Docker资源管理探秘：Docker背后的内核Cgroups机制



| 作者 [**孙远**](http://www.infoq.com/cn/profile/%E5%AD%99%E8%BF%9C)[**薛婉菊**](http://www.infoq.com/cn/profile/%E8%96%9B%E5%A9%89%E8%8F%8A) 发布于 2016年12月5日. 估计阅读时间: 13 分钟 | 道AI风控、Serverless架构、EB级存储引擎，尽在[**ArchSummit**](http://sz2017.archsummit.com/?utm_source=infoq&utm_medium=notice)![讨论](http://www.infoq.com/cn/articles/docker-resource-management-cgroups#theCommentsSection)

* [分享到：微博微信FacebookTwitter有道云笔记邮件分享](http://www.infoq.com/cn/articles/docker-resource-management-cgroups)
* [**稍后阅读**](http://www.infoq.com/cn/articles/docker-resource-management-cgroups)
* [**我的阅读清单**](http://www.infoq.com/cn/showbookmarks.action)

随着Docker技术被越来越多的个人、企业所接受，其用途也越来越广泛。Docker资源管理包含对CPU、内存、IO等资源的限制，但大部分Docker使用者在使用资源管理接口时往往只知其然而不知其所以然。  
本文将介绍Docker资源管理背后的Cgroups机制，并且列举每一个资源管理接口对应的Cgroups接口，让Docker使用者对资源管理知其然并且知其所以然。

1.Docker资源管理接口概览

|  |  |
| --- | --- |
| 格式 | 描述 |
| -m, --memory=" <数字>[<单位>]" | 内存使用限制。 数字需要使用整数，对应的单位是b, k, m, g中的一个。最小取值是4M。 |
| --memory-swap="<数字>[<单位>]" | 总内存使用限制 (物理内存 + 交换分区，数字需要使用整数，对应的单位是b, k, m, g中的一个。 |
| --memory-reservation="<数字>[<单位>]" | 内存软限制。 数字需要使用正整数，对应的单位是b, k, m, g中的一个。 |
| --kernel-memory="<数字>[<单位>]" | 内核内存限制。 数字需要使用正整数，对应的单位是b, k, m, g中的一个。最小取值是4M。 |
| --oom-kill-disable=false | 内存耗尽时是否杀掉容器 |
| --memory-swappiness="" | 调节容器内存使用交换分区的选项，取值为0和100之间的整数(含0和100)。 |
| -c, --cpu-shares=0 | CPU份额 (相对权重) |
| --cpu-period=0 | 完全公平算法中的period值 |
| --cpu-quota=0 | 完全公平算法中的quota值 |
| --cpuset-cpus="<数字>" | 限制容器使用的cpu核(0-3, 0,1) |
| --cpuset-mems="" | 限制容器使用的内存节点，该限制仅仅在NUMA系统中生效。 |
| --blkio-weight=0 | 块设备IO相对权重，取值在10值1000之间的整数（包含10和1000） |
| --blkio-weight-device="设备名称:权重值" | 指定的块设备的IO相对权重 |
| --device-read-bps="<设备路径>:<数字>[<单位>]" | 限制对某个设备的读取速率 ，数字需要使用正整数，单位是kb, mb, or gb中的一个。 |
| --device-write-bps="<设备路径>:<数字>[<单位>]" | 限制对某个设备的写速率 ，数字需要使用正整数，单位是kb, mb, or gb中的一个。 |
| --device-read-iops="<设备路径>:<数字>" | 限制对某个设备每秒IO的读取速率，数字需要使用正整数。 |
| --device-write-iops="<设备路径>:<数字>" | 限制对某个设备每秒IO的写速率，数字需要使用正整数。 |

2. Docker资源管理原理——Cgroups子系统介绍

**相关厂商内容**

[**性能优化最佳实践经验谈**](http://www.infoq.com/cn/vendorcontent/show.action?vcr=4268&utm_source=infoq&utm_medium=VCR&utm_campaign=vcr_articles_click)

[**领英大数据在线/离线生态系统中的均衡算法**](http://www.infoq.com/infoq/url.action?i=16025&t=f)

[**京东协同创新：容器平台+数据中心基础设施**](http://www.infoq.com/infoq/url.action?i=16026&t=f)

[**小米监控全生命周期如何实现全自动管理？**](http://www.infoq.com/infoq/url.action?i=16027&t=f)

[**腾讯十年运维包袱与四大创新术**](http://www.infoq.com/infoq/url.action?i=16028&t=f)

**相关赞助商**

[http://cdn4.infoqstatic.com/statics_s1_20170502-0319/resource/sponsorship/featuredcategory/6555/ASBOXtupian.jpg](http://www.infoq.com/infoq/url.action?i=16030&t=f)

**ArchSummit深圳2017，7月7-8日，深圳·华侨城洲际酒店，**[精彩内容抢先看](http://www.infoq.com/infoq/url.action?i=16029&t=f)

Cgroups是control groups的缩写，最初由google的工程师提出，后来被整合进Linux内核。Cgroups是Linux内核提供的一种可以限制、记录、隔离进程组（process groups）所使用的物理资源（如：CPU、内存、IO等）的机制。Cgroups由7个子系统组成：分别是cpuset、cpu、cpuacct、blkio、devices、freezer、memory。不同类型资源的分配和管理是由各个cgroup子系统负责完成的。

下面介绍与docker资源管理接口相关的4个子系统。

2.1 memory -- 用来限制cgroup中的任务所能使用的内存上限。

|  |  |  |
| --- | --- | --- |
| 子系统常用cgroups接口 | 描述 | 对应的docker接口 |
| cgroup/memory/memory. limit\_in\_bytes | 设定内存上限，单位是字节，也可以使用k/K、m/M或者g/G表示要设置数值的单位。 | -m, --memory="" |
| cgroup/memory/memory. memsw.limit\_in\_bytes | 设定内存加上交换分区的使用总量。通过设置这个值，可以防止进程把交换分区用光。 | --memory-swap="" |
| cgroup/memory/memory. soft\_limit\_in\_bytes | 设定内存限制，但这个限制并不会阻止进程使用超过限额的内存，只是在系统内存不足时，会优先回收超过限额的进程占用的内存，使之向限定值靠拢。 | --memory-reservation="" |
| cgroup/memory/memory. kmem.limit\_in\_bytes | 设定内核内存上限。 | --kernel-memory="" |
| cgroup/memory/memory. oom\_control | 如果设置为0，那么在内存使用量超过上限时，系统不会杀死进程，而是阻塞进程直到有内存被释放可供使用时，另一方面，系统会向用户态发送事件通知，用户态的监控程序可以根据该事件来做相应的处理，例如提高内存上限等。 | --oom-kill-disable="" |
| cgroup/memory/memory. swappiness | 控制内核使用交换分区的倾向。取值范围是0至100之间的整数（包含0和100）。值越小，越倾向使用物理内存。 | --memory-swappiness="" |

2.2 cpu -- 使用调度程序提供对 CPU 的 cgroup 任务访问。

|  |  |  |
| --- | --- | --- |
| 子系统常用cgroups接口 | 描述 | 对应的docker接口 |
| cgroup/cpu/cpu. shares | 负责CPU比重分配的接口。假设我们在cgroupfs的根目录下创建了两个cgroup（C1和C2），并且将cpu.shares分别配置为512和1024，那么当C1和C2争用CPU时，C2将会比C1得到多一倍的CPU占用率。要注意的是，只有当它们争用CPU时CPU share才会起作用，如果C2是空闲的，那么C1可以得到全部的CPU资源。 | -c, --cpu-shares="" |
| cgroup/cpu/cpu. cfs\_period\_us | 负责CPU带宽限制，需要与cpu.cfs\_quota\_us搭配使用。我们可以将period设置为1秒，将quota设置为0.5秒，那么cgroup中的进程在1秒内最多只能运行0.5秒，然后就会被强制睡眠，直到下一个1秒才能继续运行。 | --cpu-period="" |
| cgroup/cpu/cpu. cfs\_quota\_us | 负责CPU带宽限制，需要与cpu.cfs\_period\_us搭配使用。 | --cpu-quota="" |

2.3 cpuset -- 为 cgroup 中的任务分配独立 CPU（在多核系统）和内存节点

|  |  |  |
| --- | --- | --- |
| 子系统常用cgroups接口 | 描述 | 对应的docker接口 |
| cgroup/cpuset/cpuset.cpus | 允许进程使用的CPU列表（例如：0-4,9）。 | --cpuset-cpus="" |
| cgroup/cpuset/cpuset.mems | 允许进程使用的内存节点列表（例如：0-1）。 | --cpuset-mems="" |

2.4 blkio -- 为块设备设定输入/输出限制，比如物理设备（磁盘、固态硬盘、USB等）

|  |  |  |
| --- | --- | --- |
| 子系统常用cgroups接口 | 描述 | 对应的docker接口 |
| cgroup/blkio/blkio. weight | 设置权重值，取值范围是10至1000之间的整数（包含10和1000）。这跟cpu.shares类似，是比重分配，而不是绝对带宽的限制，因此只有当不同的cgroup在争用同一个块设备的带宽时，才会起作用。 | --blkio-weight="" |
| cgroup/blkio/blkio. weight\_device | 对具体的设备设置权重值，这个值会覆盖上述的blkio.weight。 | --blkio-weight-device="" |
| cgroup/blkio/blkio. throttle.read\_bps\_device | 对具体的设备，设置每秒读块设备的带宽上限。 | --device-read-bps="" |
| cgroup/blkio/blkio. throttle.write\_bps\_device | 设置每秒写块设备的带宽上限。同样需要指定设备。 | --device-write-bps="" |
| cgroup/blkio/blkio. throttle.read\_iops\_device | 设置每秒读块设备的IO次数的上限。同样需要指定设备。 | --device-read-iops="" |
| cgroup/blkio/blkio. throttle.write\_iops\_device | 设置每秒写块设备的IO次数的上限。同样需要指定设备。 | --device-write-iops="" |

3.Docker资源管理接口详解及应用示例

以下内容针对各资源管理接口做了详尽的说明。为了加深读者理解，部分接口附有测试用例。用例中的Docker版本为1.11.0。如果在你的镜像中stress命令不可用，你可以通过sudo apt-get install stress来安装stress工具。

3.1 memory子系统

3.1.1 -m, --memory=""

可以限制容器使用的内存量，对应的cgroup文件是cgroup/memory/memory.limit\_in\_bytes。  
取值范围:大于等于4M  
单位：b,k,m,g

在默认情况下，容器可以占用无限量的内存，直至主机内存资源耗尽。  
运行如下命令来确认容器内存的资源管理对应的cgroup文件。

$ docker run -it --memory 100M ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes"

104857600

可以看到，当内存限定为100M时，对应的cgroup文件数值为104857600，该数值的单位为字节，即104857600字节等于100M。

本机内存环境为：

$ free

total used free shared buff/cache available

Mem: 4050284 254668 3007564 180484 788052 3560532

Swap: 0 0 0

值得注意的是本机目前没有配置交换分区(swap)。

我们使用stress工具来证明内存限定已经生效。stress是一个压力工具，如下命令将要在容器内创建一个进程，在该进程中不断的执行占用内存(malloc)和释放内存(free)的操作。在理论上如果占用的内存少于限定值，容器会工作正常。注意，如果试图使用边界值，即试图在容器中使用stress工具占用100M内存，这个操作通常会失败，因为容器中还有其他进程在运行。

$ docker run -ti -m 100M ubuntu:14.04 stress --vm 1 --vm-bytes 50M

stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd

当在限定内存为100M的容器中，试图占用50M的内存时，容器工作正常。  
如下所示，当试图占用超过100M内存时，必然导致容器异常。

$ docker run -ti -m 100M ubuntu:14.04 stress --vm 1 --vm-bytes 101M

stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd

stress: FAIL: [1] (416) <-- worker 6 got signal 9

stress: WARN: [1] (418) now reaping child worker processes

stress: FAIL: [1] (422) kill error: No such process

stress: FAIL: [1] (452) failed run completed in 0s

注意这种情况是在系统无交换分区(swap)的情况下出现的，如果我们添加了交换分区，情况又会怎样？首先通过如下命令来添加交换分区(swap)。

$ dd if=/dev/zero of=/tmp/mem.swap bs=1M count=8192

8192+0 records in

8192+0 records out

8589934592 bytes (8.6 GB) copied, 35.2693 s, 244 MB/s

$ mkswap /tmp/mem.swap

Setting up swapspace version 1, size = 8388604 KiB

no label, UUID=55ea48e9-553d-4013-a2ae-df194f7941ed

$ sudo swapon /tmp/mem.swap

swapon: /tmp/mem.swap: insecure permissions 0664, 0600 suggested.

swapon: /tmp/mem.swap: insecure file owner 1100, 0 (root) suggested.

$ free -m

total used free shared buff/cache available

Mem: 3955 262 28 176 3665 3463

Swap: 8191 0 8191

之后再次尝试占用大于限定的内存。

$ docker run -ti -m 100M ubuntu:14.04 stress --vm 1 --vm-bytes 101M

stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd

在加入交换分区后容器工作正常，这意味着有部分存储在内存中的信息被转移到了交换分区中了。  
注意，在实际容器使用场景中，如果不对容器使用内存量加以限制的话，可能导致一个容器会耗尽整个主机内存，从而导致系统不稳定。所以在使用容器时务必对容器内存加以限制。

3.1.2 --memory-swap=""

可以限制容器使用交换分区和内存的总和，对应的cgroup文件是cgroup/memory/memory.memsw.limit\_in\_bytes。  
取值范围:大于内存限定值  
单位：b,k,m,g

运行如下命令来确认容器交换分区的资源管理对应的cgroup文件。

$ docker run -ti -m 300M --memory-swap 1G ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

1073741824

可以看到，当memory-swap限定为1G时，对应的cgroup文件数值为1073741824，该数值的单位为字节，即1073741824B等于1G。

|  |  |
| --- | --- |
| 条件 | 结果 |
| **memory=无穷大, memory-swap=无穷大** (默认条件下) | 系统不限定容器对内存和交换分区的使用量，容器能够使用主机所能提供的所有内存。 |
| **memory=L<无穷大, memory-swap=无穷大** | (设定memory限定值同时将memory-swap设置为-1) 容器的内存使用量不能超过L，但是交换分区的使用量不受限制(前提是主机支持交换分区)。 |
| **memory=L<无穷大, memory-swap=2\*L** | (设定memory限定值而不设置memory-swap值) 容器的内存使用量不能超过L，而内存使用量和交换分区的使用量不能超过两倍的L。 |
| **memory=L<无穷大, memory-swap=S<无穷大, L<=S** | (设定了memory和memory-swap的限定值) 容器的内存使用量不能超过L，而内存使用量和交换分区的使用量不能超过S。 |

例子：  
以下命令没有对内存和交换分区进行限制，这意味着容器可以使用无限多的内存和交换分区。

$ docker run -it ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes && cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

9223372036854771712

9223372036854771712

以下命令只限定了内存使用量300M，而没有限制交换分区使用量(-1意味着不做限制)。

$ docker run -it -m 300M --memory-swap -1 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes && cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

314572800

9223372036854771712

以下命令仅仅限定了内存使用量，这意味着容器能够使用300M的内存和300M的交换分区。在默认情况下，总的内存限定值(内存+交换分区)被设置为了内存限定值的两倍。

$ docker run -it -m 300M ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes && cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

314572800

629145600

以下命令限定了内存和交换分区的使用量，容器可以使用300M的内存和700M的交换分区。

$ docker run -it -m 300M --memory-swap 1G ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes && cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

314572800

1073741824

当memory-swap限定值低于memory限定值时，系统提示"Minimum memoryswap limit should be larger than memory limit"错误。

$ docker run -it -m 300M --memory-swap 200M ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.limit\_in\_bytes && cat /sys/fs/cgroup/memory/memory.memsw.limit\_in\_bytes"

docker: Error response from daemon: Minimum memoryswap limit should be larger than memory limit, see usage..

See 'docker run --help'.

如下所示，当尝试占用的内存数量超过memory-swap值时，容器出现异常。

$ docker run -ti -m 100M --memory-swap 200M ubuntu:14.04 stress --vm 1 --vm-bytes 201M

stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd

stress: FAIL: [1] (416) <-- worker 7 got signal 9

stress: WARN: [1] (418) now reaping child worker processes

stress: FAIL: [1] (422) kill error: No such process

stress: FAIL: [1] (452) failed run completed in 0s

如下所示，当占用内存值大于memory限定值但小于memory-swap时，容器运行正常。

$ docker run -ti -m 100M --memory-swap 200M ubuntu:memory stress --vm 1 --vm-bytes 180M

stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd

3.1.3 --memory-reservation=""

取值范围:大于等于0的整数  
单位：b,k,m,g  
对应的cgroup文件是cgroup/memory/memory.soft\_limit\_in\_bytes。

$ docker run -ti --memory-reservation 50M ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.soft\_limit\_in\_bytes"

52428800

通常情况下，容器能够使用的内存量仅仅由-m/--memory选项限定。如果设置了--memory-reservation选项，当内存使用量超过--memory-reservation选项所设定的值时，系统会强制容器执行回收内存的操作，使得容器内存消耗不会长时间超过--memory-reservation的限定值。

这个限制并不会阻止进程使用超过限额的内存，只是在系统内存不足时，会回收部分内存，使内存使用量向限定值靠拢。  
在以下命令中，容器对内存的使用量不会超过500M，这是硬性限制。当内存使用量大于200M而小于500M时，系统会尝试回收部分内存，使得内存使用量低于200M。

$ docker run -it -m 500M --memory-reservation 200M ubuntu:14.04 bash

在如下命令中，容器使用的内存量不受限制，但容器消耗的内存量不会长时间超过1G，因为当容器内存使用量超过1G时，系统会尝试回收内存使内存使用量低于1G。

$ docker run -it --memory-reservation 1G ubuntu:14.04 bash

3.1.4 --kernel-memory=""

该接口限制了容器对内核内存的使用，对应的cgroup文件是cgroup/memory/memory.kmem.limit\_in\_bytes。

$ docker run -ti --kernel-memory 50M ubuntu:14.04 bash -c "cat /sys/fs/cgroup/memory/memory.kmem.limit\_in\_bytes"

52428800

如下命令可以限定容器最多可以使用500M的内存。在500M内存中，内核内存最多可以占用50M。

$ docker run -it -m 500M --kernel-memory 50M ubuntu:14.04 bash

如下命令可以限定容器最多可以使用50M的内核内存，而用户空间的内存使用量不受限制。

$ docker run -it --kernel-memory 50M ubuntu:14.04 bash

3.1.5 --oom-kill-disable=false

当out-of-memory (OOM)发生时，系统会默认杀掉容器进程，如果你不想让容器进程被杀掉，可以使用该接口。接口对应的cgroup文件是cgroup/memory/memory.oom\_control。

当容器试图使用超过限定大小的内存值时，就会触发OOM。此时会有两种情况，第一种情况是当接口--oom-kill-disable=false的时候，容器会被杀掉；第二种情况是当接口--oom-kill-disable=true的时候，容器会被挂起。

以下命令设置了容器的的内存使用限制为20M，将--oom-kill-disable接口的值设置为true。查看该接口对应的cgroup文件，oom\_kill\_disable的值为1。

$ docker run -m 20m --oom-kill-disable=true ubuntu:14.04 bash -c 'cat /sys/fs/cgroup/memory/memory.oom\_control'

oom\_kill\_disable 1

under\_oom 0

oom\_kill\_disable：取值为0或1，当值为1的时候表示当容器试图使用超出内存限制时（即20M），容器会挂起。  
under\_oom：取值为0或1，当值为1的时候，OOM已经出现在容器中。

通过x=a; while true; do x=xx; done命令来耗尽内存并强制触发OOM，log如下所示。

$ docker run -m 20m --oom-kill-disable=false ubuntu:14.04 bash -c 'x=a; while true; do x=$x$x$x$x; done'

$ echo $?

137

通过上面的log可以看出,当容器的内存耗尽的时候，容器退出，退出码为137。因为容器试图使用超出限定的内存量，系统会触发OOM，容器会被杀掉，此时under\_oom的值为1。我们可以通过系统中cgroup文件(/sys/fs/cgroup/memory/docker/${container\_id}/memory.oom\_control)查看under\_oom的值（oom\_kill\_disable 1，under\_oom 1）。

当--oom-kill-disable=true的时候，容器不会被杀掉，而是被系统挂起。

$ docker run -m 20m --oom-kill-disable=true ubuntu:14.04 bash -c 'x=a; while true; do x=$x$x$x$x; done'

3.1.6 --memory-swappiness=""

该接口可以设定容器使用交换分区的趋势，取值范围为0至100的整数（包含0和100）。0表示容器不使用交换分区，100表示容器尽可能多的使用交换分区。对应的cgroup文件是cgroup/memory/memory.swappiness。

$ docker run --memory-swappiness=100 ubuntu:14.04 bash -c 'cat /sys/fs/cgroup/memory/memory.swappiness'

100

3.2 cpu子系统

3.2.1 -c, --cpu-shares=0

对应的cgroup文件是cgroup/cpu/cpu.shares。

$ docker run --rm --cpu-shares 1600 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/cpu/cpu.shares"

1600

通过--cpu-shares可以设置容器使用CPU的权重，这个权重设置是针对CPU密集型的进程的。如果某个容器中的进程是空闲状态，那么其它容器就能够使用本该由空闲容器占用的CPU资源。也就是说，只有当两个或多个容器都试图占用整个CPU资源时，--cpu-shares设置才会有效。  
我们使用如下命令来创建两个容器，它们的权重分别为1024和512。

$ docker run -ti --cpu-shares 1024 ubuntu:14.04 stress -c 1

stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

$ docker run -ti --cpu-shares 512 ubuntu:14.04 stress -c 1

stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

从如下top命令的log可以看到，第一个容器产生的进程PID为1418，CPU占用率为66.1%，第二个容器产生进程PID为1471，CPU占用率为32.9%。两个容器CPU占用率约为2:1的关系，测试结果与预期相符。

top - 18:51:50 up 9 days, 2:07, 0 users, load average: 0.62, 0.15, 0.05

Tasks: 84 total, 3 running, 81 sleeping, 0 stopped, 0 zombie

%Cpu(s): 90.4 us, 2.2 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 7.4 si, 0.0 st

KiB Mem : 2052280 total, 71468 free, 117284 used, 1863528 buff/cache

KiB Swap: 0 total, 0 free, 0 used. 1536284 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

1418 root 20 0 7312 100 0 R 66.1 0.0 0:22.92 stress

1471 root 20 0 7312 96 0 R 32.9 0.0 0:04.97 stress

3.2.2 --cpu-period=""

内核默认的Linux 调度CFS（完全公平调度器）周期为100ms,我们通过--cpu-period来设置容器对CPU的使用周期，同时--cpu-period接口需要和--cpu-quota接口一起来使用。--cpu-quota接口设置了CPU的使用值。CFS(完全公平调度器) 是内核默认使用的调度方式，为运行的进程分配CPU资源。对于多核CPU，根据需要调整--cpu-quota的值。

对应的cgroup文件是cgroup/cpu/cpu.cfs\_period\_us。以下命令创建了一个容器，同时设置了该容器对CPU的使用时间为50000（单位为微秒），并验证了该接口对应的cgroup文件对应的值。

$ docker run -ti --cpu-period 50000 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/cpu/cpu.cfs\_period\_us"

50000

以下命令将--cpu-period的值设置为50000,--cpu-quota的值设置为25000。该容器在运行时可以获取50%的cpu资源。

$ docker run -ti --cpu-period=50000 --cpu-quota=25000 ubuntu:14.04 stress -c 1

stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

从log的最后一行中可以看出，该容器的cpu使用率约为50.0%。

top - 10:36:55 up 6 min, 0 users, load average: 0.49, 0.21, 0.10

Tasks: 68 total, 2 running, 66 sleeping, 0 stopped, 0 zombie

%Cpu(s): 49.3 us, 0.0 sy, 0.0 ni, 50.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

KiB Mem : 4050748 total, 3063952 free, 124280 used, 862516 buff/cache

KiB Swap: 0 total, 0 free, 0 used. 3728860 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

770 root 20 0 7312 96 0 R 50.0 0.0 0:38.06 stress

3.2.3 --cpu-quota=0

对应的cgroup文件是cgroup/cpu/cpu.cfs\_quota\_us。

$ docker run --cpu-quota 1600 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/cpu/cpu.cfs\_quota\_us"

1600

--cpu-quota接口设置了CPU的使用值，通常情况下它需要和--cpu-period接口一起来使用。具体使用方法请参考--cpu-period选项。

3.3 cpuset子系统

3.3.1 --cpuset-cpus=""

该接口对应的cgroup文件是cgroup/cpuset/cpuset.cpus。

在多核CPU的虚拟机中，启动一个容器，设置容器只使用CPU核1，并查看该接口对应的cgroup文件会被修改为1，log如下所示。

$ docker run -ti --cpuset-cpus 1 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/cpuset/cpuset.cpus"

1

通过以下命令指定容器使用cpu核1，并通过stress命令加压。

$ docker run -ti --cpuset-cpus 1 ubuntu:14.04 stress -c 1

查看CPU资源的top命令的log如下所示。需要注意的是，输入top命令并按回车键后，再按数字键1，终端才能显示每个CPU的状态。

top - 11:31:47 up 5 days, 21:00, 0 users, load average: 0.62, 0.82, 0.77

Tasks: 104 total, 3 running, 101 sleeping, 0 stopped, 0 zombie

%Cpu0 : 0.0 us, 0.0 sy, 0.0 ni, 99.6 id, 0.0 wa, 0.0 hi, 0.4 si, 0.0 st

%Cpu1 :100.0 us, 0.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu2 : 0.3 us, 0.3 sy, 0.0 ni, 99.3 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st

%Cpu3 : 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 : 2051888 total, 1130220 free, 127972 used, 793696 buff/cache

KiB Swap: 33554416 total, 33351848 free, 202568 used. 1739888 avail Mem

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

10266 root 20 0 7312 96 0 R 100.0 0.0 0:11.92 stress

从以上log得知，只有CPU核1的负载为100%，而其它CPU核处于空闲状态，结果与预期结果相符。

3.3.2 --cpuset-mems=""

该接口对应的cgroup文件是cgroup/cpuset/cpuset.mems。

$ docker run -ti --cpuset-mems=0 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/cpuset/cpuset.mems"

0

以下命令将限制容器进程使用内存节点1、3的内存。

$ docker run -it --cpuset-mems="1,3" ubuntu:14.04 bash

以下命令将限制容器进程使用内存节点0、1、2的内存。

$ docker run -it --cpuset-mems="0-2" ubuntu:14.04 bash

3.4 blkio子系统

3.4.1 --blkio-weight=0

通过--blkio-weight接口可以设置容器块设备IO的权重，有效值范围为10至1000的整数(包含10和1000)。默认情况下，所有容器都会得到相同的权重值(500)。对应的cgroup文件为cgroup/blkio/blkio.weight。以下命令设置了容器块设备IO权重为10，在log中可以看到对应的cgroup文件的值为10。

$ docker run -ti --rm --blkio-weight 10 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.weight"

10

通过以下两个命令来创建不同块设备IO权重值的容器。

$ docker run -it --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash

$ docker run -it --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash

如果在两个容器中同时进行块设备操作（例如以下命令）的话，你会发现所花费的时间和容器所拥有的块设备IO权重成反比。

$ time dd if=/mnt/zerofile of=test.out bs=1M count=1024 oflag=direct

3.4.2 --blkio-weight-device=""

通过--blkio-weight-device="设备名:权重"接口可以设置容器对特定块设备IO的权重，有效值范围为10至1000的整数(包含10和1000)。  
对应的cgroup文件为cgroup/blkio/blkio.weight\_device。

$ docker run --blkio-weight-device "/dev/sda:1000" ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.weight\_device"

8:0 1000

以上log中的"8：0"表示sda的设备号，可以通过stat命令来获取某个设备的设备号。从以下log中可以查看到/dev/sda对应的主设备号为8，次设备号为0。

$ stat -c %t:%T /dev/sda

8:0

如果--blkio-weight-device接口和--blkio-weight接口一起使用，那么Docker会使用--blkio-weight值作为默认的权重值，然后使用--blkio-weight-device值来设定指定设备的权重值，而早先设置的默认权重值将不在这个特定设备中生效。

$ docker run --blkio-weight 300 --blkio-weight-device "/dev/sda:500" ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.weight\_device"

8:0 500

通过以上log可以看出，当--blkio-weight接口和--blkio-weight-device接口一起使用的时候，/dev/sda设备的权重值由--blkio-weight-device设定的值来决定。

3.4.3 --device-read-bps=""

该接口用来限制指定设备的读取速率，单位可以是kb、mb或者gb。对应的cgroup文件是cgroup/blkio/blkio.throttle.read\_bps\_device。

$ docker run -it --device /dev/sda:/dev/sda --device-read-bps /dev/sda:1mb ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.throttle.read\_bps\_device"

8:0 1048576

以上log中显示8:0 1000,8:0表示/dev/sda, 该接口对应的cgroup文件的值为1048576，是1MB所对应的字节数，即1024的平方。

创建容器时通过--device-read-bps接口设置设备读取速度为1MB/s。从以下log中可以看出,读取速度被限定为1.0MB/s,与预期结果相符合。

$ docker run -it --device /dev/sda:/dev/sda --device-read-bps /dev/sda:1mB ubuntu:14.04 bash

root@df1de679fae4:/# dd iflag=direct,nonblock if=/dev/sda of=/dev/null bs=5M count=1

1+0 records in

1+0 records out

5242880 bytes (5.2 MB) copied, 5.00464 s, 1.0 MB/s

3.4.4 --device-write-bps=""

该接口用来限制指定设备的写速率，单位可以是kb、mb或者gb。对应的cgroup文件是cgroup/blkio/blkio.throttle.write\_bps\_device。

$ docker run -it --device /dev/sda:/dev/sda --device-write-bps /dev/sda:1mB ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.throttle.write\_bps\_device"

8:0 1048576

以上log中显示8:0 1000，8:0表示/dev/sda, 该接口对应的cgroup文件的值为1048576，是1MB所对应的字节数，即1024的平方。

创建容器时通过--device-write-bps接口设置设备写速度为1MB/s。从以下log中可以看出,读取速度被限定为1.0MB/s,与预期结果相符合。

限速操作：

$ docker run -it --device /dev/sda:/dev/sda --device-write-bps /dev/sda:1mb ubuntu:14.04 bash

root@18dc79b91cd4:/# dd oflag=direct,nonblock of=/dev/sda if=/dev/urandom bs=10K count=1000

1000+0 records in

1000+0 records out

10240000 bytes (10 MB) copied, 10.1987 s, 1.0 MB/s

3.4.5 --device-read-iops=""

该接口设置了设备的IO读取速率，对应的cgroup文件是cgroup/blkio/blkio.throttle.read\_iops\_device。

$ docker run -it --device /dev/sda:/dev/sda --device-read-iops /dev/sda:400 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.throttle.read\_iops\_device"

8:0 400

可以通过"--device-read-iops /dev/sda:400"来限定sda的IO读取速率(400次/秒)，log如下所示。

$ docker run -ti --device /dev/sda:/dev/sda --device-read-iops /dev/sda:400 ubuntu:14.04

root@71910742c445:/# dd iflag=direct,nonblock if=/dev/sda of=/dev/null bs=1k count=1000

1000+0 records in

1000+0 records out

1024000 bytes (1.0 MB) copied, 2.42874 s, 422 kB/s

通过上面的log信息可以看出，容器每秒IO的读取次数为400，共需要读取1000次（log第二行：count=1000），测试结果显示执行时间为2.42874秒，约为2.5(1000/400)秒， 与预期结果相符。

3.4.6 --device-write-iops=""

该接口设置了设备的IO写速率，对应的cgroup文件是cgroup/blkio/blkio.throttle.write\_iops\_device。

$ docker run -it --device /dev/sda:/dev/sda --device-write-iops /dev/sda:400 ubuntu:14.04 bash -c "cat /sys/fs/cgroup/blkio/blkio.throttle.write\_iops\_device"

8:0 400

可以通过"--device-write-iops /dev/sda:400"来限定sda的IO写速率(400次/秒)，log如下所示。

$ docker run -ti --device /dev/sda:/dev/sda --device-write-iops /dev/sda:400 ubuntu:14.04

root@ef88a516d6ed:/# dd oflag=direct,nonblock of=/dev/sda if=/dev/urandom bs=1K count=1000

1000+0 records in

1000+0 records out

1024000 bytes (1.0 MB) copied, 2.4584 s, 417 kB/s

通过上面的log信息可以看出，容器每秒IO的写入次数为400，共需要写1000次（log第二行：count=1000），测试结果显示执行时间为2.4584秒，约为2.5(1000/400)秒， 与预期结果相符。

4.总结

Docker的资源管理依赖于Linux内核Cgroups机制。理解Docker资源管理的原理并不难，读者可以根据自己兴趣补充一些有针对性的测试。关于Cgroups的实现机制已经远超本文的范畴。感兴趣的读者可以自行查看[相关文章](http://www.infoq.com/cn/articles/docker-kernel-knowledge-cgroups-resource-isolation)和内核手册。

作者简介

孙远，华为中央软件研究院资深工程师，硕士毕业，9年软件行业经验。目前在华为从事容器Docker项目的测试工作。工作涉及到功能测试、性能测试、压力测试、稳定性测试、安全测试、测试管理、工程能力构建等内容。参与编写了《Docker进阶与实战》的Docker测试章节。先前曾经就职于美国风河系统公司，作为team leader从事风河Linux产品测试工作。活跃于Docker社区和内核测试ltp社区，目前有大量测试用例被开源社区接收。  
研究方向：容器技术、Docker、Linux内核、软件测试、自动化测试、测试过程改进。

薛婉菊，中软国际科技服务有限公司软件测试工程师，4年软件行业经验。目前参与容器Docker项目的测试工作，工作涉及到容器功能测试、性能测试、压力测试等内容。  
研究方向：容器技术、Docker、自动化测试。