云计算实验报告：dockee 容器引擎

1 实验背景

1.1 容器技术的意义和应用

容器技术是一种轻量级的虚拟化方法，它允许开发者将应用和其依赖环境封装在一个轻量级、可移植的容器镜像中，然后可以在任何支持容器技术的主机上运行。这种技术的出现，极大地简化了应用的部署、扩展和管理过程。

容器技术的应用场景广泛，它在软件开发和运维的多个方面都发挥着重要作用，以下是一些主要的应用场景。

首先是微服务架构。在微服务架构中，容器技术允许将复杂的应用拆分成一系列小型、独立的服务，每个服务都封装在自己的容

器中。这种模式提高了应用的可维护性、可扩展性和灵活性。容器使得服务之间的依赖关系更加清晰，便于管理和更新。

其次是云原生应用。容器技术是构建云原生应用的核心，它支持应用在云环境中的快速部署、弹性伸缩和高可用性。云原生应用通常设计为分布式系统，能够充分利用云计算资源的弹性和可扩展性，而容器技术则为这些应用提供了轻量级、可移植的运行环境。

然后是资源的调度和优化。容器技术通过共享宿主机的内核，减少了资源的占用和浪费。容器可以快速启动和停止，使得资源分配更加灵活和高效。此外，容器编排工具如 Kubernetes 可以自动管理容器的生命周期，优化资源的使用，确保应用的性能和稳定性。

最后是持续集成和持续部署（CI/CD）。容器技术在 CI/CD 流程中扮演着关键角色。开发者可以快速构建容器镜像，实现代码的自动化测试和部署。容器化的应用可以轻松地在不同的环境之间迁移，确保了开发、测试和生产环境的一致性。这大大加快了软件交付的速度，提高了开发效率。

1.2 容器技术的基本原理

容器技术的基本原理是通过操作系统级别的虚拟化来隔离应用和其依赖环境。它不依赖于传统的虚拟机技术，而是利用宿主机的内核来实现资源的隔离和限制。下面是实现容器引擎的几个关键机制。

首先是 Linux Namespace 机制。Namespace 提供了一种隔离机制，使得容器内的进程看起来像是在它们自己的独立系统中运行。每个容器可以有自己的网络、用户、进程 ID 等，而与宿主机

和其他容器隔离。Linux 内核提供了多种类型的 Namespace，包括但不限于网络、进程、用户等。

其次是 Linux cgroup（Control Groups）机制。cgroup 是一种资源管理机制，它允许管理员对进程组进行资源限制和优先级分配。通过 cgroup，容器引擎可以限制容器使用的 CPU、内存、磁盘 I/O 等资源，确保容器不会消耗过多宿主机资源，影响其他容器或宿主机的稳定性。

然后是 Chroot 机制。Chroot 是一种改变根目录的机制，使得容器内的文件系统看起来像是从指定的目录开始。通过 Chroot，容器可以拥有自己的文件系统视图，与宿主机的文件系统隔离。Chroot 常常与 Overlay 等文件系统配合使用，在这样的方式下，容器的根文件系统通常是一个只读的模板，容器的任何更改都写入到一个可写层中。

最后是容器镜像机制。容器镜像是一个轻量级的软件包，可以理解为容器的快照。容器镜像包含了运行容器所需的所有内容，包括代码、运行时、库、环境变量和配置文件。容器镜像可以快速构建、分发和部署，是容器化应用的基础。

通过以上这些机制，容器引擎能够创建一个轻量级、隔离的运行环境，使得应用可以在不同的环境中一致地运行，同时提供资源管理和安全性。容器技术的这些特性使其成为现代软件开发和运维的重要工具。

2 实验过程

2.1 进程隔离

Linux Namespace 是一种内核功能,通过它可以将一系列的系统资源(如 PID, User ID, Network 等)隔离开来。

每种 Namespace 类型负责隔离特定的资源,从而使得一个进程在某些方面仿佛运行在一个独立的系统中。

2.2 文件系统隔离

OverlayFS 是一种和 AUFS 很类似的文件系统,与 AUFS 相比,OverlayFS 有以下特性:1. 更简单地设计;2. 从 3.18 开始,就进入了 Linux 内核主线;3. 可能更快一些。

chroot 将进程的根目录更改为指定目录,从而限制进程只能访问该目录及其子目录的内容。 `OverlayFS 使用两个目录,把一个目录置放于另一个之上,并且对外提供单个统一的视角。这两个目录通常被称作层,这个分层的技术被称作 union mount。术语上,下层的目录叫做 lowerdir,上层的叫做 upperdir。对外展示的统一视图称作 merged。

2.3 资源分配

Linux Cgroups 提供了对一组进程及将来子进程的资源限制控制和统计的能力,这些资源包括 CPU、内存、存储、网络等。通过 Cgroups,可以方便地限制某个进程的资源占用,并且可以实时地监控进程的监控与统计信息。

一个 cgroup 中包含一组进程,并且可以在这个 cgroup 上增加 Linux subsystem 的各种参数配置,将一组进程和一组 subsystem 的系统参数关联起来。

subsystem 涉及到：blkio 块设备；cpu cpu 调度策略；cpuacct cgroup 中进程的 cpu 占用；cpuset 多核机器上设置 cgroup 中进程可以使用的 cpu 与内存；devices 控制 cgroup 中进程对设备的访问；memory 控制内存占用；net_cls 将 cgroup 中进程产生的网络包进行分类,以便 tc 分流；ns 使 cgroup 中的进程在新的 nameSpace 中 fork 新进程时创建进程。

2.4 网络隔离

网络隔离是指通过技术手段将不同网络段之间的通信隔离开来，以提高系统的安全性和管理效率。

在 dockee 项目中，网络隔离通常通过虚拟网络设备实现，尤其是使用桥接网络（Bridge）,默认的 dockee 网络驱动，为每个容器分配一个虚拟网卡，并连接到 dockee 创建的虚拟桥接器上，形成一个私有的内部网络。

在这样的设计下，容器通过该网络可以相互通信，但与宿主机和外部网络隔离。

3 实验结果

3.1 使用手册

dockee 是一个命令行工具，其使用方式如下。

```
USAGE:
sudo dockee [global options] command [command options] [arguments...]
COMMANDS:
exec exec a command into container
ps list all containers
logs print logs of a container
```

rm remove unused container
run create a container: my-docker run -ti [command]
stop stop a container
network container network commands
commit commit a container to image
image image commands
help, h Shows a list of commands or help for one command
GLOBAL OPTIONS:
--help, -h show help

run 命令的使用方式如下。

USAGE:
sudo dockee run [command options] [arguments...]
OPTIONs:
--it enable tty
-d detach container
-m value memory limit
--cpu value cpu limit
--cpuset value cpuset limit
-v value volume
--image value the image name used to build the container
--name value container name
-e value set environment
--net value container network
-p value port mapping
GLOBAL OPTIONs:
--debug enable debug mode
EXAMPLEs:
sudo ./dockee run -it --image busybox /bin/sh
sudo ./dockee run -d --image busybox /bin/ps
sudo ./dockee run -it --image busybox --net bridge33 '/bin/ip a'

network 命令的使用方式如下。

NAME:
dockee network - container network commands

```
USAGE:
  dockee network command [command options] [arguments...]
COMMANDs:
  create  create a container network
  list    list container network
  remove  remove container network
EXAMPLEs:
  sudo ./dockee network create --driver bridge --subnet 192.168.33.0/24
bridge33
```

3.2 跨系统环境运行

证明 dockee 支持跨系统环境运行，方式是在 Ubuntu 系统上运行 CentOS 环境。

我们在测试机上对资源分配进行测试，所使用的镜像基于 centos8。以下是拉取、并 docker 镜像转储为 dockee 镜像的过程。

```
docker pull centos
docker run -it --name res_test centos
docker export -o res_test.tar res_test
mv res_test.tar dockee/Images
```

此外，为了便于测试,我们还需要在容器中配置 python3 环境。由于 CentOS 官方已经停止维护 CentOS Linux ，因此需要做如下设置。

```
cd /etc/yum.repos.d/
sed -i 's/mirrorlist/#mirrorlist/g' /etc/yum.repos.d/CentOS-*
sed -i 's|#baseurl=http://mirror.centos.org|baseurl=http://vault.centos.org|g'
/etc/yum.repos.d/CentOS-*
yum makecache
yum update -y
yum -y install python36
```

具体演示过程参考“dockee 演示：跨系统环境运行.mp4”。

3.3 资源隔离

证明 dockee 支持证明资源隔离，方式是通过 ps、ip 等命令，显示容器与宿主机在资源上的差异。

```
sh-4.4# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
8: cif-27448@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether fe:79:04:ee:1a:a5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.33.3/24 brd 192.168.33.255 scope global cif-27448
       valid_lft forever preferred_lft forever
    inet6 fe80::fc79:4ff:feee:1aa5/64 scope link
       valid_lft forever preferred_lft forever
sh-4.4# ps
    PID TTY          TIME CMD
      1 ?        00:00:00 sh
      8 ?        00:00:00 ps
sh-4.4#
```

图：容器中的进程列表

```
root@DESKTOP-NQ98TQH:/home/zq# ps
    PID TTY          TIME CMD
   4293 pts/6    00:00:00 sudo
   4294 pts/6    00:00:00 su
   4318 pts/6    00:00:00 bash
   4334 pts/6    00:00:00 ps
root@DESKTOP-NQ98TQH:/home/zq#
```

图：主机中的进程列表

对比容器中的进程列表与主机中的进程列表存在差异，表明容器与宿主机在进程上存在差异和隔离。

```
sh-4.4# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
6: cif-22969@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether 3a:24:47:ce:ff:47 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.33.2/24 brd 192.168.33.255 scope global cif-22969
       valid_lft forever preferred_lft forever
    inet6 fe80::3824:47ff:fece:ff47/64 scope link
       valid_lft forever preferred_lft forever
sh-4.4#
```

图：容器中的网络列表

```
root@DESKTOP-NQ98TQH:/home/zq# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    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
    inet 10.255.255.254/32 brd 10.255.255.254 scope global lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1400 qdisc mq state UP group default qlen 1000
    link/ether 00:15:5d:26:5b:79 brd ff:ff:ff:ff:ff:ff
    inet 172.29.199.175/20 brd 172.29.207.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::215:5dff:fe26:5b79/64 scope link
       valid_lft forever preferred_lft forever
3: virbr0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default qlen 1000
    link/ether 52:54:00:29:32:3c brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.1/24 brd 192.168.122.255 scope global virbr0
       valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ec:29:17:00 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
5: mynetwork: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default qlen 1000
    link/ether ba:02:f5:f9:5c:5e brd ff:ff:ff:ff:ff:ff
    inet 192.168.33.1/24 brd 192.168.33.255 scope global mynetwork
       valid_lft forever preferred_lft forever
    inet6 fe80::3c1b:f4ff:fe2c:8320/64 scope link
```

图：主机中的网络列表

对比容器中的网络列表与主机中的网络列表存在差异，表明容器与宿主机在网络上存在差异和隔离。

3.4 资源分配

证明 dockee 为容器分配了定量的 CPU 和内存资源，方式是在容器中使用尽量多的资源，并通过 systemd-cgtop 命令检查资源使用情况。具体操作过程如下。

首先我们创建第一个容器,资源分配设置内存限制为 10m，cpu 设置为核 0，cpu share 设置为 20。

```
root@hecs-141140:/home/programing/dockee# ./dockee rm res1
INFO[0000] dockee is a simple container runtime implementation
root@hecs-141140:/home/programing/dockee# ./dockee run -it --name res1 --image res_test -m 10m -cpuset 0 -cpu 20 /bin/bash
INFO[0000] dockee is a simple container runtime implementation
/bin/bash
INFO[0000] dockee is a simple container runtime implementation
{"level":"info","msg":"init come on","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"command ","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"read parent pipe cmd","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"command all is /bin/bash","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"receive /bin/bash","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"Current location is [/var/lib/dockee/GU4TAMZYG4YTIMZZ/merge]","time":"2024-06-21T10:43:06+08:00"}
{"level":"info","msg":"now change dir to root","time":"2024-06-21T10:43:06+08:00"}
[root@hecs-141140 /]#
```

测试发现,能够成功分配 5M 内存,而分配 10M 内存是超出资源限制的。

```
[root@hecs-141140 /]# python3 -c "a = ' ' * (10 * 1024 * 1024);"
Killed
[root@hecs-141140 /]# python3 -c "a = ' ' * (5 * 1024 * 1024);"
[root@hecs-141140 /]#
```

接下来,我们运行如下脚本。

```
[root@hecs-141140 /]# cat test.py
import string
a = ' ' * (3 * 1024 * 1024)
while True:
    pass
[root@hecs-141140 /]# python3 test.py &
[1] 29
```

我们运行 systemd-cgtop 命令检查资源使用情况我们能够看到资源是占满的。

```
Control Group                                                  Tasks   %CPU   Memory  Input/s Output/s
/                                                                390   22.9     1.1G        -        -
GU4TAMZYG4YTIMZZ                                                   2   19.6     9.9M        -        -
```

至此，我们证明了在单个容器下的资源分配控制。

我们再创建第二个容器,资源分配设置内存限制为 50m,cpu 设置为核 0,cpu share 设置为 40,并在其中运行如下脚本。

```
root@hecs-141140:/home/programing/dockee# ./dockee run -it --name res2 --image res_test -m 50m -cpuset 0 -cpu 40 /bin/bash
INFO[0000] dockee is a simple container runtime implementation
/bin/bash
{"level":"info","msg":"command all is /bin/bash","time":"2024-06-21T10:49:58+08:00"}
INFO[0000] dockee is a simple container runtime implementation
{"level":"info","msg":"init come on","time":"2024-06-21T10:49:58+08:00"}
{"level":"info","msg":"command ","time":"2024-06-21T10:49:58+08:00"}
{"level":"info","msg":"read parent pipe cmd","time":"2024-06-21T10:49:58+08:00"}
{"level":"info","msg":"receive /bin/bash","time":"2024-06-21T10:49:58+08:00"}
{"level":"info","msg":"Current location is [/var/lib/dockee/G42DQOJWGU4DKNBW/merge]","time":"2024-06-21T10:49:58+08:00"}
{"level":"info","msg":"now change dir to root","time":"2024-06-21T10:49:58+08:00"}
[root@hecs-141140 /]# vi test.py
[root@hecs-141140 /]# cat test.py
import string
a = ' ' * (30 * 1024 * 1024)
while True:
    pass
[root@hecs-141140 /]# python3 test.py &
[1] 18
```

此时得到这两个容器的运行情况如下。

```
Control Group                                                  Tasks   %CPU   Memory  Input/s Output/s
/                                                                397   62.5     1.1G   231.6K    43.1K
G42DQOJWGU4DKNBW                                                   2   39.3    48.3M        -        -
GU4TAMZYG4YTIMZZ                                                   2   19.6     8.7M        -        -
```

可以看到,第二个容器占用了 40%的 CPU 资源,并拥有更多的内存(最多 50M),而第一个容器则只占用了 10%的 CPU 资源,且仅有最多 10M 的内存。

至此，我们证明了在多个容器下的资源分配控制。

3.5 网络通信

证明容器间实现网络通信，方式是通过 ping 命令将两个容器 ping 通。

具体演示过程参考“dockee 演示：容器间网络通信.mp4”。

4 附录

附录包括：

- dockee 项目源码.zip
- dockee 演示：跨系统环境运行.mp4
- dockee 演示：容器间网络通信.mp4