SUSE Edge 文档

SUSE Edge 文档

出版日期：2025 年 7 月 9 日

https://documentation.suse.com

目录

SUSE Edge 3.3.1 文档

欢迎阅读 SUSE Edge 文档，在这里可以找到总体体系结构概述、快速入门指南、经过验证的设计、组件用法指南、第三方集成，以及有关管理边缘计算基础架构和工作负载的最佳实践。

1 什么是 SUSE Edge?

SUSE Edge 是有针对性的、紧密集成且经过全面验证的端到端解决方案，用于解决在边缘处部署基础架构和云原生应用程序时存在的独特挑战。其核心作用是提供一个有主见但高度灵活、高度可缩放且安全的平台，涵盖初始部署映像的构建、节点置备和初始配置、应用程序部署、可观测性和完整生命周期操作。该平台从一开始就构建于同类最佳的开源软件基础之上，传承了我们 30 年来提供安全、稳定且经认证的 SUSE Linux 平台的历史，继续通过 Rancher 产品组合提供高度可缩放且功能丰富的 Kubernetes 管理。SUSE Edge 基于这些功能构建，可以提供满足众多细分市场需求的功能，包括零售、医疗、交通、物流、电信、智能制造和工业物联网 (IoT)。

2 设计理念

该解决方案的设计考虑到了客户的需求和期望千差万别，因此不存在“以一应百”的边缘平台。边缘部署促使我们解决并不断设想出一些极具挑战性的问题，包括大规模可伸缩性、网络受限情况下的可用性、物理空间限制、新的安全威胁和攻击途径、硬件体系结构和系统资源的差异、部署旧式基础架构和应用程序并与之连接的要求，以及使用寿命较长的客户解决方案。由于其中的许多挑战与传统思维方式不同（例如在数据中心或公有云中部署基础架构和应用程序），我们必须更周密地审视设计，并反复思考许多常见假设条件。

例如，我们发现极简主义、模块化和易操作性具有重要价值。极简主义对于边缘环境非常重要，因为系统越复杂，就越容易出现故障。在分析数百乃至数十万个位置后，我们发现复杂的系统会出现纷繁复杂的故障。解决方案中的模块化允许用户做出更多选择，同时消除部署的平台中不必要的复杂性。我们还需要在这些方面与易操作性之间取得平衡。人类用户在重复某个流程数千次后可能会出现失误，因此平台应确保任何潜在失误都可恢复，从而消除技术人员亲临现场解决问题的需要，同时尽力实现一致性和标准化。

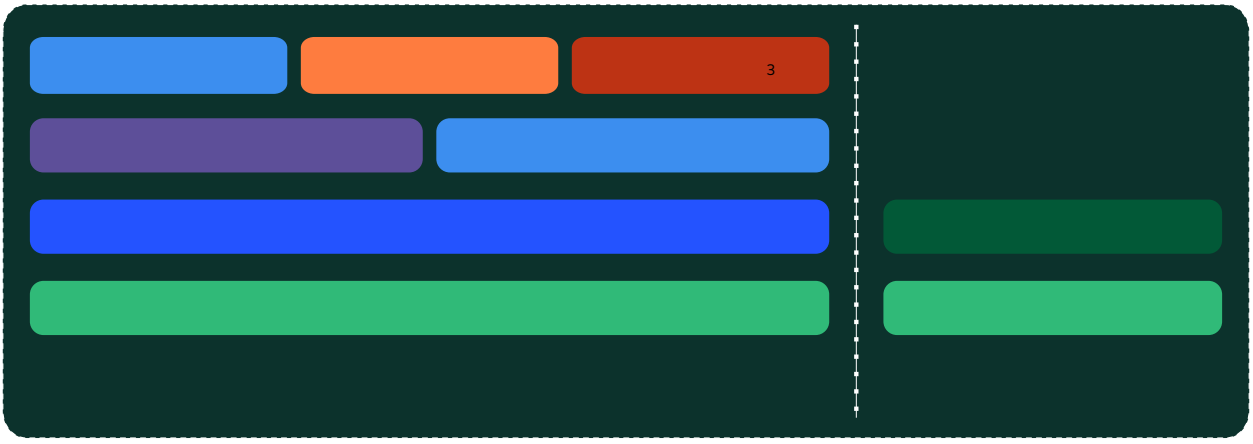
3 总体体系结构

SUSE Edge 的总体系统体系结构分为两个核心类别，即“管理”群集和“下游”群集。管理群集负责一个或多个下游群集的远程管理。不过，在某些情况下，下游群集需要在没有远程管理的情况下运行，例如在边缘站点没有外部连接并且需要独立运行的情况下。在SUSE Edge 中，用于管理群集和下游群集运维的技术组件在很大程度上是相同的，只是在系统规范和系统使用的应用程序方面可能有所不同：管理群集运行能够实现系统管理和生命周期操作的应用程序，而下游群集则是满足各项要求以支持用户应用程序的运行。

3.1 SUSE Edge 中使用的组件

SUSE Edge 由现有的 SUSE 和 Rancher 组件以及 Edge 团队构建的其他功能和组件组成，能帮助我们解决边缘计算中存在的限制和复杂性。以下是对管理群集和下游群集中所用组件的说明，并附有总体体系结构概要图（请注意，其中并未列举出所有组件）：

3.1.1 管理群集



- **管理**：SUSE Edge 的中心部分，用于管理所连下游群集的置备和生命周期。管理群集通常包括以下组件：

- 通过 Rancher Prime（第 5 章 “Rancher”）实现的多群集管理，为下游群集的初始配置以及基础架构和应用程序的持续生命周期管理提供了一个通用仪表板，还提供了全面的租户隔离和 IDP（身份提供程序）集成、第三方集成和扩展的巨大市场，以及不限供应商的 API。

- 通过 SUSE Multi-Linux Manager 实现的 Linux 系统管理，支持对下游群集上运行的底层 Linux 操作系统 *SUSE Linux Micro（第 9 章 “SUSE Linux Micro”）执行自动化的 Linux 补丁和配置管理。请注意，虽然此组件已容器化，但它目前需要在与其他管理组件分离的单独系统上运行，因此在上图中标记为“Linux 管理”。

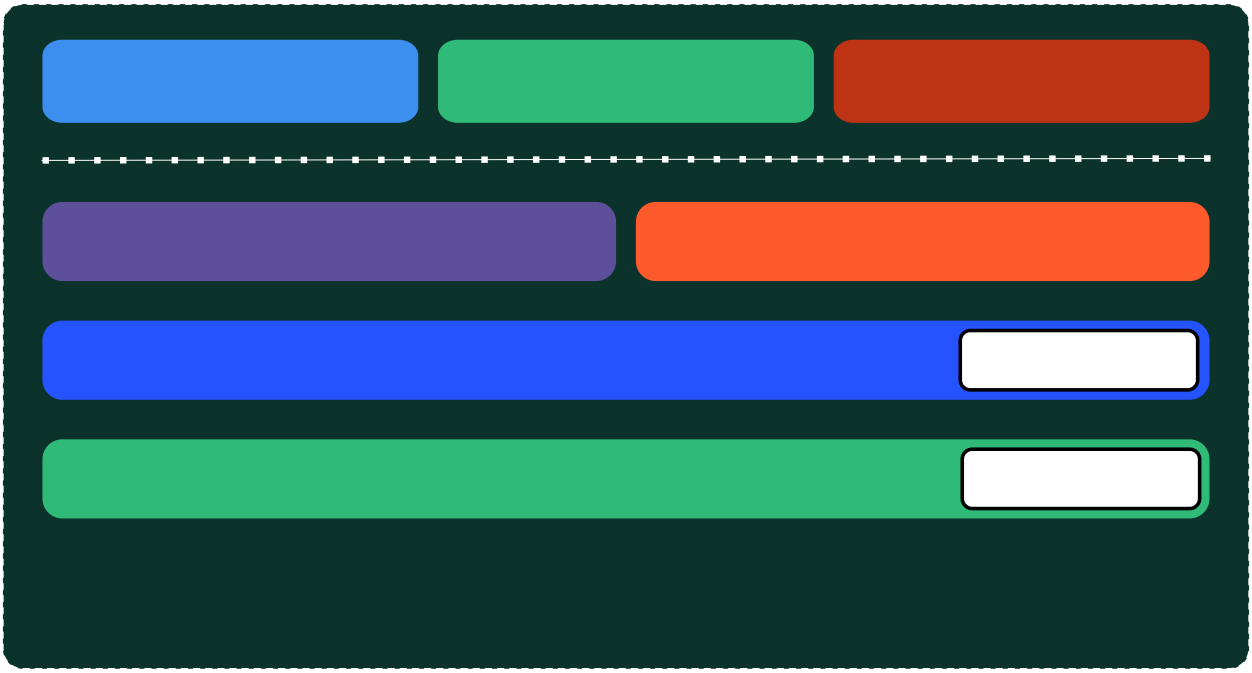
- 专用生命周期管理控制器（第 23 章 “升级控制器”），用于处理将管理群集组件升级到指定 SUSE Edge 版本的相关事务。

- 借助 Elemental（第 13 章 “Elemental”）实现的远程系统接入 Rancher Prime，支持将连接的边缘节点与所需的 Kubernetes 群集和应用程序部署进行后期绑定（例如通过 GitOps）。

- 可选的完整裸机生命周期与管理支持，通过 Metal3（第 10 章 “Metal^3”）、MetalLB（第 19 章 “MetalLB”）和 CAPI (Cluster API) 基础架构提供程序实现，支持对具有远程管理功能的裸机系统执行完整的端到端置备。

- 名为 Fleet（第 8 章 “Fleet”）的可选 GitOps 引擎，用于管理下游群集和其上安装的应用程序的置备和生命周期。
- 支撑管理群集本身的是作为基础操作系统的 SUSE Linux Micro（第 9 章 “SUSE Linux Micro”），以及作为支持管理群集应用程序的 Kubernetes 发行版的 RKE2（第 16 章 “RKE2”）。

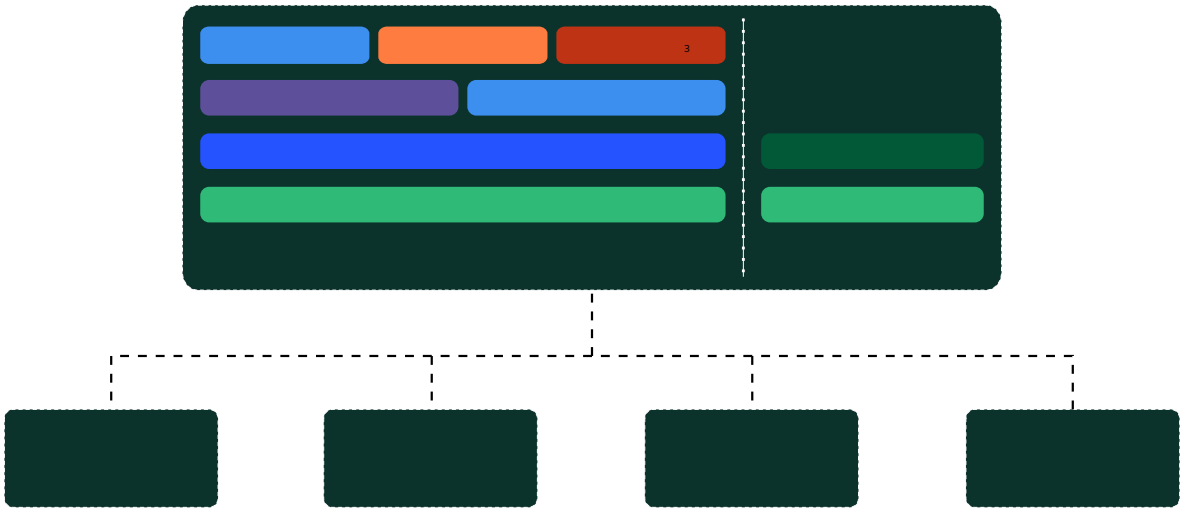
3.1.2 下游群集



- **下游**：SUSE Edge 的分布式组件，用于在边缘运行用户工作负载，即边缘位置运行的软件，通常由以下组件组成：
  - 多种可供选择的 Kubernetes 发行版，包含 K3s（第 15 章 “K3s”）和 RKE2（第 16 章 “RKE2”）等安全的轻量级发行版（RKE2 已针对政府用途和受监管行业进行强化、认证和优化）。
  - SUSE Security（第 18 章 “SUSE Security”），提供映像漏洞扫描、深度数据包检测、实时威胁和漏洞防护等安全功能。

- 借助 SUSE Storage（第 17 章 “SUSE Storage”）实现的软件块存储，提供轻量级、持久化、弹性佳且可扩缩的块存储服务。
- 轻量级、容器优化且经过强化的 Linux 操作系统 SUSE Linux Micro（第 9 章 “SUSE Linux Micro”），为在边缘运行容器和虚拟机提供了一个不可变且高弹性的操作系统。SUSE Linux Micro 适用于 AArch64 和 AMD64/Intel 64 两种体系结构，还支持实时内核，可满足对延迟敏感的应用需求（如电信领域的使用场景）。
- 对于连接的群集（即与管理群集连接的群集），会部署两个代理。一个是 Rancher System Agent，用于管理与 Rancher Prime 的连接；另一个是 venv-salt-minion，用于接收 SUSE Multi-Linux Manager 的指令，以执行 Linux 软件更新。这些代理对于离线群集的管理而言并非必需组件。

3.2 连接



上图提供了**连接**的下游群集及其与管理群集的连接的总体体系结构概览。管理群集可以部署在各种底层基础架构平台上，包括本地和云平台，具体取决于下游群集和目标管理群集之间的网络连接情况。这项功能的唯一要求是 API 和回调 URL 可以通过连接下游群集节点和管理基础架构的网络进行访问。

必须注意，建立这种连接的机制不同于下游群集部署的机制。下一节将对此进行更深入的介绍，不过先建立一个基本概念，连接的下游群集成为“受管理”群集主要有三种机制：

1. 下游群集先以“断开连接”的方式部署（例如通过 Edge Image Builder（第 11 章 “Edge Image Builder”）），然后在连接允许的情况下导入管理群集。
2. 下游群集会被配置为使用内置的初始配置机制（例如通过 Elemental（第 13 章 “Elemental”）），并且它们在首次引导时会自动注册到管理群集，以允许群集配置的后期绑定。
3. 下游群集已置备裸机管理功能 (CAPI + Metal^3)，群集一旦部署和配置（通过 Rancher Turtles 操作器），它们就会自动导入管理群集。

注意

建议采用多个管理群集，以满足大规模部署需求，在地理位置分散的环境中优化带宽利用并改善延迟问题，同时在发生故障或管理群集升级时尽量减少中断。您可以在此处 (https://ranchermanager.docs.rancher.com/v2.11/getting-started/installation-and-upgrade/installation-requirements) 找到最新的管理群集可扩展性限制和系统要求。

4 常见的 Edge 部署模式

由于操作环境和生命周期要求各不相同，我们已针对一些不同的部署模式实施了支持，这些模式与 SUSE Edge 适用的细分市场和使用场景大体一致。我们为其中的每种部署模式编写了一份快速入门指南，以帮助您根据自己的需求熟悉 SUSE Edge 平台。下面介绍了我们目前支持的三种部署模式，并附有相关快速入门页面的链接。

4.1 定向网络置备

定向网络置备适用于您知道要部署到的硬件的细节，并可以直接访问带外管理界面来编排和自动化整个置备过程的情况。在这种情况下，客户预期解决方案能够从一个中心位置全自动地置备边缘站点，并可以最大限度地减少边缘位置的手动操作，也就是说，解决方案的作用远不止是创建引导映像；您只需将物理硬件装入机架、通电并为其连接所需的网络，自动化过程就会通过带外管理（例如通过 Redfish API）启动计算机并处理基础架构的置备、初始配置和部署，全程无需用户的干预。做到这一点的关键在于管理员了解系统；他们知道哪个硬件在哪个位置，并且部署预期可以得到集中处理。

此解决方案最为稳健，因为您可以直接与硬件的管理界面交互和处理已知硬件，并且很少会遇到网络可用性方面的限制。在功能上，此解决方案广泛使用 Cluster API 和 Metal^3 来自动完成从裸机到操作系统、Kubernetes 和分层应用程序的置备，并能够在部署后与 SUSE Edge 的其他常规生命周期管理功能相衔接。此解决方案的快速入门可在第 1 章 “使用 Metal^3 实现 BMC 自动化部署”中找到。

4.2 “自主回连”网络置备

有时，在您的操作环境中，中心管理群集无法直接管理硬件（例如，您的远程网络位于防火墙后面，或者没有带外管理界面；这种情况在边缘处经常使用的“PC”型硬件中很常见）。对于这种情况，我们提供了相应的工具来远程置备群集及其工作负载，而无需知道硬件在引导时位于何处。这就是大多数人对边缘计算的看法；有成千上万种不太常见的系统在边缘位置引导、安全地自主回连、验证它们的身份，并接收有关要执行哪种操作的指令。为此，我们希望除了出厂前预先构建计算机映像，或者简单挂接引导映像（例如通过 USB）来打开系统外，置备和生命周期管理工作几乎不需要用户干预即可完成。在此方面存在的主要挑战是如何解决这些设备在各种环境下的缩放性、一致性、安全性和生命周期管理问题。

此解决方案在系统置备和初始配置方式上提供了高度的灵活性和一致性，不受系统位置、类型或规格以及首次开机时间的影响。在 SUSE Edge 中，可以通过 Edge Image Builder 十分灵活地对系统进行自定义，同时可以利用 Rancher Elemental 提供的注册功能来初始配置节点和置备 Kubernetes，并利用 SUSE Multi-Linux 来修补操作系统。此解决方案的快速入门可在第 2 章 “使用 Elemental 进行远程主机接入”中找到。

4.3 基于映像的置备

对于需要在独立、隔离或网络受限的环境中操作的客户，SUSE Edge 提供了一种解决方案供客户生成完全自定义的安装媒体，其中包含所有必要的部署制品，用于在边缘启用单节点和多节点高可用性 Kubernetes 群集，包括任何所需的工作负载或其他分层组件。全程无需与外界建立任何网络连接，且无需集中式管理平台的干预。用户体验与“自主回连”解决方案非常相似，安装媒体同样会提供给目标系统，但该解决方案会“就地引导”。在这种情况下，可将创建的群集挂接到 Rancher 进行持续管理（即从“断开连接”转换为“已连接”操作模式，无需经过大范围的重新配置或重新部署），或者群集可以继续独立运行。请注意，在这两种情况下，都可以采用相同的一致性机制来自动执行生命周期操作。

此外，使用此解决方案可以快速创建管理群集，这些管理群集可以托管同时支持“定向网络置备”和“自主回连网络置备”模型的集中式基础架构，因为它可能是置备各种 Edge 基础架构最快速、最简单的方式。此解决方案充分利用 SUSE Edge Image Builder 的功能，生成完全自定义的无人照管安装媒体；快速入门可在第 3 章 “使用 Edge Image Builder 配置独立群集”中找到。

5 SUSE Edge 堆栈验证

所有 SUSE Edge 版本均由紧密集成且经过全面验证的组件组成，这些组件作为一个整体进行版本控制。SUSE Edge 团队在集成和堆栈验证方面不断努力，不仅测试了组件之间的集成，还确保系统在因外界因素影响而发生故障的情况下仍能按预期运行。所有测试运行和结果均已向公众发布，结果和所有输入参数都可以在 ci.edge.suse.com (https://ci.edge.suse.com) 上找到。

6 完整组件列表

以下是组件的完整列表，以及对每种组件的概要说明及其在 SUSE Edge 中用法的相关链接：

- Rancher（第 5 章 “Rancher”）
- Rancher 仪表板扩展（第 6 章 “Rancher 仪表板扩展”）
- Rancher Turtles（第 7 章 “Rancher Turtles”）
- SUSE Multi-Linux Manager
- Fleet（第 8 章 “Fleet”）
- SUSE Linux Micro（第 9 章 “SUSE Linux Micro”）
- Metal³（第 10 章 “Metal³”）
- Edge Image Builder（第 11 章 “Edge Image Builder”）
- NetworkManager Configurator（第 12 章 “边缘网络”）
- Elemental（第 13 章 “Elemental”）

- Akri（第 14 章 “Akri”）
- K3s（第 15 章 “K3s”）
- RKE2（第 16 章 “RKE2”）
- SUSE Storage（第 17 章 “SUSE Storage”）
- SUSE Security（第 18 章 “SUSE Security”）
- MetalLB（第 19 章 “MetalLB”）
- KubeVirt（第 21 章 “Edge Virtualization”）
- 系统升级控制器（第 22 章 “系统升级控制器”）
- 升级控制器（第 23 章 “升级控制器”）

I 快速入门

从这里快速入门

1 使用 Metal³ 实现 BMC 自动化部署

Metal³ 是一个 CNCF 项目 (https://metal3.io/)，它为 Kubernetes 提供裸机基础架构管理功能。

Metal³ 提供 Kubernetes 原生资源来管理裸机服务器的生命周期，支持通过 Redfish (https://www.dmtf.org/standards/redfish) 等带外协议进行管理。

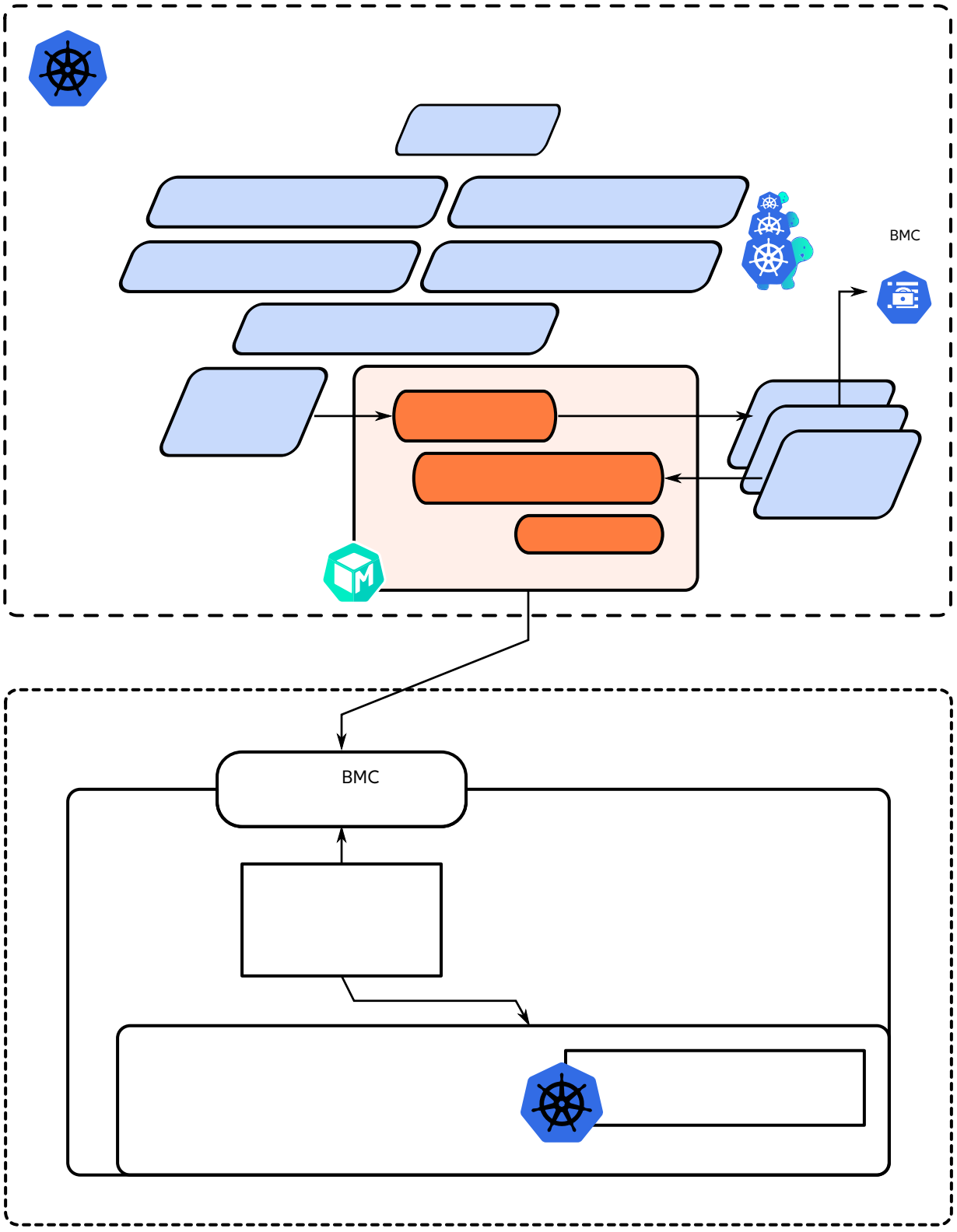
它还为 Cluster API (CAPI) (https://cluster-api.sigs.k8s.io/) 提供成熟的支持，允许通过广泛采用的不限供应商的 API 来管理跨多个基础架构提供商的基础架构资源。

1.1 为何使用此方法

此方法非常适合用于目标硬件支持带外管理，并且需要全自动化基础架构管理流程的场景。

管理群集配置为提供声明性 API 来对下游群集裸机服务器进行清单和状态管理，包括自动检查、清理和置备/取消置备。

1.2 总体体系结构



1.3 先决条件

下游群集服务器硬件和网络相关的限制具体如下所述：

- 管理群集
  - 必须与目标服务器管理/BMC API 建立网络连接
  - 必须与目标服务器控制平面网络建立网络连接
  - 对于多节点管理群集，需要配置额外的预留 IP 地址
- 要控制的主机
  - 必须支持通过 Redfish、iDRAC 或 iLO 接口进行带外管理
  - 必须支持通过虚拟媒体进行部署（目前不支持 PXE）
  - 必须与管理群集建立网络连接，以便能够访问 Metal^3 置备 API

需要安装一些工具，可以安装在管理群集上，也可以安装在能够访问管理群集的主机上。

- Kubectl (https://kubernetes.io/docs/reference/kubectl/kubectl/)、Helm (https://helm.sh) 和 Clusterctl (https://cluster-api.sigs.k8s.io/user/quick-start.html#install-clusterctl)
- Podman (https://podman.io) 或 Rancher Desktop (https://rancherdesktop.io) 等容器运行时

`SL-Micro.x86_64-6.1-Base-GM.raw` 操作系统映像文件必须从 SUSE Customer Center (https://scc.suse.com/) 或 SUSE 下载页面 (https://www.suse.com/download/sle-micro/) 下载。

1.3.1 设置管理群集

安装管理群集和使用 Metal^3 的基本步骤如下：

1. 安装 RKE2 管理群集
2. 安装 Rancher

3. 安装存储服务提供程序（可选）
4. 安装 Metal^3 依赖项
5. 通过 Rancher Turtles 安装 CAPI 依赖项
6. 为下游群集主机构建 SLEMicro 操作系统映像
7. 注册 BareMetalHost CR 以定义裸机清单
8. 通过定义 CAPI 资源创建下游群集

本指南假设已安装现有的 RKE2 群集和 Rancher（包括 cert-manager），例如已使用 Edge Image Builder 安装（第 11 章 “Edge Image Builder”）。

提示

也可以按照管理群集文档（第 40 章 “设置管理群集”）中所述，将此处所述的步骤完全自动化。

1.3.2 安装 Metal^3 依赖项

必须安装并运行 cert-manager（如果在安装 Rancher 的过程中未安装）。

必须安装持久性存储提供程序。建议使用 SUSE Storage，但对于开发/PoC 环境，也可以使用 `local-path-provisioner`。以下说明假设已将 StorageClass 标记为默认值 (https://kubernetes.io/docs/tasks/administer-cluster/change-default-storage-class/)，否则需要对 Metal^3 chart 进行额外的配置。

需要提供一个额外的 IP，该 IP 由 MetalLB (https://metallb.universe.tf/) 管理，用于为 Metal^3 管理服务提供一致的端点。此 IP 必须是控制平面子网的一部分，并且为静态配置预留（不是任何 DHCP 池的一部分）。

提示

如果管理群集仅包含单个节点，则可以避免通过 MetalLB 管理额外的浮动 IP，具体参见第 1.6.1 节 “单节点配置”

1. 首先，我们需要安装 MetalLB：

```
helm install \
  metallb oci://registry.suse.com/edge/charts/metallb \
  --namespace metallb-system \
  --create-namespace
```

2. 然后使用预留 IP 来定义 `IPAddressPool` 和 `L2Advertisement`（如下方的 `STATIC_IRONIC_IP` 所定义）：

```
export STATIC_IRONIC_IP=<STATIC_IRONIC_IP>

cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ironic-ip-pool
  namespace: metallb-system
spec:
  addresses:
  - ${STATIC_IRONIC_IP}/32
  serviceAllocation:
    priority: 100
    serviceSelectors:
    - matchExpressions:
      - {key: app.kubernetes.io/name, operator: In, values: [metal3-
ironic]}
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ironic-ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - ironic-ip-pool
EOF
```

3. 现在可以安装 Metal^3：

```
helm install \
  metal3 oci://registry.suse.com/edge/charts/metal3 \
  --namespace metal3-system \
  --create-namespace \
  --set global.ironicIP="$STATIC_IRONIC_IP"
```

4. 大约要经过两分钟，init 容器才会在此部署中运行，因此请确保所有 Pod 都已运行再继续操作：

```
kubectl get pods -n metal3-system
NAME                                                    READY   STATUS
 RESTARTS   AGE
baremetal-operator-controller-manager-85756794b-fz98d   2/2     Running   0
          15m
metal3-metal3-ironic-677bc5c8cc-55shd                   4/4     Running   0
          15m
metal3-metal3-mariadb-7c7d6fdbd8-64c7l                  1/1     Running   0
          15m
```

警告

请在 `metal3-system` 名称空间中的所有 Pod 都已运行后再继续执行以下步骤。

1.3.3 安装 Cluster API 依赖项

Cluster API 依赖项通过 Rancher Turtles Helm chart 管理：

```
cat > values.yaml <<EOF
rancherTurtles:
  features:
    embedded-capi:
      disabled: true
    rancher-webhook:
      cleanup: true
EOF
```

```
helm install \
  rancher-turtles oci://registry.suse.com/edge/charts/rancher-turtles \
  --namespace rancher-turtles-system \
  --create-namespace \
  -f values.yaml
```

一段时间后，控制器 Pod 应会在 `capi-system`、`capm3-system`、`rke2-bootstrap-system` 和 `rke2-control-plane-system` 名称空间中运行。

1.3.4 准备下游群集映像

Kiwi（第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”）和 Edge Image Builder（第 11 章 “Edge Image Builder”）用于准备将在下游群集主机上置备的经过修改的 SLEMicro 基础映像。

本指南介绍部署下游群集所需的最低限度配置。

1.3.4.1 映像配置

注意

首先，请先按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”中的说明构建一个创建群集所需的全新映像。

运行 Edge Image Builder 时，将从主机挂载一个目录，因此需要创建一个目录结构来存储用于定义目标映像的配置文件。

- `downstream-cluster-config.yaml` 是映像定义文件，有关详细信息，请参见第 3 章 “使用 Edge Image Builder 配置独立群集”。
- 下载的基础映像已经过 `xz` 压缩，必须使用 `unxz` 将其解压缩，并将其复制/移动到 `base-images` 文件夹中。

- network 文件夹是可选的，有关详细信息，请参见第 1.3.5.1.1 节 “用于静态网络配置的附加脚本”。
- custom/scripts 目录包含首次引导时运行的脚本；目前需要使用 01-fix-growfs.sh 脚本来调整部署中的操作系统根分区的大小

```
├── downstream-cluster-config.yaml
├── base-images/
│   └ SL-Micro.x86_64-6.1-Base-GM.raw
├── network/
│   └ configure-network.sh
└── custom/
    └ scripts/
        └ 01-fix-growfs.sh
```

1.3.4.1.1 下游群集映像定义文件

downstream-cluster-config.yaml 文件是下游群集映像的主配置文件。下面是通过 Metal[3] 进行部署的极简示例：

```
apiVersion: 1.2
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-GM.raw
  outputImageName: SLE-Micro-eib-output.raw
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      forceWait: true
      pools:
        - 2.suse.pool.ntp.org
      servers:
        - 10.0.0.1
        - 10.0.0.2
  kernelArgs:
```

```
    - ignition.platform.id=openstack
    - net.ifnames=1
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $USERKEY1
  packages:
    packageList:
      - jq
  sccRegistrationCode: $SCC_REGISTRATION_CODE
```

其中 `$SCC_REGISTRATION_CODE` 是从 SUSE Customer Center (https://scc.suse.com/) 中复制的注册代码，并且软件包列表包含必需的 `jq`。

`$ROOT_PASSWORD` 是 root 用户的已加密口令，可用于测试/调试目的。可以使用 `openssl passwd-6 PASSWORD` 命令生成此口令

对于生产环境，建议使用可添加到 users 块的 SSH 密钥（需将 `$USERKEY1` 替换为实际 SSH 密钥）。

注意

`net.ifnames=1` 会启用可预测网络接口命名 (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html)

这与 Metal^3 chart 的默认配置相匹配，但设置必须与配置的 chart `predictableNicNames` 值相匹配。

另请注意，`ignition.platform.id=openstack` 是必需的，如果不指定此参数，在 Metal^3 自动化流程中通过 ignition 进行 SUSE Linux Micro 配置将会失败。

虽然 `time` 部分是可选的，但强烈建议配置该部分，以免出现证书和时钟偏差方面的潜在问题。本示例中提供的值仅作说明之用，请根据您的具体要求相应调整。

1.3.4.1.2 Growfs 脚本

目前，在置备后首次引导时，需要使用一个自定义脚本 (`custom/scripts/01-fix-growfs.sh`) 来增大文件系统，使之与磁盘大小匹配。`01-fix-growfs.sh` 脚本包含以下信息：

```
#!/bin/bash
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --
noheadings)"
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

注意

使用相同的方法添加要在置备过程中执行的您自己的自定义脚本。有关详细信息，请参见第 3 章 “使用 Edge Image Builder 配置独立群集”。

1.3.4.2 映像创建

按照前面的章节准备好目录结构后，运行以下命令来构建映像：

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
 build --definition-file downstream-cluster-config.yaml
```

这会根据上述定义创建名为 `SLE-Micro-eib-output.raw` 的输出映像文件。

然后必须通过 Web 服务器提供输出映像，该服务器可以是通过 Metal3 chart 启用的媒体服务器容器（注意），也可以是其他某个本地可访问的服务器。在下面的示例中，此服务器是 `imagecache.local:8080`

注意

将 EIB 映像部署到下游群集时，还需要在 `Metal3MachineTemplate` 对象中包含该映像的 sha256 校验和。可通过以下方式生成该校验和：

```
sha256sum <image_file> > <image_file>.sha256
# On this example:
sha256sum SLE-Micro-eib-output.raw > SLE-Micro-eib-output.raw.sha256
```

1.3.5 添加 BareMetalHost 清单

注册自动部署的裸机服务器需要创建两个资源：一个用于存储 BMC 访问身份凭证的机密，以及一个用于定义 BMC 连接和其他细节的 Metal^3 BareMetalHost 资源：

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-credentials
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQ=
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: controlplane-0
  labels:
    cluster-role: control-plane
spec:
  online: true
```

```
  bootMACAddress: "00:f3:65:8a:a3:b0"
  bmc:
    address: redfish-virtualmedia://192.168.125.1:8000/redfish/v1/
Systems/68bd0fb6-d124-4d17-a904-cdf33efe83ab
    disableCertificateVerification: true
    credentialsName: controlplane-0-credentials
```

请注意以下几点：

- 机密用户名/口令必须采用 base64 编码。请注意，此值不能包含任何尾部换行符（例如，应使用 `echo -n`，而不是简单地使用 `echo`！）
- 可以现在设置 `cluster-role` 标签，也可以稍后在创建群集时设置。在以下示例中，应该设置 `control-plane` 或 `worker`
- `bootMACAddress` 必须是与主机的控制平面 NIC 匹配的有效 MAC
- `bmc` 地址用于连接 BMC 管理 API，支持以下类型：
  - `redfish-virtualmedia://<IP 地址>/redfish/v1/Systems/<系统 ID>`：Redfish 虚拟媒体，例如 SuperMicro
  - `idrac-virtualmedia://<IP 地址>/redfish/v1/Systems/System.Embedded.1`：Dell iDRAC
- 有关 BareMetalHost API 的详细信息，请参见上游 API 文档 (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md)

1.3.5.1 配置静态 IP

上面的 BareMetalHost 示例假设 DHCP 提供控制平面网络配置，但对于需要手动配置（例如静态 IP）的情况，可以提供附加配置，如下所述。

1.3.5.1.1 用于静态网络配置的附加脚本

使用 Edge Image Builder 创建基础映像时，请在 `network` 文件夹中创建以下 `configure-network.sh` 文件。

这会在首次引导时使用配置驱动器数据，并使用 NM Configurator 工具 (https://github.com/suse-edge/nm-configurator) 来配置主机网络。

```
#!/bin/bash

set -eux

# Attempt to statically configure a NIC in the case where we find a
 network_data.json
# In a configuration drive

CONFIG_DRIVE=$(blkid --label config-2 || true)
if [ -z "${CONFIG_DRIVE}" ]; then
  echo "No config-2 device found, skipping network configuration"
  exit 0
fi

mount -o ro $CONFIG_DRIVE /mnt

NETWORK_DATA_FILE="/mnt/openstack/latest/network_data.json"

if [ ! -f "${NETWORK_DATA_FILE}" ]; then
  umount /mnt
  echo "No network_data.json found, skipping network configuration"
  exit 0
fi

DESIRED_HOSTNAME=$(cat /mnt/openstack/latest/meta_data.json | tr ',{}' '\n' |
 grep '\"metal3-name\"' | sed 's/.*\"metal3-name\": \"\(.*\)\"/\1/')
echo "${DESIRED_HOSTNAME}" > /etc/hostname

mkdir -p /tmp/nmc/{desired,generated}
cp ${NETWORK_DATA_FILE} /tmp/nmc/desired/_all.yaml
umount /mnt

./nmc generate --config-dir /tmp/nmc/desired --output-dir /tmp/nmc/generated
./nmc apply --config-dir /tmp/nmc/generated
```

1.3.5.1.2 包含主机网络配置的附加机密

可为每个主机定义一个附加机密，其中包含采用 NM Configurator（第 12 章 “边缘网络”）所支持的 nmstate (https://nmstate.io/) 格式的数据。

然后通过 `preprovisioningNetworkDataName` 规范字段在 `BareMetalHost` 资源中引用该机密。

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: enp1s0
      type: ethernet
      state: up
      mac-address: "00:f3:65:8a:a3:b0"
      ipv4:
        address:
        - ip:  192.168.125.200
          prefix-length: 24
        enabled: true
        dhcp: false
    dns-resolver:
      config:
        server:
        - 192.168.125.1
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: 192.168.125.1
        next-hop-interface: enp1s0
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
```

```
  name: controlplane-0
  labels:
    cluster-role: control-plane
spec:
  preprovisioningNetworkDataName: controlplane-0-networkdata
# Remaining content as in previous example
```

注意

在某些情况下，可以省略 MAC 地址。请参见第 12.5.8 节 “统一节点配置”了解更多信息。

1.3.5.2 BareMetalHost 准备

按照上述步骤创建 BareMetalHost 资源和关联的机密后，将触发主机准备工作流程：

- 通过将虚拟媒体挂接到目标主机 BMC，引导内存盘映像
- 内存盘会检查硬件细节，并为主机做好置备准备（例如，清理磁盘中的旧数据）
- 完成此过程后，BareMetalHost `status.hardware` 字段中的硬件细节将会更新并可供验证

此过程可能需要几分钟时间，但完成后，您应该会看到 BareMetalHost 状态变为 `available`：

```
% kubectl get baremetalhost
NAME             STATE       CONSUMER   ONLINE   ERROR   AGE
controlplane-0   available              true             9m44s
worker-0         available              true             9m44s
```

1.3.6 创建下游群集

现在创建用于定义下游群集的 Cluster API 资源，以及创建计算机资源，后者会导致置备 BareMetalHost 资源，然后引导这些资源以形成 RKE2 群集。

1.3.7 控制平面部署

为了部署控制平面，我们需要定义一个如下所示的 yaml 清单，其中包含以下资源：

- 群集资源定义群集名称、网络和控制平面/基础架构提供程序的类型（在本例中为 RKE2/Metal3）
- Metal3Cluster 定义控制平面端点（单节点群集的主机 IP，多节点群集的负载平衡器端点，本示例假设使用单节点群集）
- RKE2ControlPlane 定义 RKE2 版本，以及群集引导期间所需的任何其他配置
- Metal3MachineTemplate 定义要应用于 BareMetalHost 资源的操作系统映像，hostSelector 定义要使用的 BareMetalHost
- Metal3DataTemplate 定义要传递给 BareMetalHost 的其他元数据（请注意，Edge 解决方案目前不支持 networkData）

注意

为简单起见，本示例假设使用单节点控制平面，其中 BareMetalHost 配置了 IP `192.168.125.200`。有关更高级的多节点示例，请参见第 42 章 “全自动定向网络置备”。

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: sample-cluster
  namespace: default
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
```

```
    kind: RKE2ControlPlane
    name: sample-cluster
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3Cluster
    name: sample-cluster
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3Cluster
metadata:
  name: sample-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: 192.168.125.200
    port: 6443
  noCloudProvider: true
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: sample-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: sample-cluster-controlplane
  replicas: 1
  version: v1.32.4+rke2r1
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  agentConfig:
    format: ignition
    kubelet:
      extraArgs:
```

```
        - provider-id=metal3://BAREMETALHOST_UUID
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: sample-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: sample-cluster-controlplane-template
      hostSelector:
```

```
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/SLE-Micro-eib-output.raw.sha256
        checksumType: sha256
        format: raw
        url: http://imagecache.local:8080/SLE-Micro-eib-output.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: sample-cluster-controlplane-template
  namespace: default
spec:
  clusterName: sample-cluster
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
      - key: local_hostname
        object: machine
```

根据您的环境对该示例清单进行调整后，便可通过 `kubectl` 应用该清单，然后通过 `clusterctl` 监控群集状态。

```
% kubectl apply -f rke2-control-plane.yaml

# Wait for the cluster to be provisioned
% clusterctl describe cluster sample-cluster
NAME                                                  READY  SEVERITY  REASON
 SINCE  MESSAGE
Cluster/sample-cluster                                True
 22m
├─ClusterInfrastructure - Metal3Cluster/sample-cluster  True
 27m
├─ControlPlane - RKE2ControlPlane/sample-cluster        True
 22m
```

```
| └Machine/sample-cluster-chflc                          True
 23m
```

1.3.8 工作/计算节点部署

与部署控制平面时一样，我们需要定义一个 YAML 清单，其中包含以下资源：

- MachineDeployment 定义复本（主机）数量和引导/基础架构提供程序（在本例中为 RKE2/Metal3）
- RKE2ConfigTemplate 描述代理主机引导的 RKE2 版本和首次引导配置
- Metal3MachineTemplate 定义要应用于 BareMetalHost 资源的操作系统映像，主机选择器定义要使用的 BareMetalHost
- Metal3DataTemplate 定义要传递给 BareMetalHost 的其他元数据（请注意，目前不支持 `networkData`）

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: MachineDeployment
metadata:
  labels:
    cluster.x-k8s.io/cluster-name: sample-cluster
  name: sample-cluster
  namespace: default
spec:
  clusterName: sample-cluster
  replicas: 1
  selector:
    matchLabels:
      cluster.x-k8s.io/cluster-name: sample-cluster
  template:
    metadata:
      labels:
        cluster.x-k8s.io/cluster-name: sample-cluster
    spec:
      bootstrap:
        configRef:
```

```
          apiVersion: bootstrap.cluster.x-k8s.io/v1alpha1
          kind: RKE2ConfigTemplate
          name: sample-cluster-workers
      clusterName: sample-cluster
      infrastructureRef:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
        kind: Metal3MachineTemplate
        name: sample-cluster-workers
      nodeDrainTimeout: 0s
      version: v1.32.4+rke2r1
---
apiVersion: bootstrap.cluster.x-k8s.io/v1alpha1
kind: RKE2ConfigTemplate
metadata:
  name: sample-cluster-workers
  namespace: default
spec:
  template:
    spec:
      agentConfig:
        format: ignition
        version: v1.32.4+rke2r1
        kubelet:
          extraArgs:
            - provider-id=metal3://BAREMETALHOST_UUID
        additionalUserData:
          config: |
            variant: fcos
            version: 1.4.0
            systemd:
              units:
                - name: rke2-preinstall.service
                  enabled: true
                  contents: |
                    [Unit]
                    Description=rke2-preinstall
                    Wants=network-online.target
                    Before=rke2-install.service
```

```
                    ConditionPathExists=!/run/cluster-api/bootstrap-
success.complete
                    [Service]
                    Type=oneshot
                    User=root
                    ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                    ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                    ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                    ExecStartPost=/bin/sh -c "umount /mnt"
                    [Install]
                    WantedBy=multi-user.target
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: sample-cluster-workers
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: sample-cluster-workers-template
      hostSelector:
        matchLabels:
          cluster-role: worker
      image:
        checksum: http://imagecache.local:8080/SLE-Micro-eib-output.raw.sha256
        checksumType: sha256
        format: raw
        url: http://imagecache.local:8080/SLE-Micro-eib-output.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: sample-cluster-workers-template
  namespace: default
```

```
spec:
  clusterName: sample-cluster
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
      - key: local_hostname
        object: machine
```

复制以上示例并根据您的环境进行修改后，可以通过 `kubectl` 应用该示例，然后使用 `clusterctl` 监控群集状态

```
% kubectl apply -f rke2-agent.yaml

# Wait for the worker nodes to be provisioned
% clusterctl describe cluster sample-cluster
NAME                                                READY  SEVERITY  REASON
 SINCE  MESSAGE
Cluster/sample-cluster                              True
 25m
├─ClusterInfrastructure - Metal3Cluster/sample-cluster  True
 30m
├─ControlPlane - RKE2ControlPlane/sample-cluster       True
 25m
│ └─Machine/sample-cluster-chflc                     True
 27m
└─Workers
  └─MachineDeployment/sample-cluster                 True
 22m
    └─Machine/sample-cluster-56df5b4499-zfljj        True
 23m
```

1.3.9 群集取消置备

可以通过删除上述创建步骤中应用的资源来取消置备下游群集：

```
% kubectl delete -f rke2-agent.yaml
% kubectl delete -f rke2-control-plane.yaml
```

这会触发 BareMetalHost 资源的取消置备，此过程可能需要几分钟，之后，这些资源将再次处于可用状态：

```
% kubectl get bmh
NAME             STATE            CONSUMER                              ONLINE
 ERROR   AGE
controlplane-0   deprovisioning   sample-cluster-controlplane-vlrt6   false
       10m
worker-0         deprovisioning   sample-cluster-workers-785x5        false
       10m

...

% kubectl get bmh
NAME             STATE       CONSUMER   ONLINE   ERROR   AGE
controlplane-0   available              false            15m
worker-0         available              false            15m
```

1.4 已知问题

- 目前不支持上游 IP 地址管理控制器 (https://github.com/metal3-io/ip-address-manager)，因为它与我们在 SLEMicro 中选择的网络配置工具和首次引导工具链尚不兼容。
- 此外，也不支持相关的 IPAM 资源和 Metal3DataTemplate networkData 字段。
- 目前仅支持通过 redfish-virtualmedia 进行部署。

1.5 计划的更改

- 通过 networkData 字段启用 IPAM 资源和配置支持

1.6 其他资源

SUSE Edge for Telco 文档（第 37 章 “SUSE Edge for Telco”）提供了 Metal^3 在电信领域使用场景中更高级的用法示例。

1.6.1 单节点配置

对于管理群集仅包含单个节点的测试/PoC 环境，可能不需要通过 MetalLB 管理额外的浮动 IP。

在此模式下，管理群集 API 的端点是管理群集的 IP，因此在使用 DHCP 时应预留该 IP，或者配置静态 IP，以确保管理群集 IP 不会变化 - 在下面显示为 `MANAGEMENT_CLUSTER_IP`。

为了实现此方案，必须如下所示指定 Metal^3 chart 值：

```
global:
  ironicIP: <MANAGEMENT_CLUSTER_IP>
metal3-ironic:
  service:
    type: NodePort
```

1.6.2 对虚拟媒体 ISO 挂接禁用 TLS

某些服务器供应商在将虚拟媒体 ISO 映像挂接到 BMC 时会验证 SSL 连接，这可能会导致出现问题，因为针对 Metal^3 部署生成的证书是自我签名证书。要解决此问题，可以使用如下所示的 Metal^3 chart 值仅对虚拟媒体磁盘挂接禁用 TLS：

```
global:
  enable_vmedia_tls: false
```

另一种解决方法是使用 CA 证书配置 BMC - 在这种情况下，可以使用 `kubectl` 从群集读取证书：

```
kubectl get secret -n metal3-system ironic-vmedia-cert -o yaml
```

然后可以在服务器 BMC 控制台上配置证书，不过，配置过程因供应商而异（并且不一定适用于所有供应商，如果不适用，可能需要指定 `enable_vmedia_tls` 标志）。

1.6.3 存储配置

在管理群集仅包含一个节点的测试/PoC 环境中，不需要持久性存储机制；但对于生产环境，建议在管理群集上安装 SUSE Storage (Longhorn)，以便与 Metal^3 相关的映像在 Pod 重启或重新调度时能够保持持久性。

为了实现此持久性存储机制，必须如下所示指定 Metal^3 chart 值：

```
metal3-ironic:
  persistence:
    ironic:
      size: "5Gi"
```

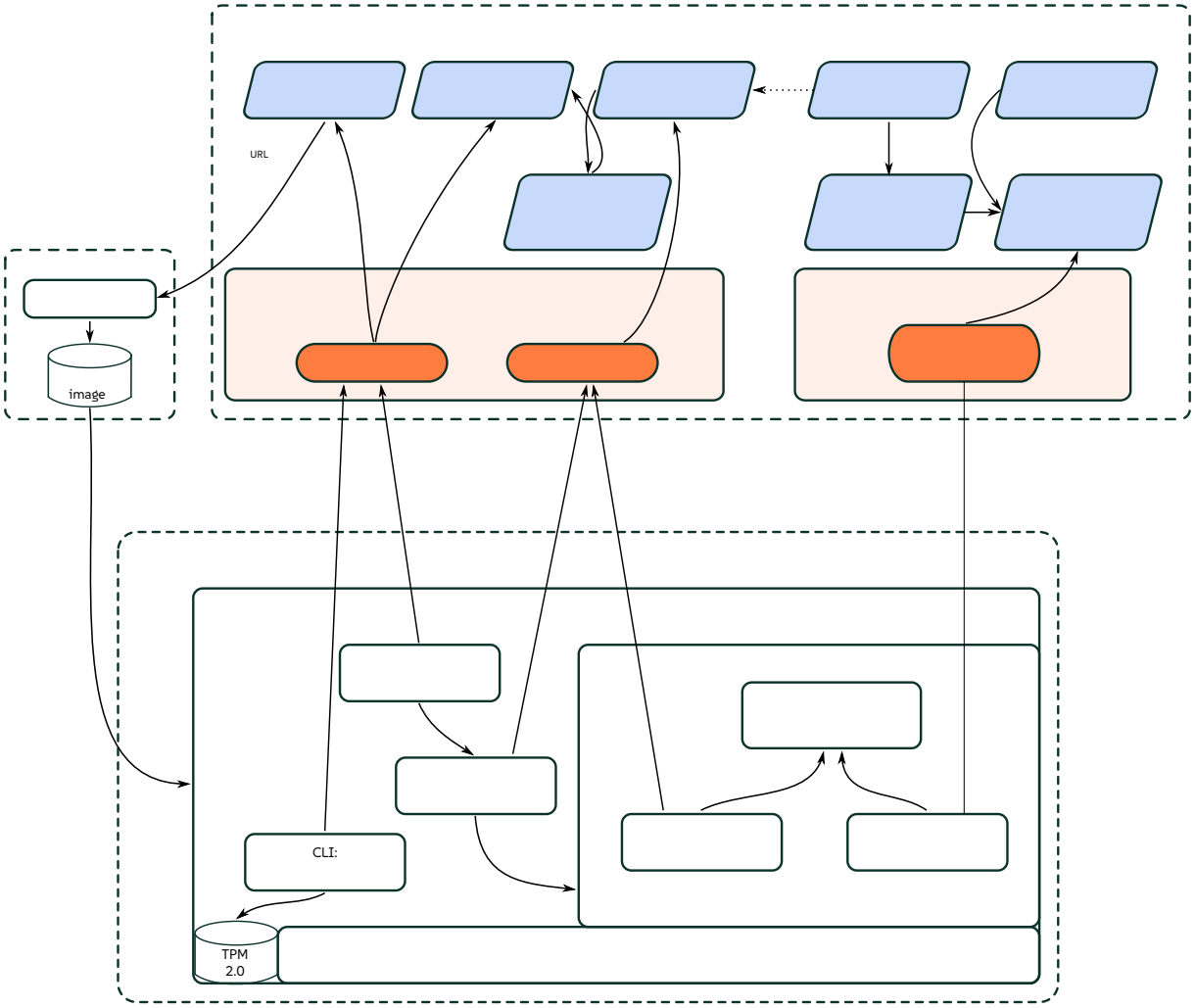
SUSE Edge for Telco 管理群集文档（第 40 章 “设置管理群集”）提供了更多有关如何为管理群集配置持久性存储机制的信息。

2 使用 Elemental 进行远程主机接入

本章介绍 SUSE Edge 中的“自主回连网络置备”解决方案。我们将使用 Elemental 来协助完成节点接入。Elemental 是一个软件堆栈，可用于注册远程主机和通过 Kubernetes 实现集中式云原生操作系统全面管理。在 SUSE Edge 堆栈中，我们将使用 Elemental 的注册功能将远程主机接入 Rancher，以便将主机集成到集中式管理平台，然后从该平台部署和管理 Kubernetes 群集以及分层组件、应用程序及其生命周期，所有这些操作都在一个中心位置完成。

此方法在以下情况下可能很有用：您要控制的设备与管理群集不在同一网络中，或没有带外管理控制器接入功能可以进行更直接的控制；您要在边缘处引导许多不同的“未知”系统，并且需要安全地大规模接入和管理这些系统。这种情况在以下领域的使用场景中很常见：零售、工业物联网，或几乎无法通过设备要安装到的网络进行控制的其他领域。

2.1 总体体系结构



2.2 所需资源

下面说明了学习本快速入门所要满足的最低系统和环境要求：

- 集中式管理群集的主机（托管 Rancher 和 Elemental 的主机）：

    - 至少 8 GB RAM 和 20 GB 磁盘空间，用于开发或测试（对于生产用途，请参见此处 (https://ranchermanager.docs.rancher.com/v2.11/getting-started/installation-and-upgrade/installation-requirements#hardware-requirements)）
- 要置备的目标节点，即边缘设备（可以使用虚拟机进行演示或测试）
    - 至少 4GB RAM、2 个 CPU 核心和 20 GB 磁盘空间
- 管理群集的可解析主机名，或用于 sslip.io 等服务的静态 IP 地址
- 用于通过 Edge Image Builder 构建安装媒体的主机
    - 运行 SLES 15 SP6、openSUSE Leap 15.6 或其他支持 Podman 的兼容操作系统。
    - 已安装 Kubectl (https://kubernetes.io/docs/reference/kubectl/kubectl/)、Podman (https://podman.io) 和 Helm (https://helm.sh)
- 用于引导的 USB 闪存盘（如果使用物理硬件）
- 下载的最新 SUSE Linux Micro 6.1 自安装 ISO 映像，该映像可在此处 (https://www.suse.com/download/sle-micro/) 获取。

注意

在初始配置过程中会重写目标计算机上的现有数据，因此请务必备份挂接到目标部署节点的所有 USB 存储设备和磁盘上的所有数据。

本指南是使用托管上游群集的 Digital Ocean droplet 以及用作下游设备的 Intel NUC 制作的。为了构建安装媒体，我们使用了 SUSE Linux Enterprise Server。

2.3 构建引导群集

首先创建一个能够托管 Rancher 和 Elemental 的群集。必须可以从下游节点所连接到的网络路由此群集。

2.3.1 创建 Kubernetes 群集

如果您使用的是超大规模云（例如 Azure、AWS 或 Google Cloud），那么，设置群集的最简单方法是使用这些云的内置工具。为简洁起见，本指南不会详细介绍使用每种选项的操作过程。

如果您要安装到裸机或其他宿主服务，同时需要提供 Kubernetes 发行版本身，我们建议您使用 RKE2 (https://docs.rke2.io/install/quickstart)。

2.3.2 设置 DNS

在继续之前，需要设置群集访问权限。与群集本身的设置一样，DNS 的配置方式因群集的托管位置而异。

提示

如果您不想处理 DNS 记录的设置（例如，这只是一个临时使用的测试服务器），可以改用 sslip.io (https://sslip.io) 之类的服务。通过此服务，可以使用 `<address>.sslip.io` 解析任何 IP 地址。

2.4 安装 Rancher

要安装 Rancher，需要访问刚刚创建的群集的 Kubernetes API。具体方式因使用的 Kubernetes 发行版而异。

对于 RKE2，kubeconfig 文件会写入 `/etc/rancher/rke2/rke2.yaml`。将此文件作为 `~/.kube/config` 保存在本地系统上。可能需要编辑该文件，以包含正确的外部可路由 IP 地址或主机名。

使用 Rancher 文档 (https://ranchermanager.docs.rancher.com/v2.11/getting-started/installation-and-upgrade/install-upgrade-on-a-kubernetes-cluster) 中所述的命令轻松安装 Rancher：

安装 cert-manager (https://cert-manager.io)：

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
```

```
helm install cert-manager jetstack/cert-manager \
 --namespace cert-manager \
 --create-namespace \
 --set crds.enabled=true
```

然后安装 Rancher 本身：

```
helm repo add rancher-prime https://charts.rancher.com/server-charts/prime
helm repo update
helm install rancher rancher-prime/rancher \
  --namespace cattle-system \
  --create-namespace \
  --set hostname=<DNS or sslip from above> \
  --set replicas=1 \
  --set bootstrapPassword=<PASSWORD_FOR_RANCHER_ADMIN> \
  --version 2.11.2
```

注意

如果目标系统是生产系统，请使用 cert-manager 配置实际证书（例如 Let's Encrypt 提供的证书）。

浏览到您设置的主机名，然后使用您的 `bootstrapPassword` 登录到 Rancher。系统将指导您完成一个简短的设置过程。

2.5 安装 Elemental

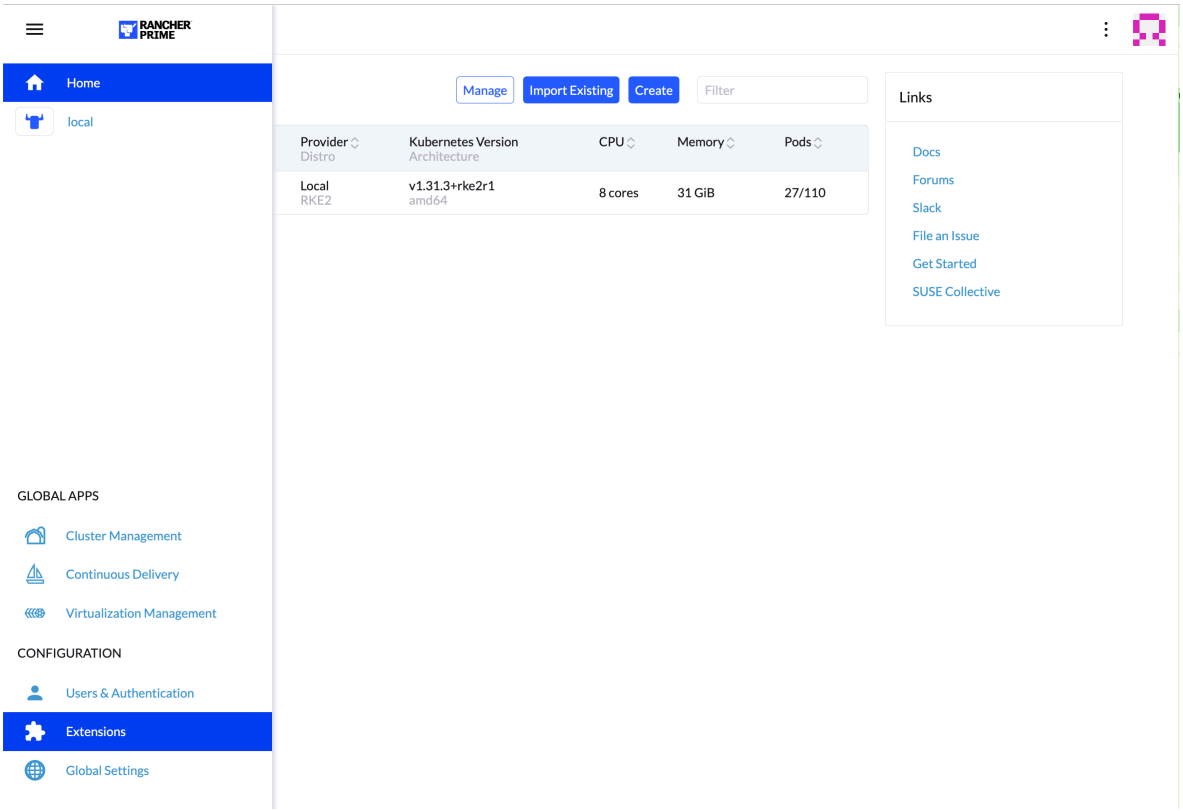
安装 Rancher 后，接下来可以安装 Elemental Operator 和所需的 CRD。Elemental 的 Helm chart 作为 OCI 制品发布，因此其安装过程比其他 chart 略简单一些。可以通过您用来安装 Rancher 的同一外壳安装 Helm chart，也可以在浏览器中通过 Rancher 的外壳安装。

```
helm install --create-namespace -n cattle-elemental-system \
 elemental-operator-crds \
 oci://registry.suse.com/rancher/elemental-operator-crds-chart \
 --version 1.6.8
```

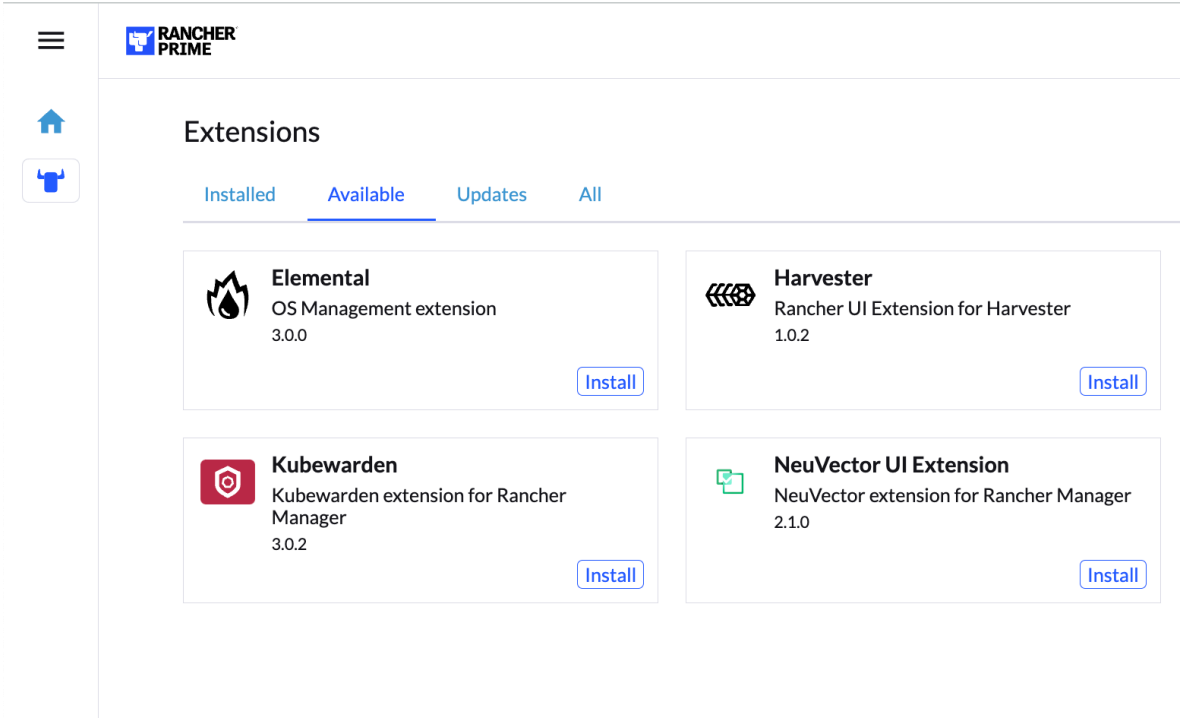
```
helm install -n cattle-elemental-system \
 elemental-operator \
 oci://registry.suse.com/rancher/elemental-operator-chart \
 --version 1.6.8
```

2.5.1 （可选）安装 Elemental UI 扩展

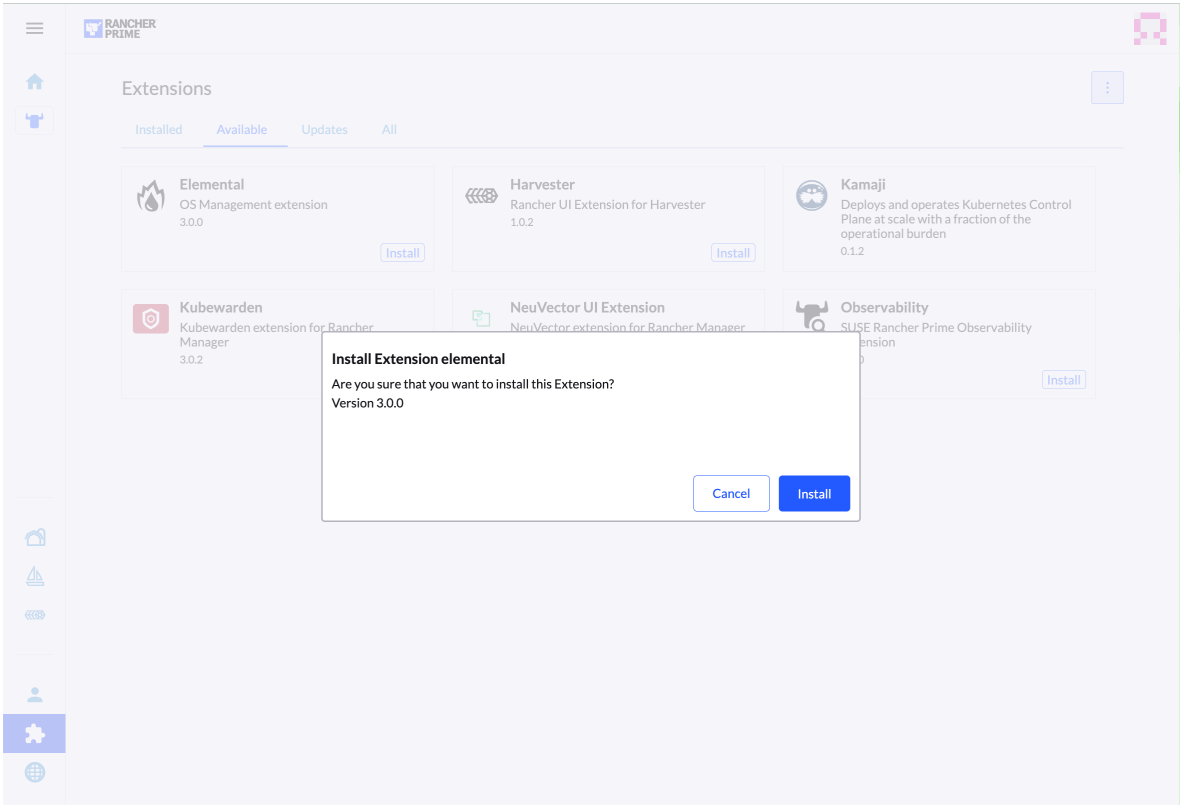
1. 要使用 Elemental UI，请登录到您的 Rancher 实例，单击左上角的三线菜单：



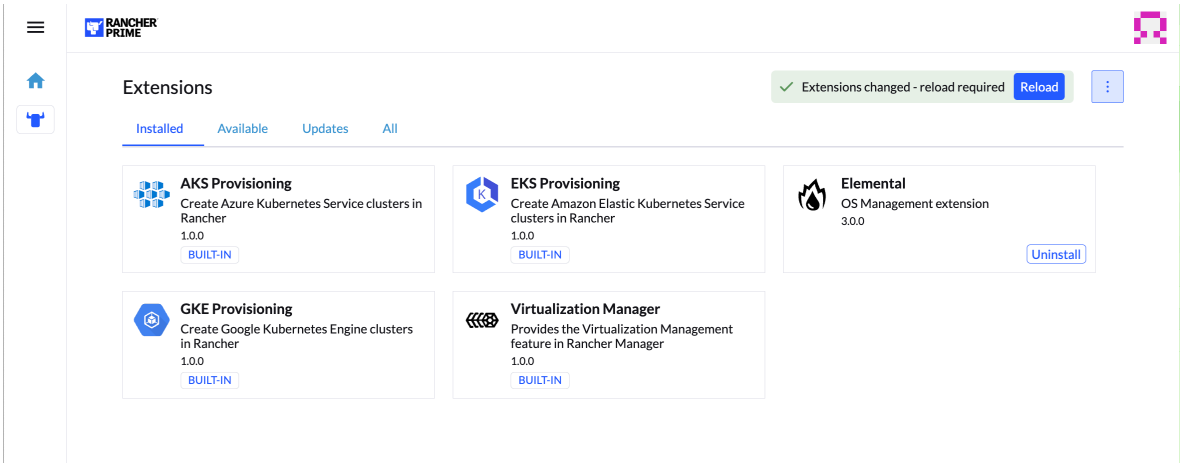
2. 在此页面上的“Available”（可用）选项卡中，单击“Elemental”卡片上的“Install”（安装）：


RANCHER PRIME
Extensions
Installed
Available
Updates
All
Elemental
OS Management extension
3.0.0
Install
Harvester
Rancher UI Extension for Harvester
1.0.2
Install
Kubewarden
Kubewarden extension for Rancher Manager
3.0.2
Install
NeuVector UI Extension
NeuVector extension for Rancher Manager
2.1.0
Install

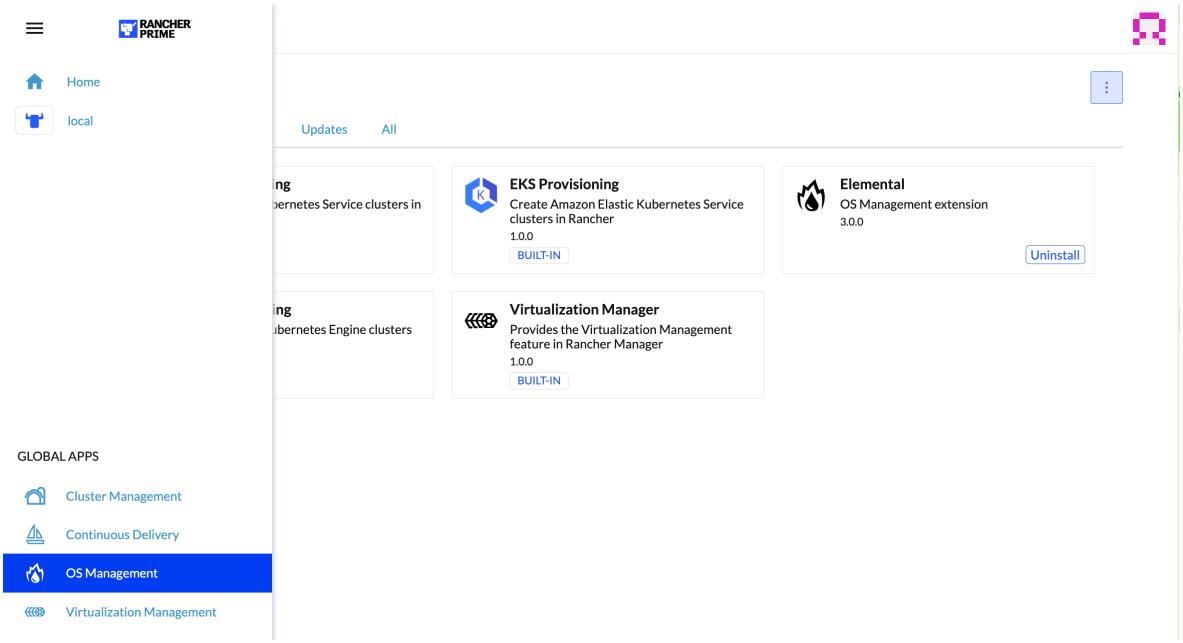
3. 确认您要安装该扩展：


RANCHER PRIME
Extensions
Installed
Available
Updates
All
Elemental
OS Management extension
3.0.0
Install
Harvester
Rancher UI Extension for Harvester
1.0.2
Install
Kamaji
Deploys and operates Kubernetes Control Plane at scale with a fraction of the operational burden
0.1.2
Kubewarden
Kubewarden extension for Rancher Manager
3.0.2
NeuVector UI Extension
NeuVector extension for Rancher Manager
Observability
SUSE Rancher Prime Observability
Install
Install Extension elemental
Are you sure that you want to install this Extension?
Version 3.0.0
Cancel
Install

4. 安装后，系统将提示您重新加载页面。



5. 重新加载后，可以通过“OS Management”（操作系统管理）全局应用程序访问 Elemental 扩展。



2.6 配置 Elemental

为方便起见，我们建议将变量 $ELEM 设置为配置目录所在的完整路径：

```
export ELEM=$HOME/elemental
```

```
mkdir -p $ELEM
```

为了能够将计算机注册到 Elemental，我们需要在 `fleet-default` 名称空间中创建 `MachineRegistration` 对象。

我们来创建该对象的基本版本：

```
cat << EOF > $ELEM/registration.yaml
apiVersion: elemental.cattle.io/v1beta1
kind: MachineRegistration
metadata:
  name: ele-quickstart-nodes
  namespace: fleet-default
spec:
  machineName: "\${System Information/Manufacturer}-\${System Information/UUID}"
  machineInventoryLabels:
    manufacturer: "\${System Information/Manufacturer}"
    productName: "\${System Information/Product Name}"
EOF

kubectl apply -f $ELEM/registration.yaml
```

注意

`cat` 命令使用反斜杠 (\) 将每个 `$` 符号转义，以免 Bash 将其模板化。如果手动复制命令，请去除反斜杠。

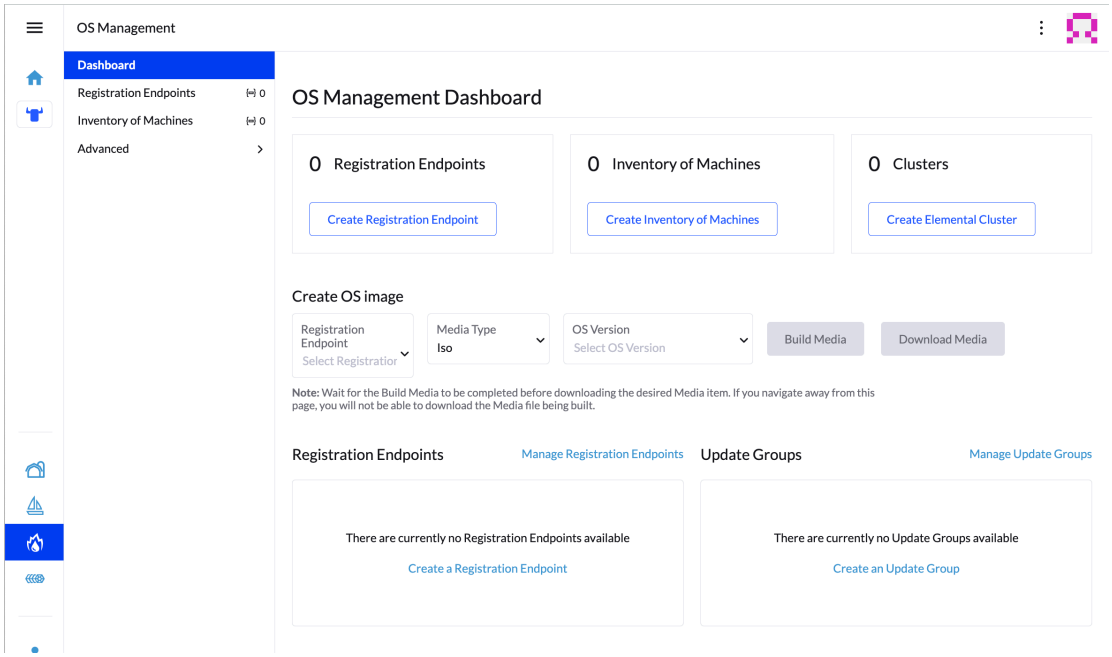
创建该对象后，找到并记下分配的端点：

```
REGISURL=$(kubectl get machineregistration ele-quickstart-nodes -n fleet-default
 -o jsonpath='{.status.registrationURL}')
```

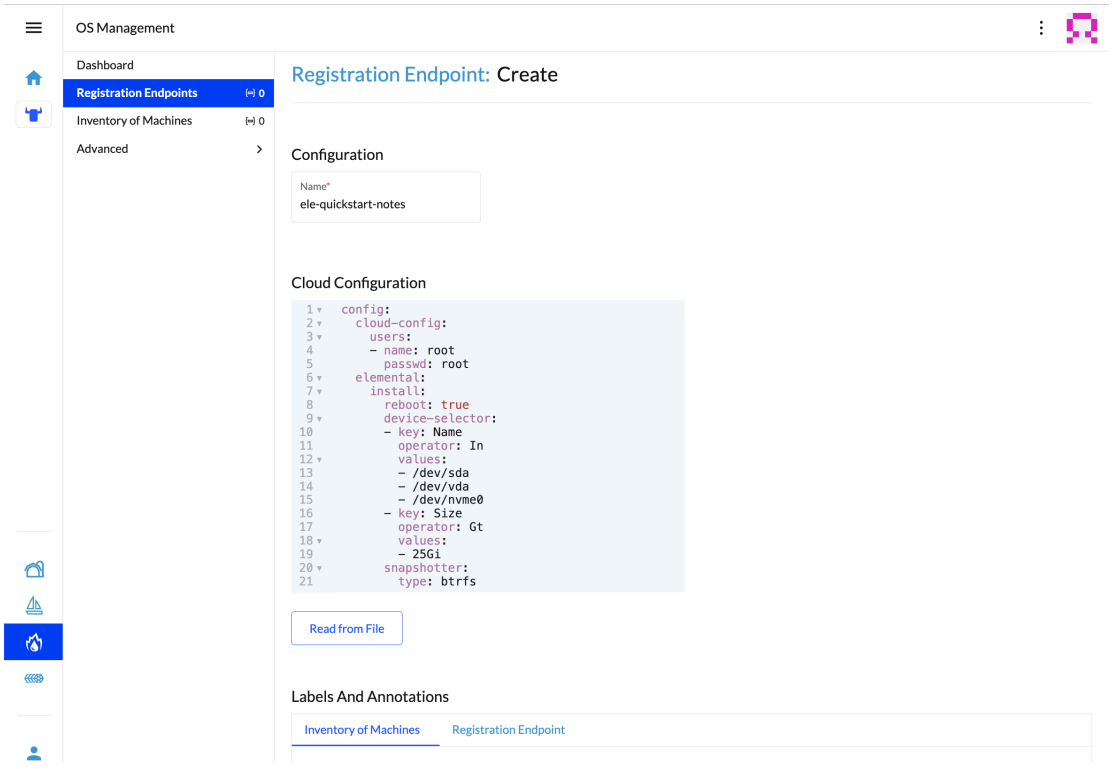
或者，可以在 UI 中执行此操作。

UI 扩展

1. 在“OS Management”（操作系统管理）扩展中，单击“Create Registration Endpoint”（创建注册端点）：


OS Management
Dashboard
Registration Endpoints 0
Inventory of Machines 0
Advanced
OS Management Dashboard
0 Registration Endpoints
Create Registration Endpoint
0 Inventory of Machines
Create Inventory of Machines
0 Clusters
Create Elemental Cluster
Create OS image
Registration Endpoint
Select Registratior
Media Type
Iso
OS Version
Select OS Version
Build Media
Download Media
Note: Wait for the Build Media to be completed before downloading the desired Media item. If you navigate away from this page, you will not be able to download the Media file being built.
Registration Endpoints
Manage Registration Endpoints
Update Groups
Manage Update Groups
There are currently no Registration Endpoints available
Create a Registration Endpoint
There are currently no Update Groups available
Create an Update Group

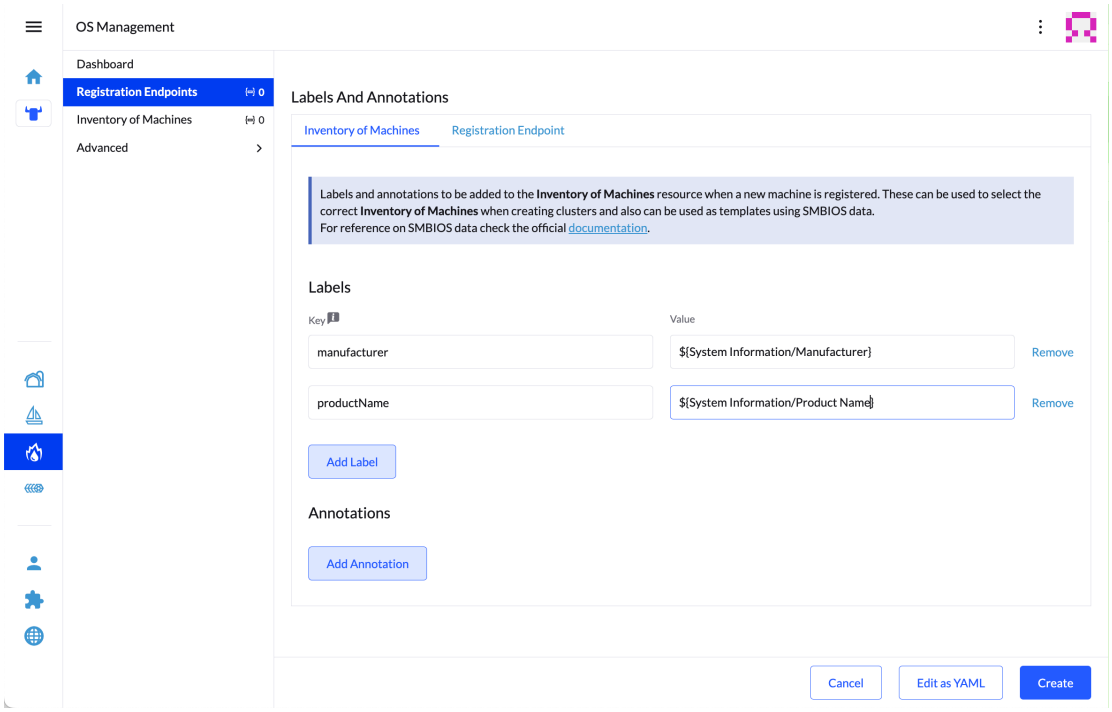
2. 为此配置命名。


OS Management
Dashboard
Registration Endpoints 0
Inventory of Machines 0
Advanced
Registration Endpoint: Create
Configuration
Name*
ele-quickstart-notes
Cloud Configuration
config:
cloud-config:
users:
- name: root
passwd: root
elemental:
install:
reboot: true
device-selector:
- key: Name
operator: In
values:
- /dev/sda
- /dev/vda
- /dev/nvme0
- key: Size
operator: Gt
values:
- 25Gi
snapshotter:
type: btrfs
Read from File
Labels And Annotations
Inventory of Machines
Registration Endpoint

注意

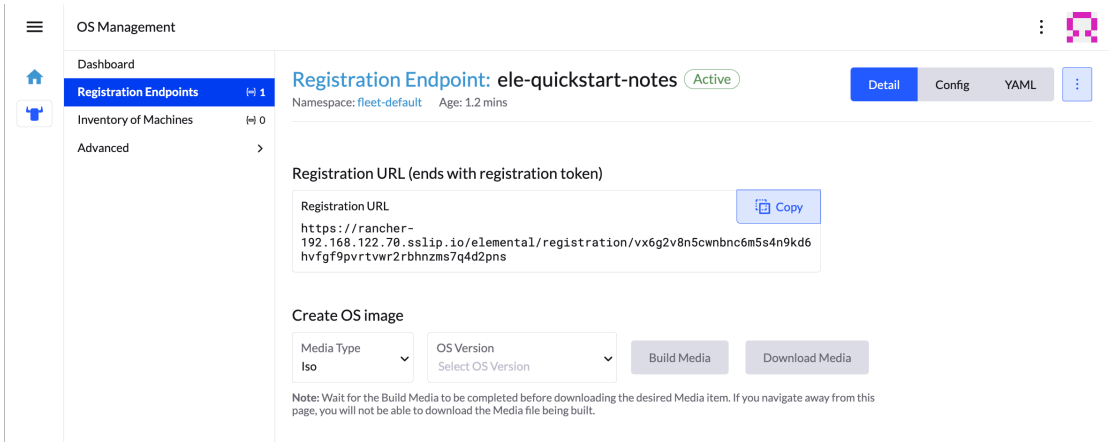
您可以忽略“Cloud Configuration”（云配置）字段，因为此处的数据将被 Edge Image Builder 中的后续步骤覆盖。

3. 接下来，向下滚动并单击“Add Label”（添加标签），为注册计算机时创建的每个资源添加标签。标签可用于区分计算机。



4. 单击“Create”（创建）以保存配置。

5. 创建注册后，您应该就会看到列出的注册 URL，此时可以单击“Copy”（复制）来复制该网址：



提示

如果您退出了该屏幕，可以单击左侧菜单中的“Registration Endpoints”（注册端点），然后单击刚刚创建的端点的名称。

此 URL 将在下一步骤中使用。

2.7 构建映像

虽然当前版本的 Elemental 提供了构建其自身安装媒体的方法，但在 SUSE Edge 3.3.1 中，我们改用 Kiwi 和 Edge Image Builder 来构建安装媒体，因此最终的系统将使用 SUSE Linux Micro (https://www.suse.com/products/micro/) 作为基础操作系统。

提示

有关使用 Kiwi 的详细信息，请先按照 Kiwi 映像构建器流程（第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”）构建全新映像；如要使用 Edge Image Builder，请参见 Edge Image Builder 入门指南（第 3 章 “使用 Edge Image Builder 配置独立群集”）和组件文档（第 11 章 “Edge Image Builder”）。

在安装了 Podman 的 Linux 系统中，创建相应目录并将 Kiwi 所构建的基础映像存放到其中：

```
mkdir -p $ELEM/eib_quickstart/base-images
```

```
cp /path/to/{micro-base-image-iso} $ELEM/eib_quickstart/base-images/
mkdir -p $ELEM/eib_quickstart/elemental
```

```
curl $REGISURL -o $ELEM/eib_quickstart/elemental/elemental_config.yaml
```

```
cat << EOF > $ELEM/eib_quickstart/eib-config.yaml
apiVersion: 1.2
image:
    imageType: iso
    arch: x86_64
    baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
    outputImageName: elemental-image.iso
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      forceWait: true
      pools:
        - 2.suse.pool.ntp.org
      servers:
        - 10.0.0.1
        - 10.0.0.2
  isoConfiguration:
    installDevice: /dev/vda
  users:
    - username: root
      encryptedPassword: \$6\$jHugJNNd3HElGsUZ\
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
  packages:
    sccRegistrationCode: XXX
EOF
```

注意

- 虽然 `time` 部分是可选的，但强烈建议配置该部分，以免出现证书和时钟偏差方面的潜在问题。本示例中提供的值仅作说明之用，请根据您的具体要求相应调整。
- 未编码的口令是 `eib`。
- 要从官方来源下载和安装必要的 RPM，则需要配置 `sccRegistrationCode`（或者，也可以手动侧载 `elemental-register` 和 `elemental-system-agent`）
- `cat` 命令使用反斜杠 (`\`) 将每个 `$` 符号转义，以免 Bash 将其模板化。如果手动复制命令，请去除反斜杠。
- 安装期间将会擦除安装设备。

```
podman run --privileged --rm -it -v $ELEM/eib_quickstart/:/eib \
 registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
 build --definition-file eib-config.yaml
```

如果要引导物理设备，我们需要将映像刻录到 USB 闪存盘中。为此请使用以下命令：

```
sudo dd if=/eib_quickstart/elemental-image.iso of=/dev/<PATH_TO_DISK_DEVICE>
 status=progress
```

2.8 引导下游节点

现在我们已创建安装媒体，接下来可以用它来引导下游节点。

对于您要使用 Elemental 控制的每个系统，请添加安装媒体并引导设备。安装后，系统会重引导并自行注册。

如果您正在使用 UI 扩展，则应会看到您的节点出现在“Inventory of Machines”（计算机清单）中。

注意

在出现登录提示之前，请勿移除安装媒体；在首次引导期间，仍需访问 USB 记忆棒上的文件。

2.9 创建下游群集

使用 Elemental 置备新群集时，需要创建两个对象。

Linux

第一个对象是 `MachineInventorySelectorTemplate`。此对象用于指定群集与清单中计算机之间的映射。

1. 创建一个选择器，用于根据标签匹配清单中的任何计算机：

```
cat << EOF > $ELEM/selector.yaml
apiVersion: elemental.cattle.io/v1beta1
kind: MachineInventorySelectorTemplate
metadata:
  name: location-123-selector
  namespace: fleet-default
spec:
  template:
    spec:
      selector:
        matchLabels:
          locationID: '123'
EOF
```

2. 将资源应用于群集：

```
kubectl apply -f $ELEM/selector.yaml
```

3. 获取计算机名称并添加匹配标签：

```
MACHINENAME=$(kubectl get MachineInventory -n fleet-default | awk
 'NR>1 {print $1}')
```

```
kubectl label MachineInventory -n fleet-default \
 $MACHINENAME locationID=123
```

4. 创建简单的单节点 K3s 群集资源，并将其应用于群集：

```
cat << EOF > $ELEM/cluster.yaml
apiVersion: provisioning.cattle.io/v1
kind: Cluster
metadata:
  name: location-123
  namespace: fleet-default
spec:
  kubernetesVersion: v1.32.4+k3s1
  rkeConfig:
    machinePools:
      - name: pool1
        quantity: 1
        etcdRole: true
        controlPlaneRole: true
        workerRole: true
        machineConfigRef:
          kind: MachineInventorySelectorTemplate
          name: location-123-selector
          apiVersion: elemental.cattle.io/v1beta1
EOF

kubectl apply -f $ELEM/cluster.yaml
```

UI 扩展

通过 UI 扩展可以采取几种简便做法。请注意，管理多个位置可能涉及大量的手动工作。

1. 如前所述，打开左侧的三线菜单并选择“OS Management”（操作系统管理）。您随即会转到管理 Elemental 系统的主屏幕。

2. 在左侧边栏上，单击“Inventory of Machines”（计算机清单）。这会打开已注册计算机的清单。

3. 要基于这些计算机创建群集，请选择所需的系统，单击“Actions”（操作）下拉列表，然后单击“Create Elemental Cluster”（创建 Elemental 群集）。这会打开“Cluster Creation”（创建群集）对话框，同时还会创建要在后台使用的 MachineSelectorTemplate。

4. 在此屏幕上，配置您要构建的群集。本快速入门选择了“K3s v1.30.5+k3s1”，其余选项保持原样。

提示

可能需要向下滚动才能看到更多选项。

创建这些对象后，您应会看到一个新的 Kubernetes 群集使用刚刚安装的新节点运行。

2.10 节点重置（可选）

SUSE Rancher Elemental 支持执行“节点重置”，从 Rancher 中删除整个群集、从群集中删除单个节点或者从计算机清单中手动删除某个节点时，可以选择性地触发该操作。当您想要重置和清理任何孤立资源，并希望自动将清理的节点放回计算机清单以便重复使用时，此功能非常有用。默认未启用此功能，因此不会清理任何已去除的系统（即，不会去除数据，任何 Kubernetes 群集资源将继续在下游群集上运行），并且需要手动干预才能擦除数据，并通过 Elemental 将计算机重新注册到 Rancher。

如果您希望默认启用此功能，则需要通过添加 `config.elemental.reset.enabled: true`，来确保 `MachineRegistration` 明确启用此功能，例如：

```
config:
  elemental:
    registration:
      auth: tpm
    reset:
      enabled: true
```

然后，所有使用此 `MachineRegistration` 注册的系统将自动在其配置中收到 `elemental.cattle.io/resettable: 'true'` 注解。如果您希望在各个节点上手动执行此操作（例如，您的现有 `MachineInventory` 没有此注解，或者您已部署节点），可以修改 `MachineInventory` 并添加 `resettable` 配置，例如：

```
apiVersion: elemental.cattle.io/v1beta1
kind: MachineInventory
metadata:
  annotations:
    elemental.cattle.io/os.unmanaged: 'true'
    elemental.cattle.io/resettable: 'true'
```

在 SUSE Edge 3.1 中，Elemental Operator 会在操作系统上设置一个标记，该标记将自动触发清理过程；它会停止所有 Kubernetes 服务、去除所有持久性数据、卸装所有 Kubernetes 服务、清理所有剩余 Kubernetes/Rancher 目录，并通过原始 Elemental `MachineRegistration` 配置强制重新注册到 Rancher。此过程会自动发生，无需任何手动干预。调用的脚本存放在 `/opt/edge/elemental_node_cleanup.sh` 中。一旦设置了标记，脚本便会通过 `systemd.path` 触发，因此会立即执行。

警告

使用 `resettable` 功能的假设条件是，预期在从 Rancher 去除节点/群集时执行的行为是擦除数据并强制重新注册。在这种情况下，必然会丢失数据，因此，请仅在您确定要执行自动重置时才使用此功能。

2.11 后续步骤

下面是使用本指南后建议阅读的一些资源：

- 第 8 章 “Fleet”中的端到端自动化
- 第 12 章 “边缘网络”中的其他网络配置选项

3 使用 Edge Image Builder 配置独立群集

Edge Image Builder (EIB) 工具可以简化为引导计算机生成随时可引导 (CRB) 的自定义磁盘映像的过程，即使在完全隔离场景中也能做到这一点。EIB 用于创建所有三个 SUSE Edge 部署空间中使用的部署映像，因为它足够灵活，可以提供最简单的自定义过程，例如通过提供全面配置的映像来添加用户或设置时区。例如，该映像可以设置复杂的网络配置、部署多节点 Kubernetes 群集、部署客户工作负载，并通过 Rancher/Elemental 和 SUSE Multi-Linux Manager 注册到集中式管理平台。EIB 以容器映像的形式运行，因此具有极高的跨平台可移植性，可确保所有必需的依赖项都是独立的，对系统中所安装用于操作该工具的软件包的影响极小。

注意

对于多节点场景，EIB 会自动部署 MetalLB 和 Endpoint Copier Operator，以便让使用构建的同一映像置备的主机自动加入 Kubernetes 群集。

有关详细信息，请阅读 Edge Image Builder 简介（第 11 章 “Edge Image Builder”）。

警告

Edge Image Builder 1.2.1 支持自定义 SUSE Linux Micro 6.1 映像，但不支持 SUSE Linux Enterprise Micro 5.5 或 6.0 等旧版本。

3.1 先决条件

- 运行 SLES 15 SP6 的 AMD64/Intel 64 构建主机（物理机或虚拟机）。
- Podman 容器引擎
- 通过 Kiwi 构建器过程（第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”）创建的 SUSE Linux Micro 6.1 自安装 ISO 映像

注意

对于非生产用途，可将 openSUSE Leap 15.6 或 openSUSE Tumbleweed 用作构建主机。其他操作系统也可能正常运行，前提是存在兼容的容器运行时。

3.1.1 获取 EIB 映像

EIB 容器映像已公开提供，可以通过在映像构建主机上运行以下命令从 SUSE Edge 注册表下载：

```
podman pull registry.suse.com/edge/3.3/edge-image-builder:1.2.1
```

3.2 创建映像配置目录

由于 EIB 在容器中运行，我们需要从主机挂载一个配置目录，以便能够指定所需的配置，并使 EIB 能够在构建过程中访问任何所需的输入文件和支持制品。此目录必须遵循特定的结构。我们在主目录中创建此目录，并将其命名为“eib”：

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR/base-images
```

在上一步骤中，我们创建了“base-images”目录，用于托管 SUSE Linux Micro 6.1 输入映像，现在我们确保将该映像复制到配置目录：

```
cp /path/to/SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso $CONFIG_DIR/
base-images/slemicro.iso
```

注意

在 EIB 运行过程中，**不会**修改原始基础映像；在 EIB 配置目录的根目录中，会使用所需的配置创建新的自定义版本。

此时，配置目录应如下所示：

```
└── base-images/
    └── slemicro.iso
```

3.3 创建映像定义文件

定义文件描述 Edge Image Builder 支持的大多数可配置选项，可以在此处 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/pkg/image/testdata/full-valid-example.yaml) 找到选项的完整示例。建议您查看上游构建映像指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md)，其中提供了比下文所述更详细的示例。首先，我们需要为操作系统映像创建一个非常简单的定义文件：

```
cat << EOF > $CONFIG_DIR/iso-definition.yaml
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
EOF
```

此定义指定我们要为基于 AMD64/Intel 64 的系统生成输出映像。用作基础需要进一步修改的映像是名为 `slemicro.iso` 的 `iso` 映像，应该位于 `$CONFIG_DIR/base-images/slemicro.iso`。此定义还概述了在 EIB 修改完映像后，输出映像将命名为 `eib-image.iso`，默认情况下，该输出映像驻留在 `$CONFIG_DIR` 中。

现在，我们的目录结构应如下所示：

```
├── iso-definition.yaml
└── base-images/
    └── slemicro.iso
```

下面的章节将介绍几个常见操作的示例：

3.3.1 配置操作系统用户

EIB 允许您为用户预先配置登录信息（例如口令或 SSH 密钥），包括设置固定的 root 口令。在此示例中，我们将修复 root 口令，而第一步是使用 OpenSSL 创建一次性的已加密口令：

```
openssl passwd -6 SecurePassword
```

此命令将输出如下所示的内容：

```
$6$G392FCbxVgnLaFw1$Ujt00mdpJ3tDHxEg1snBU3GjujQf6f8kvopu7jiCBIhRbRvMmKUqwcmXAKggaSSKeUUOEtCP
```

然后，我们可以在定义文件中添加一个名为 `operatingSystem` 的部分，其中包含 `users` 数组。最终的文件应如下所示：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword:
 $6$G392FCbxVgnLaFw1$Ujt00mdpJ3tDHxEg1snBU3GjujQf6f8kvopu7jiCBIhRbRvMmKUqwcmXAKggaSSKeUUOEtC
```

注意

还可以添加其他用户、创建主目录、设置用户 ID、添加 SSH 密钥身份验证，以及修改组信息。请参见上游构建映像指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md) 获取更多示例。

3.3.2 配置操作系统时间

虽然 `time` 部分是可选的，但强烈建议配置该部分，以免出现证书和时钟偏差方面的潜在问题。EIB 将根据此处的参数配置 chronyd 和 `/etc/localtime`。

```
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      forceWait: true
      pools:
        - 2.suse.pool.ntp.org
      servers:
        - 10.0.0.1
        - 10.0.0.2
```

- `timezone` 以“区域/地点”格式指定时区（例如“欧洲/伦敦”）。在 Linux 系统上运行 `timedatectl list-timezones` 命令即可查看完整列表。
- ntp：定义与（使用 chronyd）配置 NTP 相关的属性：
- forceWait：要求 chronyd 在启动其他服务前尝试同步时间源，超时时间为 180 秒。
- pools：指定 chronyd 将用作数据源的池列表（使用 `iburst` 缩短初始同步耗时）。
- servers：指定 chronyd 将用作数据源的服务器列表（使用 `iburst` 缩短初始同步耗时）。

注意

本示例中提供的值仅作说明之用，请根据您的具体要求相应调整。

3.3.3 添加证书

`certificates` 目录中存储的扩展名为“.pem”或“.crt”的证书文件将安装在节点系统范围的证书存储区内：

```
.
├── definition.yaml
└── certificates
    ├── my-ca.pem
    └── my-ca.crt
```

有关详细信息，请参见《Securing Communication with TLS Certificate》指南 (https://documentation.suse.com/smart/security/html/tls-certificates/index.html#tls-adding-new-certificates)。

3.3.4 配置 RPM 软件包

EIB 的主要功能之一是提供相关机制来向映像添加更多软件包，以便在安装完成时，系统能够立即利用已安装的软件包。EIB 允许用户指定以下信息：

- 映像定义中按名称列出的软件包
- 要从中搜索这些软件包的网络储存库
- SUSE Customer Center (SCC) 身份凭证，用于在官方 SUSE 储存库中搜索列出的软件包
- 通过 `$CONFIG_DIR/rpms` 目录侧载网络储存库中不存在的自定义 RPM
- 通过同一目录 (`$CONFIG_DIR/rpms/gpg-keys`) 指定 GPG 密钥，以便可以验证第三方软件包

然后，EIB 将在映像构建时运行软件包解析过程，将基础映像用作输入，并尝试提取和安装所有提供的软件包（通过列表指定或在本地提供）。EIB 将所有软件包（包括所有依赖项）下载到输出映像中存在的储存库，并指示系统在首次引导过程中安装这些软件包。在映像构建期间执行此过程可确保在首次引导期间成功将软件包安装在所需平台（例如边缘节点）上。这对于需要将其他软件包嵌入映像，而非在运行时通过网络提取的环境（例如隔离环境或网络受限环境）而言，同样具有优势。

我们通过一个简单的示例来演示这一点：我们将安装由第三方供应商提供支持的 NVIDIA 储存库中的 `nvidia-container-toolkit` RPM 软件包：

```
packages:
  packageList:
    - nvidia-container-toolkit
  additionalRepos:
    - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
```

最终的定义文件如下所示：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword:
 $6$G392FCbxVgnLaFw1$Ujt00mdpJ3tDHxEg1snBU3GjujQf6f8kvopu7jiCBIhRbRvMmKUqwcmXAKggaSSKeUUOEtC
  packages:
    packageList:
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
```

上面是一个简单的示例，但为了完整性，请在运行映像生成过程之前先下载 NVIDIA 软件包签名密钥：

```
$ mkdir -p $CONFIG_DIR/rpms/gpg-keys
$ curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey > $CONFIG_DIR/
rpms/gpg-keys/nvidia.gpg
```

警告

通过此方法添加其他 RPM 的目的是添加受支持的第三方组件或用户提供（和维护）的软件包；请不要使用此机制来添加通常不受 SUSE Linux Micro 支持的软件包。如果使用此机制从 openSUSE 储存库（不受支持）添加组件，包括来自更新版本或服务包的组件，可能会导致配置不受支持 - 尤其是当依赖项解析导致操作系统核心部分被替换时，即便最终系统看似能正常运行，也可能存在问题。如有不确定之处，请联系 SUSE 代表，以协助确定您所需配置的可支持性。

注意

在上游安装软件包指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/installing-packages.md) 中可以找到更详细的指南和更多示例。

3.3.5 配置 Kubernetes 群集和用户工作负载

EIB 的另一个特点是，可以使用它来自动部署可“就地引导”（即不需要任何形式的集中式管理基础架构来进行协调）的单节点和多节点高可用性 Kubernetes 群集。这种方式的主要应用场景是隔离式部署或网络受限环境，但即便在可完全无限制访问网络的情况下，它也可用于快速引导独立群集。

使用此方法不仅可以部署自定义操作系统，还可以指定 Kubernetes 配置、通过 Helm chart 指定任何其他分层组件，以及通过提供的 Kubernetes 清单指定任何用户工作负载。不过，使用此方法的设计原则是，默认假设用户希望实现隔离式部署，因此映像定义中指定的任何项目都将纳入映像中，其中包括用户提供的工作负载，在此过程中，EIB 会确保将定义文件所需的所有已识别镜像复制到本地，并由最终部署的系统中嵌入的映像注册表提供服务。

在下一个示例中，我们将采用现有的映像定义并指定 Kubernetes 配置（在本示例中，未列出系统及其角色，因此默认假设使用单节点群集），这会指示 EIB 置备单节点 RKE2 Kubernetes 群集。为了展示用户提供的工作负载（通过清单）和分层组件（通过 Helm）部署的自动化过程，我们将通过 SUSE Edge Helm chart 安装 KubeVirt，并通过 Kubernetes 清单安装 NGINX。需要追加到现有映像定义的附加配置如下：

```
kubernetes:
  version: v1.32.4+rke2r1
  manifests:
    urls:
      - https://k8s.io/examples/application/nginx-app.yaml
  helm:
    charts:
      - name: kubevirt
        version: 303.0.0+up0.5.0
        repositoryName: suse-edge
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
```

最终的完整定义文件现在应如下所示：

```
apiVersion: 1.2
image:
  imageType: iso
```

```
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword:
 $6$G392FCbxVgnLaFw1$Ujt00mdpJ3tDHxEg1snBU3GjujQf6f8kvopu7jiCBIhRbRvMmKUqwcmXAKggaSSKeUUOEtC
  packages:
    packageList:
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
kubernetes:
  version: v1.32.4+k3s1
  manifests:
    urls:
      - https://k8s.io/examples/application/nginx-app.yaml
  helm:
    charts:
      - name: kubevirt
        version: 303.0.0+up0.5.0
        repositoryName: suse-edge
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
```

注意

在上游文档 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md) 中可以找到多节点部署、自定义网络等选项和 Helm chart 选项/值的更多示例。

3.3.6 配置网络

在本快速入门的最后一个示例中，我们来配置在使用 EIB 生成的映像置备系统时启动的网络。请务必知道，除非提供网络配置，否则默认模式是在引导时对所有发现的接口使用 DHCP。但这并非在所有情况下都是理想配置，尤其是在 DHCP 不可用而需要提供静态配置时，或者需要设置更复杂的网络结构（例如绑定、LACP 和 VLAN）时，又或者需要覆盖某些参数（例如主机名、DNS 服务器和路由）时。

EIB 能够提供每个节点的配置（其中的相关系统由其 MAC 地址唯一标识），或者为每台计算机提供相同配置的覆盖值，这在系统 MAC 地址未知时更有用。EIB 使用一个称为 Network Manager Configurator（简称为 nmc）的附加工具，这是 SUSE Edge 团队构建的工具，用于基于 nmstate.io (https://nmstate.io/) 声明式网络纲要应用自定义网络配置，在引导时识别其正在引导的节点，并在任何服务启动之前应用所需的网络配置。

我们现在通过在所需 network 目录中特定于节点的文件中（基于所需主机名）描述期望的网络状态，为使用单一接口的系统应用静态网络配置：

```
mkdir $CONFIG_DIR/network

cat << EOF > $CONFIG_DIR/network/host1.local.yaml
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: 192.168.122.1
    next-hop-interface: eth0
    table-id: 254
  - destination: 192.168.122.0/24
    metric: 100
    next-hop-address:
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - 192.168.122.1
    - 8.8.8.8
interfaces:
```

```
- name: eth0
  type: ethernet
  state: up
  mac-address: 34:8A:B1:4B:16:E7
  ipv4:
    address:
    - ip: 192.168.122.50
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
EOF
```

警告

上述示例是针对默认 `192.168.122.0/24` 子网设置的，假设测试在虚拟机上执行，请根据您的环境进行调整，不要忘记设置 MAC 地址。由于可以使用同一映像来置备多个节点，EIB（通过 `nmc`）配置的网络取决于它是否能够根据节点 MAC 地址唯一标识节点，因此在引导期间 `nmc` 将向每台计算机应用正确的网络配置。这意味着，您需要知道所要安装到的系统的 MAC 地址。或者，默认行为是依赖 DHCP，但您可以利用 `configure-network.sh` 钩子将通用配置应用于所有节点 - 有关详细信息，请参见网络指南（第 12 章 “边缘网络”）。

最终的文件结构应如下所示：

```
├── iso-definition.yaml
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

我们刚刚创建的网络配置将被分析，必要的 NetworkManager 连接文件将自动生成并插入到 EIB 要创建的新安装映像中。这些文件将在主机置备期间应用，从而生成完整的网络配置。

注意

有关上述配置的更详细解释以及此功能的示例，请参见“边缘网络”组件（第 12 章 “边缘网络”）。

3.4 构建映像

获得基础映像和可供 EIB 使用的映像定义后，接下来我们需要构建映像。为此，只需使用 podman 结合“build”命令调用 EIB 容器，并指定定义文件：

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file iso-definition.yaml
```

该命令应输出如下所示的内容：

```
Setting up Podman API listener...
Downloading file: dl-manifest-1.yaml 100% (498/498 B, 9.5 MB/s)
Pulling selected Helm charts... 100% (1/1, 43 it/min)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Resolving package dependencies...
Rpm .......................... [SUCCESS]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% (3/3, 10 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
```

```
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% (657/657 MB, 48 MB/
s)
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% (368/368 MB, 48
 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% (35/35 MB, 50 MB/s)
Downloading file: sha256sum-amd64.txt 100% (4.3/4.3 kB, 6.2 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

构建的 ISO 映像存储在 `$CONFIG_DIR/eib-image.iso` 中：

```
├── iso-definition.yaml
├── eib-image.iso
├── _build
│   └── cache/
│       └── ...
│   └── build-<timestamp>/
│       └── ...
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

每次构建时都会在 `$CONFIG_DIR/_build/` 中创建一个带时间戳的文件夹，其中包含构建日志、构建期间使用的制品以及 `combustion` 和 `artefacts` 目录，这两个目录包含添加到 CRB 映像的所有脚本和制品。

此目录的内容如下所示：

```
├── build-<timestamp>/
│   ├── combustion/
│   │   ├── 05-configure-network.sh
│   │   ├── 10-rpm-install.sh
```

```
│   │   ├── 12-keymap-setup.sh
│   │   ├── 13b-add-users.sh
│   │   ├── 20-k8s-install.sh
│   │   ├── 26-embedded-registry.sh
│   │   ├── 48-message.sh
│   │   ├── network/
│   │   │   ├── host1.local/
│   │   │   │   └── eth0.nmconnection
│   │   │   └── host_config.yaml
│   │   ├── nmc
│   │   └── script
│   ├── artefacts/
│   │   ├── registry/
│   │   │   ├── hauler
│   │   │   ├── nginx:<version>-registry.tar.zst
│   │   │   ├── rancher_kubectl:<version>-registry.tar.zst
│   │   │   └── registry.suse.com_suse_sles_15.6_virt-operator:<version>-
registry.tar.zst
│   │   ├── rpms/
│   │   │   └── rpm-repo
│   │   │       ├── addrepo0
│   │   │       │   ├── nvidia-container-toolkit-<version>.rpm
│   │   │       │   ├── nvidia-container-toolkit-base-<version>.rpm
│   │   │       │   ├── libnvidia-container1-<version>.rpm
│   │   │       │   └── libnvidia-container-tools-<version>.rpm
│   │   │       ├── repodata
│   │   │       │   ├── ...
│   │   │       └── zypper-success
│   │   └── kubernetes/
│   │       ├── rke2_installer.sh
│   │       ├── registries.yaml
│   │       ├── server.yaml
│   │       ├── images/
│   │       │   ├── rke2-images-cilium.linux-amd64.tar.zst
│   │       │   └── rke2-images-core.linux-amd64.tar.zst
│   │       ├── install/
│   │       │   ├── rke2.linux-amd64.tar.gz
│   │       │   └── sha256sum-amd64.txt
```

```
│   │       └── manifests/
│   │           ├── dl-manifest-1.yaml
│   │           └── kubevirt.yaml
│   ├── createrepo.log
│   ├── eib-build.log
│   ├── embedded-registry.log
│   ├── helm
│   │   └── kubevirt
│   │       └── kubevirt-0.4.0.tgz
│   ├── helm-pull.log
│   ├── helm-template.log
│   ├── iso-build.log
│   ├── iso-build.sh
│   ├── iso-extract
│   │   └── ...
│   ├── iso-extract.log
│   ├── iso-extract.sh
│   ├── modify-raw-image.sh
│   ├── network-config.log
│   ├── podman-image-build.log
│   ├── podman-system-service.log
│   ├── prepare-resolver-base-tarball-image.log
│   ├── prepare-resolver-base-tarball-image.sh
│   ├── raw-build.log
│   ├── raw-extract
│   │   └── ...
│   └── resolver-image-build
│       └──...
└── cache
    └── ...
```

如果构建失败，会先在 `eib-build.log` 中记录相关信息。该日志会指出哪个组件构建失败，方便您进行调试。

此时，您应该有了一个随时可用的映像，它可以：

1. 部署 SUSE Linux Micro 6.1
2. 配置 root 口令

3. 安装 `nvidia-container-toolkit` 软件包
4. 配置嵌入的容器注册表以在本地处理内容
5. 安装单节点 RKE2
6. 配置静态网络
7. 安装 KubeVirt
8. 部署用户提供的清单

3.5 调试映像构建过程

如果映像构建过程失败，请参见上游调试指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/debugging.md)。

3.6 测试新构建的映像

有关如何测试新构建的 CRB 映像的说明，请参见上游映像测试指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/testing-guide.md)。

4 SUSE Multi-Linux Manager

SUSE Edge 中包含 SUSE Multi-Linux Manager，其作用是提供自动化与控制能力，确保边缘部署的所有节点上作为底层操作系统的 SUSE Linux Micro 始终保持最新状态。

本快速入门指南旨在帮助您尽快熟悉 SUSE Multi-Linux Manager，目标是为您的边缘节点提供操作系统更新。本快速入门指南不会涉及诸如调整存储容量、创建和管理用于过渡的额外软件通道，或是针对大规模部署场景管理用户、系统组及组织等主题。对于生产用途，我们强烈建议您熟悉内容全面的 SUSE Multi-Linux Manager 文档 (https://documentation.suse.com/suma/5.0/en/suse-manager/index.html)。

要使 SUSE Edge 能够有效使用 SUSE Multi-Linux Manager，需执行以下步骤：

- 部署并配置 SUSE Multi-Linux Manager 服务器。
- 同步 SUSE Linux Micro 软件包储存库。
- 创建系统组。
- 创建激活密钥。
- 使用 Edge Image Builder 准备用于 SUSE Multi-Linux Manager 注册的安装媒体。

4.1 部署 SUSE Multi-Linux Manager 服务器

如果您的实例已在运行最新版 SUSE Multi-Linux Manager 5.0，则可以跳过此步骤。

您可以在专用物理服务器、自有硬件上的虚拟机或云中运行 SUSE Multi-Linux Manager 服务器。针对受支持的公有云，我们提供了预配置的 SUSE Multi-Linux 服务器虚拟机映像。

在本快速入门中，我们使用的是适用于 AMD64/Intel 64 的“qcow2”映像 `SUSE-Manager-Server.x86_64-5.0.2-Qcow-2024.12.qcow2`，您可以在 https://www.suse.com/download/suse-manager/ 或 SUSE Customer Center 中找到该映像。此映像可在 KVM 等超级管理程序上作为虚拟机运行。请务必检查我们是否提供了该映像的最新版本，并在执行全新安装时使用最新版本。

您也可以在其他任何受支持的硬件体系结构上安装 SUSE Multi-Linux Manager 服务器。在这种情况下，请选择与您的硬件体系结构相匹配的映像。

下载映像后，请创建一台至少满足以下最低硬件规格的虚拟机：

- 16 GB RAM
- 4 个物理或虚拟核心
- 一个至少 100 GB 的额外块设备

使用 qcow2 映像时，无需安装操作系统。您可以直接将该映像挂接为根分区。

您需要设置网络，以便您的边缘节点日后能够通过包含完全限定域名（简称“FQDN”）的主机名访问 SUSE Multi-Linux Manager 服务器！

首次引导 SUSE Multi-Linux Manager 时，需要执行一些初始配置：

- 选择键盘布局
- 接受许可协议
- 选择您所在的时区
- 输入操作系统的 root 口令

后续步骤需要以“root”用户身份执行：

下一步需要用到两个注册代码，可在 SUSE Customer Center 中找到相应注册代码：

- SLE Micro 5.5 的注册代码
- SUSE Multi-Linux Manager 扩展的注册代码

注册 SUSE Linux Micro：

```
transactional-update register -r <REGCODE> -e <your_email>
```

注册 SUSE Multi-Linux Manager：

```
transactional-update register -p SUSE-Manager-Server/5.0/x86_64 -r <REGCODE>
```

产品字符串取决于您的硬件体系结构！例如，如果您在 64 位 Arm 系统上使用 SUSE Multi-Linux Manager，则该字符串为“SUSE-Manager-Server/5.0/aarch64”。

重引导

更新系统：

```
transactional-update
```

除非未发生任何更改，否则请重引导系统以应用更新。

SUSE Multi-Linux Manager 通过由 Podman 管理的容器提供。`mgradm` 命令会为您处理设置和配置工作。

警告

请务必为您的 SUSE Multi-Linux Manager 服务器配置包含完全限定域名（简称“FQDN”）的主机名，并确保您要管理的边缘节点能够在网络中正确解析该主机名！

在安装和配置 SUSE Multi-Linux Manager 服务器容器之前，您需要准备好之前添加的额外块设备。为此，您需要知道虚拟机为该设备分配的名称。例如，如果块设备为 `/dev/vdb`，则可使用以下命令将其配置为供 SUSE Multi-Linux Manager 使用：

```
mgr-storage-server /dev/vdb
```

部署 SUSE Multi-Linux Manager：

```
mgradm install podman <FQDN>
```

提供 CA 证书的口令。该口令应与您的登录口令不同。通常情况下，您之后无需输入此口令，但应将其记录下来。

提供“admin”用户的口令。这是登录 SUSE Multi-Linux Manager 的初始用户。您之后可以创建具有完整权限或受限权限的其他用户。

4.2 配置 SUSE Multi-Linux Manager

部署完成后，您可以使用之前提供的主机名登录 SUSE Multi-Linux Manager Web 用户界面。初始用户为“admin”，请使用上一步设置的口令登录。

下一步需要您的组织身份凭证，这些凭证可在 SUSE Customer Center 中您所在组织的“用户”选项卡的第二个子选项卡中找到。通过这些身份凭证，SUSE Multi-Linux Manager 能够同步您已订阅的所有产品。

选择管理 › 安装向导。

在组织身份凭证选项卡中，使用您在 SUSE Customer Center 中找到的用户名和口令创建新身份凭证。

前往下一个选项卡 SUSE 产品。您需要等待与 SUSE Customer Center 的首次数据同步完成。

列表加载完毕后，使用过滤器仅显示“Micro 6”。选中与边缘节点运行的硬件体系结构（x86_64 或 aarch64）对应的 SUSE Linux Micro 6.1 复选框。

单击添加产品。此操作将会添加 SUSE Linux Micro 的主软件包储存库（即“通道”），并会自动添加 SUSE Manager 客户端工具的通道作为子通道。

首次同步可能需要一段时间才能完成，具体时长取决于您的互联网连接情况。您可以在此期间开始执行后续步骤：

在“系统”>“系统组”下，至少创建一个组，以便系统在初始配置时自动加入该组。组是对系统进行分类的重要方式，通过组可以一次性对整组系统应用配置或操作。其概念类似于 Kubernetes 中的标签。

单击 + 创建组

提供一个简短名称（例如“边缘节点”）和详细说明。

在“系统”>“激活密钥”下，至少创建一个激活密钥。激活密钥可视为一种配置文件，为系统进行 SUSE Multi-Linux Manager 初始配置时会自动应用该配置。如果您希望特定边缘节点加入不同的组或使用不同的配置，可以为它们创建单独的激活密钥，之后在 Edge Image Builder 中使用这些密钥来创建自定义安装媒体。

激活密钥的一个典型高级使用场景是：将测试群集分配到包含最新更新的软件通道，而生产群集则分配到仅会获取已在测试群集中验证过的那些最新更新的软件通道。

单击 + 创建密钥

输入简短说明（例如“边缘节点”），并提供一个唯一名称来标识该密钥（例如，对于硬件体系结构为 AMD64/Intel 64 的边缘节点，可命名为“edge-x86_64”）。密钥前面会自动添加一个数字，默认组织的前缀始终为“1”；如果在 SUSE Multi-Linux Manager 中创建其他组织并为其创建密钥，前缀数字可能会不同。

如果尚未创建任何克隆的软件通道，可将“基础通道”的设置保留为“SUSE Manager Default”，这会自动为边缘节点分配正确的 SUSE 更新储存库。

在“子通道”中，针对激活密钥适用的硬件体系结构，选择“包括建议项”滑块，这会添加“SUSE-Manager-Tools-For-SL-Micro-6.1”通道。

在“组”选项卡中，添加之前创建的组。所有使用该激活密钥进行初始配置的节点都将自动添加到该组中。

4.3 使用 Edge Image Builder 创建自定义安装映像

要使用 Edge Image Builder，您只需要拥有一个可通过 Podman 启动基于 Linux 的容器的环境。

在极简实验设置中，我们实际上可以使用运行 SUSE Multi-Linux Manager 服务器的同一台虚拟机。请确保该虚拟机有充足的磁盘空间！但不建议在生产环境中采用这种设置。有关我们已测试过与 Edge Image Builder 兼容的主机操作系统，请参见第 3.1 节 “先决条件”。

以 root 身份登录 SUSE Multi-Linux Manager 服务器主机。

提取 Edge Image Builder 容器：

```
podman pull registry.suse.com/edge/3.3/edge-image-builder:1.2.1
```

创建目录 `/opt/eib` 及其子目录 `base-images`：

```
mkdir -p /opt/eib/base-images
```

在本快速入门中，我们使用的是 SUSE Linux Micro 映像的“自安装”版本。该映像之后可写入物理 USB 闪存盘，用于在物理服务器上安装系统。如果服务器支持通过 BMC（基板管理控制器）远程挂接安装 ISO，也可采用这种方式。此外，该映像还可与大多数虚拟化工具结合使用。

如果需要将映像直接预加载到物理节点，或直接从虚拟机 (VM) 启动，也可使用“原始”版本的映像。

您可以在 SUSE Customer Center 或 https://www.suse.com/download/sle-micro/ 上找到这些映像。

将映像 `SL-Micro.x86_64-6.1-Default-SelfInstall-GM.install.iso` 下载或复制到 `base-images` 目录，并将其命名为“slemicro.iso”。

在基于 Arm 的构建主机上构建 AArch64 映像属于 SUSE Edge 3.3.1 的技术预览功能。该功能很可能可以正常运行，但目前尚未提供支持。如果您想尝试该功能，则需要在 64 位 Arm 计算机上运行 Podman，并且需要将所有示例和代码段中的“x86_64”替换为“aarch64”。

在 `/opt/eib` 中，创建一个名为 `iso-definition.yaml` 的文件。这是用于 Edge Image Builder 的构建定义文件。

下面的简单示例将会安装 SL Micro 6.1、设置 root 口令和键盘映射、启动 Cockpit 图形界面，并将节点注册到 SUSE Multi-Linux Manager：

```
apiVersion: 1.0
```

```
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
  - username: root
    createHomeDir: true
    encryptedPassword: $6$aaBTHyqDRUMY1HAp$pmBY7.qLtoVlCGj32XR/
Ogei4cngc3f4OX7fwBD/gw7HWyuNBOKYbBWnJ4pvrYwH2WUtJLKMbinVtBhMDHQIY0
  keymap: de
  systemd:
    enable:
      - cockpit.socket
  packages:
    noGPGCheck: true
  suma:
    host: ${fully qualified hostname of your SUSE Multi-Linux Manager Server}
    activationKey: 1-edge-x86_64
```

Edge Image Builder 还可以配置网络、在节点上自动安装 Kubernetes，甚至通过 Helm chart 部署应用程序。如需更详尽的示例，请参见第 3 章 “使用 Edge Image Builder 配置独立群集”。

对于 `baseImage`，请指定要使用的 `base-images` 目录中 ISO 的实际名称。

在此示例中，root 口令为“root”。请参见第 3.3.1 节 “配置操作系统用户”了解如何为要使用的安全口令创建口令哈希。

将键盘映射设置为安装系统后所需的实际键盘布局。

注意

我们使用 `noGPGCheck: true` 选项，因为我们不打算提供用于检查 RPM 软件包的 GPG 密钥。请参见上游软件包安装指南 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/installing-packages.md)，获取我们建议用于生产环境的更安全设置的综合指南。

如前文所述， SUSE Multi-Linux Manager 主机需要一个完全限定的主机名，且在边缘节点引导至的网络中能够解析该主机名。

`activationKey` 的值应与您在 SUSE Multi-Linux Manager 中创建的密钥一致。

要构建在安装后能自动将边缘节点注册到 SUSE Multi-Linux Manager 的安装映像，您还需要准备两个制品：

- 用于安装 SUSE Multi-Linux Manager 管理代理的 Salt 受控端软件包
- SUSE Multi-Linux Manager 服务器的 CA 证书

4.3.1 下载 venv-salt-minion 软件包

在 `/opt/eib` 中创建子目录 `rpms`。

从 SUSE Multi-Linux Manager 服务器将 `venv-salt-minion` 软件包下载到该目录。可以通过 Web UI 获取（在“软件”>“通道列表”中找到该软件包，并从 SUSE-Manager-Tools … 通道下载），或者使用 curl 等工具从 SUSE Multi-Linux Manager 的“引导储存库”下载：

```
curl -O http://${HOSTNAME_OF_SUSE_MANAGER}/pub/repositories/slmicro/6/1/
bootstrap/x86_64/venv-salt-minion-3006.0-3.1.x86_64.rpm
```

如果已有更新的版本发布，实际软件包名称可能会有所不同。如果有多个软件包可供选择，请始终选择最新版本。

4.4 下载 SUSE Multi-Linux Manager CA 证书

在 `/opt/eib` 中创建子目录 `certificates`

从 SUSE Multi-Linux Manager 将 CA 证书下载到该目录：

```
curl -O http://${HOSTNAME_OF_SUSE_MANAGER}/pub/RHN-ORG-TRUSTED-SSL-CERT
```

警告

必须将证书重命名为 `RHN-ORG-TRUSTED-SSL-CERT.crt`。这样，Edge Image Builder 将在安装过程中确保在边缘节点上安装并激活该证书。

现在，您可以运行 Edge Image Builder：

```
cd /opt/eib
podman run --rm -it --privileged -v /opt/eib:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file iso-definition.yaml
```

如果您为 YAML 定义文件使用了不同的名称，或者想使用其他版本的 Edge Image Builder，则需要相应地调整命令。

构建完成后，您可在 `/opt/eib` 目录中找到名为 `eib-image.iso` 的安装 ISO。

II 组件

Edge 的组件列表

5 Rancher

请参见 https://ranchermanager.docs.rancher.com/v2.11 上的 Rancher 文档。

Rancher 是一个功能强大的开源 Kubernetes 管理平台，可以简化跨多个环境的 Kubernetes 群集的部署、操作和监控。无论您是在本地、云中还是边缘管理群集，Rancher 都能提供统一且集中的平台来满足您的所有 Kubernetes 需求。

5.1 Rancher 的主要功能

- **多群集管理：**Rancher 的直观界面让您可以从任何位置（公有云、专用数据中心和边缘位置）管理 Kubernetes 群集。
- **安全性与合规性：**Rancher 在您的 Kubernetes 环境中实施安全策略、基于角色的访问控制 (RBAC) 与合规性标准。
- **简化的群集操作：**Rancher 可自动执行群集置备、升级和查错，并为各种规模的团队简化 Kubernetes 操作。
- **集中式应用程序目录：**Rancher 应用程序目录提供多种 Helm chart 和 Kubernetes 操作器，使容器化应用程序的部署和管理变得简单。
- **持续交付：**Rancher 支持 GitOps 和 CI/CD 管道，可以自动化和简化应用程序交付过程。

5.2 Rancher 在 SUSE Edge 中的使用

Rancher 为 SUSE Edge 堆栈提供多项核心功能：

5.2.1 集中式 Kubernetes 管理

在采用大量分布式群集的典型边缘部署中，Rancher 充当中心控制平面来管理这些 Kubernetes 群集。它提供统一的界面用于置备、升级、监控和查错、简化操作并确保一致性。

5.2.2 简化的群集部署

Rancher 简化了轻量级 SUSE Linux Micro 操作系统上的 Kubernetes 群集创建，并通过稳健的 Kubernetes 功能简化了边缘基础架构的部署。

5.2.3 应用程序部署和管理

集成的 Rancher 应用程序目录可以简化跨 SUSE Edge 群集的容器化应用程序的部署和管理，实现无缝的边缘工作负载部署。

5.2.4 安全性和策略实施

Rancher 提供基于策略的治理工具、基于角色的访问控制 (RBAC)，以及与外部身份验证提供程序的集成。这有助于 SUSE Edge 部署保持安全且合规，在分布式环境中，这一点至关重要。

5.3 最佳实践

5.3.1 GitOps

Rancher 包含内置组件 Fleet。使用 Fleet 可以通过 git 中存储的代码管理群集配置和应用程序部署。

5.3.2 可观测性

Rancher 包含 Prometheus 和 Grafana 等内置监控和日志记录工具，可提供群集健康状况和性能的综合深入信息。

5.4 使用 Edge Image Builder 进行安装

SUSE Edge 使用第 11 章 “Edge Image Builder” 来自定义基础 SUSE Linux Micro 操作系统映像。请按照第 27.6 节 “Rancher 安装” 中所述，在 EIB 置备的 Kubernetes 群集上进行 Rancher 隔离式安装。

5.5 其他资源

- Rancher 文档 (https://rancher.com/docs/)
- Rancher 学院 (https://www.rancher.academy/)
- Rancher 社区 (https://rancher.com/community/)
- Helm Chart (https://helm.sh/)
- Kubernetes 操作器 (https://operatorhub.io/)

6 Rancher 仪表板扩展

用户、开发人员、合作伙伴及客户可以使用扩展来扩展和增强 Rancher UI。SUSE Edge 提供 KubeVirt 和 Akri 仪表板扩展。

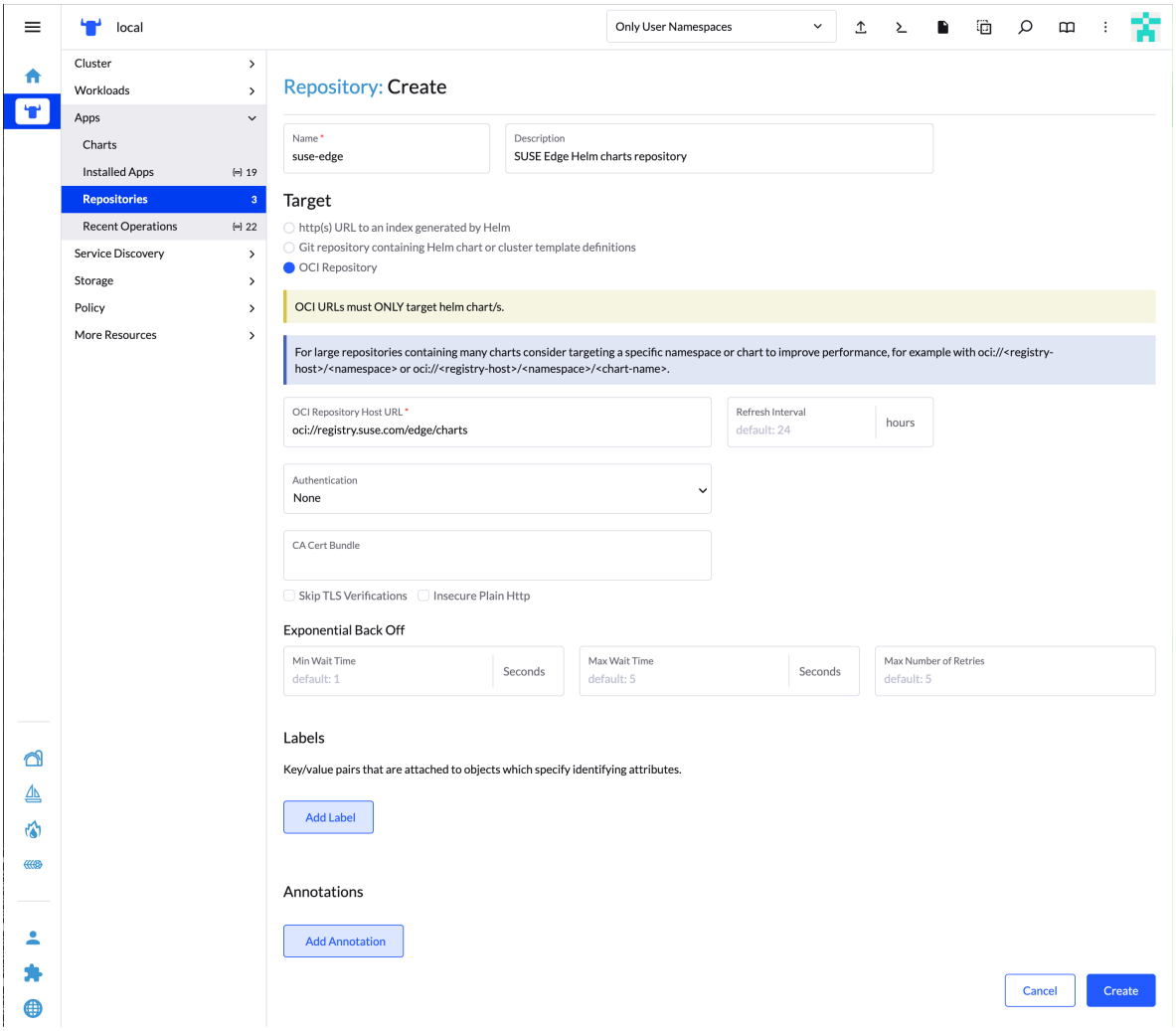
有关 Rancher 仪表板扩展的一般信息，请参见 Rancher 文档。

6.1 安装

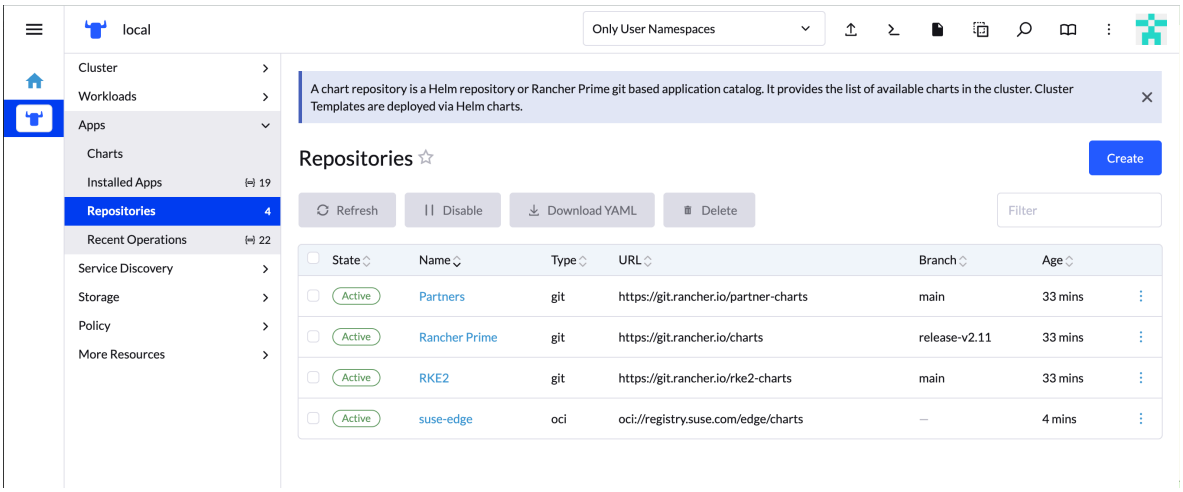
所有 SUSE Edge 3.3.1 组件（包括仪表板扩展）均作为 OCI 制品分发。要安装 SUSE Edge 扩展，可以使用 Rancher Dashboard UI、Helm 或 Fleet：

6.1.1 通过 Rancher 仪表板 UI 安装

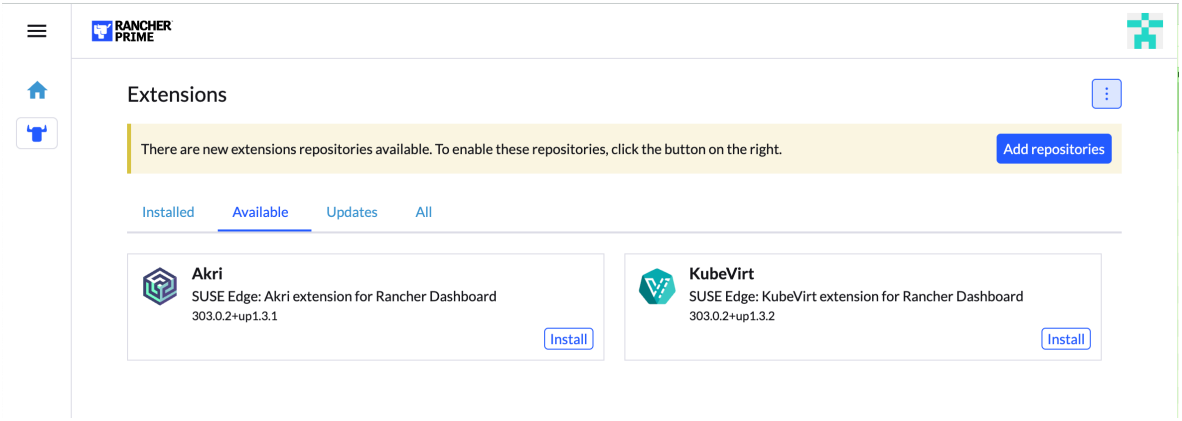
1. 单击导航侧边栏 **Configuration**（配置）部分的 **Extensions**（扩展）。
2. 在“Extensions”（扩展）页面上，单击右上角的三点菜单，然后选择 **Manage Repositories**（管理储存库）。
   每个扩展都通过各自的 OCI 制品分发，可通过 SUSE Edge Helm chart 储存库获取。
3. 在 **Repositories**（储存库）页面上，单击 Create（创建）。
4. 在表单中指定储存库名称和 URL，然后单击 Create（创建）。
   SUSE Edge Helm chart 储存库 URL：`oci://registry.suse.com/edge/charts`



5. 您会看到扩展储存库已添加到列表中，并处于 `Active`（活动）状态。



6. 导航回导航侧边栏 **Configuration**（配置）部分的 **Extensions**（扩展）。
   在 **Available**（可用）选项卡中，可以看到可供安装的扩展。



7. 在扩展卡片上单击 `Install`（安装）并确认安装。
   安装扩展后，Rancher UI 会提示重新加载页面，如安装扩展 Rancher 文档页面所述。

6.1.2 使用 Helm 进行安装

```
# KubeVirt extension
helm install kubevirt-dashboard-extension oci://registry.suse.com/edge/charts/
kubevirt-dashboard-extension --version 303.0.2+up1.3.2 --namespace cattle-ui-
plugin-system

# Akri extension
helm install akri-dashboard-extension oci://registry.suse.com/edge/charts/akri-
dashboard-extension --version 303.0.2+up1.3.1 --namespace cattle-ui-plugin-
system
```

注意

扩展需安装在 `cattle-ui-plugin-system` 名称空间中。

注意

安装扩展后，需要重新加载 Rancher 仪表板 UI。

6.1.3 使用 Fleet 进行安装

使用 Fleet 安装仪表板扩展需要定义一个 `gitRepo` 资源，该资源指向包含自定义 `fleet.yaml` 捆绑包配置文件的 Git 储存库。

```
# KubeVirt extension fleet.yaml
defaultNamespace: cattle-ui-plugin-system
helm:
  releaseName: kubevirt-dashboard-extension
  chart: oci://registry.suse.com/edge/charts/kubevirt-dashboard-extension
  version: "303.0.2+up1.3.2"
```

```
# Akri extension fleet.yaml
defaultNamespace: cattle-ui-plugin-system
helm:
  releaseName: akri-dashboard-extension
  chart: oci://registry.suse.com/edge/charts/akri-dashboard-extension
  version: "303.0.2+up1.3.1"
```

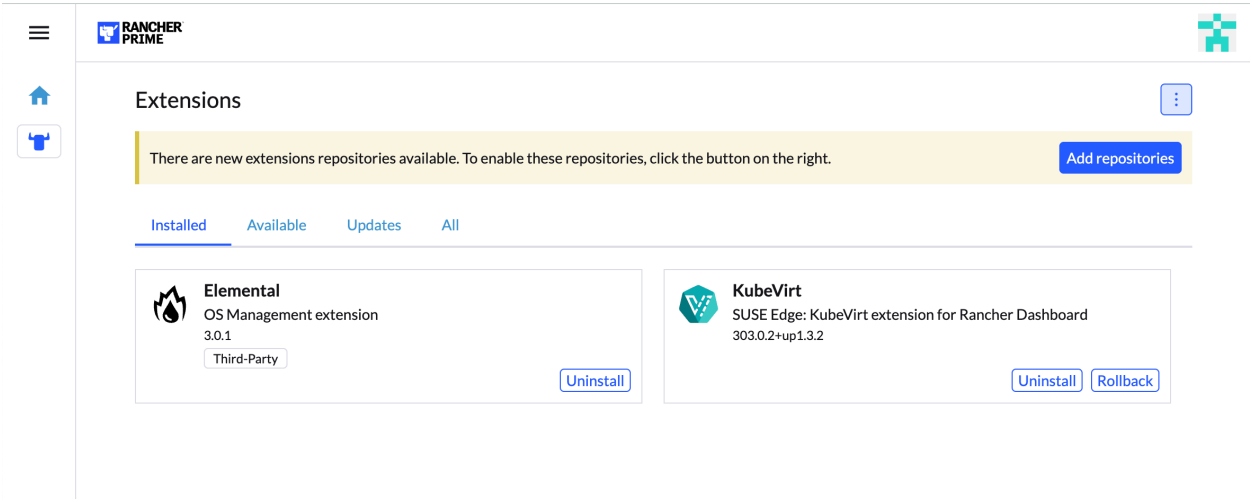
注意

必须指定 `releaseName` 属性，而且它需要与扩展名称匹配，才能正确安装扩展。

```
cat <<- EOF | kubectl apply -f -
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: edge-dashboard-extensions
  namespace: fleet-local
spec:
  repo: https://github.com/suse-edge/fleet-examples.git
  branch: main
  paths:
  - fleets/kubevirt-dashboard-extension/
  - fleets/akri-dashboard-extension/
EOF
```

有关详细信息，请参见第 8 章 “Fleet” 和 `fleet-examples` 储存库。

安装扩展后，它们将列在 **Installed**（已安装）选项卡下的 **Extensions**（扩展）部分中。由于它们不是通过 Apps/Marketplace 安装的，因此带有 `Third-Party`（第三方）标签。



6.2 KubeVirt 仪表板扩展

KubeVirt 扩展为 Rancher 仪表板 UI 提供基本虚拟机管理。第 21.7.2 节 “使用 KubeVirt Rancher 仪表板扩展”中介绍了其功能。

6.3 Akri 仪表板扩展

Akri 是一个 Kubernetes 资源接口，让您可以轻松地将异构叶设备（例如 IP 摄像头和 USB 设备）公开为 Kubernetes 群集中的资源，同时还支持公开 GPU 和 FPGA 等嵌入式硬件资源。Akri 会持续检测有权访问这些设备的节点，并根据这些节点调度工作负载。

Akri 仪表板扩展允许您使用 Rancher 仪表板用户界面来管理和监控叶设备，并在发现这些设备后运行工作负载。

第 14.5 节 “Akri Rancher 仪表板扩展”中详细介绍了扩展功能。

7 Rancher Turtles

请参见 Rancher Turtles 文档，网址为：https://documentation.suse.com/cloudnative/cluster-api/

> Rancher Turtles 是一个 Kubernetes 操作器，它提供 Rancher Manager 与 Cluster API (CAPI) 之间的集成，旨在为 Rancher 添加全面的 CAPI 支持

7.1 Rancher Turtles 的主要功能

- 通过在 CAPI 置备的群集中安装 Rancher 群集代理，自动将 CAPI 群集导入 Rancher。
- 通过 CAPI 操作器 (https://cluster-api-operator.sigs.k8s.io/) 安装并配置 CAPI 控制器依赖项。

7.2 Rancher Turtles 在 SUSE Edge 中的使用

SUSE Edge 堆栈提供了一个 Helm 封装 chart，该 chart 可使用特定配置安装 Rancher Turtles，以启用：

- 核心 CAPI 控制器组件
- RKE2 控制平面和引导提供程序组件
- Metal3（第 10 章 “Metal³”）基础架构提供程序组件

仅支持通过封装 chart 安装的默认提供程序 - 目前不支持替代的控制平面、引导和基础架构提供程序作为 SUSE Edge 堆栈的一部分。

7.3 安装 Rancher Turtles

可按照 Metal3 快速入门（第 1 章 “使用 Metal³ 实现 BMC 自动化部署”）指南或管理群集（第 40 章 “设置管理群集”）文档所述安装 Rancher Turtles。

7.4 其他资源

- Rancher 文档 (https://rancher.com/docs/)
- 《Cluster API Book》 (https://cluster-api.sigs.k8s.io/)

8 Fleet

Fleet (https://fleet.rancher.io) 是一个容器管理和部署引擎，旨在让用户更好地控制本地群集，并通过 GitOps 进行持续监控。Fleet 不仅注重缩放能力，而且还为用户提供很高的控制度和可见性，以准确监控群集上安装的组件。

Fleet 可以管理通过原始 Kubernetes YAML、Helm chart、Kustomize 的 Git 软件包或这三者的任意组合完成的部署。无论来源是什么，所有资源都会动态转换为 Helm chart，并使用 Helm 作为引擎在群集中部署所有资源。因此，用户可以获享很高的群集控制度、一致性和可审计性。

有关 Fleet 工作原理的信息，请参见 Fleet Architecture (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/architecture)。

8.1 使用 Helm 安装 Fleet

Fleet 已内置于 Rancher 中，但您也可以使用 Helm 将 Fleet 安装 (https://fleet.rancher.io/installation) 为任何 Kubernetes 群集上的独立应用程序。

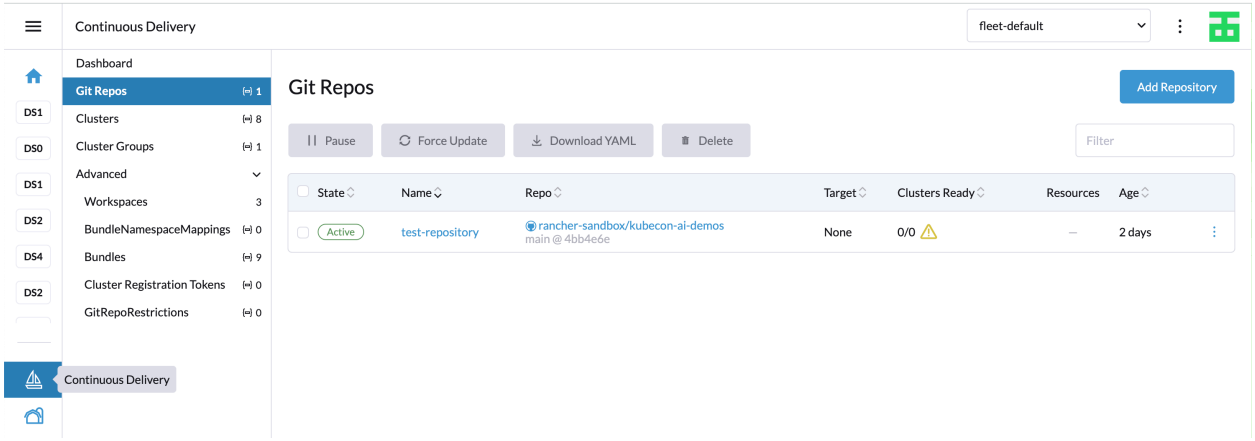
8.2 使用 Rancher 中的 Fleet

Rancher 使用 Fleet 在受管群集中部署应用程序。Fleet 的持续交付功能引入了大规模 GitOps，旨在管理在大量群集上运行的应用程序。

Fleet 作为 Rancher 的集成部分优势显著：使用 Rancher 管理的群集会在安装/导入过程中自动由 Fleet 代理部署，并且之后群集立即便能由 Fleet 管理。

8.3 在 Rancher UI 中访问 Fleet

Rancher 中预安装了 Fleet，可通过 Rancher UI 中的 **Continuous Delivery**（持续交付）选项进行管理。



“Continuous Delivery”（持续交付）部分包括以下项目：

8.3.1 Dashboard（仪表板）

所有工作空间中所有 GitOps 储存库的概览页面。仅显示包含储存库的工作空间。

8.3.2 Git repos（Git 储存库）

所选工作空间中的 GitOps 储存库列表。使用页面顶部的下拉列表选择活动工作空间。

8.3.3 Clusters（群集）

受管群集列表。默认情况下，Rancher 管理的所有群集都会添加到 `fleet-default` 工作空间。`fleet-local` 工作空间包含本地（管理）群集。在此处可以暂停或强制更新群集，或者将群集移动到另一个工作空间。可以通过编辑群集来更新用于群集分组的标签和注解。

8.3.4 Cluster groups（群集组）

在此部分，可以使用选择器对工作空间内的群集进行自定义分组。

8.3.5 Advanced（高级）

在“Advanced”（高级）部分可以管理工作空间和其他相关 Fleet 资源。

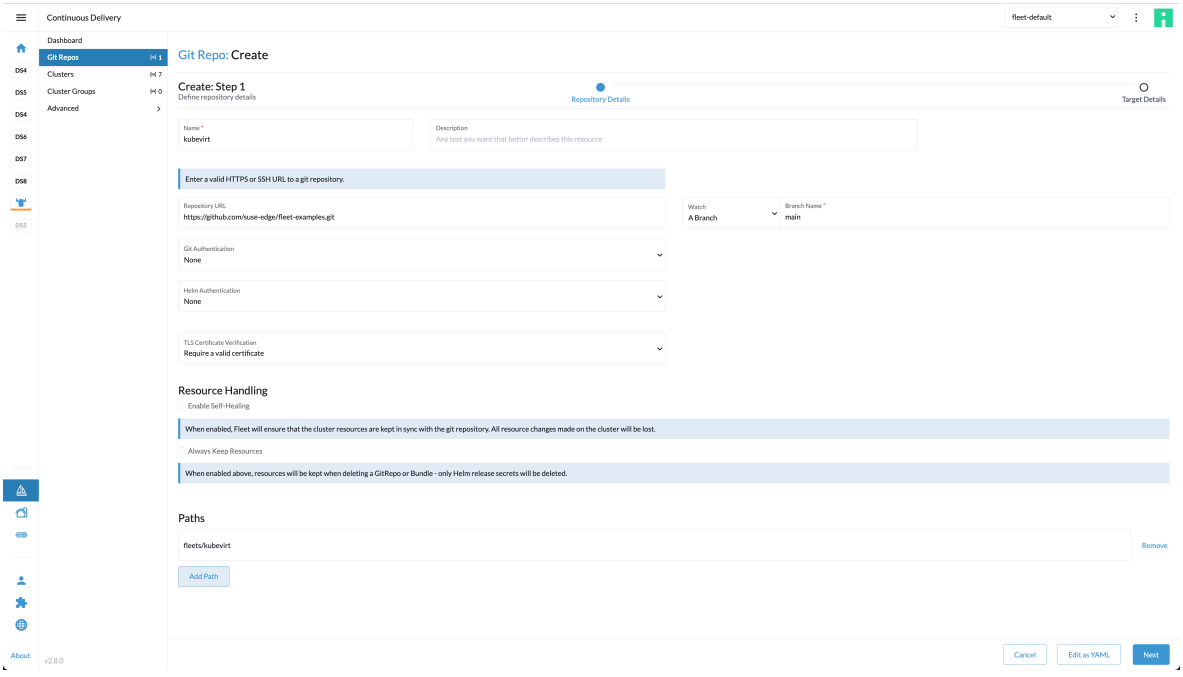
8.4 使用 Rancher 仪表板通过 Rancher 和 Fleet 安装 KubeVirt 的示例

1. 创建包含 `fleet.yaml` 文件的 Git 储存库：

```
defaultNamespace: kubevirt
helm:
  chart: "oci://registry.suse.com/edge/charts/kubevirt"
  version: "303.0.0+up0.5.0"
  # kubevirt namespace is created by kubevirt as well, we need to take
 ownership of it
  takeOwnership: true
```

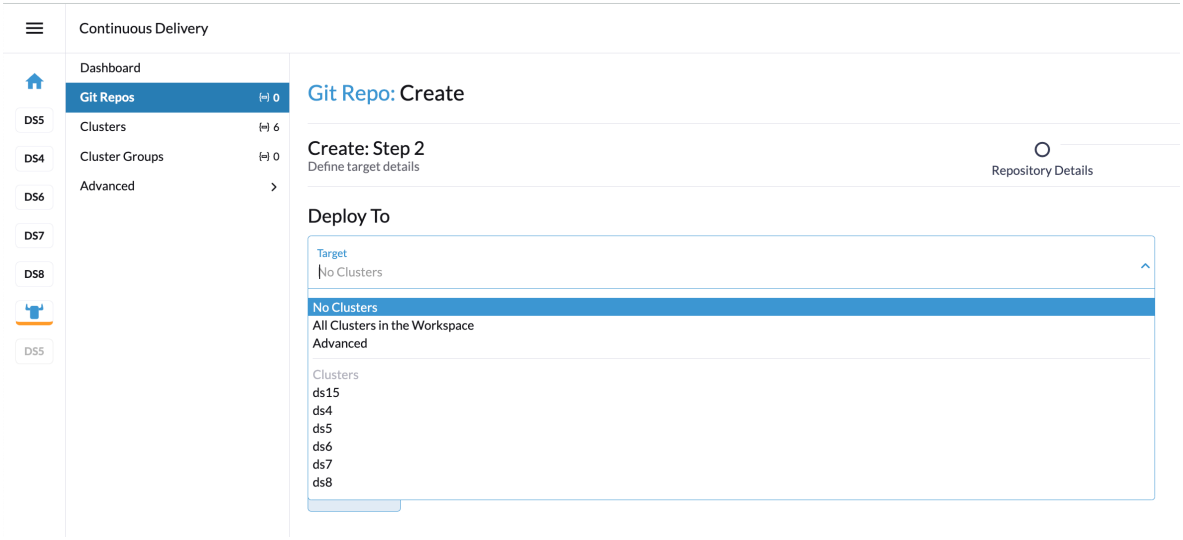
2. 在 Rancher 仪表板中，导航到 **# > Continuous Delivery（持续交付）> Git repos（Git 储存库）**，然后单击 `Add Repository`（添加储存库）。

3. 储存库创建向导将指导您完成 Git 储存库创建步骤。提供**名称**、**储存库 URL**（引用上一步骤中创建的 Git 储存库），并选择适当的分支或修订版。对于较复杂的储存库，请指定**路径**以便在单个储存库中使用多个目录。



4. 单击 `Next`（下一步）。

5. 在下一步骤中，可以定义工作负载的部署位置。在群集选择方面，可以使用多个基本选项：可以不选择任何群集、选择所有群集，或者直接选择特定的受管群集或群集组（如果已定义）。“Advanced”（高级）选项允许通过 YAML 直接编辑选择器。

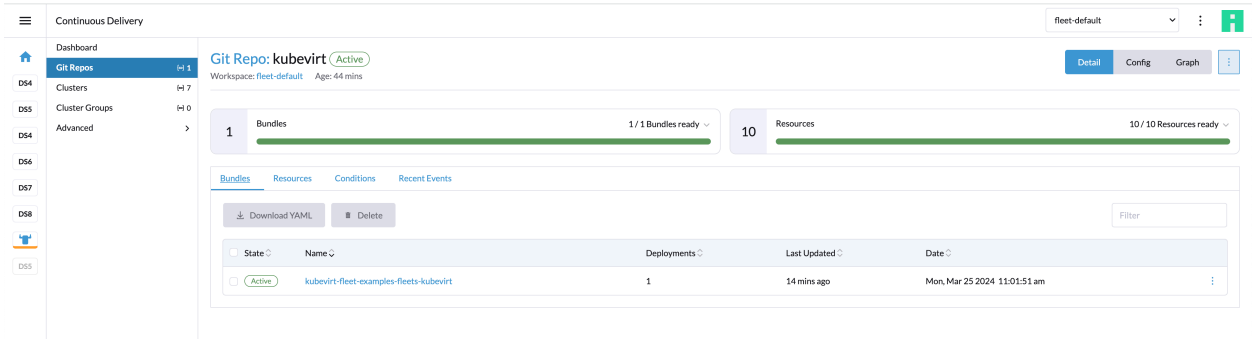


6. 单击 `Create`（创建）。系统即会创建储存库。从现在起，工作负载将在与储存库定义匹配的群集上安装并保持同步。

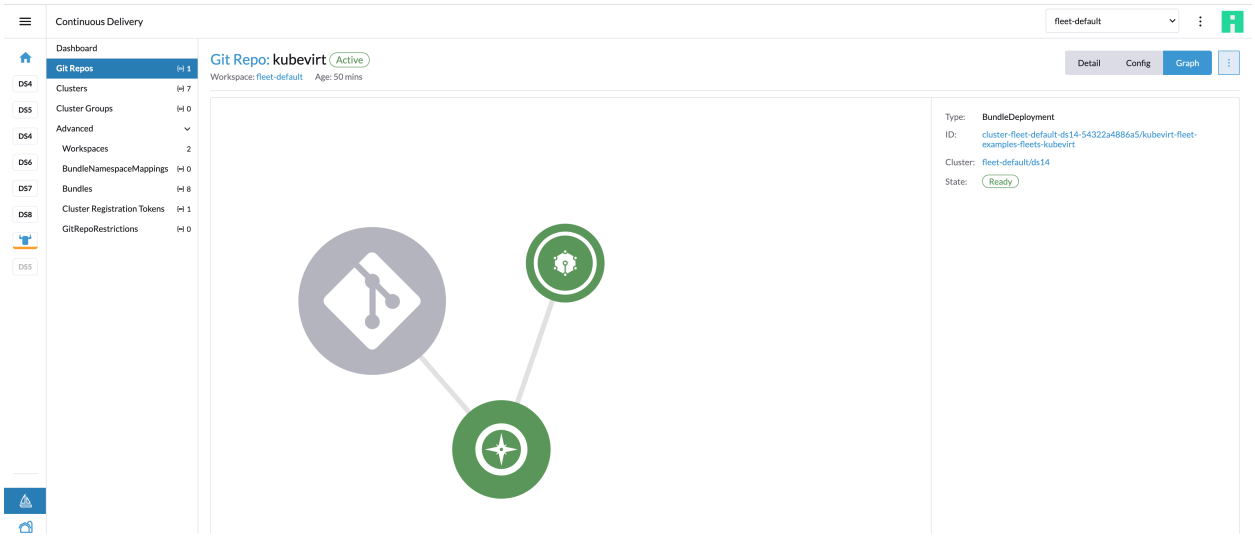
8.5 调试和查错

“Advanced”（高级）导航部分提供了低层级 Fleet 资源的概览。捆绑包 (https://fleet.rancher.io/ref-bundle-stages) 是一个内部资源，用于编排 Git 中的资源。扫描 Git 储存库时，会生成一个或多个捆绑包。

要查找与特定储存库相关的捆绑包，请转到 Git 储存库细节页面，并单击 `Bundles`（捆绑包）选项卡。



对于每个群集，捆绑包将应用于创建的 BundleDeployment 资源。要查看 BundleDeployment 细节，请单击 Git 储存库细节页面右上角的 Graph（图表）按钮。系统随即会加载 **Repo（储存库）> Bundles（捆绑包）> BundleDeployment** 图表。在图表中单击相应 BundleDeployment 可查看其细节，单击 ID 可查看 BundleDeployment YAML。



有关 Fleet 查错提示的更多信息，请参见此处 (https://fleet.rancher.io/troubleshooting)。

8.6 Fleet 示例

Edge 团队维护的一个储存库 (https://github.com/suse-edge/fleet-examples) 包含了有关使用 Fleet 安装 Edge 项目的示例。

Fleet 项目包含 fleet-examples (https://github.com/rancher/fleet-examples) 储存库，其中涵盖了 Git 储存库结构 (https://fleet.rancher.io/gitrepo-content) 的所有使用场景。

9 SUSE Linux Micro

请参见 SUSE Linux Micro 官方文档 (https://documentation.suse.com/sle-micro/6.1/)

SUSE Linux Micro 是一个安全的轻量级边缘操作系统。它将 SUSE Linux Enterprise 强化组件与开发人员需要的各种功能融入一套不可变的新式操作系统，从而形成一个可靠的基础架构平台，不仅具有同类最佳的合规性，而且易于使用。

9.1 SUSE Edge 如何使用 SUSE Linux Micro?

我们使用 SUSE Linux Micro 作为平台堆栈的基础操作系统。它为我们提供了安全、稳定且精简的构建基础。

SUSE Linux Micro 的独特之处在于它使用了文件系统 (Btrfs) 快照，一旦在升级期间出错，我们就可以轻松回滚。这样，在出现问题时，即使无法进行物理访问，也可以对整个平台安全地进行远程升级。

9.2 最佳实践

9.2.1 安装媒体

SUSE Edge 使用 Edge Image Builder（第 11 章 “Edge Image Builder”）来预配置 SUSE Linux Micro 自安装映像。

9.2.2 本地管理

SUSE Linux Micro 附带 Cockpit，让您可以通过 Web 应用程序对主机进行本地管理。

此服务默认已禁用，但可以通过启用 systemd 服务 `cockpit.socket` 来启动。

9.3 已知问题

- SUSE Linux Micro 目前不提供桌面环境，但我们正在开发容器化解决方案。

10 Metal³

Metal³ (https://metal3.io/) 是一个 CNCF 项目，它为 Kubernetes 提供裸机基础架构管理功能。

Metal³ 提供 Kubernetes 原生资源来管理裸机服务器的生命周期，支持通过 Redfish (https://www.dmtf.org/standards/redfish) 等带外协议进行管理。

它还为 Cluster API (CAPI) (https://cluster-api.sigs.k8s.io/) 提供成熟的支持，允许通过广泛采用的不限供应商的 API 来管理跨多个基础架构提供商的基础架构资源。

10.1 SUSE Edge 如何使用 Metal³?

此方法非常适合用于目标硬件支持带外管理，并且需要全自动化基础架构管理流程的场景。

此方法提供声明性 API 来对裸机服务器进行清单和状态管理，包括自动检查、清理和置备/取消置备。

10.2 已知问题

- 上游 IP 地址管理控制器 (https://github.com/metal3-io/ip-address-manager) 目前不受支持，因为它与我们选择的网络配置工具尚不兼容。
- 相关的 IPAM 资源和 Metal3DataTemplate networkData 字段也不受支持。
- 目前仅支持通过 redfish-virtualmedia 进行部署。

11 Edge Image Builder

请参见官方储存库 (https://github.com/suse-edge/edge-image-builder)。

Edge Image Builder (EIB) 工具可以简化为引导计算机生成自定义的随时可引导 (CRB) 磁盘映像的过程。使用一个这样的映像就能实现整个 SUSE 软件堆栈的端到端部署。

虽然 EIB 能够为所有置备场景创建 CRB 映像，但 EIB 在网络受限或完全隔离的隔离式部署中展现出极大价值。

11.1 SUSE Edge 如何使用 Edge Image Builder?

SUSE Edge 使用 EIB 来简化和快速配置适用于多种场景的自定义 SUSE Linux Micro 映像。这些场景包括为虚拟机和裸机引导部署，具体如下：

- K3s/RKE2 Kubernetes 的完全隔离式部署（单节点和多节点）
- 完全隔离式 Helm chart和 Kubernetes 清单部署
- 通过 Elemental API 注册到 Rancher
- Metal^3
- 自定义网络（例如静态 IP、主机名、VLAN、绑定等）
- 自定义操作系统配置（例如用户、组、口令、SSH 密钥、代理、NTP、自定义 SSL 证书等）
- 主机级和侧载 RPM 软件包的隔离式安装（包括依赖项解析）
- 注册到 SUSE Multi-Linux Manager 以进行操作系统管理
- 嵌入式容器映像
- 内核命令行参数
- 引导时启用/禁用的 Systemd 单元
- 用于任何手动任务的自定义脚本和文件

11.2 入门

有关 Edge Image Builder 用法和测试的综合文档可在此处 (https://github.com/suse-edge/edge-image-builder/tree/release-1.2/docs) 找到。

另外请参见第 3 章 “使用 Edge Image Builder 配置独立群集”，其中介绍了一种基本部署场景。

当您熟悉此工具后，可在我们的 Tips and tricks (../tips/eib.adoc) 网页中查找一些更有用的信息。

11.3 已知问题

- EIB 通过模板化 Helm chart 并分析模板中的所有映像来隔离 Helm chart。如果 Helm chart 未将其所有映像保留在模板中，而是侧载映像，则 EIB 将无法自动隔离这些映像。可通过手动将任何未检测到的映像添加到定义文件的 `embeddedArtifactRegistry` 部分，来解决此问题。

12 边缘网络

本章介绍 SUSE Edge 解决方案中的网络配置方法。我们将展示如何以声明方式在 SUSE Linux Micro 上配置 NetworkManager，并说明如何集成相关的工具。

12.1 NetworkManager 概述

NetworkManager 是用于管理主网络连接和其他连接接口的工具。

NetworkManager 将网络配置存储为包含期望状态的连接文件。这些连接以文件的形式存储在 `/etc/NetworkManager/system-connections/` 目录中。

有关 NetworkManager 的详细信息，请参见 SUSE Linux Micro 文档 (https://documentation.suse.com/sle-micro/6.1/html/Micro-network-configuration/index.html)。

12.2 nmstate 概述

nmstate 是广泛采用的库（附带 CLI 工具），它提供一个声明性 API，可用于通过预定义的纲要进行网络配置。

有关 nmstate 的详细信息，请参见上游文档 (https://nmstate.io/)。

12.3 NetworkManager Configurator (nmc) 概述

SUSE Edge 中提供的网络自定义选项通过一个称为 NetworkManager Configurator（简称为 **nmc**）的 CLI 工具实现。此工具利用 nmstate 库提供的功能，因此完全能够配置静态 IP 地址、DNS 服务器、VLAN、绑定、网桥等。我们可以使用此工具根据预定义的期望状态生成网络配置，并自动将这些配置应用于许多不同的节点。

有关 NetworkManager Configurator (nmc) 的详细信息，请参见上游储存库 (https://github.com/suse-edge/nm-configurator)。

12.4 SUSE Edge 如何使用 NetworkManager Configurator?

SUSE Edge 利用 **nmc** 在各种不同的置备模型中进行网络自定义：

- 定向网络置备场景中的自定义网络配置（第 1 章 “使用 Metal^3 实现 BMC 自动化部署”）
- 基于映像的置备场景中的声明性静态配置（第 3 章 “使用 Edge Image Builder 配置独立群集”）

12.5 使用 Edge Image Builder 进行配置

Edge Image Builder (EIB) 是可用于通过单个操作系统映像配置多个主机的工具。本节将介绍如何使用声明式方法来描述期望的网络状态，如何将这些状态转换为相应的 NetworkManager 连接，然后在置备过程中应用这些连接。

12.5.1 先决条件

如果您要学习本指南，事先需要做好以下准备：

- 一台运行 SLES 15 SP6 或 openSUSE Leap 15.6 的 AMD64/Intel 64 物理主机（或虚拟机）
- 一个可用的容器运行时（例如 Podman）
- SUSE Linux Micro 6.1 原始映像，可在此处 (https://www.suse.com/download/sle-micro/) 获取

12.5.2 获取 Edge Image Builder 容器映像

EIB 容器映像已公开提供，可以通过运行以下命令从 SUSE Edge 注册表下载：

```
podman pull registry.suse.com/edge/3.3/edge-image-builder:1.2.1
```

12.5.3 创建映像配置目录

我们来开始创建配置目录：

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR/base-images
```

确保下载的基础映像已移动到配置目录：

```
mv /path/to/downloads/SL-Micro.x86_64-6.1-Base-GM.raw $CONFIG_DIR/base-images/
```

注意

EIB 永远不会修改基础映像输入。它会创建一个包含所需修改内容的新映像。

此时，配置目录应如下所示：

```
└── base-images/
    └── SL-Micro.x86_64-6.1-Base-GM.raw
```

12.5.4 创建映像定义文件

定义文件描述了 Edge Image Builder 支持的大多数可配置选项。

首先，我们为操作系统映像创建一个非常简单的定义文件：

```
cat << EOF > $CONFIG_DIR/definition.yaml
apiVersion: 1.2
image:
  arch: x86_64
  imageType: raw
  baseImage: SL-Micro.x86_64-6.1-Base-GM.raw
  outputImageName: modified-image.raw
operatingSystem:
  users:
    - username: root
```

```
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
EOF
```

必须包含 `image` 部分，它指定输入映像、输入映像的体系结构和类型，以及输出映像的名称。`operatingSystem` 是可选部分，其中包含的配置允许用户通过 `root/eib` 用户名/口令登录到置备的系统。

注意

您可以运行 `openssl passwd -6 <password>` 来使用自己的已加密口令。

此时，配置目录应如下所示：

```
├── definition.yaml
└── base-images/
    └── SL-Micro.x86_64-6.1-Base-GM.raw
```

12.5.5 定义网络配置

期望的网络配置不是我们刚刚创建的映像定义文件的一部分。现在我们在特殊的 `network/` 目录下填充这些配置。我们来创建该目录：

```
mkdir -p $CONFIG_DIR/network
```

如前所述，NetworkManager Configurator (**nmc**) 工具要求提供预定义纲要形式的输入。您可以参见上游 NMState 示例文档 (https://nmstate.io/examples.html)，了解如何设置各种不同的网络选项。

本指南将介绍如何在三个不同的节点上配置网络：

- 使用两个以太网接口的节点
- 使用网络绑定的节点
- 使用网桥的节点

警告

不建议在生产构建中使用完全不同的网络设置，尤其是在配置 Kubernetes 群集时。网络配置通常应在节点之间或至少在给定群集内的角色之间保持同质。本指南包含的各种不同选项仅供参考。

注意

以下示例采用 IP 地址范围为 `192.168.122.1/24` 的默认 `libvirt` 网络。如果您的环境与此不同，请相应地进行调整。

我们来为第一个节点（名为 `node1.suse.com`）创建期望的状态：

```
cat << EOF > $CONFIG_DIR/network/node1.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: eth0
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address:
      next-hop-interface: eth0
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: eth0
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E1
    ipv4:
```

```
      address:
        - ip: 192.168.122.50
          prefix-length: 24
      dhcp: false
      enabled: true
    ipv6:
      enabled: false
  - name: eth3
    type: ethernet
    state: down
    mac-address: 34:8A:B1:4B:16:E2
    ipv4:
      address:
        - ip: 192.168.122.55
          prefix-length: 24
      dhcp: false
      enabled: true
    ipv6:
      enabled: false
EOF
```

在此示例中，我们将定义两个以太网接口（eth0 和 eth3）的期望状态、其请求 IP 地址、路由和 DNS 解析。

警告

必须确保列出了所有以太网接口的 MAC 地址。这些地址在置备过程中用作节点的标识符，用于确定要应用哪些配置。这就是我们使用单个 ISO 或 RAW 映像配置多个节点的方式。

接下来对第二个节点（名为 `node2.suse.com`）进行操作，该节点使用网络绑定：

```
cat << EOF > $CONFIG_DIR/network/node2.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
```

```
      next-hop-interface: bond99
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address:
      next-hop-interface: bond99
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: bond99
    type: bond
    state: up
    ipv4:
      address:
        - ip: 192.168.122.60
          prefix-length: 24
      enabled: true
    link-aggregation:
      mode: balance-rr
      options:
        miimon: '140'
      port:
        - eth0
        - eth1
  - name: eth0
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E3
    ipv4:
      enabled: false
    ipv6:
      enabled: false
  - name: eth1
    type: ethernet
```

```
    state: up
    mac-address: 34:8A:B1:4B:16:E4
    ipv4:
      enabled: false
    ipv6:
      enabled: false
EOF
```

在此示例中，我们将定义两个未启用 IP 寻址的以太网接口（eth0 和 eth1）的期望状态，以及采用轮替策略的绑定及其用于转发网络流量的相应地址。

最后，我们将创建第三个（也是最后一个）期望状态文件，该文件利用网桥，我们将其命名为 `node3.suse.com`：

```
cat << EOF > $CONFIG_DIR/network/node3.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: linux-br0
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address:
      next-hop-interface: linux-br0
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: eth0
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E5
    ipv4:
      enabled: false
```

```
    ipv6:
      enabled: false
  - name: linux-br0
    type: linux-bridge
    state: up
    ipv4:
      address:
        - ip: 192.168.122.70
          prefix-length: 24
      dhcp: false
      enabled: true
    bridge:
      options:
        group-forward-mask: 0
        mac-ageing-time: 300
        multicast-snooping: true
        stp:
          enabled: true
          forward-delay: 15
          hello-time: 2
          max-age: 20
          priority: 32768
      port:
        - name: eth0
          stp-hairpin-mode: false
          stp-path-cost: 100
          stp-priority: 32
EOF
```

此时，配置目录应如下所示：

```
├── definition.yaml
├── network/
│   ├── node1.suse.com.yaml
│   ├── node2.suse.com.yaml
│   └── node3.suse.com.yaml
└── base-images/
    └── SL-Micro.x86_64-6.1-Base-GM.raw
```

注意

network/ 目录下的文件名是特意设定的，它们与置备过程中将要设置的主机名相对应。

12.5.6 构建操作系统映像

完成所有必要配置后，接下来我们只需运行以下命令来构建映像：

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.3/edge-image-
builder:1.2.1 build --definition-file definition.yaml
```

输出应如下所示：

```
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Systemd ...................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Embedded Artifact Registry ... [SKIPPED]
Keymap ....................... [SUCCESS]
Kubernetes ................... [SKIPPED]
Certificates ................. [SKIPPED]
Building RAW image...
Kernel Params ................ [SKIPPED]
Image build complete!
```

以上输出片段告诉我们网络组件已成功配置，我们可以继续置备边缘节点。

注意

可以在所运行映像的带时间戳目录下生成的 `_build` 目录中，检查日志文件 (`network-config.log`) 和相应的 NetworkManager 连接文件。

12.5.7 置备边缘节点

我们来复制生成的原始映像：

```
mkdir edge-nodes && cd edge-nodes
for i in {1..4}; do cp $CONFIG_DIR/modified-image.raw node$i.raw; done
```

您会发现，我们复制了构建的映像四次，但仅指定了三个节点的网络配置。这是因为，我们还想展示当置备的节点与任何期望的配置都不匹配时会发生什么。

注意

本指南将为节点置备示例使用虚拟化。请确保在 BIOS 中启用必要的扩展（有关详细信息，请参见此处 (https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-virt-support.html#sec-kvm-requires-hardware)）。

我们将运行 `virt-install` 并使用复制的原始磁盘来创建虚拟机。每个虚拟机使用 10 GB RAM 和 6 个 vCPU。

12.5.7.1 置备第一个节点

我们来创建虚拟机：

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=node1.raw,format=raw
 --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default,mac=34:8A:B1:4B:16:E1 --network
 default,mac=34:8A:B1:4B:16:E2 --virt-type kvm --import
```

注意

要创建的网络接口的 MAC 地址必须与上述期望的状态中使用的 MAC 地址相同。

操作完成后，我们将看到如下所示的输出：

```
Starting install...
Creating domain...

Running text console command: virsh --connect qemu:///system console node1
Connected to domain 'node1'
Escape character is ^] (Ctrl + ])



Welcome to SUSE Linux Micro 6.0 (x86_64) - Kernel 6.4.0-18-default (tty1).

SSH host key: SHA256:XN/R5Tw43reG+QsOw480LxCnhkc/1uqMdwlI6KUBY70 (RSA)
SSH host key: SHA256:/96yGrPGKlhn04f1rb9cXv/2WJt4TtrIN5yEcN66r3s (DSA)
SSH host key: SHA256:Dy/YjBQ7LwjZGaaVcMhTWZNSOstxXBsPsvgJTJq5t00 (ECDSA)
SSH host key: SHA256:TNGqY1LRddpxD/jn/8dkT/9YmVl9hiwulqmayP+wOWQ (ED25519)
eth0: 192.168.122.50
eth1:



Configured with the Edge Image Builder
Activate the web console with: systemctl enable --now cockpit.socket

node1 login:
```

现在我们可以使用 `root:eib` 身份凭证对登录。如果需要，我们也可以通过 SSH 连接到主机，而不是如此处所示使用 `virsh 控制台`进行连接。

登录后，我们需要确认是否正确完成了所有设置。

校验是否正确设置了主机名：

```
node1:~ # hostnamectl
 Static hostname: node1.suse.com
```

```
 ...
```

校验是否正确配置了路由：

```
node1:~ # ip r
default via 192.168.122.1 dev eth0 proto static metric 100
192.168.122.0/24 dev eth0 proto static scope link metric 100
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.50 metric 100
```

校验互联网连接是否可用：

```
node1:~ # ping google.com
PING google.com (142.250.72.78) 56(84) bytes of data.
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=1 ttl=56
 time=13.2 ms
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=2 ttl=56
 time=13.4 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 13.248/13.304/13.361/0.056 ms
```

校验是否刚好配置了两个以太网接口，并且只有其中的一个接口处于活动状态：

```
node1:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
    link/ether 34:8a:b1:4b:16:e1 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.50/24 brd 192.168.122.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
```

```
    link/ether 34:8a:b1:4b:16:e2 brd ff:ff:ff:ff:ff:ff
    altname enp0s3
    altname ens3

node1:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                  TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/
NetworkManager/system-connections/eth0.nmconnection
eth1  7e211aea-3d14-59cf-a4fa-be91dac5dbba  ethernet  --      /etc/
NetworkManager/system-connections/eth1.nmconnection
```

您会发现，在期望的网络状态中，第二个接口是 `eth1`，而不是预定义的 `eth3`。之所以出现这种情况，是因为 NetworkManager Configurator (**nmc**) 能够检测到操作系统为 MAC 地址为 `34:8a:b1:4b:16:e2` 的 NIC 指定了不同的名称，并相应地调整了其设置。

通过检查置备的 Combustion 阶段来校验是否确实发生了这种情况：

```
node1:~ # journalctl -u combustion | grep nmc
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc::apply_conf] Identified host: node1.suse.com
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc::apply_conf] Set hostname: node1.suse.com
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc::apply_conf] Processing interface 'eth0'...
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc::apply_conf] Processing interface 'eth3'...
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc::apply_conf] Using interface name 'eth1' instead of the preconfigured
 'eth3'
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z
 INFO  nmc] Successfully applied config
```

我们现在将置备其余节点，但只会显示最终配置中的差异。您可以对即将置备的所有节点应用上述任意检查或所有检查。

12.5.7.2 置备第二个节点

我们来创建虚拟机：

```
virt-install --name node2 --ram 10000 --vcpus 6 --disk path=node2.raw,format=raw
 --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default,mac=34:8A:B1:4B:16:E3 --network
 default,mac=34:8A:B1:4B:16:E4 --virt-type kvm --import
```

虚拟机启动并运行后，我们可以确认此节点是否正在使用绑定的接口：

```
node2:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
 master bond99 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
3: eth1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
 master bond99 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff permaddr
 34:8a:b1:4b:16:e4
    altname enp0s3
    altname ens3
4: bond99: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc noqueue state
 UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.60/24 brd 192.168.122.255 scope global noprefixroute bond99
       valid_lft forever preferred_lft forever
```

确认路由是否正在使用绑定：

```
node2:~ # ip r
default via 192.168.122.1 dev bond99 proto static metric 100
192.168.122.0/24 dev bond99 proto static scope link metric 100
192.168.122.0/24 dev bond99 proto kernel scope link src 192.168.122.60 metric
 300
```

确保正确利用静态连接文件：

```
node2:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME    UUID                                  TYPE      DEVICE  FILENAME
bond99  4a920503-4862-5505-80fd-4738d07f44c6  bond      bond99  /etc/
NetworkManager/system-connections/bond99.nmconnection
eth0    dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/
NetworkManager/system-connections/eth0.nmconnection
eth1    0523c0a1-5f5e-5603-bcf2-68155d5d322e  ethernet  eth1    /etc/
NetworkManager/system-connections/eth1.nmconnection
```

12.5.7.3 置备第三个节点

我们来创建虚拟机：

```
virt-install --name node3 --ram 10000 --vcpus 6 --disk path=node3.raw,format=raw
 --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default,mac=34:8A:B1:4B:16:E5 --virt-type kvm
 --import
```

虚拟机启动并运行后，我们可以确认此节点是否正在使用网桥：

```
node3:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master
 linux-br0 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e5 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
3: linux-br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
 group default qlen 1000
    link/ether 34:8a:b1:4b:16:e5 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.122.70/24 brd 192.168.122.255 scope global noprefixroute linux-
br0
       valid_lft forever preferred_lft forever
```

确认路由是否正在使用网桥：

```
node3:~ # ip r
default via 192.168.122.1 dev linux-br0 proto static metric 100
192.168.122.0/24 dev linux-br0 proto static scope link metric 100
192.168.122.0/24 dev linux-br0 proto kernel scope link src 192.168.122.70 metric
 425
```

确保正确利用静态连接文件：

```
node3:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME       UUID                                  TYPE      DEVICE     FILENAME
linux-br0  1f8f1469-ed20-5f2c-bacb-a6767bee9bc0  bridge    linux-br0  /etc/
NetworkManager/system-connections/linux-br0.nmconnection
eth0       dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0       /etc/
NetworkManager/system-connections/eth0.nmconnection
```

12.5.7.4 置备第四个节点

最后，我们将置备 MAC 地址与任何预定义配置都不匹配的节点。在这种情况下，我们将默认使用 DHCP 来配置网络接口。

我们来创建虚拟机：

```
virt-install --name node4 --ram 10000 --vcpus 6 --disk path=node4.raw,format=raw
 --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default --virt-type kvm --import
```

虚拟机启动并运行后，我们可以确认此节点是否正在为其网络接口使用随机 IP 地址：

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

```
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
    link/ether 52:54:00:56:63:71 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.86/24 brd 192.168.122.255 scope global dynamic
 noprefixroute eth0
       valid_lft 3542sec preferred_lft 3542sec
    inet6 fe80::5054:ff:fe56:6371/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

校验 nmc 是否无法为此节点应用静态配置：

```
localhost:~ # journalctl -u combustion | grep nmc
Apr 23 12:15:45 localhost.localdomain combustion[1357]: [2024-04-23T12:15:45Z
 ERROR nmc] Applying config failed: None of the preconfigured hosts match local
 NICs
```

校验是否通过 DHCP 配置了以太网接口：

```
localhost:~ # journalctl | grep eth0
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7801] manager: (eth0): new Ethernet device (/org/freedesktop/
NetworkManager/Devices/2)
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7802] device (eth0): state change: unmanaged -> unavailable (reason
 'managed', sys-iface-state: 'external')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7929] device (eth0): carrier: link connected
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7931] device (eth0): state change: unavailable -> disconnected
 (reason 'carrier-changed', sys-iface-state: 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7944] device (eth0): Activation: starting connection 'Wired
 Connection' (300ed658-08d4-4281-9f8c-d1b8882d29b9)
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7945] device (eth0): state change: disconnected -> prepare (reason
 'none', sys-iface-state: 'managed')
```

```
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7947] device (eth0): state change: prepare -> config (reason
 'none', sys-iface-state: 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7953] device (eth0): state change: config -> ip-config (reason
 'none', sys-iface-state: 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7964] dhcp4 (eth0): activation: beginning transaction (timeout in
 90 seconds)
Apr 23 12:15:33 localhost.localdomain NetworkManager[704]: <info>
 [1713874533.1272] dhcp4 (eth0): state changed new lease, address=192.168.122.86

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME              UUID                                  TYPE      DEVICE
 FILENAME
Wired Connection  300ed658-08d4-4281-9f8c-d1b8882d29b9  ethernet  eth0    /var/
run/NetworkManager/system-connections/default_connection.nmconnection
```

12.5.8 统一节点配置

有时我们无法依赖已知的 MAC 地址。在这种情况下，我们可以选择所谓的**统一配置**，这样就可以在 `_all.yaml` 文件中指定设置，然后将这些设置应用于所有已置备的节点。

我们将使用不同的配置结构来构建和置备边缘节点。请按照从第 12.5.3 节 “创建映像配置目录”到第 12.5.5 节 “定义网络配置”的所有步骤进行操作。

在此示例中，我们将定义两个以太网接口（eth0 和 eth1）的期望状态 - 一个接口使用 DHCP，另一个接口使用静态 IP 地址。

```
mkdir -p $CONFIG_DIR/network

cat <<- EOF > $CONFIG_DIR/network/_all.yaml
interfaces:
- name: eth0
  type: ethernet
  state: up
  ipv4:
    dhcp: true
```

```
    enabled: true
  ipv6:
    enabled: false
- name: eth1
  type: ethernet
  state: up
  ipv4:
    address:
    - ip: 10.0.0.1
      prefix-length: 24
    enabled: true
    dhcp: false
  ipv6:
    enabled: false
EOF
```

我们来构建映像：

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.3/edge-image-
builder:1.2.1 build --definition-file definition.yaml
```

成功构建映像后，我们将使用它来创建虚拟机：

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=$CONFIG_DIR/
modified-image.raw,format=raw --osinfo detect=on,name=sle-unknown --graphics
 none --console pty,target_type=serial --network default --network default --
virt-type kvm --import
```

置备过程可能需要几分钟时间。完成后，使用提供的身份凭证登录到系统。

校验是否正确配置了路由：

```
localhost:~ # ip r
default via 192.168.122.1 dev eth0 proto dhcp src 192.168.122.100 metric 100
10.0.0.0/24 dev eth1 proto kernel scope link src 10.0.0.1 metric 101
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.100 metric 100
```

校验互联网连接是否可用：

```
localhost:~ # ping google.com
PING google.com (142.250.72.46) 56(84) bytes of data.
```

```
64 bytes from den16s08-in-f14.1e100.net (142.250.72.46): icmp_seq=1 ttl=56
 time=14.3 ms
64 bytes from den16s08-in-f14.1e100.net (142.250.72.46): icmp_seq=2 ttl=56
 time=14.2 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 14.196/14.260/14.324/0.064 ms
```

校验以太网接口是否已配置并处于活动状态：

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
    link/ether 52:54:00:26:44:7a brd ff:ff:ff:ff:ff:ff
    altname enp1s0
    inet 192.168.122.100/24 brd 192.168.122.255 scope global dynamic
 noprefixroute eth0
       valid_lft 3505sec preferred_lft 3505sec
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
    link/ether 52:54:00:ec:57:9e brd ff:ff:ff:ff:ff:ff
    altname enp7s0
    inet 10.0.0.1/24 brd 10.0.0.255 scope global noprefixroute eth1
       valid_lft forever preferred_lft forever

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                  TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/
NetworkManager/system-connections/eth0.nmconnection
eth1  0523c0a1-5f5e-5603-bcf2-68155d5d322e  ethernet  eth1    /etc/
NetworkManager/system-connections/eth1.nmconnection
```

```
localhost:~ # cat /etc/NetworkManager/system-connections/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
id=eth0
interface-name=eth0
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70

[ipv4]
dhcp-client-id=mac
dhcp-send-hostname=true
dhcp-timeout=2147483647
ignore-auto-dns=false
ignore-auto-routes=false
method=auto
never-default=false

[ipv6]
addr-gen-mode=0
dhcp-timeout=2147483647
method=disabled

localhost:~ # cat /etc/NetworkManager/system-connections/eth1.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
id=eth1
interface-name=eth1
type=802-3-ethernet
uuid=0523c0a1-5f5e-5603-bcf2-68155d5d322e

[ipv4]
address0=10.0.0.1/24
dhcp-timeout=2147483647
method=manual
```

```
[ipv6]
addr-gen-mode=0
dhcp-timeout=2147483647
method=disabled
```

12.5.9 自定义网络配置

我们已介绍 Edge Image Builder 的默认网络配置，这种配置依赖于 NetworkManager Configurator。不过，我们还可以选择通过自定义脚本来修改这种配置。虽然这种方法非常灵活而且不依赖于 MAC 地址，但它的局限性在于，在使用单个映像引导多个节点时，这种方法不太方便。

注意

建议通过 `/network` 目录下描述期望的网络状态的文件来使用默认网络配置。请仅在该行为不适用于您的使用场景时，才选择自定义脚本。

我们将使用不同的配置结构来构建和置备边缘节点。请按照从第 12.5.3 节 “创建映像配置目录”到第 12.5.5 节 “定义网络配置”的所有步骤进行操作。

在此示例中，我们将创建一个自定义脚本来对所有已置备节点上的 `eth0` 接口应用静态配置，同时去除并禁用 NetworkManager 自动创建的有线连接。如果您想要确保群集中每个节点都采用相同的网络配置，则这种做法非常有利，这样，您就不需要在创建映像之前考虑每个节点的 MAC 地址。

首先，我们将连接文件存储在 `/custom/files` 目录中：

```
mkdir -p $CONFIG_DIR/custom/files

cat << EOF > $CONFIG_DIR/custom/files/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
autoconnect-retries=1
id=eth0
interface-name=eth0
```

```
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70
wait-device-timeout=60000

[ipv4]
dhcp-timeout=2147483647
method=auto

[ipv6]
addr-gen-mode=eui64
dhcp-timeout=2147483647
method=disabled
EOF
```

创建静态配置后，我们再创建自定义网络脚本：

```
mkdir -p $CONFIG_DIR/network

cat << EOF > $CONFIG_DIR/network/configure-network.sh
#!/bin/bash
set -eux

# Remove and disable wired connections
mkdir -p /etc/NetworkManager/conf.d/
printf "[main]\nno-auto-default=*\n" > /etc/NetworkManager/conf.d/no-auto-
default.conf
rm -f /var/run/NetworkManager/system-connections/* || true

# Copy pre-configured network configuration files into NetworkManager
mkdir -p /etc/NetworkManager/system-connections/
cp eth0.nmconnection /etc/NetworkManager/system-connections/
chmod 600 /etc/NetworkManager/system-connections/*.nmconnection
EOF

chmod a+x $CONFIG_DIR/network/configure-network.sh
```

注意

默认情况下仍会包含 nmc 二进制文件，因此如果需要，也可以在 `configure-network.sh` 脚本中使用它。

警告

必须始终在配置目录中的 `/network/configure-network.sh` 下提供自定义脚本。系统将忽略存在的所有其他文件。因为无法同时使用 YAML 格式的静态配置和自定义脚本来配置网络。

此时，配置目录应如下所示：

```
├── definition.yaml
├── custom/
│   └── files/
│       └── eth0.nmconnection
├── network/
│   └── configure-network.sh
└── base-images/
    └── SL-Micro.x86_64-6.1-Base-GM.raw
```

我们来构建映像：

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.3/edge-image-
builder:1.2.1 build --definition-file definition.yaml
```

成功构建映像后，我们将使用它来创建虚拟机：

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=$CONFIG_DIR/
modified-image.raw,format=raw --osinfo detect=on,name=sle-unknown --graphics
 none --console pty,target_type=serial --network default --virt-type kvm --
import
```

置备过程可能需要几分钟时间。完成后，使用提供的身份凭证登录到系统。

校验是否正确配置了路由：

```
localhost:~ # ip r
```

```
default via 192.168.122.1 dev eth0 proto dhcp src 192.168.122.185 metric 100
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.185 metric 100
```

校验互联网连接是否可用：

```
localhost:~ # ping google.com
PING google.com (142.250.72.78) 56(84) bytes of data.
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=1 ttl=56
 time=13.6 ms
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=2 ttl=56
 time=13.6 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 13.592/13.599/13.606/0.007 ms
```

校验是否使用连接文件静态配置了以太网接口，并且该接口处于活动状态：

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
 default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
 group default qlen 1000
    link/ether 52:54:00:31:d0:1b brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.185/24 brd 192.168.122.255 scope global dynamic
 noprefixroute eth0

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                  TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/
NetworkManager/system-connections/eth0.nmconnection
```

```
localhost:~ # cat  /etc/NetworkManager/system-connections/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
autoconnect-retries=1
id=eth0
interface-name=eth0
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70
wait-device-timeout=60000

[ipv4]
dhcp-timeout=2147483647
method=auto

[ipv6]
addr-gen-mode=eui64
dhcp-timeout=2147483647
method=disabled
```

13 Elemental

Elemental 是一个软件堆栈，可用于通过 Kubernetes 实现集中式云原生操作系统全面管理。Elemental 堆栈由驻留在 Rancher 本身或边缘节点上的许多组件构成。核心组件包括：

- **elemental-operator** - 驻留在 Rancher 上的核心操作器，用于处理来自客户端的注册请求。
- **elemental-register** - 在边缘节点上运行的客户端，支持通过 `elemental-operator` 进行注册。
- **elemental-system-agent** - 驻留在边缘节点上的代理；其配置由 `elemental-register` 馈送，接收用于配置 `rancher-system-agent` 的计划。
- **rancher-system-agent** - 完全注册边缘节点后，此组件将接管 `elemental-system-agent`，并等待 Rancher Manager 接下来的计划（例如 Kubernetes 安装）。

有关 Elemental 及其与 Rancher 的关系的完整信息，请参见 Elemental 上游文档 (https://elemental.docs.rancher.com/)。

13.1 SUSE Edge 如何使用 Elemental?

我们将使用 Elemental 的部分功能来管理无法使用 Metal[3] 的远程设备（例如，没有 BMC 的设备，或者 NAT 网关后的设备）。在知道设备何时运送或运送到何处之前，操作员可以使用此工具在实验室中引导其设备。也就是说，我们利用 `elemental-register` 和 `elemental-system-agent` 组件将 SUSE Linux Micro 主机接入 Rancher，以支持“自主回连”网络置备使用场景。使用 Edge Image Builder (EIB) 创建部署映像时，可以通过在 EIB 的配置目录中指定注册配置，来使用 Elemental 通过 Rancher 完成自动注册。

注意

在 SUSE Edge 3.3.1 中，我们**不会**利用 Elemental 的操作系统管理功能，因此无法通过 Rancher 管理操作系统的修补。SUSE Edge 不会使用 Elemental 工具来构建部署映像，而是使用 Edge Image Builder 工具，后者利用注册配置。

13.2 最佳实践

13.2.1 安装媒体

SUSE Edge 建议的、可以在“自主回连网络置备”部署空间中利用 Elemental 注册到 Rancher 的部署映像构建方法是，遵循有关使用 Elemental 进行远程主机初始配置（第 2 章 “使用 Elemental 进行远程主机接入”）快速入门中详述的说明。

13.2.2 标签

Elemental 使用 `MachineInventory` CRD 跟踪其清单，并提供选择清单的方法，例如，根据标签选择要将 Kubernetes 群集部署到的计算机。这样，用户就可以在购买硬件之前，预定义其大部分（甚至所有）基础架构需求。另外，由于节点可以在其相应清单对象上添加/去除标签（结合附加标志 `--label "FOO=BAR"` 重新运行 `elemental-register`），我们可以编写脚本来发现节点的引导位置并告诉 Rancher。

13.3 已知问题

- Elemental UI 目前不知道如何构建安装媒体或更新非“Elemental Teal”操作系统。此问题在将来的版本中应会得到解决。

14 Akri

Akri 是一个 CNCF 沙箱项目，旨在发现叶设备并将其呈现为 Kubernetes 原生资源。它还允许为每个发现的设备调度一个 Pod 或作业。设备可以在节点本地，也可以联网，并可以使用多种协议。

有关 Akri 的上游文档，请访问 https://docs.akri.sh

14.1 SUSE Edge 如何使用 Akri?

警告

Akri 目前在 SUSE Edge 堆栈中以技术预览的形式提供。

每当需要发现叶设备以及针对叶设备调度工作负载时，都可以使用 Edge 堆栈中包含的 Akri。

14.2 安装 Akri

Akri 在 Edge Helm 储存库中作为 Helm chart 提供。建议的 Akri 配置方法是使用给定的 Helm chart 部署不同的组件（代理、控制器、发现处理程序），然后使用您偏好的部署机制部署 Akri 的配置 CRD。

14.3 配置 Akri

使用 `akri.sh/Configuration` 对象来配置 Akri，该对象包含有关如何发现设备，以及发现了匹配的设备时应执行什么操作的所有信息。

下面列出了示例配置的明细，其中解释了所有字段：

```
apiVersion: akri.sh/v0
kind: Configuration
metadata:
```

```
  name: sample-configuration
spec:
```

此部分描述发现处理程序的配置，您必须指定处理程序的名称（作为 Akri chart 一部分提供的处理程序包括 `udev`、`opcua`、`onvif`）。`discoveryDetails` 与特定的处理程序相关，有关其配置方法，请参见处理程序的文档。

```
  discoveryHandler:
    name: debugEcho
    discoveryDetails: |+
      descriptions:
        - "foo"
        - "bar"
```

此部分定义要为每个已发现设备部署的工作负载。该示例显示了 `brokerPodSpec` 中 `Pod` 配置的最低版本，在此处可以使用 Pod 规范的所有常规字段。其中还显示了 `resources` 部分中用于请求设备的 Akri 特定语法。

您也可以使用作业来代替 Pod，方法是改用 `brokerJobSpec` 键，并在其中提供作业的规范部分。

```
  brokerSpec:
    brokerPodSpec:
      containers:
      - name: broker-container
        image: rancher/hello-world
        resources:
          requests:
            "{{PLACEHOLDER}}" : "1"
          limits:
            "{{PLACEHOLDER}}" : "1"
```

这两个部分显示如何配置 Akri 以便为每个中介程序部署一个服务 (`instanceService`)，或指向所有中介程序 (`configurationService`)。这些部分包含与常规服务相关的所有元素。

```
  instanceServiceSpec:
    type: ClusterIp
    ports:
```

```
    - name: http
      port: 80
      protocol: tcp
      targetPort: 80
  configurationServiceSpec:
    type: ClusterIp
    ports:
    - name: https
      port: 443
      protocol: tcp
      targetPort: 443
```

`brokerProperties` 字段是一个键/值存储区，它将作为附加环境变量公开给请求已发现设备的任何 Pod。

capacity 是已发现设备的并发用户的允许数量。

```
brokerProperties:
  key: value
capacity: 1
```

14.4 编写和部署更多发现处理程序

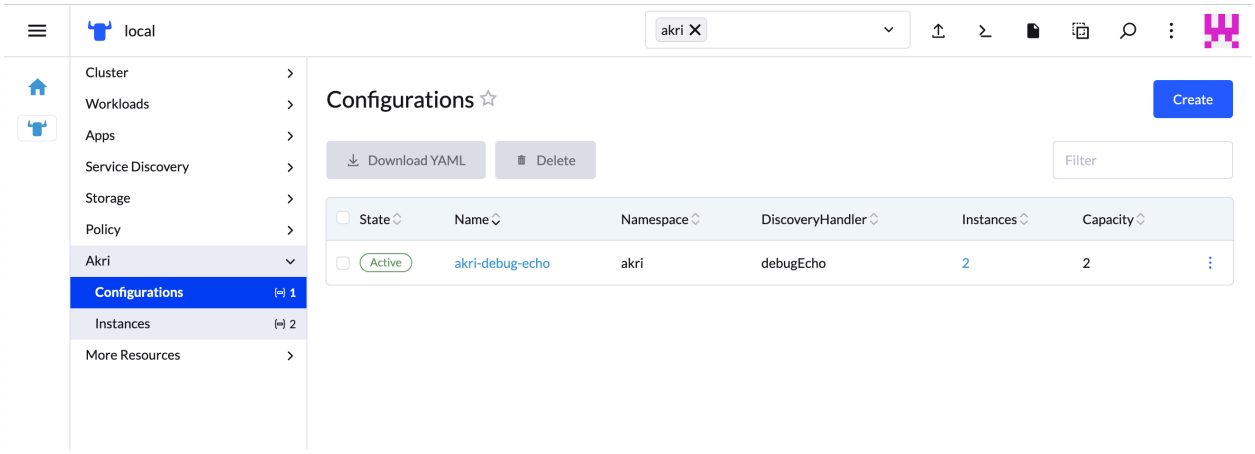
如果现有的发现处理程序无法涵盖您的设备使用的协议，您可以遵循处理程序开发指南 (https://docs.akri.sh/development/handler-development) 编写自己的发现处理程序。

14.5 Akri Rancher 仪表板扩展

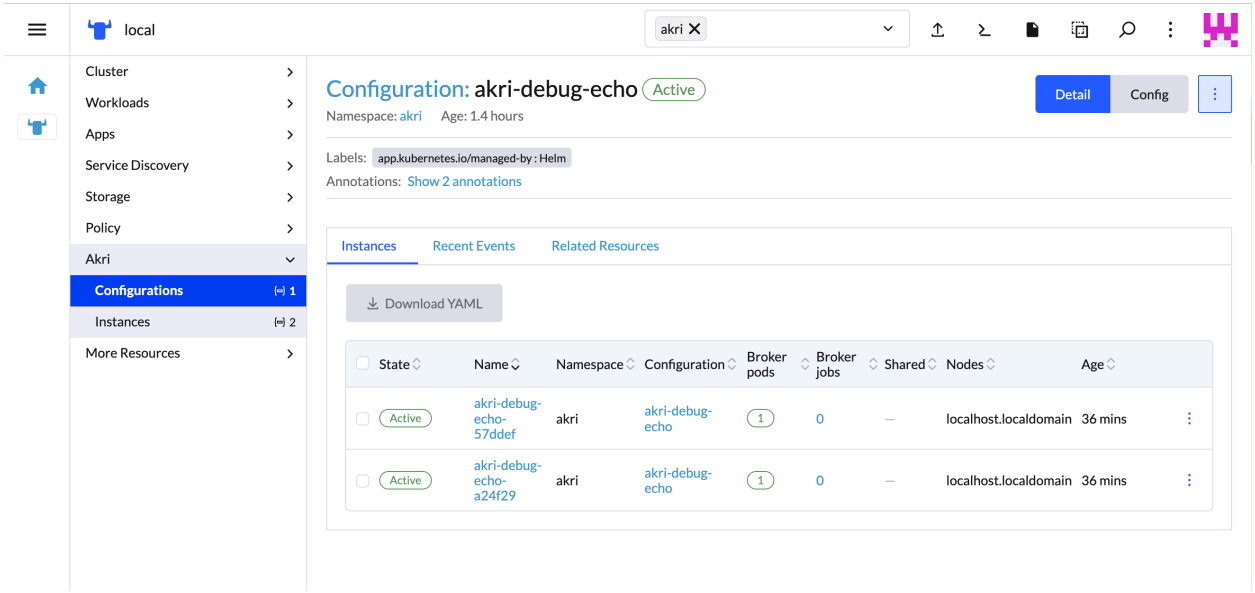
Akri 仪表板扩展允许您使用 Rancher 仪表板用户界面来管理和监控叶设备，并在发现这些设备后运行工作负载。

请参见第 6 章 “Rancher 仪表板扩展”获取安装指导。

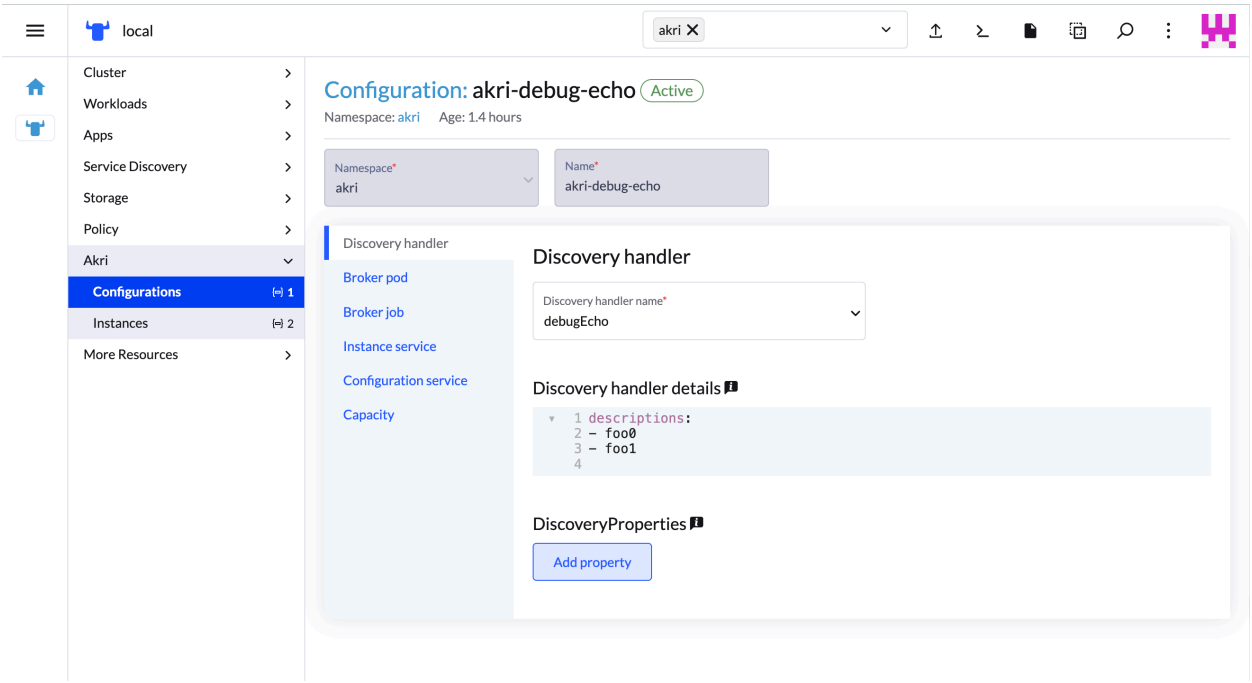
安装扩展后，您可以使用群集资源管理器导航到任何已启用 Akri 的受管群集。在 **Akri** 导航组下，可以看到 **Configurations**（配置）和 **Instances**（实例）部分。



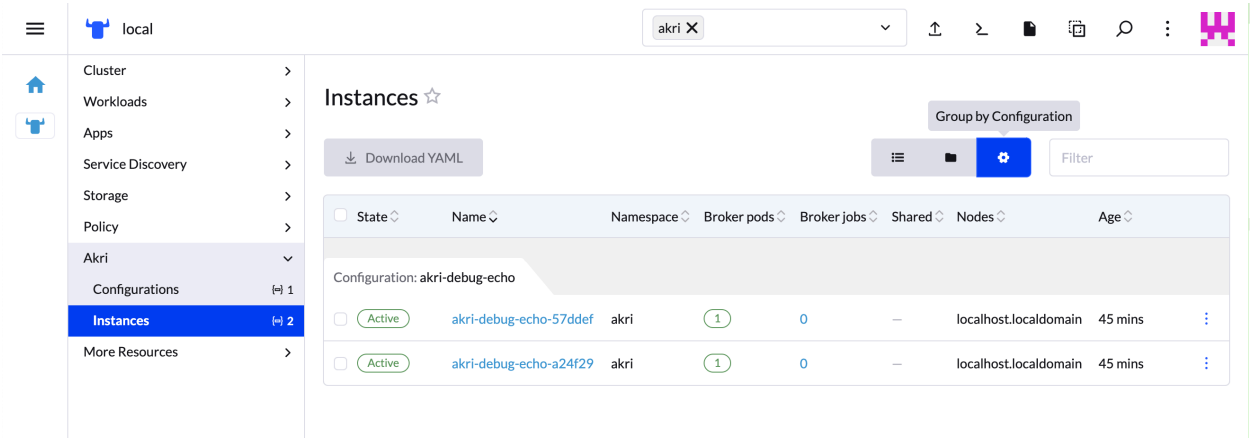
配置列表提供了有关配置发现处理程序和实例数量的信息。单击名称会打开配置细节页面。



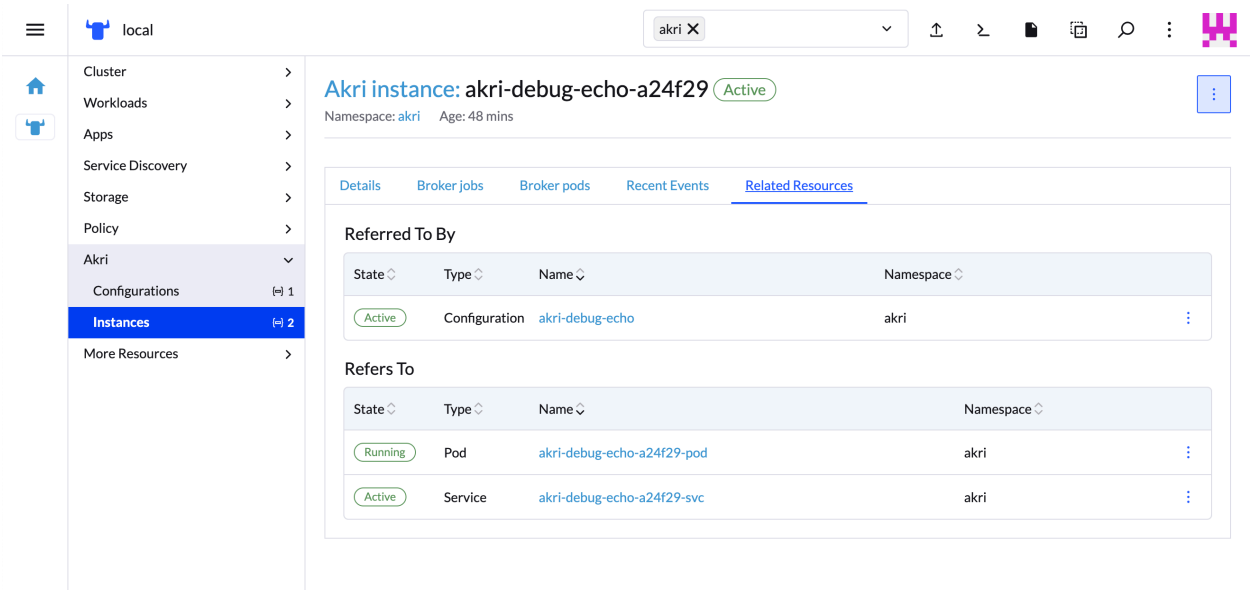
您还可以编辑或创建新的**配置**。扩展允许您选择发现处理程序、设置中介程序 Pod 或作业、自定义配置和实例服务，以及设置配置容量。



Instances（实例）列表中会列出已发现的设备。



单击**实例名称**会打开细节页面，在其中可以查看工作负载和实例服务。


local
akri
Cluster
Workloads
Apps
Service Discovery
Storage
Policy
Akri
Configurations 1
Instances 2
More Resources
Akri instance: akri-debug-echo-a24f29 Active
Namespace: akri Age: 48 mins
Details
Broker jobs
Broker pods
Recent Events
Related Resources
Referred To By
State Type Name Namespace
Active Configuration akri-debug-echo akri
Refers To
State Type Name Namespace
Running Pod akri-debug-echo-a24f29-pod akri
Active Service akri-debug-echo-a24f29-svc akri

15 K3s

K3s (https://k3s.io/) 是经过认证的高可用性 Kubernetes 发行版，专为无人照管、资源受限的远程位置或物联网设备内的生产工作负载而设计。

它封装为单个较小二进制文件，因此可以快速轻松地安装和更新。

15.1 SUSE Edge 如何使用 K3s

K3s 可用作支持 SUSE Edge 堆栈的 Kubernetes 发行版。它适合安装在 SUSE Linux Micro 操作系统上。

仅当用作后端的 etcd 不能满足您的约束条件时，才建议使用 K3s 作为 SUSE Edge 堆栈 Kubernetes 发行版。如果 etcd 可用作后端，则最好使用 RKE2（第 16 章 “RKE2”）。

15.2 最佳实践

15.2.1 安装

将 K3s 安装为 SUSE Edge 堆栈一部分的建议方法是使用 Edge Image Builder (EIB)。有关如何配置 EIB 来部署 K3s 的详细信息，请参见其文档（第 11 章 “Edge Image Builder”）。

K3s 原生支持 HA 设置以及 Elemental 设置。

15.2.2 用于 GitOps 工作流程的 Fleet

SUSE Edge 堆栈使用 Fleet 作为其首选 GitOps 工具。有关 Fleet 安装和用法的详细信息，请参见本文档中的第 8 章 “Fleet”。

15.2.3 存储管理

K3s 预配置了本地路径存储服务，这种存储服务适用于单节点群集。对于跨多个节点的群集，我们建议使用 SUSE Storage（第 17 章 “SUSE Storage”）。

15.2.4 负载平衡和 HA

如果您是使用 EIB 安装的 K3s，请参见 EIB 文档中的“HA”一章，其中已介绍本节所述的内容。

否则，您需要按照 MetalLB 文档（第 25 章 “K3s 上的 MetalLB（使用第 2 层模式）”）安装和配置 MetalLB。

16 RKE2

请参见 RKE2 官方文档 (https://docs.rke2.io/)。

RKE2 是完全符合规范且注重安全性与合规性的 Kubernetes 发行版，因为它：

- 提供默认设置和配置选项，使群集能够在最低限度的运维干预下通过 CIS Kubernetes 基准 v1.6 或 v1.23
- 支持 FIPS 140-2 合规性标准
- 在 RKE2 构建管道中使用 trivy (https://trivy.dev) 定期扫描组件中存在的 CVE

RKE2 将控制平面组件作为 kubelet 管理的静态 Pod 进行启动。嵌入式容器运行时为 containerd。

注意：RKE2 也称为 RKE 政府版，这个名称旨在体现它目前面向的另一类使用场景和行业领域。

16.1 RKE2 与 K3s 的比较

K3s 是完全合规的轻量级 Kubernetes 发行版，主要用于 Edge、IoT 和 ARM，已针对易用性和资源受限的环境进行优化。

RKE2 结合了 1.x 版 RKE（后文称为 RKE1）和 K3s 的最大优点。

RKE2 承袭了 K3s 的易用性、易操作性和部署模式。

从 RKE1 继承了与上游 Kubernetes 的紧密一致性。在某些方面，K3s 为了优化边缘部署而与上游 Kubernetes 产生了差异，但 RKE1 和 RKE2 能够保持与上游的紧密一致。

16.2 SUSE Edge 如何使用 RKE2?

RKE2 是 SUSE Edge 堆栈的基本组成部分。它位于 SUSE Linux Micro（第 9 章 “SUSE Linux Micro”）的顶层，提供部署 Edge 工作负载所需的标准 Kubernetes 接口。

16.3 最佳实践

16.3.1 安装

将 RKE2 安装为 SUSE Edge 堆栈一部分的建议方法是使用 Edge Image Builder (EIB)。有关如何配置 EIB 来部署 RKE2 的详细信息，请参见 EIB 文档（第 11 章 “Edge Image Builder”）。

EIB 足够灵活，支持 RKE2 所需的任何参数（例如指定 RKE2 版本、服务器 (https://docs.rke2.io/reference/server_config) 或代理 (https://docs.rke2.io/reference/linux_agent_config) 配置），适用于所有 Edge 使用场景。

对于涉及 Metal^3 的其他使用场景，也可以使用和安装 RKE2。在这种特殊情况下，Cluster API 提供程序 RKE2 (https://github.com/rancher-sandbox/cluster-api-provider-rke2) 会自动在使用 Edge Stack 通过 Metal^3 置备的群集上部署 RKE2。

在这种情况下，必须在涉及的不同 CRD 上应用 RKE2 配置。以下示例说明如何使用 `RKE2ControlPlane` CRD 提供不同的 CNI：

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  serverConfig:
    cni: calico
    cniMultusEnable: true
...
```

有关 Metal^3 使用场景的详细信息，请参见第 10 章 “Metal^3”。

16.3.2 高可用性

对于高可用性 (HA) 部署，EIB 会自动部署并配置 MetalLB（第 19 章 “MetalLB”）和 Endpoint Copier Operator（第 20 章 “Endpoint Copier Operator”），以向外部公开 RKE2 API 端点。

16.3.3 网络

SUSE Edge 堆栈支持 Cilium (https://docs.cilium.io/en/stable/)、Calico (https://docs.tigera.io/calico/latest/about/)，并使用 Cilium 作为其默认 CNI。如果 Pod 需要多个网络接口，也可使用 Multus (https://github.com/k8snetworkplumbingwg/multus-cni) 元插件。RKE2 独立版本支持更广泛的 CNI 选项 (https://docs.rke2.io/install/network_options)。

16.3.4 存储

RKE2 不提供任何类型的持久性存储类或操作器。对于跨多个节点的群集，建议使用 SUSE Storage（第 17 章 “SUSE Storage”）。

17 SUSE Storage

SUSE Storage 是一款可靠、易用的轻量级分布式块存储系统，专为 Kubernetes 设计。它是基于 Longhorn 的一款产品，而 Longhorn 是一个开源项目，最初由 Rancher Labs 开发，目前在 CNCF 旗下孵化。

17.1 先决条件

如果您要学习本指南，事先需要做好以下准备：

- 至少一台装有 SUSE Linux Micro 6.1 的主机，可以是物理主机，也可以是虚拟主机
- 已安装一个 Kubernetes 群集，可以是 K3s 或 RKE2
- Helm

17.2 手动安装 SUSE Storage

17.2.1 安装 Open-iSCSI

要部署并使用 SUSE Storage，需满足的一项核心要求是在所有 Kubernetes 节点上安装 `open-iscsi` 软件包并运行 `iscsid` 守护程序。之所以有此要求，是因为 Longhorn 依赖于主机上的 `iscsiadm` 来为 Kubernetes 提供持久卷。

我们来安装此软件包：

```
transactional-update pkg install open-iscsi
```

请务必注意，在操作完成后，该软件包只会安装到新快照中，因为 SUSE Linux Micro 是不可变的操作系统。要加载该软件包并让 `iscsid` 守护程序开始运行，我们必须重引导至刚刚创建的新快照。准备就绪后，发出 reboot 命令：

```
reboot
```

提示

如需更多有关安装 open-iscsi 的帮助，请参见官方 Longhorn 文档 (https://longhorn.io/docs/1.8.1/deploy/install/#installing-open-iscsi)。

17.2.2 安装 SUSE Storage

可通过多种方式在 Kubernetes 群集上安装 SUSE Storage。本指南将介绍如何通过 Helm 进行安装，但如果您想要采用其他方法，请按照官方文档 (https://longhorn.io/docs/1.8.1/deploy/install/) 中的说明操作。

1. 添加 Rancher Chart Helm 储存库：

```
helm repo add rancher-charts https://charts.rancher.io/
```

2. 从储存库提取最新的 chart：

```
helm repo update
```

3. 在 `longhorn-system` 名称空间中安装 SUSE Storage：

```
helm install longhorn-crd rancher-charts/longhorn-crd --namespace longhorn-
system --create-namespace --version 106.2.0+up1.8.1
helm install longhorn rancher-charts/longhorn --namespace longhorn-system
 --version 106.2.0+up1.8.1
```

4. 确认部署是否成功：

```
kubectl -n longhorn-system get pods
```

```
localhost:~ # kubectl -n longhorn-system get pod
NAMESPACE         NAME                                                READY
   STATUS      RESTARTS        AGE
longhorn-system   longhorn-ui-5fc9fb76db-z5dc9                        1/1
   Running     0               90s
longhorn-system   longhorn-ui-5fc9fb76db-dcb65                        1/1
   Running     0               90s
```

```
longhorn-system   longhorn-manager-wts2v                                1/1
   Running     1 (77s ago)     90s
longhorn-system   longhorn-driver-deployer-5d4f79ddd-fxgcs              1/1
   Running     0               90s
longhorn-system   instance-manager-a9bf65a7808a1acd6616bcd4c03d925b     1/1
   Running     0               70s
longhorn-system   engine-image-ei-acb7590c-htqmp                        1/1
   Running     0               70s
longhorn-system   csi-attacher-5c4bfdcf59-j8xww                         1/1
   Running     0               50s
longhorn-system   csi-provisioner-667796df57-l69vh                      1/1
   Running     0               50s
longhorn-system   csi-attacher-5c4bfdcf59-xgd5z                         1/1
   Running     0               50s
longhorn-system   csi-provisioner-667796df57-dqkfr                      1/1
   Running     0               50s
longhorn-system   csi-attacher-5c4bfdcf59-wckt8                         1/1
   Running     0               50s
longhorn-system   csi-resizer-694f8f5f64-7n2kq                          1/1
   Running     0               50s
longhorn-system   csi-snapshotter-959b69d4b-rp4gk                       1/1
   Running     0               50s
longhorn-system   csi-resizer-694f8f5f64-r6ljc                          1/1
   Running     0               50s
longhorn-system   csi-resizer-694f8f5f64-k7429                          1/1
   Running     0               50s
longhorn-system   csi-snapshotter-959b69d4b-5k8pg                       1/1
   Running     0               50s
longhorn-system   csi-provisioner-667796df57-n5w9s                      1/1
   Running     0               50s
longhorn-system   csi-snapshotter-959b69d4b-x7b7t                       1/1
   Running     0               50s
longhorn-system   longhorn-csi-plugin-bsc8c                             3/3
   Running     0               50s
```

17.3 创建 SUSE Storage 卷

SUSE Storage 利用名为 `StorageClass` 的 Kubernetes 资源来自动为 Pod 置备 `PersistentVolume` 对象。可以将 `StorageClass` 视为管理员描述其提供的存储**类**或**配置文件**的一种方式。

我们来创建一个采用默认选项的 `StorageClass`：

```
kubectl apply -f - <<EOF
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: longhorn-example
provisioner: driver.longhorn.io
allowVolumeExpansion: true
parameters:
  numberOfReplicas: "3"
  staleReplicaTimeout: "2880" # 48 hours in minutes
  fromBackup: ""
  fsType: "ext4"
EOF
```

创建 `StorageClass` 后，我们需要提供一个 `PersistentVolumeClaim` 来引用它。`PersistentVolumeClaim` (PVC) 是用户发出的存储请求。PVC 使用 `PersistentVolume` 资源。声明可以请求特定的大小和访问模式（例如，可以以读/写模式挂载声明一次，或以只读模式挂载声明多次）。

我们来创建 `PersistentVolumeClaim`：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: longhorn-volv-pvc
  namespace: longhorn-system
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: longhorn-example
```

```
  resources:
    requests:
      storage: 2Gi
EOF
```

大功告成！创建 PersistentVolumeClaim 后，我们可以继续将其挂接到 Pod。部署 Pod 时，如果有可用存储空间，Kubernetes 会创建 Longhorn 卷并将其绑定到 Pod。

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
  namespace: longhorn-system
spec:
  containers:
  - name: volume-test
    image: nginx:stable-alpine
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: volv
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: volv
    persistentVolumeClaim:
      claimName: longhorn-volv-pvc
EOF
```

提示

Kubernetes 中的存储概念是一个复杂又重要的主题。我们简单地提及了一些最常见的 Kubernetes 资源，不过建议您熟悉 Longhorn 提供的术语文档 (https://longhorn.io/docs/1.8.1/terminology/)。

对于此示例，结果应如下所示：

```
localhost:~ # kubectl get storageclass
NAME                 PROVISIONER          RECLAIMPOLICY   VOLUMEBINDINGMODE
 ALLOWVOLUMEEXPANSION   AGE
longhorn (default)   driver.longhorn.io   Delete          Immediate
 true                   12m
longhorn-example     driver.longhorn.io   Delete          Immediate
 true                   24s

localhost:~ # kubectl get pvc -n longhorn-system
NAME                STATUS   VOLUME                                     CAPACITY
   ACCESS MODES   STORAGECLASS       AGE
longhorn-volv-pvc   Bound    pvc-f663a92e-ac32-49ae-b8e5-8a6cc29a7d1e   2Gi
   RWO            longhorn-example   54s

localhost:~ # kubectl get pods -n longhorn-system
NAME                                                READY   STATUS    RESTARTS
    AGE
csi-attacher-5c4bfdcf59-qmjtz                       1/1     Running   0
    14m
csi-attacher-5c4bfdcf59-s7n65                       1/1     Running   0
    14m
csi-attacher-5c4bfdcf59-w9xgs                       1/1     Running   0
    14m
csi-provisioner-667796df57-fmz2d                    1/1     Running   0
    14m
csi-provisioner-667796df57-p7rjr                    1/1     Running   0
    14m
csi-provisioner-667796df57-w9fdq                    1/1     Running   0
    14m
csi-resizer-694f8f5f64-2rb8v                        1/1     Running   0
    14m
csi-resizer-694f8f5f64-z9v9x                        1/1     Running   0
    14m
csi-resizer-694f8f5f64-zlncz                        1/1     Running   0
    14m
csi-snapshotter-959b69d4b-5dpvj                     1/1     Running   0
    14m
```

```
csi-snapshotter-959b69d4b-lwwkv                    1/1     Running   0
    14m
csi-snapshotter-959b69d4b-tzhwc                    1/1     Running   0
    14m
engine-image-ei-5cefaf2b-hvdv5                     1/1     Running   0
    14m
instance-manager-0ee452a2e9583753e35ad00602250c5b  1/1     Running   0
    14m
longhorn-csi-plugin-gd2jx                          3/3     Running   0
    14m
longhorn-driver-deployer-9f4fc86-j6h2b             1/1     Running   0
    15m
longhorn-manager-z4lnl                             1/1     Running   0
    15m
longhorn-ui-5f4b7bbf69-bln7h                       1/1     Running   3 (14m
 ago)   15m
longhorn-ui-5f4b7bbf69-lh97n                       1/1     Running   3 (14m
 ago)   15m
volume-test                                        1/1     Running   0
    26s
```

17.4 访问 UI

如果您使用 kubectl 或 Helm 安装了 Longhorn，则需要设置入口控制器，使外部流量能够进入群集。默认不会启用身份验证。如果使用了 Rancher 目录应用程序，Rancher 已自动创建一个提供访问控制的入口控制器 (rancher-proxy)。

1. 获取 Longhorn 的外部服务 IP 地址：

   ```
   kubectl -n longhorn-system get svc
   ```

2. 检索到 `longhorn-frontend` IP 地址后，您可以通过在浏览器中导航到该前端来开始使用 UI。

17.5 使用 Edge Image Builder 进行安装

SUSE Edge 使用第 11 章 “Edge Image Builder”来自定义基础 SUSE Linux Micro 操作系统映像。本节将展示如何执行自定义操作，以置备 RKE2 群集并在其上安装 Longhorn。

我们来创建定义文件：

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR

cat << EOF > $CONFIG_DIR/iso-definition.yaml
apiVersion: 1.2
image:
  imageType: iso
  baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
  arch: x86_64
  outputImageName: eib-image.iso
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: longhorn
        version: 106.2.0+up1.8.1
        repositoryName: longhorn
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn-crd
        version: 106.2.0+up1.8.1
        repositoryName: longhorn
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
    repositories:
      - name: longhorn
        url: https://charts.rancher.io
operatingSystem:
  packages:
    sccRegistrationCode: <reg-code>
```

```
    packageList:
      - open-iscsi
  users:
  - username: root
    encryptedPassword: \$6\$jHugJNNd3HElGsUZ\
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
EOF
```

注意

可以通过 `helm.charts[].valuesFile` 下提供的独立文件自定义任何 Helm chart 值。有关详细信息，请参见上游文档 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md#kubernetes)。

我们来构建映像：

```
podman run --rm --privileged -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.3/
edge-image-builder:1.2.1 build --definition-file $CONFIG_DIR/iso-definition.yaml
```

构建映像后，可以使用它在物理主机或虚拟主机上安装操作系统。置备完成后，可以使用 `root:eib` 身份凭证对登录到系统。

确保 Longhorn 已成功部署：

```
localhost:~ # /var/lib/rancher/rke2/bin/kubectl --kubeconfig /etc/rancher/rke2/
rke2.yaml -n longhorn-system get pods
NAME                                                READY   STATUS    RESTARTS
      AGE
csi-attacher-5c4bfdcf59-qmjtz                       1/1     Running   0
      103s
csi-attacher-5c4bfdcf59-s7n65                       1/1     Running   0
      103s
csi-attacher-5c4bfdcf59-w9xgs                       1/1     Running   0
      103s
csi-provisioner-667796df57-fmz2d                    1/1     Running   0
      103s
csi-provisioner-667796df57-p7rjr                    1/1     Running   0
      103s
```

```
csi-provisioner-667796df57-w9fdq                    1/1     Running   0
      103s
csi-resizer-694f8f5f64-2rb8v                        1/1     Running   0
      103s
csi-resizer-694f8f5f64-z9v9x                        1/1     Running   0
      103s
csi-resizer-694f8f5f64-zlncz                        1/1     Running   0
      103s
csi-snapshotter-959b69d4b-5dpvj                     1/1     Running   0
      103s
csi-snapshotter-959b69d4b-lwwkv                     1/1     Running   0
      103s
csi-snapshotter-959b69d4b-tzhwc                     1/1     Running   0
      103s
engine-image-ei-5cefaf2b-hvdv5                      1/1     Running   0
      109s
instance-manager-0ee452a2e9583753e35ad00602250c5b   1/1     Running   0
      109s
longhorn-csi-plugin-gd2jx                           3/3     Running   0
      103s
longhorn-driver-deployer-9f4fc86-j6h2b              1/1     Running   0
      2m28s
longhorn-manager-z4lnl                              1/1     Running   0
      2m28s
longhorn-ui-5f4b7bbf69-bln7h                        1/1     Running   3 (2m7s
 ago)    2m28s
longhorn-ui-5f4b7bbf69-lh97n                        1/1     Running   3 (2m10s
 ago)   2m28s
```

注意

此安装不适用于完全隔离的环境。对于这种情况，请参见第 27.8 节 “SUSE Storage 安装”。

18 SUSE Security

SUSE Security 是适用于 Kubernetes 的安全解决方案，它在统一的软件包中提供 L7 网络安全性、运行时安全性、供应链安全性与合规性检查。

SUSE Security 是一款作为多容器平台部署的产品，各个容器通过不同端口和接口进行通讯。其底层采用 NeuVector 作为容器安全组件。构成 SUSE Security 平台的容器如下：

- 管理器：提供基于 Web 的控制台的无状态容器。通常只需一个，可在任何位置运行。管理器发生故障不会影响控制器或执行器的任何操作。但是，某些通知（事件）和最近的连接数据将由管理器缓存在内存中，因此这些信息的查看操作会受到影响。
- 控制器：SUSE Security 的“控制平面”必须部署在 HA 配置中，这样，配置就不会在节点发生故障时丢失。这些容器可在任何位置运行，不过，由于它们的重要性，客户通常会将它们放在“管理”节点、主节点或基础架构节点上。
- 执行器：此容器部署为 DaemonSet，因此每个要保护的节点上都有一个执行器。通常会部署到每个工作节点，但可为主节点和基础架构节点启用调度，以便将这些容器同时部署到这些节点。注意：如果执行器不在群集节点上，并且连接来自该节点上的 Pod，则 SUSE Security 会将这些容器标记为“不受管”工作负载。
- 扫描器：根据控制器的指示，使用内置 CVE 数据库执行漏洞扫描。可以部署多个扫描器来提高扫描能力。扫描器可在任何位置运行，但通常在运行控制器的节点上运行。请参见下文了解扫描器节点的大小调整注意事项。在用于构建阶段的扫描时，还可以独立调用扫描器，例如，在触发扫描、检索结果和停止扫描器的管道中。扫描器包含最新的 CVE 数据库，因此应每日更新。
- 更新器：需要更新 CVE 数据库时，更新器会通过 Kubernetes cron 作业触发扫描器的更新。请务必根据您的环境配置此设置。

在此处 (https://open-docs.neuvector.com/) 可以找到更深入的 SUSE Security 初始配置信息和最佳实践文档。

18.1 SUSE Edge 如何使用 SUSE Security?

SUSE Edge 提供了更为精简的 SUSE Security 配置，方便您着手进行边缘部署。

18.2 重要注意事项

- 扫描器容器必须有充足的内存，以便能够将要扫描的映像提取到内存并对其进行扩展。要扫描 1 GB 以上的映像，请将扫描器的内存增加至略高于最大预期映像大小。
- 在保护模式下需要高速网络连接。处于保护（内联防火墙阻止）模式的执行器需要占用 CPU 和内存来保持和检查连接以及可能的有效负载 (DLP)。增加内存并专门分配一个 CPU 核心供执行器使用可确保拥有充足的数据包过滤能力。

18.3 使用 Edge Image Builder 进行安装

SUSE Edge 使用第 11 章 “Edge Image Builder” 来自定义基础 SUSE Linux Micro 操作系统映像。请按照第 27.7 节 “SUSE Security 安装” 中所述，在 EIB 置备的 Kubernetes 群集上进行 SUSE Security 隔离式安装。

19 MetalLB

请参见 MetalLB 官方文档 (https://metallb.universe.tf/)。

MetalLB 是使用标准路由协议的裸机 Kubernetes 群集的负载平衡器实现。

在裸机环境中，设置网络负载平衡器比在云环境中要复杂得多。与云设置中的直接 API 调用不同，裸机需要通过专用网络设备或者负载平衡器和虚拟 IP (VIP) 配置的组合，来管理高可用性 (HA) 或解决单节点负载平衡器固有的潜在单一故障点 (SPOF)。这些配置不容易实现自动化，在组件会动态扩缩的 Kubernetes 部署中带来了挑战。

MetalLB 可以解决这些挑战，因为它利用 Kubernetes 模型创建 LoadBalancer 类型的服务，就如同这些服务是在云环境中运行一样，即使在裸机设置中，也能做到这一点。

为此可以采用两种不同的方法：通过 L2 模式 (https://metallb.universe.tf/concepts/layer2/)（使用 ARP **技巧**）或通过 BGP (https://metallb.universe.tf/concepts/bgp/)。大体而言，L2 不需要任何特殊网络设备，但 BGP 通常效果更好。使用哪种方法取决于使用场景。

19.1 SUSE Edge 如何使用 MetalLB?

SUSE Edge 主要通过两种方式使用 MetalLB：

- 作为负载平衡器解决方案：MetalLB 充当裸机的负载平衡器解决方案。
- 对于 HA K3s/RKE2 设置：MetalLB 允许使用虚拟 IP 地址对 Kubernetes API 进行负载平衡。

注意

为了能够公开 API，会使用 Endpoint Copier Operator（第 20 章 “Endpoint Copier Operator”）将 K8s API 端点从 `kubernetes` 服务同步到 `kubernetes-vip` LoadBalancer 服务。

19.2 最佳实践

第 25 章 “K3s 上的 MetalLB（使用第 2 层模式）” 中详细介绍了 L2 模式的 MetalLB 安装。有关在 `kube-api-server` 前端安装 MetalLB 以实现高可用性拓扑的指南，可参见第 26 章 “Kubernetes API 服务器前面的 MetalLB”。

19.3 已知问题

- K3s 附带负载平衡器解决方案 `Klipper`。要使用 MetalLB，必须禁用 `Klipper`。为此，可以按照 K3s 文档 (https://docs.k3s.io/networking) 中所述，使用 `--disable servicelb` 选项启动 K3s 服务器。

20 Endpoint Copier Operator

Endpoint Copier Operator (https://github.com/suse-edge/endpoint-copier-operator) 是一个 Kubernetes 操作器，其用途是创建 Kubernetes 服务和端点的副本，并将它们保持同步。

20.1 SUSE Edge 如何使用 Endpoint Copier Operator?

在 SUSE Edge 中，Endpoint Copier Operator 在实现 K3s/RKE2 群集的高可用性 (HA) 设置方面发挥着关键作用。此目标通过创建一个类型为 `LoadBalancer` 的 `kubernetes-vip` 服务并确保其端点与 kubernetes 端点始终保持同步来实现。系统会使用 MetalLB（第 19 章 “MetalLB”）来管理 `kubernetes-vip` 服务，因为其他节点会使用其公开 IP 地址加入群集。

20.2 最佳实践

有关使用 Endpoint Copier Operator 的完整文档，请参见此处 (https://github.com/suse-edge/endpoint-copier-operator/blob/main/README.md)。

此外，可参考我们的指南（第 25 章 “K3s 上的 MetalLB（使用第 2 层模式）”），了解如何使用 Endpoint Copier Operator 和 MetalLB 实现 K3s/RKE2 高可用性设置。

20.3 已知问题

目前，Endpoint Copier Operator 仅能用于单个服务/端点。未来计划增强其功能，以支持多个服务/端点。

21 Edge Virtualization

本节介绍如何使用 Edge Virtualization 在边缘节点上运行虚拟机。Edge Virtualization 为轻量级虚拟化使用场景而设计，这些使用场景预期会使用一个通用工作流程来部署和管理虚拟化和容器化的应用程序。

SUSE Edge Virtualization 支持两种虚拟机运行方法：

1. 在主机级别通过 libvirt+qemu-kvm 手动部署虚拟机（不涉及 Kubernetes）
2. 部署 KubeVirt 操作器来实现基于 Kubernetes 的虚拟机管理

这两种方法都有效，但下面仅介绍第二种方法。如果您要使用 SUSE Linux Micro 提供的现成可用的标准虚拟化机制，可在此处 (https://documentation.suse.com/sles/15-SP6/html/SLES-all/chap-virtualization-introduction.html) 找到详细的指南，尽管该指南主要是针对 SUSE Linux Enterprise Server 编写的，但概念几乎相同。

本指南首先介绍如何将其他虚拟化组件部署到已预先部署的系统，然后会在一个章节中介绍如何通过 Edge Image Builder 将此配置嵌入到初始部署中。如果您不想了解基础知识并想要手动完成设置，请直接跳到该章节。

21.1 KubeVirt 概述

KubeVirt 让您可以通过 Kubernetes 管理虚拟机及其他容器化工作负载。它通过在容器中运行 Linux 虚拟化堆栈的用户空间部分来实现此目的。这样可以最大程度地降低对主机系统的要求，从而简化设置和管理。

有关 KubeVirt 体系结构的详细信息，请参见上游文档 (https://kubevirt.io/user-guide/architecture/) 。

21.2 先决条件

如果您要学习本指南，事先需要做好以下准备：

- 至少有一台装有 SUSE Linux Micro 6.1 的物理主机，并且在 BIOS 中启用了虚拟化扩展（有关详细信息，请参见此处 (https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-virt-support.html#sec-kvm-requires-hardware)）。
- 已在您的节点中部署了 K3s/RKE2 Kubernetes 群集，并提供了相应的 `kubeconfig`，使超级用户能够访问该群集。
- root 用户访问权限 — 本章中的说明假设您是 root 用户，而**未**通过 `sudo` 提升特权。
- 已在本地安装 Helm (https://helm.sh/docs/intro/install/) 并建立了速度够快的网络连接，以便可以将配置推送到 Kubernetes 群集和下载所需的映像。

21.3 手动安装 Edge Virtualization

本指南不会指导您完成 Kubernetes 的部署过程，而是假设您已安装适用于 SUSE Edge 的 K3s (https://k3s.io/) 或 RKE2 (https://docs.rke2.io/install/quickstart) 版本，并已相应地配置了 kubeconfig，以便能够以超级用户身份执行标准的 `kubectl` 命令。假设您的节点构成了单节点群集，不过，此过程与多节点部署预期不会有太大的差异。

具体而言，将通过以下三个独立的 Helm chart 来部署 SUSE Edge Virtualization：

- **KubeVirt**：核心虚拟化组件，即 Kubernetes CRD、操作器，以及使 Kubernetes 能够部署和管理虚拟机的其他组件。
- **KubeVirt 仪表板扩展**：可选的 Rancher UI 扩展，用于实现基本的虚拟机管理，例如启动/停止虚拟机以及访问控制台。
- **Containerized Data Importer (CDI)**：一个附加组件，可为 KubeVirt 实现持久性存储集成，使虚拟机能够使用现有 Kubernetes 存储后端来存储数据，同时使用户能够导入或克隆虚拟机的数据卷。

其中的每个 Helm chart 将根据您当前使用的 SUSE Edge 版本进行版本控制。对于生产/支持用途，请采用 SUSE 注册表中提供的制品。

首先，请确保可以正常进行 `kubectl` 访问：

```
$ kubectl get nodes
```

此命令应会显示如下所示的输出：

```
NAME                   STATUS   ROLES                       AGE     VERSION
node1.edge.rdo.wales   Ready    control-plane,etcd,master   4h20m
 v1.30.5+rke2r1
node2.edge.rdo.wales   Ready    control-plane,etcd,master   4h15m
 v1.30.5+rke2r1
node3.edge.rdo.wales   Ready    control-plane,etcd,master   4h15m
 v1.30.5+rke2r1
```

现在您可以继续安装 **KubeVirt** 和 **Containerized Data Importer (CDI)** Helm chart：

```
$ helm install kubevirt oci://registry.suse.com/edge/charts/kubevirt --namespace
 kubevirt-system --create-namespace
$ helm install cdi oci://registry.suse.com/edge/charts/cdi --namespace cdi-
system --create-namespace
```

几分钟后，所有 KubeVirt 和 CDI 组件应会部署完成。您可以通过检查 `kubevirt-system` 和 `cdi-system` 名称空间中部署的所有资源进行验证。

校验 KubeVirt 资源：

```
$ kubectl get all -n kubevirt-system
```

此命令应会显示如下所示的输出：

```
NAME                                 READY   STATUS    RESTARTS      AGE
pod/virt-operator-5fbcf48d58-p7xpm   1/1     Running   0             2m24s
pod/virt-operator-5fbcf48d58-wnf6s   1/1     Running   0             2m24s
pod/virt-handler-t594x               1/1     Running   0             93s
pod/virt-controller-5f84c69884-cwjvd 1/1     Running   1 (64s ago)   93s
pod/virt-controller-5f84c69884-xxw6q 1/1     Running   1 (64s ago)   93s
pod/virt-api-7dfc54cf95-v8kcl        1/1     Running   1 (59s ago)   118s

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP
 PORT(S)   AGE
service/kubevirt-prometheus-metrics   ClusterIP   None            <none>
 443/TCP   2m1s
service/virt-api                    ClusterIP   10.43.56.140    <none>
 443/TCP   2m1s
service/kubevirt-operator-webhook     ClusterIP   10.43.201.121   <none>
 443/TCP   2m1s
```

```
service/virt-exportproxy              ClusterIP   10.43.83.23     <none>
 443/TCP   2m1s

NAME                           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
   NODE SELECTOR            AGE
daemonset.apps/virt-handler   1         1         1       1            1
   kubernetes.io/os=linux   93s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/virt-operator       2/2     2            2           2m24s
deployment.apps/virt-controller     2/2     2            2           93s
deployment.apps/virt-api            1/1     1            1           118s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/virt-operator-5fbcf48d58      2         2         2       2m24s
replicaset.apps/virt-controller-5f84c69884    2         2         2       93s
replicaset.apps/virt-api-7dfc54cf95           1         1         1       118s

NAME                              AGE     PHASE
kubevirt.kubevirt.io/kubevirt     2m24s   Deployed
```

校验 CDI 资源：

```
$ kubectl get all -n cdi-system
```

此命令应会显示如下所示的输出：

```
NAME                                    READY   STATUS    RESTARTS   AGE
pod/cdi-operator-55c74f4b86-692xb       1/1     Running   0          2m24s
pod/cdi-apiserver-db465b888-62lvr       1/1     Running   0          2m21s
pod/cdi-deployment-56c7d74995-mgkfn     1/1     Running   0          2m21s
pod/cdi-uploadproxy-7d7b94b968-6kxc2    1/1     Running   0          2m22s

NAME                              TYPE        CLUSTER-IP     EXTERNAL-IP
 PORT(S)    AGE
service/cdi-uploadproxy           ClusterIP   10.43.117.7    <none>        443/
TCP    2m22s
service/cdi-api                   ClusterIP   10.43.20.101   <none>        443/
TCP    2m22s
```

```
service/cdi-prometheus-metrics   ClusterIP   10.43.39.153   <none>        8080/
TCP   2m21s

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cdi-operator      1/1     1            1           2m24s
deployment.apps/cdi-apiserver     1/1     1            1           2m22s
deployment.apps/cdi-deployment    1/1     1            1           2m21s
deployment.apps/cdi-uploadproxy   1/1     1            1           2m22s

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/cdi-operator-55c74f4b86      1         1         1       2m24s
replicaset.apps/cdi-apiserver-db465b888      1         1         1       2m21s
replicaset.apps/cdi-deployment-56c7d74995    1         1         1       2m21s
replicaset.apps/cdi-uploadproxy-7d7b94b968   1         1         1       2m22s
```

要校验是否已部署 `VirtualMachine` 自定义资源定义 (CRD)，请使用以下命令：

```
$ kubectl explain virtualmachine
```

此命令应会列显 `VirtualMachine` 对象的定义，如下所示：

```
GROUP:      kubevirt.io
KIND:       VirtualMachine
VERSION:    v1

DESCRIPTION:
    VirtualMachine handles the VirtualMachines that are not running or are in a
    stopped state The VirtualMachine contains the template to create the
    VirtualMachineInstance. It also mirrors the running state of the created
    VirtualMachineInstance in its status.
(snip)
```

21.4 部署虚拟机

部署 KubeVirt 和 CDI 后，我们需要基于 openSUSE Tumbleweed (https://get.opensuse.org/tumbleweed/) 定义一个简单的虚拟机。此虚拟机采用最简单的配置，与任何其他 Pod 一样使用标准的“Pod 网络”进行网络配置。与任何没有 PVC (https://kubernetes.io/docs/concepts/storage/persistent-volumes/) 的容器一样，它也采用非持久性存储空间，因此可确保存储空间是临时性的。

```
$ kubectl apply -f - <<EOF
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: tumbleweed
  namespace: default
spec:
  runStrategy: Always
  template:
    spec:
      domain:
        devices: {}
        machine:
          type: q35
        memory:
          guest: 2Gi
        resources: {}
      volumes:
      - containerDisk:
          image: registry.opensuse.org/home/roxenham/tumbleweed-container-disk/
containerfile/cloud-image:latest
        name: tumbleweed-containerdisk-0
      - cloudInitNoCloud:
          userDataBase64:
 I2Nsb3VkLWNvbmZpZwpkaXNhYmxlX3Jvb3Q6IGZhbHNlCnNzaF9wd2F1dGg6IFRydWUKdXNlcnM6CiAgLSBkZWZhdWx
        name: cloudinitdisk
EOF
```

此命令列显的输出应会指出已创建 `VirtualMachine`：

```
virtualmachine.kubevirt.io/tumbleweed created
```

此 `VirtualMachine` 定义极其简洁，几乎未指定配置信息。它只是概括性地指明，此计算机的类型为“q35 (https://wiki.qemu.org/Features/Q35)”，具有 2 GB 内存，使用基于临时 `containerDisk` 的磁盘映像（即，存储在远程映像储存库中某个容器映像内的磁盘映像）。此定义还指定了一个 base64 编码的 cloudInit 磁盘，该磁盘仅用于在引导时创建用户和执行口令（可使用 `base64-d` 对其进行解码）。

注意

此虚拟机映像仅用于测试。该映像不受官方支持，仅用作文档示例。

此虚拟机需要几分钟时间才能完成引导，因为它需要下载 openSUSE Tumbleweed 磁盘映像，但一旦完成此过程，您可以通过检查虚拟机信息来查看有关该虚拟机的更多细节：

```
$ kubectl get vmi
```

此命令应会列显启动虚拟机的节点以及虚拟机的 IP 地址。请记住，由于它使用 Pod 网络，因此报告的 IP 地址与任何其他 Pod 一样并且可路由：

```
NAME          AGE     PHASE     IP            NODENAME                READY
tumbleweed    4m24s   Running   10.42.2.98    node3.edge.rdo.wales    True
```

在使用 CNI（例如 Cilium）将流量直接路由到 Pod 的情况下，在 Kubernetes 群集节点本身上运行这些命令时，您应该可以通过 `ssh` 直接连接到该虚拟机本身。请将下面的 IP 地址替换为分配给您的虚拟机的 IP 地址：

```
$ ssh suse@10.42.2.98
(password is "suse")
```

进入此虚拟机后，可对其进行任意操作，但请记住，它的资源有限，磁盘空间只有 1 GB。完成后，请按 `Ctrl-D` 或输入 `exit` 与 SSH 会话断开连接。

虚拟机进程仍封装在标准 Kubernetes Pod 中。`VirtualMachine` CRD 代表期望的虚拟机，但与任何其他应用程序一样，实际启动虚拟机的过程是通过 `virt-launcher` Pod（标准 Kubernetes Pod）进行的。启动的每个虚拟机都有一个对应的 `virt-launcher` Pod：

```
$ kubectl get pods
```

此命令应会针对我们定义的 Tumbleweed 虚拟机显示一个 `virt-launcher` Pod：

```
NAME                              READY   STATUS    RESTARTS   AGE
virt-launcher-tumbleweed-8gcn4   3/3     Running   0          10m
```

如果深入查看这个 `virt-launcher` Pod，您会看到它正在执行 `libvirt` 和 `qemu-kvm` 进程。我们可以进入该 Pod 本身并查看其内部工作。请注意您需要根据自己的 Pod 名称修改以下命令：

```
$ kubectl exec -it virt-launcher-tumbleweed-8gcn4 -- bash
```

进入 Pod 后，尝试运行 `virsh` 命令并查看进程。您会看到 `qemu-system-x86_64` 二进制文件正在运行，还会看到用于监控虚拟机的某些进程，以及磁盘映像的位置及网络（作为 tap 设备）的插接方式：

```
qemu@tumbleweed:/> ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ssl    0:00 /usr/bin/virt-launcher-monitor --qemu-timeout 269s --
name tumbleweed --uid b9655c11-38f7-4fa8-8f5d-bfe987dab42c --namespace default
 --kubevirt-share-dir /var/run/kubevirt --ephemeral-disk-dir /var/run/kubevirt-
ephemeral-disks --container-disk-dir /var/run/kube
   12 ?        Sl     0:01 /usr/bin/virt-launcher --qemu-timeout 269s --name
 tumbleweed --uid b9655c11-38f7-4fa8-8f5d-bfe987dab42c --namespace default --
kubevirt-share-dir /var/run/kubevirt --ephemeral-disk-dir /var/run/kubevirt-
ephemeral-disks --container-disk-dir /var/run/kubevirt/con
   24 ?        Sl     0:00 /usr/sbin/virtlogd -f /etc/libvirt/virtlogd.conf
   25 ?        Sl     0:01 /usr/sbin/virtqemud -f /var/run/libvirt/
virtqemud.conf
   83 ?        Sl     0:31 /usr/bin/qemu-system-x86_64 -name
 guest=default_tumbleweed,debug-threads=on -S -object {"qom-
type":"secret","id":"masterKey0","format":"raw","file":"/var/run/kubevirt-
private/libvirt/qemu/lib/domain-1-default_tumbleweed/master-key.aes"} -machine
 pc-q35-7.1,usb
  286 pts/0    Ss     0:00 bash
  320 pts/0    R+     0:00 ps ax

qemu@tumbleweed:/> virsh list --all
 Id   Name                 State
```

```
-----------------------------------
 1    default_tumbleweed   running

qemu@tumbleweed:/> virsh domblklist 1
 Target   Source
-------------------------------------------------------------------------------------------------
 sda      /var/run/kubevirt-ephemeral-disks/disk-data/tumbleweed-
containerdisk-0/disk.qcow2
 sdb      /var/run/kubevirt-ephemeral-disks/cloud-init-data/default/tumbleweed/
noCloud.iso

qemu@tumbleweed:/> virsh domiflist 1
 Interface   Type       Source   Model                    MAC
-------------------------------------------------------------------------------
 tap0        ethernet   -        virtio-non-transitional   e6:e9:1a:05:c0:92

qemu@tumbleweed:/> exit
exit
```

最后，我们需要删除此虚拟机以清理资源：

```
$ kubectl delete vm/tumbleweed
virtualmachine.kubevirt.io "tumbleweed" deleted
```

21.5 使用 virtctl

除了标准的 Kubernetes 命令行工具（即 `kubectl`）之外，KubeVirt 还附带了一个配套的命令行实用程序，它能让你与群集进行交互，从而填补虚拟化领域与 Kubernetes 设计初衷所面向的领域之间的一些空白。例如，`virtctl` 工具提供了管理虚拟机生命周期（启动、停止、重启等）、访问虚拟控制台、上载虚拟机映像以及与 Kubernetes 结构（如服务）交互的功能，且无需直接使用 API 或 CRD。

我们来下载最新的稳定版 `virtctl` 工具：

```
$ export VERSION=v1.4.0
$ wget https://github.com/kubevirt/kubevirt/releases/download/$VERSION/virtctl-
$VERSION-linux-amd64
```

如果您使用的是其他体系结构或非 Linux 计算机，可在此处 (https://github.com/kubevirt/kubevirt/releases) 找到其他版本。需要先将其转换为可执行文件才能继续，将其移动到 $PATH 中的某个位置可能会有帮助：

```
$ mv virtctl-$VERSION-linux-amd64 /usr/local/bin/virtctl
$ chmod a+x /usr/local/bin/virtctl
```

然后，可以使用 `virtctl` 命令行工具创建虚拟机。我们来复制前面创建的虚拟机，请注意我们会通过管道将输出直接传入 `kubectl apply`：

```
$ virtctl create vm --name virtctl-example --memory=1Gi \
    --volume-containerdisk=src:registry.opensuse.org/home/roxenham/tumbleweed-
container-disk/containerfile/cloud-image:latest \
    --cloud-init-user-data
 "I2Nsb3VkLWNvbmZpZwpkaXNhYmxlX3Jvb3Q6IGZhbHNlCnNzaF9wd2F1dGg6IFRydWUKdXNlcnM6CiAgLSBkZWZhdW
 | kubectl apply -f -
```

此命令应会显示虚拟机正在运行（由于容器映像将被缓存，因此这一次虚拟机的启动速度更快一些）：

```
$ kubectl get vmi
NAME              AGE   PHASE     IP            NODENAME               READY
virtctl-example   52s   Running   10.42.2.29    node3.edge.rdo.wales   True
```

现在我们可以使用 `virtctl` 直接连接到该虚拟机：

```
$ virtctl ssh suse@virtctl-example
(password is "suse" - Ctrl-D to exit)
```

`virtctl` 还可以使用其他许多命令。例如，如果网络出现故障，您可以使用 `virtctl console` 访问串行控制台；可以使用 `virtctl guestosinfo` 获取详细的操作系统信息，前提是已在 Guest 上安装并运行 `qemu-guest-agent`。

最后，我们来暂停再恢复该虚拟机：

```
$ virtctl pause vm virtctl-example
VMI virtctl-example was scheduled to pause
```

您会发现，`VirtualMachine` 对象显示为 **Paused**，而 `VirtualMachineInstance` 对象则显示为 **Running**，但同时显示了 **READY=False**：

```
$ kubectl get vm
NAME              AGE     STATUS    READY
virtctl-example   8m14s   Paused    False

$ kubectl get vmi
NAME              AGE     PHASE     IP            NODENAME               READY
virtctl-example   8m15s   Running   10.42.2.29    node3.edge.rdo.wales   False
```

您还会发现不再可以连接到该虚拟机：

```
$ virtctl ssh suse@virtctl-example
can't access VMI virtctl-example: Operation cannot be fulfilled on
 virtualmachineinstance.kubevirt.io "virtctl-example": VMI is paused
```

我们来恢复该虚拟机并重试：

```
$ virtctl unpause vm virtctl-example
VMI virtctl-example was scheduled to unpause
```

现在我们应该可以重新建立连接：

```
$ virtctl ssh suse@virtctl-example
suse@vmi/virtctl-example.default's password:
suse@virtctl-example:~> exit
logout
```

最后，我们将该虚拟机去除：

```
$ kubectl delete vm/virtctl-example
virtualmachine.kubevirt.io "virtctl-example" deleted
```

21.6 简单入口网络

本节介绍如何将虚拟机公开为标准 Kubernetes 服务，并通过 Kubernetes 入口服务（例如 RKE2 中的 NGINX (https://docs.rke2.io/networking/networking_services#nginx-ingress-controller) 或 K3s 中的 Traefik (https://docs.k3s.io/networking/networking-services#traefik-ingress-controller)）来提供这些虚拟机。本文档假设已正确配置这些组件，并且有一个适当的 DNS 指针指向 Kubernetes 服务器节点或入口虚拟 IP（例如通过通配符来指向），以正确解析入口。

注意

在 SUSE Edge 3.1+ 中，如果您在多服务器节点配置中使用 K3s，则可能需要为入口配置基于 MetalLB 的 VIP；对于 RKE2 则不需要这样做。

在示例环境中，部署了另一个 openSUSE Tumbleweed 虚拟机，cloud-init 用于在引导时将 NGINX 安装为简单 Web 服务器，并且配置了一条简单的返回消息，用于在调用时验证其是否按预期工作。要了解如何执行此操作，只需对以下输出中的 cloud-init 部分运行 `base64 -d` 即可。

现在我们来创建此虚拟机：

```
$ kubectl apply -f - <<EOF
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: ingress-example
  namespace: default
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        app: nginx
    spec:
      domain:
        devices: {}
        machine:
          type: q35
        memory:
          guest: 2Gi
        resources: {}
      volumes:
      - containerDisk:
          image: registry.opensuse.org/home/roxenham/tumbleweed-container-disk/
containerfile/cloud-image:latest
        name: tumbleweed-containerdisk-0
      - cloudInitNoCloud:
```

```
          userDataBase64:
 I2Nsb3VkLWNvbmZpZwpkaXNhYmxlX3Jvb3Q6IGZhbHNlCnNzaF9wd2F1dGg6IFRydWUKdXNlcnM6CiAgLSBkZWZhdWx
        name: cloudinitdisk
EOF
```

此虚拟机成功启动后，我们可以使用 `virtctl` 命令公开 `VirtualMachineInstance`，其外部端口为 `8080`，目标端口为 `80`（NGINX 默认侦听此端口）。此处我们之所以使用 `virtctl` 命令，是因为它能够识别虚拟机对象与 Pod 之间的映射。这为我们创建了新服务：

```
$ virtctl expose vmi ingress-example --port=8080 --target-port=80 --
name=ingress-example
Service ingress-example successfully exposed for vmi ingress-example
```

然后会自动创建一个适当的服务：

```
$ kubectl get svc/ingress-example
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP       PORT(S)
                   AGE
ingress-example   ClusterIP      10.43.217.19    <none>            8080/TCP
                   9s
```

接下来，如果您使用 `kubectl create ingress`，则可以创建一个指向此服务的 ingress 对象。此处请根据您的 DNS 配置修改 URL（在 ingress (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_create/kubectl_create_ingress/) 对象中称为“host”），并确保将其指向端口 `8080`：

```
$ kubectl create ingress ingress-example --rule=ingress-example.suse.local/
=ingress-example:8080
```

正确配置 DNS 后，可以立即对 URL 运行 curl 命令：

```
$ curl ingress-example.suse.local
It works!
```

我们来通过去除此虚拟机及其服务和入口资源进行清理：

```
$ kubectl delete vm/ingress-example svc/ingress-example ingress/ingress-example
virtualmachine.kubevirt.io "ingress-example" deleted
service "ingress-example" deleted
ingress.networking.k8s.io "ingress-example" deleted
```

21.7 使用 Rancher UI 扩展

SUSE Edge Virtualization 为 Rancher Manager 提供了 UI 扩展，让您可以使用 Rancher 仪表板 UI 进行基本的虚拟机管理。

21.7.1 安装

请参见第 6 章 “Rancher 仪表板扩展”获取安装指导。

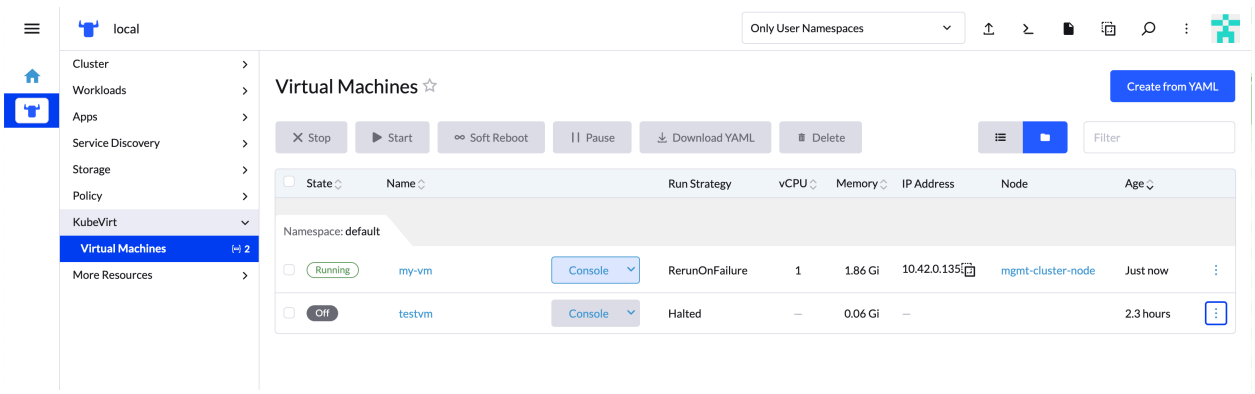
21.7.2 使用 KubeVirt Rancher 仪表板扩展

该扩展在群集资源管理器中引入了新的 **KubeVirt** 部分。此部分已添加到装有 KubeVirt 的任何受管群集。

使用该扩展，您可以直接与 KubeVirt 虚拟机资源进行交互，以管理虚拟机生命周期。

21.7.2.1 创建虚拟机

1. 单击左侧导航栏中已启用 KubeVirt 的受管群集，导航到 **Cluster Explorer**（群集资源管理器）。
2. 导航到 **KubeVirt > Virtual Machines（虚拟机）**页面，然后单击屏幕右上角的 `Create from YAML`（基于 YAML 创建）。
3. 填写或粘贴虚拟机定义，然后单击 `Create`（创建）。使用“部署虚拟机”一节中创建的虚拟机定义作为灵感来源。



21.7.2.2 虚拟机操作

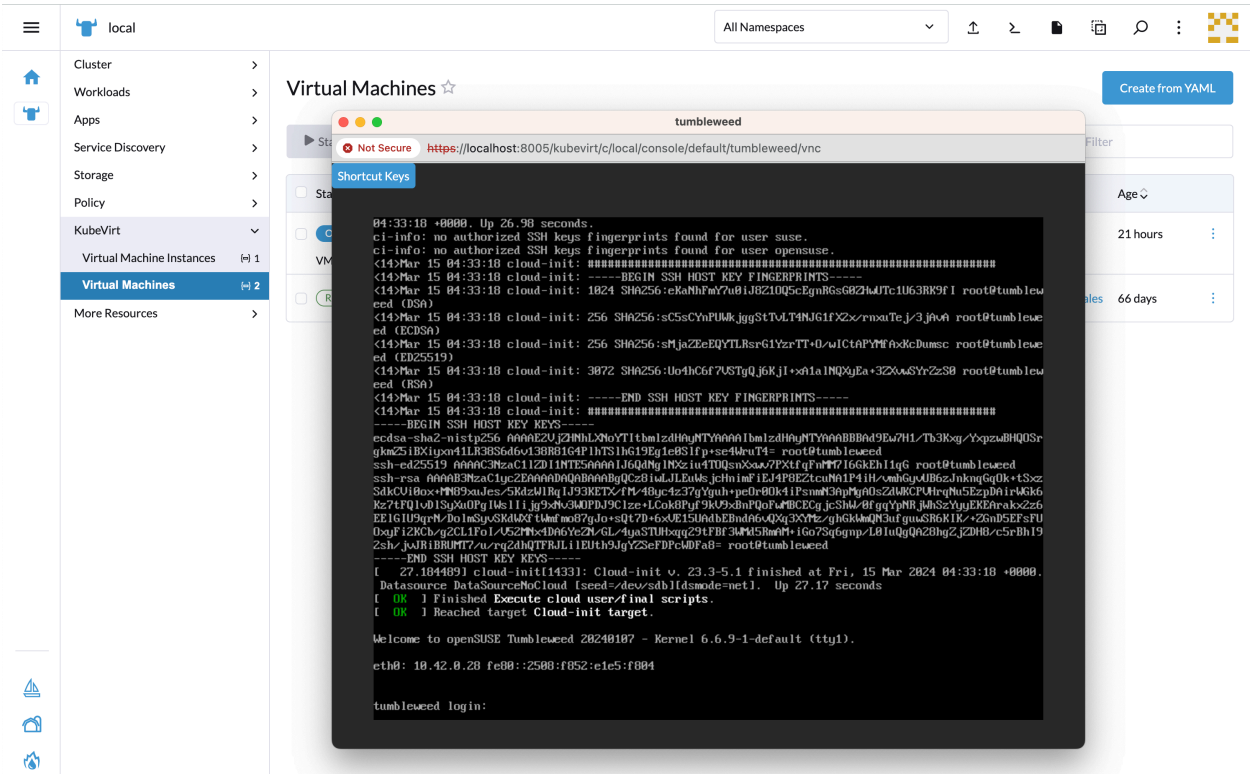
使用操作菜单（可通过每个虚拟机右侧的 # 下拉列表访问）来执行启动、停止、暂停或软重引导操作。或者，您也可以选中多个虚拟机并使用列表顶部的组操作来对它们执行操作。

执行这些操作可能会对虚拟机运行策略产生影响。有关详细信息，请参见 KubeVirt 文档中的表格 (https://kubevirt.io/user-guide/compute/run_strategies/#virtctl)。

21.7.2.3 访问虚拟机控制台

“Virtual machines”（虚拟机）列表提供了控制台下拉列表，用于通过 **VNC 或串行控制台**连接到虚拟机。此操作仅适用于正在运行的虚拟机。

在某些情况下，需要等待一段时间才能在全新启动的虚拟机上访问控制台。



21.8 使用 Edge Image Builder 进行安装

SUSE Edge 使用第 11 章 “Edge Image Builder” 来自定义基础 SUSE Linux Micro 操作系统映像。请按照第 27.9 节 “KubeVirt 和 CDI 安装” 中所述，在 EIB 置备的 Kubernetes 群集上进行 KubeVirt 和 CDI 隔离式安装。

22 系统升级控制器

请参见系统升级控制器文档 (https://github.com/rancher/system-upgrade-controller)。

> 系统升级控制器 (SUC) 旨在提供一个针对节点的 Kubernetes 原生通用升级控制器。它引入了一个新的 CRD，即计划，用于定义您的各种升级策略/要求。计划的主要意图是改变群集中的节点。

22.1 SUSE Edge 如何使用系统升级控制器?

SUSE Edge 使用 SUC 来协助完成管理群集和下游群集中与操作系统及 Kubernetes 版本升级相关的各类“Day 2”操作。

“Day 2”操作通过 SUC 计划来定义。SUC 会根据这些计划在每个节点上部署工作负载，以执行相应的“Day 2”操作。

第 23 章 “升级控制器”中也会使用 SUC。要了解 SUC 与升级控制器之间的主要区别，请参见第 23.2 节 “升级控制器与系统升级控制器”。

22.2 安装系统升级控制器

重要

从 Rancher 2.10.0 (https://github.com/rancher/rancher/releases/tag/v2.10.0) 版本开始，会自动安装系统升级控制器。

仅当您的环境**未**由 Rancher 管理，或者您的 Rancher 版本低于 `2.10.0` 时，才需要执行以下步骤。

建议您通过位于 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库中的 Fleet（第 8 章 “Fleet”）安装 SUC。

注意

`suse-edge/fleet-examples` 储存库提供的资源**必须**始终在有效的 fleet-examples 版本 (https://github.com/suse-edge/fleet-examples/releases) 中使用。要确定需使用哪个版本，请参见发行说明（第 52.1 节 “摘要”）。

如果无法使用 Fleet 安装 SUC，可以通过 Rancher 的 Helm chart 储存库进行安装，或者将 Rancher 的 Helm chart 整合到您自己的第三方 GitOps 工作流程中。

本节介绍如下内容：

- Fleet 安装（第 22.2.1 节 “系统升级控制器 Fleet 安装”）
- Helm 安装（第 22.2.2 节 “系统升级控制器 Helm 安装”）

22.2.1 系统升级控制器 Fleet 安装

使用 Fleet 时，可以通过两种资源来部署 SUC：

- GitRepo (https://fleet.rancher.io/ref-gitrepo) 资源 - 适用于有外部/本地 Git 服务器可用的使用场景。有关安装说明，请参见第 22.2.1.1 节 “系统升级控制器安装 - GitRepo”。
- 捆绑包 (https://fleet.rancher.io/bundle-add) 资源 - 适用于不支持本地 Git 服务器选项的隔离使用场景。有关安装说明，请参见第 22.2.1.2 节 “系统升级控制器安装 - 捆绑包”。

22.2.1.1 系统升级控制器安装 - GitRepo

注意

此过程也可以通过 Rancher UI 完成（如果有相关 UI）。有关详细信息，请参见在 Rancher UI 中访问 Fleet (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

在您的**管理**群集中：

1. 确定要在哪些群集上部署 SUC，方法是在**管理**群集内的适当 Fleet 工作空间中部署 SUC `GitRepo`。默认情况下，Fleet 有两个工作空间：

   - `fleet-local` - 用于需要部署在**管理**群集上的资源。
   - `fleet-default` - 用于需要部署在**下游**群集上的资源。
     有关 Fleet 工作空间的详细信息，请参见上游 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 文档。

2. 部署 `GitRepo` 资源：

   - 要在管理群集上部署 SUC，请使用：

```
kubectl apply -n fleet-local -f - <<EOF
apiVersion: fleet.cattle.io/v1alpha1
kind: GitRepo
metadata:
  name: system-upgrade-controller
spec:
  revision: release-3.3.0
  paths:
  - fleets/day2/system-upgrade-controller
  repo: https://github.com/suse-edge/fleet-examples.git
EOF
```

   - 要在下游群集上部署 SUC，请使用：

注意

在部署下方资源之前，您**必须**提供有效的目标配置，以便 Fleet 了解资源要部署至哪些下游群集。有关如何映射到下游群集的信息，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)。

```
kubectl apply -n fleet-default -f - <<EOF
apiVersion: fleet.cattle.io/v1alpha1
```

```
kind: GitRepo
metadata:
  name: system-upgrade-controller
spec:
  revision: release-3.3.0
  paths:
  - fleets/day2/system-upgrade-controller
  repo: https://github.com/suse-edge/fleet-examples.git
  targets:
  - clusterSelector: CHANGEME
  # Example matching all clusters:
  # targets:
  # - clusterSelector: {}
EOF
```

3. 验证 `GitRepo` 资源是否已部署：

```
# Namespace will vary based on where you want to deploy SUC
kubectl get gitrepo system-upgrade-controller -n <fleet-local/fleet-
default>

NAME                          REPO
   COMMIT          BUNDLEDEPLOYMENTS-READY   STATUS
system-upgrade-controller   https://github.com/suse-edge/fleet-examples.git
   release-3.3.0   1/1
```

4. 验证系统升级控制器部署：

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller   1/1     1            1           2m20s
```

22.2.1.2 系统升级控制器安装 - 捆绑包

本节说明如何使用 fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet) 在标准 Fleet 配置下构建并部署捆绑包资源。

1. 在连接网络的计算机上下载 `fleet-cli`：

注意

确保您下载的 fleet-cli 版本与群集上部署的 Fleet 版本匹配。

- 对于 Mac 用户，有一个 fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae。
- 对于 Linux 和 Windows 用户，每个 Fleet 版本 (https://github.com/rancher/fleet/releases) 都会有作为**资产**存在的二进制文件。
  - Linux AMD：

```
curl -L -o fleet-cli https://github.com/rancher/fleet/releases/download/v0.12.2/fleet-linux-amd64
```

  - Linux ARM：

```
curl -L -o fleet-cli https://github.com/rancher/fleet/releases/download/v0.12.2/fleet-linux-arm64
```

2. 将 `fleet-cli` 设为可执行文件：

```
chmod +x fleet-cli
```

3. 克隆您要使用的 `suse-edge/fleet-examples` 版本 (https://github.com/suse-edge/fleet-examples/releases)：

```
git clone -b release-3.3.0 https://github.com/suse-edge/fleet-examples.git
```

4. 导航到 `fleet-examples` 储存库中的 SUC Fleet：

```
cd fleet-examples/fleets/day2/system-upgrade-controller
```

5. 确定要在哪些群集上部署 SUC，方法是在管理群集内的适当 Fleet 工作空间中部署 SUC 捆绑包。默认情况下，Fleet 有两个工作空间：

- `fleet-local` - 用于需要部署在**管理**群集上的资源。
- `fleet-default` - 用于需要部署在**下游**群集上的资源。
  有关 Fleet 工作空间的详细信息，请参见上游 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 文档。

6. 如果您只打算在下游群集上部署 SUC，请创建一个与特定群集匹配的 `targets.yaml` 文件：

```
cat > targets.yaml <<EOF
targets:
- clusterSelector: CHANGEME
EOF
```

有关如何映射到下游群集的信息，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)

7. 继续构建捆绑包：

注意

请确保您**没有**下载 `fleet-examples/fleets/day2/system-upgrade-controller` 目录中的 fleet-cli，否则它将和捆绑包封装在一起，不建议这么做。

- 要在管理群集上部署 SUC，请执行：

```
fleet-cli apply --compress -n fleet-local -o - system-upgrade-controller . > system-upgrade-controller-bundle.yaml
```

- 要在下游群集上部署 SUC，请执行：

```
fleet-cli apply --compress --targets-file=targets.yaml -n fleet-default -o - system-upgrade-controller . > system-upgrade-controller-bundle.yaml
```

有关此过程的详细信息，请参见将 Helm Chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle)。

有关 `fleet-cli apply` 命令的详细信息，请参见 fleet apply (https://fleet.rancher.io/cli/fleet-cli/fleet_apply)。

8. 将 `system-upgrade-controller-bundle.yaml` 捆绑包传输到您的管理群集计算机：

```
scp system-upgrade-controller-bundle.yaml <machine-address>:<filesystem-path>
```

9. 在您的管理群集上，部署 `system-upgrade-controller-bundle.yaml` 捆绑包：

```
kubectl apply -f system-upgrade-controller-bundle.yaml
```

10. 在您的管理群集上，验证捆绑包是否已部署：

```
# Namespace will vary based on where you want to deploy SUC
kubectl get bundle system-upgrade-controller -n <fleet-local/fleet-default>

NAME                        BUNDLEDEPLOYMENTS-READY   STATUS
system-upgrade-controller   1/1
```

11. 根据捆绑包部署到的 Fleet 工作空间，导航到群集并验证 SUC 部署：

注意

SUC 始终部署在 **cattle-system** 名称空间中。

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller   1/1     1            1           111s
```

22.2.2 系统升级控制器 Helm 安装

1. 添加 Rancher chart 储存库：

```
helm repo add rancher-charts https://charts.rancher.io/
```

2. 部署 SUC chart：

```
helm install system-upgrade-controller rancher-charts/system-upgrade-
controller --version 106.0.0 --set global.cattle.psp.enabled=false -n
 cattle-system --create-namespace
```

此命令将安装 Edge 3.3.1 平台所需的 SUC 0.15.2 版本。

3. 验证 SUC 部署：

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller    1/1     1            1           37s
```

22.3 监控系统升级控制器计划

可以通过以下方式查看 SUC 计划：

- 通过 Rancher UI（第 22.3.1 节 “监控系统升级控制器计划 - Rancher UI”）。
- 通过群集内部的手动监控（第 22.3.2 节 “监控系统升级控制器计划 - 手动”）。

重要

为 SUC 计划部署的 Pod 在成功执行后会保持 **15** 分钟的活动状态。之后，创建这些 Pod 的相应作业会将其去除。如果要在这段时间后访问 Pod 的日志，您应该为群集启用日志记录。有关如何在 Rancher 中执行此操作的信息，请参见 Rancher 与日志记录服务的集成 (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/logging)。

22.3.1 监控系统升级控制器计划 - Rancher UI

要检查特定 SUC 计划的 Pod 日志，请执行以下操作：

1. 在左上角，选择 **# → <您的群集名称>**
2. 选择“Workloads”（工作负载）→“Pod”
3. 选择 `Only User Namespaces`（仅用户名称空间）下拉菜单，并添加 `cattle-system` 名称空间。
4. 在 Pod 过滤栏中输入 SUC 计划 Pod 的名称。该名称采用以下模板格式：`apply-<计划名称>-on-<节点名称>`

注意

对于特定的 SUC 计划，可能同时有已完成和未知的 Pod。这是正常的，原因要归咎于一些升级的性质。

5. 选择您要查看其日志的 Pod，然后导航到 **# → View Logs（查看日志）**

22.3.2 监控系统升级控制器计划 - 手动

注意

以下步骤假设 `kubectl` 已配置为连接到已部署 **SUC 计划**的群集。

1. 列出部署的 **SUC** 计划：

```
kubectl get plans -n cattle-system
```

2. 获取 **SUC** 计划的 Pod：

```
kubectl get pods -l upgrade.cattle.io/plan=<plan_name> -n cattle-system
```

注意

对于特定的 SUC 计划，可能同时有已完成和未知的 Pod。这是正常的，原因要归咎于一些升级的性质。

3. 获取 Pod 的日志：

```
kubectl logs <pod_name> -n cattle-system
```

23 升级控制器

Kubernetes 控制器可以执行以下 SUSE Edge 平台组件的升级：

- 操作系统 (SUSE Linux Micro)
- Kubernetes（K3s 和 RKE2）
- 其他组件（Rancher、Elemental、SUSE Security 等）

升级控制器 (https://github.com/suse-edge/upgrade-controller) 通过将上述组件的复杂性封装在一个面向用户的资源（该资源可作为升级**触发器**）中，来简化这些组件的升级过程。用户只需配置此资源，其余工作均由升级控制器处理。

注意

目前，升级控制器仅支持为**非隔离管理**群集进行 SUSE Edge 平台升级。有关详细信息，请参见第 23.7 节 “已知限制”。

23.1 SUSE Edge 如何使用升级控制器？

升级控制器对于将管理群集从一个 SUSE Edge 版本升级到下一个版本时需自动执行（以前是手动执行）的“Day 2”操作至关重要。

为了实现这种自动化，升级控制器使用了系统升级控制器（第 22 章 “系统升级控制器”）和 Helm 控制器 (https://github.com/k3s-io/helm-controller/) 等工具。

有关升级控制器工作原理的详细信息，请参见第 23.4 节 “升级控制器的工作原理”。

有关升级控制器的已知限制，请参见第 23.7 节 “已知限制”。

如需了解升级控制器与系统升级控制器的区别，请参见第 23.2 节 “升级控制器与系统升级控制器”。

23.2 升级控制器与系统升级控制器

系统升级控制器 (SUC)（第 22 章 “系统升级控制器”）是一款通用工具，负责将升级指令分发到特定的 Kubernetes 节点。

虽然它支持 SUSE Edge 平台的部分“Day 2”操作，但**并未**涵盖所有此类操作。此外，即便是对于受支持的操作，用户也必须手动配置、维护和部署多个 SUC 计划，因为这一过程容易出错，可能导致意外问题。

这就催生了对下面这样的工具的需求：该工具能够**自动化**并**抽象化** SUSE Edge 平台各类“Day 2”操作管理的复杂性。因此，升级控制器应运而生。它通过引入一个面向用户的单一资源来驱动升级，从而简化了升级过程。用户只需管理这一资源，其余工作均由升级控制器处理。

23.3 安装升级控制器

23.3.1 先决条件

- Helm (https://helm.sh/docs/intro/install/)
- cert-manager (https://cert-manager.io/v1.15-docs/installation/helm/#installing-with-helm)
- 系统升级控制器（第 22.2 节 “安装系统升级控制器”）
- 一个 Kubernetes 群集，可以是 K3s 或 RKE2

23.3.2 步骤

1. 在您的管理群集上安装升级控制器 Helm chart：

```
helm install upgrade-controller oci://registry.suse.com/edge/charts/
upgrade-controller --version 303.0.1+up0.1.1 --create-namespace --namespace
 upgrade-controller-system
```

2. 验证升级控制器部署：

```
kubectl get deployment -n upgrade-controller-system
```

3. 验证升级控制器 Pod：

```
kubectl get pods -n upgrade-controller-system
```

4. 验证升级控制器 Pod 日志：

```
kubectl logs <pod_name> -n upgrade-controller-system
```

23.4 升级控制器的工作原理

为了执行 Edge 版本升级，升级控制器引入了两个新的 Kubernetes 自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)：

- UpgradePlan（第 23.5.1 节 “UpgradePlan”）- 由用户创建；保存有关 Edge 版本升级的配置。
- ReleaseManifest（第 23.5.2 节 “ReleaseManifest”）- 由升级控制器创建；保存特定的 Edge 版本所对应的组件版本。**用户不得编辑此文件。**

升级控制器还会创建一个 `ReleaseManifest` 资源，该资源保存用户在 `UpgradePlan` 资源的 `releaseVersion` 属性下指定的 Edge 版本的组件数据。

之后，升级控制器会使用 `ReleaseManifest` 中的组件数据，按以下顺序升级 Edge 版本组件：

1. 操作系统 (OS)（第 23.4.1 节 “操作系统升级”）。
2. Kubernetes（第 23.4.2 节 “Kubernetes 升级”）。
3. 其他组件（第 23.4.3 节 “其他组件的升级”）。

注意

在升级过程中，升级控制器会持续向创建的 `UpgradePlan` 输出升级信息。有关如何跟踪升级过程的详细信息，请参见第 23.6 节 “跟踪升级过程”。

23.4.1 操作系统升级

升级操作系统时，升级控制器会创建命名模板如下的 SUC（第 22 章 “系统升级控制器”）计划：

- 对于与控制平面节点操作系统升级相关的 SUC 计划 - `control-plane-<os-name>-<os-version>-<suffix>`。
- 对于与工作节点操作系统升级相关的 SUC 计划 - `workers-<os-name>-<os-version>-<suffix>`。

SUC 会根据这些计划，继续在执行实际操作系统升级的群集的每个节点上创建工作负载。

根据 `ReleaseManifest`，操作系统升级可能包括：

- 仅软件包更新 - 适用于操作系统版本不随 Edge 版本变动的使用场景。
- 完整操作系统迁移 - 适用于操作系统版本随 Edge 版本变动的使用场景。

升级首先从控制平面节点开始，一次执行**一个**节点的升级。只有在控制平面节点的升级完成后，工作节点才会开始升级。

注意

如果群集有**多个**指定类型的节点，升级控制器会配置操作系统 SUC 计划来清空 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/)群集节点。

对于控制平面节点**有多个**而工作节点**只有一个**的群集，只会清空控制平面节点，反之亦然。

有关如何完全禁用节点清空的信息，请参见“UpgradePlan”（第 23.5.1 节 “UpgradePlan”）一节。

23.4.2 Kubernetes 升级

升级群集的 Kubernetes 发行版时，升级控制器会创建具有以下命名模板的 SUC（第 22 章 “系统升级控制器”）计划：

- 对于与控制平面节点 Kubernetes 升级相关的 SUC 计划 - `control-plane-<k8s-version>-<suffix>`。
- 对于与工作节点 Kubernetes 升级相关的 SUC 计划 - `workers-<k8s-version>-<suffix>`。

SUC 会根据这些计划，继续在执行实际 Kubernetes 升级的群集的每个节点上创建工作负载。

Kubernetes 升级将从控制平面节点开始，一次升级**一个**节点。只有在控制平面节点的升级完成后，工作节点才会开始升级。

注意

如果群集有**多个**指定类型的节点，升级控制器会配置 Kubernetes SUC 计划来清空 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/) 群集节点。

对于控制平面节点**有多个**而工作节点**只有一个**的群集，只会清空控制平面节点，反之亦然。

有关如何完全禁用节点清空的信息，请参见第 23.5.1 节 “UpgradePlan”。

23.4.3 其他组件的升级

目前，所有其他组件都是通过 Helm chart 安装的。有关特定版本各组件的完整列表，请参见发行说明（第 52.1 节 “摘要”）。

对于通过 EIB（第 11 章 “Edge Image Builder”）部署的 Helm chart，升级控制器会更新每个组件的现有 HelmChart CR (https://docs.rke2.io/helm#using-the-helm-crd)。

对于在 EIB 之外部署的 Helm chart，升级控制器会为每个组件创建一个 `HelmChart` 资源。

在创建/更新 `HelmChart` 资源后，升级控制器会依赖 helm-controller (https://github.com/k3s-io/helm-controller/) 来拾取此更改并继续执行实际的组件升级。

Chart 将按照 `ReleaseManifest` 中的顺序依次升级。您还可通过 `UpgradePlan` 传递其他值。如果在新 SUSE Edge 版本中某个 chart 的版本保持不变，则不会将其升级。有关这方面的详细信息，请参见第 23.5.1 节 “UpgradePlan”。

23.5 Kubernetes API 扩展

升级控制器引入的 Kubernetes API 的扩展。

23.5.1 UpgradePlan

升级控制器引入了一个名为 `UpgradePlan` 的新 Kubernetes 自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)。`UpgradePlan` 是升级控制器的指令机制，支持以下配置：

- `releaseVersion` - 群集应升级到的 Edge 版本。版本必须遵循语义 (https://semver.org)版本规范，并且应从发行说明（第 52.1 节 “摘要”）中检索。
- `disableDrain` - **可选**；指示升级控制器是否禁用节点清空 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/)。当您有具有干扰预算 (https://kubernetes.io/docs/tasks/run-application/configure-pdb/)的工作负载时，可以用到。
  - 禁用控制平面节点清空的示例：

```
spec:
  disableDrain:
    controlPlane: true
```

  - 禁用控制平面和工作节点清空的示例：

```
spec:
  disableDrain:
    controlPlane: true
    worker: true
```

- `helm` - **可选**；为通过 Helm 安装的组件指定其他值。

警告

建议仅将此字段用于对升级至关重要的值。标准 chart 值的更新应在相应 chart 升级到下一版本后进行。

- 示例：

```
spec:
  helm:
  - chart: foo
    values:
      bar: baz
```

23.5.2 ReleaseManifest

升级控制器引入了一个名为 `ReleaseManifest` 的新 Kubernetes 自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)。

`ReleaseManifest` 资源由升级控制器创建，保存**一个**特定 Edge 版本的组件数据。这意味着每个 Edge 版本升级都将由不同的 `ReleaseManifest` 资源表示。

警告

ReleaseManifest 只能由升级控制器创建。

不建议手动创建或编辑 `ReleaseManifest` 资源，否则，用户需**自行承担风险**。

版本清单附带的组件数据包括但不限于：

- 操作系统数据 - 版本、支持的体系结构、其他升级数据等
- Kubernetes 发行版数据 - RKE2 (https://docs.rke2.io)/K3s (https://k3s.io) 支持的版本
- 其他组件数据 - SUSE Helm chart 数据（位置、版本、名称等）

有关版本清单的示例，请参见上游 (https://github.com/suse-edge/upgrade-controller/blob/main/config/samples/lifecycle_v1alpha1_releasemanifest.yaml) 文档。**请注意，这只是一个示例，不可用于创建为有效的 `ReleaseManifest` 资源。**

23.6 跟踪升级过程

本节介绍如何跟踪和调试升级控制器在用户创建 `UpgradePlan` 资源后启动的升级资源过程。

23.6.1 一般信息

有关升级过程状态的一般信息，可以在升级计划的状态条件中查看。

可以通过以下方式查看升级计划资源的状态：

```
kubectl get upgradeplan <upgradeplan_name> -n upgrade-controller-system -o yaml
```

升级计划运行示例：

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  namespace: upgrade-controller-system
spec:
  releaseVersion: 3.3.1
status:
  conditions:
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Control plane nodes are being upgraded
    reason: InProgress
    status: "False"
    type: OSUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Kubernetes upgrade is not yet started
    reason: Pending
    status: Unknown
```

```
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Rancher upgrade is not yet started
    reason: Pending
    status: Unknown
    type: RancherUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Longhorn upgrade is not yet started
    reason: Pending
    status: Unknown
    type: LonghornUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: MetalLB upgrade is not yet started
    reason: Pending
    status: Unknown
    type: MetalLBUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: CDI upgrade is not yet started
    reason: Pending
    status: Unknown
    type: CDIUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: KubeVirt upgrade is not yet started
    reason: Pending
    status: Unknown
    type: KubeVirtUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: NeuVector upgrade is not yet started
    reason: Pending
    status: Unknown
    type: NeuVectorUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
```

```
    status: Unknown
```

```
    type: ElementalUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: SRIOV upgrade is not yet started
    reason: Pending
    status: Unknown
    type: SRIOVUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Akri upgrade is not yet started
    reason: Pending
    status: Unknown
    type: AkriUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Metal3 upgrade is not yet started
    reason: Pending
    status: Unknown
    type: Metal3Upgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: RancherTurtles upgrade is not yet started
    reason: Pending
    status: Unknown
    type: RancherTurtlesUpgraded
  observedGeneration: 1
  sucNameSuffix: 90315a2b6d
```

通过这种方式，您可以查看升级控制器将尝试安排升级的每个组件。每个条件都采用以下模板：

- `lastTransitionTime` - 此组件条件上一次改变状态的时间。
- `message` - 指示特定组件条件当前升级状态的消息。
- `reason` - 特定组件条件的当前升级状态。可能的 `reason` 包括：

    - `Pending` - 尚未安排特定组件的升级。
    - `Skipped` - 在群集上找不到特定组件，因此将跳过其升级。
    - `Error` - 特定组件发生了临时错误。
- `status` - 当前条件 `type` 的状态，可以是 `True`、`False` 或 `Unknown`。
- `type` - 当前已升级组件的指示器。

升级控制器会为 `OSUpgraded` 和 `KubernetesUpgraded` 类型的组件条件创建 SUC 计划。要进一步跟踪为这些组件创建的 SUC 计划，请参见第 22.3 节 “监控系统升级控制器计划”。

要进一步跟踪所有其他组件条件类型，可以查看 helm-controller (https://github.com/k3s-io/helm-controller/) 为它们创建的资源。有关详细信息，请参见第 23.6.2 节 “Helm 控制器”。

当满足以下条件时，升级控制器安排的升级计划可以标记为 `successful`：

1. 不存在 `Pending` 或 `InProgress` 的组件条件。
2. `lastSuccessfulReleaseVersion` 属性指向升级计划配置中指定的 `releaseVersion`。**升级过程成功后，升级控制器会将此属性添加至升级计划的状态。**

成功的 `UpgradePlan` 示例：

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  namespace: upgrade-controller-system
spec:
  releaseVersion: 3.3.1
status:
  conditions:
  - lastTransitionTime: "2024-10-01T06:26:48Z"
    message: All cluster nodes are upgraded
    reason: Succeeded
    status: "True"
    type: OSUpgraded
```

```
  - lastTransitionTime: "2024-10-01T06:26:59Z"
    message: All cluster nodes are upgraded
    reason: Succeeded
    status: "True"
    type: KubernetesUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart rancher upgrade succeeded
    reason: Succeeded
    status: "True"
    type: RancherUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart longhorn is not installed
    reason: Skipped
    status: "False"
    type: LonghornUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Specified version of chart metallb is already installed
    reason: Skipped
    status: "False"
    type: MetalLBUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart cdi is not installed
    reason: Skipped
    status: "False"
    type: CDIUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart kubevirt is not installed
    reason: Skipped
    status: "False"
    type: KubeVirtUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
```

```
    reason: Skipped
    status: "False"
    type: EndpointCopierOperatorUpgraded
  - lastTransitionTime: "2024-10-01T06:27:14Z"
    message: Chart elemental-operator upgrade succeeded
    reason: Succeeded
    status: "True"
    type: ElementalUpgraded
  - lastTransitionTime: "2024-10-01T06:27:15Z"
    message: Chart sriov-crd is not installed
    reason: Skipped
    status: "False"
    type: SRIOVUpgraded
  - lastTransitionTime: "2024-10-01T06:27:16Z"
    message: Chart akri is not installed
    reason: Skipped
    status: "False"
    type: AkriUpgraded
  - lastTransitionTime: "2024-10-01T06:27:19Z"
    message: Chart metal3 is not installed
    reason: Skipped
    status: "False"
    type: Metal3Upgraded
  - lastTransitionTime: "2024-10-01T06:27:27Z"
    message: Chart rancher-turtles is not installed
    reason: Skipped
    status: "False"
    type: RancherTurtlesUpgraded
```



```
  lastSuccessfulReleaseVersion: 3.3.1
  observedGeneration: 1
```

注意

以下步骤假设 `kubectl` 已配置为连接到已部署升级控制器的群集。

1. 找到特定组件的 `HelmChart` 资源：

```
kubectl get helmcharts -n kube-system
```

2. 使用 `HelmChart` 资源的名称找到由 `helm-controller` 创建的升级 Pod：

```
kubectl get pods -l helmcharts.helm.cattle.io/chart=<helmchart_name> -n
 kube-system

# Example for Rancher
kubectl get pods -l helmcharts.helm.cattle.io/chart=rancher -n kube-system
NAME                          READY   STATUS      RESTARTS   AGE
helm-install-rancher-tv9wn    0/1     Completed   0          16m
```

3. 查看特定组件 Pod 的日志：

```
kubectl logs <pod_name> -n kube-system
```

23.7 已知限制

- 下游群集的升级目前还不受升级控制器管理。有关如何升级下游群集的信息，请参见第 36 章 “下游群集”。
- 升级控制器希望通过 EIB（第 11 章 “Edge Image Builder”）部署的任何其他 SUSE Edge Helm chart 将其 HelmChart CR (https://docs.rke2.io/helm#using-the-helm-crd) 部署在 `kube-system` 名称空间中。为此，请在 EIB 定义文件中配置 `installationNamespace` 属性。有关详细信息，请参见上游 (https://github.com/suse-edge/edge-image-builder/blob/main/docs/building-images.md#kubernetes) 文档。

- 目前，升级控制器无法确定管理群集上当前运行的 Edge 版本。请确保提供的 Edge 版本高于群集上当前运行的 Edge 版本。
- 目前，升级控制器仅支持**非隔离**环境升级，还不支持**隔离式**升级。

24 SUSE Multi-Linux Manager

SUSE Edge 中包含 SUSE Multi-Linux Manager，其作用是提供自动化与控制能力，确保边缘部署的所有节点上作为底层操作系统的 SUSE Linux Micro 始终保持最新状态。

有关详细信息，请参见第 4 章 “SUSE Multi-Linux Manager”和 SUSE Multi-Linux Manager 文档 (https://documentation.suse.com/suma/5.0/en/suse-manager/index.html)。

III 操作指南

操作指南和最佳实践

25 K3s 上的 MetalLB（使用第 2 层模式）

MetalLB 是使用标准路由协议的裸机 Kubernetes 群集的负载平衡器实现。

本指南将展示如何以第 2 层 (L2) 模式部署 MetalLB。

25.1 为何使用此方法

由于以下原因，MetalLB 成为了用来实现裸机 Kubernetes 群集负载平衡的较佳选择：

1. 与 Kubernetes 本机集成：MetalLB 可无缝与 Kubernetes 集成，让用户可使用熟悉的 Kubernetes 工具和方法轻松进行部署和管理。
2. 裸机兼容性：与基于云的负载平衡器不同，MetalLB 专门用于本地部署，在这种部署中，可能无法使用传统负载平衡器或使用传统负载平衡器不现实。
3. 支持多种协议：MetalLB 支持 Layer 2 和 BGP（边界网关协议）模式，提供了很大的灵活性，适合不同的网络架构并可满足不同的要求。
4. 高可用性：MetalLB 可以将负载平衡负担分散在多个节点中，从而确保服务的高可用性和可靠性。
5. 可伸缩性：MetalLB 可以处理大规模部署，并且能够将 Kubernetes 群集扩容来满足日益增长的需求。

在第 2 层模式下，一个节点负责向本地网络播发服务。从网络的角度看，似乎为该计算机的网络接口分配了多个 IP 地址。

第 2 层模式的主要优势在于其通用性：它可以在任何以太网上正常工作，不需要任何特殊硬件，甚至不需要各种形式的路由器。

25.2 K3s 上的 MetalLB（使用 L2）

本快速入门将使用 L2 模式。这意味着我们不需要任何特殊网络设备，只需使用网络范围内的三个空闲 IP。

25.3 先决条件

- 要部署 MetalLB 的 K3s 群集。

警告

K3S 自身附带服务负载平衡器（名为 Klipper）。需要禁用 Klipper 才能运行 MetalLB (https://metallb.universe.tf/configuration/k3s/)。要禁用 Klipper，需使用 `--disable=servicelb` 标志安装 K3s。

- Helm
- 网络范围内的三个空闲 IP 地址。在本示例中为 `192.168.122.10-192.168.122.12`

重要

必须确保这三个 IP 地址未被分配。在 DHCP 环境中，为了避免出现双重分配情况，不得将这些地址添加到 DHCP 池中。

25.4 部署

我们将使用作为 SUSE Edge 解决方案一部分发布的 MetalLB Helm chart：

```
helm install \
  metallb oci://registry.suse.com/edge/charts/metallb \
  --namespace metallb-system \
  --create-namespace

while ! kubectl wait --for condition=ready -n metallb-system $(kubectl get\
 pods -n metallb-system -l app.kubernetes.io/component=controller -o name)\
 --timeout=10s; do
 sleep 2
done
```

25.5 配置

安装现已完成。接下来请使用示例值进行配置 (https://metallb.universe.tf/configuration/)：

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ip-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.122.10/32
  - 192.168.122.11/32
  - 192.168.122.12/32
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - ip-pool
EOF
```

现在，MetalLB 可供您使用。可以自定义 L2 模式的许多设置，例如：

- IPv6 和双栈服务 (https://metallb.universe.tf/usage/#ipv6-and-dual-stack-services)
- 控制自动地址分配 (https://metallb.universe.tf/configuration/_advanced_ipaddresspool_configuration/#controlling-automatic-address-allocation)

- 将地址分配范围缩小为特定的名称空间和服务 (https://metallb.universe.tf/configuration/_advanced_ipaddresspool_configuration/#reduce-scope-of-address-allocation-to-specific-namespace-and-service)
- 限制可从中声明服务的节点集 (https://metallb.universe.tf/configuration/_advanced_l2_configuration/#limiting-the-set-of-nodes-where-the-service-can-be-announced-from)
- 指定可从中声明 LB IP 的网络接口 (https://metallb.universe.tf/configuration/_advanced_l2_configuration/#specify-network-interfaces-that-lb-ip-can-be-announced-from)

还可以对 BGP (https://metallb.universe.tf/configuration/_advanced_bgp_configuration/) 进行其他许多自定义设置。

25.5.1 Traefik 和 MetalLB

默认情况下，Traefik 会随 K3s 一起部署（可以使用 `--disable=traefik` 禁用 Traefik (https://docs.k3s.io/networking#traefik-ingress-controller)），并作为 `LoadBalancer` 公开（与 Klipper 一起使用）。但是，由于需要禁用 Klipper，用于入口的 Traefik 服务仍是 `LoadBalancer` 类型。因此在部署 MetalLB 的那一刻，第一个 IP 将自动分配给 Traefik 入口。

```
# Before deploying MetalLB
kubectl get svc -n kube-system traefik
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
   AGE
traefik   LoadBalancer   10.43.44.113   <pending>     80:31093/TCP,443:32095/TCP
   28s
# After deploying MetalLB
kubectl get svc -n kube-system traefik
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)
      AGE
traefik   LoadBalancer   10.43.44.113   192.168.122.10   80:31093/TCP,443:32095/
TCP   3m10s
```

我们将在稍后的过程（第 25.6.1 节 “MetalLB 的入口”）中应用此操作。

25.6 用法

我们来创建示例部署：

```
cat <<- EOF | kubectl apply -f -
---
apiVersion: v1
kind: Namespace
metadata:
  name: hello-kubernetes
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: hello-kubernetes
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello-kubernetes
    spec:
      serviceAccountName: hello-kubernetes
      containers:
```

```
        - name: hello-kubernetes
          image: "paulbouwer/hello-kubernetes:1.10"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          env:
          - name: HANDLER_PATH_PREFIX
            value: ""
          - name: RENDER_PATH_PREFIX
            value: ""
          - name: KUBERNETES_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: KUBERNETES_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: KUBERNETES_NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: spec.nodeName
          - name: CONTAINER_IMAGE
            value: "paulbouwer/hello-kubernetes:1.10"
EOF
```

最后创建服务：

```
cat <<- EOF | kubectl apply -f -
```

```
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: hello-kubernetes
EOF
```

我们来看看此示例的实际效果：

```
kubectl get svc -n hello-kubernetes
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)
 AGE
hello-kubernetes   LoadBalancer   10.43.127.75   192.168.122.11   80:31461/TCP
 8s

curl http://192.168.122.11
<!DOCTYPE html>
<html>
<head>
    <title>Hello Kubernetes!</title>
    <link rel="stylesheet" type="text/css" href="/css/main.css">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Ubuntu:300" >
</head>
<body>

  <div class="main">
    <img src="/images/kubernetes.png"/>
```

```
    <div class="content">
      <div id="message">
  Hello world!
</div>
<div id="info">
  <table>
    <tr>
      <th>namespace:</th>
      <td>hello-kubernetes</td>
    </tr>
    <tr>
      <th>pod:</th>
      <td>hello-kubernetes-7c8575c848-2c6ps</td>
    </tr>
    <tr>
      <th>node:</th>
      <td>allinone (Linux 5.14.21-150400.24.46-default)</td>
    </tr>
  </table>
</div>
<div id="footer">
  paulbouwer/hello-kubernetes:1.10 (linux/amd64)
</div>
    </div>
  </div>

</body>
</html>
```

25.6.1 MetalLB 的入口

由于 Traefik 已用作入口控制器，我们可以通过 `Ingress` 对象公开任何 HTTP / HTTPS 流量，例如：

```
IP=$(kubectl get svc -n kube-system traefik -o
 jsonpath="{.status.loadBalancer.ingress[0].ip}")
cat <<- EOF | kubectl apply -f -
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: hello-kubernetes-ingress
  namespace: hello-kubernetes
spec:
  rules:
  - host: hellok3s.${IP}.sslip.io
    http:
      paths:
        - path: "/"
          pathType: Prefix
          backend:
            service:
              name: hello-kubernetes
              port:
                name: http
EOF
```

然后运行：

```
curl http://hellok3s.${IP}.sslip.io
<!DOCTYPE html>
<html>
<head>
    <title>Hello Kubernetes!</title>
    <link rel="stylesheet" type="text/css" href="/css/main.css">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Ubuntu:300" >
</head>
<body>

  <div class="main">
    <img src="/images/kubernetes.png"/>
    <div class="content">
      <div id="message">
  Hello world!
</div>
<div id="info">
```

```
  <table>
    <tr>
      <th>namespace:</th>
      <td>hello-kubernetes</td>
    </tr>
    <tr>
      <th>pod:</th>
      <td>hello-kubernetes-7c8575c848-fvqm2</td>
    </tr>
    <tr>
      <th>node:</th>
      <td>allinone (Linux 5.14.21-150400.24.46-default)</td>
    </tr>
  </table>
</div>
<div id="footer">
  paulbouwer/hello-kubernetes:1.10 (linux/amd64)
</div>
    </div>
  </div>

</body>
</html>
```

校验 MetalLB 是否正常工作：

```
% arping hellok3s.${IP}.sslip.io

ARPING 192.168.64.210
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=0 time=1.169 msec
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=1 time=2.992 msec
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=2 time=2.884 msec
```

在以上示例中，流量的流动方式如下：

1. `hellok3s.${IP}.sslip.io` 解析为实际 IP。
2. 然后，流量由 `metallb-speaker` Pod 处理。

3. `metallb-speaker` 将流量重定向到 `traefik` 控制器。

4. 最后，Traefik 将请求转发到 `hello-kubernetes` 服务。

26 Kubernetes API 服务器前面的 MetalLB

本指南演示如何使用 MetalLB 服务在包含三个控制平面节点的 HA 群集上向外部公开 RKE2/K3s API。为此，需要手动创建类型为 `LoadBalancer` 的 Kubernetes 服务，并创建端点。端点会保留群集中所有控制平面节点的 IP。为了使端点与群集中发生的事件（添加/去除节点或节点下线）持续保持同步，需要部署 Endpoint Copier Operator（第 20 章 “Endpoint Copier Operator”）。该操作器会监控默认 `kubernetes` 端点中发生的事件，并会自动更新受管服务以使其保持同步。由于受管服务的类型为 `LoadBalancer`，因此 MetalLB 为其分配了静态 `ExternalIP`。此 `ExternalIP` 用于与 API 服务器通讯。

26.1 先决条件

- 要在其上部署 RKE2/K3s 的三台主机。
  - 请确保这些主机的主机名不同。
  - 对于测试目的，这些主机可以是虚拟机
- 网络中至少有 2 个可用 IP（一个用于 Traefik/Nginx，另一个用于受管服务）。
- Helm

26.2 安装 RKE2/K3s

注意

如果您不想使用新群集，而要使用现有群集，请跳过此步骤并执行下一步。

首先，必须在网络中预留一个可用 IP，该 IP 稍后将用作受管服务的 `ExternalIP`。

通过 SSH 连接到第一台主机并以群集模式安装所需的发行版。

对于 RKE2，使用以下命令：

```
# Export the free IP mentioned above
```

```
export VIP_SERVICE_IP=<ip>

curl -sfL https://get.rke2.io | INSTALL_RKE2_EXEC="server \
 --write-kubeconfig-mode=644 --tls-san=${VIP_SERVICE_IP} \
 --tls-san=https://${VIP_SERVICE_IP}.sslip.io" sh -

systemctl enable rke2-server.service
systemctl start rke2-server.service

# Fetch the cluster token:
RKE2_TOKEN=$(tr -d '\n' < /var/lib/rancher/rke2/server/node-token)
```

对于 K3s，使用以下命令：

```
# Export the free IP mentioned above
export VIP_SERVICE_IP=<ip>
export INSTALL_K3S_SKIP_START=false

curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server --cluster-init \
 --disable=servicelb --write-kubeconfig-mode=644 --tls-san=${VIP_SERVICE_IP} \
 --tls-san=https://${VIP_SERVICE_IP}.sslip.io" K3S_TOKEN=foobar sh -
```

注意

确保在 `k3s server` 命令中提供 `--disable=servicelb` 标志。

重要

从现在开始，应在本地计算机上运行命令。

要从外部访问 API 服务器，需使用 RKE2/K3s VM 的 IP。

```
# Replace <node-ip> with the actual IP of the machine
export NODE_IP=<node-ip>
export KUBE_DISTRIBUTION=<k3s/rke2>

scp ${NODE_IP}:/etc/rancher/${KUBE_DISTRIBUTION}/${KUBE_DISTRIBUTION}.yaml
 ~/.kube/config && sed \
```

```
-i '' "s/127.0.0.1/${NODE_IP}/g" ~/.kube/config && chmod 600 ~/.kube/config
```

26.3 配置现有群集

注意

仅当您要使用现有的 RKE2/K3s 群集时，此步骤才有效。

要使用现有群集，应修改 `tls-san` 标志，此外还应针对 K3s 禁用 `servicelb` LB。

要更改 RKE2 或 K3s 服务器的标志，需要修改群集所有 VM 上的 `/etc/systemd/system/rke2.service` 或 `/etc/systemd/system/k3s.service` 文件，具体取决于发行版。

应在 `ExecStart` 中插入标志。例如：

对于 RKE2，使用以下命令：

```
# Replace the <vip-service-ip> with the actual ip
ExecStart=/usr/local/bin/rke2 \
    server \
        '--write-kubeconfig-mode=644' \
        '--tls-san=<vip-service-ip>' \
        '--tls-san=https://<vip-service-ip>.sslip.io' \
```

对于 K3s，使用以下命令：

```
# Replace the <vip-service-ip> with the actual ip
ExecStart=/usr/local/bin/k3s \
    server \
        '--cluster-init' \
        '--write-kubeconfig-mode=644' \
        '--disable=servicelb' \
        '--tls-san=<vip-service-ip>' \
        '--tls-san=https://<vip-service-ip>.sslip.io' \
```

然后应执行以下命令来加载新配置：

```
systemctl daemon-reload
systemctl restart ${KUBE_DISTRIBUTION}
```

26.4 安装 MetalLB

要部署 MetalLB，可以使用 K3s 上的 MetalLB（第 25 章 “K3s 上的 MetalLB（使用第 2 层模式）”）指南。

注意：确保 ip-pool IPAddressPool 的 IP 地址与先前为 LoadBalancer 服务选择的 IP 地址不重叠。

单独创建一个仅供受管服务使用的 IpAddressPool。

```
# Export the VIP_SERVICE_IP on the local machine
# Replace with the actual IP
export VIP_SERVICE_IP=<ip>

cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: kubernetes-vip-ip-pool
  namespace: metallb-system
spec:
  addresses:
  - ${VIP_SERVICE_IP}/32
  serviceAllocation:
    priority: 100
    namespaces:
      - default
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - ip-pool
  - kubernetes-vip-ip-pool
```

```
EOF
```

26.5 安装 Endpoint Copier Operator

```
helm install \
endpoint-copier-operator oci://registry.suse.com/edge/charts/endpoint-copier-
operator \
--namespace endpoint-copier-operator \
--create-namespace
```

以上命令会部署包含两个复本的 `endpoint-copier-operator` 操作器部署。其中一个复本是领导者，另一个复本在需要时接管领导者角色。

现在应部署 `kubernetes-vip` 服务，这将由操作器来协调，另外需创建具有所配置端口和 IP 的端点。

对于 RKE2，使用以下命令：

```
cat <<-EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  ports:
  - name: rke2-api
    port: 9345
    protocol: TCP
    targetPort: 9345
  - name: k8s-api
    port: 6443
    protocol: TCP
    targetPort: 6443
  type: LoadBalancer
EOF
```

对于 K3s，使用以下命令：

```
cat <<-EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: https
    port: 6443
    protocol: TCP
    targetPort: 6443
  sessionAffinity: None
  type: LoadBalancer
EOF
```

校验 `kubernetes-vip` 服务是否使用正确的 IP 地址：

```
kubectl get service kubernetes-vip -n default \
 -o=jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

确保 `default` 名称空间中的 `kubernetes-vip` 和 `kubernetes` 端点资源指向相同的 IP。

```
kubectl get endpoints kubernetes kubernetes-vip
```

如果所有设置正确，剩下的最后一项操作就是在 `Kubeconfig` 中使用 `VIP_SERVICE_IP`。

```
sed -i '' "s/${NODE_IP}/${VIP_SERVICE_IP}/g" ~/.kube/config
```

从现在开始，所有 `kubectl` 命令都将通过 `kubernetes-vip` 服务运行。

26.6 添加控制平面节点

要监控整个过程，可以打开另外两个终端选项卡。

第一个终端：

```
watch kubectl get nodes
```

第二个终端：

```
watch kubectl get endpoints
```

现在，在第二和第三个节点上执行以下命令。

对于 RKE2，使用以下命令：

```
# Export the VIP_SERVICE_IP in the VM
# Replace with the actual IP
export VIP_SERVICE_IP=<ip>

curl -sfL https://get.rke2.io | INSTALL_RKE2_TYPE="server" sh -
systemctl enable rke2-server.service


mkdir -p /etc/rancher/rke2/
cat <<EOF > /etc/rancher/rke2/config.yaml
server: https://${VIP_SERVICE_IP}:9345
token: ${RKE2_TOKEN}
EOF

systemctl start rke2-server.service
```

对于 K3s，使用以下命令：

```
# Export the VIP_SERVICE_IP in the VM
# Replace with the actual IP
export VIP_SERVICE_IP=<ip>
export INSTALL_K3S_SKIP_START=false

curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server \
 --server https://${VIP_SERVICE_IP}:6443 --disable=servicelb \
 --write-kubeconfig-mode=644" K3S_TOKEN=foobar sh -
```

27 使用 Edge Image Builder 进行隔离式部署

27.1 简介

本指南将介绍如何使用 Edge Image Builder (EIB)（第 11 章 “Edge Image Builder”）在 SUSE Linux Micro 6.1 上以完全隔离的方式部署多个 SUSE Edge 组件。使用此方法可以引导至 EIB 所创建的随时可引导 (CRB) 的自定义映像，并在 RKE2 或 K3s 群集上部署指定的组件，而无需连接到互联网，也无需执行任何手动步骤。对于想要将部署所需的所有制品预先嵌入其操作系统映像的客户而言，此配置非常理想，这样就可以在引导时立即使用这些制品。

本指南将介绍以下组件的隔离式安装：

- 第 5 章 “Rancher”
- 第 18 章 “SUSE Security”
- 第 17 章 “SUSE Storage”
- 第 21 章 “Edge Virtualization”

警告

EIB 将分析并预先下载提供的 Helm chart 和 Kubernetes 清单中引用的所有映像。但是，其中一些操作可能会尝试提取容器映像并在运行时基于这些映像创建 Kubernetes 资源。在这种情况下，如果我们想要设置完全隔离的环境，则必须在定义文件中手动指定所需的映像。

27.2 先决条件

我们假设本指南的读者已事先熟悉 EIB（第 11 章 “Edge Image Builder”）。如果您不熟悉，请阅读快速入门指南（第 3 章 “使用 Edge Image Builder 配置独立群集”），以便更好地理解下面实际操作中涉及的概念。

27.3 Libvirt 网络配置

注意

为了演示隔离式部署，本指南将使用模拟的 `libvirt` 隔离网络，并根据该网络定制以下配置。对于您自己的部署，可能需要修改下一步骤中将介绍的 `host1.local.yaml` 配置。

如果您要使用相同的 `libvirt` 网络配置，请继续阅读。否则请跳到第 27.4 节 “基础目录配置”。

我们来为 DHCP 创建 IP 地址范围为 `192.168.100.2/24` 的隔离网络配置：

```
cat << EOF > isolatednetwork.xml
<network>
  <name>isolatednetwork</name>
  <bridge name='virbr1' stp='on' delay='0'/>
  <ip address='192.168.100.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.100.2' end='192.168.100.254'/>
    </dhcp>
  </ip>
</network>
EOF
```

现在，唯一剩下的操作就是创建并启动网络：

```
virsh net-define isolatednetwork.xml
virsh net-start isolatednetwork
```

27.4 基础目录配置

基础目录配置在所有组件中都相同，现在我们就设置此配置。

首先创建所需的子目录：

```
export CONFIG_DIR=$HOME/config
```

```
mkdir -p $CONFIG_DIR/base-images
mkdir -p $CONFIG_DIR/network
mkdir -p $CONFIG_DIR/kubernetes/helm/values
```

请确保将您要使用的任何基础映像添加到 `base-images` 目录中。本指南将重点介绍此处 (https://www.suse.com/download/sle-micro/) 提供的自安装 ISO 映像。

我们来复制已下载的映像：

```
cp SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso $CONFIG_DIR/base-images/
slemicro.iso
```

注意

EIB 永远不会修改基础映像输入。

我们来创建一个包含所需网络配置的文件：

```
cat << EOF > $CONFIG_DIR/network/host1.local.yaml
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: 192.168.100.1
    next-hop-interface: eth0
    table-id: 254
  - destination: 192.168.100.0/24
    metric: 100
    next-hop-address:
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - 192.168.100.1
    - 8.8.8.8
interfaces:
- name: eth0
```

```
  type: ethernet
  state: up
  mac-address: 34:8A:B1:4B:16:E7
  ipv4:
    address:
    - ip: 192.168.100.50
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
EOF
```

此配置确保置备的系统上存在以下设置（使用指定的 MAC 地址）：

- 采用静态 IP 地址的以太网接口
- 路由
- DNS
- 主机名 (`host1.local`)

最终的文件结构现在应如下所示：

```
├── kubernetes/
│   └── helm/
│       └── values/
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

27.5 基础定义文件

Edge Image Builder 使用**定义文件**来修改 SUSE Linux Micro 映像。这些文件包含大部分可配置选项。其中的许多选项将在不同的组件部分中重复出现，因此下面列出并解释了这些选项。

提示

定义文件中自定义选项的完整列表可以在上游文档 (https://github.com/suse-edge/edge-image-builder/blob/release-1.1/docs/building-images.md#image-definition-file) 中找到

我们来看看所有定义文件中的以下字段：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.32.4+rke2r1
embeddedArtifactRegistry:
  images:
    - ...
```

`image` 部分是必需的，用于指定输入映像、输入映像的体系结构和类型，以及输出映像的名称。

`operatingSystem` 部分是可选的，其中包含的配置允许用户通过 `root/eib` 用户名/口令登录到置备的系统。

`kubernetes` 部分是可选的，用于定义 Kubernetes 类型和版本。我们将使用 RKE2 发行版。如果需要 K3s，请改用 `kubernetes.version: v1.32.4+k3s1`。除非通过 `kubernetes.nodes` 字段明确配置，否则本指南中引导的所有群集都是单节点群集。

`embeddedArtifactRegistry` 部分包含仅在运行时为特定组件引用和提取的所有映像。

27.6 Rancher 安装

注意

为便于演示，我们将大幅精简演示用的 Rancher（第 5 章 “Rancher”）部署。对于实际部署，可能需要根据您的配置添加其他制品。

Rancher 2.11.2 (https://github.com/rancher/rancher/releases/tag/v2.11.2) 版本资产包含 `rancher-images.txt` 文件，其中列出了隔离式安装所需的所有映像。

总共有超过 600 个容器映像，这意味着生成的 CRB 映像的大小约为 30GB。对于我们的 Rancher 安装，我们将精简该列表，使之与最小有效配置相当。您可以在该列表中重新添加部署所需的任何映像。

创建定义文件并在其中包含精简的映像列表：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.32.4+rke2r1
  manifests:
    urls:
    - https://github.com/cert-manager/cert-manager/releases/download/v1.15.3/
cert-manager.crds.yaml
  helm:
    charts:
      - name: rancher
        version: 2.11.2
```

```
        repositoryName: rancher-prime
        valuesFile: rancher-values.yaml
        targetNamespace: cattle-system
        createNamespace: true
        installationNamespace: kube-system
      - name: cert-manager
        installationNamespace: kube-system
        createNamespace: true
        repositoryName: jetstack
        targetNamespace: cert-manager
        version: 1.15.3
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
embeddedArtifactRegistry:
  images:
    - name: registry.rancher.com/rancher/backup-restore-operator:v7.0.1
    - name: registry.rancher.com/rancher/calico-cni:v3.29.0-rancher1
    - name: registry.rancher.com/rancher/cis-operator:v1.4.0
    - name: registry.rancher.com/rancher/flannel-cni:v1.4.1-rancher1
    - name: registry.rancher.com/rancher/fleet-agent:v0.12.2
    - name: registry.rancher.com/rancher/fleet:v0.12.2
    - name: registry.rancher.com/rancher/hardened-addon-resizer:1.8.22-
build20250110
    - name: registry.rancher.com/rancher/hardened-calico:v3.29.2-build20250306
    - name: registry.rancher.com/rancher/hardened-cluster-autoscaler:v1.9.0-
build20241126
    - name: registry.rancher.com/rancher/hardened-cni-plugins:v1.6.2-
build20250306
    - name: registry.rancher.com/rancher/hardened-coredns:v1.12.0-build20241126
    - name: registry.rancher.com/rancher/hardened-dns-node-cache:1.24.0-
build20241211
    - name: registry.rancher.com/rancher/hardened-etcd:v3.5.19-k3s1-
build20250306
    - name: registry.rancher.com/rancher/hardened-flannel:v0.26.5-build20250306
```

```
    - name: registry.rancher.com/rancher/hardened-k8s-metrics-server:v0.7.2-
build20250110
    - name: registry.rancher.com/rancher/hardened-kubernetes:v1.32.3-rke2r1-
build20250312
    - name: registry.rancher.com/rancher/hardened-multus-cni:v4.1.4-
build20250108
    - name: registry.rancher.com/rancher/hardened-whereabouts:v0.8.0-
build20250131
    - name: registry.rancher.com/rancher/k3s-upgrade:v1.32.3-k3s1
    - name: registry.rancher.com/rancher/klipper-helm:v0.9.4-build20250113
    - name: registry.rancher.com/rancher/klipper-lb:v0.4.13
    - name: registry.rancher.com/rancher/kube-api-auth:v0.2.4
    - name: registry.rancher.com/rancher/kubectl:v1.32.2
    - name: registry.rancher.com/rancher/kuberlr-kubectl:v4.0.2
    - name: registry.rancher.com/rancher/local-path-provisioner:v0.0.31
    - name: registry.rancher.com/rancher/machine:v0.15.0-rancher125
    - name: registry.rancher.com/rancher/mirrored-cluster-api-controller:v1.9.5
    - name: registry.rancher.com/rancher/nginx-ingress-controller:v1.12.1-
hardened1
    - name: registry.rancher.com/rancher/prom-prometheus:v2.55.1
    - name: registry.rancher.com/rancher/prometheus-federator:v3.0.1
    - name: registry.rancher.com/rancher/pushprox-client:v0.1.4-rancher2-client
    - name: registry.rancher.com/rancher/pushprox-proxy:v0.1.4-rancher2-proxy
    - name: registry.rancher.com/rancher/rancher-agent:v2.11.1
    - name: registry.rancher.com/rancher/rancher-csp-adapter:v6.0.0
    - name: registry.rancher.com/rancher/rancher-webhook:v0.7.1
    - name: registry.rancher.com/rancher/rancher:v2.11.1
    - name: registry.rancher.com/rancher/remotedialer-proxy:v0.4.4
    - name: registry.rancher.com/rancher/rke-tools:v0.1.111
    - name: registry.rancher.com/rancher/rke2-cloud-provider:v1.32.0-
rc3.0.20241220224140-68fbd1a6b543-build20250101
    - name: registry.rancher.com/rancher/rke2-runtime:v1.32.3-rke2r1
    - name: registry.rancher.com/rancher/rke2-upgrade:v1.32.3-rke2r1
    - name: registry.rancher.com/rancher/security-scan:v0.6.0
    - name: registry.rancher.com/rancher/shell:v0.4.0
    - name: registry.rancher.com/rancher/system-agent-installer-k3s:v1.32.3-k3s1
    - name: registry.rancher.com/rancher/system-agent-installer-rke2:v1.32.3-
rke2r1
```

```
    - name: registry.rancher.com/rancher/system-agent:v0.3.12-suc
    - name: registry.rancher.com/rancher/system-upgrade-controller:v0.15.2
    - name: registry.rancher.com/rancher/ui-plugin-catalog:4.0.1
    - name: registry.rancher.com/rancher/kubectl:v1.20.2
    - name: registry.rancher.com/rancher/shell:v0.1.24
    - name: registry.rancher.com/rancher/mirrored-ingress-nginx-kube-webhook-
certgen:v1.5.0
    - name: registry.rancher.com/rancher/mirrored-ingress-nginx-kube-webhook-
certgen:v1.5.2
```

与包含 600 多个容器映像的完整列表相比，此精简版本仅包含约 60 个容器映像，因此新 CRB 映像的大小只有大约 7GB。

我们还需要为 Rancher 创建 Helm 值文件：

```
cat << EOF > $CONFIG_DIR/kubernetes/helm/values/rancher-values.yaml
hostname: 192.168.100.50.sslip.io
replicas: 1
bootstrapPassword: "adminadminadmin"
systemDefaultRegistry: registry.rancher.com
useBundledSystemChart: true
EOF
```

警告

将 `systemDefaultRegistry` 设置为 `registry.rancher.com` 可让 Rancher 在引导时，在 CRB 映像内启动的嵌入式制品注册表中自动查找映像。省略此字段可能会导致无法在节点上找到容器映像。

我们来构建映像：

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file eib-iso-definition.yaml
```

输出应如下所示：

```
Downloading file: dl-manifest-1.yaml 100% |
                                                                                |
 (583/583 kB, 12 MB/s)
```

```
Pulling selected Helm charts... 100% |
████████████████████████████████████████████████████████████████████████████████|
 (2/2, 3 it/s)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
██████████████████████████████████████████████████████████████████|
 (56/56, 8 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% |
████████████████████████████████████████████████████| (644/644 MB, 29 MB/
s)
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% |
██████████████████████████████████████████████████| (400/400 MB, 29 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% |
██████████████████████████████████████████████████████████████████|
 (36/36 MB, 30 MB/s)
Downloading file: sha256sum-amd64.txt 100% |
████████████████████████████████████████████████████████████████████|
 (4.3/4.3 kB, 29 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
```

```
Cleanup ..................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

置备使用构建映像的节点后，可以校验 Rancher 安装：

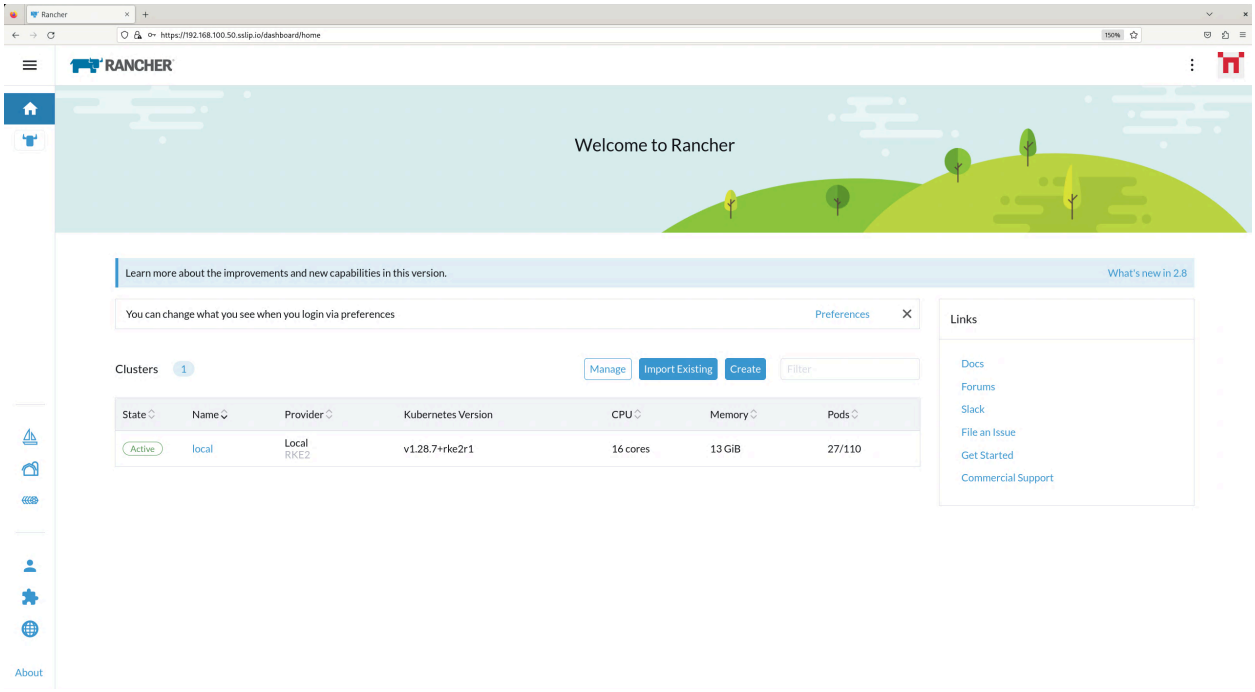
```
/var/lib/rancher/rke2/bin/kubectl get all -n cattle-system --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

输出应类似于以下内容，这表明已成功部署所有组件：

```
NAME                                          READY   STATUS      RESTARTS
 AGE
pod/helm-operation-6l6ld                      0/2     Completed   0
 107s
pod/helm-operation-8tk2v                      0/2     Completed   0
 2m2s
pod/helm-operation-blnrr                      0/2     Completed   0
 2m49s
pod/helm-operation-hdcmt                      0/2     Completed   0
 3m19s
pod/helm-operation-m74c7                      0/2     Completed   0
 97s
pod/helm-operation-qzzr4                      0/2     Completed   0
 2m30s
pod/helm-operation-s9jh5                      0/2     Completed   0
 3m
pod/helm-operation-tq7ts                      0/2     Completed   0
 2m41s
pod/rancher-99d599967-ftjkk                   1/1     Running     0
 4m15s
pod/rancher-webhook-79798674c5-6w28t          1/1     Running     0
 2m27s
pod/system-upgrade-controller-56696956b-trq5c   1/1     Running     0
 104s

NAME                      TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
    AGE
```

```
service/rancher           ClusterIP   10.43.255.80   <none>        80/TCP,443/
TCP   4m15s
service/rancher-webhook   ClusterIP   10.43.7.238    <none>        443/TCP
    2m27s

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/rancher                     1/1     1            1
 4m15s
deployment.apps/rancher-webhook             1/1     1            1
 2m27s
deployment.apps/system-upgrade-controller   1/1     1            1
 104s

NAME                                                  DESIRED   CURRENT   READY
  AGE
replicaset.apps/rancher-99d599967                     1         1         1
  4m15s
replicaset.apps/rancher-webhook-79798674c5            1         1         1
  2m27s
replicaset.apps/system-upgrade-controller-56696956b   1         1         1
  104s
```

当我们访问 `https://192.168.100.50.sslip.io` 并使用先前设置的 `adminadminadmin` 口令登录后，Rancher 仪表板即会显示：



27.7 SUSE Security 安装

与 Rancher 安装不同，SUSE Security 安装不需要在 EIB 中进行任何特殊处理。EIB 将自动隔离底层组件 NeuVector 所需的每个映像。

创建定义文件：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.32.4+rke2r1
```

```
  helm:
    charts:
      - name: neuvector-crd
        version: 106.0.1+up2.8.6
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector-values.yaml
      - name: neuvector
        version: 106.0.1+up2.8.6
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector-values.yaml
    repositories:
      - name: rancher-charts
        url: https://charts.rancher.io/
```

另外，为 NeuVector 创建 Helm 值文件：

```
cat << EOF > $CONFIG_DIR/kubernetes/helm/values/neuvector-values.yaml
controller:
  replicas: 1
manager:
  enabled: false
cve:
  scanner:
    enabled: false
    replicas: 1
k3s:
  enabled: true
crdwebhook:
  enabled: false
EOF
```

我们来构建映像：

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
```

```
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file eib-iso-definition.yaml
```

输出应如下所示：

```
Pulling selected Helm charts... 100% |
████████████████████████████████████████████████████████████████████████████████|
 (2/2, 4 it/s)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
████████████████████████████████████████████████████████████████████████████|
 (5/5, 13 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

置备使用构建映像的节点后，可以校验 SUSE Security 安装：

```
/var/lib/rancher/rke2/bin/kubectl get all -n neuvector --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

输出应类似于以下内容，这表明已成功部署所有组件：

```
NAME                                          READY   STATUS      RESTARTS
 AGE
pod/neuvector-cert-upgrader-job-bxbnz         0/1     Completed   0
 3m39s
pod/neuvector-controller-pod-7d854bfdc7-nhxjf   1/1     Running     0
 3m44s
pod/neuvector-enforcer-pod-ct8jm              1/1     Running     0
 3m44s

NAME                                     TYPE        CLUSTER-IP      EXTERNAL-
IP   PORT(S)                          AGE
service/neuvector-svc-admission-webhook   ClusterIP   10.43.234.241   <none>
    443/TCP                          3m44s
service/neuvector-svc-controller          ClusterIP   None            <none>
    18300/TCP,18301/TCP,18301/UDP   3m44s
service/neuvector-svc-crd-webhook         ClusterIP   10.43.50.190    <none>
    443/TCP                          3m44s

NAME                                    DESIRED   CURRENT   READY   UP-TO-DATE
 AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/neuvector-enforcer-pod   1         1         1       1
 1           <none>          3m44s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/neuvector-controller-pod   1/1     1            1
 3m44s

NAME                                                  DESIRED   CURRENT   READY
  AGE
replicaset.apps/neuvector-controller-pod-7d854bfdc7   1         1         1
  3m44s

NAME                                        SCHEDULE    TIMEZONE   SUSPEND
 ACTIVE   LAST SCHEDULE   AGE
```

```
cronjob.batch/neuvector-cert-upgrader-pod   0 0 1 1 *   <none>     True      0
      <none>          3m44s
cronjob.batch/neuvector-updater-pod         0 0 * * *   <none>     False     0
      <none>          3m44s

NAME                                    STATUS     COMPLETIONS   DURATION   AGE
job.batch/neuvector-cert-upgrader-job   Complete   1/1           7s
 3m39s
```

27.8 SUSE Storage 安装

Longhorn 的官方文档 (https://longhorn.io/docs/1.8.1/deploy/install/airgap/) 包含 `longhorn-images.txt` 文件，其中列出了隔离式安装所需的所有映像。我们将在定义文件中包含它们的 Rancher 容器注册表镜像副本。现在来创建此文件：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
  packages:
    sccRegistrationCode: [reg-code]
    packageList:
      - open-iscsi
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: longhorn
        repositoryName: longhorn
```

```
        targetNamespace: longhorn-system
        createNamespace: true
        version: 106.2.0+up1.8.1
      - name: longhorn-crd
        repositoryName: longhorn
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
        version: 106.2.0+up1.8.1
    repositories:
      - name: longhorn
        url: https://charts.rancher.io
embeddedArtifactRegistry:
  images:
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-attacher:v4.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-provisioner:v5.2.0
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-resizer:v1.13.2
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-snapshotter:v8.2.0
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-node-driver-
registrar:v2.13.0
    - name: registry.suse.com/rancher/mirrored-longhornio-livenessprobe:v2.15.0
    - name: registry.suse.com/rancher/mirrored-longhornio-backing-image-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-engine:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-instance-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-share-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-ui:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-support-bundle-
kit:v0.0.52
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-cli:v1.8.1
```

注意

您会注意到，定义文件列出了 `open-iscsi` 软件包。该软件包是必需的组件，因为 Longhorn 依赖于不同节点上运行的 `iscsiadm` 守护程序来为 Kubernetes 提供持久性卷。

我们来构建映像：

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file eib-iso-definition.yaml
```

输出应如下所示：

```
Setting up Podman API listener...
Pulling selected Helm charts... 100% |
 (2/2, 3 it/s)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Resolving package dependencies...
Rpm .......................... [SUCCESS]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
 (15/15, 20956 it/s)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
```

```
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% (782/782 MB, 108 MB/
s)
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% (367/367 MB, 104
 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% (34/34 MB, 108 MB/s)
Downloading file: sha256sum-amd64.txt 100% (3.9/3.9 kB, 7.5 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

置备使用构建映像的节点后，可以校验 Longhorn 安装：

```
/var/lib/rancher/rke2/bin/kubectl get all -n longhorn-system --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

输出应类似于以下内容，这表明已成功部署所有组件：

```
NAME                                                READY   STATUS
 RESTARTS   AGE
pod/csi-attacher-787fd9c6c8-sf42d                   1/1     Running   0
     2m28s
pod/csi-attacher-787fd9c6c8-tb82p                   1/1     Running   0
     2m28s
pod/csi-attacher-787fd9c6c8-zhc6s                   1/1     Running   0
     2m28s
pod/csi-provisioner-74486b95c6-b2v9s                1/1     Running   0
     2m28s
pod/csi-provisioner-74486b95c6-hwllt                1/1     Running   0
     2m28s
pod/csi-provisioner-74486b95c6-mlrpk                1/1     Running   0
     2m28s
pod/csi-resizer-859d4557fd-t54zk                    1/1     Running   0
     2m28s
```

```
pod/csi-resizer-859d4557fd-vdt5d                          1/1     Running   0
     2m28s
pod/csi-resizer-859d4557fd-x9kh4                          1/1     Running   0
     2m28s
pod/csi-snapshotter-6f69c6c8cc-r62gr                      1/1     Running   0
     2m28s
pod/csi-snapshotter-6f69c6c8cc-vrwjn                      1/1     Running   0
     2m28s
pod/csi-snapshotter-6f69c6c8cc-z65nb                      1/1     Running   0
     2m28s
pod/engine-image-ei-4623b511-9vhkb                        1/1     Running   0
     3m13s
pod/instance-manager-6f95fd57d4a4cd0459e469d75a300552     1/1     Running   0
     2m43s
pod/longhorn-csi-plugin-gx98x                             3/3     Running   0
     2m28s
pod/longhorn-driver-deployer-55f9c88499-fbm6q             1/1     Running   0
     3m28s
pod/longhorn-manager-dpdp7                                2/2     Running   0
     3m28s
pod/longhorn-ui-59c85fcf94-gg5hq                          1/1     Running   0
     3m28s
pod/longhorn-ui-59c85fcf94-s49jc                          1/1     Running   0
     3m28s

NAME                                  TYPE        CLUSTER-IP     EXTERNAL-IP
 PORT(S)    AGE
service/longhorn-admission-webhook    ClusterIP   10.43.77.89    <none>
 9502/TCP   3m28s
service/longhorn-backend              ClusterIP   10.43.56.17    <none>
 9500/TCP   3m28s
service/longhorn-conversion-webhook   ClusterIP   10.43.54.73    <none>
 9501/TCP   3m28s
service/longhorn-frontend             ClusterIP   10.43.22.82    <none>
 80/TCP     3m28s
service/longhorn-recovery-backend     ClusterIP   10.43.45.143   <none>
 9503/TCP   3m28s
```

```
NAME                                      DESIRED   CURRENT   READY   UP-TO-DATE
   AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/engine-image-ei-4623b511   1         1         1       1
   1           <none>          3m13s
daemonset.apps/longhorn-csi-plugin        1         1         1       1
   1           <none>          2m28s
daemonset.apps/longhorn-manager           1         1         1       1
   1           <none>          3m28s

NAME                                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/csi-attacher               3/3     3            3
 2m28s
deployment.apps/csi-provisioner            3/3     3            3
 2m28s
deployment.apps/csi-resizer                3/3     3            3
 2m28s
deployment.apps/csi-snapshotter            3/3     3            3
 2m28s
deployment.apps/longhorn-driver-deployer   1/1     1            1
 3m28s
deployment.apps/longhorn-ui                2/2     2            2
 3m28s

NAME                                             DESIRED   CURRENT   READY
  AGE
replicaset.apps/csi-attacher-787fd9c6c8            3         3         3
  2m28s
replicaset.apps/csi-provisioner-74486b95c6         3         3         3
  2m28s
replicaset.apps/csi-resizer-859d4557fd             3         3         3
  2m28s
replicaset.apps/csi-snapshotter-6f69c6c8cc         3         3         3
  2m28s
replicaset.apps/longhorn-driver-deployer-55f9c88499   1         1         1
  3m28s
replicaset.apps/longhorn-ui-59c85fcf94             2         2         2
  3m28s
```

27.9 KubeVirt 和 CDI 安装

KubeVirt 和 CDI 的 Helm chart 只会安装各自的操作器。系统的其余组件将由操作器来部署，这意味着，我们必须在定义文件中包含所有必要的容器映像。我们来创建定义文件：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/
zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: kubevirt
        repositoryName: suse-edge
        version: 303.0.0+up0.5.0
        targetNamespace: kubevirt-system
        createNamespace: true
        installationNamespace: kube-system
      - name: cdi
        repositoryName: suse-edge
        version: 303.0.0+up0.5.0
        targetNamespace: cdi-system
        createNamespace: true
        installationNamespace: kube-system
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
embeddedArtifactRegistry:
  images:
```

```
    - name: registry.suse.com/suse/sles/15.6/cdi-uploadproxy:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/cdi-
uploadserver:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/cdi-apiserver:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/cdi-controller:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/cdi-importer:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/cdi-cloner:1.60.1-150600.3.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-api:1.3.1-150600.5.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-controller:1.3.1-150600.5.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-launcher:1.3.1-150600.5.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-handler:1.3.1-150600.5.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-exportproxy:1.3.1-150600.5.9.1
    - name: registry.suse.com/suse/sles/15.6/virt-
exportserver:1.3.1-150600.5.9.1
```

我们来构建映像：

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file eib-iso-definition.yaml
```

输出应如下所示：

```
Pulling selected Helm charts... 100% |
 (2/2, 48 it/min)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
```

```
Populating Embedded Artifact Registry... 100% |
 (15/15, 4 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ..................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Kubernetes ................. [SUCCESS]
Certificates ............... [SKIPPED]
Cleanup .................... [SKIPPED]
Building ISO image...
Kernel Params .............. [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

置备使用构建映像的节点后，可以校验 KubeVirt 和 CDI 的安装。

校验 KubeVirt：

```
/var/lib/rancher/rke2/bin/kubectl get all -n kubevirt-system --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

输出应类似于以下内容，这表明已成功部署所有组件：

```
NAME                                READY   STATUS    RESTARTS   AGE
pod/virt-api-59cb997648-mmt67       1/1     Running   0          2m34s
pod/virt-controller-69786b785-7cc96 1/1     Running   0          2m8s
pod/virt-controller-69786b785-wq2dz 1/1     Running   0          2m8s
pod/virt-handler-2l4dm              1/1     Running   0          2m8s
pod/virt-operator-7c444cff46-nps4l  1/1     Running   0          3m1s
pod/virt-operator-7c444cff46-r25xq  1/1     Running   0          3m1s

NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP
 PORT(S)   AGE
service/kubevirt-operator-webhook   ClusterIP   10.43.167.109   <none>
 443/TCP   2m36s
service/kubevirt-prometheus-metrics ClusterIP   None            <none>
 443/TCP   2m36s
service/virt-api                    ClusterIP   10.43.18.202    <none>
 443/TCP   2m36s
```

```
service/virt-exportproxy              ClusterIP   10.43.142.188   <none>
 443/TCP   2m36s

NAME                           DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
   NODE SELECTOR            AGE
daemonset.apps/virt-handler   1         1         1       1            1
   kubernetes.io/os=linux   2m8s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/virt-api            1/1     1            1           2m34s
deployment.apps/virt-controller     2/2     2            2           2m8s
deployment.apps/virt-operator       2/2     2            2           3m1s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/virt-api-59cb997648           1         1         1       2m34s
replicaset.apps/virt-controller-69786b785     2         2         2       2m8s
replicaset.apps/virt-operator-7c444cff46      2         2         2       3m1s

NAME                              AGE    PHASE
kubevirt.kubevirt.io/kubevirt     3m1s   Deployed
```

校验 CDI：

```
/var/lib/rancher/rke2/bin/kubectl get all -n cdi-system --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

输出应类似于以下内容，这表明已成功部署所有组件：

```
NAME                                    READY   STATUS    RESTARTS   AGE
pod/cdi-apiserver-5598c9bf47-pqfxw      1/1     Running   0          3m44s
pod/cdi-deployment-7cbc5db7f8-g46z7     1/1     Running   0          3m44s
pod/cdi-operator-777c865745-2qcnj       1/1     Running   0          3m48s
pod/cdi-uploadproxy-646f4cd7f7-fzkv7    1/1     Running   0          3m44s

NAME                              TYPE        CLUSTER-IP     EXTERNAL-IP
 PORT(S)    AGE
service/cdi-api                   ClusterIP   10.43.2.224    <none>        443/
TCP    3m44s
service/cdi-prometheus-metrics    ClusterIP   10.43.237.13   <none>        8080/
TCP    3m44s
```

```
service/cdi-uploadproxy          ClusterIP   10.43.114.91   <none>        443/
TCP    3m44s

NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cdi-apiserver        1/1     1            1           3m44s
deployment.apps/cdi-deployment       1/1     1            1           3m44s
deployment.apps/cdi-operator         1/1     1            1           3m48s
deployment.apps/cdi-uploadproxy      1/1     1            1           3m44s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/cdi-apiserver-5598c9bf47       1         1         1       3m44s
replicaset.apps/cdi-deployment-7cbc5db7f8      1         1         1       3m44s
replicaset.apps/cdi-operator-777c865745        1         1         1       3m48s
replicaset.apps/cdi-uploadproxy-646f4cd7f7     1         1         1       3m44s
```

27.10 查错

如果您在构建映像时遇到任何问题，或者想要进一步测试和调试该过程，请参见上游文档 (https://github.com/suse-edge/edge-image-builder/tree/release-1.1/docs)。

28 使用 Kiwi 构建更新的 SUSE Linux Micro 映像

本章将说明如何生成更新的 SUSE Linux Micro 映像，这些映像可用于 Edge Image Builder、Cluster API (CAPI) + Metal[3]，或直接将磁盘映像写入块设备。此流程适用于需要在初始系统引导映像中包含最新补丁（以减少安装后的补丁传输量）的场景，或使用 CAPI 的场景，在该场景中，更适合使用新映像重新安装操作系统，而非就地升级主机。

该流程借助 Kiwi (https://osinside.github.io/kiwi/) 来执行映像构建。SUSE Edge 随附容器化版本，其中内置了实用辅助工具，可简化整体流程，并支持指定所需的目标**配置文件**。配置文件定义了所需输出映像的类型，常见类型如下：

- “**Base**” - SUSE Linux Micro 磁盘映像，包含精简的软件包集合（含 podman）。
- “**Base-SelfInstall**” - 基于上述“Base”的自安装映像。
- “**Base-RT**” - 与上述“Base”相同，但使用实时 (rt) 内核。
- “**Base-RT-SelfInstall**” - 基于上述“Base-RT”的自安装映像。
- “**Default**” - 基于上述“Base”的 SUSE Linux Micro 磁盘映像，另外还包含其他工具，如虚拟化堆栈、Cockpit 和 salt-minion。
- “**Default-SelfInstall**” - 基于上述“Default”的自安装映像。

有关详细信息，请参见 SUSE Linux Micro 6.1 (https://documentation.suse.com/sle-micro/6.1/html/Micro-deployment-images/index.html#alp-images-installer-type) 文档。

此过程适用于 AMD64/Intel 64 和 AArch64 两种体系结构，但并非所有映像配置文件都同时支持这两种体系结构。例如，在使用 SUSE Linux Micro 6.1 的 SUSE Edge 3.3 中，含实时内核的配置文件（即“Base-RT”或“Base-RT-SelfInstall”）目前暂不支持 AArch64 体系结构。

注意

构建主机的体系结构必须与待构建映像的体系结构一致。也就是说，要构建 AArch64 体系结构的映像，必须使用 AArch64 体系结构的构建主机；AMD64/Intel 64 体系结构同样如此 - 目前不支持跨体系结构构建。

28.1 先决条件

Kiwi 映像构建器的要求如下：

- 一台体系结构与待构建映像体系结构相同的 SUSE Linux Micro 6.1 主机（即“构建系统”）。
- 构建系统需已通过 SUSEConnect 完成注册（注册是为了从 SUSE 储存库提取最新软件包）。
- 能够连接互联网以提取所需软件包；如果通过代理连接，需预先配置构建主机的代理设置。
- 构建主机上需禁用 SELinux（因为 SELinux 标签会在容器内生成，可能与主机策略冲突）。
- 至少需要 10GB 可用磁盘空间，以容纳容器映像、构建根目录及最终生成的输出映像。

28.2 入门指南

由于存在某些限制，目前需要禁用 SELinux。请连接到 SUSE Linux Micro 6.1 映像构建主机，并确保 SELinux 已禁用：

```
# setenforce 0
```

创建一个将与 Kiwi 构建容器共享的输出目录，用于保存生成的映像：

```
# mkdir ~/output
```

从 SUSE 注册表提取最新的 Kiwi 构建器映像：

```
# podman pull registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0
(...)
```

28.3 构建默认映像

如果运行容器映像时未提供任何参数，这就是 Kiwi 映像容器的默认行为。以下命令运行 podman 时会将两个目录映射到容器：

- 底层主机的 `/etc/zypp/repos.d` SUSE Linux Micro 软件包储存库目录。
- 上文创建的输出目录 `~/output`。

Kiwi 映像容器需要按如下方式运行 `build-image` 辅助脚本：

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/
tmp/output \
    -it registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0 build-image
(...)
```

注意

如果您是首次运行此脚本，脚本预计会在启动后不久**失败**，并显示“**ERROR: Early loop device test failed, please retry the container run.**”，这是由于底层主机系统上创建的循环设备无法立即在容器映像内可见所致。只需重新运行该命令，即可顺利执行。

几分钟后，即可在本地输出目录中找到映像：

```
(...)
INFO: Image build successful, generated images are available in the 'output'
 directory.

# ls -1 output/
SLE-Micro.x86_64-6.1.changes
SLE-Micro.x86_64-6.1.packages
SLE-Micro.x86_64-6.1.raw
SLE-Micro.x86_64-6.1.verified
build
kiwi.result
kiwi.result.json
```

28.4 使用其他配置文件构建映像

要构建不同的映像配置文件，需要使用 Kiwi 容器映像辅助脚本中的“**-p**”命令选项。例如，要构建“**Default-SelfInstall**”ISO 映像，请运行：

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/
tmp/output \
    -it registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0 build-image -p
 Default-SelfInstall
(...)
```

注意

如果 `output` 目录中存在映像，Kiwi 将拒绝运行，以免数据丢失。继续操作前，需要使用 `rm -f output/*` 命令去除输出目录中的内容。

或者，要构建包含实时内核（“**kernel-rt**”）的自安装 ISO 映像，请运行：

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/
tmp/output \
    -it registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0 build-image -p Base-
RT-SelfInstall
(...)
```

28.5 构建大扇区大小的映像

有些硬件要求映像采用大扇区大小（即 **4096 字节**，而非标准的 512 字节）。容器化 Kiwi 构建器支持通过指定“**-b**”参数生成大块大小的映像。例如，要构建大扇区大小的“**Default-SelfInstall**”映像，请运行：

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/
tmp/output \
    -it registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0 build-image -p
 Default-SelfInstall -b
(...)
```

28.6 创建自定义 Kiwi 映像定义文件

对于高级使用场景，可以使用自定义 Kiwi 映像定义文件 (`SL-Micro.kiwi`) 以及任何必要的构建后脚本。这需要覆盖 SUSE Edge 团队预先打包的默认定义。

创建一个新目录，并将其映射到容器映像中辅助脚本将查找定义文件的目录 (`/micro-sdk/defs`)：

```
# mkdir ~/mydefs/
# cp /path/to/SL-Micro.kiwi ~/mydefs/
# cp /path/to/config.sh ~/mydefs/
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/
tmp/output -v ~/mydefs/:/micro-sdk/defs/ \
    -it registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0 build-image
(...)
```

警告

此操作仅适用于高级使用场景，可能会导致可支持性问题。请联系您的 SUSE 代表以获取进一步建议和指导。

要获取容器中包含的默认 Kiwi 映像定义文件，可使用以下命令：

```
$ podman create --name kiwi-builder registry.suse.com/edge/3.3/kiwi-
builder:10.2.12.0
$ podman cp kiwi-builder:/micro-sdk/defs/SL-Micro.kiwi .
$ podman cp kiwi-builder:/micro-sdk/defs/SL-Micro.kiwi.4096 .
$ podman rm kiwi-builder
$ ls ./SL-Micro.*
(...)
```

29 使用 clusterclass 部署下游群集

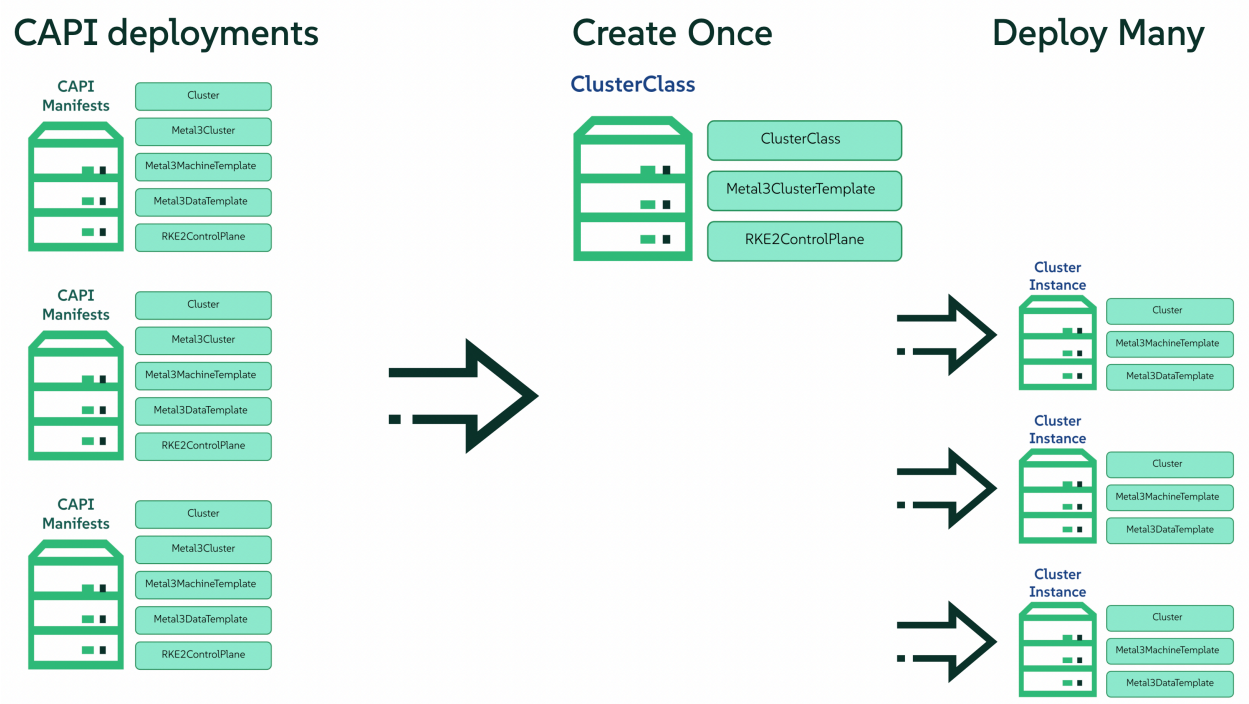
29.1 简介

置备 Kubernetes 群集是一项复杂的任务，需要具备深入的群集组件配置专业知识。随着配置变得愈发复杂，或者不同提供商的需求引入了大量提供商专用的资源定义，群集创建可能会让人望而生畏。值得庆幸的是，Kubernetes Cluster API (CAPI) 提供了一种更简洁的声明式方法，而 ClusterClass 进一步增强了这种方法。此功能引入了一种基于模板的模型，允许您定义可重用的群集类，该类封装了复杂性并提升了一致性。

29.2 什么是 ClusterClass?

CAPI 项目引入了 ClusterClass 功能，通过采用基于模板的群集实例化方法，实现了 Kubernetes 群集生命周期管理的范式转变。用户无需为每个群集独立定义资源，而是定义一个 ClusterClass，将它作为一个全面且可重用的蓝图。这种抽象表示封装了 Kubernetes 群集的期望状态和配置，让您能快速、一致地创建多个符合已定义规范的群集。这种抽象化减轻了配置负担，使部署清单更易于管理。这意味着工作负载群集的核心组件在类级别定义，这样用户便可将这些模板用作 Kubernetes 群集版本，可多次重用以置备群集。ClusterClass 的实现提供了多项关键优势，解决了传统 CAPI 大规模管理所固有的挑战：

- 大幅降低复杂性和 YAML 冗余度
- 优化维护和更新过程
- 增强部署间的一致性和标准化
- 提高可扩展性和自动化能力
- 声明式管理和强大的版本控制



29.3 当前 CAPI 置备文件示例

利用 Cluster API (CAPI) 和 RKE2 提供程序部署 Kubernetes 群集需要定义多个自定义资源。这些资源定义了群集及其底层基础架构的期望状态，使 CAPI 能够编排置备和管理生命周期。下面的代码段说明了必须配置的资源类型：

- **群集**：此资源封装了总体配置，包括将管理节点间通讯和服务发现的网络拓扑。此外，它还建立了与控制平面规范和指定的基础架构提供程序资源的必要关联，从而告知 CAPI 期望的群集体系结构及将用于置备群集的底层基础架构。
- **Metal3Cluster**：此资源定义 Metal3 特有的基础架构级属性，例如 Kubernetes API 服务器可通过其访问的外部端点。
- **RKE2ControlPlane**：此资源定义群集控制平面节点的特性和行为。此规范中配置了期望的控制平面复本数量（对确保高可用性和容错能力至关重要）、特定的 Kubernetes 发行版版本（与所选的 RKE2 版本一致），以及控制平面组件的滚动更新策略等参数。此外，此资源规定了将在群集中采用的容器网络接口 (CNI)，便于注入特定代理的配置，通常利用 Ignition 来无缝自动完成控制平面节点的 RKE2 代理置备。

- **Metal3MachineTemplate**：此资源充当将构成 Kubernetes 群集工作节点的各计算实例的创建蓝图，定义了要使用的映像。
- **Metal3DataTemplate**：此资源作为 Metal3MachineTemplate 的补充，可用于为新置备的计算机实例指定其他元数据。

```
---
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: RKE2ControlPlane
    name: emea-spa-cluster-3
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3Cluster
    name: emea-spa-cluster-3
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  controlPlaneEndpoint:
    host: 192.168.122.203
    port: 6443
  noCloudProvider: true
```

```
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: emea-spa-cluster-3
  replicas: 1
  version: v1.32.4+rke2r1
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 1
  registrationMethod: "control-plane-endpoint"
  registrationAddress: 192.168.122.203
  serverConfig:
    cni: cilium
    cniMultusEnable: true
    tlsSan:
      - 192.168.122.203
      - https://192.168.122.203.sslip.io
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        storage:
          files:
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-
operator.yaml
              overwrite: true
              contents:
                inline: |
```

```
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: endpoint-copier-operator
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/endpoint-copier-
operator
                    targetNamespace: endpoint-copier-operator
                    version: 303.0.0+up0.2.1
                    createNamespace: true
            - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: metallb
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/metallb
                    targetNamespace: metallb-system
                    version: 303.0.0+up0.14.9
                    createNamespace: true

            - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: metallb.io/v1beta1
                  kind: IPAddressPool
                  metadata:
                    name: kubernetes-vip-ip-pool
                    namespace: metallb-system
                  spec:
                    addresses:
                      - 192.168.122.203/32
```

```
                    serviceAllocation:
                      priority: 100
                      namespaces:
                        - default
                      serviceSelectors:
                        - matchExpressions:
                          - {key: "serviceType", operator: In, values:
[kubernetes-vip]}
                  ---
                  apiVersion: metallb.io/v1beta1
                  kind: L2Advertisement
                  metadata:
                    name: ip-pool-l2-adv
                    namespace: metallb-system
                  spec:
                    ipAddressPools:
                      - kubernetes-vip-ip-pool
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: v1
                  kind: Service
                  metadata:
                    name: kubernetes-vip
                    namespace: default
                    labels:
                      serviceType: kubernetes-vip
                  spec:
                    ports:
                    - name: rke2-api
                      port: 9345
                      protocol: TCP
                      targetPort: 9345
                    - name: k8s-api
                      port: 6443
                      protocol: TCP
                      targetPort: 6443
```

```
                    type: LoadBalancer
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  nodeReuse: True
  template:
    spec:
      automatedCleaningMode: metadata
      dataTemplate:
```

```
        name: emea-spa-cluster-3
      hostSelector:
        matchLabels:
          cluster-role: control-plane
          deploy-region: emea-spa
          node: group-3
      image:
        checksum: http://fileserver.local:8080/eibimage-downstream-
cluster.raw.sha256
        checksumType: sha256
        format: raw
        url: http://fileserver.local:8080/eibimage-downstream-cluster.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  clusterName: emea-spa-cluster-3
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
      - key: local_hostname
        object: machine
```

29.4 将 CAPI 置备文件转换为 ClusterClass

29.4.1 ClusterClass 定义

以下代码定义了一个 ClusterClass 资源，这是一个用于采用一致方式部署特定 Kubernetes 群集类型的声明式模板。该规范包含通用的基础架构和控制平面配置，可实现群集 Fleet 的高效置备和统一生命周期管理。下面的 clusterclass 示例中包含一些变量，这些变量将在群集实例化过程中被实际值替换。示例中使用的变量如下：

- `controlPlaneMachineTemplate`：这是用于定义要使用的控制平面计算机模板引用的名称
- `controlPlaneEndpointHost`：这是控制平面端点的主机名或 IP 地址
- `tlsSan`：这是控制平面端点的 TLS 主题备用名称

clusterclass 定义文件基于以下 3 种资源定义：

- **ClusterClass**：此资源封装了整个群集类定义，包括控制平面和基础架构模板。此外，它还包含在实例化过程中将被替换的变量列表。
- **RKE2ControlPlaneTemplate**：此资源定义控制平面模板，指定期望的控制平面节点配置。它包含复本数量、Kubernetes 版本、要使用的 CNI 等参数。此外，一些参数在实例化过程中将被替换为适当的值。
- **Metal3ClusterTemplate**：此资源定义基础架构模板，指定期望的底层基础架构配置。它包含控制平面端点、noCloudProvider 标志等参数。此外，一些参数在实例化过程中将被替换为适当的值。

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlaneTemplate
metadata:
  name: example-controlplane-type2
  namespace: emea-spa
spec:
  template:
    spec:
      infrastructureRef:
```

```
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
        kind: Metal3MachineTemplate
        name: example-controlplane    # This will be replaced by the patch
 applied in each cluster instances
        namespace: emea-spa
      replicas: 1
      version: v1.32.4+rke2r1
      rolloutStrategy:
        type: "RollingUpdate"
        rollingUpdate:
          maxSurge: 1
      registrationMethod: "control-plane-endpoint"
      registrationAddress: "default"  # This will be replaced by the patch
 applied in each cluster instances
      serverConfig:
        cni: cilium
        cniMultusEnable: true
        tlsSan:
          - "default"  # This will be replaced by the patch applied in each
 cluster instances
      agentConfig:
        format: ignition
        additionalUserData:
          config: |
            default
        kubelet:
          extraArgs:
            - provider-id=metal3://BAREMETALHOST_UUID
        nodeName: "localhost.localdomain"
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3ClusterTemplate
metadata:
  name: example-cluster-template-type2
  namespace: emea-spa
spec:
  template:
    spec:
```

```
      controlPlaneEndpoint:
        host: "default"  # This will be replaced by the patch applied in each
 cluster instances
        port: 6443
      noCloudProvider: true
---
apiVersion: cluster.x-k8s.io/v1beta1
kind: ClusterClass
metadata:
  name: example-clusterclass-type2
  namespace: emea-spa
spec:
  variables:
    - name: controlPlaneMachineTemplate
      required: true
      schema:
        openAPIV3Schema:
          type: string
    - name: controlPlaneEndpointHost
      required: true
      schema:
        openAPIV3Schema:
          type: string
    - name: tlsSan
      required: true
      schema:
        openAPIV3Schema:
          type: array
          items:
            type: string
  infrastructure:
    ref:
      kind: Metal3ClusterTemplate
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
      name: example-cluster-template-type2
  controlPlane:
    ref:
      kind: RKE2ControlPlaneTemplate
```

```
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    name: example-controlplane-type2
patches:
  - name: setControlPlaneMachineTemplate
    definitions:
      - selector:
          apiVersion: controlplane.cluster.x-k8s.io/v1beta1
          kind: RKE2ControlPlaneTemplate
          matchResources:
            controlPlane: true
        jsonPatches:
          - op: replace
            path: "/spec/template/spec/infrastructureRef/name"
            valueFrom:
              variable: controlPlaneMachineTemplate
  - name: setControlPlaneEndpoint
    definitions:
      - selector:
          apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
          kind: Metal3ClusterTemplate
          matchResources:
            infrastructureCluster: true  # Added to select InfraCluster
        jsonPatches:
          - op: replace
            path: "/spec/template/spec/controlPlaneEndpoint/host"
            valueFrom:
              variable: controlPlaneEndpointHost
  - name: setRegistrationAddress
    definitions:
      - selector:
          apiVersion: controlplane.cluster.x-k8s.io/v1beta1
          kind: RKE2ControlPlaneTemplate
          matchResources:
            controlPlane: true  # Added to select ControlPlane
        jsonPatches:
          - op: replace
            path: "/spec/template/spec/registrationAddress"
            valueFrom:
```

```
                variable: controlPlaneEndpointHost
    - name: setTlsSan
      definitions:
        - selector:
            apiVersion: controlplane.cluster.x-k8s.io/v1beta1
            kind: RKE2ControlPlaneTemplate
            matchResources:
              controlPlane: true  # Added to select ControlPlane
          jsonPatches:
            - op: replace
              path: "/spec/template/spec/serverConfig/tlsSan"
              valueFrom:
                variable: tlsSan
    - name: updateAdditionalUserData
      definitions:
        - selector:
            apiVersion: controlplane.cluster.x-k8s.io/v1beta1
            kind: RKE2ControlPlaneTemplate
            matchResources:
              controlPlane: true
          jsonPatches:
            - op: replace
              path: "/spec/template/spec/agentConfig/additionalUserData"
              valueFrom:
                template: |
                  config: |
                    variant: fcos
                    version: 1.4.0
                    storage:
                      files:
                        - path: /var/lib/rancher/rke2/server/manifests/endpoint-
copier-operator.yaml
                          overwrite: true
                          contents:
                            inline: |
                              apiVersion: helm.cattle.io/v1
                              kind: HelmChart
                              metadata:
```

```
                                name: endpoint-copier-operator
                                namespace: kube-system
                              spec:
                                chart: oci://registry.suse.com/edge/charts/
endpoint-copier-operator
                                targetNamespace: endpoint-copier-operator
                                version: 303.0.0+up0.2.1
                                createNamespace: true
                            - path: /var/lib/rancher/rke2/server/manifests/
metallb.yaml
                              overwrite: true
                              contents:
                                inline: |
                                  apiVersion: helm.cattle.io/v1
                                  kind: HelmChart
                                  metadata:
                                    name: metallb
                                    namespace: kube-system
                                  spec:
                                    chart: oci://registry.suse.com/edge/charts/
metallb
                                    targetNamespace: metallb-system
                                    version: 303.0.0+up0.14.9
                                    createNamespace: true
                            - path: /var/lib/rancher/rke2/server/manifests/metallb-
cr.yaml
                              overwrite: true
                              contents:
                                inline: |
                                  apiVersion: metallb.io/v1beta1
                                  kind: IPAddressPool
                                  metadata:
                                    name: kubernetes-vip-ip-pool
                                    namespace: metallb-system
                                  spec:
                                    addresses:
                                      - {{ .controlPlaneEndpointHost }}/32
                                    serviceAllocation:
```

```
                                    priority: 100
                                    namespaces:
                                      - default
                                    serviceSelectors:
                                      - matchExpressions:
                                        - {key: "serviceType", operator: In,
 values: [kubernetes-vip]}
                                ---
                                apiVersion: metallb.io/v1beta1
                                kind: L2Advertisement
                                metadata:
                                  name: ip-pool-l2-adv
                                  namespace: metallb-system
                                spec:
                                  ipAddressPools:
                                    - kubernetes-vip-ip-pool
                            - path: /var/lib/rancher/rke2/server/manifests/endpoint-
svc.yaml
                              overwrite: true
                              contents:
                                inline: |
                                  apiVersion: v1
                                  kind: Service
                                  metadata:
                                    name: kubernetes-vip
                                    namespace: default
                                    labels:
                                      serviceType: kubernetes-vip
                                  spec:
                                    ports:
                                    - name: rke2-api
                                      port: 9345
                                      protocol: TCP
                                      targetPort: 9345
                                    - name: k8s-api
                                      port: 6443
                                      protocol: TCP
                                      targetPort: 6443
```

```
                                type: LoadBalancer
                    systemd:
                      units:
                        - name: rke2-preinstall.service
                          enabled: true
                          contents: |
                            [Unit]
                            Description=rke2-preinstall
                            Wants=network-online.target
                            Before=rke2-install.service
                            ConditionPathExists=!/run/cluster-api/bootstrap-
success.complete
                            [Service]
                            Type=oneshot
                            User=root
                            ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                            ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/
$(jq -r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/
config.yaml"
                            ExecStart=/bin/sh -c "echo \"node-name: $(jq
 -r .name /mnt/openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/
config.yaml"
                            ExecStartPost=/bin/sh -c "umount /mnt"
                            [Install]
                            WantedBy=multi-user.target
```

29.4.2 群集实例定义

在 ClusterClass 的概念中，群集实例指的是基于定义的 ClusterClass 创建且已实例化的特定运行中群集。它代表一个具体的部署，基于 ClusterClass 中指定的蓝图直接衍生而成，具有独特的配置、资源和运行状态。这包括正在运行的一组特定计算机、网络配置以及相关的 Kubernetes 组件。要管理使用 ClusterClass 框架置备的特定已部署群集的生命周期、执行升级、进行扩容操作和实施监控，理解群集实例至关重要。

要定义一个群集实例，需要定义以下资源：

- 群集
- Metal3MachineTemplate
- Metal3DataTemplate

之前在模板（clusterclass 定义文件）中定义的变量将被替换为该群集实例化后的最终值：

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  topology:
    class: example-clusterclass-type2  # Correct way to reference ClusterClass
    version: v1.32.4+rke2r1
    controlPlane:
      replicas: 1
    variables:                         # Variables to be replaced for this
 cluster instance
      - name: controlPlaneMachineTemplate
        value: emea-spa-cluster-3-machinetemplate
      - name: controlPlaneEndpointHost
        value: 192.168.122.203
      - name: tlsSan
        value:
          - 192.168.122.203
          - https://192.168.122.203.sslip.io
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: emea-spa-cluster-3-machinetemplate
  namespace: emea-spa
spec:
  nodeReuse: True
  template:
    spec:
      automatedCleaningMode: metadata
```

```
      dataTemplate:
        name: emea-spa-cluster-3
      hostSelector:
        matchLabels:
          cluster-role: control-plane
          deploy-region: emea-spa
          cluster-type: type2
      image:
        checksum: http://fileserver.local:8080/eibimage-downstream-
cluster.raw.sha256
        checksumType: sha256
        format: raw
        url: http://fileserver.local:8080/eibimage-downstream-cluster.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  clusterName: emea-spa-cluster-3
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
```

这种方法使整个过程更加精简，一旦定义了 clusterclass，只需定义 3 种资源即可部署一个群集。

IV 提示和技巧

关于 Edge 组件的提示和技巧

30 Edge Image Builder

30.1 通用

- 如果您处于非 Linux 环境中，并且按照这些说明来构建映像，那么您可能是通过虚拟机运行 `Podman` 的。默认情况下配置的此类虚拟机只会分配到少量系统资源，这可能会在 `Edge Image Builder` 执行资源密集型操作（如 RPM 解析过程）时产生不稳定因素。您需要调整 Podman 计算机的资源，可通过 Podman Desktop（设置齿轮图标→Podman 计算机编辑图标）或直接使用 `podman-machine-set` 命令 (https://docs.podman.io/en/stable/markdown/podman-machine-set.1.html) 来完成。
- 目前，`Edge Image Builder` 无法在跨体系结构环境中构建映像，也就是说，您必须在适当系统中运行该工具来构建相应映像：
  - 在 AArch64 系统（如 Apple Silicon） 中构建 SL Micro `aarch64` 映像
  - 在 AMD64/Intel 64 系统中构建 SL Micro `x86_64`映像。

30.2 Kubernetes

- 如要创建多节点 Kubernetes 群集，则需要对定义文件中的 `kubernetes` 部分进行以下调整：

    - 在 `kubernetes.nodes` 下列出所有服务器节点和代理节点
    - 在 `kubernetes.network.apiVIP` 下设置一个虚拟 IP 地址，供所有非初始化节点加入群集时使用
    - （可选）在 `kubernetes.network.apiHost` 下设置一个 API 主机，以指定用于访问群集的域名地址。要了解有关此配置的详细信息，请参见介绍 Kubernetes 各部分的文档 (https://github.com/suse-edge/edge-image-builder/blob/main/docs/building-images.md#kubernetes)。
- `Edge Image Builder` 通过不同节点的主机名来确定它们的 Kubernetes 类型（服务器或代理）。虽然此配置在定义文件中进行管理，但对于计算机的一般网络设置，我们可以使用第 12 章 “边缘网络”中所述的 DHCP 配置。

31 Elemental

31.1 通用

31.1.1 公开 Rancher 服务

使用 RKE2 或 K3s 时，我们需要通过管理群集公开服务（此处指 Rancher），因为它们默认不会公开。RKE2 中通过 NGINX 入口控制器来实现此目的，K3s 使用的则是 Traefik。当前的工作流程建议使用 MetalLB 来公布服务（通过 L2 或 BGP 通告），并使用相应的入口控制器通过 `HelmChartConfig` 创建入口，因为创建新的入口对象时会覆盖现有的设置。

1. 安装 Rancher Prime（通过 Helm）并配置必要的值

```
hostname: rancher-192.168.64.101.sslip.io
replicas: 1
bootstrapPassword: Admin
global.cattle.psp.enabled: "false"
```

提示

有关详细信息，请参见 Rancher 安装 (https://ranchermanager.docs.rancher.com/v2.11/getting-started/installation-and-upgrade/install-upgrade-on-a-kubernetes-cluster) 文档。

2. 创建 LoadBalancer 服务以公开 Rancher

```
kubectl apply -f - <<EOF
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-ingress-nginx
  namespace: kube-system
spec:
  valuesContent: |-
```

```
    controller:
      config:
        use-forwarded-headers: "true"
        enable-real-ip: "true"
      publishService:
        enabled: true
      service:
        enabled: true
        type: LoadBalancer
        externalTrafficPolicy: Local
EOF
```

3. 使用我们之前在 Helm 值中设置的 IP 地址为该服务创建 IP 地址池

```
kubectl apply -f - <<EOF
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ingress-ippool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.64.101/32
  serviceAllocation:
    priority: 100
    serviceSelectors:
    - matchExpressions:
      - {key: app.kubernetes.io/name, operator: In, values: [rke2-ingress-
nginx]}
EOF
```

4. 为该 IP 地址池创建 L2 通告

```
kubectl apply -f - <<EOF
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ingress-l2-adv
  namespace: metallb-system
```

```
spec:
  ipAddressPools:
  - ingress-ippool
EOF
```

5. 确保 Elemental 已正确安装

   a. 在管理节点上安装 Elemental Operator 和 Elemental UI

   b. 在下游节点上添加 Elemental 配置以及注册代码，这样 Edge Image Builder 将会为计算机提供远程注册选项。

提示

有关更多信息和示例，请参见第 2.5 节 “安装 Elemental”和第 2.6 节 “配置 Elemental”。

31.2 硬件特定配置

31.2.1 可信平台模块

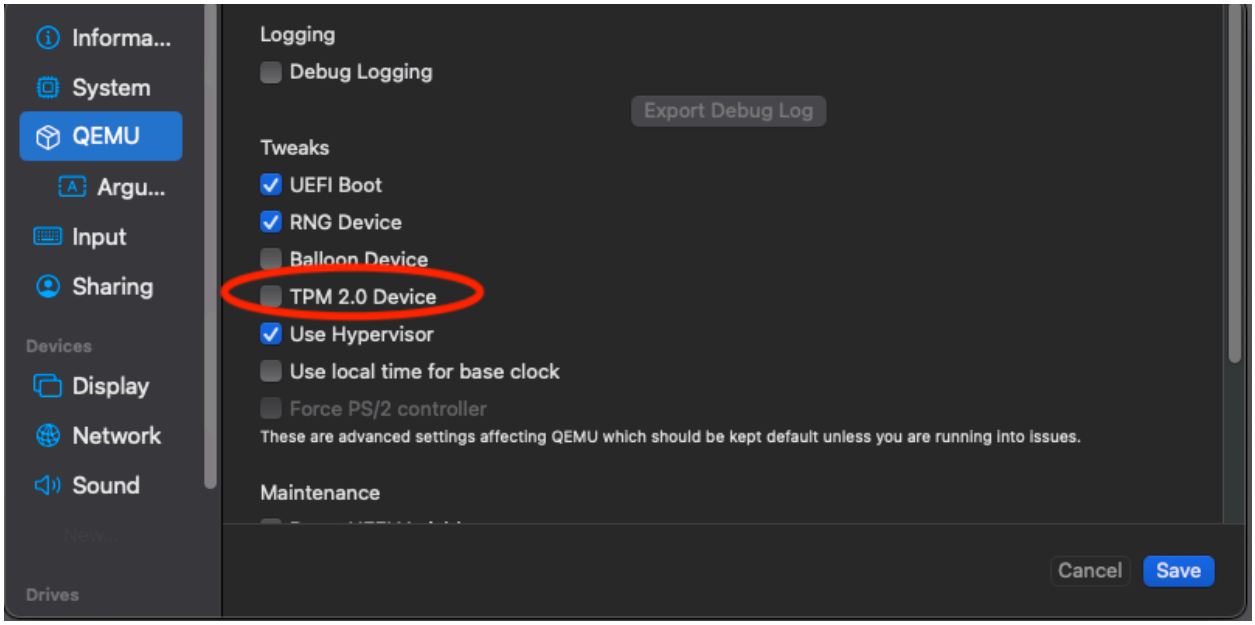
必须妥善处理可信平台模块 (https://elemental.docs.rancher.com/tpm/) (TPM) 配置，否则将导致如下所示的错误：

```
Nov 25 18:17:06 eled elemental-register[4038]: Error: registering machine:
 cannot generate authentication token: opening tpm for getting attestation data:
 TPM device not available
```

可通过以下方法之一缓解此问题：

- 在虚拟机设置中启用 TPM

MacOS 上的 UTM 示例



- 通过在 `MachineRegistration` 资源中为 TPM 种子使用负值来模拟 TPM

```
apiVersion: elemental.cattle.io/v1beta1
kind: MachineRegistration
metadata:
  name: ...
  namespace: ...
spec:
    ...
    elemental:
      ...
      registration:
        emulate-tpm: true
        emulated-tpm-seed: -1
```

- 在 `MachineRegistration` 资源中禁用 TPM

```
apiVersion: elemental.cattle.io/v1beta1
kind: MachineRegistration
metadata:
  name: ...
  namespace: ...
spec:
```

```
...
elemental:
  ...
  registration:
    emulate-tpm: false
```

V 第三方集成

如何集成第三方工具

32 NATS

NATS (https://nats.io/) 是为日益发展的超级互联世界而开发的连接技术。仅凭这一项技术，应用程序就能在云供应商、本地、边缘、Web 和移动设备的任意组合之间安全地通讯。NATS 由一系列开源产品组成，这些产品紧密集成，但可以轻松独立部署。NATS 已由全球数千家公司使用，涵盖微服务、边缘计算、移动通讯和 IoT 等使用场景，并可用于增强或取代传统的消息传递方式。

32.1 体系结构

NATS 是能够在应用程序之间以消息形式实现数据交换的基础架构。

32.1.1 NATS 客户端应用程序

应用程序可以使用 NATS 客户端库在不同的实例之间发布和订阅消息，以及发出请求和做出答复。这些应用程序通常称作客户端应用程序。

32.1.2 NATS 服务基础架构

NATS 服务由一个或多个 NATS 服务器进程提供，这些进程配置为彼此互连，提供了 NATS 服务基础架构。NATS 服务基础架构可以从一个终端设备上运行的单个 NATS 服务器进程，扩展为由许多群集组成的全球公用超级群集，这些群集跨越所有主要云提供商服务和全球所有区域。

32.1.3 简单的消息传递设计

NATS 使应用程序能够通过发送和接收消息来轻松进行通讯。这些消息按照主题字符串进行寻址和标识，并且不依赖于网络位置。数据经过编码，并构造为由发布者发送的消息。该消息由一个或多个订阅者接收、解码和处理。

32.1.4 NATS JetStream

NATS 内置了一套称为 JetStream 的分布式持久化系统，旨在解决当今流媒体技术存在的诸多问题 — 复杂性高、稳定性差以及可伸缩性不足。JetStream 还能解决发布者和订阅者之间的耦合问题（即订阅者必须处于正常运行状态，才能接收发布的消息）。有关 NATS JetStream 的详细信息，请参见此处 (https://docs.nats.io/nats-concepts/jetstream)。

32.2 安装

32.2.1 在 K3s 上安装 NATS

NATS 是为多种体系结构构建的，因此可以在 K3s 上轻松安装。（第 15 章 “K3s”）

我们创建一个值文件来重写 NATS 的默认值。

```
cat > values.yaml <<EOF
cluster:
  # Enable the HA setup of the NATS
  enabled: true
  replicas: 3

nats:
  jetstream:
    # Enable JetStream
    enabled: true

    memStorage:
      enabled: true
      size: 2Gi

    fileStorage:
      enabled: true
      size: 1Gi
      storageDirectory: /data/
EOF
```

现在我们需要通过 Helm 安装 NATS：

```
helm repo add nats https://nats-io.github.io/k8s/helm/charts/
helm install nats nats/nats --namespace nats --values values.yaml \
 --create-namespace
```

在上面创建的 values.yaml 文件中，需将以下组件放在 nats 名称空间中：

1. HA 版本的 NATS 有状态副本集，其中包含三个容器：NATS 服务器、配置重载器和指标分支。
2. NATS 箱容器，其中附带一组可用于校验设置的 NATS 实用程序。
3. JetStream 还会利用其键值后端，该后端附带与 Pod 绑定的 PVC。

32.2.1.1 测试设置

```
kubectl exec -n nats -it deployment/nats-box -- /bin/sh -l
```

1. 为测试主题创建订阅：

   ```
   nats sub test &
   ```

2. 向测试主题发送消息：

   ```
   nats pub test hi
   ```

32.2.1.2 清理

```
helm -n nats uninstall nats
rm values.yaml
```

32.2.2 NATS 用作 K3s 的后端

K3s 利用的一个组件是 KINE (https://github.com/k3s-io/kine)，它是一种适配层，能够用最初面向关系数据库的其他存储后端替代 etcd。由于 JetStream 提供了键值对 API，因此可以将 NATS 用作 K3s 群集的后端。

有一个已经合并的 PR 可以直接将内置的 NATS 包含在 K3s 中，但这项更改仍未包含 (https://github.com/k3s-io/k3s/issues/7410#issue-1692989394) 在 K3s 版本中。

出于此原因，应该手动构建 K3s 二进制文件。

32.2.2.1 构建 K3s

```
git clone --depth 1 https://github.com/k3s-io/k3s.git && cd k3s
```

以下命令会在构建标记中添加 `nats`，以在 K3s 中启用 NATS 内置功能：

```
sed -i '' 's/TAGS="ctrd/TAGS="nats ctrd/g' scripts/build
make local
```

请将 <node-ip> 替换为启动 K3s 的节点的实际 IP：

```
export NODE_IP=<node-ip>
sudo scp dist/artifacts/k3s-arm64 ${NODE_IP}:/usr/local/bin/k3s
```

注意

在本地构建 K3s 需要 buildx Docker CLI 插件。如果 `$ make local` 失败，可以手动安装 (https://github.com/docker/buildx#manual-download) 该插件。

32.2.2.2 安装 NATS CLI

```
TMPDIR=$(mktemp -d)
nats_version="nats-0.0.35-linux-arm64"
curl -o "${TMPDIR}/nats.zip" -sfL https://github.com/nats-io/natscli/releases/
download/v0.0.35/${nats_version}.zip
unzip "${TMPDIR}/nats.zip" -d "${TMPDIR}"

sudo scp ${TMPDIR}/${nats_version}/nats ${NODE_IP}:/usr/local/bin/nats
rm -rf ${TMPDIR}
```

32.2.2.3 运行用作 K3s 后端的 NATS

我们需要在节点上通过 `ssh` 进行连接，并使用指向 `nats` 的 `--datastore-endpoint` 标志运行 K3s。

注意

以下命令将 K3s 作为前台进程启动，因此您可以轻松地通过日志来查看是否出现了任何问题。为了不阻碍当前终端，可以在该命令的前面添加 `&` 标志，以将其作为后台进程启动。

```
k3s server  --datastore-endpoint=nats://
```

注意

为了将使用 NATS 后端的 K3s 服务器永久保留在您的 `slemicro` VM 上，可以运行以下脚本，以创建包含所需配置的 `systemd` 服务。

```
export INSTALL_K3S_SKIP_START=false
export INSTALL_K3S_SKIP_DOWNLOAD=true

curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server \
 --datastore-endpoint=nats://"  sh -
```

32.2.2.4 查错

可以在节点上运行以下命令来校验流的所有功能是否正常运行：

```
nats str report -a
nats str view -a
```

33 SUSE Linux Micro 上的 NVIDIA GPU

33.1 简介

本指南将演示如何通过预构建的开源驱动程序 (https://github.com/NVIDIA/open-gpu-kernel-modules) 在 SUSE Linux Micro 6.1 上实现主机级别的 NVIDIA GPU 支持。这些驱动程序是内置在操作系统中的，而非由 NVIDIA 的 GPU Operator (https://github.com/NVIDIA/gpu-operator) 动态加载。对于希望将部署所需的所有制品预先嵌入映像，且不需要动态选择驱动程序版本（即用户无需通过 Kubernetes 选择驱动程序版本）的客户而言，这种配置极具吸引力。本指南首先介绍如何将其他组件部署到已预先部署的系统，然后在一个章节中介绍如何通过 Edge Image Builder 将此配置嵌入到初始部署中。如果您不想了解基础知识并想要手动完成设置，请直接跳到该章节。

需要特别说明的是，这些驱动程序的支持服务由 SUSE 和 NVIDIA 紧密合作提供，SUSE 负责驱动程序的构建，并将其作为软件包储存库的一部分分发。不过，如果您在驱动程序的使用组合方面有任何疑问或顾虑，请咨询您的 SUSE 或 NVIDIA 客户经理以获得进一步的帮助。如果您打算使用 NVIDIA AI Enterprise (https://www.nvidia.com/en-gb/data-center/products/ai-enterprise/) (NVAIE)，请确保使用的是经过 NVAIE 认证的 GPU (https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/platform-support.html#supported-nvidia-gpus-and-systems)，这**可能**需要搭配使用 NVIDIA 的专有驱动程序。如果对此不确定，请咨询您的 NVIDIA 代表。

本指南**不会**介绍有关 NVIDIA GPU Operator 集成的更多信息。虽然其中不会介绍如何为 Kubernetes 集成 NVIDIA GPU Operator，但您仍然可以按照本指南中的大部分步骤来设置底层操作系统，并通过 NVIDIA GPU Operator Helm chart 中的 `driver.enabled=false` 标志来让 GPU Operator 使用**预安装的**驱动程序，在这种情况下，GPU Operator 会直接选择主机上安装的驱动程序。NVIDIA 在此处 (https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/install-gpu-operator.html#chart-customization-options) 提供了更详细的说明。

33.2 先决条件

如果您要学习本指南，事先需要做好以下准备：

- 至少一台装有 SUSE Linux Micro 6.1 的主机，可以是物理主机，也可以是虚拟机。
- 您的主机已附加到某个订阅，只有这样，才能访问软件包 — 可在此处 (https://www.suse.com/download/sle-micro/) 进行评估。
- 已安装兼容的 NVIDIA GPU (https://github.com/NVIDIA/open-gpu-kernel-modules#compatible-gpus)（或已**完全**直通到运行 SUSE Linux Micro 的虚拟机）。
- root 用户访问权限 — 本章中的说明假设您是 root 用户，而**未**通过 `sudo` 提升特权。

33.3 手动安装

本节介绍如何直接将 NVIDIA 驱动程序安装到 SUSE Linux Micro 操作系统上，因为 NVIDIA 开放驱动程序现在包含在核心 SUSE Linux Micro 软件包储存库中，因此，只需安装所需的 RPM 软件包就能安装这些驱动程序。无需编译或下载可执行软件包。下面介绍如何部署支持最新 GPU 的第六代 (G06) 驱动程序（有关更多信息，请参见此处 (https://en.opensuse.org/SDB:NVIDIA_drivers#Install)），请选择适合您系统中的 NVIDIA GPU 的驱动程序代系。对于新式 GPU，“G06”驱动程序是最常见的选择。

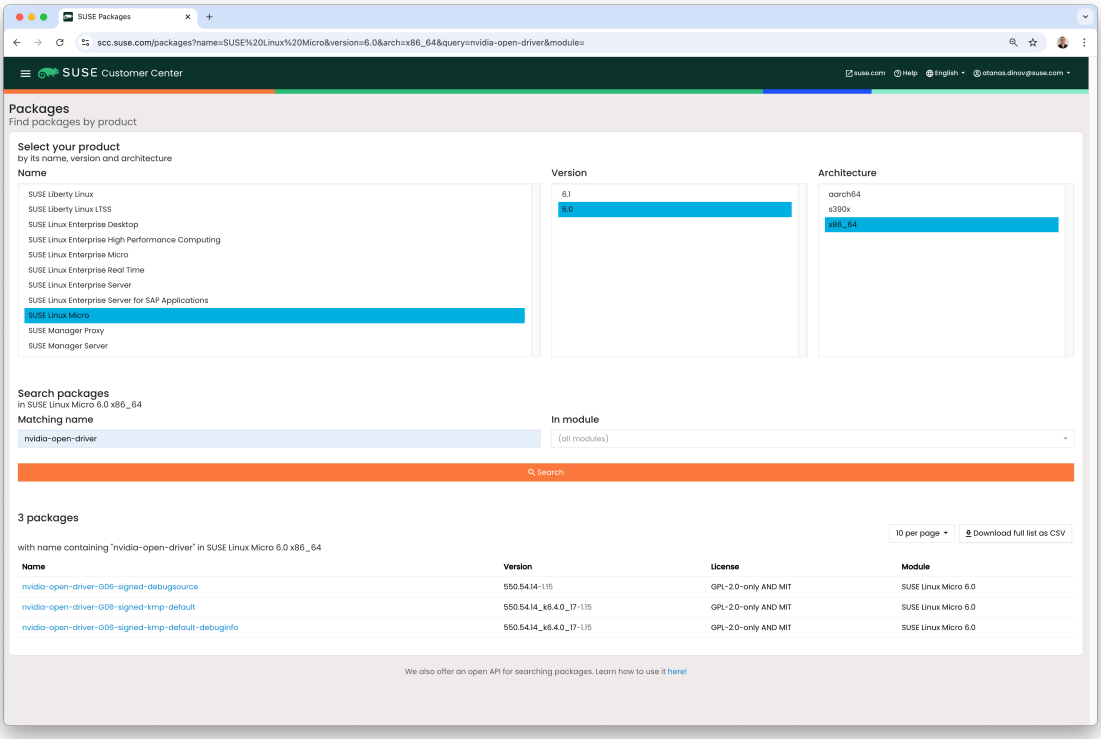
在开始之前，需要明确一点，除了 SUSE 在 SUSE Linux Micro 中随附的 NVIDIA 开放驱动程序之外，您的配置可能还需要其他 NVIDIA 组件。这些组件可能包括 OpenGL 库、CUDA 工具包、命令行实用程序（例如 `nvidia-smi`）和容器集成组件（例如 `nvidia-container-toolkit`）。其中许多组件并非由 SUSE 提供，因为它们属于 NVIDIA 的专有软件，或者由我们而非 NVIDIA 来提供这些组件并不合理。因此，在本指南的操作步骤中，我们将配置额外的储存库以便获取上述组件，并通过一些示例说明如何使用这些工具，最终搭建出一个功能完整的系统。需要注意区分 SUSE 储存库和 NVIDIA 储存库，因为两者提供的软件包版本偶尔会出现不匹配的情况。这种情况通常发生在 SUSE 发布了新版本的开放驱动程序后，NVIDIA 储存库可能需要几天时间才能提供与之匹配的对应版本软件包。

我们建议您采取以下措施来确保所选驱动程序版本与您的 GPU 兼容并符合现有的任何 CUDA 要求：

- 查看 CUDA 发行说明 (https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/)
- 检查您要部署的驱动程序版本是否在 NVIDIA 储存库 (https://download.nvidia.com/suse/sle15sp6/x86_64/) 中有匹配的版本，并确保支持组件有可用的对应软件包版本

提示

要查找 NVIDIA 开放驱动程序版本，请在目标计算机上运行 `zypper se -s nvidia-open-driver`，**或者**在 SUSE Customer Center 上适用于 AMD64/Intel64 的 SUSE Linux Micro 6.1 (https://scc.suse.com/packages?name=SUSE%20Linux%20Micro&version=6.1&arch=x86_64) 中搜索“nvidia-open-driver”。



在确认 NVIDIA 储存库中提供了对应版本后，便可以在主机操作系统上安装软件包了。为此，需要打开 `transactional-update` 会话，它会创建底层操作系统的新读/写快照，以便我们可以对不可变平台进行更改（有关 `transactional-update` 的更多说明，请参见此处 (https://documentation.suse.com/sle-micro/6.1/html/Micro-transactional-updates/transactional-updates.html)）：

```
transactional-update shell
```

在进入 `transactional-update` 外壳后，添加来自 NVIDIA 的其他软件包储存库。这样就可以提取其他实用程序，例如 `nvidia-smi`：

```
zypper ar https://download.nvidia.com/suse/sle15sp6/ nvidia-suse-main
zypper --gpg-auto-import-keys refresh
```

然后，可以安装驱动程序，并安装 `nvidia-compute-utils` 来获取其他实用程序。如果您不需要这些实用程序，可以省略其安装步骤，但为了稍后进行测试，最好现在就安装它们：

```
zypper install -y --auto-agree-with-licenses nvidia-open-driver-G06-signed-kmp
 nvidia-compute-utils-G06
```

注意

如果安装失败，可能表明所选驱动程序版本与 NVIDIA 在其储存库中提供的版本之间存在依赖项不匹配情况。请参见上一节来校验您的版本是否匹配。尝试安装不同的驱动程序版本。例如，如果 NVIDIA 储存库中的版本较低，您可以尝试在 install 命令中指定 `nvidia-open-driver-G06-signed-kmp=550.54.14`，以指定一致的版本。

接下来，如果您使用的 GPU **不**在支持列表中（支持列表可参考此处 (https://github.com/NVIDIA/open-gpu-kernel-modules#compatible-gpus)），可以尝试通过在模块层面启用支持来测试驱动程序是否能正常工作，但结果可能因设备而异 — 如果您使用的是**支持的** GPU，请跳过此步骤：

```
sed -i '/NVreg_OpenRmEnableUnsupportedGpus/s/^#//g' /etc/modprobe.d/50-nvidia-
default.conf
```

安装这些软件包后，请退出 `transactional-update` 会话：

```
exit
```

注意

在继续之前，请确保已退出 `transactional-update` 会话。

安装驱动程序后，接下来请重引导系统。由于 SUSE Linux Micro 是不可变的操作系统，因此它需要重引导至您在上一步骤中创建的新快照。驱动程序只会安装到此新快照中，因此如果系统不重引导至此新快照（会自动重引导），就无法加载驱动程序。准备就绪后，发出 reboot 命令：

```
reboot
```

系统成功重引导后，请重新登录并使用 `nvidia-smi` 工具校验驱动程序是否已成功加载，以及它是否可以访问和枚举您的 GPU：

```
nvidia-smi
```

此命令应显示如下所示的输出，请注意，以下示例中显示了两个 GPU：

```
+---------------------------------------------------------------------------------------
+
| NVIDIA-SMI 545.29.06              Driver Version: 545.29.06    CUDA Version:
 12.3     |
|-----------------------------------------+----------------------
+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
 Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
 Compute M. |
|                                         |                      |
 MIG M. |
|=========================================+======================
+======================|
|   0  NVIDIA A100-PCIE-40GB          Off | 00000000:17:00.0 Off |
      0 |
| N/A   29C    P0              35W / 250W |      4MiB / 40960MiB |      0%
 Default |
|                                         |                      |
 Disabled |
+-----------------------------------------+----------------------
+----------------------+
|   1  NVIDIA A100-PCIE-40GB          Off | 00000000:CA:00.0 Off |
      0 |
| N/A   30C    P0              33W / 250W |      4MiB / 40960MiB |      0%
 Default |
|                                         |                      |
 Disabled |
+-----------------------------------------+----------------------
+----------------------+
```

```
+-----------------------------------------------------------------------------
+
| Processes:
        |
|  GPU   GI   CI        PID   Type   Process name                          GPU
 Memory |
|        ID   ID
 Usage      |
|
=============================================================================|
|  No running processes found
        |
+-----------------------------------------------------------------------------
+
```

在 SUSE Linux Micro 系统上安装和校验 NVIDIA 驱动程序的过程到此结束。

33.4 进一步验证手动安装

在此阶段，我们只能确认的是，在主机级别，可以访问 NVIDIA 设备，并且驱动程序可以成功加载。但是，如果我们想要确保设备正常运行，可以通过一项简单测试来验证 GPU 是否可以从用户空间应用程序接收指令，最好是通过容器和 CUDA 库接收，因为实际工作负载通常使用这种方法。为此，我们可以通过安装 `nvidia-container-toolkit` (NVIDIA Container Toolkit (https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html#installing-with-zypper)) 来进一步修改主机操作系统。首先，打开另一个 `transactional-update` 外壳，请注意，在上一步骤中我们可能是通过单个事务执行此操作，后面的章节将介绍如何完全自动地执行此操作：

```
transactional-update shell
```

接下来，安装来自 NVIDIA Container Toolkit 储存库的 `nvidia-container-toolkit` 软件包：

- 下面的 `nvidia-container-toolkit.repo` 包含稳定储存库 (`nvidia-container-toolkit`) 和实验性储存库 (`nvidia-container-toolkit-experimental`)。对于生产用途，建议使用稳定储存库。默认会禁用实验性储存库。

```
zypper ar https://nvidia.github.io/libnvidia-container/stable/rpm/nvidia-
container-toolkit.repo
zypper --gpg-auto-import-keys install -y nvidia-container-toolkit
```

准备就绪后，可以退出 `transactional-update` 外壳：

```
exit
```

... 然后将计算机重引导至新快照：

```
reboot
```

注意

如前文所述，需确保已退出 `transactional-update` 外壳并重引导计算机，使更改生效。

重引导计算机后，可以校验系统是否可以使用 NVIDIA Container Toolkit 成功枚举设备。输出应该非常详细，包含 INFO 和 WARN 消息，但不包含 ERROR 消息：

```
nvidia-ctk cdi generate --output=/etc/cdi/nvidia.yaml
```

这可确保计算机上启动的任何容器都可以采用已发现的 NVIDIA GPU 设备。准备就绪后，可以运行基于 podman 的容器。通过 `podman` 执行此操作可以方便地从容器内部验证对 NVIDIA 设备的访问，稍后还可以放心地对 Kubernetes 执行同样的操作。根据 SLE BCI (https://registry.suse.com/repositories/bci-bci-base-15sp6)，为 `podman` 授予对上一条命令处理过的带标签 NVIDIA 设备的访问权限，然后直接运行 Bash 命令：

```
podman run --rm --device nvidia.com/gpu=all --security-opt=label=disable -it
 registry.suse.com/bci/bci-base:latest bash
```

现在，您将从临时 podman 容器内部执行命令。该容器无权访问您的底层系统，并且是临时性的，因此我们在此处执行的所有操作都不会保存，并且您无法破坏底层主机上的任何设置。由于我们现在处于容器中，因此可以安装所需的 CUDA 库。请再次对照此页面 (https://

docs.nvidia.com/cuda/cuda-toolkit-release-notes/) 检查驱动程序的 CUDA 版本是否正确，不过，先前 `nvidia-smi` 命令的输出应该也会显示所需的 CUDA 版本。以下示例将安装 **CUDA 12.3** 并提取许多示例、演示和开发包，以便您可以全面验证 GPU：

```
zypper ar https://developer.download.nvidia.com/compute/cuda/repos/sles15/
x86_64/ cuda-suse
zypper in -y cuda-libraries-devel-12-3 cuda-minimal-build-12-3 cuda-demo-
suite-12-3
```

成功安装后，请不要退出容器。我们将运行 `deviceQuery` CUDA 示例，它会全面验证通过 CUDA 以及从容器本身内部进行 GPU 访问的情况：

```
/usr/local/cuda-12/extras/demo_suite/deviceQuery
```

如果成功，您应会看到如下所示的输出，请注意命令结束后返回的 `Result = PASS` 消息，并注意在以下输出中，系统正确识别了两个 GPU，而您的环境中可能只有一个 GPU：

```
/usr/local/cuda-12/extras/demo_suite/deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "NVIDIA A100-PCIE-40GB"
  CUDA Driver Version / Runtime Version          12.2 / 12.1
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 40339 MBytes (42298834944
 bytes)
  (108) Multiprocessors, ( 64) CUDA Cores/MP:     6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1215 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072,
 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
```

```
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 23 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
 simultaneously) >

Device 1: <snip to reduce output for multiple devices>
     < Default (multiple host threads can use ::cudaSetDevice() with device
 simultaneously) >
> Peer access from NVIDIA A100-PCIE-40GB (GPU0) -> NVIDIA A100-PCIE-40GB
 (GPU1) : Yes
> Peer access from NVIDIA A100-PCIE-40GB (GPU1) -> NVIDIA A100-PCIE-40GB
 (GPU0) : Yes

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.3, CUDA Runtime
 Version = 12.3, NumDevs = 2, Device0 = NVIDIA A100-PCIE-40GB, Device1 = NVIDIA
 A100-PCIE-40GB
Result = PASS
```

在此处，您可以继续运行任何其他 CUDA 工作负载 — 使用编译器以及 CUDA 生态系统的任何其他组件来运行进一步的测试。完成后，可以退出容器，请注意您在容器中安装的任何内容都是临时性的（因此会丢失！），并且底层操作系统不会受到影响：

```
exit
```

33.5 使用 Kubernetes 实现

确认已在 SUSE Linux Micro 上安装并使用 NVIDIA 开放驱动程序后，我们来了解如何在同一台计算机上配置 Kubernetes。本指南不会指导您部署 Kubernetes，但假设您已安装 K3s (https://k3s.io/) 或 RKE2 (https://docs.rke2.io/install/quickstart)，并且已相应地配置 kubeconfig，以便能够以超级用户的身份执行标准 `kubectl` 命令。假设您的节点构成了单节点群集，不过，对多节点群集可以使用类似的核心步骤。首先，请确保可以正常进行 `kubectl` 访问：

```
kubectl get nodes
```

此命令应会显示如下所示的输出：

```
NAME       STATUS   ROLES                       AGE   VERSION
node0001   Ready    control-plane,etcd,master   13d   v1.32.4+rke2r1
```

您会发现，k3s/rke2 安装已检测到主机上的 NVIDIA Container Toolkit，并已将 NVIDIA 运行时集成自动配置到 `containerd`（k3s/rke2 使用的容器运行时接口）中。这一点可以通过检查 containerd `config.toml` 文件来确认：

```
tail -n8 /var/lib/rancher/rke2/agent/etc/containerd/config.toml
```

此命令必须显示如下所示的内容。对应的 K3s 位置是 `/var/lib/rancher/k3s/agent/etc/containerd/config.toml`：

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes."nvidia"]
  runtime_type = "io.containerd.runc.v2"
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes."nvidia".options]
  BinaryName = "/usr/bin/nvidia-container-runtime"
```

注意

如果未显示这些项，则可能表示检测失败。原因可能是计算机或 Kubernetes 服务未重启。如果需要，请如前文所述手动添加这些项。

接下来，我们需要将作为附加 Kubernetes 运行时的 NVIDIA `RuntimeClass` 配置为默认设置，以确保需要访问 GPU 的任何用户 Pod 请求都可以按照 `containerd` 配置中指定的方式，使用 NVIDIA Container Toolkit 通过 `nvidia-container-runtime` 进行这种访问：

```
kubectl apply -f - <<EOF
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: nvidia
handler: nvidia
EOF
```

下一步是配置 NVIDIA Device Plugin (https://github.com/NVIDIA/k8s-device-plugin)↗，该插件会将 Kubernetes 配置为利用 NVIDIA GPU 作为群集中可用的资源，并与 NVIDIA Container Toolkit 配合工作。此工具最初会检测底层主机上的所有功能，包括 GPU、驱动程序和其他功能（例如 GL），然后允许您请求 GPU 资源并将其用作应用程序的一部分。

首先，需要添加并更新 NVIDIA Device Plugin 的 Helm 储存库：

```
helm repo add nvdp https://nvidia.github.io/k8s-device-plugin
helm repo update
```

现在可以安装 NVIDIA Device Plugin：

```
helm upgrade -i nvdp nvdp/nvidia-device-plugin --namespace nvidia-device-plugin
 --create-namespace --version 0.14.5 --set runtimeClassName=nvidia
```

几分钟后，您会看到一个新的 Pod 正在运行，它将在可用节点上完成检测，并根据检测到的 GPU 数量来标记节点：

```
kubectl get pods -n nvidia-device-plugin
NAME                              READY   STATUS    RESTARTS      AGE
nvdp-nvidia-device-plugin-jp697   1/1     Running   2 (12h ago)   6d3h

kubectl get node node0001 -o json | jq .status.capacity
{
  "cpu": "128",
  "ephemeral-storage": "466889732Ki",
```

```
  "hugepages-1Gi": "0",
  "hugepages-2Mi": "0",
  "memory": "32545636Ki",
  "nvidia.com/gpu": "1",                        <----
  "pods": "110"
}
```

现在，可以创建一个 NVIDIA Pod 来尝试使用此 GPU。我们来尝试在 CUDA 基准容器上执行此操作：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: nbody-gpu-benchmark
  namespace: default
spec:
  restartPolicy: OnFailure
  runtimeClassName: nvidia
  containers:
  - name: cuda-container
    image: nvcr.io/nvidia/k8s/cuda-sample:nbody
    args: ["nbody", "-gpu", "-benchmark"]
    resources:
      limits:
        nvidia.com/gpu: 1
    env:
    - name: NVIDIA_VISIBLE_DEVICES
      value: all
    - name: NVIDIA_DRIVER_CAPABILITIES
      value: all
EOF
```

如果一切顺利，您可以在日志中看到基准测试信息：

```
kubectl logs nbody-gpu-benchmark
Run "nbody -benchmark [-numbodies=<numBodies>]" to measure performance.
 -fullscreen       (run n-body simulation in fullscreen mode)
 -fp64             (use double precision floating point values for simulation)
```

```
 -hostmem          (stores simulation data in host memory)
 -benchmark        (run benchmark to measure performance)
 -numbodies=<N>    (number of bodies (>= 1) to run in simulation)
 -device=<d>       (where d=0,1,2.... for the CUDA device to use)
 -numdevices=<i>   (where i=(number of CUDA devices > 0) to use for simulation)
 -compare          (compares simulation results running once on the default GPU
 and once on the CPU)
 -cpu              (run n-body simulation on the CPU)
 -tipsy=<file.bin> (load a tipsy model file for simulation)

NOTE: The CUDA Samples are not meant for performance measurements. Results may
 vary when GPU Boost is enabled.

> Windowed mode
> Simulation data stored in video memory
> Single precision floating point simulation
> 1 Devices used for simulation
GPU Device 0: "Turing" with compute capability 7.5

> Compute 7.5 CUDA device: [Tesla T4]
40960 bodies, total time for 10 iterations: 101.677 ms
= 165.005 billion interactions per second
= 3300.103 single-precision GFLOP/s at 20 flops per interaction
```

最后，如果您的应用程序需要 OpenGL，您可以在主机级别安装所需的 NVIDIA OpenGL 库，NVIDIA Device Plugin 和 NVIDIA Container Toolkit 可将这些库提供给容器。为此，请如下所示安装软件包：

```
transactional-update pkg install nvidia-gl-G06
```

注意

需要重引导系统才能将此软件包提供给应用程序。NVIDIA Device Plugin 会通过 NVIDIA Container Toolkit 自动重新检测此软件包。

33.6 通过 Edge Image Builder 整合配置

现在您已经在 SUSE Linux Micro 上验证了应用程序和 GPU 的完整功能，现在可能希望通过第 11 章 “Edge Image Builder”将所有配置整合到一个可部署/可使用的 ISO 或 RAW 磁盘映像。本指南不会介绍如何使用 Edge Image Builder，但提供了构建此类映像所需的配置。以下是一个映像定义示例以及必要的 Kubernetes 配置文件，确保所有必需组件都能在部署时默认安装到位。下面是该示例对应的 Edge Image Builder 目录结构：

```
.
├── base-images
│   └── SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
├── eib-config-iso.yaml
├── kubernetes
│   ├── config
│   │   └── server.yaml
│   ├── helm
│   │   └── values
│   │       └── nvidia-device-plugin.yaml
│   └── manifests
│       └── nvidia-runtime-class.yaml
└── rpms
    └── gpg-keys
        └── nvidia-container-toolkit.key
```

我们来浏览这些文件。首先，这是一个运行 K3s 的单节点群集的示例映像定义，它还会部署实用程序和 OpenGL 软件包 (`eib-config-iso.yaml`)：

```
apiVersion: 1.2
image:
  arch: x86_64
  imageType: iso
  baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
  outputImageName: deployimage.iso
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      pools:
```

```
        - 2.suse.pool.ntp.org
  isoConfiguration:
    installDevice: /dev/sda
  users:
    - username: root
      encryptedPassword: $6$XcQN1xkuQKjWEtQG
$WbhV80rbveDLJDz1c93K5Ga9JDjt3mF.ZUnhYtsS7uE52FR8mmT8Cnii/
JPeFk9jzQO6eapESYZesZHO9EslD1
  packages:
    packageList:
      - nvidia-open-driver-G06-signed-kmp-default
      - nvidia-compute-utils-G06
      - nvidia-gl-G06
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://download.nvidia.com/suse/sle15sp6/
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
    sccRegistrationCode: [snip]
kubernetes:
  version: v1.32.4+k3s1
  helm:
    charts:
      - name: nvidia-device-plugin
        version: v0.14.5
        installationNamespace: kube-system
        targetNamespace: nvidia-device-plugin
        createNamespace: true
        valuesFile: nvidia-device-plugin.yaml
        repositoryName: nvidia
    repositories:
      - name: nvidia
        url: https://nvidia.github.io/k8s-device-plugin
```

注意

这只是一个示例。您可能需要根据自己的要求和期望对其进行自定义。此外，如果使用 SUSE Linux Micro，您需要提供自己的 `sccRegistrationCode` 来解析软件包依赖项并提取 NVIDIA 驱动程序。

除此之外，还需要添加其他组件，供 Kubernetes 在引导时加载。首先，EIB 目录需要一个 `kubernetes` 目录，其中包含用于保存配置、Helm chart 值和任何其他所需清单的子目录：

```
mkdir -p kubernetes/config kubernetes/helm/values kubernetes/manifests
```

现在我们来通过选择 CNI（如果未选择，则默认为 Cilium）并启用 SELinux 来设置（可选的）Kubernetes 配置：

```
cat << EOF > kubernetes/config/server.yaml
cni: cilium
selinux: true
EOF
```

现在确保在 Kubernetes 群集上创建 NVIDIA RuntimeClass：

```
cat << EOF > kubernetes/manifests/nvidia-runtime-class.yaml
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: nvidia
handler: nvidia
EOF
```

我们将使用内置的 Helm 控制器通过 Kubernetes 本身来部署 NVIDIA Device Plugin。我们需要在 chart 的值文件中提供运行时类：

```
cat << EOF > kubernetes/helm/values/nvidia-device-plugin.yaml
runtimeClassName: nvidia
EOF
```

在继续之前，我们需要抓取 NVIDIA Container Toolkit RPM 公共密钥：

```
mkdir -p rpms/gpg-keys
```

```
curl -o rpms/gpg-keys/nvidia-container-toolkit.key https://nvidia.github.io/
libnvidia-container/gpgkey
```

所有必需的制品（包括 Kubernetes 二进制文件、容器映像、Helm chart（以及所有引用的映像））都会自动实现隔离处理，这意味着在部署时，系统默认不需要互联网连接。现在您只需从 SUSE 下载页面 (https://www.suse.com/download/sle-micro/) 抓取 SUSE Linux Micro ISO（并将其放入 `base-images` 目录），然后调用 Edge Image Builder 工具即可生成 ISO。作为示例补充，以下是用于构建映像的命令：

```
podman run --rm --privileged -it -v /path/to/eib-files/:/eib \
registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
build --definition-file eib-config-iso.yaml
```

有关更多说明，请参见 Edge Image Builder 的文档 (https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md)。

33.7 解决问题

33.7.1 nvidia-smi 找不到 GPU

使用 `dmesg` 检查内核消息。如果消息指出无法分配 `NvKMSKapDevice`，请采用“GPU 不受支持”临时解决方法：

```
sed -i '/NVreg_OpenRmEnableUnsupportedGpus/s/^#//g' /etc/modprobe.d/50-nvidia-
default.conf
```

> **注意**：如果您在上述步骤中更改了内核模块配置，则需要重新加载或重引导内核模块才能使更改生效。

VI Day 2 操作

本章介绍管理员如何在管理群集和下游群集上处理不同的“Day 2”操作任务。

34 Edge 3.3 迁移

本章介绍如何将管理群集和下游群集从 Edge 3.2 迁移到 Edge 3.3.0。

重要

始终从 Edge 3.2 的最新 z-stream 版本执行群集迁移。

始终迁移到 Edge 3.3.0 版本。对于迁移后的后续升级，请参见管理群集（第 35 章 “管理群集”）和下游群集（第 36 章 “下游群集”）章节。

34.1 管理群集

本节将介绍以下主题：

第 34.1.1 节 “先决条件” - 开始迁移前需要完成的先决步骤。

第 34.1.2 节 “升级控制器” - 如何使用第 23 章 “升级控制器”执行管理群集迁移。

第 34.1.3 节 “Fleet” - 如何使用第 8 章 “Fleet”执行管理群集迁移。

34.1.1 先决条件

34.1.1.1 升级裸机操作器 CRD

注意

仅适用于需要进行 第 10 章 “Metal³” chart 升级的群集。

Metal3 Helm chart 利用 Helm 的 CRD (https://helm.sh/docs/chart_best_practices/custom_resource_definitions/#method-1-let-helm-do-it-for-you) 目录来包含裸机操作器 (BMO) (https://book.metal3.io/bmo/introduction.html) CRD。

然而，这种方法存在一定的局限性，特别是无法使用 Helm 升级此目录中的 CRD。
有关详细信息，请参见 Helm 文档 (https://helm.sh/docs/chart_best_practices/custom_resource_definitions/#some-caveats-and-explanations)。

因此，在将 Metal[3] 升级到与 Edge 3.3.0 兼容的版本之前，用户必须手动升级底层的 BMO CRD。

在安装了 Helm 且 kubectl 已配置为指向管理群集的计算机上：

1. 手动应用 BMO CRD：

```
helm show crds oci://registry.suse.com/edge/charts/metal3 --version
 303.0.7+up0.11.5 | kubectl apply -f -
```

34.1.2 升级控制器

重要

升级控制器目前仅支持**非隔离管理**群集的 Edge 版本迁移。

本节将介绍以下主题：

第 34.1.2.1 节 “先决条件” - 针对升级控制器的先决条件。

第 34.1.2.2 节 “迁移步骤” - 使用升级控制器将管理群集迁移到新 Edge 版本的步骤。

34.1.2.1 先决条件

34.1.2.1.1 Edge 3.3 升级控制器

使用升级控制器之前，必须先确保其运行的版本能够迁移到目标 Edge 版本。

操作步骤如下：

1. 如果您已在之前的 Edge 版本中部署了升级控制器，请升级其 chart：

```
helm upgrade upgrade-controller -n upgrade-controller-system oci://
registry.suse.com/edge/charts/upgrade-controller --version 303.0.1+up0.1.1
```

2. 如果您尚**未**部署升级控制器，请按照第 23.3 节 “安装升级控制器” 中所述操作。

34.1.2.2 迁移步骤

使用升级控制器执行管理群集迁移本质上与执行升级类似。

唯一的区别是，`UpgradePlan` **必须**指定 `3.3.0` 版本：

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  # Change to the namespace of your Upgrade Controller
  namespace: CHANGE_ME
spec:
  releaseVersion: 3.3.0
```

有关如何使用上述 `UpgradePlan` 进行迁移的信息，请参见升级控制器升级过程（第 35.1 节 “升级控制器”）。

34.1.3 Fleet

注意

只要有可能，便使用第 34.1.2 节 “升级控制器” 进行迁移。

仅在升级控制器未涵盖的使用场景下，才需参考本节内容。

使用 `Fleet` 执行管理群集迁移本质上与执行升级类似。

主要区别在于：

1. **必须使用** release-3.3.0 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0) 版本 `suse-edge/fleet-examples` 储存库中的 Fleet。
2. 安排升级的 chart **必须**升级到与 `Edge 3.3.0` 版本兼容的版本。有关 `Edge 3.3.0` 组件的列表，请参见第 52.4 节 “版本 3.3.0”。

重要

为确保 `Edge 3.3.0` 迁移成功，用户必须遵守上述要点。

鉴于上述要点，用户可参考管理群集 Fleet（第 35.2 节 “Fleet”）文档，获取执行迁移需要完成的步骤的全面指南。

34.2 下游群集

第 34.2.1 节 “Fleet” - 如何使用第 8 章 “Fleet”执行下游群集迁移。

34.2.1 Fleet

使用 `Fleet` 执行下游群集迁移本质上与执行升级类似。

主要区别在于：

1. **必须使用** release-3.3.0 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0) 版本 `suse-edge/fleet-examples` 储存库中的 Fleet。
2. 安排升级的 chart **必须**升级到与 `Edge 3.3.0` 版本兼容的版本。有关 `Edge 3.3.0` 组件的列表，请参见第 52.4 节 “版本 3.3.0”。

重要

为确保 `Edge 3.3.0` 迁移成功，用户必须遵守上述要点。

鉴于上述要点，用户可参考下游群集 Fleet（第 36.1 节 “Fleet”）文档，获取执行迁移需要完成的步骤的全面指南。

35 管理群集

目前，可以通过两种方式在管理群集上执行“Day 2”操作：

1. 通过 第 23 章 “升级控制器” - 第 35.1 节 “升级控制器”
2. 通过 第 8 章 “Fleet” - 第 35.2 节 “Fleet”

35.1 升级控制器

重要

升级控制器目前仅支持**非隔离管理**群集的 Day 2 操作。

本节介绍如何执行与将管理群集从一个 Edge 平台版本升级到另一个版本相关的各种 Day 2 操作。

Day 2 操作由升级控制器（第 23 章 “升级控制器”）自动执行，包括：

- SUSE Linux Micro（第 9 章 “SUSE Linux Micro”）操作系统升级
- 第 16 章 “RKE2”或第 15 章 “K3s” Kubernetes 升级
- 其他 SUSE 组件（SUSE Rancher Prime、SUSE Security 等）升级

35.1.1 先决条件

升级管理群集之前，必须满足以下先决条件：

1. 已在 SCC 中注册的节点 - 确保群集节点的操作系统已使用相应的订阅密钥注册，且该密钥支持要升级到的 Edge 版本中指定的操作系统版本（第 52.1 节 “摘要”）。
2. 升级控制器 - 确保管理群集上已部署升级控制器。有关安装步骤，请参见第 23.3 节 “安装升级控制器”。

35.1.2 升级

1. 确定要将管理群集升级到哪个 Edge 版本（第 52.1 节 “摘要”）。

2. 在管理群集中，部署一个指定目标版本的 UpgradePlan。UpgradePlan 必须部署在升级控制器的名称空间中。

```
kubectl apply -n <upgrade_controller_namespace> -f - <<EOF
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
spec:
  # Version retrieved from release notes
  releaseVersion: 3.X.Y
EOF
```

注意

某些情况下，除了 UpgradePlan 之外，您可能还想进行其他配置。有关所有可能的配置，请参见第 23.5.1 节 “UpgradePlan”。

3. 将 UpgradePlan 部署到升级控制器的名称空间后，升级过程即会开始。

注意

有关实际升级过程的详细信息，请参见第 23.4 节 “升级控制器的工作原理”。

有关如何跟踪升级过程的信息，请参见第 23.6 节 “跟踪升级过程”。

35.2 Fleet

本节提供有关如何使用 Fleet（第 8 章 “Fleet”）组件执行“Day 2”操作的信息。

本节将介绍以下主题：

1. 第 35.2.1 节 “组件” - 执行所有“Day 2”操作需要使用的默认组件。
2. 第 35.2.2 节 “确定您的使用场景” - 概述将使用的 Fleet 自定义资源及其在不同“Day 2”操作场景中的适用性。
3. 第 35.2.3 节 “Day 2 工作流程” - 提供使用 Fleet 执行“Day 2”操作的工作流程指南。
4. 第 35.2.4 节 “操作系统升级” - 说明如何使用 Fleet 进行操作系统升级。
5. 第 35.2.5 节 “Kubernetes 版本升级” - 说明如何使用 Fleet 进行 Kubernetes 版本升级。
6. 第 35.2.6 节 “Helm chart 升级” - 说明如何使用 Fleet 进行 Helm chart 升级。

35.2.1 组件

下文将介绍为使用 Fleet 顺利执行“Day 2”操作而应在管理群集上设置的默认组件。

35.2.1.1 Rancher

可选组件；负责管理下游群集并在管理群集上部署系统升级控制器。

有关详细信息，请参见第 5 章 “Rancher”。

35.2.1.2 系统升级控制器 (SUC)

系统升级控制器负责根据通过名为计划的自定义资源提供的配置数据，在指定节点上执行任务。

系统会主动利用 **SUC** 升级操作系统和 Kubernetes 发行版。

有关 **SUC** 组件及其如何安置到 Edge 堆栈中的详细信息，请参见第 22 章 “系统升级控制器”。

35.2.2 确定您的使用场景

Fleet 使用两种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) 来实现对 Kubernetes 和 Helm 资源的管理。

下文将介绍这些资源的用途，以及在“Day 2”操作情境中它们最适合的使用场景。

35.2.2.1 GitRepo

GitRepo 是一种 Fleet（第 8 章 “Fleet”）资源，它代表一个 Git 储存库，Fleet 可从中创建捆绑包。每个捆绑包都是基于 GitRepo 资源中定义的配置路径创建的。有关详细信息，请参见 GitRepo (https://fleet.rancher.io/gitrepo-add) 文档。

在“Day 2”操作情境中，GitRepo 资源通常用于在利用 **Fleet GitOps** 方法的**非隔离**环境中部署 SUC 或 SUC 计划。

或者，**如果您通过本地 git 服务器镜像储存库设置**，则 GitRepo 资源也可用于在**隔离**环境中部署 SUC 或 SUC 计划。

35.2.2.2 捆绑包

捆绑包包含了要在目标群集上部署的**原始** Kubernetes 资源。通常它们是基于 GitRepo 资源创建的，但在某些使用场景中也可以手动部署。有关详细信息，请参见捆绑包 (https://fleet.rancher.io/bundle-add) 文档。

在“Day 2”操作情境中，捆绑包资源通常用于在不使用某种形式的**本地 GitOps** 过程（例如**本地 git 服务器**）的**隔离**环境中部署 SUC 或 SUC 计划。

或者，如果您的应用场景不允许使用 **GitOps** 工作流程（例如需使用 Git 储存库），则捆绑包资源也可用于在**非隔离**环境中部署 SUC 或 SUC 计划。

35.2.3 Day 2 工作流程

下面是在将管理群集升级到特定 Edge 版本时应遵循的“Day 2”工作流程。

1. 操作系统升级（第 35.2.4 节 “操作系统升级”）
2. Kubernetes 版本升级（第 35.2.5 节 “Kubernetes 版本升级”）
3. Helm chart 升级（第 35.2.6 节 “Helm chart 升级”）

35.2.4 操作系统升级

本节介绍如何使用第 8 章 “Fleet”和第 22 章 “系统升级控制器”执行操作系统升级。

本节将介绍以下主题：

1. 第 35.2.4.1 节 “组件” - 升级过程使用的其他组件。
2. 第 35.2.4.2 节 “概述” - 升级过程概述。
3. 第 35.2.4.3 节 “要求” - 升级过程的要求。
4. 第 35.2.4.4 节 “操作系统升级 - SUC 计划部署” - 关于如何部署负责触发升级过程的 SUC 计划的信息。

35.2.4.1 组件

本节介绍操作系统升级过程中使用的自定义组件，这些组件与默认 “Day 2” 组件（第 35.2.1 节 “组件”）不同。

35.2.4.1.1 systemd.service

特定节点上的操作系统升级由 systemd.service (https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html) 处理。

根据 Edge 版本升级时操作系统所需的不同升级类型，会创建不同的服务：

- 对于需要相同操作系统版本（如 `6.0`）的 Edge 版本，将创建 `os-pkg-update.service`。它使用 transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) 执行常规软件包升级 (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages)。
- 对于需要迁移操作系统版本（例如 `6.0` → `6.1`）的 Edge 版本，将创建 `os-migration.service`。它使用 transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) 来执行：
  a. 常规软件包升级 (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages)，这可确保所有软件包均为最新版本，以减少因软件包版本过旧而导致的迁移失败情况。
  b. 利用 `zypper migration` 命令进行操作系统迁移。

上述服务通过 SUC 计划部署到每个节点，该计划必须位于需要升级操作系统的管理群集上。

35.2.4.2 概述

通过 Fleet 和系统升级控制器（SUC）来为管理群集节点升级操作系统。

Fleet 用于将 SUC 计划部署到目标群集并对其进行管理。

注意

SUC 计划是一种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)，描述了 SUC 为在一组节点上执行特定任务而需要遵循的步骤。有关 SUC 计划的示例，请参见上游储存库 (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans)。

通过向特定 Fleet 工作区 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 部署 GitRepo (https://fleet.rancher.io/gitrepo-add) 或捆绑包 (https://fleet.rancher.io/bundle-add) 资源将操作系统 SUC 计划分发到各个群集。Fleet 会获取已部署的 GitRepo/捆绑包，并将其内容（操作系统 SUC 计划）部署到目标群集。

注意

GitRepo/捆绑包资源始终部署在管理群集上。使用 GitRepo 还是捆绑包资源取决于具体应用场景，有关详细信息，请参见第 35.2.2 节 “确定您的使用场景”。

操作系统 SUC 计划描述了以下工作流程：

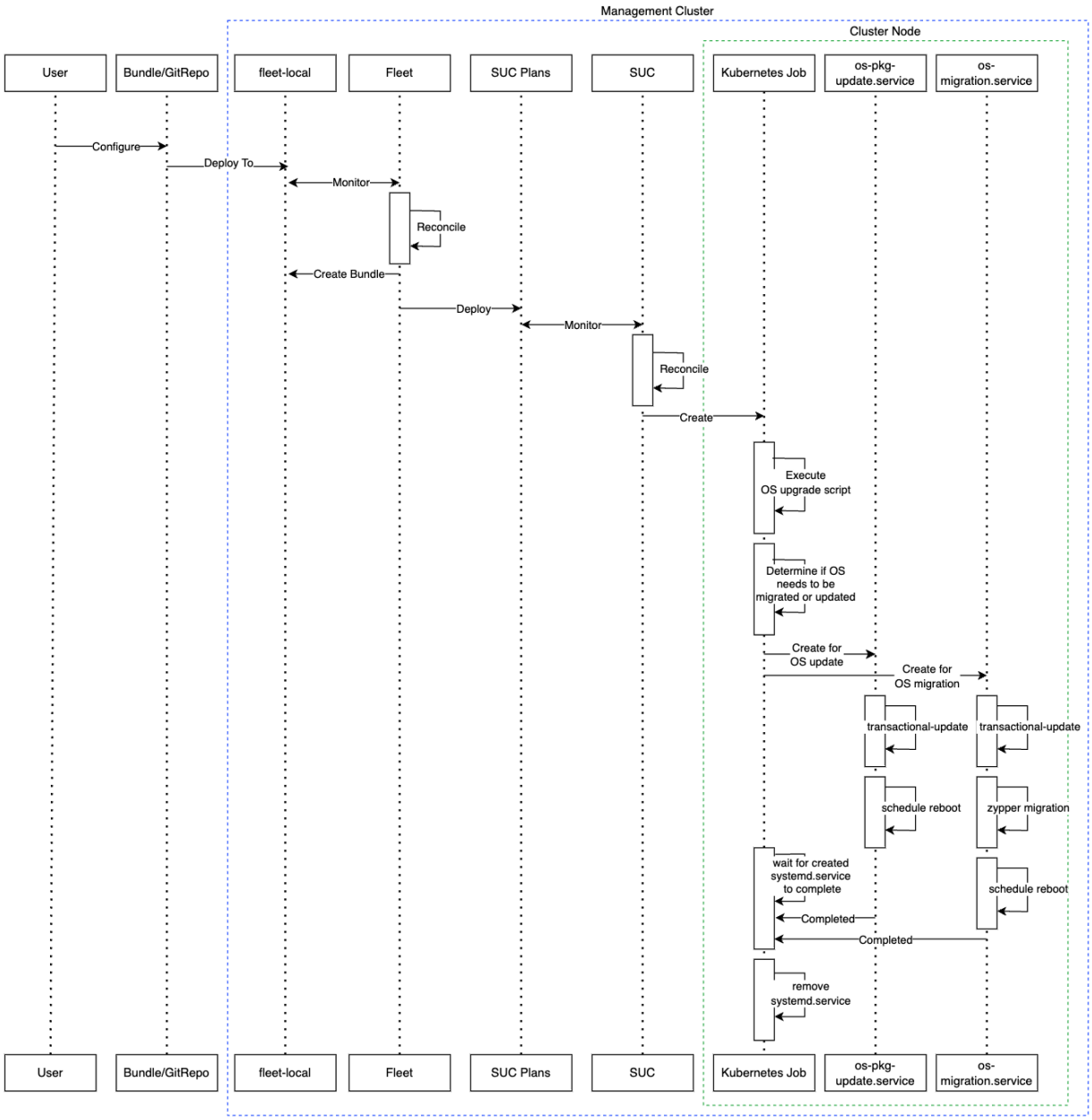
1. 执行操作系统升级前，务必要封锁 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) 节点。
2. 务必先升级控制平面节点，再升级工作节点。
3. 升级群集时，务必**逐个**节点依序升级。

部署操作系统 SUC 计划后，工作流程如下：

1. SUC 协调已部署的操作系统 SUC 计划，并在**每个节点**上创建一个 Kubernetes 作业。
2. Kubernetes 作业创建一个 systemd.service（第 35.2.4.1.1 节 “systemd.service”），用于执行软件包升级或操作系统迁移。
3. 所创建的 `systemd.service` 触发特定节点上的操作系统升级过程。

> **重要**
>
> 操作系统升级过程完成后，相应节点将重引导以应用系统更新。

下面是上述流程的示意图：



35.2.4.3 要求

一般：

1. **已在 SCC 中注册的计算机** - 所有管理群集节点都应已注册到 `https://scc.suse.com/`。只有这样，`systemd.service` 才能成功连接到所需的 RPM 储存库。

重要

对于需要进行操作系统版本迁移的 Edge 版本（例如 `6.0` → `6.1`），请确保您的 SCC 密钥支持迁移到新版本。

2. **确保 SUC 计划容忍度与节点容忍度相匹配** - 如果您的 Kubernetes 群集节点具有自定义**污点**，请确保在 **SUC 计划**中为这些污点添加容忍度 (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)。默认情况下，**SUC 计划**仅包含**控制平面**节点的容忍度。默认容忍度包括：

   - **CriticalAddonsOnly=true:NoExecute**
   - **node-role.kubernetes.io/control-plane:NoSchedule**
   - **node-role.kubernetes.io/etcd:NoExecute**

注意

其他任何容忍度必须添加到每个计划的 `.spec.tolerations` 部分。与操作系统升级相关的 **SUC 计划**可以在 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库中的 `fleets/day2/system-upgrade-controller-plans/os-upgrade` 下找到。**请确保使用有效储存库版本 (https://github.com/suse-edge/fleet-examples/releases) 标记中的计划。**

为**控制平面** SUC 计划定义自定义容忍度的示例如下：

```
apiVersion: upgrade.cattle.io/v1
kind: Plan
metadata:
  name: os-upgrade-control-plane
spec:
  ...
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
```

```
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

隔离：

1. **镜像 SUSE RPM 储存库** - 操作系统 RPM 储存库应镜像到本地，以便 `systemd.service` 可以访问它们。您可以使用 RMT (https://documentation.suse.com/sles/15-SP6/html/SLES-all/book-rmt.html) 或 SUMA (https://documentation.suse.com/suma/5.0/en/suse-manager/index.html) 来完成该操作。

35.2.4.4 操作系统升级 - SUC 计划部署

重要

对于之前使用此过程升级的环境，用户应确保完成以下步骤之一：

- 从管理群集中去除任何先前部署且与旧版 `Edge` 相关的 `SUC` 计划 - 方法是从现有的 `GitRepo/捆绑包`目标配置 (https://fleet.rancher.io/gitrepo-targets#target-matching) 中去除相应群集，或完全去除 `GitRepo/捆绑包`资源。
- 重用现有的 `GitRepo/捆绑包资源` - 方法是将资源的修订版指向一个新标签，该标签包含目标 `suse-edge/fleet-examples` 版本 (https://github.com/suse-edge/fleet-examples/releases) 的正确 Fleet。

这样做是为了避免旧版 Edge 的 SUC 计划之间发生冲突。

如果用户尝试升级，而管理群集上存在现有的 SUC 计划，他们将看到以下 Fleet 错误：

```
Not installed: Unable to continue with install: Plan <plan_name> in
 namespace <plan_namespace> exists and cannot be imported into the current
 release: invalid ownership metadata; annotation validation error..
```

如第 35.2.4.2 节 “概述”中所述，可以通过以下任意一种方式将 SUC 计划分发到目标群集来完成操作系统升级：

- Fleet GitRepo 资源 - 第 35.2.4.4.1 节 “SUC 计划部署 - GitRepo 资源”。
- Fleet 捆绑包资源 - 第 35.2.4.4.2 节 “SUC 计划部署 - 捆绑包资源”。

要确定使用哪个资源，请参见第 35.2.2 节 “确定您的使用场景”。

对于希望通过第三方 GitOps 工具部署操作系统 SUC 计划的使用场景，请参见第 35.2.4.4.3 节 “SUC 计划部署 - 第三方 GitOps 工作流程”。

35.2.4.4.1 SUC 计划部署 - GitRepo 资源

提供所需操作系统 SUC 计划的 **GitRepo** 资源可通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 35.2.4.4.1.1 节 “GitRepo 创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 35.2.4.4.1.2 节 “GitRepo 创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的操作系统升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

35.2.4.4.1.1 GitRepo 创建 - Rancher UI

要通过 Rancher UI 创建 GitRepo 资源，请遵循其官方文档 (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

Edge 团队维护着一个即用型 Fleet (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/os-upgrade)。根据您的环境的不同，该 Fleet 可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases)标记使用此 Fleet。

对于不需要在 Fleet 附带的 SUC 计划中包含自定义更改的使用场景，用户可以直接引用 `suse-edge/fleet-examples` 储存库中的 `os-upgrade` Fleet。

如果需要自定义更改（例如添加自定义容忍度），用户应从单独的储存库中引用 `os-upgrade` Fleet，以便能够根据需要将更改添加到 SUC 计划中。

有关如何配置 `GitRepo` 以使用 `suse-edge/fleet-examples` 储存库中的 Fleet 的示例，请参见此处 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/os-upgrade-gitrepo.yaml)。

35.2.4.4.1.2 GitRepo 创建 - 手动

1. 提取 **GitRepo** 资源：

```
curl -o os-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/os-upgrade-gitrepo.yaml
```

2. 编辑 **GitRepo** 配置：

   - 去除 `spec.targets` 部分 - 该部分仅用于下游群集。

```
# Example using sed
sed -i.bak '/^  targets:/,$d' os-upgrade-gitrepo.yaml && rm -f os-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
```

```
yq eval 'del(.spec.targets)' -i os-upgrade-gitrepo.yaml
```

- 将 GitRepo 的名称空间指向 fleet-local 名称空间 - 这样做是为了在管理群集上部署该资源。

```
# Example using sed
sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/' os-
upgrade-gitrepo.yaml && rm -f os-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
yq eval '.metadata.namespace = "fleet-local"' -i os-upgrade-
gitrepo.yaml
```

3. 将 **GitRepo** 资源应用于管理群集：

```
kubectl apply -f os-upgrade-gitrepo.yaml
```

4. 查看 fleet-local 名称空间下创建的 **GitRepo** 资源：

```
kubectl get gitrepo os-upgrade -n fleet-local

# Example output
NAME            REPO                                                COMMIT
      BUNDLEDEPLOYMENTS-READY   STATUS
os-upgrade      https://github.com/suse-edge/fleet-examples.git
 release-3.3.0  0/0
```

35.2.4.4.2 SUC 计划部署 - 捆绑包资源

提供所需操作系统 SUC 计划的**捆绑包**资源可以通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 35.2.4.4.2.1 节 “捆绑包创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 35.2.4.4.2.2 节 “捆绑包创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的操作系统升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

35.2.4.4.2.1 捆绑包创建 - Rancher UI

Edge 团队维护着一个可在以下步骤中使用的即用型捆绑包 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml)。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases)标记使用此捆绑包。

要通过 Rancher 的 UI 创建捆绑包，请执行以下操作：

1. 单击左上角的 **#** → **Continuous Delivery（持续交付）**
2. 转到 **Advanced**（高级）> **Bundles**（捆绑包）
3. 选择 **Create from YAML**（基于 YAML 创建）
4. 此处可通过以下方式之一创建捆绑包：

注意

在某些使用场景中，您可能需要在捆绑包附带的 SUC 计划中包含自定义更改。请确保在以下步骤生成的捆绑包中包含这些更改。

a. 手动将 `suse-edge/fleet-examples` 中的捆绑包内容 (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml)复制到 **Create from YAML**（基于 YAML 创建）页面。

b. 从所需的版本 (https://github.com/suse-edge/fleet-examples/releases)标记克隆 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库，并在 **Create from YAML**（基于 YAML 创建）页面中选择 **Read from File**（从文件读取）选项。然后导航到捆绑包位置 (`bundles/day2/system-upgrade-controller-plans/os-upgrade`) 并选择捆绑包文件。这会在**Create from YAML**（基于 YAML 创建）页面中自动填充捆绑包内容。

5. 在 Rancher UI 中编辑捆绑包：

   - 更改捆绑包的**名称空间**，使其指向 `fleet-local` 名称空间。

   ```
   # Example
   kind: Bundle
   apiVersion: fleet.cattle.io/v1alpha1
   metadata:
     name: os-upgrade
     namespace: fleet-local
   ...
   ```

   - 更改捆绑包的**目标**群集，使其指向本地（管理）群集：

   ```
   spec:
     targets:
     - clusterName: local
   ```

   ### 注意

   在某些使用场景中，您的本地群集的名称可能会有所不同。

   要获取本地群集名称，请执行以下命令：

   ```
   kubectl get clusters.fleet.cattle.io -n fleet-local
   ```

6. 选择 **Create**（创建）

35.2.4.4.2.2 捆绑包创建 - 手动

1. 提取**捆绑包**资源：

   ```
   curl -o os-upgrade-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml
   ```

2. 编辑捆绑包配置：

- 更改捆绑包的**目标**群集，使其指向本地（管理）群集：

```
spec:
  targets:
  - clusterName: local
```

注意

在某些使用场景中，您的本地群集的名称可能会有所不同。

要获取本地群集名称，请执行以下命令：

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

- 更改捆绑包的**名称空间**，使其指向 `fleet-local` 名称空间。

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: os-upgrade
  namespace: fleet-local
...
```

3. 将**捆绑包**资源应用于管理群集：

```
kubectl apply -f os-upgrade-bundle.yaml
```

4. 查看 `fleet-local` 名称空间下创建的**捆绑包**资源：

```
kubectl get bundles -n fleet-local
```

35.2.4.4.3 SUC 计划部署 - 第三方 GitOps 工作流程

在某些使用场景中，用户可能希望将操作系统 SUC 计划合并到自己的第三方 GitOps 工作流程（例如 Flux）中。

要获取所需的操作系统升级资源，首先请确定您要使用的 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记。

然后，便可以在 `fleets/day2/system-upgrade-controller-plans/os-upgrade` 中找到资源，其中：

- `plan-control-plane.yaml` 是用于**控制平面**节点的 SUC 计划资源。
- `plan-worker.yaml` 是用于**工作**节点的 SUC 计划资源。
- `secret.yaml` 是一个包含 `upgrade.sh` 脚本的机密，该脚本负责创建 systemd.service（第 35.2.4.1.1 节 “systemd.service”）。
- `config-map.yaml` 是 ConfigMap，提供 `upgrade.sh` 脚本使用的升级配置。

重要

这些计划资源由系统升级控制器解释，应部署在您要升级的每个下游群集上。有关 SUC 部署信息，请参见第 22.2 节 “安装系统升级控制器”。

为了更好地了解如何使用 GitOps 工作流程来部署操作系统升级的 **SUC 计划**，建议查看第 35.2.4.2 节 “概述”。

35.2.5 Kubernetes 版本升级

本节介绍如何使用第 8 章 “Fleet”和第 22 章 “系统升级控制器”执行 Kubernetes 升级。

本节将介绍以下主题：

1. 第 35.2.5.1 节 “组件” - 升级过程使用的其他组件。
2. 第 35.2.5.2 节 “概述” - 升级过程概述。
3. 第 35.2.5.3 节 “要求” - 升级过程的要求。
4. 第 35.2.5.4 节 “K8s 升级 - SUC 计划部署” - 关于如何部署负责触发升级过程的 SUC 计划的信息。

35.2.5.1 组件

本节介绍 K8s 升级过程中使用的自定义组件，这些组件与默认“Day 2”组件（第 35.2.1 节“组件”）不同。

35.2.5.1.1 rke2-upgrade

容器映像负责升级特定节点的 RKE2 版本。

此组件由 **SUC** 根据 **SUC 计划**创建的 Pod 分发。该计划应位于需要进行 RKE2 升级的每个**群集**上。

有关 `rke2-upgrade` 映像如何执行升级的详细信息，请参见上游 (https://github.com/rancher/rke2-upgrade/tree/master) 文档。

35.2.5.1.2 k3s-upgrade

容器映像负责升级特定节点的 K3s 版本。

此组件由 **SUC** 根据 **SUC 计划**创建的 Pod 分发。该计划应位于需要进行 K3s 升级的每个**群集**上。

有关 `k3s-upgrade` 映像如何执行升级的详细信息，请参见上游 (https://github.com/k3s-io/k3s-upgrade) 文档。

35.2.5.2 概述

通过 `Fleet` 和系统升级控制器（SUC）来为管理群集节点升级 Kubernetes 发行版。

`Fleet` 用于将 SUC 计划部署到目标群集并对其进行管理。

注意

SUC 计划是一种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)，描述了 **SUC** 为在一组节点上执行特定任务而需要遵循的步骤。有关 SUC 计划的示例，请参见上游储存库 (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans)。

通过向特定 Fleet 工作区 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 部署 GitRepo (https://fleet.rancher.io/gitrepo-add) 或捆绑包 (https://fleet.rancher.io/bundle-add) 资源将 K8s SUC 计划分发到各个群集。Fleet 会获取已部署的 GitRepo/捆绑包，并将其内容（K8s SUC 计划）部署到目标群集。

注意

GitRepo/捆绑包资源始终部署在管理群集上。使用 GitRepo 还是捆绑包资源取决于具体应用场景，有关详细信息，请参见第 35.2.2 节 “确定您的使用场景”。

K8s SUC 计划描述了以下工作流程：

1. 执行 K8s 升级前，务必要封锁 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) 节点。
2. 务必先升级控制平面节点，再升级工作节点。
3. 始终一次升级**一个**控制平面节点，一次升级**两个**工作节点。

部署 K8s SUC 计划后，工作流程如下：

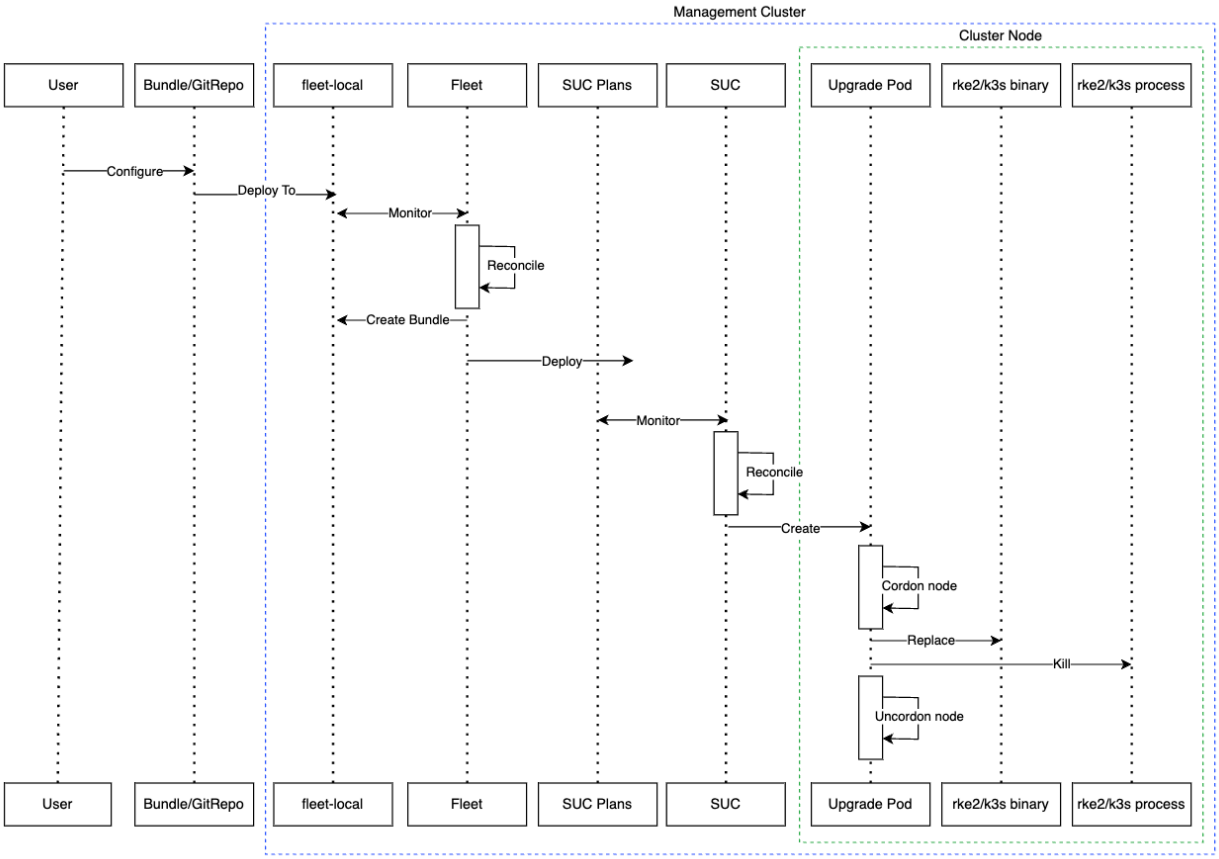
1. SUC 协调已部署的 K8s SUC 计划，并在**每个节点**上创建一个 Kubernetes 作业。
2. 根据 Kubernetes 发行版的不同，该作业将创建一个运行 rke2-upgrade（第 35.2.5.1.1 节 “rke2-upgrade”）或 k3s-upgrade（第 35.2.5.1.2 节 “k3s-upgrade”）容器映像的 Pod。
3. 创建的 Pod 将执行以下工作流程：

a. 用 `rke2-upgrade/k3s-upgrade` 映像中的二进制文件替换节点上现有的 `rke2/k3s` 二进制文件。

b. 终止正在运行的 `rke2/k3s` 进程。

4. 终止 `rke2/k3s` 进程会触发重启，启动运行已更新二进制文件的新进程，从而实现 Kubernetes 发行版版本的升级。

下面是上述流程的示意图：



35.2.5.3 要求

1. **备份您的 Kubernetes 发行版：**

a. 对于 **RKE2 群集**，请参见 RKE2 Backup and Restore (https://docs.rke2.io/datastore/backup_restore) 文档。

b. 对于 **K3s 群集**，请参见 K3s Backup and Restore (https://docs.k3s.io/datastore/backup-restore) 文档。

2. **确保 SUC 计划容忍度与节点容忍度相匹配** - 如果您的 Kubernetes 群集节点具有自定义**污点**，请确保在 **SUC 计划**中为这些污点添加容忍度 (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)。默认情况下，**SUC 计划**仅包含**控制平面**节点的容忍度。默认容忍度包括：

   - **CriticalAddonsOnly=true:NoExecute**
   - **node-role.kubernetes.io/control-plane:NoSchedule**
   - **node-role.kubernetes.io/etcd:NoExecute**

   ### 注意

   其他任何容忍度必须添加到每个计划的 `.spec.tolerations` 部分下。与 Kubernetes 版本升级相关的 **SUC 计划**可以在 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库中的以下位置找到：

   - 对于 **RKE2** - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade`
   - 对于 **K3s** - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade`

   请确保使用有效储存库版本 (https://github.com/suse-edge/fleet-examples/releases)标记中的计划。

   为 RKE2 **控制平面** SUC 计划定义自定义容忍度的示例如下：

   ```
   apiVersion: upgrade.cattle.io/v1
   kind: Plan
   metadata:
   ```

```
  name: rke2-upgrade-control-plane
spec:
  ...
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

35.2.5.4 K8s 升级 - SUC 计划部署

重要

对于之前使用此过程升级的环境，用户应确保完成以下步骤之一：

- 从管理群集中去除任何先前部署且与旧版 Edge 相关的 SUC 计划 - 方法是从现有的 GitRepo/捆绑包目标配置 (https://fleet.rancher.io/gitrepo-targets#target-matching) 中去除相应群集，或完全去除 GitRepo/捆绑包资源。
- 重用现有的 GitRepo/捆绑包资源 - 方法是将资源的修订版指向一个新标签，该标签包含目标 suse-edge/fleet-examples 版本 (https://github.com/suse-edge/fleet-examples/releases) 的正确 Fleet。

这样做是为了避免旧版 Edge 的 SUC 计划之间发生冲突。

如果用户尝试升级，而管理群集上存在现有的 SUC 计划，他们将看到以下 Fleet 错误：

```
Not installed: Unable to continue with install: Plan <plan_name> in
 namespace <plan_namespace> exists and cannot be imported into the current
 release: invalid ownership metadata; annotation validation error..
```

如第 35.2.5.2 节 “概述”中所述，可以通过以下任意一种方式将 SUC 计划分发到目标群集来完成 Kubernetes 升级：

- Fleet GitRepo 资源（第 35.2.5.4.1 节 “SUC 计划部署 - GitRepo 资源”）
- Fleet 捆绑包资源（第 35.2.5.4.2 节 “SUC 计划部署 - 捆绑包资源”）

要确定使用哪个资源，请参见第 35.2.2 节 “确定您的使用场景”。

对于希望通过第三方 GitOps 工具部署 K8s SUC 计划的使用场景，请参见第 35.2.5.4.3 节 “SUC 计划部署 - 第三方 GitOps 工作流程”。

35.2.5.4.1 SUC 计划部署 - GitRepo 资源

提供所需 K8s SUC 计划的 **GitRepo** 资源可通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 35.2.5.4.1.1 节 “GitRepo 创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 35.2.5.4.1.2 节 “GitRepo 创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的 Kubernetes 升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

35.2.5.4.1.1 GitRepo 创建 - Rancher UI

要通过 Rancher UI 创建 GitRepo 资源，请遵循其官方文档 (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

Edge 团队为 rke2 (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/rke2-upgrade) 和 k3s (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/k3s-upgrade) 这两种 Kubernetes 发行版维护着即用型 Fleet。根据您的环境的不同，这些 Fleet 可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记使用这些 Fleet。

对于不需要在这些 Fleet 附带的 SUC 计划中包含自定义更改的使用场景，用户可以直接引用 `suse-edge/fleet-examples` 储存库中的 Fleet。

如果需要自定义更改（例如添加自定义容忍度），用户应从单独的储存库中引用 Fleet，以便能够根据需要将更改添加到 SUC 计划中。

使用 `suse-edge/fleet-examples` 储存库中的 Fleet 配置 `GitRepo` 资源的示例：

- RKE2 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml)
- K3s (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/k3s-upgrade-gitrepo.yaml)

35.2.5.4.1.2 GitRepo 创建 - 手动

1. 提取 **GitRepo** 资源：
    - 对于 **RKE2** 群集：

      ```
      curl -o rke2-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml
      ```

    - 对于 **K3s** 群集：

```
curl -o k3s-upgrade-gitrepo.yaml https://raw.githubusercontent.com/
suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/k3s-
upgrade-gitrepo.yaml
```

2. 编辑 **GitRepo** 配置：

   - 去除 `spec.targets` 部分 - 该部分仅用于下游群集。

     - 对于 RKE2，使用以下命令：

       ```
       # Example using sed
       sed -i.bak '/^  targets:/,$d' rke2-upgrade-gitrepo.yaml && rm -f
        rke2-upgrade-gitrepo.yaml.bak

       # Example using yq (v4+)
       yq eval 'del(.spec.targets)' -i rke2-upgrade-gitrepo.yaml
       ```

     - 对于 K3s，使用以下命令：

       ```
       # Example using sed
       sed -i.bak '/^  targets:/,$d' k3s-upgrade-gitrepo.yaml && rm -f
        k3s-upgrade-gitrepo.yaml.bak

       # Example using yq (v4+)
       yq eval 'del(.spec.targets)' -i k3s-upgrade-gitrepo.yaml
       ```

   - 将 `GitRepo` 的名称空间指向 `fleet-local` 名称空间 - 这样做是为了在管理群集上部署该资源。

     - 对于 RKE2，使用以下命令：

       ```
       # Example using sed
       sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/'
        rke2-upgrade-gitrepo.yaml && rm -f rke2-upgrade-gitrepo.yaml.bak

       # Example using yq (v4+)
       ```

```
yq eval '.metadata.namespace = "fleet-local"' -i rke2-upgrade-
gitrepo.yaml
```

- 对于 K3s，使用以下命令：

```
# Example using sed
sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/'
 k3s-upgrade-gitrepo.yaml && rm -f k3s-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
yq eval '.metadata.namespace = "fleet-local"' -i k3s-upgrade-
gitrepo.yaml
```

3. 将 **GitRepo** 资源应用于管理群集：

```
# RKE2
kubectl apply -f rke2-upgrade-gitrepo.yaml

# K3s
kubectl apply -f k3s-upgrade-gitrepo.yaml
```

4. 查看 `fleet-local` 名称空间下创建的 **GitRepo** 资源：

```
# RKE2
kubectl get gitrepo rke2-upgrade -n fleet-local

# K3s
kubectl get gitrepo k3s-upgrade -n fleet-local

# Example output
NAME           REPO                                                COMMIT
      BUNDLEDEPLOYMENTS-READY   STATUS
k3s-upgrade    https://github.com/suse-edge/fleet-examples.git   fleet-
local   0/0
rke2-upgrade   https://github.com/suse-edge/fleet-examples.git   fleet-
local   0/0
```

35.2.5.4.2 SUC 计划部署 - 捆绑包资源

可通过以下方式之一部署**捆绑包**资源，该资源中附带了所需的 Kubernetes 升级 SUC 计划：

1. 通过 Rancher UI 部署 - 第 35.2.5.4.2.1 节 “捆绑包创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 35.2.5.4.2.2 节 “捆绑包创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的 Kubernetes 升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

35.2.5.4.2.1 捆绑包创建 - Rancher UI

Edge 团队为 rke2 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) 和 k3s (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) 这两种 Kubernetes 发行版维护着即用型捆绑包。根据您的环境的不同，这些捆绑包可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记使用此捆绑包。

要通过 Rancher 的 UI 创建捆绑包，请执行以下操作：

1. 单击左上角的 **#** → **Continuous Delivery**（持续交付）
2. 转到 **Advanced**（高级）> **Bundles**（捆绑包）
3. 选择 **Create from YAML**（基于 YAML 创建）
4. 此处可通过以下方式之一创建捆绑包：

注意

在某些使用场景中，您可能需要在捆绑包附带的 SUC 计划中包含自定义更改。请确保在以下步骤生成的捆绑包中包含这些更改。

a. 手动将 RKE2 (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) 或 K3s (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) 的捆绑包内容从 `suse-edge/fleet-examples` 复制到 **Create from YAML**（基于 YAML 创建）页面。

b. 从所需的版本 (https://github.com/suse-edge/fleet-examples/releases) 标记克隆 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) 储存库，并在 **Create from YAML**（基于 YAML 创建）页面中选择 **Read from File**（从文件读取）选项。然后导航到所需的捆绑包（对于 RKE2，为 `bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml`；对于 K3s，为 `bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml`）。这会在 **Create from YAML**（基于 YAML 创建）页面中自动填充捆绑包内容。

5. 在 Rancher UI 中编辑捆绑包：

- 更改捆绑包的**名称空间**，使其指向 `fleet-local` 名称空间。

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: rke2-upgrade
  namespace: fleet-local
...
```

- 更改捆绑包的**目标**群集，使其指向本地（管理）群集：

```
spec:
  targets:
  - clusterName: local
```

注意

在某些使用场景中，您的本地群集的名称可能会有所不同。

要获取本地群集名称，请执行以下命令：

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

6. 选择 **Create**（创建）

35.2.5.4.2.2 捆绑包创建 - 手动

1. 提取**捆绑包**资源：

   - 对于 **RKE2** 群集：

   ```
   curl -o rke2-plan-bundle.yaml https://raw.githubusercontent.com/
   suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-
   upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml
   ```

   - 对于 **K3s** 群集：

   ```
   curl -o k3s-plan-bundle.yaml https://raw.githubusercontent.com/suse-
   edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-
   upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml
   ```

2. 编辑捆绑包配置：

   - 更改捆绑包的**目标**群集，使其指向本地（管理）群集：

   ```
   spec:
     targets:
     - clusterName: local
   ```

注意

在某些使用场景中，您的本地群集的名称可能会有所不同。

要获取本地群集名称，请执行以下命令：

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

- 更改捆绑包的**名称空间**，使其指向 `fleet-local` 名称空间。

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: rke2-upgrade
  namespace: fleet-local
...
```

3. 将**捆绑包**资源应用于管理群集：

```
# For RKE2
kubectl apply -f rke2-plan-bundle.yaml

# For K3s
kubectl apply -f k3s-plan-bundle.yaml
```

4. 查看 `fleet-local` 名称空间下创建的**捆绑包**资源：

```
# For RKE2
kubectl get bundles rke2-upgrade -n fleet-local

# For K3s
kubectl get bundles k3s-upgrade -n fleet-local

# Example output
NAME           BUNDLEDEPLOYMENTS-READY   STATUS
k3s-upgrade    0/0
```

```
rke2-upgrade   0/0
```

35.2.5.4.3 SUC 计划部署 - 第三方 GitOps 工作流程

在某些使用场景中，用户可能希望将 Kubernetes 升级 SUC 计划合并到自己的第三方 GitOps 工作流程（例如 Flux）中。

要获取所需的 K8s 升级资源，首先请确定您要使用的 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) 储存库的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记。

然后，便可以在以下位置找到资源：

- 对于 RKE2 群集升级：
  - 对于控制平面节点 - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-control-plane.yaml`
  - 对于工作节点 - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-agent.yaml`
- 对于 K3s 群集升级：
  - 对于控制平面节点 - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-control-plane.yaml`
  - 对于工作节点 - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-agent.yaml`

重要

这些计划资源由系统升级控制器解释，应部署在您要升级的每个下游群集上。有关 SUC 部署信息，请参见第 22.2 节 “安装系统升级控制器”。

为了更好地了解如何使用 GitOps 工作流程来部署 Kubernetes 版本升级的 **SUC 计划**，建议查看有关使用 Fleet 进行更新的过程概述（第 35.2.5.2 节 “概述”）。

35.2.6 Helm chart 升级

本节介绍如下内容：

1. 第 35.2.6.1 节 “为隔离环境做好准备” - 包含有关如何将与 Edge 相关的 OCI chart 和映像分发到您的专用注册表的信息。
2. 第 35.2.6.2 节 “升级过程” - 包含有关不同 Helm chart 升级情形及其升级过程的信息。

35.2.6.1 为隔离环境做好准备

35.2.6.1.1 确保您有权访问 Helm chart 的 Fleet

根据环境支持的情况，选择以下选项之一：

1. 将 chart 的 Fleet 资源托管在管理群集可访问的本地 git 服务器上。
2. 使用 Fleet 的 CLI 将 Helm chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle)，以便直接使用，而不必托管于某处。Fleet 的 CLI 可以从其版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2)页面中检索，对于 Mac 用户，有一个 fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae。

35.2.6.1.2 找到 Edge 发行版本的所需资产

1. 转到“Day 2”版本 (https://github.com/suse-edge/fleet-examples/releases)页面，找到您要将 chart 升级到的 Edge 版本，然后单击 **Assets**（资产）。
2. 从“**Assets**”（资产）部分下载以下文件：

版本文件	说明
edge-save-images.sh	提取 `edge-release-images.txt` 文件中指定的映像，并将其封装到“.tar.gz”归档中。

edge-save-oci-artefacts.sh	提取与特定 Edge 版本相关的 OCI chart 映像，并将其封装到“.tar.gz”归档中。
edge-load-images.sh	从“.tar.gz”归档加载映像，将它们重新标记并推送到专用注册表。
edge-load-oci-artefacts.sh	接收包含 Edge OCI '.tgz' chart 软件包的目录作为参数，并将这些软件包加载到专用注册表中。
edge-release-helm-oci-artefacts.txt	包含与特定 Edge 版本相关的 OCI chart 映像的列表。
edge-release-images.txt	包含与特定 Edge 版本相关的映像列表。

35.2.6.1.3 创建 Edge 版本映像归档

在可以访问互联网的计算机上：

1. 将 `edge-save-images.sh` 设为可执行文件：

```
chmod +x edge-save-images.sh
```

2. 生成映像归档：

```
./edge-save-images.sh --source-registry registry.suse.com
```

3. 这将创建一个名为 `edge-images.tar.gz` 的可加载归档。

注意

如果指定了 `-i|--images` 选项，归档的名称可能会不同。

4. 将此归档复制到**隔离的**计算机：

```
scp edge-images.tar.gz <user>@<machine_ip>:/path
```

35.2.6.1.4 创建 Edge OCI chart 映像归档

在可以访问互联网的计算机上：

1. 将 `edge-save-oci-artefacts.sh` 设为可执行文件：

```
chmod +x edge-save-oci-artefacts.sh
```

2. 生成 OCI chart 映像归档：

```
./edge-save-oci-artefacts.sh --source-registry registry.suse.com
```

3. 这将创建一个名为 `oci-artefacts.tar.gz` 的归档。

注意

如果指定了 `-a|--archive` 选项，归档的名称可能会不同。

4. 将此归档复制到**隔离的**计算机：

```
scp oci-artefacts.tar.gz <user>@<machine_ip>:/path
```

35.2.6.1.5 将 Edge 版本映像加载到隔离的计算机上

在隔离的计算机上：

1. 登录到专用注册表（如果需要）：

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. 将 `edge-load-images.sh` 设为可执行文件：

```
chmod +x edge-load-images.sh
```

3. 执行脚本，传递之前**复制的** `edge-images.tar.gz` 归档：

```
./edge-load-images.sh --source-registry registry.suse.com --registry
 <REGISTRY.YOURDOMAIN.COM:PORT> --images edge-images.tar.gz
```

注意

这将从 `edge-images.tar.gz` 加载所有映像，并将它们重新标记并推送到 `--registry` 选项下指定的注册表。

35.2.6.1.6 将 Edge OCI chart 映像加载到隔离的计算机上

在隔离的计算机上：

1. 登录到专用注册表（如果需要）：

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. 将 `edge-load-oci-artefacts.sh` 设为可执行文件：

```
chmod +x edge-load-oci-artefacts.sh
```

3. 解压缩复制的 `oci-artefacts.tar.gz` 归档：

```
tar -xvf oci-artefacts.tar.gz
```

4. 这会使用命名模板 `edge-release-oci-tgz-<date>` 生成一个目录

5. 将此目录传递给 `edge-load-oci-artefacts.sh` 脚本，以将 Edge OCI chart 映像加载到专用注册表中：

注意

此脚本假设您的环境中已预装了 `Helm` CLI。有关 Helm 安装说明，请参见安装 Helm (https://helm.sh/docs/intro/install/)。

```
./edge-load-oci-artefacts.sh --archive-directory edge-release-oci-tgz-
<date> --registry <REGISTRY.YOURDOMAIN.COM:PORT> --source-registry
 registry.suse.com
```

35.2.6.1.7 在 Kubernetes 发行版中配置专用注册表

对于 RKE2，请参见 Private Registry Configuration (https://docs.rke2.io/install/private_registry)

对于 K3s，请参见 Private Registry Configuration (https://docs.k3s.io/installation/private-registry)

35.2.6.2 升级过程

本节重点介绍以下使用场景的 Helm 升级过程：

1. 第 35.2.6.2.1 节 “我有一个新群集，想要部署和管理 Edge Helm chart”
2. 第 35.2.6.2.2 节 “我想升级 Fleet 管理的 Helm chart”
3. 第 35.2.6.2.3 节 “我要升级通过 EIB 部署的 Helm chart”

重要

手动部署的 Helm chart 无法可靠升级。我们建议使用第 35.2.6.2.1 节 “我有一个新群集，想要部署和管理 Edge Helm chart”中所述的方法重新部署这些 Helm chart。

35.2.6.2.1 我有一个新群集，想要部署和管理 Edge Helm chart

本节介绍如何：

1. 第 35.2.6.2.1.1 节 “为您的 chart 准备 Fleet 资源”。
2. 第 35.2.6.2.1.2 节 “部署您的 chart 的 Fleet”。
3. 第 35.2.6.2.1.3 节 “管理部署的 Helm chart”。

35.2.6.2.1.1 为您的 chart 准备 Fleet 资源

1. 从您要使用的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases)标记处获取 Chart 的 Fleet 资源。
2. 导航到 Helm chart Fleet (`fleets/day2/chart-templates/<chart>`)

3. **如果您打算使用 GitOps 工作流程**，请将 chart Fleet 目录复制到 Git 储存库，从那里执行 GitOps。

4. **（可选）**如果需要配置 Helm chart 的**值**才能使用 Helm chart，请编辑复制的目录中 `fleet.yaml` 文件内的 `.helm.values` 配置。

5. **（可选）**在某些使用场景中，您可能需要向 chart 的 Fleet 目录添加其他资源，使该目录能够更好地适应您的环境。有关如何增强 Fleet 目录的信息，请参见 Git 储存库内容 (https://fleet.rancher.io/gitrepo-content)。

注意

在某些情况下，Fleet 为 Helm 操作设置的默认超时时长可能不足，导致发生以下错误：

```
failed pre-install: context deadline exceeded
```

在此类情况下，请在 `fleet.yaml` 文件的 `helm` 配置下添加 timeoutSeconds (https://fleet.rancher.io/ref-crds#helmoptions) 属性。

`longhorn` Helm chart 的**示例**如下：

- 用户 Git 储存库结构：

```
<user_repository_root>
├── longhorn
│   └── fleet.yaml
└── longhorn-crd
    └── fleet.yaml
```

- 填充了用户 `Longhorn` 数据的 `fleet.yaml` 内容：

```
defaultNamespace: longhorn-system

helm:
  # timeoutSeconds: 10
  releaseName: "longhorn"
  chart: "longhorn"
  repo: "https://charts.rancher.io/"
```

```
  version: "106.2.0+up1.8.1"
  takeOwnership: true
  # custom chart value overrides
  values:
    # Example for user provided custom values content
    defaultSettings:
      deletingConfirmationFlag: true

# https://fleet.rancher.io/bundle-diffs
diff:
  comparePatches:
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: engineimages.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: nodes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: volumes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
```

注意

上面只是一些示例值，用于演示基于 `longhorn` chart 创建的自定义配置。**不应**将它们视为 `longhorn` chart 的部署指南。

35.2.6.2.1.2 部署您的 chart 的 Fleet

您可以使用 GitRepo（第 35.2.6.2.1.2.1 节 “GitRepo”）或捆绑包（第 35.2.6.2.1.2.2 节 “捆绑包”）来部署 chart 的 Fleet。

注意

部署 Fleet 时，如果收到资源已修改消息，请确保在 Fleet 的 `diff` 部分添加相应的 `comparePatches` 项。有关详细信息，请参见 Generating Diffs to Ignore Modified GitRepos (https://fleet.rancher.io/bundle-diffs)。

35.2.6.2.1.2.1 GitRepo

Fleet 的 GitRepo (https://fleet.rancher.io/ref-gitrepo) 资源包含如何访问 chart 的 Fleet 资源以及需要将这些资源应用于哪些群集的相关信息。

可以通过 Rancher UI (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui) 或者通过将资源手动部署到管理群集来部署 (https://fleet.rancher.io/tut-deployment) `GitRepo` 资源。

用于**手动**部署的 **Longhorn** `GitRepo` 资源示例：

```
apiVersion: fleet.cattle.io/v1alpha1
kind: GitRepo
metadata:
  name: longhorn-git-repo
  namespace: fleet-local
spec:
  # If using a tag
  # revision: user_repository_tag
  #
  # If using a branch
  # branch: user_repository_branch
  paths:
  # As seen in the 'Prepare your Fleet resources' example
  - longhorn
  - longhorn-crd
  repo: user_repository_url
```

35.2.6.2.1.2.2 捆绑包

捆绑包 (https://fleet.rancher.io/bundle-add) 资源包含需要由 Fleet 部署的原始 Kubernetes 资源。一般情况下，建议使用 `GitRepo` 方法，但对于无法支持本地 Git 服务器的隔离环境，捆绑包可以帮助您将 Helm chart Fleet 传播到目标群集。

捆绑包的部署可以通过 Rancher UI（`Continuous Delivery`（持续交付）→ Advanced（高级）→ Bundles（捆绑包）→ `Create from YAML`（基于 YAML 创建））进行，也可以通过在正确的 Fleet 名称空间中手动部署捆绑包资源来完成。有关 Fleet 名称空间的信息，请参见上游文档 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups)。

可以利用 Fleet 的将 Helm Chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle) 方法创建用于 Edge Helm chart 的捆绑包。

下面的示例展示了如何基于 longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) 和 longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) Helm chart Fleet 模板创建捆绑包资源，以及如何手动将此捆绑包部署到管理群集。

注意

为了阐明工作流程，下面的示例使用了 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 目录结构。

1. 导航到 longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) chart Fleet 模板：

```
cd fleets/day2/chart-templates/longhorn/longhorn
```

2. 创建 `targets.yaml` 文件，该文件将指示 Fleet 应将 Helm chart 部署到哪些群集：

```
cat > targets.yaml <<EOF
targets:
# Match your local (management) cluster
- clusterName: local
EOF
```

注意

在某些使用场景中，您的本地群集的名称可能会有所不同。

要获取本地群集名称，请执行以下命令：

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

3. 使用 fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet) 将 `Longhorn` Helm chart Fleet 转换为捆绑包资源。

注意

Fleet 的 CLI 可以从其版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2) **资产**页面 (`fleet-linux-amd64`) 获取。

对于 Mac 用户，有一个 fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae。

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o -
 longhorn-bundle > longhorn-bundle.yaml
```

4. 导航到 longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) chart Fleet 模板：

```
cd fleets/day2/chart-templates/longhorn/longhorn-crd
```

5. 创建 `targets.yaml` 文件，该文件将指示 Fleet 应将 Helm chart 部署到哪些群集：

```
cat > targets.yaml <<EOF
targets:
# Match your local (management) cluster
- clusterName: local
EOF
```

6. 使用 fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet) 将 Longhorn CRD Helm chart Fleet 转换为捆绑包资源。

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o -
 longhorn-crd-bundle > longhorn-crd-bundle.yaml
```

7. 将 longhorn-bundle.yaml 和 longhorn-crd-bundle.yaml 文件部署到管理群集：

```
kubectl apply -f longhorn-crd-bundle.yaml
kubectl apply -f longhorn-bundle.yaml
```

按照以上步骤操作，将 SUSE Storage 部署到所有指定的管理群集上。

35.2.6.2.1.3 管理部署的 Helm chart

通过 Fleet 完成部署后，若要进行 Helm chart 升级，请参见第 35.2.6.2.2 节 “我想升级 Fleet 管理的 Helm chart”。

35.2.6.2.2 我想升级 Fleet 管理的 Helm chart

1. 确定需要将 chart 升级到哪个版本，以使其与目标 Edge 版本兼容。有关每个 Edge 版本兼容的 Helm chart 版本，可参见发行说明（第 52.1 节 “摘要”）。
2. 在 Fleet 监控的 Git 储存库中，根据发行说明（第 52.1 节 “摘要”）中所述，将 Helm chart 的 fleet.yaml 文件中的 chart **版本**和**储存库**更改为正确的值。
3. 提交更改并将其推送到储存库后，会触发所需 Helm chart 的升级

35.2.6.2.3 我要升级通过 EIB 部署的 Helm chart

第 11 章 “Edge Image Builder” 通过创建 HelmChart 资源并利用 RKE2 (https://docs.rke2.io/helm)/K3s (https://docs.k3s.io/helm) Helm 集成功能引入的 helm-controller 来部署 Helm chart。

为确保通过 EIB 部署的 Helm chart 成功升级，用户需要对相应的 HelmChart 资源执行升级操作。

下文提供了以下信息：

- 升级过程的一般概述（第 35.2.6.2.3.1 节 “概述”）。
- 必要的升级步骤（第 35.2.6.2.3.2 节 “升级步骤”）。
- 展示使用所述方法进行 Longhorn (https://longhorn.io) chart 升级的示例（第 35.2.6.2.3.3 节 “示例”）。
- 如何使用其他 GitOps 工具完成升级过程（第 35.2.6.2.3.4 节 “使用第三方 GitOps 工具进行 Helm chart 升级”）。

35.2.6.2.3.1 概述

通过 EIB 部署的 Helm chart 由名为 eib-charts-upgrader (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/eib-charts-upgrader) 的 Fleet 执行升级。

该 Fleet 会处理**用户提供的**数据，以**更新**一组特定的 HelmChart 资源。

更新这些资源会触发 helm-controller (https://github.com/k3s-io/helm-controller)，后者会**升级**与修改后的 HelmChart 资源相关联的 Helm chart。

用户只需执行以下操作：

1. 从本地提取 (https://helm.sh/docs/helm/helm_pull/) 需要升级的每个 Helm chart 的归档。
2. 将这些归档传递给 generate-chart-upgrade-data.sh (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/generate-chart-upgrade-data.sh) `generate-chart-upgrade-data.sh` 脚本，该脚本会将这些归档中的数据包含到 `eib-charts-upgrader` Fleet 中。
3. 将 `eib-charts-upgrader` Fleet 部署到管理群集。此操作可以通过 GitRepo 或捆绑包资源来完成。

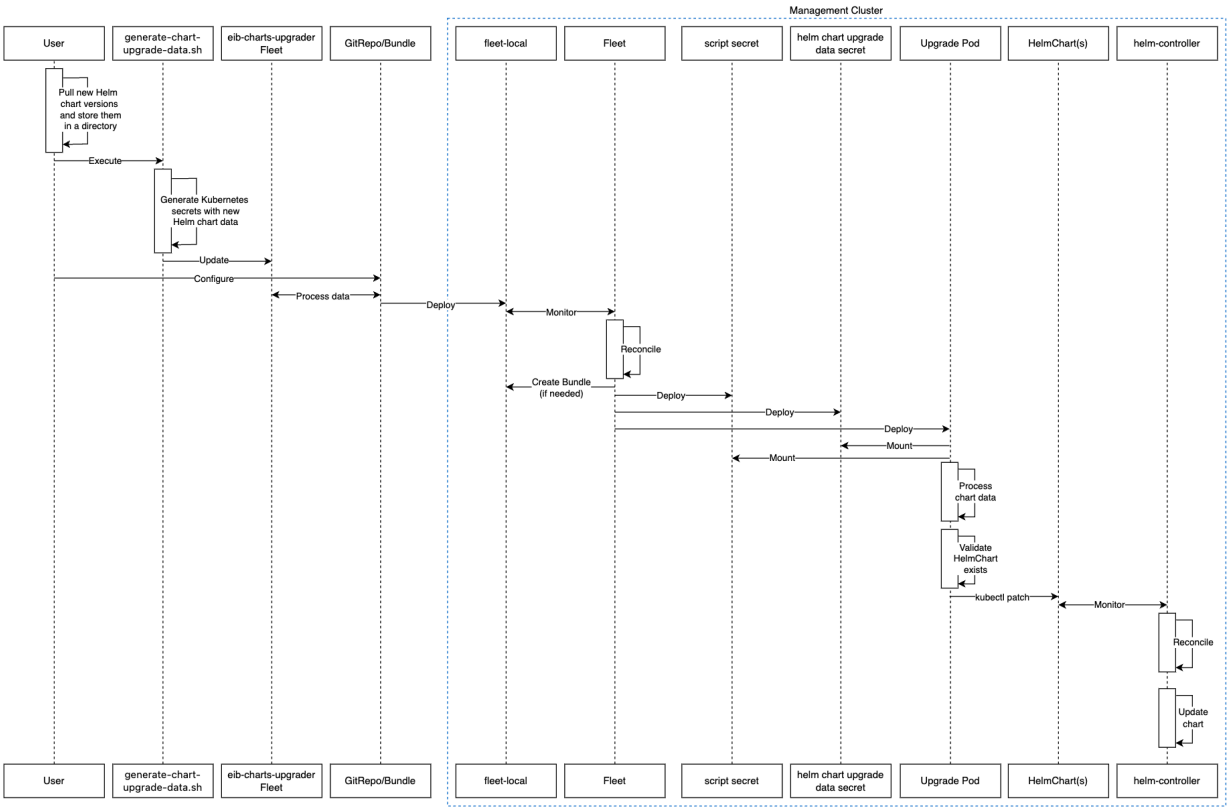
部署后，`eib-charts-upgrader` 会借助 Fleet 将其资源分发到目标管理群集。

这些资源包括：

1. 一组存储**用户提供的** Helm chart 数据的机密。
2. 一个会部署 Pod 的 Kubernetes 作业，该 Pod 会挂载前面提到的机密，并根据这些机密对相应的 HelmChart 资源进行修补 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_patch/)。

如前文所述，这会触发 `helm-controller`，进而执行实际的 Helm chart 升级。

下面是上述流程的示意图：



35.2.6.2.3.2 升级步骤

1. 从正确的版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0) 处克隆 `suse-edge/fleet-example` 储存库。
2. 创建用于存储所提取的 Helm chart 归档的目录。

```
mkdir archives
```

3. 在新创建的归档目录中，提取 (https://helm.sh/docs/helm/helm_pull/)要升级的 Helm chart 的归档：

```
cd archives
helm pull [chart URL | repo/chartname]

# Alternatively if you want to pull a specific version:
# helm pull [chart URL | repo/chartname] --version 0.0.0
```

4. 从目标版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0)的**资产**中下载 `generate-chart-upgrade-data.sh` 脚本。

5. 执行 `generate-chart-upgrade-data.sh` 脚本：

```
chmod +x ./generate-chart-upgrade-data.sh

./generate-chart-upgrade-data.sh --archive-dir /foo/bar/archives/ --fleet-path /foo/bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

对于 `--archive-dir` 目录中的每个 chart 归档，该脚本都会生成一个包含 chart 升级数据的 Kubernetes 机密 YAML 文件，并将其存储在 `--fleet-path` 指定的 Fleet 的 `base/secrets` 目录中。
`generate-chart-upgrade-data.sh` 脚本还会对 Fleet 进行其他修改，以确保生成的 Kubernetes 机密 YAML 文件能被 Fleet 部署的工作负载正确使用。

重要

用户不应更改 `generate-chart-upgrade-data.sh` 脚本生成的内容。

以下步骤因您运行的环境而异：

1. 对于支持 GitOps 的环境（例如：非隔离环境，或虽是隔离环境但支持本地 Git 服务器）：

   a. 将 `fleets/day2/eib-charts-upgrader` Fleet 复制到将用于 GitOps 的储存库中。

注意

确保该 Fleet 包含 `generate-chart-upgrade-data.sh` 脚本所做的更改。

b. 配置将用于提供 `eib-charts-upgrader` Fleet 所有资源的 `GitRepo` 资源。

i. 要通过 Rancher UI 进行 `GitRepo` 配置和部署，请参见在 Rancher UI 中访问 Fleet (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

ii. 有关 `GitRepo` 的手动配置和部署过程，请参见创建部署 (https://fleet.rancher.io/tut-deployment)。

2. 对于不支持 GitOps 的环境（例如，不允许使用本地 Git 服务器的隔离环境）：

a. 从 `rancher/fleet` 版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2) 页面下载 `fleet-cli` 二进制文件。对于 Linux，请下载 `fleet-linux-amd64`。对于 Mac 用户，可以使用 Homebrew Formulae - fleet-cli (https://formulae.brew.sh/formula/fleet-cli)。

b. 导航到 `eib-charts-upgrader` Fleet：

```
cd /foo/bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

c. 创建指示 Fleet 在哪里部署资源的 `targets.yaml` 文件：

```
cat > targets.yaml <<EOF
targets:
# To map the local(management) cluster
- clusterName: local
EOF
```

注意

在某些使用场景中，您的本地群集的名称可能会有所不同。

要获取本地群集名称，请执行以下命令：

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

d. 使用 `fleet-cli` 将 Fleet 转换为捆绑包资源：

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o -
 eib-charts-upgrade > bundle.yaml
```

这将创建一个捆绑包 (`bundle.yaml`)，其中包含来自 `eib-charts-upgrader` Fleet 的所有模板资源。
有关 `fleet apply` 命令的详细信息，请参见 fleet apply (https://fleet.rancher.io/cli/fleet-cli/fleet_apply)。
有关将 Fleet 转换为捆绑包的详细信息，请参见将 Helm Chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle)。

e. 通过以下方式之一部署捆绑包：

i. 通过 Rancher UI - 导航到 **Continuous Delivery（持续交付）→ Advanced（高级）→ Bundles（捆绑包）→ Create from YAML（基于 YAML 创建）**，然后粘贴 `bundle.yaml` 内容，或单击 `Read from File`（从文件读取）选项并传递文件。

ii. 手动 - 在管理群集内手动部署 `bundle.yaml` 文件。

执行这些步骤可成功部署 `GitRepo`/捆绑包资源。资源将由 Fleet 拾取，且其内容将部署到用户在之前的步骤中指定的目标群集上。有关该过程的概述，请参见第 35.2.6.2.3.1 节 “概述”。
有关如何跟踪升级过程的信息，请参见第 35.2.6.2.3.3 节 “示例”。

重要

成功验证 chart 升级后，去除捆绑包/`GitRepo` 资源。

这将从管理群集中去除不再需要的升级资源，确保将来不会发生版本冲突。

35.2.6.2.3.3 示例

注意

以下示例展示了如何在管理群集上将通过 EIB 部署的 Helm chart 从一个版本升级到另一个版本。请注意，本示例中使用的版本**并非**推荐版本。有关特定 Edge 版本的推荐版本，请参见发行说明（第 52.1 节 “摘要”）。

使用场景：

- 某管理群集正在运行旧版 Longhorn (https://longhorn.io)。
- 已使用以下映像定义**代码段**通过 EIB 部署群集：

```
kubernetes:
  helm:
    charts:
    - name: longhorn-crd
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    - name: longhorn
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    repositories:
    - name: rancher-charts
      url: https://charts.rancher.io/
...
```

- SUSE Storage 需要升级到与 Edge 3.3.1 版兼容的版本。这意味着它需要升级到 `106.2.0+up1.8.1`。
- 假设管理群集是**隔离式**群集，不支持本地 Git 服务器，并且具有有效的 Rancher 设置。

按照升级步骤（第 35.2.6.2.3.2 节 “升级步骤”）进行操作：

1. 从 `release-3.3.0` 标记处克隆 `suse-edge/fleet-example` 储存库。

```
git clone -b release-3.3.0 https://github.com/suse-edge/fleet-examples.git
```

2. 创建用于存储 `Longhorn` 升级归档的目录。

```
mkdir archives
```

3. 提取所需的 `Longhorn` chart 归档版本：

```
# First add the Rancher Helm chart repository
helm repo add rancher-charts https://charts.rancher.io/

# Pull the Longhorn 1.8.1 CRD archive
helm pull rancher-charts/longhorn-crd --version 106.2.0+up1.8.1

# Pull the Longhorn 1.8.1 chart archive
helm pull rancher-charts/longhorn --version 106.2.0+up1.8.1
```

4. 在 `archives` 目录之外，从 `suse-edge/fleet-examples` 版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0) 处下载 `generate-chart-upgrade-data.sh` 脚本。

5. 目录设置如下所示：

```
.
├── archives
|   ├── longhorn-106.2.0+up1.8.1.tgz
|   └── longhorn-crd-106.2.0+up1.8.1.tgz
├── fleet-examples
...
|   ├── fleets
|   |   ├── day2
|   |   |   ├── ...
|   |   |   ├── eib-charts-upgrader
|   |   |   |   ├── base
|   |   |   |   |   ├── job.yaml
```

```
|   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   ├── patches
|   |   |   |   |   |   └── job-patch.yaml
|   |   |   |   |   ├── rbac
|   |   |   |   |   |   ├── cluster-role-binding.yaml
|   |   |   |   |   |   ├── cluster-role.yaml
|   |   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   |   └── sa.yaml
|   |   |   |   |   └── secrets
|   |   |   |   |       ├── eib-charts-upgrader-script.yaml
|   |   |   |   |       └── kustomization.yaml
|   |   |   |   ├── fleet.yaml
|   |   |   |   └── kustomization.yaml
|   |   |   └── ...
|   └── ...
└── generate-chart-upgrade-data.sh
```

6. 执行 `generate-chart-upgrade-data.sh` 脚本：

```
# First make the script executable
chmod +x ./generate-chart-upgrade-data.sh

# Then execute the script
./generate-chart-upgrade-data.sh --archive-dir ./archives --fleet-path ./
fleet-examples/fleets/day2/eib-charts-upgrader
```

脚本执行后的目录结构如下所示：

```
.
├── archives
|   ├── longhorn-106.2.0+up1.8.1.tgz
|   └── longhorn-crd-106.2.0+up1.8.1.tgz
├── fleet-examples
...
|   ├── fleets
|   |   ├── day2
|   |   |   ├── ...
|   |   |   ├── eib-charts-upgrader
|   |   |   |   ├── base
```

```
|   |   |   |   |   ├── job.yaml
|   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   ├── patches
|   |   |   |   |   |   └── job-patch.yaml
|   |   |   |   |   ├── rbac
|   |   |   |   |   |   ├── cluster-role-binding.yaml
|   |   |   |   |   |   ├── cluster-role.yaml
|   |   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   |   └── sa.yaml
|   |   |   |   |   └── secrets
|   |   |   |   |       ├── eib-charts-upgrader-script.yaml
|   |   |   |   |       ├── kustomization.yaml
|   |   |   |   |       ├── longhorn-VERSION.yaml - secret created by the
 generate-chart-upgrade-data.sh script
|   |   |   |   |       └── longhorn-crd-VERSION.yaml - secret created by
 the generate-chart-upgrade-data.sh script
|   |   |   |   ├── fleet.yaml
|   |   |   |   └── kustomization.yaml
|   |   |   └── ...
|   └── ...
└── generate-chart-upgrade-data.sh
```

git 中更改的文件如下所示：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 modified:   fleets/day2/eib-charts-upgrader/base/patches/job-patch.yaml
 modified:   fleets/day2/eib-charts-upgrader/base/secrets/
kustomization.yaml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
 fleets/day2/eib-charts-upgrader/base/secrets/longhorn-VERSION.yaml
 fleets/day2/eib-charts-upgrader/base/secrets/longhorn-crd-VERSION.yaml
```

7. 为 `eib-charts-upgrader` Fleet 创建捆绑包：

   a. 首先，导航到 Fleet：

```
cd ./fleet-examples/fleets/day2/eib-charts-upgrader
```

b. 然后创建 `targets.yaml` 文件：

```
cat > targets.yaml <<EOF
targets:
- clusterName: local
EOF
```

c. 接下来，使用 `fleet-cli` 二进制文件将 Fleet 转换为捆绑包：

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o -
 eib-charts-upgrade > bundle.yaml
```

8. 通过 Rancher UI 部署捆绑包：

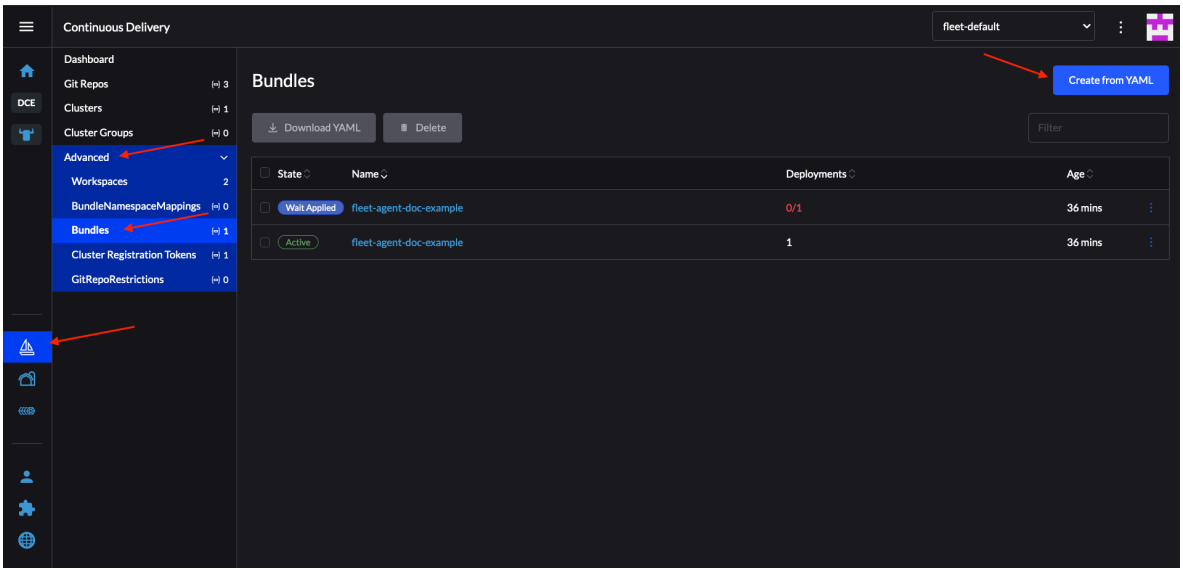


图 35.1：通过 RANCHER UI 部署捆绑包

在此处选择 **Read from File**（从文件读取），并找到系统上的 `bundle.yaml` 文件。
此时会在 Rancher UI 中自动填充捆绑包。
选择 **Create**（创建）。

9. 成功部署后，捆绑包如下所示：

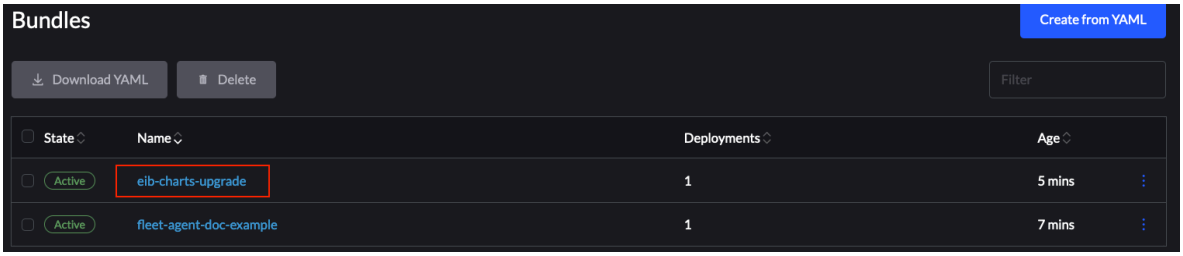
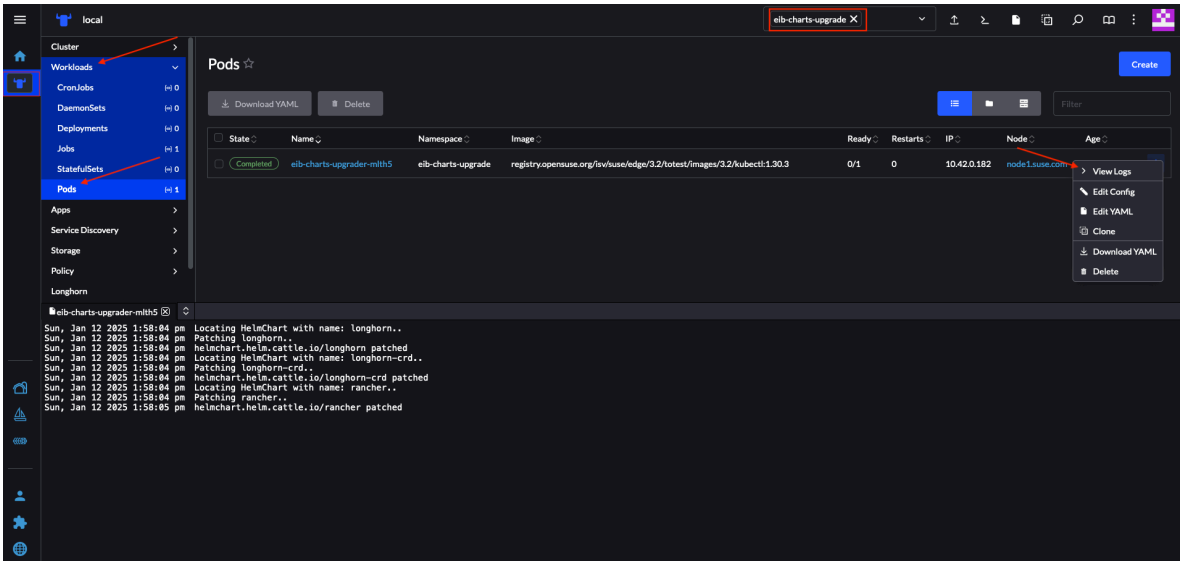


图 35.2： 已成功部署的捆绑包

成功部署捆绑包后，要监控升级过程，请执行以下操作：

1. 验证升级 Pod 的日志：



2. 现在验证 helm-controller 针对升级过程创建的 Pod 日志：

   a. Pod 名称将使用以下模板 - `helm-install-longhorn-<random-suffix>`

   b. Pod 将位于部署了 `HelmChart` 资源的名称空间中。在本例中为 `kube-system` 名称空间。

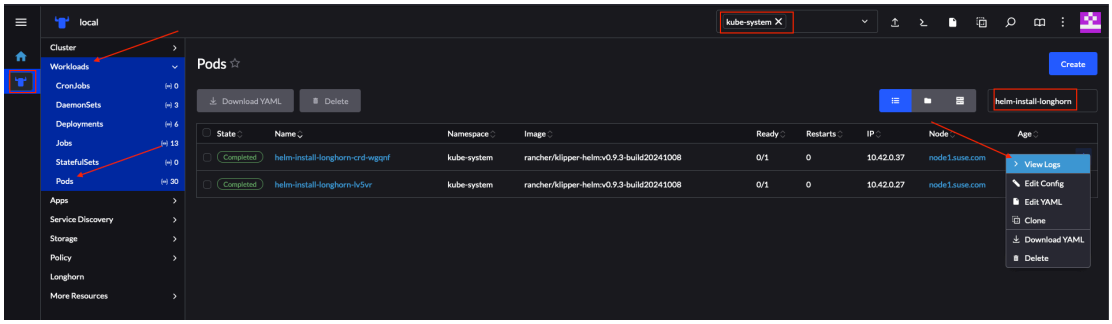


图 35.3：成功升级的 LONGHORN CHART 的日志

3. 导航到 Rancher 的 HelmChart 部分（More Resources（更多资源）→ HelmChart），验证 HelmChart 版本是否已更新。选择部署了该 chart 的名称空间，在本例中为 kube-system。

4. 最后检查 Longhorn Pod 是否正在运行。

在完成上述验证后，可以确定 Longhorn Helm chart 已升级到 106.2.0+up1.8.1 版本。

35.2.6.2.3.4 使用第三方 GitOps 工具进行 Helm chart 升级

在某些使用场景中，用户可能希望将此升级过程与 Fleet 以外的 GitOps 工作流程（例如 Flux）配合使用。

要生成升级过程所需的资源，可以使用 generate-chart-upgrade-data.sh 脚本将用户提供的数据填充到 eib-charts-upgrader Fleet。有关如何执行此操作的详细信息，请参见第 35.2.6.2.3.2 节 “升级步骤”。

完成所有设置后，可以使用 kustomize (https://kustomize.io) 生成一个可在群集中部署且完整有效的解决方案：

```
cd /foo/bar/fleets/day2/eib-charts-upgrader

kustomize build .
```

如果想将解决方案包含在 GitOps 工作流程中，可以去除 fleet.yaml 文件，并将其余内容用作有效的 Kustomize 设置。只是别忘了先运行 generate-chart-upgrade-data.sh 脚本，以便其可以将您想要升级到的 Helm chart 的数据填充到 Kustomize 设置中。

要了解如何使用此工作流程，可以查看第 35.2.6.2.3.1 节 “概述”和第 35.2.6.2.3.2 节 “升级步骤”。

36 下游群集

重要

以下步骤不适用于由 SUSE Edge for Telco（第 37 章 “SUSE Edge for Telco”）管理的下游群集。有关升级这些群集的说明，请参见第 43.2 节 “下游群集升级”。

本章介绍对下游群集的不同部分执行“Day 2”操作的可能方式。

36.1 Fleet

本节提供有关如何使用 Fleet（第 8 章 “Fleet”）组件执行“Day 2”操作的信息。

本节将介绍以下主题：

1. 第 36.1.1 节 “组件” - 执行所有“Day 2”操作时需要使用的默认组件。
2. 第 36.1.2 节 “确定您的使用场景” - 概述将使用的 Fleet 自定义资源及其在不同“Day 2”操作场景中的适用性。
3. 第 36.1.3 节 “Day 2 工作流程” - 提供使用 Fleet 执行“Day 2”操作的工作流程指南。
4. 第 36.1.4 节 “操作系统升级” - 说明如何使用 Fleet 进行操作系统升级。
5. 第 36.1.5 节 “Kubernetes 版本升级” - 说明如何使用 Fleet 进行 Kubernetes 版本升级。
6. 第 36.1.6 节 “Helm chart 升级” - 说明如何使用 Fleet 进行 Helm chart 升级。

36.1.1 组件

下文将介绍为使用 Fleet 顺利执行“Day 2”操作而应在下游群集上设置的默认组件。

36.1.1.1 系统升级控制器 (SUC)

注意

必须在每个下游群集上部署。

系统升级控制器负责根据通过名为计划的自定义资源提供的配置数据，在指定节点上执行任务。

系统会主动利用 **SUC** 升级操作系统和 Kubernetes 发行版。

有关 **SUC** 组件及其如何安置到 Edge 堆栈中的详细信息，请参见第 22 章 “系统升级控制器”。

有关如何部署 **SUC** 的信息，请先确定您的使用场景（第 36.1.2 节 “确定您的使用场景”），然后参见第 22.2.1.1 节 “系统升级控制器安装 - GitRepo”或第 22.2.1.2 节 “系统升级控制器安装 - 捆绑包”。

36.1.2 确定您的使用场景

Fleet 使用两种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) 来实现对 Kubernetes 和 Helm 资源的管理。

下文将介绍这些资源的用途，以及在 “Day 2” 操作情境中它们最适合的使用场景。

36.1.2.1 GitRepo

`GitRepo` 是一种 Fleet（第 8 章 “Fleet”）资源，它代表一个 Git 储存库，`Fleet` 可从中创建捆绑包。每个捆绑包都是基于 `GitRepo` 资源中定义的配置路径创建的。有关详细信息，请参见 GitRepo (https://fleet.rancher.io/gitrepo-add) 文档。

在 “Day 2” 操作情境中，`GitRepo` 资源通常用于在利用 **Fleet GitOps** 方法的**非隔离**环境中部署 SUC 或 SUC 计划。

或者，**如果您通过本地 git 服务器镜像储存库设置**，则 `GitRepo` 资源也可用于在**隔离**环境中部署 SUC 或 SUC 计划。

36.1.2.2 捆绑包

捆绑包包含了要在目标群集上部署的**原始** Kubernetes 资源。通常它们是基于 `GitRepo` 资源创建的，但在某些使用场景中也可以手动部署。有关详细信息，请参见捆绑包 (https://fleet.rancher.io/bundle-add) 文档。

在“Day 2”操作情境中，捆绑包资源通常用于在不使用某种形式的**本地 GitOps** 过程（例如**本地 git 服务器**）的**隔离**环境中部署 SUC 或 SUC 计划。

或者，如果您的应用场景不允许使用 **GitOps** 工作流程（例如需使用 Git 储存库），则捆绑包资源也可用于在**非隔离**环境中部署 SUC 或 SUC 计划。

36.1.3 Day 2 工作流程

下面是在将下游群集升级到特定 Edge 版本时应遵循的“Day 2”工作流程。

1. 操作系统升级（第 36.1.4 节 “操作系统升级”）
2. Kubernetes 版本升级（第 36.1.5 节 “Kubernetes 版本升级”）
3. Helm chart 升级（第 36.1.6 节 “Helm chart 升级”）

36.1.4 操作系统升级

本节介绍如何使用第 8 章 “Fleet”和第 22 章 “系统升级控制器”执行操作系统升级。

本节将介绍以下主题：

1. 第 36.1.4.1 节 “组件” - 升级过程使用的其他组件。
2. 第 36.1.4.2 节 “概述” - 升级过程概述。
3. 第 36.1.4.3 节 “要求” - 升级过程的要求。
4. 第 36.1.4.4 节 “操作系统升级 - SUC 计划部署” - 关于如何部署负责触发升级过程的 SUC 计划的信息。

36.1.4.1 组件

本节介绍操作系统升级过程中使用的自定义组件，这些组件与默认“Day 2”组件（第 36.1.1 节 “组件”）不同。

36.1.4.1.1 systemd.service

特定节点上的操作系统升级由 systemd.service (https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html) 处理。

根据 Edge 版本升级时操作系统所需的不同升级类型，会创建不同的服务：

- 对于需要相同操作系统版本（如 `6.0`）的 Edge 版本，将创建 `os-pkg-update.service`。它使用 transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) 执行常规软件包升级 (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages)。
- 对于需要迁移操作系统版本（例如 `6.0` → `6.1`）的 Edge 版本，将创建 `os-migration.service`。它使用 transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) 来执行：
  a. 常规软件包升级 (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages)，这可确保所有软件包均为最新版本，以减少因软件包版本过旧而导致的迁移失败情况。
  b. 利用 `zypper migration` 命令进行操作系统迁移。

上述服务通过 SUC 计划部署到每个节点，该计划必须位于需要升级操作系统的下游群集上。

36.1.4.2 概述

通过 `Fleet` 和系统升级控制器 (SUC) 来为下游群集节点升级操作系统。

Fleet 用于将 SUC 计划部署到目标群集并对其进行管理。

注意

SUC 计划是一种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)，描述了 SUC 为在一组节点上执行特定任务而需要遵循的步骤。有关 SUC 计划的示例，请参见上游储存库 (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans)。

通过向特定 Fleet 工作区 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 部署 GitRepo (https://fleet.rancher.io/gitrepo-add) 或捆绑包 (https://fleet.rancher.io/bundle-add) 资源将操作系统 SUC 计划分发到各个群集。Fleet 会获取已部署的 GitRepo/捆绑包，并将其内容（操作系统 SUC 计划）部署到目标群集。

注意

GitRepo/捆绑包资源始终部署在管理群集上。使用 GitRepo 还是捆绑包资源取决于具体应用场景，有关详细信息，请参见第 36.1.2 节 “确定您的使用场景”。

操作系统 SUC 计划描述了以下工作流程：

1. 执行操作系统升级前，务必要封锁 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) 节点。
2. 务必先升级控制平面节点，再升级工作节点。
3. 升级群集时，务必**逐个**节点依序升级。

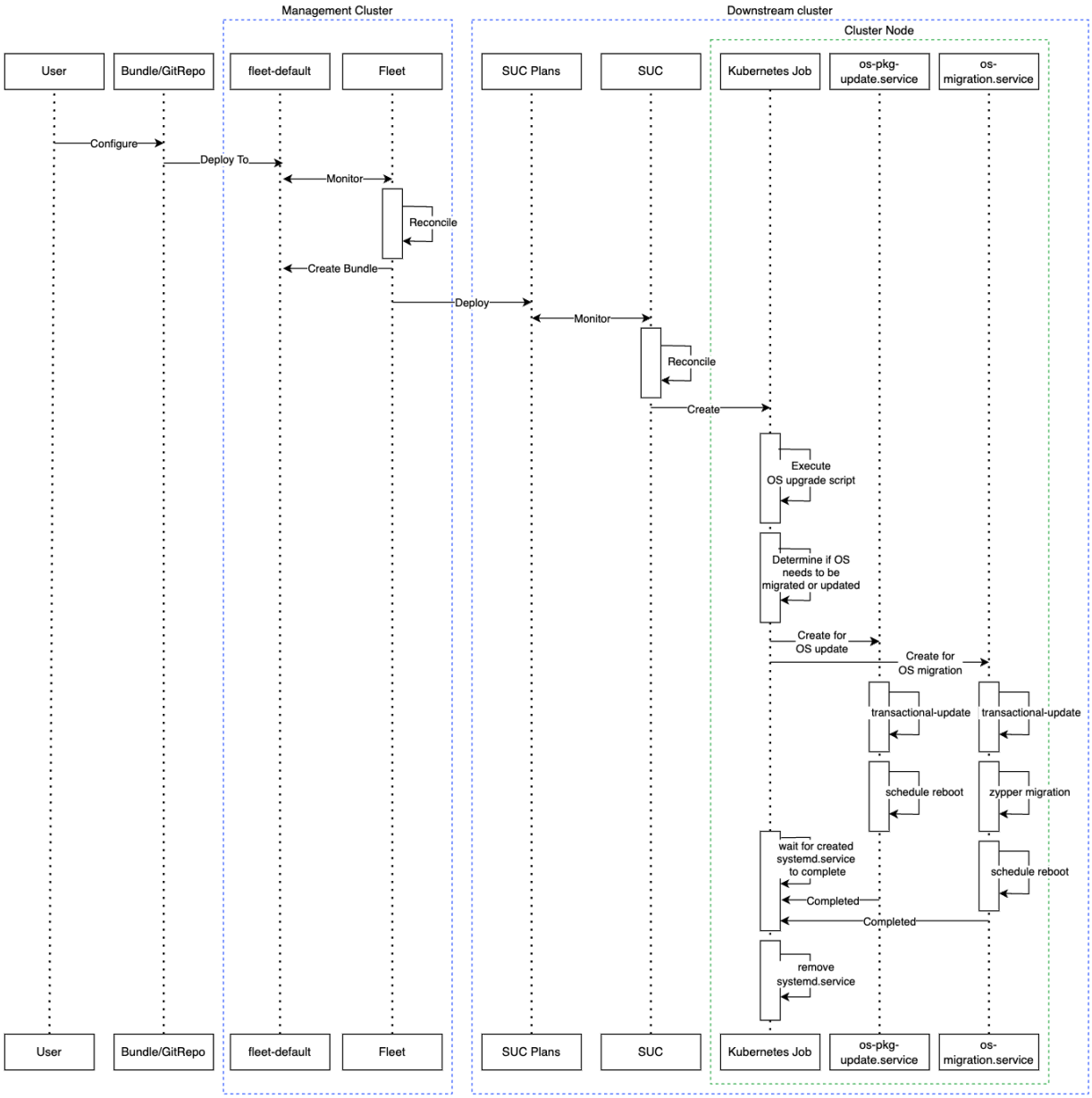
部署操作系统 SUC 计划后，工作流程如下：

1. SUC 协调已部署的操作系统 SUC 计划，并在**每个节点**上创建一个 Kubernetes 作业。
2. Kubernetes 作业创建一个 systemd.service（第 36.1.4.1.1 节 “systemd.service”），用于执行软件包升级或操作系统迁移。
3. 所创建的 systemd.service 触发特定节点上的操作系统升级过程。

重要

操作系统升级过程完成后，相应节点将重引导以应用系统更新。

下面是上述流程的示意图：



36.1.4.3 要求

一般：

1. **已在 SCC 中注册的计算机** - 所有下游群集节点都应已注册到 `https://scc.suse.com/`。只有这样，`systemd.service` 才能成功连接到所需的 RPM 储存库。

   重要

   对于需要进行操作系统版本迁移的 Edge 版本（例如 `6.0` → `6.1`），请确保您的 SCC 密钥支持迁移到新版本。

2. **确保 SUC 计划容忍度与节点容忍度相匹配** - 如果您的 Kubernetes 群集节点具有自定义**污点**，请确保在 **SUC 计划**中为这些污点添加容忍度 (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)。默认情况下，**SUC 计划**仅包含**控制平面**节点的容忍度。默认容忍度包括：

   - **CriticalAddonsOnly=true:NoExecute**
   - **node-role.kubernetes.io/control-plane:NoSchedule**
   - **node-role.kubernetes.io/etcd:NoExecute**

   注意

   其他任何容忍度必须添加到每个计划的 `.spec.tolerations` 部分。与操作系统升级相关的 **SUC 计划**可以在 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库中的 `fleets/day2/system-upgrade-controller-plans/os-upgrade` 下找到。**请确保使用有效储存库版本 (https://github.com/suse-edge/fleet-examples/releases) 标记中的计划。**

   为**控制平面** SUC 计划定义自定义容忍度的示例如下：

   ```
   apiVersion: upgrade.cattle.io/v1
   kind: Plan
   metadata:
   ```

```
  name: os-upgrade-control-plane
spec:
  ...
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

隔离：

1. **镜像 SUSE RPM 储存库** - 操作系统 RPM 储存库应镜像到本地，以便 `systemd.service` 可以访问它们。您可以使用 RMT (https://documentation.suse.com/sles/15-SP6/html/SLES-all/book-rmt.html) 或 SUMA (https://documentation.suse.com/suma/5.0/en/suse-manager/index.html) 来完成该操作。

36.1.4.4 操作系统升级 - SUC 计划部署

重要

对于之前使用此过程升级的环境，用户应确保完成以下步骤之一：

- 从下游群集中去除任何先前部署且与旧版 Edge 相关的 SUC 计划 - 方法是从现有的 GitRepo/捆绑包目标配置 (https://fleet.rancher.io/gitrepo-targets#target-matching) 中去除相应群集，或完全去除 GitRepo/捆绑包资源。
- 重用现有的 GitRepo/捆绑包资源 - 方法是将资源的修订版指向一个新标签，该标签包含目标 suse-edge/fleet-examples 版本 (https://github.com/suse-edge/fleet-examples/releases) 的正确 Fleet。

这样做是为了避免旧版 Edge 的 SUC 计划之间发生冲突。

如果用户尝试升级，而下游群集上存在现有的 SUC 计划，他们将看到以下 Fleet 错误：

```
Not installed: Unable to continue with install: Plan <plan_name> in
 namespace <plan_namespace> exists and cannot be imported into the current
 release: invalid ownership metadata; annotation validation error..
```

如第 36.1.4.2 节 “概述”中所述，可以通过以下任意一种方式将 SUC 计划分发到目标群集来完成操作系统升级：

- Fleet GitRepo 资源 - 第 36.1.4.4.1 节 “SUC 计划部署 - GitRepo 资源”。
- Fleet 捆绑包资源 - 第 36.1.4.4.2 节 “SUC 计划部署 - 捆绑包资源”。

要确定使用哪个资源，请参见第 36.1.2 节 “确定您的使用场景”。

对于希望通过第三方 GitOps 工具部署操作系统 SUC 计划的使用场景，请参见第 36.1.4.4.3 节 “SUC 计划部署 - 第三方 GitOps 工作流程”。

36.1.4.4.1 SUC 计划部署 - GitRepo 资源

提供所需操作系统 SUC 计划的 **GitRepo** 资源可通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 36.1.4.4.1.1 节 “GitRepo 创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 36.1.4.4.1.2 节 “GitRepo 创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的操作系统升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

36.1.4.4.1.1 GitRepo 创建 - Rancher UI

要通过 Rancher UI 创建 `GitRepo` 资源，请遵循其官方文档 (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

Edge 团队维护着一个即用型 Fleet (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/os-upgrade)。根据您的环境的不同，该 Fleet 可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases)标记使用此 Fleet。

对于不需要在 Fleet 附带的 SUC 计划中包含自定义更改的使用场景，用户可以直接引用 `suse-edge/fleet-examples` 储存库中的 `os-upgrade` Fleet。

如果需要自定义更改（例如添加自定义容忍度），用户应从单独的储存库中引用 `os-upgrade` Fleet，以便能够根据需要将更改添加到 SUC 计划中。

有关如何配置 `GitRepo` 以使用 `suse-edge/fleet-examples` 储存库中的 Fleet 的示例，请参见此处 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/os-upgrade-gitrepo.yaml)。

36.1.4.4.1.2 GitRepo 创建 - 手动

1. 提取 **GitRepo** 资源：

```
curl -o os-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/os-upgrade-gitrepo.yaml
```

2. 编辑 **GitRepo** 配置，在 `spec.targets` 下指定所需的目标列表。默认情况下，`suse-edge/fleet-examples` 中的 `GitRepo` 资源**不会**映射到任何下游群集。

   - 为了匹配所有群集，请将默认的 `GitRepo` **目标**更改为：

     ```
     spec:
       targets:
       - clusterSelector: {}
     ```

   - 或者，如果您要更细致地选择群集，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)

3. 将 **GitRepo** 资源应用于管理群集：

   ```
   kubectl apply -f os-upgrade-gitrepo.yaml
   ```

4. 查看 `fleet-default` 名称空间下创建的 **GitRepo** 资源：

   ```
   kubectl get gitrepo os-upgrade -n fleet-default

   # Example output
   NAME            REPO                                                COMMIT
         BUNDLEDEPLOYMENTS-READY   STATUS
   os-upgrade      https://github.com/suse-edge/fleet-examples.git
    release-3.3.0  0/0
   ```

36.1.4.4.2 SUC 计划部署 - 捆绑包资源

提供所需操作系统 SUC 计划的**捆绑包**资源可以通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 36.1.4.4.2.1 节 “捆绑包创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 36.1.4.4.2.2 节 “捆绑包创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的操作系统升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

36.1.4.4.2.1 捆绑包创建 - Rancher UI

Edge 团队维护着一个可在以下步骤中使用的即用型捆绑包 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml)。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记使用此捆绑包。

要通过 Rancher 的 UI 创建捆绑包，请执行以下操作：

1. 单击左上角的 **#** → **Continuous Delivery（持续交付）**
2. 转到 **Advanced**（高级）> **Bundles**（捆绑包）
3. 选择 **Create from YAML**（基于 YAML 创建）
4. 此处可通过以下方式之一创建捆绑包：

注意

在某些使用场景中，您可能需要在捆绑包附带的 SUC 计划中包含自定义更改。请确保在以下步骤生成的捆绑包中包含这些更改。

a. 手动将 `suse-edge/fleet-examples` 中的捆绑包内容 (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml) 复制到 **Create from YAML**（基于 YAML 创建）页面。

b. 从所需的版本 (https://github.com/suse-edge/fleet-examples/releases) 标记克隆 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库，并在 **Create from YAML**（基于 YAML 创建）页面中选择 **Read from File**（从文件读取）选项。然后导航到捆绑包位置 (`bundles/day2/system-upgrade-controller-plans/os-upgrade`) 并选择捆绑包文件。这会在**Create from YAML**（基于 YAML 创建）页面中自动填充捆绑包内容。

5. 更改捆绑包的**目标**群集：

   - 为了匹配所有下游群集，请将默认的捆绑包 `.spec.targets` 更改为：

     ```
     spec:
       targets:
       - clusterSelector: {}
     ```

   - 有关更精细的下游群集映射，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)。

6. 选择 **Create**（创建）

36.1.4.4.2.2 捆绑包创建 - 手动

1. 提取**捆绑包**资源：

   ```
   curl -o os-upgrade-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml
   ```

2. 编辑捆绑包**目标**配置，在 `spec.targets` 下提供所需的目标列表。默认情况下，`suse-edge/fleet-examples` 中的捆绑包资源**不会**映射到任何下游群集。

- 为了匹配所有群集，请将默认的捆绑包**目标**更改为：

```
spec:
  targets:
  - clusterSelector: {}
```

- 或者，如果您要更细致地选择群集，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)

3. 将**捆绑包**资源应用于管理群集：

```
kubectl apply -f os-upgrade-bundle.yaml
```

4. 查看 `fleet-default` 名称空间下创建的**捆绑包**资源：

```
kubectl get bundles -n fleet-default
```

36.1.4.4.3 SUC 计划部署 - 第三方 GitOps 工作流程

在某些使用场景中，用户可能希望将操作系统 SUC 计划合并到自己的第三方 GitOps 工作流程（例如 Flux）中。

要获取所需的操作系统升级资源，首先请确定您要使用的 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记。

然后，便可以在 `fleets/day2/system-upgrade-controller-plans/os-upgrade` 中找到资源，其中：

- `plan-control-plane.yaml` 是用于**控制平面**节点的 SUC 计划资源。
- `plan-worker.yaml` 是用于**工作**节点的 SUC 计划资源。
- `secret.yaml` 是一个包含 `upgrade.sh` 脚本的机密，该脚本负责创建 systemd.service（第 36.1.4.1.1 节 “systemd.service”）。
- `config-map.yaml` 是 ConfigMap，提供 `upgrade.sh` 脚本使用的升级配置。

重要

这些计划资源由系统升级控制器解释，应部署在您要升级的每个下游群集上。有关 SUC 部署信息，请参见第 22.2 节 “安装系统升级控制器”。

为了更好地了解如何使用 GitOps 工作流程来部署操作系统升级的 **SUC 计划**，建议查看第 36.1.4.2 节 “概述”。

36.1.5 Kubernetes 版本升级

重要

本节介绍**并非**通过 Rancher（第 5 章 “Rancher”）实例创建的下游群集的 Kubernetes 升级过程。有关如何对通过 Rancher 创建的群集进行 Kubernetes 版本升级的信息，请参见 Upgrading and Rolling Back Kubernetes (https://ranchermanager.docs.rancher.com/v2.11/getting-started/installation-and-upgrade/upgrade-and-roll-back-kubernetes#upgrading-the-kubernetes-version)。

本节介绍如何使用第 8 章 “Fleet”和第 22 章 “系统升级控制器”执行 Kubernetes 升级。

本节将介绍以下主题：

1. 第 36.1.5.1 节 “组件” - 升级过程使用的其他组件。
2. 第 36.1.5.2 节 “概述” - 升级过程概述。
3. 第 36.1.5.3 节 “要求” - 升级过程的要求。
4. 第 36.1.5.4 节 “K8s 升级 - SUC 计划部署” - 关于如何部署负责触发升级过程的 SUC 计划的信息。

36.1.5.1 组件

本节介绍 K8s 升级过程中使用的自定义组件，这些组件与默认 “Day 2” 组件（第 36.1.1 节 “组件”）不同。

36.1.5.1.1 rke2-upgrade

容器映像负责升级特定节点的 RKE2 版本。

此组件由 **SUC** 根据 **SUC 计划**创建的 Pod 分发。该计划应位于需要进行 RKE2 升级的每个**群集**上。

有关 `rke2-upgrade` 映像如何执行升级的详细信息，请参见上游 (https://github.com/rancher/rke2-upgrade/tree/master) 文档。

36.1.5.1.2 k3s-upgrade

容器映像负责升级特定节点的 K3s 版本。

此组件由 **SUC** 根据 **SUC 计划**创建的 Pod 分发。该计划应位于需要进行 K3s 升级的每个**群集**上。

有关 `k3s-upgrade` 映像如何执行升级的详细信息，请参见上游 (https://github.com/k3s-io/k3s-upgrade) 文档。

36.1.5.2 概述

通过 `Fleet` 和系统升级控制器（SUC）来为下游群集节点升级 Kubernetes 发行版。

`Fleet` 用于将 SUC 计划部署到目标群集并对其进行管理。

注意

SUC 计划是一种自定义资源 (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)，描述了 **SUC** 为在一组节点上执行特定任务而需要遵循的步骤。有关 SUC 计划的示例，请参见上游储存库 (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans)。

通过向特定 Fleet 工作区 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) 部署 GitRepo (https://fleet.rancher.io/gitrepo-add) 或捆绑包 (https://fleet.rancher.io/bundle-add) 资源将 K8s SUC 计划分发到各个群集。Fleet 会获取已部署的 GitRepo/捆绑包，并将其内容（K8s SUC 计划）部署到目标群集。

注意

GitRepo/捆绑包资源始终部署在管理群集上。使用 GitRepo 还是捆绑包资源取决于具体应用场景，有关详细信息，请参见第 36.1.2 节 “确定您的使用场景”。

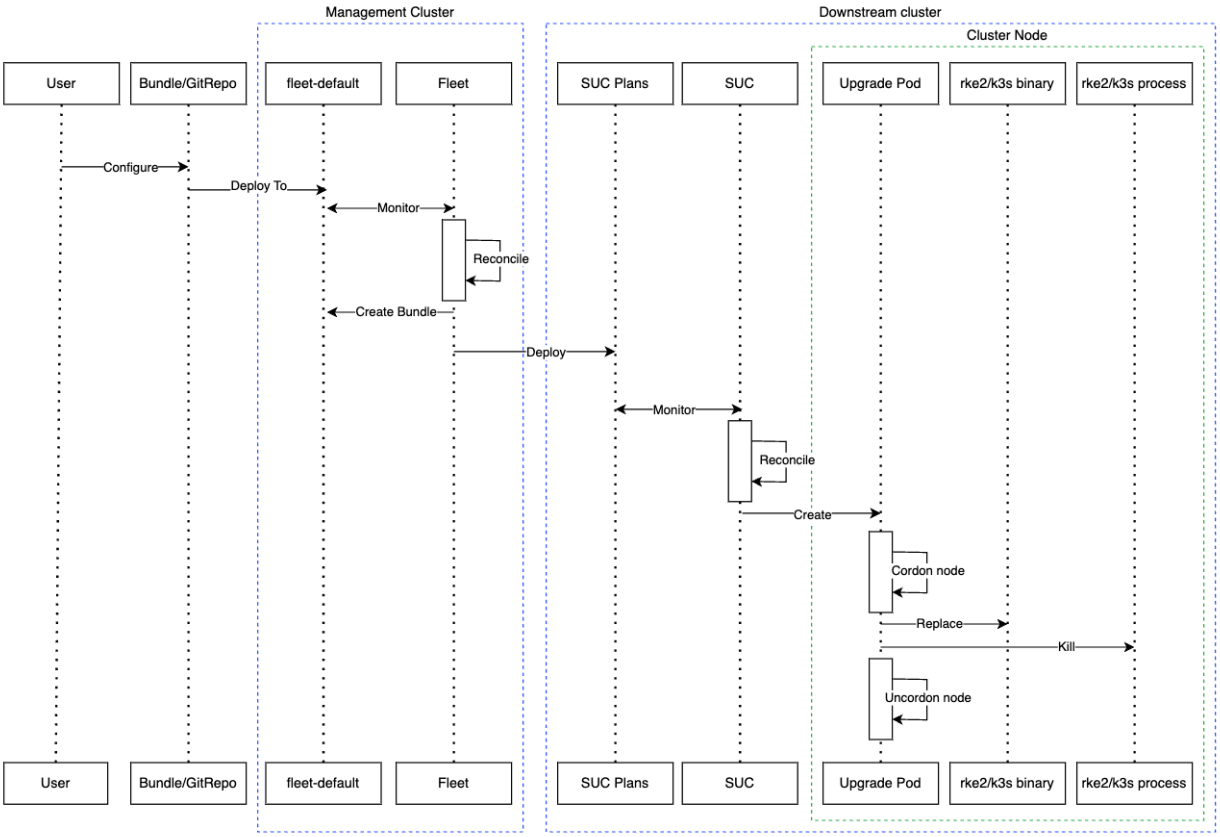
K8s SUC 计划描述了以下工作流程：

1. 执行 K8s 升级前，务必要封锁 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) 节点。
2. 务必先升级控制平面节点，再升级工作节点。
3. 始终一次升级**一个**控制平面节点，一次升级**两个**工作节点。

部署 K8s SUC 计划后，工作流程如下：

1. SUC 协调已部署的 K8s SUC 计划，并在**每个节点**上创建一个 Kubernetes 作业。
2. 根据 Kubernetes 发行版的不同，该作业将创建一个运行 rke2-upgrade（第 36.1.5.1.1 节 “rke2-upgrade”）或 k3s-upgrade（第 36.1.5.1.2 节 “k3s-upgrade”）容器映像的 Pod。
3. 创建的 Pod 将执行以下工作流程：
    a. 用 rke2-upgrade/k3s-upgrade 映像中的二进制文件替换节点上现有的 rke2/k3s 二进制文件。
    b. 终止正在运行的 rke2/k3s 进程。
4. 终止 rke2/k3s 进程会触发重启，启动运行已更新二进制文件的新进程，从而实现 Kubernetes 发行版版本的升级。

下面是上述流程的示意图：



36.1.5.3 要求

1. **备份您的 Kubernetes 发行版：**

   a. 对于 **RKE2 群集**，请参见 RKE2 Backup and Restore (https://docs.rke2.io/datastore/backup_restore) 文档。

   b. 对于 **K3s 群集**，请参见 K3s Backup and Restore (https://docs.k3s.io/datastore/backup-restore) 文档。

2. **确保 SUC 计划容忍度与节点容忍度相匹配** - 如果您的 Kubernetes 群集节点具有自定义**污点**，请确保在 **SUC 计划**中为这些污点添加容忍度 (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/)。默认情况下，**SUC 计划**仅包含**控制平面**节点的容忍度。默认容忍度包括：

- **CriticalAddonsOnly=true:NoExecute**
- **node-role.kubernetes.io/control-plane:NoSchedule**
- **node-role.kubernetes.io/etcd:NoExecute**

注意

其他任何容忍度必须添加到每个计划的 `.spec.tolerations` 部分下。与 Kubernetes 版本升级相关的 **SUC 计划**可以在 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 储存库中的以下位置找到：

- 对于 **RKE2** - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade`
- 对于 **K3s** - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade`

请确保使用有效储存库版本 (https://github.com/suse-edge/fleet-examples/releases) 标记中的计划。

为 RKE2 **控制平面** SUC 计划定义自定义容忍度的示例如下：

```
apiVersion: upgrade.cattle.io/v1
kind: Plan
metadata:
  name: rke2-upgrade-control-plane
spec:
  ...
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
```

```
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

36.1.5.4 K8s 升级 - SUC 计划部署

重要

对于之前使用此过程升级的环境，用户应确保完成以下步骤之一：

- 从下游群集中去除任何先前部署且与旧版 Edge 相关的 SUC 计划 - 方法是从现有的 GitRepo/捆绑包目标配置 (https://fleet.rancher.io/gitrepo-targets#target-matching) 中去除相应群集，或完全去除 GitRepo/捆绑包资源。
- 重用现有的 GitRepo/捆绑包资源 - 方法是将资源的修订版指向一个新标签，该标签包含目标 suse-edge/fleet-examples 版本 (https://github.com/suse-edge/fleet-examples/releases) 的正确 Fleet。

这样做是为了避免旧版 Edge 的 SUC 计划之间发生冲突。

如果用户尝试升级，而下游群集上存在现有的 SUC 计划，他们将看到以下 Fleet 错误：

```
Not installed: Unable to continue with install: Plan <plan_name> in
 namespace <plan_namespace> exists and cannot be imported into the current
 release: invalid ownership metadata; annotation validation error..
```

如第 36.1.5.2 节 “概述”中所述，可以通过以下任意一种方式将 SUC 计划分发到目标群集来完成 Kubernetes 升级：

- Fleet GitRepo 资源（第 36.1.5.4.1 节 “SUC 计划部署 - GitRepo 资源”）
- Fleet 捆绑包资源（第 36.1.5.4.2 节 “SUC 计划部署 - 捆绑包资源”）

要确定使用哪个资源，请参见第 36.1.2 节 “确定您的使用场景”。

对于希望通过第三方 GitOps 工具部署 K8s SUC 计划的使用场景，请参见第 36.1.5.4.3 节 “SUC 计划部署 - 第三方 GitOps 工作流程”。

36.1.5.4.1 SUC 计划部署 - GitRepo 资源

提供所需 K8s SUC 计划的 **GitRepo** 资源可通过以下方式之一进行部署：

1. 通过 Rancher UI 部署 - 第 36.1.5.4.1.1 节 “GitRepo 创建 - Rancher UI”（如果 Rancher 可用）。
2. 手动将相应资源部署（第 36.1.5.4.1.2 节 “GitRepo 创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的 Kubernetes 升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

36.1.5.4.1.1 GitRepo 创建 - Rancher UI

要通过 Rancher UI 创建 GitRepo 资源，请遵循其官方文档 (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

Edge 团队为 rke2 (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/rke2-upgrade) 和 k3s (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/system-upgrade-controller-plans/k3s-upgrade) 这两种 Kubernetes 发行版维护着即用型 Fleet。根据您的环境的不同，这些 Fleet 可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记使用这些 Fleet。

对于不需要在这些 Fleet 附带的 SUC 计划中包含自定义更改的使用场景，用户可以直接引用 suse-edge/fleet-examples 储存库中的 Fleet。

如果需要自定义更改（例如添加自定义容忍度），用户应从单独的储存库中引用 Fleet，以便能够根据需要将更改添加到 SUC 计划中。

使用 `suse-edge/fleet-examples` 储存库中的 Fleet 配置 `GitRepo` 资源的示例：

- RKE2 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml)
- K3s (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/gitrepos/day2/k3s-upgrade-gitrepo.yaml)

36.1.5.4.1.2 GitRepo 创建 - 手动

1. 提取 **GitRepo** 资源：
    - 对于 **RKE2** 群集：

      ```
      curl -o rke2-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml
      ```

    - 对于 **K3s** 群集：

      ```
      curl -o k3s-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/gitrepos/day2/k3s-upgrade-gitrepo.yaml
      ```

2. 编辑 **GitRepo** 配置，在 `spec.targets` 下指定所需的目标列表。默认情况下，`suse-edge/fleet-examples` 中的 `GitRepo` 资源**不会**映射到任何下游群集。
    - 为了匹配所有群集，请将默认的 `GitRepo` **目标**更改为：

      ```
      spec:
        targets:
        - clusterSelector: {}
      ```

    - 或者，如果您要更细致地选择群集，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)

3. 将 **GitRepo** 资源应用于管理群集：

```
# RKE2
kubectl apply -f rke2-upgrade-gitrepo.yaml

# K3s
kubectl apply -f k3s-upgrade-gitrepo.yaml
```

4. 查看 `fleet-default` 名称空间下创建的 **GitRepo** 资源：

```
# RKE2
kubectl get gitrepo rke2-upgrade -n fleet-default

# K3s
kubectl get gitrepo k3s-upgrade -n fleet-default

# Example output
NAME           REPO                                                COMMIT
      BUNDLEDEPLOYMENTS-READY   STATUS
k3s-upgrade    https://github.com/suse-edge/fleet-examples.git   fleet-
default   0/0
rke2-upgrade   https://github.com/suse-edge/fleet-examples.git   fleet-
default   0/0
```

36.1.5.4.2 SUC 计划部署 - 捆绑包资源

可通过以下方式之一部署**捆绑包**资源，该资源中附带了所需的 `Kubernetes 升级 SUC 计划`：

1. 通过 `Rancher UI` 部署 - 第 36.1.5.4.2.1 节 “捆绑包创建 - Rancher UI”（如果 `Rancher` 可用）。
2. 手动将相应资源部署（第 36.1.5.4.2.2 节 “捆绑包创建 - 手动”）到管理群集。

部署后，要监控目标群集节点的 Kubernetes 升级过程，请参见第 22.3 节 “监控系统升级控制器计划”。

36.1.5.4.2.1 捆绑包创建 - Rancher UI

Edge 团队为 rke2 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) 和 k3s (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) 这两种 Kubernetes 发行版维护着即用型捆绑包。根据您的环境的不同，这些捆绑包可以直接使用或用作模板。

重要

请务必通过有效的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记使用此捆绑包。

要通过 Rancher 的 UI 创建捆绑包，请执行以下操作：

1. 单击左上角的 **#** → **Continuous Delivery（持续交付）**
2. 转到 **Advanced**（高级）> **Bundles**（捆绑包）
3. 选择 **Create from YAML**（基于 YAML 创建）
4. 此处可通过以下方式之一创建捆绑包：

注意

在某些使用场景中，您可能需要在捆绑包附带的 SUC 计划中包含自定义更改。请确保在以下步骤生成的捆绑包中包含这些更改。

a. 手动将 RKE2 (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) 或 K3s (https://raw.githubusercontent.com/suse-

edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) 的捆绑包内容从 `suse-edge/fleet-examples` 复制到 **Create from YAML**（基于 YAML 创建）页面。

b. 从所需的版本 (https://github.com/suse-edge/fleet-examples/releases) 标记克隆 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) 储存库，并在 **Create from YAML**（基于 YAML 创建）页面中选择 **Read from File**（从文件读取）选项。然后导航到所需的捆绑包（对于 RKE2，为 `bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml`；对于 K3s，为 `bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml`）。这会在 **Create from YAML**（基于 YAML 创建）页面中自动填充捆绑包内容。

5. 更改捆绑包的**目标**群集：

   - 为了匹配所有下游群集，请将默认的捆绑包 `.spec.targets` 更改为：

   ```
   spec:
     targets:
     - clusterSelector: {}
   ```

   - 有关更精细的下游群集映射，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)。

6. 选择 **Create**（创建）

36.1.5.4.2.2 捆绑包创建 - 手动

1. 提取**捆绑包**资源：

   - 对于 **RKE2** 群集：

   ```
   curl -o rke2-plan-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml
   ```

   - 对于 **K3s** 群集：

```
curl -o k3s-plan-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.3.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml
```

2. 编辑捆绑包**目标**配置，在 `spec.targets` 下提供所需的目标列表。默认情况下，`suse-edge/fleet-examples` 中的捆绑包资源**不会**映射到任何下游群集。

   - 为了匹配所有群集，请将默认的捆绑包**目标**更改为：

     ```
     spec:
       targets:
       - clusterSelector: {}
     ```

   - 或者，如果您要更细致地选择群集，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)

3. 将**捆绑包**资源应用于管理群集：

   ```
   # For RKE2
   kubectl apply -f rke2-plan-bundle.yaml

   # For K3s
   kubectl apply -f k3s-plan-bundle.yaml
   ```

4. 查看 `fleet-default` 名称空间下创建的**捆绑包**资源：

   ```
   # For RKE2
   kubectl get bundles rke2-upgrade -n fleet-default

   # For K3s
   kubectl get bundles k3s-upgrade -n fleet-default

   # Example output
   NAME           BUNDLEDEPLOYMENTS-READY   STATUS
   k3s-upgrade    0/0
   rke2-upgrade   0/0
   ```

36.1.5.4.3 SUC 计划部署 - 第三方 GitOps 工作流程

在某些使用场景中，用户可能希望将 Kubernetes 升级 SUC 计划合并到自己的第三方 GitOps 工作流程（例如 Flux）中。

要获取所需的 K8s 升级资源，首先请确定您要使用的 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) 储存库的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases) 标记。

然后，便可以在以下位置找到资源：

- 对于 RKE2 群集升级：
  - 对于控制平面节点 - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-control-plane.yaml`
  - 对于工作节点 - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-agent.yaml`
- 对于 K3s 群集升级：
  - 对于控制平面节点 - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-control-plane.yaml`
  - 对于工作节点 - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-agent.yaml`

重要

这些计划资源由系统升级控制器解释，应部署在您要升级的每个下游群集上。有关 SUC 部署信息，请参见第 22.2 节 “安装系统升级控制器”。

为了更好地了解如何使用 GitOps 工作流程来部署 Kubernetes 版本升级的 **SUC 计划**，建议查看使用 Fleet 进行更新的过程概述（第 36.1.5.2 节 “概述”）。

36.1.6 Helm chart 升级

本节介绍如下内容：

1. 第 36.1.6.1 节 “为隔离环境做好准备” - 包含有关如何将与 Edge 相关的 OCI chart 和映像分发到您的专用注册表的信息。
2. 第 36.1.6.2 节 “升级过程” - 包含有关不同 Helm chart 升级情形及其升级过程的信息。

36.1.6.1 为隔离环境做好准备

36.1.6.1.1 确保您有权访问 Helm chart 的 Fleet

根据环境支持的情况，选择以下选项之一：

1. 将 chart 的 Fleet 资源托管在管理群集可访问的本地 git 服务器上。
2. 使用 Fleet 的 CLI 将 Helm chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle)，以便直接使用，而不必托管于某处。Fleet 的 CLI 可以从其版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2)页面中检索，对于 Mac 用户，有一个 fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae。

36.1.6.1.2 找到 Edge 发行版本的所需资产

1. 转到“Day 2”版本 (https://github.com/suse-edge/fleet-examples/releases)页面，找到您要将 chart 升级到的 Edge 版本，然后单击 **Assets**（资产）。
2. 从“**Assets**”（资产）部分下载以下文件：

版本文件	说明
edge-save-images.sh	提取 `edge-release-images.txt` 文件中指定的映像，并将其封装到“.tar.gz”归档中。
edge-save-oci-artefacts.sh	提取与特定 Edge 版本相关的 OCI chart 映像，并将其封装到“.tar.gz”归档中。

edge-load-images.sh	从“.tar.gz”归档加载映像，将它们重新标记并推送到专用注册表。
edge-load-oci-artefacts.sh	接收包含 Edge OCI '.tgz' chart 软件包的目录作为参数，并将这些软件包加载到专用注册表中。
edge-release-helm-oci-artefacts.txt	包含与特定 Edge 版本相关的 OCI chart 映像的列表。
edge-release-images.txt	包含与特定 Edge 版本相关的映像列表。

36.1.6.1.3 创建 Edge 版本映像归档

在可以访问互联网的计算机上：

1. 将 `edge-save-images.sh` 设为可执行文件：

```
chmod +x edge-save-images.sh
```

2. 生成映像归档：

```
./edge-save-images.sh --source-registry registry.suse.com
```

3. 这将创建一个名为 `edge-images.tar.gz` 的可加载归档。

注意

如果指定了 `-i|--images` 选项，归档的名称可能会不同。

4. 将此归档复制到**隔离的**计算机：

```
scp edge-images.tar.gz <user>@<machine_ip>:/path
```

36.1.6.1.4 创建 Edge OCI chart 映像归档

在可以访问互联网的计算机上：

1. 将 `edge-save-oci-artefacts.sh` 设为可执行文件：

```
chmod +x edge-save-oci-artefacts.sh
```

2. 生成 OCI chart 映像归档：

```
./edge-save-oci-artefacts.sh --source-registry registry.suse.com
```

3. 这将创建一个名为 `oci-artefacts.tar.gz` 的归档。

注意

如果指定了 `-a|--archive` 选项，归档的名称可能会不同。

4. 将此归档复制到**隔离的**计算机：

```
scp oci-artefacts.tar.gz <user>@<machine_ip>:/path
```

36.1.6.1.5 将 Edge 版本映像加载到隔离的计算机上

在隔离的计算机上：

1. 登录到专用注册表（如果需要）：

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. 将 `edge-load-images.sh` 设为可执行文件：

```
chmod +x edge-load-images.sh
```

3. 执行脚本，传递之前**复制的** `edge-images.tar.gz` 归档：

```
./edge-load-images.sh --source-registry registry.suse.com --registry
 <REGISTRY.YOURDOMAIN.COM:PORT> --images edge-images.tar.gz
```

注意

这将从 `edge-images.tar.gz` 加载所有映像，并将它们重新标记并推送到 `--registry` 选项下指定的注册表。

36.1.6.1.6 将 Edge OCI chart 映像加载到隔离的计算机上

在隔离的计算机上：

1. 登录到专用注册表（如果需要）：

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. 将 `edge-load-oci-artefacts.sh` 设为可执行文件：

```
chmod +x edge-load-oci-artefacts.sh
```

3. 解压缩复制的 `oci-artefacts.tar.gz` 归档：

```
tar -xvf oci-artefacts.tar.gz
```

4. 这会使用命名模板 `edge-release-oci-tgz-<date>` 生成一个目录

5. 将此目录传递给 `edge-load-oci-artefacts.sh` 脚本，以将 Edge OCI chart 映像加载到专用注册表中：

注意

此脚本假设您的环境中已预装了 `Helm` CLI。有关 Helm 安装说明，请参见安装 Helm (https://helm.sh/docs/intro/install/)。

```
./edge-load-oci-artefacts.sh --archive-directory edge-release-oci-tgz-
<date> --registry <REGISTRY.YOURDOMAIN.COM:PORT> --source-registry
 registry.suse.com
```

36.1.6.1.7 在 Kubernetes 发行版中配置专用注册表

对于 RKE2，请参见 Private Registry Configuration (https://docs.rke2.io/install/private_registry)

对于 K3s，请参见 Private Registry Configuration (https://docs.k3s.io/installation/private-registry)

36.1.6.2 升级过程

本节重点介绍以下使用场景的 Helm 升级过程：

1. 第 36.1.6.2.1 节 “我有一个新群集，想要部署和管理 Edge Helm chart”
2. 第 36.1.6.2.2 节 “我想升级 Fleet 管理的 Helm chart”
3. 第 36.1.6.2.3 节 “我要升级通过 EIB 部署的 Helm chart”

重要

手动部署的 Helm chart 无法可靠升级。我们建议使用第 36.1.6.2.1 节 “我有一个新群集，想要部署和管理 Edge Helm chart”中所述的方法重新部署这些 Helm chart。

36.1.6.2.1 我有一个新群集，想要部署和管理 Edge Helm chart

本节介绍如何：

1. 第 36.1.6.2.1.1 节 “为您的 chart 准备 Fleet 资源”。
2. 第 36.1.6.2.1.2 节 “部署您的 chart 的 Fleet”。
3. 第 36.1.6.2.1.3 节 “管理部署的 Helm chart”。

36.1.6.2.1.1 为您的 chart 准备 Fleet 资源

1. 从您要使用的 Edge 版本 (https://github.com/suse-edge/fleet-examples/releases)标记处获取 Chart 的 Fleet 资源。
2. 导航到 Helm chart Fleet (`fleets/day2/chart-templates/<chart>`)

3. **如果您打算使用 GitOps 工作流程**，请将 chart Fleet 目录复制到 Git 储存库，从那里执行 GitOps。

4. **（可选）**如果需要配置 Helm chart 的**值**才能使用 Helm chart，请编辑复制的目录中 `fleet.yaml` 文件内的 `.helm.values` 配置。

5. **（可选）**在某些使用场景中，您可能需要向 chart 的 Fleet 目录添加其他资源，使该目录能够更好地适应您的环境。有关如何增强 Fleet 目录的信息，请参见 Git 储存库内容 (https://fleet.rancher.io/gitrepo-content)。

注意

在某些情况下，Fleet 为 Helm 操作设置的默认超时时长可能不足，导致发生以下错误：

```
failed pre-install: context deadline exceeded
```

在此类情况下，请在 `fleet.yaml` 文件的 `helm` 配置下添加 timeoutSeconds (https://fleet.rancher.io/ref-crds#helmoptions) 属性。

`longhorn` Helm chart 的**示例**如下：

- 用户 Git 储存库结构：

```
<user_repository_root>
├── longhorn
│   └── fleet.yaml
└── longhorn-crd
    └── fleet.yaml
```

- 填充了用户 `Longhorn` 数据的 `fleet.yaml` 内容：

```
defaultNamespace: longhorn-system

helm:
  # timeoutSeconds: 10
  releaseName: "longhorn"
  chart: "longhorn"
  repo: "https://charts.rancher.io/"
```

```
  version: "106.2.0+up1.8.1"
  takeOwnership: true
  # custom chart value overrides
  values:
    # Example for user provided custom values content
    defaultSettings:
      deletingConfirmationFlag: true

# https://fleet.rancher.io/bundle-diffs
diff:
  comparePatches:
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: engineimages.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: nodes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: volumes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
```

注意

上面只是一些示例值，用于演示基于 `longhorn` chart 创建的自定义配置。**不应**将它们视为 `longhorn` chart 的部署指南。

36.1.6.2.1.2 部署您的 chart 的 Fleet

您可以使用 GitRepo（第 36.1.6.2.1.2.1 节 “GitRepo”）或捆绑包（第 36.1.6.2.1.2.2 节 “捆绑包”）来部署 chart 的 Fleet。

注意

部署 Fleet 时，如果收到资源已修改消息，请确保在 Fleet 的 `diff` 部分添加相应的 `comparePatches` 项。有关详细信息，请参见 Generating Diffs to Ignore Modified GitRepos (https://fleet.rancher.io/bundle-diffs)。

36.1.6.2.1.2.1 GitRepo

Fleet 的 GitRepo (https://fleet.rancher.io/ref-gitrepo) 资源包含如何访问 chart 的 Fleet 资源以及需要将这些资源应用于哪些群集的相关信息。

可以通过 Rancher UI (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui) 或者通过将资源手动部署到管理群集来部署 (https://fleet.rancher.io/tut-deployment) `GitRepo` 资源。

用于**手动**部署的 **Longhorn** `GitRepo` 资源示例：

```
apiVersion: fleet.cattle.io/v1alpha1
kind: GitRepo
metadata:
  name: longhorn-git-repo
  namespace: fleet-default
spec:
  # If using a tag
  # revision: user_repository_tag
  #
  # If using a branch
  # branch: user_repository_branch
  paths:
  # As seen in the 'Prepare your Fleet resources' example
  - longhorn
  - longhorn-crd
  repo: user_repository_url
```

```
targets:
# Match all clusters
- clusterSelector: {}
```

36.1.6.2.1.2.2 捆绑包

捆绑包 (https://fleet.rancher.io/bundle-add) 资源包含需要由 Fleet 部署的原始 Kubernetes 资源。一般情况下，建议使用 `GitRepo` 方法，但对于无法支持本地 Git 服务器的隔离环境，捆绑包可以帮助您将 Helm chart Fleet 传播到目标群集。

捆绑包的部署可以通过 Rancher UI（`Continuous Delivery`（持续交付）→ Advanced（高级）→ Bundles（捆绑包）→ `Create from YAML`（基于 YAML 创建））进行，也可以通过在正确的 Fleet 名称空间中手动部署捆绑包资源来完成。有关 Fleet 名称空间的信息，请参见上游文档 (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups)。

可以利用 Fleet 的将 Helm Chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle) 方法创建用于 Edge Helm chart 的捆绑包。

下面的示例展示了如何基于 longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) 和 longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) Helm chart Fleet 模板创建捆绑包资源，以及如何手动将此捆绑包部署到管理群集。

注意

为了阐明工作流程，下面的示例使用了 suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) 目录结构。

1. 导航到 longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) chart Fleet 模板：

   ```
   cd fleets/day2/chart-templates/longhorn/longhorn
   ```

2. 创建 `targets.yaml` 文件，该文件将指示 Fleet 应将 Helm chart 部署到哪些群集：

```
cat > targets.yaml <<EOF
targets:
# Matches all downstream clusters
- clusterSelector: {}
EOF
```

若要更精细地选择下游群集，请参见映射到下游群集 (https://fleet.rancher.io/gitrepo-targets)。

3. 使用 fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet) 将 Longhorn Helm chart Fleet 转换为捆绑包资源。

注意

Fleet 的 CLI 可以从其版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2) **资产**页面 (`fleet-linux-amd64`) 获取。

对于 Mac 用户，有一个 fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae。

```
fleet apply --compress --targets-file=targets.yaml -n fleet-default -o -
 longhorn-bundle > longhorn-bundle.yaml
```

4. 导航到 longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) chart Fleet 模板：

```
cd fleets/day2/chart-templates/longhorn/longhorn-crd
```

5. 创建 `targets.yaml` 文件，该文件将指示 Fleet 应将 Helm chart 部署到哪些群集：

```
cat > targets.yaml <<EOF
targets:
# Matches all downstream clusters
- clusterSelector: {}
EOF
```

6. 使用 fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet) 将 Longhorn CRD Helm chart Fleet 转换为捆绑包资源。

```
fleet apply --compress --targets-file=targets.yaml -n fleet-default -o -
 longhorn-crd-bundle > longhorn-crd-bundle.yaml
```

7. 将 longhorn-bundle.yaml 和 longhorn-crd-bundle.yaml 文件部署到管理群集：

```
kubectl apply -f longhorn-crd-bundle.yaml
kubectl apply -f longhorn-bundle.yaml
```

按照以上步骤操作，将 SUSE Storage 部署到所有指定的下游群集上。

36.1.6.2.1.3 管理部署的 Helm chart

通过 Fleet 完成部署后，若要进行 Helm chart 升级，请参见第 36.1.6.2.2 节 “我想升级 Fleet 管理的 Helm chart”。

36.1.6.2.2 我想升级 Fleet 管理的 Helm chart

1. 确定需要将 chart 升级到哪个版本，以使其与目标 Edge 版本兼容。有关每个 Edge 版本兼容的 Helm chart 版本，可参见发行说明（第 52.1 节 “摘要”）。
2. 在 Fleet 监控的 Git 储存库中，根据发行说明（第 52.1 节 “摘要”）中所述，将 Helm chart 的 fleet.yaml 文件中的 chart **版本**和**储存库**更改为正确的值。
3. 提交更改并将其推送到储存库后，会触发所需 Helm chart 的升级

36.1.6.2.3 我要升级通过 EIB 部署的 Helm chart

第 11 章 “Edge Image Builder” 通过创建 HelmChart 资源并利用 RKE2 (https://docs.rke2.io/helm)/K3s (https://docs.k3s.io/helm) Helm 集成功能引入的 helm-controller 来部署 Helm chart。

为确保通过 EIB 部署的 Helm chart 成功升级，用户需要对相应的 HelmChart 资源执行升级操作。

下文提供了以下信息：

- 升级过程的一般概述（第 36.1.6.2.3.1 节 “概述”）。
- 必要的升级步骤（第 36.1.6.2.3.2 节 “升级步骤”）。
- 展示使用所述方法进行 Longhorn (https://longhorn.io) chart 升级的示例（第 36.1.6.2.3.3 节 “示例”）。
- 如何使用其他 GitOps 工具完成升级过程（第 36.1.6.2.3.4 节 “使用第三方 GitOps 工具进行 Helm chart 升级”）。

36.1.6.2.3.1 概述

通过 EIB 部署的 Helm chart 由名为 eib-charts-upgrader (https://github.com/suse-edge/fleet-examples/tree/release-3.3.0/fleets/day2/eib-charts-upgrader) 的 Fleet 执行升级。

该 Fleet 会处理**用户提供的**数据，以**更新**一组特定的 HelmChart 资源。

更新这些资源会触发 helm-controller (https://github.com/k3s-io/helm-controller)，后者会**升级**与修改后的 HelmChart 资源相关联的 Helm chart。

用户只需执行以下操作：

1. 从本地提取 (https://helm.sh/docs/helm/helm_pull/) 需要升级的每个 Helm chart 的归档。
2. 将这些归档传递给 generate-chart-upgrade-data.sh (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/generate-chart-upgrade-data.sh) `generate-chart-upgrade-data.sh` 脚本，该脚本会将这些归档中的数据包含到 `eib-charts-upgrader` Fleet 中。
3. 将 `eib-charts-upgrader` Fleet 部署到管理群集。此操作可以通过 GitRepo 或捆绑包资源来完成。

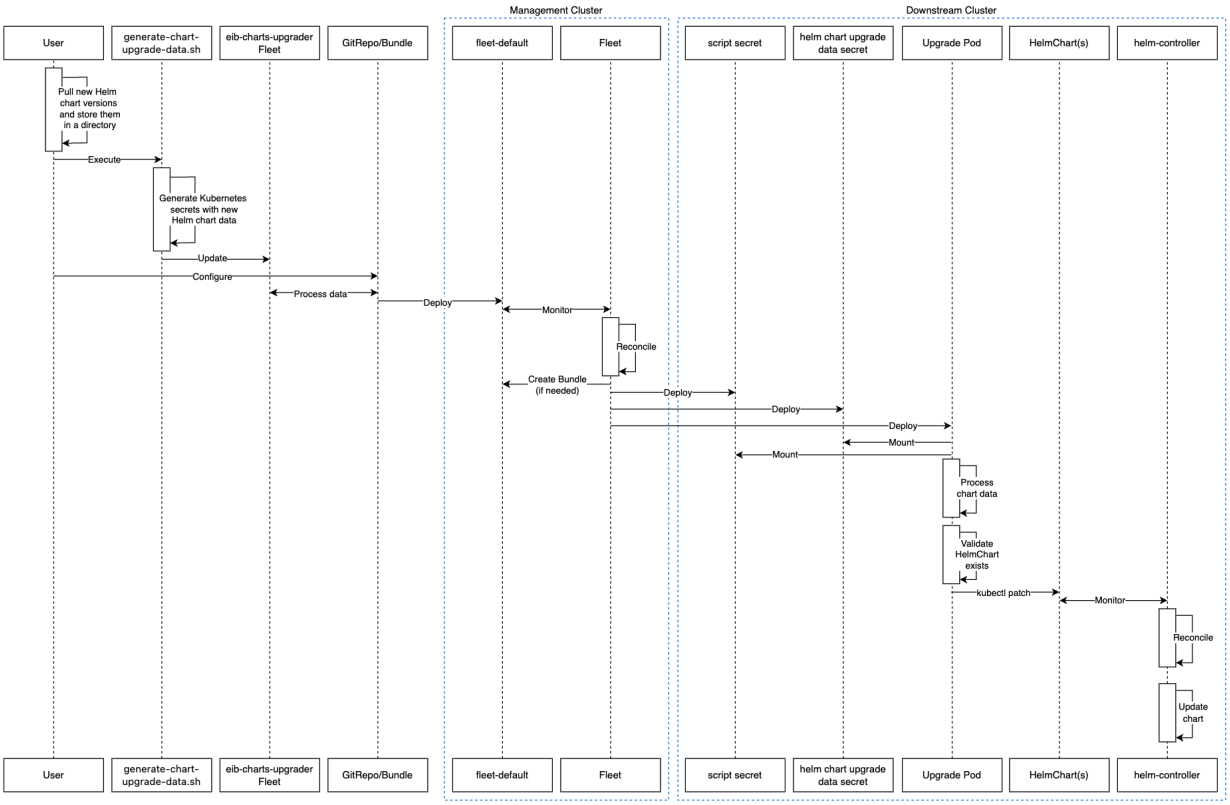
部署后，`eib-charts-upgrader` 会借助 Fleet 将其资源分发到目标下游群集。

这些资源包括：

1. 一组存储**用户提供的** Helm chart 数据的机密。
2. 一个会部署 Pod 的 Kubernetes 作业，该 Pod 会挂载前面提到的机密，并根据这些机密对相应的 HelmChart 资源进行修补 (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_patch/)。

如前文所述，这会触发 `helm-controller`，进而执行实际的 Helm chart 升级。

下面是上述流程的示意图：



36.1.6.2.3.2 升级步骤

1. 从正确的版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0)处克隆 `suse-edge/fleet-example` 储存库。
2. 创建用于存储所提取的 Helm chart 归档的目录。

```
mkdir archives
```

3. 在新创建的归档目录中，提取 (https://helm.sh/docs/helm/helm_pull/)要升级的 Helm chart 的归档：

```
cd archives
helm pull [chart URL | repo/chartname]

# Alternatively if you want to pull a specific version:
# helm pull [chart URL | repo/chartname] --version 0.0.0
```

4. 从目标版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0)的**资产**中下载 `generate-chart-upgrade-data.sh` 脚本。

5. 执行 `generate-chart-upgrade-data.sh` 脚本：

```
chmod +x ./generate-chart-upgrade-data.sh

./generate-chart-upgrade-data.sh --archive-dir /foo/bar/archives/ --fleet-path /foo/bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

对于 `--archive-dir` 目录中的每个 chart 归档，该脚本都会生成一个包含 chart 升级数据的 Kubernetes 机密 YAML 文件，并将其存储在 `--fleet-path` 指定的 Fleet 的 `base/secrets` 目录中。
`generate-chart-upgrade-data.sh` 脚本还会对 Fleet 进行其他修改，以确保生成的 Kubernetes 机密 YAML 文件能被 Fleet 部署的工作负载正确使用。

重要

用户不应更改 `generate-chart-upgrade-data.sh` 脚本生成的内容。

以下步骤因您运行的环境而异：

1. 对于支持 GitOps 的环境（例如：非隔离环境，或虽是隔离环境但支持本地 Git 服务器）：

   a. 将 `fleets/day2/eib-charts-upgrader` Fleet 复制到将用于 GitOps 的储存库中。

注意

确保该 Fleet 包含 `generate-chart-upgrade-data.sh` 脚本所做的更改。

b. 配置将用于提供 `eib-charts-upgrader` Fleet 所有资源的 `GitRepo` 资源。

i. 要通过 Rancher UI 进行 `GitRepo` 配置和部署，请参见在 Rancher UI 中访问 Fleet (https://ranchermanager.docs.rancher.com/v2.11/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui)。

ii. 有关 `GitRepo` 的手动配置和部署过程，请参见创建部署 (https://fleet.rancher.io/tut-deployment)。

2. 对于不支持 GitOps 的环境（例如，不允许使用本地 Git 服务器的隔离环境）：

a. 从 `rancher/fleet` 版本 (https://github.com/rancher/fleet/releases/tag/v0.12.2) 页面下载 `fleet-cli` 二进制文件。对于 Linux，请下载 `fleet-linux-amd64`。对于 Mac 用户，可以使用 Homebrew Formulae - fleet-cli (https://formulae.brew.sh/formula/fleet-cli)。

b. 导航到 `eib-charts-upgrader` Fleet：

```
cd /foo/bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

c. 创建指示 Fleet 在哪里部署资源的 `targets.yaml` 文件：

```
cat > targets.yaml <<EOF
targets:
# To match all downstream clusters
- clusterSelector: {}
EOF
```

有关如何映射目标群集的信息，请参见上游文档 (https://fleet.rancher.io/gitrepo-targets)。

d. 使用 `fleet-cli` 将 Fleet 转换为捆绑包资源：

```
fleet apply --compress --targets-file=targets.yaml -n fleet-default -o
 - eib-charts-upgrade > bundle.yaml
```

这将创建一个捆绑包 (`bundle.yaml`)，其中包含来自 `eib-charts-upgrader` Fleet 的所有模板资源。

有关 `fleet apply` 命令的详细信息，请参见 fleet apply (https://fleet.rancher.io/cli/fleet-cli/fleet_apply)。

有关将 Fleet 转换为捆绑包的详细信息，请参见将 Helm Chart 转换为捆绑包 (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle)。

e. 通过以下方式之一部署捆绑包：

   i. 通过 Rancher UI - 导航到 **Continuous Delivery（持续交付）→ Advanced（高级）→ Bundles（捆绑包）→ Create from YAML（基于 YAML 创建）**，然后粘贴 `bundle.yaml` 内容，或单击 `Read from File`（从文件读取）选项并传递文件。

   ii. 手动 - 在管理群集内手动部署 `bundle.yaml` 文件。

执行这些步骤可成功部署 `GitRepo`/捆绑包资源。资源将由 Fleet 拾取，且其内容将部署到用户在之前的步骤中指定的目标群集上。有关该过程的概述，请参见第 36.1.6.2.3.1 节 “概述”。

有关如何跟踪升级过程的信息，请参见第 36.1.6.2.3.3 节 “示例”。

重要

成功验证 chart 升级后，去除捆绑包/`GitRepo` 资源。

这将从下游群集中去除不再需要的升级资源，确保将来不会发生版本冲突。

注意

以下示例展示了如何在下游群集上将通过 EIB 部署的 Helm chart 从一个版本升级到另一个版本。请注意，本示例中使用的版本**并非**推荐版本。有关特定 Edge 版本的推荐版本，请参见发行说明（第 52.1 节 “摘要”）。

使用场景：

- 名为 `doc-example` 的群集正在运行旧版 Longhorn (https://longhorn.io)。
- 已使用以下映像定义**代码段**通过 EIB 部署群集：

```
kubernetes:
  helm:
    charts:
    - name: longhorn-crd
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    - name: longhorn
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    repositories:
    - name: rancher-charts
      url: https://charts.rancher.io/
...
```

- SUSE Storage 需要升级到与 Edge 3.3.1 版兼容的版本。这意味着它需要升级到 106.2.0+up1.8.1。
- 假设负责管理 doc-example 群集的管理群集是**隔离式**群集，不支持本地 Git 服务器，并且具有有效的 Rancher 设置。

按照升级步骤（第 36.1.6.2.3.2 节 “升级步骤”）进行操作：

1. 从 release-3.3.0 标记处克隆 suse-edge/fleet-example 储存库。

```
git clone -b release-3.3.0 https://github.com/suse-edge/fleet-examples.git
```

2. 创建用于存储 Longhorn 升级归档的目录。

```
mkdir archives
```

3. 提取所需的 Longhorn chart 归档版本：

```
# First add the Rancher Helm chart repository
helm repo add rancher-charts https://charts.rancher.io/

# Pull the Longhorn 1.8.1 CRD archive
helm pull rancher-charts/longhorn-crd --version 106.2.0+up1.8.1

# Pull the Longhorn 1.8.1 chart archive
helm pull rancher-charts/longhorn --version 106.2.0+up1.8.1
```

4. 在 archives 目录之外，从 suse-edge/fleet-examples 版本标记 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.3.0) 处下载 generate-chart-upgrade-data.sh 脚本。

5. 目录设置如下所示：

```
.
├── archives
|   ├── longhorn-106.2.0+up1.8.1.tgz
|   └── longhorn-crd-106.2.0+up1.8.1.tgz
├── fleet-examples
...
```

```
|   ├── fleets
|   |   ├── day2
|   |   |   ├── ...
|   |   |   ├── eib-charts-upgrader
|   |   |   |   ├── base
|   |   |   |   |   ├── job.yaml
|   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   ├── patches
|   |   |   |   |   |   └── job-patch.yaml
|   |   |   |   |   ├── rbac
|   |   |   |   |   |   ├── cluster-role-binding.yaml
|   |   |   |   |   |   ├── cluster-role.yaml
|   |   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   |   └── sa.yaml
|   |   |   |   |   └── secrets
|   |   |   |   |       ├── eib-charts-upgrader-script.yaml
|   |   |   |   |       └── kustomization.yaml
|   |   |   |   ├── fleet.yaml
|   |   |   |   └── kustomization.yaml
|   |   |   └── ...
|   |   └── ...
└── generate-chart-upgrade-data.sh
```

6. 执行 `generate-chart-upgrade-data.sh` 脚本：

```
# First make the script executable
chmod +x ./generate-chart-upgrade-data.sh

# Then execute the script
./generate-chart-upgrade-data.sh --archive-dir ./archives --fleet-path ./
fleet-examples/fleets/day2/eib-charts-upgrader
```

脚本执行后的目录结构如下所示：

```
.
├── archives
|   ├── longhorn-106.2.0+up1.8.1.tgz
|   └── longhorn-crd-106.2.0+up1.8.1.tgz
├── fleet-examples
```

```
...
|   ├── fleets
|   |   ├── day2
|   |   |   ├── ...
|   |   |   ├── eib-charts-upgrader
|   |   |   |   ├── base
|   |   |   |   |   ├── job.yaml
|   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   ├── patches
|   |   |   |   |   |   └── job-patch.yaml
|   |   |   |   |   ├── rbac
|   |   |   |   |   |   ├── cluster-role-binding.yaml
|   |   |   |   |   |   ├── cluster-role.yaml
|   |   |   |   |   |   ├── kustomization.yaml
|   |   |   |   |   |   └── sa.yaml
|   |   |   |   |   └── secrets
|   |   |   |   |       ├── eib-charts-upgrader-script.yaml
|   |   |   |   |       ├── kustomization.yaml
|   |   |   |   |       ├── longhorn-VERSION.yaml - secret created by the
 generate-chart-upgrade-data.sh script
|   |   |   |   |       └── longhorn-crd-VERSION.yaml - secret created by
 the generate-chart-upgrade-data.sh script
|   |   |   |   ├── fleet.yaml
|   |   |   |   └── kustomization.yaml
|   |   |   └── ...
|   |   └── ...
└── generate-chart-upgrade-data.sh
```

git 中更改的文件如下所示：

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 modified:   fleets/day2/eib-charts-upgrader/base/patches/job-patch.yaml
 modified:   fleets/day2/eib-charts-upgrader/base/secrets/
kustomization.yaml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

```
fleets/day2/eib-charts-upgrader/base/secrets/longhorn-VERSION.yaml
fleets/day2/eib-charts-upgrader/base/secrets/longhorn-crd-VERSION.yaml
```

7. 为 `eib-charts-upgrader` Fleet 创建捆绑包：

   a. 首先，导航到 Fleet：

   ```
   cd ./fleet-examples/fleets/day2/eib-charts-upgrader
   ```

   b. 然后创建 `targets.yaml` 文件：

   ```
   cat > targets.yaml <<EOF
   targets:
   - clusterName: doc-example
   EOF
   ```

   c. 接下来，使用 `fleet-cli` 二进制文件将 Fleet 转换为捆绑包：

   ```
   fleet apply --compress --targets-file=targets.yaml -n fleet-default -o
    - eib-charts-upgrade > bundle.yaml
   ```

   d. 现在，在您的管理群集计算机上传输 `bundle.yaml`。

8. 通过 Rancher UI 部署捆绑包：

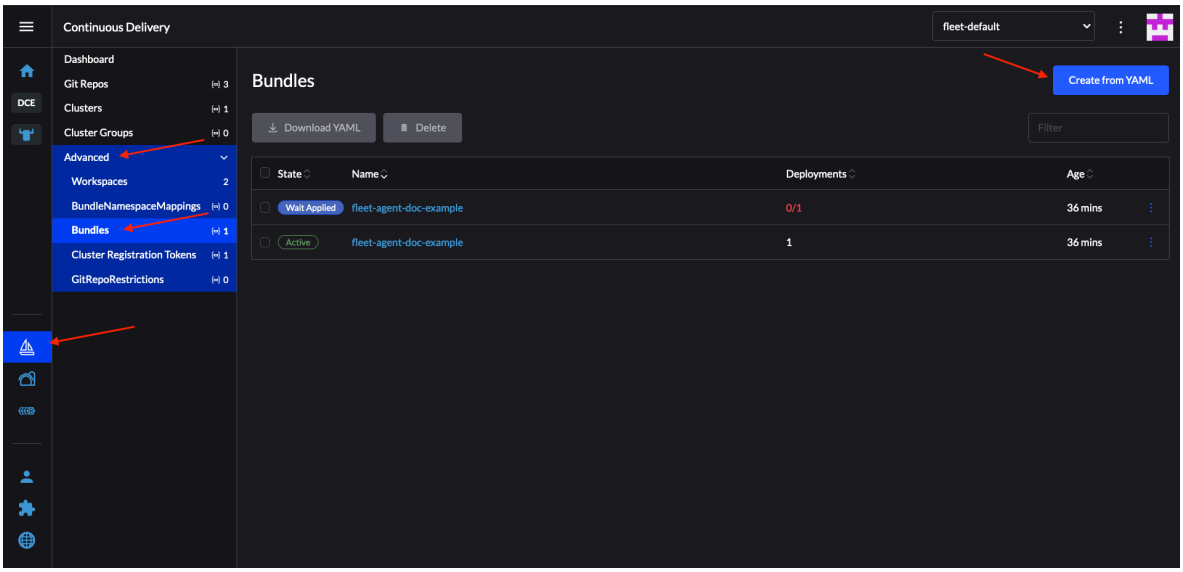


图 36.1： 通过 RANCHER UI 部署捆绑包

在此处选择 **Read from File**（从文件读取），并找到系统上的 `bundle.yaml` 文件。此时会在 Rancher UI 中自动填充捆绑包。

选择 **Create**（创建）。

9. 成功部署后，捆绑包如下所示：

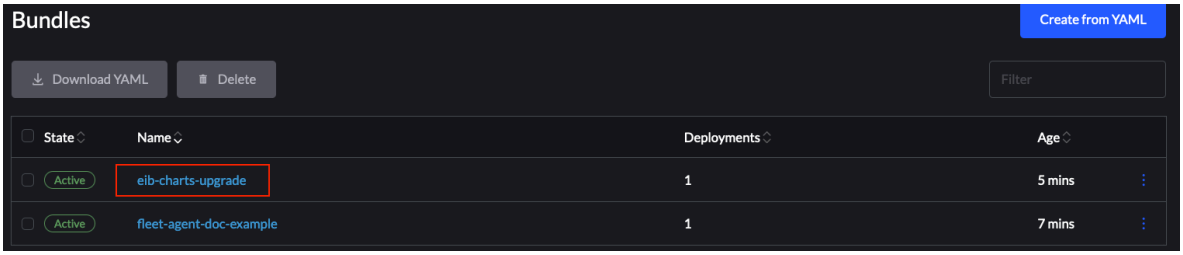
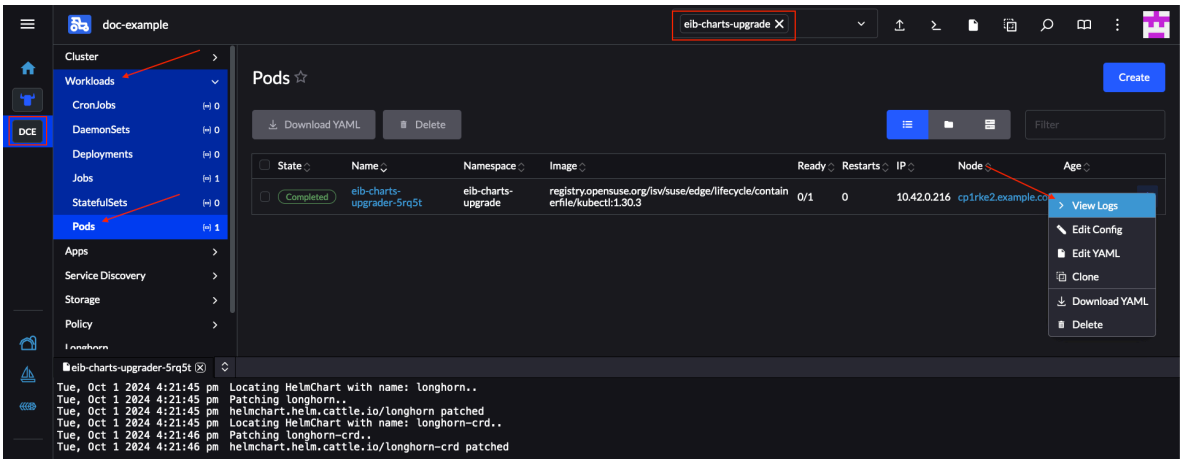


图 36.2：**已成功部署的捆绑包**

成功部署捆绑包后，要监控升级过程，请执行以下操作：

1. 验证升级 Pod 的日志：



2. 现在验证 helm-controller 针对升级过程创建的 Pod 日志：

   a. Pod 名称将使用以下模板 - `helm-install-longhorn-<random-suffix>`

   b. Pod 将位于部署了 `HelmChart` 资源的名称空间中。在本例中为 `kube-system` 名称空间。

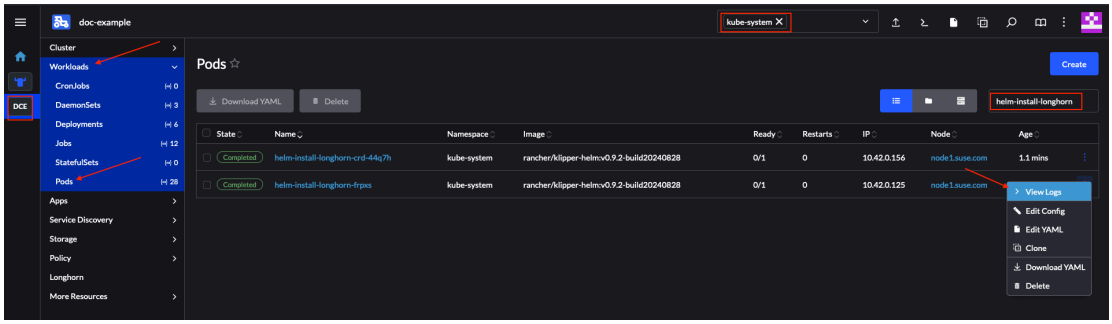


图 36.3：成功升级的 LONGHORN CHART 的日志

3. 导航到 Rancher 的 HelmChart 部分（More Resources（更多资源）→ HelmChart），验证 HelmChart 版本是否已更新。选择部署了该 chart 的名称空间，在本例中为 kube-system。

4. 最后检查 Longhorn Pod 是否正在运行。

在完成上述验证后，可以确定 Longhorn Helm chart 已升级到 106.2.0+up1.8.1 版本。

36.1.6.2.3.4 使用第三方 GitOps 工具进行 Helm chart 升级

在某些使用场景中，用户可能希望将此升级过程与 Fleet 以外的 GitOps 工作流程（例如 Flux）配合使用。

要生成升级过程所需的资源，可以使用 generate-chart-upgrade-data.sh 脚本将用户提供的数据填充到 eib-charts-upgrader Fleet。有关如何执行此操作的详细信息，请参见第 36.1.6.2.3.2 节 “升级步骤”。

完成所有设置后，可以使用 kustomize (https://kustomize.io) 生成一个可在群集中部署且完整有效的解决方案：

```
cd /foo/bar/fleets/day2/eib-charts-upgrader

kustomize build .
```

如果想将解决方案包含在 GitOps 工作流程中，可以去除 fleet.yaml 文件，并将其余内容用作有效的 Kustomize 设置。只是别忘了先运行 generate-chart-upgrade-data.sh 脚本，以便其可以将您想要升级到的 Helm chart 的数据填充到 Kustomize 设置中。

要了解如何使用此工作流程，可以查看第 36.1.6.2.3.1 节 “概述”和第 36.1.6.2.3.2 节 “升级步骤”。

VII 产品文档

此处提供 SUSE Edge for Telco 文档

37 SUSE Edge for Telco

SUSE Edge for Telco（以前称为“自适应电信基础架构平台”(ATIP)）是针对电信行业进行优化的边缘计算平台，可帮助电信公司实现创新并加速其网络的现代化改造。

SUSE Edge for Telco 是一个完整的电信云堆栈，可承载 5G 分组核心网和云化无线接入网等容器化网络功能 (CNF)。

- 能实现电信级规模下复杂边缘堆栈配置的零接触部署及生命周期管理自动化。
- 借助电信专用配置与工作负载，在电信级硬件上持续保障质量。
- 其组件专为边缘场景打造，因此资源占用更少，并且性能功耗比更高。
- 凭借不限供应商的 API 及 100% 开源特性，保持灵活的平台策略。

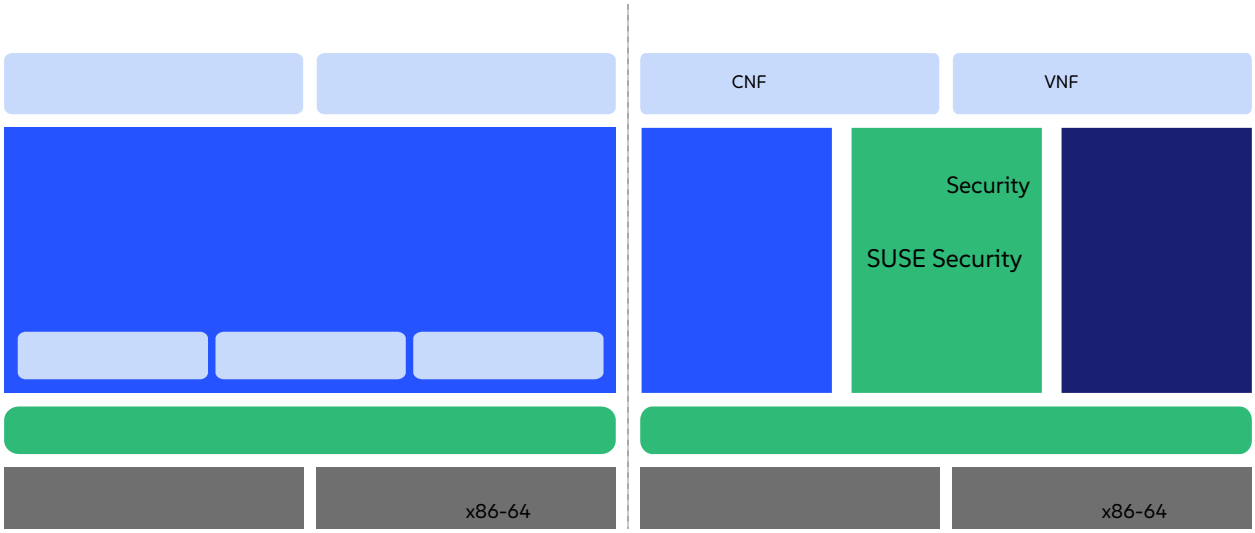
38 概念和体系结构

SUSE Edge for Telco 是旨在用于托管从核心到边缘规模的现代云原生电信应用程序的平台。

本页介绍了 SUSE Edge for Telco 的体系结构和使用的组件。

38.1 SUSE Edge for Telco 体系结构

下图显示了 SUSE Edge for Telco 的总体体系结构：



38.2 组件

有两个不同的区块 - 管理堆栈和运行时堆栈：

- **管理堆栈**：SUSE Edge for Telco 的这一组成部分用于管理运行时堆栈的置备和生命周期。管理堆栈包括以下组件：
  - 使用 Rancher 在公有云和私有云环境中进行多群集管理（第 5 章 “Rancher”）
  - 使用 Metal3（第 10 章 “Metal^3”）、MetalLB（第 19 章 “MetalLB”）和 CAPI (Cluster API) 基础架构提供程序来提供裸机支持
  - 全面的租户隔离和 IDP（身份提供程序）集成

- 第三方集成和扩展大型商城
- 不限供应商的 API 和丰富的提供商生态
- 控制 SUSE Linux Micro 事务更新
- GitOps 引擎，可以结合使用 Git 储存库和 Fleet（第 8 章 “Fleet”）来管理群集的生命周期

- **运行时堆栈**：SUSE Edge for Telco 的这一组成部分用于运行工作负载。
  - 包含 K3s（第 15 章 “K3s”）和 RKE2（第 16 章 “RKE2”）等安全轻量级发行版的 Kubernetes（RKE2 已针对政府用途和受监管行业进行强化、认证和优化）。
  - SUSE Security（第 18 章 “SUSE Security”），可用于实现映像漏洞扫描、深度数据包检测和群集内自动流量控制等安全功能。
  - 基于 SUSE Storage（第 17 章 “SUSE Storage”）构建的块存储，让您可以方便地使用云原生存储解决方案。
  - 基于 SUSE Linux Micro（第 9 章 “SUSE Linux Micro”）打造的优化操作系统，可提供安全、轻量且具备不可变特性（事务性文件系统）的容器运行环境。SUSE Linux Micro 适用于 AArch64 和 AMD64/Intel 64 体系结构，并为电信和边缘使用场景提供实时内核支持。

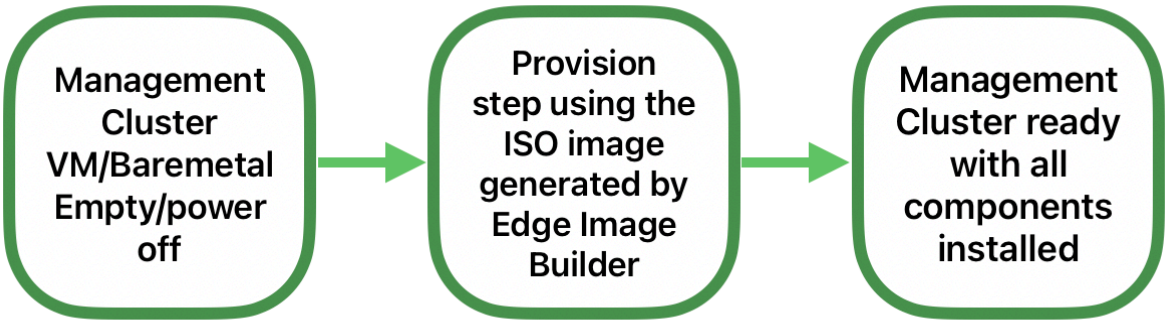
38.3 部署流程示例

下面是工作流程的简要示例，可帮助您了解管理组件与运行时组件之间的关系。

定向网络置备是可用于部署预配置了所有组件，并可以在无需人工干预的情况下运行工作负载的新下游群集的工作流程。

38.3.1 示例 1：部署装有所有组件的新管理群集

使用 Edge Image Builder（第 11 章 “Edge Image Builder”）创建包含管理堆栈的新 ISO 映像。然后，可以使用此 ISO 映像在 VM 或裸机上安装新管理群集。



注意

有关如何部署新管理群集的详细信息，请参见 SUSE Edge for Telco 管理群集指南（第 40 章 “设置管理群集”）。

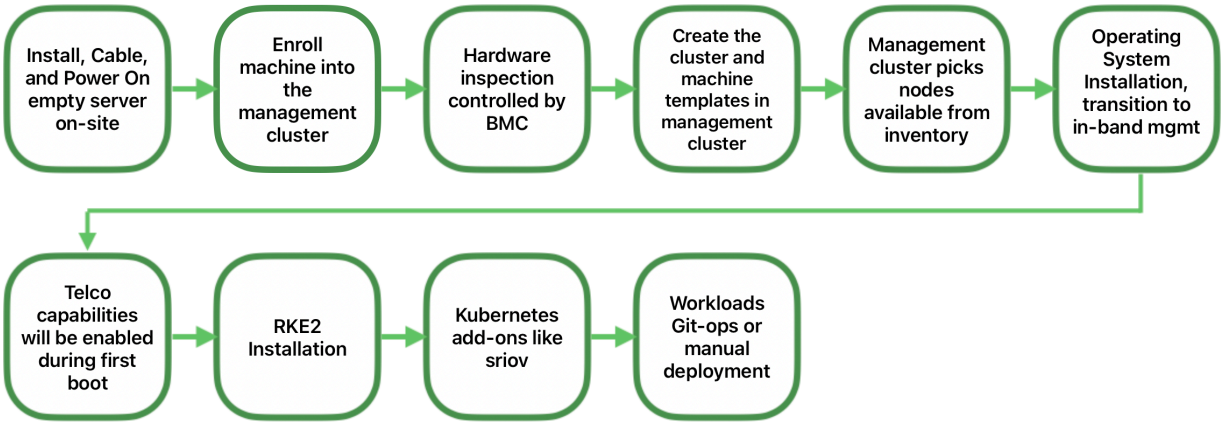
注意

有关如何使用 Edge Image Builder 的详细信息，请参见 Edge Image Builder 指南（第 3 章 “使用 Edge Image Builder 配置独立群集”）。

38.3.2 示例 2：使用电信配置文件部署单节点下游群集，使其能够运行电信工作负载

启动并运行管理群集后，我们可以使用该群集通过定向网络置备工作流程，来部署启用并配置了所有电信功能的单节点下游群集。

下图显示了部署该群集的概要工作流程：



注意

有关如何部署下游群集的详细信息，请参见 SUSE Edge for Telco 自动置备指南（第 42 章 “全自动定向网络置备”）。

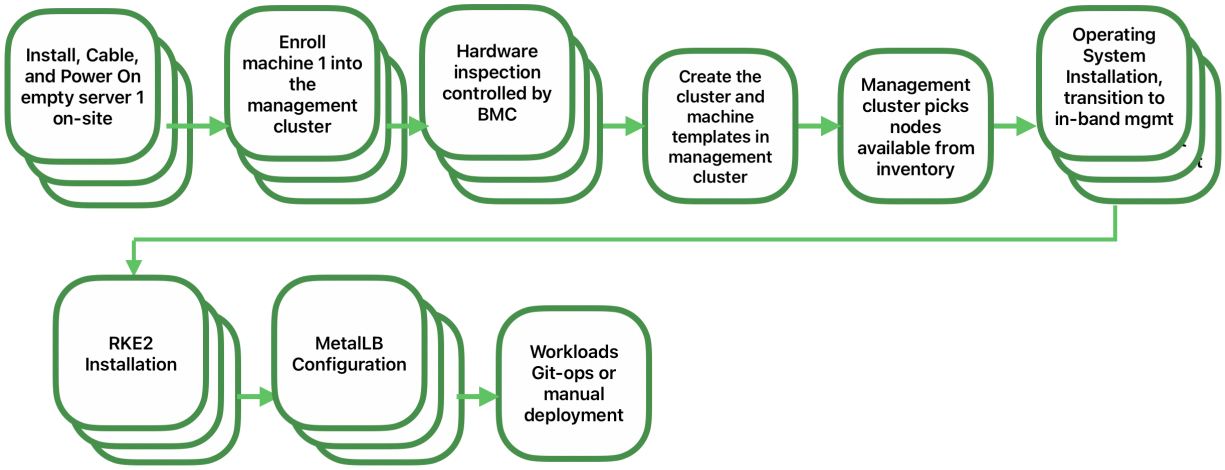
注意

有关电信功能的详细信息，请参见 SUSE Edge for Telco 电信功能指南（第 41 章 “电信功能配置”）。

38.3.3 示例 3：使用 MetalLB 作为负载平衡器部署高可用性下游群集

启动并运行管理群集后，我们可以使用该群集通过定向网络置备工作流程，来部署使用 `MetalLB` 作为负载平衡器的高可用性下游群集。

下图显示了部署该群集的概要工作流程：



注意

有关如何部署下游群集的详细信息，请参见 SUSE Edge for Telco 自动置备指南（第 42 章 “全自动定向网络置备”）。

注意

有关 `MetalLB` 的详细信息，请参见第 19 章 “MetalLB”。

39 要求和假设

39.1 硬件

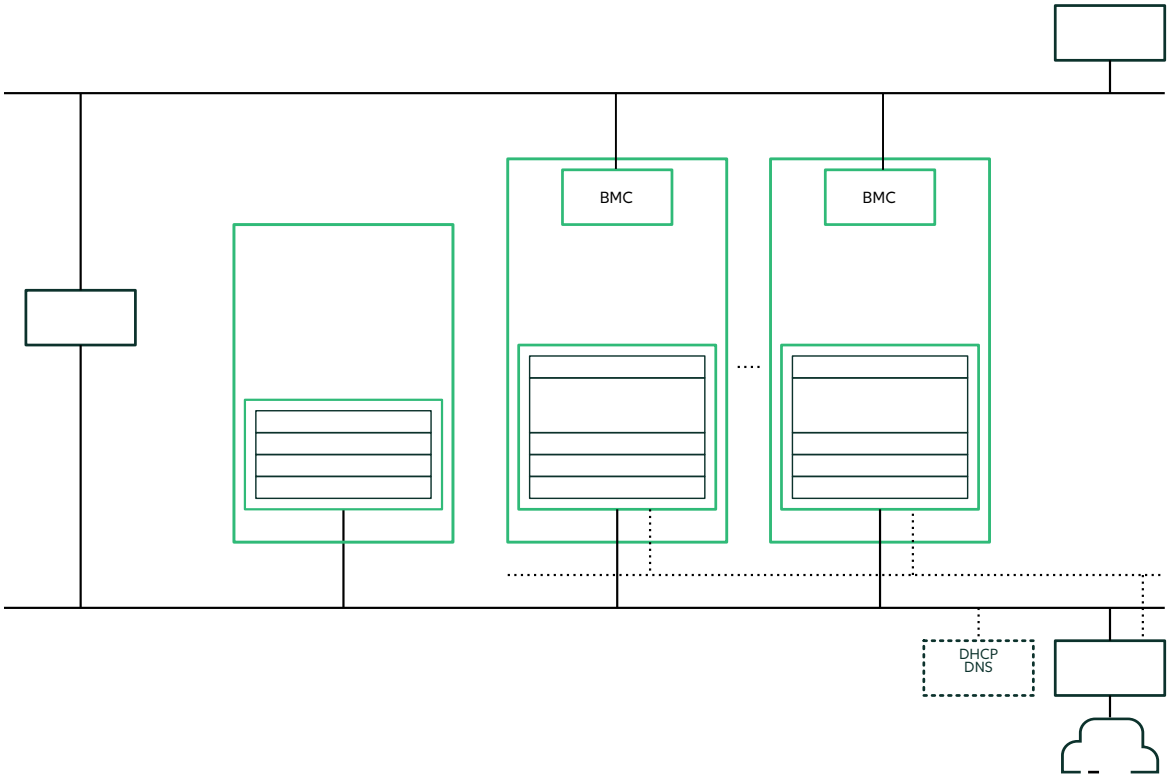
SUSE Edge for Telco 的硬件要求如下：

- **管理群集**：管理群集包含 SUSE Linux Micro、RKE2、SUSERancher Prime、Metal3 等组件，用于管理多个下游群集。服务器的硬件要求可能会因所要管理的下游群集数量而有所不同。
    - 服务器（VM 或裸机）的最低要求如下：
        - RAM：至少 8 GB（建议至少提供 16 GB）
        - CPU：至少 2 个（建议至少提供 4 个 CPU）
- **下游群集**：下游群集是用于运行电信工作负载的群集。需要满足特定的要求才能启用 SR-IOV、CPU 性能优化等特定电信功能。
    - SR-IOV：要以直通模式将 VF（虚拟功能）附加到 CNF/VNF，NIC 必须支持 SR-IOV，并且必须在 BIOS 中启用 VT-d/AMD-Vi。
    - CPU 处理器：要运行特定的电信工作负载，应该适配 CPU 处理器型号，以启用此参考表格（第 41 章 “电信功能配置”）中所述的大多数功能。
    - 使用虚拟媒体进行安装所要满足的固件要求：

服务器硬件	BMC 型号	管理
Dell 硬件	第 15 代	iDRAC9
Supermicro 硬件	01.00.25	Supermicro SMC - redfish
HPE 硬件	1.50	iLO6

39.2 网络

下图显示了电信环境的典型网络体系结构作为参考：



网络体系结构基于以下组件：

- **管理网络**：此网络用于管理下游群集节点。它用于进行带外管理。通常，此网络还会连接到独立的管理交换机，不过可以通过 VLAN 连接到同一服务交换机以隔离流量。
- **控制平面网络**：此网络用于下游群集节点与其上运行的服务之间的通讯。此网络还用于这些节点与外部服务（例如 DHCP 或 DNS 服务器）之间的通讯。在某些情况下，对于联网环境，交换机/路由器可以通过互联网处理流量。
- **其他网络**：在某些情况下，这些节点可以连接到其他网络以满足特定目的。

注意

要使用定向网络置备工作流程，管理群集必须与下游群集服务器基板管理控制器 (BMC) 建立网络连接，以便可以自动准备和置备主机。

39.3 服务（DHCP、DNS 等）

可能需要使用一些外部服务（例如 DHCP、DNS 等），具体取决于部署环境的类型：

- **联网环境**：在这种情况下，节点将连接到互联网（通过路由 L3 协议），外部服务将由客户提供。
- **离线/隔离环境**：在这种情况下，节点未建立互联网 IP 连接，因此需要通过其他服务在本地镜像定向网络置备工作流程所需的内容。
- **文件服务器**：文件服务器用于在执行定向网络置备工作流程期间存储将在下游群集节点上置备的操作系统映像。Metal3 Helm chart 可以部署媒体服务器来存储操作系统映像 — 请查看下文内容（注意），但也可以使用现有的本地 Web 服务器。

39.4 禁用 systemd 服务

对于电信工作负载，必须禁用或正确配置节点上运行的一些服务，以免对节点上运行的工作负载的性能产生任何影响（延迟）。

- rebootmgr 是当系统中存在未完成的更新时用于配置重引导策略的服务。对于电信工作负载，必须禁用或正确配置 rebootmgr 服务，以免在系统安排了更新时重引导节点，从而避免对节点上运行的服务造成任何影响。

注意

有关 rebootmgr 的详细信息，请参见 rebootmgr GitHub 储存库 (https://github.com/SUSE/rebootmgr)。

运行以下命令来校验使用的策略：

```
cat /etc/rebootmgr.conf
[rebootmgr]
window-start=03:30
window-duration=1h30m
strategy=best-effort
```

```
lock-group=default
```

可以运行以下命令来禁用 rebootmgr：

```
sed -i 's/strategy=best-effort/strategy=off/g' /etc/rebootmgr.conf
```

也可以使用 `rebootmgrctl` 命令：

```
rebootmgrctl strategy off
```

注意

可以使用定向网络置备工作流程来自动完成用于设置 `rebootmgr` 策略的此项配置。有关详细信息，请参见自动置备文档（第 42 章 “全自动定向网络置备”）。

- `transactional-update` 是一项允许在系统控制下进行自动更新的服务。对于电信工作负载，必须禁用自动更新，以免对节点上运行的服务产生任何影响。

要禁用自动更新，可以运行：

```
systemctl --now disable transactional-update.timer
systemctl --now disable transactional-update-cleanup.timer
```

- `fstrim` 是一种允许每周自动剪裁文件系统的服务。对于电信工作负载，必须禁用自动剪裁，以免对节点上运行的服务产生任何影响。

要禁用自动剪裁，可以运行：

```
systemctl --now disable fstrim.timer
```

40 设置管理群集

40.1 简介

管理群集是 SUSE Edge for Telco 的组成部分，用于管理运行时堆栈的置备和生命周期。从技术角度讲，管理群集包含以下组件：

- SUSE Linux Micro（操作系统），可以根据使用场景自定义某些配置，例如网络、存储、用户和内核参数。
- RKE2（Kubernetes 群集），可以根据使用场景将其配置为使用特定的 CNI 插件，例如 Multus、Cilium、Calico 等。
- Rancher（管理平台），用于管理群集的生命周期。
- Metal3，该组件用于管理裸机节点的生命周期。
- CAPI，该组件用于管理 Kubernetes 群集（下游群集）的生命周期。RKE2 CAPI 提供程序也用于管理 RKE2 群集的生命周期。

通过上述所有组件，管理群集可以管理下游群集的生命周期，并使用声明式方法来管理基础架构和应用程序。

注意

有关 SUSE Linux Micro 的详细信息，请参见第 9 章 “SUSE Linux Micro”

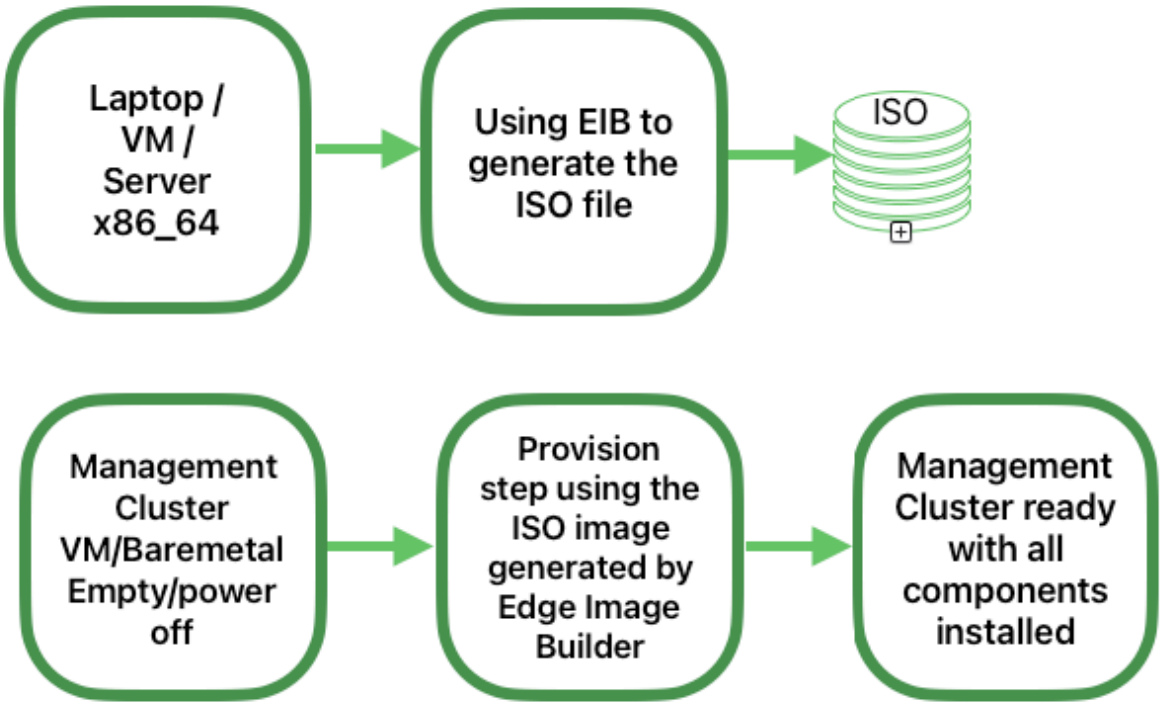
有关 RKE2 的详细信息，请参见第 16 章 “RKE2”

有关 Rancher 的详细信息，请参见第 5 章 “Rancher”

有关 Metal3 的详细信息，请参见第 10 章 “Metal^3”

40.2 设置管理群集的步骤

需要执行以下步骤来设置管理群集（使用单个节点）：



使用声明式方法设置管理群集需要执行以下主要步骤：

1. **为联网环境准备映像（第 40.3 节 “为联网环境准备映像”）**：第一步是准备包含所有必要配置的清单和文件，以便在联网环境中使用。

    - 联网环境的目录结构（第 40.3.1 节 “目录结构”）：此步骤创建一个目录结构，供 Edge Image Builder 用来存储配置文件和映像本身。
    - 管理群集定义文件（第 40.3.2 节 “管理群集定义文件”）：`mgmt-cluster.yaml` 文件是管理群集的主定义文件。其中包含有关所要创建的映像的以下信息：
        - 映像信息：与要使用基础映像创建的映像相关的信息。
        - 操作系统：要在映像中使用的操作系统配置。
        - Kubernetes：要在群集中使用的 Helm chart 和储存库、Kubernetes 版本、网络配置以及节点。
    - Custom 文件夹（第 40.3.3 节 “Custom 文件夹”）：`custom` 文件夹包含的配置文件和脚本供 Edge Image Builder 用来部署功能完备的管理群集。

    - Files 文件夹：包含管理群集要使用的配置文件。
    - Scripts 文件夹：包含管理群集要使用的脚本。
  - Kubernetes 文件夹（第 40.3.4 节 “Kubernetes 文件夹”）：`kubernetes` 文件夹包含管理群集要使用的配置文件。
    - Manifests 文件夹：包含管理群集要使用的清单。
    - Helm：包含管理群集要使用的 Helm 值文件。
    - Config 文件夹：包含管理群集要使用的配置文件。
  - Network 文件夹（第 40.3.5 节 “Network 文件夹”）：`network` 文件夹包含管理群集节点要使用的网络配置文件。

2. **为隔离环境准备映像（第 40.4 节 “为隔离环境准备映像”）**：此步骤将说明与非隔离场景相比，准备要在隔离场景中使用的清单和文件有哪些差别。
   - 定义文件中的修改（第 40.4.1 节 “定义文件中的修改”）：必须修改 `mgmt-cluster.yaml` 文件，以包含 `embeddedArtifactRegistry` 部分，并将 `images` 字段设置为要包含在 EIB 输出映像中的所有容器映像。
   - custom 文件夹中的修改（第 40.4.2 节 “custom 文件夹中的修改”）：必须修改 `custom` 文件夹，以包含用于在隔离环境中运行管理群集的资源。
     - 注册脚本：使用隔离环境时，必须去除 `custom/scripts/99-register.sh` 脚本。
   - Helm 值文件夹中的修改（第 40.4.3 节 “Helm 值文件夹中的修改”）：必须修改 `helm/values` 文件夹，以包含在隔离环境中运行管理群集所需的配置。
3. **创建映像（第 40.5 节 “映像创建”）**：此步骤使用 Edge Image Builder 工具创建映像（适用于联网场景和隔离场景）。在系统上运行 Edge Image Builder 工具之前，请先检查先决条件（第 11 章 “Edge Image Builder”）。
4. **置备管理群集（第 40.6 节 “置备管理群集”）**：此步骤使用上一步中创建的映像来置备管理群集（适用于联网场景和隔离场景）。可以使用便携式计算机、服务器、VM 或任何其他带有 USB 端口的 AMD64/Intel 64 系统来执行此步骤。

注意

有关 Edge Image Builder 的详细信息，请参见 Edge Image Builder（第 11 章 “Edge Image Builder”）和 Edge Image Builder 快速入门（第 3 章 “使用 Edge Image Builder 配置独立群集”）。

40.3 为联网环境准备映像

Edge Image Builder 用于为管理群集创建映像，在本文档中，我们将介绍设置管理群集所需的最低配置。

Edge Image Builder 在容器内运行，因此需要 Podman (https://podman.io) 或 Rancher Desktop (https://rancherdesktop.io) 等容器运行时。在本指南中，我们假设 Podman 可用。

此外，作为部署高可用性管理群集的先决条件，您需要在网络中预留三个 IP 地址：

- `apiVIP`，表示 API VIP 地址（用于访问 Kubernetes API 服务器）。
- `ingressVIP`，表示入口 VIP 地址（例如，供 Rancher UI 使用）。
- `metal3VIP`，表示 Metal3 VIP 地址。

40.3.1 目录结构

运行 EIB 时，将从主机挂载一个目录，因此首先需要创建一个目录结构，供 EIB 用来存储配置文件和映像本身。此目录的结构如下：

```
eib
├── mgmt-cluster.yaml
├── network
│   └── mgmt-cluster-node1.yaml
├── kubernetes
│   ├── manifests
│   │   ├── rke2-ingress-config.yaml
│   │   ├── neuvector-namespace.yaml
│   │   ├── ingress-l2-adv.yaml
```

```
│ │ └── ingress-ippool.yaml
│ ├── helm
│ │ └── values
│ │     ├── rancher.yaml
│ │     ├── neuvector.yaml
│ │     ├── metal3.yaml
│ │     └── certmanager.yaml
│ └── config
│     └── server.yaml
├── custom
│ ├── scripts
│ │ ├── 99-register.sh
│ │ ├── 99-mgmt-setup.sh
│ │ └── 99-alias.sh
│ └── files
│     ├── rancher.sh
│     ├── mgmt-stack-setup.service
│     ├── metal3.sh
│     └── basic-setup.sh
└── base-images
```

注意

必须从 SUSE Customer Center (https://scc.suse.com/) 或 SUSE 下载页面 (https://www.suse.com/download/sle-micro/) 下载 `SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso` 映像，并且必须将其存放在 `base-images` 文件夹下。

应检查该映像的 SHA256 校验和，确保它未遭篡改。可以在映像所下载到的位置找到校验和。

可以在 SUSE Edge GitHub 储存库中的“telco-examples”文件夹下 (https://github.com/suse-edge/atip) 找到目录结构的示例。

40.3.2 管理群集定义文件

`mgmt-cluster.yaml` 文件是管理群集的主定义文件。其中包含以下信息：

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
operatingSystem:
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
    encryptedPassword: $ROOT_PASSWORD
  packages:
    packageList:
    - git
    - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: 1.15.3
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn-crd
        version: 106.2.0+up1.8.1
        repositoryName: rancher-charts
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn
        version: 106.2.0+up1.8.1
        repositoryName: rancher-charts
        targetNamespace: longhorn-system
```

```
      createNamespace: true
      installationNamespace: kube-system
    - name: metal3
      version: 303.0.7+up0.11.5
      repositoryName: suse-edge-charts
      targetNamespace: metal3-system
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: metal3.yaml
    - name: rancher-turtles
      version: 303.0.4+up0.20.0
      repositoryName: suse-edge-charts
      targetNamespace: rancher-turtles-system
      createNamespace: true
      installationNamespace: kube-system
    - name: neuvector-crd
      version: 106.0.1+up2.8.6
      repositoryName: rancher-charts
      targetNamespace: neuvector
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: neuvector.yaml
    - name: neuvector
      version: 106.0.1+up2.8.6
      repositoryName: rancher-charts
      targetNamespace: neuvector
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: neuvector.yaml
    - name: rancher
      version: 2.11.2
      repositoryName: rancher-prime
      targetNamespace: cattle-system
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: rancher.yaml
  repositories:
    - name: jetstack
```

```
        url: https://charts.jetstack.io
      - name: rancher-charts
        url: https://charts.rancher.io/
      - name: suse-edge-charts
        url: oci://registry.suse.com/edge/charts
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
  network:
    apiHost: $API_HOST
    apiVIP: $API_VIP
  nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
```

为了解释 `mgmt-cluster.yaml` 定义文件中的字段和值，我们将此文件划分成了以下几个部分。

- 映像部分（定义文件）：

```
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
```

其中 `baseImage` 是从 SUSE Customer Center 或 SUSE 下载页面下载的原始映像。`outputImageName` 是将用于置备管理群集的新映像的名称。

- 操作系统部分（定义文件）：

```
operatingSystem:
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
```

```
    encryptedPassword: $ROOT_PASSWORD
  packages:
    packageList:
    - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

其中 `installDevice` 是用于安装操作系统的设备，`username` 和 `encryptedPassword` 是用于访问系统的身份凭证，`packageList` 是要安装的软件包列表（在安装过程中，需要在内部使用 `jq`），`sccRegistrationCode` 是在构建时用于获取软件包和依赖项的注册代码，可从 SUSE Customer Center 获取。可以如下所示使用 `openssl` 命令生成加密的口令：

```
openssl passwd -6 MyPassword!123
```

此命令会输出如下所示的内容：

```
$6$UrXB1sAGs46DOiSq$HSwi9GFJLCorm0J53nF2Sq8YEoyINhHcObHzX2R8h13mswUIsMwzx4eUzn/
rRx0QPV4JIb0eWCoNrxGiKH4R31
```

- Kubernetes 部分（定义文件）：

```
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: 1.15.3
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn-crd
        version: 106.2.0+up1.8.1
        repositoryName: rancher-charts
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn
        version: 106.2.0+up1.8.1
```

```
    repositoryName: rancher-charts
    targetNamespace: longhorn-system
    createNamespace: true
    installationNamespace: kube-system
  - name: metal3
    version: 303.0.7+up0.11.5
    repositoryName: suse-edge-charts
    targetNamespace: metal3-system
    createNamespace: true
    installationNamespace: kube-system
    valuesFile: metal3.yaml
  - name: rancher-turtles
    version: 303.0.4+up0.20.0
    repositoryName: suse-edge-charts
    targetNamespace: rancher-turtles-system
    createNamespace: true
    installationNamespace: kube-system
  - name: neuvector-crd
    version: 106.0.1+up2.8.6
    repositoryName: rancher-charts
    targetNamespace: neuvector
    createNamespace: true
    installationNamespace: kube-system
    valuesFile: neuvector.yaml
  - name: neuvector
    version: 106.0.1+up2.8.6
    repositoryName: rancher-charts
    targetNamespace: neuvector
    createNamespace: true
    installationNamespace: kube-system
    valuesFile: neuvector.yaml
  - name: rancher
    version: 2.11.2
    repositoryName: rancher-prime
    targetNamespace: cattle-system
    createNamespace: true
    installationNamespace: kube-system
    valuesFile: rancher.yaml
```

```
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-charts
        url: https://charts.rancher.io/
      - name: suse-edge-charts
        url: oci://registry.suse.com/edge/charts
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
    network:
      apiHost: $API_HOST
      apiVIP: $API_VIP
    nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
```

`helm` 部分包含要安装的 Helm chart 列表、要使用的储存库，以及所有 chart 和储存库的版本配置。

`network` 部分包含 RKE2 组件要使用的网络配置，例如 `apiHost` 和 `apiVIP`。`apiVIP` 必须是网络中未使用的 IP 地址，并且不属于 DHCP 池（如果使用 DHCP）。此外，如果我们在多节点群集中使用 `apiVIP`，apiVIP 将用于访问 Kubernetes API 服务器。`apiHost` 是 RKE2 组件要使用的 `apiVIP` 的名称解析。

`nodes` 部分包含要在群集中使用的节点列表。此示例使用的是单节点群集，但可以通过在列表中添加更多节点（通过取消注释相应的行），将其扩展为多节点群集。

注意

- 节点名称在群集中必须保持唯一。
- 可以选择使用 `initializer` 字段指定引导主机，如果不指定，列表中的第一个节点将会是引导主机。
- 需要网络配置时，节点名称必须与“Network”文件夹（第 40.3.5 节 “Network 文件夹”）中定义的主机名相同。

40.3.3 Custom 文件夹

`custom` 文件夹包含以下子文件夹：

```
...
├── custom
│ ├── scripts
│ │ ├── 99-register.sh
│ │ ├── 99-mgmt-setup.sh
│ │ └── 99-alias.sh
│ └── files
│     ├── rancher.sh
│     ├── mgmt-stack-setup.service
│     ├── metal3.sh
│     └── basic-setup.sh
...
```

- `custom/files` 文件夹包含管理群集要使用的配置文件。
- `custom/scripts` 文件夹包含管理群集要使用的脚本。

`custom/files` 文件夹包含以下文件：

- `basic-setup.sh`：包含 `Metal3`、`Rancher` 和 `MetalLB` 的配置参数。仅当您要更改要使用的名称空间时，才需修改此文件。

  ```
  #!/bin/bash
  ```

```
# Pre-requisites. Cluster already running
export KUBECTL="/var/lib/rancher/rke2/bin/kubectl"
export KUBECONFIG="/etc/rancher/rke2/rke2.yaml"

##################
# METAL3 DETAILS #
##################
export METAL3_CHART_TARGETNAMESPACE="metal3-system"

############
# METALLB #
############
export METALLBNAMESPACE="metallb-system"

############
# RANCHER #
############
export RANCHER_CHART_TARGETNAMESPACE="cattle-system"
export RANCHER_FINALPASSWORD="adminadminadmin"

die(){
  echo ${1} 1>&2
  exit ${2}
}
```

- metal3.sh：包含要使用的 Metal3 组件的配置（无需修改）。在将来的版本中将替换此脚本，以改用 Rancher Turtles 来简化配置。

```
#!/bin/bash
set -euo pipefail

BASEDIR="$(dirname "$0")"
source ${BASEDIR}/basic-setup.sh

METAL3LOCKNAMESPACE="default"
METAL3LOCKCMNAME="metal3-lock"

trap 'catch $? $LINENO' EXIT
```

```
catch() {
  if [ "$1" != "0" ]; then
    echo "Error $1 occurred on $2"
    ${KUBECTL} delete configmap ${METAL3LOCKCMNAME} -n
 ${METAL3LOCKNAMESPACE}
  fi
}

# Get or create the lock to run all those steps just in a single node
# As the first node is created WAY before the others, this should be enough
# TODO: Investigate if leases is better
if [ $(${KUBECTL} get cm -n ${METAL3LOCKNAMESPACE} ${METAL3LOCKCMNAME} -o
 name | wc -l) -lt 1 ]; then
  ${KUBECTL} create configmap ${METAL3LOCKCMNAME} -n ${METAL3LOCKNAMESPACE}
 --from-literal foo=bar
else
  exit 0
fi

# Wait for metal3
while ! ${KUBECTL} wait --for condition=ready -n
 ${METAL3_CHART_TARGETNAMESPACE} $(${KUBECTL} get pods -n
 ${METAL3_CHART_TARGETNAMESPACE} -l app.kubernetes.io/name=metal3-ironic -o
 name) --timeout=10s; do sleep 2 ; done

# Get the ironic IP
IRONICIP=$(${KUBECTL} get cm -n ${METAL3_CHART_TARGETNAMESPACE} ironic-bmo
 -o jsonpath='{.data.IRONIC_IP}')

# If LoadBalancer, use metallb, else it is NodePort
if [ $(${KUBECTL} get svc -n ${METAL3_CHART_TARGETNAMESPACE} metal3-metal3-
ironic -o jsonpath='{.spec.type}') == "LoadBalancer" ]; then
  # Wait for metallb
  while ! ${KUBECTL} wait --for condition=ready -n ${METALLBNAMESPACE}
 $(${KUBECTL} get pods -n ${METALLBNAMESPACE} -l app.kubernetes.io/
component=controller -o name) --timeout=10s; do sleep 2 ; done
```

```
  # Do not create the ippool if already created
  ${KUBECTL} get ipaddresspool -n ${METALLBNAMESPACE} ironic-ip-pool -o
 name || cat <<-EOF | ${KUBECTL} apply -f -
  apiVersion: metallb.io/v1beta1
  kind: IPAddressPool
  metadata:
    name: ironic-ip-pool
    namespace: ${METALLBNAMESPACE}
  spec:
    addresses:
    - ${IRONICIP}/32
    serviceAllocation:
      priority: 100
      serviceSelectors:
      - matchExpressions:
        - {key: app.kubernetes.io/name, operator: In, values: [metal3-
ironic]}
 EOF

  # Same for L2 Advs
  ${KUBECTL} get L2Advertisement -n ${METALLBNAMESPACE} ironic-ip-pool-l2-
adv -o name || cat <<-EOF | ${KUBECTL} apply -f -
  apiVersion: metallb.io/v1beta1
  kind: L2Advertisement
  metadata:
    name: ironic-ip-pool-l2-adv
    namespace: ${METALLBNAMESPACE}
  spec:
    ipAddressPools:
    - ironic-ip-pool
 EOF
fi

# If rancher is deployed
if [ $(${KUBECTL} get pods -n ${RANCHER_CHART_TARGETNAMESPACE} -l
 app=rancher -o name | wc -l) -ge 1 ]; then
  cat <<-EOF | ${KUBECTL} apply -f -
 apiVersion: management.cattle.io/v3
```

```
 kind: Feature
 metadata:
   name: embedded-cluster-api
 spec:
   value: false
 EOF

  # Disable Rancher webhooks for CAPI
  ${KUBECTL} delete --ignore-not-found=true
 mutatingwebhookconfiguration.admissionregistration.k8s.io mutating-
webhook-configuration
  ${KUBECTL} delete --ignore-not-found=true
 validatingwebhookconfigurations.admissionregistration.k8s.io validating-
webhook-configuration
  ${KUBECTL} wait --for=delete namespace/cattle-provisioning-capi-system --
timeout=300s
fi

# Clean up the lock cm

${KUBECTL} delete configmap ${METAL3LOCKCMNAME} -n ${METAL3LOCKNAMESPACE}
```

- rancher.sh：包含要使用的 Rancher 组件的配置（无需修改）。

```
#!/bin/bash
set -euo pipefail

BASEDIR="$(dirname "$0")"
source ${BASEDIR}/basic-setup.sh

RANCHERLOCKNAMESPACE="default"
RANCHERLOCKCMNAME="rancher-lock"

if [ -z "${RANCHER_FINALPASSWORD}" ]; then
  # If there is no final password, then finish the setup right away
  exit 0
fi
```

```
trap 'catch $? $LINENO' EXIT

catch() {
  if [ "$1" != "0" ]; then
    echo "Error $1 occurred on $2"
    ${KUBECTL} delete configmap ${RANCHERLOCKCMNAME} -n
 ${RANCHERLOCKNAMESPACE}
  fi
}

# Get or create the lock to run all those steps just in a single node
# As the first node is created WAY before the others, this should be
 enough
# TODO: Investigate if leases is better
if [ $(${KUBECTL} get cm -n ${RANCHERLOCKNAMESPACE}
 ${RANCHERLOCKCMNAME} -o name | wc -l) -lt 1 ]; then
  ${KUBECTL} create configmap ${RANCHERLOCKCMNAME} -n
 ${RANCHERLOCKNAMESPACE} --from-literal foo=bar
else
  exit 0
fi

# Wait for rancher to be deployed
while ! ${KUBECTL} wait --for condition=ready -n
 ${RANCHER_CHART_TARGETNAMESPACE} $(${KUBECTL} get pods -n
 ${RANCHER_CHART_TARGETNAMESPACE} -l app=rancher -o name) --
timeout=10s; do sleep 2 ; done
until ${KUBECTL} get ingress -n ${RANCHER_CHART_TARGETNAMESPACE}
 rancher > /dev/null 2>&1; do sleep 10; done

RANCHERBOOTSTRAPPASSWORD=$(${KUBECTL} get secret -n
 ${RANCHER_CHART_TARGETNAMESPACE} bootstrap-secret -o
 jsonpath='{.data.bootstrapPassword}' | base64 -d)
RANCHERHOSTNAME=$(${KUBECTL} get ingress -n
 ${RANCHER_CHART_TARGETNAMESPACE} rancher -o
 jsonpath='{.spec.rules[0].host}')

# Skip the whole process if things have been set already
```

```
if [ -z $(${KUBECTL} get settings.management.cattle.io first-login -
ojsonpath='{.value}') ]; then
  # Add the protocol
  RANCHERHOSTNAME="https://${RANCHERHOSTNAME}"
  TOKEN=""
  while [ -z "${TOKEN}" ]; do
    # Get token
    sleep 2
    TOKEN=$(curl -sk -X POST ${RANCHERHOSTNAME}/v3-public/
localProviders/local?action=login -H 'content-type:
 application/json' -d "{\"username\":\"admin\",\"password\":
\"${RANCHERBOOTSTRAPPASSWORD}\"}" | jq -r .token)
  done

  # Set password
  curl -sk ${RANCHERHOSTNAME}/v3/users?action=changepassword -H
 'content-type: application/json' -H "Authorization: Bearer $TOKEN" -
d "{\"currentPassword\":\"${RANCHERBOOTSTRAPPASSWORD}\",\"newPassword
\":\"${RANCHER_FINALPASSWORD}\"}"

  # Create a temporary API token (ttl=60 minutes)
  APITOKEN=$(curl -sk ${RANCHERHOSTNAME}/v3/token -H 'content-
type: application/json' -H "Authorization: Bearer ${TOKEN}" -d
 '{"type":"token","description":"automation","ttl":3600000}' | jq -
r .token)

  curl -sk ${RANCHERHOSTNAME}/v3/settings/server-url -H 'content-type:
 application/json' -H "Authorization: Bearer ${APITOKEN}" -X PUT -d
 "{\"name\":\"server-url\",\"value\":\"${RANCHERHOSTNAME}\"}"
  curl -sk ${RANCHERHOSTNAME}/v3/settings/telemetry-opt -X PUT -H
 'content-type: application/json' -H 'accept: application/json' -H
 "Authorization: Bearer ${APITOKEN}" -d '{"value":"out"}'
fi

# Clean up the lock cm
```

```
${KUBECTL} delete configmap ${RANCHERLOCKCMNAME} -n
 ${RANCHERLOCKNAMESPACE}
```

- `mgmt-stack-setup.service`：包含用于创建 systemd 服务，以便在首次引导期间运行脚本的配置（无需修改）。

```
[Unit]
Description=Setup Management stack components
Wants=network-online.target
# It requires rke2 or k3s running, but it will not fail if those
 services are not present
After=network.target network-online.target rke2-server.service
 k3s.service
# At least, the basic-setup.sh one needs to be present
ConditionPathExists=/opt/mgmt/bin/basic-setup.sh

[Service]
User=root
Type=forking
# Metal3 can take A LOT to download the IPA image
TimeoutStartSec=1800

ExecStartPre=/bin/sh -c "echo 'Setting up Management components...'"
# Scripts are executed in StartPre because Start can only run a single
 one
ExecStartPre=/opt/mgmt/bin/rancher.sh
ExecStartPre=/opt/mgmt/bin/metal3.sh
ExecStart=/bin/sh -c "echo 'Finished setting up Management
 components'"
RemainAfterExit=yes
KillMode=process
# Disable & delete everything
ExecStartPost=rm -f /opt/mgmt/bin/rancher.sh
ExecStartPost=rm -f /opt/mgmt/bin/metal3.sh
ExecStartPost=rm -f /opt/mgmt/bin/basic-setup.sh
ExecStartPost=/bin/sh -c "systemctl disable mgmt-stack-setup.service"
ExecStartPost=rm -f /etc/systemd/system/mgmt-stack-setup.service
```

```
[Install]
WantedBy=multi-user.target
```

`custom/scripts` 文件夹包含以下文件：

- `99-alias.sh` 脚本：包含管理群集在首次引导时用来加载 kubeconfig 文件的别名（无需修改）。

```
#!/bin/bash
echo "alias k=kubectl" >> /etc/profile.local
echo "alias kubectl=/var/lib/rancher/rke2/bin/kubectl" >> /etc/
profile.local
echo "export KUBECONFIG=/etc/rancher/rke2/rke2.yaml" >> /etc/profile.local
```

- `99-mgmt-setup.sh` 脚本：包含首次引导期间用于复制脚本的配置（无需修改）。

```
#!/bin/bash

# Copy the scripts from combustion to the final location
mkdir -p /opt/mgmt/bin/
for script in basic-setup.sh rancher.sh metal3.sh; do
 cp ${script} /opt/mgmt/bin/
done

# Copy the systemd unit file and enable it at boot
cp mgmt-stack-setup.service /etc/systemd/system/mgmt-stack-setup.service
systemctl enable mgmt-stack-setup.service
```

- `99-register.sh` 脚本：包含用于通过 SCC 注册代码注册系统的配置。必须正确设置 `${SCC_ACCOUNT_EMAIL}` 和 `${SCC_REGISTRATION_CODE}` 才能使用您的帐户注册系统。

```
#!/bin/bash
set -euo pipefail

# Registration https://www.suse.com/support/kb/doc/?id=000018564
if ! which SUSEConnect > /dev/null 2>&1; then
 zypper --non-interactive install suseconnect-ng
```

```
fi
SUSEConnect --email "${SCC_ACCOUNT_EMAIL}" --url "https://scc.suse.com" --
regcode "${SCC_REGISTRATION_CODE}"
```

40.3.4 Kubernetes 文件夹

kubernetes 文件夹包含以下子文件夹：

```
...
├── kubernetes
│ ├── manifests
│ │ ├── rke2-ingress-config.yaml
│ │ ├── neuvector-namespace.yaml
│ │ ├── ingress-l2-adv.yaml
│ │ └── ingress-ippool.yaml
│ ├── helm
│ │ └── values
│ │     ├── rancher.yaml
│ │     ├── neuvector.yaml
│ │     ├── metal3.yaml
│ │     └── certmanager.yaml
│ └── config
│     └── server.yaml
...
```

kubernetes/config 文件夹包含以下文件：

- server.yaml：默认安装的 CNI 插件是 Cilium，因此不需要创建此文件夹和文件。如果您需要自定义 CNI 插件，可以使用 kubernetes/config 文件夹中的 server.yaml 文件。该文件包含以下信息：

  ```
  cni:
  - multus
  ```

```
- cilium
```

注意

这是一个可选文件，用于定义某些 Kubernetes 自定义设置，例如要使用的 CNI 插件，或官方文档 (https://docs.rke2.io/install/configuration) 中所述的许多选项。

`kubernetes/manifests` 文件夹包含以下文件：

- `rke2-ingress-config.yaml`：包含用于为管理群集创建入口服务的配置（无需修改）。

```
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-ingress-nginx
  namespace: kube-system
spec:
  valuesContent: |-
    controller:
      config:
        use-forwarded-headers: "true"
        enable-real-ip: "true"
      publishService:
        enabled: true
      service:
        enabled: true
        type: LoadBalancer
        externalTrafficPolicy: Local
```

- `neuvector-namespace.yaml`：包含用于创建 `NeuVector` 名称空间的配置（无需修改）。

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    pod-security.kubernetes.io/enforce: privileged
```

```
  name: neuvector
```

- `ingress-l2-adv.yaml`：包含用于为 `MetalLB` 组件创建 `L2Advertisement` 的配置（无需修改）。

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ingress-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
    - ingress-ippool
```

- `ingress-ippool.yaml`：包含用于为 `rke2-ingress-nginx` 组件创建 `IPAddressPool` 的配置。必须正确设置 `${INGRESS_VIP}`，以定义预留给 `rke2-ingress-nginx` 组件使用的 IP 地址。

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ingress-ippool
  namespace: metallb-system
spec:
  addresses:
    - ${INGRESS_VIP}/32
  serviceAllocation:
    priority: 100
    serviceSelectors:
      - matchExpressions:
          - {key: app.kubernetes.io/name, operator: In, values: [rke2-
ingress-nginx]}
```

`kubernetes/helm/values` 文件夹包含以下文件：

- `rancher.yaml`：包含用于创建 `Rancher` 组件的配置。必须正确设置 `${INGRESS_VIP}`，以定义 `Rancher` 组件要使用的 IP 地址。用于访问 `Rancher` 组件的 URL 为 `https://rancher-${INGRESS_VIP}.sslip.io`。

```
hostname: rancher-${INGRESS_VIP}.sslip.io
bootstrapPassword: "foobar"
replicas: 1
global.cattle.psp.enabled: "false"
```

- `neuvector.yaml`：包含用于创建 `NeuVector` 组件的配置（无需修改）。

```
controller:
  replicas: 1
  ranchersso:
    enabled: true
manager:
  enabled: false
cve:
  scanner:
    enabled: false
    replicas: 1
k3s:
  enabled: true
crdwebhook:
  enabled: false
```

- `metal3.yaml`：包含用于创建 `Metal3` 组件的配置。必须正确设置 `${METAL3_VIP}`，以定义 `Metal3` 组件要使用的 IP 地址。

```
global:
  ironicIP: ${METAL3_VIP}
  enable_vmedia_tls: false
  additionalTrustedCAs: false
metal3-ironic:
  global:
    predictableNicNames: "true"
  persistence:
    ironic:
      size: "5Gi"
```

如果您想使用此 x86_64 管理群集部署 arm64 下游群集，需要在 `metal3.yaml` 文件的 `global` 部分添加 `deployArchitecture: arm64`：

```
global:
  ironicIP: ${METAL3_VIP}
  enable_vmedia_tls: false
  additionalTrustedCAs: false
  deployArchitecture: arm64
metal3-ironic:
  global:
    predictableNicNames: "true"
  persistence:
    ironic:
      size: "5Gi"
```

注意

在最新版本中，使用 `deployArchitecture: arm64` 存在一项限制。具体而言，如果您通过此指令完成下游 arm64 群集的部署，管理群集后续将只能部署该体系结构的群集。要部署 x86_64 和 arm64 两种体系结构的群集，您需要置备两个独立的管理群集。此限制将在未来版本中解除。

注意

媒体服务器是 Metal^3 中包含的可选功能（默认处于禁用状态）。要使用该 Metal3 功能，需要在前面所述的清单中配置该功能。要使用 Metal^3 媒体服务器，请指定以下变量：

- 在 global 部分，将 `enable_metal3_media_server` 设置为 `true` 以启用媒体服务器功能。
- 包含有关媒体服务器的以下配置，其中 ${MEDIA_VOLUME_PATH} 是媒体卷在媒体中的路径（例如 `/home/metal3/bmh-image-cache`）

```
metal3-media:
  mediaVolume:
    hostPath: ${MEDIA_VOLUME_PATH}
```

可以使用外部媒体服务器来存储映像，如果您要将该服务器与 TLS 配合使用，则需要修改以下配置：

- 将前面所述 `metal3.yaml` 文件中的 `additionalTrustedCAs` 设置为 `true`，以启用来自外部媒体服务器的附加可信 CA。
- 在 `kubernetes/manifests/metal3-cacert-secret.yaml` 文件夹中包含以下机密配置，以存储外部媒体服务器的 CA 证书。

```
apiVersion: v1
kind: Namespace
metadata:
  name: metal3-system
---
apiVersion: v1
kind: Secret
metadata:
  name: tls-ca-additional
  namespace: metal3-system
type: Opaque
data:
  ca-additional.crt: {{ additional_ca_cert | b64encode }}
```

`additional_ca_cert` 是外部媒体服务器的 base64 编码 CA 证书。可使用以下命令对证书进行编码并手动生成机密：

```
kubectl -n meta3-system create secret generic tls-ca-additional --from-
file=ca-additional.crt=./ca-additional.crt
```

- `certmanager.yaml`：包含用于创建 `Cert-Manager` 组件的配置（无需修改）。

```
installCRDs: "true"
```

40.3.5 Network 文件夹

`Network` 文件夹中的文件数量与管理群集中的节点数量相同。在本例中，我们只有一个节点，因此此文件夹中只有一个文件，名为 `mgmt-cluster-node1.yaml`。该文件的名称必须与 `mgmt-cluster.yaml` 定义文件的上述 network/node 部分中定义的主机名一致。

如果您需要自定义网络配置，例如要使用特定的静态 IP 地址（无 DHCP 的方案），可以使用 `network` 文件夹中的 `mgmt-cluster-node1.yaml` 文件。该文件包含以下信息：

- `${MGMT_GATEWAY}`：网关 IP 地址。
- `${MGMT_DNS}`：DNS 服务器 IP 地址。
- `${MGMT_MAC}`：网络接口的 MAC 地址。
- `${MGMT_NODE_IP}`：管理群集的 IP 地址。

```
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: ${MGMT_GATEWAY}
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - ${MGMT_DNS}
    - 8.8.8.8
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
```

```
    address:
    - ip: ${MGMT_NODE_IP}
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
```

如果您要使用 DHCP 获取 IP 地址，可使用以下配置（必须使用 `${MGMT_MAC}` 变量正确设置 MAC 地址）：

```
## This is an example of a dhcp network configuration for a management cluster
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
    dhcp: true
    enabled: true
  ipv6:
    enabled: false
```

注意

- 根据管理群集中的节点数，您可以创建更多文件（例如 `mgmt-cluster-node2.yaml`、`mgmt-cluster-node3.yaml` 等）来配置其余节点。
- `routes` 部分用于定义管理群集的路由表。

40.4 为隔离环境准备映像

本节介绍如何为隔离环境准备映像，其中只说明了与前面几节内容存在的差别。为隔离环境准备映像需要对上一节（为联网环境准备映像（第 40.3 节 “为联网环境准备映像”））的内容进行以下更改：

- 必须修改 `mgmt-cluster.yaml` 文件，以包含 `embeddedArtifactRegistry` 部分，并将 `images` 字段设置为要包含在 EIB 输出映像中的所有容器映像。
- 必须修改 `mgmt-cluster.yaml` 文件，以包含 `rancher-turtles-airgap-resources` Helm chart。
- 使用隔离环境时，必须去除 `custom/scripts/99-register.sh` 脚本。

40.4.1 定义文件中的修改

必须修改 `mgmt-cluster.yaml` 文件，以包含 `embeddedArtifactRegistry` 部分。在此部分中，`images` 字段必须设置为要包含在输出映像中的所有容器映像的列表。

注意

以下是包含 `embeddedArtifactRegistry` 部分的 `mgmt-cluster.yaml` 文件示例。请确保列出的映像包含您所需的组件版本。

此外，还必须添加 `rancher-turtles-airgap-resources` Helm chart，以创建 Rancher Turtles 隔离文档 (https://documentation.suse.com/cloudnative/cluster-api/v0.19/en/getting-started/air-gapped-environment.html) 中所述的资源。还需要 rancher-turtles chart 的 turtles.yaml 值文件来指定必要的配置。

```
apiVersion: 1.2
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
operatingSystem:
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
    encryptedPassword: $ROOT_PASSWORD
  packages:
```

```
    packageList:
    - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
kubernetes:
  version: v1.32.4+rke2r1
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: 1.15.3
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn-crd
        version: 106.2.0+up1.8.1
        repositoryName: rancher-charts
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
      - name: longhorn
        version: 106.2.0+up1.8.1
        repositoryName: rancher-charts
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
      - name: metal3
        version: 303.0.7+up0.11.5
        repositoryName: suse-edge-charts
        targetNamespace: metal3-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: metal3.yaml
      - name: rancher-turtles
        version: 303.0.4+up0.20.0
        repositoryName: suse-edge-charts
        targetNamespace: rancher-turtles-system
        createNamespace: true
```

```
      installationNamespace: kube-system
      valuesFile: turtles.yaml
    - name: rancher-turtles-airgap-resources
      version: 303.0.4+up0.20.0
      repositoryName: suse-edge-charts
      targetNamespace: rancher-turtles-system
      createNamespace: true
      installationNamespace: kube-system
    - name: neuvector-crd
      version: 106.0.1+up2.8.6
      repositoryName: rancher-charts
      targetNamespace: neuvector
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: neuvector.yaml
    - name: neuvector
      version: 106.0.1+up2.8.6
      repositoryName: rancher-charts
      targetNamespace: neuvector
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: neuvector.yaml
    - name: rancher
      version: 2.11.2
      repositoryName: rancher-prime
      targetNamespace: cattle-system
      createNamespace: true
      installationNamespace: kube-system
      valuesFile: rancher.yaml
  repositories:
    - name: jetstack
      url: https://charts.jetstack.io
    - name: rancher-charts
      url: https://charts.rancher.io/
    - name: suse-edge-charts
      url: oci://registry.suse.com/edge/charts
    - name: rancher-prime
      url: https://charts.rancher.com/server-charts/prime
```

```
    network:
      apiHost: $API_HOST
      apiVIP: $API_VIP
    nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
#       type: server
embeddedArtifactRegistry:
  images:
    - name: registry.suse.com/rancher/hardened-cluster-autoscaler:v1.9.0-
build20241203
    - name: registry.suse.com/rancher/hardened-cni-plugins:v1.6.2-build20250306
    - name: registry.suse.com/rancher/hardened-coredns:v1.12.1-build20250401
    - name: registry.suse.com/rancher/hardened-k8s-metrics-server:v0.7.2-
build20250110
    - name: registry.suse.com/rancher/hardened-multus-cni:v4.2.0-build20250326
    - name: registry.suse.com/rancher/klipper-helm:v0.9.5-build20250306
    - name: registry.suse.com/rancher/mirrored-cilium-cilium:v1.17.3
    - name: registry.suse.com/rancher/mirrored-cilium-operator-generic:v1.17.3
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-attacher:v4.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-node-driver-
registrar:v2.13.0
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-provisioner:v5.2.0
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-resizer:v1.13.2
    - name: registry.suse.com/rancher/mirrored-longhornio-csi-snapshotter:v8.2.0
    - name: registry.suse.com/rancher/mirrored-longhornio-livenessprobe:v2.15.0
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-engine:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-instance-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-
manager:v1.8.1
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-share-
manager:v1.8.1
```

```
    - name: registry.suse.com/rancher/mirrored-longhornio-longhorn-ui:v1.8.1
    - name: registry.suse.com/rancher/mirrored-sig-storage-snapshot-
controller:v8.2.0
    - name: registry.suse.com/rancher/neuvector-compliance-config:1.0.5
    - name: registry.suse.com/rancher/neuvector-controller:5.4.4
    - name: registry.suse.com/rancher/neuvector-enforcer:5.4.4
    - name: registry.suse.com/rancher/nginx-ingress-controller:v1.12.1-hardened3
    - name: registry.rancher.com/rancher/cluster-api-addon-provider-
fleet:v0.10.0
    - name: registry.rancher.com/rancher/cluster-api-operator:v0.17.0
    - name: registry.rancher.com/rancher/fleet-agent:v0.12.3
    - name: registry.rancher.com/rancher/fleet:v0.12.3
    - name: registry.rancher.com/rancher/hardened-node-feature-
discovery:v0.15.7-build20250425
    - name: registry.rancher.com/rancher/rancher-webhook:v0.7.2
    - name: registry.rancher.com/rancher/rancher/turtles:v0.20.0
    - name: registry.rancher.com/rancher/rancher:v2.11.2
    - name: registry.rancher.com/rancher/shell:v0.4.1
    - name: registry.rancher.com/rancher/system-upgrade-controller:v0.15.2
    - name: registry.suse.com/rancher/cluster-api-controller:v1.9.5
    - name: registry.suse.com/rancher/cluster-api-provider-metal3:v1.9.3
    - name: registry.suse.com/rancher/cluster-api-provider-rke2-
bootstrap:v0.16.1
    - name: registry.suse.com/rancher/cluster-api-provider-rke2-
controlplane:v0.16.1
    - name: registry.suse.com/rancher/hardened-sriov-network-operator:v1.5.0-
build20250425
    - name: registry.suse.com/rancher/ip-address-manager:v1.9.4
    - name: registry.rancher.com/rancher/kubectl:v1.32.2
```

40.4.2 custom 文件夹中的修改

- 使用隔离环境时，必须去除 `custom/scripts/99-register.sh` 脚本。如目录结构中所示，`99-register.sh` 脚本并未包含在 `custom/scripts` 文件夹中。

40.4.3 Helm 值文件夹中的修改

- `turtles.yaml`：包含为 Rancher Turtles 指定隔离操作所需的配置，请注意，具体配置取决于 rancher-turtles-airgap-resources chart 的安装。

```
cluster-api-operator:
  cluster-api:
    core:
      fetchConfig:
        selector: "{\"matchLabels\": {\"provider-components\": \"core\"}}"
    rke2:
      bootstrap:
        fetchConfig:
          selector: "{\"matchLabels\": {\"provider-components\": \"rke2-
bootstrap\"}}"
      controlPlane:
        fetchConfig:
          selector: "{\"matchLabels\": {\"provider-components\": \"rke2-
control-plane\"}}"
    metal3:
      infrastructure:
        fetchConfig:
          selector: "{\"matchLabels\": {\"provider-components\":
 \"metal3\"}}"
```

40.5 映像创建

按照前面的章节准备好目录结构后（适用于联网场景和隔离场景），运行以下命令来构建映像：

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
 build --definition-file mgmt-cluster.yaml
```

这会创建 ISO 输出映像文件，根据前面所述的映像定义，本例中该文件是 `eib-mgmt-cluster-image.iso`。

40.6 置备管理群集

上图包含前面介绍的所有组件，可以使用虚拟机或裸机服务器（使用虚拟媒体功能）根据此图置备管理群集。

41 电信功能配置

本章将阐释通过 SUSE Edge for Telco 部署的群集上与电信相关的功能配置。

将使用有关自动置备的一章（第 42 章 “全自动定向网络置备”）中所述的定向网络置备部署方法。

本章涵盖以下主题：

- 实时内核映像（第 41.1 节 “实时内核映像”）：实时内核使用的内核映像。
- 实现低延迟和高性能需指定的内核参数（第 41.2 节 “实现低延迟和高性能需指定的内核参数”）：为了在运行电信工作负载时实现最高性能和低延迟，实时内核需使用的内核参数。
- 通过 Tuned 和内核参数实现 CPU 绑定（第 41.3 节 “通过 Tuned 和内核参数实现 CPU 绑定”）：通过内核参数和 Tuned 配置文件隔离 CPU。
- CNI 配置（第 41.4 节 “CNI 配置”）：Kubernetes 群集使用的 CNI 配置。
- SR-IOV 配置（第 41.5 节 “SR-IOV”）：Kubernetes 工作负载使用的 SR-IOV 配置。
- DPDK 配置（第 41.6 节 “DPDK”）：系统使用的 DPDK 配置。
- vRAN 加速卡（第 41.7 节 “vRAN 加速 (Intel ACC100/ACC200)”）：Kubernetes 工作负载使用的加速卡配置。
- 大页（第 41.8 节 “大页”）：Kubernetes 工作负载使用的大页配置。
- Kubernetes 上的 CPU 绑定（第 41.9 节 “在 Kubernetes 上进行 CPU 绑定”）：配置 Kubernetes 和应用程序以利用 CPU 绑定的优势。
- 可感知 NUMA 的调度配置（第 41.10 节 “可感知 NUMA 的调度”）：Kubernetes 工作负载使用的可感知 NUMA 的调度配置。
- MetalLB 配置（第 41.11 节 “MetalLB”）：Kubernetes 工作负载使用的 MetalLB 配置。

- 专用注册表配置（第 41.12 节 “专用注册表配置”）：Kubernetes 工作负载使用的专用注册表配置。
- 精确时间协议配置（第 41.13 节 “精确时间协议”）：用于运行电信行业 PTP 配置的配置文件。

41.1 实时内核映像

实时内核映像不一定比标准内核更好。它是针对特定使用场景进行微调的另一种内核。经过微调的实时内核可以降低延迟，但代价是降低了吞吐量。不建议将实时内核用于一般用途，但在本例中，则建议为电信工作负载使用该内核，因为在该场景下，延迟是一项关键考虑因素。

实时内核有四大特性：

- 确定性执行：
  获得更高的可预测性 — 确保关键业务流程每次都能及时完成并提供高质量的服务，即使系统负载极高的情况下也不例外。通过围隔出关键系统资源供高优先级流程使用，可以确保为时间敏感型应用程序提供更高的可预测性。
- 低抖动：
  基于高确定性技术的低抖动有助于应用程序与现实世界保持同步。这可以为那些需要持续重复计算的服务提供帮助。
- 优先级继承：
  优先级继承是指当较高优先级的进程在完成其任务之前需要先等待较低优先级的进程完成时，较低优先级的进程可以提升为较高优先级的功能。SUSE Linux Enterprise Real Time 解决了任务关键型进程的优先级倒置问题。
- 线程中断：
  在通用操作系统中以中断模式运行的进程不可抢占。在 SUSE Linux Enterprise Real Time 中，这些中断已由可中断的内核线程封装，并允许用户定义的较高优先级进程抢占硬中断和软中断。
  在本例中，如果您安装了 `SUSE Linux Micro RT` 之类的实时映像，那么就已经安装了实时内核。可以从 SUSE Customer Center (https://scc.suse.com/) 下载实时内核映像。

注意

有关实时内核的详细信息，请访问 SUSE Real Time (https://www.suse.com/products/realtime/)。

41.2 实现低延迟和高性能需指定的内核参数

内核参数必须进行配置，以便实时内核能够正常工作，从而为运行电信工作负载提供最佳性能和低延迟。在为此使用场景配置内核参数时，需要特别注意一些重要的概念：

- 使用 SUSE 实时内核时需去除 `kthread_cpus`。此参数控制在哪些 CPU 上创建内核线程。它还控制允许哪些 CPU 用于 PID 1 及加载内核模块（由 kmod 用户空间辅助工具加载）。此参数无法被 SUSE 实时内核识别，因此没有任何效果。
- 通过 `isolcpus`、`nohz_full`、`rcu_nocbs` 和 `irqaffinity` 隔离 CPU 核心。有关 CPU 绑定技术的完整列表，请参见第 41.3 节 “通过 Tuned 和内核参数实现 CPU 绑定”。
- 将 `domain,nohz,managed_irq` 标志添加到 `isolcpus` 内核参数。如果没有任何标志，`isolcpus` 相当于只指定 `domain` 标志。这将导致指定的 CPU 无法进行调度，包括内核任务。`nohz` 标志会停止指定 CPU 上运行的调度器节拍（如果一个 CPU 上只有一个任务可运行），`managed_irq` 标志可避免在指定 CPU 上路由受管理的外部（设备）中断。需要注意的是，NVMe 设备的中断请求 (IRQ) 线路由内核全权管理，因此会被路由至非隔离（系统管理）核心。例如，本节末尾提供的命令行会导致系统中仅分配 4 个队列（外加 1 个管理/控制队列）：

  ```
  for I in $(grep nvme0 /proc/interrupts | cut -d ':' -f1); do cat /proc/irq/
  ${I}/effective_affinity_list; done | column
  39      0       19      20      39
  ```

  此行为可防止磁盘 I/O 对运行在隔离核心上的任何时间敏感型应用程序造成干扰，但可能需要对以存储为重点的工作负载进行关注和精心设计。
- 调整节拍（内核的周期性定时器中断）：

    - skew_tick=1：有时可能同时发生多个节拍。将 skew_tick 设置为 1 可使所有 CPU 不在同一时刻接收定时器节拍，而是在略微错开的时间点接收。这有助于减少系统抖动，从而实现更一致且更低的中断响应时间（这是对延迟敏感型应用程序的基本要求）。
    - nohz=on：停止空闲 CPU 上的周期性定时器节拍。
    - nohz_full=<cpu-cores>：停止专用于实时应用程序的指定 CPU 上的周期性定时器节拍。
- 通过指定 mce=off 禁用计算机检查异常 (MCE) 处理。MCE 是处理器检测到的硬件错误，禁用它们可以避免产生冗余日志。
- 添加 nowatchdog 以禁用软锁死看门狗，该看门狗通过运行在定时器硬中断上下文中的定时器实现。当定时器到期时（即检测到软锁死），它会（在硬中断上下文中）打印警告，违背所有延迟目标。即便从未到期，它也会加入定时器列表，略微增加每次定时器中断的开销。此选项还会禁用 NMI 看门狗，因此 NMI 不会造成干扰。
- nmi_watchdog=0 会禁用 NMI（不可屏蔽中断）看门狗。当使用 nowatchdog 时，可以省略此设置。
- RCU（读取 - 复制 - 更新）是一种内核机制，允许多个读取器以无锁方式并发访问共享数据。RCU 回调是在“宽限期”后触发的函数，确保所有先前的读取器都已完成操作，从而可以安全地回收旧数据。我们对 RCU 进行微调，特别是针对敏感工作负载，将这些回调从专用（绑定的）CPU 卸载，防止内核操作干扰关键的时间敏感型任务。
    - 在 rcu_nocbs 中指定绑定的 CPU，使 RCU 回调不会在这些 CPU 上运行。这有助于减轻实时工作负载的抖动和延迟。
    - rcu_nocb_poll 会使无回调 CPU 定期“轮询”，以确定是否需要处理回调。这可以减少中断开销。
    - rcupdate.rcu_cpu_stall_suppress=1 会抑制 RCU CPU 停滞警告，在高负载实时系统中，这些警告有时可能是误报。
    - rcupdate.rcu_expedited=1 会加速 RCU 操作的宽限期，使读取端关键部分的响应更加迅速。

    - `rcupdate.rcu_normal_after_boot=1` 与 rcu_expedited 一起使用时，允许 RCU 在系统引导后恢复到正常（非加速）操作模式。
    - `rcupdate.rcu_task_stall_timeout=0` 会禁用 RCU 任务停滞检测器，防止长时间运行的 RCU 任务可能引发的警告或系统暂停。
    - `rcutree.kthread_prio=99` 会将 RCU 回调内核线程的优先级设置为可能的最高值 (99)，确保在需要时能够及时调度该线程并处理 RCU 回调。
- 为使 Metal3 和 Cluster API 成功置备/取消置备群集，请添加 `ignition.platform.id=openstack`。该配置供 Metal3 Python 代理使用，该代理源自 OpenStack Ironic。
- 去除 `intel_pstate=passive`。此选项将 `intel_pstate` 配置为与通用 cpufreq 调节器搭配工作，但缺点是，为了实现这一点，它会禁用硬件管理的 P 状态 (HWP)。为了减少硬件延迟，不建议将此选项用于实时工作负载。
- 将 `intel_idle.max_cstate=0 processor.max_cstate=1` 替换为 `idle=poll`。为了避免 C 状态转换，使用 `idle=poll` 选项来禁用 C 状态转换并将 CPU 保持在最高 C 状态。`intel_idle.max_cstate=0` 选项会禁用 `intel_idle`，因此使用 `acpi_idle`，然后 `acpi_idle.max_cstate=1` 会为 acpi_idl 设置最大 C 状态。在 AMD64/Intel 64 体系结构上，第一个 ACPI C 状态始终是轮询，但它使用 `poll_idle()` 函数，这可能会因为定期读取时钟并在超时后重启 `do_idle()` 中的主循环（也涉及清除和设置 `TIF_POLL` 任务标志）而导致一些细微的延迟。相比之下，`idle=poll` 以紧凑循环运行，通过忙等待方式等待任务被重新调度。这最大限度地减少了由退出空闲状态造成的延迟，但代价是 CPU 在空闲线程中也要保持全速运行。
- 在 BIOS 中禁用 C1E。此选项对于禁用 BIOS 中的 C1E 状态非常重要，可以避免 CPU 在空闲时进入 C1E 状态。C1E 状态是一种低功耗状态，可能会在 CPU 空闲时造成延迟。

本文档的其余部分介绍了其他参数，包括大页和 IOMMU。

以下是一个包含上述调整的 32 核 Intel 服务器的内核参数示例：

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt root=UUID=77b713de-5cc7-4d4c-8fc6-
f5eca0a43cf9 skew_tick=1 rd.timeout=60 rd.retry=45 console=ttyS1,115200
```

```
 console=tty0 default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M
 hugepages=0 ignition.platform.id=openstack intel_iommu=on iommu=pt
 irqaffinity=0,31,32,63 isolcpus=domain,nohz,managed_irq,1-30,33-62
 nohz_full=1-30,33-62 nohz=on mce=off net.ifnames=0 nosoftlockup
 nowatchdog nmi_watchdog=0 quiet rcu_nocb_poll rcu_nocbs=1-30,33-62
 rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
 rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
 rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll
```

以下是一个 64 核 AMD 服务器的配置示例。在 128 个逻辑处理器 (0-127) 中，前 8 个核心 (0-7) 用于系统管理，而其余 120 个核心 (8-127) 专用于运行应用程序：

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=575291cf-74e8-42cf-8f2c-408a20dc00b8 skew_tick=1 console=ttyS1,115200
 console=tty0 default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M
 hugepages=0 ignition.platform.id=openstack amd_iommu=on iommu=pt
 irqaffinity=0-7 isolcpus=domain,nohz,managed_irq,8-127 nohz_full=8-127
 rcu_nocbs=8-127 mce=off nohz=on net.ifnames=0 nowatchdog nmi_watchdog=0
 nosoftlockup quiet rcu_nocb_poll rcupdate.rcu_cpu_stall_suppress=1
 rcupdate.rcu_expedited=1 rcupdate.rcu_normal_after_boot=1
 rcupdate.rcu_task_stall_timeout=0 rcutree.kthread_prio=99 security=selinux
 selinux=1 idle=poll
```

41.3 通过 Tuned 和内核参数实现 CPU 绑定

Tuned 是一个系统调优工具，它通过各种预定义配置文件监控系统状况以优化性能。其关键特性之一是能够为特定工作负载（如实时应用程序）隔离 CPU 核心。这可防止操作系统使用这些核心，从而避免潜在的延迟增加。

要启用和配置此功能，首先应该为我们想要隔离的 CPU 核心创建一个配置文件。在本示例中，64 个核心中，我们将 60 个核心（1-30 和 33-62）专用于应用程序，剩余 4 个核心用于系统管理。需要注意的是，隔离 CPU 的设计在很大程度上取决于实时应用程序的需求。

```
$ echo "export tuned_params" >> /etc/grub.d/00_tuned

$ echo "isolated_cores=1-30,33-62" >> /etc/tuned/cpu-partitioning-variables.conf
```

```
$ tuned-adm profile cpu-partitioning
Tuned (re)started, changes applied.
```

然后我们需要修改用于隔离 CPU 核心的 GRUB 选项以及其他重要的 CPU 使用参数。请务必根据您的当前硬件规格自定义以下选项：

参数	值	说明
isolcpus	domain,nohz, managed_irq,1-30,33-62	隔离核心 1-30 和 33-62。`domain` 表示这些 CPU 属于隔离域。`nohz` 在这些隔离的 CPU 空闲时启用无节拍操作，以减少中断。`managed_irq` 使绑定的 CPU 免受 IRQ 干扰。这考虑了 `irqaffinity=0-7`，它已将大多数 IRQ 定向到系统管理核心。
skew_tick	1	此选项允许内核在隔离的 CPU 之间错开定时器中断。
nohz	on	启用后，内核的周期性定时器中断（即“节拍”）将在任何空闲的 CPU 核心上停止。这主要有益于系统管理 CPU（0、31、32、63）。这可节省电力并减少这些通用核心上不必要的唤醒。
nohz_full	1-30,33-62	对于隔离的核心，这会停止节拍，即使 CPU 运行单个活动任务，也不例外。这意味着它使 CPU 以完全无节拍模式

参数	值	说明
		（或“动态节拍”）运行。内核仅在实际需要时才会传递定时器中断。
rcu_nocbs	1-30,33-62	此选项会将 RCU 回调处理从指定的 CPU 核心卸载。
rcu_nocb_poll		如果设置此选项，无 RCU 回调的 CPU 将定期“轮询”以确定是否需要处理回调，而不是由其他 CPU 显式唤醒。这可以减少中断开销。
irqaffinity	0,31,32,63	此选项允许内核在系统管理核心上运行中断。
idle	poll	最大限度减少因退出空闲状态造成的延迟，但代价是 CPU 在空闲线程中也要保持全速运行。
nmi_watchdog	0	此选项仅禁用 NMI 看门狗。如果设置了 `nowatchdog`，则可以省略此选项。
nowatchdog		此选项会禁用软锁死看门狗，该看门狗通过运行在定时器硬中断上下文中的定时器实现。

以下命令会修改 GRUB 配置并应用上述更改，以便下次引导时使用更改的配置：

编辑 `/etc/default/grub` 文件，在其中添加上述参数，如下所示：

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1
```

```
rd.timeout=60 rd.retry=45 console=ttyS1,115200 console=tty0
default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=0
ignition.platform.id=openstack intel_iommu=on iommu=pt irqaffinity=0,31,32,63
isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
mce=off net.ifnames=0 nosoftlockup nowatchdog nmi_watchdog=0 quiet
rcu_nocb_poll rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1
rcupdate.rcu_expedited=1 rcupdate.rcu_normal_after_boot=1
rcupdate.rcu_task_stall_timeout=0 rcutree.kthread_prio=99 security=selinux
selinux=1 idle=poll"
```

更新 GRUB 配置：

```
$ transactional-update grub.cfg
$ reboot
```

要验证重引导后是否应用了这些参数，可使用以下命令检查内核命令行：

```
$ cat /proc/cmdline
```

另外还有一个脚本可用于调整 CPU 配置，它主要执行以下步骤：

- 将 CPU 调节器设置为 `performance`。
- 取消将定时器迁移到隔离 CPU 的设置。
- 将 kdaemon 线程迁移到系统管理 CPU。
- 将隔离的 CPU 延迟设置为尽可能低的值。
- 将 vmstat 更新延迟到 300 秒。

该脚本可在 SUSE Edge for Telco 示例储存库 (https://raw.githubusercontent.com/suse-edge/atip/refs/heads/release-3.3/telco-examples/edge-clusters/dhcp-less/eib/custom/files/performance-settings.sh) 中获得。

41.4 CNI 配置

41.4.1 Cilium

Cilium 是 SUSE Edge for Telco 的默认 CNI 插件。要在 RKE2 群集上启用 Cilium 作为默认插件，需要在 /etc/rancher/rke2/config.yaml 文件中进行以下配置：

```
cni:
- cilium
```

也可以使用命令行参数来指定此配置，即，将 --cni=cilium 添加到 /etc/systemd/system/rke2-server 文件的 server 行中。

要使用下一节（第 41.5 节 “SR-IOV” [462]）中所述的 SR-IOV 网络操作器，请将 Multus 与另一个 CNI 插件（例如 Cilium 或 Calico）一起用作辅助插件。

```
cni:
- multus
- cilium
```

注意

有关 CNI 插件的详细信息，请访问 Network Options (https://docs.rke2.io/install/network_options)。

41.5 SR-IOV

SR-IOV 允许网络适配器等设备在各种 PCIe 硬件功能之间分隔对其资源的访问。可以通过多种方式部署 SR-IOV，本节将介绍两种不同的方式：

- 方式 1：使用 SR-IOV CNI 设备插件和配置映射来正确配置 SR-IOV。
- 方式 2（建议）：使用 Rancher Prime 中的 SR-IOV Helm chart 来轻松完成此部署。

方式 1 - 安装 SR-IOV CNI 设备插件并准备配置映射来正确配置 SR-IOV

- 为设备插件准备配置映射

使用 `lspci` 命令获取用于填充配置映射的信息：

```
$ lspci | grep -i acc
8a:00.0 Processing accelerators: Intel Corporation Device 0d5c

$ lspci | grep -i net
19:00.0 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.1 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.2 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.3 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
51:00.0 Ethernet controller: Intel Corporation Ethernet Controller E810-C for
 QSFP (rev 02)
51:00.1 Ethernet controller: Intel Corporation Ethernet Controller E810-C for
 QSFP (rev 02)
51:01.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:01.1 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:01.2 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:01.3 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:11.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:11.1 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:11.2 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
51:11.3 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
 Function (rev 02)
```

配置映射由一个 `JSON` 文件组成，该文件描述通过过滤器发现的设备，以及创建接口组所需的信息。此处的关键在于理解过滤器和组。过滤器用于发现设备，组用于创建接口。

可以设置过滤器：

- vendorID：`8086` (Intel)
- deviceID：`0d5c`（加速卡）
- driver：`vfio-pci`（驱动程序）
- pfNames：`p2p1`（物理接口名称）

还可以设置过滤器来匹配更复杂的接口语法，例如：

- pfNames：`["eth1#1,2,3,4,5,6"]` 或 `[eth1#1-6]`（物理接口名称）

至于组，我们可为 `FEC` 卡创建一个组，为 `Intel` 卡创建另一个组，甚至可以根据具体使用场景创建一个前缀：

- resourceName：`pci_sriov_net_bh_dpdk`
- resourcePrefix：`Rancher.io`

有很多组合方式可用于发现并创建资源组，以将一些 `VF` 分配给 Pod。

注意

有关过滤器和组的详细信息，请访问 sr-iov 网络设备插件 (https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin)。

根据硬件和使用场景设置用于匹配接口的过滤器和组后，下面的配置映射展示了一个可使用的示例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
    {
        "resourceList": [
```

```
            {
                "resourceName": "intel_fec_5g",
                "devicetype": "accelerator",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["0d5d"]
                }
            },
            {
                "resourceName": "intel_sriov_odu",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["1889"],
                    "drivers": ["vfio-pci"],
                    "pfNames": ["p2p1"]
                }
            },
            {
                "resourceName": "intel_sriov_oru",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["1889"],
                    "drivers": ["vfio-pci"],
                    "pfNames": ["p2p2"]
                }
            }
        ]
    }
```

- 准备 daemonset 文件以部署设备插件。

设备插件支持多种体系结构（arm、amd、ppc64le），因此同一文件可用于不同的体系结构，以便为每个体系结构部署多个 daemonset。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sriov-device-plugin
  namespace: kube-system
```

```
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-sriov-device-plugin-amd64
  namespace: kube-system
  labels:
    tier: node
    app: sriovdp
spec:
  selector:
    matchLabels:
      name: sriov-device-plugin
  template:
    metadata:
      labels:
        name: sriov-device-plugin
        tier: node
        app: sriovdp
    spec:
      hostNetwork: true
      nodeSelector:
        kubernetes.io/arch: amd64
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      serviceAccountName: sriov-device-plugin
      containers:
      - name: kube-sriovdp
        image: rancher/hardened-sriov-network-device-plugin:v3.7.0-build20240816
        imagePullPolicy: IfNotPresent
        args:
        - --log-dir=sriovdp
        - --log-level=10
        securityContext:
          privileged: true
        resources:
```

```
        requests:
          cpu: "250m"
          memory: "40Mi"
        limits:
          cpu: 1
          memory: "200Mi"
      volumeMounts:
      - name: devicesock
        mountPath: /var/lib/kubelet/
        readOnly: false
      - name: log
        mountPath: /var/log
      - name: config-volume
        mountPath: /etc/pcidp
      - name: device-info
        mountPath: /var/run/k8s.cni.cncf.io/devinfo/dp
    volumes:
      - name: devicesock
        hostPath:
          path: /var/lib/kubelet/
      - name: log
        hostPath:
          path: /var/log
      - name: device-info
        hostPath:
          path: /var/run/k8s.cni.cncf.io/devinfo/dp
          type: DirectoryOrCreate
      - name: config-volume
        configMap:
          name: sriovdp-config
          items:
          - key: config.json
            path: config.json
```

- 应用配置映射和 daemonset 后，将部署设备插件，发现接口并将其提供给 Pod 使用。

```
$ kubectl get pods -n kube-system | grep sriov
```

```
kube-system  kube-sriov-device-plugin-amd64-twjfl  1/1  Running  0  2m
```

- 检查节点中发现的且可供 Pod 使用的接口：

```
$ kubectl get $(kubectl get nodes -oname) -o
 jsonpath='{.status.allocatable}' | jq
{
  "cpu": "64",
  "ephemeral-storage": "256196109726",
  "hugepages-1Gi": "40Gi",
  "hugepages-2Mi": "0",
  "intel.com/intel_fec_5g": "1",
  "intel.com/intel_sriov_odu": "4",
  "intel.com/intel_sriov_oru": "4",
  "memory": "221396384Ki",
  "pods": "110"
}
```

- FEC 为 `intel.com/intel_fec_5g`，值为 1。
- 如果您部署 SR-IOV 时使用了设备插件和配置映射，但未使用 Helm chart，则 VF 为 `intel.com/intel_sriov_odu` 或 `intel.com/intel_sriov_oru`。

重要

如果此处没有接口，则继续操作就意义不大，因为 Pod 没有可用的接口。请先检查配置映射和过滤器来解决问题。

方式 2（建议）- 使用 Rancher 和 Helm chart 安装 SR-IOV CNI 和设备插件

- 获取 Helm（如果没有）：

```
$ curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
 bash
```

- 安装 SR-IOV。

```
helm install sriov-crd oci://registry.suse.com/edge/charts/sriov-crd -n sriov-
network-operator
```

```
helm install sriov-network-operator oci://registry.suse.com/edge/charts/sriov-
network-operator -n sriov-network-operator
```

- 检查已部署的资源 CRD 和 Pod：

```
$ kubectl get crd
$ kubectl -n sriov-network-operator get pods
```

- 检查节点中的标签。

所有资源都运行后，标签会自动出现在您的节点中：

```
$ kubectl get nodes -oyaml | grep feature.node.kubernetes.io/network-
sriov.capable

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

- 查看 `daemonset`，您会看到新的 `sriov-network-config-daemon` 和 `sriov-rancher-nfd-worker` 处于活动状态并已准备就绪：

```
$ kubectl get daemonset -A
NAMESPACE             NAME                              DESIRED   CURRENT   READY
  UP-TO-DATE   AVAILABLE   NODE SELECTOR
   AGE
calico-system            calico-node                       1         1         1
     1            1           kubernetes.io/os=linux
      15h
sriov-network-operator   sriov-network-config-daemon     1         1
     1       1            1           feature.node.kubernetes.io/network-
sriov.capable=true   45m
sriov-network-operator   sriov-rancher-nfd-worker        1         1         1
     1            1           <none>
      45m
kube-system              rke2-ingress-nginx-controller   1         1         1
     1            1           kubernetes.io/os=linux
      15h
kube-system              rke2-multus-ds                  1         1         1
     1            1           kubernetes.io/arch=amd64,kubernetes.io/os=linux
      15h
```

几分钟后（最多可能需要 10 分钟才会更新），将检测到节点，其上已配置了 SR-IOV 功能：

```
$ kubectl get sriovnetworknodestates.sriovnetwork.openshift.io -A
NAMESPACE              NAME     AGE
sriov-network-operator   xr11-2   83s
```

- 检查已检测到的接口。

发现的接口应为网络设备的 PCI 地址。请在主机中使用 `lspci` 命令检查此信息。

```
$ kubectl get sriovnetworknodestates.sriovnetwork.openshift.io -n kube-system -
oyaml
apiVersion: v1
items:
- apiVersion: sriovnetwork.openshift.io/v1
  kind: SriovNetworkNodeState
  metadata:
    creationTimestamp: "2023-06-07T09:52:37Z"
    generation: 1
    name: xr11-2
    namespace: sriov-network-operator
    ownerReferences:
    - apiVersion: sriovnetwork.openshift.io/v1
      blockOwnerDeletion: true
      controller: true
      kind: SriovNetworkNodePolicy
      name: default
      uid: 80b72499-e26b-4072-a75c-f9a6218ec357
    resourceVersion: "356603"
    uid: e1f1654b-92b3-44d9-9f87-2571792cc1ad
  spec:
    dpConfigVersion: "356507"
  status:
    interfaces:
    - deviceID: "1592"
      driver: ice
      eSwitchMode: legacy
      linkType: ETH
      mac: 40:a6:b7:9b:35:f0
```

```
      mtu: 1500
      name: p2p1
      pciAddress: "0000:51:00.0"
      totalvfs: 128
      vendor: "8086"
    - deviceID: "1592"
      driver: ice
      eSwitchMode: legacy
      linkType: ETH
      mac: 40:a6:b7:9b:35:f1
      mtu: 1500
      name: p2p2
      pciAddress: "0000:51:00.1"
      totalvfs: 128
      vendor: "8086"
    syncStatus: Succeeded
kind: List
metadata:
  resourceVersion: ""
```

注意

如果此处未检测到您的接口，请确保该接口存在于下一个配置映射中：

```
$ kubectl get cm supported-nic-ids -oyaml -n sriov-network-operator
```

如果此处未检测到您的设备，请编辑配置映射，添加要发现的正确值（如有必要，请重启 `sriov-network-config-daemon` daemonset）。

- 创建 `NetworkNode 策略`来配置 `VF`。

将在设备 (`rootDevices`) 中创建一些 `VF` (`numVfs`)，并在其上配置驱动程序 `deviceType` 和 `MTU`：

注意

`resourceName` 字段不得包含任何特殊字符，并且在整个群集中必须是唯一的。该示例使用 `deviceType: vfio-pci`，因为 `dpdk` 将与 `sr-iov` 结合使用。如果您不使用 `dpdk`，则 deviceType 应为 `deviceType: netdevice`（默认值）。

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-dpdk
  namespace: sriov-network-operator
spec:
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  resourceName: intelnicsDpdk
  deviceType: vfio-pci
  numVfs: 8
  mtu: 1500
  nicSelector:
    deviceID: "1592"
    vendor: "8086"
    rootDevices:
    - 0000:51:00.0
```

- 验证配置：

```
$ kubectl get $(kubectl get nodes -oname) -o jsonpath='{.status.allocatable}' |
 jq
{
  "cpu": "64",
  "ephemeral-storage": "256196109726",
  "hugepages-1Gi": "60Gi",
  "hugepages-2Mi": "0",
  "intel.com/intel_fec_5g": "1",
  "memory": "200424836Ki",
  "pods": "110",
  "rancher.io/intelnicsDpdk": "8"
```

```
}
```

- 创建 sr-iov 网络（可选操作，仅在需要不同的网络时才执行）：

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: network-dpdk
  namespace: sriov-network-operator
spec:
  ipam: |
    {
      "type": "host-local",
      "subnet": "192.168.0.0/24",
      "rangeStart": "192.168.0.20",
      "rangeEnd": "192.168.0.60",
      "routes": [{
        "dst": "0.0.0.0/0"
      }],
      "gateway": "192.168.0.1"
    }
  vlan: 500
  resourceName: intelnicsDpdk
```

- 检查创建的网络：

```
$ kubectl get network-attachment-definitions.k8s.cni.cncf.io -A -oyaml

apiVersion: v1
items:
- apiVersion: k8s.cni.cncf.io/v1
  kind: NetworkAttachmentDefinition
  metadata:
    annotations:
      k8s.v1.cni.cncf.io/resourceName: rancher.io/intelnicsDpdk
    creationTimestamp: "2023-06-08T11:22:27Z"
    generation: 1
    name: network-dpdk
    namespace: sriov-network-operator
```

```
    resourceVersion: "13124"
    uid: df7c89f5-177c-4f30-ae72-7aef3294fb15
  spec:
    config: '{ "cniVersion":"0.4.0", "name":"network-
dpdk","type":"sriov","vlan":500,"vlanQoS":0,"ipam":{"type":"host-
local","subnet":"192.168.0.0/24","rangeStart":"192.168.0.10","rangeEnd":"192.168.0.60","rout
[{"dst":"0.0.0.0/0"}],"gateway":"192.168.0.1"}
      }'
kind: List
metadata:
  resourceVersion: ""
```

41.6 DPDK

DPDK（数据平面开发包）是一组用于实现快速数据包处理的库和驱动程序。它用于加速各种 CPU 体系结构上运行的数据包处理工作负载。DPDK 包含以下组件的数据平面库和优化的网络接口控制器 (NIC) 驱动程序：

1. 队列管理器，实现无锁队列。
2. 缓冲区管理器，预先分配固定大小的缓冲区。
3. 内存管理器，在内存中分配对象池，并使用环来存储空闲对象；确保对象均匀分布在所有 DRAM 通道上。
4. 轮询模式驱动程序 (PMD)，可在没有异步通知的情况下运行，从而降低开销。
5. 作为一组库提供的数据包框架，可帮助开发数据包处理解决方案。

以下步骤说明如何启用 DPDK，以及如何从 NIC 创建供 DPDK 接口使用的 VF：

- 安装 DPDK 软件包：

```
$ transactional-update pkg install dpdk dpdk-tools libdpdk-23
$ reboot
```

- 内核参数：

要使用 DPDK，请采用一些驱动程序来启用内核中的某些参数：

参数	值	说明
iommu	pt	此选项允许为 DPDK 接口使用 vfio 驱动程序。
intel_iommu 或 amd_iommu	on	此选项允许为 VF 使用 vfio。

要启用这些参数，请将其添加到 /etc/default/grub 文件中：

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1
 rd.timeout=60 rd.retry=45 console=ttyS1,115200 console=tty0
 default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=0
 ignition.platform.id=openstack intel_iommu=on iommu=pt irqaffinity=0,31,32,63
 isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
 mce=off net.ifnames=0 nosoftlockup nowatchdog nmi_watchdog=0 quiet
 rcu_nocb_poll rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1
 rcupdate.rcu_expedited=1 rcupdate.rcu_normal_after_boot=1
 rcupdate.rcu_task_stall_timeout=0 rcutree.kthread_prio=99 security=selinux
 selinux=1 idle=poll"
```

更新 GRUB 配置，并重引导系统以应用更改：

```
$ transactional-update grub.cfg
$ reboot
```

- 加载 vfio-pci 内核模块，并在 NIC 上启用 SR-IOV：

```
$ modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
```

- 从 NIC 创建一些虚拟功能 (VF)。

例如，要为两个不同的 NIC 创建 VF，需要运行以下命令：

```
$ echo 4 > /sys/bus/pci/devices/0000:51:00.0/sriov_numvfs
$ echo 4 > /sys/bus/pci/devices/0000:51:00.1/sriov_numvfs
```

- 将新的 VF 绑定到 `vfio-pci` 驱动程序：

```
$ dpdk-devbind.py -b vfio-pci 0000:51:01.0 0000:51:01.1 0000:51:01.2
 0000:51:01.3 \
                              0000:51:11.0 0000:51:11.1 0000:51:11.2
 0000:51:11.3
```

- 检查是否正确应用了配置：

```
$ dpdk-devbind.py -s

Network devices using DPDK-compatible driver
============================================
0000:51:01.0 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:01.1 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:01.2 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:01.3 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:01.0 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:11.1 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:21.2 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio
0000:51:31.3 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci
 unused=iavf,igb_uio

Network devices using kernel driver
===================================
0000:19:00.0 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet
 1751' if=em1 drv=bnxt_en unused=igb_uio,vfio-pci *Active*
0000:19:00.1 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet
 1751' if=em2 drv=bnxt_en unused=igb_uio,vfio-pci
0000:19:00.2 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet
 1751' if=em3 drv=bnxt_en unused=igb_uio,vfio-pci
```

```
0000:19:00.3 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet
 1751' if=em4 drv=bnxt_en unused=igb_uio,vfio-pci
0000:51:00.0 'Ethernet Controller E810-C for QSFP 1592' if=eth13 drv=ice
 unused=igb_uio,vfio-pci
0000:51:00.1 'Ethernet Controller E810-C for QSFP 1592' if=rename8 drv=ice
 unused=igb_uio,vfio-pci
```

41.7 vRAN 加速 (Intel ACC100/ACC200)

随着通讯服务提供商从 4G 过渡到 5G 网络，有许多提供商正在采用虚拟化无线接入网络 (vRAN) 体系结构来提高信道容量及简化基于边缘的服务和应用程序的部署。vRAN 解决方案非常适合用于提供低延迟服务，并可以根据实时流量和网络需求灵活地增加或减少容量。

计算密集程度最高的 4G 和 5G 工作负载之一是 RAN 第 1 层 (L1) FEC，它可以解决不可靠或高干扰信道上的数据传输错误。FEC 技术可以检测并纠正 4G 或 5G 数据中有限数量的错误，因此消除了重新传输的需要。由于 FEC 加速事务不包含特定的单元状态信息，因此可以轻松虚拟化，从而带来了池化优势并可实现轻松的单元迁移。

- 内核参数

要启用 vRAN 加速，需要启用以下内核参数（如果尚未启用）：

参数	值	说明
iommu	pt	此选项允许为 DPDK 接口使用 vfio。
intel_iommu 或 amd_iommu	on	此选项允许为 VF 使用 vfio。

修改 GRUB 文件 `/etc/default/grub`，以将这些参数添加到内核命令行：

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1
 rd.timeout=60 rd.retry=45 console=ttyS1,115200 console=tty0
 default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=0
 ignition.platform.id=openstack intel_iommu=on iommu=pt irqaffinity=0,31,32,63
```

```
isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
mce=off net.ifnames=0 nosoftlockup nowatchdog nmi_watchdog=0 quiet
rcu_nocb_poll rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1
rcupdate.rcu_expedited=1 rcupdate.rcu_normal_after_boot=1
rcupdate.rcu_task_stall_timeout=0 rcutree.kthread_prio=99 security=selinux
selinux=1 idle=poll"
```

更新 GRUB 配置，并重引导系统以应用更改：

```
$ transactional-update grub.cfg
$ reboot
```

要校验重引导后是否应用了这些参数，请检查命令行：

```
$ cat /proc/cmdline
```

- 加载 vfio-pci 内核模块以启用 vRAN 加速：

```
$ modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
```

- 获取 Acc100 接口信息：

```
$ lspci | grep -i acc
8a:00.0 Processing accelerators: Intel Corporation Device 0d5c
```

- 将物理接口 (PF) 绑定到 `vfio-pci` 驱动程序：

```
$ dpdk-devbind.py -b vfio-pci 0000:8a:00.0
```

- 从物理接口 (PF) 创建虚拟功能 (VF)。

从 PF 创建 2 个 VF，并按照以下步骤将其绑定到 `vfio-pci`：

```
$ echo 2 > /sys/bus/pci/devices/0000:8a:00.0/sriov_numvfs
$ dpdk-devbind.py -b vfio-pci 0000:8b:00.0
```

- 使用建议的配置文件配置 acc100：

```
$ pf_bb_config ACC100 -c /opt/pf-bb-config/acc100_config_vf_5g.cfg
```

```
Tue Jun  6 10:49:20 2023:INFO:Queue Groups: 2 5GUL, 2 5GDL, 2 4GUL, 2 4GDL
Tue Jun  6 10:49:20 2023:INFO:Configuration in VF mode
Tue Jun  6 10:49:21 2023:INFO: ROM version MM 99AD92
Tue Jun  6 10:49:21 2023:WARN:* Note: Not on DDR PRQ version  1302020 !=
 10092020
Tue Jun  6 10:49:21 2023:INFO:PF ACC100 configuration complete
Tue Jun  6 10:49:21 2023:INFO:ACC100 PF [0000:8a:00.0] configuration complete!
```

- 检查从 FEC PF 创建的新 VF：

```
$ dpdk-devbind.py -s
Baseband devices using DPDK-compatible driver
==============================================
0000:8a:00.0 'Device 0d5c' drv=vfio-pci unused=
0000:8b:00.0 'Device 0d5d' drv=vfio-pci unused=

Other Baseband devices
======================
0000:8b:00.1 'Device 0d5d' unused=
```

41.8 大页

当某个进程使用 RAM 时，CPU 会将 RAM 标记为已由该进程使用。为提高效率，CPU 会以区块的形式分配 RAM，在许多平台上，默认的区块大小值为 4K 字节。这些区块称为页。页可以交换到磁盘等位置。

由于进程地址空间是虚拟的，CPU 和操作系统需要记住哪些页属于哪个进程，以及每个页存储在哪个位置。页数越多，内存映射的搜索时间就越长。当进程使用 1 GB 内存时，需要查找 262144 个项 (1 GB / 4 K)。如果一个页表项占用 8 个字节，则需要查找 2 MB (262144 * 8) 内存。

当前的大多数 CPU 体系结构都支持大于默认值的页，因此减少了 CPU/操作系统需要查找的项。

- 内核参数

要启用大页，我们应添加以下内核参数。在此例中，我们配置了 40 个 1G 页面，不过大页大小和确切数量应根据应用程序的内存需求进行调整：

参数	值	说明
hugepagesz	1G	此选项允许将大页的大小设置为 1 G
hugepages	40	这是先前定义的大页数量
default_hugepagesz	1G	这是用于获取大页的默认值

修改 GRUB 文件 `/etc/default/grub`，在 `GRUB_CMDLINE_LINUX` 中添加这些参数：

```
default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=0
```

更新 GRUB 配置，并重引导系统以应用更改：

```
$ transactional-update grub.cfg
$ reboot
```

要验证重引导后是否应用了这些参数，可以检查命令行：

```
$ cat /proc/cmdline
```

- 使用大页

要使用大页，需要挂载它们：

```
$ mkdir -p /hugepages
$ mount -t hugetlbfs nodev /hugepages
```

部署 Kubernetes 工作负载，并创建资源和卷：

```
...
 resources:
   requests:
     memory: "24Gi"
     hugepages-1Gi: 16Gi
     intel.com/intel_sriov_oru: '4'
   limits:
```

```
     memory: "24Gi"
     hugepages-1Gi: 16Gi
     intel.com/intel_sriov_oru: '4'
...
```

```
...
volumeMounts:
  - name: hugepage
    mountPath: /hugepages
...
volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
...
```

41.9 在 Kubernetes 上进行 CPU 绑定

41.9.1 先决条件

必须根据第 41.3 节 “通过 Tuned 和内核参数实现 CPU 绑定”一节中所述的性能配置文件微调 CPU。

41.9.2 配置 Kubernetes 以进行 CPU 绑定

配置 kubelet 参数以在 RKE2 群集中实现 CPU 管理。将以下配置块（如下例所示）添加到 `/etc/rancher/rke2/config.yaml` 文件中。确保在 `kubelet-reserved` 和 `system-reserved` 参数中指定系统管理 CPU 核心：

```
kubelet-arg:
- "cpu-manager-policy=static"
- "cpu-manager-policy-options=full-pcpus-only=true"
- "cpu-manager-reconcile-period=0s"
- "kubelet-reserved=cpu=0,31,32,63"
```

```
- "system-reserved=cpu=0,31,32,63"
```

41.9.3 为工作负载使用绑定的 CPU

根据您在工作负载上定义的请求和限制，可以用三种方式通过 kubelet 中定义的静态策略来使用该功能：

1. `BestEffort` QoS 类：如果没有定义任何 `CPU` 请求或限制，则 Pod 将调度到系统中第一个可用的 `CPU` 上。
   使用 `BestEffort` QoS 类的示例如下：

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
   ```

2. `Burstable` QoS 类：如果定义了 CPU 请求且该请求不等于限制值，或者没有定义 CPU 请求，请使用此方式。
   使用 `Burstable` QoS 类的示例如下：

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
       resources:
         limits:
           memory: "200Mi"
         requests:
           memory: "100Mi"
   ```

   或

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
       resources:
   ```

```
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "100Mi"
        cpu: "1"
```

3. Guaranteed QoS 类：如果定义了 CPU 请求且该请求等于限制值，请使用此方式。
使用 Guaranteed QoS 类的示例如下：

```
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "2"
        requests:
          memory: "200Mi"
          cpu: "2"
```

41.10 可感知 NUMA 的调度

非统一内存访问或非统一内存体系结构 (NUMA) 是 SMP（多处理器）体系结构中使用的物理内存设计，其中内存访问时间取决于相对于处理器的内存位置。在 NUMA 下，与访问非本地内存（即，另一个处理器本地的内存，或者在处理器之间共享的内存）相比，处理器可以更快地访问自身的本地内存。

41.10.1 识别 NUMA 节点

要识别 NUMA 节点，请在系统上使用以下命令：

```
$ lscpu | grep NUMA
NUMA node(s):                    1
```

```
NUMA node0 CPU(s):                   0-63
```

注意

对于此示例，只有一个 NUMA 节点显示了 64 个 CPU。

NUMA 需要在 BIOS 中启用。如果 dmesg 中没有引导过程中 NUMA 初始化的记录，则可能表示内核环缓冲区中的 NUMA 相关消息已被重写。

41.11 MetalLB

MetalLB 是裸机 Kubernetes 群集的负载平衡器实现，使用 L2 和 BGP 等标准路由协议作为广告协议。它是一个网络负载平衡器，可用于向外部公开 Kubernetes 群集中的服务（因为需要在裸机上使用 Kubernetes 服务类型 LoadBalancer）。

要在 RKE2 群集中启用 MetalLB，需要执行以下步骤：

- 使用以下命令安装 MetalLB：

```
$ kubectl apply <<EOF -f
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: metallb
  namespace: kube-system
spec:
  chart: oci://registry.suse.com/edge/charts/metallb
  targetNamespace: metallb-system
  version: 303.0.0+up0.14.9
  createNamespace: true
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: endpoint-copier-operator
  namespace: kube-system
spec:
```

```
  chart: oci://registry.suse.com/edge/charts/endpoint-copier-operator
  targetNamespace: endpoint-copier-operator
  version: 303.0.0+up0.2.1
  createNamespace: true
EOF
```

- 创建 `IpAddressPool` 和 `L2advertisement` 配置：

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: kubernetes-vip-ip-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.168.200.98/32
  serviceAllocation:
    priority: 100
    namespaces:
      - default
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
    - kubernetes-vip-ip-pool
```

- 创建端点服务来公开 `VIP`：

```
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  internalTrafficPolicy: Cluster
```

```
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: rke2-api
    port: 9345
    protocol: TCP
    targetPort: 9345
  - name: k8s-api
    port: 6443
    protocol: TCP
    targetPort: 6443
  sessionAffinity: None
  type: LoadBalancer
```

- 检查 VIP 是否已创建并且 MetalLB Pod 是否正在运行：

```
$ kubectl get svc -n default
$ kubectl get pods -n default
```

41.12 专用注册表配置

可以配置 Containerd 以连接到专用注册表，然后使用专用注册表提取每个节点上的专用映像。

启动时，RKE2 会检查 /etc/rancher/rke2/ 中是否存在 registries.yaml 文件，并指示 containerd 使用该文件中定义的任何注册表。如果您希望使用某个专用注册表，请在要使用该注册表的每个节点上以 root 身份创建此文件。

要添加专用注册表，请创建包含以下内容的 /etc/rancher/rke2/registries.yaml 文件：

```
mirrors:
  docker.io:
    endpoint:
      - "https://registry.example.com:5000"
configs:
  "registry.example.com:5000":
```

```
    auth:
      username: xxxxxx # this is the registry username
      password: xxxxxx # this is the registry password
    tls:
      cert_file:            # path to the cert file used to authenticate to the
 registry
      key_file:             # path to the key file for the certificate used to
 authenticate to the registry
      ca_file:              # path to the ca file used to verify the registry's
 certificate
      insecure_skip_verify: # may be set to true to skip verifying the
 registry's certificate
```

或者不设置身份验证：

```
mirrors:
  docker.io:
    endpoint:
      - "https://registry.example.com:5000"
configs:
  "registry.example.com:5000":
    tls:
      cert_file:            # path to the cert file used to authenticate to the
 registry
      key_file:             # path to the key file for the certificate used to
 authenticate to the registry
      ca_file:              # path to the ca file used to verify the registry's
 certificate
      insecure_skip_verify: # may be set to true to skip verifying the
 registry's certificate
```

要使注册表更改生效，需要先配置此文件再在节点上启动 RKE2，或者在配置的每个节点上重启 RKE2。

注意

有关详细信息，请查看 RKE2 containerd 注册表配置 (https://documentation.suse.com/cloudnative/rke2/latest/en/install/containerd_registry_configuration.html#_registries_configuration_file)。

41.13 精确时间协议

精确时间协议 (PTP) 是由电气和电子工程师协会 (IEEE) 开发的一种网络协议，旨在实现计算机网络中的亚微秒级时间同步。自诞生以来的几十年里，PTP 已在许多行业广泛应用。近年来，随着其作为 5G 网络关键要素的重要性凸显，在电信网络中的采用率不断提升。尽管 PTP 协议相对简单，但其配置会因应用场景的不同而有显著差异。为此，已定义并标准化多个配置文件。

本节仅介绍专用于电信行业的配置文件。因此，假设 NIC 具备时间戳能力和 PTP 硬件时钟 (PHC)。如今，所有电信级网络适配器均在硬件层面支持 PTP，但您可以通过以下命令验证此类功能：

```
# ethtool -T p1p1
Time stamping parameters for p1p1:
Capabilities:
        hardware-transmit
        software-transmit
        hardware-receive
        software-receive
        software-system-clock
        hardware-raw-clock
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off
        on
Hardware Receive Filter Modes:
        none
        all
```

将 `p1p1` 替换为用于 PTP 的接口名称。

以下几节将具体介绍如何在 SUSE Edge 上安装和配置 PTP，但以您已熟悉 PTP 的基本概念为前提。如需简要了解 PTP 以及 SUSE Edge for Telco 中包含的实现，请访问 https://documentation.suse.com/sles/html/SLES-all/cha-tuning-ptp.html。

41.13.1 安装 PTP 软件组件

在 SUSE Edge for Telco 中，PTP 功能通过 `linuxptp` 软件包实现，该软件包包含两个组件：

- `ptp4l`：控制 NIC 上的 PHC 并运行 PTP 协议的守护程序
- `phc2sys`：将系统时钟与 NIC 上经 PTP 同步的 PHC 保持同步的守护程序

要使系统同步正常工作，就必须运行这两个守护程序，并且必须根据您的设置正确配置它们。相关信息，请参见第 41.13.2 节 “为电信部署配置 PTP”。

在下游群集中集成 PTP 最简单且最佳的方式是，在 Edge Image Builder (EIB) 定义文件的 `packageList` 下添加 `linuxptp` 软件包。这样，在群集置备期间就会自动安装 PTP 控制平面软件。有关安装软件包的详细信息，请参见 EIB 文档（第 3.3.4 节 “配置 RPM 软件包”）。

以下是包含 `linuxptp` 的 EIB 清单示例：

```
apiVersion: 1.0
image:
  imageType: RAW
  arch: x86_64
  baseImage: {micro-base-rt-image-raw}
  outputImageName: eibimage-slmicrort-telco.raw
operatingSystem:
  time:
    timezone: America/New_York
  kernelArgs:
    - ignition.platform.id=openstack
    - net.ifnames=1
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
      - fstrim
      - time-sync.target
    enable:
      - ptp4l
      - phc2sys
  users:
    - username: root
      encryptedPassword: ${ROOT_PASSWORD}
  packages:
    packageList:
```

```
    - jq
    - dpdk
    - dpdk-tools
    - libdpdk-23
    - pf-bb-config
    - open-iscsi
    - tuned
    - cpupower
    - linuxptp
  sccRegistrationCode: ${SCC_REGISTRATION_CODE}
```

注意

SUSE Edge for Telco 中包含的 `linuxptp` 软件包默认不会启用 `ptp4l` 和 `phc2sys`。如果在置备时部署了针对特定系统的配置文件（请参见第 41.13.3 节 “Cluster API 集成”），则应启用它们。可通过将其添加到清单的 `systemd` 部分来实现，如上例所示。

按照 EIB 文档（第 3.4 节 “构建映像”）中所述的常规过程构建映像，并使用该映像部署群集。如果您是 EIB 新手，请从第 11 章 “Edge Image Builder”开始。

41.13.2 为电信部署配置 PTP

许多电信应用场景要求严格保持偏差极小的相位和时间同步，为此定义了两个面向电信的配置文件：ITU-T G.8275.1 和 ITU-T G.8275.2。这两个配置文件均设置了高频率同步消息，以及其他独特特性（如使用替代的最佳主时钟算法 (BMCA)）。这种特性要求 `ptp4l` 所使用的配置文件中包含特定设置，以下几节将提供相关参考信息。

注意

- 以下两节仅介绍时间接收器配置中的普通时钟场景。
- 对于任何此类配置文件，都必须在精心规划的 PTP 基础架构中使用。
- 要用于您的特定 PTP 网络，可能需要额外微调配置，请务必检查并根据需要调整所提供的示例。

41.13.2.1 PTP 配置文件 ITU-T G.8275.1

G.8275.1 配置文件具有以下特点：

- 直接运行在以太网上，需要完整的网络支持（相邻节点/交换机必须支持 PTP）。
- 默认域设置为 24。
- 数据集比较基于 G.8275.x 算法及其在 `priority2` 之后的 `localPriority` 值。

将以下内容复制到名为 `/etc/ptp4l-G.8275.1.conf` 的文件中：

```
# Telecom G.8275.1 example configuration
[global]
domainNumber                    24
priority2   255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout  3
serverOnly                      0
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
```

创建好文件后，必须在 `/etc/sysconfig/ptp4l` 中引用该文件，才能使守护程序正确启动。可通过将 `OPTIONS=` 行更改为以下内容实现：

```
OPTIONS="-f /etc/ptp4l-G.8275.1.conf -i $IFNAME --message_tag ptp-8275.1"
```

具体说明：

- `-f` 指定要使用的配置文件名称（此处为 `/etc/ptp4l-G.8275.1.conf`）。
- `-i` 指定要使用的接口名称，请将 `$IFNAME` 替换为实际接口名称。
- `--message_tag` 可用于在系统日志中更好地识别 ptp4l 输出，这是可选参数。

完成上述步骤后，必须（重新）启动 `ptp4l` 守护程序：

```
# systemctl restart ptp4l
```

运行以下命令，通过观察日志来检查同步状态：

```
# journalctl -e -u ptp4l
```

41.13.2.2 PTP 配置文件 ITU-T G.8275.2

G.8275.2 配置文件具有以下特点：

- 运行在 IP 上，不需要完整的网络支持（相邻节点/交换机可以不支持 PTP）。
- 默认域设置为 44。
- 数据集比较基于 G.8275.x 算法及其在 `priority2` 之后的 `localPriority` 值。

将以下内容复制到名为 `/etc/ptp4l-G.8275.2.conf` 的文件中：

```
# Telecom G.8275.2 example configuration
[global]
domainNumber                    44
priority2   255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             0
serverOnly                      0
hybrid_e2e                      1
inhibit_multicast_service       1
unicast_listen                  1
unicast_req_duration            60
logSyncInterval                 -5
logMinDelayReqInterval          -4
announceReceiptTimeout  2
#
# Customize the following for slave operation:
#
[unicast_master_table]
table_id                        1
```

```
logQueryInterval                2
UDPv4                           $PEER_IP_ADDRESS
[$IFNAME]
unicast_master_table            1
```

请确保替换以下占位符：

- `$PEER_IP_ADDRESS` - 要与之通讯的下一个 PTP 节点的 IP 地址，例如提供同步的主时钟或边界时钟。
- `$IFNAME` - 告知 `ptp4l` 使用哪个接口进行 PTP。

创建好文件后，必须在 `/etc/sysconfig/ptp4l` 中引用该文件以及 PTP 所用接口的名称，才能使守护程序正确启动。可通过将 `OPTIONS=` 行更改为以下内容实现：

```
OPTIONS="-f /etc/ptp4l-G.8275.2.conf --message_tag ptp-8275.2"
```

具体说明：

- `-f` 指定要使用的配置文件名称（此处为 `/etc/ptp4l-G.8275.2.conf`）。
- `--message_tag` 可用于在系统日志中更好地识别 ptp4l 输出，这是可选参数。

完成上述步骤后，必须（重新）启动 `ptp4l` 守护程序：

```
# systemctl restart ptp4l
```

运行以下命令，通过观察日志来检查同步状态：

```
# journalctl -e -u ptp4l
```

41.13.2.3 phc2sys 的配置

建议在配置 `phc2sys` 之前先完成 `ptp4l` 的全部配置，不过这不是强制要求。`phc2sys` 不需要配置文件，其执行参数可通过 `/etc/sysconfig/ptp4l` 中的 `OPTIONS=` 变量单独控制，方式与 `ptp4l` 类似：

```
OPTIONS="-s $IFNAME -w"
```

其中，`$IFNAME` 是已在 ptp4l 中设置的接口名称，将用作系统时钟的同步源。此参数用于识别源 PHC。

41.13.3 Cluster API 集成

当通过管理群集和定向置备功能部署群集时，配置文件及 `/etc/sysconfig` 中的两个配置变量均可在置备时部署到主机上。以下是某个群集定义的节选，重点展示了经过修改的 `RKE2ControlPlane` 对象，该对象会在所有主机上部署相同的 G.8275.1 配置文件：

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  serverConfig:
    cni: canal
  agentConfig:
    format: ignition
    cisProfile: cis
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
```

```
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
        storage:
          files:
            - path: /etc/ptp4l-G.8275.1.conf
              overwrite: true
              contents:
                inline: |
                  # Telecom G.8275.1 example configuration
                  [global]
                  domainNumber                    24
                  priority2                       255
                  dataset_comparison              G.8275.x
                  G.8275.portDS.localPriority     128
                  G.8275.defaultDS.localPriority  128
                  maxStepsRemoved                 255
                  logAnnounceInterval             -3
                  logSyncInterval                 -4
                  logMinDelayReqInterval          -4
                  announceReceiptTimeout          3
                  serverOnly                      0
                  ptp_dst_mac                     01:80:C2:00:00:0E
                  network_transport               L2
              mode: 0644
              user:
```

```
              name: root
            group:
              name: root
          - path: /etc/sysconfig/ptp4l
            overwrite: true
            contents:
              inline: |
                ## Path:           Network/LinuxPTP
                ## Description:    Precision Time Protocol (PTP): ptp4l
settings
                ## Type:           string
                ## Default:        "-i eth0 -f /etc/ptp4l.conf"
                ## ServiceRestart: ptp4l
                #
                # Arguments when starting ptp4l(8).
                #
                OPTIONS="-f /etc/ptp4l-G.8275.1.conf -i $IFNAME --message_tag
ptp-8275.1"
            mode: 0644
            user:
              name: root
            group:
              name: root
          - path: /etc/sysconfig/phc2sys
            overwrite: true
            contents:
              inline: |
                ## Path:           Network/LinuxPTP
                ## Description:    Precision Time Protocol (PTP): phc2sys
settings
                ## Type:           string
                ## Default:        "-s eth0 -w"
                ## ServiceRestart: phc2sys
                #
                # Arguments when starting phc2sys(8).
                #
                OPTIONS="-s $IFNAME -w"
            mode: 0644
```

```
            user:
              name: root
            group:
              name: root
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

除了其他变量外，必须为上述定义补全接口名称及其他 Cluster API 对象（如第 42 章 “全自动定向网络置备”中所述）。

注意

- 仅当群集中的硬件保持统一且所有主机都需要采用相同配置（包括接口名称）时，这种方法才适用。
- 也可以使用其他方法，未来版本中将会介绍相关内容。

至此，您的主机应已具备可正常运行的 PTP 堆栈，并将开始协商其 PTP 角色。

42 全自动定向网络置备

42.1 简介

定向网络置备是用于自动置备下游群集的功能。如果您有许多下游群集需要置备并希望自动完成该过程，此功能将非常有用。

管理群集（第 40 章 “设置管理群集”）会自动部署以下组件：

- SUSE Linux Micro RT（操作系统），可以根据使用场景自定义网络、存储、用户和内核参数等配置。
- RKE2（Kubernetes 群集），默认的 CNI 插件为 Cilium。根据具体的应用场景，可以使用某些 CNI 插件，例如 Cilium+Multus。
- SUSE Storage
- SUSE Security
- MetalLB 可用作高可用性多节点群集的负载平衡器。

注意

有关 SUSE Linux Micro 的详细信息，请参见第 9 章 “SUSE Linux Micro”。有关 RKE2 的详细信息，请参见第 16 章 “RKE2”。有关 SUSE Storage 的详细信息，请参见第 17 章 “SUSE Storage”。有关 SUSE Security 的详细信息，请参见第 18 章 “SUSE Security”。

以下章节介绍了不同的定向网络置备工作流程，以及可添加到置备过程的一些附加功能：

- 第 42.2 节 “为联网场景准备下游群集映像”
- 第 42.3 节 “为隔离场景准备下游群集映像”
- 第 42.4 节 “使用定向网络置备来置备下游群集（单节点）”
- 第 42.5 节 “使用定向网络置备来置备下游群集（多节点）”

- 第 42.6 节 “高级网络配置”
- 第 42.7 节 “电信功能（DPDK、SR-IOV、CPU 隔离、大页、NUMA 等）”
- 第 42.8 节 “专用注册表”
- 第 42.9 节 “在隔离场景中置备下游群集”

注意

以下几节介绍如何使用 SUSE Edge for Telco 为不同的定向网络置备工作流程场景做好准备工作。有关不同的部署配置选项示例（包括隔离环境、DHCP 和无 DHCP 网络、专用容器注册表等），请参见 SUSE Edge for Telco 储存库 (https://github.com/suse-edge/atip/tree/release-3.3/telco-examples/edge-clusters)。

42.2 为联网场景准备下游群集映像

Edge Image Builder（第 11 章 “Edge Image Builder”）用于准备经过修改、将置备到下游群集主机上的 SLEMicro 基础映像。

可以通过 Edge Image Builder 完成大部分配置，但本指南仅介绍设置下游群集所需的最低限度配置。

42.2.1 联网场景的先决条件

- 需要安装 Podman (https://podman.io) 或 Rancher Desktop (https://rancherdesktop.io) 等容器运行时，以便能够运行 Edge Image Builder。
- 基础映像将按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述通过 `Base-SelfInstall` 配置文件（对于实时内核，会使用 `Base-RT-SelfInstall` 配置文件）构建。为 x86-64 和 aarch64 这两种体系结构构建基础映像的过程是相同的。
- 要部署 aarch64 下游群集，必须在部署管理群集之前，在 `metal3.yaml` 文件中设置 `deployArchitecture: arm64` 参数（如管理群集文档（??? [436]）中所述）。必须进行此设置，才能确保下游群集使用正确的体系结构。

注意

构建主机的体系结构必须与待构建映像的体系结构一致。也就是说，要构建 `aarch64` 体系结构的映像，必须使用 `aarch64` 体系结构的构建主机；`x86-64` 体系结构同样如此 - 目前不支持跨体系结构构建。

42.2.2 联网场景的映像配置

运行 Edge Image Builder 时，将从主机挂载一个目录，因此需要创建一个目录结构来存储用于定义目标映像的配置文件。

- `downstream-cluster-config.yaml` 是映像定义文件，有关详细信息，请参见第 3 章 “使用 Edge Image Builder 配置独立群集”。
- 基础映像文件夹将包含按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述使用 `Base-SelfInstall` 配置文件（对于实时内核，会使用 `Base-RT-SelfInstall` 配置文件）生成的输出原始映像。必须将该映像复制/移动到 `base-images` 文件夹下。
- `network` 文件夹是可选的，有关详细信息，请参见第 42.2.2.6 节 “高级网络配置的附加脚本”。
- `custom/scripts` 目录包含要在首次引导时运行的脚本：
  1. `01-fix-growfs.sh` 脚本，用于在部署时调整操作系统根分区的大小
  2. `02-performance.sh` 脚本（可选），用于配置系统以调优性能。
  3. `03-sriov.sh` 脚本（可选），用于为 SR-IOV 配置系统。
- `custom/files` 目录包含映像创建过程中要复制到该映像的 `performance-settings.sh` 和 `sriov-auto-filler.sh` 文件。

```
├── downstream-cluster-config.yaml
├── base-images/
│   └ SL-Micro.x86_64-6.1-Base-GM.raw
├── network/
```

```
|   └ configure-network.sh
└── custom/
    └ scripts/
    |   └ 01-fix-growfs.sh
    |   └ 02-performance.sh
    |   └ 03-sriov.sh
    └ files/
        └ performance-settings.sh
        └ sriov-auto-filler.sh
```

42.2.2.1 下游群集映像定义文件

`downstream-cluster-config.yaml` 文件是下游群集映像的主配置文件。下面是通过 Metal[3] 进行部署的极简示例：

```
apiVersion: 1.2
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-GM.raw
  outputImageName: eibimage-output-telco.raw
operatingSystem:
  kernelArgs:
    - ignition.platform.id=openstack
    - net.ifnames=1
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
      - fstrim
      - time-sync.target
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $USERKEY1
```

```
  packages:
    packageList:
      - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

其中，`$SCC_REGISTRATION_CODE` 是从 SUSE Customer Center (https://scc.suse.com/) 中复制的注册代码，并且软件包列表包含必需的 `jq`。

`$ROOT_PASSWORD` 是 root 用户的已加密口令，可用于测试/调试目的。可以使用 `openssl passwd-6 PASSWORD` 命令生成此口令

对于生产环境，建议使用可添加到 users 块的 SSH 密钥（需将 `$USERKEY1` 替换为实际 SSH 密钥）。

注意

`arch: x86_64` 为映像的体系结构。对于 arm64 体系结构，请使用 `arch: aarch64`。

`net.ifnames=1` 会启用可预测网络接口命名 (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html)

这与 metal3 chart 的默认配置相匹配，但设置必须与配置的 chart `predictableNicNames` 值相匹配。

另请注意，`ignition.platform.id=openstack` 是必需的，如果不指定此参数，在 Metal[3] 自动化流程中通过 ignition 配置 SLEMicro 时将会失败。

42.2.2.2 Growfs 脚本

目前，在置备后首次引导时，需要使用一个自定义脚本 (`custom/scripts/01-fix-growfs.sh`) 来增大文件系统，使之与磁盘大小匹配。`01-fix-growfs.sh` 脚本包含以下信息：

```
#!/bin/bash
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --
noheadings)"
```

```
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

42.2.2.3 性能脚本

以下可选脚本（`custom/scripts/02-performance.sh`）可用于配置系统以调优性能：

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/performance-settings

# copy the artifacts
cp performance-settings.sh /opt/performance-settings/
```

`custom/files/performance-settings.sh` 是可用于配置系统以调优性能的脚本，可从以下链接 (https://github.com/suse-edge/atip/blob/release-3.3/telco-examples/edge-clusters/dhcp/eib/custom/files/performance-settings.sh) 下载。

42.2.2.4 SR-IOV 脚本

以下可选脚本 (`custom/scripts/03-sriov.sh`) 可用于为 SR-IOV 配置系统：

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/sriov
# copy the artifacts
```

```
cp sriov-auto-filler.sh /opt/sriov/sriov-auto-filler.sh
```

`custom/files/sriov-auto-filler.sh` 是可用于为 SR-IOV 配置系统的脚本，可从以下链接 (https://github.com/suse-edge/atip/blob/release-3.3/telco-examples/edge-clusters/dhcp/eib/custom/files/sriov-auto-filler.sh) 下载。

注意

使用相同的方法添加要在置备过程中执行的您自己的自定义脚本。有关详细信息，请参见第 3 章 “使用 Edge Image Builder 配置独立群集” 。

42.2.2.5 电信工作负载的附加配置

要启用 `dpdk`、`sr-iov` 或 `FEC` 等电信功能，可能需要提供附加软件包，如以下示例中所示。

```
apiVersion: 1.2
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.1-Base-GM.raw
  outputImageName: eibimage-output-telco.raw
operatingSystem:
  kernelArgs:
    - ignition.platform.id=openstack
    - net.ifnames=1
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
      - fstrim
      - time-sync.target
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $user1Key1
```

```
  packages:
    packageList:
      - jq
      - dpdk
      - dpdk-tools
      - libdpdk-23
      - pf-bb-config
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

其中，`$SCC_REGISTRATION_CODE` 是从 SUSE Customer Center (https://scc.suse.com/) 中复制的注册代码，并且软件包列表包含要用于 Telco 配置文件的最低限度的软件包。

注意

`arch: x86_64` 为映像的体系结构。对于 arm64 体系结构，请使用 `arch: aarch64`。

42.2.2.6 高级网络配置的附加脚本

如果您需要配置静态 IP 或第 42.6 节 “高级网络配置”中所述的更高级网络方案，则需要提供以下附加配置。

在 `network` 文件夹中创建以下 `configure-network.sh` 文件 - 这会在首次引导时使用配置驱动器数据，并使用 NM Configurator 工具 (https://github.com/suse-edge/nm-configurator) 来配置主机网络。

```
#!/bin/bash

set -eux

# Attempt to statically configure a NIC in the case where we find a
 network_data.json
# In a configuration drive

CONFIG_DRIVE=$(blkid --label config-2 || true)
if [ -z "${CONFIG_DRIVE}" ]; then
  echo "No config-2 device found, skipping network configuration"
  exit 0
```

```
fi

mount -o ro $CONFIG_DRIVE /mnt

NETWORK_DATA_FILE="/mnt/openstack/latest/network_data.json"

if [ ! -f "${NETWORK_DATA_FILE}" ]; then
  umount /mnt
  echo "No network_data.json found, skipping network configuration"
  exit 0
fi

DESIRED_HOSTNAME=$(cat /mnt/openstack/latest/meta_data.json | tr ',{}' '\n' |
 grep '\"metal3-name\"' | sed 's/.*\"metal3-name\": \"\(.*\)\"/\1/')
echo "${DESIRED_HOSTNAME}" > /etc/hostname

mkdir -p /tmp/nmc/{desired,generated}
cp ${NETWORK_DATA_FILE} /tmp/nmc/desired/_all.yaml
umount /mnt

./nmc generate --config-dir /tmp/nmc/desired --output-dir /tmp/nmc/generated
./nmc apply --config-dir /tmp/nmc/generated
```

42.2.3 映像创建

按照前面的章节准备好目录结构后，运行以下命令来构建映像：

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
 build --definition-file downstream-cluster-config.yaml
```

这会根据上述定义创建名为 `eibimage-output-telco.raw` 的输出 ISO 映像文件。

然后必须通过 Web 服务器提供输出映像，该服务器可以是根据管理群集文档（注意）启用的媒体服务器容器，也可以是其他某个本地可访问的服务器。在下面的示例中，此服务器是 `imagecache.local:8080`

42.3 为隔离场景准备下游群集映像

Edge Image Builder（第 11 章 “Edge Image Builder”）用于准备经过修改、将置备到下游群集主机上的 SLEMicro 基础映像。

可以通过 Edge Image Builder 完成大部分配置，但本指南仅介绍为隔离场景设置下游群集所需的最低限度配置。

42.3.1 隔离场景的先决条件

- 需要安装 Podman (https://podman.io) 或 Rancher Desktop (https://rancherdesktop.io) 等容器运行时，以便能够运行 Edge Image Builder。
- 基础映像将按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述通过 `Base-SelfInstall` 配置文件（对于实时内核，会使用 `Base-RT-SelfInstall` 配置文件）构建。为 x86-64 和 aarch64 这两种体系结构构建基础映像的过程是相同的。
- 要部署 aarch64 下游群集，必须在部署管理群集之前，在 `metal3.yaml` 文件中设置 `deployArchitecture: arm64` 参数（如管理群集文档（??? [436]）中所述）。必须进行此设置，才能确保下游群集使用正确的体系结构。
- 如果您要使用 SR-IOV 或任何其他需要容器映像的工作负载，则必须部署并事先配置一个本地专用注册表（设置/未设置 TLS 和/或身份验证）。此注册表用于存储 Helm chart OCI 映像及其他映像。

注意

构建主机的体系结构必须与待构建映像的体系结构一致。也就是说，要构建 `aarch64` 体系结构的映像，必须使用 `aarch64` 体系结构的构建主机；`x86-64` 体系结构同样如此 - 目前不支持跨体系结构构建。

42.3.2 隔离场景的映像配置

运行 Edge Image Builder 时，将从主机挂载一个目录，因此需要创建一个目录结构来存储用于定义目标映像的配置文件。

- `downstream-cluster-airgap-config.yaml` 是映像定义文件，有关详细信息，请参见第 3 章 “使用 Edge Image Builder 配置独立群集”。
- 基础映像文件夹将包含按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述使用 `Base-SelfInstall` 配置文件（对于实时内核，会使用 `Base-RT-SelfInstall` 配置文件）生成的输出原始映像。必须将该映像复制/移动到 `base-images` 文件夹下。
- `network` 文件夹是可选的，有关详细信息，请参见第 42.2.2.6 节 “高级网络配置的附加脚本”。
- `custom/scripts` 目录包含要在首次引导时运行的脚本：
  1. `01-fix-growfs.sh` 脚本，用于在部署时调整操作系统根分区的大小。
  2. `02-airgap.sh` 脚本，用于在为隔离环境创建映像时，将映像复制到正确的位置。
  3. `03-performance.sh` 脚本（可选），用于配置系统以调优性能。
  4. `04-sriov.sh` 脚本（可选），用于为 SR-IOV 配置系统。
- `custom/files` 目录包含在映像创建过程中要复制到映像的 `rke2` 和 `cni` 映像。此外，还可以包含可选的 `performance-settings.sh` 和 `sriov-auto-filler.sh` 文件。

```
├── downstream-cluster-airgap-config.yaml
├── base-images/
|   └ SL-Micro.x86_64-6.1-Base-GM.raw
├── network/
|   └ configure-network.sh
└── custom/
    └ files/
    |   └ install.sh
    |   └ rke2-images-cilium.linux-amd64.tar.zst
    |   └ rke2-images-core.linux-amd64.tar.zst
    |   └ rke2-images-multus.linux-amd64.tar.zst
    |   └ rke2-images.linux-amd64.tar.zst
    |   └ rke2.linux-amd64.tar.zst
    |   └ sha256sum-amd64.txt
    |   └ performance-settings.sh
```

```
    |   └ sriov-auto-filler.sh
    └ scripts/
        └ 01-fix-growfs.sh
        └ 02-airgap.sh
        └ 03-performance.sh
        └ 04-sriov.sh
```

42.3.2.1 下游群集映像定义文件

`downstream-cluster-airgap-config.yaml` 文件是下游群集映像的主配置文件，上一节（第 42.2.2.5 节 “电信工作负载的附加配置”）已介绍其内容。

42.3.2.2 Growfs 脚本

目前，在置备后首次引导时，需要使用一个自定义脚本 (`custom/scripts/01-fix-growfs.sh`) 来增大文件系统，使之与磁盘大小匹配。`01-fix-growfs.sh` 脚本包含以下信息：

```
#!/bin/bash
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --
noheadings)"
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

42.3.2.3 隔离脚本

在映像创建过程中，需要使用以下脚本 (`custom/scripts/02-airgap.sh`) 将映像复制到正确的位置：

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/rke2-artifacts
mkdir -p /var/lib/rancher/rke2/agent/images

# copy the artifacts
cp install.sh /opt/
cp rke2-images*.tar.zst rke2.linux-amd64.tar.gz sha256sum-amd64.txt /opt/rke2-
artifacts/
```

42.3.2.4 性能脚本

以下可选脚本（`custom/scripts/03-performance.sh`）可用于配置系统以调优性能：

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/performance-settings

# copy the artifacts
cp performance-settings.sh /opt/performance-settings/
```

`custom/files/performance-settings.sh` 是可用于配置系统以调优性能的脚本，可从以下链接 (https://github.com/suse-edge/atip/blob/release-3.3/telco-examples/edge-clusters/dhcp/eib/custom/files/performance-settings.sh) 下载。

42.3.2.5 SR-IOV 脚本

以下可选脚本 (`custom/scripts/04-sriov.sh`) 可用于为 SR-IOV 配置系统：

```
#!/bin/bash
```

```
# create the folder to extract the artifacts there
mkdir -p /opt/sriov
# copy the artifacts
cp sriov-auto-filler.sh /opt/sriov/sriov-auto-filler.sh
```

`custom/files/sriov-auto-filler.sh` 是可用于为 SR-IOV 配置系统的脚本，可从以下链接 (https://github.com/suse-edge/atip/blob/release-3.3/telco-examples/edge-clusters/dhcp/eib/custom/files/sriov-auto-filler.sh) 下载。

42.3.2.6 隔离场景的自定义文件

`custom/files` 目录包含映像创建过程中要复制到该映像的 `rke2` 和 `cni` 映像。为了轻松生成映像，请使用以下脚本 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-save-images.sh) 和此处 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-release-rke2-images.txt) 的映像列表在本地准备这些映像，以生成需要包含在 `custom/files` 中的制品。另外，可以从此处 (https://get.rke2.io/) 下载最新的 `rke2-install` 脚本。

```
$ ./edge-save-rke2-images.sh -o custom/files -l ~/edge-release-rke2-images.txt
```

下载映像后，目录结构应如下所示：

```
└── custom/
    └ files/
        └ install.sh
        └ rke2-images-cilium.linux-amd64.tar.zst
        └ rke2-images-core.linux-amd64.tar.zst
        └ rke2-images-multus.linux-amd64.tar.zst
        └ rke2-images.linux-amd64.tar.zst
        └ rke2.linux-amd64.tar.zst
        └ sha256sum-amd64.txt
```

42.3.2.7 预加载包含隔离场景和 SR-IOV 所需映像的专用注册表（可选）

如果您要在隔离场景中使用 SR-IOV 或要使用任何其他工作负载映像，必须按照以下步骤预加载包含这些映像的本地专用注册表：

- 下载、提取 helm-chart OCI 映像并将其推送到专用注册表
- 下载、提取所需的其余映像并将其推送到专用注册表

可使用以下脚本下载、提取映像并将其推送到专用注册表。本节将通过一个示例来说明如何预加载 SR-IOV 映像，但您也可以使用相同的方法来预加载任何其他自定义映像：

1. 预加载 SR-IOV 所需的 helm-chart OCI 映像：

   a. 必须创建一个包含所需 helm-chart OCI 映像的列表：

   ```
   $ cat > edge-release-helm-oci-artifacts.txt <<EOF
   edge/sriov-network-operator-chart:303.0.2+up1.5.0
   edge/sriov-crd-chart:303.0.2+up1.5.0
   EOF
   ```

   b. 使用以下脚本 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-save-oci-artefacts.sh) 和上面创建的列表生成本地 tarball 文件：

   ```
   $ ./edge-save-oci-artefacts.sh -al ./edge-release-helm-oci-
   artifacts.txt -s registry.suse.com
   Pulled: registry.suse.com/edge/charts/sriov-network-
   operator:303.0.2+up1.5.0
   Pulled: registry.suse.com/edge/charts/sriov-crd:303.0.2+up1.5.0
   a edge-release-oci-tgz-20240705
   a edge-release-oci-tgz-20240705/sriov-network-operator-
   chart-303.0.2+up1.5.0.tgz
   a edge-release-oci-tgz-20240705/sriov-crd-chart-303.0.2+up1.5.0.tgz
   ```

   c. 使用以下脚本 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-load-oci-artefacts.sh) 将 tarball 文件上载到专用注册表（例如 `myregistry:5000`），以在您的注册表中预加载上一步下载的 Helm chart OCI 映像：

   ```
   $ tar zxvf edge-release-oci-tgz-20240705.tgz
   ```

```
$ ./edge-load-oci-artefacts.sh -ad edge-release-oci-tgz-20240705 -r
 myregistry:5000
```

2. 预加载 SR-IOV 所需的其余映像：

   a. 在这种情况下，必须包含电信工作负载的 sr-iov 容器映像（例如，作为参考，您可以从 helm-chart 值 (https://github.com/suse-edge/charts/blob/main/charts/sriov-network-operator/1.5.0/values.yaml) 获取这些映像）

```
$ cat > edge-release-images.txt <<EOF
rancher/hardened-sriov-network-operator:v1.3.0-build20240816
rancher/hardened-sriov-network-config-daemon:v1.3.0-build20240816
rancher/hardened-sriov-cni:v2.8.1-build20240820
rancher/hardened-ib-sriov-cni:v1.1.1-build20240816
rancher/hardened-sriov-network-device-plugin:v3.7.0-build20240816
rancher/hardened-sriov-network-resources-injector:v1.6.0-build20240816
rancher/hardened-sriov-network-webhook:v1.3.0-build20240816
EOF
```

   b. 必须使用以下脚本 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-save-images.sh) 和上面创建的列表，在本地生成包含所需映像的 tarball 文件：

```
$ ./edge-save-images.sh -l ./edge-release-images.txt -s
 registry.suse.com
Image pull success: registry.suse.com/rancher/hardened-sriov-network-
operator:v1.3.0-build20240816
Image pull success: registry.suse.com/rancher/hardened-sriov-network-
config-daemon:v1.3.0-build20240816
Image pull success: registry.suse.com/rancher/hardened-sriov-
cni:v2.8.1-build20240820
Image pull success: registry.suse.com/rancher/hardened-ib-sriov-
cni:v1.1.1-build20240816
Image pull success: registry.suse.com/rancher/hardened-sriov-network-
device-plugin:v3.7.0-build20240816
Image pull success: registry.suse.com/rancher/hardened-sriov-network-
resources-injector:v1.6.0-build20240816
Image pull success: registry.suse.com/rancher/hardened-sriov-network-
webhook:v1.3.0-build20240816
```

```
Creating edge-images.tar.gz with 7 images
```

c. 使用以下脚本 (https://github.com/suse-edge/fleet-examples/blob/release-3.3.0/scripts/day2/edge-load-images.sh) 将 tarball 文件上载到专用注册表（例如 `myregistry:5000`），以在您的专用注册表中预加载上一步下载的映像：

```
$ tar zxvf edge-release-images-tgz-20240705.tgz
$ ./edge-load-images.sh -ad edge-release-images-tgz-20240705 -r
 myregistry:5000
```

42.3.3 为隔离场景创建映像

按照前面的章节准备好目录结构后，运行以下命令来构建映像：

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.3/edge-image-builder:1.2.1 \
 build --definition-file downstream-cluster-airgap-config.yaml
```

这会根据上述定义创建名为 `eibimage-output-telco.raw` 的输出 ISO 映像文件。

然后必须通过 Web 服务器提供输出映像，该服务器可以是根据管理群集文档（注意）启用的媒体服务器容器，也可以是其他某个本地可访问的服务器。在下面的示例中，此服务器是 `imagecache.local:8080`。

42.4 使用定向网络置备来置备下游群集（单节点）

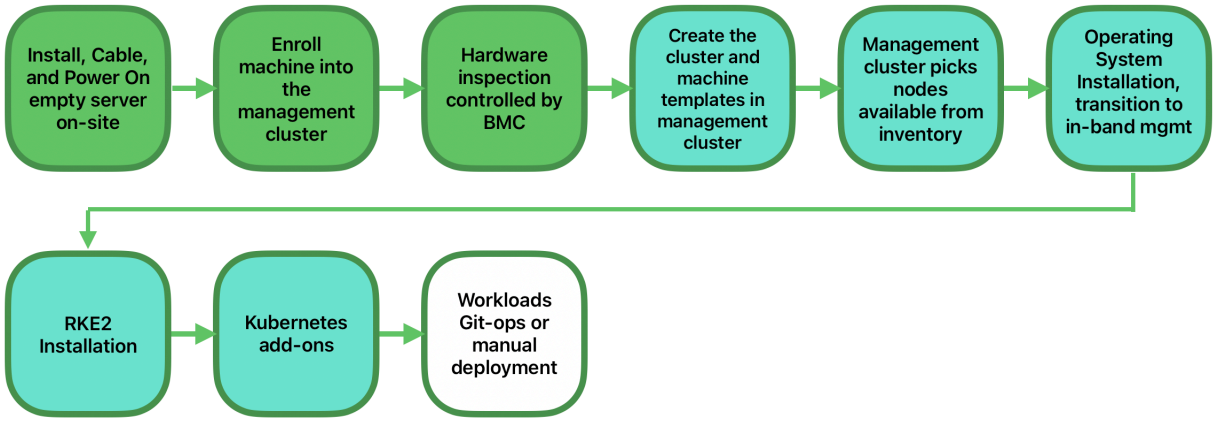
本节介绍用于通过定向网络置备自动置备单节点下游群集的工作流程。这是自动置备下游群集的最简单的方法。

要求

- 如前面的章节（第 42.2 节 “为联网场景准备下游群集映像”）所述使用 EIB 生成的、附带用于设置下游群集的最低限度配置的映像，必须位于管理群集上您按照注意一节所述配置的确切路径中。
- 管理服务器已创建并可在后续章节中使用。有关详细信息，请参见管理群集相关的一章第 40 章 “设置管理群集”。

工作流程

下图显示了用于通过定向网络置备自动置备单节点下游群集的工作流程：



可以执行两个不同的步骤来使用定向网络置备自动置备单节点下游群集：

1. 登记裸机主机，使其在置备过程中可用。
2. 置备裸机主机，以安装并配置操作系统和 Kubernetes 群集。

登记裸机主机

第一步是在管理群集中登记新的裸机主机，使其可供置备。为此，必须在管理群集中创建以下文件 (`bmh-example.yaml`)，以指定要使用的 `BMC` 身份凭证以及要登记的 `BaremetalHost` 对象：

```
apiVersion: v1
kind: Secret
metadata:
  name: example-demo-credentials
type: Opaque
data:
  username: ${BMC_USERNAME}
  password: ${BMC_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: example-demo
```

```
  labels:
    cluster-role: control-plane
spec:
  online: true
  bootMACAddress: ${BMC_MAC}
  rootDeviceHints:
    deviceName: /dev/nvme0n1
  bmc:
    address: ${BMC_ADDRESS}
    disableCertificateVerification: true
    credentialsName: example-demo-credentials
```

其中：

- `${BMC_USERNAME}` — 新裸机主机的 BMC 用户名。
- `${BMC_PASSWORD}` — 新裸机主机的 BMC 口令。
- `${BMC_MAC}` — 要使用的新裸机主机的 MAC 地址。
- `${BMC_ADDRESS}` — 裸机主机 BMC 的 URL（例如 `redfish-virtualmedia://192.168.200.75/redfish/v1/Systems/1/`）。要了解有关硬件提供商支持的不同选项的详细信息，请访问此链接 (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md)。

注意

如果未在映像构建时或通过 `BareMetalHost` 定义指定主机的网络配置，系统将使用自动配置机制（DHCP、DHCPv6、SLAAC）。有关详细信息或复杂配置，请参见第 42.6 节 “高级网络配置”。

创建该文件后，必须在管理群集中执行以下命令，才能在管理群集中开始登记新的裸机主机：

```
$ kubectl apply -f bmh-example.yaml
```

随后会登记新裸机主机对象，其状态将从正在注册依次变为正在检查和可用。可使用以下命令检查状态变化：

```
$ kubectl get bmh
```

注意

在验证 BMC 身份凭证之前，BaremetalHost 对象将一直处于正在注册状态。验证身份凭证后，BaremetalHost 对象的状态将变为正在检查，此步骤可能需要一段时间（最长 20 分钟），具体取决于所用的硬件。在检查阶段，将检索硬件信息并更新 Kubernetes 对象。请使用以下命令检查信息：kubectl get bmh -o yaml。

置备步骤

裸机主机已登记并可供使用后，下一步是置备裸机主机，以安装并配置操作系统和 Kubernetes 群集。为此，必须在管理群集中创建以下文件 (capi-provisioning-example.yaml) 并在其中包含以下信息（可以通过合并以下块来生成 capi-provisioning-example.yaml）。

注意

只需将 ${...} 中的值替换为实际值。

下面的块是群集定义，可在其中使用 pods 和 services 块来配置网络。此外，它还包含对要使用的控制平面和基础架构（使用 Metal3 提供程序）对象的引用。

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: single-node-cluster
  namespace: default
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: RKE2ControlPlane
    name: single-node-cluster
```

```
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3Cluster
    name: single-node-cluster
```

对于双栈 Pod 和服务的部署，可以改为使用以下定义：

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: single-node-cluster
  namespace: default
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
        - fd00:bad:cafe::/48
    services:
      cidrBlocks:
        - 10.96.0.0/12
        - fd00:bad:bad:cafe::/112
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: RKE2ControlPlane
    name: single-node-cluster
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3Cluster
    name: single-node-cluster
```

重要

IPv6 和双栈部署处于技术预览状态，不提供官方支持。

`Metal3Cluster` 对象指定要配置的控制平面端点（请替换 `${DOWNSTREAM_CONTROL_PLANE_IP}`）以及 `noCloudProvider`，因为使用了一个裸机节点。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3Cluster
metadata:
  name: single-node-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: ${DOWNSTREAM_CONTROL_PLANE_IP}
    port: 6443
  noCloudProvider: true
```

RKE2ControlPlane 对象指定要使用的控制平面配置，Metal3MachineTemplate 对象指定要使用的控制平面映像。此外，它还包含有关要使用的复本数（在本例中为 1）以及要使用的 CNI 插件（在本例中为 Cilium）的信息。agentConfig 块包含要使用的 Ignition 格式，以及用于配置 RKE2 节点的 additionalUserData，其中包含名为 rke2-preinstall.service 的 systemd 等信息，用于在置备过程中使用 Ironic 信息自动替换 BAREMETALHOST_UUID 和 node-name。为了启用 multus 和 cilium，会在 rke2 服务器清单目录中创建名为 rke2-cilium-config.yaml 的文件，其中包含要使用的配置。最后一个信息块包含要使用的 Kubernetes 版本。${RKE2_VERSION} 是要使用的 RKE2 版本，请替换此值（例如替换为 v1.32.4+rke2r1）。

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
```

```
  serverConfig:
    cni: cilium
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
        storage:
          files:
            # https://docs.rke2.io/networking/multus_sriov#using-multus-with-
cilium
            - path: /var/lib/rancher/rke2/server/manifests/rke2-cilium-
config.yaml
              overwrite: true
              contents:
                inline: |
```

```
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChartConfig
                  metadata:
                    name: rke2-cilium
                    namespace: kube-system
                  spec:
                    valuesContent: |-
                      cni:
                        exclusive: false
              mode: 0644
              user:
                name: root
              group:
                name: root
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

`Metal3MachineTemplate` 对象指定以下信息：

- `dataTemplate`，用作对模板的引用。
- `hostSelector`，在与登记过程中创建的标签匹配时使用。
- `image`，用作对上一节（第 42.2 节 “为联网场景准备下游群集映像”）中使用 `EIB` 生成的映像的引用；`checksum` 和 `checksumType`，用于验证映像。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: single-node-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: single-node-cluster-controlplane-template
      hostSelector:
```

```
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/eibimage-output-telco.raw.sha256
        checksumType: sha256
        format: raw
        url: http://imagecache.local:8080/eibimage-output-telco.raw
```

`Metal3DataTemplate` 对象指定下游群集的 `metaData`。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: single-node-cluster-controlplane-template
  namespace: default
spec:
  clusterName: single-node-cluster
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
      - key: local_hostname
        object: machine
```

通过合并上述块创建该文件后，必须在管理群集中执行以下命令才能开始置备新的裸机主机：

```
$ kubectl apply -f capi-provisioning-example.yaml
```

42.5 使用定向网络置备来置备下游群集（多节点）

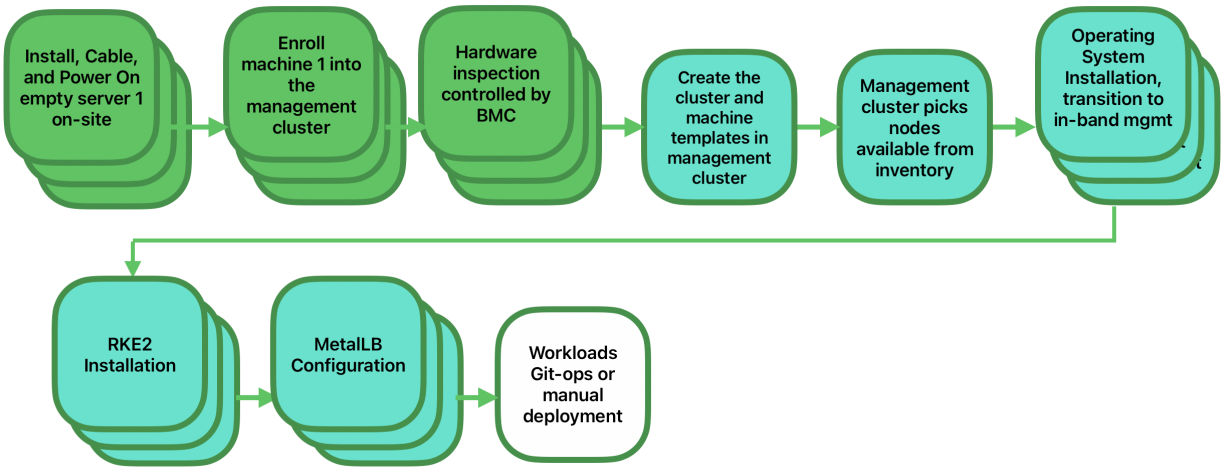
本节介绍用于通过定向网络置备和 `MetalLB`（用作负载平衡器策略）自动置备多节点下游群集的工作流程。这是自动置备下游群集的最简单的方法。下图显示了用于通过定向网络置备和 `MetalLB` 自动置备多节点下游群集的工作流程。

要求

- 如前面的章节（第 42.2 节 “为联网场景准备下游群集映像”）所述使用 EIB 生成的、附带用于设置下游群集的最低限度配置的映像，必须位于管理群集上您按照注意一节所述配置的确切路径中。
- 管理服务器已创建并可在后续章节中使用。有关详细信息，请参见管理群集相关的一章：第 40 章 “设置管理群集”。

工作流程

下图显示了用于通过定向网络置备自动置备多节点下游群集的工作流程：



1. 登记三个裸机主机，使其在置备过程中可用。
2. 置备三个裸机主机，以使用 MetalLB 安装并配置操作系统和 Kubernetes 群集。

登记裸机主机

第一步是在管理群集中登记三个裸机主机，使其可供置备。为此，必须在管理群集中创建以下文件（bmh-example-node1.yaml、bmh-example-node2.yaml 和 bmh-example-node3.yaml），以指定要使用的 BMC 身份凭证，以及要在管理群集中登记的 BaremetalHost 对象。

注意

- 只需将 `${...}` 中的值替换为实际值。
- 本节只会指导您完成一个主机的置备过程。这些步骤同样适用于另外两个节点。

```
apiVersion: v1
kind: Secret
metadata:
  name: node1-example-credentials
type: Opaque
data:
  username: ${BMC_NODE1_USERNAME}
  password: ${BMC_NODE1_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: node1-example
  labels:
    cluster-role: control-plane
spec:
  online: true
  bootMACAddress: ${BMC_NODE1_MAC}
  bmc:
    address: ${BMC_NODE1_ADDRESS}
    disableCertificateVerification: true
    credentialsName: node1-example-credentials
```

其中:

- `${BMC_NODE1_USERNAME}` — 第一个裸机主机的 BMC 用户名。
- `${BMC_NODE1_PASSWORD}` — 第一个裸机主机的 BMC 口令。

- `${BMC_NODE1_MAC}` — 第一个裸机主机的要使用的 MAC 地址。
- `${BMC_NODE1_ADDRESS}` — 第一个裸机主机的 BMC 的 URL（例如 `redfish-virtualmedia://192.168.200.75/redfish/v1/Systems/1/`）。要了解有关硬件提供商支持的不同选项的详细信息，请访问此链接 (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md)。

注意

- 如果未在映像构建时或通过 `BareMetalHost` 定义指定主机的网络配置，系统将使用自动配置机制（DHCP、DHCPv6、SLAAC）。有关详细信息或复杂配置，请参见第 42.6 节 “高级网络配置”。
- 尚不支持多节点双栈群集或仅支持 IPv6 的群集。

创建该文件后，必须在管理群集中执行以下命令，才能在管理群集中开始登记裸机主机：

```
$ kubectl apply -f bmh-example-node1.yaml
$ kubectl apply -f bmh-example-node2.yaml
$ kubectl apply -f bmh-example-node3.yaml
```

随后会登记新裸机主机对象，其状态将从正在注册依次变为正在检查和可用。可使用以下命令检查状态变化：

```
$ kubectl get bmh -o wide
```

注意

在验证 BMC 身份凭证之前，`BaremetalHost` 对象将一直处于正在注册状态。验证身份凭证后，`BaremetalHost` 对象的状态将变为正在检查，此步骤可能需要一段时间（最长 20 分钟），具体取决于所用的硬件。在检查阶段，将检索硬件信息并更新 Kubernetes 对象。请使用以下命令检查信息：`kubectl get bmh -o yaml`。

置备步骤

三个裸机主机已登记并可供使用后，下一步是置备裸机主机，以安装和配置操作系统与 Kubernetes 群集，并创建用于管理该操作系统和群集的负载平衡器。为此，必须在管理群集中创建以下文件 (`capi-provisioning-example.yaml`) 并在其中包含以下信息（可以通过合并以下块来生成 capi-provisioning-example.yaml）。

注意

- 只需将 `${...}` 中的值替换为实际值。
- `VIP` 地址是尚未分配给任何节点的预留 IP 地址，用于配置负载平衡器。

下面的内容为群集定义，可在其中使用 `pods` 和 `services` 块来配置群集网络。此外，它还包含对要使用的控制平面和基础架构（使用 `Metal3` 提供程序）对象的引用。

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: multinode-cluster
  namespace: default
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: RKE2ControlPlane
    name: multinode-cluster
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3Cluster
    name: multinode-cluster
```

Metal3Cluster 对象指定使用已预留的 VIP 地址（请替换 ${DOWNSTREAM_VIP_ADDRESS}）的待配置控制平面端点，以及 noCloudProvider（因为使用了三个裸机节点）。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3Cluster
metadata:
  name: multinode-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: ${EDGE_VIP_ADDRESS}
    port: 6443
  noCloudProvider: true
```

RKE2ControlPlane 对象指定要使用的控制平面配置，Metal3MachineTemplate 对象指定要使用的控制平面映像。

- 要使用的复本数（在本例中为 3）。
- 负载平衡器要使用的通告模式（address 使用 L2 实现），以及要使用的地址（请将 ${EDGE_VIP_ADDRESS} 替换为 VIP 地址）。
- serverConfig，其中包含要使用的 CNI 插件（在本例中为 Cilium）；用于配置 VIP 地址的 tlsSan。
- agentConfig 块包含要使用的 Ignition 格式以及用于配置 RKE2 节点的 additionalUserData，其中的信息如下：
  - 名为 rke2-preinstall.service 的 systemd 服务，用于在置备过程中使用 Ironic 信息自动替换 BAREMETALHOST_UUID 和 node-name。
  - storage 块，其中包含用于安装 MetalLB 和 endpoint-copier-operator 的 Helm chart。

    - `metalLB` 自定义资源文件，其中包含要使用的 `IPaddressPool` 和 `L2Advertisement`（请将 `${EDGE_VIP_ADDRESS}` 替换为 `VIP` 地址）。
    - 用于配置 `kubernetes-vip` 服务的 `end-svc.yaml` 文件，`MetalLB` 使用该服务来管理 `VIP` 地址。
- 最后一个信息块包含要使用的 Kubernetes 版本。`${RKE2_VERSION}` 是要使用的 `RKE2` 版本，请替换此值（例如替换为 `v1.32.4+rke2r1`）。

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: multinode-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: multinode-cluster-controlplane
  replicas: 3
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  registrationAddress: ${EDGE_VIP_ADDRESS}
  serverConfig:
    cni: cilium
    tlsSan:
      - ${EDGE_VIP_ADDRESS}
      - https://${EDGE_VIP_ADDRESS}.sslip.io
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
```

```
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
        storage:
          files:
            # https://docs.rke2.io/networking/multus_sriov#using-multus-with-
cilium
            - path: /var/lib/rancher/rke2/server/manifests/rke2-cilium-
config.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChartConfig
                  metadata:
                    name: rke2-cilium
                    namespace: kube-system
                  spec:
                    valuesContent: |-
                      cni:
```

```
                        exclusive: false
              mode: 0644
              user:
                name: root
              group:
                name: root
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-
operator.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: endpoint-copier-operator
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/endpoint-copier-
operator
                    targetNamespace: endpoint-copier-operator
                    version: 303.0.0+up0.2.1
                    createNamespace: true
            - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: metallb
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/metallb
                    targetNamespace: metallb-system
                    version: 303.0.0+up0.14.9
                    createNamespace: true

            - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
```

```
              overwrite: true
              contents:
                inline: |
                  apiVersion: metallb.io/v1beta1
                  kind: IPAddressPool
                  metadata:
                    name: kubernetes-vip-ip-pool
                    namespace: metallb-system
                  spec:
                    addresses:
                      - ${EDGE_VIP_ADDRESS}/32
                    serviceAllocation:
                      priority: 100
                      namespaces:
                        - default
                      serviceSelectors:
                        - matchExpressions:
                          - {key: "serviceType", operator: In, values:
[kubernetes-vip]}
                  ---
                  apiVersion: metallb.io/v1beta1
                  kind: L2Advertisement
                  metadata:
                    name: ip-pool-l2-adv
                    namespace: metallb-system
                  spec:
                    ipAddressPools:
                      - kubernetes-vip-ip-pool
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: v1
                  kind: Service
                  metadata:
                    name: kubernetes-vip
                    namespace: default
                    labels:
```

```
                    serviceType: kubernetes-vip
                spec:
                  ports:
                  - name: rke2-api
                    port: 9345
                    protocol: TCP
                    targetPort: 9345
                  - name: k8s-api
                    port: 6443
                    protocol: TCP
                    targetPort: 6443
                  type: LoadBalancer
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "Node-multinode-cluster"
```

Metal3MachineTemplate 对象指定以下信息：

- dataTemplate，用作对模板的引用。
- hostSelector，在与登记过程中创建的标签匹配时使用。
- image，用作对上一节（第 42.2 节 “为联网场景准备下游群集映像”）中使用 EIB 生成的映像的引用；checksum 和 checksumType，用于验证映像。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
  name: multinode-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: multinode-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
```

```
      image:
        checksum: http://imagecache.local:8080/eibimage-output-telco.raw.sha256
        checksumType: sha256
        format: raw
        url: http://imagecache.local:8080/eibimage-output-telco.raw
```

Metal3DataTemplate 对象指定下游群集的 metaData。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3DataTemplate
metadata:
  name: multinode-cluster-controlplane-template
  namespace: default
spec:
  clusterName: multinode-cluster
  metaData:
    objectNames:
      - key: name
        object: machine
      - key: local-hostname
        object: machine
      - key: local_hostname
        object: machine
```

通过合并上述块创建该文件后，必须在管理群集中执行以下命令开始置备三个新的裸机主机：

```
$ kubectl apply -f capi-provisioning-example.yaml
```

42.6 高级网络配置

定向网络置备工作流程允许在下游群集中使用静态 IP、绑定、VLAN 等网络配置。

以下章节将介绍使用高级网络配置置备下游群集需额外执行的步骤。

要求

- 使用 EIB 生成的映像必须包含第 42.2.2.6 节 “高级网络配置的附加脚本”一节中所述的 network 文件夹和脚本。

配置

在继续操作之前，请参考以下章节，了解登记和置备主机所需执行的步骤：

- 使用定向网络置备来置备下游群集（单节点）（第 42.4 节 “使用定向网络置备来置备下游群集（单节点）”）
- 使用定向网络置备来置备下游群集（多节点）（第 42.5 节 “使用定向网络置备来置备下游群集（多节点）”）

任何高级网络配置都必须在登记时通过 `BareMetalHost` 主机定义以及包含 `nmstate` 格式 `networkData` 块的关联机密来应用。以下示例文件定义了一个包含所需 `networkData` 的机密，该机密会为下游群集主机请求静态 `IP` 和 `VLAN`：

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: ${CONTROLPLANE_INTERFACE}
      type: ethernet
      state: up
      mtu: 1500
      identifier: mac-address
      mac-address: "${CONTROLPLANE_MAC}"
      ipv4:
        address:
        - ip:  "${CONTROLPLANE_IP}"
          prefix-length: "${CONTROLPLANE_PREFIX}"
        enabled: true
        dhcp: false
    - name: floating
      type: vlan
      state: up
      vlan:
        base-iface: ${CONTROLPLANE_INTERFACE}
        id: ${VLAN_ID}
```

```
    dns-resolver:
      config:
        server:
        - "${DNS_SERVER}"
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: "${CONTROLPLANE_GATEWAY}"
        next-hop-interface: ${CONTROLPLANE_INTERFACE}
```

如您所见，该示例展示了启用配备静态 IP 的接口的配置，以及使用基础接口启用 VLAN 的配置，在实际应用时，需根据您的基础架构将以下变量替换为实际值：

- `${CONTROLPLANE1_INTERFACE}` — 用于边缘群集的控制平面接口（例如 `eth0`）。如果包含 `identifier: mac-address`，系统将会根据 MAC 地址自动检测命名，因此可以使用任何接口名称。
- `${CONTROLPLANE1_IP}` — 用作边缘群集端点的 IP 地址（必须与 kubeapi-server 端点匹配）。
- `${CONTROLPLANE1_PREFIX}` — 用于边缘群集的 CIDR（例如，如果您要使用 `/24` 子网掩码，请指定 `24`；也可以指定 `255.255.255.0`）。
- `${CONTROLPLANE1_GATEWAY}` — 用于边缘群集的网关（例如 `192.168.100.1`）。
- `${CONTROLPLANE1_MAC}` — 用于控制平面接口的 MAC 地址（例如 `00:0c:29:3e:3e:3e`）。
- `${DNS_SERVER}` — 用于边缘群集的 DNS（例如 `192.168.100.2`）。
- `${VLAN_ID}` — 用于边缘群集的 VLAN ID（例如 `100`）。

任何其他符合 `nmstate` 标准的定义都可用于配置下游群集的网络，以适应特定要求。例如，可以指定静态双栈配置：

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
```

```
stringData:
  networkData: |
    interfaces:
    - name: ${CONTROLPLANE_INTERFACE}
      type: ethernet
      state: up
      mac-address: ${CONTROLPLANE_MAC}
      ipv4:
        enabled: true
        dhcp: false
        address:
        - ip: ${CONTROLPLANE_IP_V4}
          prefix-length: ${CONTROLPLANE_PREFIX_V4}
      ipv6:
        enabled: true
        dhcp: false
        autoconf: false
        address:
        - ip: ${CONTROLPLANE_IP_V6}
          prefix-length: ${CONTROLPLANE_PREFIX_V6}
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: ${CONTROLPLANE_GATEWAY_V4}
        next-hop-interface: ${CONTROLPLANE_INTERFACE}
      - destination: ::/0
        next-hop-address: ${CONTROLPLANE_GATEWAY_V6}
        next-hop-interface: ${CONTROLPLANE_INTERFACE}
    dns-resolver:
      config:
        server:
        - ${DNS_SERVER_V4}
        - ${DNS_SERVER_V6}
```

与前面的示例一样，请根据您的基础架构将以下变量替换为实际值：

- `${CONTROLPLANE_IP_V4}` - 分配给主机的 IPv4 地址
- `${CONTROLPLANE_PREFIX_V4}` - 主机 IP 所属网络的 IPv4 前缀

- `${CONTROLPLANE_IP_V6}` - 分配给主机的 IPv6 地址
- `${CONTROLPLANE_PREFIX_V6}` - 主机 IP 所属网络的 IPv6 前缀
- `${CONTROLPLANE_GATEWAY_V4}` - 与默认路由匹配的流量所用网关的 IPv4 地址
- `${CONTROLPLANE_GATEWAY_V6}` - 与默认路由匹配的流量所用网关的 IPv6 地址
- `${CONTROLPLANE_INTERFACE}` - 地址要分配到的接口的名称，该接口同时供与默认路由匹配的出站流量使用（适用于 IPv4 和 IPv6）
- `${DNS_SERVER_V4}` 和/或 `${DNS_SERVER_V6}` - 要使用的 DNS 服务器的 IP 地址（可指定单项或多项，支持 IPv4 和/或 IPv6 地址）

重要

IPv6 和双栈部署处于技术预览状态，不提供官方支持。

注意

有关更复杂的示例（包括仅支持 IPv6 的配置和双栈配置），您可以参考 SUSE Edge for Telco 示例代码库 (https://github.com/suse-edge/atip/tree/main/telco-examples/edge-clusters)。

最后，无论网络配置详细信息如何，都要确保通过在 `BaremetalHost` 对象中附加 `preprovisioningNetworkDataName` 来引用该机密，以便成功在管理群集中登记主机。

```
apiVersion: v1
kind: Secret
metadata:
  name: example-demo-credentials
type: Opaque
data:
  username: ${BMC_USERNAME}
  password: ${BMC_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
```

```
kind: BareMetalHost
metadata:
  name: example-demo
  labels:
    cluster-role: control-plane
spec:
  online: true
  bootMACAddress: ${BMC_MAC}
  rootDeviceHints:
    deviceName: /dev/nvme0n1
  bmc:
    address: ${BMC_ADDRESS}
    disableCertificateVerification: true
    credentialsName: example-demo-credentials
  preprovisioningNetworkDataName: controlplane-0-networkdata
```

注意

- 如果需要部署多节点群集，必须对每个节点执行相同的过程。
- 目前不支持 `Metal3DataTemplate`、`networkData` 和 `Metal3 IPAM`；只有通过静态机密进行的配置才完全受支持。

42.7 电信功能（DPDK、SR-IOV、CPU 隔离、大页、NUMA 等）

定向网络置备工作流程允许将下游群集中使用的电信功能自动化，以便在这些服务器上运行电信工作负载。

要求

- 如前面的章节（第 42.2 节 “为联网场景准备下游群集映像”）所述使用 EIB 生成的映像必须位于管理群集上您按照注意一节所述配置的确切路径中。
- 使用 EIB 生成的映像必须包含第 42.2.2.5 节 “电信工作负载的附加配置”一节中所述的特定电信软件包。
- 管理服务器已创建并可在后续章节中使用。有关详细信息，请参见管理群集相关的一章：第 40 章 “设置管理群集”。

配置

基于以下两个章节的内容登记和置备主机：

- 使用定向网络置备来置备下游群集（单节点）（第 42.4 节 “使用定向网络置备来置备下游群集（单节点）”）
- 使用定向网络置备来置备下游群集（多节点）（第 42.5 节 “使用定向网络置备来置备下游群集（多节点）”）

本节将介绍以下电信功能：

- DPDK 和 VF 创建
- 工作负载使用的 SR-IOV 和 VF 分配
- CPU 隔离和性能调优
- 大页配置
- 内核参数调优

注意

有关电信功能的详细信息，请参见第 41 章 “电信功能配置”。

启用上述电信功能所需做出的更改都可以在 `capi-provisioning-example.yaml` 置备文件的 `RKE2ControlPlane` 块中完成。`capi-provisioning-example.yaml` 文件中的其余信息与置备相关章节（第 42.4 节 “使用定向网络置备来置备下游群集（单节点）” [511]）中提供的信息相同。

明确地说，为启用电信功能而需在该块 (`RKE2ControlPlane`) 中进行的更改如下：

- 指定 preRKE2Commands，该参数用于在 RKE2 安装过程开始之前执行命令。本例使用 modprobe 命令启用 vfio-pci 和 SR-IOV 内核模块。
- 指定 ignition 文件 /var/lib/rancher/rke2/server/manifests/configmap-sriov-custom-auto.yaml，该文件用于定义要创建且向工作负载公开的接口、驱动程序及 VF 数量。
    - 只有 sriov-custom-auto-config 配置映射中的值可以替换为实际值。
        - ${RESOURCE_NAME1} — 用于第一个 PF 接口的资源名称（例如 sriov-resource-du1）。它将添加到前缀 rancher.io 的后面，供工作负载用作标签（例如 rancher.io/sriov-resource-du1）。
        - ${SRIOV-NIC-NAME1} — 要使用的第一个 PF 接口的名称（例如 eth0）。
        - ${PF_NAME1} — 要使用的第一个物理功能 PF 的名称。可以使用此名称生成更复杂的过滤器（例如 eth0#2-5）。
        - ${DRIVER_NAME1} — 用于第一个 VF 接口的驱动程序名称（例如 vfio-pci）。
        - ${NUM_VFS1} — 要为第一个 PF 接口创建的 VF 数量（例如 8）。
- 提供 /var/sriov-auto-filler.sh，用作高层级配置映射 sriov-custom-auto-config 与包含底层硬件信息的 sriovnetworknodepolicy 之间的转换器。创建此脚本的目的是简化用户操作，让其无需提前了解复杂的硬件信息。不需要在此文件中进行更改，但如果我们需要启用 sr-iov 并创建 VF，则应该提供此脚本。
- 用于启用以下功能的内核参数：

参数	值	说明
isolcpus	domain,nohz, managed_irq,1-30,33-62	隔离核心 1-30 和 33-62。
skew_tick	1	允许内核在隔离的 CPU 之间错开定时器中断。

nohz	on	允许内核在系统空闲时在单个 CPU 上运行定时器节拍周期。
nohz_full	1-30,33-62	内核引导参数是当前用于配置完整 dynticks 及 CPU 隔离的主接口。
rcu_nocbs	1-30,33-62	允许内核在系统空闲时在单个 CPU 上运行 RCU 回调。
irqaffinity	0,31,32,63	允许内核在系统空闲时在单个 CPU 上运行中断。
idle	poll	最大限度地减少因退出空闲状态造成的延迟。
iommu	pt	允许为 dpdk 接口使用 vfio。
intel_iommu	on	允许为 VF 使用 vfio。
hugepagesz	1G	允许将大页的大小设置为 1 G。
hugepages	40	先前定义的大页数量。
default_hugepagesz	1G	用于启用大页的默认值。
nowatchdog		禁用看门狗。
nmi_watchdog	0	禁用 NMI 看门狗。

- 以下 systemd 服务用于启用下述功能：
  - `rke2-preinstall.service`，用于在置备过程中使用 Ironic 信息自动替换 `BAREMETALHOST_UUID` 和 `node-name`。
  - `cpu-partitioning.service`，用于启用 `CPU` 核心隔离（例如 `1-30,33-62`）。

    - performance-settings.service，用于调优 CPU 性能。
    - sriov-custom-auto-vfs.service，用于执行以下操作：安装 sriov Helm chart，等待自定义资源创建完成，运行 /var/sriov-auto-filler.sh 以替换配置映射 sriov-custom-auto-config 中的值，并创建工作负载要使用的 sriovnetworknodepolicy。
- ${RKE2_VERSION} 是要使用的 RKE2 版本，请替换此值（例如替换为 v1.32.4+rke2r1）。

做出上述所有更改后，capi-provisioning-example.yaml 中的 RKE2ControlPlane 块如下所示：

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  serverConfig:
    cni: calico
    cniMultusEnable: true
  preRKE2Commands:
    - modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
```

```
        variant: fcos
        version: 1.4.0
        storage:
          files:
            - path: /var/lib/rancher/rke2/server/manifests/configmap-sriov-
custom-auto.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: v1
                  kind: ConfigMap
                  metadata:
                    name: sriov-custom-auto-config
                    namespace: kube-system
                  data:
                    config.json: |
                      [
                         {
                           "resourceName": "${RESOURCE_NAME1}",
                           "interface": "${SRIOV-NIC-NAME1}",
                           "pfname": "${PF_NAME1}",
                           "driver": "${DRIVER_NAME1}",
                           "numVFsToCreate": ${NUM_VFS1}
                         },
                         {
                           "resourceName": "${RESOURCE_NAME2}",
                           "interface": "${SRIOV-NIC-NAME2}",
                           "pfname": "${PF_NAME2}",
                           "driver": "${DRIVER_NAME2}",
                           "numVFsToCreate": ${NUM_VFS2}
                         }
                      ]
              mode: 0644
              user:
                name: root
              group:
                name: root
            - path: /var/lib/rancher/rke2/server/manifests/sriov-crd.yaml
```

```
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: sriov-crd
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/sriov-crd
                    targetNamespace: sriov-network-operator
                    version: 303.0.2+up1.5.0
                    createNamespace: true
            - path: /var/lib/rancher/rke2/server/manifests/sriov-network-
operator.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: sriov-network-operator
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/sriov-network-
operator
                    targetNamespace: sriov-network-operator
                    version: 303.0.2+up1.5.0
                    createNamespace: true
        kernel_arguments:
          should_exist:
            - intel_iommu=on
            - iommu=pt
            - idle=poll
            - mce=off
            - hugepagesz=1G hugepages=40
            - hugepagesz=2M hugepages=0
            - default_hugepagesz=1G
```

```
            - irqaffinity=${NON-ISOLATED_CPU_CORES}
            - isolcpus=domain,nohz,managed_irq,${ISOLATED_CPU_CORES}
            - nohz_full=${ISOLATED_CPU_CORES}
            - rcu_nocbs=${ISOLATED_CPU_CORES}
            - rcu_nocb_poll
            - nosoftlockup
            - nowatchdog
            - nohz=on
            - nmi_watchdog=0
            - skew_tick=1
            - quiet
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
            - name: cpu-partitioning.service
              enabled: true
              contents: |
                [Unit]
                Description=cpu-partitioning
                Wants=network-online.target
```

```
                After=network.target network-online.target
                [Service]
                Type=oneshot
                User=root
                ExecStart=/bin/sh -c "echo isolated_cores=${ISOLATED_CPU_CORES}
 > /etc/tuned/cpu-partitioning-variables.conf"
                ExecStartPost=/bin/sh -c "tuned-adm profile cpu-partitioning"
                ExecStartPost=/bin/sh -c "systemctl enable tuned.service"
                [Install]
                WantedBy=multi-user.target
            - name: performance-settings.service
              enabled: true
              contents: |
                [Unit]
                Description=performance-settings
                Wants=network-online.target
                After=network.target network-online.target cpu-
partitioning.service
                [Service]
                Type=oneshot
                User=root
                ExecStart=/bin/sh -c "/opt/performance-settings/performance-
settings.sh"
                [Install]
                WantedBy=multi-user.target
            - name: sriov-custom-auto-vfs.service
              enabled: true
              contents: |
                [Unit]
                Description=SRIOV Custom Auto VF Creation
                Wants=network-online.target  rke2-server.target
                After=network.target network-online.target rke2-server.target
                [Service]
                User=root
                Type=forking
                TimeoutStartSec=900
```

```
                ExecStart=/bin/sh -c "while ! /var/lib/rancher/rke2/bin/kubectl
 --kubeconfig=/etc/rancher/rke2/rke2.yaml wait --for condition=ready nodes --
all ; do sleep 2 ; done"
                ExecStartPost=/bin/sh -c "while [ $(/var/lib/rancher/
rke2/bin/kubectl --kubeconfig=/etc/rancher/rke2/rke2.yaml get
 sriovnetworknodestates.sriovnetwork.openshift.io --ignore-not-found --no-
headers -A | wc -l) -eq 0 ]; do sleep 1; done"
                ExecStartPost=/bin/sh -c "/opt/sriov/sriov-auto-filler.sh"
                RemainAfterExit=yes
                KillMode=process
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

通过合并上述块创建该文件后，必须在管理群集中执行以下命令才能开始使用电信功能置备新的下游群集：

```
$ kubectl apply -f capi-provisioning-example.yaml
```

42.8 专用注册表

可以配置专用注册表作为工作负载所使用映像的镜像。

为此，我们可以创建机器，并在其中包含有关下游群集要使用的专用注册表的信息。

```
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-cert
  namespace: default
data:
  tls.crt: ${TLS_CERTIFICATE}
  tls.key: ${TLS_KEY}
  ca.crt: ${CA_CERTIFICATE}
type: kubernetes.io/tls
```

```
---
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-auth
  namespace: default
data:
  username: ${REGISTRY_USERNAME}
  password: ${REGISTRY_PASSWORD}
```

`tls.crt`、`tls.key` 和 `ca.crt` 是用于对专用注册表进行身份验证的证书。`username` 和 `password` 是用于对专用注册表进行身份验证的身份凭证。

注意

在机密中使用 `tls.crt`、`tls.key`、`ca.crt`、`username` 和 `password` 之前，必须对它们进行 base64 编码。

做出上述所有更改后，`capi-provisioning-example.yaml` 中的 `RKE2ControlPlane` 块如下所示：

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  privateRegistriesConfig:
```

```
    mirrors:
      "registry.example.com":
        endpoint:
          - "https://registry.example.com:5000"
    configs:
      "registry.example.com":
        authSecret:
          apiVersion: v1
          kind: Secret
          namespace: default
          name: private-registry-auth
        tls:
          tlsConfigSecret:
            apiVersion: v1
            kind: Secret
            namespace: default
            name: private-registry-cert
  serverConfig:
    cni: calico
    cniMultusEnable: true
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
```

```
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

其中，`registry.example.com` 是下游群集使用的专用注册表的示例名称，应将其替换为实际值。

42.9 在隔离场景中置备下游群集

定向网络置备工作流程允许在隔离场景中自动置备下游群集。

42.9.1 隔离场景的要求

1. 使用 EIB 生成的原始映像必须包含用于在隔离场景中运行下游群集的特定容器映像（helm-chart OCI 和容器映像）。有关详细信息，请参见第 42.3 节 “为隔离场景准备下游群集映像”。
2. 如果使用 SR-IOV 或任何其他自定义工作负载，则必须按照有关预加载专用注册表的一节（第 42.3.2.7 节 “预加载包含隔离场景和 SR-IOV 所需映像的专用注册表（可选）”）所述，在专用注册表中预加载用于运行工作负载的映像。

42.9.2 在隔离场景中登记裸机主机

在管理群集中登记裸机主机的过程与上前面的章节（第 42.4 节 “使用定向网络置备来置备下游群集（单节点）” [509]）中所述的过程相同。

42.9.3 在隔离场景中置备下游群集

需要进行一些重大更改才能在隔离场景中置备下游群集：

1. `capi-provisioning-example.yaml` 文件中的 `RKE2ControlPlane` 必须包含 `spec.agentConfig.airGapped: true` 指令。
2. 必须按照有关专用注册表的一节（第 42.8 节 “专用注册表”）所述，将专用注册表配置包含在 `capi-provisioning-airgap-example.yaml` 文件的 `RKE2ControlPlane` 块中。
3. 如果您使用的是 SR-IOV 或任何其他需要安装 helm-chart 的 `AdditionalUserData` 配置（combustion 脚本），则必须修改配置内容以引用专用注册表，而不是使用公共注册表。

以下示例显示了 `capi-provisioning-airgap-example.yaml` 文件的 `AdditionalUserData` 块中的 SR-IOV 配置，以及为了引用专用注册表而需进行的修改。

- 专用注册表机密引用
- 在 Helm-Chart 定义中使用专用注册表而不是公共 OCI 映像。

```
# secret to include the private registry certificates
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-cert
  namespace: default
data:
  tls.crt: ${TLS_BASE64_CERT}
  tls.key: ${TLS_BASE64_KEY}
  ca.crt: ${CA_BASE64_CERT}
type: kubernetes.io/tls
---
# secret to include the private registry auth credentials
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-auth
```

```
  namespace: default
data:
  username: ${REGISTRY_USERNAME}
  password: ${REGISTRY_PASSWORD}
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  privateRegistriesConfig:       # Private registry configuration to add your
 own mirror and credentials
    mirrors:
      docker.io:
        endpoint:
          - "https://$(PRIVATE_REGISTRY_URL)"
    configs:
      "192.168.100.22:5000":
        authSecret:
          apiVersion: v1
          kind: Secret
          namespace: default
          name: private-registry-auth
        tls:
          tlsConfigSecret:
            apiVersion: v1
            kind: Secret
```

```
            namespace: default
            name: private-registry-cert
          insecureSkipVerify: false
  serverConfig:
    cni: calico
    cniMultusEnable: true
  preRKE2Commands:
    - modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
  agentConfig:
    airGapped: true       # Airgap true to enable airgap mode
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        storage:
          files:
            - path: /var/lib/rancher/rke2/server/manifests/configmap-sriov-
custom-auto.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: v1
                  kind: ConfigMap
                  metadata:
                    name: sriov-custom-auto-config
                    namespace: sriov-network-operator
                  data:
                    config.json: |
                      [
                         {
                           "resourceName": "${RESOURCE_NAME1}",
                           "interface": "${SRIOV-NIC-NAME1}",
                           "pfname": "${PF_NAME1}",
                           "driver": "${DRIVER_NAME1}",
                           "numVFsToCreate": ${NUM_VFS1}
                         },
                         {
```

```
                    "resourceName": "${RESOURCE_NAME2}",
                    "interface": "${SRIOV-NIC-NAME2}",
                    "pfname": "${PF_NAME2}",
                    "driver": "${DRIVER_NAME2}",
                    "numVFsToCreate": ${NUM_VFS2}
                  }
                ]
      mode: 0644
      user:
        name: root
      group:
        name: root
    - path: /var/lib/rancher/rke2/server/manifests/sriov.yaml
      overwrite: true
      contents:
        inline: |
          apiVersion: v1
          data:
            .dockerconfigjson: ${REGISTRY_AUTH_DOCKERCONFIGJSON}
          kind: Secret
          metadata:
            name: privregauth
            namespace: kube-system
          type: kubernetes.io/dockerconfigjson
          ---
          apiVersion: v1
          kind: ConfigMap
          metadata:
            namespace: kube-system
            name: example-repo-ca
          data:
            ca.crt: |-
              -----BEGIN CERTIFICATE-----
              ${CA_BASE64_CERT}
              -----END CERTIFICATE-----
          ---
          apiVersion: helm.cattle.io/v1
          kind: HelmChart
```

```
metadata:
  name: sriov-crd
  namespace: kube-system
spec:
  chart: oci://${PRIVATE_REGISTRY_URL}/sriov-crd
  dockerRegistrySecret:
    name: privregauth
  repoCAConfigMap:
    name: example-repo-ca
  createNamespace: true
  set:
    global.clusterCIDR: 192.168.0.0/18
    global.clusterCIDRv4: 192.168.0.0/18
    global.clusterDNS: 10.96.0.10
    global.clusterDomain: cluster.local
    global.rke2DataDir: /var/lib/rancher/rke2
    global.serviceCIDR: 10.96.0.0/12
  targetNamespace: sriov-network-operator
  version: 303.0.2+up1.5.0
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: sriov-network-operator
  namespace: kube-system
spec:
  chart: oci://${PRIVATE_REGISTRY_URL}/sriov-network-operator
  dockerRegistrySecret:
    name: privregauth
  repoCAConfigMap:
    name: example-repo-ca
  createNamespace: true
  set:
    global.clusterCIDR: 192.168.0.0/18
    global.clusterCIDRv4: 192.168.0.0/18
    global.clusterDNS: 10.96.0.10
    global.clusterDomain: cluster.local
    global.rke2DataDir: /var/lib/rancher/rke2
```

```
                  global.serviceCIDR: 10.96.0.0/12
                targetNamespace: sriov-network-operator
                version: 303.0.2+up1.5.0
        mode: 0644
        user:
          name: root
        group:
          name: root
    kernel_arguments:
      should_exist:
        - intel_iommu=on
        - iommu=pt
        - idle=poll
        - mce=off
        - hugepagesz=1G hugepages=40
        - hugepagesz=2M hugepages=0
        - default_hugepagesz=1G
        - irqaffinity=${NON-ISOLATED_CPU_CORES}
        - isolcpus=domain,nohz,managed_irq,${ISOLATED_CPU_CORES}
        - nohz_full=${ISOLATED_CPU_CORES}
        - rcu_nocbs=${ISOLATED_CPU_CORES}
        - rcu_nocb_poll
        - nosoftlockup
        - nowatchdog
        - nohz=on
        - nmi_watchdog=0
        - skew_tick=1
        - quiet
    systemd:
      units:
        - name: rke2-preinstall.service
          enabled: true
          contents: |
            [Unit]
            Description=rke2-preinstall
            Wants=network-online.target
            Before=rke2-install.service
            ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
```

```
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
            - name: cpu-partitioning.service
              enabled: true
              contents: |
                [Unit]
                Description=cpu-partitioning
                Wants=network-online.target
                After=network.target network-online.target
                [Service]
                Type=oneshot
                User=root
                ExecStart=/bin/sh -c "echo isolated_cores=${ISOLATED_CPU_CORES}
 > /etc/tuned/cpu-partitioning-variables.conf"
                ExecStartPost=/bin/sh -c "tuned-adm profile cpu-partitioning"
                ExecStartPost=/bin/sh -c "systemctl enable tuned.service"
                [Install]
                WantedBy=multi-user.target
            - name: performance-settings.service
              enabled: true
              contents: |
                [Unit]
                Description=performance-settings
                Wants=network-online.target
                After=network.target network-online.target cpu-
partitioning.service
                [Service]
                Type=oneshot
                User=root
```

```
                ExecStart=/bin/sh -c "/opt/performance-settings/performance-
settings.sh"
                [Install]
                WantedBy=multi-user.target
            - name: sriov-custom-auto-vfs.service
              enabled: true
              contents: |
                [Unit]
                Description=SRIOV Custom Auto VF Creation
                Wants=network-online.target  rke2-server.target
                After=network.target network-online.target rke2-server.target
                [Service]
                User=root
                Type=forking
                TimeoutStartSec=1800
                ExecStart=/bin/sh -c "while ! /var/lib/rancher/rke2/bin/kubectl
 --kubeconfig=/etc/rancher/rke2/rke2.yaml wait --for condition=ready nodes --
timeout=30m --all ; do sleep 10 ; done"
                ExecStartPost=/bin/sh -c "/opt/sriov/sriov-auto-filler.sh"
                RemainAfterExit=yes
                KillMode=process
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

43 生命周期操作

本节介绍通过 SUSE Edge for Telco 部署的群集的生命周期管理操作。

43.1 管理群集升级

管理群集的升级涉及到多个组件。有关需要升级的常规组件的列表，请参见 Day 2 管理群集（第 35 章 “管理群集”）文档。

下面介绍了针对此设置组件的升级过程。

升级 Metal³

要升级 Metal3，请使用以下命令更新 Helm 储存库缓存，并从 Helm chart 储存库提取用于安装 Metal3 的最新 chart：

```
helm repo update
helm fetch suse-edge/metal3
```

然后，最简单的升级方式是将当前配置导出到某个文件，然后使用该文件升级 Metal3 版本。如果需要对新版本进行任何更改，可以先编辑该文件，然后再升级。

```
helm get values metal3 -n metal3-system -o yaml > metal3-values.yaml
helm upgrade metal3 suse-edge/metal3 \
  --namespace metal3-system \
  -f metal3-values.yaml \
  --version=303.0.7+up0.11.5
```

43.2 下游群集升级

升级下游群集涉及到更新多个组件。以下章节介绍了每个组件的升级过程。

升级操作系统

对于此过程，请参考此章节(第 42.2 节 “为联网场景准备下游群集映像”)来构建包含新操作系统版本的新映像。使用 EIB 生成此新映像后，下一置备阶段将使用提供的新操作系统版本。下一步骤将使用新映像来升级节点。

升级 RKE2 群集

需要做出以下更改才能使用自动化工作流程升级 RKE2 群集：

- 按照第 42.4 节 “使用定向网络置备来置备下游群集（单节点）” [511]中所述更改 `capi-provisioning-example.yaml` 中的 `RKE2ControlPlane` 块：
  - 指定所需的 `rolloutStrategy`。
  - 将 RKE2 群集的版本更改为新版本（请替换以下代码中的 `${RKE2_NEW_VERSION}`）。

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta1
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  version: ${RKE2_NEW_VERSION}
  replicas: 1
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  serverConfig:
    cni: cilium
  rolloutStrategy:
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
```

```
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                ExecStart=/bin/sh -c "sed -i \"s/BAREMETALHOST_UUID/$(jq -
r .uuid /mnt/openstack/latest/meta_data.json)/\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
        - provider-id=metal3://BAREMETALHOST_UUID
    nodeName: "localhost.localdomain"
```

- 按照第 42.4 节 “使用定向网络置备来置备下游群集（单节点）” [511]中所述更改 `capi-provisioning-example.yaml` 中的 `Metal3MachineTemplate` 块：
  - 将映像名称与校验和更改为在上一步骤中生成的新版本。
  - 将指令 `nodeReuse` 设置为 `true`，以避免创建新节点。
  - 将指令 `automatedCleaningMode` 设置为 `metadata`，以启用节点自动清理。

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
kind: Metal3MachineTemplate
metadata:
```

```
  name: single-node-cluster-controlplane
  namespace: default
spec:
  nodeReuse: True
  template:
    spec:
      automatedCleaningMode: metadata
      dataTemplate:
        name: single-node-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/${NEW_IMAGE_GENERATED}.sha256
        checksumType: sha256
        format: raw
        url: http://imagecache.local:8080/${NEW_IMAGE_GENERATED}.raw
```

完成这些更改后，可以使用以下命令将 `capi-provisioning-example.yaml` 文件应用于群集：

```
kubectl apply -f capi-provisioning-example.yaml
```

VIII 查错

本节提供诊断和解决常见 SUSE Edge 部署及操作问题的指导。内容涵盖多个主题，包括针对特定组件的查错步骤、关键工具及相关日志位置。

44 通用查错原则

在深入了解特定组件问题之前，请考虑以下通用原则：

- **检查日志**：日志是主要的信息来源。大多数情况下，错误消息会自行说明问题，且包含失败原因的提示。
- **检查时钟**：系统间的时钟差异可能导致各种错误。确保时钟同步。可通过 EIB 配置在引导时强制时钟同步，请参见“配置操作系统时间”（第 3 章 “使用 Edge Image Builder 配置独立群集”）。
- **引导问题**：如果系统在引导过程中陷入停滞状态，则记录显示的最后一条消息。可访问控制台（物理连接或通过 BMC）查看引导消息。
- **网络问题**：验证网络接口配置 (`ip a`)、路由表 (`ip route`)，测试与其他节点及外部服务的连通性（`ping`、`nc`）。确保防火墙规则未封锁必要端口。
- **验证组件状态**：使用 `kubectl get` 和 `kubectl describe` 查看 Kubernetes 资源。使用 `kubectl get events --sort-by='.lastTimestamp' -n <namespace>` 查看特定 Kubernetes 名称空间的事件。
- **验证服务状态**：使用 `systemctl status <service>` 检查 systemd 服务状态。
- **检查语法**：软件对配置文件的结构和语法有特定要求。例如，对于 YAML 文件，可使用 `yamllint` 或类似工具验证语法正确性。
- **隔离问题**：尝试将问题缩小到特定组件或层级（例如，网络、存储、操作系统、Kubernetes、Metal^3、Ironic 等）。
- **文档参考**：始终参考官方 SUSE Edge 文档 (https://documentation.suse.com/suse-edge/) 及上游文档以获取详细信息。
- **版本**：SUSE Edge 是经过精心设计且全面测试的 SUSE 各组件版本集合。每个 SUSE Edge 版本中各组件的版本信息可在 SUSE Edge 支持矩阵 (https://documentation.suse.com/suse-edge/support-matrix/html/support-matrix/index.html) 中找到。
- **已知问题**：每个 SUSE Edge 版本的发行说明中都含有“已知问题”部分，包含有关将在未来版本中修复但可能影响当前版本的问题的信息。

45 Kiwi 查错

Kiwi 用于生成更新的 SUSE Linux Micro 映像，供 Edge Image Builder 使用。

常见问题

- **SL Micro 版本不匹配**：构建主机的操作系统版本必须与待构建的操作系统版本匹配（例如，SL Micro 6.0 主机 → SL Micro 6.0 映像）。
- **SELinux 处于强制模式**：由于存在某些限制，目前需要临时禁用 SELinux 才能使用 Kiwi 构建映像。请使用 `getenforce` 检查 SELinux 状态，并在运行构建过程前使用 `setenforce 0` 将其禁用。
- **构建主机未注册**：构建过程会使用构建主机订阅从 SUSE SCC 提取软件包。如果主机未注册，构建将会失败。
- **循环设备测试失败**：首次执行 Kiwi 构建过程时，启动后不久就会失败，并显示“ERROR: Early loop device test failed, please retry the container run.”，这是由于底层主机系统上正在创建的循环设备无法立即在容器映像内可见所致。重新运行该 Kiwi 构建过程，通常即可顺利进行。
- **权限缺失**：构建过程需要以 root 用户（或通过 sudo）运行。
- **权限错误**：运行容器时，应为构建过程指定 `--privileged` 标志。请仔细检查是否指定了该标志。

日志

- **构建容器日志**：检查构建容器的日志。生成的日志存放在用于存储制品的目录中。也可通过 docker logs 或 podman logs 获取必要信息。
- **临时构建目录**：Kiwi 会在构建过程中创建临时目录。如果主要输出的信息不充足，可在这些目录中查看中间日志或制品。

查错步骤

1. **查看 `build-image` 输出**：控制台输出中的错误消息通常具有明确指示性。
2. **检查构建环境**：确保运行 Kiwi 的计算机满足 Kiwi 本身的所有先决条件（例如，docker/podman、SELinux、充足的磁盘空间）。

3. **检查构建容器日志**：查看失败容器的日志以获取更详细的错误信息（见上文）。
4. **验证定义文件**：如果使用自定义 Kiwi 映像定义文件，请仔细检查文件是否有拼写错误或语法问题。

注意

请参考 Kiwi 查错指南 (https://documentation.suse.com/appliance/kiwi-9/html/kiwi/troubleshooting.html)。

46 Edge Image Builder (EIB) 查错

EIB 用于创建自定义 SUSE Edge 映像。

常见问题

- **错误的 SCC 代码**：确保 EIB 定义文件中使用的 SCC 代码与 SL Micro 版本和体系结构相匹配。
- **依赖项缺失**：确保构建环境中没有缺失的软件包或工具。
- **映像大小不正确**：对于原始映像，必须指定 `diskSize` 参数，其值很大程度上取决于映像中包含的映像、RPM 及其他制品。
- **权限**：如果在 custom/files 目录中存储脚本，则确保脚本具有可执行权限，因为这些文件仅在 Combustion 阶段可用，EIB 不会对其进行修改。
- **操作系统组依赖项**：创建需要设置自定义用户和组的映像时，应明确预先创建作为“`primaryGroup`”的组。
- **操作系统用户的 SSH 密钥需要主目录**：创建需要设置带 SSH 密钥的用户的映像时，还需使用 `createHomeDir=true` 创建主文件夹。
- **Combustion 问题**：EIB 依赖 Combustion 来自定义操作系统并部署其他所有 SUSE Edge 组件，包括存放在 custom/scripts 文件夹中的自定义脚本。需要注意的是，Combustion 过程在 `initrd` 阶段执行，因此脚本执行时系统尚未完全引导。
- **Podman 计算机大小**：如第 IV 部分 “提示和技巧”中所述，在非 Linux 操作系统上，需要验证 Podman 计算机是否有充足的 CPU/内存来运行 EIB 容器。

日志

- **EIB 输出**：`eib build` 命令的控制台输出可提供极为重要的信息。
- **构建容器日志**：检查构建容器的日志。生成的日志存放在用于存储制品的目录中。也可通过 `docker logs` 或 `podman logs` 获取必要信息。

注意

有关详细信息，请参见调试 (https://github.com/suse-edge/edge-image-builder/blob/main/docs/debugging.md)。

- **临时构建目录**：EIB 会在构建过程中创建临时目录。如果主要输出的信息不充足，可在这些目录中查看中间日志或制品。
- **Combustion 日志**：如果使用 EIB 构建的映像因任何原因无法引导，可以使用 root 外壳。连接到主机控制台（物理连接或通过 BMC 等方式），使用 `journalctl -u combustion` 检查 Combustion 日志，或使用 `journalctl` 查看所有操作系统日志，确定失败的根本原因。

查错步骤

1. **查看 `eib-build` 输出**：控制台输出中的错误消息通常具有明确指示性。
2. **检查构建环境**：确保运行 EIB 的计算机满足 EIB 本身的所有先决条件（例如，docker/podman、充足的磁盘空间）。
3. **检查构建容器日志**：查看失败容器的日志以获取更详细的错误信息（见上文）。
4. **验证 `eib` 配置**：仔细检查 `eib` 配置文件是否有拼写错误，或者其中的源文件/构建脚本路径是否不正确。
    - **单独测试组件**：如果 EIB 构建涉及自定义脚本或阶段，单独运行它们以隔离故障。

注意

请参见 Edge Image Builder 调试 (https://github.com/suse-edge/edge-image-builder/blob/main/docs/debugging.md)。

47 Edge 网络 (NMC) 查错

SL Micro EIB 映像中注入了 NMC，后者用于在系统引导时通过 Combustion 配置 Edge 主机的网络。在 Metal3 工作流程中，它也会作为检查过程的一部分执行。当主机首次引导时或在 Metal3 检查过程中，可能会发生问题。

常见问题

- **主机首次引导时无法正常引导**：格式错误的网络定义文件可能导致 Combustion 阶段失败，进而使主机回退到 root 外壳。
- **未正确生成文件**：确保网络文件符合 NMState (https://nmstate.io/examples.html) 格式。
- **网络接口配置不正确**：确保 MAC 地址与主机上使用的接口匹配。
- **接口名称不匹配**：内核参数 `net.ifnames=1` 会启用网络接口的可预测命名方案 (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html)，因此不再使用 `eth0` 命名方案，而是采用 `enp2s0` 等其他命名方案。

日志

- **Combustion 日志**：由于 nmc 在 Combustion 阶段使用，可在待置备的主机上使用 `journalctl -u combustion` 检查 Combustion 日志。
- **NetworkManager 日志**：在 Metal^3 部署工作流程中，nmc 是 IPA 执行过程的一个环节，它通过 systemd 的 ExecStartPre 功能作为 NetworkManager 服务的依赖项运行。可以在 IPA 主机上使用 `journalctl -u NetworkManager` 检查 NetworkManager 日志（参见第 49 章 “定向网络置备查错”，了解如何在主机通过 IPA 引导时访问主机）。

查错步骤

1. **验证 yaml 语法**：nmc 配置文件是 yaml 文件，可使用 `yamllint` 或类似工具检查其语法正确性。
2. **手动运行 nmc**：由于 nmc 是 EIB 容器的一部分，可使用本地 Podman 命令来调试问题。
    a. 创建一个用于存储 nmc 文件的临时文件夹。

```
mkdir -p ${HOME}/tmp/foo
```

b. 将 nmc 文件保存到该位置。

```
❯ tree --noreport ${HOME}/tmp/foo
/Users/johndoe/tmp/foo
├── host1.example.com.yaml
└── host2.example.com.yaml
```

c. 以 nmc 为入口点执行 generate 命令来运行 EIB 容器，模拟 nmc 在 combustion 阶段执行的任务：

```
podman run -it --rm -v ${HOME}/tmp/foo:/tmp/foo:Z --entrypoint=/usr/
bin/nmc registry.suse.com/edge/3.3/edge-image-builder:1.2.0 generate
 --config-dir /tmp/foo --output-dir /tmp/foo/

[2025-06-04T11:58:37Z INFO  nmc::generate_conf] Generating config from
 "/tmp/foo/host2.example.com.yaml"...
[2025-06-04T11:58:37Z INFO  nmc::generate_conf] Generating config from
 "/tmp/foo/host1.example.com.yaml"...
[2025-06-04T11:58:37Z INFO  nmc] Successfully generated and stored
 network config
```

d. 观察临时文件夹中生成的日志和文件。

48 自主回连场景查错

自主回连场景涉及使用 Elemental 回连管理群集，以及使用 EIB 创建包含 elemental-registration 组件的操作系统映像。当主机首次引导时、在 EIB 构建过程中或尝试注册到管理群集时，可能会发生问题。

常见问题

- **系统注册失败**：节点未在 UI 中注册。确保主机正常引导、能够与 Rancher 通讯、时钟保持同步，并且 Elemental 服务正常工作。
- **系统置备失败**：节点已注册但置备失败。确保主机能够与 Rancher 通讯、时钟保持同步，并且 Elemental 服务正常工作。

日志

- **系统日志**：`journalctl`
- **Elemental-system-agent 日志**：`journalctl -u elemental-system-agent`
- **K3s/RKE2 日志**：`journalctl -u k3s` 或 `journalctl -u rke2-server`（或 `rke2-agent`）
- **Elemental Operator Pod**：`kubectl logs -n cattle-elemental-system -l app=elemental-operator`

查错步骤

1. **查看日志**：检查 Elemental Operator Pod 日志以确定是否存在任何问题。如果节点已引导，则检查主机日志。
2. **检查 MachineRegistration 和 TPM**：默认会使用 TPM 进行身份验证 (https://elemental.docs.rancher.com/authentication/)，但对于没有 TPM 的主机，也有替代方案。

49 定向网络置备查错

定向网络置备场景涉及使用 Metal^3 和 CAPI 组件置备下游群集，以及使用 EIB 创建操作系统映像。当主机首次引导时，或者在检查或置备过程中，可能会发生问题。

常见问题

- **固件过旧**：验证物理主机上使用的所有固件是否为最新版本，包括 BMC 固件（有时 Metal^3 需要特定版本/更新的固件 (https://book.metal3.io/bmo/supported_hardware#redfish-and-its-variants)）。
- **置备因 SSL 错误失败**：如果提供映像的 Web 服务器使用 https，需要配置 Metal^3 以在 IPA 映像中注入并信任证书。请参见第 40.3.4 节 “Kubernetes 文件夹”了解如何将 `ca-additional.crt` 文件包含到 Metal^3 chart 中。
- **使用 IPA 引导主机时的证书问题**：某些服务器供应商在将虚拟媒体 ISO 映像挂接到 BMC 时会验证 SSL 连接，这可能导致出现问题，因为针对 Metal3 部署生成的证书是自我签名证书。在这种情况下，主机在引导时可能会回退到 UEFI 外壳。请参见第 1.6.2 节 “对虚拟媒体 ISO 挂接禁用 TLS”了解如何解决此问题。
- **名称或标签引用错误**：如果群集通过错误的名称或标签引用节点，群集可能显示为已部署，但 BMH 仍处于“可用”状态。请仔细检查 BMH 关联对象的引用。
- **BMC 通讯问题**：确保管理群集上运行的 Metal^3 Pod 能够访问待置备主机的 BMC（通常 BMC 网络的限制极为严格）。
- **裸机主机状态不正确**：BMH 对象在其生命周期（状态机流转过程 (https://book.metal3.io/bmo/state_machine)）中会经历不同状态（正在检查、正在准备、已置备等）。如果检测到状态不正确，请使用以下命令检查 BMH 对象的 `status` 字段以获取更多信息：`kubectl get bmh <name> -o jsonpath='{.status}'| jq`。
- **主机无法取消置备**：如果主机取消置备失败，可尝试为 BMH 对象添加“detached”注解后再尝试去除，具体命令为 `kubectl annotate bmh/<BMH> baremetalhost.metal3.io/detached=""`。
- **映像错误**：验证通过 EIB 为下游群集构建的映像是否可用、是否具有正确的校验和，且其大小不会导致解压失败或超出磁盘容量。

- **磁盘大小不匹配**：默认情况下，磁盘不会自动扩容至其最大容量。如第 1.3.4.1.2 节“Growfs 脚本”所述，通过 EIB 为下游群集主机构建映像时，需要包含一个 growfs 脚本。

- **清理过程卡住**：清理过程会重试多次。如果由于主机问题导致清理过程无法继续，需要先将 BMH 对象的 `automatedCleanMode` 字段设置为 `disabled`，以禁用清理功能。

警告

当清理过程耗时过长或失败时，不建议手动去除终结器。这样做会从 Kubernetes 中去除主机记录，但会在 Ironic 中残留该记录。后台当前运行的操作会继续，再次添加主机时可能会因发生冲突而失败。

- **Metal3/Rancher Turtles/CAPI Pod 问题**：所有必需组件的部署流程如下：
  - Rancher Turtles 控制器部署 CAPI 操作器控制器。
  - CAPI 操作器控制器随后部署提供程序控制器（CAPI 核心、CAPM3 和 RKE2 控制平面/引导组件）。

验证所有 Pod 是否正常运行，若非如此，请检查日志。

日志

- **Metal^3 日志**：检查不同 Pod 的日志。

```
kubectl logs -n metal3-system -l app.kubernetes.io/component=baremetal-
operator
kubectl logs -n metal3-system -l app.kubernetes.io/component=ironic
```

注意

metal3-ironic Pod 至少包含 4 个不同的容器（`ironic-httpd`、`ironic-log-watch`、`ironic` 和 `ironic-ipa-downloader` (init)）。带 `-c` 标志运行 `kubectl logs` 可校验其中每个容器的日志。

注意

`ironic-log-watch` 容器会在检查/置备主机后从主机公开控制台日志（前提是网络连接允许将这些日志发送回管理群集）。当发生置备错误，但您无法直接访问 BMC 控制台日志时，此功能非常有用。

- **Rancher Turtles 日志**：检查不同 Pod 的日志。

```
kubectl logs -n rancher-turtles-system -l control-plane=controller-manager
kubectl logs -n rancher-turtles-system -l app.kubernetes.io/name=cluster-
api-operator
kubectl logs -n rke2-bootstrap-system -l cluster.x-k8s.io/
provider=bootstrap-rke2
kubectl logs -n rke2-control-plane-system -l cluster.x-k8s.io/
provider=control-plane-rke2
kubectl logs -n capi-system -l cluster.x-k8s.io/provider=cluster-api
kubectl logs -n capm3-system -l cluster.x-k8s.io/provider=infrastructure-
metal3
```

- **BMC 日志**：BMC 通常会提供一个可用于执行大多数交互操作的 UI，其中一般都会有一个“日志”部分，您可在其中查看潜在问题（无法访问映像、硬件故障等）。
- **控制台日志**：连接到 BMC 控制台（通过 BMC Web UI、串行接口等），检查实时写入的日志中的错误。

查错步骤

1. **检查 `BareMetalHost` 状态**：
    - 运行 `kubectl get bmh -A` 可显示当前状态。查找 `provisioning`、`ready`、`error`、`registering`。
    - 运行 `kubectl describe bmh -n <namespace> <bmh_name>` 可提供详细事件和状态，解释 BMH 可能出现卡死状态的原因。
2. **测试 RedFish 连通性**：

- 从 Metal^3 控制平面使用 `curl` 测试能否通过 redfish 连接到 BMC。
- 确保在 `BareMetalHost-Secret` 定义中提供了正确的 BMC 身份凭证。

3. **验证 Turtles/CAPI/Metal3 Pod 状态**：分别使用以下命令确保管理群集上的容器处于正常运行状态：`kubectl get pods -n metal3-system` 和 `kubectl get pods -n rancher-turtles-system`（另请参见 `capi-system`、`capm3-system`、`rke2-bootstrap-system` 和 `rke2-control-plane-system`）。

4. **验证待置备的主机可访问 Ironic 端点**：待置备的主机需要能够访问 Ironic 端点以向 Metal^3 回传状态信息。请使用 `kubectl get svc -n metal3-system metal3-metal3-ironic` 获取 IP，然后通过 `curl/nc` 尝试访问该端点。

5. **验证 BMC 可访问 IPA 映像**：IPA 由 Ironic 端点提供，BMC 必须能够访问 IPA，因为后者用作虚拟 CD。

6. **验证待置备的主机可访问操作系统映像**：运行 IPA 时，待置备主机自身必须能够访问用于置备该主机的映像，因为该映像会被临时下载并写入磁盘。

7. **检查 Metal^3 组件日志**：请参见上文。

8. **重新触发 BMH 检查**：如果检查失败或可用主机的硬件发生变化，可通过为 BMH 对象添加注解 `inspect.metal3.io: ""` 触发新的检查过程。有关详细信息，请参见 Metal^3 控制检查 (https://book.metal3.io/bmo/inspect_annotation) 指南。

9. **裸机 IPA 控制台**：您可以通过以下几种方法来排查 IPA 问题：

    - 启用“自动登录”：这样 root 用户在连接到 IPA 控制台时便会自动登录。

警告

此功能仅用于调试目的，因为这会授予对主机的完全访问权限。

要启用自动登录功能，应将 Metal3 helm 的 `global.ironicKernelParams` 设置为 `console=ttyS0 suse.autologin=ttyS0`（根据控制台的不同，可将 `ttyS0` 替换为其他值）。然后需要重新部署 Metal[3] chart。（注意：`ttyS0` 是示例，具体的值应与实际终端匹配。例如在裸机上，该值通常为 `tty1`，可通过查看主机引导时 IPA 内存盘的控制台输出来确定，因为其中的 `/etc/issue` 会列显控制台名称。）
另一种方法是更改 `metal3-system` 名称空间内 `ironic-bmo` configmap 中的 `IRONIC_KERNEL_PARAMS` 参数。通过 `kubectl` edit 进行操作会更简单，但更新 chart 时，该参数会被覆盖。之后需要使用 `kubectl delete pod -n metal3-system -l app.kubernetes.io/component=ironic` 重启 Metal[3] Pod。

- 为 IPA 的 root 用户注入 SSH 密钥。

警告

此功能仅用于调试目的，因为这会授予对主机的完全访问权限。

可通过 Metal[3] Helm 的 `debug.ironicRamdiskSshKey` 参数注入 root 用户的 SSH 密钥。然后需要重新部署 Metal[3] chart。
另一种方法是更改 `metal3-system` 名称空间内 `ironic-bmo configmap` 中的 `IRONIC_RAMDISK_SSH_KEY` 参数。通过 `kubectl` edit 进行操作会更简单，但更新 chart 时，该参数会被覆盖。之后需要使用 `kubectl delete pod -n metal3-system -l app.kubernetes.io/component=ironic` 重启 Metal[3] Pod。

注意

请参见 CAPI 查错 (https://cluster-api.sigs.k8s.io/user/troubleshooting) 指南和 Metal[3] 查错 (https://book.metal3.io/troubleshooting) 指南。

50 对其他组件查错

其他 SUSE Edge 组件的查错指南可在相应组件的官方文档中找到：

- SUSE Linux Micro 查错 (https://documentation.suse.com/smart/micro-clouds/html/SLE-Micro-5.5-admin/index.html#id-1.10)
- RKE2 已知问题 (https://docs.rke2.io/known_issues)
- K3s 已知问题 (https://docs.k3s.io/known-issues)
- Rancher 常规查错 (https://ranchermanager.docs.rancher.com/troubleshooting/general-troubleshooting)
- SUSE Multi-Linux Manager 查错 (https://documentation.suse.com/multi-linux-manager/5.1/en/docs/administration/troubleshooting/tshoot-intro.html)
- Elemental 支持 (https://elemental.docs.rancher.com/troubleshooting-support/)
- Rancher Turtles 查错 (https://turtles.docs.rancher.com/turtles/stable/en/troubleshooting/troubleshooting.html)
- Longhorn 查错 (https://longhorn.io/docs/1.8.1/troubleshoot/troubleshooting/)
- Neuvector 查错 (https://open-docs.neuvector.com/next/troubleshooting/troubleshooting/)
- Fleet 查错 (https://fleet.rancher.io/troubleshooting)

您还可以访问 SUSE 知识库 (https://www.suse.com/support/kb/)。

51 收集支持团队所需的诊断信息

联系 SUSE 支持团队时，提供全面的诊断信息至关重要。

需收集的基本信息

- **详细问题描述**：发生了什么、何时发生、操作过程、预期行为及实际行为。
- **重现步骤**：能否稳定重现问题？如果可以，请列出确切步骤。
- **组件版本**：SUSE Edge 版本、各组件版本（RKE2/K3、EIB、Metal^3、Elemental 等）。
- **相关日志**：
  - `journalctl` 输出（尽可能按服务过滤，或提供完整的引导日志）。
  - Kubernetes Pod 日志 (kubectl logs)。
  - Metal^3/Elemental 组件日志。
  - EIB 构建日志及其他日志。
- **系统信息**：
  - `uname -a`
  - `df -h`
  - `ip a`
  - `/etc/os-release`
- **配置文件**：Elemental、Metal^3、EIB 的相关配置文件，例如 Helm chart 值文件、configmap 等。
- **Kubernetes 信息**：节点、服务、部署等。
- **受影响的 Kubernetes 对象**：BMH、MachineRegistration 等。

收集方式

- **对于日志**：将命令输出重定向到文件（例如，使用 `journalctl -u k3s > k3s_logs.txt`）。

- **对于 Kubernetes 资源**：使用 `kubectl get <resource> -o yaml > <resource_name>.yaml` 获取详细的 YAML 定义。
- **对于系统信息**：收集上述命令的输出。
- **对于 SL Micro**：参考 SUSE Linux Micro 查错指南 (https://documentation.suse.com/sle-micro/5.5/html/SLE-Micro-all/cha-adm-support-slemicro.html) 文档，了解如何使用 `supportconfig` 收集支持团队所需的系统信息。
- **对于 RKE2/Rancher**：参考《The Rancher v2.x Linux log collector script》 (https://www.suse.com/support/kb/doc/?id=000020191) 一文，运行 Rancher v2.x Linux 日志收集器脚本。

联系支持团队： 有关如何联系 SUSE 支持团队的详细信息，请参见《How-to effectively work with SUSE Technical Support》 (https://www.suse.com/support/kb/doc/?id=000019452) 一文，以及 SUSE Technical Support Handbook (https://www.suse.com/support/handbook/) 页面中提供的支持手册。

IX 附录

52 发行说明

52.1 摘要

SUSE Edge 3.3 是紧密集成且经过全面验证的端到端解决方案，用于解决在边缘处部署基础架构和云原生应用程序时存在的独特挑战。其核心作用是提供一个有主见但高度灵活、高度可缩放且安全的平台，涵盖初始部署映像的构建、节点置备和初始配置、应用程序部署、可观测性和生命周期管理。

该解决方案的设计考虑到了客户的需求和期望千差万别，因此不存在“以一应百”的边缘平台。边缘部署促使我们解决并不断设想出一些极具挑战性的问题，包括大规模可伸缩性、网络受限情况下的可用性、物理空间限制、新的安全威胁和攻击途径、硬件体系结构和系统资源的差异、部署旧式基础架构和应用程序并与之连接的要求，以及使用寿命较长的客户解决方案。

SUSE Edge 完全基于一流的开源软件构建而成，这既符合我们 30 年来交付安全、稳定且经过认证的 SUSE Linux 平台的历史，也延续了我们通过 Rancher 产品组合提供高可可缩放且功能丰富的 Kubernetes 管理方案的经验。SUSE Edge 依托这些既有能力，提供能够满足众多市场领域需求的功能，包括零售、医疗、交通、物流、电信、智能制造业以及工业物联网等。

注意

SUSE Edge for Telco（以前称为“自适应电信基础架构平台”(ATIP)）是 SUSE Edge 的衍生产品（或下游产品），它经过进一步优化并包含更多组件，使平台能够解决电信使用场景的要求。除非明确说明，否则所有发行说明都适用于 SUSE Edge 3.3 和 SUSE Edge for Telco 3.3。

52.2 简介

除非明确指定和说明，否则下面的发行说明在所有体系结构中都是相同的，并且最新版本以及所有其他 SUSE 产品的发行说明始终在 https://www.suse.com/releasenotes 上在线提供。

说明项只会列出一次，但是，非常重要并且属于多个章节的说明项可能会在多处提到。发行说明通常只会列出两个后续版本之间发生的更改。本发行说明可能重复了旧产品版本的发行说明中的某些重要说明项。为了方便识别这些说明项，发行说明中会通过一条备注来指出这一点。

但是，重复的说明项只是出于便利而提供。因此，如果您跳过了一个或多个版本，另请查看所跳过版本的发行说明。如果您只阅读最新版本的发行说明，或许会漏掉可能会对系统行为产生影响的重要更改。SUSE Edge 版本定义为 x.y.z，其中“x”表示主要版本，“y”表示次要版本，“z”表示补丁版本（也称为“z-stream”）。SUSE Edge 产品生命周期基于某个次要版本（例如“3.3”）定义，但在其整个生命周期中会提供后续补丁更新，例如“3.3.1”。

注意

各个 SUSE Edge z-stream 版本紧密集成，并已作为版本受控的堆栈进行全面测试。将任何一个组件升级到其他版本（而不是上面列出的版本）都可能会导致系统停机。虽然可以在未经测试的配置中运行 Edge 群集，但我们不建议这样做，而且这样可能需要花费更长的时间来通过支持渠道获得问题的解决方法。

52.3 版本 3.3.1

发布日期：2025 年 6 月 13 日

摘要：SUSE Edge 3.3.1 是 SUSE Edge 3.3 系列版本中的第一个 z-stream 版本。

52.3.1 新功能

- 已更新到 Kubernetes 1.32.4 和 Rancher Prime 2.11.2 发行说明 (https://github.com/rancher/rancher/releases/tag/v2.11.2)
- 已更新到 SUSE Security (Neuvector) 5.4.4 发行说明 (https://open-docs.neuvector.com/releasenotes/5x#544-may-2025)
- 已更新到 Rancher Turtles 0.20.0 发行说明 (https://turtles.docs.rancher.com/turtles/stable/en/changelogs/changelogs/v0.20.0.html)

52.3.2 Bug 和安全修复

- 在某些情况下，如果使用的管理群集是从 Edge 3.2 升级到 3.3.0 的，那么通过 CAPI 进行的下游滚动升级可能会导致计算机处于“删除中”状态。此问题已通过 RKE2 CAPI 提供程序更新得到解决（上游 RKE2 提供程序问题 661 (https://github.com/rancher/cluster-api-provider-rke2/issues/661)）

- 在通过 nm-configurator 配置网络时，3.3.0 版本中某些通过 MAC 识别接口的配置无法生效。此问题已通过以下方式解决：将 NMC 更新至 0.3.3 版本并相应更新 EIB 和 Metal^3 IPA 下载器容器映像（上游 NM Configurator 问题 (https://github.com/suse-edge/nm-configurator/issues/163)）

- 对于 3.3.0 版本中长时间运行的 Metal^3 管理群集，证书过期可能导致 baremetal-operator 与 ironic 的连接失败，此时需要手动重启 Pod 来作为临时解决方法。此问题已通过更新 Metal^3 chart 解决（SUSE Edge chart 问题 (https://github.com/suse-edge/charts/issues/178)）

- 以前，Rancher UI 无法在应用程序目录中列出 OCI 注册表中的 SUSE Edge chart。此问题已通过将 Rancher 更新至 2.11.2 版本解决（Rancher 问题 (https://github.com/rancher/rancher/issues/48746)）

52.3.3 已知问题

警告

如果要部署新群集，请先按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述构建全新映像。现在，无论是要创建 AMD64/Intel 64 还是 AArch64 体系结构的群集（包括管理群集和下游群集），都必须先执行此步骤。

- 在 SUSE Linux Micro 6.1 中使用 `toolbox` 时，默认容器映像不包含先前 5.5 版本中的部分工具。临时解决方法是将 toolbox 配置为使用先前的 `suse/sle-micro/5.5/toolbox` 容器映像，具体配置选项可参考 `toolbox --help`。

- 在某些情况下，通过 CAPI 进行滚动升级可能会导致计算机处于“删除中”状态，此问题将在未来的更新中解决（上游 RKE2 提供程序问题 655 (https://github.com/rancher/cluster-api-provider-rke2/issues/655)）

- 由于提供了与 3.3.0 中所述 CVE-2025-1974 (https://nvd.nist.gov/vuln/detail/CVE-2025-1974) 相关的修复，SUSE Linux Micro 6.1 **必须**进行更新，以包含 `>=6.4.0-26-default` 或 `>=6.4.0-30-rt`（实时内核）版本的内核。如果未进行该更新，ingress-nginx Pod 将持续处于 `CrashLoopBackOff` 状态。要进行内核更新，可在主机上运行 `transactional-update` 以更新所有软件包，或运行 `transactional-update pkg update kernel-default`（或 kernel-rt）仅更新内核，随后重引导主机。如果您是在部署新群集，请按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”中的说明构建包含最新内核的全新映像。

- 已发现 Kubernetes 作业控制器存在一个 bug，在特定状况下可能导致 RKE2/K3s 节点处于 `NotReady` 状态（参见 RKE2 问题 #8357 (https://github.com/rancher/rke2/issues/8357)）。错误消息可能如下所示：

```
E0605 23:11:18.489721    »···1 job_controller.go:631] "Unhandled Error"
 err="syncing job: tracking status: adding uncounted pods to status: Operation
 cannot be fulfilled on jobs.batch \"helm-install-rke2-ingress-nginx\":
 StorageError: invalid object, Code: 4, Key: /registry/jobs/kube-system/helm-
install-rke2-ingress-nginx, ResourceVersion: 0, AdditionalErrorMsg: Precondition
 failed: UID in precondition: 0aa6a781-7757-4c61-881a-cb1a4e47802c, UID in
 object meta: 6a320146-16b8-4f83-88c5-fc8b5a59a581" logger="UnhandledError"
```

临时解决方法是使用 `crictl` 重启 `kube-controller-manager`，命令如下：

```
export CONTAINER_RUNTIME_ENDPOINT=unix:///run/k3s/containerd/containerd.sock
export KUBEMANAGER_POD=$(/var/lib/rancher/rke2/bin/crictl ps --label
 io.kubernetes.container.name=kube-controller-manager --quiet)
/var/lib/rancher/rke2/bin/crictl stop ${KUBEMANAGER_POD} && \
/var/lib/rancher/rke2/bin/crictl rm ${KUBEMANAGER_POD}
```

- 在 RKE2/K3s 1.31 和 1.32 版本中，由于存在与 `overlayfs` 相关的某些条件限制，用于存储 CNI 配置的 `/etc/cni` 目录可能无法向 `containerd` 触发文件写入通知（参见 RKE2 问题 #8356 (https://github.com/rancher/rke2/issues/8356)）。这会导致 RKE2/K3s 部署停滞在等待 CNI 启动的状态，且 RKE2/K3s 节点会保持 `NotReady` 状态。在节点级别，通过 `kubectl describe node <affected_node>` 可观察到该问题：

```
<200b><200b>Conditions:
  Type         »Status  LastHeartbeatTime             »·LastTransitionTime
      »·Reason                   »··Message
  ----         »------  -----------------             »·------------------
      »·------                   »··-------
  Ready        »False   Thu, 05 Jun 2025 17:41:28 +0000   Thu, 05 Jun 2025
 14:38:16 +0000   KubeletNotReady          »··container runtime network not
 ready: NetworkReady=false reason:NetworkPluginNotReady message:Network plugin
 returns error: cni plugin not initialized
```

临时解决方法是在 RKE2 启动前将 tmpfs 卷挂载到 `/etc/cni` 目录。这会避免使用 overlayfs（overlayfs 会导致 containerd 漏收通知），且每次节点重启及 Pod 初始化容器重新运行时，配置都会被重写。如果使用 EIB，可在 `custom/scripts` 目录中添加 `04-tmpfs-cni.sh` 脚本（如此处[https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md#custom]所述），脚本内容如下：

```
#!/bin/bash
mkdir -p /etc/cni
mount -t tmpfs -o mode=0700,size=5M tmpfs /etc/cni
echo "tmpfs /etc/cni tmpfs defaults,size=5M,mode=0700 0 0" >> /etc/fstab
```

52.3.4 组件版本

下表描述了构成 3.3.1 版本的各个组件，包括版本、Helm chart 版本（如果适用），以及可从中提取已发布的二进制格式制品的位置。有关用法和部署示例，请参见相关文档。

名称	版本	Helm Chart 版本	制品位置（URL/映像）

SUSE Linux Micro	6.1（最新）	不适用	SUSE Linux Micro 下载页面 (https://www.suse.com/download/sle-micro/) SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso (sha256 70b9be28f2d92bc3b228412e4fc2b SL-Micro.x86_64-6.1-Base-RT-SelfInstall-GM.install.iso (sha256 9ce83e4545d4b36c7c6a44f7841dc SL-Micro.x86_64-6.1-Base-GM.raw.xz (sha256 36e3efa55822113840dd76fdf6914e SL-Micro.x86_64-6.1-Base-RT-GM.raw.xz (sha256 2ee66735da3e1da107b4878e73ae
SUSE Multi-Linux Manager	5.0.3	不适用	SUSE Multi-Linux Manager 下载页面 (https://www.suse.com/download/suse-manager/)

K3s	**1.32.4**	不适用	上游 K3s 版本 (https://github.com/k3s-io/k3s/releases/tag/v1.32.4%2Bk3s1)
RKE2	**1.32.4**	不适用	上游 RKE2 版本 (https://github.com/rancher/rke2/releases/tag/v1.32.4%2Brke2r1)
SUSE Rancher Prime	**2.11.2**	2.11.2	Rancher Prime Helm 储存库 (https://charts.rancher.com/server-charts/prime/index.yaml) Rancher 2.11.1 容器映像 (https://github.com/rancher/rancher/releases/download/v2.11.1/rancher-images.txt)
SUSE Storage	1.8.1	106.2.0+up1.8.1	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml) registry.suse.com/rancher/mirrored-longhornio-csi-attacher:v4.8.1

			registry.suse.com/rancher/mirrored-longhornio-csi-provisioner:v5.2.0 registry.suse.com/rancher/mirrored-longhornio-csi-resizer:v1.13.2 registry.suse.com/rancher/mirrored-longhornio-csi-snapshotter:v8.2.0 registry.suse.com/rancher/mirrored-longhornio-csi-node-driver-registrar:v2.13.0 registry.suse.com/rancher/mirrored-longhornio-livenessprobe:v2.15.0 registry.suse.com/rancher/mirrored-longhornio-backing-image-manager:v1.8.1 registry.suse.com/rancher/mirrored-longhornio-longhorn-engine:v1.8.1

			registry.suse.com/rancher/mirrored-longhornio-longhorn-instance-manager:v1.8.1 registry.suse.com/rancher/mirrored-longhornio-longhorn-manager:v1.8.1 registry.suse.com/rancher/mirrored-longhornio-longhorn-share-manager:v1.8.1 registry.suse.com/rancher/mirrored-longhornio-longhorn-ui:v1.8.1 registry.suse.com/rancher/mirrored-longhornio-support-bundle-kit:v0.0.52 registry.suse.com/rancher/mirrored-longhornio-longhorn-cli:v1.8.1
SUSE Security	**5.4.4**	**106.0.1+up2.8.6**	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml) ↗

			registry.suse.com/rancher/neuvector-controller:5.4.4 **registry.suse.com/rancher/neuvector-enforcer:5.4.4** **registry.suse.com/rancher/neuvector-manager:5.4.4** **registry.suse.com/rancher/neuvector-compliance-config:1.0.5** registry.suse.com/rancher/neuvector-registry-adapter:0.1.6 registry.suse.com/rancher/neuvector-scanner:6 **registry.suse.com/rancher/neuvector-updater:0.0.3**
Rancher Turtles (CAPI)	0.20.0	303.0.4+up0.20.0	**registry.suse.com/edge/charts/rancher-turtles:303.0.3_up0.20.0** **registry.rancher.com/rancher/rancher/turtles:v0.20.0**

			registry.rancher.com/rancher/cluster-api-operator:v0.17.0 registry.rancher.com/rancher/cluster-api-metal3-controller:v1.9.3 registry.rancher.com/rancher/cluster-api-metal3-ipam-controller:v1.9.4 registry.suse.com/rancher/cluster-api-controller:v1.9.5 **registry.suse.com/rancher/cluster-api-provider-rke2-bootstrap:v0.16.1** **registry.suse.com/rancher/cluster-api-provider-rke2-controlplane:v0.16.1**
Rancher Turtles 隔离资源	**0.20.0**	**303.0.4+up0.20.0**	**registry.suse.com/edge/charts/rancher-turtles-airgap-resources:303.0.3_up0.20.0**
Metal[3]	**0.11.5**	**303.0.7+up0.11.5**	**registry.suse.com/edge/charts/metal3:303.0.7_up0.11.5**

			registry.suse.com/edge/3.3/baremetal-operator:0.9.1.1 registry.suse.com/edge/3.3/ironic:26.1.2.4 **registry.suse.com/edge/3.3/ironic-ipa-downloader:3.0.7** registry.suse.com/edge/mariadb:10.6.15.1
MetalLB	0.14.9	303.0.0+up0.14.9	registry.suse.com/edge/charts/metallb:303.0.0_up0.14.9 registry.suse.com/edge/3.3/metallb-controller:v0.14.8 registry.suse.com/edge/3.3/metallb-speaker:v0.14.8 registry.suse.com/edge/3.3/frr:8.4 registry.suse.com/edge/3.3/frr-k8s:v0.0.14
Elemental	1.6.8	1.6.8	registry.suse.com/rancher/elemental-operator-chart:1.6.8

			registry.suse.com/rancher/elemental-operator-crds-chart:1.6.8 registry.suse.com/rancher/elemental-operator:1.6.8
Elemental 仪表板扩展	3.0.1	3.0.1	Elemental 扩展 Helm chart (https://github.com/rancher/ui-plugin-charts/tree/4.0.0/charts/elemental/3.0.1)
Edge Image Builder	**1.2.1**	不适用	**registry.suse.com/edge/3.3/edge-image-builder:1.2.1**
NM Configurator	**0.3.3**	不适用	NMConfigurator 上游版本 (https://github.com/suse-edge/nm-configurator/releases/tag/v0.3.3)
KubeVirt	1.4.0	303.0.0+up0.5.0	registry.suse.com/edge/charts/kubevirt:303.0.0_up0.5.0 registry.suse.com/suse/sles/15.6/virt-operator:1.4.0

			registry.suse.com/suse/sles/15.6/virt-api:1.4.0 registry.suse.com/suse/sles/15.6/virt-controller:1.4.0 registry.suse.com/suse/sles/15.6/virt-exportproxy:1.4.0 registry.suse.com/suse/sles/15.6/virt-exportserver:1.4.0 registry.suse.com/suse/sles/15.6/virt-handler:1.4.0 registry.suse.com/suse/sles/15.6/virt-launcher:1.4.0
KubeVirt 仪表板扩展	1.3.2	303.0.2+up1.3.2	registry.suse.com/edge/charts/kubevirt-dashboard-extension:303.0.2_up1.3.2
Containerized Data Importer	1.61.0	303.0.0+up0.5.0	registry.suse.com/edge/charts/cdi:303.0.0_up0.5.0 registry.suse.com/suse/sles/15.6/cdi-operator:1.61.0 registry.suse.com/suse/sles/15.6/cdi-controller:1.61.0

			registry.suse.com/suse/sles/15.6/cdi-importer:1.61.0 registry.suse.com/suse/sles/15.6/cdi-cloner:1.61.0 registry.suse.com/suse/sles/15.6/cdi-apiserver:1.61.0 registry.suse.com/suse/sles/15.6/cdi-uploadserver:1.61.0 registry.suse.com/suse/sles/15.6/cdi-uploadproxy:1.61.0
Endpoint Copier Operator	0.2.0	303.0.0+up0.2.1	registry.suse.com/edge/charts/endpoint-copier-operator:303.0.0_up0.2.1 registry.suse.com/edge/3.3/endpoint-copier-operator:0.2.0
Akri（技术预览）	0.12.20	303.0.0+up0.12.20	registry.suse.com/edge/charts/akri:303.0.0_up0.12.20 registry.suse.com/edge/charts/akri-dashboard-extension:303.0.0_up1.3.1

			registry.suse.com/edge/3.3/akri-agent:v0.12.20 registry.suse.com/edge/3.3/akri-controller:v0.12.20 registry.suse.com/edge/3.3/akri-debug-echo-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-onvif-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-opcua-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-udev-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-webhook-configuration:v0.12.20
SR-IOV 网络操作器	1.5.0	303.0.2+up1.5.0	registry.suse.com/edge/charts/sriov-network-operator:303.0.2_up1.5.0

			registry.suse.com/edge/charts/sriov-crd:303.0.2_up1.5.0
系统升级控制器	0.15.2	106.0.0	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml) registry.suse.com/rancher/system-upgrade-controller:v0.15.2
升级控制器	**0.1.1**	**303.0.1+up0.1.1**	**registry.suse.com/edge/charts/upgrade-controller:303.0.1_up0.1.1** registry.suse.com/edge/3.3/upgrade-controller:0.1.1 **registry.suse.com/edge/3.3/kubectl:1.32.4 registry.suse.com/edge/3.3/release-manifest:3.3.1**
Kiwi 构建器	10.2.12.0	不适用	registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0

52.4 版本 3.3.0

发布日期：2025 年 5 月 20 日

摘要：SUSE Edge 3.3.0 是 SUSE Edge 3.3 系列版本中的第一个版本。

52.4.1 新功能

- 已更新到 Kubernetes 1.32 和 Rancher Prime 2.11
- 已将操作系统更新到 SUSE Linux Micro 6.1 (https://documentation.suse.com/sle-micro/6.1)
- 更新了 Rancher Turtles、Cluster API 和 Metal3/Ironic 版本
- 现在提供了一个容器映像，用于支持构建更新后的 SUSE Linux Micro 映像。有关详细信息，请参见第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”。
- 现在可通过定向网络置备流程部署 AArch64 下游群集。有关详细信息，请参见第 42 章 “全自动定向网络置备”。
- 在技术预览版中，现在可通过定向网络置备流程部署双栈下游群集。
- 在技术预览版中，现在支持精确时间协议 (PTP) 配置。有关详细信息，请参见第 52.5 节 “技术预览”。

52.4.2 Bug 和安全修复

- 通过为 RKE2 中的 ingress-nginx 应用补丁以修复 CVE-2025-1974 (https://nvd.nist.gov/vuln/detail/CVE-2025-1974) 漏洞。有关详细信息，请参见此处 (https://kubernetes.io/blog/2025/03/24/ingress-nginx-cve-2025-1974/)。
- SUSE Storage (Longhorn) 1.8.1 包含多项 bug 修复，包括：
  - 修复卷 FailedMount 问题，该问题可能会导致卷挂接失败（上游 Longhorn 问题 (https://github.com/longhorn/longhorn/issues/9939)）
  - 修复引擎卡在停止状态的问题，该问题可能会阻止卷挂接（上游 Longhorn 问题 (https://github.com/longhorn/longhorn/issues/9938)）

- Metal^3 chart 更新包含多项 bug 修复，包括：
  - 在包含 DHCP 服务器的网络中部署使用静态 IP 的群集时出现的 bug 已修复（上游问题 (https://github.com/suse-edge/charts/pull/196)）
- MetalLB chart 包含一项修复，现可确保在启用 frr-k8s 的情况下使用下游映像
- 已将 Kiwi 构建器更新到 10.2.12，以适配 Kiwi 近期的安全变更 - 映像验证的校验和方法从 md5 变为 sha256 (https://github.com/OSInside/kiwi/commit/d4d39e481aaff8be28337a9c76c3913a8a482628)。
- 已重新构建 Edge Image Builder 映像，以包含上述更新的 MetalLB 版本并适配 Kiwi 的变更。

52.4.3 已知问题

警告

如果要部署新群集，请先按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”所述构建全新映像。现在，无论是要创建 AMD64/Intel 64 还是 AArch64 体系结构的群集（包括管理群集和下游群集），都必须先执行此步骤。

- 在 SUSE Linux Micro 6.1 中使用 `toolbox` 时，默认容器映像不包含先前 5.5 版本中的部分工具。临时解决方法是将 toolbox 配置为使用先前的 `suse/sle-micro/5.5/toolbox` 容器映像，具体配置选项可参考 `toolbox --help`。
- 在某些情况下，通过 CAPI 进行滚动升级可能会导致计算机处于“删除中”状态，此问题将在未来的更新中解决（上游 RKE2 提供程序问题 655 (https://github.com/rancher/cluster-api-provider-rke2/issues/655)）
- 在某些情况下，如果使用的管理群集是从 Edge 3.2 升级的，那么通过 CAPI 进行的下游滚动升级可能会导致计算机处于“删除中”状态。此问题将通过未来的更新得到解决（上游 RKE2 提供程序问题 661 (https://github.com/rancher/cluster-api-provider-rke2/issues/661)）

- 使用 RKE2 1.32.3（此版本修复了 CVE-2025-1974 (https://nvd.nist.gov/vuln/detail/CVE-2025-1974) 漏洞）时，由于需要应用 SELinux 内核补丁，SUSE Linux Micro 6.1 **必须**进行更新，以包含 `>=6.4.0-26-default` 或 `>=6.4.0-30-rt`（实时内核）版本的内核。如果未进行该更新，ingress-nginx Pod 将持续处于 `CrashLoopBackOff` 状态。要进行内核更新，可在主机上运行 `transactional-update` 以更新所有软件包，或运行 `transactional-update pkg update kernel-default`（或 kernel-rt）仅更新内核，随后重引导主机。如果您是在部署新群集，请按照第 28 章 “使用 Kiwi 构建更新的 SUSE Linux Micro 映像”中的说明构建包含最新内核的全新映像。

- 当通过 nm-configurator 配置网络时，某些通过 MAC 地址识别接口的配置目前无法正常工作，此问题将在未来更新中解决（上游 NM Configurator 问题 (https://github.com/suse-edge/nm-configurator/issues/163)）

- 对于长时间运行的 Metal^3 管理群集，证书过期可能导致 baremetal-operator 与 ironic 的连接失败，此时需要手动重启 Pod 来作为临时解决方法（SUSE Edge chart 问题 (https://github.com/suse-edge/charts/issues/178)）

- 已发现 Kubernetes 作业控制器存在一个 bug，在特定状况下可能导致 RKE2/K3s 节点处于 `NotReady` 状态（参见 RKE2 问题 #8357 (https://github.com/rancher/rke2/issues/8357)）。错误消息可能如下所示：

```
E0605 23:11:18.489721    1 job_controller.go:631] "Unhandled Error" err="syncing
 job: tracking status: adding uncounted pods to status: Operation cannot be
 fulfilled on jobs.batch \"helm-install-rke2-ingress-nginx\": StorageError:
 invalid object, Code: 4, Key: /registry/jobs/kube-system/helm-install-rke2-
ingress-nginx, ResourceVersion: 0, AdditionalErrorMsg: Precondition failed:
 UID in precondition: 0aa6a781-7757-4c61-881a-cb1a4e47802c, UID in object meta:
 6a320146-16b8-4f83-88c5-fc8b5a59a581" logger="UnhandledError"
```

临时解决方法是使用 `crictl` 重启 `kube-controller-manager`，命令如下：

```
export CONTAINER_RUNTIME_ENDPOINT=unix:///run/k3s/containerd/containerd.sock
export KUBEMANAGER_POD=$(/var/lib/rancher/rke2/bin/crictl ps --label
 io.kubernetes.container.name=kube-controller-manager --quiet)
/var/lib/rancher/rke2/bin/crictl stop ${KUBEMANAGER_POD} && \
/var/lib/rancher/rke2/bin/crictl rm ${KUBEMANAGER_POD}
```

- 在 RKE2/K3s 1.31 和 1.32 版本中，由于存在与 `overlayfs` 相关的某些条件限制，用于存储 CNI 配置的 `/etc/cni` 目录可能无法向 `containerd` 触发文件写入通知（参见 RKE2 问题 #8356 (https://github.com/rancher/rke2/issues/8356)）。这会导致 RKE2/K3s 部署停滞在等待 CNI 启动的状态，且 RKE2/K3s 节点会保持 `NotReady` 状态。在节点级别，通过 `kubectl describe node <affected_node>` 可观察到该问题：

```
Conditions:
  Type          Status  LastHeartbeatTime              LastTransitionTime
      Reason                      Message
  ----          ------  -----------------              ------------------
      ------                      -------
  Ready         False   Thu, 05 Jun 2025 17:41:28 +0000   Thu, 05 Jun 2025
 14:38:16 +0000   KubeletNotReady           container runtime network not ready:
 NetworkReady=false reason:NetworkPluginNotReady message:Network plugin returns
 error: cni plugin not initialized
```

临时解决方法是在 RKE2 启动前将 tmpfs 卷挂载到 `/etc/cni` 目录。这会避免使用 overlayfs（overlayfs 会导致 containerd 漏收通知），且每次节点重启及 Pod 初始化容器重新运行时，配置都会被重写。如果使用 EIB，可在 `custom/scripts` 目录中添加 `04-tmpfs-cni.sh` 脚本（如此处[https://github.com/suse-edge/edge-image-builder/blob/release-1.2/docs/building-images.md#custom]所述），脚本内容如下：

```
#!/bin/bash
mkdir -p /etc/cni
mount -t tmpfs -o mode=0700,size=5M tmpfs /etc/cni
echo "tmpfs /etc/cni tmpfs defaults,size=5M,mode=0700 0 0" >> /etc/fstab
```

52.4.4 组件版本

下表描述了构成 3.3.0 版本的各个组件，包括版本、Helm chart 版本（如果适用），以及可从中提取已发布的二进制格式制品的位置。有关用法和部署示例，请参见相关文档。

名称	版本	Helm Chart 版本	制品位置（URL/映像）

SUSE Linux Micro	6.1（最新）	不适用	SUSE Linux Micro 下载页面 (https://www.suse.com/download/sle-micro/) SL-Micro.x86_64-6.1-Base-SelfInstall-GM.install.iso (sha256 70b9be28f2d92bc3b228412e4fc2b SL-Micro.x86_64-6.1-Base-RT-SelfInstall-GM.install.iso (sha256 9ce83e4545d4b36c7c6a44f7841dc SL-Micro.x86_64-6.1-Base-GM.raw.xz (sha256 36e3efa55822113840dd76fdf6914e SL-Micro.x86_64-6.1-Base-RT-GM.raw.xz (sha256 2ee66735da3e1da107b4878e73ae
SUSE Multi-Linux Manager	5.0.3	不适用	SUSE Multi-Linux Manager 下载页面 (https://www.suse.com/download/suse-manager/)

K3s	1.32.3	不适用	上游 K3s 版本 (https://github.com/k3s-io/k3s/releases/tag/v1.32.3%2Bk3s1)
RKE2	1.32.3	不适用	上游 RKE2 版本 (https://github.com/rancher/rke2/releases/tag/v1.32.3%2Brke2r1)
SUSE Rancher Prime	2.11.1	2.11.1	Rancher Prime Helm 储存库 (https://charts.rancher.com/server-charts/prime/index.yaml) Rancher 2.11.1 容器映像 (https://github.com/rancher/rancher/releases/download/v2.11.1/rancher-images.txt)
SUSE Storage	1.8.1	106.2.0+up1.8.1	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml) registry.suse.com/rancher/mirrored-longhornio-csi-attacher:v4.7.0

			registry.suse.com/rancher/mirrored-longhornio-csi-provisioner:v4.0.1-20241007 registry.suse.com/rancher/mirrored-longhornio-csi-resizer:v1.12.0 registry.suse.com/rancher/mirrored-longhornio-csi-snapshotter:v7.0.2-20241007 registry.suse.com/rancher/mirrored-longhornio-csi-node-driver-registrar:v2.12.0 registry.suse.com/rancher/mirrored-longhornio-livenessprobe:v2.14.0 registry.suse.com/rancher/mirrored-longhornio-backing-image-manager:v1.7.2 registry.suse.com/rancher/mirrored-longhornio-longhorn-engine:v1.7.2

			registry.suse.com/rancher/mirrored-longhornio-longhorn-instance-manager:v1.7.2 registry.suse.com/rancher/mirrored-longhornio-longhorn-manager:v1.7.2 registry.suse.com/rancher/mirrored-longhornio-longhorn-share-manager:v1.7.2 registry.suse.com/rancher/mirrored-longhornio-longhorn-ui:v1.7.2 registry.suse.com/rancher/mirrored-longhornio-support-bundle-kit:v0.0.45 registry.suse.com/rancher/mirrored-longhornio-longhorn-cli:v1.7.2
SUSE Security	5.4.3	106.0.0+up2.8.5	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml)

			registry.suse.com/rancher/neuvector-controller:5.4.3 registry.suse.com/rancher/neuvector-enforcer:5.4.3 registry.suse.com/rancher/neuvector-manager:5.4.3 registry.suse.com/rancher/neuvector-compliance-config:1.0.4 registry.suse.com/rancher/neuvector-registry-adapter:0.1.6 registry.suse.com/rancher/neuvector-scanner:6 registry.suse.com/rancher/neuvector-updater:0.0.2
Rancher Turtles (CAPI)	0.19.0	303.0.2+up0.19.0	registry.suse.com/edge/charts/rancher-turtles:303.0.2_up0.19.0 registry.rancher.com/rancher/rancher/turtles:v0.19.0

			registry.rancher.com/rancher/cluster-api-operator:v0.17.0 registry.rancher.com/rancher/cluster-api-metal3-controller:v1.9.3 registry.rancher.com/rancher/cluster-api-metal3-ipam-controller:v1.9.4 registry.suse.com/rancher/cluster-api-controller:v1.9.5 registry.suse.com/rancher/cluster-api-provider-rke2-bootstrap:v0.15.1 registry.suse.com/rancher/cluster-api-provider-rke2-controlplane:v0.15.1
Rancher Turtles 隔离资源	0.19.0	303.0.2+up0.19.0	registry.suse.com/edge/charts/rancher-turtles-airgap-resources:303.0.2_up0.19.0
Metal[3]	0.11.3	303.0.5+up0.11.3	registry.suse.com/edge/charts/metal3:303.0.5_up0.11.3

			registry.suse.com/edge/3.3/baremetal-operator:0.9.1 registry.suse.com/edge/3.3/ironic:26.1.2.4 registry.suse.com/edge/3.3/ironic-ipa-downloader:3.0.6 registry.suse.com/edge/mariadb:10.6.15.1
MetalLB	0.14.9	303.0.0+up0.14.9	registry.suse.com/edge/charts/metallb:303.0.0_up0.14.9 registry.suse.com/edge/3.3/metallb-controller:v0.14.8 registry.suse.com/edge/3.3/metallb-speaker:v0.14.8 registry.suse.com/edge/3.3/frr:8.4 registry.suse.com/edge/3.3/frr-k8s:v0.0.14
Elemental	1.6.8	1.6.8	registry.suse.com/rancher/elemental-operator-chart:1.6.8

			registry.suse.com/rancher/elemental-operator-crds-chart:1.6.8 registry.suse.com/rancher/elemental-operator:1.6.8
Elemental 仪表板扩展	3.0.1	3.0.1	Elemental 扩展 Helm chart (https://github.com/rancher/ui-plugin-charts/tree/4.0.0/charts/elemental/3.0.1)
Edge Image Builder	1.2.0	不适用	registry.suse.com/edge/3.3/edge-image-builder:1.2.0
NM Configurator	0.3.2	不适用	NMConfigurator 上游版本 (https://github.com/suse-edge/nm-configurator/releases/tag/v0.3.2)
KubeVirt	1.4.0	303.0.0+up0.5.0	registry.suse.com/edge/charts/kubevirt:303.0.0_up0.5.0 registry.suse.com/suse/sles/15.6/virt-operator:1.4.0

			registry.suse.com/suse/sles/15.6/virt-api:1.4.0 registry.suse.com/suse/sles/15.6/virt-controller:1.4.0 registry.suse.com/suse/sles/15.6/virt-exportproxy:1.4.0 registry.suse.com/suse/sles/15.6/virt-exportserver:1.4.0 registry.suse.com/suse/sles/15.6/virt-handler:1.4.0 registry.suse.com/suse/sles/15.6/virt-launcher:1.4.0
KubeVirt 仪表板扩展	1.3.2	303.0.2+up1.3.2	registry.suse.com/edge/charts/kubevirt-dashboard-extension:303.0.2_up1.3.2
Containerized Data Importer	1.61.0	303.0.0+up0.5.0	registry.suse.com/edge/charts/cdi:303.0.0_up0.5.0 registry.suse.com/suse/sles/15.6/cdi-operator:1.61.0 registry.suse.com/suse/sles/15.6/cdi-controller:1.61.0

			registry.suse.com/suse/sles/15.6/cdi-importer:1.61.0 registry.suse.com/suse/sles/15.6/cdi-cloner:1.61.0 registry.suse.com/suse/sles/15.6/cdi-apiserver:1.61.0 registry.suse.com/suse/sles/15.6/cdi-uploadserver:1.61.0 registry.suse.com/suse/sles/15.6/cdi-uploadproxy:1.61.0
Endpoint Copier Operator	0.2.0	303.0.0+up0.2.1	registry.suse.com/edge/charts/endpoint-copier-operator:303.0.0_up0.2.1 registry.suse.com/edge/3.3/endpoint-copier-operator:0.2.0
Akri（技术预览）	0.12.20	303.0.0+up0.12.20	registry.suse.com/edge/charts/akri:303.0.0_up0.12.20 registry.suse.com/edge/charts/akri-dashboard-extension:303.0.0_up1.3.1

			registry.suse.com/edge/3.3/akri-agent:v0.12.20 registry.suse.com/edge/3.3/akri-controller:v0.12.20 registry.suse.com/edge/3.3/akri-debug-echo-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-onvif-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-opcua-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-udev-discovery-handler:v0.12.20 registry.suse.com/edge/3.3/akri-webhook-configuration:v0.12.20
SR-IOV 网络操作器	1.5.0	303.0.2+up1.5.0	registry.suse.com/edge/charts/sriov-network-operator:303.0.2_up1.5.0

			registry.suse.com/edge/charts/sriov-crd:303.0.2_up1.5.0
系统升级控制器	0.15.2	106.0.0	Rancher chart Helm 储存库 (https://charts.rancher.io/index.yaml) registry.suse.com/rancher/system-upgrade-controller:v0.15.2
升级控制器	0.1.1	303.0.0+up0.1.1	registry.suse.com/edge/charts/upgrade-controller:303.0.0_up0.1.1 registry.suse.com/edge/3.3/upgrade-controller:0.1.1 registry.suse.com/edge/3.3/kubectl:1.30.3 registry.suse.com/edge/3.3/release-manifest:3.3.0
Kiwi 构建器	10.2.12.0	不适用	registry.suse.com/edge/3.3/kiwi-builder:10.2.12.0

52.5 技术预览

除非另有说明，否则下述内容适用于 3.3.0 版本和所有后续 z-stream 版本。

- Akri 为技术预览功能，不涵盖在标准支持范围内。
- IPv6 及双栈下游部署为技术预览功能，不涵盖在标准支持范围内。
- 下游部署中的精确时间协议 (PTP) 为技术预览功能，不涵盖在标准支持范围内。

52.6 组件验证

可以使用软件材料清单 (SBOM) 数据来验证上述组件 - 例如，如下所述使用 `cosign`：

从 SUSE 签名密钥来源 (https://www.suse.com/support/security/keys/) 下载 SUSE Edge 容器公共密钥：

```
> cat key.pem
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA7N0S2d8LFKW4WU43bq7Z
IZT537xlKe17OQEpYjNrdtqnSwA0/jLtK83m7bTzfYRK4wty/so0g3BGo+x6yDFt
SVXTPBqnYvabU/j7UKaybJtX3jc4SjaezeBqdi96h6yEslvg4VTZDpy6TFP5ZHxZ
A0fX6m5kU2/RYhGXItoeUmL5hZ+APYgYG4/455NBaZT2yOywJ6+1zRgpR0cRAekI
OZXl51k0ebsGV6ui/NGECO6MB5e3arAhszf8eHDE02FeNJw5cimXkgDh/1Lg3KpO
dvUNm0EPWvnkNYeMCKR+687QG0bXqSVyCbY6+HG/HLkeBWkv6Hn41oeTSLrjYVGa
T3zxPVQM726sami6pgZ5vULyOleQuKBZrlFhFLbFyXqv1/DokUqEppm2Y3xZQv77
fMNogapp0qYz+nE3wSK4UHPd9z+2bq5WEkQSalYxadyuqOzxqZgSoCNoX5iIuWte
Zf1RmHjiEndg/2UgxKUysVnyCpiWoGbalM4dnWE24102050Gj6M4B5fe73hbaRlf
NBqP+97uznnRlSl8FizhXzdzJiVPcRav1tDdRUyDE2XkNRXmGfD3aCmILhB27SOA
Lppkouw849PWBt9kDMvzelUYLpINYpHRi2+/eyhHNlufeyJ7e7d6N9VcvjR/6qWG
64iSkcF2DTW61CN5TrCe0k0CAwEAAQ==
-----END PUBLIC KEY-----
```

验证容器映像哈希，例如，使用 `crane` 进行验证：

```
> crane digest registry.suse.com/edge/3.3/baremetal-operator:0.9.1 --platform
 linux/amd64
sha256:02c5590cd51b1a1ea02f9908f2184ef4fbc856eb0197e804a7d57566d9278ddd
```

注意

对于多体系结构映像，获取摘要时还需要指定平台，例如 `--platform linux/amd64` 或 `--platform linux/arm64`。如果不这样做，后续步骤中会出现错误 (`Error: no matching attestations`)。

使用 `cosign` 进行验证：

```
> cosign verify-attestation --type spdxjson --key
 key.pem registry.suse.com/edge/3.3/baremetal-
operator@sha256:02c5590cd51b1a1ea02f9908f2184ef4fbc856eb0197e804a7d57566d9278ddd
 > /dev/null
#
Verification for registry.suse.com/edge/3.3/baremetal-
operator@sha256:02c5590cd51b1a1ea02f9908f2184ef4fbc856eb0197e804a7d57566d9278ddd
 --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - The claims were present in the transparency log
  - The signatures were integrated into the transparency log when the
 certificate was valid
  - The signatures were verified against the specified public key
```

按照 SUSE SBOM 文档 (https://www.suse.com/support/security/sbom/) 中所述提取 SBOM 数据：

```
> cosign verify-attestation --type spdxjson --key
 key.pem registry.suse.com/edge/3.2/baremetal-
operator@sha256:d85c1bcd286dec81a3806a8fb8b66c0e0741797f23174f5f6f41281b1e27c52f
 | jq '.payload | @base64d | fromjson | .predicate'
```

52.7 升级步骤

有关如何升级到新版本的详细信息，请参见第 VI 部分 “Day 2 操作”。

52.8 产品支持生命周期

SUSE Edge 拥有 SUSE 屡获殊荣的支持服务作为后盾。SUSE 是公认的技术领导者，在提供企业级品质支持服务方面有着久经验证的历史。有关详细信息，请参见 https://www.suse.com/lifecycle 和支持策略页面 (https://www.suse.com/support/policy.html)。如果您在提交支持案例、SUSE 如何划分严重性级别或支持范围方面有任何疑问，请参见技术支持手册 (https://www.suse.com/support/handbook/)。

SUSE Edge“3.3”版本提供 18 个月的生产支持，其中前 6 个月提供“全面支持”，随后 12 个月提供“维护支持”。在这些支持阶段结束后，产品将进入“生命周期终止” (EOL) 状态，不再提供任何支持。有关各生命周期阶段的详细信息，请参见下表：

全面支持（6 个月）	在全面支持期内，会发布紧急和选定的高优先级 bug 修复，其他所有补丁（非紧急修复、功能增强、新功能）将按常规发布计划发布。
维护支持（12 个月）	在此期间，仅通过补丁发布关键修复。其他 bug 修复可能由 SUSE 酌情发布，但不做保证。
生命周期终止 (EOL)	产品版本到达生命周期终止日期后，客户可根据产品许可协议继续使用该产品，但 SUSE 的支持计划不再适用于已进入生命周期终止状态的产品版本。

除非明确说明，否则列出的所有组件均被视为正式发布 (GA) 版本，并涵盖在 SUSE 的标准支持范围内。某些组件可能以“技术预览”版本的形式列出，SUSE 通过此类版本让客户提前体验尚未正式发布的特性和功能以进行评估，但这些组件没有标准支持政策的保障，不建议将其用于生产使用场景。SUSE 非常欢迎大家提供针对技术预览组件的改进意见和建议，但如果技术预览功能无法满足客户的需求或者达不到我们要求的成熟度，SUSE 保留在正式发布之前弃用该功能的权利。

请注意，SUSE 必须偶尔弃用功能或更改 API 规范。弃用功能或更改 API 的原因包括相应功能已更新或由新的实现取代、有新的功能集、上游技术不再可用，或者上游社区引入了不兼容的更改。在给定的次要版本 (x.z) 中，这种情况预期永远不会发生，因此所有 z-stream 版本都将保持 API 兼容性和原有功能。SUSE 将尽力在发行说明中提供弃用警告并发出充足的通告，同时提供解决方法、建议和缓解措施，以最大程度地减少服务中断。

SUSE Edge 团队也欢迎社区提出反馈，相关问题可在 https://www.github.com/suse-edge 中的相应代码储存库内提出。

52.9 获取源代码

本 SUSE 产品包含根据 GNU 通用公共许可证 (GPL) 和其他各种开源许可证授权给 SUSE 的组件。根据 GPL 要求，SUSE 必须提供与 GPL 授权组件对应的源代码，此外还须严格遵守其他所有开源许可要求。因此，SUSE 会公开所有源代码，这些源代码通常可在 SUSE Edge GitHub 储存库 (https://www.github.com/suse-edge)、用于依赖组件的 SUSE Rancher GitHub 储存库 (https://www.github.com/rancher) 中找到；具体到 SUSE Linux Micro，其源代码可在 https://www.suse.com/download/sle-micro (https://www.suse.com/download/sle-micro/) 的“Medium 2”处下载。

52.10 法律声明

SUSE 不对本文档的内容或其使用作出任何陈述或保证，并明确否认对适销性或任何特定用途的适用性作出任何明示或暗示保证。此外，SUSE 保留随时修订本出版物和更改其内容的权利，且无义务将此类修订或更改通知任何个人或实体。

此外，SUSE 不对任何软件作出任何陈述或保证，并明确否认对适销性或任何特定用途的适用性作出任何明示或暗示保证。此外，SUSE 保留随时更改 SUSE 软件任何或所有部分的权利，且无义务将此类更改通知任何个人或实体。

根据本协议提供的任何产品或技术信息可能受到美国出口管制和其他国家/地区贸易法的约束。您同意遵守所有出口管制法规，并在出口、再出口或进口可交付产品之前取得所有必要的许可证或分类证书。您同意不向当前美国出口排除清单上的实体，或美国出口法中规定的任何禁运国家/地区或支持恐怖主义的国家/地区出口或再出口。您同意不将可交付产品用于

受禁核武器、导弹或生化武器的最终用途。有关出口 SUSE 软件的详细信息，请访问 https://www.suse.com/company/legal/。如果您未能取得任何必要的出口许可，SUSE 对此不承担任何责任。

版权所有 © 2024 SUSE LLC。

本发行说明文档根据 Creative Commons Attribution-NoDerivatives 4.0 国际许可证 (CC-BY-ND-4.0) 授权。您应已随本文档收到了一份许可证。如果没有，请访问 https://creativecommons.org/licenses/by-nd/4.0/。

SUSE 拥有与本文档所述产品中体现的技术相关的知识产权。具体而言，这些知识产权可能包括但不限于 https://www.suse.com/company/legal/ 中列出的一项或多项美国专利，以及美国和其他国家/地区的一项或多项其他专利或正在申请的专利。

有关 SUSE 商标，请参见 SUSE 商标和服务标记列表 (https://www.suse.com/company/legal/)。所有第三方商标均是其各自所有者的财产。有关 SUSE 品牌信息和使用要求，请参见 https://brand.suse.com/ 上发布的准则。