容器底层实现技术

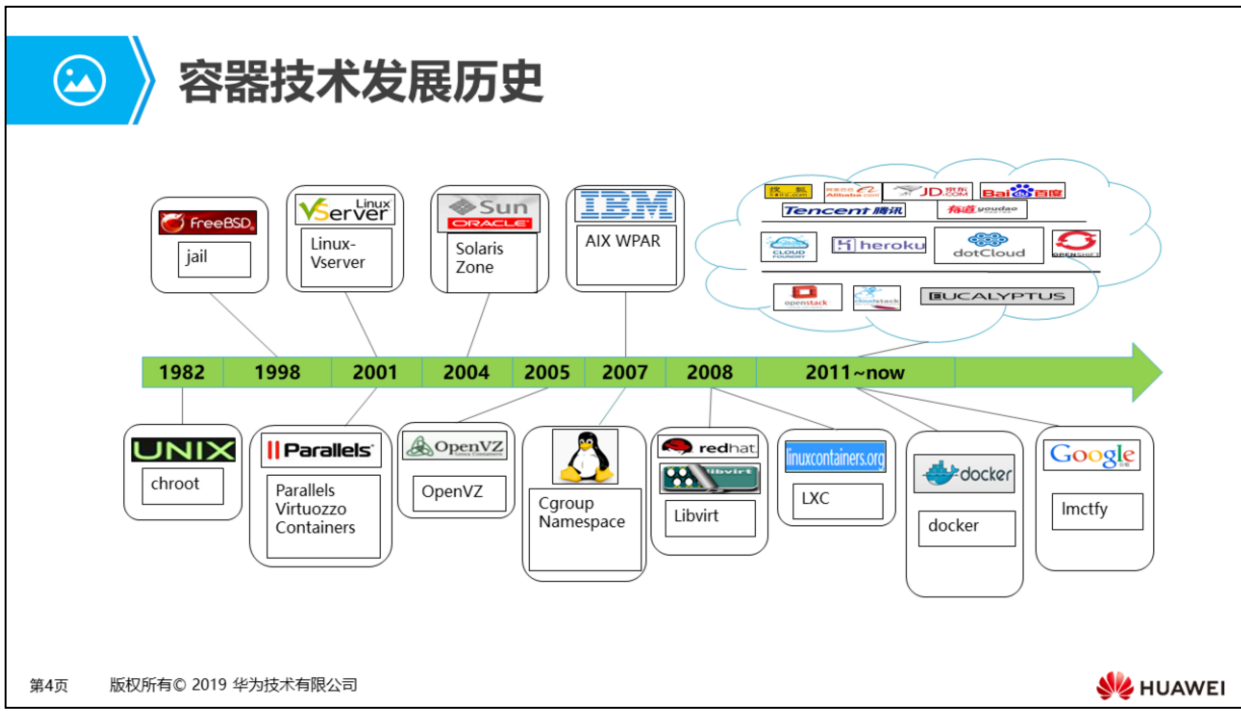
前言

- 本章主要介绍容器实现的核心技术Namespace和Cgroup，学习容器技术的本质，并对容器使用的资源进行限制。

目标

- 学完本课程后，您将能够：
  - 描述namespace实现
  - 描述cgroup实现
  - 掌握容器资源限制方法

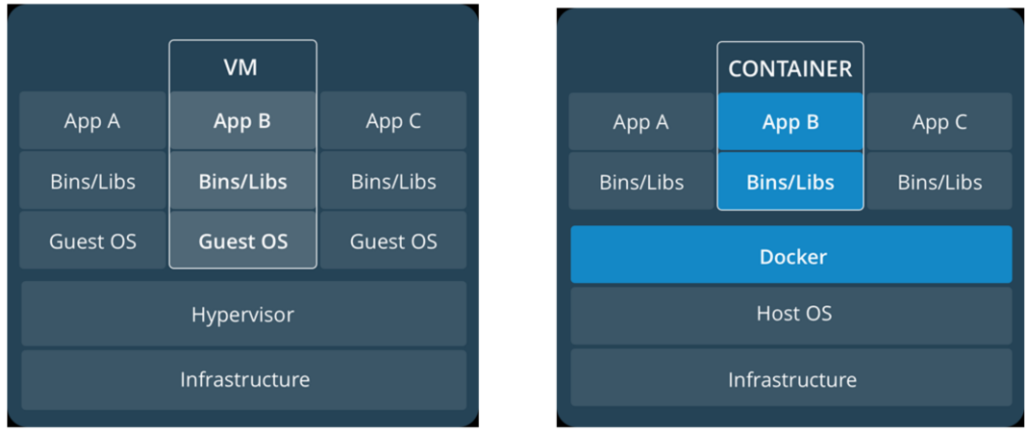
目录



- 从技术上而言，容器最早可以追溯到20世纪80年代初期的chroot。并且Docker中使用的关键技术如Cgroup和Namespace等在Linux中早已成熟。早在1982年，chroot技术实现根文件系统切换，实现了有限的文件系统隔离。2000年，Linux内核版本2.3.41中引入pivot_root技术进行Linux根文件系统切换，避免chroot带来的安全性问题。
- 整个容器技术的发展概况如下：
  - 在2000 年左右， 市场上出现了一些商用的容器技术， 比如Linux-VServer和SWsoft（现在的Odin）开发的Virtuozzo，虽然这些技术相对当时的XEN和KVM，有明显的性能提升，但是因为各种原因，并未在当时引起市场太多的关注。（注意这里只讨论Linux 系统上的容器技术，同时期还有很多有名的非Linux平台的容器技术，比如FreeBSD的jail、Solaris上的Zone等。）
  - 2005年，同样是Odin公司，在Virtuozzo的基础上发布了OpenVZ 技术，同时开始推动OpenVZ中的核心容器技术进入Linux内核主线，而此时IBM等公司也在推动类似的技术，最后在社区的合作下，形成了目前大家看到的Cgroup和Namespace，这时，容器技术才开始逐渐进入大众的视野。
  - 随着容器技术在内核主线中的不断成熟和完善，2013年诞生的Docker真正让容器技术得到了全世界技术公司和开发人员的关注。

Docker容器实现原理

- Docker容器在实现上通过namespace技术实现进程隔离，通过cgroup技术实现容器进程可用资源的限制。docker启动一个容器时，实际是创建了带多个namespace参数的进程。



  HUAWEI

- 容器其实是一种沙盒技术，需要为应用提供一个隔离的运行环境。
  - 容器中的应用实际上是运行在宿主机上的一个进程，通过namespace技术为该进程创建一个隔离的环境，并通过cgroup技术对该进程的可用资源进行限制。（容器本质上就是一个加了限定参数的进程。）
  - Namespace和cgroup是Linux Kernel中原有的特性。
- 虚拟机通过Hypervisor来虚拟硬件资源，再使用这些虚拟硬件资源创建VM，安装VM OS从而达到资源的隔离；而docker则是通过在host os内核上指定namespace参数来达到资源隔离。
- 运行在容器里的应用进程，跟宿主机上的其他进程一样，都由宿主机操作系统统一管理，只不过这些被隔离的进程拥有额外设置过的Namespace参数。

- 注：图片来自docker官网。

Namespace

- Namespace：命名空间
  - 作用：资源隔离
  - 原理：namespace将内核的全局资源进行封装，使得每个namespace都有一份独立的资源。因此不同进程在各自namespace内对同一种资源的使用不会相互干扰。

Namespace类型	系统调用参数	隔离内容	引入的内核版本
PID namespace	CLONE_NEWPID	进程空间（进程ID）	Linux 2.6.24
Mount namespace	CLONE_NEWNS	文件系统挂载点	Linux 2.6.19
Network namespace	CLONE_NEWNET	网络资源：网络设备、端口等	始于Linux 2.6.24完成于Linux 2.6.29
User namespace	CLONE_NEWUSER	用户ID和用户组ID	始于Linux 2.6.23完成于Linux 3.8
UTS namespace	CLONE_NEWUTS	主机名和域名	Linux 2.6.19
IPC namespace	CLONE_NEWIPC	信号量、消息队列和共享内存	Linux 2.6.19

- Namespace实际上是Linux系统上创建新进程时的一个可选参数。
- 实际上在创建Docker容器时，指定了这个进程所需要启用的一组namespace参数。通过namespace机制的隔离，容器只能见到当前Namespace中所限定的资源、文件、设备、状态或配置。以此实现应用运行环境的隔离。

PID namespace隔离示例 (1)

- 以交互模式启动一个centos容器，并在其中运行/bin/bash程序。执行ps命令查看到“/bin/bash”是PID=1的进程，即Docker将其隔离于宿主机中的其他进程。

```
[root@localhost ~]# docker run -it centos /bin/bash
[root@24b87937f13d /]# ps axf
   PID TTY      STAT   TIME COMMAND
     1 pts/0    Ss     0:00 /bin/bash
    14 pts/0    R+     0:00 ps axf
```

- 打开另一个终端，使用docker inspect查看容器进程在宿主机上的真实PID。实际上，该容器上运行的”/bin/bash”在宿主机上是PID=96745的进程。

```
[root@localhost ~]# docker inspect 24b87937f13d | grep Pid
            "Pid": 96745,
            "PidMode": "",
            "PidsLimit": 0,
```

还有什么方式可以查看到容器进程真实的PID?

HUAWEI

- 容器中的用户应用实际上是容器里PID=1的进程。
- 容器中的“应用“认为其是PID namespace进程空间中的1号进程（无法”看到“宿主机进程空间中的其他进程，也无法”看到“其他PID namespace中的进程），只能查看到mount namespace中挂载的文件和目录，只能访问到network namespace中的网络设备。综上，容器为应用的运行提供了一个隔离的环境。
- 在应用运行过程中，宿主机上并不存在一个所谓的容器，容器只是一个抽象概念。Docker实际上是给应用创建了一个隔离的环境。

- 注：centos系统上PID=1的进程是systemd。

PID namespace隔离示例 (2)

- 分别在宿主机和容器中查看该容器进程相关的namespace信息，发现两者是一致的。
  - 注：每个进程在/proc下都有一个目录，存放namespace相关信息。

```
[root@localhost ns]# ll /proc/96745/ns
total 0
lrwxrwxrwx. 1 root root 0 Aug  9 13:56 ipc -> ipc:[4026532192]
lrwxrwxrwx. 1 root root 0 Aug  9 13:56 mnt -> mnt:[4026532190]
lrwxrwxrwx. 1 root root 0 Aug  9 13:17 net -> net:[4026532195]
lrwxrwxrwx. 1 root root 0 Aug  9 13:56 pid -> pid:[4026532193]
lrwxrwxrwx. 1 root root 0 Aug  9 13:56 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Aug  9 13:56 uts -> uts:[4026532191]
```

```
[root@24b87937f13d 1]# ll /proc/1/ns
total 0
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 ipc -> ipc:[4026532192]
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 mnt -> mnt:[4026532190]
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 net -> net:[4026532195]
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 pid -> pid:[4026532193]
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 user -> user:[4026531837]
lrwxrwxrwx. 1 root root 0 Aug  9 19:16 uts -> uts:[4026532191]
```


- Linux下的/proc目录存储的是记录当前内核运行状态的一系列特殊文件。通过访问这些文件，可以查看系统以及当前正在运行的进程的信息，如CPU、内存使用率等。

- 一个容器中的应用，在宿主机OS上是一个普通进程，与宿主机OS上其他进程在资源使用上存在竞争关系。需要通过技术手段对该进程能够使用的资源进行限制。
- Linux Cgroups于2006年，由Google发起。Linux Cgroups是Linux内核中用于对进程进行资源限制的一个重要功能。
- Cgroups中各个子系统对应的路径下有都有多个配置文件，通过这些配置文件可以对相应的资源进行限制，如：
  - cpu type cgroup：设置进程的cpu使用限制。
  - cpuset type cgroup：为进程分配单独的CPU核和对应的内存节点。
  - memory type cgroup：设置进程的内存使用限制。
  - blkio type cgroup：设置进程的块设备I/O使用限制。

目录

HUAWEI

CPU资源限制

- 可通过如下参数，对容器的可用CPU资源进行限制：
  - --cpu-shares：权重值，表示该进程能使用的CPU资源的权重值。
  - cpu.cfs_period_us和cpu.cfs_quota_us：这两个配置参数一般配合使用，表示限制进程在长度为cpu.cfs_period_us的一段时间内，只能被分配到总量为cpu.cfs_quota_us的CPU时间。
    - 例如： 某容器的cpu.cfs_period_us=100000， cpu.cfs_quota_us=10000。则表示该容器只能使用10%的CPU资源。

- 对进程CPU使用限制的可配置参数，详见cgroup下cpu子系统中的配置文件：ll /sys/fs/cgroup/cpu。

CPU资源限制示例 (1)

- 启动一个容器，进行压力测试，设置cpu权重值为1024。

```
[root@localhost /]# docker run --name huawei1 -it -c 1024 progrium/stress --cpu 1
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] --> hogcpu worker 1 [6] forked
```

- 使用top命令查看系统宿主机cpu使用率。

```
top - 11:02:46 up 30 days,  5:48,  4 users,  load average: 2.09, 2.07, 2.10
Tasks: 105 total,   2 running, 102 sleeping,   1 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1777256 total,   120904 free,   341496 used,  1314856 buff/cache
KiB Swap:  1048572 total,  1037820 free,    10752 used.  1053336 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
70900 root      20   0    7304    100      0 R 99.3  0.0 200:14.67 stress
    1 root      20   0   54404   6212   3624 S  0.3  0.3   0:51.94 systemd
89355 root      20   0  161880   2216   1572 R  0.3  0.1   0:00.11 top
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.08 kthreadd
    3 root      20   0       0      0      0 S  0.0  0.0   0:02.37 ksoftirqd/0
```

Cpu权重是否生效?

HUAWEI

- 压力测试的镜像可使用”dokcer search stress”命令搜索，或者到Docker Hub官网搜索。

CPU资源限制示例 (2)

- 启动第二个容器，cpu权重值依旧设置为1024。

```
[root@localhost /]# docker run --name huawei2 -it -c 1024 polinux/stress-ng --cpu 1
stress-ng: info:  [1] defaulting to a 86400 second run per stressor
stress-ng: info:  [1] dispatching hogs: 1 cpu
stress-ng: info:  [1] cache allocate: default cache size: 16384K
```

- 再次使用top命令查看宿主机系统cpu使用率。

```
top - 11:01:25 up 30 days,  5:47,  4 users,  load average: 2.03, 2.06, 2.10
Tasks: 107 total,   3 running, 103 sleeping,   1 stopped,   0 zombie
%Cpu(s):100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1777256 total,   117180 free,   342692 used,  1317384 buff/cache
KiB Swap:  1048572 total,  1037820 free,    10752 used.  1049804 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
72058 root      20   0    7304     96      0 R  49.7   0.0 177:53.70 stress
70900 root      20   0    7304    100      0 R  49.3   0.0 199:28.70 stress
70822 root      20   0  308436  16880  10652 S   0.3   0.9   0:00.87 docker
89355 root      20   0  161880   2216   1572 R   0.3   0.1   0:00.01 top
175586 root     20   0  336432  29168  12460 S   0.3   1.6   2:37.74 docker-containe
    1 root      20   0   54404   6212   3624 S   0.0   0.3   0:51.91 systemd
```

你能找到对该容器进行CPU限制的配置文件吗?

HUAWEI

- 注：只有容器间需要竞争使用CPU资源使用时，--cpu-shares权重值才会起作用。

CPU资源限制示例 (3)

- 查看刚才创建的两个名为huawei1和huawei2容器的CONTAINED ID。

```
[root@localhost ~]# docker ps -f name="huawei*"
CONTAINER ID   IMAGE              COMMAND                  CREATED          STATUS          PORTS     NAMES
784e2e058291   progrium/stress    "/usr/bin/stress --v…"   24 seconds ago   Up 22 seconds             huawei2
bca245a8e3ba   progrium/stress    "/usr/bin/stress --v…"   39 seconds ago   Up 28 seconds             huawei1
```

- 已经自动为这两个容器在/sys/fs/cgroup/cpu/docker目录下创建了相应的文件夹。

```
[root@localhost ~]# cd /sys/fs/cgroup/cpu/docker/
[root@localhost docker]# ll
total 0
drwxr-xr-x. 2 root root 0 Aug  9 12:33 784e2e05829128bdc4c38cd249f0c2691a10c19fc4794f86602184ba990042a1
drwxr-xr-x. 2 root root 0 Aug  7 00:38 bca245a8e3ba26cc967c35d10be51b3c3e95faf22aae988e45b76bc1441e56ee
-rw-r--r--. 1 root root 0 Jul 10 09:14 cgroup.clone_children
--w--w--w-. 1 root root 0 Jul 10 09:14 cgroup.event_control
-rw-r--r--. 1 root root 0 Jul 10 09:14 cgroup.procs
```

- 查看容器的cpu.shares参数值和tasks值。其中tasks值即为该容器进程在宿主机上的PID。

```
[root@localhost docker]# cat 784e2e05829128bdc4c38cd249f0c2691a10c19fc4794f86602184ba990042a1/cpu.shares
1024
[root@localhost docker]# cat 784e2e05829128bdc4c38cd249f0c2691a10c19fc4794f86602184ba990042a1/tasks
94164
```

- 可在宿主机中查看cat /proc/94164/cgroup验证该容器进程当前生效的cgroup配置。

  HUAWEI

- Docker在/sys/fs/cgroup/cpu/docker目录下，为每个容器创建了一个用于CPU资源使用限制的文件夹，每个文件夹中有一组CPU资源限制文件。
- 其他资源子系统如memory、blkio等也在其目录下为每个容器创建一个相应的文件夹，用于相应资源的使用限制。

- 通过上例可知：一个正在运行的Docker容器，实际是一个启用了多个Linux Namespace的应用进程，而该进程可使用的资源受Cgroups配置的限制。

内存资源限制

- 默认情况下，宿主机不限制容器对内存资源的使用。可使用如下参数来控制容器对内存资源的使用：
  - --memory：设置内存资源的使用限额
  - --memory-swap：设置内存和SWAP资源的使用限额
- 对进程内存使用限制的详细配置参数在/sys/fs/cgroup/memory目录：

```
[root@localhost ~]# ls /sys/fs/cgroup/memory/
cgroup.clone_children  memory.kmem.limit_in_bytes          memory.limit_in_bytes           memory.oom_control          release_agent
cgroup.event_control   memory.kmem.max_usage_in_bytes      memory.max_usage_in_bytes       memory.pressure_level       system.slice
cgroup.procs           memory.kmem.slabinfo                memory.memsw.failcnt            memory.soft_limit_in_bytes  tasks
cgroup.sane_behavior   memory.kmem.tcp.failcnt             memory.memsw.limit_in_bytes     memory.stat                 user.slice
docker                 memory.kmem.tcp.limit_in_bytes      memory.memsw.max_usage_in_bytes memory.swappiness
memory.failcnt         memory.kmem.tcp.max_usage_in_bytes  memory.memsw.usage_in_bytes     memory.usage_in_bytes
memory.force_empty     memory.kmem.tcp.usage_in_bytes      memory.move_charge_at_immigrate memory.use_hierarchy
memory.kmem.failcnt    memory.kmem.usage_in_bytes          memory.numa_stat                notify_on_release
```


- 容器可使用的内存资源包括内存和SWAP资源。

内存资源限制示例

- 启动一个容器，使其最多使用400M内存和100M swap。并进行压力测试。

```
[root@localhost /]# docker run -it -m 400M --memory-swap=500M progrium/stress --vm 1
--vm-bytes 450M
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: dbug: [1] using backoff sleep of 3000us
stress: dbug: [1] --> hogvm worker 1 [6] forked
stress: dbug: [6] allocating 471859200 bytes ...
stress: dbug: [6] touching bytes in strides of 4096 bytes ...
stress: dbug: [6] freed 471859200 bytes
stress: dbug: [6] allocating 471859200 bytes ...
stress: dbug: [6] touching bytes in strides of 4096 bytes ...
```

- 以上过程循环在“分配450M内存，释放450M内存”。

Block IO限制

- Block IO指的是磁盘的读写，可通过如下3种方式限制容器读写磁盘的带宽。
  - 设置相对权重
    - --blkio-weight　　Block IO (relative weight)
  - 设置bps：每秒读写的数据量
    - --device-read-bps　　Limit read rate (bytes per second) from a device
    - --device-write-bps　　Limit write rate (bytes per second) to a device
  - 设置iops：每秒IO次数
    - --device-read-iops　　Limit read rate (IO per second) from a device
    - --device-write-iops　　Limit write rate (IO per second) to a device

知识小考

- Docker到底为Container虚拟了什么资源?
- 容器的安全性、隔离性为什么达不到虚拟机的级别?
- Cgroups能否对容器的资源使用下限进行锁定?
- Namespace是否可以隔离系统上的所有资源?

实验&实训任务

- 实验任务
  - 请按照实验手册1.6部分完成Namespace和cgroups部分实验。
- 实训任务
  - 请灵活使用本章节课程及实验手册中学到的知识，按照实验手册1.6.4章节完成Namespace和Cgroups实训任务。

思考题

1. 限制Block IO带宽就可以限制所有IO的读写速度。 T or F
2. 默认情况下，系统不对容器的CPU资源进行限制。T or F
3. 宿主机是否有权限杀死一个docker启动的容器？命令是什么？
4. 下列哪一项为Container分配所需的资源？
   A. Linux Kernel
   B. Docker Engine
   C. Docker Daemon
   D. namespace

  HUAWEI

- 参考答案：
  - F。
  - F。
  - 有权限。kill -9 pid。
  - A。

本章总结

- Namespace实现原理
- Cgroups实现原理
- 容器对CPU、内存、Block IO资源的使用限制


谢 谢
www.huawei.com