Introducing Linux Network Namespaces

Wednesday, September 4, 2013 in [Linux](http://blog.scottlowe.org/category/linux/), [Networking](http://blog.scottlowe.org/category/networking/) by [slowe](http://blog.scottlowe.org/author/slowe/) | [54 comments](http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/#comments)

In this post, I’m going to introduce you to the concept of Linux network namespaces. While it might seem a bit esoteric right now, trust me that there is a reason why I’m introducing you to network namespaces—if you, like me, are on a journey to better understand OpenStack, you’ll almost certainly run into network namespaces again.

So what are network namespaces? Generally speaking, an installation of Linux shares a single set of network interfaces and routing table entries. You can modify the routing table entries using policy routing ([here’s](http://blog.scottlowe.org/2013/05/29/a-quick-introduction-to-linux-policy-routing/) an introduction I wrote and [here’s](http://blog.scottlowe.org/2013/05/30/a-use-case-for-policy-routing-with-kvm-and-open-vswitch/) a write-up on a potential use case for policy routing), but that doesn’t fundamentally change the fact that the set of network interfaces and routing tables/entries are shared across the entire OS. Network namespaces change that fundamental assumption. With network namespaces, you can have different and separate instances of network interfaces and routing tables that operate independent of each other.

This concept is probably best illustrated through some examples. Along the way, I’ll introduce a few new ideas as well. First, though, I need to provide some assumptions.

## Assumptions

Throughout these examples, I’m using Ubuntu Server 12.04.3 LTS. Please note that support for network namespaces varies between Linux distributions; Ubuntu supports them but Red Hat doesn’t. (I’m not sure about Fedora. If you know, speak up in the comments.) If you’re thinking about using network namespaces, be sure your Linux distribution includes support.

Further, I’ll assume that you’re either running as root, or that you will prepend sudo to the commands listed here as necessary.

## Creating and Listing Network Namespaces

Creating a network namespace is actually quite easy. Just use this command:

ip netns add <new namespace name>

For example, let’s say you wanted to create a namespace called “blue”. You’d use this command:

ip netns add blue

To verify that the network namespace has been created, use this command:

ip netns list

You should see your network namespace listed there, ready for you to use.

## Assigning Interfaces to Network Namespaces

Creating the network namespace is only the beginning; the next part is to assign interfaces to the namespaces, and then configure those interfaces for network connectivity. One thing that threw me off early in my exploration of network namespaces was that you couldn’t assign physical interfaces to a namespace. How in the world were you supposed to use them, then?

It turns out you can only assign virtual Ethernet (veth) interfaces to a network namespace. Virtual Ethernet interfaces are an interesting construct; they always come in pairs, and they are connected like a tube—whatever comes in one veth interface will come out the other peer veth interface. As a result, you can use veth interfaces to connect a network namespace to the outside world via the “default” or “global” namespace where physical interfaces exist.

Let’s see how that’s done. First, you’d create the veth pair:

ip link add veth0 type veth peer name veth1

I found a few sites that repeated this command to create veth1 and link it to veth0, but my tests showed that both interfaces were created and linked automatically using this command listed above. Naturally, you could substitute other names for veth0 and veth1, if you wanted.

You can verify that the veth pair was created using this command:

ip link list

You should see a pair of veth interfaces (using the names you assigned in the command above) listed there. Right now, they both belong to the “default” or “global” namespace, along with the physical interfaces.

Let’s say that you want to connect the global namespace to the blue namespace. To do that, you’ll need to move one of the veth interfaces to the blue namespace using this command:

ip link set veth1 netns blue

If you then run the ip link list command again, you’ll see that the veth1 interface has disappeared from the list. It’s now in the blue namespace, so to see it you’d need to run this command:

ip netns exec blue ip link list

Whoa! That’s a bit of a complicated command. Let’s break it down:

* The first part, ip netns exec, is how you execute commands in a different network namespace.
* Next is the specific namespace in which the command should be run (in this case, the blue namespace).
* Finally, you have the actual command to be executed in the remote namespace. In this case, you want to see the interfaces in the blue namespace, so you run ip link list.

When you run that command, you should see a loopback interface and the veth1 interface you moved over earlier.

## Configuring Interfaces in Network Namespaces

Now that veth1 has been moved to the blue namespace, we need to actually configure that interface. Once again, we’ll use the ip netns exec command, this time to configure the veth1 interface in the blue namespace:

ip netns exec blue ifconfig veth1 10.1.1.1/24 up

As before, the format this command follows is:

ip netns exec <network namespace> <command to run against that namespace>

In this case, you’re using ifconfig to assign an IP address to the veth1 interface and bring that interface up. (Note: you could use the ip addr, ip route, and ip link commands to accomplish the same thing.)

Once the veth1 interface is up, you can verify that the network configuration of the blue namespace is completely separate by just using a few different commands. For example, let’s assume that your “global” namespace has physical interfaces in the 172.16.1.0/24 range, and your veth1 interface is in a separate namespace and assigned something from the 10.1.1.0/24 range. You could verify how network namespaces keep the network configuration separate using these commands:

* ip addr list in the global namespace will not show any 10.1.1.0/24-related interfaces or addresses.
* ip netns exec blue ip addr list will show only the 10.1.1.0/24-related interfaces and addresses, and will not show any interfaces or addresses from the global namespace.
* Similarly, ip route list in each namespace will show different routing table entries, including different default gateways.

## Connecting Network Namespaces to the Physical Network

This part of it threw me for a while. I can’t really explain why, but it did. Once I’d figured it out, it was obvious. To connect a network namespace to the physical network, just use a bridge. In my case, I used an Open vSwitch (OVS) bridge, but a standard Linux bridge would work as well. Place one or more physical interfaces as well as one of the veth interfaces in the bridge, and—bam!—there you go. Naturally, if you had different namespaces, you’d probably want/need to connect them to different physical networks or different VLANs on the physical network.

So there you go—an introduction to Linux network namespaces. It’s quite likely I’ll build on this content later, so while it seems a bit obscure right now just hang on to this knowledge. In the meantime, if you have questions, clarifications, or other information worth sharing with other readers, please feel free to speak up in the comments.

[**Docker实践(4)—network namespace与veth pair**](http://www.cnblogs.com/hustcat/p/3928261.html)

**network namespace**

**创建network namespace**

# ip netns add blue

# ip netns list

blue

**添加网口到namespace**

先创建veth

# ip link add veth0 type veth peer name veth1

在当前namespace可以看到veth0和veth1

# ip link list

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

2: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UP qlen 1000

    link/ether 00:0c:29:b2:cf:72 brd ff:ff:ff:ff:ff:ff

3: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000

    link/ether ae:0d:00:e1:11:38 brd ff:ff:ff:ff:ff:ff

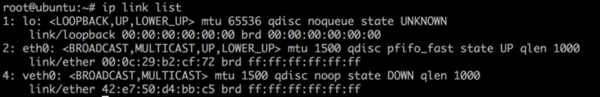
4: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000

    link/ether 42:e7:50:d4:bb:c5 brd ff:ff:ff:ff:ff:ff

将veth1加到namespace “blue”

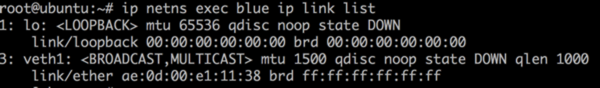
# ip link set veth1 netns blue

此时，当前namepapce只能看到veth0。



通过如下命令可以查看blue namespace的网口

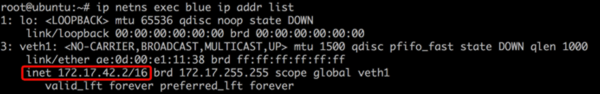
# ip netns exec blue ip link list



**配置network namespace的网口**

通过ip netns exec可以配置namespace的网口

# ip netns exec blue ifconfig veth1 172.17.42.2/16 up



**network namespace的网口与物理网卡的通信**

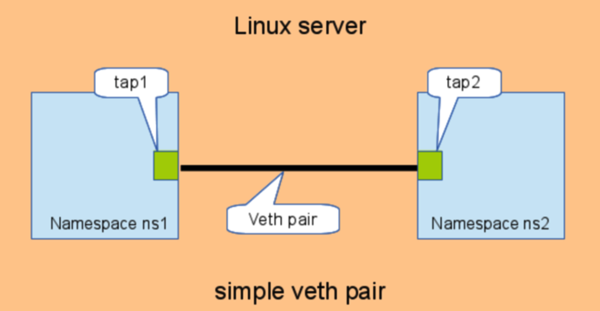
通过bridge来实现。参见veth pair一节。

主要参考

[0][Introducing Linux Network Namespaces](http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces/)

**veth pair**

veth pair是用于不同network namespace间进行通信的方式，veth pair将一个network namespace数据发往另一个network namespace的veth。如下：



# add the namespaces

ip netns add ns1

ip netns add ns2

# create the veth pair

ip link add tap1 type veth peer name tap2

# move the interfaces to the namespaces

ip link set tap1 netns ns1

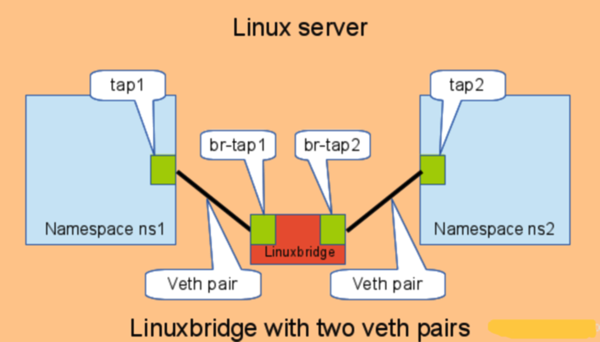
ip link set tap2 netns ns2

# bring up the links

ip netns exec ns1 ip link set dev tap1 up

ip netns exec ns2 ip link set dev tap2 up

如果多个network namespace需要进行通信，则需要借助bridge：



# add the namespaces

ip netns add ns1

ip netns add ns2

# create the switch

BRIDGE=br-test

brctl addbr $BRIDGE

brctl stp   $BRIDGE off

ip link set dev $BRIDGE up

#

#### PORT 1

# create a port pair

ip link add tap1 type veth peer name br-tap1

# attach one side to linuxbridge

brctl addif br-test br-tap1

# attach the other side to namespace

ip link set tap1 netns ns1

# set the ports to up

ip netns exec ns1 ip link set dev tap1 up

ip link set dev br-tap1 up

#

#### PORT 2

# create a port pair

ip link add tap2 type veth peer name br-tap2

# attach one side to linuxbridge

brctl addif br-test br-tap2

# attach the other side to namespace

ip link set tap2 netns ns2

# set the ports to up

ip netns exec ns2 ip link set dev tap2 up

ip link set dev br-tap2 up

#

内核实现

veth的实现与loopback interface类似，比较简单：

//drivers/net/veth.c

static netdev\_tx\_t veth\_xmit(struct sk\_buff \*skb, struct net\_device \*dev)

{

struct net\_device \*rcv = NULL;

struct veth\_priv \*priv, \*rcv\_priv;

priv = netdev\_priv(dev);

rcv = priv->peer;

rcv\_priv = netdev\_priv(rcv);

stats = this\_cpu\_ptr(priv->stats);

length = skb->len;

   //转发给peer

if (dev\_forward\_skb(rcv, skb) != NET\_RX\_SUCCESS)

goto rx\_drop;

**NETIF\_F\_NETNS\_LOCAL**

NETIF\_F\_NETNS\_LOCAL是网络设备的一个特性，设置该特性的网络设备，不允许在不同network namespace间移动。这类设备也叫做本地设备（local devices）。

Loopback，VXLAN，PPP，bridge都是这类设备。可以通过ethtool -k，或者ethtool –show- features查看该值：

# ethtool -k br0

netns-local: on [fixed]

如果对这类设备network namespace，会报下面的错误：

# ip link set br0 netns ns1

RTNETLINK answers: Invalid argument

参考《Resource management:Linux kernel Namespaces and cgroups》

主要参考

[0][Linux Switching – Interconnecting Namespaces](http://www.opencloudblog.com/?p=66)

Network Namespace可以实现网络的隔离，有点像路由器里的VRF。在虚拟化和LXC中有很重要的用处。

#### 创建Network Namespace

ip netns add

例如：

ip netns add test

查看namespace

ip netns list

#### 给Namespace添加接口

创建的Namespace不能添加真实的物理接口，只能添加虚拟接口veth（virtual Ethernet interface），它们经常成对出现并且像一个管道一样连在一起。

创建一对veth：veth0和veth1

ip link add veth0 type veth peer name veth1

通过命令可以查看我们创建的veth

[root@controller0 ~]*# ip link list*

1: lo: mtu 16436 qdisc noqueue state UNKNOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: eth0: mtu 1500 qdisc pfifo\_fast state UNKNOWN qlen 1000

link/ether 08:00:27:ec:3c:70 brd ff:ff:ff:ff:ff:ff

3: eth1: mtu 1500 qdisc pfifo\_fast state UNKNOWN qlen 1000

link/ether 08:00:27:d1:f2:b3 brd ff:ff:ff:ff:ff:ff

4: eth2: mtu 1500 qdisc pfifo\_fast state UNKNOWN qlen 1000

link/ether 08:00:27:ad:03:e8 brd ff:ff:ff:ff:ff:ff

5: eth3: mtu 1500 qdisc pfifo\_fast state UP qlen 1000

link/ether 08:00:27:b2:eb:13 brd ff:ff:ff:ff:ff:ff

6: virbr0: mtu 1500 qdisc noqueue state UNKNOWN

link/ether 52:54:00:eb:0e:7e brd ff:ff:ff:ff:ff:ff

7: virbr0-nic: mtu 1500 qdisc noop state DOWN qlen 500

link/ether 52:54:00:eb:0e:7e brd ff:ff:ff:ff:ff:ff

10: veth1: mtu 1500 qdisc noop state DOWN qlen 1000

link/ether 86:e4:2c:b1:77:d0 brd ff:ff:ff:ff:ff:ff

11: veth0: mtu 1500 qdisc noop state DOWN qlen 1000

link/ether 82:bf:54:c0:5c:a9 brd ff:ff:ff:ff:ff:ff

现在这两个veth都是属于默认（global）的Network Namespace，下面我们把veth0放到test的namespace里，veth1保留在global的namespace里。

[root@controller0 ~]*# ip link set veth0 netns test*

[root@controller0 ~]*# ip netns exec test ip a*

9: lo: mtu 16436 qdisc noop state DOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

11: veth0: mtu 1500 qdisc noop state DOWN qlen 1000

link/ether 82:bf:54:c0:5c:a9 brd ff:ff:ff:ff:ff:ff

发现veth0已经跑到test这个namespace里了，全局的network namespace里已没有了veth0.

目前veth0和veth1时down的状态，下面我们为两个veth对配置IP地址

ip netns **exec** test ip addr add 192.168.10.2/24 dev veth0

ip netns **exec** test ip link **set** veth0 up

[root@controller0 ~]*# ip netns exec test ip a*

9: lo: mtu 16436 qdisc noop state DOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

11: veth0: mtu 1500 qdisc pfifo\_fast state DOWN qlen 1000

link/ether 82:bf:54:c0:5c:a9 brd ff:ff:ff:ff:ff:ff

inet 192.168.10.2/24 scope global veth0

[root@controller0 ~]*#*

给veth1配置IP地址，veth1在global的Network Namespace里

ip addr add 192.168.10.1/24 dev veth1 up

[root@controller0 ~]*# ip a*

10: veth1: mtu 1500 qdisc pfifo\_fast state UP qlen 1000

link/ether 86:e4:2c:b1:77:d0 brd ff:ff:ff:ff:ff:ff

inet 192.168.10.1/24 scope global veth1

inet6 fe80::84e4:2cff:feb1:77d0/64 scope link

valid\_lft forever preferred\_lft forever

[root@controller0 ~]*# ip netns exec test ip a*

9: lo: mtu 16436 qdisc noop state DOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

11: veth0: mtu 1500 qdisc pfifo\_fast state UP qlen 1000

link/ether 82:bf:54:c0:5c:a9 brd ff:ff:ff:ff:ff:ff

inet 192.168.10.2/24 scope global veth0

inet6 fe80::80bf:54ff:fec0:5ca9/64 scope link

valid\_lft forever preferred\_lft forever

可以看到veth0和veth1都up了起来。验证一下连通性。

[root@controller0 ~]*# ping 192.168.10.2*

PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.

64 bytes from 192.168.10.2: icmp\_seq=1 ttl=64 time=0.084 ms

64 bytes from 192.168.10.2: icmp\_seq=2 ttl=64 time=0.102 ms

^C

--- 192.168.10.2 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1326ms

rtt min/avg/max/mdev = 0.084/0.093/0.102/0.009 ms

[root@controller0 ~]*# ip netns exec test ping 192.168.10.1*

PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.

64 bytes from 192.168.10.1: icmp\_seq=1 ttl=64 time=0.076 ms

64 bytes from 192.168.10.1: icmp\_seq=2 ttl=64 time=0.076 ms

^C

--- 192.168.10.1 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1552ms

rtt min/avg/max/mdev = 0.076/0.076/0.076/0.000 ms

[root@controller0 ~]*#*

从外往里ping和从里往外ping都是通的。

参考： https://github.com/yongluo2013/osf-openstack-training/blob/master/installation/How-to-connection-ns-outside.md

# 网络虚拟化技术（一）: Linux网络虚拟化

01 March 2013

笔者现在在做云计算的网络设计，涉及到上百台服务器与交换机，在实测前必须进行原型测试，但是我只有一个工作用开发机，本文介绍一种方法，使用这一台机器模拟数十台服务器以及路由器的网络环境。

Linux的网络虚拟化是LXC项目中的一个子项目，LXC包括文件系统虚拟化，进程空间虚拟化，用户虚拟化，网络虚拟化，等等 [[1](https://blog.kghost.info/2013/03/01/linux-network-emulator/#_footnote_1)]，这里使用LXC的网络虚拟化来模拟多个网络环境。

## 创建虚拟网络环境

使用命令

|  |
| --- |
| $ ip netns add net0 |

可以创建一个完全隔离的新网络环境，这个环境包括一个独立的网卡空间，路由表，ARP表，ip地址表，iptables，ebtables，等等。总之，与网络有关的组件都是独立的。

ip命令需要root权限的，但是由于本文大量使用ip命令，于是笔者给ip命令添加了capability，使普通用户也能使用ip命令

使用命令

|  |
| --- |
| $ ip netns list  net0 |

可以看到我们刚才创建的网络环境

## 进入虚拟网络环境

使用命令

|  |
| --- |
| $ ip netns **exec** net0 `**command**` |

我们可以在 net0 虚拟环境中运行任何命令

|  |
| --- |
| $ ip netns **exec** net0 bash  $ ip ad  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00 |

这样我们可以在新的网络环境中打开一个shell，可以看到，新的网络环境里面只有一个lo设备，并且这个lo设备与外面的lo设备是不同的，之间不能互相通讯。

## 连接两个网络环境

新的网络环境里面没有任何网络设备，并且也无法和外部通讯，就是一个孤岛，通过下面介绍的这个方法可以把两个网络环境连起来，简单的说，就是在两个网络环境之间拉一根网线

|  |
| --- |
| $ ip netns add net1 |

先创建另一个网络环境net1，我们的目标是把net0与net1连起来

|  |
| --- |
| $ ip link add **type** veth  $ ip ad  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  81: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000  link/ether 12:39:09:81:3a:dd brd ff:ff:ff:ff:ff:ff  82: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000  link/ether 32:4f:fd:cc:79:1b brd ff:ff:ff:ff:ff:ff |

这里创建连一对veth虚拟网卡，类似pipe，发给veth0的数据包veth1那边会收到，发给veth1的数据包veth0会收到。就相当于给机器安装了两个网卡，并且之间用网线连接起来了

|  |
| --- |
| $ ip link **set** veth0 netns net0  $ ip link **set** veth1 netns net1 |

这两条命令的意思就是把veth0移动到net0环境里面，把veth1移动到net1环境里面，我们看看结果

|  |
| --- |
| $ ip ad  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  $ ip netns **exec** net0 ip ad  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  81: veth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000  link/ether 12:39:09:81:3a:dd brd ff:ff:ff:ff:ff:ff  $ ip netns **exec** net1 ip ad  1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN  link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00  82: veth1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000  link/ether 32:4f:fd:cc:79:1b brd ff:ff:ff:ff:ff:ff |

veth0 veth1已经在我们的环境里面消失了，并且分别出现在net0与net1里面。下面我们简单测试一下net0与net1的联通性

|  |
| --- |
| $ ip netns **exec** net0 ip link **set** veth0 up  $ ip netns **exec** net0 ip address add 10.0.1.1/24 dev veth0  $ ip netns **exec** net1 ip link **set** veth1 up  $ ip netns **exec** net1 ip address add 10.0.1.2/24 dev veth1 |

分别配置好两个设备，然后用ping测试一下联通性：

|  |
| --- |
| $ ip netns **exec** net0 ping -c 3 10.0.1.2  PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.  64 bytes from 10.0.1.2: icmp\_req=1 ttl=64 time=0.101 ms  64 bytes from 10.0.1.2: icmp\_req=2 ttl=64 time=0.057 ms  64 bytes from 10.0.1.2: icmp\_req=3 ttl=64 time=0.048 ms  --- 10.0.1.2 ping statistics ---  3 packets transmitted, 3 received, 0% packet loss, time 1999ms  rtt min/avg/max/mdev = 0.048/0.068/0.101/0.025 ms |

## 一个稍微复杂的网络环境

创建虚拟网络环境并且连接网线

|  |
| --- |
| ip netns add net0  ip netns add net1  ip netns add bridge  ip link add **type** veth  ip link **set** dev veth0 name net0-bridge netns net0  ip link **set** dev veth1 name bridge-net0 netns bridge  ip link add **type** veth  ip link **set** dev veth0 name net1-bridge netns net1  ip link **set** dev veth1 name bridge-net1 netns bridge |

在bridge中创建并且设置br设备

|  |
| --- |
| ip netns **exec** bridge brctl addbr br  ip netns **exec** bridge ip link **set** dev br up  ip netns **exec** bridge ip link **set** dev bridge-net0 up  ip netns **exec** bridge ip link **set** dev bridge-net1 up  ip netns **exec** bridge brctl addif br bridge-net0  ip netns **exec** bridge brctl addif br bridge-net1 |

然后配置两个虚拟环境的网卡

|  |
| --- |
| ip netns **exec** net0 ip link **set** dev net0-bridge up  ip netns **exec** net0 ip address add 10.0.1.1/24 dev net0-bridge |

|  |
| --- |
| ip netns **exec** net1 ip link **set** dev net1-bridge up  ip netns **exec** net1 ip address add 10.0.1.2/24 dev net1-bridge |

测试

|  |
| --- |
| $ ip netns **exec** net0 ping -c 3 10.0.1.2  PING 10.0.1.2 (10.0.1.2) 56(84) bytes of data.  64 bytes from 10.0.1.2: icmp\_req=1 ttl=64 time=0.121 ms  64 bytes from 10.0.1.2: icmp\_req=2 ttl=64 time=0.072 ms  64 bytes from 10.0.1.2: icmp\_req=3 ttl=64 time=0.069 ms  --- 10.0.1.2 ping statistics ---  3 packets transmitted, 3 received, 0% packet loss, time 1999ms  rtt min/avg/max/mdev = 0.069/0.087/0.121/0.025 ms |

## 配置lldpd检查线路链接情况

随着虚拟网络环境增加，环境中网卡数量也在不断增加，经常会忘记环境中哪些网卡连接到哪里，通过 lldp [[2](https://blog.kghost.info/2013/03/01/linux-network-emulator/#_footnote_2)]协议，我们可以清楚看到每个网卡连接到了哪些环境中的哪个网卡。

github 上有一个 lldp 在 linux 下的开源实现 [[3](https://blog.kghost.info/2013/03/01/linux-network-emulator/#_footnote_3)]，通过在每个环境中起一个 lldp daemon，我们就可以实时查看每个网卡的连接情况

**Bridge 上 lldp 的数据**

|  |
| --- |
| $ lldpcli show neighbors  LLDP neighbors:  Interface: bridge-net0, via: LLDP, RID: 2, Time: 0 day, 00:06:53  Chassis:  ChassisID: mac 82:be:2a:ec:70:69  SysName: localhost  SysDescr: net0  Capability: Bridge, off  Capability: Router, off  Capability: Wlan, off  Port:  PortID: mac 82:be:2a:ec:70:69  PortDescr: net0-bridge  Interface: bridge-net1, via: LLDP, RID: 1, Time: 0 day, 00:06:53  Chassis:  ChassisID: mac b2:34:28:b1:be:49  SysName: localhost  SysDescr: net1  Capability: Bridge, off  Capability: Router, off  Capability: Wlan, off  Port:  PortID: mac b2:34:28:b1:be:49  PortDescr: net1-bridge |