# 官方文档：

<https://docs.docker.com/config/containers/resource_constraints/>

By default, a container has no resource constraints and can use as much of a given resource as the host’s kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the docker run command. This section provides details on when you should set such limits and the possible implications of setting them.

Many of these features require your kernel to support Linux capabilities. To check for support, you can use the [docker info](https://docs.docker.com/engine/reference/commandline/info/)command. If a capability is disabled in your kernel, you may see a warning at the end of the output like the following:

默认情况下,容器没有资源限制，可以使用系统所有资源。docker 通过 docker run 配置容器的内存，cpu, 磁盘io使用量。

其中许多功能都要求您的内核支持Linux功能。 要检查支持，可以使用docker info命令。 如果内核中禁用了某项功能，您可能会在输出结尾处看到警告，如下所示：

WARNING: No swap limit support

<https://docs.docker.com/config/containers/resource_constraints/>

内存限制：

对于Linux 主机，如果没有足够的内容来执行重要的系统任务，将会抛出 OOM 或者 Out of Memory Exception(内存溢出、内存泄漏、内存异常), 随后系统会开始杀死进程以释放内存。每个进程都有可能被 kill，包括Dockerd和其它的应用程序。如果重要的系统进程被Kill,会导致整个系统宕机。

产生 OOM 异常时，Docker尝试通过调整Docker守护程序上的OOM优先级来减轻这些风险，以便它比系统上的其他进程更不可能被杀死。 容器上的OOM优先级未调整，这使得单个容器被杀死的可能性比Docker守护程序或其他系统进程被杀死的可能性更大，不推荐通过在守护程序或容器上手动设置--oom-score-adj为极端负数，或通过在容器上设置--oom-kill-disable来绕过这些安全措施。

## 限制容器对内存的访问

Docker 可以强制执行硬性内存限制，即只允许容器使用给定的内存大小。

Docker 也可以执行非硬性内存限制，即容器可以使用尽可能多的内存，除非内核检测到主机上的内存不够用了。

Most of these options take a positive integer, followed by a suffix of b, k, m, g, to indicate bytes, kilobytes, megabytes, or gigabytes.

内存限制参数：

-m or --memory= :容器可以使用的最大内存量，如果您设置此选项，则允许的最小值为4m （4兆字节）。

--memory-swap \* :容器可以使用的交换分区大小，要在设置物理内存限制的前提才能设置交换分区的限制

--memory-swappiness :设置容器使用交换分区的倾向性，值越高表示越倾向于使用swap分区，范围为0-100，0为能不用就不用，100为能用就用

--kernel-memory ：容器可以使用的最大内核内存量，最小为4m，由于内核内存与用户空间内存隔离，因此无法与用户空间内存直接交换，因此内核内存不足的容器可能会阻塞宿主主机资源，这会对主机和其他容器产生副作用。

--memory-reservation :允许您指定小于--memory的软限制，当Docker检测到主机上的争用或内存不足时会激活该限制，如果使用--memory-reservation，则必须将其设置为低于--memory才能使其优先。 因为它是软限制，所以不能保证容器不超过限制。

--oom-kill-disable：默认情况下，发生OOM时，kernel会杀死容器内进程，但是可以使用--oom-kill-disable参数，可以禁止oom发生在指定的容器上，即 仅在已设置-m / - memory选项的容器上禁用OOM，如果-m 参数未配置，产生OOM时，主机为了释放内存还会杀死系统进程

swap限制：

swap限制参数--memory-swap 只有在设置了 --memory 后才会有意义。使用Swap,可以让容器将超出限制部分的内存置换到磁盘上。WARNING：经常将内存交换到磁盘的应用程序会降低性能

不同的设置会产生不同的效果：

--memory-swap：值为正数， 那么--memory和--memory-swap都必须要设置，--memory-swap表示你能使用的内存和swap分区大小的总和，例如： --memory=300m, --memory-swap=1g, 那么该容器能够使用 300m 内存和 700m swap，即--memory是实际物理内存大小值不变，而实际的计算方式为(--memory-swap)-(--memory)=容器可用swap

--memory-swap：如果设置为0，则忽略该设置，并将该值视为未设置，即未设置交换分区。

--memory-swap：如果等于--memory的值，并且--memory设置为正整数，容器无权访问swap即也没有设置交换分区

--memory-swap：如果设置为unset，如果宿主机开启了swap，则实际容器的swap值为2x( --memory)，即两倍于物理内存大小，但是并不准确。

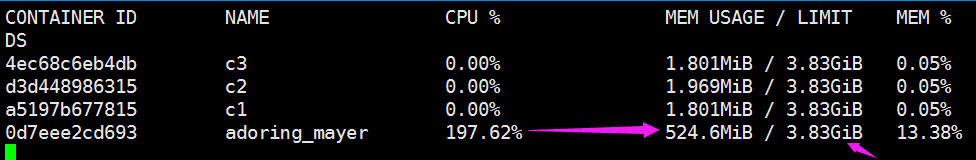
--memory-swap：如果设置为-1，如果宿主机开启了swap，则容器可以使用主机上swap的最大空间。

### 未做内存限制：

**未作限制可以利用到系统内存最大空间**

root@docker-node2:~# docker run -it --rm lorel/docker-stress-ng --vm 2 --vm-bytes 256M

root@docker-node2:~# docker stats



### 容器内存软限制：

#docker run -it --rm --oom-kill-disable --memory 128m --memory-reservation 64m lorel/docker-stress-ng --vm 2 --vm-bytes 256M

# cat /sys/fs/cgroup/memory/docker/f容器ID/memory.soft\_limit\_in\_bytes

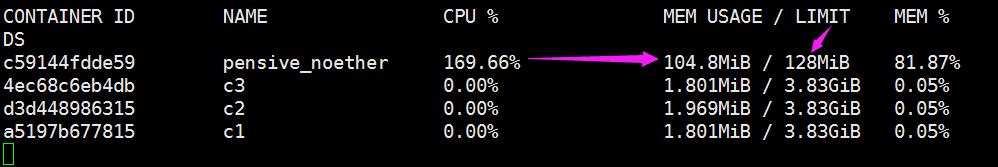
67108864 #返回的限制结果

### 内存硬限制：

root@docker-node2:~# docker run -it --rm -m 128m lorel/docker-stress-ng --vm 2 --vm-bytes 256M

# cat /sys/fs/cgroup/ /docker/容器ID/memory.limit\_in\_bytes

134217728 #返回的限制结果



**注：通过echo命令可以改内存限制的值，但是可以在原基础之上增大内存限制，缩小内存限制会报错write error: Device or resource busy**

### 交换分区限制：

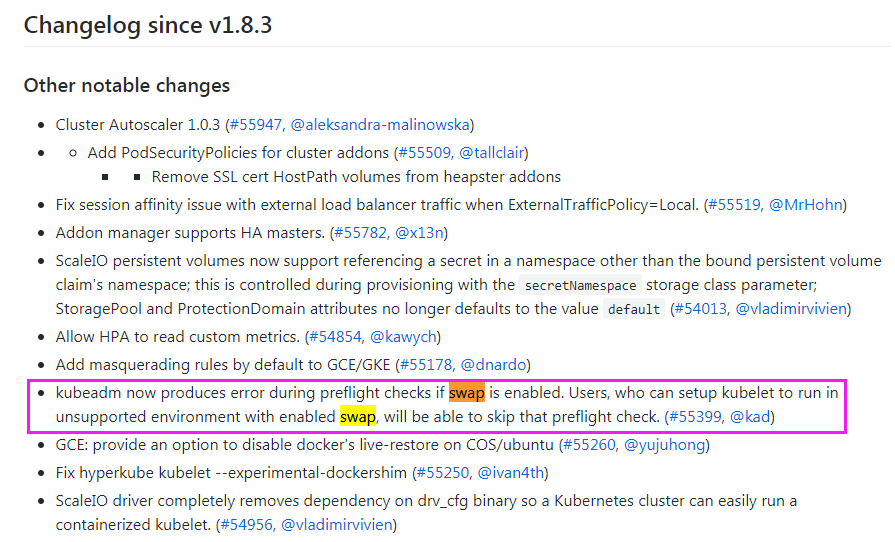
# docker run -it --rm --oom-kill-disable --memory 128m --memory-swap 192m centos bash

# cat /sys/fs/cgroup/memory/docker/容器ID/memory.memsw.limit\_in\_bytes

201326592 #返回值

### K8s 1.8.3更新日志：

<https://github.com/kubernetes/kubernetes/blob/release-1.8/CHANGELOG-1.8.md>



### 关闭一个容器的oom机制：

root@docker-node2:~# docker run -it --rm --oom-kill-disable --memory 128m lorel/docker-stress-ng --vm 2 --vm-bytes 256M

验证：

# cat /sys/fs/cgroup/memory/docker/容器ID/memory.oom\_control

oom\_kill\_disable 1

under\_oom 1

oom\_kill 0

## CPU：

一个宿主机，有几十个核心的CPU，CPU为可压缩资源，但是有成百上千的进程，那么这么多进程怎么执行的？

实时优先级：0-99

非实时优先级(nice)：-20-19，对应100-139的进程优先级

Linux kernel进程的调度基于CFS(Completely Fair Scheduler)，完全公平调度

CPU密集型的场景：优先级越低越好，计算密集型任务的特点是要进行大量的计算，消耗CPU资源，比如计算圆周率、对视频进行高清解码等等，全靠CPU的运算能力

IO密集型的场景：优先级值高点，涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成（因为IO的速度远远低于CPU和内存的速度），比如Web应用，高并发，数据量大的动态网站来说，数据库应该为IO密集型。

#磁盘调度算法

root@docker-node2:~# cat /sys/block/ /queue/scheduler

noop deadline [cfq]

默认情况下，每个容器对主机CPU周期的访问权限是不受限制的，但是我们可以设置各种约束来限制给定容器访问主机的CPU周期，大多数用户使用的是默认的CFS调度方式，在Docker 1.13及更高版本中，还可以配置实时优先级。

参数：

--cpus= 指定容器可以使用多少可用CPU资源。例如，如果主机有两个CPU，并且您设置了--cpus =“1.5”，那么该容器将保证最多可以访问一个半的CPU。这相当于设置--cpu-period =“100000”和--cpu-quota =“150000”。在Docker 1.13和更高版本中可用。

--cpu-period :设置CPU CFS调度程序周期，它与--cpu-quota一起使用，，默认为100微妙，范围从 100ms~1s，即[1000, 1000000]

--cpu-quota :在容器上添加CPU CFS配额，也就是cpu-quota / cpu-period的值，通常使用--cpus设置此值

--cpuset-cpus 主要用于指定容器运行的CPU编号，也就是我们所谓的绑核。

root@docker-node2:~# cat /sys/fs/cgroup/cpuset/docker/4ec68c6eb4db89d4d2907c5ea38941945c75d8431d8960855e026d3b26091641/cpuset.cpus

0-1

--cpuset-mem 设置使用哪个cpu的内存，仅对 非统一内存访问(NUMA)架构有效

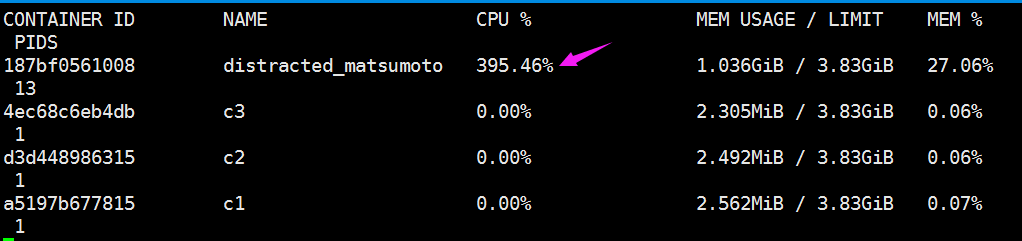
--cpu-shares 主要用于cfs中调度的相对权重，,cpushare值越高的，将会分得更多的时间片，默认的时间片1024，最大262144

## 测试CPU限制

### 未限制容器CPU：

**对于一台四核的服务器，如果不做限制，容器会把宿主机的CPU全部占完：**

root@docker-node2:~# docker run -it --rm lorel/docker-stress-ng --cpu 4 --vm 4

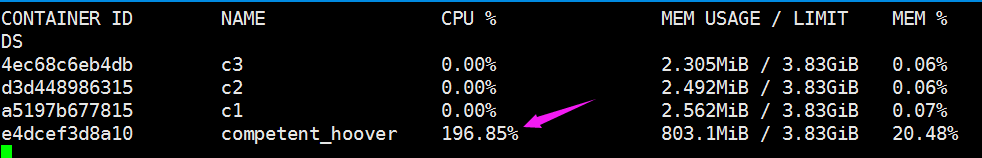


**# cat /sys/fs/cgroup/cpuset/docker/容器ID/cpuset.cpus**

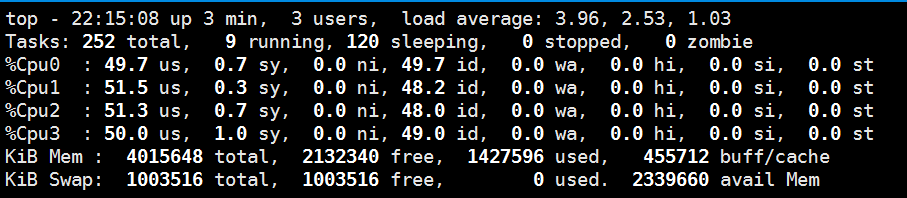
**0-3 #返回值**

### 限制容器CPU：

root@docker-node2:~# docker run -it --rm --cpus 2 lorel/docker-stress-ng --cpu 4 --vm 4



### 宿主机CPU利用率：



将分配给容器的2核心分配到了每一核心宿主机CPU上，也就是进程可以利用任意一核心的宿主机CPU

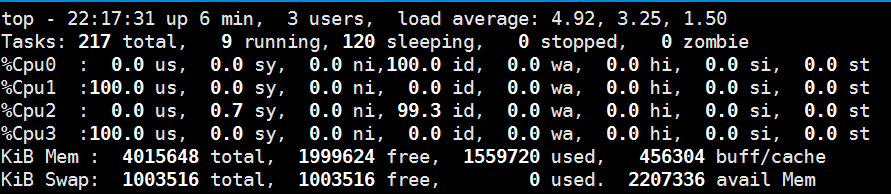
### 将容器运行到指定的CPU上：

# docker run -it --rm --cpus 2 --cpuset-cpus 1,3 lorel/docker-stress-ng --cpu 4 --vm 4

# cat /sys/fs/cgroup/cpuset/docker/容器ID /cpuset.cpus

1,3

### 宿主机CPU利用率：

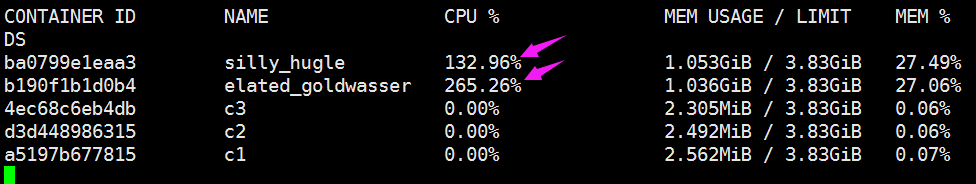


### 基于cpu—shares对CPU进行切分：

**启动两个容器，一个—cpu-shares为1000，一个为500，观察最终效果，基本上shares值500的容器对CPU的利用率是shares值为1000容器的一般：**

root@docker-node2:~# docker run -it --rm --cpu-shares 1000 lorel/docker-stress-ng --cpu 4 --vm 4

root@docker-node2:~# docker run -it --rm --cpu-shares 500 lorel/docker-stress-ng --cpu 4 --vm 4



# cat /sys/fs/cgroup/cpu/docker/容器ID/cpu.shares

1000

# cat /sys/fs/cgroup/cpu/docker/容器PID/cpu.shares

500

**动态修改CPU shares值：**

root@docker-node2:~# echo 1000 > /sys/fs/cgroup/cpu/docker/f容器ID/cpu.shares

root@docker-node2:~# echo 2000 > /sys/fs/cgroup/cpu/docker/f容器ID/cpu.shares

root@docker-node1:~# cat aa.sh

#!/bin/bash

x=0

while [ True ];do

x=$x+1

done;