参考资料

Linux Namespace：https://lwn.net/Articles/531114/
Cgroup: https://access.redhat.com/documentation/zh-cn/red_hat_enterprise_linux/7/html/resource_management_guide/index

RootFs

rootfs 是Docker 容器在启动时内部进程可见的文件系统，即Docker容器的根目录。rootfs通常包含一个操作系统运行所需的文件系统，例如可能包含经典的类Unix操作系统中的目录系统，如/dev、/proc、/bin、/etc、/lib、/usr、/tmp及运行Docker容器所需的配置文件、工具等。

Linux Namespace

Namespace是 Linux 内核用来隔离内核资源的方式。Linux实现了六种不同类型的命名空间。每个命名空间的用途是将特定的全局系统资源包装在抽象中，使命名空间中的进程看起来它们具有自己的全局资源独立实例。命名空间的总体目标之一是支持容器的实现。

Namespace	隔离内容
Mount	文件系统挂载点
IPC	进程间通信资源，即系统VIPC对象和POSIX消息队列
PID	进程ID
Network	网络设备、IP 地址、IP 路由表、/proc/net目录、端口号
UTS	主机名与网络信息服务域名
User	用户和用户组
Cgroup	Cgroup根目录

进程命名空间

lsns 命令说明

列出系统命名空间

```
-p、 --task<pid>打印进程命名空间
```

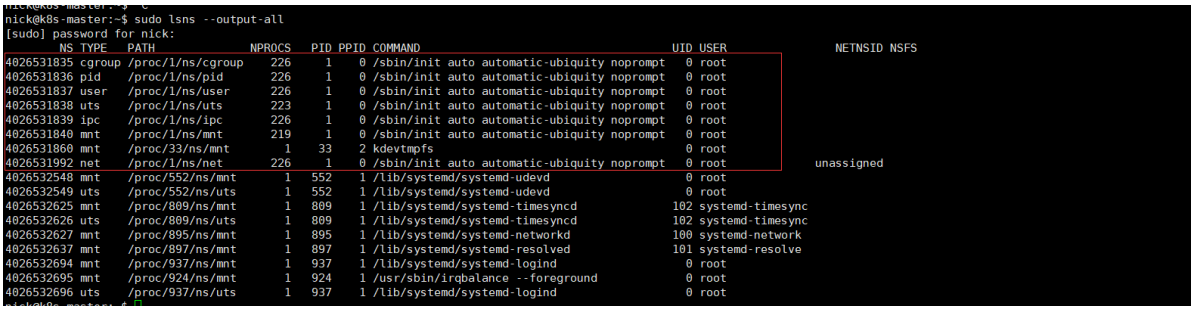
1. NS：命名空间标识符（索引节点号）
2. TYPE：命名空间类型
3. PATH：命名空间的PATH路径
4. NPROCS：命名空间中的进程数
5. PID：命名空间中的最小PID
6. PPID：PID的父级PID
7. COMMAND：PID的命令行
8. UID：PID的UID

9. USER：PID的User
10. NETNSID：网络子系统使用的命名空间ID
11. NSFS：nsfs 文件系统挂载点（通常用于网络子系统）

查看元祖进程命名空间

1. 列出系统所有命名空间

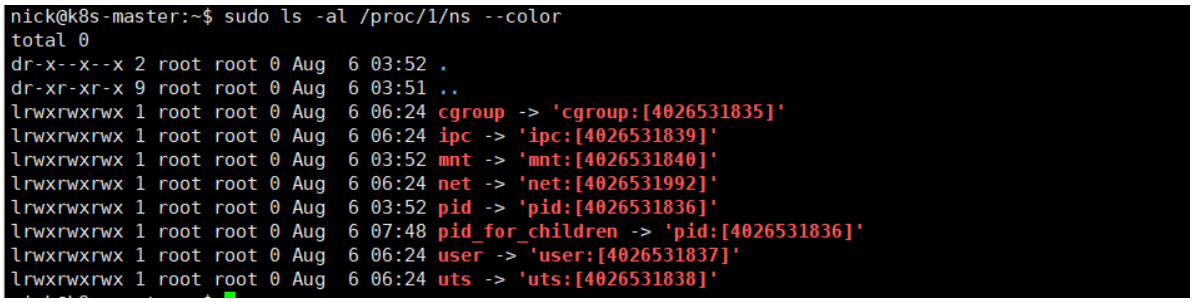
```
sudo lsns --output-all
```



上图红色框内命名空间所属进程ID为1，表示元祖进程的命名空间，即系统默认命名空间。进程没有特殊指定需要创建新的命名空间的情况下，命名空间将与父进程保持一致。

2. 通过文件查看元祖进程命名空间

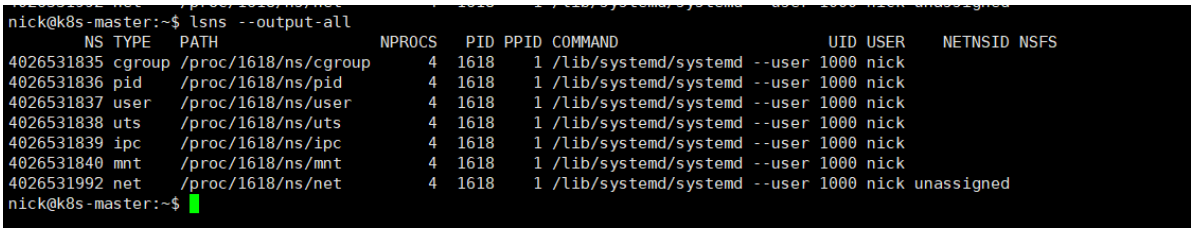
```
sudo ls -al /proc/1/ns --color
```



查看当前用户进程命名空间

1. 查看当前用户进程命名空间列表

```
lsns --output-all
```



2. fork一个新的进程，并且不共享父进程命名空间

```
# 创建新的进程
# 若没有指定-U则需要超级权限
unshare --fork -m -u -i -n -p -U -C  sleep 100
```

```
# 查看所有命名空间
lsns --output-all
```

```
nick@k8s-master:~$ lsns --output-all
        NS TYPE   PATH                   NPROCS   PID  PPID COMMAND                                             UID USER    NETNSID NSFS
4026531835 cgroup /proc/1618/ns/cgroup       5  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531836 pid    /proc/1618/ns/pid          6  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531837 user   /proc/1618/ns/user         5  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531838 uts    /proc/1618/ns/uts          5  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531839 ipc    /proc/1618/ns/ipc          5  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531840 mnt    /proc/1618/ns/mnt          5  1618     1 /lib/systemd/systemd --user                        1000 nick
4026531992 net    /proc/1618/ns/net          5  1618     1 /lib/systemd/systemd --user                        1000 nick unassigned
4026532830 user   /proc/17094/ns/user        2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick
4026532831 mnt    /proc/17094/ns/mnt         2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick
4026532832 uts    /proc/17094/ns/uts         2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick
4026532833 ipc    /proc/17094/ns/ipc         2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick
4026532834 pid    /proc/17095/ns/pid         1 17095 17094 sleep 100                                         1000 nick
4026532835 cgroup /proc/17094/ns/cgroup      2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick
4026532837 net    /proc/17094/ns/net         2 17094 15540 unshare --fork -m -u -i -n -p -U -C sleep 100 1000 nick unassigned
nick@k8s-master:~$
```

新fork出来的进程，在指定新命名空间后，其命名空间字段的值与系统默认命名空间不一致，说明进程创建了新的命名空间。

容器进程命名空间

查看容器进程命名空间列表

1. 运行容器，获取进程ID

```
# 启动nginx 容器
docker run -d --name mynginx nginx
# 获取nginx主进程ID
docker top mynginx
# 查看进程命名空间
sudo lsns -p <pid> --output-all
```

2. 查看容器进程的命名空间情况

```
nick@k8s-master:~/work/helloworld$ sudo lsns -p 7203 --output-all
        NS TYPE   PATH             NPROCS   PID  PPID COMMAND                                  UID USER NETNSID NSFS
4026531835 cgroup /proc/1/ns/cgroup    238     1     0 /sbin/init auto automatic-ubiquity noprompt   0 root
4026531837 user   /proc/1/ns/user      237     1     0 /sbin/init auto automatic-ubiquity noprompt   0 root
4026532640 mnt    /proc/7203/ns/mnt      5  7203  7177 nginx: master process nginx -g daemon off;    0 root
4026532641 uts    /proc/7203/ns/uts      5  7203  7177 nginx: master process nginx -g daemon off;    0 root
4026532642 ipc    /proc/7203/ns/ipc      5  7203  7177 nginx: master process nginx -g daemon off;    0 root
4026532643 pid    /proc/7203/ns/pid      5  7203  7177 nginx: master process nginx -g daemon off;    0 root
4026532645 net    /proc/7203/ns/net      5  7203  7177 nginx: master process nginx -g daemon off;    0 root      0 /run/docker/netns/9e5a1ff6b1e9
nick@k8s-master:~/work/helloworld$
```

nginx容器默认使用了mnt、uts、ipc、pid、net 命名空间隔离，而user与cgroup则继承系统默认命名空间。网络命名空间指定了文件系统挂载点

容器进程命名空间的具体体现

1. 开启docker user命名空间配置，/etc/docker/daemon.json 文件添加以下选项

```
// 默认生成
"userns-remap":"default"
或
// 指定已存在用户和组
"userns-remap":"user:group"
```

2. 重启docker服务

```
sudo systemctl restart docker.service
```

3. 宿主机上查看docker容器默认生成的用户配置

```
cat /etc/subuid

cat /etc/subgid
id <user>
```

/etc/subuid文件：dockremap:165536:65536 表示宿主机使用dockremap用户，容器使用其从属ID，范围从0~65536，与之对应的宿主机ID范围：165536~165536+65536
/etc/subgid文件：针对用户组与/etc/subuid 类似

4. User命名空间：启动新的nginx容器，查看user命名空间

```
# 运行容器，指定私有cgroupns,指定user
docker run -d --cgroupns private --user root  --name mynginx1 nginx

# 查看容器在宿主机上的进程信息，UID显示并不是root
docker top mynginx1

# 与容器交互,查看当前用户信息，显示为root,也可通过id查看用户信息
docker exec -it mynginx1 bash

# 查看进程命名空间，进程拥有独立的命名空间
sudo lsns -p <pid> --output-all
```

```
nick@k8s-master:~/work/helloworld$ sudo lsns --output-all -p 20135
        NS TYPE   PATH                NPROCS   PID  PPID COMMAND                                  UID USER   NETNSID NSFS
4026532640 user   /proc/20135/ns/user       5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
4026532641 mnt    /proc/20135/ns/mnt        5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
4026532642 uts    /proc/20135/ns/uts        5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
4026532643 ipc    /proc/20135/ns/ipc        5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
4026532644 pid    /proc/20135/ns/pid        5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
4026532646 net    /proc/20135/ns/net        5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536       0 /run/docker/netns/f22a4add6f81
4026532707 cgroup /proc/20135/ns/cgroup     5 20135 20112 nginx: master process nginx -g daemon off; 165536 165536
nick@k8s-master:~/work/helloworld$
```

5. UTS命名空间：启动新容器，设置hostname与domain

```
# 运行容器，指定hostname与域名
docker run -d --domainname abc.nick.com  --hostname abcdefg --userns host --name
mynginx2 nginx

# 与容器交互，进入交互模式
docker exec -it mynginx2 bash

# 访问hostname 与 domainname
hostname
domainname

# 通过hostname与domainname访问应用
curl http://abcdefg
curl http://abcdefg.abc.nick.com

# 通过文件查看hostname与domainname
cat /proc/sys/kernel/hostname
cat /proc/sys/kernel/domainname
```

6. mount、PID、Network 命名空间：启动一个工具容器

```
# 运行工具容器
docker run -dit --name mycurl radial/busyboxplus:curl

# 进入交互模式
docker exec -it mycurl sh
```

mount命名空间：容器内部执行mount 与宿主机内执行mount命令对比，即可看出各自拥有不同的mounts。mounts文件位于：/proc/mounts 和 /proc/{PID}/mounts。
mounts文件列说明：

- Device mount的设备
- Mount Point 挂载点，也就是挂载的路径
- File System Type 文件系统类型，如ext4、xfs等
- Options 挂载选项，包括读写权限等参数
- 无用内容，保持内容和/etc/fstab格式一致
- 无用内容，保持内容和/etc/fstab格式一致

PID命名空间：容器内部进程ID为1，宿主机内进程ID不为1

```
[ root@6c845435004d:/ ]$ ps
PID   USER     COMMAND
    1 root     /bin/sh
   53 root     sh
   67 root     ps
[ root@6c845435004d:/ ]$
```

```
nick@k8s-master:~$ docker top mycurl
UID                 PID                 PPID                C                   STIME
165536              23586               23556               0                   06:56
165536              23851               23556               0                   06:57
nick@k8s-master:~$
```

NetWork命名空间：通过ifconfig工具，查看网络信息。容器与宿主机网络完全是两个独立的网络栈

```
[ root@6c845435004d:/ ]$ ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:AC:11:00:04
          inet addr:172.17.0.4  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1076 (1.0 KiB)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

```
nick@k8s-master:~$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:59ff:fe3f:d7b9  prefixlen 64  scopeid 0x20<link>
        ether 02:42:59:3f:d7:b9  txqueuelen 0  (Ethernet)
        RX packets 745  bytes 41677 (41.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 801  bytes 12361082 (12.3 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.239.149  netmask 255.255.255.0  broadcast 192.168.239.255
        inet6 fe80::20c:29ff:feca:35a3  prefixlen 64  scopeid 0x20<link>
        ether 00:0c:29:ca:35:a3  txqueuelen 1000  (Ethernet)
        RX packets 1305795  bytes 1814933519 (1.8 GB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 102417  bytes 12781556 (12.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 3738  bytes 382066 (382.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 3738  bytes 382066 (382.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

cgroups

cgroup全称是control groups，被整合在了linux内核当中，把进程（tasks）放到组里面，对组设置权限，对进程进行控制。可以理解为用户和组的概念，用户会继承它所在组的权限

cpu子系统：

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：

1. 完全公平调度程序（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）
2. 实时调度程序（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

CFS

1. cpu.cfs_period_us：此参数可以设定重新分配 cgroup 可用 CPU 资源的时间间隔，单位为微秒，上限1秒，下限1000微秒。即设置单个CPU重新分配周期
2. cpu.cfs_quota_us：此参数可以设定在某一阶段（由 cpu.cfs_period_us 规定）某个 cgroup 中所有任务可运行的时间总量，单位为微秒。即每个周期时间内，可以使用多长时间的CPU（单个），该值可以大于cfs_period_us的值，表示可以利用多个CPU来满足CPU使用时长
3. cpu.shares：用一个整数来设定cgroup中任务CPU可用时间的相对比例。该参数是对系统所有CPU做分配，不是单个CPU。
4. cpu.stat：报告 CPU 时间统计，

nr_periods： 经过的周期间隔数
nr_throttled： cgroup 中任务被节流的次数（即耗尽所有按配额分得的可用时间后，被禁止运行）
throttled_time： cgroup 中任务被节流的时间总计（以纳秒为单位）

RT

RT 调度程序与 CFS 类似，但只限制实时任务对 CPU 的存取。

1. 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），只可用于实时调度任务
2. 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

示例

1. 一个 cgroup 使用一个 CPU 的 25%，同时另一个 cgroup 使用此 CPU 的 75%

```
echo 250 > /cgroup/cpu/blue/cpu.shares
echo 750 > /cgroup/cpu/red/cpu.shares
```

2. 一个 cgroup 完全使用一个 CPU

```
echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
echo 10000 > /cgroup/cpu/red/cpu.cfs_period_us
```

3. 一个 cgroup 使用 CPU 的 10%

```
echo 10000 > /cgroup/cpu/red/cpu.cfs_quota_us
echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

4. 多核系统中，如要让一个 cgroup 完全使用两个 CPU 核

```
echo 200000 > /cgroup/cpu/red/cpu.cfs_quota_us
echo 100000 > /cgroup/cpu/red/cpu.cfs_period_us
```

cpuset子系统：

可以为 cgroup 分配独立 CPU 和内存节点

1. cpuset.cpu_exclusive：包含标签（0 或者 1），它可以指定：其它 cpuset 及其父、子 cpuset 是否可共享该 cpuset 的特定 CPU。默认情况下（0），CPU 不会专门分配给某个 cpuset
2. cpuset.cpus（强制）：设定该 cgroup 任务可以访问的 CPU。这是一个逗号分隔列表，格式为 ASCII，小横线（"-"）代表范围。例如：0-2,16 表示cpu 0、1、2 和 16
3. cpuset.mem_exclusive：包含标签（0 或者 1），它可以指定：其它 cpuset 是否可共享该 cpuset 的特定内存节点。默认情况下（0），内存节点不会专门分配给某个 cpuset 。为某个 cpuset 保留其专用内存节点（1）与使用 cpuset.mem_hardwall 参数启用内存 hardwall 功能是一样的
4. cpuset.mem_hardwall：包含标签（0 或者 1），它可以指定：内存页和缓冲数据的 kernel 分配是否受到 cpuset 特定内存节点的限制。默认情况下 0，页面和缓冲数据在多用户进程间共享。启用 hardwall 时（1）每个任务的用户分配可以保持独立
5. cpuset.memory_migrate：包含一个标签（0 或者 1），用来指定当 cpuset.mems 的值更改时，是否应该将内存中的页迁移到新节点。默认情况下禁止内存迁移（0）且页就保留在原来分配的节点中，即使此节点不再是 cpuset.mems 指定的节点。如果启用（1），系统会将页迁移到 cpuset.mems 指定的新参数的内存节点中，如果可能的话会保留其相对位置。
6. cpuset.memory_pressure：一份只读文件，包含该 cpuset 进程生成的“内存压力”运行平均。启用 cpuset.memory_pressure_enabled 时，该伪文件中的值会自动更新，除非伪文件包含 0 值。
7. cpuset.memory_pressure_enabled：包含标签（0 或者 1），它可以设定系统是否计算该 cgroup 进程生成的“内存压力”。计算出的值会输出到 cpuset.memory_pressure，代表进程试图释放被占用内存的速率，报告值为：每秒尝试回收内存的整数值再乘以 1000。
8. cpuset.memory_spread_page：包含标签（0 或者 1），它可以设定文件系统缓冲是否应在该 cpuset 的内存节点中均匀分布。默认情况下 0，系统不会为这些缓冲平均分配内存页面，缓冲被置于生成缓冲的进程所运行的同一节点中。
9. cpuset.memory_spread_slab：包含标签（0 或者 1），它可以设定是否在 cpuset 间平均分配用于文件输入 / 输出操作的 kernel 高速缓存板。默认情况下 0，kernel 高速缓存板不被平均分配，

高速缓存板被置于生成它们的进程所运行的同一节点中。

10. cpuset.mems（强制）：设定该 cgroup 中任务可以访问的内存节点。这是一个逗号分隔列表，格式为 ASCII，小横线（"-"）代表范围。例如：0-2,16 表示内存节点 0、1、2 和 16。内存节点：内存被划分为节点，每一个节点关联到一个cpu
11. cpuset.sched_load_balance：包含标签（0 或者 1），它可以设定 kernel 是否在该 cpuset 的 CPU 中平衡负载。默认情况下 1，kernel 将超载 CPU 中的进程移动到负载较低的 CPU 中以便平衡负载。如果父cgroup设置了，子cgroup的设置将没有任何作用
12. cpuset.sched_relax_domain_level：包含 -1 到一个小正数间的整数，它代表 kernel 应尝试平衡负载的 CPU 宽度范围。如果禁用 cpuset.sched_load_balance，则该值无意义

值	效果
-1	平衡负载的系统默认值
0	不执行直接负载平衡；负载平衡只是阶段性的
1	对同一核中的线程进行直接负载平衡
2	对同一软件包中的线程进行直接负载平衡
3	对同一节点或者扇叶中的线程进行直接负载平衡
4	对不使用统一内存访问（NUMA）构架中的多个 CPU 进行直接负载平衡
5	对使用统一内存访问（NUMA）构架中的多个 CPU 进行直接负载平衡

cpuacct子系统

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

1. 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

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

2. 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）
3. 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）

memory子系统：

自动生成 cgroup 任务使用内存资源的报告，并限定这些任务所用内存的大小

1. memory.failcnt：报告内存达到 memory.limit_in_bytes 设定的限制值的次数
2. memory.force_empty：当设定为 0 时，该 cgroup 中任务所用的所有页面内存都将被清空。这个接口只可在 cgroup 没有任务时使用。如果无法清空内存，请在可能的情况下将其移动到父 cgroup 中。移除 cgroup 前请使用 memory.force_empty 参数以免将废弃的页面缓存移动到它的父 cgroup 中
3. memory.limit_in_bytes：设定用户内存（包括文件缓存）的最大用量。如果没有指定单位，则该数值将被解读为字节。但是可以使用后缀代表更大的单位 —— k 或者 K 代表千字节，m 或者 M 代表兆字节 ， g 或者 G 代表千兆字节。在 memory.limit_in_bytes 中写入 -1 可以移除全部已有限制。
4. memory.max_usage_in_bytes：报告 cgroup 中进程所用的最大内存量（以字节为单位）

5. move_charge_at_immigrate：当将一个task移动到另一个cgroup中时，此task的内存页可能会被重新统计到新的cgroup中，这取决于是否设置了move_charge_at_immigrate
6. numa_stat： 每个numa节点的内存使用数量
7. memory.oom_control：设置or查看内存超限控制信息(OOM killer)
8. memory.pressure_level：设置内存压力通知
9. memory.soft_limit_in_bytes：内存软限制
10. memory.stat：报告大范围内存统计
11. memory.swappiness：将 kernel 倾向设定为换出该 cgroup 中任务所使用的进程内存，而不是从页高速缓冲中再生页面。
12. memory.usage_in_bytes：报告 cgroup 中进程当前所用的内存总量（以字节为单位）
13. memory.use_hierarchy：包含标签（0 或者 1），它可以设定是否将内存用量计入 cgroup 层级的吞吐量中。如果启用（1），内存子系统会从超过其内存限制的子进程中再生内存。默认情况下（0），子系统不从任务的子进程中再生内存

内核内存：专用于Linux内核系统服务使用，是不可swap的

14. memory.kmem.failcnt：报告内核内存达到 memory.kmem.limit_in_bytes 设定的限制值的次数
15. memory.kmem.limit_in_bytes：设定内核内存（包括文件缓存）的最大用量。如果没有指定单位，则该数值将被解读为字节
16. memory.kmem.max_usage_in_bytes：报告 cgroup 中进程所用的最大内核内存量（以字节为单位）
17. memory.kmem.slabinfo：查看内核内存分配情况
18. memory.kmem.usage_in_bytes：报告 cgroup 中进程当前所用的内核内存总量（以字节为单位）
19. memory.kmem.tcp.failcnt：报告tcp缓存内存达到memory.kmem.tcp.limit_in_bytes设定限制值的次数
20. memory.kmem.tcp.limit_in_bytes：设置或查看TCP缓冲区的内存使用限制
21. memory.kmem.tcp.max_usage_in_bytes：报告cgroup中进程所用的最大tcp缓存内存量
22. memory.kmem.tcp.usage_in_bytes：报告cgroup中进程当前所用TCP缓冲区的内存使用量

示例

1. cgroup 中任务可用的内存量设定为 100MB

```
echo 104857600 > memory.limit_in_bytes
```

blkio子系统

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。

1. blkio.reset_stats：此参数用于重设其它伪文件记录的统计数据。请在此文件中写入整数来为 cgroup 重设统计数据
2. blkio.throttle.io_service_bytes：此参数用于报告 cgroup 传送到具体设备或者由具体设备中传送出的字节数。
3. blkio.throttle.io_serviced：此参数用于报告 cgroup 根据节流方式在具体设备中执行的 I/O 操作数。
4. blkio.throttle.read_bps_device：此参数用于设定设备执行“读”操作字节的上限。“读”的操作率以每秒的字节数来限定。
5. blkio.throttle.read_iops_device：此参数用于设定设备执行“读”操作次数的上限。“读”的操作率以每秒的操作次数来表示。
6. blkio.throttle.write_bps_device：此参数用于设定设备执行“写”操作次数的上限。“写”的操作率用“字节/秒”来表示

7. blkio.throttle.write_iops_device：此参数用于设定设备执行 “写” 操作次数的上限。“写”的操作率以每秒的操作次数来表示。

devices子系统

允许或者拒绝 cgroup 任务存取设备

1. devices.allow：指定 cgroup 任务可访问的设备
2. devices.deny：指定 cgroup 任务无权访问的设备
3. devices.list：报告 cgroup 任务对其访问受限的设备

freezer子系统

暂停或者恢复 cgroup 中的任务

1. freezer.state：

FROZEN： cgroup 中的任务已被暂停
FREEZING：系统正在暂停 cgroup 中的任务
THAWED： cgroup 中的任务已恢复

net_cls子系统

使用等级识别符（classid）标记网络数据包，这让 Linux 流量管控器（tc）可以识别从特定 cgroup 中生成的数据包。可配置流量管控器，让其为不同 cgroup 中的数据包设定不同的优先级

1. net_cls.classid： 包含表示流量控制 handle 的单一数值。从 net_cls.classid 文件中读取的 classid 值是十进制格式，但写入该文件的值则为十六进制格式

net_prio子系统

可以为各个 cgroup 中的应用程序动态配置每个网络接口的流量优先级。网络优先级是一个分配给网络流量的数值，可在系统内部和网络设备间使用。网络优先级用来区分发送、排队以及丢失的数据包

1. net_prio.prioidx：只读文件。它包含一个特有整数值，kernel 使用该整数值作为这个 cgroup 的内部代表。
2. net_prio.ifpriomap：包含优先级图谱，这些优先级被分配给源于此群组进程的流量以及通过不同接口离开系统的流量

perf_event

允许使用perf工具来监控cgroup

hugetlb

允许使用大篇幅的虚拟内存页，并且给这些内存页强制设定可用资源量