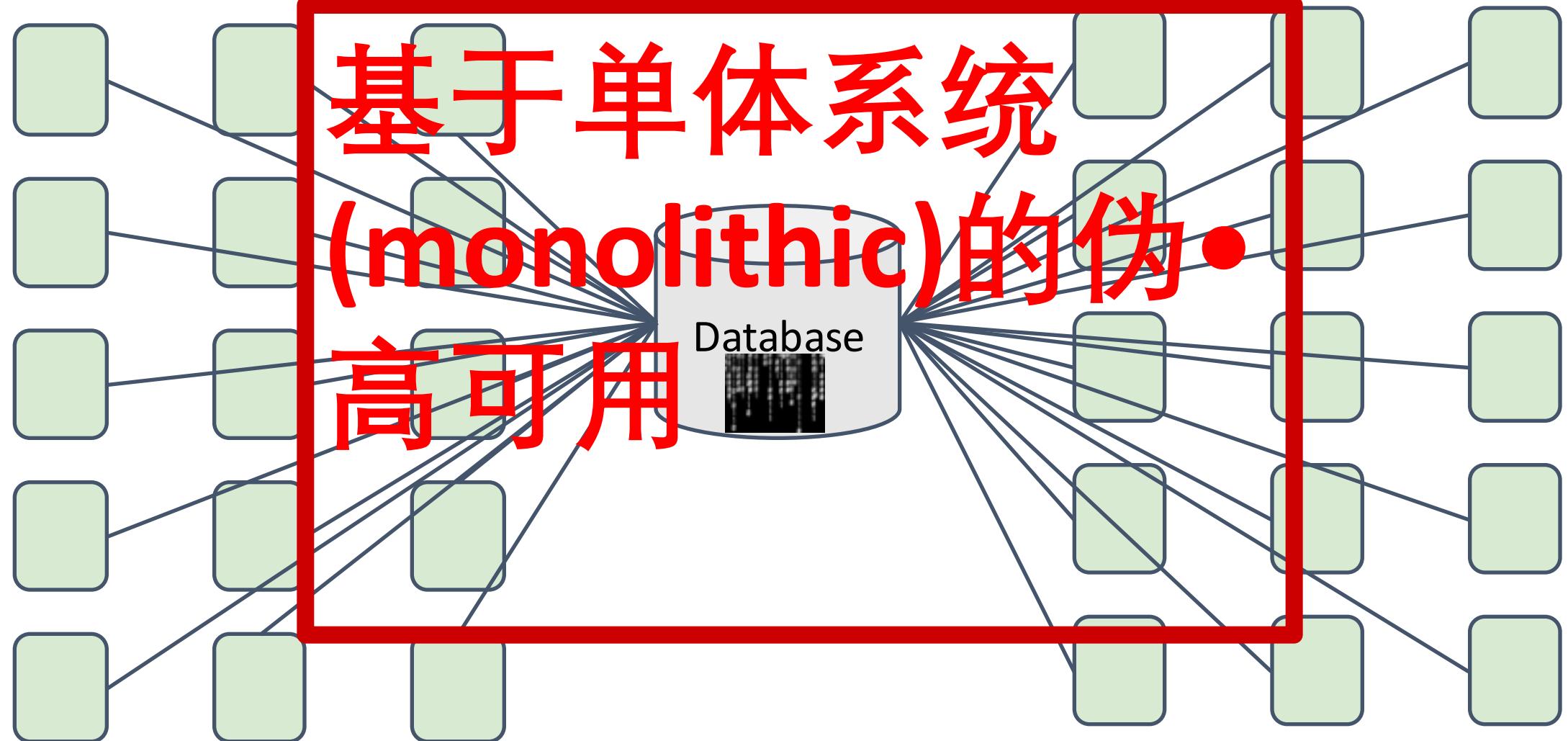


从系统架构谈起

基于单数据库实例的方案



基于微服务的方案

a/v1.0

b/v1.0

c/v1.0

f/v1.1

i/v1.0

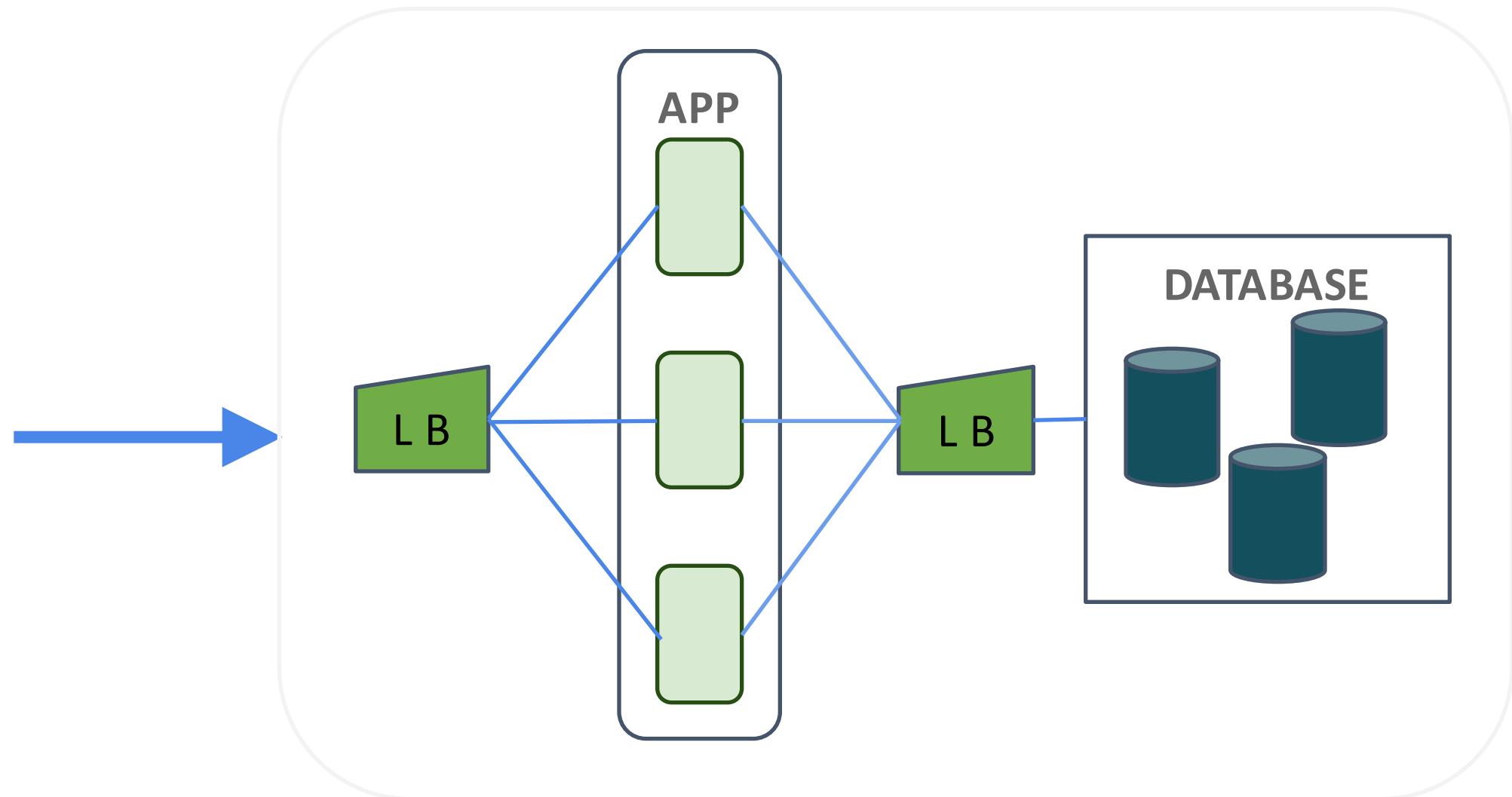
j/v1.0

按功能模块划分的独立数据库系统和单一功能组件



真·高可用

微服务组件集群



安全

负载，质量保证，
优先级

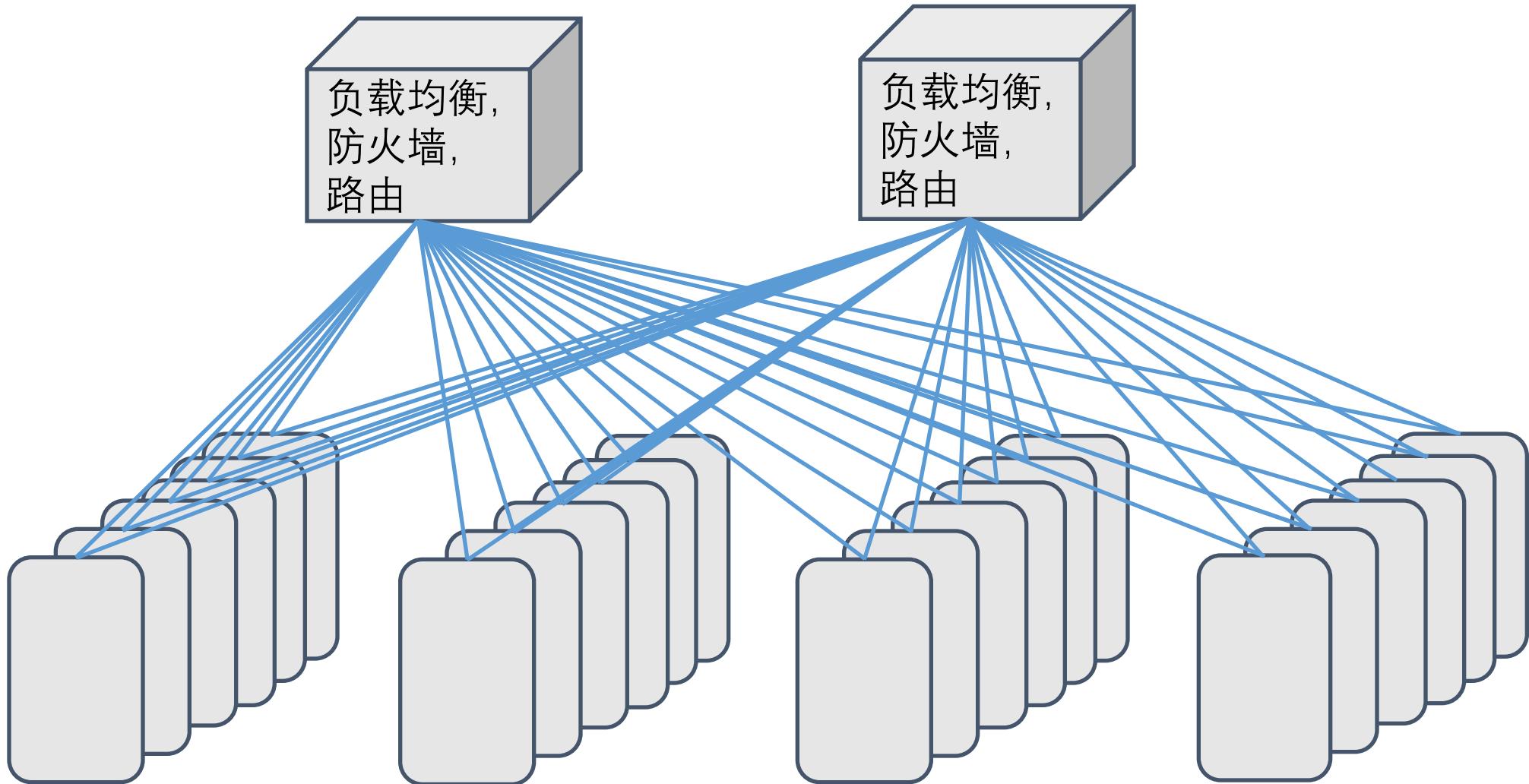
流量优化(Traffic shaping)

权限控制

应用优化(Optimization)

应用亲和

Middle Box 瓶颈



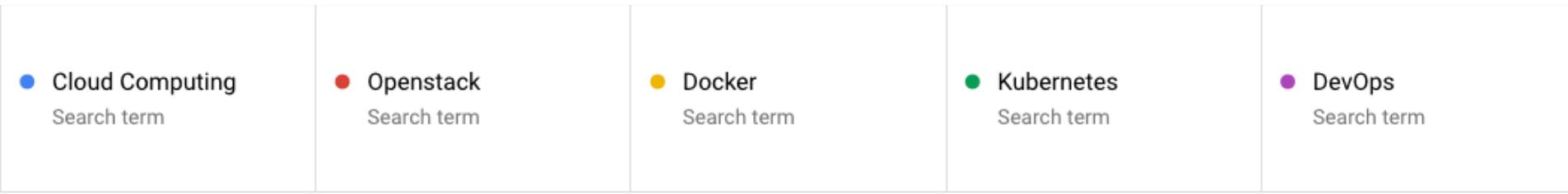
503
SERVICE UNAVAILABLE
ERROR

如何解决？

- 为所有角色抽象基础架构
 - 开发人员
 - DBA
 - 安全专员(SEC)
- 引入容器技术
- 构建面向应用的架构 (Application-oriented infrastructure)
 - 容器封装了应用环境，把很多机器和操作系统的细节从应用开发者和部署底层那里抽象了出来。
 - 因为设计良好的容器和镜像的作用范围是一个很小的应用，因此管理容器意味着管理应用而非机器，极大简化了应用的部署和管理。

第一部分 容器

Docker

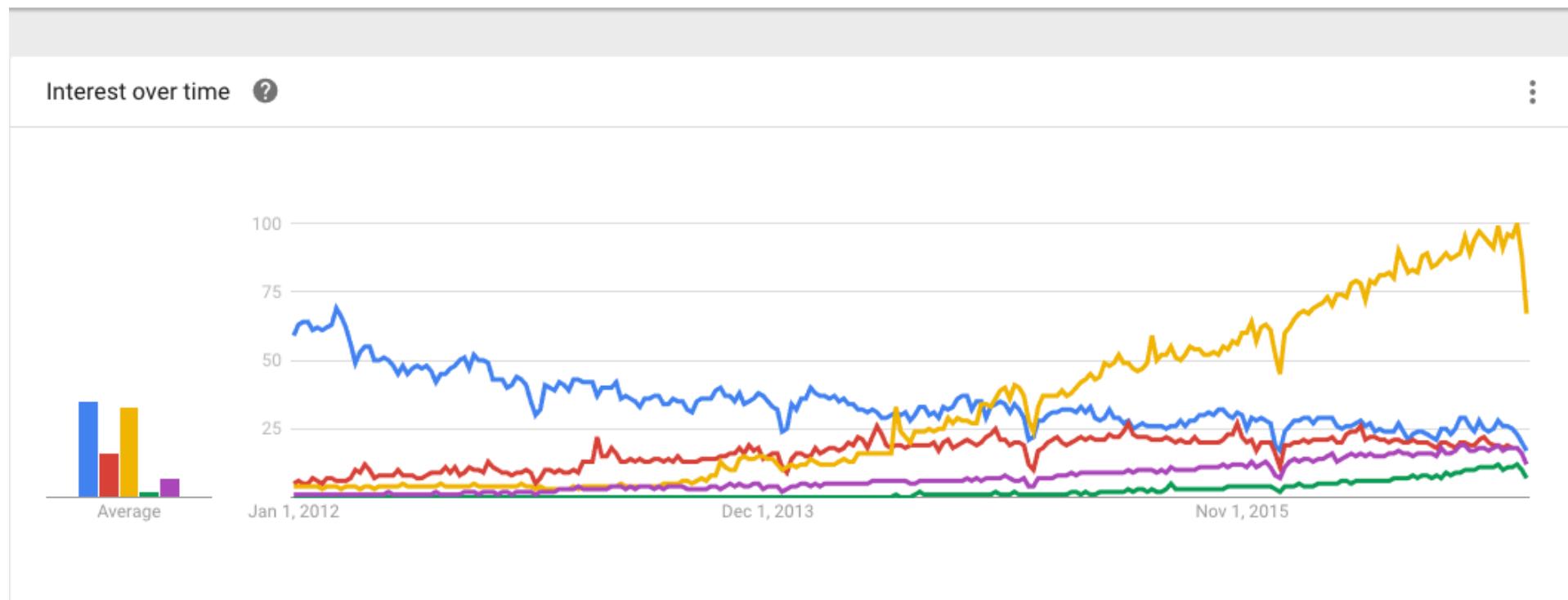


Worldwide ▾

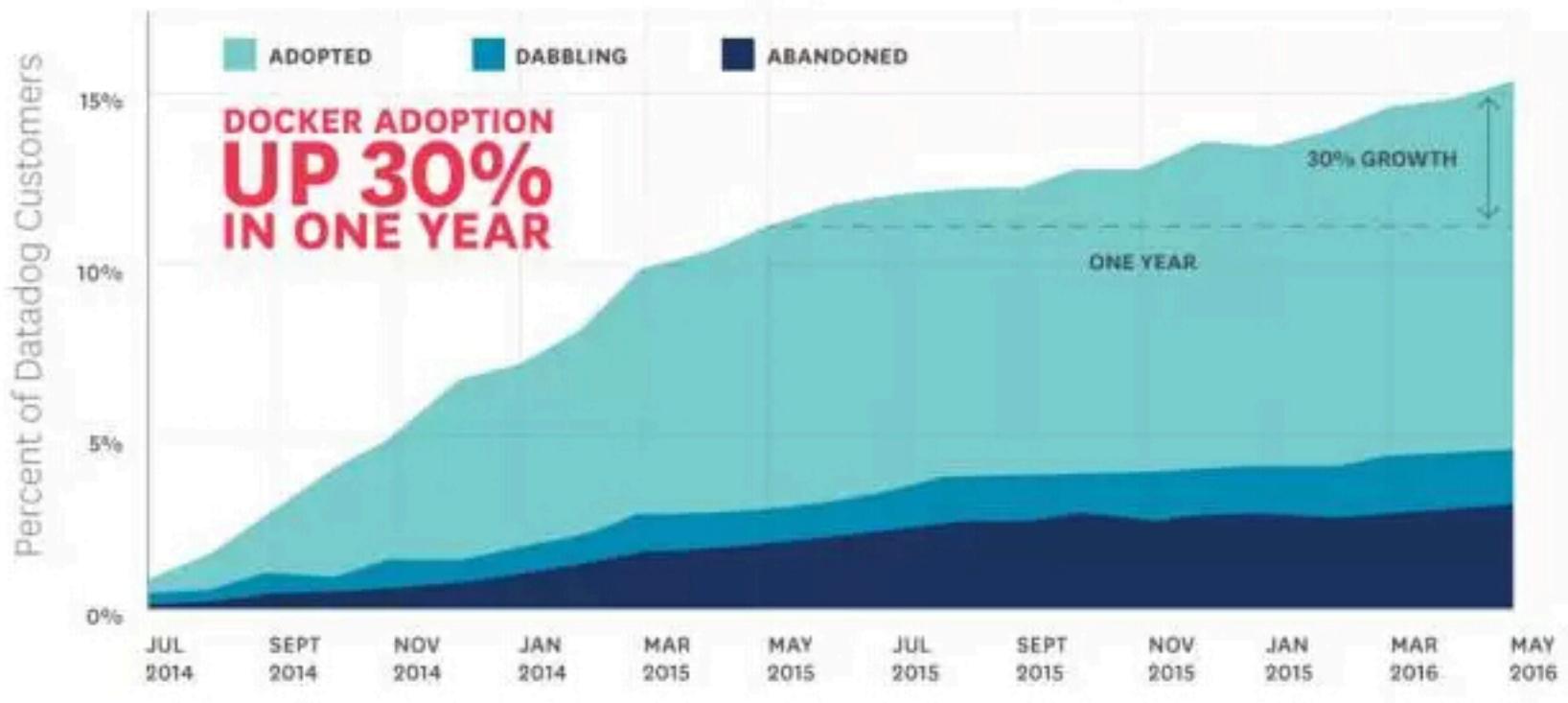
Past 5 years ▾

All categories ▾

Web Search ▾



Docker Adoption Behavior



Source: Datadog

尽管Docker技术目前还处于不稳定的发展与标准定制阶段，但这门技术已经呈现了及其火热的增长势态。

Docker到底是什么？动静为什么这么大？

- 许多公司正在以惊人的速度采用Docker
- Docker不仅仅是RedHat等Linux巨头眼里的宠儿，连微软也在积极热烈地拥抱Docker



Contact Docker

Stay informed about Docker for Windows

ANNOUNCING DOCKER CONTAINER PLATFORM FOR WINDOWS SERVER 2016

Docker and Microsoft have partnered to bring the agility, portability, and security benefits of the Docker platform to every edition of Windows Server 2016. Windows Server 2016 Containers, powered by Docker Engine, brings containers to native Windows applications and expands the toolset for traditional Docker Linux developers and IT pros.

Docker到底是什么？有什么好处？

- Docker的思想来自于集装箱，集装箱解决了什么问题？
 - 封装
 - 标准化
 - 隔离



Docker

- 基于 Linux 内核的 cgroup, namespace, 以及 AUFS 类的 Union FS 等技术，对进程进行封装隔离，属于操作系统层面的虚拟化技术。
- 由于隔离的进程独立于宿主和其它的隔离的进程，因此也称其为容器。
- 最初实现是基于 LXC，从 0.7 以后开始去除 LXC，转而使用自行开发的 libcontainer，从 1.11 开始，则进一步演进为使用 runC 和 containerd。
- Docker 在容器的基础上，进行了进一步的封装，从文件系统、网络互联到进程隔离等等，极大的简化了容器的创建和维护。
- 使得 Docker 技术比虚拟机技术更为轻便、快捷。

为什么要用Docker

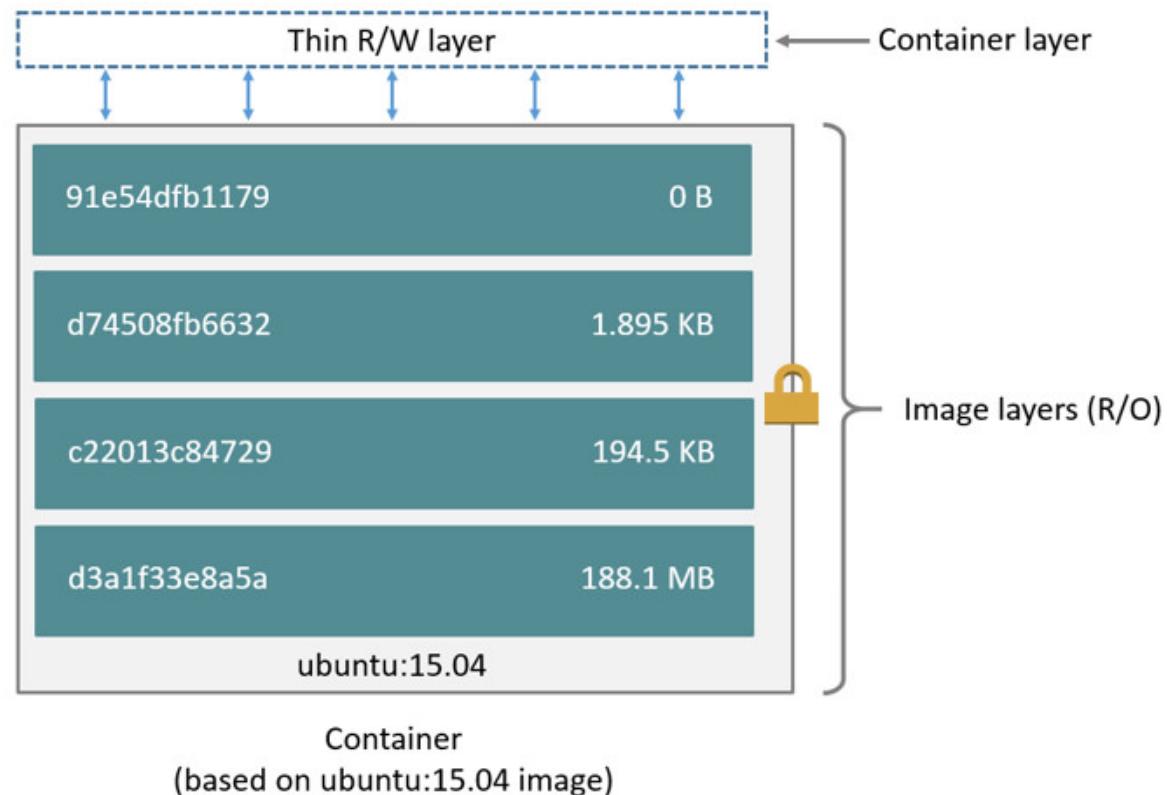
- 更高效的利用系统资源
- 更快速的启动时间
- 一致的运行环境
- 持续交付和部署
- 更轻松的迁移
- 更轻松的维护和扩展

Docker容器

- 镜像 (Image) 和容器 (Container) 的关系，就像是面向对象程序设计中的类和实例一样，镜像是静态的定义，容器是镜像运行时的实体。容器可以被创建、启动、停止、删除、暂停等。
- 容器的实质是进程，但与直接在宿主执行的进程不同，容器进程运行于属于自己的独立的命名空间。因此容器可以拥有自己的 root 文件系统、自己的网络配置、自己的进程空间，甚至自己的用户 ID 空间。容器内的进程是运行在一个隔离的环境里，使用起来，就好像是在一个独立于宿主的系统下操作一样。

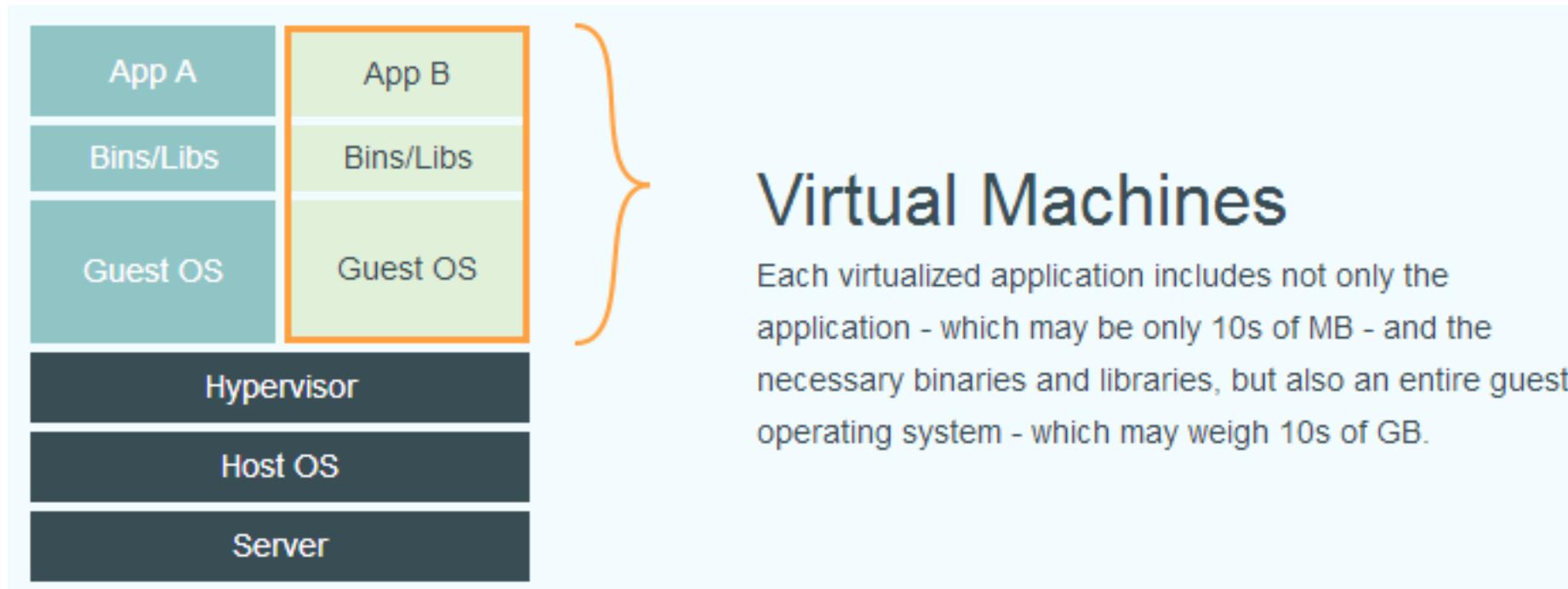
Docker容器

- 每一个容器运行时，是以镜像为基础层，在其上创建一个当前容器的存储层，我们可以称这个为容器运行时读写而准备的存储层为容器存储层。容器存储层的生存周期和容器一样，容器消亡时，容器存储层也随之消亡。因此，任何保存于容器存储层的信息都会随容器删除而丢失。



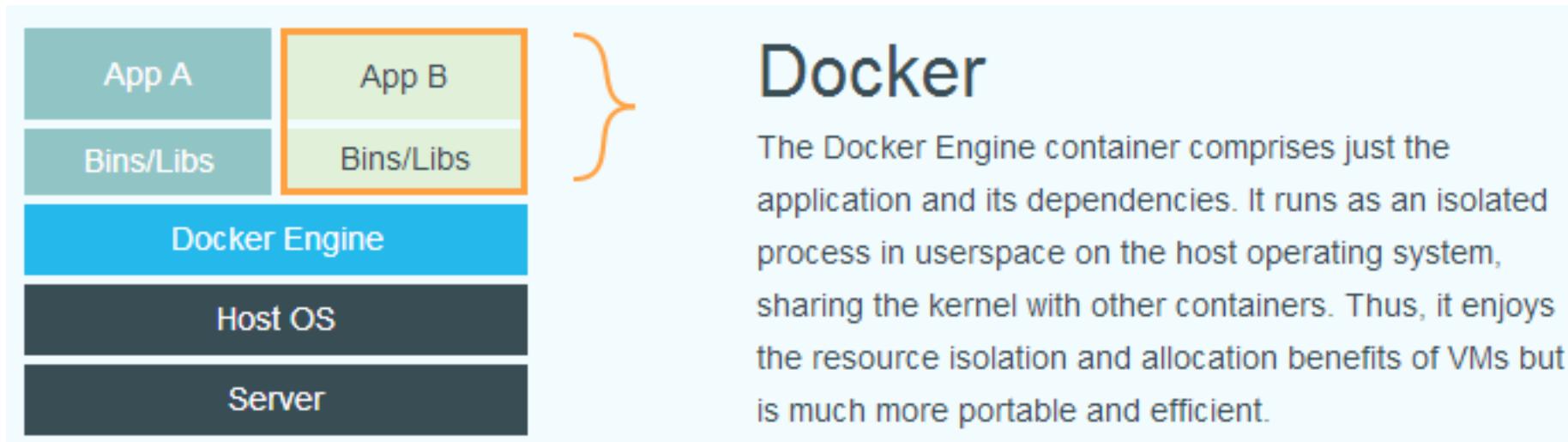
VM vs Docker

- 传统虚拟机技术是虚拟出一套硬件后，在其上运行一个完整操作系统，在该系统上再运行所需应用进程。



VM vs Docker

- 而容器内的应用进程直接运行于宿主的内核，容器内没有自己的内核，而且也没有进行硬件虚拟。因此容器要比传统虚拟机更为轻便。



性能对比

特性	容器	虚拟机
启动	秒级	分钟级
硬盘使用	一般为 MB	一般为 GB
性能	接近原生	弱于
系统支持量	单机支持上千个容器	一般几十个

隔离性: Linux Namespace(ns)

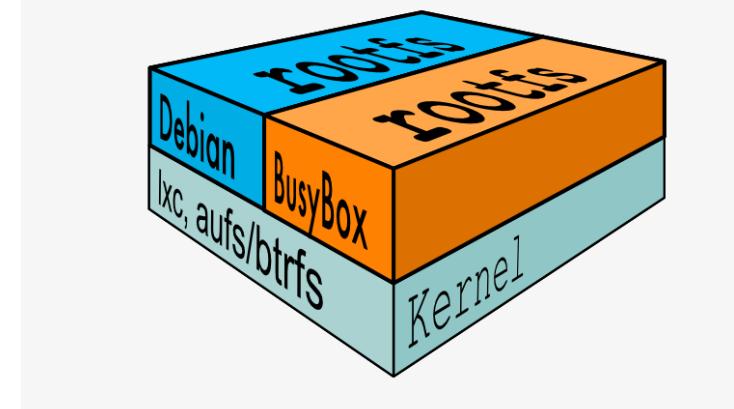
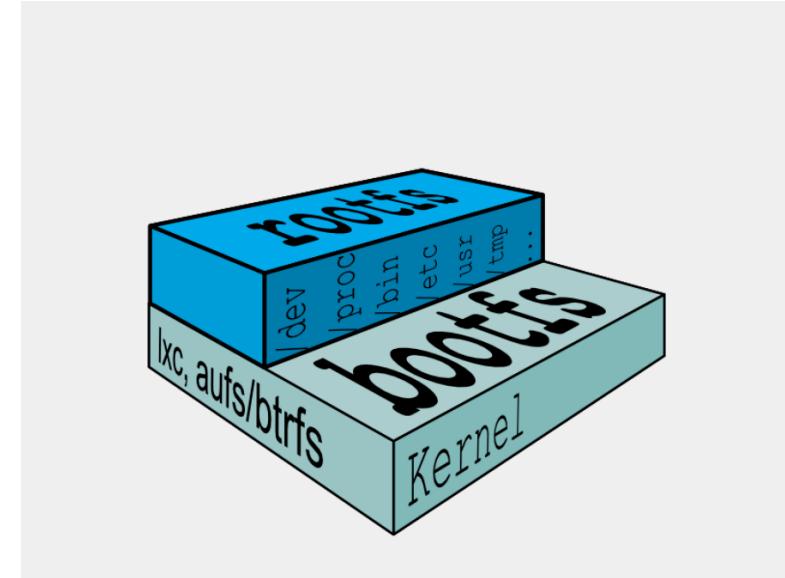
- **pid namespace**
 - 不同用户的进程就是通过pid namespace隔离开的，且不同 namespace 中可以有相同pid。
 - 有了 pid namespace, 每个namespace中的pid能够相互隔离
- **net namespace**
 - 网络隔离是通过net namespace实现的， 每个net namespace有独立的 network devices, IP addresses, IP routing tables, /proc/net 目录。
 - docker默认采用veth的方式将container中的虚拟网卡同host上的一个docker bridge: docker0连接在一起。
- **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
 - 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。
- **mnt namespace**
 - 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。
- **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, 也就是说可以在container内部用container内部的用户执行程序而非Host上的用户。

可配额/可度量 - Control Groups (cgroups)

- cgroups 实现了对资源的配额和度量。
 - 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。
 - cpu 这个子系统使用调度程序为cgroup任务提供cpu的访问。
 - cpuacct 产生cgroup任务的cpu资源报告。
 - 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。
 - devices 允许或拒绝cgroup任务对设备的访问。
 - freezer 暂停和恢复cgroup任务。
 - memory 设置每个cgroup的内存限制以及产生内存资源报告。
 - net_cls 标记每个网络包以供cgroup方便使用。
 - ns 名称空间子系统。

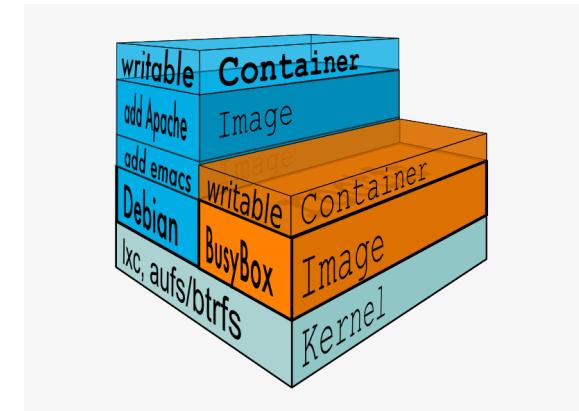
Docker的文件系统

- Docker支持Aufs, Devicemapper, Btrfs和overlayFS, ZFS, VFS
- 典型的Linux文件系统组成
 - bootfs(boot file system)
 - Bootloader - 引导加载kernel
 - Kernel - 当kernel被加载到内存中后 umount bootfs。
 - rootfs (root file system)
 - /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



Docker启动

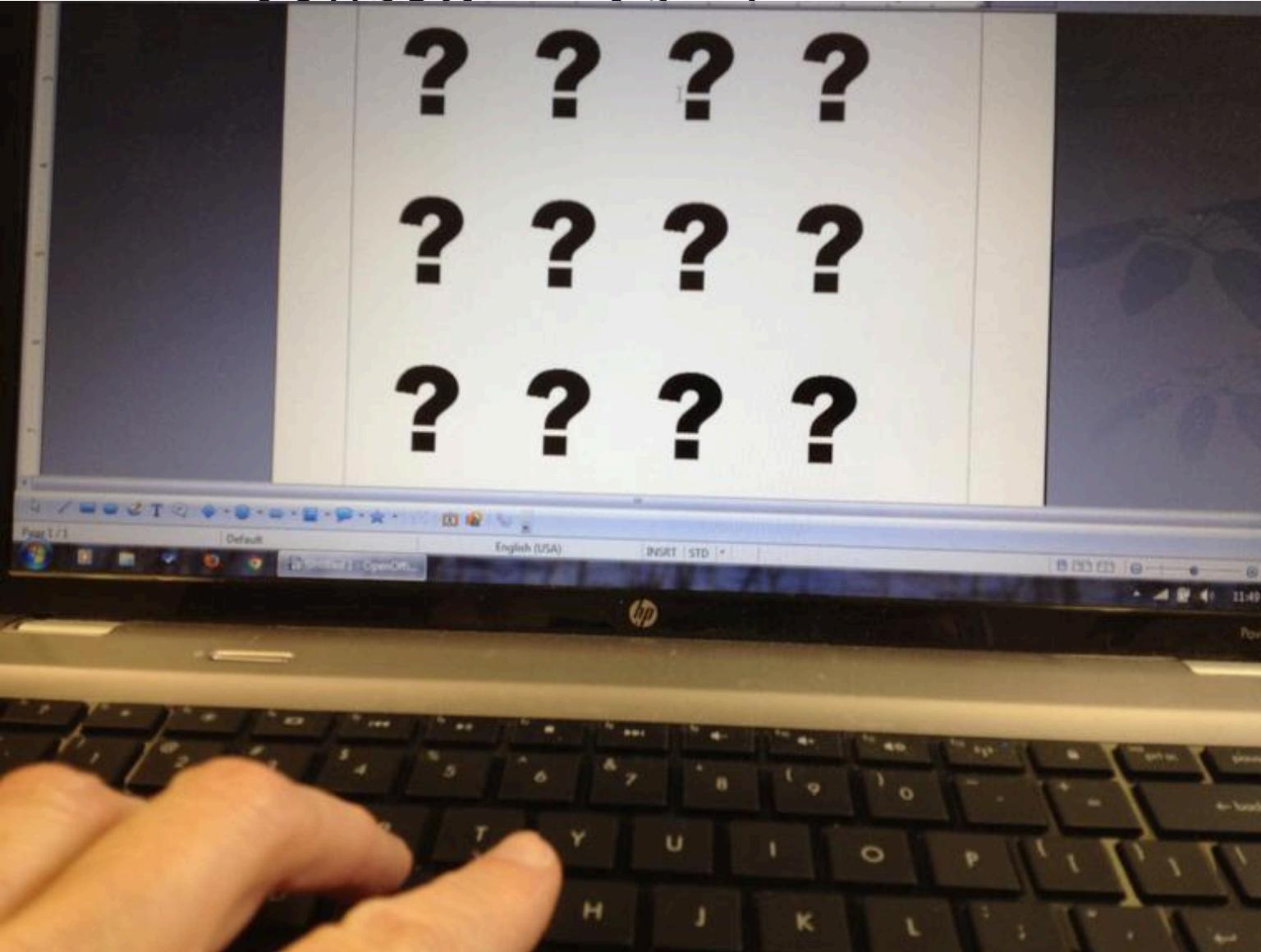
- Linux
 - 在启动后，首先将 rootfs 设置为 readonly, 进行一系列检查, 然后将其切换为 "readwrite" 供用户使用。
- Docker启动
 - 初始化时也是将 rootfs 以readonly方式加载并检查, 然而接下来利用 union mount 的方式将一个 readwrite 文件系统挂载在 readonly 的rootfs之上。
 - 并且允许再次将下层的 FS(file system) 设定为readonly 并且向上叠加。
 - 这样一组readonly和一个writeable的结构构成一个container的运行时态, 每一个FS被称作一个FS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



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

讨论： Docker真的比VM好吗？



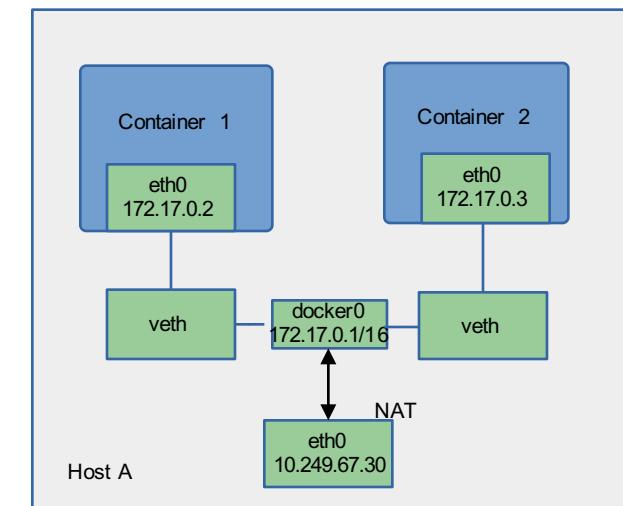
容器网络

Docker支持的网络模式

- Null(--net=None)
 - 把容器放入独立的网络空间但不做任何网络配置
 - 用户需要通过运行docker network命令来完成网络配置
- Host
 - 使用主机网络名空间，复用主机网络
- Container
 - 重用其他容器的网络
- Bridge(--net=bridge)
 - 使用Linux网桥和iptables提供容器互联，docker在每台主机上创建一个名叫docker0的网桥，通过veth pair来连接该主机的每一个endpoint
- Overlay(libnetwork, libkv)
 - 通过网络封包实现
- Remote(work with remote drivers)
 - Underlay
 - 使用现有底层网络，为每一个容器配置可路由的网络IP
 - Overlay
 - 通过网络封包实现

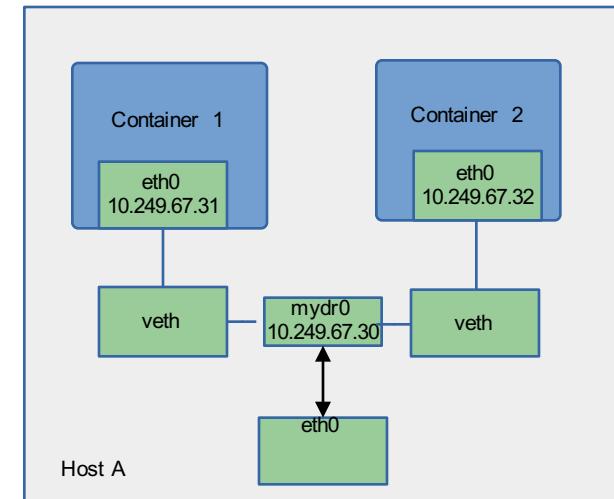
默认模式- 网桥和NAT

- 为主机eth0分配ip 10.249.67.30
- 启动docker daemon, 查看主机iptables
 - POSTROUTING -A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
- 在主机启动容器
 - docker run -d --name ssh -p 2333:22 centos-ssh
 - docker会以标准模式配置网络
 - 创建veth pair
 - 将veth pair的一端连接到docker0网桥
 - veth pair的另外一端设置为容器名空间的eth0
 - 为容器名空间的eth0分配ip.
 - 主机上的Iptables 规则
 - PREROUTING -A DOCKER ! -i docker0 -p tcp -m tcp --dport 2333 -j DNAT --to-destination 172.17.0.2:22

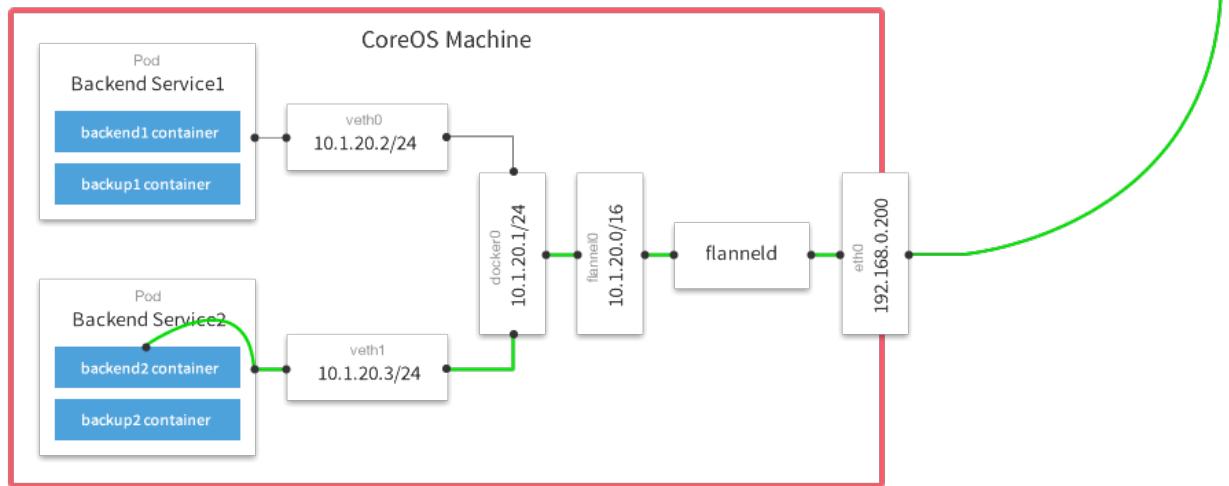
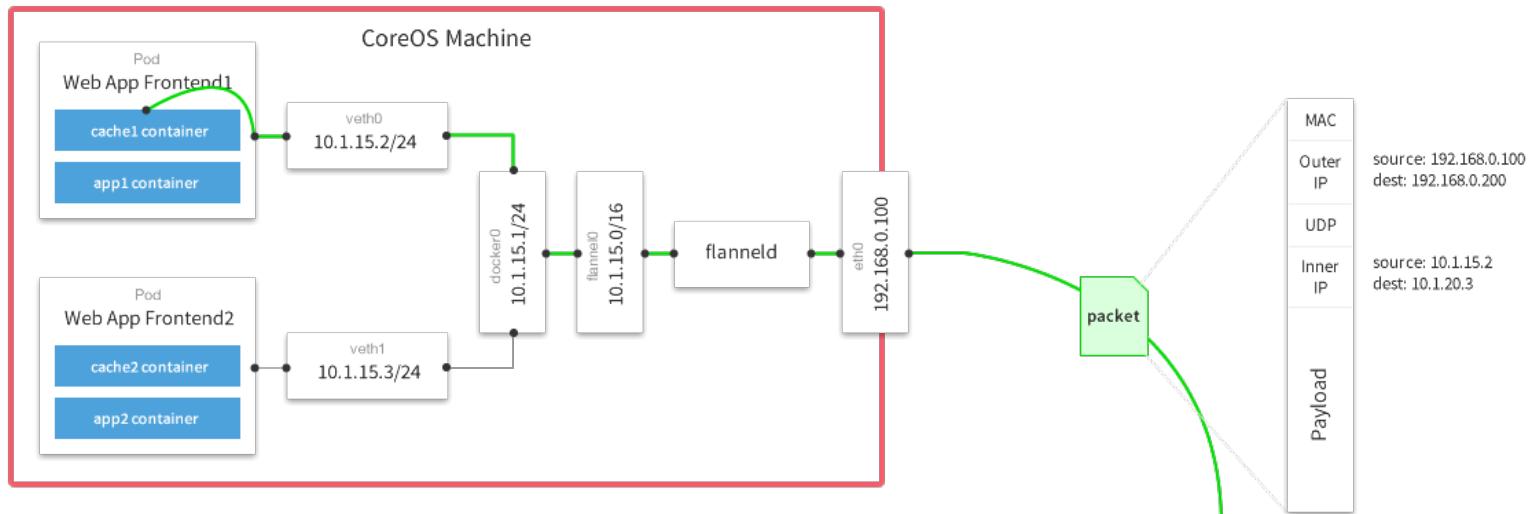


Underlay

- 采用Linux 网桥设备(sbrctl), 通过物理网络连通容器
- 创建新的网桥设备mydr0
- 将主机网卡加入网桥
- 把主机网卡的地址配置到网桥，并把默认路由规则转移到网桥mydr0
- 启动容器
- 创建veth对，并且把一个peer添加到网桥mydr0
- 配置容器把veth的另一个peer分配给容器网卡



Overlay network sample – Flannel



创建docker 镜像

- 定义dockerfile

```
FROM ubuntu
```

```
# so apt-get doesn't complain
ENV DEBIAN_FRONTEND=noninteractive
RUN sed -i 's/^exit 101/exit 0/' /usr/sbin/policy-rc.d
```

```
RUN \
apt-get update && \
apt-get install -y ca-certificates && \
apt-get install -y curl && \
rm -rf /var/lib/apt/lists/*
```

```
ADD ./bin/eic eic
ENTRYPOINT ["/eic"]
```

- docker build

Docker镜像管理

- docker save/load
- docker tag
- docker push/pull

镜像仓库

- Docker hub
 - <https://hub.docker.com/>
- 创建私有镜像仓库
 - sudo docker run -d -p 5000:5000 registry

容器操作

- 启动
 - docker run
 - -it 交互
 - -d 后台运行
 - -p 端口映射
 - -v 磁盘挂载
 - 启动已终止容器
 - docker start
 - 停止容器
 - docker stop
 - 进入容器
 - docker attach
 - 通过nsenter
 - PID=\$(docker inspect --format "{{ .State.Pid }}" <container>)
 - \$ nsenter --target \$PID --mount --uts --ipc --net --pid

Docker优势

- 与宿主机共享内核
 - 不需要再启动内核，所以应用扩缩容时可以秒速启动。
 - 资源利用率高，直接使用宿主机内核调度资源，性能损失小。
 - 方便的CPU、内存资源调整。
 - 能实现秒级快速回滚。
- 封装性：
 - 一键启动所有依赖服务，测试不用为搭建环境犯愁，PE也不用为建站复杂担心。
 - 镜像一次编译，随处使用
 - 测试、生产环境高度一致（数据除外）。
- 隔离性
 - 应用的运行环境和宿主机环境无关，完全由镜像控制，一台物理机上部署多种环境的镜像测试。
 - 多个应用版本可以并存在机器上。
- 镜像增量分发，由于采用了Union FS，简单来说就是支持将不同的目录挂载到同一个虚拟文件系统下，并实现一种layer的概念，每次发布只传输变化的部分，节约带宽。
- 社区活跃
 - Docker命令简单、易用，社区十分活跃，且周边组件丰富。

Docker的劣势？

- 讨论

第二部分 kubernetes架构 基础

议程

- Kubernetes架构
 - Kubernetes由来
 - Google Borg简介
 - Kubernetes
 - 集群管理软件应遵循的规范
 - kubernetes所遵循的规范
 - 技术架构
 - 组件
 - 组件间的通讯机制
 - 分布式存储
- kubernetes 安装
 - 环境准备
 - 搭建kubernetes集群

Google Borg简介

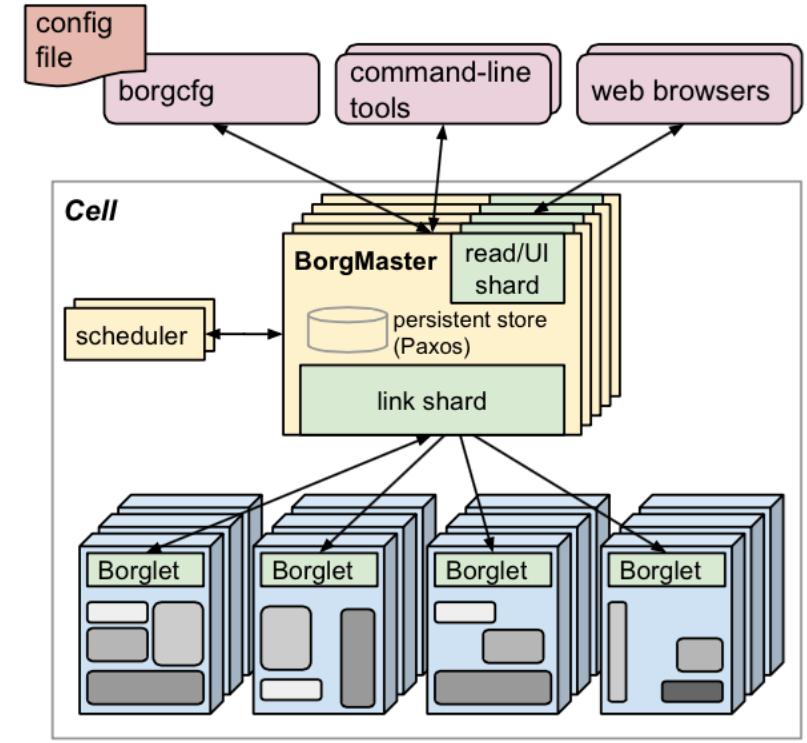
- 特性
 - 物理资源利用率高。
 - 服务器共享，在进程级别做隔离。
 - 应用高可用，故障恢复时间短。
 - 调度策略灵活。
 - 应用接入和使用方便。提供了完备的Job描述语言，服务发现，实时状态监控和诊断工具。
- 优势
 - 对外隐藏底层资源管理和调度、故障处理等。
 - 实现应用的高可靠和高可用。
 - 足够弹性，支持应用跑在成千上万的机器上。

基本概念

- Workload
 - prod: 在线任务，长期运行、对延时敏感、面向终端用户等，比如Gmail, Google Docs, Web Search服务等。
 - non-prod: 离线任务，也称为批处理任务(batch)，比如一些分布式计算服务等。
- Cell
 - 一个Cell上跑一个集群管理系统Borg。
 - 通过定义Cell可以让Borg对服务器资源进行统一抽象，作为用户就无需知道自己的应用跑在哪台机器上，也不用关心资源分配、程序安装、依赖管理、健康检查及故障恢复等。
- Job和Task
 - 用户以Job的形式提交应用部署请求。一个Job包含一个或者多个相同的Task，每个Task运行相同的一份应用程序，Task数量就是应用的副本数。
 - 每个Job可以定义一些属性、元信息和优先级，优先级涉及到抢占式调度过程。
- Naming
 - Borg的服务发现通过BNS(Borg name service)来实现。
 - 50.jfoo.ubar.cc.borg.google.com可表示在一个名为cc的Cell中由用户ubar部署的一个名为jfoo的Job下的第50个Task。

Borg 架构

- Borgmaster主进程
 - 处理客户端RPC请求，比如创建Job，查询Job等。
 - 维护系统组件和服务的状态，比如服务器、Task等。
 - 负责与Borglet通信。
- scheduler进程
 - 调度策略
 - worst fit
 - best fit
 - hybrid
 - 调度优化
 - Score caching: 当服务器或者任务的状态未发生变更或者变更很少时，直接采用缓存数据，避免重复计算。
 - Equivalence classes: 调度同一job下多个相同的Task只需计算一次。
 - Relaxed randomization: 引入一些随机性，即每次随机选择一些机器，只要符合需求的服务器数量达到一定值时，就可以停止计算，无需每次对Cell中所有服务器进行feasibility checking。
- Borglet
 - Borglet是部署在所有服务器上的Agent，负责接收Borgmaster进程的指令。



应用高可用

- 被抢占的non-prod任务放回pending queue，等待重新调度。
- 多副本应用跨故障域部署。所谓故障域有大有小，比如相同机器、相同机架或相同电源插座等，一挂全挂。
- 对于类似服务器或操作系统升级的维护操作，避免大量服务器同时进行。
- 支持幂等性，支持客户端重复操作。
- 当服务器状态变为不可用时，要控制重新调度任务的速率。因为Borg无法区分是节点故障还是出现了短暂的网络分区，如果是后者，静静地等待网络恢复更利于保障服务可用性。
- 当某种"任务@服务器"的组合出现故障时，下次重新调度时需避免这种组合再次出现，因为极大可能会再次出现相同故障。
- 记录详细的内部信息，便于故障排查和分析。
- 保障应用高可用的关键性设计原则是：无论何种原因，即使Borgmaster或者Borglet挂掉、失联，都不能杀掉正在运行的服务(Task)。

Borg系统自身高可用

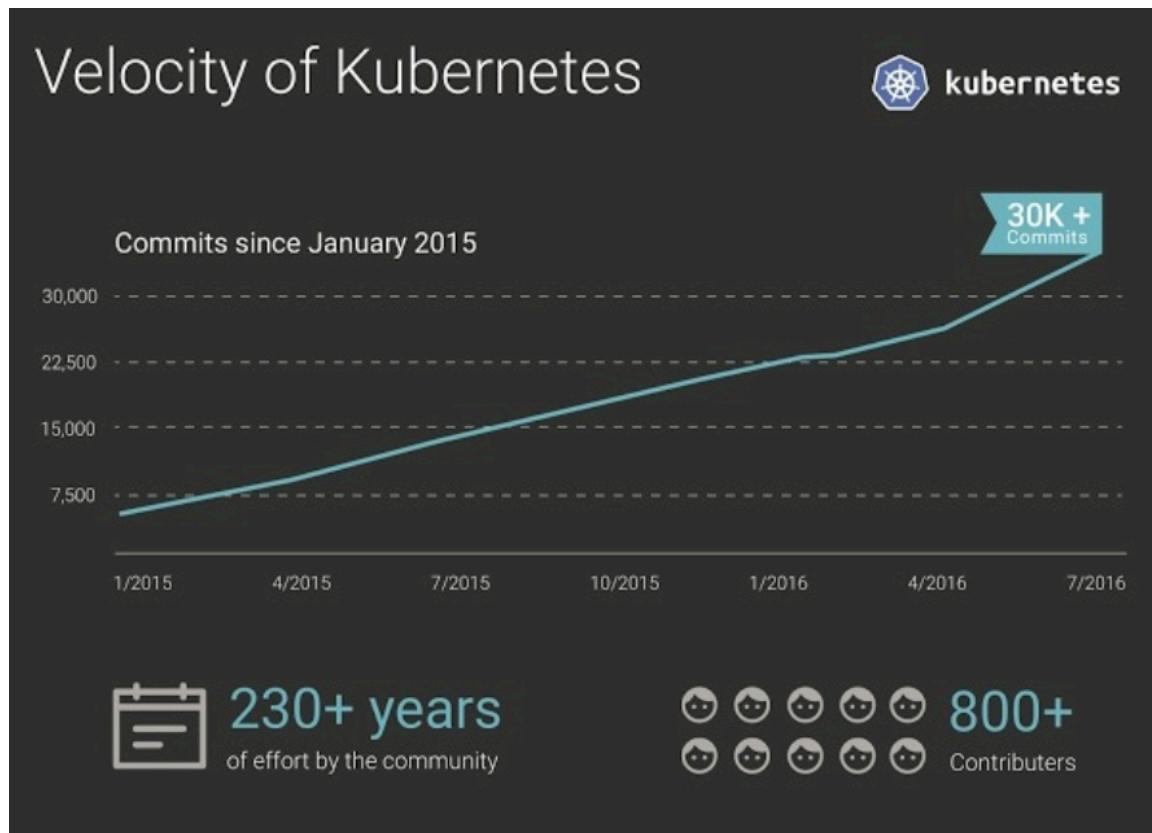
- Borgmaster组件多副本设计。
- 采用一些简单的和底层(low-level)的工具来部署Borg系统实例，避免引入过多的外部依赖。
- 每个Cell的Borg均独立部署，避免不同Borg系统相互影响。

资源利用率

- 通过将在线任务(prod)和离线任务(non-prod, batch)混合部署，空闲时，离线任务可以充分利用计算资源，繁忙时，在线任务通过抢占的方式保证优先得到执行，合理地利用资源。
- 98%的服务器实现了混部。
- 90%的服务器中跑了超过25个Task和4500个线程。
- 在一个中等规模的Cell里，在线任务和离线任务独立部署比混合部署所需的服务器数量多出约20%-30%。可以简单算一笔账，Google的服务器数量在千万级别，按20%算也是百万级别，大概能省下的服务器采购费用就是百亿级别了，这还不包括省下的机房等基础设施和电费等费用。

我们需要什么样的集群管理系统

kubernetes的崛起



Ten most-discussed repositories

	KUBERNETES/KUBERNETES	388.1K
	OPENShift/ORIGIN	91.1K
	CMS-SW/CMSSW	80.1K
	MICROSOFT/VSCODE	78.7K
	RUST-LANG/RUST	75.6K
	.NET/COREFX	75.2K
	TGSTATION/TGSTATION	74.8K
	NODEJS/NODE	66.3K
	SERVO/SERVO	54.9K
	ANSIBLE/ANSIBLE	53.9K

Kubernetes职位需求

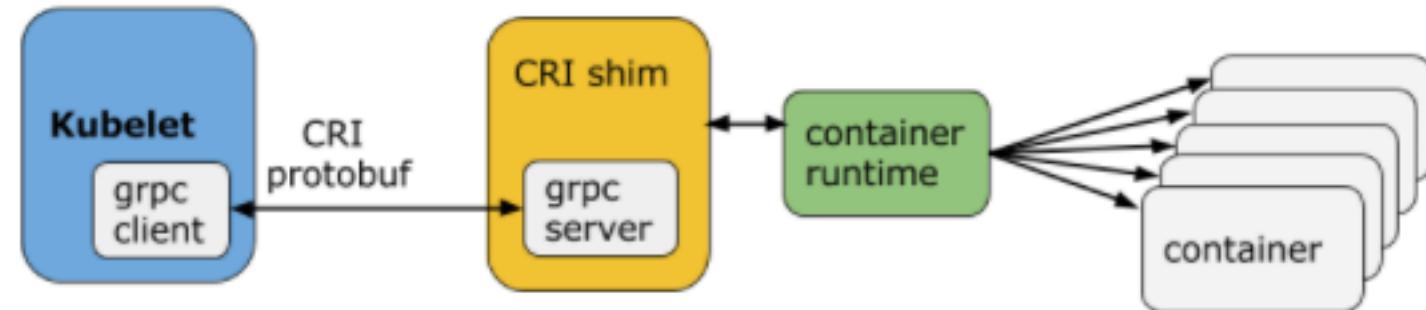
KUBERNETES WORLDWIDE JOB POSTINGS



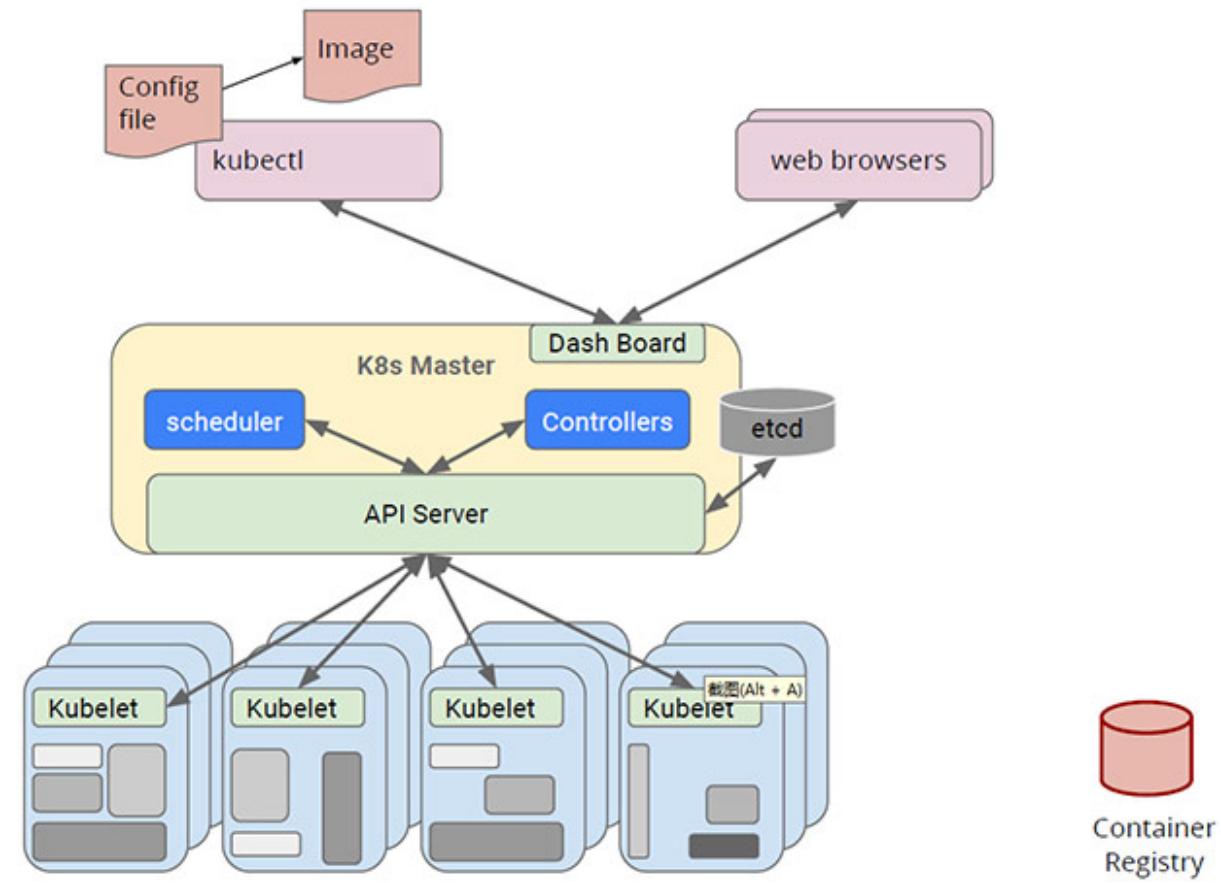
什么是Kubernetes (k8s)?

Kubernetes是谷歌开源的容器集群管理系统，是Google多年大规模容器管理技术Borg的开源版本，主要功能包括：

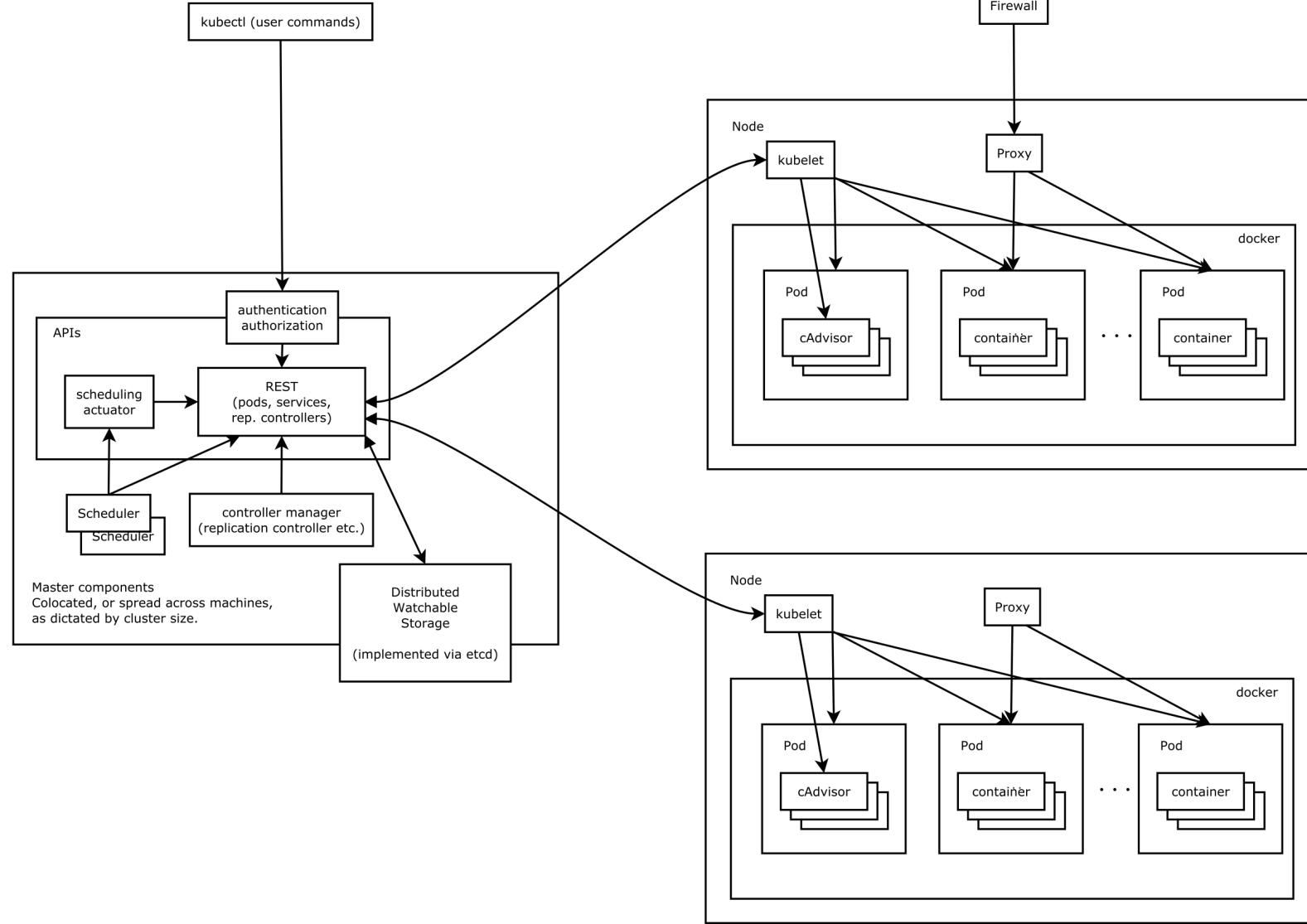
- 基于容器的应用部署、维护和滚动升级
- 负载均衡和服务发现
- 跨机器和跨地区的集群调度
- 自动伸缩
- 无状态服务和有状态服务
- 广泛的Volume支持
- 插件机制保证扩展性



Kubernetes采用与Borg类似架构



主要组件



Kubernetes 的主节点 (Master Node)

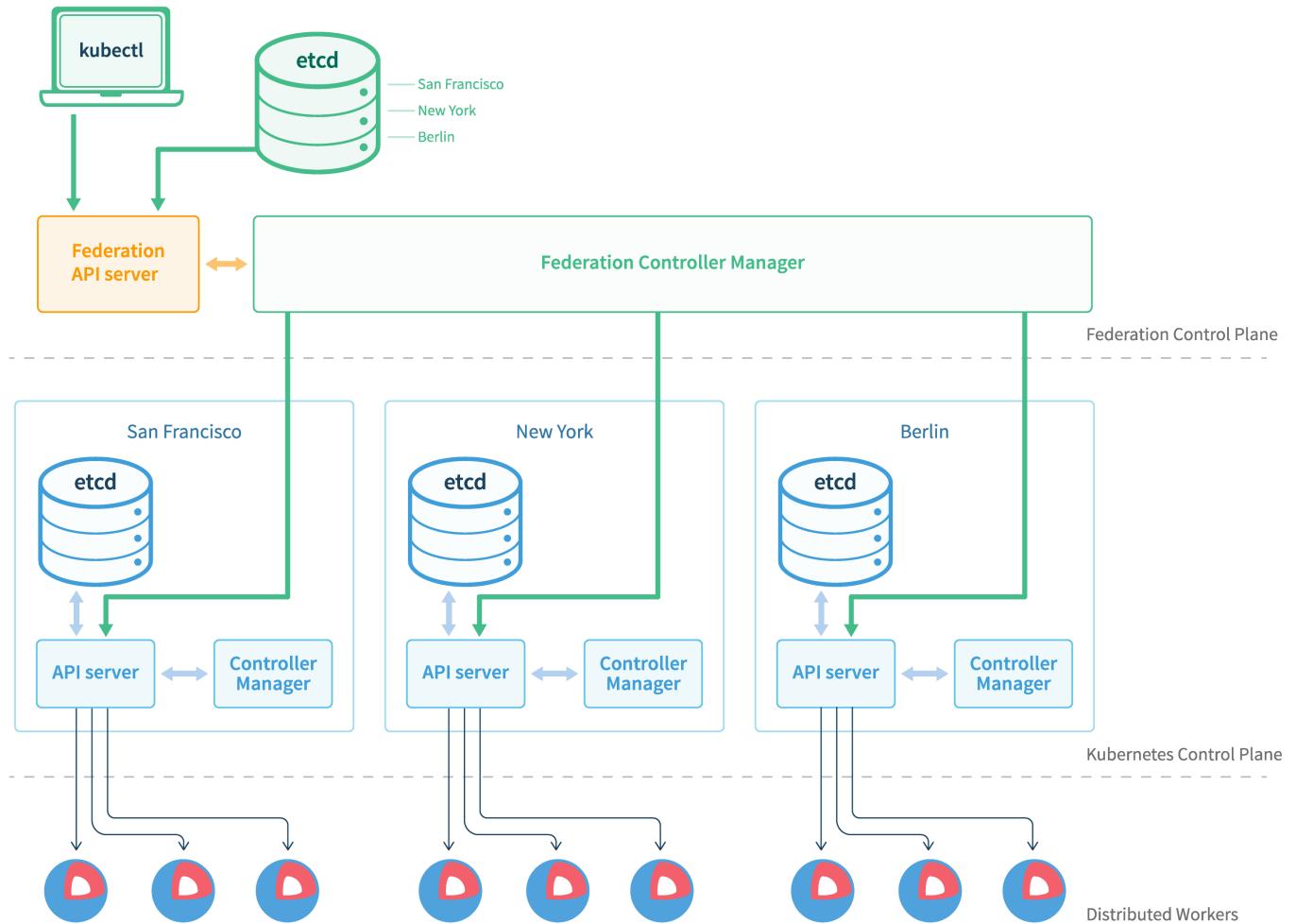
- API服务器 (API Server) : 这是Kubernetes控制面板中唯一带有用户可访问API以及用户可交互的组件。API服务器会暴露一个restful的Kubernetes API并使用JSON格式的清单文件 (manifest files)。
- 集群的数据存储 (Cluster Data Store) : Kubernetes使用“etcd”。这是一个强大的、稳定的、高可用的键值存储，被Kubernetes用于长久储存所有的API对象。可以认为它就是集群的“真相之源”。
- 控制管理器 (Controller Manager) : 被称为“kube-controller manager”，它运行着所有处理集群日常任务的控制器。包括了节点控制器、副本控制器、端点 (endpoint) 控制器以及服务账户和令牌控制器。每一个控制器都独立工作以维护其所需的状态。
- 调度器 (Scheduler) : 调度器会监控新建的pods (一组或一个容器) 并将其分配给节点。
- 仪表板 (Dashboard) (可选) : Kubernetes提供网页UI来简化用户与API服务器 (API Server) 的交互。

Kubernetes的工作节点（Worker Node）

- Kubernetes中第二重要的部分就是工作节点。鉴于主节点负责管理集群，工作节点就负责运行容器，并提供Kubernetes的运行环境。
- 工作节点包含了一个kubelet。它是主节点代理（primary node agent）。它会监控已被分配给该节点的pods中的API服务器。Kuberet执行任务并维护一个将pod状态报告给主节点的渠道。
- 每个pod里都有容器，kubelet通过Docker（拉取镜像、启动和停止容器等）来运行这些容器。它还会定期执行被请求的容器的健康探测程序。除了Docker之外，Kubernetes也支持RKT，此外社区也正在积极努力的支持OCI。
- 工作节点里的另一个组件是kube-proxy。它负责节点的网络，在主机上维护网络规则并执行连接转发。它还负责对正在服务的pods进行负载平衡。

集群联邦

集群联邦（Federation）用于跨可用区的Kubernetes集群。



核心组件

- etcd保存了整个集群的状态；
- apiserver提供了资源操作的唯一入口，并提供认证、授权、访问控制、API注册和发现等机制；
- controller manager负责维护集群的状态，比如故障检测、自动扩展、滚动更新等；
- scheduler负责资源的调度，按照预定的调度策略将Pod调度到相应的机器上；
- kubelet负责维持容器的生命周期，同时也负责Volume (CVI) 和网络 (CNI) 的管理；
- Container runtime负责镜像管理以及Pod和容器的真正运行 (CRI) ；
- kube-proxy负责为Service提供cluster内部的服务发现和负载均衡；

推荐的Add-ons

- kube-dns负责为整个集群提供DNS服务
- Ingress Controller为服务提供外网入口
- Heapster提供资源监控
- Dashboard提供GUI
- Federation提供跨可用区的集群
- Fluentd-elasticsearch提供集群日志采集、存储与查询

各组件之间如何协同工作?

- 就像所有好用的自动化工具一样，Kubernetes使用了对象说明或蓝图来管理系统该如何运行。你只需告诉Kubernetes你想要发生些什么事，其他的事情它自己就会处理。就好比是雇佣一个承包商来翻修你的厨房。你并不需要知道他们具体在做什么。您只需指定好结果，批准施工蓝图就可以让他们去处理剩下的部分。Kubernetes也以同样的方式工作。Kubernetes的操作基于声明式模型，在清单文件（manifest file）中定义的对象说明表明了集群该怎么运行。用不到使用一堆的命令，Kubernetes会做所需要的事情来完成给定的目标。

命令式（Imperative） vs 声明式（Declarative）

- 声明式系统关注“做什么”
 - 在软件工程领域，声明式系统指程序代码描述系统应该做什么而不是怎么做。仅限于描述要达到什么目的，如何达到目的交给系统。
- 命令式系统关注“如何做”
 - 在软件工程领域，命令式系统是写出解决某个问题，完成某个任务，或者达到某个目标的明确步骤。此方法明确写出系统应该执行某指令，并且期待系统返回期望结果。

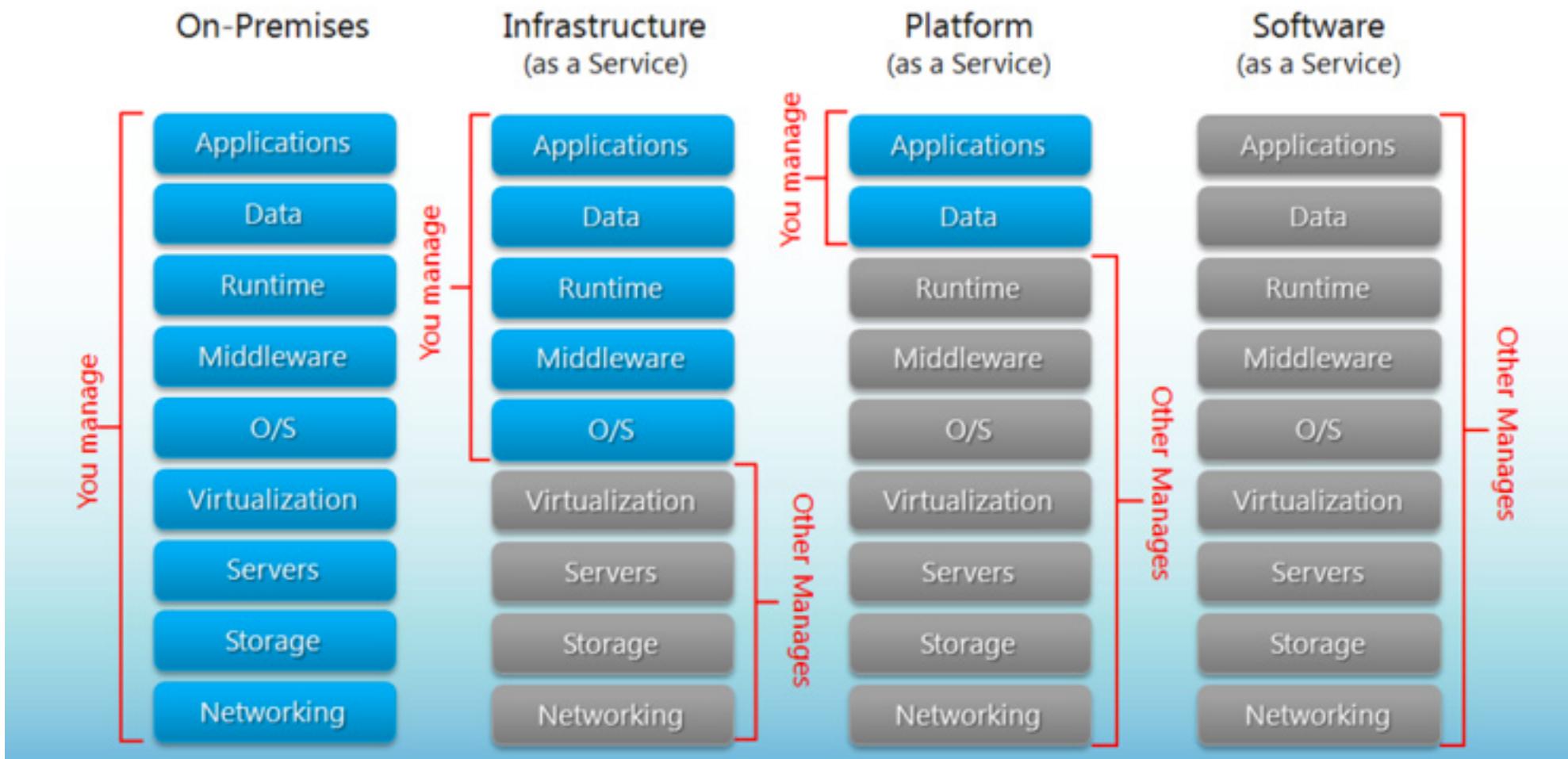


声明式(Declarative)系统规范

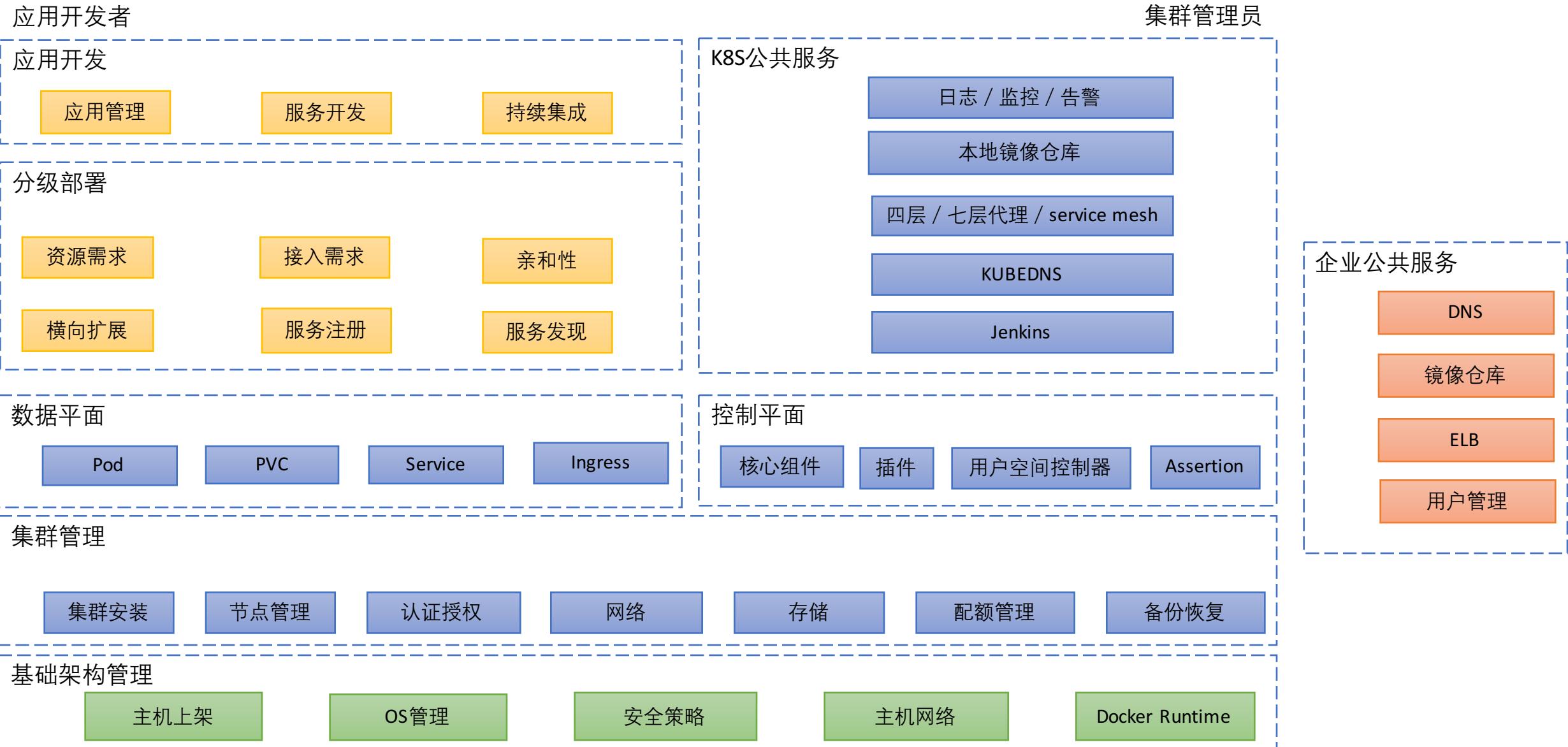
- 命令式的
 - 我要你做什么，怎么做，请严格按照我说的做。
- 声明式
 - 我需要你帮我做点事，但是我只告诉你我需要你做什么，不是你应该怎么做。
 - 直接声明
 - 我直接告诉你我需要什么
 - 间接声明
 - 我不直接告诉你我的需求，我会把我的需求放在特定的地方，请在方便的时候拿出来处理。
- 幂等性
 - 状态固定，每次我要你做事，请给我返回相同结果。
- 面向对象的
 - 把一切抽象成对象

理解Kubernetes

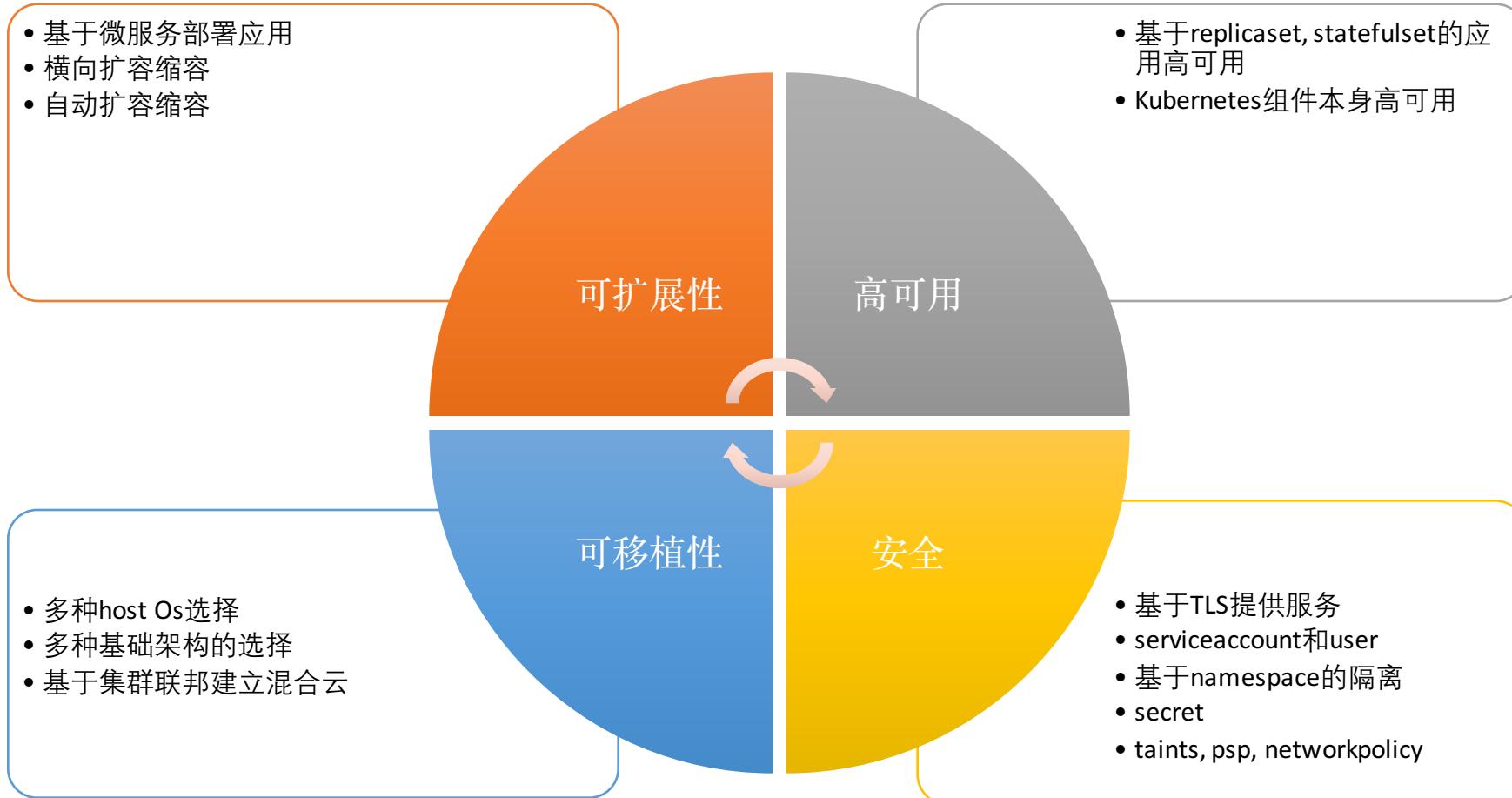
云计算的传统分类



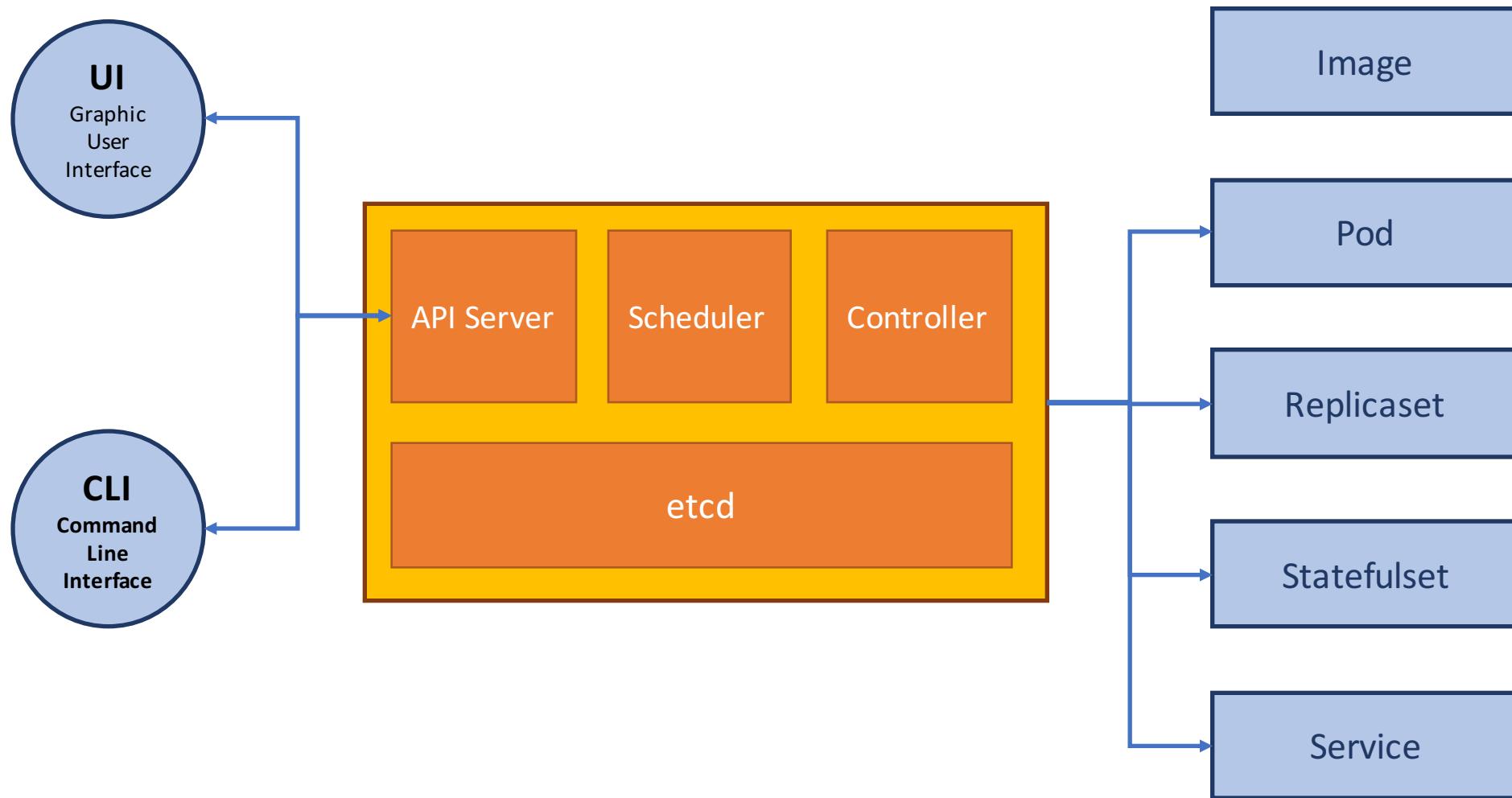
Kubernetes生态系统



Kubernetes设计理念



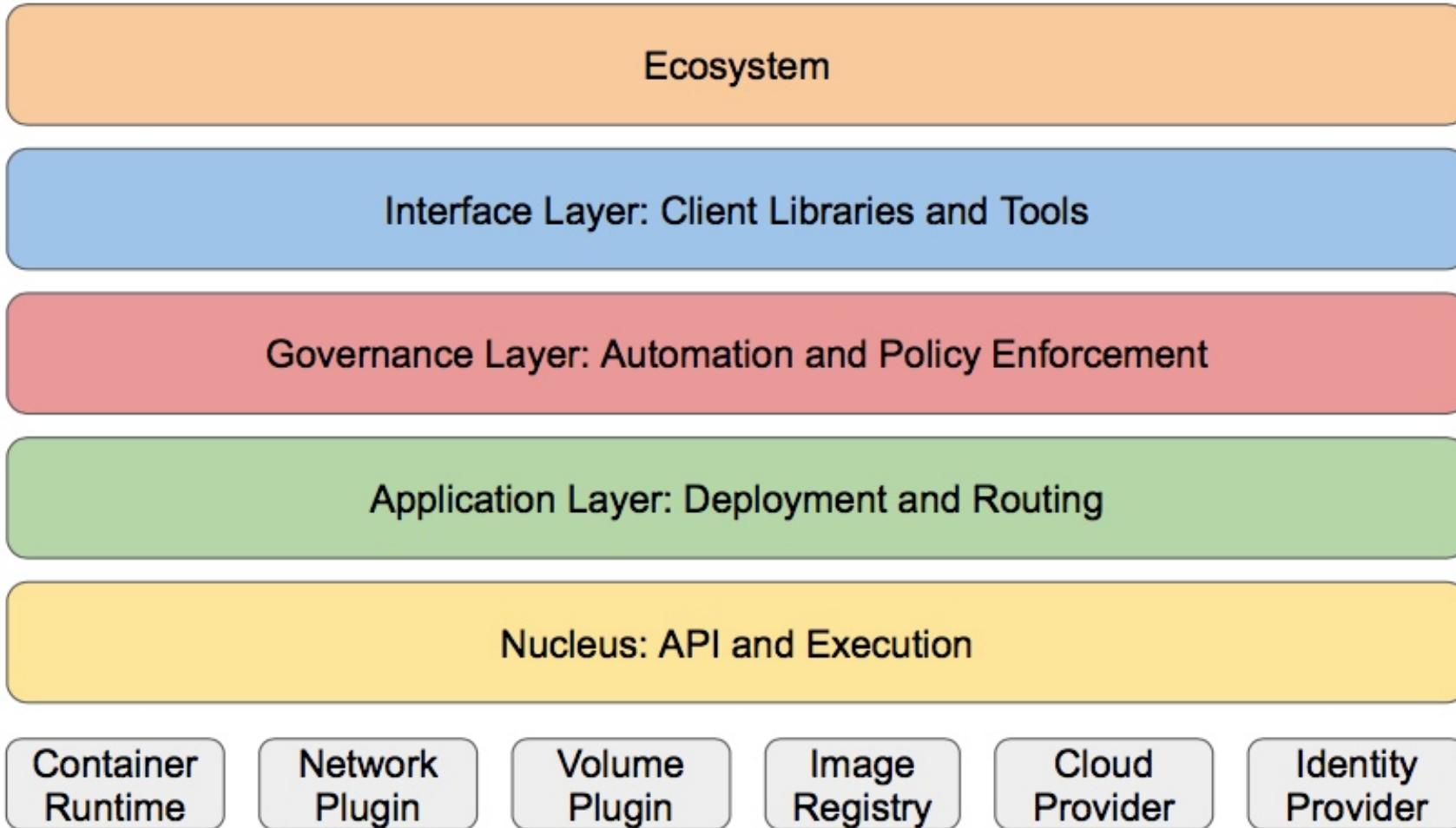
Kubernetes Master



分层架构

- 核心层：Kubernetes最核心的功能，对外提供API构建高层的应用，对内提供插件式应用执行环境
- 应用层：部署（无状态应用、有状态应用、批处理任务、集群应用等）和路由（服务发现、DNS解析等）
- 管理层：系统度量（如基础设施、容器和网络的度量），自动化（如自动扩展、动态Provision等）以及策略管理（RBAC、Quota、PSP、NetworkPolicy等）
- 接口层：kubectl命令行工具、客户端SDK以及集群联邦
- 生态系统：在接口层之上的庞大容器集群管理调度的生态系统，可以划分为两个范畴
 - Kubernetes外部：日志、监控、配置管理、CI、CD、Workflow、FaaS、OTS应用、ChatOps等
 - Kubernetes内部：CRI、CNI、CVI、镜像仓库、Cloud Provider、集群自身的配置和管理等

分层架构



Kubernetes设计理念

- 设计理念与分布式系统
 - 分析和理解Kubernetes的设计理念可以使我们更深入地了解Kubernetes系统，更好地利用它管理分布式部署的云原生应用，另一方面也可以让我们借鉴其在分布式系统设计方面的经验。

API设计原则

- 所有API应该是声明式的。声明式的操作，相对于命令式操作，对于重复操作的效果是稳定的，这对于容易出现数据丢失或重复的分布式环境来说是很重要的。另外，声明式操作更容易被用户使用，可以使系统向用户隐藏实现的细节，同时也保留了系统未来持续优化的可能性。此外，声明式的API还隐含了所有的API对象都是名词性质的，例如Service、Volume这些API都是名词，这些名词描述了用户所期望得到的一个目标对象。
- API对象是彼此互补而且可组合的。这实际上鼓励API对象尽量实现面向对象设计时的要求，即“高内聚，松耦合”，对业务相关的概念有一个合适的分解，提高分解出来的对象的可重用性。
- 高层API以操作意图为基础设计。如何能够设计好API，跟如何能用面向对象的方法设计好应用系统有相通的地方，高层设计一定是从业务出发，而不是过早的从技术实现出发。因此，针对Kubernetes的高层API设计，一定是以K8s的业务为基础出发，也就是以系统调度管理容器的操作意图为基础设计。
- 低层API根据高层API的控制需要设计。设计实现低层API的目的，是为了被高层API使用，考虑减少冗余、提高重用性的目的，低层API的设计也要以需求为基础，要尽量抵抗受技术实现影响的诱惑。
- 尽量避免简单封装，不要有在外部API无法显式知道的内部隐藏的机制。简单的封装，实际没有提供新的功能，反而增加了对所封装API的依赖性。内部隐藏的机制也是非常不利于系统维护的设计方式，例如StatefulSet和ReplicaSet，本来就是两种Pod集合，那么Kubernetes就用不同API对象来定义它们，而不会说只用同一个ReplicaSet，内部通过特殊的算法再来区分这个ReplicaSet是有状态的还是无状态。
- API操作复杂度与对象数量成正比。这一条主要是从系统性能角度考虑，要保证整个系统随着系统规模的扩大，性能不会迅速变慢到无法使用，那么最低的限定就是API的操作复杂度不能超过 $O(N)$ ，N是对象的数量，否则系统就不具备水平伸缩性了。
- API对象状态不能依赖于网络连接状态。由于众所周知，在分布式环境下，网络连接断开是经常发生的事情，因此要保证API对象状态能应对网络的不稳定，API对象的状态就不能依赖于网络连接状态。
- 尽量避免让操作机制依赖于全局状态，因为在分布式系统中要保证全局状态的同步是非常困难的。

架构设计原则

- 只有apiserver可以直接访问etcd存储，其他服务必须通过Kubernetes API来访问集群状态
- 单节点故障不应该影响集群的状态
- 在没有新请求的情况下，所有组件应该在故障恢复后继续执行上次最后收到的请求（比如网络分区或服务重启等）
- 所有组件都应该在内存中保持所需要的状态，apiserver将状态写入etcd存储，而其他组件则通过apiserver更新并监听所有的变化
- 优先使用事件监听而不是轮询

核心技术概念和API对象

- API对象是K8s集群中的管理操作单元。K8s集群系统每支持一项新功能，引入一项新技术，一定会新引入对应的API对象，支持对该功能的管理操作。例如副本集Replica Set对应的API对象是RS。
- 每个API对象都有3大类属性：元数据metadata、规范spec和状态status。元数据是用来标识API对象的，每个对象都至少有3个元数据：namespace，name和uid；除此以外还有各种各样的标签labels用来标识和匹配不同的对象，例如用户可以用标签env来标识区分不同的服务部署环境，分别用env=dev、env=testing、env=production来标识开发、测试、生产的不同服务。规范描述了用户期望K8s集群中的分布式系统达到的理想状态（Desired State），例如用户可以通过复制控制器Replication Controller设置期望的Pod副本数为3；status描述了系统实际当前达到的状态（Status），例如系统当前实际的Pod副本数为2；那么复制控制器当前的程序逻辑就是自动启动新的Pod，争取达到副本数为3。
- K8s中所有的配置都是通过API对象的spec去设置的，也就是用户通过配置系统的理想状态来改变系统，这是k8s重要设计理念之一，即所有的操作都是声明式（Declarative）的而不是命令式（Imperative）的。声明式操作在分布式系统中的好处是稳定，不怕丢操作或运行多次，例如设置副本数为3的操作运行多次也还是一个结果，而给副本数加1的操作就不是声明式的，运行多次结果就错了。

Kubernetes基础

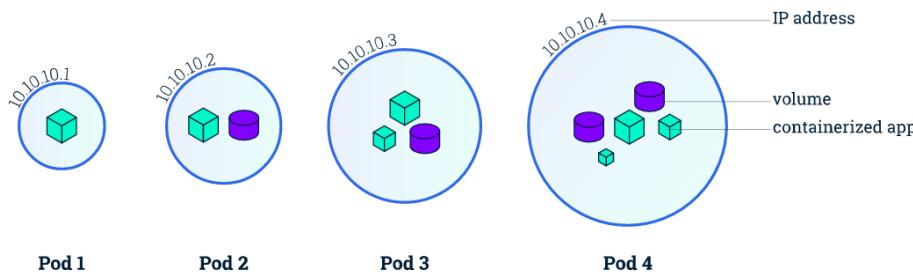
- Kubernetes基本对象
 - Container
 - Pod
 - Node
 - Namespace
 - Service
 - Label
 - Annotations

Container

- Container（容器）是一种便携式、轻量级的操作系统级虚拟化技术。它使用namespace隔离不同的软件运行环境，并通过镜像自包含软件的运行环境，从而使得容器可以很方便的在任何地方运行。
- 由于容器体积小且启动快，因此可以在每个容器镜像中打包一个应用程序。这种一对一的应用镜像关系拥有很多好处。使用容器，不需要与外部的基础架构环境绑定，因为每一个应用程序都不需要外部依赖，更不需要与外部的基础架构环境依赖。完美解决了从开发到生产环境的一致性问题。
- 容器同样比虚拟机更加透明，这有助于监测和管理。尤其是容器进程的生命周期由基础设施管理，而不是由容器内的进程对外隐藏时更是如此。最后，每个应用程序用容器封装，管理容器部署就等同于管理应用程序部署。
- 在Kubernetes必须要使用Pod来管理容器，每个Pod可以包含一个或多个容器。

Pod

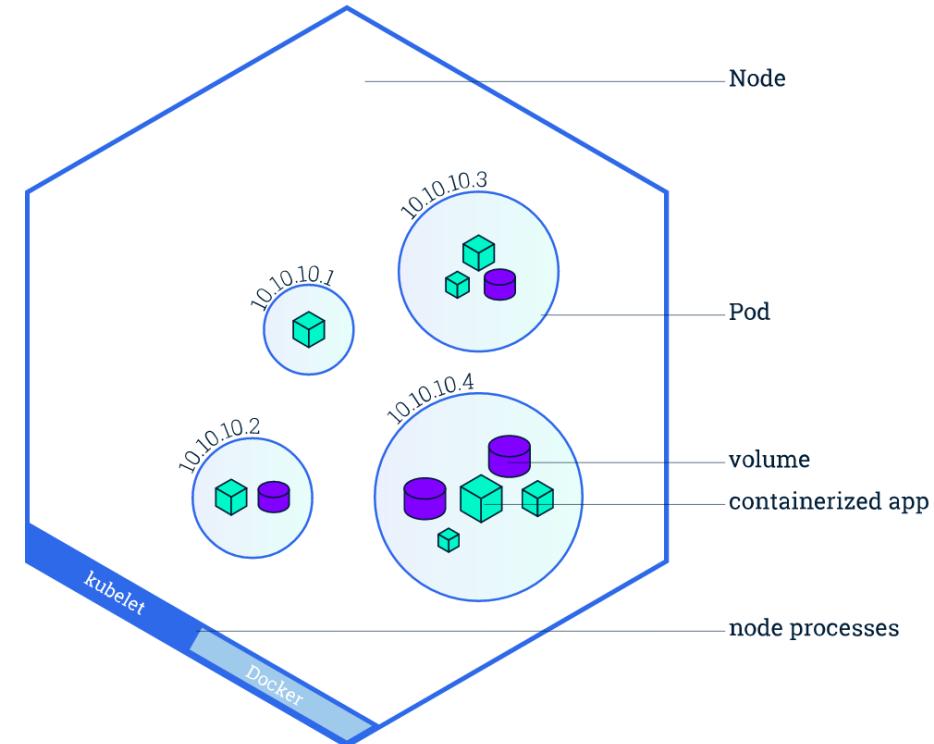
- Pod是一组紧密关联的容器集合，它们共享PID、IPC、Network和UTS namespace，是Kubernetes调度的基本单位。
- Pod的设计理念是支持多个容器在一个Pod中共享网络和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。



```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
  ports:
    - containerPort: 80
```

Node

Node是Pod真正运行的主机，可以物理机，也可以是虚拟机。为了管理Pod，每个Node节点上至少要运行container runtime（比如docker或者rkt）、kubelet和kube-proxy服务。

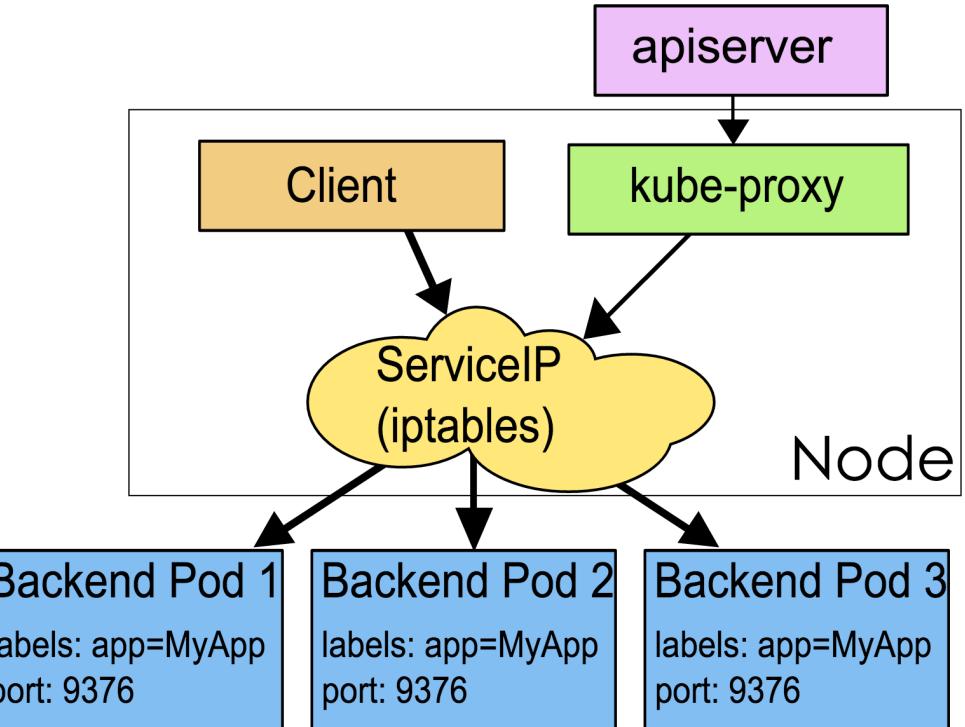


Namespace

- Namespace是对一组资源和对象的抽象集合，比如可以用来将系统内部的对象划分为不同的项目组或用户组。
- 常见的pods, services, replication controllers和deployments等都是属于某一个namespace的（默认是default），而node, persistentVolumes等则不属于任何namespace。

Service

- Service是应用服务的抽象，通过labels为应用提供负载均衡和服务发现。匹配labels的Pod IP和端口列表组成endpoints，由kube-proxy负责将服务IP负载均衡到这些endpoints上。
- 每个Service都会自动分配一个cluster IP（仅在集群内部可访问的虚拟地址）和DNS名，其他容器可以通过该地址或DNS来访问服务，而不需要了解后端容器的运行。



Service Spec

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
    - port: 8078 # the port that this service should serve on
      name: http
      # the container on each pod to connect to, can be a name
      # (e.g. 'www') or a number (e.g. 80)
    targetPort: 80
    protocol: TCP
  selector:
    app: nginx
```

Label

- Label是识别Kubernetes对象的标签，以key/value的方式附加到对象上（key最长不能超过63字节，value可以为空，也可以是不超过253字节的字符串）。
- Label不提供唯一性，并且实际上经常是很多对象（如Pods）都使用相同的label来标志具体的应用。
- Label定义好后其他对象可以使用Label Selector来选择一组相同label的对象（比如ReplicaSet和Service用label来选择一组Pod）。Label Selector支持以下几种方式：
 - 等式，如app=nginx和env!=production
 - 集合，如env in (production, qa)
 - 多个label（它们之间是AND关系），如app=nginx,env=test

Annotations

- Annotations是key/value形式附加于对象的注解。不同于Labels用于标志和选择对象，Annotations则是用来记录一些附加信息，用来辅助应用部署、安全策略以及调度策略等。比如deployment使用annotations来记录rolling update的状态。

Try it

- 通过类似docker run 的命令在k8s运行容器
 - kubectl run --image=nginx:alpine nginx-app --port=80
 - kubectl get deployment
 - kubectl describe deployment/rs/pod
- kubectl expose deployment nginx-app --port=80 --target-port=80
 - kubectl describe svc
 - kubectl describe ep

Pod

- K8s有很多技术概念，同时对应很多API对象，最重要的也是最基础的是微服务Pod。Pod是在K8s集群中运行部署应用或服务的最小单元，它是可以支持多容器的。Pod的设计理念是支持多个容器在一个Pod中共享网络地址和文件系统，可以通过进程间通信和文件共享这种简单高效的方式组合完成服务。Pod对多容器的支持是K8s最基础的设计理念。比如你运行一个操作系统发行版的软件仓库，一个Nginx容器用来发布软件，另一个容器专门用来从源仓库做同步，这两个容器的镜像不太可能是一个团队开发的，但是他们一块儿工作才能提供一个微服务；这种情况下，不同的团队各自开发构建自己的容器镜像，在部署的时候组合成一个微服务对外提供服务。
- Pod是K8s集群中所有业务类型的基础，可以看作运行在K8s集群中的小机器人，不同类型的业务就需要不同类型的小机器人去执行。目前K8s中的业务主要可以分为长期伺服型（long-running）、批处理型（batch）、节点后台支撑型（node-daemon）和有状态应用型（stateful application）；分别对应的小机器人控制器为Deployment、Job、DaemonSet和StatefulSet，本文后面会一一介绍。

复制控制器 (Replication Controller, RC)

- RC是K8s集群中最早的保证Pod高可用的API对象。通过监控运行中的Pod来保证集群中运行指定数目的Pod副本。指定的数目可以是多个也可以是1个；少于指定数目，RC就会启动运行新的Pod副本；多于指定数目，RC就会杀死多余的Pod副本。即使在指定数目为1的情况下，通过RC运行Pod也比直接运行Pod更明智，因为RC也可以发挥它高可用的能力，保证永远有1个Pod在运行。RC是K8s较早期的技术概念，只适用于长期伺服型的业务类型，比如控制小机器人提供高可用的Web服务。

副本集 (Replica Set, RS)

- RS是新一代RC，提供同样的高可用能力，区别主要在于RS后来居上，能支持更多种类的匹配模式。副本集对象一般不单独使用，而是作为Deployment的理想状态参数使用。

部署 (Deployment)

- 部署表示用户对K8s集群的一次更新操作。部署是一个比RS应用模式更广的API对象，可以是创建一个新的服务，更新一个新的服务，也可以是滚动升级一个服务。滚动升级一个服务，实际是创建一个新的RS，然后逐渐将新RS中副本数增加到理想状态，将旧RS中的副本数减小到0的复合操作；这样一个复合操作用一个RS是不太好描述的，所以用一个更通用的Deployment来描述。以K8s的发展方向，未来对所有长期伺服型的业务的管理，都会通过Deployment来管理。

服务 (Service)

- RC、RS和Deployment只是保证了支撑服务的微服务Pod的数量，但是没有解决如何访问这些服务的问题。一个Pod只是一个运行服务的实例，随时可能在一个节点上停止，在另一个节点以一个新的IP启动一个新的Pod，因此不能以确定的IP和端口号提供服务。要稳定地提供服务需要服务发现和负载均衡能力。服务发现完成的工作，是针对客户端访问的服务，找到对应的的后端服务实例。在K8s集群中，客户端需要访问的服务就是Service对象。每个Service会对应一个集群内部有效的虚拟IP，集群内部通过虚拟IP访问一个服务。在K8s集群中微服务的负载均衡是由Kube-proxy实现的。Kube-proxy是K8s集群内部的负载均衡器。它是一个分布式代理服务器，在K8s的每个节点上都有一个；这一设计体现了它的伸缩性优势，需要访问服务的节点越多，提供负载均衡能力的Kube-proxy就越多，高可用节点也随之增多。与之相比，我们平时在服务器端使用反向代理作负载均衡，还要进一步解决反向代理的高可用问题。

任务 (Job)

- Job是K8s用来控制批处理型任务的API对象。批处理业务与长期伺服业务的主要区别是批处理业务的运行有头有尾，而长期伺服业务在用户不停止的情况下永远运行。Job管理的Pod根据用户的设置把任务成功完成就自动退出了。成功完成的标志根据不同的spec.completions策略而不同：单Pod型任务有一个Pod成功就标志完成；定数成功型任务保证有N个任务全部成功；工作队列型任务根据应用确认的全局成功而标志成功。
-

后台支撑服务集 (DaemonSet)

- 长期伺服型和批处理型服务的核心在业务应用，可能有些节点运行多个同类业务的Pod，有些节点上又没有这类Pod运行；而后台支撑型服务的核心关注点在K8s集群中的节点（物理机或虚拟机），要保证每个节点上都有一个此类Pod运行。节点可能是所有集群节点也可能是通过nodeSelector选定的一些特定节点。典型的后台支撑型服务包括，存储，日志和监控等在每个节点上支撑K8s集群运行的服务。

有状态服务集 (StatefulSet)

- K8s在1.3版本里发布了Alpha版的PetSet以支持有状态服务，并从1.5版本开始重命名为StatefulSet。在云原生应用的体系里，有下面两组近义词：第一组是无状态（stateless）、牲畜（cattle）、无名（nameless）、可丢弃（disposable）；第二组是有状态（stateful）、宠物（pet）、有名（having name）、不可丢弃（non-disposable）。RC和RS主要是控制提供无状态服务的，其所控制的Pod的名字是随机设置的，一个Pod出故障了就被丢弃掉，在另一个地方重启一个新的Pod，名字变了、名字和启动在哪儿都不重要，重要的只是Pod总数；而StatefulSet是用来控制有状态服务，StatefulSet中的每个Pod的名字都是事先确定的，不能更改。StatefulSet中Pod的名字的作用，并不是《千与千寻》的人性原因，而是关联与该Pod对应的状态。
- 对于RC和RS中的Pod，一般不挂载存储或者挂载共享存储，保存的是所有Pod共享的状态，Pod像牲畜一样没有分别（这似乎也确实意味着失去了人性特征）；对于StatefulSet中的Pod，每个Pod挂载自己独立的存储，如果一个Pod出现故障，从其他节点启动一个同样名字的Pod，要挂载上原来Pod的存储继续以它的状态提供服务。
- 适合于StatefulSet的业务包括数据库服务MySQL和PostgreSQL，集群化管理服务Zookeeper、etcd等有状态服务。StatefulSet的另一种典型应用场景是作为一种比普通容器更稳定可靠的模拟虚拟机的机制。传统的虚拟机正是一种有状态的宠物，运维人员需要不断地维护它，容器刚开始流行时，我们用容器来模拟虚拟机使用，所有状态都保存在容器里，而这已被证明是非常不安全、不可靠的。使用StatefulSet，Pod仍然可以通过漂移到不同的节点提供高可用，而存储也可以通过外挂的存储来提供高可靠性，StatefulSet做的只是将确定的Pod与确定的存储关联起来保证状态的连续性。StatefulSet还只在Alpha阶段，后面的设计如何演变，我们还要继续观察。

集群联邦 (Federation)

- K8s在1.3版本里发布了beta版的Federation功能。在云计算环境中，服务的作用距离范围从近到远一般可以有：同主机（Host, Node）、跨主机同可用区（Available Zone）、跨可用区同地区（Region）、跨地区同服务商（Cloud Service Provider）、跨云平台。K8s的设计定位是单一集群在同一个地域内，因为同一个地区的网络性能才能满足K8s的调度和计算存储连接要求。而联合集群服务就是为提供跨Region跨服务商K8s集群服务而设计的。
- 每个K8s Federation有自己的分布式存储、API Server和Controller Manager。用户可以通过Federation的API Server注册该Federation的成员K8s Cluster。当用户通过Federation的API Server创建、更改API对象时，Federation API Server会在自己所有注册的子K8s Cluster都创建一份对应的API对象。在提供业务请求服务时，K8s Federation会先在自己的各个子Cluster之间做负载均衡，而对于发送到某个具体K8s Cluster的业务请求，会依照这个K8s Cluster独立提供服务时一样的调度模式去做K8s Cluster内部的负载均衡。而Cluster之间的负载均衡是通过域名服务的负载均衡来实现的。
- 所有的设计都尽量不影响K8s Cluster现有的工作机制，这样对于每个子K8s集群来说，并不需要更外层的有一个K8s Federation，也就是意味着所有现有的K8s代码和机制不需要因为Federation功能有任何变化。

存储卷 (Volume)

- K8s集群中的存储卷跟Docker的存储卷有些类似，只不过Docker的存储卷作用范围为一个容器，而K8s的存储卷的生命周期和作用范围是一个Pod。每个Pod中声明的存储卷由Pod中的所有容器共享。K8s支持非常多的存储卷类型，特别的，支持多种公有云平台的存储，包括AWS，Google和Azure云；支持多种分布式存储包括GlusterFS和Ceph；也支持较容易使用的主机本地目录hostPath和NFS。K8s还支持使用Persistent Volume Claim即PVC这种逻辑存储，使用这种存储，使得存储的使用者可以忽略后台的实际存储技术（例如AWS，Google或GlusterFS和Ceph），而将有关存储实际技术的配置交给存储管理员通过Persistent Volume来配置。

持久存储卷 (Persistent Volume, PV) 和持久存储卷声明 (Persistent Volume Claim, PVC)

- PV和PVC使得K8s集群具备了存储的逻辑抽象能力，使得在配置Pod的逻辑里可以忽略对实际后台存储技术的配置，而把这项配置的工作交给PV的配置者，即集群的管理者。存储的PV和PVC的这种关系，跟计算的Node和Pod的关系是非常类似的；PV和Node是资源的提供者，根据集群的基础设施变化而变化，由K8s集群管理员配置；而PVC和Pod是资源的使用者，根据业务服务的需求变化而变化，由K8s集群的使用者即服务的管理员来配置。

节点 (Node)

- K8s集群中的计算能力由Node提供，最初Node称为服务节点Minion，后来改名为Node。K8s集群中的Node也就等同于Mesos集群中的Slave节点，是所有Pod运行所在的工作主机，可以是物理机也可以是虚拟机。不论是物理机还是虚拟机，工作主机的统一特征是上面要运行kubelet管理节点上运行的容器。

密钥对象 (Secret)

- Secret是用来保存和传递密码、密钥、认证凭证这些敏感信息的对象。使用Secret的好处是可以避免把敏感信息明文写在配置文件里。在K8s集群中配置和使用服务不可避免的要用到各种敏感信息实现登录、认证等功能，例如访问AWS存储的用户名密码。为了避免将类似的敏感信息明文写在所有需要使用的配置文件中，可以将这些信息存入一个Secret对象，而在配置文件中通过Secret对象引用这些敏感信息。这种方式的好处包括：意图明确，避免重复，减少暴漏机会。

用户帐户（User Account）和服务帐户（Service Account）

- 顾名思义，用户帐户为人提供账户标识，而服务账户为计算机进程和K8s集群中运行的Pod提供账户标识。用户帐户和服务帐户的一个区别是作用范围；用户帐户对应的是人的身份，人的身份与服务的namespace无关，所以用户账户是跨namespace的；而服务帐户对应的是一个运行中程序的身份，与特定namespace是相关的。

名字空间 (Namespace)

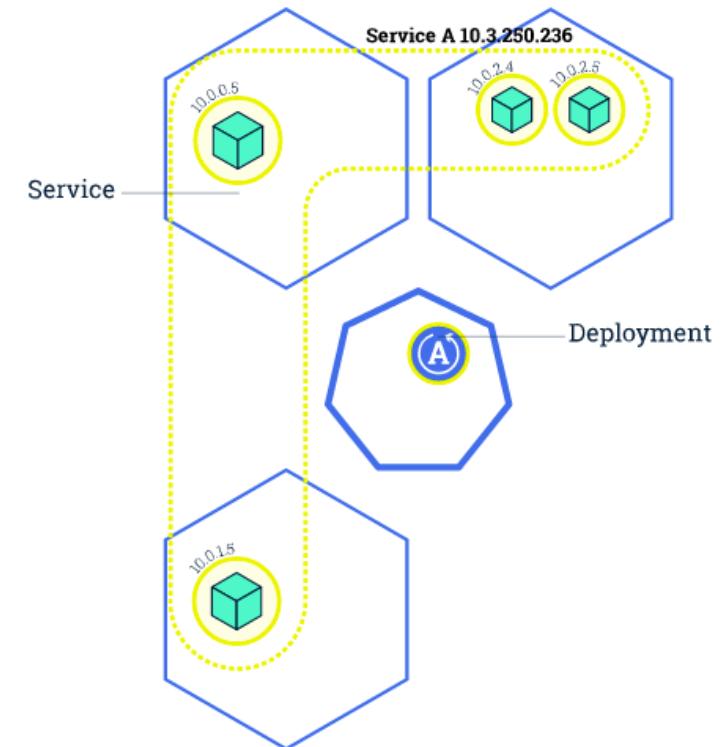
- 名字空间为K8s集群提供虚拟的隔离作用，K8s集群初始有两个名字空间，分别是默认名字空间default和系统名字空间kube-system，除此以外，管理员可以创建新的名字空间满足需要。

RBAC访问授权

- K8s在1.3版本中发布了alpha版的基于角色的访问控制（Role-based Access Control, RBAC）的授权模式。相对于基于属性的访问控制（Attribute-based Access Control, ABAC），RBAC主要是引入了角色（Role）和角色绑定（RoleBinding）的抽象概念。在ABAC中，K8s集群中的访问策略只能跟用户直接关联；而在RBAC中，访问策略可以跟某个角色关联，具体的用户在跟一个或多个角色相关联。显然，RBAC像其他新功能一样，每次引入新功能，都会引入新的API对象，从而引入新的概念抽象，而这一新的概念抽象一定会使集群服务管理和使用更容易扩展和重用。

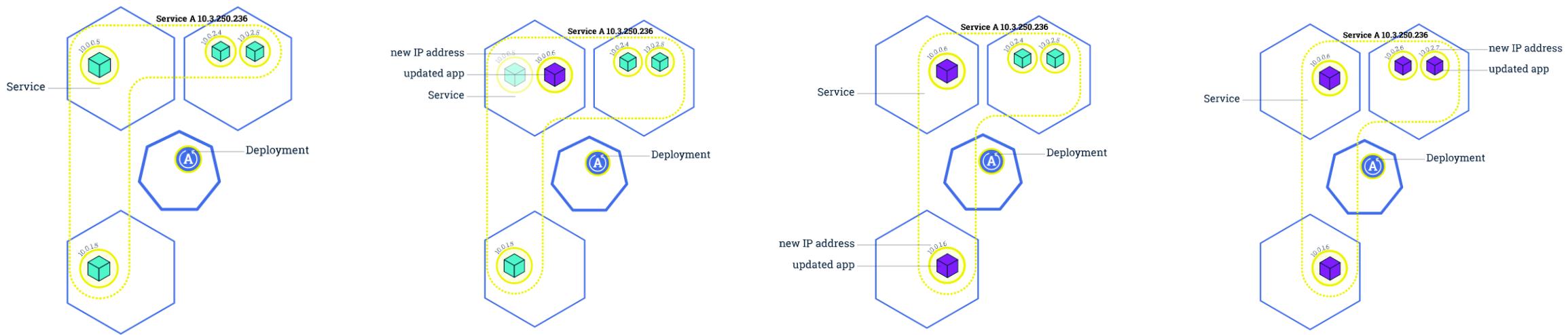
扩展应用

- 通过修改Deployment中副本的数量（replicas），可以动态扩展或收缩应用：
- 这些自动扩展的容器会自动加入到service中，而收缩回收的容器也会自动从service中删除。
- `kubectl scale --replicas=3 deployment/nginx-app`
- `kubectl get deploy`



滚动升级

- `kubectl rolling-update frontend-v1 frontend-v2 --image=image:v2`



- 在滚动升级的过程中，如果发现了失败或者配置错误，还可以随时回滚：
- `kubectl rolling-update frontend-v1 frontend-v2 --rollback`

资源限制

- Kubernetes通过cgroups提供容器资源管理的功能，可以限制每个容器的CPU和内存使用，比如对于刚才创建的deployment，可以通过下面的命令限制nginx容器最多只用50%的CPU和128MB的内存：
- \$ kubectl set resources deployment nginx-app -c=nginx --limits=cpu=500m,memory=128Mi
- deployment "nginx" resource requirements updated

这等同于在每个Pod中设置resources limits

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: nginx
    name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
      resources:
        limits:
          cpu: "500m"
          memory: "128Mi"
```

健康检查

- Kubernetes作为一个面向应用的集群管理工具，需要确保容器在部署后确实处在正常的运行状态。Kubernetes提供了两种探针（Probe，支持exec、tcpSocket和http方式）来探测容器的状态：
- LivenessProbe：探测应用是否处于健康状态，如果不健康则删除并重新创建容器
- ReadinessProbe：探测应用是否启动完成并且处于正常服务状态，如果不正常则不会接收来自Kubernetes Service的流量
- 对于已经部署的deployment，可以通过`kubectl edit deployment/nginx-app`来更新manifest，增加健康检查部分：

健康检查spec

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx-default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          imagePullPolicy: Always
          name: http
          resources: {}
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
          resources:
            limits:
              cpu: "500m"
              memory: "128Mi"
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 15
        timeoutSeconds: 1
      readinessProbe:
        httpGet:
          path: /ping
          port: 80
        initialDelaySeconds: 5
        timeoutSeconds: 1
```