容器技术

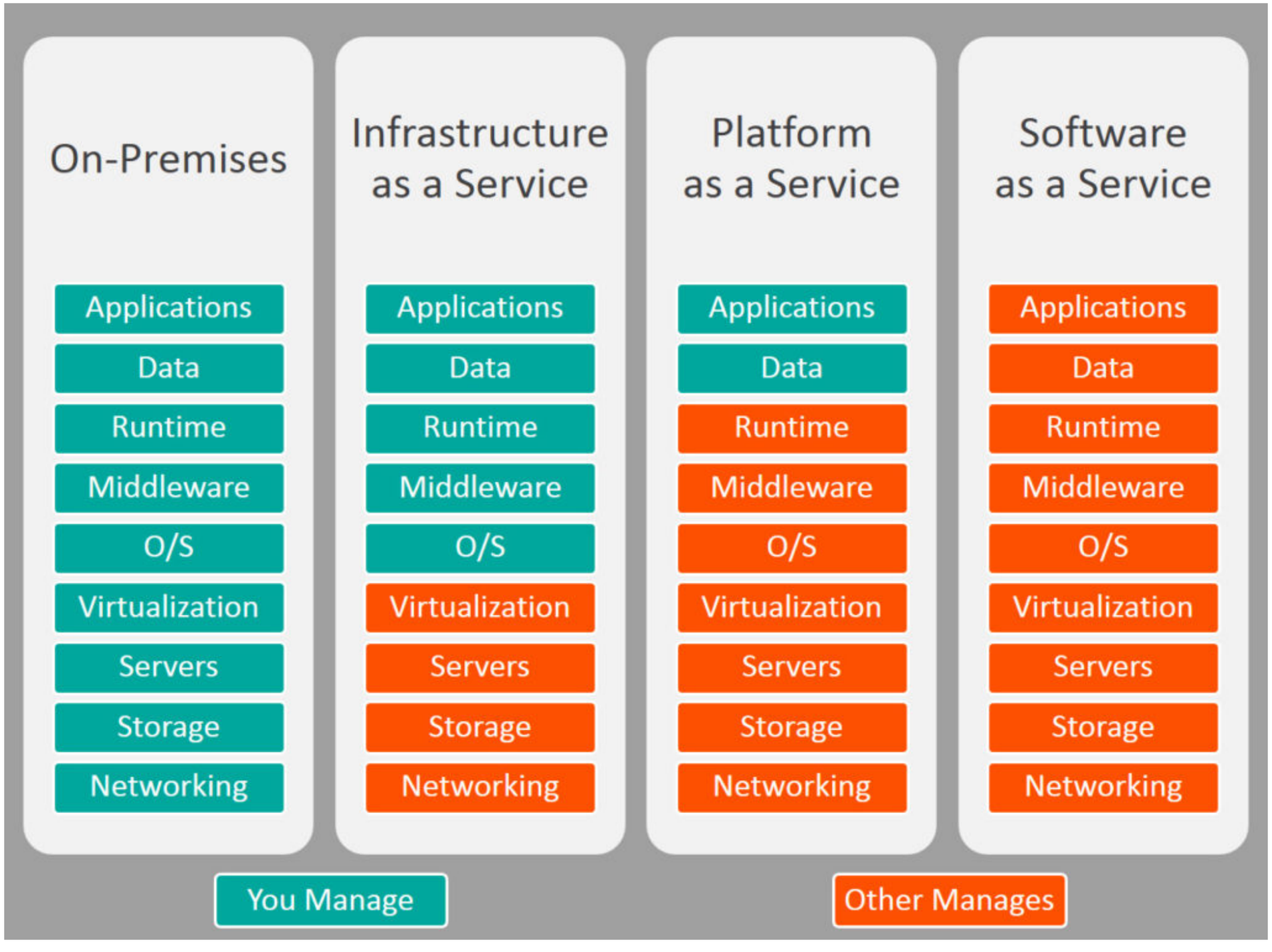
张晓宁

- PaaS时代
- 为什么是Docker?
- PaaS的两条腿
- 容器基础
- Docker的崛起对技术人员的影响
- 简单聊聊Kubernetes

PaaS时代

- 一张老图
- PaaS的价值
- 举个栗子
- 并不新鲜的沙盒技术

一张老图


On-Premises
Infrastructure as a Service
Platform as a Service
Software as a Service
Applications
Data
Runtime
Middleware
O/S
Virtualization
Servers
Storage
Networking
You Manage
Other Manages

PaaS的价值

- 2013年：相较于如日中天的AWS和OpenStack，以 Cloud Foundry 为代表的开源 PaaS项目逐渐被云计算从业者接受

- 各大厂开始构建以开源PaaS为核心的平台层服务能力

- PaaS项目被接纳的主要原因是它将原本的“服务器托管”，升级为“应用托管”

- 为了解决云端和本地环境不一致的问题，PaaS开源项目在当时是解决这个问题的最佳方案

举个栗子

- 首先在云主机上部署一个Cloud Foundry项目
- 然后执行cf push myapp就能将本地应用部署到云端
- Cloud Foundry的核心组件就是一套应用的打包和分发机制
- cf push的核心就是将应用和脚本打包传到云端，然后调度云主机Agent启动
- 重点来了，为了在一台云主机上启动多个应用，Cloud Foundry会使用操作系统的Cgroups和Namespace机制为每个应用创建单独的沙盒环境

举个栗子

Java Node Go Python PHP Ruby .NET

```
[INFO] Finished at: 2017-05-19T17:23:01+02:00
[INFO] Final Memory: 30M/313M
[INFO] ------------------------------------------------------------------------
%

evoila.de (cbrinker) ➜  demo cf push hello -d meshcloud.io -p target/demo-0.0.1-SNAPSHOT.jar
Creating app hello in org meshstack / space development-sb as admin...
OK

Creating route hello.meshcloud.io...
OK

Binding hello.meshcloud.io to hello...
OK

Uploading hello...
Uploading app files from: /var/folders/kc/2pt53r9j6kjf37w91xpmnt0r0000gn/T/unzipped-app522824198
Uploading 315.3K, 87 files
Done uploading
```

01:52

并不新鲜的沙盒技术

- Linux Namespaces起源于2002年的2.4.19内核，用于对内核资源进行分区，使一组进程看到一组资源，而另一组进程看到一组不同的资源

- 从内核版本4.10开始，Namespace分为7种：Mount/Process ID/Network/Interprocess Communication/UTS/User ID/Control Groups

- Control Groups最早由Google工程是2006年发起，名称叫process containers。2007年被重命名为cgroup，并且合并到2.6.24版内核中。用来限制、控制和分离一个进程组群的资源(CPU/内存/磁盘/网络/etc/等)

为什么是Docker?

- 传统PaaS项目的问题

- 小鲸鱼突围的秘密武器

传统PaaS项目的问题

- PaaS的应用托管能力的价值无需置疑，但是传统PaaS的打包能力是不断遭到用户吐槽的一个功能

- 打包的问题之一是用户需要为PaaS应用托管的每种语言和框架，甚至是每个版本维护一个打好的包，且这个打包过程没有任何章法可循

- 打包的问题之二是同一个包在本地运行的好好的，到PaaS里想要运行起来可能需要一个试错的过程

- 结果就成了一键部署一分钟，打包调试2小时

小鲸鱼突围的秘密武器

- Docker镜像巧妙的解决了打包调试2小时的问题
- 简而言之：Docker镜像是包含一个完整“操作系统”和应用运行环境的压缩包
- 比起传统PaaS的应用文件+脚本的组合要更加完整
- 这个压缩包实现了多个环境的高度一致
- docker build打包，docker run运行一个沙盒环境
- 对于传统PaaS项目，将开发者群体放在至高无上位置的Docker，简直就是“降维打击”
- 这是现今为止Cloud Native最完美的解决方案

PaaS的两条腿

- 老战场新打法

- 尘埃落定

老战场新打法

- Docker镜像虽然解放了开发者的生产力，但是并不能代替PaaS做大规模部署
- 随着Docker的迅速走红，PaaS以新名字CaaS迎来了一个新的高潮
- Cloud Foundry项目因为考虑到与Docker公司存在竞争关系，且错误的判断了Docker的普及程度，未及时使用Docker作为底层依赖。导致将自己辛苦教育的用户和市场拱手让人
- 反而一些嗅到新契机的创业公司在第一时间推出了Docker容器集群管理的开源项目
- Docker走红后，dotCloud公司在2013年将自己改名为Docker，2014年DockerCon上发布容器管理群集项目Swarm。此时的Docker公司离自己的人生巅峰只有一步之遥

老战场新打法

- 再次回到老问题上：如何让开发者部署到我的PaaS项目上?

- 相较于其他容器集群项目，Swarm是一个完整的整体来提供集群管理功能，而且可以直接使用原本的容器管理API来完成集群管理

- 此时不差钱的Docker公司还收购了容器编排项目Fig，也就是现在大家认识的Compose，加上Swarm的集群管理能力组成一个完整的PaaS项目

- 同时Mesos项目天生的两层调度机制，让它非常容易从大数据领域转而支持PaaS业务，靠着超大规模集群管理经验，很快就成了Swarm的有力竞争对手

- 2014年注定是个神奇的年份，上面这俩货打的欢实的时候，Google爸爸赶到了战场。没错，一个叫Kubernetes的王者即将诞生

尘埃落定

- Docker公司激进的商业化进程开始逐渐引起社区和其他玩家的不满
- 2015年不满情绪达到了高潮，其他玩家发起了OCI组织希望将容器运行时和镜像的实现从Docker项目中完全剥离，来改变Docker公司一家独大的现状。未遂
- 几个小伙伴又发起了CNCF基金会，以Kubernetes项目为基础，建立开源基础设施领域厂商主导的，独立运营的PaaS社区，来对抗Docker公司的商业生态
- 基于Google多年容器化基础设施经验的沉淀Borg，Kubernetes一开始就以“超前”的设计思想避免了与Swarm和Mesos同质化竞争，很快就变成了真香系列
- 惊慌失措的Docker更加激进的在2016年宣布放弃Swarm，将容器编排和集群管理全部内置到Docker项目当中。然而这种违反软件工程原则的做法，注定不会有什么好结果
- Kubernetes的应对策略是反其道而行之，以更加民主化的架构推向社区。很快就在容器社区中催生出大量的二次创新

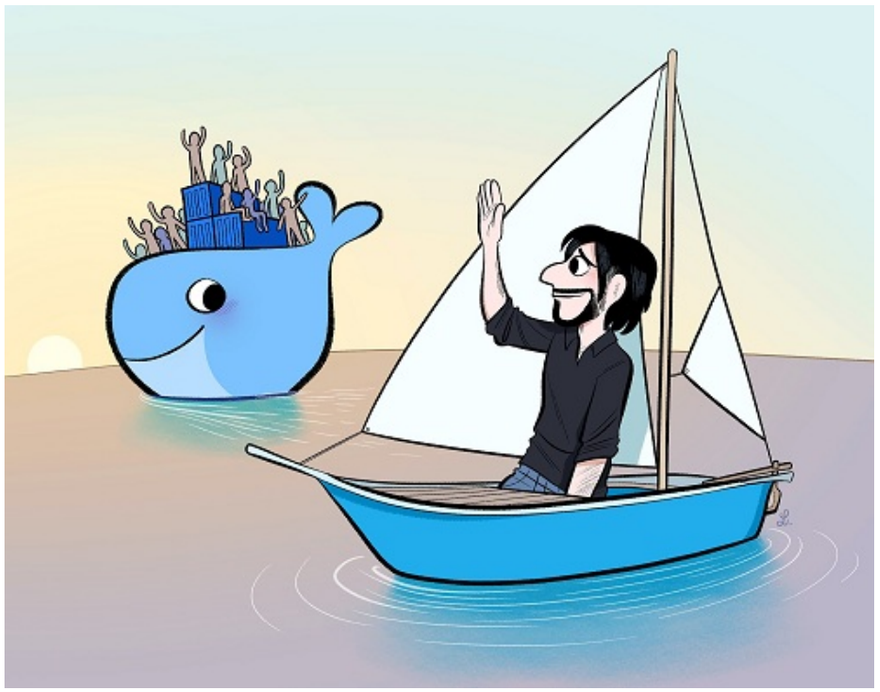
尘埃落定

- 随着Kubernetes的崛起，摸奖失败的Docker公司2017年将Docker项目的运行时捐赠给CNCF社区，并将Docker项目改名为Moby交给社区维护，Docker公司商业产品占有Docker这个商标

- 2017年10月Docker公司项目将在自己主打产品Docker企业版中内置Kubernetes项目

- 2018年3月Docker公司CTO Solomon Hykes辞职

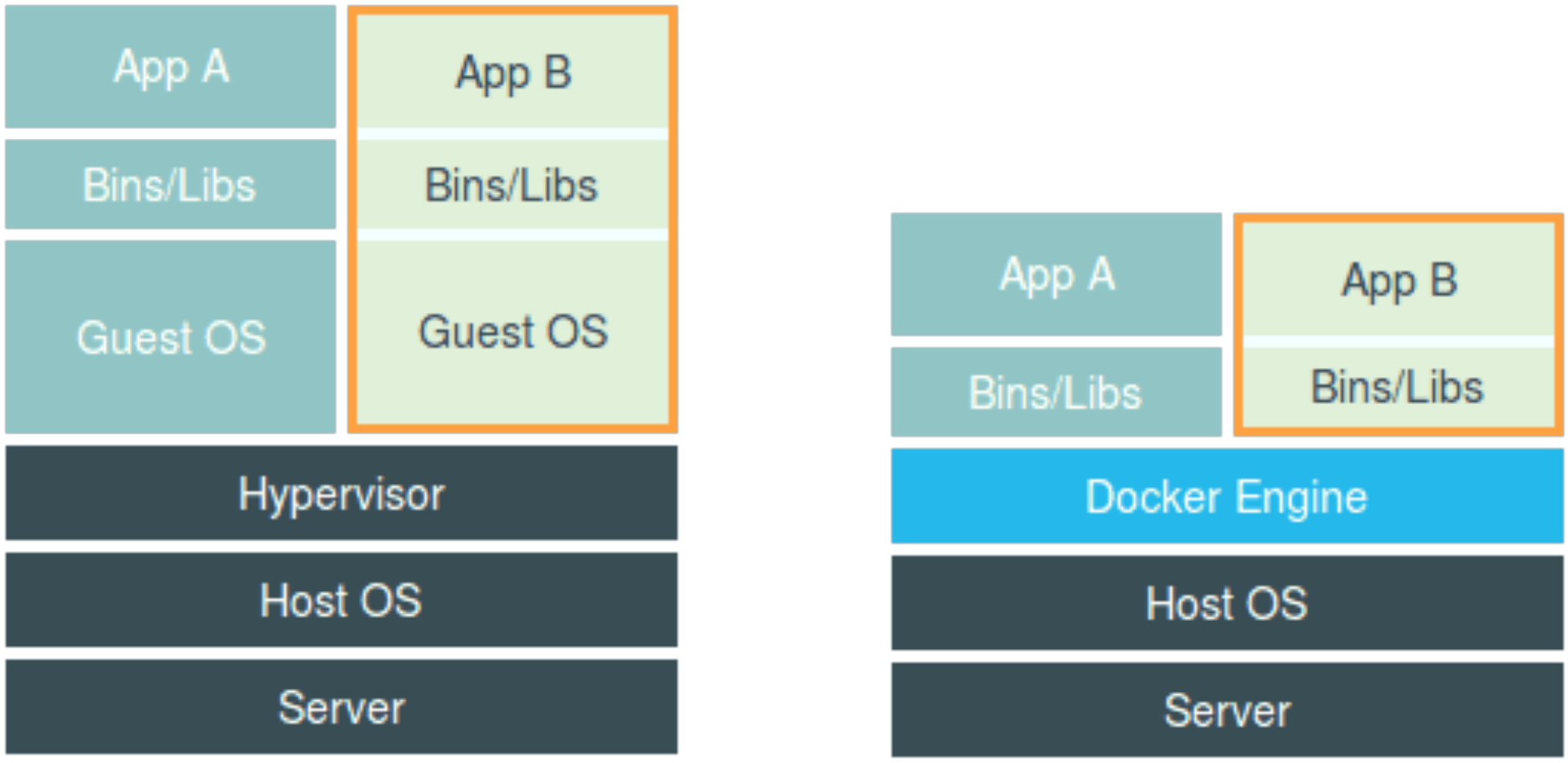
- 容器编排的价值已经显而易见



容器基础

- 容器的边界
- 容器镜像
- 真正的认识Docker

容器的边界


App A
Bins/Libs
Guest OS
App B
Bins/Libs
Guest OS
Hypervisor
Host OS
Server
App A
Bins/Libs
App B
Bins/Libs
Docker Engine
Host OS
Server

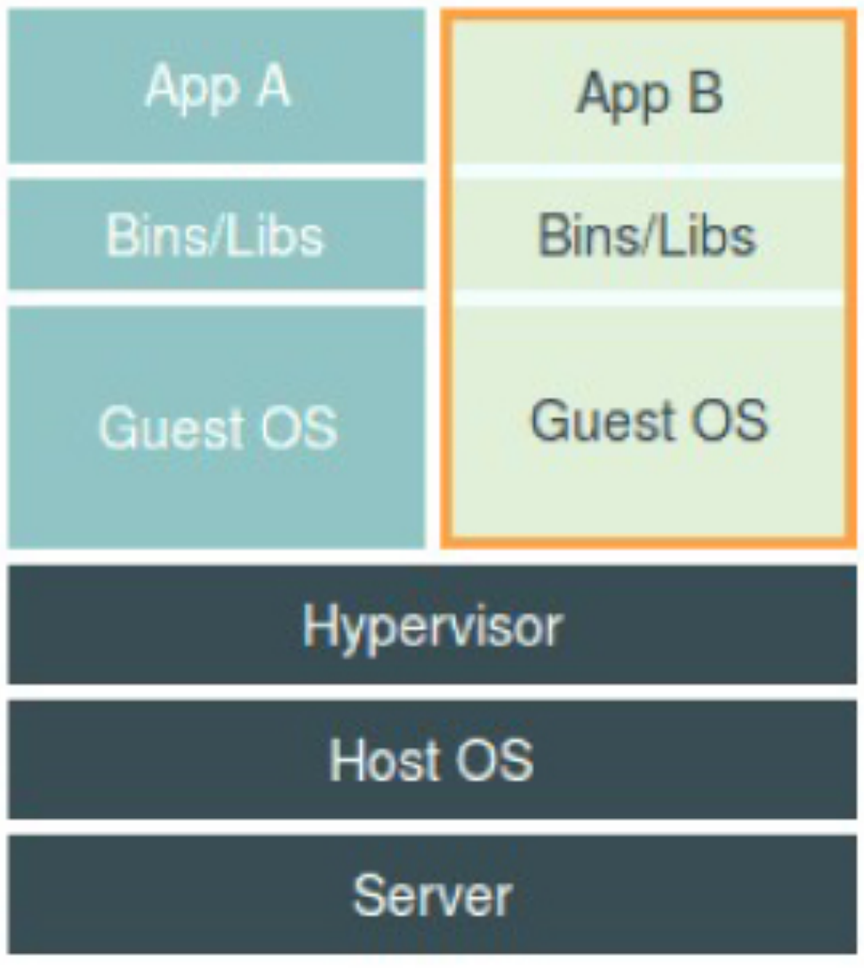
容器的边界

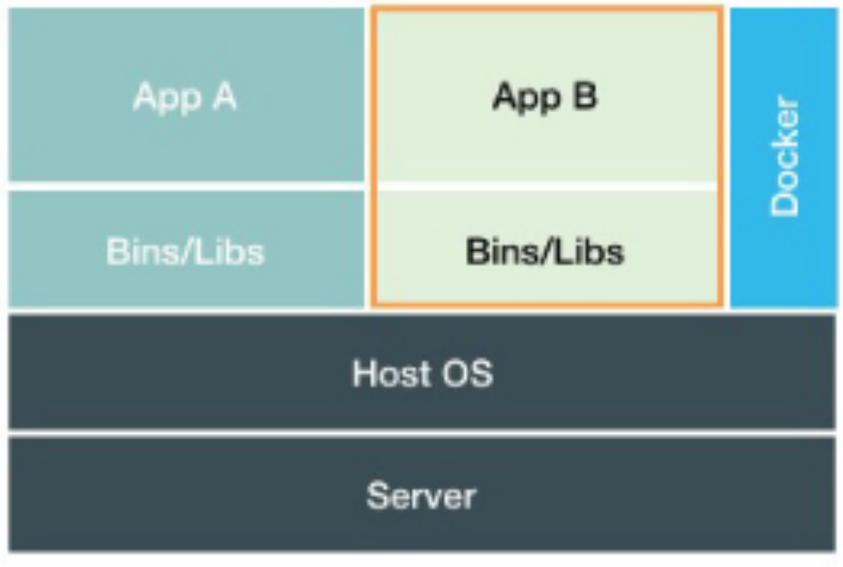
- Cgroups主要用来限制，这个比较好理解，主要限制进程使用的各项资源

- Namespace用来修改进程的视图，效果就是让应用进程1001以为自己是1号进程，且看不到宿主机上的其他进程，可实际上在宿主机操作系统中，依然是1001进程

- 大家现在思考一下，上一张图正确的应该是什么样子?

容器的边界


App A
App B
Bins/Libs
Bins/Libs
Guest OS
Guest OS
Hypervisor
Host OS
Server


App A
App B
Docker
Bins/Libs
Bins/Libs
Host OS
Server

容器的边界

- 虚拟化技术通过运行一个完整操作系统，来隔离和限制用户进程，不可避免地造成了额外的资源消耗

- 容器化技术实际只是宿主机的一个特殊进程，只是通过内核提供的隔离技术进行了隔离和限制。

- 相比虚拟化，容器化性价比更高。

- 但是来了，这也意味着多个容器使用的是同一宿主机内核，那些无法通过Namespace隔离的资源也是共享的，比如：时间。同时各项资源也会被宿主机上的其他进程占用。

- 如果你的应用需要配置内核参数，它的影响降是整个宿主机上的所有容器进程

容器镜像

- 进程视图得到了隔离，资源也被限制了，那容器进程看到的文件系统是什么样子？这是一个简单的Mount Namespace问题吗?

- 事实上Mount Namespace修改的是，容器进程对文件系统“挂载点”的认知。只有挂载发生之后，进程视图才会改变。在此之前，新创建的容器会直接继承宿主机的各个挂载点

- 所以最直接的做法是在容器的“/“根目录挂载一个完整操作系统的文件系统

- 挂载到容器根目录上，用来为容器进程提供隔离执行环境的文件系统，就是容器镜像，也叫rootfs 根文件系统。这就是多环境高度一致的秘密

- 基于刚才的容器边界，rootfs并不包含操作系统内核

容器镜像

- 新的问题又来了，难道每开发一个应用就要制作一次rootfs吗？当多份的rootfs被差异化修改后，就是极度的碎片化

- Docker引入了layer的概念来解决这个问题，用户制作镜像的每一步操作都会生成一个层，就变成了一个增量的rootfs

- 分层后的layer通过联合文件系统UnionFS将多个不同位置的目录联合挂载到同一目录下(演示)

- Dockerfile使用大写高亮的原语描述要构建的Docker镜像，原语按顺序处理。且每个原语都会生成一个镜像层

真正的认识Docker

- 容器中的一个非常重要的原则：容器是一个“单进程”模型。容器的设计就是希望应用能和容器同生共死，这对CaaS的容器编排有着非常重要的意义。

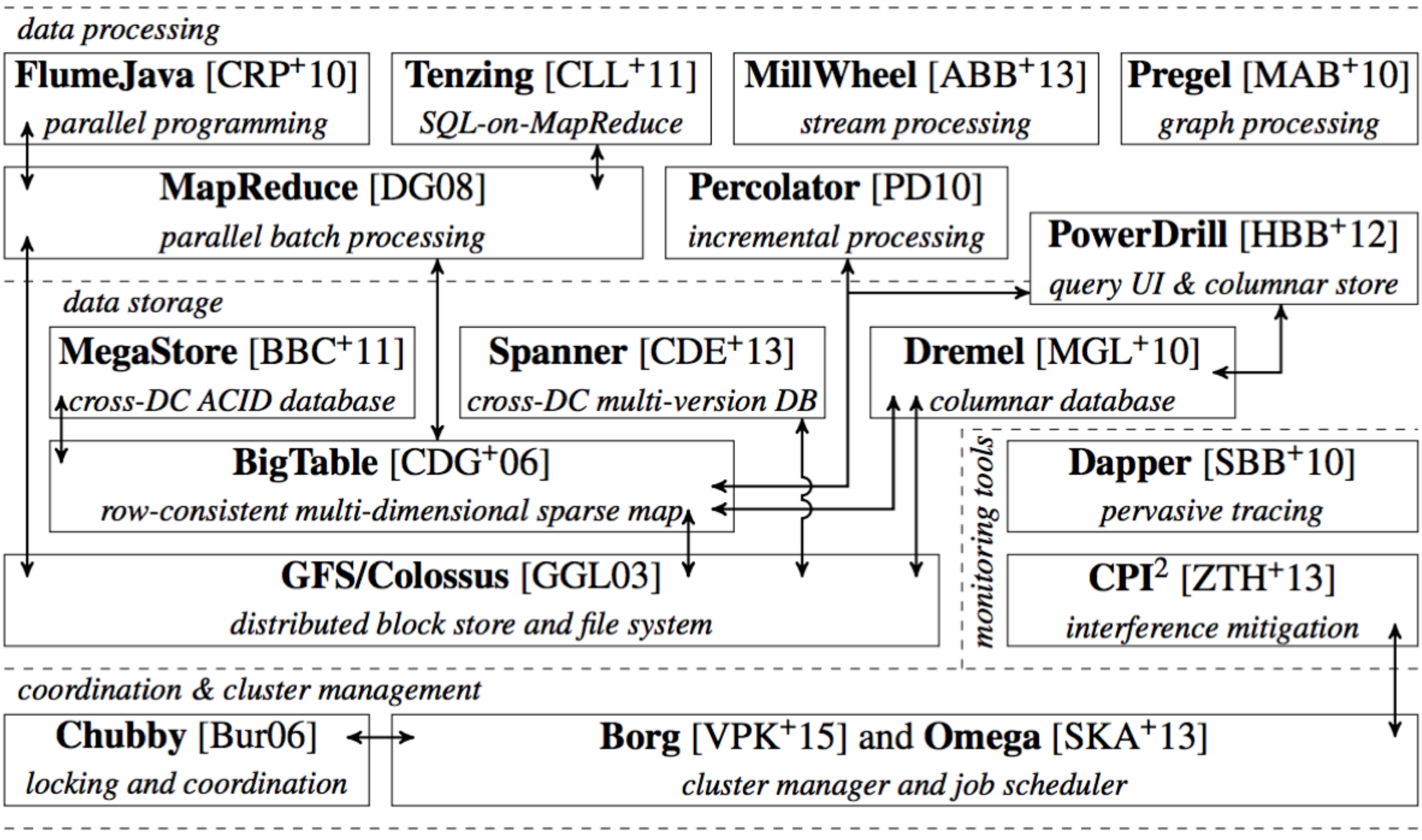
- Docker最核心的三件事：启动Linux Namespace，设置Cgroups，切换进程根目录rootfs

- 大多数开发者对应用依赖的理解，一直局限在编程语言层面，比如Java的pom.xml，golang的Godeps.json。实际上操作系统本身才是应用运行的最完整依赖，Docker完美的解决了这个依赖管理

Docker的崛起对技术人员的影响

- PaaS的最终用户和收益者，是为这个PaaS编写应用的开发者
- Docker的成功让PaaS和开发者之间变的前所未有的亲密
- 不需要精通Linux内核原理，就可以使用Docker轻松的打包一个随处可运行的Docker镜像
- 随着云计算的不断普及，Docker将是其中一个关键的底层技术，作为开发者都应该对其有充分的了解

简单聊聊Kubernetes



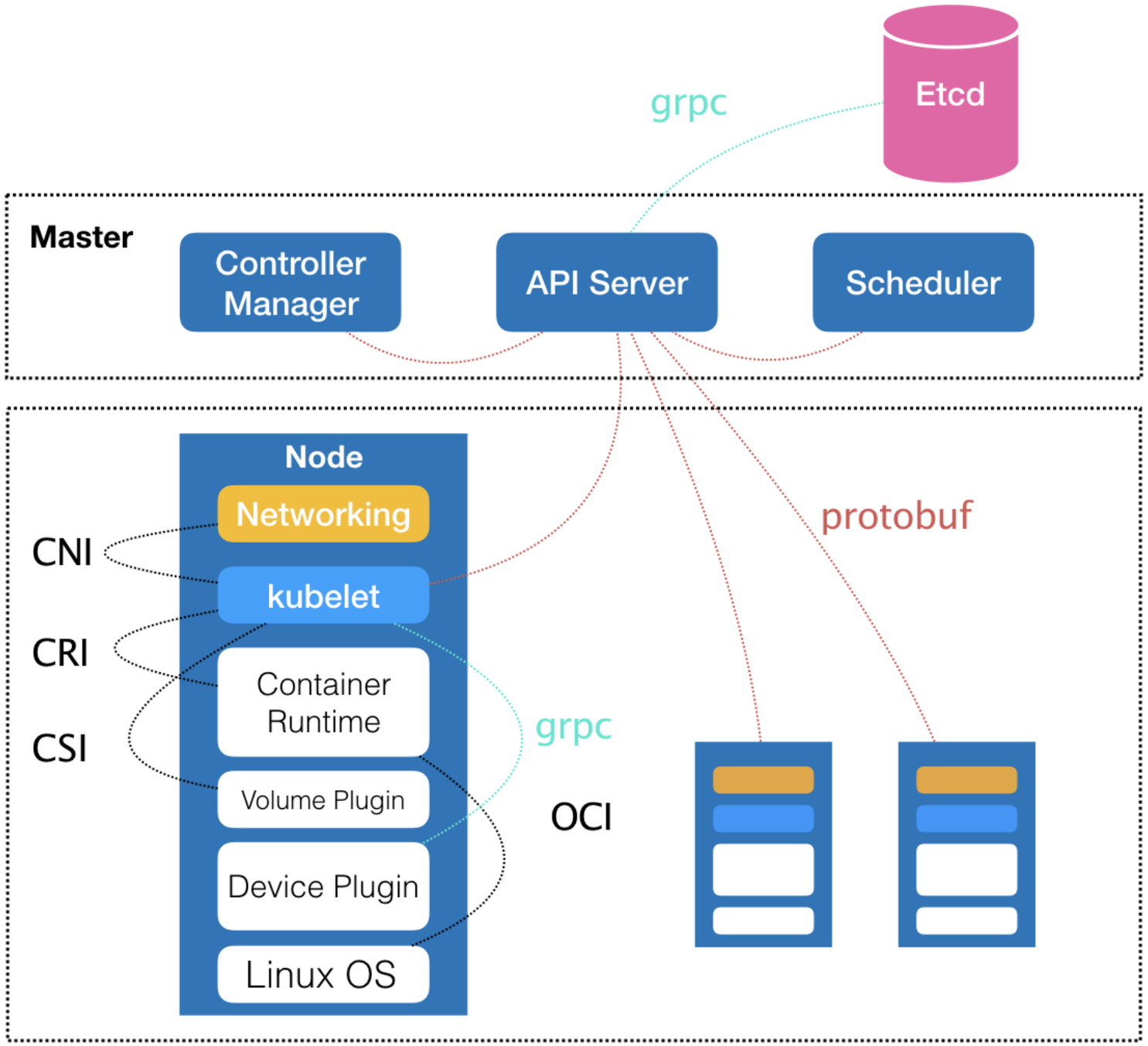
简单聊聊Kubernetes

- 基于Google 2015年发布的Borg论文，Kubernetes项目的理论基础要比工程实践走得靠前的多

- Kubernetes从一开始就站上了一个其他容器编排项目难以企及的高度

- 尽管发布之初被批评是“曲高和寡”，相比Docker公司的稚嫩和Mesos社区的老迈，kubernetes项目在Borg体系的指导下，体现出了其独有的先进性和完备性，，社区的真香系列来的很快

简单聊聊Kubernetes



简单聊聊Kubernetes

- 主要有Master和Node两部分组成，分别对应着控制节点和计算节点。本质上其实就是控制器模式

- Master节点由三个独立组件组合而成，kube-apiserver负责api服务，kube-scheduler负责调度，kube-controller-manager负责编排

- 集群数据持久化数据保存在Etcd中

- 计算节点中的核心组件是kubelet，主要负责与容器运行时(比如Docker)交互，这个交互依赖的是CRI(Container Runtime interface)的远程调用接口

- kubernetes并没有把Docker作为整个架构的核心，而只是一个最底层的容器运行时实现

- kubernetes着重解决的问题是运行在大规模集群中的各种任务之间，各种各样的关系，这些关系的处理，才是作业编排和管理系统最困难的地方