**docker技术剖析--docker资源限制及应用总结 for centos7.2**

2016-09-03 16:19:32

标签：[target](http://blog.51cto.com/tag-target.html) [blank](http://blog.51cto.com/tag-blank.html) [世外桃源](http://blog.51cto.com/tag-%E4%B8%96%E5%A4%96%E6%A1%83%E6%BA%90.html)

<http://hongge.blog.51cto.com/>

Docker(linux container)所依赖的底层技术

**1 Namespace**

用来做容器的隔离，有了namespace，在docker container里头看来，就是一个完整的linux的世界。在host看来，container里的进程，就是一个普通的host进程，namespace提供这种pid的映射和隔离效果，host承载着container，就好比一个世外桃源。

namespace包括：pid namespace、pid namespace、ipc namespace、mnt namespace、uts namespace、user namespace

例如我们运行一个容器

[image](http://s3.51cto.com/wyfs02/M02/86/C8/wKioL1fKh3jwct9uAAAsx2emMsc309.png)

查看容器的进程号

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKiom1fKh3mQ1uS9AAAfv3x-pGw882.png)

可以看到该容器的pid是3894，在宿主的/proc目录下存在3894进程的目录

通过kill可以结束该容器

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKiom1fKh3mSFcIzAAAfqYMjvOo802.png)

**查看/proc/[pid]/ns文件**

从3.8版本的内核开始，用户就可以在/proc/[pid]/ns文件下看到指向不同namespace号的文件，效果如下所示，形如[4026531839]者即为namespace号。

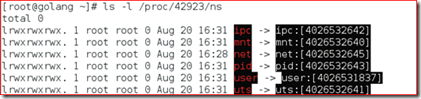
我们运行一个容器并获取容器的pid

[](http://s3.51cto.com/wyfs02/M02/86/C8/wKioL1fKh3rzNL4QAABs8-olXRI359.png)

获取容器的pid

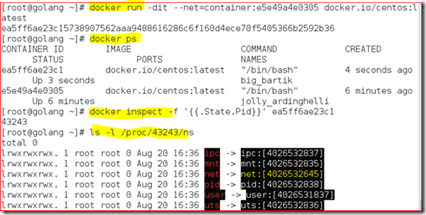
[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh3qxRZL3AAAgEWnEsWo719.png)

#ls -l /proc/pid/ns <<pid 表示应用容器的PID

[](http://s3.51cto.com/wyfs02/M02/86/C8/wKioL1fKh3uh3l2RAADCqUX5GKQ808.png)

如果两个进程指向的namespace编号相同，就说明他们在同一个namespace下，否则则在不同namespace里面。

例如我们再创建一个容器，网络模式为container （使用 --net=container:NAMEorID 指定）

[](http://s3.51cto.com/wyfs02/M00/86/C9/wKiom1fKh3yi7eU5AAFV7W96E3Q031.png)

从上面可以看出两个容器的net namespace编号相同，说明他们在同一个net namespace下，共用一个网络。

Docker使用了pid、network、mnt、ipc、uts等命名空间来隔离进程、网络、文件系统等资源。注意，由于Linux并不是namespace了所有东西（如cgroups、/sys、SELinux、/dev/sd\*、内核模块等），仅靠这几个namespace是无法实现像KVM那样的完全资源隔离的。

pid namespace：PID namespace隔离非常实用，它对进程PID重新标号，即两个不同namespace下的进程可以有同一个PID，实现进程隔离，容器只能看到自己的进程，并且每个容器都有一个pid为1的父进程，kill掉该进程容器内的所有进程都会停止；

net namespace：实现网络隔离，每个容器都可以设置自己的interface、routers、iptables等；docker默认采用veth的方式将container中的虚拟网卡同host上的一个docker bridge: docker0连接在一起；

ipc namespace：container中进程交互还是采用linux常见的进程间交互方法(interprocess communication - IPC)，容器中进程间通信采用的方法包括常见的信号量、消息队列和共享内存。然而与虚拟机不同的是，容器内部进程间通信对宿主机来说，实际上是具有相同PID namespace中的进程间通信，在同一个IPC namespace下的进程彼此可见，而与其他的IPC namespace下的进程则互相不可见。

mnt namespace：通过隔离文件系统挂载点对隔离文件系统提供支持，不同mnt namespace中的文件结构发生变化也互不影响。你可以通过/proc/[pid]/mounts查看到所有挂载在当前namespace中的文件系统，还可以通过/proc/[pid]/mountstats看到mount namespace中文件设备的统计信息，包括挂载文件的名字、文件系统类型、挂载位置等等

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKioL1fKh33TafAXAAAIhK2w61g604.png)

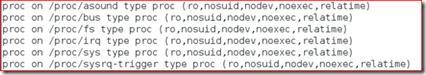
注：43234是容器的进程号

uts namspace：UTS namespace提供了主机名和域名的隔离，这样每个容器就可以拥有了独立的主机名和域名，在网络上可以被视作一个独立的节点而非宿主机上的一个进程。

user namespace：每个container可以有不同的 user 和 group id, 也就是说可以在container内部用container内部的用户执行程序而非Host上的用户。

对于容器所依赖的内核文件系统（这些都是non-namespaced），为了保证安全性，docker将其限制为只读的，例如进入一个容器执行mount命令：

#mount

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh33z-B_MAACGMjX-lcQ022.png)

**2 Cgroups**

在前面了解了Docker背后使用的资源隔离技术namespace，通过系统调用构建一个相对隔离的shell环境，也可以称之为一个简单的“容器”。下面我们则要开始讲解另一个强大的内核工具——cgroups。他不仅可以限制被namespace隔离起来的资源，还可以为资源设置权重、计算使用量、操控进程启停等等。所以cgroups（Control groups）实现了对资源的配额和度量。

cgroups是什么？

cgroups（Control Groups）最初叫Process Container，由Google工程师（Paul Menage和Rohit Seth）于2006年提出，后来因为Container有多重含义容易引起误解，就在2007年更名为Control Groups，并被整合进Linux内核。顾名思义就是把进程放到一个组里面统一加以控制

groups的作用

通俗的来说，cgroups可以限制、记录、隔离进程组所使用的物理资源（包括：CPU、memory、IO等），为容器实现虚拟化提供了基本保证，是构建Docker等一系列虚拟化管理工具的基石。Cgroups提供了以下四大功能。

1）资源限制（Resource Limitation）：cgroups可以对进程组使用的资源总额进行限制。如设2）定应用运行时使用内存的上限，一旦超过这个配额就发出OOM（Out of Memory）。

优先级分配（Prioritization）：通过分配的CPU时间片数量及硬盘IO带宽大小，实际上就相当于控制了进程运行的优先级。

3）资源统计（Accounting）： cgroups可以统计系统的资源使用量，如CPU使用时长、内存用量等等，这个功能非常适用于计费。

4）进程控制（Control）：cgroups可以对进程组执行挂起、恢复等操作。

下面就介绍cgroup如何做到内存，cpu和io速率的隔离

本文用脚本运行示例进程，来验证Cgroups关于cpu、内存、io这三部分的隔离效果。

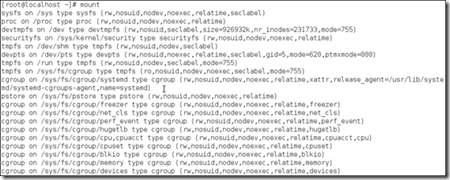
测试机器环境

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh36AwB7DAAARXOD8m18983.png)

启动Cgroups

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh37DK30uAAAv4b9tRJs980.png)

执行mount命令查看cgroup的挂载点

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKiom1fKh3-ReBBPAAExQmRhYsw234.png)

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。

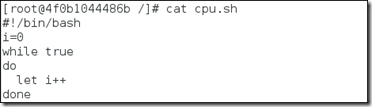
ns：命名空间子系统

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：

[](http://s3.51cto.com/wyfs02/M00/86/C9/wKioL1fKh4CDuoYxAAAVrRoLMB8071.png)

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4Ci624AAAA2PlLVtTQ055.png)

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90%多的cpu资源

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh4CzYJzhAAAmta1gCCY314.png)

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：

#systemd-cgls

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4Hgd3NrAABBfC9ylK0993.png)

注：4813就是我们所运行的容器pid

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh4KzwEkbAAAqcBZW96Q302.png)

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKiom1fKh4LDv0bwAAA5jjmxk_c328.png)

将cpu.cfs\_quota\_us设为50000，相对于cpu.cfs\_period\_us的100000是50%

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4Ohgi0hAAAfXK2eEzg340.png)

进入容器，再次执行脚本，打开宿主机的另一个终端执行top命令

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

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh4PDGLl8AAAsU4-kRVY973.png)

**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（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。

**CFS调度策略下的配置**

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。例如：

使用命令docker run -tid –cpu-shares 100 镜像，创建容器，则最终生成的cgroup的cpu份额配置可以下面的文件中找到：

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

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh4TQhr2VAAA3k03BlXs817.png)

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh4SQdyXBAAA4XZnrADs117.png)

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。

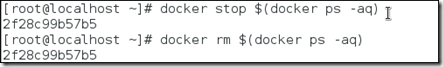
该加权值是一个整数（必须大于等于2）表示相对权重，最后除以权重总和算出相对比例，按比例分配CPU时间。

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：

先删除docker主机上运行的容器

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKiom1fKh4Xjlv8hAAAczPLdUoE354.png)

Docker**通过--cpu-shares 指定CPU份额**

运行一个容器指定cpu份额为1024

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4WguOsxAAAr3aZZ5KU792.png)

注：

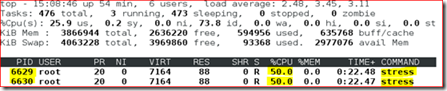
**--cpu-shares 指定CPU份额，默认就是1024**

**--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。

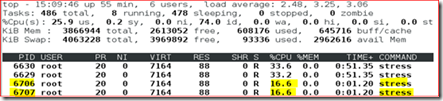
在docker宿主机上打开一个terminal执行top

[](http://s3.51cto.com/wyfs02/M00/86/C9/wKiom1fKh4bC51D2AACh3qcJA8A027.png)

然后再启动一个容器， --cpu-shares为512。

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh4eigQn4AAAd8pW-BF0424.png)

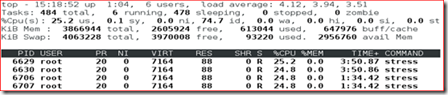
查看top的现实结果

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKiom1fKh4eDQcRxAACRm8VskH0257.png)

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

将container1的cpu.shares改为512，

#echo “512” > /sys/fs/cgroup/cpu/system.slice/docker-<容器的完整长ID>/cpu.shares

[](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4qQPqvlAACW69pabFM136.png)

可以看到两个容器的CPU占比趋于平均

设定CPU使用周期使用时间上限

cgroups 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）。

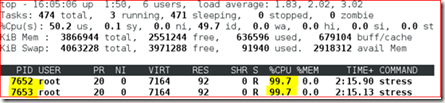
当然，在多核情况下，若cfs\_quota\_us是cfs\_period\_us的两倍，就表示在两个核上完全使用CPU，例如如果允许容器进程需要完全占用两个CPU，则可以将cpu-period设置为100000（即0.1秒），cpu-quota设置为200000（0.2秒）。

使用示例：

使用命令docker run创建容器

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh4rAMmGLAAAomWhs1Z4224.png)

在宿主机上执行top

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKiom1fKh4vCKCxsAACt1UohOcQ903.png)

从上图可以看到基本占了100%的cpu资源

则最终生成的cgroup的cpu周期配置可以下面的目录中找到：

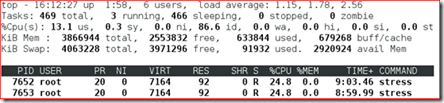
/sys/fs/cgroup/cpu/system.slice/docker-<容器的完整长ID>/

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKioL1fKh4zCyc58AABGP5DILJ0061.png)

修改容器的cpu.cfs\_period\_us 和 cpu.cfs\_quota\_us值

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh4zwpMHtAABioYVekq4880.png)

执行top查看cpu资源

[](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh42jXiRAAACo1eHQBPQ755.png)

从上图可以看到基本占了50%的cpu资源

**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。

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh46RUjG9AABOS2FmSEM635.png)

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：

[](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh46zlHHlAABWSXqW4UU603.png)

表示创建的容器只能用0、1、2这三个内核。最终生成的cgroup的cpu内核配置如下：

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKioL1fKh4_z5Wi7AAAqP4VYlWs271.png)

**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memory node，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。

**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 --memory 128m 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。

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh4_QtbZuAAA6EFH-r7s775.png)

可以发现，使用256MB进行压力测试时，由于超过了内存上限（128MB内存+128MB swap），进程被OOM(out of memory)杀死。

使用250MB进行压力测试时，进程可以正常运行。

[image](http://s3.51cto.com/wyfs02/M00/86/C9/wKioL1fKh5CBhnOyAAA_Y85BE94918.png)

通过docker stats可以查看到容器的内存已经满负载了。

#docker stats test2

[image](http://s3.51cto.com/wyfs02/M01/86/C9/wKioL1fKh5HC8Oo5AAAh_JiLJIo764.png)

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)。

#cat /sys/fs/cgroup/memory/system.slice/docker-<容器的完整ID>/memory.limit\_in\_bytes   
134217728

#cat /sys/fs/cgroup/memory/system.slice/docker-<容器的完整ID>/memory.memsw.limit\_in\_bytes   
268435456

[image](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh5GgvBZQAABBEkG6YO8446.png)

**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。

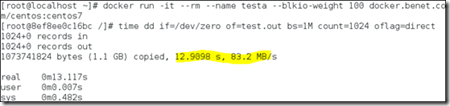
--blkio-weight-device： 针对特定设备的IO加权控制。其格式为DEVICE\_NAME:WEIGHT

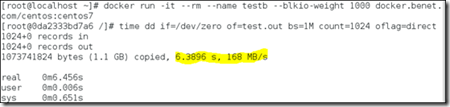
**磁盘IO配额控制示例**

**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

[](http://s3.51cto.com/wyfs02/M02/86/C9/wKiom1fKh5OhVRqCAABnbB8aUUE788.png)

[](http://s3.51cto.com/wyfs02/M02/86/C9/wKioL1fKh5Phvq_8AABpHBWd9lQ333.png)

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

**3 Chroot**

如何在container里头，看到的文件系统，就是一个完整的linux系统，有/etc、/lib 等，通过chroot实现

**4 Veth**

container里，执行ifconfig可以看到eth0的网卡，如何通信呢？其实是在host上虚拟了一张网卡出来（veth73f7），跟container里的网卡做了桥接，所有从container出来的流量都要过host的虚拟网卡，进container的流量也是如此。

**5 Union FS**

对于这种叠加的文件系统，有一个很好的实现是AUFS，这个可以做到以文件为粒度的copy-on-write，为海量的container的瞬间启动。

**6 Iptables, netfilter**

主要用来做ip数据包的过滤，比如可以做container之间无法通信，container可以无法访问host的网络，但是可以通过host的网卡访问外网等这样的网络策略

二、学习[Docker](http://udn.yyuap.com/portal.php?mod=topic&topicid=5)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。