原创 多选参数 于 2020-11-26 11:30:00 发布 189 收藏 6 版权
文章标签： 网络 docker linux java 操作系统

Namespace

Linux Namespace 是 Linux 提供的一种内核级别环境隔离的方法。这种隔离机制和 chroot 很类似，chroot 是把某个目录修改为根目录，从而无法访问外部的内容。Linux Namesapce 在此基础之上，提供了对 UTS、IPC、Mount、PID、Network、User 等的隔离机制，如下所示。

分类	系统调用参数	相关内核版本
Mount Namespaces	CLONE_NEWNS	Linux 2.4.19
UTS Namespaces	CLONE_NEWUTS	Linux 2.6.19
IPC Namespaces	CLONE_NEWIPC	Linux 2.6.19
PID Namespaces	CLONE_NEWPID	Linux 2.6.19
Network Namespaces	CLONE_NEWNET	始于Linux 2.6.24 完成于 Linux 2.6.29
User Namespaces	CLONE_NEWUSER	始于 Linux 2.6.23 完成于 Linux 3.8)

> ★
> Linux Namespace 官方文档：Namespaces in operation
> ”

namespace 有三个系统调用可以使用：

`clone()` --- 实现线程的系统调用，用来创建一个新的进程，并可以通过设计上述参数达到隔离。

`unshare()` --- 使某个进程脱离某个 namespace

`setns(int fd, int nstype)` --- 把某进程加入到某个 namespace

下面使用这几个系统调用来演示 Namespace 的效果，更加详细地可以看 DOCKER基础技术：LINUX NAMESPACE（上）、 DOCKER基础技术：LINUX NAMESPACE（下）。

UTS Namespace

UTS Namespace 主要是用来隔离主机名的，也就是每个容器都有自己的主机名。我们使用如下的代码来进行演示。注意：假如在容器内部没有设置主机名的话会使用主机的主机名的；假如在容器内部设置了主机名但是没有使用 CLONE_NEWUTS 的话那么改变的其实是主机的主机名。

```
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <stdio.h>
#include <sched.h>
#include <signal.h>
#include <unistd.h>

#define STACK_SIZE (1024 * 1024)
```


分类专栏
拿下数据结构与算法 2篇
VUE 学习 5篇
杂文 3篇
可读代码编写炸鸡 9篇
Redis 从入门到放弃 5篇
Git 操作全系列 4篇
经典的 jQuery 2篇
爪哇 2篇

```c
static char container_stack[STACK_SIZE];

char* const container_args[] = {
    "/bin/bash",
    NULL
};

int container_main(void* arg) {
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container_dawn", 15);
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    printf("Parent [%5d] - start a container!\n", getpid());
    int container_id = clone(container_main, container_stack + STACK_SIZE,
                             CLONE_NEWUTS | SIGCHLD, NULL);
    waitpid(container_id, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

```
root@dawn ~/docker_ws/mydocker
# ./mycontainer_uts
Parent [31904] - start a container!
Container [31905] - inside the container!
root@container_dawn:~/docker_ws/mydocker# hostname
container_dawn
root@container_dawn:~/docker_ws/mydocker# exit
exit
Parent - container stopped!
root@dawn ~/docker_ws/mydocker
# hostname
dawn
```

PID Namespace

每个容器都有自己的进程环境中，也就是相当于容器内进程的 PID 从 1 开始命名，此时主机上的 PID 其实也还是从 1 开始命名的，就相当于有两个进程环境：一个主机上的从 1 开始，另一个容器里的从 1 开始。

为啥 PID 从 1 开始就相当于进程环境的隔离了呢？因此在传统的 UNIX 系统中，PID 为 1 的进程是 init，地位特殊。它作为所有进程的父进程，有很多特权。另外，其还会检查所有进程的状态，我们知道如果某个进程脱离了父进程（父进程没有 wait 它），那么 init 就会负责回收资源并结束这个子进程。所以要想做到进程的隔离，首先需要创建出 PID 为 1 的进程。

但是，【kubernetes 里面的话】

```c
int container_main(void* arg) {
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container_dawn", 15);
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    printf("Parent [%5d] - start a container!\n", getpid());
    int container_id = clone(container_main, container_stack + STACK_SIZE,
                             CLONE_NEWUTS | CLONE_NEWPID | SIGCHLD, NULL);
```

```
    waitpid(container_id, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

```
┌─root@dawn ~/docker_ws/mydocker
└─# ./mycontainer_pid
Parent [32020] - start a container!
Container [    1] - inside the container!
root@container_dawn:~/docker_ws/mydocker# echo $$
1
root@container_dawn:~/docker_ws/mydocker#
```

如果此时你在子进程的 shell 中输入 ps、top 等命令，我们还是可以看到所有进程。这是因为，ps、top 这些命令是去读 /proc 文件系统，由于此时文件系统并没有隔离，所以父进程和子进程通过命令看到的情况都是一样的。

IPC Namespace

常见的 IPC 有共享内存、信号量、消息队列等。当使用 IPC Namespace 把 IPC 隔离起来之后，只有同一个 Namespace 下的进程才能相互通信，因为主机的 IPC 和其他 Namespace 中的 IPC 都是看不到了的。而这个的隔离主要是因为创建出来的 IPC 都会有一个唯一的 ID，那么主要对这个 ID 进行隔离就好了。

想要启动 IPC 隔离，只需要在调用 clone 的时候加上 CLONE_NEWIPC 参数就可以了。

```
int container_main(void* arg) {
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container_dawn", 15);
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    printf("Parent [%5d] - start a container!\n", getpid());
    int container_id = clone(container_main, container_stack + STACK_SIZE,
                             CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWIPC | SIGCHLD, NULL);
    waitpid(container_id, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

Mount Namespace

Mount Namespace 可以让容器有自己的 root 文件系统。需要注意的是，在通过 CLONE_NEWNS 创建 mount namespace 之后，父进程会把自己的文件结构复制给子进程中。所以当子进程中不重新 mount 的话，子进程和父进程的文件系统视图是一样的，假如想要改变容器进程的视图，一定需要重新 mount（这个是 mount namespace 和其他 namespace 不同的地方）。

另外，子进程中新的 namespace 中的所有 mount 操作都只影响自身的文件系统（注意这边是 mount 操作，而创建文件等操作都是会有所影响的），而不对外界产生任何影响，这样可以做到比较严格地隔离（当然这边是除 share mount 之外的）。

下面我们重新挂载子进程的 `/proc` 目录，从而可以使用 ps 来查看容器内部的情况。

```
int container_main(void* arg) {
    printf("Container [%5d] - inside the container!\n", getpid());

    sethostname("container_dawn", 15);

    if (mount("proc", "/proc", "proc", 0, NULL) !=0 ) {
```

```
        perror("proc");
    }

    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    printf("Parent [%5d] - start a container!\n", getpid());
    int container_id = clone(container_main, container_stack + STACK_SIZE,
                             CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | SIGCHLD, NULL);
    waitpid(container_id, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

```
root@dawn ~/docker_ws/mydocker
# ./mycontainer_mount
Parent [32715] - start a container!
Container [    1] - inside the container!
root@container_dawn:~/docker_ws/mydocker# ps -elf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 -  7100 wait   15:33 pts/25   00:00:00 /bin/bash
0 R root          11       1  0  80   0 - 11108 -      15:33 pts/25   00:00:00 ps -elf
```

★

这里会有个问题就是在退出子进程之后，当再次使用 `ps -elf` 的时候会报错，如下所示

```
root@dawn ~/docker_ws/mydocker
# ./mycontainer_mount
Parent [  421] - start a container!
Container [    1] - inside the container!
root@container_dawn:~/docker_ws/mydocker# ps -elf
F S UID          PID    PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S root           1       0  0  80   0 -  7100 wait   15:51 pts/25   00:00:00 /bin/bash
0 R root          11       1  0  80   0 - 11108 -      15:51 pts/25   00:00:00 ps -elf
root@container_dawn:~/docker_ws/mydocker# exit
exit
Parent - container stopped!
root@dawn ~/docker_ws/mydocker
# ps -elf
Error, do this: mount -t proc proc /proc
root@dawn ~/docker_ws/mydocker
# mount -t proc proc /proc
```

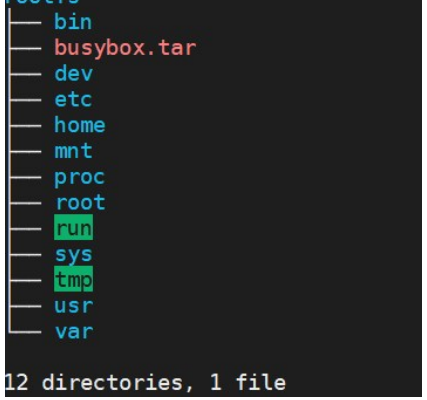
这是因为 /proc 是 share mount，对它的操作会影响所有的 mount namespace，可以看这里：
http://unix.stackexchange.com/questions/281844/why-does-child-with-mount-namespace-affect-parent-mounts

”

上面仅仅重新 mount 了 `/proc` 这个目录，其他的目录还是跟父进程一样视图的。一般来说，容器创建之后，容器进程需要看到的是一个独立的隔离环境，而不是继承宿主机的文件系统。接下来演示一个山寨镜像，来模仿 Docker 的 Mount Namespace。也就是给子进程实现一个较为完整的独立的 root 文件系统，让这个进程只能访问自己构成的文件系统中的内容（想想我们平常使用 Docker 容器的样子）。

首先我们使用 `docker export` 将 busybox 镜像导出成一个 rootfs 目录，这个 rootfs 目录的情况如图所示，已经包含了 `/proc`、`/sys` 等特殊的目录。

```
root@dawn ~/docker_ws/mydocker
# tree rootfs -L 1
rootfs
```

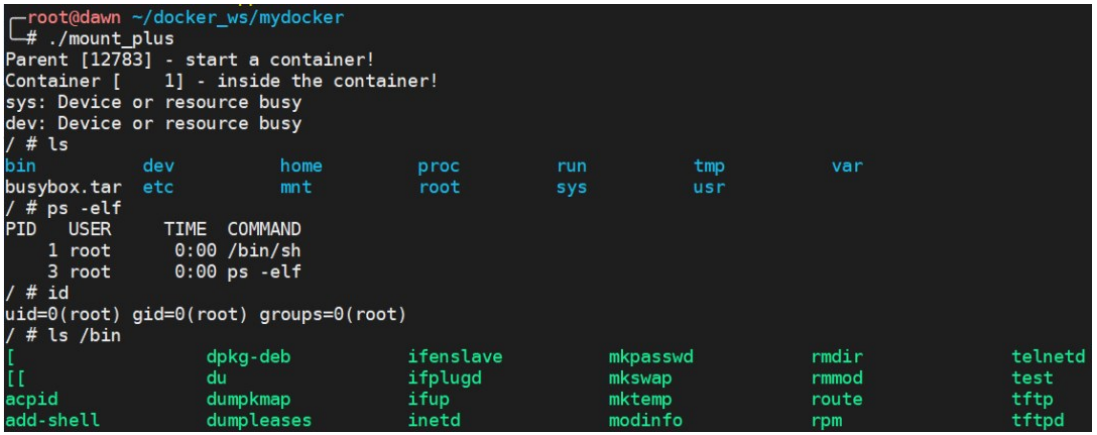


之后我们在代码中将一些特殊目录重新挂载，并使用 `chroot()` 系统调用将进程的根目录改成上文的 `rootfs` 目录。

```
char* const container_args[] = {
    "/bin/sh",
    NULL
};

int container_main(void* arg) {
    printf("Container [%5d] - inside the container!\n", getpid());
    sethostname("container_dawn", 15);

    if (mount("proc", "rootfs/proc", "proc", 0, NULL) != 0) {
        perror("proc");
    }
    if (mount("sysfs", "rootfs/sys", "sysfs", 0, NULL)!=0) {
        perror("sys");
    }
    if (mount("none", "rootfs/tmp", "tmpfs", 0, NULL)!=0) {
        perror("tmp");
    }
    if (mount("udev", "rootfs/dev", "devtmpfs", 0, NULL)!=0) {
        perror("dev");
    }
    if (mount("devpts", "rootfs/dev/pts", "devpts", 0, NULL)!=0) {
        perror("dev/pts");
    }
    if (mount("shm", "rootfs/dev/shm", "tmpfs", 0, NULL)!=0) {
        perror("dev/shm");
    }
    if (mount("tmpfs", "rootfs/run", "tmpfs", 0, NULL)!=0) {
        perror("run");
    }

    if ( chdir("./rootfs") || chroot("./") != 0 ){
        perror("chdir/chroot");
    }

    // 改变根目录之后，那么 /bin/bash 是从改变之后的根目录中搜索了
    execv(container_args[0], container_args);
    perror("exec");
    printf("Something's wrong!\n");
    return 1;
}

int main() {
```

```
    printf("Parent [%5d] - start a container!\n", getpid());
    int container_id = clone(container_main, container_stack + STACK_SIZE,
                             CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | SIGCHLD, NULL);
    waitpid(container_id, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

最后，查看实现效果如下图所示。



实际上，Mount Namespace 是基于 chroot 的不断改良才被发明出来的，chroot 可以算是 Linux 中第一个 Namespace。那么上面被挂载在容器根目录上、用来为容器镜像提供隔离后执行环境的文件系统，就是所谓的容器镜像，也被叫做 rootfs（根文件系统）。需要明确的是，**rootfs 只是一个操作系统所包含的文件、配置和目录，并不包括操作系统内核。**

User Namespace

容器内部看到的 UID 和 GID 和外部是不同的了，比如容器内部针对 dawn 这个用户显示的是 0，但是实际上这个用户在主机上应该是 1000。要实现这样的效果，需要把容器内部的 UID 和主机的 UID 进行映射，需要修改的文件是 `/proc/<pid>/uid_map` 和 `/proc/<pid>/gid_map`，这两个文件的格式是

```
ID-INSIDE-NS  ID-OUTSIDE-NS LENGTH
```

ID-INSIDE-NS ：表示在容器内部显示的 UID 或 GID

ID-OUTSIDE-NS：表示容器外映射的真实的 UID 和 GID

LENGTH：表示映射的范围，一般为 1，表示一一对应

比如，下面就是将真实的 uid=1000 的映射为容器内的 uid =0：

```
$ cat /proc/8353/uid_map
0       1000          1
```

再比如，下面则表示把 namesapce 内部从 0 开始的 uid 映射到外部从 0 开始的 uid，其最大范围是无符号 32 位整型（下面这条命令是在主机环境中输入的）。

```
$ cat /proc/$$/uid_map
0          0 4294967295
```

默认情况，设置了 CLONE_NEWUSER 参数但是没有修改上述两个文件的话，容器中默认情况下显示为 65534，这是因为容器找不到真正的 UID，所以就设置了最大的 UID。如下面的代码所示：

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/mount.h>
#include <sys/capability.h>
#include <stdio.h>
#include <sched.h>
#include <signal.h>
#include <unistd.h>

#define STACK_SIZE (1024 * 1024)

static char container_stack[STACK_SIZE];
char* const container_args[] = {
    "/bin/bash",
    NULL
};

int container_main(void* arg) {

    printf("Container [%5d] - inside the container!\n", getpid());

    printf("Container: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
            (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    printf("Container [%5d] - setup hostname!\n", getpid());

    //set hostname
    sethostname("container",10);

    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    const int gid=getgid(), uid=getuid();

    printf("Parent: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
            (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    printf("Parent [%5d] - start a container!\n", getpid());

    int container_pid = clone(container_main, container_stack+STACK_SIZE,
            CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWUSER | SIGCHLD, NULL);


    printf("Parent [%5d] - Container [%5d]!\n", getpid(), container_pid);

    printf("Parent [%5d] - user/group mapping done!\n", getpid());

    waitpid(container_pid, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
```

```
}
```

当我以 dawn 这个用户执行的该程序的时候，那么会显示如下图所示的效果。使用 root 用户的时候是同样的：

```
dawn@dawn:~$ ./user2
Parent: eUID = 1000;  eGID = 1000, UID=1000, GID=1000
Parent [ 8062] - start a container!
Parent [ 8062] - Container [ 8063]!
Container [    1] - inside the container!
Container: eUID = 65534;  eGID = 65534, UID=65534, GID=65534
nobody@container:~$
```

```
root@dawn ~/docker_ws/mydocker
# ./user
Parent: eUID = 0;  eGID = 0, UID=0, GID=0
Parent [ 8229] - start a container!
Parent [ 8229] - Container [ 8230]!
Container [    1] - inside the container!
Container: eUID = 65534;  eGID = 65534, UID=65534, GID=65534
nobody@container:~/docker_ws/mydocker$
```

接下去，我们要开始来实现映射的效果了，也就是让 dawn 这个用户在容器中显示为 0。代码是几乎完全拿耗子叔的博客上的，链接可见文末：

```c
int pipefd[2];

void set_map(char* file, int inside_id, int outside_id, int len) {
    FILE* mapfd = fopen(file, "w");
    if (NULL == mapfd) {
        perror("open file error");
        return;
    }
    fprintf(mapfd, "%d %d %d", inside_id, outside_id, len);
    fclose(mapfd);
}

void set_uid_map(pid_t pid, int inside_id, int outside_id, int len) {
    char file[256];
    sprintf(file, "/proc/%d/uid_map", pid);
    set_map(file, inside_id, outside_id, len);
}

int container_main(void* arg) {

    printf("Container [%5d] - inside the container!\n", getpid());

    printf("Container: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
            (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    /* 等待父进程通知后再往下执行（进程间的同步） */
    char ch;
    close(pipefd[1]);
    read(pipefd[0], &ch, 1);

    printf("Container [%5d] - setup hostname!\n", getpid());
    //set hostname
    sethostname("container",10);

    //remount "/proc" to make sure the "top" and "ps" show container's information
    mount("proc", "/proc", "proc", 0, NULL);

```

```
    execv(container_args[0], container_args);
    printf("Something's wrong!\n");
    return 1;
}

int main() {
    const int gid=getgid(), uid=getuid();

    printf("Parent: eUID = %ld;  eGID = %ld, UID=%ld, GID=%ld\n",
            (long) geteuid(), (long) getegid(), (long) getuid(), (long) getgid());

    pipe(pipefd);

    printf("Parent [%5d] - start a container!\n", getpid());

    int container_pid = clone(container_main, container_stack+STACK_SIZE,
            CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWUSER | SIGCHLD, NULL);


    printf("Parent [%5d] - Container [%5d]!\n", getpid(), container_pid);

    //To map the uid/gid,
    //   we need edit the /proc/PID/uid_map (or /proc/PID/gid_map) in parent
    set_uid_map(container_pid, 0, uid, 1);

    printf("Parent [%5d] - user/group mapping done!\n", getpid());

    /* 通知子进程 */
    close(pipefd[1]);

    waitpid(container_pid, NULL, 0);
    printf("Parent - container stopped!\n");
    return 0;
}
```

实现的最终效果如图所示，可以看到在容器内部将 dawn 这个用户 UID 显示为了 0（root），但其实这个容器中的 /bin/bash 进程还是以一个普通用户，也就是 dawn 来运行的，只是显示出来的 UID 是 0，所以当查看 /root 目录的时候还是没有权限。

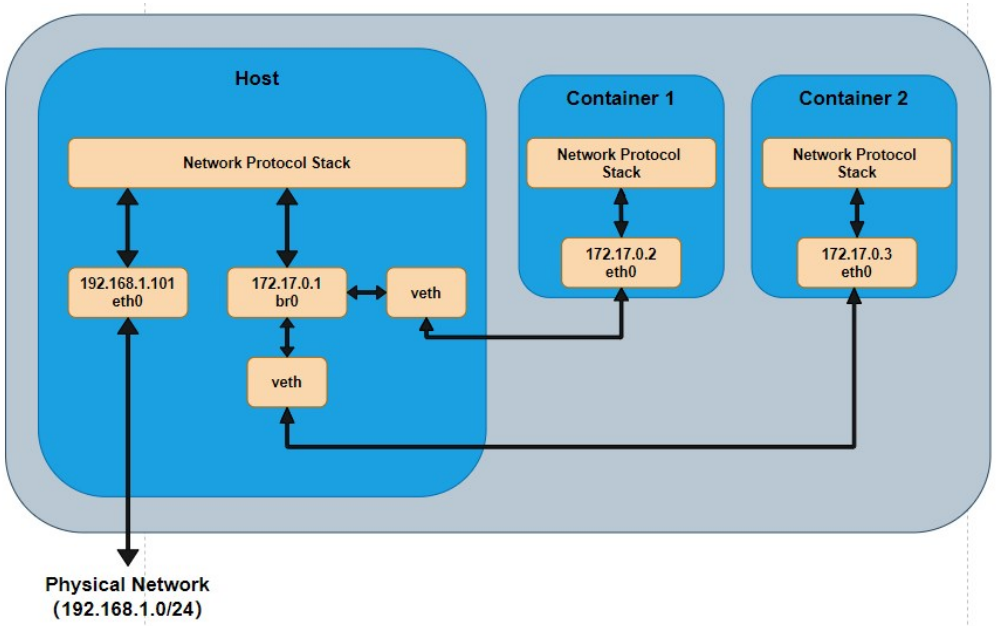
```
dawn@dawn:~$ ./user
Parent: eUID = 1000;  eGID = 1000, UID=1000, GID=1000
Parent [ 8325] - start a container!
Parent [ 8325] - Container [ 8326]!
Parent [ 8325] - user/group mapping done!
Container [    1] - inside the container!
Container: eUID = 0;  eGID = 65534, UID=0, GID=65534
Container [    1] - setup hostname!
root@container:~# id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
root@container:~# ls /root/
ls: cannot open directory '/root/': Permission denied
```

User Namespace 是以普通用户运行的，但是别的 Namespace 需要 root 权限，那么当使用多个 Namespace 该怎么办呢？我们可以先用一般用户创建 User Namespace，然后把这个一般用户映射成 root，那么在容器内用 root 来创建其他的 Namespace。

Network Namespace

隔离容器中的网络，每个容器都有自己的虚拟网络接口和 IP 地址。在 Linux 中，可以使用 ip 命令创建 Network Namespace（Docker 的源码中，它没有使用 ip 命令，而是自己实现了 ip 命令内的一些功能）。

下面就使用 ip 命令来讲解一下 Network Namespace 的构建，以 bridge 网络为例。bridge 网络的拓扑图一般如下图所示，其中 br0 是 Linux 网桥。



在使用 Docker 的时候，如果启动一个 Docker 容器，并使用 ip link show 查看当前宿主机上的网络情况，那么你会看到有一个 docker0 还有一个 `veth****` 的虚拟网卡，这个 veth 的虚拟网卡就是上图中 veth，而 docker0 就相当于上图中的 br0。

那么，我们可以使用下面这些命令即可创建跟 docker 类似的效果（参考自耗子叔的博客，链接见文末参考，结合上图加了一些文字）。

```
## 1. 首先，我们先增加一个网桥 lxcbr0，模仿 docker0
brctl addbr lxcbr0
brctl stp lxcbr0 off
ifconfig lxcbr0 192.168.10.1/24 up #为网桥设置IP地址

## 2. 接下来，我们要创建一个 network namespace ，命名为 ns1

# 增加一个 namesapce 命令为 ns1 （使用 ip netns add 命令）
ip netns add ns1

# 激活 namespace 中的 loopback，即127.0.0.1（使用 ip netns exec ns1 相当于进入了 ns1 这个 namespace，那么 ip link set
ip netns exec ns1   ip link set dev lo up

## 3. 然后，我们需要增加一对虚拟网卡

# 增加一对虚拟网卡，注意其中的 veth 类型。这里有两个虚拟网卡：veth-ns1 和 lxcbr0.1，veth-ns1 网卡是要被安到容器中的，而 lxcb
ip link add veth-ns1 type veth peer name lxcbr0.1

# 把 veth-ns1 按到 namespace ns1 中，这样容器中就会有一个新的网卡了
ip link set veth-ns1 netns ns1

# 把容器里的 veth-ns1 改名为 eth0 （容器外会冲突，容器内就不会了）
ip netns exec ns1  ip link set dev veth-ns1 name eth0

# 为容器中的网卡分配一个 IP 地址，并激活它
ip netns exec ns1 ifconfig eth0 192.168.10.11/24 up


# 上面我们把 veth-ns1 这个网卡按到了容器中，然后我们要把 lxcbr0.1 添加上网桥上
brctl addif lxcbr0 lxcbr0.1
```

```
31
32 # 为容器增加一个路由规则，让容器可以访问外面的网络
33 ip netns exec ns1     ip route add default via 192.168.10.1
34
35 ## 4. 为这个 namespace 设置 resolv.conf，这样，容器内就可以访问域名了
36 echo "nameserver 8.8.8.8" > conf/resolv.conf
```

上面基本上就相当于 docker 网络的原理，只不过：

> Docker 不使用 ip 命令而是，自己实现了 ip 命令内的一些功能。
>
> Docker 的 resolv.conf 没有使用这样的方式，而是将其写到指定的 resolv.conf 中，之后在启动容器的时候将其和 hostname、host 一起以只读的方式加载到容器的文件系统中。
>
> docker 使用进程的 PID 来做 network namespace 的名称。

同理，我们还可以使用如下的方式为正在运行的 docker 容器增加一个新的网卡

```
1 ip link add peerA type veth peer name peerB
2 brctl addif docker0 peerA
3 ip link set peerA up
4 ip link set peerB netns ${container-pid}
5 ip netns exec ${container-pid} ip link set dev peerB name eth1
6 ip netns exec ${container-pid} ip link set eth1 up
7 ip netns exec ${container-pid} ip addr add ${ROUTEABLE_IP} dev eth1
```

Namespace 情况查看

Cgroup 的操作接口是文件系统，位于 `/sys/fs/cgroup` 中。假如想查看 namespace 的情况同样可以查看文件系统，namespace 主要查看 `/proc/<pid>/ns` 目录。

我们以上面的 [PID Namespace 程序](#PID Namespace) 为例，当这个程序运行起来之后，我们可以看到其 PID 为 11702。

```
┌─root@dawn ~/docker_ws/mydocker
└─# ./mycontainer_pid
Parent [11702] - start a container!
Container [    1] - inside the container!
root@container_dawn:~/docker_ws/mydocker#
```

之后，我们保持这个子进程运行，然后打开另一个 shell，查看这个程序创建的子进程的 PID，也就是容器中运行的进程在主机中的 PID。

最后，我们分别查看 `/proc/11702/ns` 和 `/proc/11703/ns` 这两个目录的情况，也就是查看这两个进程的 namespace 情况。可以看到其中 cgroup、ipc、mnt、net、user 都是同一个 ID，而 pid、uts 是不同的 ID。如果两个进程的 namespace 编号相同，那么表示这两个进程位于同一个 namespace 中，否则位于不同 namespace 中。

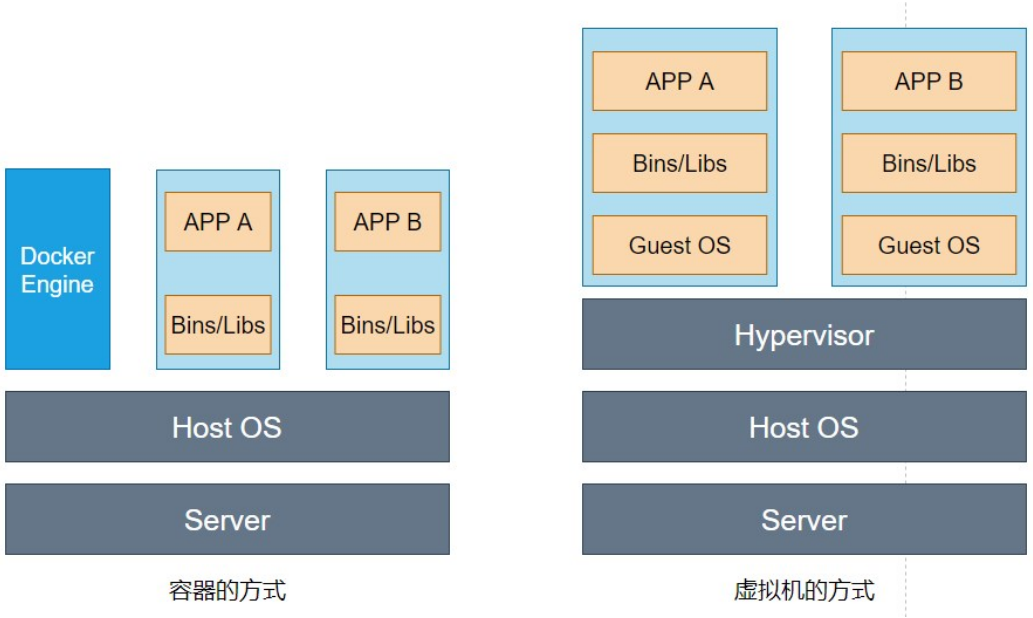
```
┌─root@dawn ~/docker_ws/mydocker
└─# pstree -p 11702
mycontainer_pid(11702)──bash(11703)
┌─root@dawn ~/docker_ws/mydocker
└─# ll /proc/11702/ns/
total 0
lrwxrwxrwx 1 root root 0 Oct  7 20:52 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 net -> net:[4026532009]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Oct  7 20:52 pid_for_children -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 uts -> uts:[4026531838]
```

```
└─# ll /proc/11703/ns/
total 0
lrwxrwxrwx 1 root root 0 Oct  7 20:52 cgroup -> cgroup:[4026531835]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 net -> net:[4026532009]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 pid -> pid:[4026532548]
lrwxrwxrwx 1 root root 0 Oct  7 20:52 pid_for_children -> pid:[4026532548]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Oct  7 20:51 uts -> uts:[4026532547]
```

如果可以查看 ns 的情况之外，这些文件一旦被打开，只要 fd 被占用着，即使 namespace 中所有进程都已经结束了，那么创建的 namespace 也会一直存在。比如可以使用 `mount --bind /proc/11703/ns/uts ~/uts`，让 11703 这个进程的 UTS Namespace 一直存在。

总结

Namespace 技术实际上修改了应用进程看待整个计算机“视图”，即它的”视图“已经被操作系统做了限制，只能”看到“某些指定的内容，这仅仅对应用进程产生了影响。**但是对宿主机来说，这些被隔离了的进程，其实还是进程，跟宿主机上其他进程并无太大区别，都由宿主机统一管理。只不过这些被隔离的进程拥有额外设置过的 Namespace 参数。**那么 Docker 项目在这里扮演的，更多是旁路式的辅助和管理工作。如下左图所示



容器的方式　　虚拟机的方式

因此，相比虚拟机的方式，容器会更受欢迎。这是假如使用虚拟机的方式作为应用沙盒，那么必须要由 Hypervisor 来负责创建虚拟机，这个虚拟机是真实存在的，并且里面必须要运行一个完整的 Guest OS 才能执行用户的应用进程。这样就导致了采用虚拟机的方式之后，不可避免地带来额外的资源消耗和占用。根据实验，一个运行着 CentOS 的 KVM 虚拟机启动后，在不做优化的情况下，虚拟机就需要占用 100-200 MB 内存。此外，用户应用运行在虚拟机中，它对宿主机操作系统的调用就不可避免地要经过虚拟机软件的拦截和处理，这本身就是一层消耗，尤其对资源、网络和磁盘 IO 的损耗非常大。

而假如使用容器的方式，容器化之后应用本质还是宿主机上的一个进程，这也就意味着因为虚拟机化带来的性能损耗是不存在的；而另一方面使用 Namespace 作为隔离手段的容器并不需要单独的 Guest OS，这就使得容器额外的资源占用几乎可以忽略不计。

总得来说，**“敏捷”和“高性能”是容器相对于虚拟机最大的优势**，也就是容器能在 PaaS 这种更加细粒度的资源管理平台上大行其道的重要原因。

但是！基于 Linux Namespace 的隔离机制相比于虚拟化技术也有很多不足之处，其中最主要的问题就是隔离不彻底。

首先，容器只是运行在宿主机上的一种特殊进程，那么容器之间使用的还是同一个宿主机上的操作系统。尽管可以在容器里面通过 mount namesapce 单独挂载其他不同版本的操作系统文件，比如 centos、ubuntu，但是这并不能改变共享宿主机内核的事实。这就意味着你要在 windows 上运行 Linux 容器，或者在低版本的 Linux 宿主机上运行高版本的 Linux 容器都是行不通的。

而拥有虚拟机技术和独立 Guest OS 的虚拟机就要方便多了。

其次，在 Linux 内核中，有很多资源和对象都是不能被 namespace 化的，比如时间。假如你的容器中的程序使用 `settimeofday(2)` 系统调用修改了时间，整个宿主机的时间都会被随之修改。

相比虚拟机里面可以随意折腾的自由度，在容器里部署应用的时候，“什么能做，什么不能做”是用户必须考虑的一个问题。之外，容器给应用暴露出来的攻击面是相当大的，应用“越狱”的难度也比虚拟机低很多。虽然，实践中可以使用 Seccomp 等技术对容器内部发起的所有系统调用进行过滤和甄别来进行安全加固，但这种方式因为多了一层对系统调用的过滤，也会对容器的性能产生影响。因此，在生产环境中没有人敢把运行在物理机上的 Linux 容器直接暴露到公网上。

另外，容器是一个“单进程”模型。容器的本质是一个进程，用户的应用进程实际上就是容器里 PID=1 的进程，而这个进程也是后续创建的所有进程的父进程。这也就意味着，在一个容器中，你没办法同时运行两个不同的应用，除非能事先找到一个公共的 PID=1 的程序来充当两者的父进程，比如使用 systemd 或者 supervisord。容器的设计更多是希望容器和应用同生命周期的，而不是容器还在运行，而里面的应用早已经挂了。

> ★
> 上面这段话个人的理解是：因为创建出子进程之后，子进程需要运行的，而此时父进程需要等待子进程运行结束，相当于只有子进程在运行。比如容器中的第一个进程往往就是业务需要的进程，也就是 entrypoint 指定的程序运行起来的进程。而创建出子进程会导致这个进程被暂停。即使将子进程改为后台执行，但是由于容器中 PID=1 的进程压根没有管理后台进程的能力，所以还是会有进程无法管理。
>
> ”

巨人的肩膀

极客时间---《深入剖析 Kubernetes》---张磊老师

DOCKER基础技术：LINUX NAMESPACE（上）

DOCKER基础技术：LINUX NAMESPACE（下）。

78 原创 | 8万+ 周排名 | 32万+ 总排名 | 1万+ 访问 | 等级
436 积分 | 14 粉丝 | 8 获赞 | 11 评论 | 68 收藏
私信 关注

搜博主文章

热门文章

毕业设计（基于TensorFlow的深度学习与研究）之总结概述篇 2522

乐高 42083 布加迪 Chiron（多图流量预警） 2080

国产积木---克尔维特（多图流量预警） 692

逻辑回归实战：从疝气病症预测病马的死亡率 480

一文讲透 Serverless 到底是什么 414

最新评论

毕业设计（基于TensorFlow的深度学习…
W001123456789: 百度盘里面的文档已经被删除了 https://pan.baidu.com/s/1jxCg…

毕业设计（基于TensorFlow的深度学习…
YOLO1314: 分享失效了、能麻烦楼主发一下吗？😘😘😘

Redis 有序集合使用的跳表到底是什么
不吃西红柿丶: 很棒，期待大佬后续作品

容器底层 --- 超细节的 Namespace 机制…
乎你: 服装赞助商---林品如，大佬也太秀了吧~~~

容器底层-UnionFS 工作原理-AUFS 和 D…
不吃西红柿丶: 我最近也在学这个，博文写的挺好，谢谢宝藏博主~

您愿意向朋友推荐“博客详情页”吗？

强烈不推荐 不推荐 一般般 推荐 强烈推荐

最新文章

说说 JVM 的类加载机制『非专业』

Redis 有序集合使用的跳表到底是什么

图图的存储、BFS、DFS（听说叠词很可爱）

登录 查看 6 条热评

关于容器底层,超详细的 Namespace 机制讲解_CSDN云计算 1-10
关于容器底层,超详细的 Namespace 机制讲解 来源| 多选参数 责编| 寇雪芹 头图| 下载于视觉中国 Namespace Linux Namespace 是 Linux 提供的一种内核…

玩坏docker笔记(十):容器底层技术cgroup+namespace_Big... 1-9
Linux实现这种方式的技术是namespace,namespace管理着docker host中全局唯一的资源,并可以让每个容器都觉得只有自己在使用他。换句话说,namesp…

深入理解容器技术——基础讲解 少年单排，记录点滴 1374
容器技术近年来越来越火，作为云原生技术的最底层基石，要开发云原生应用，就有必要对于容器技术有一个更加深入的了解。容器的概念特别多，dock…

Linux进程的管理与调度（二）--- Linux的命名空间 namespace 详解 |~~~热爱生活、努力学习的小伙汁~~~| 445
Linux进程的管理与调度（二）--- Linux的命名空间 namespaces 详解 Linux NameSpaces 机制提供了一种资源隔离方案。 PID，IPC，Network 等系统资…

容器底层基础概念_weixin_30267697的博客 8-3
分层构建 联合挂载的机制来实现 镜像包含启动容器所需要的文件系统及其内容 bootfs用来启动并引导用户空间 引导完成后...docker pull registry.cn-hb.co…

基于Docker及Kubernetes技术构建容器云（PaaS）平台概述 热门推荐 胡伟煌的博客 3万+
本文个人博客地址为：http://www.huweihuang.com/article/kubernetes/paas-based-on-docker&kubernetes/本文个人博客地址为：https://huwei…

linux namespace-之使用 shichaog的专栏 2万+
namespace命名空间-之应用   &n…

Docker核心原理之namespace 我不是大牛 1万+
Docker背后的内核知识 当谈论docker时，常常会聊到docker的实现方式。很多开发者都知道，docker容器本质上是宿主机的进程，Docker通过namespac…

一文彻底搞懂Docker中的namespace 神技圈子的博客 1313
什么是namespace namespace是对全局系统资源的一种封装隔离。这样可以让不同namespace的进程拥有独立的全局系统资源。这样改变一个namespac…

docker基础:从chroot理解namespace的隔离 知行合一 止于至善 1万+
docker与linux内核的两个重要特性关系无比密切：namespace和cgroup。namespace实现了资源的隔离，而cgroup实现了控制。而namespace中隔离分…

彻底搞懂容器技术的基石： namespace （上） 最新发布 k8s 生态 457
大家好，我是张晋涛。目前我们所提到的容器技术、虚拟化技术（不论何种抽象层次下的虚拟化技术）都能做到资源层面上的隔离和限制。对于容器技术…

容器学习笔记（二）Namespace tyro_blog的博客 222
Namespace 这一节，我给大家介绍docker的namespace是如何实现资源隔离的 Namespace其实是linux很早就有的一个功能吧，但是因为docker它才被更…

docker的user namespace功能 weixin_33728268的博客 1459
docker的user namespace功能： 默认情况下：容器默认是以root账号进入的，即使指定默认账号，用户也可以通过--user 0来切到root账号，容器里的roo…

docker系列--namespace解读 weixin_34345560的博客 1127
前言 理解docker，主要从namesapce，cgroups，联合文件，运行时(runC)，网络几个方面。接下来我们会花一些时间，分别介绍。 docker系列--names…

容器基础Namespace和Cgroups Leon的博客 1760
隔离-Namespace Namespace-进程空间 可以区分容器内进程和宿主机进程的PID ，Mount 、NetWork 容器是一组有着自己Namespace 参数 的 进程、目…

【实现简单的容器】- docker基础技术之namespace 风格色的博客 925
简介 docker是一个是用来linux namespace 和 cgroups 的虚拟化工具。 下面几个小节学习linux namespace技术，以及使用golang实现。 环境： ubuntu …

【docker 底层知识】Linux 内核namespace 原理 zhonglinzhang 7191
mount namespace 隔离文件系统挂载点对隔离文件系统提供支持，/proc/{pid}/mounts，/proc/{pid}/mountstats查看文件设备统计信息，包括挂载文件的名…

在linux中再添加一张网卡,给网络命名空间增加或删除网卡(容器通用) weixin_29679557的博客 172
上次学习了网络命名空间，那么如何进入容器的命名空间呢？运行的容器都对应一个PID，每个PID都有自己的命名空间。在/proc/[pid]/ns目录下包含该进…

Docker技术原理之Linux Namespace（容器隔离） __拂晓__的博客 8074
0.前言 首先要知道一个运行的容器，其实就是一个受到隔离和资源限制的Linux进程——对，它就是一个进程。而本文主要来探讨Docker容器实现隔离用…

容器底层实现技术Namespace/Cgroup 小胡子 70
Docker容器实现原理 Docker容器在实现上是通过namespace技术来进行进程隔离，通过cgroup技术实现容器进程可用资源的限制，当docker启动一个容…

2021年 5篇 2020年 82篇