

## 12 Docker 隔离的本质： namespace

更新时间：2020-08-17 14:00:22



人的一生可能燃烧也可能腐朽，我不能腐朽，我愿意燃烧起来！——奥斯特洛夫斯基

我们知道 Docker 或者说容器技术的一个核心优势就是资源隔离性，那么这篇文章我们就来看一下资源隔离技术的内核支持，也就是 namespace 技术。

### 1. namespace 简介

namespace 的中文一般翻译成命名空间，我们也可以将 linux 的 namespace 理解成一系列的资源的抽象的集合。每个进程都有一个 namespace 属性，进程的 namespace 可以相同。对于同属于一个 namespace 中进程，可以感知到彼此的存在和变化，而对外界的进程一无所知，而这正是 docker 所需要的。

关于 namespace 的更多技术，我们可以通过 linux 自带的 manpage 查看，链接在文末的参考链接里面。

### 2. namespace 种类

Linux 内核中提供了 6 中隔离支持，分别是：IPC 隔离、网络隔离、挂载点隔离、进程编号隔离、用户和用户组隔离、主机名和域名隔离。

Namespace	flag	隔离内容
IPC	CLONE_NEWIPC	IPC（信号量、消息队列和共享内存等）隔离
Network	CLONE_NEWNET	网络隔离（网络栈、端口等）
Mount	CLONE_NEWNS	挂载点（文件系统）

Namespace	flag	隔离内容
PID	CLONE_NEWPID	进程编号
User	CLONE_NEWUSER	用户和用户组
UTS	CLONE_NEWUTS	主机名和域名

每个进程都有一个 `namespace`, 在 `/proc/<pid>/ns` 下面, 下面是一个示例:

```
[root@xxx ns]# ls -al
total 0
dr-x--x--x 2 root root 0 Nov 3 16:16 .
dr-xr-xr-x 9 root root 0 Nov 3 15:50 ..
lrwxrwxrwx 1 root root 0 Nov 3 16:16 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 Nov 3 16:16 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 Nov 3 16:16 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 Nov 3 16:16 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 Nov 3 16:16 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Nov 3 16:16 uts -> uts:[4026531838]
```

如上图, 我们可以看到 `ns` 目录下共有 6 个 `link` 文件, 分别为 `ipc`, `mnt`, `net`, `pid`, `user`, `uts`, 分别对应了我们上面提到的 6 中隔离技术。对于我们直接运行宿主机上并且没有做资源隔离的进程, 这 6 个 `link` 文件指向的目标文件也都是-致的。而对于 `docker` 进程, `ns` 目录下的 `link` 文件和宿主机上的 `link` 文件是不一样的, 也就是说他们属于不同的 `namespace` 空间。

### 3. namespace api

我们可以通过 `Linux` 系统提供的系统调用来看一下 `namespace` 技术的使用细节。系统调用包括:

#### clone

`clone` 会创建一个新的进程, 函数原型如下:

```
#define __GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg, ...
          /* pid_t *ptid, void *newtls, pid_t *ctid */);
```

几个形参的意思分别是:

- **fn:** 新的进程执行的函数;
- **child\_stack:** 新的进程的栈空间;
- **flags:** 表示使用哪些 `CLONE_*` 标志位, 与 `namespace` 相关的参数主要包括 `CLONE_NEWPIC`、`CLONE_NEWS`、`CLONE_NEWWNET`、`CLONE_NEWPID`、`CLONE_NEWUSERS` 和 `CLONE_NEWUTS`, 分别对应不同的 `namespace`。

#### setns

`setns()` 的函数原型如下:

```
#define __GNU_SOURCE      /* See feature_test_macros(7) */
#include <sched.h>

int setns(int fd, int nstype);
```

我们可以通过系统调用 **setns()** 加入到一个已经存成在 **namespace** 中。这个 api 的一个实际使用例子就是我们执行 **docker exec** 命令进入到容器内部：在终端执行命令 **docker exec** 相当于 **fork** 一个子进程，然后将该进程加入到我们参数指定 **docker** 进程中，这样我们就得到了和 **docker** 进程内部一样的隔离视图了。

### unshare

**unshare()** 的函数原型如下：

```
#define _GNU_SOURCE
#include <sched.h>

int unshare(int flags);
```

**unshare** 相当于对当前进程进行隔离，我们不需要启动一个新的进程就可以启动隔离的效果。Linux 的 **unshare** 命令就是基于这个 api 来实现的。这里暂时就不做展开了。

### ioctl

**ioctl()** 的函数原型如下：

```
#include <sys/ioctl.h>

int ioctl(int fd, unsigned long request, ...);
```

其中 **fd** 是文件描述符，当 **fd** 指向 **ns** 文件的时候，我们就可以通过 **ioctl** 去获取一些 **namespace** 信息。这个系统调用 Docker 中也没有使用，所以这里限于篇幅，不再展开。感兴趣的同学可以参考这条 manpage: [ioctl\\_ns](#) 。

## 4. namespace 代码示例

下面我们通过几个代码 **demo**，来更深入地理解一下 **namespace** 技术。首先我们通过 **clone** 系统调用来创建一个进程隔离的子进程。

```
// 子进程的函数主题
int child_fn() {
    // system 函数可以让我们的程序执行 shell 命令
    system("mount -t proc proc /proc");
    system("ps aux");
    printf("child pid: %d\n", getpid());
    return 0;
}

int main() {
    // 子进程的栈空间大小
    int CHILD_STACK_SIZE = 1024 * 1024;

    // 子进程的栈空间
    char child_stack[CHILD_STACK_SIZE];

    // 创建子进程：
    // 1. child_fn 表示子进程的主题是函数 child_fn()
    // 2. child_stack + CHILD_STACK_SIZE 表示子进程的栈空间，其实就是局部变量 child_stack
    // 3. CLONE_NEWPID 表示子进程使用新的 PID namespace; SIGCHLD 表示接收信号
    int child_pid = clone(child_fn, child_stack + CHILD_STACK_SIZE, CLONE_NEWPID | SIGCHLD, NULL);

    // waitpid 表示父进程等待子进程退出。如果不加这行代码，父进程将会直接退出，子进程就变成了孤儿进程
    waitpid(child_pid, NULL, 0);
    return 0;
}
```

上面程序中 `child_fn` 就是我们子进程运行的函数实体。在子进程中我们先执行了 `/proc` 挂载，这么做的原因是 `ps` 命令是查看的 `/proc` 目录，如果我们创建了子进程之后而没有挂载 `/proc`，那么看到的还是原来的进程列表。这里我们先进行 `/proc` 目录挂载，然后执行 `ps`，执行结果如下：

```
[root@xxx ~]# gcc clone_pid.c
[root@xxx ~]# ./a.out
USER      PID %CPU %MEM   VSZ RSS TTY      STAT START  TIME COMMAND
root      1 0.0 0.0  5068  92 pts/2  S+ 17:13  0:00 ./a.out
root      3 0.0 0.0 151064 1792 pts/2  R+ 17:13  0:00 ps aux
child pid: 1
```

我们可以看到在进行了进程隔离的子进程空间中一号进程就是我们的子进程，并且看不到其他进程。

大家可以参考我上面的代码示例，进行其他的 `namespace` 相关操作。

## 5. 参考：

1. [namespace man-page](#)

}

← 11 动手实践：从 0 到 1 构建自己的 Docker 应用

13 Docker 资源限制的幕后主使：cgroup →