2021.3.9 Docker 资源限制

docker 容器技术底层是通过 Cgroup（Cgroup Group 控制组）实现容器对物理资源使用的限制，限制的资源包括 CPU，内存，磁盘三个方面。基本覆盖的常见的资源和使用量控制

Cgroup 是 linux 内核提供的一种可以限制，记录，隔离进程所使用的物理资源的机制，被 LXC 及 docker 等项用于实现进程的资源控制

Cgroup 是提供将进程进行分组化管理的功能和接口的基础结构，docker 中 i/o 或内存的分配控制等具体的资源管理功能都是通过 Cgroup 功能来实现的。这些具体的资源管理功能称为 Cgroup 子系统，以下是对几大系统的介绍

·blkio ：限制快设备的输入输出控制。如：磁盘，光盘，USB 等

·cpu ：限制 cpu 资源的访问

·cpuacct ：产生 Cgroup 的任务的 cpu 资源报告

·cpuset ：限制分配单独的 cpu 和内存资源

·devices ：允许拒绝对设备的访问

·freezer ：暂停和恢复 Cgroup 任务

·memory ：设置每个 Cgroup 的内存限制以及产生内存资源报告

·net_cls :用于标记每个网络包

·ns ：命名空间子系统

·perf_event ：增加了对每个 group 的检测跟踪能力，可以检测属于某个特定的 group 的所有线程以及运行在特定的 cpu 上的线

程

使用下面的 dockerfile，来创建一个基于 centos 的 tress 工具镜像

```
[root@server04 ~]# cat centos-7-x86_64.tar.gz  | docker import - centos:7
#导入一个 centos 的镜像
[root@server04 ~]#  mkdir  stress
[root@server04 ~]#  cd  stress
[root@server04 ~]#  vim Dockerfile
FROM centos:7          #基于 centos 构建
MAINTAINER crushlinux "crushlinux@163.com"   #作者描述（无关紧要）
RUN yum -y install wget   #安装 wget
RUN wget -O /etc/yum.repos.d/epel.repo http://mirrors.aliyun.com/repo/epel-7.repo   #下载 aliyun 的 repo 源
RUN yum -y install stress     #安装工具包
[root@server04 strees]# docker build -t centos:stress .   #开始构建镜像
[root@server04 strees]# docker  images
REPOSITORY        TAG       IMAGE ID       CREATED          SIZE
centos            stress    318feac0bc5c   6 minutes ago    844MB
```

1）CPU 使用率限制

在 centos7 中可以通过修改对应的 Cgroup 配置文件 cpu.cfs_qouta_us 的值来实现，直接执行 echo 命令将设定的值导入到文件中就会立即生效

例如：将容器 face0aee201c 的 cpu 使用设置为 20000，设置 cpu 的使用率限定为 20%

```
[root@server04 ]# docker run -itd centos:stress /bin/bash #生成一个容器（注意好
```

容器 ID 号下面的实验中需要用到）

68f9a5e28097a123a97134ec1934c26f2fa7a7b599743c2be7fb06fee72cb7e0

[root@server04]# echo "2000" > /sys/fs/cgroup/cpu,cpuacct/docker/ID 号/cpu.cfs_quota_us #将值输出到文件中，以控制使用率为百分之 20%

[root@server04]#cat /sys/fs/cgroup/cpu,cpuacct/docker/ID 号/cpu.cfs_quota_us # 查看值

20000

测试：进入容器内

[root@server04 strees]# docker attach 68f9a #进入容器

[root@68f9a5e28097 /]# stress -c 20 #表示执行二十个进程

stress: info: [80] dispatching hogs: 50 cpu, 0 io, 0 vm, 0 hdd

[root@server04 ~]# top #查看系统的使用情况

```
top - 15:28:04 up  1:55,  2 users,  load average: 0.01, 0.12, 0.14
Tasks: 169 total,  21 running, 148 sleeping,   0 stopped,   0 zombie
%Cpu(s): 10.1 us,  0.0 sy,  0.0 ni, 89.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  2027904 total,   191032 free,   401120 used,  1435752 buff/cache
KiB Swap:  2097148 total,  2095612 free,     1536 used.  1435964 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
68242 root      20   0    7260     96      0 R   1.3  0.0   0:00.44 stress
68243 root      20   0    7260     96      0 R   1.3  0.0   0:00.50 stress
68244 root      20   0    7260     96      0 R   1.3  0.0   0:00.50 stress
68246 root      20   0    7260     96      0 R   1.3  0.0   0:00.50 stress
68249 root      20   0    7260     96      0 R   1.3  0.0   0:00.42 stress
68252 root      20   0    7260     96      0 R   1.3  0.0   0:00.42 stress
68235 root      20   0    7260     96      0 R   1.0  0.0   0:00.47 stress
68236 root      20   0    7260     96      0 R   1.0  0.0   0:00.43 stress
68237 root      20   0    7260     96      0 R   1.0  0.0   0:00.43 stress
68239 root      20   0    7260     96      0 R   1.0  0.0   0:00.42 stress
68241 root      20   0    7260     96      0 R   1.0  0.0   0:00.43 stress
68245 root      20   0    7260     96      0 R   1.0  0.0   0:00.42 stress
68247 root      20   0    7260     96      0 R   1.0  0.0   0:00.42 stress
68250 root      20   0    7260     96      0 R   1.0  0.0   0:00.42 stress
68253 root      20   0    7260     96      0 R   1.0  0.0   0:00.43 stress
68254 root      20   0    7260     96      0 R   1.0  0.0   0:00.47 stress
68238 root      20   0    7260     96      0 R   0.7  0.0   0:00.41 stress
68240 root      20   0    7260     96      0 R   0.7  0.0   0:00.46 stress
68248 root      20   0    7260     96      0 R   0.7  0.0   0:00.41 stress
68251 root      20   0    7260     96      0 R   0.7  0.0   0:00.49 stress
  756 root      20   0  324784   3372   1844 S   0.3  0.2   0:07.43 vmtoolsd
```

2）cpu 的共享比例

当多个容器任务运行时，很难计算 CPU 的使用率，为了使容器合理使用 CPU 资源，可以通过--cpu-shares 选项设置容器按比例共享 CPU 资源，这种方式还可以实现 CPU 使用率的动态调整。

命令中的--cpu-shares 选项值不能保证可以获得 1 个 vcpu 或者多少 GHz 的 CPU 资源，仅仅只是一个弹性的加权值。

例如，运行 3 个新建容器 A、B、C。占用 CPU 资源的比例为 1:1:2

```
docker run --name A -itd --cpu-shares 1024 centos:stress /bin/bash
docker run --name B -itd --cpu-shares 1024 centos:stress /bin/bash
docker run --name C -itd --cpu-shares 2048 centos:stress /bin/bash
```

如果又一个容器 D 需要更多 CPU 资源，则可以将其--cpu-shares 的值设置为 4096，那么 ABCD 的 CPU 资源比例变为 1:1:2:4。

默认情况下，每个 docker 容器的 cpu 份额都是 1024

默认情况下，每个 docker 容器的 cpu 份额都是 1024。单独一个容器的份额是没有意义的。只有在同时运行多个容器时，容器的 CPU 加权的效果才能体现出来。例如，两个容器 A、B 的 CPU 份额分别为 1000 和 500，在 CPU 进行时间片分配的时候，容器 A 比容器 B 多一倍的机会获得 CPU 的时间片。但分配的结果取决于当时主机和其他容器的运行状态，实际上也无法保证容器 A 一定能获得 CPU 时间片。比如容器 A 的进程一直是空闲的，那么容器 B 是可以获取比容器 A 更多的 CPU 时间片的。极端情况下，比如说主机上只运行了一个容器，即使它的 CPU 份额只有 50，它也可以独占整个主机的 CPU 资源。

Cgroups 只在容器分配的资源紧缺时，也就是说在需要对容器使用的资源进行限制时才会生效。因此无法单纯根据某个容器的 CPU 份额来确定有多少 CPU 资源分配给它，资源分配结果取决于同时运行的其他容器的 CPU 分配和容器中进程运行情况。

换句话说,可以通过 cpu shares 可以设置容器使用 CPU 的优先级，比如启动了两个容器及运行查看 CPU 使用百分比。

cpu 成二倍的占用率

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
68589 root      20   0    7260     92      0 R  15.3  0.0   0:03.60 stress
68593 root      20   0    7260     92      0 R  15.3  0.0   0:03.21 stress
68591 root      20   0    7260     92      0 R  13.6  0.0   0:03.55 stress
68595 root      20   0    7260     92      0 R  13.6  0.0   0:03.40 stress
68597 root      20   0    7260     92      0 R  12.6  0.0   0:03.12 stress
68598 root      20   0    7260     92      0 R  12.6  0.0   0:03.38 stress
68590 root      20   0    7260     92      0 R  12.3  0.0   0:03.13 stress
68592 root      20   0    7260     92      0 R  12.3  0.0   0:03.22 stress
68594 root      20   0    7260     92      0 R  12.3  0.0   0:03.07 stress
68596 root      20   0    7260     92      0 R  12.3  0.0   0:03.09 stress
68535 root      20   0    7260     96      0 R   7.6  0.0   0:02.09 stress
68532 root      20   0    7260     96      0 R   7.0  0.0   0:01.78 stress
68534 root      20   0    7260     96      0 R   7.0  0.0   0:01.73 stress
68527 root      20   0    7260     96      0 R   6.6  0.0   0:01.85 stress
68528 root      20   0    7260     96      0 R   6.6  0.0   0:01.87 stress
68529 root      20   0    7260     96      0 R   6.6  0.0   0:01.90 stress
68536 root      20   0    7260     96      0 R   6.6  0.0   0:01.71 stress
68530 root      20   0    7260     96      0 R   6.0  0.0   0:01.71 stress
68531 root      20   0    7260     96      0 R   6.0  0.0   0:01.73 stress
68533 root      20   0    7260     96      0 R   6.0  0.0   0:01.66 stress
```

3）cpu 周期限制（作为了解）

▪7.3、CPU 周期限制

docker 提供了--cpu-period、--cpu-quota 两个参数控制容器可以分配到的 CPU 时钟周期。

- --cpu-period 是用来指定容器对 CPU 的使用要在多长时间内做一次重新分配。
- --cpu-quota 是用来指定在这个周期内，最多可以有多少时间用来跑这个容器。与 --cpu-shares 不同的是：这种配置是指定一个绝对值，而且没有弹性在里面，容器对 CPU 资源的使用绝对不会超过配置的值。

cpu-period 和 cpu-quota 的单位为微秒（μs）。cpu-period 的最小值为 1000 微秒，最大值为 1 秒（10^6 μs），默认值为 0.1 秒（100000 μs）。cpu-quota 的值默认为-1，表示不做控制。cpu-period、cpu-quota 这两个参数一般联合使用。

举个例子，如果容器进程需要每 1 秒使用单个 CPU 的 0.2 秒时间，可以将 cpu-period 设置为 1000000（即 1 秒），cpu-quota 设置为 200000（0.2 秒）。当然，在多核情况下，如果允许容器进程需要完全占用两个 CPU，则可以将 cpu-period 设置为 100000（即 0.1 秒），cpu-quota 设置为 200000（0.2 秒）。

```
[root@localhost ~]# docker run -it --cpu-period 1000000 --cpu-quota 200000 centos:stress
/bin/bash
[root@2474ee0f7e62 /]# cat /sys/fs/cgroup/cpu/cpu.cfs_period_us
1000000
[root@2474ee0f7e62 /]# cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
200000
```

4）cpu 的核心限制（重点）

多核 cpu 的服务器 docker 还可以控制容器运行限定使用那些 cpu 内核，可以使

用--cpuset-cpus 选项来使某个程序独享 cpu 核心，以便提高器其处理的速度，对应的 Cgoup 文件为/ect/fs/cgroup/cpuset/docker 容器 ID 号/cpuset.spus。选项后直接跟参数 1.2.3.…表示第一个内核，第二个内核，第三个内核，与/proc/cpuinfo 中的标号相同

如果服务器有 16 个核心，那么 cpu 编号为 0~15，使新建容器绑定第 1~4 核使用：

```
[root@server04 ~]# Dockere run --cpuset-cpus 0.1.2.3 centos:stress /bin/bash
或者
[root@server04 ~]# Docker run --itd --cpuset-cpus 0-3 centos:stress /bin/bash （两种方式）
```

那么该容器内进程只会在 0.1.2.3 的 cpu 上运行

通过以下指令可以查看出容器中的进程与 cpu 的绑定关系，可以认为达到了绑定 cpu 内核的目的

```
[root@server04 ~]#docker exec ID 号 taskset -c -p 1
pid 1’s current affinity list: 0-3  //容器内部第一个进程编号一般为 1
```

尽量使用绑定内核的方式分配 cpu 资源给容器进程使用，然后再配合--cpu-shares 选项动态调整 cpu 使用资源的比例

5）cpu 配额控制参数的混合案例

通过--cpuset-cpus 指定容器 A 使用 CPU 内核 0，容器 B 只是用 CPU 内核 1。在主机上只有这两个容器使用对应 CPU 内核的情况，它们各自占用全部的内核资

源，--cou-shares 没有明显效果。

--cpuset.cpus、--cpuset-mems 参数只在多核、多内存节点上的服务器上有效，并且必须与实际的物理配置匹配，否则也无法达到资源控制的目的。

在系统具有多个 CPU 内核的情况下，需要通过 cpuset-cpus 为容器 CPU 内核才能比较方便地进行测试。

用下例命令创建测试用的容器

```
[root@server04~]# docker run --itd --name cpu3 --cpuset-cpus3 --cpu-shares 512 centos:stress stress -c 1

[root@server04~]# docker exec -it 容器 ID /bin/bash
```

输入 top 命令查看

```
top - 08:47:56 up 57 days,  5:11,  0 users,  load average: 2.00, 1.87, 4.65
Tasks:   4 total,   2 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.0 us,  0.0 sy,  0.0 ni, 99.3 id,  0.3 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu1  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 98586928 total, 94671680 free,   988612 used,  2926644 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used. 96336288 avail Mem
```

```
[root@server04~]# docker run -itd --name cpu 4 --cpuset-cpus 3 --cpu-shares 1024 centos:strees stress -c 1

[root@server04~]# docker exec -it 容器 ID /bin/bash
```

输入 top 命令查看

```
top - 08:47:12 up 57 days,  5:10,  0 users,  load average: 2.00, 1.85, 4.78
Tasks:   4 total,   2 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu0  :  0.3 us,  0.0 sy,  0.0 ni, 99.3 id,  0.3 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu1  :  0.3 us,  0.3 sy,  0.0 ni, 99.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  0.3 us,  0.0 sy,  0.0 ni, 99.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 98586928 total, 94671408 free,   988884 used,  2926636 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used. 96336008 avail Mem
```

6）内存限制

与操作系统类似，容器可使用的内存包括两部分：物理内存和 swap。Docker 通过下面两组参数来控制容器内存的使用量。↵

-m 或--memory:设置内存的使用限额。例如 100M，1024M

---memory-swap: 设置内存的 swap 的使用限额

当执行如下命令

```
[root@server04~]# docker run -it -m 200M --memory-swap=300M progrium/stress --vm 1 --vm-bytes 280M  #下载了一个新的镜像
```

--vm 1：启动一个内存工作线程

--vm-nytes 280M:每个线程分配 280M 内存

7）block IO 限制

默认的情况的下，所有容器能平等的读写磁盘，可以通过设置 --blkio-weight 参数来改变容器 block IO 的优先级

--blkio-weight 与--cpu-shares 类似，设置相对权重值。默认为 500

再下面的例子中，容器 A 读写磁盘的带宽是容器 B 的两倍

```
[root@localhost ~]# docker run -it --name container_A --blkio-weight 600 centos:stress /bin/bash
[root@a4b63e72a03b /]# cat /sys/fs/cgroup/blkio/blkio.weight
600
```

```
[root@localhost ~]# docker run -it --name container_B --blkio-weight 300 centos:stress /bin/bash
[root@ee688a3bba2b /]# cat /sys/fs/cgroup/blkio/blkio.weight
300
```

8）限制 bps 和 iops

如果在一台服务器上进行容器的混合部署，那么会存在同时几个程序写磁盘数据的情况，这时可以通过--device-write-iops 选项来限制每秒写 io 次数来限制制定设备的写速度。相应的还有--device-read-iops 选项可以限制读取 IO 的速度，但是这种方法只能限制设备，而不能限制分区，相应的 Cgroup 写配置文件为 /sys/fs/cgroup/blkio/ 容器 ID/blkio.throttle.write_iops_device

- bps 是 byte per second，每秒读写的数据量。
- iops 是 io per second，每秒 IO 的次数。

可通过以下参数控制容器的 bps 和 iops：

- --device-read-bps，限制读某个设备的 bps
- --device-write-bps，限制写某个设备的 bps
- --device-read-iops，限制读某个设备的 iops
- --device-write-iops，限制写某个设备的 iops

下面的示例是限制容器写/dev/sda 的速率为 5 MB/s。

```
[root@localhost ~]# docker run -it --device-write-bps /dev/sda:5MB centos:stress /bin/bash
[root@7fc31cd1514b /]# dd if=/dev/zero of=test bs=1M count=100 oflag=direct
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 20.0051 s, 5.2 MB/s
```

通过 dd 测试在容器中写磁盘的速度。因为容器的文件系统是在 host /dev/sda 上的，在容器中写文件相当于对宿主机/dev/sda 进行写操作。另外，oflag=direct 指定用 direct IO 方式写文件，这样--device-write-bps 才能生效。

结果表明，bps 5.2 MB/s 在限速 5MB/s 左右。作为对比测试，如果不限速，结果如下：

```
[root@localhost ~]# docker run -it centos:stress /bin/bash
[root@9128099a8947 /]# dd if=/dev/zero of=test bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (105 MB) copied, 0.384372 s, 273 MB/s
```