# 解读Docker网络核心原理

在深入Docker内部的网络原理之前，我们先从一个用户的角度来直观感受一下Docker的网络架构和基本操作是怎么样的。

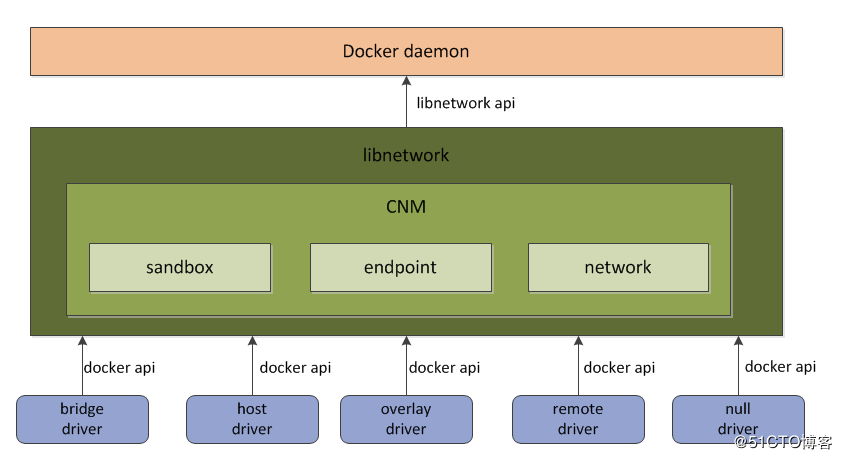
## Docker网络架构

 Docker在1.9版本中（现在都1.17了）引入了一整套docker network子命令和跨主机网络支持。这允许用户可以根据他们应用的拓扑结构创建虚拟网络并将容器接入其所对应的网络。

 其实，早在Docker1.7版本中，网络部分代码就已经被抽离并单独成为了Docker的网络库，即libnetwork。在此之后，容器的网络模式也被抽像变成了统一接口的驱动。

 为了标准化网络的驱动开发步骤和支持多种网络驱动，Docker公司在libnetwork中使用了CNM（Container Network Model）。CNM定义了构建容器虚拟化网络的模型。同时还提供了可以用于开发多种网络驱动的标准化接口和组件。

 libnetwork和Docker daemon及各个网络驱动的关系可以通过下面的图进行形象的表示。



  如上图所示，Docker daemon通过调用libnetwork对外提供的API完成网络的创建和管理等功能。libnetwrok中则使用了CNM来完成网络功能的提供。而CNM中主要有沙盒（sandbox）、端点（endpoint）、网络（network）这3种组件。

  libnetwork中内置的5种驱动则为libnetwork提供了不同类型的网络服务。接下来分别对CNM中的3个核心组件和libnetwork5种内置驱动进行介绍。

## CNM核心组件

**1、沙盒（sandbox）**  
 一个沙盒也包含了一个容器网络栈的信息。沙盒可以对容器的接口、路由和DNS设置等进行管理。沙盒的实现可以是Linux netwrok namespace、FreeBSD jail或者类似的机制。一个沙盒可以有多个端点和多个网络。

**2、端点（endpoint）**  
 一个端点可以加入一个沙盒和一个网络。端点的实现可以是veth pair、Open vSwitch内部端口或者相似的设备。一个端点只可以属于一个网络并且只属于一个沙盒。

**3、网络（network）**  
 一个网络是一组可以直接互相联通的端点。网络的实现可以是Linux bridge、VLAN等。一个网络可以包含多个端点

## libnetwork内置驱动

libnetwork共有5种内置驱动：bridge驱动、host驱动、overlay驱动、remote驱动、null驱动。

**1、bridge驱动**  
 此驱动为Docker的默认设置驱动，使用这个驱动的时候，libnetwork将创建出来的Docker容器连接到Docker网桥上。作为最常规的模式，bridge模式已经可以满足Docker容器最基本的使用需求了。然而其与外界通信使用NAT，增加了通信的复杂性，在复杂场景下使用会有诸多限制。

**2、host驱动**  
 使用这种驱动的时候，libnetwork将不为Docker容器创建网络协议栈，即不会创建独立的network namespace。Docker容器中的进程处于宿主机的网络环境中，相当于Docker容器和宿主机共同用一个network namespace，使用宿主机的网卡、IP和端口等信息。

 但是，容器其他方面，如文件系统、进程列表等还是和宿主机隔离的。host模式很好地解决了容器与外界通信的地址转换问题，可以直接使用宿主机的IP进行通信，不存在虚拟化网络带来的额外性能负担。但是host驱动也降低了容器与容器之间、容器与宿主机之间网络层面的隔离性，引起网络资源的竞争与冲突。  
 因此可以认为host驱动适用于对于容器集群规模不大的场景。

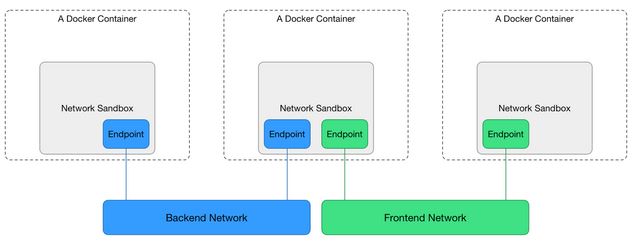
**3、overlay驱动**  
 此驱动采用IETE标准的VXLAN方式，并且是VXLAN中被普遍认为最适合大规模的云计算虚拟化环境的SDN controller模式。在使用过程中，需要一个额外的配置存储服务，例如Consul、etcd和zookeeper。还需要在启动Docker daemon的时候额外添加参数来指定所使用的配置存储服务地址。

**4、remote驱动**  
 这个驱动实际上并未做真正的网络服务实现，而是调用了用户自行实现的网络驱动插件，使libnetwork实现了驱动的可插件化，更好地满足了用户的多种需求。用户只需要根据libnetwork提供的协议标准，实现其所要求的各个接口并向Docker daemon进行注册。

**5、null驱动**  
 使用这种驱动的时候，Docker容器拥有自己的network namespace，但是并不为Docker容器进行任何网络配置。也就是说，这个Docker容器除了network namespace自带的loopback网卡名，没有其他任何网卡、IP、路由等信息，需要用户为Docker容器添加网卡、配置IP等。  
 这种模式如果不进行特定的配置是无法正常使用的，但是优点也非常明显，它给了用户最大的自由度来自定义容器的网络环境。

## libnetwork官方示例

 我们初步了解了libnetwork中各个组件和驱动后，为了能深入的理解libnetwork中的CNM模型和熟悉docker network子命令的使用，我们来通过libnetwork官方github上的示例进行验证一下，如下图所示：



 在上图示例中，使用Docker 默认的bridge驱动进行演示。在此例中，会在Docker上组成一个网络拓扑的应用：

* 它有两个网络，其中backend network为后端网络，frontend network则为前端网络，两个网络互不联通。(这两个网络呆会儿演示的时候会创建出来)
* 其中容器1和容器3各拥有一个端点，并且分别加入后端网络（backend network）和前端网络（frontend network）中。而容器2则有两个端点，它们分别加入到后端网络和前端网络。

1、通过以下命令分别创建名为backend、frontend两个网络：

**#** docker network create backend

**#** docker network create frontend

2、使用docker network ls 可以查看这台主机上的所有Docker网络：

*# docker network ls*

NETWORK ID NAME DRIVER SCOPE

879f8d1788ba backend bridge local

6c16e2b4122d bridge bridge local

d150ba23bdc0 frontend bridge local

1090f32081e8 host host local

7bf28f042f6c none null local

 除了刚才创建的backend和frontend之外，还有3个网络。这3个网络是Docker daemon默认创建的，分别使用了3种不同的驱动，而这3种驱动则对应了Docker原来的3种网络模式。需要注意的是，3种内置的默认网络是无法使用docker network rm进行删除的，不信你们试一下。

3、接下来创建3个容器，并使用下面的命令将名为c1和c2的容器加入到backend网络中，将名为c3的容器加入到frontend网络中：

**#** docker run -itd --name c1 --net backend busybox

**#** docker run -itd --name c2 --net backend busybox

**#** docker run -itd --name c3 --net frontend busybox

然后，分别进入c1和c3容器使用ping命令测试其与c2的连通性，因为c1和c2都在backend网络中，所以两者可以连通。但是，因为c3和c2不在一个网络中，所以两个容器之间不能连通：

*#进入c1容器ping c2通、ping c3不通。其它两个容器就不进入演示了，大家自己可以试一下：*

*# docker exec -it c1 sh*

*# ping c2*

PING c2 (172.19.0.3): 56 data bytes

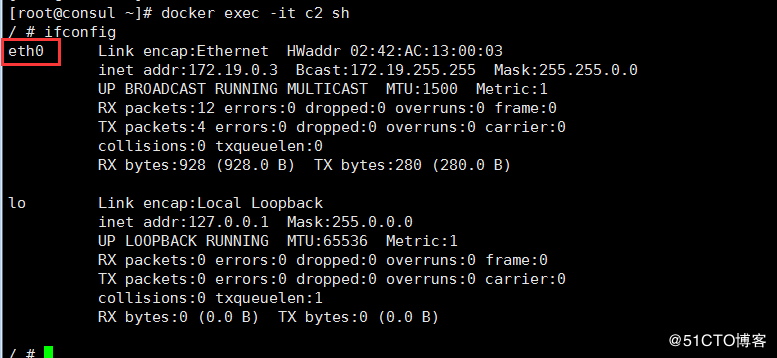
64 bytes from 172.19.0.3: seq=0 ttl=64 time=0.065 ms

64 bytes from 172.19.0.3: seq=1 ttl=64 time=0.050 ms

*# ping c3*

ping: bad address 'c3'

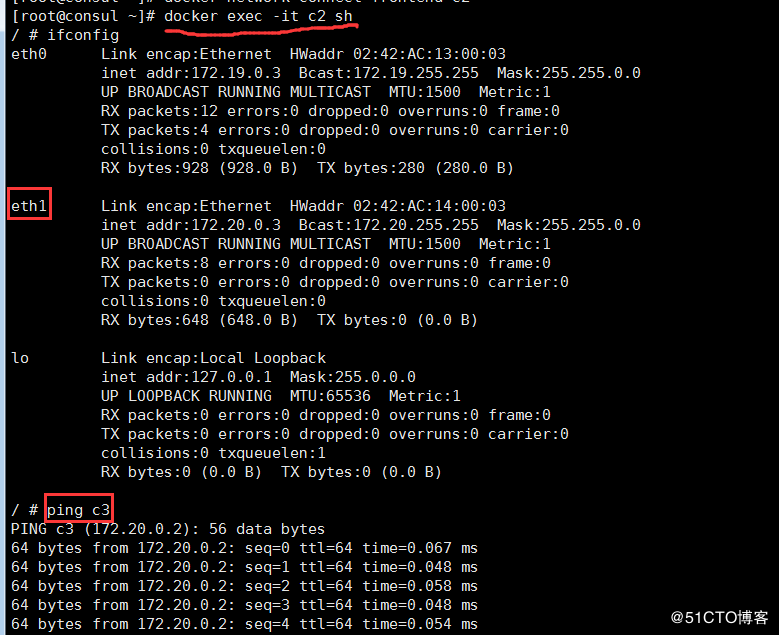
并且，可以进入c2容器中使用命令ifconfig来查看此容器中的网卡及配置情况。可以看到，容器中只有一块以太网卡，其名称为eth0，并且配置了和网桥backend同在一个IP段的IP地址，这个网卡就是CNM模型中的端点：



4、最后，使用如下命令将c2容器加入到frontend网络中：

**#** docker network connect frontend c2

再次，在c2容器中使用命令ifconfig来查看此容器中的网卡及配置情况。发现多了一块名为eth1的以太网卡，并且其IP和网桥frontend同在一个IP段。测试c2与c3的连通性后，可以发现两者已经连通。



可以看出，docker network connect命令会在所连接的容器中创建新的网卡，以完成其与所指定网络的连接。

## 分析bridge驱动实现机制

前面我们演示了bridge驱动下的CNM使用方式，下面来分析一下bridge驱动的实现机制又是怎样的。

* **docker0网桥**  
  当在一台未经过特殊网络配置的centos 或 ubuntu机器上安装完docker之后，在宿主机上通过ifconfig命令可以看到多了一块名为docker0的网卡，假设IP为 172.17.0.1/16。有了这样一块网卡，宿主机也会在内核路由表上添加一条到达相应网络的静态路由，可通过route -n查看：
* # **route** **-n**
* **Kernel** **IP** **routing** **table**
* **Destination** **Gateway** **Genmask** **Flags** **Metric** **Ref** **Use** **Iface**
* ...
* 172.17.0.0 0.0.0.0 255.255.0.0 **U** 0 0 0 **docker0**

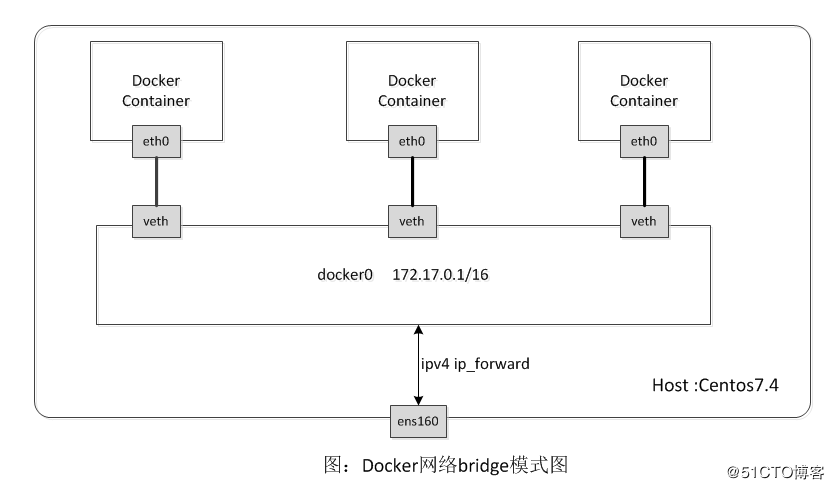
...

此条路由表示所有目的IP地址为172.17.0.0/16的数据包从docker0网卡转发。

 然后使用docker run命令创建一个执行shell(/bin/bash)的Docker容器，假设容器名称为con1。  
 在con1容器中可以看到它有两个网卡lo和eth0。lo设备不必多说，是容器的回环网卡；eth0即为容器与外界通信的网卡，eth0的ip 为 172.17.0.2/16，和宿主机上的网桥docker0在同一个网段。

 查看con1的路由表，可以发现con1的默认网关正是宿主机的docker0网卡，通过测试， con1可以顺利访问外网和宿主机网络，因此表明con1的eth0网卡与宿主机的docker0网卡是相互连通的。

 这时在来查看(ifconfig)宿主机的网络设备，会发现有一块以“veth”开头的网卡，如veth60b16bd，我们可以大胆猜测这块网卡肯定是veth设备了，而veth pair总是成对出现的。veth pair通常用来连接两个network namespace，那么另一个应该是Docker容器con1中的eth0了。之前已经判断con1容器的eth0和宿主机的docker0是相连的，那么veth60b16bd也应该是与docker0相连的，不难想到，docker0就不只是一个简单的网卡设备了，而是一个网桥。

 真实情况正是如此，下图即为Docker默认网络模式（bridge模式）下的网络环境拓扑图，创建了docker0网桥，并以eth pair连接各容器的网络，容器中的数据通过docker0网桥转发到eth0网卡上。  
  
 这里的网桥概念等同于交换机，为连在其上的设备转发数据帧。网桥上的veth网卡设备相当于交换机上的端口，可以将多个容器或虚拟机连接在上面，这些端口工作在二层，所以是不需要配置IP信息的。图中docker0网桥就为连在其上的容器转发数据帧，使得同一台宿主机上的Docker容器之间可以相互通信。  
 大家应该注意到docker0既然是二层设备，它上面怎么设置了IP呢？docker0是普通的linux网桥，它是可以在上面配置IP的，可以认为其内部有一个可以用于配置IP信息的网卡接口（如同每一个Open vSwitch网桥都有一个同名的内部接口一样）。在Docker的桥接网络模式中，docker0的IP地址作为连于之上的容器的默认网关地址存在。

 在Linux中，可以使用brctl命令查看和管理网桥（需要安装bridge-utils软件包），比如查看本机上的Linux网桥以及其上的端口：

*# brctl show*

bridge name bridge id STP enabled interfaces

docker0 8000.02420b69b449 no veth1b11267

更多关于brctl命令的功能和用法，大家通过man brctl或brctl --help查阅。

 docker0网桥是在Docker daemon启动时自动创建的，其IP默认为172.17.0.1/16，之后创建的Docker容器都会在docker0子网的范围内选取一个未占用的IP使用，并连接到docker0网桥上。

 除了使用docker0网桥外，还可以使用自己创建的网桥，比如创建一个名为br0的网桥，配置IP：

**#** brctl addbr br0

**#** ifconfig br0 18.18.0.1

* **iptables规则**  
   Docker安装完成后，将默认在宿主机系统上增加一些iptables规则，以用于Docker容器和容器之间以及和外界的通信，可以使用iptables-save命令查看。其中nat表中的POSTROUTING链有这么一条规则：

-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE

**参数说明：**  
-s ：源地址172.17.0.0/16  
-o：指定数据报文流出接口为docker0  
-j ：动作为MASQUERADE（地址伪装）

 上面这条规则关系着Docker容器和外界的通信，含义是：将源地址为172.17.0.0/16的数据包（即Docker容器发出的数据），当不是从docker0网卡发出时做SNAT。这样一来，从Docker容器访问外网的流量，在外部看来就是从宿主机上发出的，外部感觉不到Docker容器的存在。  
 那么，外界想到访问Docker容器的服务时该怎么办呢？我们启动一个简单的web服务容器，观察iptables规则有何变化。

1、首先启动一个 tomcat容器，将其8080端口映射到宿主机上的8080端口上：

**#docker** **run** **-itd** **--name** **tomcat01** **-p** 8080:8080 **tomcat**:latest

2、然后查看iptabels规则，省略部分无用信息：

*#iptables-save*

\*nat

-A POSTROUTING -s 172.17.0.4/32 -d 172.17.0.4/32 -p tcp -m tcp --dport 8080 -j MASQUERADE

...

\*filter

-A DOCKER -d 172.17.0.4/32 ! -i docker0 -o docker0 -p tcp -m tcp --dport 8080 -j ACCEPT

 可以看到，在nat、filter的Docker链中分别增加了一条规则，这两条规则将访问宿主机8080端口的流量转发到了172.17.0.4的8080端口上（真正提供服务的Docker容器IP和端口），所以外界访问Docker容器是通过iptables做DNAT（目的地址转换）实现的。  
 此外，Docker的forward规则默认允许所有的外部IP访问容器，可以通过在filter的DOCKER链上添加规则来对外部的IP访问做出限制，比如只允许源IP192.168.0.0/16的数据包访问容器，需要添加如下规则：

iptables -I DOCKER -i docker0 ! -s 192.168.0.0/16 -j DROP

 不仅仅是与外界间通信，Docker容器之间互个通信也受到iptables规则限制。同一台宿主机上的Docker容器默认都连在docker0网桥上，它们属于一个子网，这是满足相互通信的第一步。同时，Docker daemon会在filter的FORWARD链中增加一条ACCEPT的规则（--icc=true）：

-A FORWARD -i docker0 -o docker0 -j ACCEPT

 这是满足相互通信的第二步。当Docker datemon启动参数--icc(icc参数表示是否允许容器间相互通信)设置为false时，以上规则会被设置为DROP，Docker容器间的相互通信就被禁止，这种情况下，想让两个容器通信就需要在docker run时使用 --link选项。

 在Docker容器和外界通信的过程中，还涉及了数据包在多个网卡间的转发（如从docker0网卡转发到宿主机ens160网卡），这需要内核将ip-forward功能打开，即将ip\_forward系统参数设1。Docker daemon启动的时候默认会将其设为1（--ip-forward=true），也可以通过命令手动设置：

**#** echo 1 > /proc/sys/net/ipv4/ip\_forward

**#** cat /proc/sys/net/ipv4/ip\_forward

1

* **Docker容器的DNS和主机名**  
   同一个Docker镜像可以启动很多Docker容器，通过查看，它们的主机名并不一样，也即是说主机名并非是被写入镜像中的。实际上容器中/etc/目录下有3个文件是容器启动后被虚拟文件覆盖的，分别是/etc/hostname、/etc/hosts、/etc/resolv.conf，通过在容器中运行mount命令可以查看：
* *# docker exec -it tomcat01 bash*
* root@3d95d30c69d3:/usr/local/tomcat*# mount*
* ...
* /dev/mapper/centos-root on /etc/resolv.conf type xfs (rw,relatime,attr2,inode64,noquota)
* /dev/mapper/centos-root on /etc/hostname type xfs (rw,relatime,attr2,inode64,noquota)
* /dev/mapper/centos-root on /etc/hosts type xfs (rw,relatime,attr2,inode64,noquota)

...

 这样能解决主机名的问题，同时也能让DNS及时更新（改变resolv.conf）。由于这些文件的维护方法随着Docker版本演进而不断变化，因此尽量不修改这些文件，而是通过Docker提供的参数进行相关设置，配置方式如下：

* -h HOSTNAME 或 --hostname=HOSTNAME：设置容器的主机名，此名称会写在/etc/hostname和/etc/hosts文件中，也会在容器的bash提示符看到。但是在外部，容器的主机名是无法查看的，不会出现在其他容器的hosts文件中，即使使用docker ps命令也查看不到。此参数是docker run命令的参数，而非docker daemon的启动参数。
* --dns=IP\_ADDRESS...：为容器配置DNS，写在/etc/resolv.conf中。该参数即可以在docker daemon 启动的时候设置，也可以在docker run时设置，默认为8.8.8或8.8.4.4。

注意：对以上3个文件的修改不会被docker commit保存，也就是不会保存在镜像中，重启容器也会导致修改失效。另外，在不稳定的网络环境下使用需要特别注意DNS的设置。

http://blog.51cto.com/ganbing/2087598