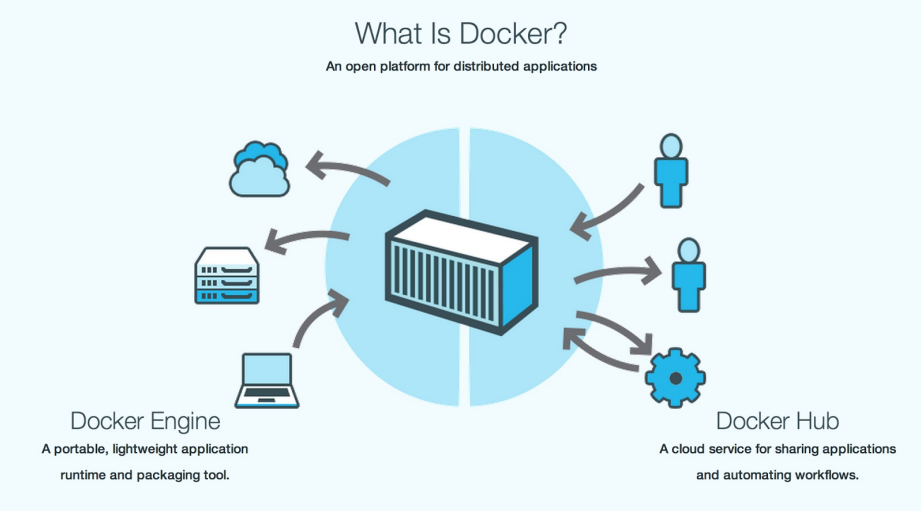
深入浅出Docker（一）：Docker核心技术预览

1. 背景

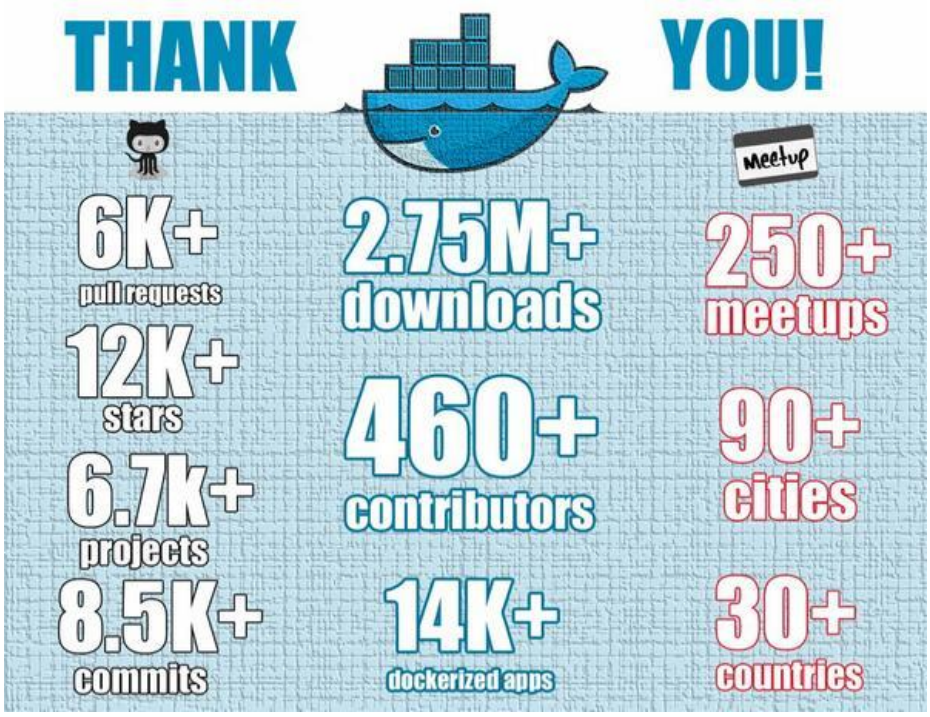
1.1. 由PaaS到Container

2013年2月，前Gluster的CEO Ben Golub和dotCloud的CEO Solomon Hykes坐在一起聊天时，Solomon谈到想把dotCloud内部使用的Container容器技术单独拿出来开源，然后围绕这个技术开一家新公司 提供技术支持。28岁的Solomon在使用python开发dotCloud的PaaS云时发现，使用 LXC(Linux Container) 技术可以打破产品发布过程中应用开发工程师和系统工程师两者之间无法轻松协作发布产品的难题。这个Container容器技术可以把开发者从日常部署应用 的繁杂工作中解脱出来，让开发者能专心写好程序；从系统工程师的角度来看也是一样，他们迫切需要从各种混乱的部署文档中解脱出来，让系统工程师专注在应用 的水平扩展、稳定发布的解决方案上。他们越深入交谈，越觉得这是一次云技术的变革，紧接着在2013年3月Docker 0.1发布，拉开了基于云计算平台发布产品方式的变革序幕。

1.2 Docker简介

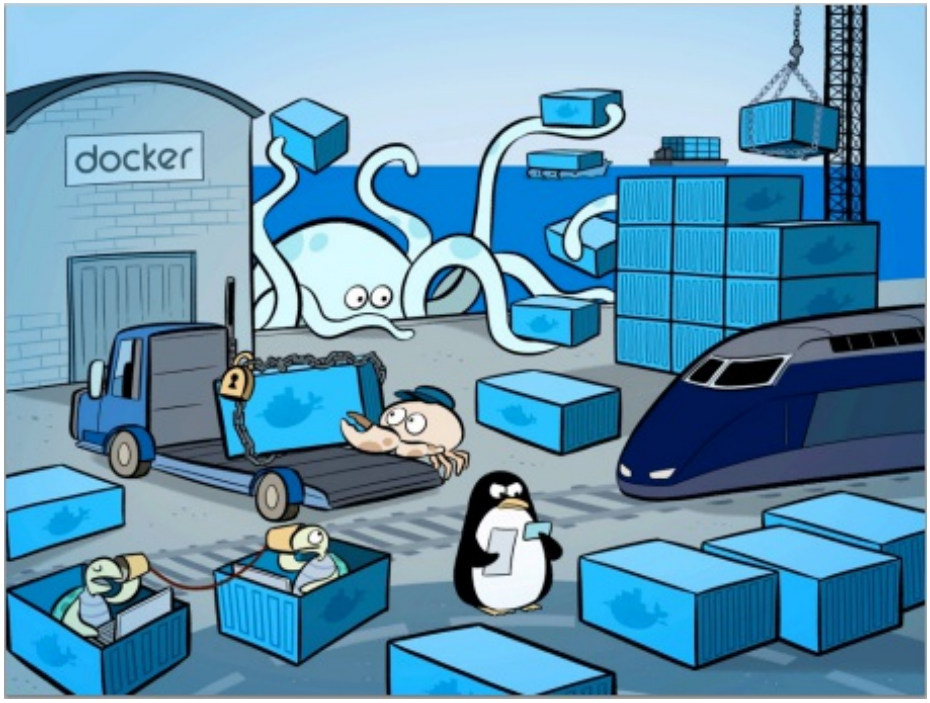


Docker 是 Docker.Inc 公司开源的一个基于 LXC技术之上构建的Container容器引擎， 源代码托 管在 GitHub 上, 基于Go语言并遵从Apache2.0协议开源。 Docker在2014年6月召开DockerConf 2014技术大会吸引了IBM、Google、RedHat等业界知名公司的关注和技术支持，无论是从 GitHub 上的代码活跃度，还是Redhat宣布在RHEL7中正式支持Docker, 都给业界一个信号，这是一项创新型的技术解决方案。 就连 Google 公司的 Compute Engine 也支持 docker 在其之上运行, 国内“BAT”先锋企业百度Baidu App Engine(BAE)平台也是以Docker作为其PaaS云基础。



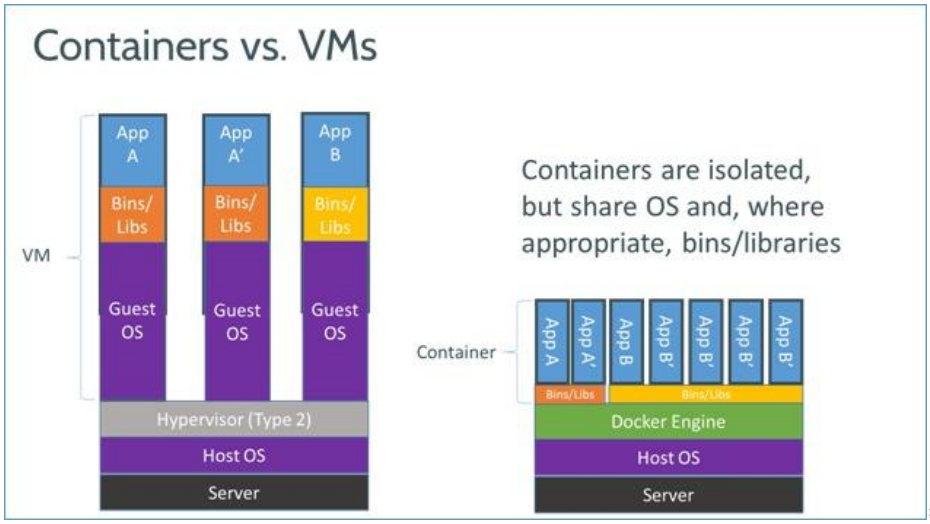
Docker产生的目的就是为了解决以下问题:

1) 环境管理复杂: 从各种OS到各种中间件再到各种App，一款产品能够成功发布，作为开发者需要关心的东西太多，且难于管理，这个问题在软件行业中普遍存在并需要直接面 对。Docker可以简化部署多种应用实例工作，比如Web应用、后台应用、数据库应用、大数据应用比如Hadoop集群、消息队列等等都可以打包成一个 Image部署。如图所示：



2) 云计算时代的到来: AWS的成功, 引导开发者将应用转移到云上, 解决了硬件管理的问题，然而软件配置和管理相关的问题依然存在 (AWS cloudformation是这个方向的业界标准, 样例模板可参考这里)。Docker的出现正好能帮助软件开发者开阔思路，尝试新的软件管理方法来解决这个问题。

3) 虚拟化手段的变化: 云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分配的资源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。如图所示：



=

4) LXC的便携性: 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便携性，决定其构建出的环境难于分发和标准化管理(相对于KVM之类 image和snapshot的概念)。Docker就在这个问题上做出了实质性的创新方法。

1.3 Docker的Hello World

以Fedora 20作为主机为例，直接安装docker-io：

```
$ sudo yum -y install docker-io
```

启动docker后台Daemon:

```
$ sudo systemctl start docker
```

跑我们第一个Hello World容器：

```
$ sudo docker run -i -t fedora /bin/echo hello world
Hello world
```

可以看到在运行命令行后的下一行会打印出经典的Hello World字符串。

2. 核心技术预览

Docker核心是一个操作系统级虚拟化方法, 理解起来可能并不像VM那样直观。我们从虚拟化方法的四个方面：**隔离性、可配额/可度量、便携性、安全性**来详细介绍Docker的技术细节。

2.1. 隔离性: Linux Namespace(ns)

每个用户实例之间相互隔离, 互不影响。 一般的硬件虚拟化方法给出的方法是VM，而LXC给出的方法是container，更细一点讲就是kernel namespace。其中**pid、net、ipc、mnt、uts、user**等namespace将container的进程、网络、消息、文件系统、UTS("UNIX Time-sharing System")和用户空间隔离开。

1) **pid namespace**

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。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 Docker in Docker。

2) **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。docker默认采用veth的方式将container中的虚拟网卡同host上的一个docker bridge: docker0连接在一起。

3) **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 不同的是，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 32 位 ID。

4) **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。

5) **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。

6) **user namespace**

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

2.2 可配额/可度量 - 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。 groups可以限制blkio、cpu、cpuacct、cpuset、devices、freezer、memory、net_cls、ns九大子系 统的资源，以下是每个子系统的详细说明：

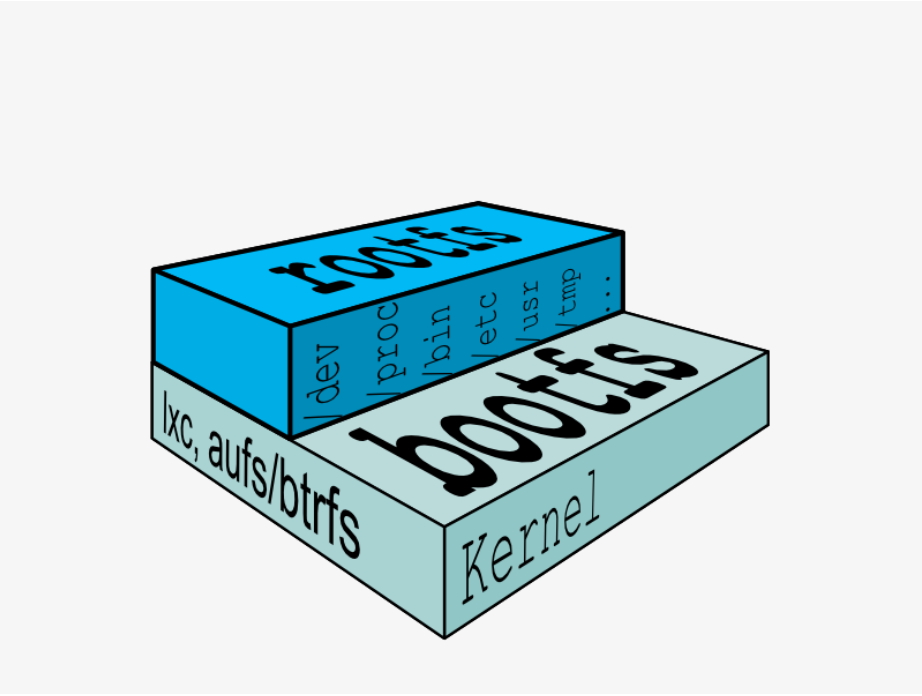
1. blkio 这个子系统设置限制每个块设备的输入输出控制。例如:磁盘，光盘以及usb等等。
2. cpu 这个子系统使用调度程序为cgroup任务提供cpu的访问。
3. cpuacct 产生cgroup任务的cpu资源报告。
4. cpuset 如果是多核心的cpu，这个子系统会为cgroup任务分配单独的cpu和内存。
5. devices 允许或拒绝cgroup任务对设备的访问。
6. freezer 暂停和恢复cgroup任务。
7. memory 设置每个cgroup的内存限制以及产生内存资源报告。
8. net_cls 标记每个网络包以供cgroup方便使用。
9. ns 名称空间子系统。

以上九个子系统之间也存在着一定的关系.详情请参阅官方文档。

2.3 便携性: AUFS

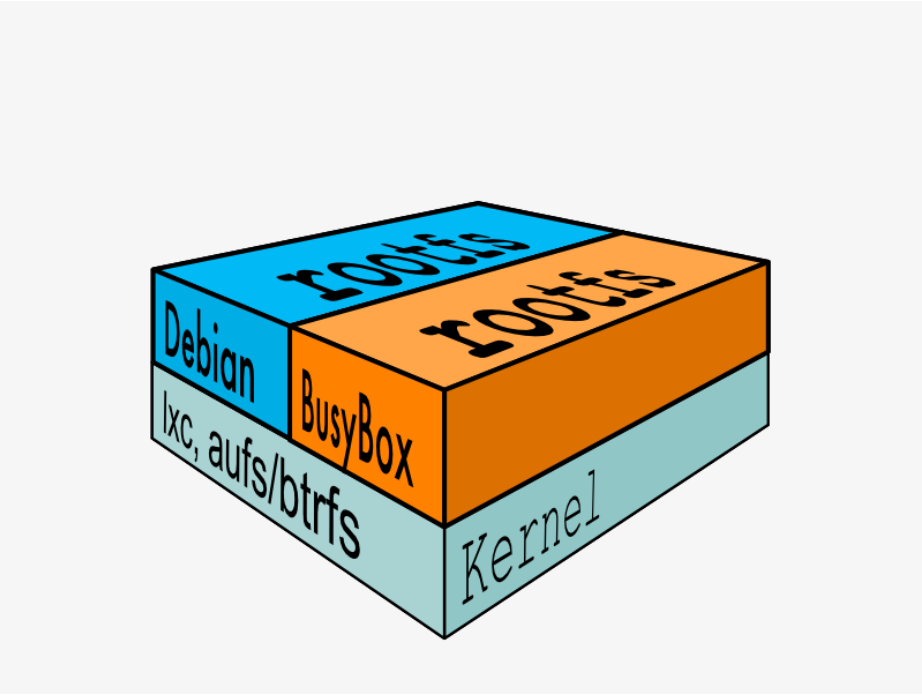
AUFS (AnotherUnionFS) 是一种 Union FS, 简单来说就是支持将不同目录挂载到同一个虚拟文件系统下(unite several directories into a single virtual filesystem)的文件系统, 更进一步的理解, AUFS支持为每一个成员目录(类似Git Branch)设定readonly、readwrite 和 whiteout-able 权限, 同时 AUFS 里有一个类似分层的概念, 对 readonly 权限的 branch 可以逻辑上进行修改(增量地, 不影响 readonly 部分的)。通常 Union FS 有两个用途, 一方面可以实现不借助 LVM、RAID 将多个disk挂到同一个目录下, 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正是基于此方法可以允许在 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 为例来介绍 docker 对AUFS特性的运用。

典型的启动Linux运行需要两个FS: bootfs + rootfs:

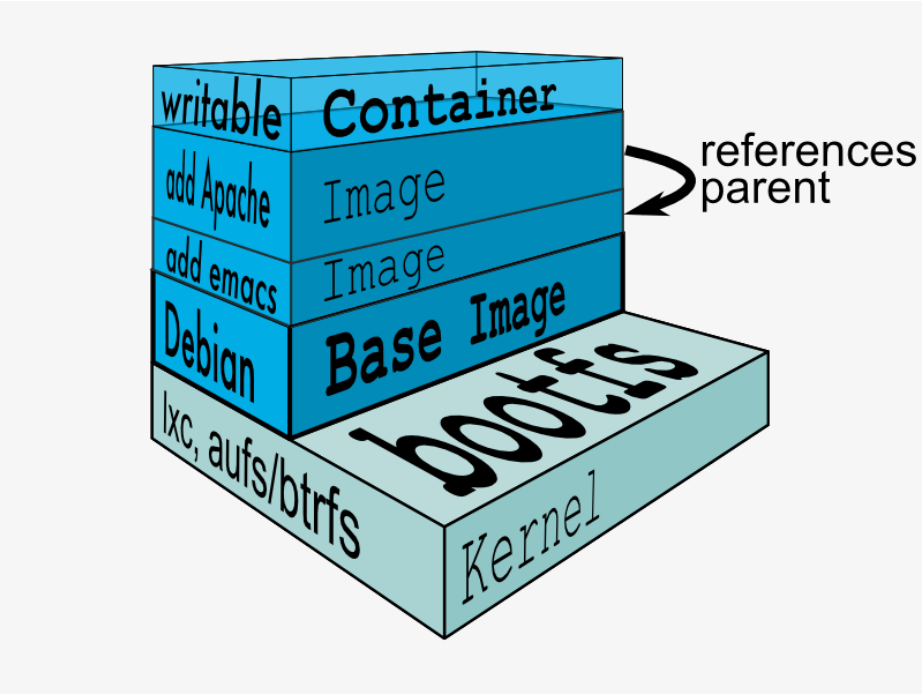


bootfs (boot file system) 主要包含 bootloader 和 kernel, bootloader主要是引导加载kernel, 当boot成功后 kernel 被加载到内存中后 bootfs就被umount了. rootfs (root file system) 包含的就是典型 Linux 系统中的 /dev, /proc,/bin, /etc 等标准目录和文件。

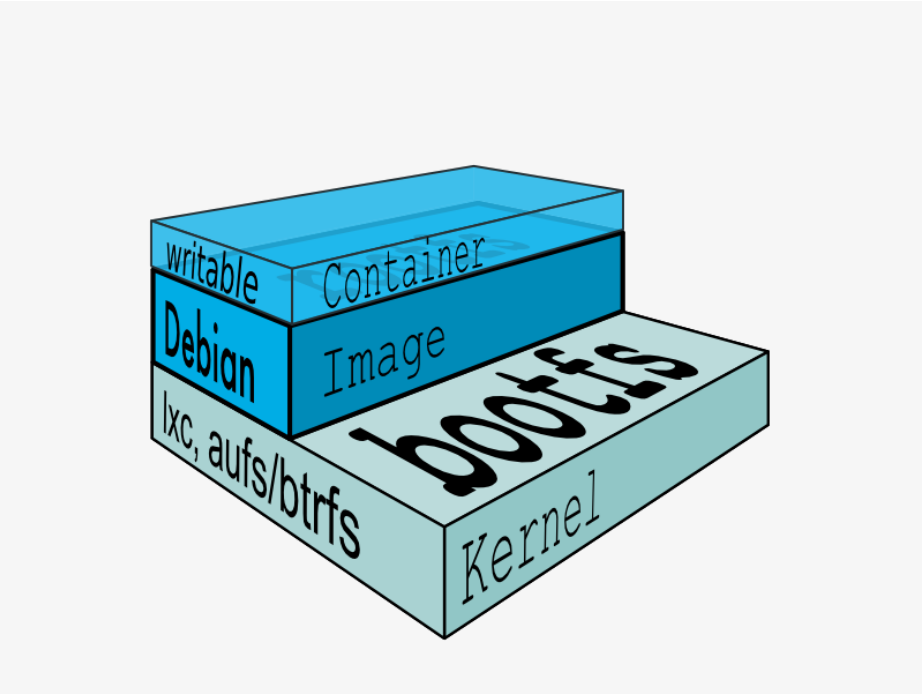
对于不同的linux发行版, bootfs基本是一致的, 但rootfs会有差别, 因此不同的发行版可以公用bootfs 如下图:



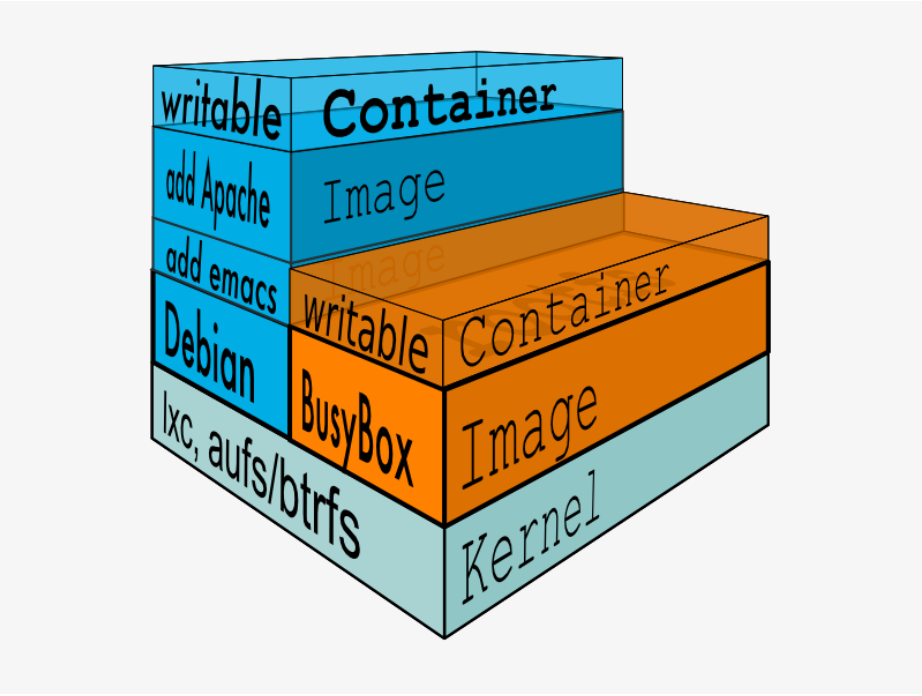
典型的Linux在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 "readwrite" 供用户使用。在Docker中，初始化时也是将 rootfs 以readonly方式加载并检查，然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上，并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加, 这样一组readonly和一个writeable的结构构成一个container的运行时态, 每一个FS被称作一个FS层。如下图:



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writeable层中。这样由于不存在竞争, 多个container可以共享readonly的FS层。 所以Docker将readonly的FS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writeable层中, image不保存用户状态，只用于模板、新建和复制使用。



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会先加载这个image和依赖的父images以及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:



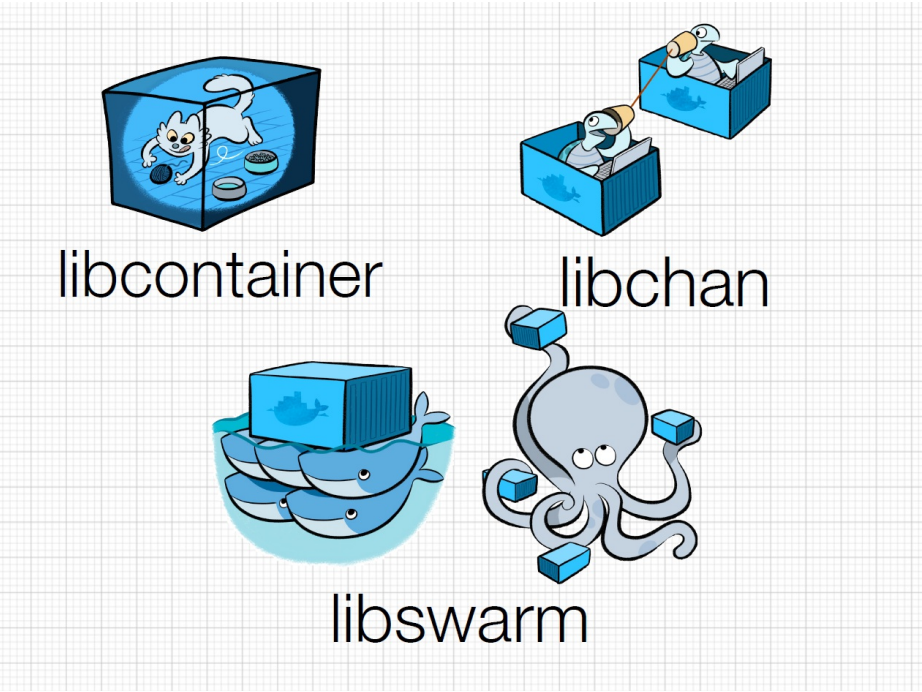
2.4 安全性: AppArmor, SELinux, GRSEC

安全永远是相对的，这里有三个方面可以考虑Docker的安全特性:

1. 由kernel namespaces和cgroups实现的Linux系统固有的安全标准;
2. Docker Deamon的安全接口;
3. Linux本身的安全加固解决方案,类如AppArmor, SELinux;

由于安全属于非常具体的技术，这里不在赘述，请直接参阅Docker官方文档。

3. 最新子项目介绍



我们再来看看Docker社区还有哪些子项目值得我们去好好研究和学习。基于这个目的，我把有趣的核心项目给大家罗列出来，让热心的读者能快速跟进自己感兴趣的项目:

1. Libswarm，是Solomon Hykes (Docker的CTO) 在DockerCon 2014峰会上向社区介绍的新“乐高积木”工具: 它是用来统一分布式系统的网络接口的API。Libswarm要解决的问题是，基于Docker构建的分布式应用已经催生了多个基于Docker的服务发 现(Serivce Discovery)项目，例如etcd, fleet, geard, mesos, shipyard, serf等等，每一

套解决方案都有自己的通讯协议和使用方法，使用其中的任意一款都会局限在某一个特定的技术范围内。所以Docker的CTO就想用 libswarm暴露出通用的API接口给分布式系统使用，打破既定的协议限制。目前项目还在早期发展阶段，值得参与。

2. Libchan，是一个底层的网络库，为上层 Libswarm 提供支持。相当于给Docker加上了ZeroMQ或RabbitMQ，这里自己实现网络库的好处是对Docker做了特别优化，更加轻量级。一般开发者 不会直接用到它，大家更多的还是使用Libswarm来和容器交互。喜欢底层实现的网络工程师可能对此感兴趣，不妨一看。
3. Libcontainer，Docker技术的核心部分，单独列出来也是因为这一块的功能相对独立，功能代码的迭代升级非常快。想了解Docker最新的支持特性应该多关注这个模块。

4. 总结

Docker社区一直在面对技术挑战，从容地给出自己的解决方案。云计算发展至今，有很多重要的问题没有得到妥善解决，Docker正在尝试让主流 厂商接受并应用它。至此，以上Docker技术的预览到此告一段落，笔者也希望读者能结合自己的实际情况，尝试使用Docker技术。因为只有在亲自体会 的基础之上，像Docker这样的云技术才会产生更大的价值。

5. 作者简介

肖德时, Red Hat Engineering Service/HSS 内部工具组Team Lead. Nodejs开源项目nodejs-cantas Lead Developer。擅长企业内部工具的设计以及实现。开源课程Rails Starter的发起人。rubygem: lazy_high_charts的Maintainer。twitter账号：xds2000，邮箱：xiaods@gmail.com

6. 参考文献:

1. https://tiewei.github.io/cloud/Docker-Getting-Start/
2. http://docs.docker.com/articles/
3. http://www.slideshare.net/shykes/docker-the-road-ahead
4. http://www.centurylinklabs.com/meet-docker-ceo-ben-golub/
5. http://lwn.net/Articles/531114/
6. http://en.wikipedia.org/wiki/Aufs
7. http://docs.docker.io/en/latest/terms/filesystem/
8. http://docs.docker.io/en/latest/terms/layer/
9. http://docs.docker.io/en/latest/terms/image/
10. http://docs.docker.io/en/latest/terms/container/
11. https://stackoverflow.com/questions/17989306/what-does-docker-add-to-just-plain-lxc