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。上篇和本篇，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。

在上一篇中，我们利用 namespace 和 chroot 等技术，创建出了一个隔离的环境。但是距离我们预期的"容器"形态，还差了一些。本篇，我们将继续完善它，进而达到我们的预期。

使用 cgroups 进行资源管理

在之前的内容《容器的核心：cgroups》中，我为你介绍了 Docker 如何使用 cgroups 进行资源管理，同时也大致介绍了 cgroups 相关的概念和对配置文件的解读等，但并没有介绍在脱离 Docker 时如何操作 cgroups，本篇让我们一起来探索它吧！

预装依赖

首先，我们来安装一个工具，以便于我们后续的实践：

CentOS/Fedora：

```
sudo yum install -y libcgroup-tools
```

复制

Debian/Ubuntu：

```
sudo apt-get install -y cgroup-bin
```

复制

其他发行版，请自行搜索查询。

创建 cgroups

通过前面的学习，我们知道了容器的资源管理或限制是通过 cgroups 完成的。为了让我们的隔离环境更完善，我们来创建一个可用于限制内存和 CPU 的 cgroups 作为演示。

前面的工具，提供了很多有用的功能，我们这里用到的是 cgcreate，用它来创建一个新的 cgroups。

```
(MoeLove) ➜  ~ sudo cgcreate -g cpu,cpuacct,memory:customized_container
```

根据前面讲过的内容，我们来查看下刚才创建的 cgroups：

```
(MoeLove) ➜  ~ sudo ls /sys/fs/cgroup/cpu,cpuacct/customized_container
cgroup.clone_children  cpuacct.usage_all          cpuacct.usage_sys   cpu.shares
cgroup.procs           cpuacct.usage_percpu       cpuacct.usage_user  cpu.stat
cpuacct.stat           cpuacct.usage_percpu_sys   cpu.cfs_period_us   notify_on_re
cpuacct.usage          cpuacct.usage_percpu_user  cpu.cfs_quota_us    tasks
(MoeLove) ➜  ~ sudo ls   /sys/fs/cgroup/memory/customized_container
cgroup.clone_children           memory.kmem.tcp.max_usage_in_bytes  memory.oom_con
cgroup.event_control            memory.kmem.tcp.usage_in_bytes      memory.pressur
cgroup.procs                    memory.kmem.usage_in_bytes          memory.soft_li
memory.failcnt                  memory.limit_in_bytes               memory.stat
memory.force_empty              memory.max_usage_in_bytes           memory.swappin
memory.kmem.failcnt             memory.memsw.failcnt                memory.usage_i
memory.kmem.limit_in_bytes      memory.memsw.limit_in_bytes         memory.use_hie
memory.kmem.max_usage_in_bytes  memory.memsw.max_usage_in_bytes     notify_on_rele
memory.kmem.slabinfo            memory.memsw.usage_in_bytes         tasks
memory.kmem.tcp.failcnt         memory.move_charge_at_immigrate
memory.kmem.tcp.limit_in_bytes  memory.numa_stat
```

可以看到，通过使用 cgcreate 创建的 customized_container 相关的文件都已经自动创建好了。

配置 cgroups

之前在《容器资源管理》和《容器的核心：cgroups》两篇中，都有介绍过如何使用 Docker 设置容器可用的 CPU 及内存资源，也介绍过验证方法，这里就不再赘述了。我们直接为隔离环境配置 cgroups，配置 0.5 CPU 以及 10M 内存的限制。

先来看看当前 cgroups 中 CPU 的配置：

```
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/cpu,cpuacct/customized_container/cpu.cfs_
100000
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/cpu,cpuacct/customized_container/cpu.cfs_
-1
```

可以看到现在没有对 CPU 的配额进行限制。我们可以使用 cgset 来完成此配置（直接编辑配置文件也可以，但使用工具会更直观一些）：

复制

```
(MoeLove) ➜  ~ sudo cgset -r cpu.cfs_quota_us=50000 customized_container
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/cpu,cpuacct/customized_container/cpu.cfs_
50000
```

所以现在可用的 CPU 便是 50000/100000=0.5 了。

至于内存的话，也可以用同样的方式来完成：

复制

```
# 查看此 cgroups 中内存的限制
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/memory/customized_container/memory.limit_
9223372036854771712
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/memory/customized_container/memory.memsw.
9223372036854771712
```

限制它的可用内存为 10M，无可用 swap：

复制

```
# 限制可用内存为 10M
(MoeLove) ➜  ~ sudo cgset -r  memory.limit_in_bytes=10485760 customized_container
(MoeLove) ➜  ~ sudo cgset -r  memory.memsw.limit_in_bytes=10485760 customized_con
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/memory/customized_container/memory.limit_
10485760
(MoeLove) ➜  ~ sudo cat  /sys/fs/cgroup/memory/customized_container/memory.memsw.
10485760
```

在 cgroups 中运行隔离环境

接下来，我们使用 cgexec 在已经配置好的 cgroups 下运行隔离环境，以便达到对隔离环境的资源控制。

```
(MoeLove) ➜  ~ sudo cgexec -g cpu,cpuacct,memory:customized_container unshare -im
[sudo] tao 的密码：
[root@bogon]/home/tao# cd alpine
[root@bogon]/home/tao/alpine# chroot rootfs /bin/sh
(MoeLove) ➜  ~
(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/"
```

当前已经在 cgroups 控制的隔离环境中了，现在我们可以执行一个死循环用来测试之前的设置是否生效，比如 CPU：

```
# 写一个死循环
(MoeLove) ➜  ~ i=0; while true; do i=i+i; done
```

在**主机**上，通过 top 命令便可看到当前脚本执行的情况了：

```
# 只保留了当前程序的信息
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
29626 root      20   0    1596   1072    876 R  49.8   0.0   3:09.00 /bin/sh
```

可以看到，这里的 CPU 占用也就一直在 50% 左右了。

说明我们的配置已经生效，我们创建的隔离环境已经在受 cgroups 的资源限制了。

为容器配置网络

由于我们也为此环境隔离了 Network namespace（即：传递给 unshare 的 -n 参数），所以当前的隔离环境中默认是无法访问外部网络的。从以下的输出可以看到这里只有 lo 接口，而且是 DOWN 状态。

```
(MoeLove) ➜  ~ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

当我们想要为此隔离环境配置隔离的网络时，我们可以新建一个 Network namespace，为其配置好网络，然后让隔离环境使用该 Network namespace 便可。

我们使用 ip 工具来进行网络配置，它在 iproute2 包中，如果发现未安装，可按下述方式进行安装：

复制

```
# CentOS/Fedora
(MoeLove) ➜  ~ sudo yum install -y iproute

# Debian/Ubuntu
(MoeLove) ➜  ~ sudo yum install -y iproute2
```

创建 Network namespace

我们可以通过 `ip netns` 的相关子命令完成对 Network namespace 相关的操作。我们可以使用以下命令创建一个 Network namespace：

复制

```
(MoeLove) ➜  ~ sudo ip netns add new_container
```

这里所传递的名字即是 Network namespace 的名字，执行此命令后，会在 /var/run/netns 目录下创建一个对应的文件：

复制

```
(MoeLove) ➜  ~ ls -l --time-style='+' /var/run/netns
total 0
-r--r--r--. 1 root root 0  new_container
```

我们也可以用以下命令来直接查看已创建的 Network namespace：

复制

```
(MoeLove) ➜  ~ sudo ip netns list
new_container
```

配置 veth-pair

由于我们的隔离环境中，除 lo 外没有其他的网络接口，所以我们使用 veth-pair 来做，它实际就是一对虚拟设备接口。（对于网络部分，我们在后续的内容中也会有更加详细的讲述的）

另外，我们在创建完 veth-pair 后，将它与刚才创建好的 Network namespace 相关联：

复制

```
# 创建 veth-pair
(MoeLove) ➜  ~ sudo ip link add veth1 type veth peer name br-veth1
```

创建完成后，使用 `ip link ls` 可检查刚创建的设备：

复制

```
# 省略了一些输出
(MoeLove) ➜  ~ sudo ip link ls
...
35: br-veth1@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN m
    link/ether 66:2b:fe:7a:0a:3b brd ff:ff:ff:ff:ff:ff
36: veth1@br-veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN m
    link/ether 42:ec:b0:db:df:bc brd ff:ff:ff:ff:ff:ff
```

接下来，将设备于 Network namespace 相关联：

复制

```
# 将 veth-pair 与 Network namespace 相关联
(MoeLove) ➜  ~ sudo ip link set veth1 netns new_container
```

我们再来查看下所有的连接：

复制

```
# 省略了部分输出
(MoeLove) ➜  ~ sudo ip link ls
...
35: br-veth1@if36: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAU
    link/ether 66:2b:fe:7a:0a:3b brd ff:ff:ff:ff:ff:ff link-netns new_container
...
```

可以看到，当我们把 veth-pair 与 Network namespace 关联之后，能直接看到的结果就少了一个。这里的原因是因为它被放入了对应的那个 namespace 中了。

再次查看 `ip netns list` 也会看到此时的状态有所不同。

复制

```
(MoeLove) ➜  ~ ip netns list
new_container (id: 0)
```

查看 namespace 中的状态：

复制

```
(MoeLove) ➜  ~ sudo ip netns exec new_container ip a list
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
36: veth1@if35: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group defaul
    link/ether 42:ec:b0:db:df:bc brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

可以看到已经有了可用于连接的 veth1 接口了。但我们也会发现，该设备上还没配置过 IP，我们使用 `ip addr add` 进行添加：

```
(MoeLove) ➜  ~ sudo ip netns exec new_container ip addr add 192.168.1.11/24 dev v
# 检查结果
(MoeLove) ➜  ~ sudo ip netns exec new_container ip a list
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
36: veth1@if35: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group defaul
    link/ether 42:ec:b0:db:df:bc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.1.11/24 scope global veth1
       valid_lft forever preferred_lft forever
```

到这里为止，虽然我们已经为 veth-pair 配置了 IP 等信息，但我们的目标是让该“隔离环境”可以具备网络访问，很明显现在还不够（可以用 ping 测试下），我们需要再配置一个网桥，用于组网使用。

```
# 创建名为 br1 的网桥
(MoeLove) ➜  ~ sudo ip link add name br1 type bridge
# 设置它为 UP 状态
(MoeLove) ➜  ~ sudo ip link set br1 up
```

接下来，我们将 namespace 中的网络接口也都启用：

```
# 设置为 UP 状态;
(MoeLove) ➜  ~ sudo ip link set br-veth1 up
# 将 namespace 中的网卡也都启动
(MoeLove) ➜  ~ sudo ip netns exec new_container ip link set veth1 up
# 将网卡绑定到网桥上
(MoeLove) ➜  ~ sudo ip link set br-veth1 master br1
#  最后使用 bridge 检查一下配置
(MoeLove) ➜  ~ bridge link show br1
5: virbr0-nic: <BROADCAST,MULTICAST> mtu 1500 master virbr0 state disabled priori
35: br-veth1@if36: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br1 state fo
```

现在对于我们而言，网桥还没设置地址，所以还无法正常访问。

复制

```
# 设置网桥地址
(MoeLove) ➜  ~ sudo ip addr add 192.168.1.10/24 brd + dev br1

# 检查路由表是否已经正确写入
(MoeLove) ➜  ~ ip r|grep 192.168.1.10
192.168.1.0/24 dev br1 proto kernel scope link src 192.168.1.10
```

当上述内容都已经配置完成后，我们需要进行个简单的测试：

复制

```
(MoeLove) ➜  ~ sudo ip netns exec new_container 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
36: veth1@if35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 42:ec:b0:db:df:bc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.1.11/24 scope global veth1
       valid_lft forever preferred_lft forever
    inet6 fe80::40ec:b0ff:fedb:dfbc/64 scope link
       valid_lft forever preferred_lft forever
```

可以看到现在的 Network namespace 的 IP 是 192.168.1.11 在本地使用 ping 来做做测试。

复制

```
# 在主机上进行测试
(MoeLove) ➜  ~ ping -c 1 192.168.1.11
PING 192.168.1.11 (192.168.1.11) 56(84) bytes of data.
64 bytes from 192.168.1.11: icmp_seq=1 ttl=64 time=0.063 ms

--- 192.168.1.11 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.063/0.063/0.063/0.000 ms
```

可以看到主机上已经可以成功 ping 通了。

同时，我们也为它把路由表写上：

复制

```
(MoeLove) ➜  ~ sudo ip netns exec new_container ip route 192.168.1.0/24 dev veth1
```

现在它可以顺利地和主机进行通信了，但是很明显，它由于缺乏默认路由，现在是访问不到其他地方的。我们来为它增加默认路由：

```
(MoeLove) ➜  ~ sudo ip netns exec new_container ip route add default via 192.168.
```

但是，现在就结束了吗？其实还缺少一步。我们现在数据可以出去，但是却无法回到 192.168.1.0/24 这个网络，所以我们需要写一条 iptables 的 NAT 规则。

```
(MoeLove) ➜  ~ sudo iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -j MASQUERAD
```

当然，除了增加这条规则外，我们也需要确认下是否开启了数据包转发：

```
(MoeLove) ➜  ~ sudo sysctl -w net.ipv4.ip_forward=1
```

最终结果：我们使用当前的 new_container Network namespace 时候，与主机完全隔离，并且可以与外部网络互通。

```
(MoeLove) ➜  ~ sudo ip netns exec new_container ping -c 1 1.1.1.1
PING 1.1.1.1 (1.1.1.1) 56(84) bytes of data.
64 bytes from 1.1.1.1: icmp_seq=1 ttl=51 time=290 ms

--- 1.1.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 289.579/289.579/289.579/0.000 ms
```

将 Network namespace 应用于“隔离环境”

复制

```
(MoeLove) ➜  ~ sudo cgexec -g cpu,cpuacct,memory:customized_container ip netns ex
[sudo] tao 的密码：
[root@bogon]/home/tao# cd alpine
[root@bogon]/home/tao/alpine# chroot rootfs /bin/sh
(MoeLove) ➜  ~
(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/"
(MoeLove) ➜  ~ ip r
default via 192.168.1.10 dev veth1
192.168.1.0/24 dev veth1 proto kernel scope link src 192.168.1.11
[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
36: veth1@if35: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether 42:ec:b0:db:df:bc brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.1.11/24 scope global veth1
       valid_lft forever preferred_lft forever
    inet6 fe80::40ec:b0ff:fedb:dfbc/64 scope link
       valid_lft forever preferred_lft forever
```

所以经过两个小节，我们使用了一些已有的工具，自己来动手实现了个“容器”环境，基本形态就如上所示。先创建好 Network namespace，接下来通过 cgexec 进入 cgroups 的管理中，之后使用该 namespace 同时在 unshare 时，不需要在添加 Network namespace 的隔离即可。

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二部分，这里我们主要是利用了 Linux 现有的工具和技术，这样做的目的主要是为了让我们的注意力更多的集中于“容器”是如何创建的，或者说如何利用前面的知识来实现它。

在本篇，我们的重点分两大类：cgroups 和 Network namespace。

- -在 cgroups 的部分，通过实际的动手实践，可以便于掌握如何控制 cgroups；
- 而在 Network namespace 的部分，则涉及了一些网络相关的知识。如果你对这块内容不是很熟悉，也请不用担心，这里的内容实际上算是为之后的内容做铺垫。

在后续的**网络篇**的部分，我会再详细地为你介绍，以及这种模型在 Docker 和 Kubernetes 中是如何工作及应用的。

至此，本课程的第二部分“容器篇”到此就要告一段落了，接下来，我们会进入到“镜像篇”的学习，我会为你介绍容器的生命周期和镜像的本质，镜像的分发及用户认证授权的逻辑等内容。