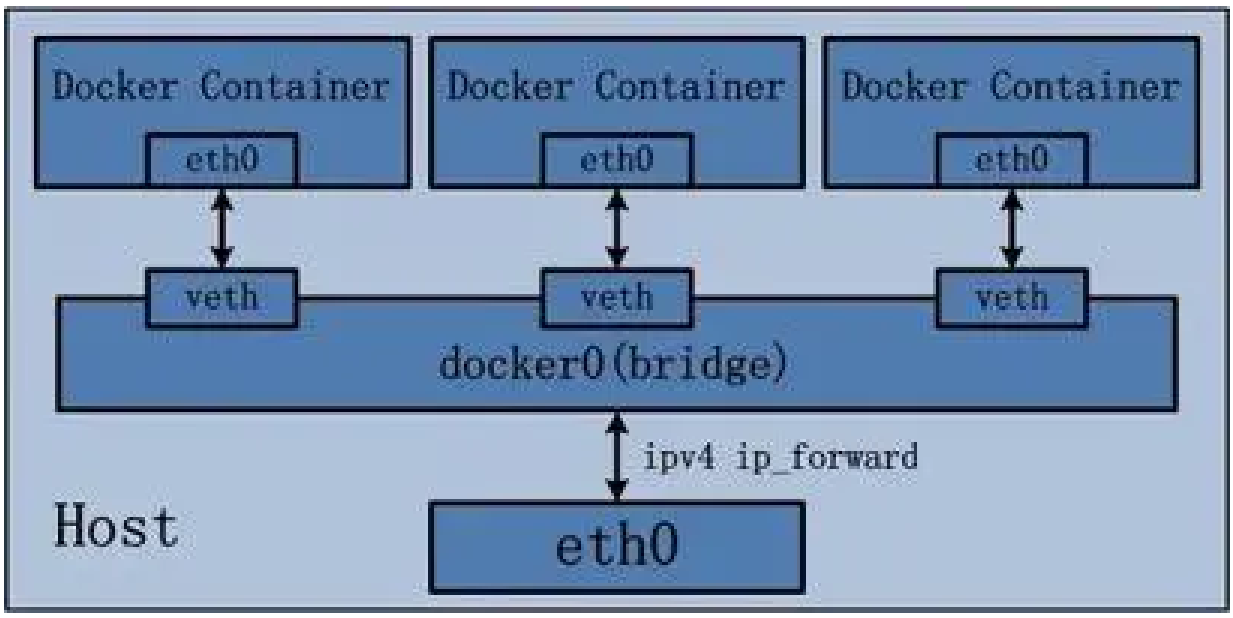
文章已收录我的仓库：Java学习笔记与免费书籍分享

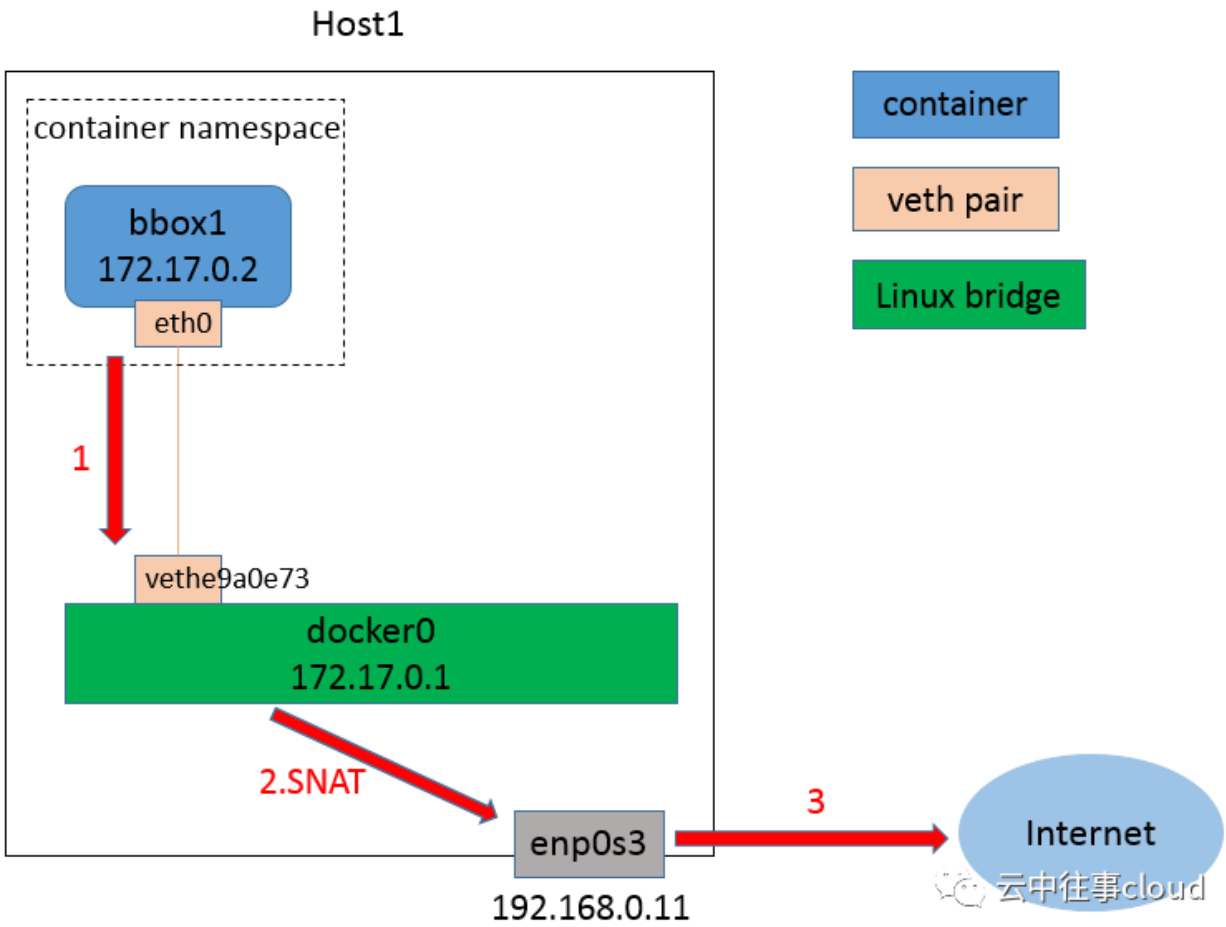
Docker网络模型

bridge模式

bridge模式是docker默认的网络模式，也是使用最频繁的一个模式，我们主要理解该模式。



容器向外发送信息



在docker启动时，如果不显式添加参数，则默认使用该模式，启动时docker会创建一个虚拟网卡，被命名为docker0，docker0一端管辖着docker容器网络，docker容器网络一般以内部地址172(被占用则改用其他内部地址)开头，一端连接宿主机物理网卡，起着桥接的作用，这也是该模式名称的由来。此后，每创建一个容器，docker就会创建一个**虚拟网络设备接口veth-pair**(上图中的eth0<->veth)，一个接口用于由docker0向容器发送消息，另一个接口用于由容器向外发送数据。

在docker0与宿主机物理接口中eth0，还存在一个ip forward(ip转发)规则，所有由docker0网络发往外部的数据，其源地址都会被映射为物理机的源地址，当数据包到达真实物理网卡时，就好像是由宿主机发出的数据包一样，然后，eth0物理网卡用自己的MAC地址将数据包封装成帧，进行链路传输，这就是整个发送的流程。

现在开始进行验证，注意docker需要运行在Linux机器上，windows下powershell部分命令无法执行。

在Linux宿主机中键入`ip addr`命令查看网口，存在如下信息：

```
docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
UP group default
    link/ether 02:42:8c:85:1a:f3 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:8cff:fe85:1af3/64 scope link
       valid_lft forever preferred_lft forever
```

发现确实存在docker0接口，其ip为`172.17.0.1`，并且采用CIDR方式进行路由，子网掩码为`255.255.0.0`。

创建一个容器bbox1**并进入**，键入`ifconfig`查看接口信息：

```
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:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2043 (1.9 KiB)  TX bytes:516 (516.0 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)
```

发现确实存在etho接口(lo是回环接口)，并且bbox1的ip地址为`172.17.0.2`，键入`route -n`查看容器的路由转发规则：

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         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
```

发现容器的默认网关是`172.17.0.1`，即docker0，0.0.0.0这个地址代表默认地址，即如果在路由表中不存在地址，则默认其为0.0.0.0，docker0收到数据包后，就像普通数据包一样试图发往物理网卡，在Linux中，数据包发往物理网卡转发之前会先交由内核路由，内核路由会判断是否需要进行NAT转换，经过相应处理之后再进行网卡路由选择转发。

> 此处内核充当NAT路由，为软件路由，为网卡前的一层抽象，对主机内进程是透明的

回到宿主机中(另外开一个终端)，在宿主机命令行键入`iptables -t nat -vnL`命令查看NAT转换规则，发现有如下信息：

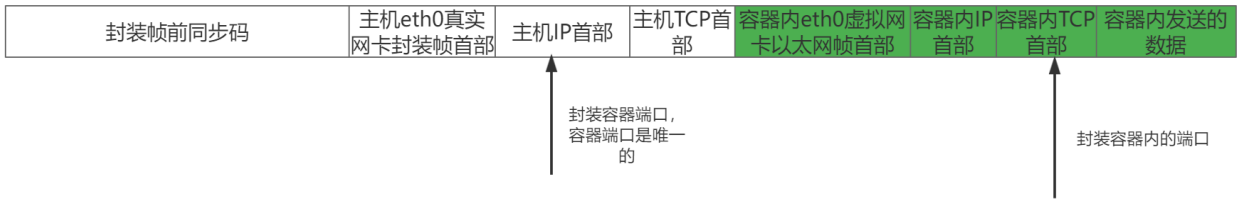
```
Chain POSTROUTING (policy ACCEPT 21 packets, 1479 bytes)
 pkts bytes target     prot opt in   out       source
destination
    3   202 MASQUERADE  all  --  *   !docker0  172.17.0.0/16
0.0.0.0/0
```

发现由172.17.0.0/16发往0.0.0.0/0的包会被内核应用MASQUERADE规则，MASQUERADE规则会将该ip分组的源地址伪装成真实的物理机地址，这样一个虚拟的不存在的容器地址就变成了真实的、存在的物理机地址，因此可以与外界进行通信。

> 172.17.0.0/16 代表 所有 源地址 & 255.255.0.0(16个1) = 172.17.0.0 的地址，这里就是容器内网地址
>
> 同理，0.0.0.0/0 意味着任意地址

之前有个学弟问我，为什么其他服务器能回应由容器内发出的ping，当数据包发送到物理机上，又是怎么到达容器内的呢？

这里涉及到docker的二层封装知识，当docker0收到容器内发来的数据包时，会将收到的数据包作为一个整体当作数据，重新进行TCP/IP封装，其中TCP端口号为容器端口号，要知道，容器不过是主机上的一个进程，其端口号是唯一的，这样当主机收到数据包时直接交由TCP层，TCP层根据端口号直接将数据交付容器，此时容器就好像刚刚收到一个完整的数据包一样。



外部向容器发送信息

要知道，外部主机只知道宿主机的ip，容器对其是完全不可见的，因此必须映射物理机的地址到容器上。

在上面的例子中，外部主机得知了容器的具体端口号，因此可以将数据交由容器，但这只是特殊的情况，外部机仍然无法访问具体的容器内的服务，例如：容器内的8080端口开放tomcat服务。

这种情况下，需要开启端口映射，例如将容器内的 8080 端口映射到主机的 80 端口，这样外部机访问主机的 80 端口即可访问容器的的 8080 端口，启动时可通过参数 `-p [主机端口][容器端口]` 开启次映射，当配置映射时，docker会自动配置Linux内核的SNAT规则与转发表，对所有目的地址为 `[主机IP][配置的主机端口]` 转换为 `[容器IP][配置的主机端口]`，并转发与docker0接口，docker0接口进行查表，交由具体的容器。

自定义网络规则

创建之前，查看当前网络，可以发现docker安装时创建了bridge、host、none三个网络。

```
[root@docker1 ~]# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
c1bb643c9c5a        bridge              bridge              local
59364623cee2        host                host                local
fb704391fb47        none                null                local
```

键入`docker network create --driver=bridge --subnet=172.08.21.0/24 --gateway=172.08.21.1 my-net`以bridge模式创建自定义网络，此时指定网段172.08.21.0/24，指定网关172.08.21.1，此时发现多了`my-net`网络：

```
PS C:\Users\happysnakers> docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
4f40a1f35b48   bridge    bridge    local
388c0fa8a67d   host      host      local
92462d1ae4e8   my-net    bridge    local
306db96b7739   none      null      local
```

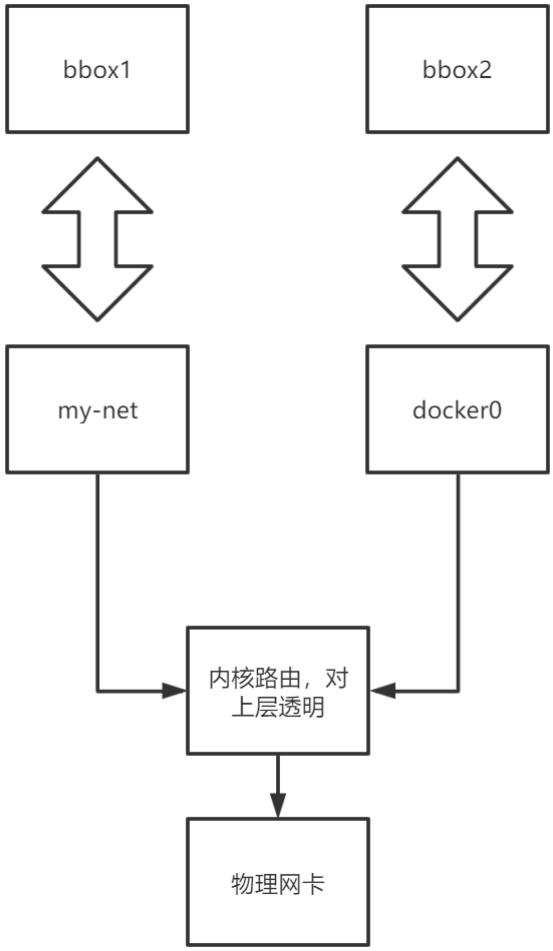
利用busybox镜像创建bbox1容器，`docker run -it --rm --network=my-net --name bbox1 --ip=172.08.21.65 busybox`，注意指定的ip必须与之前定义网络时指定的网段相符合。

新开终端，利用busybox镜像创建bbox2容器，`docker run -it --rm --name bbox2 busybox`，此处利用默认的docker0配置。

打印bbox1中的网络配置:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:08:15:41
          inet addr:172.8.21.65  Bcast:172.8.21.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1076 (1.0 KiB)  TX bytes:378 (378.0 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)
```

ip 确实为我们指定的ip地址，在bbox2中尝试`ping 172.8.21.65`，发现ping不通。



此时网络图如上，看来一切原因都在内核路由中，回到主机中，键入`route -n`查看主机的路由:

```
Kernel IP routing table
Destination     Gateway         Genmask             Iface
0.0.0.0         192.168.0.1     0.0.0.0             enp0s3
172.17.0.0      0.0.0.0         255.255.0.0         docker0
172.8.21.0      0.0.0.0         255.255.255.0       br-757bb8fb1878
192.168.0.0     0.0.0.0         255.255.255.0       enp0s3
```

从路由中看出各网段是未封闭的，应当是可以通信的，但我们之前说过，路由转发前会动过SNAT转换，问题应该是出现在NAT转换过程中了，键入`iptables -t filter -vnL`命令查看过滤表：

```
Chain DOCKER-ISOLATION-STAGE-1 (1 references)
 pkts bytes target     prot opt in     out     source
destination
    0     0 DOCKER-ISOLATION-STAGE-2  all  --  br-757bb8fb1878 !br-
757bb8fb1878  0.0.0.0/0            0.0.0.0/0
   33  6167 DOCKER-ISOLATION-STAGE-2  all  --  docker0 !docker0
0.0.0.0/0            0.0.0.0/0
   76  9290 RETURN     all  --  *      *       0.0.0.0/0
0.0.0.0/0
```

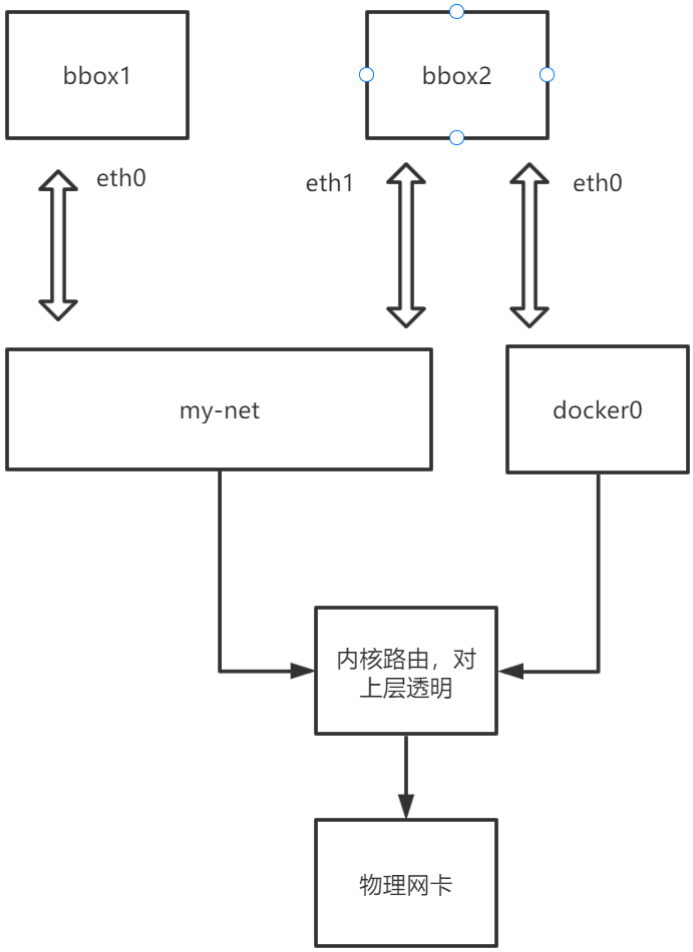
可以看到向docker0和br-757bb8fb1878(my-net)接口的发出任意包都会被应用DOCKER-ISOLATION-STAGE-2规则，事实上在DOCKER-ISOLATION-STAGE-2之前还有一个DOCKER-ISOLATION-STAGE-1规则，DOCKER-ISOLATION-STAGE-1规则过滤任何从docker容器内发出的包，将该包交由DOCKER-ISOLATION-STAGE-2链处理，DOCKER-ISOLATION-STAGE-2过滤任意发往docker容器的包，而对不发往docker容器的包返回给正常链处理(RETURN规则)，这样经DOCKER-ISOLATION-STAGE-1和DOCKER-ISOLATION-STAGE-2过滤后的包即是由不同brigde中的容器之间的数据包了。

执行`docker network connect my-net bbox2`命令将bbox2容器连接到`my-net`网络(bbox1所属网络)，在bbox2中尝试`ping 172.8.21.65`，发现能够ping通，网络确实互联。

键入`route -n`和`ifconfig`查看bbox2的路由转发规则与接口：

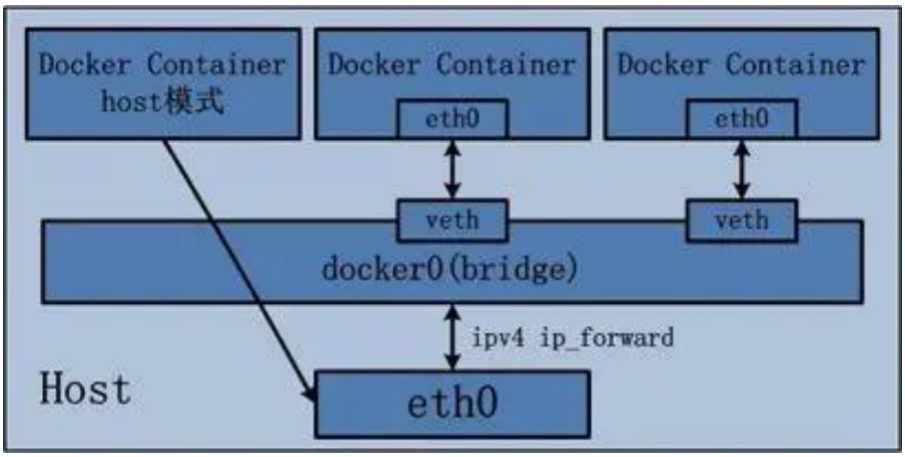
```
/ # route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
0.0.0.0         172.17.0.1      0.0.0.0         UG    0      0        0
eth0
172.8.21.0      0.0.0.0         255.255.255.0   U     0      0        0
eth1
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0
eth0

/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:172.17.255.255  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:866 (866.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:08:15:02
          inet addr:172.8.21.2  Bcast:172.8.21.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1216 (1.1 KiB)  TX bytes:378 (378.0 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)
```

发现发往172.8.21.0/24的数据包会被发往eth1接口，并且发现eth1接口的目的地址inet addr确实为我们所配置网络的网关，成功互联，此时网络图大概如下所示：



host模式

host模式即主机模式，这时候一个容器就是正常的主机的一个进程，容器网络就是主机网络，容器和宿主机共享同一个网络命名空间，换言之，容器的IP地址即为宿主机的IP地址。所以容器可以和宿主机一样，使用宿主机的任意网卡，实现和外界的通信。其网络模型可以参照下图：



采用这种模式容器的确是可以访问外部主机，原理就是主机上的一个进程访问外网，这种模式缺点是很明显的，容器内不再虚拟化端口，容器内所有的端口会会占用主机端口，各

个容器之间会竞争端口，十分不安全。

none模式

在这种模式下，容器有独立的网络栈，但不包含任何网络配置，只具有lo这个loopback网卡用于进程通信。也就是说，none模式为容器做了最少的网络设置，但是俗话说得好“少即是多”，在没有网络配置的情况下，通过第三方工具或者手工的方式，开发这任意定制容器的网络，提供了最高的灵活性。