容器基础技术介绍与实践

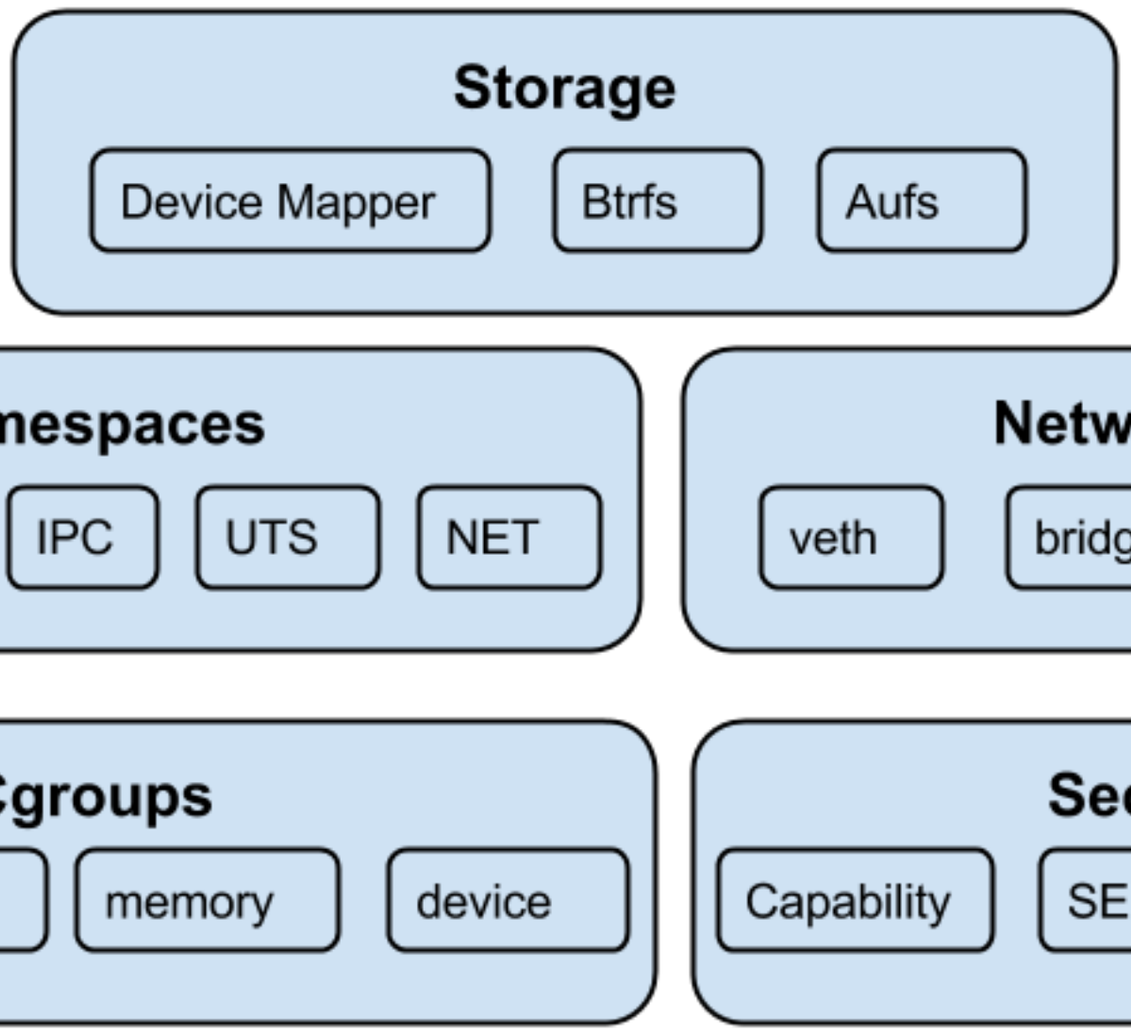
陈显鹭

2017.3.14

docker

docker DaoCloud IT大咖说 不止于技术

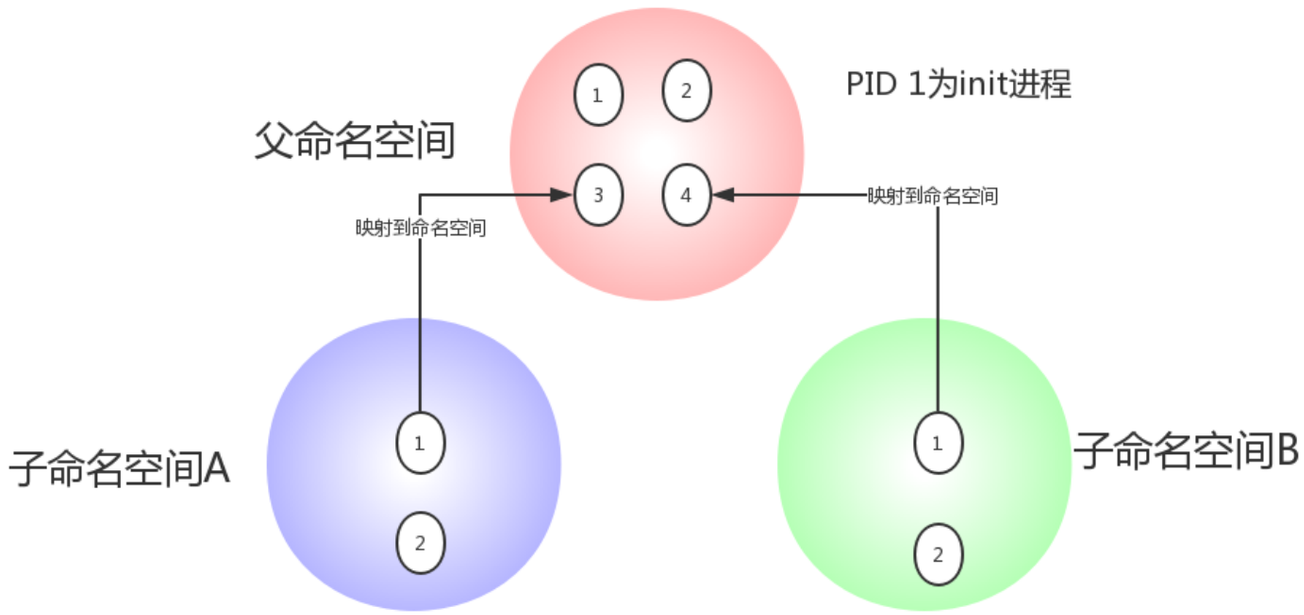
Linux Kernel



Linux Namespace

- Linux Namespace 是kernel 的一个功能，它可以隔离一系列系统的资源

- 资源包括进程树，网络接口，挂载点等等



Linux Namespace

- Namespace 的API主要使用三个系统调用
- clone() - 创建新进程。根据系统调用参数来判断哪些类型的namespace被创建，而且它们的子进程也会被包含到这些namespace中
- unshare() - 将进程移出某个namespace
- setns() - 将进程加入到namespace中

Linux Namespace 类型

Namespace类型	系统调用参数	内核版本
Mount namespaces	CLONE_NEWNS	2.4.19
UTS namespaces	CLONE_NEWUTS	2.6.19
IPC namespaces	CLONE_NEWIPC	2.6.19
PID namespaces	CLONE_NEWPID	2.6.24
Network namespaces	CLONE_NEWNET	2.6.29
User namespaces	CLONE_NEWUSER	3.8

- UTS Namespace: 主要隔离nodename和domainname两个系统标识。在UTS namespace里面，每个 namespace 允许有自己的hostname。

- IPC Namespace: 隔离 System V IPC 和POSIX message queues.每一个IPC Namespace都有他们自己的System V IPC 和POSIX message queue。

- Mount Namespace: 隔离各个进程看到的挂载点视图.在mount namespace 中调用mount()和umount()仅仅只会影响当前namespace内的文件系统，而对全局的文件系统是没有影响的。

- User Namespace: 主要是隔离用户的用户组ID以及权限，由于涉及到权限问题，可以说是最复杂的一个namespace

- Network Namespace:隔离网络设备，IP地址端口等网络栈的namespace, 可以让每个容器拥有自己独立的网络设备（虚拟的），而且容器内的应用可以绑定到自己的端口不相互冲突。在宿主机上搭建网桥后，就能很方便的实现容器之间的通信，而且不同容器上的应用可以使用相同的端口。

PID Namespace 介绍

- PID namespace是用来隔离进程 id
- 同样的一个进程在不同的 PID Namespace 里面可以

```
root@acs-node-1:/home/vagrant# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PO
1a963a4bb2c4        nginx               "nginx -g 'daemon ..."   2 seconds ago       Up 2 seconds        80
root@acs-node-1:/home/vagrant# ps -ef | grep nginx
root     10198 10180  0 07:27 ?        00:00:00 nginx: master process nginx -g daemon off;
sshd     10225 10198  0 07:27 ?        00:00:00 nginx: worker process
root     10235 10036  0 07:27 pts/0    00:00:00 grep --color=auto nginx
root@acs-node-1:/home/vagrant# docker exec -ti 1a96 ps -ef
UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 07:27 ?        00:00:00 nginx: master process nginx -g d
nginx         8     1  0 07:27 ?        00:00:00 nginx: worker process
root          9     0  0 07:28 ?        00:00:00 ps -ef
```

PID Namespace Demo



```
package main

import (
    "log"
    "os"
    "os/exec"
    "syscall"
)

func main() {
    cmd := exec.Command("sh")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWIPC | syscall.CLONE_NEWPID,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Run(); err != nil {
        log.Fatal(err)
    }
}
```

运行代码，分别在两个termainal内查看
一个在namespace内显示进程pid为1
另外一个宿主机显示pid为20190
与刚才docker 容器行为相似

```
root@iZ254rt8xf1Z:~/gocode/src/book# go run main.go
# echo $$
 1
```

```
root@iZ254rt8xf1Z:~# pstree -pl
 |-sshd(894)-+-sshd(9455)---bash(9475)---bash(19619)
    |           |-sshd(19715)---bash(19734)
    |           |-sshd(19853)---bash(19872)---go(20179)-+-main(20190)-+-sh(20193)
    |           |                                       |             |-{main}(20191)
    |           |                                       |             `-{main}(20192)
    |           |                                       |-{go}(20180)
    |           |                                       |-{go}(20181)
    |           |                                       |-{go}(20182)
    |           |                                       `-{go}(20186)
    |           `-sshd(20124)---bash(20144)---pstree(20196)
```

Linux Cgroups

- Linux Cgroups(Control Groups) 提供了对一组进程及将来的子进程的资源的限制，控制和统计的能力这些资源包括CPU，内存，存储，网络等

- Cgroups共有三大组件

- cgroup: 是对进程分组管理的一种机制，一个cgroup包含一组进程，并可以在这个cgroup上增加Linux subsystem的各种参数的配置，将一组进程和一组subsystem的系统参数关联起来。

- subsystem:一组资源控制的模块,一般包含 cpu, devices, memory, net_prio。每个subsystem会关联到定义了相应限制的cgroup上，并对这个cgroup中的进程做相应的限制和控制

- hierarchy ： 把一组cgroup串成一个树状的结构，一个这样的树便是一个hierarchy，通过这种树状的结构，Cgroups可以做到继承。比如我的系统对一组定时的任务进程通过cgroup1限制了CPU的使用率，然后其中有一个定时dump日志的进程还需要限制磁盘IO,为了避免限制了影响到其他进程，就可以创建cgroup2继承于cgroup1并限制磁盘的IO，这样cgroup2便继承了cgroup1中的CPU的限制，并且又增加了磁盘IO的限制而不影响到cgroup1中的其他进程

- 通过上面的组件的描述我们就不难看出，Cgroups的是靠这三个组件的相互协作实现的，那么这三个组件是什么关系呢？

- 系统在创建新的hierarchy之后，系统中所有的进程都会加入到这个hierarchy的根cgroup节点中，这个cgroup根节点是hierarchy默认创建，后面在这个hierarchy中创建cgroup都是这个根cgroup节点的子节点。

- 一个subsystem只能附加到一个hierarchy上面

- 一个hierarchy可以附加多个subsystem

- 一个进程可以作为多个cgroup的成员，但是这些cgroup必须是在不同的hierarchy中

- 一个进程fork出子进程的时候，子进程是和父进程在同一个cgroup中的，也可以根据需要将其移动到其他的cgroup中。

- 这几句话现在不理解暂时没关系，后面我们实际使用过程中会逐渐的了解到他们之间的联系的。

Linux Cgroups Demo

- 上面了解到Cgroups中的hierarchy是一种树状的组织结构，Kernel为了让对Cgroups的配置更直观，Cgroups通过一个虚拟的树状文件系统去做配置的，通过层级的目录虚拟出cgroup树，下面我们就以一个配置的例子来了解下如何操作Cgroups。

- 首先，我们要创建并挂载一个hierarchy(cgroup树)：

```
➜ ~ mkdir cgroup-test # 创建一个hierarchy挂载点
➜ ~ sudo mount -t cgroup -o none,name=cgroup-test cgroup-test ./cgroup-test # 挂载一个hierarchy
➜ ~ ls ./cgroup-test # 挂载后我们就可以看到系统在这个目录下生成了一些默认文件
cgroup.clone_children  cgroup.procs  cgroup.sane_behavior  notify_on_release  release_agent  tasks
```

这些文件就是这个hierarchy中根节点cgroup配置项了，上面这些文件分别的意思是：

1. cgroup.clone_children cpuset的subsystem会读取这个配置文件，如果这个的值是1(默认是0)，子cgroup才会继承父cgroup的cpuset的配置。
2. cgroup.procs是树中当前节点的cgroup中的进程组ID，现在我们在根节点，这个文件中是会有现在系统中所有进程组ID。
3. notify_on_release和release_agent会一起使用，notify_on_release表示当这个cgroup最后一个进程退出的时候是否执行 。
4. release_agent，release_agent则是一个路径，通常用作进程退出之后自动清理掉不再使用的cgroup。
5. tasks也是表示该cgroup下面的进程ID，如果把一个进程ID写到tasks文件中，便会将这个进程加入到这个cgroup中。

- 我们创建在刚才创建的hierarchy的根cgroup中扩展出两个子cgroup

- 在一个cgroup的目录下创建文件夹，kernel就会把文件夹标记为这个cgroup的子cgroup，他们会继承父cgroup的属性。

```
➜ [cgroup-test] sudo mkdir cgroup-1 # 创建子cgroup "cgroup-1"
➜ [cgroup-test] sudo mkdir cgroup-2 # 创建子cgroup "cgroup-2"
➜ [cgroup-test] tree
.
|-- cgroup-1
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   `-- tasks
|-- cgroup-2
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   `-- tasks
|-- cgroup.clone_children
|-- cgroup.procs
|-- cgroup.sane_behavior
|-- notify_on_release
|-- release_agent
`-- tasks
```

在cgroup中添加和移动进程



一个进程在一个Cgroups的hierarchy中只能存在在一个cgroup节点上，系统的所有进程默认都会在根节点，可以将进程在cgroup节点间移动，只需要将进程ID写到移动到的cgroup节点的tasks文件中。

。

```
➜ [cgroup-1] echo $$
7475
➜ [cgroup-1] sudo sh -c "echo $$ >> tasks" # 将我所在的终端的进程移动到cgroup-1中
➜ [cgroup-1] cat /proc/7475/cgroup
13:name=cgroup-test:/cgroup-1
```

通过subsystem限制cgroup中进程的资源

上面我们创建hierarchy的时候，但这个hierarchy并没有关联到任何subsystem，所以没办法通过那个hierarchy中的cgroup限制进程的资源占用，其实系统默认就已经把每个subsystem创建了一个默认的hierarchy，比如memory的hierarchy

```
➜  ~ mount | grep memory
cgroup on /sys/fs/cgroup/memory type cgroup (rw,nosuid,nodev,noexe
```

在/sys/fs/cgroup/memory目录便是挂在了memory subsystem的hierarchy

```
➜ [memory] # 首先，我们不做限制启动一个占用内存的stress进程
➜ [memory] stress --vm-bytes 200m --vm-keep -m 1
➜ [memory] # 创建一个cgroup
➜ [memory] sudo mkdir test-limit-memory && cd test-limit-memory
➜ [test-limit-memory] # 设置最大cgroup最大内存占用为100m
➜ [test-limit-memory] sudo sh -c "echo "100m" > memory.limit_in_bytes"
➜ [test-limit-memory] # 将当前进程移动到这个cgroup中
➜ [test-limit-memory] sudo sh -c "echo $$ > tasks"
➜ [test-limit-memory] # 再次运行占用内存200m的的stress进程
➜ [test-limit-memory] stress --vm-bytes 200m --vm-keep -m 1
```

```
# 未限制前内存占用约200MB (2G * 10%)
PID  PPID    TIME+ %CPU %MEM  PR  NI S    VIRT    RES   UID COMMAND
8336  8335   0:08.23 99.0 10.0  20   0 R  212284 205060   1000 stress
8335  7475   0:00.00  0.0  0.0  20   0 S    7480    876   1000 stress

# 限制后内存占用约 100MB (2G * 5%)
PID  PPID    TIME+ %CPU %MEM  PR  NI S    VIRT    RES   UID COMMAND
8310  8309   0:01.17  7.6  5.0  20   0 R  212284 102056   1000 stress
8309  7475   0:00.00  0.0  0.0  20   0 S    7480    796   1000 stress
```

Docker是如何使用Cgroups

➜ ~ # docker run -m 设置内存限制
➜ ~ sudo docker run -itd -m 128m ubuntu
957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11
➜ ~ # docker会为每个容器在系统的hierarchy中创建cgroup
➜ ~ cd
/sys/fs/cgroup/memory/docker/957459145e9092618837cf94a1cb356e206f2f0da560b40cb3
1035e442d3df11
➜ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 # 查看
cgroup的内存限制
➜ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 cat
memory.limit_in_bytes
134217728 (128 * 1024 * 1024)
➜ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 # 查看
cgroup中进程所使用的内存大小
➜ 957459145e9092618837cf94a1cb356e206f2f0da560b40cb31035e442d3df11 cat
memory.usage_in_bytes
430080

Union File System

- Union File System又简称UnionFS，是一种为Linux，FreeBSD和NetBSD操作系统设计的把其他文件系统联合到一个联合挂载点的文件系统服务

- 使用branch把不同文件系统的文件和目录“透明地”覆盖，形成一个单一一致的文件系统

- 这些branch或者是read-only的或者是read-write的，所以当对这个虚拟后的联合文件系统进行写操作的时候，系统是真正写到了一个新的文件中

- 看起来这个虚拟后的联合文件系统是可以对任何文件进行操作的，但是其实它并没有改变原来的文件，这是因为unionfs用到了一个重要的资管管理技术叫写时复制

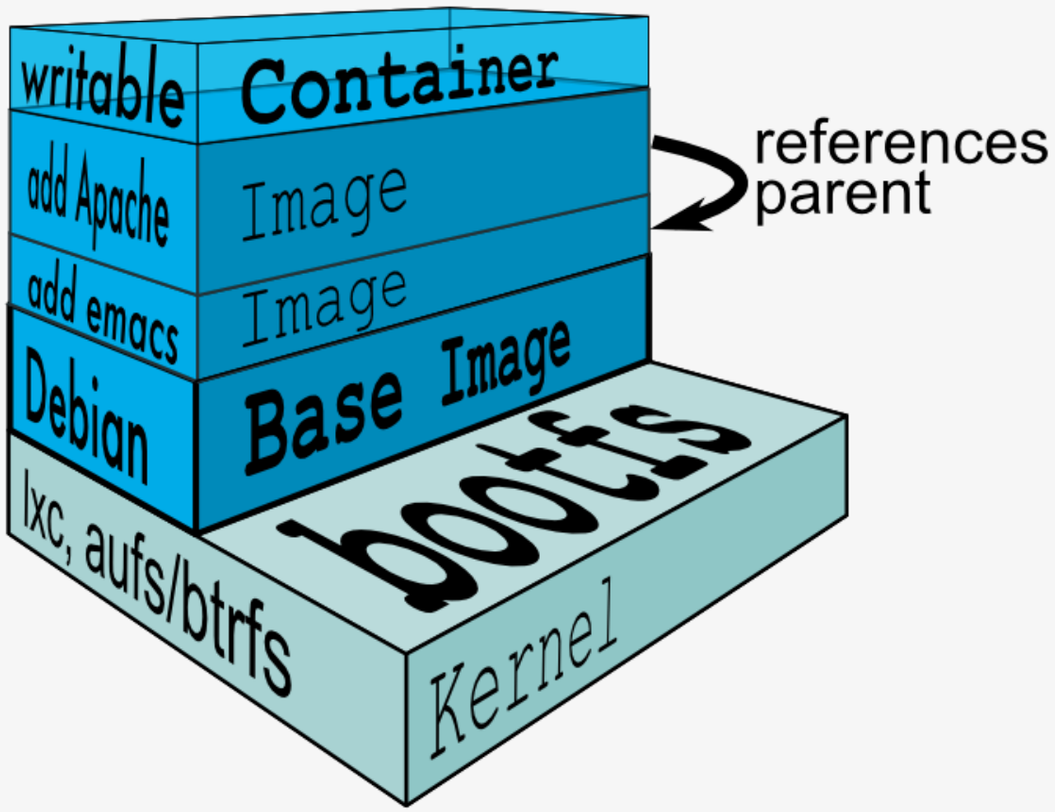
- 写时复制（copy-on-write，下文简称CoW），也叫隐式共享，是一种对可修改资源实现高效复制的资源管理技术

- 如果一个资源是重复的，但没有任何修改，这时候并不需要立即创建一个新的资源,这个资源可以被新旧实例共享

- 通过这种资源共享的方式，可以显著地减少未修改资源复制带来的消耗，但是也会在进行资源修改的时候增加小部分的开销

Docker and AUFS

- 每一个Docker image都是由一系列的read-only layers组成
- Docker使用AUFS的CoW技术来实现image layer共享和减少磁盘空间占用
- CoW意味着一旦某个文件只有很小的部分有改动，AUFS也需要复制整个文件。这种设计会对容器性能产生一定的影响，尤其是在待拷贝的文件很大，或者位于很多image layers的下方，或AUFS需要深度搜索目录结构树的时候
- 对于一个容器，每一个image layer最多只需要拷贝一次。后续的改动都会在第一次拷贝的container layer上进行。



AUFS Demo



- 把container-layer和4个名为image-layer${n}的文件夹使用aufs的方式来挂载到刚刚创建的mnt目录下

```
$ cd /home/qinyujia/aufs

$ ls
container-layer  image-layer1  image-layer2  image-layer3  image-layer4  mnt

$ cat container-layer.txt
I am container layer

$ cat image-layer1/image-layer1.txt
I am image layer 1

$ cat image-layer2/image-layer2.txt
I am image layer 2

$ cat image-layer3/image-layer3.txt
I am image layer 3

$ cat image-layer4/image-layer4.txt
I am image layer 4
```

```
$ sudo mount -t aufs -o dirs=./container-layer:./image-layer4:./image-layer3:./image-layer2:./image-layer1 none ./mnt

$ tree mnt
mnt
├── container-layer.txt(rw)
├── image-layer1.txt(ro)
├── image-layer2.txt(ro)
├── image-layer3.txt(ro)
└── image-layer4.txt(ro)
```

下面我们要执行一个有意思的操作

```
echo -e "\nwrite to mnt's image-layer4.txt" >> ./mnt/image-layer4.txt
```

```
$ cat ./mnt/image-layer4.txt
I am image layer 4

write to mnt's image-layer4.txt
```

```
$ cat image-layer4/image-layer4.txt
I am image layer 4
```

- 接下来，当我们检查container-layer文件夹的时候，发现多了一个名为image-layer4.txt的文件，文件的内容是

- 当我们尝试向mnt/image-layer4.txt文件进行写操作的时候，系统首先在mnt目录下查找名为image-layer4.txt的文件，将其拷贝到read-write层的container-layer目录中

- 接着对container-layer目录中的image-layer4.txt文件进行写操作

```
$ ls container-layer/
container-layer.txt  image-layer4.txt

$cat container-layer/image-layer4.txt
I am image layer 4

write to mnt's image-layer1.txt
```

runC

- runC是由Docker公司libcontainer项目发展而来，目前托管于OCI组织

- Linux基金会在2015年6月成立了OCI(Open Container Initiative)组织，旨在围绕容器格式定义和运行时配置制定一个开放的工业化标准。该组织主要由Docker， Google, IBM， Microsoft， Red Hat和其他许多合作伙伴创立。

- runC是一个轻量级的容器运行引擎，它包括所有Docker使用的和容器相关的系统调用的代码。是一份正式的容器标准，由Open Container Project 管理挂靠在Linux 基金会下。可以说这是真正的业界标准

- 一份抽象的最小化runc代码

```
import ****

func main() {
    if os.Args[0] == "/proc/self/exe" {
        //容器进程
        fmt.Printf("current pid %d", syscall.Getpid())
        fmt.Println()
        cmd := exec.Command("sh", "-c", `stress --vm-bytes 200m --vm-keep -m 1`)
        cmd.SysProcAttr = &syscall.SysProcAttr{
        }
        cmd.Stdin = os.Stdin
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr

        if err := cmd.Run(); err != nil {
            fmt.Println(err)
            os.Exit(1)
        }
    }

    cmd := exec.Command("/proc/self/exe")
    cmd.SysProcAttr = &syscall.SysProcAttr{
        Cloneflags: syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_NEWNS,
    }
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    if err := cmd.Start(); err != nil {
        fmt.Println("ERROR", err)
        os.Exit(1)
    } else {
        //得到fork出来进程映射在外部命名空间的pid
        fmt.Printf("%v", cmd.Process.Pid)
    }
    cmd.Process.Wait()
}
```

Docker containerd

- 早在16年3月，Docker 1.11的Docker Engine里就包含了containerd，而现在则是把containerd从Docker Engine里彻底剥离出来，作为一个独立的开源项目独立发展，目标是提供一个更加开放、稳定的容器运行基础设施

- containerd并不是直接面向最终用户的，而是主要用于集成到更上层的系统里，比如Swarm, Kubernetes, Mesos等容器编排系统

- containerd以Daemon的形式运行在系统上，通过unix domain docket暴露很低层的gRPC API，上层系统可以通过这些API管理机器上的容器

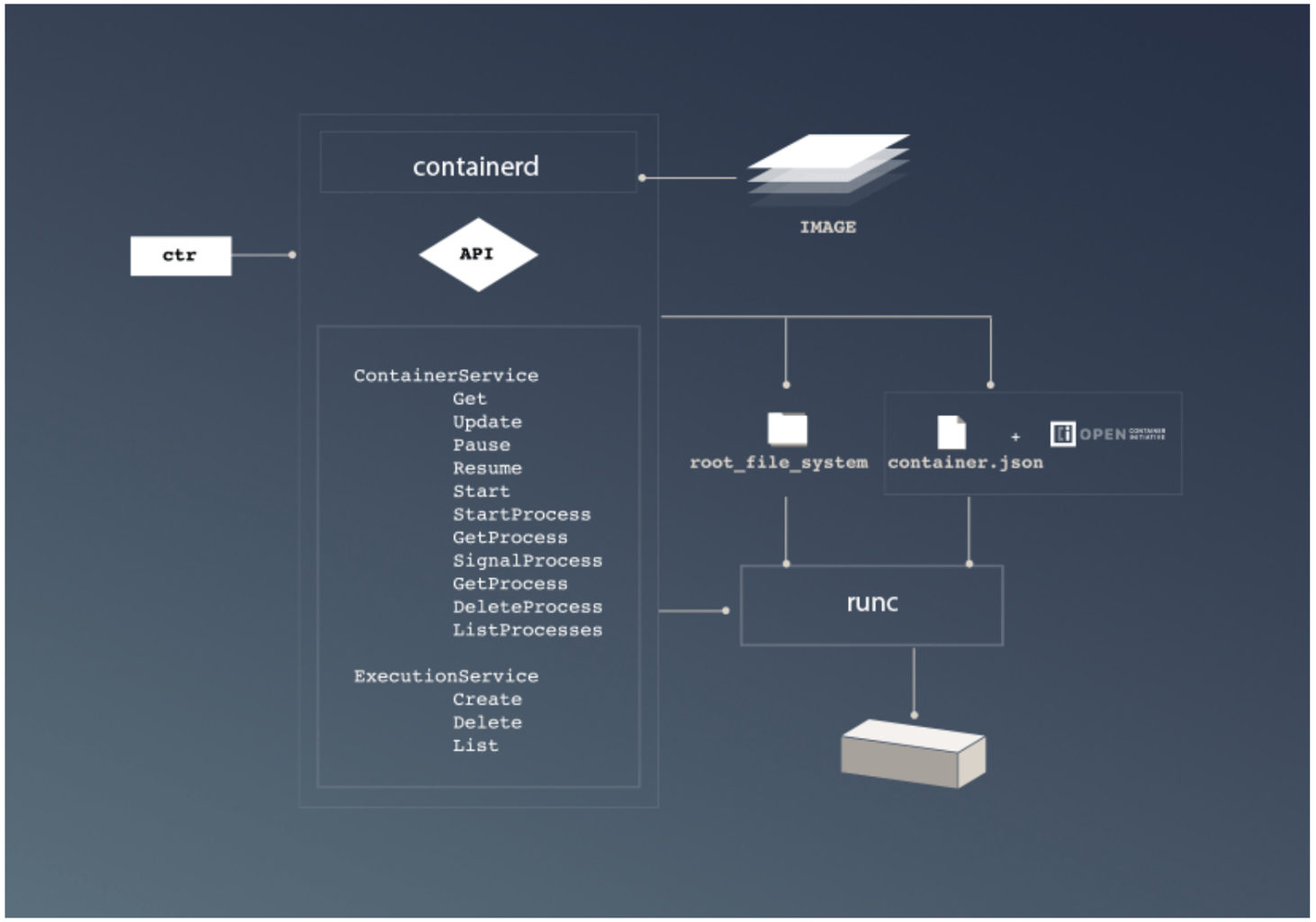
- 每个containerd只负责一台机器，Pull镜像，对容器的操作（启动、停止等），网络，存储都是由containerd完成

- 具体运行容器由runC负责，实际上只要是符合OCI规范的容器都可以支持

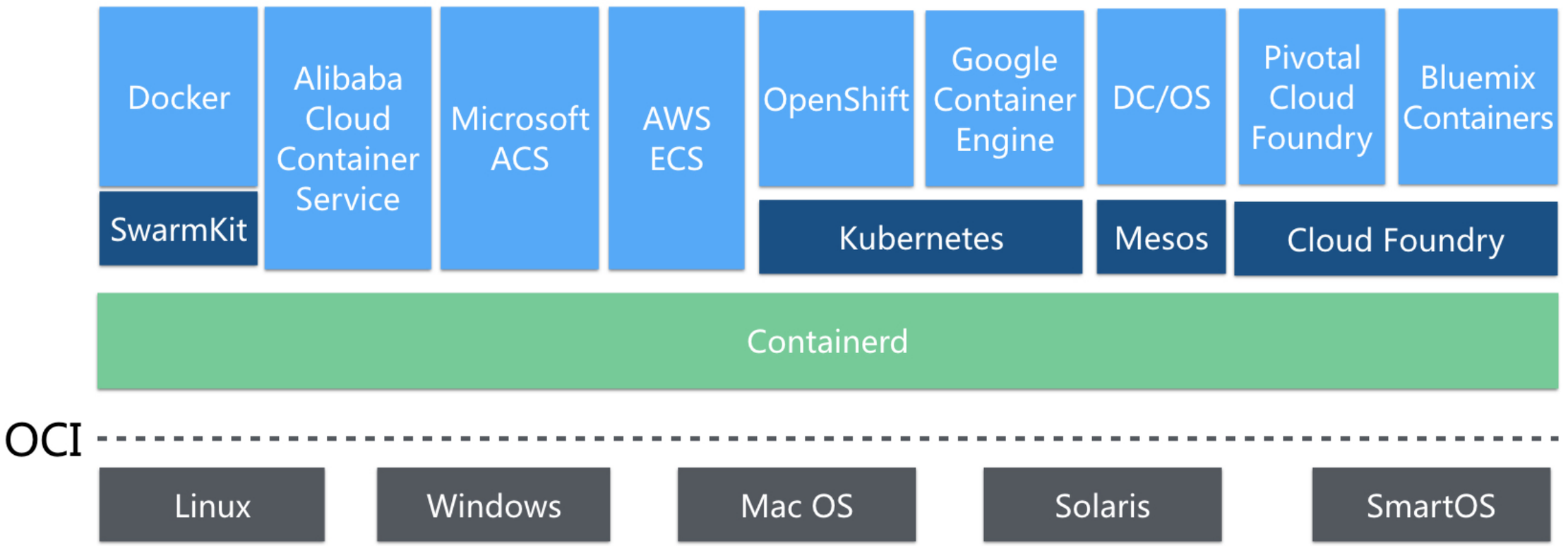
- 这对于社区和整个Docker生态来说是一件好事。对于Docker社区的开发者来说，独立的containerd更简单清晰，基于containerd增加新特性也会比以前容易。

- 对于容器编排服务来说，运行时只需要使用containerd+runC，更加轻量，容易管理

- Docker为了表示对于社区和生态的诚意，特意强调了containerd中立的地位，符合各方利益。可以预见containerd将成为Docker平台的一块重要组件。阿里云, AWS, Google, IBM和Microsoft将参与到containerd的开发中



Containerd's Role in Container Ecosystem



MORE THAN JUST CLOUD | Alibaba Cloud