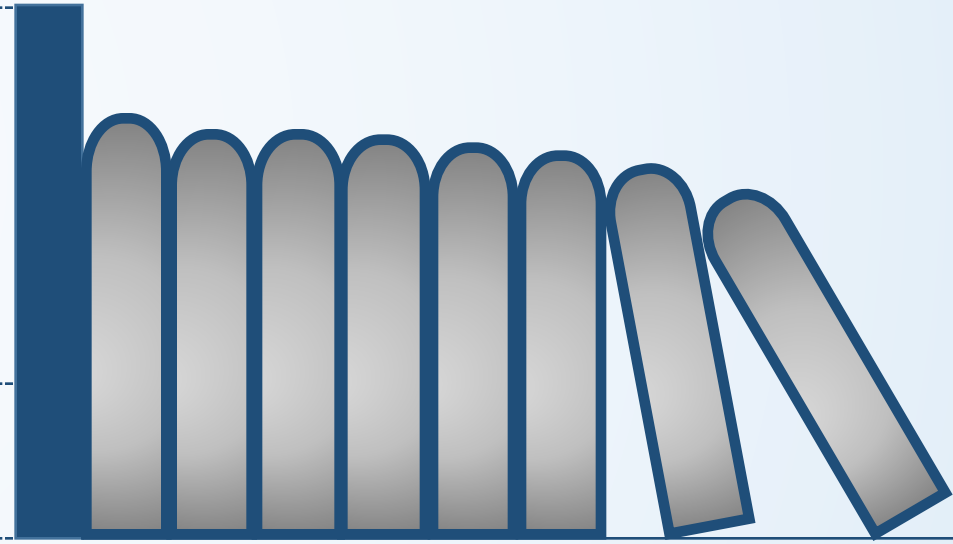
容器技术

主讲人：宋小金



目录

预期收获

- 了解Cgroup原理
- 了解Namespace原理

Docker进程隔离技术

Docker的进程级隔离，采用的核心技术主要有以下2个：

- Cgroup
- Namespace

Cgroup

Cgroup是 Control Groups的简称，被直译为控制组，它主要是做*资源控制*，用于*限制和隔离一组进程* (Process Groups) *对系统资源* (如 CPU Memory I/O 等) 的使用。

其原理是将*一组进程放在一个控制组*里，通过给这个*控制组分配指定的可用资源*，达到控制这一组进程可用资源的目的。

从实现的角度来看，Cgroup实现了一个通用的进程分组的架构，而不同资源的具体管理则是由*各个Cgroup子系统*实现的。

Cgroup

截止到内核4.1版本，Cgroup中实现的子系统如下：

- *Devices*
- *cpuset*
- *memory子系统*
- *cpu子系统*
- *cpuacct 子系统*
- *blkio*

Cgroup devices

devices：

devices用来控制Cgroup中的*进程对哪些设备有访问权限*，接口如下：

- **devices.list**：设备列表，有3个域，分别为：
  - 类型：可以是a（所有设备）、b（块设备）、c（字符设备）。
  - 设备号：格式为major:minor的设备号。
  - 权限：r（读）、w（写）、m（创建设备节点mknod）。

```
[root@k8s-0104-23008 home]# ls -l /dev/null
crw-rw-rw- 1 root root 1, 3 Jan  2 16:10 /dev/null
[root@k8s-0104-23008 home]# cat /sys/fs/cgroup/devices/docker/devices.list
a *:* rwm
```

Cgroup devices

devices：

devices用来控制Cgroup的*进程对哪些设备有访问权限*，其接口如下：

- **device.allow**: 是一个只写文件，格式与devices.list中的说明一致。通写入相应信息（例如：c 1:3 r）到该文件，就可以*允许指定设备访问权限*。）。

```
[root@k8s-0104-23008 home]# ls -l /sys/fs/cgroup/devices/docker/devices.allow
--w------- 1 root root 0 Jan  2 16:12 /sys/fs/cgroup/devices/docker/devices.allow
```

Cgroup devices

devices：

devices用来控制Cgroup的*进程对哪些设备有访问权限*，其接口如下：

- **devices.deny**: 刚好与device.allow启的作用相反，通写入相应信息到该文件，就可以*禁止指定设备访问权限*。

```
[root@k8s-0104-23008 home]# ls -l /sys/fs/cgroup/devices/docker/devices.deny
--w------- 1 root root 0 Jan  2 16:12 /sys/fs/cgroup/devices/docker/devices.deny
```

Cgroup cpuset

cpuset

用于分配指定的CPU和内存节点，包括cpuset.cpus、cpuset.mems 等接口， 用来*限制控制组中的进程可以使用的 cpu 核心和内存节点*。

现已广泛用于KVM与容器等场景上。

Cgroup cpuset

cpuset举例：

第一步：创建一个控制组：*$ sudo mkdir /sys/fs/cgroup/cpuset/mygroup*

第二步：限制控制组mygroup只能使用cpu0 cpu2和内存节点0：

$ echo 0 >/sys/fs/cgroup/cpuset/mygroup/cpuset.mems

$ echo 0,2 >/sys/fs/cgroup/cpuset/mygroup/cpuset.cpus

第三步：将当前终端进程加入到mygroup控制组

$ echo $$ > /sys/fs/cgroup/cpuset/mygroup/tasks

第四步验证：查看设置结果, 当前进程只能使用cpu0、cpu2和内存节点0：

$ echo “hello world” > /dev/null

$ cat /proc/$$/status |grep ‘_allowed’

Cpus_allowed: 00000005

Cpus_allowed_list: 0,2

Mems_allowed:00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0
0000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,000000
00,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000001

Mems_allowed_list: 0

Cgroup memory子系统

memory子系统：

用来*限制Cgroup组所能使用的内存*，主要有如下接口：

- memory.limit_*in*_bytes: 设定内存使用上限，单位可以使用k/K，m/M，g/G等。
- memory.memsw.limit_*in*_bytes: 设定*内存加交换分区*的使用总量，防止交换分区耗光。
- memory.oom_control: 决定一个进程在申请*内存超限时，是否会被系统kill掉*。包含一个标志（0或1）来开启或者关闭cgroup的OOM killer。如果开启（1），任务如果尝试申请内存超过允许，就会被*系统OOM killer终止*。OOM killer在每个使用cgroup内存子系统中都是*默认开启*的。如果需要关闭，则可以向memory.oom_control文件写入1。>>如果OOM killer关闭，那么进程尝试申请的*内存超过允许*，那么它就*会被暂停，直到额外的内存被释放*。
- memory.stat：汇报*内存使用信息*，包括当前资源总量、使用量、换页次数、活动页数量等等。

Cgroup cpu子系统

cpu子系统：

每个进程能够*占用CPU多长时间, 什么时候能够占用CPU*是和系统的调度密切相关的.Linux系统中有多种调度策略, 各种调度策略有其适用的场景, 也很难说哪种调度策略是最优的.

通过 cgroups 来*管理进程使用的 CPU 资源*。

先说控制进程的 cpu 使用，在一个机器上运行多个可能消耗大量资源的程序时，我们不希望出现某个程序占据了所有的资源，导致其他程序无法正常运行，或者造成系统假死无法维护。这时候用 cgroups 就可以很好地控制进程的资源占用。

Cgroup cpu子系统

cpu子系统：

cpu 子系统可以*调度 cgroup 对 CPU 的获取量*。 可用以下两个调度程序来管理对 CPU资源的获取：

- 完全公平调度 Completely Fair Scheduler (CFS) ：一个比例分配调度程序，可根据*任务优先级／权重*或 cgroup 分得的份额，在任务群组（cgroups）间按比例分配 CPU 时间（CPU 带宽）。

- 实时调度 Real-Time scheduler (RT) ： 一个任务调度程序，可对*实时任务使用 CPU 的时间*进行限定。

Cgroup cpu子系统

cpu子系统：

cpu子系统有如下接口：

- **资源组的CPU使用权重**（cpu.shares）：不是限制进程能使用的绝对的 cpu 时间，而是各*控制组之间的配额*，通常情况下，这种方式在保证公平的情况下能更充分利用资源。

举例：比如，通过对2个不同控制组的配额进行设置后，当两个组中的进程都*满负荷运行*时， mygroup2控制组中的所有任务进程所能占用的 cpu 就是mygroup中的全部任务进程的两倍。如果其他控制组中的进程*闲*着，那某一个组的进程完全可以用满全部 cpu。

Cgroup cpu子系统

cpu子系统：

cpu子系统有如下接口：

- **限制资源组的CPU使用硬上限：**
  - cpu.cfs_*period*_us：表示将***cpu时间片***分成cpu.cfs_*period*_us 份。
  - cpu.cfs_*quota*_us： 表示当前这个组中的task(/cgroup/mave/tasks中的taskid)将*分配多少比例的cpu时间片*，单位为微秒。一旦 cgroup 中任务*用完按配额分得的时间*，它们就会被在此阶段的时间提醒限制流量，并在进入*下阶段前禁止运行*。

举例：将period设置为1秒，qutoa设置为0.5秒，那么在Cgoup的进程在一秒内最多可以运行0.5秒，然后会被强制休眠，直至下一秒才能继续运行。

Cgroup cpu子系统

cpu子系统：

cpu子系统有如下接口：

- **对cgroup 中的进程组中的实时进程进行 CPU使用时间的控制**

所谓的实时进程, 也就是那些对*响应时间要求比较高*的进程， 这类进程需要在限定的时间内处理用户的请求, 因此, 在限定的这段时间内, 需要占用所有CPU资源, 并且不能被其它进程打断。

因为实时进程的CPU优先级高, 并且*未处理完之前是不会释放CPU*资源的.

Cgroup cpu子系统

cpu子系统：

cpu子系统有如下接口：

- **对cgroup 中的进程组中的实时进程进行 CPU使用时间的控制，2个接口参数：**
  - cpu.rt_*period*_us：此参数可以设定在某个时间段中， 每隔多久，cgroup 对 CPU 资源的存取就要重新分配，单位为微秒（μs，这里以“us”表示）。
  - cpu.rt_*runtime*_us >>>此参数可以指定在某个时间段中， cgroup 中的任务对 CPU 资源的最长连续访问时间，单位为微秒（μs，这里以“us”表示），只可用于实时调度任务。

Cgroup cpuacct 子系统

cpuacct 子系统：

CPU 统计（CPU accounting）（cpuacct）子系统会自动生成报告来显示 cgroup 任务所使用的 CPU 资源，其中包括子群组任务。报告有三种：

cpuacct.usage：报告此 cgroup 中*所有任务（包括层级中的低端任务）使用 CPU 的总时间*（纳秒：nanoseconds）。

cpuacct.stat：报告此 cgroup 的所有任务（包括层级中的低端任务）使用的用户和系统 CPU 时间，方式如下：

- user — 用户模式中任务使用的 CPU 时间。
- system — 系统（kernel）模式中任务使用的 CPU 时间

cpuacct.usage_percpu：报告 cgroup 中所有任务（包括层级中的低端任务）在每个 CPU 中使用的 CPU 时间（纳秒）。

Cgroup blkio

blkio：

块 I/O（blkio）子系统可以控制并监控 cgroup 中的任务*对块设备 I/O 的存取*。对一些伪文件写入值可以限制存取次数或带宽，从伪文件中读取值可以获得关于 I/O 操作的信息。

blkio 子系统给出两种方式来控制对 I/O 的存取

- 权重分配：用于完全公平列队 I/O 调度程序 （Completely Fair Queuing I/O scheduler），用此方法，可以给指定的 cgroup 设定权重。这意味着每个 cgroup 都有一个预留的 I/O 操作设定比例（根据 cgroup 的权重）。
- I/O 节流（上限）：当一个指定设备执行 I/O 操作时，此方法可为其操作次数设定上限。这意味着一个设备的“读”或者“写”的操作次数是可以限定的。

Namespace

Namespace

Linux 内核从版本 2.4.19的概念。其目的是将某个特定的全局系统资源通过抽象方法使得namespace 中的进程看起来拥有它们自己的隔离的全局系统资源实例。

当前一个container对应进程的Namespace信息:

```
$ sudo ls -l /proc/17780/ns
total 0
lrwxrwxrwx 1 root root 0 May 9 01:21 ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0 May 9 01:21 mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0 May 9 01:21 net -> net:[4026531956]
lrwxrwxrwx 1 root root 0 May 9 01:21 pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0 May 9 01:21 uts -> uts:[4026531838]
```

Namespace

Namespace

Linux 内核中实现了六种 namespace

Namespace	内容
UTS	主机名与域名
IPC	信号量、消息队列和共享内存
PID	进程编号
Network	网络设备、网络栈、端口等
Mount	挂载点（文件系统）
User	用户和用户组

Namespace

举例：Mount

第一步：使用unshare隔离mnt namespace：unshare --mount /bin/bash
第二步：挂载tmpfs：$ mount -t tmpfs tmpfs /tmp/mnt_isolation
第三步：进入/tmp/mnt_isolation，创建文件。
第四步：另起终端，查看/tmp/mnt_isolation内容

```
!408 # echo $$
3016

root@bjo-ep-018.dev.fwmrm.net:/tmp/mnt_isolation · 10:37
 AM Sat May 05 ·
!409 # ls
total 0
0 linux-1   0 linux-2  0 linux-4  0 linux-6  0 linux-8
0 linux-10  0 linux-3  0 linux-5  0 linux-7  0 linux-9
```

```
!393 # echo $$
2718

root@bjo-ep-018.dev.fwmrm.net:/tmp/mnt_isolation
!394 # ls
total 0

root@bjo-ep-018.dev.fwmrm.net:/tmp/mnt_isolation
!395 #
```

IPC

第一步：创建一个消息队列

$ ipcmk --queue

Message queue id: 0

第二步：查看创建的消息队列：

```
!5044 $ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x7d3ce53b 0          ts1        644        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x6c03e83e 0          zabbix     600        657056     6

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x7a03e83e 0          zabbix     600        13
```

IPC

IPC

第三步：使用unshare隔离ipc namespace，已经切换为新进程了。

$ echo $$

16725

$ unshare --ipc /bin/bash

$ echo $$

21201

第四步：被隔离了ipc namespac的当前bash中确认ipc状况，结果是看不到，所以在被隔离的namespace中创建的内容外部也看不到得到了验证。

```
!288 # ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

课程回顾

已学知识要点

- *通过Docker概览架构，加深对Docker理解*
- *了解Cgroup以及及其主要接口*
- *了解Namesapce以及及其主要接口*