### 网络栈

Linux当中的一个网络栈通常由这几个部分构成：网卡(Network Interface)、路由表(Route Table)、回环设备(Loopback Device)、iptables规则。

回环设备的作用：

在Unix-like系统中，它是一个虚拟设备(pseudo-device)，使得计算机文件可以作为块设备来访问。

虚拟设备pseudo -devices：在Unix-like OS当中，并不是所有的设备节点都需要和真实的物理设备相连，所有没有这种关联(lack this correspoondence)的设备组成了 pseudo-device 组.

![](data:text/html; charset=utf-8;base64,)

Linux的系统实现结构

### 容器网络与交互

#### 容器自身的网络

存在两种方式来进行：

1. 直接使用宿主机的网络。例如:docker run -d -net=host --name nginx-host nginx
2. 利用Network Namespace实现隔离独立的网络栈。

**在拥有自己独立的网络栈之后，如何实现不同容器间的网络交互？**

为了实现多台容器之间的交互，需要类似于交换机功能的设备支持，在Linux当中，**起到虚拟交换功能的设备是网桥（Bridge）**，工作在数据链路层，通过MAC进行数据包转发。

Docker默认在宿主机上创建一个docker0网桥，通过连接他，来实现容器间的通信。

如何将一个容器“连接”到docker0网桥上呢？

使用一种名为**Veth Pair**的虚拟设备，它有以下几个特点：

1. 一旦创建出来，都是以**两张虚拟网卡**的形式成对出现。
2. 两张网卡可以直接进行互通，与Network Namspace无关。

Veth Pair 可以视为网线一般的作用，一端网卡连接容器，另一端连接网桥。

下面以一个例子来解释：

创建一个nginx-1：

docker run -d --name=nginx-1 nginx

进入**容器nginx-1**并查看其网络配置，发现创建了一个eth0网卡，同时route查看路由表，发现eth0是默认的路由设备，负责处理所有172.17.0.0/16的所有路由：

root@6f87d7062b75:/# ifconfig  
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
 inet 172.17.0.2 netmask 255.255.0.0 broadcast 0.0.0.0  
 inet6 fe80::42:acff:fe11:2 prefixlen 64 scopeid 0x20<link>  
 ether 02:42:ac:11:00:02 txqueuelen 0 (Ethernet)  
 RX packets 5529 bytes 9088975 (8.6 MiB)  
 RX errors 0 dropped 0 overruns 0 frame 0  
 TX packets 5453 bytes 380548 (371.6 KiB)  
 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  
 inet6 ::1 prefixlen 128 scopeid 0x10<host>  
 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  
   
root@6f87d7062b75:/# route  
Kernel IP routing table  
Destination Gateway Genmask Flags Metric Ref Use Iface  
default 172.17.0.1 0.0.0.0 UG 0 0 0 eth0  
172.17.0.0 0.0.0.0 255.255.0.0 U 0 0 0 eth0

路由规则的网关（Gateway）是 0.0.0.0，这就意味着这是一条**直连**规则，即：凡是匹配到这条规则的 IP

包，**应该经过本机的 eth0 网卡，通过二层网络直接发往目的主机。**

在外部查看**宿主机**的网络,可以看到创建了网桥docker0和对应的veth pair网卡设备：

[lyq@localhost go]$ ifconfig  
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
 inet 172.17.0.1 netmask 255.255.0.0 broadcast 0.0.0.0  
 inet6 fe80::42:75ff:febb:8b19 prefixlen 64 scopeid 0x20<link>  
 ether 02:42:75:bb:8b:19 txqueuelen 0 (Ethernet)  
 RX packets 8 bytes 544 (544.0 B)  
 RX errors 0 dropped 0 overruns 0 frame 0  
 TX packets 14 bytes 1766 (1.7 KiB)  
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  
   
vethc2b67ca: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
 inet6 fe80::5c53:6dff:fe64:b36f prefixlen 64 scopeid 0x20<link>  
 ether 5e:53:6d:64:b3:6f txqueuelen 0 (Ethernet)  
 RX packets 8 bytes 656 (656.0 B)  
 RX errors 0 dropped 0 overruns 0 frame 0  
 TX packets 22 bytes 2422 (2.3 KiB)  
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

之后新建另一个niginx-2，查看其网络配置也是同理：

//略

宿主机也会生成一个新的veth设备:

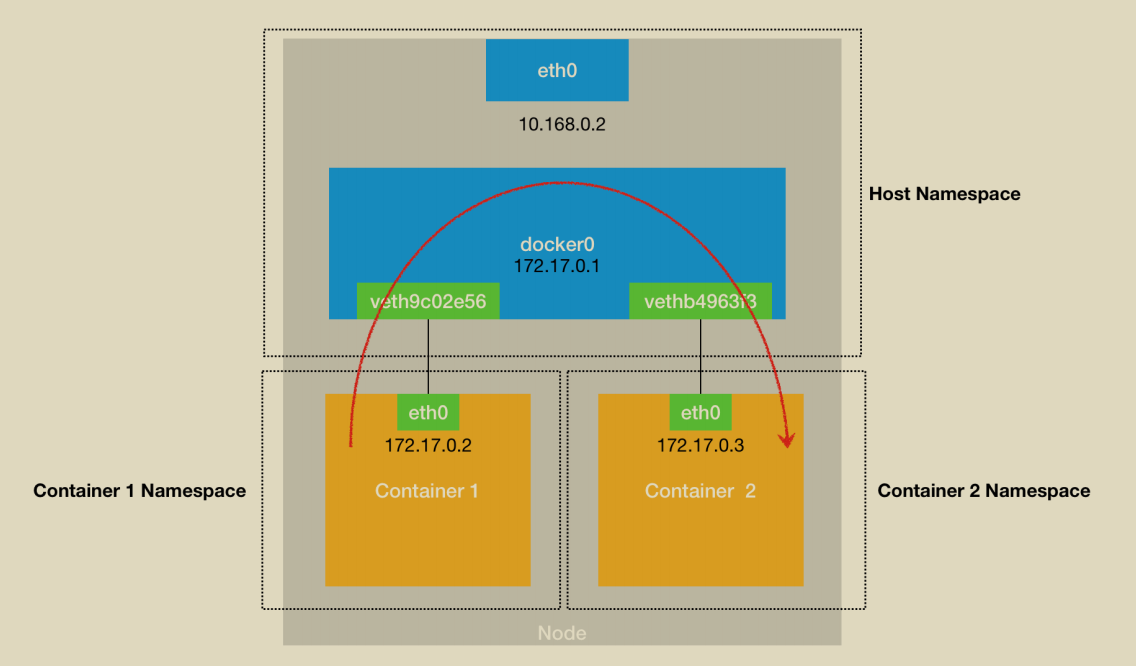
[lyq@localhost go]$ ifconfig  
veth007d232: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  
 inet6 fe80::c800:adff:fe35:aa74 prefixlen 64 scopeid 0x20<link>  
 ether ca:00:ad:35:aa:74 txqueuelen 0 (Ethernet)  
 RX packets 8 bytes 656 (656.0 B)  
 RX errors 0 dropped 0 overruns 0 frame 0  
 TX packets 8 bytes 656 (656.0 B)  
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

通过brctl show查看网关设备，可以发现那两个veth pair的网卡都插在网桥docker0上：

[lyq@localhost go]$ brctl show  
bridge name bridge id STP enabled interfaces  
docker0 8000.024275bb8b19 no veth007d232  
 vethc2b67ca  
virbr0 8000.5254009a25c6 yes virbr0-nic

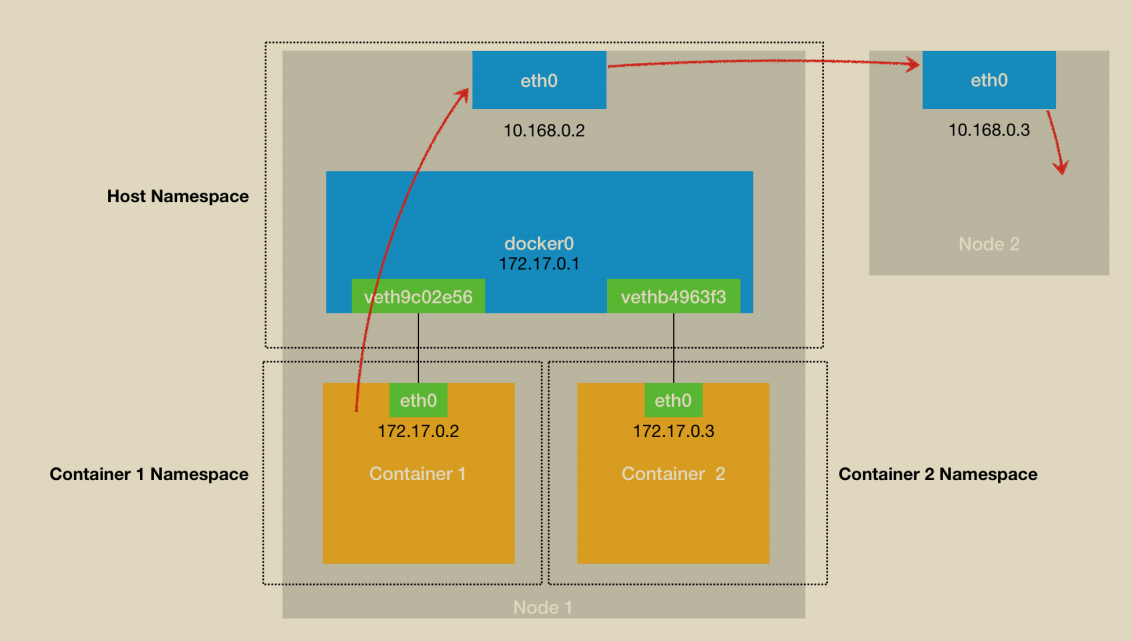
此时，在nginx-1当中ping容器nginx-2：可以发现是畅通的。

整个网络的处理流程如下图所示：

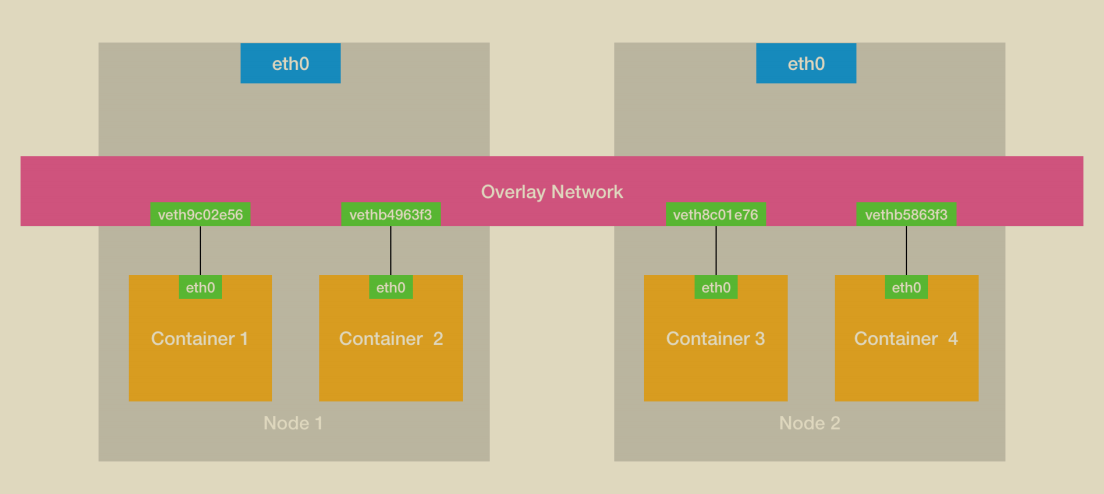


docker0充当一个二层交换机的作用，利用ARP广播找到IP对应的MAC地址。

#### 容器连接其他宿主机



docker的默认配置下，是不允许不同宿主机上的容器进行IP互相访问的，为了创建公用的网桥将所有容器都联合到一起，便于所有容器都可以一起通信，使用软件的方法构建如下的形式：



要在已有的宿主机网络上，再通过软件构建一个覆盖在已有宿主机网络之上的、可以把所有容器连通在一起的虚拟网络，称之为Overlay Network(覆盖网络).

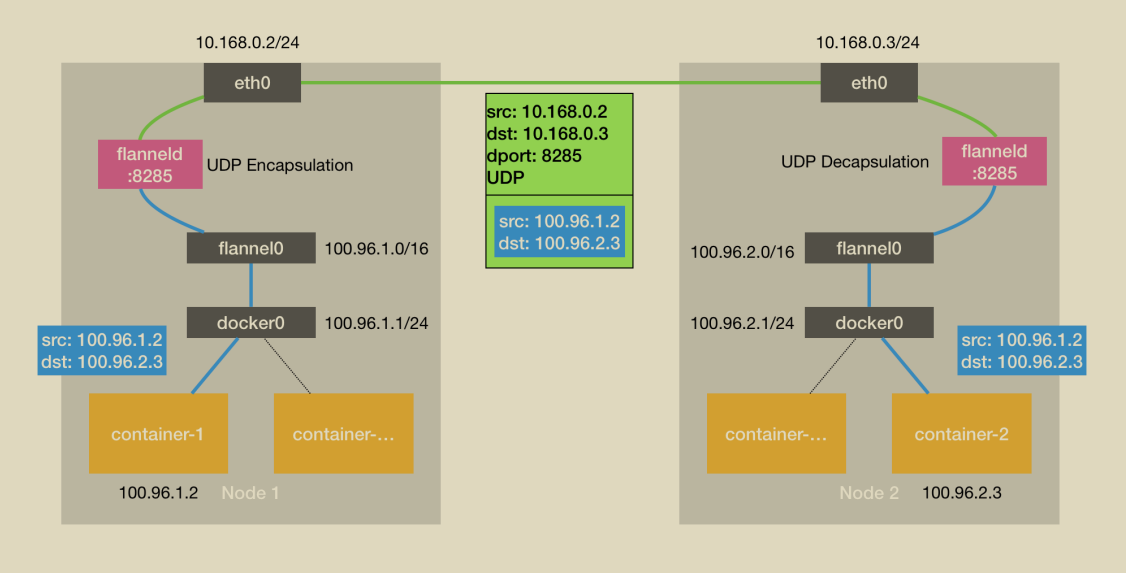
### 跨主机网络

为了解决跨主机通信的问题，社区出现了许多的通信方案。从CoreOS的Flannel项目说起，该项目的后端实现提供了三种主流的网络实现方案。

1. UDP
2. VXLAN
3. host-gw

#### UDP模式

最早得到支持，却是性能最差的，已经被弃用。



基于Flannel的UDP模式的跨主机通信的基本原理

流程中几个关键的步骤和问题所在：

首先，flannel0设备，其本身是TUN（Tunnel）设备，工作在第三层，用于在OS的内核与用户程序之间进行IP包的传输，即在内核态和用户态之间进行传输。

数据包进入docker0之后，由于路由规则并不匹配，会出现在宿主机当中。根据Flannel先创建好的路由规则，这个IP包会进入flannel0设备当中，flannel0设备获得了IP包，会将其**传递到Flanneld进程**当中（位于**用户态**，Flannel项目**在每个宿主机上的主进程**），随后将其转发给Node2.

第二个关键问题：flanneld是如何知晓目的地是Node2的呢？

1. **子网**。每一个Flannel的容器网络中，一个宿主机上的不同容器都同属于一个相同的子网，从图中可以得到Node1的子网是100.96.1.0/24，Node—100.96.2.0/24。而宿主机和其子网的对应关系存储在etcd当中。

$ etcdctl ls /coreos.com/network/subnets  
/coreos.com/network/subnets/100.96.1.0-24  
/coreos.com/network/subnets/100.96.2.0-24  
/coreos.com/network/subnets/100.96.3.0-24

查看找子网对应的宿主机IP地址：

$ etcdctl get /coreos.com/network/subnets/100.96.2.0-24  
{"PublicIP":"10.168.0.3"}

第三个关键问题：整个流程需要正确，则要使得docker0网桥的地址范围在flannel为宿主机分配的子网之内。可以进行设置：

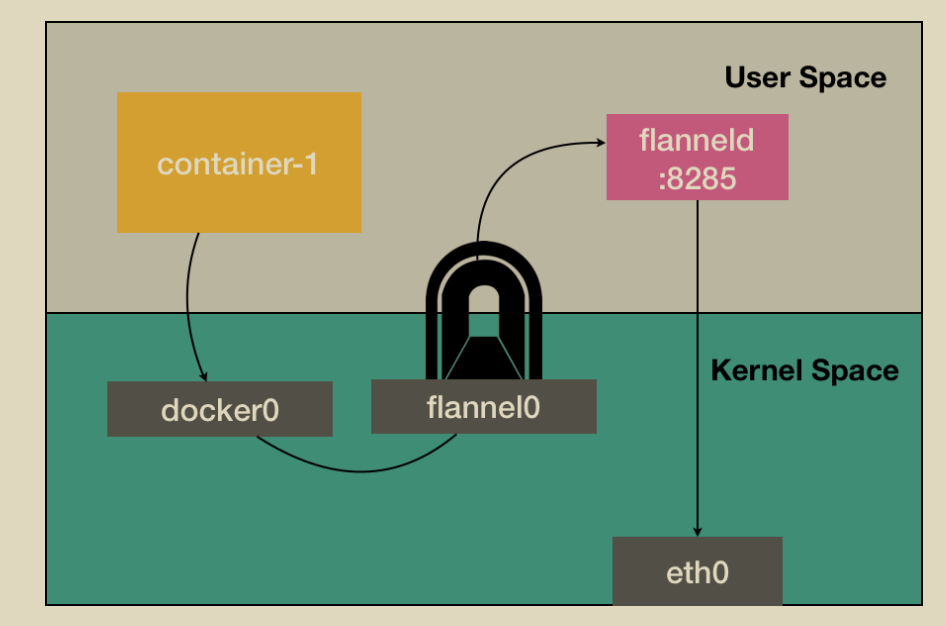
$ FLANNEL\_SUBNET=100.96.1.1/24  
$ docker --bip=$FLANNEL\_SUBNET ...

**整体而言，UDP模式提供了一个三层的Overlay网络。**给不同宿主机间打通了隧道。

##### 问题所在—性能

**用户态和内核态的切换过于频繁。**

在发出IP包时，产生的切换：



**我们在进行系统级编程的时候，有一个非常重要的优化原则，就是要减少用户态到内核态的切换次数，并且把核心的处理逻辑都放在内核态进行**。

#### VXLAN模式—未完

VXLAN—Virtual Extensible LAN(虚拟可扩展局域网)，Linux本身支持的网络虚拟化技术。

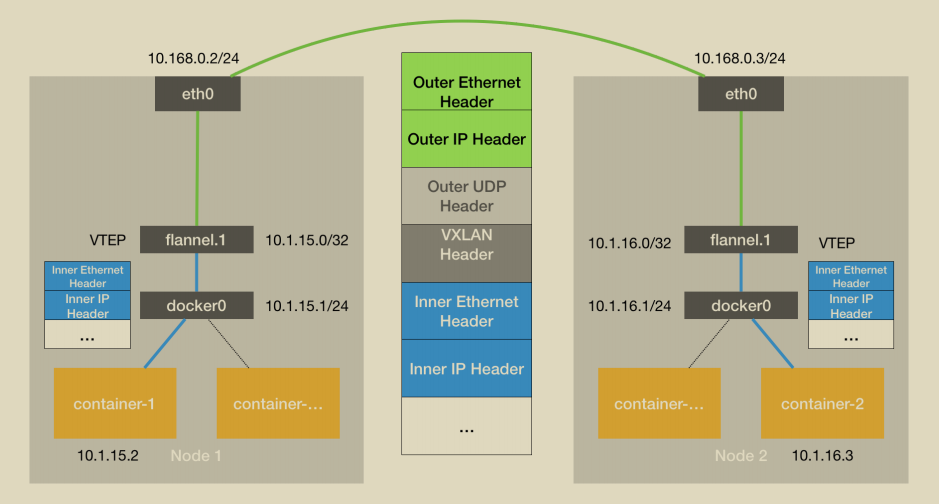
和UDP相比，它的优势在于：**VXLAN 可以完全在内核态实现上述封装和解封装的工作**，从而通过与前面相似的“隧道”机制，构建出覆盖网络（Overlay Network）。

设计思想：在已有的三层网络基础之上，再**添加一层由OS实现的VXLAN模块负责维护的二层网络**，达到一个类似局域网的效果。

为了能在二层网络上打通“隧道”，需要在宿主机上设置一个特殊的网络设备作为“隧道”的两端，即**VTEP设备**（虚拟隧道端点）。

VTEP设备的作用：

* 处理的对象是二层的数据帧
* 整个流程都在内核中进行



flannel.1即是VTEP设备，同时具有IP和MAC地址。

流程当中的几个关键：

首先，IP包从container1中传输到flannel.1当中后，进入了VTEP隧道的入口，之后需要找到出口的目的地址，**如何构建VTEP设备间的互连？**

当一个节点加入flannel的容器网络之后，所有节点都会添加一条路由规则，以此得知**目的VTEP设备的IP地址**，例如，当Node2加入后，Node1就会有一条如下的路由规则：

$ route -n  
Kernel IP routing table  
Destination Gateway Genmask Flags Metric Ref Use Iface  
...  
10.1.16.0 10.1.16.0 255.255.255.0 UG 0 0 0 flannel.1

指明发往10.1.16.0/24的包，都需要通过接口flannel.1进行转发，目的网关是10.1.16.0，也就是Node2的flannel.1的IP。

其次，VXLAN是希望构成二层数据帧的网络，**如何实现数据帧格式的传输，如何获取对应目的VTEP设备的MAC地址？**

通过ARP表，根据IP地址查询到MAC地址。

数据帧的格式如下，在之前的数据包的基础之上，添加头部信息Header，即图中Inner Ethernet Header：

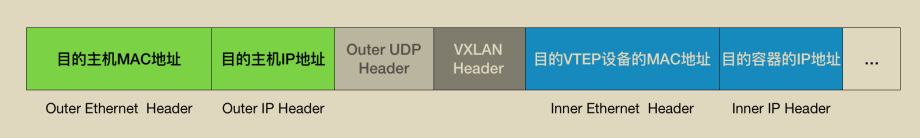


需要注意的是，在实际的Linux数据传输中，它并不能进入宿主机网络的二层网络进行数据帧传输，而会被进一步封装成宿主机网络的数据帧，通过eth0网卡传输。

VXLAN这个是标记，表示它是VXLAN需要使用的数据帧。其中一个重要的标记VNI，是VTEP用来识别该数据帧是否归自己来处理的重要标识。默认为1，也就是flannel.1的来由。

随后会将其以UDP的形式发送出去，而UDP中目的宿主机的IP地址，可以通过之前VTEP设备的目的IP地址获得。

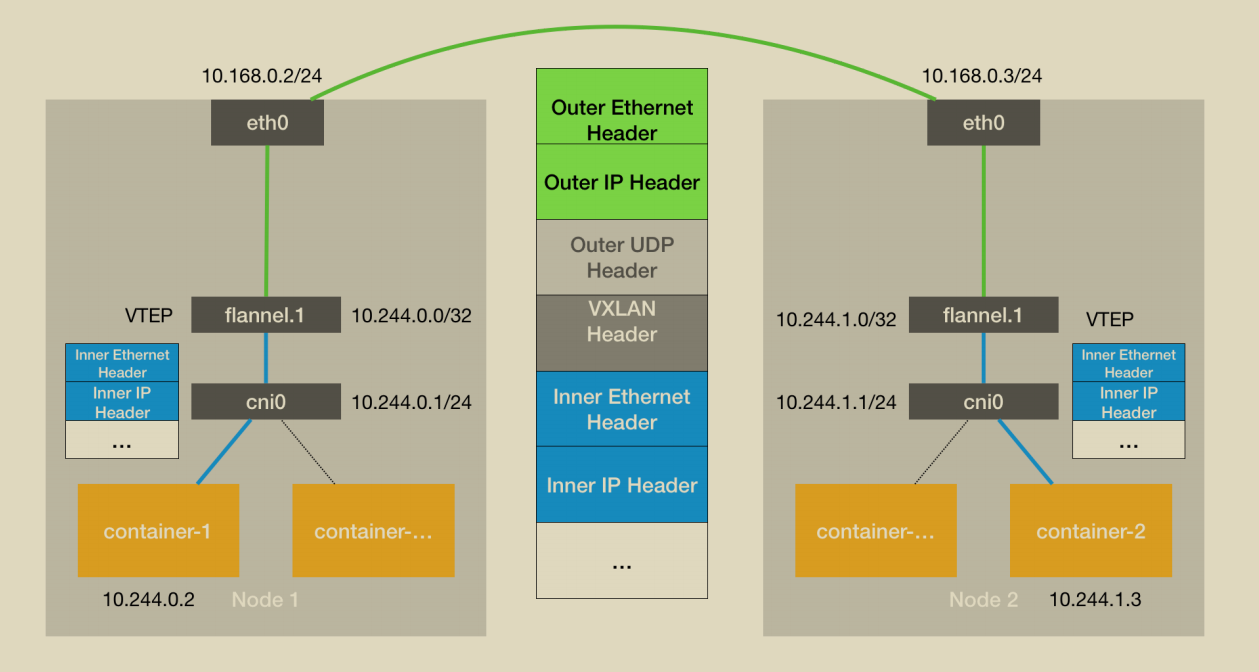
最终构成的数据帧形式为：



### K8s的网络模型

K8s对于网络的处理流程和上面提到的VXLAN的处理方式基本一样，区别在它使用CNI接口替代了docker0网桥。

整个的流程如下所示：



需要注意的是：，CNI 网桥只是接管所有 CNI 插件负责的、即 Kubernetes 创建的容器（Pod）。如果额外使用docker run 创建容器，则依旧会连接到docker0网桥之上。

使用CNI网桥而放弃docker0的原因主要有以下两方面：

1. K8s没有使用Docker的网络模型，不希望、也没有能力使用docker0
2. 这也和Pod的配置有关，即Infra容器的Network Namespace有关。

### CNI网络插件—-未完

CNI的设计思想是这样的：**Kubernetes 在启动 Infra 容器之后，就可以直接调用CNI 网络插件，为这个 Infra 容器的 Network Namespace，配置符合预期的网络栈。**

### k8s的网络隔离—未完

### k8s的三层网络方案—未完

### Service、DNS和服务发现

典型的Service定义：

apiVersion: v1  
kind: Service  
metadata:  
 name: hostnames  
spec:  
 selector:  
 app: hostnames  
 ports:  
 - name: default  
 protocol: TCP  
 port: 80  
 targetport: 9376

使用spec.selector选择Pod，被选中的Pod称为Endpoints，通过kubectl get endpoints hostnames可以查看到，只有**处于Running状态并且readinessProbe检查通过的Pod才会出现在Service的Endpoints列表当中**。

#### ClusterIP模式下的Service

查看:

$ kubectl get svc hostnames  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE  
hostnames ClusterIP 10.0.1.175 <none> 80/TCP 5s

访问Service的VIP，即CLUSTER-IP就可以知道它所代理的Pod:

$ curl 10.0.1.175:80  
hostnames-0uton

每次访问都会返回不同的Pod，这是因为该模式下**Service采用的是轮询模式来进行负载均衡。**

#### Service的实现原理

**Service是由kube-proxy组件和iptables来实现的。**

一旦Service被提交，则kube-proxy可以通过Service的Informer立即感知到该对象的添加，随后，他会在宿主机上创建一条iptables规则：

这条规则会为Service的VIP访问设置一个固定的入口地址，跳转到一个固定的iptables规则上进行处理。

这个iptables规则是一组规则的集合，是**一组随机模式(-mode random)的iptables链**，这些链分别指向***Service所代理的Pod，所以这些链也是实现负载均衡的位置。***

iptables规则的匹配，从上至下按顺序进行的，为了保证被选中的概率相同，需要对probability字段的值进行设置。

具体的转发原理：这些规则都是DNAT规则，其作用就是在路由之前（PREROUTING检查点之前），**将流入的IP包的目的地址和端口，改成-to-destination所指定的新的目的地址和端口，也就是那些被代理的Pod的IP地址和端口。**

一系列处理之后，**原本访问Service 的VIP的IP包已经被修改为了访问被代理的Pod的IP包。**

#### IPVS模式的Service

##### CulsterIP的缺陷

Service的需求：1. Pod的IP并不固定 2. 一组Pod实例之间会有负载均衡的需求

#### 从外界访问Service–未完

### Service与Ingress