

原创

# docker底层原理介绍

 mb5cd21e691f31a [关注](#)

2019-12-09 14:31:50 421人阅读 0人评论

链接：<https://blog.51cto.com/14320361/2457143>

## 1.docker介绍

### 1.1什么是docker

Docker 是一个开源的应用容器引擎，基于 Go 语言 并遵从Apache2.0协议开源。

Docker 可以让开发者打包他们的应用以及依赖包到一个轻量级、可移植的容器中，然后发布到任何流行的 Linux 机器上。

## 1.2docker能解决什么问题

### 1.2.1高效有序利用资源

- 机器资源有限；
- 单台机器得部署多个应用；
- 应用之间互相隔离；
- 应用之间不能发生资源抢占，每个应用只能使用事先注册申请的资源。

### 1.2.2一次编译，到处运行

类似于java代码，应用及依赖的环境构建一次，可以到处运行。

## 1.2.docker底层原理介绍

### 1.2.1Linux的namespace和cgroup简单理解

namespace:类似于JAVA的命名空间

controll groups： controll （system resource） （for） （process） groups

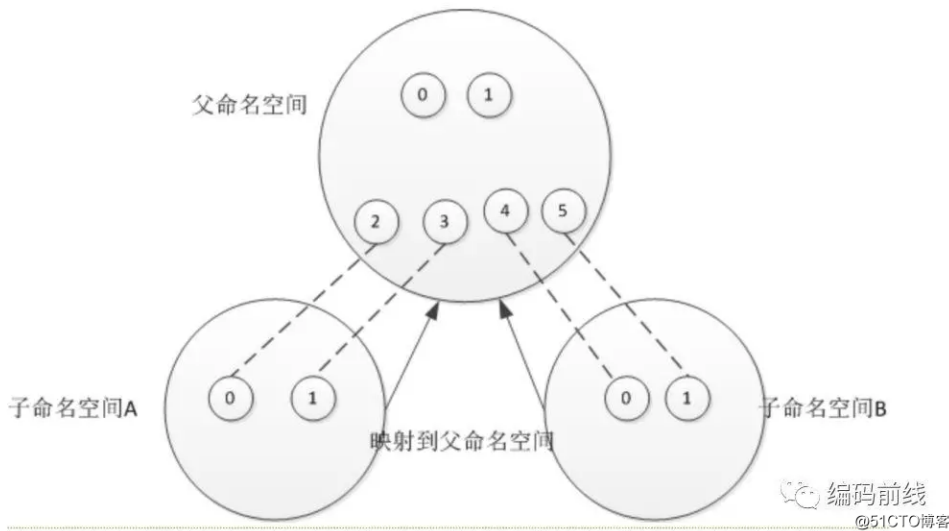
### 1.2.2Linux中的namespace

在Linux系统中，可以同时存在多用户多进程，那么对他们的运行协调管理，通过进程调度和进度管理可以解决，但是，整体资源是有限的，怎么把有限的资源（进程号、网络资源等等）合理分配给各个用户所在的进程？



Linux Namespaces机制提供一种资源隔离方案。PID,IPC,Network等系统资源不再是全局性的，而是属于某个特定的Namespace。每个namespace下的资源对于其他namespace下的资源都是透明，不可见的。因此在操作系统层面上看，就会出现多个相同pid的进程。系统中可以同时存在两个进程号为0,1,2的进程，由于属于不同的namespace，所以它们之间并不冲突。而在用户层面上只能看到属于用户自己namespace下的资源，

例如使用ps命令只能列出自己namespace下的进程。这样每个namespace看上去就像一个单独的Linux系统。



命名空间建立系统的不同视图，对于每一个命名空间，从用户看起来，应该像一台单独的Linux计算机一样，有自己的init进程(PID为0)，其他进程的PID依次递增，A和B空间都有PID为0的init进程，子容器的进程映射到父容器的进程上，父容器可以知道每一个子容器的运行状态，而子容器与子容器之间是隔离的。

```
<colgroup style="margin: 0px; padding: 0px; max-width: 100%; box-sizing: border-box !important; word-wrap: break-word !important;"><col style="margin: 0px; padding: 0px; max-width: 100%; box-sizing: border-box !important; word-wrap: break-word !important;"><col style="margin: 0px; padding: 0px; max-width: 100%; box-sizing: border-box !important; word-wrap: break-word !important;"><col style="margin: 0px; padding: 0px; max-width: 100%; box-sizing: border-box !important; word-wrap: break-word !important;"></colgroup>
```

- |
- namespace
- |
- 引入的相关内核版本
- |
- 被隔离的全局系统资源
- |
- 在容器语境下的隔离效果

Mount namespaces	Linux 2.4.19	文件系统挂载点

将一个文件系统的顶层目录挂到另一个文件系统的子目录上，使它们成为一个整体，称为挂载。把该子目录称为挂载点。

Mount namespace用来隔离文件系统的挂载点,使得不同的mount namespace拥有自己独立的挂载点信息,不同的namespace之间不会相互影响，这对于构建用户或者容器自己的文件系统目录非常有用。

- |
- UTS namespaces | Linux 2.6.19 | nodename 和 domainname |

UTS, UNIX Time-sharing System namespace提供了主机名和域名的隔离。能够使得子进程有独立的主机名和域名(hostname)，这一特性在Docker容器技术中被用到，使得docker容器在网络上被视作一个独立的节点，而不仅仅是宿主机上的一个进程。

- |
- IPC namespaces | Linux 2.6.19 | 特定的进程间通信资源，包括System V IPC 和 POSIX message queues |



在线客服

IPC全称 Inter-Process Communication，是Unix/Linux下进程间通信的一种方式，IPC有共享内存、信号量、消息队列等方法。所以，为了隔离，我们也需要把IPC给隔离开来，这样，只有在同一个Namespace下的进程才能相互通信。如果你熟悉IPC的原理的话，你会知道，IPC需要有一个全局的ID，即使是全局的，那么就意味着我们的Namespace需要对这个ID隔离，不能让别的Namespace的进程看到。

|  
| PID namespaces | Linux 2.6.24 | 进程 ID 数字空间 (process ID number space) |

PID namespaces用来隔离进程的ID空间，使得不同pid namespace里的进程ID可以重复且相互之间不影响。

PID namespace可以嵌套，也就是说有父子关系，在当前namespace里面创建的所有新的namespace都是当前namespace的子namespace。父namespace里面可以看到所有子孙后代namespace里的进程信息，而子namespace里看不到祖先或者兄弟namespace里的进程信息。

|  
| Network namespaces | 始于Linux 2.6.24 完成于 Linux 2.6.29 | 网络相关的系统资源 | 每个容器用有其独立的网络设备，IP 地址，IP 路由表，/proc/net 目录，端口号等等。这也使得一个 host 上多个容器内的同一个应用都绑定到各自容器的 80 端口上。|

| User namespaces | 始于 Linux 2.6.23 完成于 Linux 3.8) | 用户和组 ID 空间 |

User namespace用来隔离user权限相关的Linux资源，包括user IDs and group IDs。

这是目前实现的namespace中最复杂的一个，因为user和权限息息相关，而权限又事关容器的安全，所以稍有不慎，就会出安全问题。

在不同的user namespace中，同样一个用户的user ID 和group ID可以不一样，换句话说，一个用户可以在父user namespace中是普通用户，在子user namespace中是超级用户

## 1.3 Namespace (名称空间)

### 用来隔离容器

```
[root@localhost ~]# docker run -it --name test centos /bin/bash
//进入到容器里面
```

```
[root@localhost ~]# docker run -it --name test centos /bin
/bash
Unable to find image 'centos:latest' locally
latest: Pulling from library/centos
729ec3a6ada3: Pull complete
Digest: sha256:6ab380c5a5acf71c1b6660d645d2cd79cc8ce91b38e
0352cbf9561e050427baf
Status: Downloaded newer image for centos:latest
[root@41052cceb473 /]#
```

```
[root@41052cceb473 /]# ls
//查看一下和宿主机差不多，都是从宿主机链接过来的
```

```
[root@41052cceb473 /]# ls
bin  home  lost+found  opt  run  sys  var
dev  lib   media       proc sbin tmp
etc  lib64 mnt         root srv  usr
```

```
[root@41052cceb473 /]# uname -r
//查看一下内核，和宿主机也是一样的
```

```
[root@localhost ~]# uname -r
3.10.0-693.el7.x86_64
```

如果虚拟机内服务对内核版有要求，这个服务就不太适合用docker来实现了，因为docker就是共用宿主机的内核，可以使用kvm之类的虚拟机。



在线  
客服

```
[root@localhost ~]# docker pull ubuntu
//使用docker下载一个Ubuntu

[root@localhost ~]# docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
7ddbc47eeb70: Pull complete
c1bbdc448b72: Pull complete
8c3b70e39044: Pull complete
45d437916d57: Pull complete
Digest: sha256:44ed9bfa3f850417f4036afc34ce97559c464c708a9
b42f8be14392921e8bc42
Status: Downloaded newer image for ubuntu:latest

[root@localhost ~]# docker images
//查看一下
```

REPOSITORY	SIZE	TAG	IMAGE ID	CREATED
centos	203MB	7	5e35e350aded	3 weeks ag
ubuntu	64.2MB	latest	775349758637	5 weeks ag
centos	220MB	latest	0f3e07c0138f	2 months a

```
[root@localhost ~]# docker run -it ubuntu:latest /bin/bash
//进入ubuntu环境
root@aafbee6750865:/# ls /
//查看一下
```

```
root@aafbee6750865:/# ls /
bin dev home lib64 mnt proc run srv tmp var
boot etc lib media opt root sbin sys usr
```

```
root@48c8dd7b098e:/# uname -r
//查看一下内核
```

```
root@48c8dd7b098e:/# uname -r
3.10.0-693.el7.x86_64
```

Docker本身不占用任何端口，他一般是在后台运行，无论在docker里进行什么操作（系统、服务）对于docker来说他们仅仅就是一个进程

Run-□centos系统（nginx，web）

Busybox：欺骗层。欺骗docker中的虚拟机是在自己独立的环境中

解耦：解除耦合、冲突。

耦合：冲突现象。

#### 1.4 Namespace操作

/proc /sys:虚拟文件系统，伪目录文件

```
[root@localhost ~]# cd /proc/
[root@localhost proc]# ls

[root@localhost proc]# ls
1      1898  2259  3864  5648  719      key-users
10     19    2261  387   6      721      kmsg
1000   1955  2263  388   605     722      kpagecoun
1001   1960  2269  401   606     724      kpageflag
1004   2     2270  402   607     729      loadavg
1005   20    2274  403   608     731      locks
1009   2053  2287  404   609     732      mdstat
1020   2068  2291  405   610     741      meminfo
1022   2073  2293  406   611     8        misc
1023   2075  23    407   612     9        modules
1024   2094  2338  408   620     98       mounts
1059   21    2390  409   621     acpi     mpt
```

在线  
客服

```
[root@localhost proc]# echo $$
//当前的进程编号
3864
[root@localhost proc]# cd 3864
[root@localhost 3864]# cd ns
[root@localhost ns]##
//可以看到一闪一闪的
```

```
[root@localhost ns]# ll
总用量 0
lrwxrwxrwx 1 root root 0 12月 9 09:41 ipc ->
lrwxrwxrwx 1 root root 0 12月 9 09:41 mnt ->
lrwxrwxrwx 1 root root 0 12月 9 09:41 net ->
lrwxrwxrwx 1 root root 0 12月 9 09:41 pid ->
lrwxrwxrwx 1 root root 0 12月 9 09:41 user ->
lrwxrwxrwx 1 root root 0 12月 9 09:41 uts ->
```

```
[root@localhost ns]# ls
[root@localhost ns]# ls
ipc mnt net pid user uts
```

IPC:共享内存、消息队列

MNT:挂载点、文件系统

NET:网络栈

PID: 进程编号

USER:用户、组

UTS:主机名、域名

namespec这六项隔离，实现了容器与宿主机，容器与容器之间的隔离

//创建一个用户并设置密码

```
[root@localhost ns]# useradd bdqn
[root@localhost ns]# echo 123.com | passwd --stdin bdqn
[root@localhost ns]# id bdqn
```

```
[root@localhost ns]# id bdqn
uid=1001(bdqn) gid=1001(bdqn) 组=1001(bdqn)
```

查看docker进程

```
[root@localhost ns]# docker ps -a
[root@localhost ns]# docker ps -a
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
48c8dd7b098e   ubuntu:latest  "/bin/bash"            About an h    Exited (0) 41 minutes ago
afbee6750865   ubuntu:latest  "/bin/bash"            About an h    Exited (0) About an hour ago
41052cceb473   centos         "/bin/bash"            About an h    Exited (0) About an hour ago
```

```
[root@localhost ns]# docker start test
//启动centos
[root@localhost ns]# docker exec -it test /bin/bash
//进入docker容器
[root@41052cceb473 /]# id bdqn
```

```
[root@41052cceb473 /]# id bdqn
id: 'bdqn': no such user
```

```
[root@41052cceb473 /]# echo $$
[root@41052cceb473 /]# echo $$
14
```



在线  
客服

## 2.1linux cgroup介绍

### 2.1.1有了namespace为什么还要cgroup:

Docker 容器使用 linux namespace 来隔离其运行环境，使得容器中的进程看起来就像一个独立环境中运行一样。但是，光有运行环境隔离还不够，因为这些进程还是可以不受限制地使用系统资源，比如网络、磁盘、CPU以及内存等。关于其目的，一方面，是为了防止它占用了太多的资源而影响到其它进程；另一方面，在系统资源耗尽的时候，linux 内核会触发 OOM，这会让一些被杀掉的进程成了无辜的替死鬼。因此，为了让容器中的进程更加可控，Docker 使用 Linux cgroups 来限制容器中的进程允许使用的系统资源。

### 2.1.2原理

Linux Cgroup 可为系统中所运行任务（进程）的用户定义组群分配资源 — 比如 CPU 时间、系统内存、网络带宽或者这些资源的组合。可以监控管理员配置的 cgroup，拒绝 cgroup 访问某些资源，甚至在运行的系统中动态配置 cgroup。所以，可以将 controll groups 理解为 controller（system resource）（for）（process）groups，也就是说它以一组进程为目标进行系统资源分配和控制。它主要提供了如下功能：

Resource limitation: 限制资源使用，比如内存使用上限以及文件系统的缓存限制。

Prioritization: 优先级控制，比如：CPU利用和磁盘IO吞吐。

Accounting: 一些审计或一些统计，主要目的是为了计费。

Controll: 挂起进程，恢复执行进程。

使用 cgroup，系统管理员可更具体地控制对系统资源的分配、优先顺序、拒绝、管理和监控。可更好地根据任务和用户分配硬件资源，提高总体效率。

在实践中，系统管理员一般会利用CGroup做下面这些事：

隔离一个进程集合（比如：nginx的所有进程），并限制他们所消费的资源，比如绑定CPU的核。

为这组进程分配其足够使用的内存

为这组进程分配相应的网络带宽和磁盘存储限制

限制访问某些设备（通过设置设备的白名单）



## 2.1.3Cgroup(控制组)操作

## 资源的限制，docker对于资源的占用

```
[root@localhost ~]# cd /sys/fs/cgroup/
//对cpu，内存限制的目录
[root@localhost cgroup]# ls
```

```
[root@localhost ~]# cd /sys/fs/cgroup/
[root@localhost cgroup]# ls
blkio      cpu,cpuacct  freezer     net_cls     perf_event
cpu        cpuset       hugetlb     net_cls,net_prio  pids
cpuacct    devices      memory      net_prio    systemd
```

```
[root@localhost cgroup]# cd cpu
[root@localhost cpu]# ls
```

```
[root@localhost cgroup]# cd cpu
[root@localhost cpu]# ls
cgroup.clone_children  cpu.cfs_period_us  notify_on_release
cgroup.event_control  cpu.cfs_quota_us  release_agent
cgroup.procs           cpu.rt_period_us  system.slice
cgroup.sane_behavior  cpu.rt_runtime_us tasks
cpuacct.stat          cpu.shares        user.slice
cpuacct.usage         cpu.stat          @51CTO博客
cpuacct.usage_percpu  docker
```

cpu.shares: 权重

tasks: 这个文件内的数字，记录的是进程编号。PID

```
[root@localhost cpu]# cd docker/
[root@localhost docker]# ls
```

在线  
客服



```
[root@localhost ~]# ls
41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70/
cgroup.clone_children
cgroup.event_control
cgroup.procs
cpuacct.stat
cpuacct.usage
cpuacct.usage_percpu
cpu.cfs_period_us
```

@51CTO博客

```
[root@localhost ~]# cat tasks
```

//里面是空的

```
[root@localhost ~]# cd 41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70/
```

```
[root@localhost 41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70]# ls
```

```
41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70]# ls
cgroup.clone_children  cpuacct.usage_percpu  cpu.shares
cgroup.event_control  cpu.cfs_period_us    cpu.stat
cgroup.procs          cpu.cfs_quota_us     notify_on_release
cpuacct.stat          cpu.rt_period_us     tasks
cpuacct.usage         cpu.rt_runtime_us
```

@51CTO博客

```
[root@localhost 41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70]# cat tasks
```

```
41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70]# cat tasks
7875
8457
```

@51CTO博客

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

```
41052cceb4739fa8e0ddd2ffa733a78cd1043b3dff874cd266c009391a34d70]# docker ps
CONTAINER ID   IMAGE      PORTS          COMMAND                  NAMES      CREATED
41052cceb473   centos     Up About an hour   "/bin/bash"             test       2 hours ago
```

@51CTO博客

#### 四大功能:

- 1) 资源的限制: cgroup可以对进程组使用的资源总额进行限制
- 2) 优先级分配: 通过分配的cpu时间片数量以及硬盘IO带宽的大小, 实际上相当于控制了进程运行的优先级
- 3) 资源统计: group可以统计系统资源使用量, 比如gpu使用时间, 内存使用量等, 用于按量计费。同时还支持挂起功能, 也就是说通过cgroup把所有资源限制起来, 对资源都不能使用, 注意着并不是说我们的程序不能使用, 只是不能使用资源, 处于等待状态。
- 4) 进程控制: 可以对进程组执行挂起、恢复等操作。

#### 2.1.4 内存限额

容器内存包括两个部分: 物理内存和swap

可以通过参数控制容器内存的使用量:

-m或者--memory:设置内存的使用限额

--memory-swap:设置内存+ swap的使用限额

#### 举个例子:

运行一个容器, 并且限制该容器最多使用200M内存和100M的swap

```
[root@localhost ~]# docker run -it -m 200M --memory-swap 300M centos:7
```

```
[root@fba67fec2718 ~]# cd /sys/fs/cgroup/
```

```
[root@fba67fec2718 cgroup]# ls
```

```
[root@fba67fec2718 cgroup]# ls
blkio      cpuacct  freezer  net_cls  perf_event
cpu        cpuset  hugetlb  net_cls,net_prio  pids
cpu,cpuacct devices  memory   net_prio  systemd
```

@51CTO博客

```
[root@fba67fec2718 cgroup]# cd memory/
```

```
[root@fba67fec2718 memory]# ls
```

```
[root@fba67fec2718 memory]# cat memory.limit_in_bytes
```

//查看内存使用限制, (单位字节)

```
[root@fba67fec2718 memory]# cat memory.limit_in_bytes
209715200
```

@51CTO博客

```
[root@fba67fec2718 memory]# cat memory.memsw.limit_in_bytes
//查看交换分区，内存+swap限制
```

```
[root@fba67fec2718 memory]# cat memory.memsw.limit_in_bytes
314572800 @51CTO博客
```

### 运行一个新容器，并且不限制该容器

```
[root@localhost ~]# docker run -it centos:7
[root@5be901bfb093 /]# cd /sys/fs/cgroup/memory/
[root@5be901bfb093 memory]# cat memory.limit_in_bytes
//查看内存限制
```

```
[root@5be901bfb093 memory]# cat memory.limit_in_bytes
9223372036854771712 @51CTO博客
```

```
[root@5be901bfb093 memory]# cat memory.memsw.limit_in_bytes
//查看交换分区，内存+swap限制
```

```
[root@5be901bfb093 memory]# cat memory.memsw.limit_in_bytes
9223372036854771712 @51CTO博客
```

对比一个没有限制的容器，我们会发现，如果运行容器之后不限制内存的话，意味着没有限制。

### 2.1.5 CPU使用

通过-c或者--cpu-shares设置容器使用cpu的权重。如果不设置默认为1024。

#### 举个例子：

没有限制

```
[root@localhost ~]# docker run -it --name containerA centos:7
//没有限制，1024
[root@8683d8ff8234 /]# cd /sys/fs/cgroup/cpu
[root@8683d8ff8234 cpu]# cat cpu.shares
```

```
[root@8683d8ff8234 cpu]# cat cpu.shares
1024 @51CTO博客
```

限制CPU使用权重为512

```
[root@localhost ~]# docker run -it --name containerB -c 512 centos:7
//限制CPU使用权重为512
[root@d919d906295d /]# cd /sys/fs/cgroup/cpu
//可以看到cpu已经限制了
```

```
[root@d919d906295d cpu]# cat cpu.shares
512 @51CTO博客
```

### 2.1.6 容器的Block IO

#### 磁盘的读写。

Docker中可以通过设置权重，限制bps和iops的方式控制容器读写磁盘的IO

bps:每秒读写的数据量byte per second

iops:每秒IO的次数 io per second。

默认情况下，所有容器都能够平等的读写磁盘，也可以通过--blkio-weight参数改变容器的blockIO的优先级。

--device-read-bps:显示读取某个设备的bps。

--device-write-bps:显示写入某个设备的bps。

--device-read-iops:显示读取某个设备的iops。

--device-write-iops:显示写入某个设备的iops。

限制testA这个容器，写入/dev/sda这块磁盘的bps为30MB

```
[root@localhost ~]# docker run -it --name testA --device-write-bps /dev/sda:30MB centos:7
```



在线  
客服



```
[root@60e59e96fc16 /]# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct
//从/dev/zero输入，然后输出到test.out文件中，每次大小为1M，总共800次,oflag=direct 用来指定directI/O方式

[root@60e59e96fc16 /]# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct
0 oflag=direct
800+0 records in
800+0 records out
838860800 bytes (839 MB) copied, 26.6194 s, 31.5 MB/s

real    0m26.629s
user    0m0.000s
sys     0m0.822s

[root@60e59e96fc16 /]# du -h test.out
800M    test.out
```

docker没有限制

```
[root@localhost ~]# docker run -it --name testc centos:7
[root@5bf5f3d60d0e /]# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct

[root@5bf5f3d60d0e /]# time dd if=/dev/zero of=test.out bs=1M count=800 oflag=direct
00 oflag=direct
800+0 records in
800+0 records out
838860800 bytes (839 MB) copied, 1.15669 s, 725 MB/s

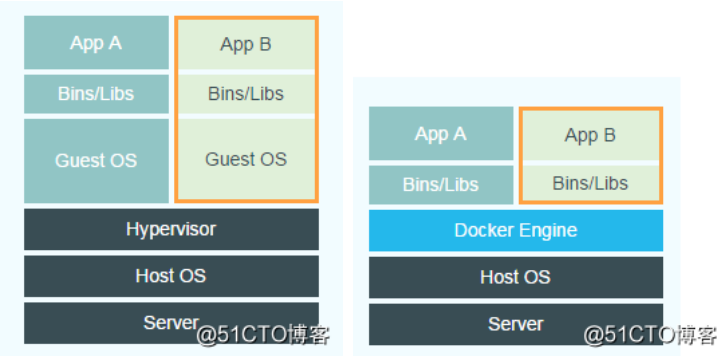
real    0m1.158s
user    0m0.000s
sys     0m0.996s

[root@5bf5f3d60d0e /]# du -h test.out
800M    test.out
```

### 3.Docker虚拟化与普通虚拟化的区别是什么？

**虚拟机：**  
我们传统的虚拟机需要模拟整台机器包括硬件，每台虚拟机都需要有自己的操作系统，虚拟机一旦被开启，预分配给他的资源将全部被占用。每一个虚拟机包括应用，必要的二进制和库，以及一个完整的用户操作系统。

**Docker：**  
容器技术是和我们的宿主主机共享硬件资源及操作系统可以实现资源的动态分配。容器包含应用和其所有的依赖包，但是与其他容器共享内核。容器在宿主主机操作系统中，在用户空间以分离的进程运行。



虚拟机和容器都是在硬件和操作系统以上的，虚拟机有Hypervisor层，Hypervisor是整个虚拟机的核心所在。他为虚拟机提供了虚拟的运行平台，管理虚拟机的操作系统运行。每个虚拟机都有自己的系统和系统库以及应用。

容器没有Hypervisor这一层，并且每个容器是和宿主主机共享硬件资源及操作系统，那么由Hypervisor带来性能的损耗，在linux容器这边是不存在的。

但是虚拟机技术也有其优势，能为应用提供一个更加隔离的环境，不会因为应用程序的漏洞给宿主机造成任何问题。同时还支持跨操作系统的虚拟化，例如你可以在linux操作系统下运行windows虚拟机。

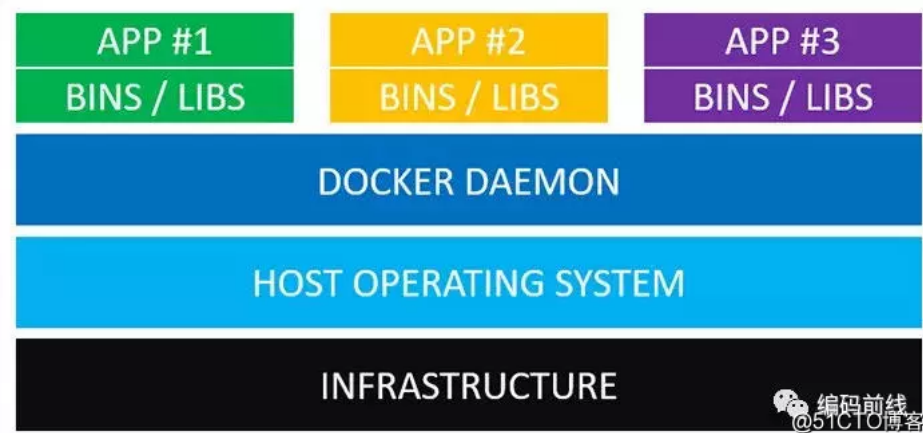
从虚拟化层面来看，传统虚拟化技术是对硬件资源的虚拟，容器技术则是对进程的虚拟，从而可提供更轻量级的虚拟化，实现进程和资源的隔离。

从架构来看，Docker比虚拟化少了两层，取消了hypervisor层和GuestOS层，使用 Docker Engine 进行调度和隔离，所有应用共用主机操作系统，因此在体量上，Docker较虚拟机更轻量级，在性能上优于虚拟化，接近裸机性能。从应用场景来看，Docker和虚拟化则有各自擅长的领域，在软件开发、测试场景和生产运维场景中各有优劣

具体对比：

- 1. docker启动快速属于秒级别。虚拟机通常需要几分钟去启动。
- 2. docker需要的资源更少，docker在操作系统级别进行虚拟化，docker容器和内核交互，几乎没有性能损耗，性能优于通过Hypervisor层与内核层的虚拟化。；
- 3. docker更轻量，docker的架构可以共用一个内核与共享应用程序库，所占内存极小。同样的硬件环境，Docker运行的镜像数远多于虚拟机数量。对系统的利用率非常高
- 4. 与虚拟机相比，docker隔离性更弱，docker属于进程之间的隔离，虚拟机可实现系统级别隔离；
- 5. 安全性：docker的安全性也更弱。Docker的租户root和宿主机root等同，一旦容器内的用户从普通用户权限提升为root权限，它就直接具备了宿主机的root权限，进而可进行无限制的操作。虚拟机租户root权限和宿主机的root虚拟机权限是分离的，并且虚拟机利用如Intel的VT-d和VT-x的ring-1硬件隔离技术，这种隔离技术可以防止虚拟机突破和彼此交互，而容器至今还没有任何形式的硬件隔离，这使得容器容易受到\*\*\*。
- 6. 可管理性：docker的集中化管理工具还不算成熟。各种虚拟化技术都有成熟的管理工具，例如VMware vCenter提供完备的虚拟机管理能力。
- 7. 高可用和可恢复性：docker对业务的高可用支持是通过快速重新部署实现的。虚拟化具备负载均衡，高可用，容错，迁移和数据保护等经过生产实践检验的成熟保障机制，VMware可承诺虚拟机99.999%高可用，保证业务连续性。
- 8. 快速创建、删除：虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节约大量时间。
- 9. 交付、部署：虚拟机可以通过镜像实现环境交付的一致性，但镜像分发无法体系化；Docker在Dockerfile中记录了容器构建过程，可在集群中实现快速分发和快速部署；

3.1.1 docker结构介绍



基础设施(Infrastructure)。

主操作系统(Host Operating System)。所有主流的Linux发行版都可以运行Docker。对于MacOS和Windows，也有一些办法"运行"Docker。

Docker守护进程(Docker Daemon)。Docker守护进程取代了Hypervisor，它是运行在操作系统之上的后台进程，负责管理Docker容器。

各种依赖。对于Docker，应用的所有依赖都打包在Docker镜像中，Docker容器是基于Docker镜像创建的。

应用。应用的源代码与它的依赖都打包在Docker镜像中，不同的应用需要不同的Docker镜像。不同的应用运行在不同的Docker容器中，它们是相互隔离的。

Docker守护进程可以直接与主操作系统进行通信，为各个Docker容器分配资源；它还可以将容器与主操作系统隔离，并将各个容器互相隔离。虚拟机启动需要数分钟，而Docker容器可以在数毫秒内启动。由于没有臃肿的从操作系统，Docker可以节省大量的磁盘空间以及其他系统资源；虚拟机更擅长于资源的完全隔离。

链接：<https://blog.51cto.com/14320361/2457143>



在线客服

©著作权归作者所有：来自51CTO博客作者mb5cd21e691f31a的原创作品，如需转载，请注明出处，否则将追究法律责任

docker

kvm

Docker

1

收藏

分享

上一篇：花式安装Docker

下一篇：Docker的基本操作命令



mb5cd21e691f31a

104篇文章，26W+人气，36粉丝

世上从没有白费的努力，也没有碰巧的成功！

关注



提问和评论都可以，用心的回复会被更多人看到和认可


Ctrl+Enter 发布

取消

发布

推荐专栏

更多



VMware vSAN中小企业应用案例


掌握VMware超融合技术

共41章 | 王春海

¥ 51.00

454人订阅

订阅



网工2.0晋级攻略 ——零基础入门Python/Ansible

网络工程师2.0进阶指南

共30章 | 姜汁啤酒

¥ 51.00

2015人订阅

订阅

猜你喜欢

- MySQL的建库、建表、建约束与存储引擎
- Docker(一)：Docker入门教程
- docker安装WordPress-web mysql分布式安装
- “深入浅出”来解读Docker网络核心原理
- Kubernetes(K8S)集群管理Docker容器（部署篇）
- Spring Boot 2.0(四)：使用 Docker 部署 Spring Boot
- 【AWS征文】AWS Lambda 借助 Serverless Framewor...
- k8s集群的三种Web-UI界面部署
- K8s——数据持久化
- K8s之Helm工具详解

- python中Ansible模块的Playbook理解
- keepalived+nginx+docker实现负载均衡高可用服务
- Jenkins与Docker的自动化CI/CD实战
- 企业级Docker镜像仓库Harbor部署与使用
- kvm虚拟化学习笔记(一)之kvm虚拟化环境安装
- 获取Centos7安装Docker各种姿势（指定版本）
- Jenkins+gitlab针对k8s集群实现CI/CD
- K8s——Ingress-nginx原理及配置
- ReplicaSet && DaemonSet 资源对象
- K8S的名称空间创建&&版本的升级、回滚操作

在线客服

好课推荐

更多



Docker虚拟化容器—Docker环境搭建  
988人学习  
**免费试看**



Docker企业应用实战  
12963人学习  
**免费试看**



Docker 网络详解  
702人学习  
**免费试看**



Docker虚拟化容器  
1615人学习  
**免费试看**



在线客服