"limits":{"cpu":"2","memor
y":"4Gi"}}}
]'
```

- 迁移到实例类型 — 在一个请求中添加引用，并移除模板包含的所有规格字段：`resources` 中的 CPU 和内存请求与限制、`domain.cpu` 中的 sockets / cores / threads，以及 `domain.memory.guest`。任何字段未移除都会与实例类型冲突，导致更新被拒绝（例如 `VM field(s) spec.template.spec.domain.cpu.cores conflicts with selected instance type`）。无论模板采用何种结构，使用将所有这些字段设为 `null` 的合并 patch 都可行 — 不存在的字段会直接跳过：

```
kubectl patch vm web-01 -n demo --type merge -p '{
  "spec": {
    "instancetype": {"kind": "VirtualMachineClusterInstancetype", "nam
e": "u1.medium"},
    "template": {"spec": {"domain": {
      "resources": {"requests": {"cpu": null, "memory": null},
                    "limits":   {"cpu": null, "memory": null}},
      "cpu":    {"sockets": null, "cores": null, "threads": null},
      "memory": {"guest": null}
    }}}
  }
}'
```

  其他设置（例如 `domain.cpu.model`）会保留；如果实例类型也设置了其中某项，请同时从模板中移除该项。patch 可以在模板中留下空的 `resources`、`cpu` 和 `memory` 对象 — 例如，在没有 `domain.cpu` 的虚拟机上留下 `cpu: {}`。这些对象不会造成影响：实例类型只会与上面列出的各个规格字段冲突，不会与空对象冲突。如果未指定首选项，guest CPU 数量会放置在 sockets 中。控制台还会在注释 `vm.cpaas.io/previous-size` 中记录已移除的字段，以便 **Update Specifications** 还原这些字段；通过 YAML 迁移则不会记录。

- **Update Tags and Annotations**：

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

- **Live Migration** — 创建 `VirtualMachineInstanceMigration`；请参阅虚拟机实时迁移。

导入的虚拟机

当虚拟机的标签（如果虚拟机完全没有标签，则为其模板的标签）中不包含任何含有平台标签域的键（默认值为 `cpaas.io`）时，控制台会将该虚拟机视为已导入。这通常是指从其他平台按原样导入的虚拟机，例如通过 Forklift 导入。对于此类虚拟机，**Update Specifications**、**Reset Password**、**Update Key**、**Reinstall Operating System** 和 **Configure IP** 均处于禁用状态；**Migrate to Instance Type** 会报告无法迁移（批量迁移时也会跳过该虚拟机），并且需要通过其 YAML 进行更新。带有平台标签的导入虚拟机像其他虚拟机一样进行管理：可以在其规格路径上更新规格，也可以将其迁移到实例类型。

查看不受限制：当导入的虚拟机引用了实例类型（`spec.instancetype`）时，其详情页会像其他虚拟机一样显示请求的实例类型、解析后的实例类型以及其运行所使用的大小。要调整其大小，请在 YAML 中更改该引用 — 请参阅 Using the API：

```
# The instance types a virtual machine can reference:
kubectl get virtualmachineclusterinstancetypes
kubectl get virtualmachineinstancetypes -n demo
```

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。

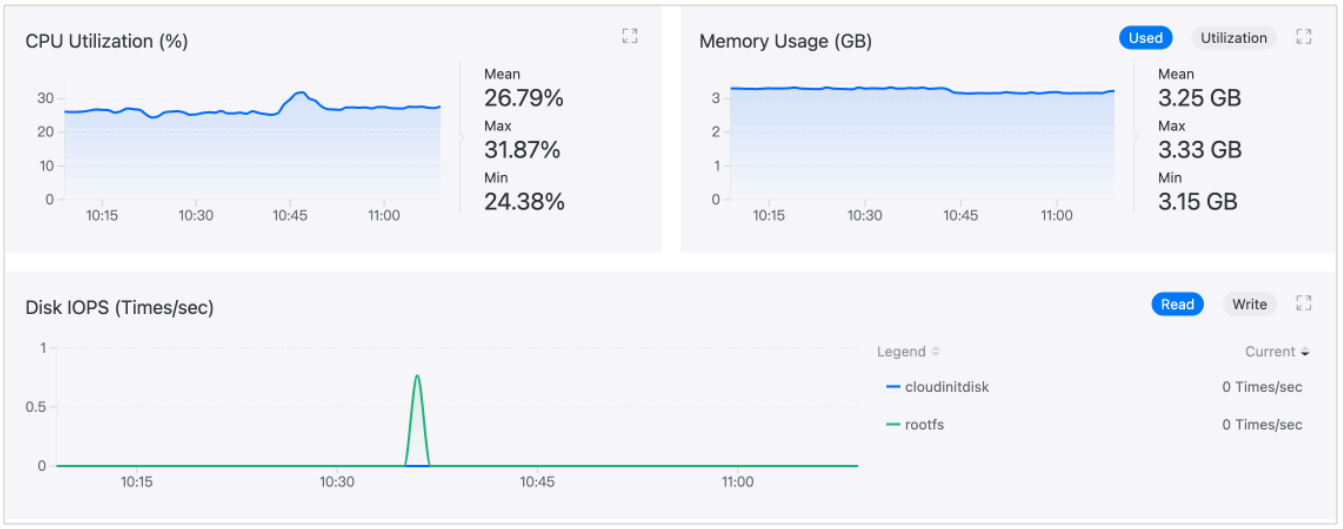
直观呈现的监控数据可为运维巡检或性能调优提供决策支持，而完善的告警和通知机制则有助于保障虚拟机的稳定运行。

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 *内存使用率*。 - 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（vTP

迁移

CPU 和内存热插拔

工作原理

前提条件

步骤 1：创建具有热插拔余量的 VM

步骤 2：热插拔 CPU

步骤 3：热插拔内存

缩减：哪些操作可在线完成，哪些需要重启

已验证的行为总结

故障排除

KubeVirt 节点维护

KubeVirt 如何处理排空

维护窗口之前

排空节点

完成维护

意外的节点故障

使用 Desche

虚拟机恢复

操作步骤

在 KubeVirt 上克隆虚拟机

确保先决条件

在 StorageClass 之间迁移虚拟机

快速开始

理解 VirtualMachineClone 对象

前提条件

准备迁移

使用 DataVolume 克隆进行冷迁移

配置虚拟机的高可用性

运行策略和驱逐策略

概述

术语表

组件概述

隔离和修复期间的事件流

操作步骤

从现有虚拟机

前提条件

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 `kubectl` 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=$(kubectl get kubevirt kubevirt -n kubevirt -o jsonpath='{.statu
s.observedKubeVirtVersion}')
curl -L -o virtctl "https://github.com/kubevirt/kubevirt/releases/downloa
d/${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 Seria
l 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 po
d 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 rea
chable):
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 d
emo
```

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 migra
tion
```

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：

```yaml
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：

```yaml
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：源主机会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。

4. 在 **OperatorHub** 选项卡中，点击 **KubeVirt HyperConverged Cluster Operator** 卡片上的部署。
5. 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)。

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

5. 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 jso
npath='{.status.phase}{"\n"}'
```

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 jsonpat
h='{.spec.liveMigrationConfig}'
# {"parallelMigrationsPerCluster":5,"parallelOutboundMigrationsPerNode":
2,
#  "completionTimeoutPerGiB":150,"progressTimeout":150,
#  "allowAutoConverge":false,"allowPostCopy":false}
```

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

字段	默认值	含义
`allowAutoConverge`	false	限制 guest CPU 以帮助迁移收敛
`allowPostCopy`	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：

```yaml
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

本文介绍如何在不关闭虚拟机（VM）的情况下向正在运行的虚拟机添加 CPU 和内存，以及同样重要的内容：不同客户机操作系统如何响应热插拔。平台侧对所有操作系统都相同，但客户机是否实际*使用*新资源取决于客户机内核或驱动程序。以下标记为*已验证*的各操作系统结果，均在包含 CentOS 7.9、Ubuntu 20.04、Debian 13 和 Windows Server 2022 客户机的运行中集群上进行测试；关于其他版本（Debian 12、Ubuntu 22.04/24.04、RHEL 8.10/9.4 以及其他 Windows 版本）的兼容性说明基于上游和供应商文档，而非这些测试。

目录

工作原理

前提条件

步骤 1：创建具有热插拔余量的 VM

步骤 2：热插拔 CPU

客户机操作系统行为 — CPU

步骤 3：热插拔内存

客户机操作系统行为 — 内存

缩减：哪些操作可在线完成，哪些需要重启

已验证的行为总结

故障排除

工作原理

该平台的虚拟化基于 KubeVirt，KubeVirt 通过 `LiveUpdate` 部署策略（平台默认策略）将 CPU 和内存更改应用于正在运行的 VM：

- **CPU** 热插拔通过增加 CPU 插槽数量来添加 vCPU。热插拔上限为 `maxSockets`；额外插槽会预先连接为空插槽，供 QEMU 稍后填充。
- 内存热插拔通过 **virtio-mem** 设备扩展客户机内存。上限为 `maxGuest`；`guest` 与 `maxGuest` 之间的差值由 virtio-mem 设备支持，该设备可在运行时调整大小——只要大小保持不低于启动时内存，就可以双向调整（缩容为尽力而为，请参见下文）。
- 两项更改都通过自动在线迁移交付：编辑 VM 规格后，平台会将 VM 迁移到新的 `virt-launcher` pod，其资源分配与新大小匹配。无需手动触发迁移——请注意，每次热插拔都会产生一次迁移，并且无法进行在线迁移的 VM 也无法执行热插拔。

进度会报告在 VirtualMachineInstance（VMI）上：

- 条件 `HotVCPUChange` / `HotMemoryChange` — 热插拔正在进行。
- `status.currentCPUTopology` — domain 当前具有的 CPU 拓扑。
- `status.memory` — `guestAtBoot` / `guestCurrent` / `guestRequested`；待处理的内存热插拔会显示 `guestRequested` > `guestCurrent`。
- VM 上的条件 `RestartRequired` — 请求的更改无法在线应用（例如减少 CPU 插槽，或将内存降至启动时的值以下）。

前提条件

- VM 必须支持在线迁移：VMI 上的 `LiveMigratable` 条件必须为 `True`。热插拔依赖在线迁移；不支持迁移的 VM 无法在线调整大小。从存储角度来看，这意味着每个由 **PVC/DataVolume** 支持的磁盘都需要使用具有 `ReadWriteMany` 访问模式的共享存储——包括由 RWO PVC 支持的 CD-ROM。非 PVC 支持的卷（`containerDisk`、cloud-init、`emptyDisk`）不需要 RWX。非存储因素（主机设备、某些 CPU 配置）也可能导致 VM 无法迁移；请注意，该条件消息只会报告它找到的*第一个*阻塞原因。
- VM 必须具有热插拔余量（`maxSockets` / `maxGuest`）。有效上限会在 VMI 启动时固定——要么来自 VM 规格中的显式值，要么在省略时由平台自动计算（如下所述）。运行中的 VM 无法提高这些上限：超出当前上限进行扩展需要重启。
- 对于内存热插拔，客户机必须具有能够协商平台 QEMU 所提供功能集的 virtio-mem 驱动程序——尤其是 `UNPLUGGED_INACCESSIBLE` 功能。对于 Linux 客户机，这是内核能力（请参

见下面按操作系统列出的结果）；对于 Windows 客户机，这是 virtio-win 软件包中的 `viomem` 驱动程序，该驱动程序在现有 VM 中默认未安装（请参见下面的 Windows 行）。在尝试热插拔后，可以在 Linux 客户机中检查：

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

步骤 1：创建具有热插拔余量的 VM

VM 模板中的相关字段：

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
spec:
  template:
    spec:
      domain:
        cpu:
          sockets: 1        # current vCPU count = sockets × cores × thre
ads
          cores: 1
          threads: 1
          maxSockets: 4     # hotplug ceiling: up to 4 sockets without re
start
        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（默认为启动值的 4 倍；比例和绝对上限可通过 `liveUpdateConfiguration` 在集群范围内配置）。计算出的值会写入 VMI，而不是 VM 规格——通过 Web 控制台创建的 VM，其模板仅包含 `resources.requests` / `resources.limits`，仍会获得完整的热插拔余量。我们仍然建议显式设置最大值，以便明确选择上限。

在客户机内部，这些余量在启动时已经可见：使用上述规格创建的 VM 显示 1 个 CPU 处于在线状态，CPU 1–3 作为离线插槽存在，并具有 2Gi 内存。

WARNING

对于模板仅通过 `resources.requests` / `resources.limits` 表示大小的 VM——这是 Web 控制台生成的配置——第一次调整大小时，每种资源的行为有所不同（已针对这类 VM 进行验证）：

- 内存：向运行中的 VM 添加 `domain.memory.guest` 会被视为在线更新。热插拔正常进行——无需重启。
- **CPU**：模板没有 `cpu` 块，而*添加该块*（即使仅添加 `cpu.sockets`）也是无法在线更新的更改。VM 会获得 `RestartRequired`（`a non-live-updatable field was changed in the template spec`）；因此，第一次 CPU 调整大小需要重启一次，之后规格会包含显式的 `cpu` 块，后续每次增加插槽都会在线应用。
- 不要将两者结合：如果一次补丁同时添加 `cpu` 块和 `memory.guest`，CPU 更改会设置 `RestartRequired`，该条件也会禁止（原本可以在线应用的）内存热插拔。如果需要在线应用内存更改，请单独为内存创建补丁。

步骤 2：热插拔 CPU

增加运行中 VM 的插槽数：

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

接下来会发生以下情况（通常在 ~30 秒内完成）：

1. VMI 获得 `HotVCPUChange` 条件。
2. 平台会自动将 VM 实时迁移到新的 virt-launcher pod。
3. `status.currentCPUTopology` 报告新的插槽数，QEMU 随即热添加 vCPU。

观察过程：

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

客户机操作系统行为 — CPU

对于每种操作系统，vCPU 都会在 hypervisor 层面进行热添加。但客户机是否会立即使用该 vCPU，取决于客户机自身：

客户机操作系统	行为（已验证）
CentOS 7.9	新 vCPU 会自动上线。RHEL 系列自带 `40-redhat.rules`，会无条件将任何热添加的 CPU 上线。迁移完成后几秒内，`nproc` 就会反映新的数量。
Ubuntu 20.04	新 vCPU 会出现，但保持离线状态（`nproc` 不变）。Ubuntu 的 `40-vm-hotadd.rules` 仅会在 Hyper-V 和 Xen 上自动将 CPU 上线——不适用于 KVM/QEMU 客户机。
Debian 13	与 Ubuntu 相同——vCPU 已添加，但保持离线状态；Debian 完全没有自动上线规则。
Windows Server 2022 (Standard)	新 vCPU 会自动且立即被识别——在无需重启且无需手动操作的实时会话中，`%NUMBER_OF_PROCESSORS%` 已反映新的数量。

要在 Ubuntu/Debian 上将 CPU 上线（以及在任何没有覆盖 KVM 的 udev 规则的其他发行版上执行此操作），可以手动将其上线：

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

或者安装 udev 规则（这正是 RHEL 提供的规则），以便后续热插拔的 CPU 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
```

步骤 3：热插拔内存

增加运行中 VM 的客户机内存：

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

平台会对 VM 执行实时迁移，并调整 virtio-mem 设备的大小。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
```

客户机操作系统行为 — 内存

此时，客户机驱动程序的支持情况决定了一切。对于无法使用 virtio-mem 的客户机，热插拔不会失败，而是会静默地保持待处理状态：

客户机操作系统	内核 / 驱动程序	行为（已验证）
Debian 13	6.12	有效。 修补后约 30 秒，`guestCurrent` 达到新的大小；所有热插拔的内存都会自动上线，无需手动操作（与 CPU 热插拔不同）。请注意，自动上线是镜像本身的属性，而非 virtio-mem 驱动程序的属性：此镜像的内核启用了内存块自动上线（`/sys/devices/system/memory/auto_online_blocks` 报告 `online`）。对于没有此类策略（内核默认设置、引导参数或 udev 规则）的客户机，热插拔的内存块会保持离线状态，virtio-mem 也会暂停后续插入操作——如果在其他镜像上热插拔停滞，请检查 `auto_online_blocks`。
CentOS 7.9	3.10	不会生效。 此内核没有 virtio-mem 驱动程序。`guestRequested` 会一直高于 `guestCurrent`；不会报错，也不会引发 `RestartRequired` 条件。
Ubuntu 20.04	5.4 (GA)	与 CentOS 7 相同——该内核版本早于 virtio-mem，因此完全不支持它。

客户机操作系统	内核 / 驱动程序	行为（已验证）
Ubuntu 20.04	5.15 (HWE)	仍然无法使用，但原因更隐蔽：`virtio_mem` 模块已加载，但设备会拒绝它——`dmesg` 显示 `virtio_mem virtio1: virtio: device refuses features: 3`，因为平台的 QEMU 要求内核不具备的 `UNPLUGGED_INACCESSIBLE` 功能。请注意，Ubuntu 22.04 的 GA 内核同样是 5.15，也不具备此功能——请使用 22.04 HWE 内核或 Ubuntu 24.04+。
Windows Server 2022	virtio-win `viomem`	安装 **`viomem`** 驱动程序后即可使用——已通过实时测试验证：8Gi → 10Gi 的更改在线生效，客户机立即报告新的总量，无需重启。如果没有该驱动程序，virtio-mem PCI 设备（`PCI\VEN_1AF4&DEV_1058`）会在设备管理器中显示错误，请求则会一直处于待处理状态。平台提供的 virtio-win ISO 中包含该驱动程序（已通过 virtio-win 0.1.266 验证）：从提升权限的 shell 中安装，然后重试，或等待待处理的请求生效。

从挂载的 virtio-win CD-ROM 安装 Windows 驱动程序（驱动器号可能不同）：

```
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 客户机工具安装程序（`virtio-win-guest-tools.exe`，位于同一 ISO 中）在当前版本中也会默认安装 `viomem`，因此，使用较新版本客户机工具部署的 VM 开箱即可支持内存热插拔。该驱动程序仅适用于 x64（较新版本的 Windows 上还适用于 ARM64）。

关于待处理状态的说明：

- 待处理请求不会丢失。在 VM 下次重启时，它会作为普通引导内存应用（`guestAtBoot` 会变为请求的大小——已在 CentOS 7 上验证）。对于没有可用驱动程序的客户机，内存“热插拔”会平稳退化为计划中的冷调整。或者，将 `memory.guest` 设回当前实时值即可取消待处理请求——无需 `RestartRequired` 即可接受回退，因为实际上并未将内存缩减到低于引导时的大小。
- 管理员可以从节点侧确认此机制：virt-launcher pod 内的 `virsh dumpxml <domain>` 会显示一个 `<memory model='virtio-mem'>` 设备，其 `requested` 大小会随每次热插拔变

化，而 `current` 则反映客户机驱动程序实际接受的大小（没有可用 virtio-mem 的客户机上为 `current: 0`）。

缩减：哪些操作可在线完成，哪些需要重启

这两种资源的行为不同（已在 Linux 和 Windows 客户机上验证）：

- 内存可以在线缩减——但属于尽力而为的请求，并且最多只能缩减到引导时的值。 将 `memory.guest` 降低到任何 ≥ `guestAtBoot` 的值时，可以在线*接受*：不会出现 `RestartRequired`，会执行一次自动实时迁移，并请求 virtio-mem 设备归还内存。客户机是否能够完全执行该请求无法保证——这取决于客户机实际上能够释放热插拔内存范围的多少（内存压力、不可移动的内核分配、锁定页以及上线策略都会产生影响）。务必通过比较 `guestCurrent` 和 `guestRequested` 来确认结果。在空闲客户机上，我们的测试中完整缩减在一分钟内完成（Debian 13 上从 4Gi → 3Gi，Windows Server 2022 上从 10Gi → 9Gi → 8Gi）；在处于人为内存压力下的 Debian 13 客户机上，相同请求执行到中途便停滞（`guestRequested: 2Gi`，`guestCurrent` 停留在约 2.1Gi），没有错误，也没有 condition——即使移除压力后，仍有少量内存无法回收，因为内核分配已落在热插拔内存块中。部分完成的缩减不属于失败状态：已释放的部分会被归还，请求会保持待处理状态，你可以随时将 spec 恢复为当前实时值。
- 无法在线将内存降低到 **`guestAtBoot`** 以下：VM 会获得 `RestartRequired`（`memory updated in template spec to a value lower than what the VM started with`），并且在重启前不会发生任何变化。换句话说，在线缩减只能尝试回收之前热插拔的内存，不能回收引导内存。
- **CPU** 完全无法在线缩减。 `sockets` 的任何减少都会设置 `RestartRequired`（`Reduction of CPU socket count requires a restart`）；拓扑结构在重启前不会改变。

将 spec 恢复为当前实时值会清除待处理的 `RestartRequired` condition。

已验证的行为总结

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（socket ↑）	通过自动实时迁移应用	vCPU 自动上线	已添加但处于离线状态——手动上线或使用 udev 规则	已添加但处于离线状态——手动上线或使用 udev 规则
内存热插拔（客户机 ↑）	通过自动实时迁移调整 virtio-mem 大小	永久待处理；下次重启时应用	永久待处理（5.15 无法完成功能协商）；下次重启时应用	约 30 秒后上线，内存块自动上线
内存缩减（客户机 ↓，≥ 引导值）	通过自动实时迁移在线请求；完成情况取决于客户机是否释放内存	—（没有可用驱动程序）	—（没有可用驱动程序）	空闲时在线完成；内存压力下会部分停滞（无错误）
内存缩减（客户机 ↓，< 引导值）	不支持在线缩减——`RestartRequired`	—	—	—
CPU 缩减（socket ↓）	不支持在线缩减——`RestartRequired`	—	—	—

关于内存热插拔的实用指导：关键在于客户机驱动程序是否支持 `UNPLUGGED_INACCESSIBLE` virtio-mem 功能，而各发行版内核会独立于版本号对该功能进行回移植——因此不要仅依据 `uname -r` 进行判断。已知可正常工作的客户机包括 Debian 12+、Ubuntu 24.04+（或使用 HWE 内核的 22.04——GA 5.15 内核无法工作），以及对该功能进行了回移植的 RHEL 系列发行版（RHEL 9.4+ 和 8.10 的文档说明支持 virtio-mem 客户机；但请注意，RHEL 8.10 默认仅支持热*插入*：除非客户机使用 `memhp_default_state=online_movable` 引导，否则会禁用内存拔出）；经过测试的 Windows Server 2022 客户机在安装平台 virtio-win ISO 中的 `viomem` 驱动程序后即可工作。如有疑问，请执行一次热插拔，并按照上文所示检查 `dmesg` / 设备管理器。对于 CPU 热插拔，此处测试的四个客户机均成功完成了管理程序侧的热添加；客户机是否无需额外步骤即可使用这些 vCPU 则各不相同——RHEL 系列和 Windows Server 2022 可以，Debian/Ubuntu 需要使用上文的 udev 规则，其他 Windows 版本可能有所不同（Microsoft 将

CPU 热添加与特定 Server SKU 中的动态硬件分区支持相关联——在依赖该功能前，请先确认所用版本是否支持）。

故障排除

- 热插拔似乎卡住 — 检查 VM 是否可以实时迁移（VMI 上的 `LiveMigratable` condition、命名空间中的迁移事件）。热插拔通过实时迁移实现，因此任何会阻止迁移的因素（由 RWO 支持的 PVC 磁盘、节点约束）都会阻止热插拔。condition 消息只显示第一个阻塞原因——解决一个问题后，可能会发现另一个问题。
- **`guestRequested` > `guestCurrent`**（扩容请求卡住） — 客户机未接受内存。在 Linux 上，运行 `dmesg | grep virtio_mem`：`device refuses features` 表示驱动程序缺少 `UNPLUGGED_INACCESSIBLE` 功能；完全没有输出表示驱动程序从未绑定——可能是内核不支持 virtio-mem，也可能只是模块尚未加载，因此在认定内核版本太旧之前，请先检查 `lsmod | grep virtio_mem`。如果驱动程序正常，请检查上线策略（`cat /sys/devices/system/memory/auto_online_blocks`；新增内存块仍处于离线状态时，virtio-mem 会暂停插入）。在 Windows 上，扩容卡住表示 `viomem` 驱动程序缺失、版本太旧、未绑定或处于错误状态——在这些情况下，设备管理器都会将 virtio-mem PCI 设备（`PCI\VEN_1AF4&DEV_1058`）显示为错误状态。如上所示，从平台的 virtio-win ISO 安装当前驱动程序即可解决常见问题；随后待处理的请求会在无需重启的情况下生效。无论哪种情况，请求都不会丢失——升级客户机，或安排重启以冷态应用内存。
- **`guestRequested` < `guestCurrent`**（缩容请求卡住） — 通常是客户机无法释放热插拔内存范围中的相应部分（内存正在使用、页面已锁定，或不可移动的内核分配落在热插拔内存块中）——但首先请确认客户机确实支持*并已启用*内存拔出：某些能够正常热插入内存的客户机默认禁用了内存拔出（例如，RHEL 8.10 需要使用 `memhp_default_state=online_movable` 引导）。请求会在后台持续重试，之后可能完成、部分完成，或始终无法完全完成。请降低客户机的内存压力，接受部分完成的结果，或将 `memory.guest` 恢复为当前实时值以取消剩余请求。
- `nproc` 中看不到新的 **vCPU** — vCPU 已热添加，但处于离线状态。检查 `cat /sys/devices/system/cpu/online` 与 `offline`，并按照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 缩减、内存低于引导值、首次添加 `cpu` 块，或提高 `maxSockets` / `maxGuest`）。你可以恢复 spec 以清除此 condition，或重启 VM 以应用更改。

KubeVirt 节点维护

使用本操作手册执行单个正常工作节点的计划维护。它会封锁节点，请求 Kubernetes 将其排空，并由 KubeVirt 在节点停止服务前疏散 VM。能够实时迁移的 VM 会始终保持运行，但客户机可能会短暂停顿，并且 CPU、内存和网络使用量可能增加。无法实时迁移的 VM 会在另一个节点上重新启动，因此应在维护窗口开始前而不是维护期间识别这些 VM。

本操作手册不适用于故障或网络分区节点的恢复。在这种情况下，请先隔离该节点，然后按照 VM 节点故障迁移与恢复操作。旧节点可能仍在运行 VM 时，请勿重新创建 VM。

> **WARNING**
>
> 切勿使用 `--force`、`--disable-eviction`、`--grace-period=0`，也不要强制删除 `virt-launcher` Pod 来完成计划排空。这些选项可能绕过 KubeVirt 的疏散流程并中断客户机。

目录

KubeVirt 如何处理排空

维护窗口之前

1. 确认工作节点容量
2. 确认驱逐策略
3. 列出无法迁移的 VM
4. 根据迁移限制规划排空

排空节点

完成维护

意外的节点故障

KubeVirt 如何处理排空

`virt-launcher` 是 Kubernetes Pod，但客户机是 KubeVirt VMI。当 Kubernetes Eviction API 收到针对 launcher 的驱逐请求时，`virt-launcher-eviction-interceptor.kubevirt.io` 准入 webhook 会将可迁移的 VMI 标记为待疏散，并以 **HTTP 429** 拒绝该请求，因此驱逐操作本身不会停止客户机。KubeVirt 会在后台对 VMI 执行实时迁移。只有迁移完成且源节点上的 launcher 不再处于活动状态后，才会允许驱逐。`kubectl drain` 会自动重试，因此排空期间出现 429 是预期的进展，而不是故障。

排空过程中迁移 VM 时，迁移完成前会输出拦截器的消息：

```
evicting pod default/virt-launcher-centos-vm-qvkfn
error when evicting pods/"virt-launcher-centos-vm-qvkfn" -n "default" (wi
ll retry after 5s): admission webhook "virt-launcher-eviction-intercepto
r.kubevirt.io" denied the request: Evacuation in progress: Eviction trigg
ered evacuation of VMI "default/centos-vm"
evicting pod default/virt-launcher-centos-vm-qvkfn
pod/virt-launcher-centos-vm-qvkfn evicted
```

首先从 VMI 读取有效驱逐策略；如果未设置，则从 KubeVirt 集群配置读取：

策略	排空行为	适用情况
`LiveMigrateIfPossible`	迁移所有可迁移的 VMI；驱逐其余 VMI，并遵循其运行策略。排空始终会完成。	推荐策略，也是随软件包提供的策略。 了解哪些 VM 无法迁移，并接受它们重新启动。
`LiveMigrate`	迁移可实时迁移的 VMI。只要有一个 VMI 无法迁移，排空就会无限期受阻。	仅适用于不允许停止任何 VM，且与停止 VM 相比，阻止节点维护的风险更小的情况。
`None`	不请求迁移；VM 会随节点一起停止。	不适用于需要保持 VM 运行的场景。
`External`	阻止驱逐，由外部控制器处理。	仅在使用该控制器及其操作手册时使用。

实际上，VM 可能因多种原因而无法迁移，例如 RWO 存储、主机设备、某些网络绑定，或目标节点无法满足的 CPU 型号。`LiveMigrateIfPossible` 可避免这些情况导致常规节点维护受阻：可迁移的 VM 会在不中断的情况下迁移，只有确实无法迁移的 VM 才会在其他位置重新启动。

WARNING

请检查现有集群的集群级值；该值可能为 `None`。 软件包提供的值为 `spec.evictionStrategy: LiveMigrateIfPossible`，因此全新安装可安全排空。`HyperConverged` 若在此之前创建，或基于空的 `spec` 创建，则没有显式值——而在 ACP 上，HCO 自身的回退值会解析为 `None`，因为它用于推导默认值的集群基础架构资源在 ACP 集群上不存在，导致即使在多节点集群上，`hco.status.infrastructureHighlyAvailable` 仍为 `false`。在此类集群上，排空会停止所有未设置 VM 级策略的 VM。在安排任何维护前，请使用步骤 2 中的命令读取有效值；如果集群级值仍为空或为 `None`，请进行设置：

```
kubectl patch hyperconverged kubevirt-hyperconverged -n kubevirt --
type=merge \
  -p '{"spec":{"evictionStrategy":"LiveMigrateIfPossible"}}'
```

维护窗口之前

在安排变更前完成以下验证。排空不是 VM 兼容性测试：在 `LiveMigrateIfPossible` 下，无法迁移的 VM 会在未事先通知你的情况下于其他节点上重新启动；而在 `LiveMigrate` 下，它会导致节点无法完成排空。无论哪种情况，都应在维护窗口开始前，而不是维护期间，获取无法迁移的 VM 列表。

1. 确认工作节点容量

至少保留两个 `Ready` 且可调度的工作节点，并确认目标节点能够满足每个 VM 的 CPU、内存、设备、亲和性、污点、容忍度、拓扑和网络要求。

```
kubectl get nodes -o wide
kubectl describe node <possible-destination>
kubectl get vmi -A -o wide
kubectl get pod -A -l kubevirt.io=virt-launcher -o wide
```

除非容量规划已获批准，否则不要同时对两个工作节点开始维护。

2. 确认驱逐策略

对于节点上的每个 VMI，检查显式设置的 VMI/VM 值。如果为空，则检查已安装的 KubeVirt 配置；不要根据 ACP 版本号推断值。HCO 字段为 `spec.evictionStrategy`。应发现实际提供服务的 CRD 版本，而不是硬编码版本，这样如果后续版本提供不同版本，检查仍可正常工作。

```
kubectl get vmi <vmi-name> -n <vm-namespace> \
  -o jsonpath='evictionStrategy={.spec.evictionStrategy}{" node="}{.statu
s.nodeName}{"\n"}'
kubectl get vm <vm-name> -n <vm-namespace> \
  -o jsonpath='runStrategy={.spec.runStrategy}{" templateEvictionStrategy
="}{.spec.template.spec.evictionStrategy}{"\n"}'
kubectl get kubevirt -n kubevirt \
  -o jsonpath='{range .items[*]}{.metadata.name}{" evictionStrategy="}{.s
pec.configuration.evictionStrategy}{"\n"}{end}'
kubectl get crd hyperconvergeds.hco.kubevirt.io \
  -o jsonpath='{range .spec.versions[*]}{.name}{" served="}{.served}{" st
orage="}{.storage}{"\n"}{end}'
HCO_VERSION="$(
  kubectl get crd hyperconvergeds.hco.kubevirt.io \
    -o jsonpath='{range .spec.versions[?(@.served==true)]}{.name}{"\t"}{.
storage}{"\n"}{end}' \
    | awk '$2 == "true" { print $1; exit }'
)"
test -n "$HCO_VERSION" || { echo "no served HCO CRD version found" >&2; e
xit 1; }
kubectl explain hyperconverged.spec.evictionStrategy \
  --api-version="hco.kubevirt.io/${HCO_VERSION}"
kubectl get hyperconverged kubevirt-hyperconverged -n kubevirt \
  -o jsonpath='hco evictionStrategy={.spec.evictionStrategy}{" HA="}{.sta
tus.infrastructureHighlyAvailable}{"\n"}'
```

在 ACP 4.4 上，CRD 以 `hco.kubevirt.io/v1beta1` 的形式提供，上述命令会输出 `None` / `LiveMigrate` / `LiveMigrateIfPossible` / `External` 枚举。如果 `kubectl explain` 无法在提供服务的版本上解析 `spec.evictionStrategy`，请停止操作，并针对已安装的 API 版本更新本文档及其兼容性测试，不要猜测其他字段路径。

如果 VMI 和 VM 上的 `evictionStrategy` 都为空，则表示使用集群级值——也就是软件包设置的值；对于早于该设置的集群，则使用 `None`。

大多数 VM 无需设置 VM 级值：继承集群的 `LiveMigrateIfPossible` 是预期行为。只有当某个 VM 的处理方式必须不同于集群时，才显式设置该值，并在维护窗口前于该 VM 的事实来源（GitOps 清单、表单或 API）中设置。需要在中断后重新创建并启动 VM 时，设置 `runStrategy: Always`。如果 VM 仍具有已弃用的 `spec.running` 字段，请勿向其添加 `runStrategy`；应更新源清单，并且只使用其中一个字段。

VM 清单的相关部分如下：

```
spec:
  runStrategy: Always
  template:
    spec:
      evictionStrategy: LiveMigrateIfPossible
```

应用此变更时，保留 VM 现有的域、网络、卷和放置设置。只有在 VM 不由 GitOps 或其他控制器管理时，才可以直接进行修补；否则应更新该事实来源。

更改 VM 模板不会更改已经运行的 VMI。如果正在运行的 VMI 的有效策略仍为 `None`，则无论当前 VM 模板中的设置如何，排空都会将其停止；请在获批准的维护窗口期间重新创建或重启该 VM，以使其采用新值。绝不要通过修补正在运行的 VMI 来强制执行此更改。

3. 列出无法迁移的 VM

对于当前位于维护节点上的每个 VMI，读取 `Ready` 和 `LiveMigratable`。检查 PVC 是否已绑定，并确认存储、网络、设备、CPU 型号和放置规则允许使用目标节点。RWX 等共享存储很常见，但 `LiveMigratable` 条件才是已安装 KubeVirt 版本的权威检查依据。

```
kubectl get vmi <vmi-name> -n <vm-namespace> -o wide
kubectl get vmi <vmi-name> -n <vm-namespace> \
  -o jsonpath='{range .status.conditions[?(@.type=="LiveMigratable")]}Liv
eMigratable={.status}{" reason="}{.reason}{" message="}{.message}{"\n"}{e
nd}'
kubectl get pvc -n <vm-namespace>
kubectl get vmim -n <vm-namespace>
```

具有 `LiveMigratable=False` 的 VMI 在 `LiveMigrateIfPossible` 下会被停止，并在其他节点上重新启动。这正是预期的取舍——一个难以处理的 VM 不应阻塞节点维护——但只有在这是一次已知且已沟通的重启，而不是突发情况时，才可以接受。对于每个此类 VM，读取该条件的 `reason` 和 `message`，然后在维护窗口前决定采取以下措施：修复阻碍因素（例如将其移至 RWX 存储）、接受重启并告知其所有者，或提前自行将 VM 移出该节点。

4. 根据迁移限制规划排空

迁移设置决定排空能否在超时时间内完成。KubeVirt 一次只会从一个节点迁移有限数量的 VM，因此包含大量 VM 的节点会分批排空，而不是并行排空：

```
kubectl get hyperconverged kubevirt-hyperconverged -n kubevirt \
  -o jsonpath='{.spec.liveMigrationConfig}{"\n"}'
kubectl get kubevirt -n kubevirt \
  -o jsonpath='{range .items[*]}{.spec.configuration.migrations}{"\n"}{en
d}'
```

默认值为 `parallelOutboundMigrationsPerNode: 2`、`parallelMigrationsPerCluster: 5`、`completionTimeoutPerGiB: 150` 和 `progressTimeout: 150`。在每个节点默认同时进行两项出站迁移的情况下，排空十个 VM 至少需要五批；请据此设置 `--timeout`，或者在维护窗口期间提高 `hco.spec.liveMigrationConfig` 中的限制，并在之后恢复原值。超过 `completionTimeoutPerGiB` 或停滞时间超过 `progressTimeout` 的迁移会被取消，随后排空会报告该 VM 仍然存在。

排空节点

1. 在更改节点之前，记录节点名称、每个 VMI UID 以及当前 VMI 所在的节点。

2. 隔离健康节点。隔离会阻止新工作负载放置到该节点上，但不会停止现有 VM。

```
kubectl cordon <maintenance-node>
```

3. 使用 Kubernetes Eviction API 执行排空，并等待 KubeVirt 迁移 VMI。不要手动删除 launcher Pod，也不要创建一次性驱逐并将其首次返回的 429 响应视为失败。

```
kubectl drain <maintenance-node> \
  --ignore-daemonsets \
  --delete-emptydir-data \
  --timeout=30m
```

4. 在单独的终端中监视迁移和排空：

```
kubectl get vmim -A -w
kubectl get vmi -A -o wide -w
kubectl get pod -A -l kubevirt.io=virt-launcher -o wide -w
kubectl get events -A --sort-by=.lastTimestamp
```

对于预期会迁移的 VM，仅当匹配的 VMIM 处于 `Succeeded` 状态、其 `status.migrationState.sourceNode` 为维护节点且 `targetNode` 为另一个工作节点、VMI UID 未改变、VMI 仍处于 `Running` 和 `Ready`，并且仅剩一个活动 launcher 时，才接受此次移动。Pod 名称发生变化、收到 429 响应或仅有驱逐日志，都不能作为迁移证据。未改变的 **VMI UID** 是决定性检查项：UID 发生变化意味着 VM 被重新创建，而不是实时迁移。在 `LiveMigrateIfPossible` 下，不可迁移的 VM 根本不会生成 VMIM——它会被驱逐并重新创建，因此其 VMI UID 会发生变化，并会短暂离开 `Running`。确认每个此类 VM 都在步骤 3 的列表中；如果某个 VM 未列在该列表中却发生了重启，说明迁移因尚未查明的原因失败，应在继续之前进行调查。

迁移后，源 launcher 会在 `Completed` 阶段中保持列出一段时间。请筛选正在运行的 Pod，避免将其误认为第二个活动 launcher：

```
kubectl get pod -A -l kubevirt.io=virt-launcher -o wide \
  --field-selector=status.phase=Running
```

5. 仅当 `kubectl drain` 报告成功且没有 VMI 或 launcher Pod 分配到该节点后，才开始操作系统或硬件工作：

```
kubectl get vmi -A -o wide
kubectl get pod -A -l kubevirt.io=virt-launcher \
  --field-selector spec.nodeName=<maintenance-node> -o wide
```

如果排空超时或报告某个 VMI 被阻塞，请保持节点处于隔离状态，保留 VMIM/VMI 事件和条件，并解决报告的容量、存储、放置、PDB 或迁移限制问题。不要强制排空。

完成维护

节点恢复正常且 `Ready` 后，使其重新变为可调度状态：

```
kubectl get node <maintenance-node>
kubectl uncordon <maintenance-node>
kubectl get node <maintenance-node>
```

在关闭变更前，重新检查已移动的 VM 及其应用健康状况，并确认不存在重复的活动 launcher。

意外的节点故障

此计划内操作步骤无法安全地恢复已断电或发生网络分区的节点。首先使用已批准的带外隔离或节点修复工作流，然后根据该工作流处理过期的 VMI、launcher 和卷附加资源。绝不能允许 KubeVirt 运行来宾系统的两个副本。

使用 Descheduler 重新平衡 KubeVirt 虚拟机

Alauda Container Platform Descheduler 插件可以请求驱逐 `virt-launcher` Pod。KubeVirt 会拦截该请求，并对 VMI 执行实时迁移，而不是停止 VM。请将其用作次要且有界的重新平衡机制。对于发生故障或需要有意维护的节点，请改为遵循 KubeVirt 节点维护。

本页面介绍 ACP ModulePlugin 软件包。该软件包通过自身的资源进行配置——策略 ConfigMap 和 CronJob（或 Deployment）——不会安装 descheduler CRD；也不存在 operator 级别的 API 可用于对其进行配置。

以下操作步骤已在 ACP 4.4（Kubernetes v1.35）以及 KubeVirt v1.8.4（HCO 1.18.1）上完成验证，使用的 Descheduler 软件包构建版本默认渲染 `EvictionsInBackground` 功能门（任何新于 `v0.36.0-4` 的软件包版本）。请从运行中的集群读取已安装的版本；软件包和 KubeVirt 独立演进。

目录

Descheduler 驱逐如何变为实时迁移

先决条件

软件包渲染功能门

VM 使用迁移驱逐策略时支持实时迁移

发现插件

为什么默认策略永远不会迁移 VM

配置有界 VM 再平衡策略

验证重新平衡

停止并回滚

兼容性检查

Descheduler 驱逐如何变为实时迁移

1. Descheduler 选择一个 `virt-launcher` Pod，并调用 Eviction API。Eviction 对象携带注解 `requested-by: sigs.k8s.io/descheduler`。
2. KubeVirt 会在创建 每个 `virt-launcher` Pod 时添加注解 `descheduler.alpha.kubernetes.io/request-evict-only: ""`，无论驱逐策略是什么。Descheduler 只会对携带此注解的 Pod 应用其后台驱逐处理。
3. KubeVirt webhook `virt-launcher-eviction-interceptor.kubevirt.io` 根据 VMI 的有效驱逐策略决定后续行为： 如果已设置，则使用 VMI 的 `spec.evictionStrategy`；否则使用 `KubeVirt` 资源中的集群默认值（`spec.configuration.evictionStrategy`，由 HCO 根据 `HyperConverged.spec.evictionStrategy` 渲染）。

有效策略	**VMI** 可实时迁移	**Webhook** 响应	结果
`LiveMigrate`	是	HTTP 429 `Eviction triggered evacuation of VMI "<ns>/<name>"`	VMI 被标记为待疏散并执行实时迁移
`LiveMigrate`	否	HTTP 429 `... is not live-migratable`	不执行任何操作；Descheduler 记录错误并在下一周期重试
`LiveMigrateIfPossible`	是	HTTP 429 `Eviction triggered evacuation ...`	VMI 被标记为待疏散并执行实时迁移
`LiveMigrateIfPossible`	否	允许	**Pod** 被删除；**VM** 在另一个节点上重启
`None` 或未设置	任意	允许	**Pod** 被删除；**VM** 在另一个节点上重启

当 VMI 被标记为待疏散时，KubeVirt 会在 VMI 上设置 `status.evacuationNodeName`，向 VMI 添加注解 `kubevirt.io/eviction-source: descheduler`，并创建一个名为 `kubevirt-evacuation-<suffix>` 的 `VirtualMachineInstanceMigration`，其带有注解 `kubevirt.io/evacuationMigration: <source-node>`。迁移处于等待状态期间，KubeVirt 还会使用 `descheduler.alpha.kubernetes.io/eviction-in-progress: "true"` 对源 Pod 添加注解。迁移完成后， `virt-handler` 会移除 `eviction-source` 并清除 `evacuationNodeName`，因此这两个标记仅在疏散进行期间可见。

启用 `EvictionsInBackground=true` 后，Descheduler 会将该特定的 429 视为成功的驱逐*请求*：它会记录 `Eviction in background assumed`，将该请求计入 `maxNoOfPodsToEvict*`，并记住该 Pod。在下一周期中，它会看到 `eviction-in-progress` 注解，并记录 `Eviction in background already requested (ignoring)`，而不是再次发起请求。没有该功能门时，429 会被记录为驱逐失败，并且同一个 launcher 可能在每个周期再次被提名。

> **WARNING**
>
> 当 VM 的有效策略为 `None`，或为 `LiveMigrateIfPossible` 且 VMI 不可迁移时，Descheduler 驱逐其 launcher 后，VM 会重启而不会迁移。在允许驱逐 VM launcher 之前，请验证其策略。

先决条件

软件包渲染功能门

当前软件包会根据自身的 values 渲染功能门；无需管理员配置。在运行 VM 的业务集群上验证：

```
kubectl -n descheduler get cronjob descheduler \
  -o jsonpath='{.spec.jobTemplate.spec.template.spec.containers[0].args}
{"\n"}'
```

预期输出：

```
["--policy-config-file=/policy-dir/policy.yaml","--feature-gates=Eviction
sInBackground=true","--v=3"]
```

如果缺少该参数，说明已安装的软件包早于此变更： 升级 **Alauda Container Platform Descheduler** 插件。不要手动编辑生成的 CronJob 或 Deployment。要查看下文使用的 `Eviction in background ...` 日志行，还需要 `--v=3` 详细程度。

VM 使用迁移驱逐策略时支持实时迁移

VM 必须使用可从目标节点访问的共享存储（例如 RWX 块存储），报告 `LiveMigratable=True`，并将 `LiveMigrate` 或 `LiveMigrateIfPossible` 作为有效驱逐策略。 ACP 随附的 HCO 示例会设置 `spec.evictionStrategy: LiveMigrateIfPossible`；在 ACP 上，使用空 `spec` 创建的 `HyperConverged` 默认为 `None`。检查这两个层级：

```
kubectl -n kubevirt get kubevirt kubevirt-kubevirt-hyperconverged \
  -o jsonpath='{.spec.configuration.evictionStrategy}{"\n"}'
kubectl get vmi <vmi-name> -n <vm-namespace> \
  -o jsonpath='{.spec.evictionStrategy} {.status.conditions[?(@.type=="Li
veMigratable")].status}{"\n"}'
```

VMI 值会在创建 VMI 时固定。更改 `HyperConverged.spec.evictionStrategy` 会立即传播到 `KubeVirt` 资源， 但只会影响之后创建的 VMI；现有 VMI 会保留其记录的值（例如 `None`），直到 VM 重启。

发现插件

按照工作负载重新平衡（Descheduler）中的说明，从 管理员 **> Marketplace >** 集群插件 安装 **Alauda Container Platform Descheduler**。

`ModuleInfo` 仅由 `global` （管理）集群提供；在业务集群上运行 该命令会失败，并显示 `the server doesn't have a resource type "moduleinfo"`。策略 ConfigMap 和 CronJob 位于业务集群上，位于软件包创建的 `descheduler` 命名空间中。

在 `global` 集群上：

```
kubectl get moduleinfo -l cpaas.io/module-name=descheduler \
  -o custom-columns='NAME:.metadata.name,CLUSTER:.metadata.labels.cpaas\.
io/cluster-name,PHASE:.status.phase,VERSION:.spec.version'
```

在业务集群上：

```
kubectl -n descheduler get cronjob,configmap,job -l app.kubernetes.io/nam
e=descheduler
kubectl -n descheduler get configmap descheduler -o jsonpath='{.data.poli
cy\.yaml}'
```

该策略是原始的 `descheduler/v1alpha2` `DeschedulerPolicy`：使用本页面所述的上游插件和参数进行配置；它没有预定义的 KubeVirt 配置文件或解除调度模式设置。（`LowNodeUtilization.args.evictionLimits.node` 是常规的上游字段，可以与顶层 `maxNoOfPodsToEvict*` 限制结合使用。）

为什么默认策略永远不会迁移 VM

该软件包提供一个 `default` 配置文件，其 `DefaultEvictor` 配置为：

```
- name: DefaultEvictor
  args:
    podProtections:
      defaultDisabled:
        - PodsWithLocalStorage
      extraEnabled:
        - PodsWithPVC
```

- `PodsWithPVC` 会保护挂载 PersistentVolumeClaim 的每个 Pod。因此，PVC 支持的 VM 的所有 `virt-launcher` Pod 都会被跳过，所以默认策略对 VM 是安全的，但无法对其进行再平衡。
- `PodsWithLocalStorage` 必须保持禁用：每个 `virt-launcher` Pod 都包含多个 `emptyDir` 卷（`private`、`public`、`sockets`、`ephemeral-disks`、`container-disks`、`hotplug-disks`，……）。重新启用该保护后，即使没有 PVC，launcher 也无法被驱逐。

要对 VM 进行再平衡，必须在限定为目标 VM 的配置文件中，从 `extraEnabled` 中移除 `PodsWithPVC`。

配置有界 VM 再平衡策略

编辑 ConfigMap 是受支持的配置路径。ACP 会将此编辑记录为插件命名空间中的 `ResourcePatch`（API server 会打印 `Warning: The operation is allowed, and ResourcePatch rp-<suffix> is created`），因此图表协调不会覆盖此更改。每次 CronJob 运行时都会在启动时读取 ConfigMap；无需重启。

首先备份生成的策略：

```
kubectl -n descheduler get configmap descheduler -o yaml \
  > descheduler-policy.before-vm-rebalance.yaml
kubectl -n descheduler edit configmap descheduler
```

以下 `data.policy.yaml` 用于验证运行。它将 Descheduler 限制为测试中的两个工作节点，仅处理测试 VM 的 launcher，并允许每个周期执行一次迁移：

```yaml
apiVersion: "descheduler/v1alpha2"
kind: "DeschedulerPolicy"
nodeSelector: "kubernetes.io/hostname in (<worker-a>,<worker-b>)"
maxNoOfPodsToEvictPerNode: 1
maxNoOfPodsToEvictPerNamespace: 1
maxNoOfPodsToEvictTotal: 1
profiles:
- name: vm-rebalance
  pluginConfig:
  - name: DefaultEvictor
    args:
      nodeFit: true
      labelSelector:
        matchLabels:
          app: desched-test
      podProtections:
        defaultDisabled:
        - PodsWithLocalStorage
  - name: LowNodeUtilization
    args:
      thresholds:
        cpu: 70
        memory: 60
        pods: 20
      targetThresholds:
        cpu: 75
        memory: 68
        pods: 30
  plugins:
    balance:
      enabled:
      - LowNodeUtilization
```

字段说明：

- `nodeSelector` 同时限制被分类的节点和驱逐来源。测试时不要包含 control-plane 节点。
- `DefaultEvictor.args.labelSelector` 限制候选 Pod。VM 的 `spec.template.metadata.labels` 中设置的标签会复制到其 `virt-launcher` Pod，因此只需设置 VM 标签即可限定测试范围。

- `LowNodeUtilization.args.evictableNamespaces` 仅接受 `exclude`。`include` 列表会导致每次运行都因 `only Exclude namespaces can be set, inclusion is not supported` 失败，并且 CronJob 会持续重试失败的 Job，直到修复策略。
- `LowNodeUtilization` 根据资源请求而非使用量对节点进行分类。`virt-launcher` 会请求 VM 内存加上 KubeVirt 开销（对于 512Mi guest，约为 270Mi），以及 100m CPU，除非 VM 设置了 CPU 请求。应从 Descheduler 日志中的分类行（见下文）获取阈值，而不是从 `kubectl top` 获取。
- `maxNoOfPodsToEvict*` 限制（以及设置了 `LowNodeUtilization.args.evictionLimits.node` 时的该限制）会统计后台驱逐请求，因此可以限制每个周期的并发迁移数量。

仅通过此安装的策略 ConfigMap 和 CronJob（或 Deployment）更改行为。

验证重新平衡

在至少有两个可调度工作节点的临时命名空间中运行测试：

1. 使用范围标签创建测试 VM，等待 `LiveMigratable=True`，并记录源节点和 VMI UID：

```
kubectl get vmi -n <vm-namespace> \
  -o custom-columns='NAME:.metadata.name,UID:.metadata.uid,NODE:.statu
s.nodeName,STRATEGY:.spec.evictionStrategy'
```

2. 创建有界且可逆的不平衡状态（例如，使用首选节点亲和性将测试 VM 放置在 一个工作节点上），使一个工作节点高于 `targetThresholds`，另一个低于 `thresholds`。
3. 等待下一次 CronJob 运行（默认每两分钟一次）并读取其 日志：

```
kubectl -n descheduler logs -l app.kubernetes.io/name=descheduler --tai
l=200 \
  | grep -E 'classified|removablePods|Eviction in background|Number of
evictions'
```

成功的请求如下所示：

```
"Node has been classified" category="underutilized" node="<worker-b>" u
sagePercentage={"cpu":66,"memory":57,"pods":16}
"Node has been classified" category="overutilized" node="<worker-a>" us
agePercentage={"cpu":66,"memory":72,"pods":18}
"Pods on node" node="<worker-a>" allPods=46 nonRemovablePods=44 removab
lePods=2
"Eviction in background assumed" pod="<vm-namespace>/virt-launcher-<vm>
-<suffix>"
"Number of evictions/requests" totalEvicted=0 evictionRequests=1
```

`totalEvicted=0 evictionRequests=1` 是预期结果：Pod 未被驱逐，而是请求了迁移。

4. 观察 KubeVirt 完成迁移：

```
kubectl get vmim -n <vm-namespace> \
  -o custom-columns='NAME:.metadata.name,PHASE:.status.phase,VMI:.spec.
vmiName,SOURCE:.status.migrationState.sourceNode,TARGET:.status.migrati
onState.targetNode'
kubectl get vmi -n <vm-namespace> \
  -o custom-columns='NAME:.metadata.name,UID:.metadata.uid,NODE:.statu
s.nodeName,EVACUATION:.status.evacuationNodeName'
kubectl get events -n <vm-namespace> --sort-by=.lastTimestamp | grep -i
-E 'evacuation|migrat'
```

`VirtualMachineInstanceMigration` 与 Descheduler 请求在同一秒创建， 名称为 `kubevirt-evacuation-<suffix>`。一个 512Mi 的测试 VM 约在十秒内完成预拷贝迁移。

5. 仅当 VMI 在不同节点上运行、`VirtualMachineInstanceMigration` 达到 `Succeeded` 且源节点和目标节点符合预期、VMI UID 未改变、 VM 仍处于 `Running` / `Ready` 状态，并且旧 launcher 已终止时，测试才算通过。单独的 Descheduler 日志 行、Eviction API 响应或发生变化的 Pod 名称，都不能作为迁移结果。

Descheduler 会在每次运行时重新评估集群。使用上述限制时， 它每个周期移动一个 VM，直到没有节点低于 `thresholds`，此时 日志会显示 `No node is underutilized, nothing to do here`。如果两个节点 持续在两个类别之间来回切换，说明对于工作负载的大小而言阈值过于接近；应扩大 `thresholds` 与 `targetThresholds` 之间的差距， 而不是降低驱逐限制。

要在迁移进行期间区分由 Descheduler 触发的迁移和由 drain 触发的迁移，请读取 VMI 注解：

```
kubectl get vmi <vmi-name> -n <vm-namespace> \
  -o jsonpath='{.metadata.annotations.kubevirt\.io/eviction-source}
{"\n"}'
```

迁移完成后会移除该注解；之后，将 `kubevirt-evacuation-*` 迁移时间戳与 Descheduler Job 日志中的 `Eviction in background assumed` 行进行关联。

停止并回滚

如果迁移重复发生或一直处于待处理状态，请暂停 CronJob，然后移除策略覆盖。删除 `ResourcePatch` 会在几秒内恢复由 chart 管理的策略；或者重新应用备份文件。

```
kubectl -n descheduler patch cronjob descheduler \
  --type=merge -p '{"spec":{"suspend":true}}'
kubectl -n descheduler get resourcepatch
kubectl -n descheduler delete resourcepatch <rp-name>
kubectl -n descheduler get configmap descheduler -o jsonpath='{.data.poli
cy\.yaml}' | head -5
kubectl get vmim -A
kubectl -n descheduler patch cronjob descheduler \
  --type=merge -p '{"spec":{"suspend":false}}'
```

在重新启用 CronJob 之前，请等待所有 `VirtualMachineInstanceMigration` 对象达到 `Succeeded` 或 `Failed`。

兼容性检查

作为本页面依据的 HCO 兼容性套件会在安装可选插件后断言以下内容：

- 存在带标签的 Descheduler ConfigMap，并且 `data.policy.yaml` 可解析为带有 profile 的 `DeschedulerPolicy`；
- 使用该策略的 CronJob 或 Deployment 会以 `--feature-gates=EvictionsInBackground=true` 渲染（这是针对 package 的回归检查，而非管理员设置）；

- 如果提供了独立的 `KubeDescheduler` CRD，则会独立检查其 profile 和字段 schema；
- KubeVirt HCO 和 VM 驱逐策略路径仍可被发现。

实际的重新平衡结果仍需要一个真实的双工作节点 KubeVirt 集群。 每次运行都记录 ACP release、Kubernetes server version、KubeVirt version、HCO version 和 Descheduler plugin version。如果任何 API 或参数发生变化，请同时更新此页面和 兼容性测试。

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。

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：

```yaml
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 st
arts 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.i
o/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：

```yaml
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 annotati
ons,
  # then negate the ones to drop. On Kube-OVN, drop the per-VM OVN annota
tions
  # (fixed IP / pod-bridge live-migration) so the clone gets a fresh iden
tity:
  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 `events` 和 `status.conditions` 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

本文介绍如何将虚拟机（VM）磁盘从一个 StorageClass 迁移到另一个 StorageClass。当需要将 VM 存储迁移到其他后端、存储池或 CSI 驱动程序时，可以使用此操作步骤，例如从 Ceph RBD 迁移到 CephFS、在两个 Ceph RBD 存储池之间迁移，或从平台提供的存储迁移到客户提供的 StorageClass。

有两种迁移方法可用：

方法	VM 状态	推荐场景
使用 CDI 克隆进行冷迁移	已停止	当源存储或目标存储仅支持 `ReadWriteOnce`、涉及本地存储或拓扑感知型存储，或者最大兼容性比避免停机更重要时使用。
实时存储迁移	运行中	仅当 VM、源存储、目标存储、网络和目标节点调度条件均支持实时迁移时使用。

目录

前提条件
准备迁移
使用 DataVolume 克隆进行冷迁移
停止 VM
创建目标 DataVolume
更新 VM 磁盘引用
启动并验证 VM

在线存储迁移

确认在线迁移就绪

创建迁移对象

故障排除

没有 VM 准备好进行在线存储迁移

在线迁移卡在调度阶段

目标 DataVolume 未完成

在线存储迁移未达到 Completed

迁移资源在清理期间卡住

前提条件

- 已启用 Alauda Virtualization，并包含实时存储迁移支持。
- 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` 或由 PVC 支持的磁盘。
- 目标 StorageClass 具有足够的容量，并支持所需的 `accessModes` 和 `volumeMode`。
- 对于冷迁移，请安排维护时段，因为在克隆磁盘和更新 VM 定义期间，VM 会处于停止状态。
- 对于实时存储迁移，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>
```

在创建目标磁盘前检查目标 StorageProfile：

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

从目标 StorageProfile 支持的声明属性集中选择 `accessModes` 和 `volumeMode`。例如，Ceph RBD 通常与 `volumeMode: Block` 搭配使用，而 CephFS 通常与 `volumeMode: Filesystem` 搭配使用。

对于 `VirtualMachineStorageMigrationPlan`（用于实时存储迁移，本文档后面会对此进行介绍），你还可以将 `accessModes` 或 `volumeMode` 设置为 `Auto`，让存储迁移控制器从 StorageProfile 中查找该值。对于手动创建的 `DataVolume`，请使用明确的 Kubernetes PVC 值，例如 `ReadWriteOnce`、`ReadWriteMany`、`Block` 或 `Filesystem`。

如果计划使用实时存储迁移，请检查正在运行的 VMI 条件：

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

仅当输出包含以下两个条件时，才继续进行实时存储迁移：

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

如果任一条件缺失或设置为 `False`，请使用冷迁移，或先解决报告的限制再重试实时迁移。

使用 **DataVolume** 克隆进行冷迁移

冷迁移是跨 CSI 驱动程序兼容性最高的方法。它会停止 VM，将源 PVC 克隆到使用目标 StorageClass 的新 DataVolume 中，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 版本会警告 `spec.running` 已弃用，建议改用 `spec.runStrategy`。如果 VM 已使用 `spec.running`，请勿在移除 `spec.running` 之前设置 `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：

```yaml
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>
```

应用清单：

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

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

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

对于具有 `volumeBindingMode: WaitForFirstConsumer` 的 StorageClass，DataVolume 可能会保持在 `PendingPopulation`，直到 VM 启动并成为第一个使用者。对于本地存储或拓扑感知型存储，这是预期行为。继续更新 VM，启动 VM，并监控 DataVolume 和 VMI。

> **注意**
>
> 从块模式迁移到文件系统模式时，请求的目标大小应大于源磁盘大小。增加 5% 到 10% 是一个实用的起始值，因为目标文件系统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。

编辑 `spec.dataVolumeTemplates` 和 `spec.template.spec.volumes` 等数组时，请保留所有必需条目。如果 VM 具有其他磁盘、cloud-init 卷、密钥或配置映射，请将它们包含在最终的 VM 规范中，以免被移除。

补丁内容示例：

```
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>: {}
```

应用补丁：

```
kubectl patch vm "$VM" -n "$NAMESPACE" --type=merge --patch-file=<vm-patc
h-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 的状态：

```
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"
```

VM 运行且来宾工作负载检查通过后，只有在确认旧 DataVolume 或 PVC 不再被 VM 引用后，才能将其删除。

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

在线存储迁移

仅当 VM 必须保持运行且满足所有在线迁移前提条件时，才使用在线存储迁移。

确认在线迁移就绪

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上。
- VM 不会在 `.status.conditions` 中报告 `RestartRequired=True`。如果 VM 有需要重启的待处理更改，请在开始在线存储迁移前重启 VM，或解决这些待处理更改。
- 如果集群包含不同的 CPU 型号，VM 使用所有候选节点都支持的 CPU 型号。
- 如果 VM 在 pod 网络上使用 bridge 绑定，则 VM 具有以下注解：

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

创建迁移对象

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

如果使用 Kubernetes manifest，请创建一个 `VirtualMachineStorageMigrationPlan` 和一个 `VirtualMachineStorageMigration`。仅有计划不会执行迁移——迁移对象才是触发器：

```
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 migra
tion;
  # the source storage is NOT auto-deleted — remove it manually once veri
fied.
  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 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 将 VM 磁盘引用更新为目标 DataVolume。
3. KubeVirt 通过在线迁移执行工作负载更新。
4. ACP 清理已完成的工作负载更新资源。
5. `VirtualMachineStorageMigration` 达到 `Completed`。

当迁移达到 `Completed` 时，VM 规范会引用目标 DataVolume，VM 继续在迁移后的磁盘上运行。在删除任何旧存储对象之前，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}{" -> "}{.dat
aVolume.name}{"\n"}{end}'

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

迁移成功且来宾工作负载检查通过后，仅当旧 DataVolume 或 PVC 不再被 VM 或任何回滚操作步骤引用时，才能将其删除。

故障排除

没有 VM 准备好进行在线存储迁移

描述迁移计划：

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

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

如果任一条件为 false，请先使用冷迁移或修复根本原因。同时确认目标 `accessModes` 和 `volumeMode` 受目标 StorageProfile 支持，并确认 VM 未报告 `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、需要重启的待处理 VM 更改，以及节点调度约束。

在线迁移卡在调度阶段

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：

- 无法将目标 launcher pod 调度到不同于源节点的节点上。
- VM 使用的 CPU 模型或 CPU 功能集仅存在于源节点上。
- 无法将目标 PVC 附加到目标节点或挂载到目标节点上。

使用可移植的 CPU 模型，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 事件：

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

对于 `WaitForFirstConsumer` 存储，在消费者 pod 调度之前，出现 `PendingPopulation` 可能是正常现象。对于从块存储迁移到文件系统的场景，如果 CDI 报告目标 PVC 太小，请增大请求的目标大小后重试。

在线存储迁移未达到 **Completed**

如果 `VirtualMachineStorageMigration` 未达到 `Completed`，请检查存储迁移资源和底层 KubeVirt 迁移：

```
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，请继续按照在线迁移问题进行故障排除。如果 `VirtualMachineInstanceMigration` 已成功完成，但存储迁移资源仍未完成，请收集迁移资源、迁移计划、VM、VMI、PVC、DataVolume 和控制器日志，然后联系支持团队。

迁移资源在清理期间卡住

首先删除迁移资源：

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

如果 finalizer 阻止删除，请仅在确认没有正在运行的活动迁移后移除它：

```
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"}]'
```

在因在线存储迁移中断而重启 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}{" -> "}{.dat
aVolume.name}{"\n"}{end}'
```

如果 VM 规范已更新但迁移未完成，请先完成迁移，或者将 VM 磁盘引用恢复为预期的 DataVolume，然后再启动 VM。

KubeVirt™ 是 The Linux Foundation 的商标。Alauda 是独立供应商。本产品与 The Linux Foundation 没有关联，也未获得其认可或赞助。所有商标均归其各自所有者所有，此处使用商标仅用于标识目的。

配置虚拟机的高可用性

目录

运行策略和驱逐策略

概述

术语表

组件概述

隔离和修复期间的事件流

操作步骤

Operator 列表

部署 Node Remediation Operator

配置 Node Remediation Operator（可选）

配置 Node Remediation 模板（可选）

部署 Node Fault Detection Operator

创建 NodeHealthCheck 实例

验证（可选）

运行策略和驱逐策略

虚拟机规范中的两个字段决定了虚拟机在发生故障和进行节点维护期间的行为：

- 运行策略（`spec.runStrategy`）控制所需的电源状态和重启行为：`Always`（保持运行，停止后重启）、`RerunOnFailure`（仅在发生故障后重启）、`Manual`（状态仅由启动/停止/重启控制）和 `Halted`（保持停止）。较旧的 ACP 虚拟机可能仍使用已弃用的 `spec.running` 布尔值（`true` / `false`）；请勿在同一虚拟机上同时设置这两个字段。
- 驱逐策略（`spec.template.spec.evictionStrategy`）控制节点因维护或升级而进行排空时的行为：
  - `LiveMigrate` — 将虚拟机实时迁移到其他节点，而不是将其终止。
  - `LiveMigrateIfPossible` — 如果虚拟机支持迁移，则执行实时迁移；否则按 `None` 的方式处理。
  - `None` — 不采取特殊操作（虚拟机会随节点一起关闭）。

  `LiveMigrateIfPossible` 是推荐值，也是 operator bundle 在集群级别提供的值。使用 `evictionStrategy: LiveMigrate` 的不可迁移虚拟机（例如使用 RWO 存储或直通设备的虚拟机）会阻止节点排空，因此也会阻止集群升级，直到手动停止或迁移该虚拟机；实际上，虚拟机可能因多种原因变得不可迁移。`LiveMigrateIfPossible` 可确保维护继续进行：可迁移的虚拟机会在不中断的情况下迁移，只有其余虚拟机会在另一个节点上重启。仅对绝不能停止的虚拟机选择 `LiveMigrate`，并接受它可能阻止节点维护。无论采用哪种方式，都应为必须在维护期间保持运行且无需重启的虚拟机规划共享存储（请参阅在存储类之间迁移虚拟机）。

有关 operator 工作流，请参阅 KubeVirt 节点维护。

概述

硬件并非完美无缺，软件也包含错误。当发生节点级故障（例如内核挂起或网络接口控制器（NIC）发生故障）时，集群所需执行的工作量并不会减少，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）可能要求至多一个实例运行。

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。因此，在启动工作负载恢复（称为修复）之前，务必确保节点达到安全状态（称为隔离）；理想情况下，还应恢复节点本身。

依赖管理员干预来确认节点和工作负载的真实状态并不总是可行。为便于执行此类干预，Alauda Container Platform 提供了多个用于自动执行故障检测、隔离和修复的组件。

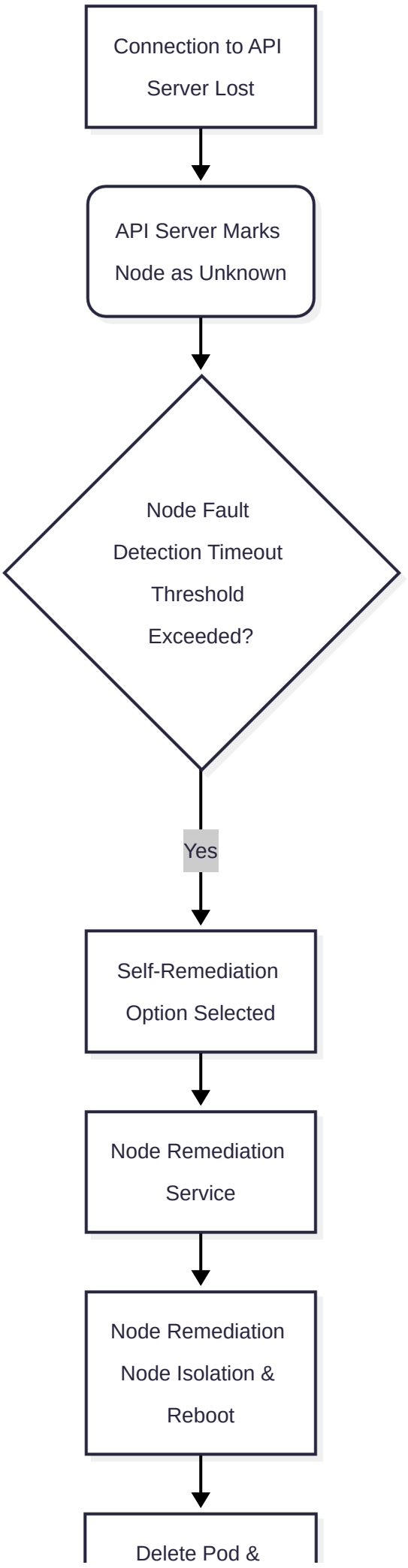
术语表

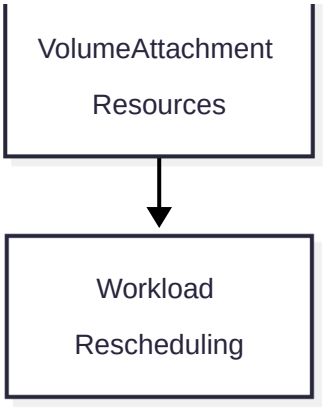
简称	产品
节点修复	Alauda Container Platform Node Remediation
节点故障检测	Alauda Container Platform Node Fault Detection

组件概述

- **Node Remediation Operator**
  Node Remediation Operator（Alauda Container Platform Node Remediation，基于 SelfNodeRemediation 项目）是 Alauda Container Platform 的附加 Operator，实现了外部隔离和修复系统，可重启运行状况不佳的节点并删除 Pod 和 VolumeAttachments 等资源。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，Node Remediation 不需要任何管理接口，例如 Intelligent Platform Management Interface（IPMI），也不需要用于节点置备的 API。
  Node Remediation 可供 Machine Health Check 或 Node Fault Detection 等故障检测系统使用。
- **Node Fault Detection Operator**
  Node Fault Detection Operator（Alauda Container Platform Node Fault Detection，基于 NodeHealthCheck 项目）是 Alauda Container Platform 的附加 Operator，实现了监控节点状况的故障检测系统。它没有内置的隔离或修复系统，因此必须配置提供这些功能的外部系统。默认情况下，它配置为使用 Node Remediation 系统。

隔离和修复期间的事件流


Connection to API Server Lost
API Server Marks Node as Unknown
Node Fault Detection Timeout Threshold Exceeded?
Yes
Self-Remediation Option Selected
Node Remediation Service
Node Remediation Node Isolation & Reboot
Delete Pod &



操作步骤

1 Operator 列表

- 下载与平台架构对应的 **Alauda Container Platform Node Remediation** 安装包。
- 使用“上架软件包”机制上架 **Alauda Container Platform Node Remediation** 安装包。
- 下载与平台架构对应的 **Alauda Container Platform Node Fault Detection** 安装包。
- 使用“上架软件包”机制上架 **Alauda Container Platform Node Fault Detection** 安装包。

2 部署 Node Remediation Operator

1. 登录并进入管理员页面。
2. 单击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Container Platform Node Remediation**，单击安装，进入 **Install Alauda Container Platform Node Remediation** 页面。
   配置参数：

参数	建议配置
Channel	默认频道为 `stable`。
Installation Mode	`Cluster`：集群中的所有命名空间共享一个 Operator 实例来创建和管理资源，从而降低资源使用量。

参数	建议配置
Installation Place	选择 `Recommended`，Namespace 仅支持 **workload-availability**。
Upgrade Strategy	`Manual`：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 Node Remediation Operator（可选）

Node Remediation Operator 会创建名为 `self-node-remediation-config` 的 `SelfNodeRemediationConfig` CR。该 CR 创建在 Node Remediation Operator 所在的命名空间中。

> **Note**
>
> 更改 `SelfNodeRemediationConfig` CR 会重新创建 Node Remediation daemon set。

`SelfNodeRemediationConfig` CR 的内容类似于以下 YAML 文件：

```yaml
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	说明
safeTimeToAssumeNodeRebootedSeconds	指定一个可选的时间段，Operator 在恢复运行于运行状况不佳节点上的受影响工作负载之前等待。若在故障节点上的 Pod 仍在运行时启动替换 Pod，可能会导致数据损坏并违反只运行一次语义。Operator 会根据 ApiServerTimeout、ApiCheckInterval、MaxApiErrorThreshold、PeerDialTimeout 和 PeerRequestTimeout 字段的值，以及修复时的 watchdog 超时时间和集群规模，计算最短等待时间。

参数	说明
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，Node Remediation Operator 会自动检测 softdog 设备路径。 如果 watchdog 设备不可用，`SelfNodeRemediationConfig` CR 会使用软件重启。
isSoftwareRebootEnabled	指定是否启用对运行状况不佳节点的软件重启。默认情况下，`isSoftwareRebootEnabled` 的值为 `true`。要禁用软件重启，请将参数值设为 `false`。
apiServerTimeout	指定检查与每个 API server 的连接情况时的超时时间。超时后，Operator 会启动修复。超时时间必须大于或等于 10 毫秒。
apiCheckInterval	指定检查与每个 API server 的连接情况的频率。超时时间必须大于或等于 1 秒。
maxApiErrorThreshold	指定阈值。达到此阈值后，节点会开始联系其对等节点。阈值必须大于或等于 1 秒。
peerApiServerTimeout	指定对等节点连接 API server 的超时时间。超时时间必须大于或等于 10 毫秒。
peerDialTimeout	指定与对等节点建立连接的超时时间。超时时间必须大于或等于 10 毫秒。
peerRequestTimeout	指定从对等节点获取响应的超时时间。超时时间必须大于或等于 10 毫秒。
peerUpdateInterval	指定更新对等节点信息（如 IP 地址）的频率。超时时间必须大于或等于 10 秒。
hostPort	指定一个可选值，以更改 Node Remediation agent 用于内部通信的端

参数	说明
	口。该值必须大于 0。默认端口为 30001。
customDsTolerations	指定自定义容忍度，使运行在 DaemonSet 上的 Node Remediation agent 支持修复不同类型的节点。

Note

- 默认情况下，Node Remediation Operator 会在部署命名空间中创建 CR。
- 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 Node Remediation。

4 配置 Node Remediation 模板（可选）

Node Remediation Operator 还会创建 SelfNodeRemediationTemplate Custom Resource Definition（CRD）。此 CRD 定义节点的修复策略，旨在更快地恢复工作负载。可用的修复策略如下：

- **Automatic**
  此修复策略通过让 Node Remediation Operator 为集群选择最合适的修复策略，简化修复流程。该策略会检查集群中是否可用 `OutOfServiceTaint` 策略。如果 `OutOfServiceTaint` 策略可用，Operator 会选择 `OutOfServiceTaint` 策略。如果 `OutOfServiceTaint` 策略不可用，Operator 会选择 `ResourceDeletion` 策略。`Automatic` 是默认修复策略。
- **ResourceDeletion**
  此修复策略会删除节点上的 Pod，而不是删除节点对象。
- **OutOfServiceTaint**
  此修复策略通过在节点上设置 `OutOfServiceTaint` 策略，隐式删除节点上的 Pod 和关联的 volume attachment，而不是删除节点对象。

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

SelfNodeRemediationTemplate CR 的内容类似于以下 YAML 文件：

```yaml
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	说明
remediation_strategy	值：Automatic、ResourceDeletion、OutOfServiceTaint

5 部署 Node Fault Detection Operator

1. 登录并进入管理员页面。
2. 单击 **Marketplace** > **OperatorHub**，进入 **OperatorHub** 页面。
3. 找到 **Alauda Container Platform Node Fault Detection**，单击安装，进入 **Install Alauda Container Platform Node Fault Detection** 页面。
   配置参数：

参数	建议配置
Channel	默认频道为 `stable`。
Installation Mode	`Cluster`：集群中的所有命名空间共享一个 Operator 实例来创建和管理资源，从而降低资源使用量。

参数	建议配置
Installation Place	选择 `Recommended`，Namespace 仅支持 **workload-availability**。
Upgrade Strategy	`Manual`：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

在集群控制平面节点上执行以下命令：

Command

```
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
```

Example

```
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	说明
name	资源名称
minHealthy	指定健康节点的最低比例。只有当健康节点的比例大于或等于此值时，才会修复故障节点。默认值为 51%
selector	指定 LabelSelector，以匹配要检查并自动修复的节点。请避免在同一个实例中同时指定控制平面节点和工作节点

7 验证（可选）

> 模拟虚拟机运行所在节点发生故障，并确认虚拟机是否会自动调度到其他节点上运行。

Kubernetes® 是 The Linux Foundation 在美国及其他国家/地区的注册商标。 NodeHealthCheck 和 SelfNodeRemediation 是 medik8s Community 的项目。 Alauda 是一家独立供应商。本产品与 The Linux Foundation、medik8s 项目无关联，也未获得其认可或赞助。所有商标均为其各自所有者的财产，在此仅用于识别。

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：

```yaml
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: cephrbd
    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

节点故障时的 VM 迁移和恢复

计划内维护

从意外的节点故障中恢复

热迁移错误消息与解决方案

启动和镜像导

读取来宾启动日

诊断卡住的镜像

捕获虚拟机内存

节点故障时的 VM 迁移和恢复

有两种不同的节点事件，需要采用不同的操作步骤：

- 对于计划内维护，首先对节点执行 cordon，然后在节点仍处于正常状态时对其 VM 执行实时迁移。请参阅 KubeVirt 节点维护。
- 对于意外的节点故障，首先隔离节点。KubeVirt 无法区分已断电的节点和临时网络分区， 因此在隔离节点前重新创建 VM 可能会导致同一来宾系统运行两个副本。

不要将故障操作步骤用于计划内排空，也不要使用计划内排空来恢复已无法访问的节点。

目录

计划内维护

从意外的节点故障中恢复

1. 保留诊断信息
2. 隔离故障节点
3. 执行已批准的修复操作
4. 验证一个替代 VMI

计划内维护

node-maintenance 操作指南是执行平稳变更窗口的权威操作步骤。它会检查容量和 `LiveMigratable=True`，对源节点执行 cordon，逐个迁移 VM，并在节点排空前验证

VMIM、VMI 和 launcher。特别是，不要为了使排空成功返回而强制删除 launcher Pod。

从意外的节点故障中恢复

1. 保留诊断信息

在更改资源前，记录节点、VMI、launcher Pod 和卷附件：

```
kubectl get node <failed-node> -o yaml
kubectl get vmi -A -o wide
kubectl get pod -A -l kubevirt.io=virt-launcher -o wide
kubectl get volumeattachment -o wide
kubectl get events -A --sort-by=.lastTimestamp
```

2. 隔离故障节点

使用已批准的带外电源隔离方式，或集群的节点修复堆栈（例如，已配置的 Node Fault Detection/Node Remediation 工作流）。在恢复 VM 前，确认旧节点无法运行来宾系统，也无法写入其存储。

不要将 `NotReady`、丢失的 API 连接或过期的 Pod 阶段视为来宾系统已停止的证据。 在隔离节点前，不要删除 `virt-launcher` Pod 或 `VolumeAttachment`。

3. 执行已批准的修复操作

隔离节点后，按照已安装的 Kubernetes、CSI 和 node-remediation 版本对应的修复运行手册执行操作。 该运行手册可能会应用 `node.kubernetes.io/out-of-service` 污点或删除过期资源，以便卸离卷。 具体操作取决于存储驱动程序，不得使用未经审核的 `kubectl delete` 命令替代。

4. 验证一个替代 VMI

等待 VM 控制器在健康节点上创建一个替代 VMI。验证 VM 仍处于预期的运行状态，新 VMI 为 `Ready`，并且只有一个活动 launcher：

```
kubectl get vm -A -o wide
kubectl get vmi -A -o wide
kubectl get pod -A -l kubevirt.io=virt-launcher -o wide
kubectl get volumeattachment -o wide
```

如果 VMI 仍处于卡住状态，请保留其事件和条件消息，并返回隔离/修复运行手册。不要将 `--force` 或 `--grace-period=0` 作为第一响应；这些选项可能会导致 VM 脑裂，或使存储处于不安全状态。

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 `metadata.annotations` 和 `spec.template.metadata.annotations` 字段中是否存在 `"kubevirt.io/allow-pod-bridge-network-live-migration": "true"` 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 Runni
ng=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。

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

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 NIC
前提条件
操作步骤

介绍

Alauda Virtualization 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
  完整的 IPv6 支持。
- 静态 IP 保留
  确保 VM 重启后仍保留相同的 IP 地址，符合传统 VM 的使用模式。
- 多网络模式支持
  支持容器网络和 SR-IOV 等多种网络模式，满足不同用户场景的需求。

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

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

前提条件

操作步骤

配置多 NIC

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 `IP/32` 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 `curl -vvv <domain>` 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：

```yaml
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 qu
iet 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<glo
bal>
        inet6 fe80::a236:9fff:fe29:6c00  prefixlen 64  scopeid 0x2
0<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 0x2
0<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=eno
1 drv=tg3 unused=vfio-pci
0000:01:00.1 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=eno
2 drv=tg3 unused=vfio-pci
0000:02:00.0 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=eno
3 drv=tg3 unused=vfio-pci
0000:02:00.1 'NetXtreme BCM5720 Gigabit Ethernet PCIe 165f' if=eno
4 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。

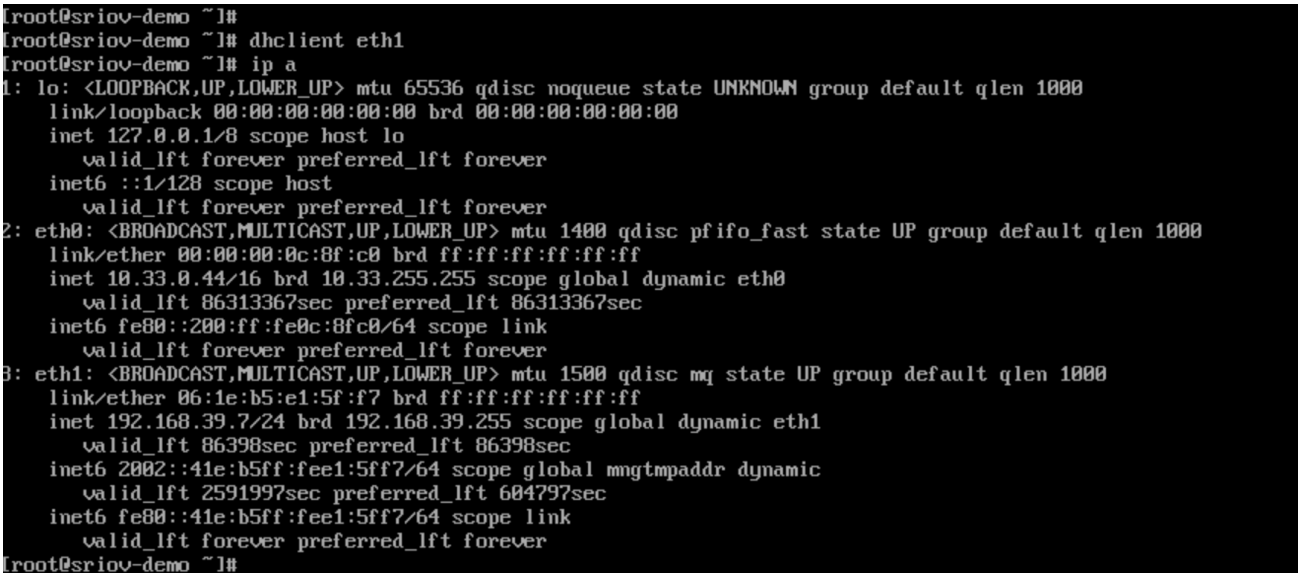
12 配置 IP 地址

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。



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 NIC 虚拟机

结合使用 Kube-OVN 和 Multus，为虚拟机提供多 NIC 支持

目录

前提条件
操作步骤
创建辅助网络
创建多 NIC 虚拟机
配置新 NIC 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 或更高版本。
- 使用 Kube-OVN 作为 CNI。
- 已安装 Alauda Container Platform Multi-Network

操作步骤

1 创建辅助网络

1. 创建 NetworkAttachmentDefinition

   在集群控制平面节点上执行以下命令：

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：

- name：NetworkAttachmentDefinition 的名称。
- namespace：NetworkAttachmentDefinition 的命名空间。必须使用与虚拟机相同的命名空间。
- provider：当前 NetworkAttachmentDefinition 的 `<name>.<namespace>.ovn`。Kube-OVN 将使用此信息查找对应的 Subnet 资源。请注意，后缀必须设置为 ovn。

2. 创建 Kube-OVN Subnet
   如果将 Kube-OVN 用作辅助网络接口，则 provider 应设置为对应 NetworkAttachmentDefinition 的 `<name>.<namespace>.ovn`，并且必须以 ovn 作为后缀。
   在集群控制平面节点上执行以下命令：

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：

- name：子网的名称。
- provider：NetworkAttachmentDefinition 的 provider。
- cidrBlock：子网 CIDR。
- gateway：网关地址。
- excludeIps：不会自动分配的预留 IP 集合。例如，可将其用作计算组件的固定 IP 地址。

2 创建多 NIC 虚拟机

1. 通过 UI 创建虚拟机
2. 切换到 **YAML** 视图，并向虚拟机添加另一个 NIC
   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
          # new interface
          - bridge: {}
            name: dyniface1
      networks:
      - name: default
        pod: {}
      # new network
      - multus:
          networkName: <networkName>
        name: dyniface1
```

networkName 是 NetworkAttachmentDefinition 的名称。

3 配置新 NIC 的网络

虚拟机启动后，需要进入虚拟机并手动为新添加的 NIC 配置网络。

4 热插拔网络接口

支持在运行中的虚拟机上热插拔网络接口

对于使用 virtio 模型并通过 bridge 绑定或 SR-IOV 绑定连接的接口，支持热插入。

仅对于通过 bridge 绑定连接的接口，支持热拔出。

1. 向运行中的 VM 添加接口
   使用 kubectl edit 修改虚拟机的 YAML 配置

```
spec:
  template:
    spec:
      domain:
        devices:
          interfaces:
          - bridge: {}
            name: default
          # new interface
          - bridge: {}
            name: dyniface1
      networks:
      - name: default
        pod: {}
      # new network
      - multus:
          networkName: <networkName>
        name: dyniface1
```

2. 从运行中的 VM 移除接口

   使用 kubectl edit 修改虚拟机的 YAML 配置

```
spec:
  template:
    spec:
      domain:
        devices:
          interfaces:
          - bridge: {}
            name: default
          # set the interface state to absent
          - bridge: {}
            name: dyniface1
            state: absent
      networks:
      - name: default
        pod: {}
      # new network
      - 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

Alauda Virtualization Storage 通过无缝集成 Kubernetes 原生存储机制，为虚拟机（VM）提供持久化存储能力。它利用 **PersistentVolumeClaim**（PVC）存储 VM 磁盘数据，并使用 **Container Storage Interface**（CSI）与各种存储系统集成。此外，还使用 **Containerized Data Importer**（CDI）初始化 VM 磁盘数据。在此基础上，该平台扩展了 VM 磁盘管理的高级功能，实现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 VM 磁盘操作都可以通过 **Web UI** 轻松完成，从而最大限度地减少对 CLI 专业知识的需求。
- VM 磁盘生命周期管理
  配置关联的 VM 终止时是否应自动删除 VM 磁盘。

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

数据磁盘可用于满足业务的数据持久化需求。

数据磁盘由 CDI `DataVolume` 提供支持（它会创建同名的 PVC）。以下操作分别对应：创建 `DataVolume`，将其热插拔到虚拟机（`addvolume`），扩展其 PVC，将其卸载（`removevolume`），以及删除 `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

StorageProfile 和跨命名空间克隆范围

创建虚拟磁盘

为虚拟机创建数据磁盘。每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：虚拟机处于运行状态时，可以在线挂载虚拟磁盘。

操作步骤

1. 进入容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟磁盘。
3. 点击创建虚拟磁盘。
4. 按照以下说明配置信息。

参数	描述
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。
存储类	平台通过自动创建和管理持久卷声明来维护虚拟机磁盘。你需要指定动态创建持久卷声明所需的存储类。 不同的存储类支持不同的卷模式。如果所选卷模式没有可用的存储类，请联系管理员添加。 磁盘的访问模式遵循存储类的 StorageProfile：对于所选卷模式，如果 StorageProfile 提供 **ReadWriteMany**，则使用该访问模式（实时迁移需要此模式）；否则使用 StorageProfile 提供的访问模式。
随 **VM** 删除	启用后，删除虚拟机时也会删除磁盘数据。

参数	描述
挂载	- 不挂载：仅创建虚拟磁盘；需要时可稍后挂载。 - 挂载到 **VM**：选择要挂载虚拟磁盘的目标虚拟机。

5. 点击创建。

使用 API

将空白数据磁盘创建为 `DataVolume`：

```yaml
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

将数据磁盘挂载到虚拟机，将已创建的虚拟磁盘附加到目标虚拟机。

注意：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，点击虚拟化 > 虚拟磁盘。
3. 在要挂载的虚拟磁盘旁，点击⋮> 挂载。
4. 选择目标虚拟机，然后点击挂载。

使用 API

将磁盘热插拔到正在运行的虚拟机中。使用 `virtctl` 时，使用 `--persist`，使磁盘添加到虚拟机规格中，并在重启后保留：

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

此操作会发出 `addvolume` 子资源请求（`AddVolumeOptions`）。随后，磁盘会显示在 `VirtualMachineInstance.status.volumeStatus` 中，并从 `AttachedToNode` 变为 `Ready`。如果不使用 `--persist`，热插拔只会应用于正在运行的实例，重启后不会保留。

扩展虚拟磁盘

扩展已挂载到虚拟机的系统磁盘和数据磁盘。

操作步骤

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

使用 API

修补 PVC 请求的大小（StorageClass 必须允许扩展卷）：

```
kubectl patch pvc datadisk-1 -n demo --type merge \
  -p '{"spec":{"resources":{"requests":{"storage":"20Gi"}}}}'
```

`status.capacity.storage` 会扩展到新大小；Ceph RBD 等 CSI 驱动支持在线扩展（无需重启）。

卸载虚拟磁盘

从虚拟机卸载数据磁盘；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，点击虚拟化 > 虚拟磁盘。
3. 在要卸载的虚拟磁盘旁，点击⋮> 卸载并确认。

使用 API

分离热插拔磁盘：

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

此操作会发出 `removevolume` 子资源请求，并分离磁盘（该条目会从 `VirtualMachineInstance.status.volumeStatus` 中移除）。KubeVirt API 支持从正在运行的虚拟机中分离磁盘，但请确保在移除磁盘前，客户机内部不再使用该磁盘（已在操作系统级别卸载），以避免数据丢失。

删除虚拟磁盘

只有处于未挂载状态的虚拟磁盘才能删除。

注意：系统磁盘不能删除。

操作步骤

1. 访问容器平台。
2. 在左侧导航栏中，点击虚拟化 > 虚拟磁盘。
3. 在要删除的虚拟磁盘旁，点击⋮> 删除并确认。

使用 API

```
kubectl delete datavolume datadisk-1 -n demo
```

后端 PVC 由 `DataVolume` 所有，并会随其一同进行垃圾回收。如果 CDI 已经对 `DataVolume` 执行了垃圾回收，请直接删除 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`，用于描述如何在该后端置备和克隆磁盘。检查该对象可查看支持的访问模式、卷模式和克隆方法：

```
kubectl get storageprofile rbd-1 -o jsonpath='cloneStrategy={.status.clon
eStrategy} claimPropertySets={.status.claimPropertySets}{"\n"}'
# cloneStrategy=csi-clone claimPropertySets=[{RWX,Block},{RWO,Block},{RW
O,Filesystem}]
```

`cloneStrategy`（`csi-clone`、`snapshot` 或主机辅助的 `copy`）决定可启动卷或虚拟机克隆如何复制磁盘；快照或克隆能否执行，取决于 StorageClass 是否具备这些功能。

跨命名空间克隆范围：创建虚拟机或可启动卷时，控制台仅提供来自当前命名空间和 **`kube-public`** 的启动镜像作为克隆源。此 UI 防护措施（而非 RBAC）是此平台支持的边界：将共享

的黄金镜像发布到 `kube-public`，内置的 `os-images.kubevirt.io:view` 角色可让每位经过身份验证的用户克隆这些镜像。

> CDI 上游的 `datavolumes/source` 权限（`datavolume-cloner` ClusterRole）无法可靠地限制 CSI 路径克隆，因此绑定该权限并非在任意非公开命名空间中公开镜像的受支持或充分方式。共享镜像请使用 `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

Alauda Virtualization 提供虚拟机快照功能，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、人为错误（例如意外删除文件）或恶意攻击（例如勒索软件）而发生数据丢失时，快照可作为恢复运行的最后一道防线。

使用限制

- 虚拟机运行（在线）或停止时均可创建快照。要创建应用一致性快照，请在客户机中安装 QEMU guest agent；否则，在线快照为崩溃一致性快照。回滚快照前，必须先停止虚拟机。

- 虚拟机磁盘必须使用支持卷快照的存储类（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`：

```yaml
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='phase={.status.phase} readyToUse={.status.readyToUse} {.st
atus.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后，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：

```yaml
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
```

```bash
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              BackupIte
mAction
kubevirt-velero-plugin/backup-datavolume-action              BackupIte
mAction
kubevirt-velero-plugin/backup-virtualmachine-action          BackupIte
mAction
kubevirt-velero-plugin/backup-virtualmachine-action          BackupIte
mAction
kubevirt-velero-plugin/backup-virtualmachineinstance-action  BackupIte
mAction
kubevirt-velero-plugin/backup-virtualmachineinstance-action  BackupIte
mAction
kubevirt-velero-plugin/restore-pod-action                    RestoreIt
emAction
kubevirt-velero-plugin/restore-pod-action                    RestoreIt
emAction
kubevirt-velero-plugin/restore-pvc-action                    RestoreIt
emAction
kubevirt-velero-plugin/restore-pvc-action                    RestoreIt
emAction
kubevirt-velero-plugin/restore-vm-action                     RestoreIt
emAction
kubevirt-velero-plugin/restore-vm-action                     RestoreIt
emAction
kubevirt-velero-plugin/restore-vmi-action                    RestoreIt
emAction
kubevirt-velero-plugin/restore-vmi-action                    RestoreIt
emAction
```

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：

```yaml
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                          ERRORS   WARNINGS   CREAT
ED                          EXPIRES   STORAGE LOCATION   SELECTOR
example-backup   WaitingForPluginOperations   0        0          2025-
01-25 14:14:08 +0000 UTC   29d       default            <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   Completed   0        0          2025-01-25 14:14:08 +0
000 UTC   29d       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 res
tore 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-7lfcm   Completed   11m       21474836480   21474836480
default            15m   192.168.254.66
example-restore-cjcd8   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：

```yaml
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。