docker网络

原创 Mr大表哥 2017-03-04 16:09:25 评论(0) 402人阅读

一、 Docker 中的网络功能介绍

默认情况下，容器可以建立到外部网络的连接，但是外部网络无法连接到容器。

Docker 允许通过外部访问容器或容器互联的方式来提供网络服务

外部访问容器:

容器中可以运行一些网络应用，要让外部也可以访问这些应用，可以通过 -P 或 -p 参数来指定端口映射。（当使用 –P(大写) 标记时，Docker 会随机映射一个随机的端口到内部容器开放的网络端口。）

注意：-P使用时需要指定--expose选项或dockerfile中用expose指定容器要暴露的端口，指定需要对外提供服务的端口。（我在这已经在dockerfile里面用expose指定了，如下图所示）

```
[root@localhost sshd_dockerfile]# cat Dockerfile
FROM docker.benet.com/centos:centos6
MAINTAINER from cyh@example.com
RUN yum -y install openssh-server sudo httpd
RUN useradd admin
RUN echo "admin:admin" | chpasswd
RUN echo "admin  ALL=(ALL)  ALL" >> /etc/sudoers
RUN ssh-keygen -t dsa -f /etc/ssh/ssh_host_dsa_key
RUN ssh-keygen -t rsa -f /etc/ssh/ssh_host_rsa_key
RUN mkdir -p /var/run/sshd
RUN mkdir -p /home/admin/.ssh
RUN sed -ri 's/session reqired pam_loginuid.so/#session requied pam_loginuid.so/g' /etc/pam.d/sshd
ADD authorized_keys /home/admin/.ssh/authorized_keys
RUN sed -ri 's/#ServerName www.example.com:80/ServerName www.benet.com/g' /etc/httpd/conf/httpd.conf
ADD run.sh /run.sh
RUN chmod 775 /run.sh
EXPOSE 22 80 443
CMD ["/bin/bash","/run.sh"]
```

【准备工作：运行一个容器，提供web服务和ssh服务

宿主机启用路由转发（net.ipv4.ip_forward=1）】

1）使用-P

```
[root@localhost ~]# docker run -dit --name=ssh_http -P centos:http
abe20cac6336610af9824bcdd35bc36db05755f25b9f986322f5364722f78e2b
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                                                                   NAMES
abe20cac6336        centos:http         "/bin/bash /run.sh"      9 seconds ago       Up 8 seconds        0.0.0.0
:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp    ssh_http
[root@localhost ~]#
```

使用 docker ps 可以看到，本地主机的32770被映射到了容器的22端口，本地主机的32769被映射到了容器的80端口，本地主机的32768被映射到了容器的443 端口。

测试：此时访问本机的 32770端口即可访问容器内 ssh 应用。（账号和密码都是admin）

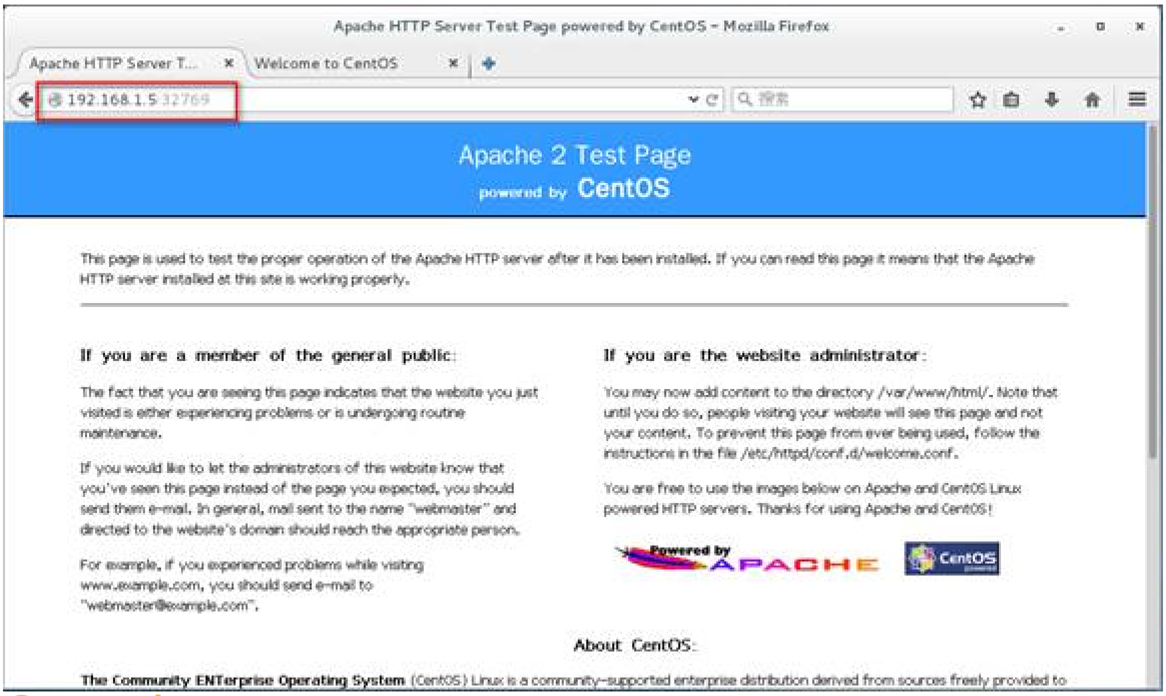
```
[root@localhost ~]# ssh admin@192.168.1.5 -p 32770
The authenticity of host '[192.168.1.5]:32770 ([192.168.1.5]:32770)' can't be established.
RSA key fingerprint is 70:8d:e2:a3:b3:26:b0:91:9c:d0:df:7e:a8:de:8b:91.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.5]:32770' (RSA) to the list of known hosts.
admin@192.168.1.5's password:
[admin@abe20cac6336 ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:50 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:7180 (7.0 KiB)  TX bytes:5747 (5.6 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[admin@abe20cac6336 ~]$
```

注明：192.168.1.5是宿主机的IP地址

查看容器运行的httpd和sshd进程

```
[admin@abe20cac6336 ~]$ pgrep httpd
8
9
10
11
12
13
14
15
16
[admin@abe20cac6336 ~]$ pgrep sshd
7
17
19
[admin@abe20cac6336 ~]$
```

测试httpd服务此时访问本机的 32769端口即可访问容器内 web 应用



2）使用-p

-p（小写）则可以指定要映射的端口，并且，在一个指定端口上只可以绑定一个容器。

支持的格式有

ip:hostPort:containerPort | ip::containerPort| hostPort:containerPort

注意:

容器有自己的内部网络和 ip 地址（使用 docker inspect 可以获取所有的变量。）

-p 标记可以多次使用来绑定多个端口

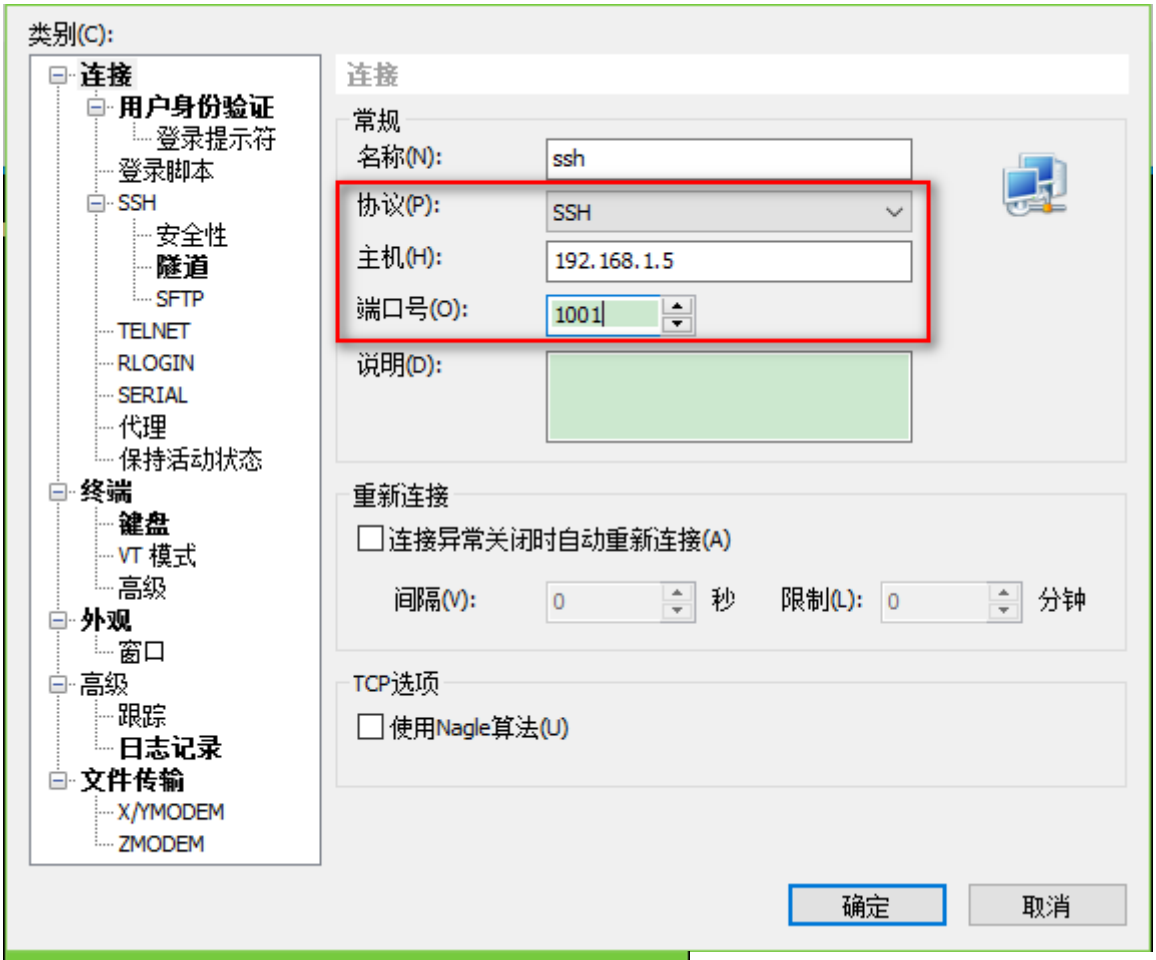
```
[root@localhost ~]# docker run -dit --name=ssh_http2 -p 8001:80 -p 1001:22 -p 4433:443 centos:http
d8a96bfe5fb4e24c0b4b2844b3699b3cb2ae2e2bc7de3a37c53fe4ce30f5f13e
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
                                                                   NAMES
d8a96bfe5fb4        centos:http         "/bin/bash /run.sh"      22 seconds ago      Up 17 seconds       0.0.0.0
:1001->22/tcp, 0.0.0.0:8001->80/tcp, 0.0.0.0:4433->443/tcp      ssh_http2
abe20cac6336        centos:http         "/bin/bash /run.sh"      39 minutes ago      Up 39 minutes       0.0.0.0
:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp   ssh_http
[root@localhost ~]#
```

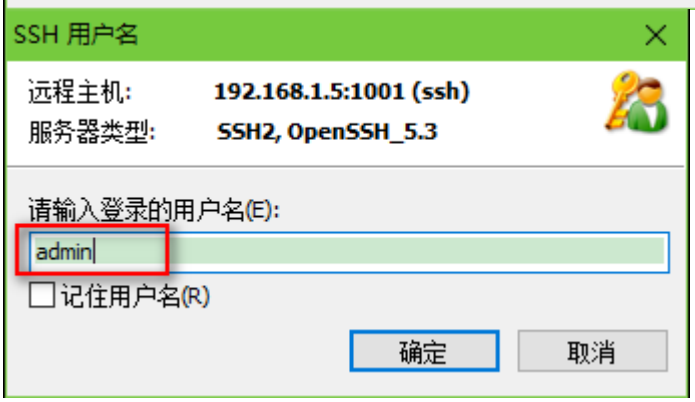
映射所有接口地址：
使用 hostPort:containerPort 格式，将本地的1001端口映射到容器的 22 端口，本地的8001端口映射到容器的80端口，本地的4433端口映射到容器的443端口

测试:

①ssh测试:

使用xshell工具


类别(C):
连接
用户身份验证
登录提示符
登录脚本
SSH
安全性
隧道
SFTP
TELNET
RLOGIN
SERIAL
代理
保持活动状态
终端
键盘
VT 模式
高级
外观
窗口
高级
跟踪
日志记录
文件传输
X/YMODEM
ZMODEM
连接
常规
名称(N): ssh
协议(P): SSH
主机(H): 192.168.1.5
端口号(O): 1001
说明(D):
重新连接
连接异常关闭时自动重新连接(A)
间隔(V): 0 秒
限制(L): 0 分钟
TCP选项
使用Nagle算法(U)
确定
取消


SSH 用户名
远程主机: 192.168.1.5:1001 (ssh)
服务器类型: SSH2, OpenSSH_5.3
请输入登录的用户名(E):
admin
记住用户名(R)
确定
取消


```
[admin@d8a96bfe5fb4 ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:68 errors:0 dropped:0 overruns:0 frame:0
          TX packets:52 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:7350 (7.1 KiB)  TX bytes:8459 (8.2 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[admin@d8a96bfe5fb4 ~]$
```

②测试web访问

映射到指定地址的指定端口

可以使用 ip:hostPort:containerPort 格式，指定映射使用一个特定地址，比如宿主机网卡配置的一个地址192.168.1.5

```
[root@localhost ~]# docker run -dit -p 192.168.1.5:1000:22 -p 192.168.1.5:8000:80 -p 192.168.1.5:4000:443 centos:http
382094b69f4edc02d3a4f53057db5d2d4b60a0e3c9197d2ee973b8f27ad54211
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
382094b69f4e        centos:http         "/bin/bash /run.sh"   8 seconds ago       Up 6 seconds        192.168.1.5:1000->22/tcp, 192.168.1.5:8000->80/tcp, 192.168.1.5:4000->443/tcp   hopeful_poincare
d8a96bfe5fb4        centos:http         "/bin/bash /run.sh"   17 minutes ago      Up 17 minutes       0.0.0.0:1001->22/tcp, 0.0.0.0:8001->80/tcp, 0.0.0.0:4433->443/tcp   ssh_http2
abe20cac6336        centos:http         "/bin/bash /run.sh"   56 minutes ago      Up 56 minutes       0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp   ssh_http
[root@localhost ~]#
```

映射到指定地址的任意端口

使用 ip::containerPort 绑定192.168.1.5的任意端口到容器的22、80、443端口，本地主机会自动分配一个口。

```
[root@localhost ~]# docker run -dit -p 192.168.1.5::22 -p 192.168.1.5::80 -p 192.168.1.5::443 centos:http
b045212dcbcbd7674ee7ad4876984516 4a406c9857e4fc4da1bd69a76eb49763
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
b045212dcbcb        centos:http         "/bin/bash /run.sh"   8 seconds ago       Up 4 seconds        192.168.1.5:32773->22/tcp, 192.168.1.5:32772->80/tcp, 192.168.1.5:32771->443/tcp   modest_blackwell
382094b69f4e        centos:http         "/bin/bash /run.sh"   2 minutes ago       Up 2 minutes        192.168.1.5:1000->22/tcp, 192.168.1.5:8000->80/tcp, 192.168.1.5:4000->443/tcp   hopeful_poincare
d8a96bfe5fb4        centos:http         "/bin/bash /run.sh"   19 minutes ago      Up 19 minutes       0.0.0.0:1001->22/tcp, 0.0.0.0:8001->80/tcp, 0.0.0.0:4433->443/tcp   ssh_http2
abe20cac6336        centos:http         "/bin/bash /run.sh"   58 minutes ago      Up 58 minutes       0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp   ssh_http
[root@localhost ~]#
```

注：还可以使用 udp 标记来指定 udp 端口

```
[root@localhost ~]# docker run -dit --name=bdqn -p 192.168.1.5:5000:50/udp centos:http
adf3bc1d3a0735598c3e00a1a4d4cb21359a387f2fe0636476f6f5a411d13c7c
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
adf3bc1d3a07        centos:http         "/bin/bash /run.sh"   5 seconds ago       Up 3 seconds        22/tcp, 80/tcp, 443/tcp, 192.168.1.5:5000->50/udp   bdqn
b045212dcbcb        centos:http         "/bin/bash /run.sh"   4 minutes ago       Up 4 minutes        192.168.1.5:32773->22/tcp, 192.168.1.5:32772->80/tcp, 192.168.1.5:32771->443/tcp   modest_blackwell
382094b69f4e        centos:http         "/bin/bash /run.sh"   7 minutes ago       Up 7 minutes        192.168.1.5:1000->22/tcp, 192.168.1.5:8000->80/tcp, 192.168.1.5:4000->443/tcp   hopeful_poincare
d8a96bfe5fb4        centos:http         "/bin/bash /run.sh"   24 minutes ago      Up 24 minutes       0.0.0.0:1001->22/tcp, 0.0.0.0:8001->80/tcp, 0.0.0.0:4433->443/tcp   ssh_http2
abe20cac6336        centos:http         "/bin/bash /run.sh"   About an hour ago   Up About an hour    0.0.0.0:32770->22/tcp, 0.0.0.0:32769->80/tcp, 0.0.0.0:32768->443/tcp   ssh_http
[root@localhost ~]#
```

查看映射端口配置（使用 docker port 来查看当前映射的端口配置，也可以查看到绑定的地址）

```
[root@localhost ~]# docker port ssh_http
80/tcp -> 0.0.0.0:32769
22/tcp -> 0.0.0.0:32770
443/tcp -> 0.0.0.0:32768
[root@localhost ~]#
```

```
[root@localhost ~]# docker port hopeful_poincare
22/tcp -> 192.168.1.5:1000
443/tcp -> 192.168.1.5:4000
80/tcp -> 192.168.1.5:8000
[root@localhost ~]#
```

Docker NATiptables实现

默认情况下，容器可以主动访问到外部网络的连接，但是外部网络无法访问到容器

1）容器访问外部实现

容器所有到外部网络的连接，源地址都会被 NAT 成本地系统的 IP 地址（即docker0地址）。这是使用 iptables 的源地址伪装操作实现的。

（docker服务开启后，docker会自动在iptables的nat表中创建地址伪装，默认允许内访外）

查看主机的 NAT 规则：

```
[root@localhost ~]# iptables -t nat -vnL
Chain POSTROUTING (policy ACCEPT 9 packets, 638 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2   140 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0
```

其中，上述规则将所有源地址在 172.17.0.0/16 网段，目标地址为其他网段（外部网络）的流量动态伪装为从系统网卡发出。MASQUERADE 跟传统 SNAT 的好处是它能动态从网卡获取地址。

2）外部访问容器实现

容器允许外部访问，可以在 docker run 时候通过 -p 或 -P 参数来启用，不管用那种办法，其实也是在本地的 iptable 的nat 表中添加相应的规则。

①使用 -P 时：

```
Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32768 to:172.17.0.2:443
   12   656 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32769 to:172.17.0.2:80
    1    60 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32770 to:172.17.0.2:22
    9   500 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:8001 to:172.17.0.3:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:4433 to:172.17.0.3:443
    1    52 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:1001 to:172.17.0.3:22
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:8000 to:172.17.0.4:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:4000 to:172.17.0.4:443
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:1000 to:172.17.0.4:22
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32771 to:172.17.0.5:443
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32772 to:172.17.0.5:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32773 to:172.17.0.5:22
    0     0 DNAT       udp  --  !docker0 *      0.0.0.0/0            192.168.1.5         udp dpt:5000 to:172.17.0.6:50
```

②使用 -p时

```
Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32768 to:172.17.0.2:443
   12   656 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32769 to:172.17.0.2:80
    1    60 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:32770 to:172.17.0.2:22
    9   500 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:8001 to:172.17.0.3:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:4433 to:172.17.0.3:443
    1    52 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0           tcp dpt:1001 to:172.17.0.3:22
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:8000 to:172.17.0.4:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:4000 to:172.17.0.4:443
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:1000 to:172.17.0.4:22
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32771 to:172.17.0.5:443
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32772 to:172.17.0.5:80
    0     0 DNAT       tcp  --  !docker0 *      0.0.0.0/0            192.168.1.5         tcp dpt:32773 to:172.17.0.5:22
    0     0 DNAT       udp  --  !docker0 *      0.0.0.0/0            192.168.1.5         udp dpt:5000 to:172.17.0.6:50
```

docker0 网桥

Docker服务默认会创建一个 docker0 网桥（其上有一个 docker0 内部接口），它在内核层连通了其他的物理或虚拟网卡，这就将所有容器和本地主机都放到同一个物理网络。

Docker 默认指定了 docker0 接口的 IP 地址和子网掩码，让主机和容器之间可以通过网桥相互通信

由于目前 Docker 网桥是 Linux 网桥，用户可以使用 brctl show来查看网桥和端口连接信息。

```
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02423385a07d       no              veth1239c79
                                                        veth2cabc2c
                                                        veth53d3292
                                                        veth64cdb64
                                                        vethcdea4ae
virbr0          8000.000000000000       yes
[root@localhost ~]#
```

虚拟网桥不需要STP(即生成树协议)

注：brctl 命令在centos中可以使用
yum -y install bridge-utils 来安装，默认是已安装的！

```
[root@localhost ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:33ff:fe85:a07d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:33:85:a0:7d  txqueuelen 0  (Ethernet)
        RX packets 513  bytes 98334 (96.0 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 598  bytes 59905 (58.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
[root@localhost ~]# ssh admin@192.168.1.5 -p 1001
The authenticity of host '[192.168.1.5]:1001 ([192.168.1.5]:1001)' can't be established.
RSA key fingerprint is 70:8d:e2:a3:b3:26:b0:91:9c:d0:df:7e:a8:de:8b:91.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[192.168.1.5]:1001' (RSA) to the list of known hosts.
admin@192.168.1.5's password:
Last login: Tue Feb  7 11:32:48 2017 from 192.168.1.4
[admin@d8a96bfe5fb4 ~]$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
12: eth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:3/64 scope link
       valid_lft forever preferred_lft forever
[admin@d8a96bfe5fb4 ~]$ ip r
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3
[admin@d8a96bfe5fb4 ~]$
```

Docker 网络配置

Docker 四种网络模式

docker run 创建 Docker 容器时，可以用 --net 选项指定容器的网络模式，Docker 有以下 4 种网络模式：

host 模式，使用 --net=host 指定。

container 模式，使用 --net=container:NAMEorID 指定。

none 模式，使用 --net=none 指定。

bridge 模式，使用 --net=bridge 指定，默认设置。

host模式

如果启动容器的时候使用host 模式，那么这个容器将不会获得一个独立的NetworkNamespace，而是和宿主机共用一个 Network Namespace。容器将不会虚拟出自己的网卡，配置自己的 IP 等，而是使用宿主机的 IP 和端口。

例如，我们在192.168.1.5/24 的机器上用 host 模式启动一个含有 web 应用的 Docker 容器，监听 tcp 80 端口。当我们在容器中执行任何类似 ifconfig 命令查看网络环境时，看到的都是宿主机上的信息。而外界访问容器中的应用，则直接使用192.168.1.5:80 即可，不用任何 NAT 转换，就如直接跑在宿主机中一样。但是，容器的其他方面，如文件系统、进程列表等还是和宿主机隔离的。

启动容器前，执行pgrep http查看宿主机httpd进程。

```
[root@localhost ~]# pgrep httpd
[root@localhost ~]#
```

上面显示结果说明宿主机没有httpd进程运行

```
[root@localhost ~]# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
centos              http                b097bfe56b56        6 months ago        291.7 MB
docker.io/centos    latest              50dae1ee8677        6 months ago        196.7 MB
docker.io/centos    centos6             cf2c3ece5e41        7 months ago        194.6 MB
[root@localhost ~]# docker run -dit --name=test1 --net=host centos:http
099b9efe9a8f131eadb71a4382b3db2f74fa217986535b153d929ed8bb328c46
[root@localhost ~]# pgrep httpd
56614
56615
56616
56617
56618
56619
56620
56621
56622
[root@localhost ~]#
```

```
[root@localhost ~]# firewall-cmd --add-port=80/tcp
success
[root@localhost ~]#
```

container模式

这个模式指定新创建的容器和已经存在的一个容器共享一个 Network Namespace，而不是和宿主机共享。新创建的容器不会创建自己的网卡，配置自己的 IP，而是和一个指定的容器共享 IP、端口范围等。同样，两个容器除了网络方面，其他的如文件系统、进程列表等还是隔离的。两个容器的进程可以通过 lo 网卡设备通信。

```
[root@localhost ~]# docker run -dit --name=test2 docker.io/centos:centos6
b15c171922e5bc19fd07c4d92fdc0b0f37fc2958f050940c029ffa86ca27f809
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE                      COMMAND                  CREATED             STATUS
NAMES
b15c171922e5        docker.io/centos:centos6   "/bin/bash"              12 seconds ago      Up 11 seconds
test2
099b9efe9a8f        centos:http                "/bin/bash /run.sh"      11 minutes ago      Up 11 minutes
test1
[root@localhost ~]# docker run -it --name=test3 --net=container:b15c171922e5 docker.io/centos:centos6
[root@b15c171922e5 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 b)  TX bytes:648 (648.0 b)
```

先建一个默认模式桥接模式的容器，命名叫test2

可以看见test3容器的IP是172.16.0.2，而172.16.0.2正是test2容器的IP地址

```
[root@localhost ~]# docker attach test2
[root@b15c171922e5 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 b)  TX bytes:648 (648.0 b)
```

进入正在运行的test2容器里面

可以看见test2的ip就是172.16.0.2，说明上一个截图上test3的ip共用的就是test2的ip

none模式

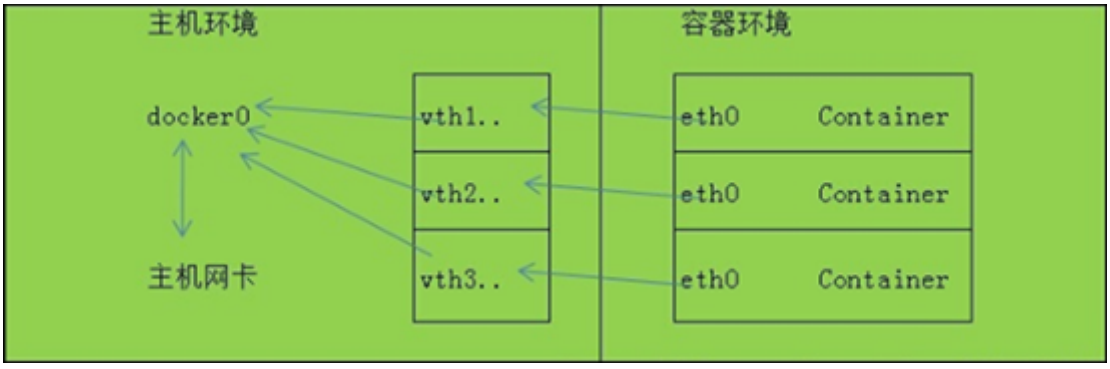
这个模式和前两个不同。在这种模式下，Docker 容器拥有自己的 Network Namespace，但是，并不为 Docker容器进行任何网络配置。也就是说，这个 Docker 容器没有网卡、IP、路由等信息。需要我们自己为 Docker 容器添加网卡、配置 IP 等。

bridge模式

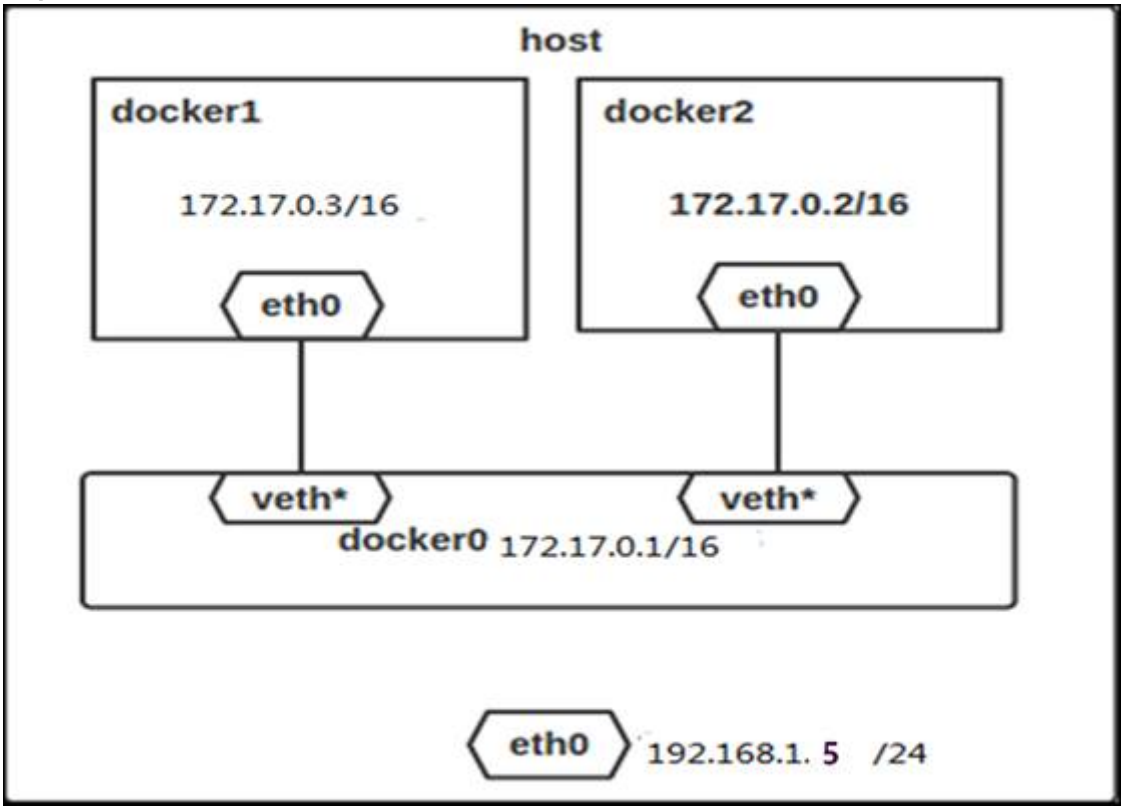
当 docker 启动时，会在主机上创建一个docker0 的虚拟网卡。他随机挑选 RFC1918 私有网络中的一段地址给docker0 。比如 172.17.0.1/16,16 位掩码的网段可以拥有 65534 个地址可以使用，这对主机和容器来说应该足够了。

docker0 不是普通的网卡，他是桥接到其他网卡的虚拟网卡，容器使用它来和主机相互通信。当创建一个 docker 容器的时候，它就创建了一个对接口，当数据包发送到一个接口时，另外一个接口也可以收到相同的数据包，它们是绑在一起的一对孪生接口。这对接口在容器中那一端的的名字是 eth0 ，宿主主机端的会指定一个唯一的名字，比如 vethAQI2QT 这样的名字。

所有的 veth* 的接口都会桥接到 docker0，这样 docker 就创建了在主机和所有容器之间一个虚拟共享网。



bridge 模式是 Docker 默认的网络设置，此模式会为每一个容器分配 NetworkNamespace、设置 IP 等，并将一个主机上的Docker 容器连接到一个虚拟网桥上。当 Docker server 启动时，会在主机上创建一个名为 docker0 的虚拟网桥，此主机上启动的 Docker 容器会连接到这个虚拟网桥上。虚拟网桥的工作方式和物理交换机类似，这样主机上的所有容器就通过交换机连在了一个二层网络中。接下来就要为容器分配 IP 了，Docker 会从 RFC1918 所定义的私有 IP 网段中，选择一个和宿主机不同的IP地址和子网分配给 docker0，连接到 docker0 的容器就从这个子网中选择一个未占用的 IP 使用。如一般 Docker 会使用 172.17.0.0/16 这个网段，并将 172.17.0.1/16 分配给 docker0 网桥（在主机上使用 ifconfig 命令是可以看到 docker0 的，可以认为它是网桥的管理接口，在宿主机上作为一块虚拟网卡使用）



Docker完成以上网络配置的过程大致是这样的：

1.在主机上创建一对虚拟网卡veth pair设备。veth设备总是成对出现的，它们组成了一个数据的通道，数据从一个设备进入，就会从另一个设备出来。因此，veth设备常用来连接两个网络设备。

2.Docker将veth pair设备的一端放在新创建的容器中，并命名为eth0。另一端放在主机中，以vethd6368d7这样类似的名字命名，并将这个网络设备加入到docker0网桥中，可以通过brctl show命令查看。

注：brctl 工具依赖 bridge-utils 软件包。

```
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02423385a07d       no              vethd6368d7
virbr0          8000.000000000000       yes
```

3.从docker0子网中分配一个IP给容器使用，并设置docker0的IP地址为容器的默认网关。(容器内部访问外网以及容器和主机之间的端口映射都是通过Iptables实现的，可以查看Iptables表分析)

查看当前 docker0地址

```
[root@localhost ~]# ifconfig docker0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:33ff:fe85:a07d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:33:85:a0:7d  txqueuelen 0  (Ethernet)
        RX packets 592  bytes 108249 (105.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 686  bytes 71009 (69.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

在容器运行时，每个容器都会分配一个特定的虚拟机口并桥接到 docker0。每个容器都会配置同docker0 ip 相同网段的专用 ip 地址，docker0 的 IP 地址被用于所有容器的默认网关。

```
[root@localhost ~]# ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:33ff:fe85:a07d  prefixlen 64  scopeid 0x20<link>
        ether 02:42:33:85:a0:7d  txqueuelen 0  (Ethernet)
        RX packets 592  bytes 108249 (105.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 686  bytes 71009 (69.3 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.5  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 1000  (Ethernet)
        RX packets 568896  bytes 800547224 (763.4 MiB)
        RX errors 0  dropped 40  overruns 0  frame 0
        TX packets 70891  bytes 5053738 (4.8 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 386  bytes 46303 (45.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 386  bytes 46303 (45.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

vethd6368d7: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::f4e6:86ff:fe81:eb74  prefixlen 64  scopeid 0x20<link>
        ether f6:e6:86:81:eb:74  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8  bytes 648 (648.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

virbr0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 192.168.122.1  netmask 255.255.255.0  broadcast 192.168.122.255
        ether 00:00:00:00:00:00  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
```

```
[root@b15c171922e5 /]# 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
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
[root@b15c171922e5 /]#
```

```
[root@localhost ~]# docker run -dit --name=test4 docker.io/centos:centos6
152d67acbc19b513f09772680d0c1f43f2f94ebb42b7312cca8605647d569c5e
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02423385a07d       no              vethd6368d7
                                                        vethde683cf
virbr0          8000.000000000000       yes
[root@localhost ~]#
```

```
[root@localhost ~]# docker attach test4
[root@152d67acbc19 /]# 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
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
[root@152d67acbc19 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:03
          inet addr:172.17.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 b)  TX bytes:648 (648.0 b)
```

以上, docker0 扮演着test2和test4这两个容器的虚拟接口 vethxx interface 桥接的角色。

执行docker network inspect bridge查看所有桥接网络的详细信息

```
[root@localhost ~]# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "e31dbee0b927b066c7d73524bfb6803081e97b5244ad340d013cc062daa2ac07",
        "Scope": "local",
        "Driver": "bridge",
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16"
                }
            ]
        },
        "Containers": {
            "152d67acbc19b513f09772680d0c1f43f2f94ebb42b7312cca8605647d569c5e": {
                "Name": "test4",
                "EndpointID": "ab05bb9692112bda95582a0589ea712a5e3f18db13f6813b3d055bbae1947198",
                "MacAddress": "02:42:ac:11:00:03",
                "IPv4Address": "172.17.0.3/16",
                "IPv6Address": ""
            },
            "b15c171922e5bc19fd07c4d92fdc0b0f37fc2958f050940c029ffa86ca27f809": {
                "Name": "test2",
                "EndpointID": "88e1f51b7b11374c6a9ed55ce66e51a0e4759087ec5276143e2637333aff75cf",
                "MacAddress": "02:42:ac:11:00:02",
                "IPv4Address": "172.17.0.2/16",
                "IPv6Address": ""
            }
        },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        }
    }
]
```

自定义网桥

除了默认的 docker0 网桥，用户也可以指定网桥来连接各个容器。在启动 Docker 服务的时候，使用 -b BRIDGE 或 --bridge=BRIDGE 来指定使用的网桥。

Docker 允许你管理 docker0 桥接或者通过-b选项自定义桥接网卡，需要安装bridge-utils软件包。

基本步骤如下：

1. 确保 docker 的进程是停止的
2. 创建自定义网桥
3. 给网桥分配特定的 ip
4. 以 -b 的方式指定网桥

具体操作步骤：

1）如果服务已经运行，那需要先停止服务，并删除旧的网桥

```
[root@localhost ~]# systemctl stop docker
[root@localhost ~]# ip link set dev docker0 down
[root@localhost ~]# brctl delbr docker0
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.000000000000       yes
[root@localhost ~]#
```

2）然后创建一个网桥 bridge0，给网桥分配特定的 ip

```
[root@localhost ~]# brctl addbr bridge0
[root@localhost ~]# ip addr add 192.168.10.1/24 dev bridge0
[root@localhost ~]# ip link set dev bridge0 up
[root@localhost ~]#
```

3）查看确认网桥创建并启动

```
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
bridge0         8000.000000000000       no
virbr0          8000.000000000000       yes
[root@localhost ~]#
```

或者

```
[root@localhost ~]# ip addr show bridge0
26: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
    link/ether ea:c6:ae:77:49:94 brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
    inet6 fe80::e8c6:aeff:fe77:4994/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]#
```

4）修改/etc/sysconfig/docker文件（添加截图中黄色部分）

```
# Modify these options if you want to change the way the docker daemon runs
OPTIONS='--selinux-enabled --log-driver=journald -b=bridge0'
DOCKER_CERT_PATH=/etc/docker
```

5）启动 Docker 服务

```
[root@localhost ~]# systemctl start docker
[root@localhost ~]#
```

6）新建一个容器，可以看到它已经桥接到了 bridge0 上

```
[root@localhost ~]# docker run -dit --name=test5 docker.io/centos:centos6
b3e63f13e4784c0bd16f8eead628a53fe3b7ce68d2dd4a2c9471cabce456a240
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
bridge0         8000.5ad373832f1f       no              veth0b0d58a
virbr0          8000.000000000000       yes
[root@localhost ~]#
```

7）进入容器，查看容器的IP

```
[root@localhost ~]# docker attach test5
[root@b3e63f13e478 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:0A:02
          inet addr:192.168.10.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:c0ff:fea8:a02/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 b)  TX bytes:648 (648.0 b)
[root@b3e63f13e478 /]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.10.1    0.0.0.0         UG    0      0        0 eth0
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
[root@b3e63f13e478 /]#
```

小结：docker 服务启动成功并绑定容器到新的网桥，新建一个容器，你会看到它的 ip 是我们的设置的新 ip段，docker 会自动检测到它。用 brctl show 可以看到容器启动或则停止后网桥的配置变化，在容器中使用 ip a 和 ip r 来查看 ip 地址配置和路由信息。

【让我们回顾一些基础知识：

机器需要一个网络接口来发送和接受数据包，路由表来定义如何到达哪些地址段。这里的网络接口可以不是物理接口。事实上，每个 linux 机器上的 lo 环回接口（docker 容器中也有）就是一个完全的linux 内核虚拟接口，它直接复制发送缓存中的数据包到接收缓存中。 docker 让宿主主机和容器使用特殊的虚拟接口来通信，通信的 2 端叫“ peers“，他们在主机内核中连接在一起，所以能够相互通信。创建他们很简单，前面介绍过了。】

docker 创建容器的思想步骤如下：

创建一对虚拟接口

其中宿主主机一端使用一个名字比如 veth65f9 ， 他是唯一的，另外一端桥接到默认的 docker0, 或其它你指定的桥接网卡。

主机上的 veth65f9 这种接口映射到新的新容器中的名称通常是 eth0, 在容器这个隔离的 networknamespace 中，它是唯一的，不会与其他接口名字和它冲突。

从主机桥接网卡的地址段中获取一个空闲地址给 eth0 使用，并设定默认路由到桥接网卡。

完成这些之后，容器就可以使用这eth0 虚拟网卡来连接其他容器和其他网络。

你也可以为特殊的容器设定特定的参数，在 docker run 的时候使用 --net ，它有 4 个可选参数：

--net=bridge ： 默认连接到 docker0 网桥。

--net=host ： 告诉 docker 不要将容器放到隔离的网络堆栈中。尽管容器还是有自己的文件系统、进程列表和资源限制。但使用 ip addr 命令这样命令就

可以知道实际上此时的的容器处于和 docker 宿主主机的一样的网络级别，它拥有完全的宿主主机接口访问权限。虽然它不允许容器重新配置主机的网络堆栈，除非 --privileged=true，—但是容器进程可以跟其他 root 进程一样可以打开低数字的端口，可以访问本地网络服务比如 D-bus ，还可以让容器做一些意想不到的事情，比如重启主机，使用这个选项的时候要非常小心！

--net=container:NAME_or_ID ： 告诉 docker 将新容器的进程放到一个已经存在的容器的网络堆栈中，新容器进程有它自己的文件系统、进程列表和资源限制，但它会和那个已经存在的容器共享 ip 地址和端口，他们之间来可以通过环回接口通信。

--net=none ： 告诉 docker 将新容器放到自己的网络堆栈中，但是不要配置它的网络。

下面通过配置一个以 --net=none 启动的容器，使他达到跟平常一样具有访问网络的权限。来介绍docker是如何连接到容器中的。

1）启动一个运行 /bin/bash的容器，并指定 --net=none

```
[root@localhost ~]# docker run -it --name=test6 --net=none docker.io/centos:centos6
[root@dfc44319a880 /]# ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

2）再开启一个新的终端，查找这个容器的进程 id ，然后创建它的命名空间，后面的
ip netns 会用到

```
[root@localhost ~]# pid=$(docker inspect -f '{{.State.Pid}}' dfc44319a880)
[root@localhost ~]# echo $pid
59371
[root@localhost ~]# mkdir -p /var/run/netns
[root@localhost ~]# ln -s /proc/$pid/ns/net /var/run/netns/$pid
[root@localhost ~]# ls /var/run/netns/$pid
/var/run/netns/59371
[root@localhost ~]#
```

3）检查桥接网卡的 ip 和子网掩码

```
[root@localhost ~]# brctl show
bridge name     bridge id               STP enabled     interfaces
bridge0         8000.5ad373832f1f       no              veth0b0d58a
virbr0          8000.000000000000       yes
[root@localhost ~]# ip addr show bridge0
26: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 5a:d3:73:83:2f:1f brd ff:ff:ff:ff:ff:ff
    inet 192.168.10.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
    inet6 fe80::e8c6:aeff:fe77:4994/64 scope link
       valid_lft forever preferred_lft forever
[root@localhost ~]#
```

4）创建一对“peer”接口 A和 B ，绑定 A到网桥，并启用它

```
[root@localhost ~]# ip link add A type veth peer name B
[root@localhost ~]# brctl addif bridge0 A
[root@localhost ~]# ip link set A up
[root@localhost ~]# ifconfig
A: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether a6:a2:b3:77:b9:fb  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

bridge0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.10.1  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::e8c6:aeff:fe77:4994  prefixlen 64  scopeid 0x20<link>
        ether 5a:d3:73:83:2f:1f  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 536 (536.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 26  bytes 3871 (3.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

5）将 B 放到容器的网络命名空间，命名为 eth0，配置一个空闲的 ip

```
[root@localhost ~]# ip link set B netns $pid
[root@localhost ~]# ip netns exec $pid ip link set dev B name eth0
[root@localhost ~]# ip netns exec $pid ip link set eth0 up
[root@localhost ~]# ip netns exec $pid ip addr add 192.168.10.100/24 dev eth0
[root@localhost ~]# ip netns exec $pid ip route add default via 192.168.10.1
[root@localhost ~]#
```

6）自此，你就可以像平常一样使用网络了

```
[root@localhost ~]# docker attach test6
[root@dfc44319a880 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 42:5E:56:3D:5D:E6
          inet addr:192.168.10.100  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::405e:56ff:fe3d:5de6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:648 (648.0 b)  TX bytes:648 (648.0 b)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[root@dfc44319a880 /]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.10.1    0.0.0.0         UG    0      0        0 eth0
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
[root@dfc44319a880 /]# nslookup www.baidu.com
Server:         192.168.1.1
Address:        192.168.1.1#53

Non-authoritative answer:
www.baidu.com   canonical name = www.a.shifen.com.
Name:   www.a.shifen.com
Address: 61.135.169.121
Name:   www.a.shifen.com
Address: 61.135.169.125
```

当你退出容器后， docker 清空容器，容器的 eth0 随网络命名空间一起被摧毁， A 接口也被自动从docker0 取消注册。不用其他命令，所有东西都被清理掉了！

注意 ip netns exec 命令，它可以让我们像 root 一样配置网络命名空间。但在容器内部无法使用，因为统一的安全策略，docker限制容器进程配置自己的网络。使用 ip netns exec 可以让我们不用设置--privileged=true 就可以完成一些可能带来危险的操作。

扩展1）创建一个点到点连接（不做重点）

默认情况下，Docker 会将所有容器连接到由 docker0 提供的虚拟子网中。

用户有时候需要两个容器之间可以直连通信，而不用通过主机网桥进行桥接。

解决办法很简单：创建一对 peer 接口，分别放到两个容器中，配置成点到点链路类型即可。

首先启动 2 个容器：

```
#docker run -i -t --rm --net=none 镜像id/bin/bash
root@1f1f4c1f931a:/#
#docker run -i -t --rm --net=none镜像id/bin/bash
root@12e343489d2f:/#
```

找到进程号，然后创建网络名字空间的跟踪文件。

```
#docker inspect -f '``.`State`.`Pid`' 1f1f4c1f931a
2989
#docker inspect -f '``.`State`.`Pid`' 12e343489d2f
3004
#mkdir -p /var/run/netns
#ln -s /proc/2989/ns/net /var/run/netns/2989
#ln -s /proc/3004/ns/net /var/run/netns/3004
```

创建一对 peer 接口，然后配置路由

```
#ip link add A type veth peer name B

#ip link set A netns 2989
#ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
#ip netns exec 2989 ip link set A up
#ip netns exec 2989 ip route add 10.1.1.2/32 dev A

#ip link set B netns 3004
#ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
```

#ip netns exec 3004 ip link set B up
#ip netns exec 3004 ip route add 10.1.1.1/32 dev B
现在这 2 个容器就可以相互 ping 通，并成功建立连接。点到点链路不需要子网和子网掩码

扩展2）DNS/HOSTNAME自定义

Docker 没有为每个容器专门定制镜像，那么怎么自定义配置容器的主机名和 DNS 配置呢？秘诀就是它利用虚拟文件来挂载到来容器的 3 个相关配置文件。
在容器中使用 mount 命令可以看到挂载信息：注（mount命令软件包util-linux）
mount
...

...
这种机制可以让宿主主机 DNS 信息发生更新后，所有 Docker 容器的 dns 配置通过/etc/resolv.conf文件立刻得到更新。
如果用户想要手动指定容器的配置，可以利用下面的选项。
-h HOSTNAME or --hostname=HOSTNAME 设定容器的主机名，它会被写到容器内的/etc/hostname 和 /etc/hosts。但它在容器外部看不到，既不会在 docker ps 中显示，也不会在其他的容器的 /etc/hosts 看到。

--link=CONTAINER_NAME:ALIAS 选项会在创建容器的时候，添加一个其他容器的主机名到
/etc/hosts 文件中，让新容器的进程可以使用主机名ALIAS 就可以连接它。
--dns=IP_ADDRESS 添加 DNS 服务器到容器的 /etc/resolv.conf 中，让容器用这个服务器来解析所有不在/etc/hosts 中的主机名。
--dns-search=DOMAIN 设定容器的搜索域，当设定搜索域为.example.com 时，在搜索一个名为 host的主机时， DNS 不仅搜索 host，还会搜索host.example.com。注意：如果没有上述最后 2 个选项，Docker 会默认用主机上的 /etc/resolv.conf 来配置容器。
具体其他选项可以查看docker run --help帮助

容器互联

使用--link参数可以让容器之间安全的进行交互。

1）下面先创建一个新的数据库容器

```
[root@localhost ~]# docker run -dit --name=dbserver docker.io/centos:centos6
b3b2e4382d4c1d0713b0988969c9bd458e76220a700346795c92032a489ea873
```

2）然后创建一个新的 web 容器，并将它连接到 dbserver 容器

```
[root@localhost ~]# docker run -dit -P --name=web --link dbserver:db docker.io/centos:centos6
82f2adaed2d96038a0d4d4e9a9c4374eb1aa0a60b9e7586fae12ad458314ba1a
[root@localhost ~]#
```

--link 标记的格式： --link name:alias ， name 是我们要链接的容器的名称， alias 是这个链接的别名。

3)使用docker ps来查看容器的连接

```
[root@localhost ~]# docker ps --no-trunc
CONTAINER ID                                                       IMAGE                        COMMAND       CREATED
    STATUS              PORTS               NAMES
82f2adaed2d96038a0d4d4e9a9c4374eb1aa0a60b9e7586fae12ad458314ba1a   docker.io/centos:centos6     "/bin/bash"   2 minutes ago
    Up 2 minutes                            web
b3b2e4382d4c1d0713b0988969c9bd458e76220a700346795c92032a489ea873   docker.io/centos:centos6     "/bin/bash"   5 minutes ago
    Up 4 minutes                            dbserver,web/db
[root@localhost ~]#
```

可以查看容器之间的链接情况

我们可以看到我们命名的容器， dbserver 和 web ， dbserver 容器的 names 列有 dbserver 也有 web/db，这表示 web 容器链接到 dbserver容器，他们是一个父子关系。在这个 link 中， 2 个容器中有一对父子关系。 docker 在 2 个容器之间创建了一个安全的连接，而且不用映射dbserver容器的端口到宿主主机上。所以在启动 db 容器的时候也不用 -p 和 -P 标记。使用 link 之后我们就可以不用暴露数据库端口到网络上。

注意：你可以链接多个子容器到父容器，比如我们可以链接多个 db 到 web 容器上。

4)docker 会添加子容器的 host 信息到父容器的 /etc/hosts 的文件中，我们来查看父容器（web）的hosts文件

```
[root@localhost ~]# docker attach web
[root@82f2adaed2d9 /]# cat /etc/hosts
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.2      db b3b2e4382d4c dbserver
172.17.0.3      82f2adaed2d9
[root@82f2adaed2d9 /]#
```

172.17.0.2是dbserver容器的IP地址；172.17.0.3是web容器的IP地址

这里有 2 个 hosts，第一个是 dbserver 容器，web 容器用 id 作为他的主机名，第二个是 web 容器的 ip 和主机名。可以在 web 容器中安装 ping 命令来测试跟dbserver容器的连通。

5）在 web 容器中安装 ping 命令来测试跟dbserver容器的连通。

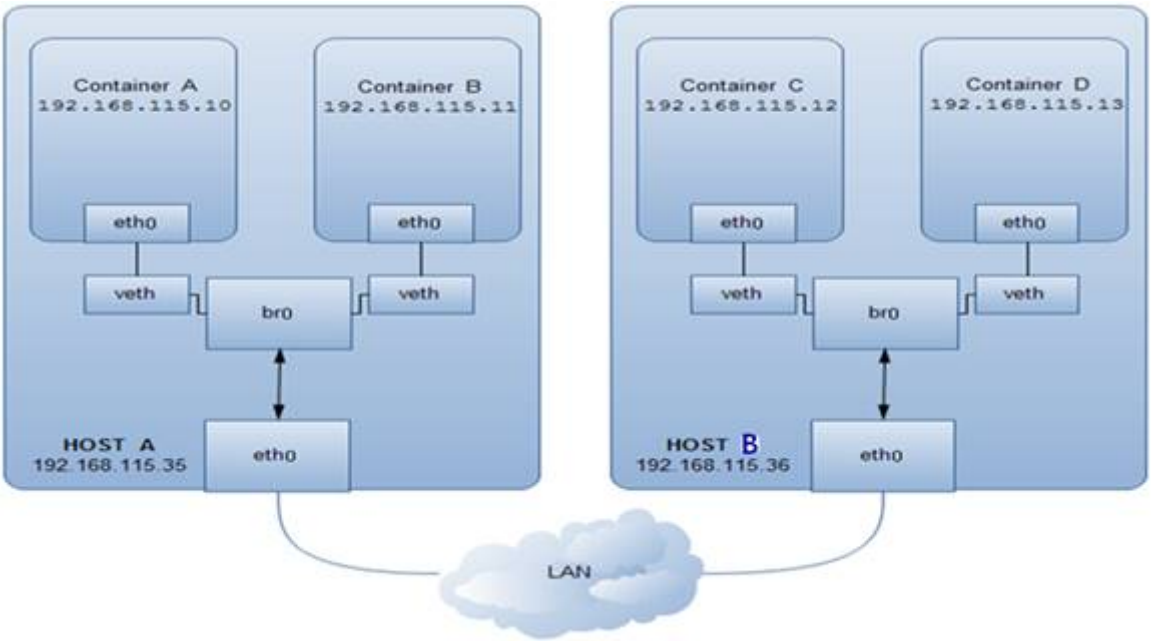
注意：官方的镜像默认没有安装 ping，需要自行安装，软件包名iputils

```
[root@82f2adaed2d9 /]# yum -y install iputils
```

```
[root@82f2adaed2d9 /]# ping dbserver
PING db (172.17.0.2) 56(84) bytes of data.
64 bytes from db (172.17.0.2): icmp_seq=1 ttl=64 time=0.127 ms
64 bytes from db (172.17.0.2): icmp_seq=2 ttl=64 time=0.162 ms
64 bytes from db (172.17.0.2): icmp_seq=3 ttl=64 time=0.063 ms
64 bytes from db (172.17.0.2): icmp_seq=4 ttl=64 time=0.114 ms
64 bytes from db (172.17.0.2): icmp_seq=5 ttl=64 time=0.055 ms
^C
--- db ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4279ms
rtt min/avg/max/mdev = 0.055/0.104/0.162/0.040 ms
[root@82f2adaed2d9 /]#
```

附：在bridge模式下，连在同一网桥上的容器可以相互通信（若出于安全考虑，也可以禁止它们之间通信，方法是在DOCKER_OPTS变量中设置--icc=false，这样只有使用--link才能使两个容器通信）。

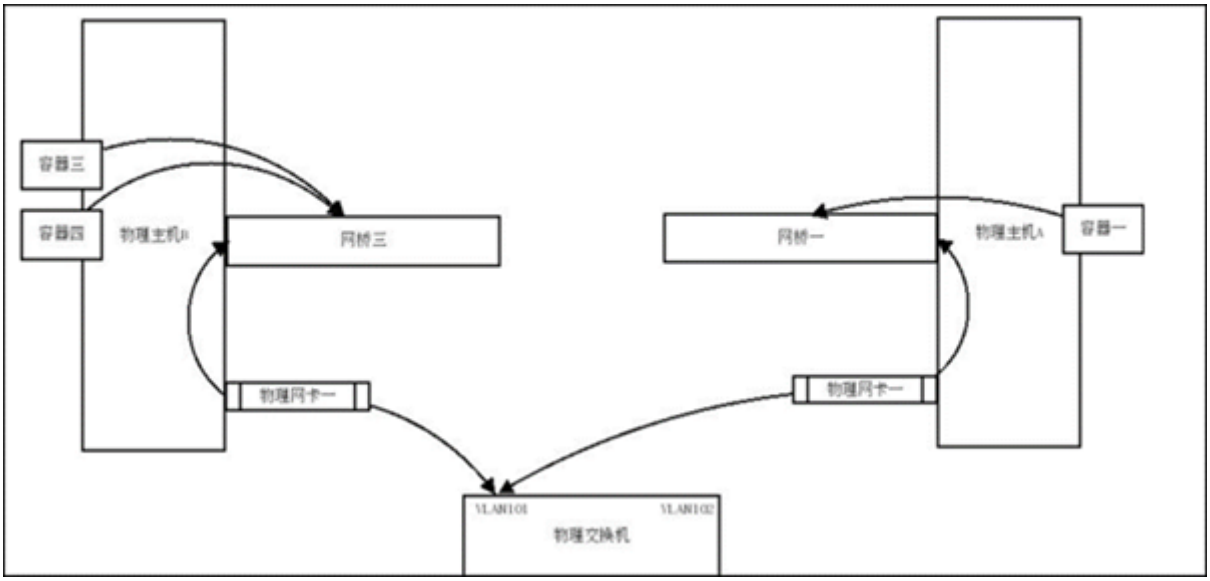
多台物理主机之间的容器互联（暴露容器到真实网络中）



docker 默认的桥接网卡是 docker0 。它只会在本机桥接所有的容器网卡，举例来说容器的虚拟网卡在主机上看一般叫做 vethxxx，而 docker 只是把所有这些网卡桥接在一起。

这样就可以把这个网络看成是一个私有的网络，通过 nat 连接外网，如果要让外网连接到容器中，就需要做端口映射，即 -p 参数。

如果在企业内部应用，或则做多个物理主机的集群，可能需要将多个物理主机的容器组到一个物理网络中来，那么就需要将这个网桥桥接到我们指定的网卡上。



主机 A 的网卡一和主机 B 的网卡三都连着物理交换机的同一个 vlan 101，这样网桥一和网桥三就相当于在同一个物理网络中了，而容器一、容器三、容器四也在同一物理网络中了，他们之间可以相互通信，而且可以跟同一 vlan 中的其他物理机器互联。

这样就直接把容器暴露到物理网络上了，多台物理主机的容器也可以相互联网了。需要注意的是，这样就需要自己来保证容器的网络安全了。

不同容器之间的通信可以借助于 pipework 这个工具

pipework是由Docker的工程师JérômePetazzoni开发的一个Docker网络配置工具，由200多行shell实现，方便易用。

下载地址：wget https://github.com/jpetazzo/pipework.git

1）解压缩pipwork软件

```
[root@localhost 20170209_102914]# cp pipework-master.zip /usr/src/
[root@localhost 20170209_102914]# cd /usr/src/
[root@localhost src]# unzip pipework-master.zip
Archive:  pipework-master.zip
e56f189a7b02fa083704dabf654ee6b9b008ff5c
   creating: pipework-master/
 extracting: pipework-master/.gitignore
  inflating: pipework-master/LICENSE
  inflating: pipework-master/README.md
  inflating: pipework-master/docker-compose.yml
   creating: pipework-master/doctoc/
  inflating: pipework-master/doctoc/Dockerfile
  inflating: pipework-master/pipework
  inflating: pipework-master/pipework.spec
[root@localhost src]# cp -p /usr/src/pipework-master/pipework /usr/local/bin/
[root@localhost src]#
```

2）安装相应依赖软件

```
[root@localhost src]# yum -y install bridge-utils
已加载插件：fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: ftp.sjtu.edu.cn
 * extras: ftp.sjtu.edu.cn
 * updates: ftp.sjtu.edu.cn
软件包 bridge-utils-1.5-9.el7.x86_64 已安装并且是最新版本
无须任何处理
[root@localhost src]#
```

这个软件默认已经安装，如果没安装则必须安装

3）配置桥接网络

```
[root@localhost ~]# cd /etc/sysconfig/network-scripts/
[root@localhost network-scripts]# vim ifcfg-eno16777736
```

```
TYPE=Ethernet
BOOTPROTO=none
NM_CONTROLLED=no
HWADDR=00:0c:29:e7:13:d3
DEFROUTE=yes
PEERDNS=yes
PEERROUTES=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_PEERDNS=yes
IPV6_PEERROUTES=yes
IPV6_FAILURE_FATAL=no
NAME=eno16777736
UUID=9ec6e9f5-c342-426c-ae6c-91a159e5a36e
DEVICE=eno16777736
ONBOOT=yes
BRIDGE="br0"
```

```
[root@localhost network-scripts]# vim ifcfg-br0
```

```
DEVICE=br0
BOOTPROTO=static
NM_CONTROLLED=no
ONBOOT=yes
TYPE=Bridge
IPADDR=192.168.1.100
NETMASK=255.255.255.0
```

4）重启network服务

```
[root@localhost network-scripts]# systemctl restart network
[root@localhost network-scripts]# ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.100  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 0  (Ethernet)
        RX packets 7  bytes 449 (449.0 B)
        RX errors 0  dropped 4  overruns 0  frame 0
        TX packets 11  bytes 1483 (1.4 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:e0ff:fe77:d591  prefixlen 64  scopeid 0x20<link>
        ether 02:42:e0:77:d5:91  txqueuelen 0  (Ethernet)
        RX packets 14472  bytes 776443 (758.2 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21041  bytes 96802044 (92.3 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 1000  (Ethernet)
        RX packets 642549  bytes 901182787 (859.4 MiB)
        RX errors 0  dropped 40  overruns 0  frame 0
```

```
[root@localhost network-scripts]# brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.000c29e713d3       no              eno16777736
docker0         8000.0242e077d591       no
virbr0          8000.000000000000       yes
[root@localhost network-scripts]#
```

把 docker 的桥接指定为 br0，这样跨主机不同容器之间通过 pipework 新建 docker 容器的网卡桥接到 br0，这样跨主机容器之间就可以通信了。

5）关掉docker服务，并修改docker服务的配置文件

```
[root@localhost ~]# systemctl stop docker
[root@localhost ~]# vim /etc/sysconfig/docker
```

```
1 # /etc/sysconfig/docker
2
3 # Modify these options if you want to change the way the
4 OPTIONS='--selinux-enabled --log-driver=journald -b=br0'
5 DOCKER_CERT_PATH=/etc/docker
```

```
[root@localhost ~]# systemctl start docker
[root@localhost ~]#
```

6）把 docker 默认桥接指定到了br0，则最好在创建容器的时候加上--net=none，防止自动分配的 IP 在局域网中有冲突。使用镜像运行一个容器

```
[root@localhost ~]# docker run -dit --name=httpd --net=none centos:http
b479f7e3ab0abd5f776c847c2b73e01da911f137b61999473d39a425e78b461d
[root@localhost ~]#
```

```
[root@localhost ~]# docker run -dit --name=httpd --net=none centos:http
cec343403a1a2d121f7f0a35e3e23a65241714d0108c6b67d10c886f72fd3c41
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
           NAMES
cec343403a1a        centos:http         "/bin/bash /run.sh"   19 seconds ago      Up 17 seconds
           httpd
[root@localhost ~]# pipework br0 -i eth0 cec343403a1a 192.168.1.200/24@192.168.1.1
[root@localhost ~]#
```

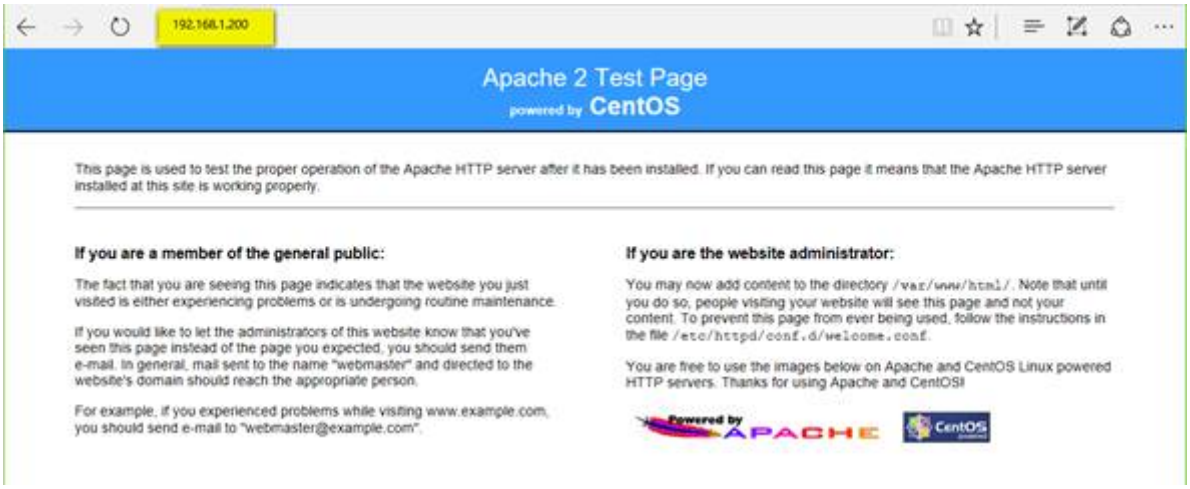
192.168.1.1是本网段的路由器的网关（即路由器接口的IP），不是br0的IP地址

注：默认不指定网卡设备名，则默认添加为 eth1。

注：另外 pipework 不能添加静态路由，如果有需求则可以在 run 的时候加上 --privileged=true 权限在容器中手动添加，但这种安全性有缺陷，可以通过 ip netns 操作

```
[root@localhost ~]# ssh admin@192.168.1.200
The authenticity of host '192.168.1.200 (192.168.1.200)' can't be established.
RSA key fingerprint is 70:8d:e2:a3:b3:26:b0:91:9c:d0:df:7e:a8:de:8b:91.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.1.200' (RSA) to the list of known hosts.
admin@192.168.1.200's password:
[admin@cec343403a1a ~]$ ifconfig
eth0      Link encap:Ethernet  HWaddr D2:75:74:7A:D4:E4
          inet addr:192.168.1.200  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::d075:74ff:fe7a:d4e4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:84 errors:0 dropped:32 overruns:0 frame:0
          TX packets:41 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9339 (9.1 KiB)  TX bytes:5899 (5.7 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          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:0
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

[admin@cec343403a1a ~]$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
[admin@cec343403a1a ~]$
```

7）从一台客户机上访问容器提供的web服务



8)使用ip netns添加静态路由，避免创建容器使用--privileged=true选项造成一些不必要的安全问题

```
[root@localhost ~]# docker inspect --format="{{.State.Pid}}" cec343403a1a
66570
[root@localhost ~]# ln -s /proc/66570/ns/net /var/run/netns/66570
[root@localhost ~]# ip netns exec 66570 ip route add 192.168.0.0/24 dev eth0 via 192.168.1.1
[root@localhost ~]#
```

9)进入容器查看路由记录

```
[root@localhost ~]# docker exec -it httpd /bin/bash
[root@cec343403a1a /]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth0
192.168.0.0     192.168.1.1     255.255.255.0   UG    0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
[root@cec343403a1a /]#
```

在其它宿主机进行相应的配置，新建容器并使用 pipework 添加虚拟网卡桥接到 br0，测试通信情况即可。

注：可以删除 docker0，直接把 docker 的桥接指定为 br0。也可以保留使用默认的配置，这样单主机容器之间的通信可以通过 docker0，而跨主机不同容器之间通过 pipework 新建 docker 容器的网卡桥接到 br0，这样跨主机容器之间就可以通信了。

扩展：

pipework可以在下面用三个场景来使用和工作原理。

一、将Docker容器配置到本地网络环境中

为了使本地网络中的机器和Docker容器更方便的通信，我们经常会有将Docker容器配置到和主机同一网段的需求。这个需求其实很容易实现，我们只要将Docker容器和主机的网卡桥接起来，再给Docker容器配上IP就可以了。

下面我们来操作一下，我主机A地址为192.168.1.4/24,网关为192.168.1.1,需要给Docker容器的地址配置为192.168.1.100/24。

在主机A上做如下操作：

安装pipework

下载地址：wgethttps://github.com/jpetazzo/pipework.git

unzip pipework-master.zip

1）解压缩pipework

```
[root@localhost media]# cp pipework-master.zip /usr/src/
[root@localhost media]# cd /usr/src/
[root@localhost src]# unzip pipework-master.zip
Archive:  pipework-master.zip
e56f189a7b02fa083704dabf654ee6b9b008ff5c
   creating: pipework-master/
 extracting: pipework-master/.gitignore
  inflating: pipework-master/LICENSE
  inflating: pipework-master/README.md
  inflating: pipework-master/docker-compose.yml
   creating: pipework-master/doctoc/
  inflating: pipework-master/doctoc/Dockerfile
  inflating: pipework-master/pipework
  inflating: pipework-master/pipework.spec
[root@localhost src]# cp -p /usr/src/pipework-master/pipework /usr/local/bin/
[root@localhost src]#
```

2)启动Docker容器

```
[root@localhost ~]# docker run -dit --name=test1 docker.io/centos:centos6
7ca083bd3385ac4bd6d38d8999e15d1f87e18397bd8b5c4e55266c0029021a71
[root@localhost ~]#
```

3)配置容器网络，并连到网桥br0上。网关在IP地址后面加@指定。

```
[root@localhost ~]# pipework br0 test1 192.168.1.100/24@192.168.1.1
[root@localhost ~]#
```

pipework会检查是否存在br0网桥，若不存在，就自己创建。所以br0不用提前建好。

```
[root@localhost ~]# ifconfig
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.4  netmask 255.255.255.0  broadcast 0.0.0.0
        inet6 fe80::70ed:92ff:fe89:63ef  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 0  (Ethernet)
        RX packets 35614  bytes 231763244 (221.0 MiB)
        RX errors 0  dropped 326  overruns 0  frame 0
        TX packets 31258  bytes 2149100 (2.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        inet6 fe80::42:e5ff:fe19:3b90  prefixlen 64  scopeid 0x20<link>
        ether 02:42:e5:19:3b:90  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 536 (536.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1758 (1.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 1000  (Ethernet)
        RX packets 535325  bytes 770253437 (734.5 MiB)
        RX errors 0  dropped 36  overruns 0  frame 0
        TX packets 69589  bytes 4753636 (4.5 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

4)将主机eno16777736桥接到br0上，并把eno16777736的IP配置在br0上。

```
[root@localhost ~]# ip addr add 192.168.1.4/24 dev br0
[root@localhost ~]# ip addr del 192.168.1.4/24 dev eno16777736
[root@localhost ~]# brctl addif br0 eno16777736
[root@localhost ~]# ip route del default
RTNETLINK answers: No such process
[root@localhost ~]# ip route add default via 192.168.1.1 dev br0
[root@localhost ~]#
```

忽略掉这个提示即可

注：如果是远程操作，中间网络会断掉，所以放在一条命令中执行。

```
ip addr add 192.168.1.102/24 dev br0; \ ip addr del
192.168.1.102/24 dev enp0s3; \ brctl addif br0 enp0s3; \ ip route
del default; \ ip route add default via 192.168.1.1 dev br0
```

5)完成上述步骤后，我们发现Docker容器已经可以使用新的IP和主机网络里的机器相互通信了。

进入容器内部查看容器的地址

```
[root@localhost ~]# docker attach test1
[root@7ca083bd3385 /]# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2406 (2.3 KiB)  TX bytes:648 (648.0 b)

eth1      Link encap:Ethernet  HWaddr 6E:5B:B2:F3:0F:49
          inet addr:192.168.1.100  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::6c5b:b2ff:fef3:f49/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:141 errors:0 dropped:64 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:10918 (10.6 KiB)  TX bytes:780 (780.0 b)

[root@7ca083bd3385 /]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.1.1     0.0.0.0         UG    0      0        0 eth1
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 eth0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 eth1
[root@7ca083bd3385 /]#
```

6)在本网段内用一台客户机ping test1容器

```
Microsoft Windows [版本 10.0.14393]
(c) 2016 Microsoft Corporation。保留所有权利。

C:\Users\81959>ping 192.168.1.100

正在 Ping 192.168.1.100 具有 32 字节的数据:
来自 192.168.1.100 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.100 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.100 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.1.100 的回复: 字节=32 时间<1ms TTL=64

192.168.1.100 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 0ms, 最长 = 0ms, 平均 = 0ms

C:\Users\81959>
```

pipework工作原理分析：

那么容器到底发生了哪些变化呢？我们docker attach到test1上，发现容器中多了一块eth1的网卡，并且配置了192.168.1.100/24的IP，而且默认路由也改为了192.168.1.1。这些都是pipework帮我们配置的。

首先pipework检查是否存在br0网桥，若不存在，就自己创建。

创建veth pair设备，用于为容器提供网卡并连接到br0网桥。

使用docker inspect找到容器在主机中的PID，然后通过PID将容器的网络命名空间链接到/var/run/netns/目录下。这么做的目的

是，方便在主机上使用ip netns命令配置容器的网络。因为，在Docker容器中，我们没有权限配置网络环境。

将之前创建的veth pair设备分别加入容器和网桥中。在容器中的名称默认为eth1，可以通过pipework的-i参数修改该名称。

然后就是配置新网卡的IP。若在IP地址的后面加上网关地址，那么pipework会重新配置默认路由。这样容器通往外网的流量会经由新配置的eth1出去，而不是通过eth0和docker0。（若想完全抛弃自带的网络设置，在启动容器的时候可以指定--net=none）
以上就是pipework配置Docker网络的过程，这和Docker的bridge模式有着相似的步骤。事实上，Docker在实现上也采用了相同的底层机制。
通过源代码，可以看出，pipework通过封装Linux上的ip、brctl等命令，简化了在复杂场景下对容器连接的操作命令，为我们配置复杂的网络拓扑提供了一个强有力的工具。当然，如果想了解底层的操作，我们也可以直接使用这些Linux命令来完成工作，甚至可以根据自己的需求，添加额外的功能。

二、单主机Docker容器VLAN划分

pipework不仅可以使用Linuxbridge连接Docker容器，还可以与OpenVswitch结合，实现Docker容器的VLAN划分。下面，就来简单演示一下，在单机环境下，如何实现Docker容器间的二层隔离。
为了演示隔离效果，我们将4个容器放在了同一个IP网段中。但实际他们是二层隔离的两个网络，有不同的广播域。

1）安装openvswitch的依赖软件，搭建其安装运行基础环境（直接用在线yum源安装即可）

```
[root@localhost ~]# yum -y install gcc make python-devel openssl-devel kernel-devel graphviz kernel-debug-devel autoconf automake rpm-build redhat-rpm-config libtool
```

2）下载并打包（即源码包重新封装成rpm包）openvswitch的包（我这已经下载好了，我直接解压缩打包即可）

wget http://openvswitch.org/releases/openvswitch-2.3.1.tar.gz

```
[root@localhost media]# tar zxf openvswitch-2.3.1.tar.gz -C /usr/src/
[root@localhost media]# mkdir -p ~/rpmbuild/SOURCES
[root@localhost media]# cp openvswitch-2.3.1.tar.gz ~/rpmbuild/SOURCES/
[root@localhost media]# sed 's/openvswitch-kmod, //g' /usr/src/openvswitch-2.3.1/rhel/openvswitch.spec > /usr/src/openvswitch-2.3.1/rhel/openvswitch_no_kmod.spec
[root@localhost media]# rpmbuild -bb --without check /usr/src/openvswitch-2.3.1/rhel/openvswitch_no_kmod.spec
```

3）之后会在~/rpmbuild/RPMS/x86_64/里有2个文件

```
[root@localhost ~]# ls -l ~/rpmbuild/RPMS/x86_64/
总用量 9552
-rw-r--r--. 1 root root 2013832 2月  10 20:34 openvswitch-2.3.1-1.x86_64.rpm
-rw-r--r--. 1 root root 7763720 2月  10 20:34 openvswitch-debuginfo-2.3.1-1.x86_64.rpm
[root@localhost ~]#
```

4）selinux必须关闭

```
[root@localhost ~]# vim /etc/sysconfig/selinux

  7 SELINUX=disabled
  8 # SELINUXTYPE= can take
  9 #     targeted - Targete
 10 #     minimum - Modifica
 11 #     mls - Multi Level
 12 SELINUXTYPE=targeted
```

```
[root@localhost ~]# reboot
```

5）安装2个文件的第一个就行

```
[root@localhost ~]# yum localinstall ~/rpmbuild/RPMS/x86_64/openvswitch-2.3.1-1.x86_64.rpm

总计：8.2 M
安装大小：8.2 M
Is this ok [y/d/N]: y
```

6）启动

```
[root@localhost ~]# /sbin/chkconfig openvswitch on
[root@localhost ~]# /sbin/service openvswitch start
Starting openvswitch (via systemctl):                      [  OK  ]
[root@localhost ~]#
```

或者

```
[root@localhost ~]# systemctl start openvswitch
```

7）查看状态

```
[root@localhost ~]# /sbin/service openvswitch status
ovsdb-server is running with pid 13183
ovs-vswitchd is running with pid 13196
[root@localhost ~]#
```

或者

```
[root@localhost ~]# systemctl status openvswitch
● openvswitch.service - LSB: Open vSwitch switch
   Loaded: loaded (/etc/rc.d/init.d/openvswitch; bad; vendor preset: disabled)
   Active: active (running) since 五 2017-02-10 20:46:46 CST; 4min 12s ago
     Docs: man:systemd-sysv-generator(8)
  Process: 13158 ExecStart=/etc/rc.d/init.d/openvswitch start (code=exited, status=0/SUCCESS)
   CGroup: /system.slice/openvswitch.service
           ├─13182 ovsdb-server: monitoring pid 13183 (healthy)
           ├─13183 ovsdb-server /etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=...
           ├─13195 ovs-vswitchd: monitoring pid 13196 (healthy)
           └─13196 ovs-vswitchd unix:/var/run/openvswitch/db.sock -vconsole:emer -vsyslog:err -vfile:info ...

2月 10 20:46:46 localhost systemd[1]: Starting LSB: Open vSwitch switch...
2月 10 20:46:46 localhost openvswitch[13158]: /etc/openvswitch/conf.db does not exist ... (warning).
2月 10 20:46:46 localhost openvswitch[13158]: Creating empty database /etc/openvswitch/conf.db [  OK  ]
2月 10 20:46:46 localhost openvswitch[13158]: Starting ovsdb-server [  OK  ]
2月 10 20:46:46 localhost openvswitch[13158]: Configuring Open vSwitch system IDs [  OK  ]
2月 10 20:46:46 localhost openvswitch[13158]: Inserting openvswitch module [  OK  ]
2月 10 20:46:46 localhost openvswitch[13158]: Starting ovs-vswitchd [  OK  ]
2月 10 20:46:46 localhost openvswitch[13158]: Enabling remote OVSDB managers [  OK  ]
2月 10 20:46:46 localhost systemd[1]: Started LSB: Open vSwitch switch.
[root@localhost ~]#
```

8）安装pipework过程略，参考前面的操作

9）创建交换机，把物理网卡加入ovs1

```
[root@localhost ~]# ovs-vsctl add-br ovs1
[root@localhost ~]# ovs-vsctl add-port ovs1 eno16777736
[root@localhost ~]# ip link set ovs1 up
[root@localhost ~]# ifconfig eno16777736 0
[root@localhost ~]# ifconfig ovs1 192.168.1.100
[root@localhost ~]# ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 0.0.0.0
        ether 02:42:9c:a5:dc:47  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eno16777736: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 1000  (Ethernet)
        RX packets 1208  bytes 156626 (152.9 KiB)
        RX errors 0  dropped 4  overruns 0  frame 0
        TX packets 166  bytes 16942 (16.5 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 4  bytes 340 (340.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 340 (340.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ovs1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.100  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::20c:29ff:fee7:13d3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:e7:13:d3  txqueuelen 0  (Ethernet)
        RX packets 403  bytes 96726 (94.4 KiB)
        RX errors 0  dropped 221  overruns 0  frame 0
        TX packets 25  bytes 3713 (3.6 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

10）在主机A上创建4个Docker容器，test1、test2、test3、test4

```
[root@localhost ~]# docker run -dit --name=test1 docker.io/centos:latest
afd242ab4d926940e647bc23e4b8fd67be381e9b5ccb154342d6d9ca5cd9d459
[root@localhost ~]# docker run -dit --name=test2 docker.io/centos:latest
ae4448f6bb2c6d3aac95305e3bad673d8922cc1278799b0c850422758d311c41
[root@localhost ~]# docker run -dit --name=test3 docker.io/centos:latest
e0b711b2f0dcf65703517eff4362095d44b8af80bea51a5ae2120f637a8afb70
[root@localhost ~]# docker run -dit --name=test4 docker.io/centos:latest
c441fde01dbcf66020fe0c41443cc35c9efd12166b93f65fb6da605a44c5852b
[root@localhost ~]#
```

11）将test1，test2划分到一个vlan中，vlan在mac地址后加@指定，此处mac地址省略。

```
[root@localhost ~]# pipework ovs1 test1 192.168.1.110/24@192.168.1.1 @100
[root@localhost ~]# pipework ovs1 test2 192.168.1.120/24@192.168.1.1 @100
[root@localhost ~]# pipework ovs1 test3 192.168.1.130/24@192.168.1.1 @200
[root@localhost ~]# pipework ovs1 test4 192.168.1.140/24@192.168.1.1 @200
[root@localhost ~]#
```

12）完成上述操作后，使用docker attach连到容器中，然后用ping命令测试连通性，发现test1和test2可以相互通信，但与test3和test4隔离。这样，一个简单的VLAN隔离容器网络就已经完成。

```
[root@localhost ~]# docker attach test1
[root@afd242ab4d92 /]# ping 192.168.1.120
PING 192.168.1.120 (192.168.1.120) 56(84) bytes of data.
64 bytes from 192.168.1.120: icmp_seq=1 ttl=64 time=0.317 ms
64 bytes from 192.168.1.120: icmp_seq=2 ttl=64 time=0.043 ms
^C
--- 192.168.1.120 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.043/0.180/0.317/0.137 ms
[root@afd242ab4d92 /]# ping 192.168.1.130
PING 192.168.1.130 (192.168.1.130) 56(84) bytes of data.
From 192.168.1.110 icmp_seq=1 Destination Host Unreachable
From 192.168.1.110 icmp_seq=2 Destination Host Unreachable
From 192.168.1.110 icmp_seq=3 Destination Host Unreachable
From 192.168.1.110 icmp_seq=4 Destination Host Unreachable
^C
--- 192.168.1.130 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3003ms
pipe 4
[root@afd242ab4d92 /]# ping 192.168.1.140
PING 192.168.1.140 (192.168.1.140) 56(84) bytes of data.
From 192.168.1.110 icmp_seq=1 Destination Host Unreachable
From 192.168.1.110 icmp_seq=2 Destination Host Unreachable
From 192.168.1.110 icmp_seq=3 Destination Host Unreachable
From 192.168.1.110 icmp_seq=4 Destination Host Unreachable
^C^C
--- 192.168.1.140 ping statistics ---
6 packets transmitted, 0 received, +4 errors, 100% packet loss, time 5006ms
pipe 4
[root@afd242ab4d92 /]#
```

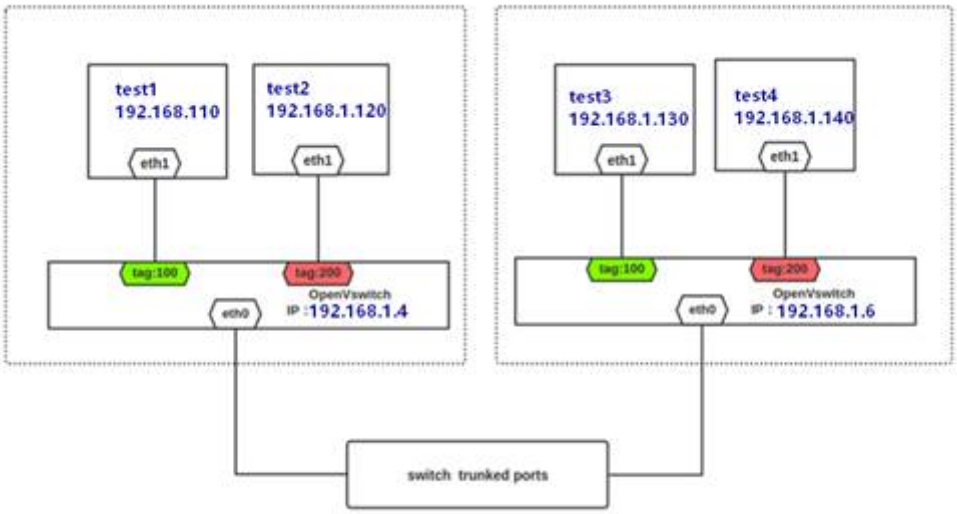
```
[root@localhost ~]# docker attach test4
[root@c441fde01dbc /]# ping 192.168.1.110
PING 192.168.1.110 (192.168.1.110) 56(84) bytes of data.
From 192.168.1.140 icmp_seq=1 Destination Host Unreachable
From 192.168.1.140 icmp_seq=2 Destination Host Unreachable
From 192.168.1.140 icmp_seq=3 Destination Host Unreachable
From 192.168.1.140 icmp_seq=4 Destination Host Unreachable
^C
--- 192.168.1.110 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3002ms
pipe 4
[root@c441fde01dbc /]# ping 192.168.1.120
PING 192.168.1.120 (192.168.1.120) 56(84) bytes of data.
From 192.168.1.140 icmp_seq=1 Destination Host Unreachable
From 192.168.1.140 icmp_seq=2 Destination Host Unreachable
From 192.168.1.140 icmp_seq=3 Destination Host Unreachable
From 192.168.1.140 icmp_seq=4 Destination Host Unreachable
^C
--- 192.168.1.120 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 3000ms
pipe 4
[root@c441fde01dbc /]# ping 192.168.1.130
PING 192.168.1.130 (192.168.1.130) 56(84) bytes of data.
64 bytes from 192.168.1.130: icmp_seq=1 ttl=64 time=0.386 ms
64 bytes from 192.168.1.130: icmp_seq=2 ttl=64 time=0.047 ms
64 bytes from 192.168.1.130: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 192.168.1.130: icmp_seq=4 ttl=64 time=0.067 ms
^C
--- 192.168.1.130 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 0.041/0.135/0.386/0.145 ms
[root@c441fde01dbc /]#
```

注：由于OpenVswitch本身支持VLAN功能，所以这里pipework所做的工作和之前介绍的基本一样，只不过将Linux bridge替换成了OpenVswitch，在将veth pair的一端加入ovs0网桥时，指定了tag。底层操作如下：

```
ovs-vsctl add-port ovs0 veth* tag=100
```

三、多主机Docker容器的VLAN划分

上面介绍完了单主机上VLAN的隔离，下面我们将情况延伸到多主机的情况。有了前面两个例子做铺垫，这个也就不难了。为了实现这个目的，我们把宿主机上的网卡桥接到各自的OVS网桥上，然后再为容器配置IP和VLAN就可以了。我们实验环境如下，主机A和B各有一块网卡eno16777736，IP地址分别为192.168.1.4/24、192.168.1.6/24。在主机A上创建两个容器test1、test2，分别在VLAN100和VLAN 200上。在主机B上创建test3、test4，分别在VLAN100和VLAN 200 上。最终，test1可以和test3通信，test2可以和test4通信



1)在主机A上

创建Docker容器

```
docker run -dit --name test1 docker.io.centos:latest
docker run -dit --name test2 docker.io.centos:latest
```

划分VLAN

```
pipework br0 test1 192.168.110/24@192.168.1.1  @100
pipework br0 test2 192.168.120/24@192.168.1.1  @200
```

将eno16777736桥接到br0上

```
ip addr add 192.168.1.4/24 dev br0
ip addr del 192.168.1.4/24 dev eno16777736
ovs-vsctl add-port br0 eno16777736
ip route del default
ip route add default gw 192.168.1.1 dev br0
```

2)在主机B上

创建Docker容器

```
docker run -dit --name test3 docker.io.centos:latest
docker run -dit --name test4 docker.io.centos:latest
```

划分VLAN

```
pipework br0 test3 192.168.1.130/24@192.168.1.1  @100
pipework br0 test4 192.168.1.140/24@192.168.1.1  @200
```

将eno16777736桥接到br0上

```
ip addr add 192.168.1.6/24 dev br0
ip addr del 192.168.1.6/24 dev eno16777736
ovs-vsctl add-port br0 eno16777736
ip route del default
ip route add default gw 192.168.1.1 dev br0
```

完成上面的步骤后，主机A上的test1和主机B上的test3容器就划分到了一个VLAN中，并且与主机A上的test2和主机B上的test4隔离（主机eno16777736网卡需要设置为混杂模式，连接主机的交换机端口应设置为trunk模式，即允许VLAN 100和VLAN200的包通过）。

注：除此之外，pipework还支持使用macvlan设备、设置网卡MAC地址等功能。不过，pipework有一个缺陷，就是配置的容器在关掉重启后，之前的设置会丢失。

```
[root@localhost Desktop]# ifconfig enp0s3
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::a00:27ff:fe79:4f95  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:79:4f:95  txqueuelen 1000  (Ethernet)
        RX packets 14173  bytes 4674203 (4.4 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 217  bytes 20212 (19.7 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

其中promisc表示网卡混杂模式

其他参数的含义：

UP：表示网卡开启状态；

BROADCAST：表示支持广播；

promisc：表示网卡混杂模式；

RUNNING：表示网卡的网线被接上；

MULTICAST：表示支持组播；

MTU：表示MaximumTrasmission Unit 最大传输单元(字节)，即此接口一次所能传输的最大封包；

RX：表示网络由激活到目前为止接收的数据包；

TX：表示网络由激活到目前为止发送的数据包；

collisions：表示网络信号冲突的情况；

txqueuelen：表示传输缓冲区长度大小；

设置网卡工作模式

#ifconfig 网卡名 promisc 设置混杂

#ifconfig 网卡名 -promisc 取消混杂

网卡工作模式有4种，分别是：
广播(Broadcast)模式
多播(Multicast)模式
单播模式（Unicast）
混杂模式（Promiscuous）
在混杂模式下的网卡能够接收一切通过它的数据，而不管该数据目的地址是否是它。如果通过程序将网卡的工作模式设置为 “混杂模式”，那么网卡将接受所有流经它的数据帧，这实际上就是Sniffer工作的基本原理：让网卡接收一切他所能接收的数据。Sniffer就是一种能将本地网卡状态设成混杂（promiscuous）状态的软件，当网卡处于这种"混杂"方式时，它对所有遇到的每一个数据帧都 产生一个硬件中断以便提醒操作系统处理流经该物理媒体上的每一个报文包。可见，Sniffer工作在网络环境中的底层，它会拦截所有的正在网络上传送的数据，并且通过相应的软件处理，可以实时分析这些数据的内容，进而分析所处的网络状态和整体布局。

扩展：利用Weave实现跨主机容器互联