104Kubernetes 系列（九十七）如何跟踪 Kubernetes Pod 到 Pod 的流量

我们将探索 Kubernetes 网络的复杂性，并深入研究管理 Pod 到 Pod 通信的基本原则和机制。

在本文中，我们将重点讨论 VirtualBox 默认网络布局背景下的 Kubernetes 网络模型。通过检查此设置，我们将深入了解 Kubernetes 环境中 Pod 到 Pod 的通信如何工作。我们将揭示实现集群内 Pod 之间无缝通信的基本概念和机制。

Kubernetes 网络

在 Kubernetes 网络领域，必须注意的是，它并不与本机网络实现捆绑在一起。相反，它采用灵活的方法，使用容器运行时接口（CRI）和容器网络接口（CNI）标准解耦集群网络实现。这些标准充当第三方提供商提供自己的 Kubernetes 网络实现插件的网关，而这些插件又在设置 Kubernetes 集群时提供了多种可供选择的选项。

鉴于这种多样性，很明显，Kubernetes 用户可以使用多种替代方案，而不是单一的指定网络解决方案。这个广泛的选项中的每个选项都有其自己的一组独特的特征和复杂性，这可以显著影响最终的集群网络实施。这种广泛的选择确保组织可以选择最符合其特定要求的网络方法，同时考虑可扩展性、性能、安全性以及与现有基础设施的兼容性。

值得强调的是，多样化的 Kubernetes 网络选项使组织能够微调其集群的网络配置，以满足其独特的需求。然而，如此丰富的选择也可能带来挑战，因为选择过程需要仔细评估与每个网络解决方案相关的权衡和考虑因素。通过了解不同 Kubernetes 网络选项的细微差别，组织可以做出明智的决策，以促进 Kubernetes 集群内的最佳性能、可靠的通信和简化的管理。

当谈到 Kubernetes CRI 和 CNI 插件时，您有一系列流行的选择。在 CRI 方面，containerd 和 cri-o 成为两种流行的选择，而在 CNI 方面，Calico 和 Flannel 脱颖而出，成为流行的替代方案。这些插件为 Kubernetes 集群内的容器运行时和网络管理提供了基本的功能和特性，允许您选择最符合您的特定要求和偏好的选项。

Kubernetes 发行版

一个功能齐全的 Kubernetes 集群需要 CRI 和 CNI 插件和谐共存。插件之间的这种共存需要配置工作以确保它们按预期工作。因此，Kubernetes 发行版作为一种让大众更容易使用 Kubernetes 的替代方式应运而生。

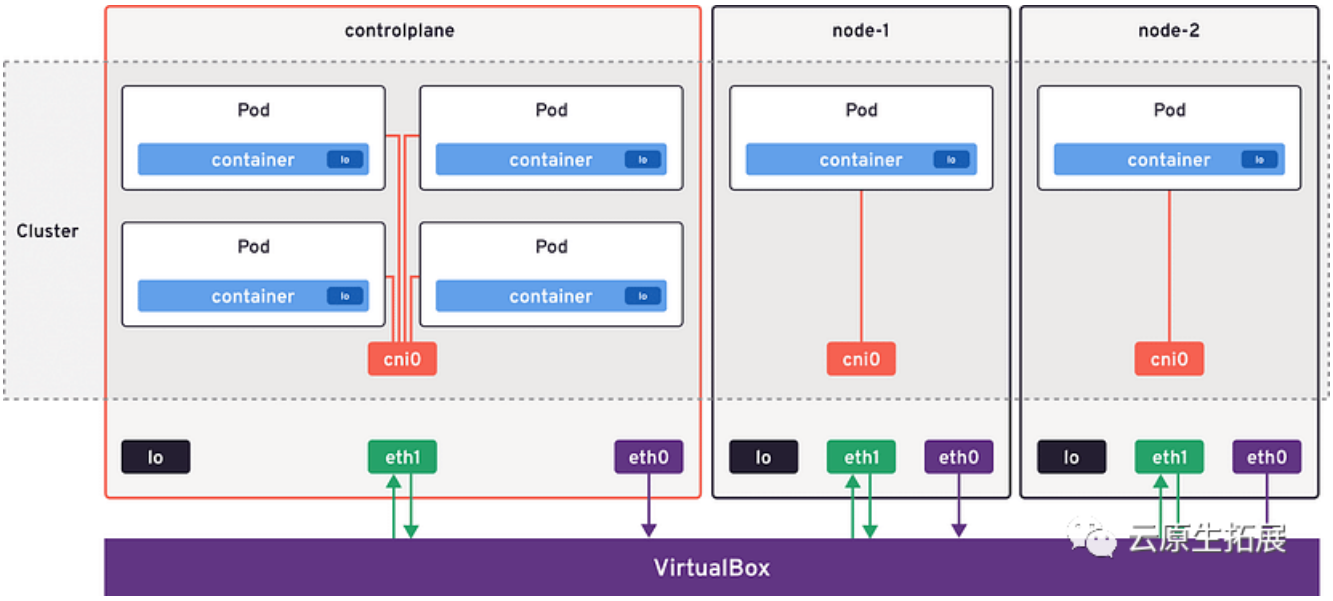
Kubernetes 发行版安装和配置所有 Kubernetes 组件，包括 CRI 和 CNI 插件。您无需担心底层平台配置细节。它们全部收集在一起，供您直接进入 Kubernetes 应用程序。您需要做的就是运行命令，等待几分钟让平台部署所有资源，并在部署过程完成后访问为您提供的 Kubernetes API。

您可以找到 Red Hat Openshift、AWS EKS、Google GKS 等商业发行版以及 OKD 和 minikube 等开源发行版。

在本文中，您将使用 vagrant-k8s-lab（https://gitlab.com/areguera/vagrant-k8s-lab） 探索 Pod 到 Pod 的流量。这个小型开源 Kubernetes 发行版在本地启动集群，使用带有 VirtualBox 提供程序的 Vagrant 和 Ansible 角色来配置集群机器。vagrant-k8-lab 的存在仅用于学习目的，并且它很乐意接受任何人的贡献。

Kubernetes 集群架构

vagrant-k8-lab项目在所有集群节点中安装和配置containerd CRI和flannel CNI插件。集群架构如下所示：



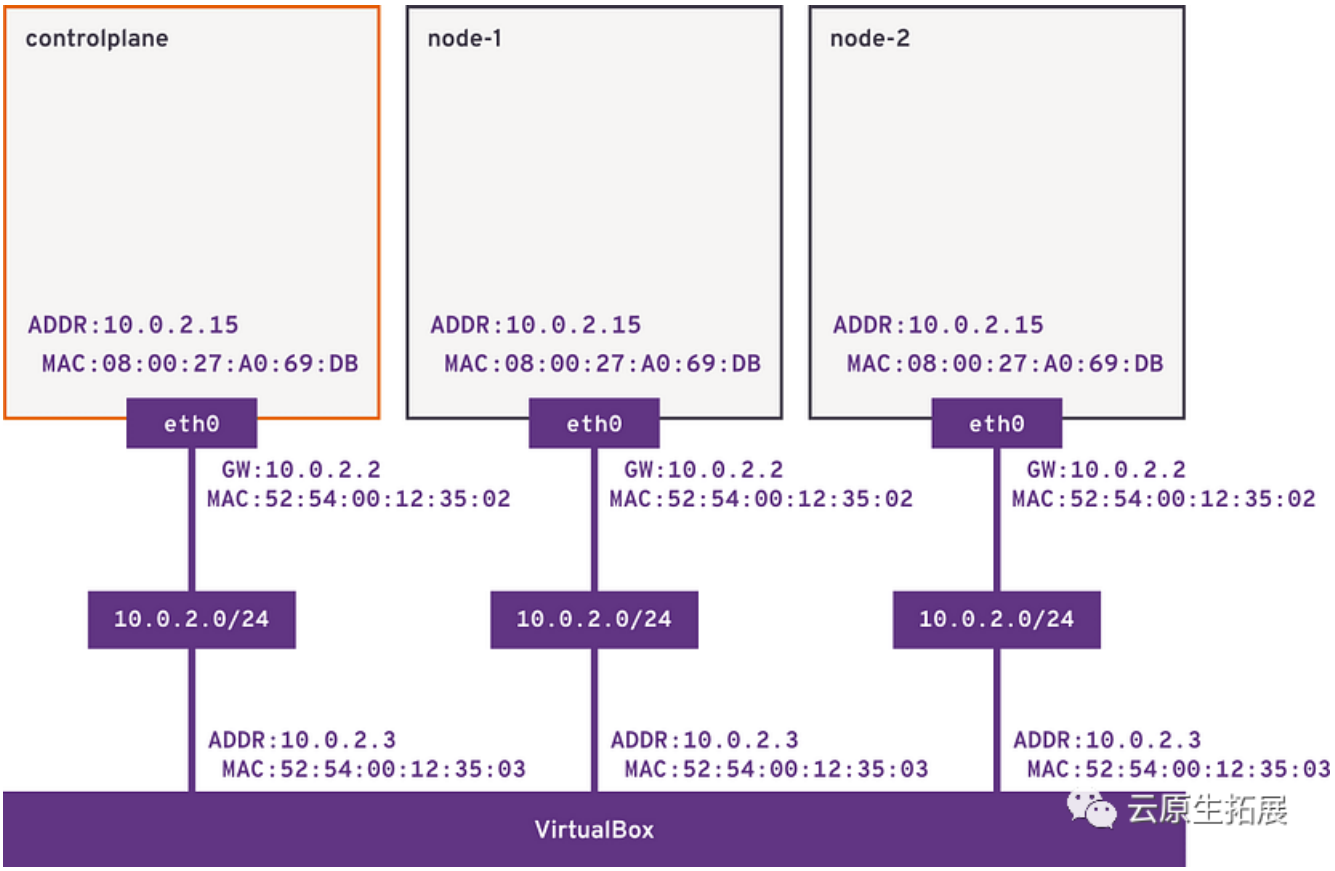
上图描述了您在本文中使用的关于 Pod 到 Pod 通信的集群架构。请注意，每个节点都有四个网络接口。网络接口为 lo、eth0、eth1 和 cni0。每个接口都有其存在的目的和理由。此外，集群已经部署了几个 pod，用于本文后面的测试目的。

Kubernetes 节点到互联网流量 (eth0)

CRI 插件需要 Kubernetes 集群中的出站流量才能从外部容器仓库下载容器镜像。操作系统包管理器还必须访问远程包存储库以下载系统更新。这两个动作要求 Kubernetes 节点有出站流量。

例如，当 Kubernetes Scheduler 命令在节点中运行的 Kubelet 部署 pod 时，该节点的 CRI 插件负责联系外部仓库以下载 Kubernetes Scheduler 命令部署的 pod 清单中定义的容器镜像。如果该节点无法与外部容器仓库通信，则 Pod 创建将失败。

当您使用 VirtualBox 虚拟机部署 Kubernetes 节点时，它们是使用已在 NAT 模式下连接的 eth0 网络接口创建的。在IP级别，客户操作系统接收10.0.2.15/24地址，并可以通过10.0.2.3/24地址与VirtualBox通信。此网络布局称为 VirtualBox 默认网络布局。只要运行 VirtualBox 的主机具有所需的连接，它就允许虚拟机与外部资源进行通信。



VirtualBox 默认网络布局可能看起来违反直觉，直到您意识到虚拟机连接到不同的网络，即使它们都使用相同的地址空间。此 VirtualBox 设计选择允许在所有虚拟机上使用一致的 IP 寻址模式进行外部世界通信。

在 VirtualBox 的默认网络布局中，虚拟机之间的连接是单独处理的。每个虚拟机都连接到 VirtualBox 内自己独立的“10.0.2.0/24”网络，可能在内部为每个虚拟机使用唯一的虚拟链接。因此，虚拟机可以与外部世界通信，但不能在它们之间通信。

实际案例

1. 使用 nginx 镜像创建一个名为 nginx 的 pod：

```
[vagrant@node-1 ~]$ kubectl run nginx --image=nginx:latest
```

2. 检查 Pod 状态：

```
[vagrant@node-1 ~]$ kubectl get pods/nginx
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          10m
```

Pod 状态为“正在运行”。好的！此操作需要 Containerd 联系远程容器仓库（例如registry.docker.io）并下载 nginx 容器镜像。如果 Pod 正在运行，则意味着这些操作在 Node-1 中成功执行，假设 Node-1 是 Kubernetes Scheduler 选择运行操作的节点。您可以通过运行以下命令获得有关这些操作的更多信息：

```
[vagrant@node-1 ~]$ kubectl describe pods/nginx
[vagrant@node-1 ~]$ journalctl -f -u containerd
```

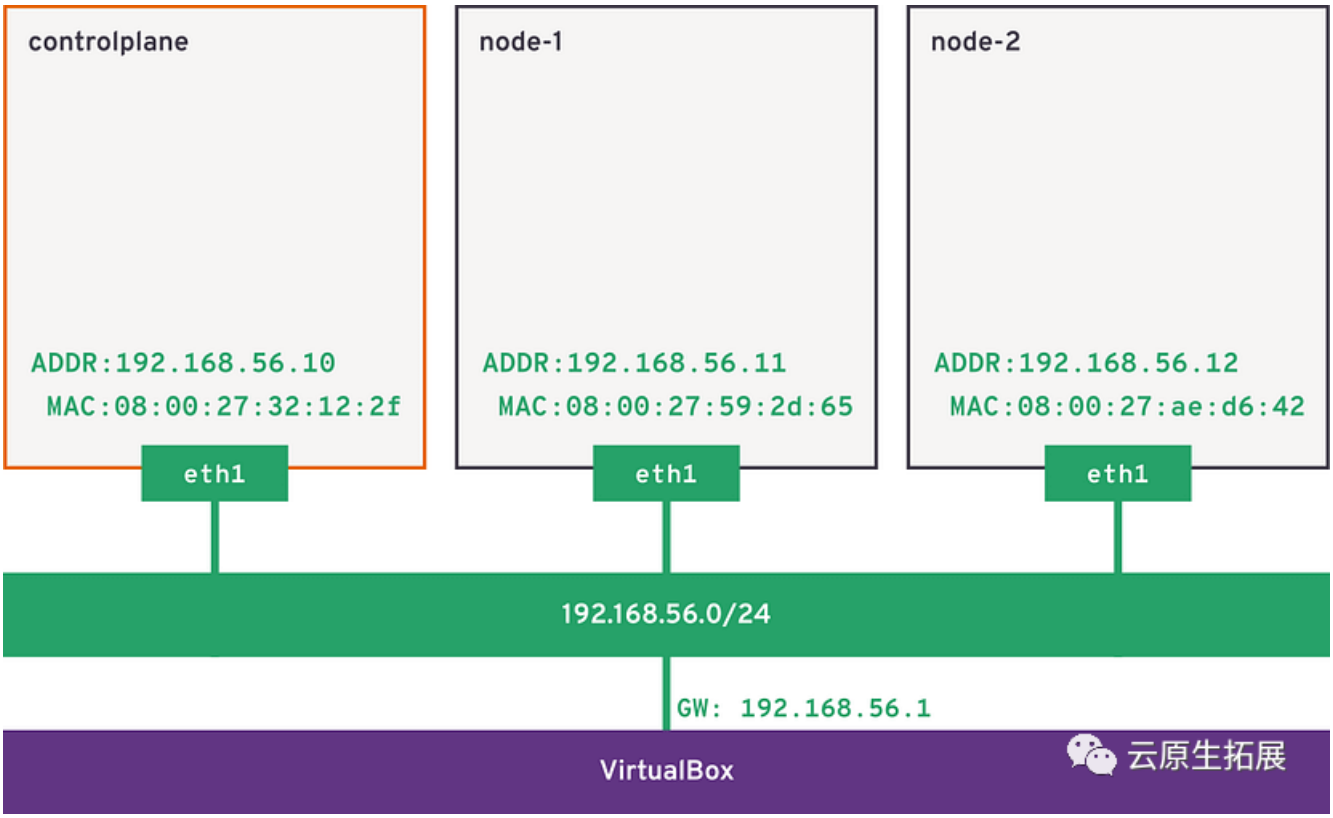
Kubernetes 节点到节点流量 (eth1)

当使用 host-gw 后端配置 Flannel CNI 插件时，要求 Kubernetes 节点之间具有直接连接，例如，将每个节点连接到相同的物理以太网链路。 eth0网络接口不能满足这样的需求，因为它使用VirtualBox的NAT网络类型。为了满足 Flannel CNI 插件的需求，需要为 Kubernetes 集群中的所有节点附加不同的网络类型。

VirtualBox 有不同的网络类型，您在创建新网络时可以选择。 Kubernetes 节点在本文中使用“仅主机网络”类型。

VirtualBox 的仅主机网络类型允许集群内的节点之间进行通信。默认情况下，Flannel CNI 插件配置使用 eth0 网络接口进行节点到节点通信。本文使用的集群安装过程将默认 CNI 插件配置更改为引用 eth1，即用于节点间通信的新网络接口值。

以下是附加了 eth1 网络的集群的直观表示：



将 eth1 连接到 VirtualBox 默认网络布局”中，节点间通信通过 192.168.56.0/24 网络地址进行。每个节点都有一个在此范围内分配的 IP 地址。从控制平面的 192.168.56.10/24 开始，然后是第一个工作节点的 192.168.56.11/24，以及第二个工作节点的 192.168.56.12/24。

实际例子

1. 检查从控制平面节点到节点 1 worker 的连接：

```
[vagrant@controlplane ~]$ traceroute -n 192.168.56.11
traceroute to 192.168.56.11 (192.168.56.11), 30 hops max, 60 byte packets
 1 192.168.56.11 0.900 ms 0.850 ms 0.717 ms
```

2. 检查从控制平面节点到节点 2 worker 的连接：

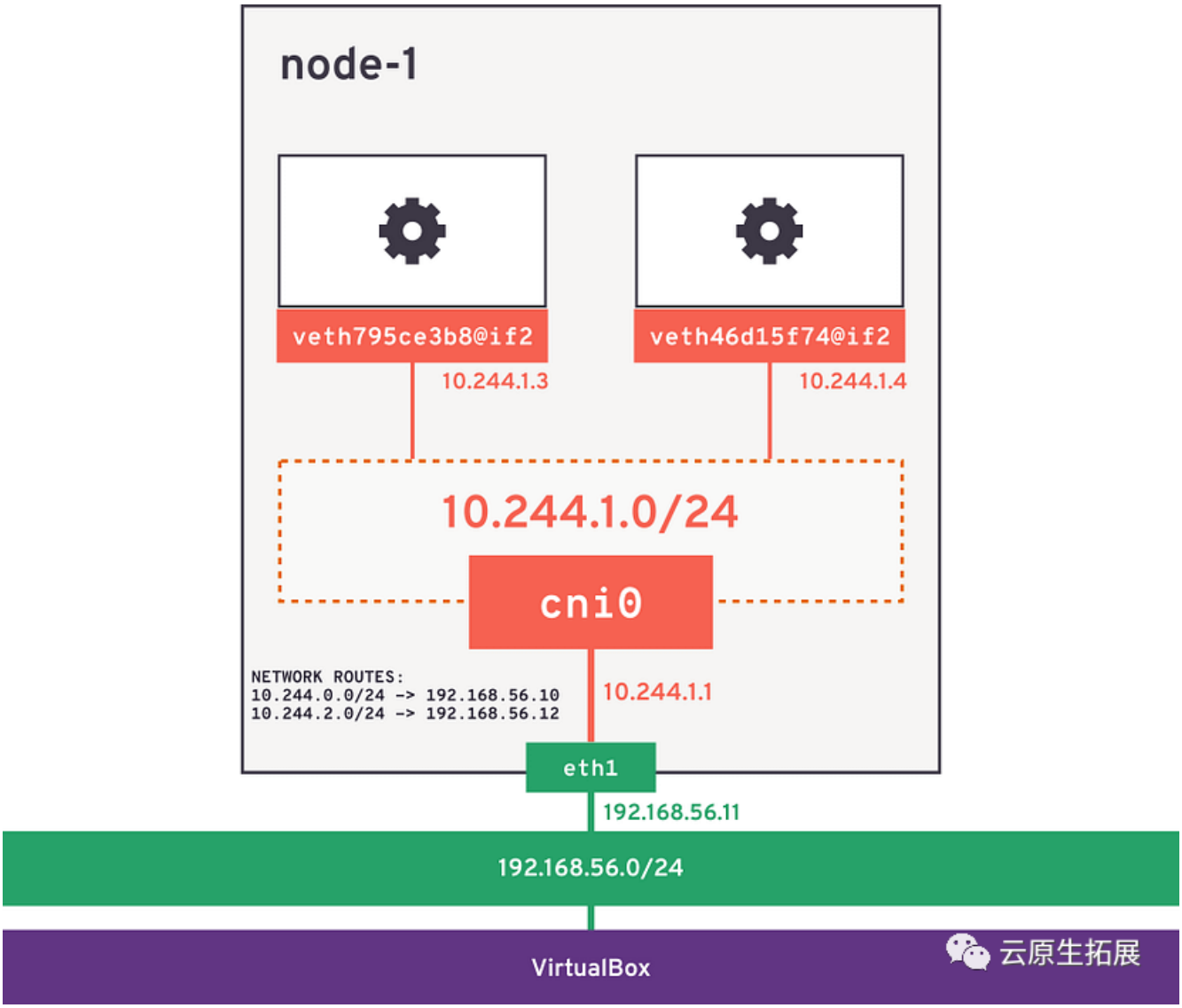
```
[vagrant@controlplane ~]$ traceroute -n 192.168.56.12
traceroute to 192.168.56.12 (192.168.56.12), 30 hops max, 60 byte packets
 1 192.168.56.12 1.003 ms 0.837 ms 0.851 ms
```

经实际验证，控制平面节点1和节点2可达。出色的！

同一节点中 Pod 到 Pod 的流量 (cni0)

当您在单个节点中运行 Kubernetes 应用程序时，会发生 Pod 到 Pod 的流量。在这种情况下，Pod 必须能够共享信息并拥有允许永久通信的 IP 地址。请记住，pod 是暂时的；它们可以随时被销毁和创建。实现这一点的网络逻辑主要由您在集群中安装和配置的 CNI 插件管理。

让我们可视化 Node-1 的架构，其中有两个 Pod 正在运行：



在连接到节点 1 的网络接口中，cni0 网络接口连接节点 1（创建控制平面节点后我们连接到集群的第一个节点）中的所有 Pod。通过 cni0 网络接口，CNI 插件管理 Kubernetes 网络资源（例如 Pod）的 IP 地址分配。每当您创建网络资源时，它都会收到一个动态IP地址，当资源被销毁时，分配的IP地址将被释放以供以后重新分配。为了保留已分配和未分配的 IP 地址之间的关系，Flannel CNI 插件使用 Kubernetes 存储服务 etcd。

实际例子

1. 使用 httpd 镜像创建一个名为 httpd 的 pod。

```
[vagrant@node-1 ~]$ kubectl run httpd --image=httpd
```

2. 检查 httpd pod 状态以获取其 IP 地址。假设它是 10.244.1.3/24：

```
[vagrant@node-1 ~]$ kubectl get pods -o wide
```

3. 使用 busybox 镜像创建一个新 pod，以针对 httpd pod IP 地址运行 traceroute 命令：

```
[vagrant@node-1 ~]$ kubectl run traceroute --image=busybox -- traceroute 10.244.1.3
```

4. 检查traceroute pod中的日志输出：

```
[vagrant@node-1 ~]$ kubectl logs traceroute
traceroute to 10.244.1.3 (10.244.1.3), 30 hops max, 80 byte packets
 1  10.244.1.3 (10.244.1.3)  0.018 ms  0.009 ms  0.005 ms
```

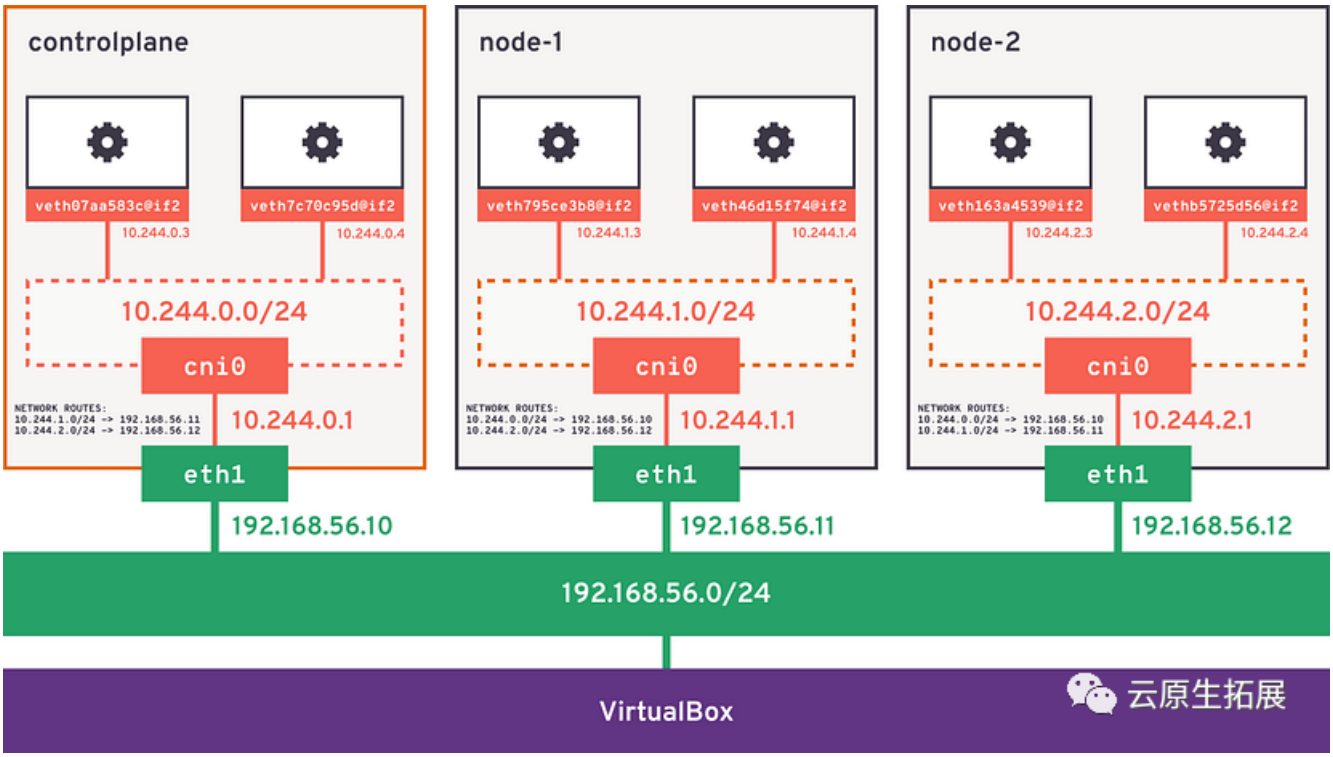
跟踪路由日志显示到达目标 Pod 的一跳，这是预期的，考虑到两者都位于同一 10.244.1.0/24 网络中。根据证据，我们可以得出结论，当 Pod 在同一节点中运行时，通信是通过 cni0 网络虚拟接口在本地进行的。在这些情况下，网络流量永远不会离开节点。

不同节点之间的 Pod 到 Pod 流量 (cni0+eth1+cni0)

具有单个节点的 Kubernetes 集群对于某些概念测试很有用，但不能为您的工作负载提供冗余。如果节点发生故障，您的应用程序就会失败。冗余是在多节点集群中运行应用程序的结果。运行多节点集群时，了解流量如何从一个资源流向另一个资源可以为您在故障排除期间提供强大的工具。这种理解很大程度上取决于 Kubernetes 集群中配置的 CNI 插件，因为每个 CNI 插件都有其解决此问题的方法。

节点需要知道在集群内将流量发送到哪里。为了解决这个问题，当配置了 host-gw 后端时，Flannel 插件使用操作系统路由表来管理每个节点上的此类知识。当您向集群添加新节点或从集群中删除旧节点时，CNI 插件会自动涵盖所有这些工作。您不需要自己维护路由表。 CNI 插件可以帮您做到这一点。当你想到这一点时，工作量很大！

让我们将 Kubernetes 集群可视化更进一步，考虑三个节点，每个节点运行两个 pod，位于通过 eth1 网络连接的 3 个不同的 CNI 网络中，并且已经为操作系统设置了网络路由，以便知道在集群内将包发送到哪里：



让我们打开一个终端并重复我们之前所做的相同的跟踪路由练习，现在考虑不同节点上的 Pod。

实际例子

1. 创建一个名为 httpd 的 pod，使用 httpd 镜像，并确保将其放置在 node-1 上：

```
[vagrant@controlplane ~]$ kubectl run httpd --image=httpd \
 --overrides='{"spec": { "nodeSelector": {"kubernetes.io/hostname": "node-1"}}}'
```

2. 检查 pod 的状态，获取 pod httpd 的 IP 地址（例如 10.244.1.3）。该 Pod 是目标：

```
[vagrant@controlplane ~]$ kubectl get pods -o wide
```

3. 创建一个名为traceroute的pod，使用busybox镜像，并以httpd pod的IP地址作为参数运行traceroute命令。确保此 pod 放置在节点 2 上：

```
[vagrant@controlplane ~]$ kubectl run traceroute --image=busybox \
 --overrides='{"spec": { "nodeSelector": {"kubernetes.io/hostname": "node-2"}}}' \
 -- traceroute 10.244.1.3
```

4. 检查traceroute pod中的日志输出：

```
[vagrant@controlplane ~]$ kubectl logs traceroute
traceroute to 10.244.1.3 (10.244.1.3), 30 hops max, 46 byte packets
 1  10.244.2.1 (10.244.2.1)  0.007 ms  0.008 ms  0.004 ms
 2  192.168.56.11 (192.168.56.11)  0.575 ms  0.411 ms  0.396 ms
 3  10.244.1.3 (10.244.1.3)  0.614 ms  0.580 ms  0.498 ms
```

这次traceroute 输出显示了三跳：

- 第一个是 10.244.2.1，在节点 2 的 cni0 网络接口中配置的网关地址。
- 第二跳是 192.168.56.11，这是节点 1 的 eth1 网络接口的 IP 地址。
- 第三个也是最后一个跃点是 10.244.1.3，该 IP 地址附加到节点 1 的 cni0 网络，并分配给我们想要测试连接的 Pod。

我们可以得出结论：当 pod 位于不同节点时，通信在每个节点所附的 cni0 和 eth1 接口之间路由。

总结

在本文中，我们深入研究了 VirtualBox 背景下的 Kubernetes 网络，提供了命令行示例和插图，以阐明 Pod 到 Pod 的通信。然而，重要的是要认识到我们的探索只触及了这个广阔主题的表面。

当您继续探索 Kubernetes 网络时，很明显，了解网络包的流动方式对于有效识别和排除网络问题至关重要。无论是构建您自己的 Kubernetes 集群学习实验室还是管理现有部署，当出现意外的通信问题时，这些知识都是非常宝贵的。

我鼓励您在继续学习和探索 Kubernetes 网络的复杂性时让好奇心引导您。跟踪 Pod 到 Pod 的流量，揭示底层机制，并迎接在 Kubernetes 环境中保持可靠和高效通信的挑战。请记住，总是有更多东西有待发现，因此请不断扩展您的专业知识并增强排除故障和优化 Kubernetes 网络的能力。