[Docker 核心技术与实现原理 1](#_Toc1190293198)

[Namespaces 3](#_Toc103705657)

[进程 5](#_Toc1293095466)

[网络 14](#_Toc1851890597)

[Libnetwork 21](#_Toc940640319)

[挂载点 23](#_Toc348469864)

[Chroot 27](#_Toc1338004198)

[CGroups 27](#_Toc1529248673)

[UnionFS 36](#_Toc278074690)

[存储驱动 37](#_Toc997796500)

[AUFS 39](#_Toc329691763)

[其他存储驱动 42](#_Toc1493693817)

[总结 44](#_Toc365109539)

[五个Docker监控工具的对比 44](#_Toc1071829656)

[Docker Stats命令 45](#_Toc189770796)

[CAdvisor 45](#_Toc2062140711)

[Scout 47](#_Toc724638929)

[Data Dog 52](#_Toc1134038523)

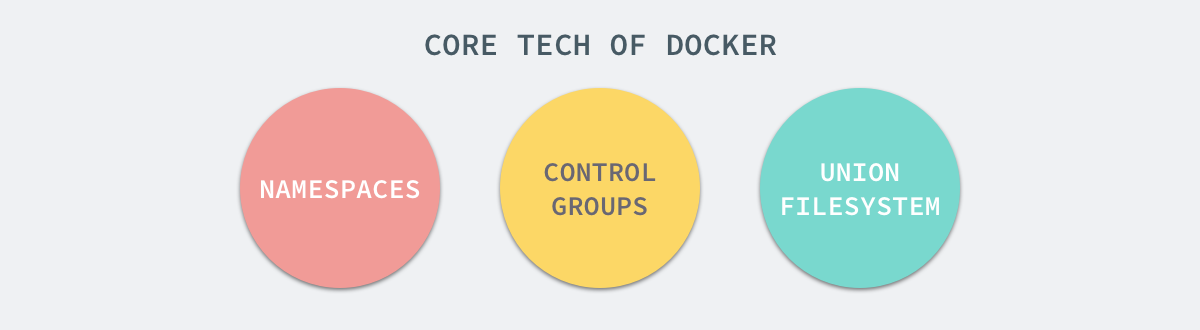
[Sensu Monitoring Framework 57](#_Toc35885328)

# Docker 核心技术与实现原理

提到虚拟化技术，我们首先想到的一定是 Docker，经过四年的快速发展 Docker 已经成为了很多公司的标配，也不再是一个只能在开发阶段使用的玩具了。作为在生产环境中广泛应用的产品，Docker 有着非常成熟的社区以及大量的使用者，代码库中的内容也变得非常庞大。

[](http://dockone.io/uploads/article/20171207/af8f0d8226ac18dae850485d865f7210.png)

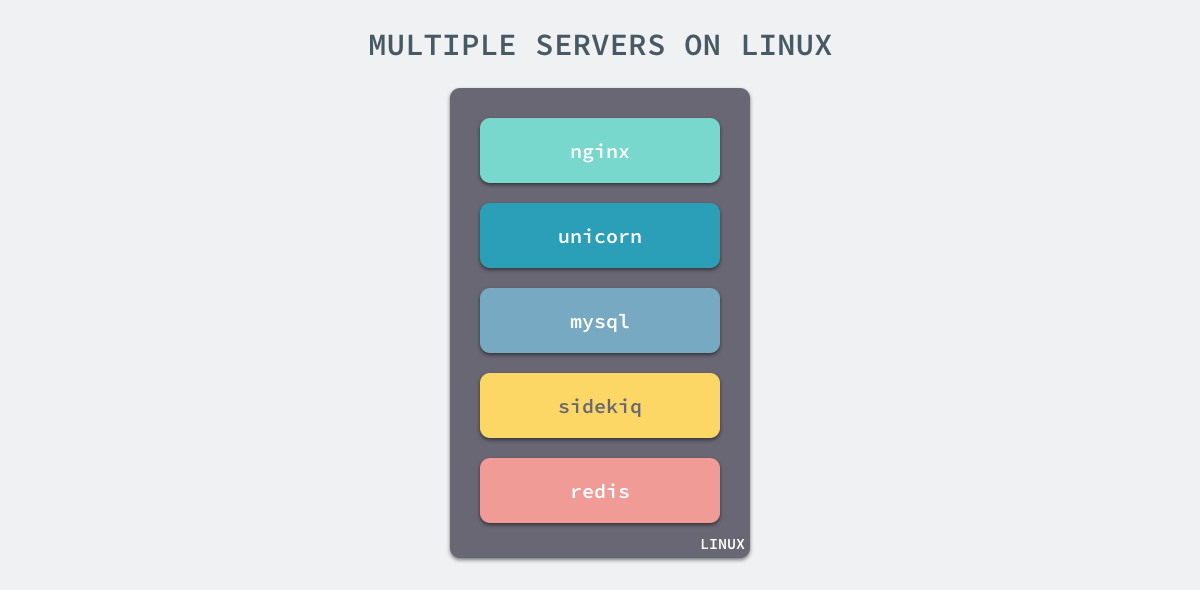
同样，由于项目的发展、功能的拆分以及各种奇怪的改名 PR，让我们再次理解 Docker 的的整体架构变得更加困难。  
  
虽然 Docker 目前的组件较多，并且实现也非常复杂，但是本文不想过多的介绍 Docker 具体的实现细节，我们更想谈一谈 Docker 这种虚拟化技术的出现有哪些核心技术的支撑。

[](http://dockone.io/uploads/article/20171207/8a83b7f6a32a0c2bb47f695744b82167.png)

首先，Docker 的出现一定是因为目前的后端在开发和运维阶段确实需要一种虚拟化技术解决开发环境和生产环境环境一致的问题，通过 Docker 我们可以将程序运行的环境也纳入到版本控制中，排除因为环境造成不同运行结果的可能。但是上述需求虽然推动了虚拟化技术的产生，但是如果没有合适的底层技术支撑，那么我们仍然得不到一个完美的产品。本文剩下的内容会介绍几种 Docker 使用的核心技术，如果我们了解它们的使用方法和原理，就能清楚 Docker 的实现原理。

### Namespaces

命名空间（namespaces）是 Linux 为我们提供的用于分离进程树、网络接口、挂载点以及进程间通信等资源的方法。在日常使用 Linux 或者 macOS 时，我们并没有运行多个完全分离的服务器的需要，但是如果我们在服务器上启动了多个服务，这些服务其实会相互影响的，每一个服务都能看到其他服务的进程，也可以访问宿主机器上的任意文件，这是很多时候我们都不愿意看到的，我们更希望运行在同一台机器上的不同服务能做到完全隔离，就像运行在多台不同的机器上一样。

[](http://dockone.io/uploads/article/20171207/6c9b0c0b64f316a334b1bbb9078b6328.png)

在这种情况下，一旦服务器上的某一个服务被入侵，那么入侵者就能够访问当前机器上的所有服务和文件，这也是我们不想看到的，而 Docker 其实就通过 Linux 的 Namespaces 对不同的容器实现了隔离。  
  
Linux 的命名空间机制提供了以下七种不同的命名空间，包括 CLONE\_NEWCGROUP、CLONE\_NEWIPC、CLONE\_NEWNET、CLONE\_NEWNS、CLONE\_NEWPID、CLONE\_NEWUSER 和 CLONE\_NEWUTS，通过这七个选项我们能在创建新的进程时设置新进程应该在哪些资源上与宿主机器进行隔离。

### 进程

进程是 Linux 以及现在操作系统中非常重要的概念，它表示一个正在执行的程序，也是在现代分时系统中的一个任务单元。在每一个 \*nix 的操作系统上，我们都能够通过 ps 命令打印出当前操作系统中正在执行的进程，比如在 Ubuntu 上，使用该命令就能得到以下的结果：

$ ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD

root         1     0  0 Apr08 ?        00:00:09 /sbin/init

root         2     0  0 Apr08 ?        00:00:00 [kthreadd]

root         3     2  0 Apr08 ?        00:00:05 [ksoftirqd/0]

root         5     2  0 Apr08 ?        00:00:00 [kworker/0:0H]

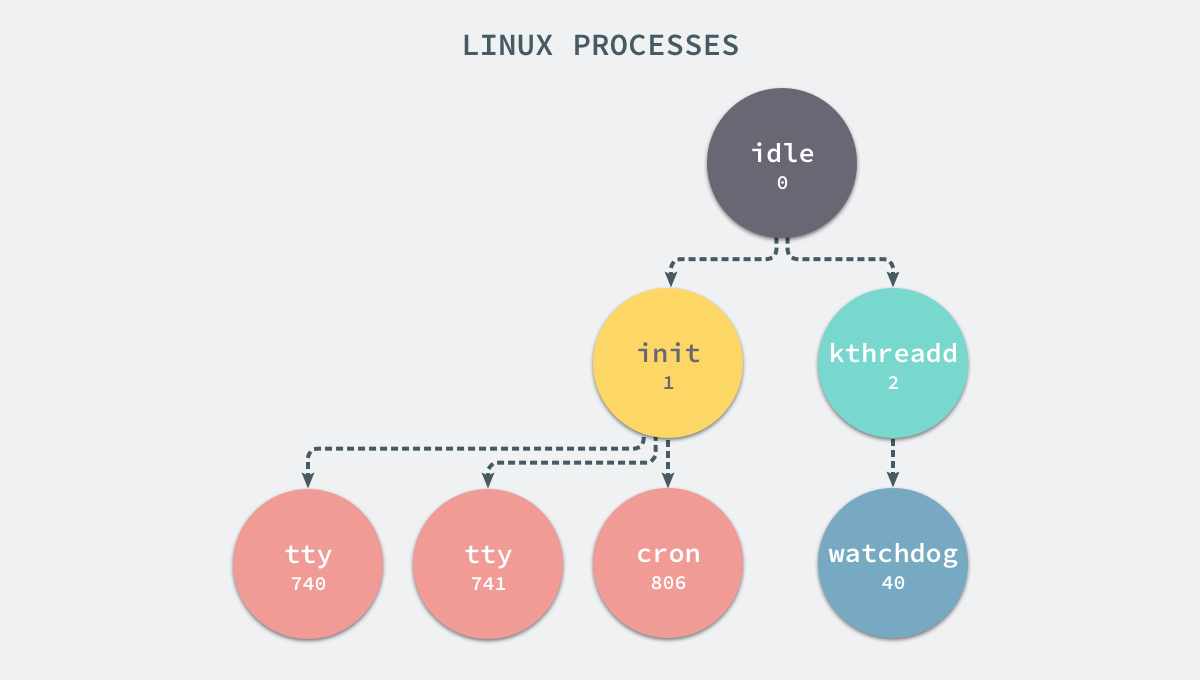
root         7     2  0 Apr08 ?        00:07:10 [rcu\_sched]

root        39     2  0 Apr08 ?        00:00:00 [migration/0]

root        40     2  0 Apr08 ?        00:01:54 [watchdog/0]

...

当前机器上有很多的进程正在执行，在上述进程中有两个非常特殊，一个是 pid 为 1 的 /sbin/init 进程，另一个是 pid 为 2 的 kthreadd 进程，这两个进程都是被 Linux 中的上帝进程 idle 创建出来的，其中前者负责执行内核的一部分初始化工作和系统配置，也会创建一些类似 getty 的注册进程，而后者负责管理和调度其他的内核进程。

[](http://dockone.io/uploads/article/20171207/b82f13c83a0a30a0c68d34f5b3660a14.png)

如果我们在当前的 Linux 操作系统下运行一个新的 Docker 容器，并通过 exec 进入其内部的 bash 并打印其中的全部进程，我们会得到以下的结果：

root@iZ255w13cy6Z:~# docker run -it -d ubuntu

b809a2eb3630e64c581561b08ac46154878ff1c61c6519848b4a29d412215e79

root@iZ255w13cy6Z:~# docker exec -it b809a2eb3630 /bin/bash

root@b809a2eb3630:/# ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD

root         1     0  0 15:42 pts/0    00:00:00 /bin/bash

root         9     0  0 15:42 pts/1    00:00:00 /bin/bash

root        17     9  0 15:43 pts/1    00:00:00 ps -ef

在新的容器内部执行 ps 命令打印出了非常干净的进程列表，只有包含当前 ps -ef 在内的三个进程，在宿主机器上的几十个进程都已经消失不见了。  
  
当前的 Docker 容器成功将容器内的进程与宿主机器中的进程隔离，如果我们在宿主机器上打印当前的全部进程时，会得到下面三条与 Docker 相关的结果：

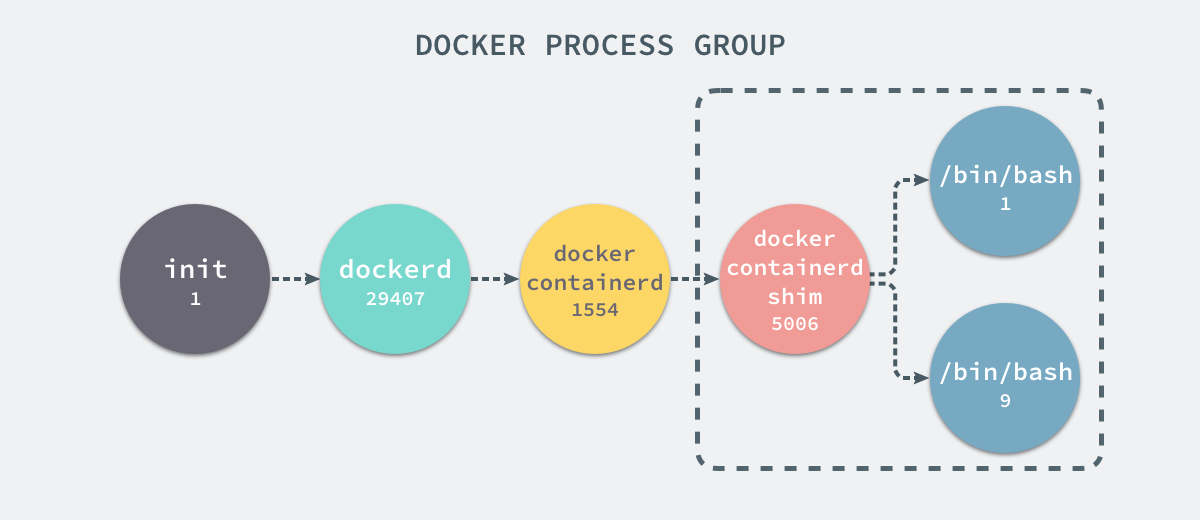
UID        PID  PPID  C STIME TTY          TIME CMD

root     29407     1  0 Nov16 ?        00:08:38 /usr/bin/dockerd --raw-logs

root      1554 29407  0 Nov19 ?        00:03:28 docker-containerd -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --metrics-interval=0 --start-timeout 2m --state-dir /var/run/docker/libcontainerd/containerd --shim docker-containerd-shim --runtime docker-runc

root      5006  1554  0 08:38 ?        00:00:00 docker-containerd-shim b809a2eb3630e64c581561b08ac46154878ff1c61c6519848b4a29d412215e79 /var/run/docker/libcontainerd/b809a2eb3630e64c581561b08ac46154878ff1c61c6519848b4a29d412215e79 docker-runc

在当前的宿主机器上，可能就存在由上述的不同进程构成的进程树：

[](http://dockone.io/uploads/article/20171207/9cd5a6d0d91cfd1d9bfb2318c075bb96.png)

这就是在使用 clone(2) 创建新进程时传入 CLONE\_NEWPID 实现的，也就是使用 Linux 的命名空间实现进程的隔离，Docker 容器内部的任意进程都对宿主机器的进程一无所知。

containerRouter.postContainersStart

└── daemon.ContainerStart

└── daemon.createSpec

    └── setNamespaces

        └── setNamespace

Docker 的容器就是使用上述技术实现与宿主机器的进程隔离，当我们每次运行 docker run 或者 docker start 时，都会在下面的方法中创建一个用于设置进程间隔离的 Spec：

func (daemon \*Daemon) createSpec(c \*container.Container) (\*specs.Spec, error) {

s := oci.DefaultSpec()

// ...

if err := setNamespaces(daemon, &s, c); err != nil {

    return nil, fmt.Errorf("linux spec namespaces: %v", err)

}

return &s, nil

}

在 setNamespaces 方法中不仅会设置进程相关的命名空间，还会设置与用户、网络、IPC 以及 UTS 相关的命名空间：

func setNamespaces(daemon \*Daemon, s \*specs.Spec, c \*container.Container) error {

// user

// network

// ipc

// uts

// pid

if c.HostConfig.PidMode.IsContainer() {

    ns := specs.LinuxNamespace{Type: "pid"}

    pc, err := daemon.getPidContainer(c)

    if err != nil {

        return err

    }

    ns.Path = fmt.Sprintf("/proc/%d/ns/pid", pc.State.GetPID())

    setNamespace(s, ns)

} else if c.HostConfig.PidMode.IsHost() {

    oci.RemoveNamespace(s, specs.LinuxNamespaceType("pid"))

} else {

    ns := specs.LinuxNamespace{Type: "pid"}

    setNamespace(s, ns)

}

return nil

}

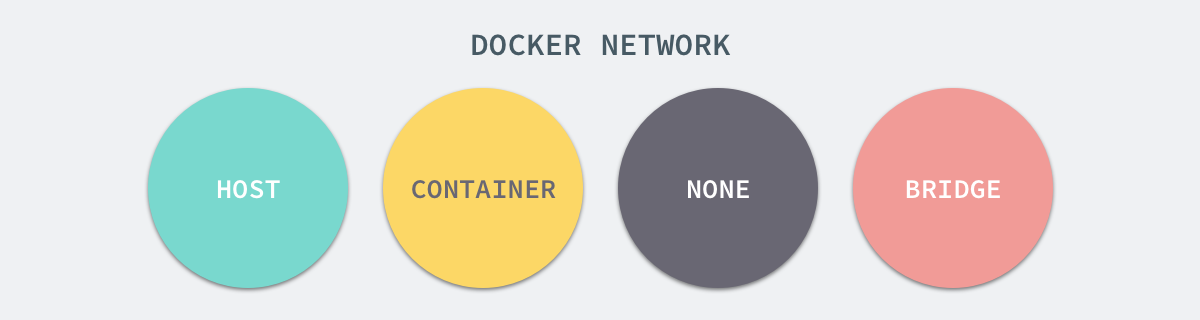
所有命名空间相关的设置 Spec 最后都会作为 Create 函数的入参在创建新的容器时进行设置：

daemon.containerd.Create(context.Background(), container.ID, spec, createOptions)

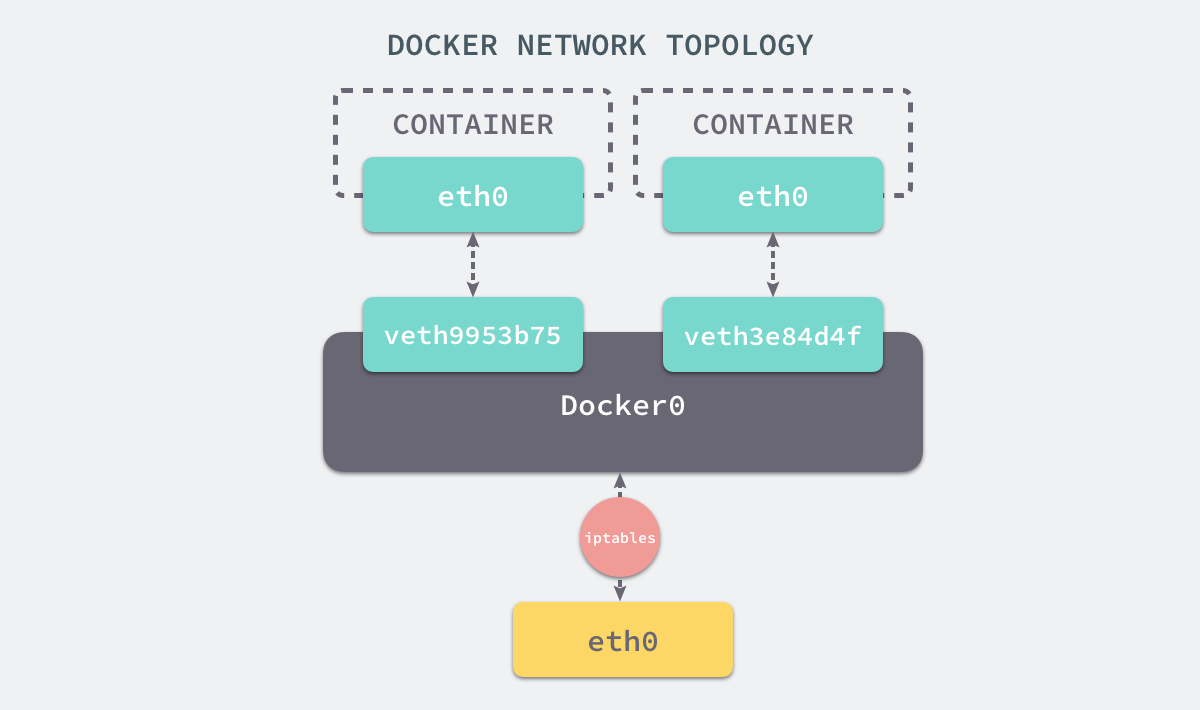
所有与命名空间的相关的设置都是在上述的两个函数中完成的，Docker 通过命名空间成功完成了与宿主机进程和网络的隔离。

### 网络

如果 Docker 的容器通过 Linux 的命名空间完成了与宿主机进程的网络隔离，但是却有没有办法通过宿主机的网络与整个互联网相连，就会产生很多限制，所以 Docker 虽然可以通过命名空间创建一个隔离的网络环境，但是 Docker 中的服务仍然需要与外界相连才能发挥作用。  
每一个使用 docker run 启动的容器其实都具有单独的网络命名空间，Docker 为我们提供了四种不同的网络模式，Host、Container、None 和 Bridge 模式。

[](http://dockone.io/uploads/article/20171207/27e9d4e1b59c07c44f452279831c03e5.png)

在这一部分，我们将介绍 Docker 默认的网络设置模式：网桥模式。在这种模式下，除了分配隔离的网络命名空间之外，Docker 还会为所有的容器设置 IP 地址。当 Docker 服务器在主机上启动之后会创建新的虚拟网桥 docker0，随后在该主机上启动的全部服务在默认情况下都与该网桥相连。

[](http://dockone.io/uploads/article/20171207/7434f774804468e6e74e20e48d2624db.png)

在默认情况下，每一个容器在创建时都会创建一对虚拟网卡，两个虚拟网卡组成了数据的通道，其中一个会放在创建的容器中，会加入到名为 docker0 网桥中。我们可以使用如下的命令来查看当前网桥的接口：

$ brctl show

bridge name bridge id       STP enabled interfaces

docker0     8000.0242a6654980   no      veth3e84d4f

                                    veth9953b75

docker0 会为每一个容器分配一个新的 IP 地址并将 docker0 的 IP 地址设置为默认的网关。网桥 docker0 通过 iptables 中的配置与宿主机器上的网卡相连，所有符合条件的请求都会通过 iptables 转发到 docker0 并由网桥分发给对应的机器。

$ iptables -t nat -L

Chain PREROUTING (policy ACCEPT)

target     prot opt source               destination

DOCKER     all  --  anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain DOCKER (2 references)

target     prot opt source               destination

RETURN     all  --  anywhere             anywhere

我们在当前的机器上使用 docker run -d -p 6379:6379 redis 命令启动了一个新的 Redis 容器，在这之后我们再查看当前 iptables 的 NAT 配置就会看到在 DOCKER 的链中出现了一条新的规则：

DNAT       tcp  --  anywhere             anywhere             tcp dpt:6379 to:192.168.0.4:6379

上述规则会将从任意源发送到当前机器 6379 端口的 TCP 包转发到 192.168.0.4:6379 所在的地址上。  
  
这个地址其实也是 Docker 为 Redis 服务分配的 IP 地址，如果我们在当前机器上直接 ping 这个 IP 地址就会发现它是可以访问到的：

$ ping 192.168.0.4

PING 192.168.0.4 (192.168.0.4) 56(84) bytes of data.

64 bytes from 192.168.0.4: icmp\_seq=1 ttl=64 time=0.069 ms

64 bytes from 192.168.0.4: icmp\_seq=2 ttl=64 time=0.043 ms

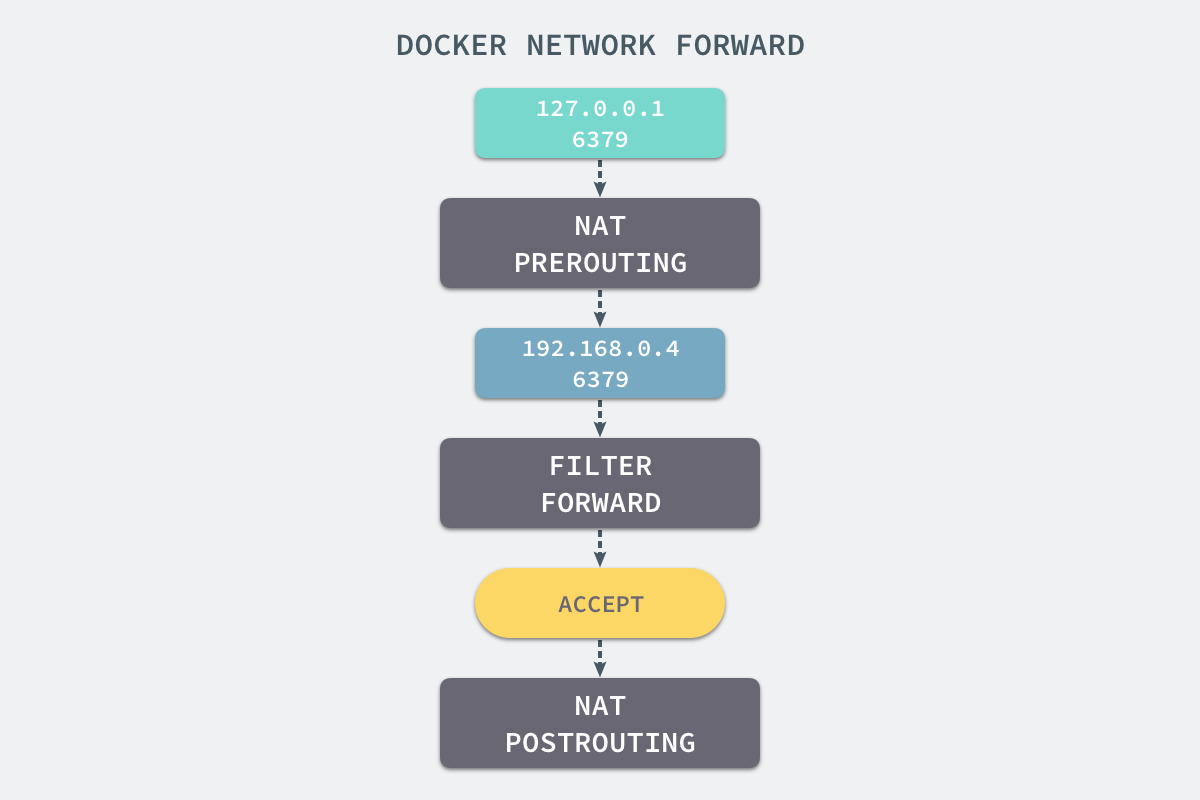
^C

--- 192.168.0.4 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 999ms

rtt min/avg/max/mdev = 0.043/0.056/0.069/0.013 ms

从上述的一系列现象，我们就可以推测出 Docker 是如何将容器的内部的端口暴露出来并对数据包进行转发的了；当有 Docker 的容器需要将服务暴露给宿主机器，就会为容器分配一个 IP 地址，同时向 iptables 中追加一条新的规则。

[](http://dockone.io/uploads/article/20171207/2351ae2fab58876f247f7e3be847ff21.png)

当我们使用 redis-cli 在宿主机器的命令行中访问 127.0.0.1:6379 的地址时，经过 iptables 的 NAT PREROUTING 将 ip 地址定向到了 192.168.0.4，重定向过的数据包就可以通过 iptables 中的 FILTER 配置，最终在 NAT POSTROUTING 阶段将 ip 地址伪装成 127.0.0.1，到这里虽然从外面看起来我们请求的是 127.0.0.1:6379，但是实际上请求的已经是 Docker 容器暴露出的端口了。

$ redis-cli -h 127.0.0.1 -p 6379 ping

PONG

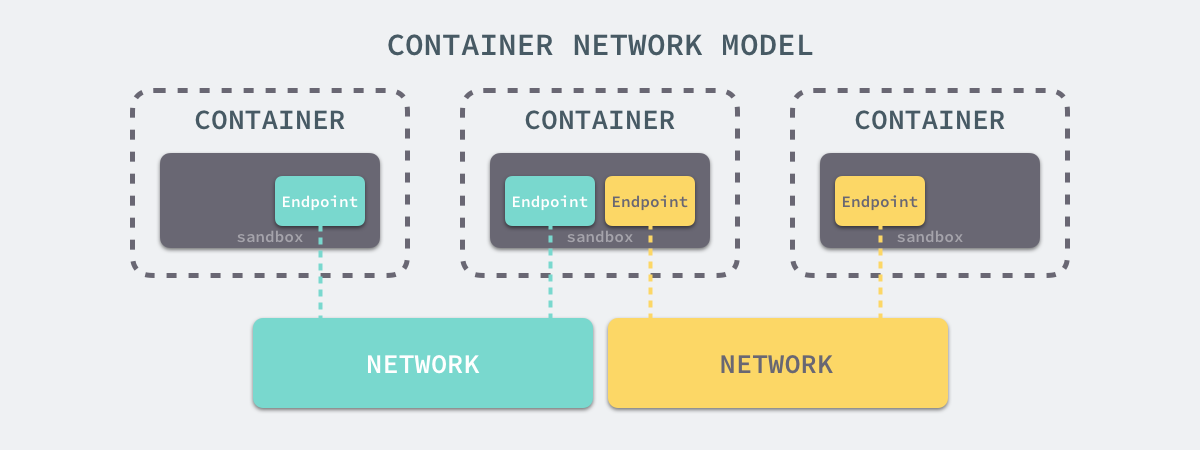
Docker 通过 Linux 的命名空间实现了网络的隔离，又通过 iptables 进行数据包转发，让 Docker 容器能够优雅地为宿主机器或者其他容器提供服务。

### Libnetwork

整个网络部分的功能都是通过 Docker 拆分出来的 libnetwork 实现的，它提供了一个连接不同容器的实现，同时也能够为应用给出一个能够提供一致的编程接口和网络层抽象的容器网络模型。

The goal of libnetwork is to deliver a robust Container Network Model that provides a consistent programming interface and the required network abstractions for applications.

libnetwork 中最重要的概念，容器网络模型由以下的几个主要组件组成，分别是 Sandbox、Endpoint 和 Network：

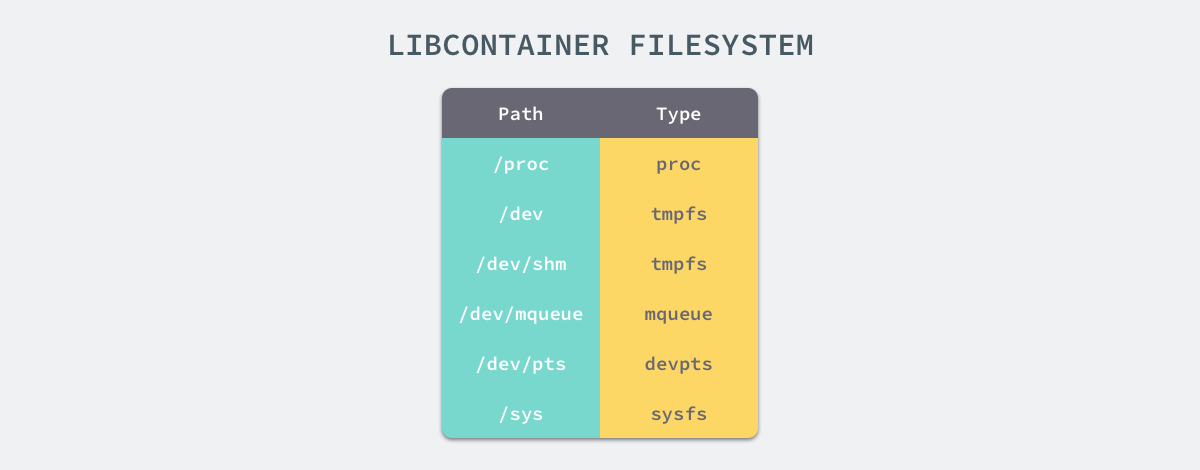
[](http://dockone.io/uploads/article/20171207/2cc24dc3943f97ccd51a926a084d09cb.png)

在容器网络模型中，每一个容器内部都包含一个 Sandbox，其中存储着当前容器的网络栈配置，包括容器的接口、路由表和 DNS 设置，Linux 使用网络命名空间实现这个 Sandbox，每一个 Sandbox 中都可能会有一个或多个 Endpoint，在 Linux 上就是一个虚拟的网卡 veth，Sandbox 通过 Endpoint 加入到对应的网络中，这里的网络可能就是我们在上面提到的 Linux 网桥或者 VLAN。

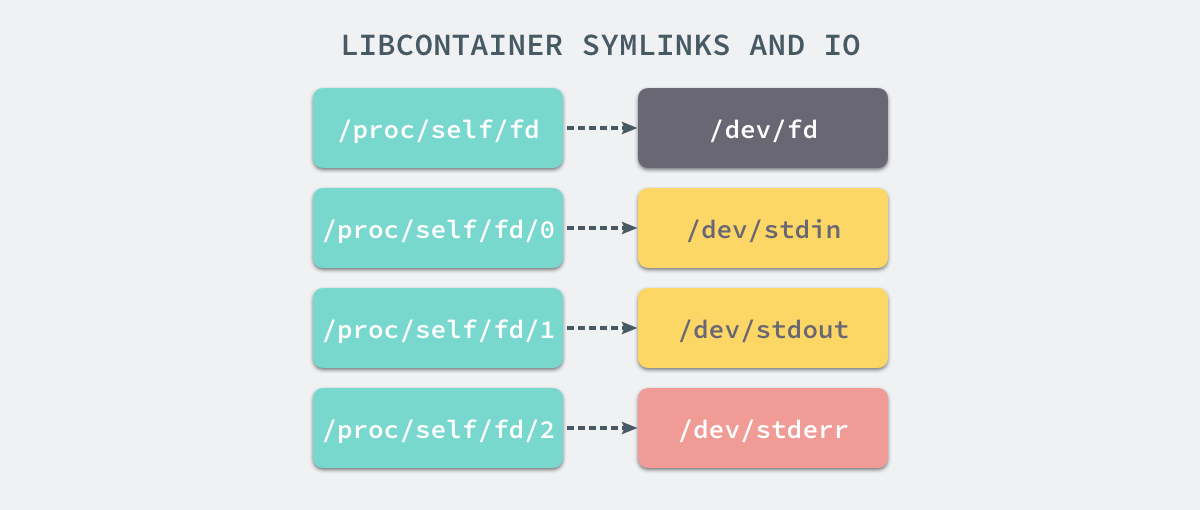
想要获得更多与 libnetwork 或者容器网络模型相关的信息，可以阅读 [Design · libnetwork](https://github.com/docker/libnetwork/blob/master/docs/design.md) 了解更多信息，当然也可以阅读源代码了解不同 OS 对容器网络模型的不同实现。

### 挂载点

虽然我们已经通过 Linux 的命名空间解决了进程和网络隔离的问题，在 Docker 进程中我们已经没有办法访问宿主机器上的其他进程并且限制了网络的访问，但是 Docker 容器中的进程仍然能够访问或者修改宿主机器上的其他目录，这是我们不希望看到的。  
  
在新的进程中创建隔离的挂载点命名空间需要在 clone 函数中传入 CLONE\_NEWNS，这样子进程就能得到父进程挂载点的拷贝，如果不传入这个参数子进程对文件系统的读写都会同步回父进程以及整个主机的文件系统。  
  
如果一个容器需要启动，那么它一定需要提供一个根文件系统（rootfs），容器需要使用这个文件系统来创建一个新的进程，所有二进制的执行都必须在这个根文件系统中。

[](http://dockone.io/uploads/article/20171207/4c72763918c8d8464e8a22261350a370.png)

想要正常启动一个容器就需要在 rootfs 中挂载以上的几个特定的目录，除了上述的几个目录需要挂载之外我们还需要建立一些符号链接保证系统 IO 不会出现问题。

[](http://dockone.io/uploads/article/20171207/d23afdb0e5f64ab4a6c77175fd006e56.png)

为了保证当前的容器进程没有办法访问宿主机器上其他目录，我们在这里还需要通过 libcotainer 提供的 pivor\_root 或者 chroot 函数改变进程能够访问个文件目录的根节点。

// pivor\_root

put\_old = mkdir(...);

pivot\_root(rootfs, put\_old);

chdir("/");

unmount(put\_old, MS\_DETACH);

rmdir(put\_old);

// chroot

mount(rootfs, "/", NULL, MS\_MOVE, NULL);

chroot(".");

chdir("/");

到这里我们就将容器需要的目录挂载到了容器中，同时也禁止当前的容器进程访问宿主机器上的其他目录，保证了不同文件系统的隔离。

这一部分的内容是作者在 libcontainer 中的 [SPEC.md](https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md) 文件中找到的，其中包含了 Docker 使用的文件系统的说明，对于 Docker 是否真的使用 chroot 来确保当前的进程无法访问宿主机器的目录，作者其实也没有确切的答案，一是 Docker 项目的代码太多庞大，不知道该从何入手，作者尝试通过 Google 查找相关的结果，但是既找到了无人回答的[问题](https://forums.docker.com/t/does-the-docker-engine-use-chroot/25429)，也得到了与 SPEC 中的描述有冲突的[答案](https://www.quora.com/Do-Docker-containers-use-a-chroot-environment)，如果各位读者有明确的答案可以在博客下面留言，非常感谢。

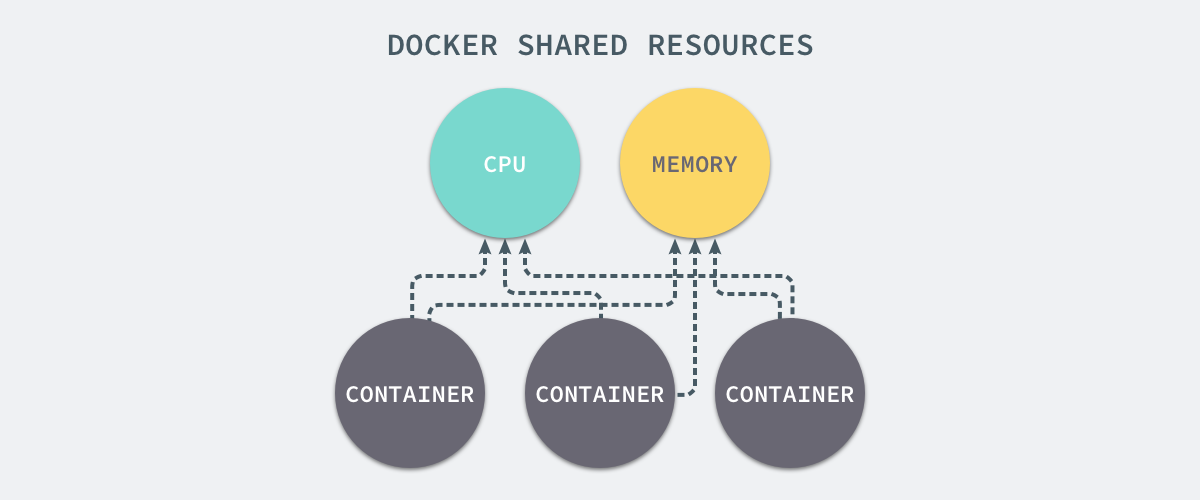
### Chroot

在这里不得不简单介绍一下 chroot（change root），在 Linux 系统中，系统默认的目录就都是以 / 也就是根目录开头的，chroot 的使用能够改变当前的系统根目录结构，通过改变当前系统的根目录，我们能够限制用户的权利，在新的根目录下并不能够访问旧系统根目录的结构个文件，也就建立了一个与原系统完全隔离的目录结构。

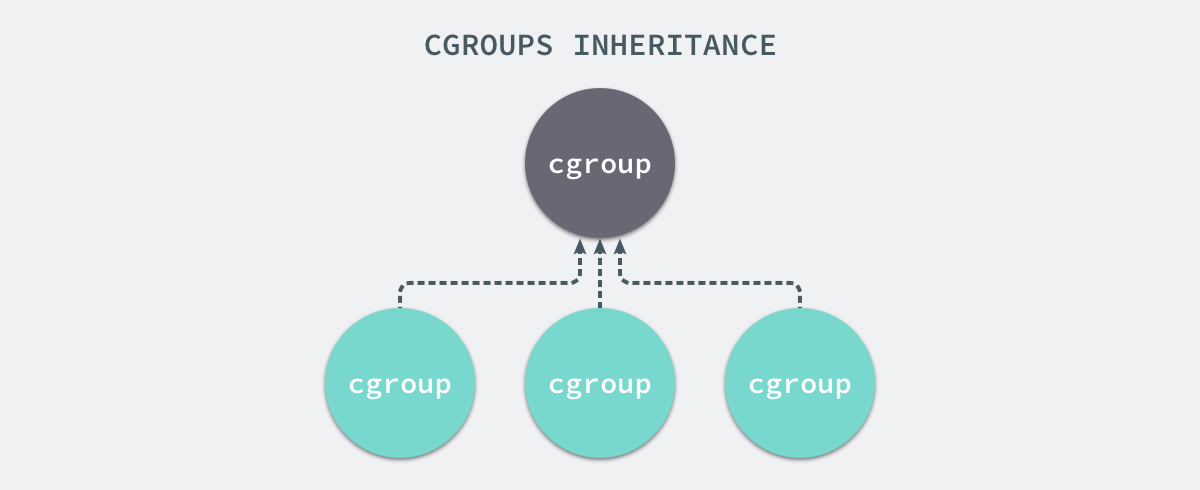
与 chroot 的相关内容部分来自《[理解 chroot](https://www.ibm.com/developerworks/cn/linux/l-cn-chroot/index.html)》一文，各位读者可以阅读这篇文章获得更详细的信息。

### CGroups

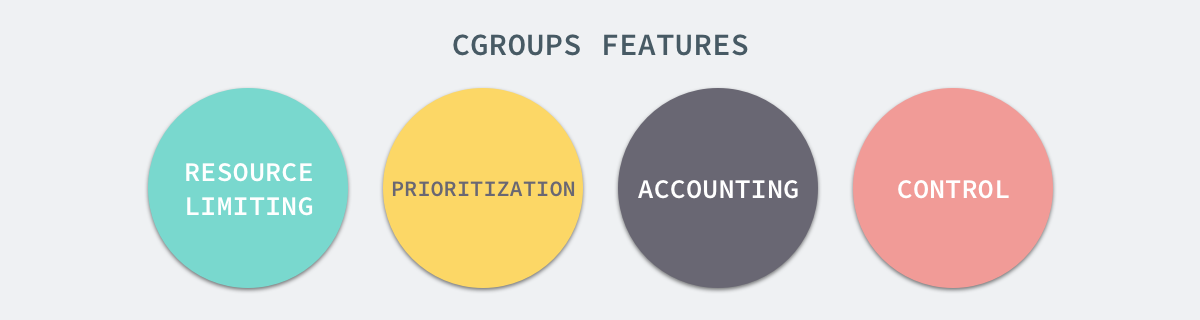
我们通过 Linux 的命名空间为新创建的进程隔离了文件系统、网络并与宿主机器之间的进程相互隔离，但是命名空间并不能够为我们提供物理资源上的隔离，比如 CPU 或者内存，如果在同一台机器上运行了多个对彼此以及宿主机器一无所知的『容器』，这些容器却共同占用了宿主机器的物理资源。

[](http://dockone.io/uploads/article/20171207/c9e83d7d7fce13df25b1be728dbc0deb.png)

如果其中的某一个容器正在执行 CPU 密集型的任务，那么就会影响其他容器中任务的性能与执行效率，导致多个容器相互影响并且抢占资源。如何对多个容器的资源使用进行限制就成了解决进程虚拟资源隔离之后的主要问题，而 Control Groups（简称 CGroups）就是能够隔离宿主机器上的物理资源，例如 CPU、内存、磁盘 I/O 和网络带宽。  
  
每一个 CGroup 都是一组被相同的标准和参数限制的进程，不同的 CGroup 之间是有层级关系的，也就是说它们之间可以从父类继承一些用于限制资源使用的标准和参数。

[](http://dockone.io/uploads/article/20171207/d9d1d7fe30f5fe14bf5b9ac337bb7c06.png)

Linux 的 CGroup 能够为一组进程分配资源，也就是我们在上面提到的 CPU、内存、网络带宽等资源，通过对资源的分配，CGroup 能够提供以下的几种功能：

[](http://dockone.io/uploads/article/20171207/18627032d48931c7184714b800c3bcb5.png)

在 CGroup 中，所有的任务就是一个系统的一个进程，而 CGroup 就是一组按照某种标准划分的进程，在 CGroup 这种机制中，所有的资源控制都是以 CGroup 作为单位实现的，每一个进程都可以随时加入一个 CGroup 也可以随时退出一个 CGroup。  
——[CGroup 介绍、应用实例及原理描述](https://www.ibm.com/developerworks/cn/linux/1506_cgroup/index.html)

Linux 使用文件系统来实现 CGroup，我们可以直接使用下面的命令查看当前的 CGroup 中有哪些子系统：

$ lssubsys -m

cpuset /sys/fs/cgroup/cpuset

cpu /sys/fs/cgroup/cpu

cpuacct /sys/fs/cgroup/cpuacct

memory /sys/fs/cgroup/memory

devices /sys/fs/cgroup/devices

freezer /sys/fs/cgroup/freezer

blkio /sys/fs/cgroup/blkio

perf\_event /sys/fs/cgroup/perf\_event

hugetlb /sys/fs/cgroup/hugetlb

大多数 Linux 的发行版都有着非常相似的子系统，而之所以将上面的 cpuset、cpu 等东西称作子系统，是因为它们能够为对应的控制组分配资源并限制资源的使用。  
  
如果我们想要创建一个新的 cgroup 只需要在想要分配或者限制资源的子系统下面创建一个新的文件夹，然后这个文件夹下就会自动出现很多的内容，如果你在 Linux 上安装了 Docker，你就会发现所有子系统的目录下都有一个名为 Docker 的文件夹：

$ ls cpu

cgroup.clone\_children

...

cpu.stat

docker

notify\_on\_release

release\_agent

tasks

$ ls cpu/docker/

9c3057f1291b53fd54a3d12023d2644efe6a7db6ddf330436ae73ac92d401cf1

cgroup.clone\_children

...

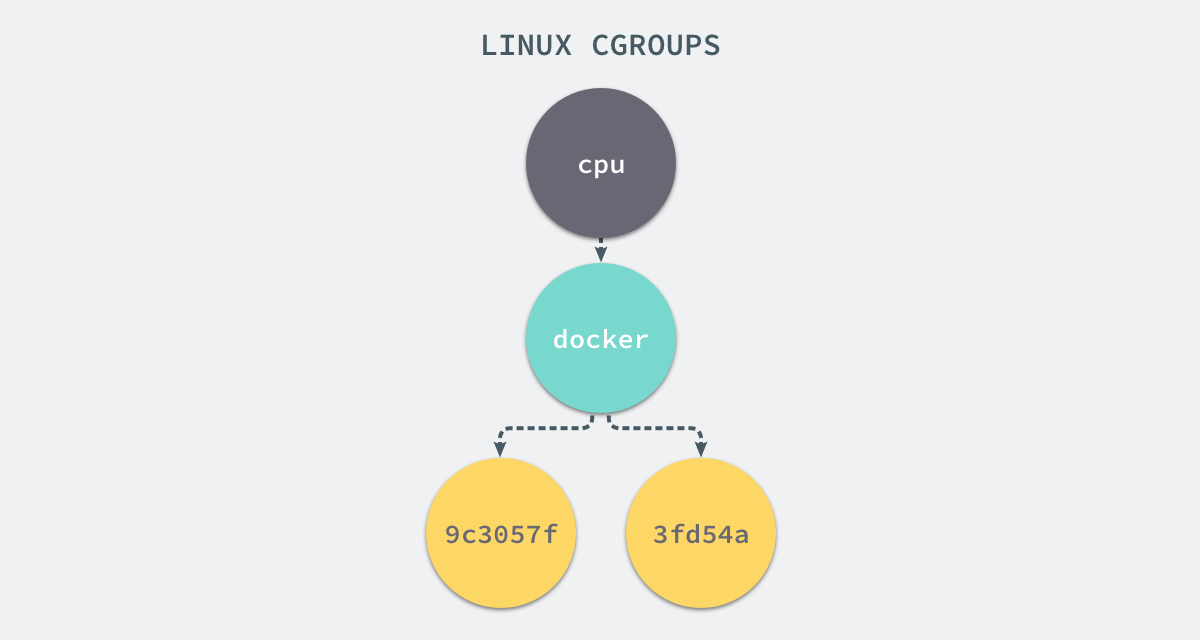
cpu.stat

notify\_on\_release

release\_agent

tasks

9c3057xxx 其实就是我们运行的一个 Docker 容器，启动这个容器时，Docker 会为这个容器创建一个与容器标识符相同的 CGroup，在当前的主机上 CGroup 就会有以下的层级关系：

[](http://dockone.io/uploads/article/20171207/82fcf20839d65f7c59df4f92ed3b18de.png)

每一个 CGroup 下面都有一个 tasks 文件，其中存储着属于当前控制组的所有进程的 pid，作为负责 cpu 的子系统，cpu.cfs\_quota\_us 文件中的内容能够对 CPU 的使用作出限制，如果当前文件的内容为 50000，那么当前控制组中的全部进程的 CPU 占用率不能超过 50%。  
  
如果系统管理员想要控制 Docker 某个容器的资源使用率就可以在 docker 这个父控制组下面找到对应的子控制组并且改变它们对应文件的内容，当然我们也可以直接在程序运行时就使用参数，让 Docker 进程去改变相应文件中的内容。

$ docker run -it -d --cpu-quota=50000 busybox

53861305258ecdd7f5d2a3240af694aec9adb91cd4c7e210b757f71153cdd274

$ cd 53861305258ecdd7f5d2a3240af694aec9adb91cd4c7e210b757f71153cdd274/

$ ls

cgroup.clone\_children  cgroup.event\_control  cgroup.procs  cpu.cfs\_period\_us  cpu.cfs\_quota\_us  cpu.shares  cpu.stat  notify\_on\_release  tasks

$ cat cpu.cfs\_quota\_us

50000

当我们使用 Docker 关闭掉正在运行的容器时，Docker 的子控制组对应的文件夹也会被 Docker 进程移除，Docker 在使用 CGroup 时其实也只是做了一些创建文件夹改变文件内容的文件操作，不过 CGroup 的使用也确实解决了我们限制子容器资源占用的问题，系统管理员能够为多个容器合理的分配资源并且不会出现多个容器互相抢占资源的问题。

### UnionFS

Linux 的命名空间和控制组分别解决了不同资源隔离的问题，前者解决了进程、网络以及文件系统的隔离，后者实现了 CPU、内存等资源的隔离，但是在 Docker 中还有另一个非常重要的问题需要解决 - 也就是镜像。  
  
镜像到底是什么，它又是如何组成和组织的是作者使用 Docker 以来的一段时间内一直比较让作者感到困惑的问题，我们可以使用 docker run 非常轻松地从远程下载 Docker 的镜像并在本地运行。  
  
Docker 镜像其实本质就是一个压缩包，我们可以使用下面的命令将一个 Docker 镜像中的文件导出：

$ docker export $(docker create busybox) | tar -C rootfs -xvf -

$ ls

bin  dev  etc  home proc root sys  tmp  usr  var

你可以看到这个 busybox 镜像中的目录结构与 Linux 操作系统的根目录中的内容并没有太多的区别，可以说 Docker 镜像就是一个文件。

### 存储驱动

Docker 使用了一系列不同的存储驱动管理镜像内的文件系统并运行容器，这些存储驱动与 Docker 卷（volume）有些不同，存储引擎管理着能够在多个容器之间共享的存储。  
  
想要理解 Docker 使用的存储驱动，我们首先需要理解 Docker 是如何构建并且存储镜像的，也需要明白 Docker 的镜像是如何被每一个容器所使用的；Docker 中的每一个镜像都是由一系列只读的层组成的，Dockerfile 中的每一个命令都会在已有的只读层上创建一个新的层：

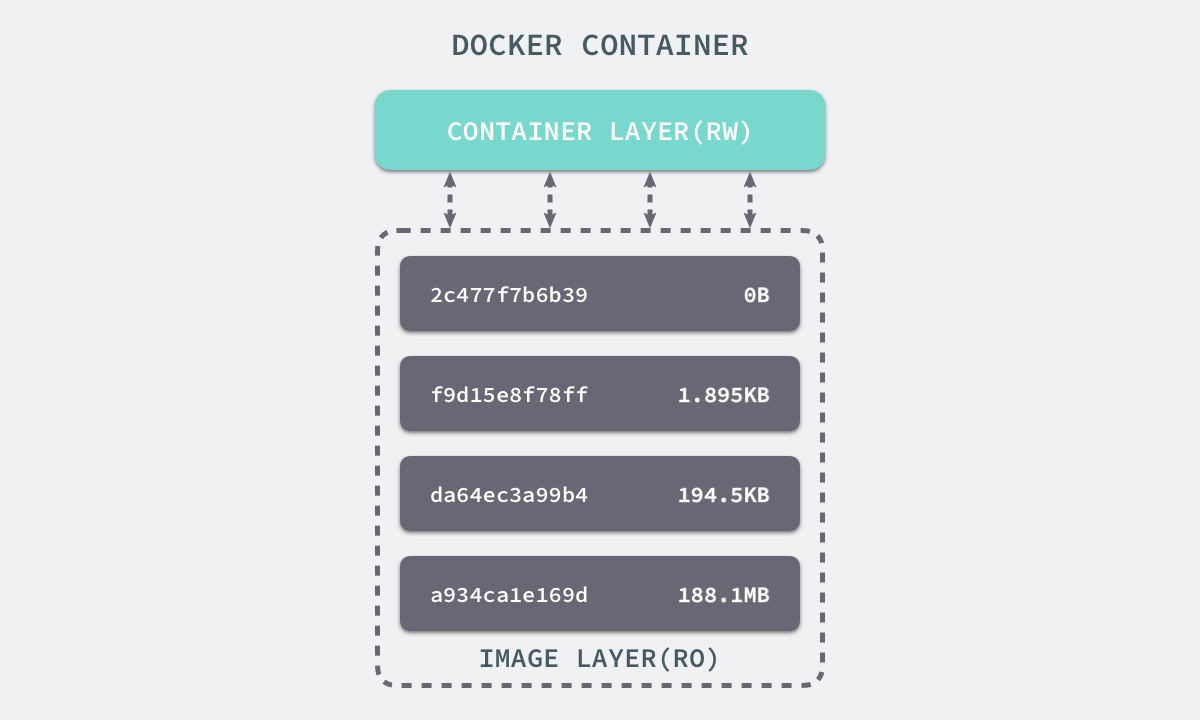
FROM ubuntu:15.04

COPY . /app

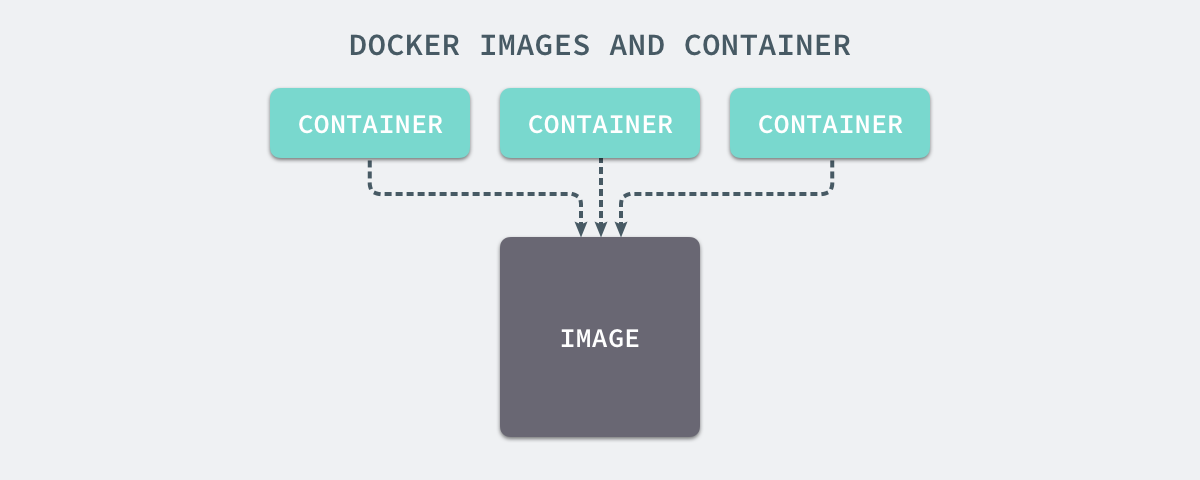
RUN make /app

CMD python /app/app.py

容器中的每一层都只对当前容器进行了非常小的修改，上述的 Dockerfile 文件会构建一个拥有四层 layer 的镜像：

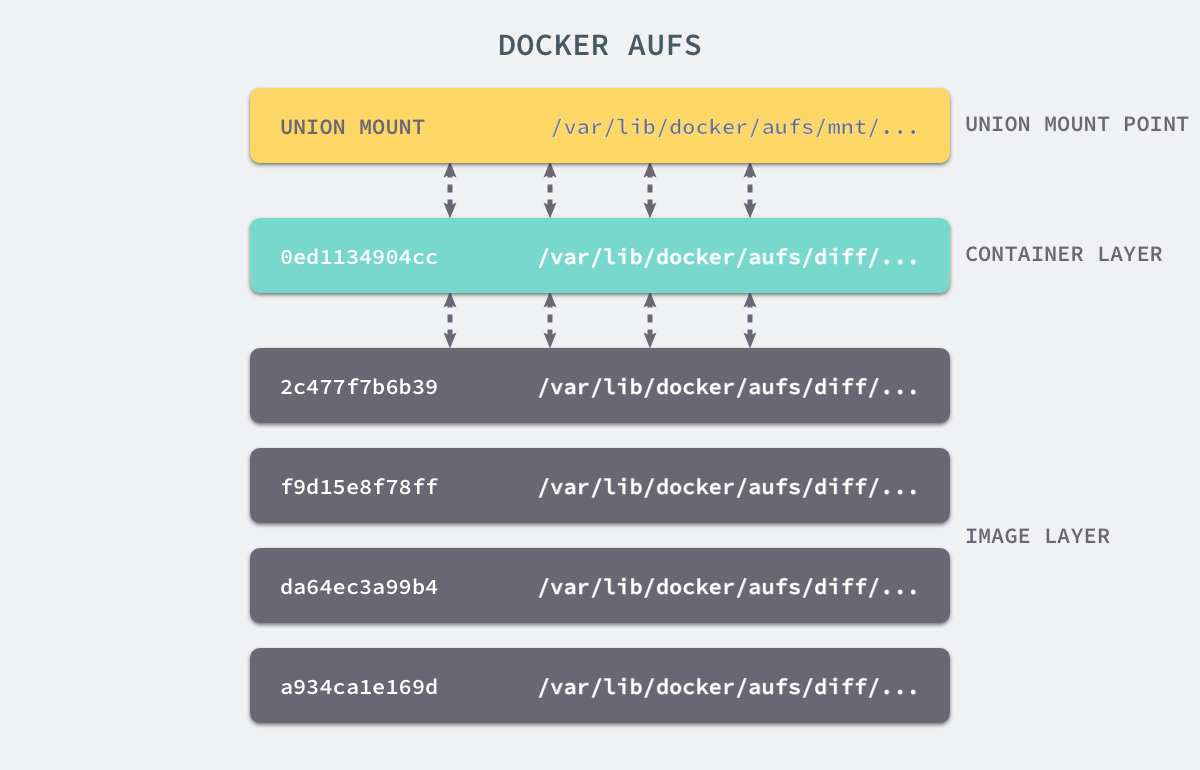
[](http://dockone.io/uploads/article/20171207/04cf050b960367caaf1397b1f4e33583.png)

当镜像被 docker run 命令创建时就会在镜像的最上层添加一个可写的层，也就是容器层，所有对于运行时容器的修改其实都是对这个容器读写层的修改。  
  
容器和镜像的区别就在于，所有的镜像都是只读的，而每一个容器其实等于镜像加上一个可读写的层，也就是同一个镜像可以对应多个容器。

[](http://dockone.io/uploads/article/20171207/6090646fdca9e68b221a2798bcb321cb.png)

### AUFS

UnionFS 其实是一种为 Linux 操作系统设计的用于把多个文件系统『联合』到同一个挂载点的文件系统服务。而 AUFS 即 Advanced UnionFS 其实就是 UnionFS 的升级版，它能够提供更优秀的性能和效率。  
  
AUFS 作为联合文件系统，它能够将不同文件夹中的层联合（Union）到了同一个文件夹中，这些文件夹在 AUFS 中称作分支，整个『联合』的过程被称为联合挂载（Union Mount）：

[](http://dockone.io/uploads/article/20171207/99b9ef20e7266a86e18d684db8fa3fee.png)

每一个镜像层或者容器层都是 /var/lib/docker/ 目录下的一个子文件夹；在 Docker 中，所有镜像层和容器层的内容都存储在 /var/lib/docker/aufs/diff/ 目录中：

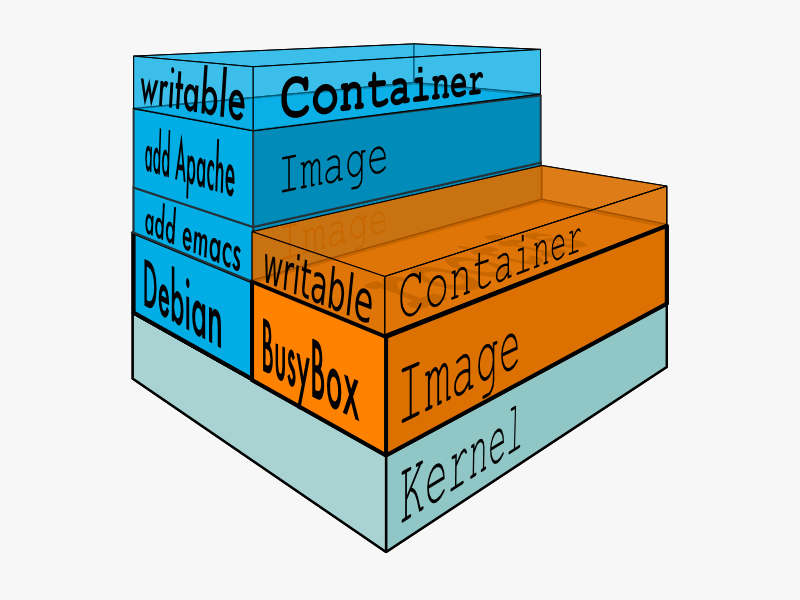
$ ls /var/lib/docker/aufs/diff/00adcccc1a55a36a610a6ebb3e07cc35577f2f5a3b671be3dbc0e74db9ca691c       93604f232a831b22aeb372d5b11af8c8779feb96590a6dc36a80140e38e764d8

00adcccc1a55a36a610a6ebb3e07cc35577f2f5a3b671be3dbc0e74db9ca691c-init  93604f232a831b22aeb372d5b11af8c8779feb96590a6dc36a80140e38e764d8-init

019a8283e2ff6fca8d0a07884c78b41662979f848190f0658813bb6a9a464a90       93b06191602b7934fafc984fbacae02911b579769d0debd89cf2a032e7f35cfa

...

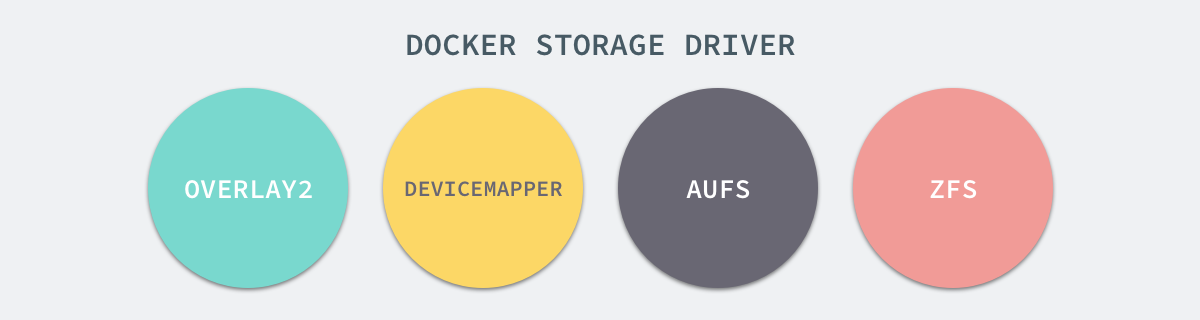
而 /var/lib/docker/aufs/layers/ 中存储着镜像层的元数据，每一个文件都保存着镜像层的元数据，最后的 /var/lib/docker/aufs/mnt/ 包含镜像或者容器层的挂载点，最终会被 Docker 通过联合的方式进行组装。

[](http://dockone.io/uploads/article/20171207/7daece208870c165deb6fac7fdb194d1.png)

上面的这张图片非常好的展示了组装的过程，每一个镜像层都是建立在另一个镜像层之上的，同时所有的镜像层都是只读的，只有每个容器最顶层的容器层才可以被用户直接读写，所有的容器都建立在一些底层服务（Kernel）上，包括命名空间、控制组、rootfs 等等，这种容器的组装方式提供了非常大的灵活性，只读的镜像层通过共享也能够减少磁盘的占用。

### 其他存储驱动

AUFS 只是 Docker 使用的存储驱动的一种，除了 AUFS 之外，Docker 还支持了不同的存储驱动，包括 aufs、devicemapper、overlay2、zfs 和 vfs 等等，在最新的 Docker 中，overlay2 取代了 aufs 成为了推荐的存储驱动，但是在没有 overlay2 驱动的机器上仍然会使用 aufs 作为 Docker 的默认驱动。

[](http://dockone.io/uploads/article/20171207/2873e796186363e978dee57baf167ff9.png)

不同的存储驱动在存储镜像和容器文件时也有着完全不同的实现，有兴趣的读者可以在 Docker 的官方文档 Select a storage driver 中找到相应的内容。  
  
想要查看当前系统的 Docker 上使用了哪种存储驱动只需要使用以下的命令就能得到相对应的信息：

$ docker info | grep Storage

Storage Driver: aufs

作者的这台 Ubuntu 上由于没有 overlay2 存储驱动，所以使用 aufs 作为 Docker 的默认存储驱动。

### 总结

Docker 目前已经成为了非常主流的技术，已经在很多成熟公司的生产环境中使用，但是 Docker 的核心技术其实已经有很多年的历史了，Linux 命名空间、控制组和 UnionFS 三大技术支撑了目前 Docker 的实现，也是 Docker 能够出现的最重要原因。  
  
作者在学习 Docker 实现原理的过程中查阅了非常多的资料，从中也学习到了很多与 Linux 操作系统相关的知识，不过由于 Docker 目前的代码库实在是太过庞大，想要从源代码的角度完全理解 Docker 实现的细节已经是非常困难的了，但是如果各位读者真的对其实现细节感兴趣，可以从 Docker CE 的源代码开始了解 Docker 的原理。

# 五个Docker监控工具的对比

【编者的话】这篇文章作者是Usman，他是服务器和基础架构工程师，有非常丰富的分布式构建经验。该篇文章主要分析评估了五种Docker监控工具，包括免费的和不免费的：Docker Stats、CAdvisor、Scout、Data Dog以及Sensu。不过作者还是推荐使用Data Dog。另外还有两个工具：Prometheus与Sysdig Cloud会在下一篇做介绍分析，敬请期待。  
  
随着Docker被大规模的部署应用，如何通过可视化的方式了解Docker环境的状态以及健康变得越来越重要。这篇文章我们来回顾下监控容器的常用工具。我会基于以下标准评估这些工具：

易于部署

信息呈现的详细度

整个部署过程中日志的聚集程度

数据报警能力

是否可以监控非Docker的资源

成本

这些评估标准可能并不全面，但是我试图强调的是最常用的工具以及优化此六项评估标准的工具。

## Docker Stats命令

本文中所有使用的命令只在亚马逊EC2上的RancherOS实例中测试过。但是我想它们应该可以在任何的Docker容器中运行。  
  
我将讨论的第一个工具是Docker本身。你可能不知道Docker客户端已经提供了基本的命令行工具来检查容器的资源消耗。想要查看容器统计信息只需运行docker stats [CONTAINER\_NAME]。这样就可以查看每个容器的CPU利用率、内存的使用量以及可用内存总量。请注意，如果你没有限制容器内存，那么该命令将显示您的主机的内存总量。但它并不意味着你的每个容器都能访问那么多的内存。另外，还可以看啊都容器通过网络发送和接收的数据总量。

$ docker stats determined\_shockley determined\_wozniak prickly\_hypatia

CONTAINER             CPU %               MEM USAGE/LIMIT       MEM %               NET I/O

determined\_shockley   0.00%               884 KiB/1.961 GiB     0.04%               648 B/648 B

determined\_wozniak    0.00%               1.723 MiB/1.961 GiB   0.09%               1.266 KiB/648 B

prickly\_hypatia       0.00%               740 KiB/1.961 GiB     0.04%               1.898 KiB/648 B

如果想要看到更为详细的容器属性，还可以通过netcat，使用Docker远程API来查看（见下文）。发送一个HTTP GET请求/containers/[CONTAINER\_NAME]，其中CONTAINER\_NAME是你想要统计的容器名称。你可以从[这里](https://gist.github.com/usmanismail/0c4922ffec4a0220d385)看到一个容器stats请求的完整响应信息。在上述的例子中你会得到缓存、交换空间以及内存的详细信息。如果要了解什么是metrics，那么你就需要精读Docker文档的[Run Metrics部分](https://docs.docker.com/articles/runmetrics/)。

评分：

1. 易于部署程度：※※※※※  
2. 信息详细程度：※※※※※  
3. 集成度：无  
4. 生成警报的能力：无  
5. 监测非Docker的资源的能力：无  
6. 成本：免费

## CAdvisor

我们可以使用docker stats命令和远程API来获取容器的状态信息。但是，如果你想要在图形界面中直接查看这些信息，那你就需要诸如[CAdvisor](https://github.com/google/cadvisor)这类的工具。CAdvisor提供了早docker stats命令所显示的数据的可视化界面。运行以下Docker命令，并在浏览器里访问http://&lt;your-hostname>:8080/可以看到CAdvisor的界面。你将看到CPU的使用率、内存使用率、网络吞吐量以及磁盘空间利用率。然后，你可以通过点击在网页顶部的Docker Containers链接，然后选择某个容器来详细了解它的使用情况。

docker run                                      \

--volume=/:/rootfs:ro                         \

--volume=/var/run:/var/run:rw                 \

--volume=/sys:/sys:ro                         \

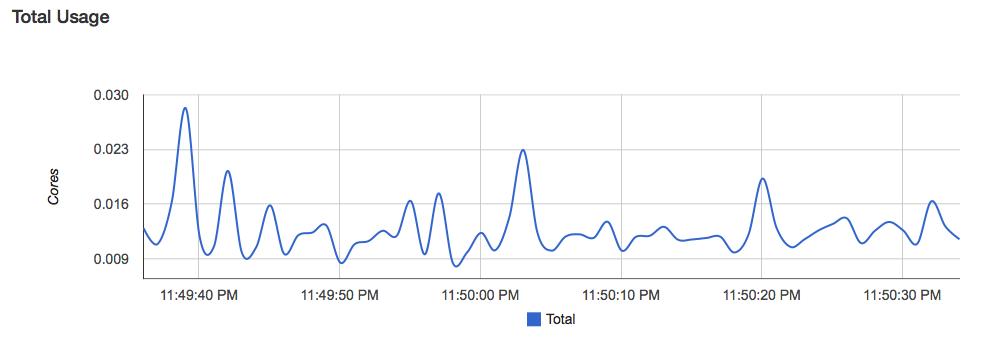
--volume=/var/lib/docker/:/var/lib/docker:ro  \

--publish=8080:8080                           \

--detach=true                                 \

--name=cadvisor                               \

google/cadvisor:latest

[](http://dockone.io/uploads/article/20150525/35c654a1c1f880702fe3bea6d9fd5e90.png)

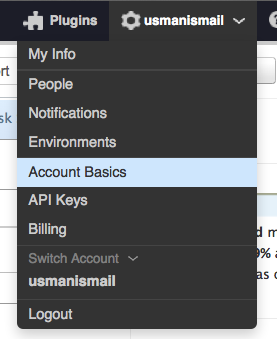
CAdvisor是一个易于设置并且非常有用的工具，我们不用非要SSH到服务器才能查看资源消耗，而且它还给我们生成了图表。此外，当集群需要额外的资源时，压力表（pressure gauges ）提供了快速预览。而且，与本文中的其他的工具不一样的是CAdvisor是免费的，并且还开源。另外，它的资源消耗也比较低。但是，它有它的局限性，它只能监控一个Docker主机。因此，如果你是多节点的话，那就比较麻烦了，你得在所有的主机上都安装一个CAdvisor，者肯定特别不方便。值得注意的是，如果你使用的是Kubernetes，你可以使用[heapster](https://github.com/GoogleCloudPlatform/heapster)来监控多节点集群。另外，在图表中的数据仅仅是时长一分钟的移动窗口，并没有方法来查看长期趋势。如果资源使用率在危险水平，它却没有生成警告的机制。如果在Docker节点的资源消耗方面，你没有任何可视化界面，那么CAdvisor是一个不错的开端来带你步入容器监控，然而如果你打算在你的容器中运行任何关键任务，那你就需要一个更强大的工具或者方法。

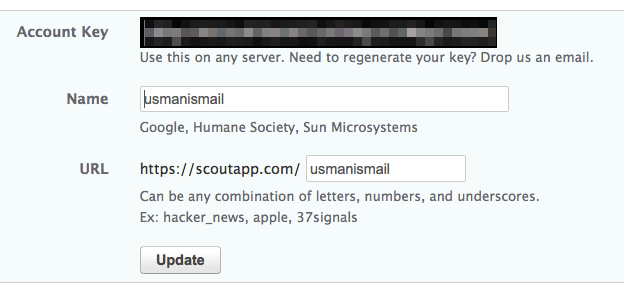
评分：（忽略了heapster，因为它仅支持Kubernetes）

1. 易于部署程度：※※※※※  
2. 信息详细程度：※※  
3. 集成度：※  
4. 生成警报的能力：无  
5. 监测非Docker的资源的能力：无  
6. 成本：免费

## Scout

下一个Docker监控的方法是Scout，它解决了CAdvisor的局限性。 Scout是一个应用监控服务，它能够从很多主机和容器中获得各项监测数据，并将数据呈现在有更长时间尺度的图标中。它也可以基于这些指标生成警报。要获取Scout并运行，第一步，在[scoutapp.com](https://scoutapp.com/)注册一个Scout帐户，免费的试用账号足以用来集成测试。一旦你创建了自己的帐户并登录，点击右上角的帐户名称，然后点击Account Basics来查看你的Account Key，你需要这个Key从Docker服务器来发送指标。

[](http://dockone.io/uploads/article/20150525/7cf7b0501d2fe19e84235bf3059b0d07.png)

[](http://dockone.io/uploads/article/20150525/d1441536f4084971652ac5acf8fcdd00.png)

现在在你的主机上，创建一个名为scouts.yml的文件并将下面的文字复制到该文件中，用上边得到的Key替换到account\_key。您可以对主机指定任何有意义的变量：display\_name、environment与roles等属性。当他们在scout界面上呈现时，这些将用于分离各种指标。我假设有一组网站服务器列表正在运行Docker，它们都将采用如下图所示的变量。

# account\_key is the only required value

account\_key: YOUR\_ACCOUNT\_KEY

hostname: web01-host

display\_name: web01

environment: production

roles: web

现在，你可以使用scout配置文件通过Docker-scout插件来运行scout。

docker run -d  --name scout-agent                              \

-v /proc:/host/proc:ro                                               \

-v /etc/mtab:/host/etc/mtab:ro                                   \

-v /var/run/docker.sock:/host/var/run/docker.sock:ro    \

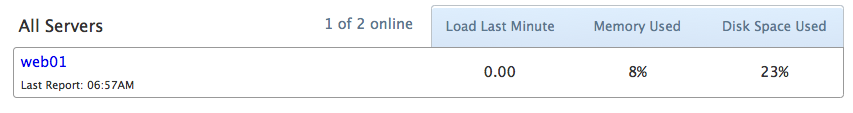
-v `pwd`/scoutd.yml:/etc/scout/scoutd.yml                     \

-v /sys/fs/cgroup/:/host/sys/fs/cgroup/                           \

--net=host --privileged                                                   \

soutapp/docker-scout

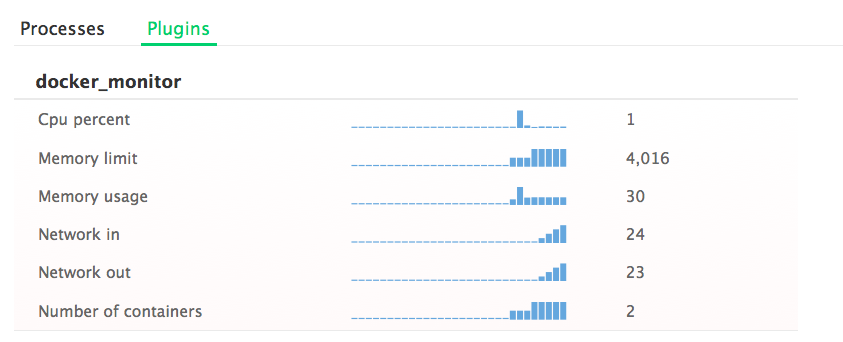
这样你查看Scout网页就能看到一个条目，其中display\_name参数（web01）就是你在scoutd.yml里面指定的。

[](http://dockone.io/uploads/article/20150525/29910859cdcb3d9157953788b9245400.png)

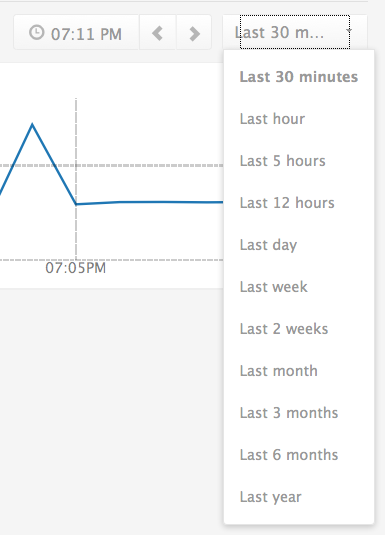
如果你点击它（web01）就会显示主机的详细信息。其中包括任何运行在你主机上的进程计数、cpu使用率以及内存利用率，值得注意的是在docker内部并没有进程的限制。

[](http://dockone.io/uploads/article/20150525/c4bc2c58260871ba2dda04333f957184.png)

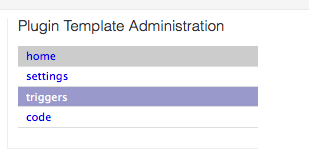
如果要添加Docker监控服务，需要单击Roles选项卡，然后选择所有服务。现在点击+插件模板按钮，接下来的Docker监视器会加载详细信息视图。一旦详细信息呈现出来，选择安装插件来添加到您的主机。接着会给你提供一个已安装插件的名称以及需指定要监视的容器。如果该字段是空的，插件将监控主机上所有的容器。点击完成按钮，一分钟左右你就可以在[Server Name] > Plugins中看到从Docker监控插件中获取的详细信息。该插件为每个主机显示CPU使用率、内存使用率、网络吞吐量以及容器的数量。

[](http://dockone.io/uploads/article/20150525/af2a47b769f00f4bdf9e26379309b5b0.png)

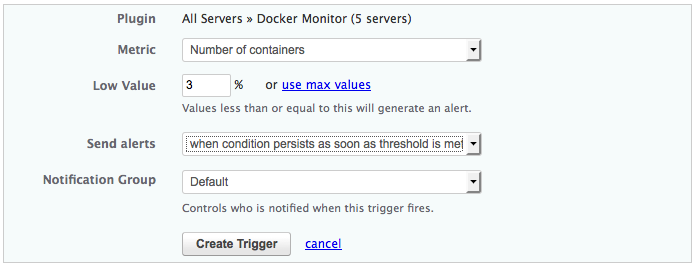
你点击任何一个图表，都可以拉取该指标的详细视图，该视图可以让你看到时间跨度更长的趋势。

[](http://dockone.io/uploads/article/20150525/15c39df8c20b738ba39d75eb29365b15.png)

该视图还支持过滤基于环境和服务器角色的指标。此外，你可以创建“Triggers”或警报，如果指标高于或低于配置的阈值它就给你发送电子邮件。这就允许您设置自动警报来通知您，比如，如果你的一些容器异常关闭以及容器计数低于一定数量。您还可以设置对平均CPU利用率的警报，举例来说，如果你正在运行的容器超过CPU利用率而发热，你会得到一个警告，当然你可以开启更多的主机到你的Docker集群。  
  
要创建触发器，请选择顶部菜单的Roles>All Servers，然后选择plugins部分的Docker monitor。然后在屏幕的右侧的Plugin template Administration菜单里选择triggers。您现在应该看到一个选项“Add a Trigger”，它将应用到整个部署。

[](http://dockone.io/uploads/article/20150525/27ba374316b40e516bb15cd0e2fd865e.png)

下面是一个触发器的例子，如果部署的容器数量低于3就会发出警报。

[](http://dockone.io/uploads/article/20150525/852c58e6989dad25c71d8ea1612f69da.png)

它的创建是为“所有的服务器”，当然你也可以用不同的角色标记你的主机使用服务器上创建的scoutd.yml文件。使用角色。你可以通过使用不同角色来应用触发器到部署的服务器的一个子集上。例如，你可以设置一个当在你的网络的节点的容器数量低于一定数量时的警报。即使是基于角色的触发器我仍然觉得Scout的警报系统可能做的更好。这是因为许多Docker部署具有相同主机上的多种多样的容器。在这种情况下为特定类型的容器设置触发器将是不可能的由于角色被应用到主机上的所有容器。  
  
比起CAdvisor，使用Scout的另一个优点是，它有[大量的插件](https://scoutapp.com/plugin_urls)，除了Docker信息他们可以吸收其他有关你的部署的数据。这使得Scout是你的一站式监控系统，而无需对系统的各种资源来安装各种不同的监控系统。  
  
Scout的一个缺点是，它不显示有关每个主机上像CAdvisor的单独容器的详细信息。这是个问题，如果你在同一台服务器上运行大量的容器。例如，如果你想有一个触发器来提醒您的Web容器的警报，但不是Jenkins容器，这时Scout就无法支持该情况。尽管有这个缺点，Scout还是一个相当有用的工具来监控你的Docker部署。当然这要付出一些代价，每个监控的主机十美元。如果你要运行一个有多台主机的超大部署，这个代价会是个考虑因素。

评分：

1. 易于部署程度：※※※※  
2. 信息详细程度：※※  
3. 集成度：※※※  
4. 生成警报的能力：※※※  
5. 监测非Docker的资源的能力：支持  
6. 成本：每个主机$10

## Data Dog

从Scout移步到另一个监控服务——DataDog，它既解决几个Scout的缺点又解除了CAdvisor的局限性。要使用DataDog，先在[https://www.datadoghq.com/](https://www.datadoghq.com/" \t "/root/文档\\x/_blank)注册一个DataDog账户。一旦你登录到您的帐户，您将看到支持集成的每种类型的指令列表。从列表中选择Docker，你会得到一个Docker run命令（如下），将其复制到你的主机。该命令需要你的预先设置的API密钥，然后你可以运行该命令。大约45秒钟后您的代理将开始向DataDog系统报告。

docker run -d --privileged --name dd-agent             \

-h `hostname`                                      \

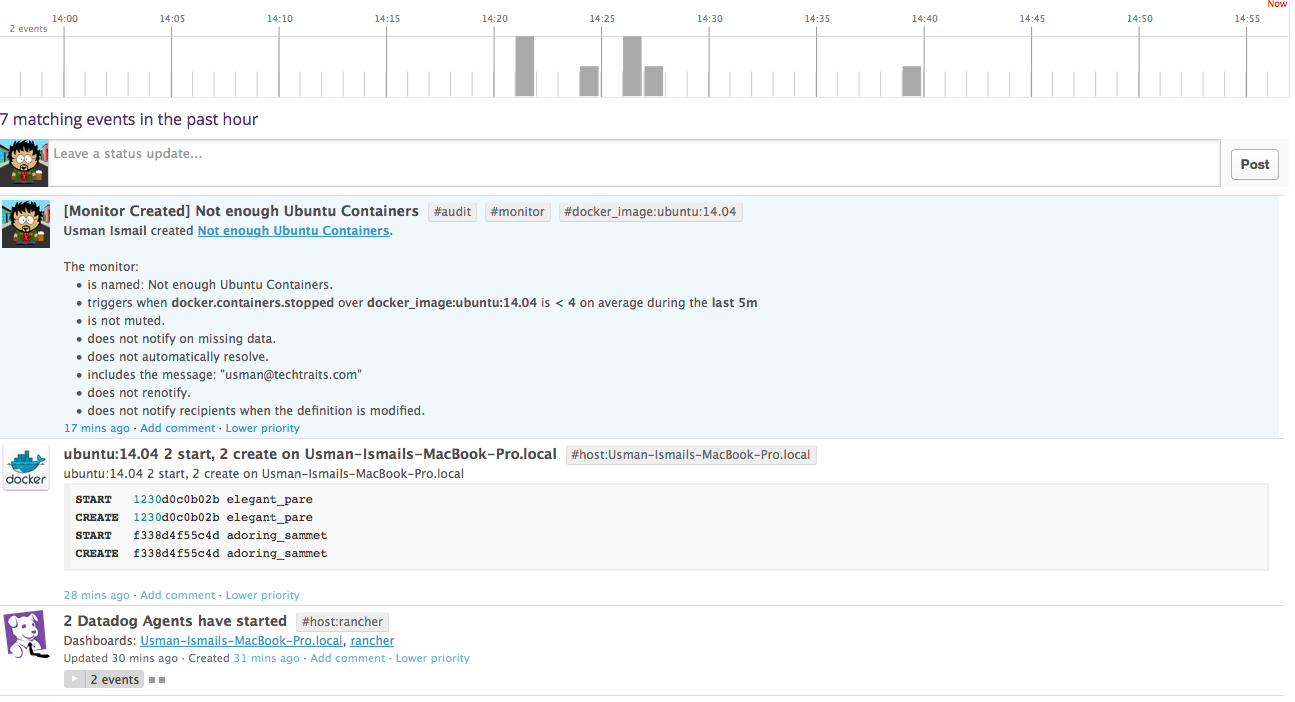
-v /var/run/docker.sock:/var/run/docker.sock       \

-v /proc/mounts:/host/proc/mounts:ro               \

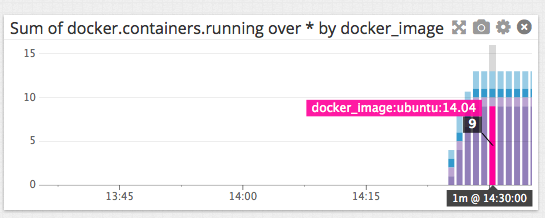
-v /sys/fs/cgroup/:/host/sys/fs/cgroup:ro          \

-e API\_KEY=YOUR\_API\_KEY datadog/docker-dd-agent    \

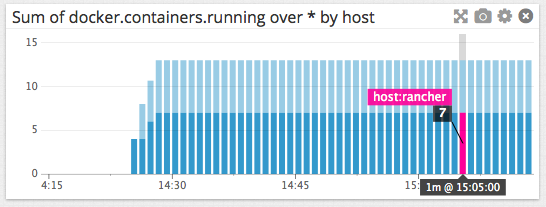
此时，容器提示你可以在DataDog Web的Events tab上处理和查看有关集群的所有动态。所有容器的启动和终止都是事件流的一部分。

[](http://dockone.io/uploads/article/20150525/a28a4b2cde0331788067f2ea474213f1.png)

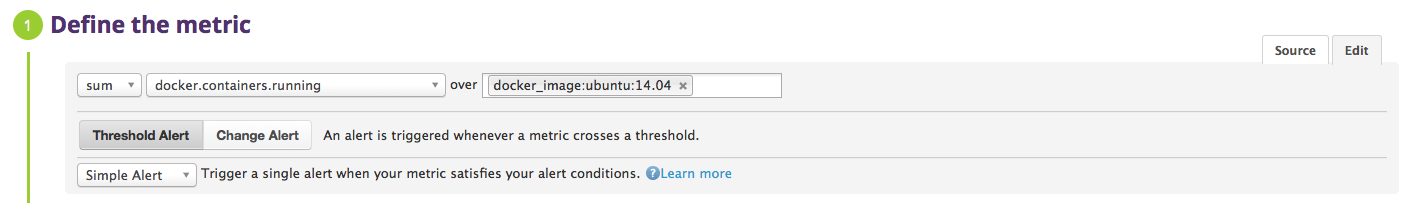
您也可以点击Dashboards标签并点击创建仪表板以合计您整个群集的指标。 Datadog收集在系统中运行的所有容器的CPU使用率、内存以及I/O的指标。此外，也可以获得容器运行和停止次数以及Docker的镜像数量。Dashboard视图可以创建任何数据的图标，或者设置整个部署、主机群、容器镜像指标的图表。例如下图显示了运行容器的数量并加以镜像类型分类，此刻在我的集群运行了9个Ubuntu:14.04的容器。

[](http://dockone.io/uploads/article/20150525/477efd7d9a94fd754608afd2bf8ab7bd.png)

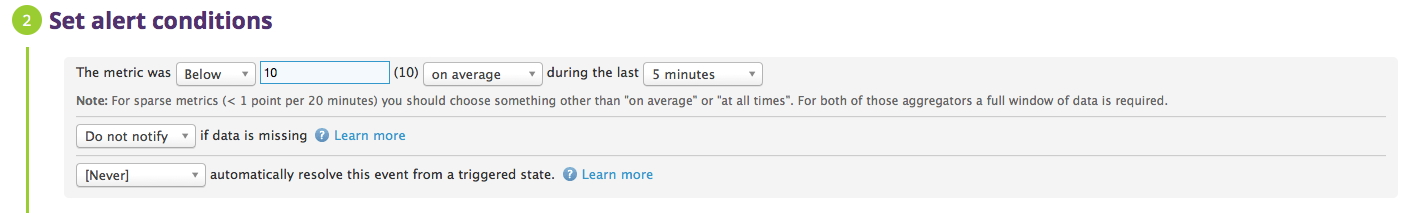
您还可以通过主机分类同样的数据，如下图所示，7个容器在我的Rancher主机上运行，其余的在我的本地的笔记本电脑。

[](http://dockone.io/uploads/article/20150525/6c8d6654c3b280ec3d1dfffc2a175a08.png)

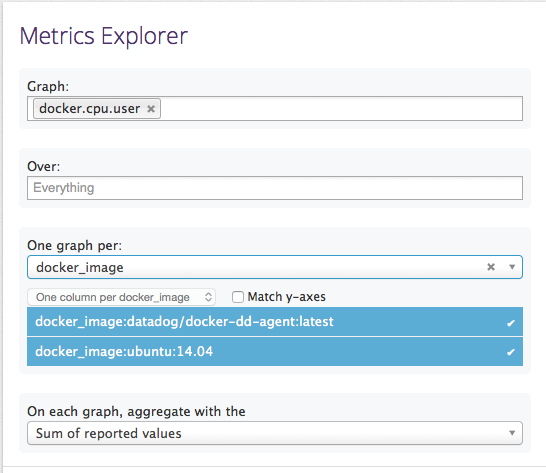
DataDog还支持一种称为Monitors的警报功能。DataDog的一个monitor相当于Scout的一个触发器，并允许您定义各种指标的阈值。 比起Scout，DataDog的警报系统相当灵活与详细。下面的例子说明如何指定您监视的Ubuntu容器的终止，因此你会监视从Ubuntu:14.04的Docker镜象所创建容器的docker.containers.running信息。

[](http://dockone.io/uploads/article/20150525/d44927f0557fc50f15deeb52d1113be3.png)

然后，特定的警报情况是，如果在我们的部署中最后5分钟有（平均）少于十个Ubuntu容器，你就会被警报。尽管这里没有显示，你会被要求填写发送出去时的指定消息在这个警报被触发后，而且还有受到此警报的目标。在当前的例子中，我用一个简单的绝对阈值。您也可以指定一个基于增量的警报，比如是在最后五分钟里停止的容器的平均计数是四的警报。

[](http://dockone.io/uploads/article/20150525/7b9014b99e95a4c6d7002c02c370acaf.png)

最后，使用Metrics Explorer选项卡可以临时聚集你的指标来帮助调试问题或者提取具体的数据信息。该视图允许您基于对容器镜像或主机绘制任何指标的图表。您可以将输出的数据组合成一个单一的图形或者通过镜像或主机的分组来生成一组图形。

[](http://dockone.io/uploads/article/20150525/56fdde3a2037b9a4c0b98bdef025e032.png)

DataDog相比scout在某些功能上做了显著地改善，方便使用以及用户友好的设计。然而这一级别伴随着额外的成本，因为每个DataDog agent价格为$15。

评分：

1. 易于部署程度：※※※※※  
2. 信息详细程度：※※※※※  
3. 集成度：※※※※※  
4. 生成警报的能力：支持  
5. 监测非Docker的资源的能力：※※※※※  
6. 成本：每个主机$15

## Sensu Monitoring Framework

Scout和Datadog提供集中监控和报警系统，然而他们都是被托管的服务，大规模部署的话成本会很突出。如果你需要一个自托管、集中指标的服务，你可以考虑[sensu open source monitoring framework](https://sensuapp.org/)。要运行Sensu服务器可以使用[hiroakis/docker-sensu-server](https://registry.hub.docker.com/u/hiroakis/docker-sensu-server/)容器。这个容器会安装sensu-server、uchiwa Web界面、Redis、rabbitmq-server以及sensu-api。不幸的是sensu不支持Docker。但是，使用插件系统，您可以配置支持容器指标以及状态检查。  
  
在开启sensu服务容器之前，你必须定义一个可以加载到服务器中检查。创建一个名为check-docker.json的文件并添加以下内容到此文件。这个文件告诉Sensu服务器在所有有docker标签的客户端上每十秒运行一个名为load-docker-metrics.sh的脚本。

{

"checks": {

"load\_docker\_metrics": {

  "type": "metric",

  "command": "load-docker-metrics.sh",

  "subscribers": [

    "docker"

  ],

  "interval": 10

}

}

}

现在，您可以使用下面的命令通过我们的检查配置文件来运行Sensu服务器Docker容器。一旦你运行该命令，你就可以在浏览器输入http://YOUR\_SERVER\_IP:3000来访问uchiwa界面。

docker run -d --name sensu-server                                           \

-p 3000:3000                                                            \

-p 4567:4567                                                            \

-p 5671:5671                                                            \

-p 15672:15672                                                          \

-v $PWD/check-docker.json:/etc/sensu/conf.d/check-docker.json           \

hiroakis/docker-sensu-server

这样Sensu服务器就开启了，你就可以对每个运行有我们的Docker容器的主机上开启sensu客户端。你告诉容器将有一个名为load-docker-metrics.sh的脚本，所以让我们创建脚本，并将其放到我们的客户端容器内。创建该文件并添加如下所示的文本，将HOST\_NAME替换为您的主机的逻辑名称。下面的脚本是为运行容器、所有容器以及镜像而使用Docker远程API来拉取元数据。然后它打印出来sensu的键值标示的值。该sensu服务器将读取标准输出并收集这些指标。这个例子只拉取这三个值，但根据需要，你可以使脚本尽可能详细。请注意，你也可以添加多个检查脚本，如thos，只要早前在服务配置文件中你引用过它们。你也可以定义你想要检查运行Docker容器数量降至三个以下的失败。你还可以使检查通过从检查脚本返回一个非零值失败。

#!/bin/bash

set -e

# Count all running containers

running\_containers=$(echo -e "GET /containers/json HTTP/1.0\r\n" | nc -U /var/run/docker.sock \

| tail -n +5                                                           \

| python -m json.tool                                                  \

| grep \"Id\"                                                          \

| wc -l)

# Count all containers

total\_containers=$(echo -e "GET /containers/json?all=1 HTTP/1.0\r\n" | nc -U /var/run/docker.sock \

| tail -n +5 \

| python -m json.tool \

| grep \"Id\" \

| wc -l)

# Count all images

total\_images=$(echo -e "GET /images/json HTTP/1.0\r\n" | nc -U /var/run/docker.sock \

| tail -n +5 \

| python -m json.tool \

| grep \"Id\" \

| wc -l)

echo "docker.HOST\_NAME.running\_containers ${running\_containers}"

echo "docker.HOST\_NAME.total\_containers ${total\_containers}"

echo "docker.HOST\_NAME.total\_images ${total\_images}"

if [ ${running\_containers} -lt 3 ]; then

exit 1;

fi

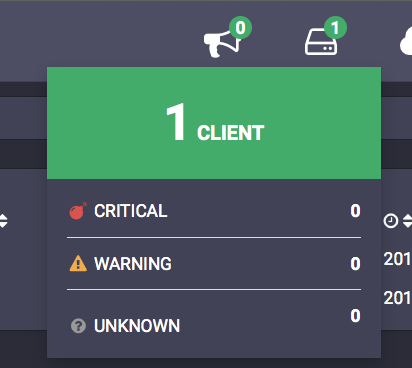
现在你已经定义了Docker载入指标检查，那就需要使用[usman/sensu-client](https://registry.hub.docker.com/u/usman/sensu-client/)容器来启动sensu客户端。您可以使用如下所示的命令启动sensu客户端。需要注意的是，容器必须以privileged来运行以便能够访问Unix sockets，它必须有Docker socket挂载以及你上面定义的load-docker-metrics.sh脚本。确保load-docker-metrics.sh脚本在你的主机的权限标记为可执行。容器也需要将SENSU\_SERVER\_IP、RABIT\_MQ\_USER、RABIT\_MQ\_PASSWORD、CLIENT\_NAME以及CLIENT\_IP作为参数，请指定这些参数到您设置的值。其中RABIT\_MQ\_USER与RABIT\_MQ\_PASSWORD默认值是sensu和password。

docker run -d --name sensu-client --privileged                                \

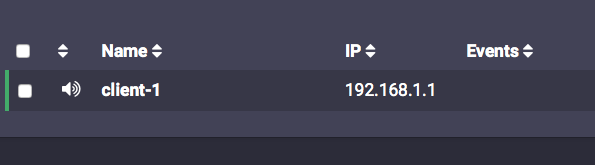
-v $PWD/load-docker-metrics.sh:/etc/sensu/plugins/load-docker-metrics.sh  \

-v /var/run/docker.sock:/var/run/docker.sock                              \

usman/sensu-client SENSU\_SERVER\_IP RABIT\_MQ\_USER RABIT\_MQ\_PASSWORD CLIENT\_NAME CLIENT\_IP

[](http://dockone.io/uploads/article/20150525/9a65fee9fd150d8d8935eaff27b4abe7.png)

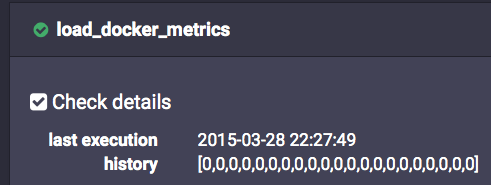
运行完此命令，一会儿你应该看到客户端计数增加1在uchiwa界面。如果您点击客户端图标，你应该看到，包括你刚才添加的客户端的客户端名单。我的客户端1是client-1以及指定的主机IP为192.168.1.1。

[](http://dockone.io/uploads/article/20150525/c2fbd10c19cef81888b7a5b043899fbd.png)

如果你点击客户端名称你应该得到检查的进一步细节。你可以看到load\_docker\_metrics检查在3月28日的10:22运行过。

[](http://dockone.io/uploads/article/20150525/f3483381af3a878a18e2ab1bbf5935f3.png)

如果你点击检查名称就可以看到检查运行的进一步细节。零表明没有错误，如果脚本失败（例如，如果您的Docker守护进程死掉），你会看到一个错误代码（非零）值。虽然在目前的文章中没有涉及这个，你也还可以使用[Handlers](http://sensuapp.org/docs/0.11/adding_a_handler)在sensu设置这些检查失败处理程序来提醒您。此外，uchiwa只显示检查的值，而不是收集的指标。需要注意的是sensu不存储所收集的指标，它们必须被转发到一个时间序列的数据库如InfluxDB或Graphite。这也是通过Handlers做到的。如何配置指标转发到graphite[可以参考这里](http://www.joemiller.me/2012/02/02/sensu-and-graphite/)。

[](http://dockone.io/uploads/article/20150525/5e860c6b296f7911ea9efa8d3a1b0e4d.png)

Sensu支持我们所有的评价标准，你可以对我们Docker容器和主机收集尽可能多的细节。此外，你能够聚合所有主机的值到一个地方，并对这些检查发出警报。这些警报并没有DataDog或Sc​​out的先进，因为你仅能够提​​醒单独的主机上检查失败。然而，Sensu的大缺点是部署的难度。虽然我已经使用Docker容器自动部署许多步骤，Sensu仍然是一个需要我们安装，启动和分开维护Redis、RabitMQ、Sensu API、uchiwa与Sensu Core的复杂系统。此外，你将需要更多的工具，如Graphite来呈现指标值以及生产部署需要定制容器，今天我已经使用了一个容器为了密码的安全以及自定义的SSL证书。除了您重启容器后才能添加更多的检查，你将不得不重新启动Sensu服务，因为这是它开始收集新的标准的唯一途径。由于这些原因，我对Sensu的在易于部署的评价相当的低。

评分：

1. 易于部署程度：※  
2. 信息详细程度：※※※※  
3. 集成度：※※※※  
4. 生成警报的能力：支持但有限  
5. 监测非Docker的资源的能力：※※※※※  
6. 成本：免费

## 总结

今天的文章涵盖了多种选项用于监控Docker容器，从免费的选择， 如Docker stats、CAdvisor或Sensu，到有偿服务，如Scout和DataDog。我的研究到目前为止DataDog似乎是用于监控Docker部署的最好的系统。只需几秒的安装以及单行命令，所有主机都在同一个地方报告指标，在UI方面，历史趋势是显而易见的，并且Datadog支持更深层次的指标以及报警。然而，$15一个主机系统对于大型部署是昂贵的。对于较大规模，自托管部署，Sensu是能够满足大多数的要求，不过在建立和管理一个Sensu集群的复杂性可能让人望而却步。很显然，有很多其他的自托管的选项，如Nagios或Icinga，他们都类似Sensu。  
  
但愿今天这篇文章会给你一些想法对于监视容器的选择。我会继续调查其他选项，包括使用CollectD、Graphite或InfluxDB与Grafana的更精简的自我管理的容器监控系统。敬请关注更多的细节。  
  
其他信息：发布本文后，我有一些建议去评估Prometheus和Sysdig云，两个非常好的监控Docker的选择。我在这两个服务上花了一些时间，并添加了第二部分到这个文章中。你可以在[这里](http://rancher.com/docker-monitoring-continued-prometheus-and-sysdig/)（译注：过后会翻译这篇）找到它。  
  
想要了解更多关于监控和管理Docker，请参加我们的下一个Rancher在线meetup。  
  
原文：[Comparing Five Monitoring Options for Docker](http://rancher.com/comparing-monitoring-options-for-docker-deployments/) （翻译：[田浩浩](http://dockone.io/people/llitfkitfk) 校对：魏小红）