Linux Namespace

一、namespace 的概念

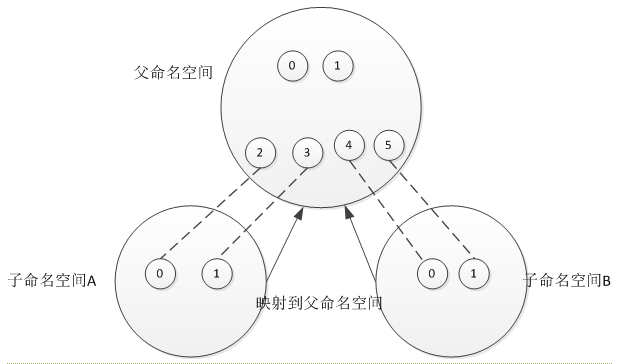
namespace 是 Linux 内核用来隔离内核资源的方式。

通过 namespace 可以让一些进程只能看到与自己相关的一部分资源，而另外一些进程也只能看到与它们自己相关的资源，这两拨进程根本就感觉不到对方的存在。具体的实现方式是把一个或多个进程的相关资源指定在同一个 namespace 中。

Linux namespaces 是对全局系统资源的一种封装隔离，使得处于不同 namespace 的进程拥有独立的全局系统资源，改变一个 namespace 中的系统资源只会影响当前 namespace 里的进程，对其他 namespace 中的进程没有影响。

在操作系统层面上看，就会出现多个相同pid的进程。系统中可以同时存在两个进程号为0,1,2的进程，由于属于不同的namespace，所以它们之间并不冲突。而在用户层面上只能看到属于用户自己namespace下的资源，例如使用ps命令只能列出自己namespace下的进程。这样每个namespace看上去就像一个单独的Linux系统。



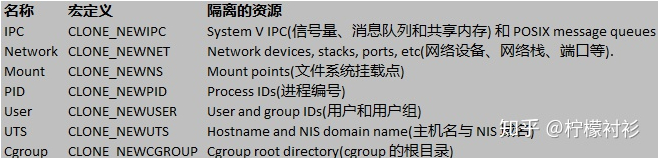


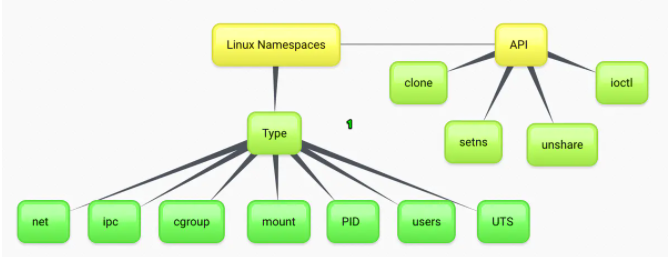
命名空间建立系统的不同视图，对于每一个命名空间，从用户看起来，应该像一台单独的Linux计算机一样，有自己的init进程(PID为0)，其他进程的PID依次递增，A和B空间都有PID为0的init进程，子容器的进程映射到父容器的进程上，父容器可以知道每一个子容器的运行状态，而子容器与子容器之间是隔离的。

二、namespace 的用途

Linux 内核实现 namespace 的一个主要目的就是**实现轻量级虚拟化(容器)服务**。在同一个 namespace 下的进程可以感知彼此的变化，而对外界的进程一无所知。这样就可以让容器中的进程产生错觉，认为自己置身于一个独立的系统中，从而达到隔离的目的。也就是说 linux 内核提供的 namespace 技术为 docker 等容器技术的出现和发展提供了基础条件。

为了在分布式的环境下进行通信和定位，容器必须要有独立的 IP、端口和路由等，这就需要对网络进行隔离；同时容器还需要一个独立的主机名以便在网络中标识自己；接下来还需要进程间的通信、用户权限等的隔离；最后，运行在容器中的应用需要有进程号(PID)，自然也需要与宿主机中的 PID 进行隔离。也就是说这六种隔离能力是实现一个容器的基础。 linux 内核的 namespace 特性提供了如下的隔离能力：





三、Linux系统下的namespace类型



四、namespace相关系统调用

Linux 提供了多个 API 用来操作 namespace：

* **clone** 创建新进程，flags参数可以用来创建新的namespace
* **setns** 让进程加入存在的namespace
* **unshare** 将调用进程移动到新的namespace中
* **ioctl\_ns** 查看namespace的信息

为了确定隔离的到底是哪项 namespace，在使用这些 API 时，通常需要指定一些调用参数：CLONE\_NEWIPC、CLONE\_NEWNET、CLONE\_NEWNS、CLONE\_NEWPID、CLONE\_NEWUSER、CLONE\_NEWUTS 和 CLONE\_NEWCGROUP。如果要同时隔离多个 namespace，可以使用 | (按位或)组合这些参数。同时还可以通过 /proc 下面的一些文件来操作 namespace。

* 进程namespace目录

/proc/[pid]/ns/cgroup 进程cgroup命名空间句柄  
/proc/[pid]/ns/ipc 进程IPC命名空间句柄  
/proc/[pid]/ns/mnt 进程mount命名空间句柄  
/proc/[pid]/ns/net 进程网络命名空间句柄  
/proc/[pid]/ns/pid 进程PID命名空间句柄  
/proc/[pid]/ns/pid\_for\_children 该进程的子进程的PID命名空间句柄  
/proc/[pid]/ns/time 进程time命名空间句柄  
/proc/[pid]/ns/user 进程user命名空间句柄  
/proc/[pid]/ns/uts 进程UTS命名空间句柄

* namespace limit

/proc/sys/user目录  
max\_cgroup\_namespaces  
max\_inotify\_instances  
max\_inotify\_watches  
max\_ipc\_namespaces  
max\_mnt\_namespaces  
max\_net\_namespaces  
max\_pid\_namespaces  
max\_user\_namespaces  
max\_uts\_namespaces

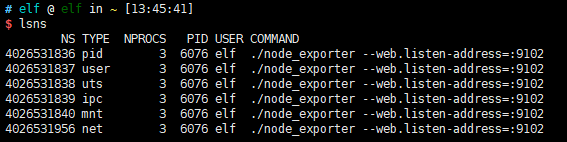
五、namespace 生命周期

namespace中的最后一个进程结束或离开namespace，该namespace生命周期结束。

六、相关命令

* 查看系统下所有的namespace

**lsns**



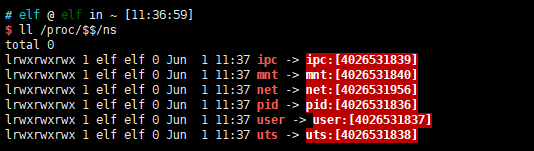
* 查看进程所属的 namespace

从版本号为 3.8 的内核开始，**/proc/[pid]/ns** 目录下会包含进程所属的 namespace 信息，使用下面的命令可以查看当前进程所属的 namespace 信息：

**ls /proc/$$/ns**



**ll /proc/$$/ns**



首先，这些 namespace 文件都是链接文件。链接文件的内容的格式为 xxx:[inode number]。其中的 xxx 为 namespace 的类型，inode number 则用来标识一个 namespace，我们也可以把它理解为 namespace 的 ID。如果两个进程的某个 namespace 文件指向同一个链接文件，说明其相关资源在同一个 namespace 中。

其次，在 /proc/[pid]/ns 里放置这些链接文件的另外一个作用是，一旦这些链接文件被打开，只要打开的文件描述符(fd)存在，那么就算该 namespace 下的所有进程都已结束，这个 namespace 也会一直存在，后续的进程还可以再加入进来。

除了打开文件的方式，我们还可以通过文件挂载的方式阻止 namespace 被删除。比如可以把当前进程中的 uts 挂载到 ~/uts 文件：

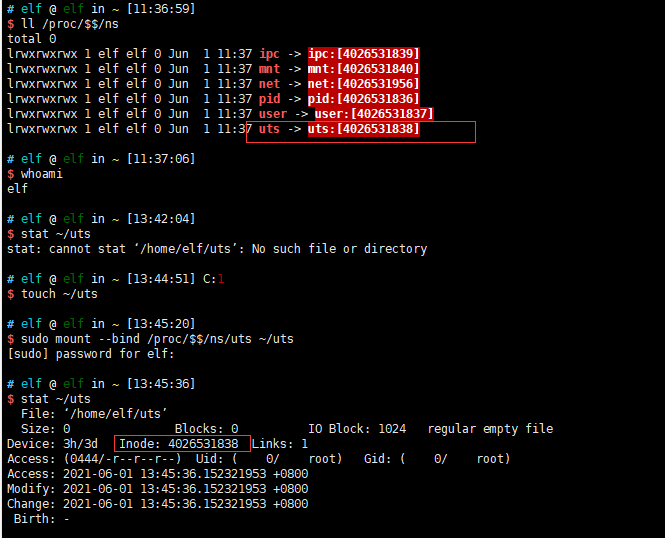
**touch ~/uts**

**sudo mount --bind /proc/$$/ns/uts ~/uts**

使用 stat 命令检查下结果：

**stat ~/uts**

~/uts 的 inode 和链接文件中的 inode number 是一样的，它们是同一个文件。



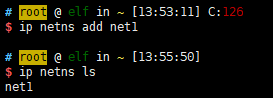
七、通过 IP 命令操作来简单介绍 network namespace 的基本概念和用法

和 network namespace 相关的操作的子命令是 **ip netns**

1. ip netns add xx 创建一个 namespace

**ip netns add net1**

**ip netns ls**



1. ip netns exec xx yy 在新 namespace xx 中执行 yy 命令

**ip netns exec net1 ip addr**

**ip netns exec net1 bash // 在 net1 中打开一个shell终端**

**ip addr // 在net1中的shell终端**

**exit // 退出net1**

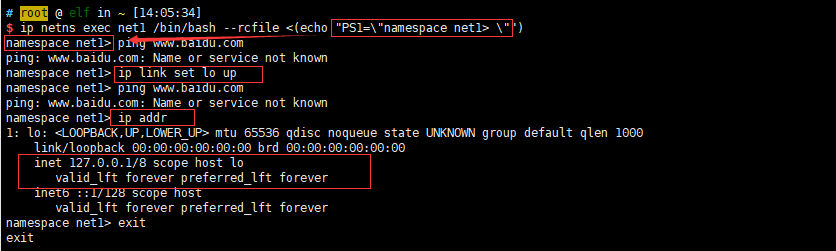


上面 bash 不好区分是当前是在哪个 shell，可以采用下面的方法解决：

**ip netns exec net1 /bin/bash --rcfile <(echo "PS1=\"namespace net1> \"")**

**namespace net1> ping** [**www.baidu.com**](http://www.baidu.com)

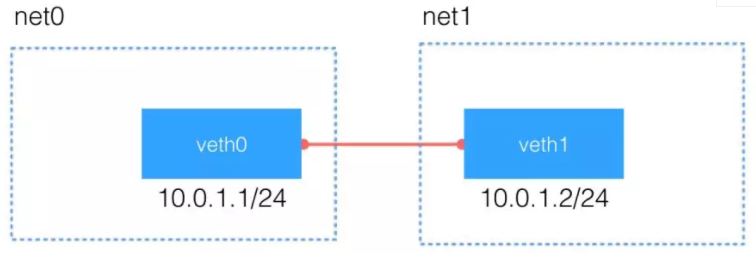
每个 namespace 在创建的时候会自动创建一个回环接口 lo ，默认不启用，可以通过 ip link set lo up 启用。



1. network namespace之间的通信

新创建的 namespace 默认不能和主机网络，以及其他 namespace 通信。

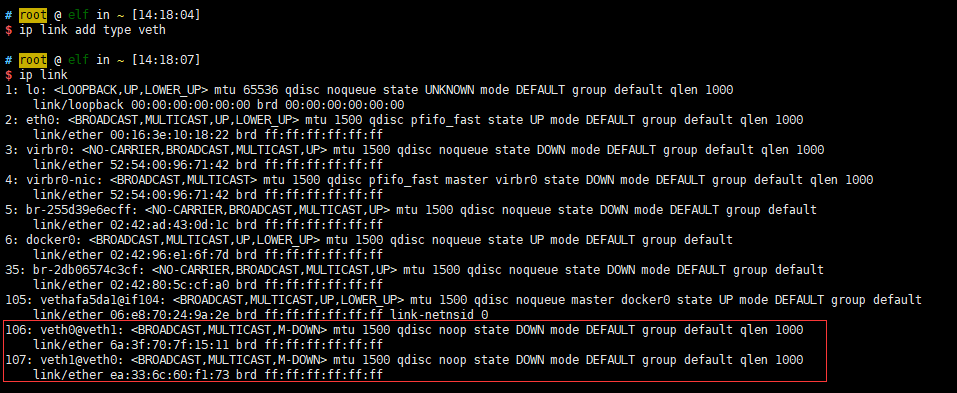
可以使用 Linux 提供的 **veth pair** 来完成通信。下面显示两个 namespace 之间通信的网络拓扑：



3.1 ip link add type veth 创建 veth pair

**ip link add type veth**

**ip link**



使用命令 **ip link add xxx type veth peer name yyy** 指定 veth pair 的名字。

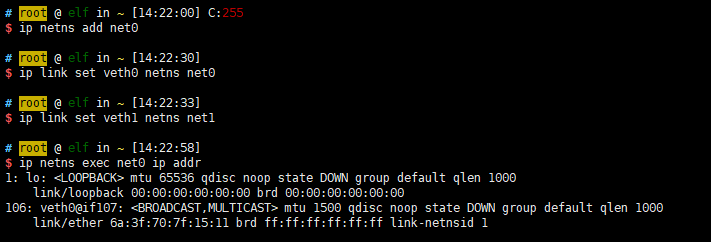
3.2 ip link set xx netns yy 将 veth xx 加入到 namespace yy 中

**ip netns add net0**

**ip link set veth0 netns net0**

**ip link set veth1 netns net1**

**ip netns exec net0 ip addr**



3.3 给 veth pair 配上 ip 地址

**ip netns exec net0 ip link set veth0 up**

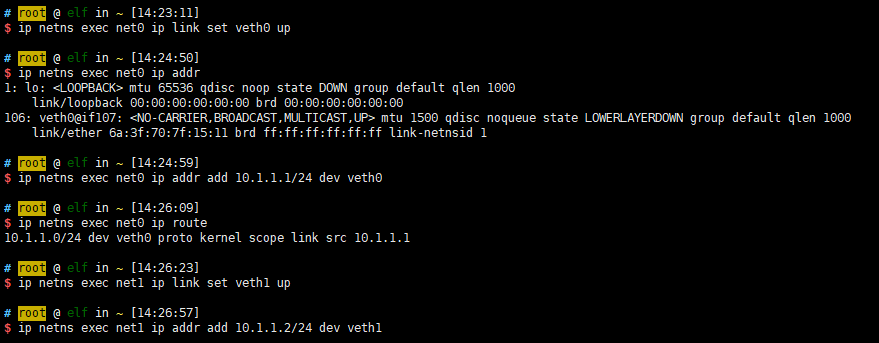
**ip netns exec net0 ip addr**

**ip netns exec net0 ip addr add 10.1.1.1/24 dev veth0**

**ip netns exec net0 ip route**

**ip netns exec net1 ip link set veth1 up**

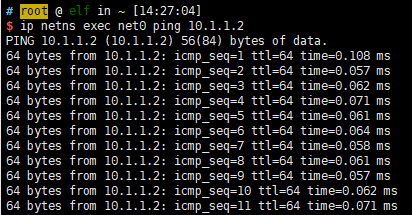
**ip netns exec net1 ip addr add 10.1.1.2/24 dev veth1**



可以看到，在配完 ip 之后，还自动生成了对应的路由表信息。

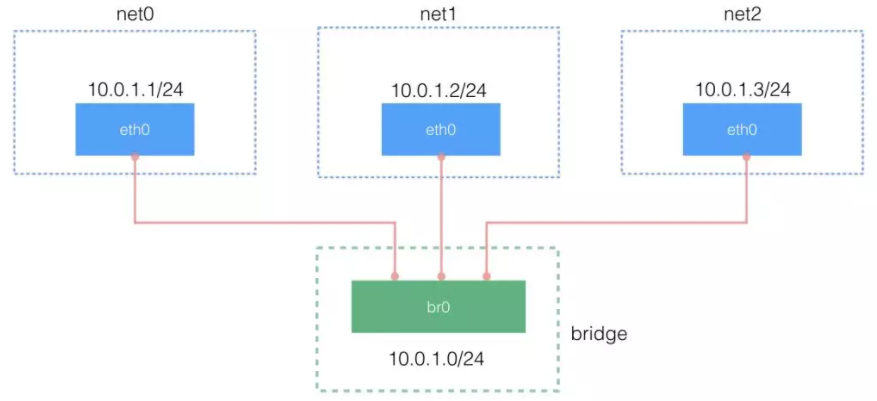
3.4. ping 测试两个 namespace 的连通性

**ip netns exec net0 ping 10.1.1.2**



1. 多个不同 namespace 之间的通信

2 个 namespace 之间通信可以借助 veth pair ，多个 namespace 之间的通信则可以使用 bridge 来转接，不然每两个 namespace 都去配 veth pair 将会是一件麻烦的事。下面就看看如何使用 bridge 来转接。拓扑图如下：



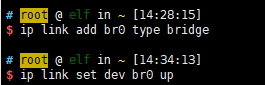
4.1 使用 ip link 和 brctl 创建 bridge

通常 Linux 中和 bridge 有关的操作是使用命令 brctl (yum install -y bridge-utils ) 。但为了前后照应，这里都用 ip 相关的命令来操作。

**// 建立一个 bridge**

**ip link add br0 type bridge**

**ip link set dev br0 up**



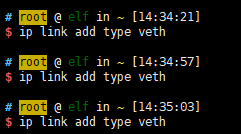
4.2 创建 veth pair

**//（1）创建 3 个 veth pair**

**ip link add type veth**

**ip link add type veth**

**ip link add type veth**



4.3 将 veth pair 的一头挂到 namespace 中，一头挂到 bridge 上，并设 IP 地址#

**// （1）配置第 1 个 net0**

**ip link set dev veth1 netns net0**

**ip netns exec net0 ip link set dev veth1 name eth0**

**ip netns exec net0 ip addr add 10.0.1.1/24 dev eth0**

**ip netns exec net0 ip link set dev eth0 up**

**ip link set dev veth0 master br0**

**ip link set dev veth0 up**

**// （2）配置第 2 个 net1**

**ip link set dev veth3 netns net1**

**ip netns exec net1 ip link set dev veth3 name eth0**

**ip netns exec net1 ip addr add 10.0.1.2/24 dev eth0**

**ip netns exec net1 ip link set dev eth0 up**

**ip link set dev veth2 master br0**

**ip link set dev veth2 up**

**// （3）配置第 3 个 net2**

**ip link set dev veth5 netns net2**

**ip netns exec net2 ip link set dev veth5 name eth0**

**ip netns exec net2 ip addr add 10.0.1.3/24 dev eth0**

**ip netns exec net2 ip link set dev eth0 up**

**ip link set dev veth4 master br0**

**ip link set dev veth4 up**



这样之后，竟然通不了，经查阅 参见 ，

<https://segmentfault.com/q/1010000010011053/a-1020000010025650>

原因是因为系统为bridge开启了iptables功能，导致所有经过br0的数据包都要受iptables里面规则的限制，而docker为了安全性（我的系统安装了 docker），将iptables里面filter表的FORWARD链的默认策略设置成了drop，于是所有不符合docker规则的数据包都不会被forward，导致这种情况ping不通。

解决办法有两个，二选一：

1. 关闭系统bridge的iptables功能，这样数据包转发就不受iptables影响了：echo 0 > /proc/sys/net/bridge/bridge-nf-call-iptables
2. 为br0添加一条iptables规则，让经过br0的包能被forward：iptables -A FORWARD -i br0 -j ACCEPT

第一种方法不确定会不会影响docker，建议用第二种方法。

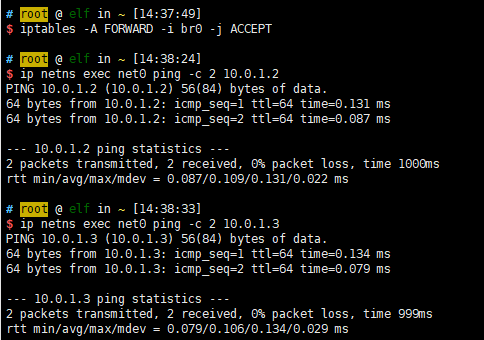
这里采用以下方法解决：

**iptables -A FORWARD -i br0 -j ACCEPT**

结果：

**ip netns exec net0 ping -c 2 10.0.1.2**

**ip netns exec net0 ping -c 2 10.0.1.3**



参考链接：

<https://www.jianshu.com/p/319a0cef1f1e?open_source=weibo_search>

<https://zhuanlan.zhihu.com/p/159362517>

<https://www.cnblogs.com/bakari/p/10443484.html#3956315454>