

You have 1 free member-only story left this month. [Sign up for Medium and get an extra one](#)

Containers from Zero To Hero: Container Network

Part 7 of a series of articles about learning containers and becoming a hero!

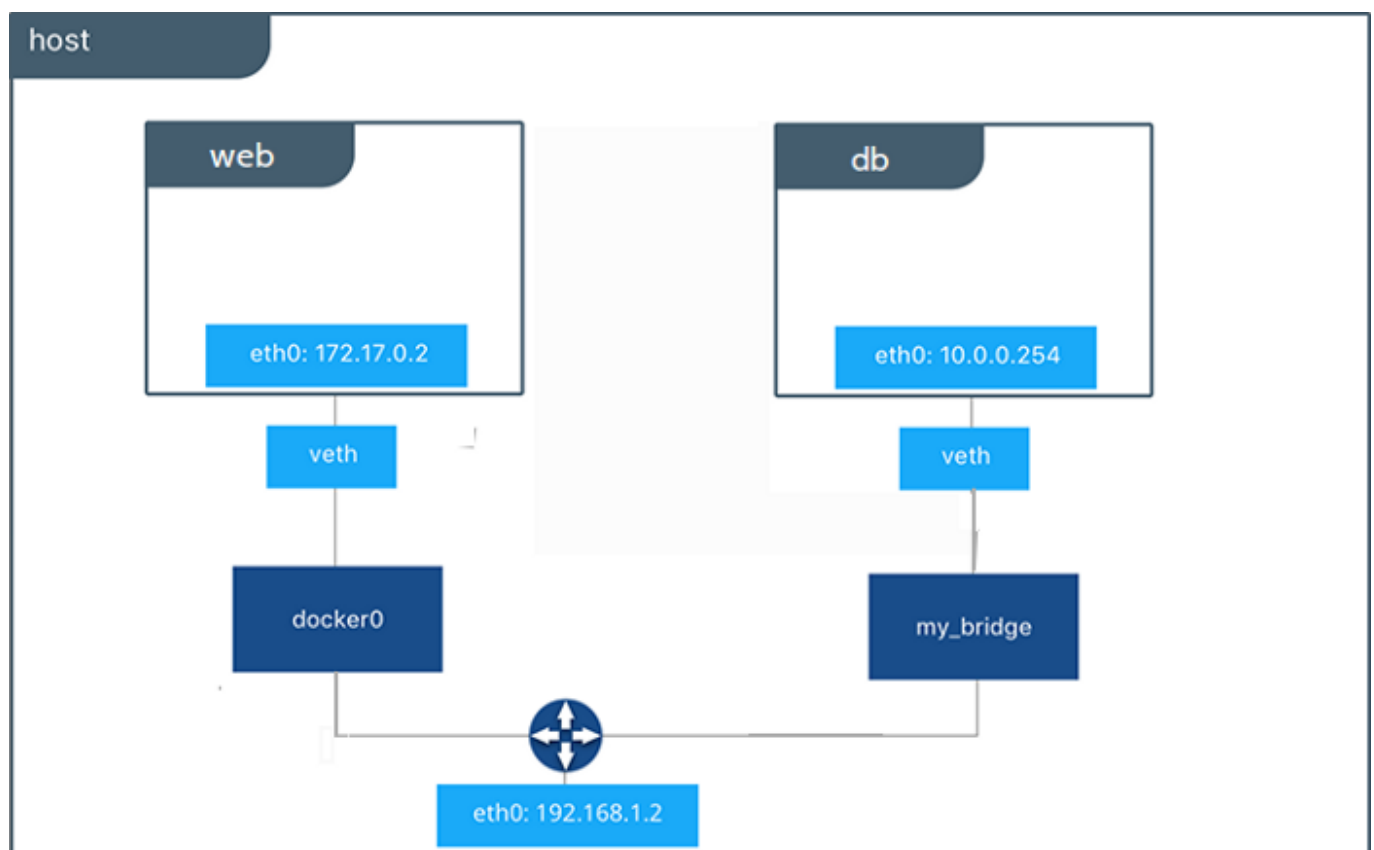


Tony Li Xu

Follow



Sep 16 · 6 min read ★



- Previous: [Container File System](#)
- Next: TBD

In my previous article, I talked about [Container File System](#). Container uses [OverlayFS](#) file system type to reduce the data redundancy between different container images, and it has [upper](#), [lower](#), [merged](#) and [work](#) four different layers. In this article, I will talk about [Container Network](#).

Understand Container Network Namespace

Regarding Network Namespace, if we take it literally, we know that it is the isolation of the network on a Linux node, but which part of the network resources does it specifically isolate?

Let's start with `Linux Programmer's Manual`. In this manual, it has a short description and listed some of main resources of Network Namespace.

network_namespaces(7) — Linux manual page

[NAME](#) | [DESCRIPTION](#) | [SEE ALSO](#) | [COLOPHON](#)

NETWORK_NAMESPACES(7) **Linux Programmer's Manual** **NETWORK_NAMESPACES(7)**

NAME [top](#)

`network_namespaces` - overview of Linux network namespaces

DESCRIPTION [top](#)

Network namespaces provide isolation of the system resources associated with networking: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the `/proc/net` directory (which is a symbolic link to `/proc/PID/net`), the `/sys/class/net` directory, various files under `/proc/sys/net`, port numbers (sockets), and so on. In addition, network namespaces isolate the UNIX domain abstract socket namespace (see `unix(7)`).

A physical network device can live in exactly one network namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).

A virtual network (`veth(4)`) device pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a namespace is freed, the `veth(4)` devices that it contains are destroyed.

Use of network namespaces requires a kernel that is configured with the `CONFIG_NET_NS` option.

Network Namespace

Network Namespace Resources

- **Network equipment.** Refers to `lo`, `eth0` and other network equipments. You can see them through the `ip link` command.

- **IPv4 and IPv6 protocol stacks.** From protocol stacks we know that the IP layer and the TCP and UDP protocol stacks above also work independently for each Namespace. Therefore, for many protocols such as IP, TCP, and UDP, their related parameters are also independent of each Namespace. Most of these parameters are under the `/proc/sys/net/` directory, and also include TCP and UDP port resources.
- **IP routing table.** This resource is also relatively easy to understand. You can run the `ip route` command in different Network Namespaces to see different routing tables.
- **Firewall rules.** In fact, I am talking about `iptables` rules, and `iptables` rules can be configured independently in each Namespace.
- **Network status information.** You can get this information from `/proc/net` and `/sys/class/net`. The status here basically includes the status information of the previous four kinds of resources.

How Do We Create Network NameSpace?

We can create a new Network Namespace by calling the `clone()` or `unshare()` functions.

clone() function

When a new process is created, along with the creation of the new process, a new Network Namespace is also created. This method is actually implemented by attaching the `CLONE_NEWNET` flag to the `clone()` system call. Just in case you are interested in the source code:

```
int new_netns(void *para)
{
    printf("New Namespace Devices:\n");
    system("ip link");
    printf("\n\n");

    sleep(100);
    return 0;
}

int main(void)
{
    pid_t pid;

    printf("Host Namespace Devices:\n");
    system("ip link");
    printf("\n\n");

    pid =
        clone(new_netns, stack + STACK_SIZE, CLONE_NEWNET |
SIGCHLD, NULL);
    if (pid == -1)
        errExit("clone");
}
```

```

        if (waitpid(pid, NULL, 0) == -1)
            errExit("waitpid");

        return 0;
    }

```

unshare() function

We can call the `unshare()` system call to directly change the Network Namespace of the current process. Again, I post the source code for you interest:

```

int main(void)
{
    pid_t pid;

    printf("Host Namespace Devices:\n");
    system("ip link");
    printf("\n\n");

    if (unshare(CLONE_NEWNET) == -1)
        errExit("unshare");

    printf("New Namespace Devices:\n");
    system("ip link");
    printf("\n\n");

    return 0;
}

```

Note: Not only the Network Namespace, but other Namespaces are also established through the `clone()` or `unshare()` function calls. Even the container creation program, such as `runC`, also uses `unshare()` to create a Namespace for the newly created container. `runC` is a CLI tool for spawning and running containers on Linux according to the OCI specification (<https://github.com/opencontainers/runc>).

After the Network Namespace is created, we can run the command `lsns -t net` on the host to view the existing Network Namespace in the system. Of course, `lsns` can also be used to view other Namespaces.

Let's run a quick example:

```

# gcc -o clone-ns clone-ns.c
# ls
clone-ns.c clone-ns
# ./clone-ns
Host Namespace Devices:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN

```

```

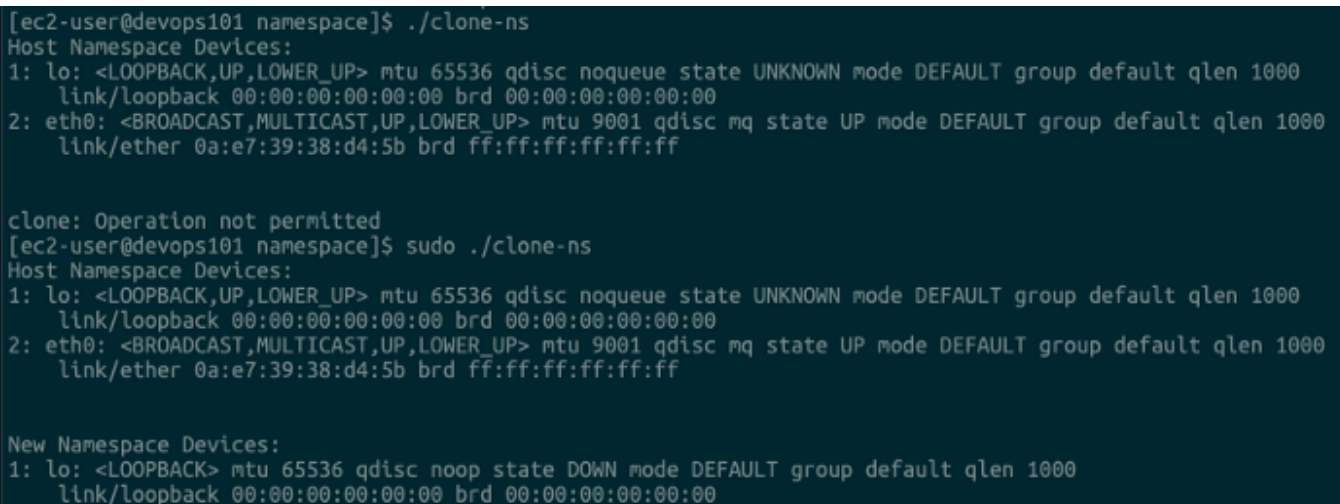
mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP
mode DEFAULT group default qlen 1000
    link/ether 0a:e7:39:38:d4:5b brd ff:ff:ff:ff:ff:ff

clone: Operation not permitted
[ec2-user@devops101 namespace]$ sudo ./clone-ns
Host Namespace Devices:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP
mode DEFAULT group default qlen 1000
    link/ether 0a:e7:39:38:d4:5b brd ff:ff:ff:ff:ff:ff

New Namespace Devices:
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

```

Screenshot:



```

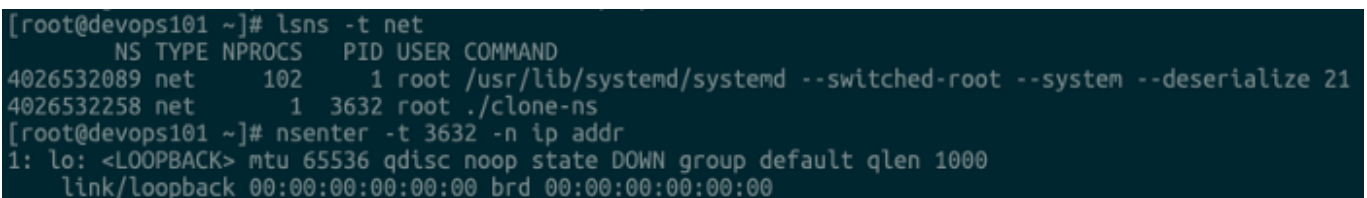
[ec2-user@devops101 namespace]$ ./clone-ns
Host Namespace Devices:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0a:e7:39:38:d4:5b brd ff:ff:ff:ff:ff:ff

clone: Operation not permitted
[ec2-user@devops101 namespace]$ sudo ./clone-ns
Host Namespace Devices:
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0a:e7:39:38:d4:5b brd ff:ff:ff:ff:ff:ff

New Namespace Devices:
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

```

Network Namespace Screenshot



```

[root@devops101 ~]# lsns -t net
NS TYPE NPROCS  PID USER COMMAND
4026532089 net      102    1 root /usr/lib/systemd/systemd --switched-root --system --deserialize 21
4026532258 net       1  3632 root ./clone-ns
[root@devops101 ~]# nsenter -t 3632 -n ip addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

```

lsns and nsenter command

How Do We Set Container Network Parameter?

Now we understand a bit more about `Network Namespace` , let's see how we can set container network parameters.

But first of all, you need to understand that the network parameters of the `Network Namespace` in the container are not completely inherited from the Host Namespace of the host, nor are they completely reinitialized when a new `Network Namespace` is established.

Let's use our `httpd` container as an example, and try to update network settings inside this container and see what happens:

```
[root@devops101 ~]# docker run -d --name httpd registry/httpd:v1
b4e702116ae5e3bd9c771f603ab4a7113c822897339b9ff326456ef494be4a43
[root@devops101 ~]# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS        NAMES
b4e702116ae5   registry/httpd "/sbin/httpd -D FORE..." 4 seconds ago  Up 3 seconds  80           httpd
[root@devops101 ~]# docker exec -it httpd /bin/bash
[root@b4e702116ae5 /]# echo 600 > /proc/sys/net/ipv4/tcp_keepalive_time
bash: /proc/sys/net/ipv4/tcp_keepalive_time: Read-only file system
```

Update network settings in container

Aha, we couldn't do it since the `/proc/sys` is mounted as `read-only`. We can verify by running the following command:

```
[root@b4e702116ae5 /]# cat /proc/mounts | grep "proc/sys"
proc /proc/sys proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sysrq-trigger proc ro,nosuid,nodev,noexec,relatime 0 0
```

Why is `/proc/sys` read-only mount in the container? For security considerations, `runC` handled all the `/proc` and `/sys`-related directories in the container as read-only mounts by default.

Then how do we do it? If you have root privileges on the host. The simplest and rude way is to use the `nsenter` command we mentioned earlier to modify the network parameters in the container. However, **this method is obviously not allowed in the production environment**, because we will not allow users to have the login permission of the host.

Generally speaking, such changes should be made only before the application in the container has been started. Otherwise many tcp links will be established already, so even if the new parameters are changed, the established links will not take effect. This requires restarting the application. We all know that application restarts are usually avoided in the production environment, which is obviously inappropriate.

So the best time to change network parameters is obviously when the container that has just been started, and the application in the container has not been started.

docker sysctl

In fact, `runC` also reserved a modification interface before doing a read-only mount on the `proc/sys` directory, which is used to modify the parameters under “/proc/sys” in the container, which are also `sysctl` parameters.

For example:

```
[root@devops101 ~]# docker run -d --name httpd --sysctl
net.ipv4.tcp_keepalive_time=600 registry/httpd:v1
bbba53b0deb4ca3c4221fbf6e8bd82aee4678da2c80844f5260a4c427e441ce9
[root@devops101 ~]# docker exec httpd cat
/proc/sys/net/ipv4/tcp_keepalive_time
600
```

Conclusion

I summarized tools/commands for network namespace parameter update in the following diagram:

Network Namespace Tools

Name	Purpose
ip netns	Process network namespace management
unshare	Create a new namespace
lsns	Check all namespaces on the host
nsenter	Run command in different namespaces

Network Namespace Tools

I hope you enjoyed this article, and I will see you soon in my next container article!

More content at plainenglish.io

Docker

Containers

DevOps

AWS

Programming

Get the Medium app

