文档名称：ZDNS-Zcloud-kubernetes CNI-Flannel 原理解读
文档分类：ZDNS-Zcloud-kubernetes 文档
服务等级：
文档编号：
作　　者: 余春云
联系电话：18811483030
电子邮件：yuchunyun@zdns.cn

Kubernetes CNI-Flannel 原理解读

文档变更历史记录：

变更日期	变更人	变更内容摘要	组长确认
2019-03-19	余春云	初次建立文档	

目录

源码路径： https://github.com/containernetworking/plugins
flannel cni 路径： plugins/plugins/meta/flannel/flannel.go

CNI 基本思想

Container Runtime 在创建容器时，先创建好 network namespace，然后调用 CNI 插件为这个 ns 配置网络，其后再启动容器内的进程。

使用 CNI 后，容器的 IP 分配就变成了如下步骤：

- kubelet 先创建 pause 容器生成 network namespace
- 调用网络 CNI driver
- CNI driver 根据配置调用具体的 cni 插件
- cni 插件给 pause 容器配置网络
- pod 中其他的容器都使用 pause 容器的网络

CNI 规范

CNI(Container Network Interface) 是 google 和 CoreOS 主导制定的容器网络标准，是一个协议。这个协议连接了两个组件：容器管理系统和网络插件。它们之间通过 JSON 格式的文件进行通信，实现容器的网络功能。

CNI 的目的在于定义一个标准的接口规范，使得 kubernetes 在增删 POD 的时候，能够按照规范向 CNI 实例提供标准的输入并获取标准的输出，再将输出作为 kubernetes 管理这个 POD 的网络的参考。

在满足这个输入输出以及调用标准的 CNI 规范下，kubernetes 委托 CNI 实例去管理 POD 的网络资源并为 POD 建立互通能力。具体包括：创建容器网络空间（network namespace）、把网络接口（interface）放到对应的网络空间、给网络接口分配 IP 等等。

CNI 插件必须实现一个可执行文件，这个文件可以被容器管理系统（例如 rkt 或 Kubernetes）调用。

CNI 插件必须实现容器网络添加，容器网络删除，IP 分配等功能

CNI 设计考量

- 容器运行时必须在调用任何插件之前为容器创建一个新的网络命名空间。
- 运行时必须确定这个容器应属于哪个网络，并为每个网络确定哪些插件必须被执行。
- 网络配置采用 JSON 格式，可以很容易地存储在文件中。网络配置包括必填字段，如 name 和 type 以及插件（类型）。网络配置允许字段在调用之间改变值。为此，有一个可选的字段 args，必须包含不同的信息。
- 容器运行时必须按顺序为每个网络执行相应的插件，将容器添加到每个网络中。
- 在完成容器生命周期后，运行时必须以相反的顺序执行插件（相对于执行添加容器的顺序）以将容器与网络断开连接。

- 容器运行时不能为同一容器调用并行操作，但可以为不同的容器调用并行操作。
- 容器运行时必须为容器订阅 ADD 和 DEL 操作，这样 ADD 后面总是跟着相应的 DEL。 DEL 可能跟着额外的 DEL，但是，插件应该允许处理多个 DEL（即插件 DEL 应该是幂等的）。
- 容器必须由 ContainerID 唯一标识。存储状态的插件应该使用（网络名称，容器 ID）的主键来完成。
- 运行时不能调用同一个网络名称或容器 ID 执行两次 ADD（没有相应的 DEL）。换句话说，给定的容器 ID 必须只能添加到特定的网络一次。

CNI 插件类型

基础可执行文件，按照功能分三类：

Main 插件

用来创建具体网络设备接口的二进制文件。

例如：birdge(网桥设备)、ipvlan、lookback(lo 设备)、ptp(Veth Pair 设备)、macvlan、vlan

IPAM 插件

负责分配 IP 地址的二进制文件。

例如：dhcp,这个文件会向 dhcp 服务器发起请求；host-local,则会使用预先配置的 IP 地址来进行分配。

Meta 插件

拥有部分功能的插件，必须依赖其他的 main plugin 链式使用。

例如 flannel,就是专门为 Flannel 项目提供的 CNI 插件，根据 flannel 的配置文件创建接口。

CNI&源码简易解读及调用逻辑步骤

源码：kubelet 调用 CNI 接口

// kubernetes/pkg/kubelet/dockershim/network/cni/cni.go
调用 cni 架构接口 AddNetworkList
调用 cni 架构接口 DelNetworkList

源码：CNI 接口

// github.com/containernetworking/cni/libcni/api.go
CNI 接口只有四个方法，添加网络、删除网络、添加网络列表、删除网络列表

```
type CNI interface {
    AddNetworkList(net *NetworkConfigList, rt *RuntimeConf) (types.Result, error)
    DelNetworkList(net *NetworkConfigList, rt *RuntimeConf) error
    AddNetwork(net *NetworkConfig, rt *RuntimeConf) (types.Result, error)
    DelNetwork(net *NetworkConfig, rt *RuntimeConf) error
}
```

AddNetworkList 方法：从上至下调用/etc/cni/net.d/10-flannel.conflist 中 plugins 中 type 字段的命令
AddNetworkList 调用 /opt/cni/bin/flannel 命令
AddNetworkList 调用 /opt/cni/bin/portmap 命令
DelNetworkList 方法：逆向调用

插件参数传递

环境变量传递

调用插件的时候，这些参数会通过环境变量进行传递：
CNI_COMMAND：要执行的操作，可以是 ADD（把容器加入到某个网络）、DEL（把容器从某个网络中删除）
CNI_CONTAINERID：容器的 ID，比如 ipam 会把容器 ID 和分配的 IP 地址保存下来。可选的参数，但是推荐传递过去。需要保证在管理平台上是唯一的，如果容器被删除后可以循环使用
CNI_NETNS：容器的 network namespace 文件，访问这个文件可以在容器的网络 namespace 中操作
CNI_IFNAME：要配置的 interface 名字，比如 eth0
CNI_ARGS：额外的参数，是由分号;分割的键值对，比如 “FOO=BAR;hello=world”
CNI_PATH：CNI 二进制查找的路径列表，多个路径用分隔符 : 分隔

标准输入传递

网络信息主要通过标准输入，作为 JSON 字符串传递给插件，必须的参数包括：
cniVersion：CNI 标准的版本号。因为 CNI 在演化过程中，不同的版本有不同的要求
name：网络的名字，在集群中应该保持唯一
type：网络插件的类型，也就是 CNI 可执行文件的名称
args：额外的信息，类型为字典
ipMasq：是否在主机上为该网络配置 IP masquerade

ipam：IP 分配相关的信息，类型为字典
dns：DNS 相关的信息，类型为字典
插件接到这些数据，从输入和环境变量解析到需要的信息，根据这些信息执行程序逻辑

输出

把结果返回给调用者，返回的结果中一般包括这些参数：
IPs assigned to the interface：网络接口被分配的 ip，可以是 IPv4、IPv6 或者都有
DNS 信息：包含 nameservers、domain、search domains 和其他选项的字典

/opt/cni/bin/flannel 命令

源码：flannel 4 个主要方法

```
//    github.com/containernetworking/plugins/blob/master/plugins/meta/flannel/flannel.go
func main() {
	// TODO: implement plugin version
	skel.PluginMain(cmdAdd, cmdGet, cmdDel, version.All, "TODO")
}
```

cmdAdd 函数

一：构造结构体（生成配置文件）

结构体说明

name (string, required): the name of the network.
type (string, required): "bridge".
bridge (string, optional): name of the bridge to use/create. Defaults to "cni0".
isGateway (boolean, optional): assign an IP address to the bridge. Defaults to false.
isDefaultGateway (boolean, optional): Sets isGateway to true and makes the assigned IP the default route. Defaults to false.
forceAddress (boolean, optional): Indicates if a new IP address should be set if the previous value has been changed. Defaults to false.
ipMasq (boolean, optional): set up IP Masquerade on the host for traffic originating from this network and destined outside of it. Defaults to false.
mtu (integer, optional): explicitly set MTU to the specified value. Defaults to the value chosen by the kernel.
hairpinMode (boolean, optional): set hairpin mode for interfaces on the bridge. Defaults to false.
ipam (dictionary, required): IPAM configuration to be used for this network.
promiscMode (boolean, optional): set promiscuous mode on the bridge. Defaults to false.

主要函数

loadNetConf：主要是得到 NetConf 结构体配置信息。通过加载/etc/cni/net.d/10-flannel.conflist 和/run/flannel/subnet.env，验证参数合法性以及对于未设置的参数初始化，保存到/var/lib/cni/flannel/<容器 ID>文件中。

注意

/run/flannel/subnet.env 文件依赖于后面 flanneld 进程来生成。其中的 subnet 字段表明给该主机分配的可用子网。

如果 flanneld 指定--kube-subnet-mgr 参数，则由 kube-api 管理子网，实际上是读取 Node 的 podCIDR 属性。该属性又依赖 Kube-Controller-Manager 开启子网划分功能。

如果 kube-Controller-Manager 启动时开启了--allocate-node-cidrs 参数（默认 false），则启动 node-ipam-controller 控制器，它会从 kube-Controller-Manager 启动参数--cluster-cidr 所定义的集群 Pod 网络段中给每个 Node 分配和管理一个小的网络段，并编辑 Node 的 PodCIDR 属性。

配置文件示例

默认指定 delegate 为 bridge

默认网桥名称为 cni0

默认的 ipmi 为 host-local

```
{
    "cniVersion": "0.3.1",
    "forceAddress": true,
    "ipMasq": false,
    "ipam": {
        "routes": [
            {
                "dst": "10.42.0.0/16"
            }
```

```
            ],
            "subnet": "10.42.5.0/24",
            "type": "host-local"
        },
        "isDefaultGateway": true,
        "isGateway": true,
        "mtu": 1450,
        "name": "cbr0",
        "type": "bridge"
}
```

二：调用 /opt/cni/bin/bridge 命令

主要函数

setupBridge 函数

- setupBridge 里面调用 ensureBridge，前面设置了 N 多系统调用参数
- 通过 netlink.LinkAdd(br)创建网桥
  相当于 ip link add br-test type bridge
- 通过 netlink.LinkSetUp(br)启动网桥
  相当于 ip link set dev br-test up

setupVeth 函数

- 调用 netlink.LinkAdd(veth)创建 veth，这个是一个管道，Linux 的网卡对，在容器对应的 namespace 下创建好虚拟网络接口
  相当于 ip link add test-veth0 type veth peer name test-veth1
- 调用 netlink.LinkSetUp(contVeth)启动容器端网卡
  相当于 ip link set dev test-veth0 up
- 调用 netlink.LinkSetNsFd(hostVeth, int(hostNS.Fd()))将 host 端加入 namespace 中
  相当于 ip link set $link netns $ns
- 调用 netlink.LinkSetMaster(hostVeth, br)绑到 bridge
  相当于 ip link set dev test-veth0 master br-test

三：调用 /opt/cni/bin/host-local 命令

ipam.ExecAdd 函数

获取 IP 地址

calcGateways 函数

根据 IP 地址计算对应的路由和网关

ipam.ConfigureIface 函数

将 IP 地址设置到对应的虚拟网络接口上
相当于 ifconfig test-veth0 192.168.209.135/24 up

enableIPForward(gws.family)函数

开启 ip 转发，路径/proc/sys/net/ipv4/ip_forward 写入值 1

四：ip.SetupIPMasq 函数

调用 ip.SetupIPMasq 建立 iptables 规则。使用 iptables 增加容器私有网网段到外部网段的 masquerade 规则，这样容器内部访问外部网络时会进行 snat，在很多情况下配置了这条路由后容器内部才能访问外网。（这里代码中会做 exist 检查，防止生成重复的 iptables 规则）；

实现功能

ADD 命令

- 执行 ADD 命令时，brdige 组件创建一个指定名字的网桥，如果网桥已经存在，就使用已有的网桥；
- 创建 vethpair，将 node 端的 veth 设备连接到网桥上；
- 从 ipam 获取一个给容器使用的 ip 数据，并根据返回的数据计算出容器对应的网关；
- 进入容器网络名字空间，修改容器中网卡名和网卡 ip，以及配置路由，并进行 arp 广播（注>- 意我们只为 vethpair 的容器端配置 ip，node 端是没有 ip 的）；
- 如果 IsGW=true，将网桥配置为网关，具体方法是：将第三步计算得到的网关 IP 配置到网桥上同时根据需要将网桥上其他 ip 删除。最后开启网桥的 ip_forward 内核参数；
- 如果 IPMasq=true，使用 iptables 增加容器私有网网段到外部网段的 masquerade 规则，这样容器内部访问外部网络时会进行 snat，在很多情况下配置了这条路由后容器内部才能访问外网。（这里代码中会做 exist 检查，防止生成重复的

iptables 规则）；

- 配置结束，整理当前网桥的信息，并返回给调用者。

DEL 命令

- 根据命令执行的参数，确认要删除的容器 ip，调用 ipam 的 del 命令，将 IP 还回 IP pool；
- 进入容器的网络名字空间，根据容器 IP 将对应的网卡删除；
- 如果 IPMasq=true，在 node 上删除创建网络时配置的几条 iptables 规则。

/opt/cni/bin/portmap 命令

用于在 node 根据 pod 中 ports 配置 iptables 规则链，进行 SNAT、DNAT 和端口转发。

portmap 组件通常在 main 组件执行完毕后执行，因为它的执行参数仰赖之前的组件提供。

capabilities 中的 portMappings 参数设置 true 则会开启 ports 中信息增加 DNAT, SNAT

实现功能

添加容器网络操作

生成 CNI-HOSTPORT-DNAT 链路存在于 NAT 表中
根据 pod 对应 yaml 文件中的 ports 段内容 生成对应每一个 pod 的规则
生成 CNI-HOSTPORT-SNAT 链路存在于 NAT 表中
根据 pod 对应 yaml 文件中的 ports 段内容 生成对应每一个 pod 的规则

删除容器网络操作

删除 pod 的规则链表
情况 SNAT 和 DNAT 链表中容器

Flannel 源码简易解读及调用逻辑步骤

Flannel 和 Flanneld 的区别和关联

此处的 flannel 是 coreos 维护的，不是前面 CNI 里的 flannel。

- CNI 的 flannel 是二进制文件，而 coreos 的 flannel 实际上是 flanneld
- CNI 的 flannel 主要用于给容器分配和管理 IP 地址，而 coreos 的 flannel 主要用于通

过不同 backend 实现 pod 与 pod 间通信

- CNI 的 flannel 主要被 kubelet 在创建 Pod 时调用，CoreOS 的 flanneld 是一个长期运行的后台进程，用于监听 Api 的 Node 变化
- CNI 的 flannel 在给 Pod 分配 IP 地址时需要依赖 Flanneld 生成的配置文件 subnet.env，这个文件可以自己手动编写，然后 Pod 就可以正常的获取 IP 地址了。

核心概念

- network 负责网络的管理（以后的方向是多网络模型，一个主机上同时存在多种网络模式），根据每个网络的配置调用 subnet;
- subnet 负责和 etcd/kube 交互，把 etcd/kube 中的信息转换为 flannel 的子网数据结构，并对 etcd/kube 进行子网和网络的监听;
- backend 接受 subnet 的监听事件，负责增删相应的路由规则

代码解读

https://github.com/coreos/flannel

除了可执行文件的入口 main.go 之外，有 backend，network，pkg 和 subnet 这么几个代码相关的文件夹。

network 主要是 iptables 相关。主要是供 main 函数根据设置进行 MasqRules 和 ForwardRules 规则的设定。

pkg 主要是抽象封装的 ip 功能库。

backed 主要是后端实现，目前支持 udp、vxlan、host-gw 等。

subnet 子网管理。主要支持 etcdv2 和 k8s 两种实现。

Main 函数

1：注册所有的 backend

2：解析命令行参数列表

3：找到外部接口

iface 和 iface-regex 两个参数有关。这两个参数每一个可以指定多个。flannel 将按照下面的优先顺序来选取：

- 如果”–iface”和”—-iface-regex”都未指定时，则直接选取默认路由所使用的输出网卡
- 如果”–iface”参数不为空，则依次遍历其中的各个实例，直到找到和该网卡名或 IP 匹配的实例为止
- 如果”–iface-regex”参数不为空，操作方式和 2)相同，唯一不同的是使用正则表达式

去匹配

最后，对于集群间交互的 Public IP，我们同样可以通过启动参数”–public-ip”进行指定。否则，将使用上文中获取的网卡的 IP 作为 Public IP。

4：newSubnetManager 函数

创建 SubnetManager（sm）：子网管理器负责子网的创建、更新、添加、删除、监听等。它会去 listWatch k8s apiserver 的 node api，通过 ksm.events 与 backend 通信。ksm.events 是一个长度为 5000 的 subnet.Event chan，ksm 作为生产者，会将从 k8s 那里 watch 到的信息封装(nodeToLease，节点信息转为租约信息)后写到 ksm.events 里 去。

如果没有设置 kubernetes 子网管理的话，默认使用的就是通过 etcd 做子网管理；如果传参是 kube-subnet-mgr 那么将启动 kube 的子网管理。其实 kubernetes 没有网络管理，只有一个网络策略，见 subnet/kube/kube.go。所谓的网络管理就是 listwatch node 节点，原理就是放到 node 的 annotations，通过这种方式去替换 etcd 的作用，只不过不再直接 watch etcd 了，通过 kubernetes 的 api listwatch 机制，把 event 放到 subnet.Event 这个管道里面，然后 WatchLease 从管道中获取。这样设计相比直接用 etcd 的好处的话，是它可以配合网络策略实现网络隔离。应为有了这些节点信息，配合 calico 的网络策略，就可以很好的控制网络隔离，例如 canal。

5：getConfig 函数

获取网络配置，如果使用 etcd 类型子网管理器，则从 etcd 接口读取配置，如果使用 kube 类型子网管理器，则从 kubeapi 获取配置

- SubnetLen 表示每个主机分配的 subnet 大小，我们可以在初始化时对其指定，否则使用默认配置。在默认配置的情况下，如果集群的网络地址空间大于/24，则 SubnetLen 配置为 24，否则它比集群网络地址空间小 1，例如集群的大小为/25，则 SubnetLen 的大小为/26
- SubnetMin 是集群网络地址空间中最小的可分配的 subnet，可以手动指定，否则默认配置为集群网络地址空间中第一个可分配的 subnet。
- SubnetMax 表示最大可分配的 subnet
- BackendType 为使用的 backend 的类型，如未指定，则默认为“udp”
- Backend 中会包含 backend 的附加信息，例如 backend 为 vxlan 时，其中会存储 vtep 设备的 mac 地址

6：创建 backend.Manager，使用它来创建 backend、注册网络，然后执行其 run 函数

RegisterNetwork 过程

在需要后端要管理某个网络的时候被调用，目标就是注册要管理的网络信息到自己的

数据结构中，以便后面使用

Vxlan

1：创建 flannel（vxlan）接口。netlink 下内核创建 flannel.1 接口，需要指定 vni（默认为 1），出接口，源地址，目的端口，nolearning(netlink.Vxlan)。相当于：

ip link add $DEVNAME type vxlan id $VNI dev eth0 local $IP dstport $PORT nolearning

2：调用 kube subnet manager 获得一个子网

3：如果从 apiserver 查询 node 的 backend Annotations 跟实际 node 的信息不一致(例如 VTEP MAC 不同)，则向 node 打 patch，更新 Anootations。此更新会被其他 node watch 到，从而更新其本地的 fdb table

4：设置 flannel.1 接口地址，激活接口，并增加网段路由。相当于

ip address add $VXSUBNET dev $DEVNAME

ip link set $DEVNAME up

ip route add $SUBNET dev $DEVNAME scope global

5：返回一个符合 backend.Network 接口的 netwrok 结构。

注：在生产 backend 以后，会启动一个协程，在 flanneld 退出运行之前，将会执行激活的 backend map 中删除操作。

7：ipMasq 建立 iptables 规则

8：WriteSubnetFile 函数

将配置信息写入文件/run/flannel/subnet.env 中

Run 函数

在 forEachNetwork()中会遍历 m *Manager 中启动的网络模式，存放到参数 n 中，对于每一个网络 n，启动一个 groutine 执行 func (m *Manager) runNetwork(n *Network)

runNetwork(n *Network)

Run()的核心就是循环调用 runOnce()，其核心逻辑如下：

读取 etcd/kube 中的值，根据获得的值做一些初始化的工作

retryInit()调用 backend 的 Run 函数，让 backend 在后台运行 n.bn.Run(ctx)

监听子网在 etcd/kube 中的变化，执行对应的操作

在 goroutine 里监听所有从 ksm.events 里监听到的 subnet 事件，然后交由处理函数 handleSubnetEvents 完成。

handleSubnetEvents 函数

事件主要是 subnet.EventAdded 和 subnet.EventRemoved 两个。

Vxlan

subnet Added 事件，添加 ARP 表、添加 FDB 表、更新路由表。
subnet Removed 事件，删除路由表、删除 FDB 表、删除 ARP 表。

ARP 表

当应用访问 VXLAN 网络里的另一个应用时，我们只有一个 IP 地址，这时需要将 IP 转换成 MAC 地址，才能在数据链路层通信。所以 ARP 表记录的就是 IP 与 VTEP MAC 的映射。

在 flannel 网络中，flanneld 会在启动时监听 RTM_GETNEIGH(vxlanDevice.MonitorMisses)，flanneld 相当于 arp daemon；arp 请求会在内核中通过 netlink 发送请求到用户态的 flanneld，由 flanneld 根据该 arp 请求 ip 所属网段(该 node 上所有 ip 的 MAC 也就是 VTEP 的 MAC)，查询其记录的 routes 信息，然后 netlink 下发内核。

相当于
ip neighbor add/replace $ip-on-node-2 lladdr $mac-of-vtep-on-node-2 dev flannel.1
查看：ip neighbor show dev flannel.1

FDB 表

当我们知道了 VTEP MAC 的地址后，我们仍不知道这个 VTEP 在那一台主机上，也不知道 VTEP 在哪个 VXLAN 网络（即 VNI、VXLAN Network Identifier 是否一样）内，仍然无法通信。这时候就需要记录，VTEP MAC 与 VTEP 的 UDP 地址，及该 VTEP 所属的 VNI。

跟数据中心的 vxlan 不一样，flannel 网络中的 VTEP 的 MAC 并不是通过组播学习的，而是通过 apiserver 去做的同步(或者是 etcd)。各个节点会将自己的 VTEP 信息上报给 apiserver，而 apiserver 会再同步给各节点上正在 watch node api 的 listener(flanneld)，flanneld 拿到了更新消息后，再通过 netlink 下发到内核，更新 fdb 表项，从而达到了整个集群的同步。

相当于
bridge fdb add/append $mac-of-vtep-on-node-2 dev $DEVNAME dst $DESTIP
查看：bridge fdb show dev flannel.1

host-gw

subnet Added 事件，添加路由表。
subnet Removed 事件，删除路由表。

Vxlan

说明

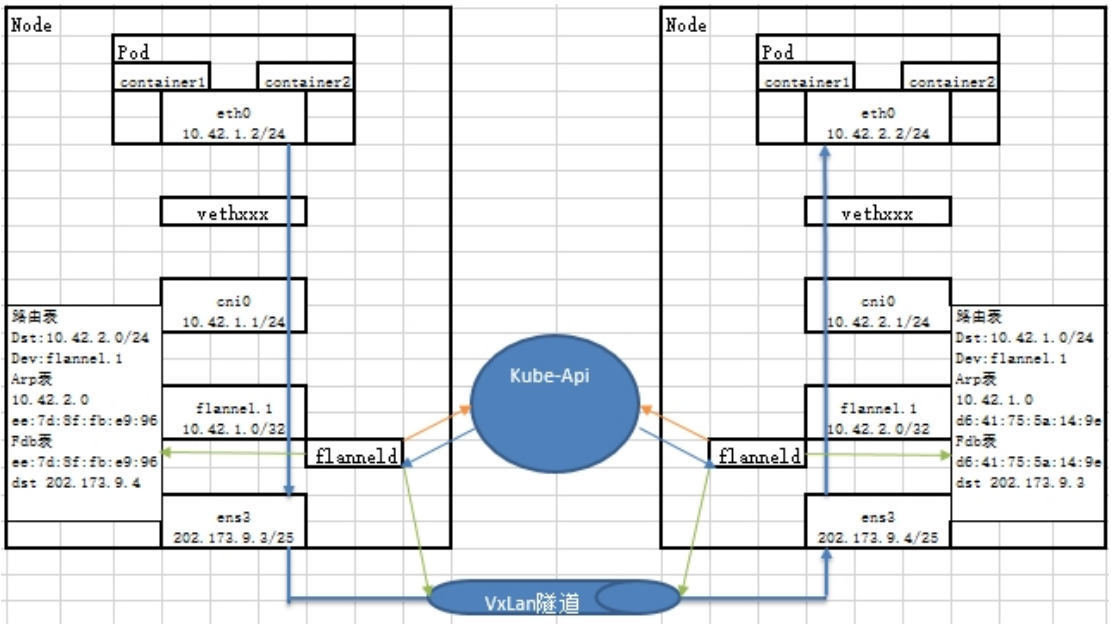
Vxlan 是一种基于 IP 网络(L3)的基础上虚拟 L2 网络连接的解决方案。为多租户平台提供了虚拟网络强大的扩展能力和隔离性。是"软件定义网络"(Software-defined Networking, 简称 SDN)的协议之一。

通俗来讲，vxlan 在多个主机原有的 IP 网络（可能无法直接在 L2 直接通信）中抽象出很多自定义的网络。

这里有一个关键的设备 vtep(VXLAN Tunnel End Point)承担了自定义虚拟子网中不同网段的 L2 通信的转发。

注意：Linux 内核的 vxlan 支持，要求版本 3.7+，推荐升级到 3.9+。

数据流程图



1：容器里发出数据包，经过容器网卡 eth0，通过 veth 到达主机 vethxxxx 设备

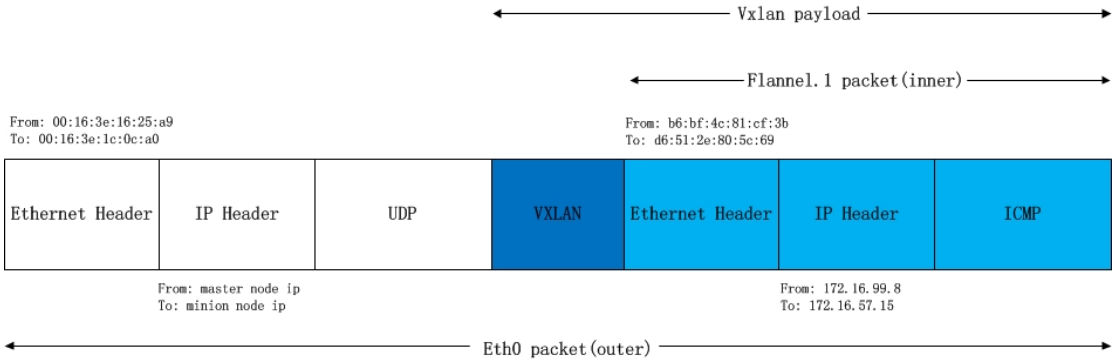
2：主机端 veth 设备通过 bridge 桥接到 cni0 网卡，发现数据包并不是给自己，寻找下一跳

3：查找路由表：ip route，发现该数据包应该通过本机的 vetp 设备 flannel.1 接口发出，网关地址是对端主机 flannel.1 接口的 IP

4：由于 flannel.1 是 vetp 设备，会对通过它的数据根据 vxlan 的协议标准进行二层封装转发，而二层转发的前提是需要获取对端主机的 mac 地址。vxlan 并不会在二层发 arp 包，而是由 linux kernel 的 “L3 MISS"事件，将 arp 发到用户空间和 flanneld 程序。flanneld 程序收到“L3 MISS”内核事件以及 ARP 请求后，并不会向外网发送 arp request，而是从 kube-api 的 node 信息查找匹配该地址的子网的 vtep 信息。找到目的地后，flanneld 将查询到的信息存入 arp 缓存

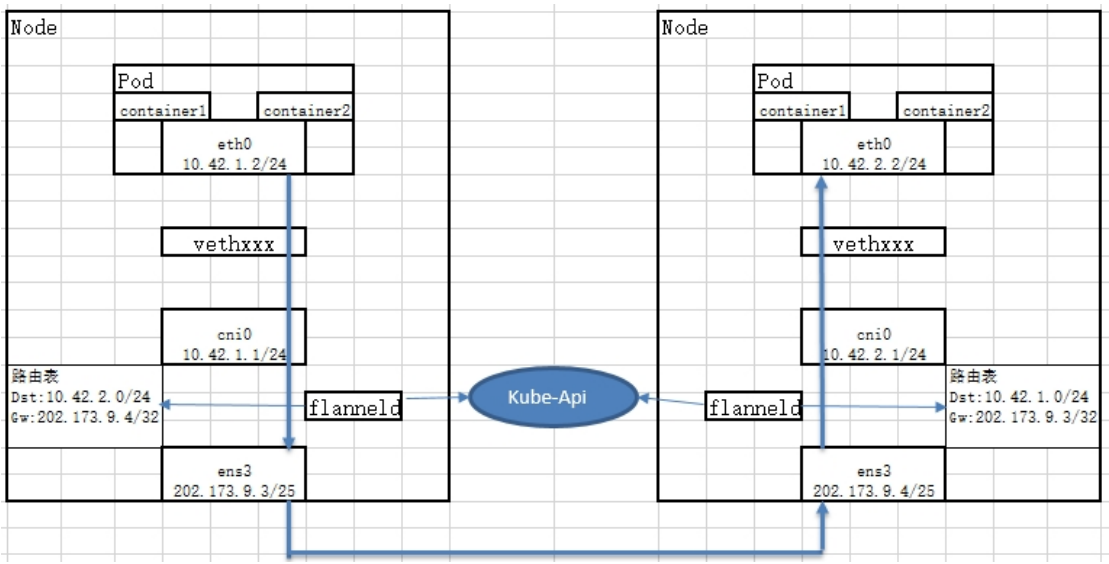
5：查找 arp 表：ip neigh show，查到对端主机 flannel.1 接口的 IP 的 MAC 地址，然后进行封装。
6：查看 fdb 表：bridge fdb show dev flannel.1，得到对端的 IP 地址后进行 UDP 发送，目的端口是 8472.
7：对端节点接受到数据包之后，会识别出这是一个 vxlan 的封包，并将包交给对应的 vtep 设备 flannel.1，再经由 bridge 传送给 container。

数据包内容格式



Host-gw

数据流程图



局限性

host-gw 要求主机网络二层直连，或者说跨宿主机网络通信需要物理路由支持

与 Vxlan 差异

适用性

host-gw 只能运行在二层网络（开启 Directrouting 另说）
Vxlan 可以运行在二层或者三层

数据发送方式

host-gw 把每个主机配置成网关，直接路由原始封包
Vxlan 则是是在主机间建立隧道，让不同的主机在一个大的网内

性能

由于 vxlan 需要对数据进行封包和拆包，性能低于 host-gw。且 Vetp 设备（flannel.1）在包转发过程中存在用户空间与内核空间切换的过程造成额外开销。

网卡

host-gw 只需要一个桥接网卡即可
vxlan 则需要多一个 flannel.1 的 vetp 设备网卡

MTU

host-gw 的桥接网卡 cni0 的 MTU 是 1500
vxlan 由于需要经过 flannel.1 进行 vxlan 封装，因此 cni0 网卡的 MTU 是 1450

SNAT

三种实现方式

CNI

https://github.com/containernetworking/plugins/blob/master/pkg/ip/ipmasq_linux.go

特征说明

SNAT 规则是在 pod 被创建时调用 ip.SetupIPMasq 函数生成的。
针对每一个 pod 都有一个 SNAT 规则。
默认 multicastNet "244.0.0.0/24"不进行 masquerade。

源码步骤

1：为 pod 创建对应的 chain
2：在 chain 中 ACCEPT 接收所有发往该节点 subnet 的数据包
3：在 chain 中对多播包以外的数据包进行 masquerade
4：将所有流量导入 chain 中

配置

cni-conf.json

```
"delegate":{
   "ipMasq": true,
   ……
}
```

Iptables 效果

Chain POSTROUTING (policy ACCEPT)
num target prot opt source destination
3 CNI-28dca6e4a81e0524babddafb all -- 10.42.3.0/24 0.0.0.0/0 /* name: "cbr0" id: "95e5d3009eb344738968bce9381ef1ce6d2fc8c3084a1bff3bde40ded094ee37" */

Chain CNI-28dca6e4a81e0524babddafb (1 references)
num target prot opt source destination
1 ACCEPT all -- 0.0.0.0/0 10.42.3.0/24 /* name: "cbr0" id: "95e5d3009eb344738968bce9381ef1ce6d2fc8c3084a1bff3bde40ded094ee37" */
2 MASQUERADE all -- 0.0.0.0/0 !224.0.0.0/4 /* name: "cbr0" id: "95e5d3009eb344738968bce9381ef1ce6d2fc8c3084a1bff3bde40ded094ee37" */

Flanneld

https://github.com/coreos/flannel/blob/master/network/iptables.go

特征说明

SNAT 规则是在部署 flanneld 时调用 MasqRules 函数生成的
SNAT 规则是针对集群和 subnet 的，不是针对 pod 的
Pod 创建时不再生成单独的 SNAT 规则

源码步骤

1：设置集群内部 cluster-cidr 数据包进行流量转发（不做 SNAT）
2：对多播包以外的数据包进行 masquerade
3：集群外访问本节点 subnet 的数据包进行流量转发（不做 SNAT）
4：集群外访问集群内部 cluster-cidr 数据包进行 masquerade

配置

/opt/bin/flannel --ip-masq

Iptables 效果

Chain POSTROUTING (policy ACCEPT)

num	target	prot	opt	source	destination	
3	0	0	RETURN	all --	*	*	10.42.0.0/16	10.42.0.0/16
4	0	0	MASQUERADE	all --	*	*	10.42.0.0/16	!224.0.0.0/4
5	0	0	RETURN	all --	*	*	!10.42.0.0/16	10.42.4.0/24
6	0	0	MASQUERADE	all --	*	*	!10.42.0.0/16	10.42.0.0/16

ip-masq-agent 组件

http://acs-public.oss-cn-hangzhou.aliyuncs.com/kubernetes/network/ip-masq-agent.yaml

意义

灵活的控制集群的 SNAT 规则

特征说明

默认 SNAT 所有数据包，针对自定义的多个网络进行排除
需要通过 DaemonSet 方式部署一个 Pod，赋予特权进行 iptables 修改

配置

Vim ip-masq-agent.yaml

```
    nonMasqueradeCIDRs:
      - 10.42.0.0/16
```

后期变更通过修改 ConfigMap 实现，需要等待配置同步
kubectl edit configmaps -n kube-system ip-masq-agent

Iptables 效果

Chain POSTROUTING (policy ACCEPT)
num target prot opt source destination
3 IP-MASQ-AGENT all -- 0.0.0.0/0 0.0.0.0/0 /* ip-masq-agent: ensure nat POSTROUTING directs all non-LOCAL destination traffic to our custom IP-MASQ-AGENT chain */ ADDRTYPE match dst-type !LOCAL

Chain IP-MASQ-AGENT (1 references)
num target prot opt source destination
1 RETURN all -- 0.0.0.0/0 10.42.0.0/16 /* ip-masq-agent: cluster-local traffic should not be subject to MASQUERADE */ ADDRTYPE match dst-type !LOCAL
2 MASQUERADE all -- 0.0.0.0/0 0.0.0.0/0 /* ip-masq-agent: outbound traffic should be subject to MASQUERADE (this match must come after cluster-local CIDR matches) */ ADDRTYPE match dst-type !LOCAL

注意：IpMasq 和 SetupIPMasq

https://github.com/containernetworking/plugins/blob/master/plugins/meta/flannel/flannel_linux.go

```
 if !hasKey(n.Delegate,
 "ipMasq") {
                                    // if flannel is not doing ipmasq, we should
                                    ipmasq := !*fenv.ipmasq
                                    n.Delegate["ipMasq"] = ipmasq
```

如果 Flanneld 没有开启 ipmasq，CNI 的 flannel 则默认会取反（开启 ipmasq，源码里注

释分部分写 flannel 实际上应该是 flanneld）。

如果想同时关闭两者，则需要将 cni-conf.json 中 delegate.ipMasq 置为 false，然后 flanneld 运行是不指定 ip-masq 即可。

如果 Flanneld 和 CNI 的配置文件 cni-conf.json 都开启 ipmasq，则两则都会生效（iptables 规则中同时存在）