Tony Bai

一个程序员的心路历程

- 关于我
- 文章列表

理解Kubernetes网络之Flannel网络

- 一月 17, 2017
- 3条评论

第一次采用kube-up.sh脚本方式安装的Kubernetes cluster目前运行良好，master node上的组件状态也始终是"没毛病"：

```
# kubectl get cs
NAME                 STATUS    MESSAGE              ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   {"health": "true"}
```

不过在第二次尝试用kubeadm安装和初始化Kubernetes cluster时遇到的各种网络问题还是让我"心有余悸"。于是趁上个周末，对Kubernetes的网络原理进行了一些针对性的学习。这里把对Kubernetes网络的理解记录一下和大家一起分享。

Kubernetes支持Flannel、Calico、Weave network等多种cni网络Drivers，但由于学习过程使用的是第一个cluster的Flannel网络，这里的网络原理只针对k8s+Flannel网络。

一、环境+提示

凡涉及到Docker、Kubernetes这类正在active dev的开源项目的文章，我都不得不提一嘴，那就是随着K8s以及flannel的演化，本文中的一些说法可能不再正确。提醒大家：阅读此类技术文章务必结合"环境"。

这里我们使用的环境就是我第一次建立k8s cluster的环境：

```
# kube-apiserver --version
Kubernetes v1.3.7

# /opt/bin/flanneld -version
0.5.5

# /opt/bin/etcd -version
etcd Version: 3.0.12
Git SHA: 2d1e2e8
Go Version: go1.6.3
Go OS/Arch: linux/amd64
```

另外整个集群搭建在阿里云上，每个ECS上的OS及kernel版本：Ubuntu 14.04.4 LTS，3.19.0-70-generic。

在我的测试环境，有两个node：master node和一个minion node。master node参与workload的调度。所以你基本可以认为有两个minion node即可。

二、Kubernetes Cluster中的几个"网络"

之前的k8s cluster采用的是默认安装，即直接使用了配置脚本中(kubernetes/cluster/ubuntu/config-default.sh)自带的一些参数，比如：

```
//摘自kubernetes/cluster/ubuntu/config-default.sh

export nodes=${nodes:-"root@master_node_ip root@minion_node_ip")
export SERVICE_CLUSTER_IP_RANGE=${SERVICE_CLUSTER_IP_RANGE:-192.168.3.0/24}
export FLANNEL_NET=${FLANNEL_NET:-172.16.0.0/16}
```

从这里我们能够识别出三个"网络"：

- node network：承载kubernetes集群中各个"物理"Node(master和minion)通信的网络；
- service network：由kubernetes集群中的Services所组成的"网络"；
- flannel network： 即Pod网络，集群中承载各个Pod相互通信的网络。

node network自不必多说，node间通过你的本地局域网（无论是物理的还是虚拟的）通信。

service network比较特殊，每个新创建的service会被分配一个service IP，在当前集群中，这个IP的分配范围是192.168.3.0/24。不过这个IP并不"真实"，更像一个"占位符"并且只有入口流量，所谓的"network"也是"名不符实"的，后续我们会详尽说明。

flannel network是我们要理解的重点，cluster中各个Pod要实现相互通信，必须走这个网络，无论是在同一node上的Pod还是跨node的Pod。我们的cluster中，flannel net的分配范围是：172.16.0.0/16。

在进一步挖掘"原理"之前，我们先来直观认知一下service network和flannel network：

Service network(看cluster-ip一列)：

```
# kubectl get services
NAME          CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
index-api     192.168.3.168   <none>        30080/TCP   18d
kubernetes    192.168.3.1     <none>        443/TCP     94d
my-nginx      192.168.3.179   <nodes>       80/TCP      90d
nginx-kit     192.168.3.196   <nodes>       80/TCP      12d
rbd-rwat-api  192.168.3.22    <none>        8080/TCP    60d
```

Flannel network（看IP那列）：

```
# kubectl get pod -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
my-nginx-2395715568-gp1jv         1/1       Running   6          91d       172.16.99.3   {master node ip}
nginx-kit-3872865736-rc8hr        2/2       Running   0          12d       172.16.57.7   {minion node ip}
... ...
```

三、平坦的Flannel网络

1、Kubenetes安装后的网络状态

首先让我们来看看：kube-up.sh在安装k8s集群时对各个K8s Node都动了什么手脚！

a) 修改docker default配置

在ubuntu 14.04下，docker的配置都在/etc/default/docker文件中。如果你曾经修改过该文件，那么kube-up.sh脚本方式安装完kubernetes后，你会发现/etc/default/docker已经变样了，只剩下了一行：

```
master node:
DOCKER_OPTS=" -H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock --bip=172.16.99.1/24 --mtu=1450"

minion node:
DOCKER_OPTS=" -H tcp://127.0.0.1:4243 -H unix:///var/run/docker.sock --bip=172.16.57.1/24 --mtu=1450"
```

可以看出kube-up.sh修改了Docker daemon的-bip选项，使得该node上docker daemon在该node的flannel subnet范围以内为启动的Docker container分配IP地址。

b) 在etcd中初始化flannel网络数据

多个node上的Flannel依赖一个etcd cluster来做集中配置服务，etcd保证了所有node上flanned所看到的配置是一致的。同时每个node上的flanned监听etcd上的数据变化，实时感知集群中node的变化。

我们可以通过etcdctl查询到这些配置数据：

```
master node:

//flannel network配置
# etcdctl --endpoints http://127.0.0.1:{etcd listen port} get  /coreos.com/network/config
{"Network":"172.16.0.0/16", "Backend": {"Type": "vxlan"}}

# etcdctl --endpoints http://127.0.0.1:{etcd listen port} ls  /coreos.com/network/subnets
/coreos.com/network/subnets/172.16.99.0-24
/coreos.com/network/subnets/172.16.57.0-24

//某一node上的flanne subnet和vtep配置
# etcdctl --endpoints http://127.0.0.1:{etcd listen port} get  /coreos.com/network/subnets/172.16.99.0-24
{"PublicIP":"{master node ip}","BackendType":"vxlan","BackendData":{"VtepMAC":"b6:bf:4c:81:cf:3b"}}

minion node:
# etcdctl --endpoints http://127.0.0.1:{etcd listen port} get  /coreos.com/network/subnets/172.16.57.0-24
{"PublicIP":"{minion node ip}","BackendType":"vxlan","BackendData":{"VtepMAC":"d6:51:2e:80:5c:69"}}
```

或用etcd 提供的rest api：

```
# curl -L http://127.0.0.1:{etcd listen port}/v2/keys/coreos.com/network/config
{"action":"get","node":{"key":"/coreos.com/network/config","value":"{\"Network\":\"172.16.0.0/16\", \"Backend\": {\"Type\": \"vxlan\"}}","modifiedIndex":5,"createdIndex":5}}
```

c) 启动flanneld

kube-up.sh在每个Kubernetes node上启动了一个flanneld的程序：

```
# ps -ef|grep flanneld

master node:
root      1151     1  0  2016 ?        00:02:34 /opt/bin/flanneld --etcd-endpoints=http://127.0.0.1:{etcd listen port} --ip-masq --iface={master node ip}

minion node:
root     11940     1  0  2016 ?        00:07:05 /opt/bin/flanneld --etcd-endpoints=http://{master node ip}:{etcd listen port} --ip-masq --iface={minion node ip}
```

一旦flanneld启动，它将从etcd中读取配置，并请求获取一个subnet lease(租约)，有效期目前是24hrs，并且监视etcd的数据更新。flanneld一旦获取subnet租约、配置完backend，它会将一些信息写入/run/flannel/subnet.env文件。

```
master node :
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=172.16.0.0/16
FLANNEL_SUBNET=172.16.99.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=true

minion node:
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=172.16.0.0/16
FLANNEL_SUBNET=172.16.57.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=true
```

当然flanneld的最大意义在于根据etcd中存储的全cluster的subnet信息，跨node传输flannel network中的数据包，这个后面会详细说明。

d) 创建flannel.1 网络设备、更新路由信息

各个node上的网络设备列表新增一个名为flannel.1的类型为vxlan的网络设备：

```
master node:

# ip -d link show
4: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/ether b6:bf:4c:81:cf:3b brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 1 local {master node local ip} dev eth0 port 0 0 nolearning ageing 300

minion node:

349: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/ether d6:51:2e:80:5c:69 brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 1 local  {minion node local ip} dev eth0 port 0 0 nolearning ageing 300
```

从flannel.1的设备信息来看，它似乎与eth0存在着某种bind关系。这是在其他bridge、veth设备描述信息中所没有的。

flannel.1设备的ip：

```
master node:

flannel.1 Link encap:Ethernet  HWaddr b6:bf:4c:81:cf:3b
          inet addr:172.16.99.0  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:5993274 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5829044 errors:0 dropped:292 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1689890445 (1.6 GB)  TX bytes:1144725704 (1.1 GB)

minion node:

flannel.1 Link encap:Ethernet  HWaddr d6:51:2e:80:5c:69
          inet addr:172.16.57.0  Bcast:0.0.0.0  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:6294640 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5755599 errors:0 dropped:25 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:989362527 (989.3 MB)  TX bytes:1861492847 (1.8 GB)
```

可以看到两个node上的flannel.1的ip与k8s cluster为两个node上分配subnet的ip范围是对应的。

下面是两个node上的当前路由表：

```
master node:
```

```
# ip route
... ...
172.16.0.0/16 dev flannel.1  proto kernel  scope link  src 172.16.99.0
172.16.99.0/24 dev docker0  proto kernel  scope link  src 172.16.99.1
... ...

minion node:

# ip route
... ...
172.16.0.0/16 dev flannel.1
172.16.57.0/24 dev docker0  proto kernel  scope link  src 172.16.57.1
... ...
```

以上信息将为后续数据包传输分析打下基础。

c) 平坦的flannel network

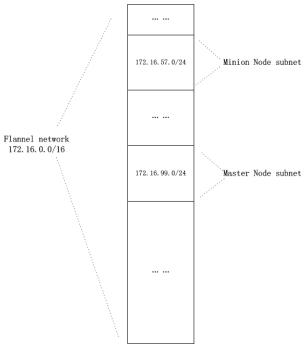
从以上kubernetes和flannel network安装之后获得的网络信息，我们能看出flannel network是一个flat network。在flannel：172.16.0.0/16这个大网下，每个kubernetes node从中分配一个子网片段(/24)：

```
master node :
   --bip=172.16.99.1/24

minion node :
   --bip=172.16.57.1/24

root@node1:~# etcdctl --endpoints http://127.0.0.1:{etcd listen port} ls  /coreos.com/network/subnets
/coreos.com/network/subnets/172.16.99.0-24
/coreos.com/network/subnets/172.16.57.0-24
```

用一张图来诠释可能更为直观：

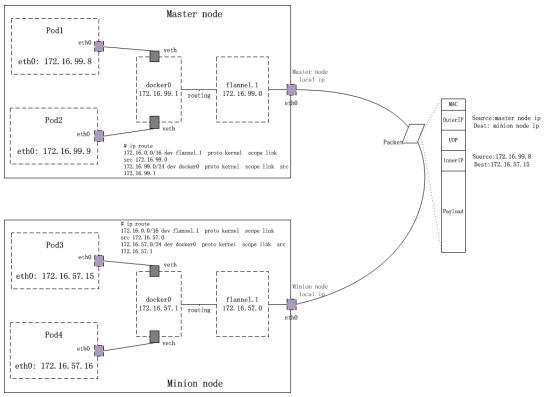


这个是不是有些像x86-64的虚拟内存寻址空间啊（同样是平坦内存地址访问模型）！

在平坦的flannel network中，每个pod都会被分配唯一的ip地址，且每个k8s node的subnet各不重叠，没有交集。不过这样的subnet分配模型也有一定弊端，那就是可能存在ip浪费：一个node上有200多个flannel ip地址(xxx.xxx.xxx.xxx/24)，如果仅仅启动了几个Pod，那么其余ip就处于闲置状态。

2、Flannel网络通信原理

这里我们模仿flannel官方的那幅原理图，画了一幅与我们的实验环境匹配的图，作为后续讨论flannel网络通信流程的基础：



如上图所示，我们来看看从pod1：172.16.99.8发出的数据包是如何到达pod3：172.16.57.15的（比如：在pod1的某个container中ping -c 3 172.16.57.15）。

a) 从Pod出发

由于k8s更改了docker的DOCKER_OPTS，显式指定了–bip，这个值与分配给该node上的subnet的范围是一致的。这样一来，docker引擎每次创建一个Docker container，该container被分配到的ip都在flannel subnet范围内。

当我们在Pod1下的某个容器内执行ping -c 3 172.16.57.15，数据包便开始了它在flannel network中的旅程。

Pod是Kubernetes调度的基本unit。Pod内的多个containers共享一个network namespace。kubernetes在创建Pod时，首先先创建pause容器，然后再以pause的network namespace为基础，创建pod内的其他容器（-net=container:xxx），这样Pod内的所有容器便共享一个network namespace，这些容器间的访问直接通过localhost即可。比如Pod下A容器启动了一个服务，监听8080端口，那么同一个Pod下面的另外一个B容器通过访问localhost:8080即可访问到A容器下面的那个服务。

在之前的《理解Docker容器网络之Linux Network Namespace》一文中，我相信我已经讲清楚了单机下Docker容器数据传输的路径。在这个环节中，数据包的传输路径也并无不同。

我们看一下Pod1中某Container内的路由信息：

```
# docker exec ba75f81455c7 ip route
default via 172.16.99.1 dev eth0
172.16.99.0/24 dev eth0  proto kernel  scope link  src 172.16.99.8
```

目的地址172.16.57.15并不在直连网络中，因此数据包通过default路由出去。default路由的路由器地址是172.16.99.1，也就是上面的docker0 bridge的IP地址。相当于docker0 bridge以“三层的工作模式”直接接收到来自容器的数据包(而并非从bridge的二层端口接收)。

b) docker0与flannel.1之间的包转发

数据包到达docker0后，docker0的内核栈处理程序发现这个数据包的目的地址是172.16.57.15，并不是真的要送给自己，于是开始为该数据包找下一hop。根据master node上的路由表：

```
master node :

# ip route
... ...
172.16.0.0/16 dev flannel.1  proto kernel  scope link  src 172.16.99.0
172.16.99.0/24 dev docker0  proto kernel  scope link  src 172.16.99.1
... ...
```

我们匹配到“172.16.0.0/16”这条路由！这是一条直连路由，数据包被直接送到flannel.1设备上。

c) flannel.1设备以及flanneld的功用

flannel.1是否会重复docker0的套路呢：包不是发给自己，转发数据包？会，也不会。

“会”是指flannel.1肯定要将包转发出去，因为毕竟包不是给自己的（包目的ip是172.16.57.15, vxlan设备ip是172.16.99.0）。
“不会”是指flannel.1不会走寻常套路去转发包，因为它是一个vxlan类型的设备，也称为vtep，virtual tunnel end point。

那么它到底是怎么处理数据包的呢？这里涉及一些Linux内核对vxlan处理的内容，详细内容可参见本文末尾的参考资料。

flannel.1收到数据包后，由于自己不是目的地，也要尝试将数据包重新发送出去。数据包沿着网络协议栈向下流动，在二层时需要封二层以太包，填写目的mac地址，这时一般应该发出arp：“who is 172.16.57.15”。但vxlan设备的特殊性就在于它并没有真正在二层发出这个arp包，因为下面的这个内核参数设置：

```
master node:

# cat /proc/sys/net/ipv4/neigh/flannel.1/app_solicit
3
```

而是由linux kernel引发一个“L3 MISS”事件并将arp请求发到用户空间的flanned程序。

flanned程序收到“L3 MISS”内核事件以及arp请求(who is 172.16.57.15)后，并不会向外网发送arp request，而是尝试从etcd查找该地址匹配的子网的vtep信息。在前面章节我们曾经展示过etcd中Flannel network的配置信息：

```
master node:

# etcdctl --endpoints http://127.0.0.1:{etcd listen port} ls  /coreos.com/network/subnets
/coreos.com/network/subnets/172.16.99.0-24
/coreos.com/network/subnets/172.16.57.0-24

# curl -L http://127.0.0.1:{etcd listen port}/v2/keys/coreos.com/network/subnets/172.16.57.0-24
{"action":"get","node":{"key":"/coreos.com/network/subnets/172.16.57.0-24","value":"{\"PublicIP\":\"{minion node local ip}\",\"BackendType\":\"vxlan\",\"BackendData\":{\"VtepMAC\":\"d6:51:2e:80:5c:69\"}}","expiration":"2017-01-17T09:46:20.607339725Z","ttl":21496,"modifiedIndex":2275460,"createdIndex":2275460}}
```

flanneld从etcd中找到了答案：

```
subnet: 172.16.57.0/24
public ip: {minion node local ip}
VtepMAC: d6:51:2e:80:5c:69
```

我们查看minion node上的信息，发现minion node上的flannel.1 设备mac就是d6:51:2e:80:5c:69：

```
minion node:

#ip -d link show

349: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/ether d6:51:2e:80:5c:69 brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 1 local 10.46.181.146 dev eth0 port 0 0 nolearning ageing 300
```

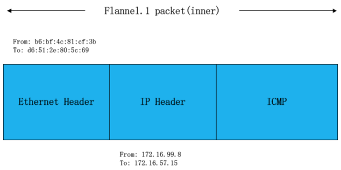
接下来，flanned将查询到的信息放入master node host的arp cache表中：

```
master node:

#ip n |grep 172.16.57.15
172.16.57.15 dev flannel.1 lladdr d6:51:2e:80:5c:69 REACHABLE
```

flanneld完成这项工作后，linux kernel就可以在arp table中找到 172.16.57.15对应的mac地址并封装二层以太包了。

到目前为止，已经呈现在大家眼前的封包如下图：



不过这个封包还不能在物理网络上传输，因为它实际上只是vxlan tunnel上的packet。

d) kernel的vxlan封包

我们需要将上述的packet从master node传输到minion node，需要将上述packet再次封包。这个任务在backend为vxlan的flannel network中由linux kernel来完成。

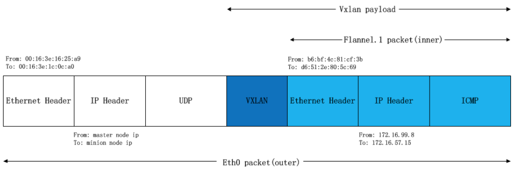
flannel.1为vxlan设备，linux kernel可以自动识别，并将上面的packet进行vxlan封包处理。在这个封包过程中，kernel需要知道该数据包究竟发到哪个node上去。kernel需要查看node上的fdb(forwarding database)以获得上面对端vtep设备（已经从arp table中查到其mac地址：d6:51:2e:80:5c:69）所在的node地址。如果fdb中没有这个信息，那么kernel会向用户空间的flanned程序发起"L2 MISS"事件。flanneld收到该事件后，会查询etcd，获取该vtep设备对应的node的"Public IP"，并将信息注册到fdb中。

这样Kernel就可以顺利查询到该信息并封包了：

```
master node:

# bridge fdb show dev flannel.1|grep d6:51:2e:80:5c:69
d6:51:2e:80:5c:69 dst {minion node local ip} self permanent
```

由于目标ip是minion node，查找路由表，包应该从master node的eth0发出，这样src ip和src mac地址也就确定了。封好的包示意图如下：



e) kernel的vxlan拆包

minion node上的eth0接收到上述vxlan包，kernel将识别出这是一个vxlan包，于是拆包后将flannel.1 packet转给minion node上的vtep（flannel.1）。minion node上的flannel.1再将这个数据包转到minion node上的docker0，继而由docker0传输到Pod3的某个容器里。

3、Pod内到外部网络

我们在Pod中除了可以与pod network中的其他pod通信外，还可以访问外部网络，比如：

```
master node:
# docker exec ba75f81455c7 ping -c 3 baidu.com
PING baidu.com (180.149.132.47): 56 data bytes
64 bytes from 180.149.132.47: icmp_seq=0 ttl=54 time=3.586 ms
64 bytes from 180.149.132.47: icmp_seq=1 ttl=54 time=3.752 ms
64 bytes from 180.149.132.47: icmp_seq=2 ttl=54 time=3.722 ms
--- baidu.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 3.586/3.687/3.752/0.072 ms
```

这个通信与vxlan就没有什么关系了，主要是通过docker引擎在iptables的POSTROUTING chain中设置的MASQUERADE规则：

```
mastre node:

#iptables -t nat -nL
... ...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.16.99.0/24       0.0.0.0/0
... ...
```

docker将容器的pod network地址伪装为node ip出去，包回来时再snat回容器的pod network地址，这样网络就通了。

四、"不真实"的Service网络

每当我们在k8s cluster中创建一个service，k8s cluster就会在–service-cluster-ip-range的范围内为service分配一个cluster-ip，比如本文开始时提到的：

```
# kubectl get services
NAME           CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
index-api      192.168.3.168   <none>        30080/TCP   18d
kubernetes     192.168.3.1     <none>        443/TCP     94d
my-nginx       192.168.3.179   <nodes>       80/TCP      90d
nginx-kit      192.168.3.196   <nodes>       80/TCP      12d
rbd-rest-api   192.168.3.22    <none>        8080/TCP    60d
```

这个cluster-ip只是一个虚拟的ip，并不真实绑定某个物理网络设备或虚拟网络设备，仅仅存在于iptables的规则中：

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
KUBE-SERVICES  all  --  0.0.0.0/0            0.0.0.0/0            /* kubernetes service portals */

# iptables -t nat -nL|grep 192.168.3
Chain KUBE-SERVICES (2 references)
target     prot opt source               destination
KUBE-SVC-XGLOHA7QRQ3V22RZ  tcp  --  0.0.0.0/0            192.168.3.182        /* kube-system/kubernetes-dashboard: cluster IP */ tcp dpt:80
KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  0.0.0.0/0            192.168.3.1          /* default/kubernetes:https cluster IP */ tcp dpt:443
KUBE-SVC-AU252PRZZQGOERSG  tcp  --  0.0.0.0/0            192.168.3.22         /* default/rbd-rest-api: cluster IP */ tcp dpt:8080
KUBE-SVC-TCOU7JCQXEZGVUNU  udp  --  0.0.0.0/0            192.168.3.10         /* kube-system/kube-dns:dns cluster IP */ udp dpt:53
KUBE-SVC-BEPXDJBUHFCSYIC3  tcp  --  0.0.0.0/0            192.168.3.179        /* default/my-nginx: cluster IP */ tcp dpt:80
KUBE-SVC-UQG6736T32JE3S7H  tcp  --  0.0.0.0/0            192.168.3.196        /* default/nginx-kit: cluster IP */ tcp dpt:80
KUBE-SVC-ERIFXISQEP7F7OF4  tcp  --  0.0.0.0/0            192.168.3.10         /* kube-system/kube-dns:dns-tcp cluster IP */ tcp dpt:53
... ...
```

可以看到在PREROUTING环节，k8s设置了一个target: KUBE-SERVICES。而KUBE-SERVICES下面又设置了许多target，一旦destination和dstport匹配，就会沿着chain进行处理。

比如：当我们在pod网络curl 192.168.3.22 8080时，匹配到下面的KUBE-SVC-AU252PRZZQGOERSG target：

```
KUBE-SVC-AU252PRZZQGOERSG  tcp  --  0.0.0.0/0            192.168.3.22         /* default/rbd-rest-api: cluster IP */ tcp dpt:8080
```

沿着target，我们看到"KUBE-SVC-AU252PRZZQGOERSG"对应的内容如下：

```
Chain KUBE-SVC-AU252PRZZQGOERSG (1 references)
target     prot opt source               destination
KUBE-SEP-I6L4LR53UYF7FORX  all  --  0.0.0.0/0            0.0.0.0/0            /* default/rbd-rest-api: */ statistic mode random probability 0.50000000000
KUBE-SEP-LBWOKUH4CUTN7XKH  all  --  0.0.0.0/0            0.0.0.0/0            /* default/rbd-rest-api: */

Chain KUBE-SEP-I6L4LR53UYF7FORX (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  172.16.99.6          0.0.0.0/0            /* default/rbd-rest-api: */
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/rbd-rest-api: */ tcp to:172.16.99.6:8080

Chain KUBE-SEP-LBWOKUH4CUTN7XKH (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  172.16.99.7          0.0.0.0/0            /* default/rbd-rest-api: */
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            /* default/rbd-rest-api: */ tcp to:172.16.99.7:8080

Chain KUBE-MARK-MASQ (17 references)
target     prot opt source               destination
MARK       all  --  0.0.0.0/0            0.0.0.0/0            MARK or 0x4000
```

请求被按5：5开的比例分发（起到负载均衡的作用）到KUBE-SEP-I6L4LR53UYF7FORX 和KUBE-SEP-LBWOKUH4CUTN7XKH，而这两个chain的处理方式都是一样的，那就是先做mark，然后做dnat，将service ip改为pod network中的Pod IP，进而请求被实际传输到某个service下面的pod中处理了。

五、参考资料

- How VXLAN works on Linux&VTEP implementation with Flannel
- Virtual switching technologies and Linux bridge
- How Flannel's VXLAN backend works 建议用google翻译将网页从日文翻译成英文再看^0^。
- Software Defined Networking using VXLAN

0

Related posts:

1. 使用Kubeadm安装Kubernetes-Part2
2. 使用Kubeadm安装Kubernetes
3. 一篇文章带你了解Kubernetes安装
4. 理解Docker跨多主机容器网络
5. 理解Docker容器端口映射

2条评论 最新 最早 最热

JackpGao
真的是好复杂。
2月28日 回复 顶 转发

王博
从头到尾，点到每个细节，写得真好!
3月8日 回复 顶 转发

这里是Tony Bai的个人Blog，欢迎访问、订阅和留言！订阅Feed请点击上面图片。

如果您觉得这里的文章对您有帮助，请扫描上方二维码进行捐赠，加油后的Tony Bai将会为您呈现更多精彩的文章，谢谢！

如果您喜欢通过微信App浏览本站内容，可以扫描下方二维码，订阅本站官方微信订阅号"iamtonybai"；点击二维码，可直达本人官方微博主页^_^：

本站Powered by Digital Ocean VPS。

选择Digital Ocean VPS主机，即可获得10美元现金充值，可免费使用两个月哟！

著名主机提供商Linode 10$优惠码：linode10，在这里注册即可免费获得。

阿里云推荐码：1WFZ0V，立享9折！

bigwhite.cn Getoil.com

my LinkedIn profile

文章

- 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
- 在Kubernetes Pod中使用Service Account访问API Server
- Kubernetes集群Pod使用Host的本地时区设置
- Kubernetes Pod无法挂载ceph RBD存储卷的临时解决方法
- Kubernetes集群中Service的滚动更新
- TensorFlow入门：带基础理立第一个神经网络
- Go 1.8中值得关注的几个变化
- 以Kubeadm方式安装的Kubernetes集群的探索
- Kubernetes Dashboard集成Heapster
- Kubernetes集群Dashboard插件安装

评论

- Hugs 在 Go 1.8中值得关注的几个变化
  多谢一贯的详细分享.
- 阿基 在 使用Kubeadm安装Kubernetes
  flannel网络，这个文件的内容/run/flannel/subnet.env， master节点...
- 王博 在 理解Kubernetes网络之Flannel网络
  从头到尾, 点到每个细节,写得真好!
- Eiidii0ni 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  其实我觉得，最起码kibana要能打开页面，能连接es获取数据是第二步...
- bigwhite 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  似乎还是网络问题，这个issue似乎和你的很想：https://discuss.elastic.co...
- Eiidii0ni 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  我重新跑了kibana的rc，起来还是这样....
- Eiidii0ni 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  你觉得还有什么可能性造成timeout
- bigwhite 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  看这里的es的日志也都很正常，暂没看出问题在哪。
- Eiidii0ni 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  而且我能访问到ES# curl -L http://localhost:8080/api/v1/pr...
- Eiidii0ni 在 使用Fluentd和ElasticSearch Stack实现Kubernetes的集群Logging
  # kubectl logs elasticsearch-logging-v1-24xbx --na...

下一页 »

分类

- 光影汇 (7)
- 影音坊 (36)
- 思考馆 (64)
- 技术点 (506)
- 教育记 (1)
- 杂货铺 (75)
- 生活簿 (153)
- 职场录 (14)
- 读书吧 (14)
- 运动迷 (76)
- 驴友秀 (40)

标签

Blog Blogger C Cms docker English GCC github GNU Go Golang Google Java Kernel Kubernetes Linux Mess Opensource Programmer Python Solaris Subversion Ubuntu Unix Windows 世界杯 博客 学习 旅游 工作 巴萨 开源 思考 感悟 摄影 旅游 美食 舞蹈 生活 体育 程序员 编译器 翻译 足球 驴友

归档

- 2017 年三月 (2)
- 2017 年二月 (5)
- 2017 年一月 (7)
- 2016 年十二月 (7)
- 2016 年十一月 (7)
- 2016 年十月 (3)
- 2016 年九月 (2)
- 2016 年八月 (1)
- 2016 年六月 (2)
- 2016 年五月 (2)
- 2016 年四月 (2)
- 2016 年三月 (2)
- 2016 年二月 (3)
- 2016 年一月 (2)
- 2015 年十二月 (1)
- 2015 年十一月 (1)
- 2015 年十月 (1)
- 2015 年九月 (3)
- 2015 年八月 (5)
- 2015 年七月 (6)
- 2015 年六月 (4)
- 2015 年五月 (1)
- 2015 年四月 (2)
- 2015 年三月 (2)
- 2015 年一月 (2)
- 2014 年十二月 (5)
- 2014 年十一月 (8)
- 2014 年十月 (9)
- 2014 年九月 (2)
- 2014 年八月 (1)
- 2014 年七月 (1)
- 2014 年五月 (2)
- 2014 年四月 (5)
- 2014 年三月 (4)
- 2014 年二月 (1)
- 2014 年一月 (1)
- 2013 年十二月 (3)
- 2013 年十一月 (5)
- 2013 年十月 (6)
- 2013 年九月 (4)
- 2013 年八月 (5)
- 2013 年七月 (6)
- 2013 年六月 (2)
- 2013 年五月 (6)
- 2013 年四月 (3)
- 2013 年三月 (7)
- 2013 年二月 (4)
- 2013 年一月 (6)
- 2012 年十二月 (8)
- 2012 年十一月 (10)
- 2012 年十月 (5)
- 2012 年九月 (3)
- 2012 年八月 (10)
- 2012 年七月 (4)
- 2012 年六月 (2)
- 2012 年五月 (4)
- 2012 年四月 (10)
- 2012 年三月 (8)
- 2012 年二月 (6)
- 2012 年一月 (6)
- 2011 年十二月 (4)
- 2011 年十一月 (4)
- 2011 年十月 (5)
- 2011 年九月 (8)

- 2011 年八月 (7)
- 2011 年七月 (6)
- 2011 年六月 (7)
- 2011 年五月 (8)
- 2011 年四月 (6)
- 2011 年三月 (10)
- 2011 年二月 (7)
- 2011 年一月 (10)
- 2010 年十二月 (7)
- 2010 年十一月 (6)
- 2010 年十月 (7)
- 2010 年九月 (12)
- 2010 年八月 (8)
- 2010 年七月 (3)
- 2010 年六月 (5)
- 2010 年五月 (4)
- 2010 年四月 (2)
- 2010 年三月 (6)
- 2010 年二月 (4)
- 2010 年一月 (6)
- 2009 年十二月 (6)
- 2009 年十一月 (6)
- 2009 年十月 (5)
- 2009 年九月 (8)
- 2009 年八月 (6)
- 2009 年七月 (7)
- 2009 年六月 (1)
- 2009 年五月 (3)
- 2009 年四月 (4)
- 2009 年三月 (6)
- 2009 年二月 (4)
- 2009 年一月 (8)
- 2008 年十二月 (9)
- 2008 年十一月 (5)
- 2008 年十月 (10)
- 2008 年九月 (9)
- 2008 年八月 (13)
- 2008 年七月 (3)
- 2008 年六月 (1)
- 2008 年五月 (7)
- 2008 年四月 (4)
- 2008 年三月 (9)
- 2008 年二月 (11)
- 2008 年一月 (15)
- 2007 年十二月 (11)
- 2007 年十一月 (14)
- 2007 年十月 (4)
- 2007 年九月 (5)
- 2007 年八月 (1)
- 2007 年七月 (10)
- 2007 年六月 (10)
- 2007 年五月 (10)
- 2007 年四月 (8)
- 2007 年三月 (15)
- 2007 年二月 (4)
- 2007 年一月 (17)
- 2006 年十二月 (18)
- 2006 年十一月 (9)
- 2006 年十月 (11)
- 2006 年九月 (6)
- 2006 年八月 (5)
- 2006 年七月 (22)
- 2006 年六月 (35)
- 2006 年五月 (24)
- 2006 年四月 (26)
- 2006 年三月 (25)
- 2006 年二月 (18)
- 2006 年一月 (15)
- 2005 年十二月 (10)
- 2005 年十一月 (10)
- 2005 年九月 (13)
- 2005 年八月 (11)
- 2005 年七月 (6)
- 2005 年六月 (2)
- 2005 年五月 (3)
- 2005 年四月 (6)
- 2005 年三月 (1)
- 2005 年一月 (15)
- 2004 年十二月 (9)
- 2004 年十一月 (14)
- 2004 年十月 (2)
- 2004 年九月 (2)

私人

- 我的女儿

链接

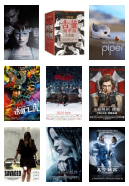
- @douban
- @flickr
- @github
- @googlecode
- @picasa
- @slideshare
- @twitter
- @weibo
- Hotgrran
- Lionel Messi
- Puras He
- 梦阳风暴
- 过眼云烟

开源项目

- bsildc
- chehave
- lcut

翻译项目

- C语言编码风格和标准
- 《Programming in Haskell》中文翻译项目



我的豆瓣主页