

TechRest (/)

[Categories \(/categories.html\)](#) [Archive \(/archive.html\)](#) [Tags \(/tags.html\)](#)

[About \(/about.html\)](#) [Pages \(/pages.html\)](#)

Docker Getting Start: Related Knowledge

Docker 介绍: 相关技术

13 December 2013

Abstract

本文在现有文档的基础上总结了以下几点内容

1. docker的介绍，包括由来、适用场景等
2. docker背后的一系列技术 - namespace, cgroup, lxc, aufs等
3. docker在利用LXC的同时提供了哪些创新
4. 笔者对docker这种container, PaaS的一些理解
5. docker存在的问题和现有的解决思路

Docker 简介

Docker is an open-source engine that automates the deployment of any application as a lightweight, portable, self-sufficient container that will run virtually anywhere.

Docker (<http://www.docker.io/>) 是 PaaS 提供商 dotCloud (<https://www.dotcloud.com/>) 开源的一个基于 LXC 的高级容器引擎，源代码 (<https://github.com/dotcloud/docker>) 托管在 Github 上，基于 go 语言并遵从 Apache2.0 协议开源。Docker 近期非常火热，无论是从 github 上的代码活跃度，还是 Redhat 在 RHEL6.5 中集成对 Docker 的支持 (<http://developerblog.redhat.com/2013/11/26/rhel6-5-ga/>)，就连 Google 家的 Compute Engine 也支持 docker 在其之上运行 (<http://googlecloudplatform.blogspot.com/2013/12/google-compute-engine-is-now-generally-available.html>)，最近百度也用 Docker 作为其 PaaS 的基础 (<http://blog.docker.io/2013/12/baidu-using-docker-for-its-paas/>) (不知道规模多大)。

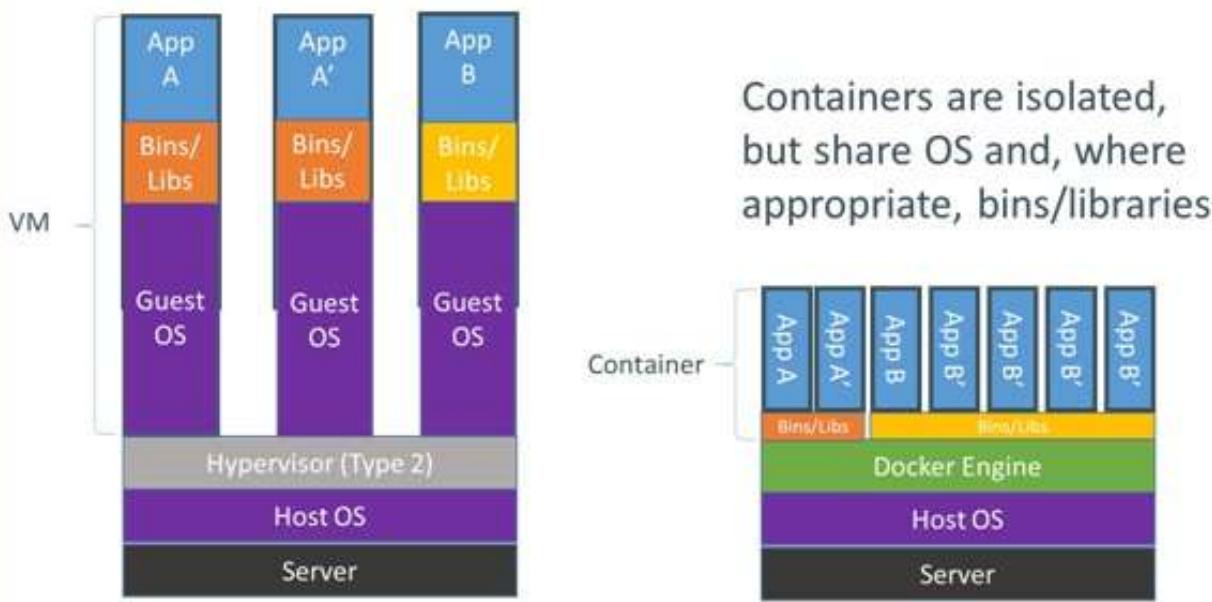
一款开源软件能否在商业上成功，很大程度上依赖三件事 - 成功的 user case, 活跃的社区和一个好故事。dotCloud 自家的 PaaS 产品建立在 docker 之上，长期维护且有大量的用户，社区也十分活跃，接下来我们看看 docker 的故事。

- 环境管理复杂 - 从各种 OS 到各种中间件到各种 app, 一款产品能够成功作为开发者需要关心的东西太多，且难于管理，这个问题几乎在所有现代 IT 相关行业都需要面对
- 云计算时代的到来 - AWS 的成功，引导开发者将应用转移到 cloud 上，解决了硬件管理的问题，然而中间件相

关的问题依然存在(所以openstack HEAT和AWS cloudformation都着力解决这个问题)。开发者思路变化提供了可能性。

- 虚拟化手段的变化 - **cloud** 时代采用标配硬件来降低成本，采用虚拟化手段来满足用户按需使用的需求以及保证可用性和隔离性。然而无论是**KVM**还是**Xen**在**docker**看来，都在浪费资源，因为用户需要的是高效运行环境而非**OS**，**GuestOS**既浪费资源又难于管理，更加轻量级的**LXC**更加灵活和快速
- **LXC**的移动性 - **LXC**在**linux 2.6**的**kernel**里就已经存在了，但是其设计之初并非为云计算考虑的，缺少标准化的描述手段和容器的可迁移性，决定其构建出的环境难于迁移和标准化管理(相对于**KVM**之类**image**和**snapshot**的概念)。**docker**就在这个问题上做出实质性的革新。这正式笔者第一次听说**docker**时觉得最独特的地方。

Containers vs. VMs



面对上述几个问题，**docker**设想是交付运行环境如同海运，**OS**如同一个货轮，每一个在**OS**基础上的软件都如同一个集装箱，用户可以通过标准化手段自由组装运行环境，同时集装箱的内容可以由用户自定义，也可以由专业人员制造。这样，交付一个软件，就是一系列标准化组件的集合的交付，如同乐高积木，用户只需要选择合适的积木组合，并且在最顶端署上自己的名字(最后一个标准化组件是用户的**app**)。这也就是基于**docker**的**PaaS**产品的原型。

What Docker Can Do

在**docker**的网站上提到了**docker**的典型场景：

- Automating the packaging and deployment of applications
- Creation of lightweight, private PaaS environments
- Automated testing and continuous integration/deployment
- Deploying and scaling web apps, databases and backend services

由于其基于**LXC**的轻量级虚拟化的特点，**docker**相比**KVM**之类最明显的特点就是启动快，资源占用小。因此对于构建隔离的标准化的运行环境，轻量级的**PaaS**(如**dokku** (<https://github.com/progrium/dokku>))，构建自动化测试和持续集成环境，以及一切可以横向扩展的应用(尤其是需要快速启停来应对峰谷的**web**应用)。

1. 构建标准化的运行环境，现有的方案大多是在一个**base OS**上运行一套puppet/chef，或者一个image文件，其缺点是前者需要**base OS**许多前提条件，后者几乎不可以修改(因为**copy on write** 的文件格式在运行时**rootfs**是**read only**的)。并且后者文件体积大，环境管理和版本控制本身也是一个问题。
2. PaaS环境是不言而喻的，其设计之初和dotcloud的案例都是将其作为PaaS产品的环境基础
3. 因为其标准化构建方法(**buildfile**)和良好的REST API，自动测试和持续集成/部署能够很好的集成进来
4. 因为LXC轻量级的特点，其启动快，而且**docker**能够只加载每个**container**变化的部分，这样资源占用小，能够在单机环境下与KVM之类的虚拟化方案相比能够更加快速和占用更少资源

What Docker Can NOT Do

Docker并不是全能的，设计之初也不是KVM之类虚拟化手段的替代品，个人简单总结了几点

1. Docker是基于Linux 64bit的，无法在windows/unix或32bit的linux环境下使用(虽然64-bit现在很普及了)
2. LXC是基于cgroup等linux kernel功能的，因此**container**的guest系统只能是linux base的
3. 隔离性相比KVM之类的虚拟化方案还是有些欠缺，所有**container**公用一部分的运行库
4. 网络管理相对简单，主要是基于**namespace**隔离
5. cgroup的cpu和cpuset提供的cpu功能相比KVM的等虚拟化方案相比难以度量(所以dotcloud主要是安内存收费)
6. docker对disk的管理比较有限
7. container随着用户进程的停止而销毁，**container**中的log等用户数据不便收集

针对1-2，有windows base应用的需求的基本可以pass了；3-5主要是看用户的需求，到底是需要一个**container**还是一个VM，同时也决定了**docker**作为 IaaS 不太可行。针对6,7虽然是**docker**本身不支持的功能，但是可以通过其他手段解决(disk quota, `mount --bind`)。总之，选用**container**还是vm，就是在隔离性和资源复用性上做**tradeoff**

另外即便**docker** 0.7能够支持非AUFS的文件系统，但是由于其功能还不稳定，商业应用或许会存在问题，而AUFS的稳定版需要kernel 3.8，所以如果想复制dotcloud的成功案例，可能需要考虑升级kernel或者换用ubuntu的server版本(后者提供deb更新)。我想这也是为什么开源界更倾向于支持ubuntu的原因(kernel版本)

Docker Usage

由于篇幅所限，这里就不再展开翻译，可参见链接 - <http://docs.docker.io/en/latest/use/> (<http://docs.docker.io/en/latest/use/>)

Docker Build File

由于篇幅所限，这里就不再展开翻译，可参见链接 - <http://docs.docker.io/en/latest/use/builder/> (<http://docs.docker.io/en/latest/use/builder/>)

Docker's Trick

What Docker Needs

Docker核心解决的问题是利用LXC来实现类似VM的功能，从而利用更加节省的硬件资源提供给用户更多的计算资源。同VM的方式不同，LXC (<http://en.wikipedia.org/wiki/LXC>) 其并不是一套硬件虚拟化方法 (http://en.wikipedia.org/wiki/Platform_virtualization) - 无法归属到全虚拟化、部分虚拟化和半虚拟化中的任意一个，而是一个操作系统级虚拟化 (http://en.wikipedia.org/wiki/Operating_system-level_virtualization)方法，理解起来可能并不像VM那样直观。所以我们从虚拟化要**docker**要解决的问题出发，看看他是怎么满足用户虚拟化需求的。

用户需要考虑虚拟化方法，尤其是硬件虚拟化方法，需要借助其解决的主要是以下4个问题：

- 隔离性 - 每个用户实例之间相互隔离，互不影响。硬件虚拟化方法给出的方法是VM, LXC给出的方法是container，更细一点是kernel namespace
- 可配额/可度量 - 每个用户实例可以按需提供其计算资源，所使用的资源可以被计量。硬件虚拟化方法因为虚拟了CPU, memory可以方便实现，LXC则主要是利用cgroups来控制资源
- 移动性 - 用户的实例可以很方便地复制、移动和重建。硬件虚拟化方法提供snapshot和image来实现，docker(主要)利用AUFS实现
- 安全性 - 这个话题比较大，这里强调是host主机的角度尽量保护container。硬件虚拟化的方法因为虚拟化的水平比较高，用户进程都是在KVM等虚拟机容器中翻译运行的，然而对于LXC，用户的进程是lxc-start进程的子进程，只是在Kernel的namespace中隔离的，因此需要一些kernel的patch来保证用户的运行环境不会受到来自host主机的恶意入侵，dotcloud(主要是)利用kernel grsec patch解决的。

Linux Namespace (ns)

LXC所实现的隔离性主要是来自kernel的namespace，其中pid, net, ipc, mnt, uts等namespace将container的进程，网络，消息，文件系统和hostname隔离开。

pid namespace

之前提到用户的进程是lxc-start进程的子进程，不同用户的进程就是通过pid namespace隔离开的，且不同namespace中可以有相同PID。具有以下特征：

1. 每个namespace中的pid是有自己的pid=1的进程(类似/sbin/init进程)
2. 每个namespace中的进程只能影响自己的同一个namespace或子namespace中的进程
3. 因为/proc包含正在运行的进程，因此在container中的pseudo-filesystem的/proc目录只能看到自己namespace中的进程
4. 因为namespace允许嵌套，父namespace可以影响子namespace的进程，所以子namespace的进程可以在父namespace中看到，但是具有不同的pid

正是因为以上的特征，所有的LXC进程在docker中的父进程为docker进程，每个lxc进程具有不同的namespace。同时由于允许嵌套，因此可以很方便的实现LXC in LXC

net namespace

有了pid namespace，每个namespace中的pid能够相互隔离，但是网络端口还是共享host的端口。网络隔离是通过net namespace实现的，每个net namespace有独立的network devices, IP addresses, IP routing tables, /proc/net目录。这样每个container的网络就能隔离开来。LXC在此基础上有5种网络类型，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)，包括常见的信号量、消息队列和共享内存。然而同VM不同，container的进程间交互实际上还是host上具有相同pid namespace中的进程间交互，因此需要在IPC资源申请时加入namespace信息 - 每个IPC资源有一个唯一的32bit ID。

mnt namespace

类似chroot，将一个进程放到一个特定的目录执行。mnt namespace允许不同namespace的进程看到的文件结构不同，这样每个namespace中的进程所看到的文件目录就被隔离开了。同chroot不同，每个namespace中的container在/proc/mounts的信息只包含所在namespace的mount point。

uts namespace

UTS("UNIX Time-sharing System") namespace允许每个container拥有独立的hostname和domain name，使其在网络上可以被视作一个独立的节点而非Host上的一个进程。

user namespace

每个container可以有不同的 user 和 group id, 也就是说可以以container内部的用户在container内部执行程序而非Host上的用户。

有了以上6种namespace从进程、网络、IPC、文件系统、UTS和用户角度的隔离，一个container就可以对外展现出一个独立计算机的能力，并且不同container从OS层面实现了隔离。然而不同namespace之间资源还是相互竞争的，仍然需要类似 `ulimit` 来管理每个container所能使用的资源 - LXC 采用的是 `cgroup`。

参考文献

[1]<http://blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part> (<http://blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part>)

[2]<http://lwn.net/Articles/531114/> (<http://lwn.net/Articles/531114/>)

Control Groups (cgroups)

`cgroups` 实现了对资源的配额和度量。`cgroups` 的使用非常简单，提供类似文件的接口，在 `/cgroup` 目录下新建一个文件夹即可新建一个group，在此文件夹中新建 `task` 文件，并将pid写入该文件，即可实现对该进程的资源控制。具体的资源配置选项可以在该文件夹中新建子 `subsystem`，`{子系统前缀}.{资源项}` 是典型的配置方法，如 `memory.usage_in_bytes` 就定义了该group 在 `subsystem memory` 中的一个内存限制选项。另外，`cgroups` 中的 `subsystem` 可以随意组合，一个 `subsystem` 可以在不同的group中，也可以一个group包含多个 `subsystem` - 也就是说一个 `subsystem`

关于术语定义

A *cgroup* associates a set of tasks with a set of parameters for one or more subsystems.

A *subsystem* is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.

我们主要关心 `cgroups` 可以限制哪些资源，即有哪些 `subsystem` 是我们关心。

cpu : 在 `cgroup` 中，并不能像硬件虚拟化方案一样能够定义CPU能力，但是能够定义CPU轮转的优先级，因此具有较高CPU优先级的进程会更可能得到CPU运算。通过将参数写入 `cpu.shares`，即可定义改 `cgroup` 的CPU优先级 - 这里是一个相对权重，而非绝对值。当然在 `cpu` 这个 `subsystem` 中还有其他可配置项，手册中有详细说明。

cpusets : cpusets 定义了有几个CPU可以被这个group使用，或者哪几个CPU可以供这个group使用。在某些场景下，单CPU绑定可以防止多核间缓存切换，从而提高效率

memory : 内存相关的限制

blkio : block IO相关的统计和限制，byte/operation统计和限制(IOPS等)，读写速度限制等，但是这里主要统计的都是同步IO

net_cls, cpuacct, devices, freezer 等其他可管理项。

参考文献

<http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-24-c> (<http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-24-c>)

<http://en.wikipedia.org/wiki/Cgroups> (<http://en.wikipedia.org/wiki/Cgroups>)

<https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>
(<https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt>)

LinuX Containers(LXC)

借助于namespace的隔离机制和cgroup限额功能， LXC提供了一套统一的API和工具来建立和管理container，LXC利用了如下 kernel 的features:

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Apparmor and SELinux profiles
- Seccomp policies
- Chroots (using pivot_root)
- Kernel capabilities
- Control groups (cgroups)

LXC 向用户屏蔽了以上 kernel 接口的细节，提供了如下的组件大大简化了用户的开发和使用工作:

- The liblxc library
- Several language bindings (python3, lua and Go)
- A set of standard tools to control the containers
- Container templates

LXC 旨在提供一个共享kernel的 OS 级虚拟化方法，在执行时不用重复加载Kernel，且container的kernel与host共享，因此可以大大加快container的启动过程，并显著减少内存消耗。在实际测试中，基于LXC的虚拟化方法的IO和CPU性能几乎接近 baremetal 的性能(论据参见文献[3])，大多数数据有相比 Xen具有优势。当然对于KVM这种也是通过Kernel进行隔离的方式，性能优势或许不是那么明显，主要还是内存消耗和启动时间上的差异。在参考文献[4]中提到了利用iozone进行 Disk IO吞吐量测试KVM反而比LXC要快，而且笔者在device mapping driver下重现同样case的实验中也确实能得到如此结论。参考文献[5]从网络虚拟化中虚拟路由的场景(个人理解是网络IO和CPU角度)比较了KVM和LXC，得到结论是KVM在性能和隔离性的平衡上比LXC更优秀 - KVM在吞吐量上略差于LXC，但CPU的隔离可管理项比LXC更明确。

关于CPU, DiskIO, network IO 和 memory 在KVM和LXC中的比较还是需要更多的实验才能得出可信服的结论。

参考文献

[1]<http://linuxcontainers.org/> (<http://linuxcontainers.org/>)

[2]<http://en.wikipedia.org/wiki/LXC> (<http://en.wikipedia.org/wiki/LXC>)

[3]<http://marceloneves.org/papers/pdp2013-containers.pdf> (<http://marceloneves.org/papers/pdp2013-containers.pdf>) (性能测试)

[4]<http://www.spinics.net/lists/linux-containers/msg25750.html> (<http://www.spinics.net/lists/linux-containers/msg25750.html>) (与KVM IO比较)

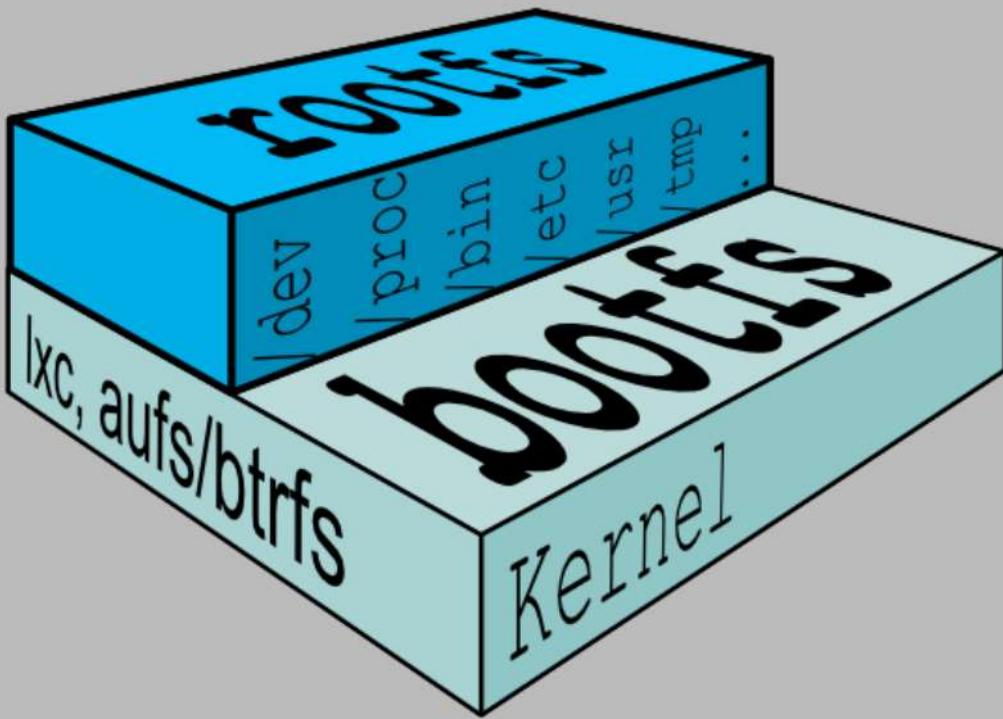
[5]<http://article.sciencepublishinggroup.com/pdf/10.11648.j.ajnc.20130204.11.pdf>
(<http://article.sciencepublishinggroup.com/pdf/10.11648.j.ajnc.20130204.11.pdf>)

AUFS

Docker对container的使用基本是建立在LXC基础之上的，然而LXC存在的问题是难以移动 - 难以通过标准化的模板制作、重建、复制和移动 container。在以VM为基础的虚拟化手段中，有image和snapshot可以用于VM的复制、重建以及移动的功能。想要通过container来实现快速的大规模部署和更新，这些功能不可或缺。Docker正是利用AUFS来实现对container的快速更新 - 在docker0.7中引入了storage driver，支持AUFS, VFS, device mapper，也为BTRFS以及ZFS引入提供了可能。但除了AUFS都未经过dotcloud的线上使用，因此我们还是从AUFS的角度介绍。

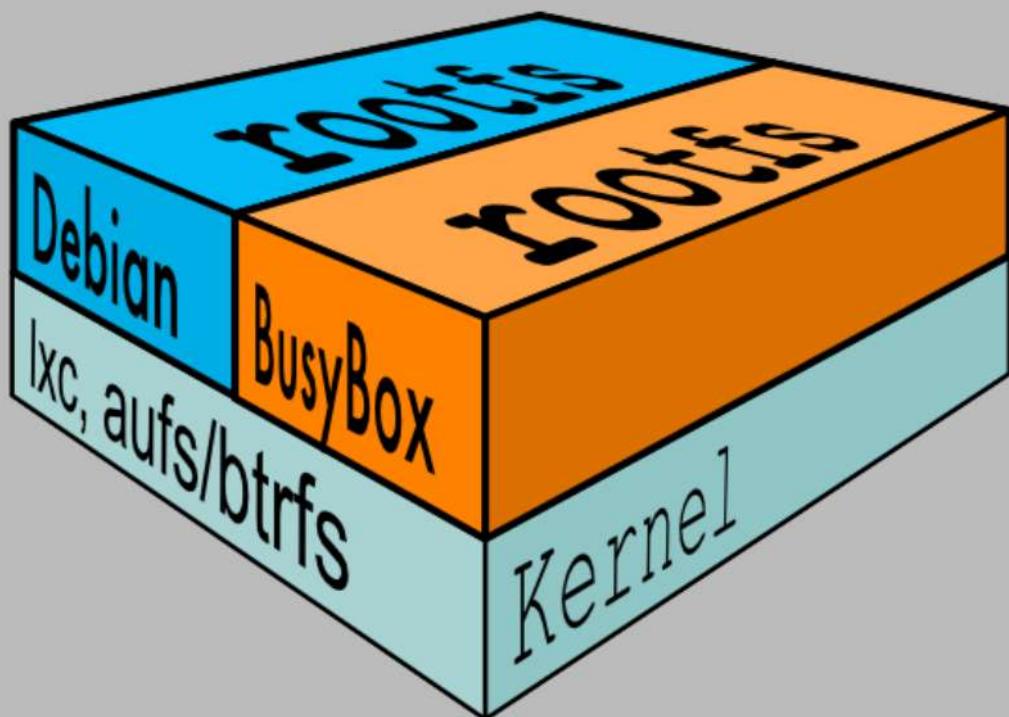
AUFS (AnotherUnionFS) 是一种 Union FS，简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统，更进一步地，AUFS支持为每一个成员目录(AKA branch)设定'readonly', 'readwrite' 和 'whiteout-able' 权限，同时AUFS里有一个类似 分层的概念，对 readonly 权限的branch可以逻辑上进行修改(增量地，不影响readonly部分的)。通常 Union FS有两个用途，一方面可以实现不借助 LVM, RAID 将多个disk挂载到一个目录下，另一个更常用的就是将一个readonly的branch和一个writeable的branch联合在一起，Live CD正是基于此可以允许在 OS image 不变的基础上允许用户在其上进行一些写操作。Docker在AUFS上构建的container image也正是如此，接下来我们从启动container中的linux为例介绍docker在AUFS特性的运用。

典型的Linux启动到运行需要两个FS - bootfs + rootfs (从功能角度而非文件系统角度)

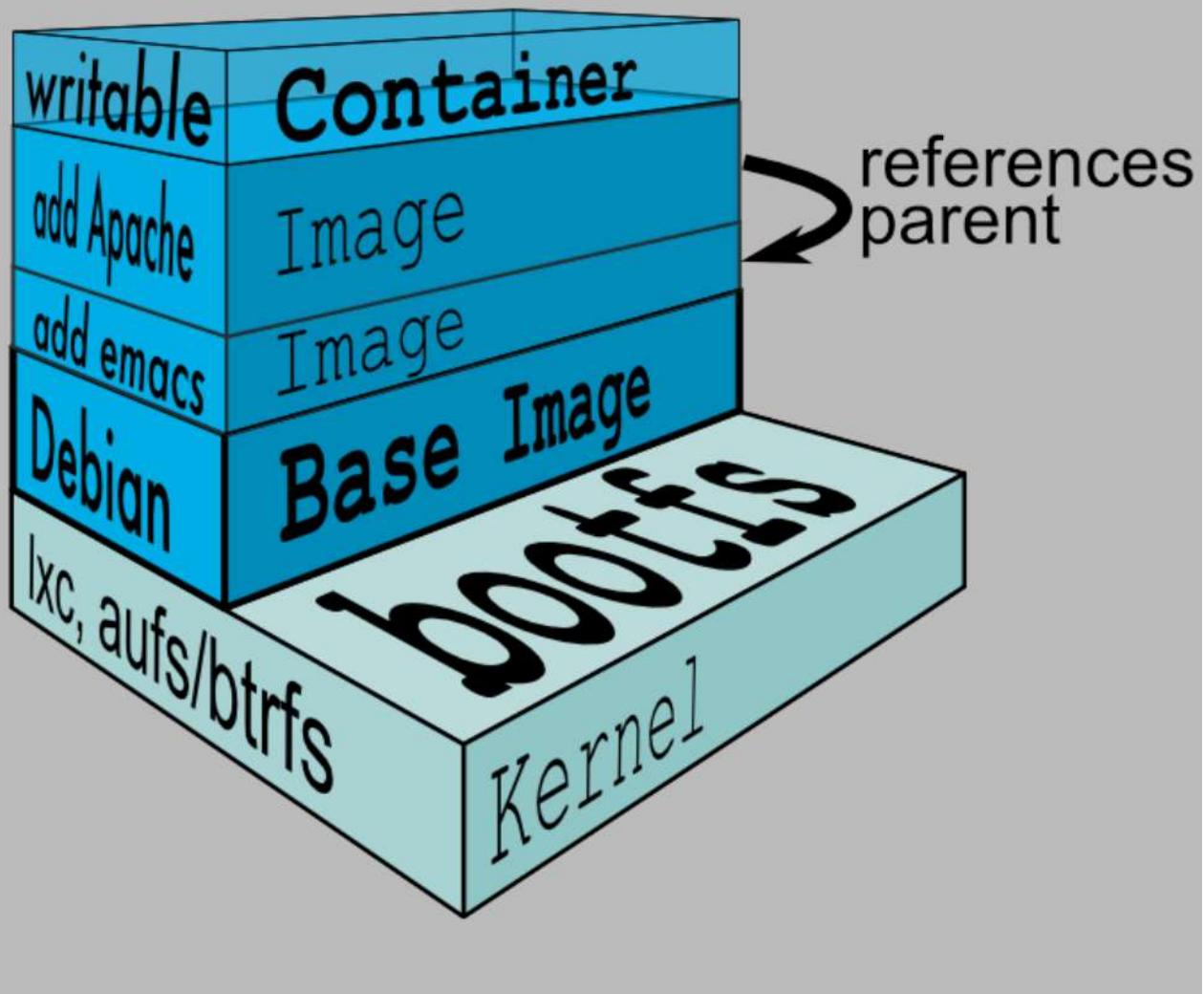


bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了。rootfs (root file system) 包含的就是典型 Linux 系统中的 `/dev`, `/proc`, `/bin`, `/etc` 等标准目录和文件。

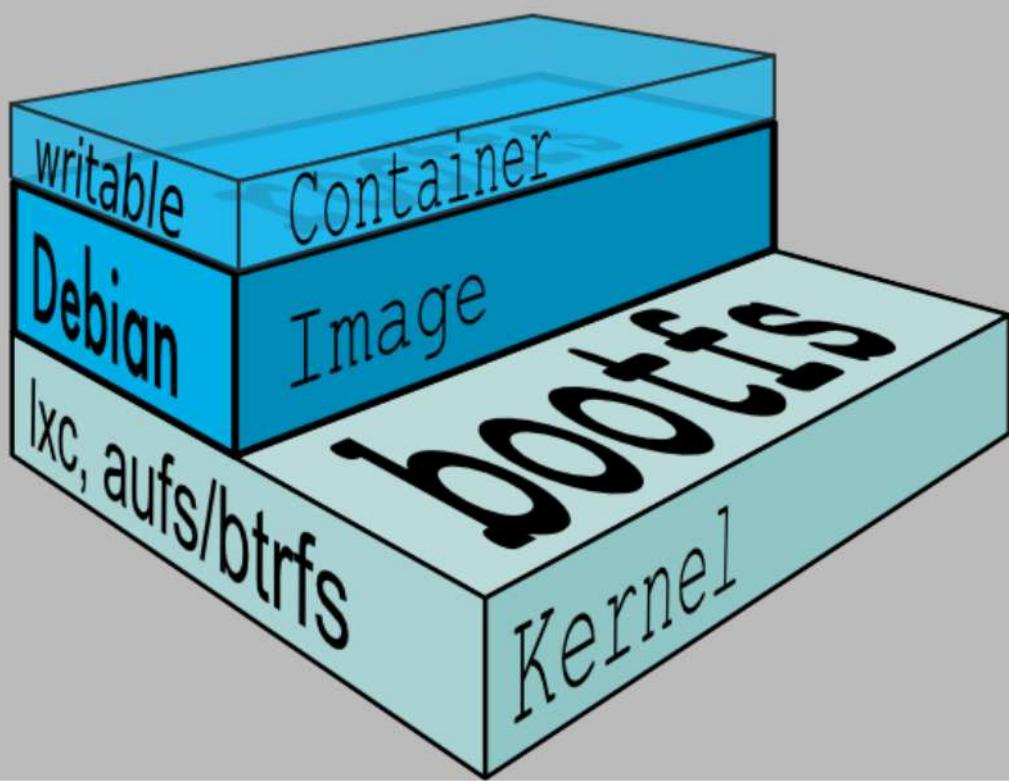
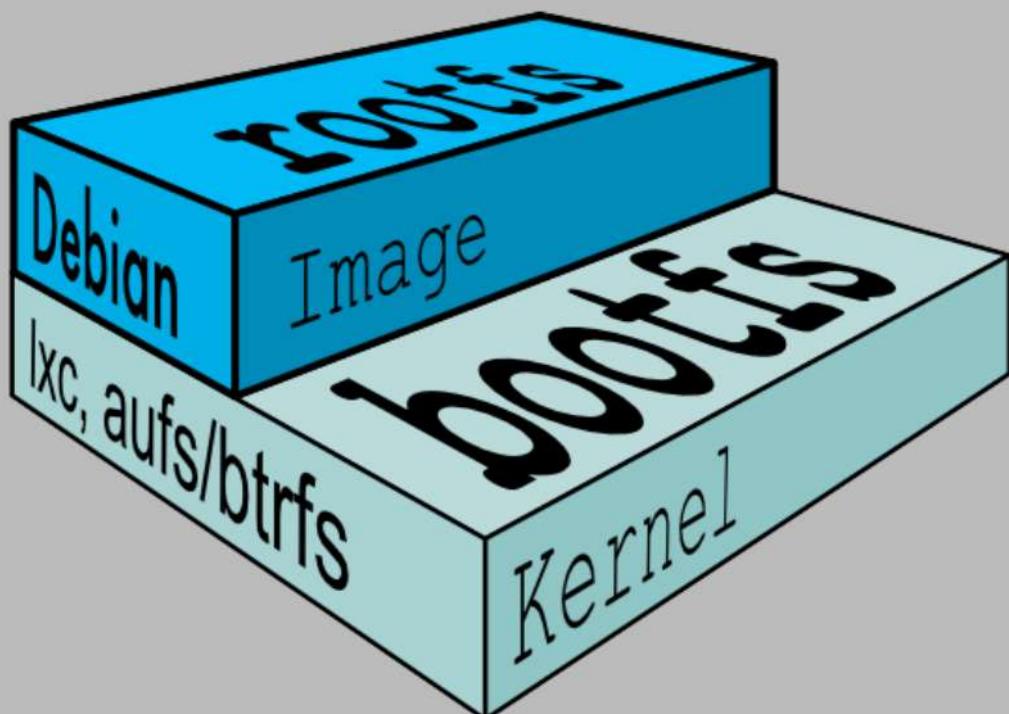
由此可见对于不同的linux发行版，bootfs基本是一致的，rootfs会有差别，因此不同的发行版可以公用bootfs 如下图：



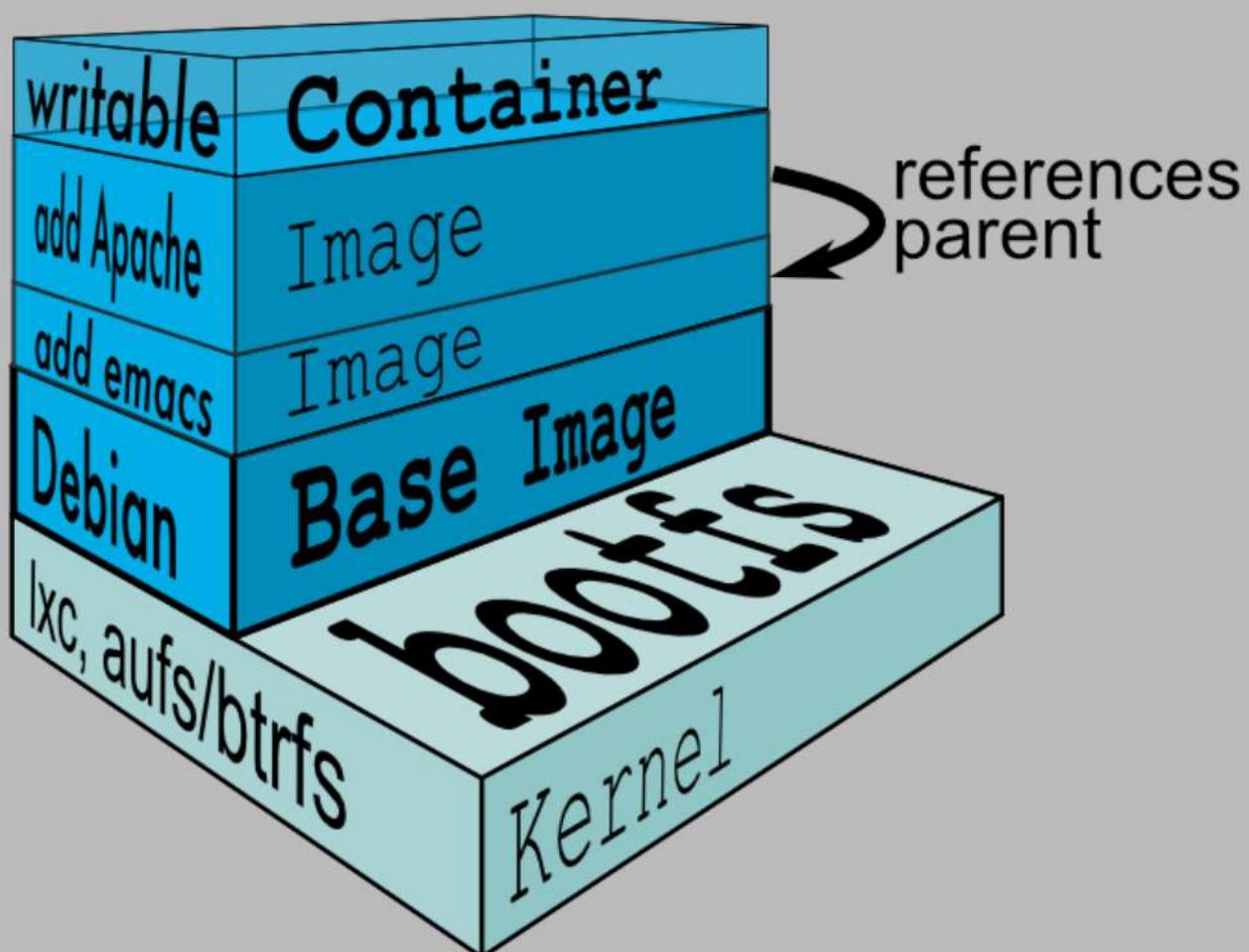
典型的Linux在启动后，首先将 rootfs 置为 `readonly`，进行一系列检查，然后将其切换为 "`readwrite`" 供用户使用。在docker中，起初也是将 rootfs 以`readonly`方式加载并检查，然而接下来利用 `union mount` 的将一个 `readwrite` 文件系统挂载在 `readonly` 的rootfs之上，并且允许再次将下层的 `file system` 设定为`readonly` 并且向上叠加，这样一组`readonly`和一个`writeable`的结构构成一个container的运行目录，每一个被称作一个Layer。如下图：



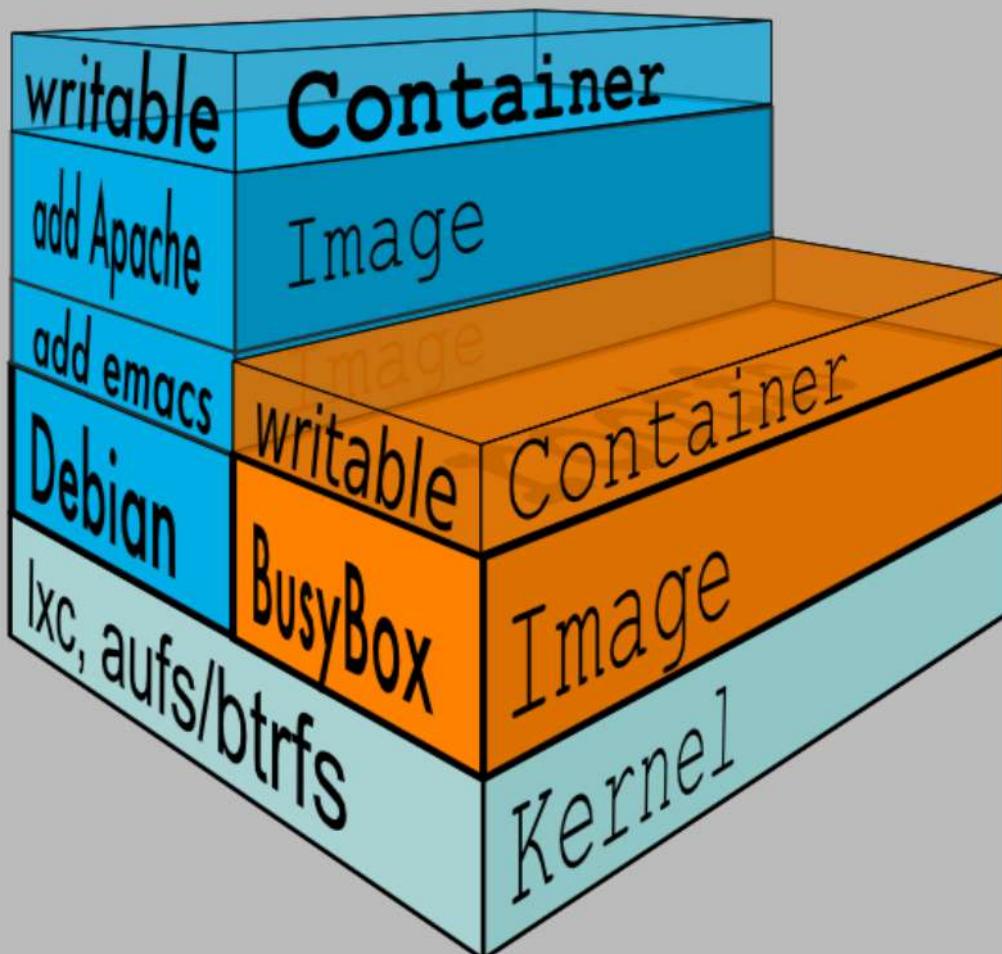
得益于AUFS的特性, 每一个对readonly层文件/目录的修改都只会存在于上层的**writable**层中。这样由于不存在竞争, 多个container可以共享**readonly**的layer。所以docker将**readonly**的层称作 "**image**" - 对于**container**而言整个**rootfs**都是**read-write**的, 但事实上所有的修改都写入最上层的**writable**层中, **image**不保存用户状态, 可以用于模板、重建和复制。



上层的image依赖下层的image，因此docker中把下层的image称作父image，没有父image的image称作base image



因此想要从一个image启动一个container，docker会先加载其父image直到base image，用户的进程运行在writeable的layer中。所有parent image中的数据信息以及ID、网络和lxc管理的资源限制等具体container的配置，构成一个docker概念上的container。如下图：



由此可见，采用AUFS作为docker的container的文件系统，能够提供如下好处：

1. 节省存储空间 - 多个container可以共享base image存储
2. 快速部署 - 如果要部署多个container，base image可以避免多次拷贝
3. 内存更省 - 因为多个container共享base image，以及OS的disk缓存机制，多个container中的进程命中缓存内容的几率大大增加
4. 升级更方便 - 相比于 copy-on-write 类型的FS，base-image也是可以挂载为可writeable的，可以通过更新base image而一次性更新其之上的container
5. 允许在不更改base-image的同时修改其目录中的文件 - 所有写操作都发生在最上层的writable层中，这样可以大大增加base image能共享的文件内容。

以上5条 1-3 条可以通过 copy-on-write 的FS实现，4可以利用其他的union mount方式实现，5只有AUFS实现的很好。这也是为什么Docker一开始就建立在AUFS之上。

由于AUFS并不会进入linux主干 (According to Christoph Hellwig, linux rejects all union-type filesystems but UnionMount.)，同时要求kernel版本3.0以上(docker推荐3.8及以上)，因此在RedHat工程师的帮助下在 docker0.7版本中实现了driver机制，AUFS只是其中的一个driver，在RHEL中采用的则是Device Mapper的方式实现的container文件系统，相关内容在下文会介绍。

参考文献

[1]<https://groups.google.com/forum/#!topic/docker-dev/KcCT0bACksY>
<https://groups.google.com/forum/#!topic/docker-dev/KcCT0bACksY>

[2]<http://blog.docker.io/2013/11/docker-0-7-docker-now-runs-on-any-linux-distribution/>
(<http://blog.docker.io/2013/11/docker-0-7-docker-now-runs-on-any-linux-distribution/>)

[3]<http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-34-a> (<http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-34-a>)

[4]<http://aufs.sourceforge.net/aufs.html> (<http://aufs.sourceforge.net/aufs.html>)

[5]<http://aufs.sourceforge.net/> (<http://aufs.sourceforge.net/>)

[6]<http://en.wikipedia.org/wiki/Aufs> (<http://en.wikipedia.org/wiki/Aufs>)

[7]<http://docs.docker.io/en/latest/terms/filesystem/> (<http://docs.docker.io/en/latest/terms/filesystem/>)

[8]<http://docs.docker.io/en/latest/terms/layer/> (<http://docs.docker.io/en/latest/terms/layer/>)

[9]<http://docs.docker.io/en/latest/terms/image/> (<http://docs.docker.io/en/latest/terms/image/>)

[10]<http://docs.docker.io/en/latest/terms/container/> (<http://docs.docker.io/en/latest/terms/container/>)

GRSEC

`grsec` 是linux kernel安全相关的patch, 用于保护host防止非法入侵。由于其并不是docker的一部分, 我们只进行简单的介绍。`grsec` 可以主要从4个方面保护进程不被非法入侵:

- 随机地址空间 - 进程的堆区地址是随机的
- 用只读的memory management unit来管理进程流程, 堆区和栈区内存只包含数据结构/函数/返回地址和数据, 是non-executable
- 审计和Log可疑活动
- 编译期的防护

安全永远是相对的, 这些方法只是告诉我们可以从这些角度考虑container类型的安全问题可以关注的方面。

参考文献

[1] <http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-44-g> (<http://blog.dotcloud.com/kernel-secrets-from-the-paas-garage-part-44-g>)

[2] <http://grsecurity.net/> (<http://grsecurity.net/>)

What docker do more than LXC

看似docker主要的OS级虚拟化操作是借助LXC, AUFS只是锦上添花。那么肯定会有好奇docker到底比LXC多了些什么。无意中发现 stackoverflow 上正好有人问这个问题, 回答者是Dotcloud的创始人, 出于备忘目的原文摘录如下。

<http://stackoverflow.com/questions/17989306/what-does-docker-add-to-just-plain-lxc>
(<http://stackoverflow.com/questions/17989306/what-does-docker-add-to-just-plain-lxc>)

On top of this low-level foundation of kernel features, Docker offers a high-level tool with several powerful functionalities:

- Portable deployment across machines. Docker defines a format for bundling an application and all its dependencies into a single object which can be transferred to any docker-enabled machine, and executed there with the guarantee that the execution environment exposed to the application will be the same. Lxc implements process sandboxing, which is an important pre-requisite for portable deployment, but that

alone is not enough for portable deployment. If you sent me a copy of your application installed in a custom lxc configuration, it would almost certainly not run on my machine the way it does on yours, because it is tied to your machine's specific configuration: networking, storage, logging, distro, etc. Docker defines an abstraction for these machine-specific settings, so that the exact same docker container can run - unchanged - on many different machines, with many different configurations.

- Application-centric. Docker is optimized for the deployment of applications, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the lxc helper scripts focus on containers as lightweight machines - basically servers that boot faster and need less ram. We think there's more to containers than just that.
- Automatic build. Docker includes a tool for developers to automatically assemble a container from their source code, with full control over application dependencies, build tools, packaging etc. They are free to use make, maven, chef, puppet, salt, debian packages, rpms, source tarballs, or any combination of the above, regardless of the configuration of the machines.
- Versioning. Docker includes git-like capabilities for tracking successive versions of a container, inspecting the diff between versions, committing new versions, rolling back etc. The history also includes how a container was assembled and by whom, so you get full traceability from the production server all the way back to the upstream developer. Docker also implements incremental uploads and downloads, similar to "git pull", so new versions of a container can be transferred by only sending diffs.
- Component re-use. Any container can be used as an "base image" to create more specialized components. This can be done manually or as part of an automated build. For example you can prepare the ideal python environment, and use it as a base for 10 different applications. Your ideal postgresql setup can be re-used for all your future projects. And so on.
- Sharing. Docker has access to a public registry (<http://index.docker.io>) where thousands of people have uploaded useful containers: anything from redis, couchdb, postgres to irc bouncers to rails app servers to hadoop to base images for various distros. The registry also includes an official "standard library" of useful containers maintained by the docker team. The registry itself is open-source, so anyone can deploy their own registry to store and transfer private containers, for internal server deployments for example.
- Tool ecosystem. Docker defines an API for automating and customizing the creation and deployment of containers. There are a huge number of tools integrating with docker to extend its capabilities. PaaS-like deployment (Dokku, Deis, Flynn), multi-node orchestration (maestro, salt, mesos, openstack nova), management dashboards (docker-ui, openstack horizon, shipyard), configuration management (chef, puppet), continuous integration (jenkins, strider, travis), etc. Docker is rapidly establishing itself as the standard for container-based tooling.

What we can do with Docker

有了docker这么个强有力的工具，更多的玩家希望了解围绕docker能做什么

Sandbox

作为sandbox大概是container的最基本想法了 - 轻量级的隔离机制, 快速重建和销毁, 占用资源少。用docker在开发者的单机环境下模拟分布式软件部署和调试, 可谓又快又好。同时docker提供的版本控制和image机制以及远程image管理, 可以构建类似git的分布式开发环境。可以看到用于构建多平台image的packer (<http://www.packer.io/>)以及同一作者的vagrant (<http://www.vagrantup.com/>)已经在这方面有所尝试了, 笔者会后续的blog中介绍这两款来自同一geek的精致小巧的工具。

PaaS

dotcloud、heroku以及cloudfoundry都试图通过container来隔离提供给用户的runtime和service，只不过dotcloud采用docker, heroku采用LXC, cloudfoundry采用自己开发的基于cgroup的warden。基于轻量级的隔离机制提供给用户PaaS服务是比较常见的做法 - PaaS 提供给用户的并不是OS而是runtime+service, 因此OS级别的隔离机制向用户屏蔽的细节已经足够。而docker的很多分析文章提到『能够运行任何应用的“PaaS”云』只是从image的角度说明docker可以从通过构建image实现用户app的打包以及标准服务service image的复用, 而非常见的buildpack的方式。

由于对Cloud Foundry和docker的了解, 接下来谈谈笔者对PaaS的认识。PaaS号称的platform一直以来都被当做一组多语言的runtime和一组常用的middleware, 提供这两样东西即可被认为是一个满足需求的PaaS。然而PaaS对能部署在其上的应用要求很高:

- 运行环境要简单 - buildpack虽然用于解决类似问题, 但仍然不是很理想
- 要尽可能的使用service - 常用的mysql, apache倒能理解, 但是类似log之类的如果也要用service就让用户接入PaaS平台, 让用户难以维护
- 要尽可能的使用"平台" - 单机环境构建出目标PaaS上运行的实际环境比较困难, 开发测试工作都离不开"平台"
- 缺少可定制性 - 可选的中间件有限, 难于调优和debug。

综上所述部署在PaaS上的应用几乎不具有从老平台迁移到之上的可能, 新应用也难以进入参数调优这种深入的工作。个人理解还是适合快速原型的展现, 和短期应用的尝试。

然而docker确实从另一个角度(类似IaaS+orchestration tools)实现了用户运行环境的控制和管理, 然而又基于轻量级的LXC机制, 确实是一个了不起的尝试。笔者也认为IaaS + 灵活的orchestration tools(深入到app层面的管理如bosh)是交付用户环境最好的方式。

Open Solution

前文也提到docker存在disk/network不便限额和在较低版本kernel中(如RHEL的2.6.32)AUFS不支持的问题。本节尝试给出解答。

disk/network quota

虽然cgroup提供IOPS之类的限制机制, 但是从限制用户能使用的磁盘大小和网络带宽上还是非常有限的。

Disk/network的quota现在有两种思路:

- 通过docker run -v命令将外部存储mount到container的目录下, quota从Host方向限制, 在device mapper driver中更采用实际的device因此更好控制。参考[1]
- 通过使用disk quota来限制AUFS的可操作文件大小。类似cloud foundry warden的方法, 维护一个UID池, 每次创建container都从中取一个user name, 在container里和Host上用这个username创建用户, 在Host上用setquota限制该username的UID的disk。网络上由于docker采用veth的方式, 可以采用tc来控制host上的veth的设备。参考[2]

参考文献:

[1]<https://github.com/dotcloud/docker/issues/111> (<https://github.com/dotcloud/docker/issues/111>)

[2]<https://github.com/dotcloud/docker/issues/471> (<https://github.com/dotcloud/docker/issues/471>)

RHEL 6.5

这里简单介绍下device mapper driver的思路，参考文献[2]中的讨论非常有价值。docker的dirver要利用snapshot机制，起初的fs是一个空的ext4的目录，然后写入每个layer。每次创建image其实就是对其父image/base image进行snapshot，然后在此snapshot上的操作都会被记录在fs的metadata中和AUFS layer(没读代码不是很理解?)，`docker commit` 将 diff信息在parent image上执行一遍。这样创建出来的image就可以同当前container的运行环境分离开独立保存了。

这里仅仅查看材料理解不是很透彻，还是需要深入代码去了解详情。贴出 mail list 的片段，如果有理解的请不吝赐教。

The way it works is that we set up a device-mapper thin provisioning pool with a single base device containing an empty ext4 filesystem. Then each time we create an image we take a snapshot of the parent image (or the base image) and manually apply the AUFS layer to this. Similarly we create snapshots of images when we create containers and mount these as the container filesystem.

"docker diff" is implemented by just scanning the container filesystem and the parent image filesystem, looking at the metadata for changes. Theoretically this can be fooled if you do in-place editing of a file (not changing the size) and reset the mtime/ctime, but in practice I think this will be good enough.

"docker commit" uses the above diff command to get a list of changed files which are used to construct a tarball with files and AUFS whiteouts (for deletes). This means you can commit containers to images, run new containers based on the image, etc. You should be able to push them to the index too (although I've not tested this yet).

Docker looks for a "docker-pool" device-mapper device (i.e. /dev/mapper/docker-pool) when it starts up, but if none exists it automatically creates two sparse files (100GB for the data and 2GB for the metadata) and loopback mount these and sets these up as the block devices for docker-pool, with a 10GB ext4 fs as the base image.

This means that there is no need for manual setup of block devices, and that generally there should be no need to pre-allocate large amounts of space (the sparse files are small, and we things up so that discards are passed through all the way back to the sparse loopbacks, so deletes in a container should fully reclaim space.

目前已知存在的问题是删除的image的 block 文件没有被删除，

见<https://github.com/dotcloud/docker/issues/3182> (<https://github.com/dotcloud/docker/issues/3182>)，笔者发现此问题前4个小时作者给出了原因，看起来是kernel的issue，在讨论中包含work around的方法。

参考文献：

[1]<http://blog.docker.io/2013/11/docker-0-7-docker-now-runs-on-any-linux-distribution/>
 (<http://blog.docker.io/2013/11/docker-0-7-docker-now-runs-on-any-linux-distribution/>)

[2]<https://groups.google.com/forum/#topic/docker-dev/KcCT0bACksY>
 (<https://groups.google.com/forum/#topic/docker-dev/KcCT0bACksY>)

Summary

本文总结了以下几点内容

1. docker的介绍，包括由来、适用场景等
2. docker背后的一系列技术 - namespace, cgroup, lxc, aufs等
3. docker在利用LXC的同时提供了哪些创新
4. 笔者对docker这种container, PaaS的一些理解
5. docker存在的问题和现有的解决思路

希望能对想要了解docker的朋友有所帮助，更细致的了解还是得深入代码，了解个中原委。

docker@github - <https://github.com/dotcloud/docker> (<https://github.com/dotcloud/docker>)

docker_maillist - <https://groups.google.com/forum/#!forum/docker-dev>
(<https://groups.google.com/forum/#!forum/docker-dev>)

cloud ² (/categories.html#cloud-ref)

Docker ¹ (/tags.html#Docker-ref)

← Previous (/log/LOG-Cisco-UCS-Baremetal-Driver)

Archive (/archive.html)

Next → (/devops/howto-use-cgroup)

Create an Issue () or comment below

社交帐号登录: [微博](#) [QQ](#) [人人](#) [豆瓣](#) [更多»](#)



说点什么吧...

发布

0条评论

最新 最早 最热

还没有评论，沙发等你来抢

TechRest正在使用多说 (<http://duoshuo.com>)

© 2014 Tie Wei (/about.html) All rights reserved.