自己动手写容器...

这是本专栏的第二部分：容器篇，共 8 篇，帮助大家由浅入深地认识和掌握容器。前面，我为你介绍了容器生命周期和资源管理相关的内容，让你对容器有了更加灵活的控制。之后从进程的角度带你认识了容器以及容器的两项核心技术 cgroups 和 namespace。本篇，我们进入实践环节，自己动手写容器，以便让你对容器有更加深刻的理解和认识。

经过了前面内容的学习，想必你对 Docker 容器已经不那么陌生了。从容器的生命周期管理，到对容器的资源控制，以及从进程的角度对容器的剖析，再到前两篇的 cgroups 和 namespace，你是否想更进一步地理解容器呢？

本篇内容会分上下两篇，通过前面已经学习到的 namespace 和 cgroups 等知识，来自己实现容器。

容器的基本属性

要自己实现容器，我们应该对容器进行更明确地定义，以便确认需求。

首先，容器一定是具有隔离性的，具备了隔离性才能称之为“容器”；其次，容器的资源可以被管理；再者按照我们之前使用 Docker 容器的经验来看，容器应该可以从“镜像”启动；还有启动后的容器，应该可以具备网络连接；最后，我们可以进入到容器中执行命令。

按照我们前面学到的内容，namespace 和 cgroups 肯定是实现资源的隔离和管理的关键技术；其他的问题就是镜像以及和网络相关的部分了。是不是这样拆解后感觉蛮简单的？我们开始逐步去实现它。

创建隔离环境

上一篇已经介绍过，我们可以通过使用 unshare 这个工具来很简单地达到隔离 namespace 资源的目的。本篇我们同样使用它来创建隔离环境。

我们传递对应参数给 unshare，将 Mount、PID、Network、IPC 及 UTS namespace 进行隔离：

复制

```
(MoeLove) ➜  ~ sudo unshare -imnpu --mount-proc --fork zsh
[sudo] tao 的密码：
[root@bogon]/home/tao# ps -ef
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 18:22 pts/28   00:00:00 zsh
root          28     1  0 18:22 pts/28   00:00:00 ps -ef
```

上面通过 ps 命令来验证 PID namespace 的隔离，下面通过 ip 命令来验证 Network namespace 的隔离：

```
[root@bogon]/home/tao# ip a
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
```

复制

接下来，我们也验证下 IPC namespace 是否成功隔离吧。

我们可以很简单的在主机上通过 `ipcmk -Q` 来创建一个消息队列，并通过 `ipcs -q` 进行查看：

```
(MoeLove) ➜  ~ ipcmk -Q
消息队列 id：0
(MoeLove) ➜  ~ ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者  权限     已用字节数 消息
0xe1f61b0c 0          tao        644        0            0
```

复制

当进入到刚才的通过 unshare 创建的隔离环境中，再通过 `ipcs -q` 命令便会发现查不到刚才的队列，IPC namespace 成功隔离。

```
[root@bogon]/home/tao# ipcs -q

--------- 消息队列 -----------
键         msqid      拥有者  权限     已用字节数 消息
```

复制

当然，你也会发现，此刻的隔离环境与我们通常使用 Docker 启动的容器环境并不相同，其中最大的区别是，当前我们仍然在使用主机的根文件系统（或者说，我们现在仍然能访问到主机的文件或目录）。

隔离主机的根文件系统

在本专栏的第一篇《Docker 的前世今生》中，我便为你介绍过一个有用的技术 **chroot**，使用 chroot 可以改变进程可见的根目录。我们使用它来完成我们的目标。

首先我们需要获得一个最基本的根文件系统，这里我们以体积很小的 Alpine Linux 为例。我们可以直接使用 Docker 从 Alpine Linux 的 Docker 镜像中提取其 rootfs。

复制

```
# 以下内容为在主机上操作
# 先创建一个空目录
(MoeLove) ➜  ~ mkdir alpine
# 进入该目录
(MoeLove) ➜  ~ cd alpine
# 使用 docker pull 镜像
(MoeLove) ➜  alpine docker pull alpine:3.9
3.9: Pulling from library/alpine
Digest: sha256:7746df395af22f04212cd25a92c1d6dbc5a06a0ca9579a229ef43008d4d1302a
Status: Image is up to date for alpine:3.9
docker.io/library/alpine:3.9
# 使用 docker save 来将镜像保存为 alpine.tar
(MoeLove) ➜  alpine docker save -o alpine.tar alpine:3.9
# 创建一个目录并将镜像解压至其中
(MoeLove) ➜  alpine mkdir image
(MoeLove) ➜  alpine tar -C image -xvf alpine.tar
055936d3920576da37aa9bc460d70c5f212028bda1c08c0879aedf03d7a66ea1.json
3accb6cdfe3a90f13a043b579d584d2aa7f7e3b9ea2486531658d8ac43b7bb48/
3accb6cdfe3a90f13a043b579d584d2aa7f7e3b9ea2486531658d8ac43b7bb48/VERSION
3accb6cdfe3a90f13a043b579d584d2aa7f7e3b9ea2486531658d8ac43b7bb48/json
3accb6cdfe3a90f13a043b579d584d2aa7f7e3b9ea2486531658d8ac43b7bb48/layer.tar
manifest.json
repositories
# 创建新目录用于存放 alpine 的 rootfs
(MoeLove) ➜  alpine mkdir rootfs
(MoeLove) ➜  alpine tar -C rootfs -xf image/3accb6cdfe3a90f13a043b579d584d2aa7f7e
(MoeLove) ➜  alpine ls rootfs
bin  dev  etc  home  lib  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  u
```

至此，我们在主机上已经得到了 Alpine Linux 的 rootfs，至于为何我们可以通过上述方法拿到其正确的 rootfs，我会在下一部分“镜像篇”中为你介绍，此处我们先回到本篇的重点，用此处得到的 rootfs 作为隔离环境的根文件系统。

接下来我们回到刚才的隔离环境中，执行 chroot 命令：

复制

```
# 以下内容是在隔离环境中执行的
[root@bogon]/home/tao# cd alpine
[root@bogon]/home/tao/alpine# chroot rootfs /bin/sh
(MoeLove) ➜  ~ cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.9.4
PRETTY_NAME="Alpine Linux v3.9"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

可以看到，当使用从 Alpine Linux 的 Docker 镜像中提取出来的 rootfs 执行 chroot 后，现在查看 /etc/os-release 已经可以看到当前是在 Alpine Linux 中了。

PS：也许你会好奇我上面输出中的 `(MoeLove) ➜ ~` 前缀，这是因为我的 PS1 变量设置的就是这个值：

复制

```
# 在隔离环境中
(MoeLove) ➜  ~ echo $PS1
(MoeLove) ➜ ~
```

接下来，我们尝试创建个文件，看看会有什么效果：

复制

```
# 在隔离环境中
(MoeLove) ➜  ~ echo 'in container' > note
(MoeLove) ➜  ~ cat note
in container
```

在主机中，看看 rootfs 目录中的变化：

复制

```
# 在主机上执行
(MoeLove) ➜  ~ ls alpine/rootfs
bin  dev  etc  home  lib  media  mnt  note  opt  proc  root  run  sbin  srv  sys
(MoeLove) ➜  ~ cat alpine/rootfs/note
in container
```

由此可以看到，容器中文件的变化，实际上也会反映到主机的文件系统中；另外还有一个值得注意的点：我们所创建的容器，它的根文件系统便是我们上面示例中的 rootfs 目录，如果需要创建多个“容器”，为了不干扰彼此的文件系统，则需要相应数量的 rootfs 目录，这样对于我们的存储而言消耗还是比较大的（线性增长）。

所以 Docker 在处理这部分的时候，使用了 Union file system 的技术来进行存储资源的重利用，这些内容在后续的“存储篇”中会具体介绍，本篇暂且跳过。

总结

在本篇中，我为你介绍了如何自己来创建一个容器的第一部分，这里我们主要是利用了 Linux 现有的工具和技术，并没有自己单独去写代码实现它。这样做的目的主要是为了让我们的注意力更多地集中于“容器”是如何创建的，或者说如何利用前面的知识来实现它。

在这其中，我们也引出来了**“镜像”**，以及“存储篇”将会深入介绍的存储驱动相关的知识，不过这些会在后续内容中逐步深入，本篇中就暂且略过了。

现在已经成功地创建出来了一个独立的隔离环境，下一篇，我们会继续完善这个“容器”，为它做资源管理和为它配置网络等。