

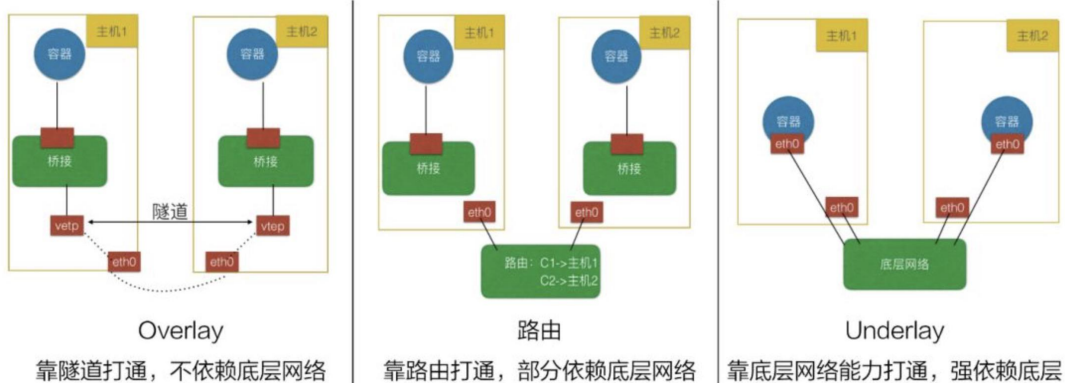
一：容器网络：

<https://kubernetes.io/zh/docs/concepts/cluster-administration/networking/>

容器网络的目的：

- 1.实现同一个 pod 中的不同容器通信(LNMP)
- 2.实现 pod 与 pod 同主机与跨主机的容器通信(微服务)
- 3.pod 和服务之间的通信(nginx 通过调用 tomcat)
- 4.pod 与 k8s 之外的网络通信
外部到 pod（客户端的请求）
pod 到外部（响应报文）

CNI插件通常有三种实现模式



1.1：网络通信方式：

1.1.1：二层通信：

基于目标 mac 地址通信，不可跨局域网通信，通常是由交换机实现报文转发。

private mode:

private 模式下，同一父接口下的子接口之间彼此隔离，不能通信，从外部也无法访问。

vepa(Virtual Ethernet Port Aggregator, 虚拟以太网端口聚合器) mode:

vepa 模式下，子接口之间的通信流量需要导出到外部支持 802.1Qbg/VPEA 功能的交换机上（可以是物理的或者虚拟的），经由外部交换机转发，再绕回来。

bridge mode:

bridge 模式下，模拟的是 Linux bridge 的功能，但比 bridge 要好的一点是每个接口的 MAC 地址是已知的，不用学习，所以这种模式下，子接口之间就是直接可以通信的。

passthru mode(直通模式):

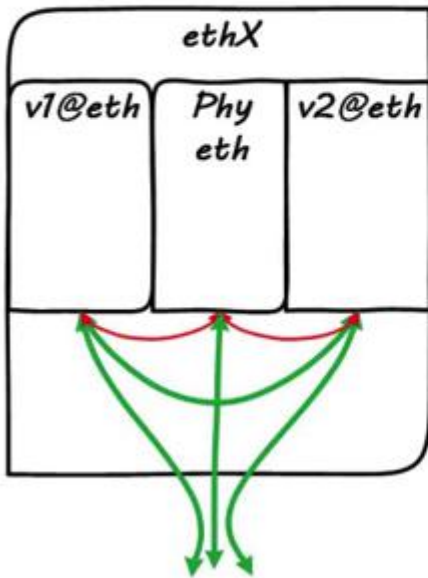
passthru 模式，只允许单个子接口连接父接口。

source mode:

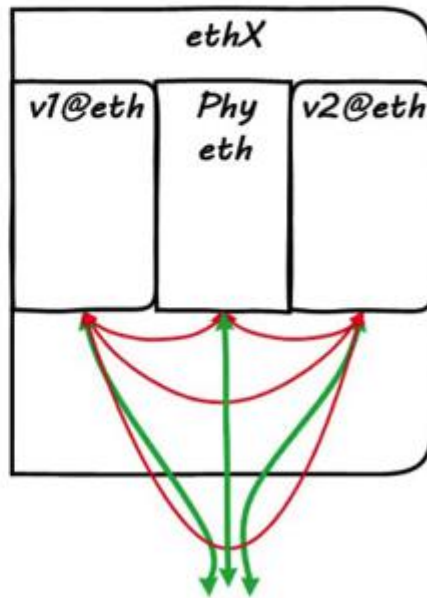
这种模式，只接收源 mac 为指定的 mac 地址的报文。



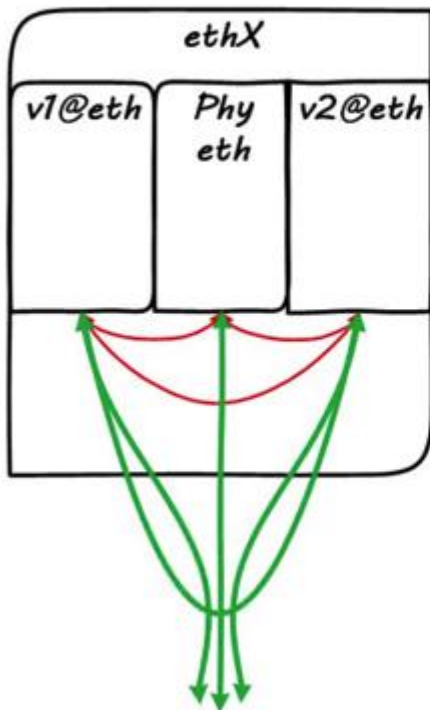
Bridge Macvlan



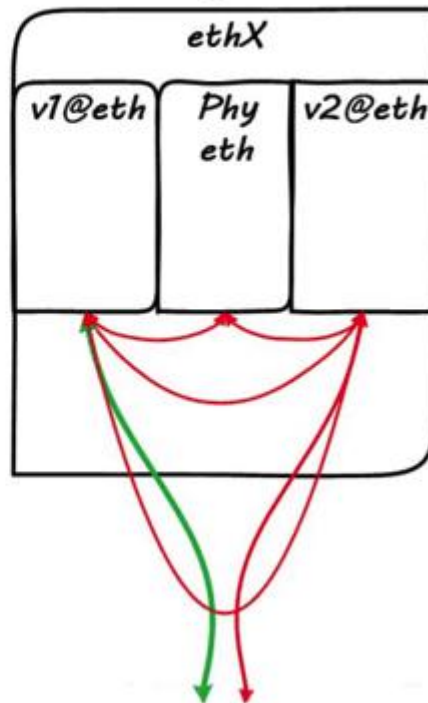
Private Macvlan



VEPA Macvlan

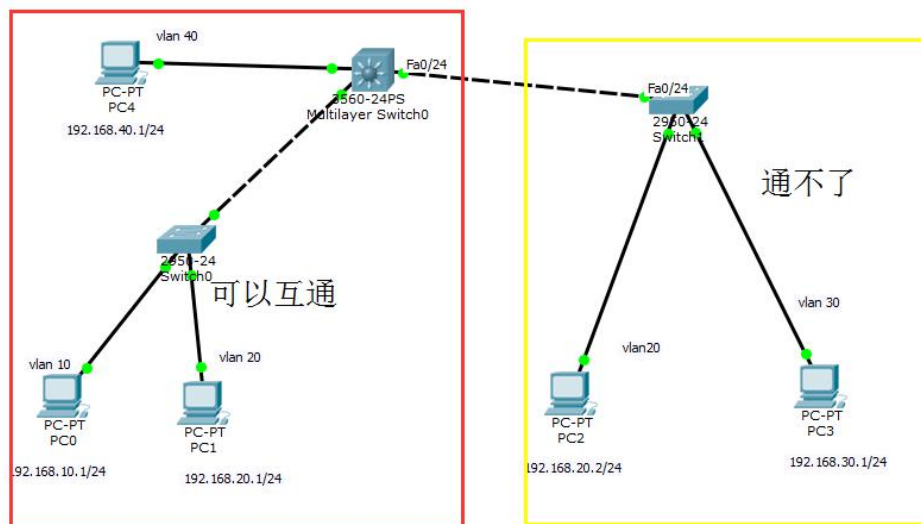
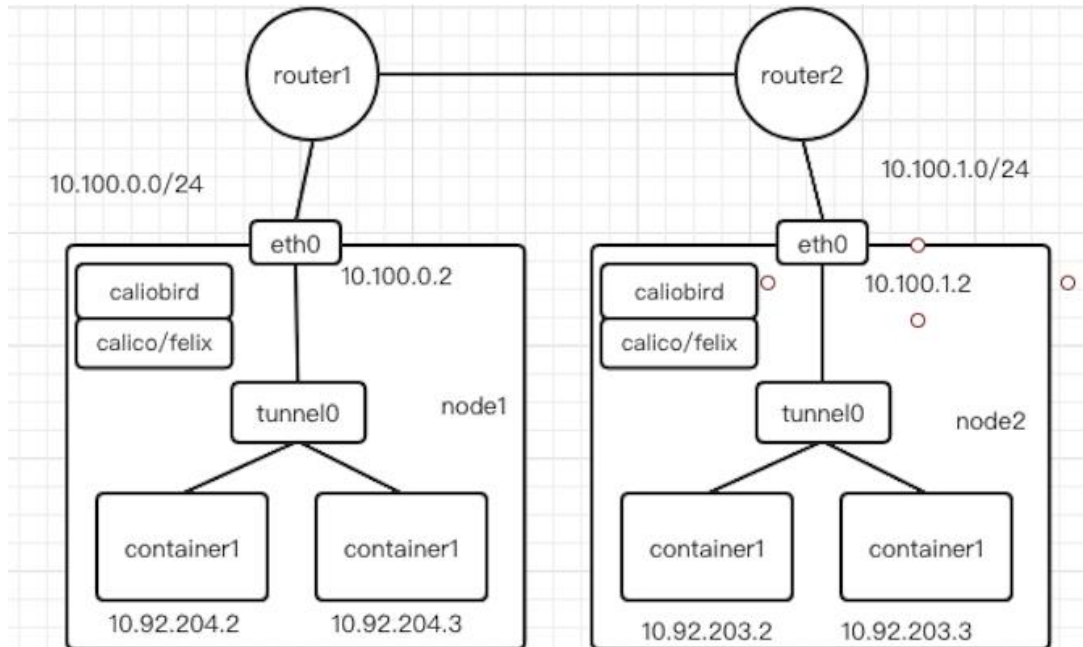


Passthrough Macvlan



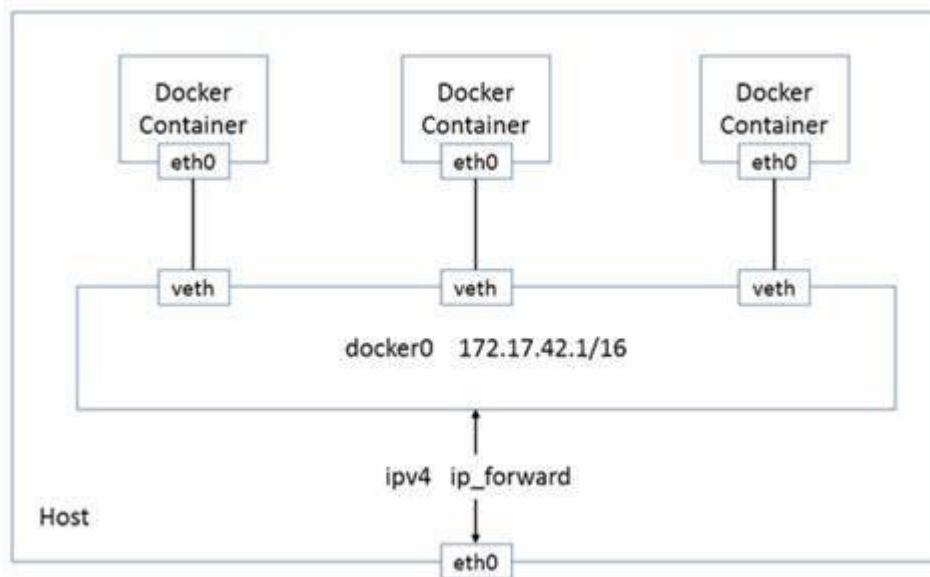
1.1.2: 三层通信:

基于目标 IP 通信，也叫做 IP 交换技术，解决了跨局域网、跨网络通信，通常是由路由器、防火墙、三层交换机等。



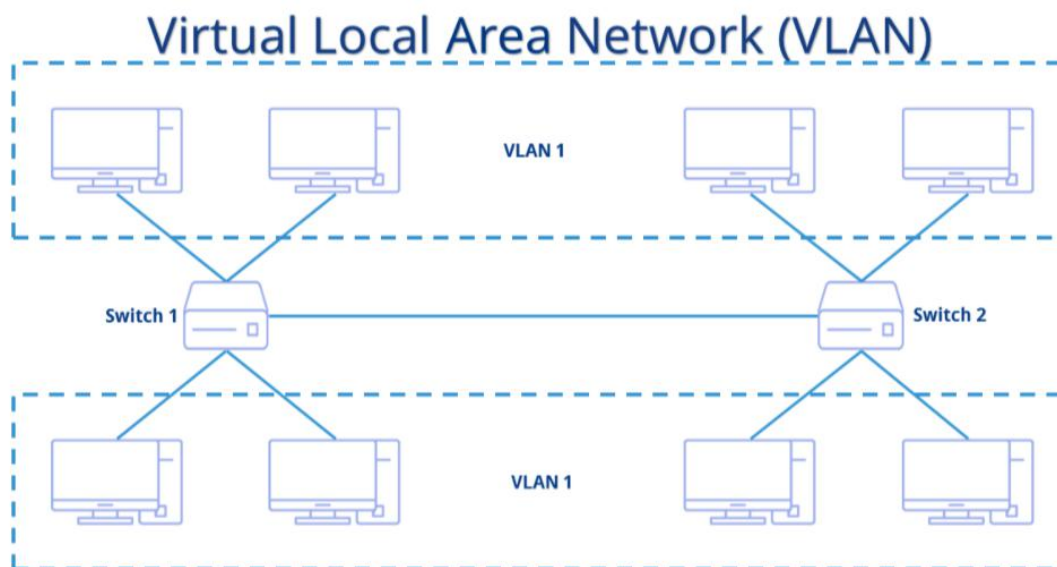
1.1.3: 网桥(bridge):

安装完 docker 之后会默认生成一个 docker 的网桥设备，网桥设备通过 mac 地址转发报文到各个容器，如果容器要访问当前宿主机意外的容器或者网络，则会使用宿主机的路由功能进行源地址转换。



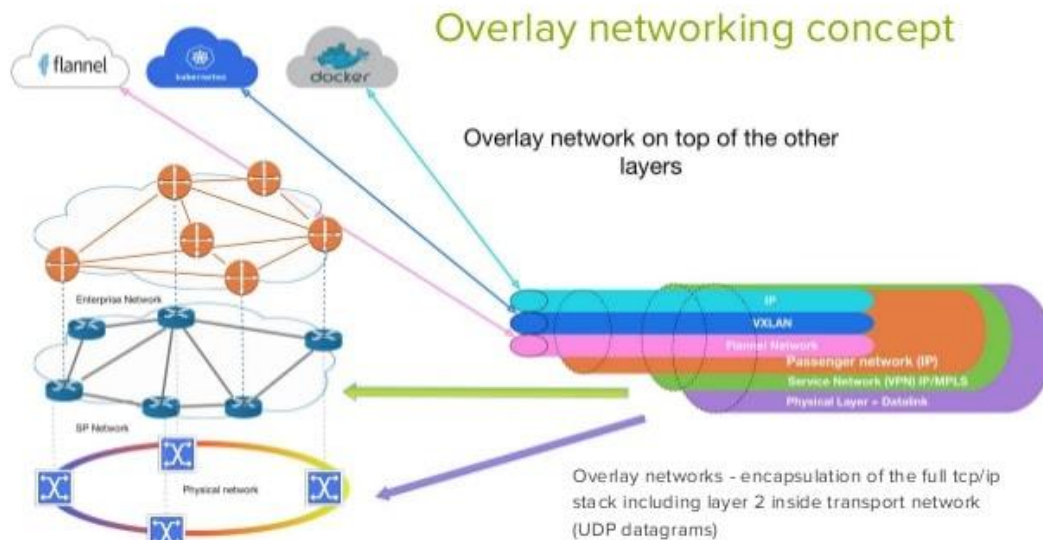
1.1.4: Vlan:

VLAN (Virtual Local Area Network) 即虚拟局域网，是将一个物理(交换机)的网络在逻辑上划分成多个广播域的通信技术，VLAN 内的主机间可以直接通信，而 VLAN 网络外的主机需要通过三层网络设备转发才可以通信，因此一个 vlan 可以将服务器的广播报文限制在一个 VLAN 内，从而降低单个网络环境中的广播报文，vlan 采用 12 位标识 vlan ID，即一个交换机设备最多为 $2^{12}=4096$ 个 vlan。



1.1.5: Overlay 网络简介:

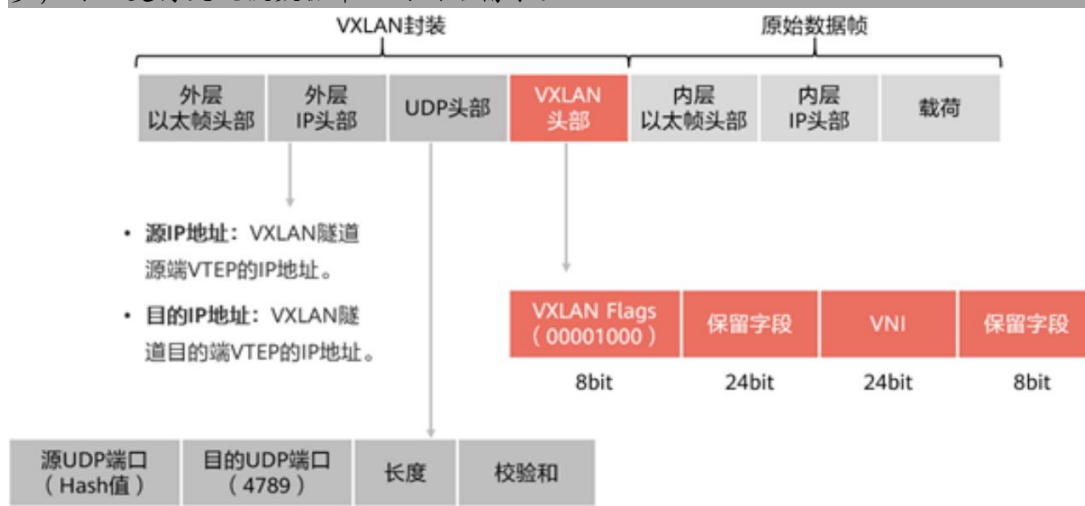
叠加网络或者覆盖网络，在物理网络的基础之上叠加实现新的虚拟网络，即可使网络的中的容器可以相互通信：



4

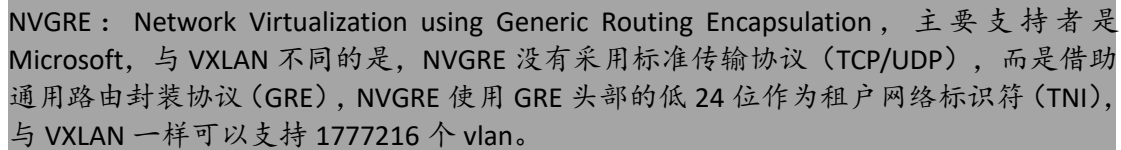
1.1.5.1: overlay 网络实现方式:

VxLAN: VxLAN 全称是 Visual eXtensible Local Area Network (虚拟扩展本地局域网), 主要有 Cisco 推出, vxlan 是一个 VLAN 的扩展协议, 是由 IETF 定义的 NVO3 (Network Virtualization over Layer 3) 标准技术之一, VXLAN 的特点是将 L2 的以太网帧封装到 UDP 报文 (即 L2 over L4) 中, 并在 L3 网络中传输, 即使用 MAC in UDP 的方法对报文进行重新封装, VxLAN 本质上是一种 overlay 的隧道封装技术, 它将 L2 的以太网帧封装成 L4 的 UDP 数据报, 然后在 L3 的网络中传输, 效果就像 L2 的以太网帧在一个广播域中传输一样, 实际上 L2 的以太网帧跨越了 L3 网络传输, 但是缺不受 L3 网络的限制, vxlan 采用 24 位标识 vlan ID 号, 因此可以支持 $2^{24}=16777216$ 个 vlan, 其可扩展性比 vlan 强大的多, 可以支持大规模数据中心的网络需求。



VTEP(VXLAN Tunnel Endpoint vxlan 隧道端点), VTEP 是 VXLAN 网络的边缘设备, 是 VXLAN 隧道的起点和终点, VXLAN 对用户原始数据帧的封装和解封装均在 VTEP 上进行, 用于 VXLAN 报文的封装和解封装, VTEP 与物理网络相连, 分配的地址为物理网 IP 地址, VXLAN 报文中源 IP 地址为本节点的 VTEP 地址, VXLAN 报文中目的 IP 地址为对端节点的 VTEP 地址, 一对 VTEP 地址就对应着一个 VXLAN 隧道, 服务器上的虚拟交换机(隧道 flannel.1

VNI (VXLAN Network Identifier) : VXLAN 网络标识 VNI 类似 VLAN ID,用于区分 VXLAN 段,不同 VXLAN 段的虚拟机不能直接二层相互通信,一个 VNI 表示一个租户,即使多个终端用户属于同一个 VNI,也表示一个租户。



```
root@k8s-node1:~# tcpdump udp port 8472
```

Underlay 网络就是传统 IT 基础设施网络，由交换机和路由器等设备组成，借助以太网协议、路由协议和 VLAN 协议等驱动，它还是 Overlay 网络的底层网络，为 Overlay 网络提供数据通信服务。容器网络中的 Underlay 网络是指借助驱动程序将宿主机的底层网络接口直接暴露

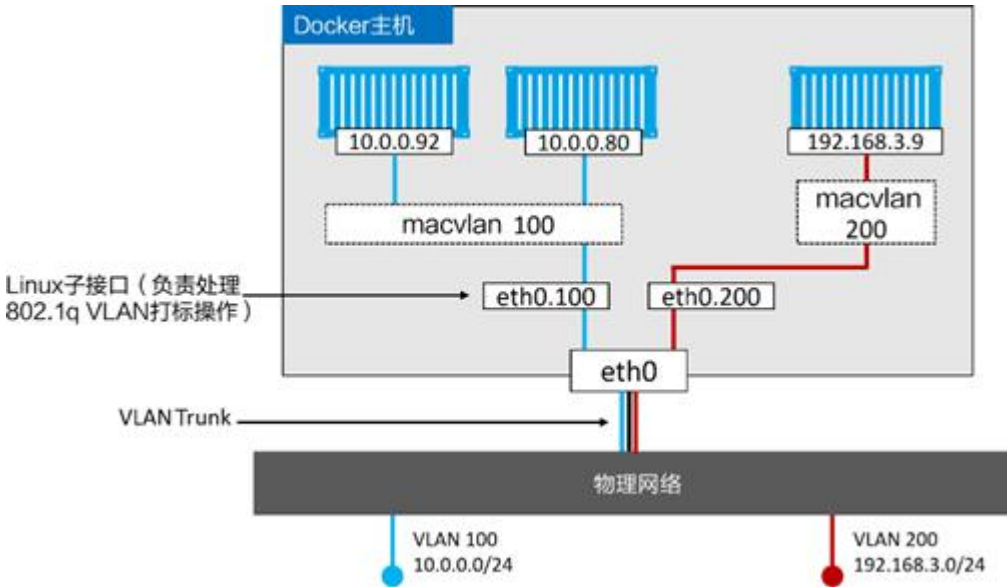
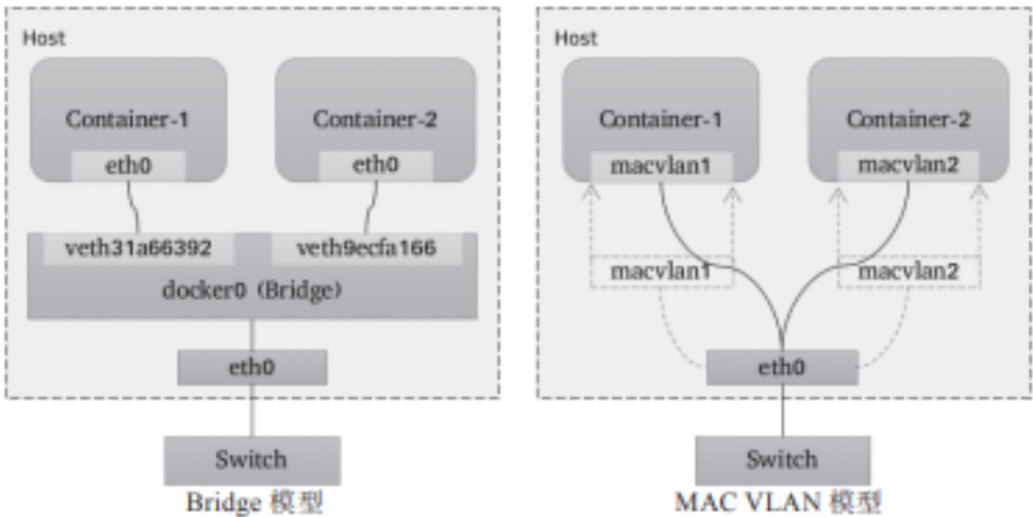
给容器使用的一种网络构建技术，较为常见的解决方案有 MAC VLAN、IP VLAN 和直接路由等。

Underlay 依赖于网络网络进行跨主机通信。

1.1.6.1: bridge 与 macvlan 模式:

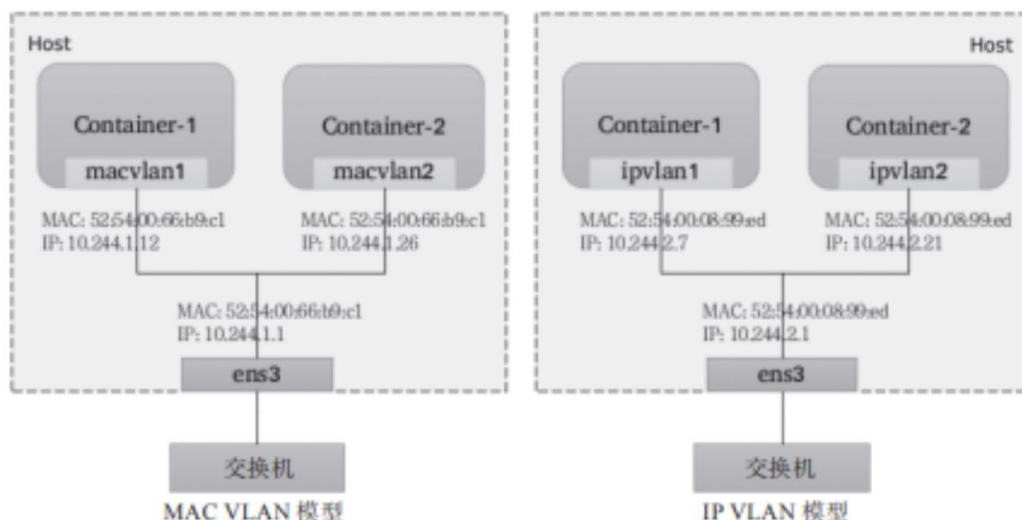
Bridge: 桥接模式

MAC VLAN: 支持在同一个以太网接口上虚拟出多个网络接口(子接口), 每个虚拟接口都拥有唯一的 MAC 地址并可配置网卡子接口 IP。



1.1.6.2: IP VLAN:

IP VLAN 类似于 MAC VLAN, 它同样创建新的虚拟网络接口并为每个接口分配唯一的 IP 地址, 不同之处在于, 每个虚拟接口将共享使用物理接口的 MAC 地址, 从而不再违反防止 MAC 欺骗的交换机的安全策略, 且不要求在物理接口上启用混杂模式



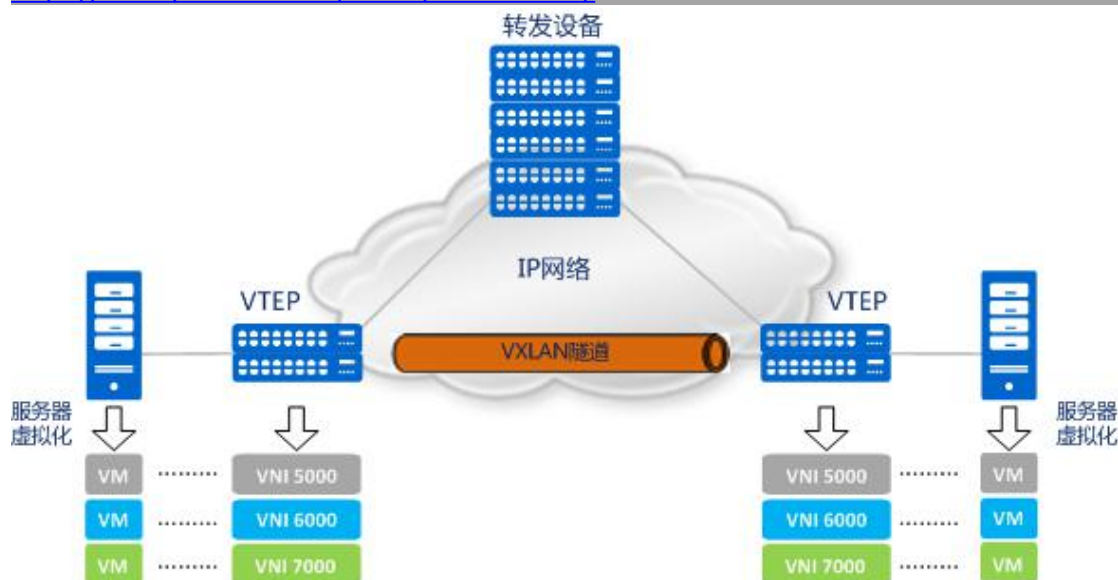
IP VLAN 有 L2 和 L3 两种模型，其中 IP VLAN L2 的工作模式类似于 MAC VLAN 被用作网桥或交换机，而 IP VLAN L3 模式中，子接口地址不一样，但是公用宿主机的 MAC 地址。虽然支持多种网络模型，但 MAC VLAN 和 IP VLAN 不能同时在同一物理接口上使用。一般使用 MAC VLAN。

Linux 内核自 4.2 版本后才支持 IP VLAN。

1.2: VXLAN 通信过程:

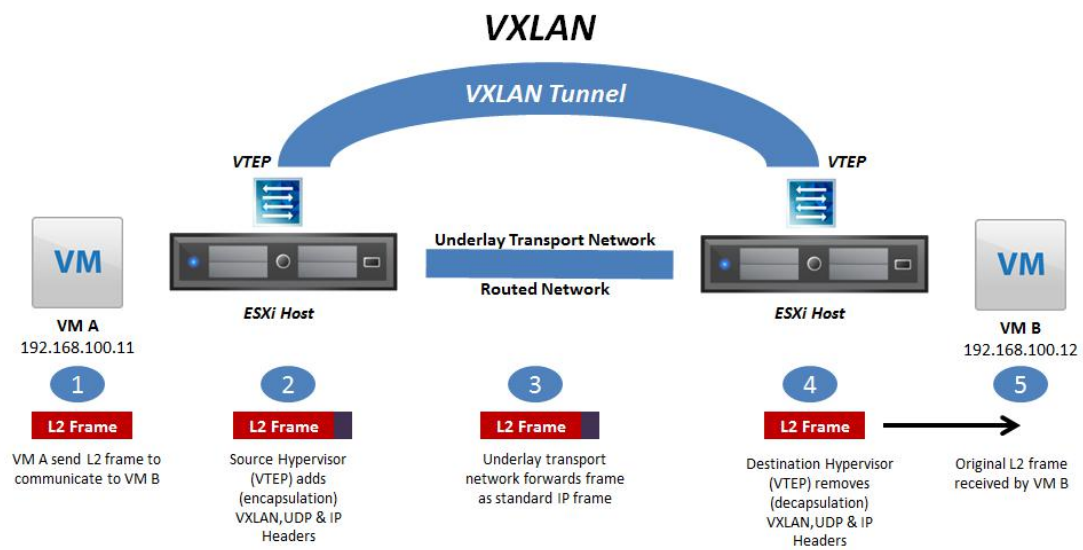
<https://support.huawei.com/enterprise/zh/doc/EDOC1100087027>

<https://www.pianshen.com/article/1890293327/>

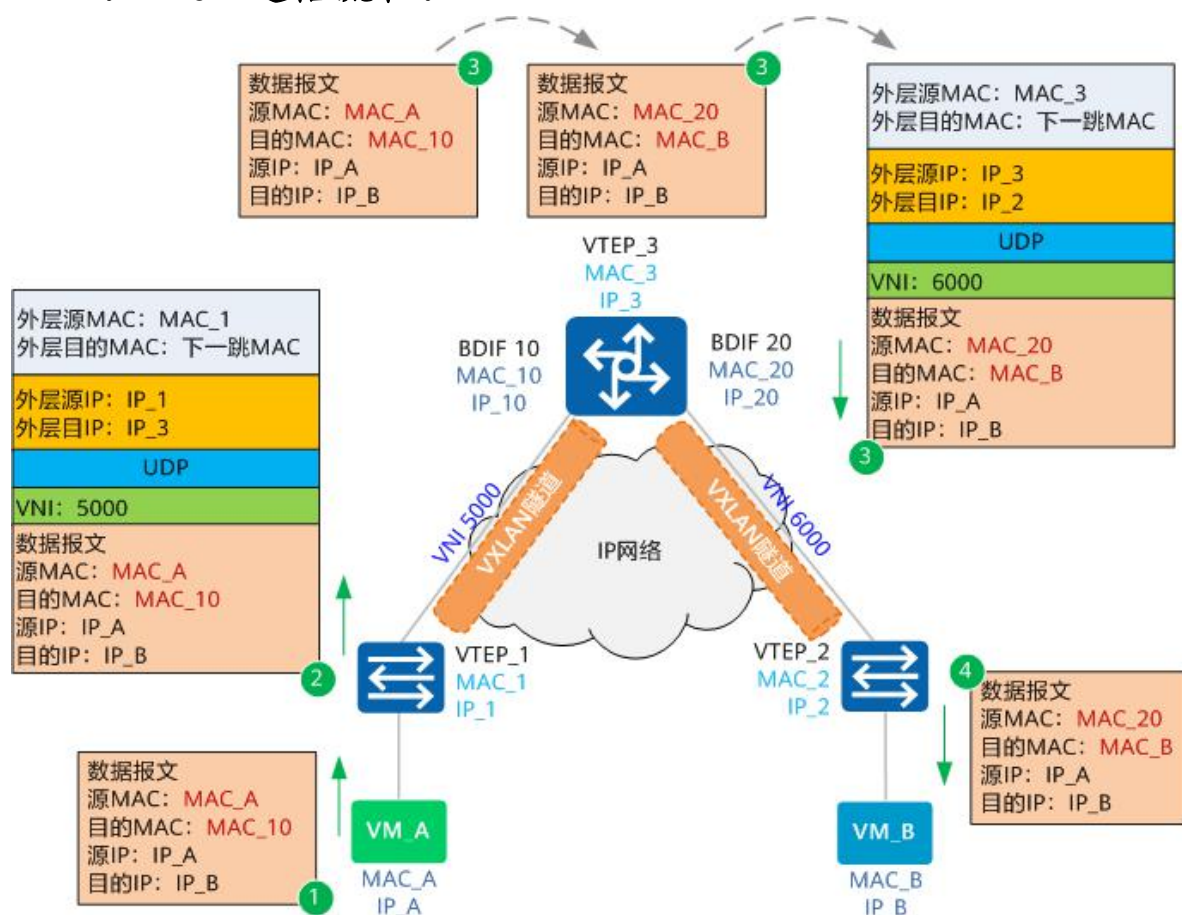


1.2.1: vxlan 简单通信流程:

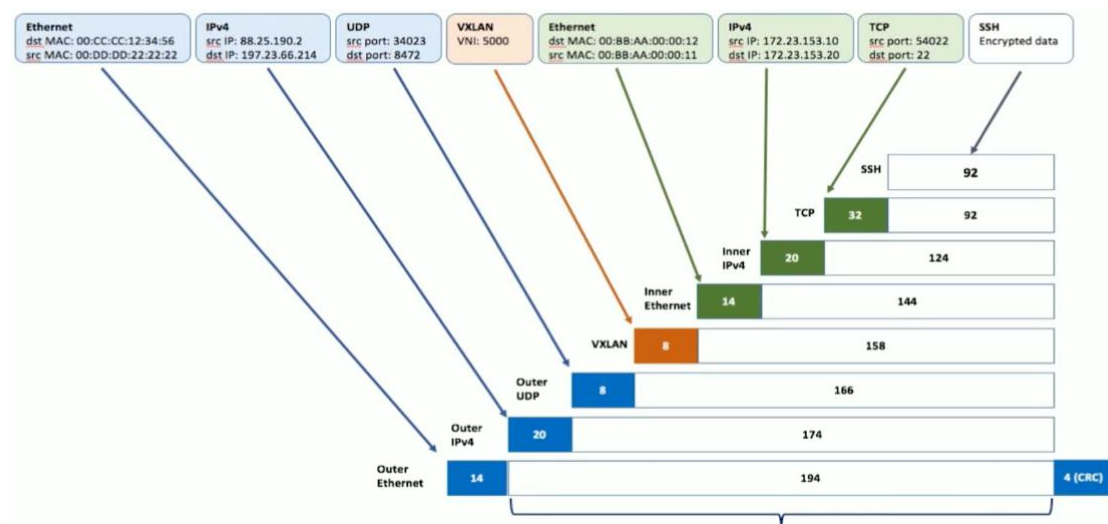
- 1.VM A 发送 L2 帧与 VM 请求与 VM B 通信。
- 2.源宿主机 VTEP 添加或者封装 VXLAN、UDP 及 IP 头部报文。
- 3.网络层设备将封装后的报文通过标准的报文在三层网络进行转发到目标主机。
- 4.目标宿主机 VTEP 删除或者解封装 VXLAN、UDP 及 IP 头部。
- 5.将原始 L2 帧发送给目标 VM。



1.2.2: vxlan 通信流程:



1.3.3: vxlan 报文格式:



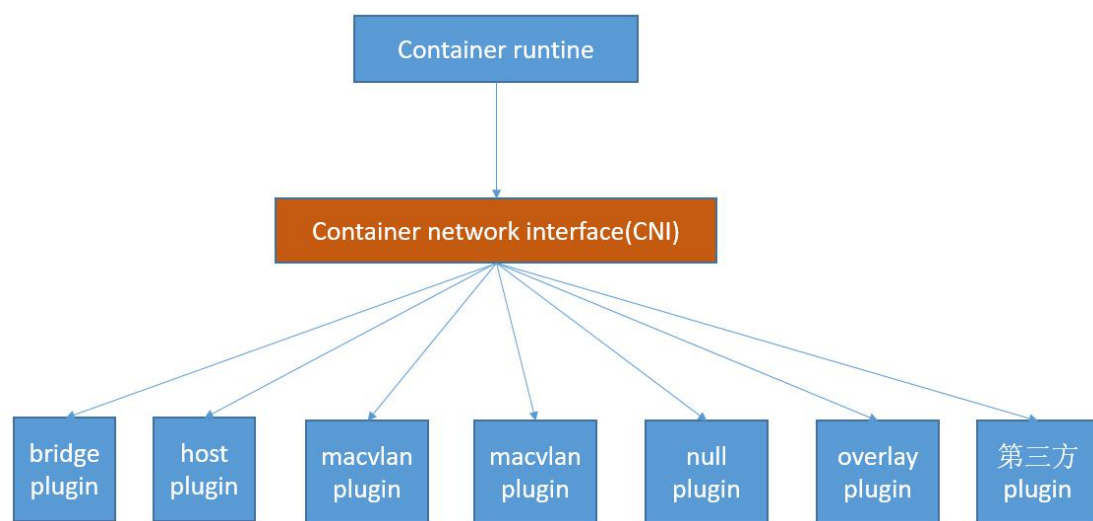
1.3: Docker 对网络的支持:

Container network interface (CNI)

```
root@docker-node2:~# docker info
```

```
Containers: 2
```

Running: 1
Paused: 0
Stopped: 1
Images: 2
Server Version: 18.09.9
Storage Driver: overlay2
Backing Filesystem: extfs
Supports d_type: true
Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
Volume: local
Network: bridge host macvlan null overlay



1.4: Docker 跨主机通信方案总结:

Bridge 网络, docker0 就是默认的桥接网络

Docker 网络驱动:

Overlay: 基于 VXLAN、NVGRE 等封装技术实现 overlay 叠加网络

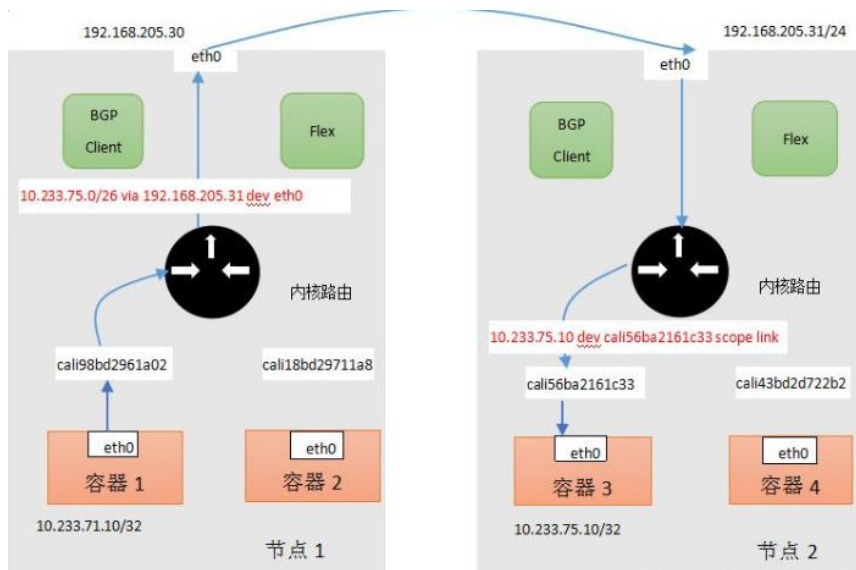
Macvlan: 基于 Docker 宿主机物理网卡的不同子接口实现多个虚拟 vlan, 一个子接口就是一个虚拟 vlan, 容器通过宿主机的路由功能和外网保持通信。

1.4.1: 第三方项目:

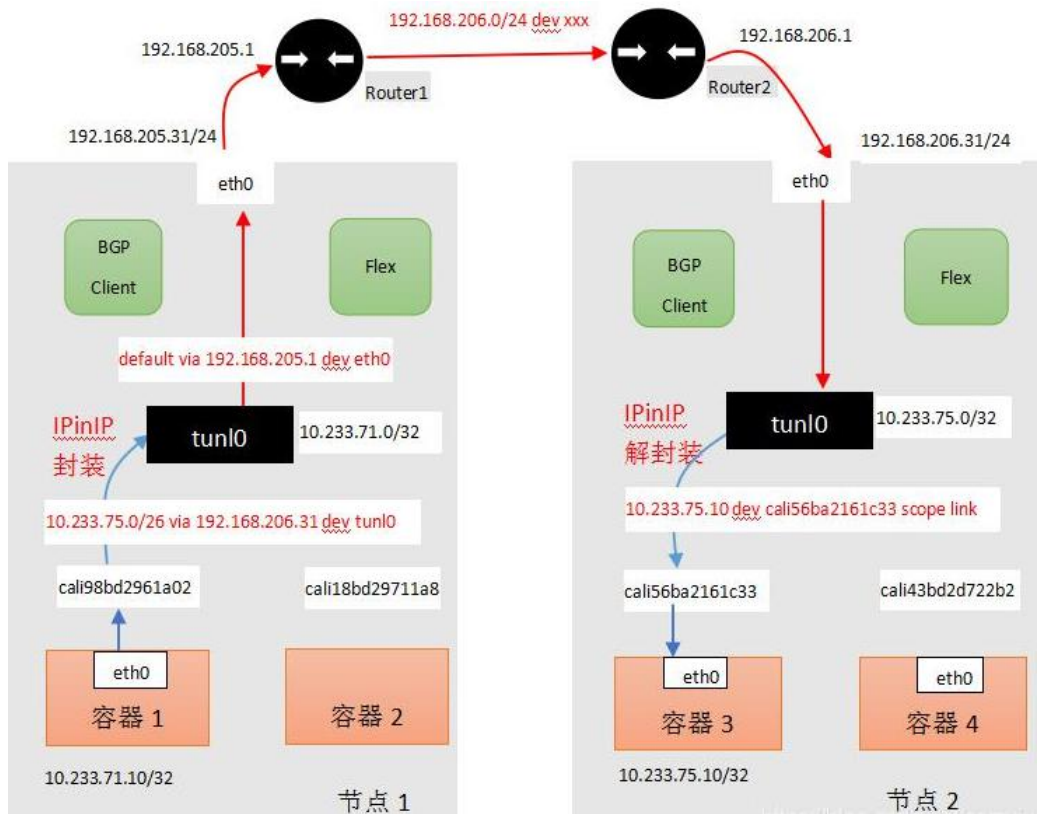
1.4.1.1: Calico—BGP+IPIP:

已被 github 等公司用于生产环境

未开启 IPIP:



已开启 IPinP:



1.4.1.2: Flannel:

Flannel—VXLAN(内核 3.12 或以上)/UDP

1.4.1.3: OpenvSwitch:

OpenvSwitch—VXLAN(内核 3.12 或以上)/UDP

1.4.1.4: Canal:

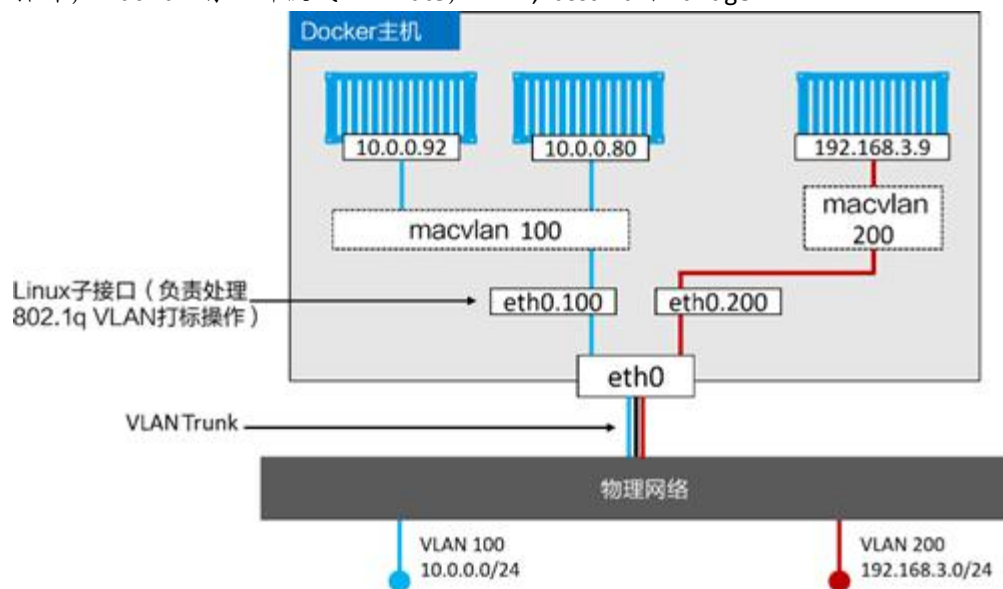
Canal—合并 flannel 和 calico 的功能

1.4.1.5 Weave:

Weave—VXLAN(内核 3.12 或以上)和 UDP

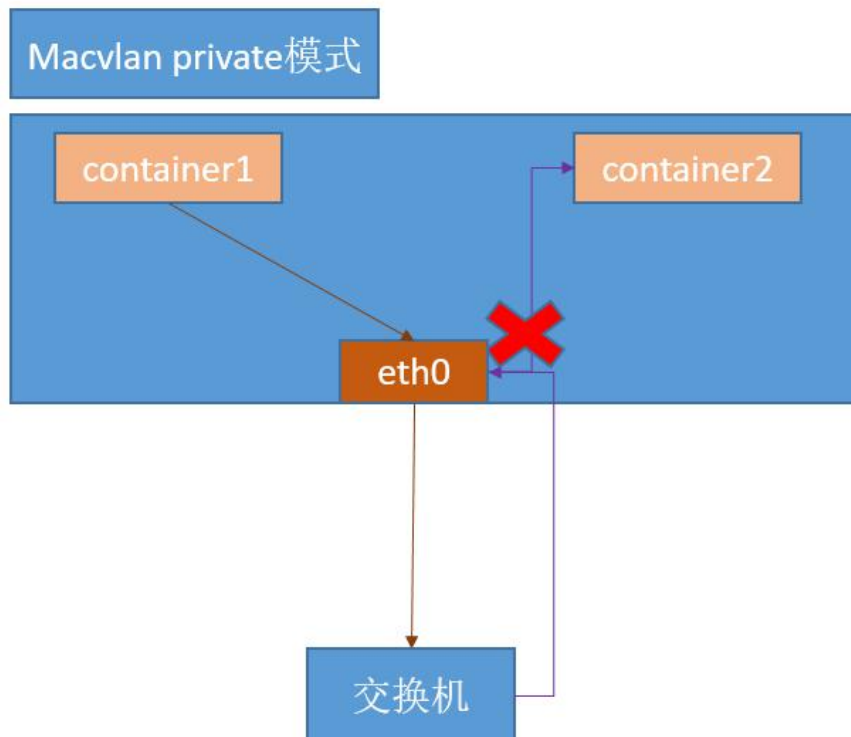
二： Docker 多主机通信之 macvlan:

macvlan 是 Linux kernel 支持的特性，macvlan 类似于宿主机网卡子接口，但相比网卡子接口来说，macvlan 接口拥有自己独立的 mac 地址，因此使用 macvlan 接口可以允许更多的二层操作，macvlan 有四种模式：Private, VEPA, Passthru 和 bridge



2.1: Private(私有)模式:

在 Private 模式下，同一个宿主机下的容器不能通信，即使通过交换机再把数据报文转发回来也不行。



```
root@docker-node1:~# docker network create -d macvlan --subnet=172.31.0.0/21
--gateway=172.31.7.254 -o parent=eth0 -o macvlan_mode=private
jiede_macvlan_private

root@docker-node2:~# docker run -it --rm --net=jiede_macvlan_private
--name=container4 --ip=172.31.5.222 centos:7.7.1908 /bin/bash
root@docker-node2:~# docker run -it --rm --net=jiede_macvlan_private
--name=container5 --ip=172.31.5.223 centos:7.7.1908 /bin/bash
[root@87e973b98d3f /]# ping 172.31.5.222
PING 172.31.5.222 (172.31.5.222) 56(84) bytes of data.
```

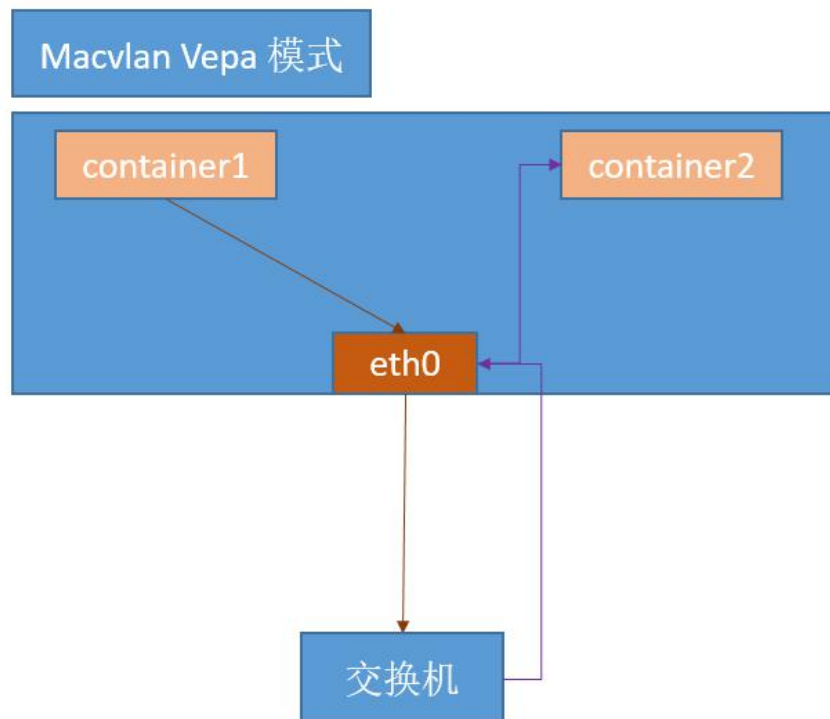
验证 private 模式 macvlan 网络类型:

```
root@docker-node2:~# docker network inspect jiede_macvlan_private
```

```
{
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {},
  "Options": {
    "macvlan_mode": "private",
    "parent": "eth0"
  },
  "Labels": {}
}
]
root@docker-node2:~#
```

2.2: VEPA 模式:

虚拟以太网端口汇聚器（Virtual Ethernet Port Aggregator，简称 VEPA），在这种模式下，macvlan 内的容器不能直接接收在同一个物理网卡的容器的请求数据包，但是可以经过交换机的(端口回流)再转发回来可以实现通信。



```
#docker network create -d macvlan --subnet=172.31.6.0/21 --gateway=172.31.7.254 -o
parent=eth0 -o macvlan_mode=vepa jiege_macvlan_vepa
#docker run -it --net=jiege_macvlan_vepa --ip=172.31.6.201 --name=container1
centos:7.8.2003 /bin/bash
#docker run -it --net=jiege_macvlan_vepa --ip=172.31.6.202 --name=container2
centos:7.8.2003 /bin/bash
```

验证 vepa 模式 macvlan 网络类型:

```
root@docker-node2:~# docker network inspect jiege_macvlan_vepa
```

```
{
  "Internal": false,
  "Attachable": false,
  "Ingress": false,
  "ConfigFrom": {
    "Network": ""
  },
  "ConfigOnly": false,
  "Containers": {},
  "Options": {
    "macvlan_mode": "vepa",
    "parent": "eth0"
  },
  "Labels": {}
}
]
root@docker-node2:~#
```

2.3: passthru(直通)模式:

Passthru 模式下该 macvlan 只能创建一个容器,当运行一个容器后再创建其他容器则会报错。

```
root@docker-node2:~# docker network create -d macvlan --subnet=172.31.0.0/21
--gateway=172.31.7.254 -o parent=eth0 -o macvlan_mode=passthru jiege_macvlan_passthru
3b996c342bdb7f1735a04ae8a24f26ff6703a5222e1524c2f72dbb433cb66d3e
```

只能运行一个容器, 再创建第二个容器将会保存如下:

```
root@docker-node2:~# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
fc6aaa79d70d        bridge              bridge              local
46d04c94fd5e        host                host                local
3b996c342bdb        jiege_macvlan_passthru macvlan             local
8f4948e4e186        none                null                local
root@docker-node2:~# docker ps
CONTAINER ID        IMAGE               COMMAND              CREATED            STATUS
PORTS              NAMES
866a3ee05f91        centos:7.7.1908    "/bin/bash"         2 minutes ago     Up 2 minutes
container1
root@docker-node2:~# docker run -it --rm --net=jiege_macvlan_passthru centos:7.7.1908
docker: Error response from daemon: failed to create the macvlan port: invalid argument.
root@docker-node2:~# _
```

2.4: bridge 模式:

在 bridge 这种模式下, 使用同一个宿主机网络的 macvlan 容器可以直接通信。

2.4.1: node1:

```
root@docker-node1:~# docker network create -d macvlan --subnet=172.31.0.0/21
--gateway=172.31.7.254 -o parent=eth0 -o macvlan_mode=bridge jiege_macvlan_bridge
7b456dba3d306c4b55f943dbe766349aedec86122f0387e03f4e45b7ae578b6e
```

创建两个容器验证单机通信:

```
root@docker-node1:~# docker run -it --net=jiege_macvlan_bridge --ip=172.31.6.201
--name=container1 centos:7.8.2003 /bin/bash
```

```
root@docker-node1:~# docker run -it --net=jiege_macvlan_bridge --ip=172.31.6.202
--name=container2 centos:7.8.2003 /bin/bash
```

```
[root@f8ae0c8dcdf8 /]# yum install net-tools -y
```

```
[root@f8ae0c8dcdf8 /]# ping 172.31.6.201
```

```
PING 172.31.6.201 (172.31.6.201) 56(84) bytes of data.
```

```
64 bytes from 172.31.6.201: icmp_seq=1 ttl=64 time=0.030 ms
```

```
64 bytes from 172.31.6.201: icmp_seq=2 ttl=64 time=0.036 ms
```

验证网络类型:

```
root@docker-node2:~# docker network inspect jiege_macvlan_vep
```

```

    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
        "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {
        "macvlan_mode": "bridge",
        "parent": "eth0"
    },
    "Labels": {}
}
]
root@docker-node2:~#

```

2.4.2: node2:

```

root@docker-node2:~# docker network create -d macvlan --subnet=172.31.0.0/21
--gateway=172.31.7.254 -o parent=eth0 -o macvlan_mode=bridge jiege_macvlan_bridge
1d2fe27b44fb3f6aafd371ff8ad1abe2f7fb0c87778af7ff16786341151b15c5

```

创建两个容器验证单机通信:

```

root@docker-node2:~# docker run -it --net=jiege_macvlan_bridge --ip=172.31.6.203
--name=container1 centos:7.8.2003 /bin/bash
[root@daa4d05fc762 /]# yum install net-tools -y
root@docker-node2:~# docker run -it --net=jiege_macvlan_bridge --ip=172.31.6.204
--name=container2 centos:7.8.2003 /bin/bash
[root@daa4d05fc762 /]# ping 172.31.6.203
PING 172.31.6.203 (172.31.6.203) 56(84) bytes of data.
64 bytes from 172.31.6.203: icmp_seq=1 ttl=64 time=0.018 ms
64 bytes from 172.31.6.203: icmp_seq=2 ttl=64 time=0.032 ms

```

验证跨主机容器通信:

```

[root@daa4d05fc762 /]# ping 172.31.6.201
PING 172.31.6.201 (172.31.6.201) 56(84) bytes of data.
64 bytes from 172.31.6.201: icmp_seq=1 ttl=64 time=1.62 ms
64 bytes from 172.31.6.201: icmp_seq=2 ttl=64 time=0.350 ms

```

注:

在部分服务器需要开启网卡混杂模式, 混杂模式 (Promiscuous Mode) 是指一台服务器能够接收所有经过它的数据流, 而不论其目的地址是否是当前服务器。

```
# ifconfig eth0 promisc
```

2.5: 一个服务器如果多个 macvlan:

如何在同一个 linux 服务器配置多个 macvlan:

2.5.1: 将服务器对应的交换机接口设置为 trunk 模式(如果是 access 模式)

```

#interface fastEthernet 0/1
#switchport mode trunk

```

2.5.2: Linux 服务器添加网卡子接口

```
root@docker-node2:~# ifconfig eth0:1 172.31.2.202 netmask 255.255.248.0
```

2.5.3: 在 Linux 网卡子接口添加新的 macvlan:

```
root@docker-node2:~# docker network create -d macvlan --subnet=172.20.0.0/21
--gateway=172.20.7.254 -o parent=eth0.1 -o macvlan_mode=private
jiege_macvlan_private1
b8f859a2ece834655da5b7c0601afc1bf07f6cc1e51a9a418dd33f2298967e68
```

三： Docker 多主机通信之 overlay:

3.1: 基础环境:

Docker 基于 overlay 通信需要满足两个条件中的任意一个:

Docker 主机运行在 swarn 模式

使用外部键值存储容器数据, kubernetes 环境中使用此方式

3.2: 使用外部键值存储基础环境:

各 docker 主机名必须唯一, 因为键值存储服务使用主机名区分集群中的 node 节点
内核版本推荐 3.10 或以上

各主机安装较新版本的 docker

安装键值存储服务器, docker 目前支持的键值存储服务有 etcd、zookeeper、consul

3.3: 部署键值存储服务 consul 集群:

<https://github.com/hashicorp/consul> #官方网站

<https://www.consul.io/downloads.html> #下载地址

<https://releases.hashicorp.com/consul/>

3.4: 单机 consul:

3.4.1: 安装单机 consul:

```
# unzip consul_1.7.3_linux_amd64.zip
# mv consul /usr/bin/
# mkdir /var/lib/consul
# consul agent -server -bootstrap-expect 1 -data-dir /var/lib/consul -node 172.31.7.221
-bind=172.31.7.221 -client=0.0.0.0 -datacenter BJ-SJHL -ui
```

Agent: agent 是 consul 的核心进程, 负责维护成员资格信息、注册服务、运行检查、响应查询等, consul agent 要在 Consul 群集中的每个节点上运行。

Server: 以 server 身份启动

bootstrap-expect NUM: 集群要求的 server 数量。

data-dir: 数据目录。

node: 节点 id, 集群中的每个 node 必须有一个唯一的名称, 默认情况下 Consul 使用机器的 hostname

bind: 监听的 ip 地址, 默认绑定 0.0.0.0。

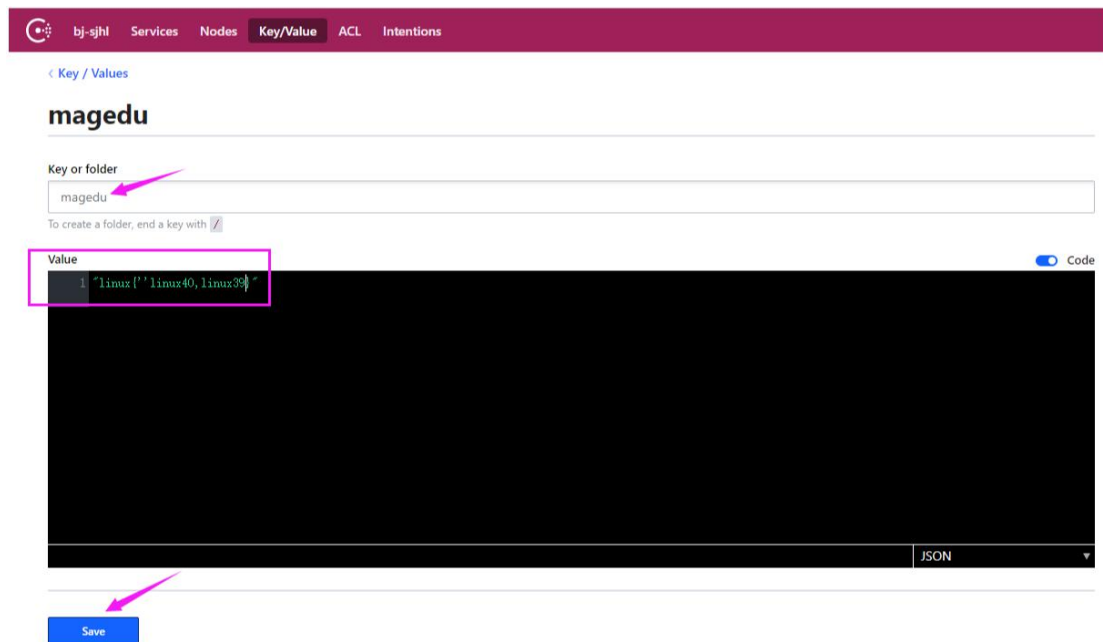
client: 允许那些客户端 IP 地址访问, 0.0.0.0 是指任何地址都可以访问 (不加这个, 下面的 ui:8500 无法访问)

ui: 启动 web 界面。端口是 IP:8500

-config-dir 指定配置目录, Consul 会加载其中的所有配置文件

-datacenter 指定数据中心名称, 默认是 dc1

3.4.2: 访问 web 界面:



The screenshot shows the Consul web interface. At the top, there's a navigation bar with tabs: 'bj-sjhl', 'Services', 'Nodes', 'Key/Value' (selected), 'ACL', and 'Intentions'. Below the navigation bar, there's a breadcrumb link '< Key / Values'. The main heading is 'magedu'. Underneath, there's a 'Key or folder' input field containing 'magedu'. A pink arrow points to this field. Below the input field, there's a note: 'To create a folder, end a key with /'. The 'Value' section is highlighted with a pink box. It contains a text area with the value '1 "linux('linux40,linux3%'. A pink arrow points to the 'Save' button at the bottom left. The 'Code' toggle is turned on. The output format is set to 'JSON'.

3.4.3: consul service 文件:

```
# cat /lib/systemd/system/consul.service
[Unit]
Description=Consul Server
Documentation=https://www.consul.io
After=network.target

[Service]
Restart=on-failure
WorkingDirectory=/var/lib/consul
ExecStart=/usr/bin/consul agent -server -bootstrap-expect 1 -data-dir /var/lib/consul -node
172.31.7.221 -bind=172.31.7.221 -client=0.0.0.0 -datacenter bj-sjhl-ui

[Install]
WantedBy=multi-user.target
```

3.4.4: 通过 service 文件启动 consul:

```
# systemctl daemon-reload
# systemctl start consul
# systemctl enable consul
```

3.5: 配置 docker 使用 consul:

配置 docker 使用 consul

3.5.1: docker 配置文件:

各 docker 节点配置文件修改如下:

```
# vim /lib/systemd/system/docker.service

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375
--containerd=/run/containerd/containerd.sock --cluster-store consul://172.31.7.221:8500
--cluster-advertise 172.31.6.101:2375
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always

# systemctl daemon-reload && systemctl daemon-reload
```

3.5.2: consul 界面验证数据:

← → ↻ ⓘ 不安全 | 172.31.7.221:8500/ui/bj-sjhl/kv/docker/nodes/

bj-sjhl Services Nodes **Key/Value** ACL Intentions

< Key / Values < docker

nodes

Name
172.31.6.101:2375
172.31.6.102:2375
172.31.6.103:2375

3.5.3.1: node1 创建 overlay 网络:

```
root@docker-node1:~# docker network create -d overlay --subnet 10.100.0.0/20 --gateway 10.100.15.254 jiege_overlay_docker_net
70d4117d0346c16613eafcada9363cfd9a2aea3cf6ee0795a0faf1c8483b1932

root@docker-node1:~# docker network list
```

NETWORK ID	NAME	DRIVER	SCOPE
78dde2d630d5	bridge	bridge	local
ee1c5ec72458	docker_gwbridge	bridge	local
f09aecaca870	host	host	local
70d4117d0346	jiege_overlay_docker_net	overlay	global
07333c1e0b75	multihost	overlay	global
3b60d9e1a73b	none	null	local

3.6.3.2: 验证 overlay 网络

```
root@docker-node1:~# docker network inspect jiege_overlay_docker_net
[
  {
    "Name": "jiege_overlay_docker_net",
    "Id": "70d4117d0346c16613eafcada9363cfd9a2aea3cf6ee0795a0faf1c8483b1932",
    "Created": "2020-05-11T17:12:53.251660763+08:00",
    "Scope": "global",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {
          "Subnet": "10.100.0.0/20",
          "Gateway": "10.100.15.254"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {},
    "Options": {},
    "Labels": {}
  }
]
```

3.6: 创建容器并验证容器夸主机通信:

在不通的 docker 宿主机分别创建容器，验证当前宿主机的容器通信以及夸宿主机的容器通信是否正常。

3.6.1: 验证当前宿主机容器通信:

分别启动两个终端，每个终端创建一个容器，验证能否正常通信。

3.6.1.1: 启动第一个容器:

```
root@docker-node1:~# docker run -it --net=jiege_overlay_docker_net busybox sh
/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:64:00:01
          inet addr:10.100.0.1  Bcast:10.100.15.255  Mask:255.255.240.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

3.6.1.2: 启动第二个容器并验证:

```
root@docker-node1:~# docker run -it --net=jiege_overlay_docker_net busybox sh
/# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 02:42:0A:64:00:02
```

```
inet addr:10.100.0.2 Bcast:10.100.15.255 Mask:255.255.240.0
UP BROADCAST RUNNING MULTICAST MTU:1450 Metric:1
RX packets:7 errors:0 dropped:0 overruns:0 frame:0
TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:574 (574.0 B) TX bytes:532 (532.0 B)

/ # ping 10.100.0.1
PING 10.100.0.1 (10.100.0.1): 56 data bytes
64 bytes from 10.100.0.1: seq=0 ttl=64 time=0.078 ms
64 bytes from 10.100.0.1: seq=1 ttl=64 time=0.050 ms
64 bytes from 10.100.0.1: seq=2 ttl=64 time=0.053 ms
64 bytes from 10.100.0.1: seq=3 ttl=64 time=0.053 ms
```

3.6.1.2: 验证外网通信:

```
root@docker-node1:~# docker run -it --net=jiege_overlay_docker_net busybox sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:64:00:01
          inet addr:10.100.0.1 Bcast:10.100.15.255 Mask:255.255.240.0
          UP BROADCAST RUNNING MULTICAST MTU:1450 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2 Bcast:172.18.255.255 Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:836 (836.0 B) TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

/ # ping www.magedu.com #验证外网通信
PING www.magedu.com (101.200.188.230): 56 data bytes
64 bytes from 101.200.188.230: seq=0 ttl=127 time=6.471 ms
64 bytes from 101.200.188.230: seq=1 ttl=127 time=6.898 ms
64 bytes from 101.200.188.230: seq=2 ttl=127 time=6.599 ms
```

3.6.1.3: 验证夸主机通信:

在 docker node2 创建容器, 验证夸主机的容器通信

```
root@docker-node2:~# docker run -it --net=jiege_overlay_docker_net busybox sh
/ # ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 02:42:0A:64:00:03
          inet addr:10.100.0.3  Bcast:10.100.15.255  Mask:255.255.240.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping 10.100.0.2
PING 10.100.0.2 (10.100.0.2): 56 data bytes
64 bytes from 10.100.0.2: seq=0 ttl=64 time=1.499 ms
64 bytes from 10.100.0.2: seq=1 ttl=64 time=0.432 ms
64 bytes from 10.100.0.2: seq=2 ttl=64 time=0.465 ms
64 bytes from 10.100.0.2: seq=3 ttl=64 time=0.340 ms
^C
--- 10.100.0.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.340/0.684/1.499 ms
/ # ping 10.100.0.1
PING 10.100.0.1 (10.100.0.1): 56 data bytes
64 bytes from 10.100.0.1: seq=0 ttl=64 time=0.526 ms
64 bytes from 10.100.0.1: seq=1 ttl=64 time=0.354 ms
^C
--- 10.100.0.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 0.354/0.440/0.526 ms
/ #
```

3.7: Consul 集群:

三台主机, IP 地址分别是 172.31.7.207,172.31.7.208,172.31.7.209

<https://www.consul.io/docs/agent/options.html>

3.7.1: 集群节点启动 consul 服务:

node1:

```
# unzip consul_1.7.3_linux_amd64.zip
# mv consul /usr/bin/
# mkdir /var/lib/consul
# cat /lib/systemd/system/consul.service
[Unit]
Description=Consul Server
Documentation=https://www.consul.io
After=network.target

[Service]
Restart=on-failure
WorkingDirectory=/var/lib/consul
ExecStart=/usr/bin/consul agent -server -bootstrap-expect 3 -data-dir /var/lib/consul -node
172.31.7.221 -bind=172.31.7.221 -client=0.0.0.0 -datacenter bj-sjhl -ui

[Install]
```



```
WantedBy=multi-user.target
```

```
# systemctl daemon-reload && systemctl start consul && systemctl enable consul
```

Node2:

```
# unzip consul_1.7.3_linux_amd64.zip
```

```
# mv consul /usr/bin/
```

```
# mkdir /var/lib/consul
```

```
# cat /lib/systemd/system/consul.service
```

```
[Unit]
```

```
Description=Consul Server
```

```
Documentation=https://www.consul.io
```

```
After=network.target
```

```
[Service]
```

```
Restart=on-failure
```

```
WorkingDirectory=/var/lib/consul
```

```
ExecStart=/usr/bin/consul agent -server -bootstrap-expect 3 -data-dir /var/lib/consul -node  
172.31.7.222 -bind=172.31.7.222 -client=0.0.0.0 -datacenter bj-sjhl -ui
```

```
[Install]
```

```
WantedBy=multi-user.target
```

```
# systemctl daemon-reload && systemctl start consul && systemctl enable consul
```

Node3:

```
# unzip consul_1.7.3_linux_amd64.zip
```

```
# mv consul /usr/bin/
```

```
# mkdir /var/lib/consul
```

```
# cat /lib/systemd/system/consul.service
```

```
[Unit]
```

```
Description=Consul Server
```

```
Documentation=https://www.consul.io
```

```
After=network.target
```

```
[Service]
```

```
Restart=on-failure
```

```
WorkingDirectory=/var/lib/consul
```

```
ExecStart=/usr/bin/consul agent -server -bootstrap-expect 1 -data-dir /var/lib/consul -node  
172.31.7.223 -bind=172.31.7.223 -client=0.0.0.0 -datacenter bj-sjhl -ui
```

```
[Install]
```

```
WantedBy=multi-user.target
```

```
# systemctl daemon-reload && systemctl start consul && systemctl enable consul
```

3.6.2: 将 node 加入 consul 集群:

将三台 node 中的任意两台 node 加入到另外一台 node。

比如在 node2 和 node3 加入到 node1, 加入集群后会选举出集群 leader, 否则目前集群无法使用且报错 agent: Coordinate update error: error="No cluster leader".

```
root@etcd2:~# consul join 172.31.7.221
Successfully joined cluster by contacting 1 nodes.
```

```
root@etcd3:~ # consul join 172.31.7.221
Successfully joined cluster by contacting 1 nodes.
```

3.6.3: 验证 consul 集群:

3.6.3.1: 验证集群节点状态:

Node	Address	Status	Type	Build	Protocol	DC	Segment
172.31.7.221	172.31.7.221:8301	alive	server	1.7.3	2	bj-sjhl	<all>
172.31.7.222	172.31.7.222:8301	alive	server	1.7.3	2	bj-sjhl	<all>
172.31.7.223	172.31.7.223:8301	alive	server	1.7.3	2	bj-sjhl	<all>

3.6.3.2: 验证集群 leader 与 follower 状态:

```
# consul operator raft list-peers
```

Node	ID	Address	State	Voter	RaftProtocol
172.31.7.223	976ba947-ea4c-74c1-75db-3734374c1c2a	172.31.7.223:8300	follower	true	3
172.31.7.222	93a0bbf2-f79d-85d6-f6fd-6d9808bb8961	172.31.7.222:8300	leader	true	3
172.31.7.221	0d9ffbb8-5dde-2a50-4bd0-acd167c2b098	172.31.7.221:8300	follower	true	3

3.6.3.3: 验证 consul web 界面:

← → ↻ 不安全 | 172.31.7.221:8500/ui/bj-sjhl/nodes

bj-sjhl Services Nodes Key/Value ACL Intentions

Nodes 3 total

All (Any Status) ☒ Critical Checks ☐ Warning Checks ☒ Passing Checks

Healthy Nodes

172.31.7.221
172.31.7.221

172.31.7.222
172.31.7.222

172.31.7.223
172.31.7.223

3.8: overlay 网络验证:

通过 network namespace 验证 docker overlay 网络。

3.8.1: 验证 network namespace:

```
root@docker-node1:~# brctl show
bridge name      bridge id                STP enabled  interfaces
docker0          8000.02421c5b93f1        no
docker_gwbridge  8000.02422fa88df3        no          veth53e9b61
                                   veth60bf94d
```

```
root@docker-node1:~# ll /var/run/docker/netns/
total 0
drwxr-xr-x 2 root root 100 May 11 19:16 ./
drwx----- 8 root root 180 May 11 18:49 ../
-r--r--r-- 1 root root   0 May 11 19:16 13e44a33bda9
-r--r--r-- 1 root root   0 May 11 18:49 1980b49a37ab
-r--r--r-- 1 root root   0 May 11 18:49 1-f55a3fc49c
```

3.8.2: 将 docker ns 创建软连接:

```
root@docker-node1:~# ln -sv /var/run/docker/netns/ /var/run/netns
'/var/run/netns' -> '/var/run/docker/netns/'
```

```
root@docker-node1:~# ip net ls #验证命令空间
13e44a33bda9 (id: 2)
1980b49a37ab (id: 1)
1-f55a3fc49c (id: 0)
```

3.8.3: 验证网络设备的 namespace:

```
root@docker-node1:~# ip netns exec 1-f55a3fc49c ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    inet 10.200.15.254 netmask 255.255.240.0 broadcast 10.200.15.255
    ether 52:2e:a2:99:b0:88 txqueuelen 0 (Ethernet)
    RX packets 7 bytes 196 (196.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    ether ca:14:ad:97:92:58 txqueuelen 0 (Ethernet)
    RX packets 19 bytes 1300 (1.3 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 13 bytes 1132 (1.1 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
```

```

        ether 66:22:a5:dd:61:42  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vxlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        ether 52:2e:a2:99:b0:88  txqueuelen 0  (Ethernet)
        RX packets 11  bytes 894 (894.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11  bytes 810 (810.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@docker-node1:~# ip netns  exec 1-f55a3fc49c ip addr
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
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 52:2e:a2:99:b0:88 brd ff:ff:ff:ff:ff:ff
    inet 10.200.15.254/20 brd 10.200.15.255 scope global br0
        valid_lft forever preferred_lft forever
6: vxlan0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state
UNKNOWN group default
    link/ether 52:2e:a2:99:b0:88 brd ff:ff:ff:ff:ff:ff link-netnsid 0
8: veth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP
group default
    link/ether ca:14:ad:97:92:58 brd ff:ff:ff:ff:ff:ff link-netnsid 1
12: veth1@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state
UP group default
    link/ether 66:22:a5:dd:61:42 brd ff:ff:ff:ff:ff:ff link-netnsid 2

```

3.8.4: 验证容器的 network namespace:

```

root@docker-node1:~# ip netns  exec 1980b49a37ab ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
    inet 10.200.0.1  netmask 255.255.240.0  broadcast 10.200.15.255
    ether 02:42:0a:c8:00:01  txqueuelen 0  (Ethernet)
    RX packets 13  bytes 1132 (1.1 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 19  bytes 1300 (1.3 KB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
    inet 172.18.0.2  netmask 255.255.0.0  broadcast 172.18.255.255
    ether 02:42:ac:12:00:02  txqueuelen 0  (Ethernet)
    RX packets 36  bytes 2712 (2.7 KB)
    RX errors 0  dropped 0  overruns 0  frame 0
    TX packets 0  bytes 0 (0.0 B)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

```

```

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 4 bytes 436 (436.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 4 bytes 436 (436.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@docker-node1:~# ip netns exec 13e44a33bda9 ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    inet 10.200.0.3 netmask 255.255.240.0 broadcast 10.200.15.255
    ether 02:42:0a:c8:00:03 txqueuelen 0 (Ethernet)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.18.0.3 netmask 255.255.0.0 broadcast 172.18.255.255
    ether 02:42:ac:12:00:03 txqueuelen 0 (Ethernet)
    RX packets 22 bytes 1636 (1.6 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

3.8.5: 验证容器 arp 表:

```

/# arp -a #最近和当前容器通信过的其他容器的 MAC 地址
01df34757dba.jiege_overlay_docker_net (10.200.0.3) at 02:42:0a:c8:00:03 [ether] on eth0
31f0d3b4a1c7.jiege_overlay_docker_net (10.200.0.2) at 02:42:0a:c8:00:02 [ether] on eth0

/# arp -a
? (172.18.0.1) at 02:42:2f:a8:8d:f3 [ether] on eth1 #网关的 MAC 地址
42f391ad9e72.jiege_overlay_docker_net (10.200.0.1) at 02:42:0a:c8:00:01 [ether] on eth0

```

3.8.6: 验证 iptables

POSTROUTING 是源地址转换
PREROUTING 是目的地址转换


```

root@docker-node1:~# iptables -vnl
Chain INPUT (policy ACCEPT 34542 packets, 6019K bytes)
pkts bytes target prot opt in out source destination

Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts bytes target prot opt in out source destination
28 2626 DOCKER-USER all -- * * 0.0.0.0/0 0.0.0.0/0
28 2626 DOCKER-ISOLATION-STAGE-1 all -- * * 0.0.0.0/0 0.0.0.0/0 0.0.0.0/0
0 0 ACCEPT all -- * docker0 0.0.0.0/0 0.0.0.0/0 ctstate RELATED,ESTABLISHED
0 0 DOCKER all -- * docker0 0.0.0.0/0 0.0.0.0/0
0 0 ACCEPT all -- docker0 !docker0 0.0.0.0/0 0.0.0.0/0
0 0 ACCEPT all -- docker0 docker0 0.0.0.0/0 0.0.0.0/0
14 1610 ACCEPT all -- * docker_gwbridge 0.0.0.0/0 0.0.0.0/0 ctstate RELATED,ESTABLISHED
0 0 DOCKER all -- * docker_gwbridge 0.0.0.0/0 0.0.0.0/0
14 1016 ACCEPT all -- docker_gwbridge !docker_gwbridge 0.0.0.0/0 0.0.0.0/0
0 0 DROP all -- docker_gwbridge docker_gwbridge 0.0.0.0/0 0.0.0.0/0

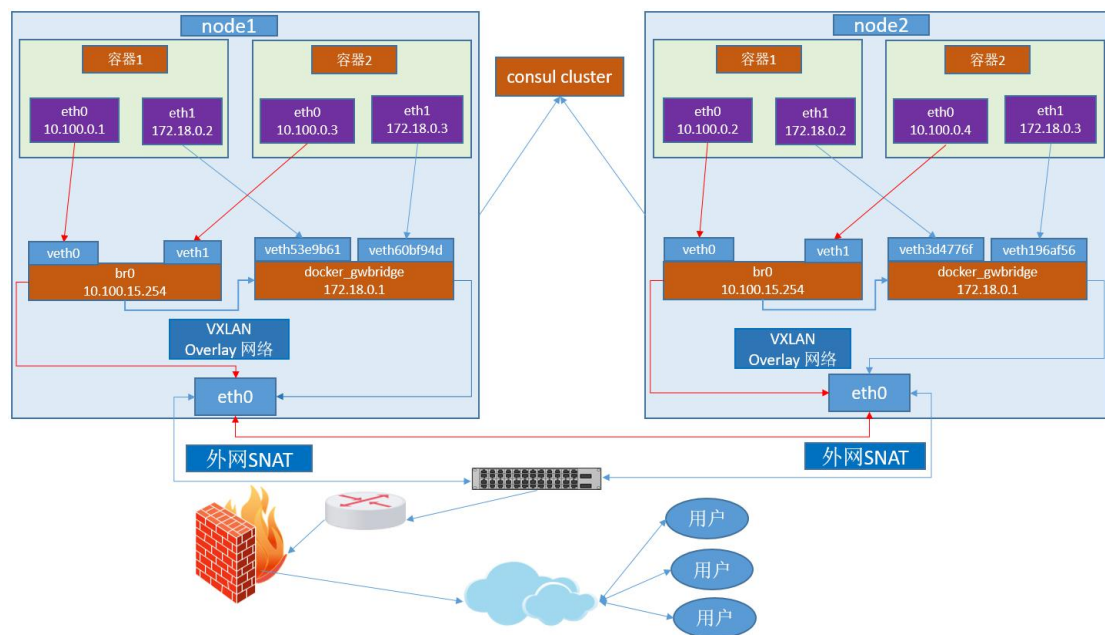
Chain OUTPUT (policy ACCEPT 33069 packets, 5123K bytes)
pkts bytes target prot opt in out source destination

Chain DOCKER (2 references)
pkts bytes target prot opt in out source destination

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
pkts bytes target prot opt in out source destination
0 0 DOCKER-ISOLATION-STAGE-2 all -- docker0 !docker0 0.0.0.0/0 0.0.0.0/0
14 1016 DOCKER-ISOLATION-STAGE-2 all -- docker_gwbridge !docker_gwbridge 0.0.0.0/0 0.0.0.0/0
28 2626 RETURN all -- * * 0.0.0.0/0 0.0.0.0/0

Chain DOCKER-ISOLATION-STAGE-2 (2 references)
pkts bytes target prot opt in out source destination
0 0 DROP all -- * docker0 0.0.0.0/0 0.0.0.0/0
0 0 DROP all -- * docker_gwbridge 0.0.0.0/0 0.0.0.0/0
14 1016 RETURN all -- * * 0.0.0.0/0 0.0.0.0/0

```



3.9: 创建多个网络:

3.9.1: 创建另外一个 overlay 网络:

```

root@docker-node1:~# docker network create -d overlay --subnet 10.10.0.0/20
--gateway 10.10.15.254 jiege_overlay_docker_net1
fe270945b9fe059e65ddd7daa5c68b256e13a666234e3b9626fa2183cb5bb1ab

```

```
root@docker-node1:~# ip net ls
13e44a33bda9 (id: 2)
1980b49a37ab (id: 1)
1- f55a3fc49c (id: 0)
```

3.9.2: 验证网络:

```
root@docker-node1:~# ip netns exec 1-f55a3fc49c ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    inet 10.200.15.254 netmask 255.255.240.0 broadcast 10.200.15.255
    ether 52:2e:a2:99:b0:88 txqueuelen 0 (Ethernet)
    RX packets 8 bytes 224 (224.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    ether ca:14:ad:97:92:58 txqueuelen 0 (Ethernet)
    RX packets 24 bytes 1678 (1.6 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 18 bytes 1510 (1.5 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

veth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    ether 66:22:a5:dd:61:42 txqueuelen 0 (Ethernet)
    RX packets 5 bytes 378 (378.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 5 bytes 378 (378.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

vxlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1450
    ether 52:2e:a2:99:b0:88 txqueuelen 0 (Ethernet)
    RX packets 11 bytes 894 (894.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 11 bytes 810 (810.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

3.9.3: 创建容器并测试网络:

```
root@docker-node1:~# docker run -it --net=jiege_overlay_docker_net1 busybox sh
/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:0A:00:01
```

```

    inet addr:10.10.0.1  Bcast:10.10.15.255  Mask:255.255.240.0
    UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
    RX packets:0 errors:0 dropped:0 overruns:0 frame:0
    TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:12:00:04
    inet addr:172.18.0.4  Bcast:172.18.255.255  Mask:255.255.0.0
    UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
    RX packets:5 errors:0 dropped:0 overruns:0 frame:0
    TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:0
    RX bytes:426 (426.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
    inet addr:127.0.0.1  Mask:255.0.0.0
    UP LOOPBACK RUNNING  MTU:65536  Metric:1
    RX packets:0 errors:0 dropped:0 overruns:0 frame:0
    TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
    collisions:0 txqueuelen:1000
    RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # ping www.baidu.com
PING www.baidu.com (61.135.169.125): 56 data bytes
64 bytes from 61.135.169.125: seq=0 ttl=127 time=3.874 ms
64 bytes from 61.135.169.125: seq=1 ttl=127 time=4.290 ms

```

四：weave：

<https://www.weave.works/>

4.1：weave 简介：

Weave 是 Github 上一个比较热门的 Docker 容器网络方案，具有非常好的易用性且功能强大。Weave 的框架它包含了两大主要组件：

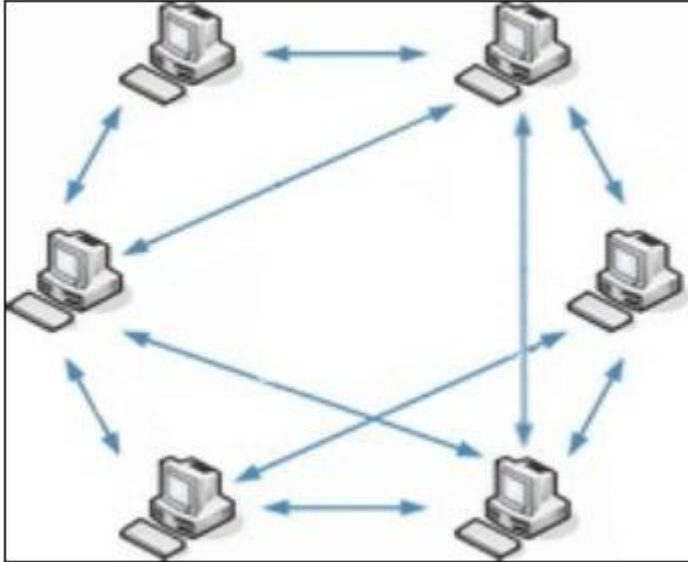
- 1) Weave: 用户态的 shell 脚本，用于安装 Weave，将 container 连接到 Weave 虚拟网络。并为它们分配 IP。
- 2) Weaver: 运行于 container 内，每个 Weave 网络内的主机都要运行，是一个 Go 语言实现的虚拟网络路由器。不同主机之间的网络通信依赖于 Weaver 路由。

Weave 通过创建虚拟网络使 Docker 容器能够跨主机通信并能够自动相互发现。通过 weave 网络，由多个容器构成的基于微服务架构的应用可以运行在任何地方：主机，多主机，云上或者数据中心。应用程序使用网络就好像容器是插在同一个网络交换机上一样，不需要配置端口映射，连接等。在 weave 网络中，使用应用容器提供的服务可以暴露给外部，而不用管它们运行在何处。类似地，现存的内部系统也可以接受来自于应用容器的请求，而不管容器运行于何处。

一个 Weave 网络由一系列的对等网络端点(Peer-to-Peer)构成,每个节点都会运行一个 weave 路由器, Weave 路由器之间建立起 TCP 连接,通过这个连接进行心跳握手和拓扑信息交换,这些连接可以通过配置进行加密。

peers 之间还会建立 UDP 连接,也可以进行加密,这些 UDP 连接用于网络包的封装,这些连接是双工的而且可以穿越防火墙。

Weave 网络在主机上创建一个网桥,每个容器通过 veth pair 连接到网桥上,容器由用户或者 weave 网络的 IPADM 分配 IP 地址。



4.2: weave 优点:

<https://www.weave.works/docs/net/latest/overview/>

Weave 的优点:

配置简单: 相对 flannel 和 calico 更简单

不需要额外存储服务: flannel 和 calico 依赖 etcd 或 zookeeper 或者 consul

支持服务发现: 支持通过容器名称访问容器(服务发现), 还可以对运行在不同主机的多个同名的容器提供负载均衡的功能。

Weave 网络性能很强: 自动选择两个节点最快的路径, 不需要交换机进行路由选择和转发, 因此可以实现接近于物理网络的性能。

支持组播: 支持部分服务的组播功能

支持编排服务: 可以与 kubernetes 等编排工具完美结合使用。

4.3: weave 的网络工作方式:

从下图的网络模型图中可以看出, 对每一个 weave 网络中的容器, weave 都会创建一个网桥, 并且在网桥和每个容器之间创建一个 veth pair, 一端作为容器网卡加入到容器的网络命名空间中, 并为容器网卡配置 ip 和相应的掩码, 一端连接在网桥上, 最终通过宿主机上 weave router 将流量转发到对端主机上。

其基本过程如下:

1) 容器流量通过 veth pair 到达宿主机上 weave router 网桥上。

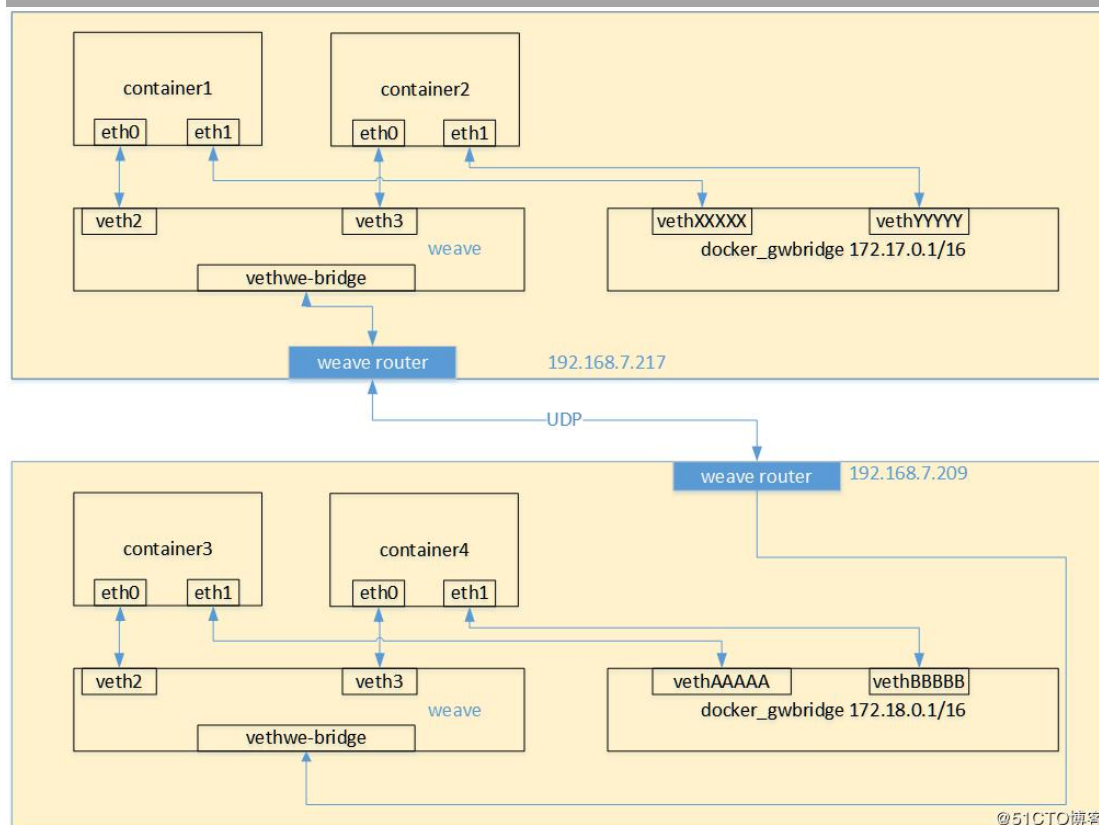
2) weave router 在混杂模式下使用 pcap 在网桥上截获网络数据包, 并排除由内核直接通过网桥转发的数据流量, 例如本子网内部、本地容器之间的数据以及宿主机和本地容

器之间的流量。捕获的包通过 UDP 转发到所其他主机的 weave router 端。

3) 在接收端, weave router 通过 pcap 将包注入到网桥上的接口, 通过网桥上的 veth pair, 将流量分发到容器的网卡上。

weave 默认基于 UDP 承载容器之间的数据包, 并且可以完全自定义整个集群的网络拓扑, 但从性能和使用角度来看, 还是有比较大的缺陷的:

- 1) weave 自定义容器数据包的封包解包方式, 不够通用, 传输效率比较低, 性能上的损失也比较大。
- 2) 集群配置比较负载, 需要通过 weave 命令行来手工构建网络拓扑, 在大规模集群的情况下, 加重了管理员的负担。



<https://blog.51cto.com/dengaosky/2069478>

<https://www.cnblogs.com/xiangsikai/p/9900250.html>

4.4: 部署 weave 网络:

<https://www.weave.works/docs/net/latest/install/installing-weave/>

<https://www.weave.works/docs/net/latest/install/systemd/> #systemd 启动

环境要求:

内核: 3.8 或以上, Docker: 1.10 或以上

TCP 6783, UDP 6783/6784, 分别是 weave 的管理端口和数据端口

各主机配置不同的主机名

Node1: 172.31.6.101

```
#wget -O /usr/local/bin/weave
https://raw.githubusercontent.com/zettio/weave/master/weave

#chmod a+x /usr/local/bin/weave
#weave launch 172.31.6.102 172.31.6.103 #本机和其他 node 创建 wave 网络(同一个 vxlan,
默认会创建一个 10.32.0.0/12 的子网 )
# docker run -it --rm --net=weave busybox
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2  Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:942 (942.0 B)  TX bytes:0 (0.0 B)

ethwe0     Link encap:Ethernet  HWaddr 0E:32:A9:DE:42:FA
          inet addr:10.40.0.0  Bcast:10.47.255.255  Mask:255.240.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1012 (1012.0 B)  TX bytes:42 (42.0 B)

lo         Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # route -n
Kernel IP routing table
Destination    Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0        172.18.0.1     0.0.0.0         UG    0      0      0 eth0
10.32.0.0      0.0.0.0        255.240.0.0     U     0      0      0 ethwe0
172.18.0.0     0.0.0.0        255.255.0.0     U     0      0      0 eth0
224.0.0.0      0.0.0.0        240.0.0.0       U     0      0      0 ethwe0
```

node2: 172.31.6.102

```
#wget -O /usr/local/bin/weave
https://raw.githubusercontent.com/zettio/weave/master/weave

#chmod a+x /usr/local/bin/weave
#weave launch 172.31.6.101 172.31.6.103
# docker run -it --rm --net=weave busybox
/ # ifconfig
```

```
eth0      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2  Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:946 (946.0 B)  TX bytes:0 (0.0 B)

ethwe0    Link encap:Ethernet  HWaddr CE:D1:94:BB:A2:01
          inet addr:10.36.0.0  Bcast:10.47.255.255  Mask:255.240.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516 (516.0 B)  TX bytes:42 (42.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ # route -n
Kernel IP routing table
Destination        Gateway            Genmask           Flags Metric Ref    Use Iface
0.0.0.0            172.18.0.1        0.0.0.0           UG        0      0        0 eth0
10.32.0.0          0.0.0.0           255.240.0.0       U         0      0        0 ethwe0
172.18.0.0         0.0.0.0           255.255.0.0       U         0      0        0 eth0
224.0.0.0          0.0.0.0           240.0.0.0         U         0      0        0 ethwe0

验证容器之间的夸主机通信:
```

```
Node3: 172.31.6.103

#wget -O /usr/local/bin/weave
https://raw.githubusercontent.com/zettio/weave/master/weave

#chmod a+x /usr/local/bin/weave
# weave launch 172.31.6.101 172.31.6.102
# docker run -it --rm --net=weave busybox

/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:12:00:02
          inet addr:172.18.0.2  Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
```

```

collisions:0 txqueuelen:0
RX bytes:856 (856.0 B) TX bytes:0 (0.0 B)

ethwe0    Link encap:Ethernet  HWaddr 7A:FD:AE:51:D3:0A
          inet addr:10.32.0.1 Bcast:10.47.255.255 Mask:255.240.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1376 Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516 (516.0 B) TX bytes:42 (42.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.0
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

4.5: 创建容器并验证网络通信:

在 node1 上的容器作为测试发起者，分别测试同主机内的容器通信、容器与其他多个宿主主机上的容器通信、容器到其他宿主机网络通信:

4.5.1: 验证容器通信:

同主机的容器间通信:

```

# docker run -it --rm --net=weave busybox
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:12:00:03
          inet addr:172.18.0.3 Bcast:172.18.255.255 Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516 (516.0 B) TX bytes:0 (0.0 B)

ethwe0    Link encap:Ethernet  HWaddr 82:E9:08:C0:12:7A
          inet addr:10.40.0.1 Bcast:10.47.255.255 Mask:255.240.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1376 Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516 (516.0 B) TX bytes:42 (42.0 B)

/ # traceroute 10.40.0.0
traceroute to 10.40.0.0 (10.40.0.0), 30 hops max, 46 byte packets
 1  fe227ea7ccc8.weave (10.40.0.0)  0.003 ms  0.002 ms  0.001 ms

```

容器跨主机通信:

```

/ # traceroute 10.36.0.0
traceroute to 10.36.0.0 (10.36.0.0), 30 hops max, 46 byte packets

```

```

1  10.36.0.0 (10.36.0.0)  0.661 ms  0.453 ms  0.760 ms
/#
/# traceroute 10.32.0.1
traceroute to 10.32.0.1 (10.32.0.1), 30 hops max, 46 byte packets
1  10.32.0.1 (10.32.0.1)  2.712 ms  0.410 ms  0.981 ms
/#

```

容器到其他主机通信:

```

/# traceroute 172.31.6.102
traceroute to 172.31.6.102 (172.31.6.102), 30 hops max, 46 byte packets
1  172.18.0.1 (172.18.0.1)  0.003 ms  0.002 ms  0.001 ms
2  172.31.6.102 (172.31.6.102)  0.387 ms  0.874 ms  2.145 ms
/#
/# traceroute 172.31.6.103
traceroute to 172.31.6.103 (172.31.6.103), 30 hops max, 46 byte packets
1  172.18.0.1 (172.18.0.1)  0.003 ms  0.002 ms  0.002 ms
2  172.31.6.103 (172.31.6.103)  1.904 ms  0.417 ms  0.668 ms

```

4.5.2: 基于 weave sock 创建容器:

```

# export DOCKER_HOST=unix:///var/run/weave/weave.sock
# export DOCKER_HOST=unix:///var/run/weave/weave.sock
# docker run -it --rm busybox
/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:696 (696.0 B)  TX bytes:0 (0.0 B)

ethwe     Link encap:Ethernet  HWaddr 22:E2:64:82:B6:E1
          inet addr:10.40.0.2  Bcast:10.47.255.255  Mask:255.240.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:308 (308.0 B)  TX bytes:42 (42.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

```

4.5.3: 验证命令:

```
# weave version
weave script 2.6.2
weave 2.6.2

# weave status connections
-> 172.31.6.101:6783      established fastdp 1a:fb:8d:d5:91:b2(docker-node1.magedu.local)
mtu=1376
-> 172.31.6.103:6783      established fastdp 06:c6:8c:a4:65:35(docker-node3.magedu.local)
mtu=1376

# weave ps
weave:expose 1a:fb:8d:d5:91:b2
243689170f63 22:e2:64:82:b6:e1 10.40.0.2/12
e502a2562572 82:e9:08:c0:12:7a 10.40.0.1/12
fe227ea7ccc8 0e:32:a9:de:42:fa 10.40.0.0/12

# weave status

    Version: 2.6.2 (up to date; next check at 2020/05/18 10:08:50)

    Service: router
    Protocol: weave 1..2
        Name: 1a:fb:8d:d5:91:b2(docker-node1.magedu.local)
    Encryption: disabled
    PeerDiscovery: enabled
        Targets: 2
    Connections: 2 (2 established)
        Peers: 3 (with 6 established connections)
    TrustedSubnets: none

    Service: ipam
        Status: ready
        Range: 10.32.0.0/12
    DefaultSubnet: 10.32.0.0/12

    Service: dns
        Domain: weave.local.
    Upstream: 127.0.0.53
        TTL: 1
    Entries: 0

    Service: proxy
    Address: unix:///var/run/weave/weave.sock

    Service: plugin (legacy)
    DriverName: weave、

# weave status peers
```

```
1a:fb:8d:d5:91:b2(docker-node1.magedu.local)
  <- 172.31.6.103:57735      06:c6:8c:a4:65:35(docker-node3.magedu.local) established
  <- 172.31.6.102:34469      ce:99:a4:6c:ac:6b(docker-node2.magedu.local) established
06:c6:8c:a4:65:35(docker-node3.magedu.local)
  -> 172.31.6.101:6783      1a:fb:8d:d5:91:b2(docker-node1.magedu.local) established
  <- 172.31.6.102:37511      ce:99:a4:6c:ac:6b(docker-node2.magedu.local) established
ce:99:a4:6c:ac:6b(docker-node2.magedu.local)
  -> 172.31.6.101:6783      1a:fb:8d:d5:91:b2(docker-node1.magedu.local) established
  -> 172.31.6.103:6783      06:c6:8c:a4:65:35(docker-node3.magedu.local) established
```

五：Open vSwitch：

Open vSwitch 即开放的虚拟交换机，简称 OVS，基于 C 语言编写，支持 Apache 2.0 协议许可，官方网站 <http://www.openvswitch.org/>，常用于在虚拟化、Docker 等环境实现网络虚拟化技术(内核版本 3.10 及以上)，是实现分布式虚拟网络的企业级开源软件，其支持计算机网络中使用的多种协议和标准，具体如下：

Open vSwitch 特性：

支持通过 NetFlow sFlow IPFIX, SPAN, RSPAN, 和 GRE-tunneled 镜像使虚拟机内部通讯可以被监控；

支持 LACP (IEEE 802.1AX-2008)（多端口绑定）协议；

支持标准的 802.1Q VLAN 模型以及 trunk 模式；

支持 BFD 和 802.1ag 链路状态监测；

支持 STP (IEEE 802.1D-1998)；

支持细粒度的 Qos；

支持 HFSC 系统级别的流量控制队列；

支持每虚拟机网卡的流量的流量控制策略；

支持基于源 MAC 负载均衡模式、主备模式、L4 哈希模式的多端口绑定；

支持 OpenFlow 协议（包括许多虚拟化的增强特性）；

支持 IPV6

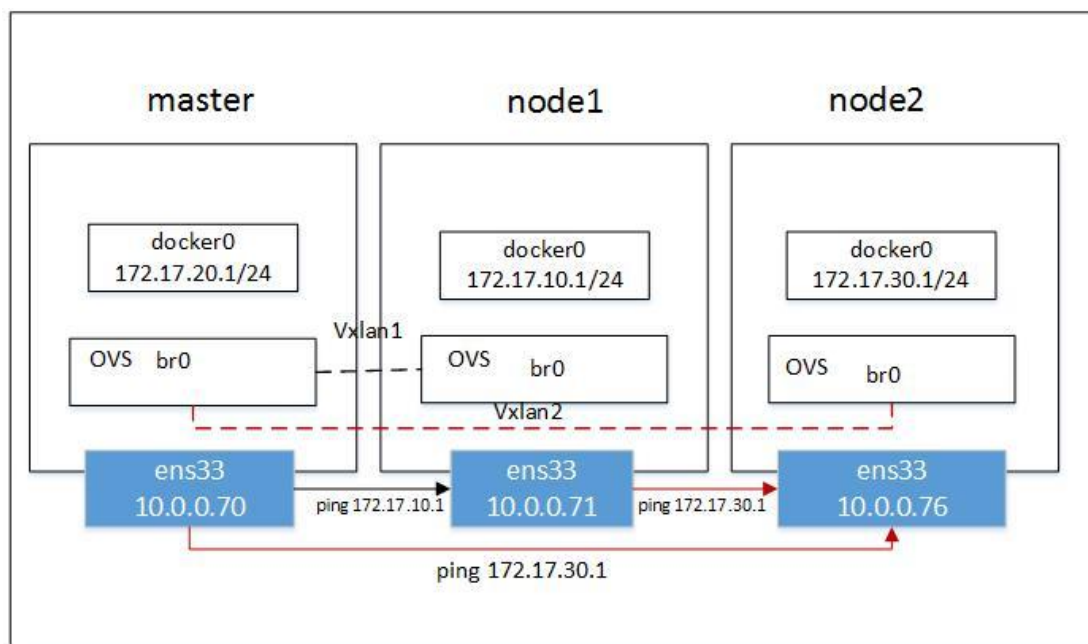
支持多种隧道协议（(GRE, VXLAN, IPsec, GRE and VXLAN over IPsec）

支持通过 C 或者 Python 接口远程配置；

支持内核态和用户态的转发引擎设置；

支持多列表转发的发送缓存引擎；

16 支持转发层抽象以容易的定向到新的软件或者硬件平台；



5.1: Open vSwitch 组件:

ovs-vswitchd: 实现虚拟交换机的守护程序

ovsdb-server: 轻型数据库服务器, ovs-vswitchd 查询数据以获取其配置

ovs-dbctl: 用于配置交换机内核的工具

ovs-vsctl: 用于查看和更新 ovs-vswitchd 的工具

ovs-appctl: 一个用于将命令发送到正在运行的 Open vSwitch 守护程序的实用程序

Open vSwitch 还提供了一些其他工具:

ovs-ofctl: 用于查看和控制 OpenFlow(管理协议、接口)交换机和控制器的实用程序

ovs-pki: 一个用于创建和管理 OpenFlow 交换机的公共密钥基础结构的实用程序

ovs-testcontroller: 一个简单的 OpenFlow 控制器, 可能对测试有用 (尽管不适用于生产)

ovs-tcpdump: 够解析 OpenFlow(管理协议的)消息。

编译安装要对内核进行部分功能替换

5.2: 安装:

编译安装对 linux kernel 版本的需求:

<http://docs.openvswitch.org/en/latest/faq/releases/>

5.2.1: 编译安装:

<http://docs.openvswitch.org/en/latest/intro/install/general/#installation-requirements>

下载或 clone 最新源码包:

```
git clone https://github.com/openvswitch/ovs.git
```


5.2.2: 软件包安装:

<http://docs.openvswitch.org/en/latest/intro/install/distributions/#debian>

```
# apt install openvswitch-switch bridge-utils

# lsmod | grep openvswitch
openvswitch          131072  0
nsh                  16384  1 openvswitch
nf_nat_ipv6          16384  1 openvswitch
nf_nat_ipv4          16384  1 openvswitch
nf_defrag_ipv6       20480  2 nf_conntrack_ipv6,openvswitch
nf_nat               32768  3 nf_nat_ipv6,nf_nat_ipv4,openvswitch
nf_conntrack         131072  6
nf_conntrack_ipv6,nf_conntrack_ipv4,nf_nat,nf_nat_ipv6,nf_nat_ipv4,openvswitch
libcrc32c            16384  4 nf_conntrack,nf_nat,openvswitch,raid456
```