**Docker网络配置**

**摘要**: 当docker启动时，它会在宿主机器上创建一个名为docker0的虚拟网络接口。它会从RFC 1918定义的私有地址中随机选择一个主机不用的地址和子网掩码，并将它分配给docker0。例如当我启动docker几分钟后它选择了172.17.42. ...

|  |
| --- |
| **摘要**  　　当docker启动时，它会在宿主机器上创建一个名为docker0的虚拟网络接口。它会从[RFC 1918](http://tools.ietf.org/html/rfc1918)定义的私有地址中随机选择一个主机不用的地址和子网掩码，并将它分配给docker0。例如当我启动docker几分钟后它选择了172.17.42.1/16－一个16位的子网掩码为主机和它的容器提供了65,534个ip地址。  **注意:** 本文讨论了Docker的高级网络配置和选项。通常你不会用到这些。如果你想查看一个较为简单的Docker网络介绍和容器概念介绍来着手，请参见[Docker用户指南](https://docs.docker.com/userguide/dockerlinks/).    　　但docker0并不是正常的网络接口。它只是一个在绑定到这上面的其他网卡间自动转发数据包的虚拟以太网桥。它可以使容器与主机相互通信。每次Docker创建一个容器，它就会创建一对对等接口(peer interface)，类似于一个管子的两端－在这边可以收到另一边发送的数据包。Docker会将对等接口中的一个做为eth0接口连接到容器上，并使用类似于vethAQI2QT这样的惟一名称来持有另一个，该名称取决于主机的命名空间。通过将所有veth\*接口绑定到docker0桥接网卡上，Docker在主机和所有Docker容器间创建一个共享的虚拟子网。  　　本文其他部分将会讲解使用Docker选项的所有方式，并且－在高级模式下－使用纯linux网线配置命令来  　　调整，补充，或完全替代Docker的默认网络配置。    **Docker选项快速指南**  　　这里有一份关于Docker网络配置的命令行选项列表，省去您查找相关资料的麻烦。    　　一些网络配置的命令行选项只能在服务器启动时提供给Docker服务器。并且一旦启动起来就无法改变。  　　一些网络配置命令选项只能在启动时提供给Docker服务器，并且在运行中不能改变:   * -b BRIDGE或--bridge=BRIDGE— see    [建立自己的网桥](https://docs.docker.com/articles/networking/#bridge-building) * --bip=CIDR— see    [定制docker0](https://docs.docker.com/articles/networking/#docker0) * -H SOCKET...或--host=SOCKET...—   它看起来像是在设置容器的网络，但实际却恰恰相反：它告诉Docker服务器要接收命令的通道，例如“run container"和"stop container"。 * --icc=true|false— see    [容器间通信](https://docs.docker.com/articles/networking/#between-containers) * --ip=IP\_ADDRESS— see    [绑定容器端口](https://docs.docker.com/articles/networking/#binding-ports) * --ip-forward=true|false— see    [容器间通信](https://docs.docker.com/articles/networking/#between-containers) * --iptables=true|false— see   [容器间通信](https://docs.docker.com/articles/networking/#between-containers) * --mtu=BYTES— see    [定制docker0](https://docs.docker.com/articles/networking/#docker0)   　　有两个网络配置选项可以在启动时或调用docker run时设置。当在启动时设置它会成为docker run的默认值:   * --dns=IP\_ADDRESS...— see    [配置DNS](https://docs.docker.com/articles/networking/#dns) * --dns-search=DOMAIN...— see    [配置DNS](https://docs.docker.com/articles/networking/#dns)   　　最后，一些网络配置选项只能在调用docker run时指出，因为它们要为每个容器做特定的配置:   * -h HOSTNAME或--hostname=HOSTNAME— see    [配置DNS](https://docs.docker.com/articles/networking/#dns) 和  [Docker与容器连接原理](https://docs.docker.com/articles/networking/" \l "container-networking) * --link=CONTAINER\_NAME:ALIAS— see   [配置DNS](https://docs.docker.com/articles/networking/#dns) and    [容器间通信](https://docs.docker.com/articles/networking/#between-containers) * --net=bridge|none|container:NAME\_or\_ID|host— see   [Docker与容器连接原理](https://docs.docker.com/articles/networking/" \l "container-networking) * -p SPECor--publish=SPEC— see    [绑定容器端口](https://docs.docker.com/articles/networking/#binding-ports) * -P或--publish-all=true|false— see    [绑定容器端口](https://docs.docker.com/articles/networking/#binding-ports)   　　接下来的部分会对以上话题从易到难做出逐一解答。  **配置DNS**  　　怎样为Docker提供的每一个容器进行主机名和DNS配置，而不必建立自定义镜像并将主机名写到里面？它的诀窍是覆盖三个至关重要的在/etc下的容器内的虚拟文件，那几个文件可以写入新的信息。你可以在容器内部运行mount看到这个：  $$ mount.../dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 .../dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...tmpfs on /etc/resolv.conf type tmpfs ......  　　这样的配置允许Docker去做聪明的事情，类似于当主机接收到新的DHCP配置之后，保持resolv.conf的数据到所有的容器中。Docker怎样维护在容器内的这些文件从Docker的一个版本到下一个版本的具体细节，你应该抛开这些单独的文件本身并且使用下面的Docker选项代替。  　　有四种不同的选项会影响容器守护进程的服务名称。  　　1. -h HOSTNAME 或者 --hostname=HOSTNAME  --设置容器的主机名，仅本机可见。这种方式是写到/etc/hostname ，以及/etc/hosts 文件中，作为容器主机IP的别名，并且将显示在容器的bash中。不过这种方式设置的主机名将不容易被容器之外可见。这将不会出现在 docker ps 或者 其他的容器的/etc/hosts 文件中。  　　2. --link=CONTAINER\_NAME:ALIAS  --使用这个选项去run一个容器将在此容器的/etc/hosts文件中增加一个主机名ALIAS，这个主机名是名为CONTAINER\_NAME 的容器的IP地址的别名。这使得新容器的内部进程可以访问主机名为ALIAS的容器而不用知道它的IP。--link= 关于这个选项的详细讨论请看：    [Communication between containers](https://docs.docker.com/articles/networking/#between-containers).  　　3. --dns=IP\_ADDRESS --设置DNS服务器的IP地址，写入到容器的/etc/resolv.conf文件中。当容器中的进程尝试访问不在/etc/hosts文件中的主机A时，容器将以53端口连接到IP\_ADDRESS这个DNS服务器去搜寻主机A的IP地址。  　　4. --dns-search=DOMAIN --设置DNS服务器的搜索域，以防容器尝试访问不完整的主机名时从中检索相应的IP。这是写入到容器的/etc/resolv.conf文件中的。当容器尝试访问主机 host，而DNS搜索域被设置为 example.com ,那么DNS将不仅去查寻host主机的IP，还去查询host.example.com的IP。  　　在docker中，如果启动容器时缺少以上最后两种选项设置时，将使得容器的/etc/resolv.conf文件看起来和宿主主机的/etc/resolv.conf文件一致。这些选项将修改默认的设置。    **容器间通信**  　　在操作系统层面上，决定两个容器间的通信能否得到控制，有以下三个因素。   1. 网络拓扑逻辑是否连接上了容器的网络接口。默认情况下Docker将把所有容器绑定到一个 singledocker0bridge,并为两个容器间的包传输提供路径。参见本文档后续部分---其他可能的拓扑逻辑 2. 主机是否要发送IP包？这由ip\_forward系统参数控制。如果这个参数设为1，那么数据包只能在容器间传输。通常情况下，让Docker服务器使用它的默认设置  --ip-forward=true ,   Docker在启动的时候会把ip\_forwardsh.  要检查设置或手动设置参数，可以这样做： 3. # Usually not necessary: turning on forwarding,# on the host where your Docker server is running$ cat /proc/sys/net/ipv4/ip\_forward0$ sudo echo 1 > /proc/sys/net/ipv4/ip\_forward   $ cat /proc/sys/net/ipv4/ip\_forward1   1. iptables是否允许特殊连接?如果你把设置 --iptables=false,当守护进程启动时，Docker不会改变你的系统iptables规则。另外，如果你保留默认设置  --icc=true，Docker服务器或向FORWARD链添加一个带有全局ACCEPT策略的默认规则。如果不保留默认设置，系统会把策略设为DROP.     　　几乎所有人使用docker都希望ip\_forward 是打开的，至少使容器间的通讯成为可能。但是否同意 --icc=true 或者更改为 --icc=false 使得iptables 可以保护容器以及宿主主机不被任意地端口扫描、避免被已经被渗透的容器所访问，这是一个策略问题。（在ubuntu，是编辑/etc/default/docker文件中的DOCKER\_OPTS参数，然后重启docker服务）  　　如果你选择最安全的设置 --icc=false ，那么当你想让它们彼此提供服务的时候如何让它们相互通讯？  　　答案是：使用前文提到的 --link=CONTAINER\_NAME:ALIAS 选项。如果docker守护进程正在以 --icc=false 和 --iptables=true 参数运行，当以选项 --link= 执行 docker run 命令时，docker服务将插入一部分 iptables ACCEPT 规则使得新容器可以连接其他容器所暴露出来的端口（此端口指前文在 Dockerfile 中提到的EXPOSE这一行）。更多详细文档介绍请看：[linking Docker containers](https://docs.docker.com/userguide/dockerlinks)。  **注意**: --link 选项中的 CONTAINER\_NAME 的值必须是 docker自动分配的容器名称，比如 stupefied\_pare， 或者是在执行docker run 的时候用 --name= 指定的容器名称.  这不能使一个docker无法识别的主机名。 |

你可以在你的Docker主机上运行iptables命令，来观察FORWARD链是否有默认的ACCEPT或DROP策略

# When --icc=false, you should see a DROP rule:$ sudo iptables -L -n...Chain FORWARD (policy ACCEPT)target     prot opt source               destination

DROP       all  --  0.0.0.0/0            0.0.0.0/0...# When a --link= has been created under --icc=false,# you should see port-specific ACCEPT rules overriding# the subsequent DROP policy for all other packets:$ sudo iptables -L -n...Chain FORWARD (policy ACCEPT)target     prot opt source               destination

ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tcp spt:80ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tcp dpt:80DROP       all  --  0.0.0.0/0            0.0.0.0/0

**注意**: Docker的iptables规则完全显示了容器相互间的原始IP地址，所以一个容器到另一个容器的连接，需要显示地显示出第一个容器的原始IP地址。

**为主机绑定容器端口**

　　默认情况下，Docker容器可以连接到外部区域，但外部区域不能连接到容器。在Docker启动时，由于它在主机上创建了一个iptables伪装规则，使得每一个输出连接看起来都是由主机IP地址建立起来的。

# You can see that the Docker server creates a# masquerade rule that let containers connect# to IP addresses in the outside world:$ sudo iptables -t nat -L -n...Chain POSTROUTING (policy ACCEPT)target     prot opt source               destination

MASQUERADE  all  --  172.17.0.0/16       !172.17.0.0/16...

　　当调用docker run的时候，如果你想让容器接受输入连接，你需要提供特殊选项。这些选项的详细说明在 [Docker User Guide](https://docs.docker.com/userguide/dockerlinks).  有两种方法可以实现。

　　首先，你可以提供 -P 或者 --publish-all=true|false 选项参数来执行 docker run 命令，这将会识别所有在dockerfile中暴露的端口，并且随机映射到 49000-49900 之间的主机端口。这看起来是一个很大的不便，当你要启动一个新的容器时你需要知道那个主机端口已经被映射。

　　更方便的操作是使用 -p SPEC 或者 --publish=SPEC 选项，这两个选项让你明确的指定docker容器的端口映射到任意的主机端口中，不局限于49000-49900.

　　无论如何，你应该通过审查你的NAT表，去看看docker在你的网络占做了什么。

# What your NAT rules might look like when Docker

# is finished setting up a -P forward:

$ iptables -t nat -L -n

...Chain

DOCKER (2 references)

target     prot opt source               destination

DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:49153 to:172.17.0.2:80

# What your NAT rules might look like when Docker

# is finished setting up a -p 80:80 forward:

Chain

DOCKER (2 references)

target     prot opt source               destination

DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:80 to:172.17.0.2:80

　　可以看到，docker暴露了这些容器的端口到通配IP地址：0.0.0.0 ，这个通配IP地址可以匹配宿主主机上任意一个可以进入的端口。如果你希望更多的限制，并且只允许容器服务通过特殊的宿主主机的外部网络接口来相互联系，那么你有两种选择。当你执行 docker run 命令时，你可以使用 -p IP:host\_port:container\_port 或者  -p IP::port 来明确地绑定外部接口。

　　或者如果你希望dokcer永远转发到一个特殊的IP地址上，你可以编辑你的docker系统设置文件（ubuntu系统的设置方法为：编辑 /etc/default/docker文件，改写DOCKER\_OPTS参数），增加选项 --ip=IP\_ADDRESS 。修改完之后记得重启你的docker服务。

　　如果你希望更详细的指导，请参考： [Docker User Guide](https://docs.docker.com/userguide/dockerlinks/) .

**定制 docker0**

　　默认地，docker服务会在linux内核新建一个网络桥接docker0，使得物理主机和其他虚拟网络接口之间可以传递发送数据包，因此，这表现如一个独立的网络。

　　docker0有一个IP地址和子网掩码，使得物理主机可以从容器的桥接网络接收和发送数据包。并且给这个桥接网络一个MTU（最大传输单元）或者说网络接口允许的最大包长度-例如1,500 bytes 或者从docker的宿主主机上的网络接口拷贝的数值。在服务启动的时候两者都是可配置的：

* --bip=CIDR— 为docker0桥接网络提供一个特殊的IP地址和一个子网掩码, 使用标准的 CIDR 记法例如192.168.1.5/24.
* --mtu=BYTES— 从写docker0的最大数据包长度。

　　在ubuntu系统上，你可以增加以上的配置到 /etc/default/docker 文件中的DOCKER\_OPTS参数中，然后重启docker服务。

　　当你有一个或多个正常运行的容器时，你可以通过在主机上运行brct1命令，观察interfaces列的输出，来确定Docker已经将这些容器正确地连接到docker0网桥。下面是一个连接了两个不同容器的主机：

# Display bridge info$ sudo brctl show

bridge name     bridge id               STP enabled     interfaces

docker0         8000.3a1d7362b4ee       no              veth65f9

                                                        vethdda6

　　如果你的Docker主机还没安装brct1命令，那么你可以在Ubuntu上运行sudo apt-get install bridge-utils来安装它。

　　最后，每次新建一个容器的时候都会用到docker0 桥接网络。每次在执行docker run命令新建一个容器的时候，docker从可利用的桥接网络中随机选择一个未被使用的IP地址，以及使用桥接网络的子网掩码，用来配置容器 eth0网络接口。docker宿主主机的IP地址被docker容器作为默认的网关。

# The network, as seen from a container$ sudo docker run -i -t --rm base /bin/bash

$$ ip addr show eth024: eth0: <BROADCAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP group default qlen 1000

    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff

    inet 172.17.0.3/16 scope global eth0

       valid\_lft forever preferred\_lft forever

    inet6 fe80::306f:e0ff:fe35:5791/64 scope link

       valid\_lft forever preferred\_lft forever

$$ ip routedefault via 172.17.42.1 dev eth0172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3$$ exit

　　记住docker的宿主主机无法转发docker容器的数据包到因特网上，除非它的ip\_forward 系统设置为1，详情请看： [Communication between containers](https://docs.docker.com/articles/networking/#between-containers) 。

**建立你自己的桥接网络**

　　如果你希望建立完整的自己的桥接网络，你可以在启动docker之前用 -b BRIDGE 或者 --bridge=BRIDGE选项参数高数docker使用你自己的桥接网络。如果你已经用docker0启动docker了，你需要停止docker服务然后移除docker0.

# Stopping Docker and removing docker0$ sudo service docker stop

$ sudo ip link set dev docker0 down

$ sudo brctl delbr docker0

　　然后，在启动docker服务之前，新建你自己的桥接网络，写上你想要的配置。接下来我们新建一个简单的桥接网络，刚好用这些选项来定做docker0 ，这刚好足够说明这个技术。

# Create our own bridge$ sudo brctl addbr bridge0

$ sudo ip addr add 192.168.5.1/24 dev bridge0

$ sudo ip link set dev bridge0 up# Confirming that our bridge is up and running$ ip addr show bridge04: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default

    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff

    inet 192.168.5.1/24 scope global bridge0

       valid\_lft forever preferred\_lft forever# Tell Docker about it and restart (on Ubuntu)$ echo 'DOCKER\_OPTS="-b=bridge0"' >> /etc/default/docker

$ sudo service docker start

　　运行结构应该是docker服务成功启动，已经准备好绑定容器到桥接网络上。当核实好桥接网络的配置之后，尝试着新建一个容器，你将看到容器的IP地址是在你的新的桥接网络范围内的，这是docker自动检测的。

　　正如前文所述，可以用 brctl show 命令查看，新增或者移除网络接口，可以在docker容器中执行 ip addr 和 ip route 命令查看IP地址是否是从网桥IP段分配的，以及docker的宿主主机的IP是否被作为默认网关。

**Docker 如何使容器连接到网络**

　　docker是正在发展中的，并会持续提升网络配置的逻辑。当前命令行是很难满足docker新建容器时所需要的网络配置。

让我们回顾一些基础知识。

　　通讯的时候使用网际协议（IP），一个机器需要访问至少一个网络接口用来发送和接收包，路由表定义了通过接口可达IP地址范围。网络接口不一定非是物理设备。实际上，在每一个Linux机器（和每个Docker容器内部）的lo回环接口都是有效的而且完全是虚拟的——Linux内核简单地拷贝回环（数据）包，直接从发送者的内存放入接收者的内存。

　　Docker使用特殊的虚拟接口让容器在主机间通讯——成对的虚拟接口被叫做“peers”，它被链接到主机内核的内部，因此（数据）包能在他们之间传输。他们简单创建，待会儿我们将会看到。

　　Docker配置容器的步骤是：

1.创建一对虚拟接口

2.在主Docker主机内部给它一个唯一的名称，比如veth65f9，绑定它到docker0或者Docker使用的任何网桥上

3.让其他的接口翻墙进入新的容器（已经提供了lo接口），在容器的独立和唯一网络接口命名空间内，重新命名它为更漂亮的名字eth0，名称不要和其他的物理接口冲突。

4.在网桥的网络地址访问内给容器的eth0一个新的IP地址，设置它的缺省路由为Docker主机在网桥上拥有的IP地址。

这些步骤结束后，容器将立即拥有一个eth0（虚拟）网卡，并会发现它自己可以和其他的容器以及互联网通讯。

　　你可以使用 --net= 这个选项来执行 docker run 启动一个容器，这个选项有一下可选参数。

* --net=bridge— 默认选项，用网桥的方式来连接docker容器。
* --net=host— 高数docker跳过配置容器的独立网络栈。本质上来说，这个参数告诉docker不去打包容器的网络层。当然，docker 容器的进程仍然被限制在它自己独有的文件系统、进程列表以及其他资源中。一个快速命令 ip addr 将像你展示docker的网络，它是建立在docker 宿主主机上的，有完整的权限去访问宿主主机的网络接口。注意这不意味着docker容器可以去重新配置宿主主机的网络栈，重新配置是需要 --privaleged=true 这个选项参数的，但是这个选项参数会让docker容器打开大量的端口以及其他的系统的超级管理权限的进程。这也会允许容器去访问宿主主机的网络服务，比如 D-bus。这会使docker容器里的进程有有权限去做一些意想不到的事，比如重启你的宿主主机。所以要谨慎使用这个选项参数。
* --net=container:NAME\_or\_ID— 告诉docker让这个新建的容器使用已有容器的网络配置。这个新建的容器将配置新的自己的文件系统和进程列表以及其他资源限制，但是将共享这个指定的容器的网络IP地址以及端口号，使得这两个容器可以通过 loopback接口相互访问。
* --net=none— 告诉docker为新建的容器建立一个网络栈，但不对这个网络栈进行任何配置，在这个文档的最后将介绍如何让你去建立自定义的网络配置。

去了解以下这一步是非常必要的，如果你在建立容器的时候使用 --net=none  这个选项参数。以下是一些命令去去配置自定义网络，就好像你让docker完全去自己配置一样。

# At one shell, start a container and# leave its shell idle and running$ sudo docker run -i -t --rm --net=none base /bin/bash

root@63f36fc01b5f:/#

# At another shell, learn the container process ID

# and create its namespace entry in /var/run/netns/# for the "ip netns" command we will be using below$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f2778$ pid=2778$ sudo mkdir -p /var/run/netns

$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid# Check the bridge's IP address and netmask$ ip addr show docker021: docker0: ...inet 172.17.42.1/16 scope global docker0...# Create a pair of "peer" interfaces A and B,# bind the A end to the bridge, and bring it up$ sudo ip link add A type veth peer name B

$ sudo brctl addif docker0 A

$ sudo ip link set A up# Place B inside the container's network namespace,# rename to eth0, and activate it with a free IP$ sudo ip link set B netns $pid

$ sudo ip netns exec $pid ip link set dev B name eth0

$ sudo ip netns exec $pid ip link set eth0 up

$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0

$ sudo ip netns exec $pid ip route add default via 172.17.42.1

到这一步你的容器应该可以正常运行网络操作了。

当你最后退出shell以及清理掉这个容器的时候，这个容器的虚拟网络 eth0 将在网络接口A 被清除后被消除，也会自动在网桥 docker0 上销毁。所以不用你执行其他的命令，所有的东西将被清理。当然，是几乎所有的东西：

# Clean up dangling symlinks in /var/run/netnsfind -L /var/run/netns -type l -delete

　　还要注意上面的脚本使用了现代的ip命令行替代旧的弃用的封装，类似ipconfig和route，这些老的命令行还是会一直呆在我们的容器内部工作。如果你很忙碌的话，ip addr命令行也可以只键入ip a。

　　总之，注意这个ip netns exec重要的命令行，它让我们以root用户进入内部并配置一个网络命名空间。如果在容器内部运行，类似的命令行可能不会工作，因为安全容器化的部分是Docker剥离容器的处理过程，这个过程要正确地配置自己的网络。使用ip netns exec可以让我们完成配置，还避免了运行容器自身--privileged=true的危险步骤。

**工具和实例**

　　在把自定义网络拓扑逻辑分类成下面几个部分之前，你应该关注一些外部工具盒关于配置的实例。下面就有两个例子：

* J&eacute;r&ocirc;me Petazzoni 创建了一个pipework的shell脚本，帮助你在复杂场景下建立容器间连接:    <https://github.com/jpetazzo/pipework>
* Brandon Rhodes 为下一个版本的Python网络编程基金会创建了整个的网络拓扑逻辑，包括路由器NAT'd防火墙，和提供HTTP,SMTP,POP,IMAP,Telnet,SSH,FTP的服务器:    <https://github.com/brandon-rhodes/fopnp/tree/m/playground>

　　两个工具都使用网络命令并和之前见到的版本很相似，在下面章节会看到。

**建立点对点连接**

　　缺省情况下, Docker通过docker0将所有的容器添加到虚拟子网中。你能够按照[Building your own bridge](https://docs.docker.com/articles/networking/" \l "bridge-building)中的方法创建你自己的桥让容器连接到不同的虚拟子网。启动容器时使用命令docker run --net=none，然后使用shell命令添加容器到你自己的桥，方法见[How Docker networks a container](https://docs.docker.com/articles/networking/" \l "container-networking)。

　　但是有时你想让两个特别的容器能够直接通讯，不用绑定到主机的以太网桥上。

　　解决方案是简单的。创建一对对等接口，将他们放到容器中，并将其配置为经典的点对点链接。两个容器就能够直接通讯了（当然要告诉每个容器对方的IP地址）。您可能会调整在上一节的指示去这样的事情：

# Start up two containers in two terminal windows$ sudo docker run -i -t --rm --net=none base /bin/bash

root@1f1f4c1f931a:/#

$ sudo docker run -i -t --rm --net=none base /bin/bash

root@12e343489d2f:/#

# Learn the container process IDs

# and create their namespace entries

$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a

2989

$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f

3004

$ sudo mkdir -p /var/run/netns

$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004# Create the "peer" interfaces and hand them out$ sudo ip link add A type veth peer name B

$ sudo ip link set A netns 2989$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A

$ sudo ip netns exec 2989 ip link set A up

$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B

$ sudo ip netns exec 3004 ip link set B up

$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B

　　两个容器应该可以相互ping通以确定连接成功。点对点链接不依赖于子网或子网掩码，但是ip route需要确认一些其他的单一IP地址是连接到了特定的网络接口。

　　请注意点对点链接可以安全的和其他类型的网络连接混合使用。如果你想用点对点链接替换容器的正常网络连接，启动的时候，不需要带参数--net=none。

　　在Docker主机和容器之间创建点对点链接是这个模板最终的排列方式，它允许主机和有单一IP地址的容器通讯。除非你有很特别的网络需求，让你尝试使用这样的解决方案，正如我们前面探讨的，使用--icc=false锁定跨容器的通讯是更好的方案。