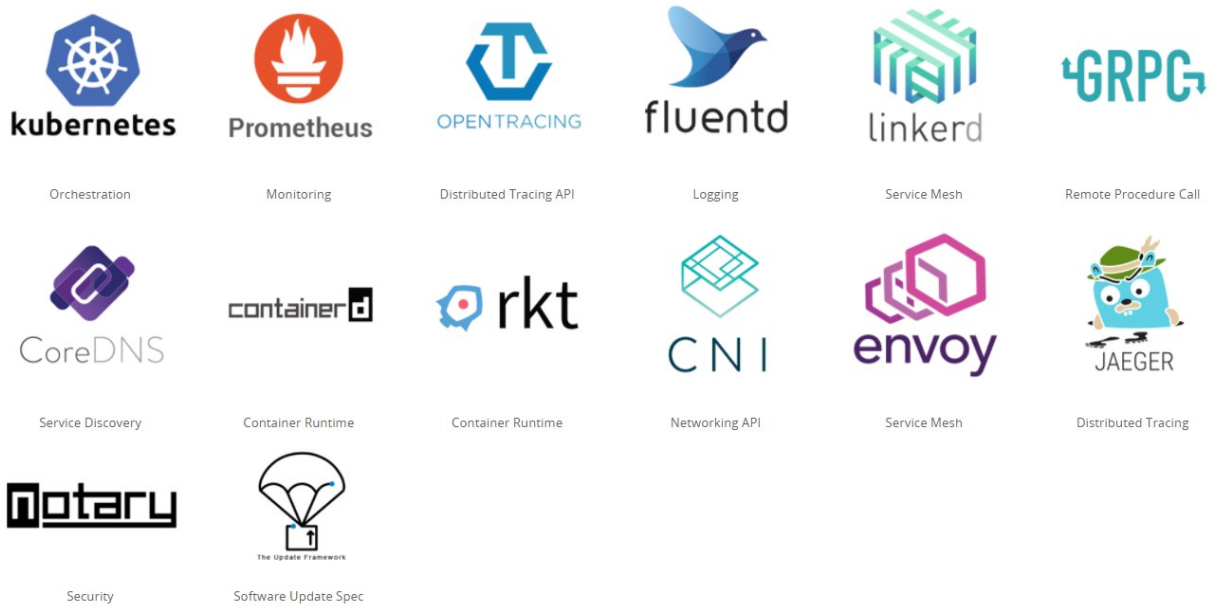
第一章 发展历史

关注几个机构和平台：

几大云厂家的容器编排工具：
GCE（Google）
ECS（AWS）
DDC（Docker退出的商业化容器管理平台）

CNCF（Cloud Native Computing Foundation）：K8S属于这个基金会，除此之外该基金会下面的几个监控，日志相关的项目页值得关注。



OCI（Open Container Initative）：关于容器标准化组织。

第二章 Docker基础

1. Linux cgroups 和 namespace功能

cgroups主要用于限制进程的资源使用率（这里的资源包括：CPU、内存、IO等等）
参考：
http://blog.csdn.net/liumiaocn/article/details/52589880
https://www.ibm.com/developerworks/cn/linux/1506_cgroup/index.html

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

namespace是Linux的一个内特特性，可以隔离文件系统、网络、PID等等资源。说白了应该是Docker的底层实现基础。网上的资料比较多，属于一个比较完善的特性。

---

2. Boot2Docker

Windows环境和OS X环境需要的工具。

---

3. 关于命令

表2-1　Docker子命令分类

子命令分类	子　命　令
Docker环境信息	info、version
容器生命周期管理	Create、exec、kill、pause、restart、rm、run、start、stop、unpause
镜像仓库命令	login、logout、pull、push、search
镜像管理	build、images、import、load、rmi、save、tag、commit
容器运维操作	attach、export、inspect、port、ps、rename、stats、top、wait、cp、diff、update
容器资源管理	volume、network
系统日志信息	events、history、logs

---

4. 关于Docker的一个Demo

使用django、redis、haproxy创建了一个关于docker应用栈！
可以熟悉一个这三个工具。

第三章 核心原理

Docker 容器的本质是宿主机上的进程。Docker通过namespace实现了资源隔离，通过cgroups实现了资源限制，通过copy-on-write实现了高效的文件操作。

---

1. namespace 资源隔离

表3-1　namespace的6项隔离

namespace	系统调用参数	隔离内容
UTS	CLONE_NEWUTS	主机名与域名
IPC	CLONE_NEWIPC	信号量、消息队列和共享内存
PID	CLONE_NEWPID	进程编号
Network	CLONE_NEWNET	网络设备、网络栈、端口等
Mount	CLONE_NEWNS	挂载点（文件系统）
User	CLONE_NEWUSER	用户和用户组

使用*namespace API*，书中提到了四种方式，值得注意的是namespace在Linux上的表现实质上是一个文件描述符！！：

1. 通过**clone()**创建新进程，同时创建新的namespace；
2. 通过**/proc/< pid >/ns**中的软连接绑定某个ns，使后续的进程能够加入（这个目录下面软连接展示了6个namespace的ID，只要这些软连接的文件描述符被打开，或者被mount，对应的ns都不会消亡）；
3. 通过**setenv()**调用将进程加入某个ns（注意pid例外，原进程的pid依然在旧的ns中，之后创建的子进程的pid加入该ns）；
4. 通过unshare()在原先进程上实现隔离（注意pid例外，原进程的pid依然在旧的ns中，之后创建的子进程的pid加入该ns）；

关于*PID namespace*

1. root namespace可以看到所有ns中的内容；
2. 第一个启动的进程在新的ns，对应的PID为1，相当于init进程；
3. 容器中第一个进程被销毁，相当于init被销毁，所有其他进程被结束，ns被收回，容器退出；

关于*mount namespace*

1. 容器内的所有挂载，以及挂载状态：/proc/< pid >/mounts、/proc/< pid >/mountstats；
2. 关于挂载传播（mount propagation）；

关于*user namespace*

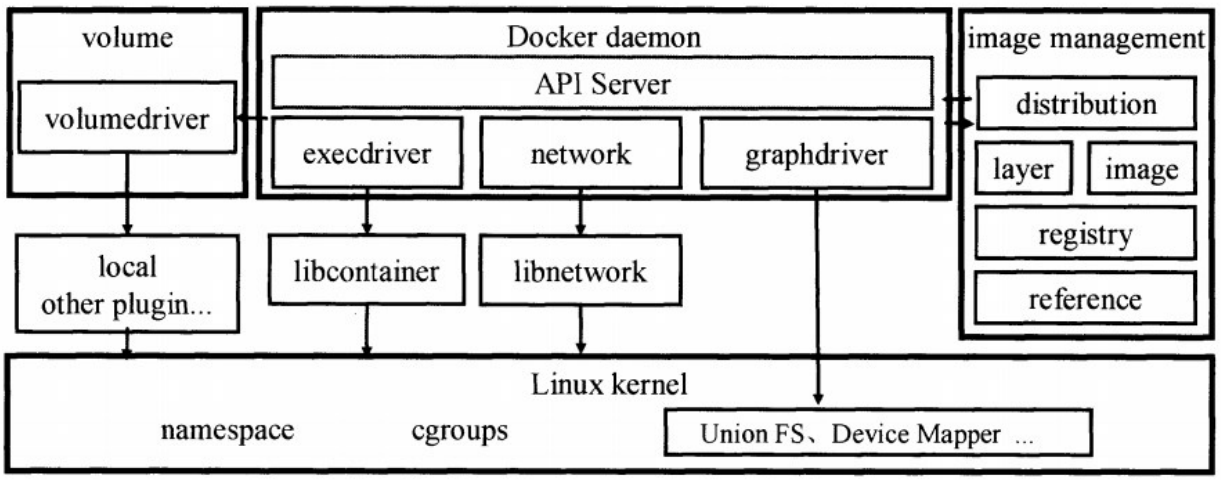
1. 第一个进程有该namespace中的全部权限；
2. 如何建立内部user和外部user的映射（/proc/[pid]/uid_map和/proc/[pid]/gid_map）；

2. cgroups文件系统

略

---

3. Docker的总体框架



docker daemon下面涉及的模块：

1. 镜像管理：包含distribution、registry、layer、image、reference；
2. execdriver：主要包含操作系统资源的隔离，进行一些有关namespace、cgroups、以及其他的相关操作。最主要的实现是libcontainer
3. volumedriver：docker volume的实现，默认的是local；
4. graphdriver：镜像层和容器层相关的读写，由该模块负责；
5. network：libnetwork完成 ------ CNM模型；

***docker daemon*启动过程**：

1. 初始化server模块，包括：创建sock、开启远程访问端口；
2. 创建daemon 对象（NewDaemon）；
3. 绑定server api和daemon对象；

***docker daemon*文件环境**：
容器配置文件目录/var/lib/docker/containers/[containers id]，该目录下主要是容器的配置文件。

```
├── containers
│   ├── [ container-id ]
```

```
3  │   │   ├── config.v2.json
4  │   │   ├── hostconfig.json
5  │   │   ├── hostname
6  │   │   ├── hosts
7  │   │   ├── resolv.conf
8  │   │   ├── resolv.conf.hash
9  │   │   ├── secrets
10 │   │   └── shm
11 │   ├── ......
```

config.v2.json中包含了容器的所有配置，但是该目录下的文件不能修改（修改之后会被daemon进程重置！）。

graphdriver目录，/var/lib/docker/[driver type]。使用devicemapper时目录结构如下：

```
1  ├── devicemapper
2  │   ├── devicemapper # 使用loop-lvm该目录下的两个文件，相当于块设备。如果用driect-l
3  │   │   ├── data
4  │   │   └── metadata
5  │   ├── metadata # 官方解释这个目录包含：devicemapper、images、layer的信息（json格
6  │   │   ├── base
7  │   │   ├── deviceset-metadata
8  │   │   ├── transaction-metadata
9  │   │   ├── [ layer id ?]
10 │   │   └── ......
11 │   └── mnt # 通过inspect命令可以查到对GraphDriver中包含对应的容器层ID，并且还能通过
12 │       ├── [ DeviceName id ?]
13 │       ├── [ DeviceName id ?]
14 │       ├── ......
```

和graphdriver有关的代码在daemon/graphdriver

镜像目录：/var/lib/docker/image，镜像目录的架构和用什么graphdriver有关。

```
1  ├── image
2  │   └── devicemapper  # 存储的是元数据
3  │       ├── distribution # 没搞懂是用来干啥的？？？
```

```
4  |       ├── imagedb  # 镜像的元数据
5  |       |   ├── content/sha256/[image-id]  # 镜像的具体信息（json）
6  |       |   ├──metadata/sha256/[image-id]  # 具体什么内容？？
7  |       ├── layerdb    # 镜像层/容器层 元数据存储目录
8  |       |   ├── mounts # 对应容器层
9  |       |   |   ├── [container-id]
10 |       |   |   |   ├── init-id # devicemapper时，对应的是容器设备（df -h可以看到,/de
11 |       |   |   |   ├── mount-id # devicemapper时和init-id内容一致
12 |       |   |   |   └── parent # 内容为：sha256:layer_id，在/var/lib/docker/image/d
13 |       |   |   ├── ......
14 |       |   ├── sha256 # 这个好像是镜像层
15 |       |   |   ├── [ layer-id ]
16 |       |   |   |   ├── cache-id  # 这个对应到devicemapper目录的快照ID，在/var/lib/d
17 |       |   |   |   ├── diff
18 |       |   |   |   ├── parent # 上一层layer-id，如果没有上层就没有parent文件
19 |       |   |   |   ├── size
20 |       |   |   |   └── tar-split.json.gz
21 |       |   |   ├── ......
22 |       |   └── tmp
23 |       └── repositories.json  # 包含镜像名、镜像sha256、镜像ID之间的映射关系
```

volume目录/var/lib/docker/volumes

---

docker client：client模式下，主要的工作是解析用户命令，并向指定的daemon进程POST指定的消息。需要注意的有以下几点：

1. 通过--host可以指定要连接的Docker deamon位置，如果系统的环境变量DOCKER_HOST不为空，说明用户指定了HOST。默认时，指定的是unix:///var/run/docker.sock，通过添加启动项“-H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375”，可以同时开启本地监听和远程监听。
2. 新版本docker的代码中，每一个命令对应一个go文件。这些文件在cli/command目录下，并且根据container，image等类型分类。
   如docker ps对应的go文件是：
   cli/command/container/list.go（并不是ps.go文件）

---

4. libcontainer

1. libcontainer对Docker容器做了一层更高级的抽象，他定义了Process和Container来对应Linux中“进程”和“容器”的关系。
2. 在Docker中，Container是一个平台无关的抽象概念，而在libcontainer中本质上是一组位于独立命名空间的、平台相关的进程。
3. 在容器初始化完成之前，在Docker Daemon进程和Container的init进程（namespace中的第一个进程）通过管道通信。初始化完成之后管道随即关闭。
4. Docker Deamon通过libcontainer完成：容器的创建和初始化、生命周期管理、进程管理。

---

5. 镜像技术

几个概念：

1. 镜像用到的关键技术包括：分层（联合挂载或者快照）、COW、内容寻址（主要是每个layer层使用内容的sha256作为唯一标识）
2. diff-id和chain-id如何计算
    1. 对于最底层（没有父层），diff-id=chain-id
    2. Image的最底层chain-id（layer-id）记录在var/lib/docker/image/devicemapper/imagedb/content/sha256/[image-id]文件的rootfs字段中。
    3. chain-id=sha256（chain-id(n-1) diff-id(n)），该层的diff-id通过该层的文件包计算用sha256计算出来。（layer/layer.go:createChainIDFromParent接口）
3. 关于几种存储的文件组织形式可以参考附件

---

6. volume

1. 共享挂载volume，使用--volumes-from标签。（下面的例子创建了一个匿名volume，挂载在/data目录，之后共享给其他两个容器）

```
$ sudo docker run --name vol_data -v /data ubuntu echo "This is a data-only container"
$ sudo docker run -it --name vol_share1 --volumes-from vol_data ubuntu /bin/bash
$ sudo docker run -it --name vol_share2 --volumes-from vol_data ubuntu /bin/bash
```

2. 同步删除匿名volume

   docker rm -v

3. 简易备份方法&恢复方法：

```
$ sudo docker run --rm --volumes-from vol_simple -v $(pwd):/backup ubuntu tar cvf /backup/data.tar /data
```

```
$ sudo docker run -it --name vol_bck -v /data ubuntu /bin/bash
$ sudo docker run --rm --volumes-from vol_bck -v $(pwd):/backup ubuntu tar xvf /backup/data.tar -C /
```

4. local-volume的原理，本质上是使用了linux的mount -bind进行绑定挂载（可以绑定文件或者目录）

---

7. 网络原理

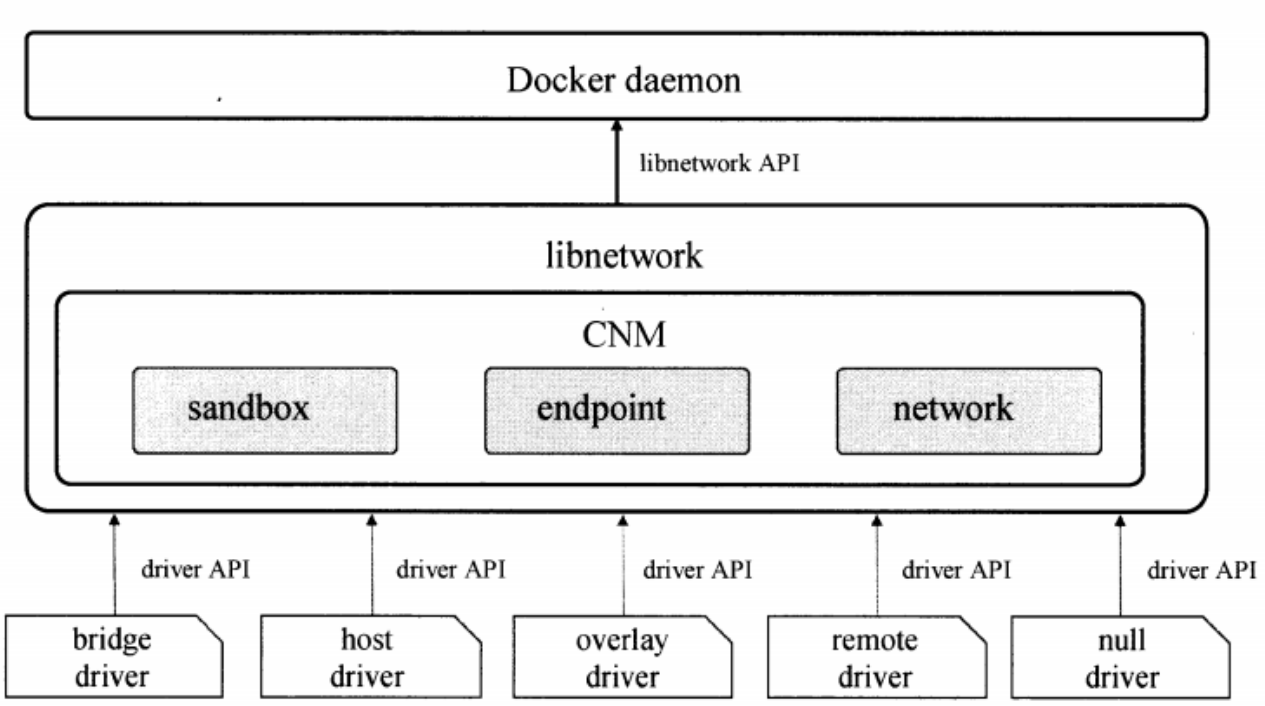


图3-16　Docker网络虚拟化架构

一个网络拓扑的案例

拓扑结构如下：

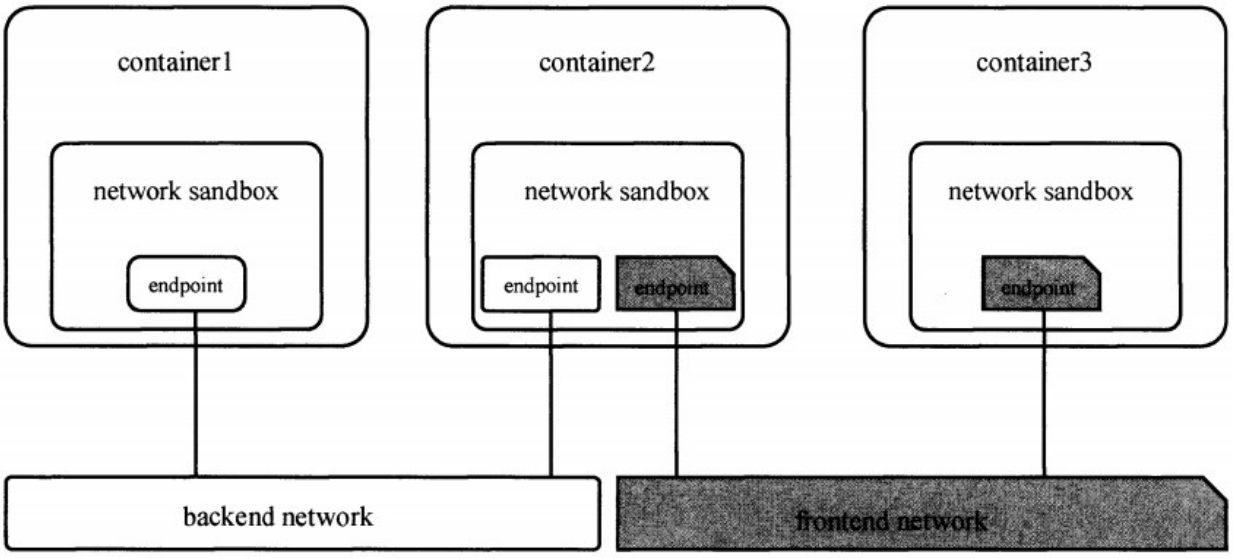


图3-17　CNM主要组件示例图[①]

```
docker network create backend # 这里每创建一个network，在宿主机上就多了一个网桥。
docker network create frontend
docker network ls # 可以看到三个默认的网络，包括： bridge，host，none
docker run -itd --name container1 --net backend busybox
docker run -itd --name container2 --net backend busybox
docker run -itd --name container3 --net frontend busybox

docker network connect frontend container2 # 执行完这个命令之后，container2中多了
```

关于bridge驱动：

1. 默认情形下，容器的默认网关指向宿主机中的docker0这个设备。并且宿主机上会出现一个veth设备。容器中发送的数据，实际上在二层上都被发送到了对应veth设备中，该设备又发送到了docker0这个网桥中。

```
27: vethc316da3@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
    link/ether 96:df:87:3e:e7:1d brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::94df:87ff:fe3e:e71d/64 scope link
       valid_lft forever preferred_lft forever
```

2. iptable做的工作：
    1. 源地址转换，使来自容器的IP包看起来像宿主机发出的（SNAT）：
       `-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE`
    2. 端口映射，利用IPTABLE将容器的端口和主机端口联通。发到主机该端口上的IP包，都会被转发到容器IP地址的指定端口（DNAT)。
    3. 容器通信，要求：`-A FORWARD -i docker0 -o docker0 -j ACCEPT`，并打开Linux的网卡转发功能。

使用ip命令创建net namespace：

```
# 下面的例子，通过veth-pair 在两个net-namespace之间，创建了一个点对点的网络拓扑。

ip netns add nstest # 创建名为nstest的网络空间
ip netns list # 打印当前所有的网络空间
ip netns delete nstest # 删除名为nstest的网络空间
ip netns exec nstest ip link set dev lo up # 启动nstest默认创建的lo网卡
ip link add veth-a type veth peer name veth-b # 创建veth-pair
ip link set veth-b netns nstest # 将veth-b放到nstest空间
ip addr add 10.0.0.1/24 dev veth-a #配置veth-a、veth-b的ip地址，并且启动
ip link set dev veth-a up
ip netns exec nstest ip addr add 10.0.0.2/24 dev veth-b
ip netns exec nstest ip link set dev veth-b up

```

通过上面的配置后，两个namespace之间，可以互相ping同veth的地址。ip命令会默认生成对应的route信息。

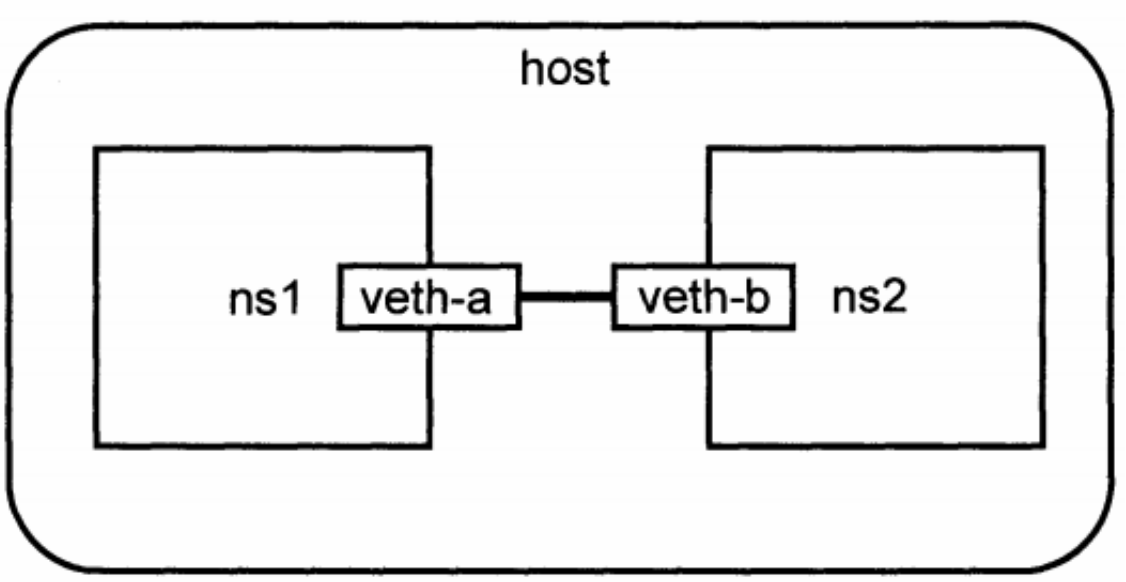
新空间中的：

```
[root@vmdocker17 ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 veth-b
```

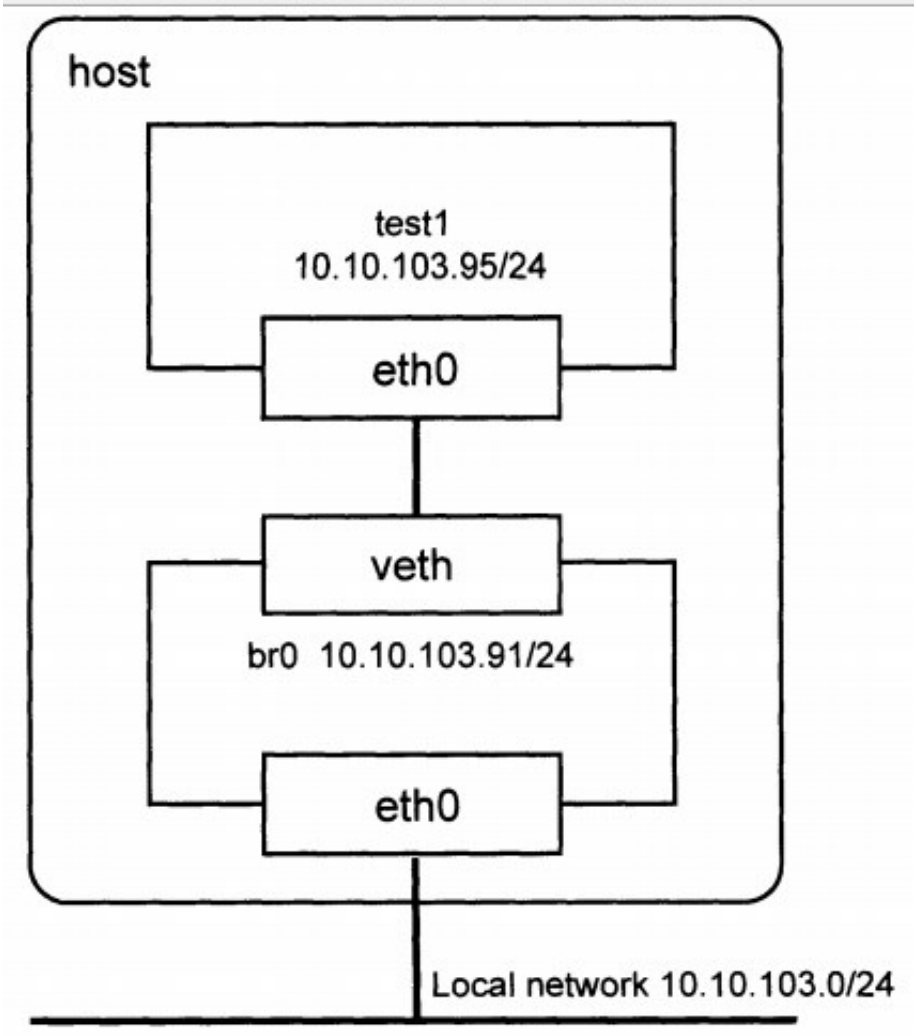
默认空间中的：

```
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         172.24.10.1     0.0.0.0         UG    100    0        0 eth0
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 veth-a
169.254.169.254 172.24.10.2     255.255.255.255 UGH   100    0        0 eth0
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
172.24.10.0     0.0.0.0         255.255.255.0   U     100    0        0 eth0
```

网络拓扑为：



如果想要让nstest空间能够访问其他网络，那么需要借助网桥，下面的例子展示了一个简单的拓扑关系，这里要求网桥ip和物理网卡ip一致，且nstest空间的网卡IP和物理卡在同一个网段：



使用ip netns命令配置容器的网络，通过工具nsenter也能实现同样的效果。有一点需要注意的是，由于ip netns命令只能管理/var/run/netns目录下的netns，因此需要将容器的网络ns软连接到该目录下。

```
mkdir -p /var/run/netns
ln -s /proc/$pid/ns/net /var/run/netns/$pid/ns/net
```