[**理解Docker（3）：Docker 使用 Linux namespace 隔离容器的运行环境**](http://www.cnblogs.com/sammyliu/p/5878973.html)

本系列文章将介绍Docker的有关知识：

（1）[Docker 安装及基本用法](http://www.cnblogs.com/sammyliu/p/5875470.html)

（2）[Docker 镜像](http://www.cnblogs.com/sammyliu/p/5877964.html%20)

（3）[Docker 容器的隔离性 - 使用 Linux namespace 隔离容器的运行环境](http://www.cnblogs.com/sammyliu/p/5878973.html%20)

（4）[Docker 容器的隔离性 - 使用 cgroups 限制容器使用的资源](http://www.cnblogs.com/sammyliu/p/5886833.html%20)

（5）[Docker 网络](http://www.cnblogs.com/sammyliu/p/5894191.html%20)

**1. 基础知识：Linux namespace 的概念**

    Linux 内核从版本 2.4.19 开始陆续引入了 namespace 的概念。其目的是将某个特定的全局系统资源（global system resource）通过抽象方法使得namespace 中的进程看起来拥有它们自己的隔离的全局系统资源实例（The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. ）。Linux 内核中实现了六种 namespace，按照引入的先后顺序，列表如下：

| **namespace** | **引入的相关内核版本** | **被隔离的全局系统资源** | **在容器语境下的隔离效果** |
| --- | --- | --- | --- |
| **Mount namespaces** | [Linux 2.4.19](http://lwn.net/2001/0301/a/namespaces.php3) | 文件系统挂接点 | 每个容器能看到不同的文件系统层次结构 |
| **UTS namespaces** | [Linux 2.6.19](http://lwn.net/Articles/179345/) | nodename 和 domainname | 每个容器可以有自己的 hostname 和 domainame |
| **IPC namespaces** | [Linux 2.6.19](http://lwn.net/Articles/187274/) | 特定的进程间通信资源，包括[System V IPC](http://www.kernel.org/doc/man-pages/online/pages/man7/svipc.7.html) 和  [POSIX message queues](http://www.kernel.org/doc/man-pages/online/pages/man7/mq_overview.7.html) | 每个容器有其自己的 System V IPC 和 POSIX 消息队列文件系统，因此，只有在同一个 IPC namespace 的进程之间才能互相通信 |
| **PID namespaces** | [Linux 2.6.24](http://lwn.net/Articles/259217/) | 进程 ID 数字空间 （process ID number space） | 每个 PID namespace 中的进程可以有其独立的 PID； 每个容器可以有其 PID 为 1 的root 进程；也使得容器可以在不同的 host 之间迁移，因为 namespace 中的进程 ID 和 host 无关了。这也使得容器中的每个进程有两个PID：容器中的 PID 和 host 上的 PID。 |
| **Network namespaces** | [始于Linux 2.6.24 完成于 Linux 2.6.29](http://lwn.net/Articles/219794/) | 网络相关的系统资源 | 每个容器用有其独立的网络设备，IP 地址，IP 路由表，/proc/net 目录，端口号等等。这也使得一个 host 上多个容器内的同一个应用都绑定到各自容器的 80 端口上。 |
| **User namespaces** | [始于 Linux 2.6.23 完成于 Linux 3.8)](http://lwn.net/Articles/528078/) | 用户和组 ID 空间 | 在 user namespace 中的进程的用户和组 ID 可以和在 host 上不同； 每个 container 可以有不同的 user 和 group id；一个 host 上的非特权用户可以成为 user namespace 中的特权用户； |

Linux namespace 的概念说简单也简单说复杂也复杂。简单来说，我们只要知道，处于某个 namespace 中的进程，能看到独立的它自己的隔离的某些特定系统资源；复杂来说，可以去看看 Linux 内核中实现 namespace 的原理，网络上也有大量的文档供参考，这里不再赘述。

**2. Docker 容器使用 linux namespace 做运行环境隔离**

当 Docker 创建一个容器时，它会创建新的以上六种 namespace 的实例，然后把容器中的所有进程放到这些 namespace 之中，使得Docker 容器中的进程只能看到隔离的系统资源。

**2.1 PID namespace**

我们能看到同一个进程，在容器内外的 PID 是不同的：

* 在容器内 PID 是 1，PPID 是 0。
* 在容器外 PID 是 2198， PPID 是 2179 即 docker-containerd-shim 进程.

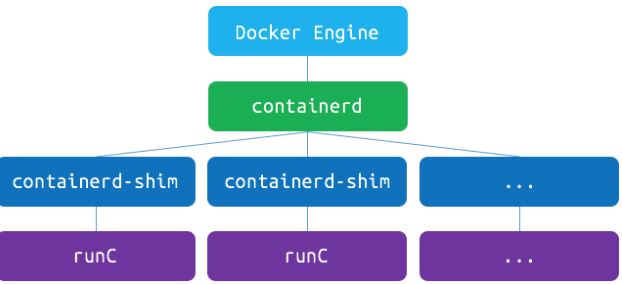
[复制代码](javascript:void(0);)

root@devstack:/home/sammy# ps -ef | grep python  
root 2198 2179 0 00:06 ? 00:00:00 python app.py  
  
root@devstack:/home/sammy# ps -ef | grep 2179  
root 2179 765 0 00:06 ? 00:00:00 docker-containerd-shim 8b7dd09fbcae00373207f01e2acde45740871c9e3b98286b5458b4ea09f41b3e /var/run/docker/libcontainerd/8b7dd09fbcae00373207f01e2acde45740871c9e3b98286b5458b4ea09f41b3e docker-runc  
root 2198 2179 0 00:06 ? 00:00:00 python app.py  
root 2249 1692 0 00:06 pts/0 00:00:00 grep --color=auto 2179

root@devstack:/home/sammy# docker exec -it web31 ps -ef  
UID PID PPID C STIME TTY TIME CMD  
root 1 0 0 16:06 ? 00:00:00 python app.py

[复制代码](javascript:void(0);)

关于 containerd，containerd-shim 和 container 的关系，[文章](https://github.com/crosbymichael/dockercon-2016/blob/master/Creating%20Containerd.pdf) 中的下图可以说明：



* Docker 引擎管理着镜像，然后移交给 containerd 运行，containerd 再使用 runC 运行容器。
* Containerd 是一个简单的守护进程，它可以使用 runC 管理容器，使用 gRPC 暴露容器的其他功能。它管理容器的开始，停止，暂停和销毁。由于容器运行时是孤立的引擎，引擎最终能够启动和升级而无需重新启动容器。
* runC是一个轻量级的工具，它是用来运行容器的，只用来做这一件事，并且这一件事要做好。runC基本上是一个小命令行工具且它可以不用通过Docker引擎，直接就可以使用容器。

因此，容器中的主应用在 host 上的父进程是 containerd-shim，是它通过工具 runC 来启动这些进程的。

这也能看出来，pid namespace 通过将 host 上 PID 映射为容器内的 PID， 使得容器内的进程看起来有个独立的 PID 空间。

**2.2 UTS namespace**

类似地，容器可以有自己的 hostname 和 domainname：

root@devstack:/home/sammy# hostname

devstack

root@devstack:/home/sammy# docker exec -it web31 hostname

8b7dd09fbcae

**2.3 user namespace**

在 Docker 1.10 版本之前，Docker 是不支持 user namespace。也就是说，默认地，容器内的进程的运行用户就是 host 上的 root 用户，这样的话，当 host 上的文件或者目录作为 volume 被映射到容器以后，容器内的进程其实是有 root 的几乎所有权限去修改这些 host 上的目录的，这会有很大的安全问题。

举例：

* 启动一个容器： docker run -d -v /bin:/host/bin --name web34 training/webapp python app.py
* 此时进程的用户在容器内和外都是root，它在容器内可以对 host 上的 /bin 目录做任意修改：

root@devstack:/home/sammy# docker exec -ti web34 id

uid=0(root) gid=0(root) groups=0(root)

root@devstack:/home/sammy# id

uid=0(root) gid=0(root) groups=0(root)

而 Docker 1.10 中引入的 user namespace 就可以让容器有一个 “假”的  root 用户，它在容器内是 root，在容器外是一个非 root 用户。也就是说，user namespace 实现了 host users 和 container users 之间的映射。

启用步骤：

1. 修改 /etc/default/docker 文件，添加行  DOCKER\_OPTS="--userns-remap=default"
2. 重启 docker 服务，此时 dockerd 进程为 /usr/bin/dockerd --userns-remap=default --raw-logs
3. 然后创建一个容器：docker run -d -v /bin:/host/bin --name web35 training/webapp python app.py
4. 查看进程在容器内外的用户：

root@devstack:/home/sammy# ps -ef | grep python

231072 1726 1686 0 01:44 ? 00:00:00 python app.py

root@devstack:/home/sammy# docker exec web35 ps -ef

UID PID PPID C STIME TTY TIME CMD

root 1 0 0 17:44 ? 00:00:00 python app.py

* 查看文件/etc/subuid 和 /etc/subgid，可以看到 dockermap 用户在host 上的 uid 和 gid 都是 231072：

[复制代码](javascript:void(0);)

root@devstack:/home/sammy# cat /etc/subuid

sammy:100000:65536

stack:165536:65536

dockremap:231072:65536

root@devstack:/home/sammy# cat /etc/subgid  
sammy:100000:65536  
stack:165536:65536  
dockremap:231072:65536

[复制代码](javascript:void(0);)

* 再看文件/proc/1726/uid\_map，它表示了容器内外用户的映射关系，即将host 上的 231072 用户映射为容器内的 0 （即root）用户。

root@devstack:/home/sammy# cat /proc/1726/uid\_map

0 231072 65536

* 现在，我们试图在容器内修改 host 上的 /bin 文件夹，就会提示权限不足了：

root@80993d821f7b:/host/bin# touch test2

touch: cannot touch 'test2': Permission denied

这说明通过使用 user namespace，使得容器内的进程运行在非 root 用户，我们就成功地限制了容器内进程的权限。

其他的几个 namespace，比如 network，mnt 等，比较简单，这里就不多说了。总之，Docker 守护进程为每个容器都创建了六种namespace 的实例，使得容器中的进程都处于一种隔离的运行环境之中：

[复制代码](javascript:void(0);)

root@devstack:/proc/1726/ns# ls -l

total 0

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:45 ipc -> ipc:[4026532210]

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:45 mnt -> mnt:[4026532208]

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:44 net -> net:[4026532213]

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:45 pid -> pid:[4026532211]

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:45 user -> user:[4026532207]

lrwxrwxrwx 1 231072 231072 0 Sep 18 01:45 uts -> uts:[4026532209]

[复制代码](javascript:void(0);)

**2.4 network namespace**

  默认情况下，当 docker 实例被创建出来后，使用 ip netns  命令无法看到容器实例对应的 network namespace。这是因为 ip netns 命令是从 /var/run/netns 文件夹中读取内容的。

步骤：

1. 找到容器的主进程 ID

root@devstack:/home/sammy# docker inspect --format '{{.State.Pid}}' web5

2704

1. 创建  /var/run/netns 目录以及符号连接

root@devstack:/home/sammy# mkdir /var/run/netns

root@devstack:/home/sammy# ln -s /proc/2704/ns/net /var/run/netns/web5

1. 此时可以使用 ip netns 命令了

[复制代码](javascript:void(0);)

root@devstack:/home/sammy# ip netns

web5

root@devstack:/home/sammy# ip netns exec web5 ip addr

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 65536 qdisc noqueue state UNKNOWN group default

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

15: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc noqueue state UP group default

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

[复制代码](javascript:void(0);)

**3. Docker run 命令中 namespace 中相关参数**

Docker run 命令有几个参数和 namespace 相关：

* --ipc string IPC namespace to use
* --pid string PID namespace to use
* --userns string User namespace to use
* --uts string UTS namespace to use

**3.1 --userns**

--userns：指定容器使用的 user namespace

* 'host': 使用 Docker host user namespace
* '': 使用由 `--userns-remap‘ 指定的 Docker deamon user namespace

你可以在启用了 user namespace 的情况下，强制某个容器运行在 host user namespace 之中：

root@devstack:/proc/2835# docker run -d -v /bin:/host/bin --name web37 --userns host training/webapp python app.py

9c61e9a233abef7badefa364b683123742420c58d7a06520f14b26a547a9476c

root@devstack:/proc/2835# ps -ef | grep python

root 2962 2930 1 02:17 ? 00:00:00 python app.py

否则默认的话，就会运行在特定的 user namespace 之中了。

**3.2 --pid**

同样的，可以指定容器使用 Docker host pid namespace，这样，在容器中的进程，可以看到 host 上的所有进程。注意此时不能启用 user namespace。

[复制代码](javascript:void(0);)

root@devstack:/proc/2962# docker run -d -v /bin:/host/bin --name web38 --pid host --userns host training/webapp python app.py

f40f6702b61e3028a6708cdd7b167474ddf2a98e95b6793a1326811fc4aa161d

root@devstack:/proc/2962#

root@devstack:/proc/2962# docker exec -it web38 bash

root@f40f6702b61e:/opt/webapp# ps aux

USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND

root 1 0.0 0.1 33480 2768 ? Ss 17:40 0:01 /sbin/init

root 2 0.0 0.0 0 0 ? S 17:40 0:00 [kthreadd]

root 3 0.0 0.0 0 0 ? S 17:40 0:00 [ksoftirqd/0]

root 5 0.0 0.0 0 0 ? S< 17:40 0:00 [kworker/0:0H]

root 6 0.0 0.0 0 0 ? S 17:40 0:00 [kworker/u2:0]

root 7 0.0 0.0 0 0 ? S 17:40 0:00 [rcu\_sched]  
......

[复制代码](javascript:void(0);)

**3.3 --uts**

同样地，可以使容器使用 Docker host uts namespace。此时，最明显的是，容器的 hostname 和 Docker hostname 是相同的。

root@devstack:/proc/2962# docker run -d -v /bin:/host/bin --name web39 --uts host training/webapp python app.py

38e8b812e7020106bf8d3952b88085028fc87f4427af0c3b0a29b6a69c979221

root@devstack:/proc/2962# docker exec -it web39 bash

root@devstack:/opt/webapp# hostname

devstack