docker network

Docker是目前最火的轻量级容器技术，但是Docker在网络方面比较薄弱，在原生态的Docker世界中，Docker网络不具备跨宿主机的能力，但是在云计算领域，大多数系统都是采用分布式技术来设计和实现，因此网络问题的解决势在必行。在此环境下，很多企业都开发了各自的产品来完善Docker的网络:

* CoreOS团队针对Kubernetes设计的网络overlay工具Flannel
* Docker官方设计的SDN网络解决方案Pipework及SocketPlane
* Zett.io公司开发的通过虚拟网络连接跨宿主机容器的工具Weave

还有其他的网络方案。

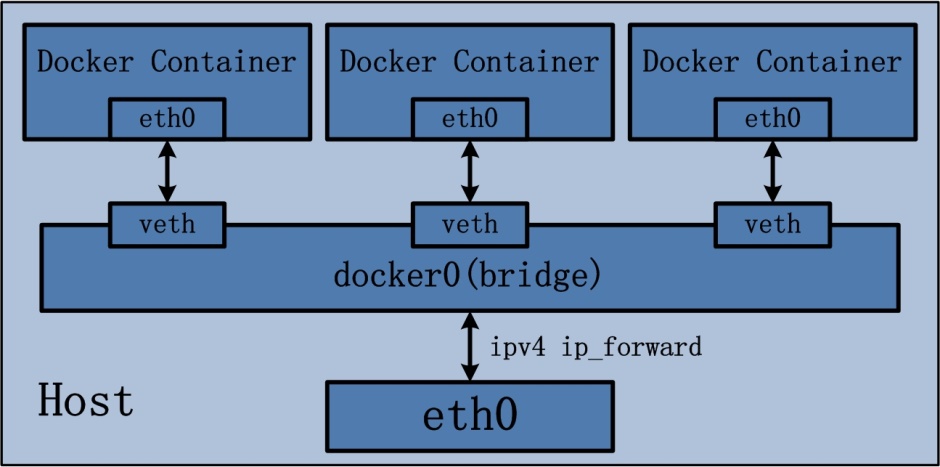
# 1.Docker自带网络模式

Docker有4种不同的网络模式：

* bridge桥接模式，使用—net-bridge指定，默认设置
* host模式，使用—net=host指定
* container模式，使用—net=containe:NAME\_or\_ID指定
* none模式，使用—net=none指定

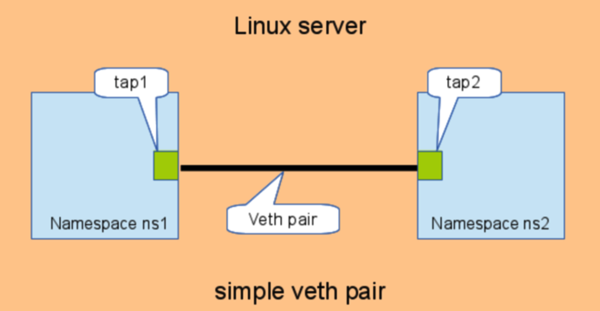
## 1.1 bridge模式

Docker Container的bridge（桥接）模式是目前Docker开发者最常用的网络模式。Bridge桥接模式为Docker Container创建独立的网络栈，保证容器内的进程组使用独立的网络环境，实现容器间、容器与宿主机之间的网络栈隔离。另外，Docker通过宿主机上的网桥（docker0）来连通容器内部的网络栈与宿主机的网络栈，实现容器与宿主机乃至外界的网络通信，bridge桥接模式示意图如下所示：



实现步骤如下：

1. Docker Daemon使用veth pair技术，在宿主机上创建两个虚拟机网络接口设备，假设为veth0和veth1。Veth Pair技术的特性是veth之间接受到报文之后都会传输给另一方。



1. Docker Daemon将veth0附加到Docker Daemon 创建的docker0网桥上，保证宿主机的网络报文可以发往veth0
2. Docker Daemon将veth1添加到Docker Container的namespace下改名为eth0，宿主机的网络报文若发往veth0，则立即被eth0接收，实现宿主机到Docker Container的网络连通性；同时保证Docker Container单独使用eth0，实现容器网络环境的隔离型。

Bridge桥接模式，从原理上实现了Docker Container到宿主机乃至其他机器的网络连通性，但是由于宿主机的IP地址和veth pair的IP地址均不在同一个网段，仅依靠veth pair和namespace技术不足以让宿主机以外的网络主动发现Docker Container存在。

为了使得Docker Container可以让宿主机以外的世界感知到容器内部暴露的服务，Docker采用NAT（Network Address Translation，网络地址转换）的方式，让宿主机以外的世界可以主动将网络报文发送至容器内部。当Docker Container需要暴露服务时，内部服务必须监听容器IP和端口号port\_0，以便外界主动发起访问请求。由于宿主机以外的世界，只知道宿主机eth0的网络地址，而并不知道Docker Container的IP地址，哪怕就算知道Docker Container的IP地址，从二层网络的角度来讲，外界也无法直接通过Docker Container的IP地址访问容器内部应用。

因此，Docker使用NAT方法，将容器内部的服务监听的端口与宿主机的某一个端口port\_1进行“绑定”。 外界访问Docker Container内部服务的流程为：

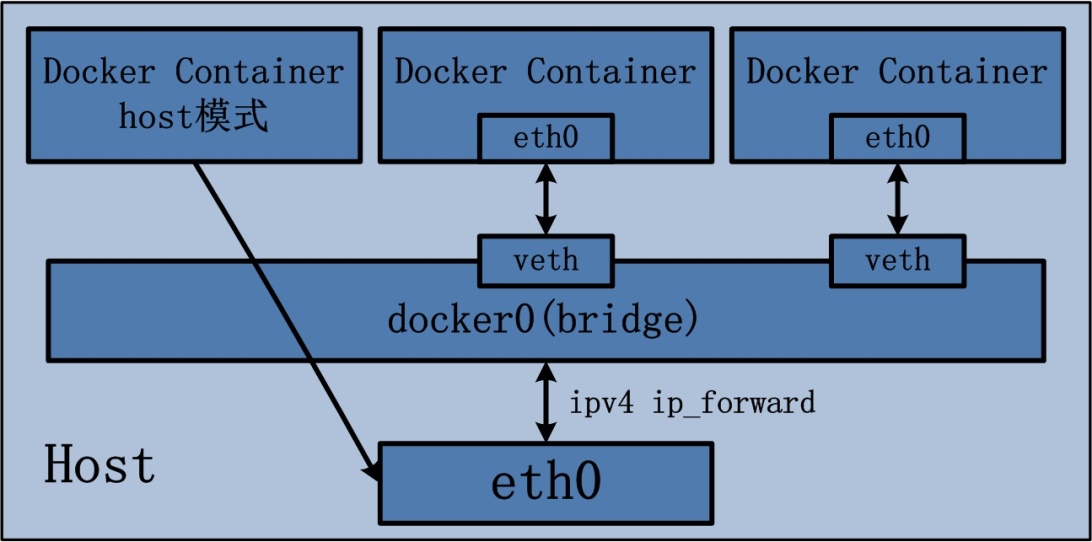
1. 外界访问宿主机的IP以及宿主机的端口port\_1；
2. 当宿主机接收到这样的请求之后，由于DNAT规则的存在，会将该请求的目的IP（宿主机eth0的IP）和目的端口port\_1进行转换，转换为容器IP和容器的端口port\_0;
3. 由于宿主机认识容器IP，故可以将请求发送给veth pair；
4. veth pair的veth0将请求发送至容器内部的eth0，最终交给内部服务进行处理。

使用DNAT方法，可以使得Docker宿主机以外的世界主动访问Docker Container内部服务。那么Docker Container如何访问宿主机以外的世界呢。以下简要分析Docker Container访问宿主机以外世界的流程：

1. Docker Container内部进程获悉宿主机以外服务的IP地址和端口port\_2，于是Docker Container发起请求。容器的独立网络环境保证了请求中报文的源IP地址为容器IP（即容器内部eth0），另外Linux内核会自动为进程分配一个可用源端口（假设为port\_3）;
2. 请求通过容器内部eth0发送至veth pair的另一端，到达veth0，也就是到达了网桥（docker0）处；
3. docker0网桥开启了数据报转发功能（/proc/sys/net/ipv4/ip\_forward），故将请求发送至宿主机的eth0处；、
4. 宿主机处理请求时，使用SNAT对请求进行源地址IP转换，即将请求中源地址IP（容器IP地址）转换为宿主机eth0的IP地址；
5. 宿主机将经过SNAT转换后的报文通过请求的目的IP地址（宿主机以外世界的IP地址）发送至外界。

## 1.2 host模式

Docker使用Linux Namespace隔离进程，其中Network Namespace隔离网络，在Bridge模式下一个Docker容器会分配一个独立的Network Namespace。但是使用host模式，容器和主机共用一个Network Namespace，容器不会虚拟机自己的网络、配置自己的ip，而是使用宿主机的ip和端口，如下图所示：



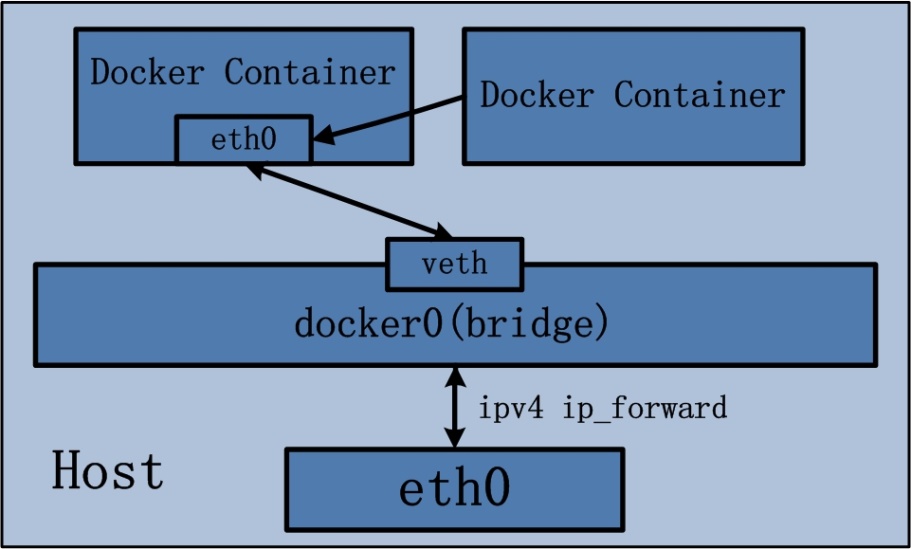
最左侧的Docker Container采用host网络模式，其使用宿主机的eth0实现与外界的通信。

Docker Container的host网络模式在实现过程中，由于不需要额外的网桥以及虚拟网卡，故不会涉及docker0以及veth pair。上文namespace的介绍中曾经提到，父进程在创建子进程时，如果不使用CLONE\_NEWNET这个参数标志，那么创建出的子进程会与父进程共享同一个网络namespace。Docker就是采用了这个简单的原理，在创建进程启动容器的过程中，没有传入CLONE\_NEWNET参数标志，实现Docker Container与宿主机共享同一个网络环境，即实现host网络模式。

可以说，Docker Container的网络模式中，host模式是bridge桥接模式很好的补充。采用host模式的Docker Container，可以直接使用宿主机的IP地址与外界进行通信，若宿主机的eth0是一个公有IP，那么容器也拥有这个公有IP。同时容器内服务的端口也可以使用宿主机的端口，无需额外进行NAT转换。当然，有这样的方便，肯定会损失部分其他的特性，最明显的是Docker Container网络环境隔离性的弱化，即容器不再拥有隔离、独立的网络栈。另外，使用host模式的Docker Container虽然可以让容器内部的服务和传统情况无差别、无改造的使用，但是由于网络隔离性的弱化，该容器会与宿主机共享竞争网络栈的使用；另外，容器内部将不再拥有所有的端口资源，原因是部分端口资源已经被宿主机本身的服务占用，还有部分端口已经用以bridge网络模式容器的端口映射

## 1.3 container模式

该模式是一种较为特别的网络模式，在这种模式下Docker Container会使用其他容器的网络环境，容器的网络隔离型处于bridge桥接模式和host模式之间。Docker Container共享其他容器的网络环境，这两个容器之间不存在网络隔离，但是与宿主机及其他容器存在网络隔离，网络模式如下图：



右侧的Docker Container采用other Container网络模式，使用的网络环境即为左侧Docker Container bridge桥接模式下的网络。

Docker Container的other container网络模式在实现过程中，不涉及网桥，同样也不需要创建虚拟网卡veth pair。完成other container网络模式的创建只需要两个步骤：

1. 查找other container（即需要被共享网络环境的容器）的网络namespace；
2. 将新创建的Docker Container（也是需要共享其他网络的容器）的namespace，使用other container的namespace。

Docker Container的other container网络模式，可以用来更好的服务于容器间的通信。在这种模式下的Docker Container可以通过localhost来访问namespace下的其他容器，传输效率较高。虽然多个容器共享网络环境，但是多个容器形成的整体依然与宿主机以及其他容器形成网络隔离。另外，这种模式还节约了一定数量的网络资源。但是需要注意的是，它并没有改善容器与宿主机以外世界通信的情况。

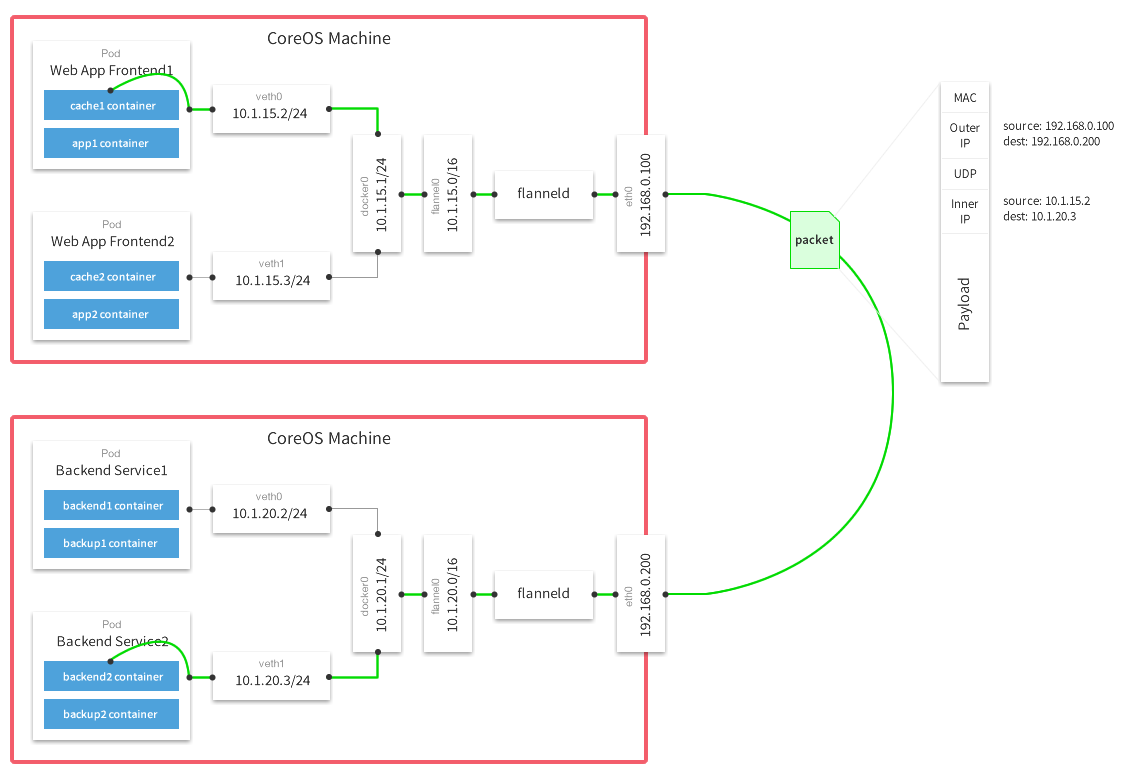
## 1.4 none模式

网络环境为none，不为Docker Container创建任何网络环境，采用这种模式，容器内部只能使用Loopback网络设备。

可以说none模式为Docker Container做了极少的网络设定，但是俗话说得好“少即是多”，在没有网络配置的情况下，作为Docker开发者，才能在这基础做其他无限多可能的网络定制开发。

# 2.flannel的使用及分析

Docker自身的网络功能比较简单，不能满足很多复杂的应用场景，例如跨主机的Container网络通信。因此，有很多开源项目用来改善Docker的网络功能，如[pipework](https://github.com/jpetazzo/pipework)、[weave](https://github.com/zettio/weave)、[flannel](https://github.com/coreos/flannel)等。Flannel是CoreOS团队针对Kubernetes设计的一个网络规划服务，简单来说，它的功能是让集群中的不同节点主机创建的Docker容器都具有全集群唯一的虚拟IP地址。Flannel的架构如下：



容器间的交互流程如下：

1. 容器直接使用目标容器的ip访问，默认通过容器内部的eth0发送出去。
2. 报文通过veth pair被发送到vethXXX。
3. vethXXX是直接连接到虚拟交换机docker0的，报文通过虚拟bridge docker0发送出去。
4. 查找路由表，外部容器ip的报文都会转发到flannel0虚拟网卡，这是一个P2P的虚拟网卡，然后报文就被转发到监听在另一端的flanneld。
5. flanneld通过etcd维护了各个节点之间的路由表，把原来的报文UDP封装一层，通过配置的iface发送出去。
6. 报文通过主机之间的网络找到目标主机。
7. 报文继续往上，到传输层，交给监听在8285端口的flanneld程序处理。
8. 数据被解包，然后发送给flannel0虚拟网卡。
9. 查找路由表，发现对应容器的报文要交给docker0。
10. docker0找到连到自己的容器，把报文发送过去。

## 2.1 Flannel的安装

1）安装etcd和flannel

*yum –y install etcd flannel*

2）配置/etc/etcd/etcd.conf，修改以下参数：

*ETCD\_LISTEN\_CLIENT\_URLS="*[*http://0.0.0.0:2379*](http://0.0.0.0:2379/)*"*

*ETCD\_ADVERTISE\_CLIENT\_URLS="*[*http://0.0.0.0:2379*](http://0.0.0.0:2379/)*"*

3）启动etcd

*systemctl start etcd*

4）在MASTER节点上修改etcd的配置，设定flannel所使用的子网范围

*etcdctl   mk /*[*atomic.io/network/config*](http://atomic.io/network/config)*'{"Network":"172.17.0.0/16",   "SubnetMin": "172.17.1.0", "SubnetMax":   "172.17.254.0"}'*

5）对所有的节点修改flannel配置etc/sysconfig/flanneld，设定etcd的相关信息，其中10.254.20.110为etcd的ip地址

*FLANNEL\_ETCD\_ENDPOINTS="*[*http://10.254.20.110:2379*](http://10.254.20.110:2379/)*"*

*FLANNEL\_ETCD="*[*http://10.254.20.110:2379*](http://10.254.20.110:2379/)*"*

6）启动flannel

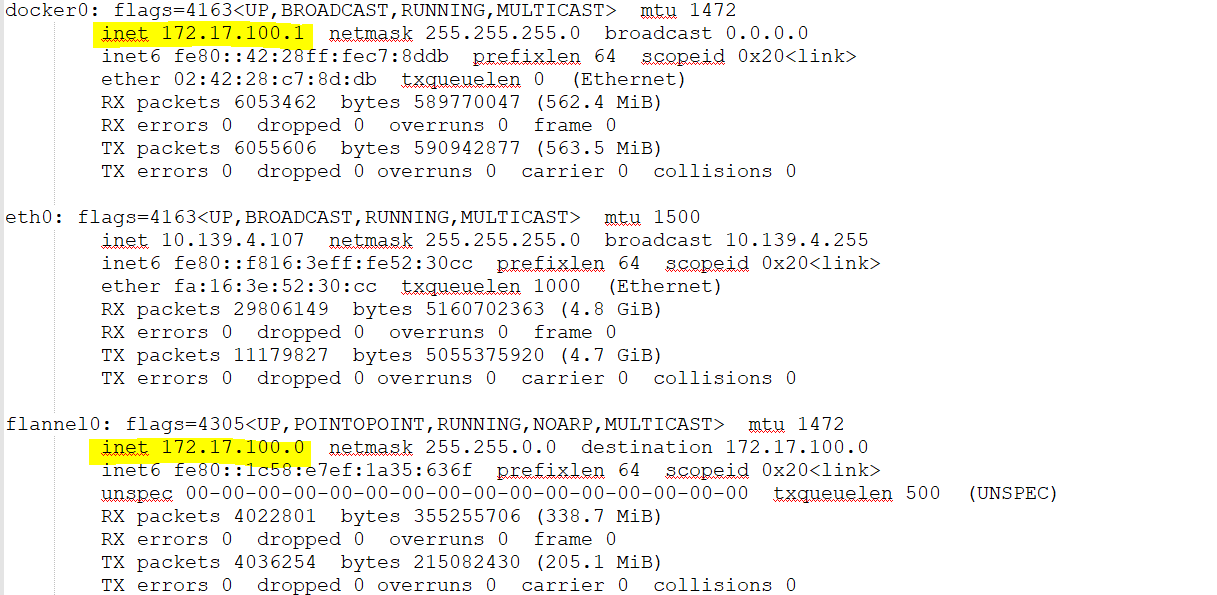
*systemctl start flanneld*

7）启动docker

*systemctl start docker*

## 2.2 确认信息及测试

启动etcd、flannel和docker后，查看ifconfig



docker0的ip和flannel0的子网范围一致。

在两个主机上启动docker container

*# docker run -d docker.io/ubuntu:14.04 /bin/sh -c "while true; do echo hello world; sleep 2; done"*

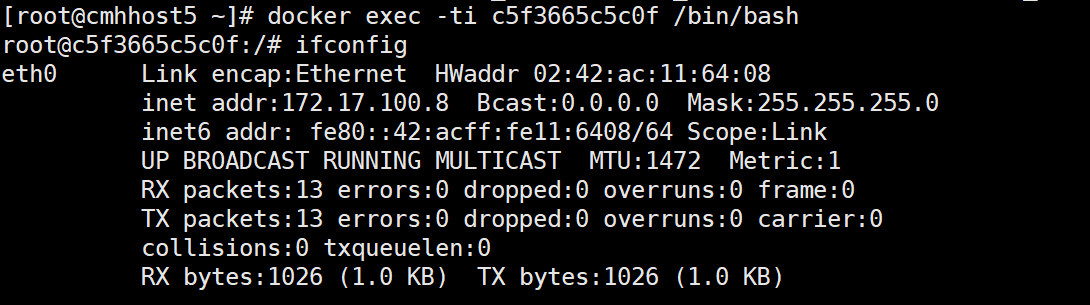
*c5f3665c5c0fb42c6fd68470c511de3a932b467900b4872f38ace41665a2bb72*

*[root@cmhhost5 ~]# docker ps*

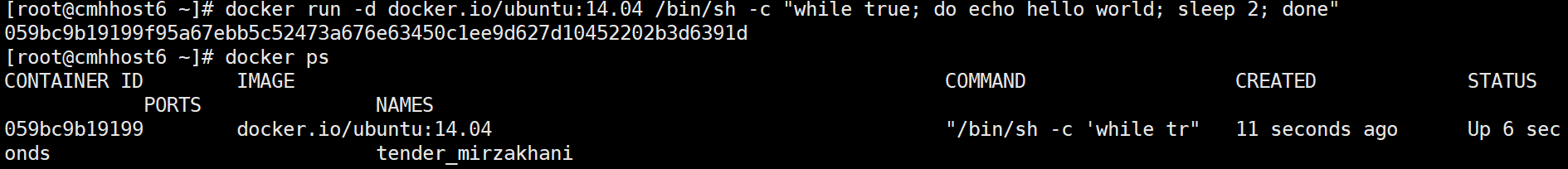
*CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES*

*c5f3665c5c0f docker.io/ubuntu:14.04 "/bin/sh -c 'while tr" 9 seconds ago Up 3 seconds adoring\_knuth*

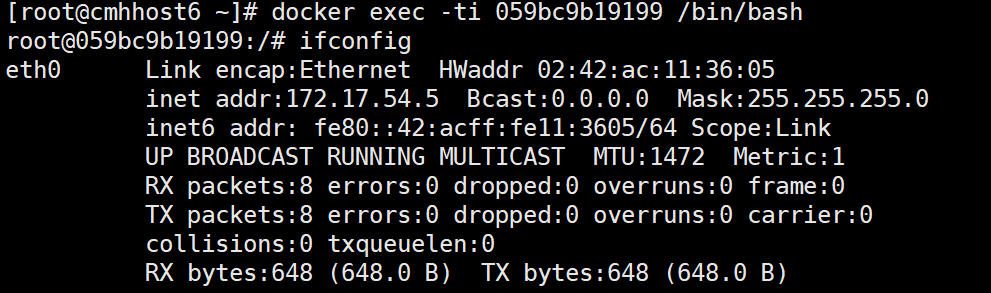
查看container的ip



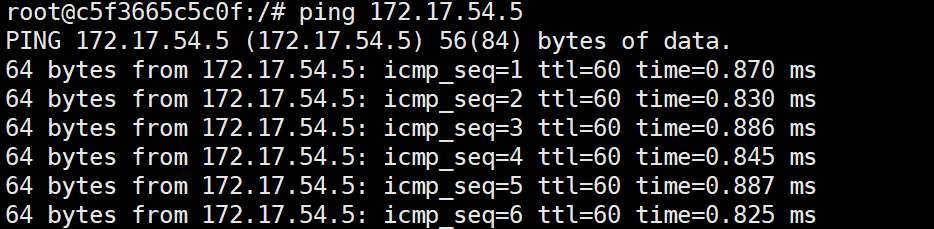
在cmhhost6.novalocal主机上启动container



查看container的ip:



在一个容器上ping另外一个宿主机的container，结果如下：



实现了跨主机的container通信。

# 参考文献：

http://blog.daocloud.io/docker-source-code-analysis-part7-first/

http://dockone.io/article/1489

https://github.com/coreos/flannel

https://www.hi-linux.com/posts/30481.html

http://dockone.io/article/618