docker资源限制及应用总结

原创 Mr大表哥 2017-03-20 14:34:30 评论(0) 229人阅读

博主QQ：819594300
博客地址：http://zpf666.blog.51cto.com/
有什么疑问的朋友可以联系博主，博主会帮你们解答，谢谢支持！

下面介绍cgroup如何做到内存，cpu和io速率的隔离

一、CPU资源控制

CPU资源的控制也有两种策略：
一种是完全公平调度（CFS：Completely Fair Scheduler）策略，提供了限额和按比例分配两种方式进行资源控制；
另一种是实时调度（RTS：Real-Time Scheduler）策略，针对实时进程按周期分配固定的运行时间。配置时间都以微秒（μs）为单位，文件名中用us表示

1、CFS调度策略下的配置

(1)按权重比例设定CPU的分配
docker提供了--cpu-shares参数，在创建容器时指定容器所使用的CPU份额值。

```
[root@localhost ~]# docker run -dit --cpu-shares 100 docker.io/centos:latest
c6ed167e44a5ee6f88d333fbba69c76daa93bb70d7b32c3ddc5c1c8d4f98810a
[root@localhost ~]# cat /sys/fs/cgroup/cpu/system.slice/docker-c6ed167e44a5ee6f88d333fbba69c76daa93bb70d7b32c3ddc5c1c8d4f98810a.scope/cpu.shares
100
[root@localhost ~]#
```

以上cat 后面的路径是docker1.10版本的路径，如果安装的是docker1.12版本的，路径是"/sys/fs/cgroup/cpu/docker/容器完整ID/cpu.shares"

cpu-shares的值不能保证可以获得1个vcpu或者多少GHz的CPU资源，仅仅只是一个加权值。
该加权值是一个整数（必须大于等于2）表示相对权重，最后除以权重总和算出相对比例，按比例分配CPU时间。
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。如果容器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。

cpu-shares演示案例：

1）先删除docker主机上运行的容器，然后docker通过--cpu-shares指定CPU份额

运行一个容器指定cpu份额为1024。

```
[root@localhost ~]# grep processor /proc/cpuinfo
processor       : 0
processor       : 1
```

这个命令看一下有cpu有几个内核

```
[root@localhost ~]# docker rm -f $(docker ps -aq)
c6ed167e44a5
[root@localhost ~]# docker run -it --rm --name=benet1 --cpu-shares 1024 --cpuset-cpus 0 docker.io/jess/stress:latest --cpu 2
stress: info: [1] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
```

上面那条命令的意思是创建一个容器叫benet1，如果退出即终止删除，--cpu-shares 1024的意思是指定的cpu的份额是1024（其实默认就是1024），--cpuset-cpus 0的意思是把该容器绑定给cpu的第一个内核（第一个内核标号是0，第二个内核的标号是1），后面使用的是stress镜像），最后的--cpu 2的意思是（--cpu是stress命令的选项）表示产生2个进程每个进程都反复不停的计算随机数的平方根，stress命令是linux下的一个压力测试工具。

注：

--cpu-shares 指定 CPU 份额，默认就是 1024

--cpuset-cpus 可以绑定 CPU。例如，指定容器在--cpuset-cpus 0,1 或--cpuset-cpus 0-3

--cpu 是 stress 命令的选项表示产生 n 个进程每个进程都反复不停的计算随机数的平方根

stress 命令是 linux 下的一个压力测试工具。

2）在docker宿主机上打开一个伪终端执行top命令

```
top - 18:25:16 up 49 min,  3 users,  load average: 2.00, 1.82, 1.13
Tasks: 452 total,   5 running, 447 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.9 us,  0.5 sy,  0.0 ni, 48.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,   248152 free,   628328 used,   992208 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.   995548 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13869 root      20   0    7164     88      0 R  50.3  0.0   5:48.60 stress
13868 root      20   0    7164     88      0 R  50.0  0.0   5:48.60 stress
12253 root      20   0 1751264 231180  48028 S   1.3 12.4   0:34.91 gnome-shell
 2624 root      20   0  245372  42196   9084 S   0.3  2.3   0:11.26 Xorg
14038 root      20   0  146412   2384   1424 R   0.3  0.1   0:00.10 top
```

3）然后再启动一个容器，--cpu-shares设为512

```
[root@localhost ~]# docker run -it --rm --name=benet2 --cpu-shares 512 --cpuset-cpus 0 docker.io/jess/stress:latest --cpu 2
stress: info: [1] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
```

4）查看top的现实结果

```
top - 19:08:28 up  1:32,  4 users,  load average: 3.88, 3.96, 3.81
Tasks: 465 total,   7 running, 458 sleeping,   0 stopped,   0 zombie
%Cpu(s): 52.6 us,  0.3 sy,  0.0 ni, 47.1 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,    90764 free,   703380 used,  1074544 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.   861976 avail Mem

12253 root      20   0 1761976 242192  48032 S   0.3 13.0   0:45.72 gnome-shell
13868 root      20   0    7164     88      0 R  33.6  0.0  20:47.85 stress
13869 root      20   0    7164     88      0 R  33.2  0.0  20:47.85 stress
51015 root      20   0    7164     88      0 R  16.6  0.0   0:08.86 stress
51016 root      20   0    7164     88      0 R  16.6  0.0   0:08.86 stress
```

可以看到container1的CPU占比为1024/(1024+512)=2/3，container2的CPU占比为512/(1024+512)=1/3

5）将container2的cpu.shares改为1024，

#echo “1024”>/sys/fs/cgroup/cpu/system.slice/docker-<容器的完整长ID>/cpu.shares

```
[root@localhost ~]# docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS
  PORTS               NAMES
86e27c8684a0        docker.io/jess/stress:latest   "stress --cpu 2"    4 minutes ago       Up 4 minutes
                      benet2
87da9697b7d8        docker.io/jess/stress:latest   "stress --cpu 2"    58 minutes ago      Up 58 minutes
                      benet1
[root@localhost ~]# echo "1024" > /sys/fs/cgroup/cpu/system.slice/docker-86e27c8684a0c40c623c2b0eb3cd7f27f609b8684f643670102930b902baba64.scope/cpu.shares
[root@localhost ~]#
```

6）再来查看top的现实结果

```
top - 19:15:38 up  1:39,  5 users,  load average: 4.05, 4.03, 3.90
Tasks: 463 total,   7 running, 456 sleeping,   0 stopped,   0 zombie
%Cpu(s): 50.8 us,  0.2 sy,  0.0 ni, 49.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,    77312 free,   715572 used,  1075804 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.   849384 avail Mem

 12253 root      20   0 1761976 242192  48032 S   0.3 13.0   0:45.72 gnome-shell
 13868 root      20   0    7164     88      0 R  24.9  0.0  22:56.19 stress
 13869 root      20   0    7164     88      0 R  24.9  0.0  22:56.18 stress
 51015 root      20   0    7164     88      0 R  24.9  0.0   1:35.45 stress
 51016 root      20   0    7164     88      0 R  24.9  0.0   1:35.44 stress
```

（2）设定CPU使用周期使用时间上限

groups 里，可以用 cpu.cfs_period_us和 cpu.cfs_quota_us 来限制该组中的所有进程在单位时间里可以使用的 cpu 时间。 cpu.cfs_period_us就是时间周期，默认为 100000，即百毫秒。 cpu.cfs_quota_us 就是在这期间内可使用的 cpu 时间，默认 -1，即无限制。

cpu.cfs_period_us：设定时间周期（单位为微秒（μs）），必须与cfs_quota_us配合使用。

cpu.cfs_quota_us：设定周期内最多可使用的时间（单位为微秒（μs））。这里的配置指task对单个cpu的使用上限。

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，若cfs_quota_us是cfs_period_us的两倍，就表示在两个核上完全使用CPU，例如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）。

使用示例：

1）使用命令docker run创建容器

```
[root@localhost ~]# docker run --rm -it --cpu-period 100000 --cpu-quota 200000 --cpuset-cpus 0,1 docker.io/jess/stress:latest --cpu 2
stress: info: [1] dispatching hogs: 2 cpu, 0 io, 0 vm, 0 hdd
```

2）在宿主机上执行top，从下图可以看到基本占了100%的cpu资源

```
top - 19:33:08 up  1:57,  5 users,  load average: 2.18, 1.50, 2.40
Tasks: 453 total,   4 running, 449 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.2 sy,  0.0 ni,  0.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,    99108 free,   721088 used,  1048492 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.   846064 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
51599 root      20   0    7164     92      0 R  98.7  0.0   2:55.50 stress
51600 root      20   0    7164     92      0 R  98.3  0.0   2:55.48 stress
```

3）则最终生成的cgroup的cpu周期配置可以下面的目录中找到：/sys/fs/cgroup/cpu/system.slice/docker-<容器的完整长ID>/

```
[root@localhost ~]# cat /sys/fs/cgroup/cpu/system.slice/docker-a92ae30525c811335735487644b2e7211ff7e9a4c493ef12ba2af831e4c78540.scope/cpu.cfs_period_us
100000
[root@localhost ~]# cat /sys/fs/cgroup/cpu/system.slice/docker-a92ae30525c811335735487644b2e7211ff7e9a4c493ef12ba2af831e4c78540.scope/cpu.cfs_quota_us
200000
[root@localhost ~]#
```


4）修改容器的cpu.cfs_period_us和 cpu.cfs_quota_us值

```
[root@localhost ~]# echo "1000000" > /sys/fs/cgroup/cpu/system.slice/docker-a92ae30525c811335735487644b2e7211ff7e9a4c493ef12ba2af831e4c78540.scope/cpu.cfs_period_us
[root@localhost ~]# echo "500000" > /sys/fs/cgroup/cpu/system.slice/docker-a92ae30525c811335735487644b2e7211ff7e9a4c493ef12ba2af831e4c78540.scope/cpu.cfs_quota_us
[root@localhost ~]#
```


5）再执行top查看cpu资源，从下图可以看到基本占了50%的cpu资源。

```
top - 19:45:05 up  2:09,  5 users,  load average: 0.51, 1.77, 2.24
Tasks: 452 total,   4 running, 448 sleeping,   0 stopped,   0 zombie
%Cpu(s): 26.4 us,  0.3 sy,  0.0 ni, 73.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,    96144 free,   723868 used,  1048676 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.   843084 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
51599 root      20   0    7164     92      0 R  24.9  0.0  13:06.54 stress
51600 root      20   0    7164     92      0 R  24.9  0.0  13:07.92 stress
```


2、RT调度策略下的配置

实时调度策略与公平调度策略中的按周期分配时间的方法类似，也是在周期内分配一个固定的运行时间。

cpu.rt_period_us ：设定周期时间。

cpu.rt_runtime_us：设定周期中的运行时间

```
[root@localhost ~]# ls -l /sys/fs/cgroup/cpu/system.slice/docker-a9679e330ec37e6d7c1c06e528740a756bf3e39eecb8e62f929d0802e3fd966e.scope/
总用量 0
-rw-r--r--. 1 root root 0 2月  23 19:48 cgroup.clone_children
--w--w--w-. 1 root root 0 2月  23 19:48 cgroup.event_control
-rw-r--r--. 1 root root 0 2月  23 19:48 cgroup.procs
-r--r--r--. 1 root root 0 2月  23 19:48 cpuacct.stat
-rw-r--r--. 1 root root 0 2月  23 19:48 cpuacct.usage
-r--r--r--. 1 root root 0 2月  23 19:48 cpuacct.usage_percpu
-rw-r--r--. 1 root root 0 2月  23 19:48 cpu.cfs_period_us
-rw-r--r--. 1 root root 0 2月  23 19:48 cpu.cfs_quota_us
-rw-r--r--. 1 root root 0 2月  23 19:48 cpu.rt_period_us
-rw-r--r--. 1 root root 0 2月  23 19:48 cpu.rt_runtime_us
-rw-r--r--. 1 root root 0 2月  23 19:48 cpu.shares
-r--r--r--. 1 root root 0 2月  23 19:48 cpu.stat
-rw-r--r--. 1 root root 0 2月  23 19:48 notify_on_release
-rw-r--r--. 1 root root 0 2月  23 19:48 tasks
[root@localhost ~]#
```


cpuset - CPU绑定

对多核CPU的服务器，docker还可以控制容器运行限定使用哪些cpu内核和内存节点，即使用 - cpuset-cpus和 - cpuset-mems参数。对具有NUMA拓扑（具有多CPU、多内存节点）的服务器尤其有用，可以对需要高性能计算的容器进行性能最优的配置。如果服务器只有一个内存节点，则 - cpuset-mems的配置基本上不会有明显效果

注：

现在的机器上都是有多个CPU和多个内存块的。以前我们都是将内存块看成是一大块内存，所有CPU到这个共享内存的访问消息是一样的。但是随着处理器的增加，共享内存可能会导致内存访问冲突越来越厉害，且如果内存访问达到瓶颈的时候，性能就不能随之增加。NUMA（Non-Uniform Memory Access）就是这样的环境下引入的一个模型。比如一台机器是有2个处理器，有4个内存块。我们将1个处理器和两个内存块合起来，称为

一个NUMA node，这样这个机器就会有两个NUMA node。在物理分布上，NUMA node的处理器和内存块的物理距离更小，因此访问也更快。比如这台机器会分左右两个处理器（cpu1，cpu2），在每个处理器两边放两个内存块(memory1.1,memory1.2，memory2.1,memory2.2)，这样NUMA node1的cpu1访问memory1.1和memory1.2就比访问memory2.1和memory2.2更快。所以使用NUMA的模式如果能尽量保证本node内的CPU只访问本node内的内存块，那这样的效率就是最高的。

使用示例：

```
[root@localhost ~]# grep processor /proc/cpuinfo
processor       : 0
processor       : 1
processor       : 2
processor       : 3
[root@localhost ~]#
```



```
[root@localhost ~]# docker run -dit --name=test1 --cpuset-cpus 0-2 docker.io/centos:latest
63270f4aa78bf01f5dc5dbd6dd57884ecfd8aae957a6a2217a995f8ee077f820
[root@localhost ~]# docker exec test1 taskset -c -p 1
pid 1's current affinity list: 0-2
[root@localhost ~]#
```

--cpuset-cpus 0-2:表示创建的容器只能用0、1、2这三个内核

例子：cpuset.cpus：在这个文件中填写cgroup可使用的CPU编号，如0-2,16代表 0、1、2和16这4个CPU。

docker exec <容器ID或容器名> taskset -c -p 1(容器内部第一个进程编号一般为1)，可以看到容器中进程与CPU内核的绑定关系，可以认为达到了绑定CPU内核的目的。

最终生成的cgroup的cpu内核配置如下：

```
[root@localhost ~]# cat /sys/fs/cgroup/cpuset/system.slice/docker-63270f4aa78bf01f5dc5dbd6dd57884ecfd8aae957a6a2217a995f8ee077f820.scope/cpuset.cpus
0-2
[root@localhost ~]#
```


cpuset.cpus：在这个文件中填写cgroup可使用的CPU编号，如0-2,16代表 0、1、2和16这4个CPU。

cpuset.mems：与CPU类似，表示cgroup可使用的memorynode，格式同上

通过docker exec <容器ID>taskset -c -p 1(容器内部第一个进程编号一般为1)，可以看到容器中进程与CPU内核的绑定关系，可以认为达到了绑定CPU内核的目的。

总结：

CPU配额控制参数的混合使用

当上面这些参数中时，cpu-shares控制只发生在容器竞争同一个内核的时间片时，如果通过cpuset-cpus指定容器A使用内核0，容器B只是用内核1，在主机上只有这两个容器使用对应内核的情况，它们各自占用全部的内核资源，cpu-shares没有明显效果。

cpu-period、cpu-quota这两个参数一般联合使用，在单核情况或者通过cpuset-cpus强制容器使用一个cpu内核的情况下，即使cpu-quota超过cpu-period，也不会使容器使用更多的CPU资源。

cpuset-cpus、cpuset-mems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。

二、内存配额控制

和CPU控制一样，docker也提供了若干参数来控制容器的内存使用配额，可以控制容器的swap大小、可用内存大小等各种内存方面的控制。主要有以下参数：

Docker提供参数“-m,--memory=”限制容器的内存使用量，如果不设置-m，则默认容器内存是不设限的，容器可以使用主机上的所有空闲内存

内存配额控制使用示例

设置容器的内存上限，参考命令如下所示

#docker run -dit --memory128m 镜像

默认情况下，除了 - memory指定的内存大小以外，docker还为容器分配了同样大小的swap分区，也就是说，上面的命令创建出的容器实际上最多可以使用256MB内存，而不是128MB内存。如果需要自定义swap分区大小，则可以通过联合使用 - memory - swap参数来实现控制。

1）用256M内存空间进行压力测试

```
[root@localhost ~]# docker run -dit --name=test1 --memory 128M docker.io/jess/stress:latest --vm 1 --vm-bytes
 256M --vm-hang 0
371d91b9d33664309b71ef3164801a53215eeb456f10cc24c88419a264f33944
```

--memory 128M:设置内存配额为128，其实实际的配额是128*2=256M，因为创建内存配额的时候，默认创建一个同配额大小的swap分区，所以是256M，现在我们在这个容器里面创建一个进程，进程所使用的内存空间设置为256M，并且设置为进程不休眠，即没有休眠限制，如果设置的是一个实际的数字，则默认是单位是秒，再达到这个秒数后，进入休眠状态。

2）查看容器状态

```
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE                                 COMMAND                  CREATED             STATUS
                    PORTS               NAMES
371d91b9d336        docker.io/jess/stress:latest          "stress --vm 1 --vm-b"   About a minute ago  Exited (1) A
bout a minute ago                       test1
[root@localhost ~]#
```

可以发现，使用256MB进行压力测试时，由于超过或等于了内存上限（128MB内存+128MB swap），进程被OOM(out of memory)杀死。所以容器是终止状态。

3）再使用250MB进行压力测试

```
[root@localhost ~]# docker run -dit --name=test2 --memory 128M docker.io/jess/stress:latest --vm 1 --vm-bytes
 250M --vm-hang 0
a951314e85249713cfa04eda127f9928e60a07682e4e4870b34b5c97194f0071
[root@localhost ~]#
```

4)再次查看容器，进程可以正常运行。

```
[root@localhost ~]# docker ps -a
CONTAINER ID        IMAGE                                 COMMAND                  CREATED             STATUS
              PORTS               NAMES
a951314e8524        docker.io/jess/stress:latest          "stress --vm 1 --vm-b"   About a minute ago  Up About a m
inute                           test2
371d91b9d336        docker.io/jess/stress:latest          "stress --vm 1 --vm-b"   10 minutes ago      Exited (1) 1
0 minutes ago                   test1
[root@localhost ~]#
```

5）通过docker stats可以查看到容器的内存使用情况

```
[root@localhost ~]# docker stats
```

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I
/O
a951314e8524        0.00%               92.78 MB / 134.2 MB   69.13%              648 B / 648 B       6.584 M
B / 352.6 MB
```

6）对上面的命令创建的容器，可以查看到在cgroups的配置文件中，查看到容器的内存大小为128MB（128×1024×1024=134217728B），内存和swap加起来大小为256MB（256×1024×1024=268435456B）。

```
[root@localhost ~]# cat /sys/fs/cgroup/memory/system.slice/docker-a951314e85249713cfa04eda127f9928e60a07682e4e4870b34b5c97194f0071.scope/memory.limit_in_bytes
134217728
[root@localhost ~]# cat /sys/fs/cgroup/memory/system.slice/docker-a951314e85249713cfa04eda127f9928e60a07682e4e4870b34b5c97194f0071.scope/memory.memsw.limit_in_bytes
268435456
[root@localhost ~]#
```


三、磁盘IO配额控制

主要包括以下参数：

--device-read-bps：限制此设备上的读速度（bytes per second），单位可以是kb、mb或者gb。--device-read-iops：通过每秒读IO次数来限制指定设备的读速度。

--device-write-bps：限制此设备上的写速度（bytes per second），单位可以是kb、mb或者gb。

--device-write-iops：通过每秒写IO次数来限制指定设备的写速度。

--blkio-weight：容器默认磁盘IO的加权值，有效值范围为10-1000。

--blkio-weight-device：针对特定设备的IO加权控制。其格式为DEVICE_NAME:WEIGHT

磁盘IO配额控制示例：

blkio-weight

使用下面的命令创建两个 - blkio-weight值不同的容器：

在容器中同时执行下面的dd命令，进行测试

```
[root@localhost ~]# docker run -it --rm --name testa --blkio-weight 100 docker.benet.com/centos:centos7
[root@8ef8ee0c16bc /]# time dd if=/dev/zero of=test.out bs=1M count=1024 oflag=direct
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 12.9098 s, 83.2 MB/s

real    0m13.117s
user    0m0.007s
sys     0m0.482s
```



```
[root@localhost ~]# docker run -it --rm --name testb --blkio-weight 1000 docker.benet.com/centos:centos7
[root@0da2333bd7a6 /]# time dd if=/dev/zero of=test.out bs=1M count=1024 oflag=direct
1024+0 records in
1024+0 records out
1073741824 bytes (1.1 GB) copied, 6.3896 s, 168 MB/s

real    0m6.456s
user    0m0.006s
sys     0m0.651s
```


注：oflag=direct规避掉文件系统的cache，把写请求直接封装成io指令发到硬盘

通俗的解释就是，一般情况下数据一般先写入到内存的缓存区，然后再从缓存区写入到硬盘中去，但是使用了这一条命令后，则不用先把数据写入到内存缓存区，而是直接写入到硬盘中去，则是为了实验的准确性。

四、学习Docker也有一段时间了，了解了Docker的基本实现原理，也知道了Docker的使用方法，这里对Docker的一些典型应用场景做一个总结

1、配置简化

这是Docker的主要使用场景。将应用的所有配置工作写入Dockerfile中，创建好镜像，以后就可以无限次使用这个镜像进行应用部署了。这大大简化了应用的部署，不需要为每次部署都进行繁琐的配置工作，实现了一次打包，多次部署。这大大加快了应用的开发效率，使得程序员可以快速搭建起开发测试环境，不用关注繁琐的配置工作，而是将所有精力都尽可能用到开发工作中去。

2、代码流水线管理

代码从开发环境到测试环境再到生产环境，需要经过很多次中间环节，Docker给应用提供了一个从开发到上线均一致的环境，开发测试人员均只需关注应用的代码，使得代码的流水线变得非常简单，这样应用才能持续集成和发布。

3、快速部署

在虚拟机之前，引入新的硬件资源需要消耗几天的时间。Docker的虚拟化技术将这个时间降到了几分钟，Docker只是创建一个容器进程而无需启动操作系统，这个过程只需要秒级的时间。

4、应用隔离

资源隔离对于提供共享hosting服务的公司是个强需求。如果使用VM，虽然隔离性非常彻底，但部署密度相对较低，会造成成本增加。
Docker容器充分利用linux内核的namespace提供资源隔离功能。结合cgroups，可以方便的设置每个容器的资源配额。既能满足资源隔离的需求，又能方便的为不同级别的用户设置不同级别的配额限制。

5、服务器资源整合

正如通过VM来整合多个应用，Docker隔离应用的能力使得Docker同样可以整合服务器资源。由于没有额外的操作系统的内存占用，以及能在多个实例之间共享没有使用的内存，Docker可以比VM提供更好的服务器整合解决方案。
通常数据中心的资源利用率只有30%，通过使用Docker并进行有效的资源分配可以提高资源的利用率。

6、多版本混合部署

随着产品的不断更新换代，一台服务器上部署多个应用或者同一个应用的多个版本在企业内部非常常见。但一台服务器上部署同一个软件的多个版本，文件路径、端口等资源往往会发生冲突，造成多个版本无法共存的问题。
如果用docker，这个问题将非常简单。由于每个容器都有自己独立的文件系统，所以根本不存在文件路径冲突的问题；对于端口冲突问题，只需要在启动容器时指定不同的端口映射即可解决问题。

7、版本升级回滚

一次升级，往往不仅仅是应用软件本身的升级，通过还会包含依赖项的升级。但新旧软件的依赖项很可能是不同的，甚至是有冲突的，所以在传统的环境

下做回滚一般比较困难。
如果使用docker，我们只需要每次应用软件升级时制作一个新的docker镜像，升级时先停掉旧的容器，然后把新的容器启动。需要回滚时，把新的容器停掉，旧的启动即可完成回滚，整个过程各在秒级完成，非常方便。

8、内部开发环境

在容器技术出现之前，公司往往是通过为每个开发人员提供一台或者多台虚拟机来充当开发测试环境。开发测试环境一般负载较低，大量的系统资源都被浪费在虚拟机本身的进程上了。
Docker容器没有任何CPU和内存上的额外开销，很适合用来提供公司内部的开发测试环境。而且由于Docker镜像可以很方便的在公司内部共享，这对开发环境的规范性也有极大的帮助。

9、PaaS

使用Docker搭建大规模集群，提供PaaS。这一应用是最有前景的一个了，目前已有很多创业公司在使用Docker做PaaS了，例如云雀云平台。用户只需提交代码，所有运维工作均由服务公司来做。而且对用户来说，整个应用部署上线是一键式的，非常方便。

10、云桌面

在每一个容器内部运行一个图形化桌面，用户通过RDP或者VNC协议连接到容器。该方案所提供的虚拟桌面相比于传统的基于硬件虚拟化的桌面方案更轻量级，运行速率大大提升。不过该方案仍处于实验阶段，不知是否可行。可以参考一下Docker-desktop方案。

五、补充知识点

1）测试机器环境

```
[root@localhost ~]# cat /etc/redhat-release
CentOS Linux release 7.2.1511 (Core)
[root@localhost ~]#
```


2）执行mount命令查看cgroup的挂载点

```
[root@localhost ~]# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devtmpfs on /dev type devtmpfs (rw,nosuid,seclabel,size=918824k,nr_inodes=229706,mode=755)
securityfs on /sys/kernel/security type securityfs (rw,nosuid,nodev,noexec,relatime)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev,seclabel)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,seclabel,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,nodev,seclabel,mode=755)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noexec,seclabel,mode=755)
cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,release_agent=/usr/lib/systemd/systemd-cgroups-agent,name=systemd)
pstore on /sys/fs/pstore type pstore (rw,nosuid,nodev,noexec,relatime)
cgroup on /sys/fs/cgroup/freezer type cgroup (rw,nosuid,nodev,noexec,relatime,freezer)
cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup (rw,nosuid,nodev,noexec,relatime,cpuacct,cpu)
cgroup on /sys/fs/cgroup/blkio type cgroup (rw,nosuid,nodev,noexec,relatime,blkio)
cgroup on /sys/fs/cgroup/perf_event type cgroup (rw,nosuid,nodev,noexec,relatime,perf_event)
cgroup on /sys/fs/cgroup/net_cls type cgroup (rw,nosuid,nodev,noexec,relatime,net_cls)
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/devices type cgroup (rw,nosuid,nodev,noexec,relatime,devices)
cgroup on /sys/fs/cgroup/hugetlb type cgroup (rw,nosuid,nodev,noexec,relatime,hugetlb)
cgroup on /sys/fs/cgroup/cpuset type cgroup (rw,nosuid,nodev,noexec,relatime,cpuset)
configfs on /sys/kernel/config type configfs (rw,relatime)
```

从上图可以看到 cgroup 挂载在/sys/fs/cgroup 目录

groups 可以限制 blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net_cls、ns 等系统的资源，以下是主要子系统的说明：

blkio 这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及 usb 等等。

cpu 这个子系统使用调度程序为 cgroup 任务提供 cpu 的访问。

cpuacct 产生 cgroup 任务的 cpu 资源报告。

cpuset 如果是多核心的 cpu，这个子系统会为 cgroup 任务分配单独的 cpu 和内存。

devices 允许或拒绝 cgroup 任务对设备的访问。

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 标记每个网络包以供 cgroup 方便使用，它通过使用等级识别符(classid)标记网络数据包，从而允许 Linux 流量控制程序（TC：Traffic Controller）识别从具体 cgroup 中生成的数据包。

ns：命名空间子系统

3）cgroups管理进程cpu资源

我们先看一个限制cpu资源的例子：

跑一个耗cpu的脚本

运行一个容器，在容器内创建脚本并运行脚本，脚本内容

```
[root@localhost ~]# docker run -it --name=test docker.io/centos:latest
[root@ab55f7962955 /]# vi cpu.sh
```

```
#!/bin/bash
i=0
while true
do
  let i++
done
```

```
[root@ab55f7962955 /]# chmod +x cpu.sh
[root@ab55f7962955 /]# ./cpu.sh
```

将容器切换到后台运行

在宿主机上top可以看到这个脚本基本占了100%的cpu资源

```
top - 21:18:43 up 54 min,  2 users,  load average: 0.86, 0.33, 0.33
Tasks: 462 total,   2 running, 460 sleeping,   0 stopped,   0 zombie
%Cpu(s): 24.0 us,  1.7 sy,  0.0 ni, 74.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  1868688 total,   863792 free,   627576 used,   377320 buff/cache
KiB Swap:  4194300 total,  4194300 free,        0 used.  1006612 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 5925 root      20   0   11636   1108    924 R 100.0  0.1   0:53.68 cpu.sh
```

下面用cgroups控制这个进程的cpu资源

对于centos7来说，通过systemd-cgls来查看系统cgroups tree：

```
[root@localhost ~]# systemd-cgls
```

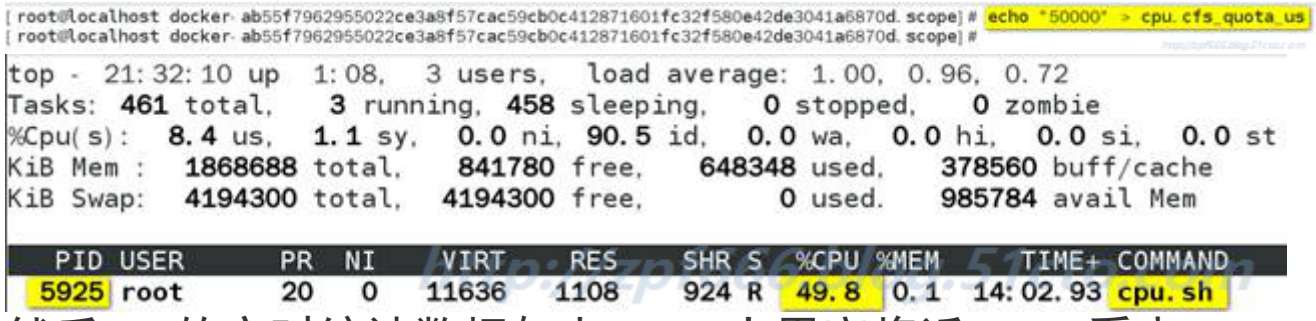
注：5925就是我们所运行的容器中脚本cpu.sh的pid进程号

说明：

cpu.cfs_period_us 就是时间周期，默认为100000，即百毫秒。

cpu.cfs_quota_us 就是在这期间内可使用的 cpu 时间， 默认 -1，即无限制。

现在将cpu.cfs_quota_us设为50000，相对于cpu.cfs_period_us的100000是50%，然后再用top看结果



然后top的实时统计数据如上，cpu占用率将近50%，看来cgroups关于cpu的控制起了效果。