



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

SSE316 : 云计算技术 Cloud Computing Technology

陈壮彬

软件工程学院

<https://zbchern.github.io/sse316.html>



总票代

【海口】周杰伦 2023嘉年华世界巡回 演唱会--海口站

¥500-¥2000

天津站

2020.04.25-2023....

海口站

06.30-07.02

呼和浩特站

08.17-08.20

太原站

09.21-09.24

2023.06.30-07.02

约120分钟 (以现场为准)



海口市 | 海口市五源河体育中心体育场

海南省海口市秀英区长流镇长流一号路五源河体育场

无座位图 共有 3 个演出 ▶

服务 ✕ 不支持退 ✕ 不支持选座 ✓ 快递票 ✓ 纸质发票

预售 本商品为预售商品，正式开票后将第一时间为您配票



大麦想看

已想看

128.6 万人

超过 99% 同类演出



435.1km



详情 评价 须知 推荐

温馨提示

1. 购买前请您提前规划好行程，做好相应准备，以免影响您的正



暂时无票
查看其它

春回演艺现场|周杰伦演唱会30秒售罄,音乐节遍地开花

#周杰伦演唱会门票#话题冲上热搜。“付款界面还没打开,就已经提示售罄。”一位手持三台手机同步开抢的歌迷哀叹,不知道是谁抢...

新浪新闻 2月28日

数百万人疯抢,30秒卖光!这位男星又刷屏了!

30秒抢光数万张票 大麦网显示,目前周杰伦2023嘉年华世界巡回演唱会共确定了4个城市的演出。分别为6月份的海口、8月份呼...

Globe 光明网 2月28日

30秒空!演出行业重启,一手复苏一手抢票难,含“杰伦...

“激动的心,颤抖的手,又要抢票了!”周杰伦微博超话里一片尖叫。2月27日, #周杰伦演唱会、大麦#等词条冲上微博热搜。大麦网...

风口 第一风口 2月28日

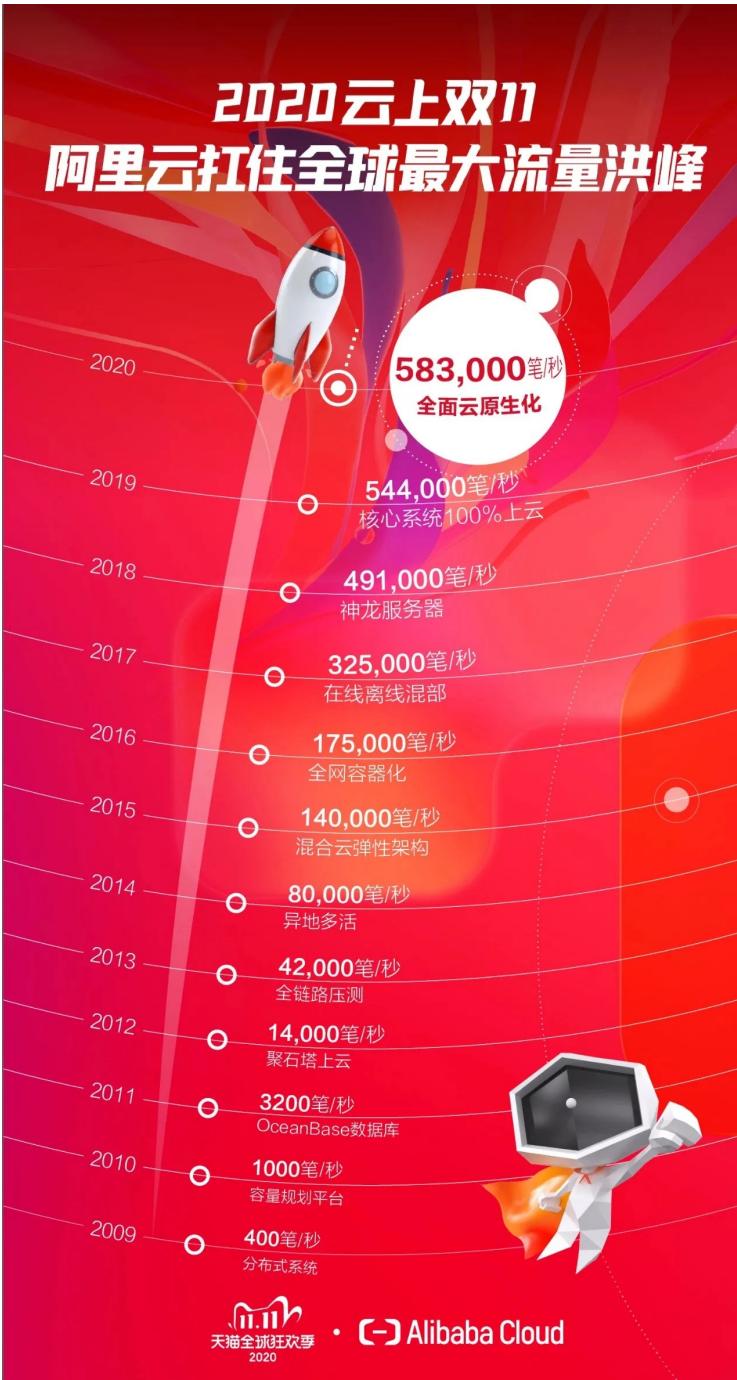


帮助



我想看

缺货登记



2009年 - 分布式系统 ↘

400笔交易/秒

2020年 - 全面云原生化 ↘

583,000笔交易/秒 !



1457倍 !

2020年春运，40天里，12306网站**最高峰日点击量为1495亿次**，相当于每个中国人都要点上100次，平均1秒的点击量约为170万次，最高单天售出1443万张车票。



VS



2020年双十一淘宝订单峰值为每秒58.3万笔，如果该峰值维持全天（当然不可能），全天成交503亿单，其**流量规模也仅是12306的1/3**。



云原生技术

Cloud Native



服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes



服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

云上服务的发展



在2015年之前，对于大多数应用来说，云只是一个用于计算的场所，开发人员要做的仅仅是将原来在私有数据中心的应用和数据迁移到云端。



在迁移过程中，应用无须重新编写，只需要重新部署，因为云平台提供的计算、存储、网络等，完全兼容应用迁移之前的运行环境。

云上服务的发展 (cont' d)



也就是说，软件的架构没有发生变化，只是软件运行的平台发生了变化，对应用无感。



因此，**软件需要自行解决分布式场景下的各种问题**，包括稳定性、组件或服务之间的数据同步、整体的高可用或容灾、资源利用率不高、端到端链路追踪等。



云上服务的发展 (cont' d)



云平台为了帮助应用解决上述分布式复杂性问题，不断推出高质量的组件和服务。

但是由于应用架构本身没有变化，云平台只能从局部提升应用的效率。那么“木桶效应”便会凸显，即整个系统的性能将取决于能力最差的组件。

云上服务的发展 (cont' d)



应用原先的架构是为了“老”的分布式运行环境而设计的，而云环境与传统的分布式环境不同。

需要对这些“老”架构以及围绕这些架构建立的技术栈、工具链、交付体系进行升级，依托于云技术栈将其重新部署、部分重构甚至全部重写，才能将应用变成“云原生的”，从而保证能够充分利用云计算的能力。

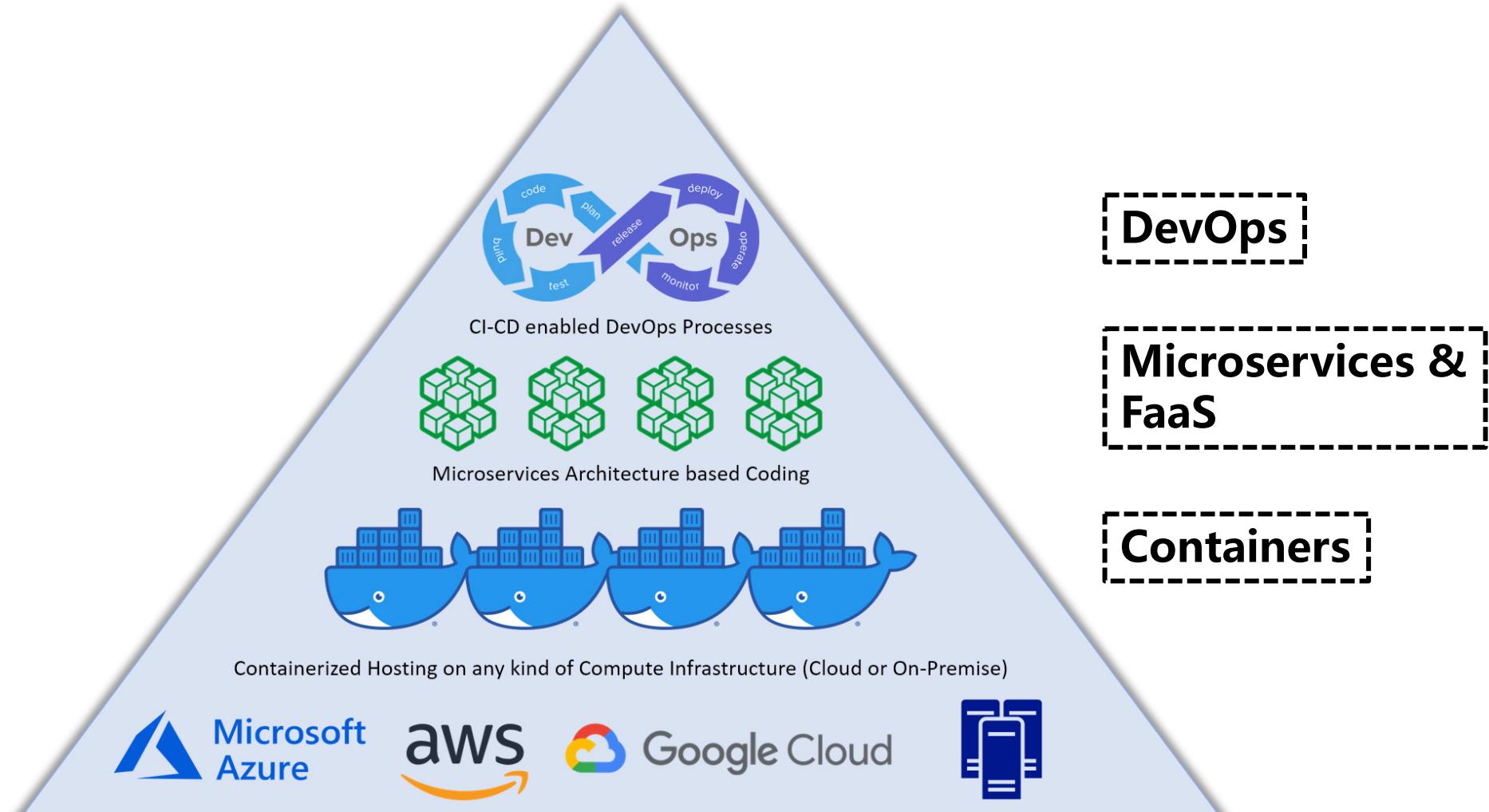
云原生的定义



云原生架构与技术是一种设计、构建和运行工作负载的方法，这些工作负载在云中构建并充分利用云计算模型。

Cloud-native architecture and technologies are an approach to designing, constructing, and operating workloads that are built in the cloud and take full advantage of the cloud computing model.

云原生技术栈



Cloud Native Technologies Pyramid

云原生的意义



云原生是云计算的再升级

DevOps

云原生应用鼓励DevOps方法，以加速应用程序的开发和部署。

微服务

云原生应用采用微服务架构，便于开发、测试和维护，提高系统可扩展性和弹性。

容器化

云原生应用使用容器技术进行部署，简化了部署过程并提高了应用程序的可移植性。



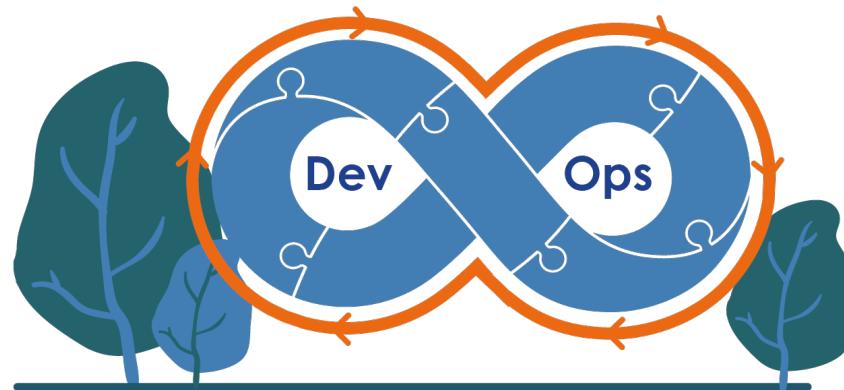
服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

DevOps



DevOps是一种跨职能的软件开发和运维方法，旨在强化**开发（Development）**和**运维（Operations）**团队之间的协作和沟通。通过自动化基础设施部署和持续集成/持续部署（CI/CD）等流程，DevOps可以提高软件交付的速度、质量和可靠性。





传统应用开发方式

- 部署和运维过程独立
 - ✓ 在传统的运维方式中，开发和运维团队分离，导致**沟通和协作效率低下**。开发人员关注代码开发，而运维人员负责代码的部署和维护。
- 手动操作
 - ✓ 以前的运维方式通常依赖于手动操作，如手动部署软件和手动执行测试。这可能导致部署速度慢，**容易出现人为错误**。
- 长周期发布
 - 软件发布周期较长，新功能和修复的部署可能需要等待数周甚至数月。这使得开发和运维团队**难以快速响应市场需求和解决问题**。

DevOps运维实践



Infrastructure as Code

- ✓ IaC 是一种管理和配置基础设施的方法，将基础设施的定义和配置编写为代码。
- ✓ 通过使用类似于软件开发的方法（例如版本控制、自动化测试和持续集成/持续部署），IaC 可以帮助开发和运维团队更快速、可靠、可重复地创建和管理基础设施资源。

DevOps运维实践



- ✓ 将声明式的基础设施和应用配置**存储在 Git 仓库中**，用于管理它们的状态。
- ✓ 核心思想是将声明式的基础设施和应用配置**存储在 Git 仓库中**，通过**持续集成/持续部署（CI/CD）流程实现自动化**。



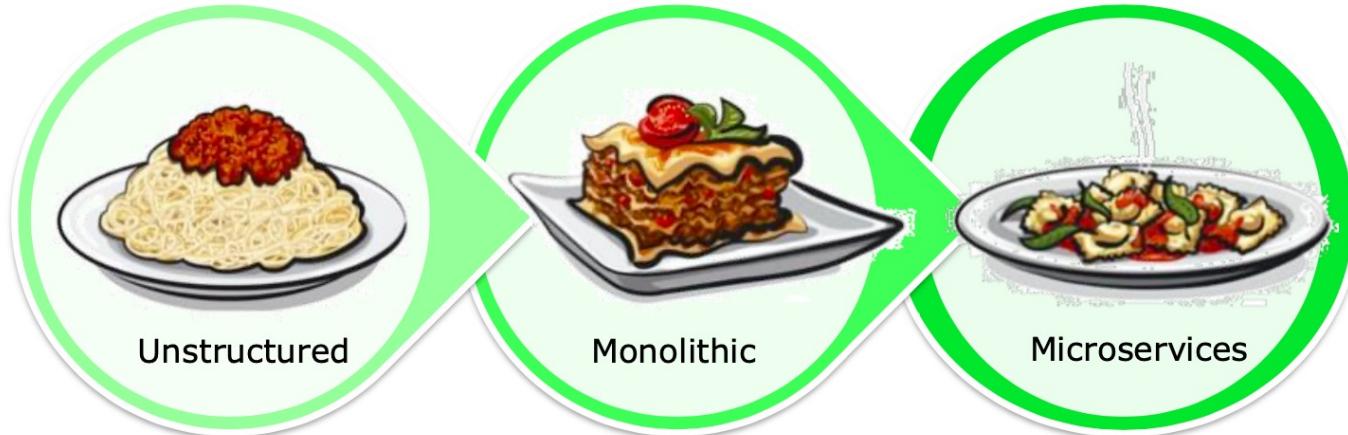
服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

微服务



微服务（Microservices）是一种软件架构风格，它将一个大型应用程序分解为多个较小、独立的、可单独部署和扩展的服务。每个微服务负责处理特定的业务功能，它们之间通过轻量级的通信协议（如 HTTP/REST、gRPC 等）相互协作。

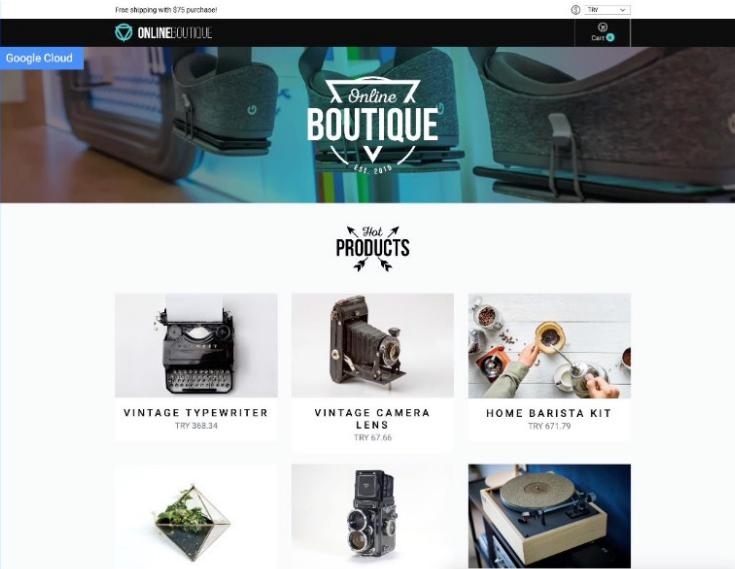


软件架构演进

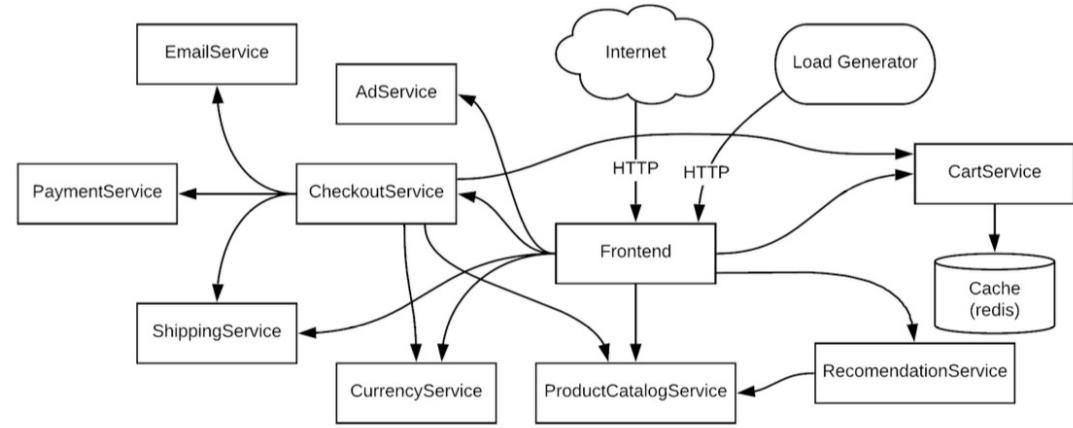
微服务例子



User's view (frontend)



System admin's view (backend)



传统部署方法：把整个单体应用部署在一个虚机上

微服务部署方法：把每个组件部署在不同的容器里

微服务的优点



敏捷性

每个服务有独立的开发、测试和部署流程，不会影响其他服务的功能。

可扩展性

对瓶颈微服务进行精准扩容，更容易地适应不断变化的业务需求。

可靠性

每个微服务都是独立的，发生故障时只会影响单个服务，而不会影响整个应用程序。

等等...

“有服务器”开发



- 开发者需要关注

前后端交互

资源管理

性能优化

部署和版本
控制

故障排查和
监控

...





Serverless和FaaS

Serverless

- ❖ 一种云计算执行模型，它使开发者能够构建和运行应用程序，而**无需关注底层基础设施的管理和运维。**
- ❖ 在 Serverless 模型中，云服务提供商根据实际的使用量动态分配资源，并按需收费。这种模型降低了运维复杂性，使开发团队能够**专注于应用程序的功能开发。**

FaaS

- ❖ FaaS (Function as a Service) 是Serverless架构中的一个关键组件，允许开发者在Serverless环境中**部署和运行单一功能或一小段代码。**
- ❖ FaaS提供商会负责**代码的运行环境、资源分配、扩展以及其他底层设施问题。**开发者只需编写函数逻辑，并根据特定事件（如HTTP请求等）**触发函数执行。**
- ❖ 函数被触发时加载到内存，完成后销毁。

FaaS例子



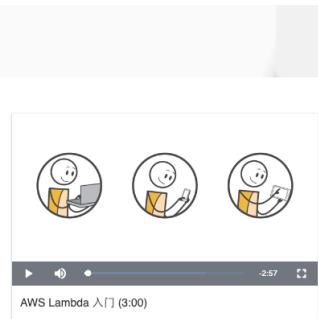
AWS Lambda Introduction

AWS Lambda 入门

此处提供了相关教程和文档，以指导您开始使用 AWS Lambda 构建无服务器应用程序。此外，您还可以了解一些无服务器应用程序开发工具，例如 [AWS 无服务器应用程序模型 \(SAM\)](#) 或 [AWS Cloud9](#)。

另一种入门方法是使用 [AWS Serverless Application Repository](#)，它能帮助您快速部署预构建应用程序。

如要进一步深入到具体的使用案例中，您还可以找到 [Web 应用部署](#)、[数据处理](#)、[移动端部署](#) 和 [边缘计算](#) 的相关资源。



<https://aws.amazon.com/cn/lambda/getting-started/>

Run a Serverless "Hello, World!" with AWS Lambda

Run a Serverless "Hello, World!" with AWS Lambda

TUTORIAL

Overview

In this tutorial, you will learn the basics of running code on AWS Lambda without provisioning or managing servers. We will walk through how to create a Hello World Lambda function using the AWS Lambda console. We will then show you how to manually invoke the Lambda function using sample event data and review your output metrics.

Everything done in this tutorial is [Free Tier](#) eligible.

✓ AWS experience	Beginner
⌚ Time to complete	10 minutes
💲 Cost to complete	Free Tier
📌 Requires	<ul style="list-style-type: none">AWS accountRecommended browser: The latest version of Chrome or Firefox <small>[**]Accounts created within the past 24 hours might not yet have access to the services required for this tutorial.</small>
☁ Services used	AWS Lambda
>Last updated	August 23, 2022

<https://aws.amazon.com/cn/getting-started/hands-on/run-serverless-code/>



服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

全虚拟化



优势

- ❖ 兼容现有应用程序
- ❖ 支持不同操作系统
- ❖ 每个VM有自己的执行环境
 - 内核版本、库
- ❖ 极好的隔离性，有硬件机制支持
 - CPU、memory

缺点

- ❖ 运行客户OS所需的开销较大
 - 几百兆内存
 - CPU
- ❖ 需要配置并更新每一个客户OS
 - 运维开销不能忽略
- ❖ 客户OS重启时间长
 - 几十秒甚至更多
 - fire and die应用无法接受

轻量级虚拟化



提供全虚拟化所具备的良好特征（可扩展性、弹性、隔离），同时**消耗更少的资源**。

- 轻量级虚拟化适合下列场景
 - ✓ 传统虚拟机开销太大
 - ✓ 需要能够快速部署、迁移和销毁的隔离的环境，开销小或根本没有
 - ✓ 能够**垂直扩展**（同一台机器上部署数千个“轻量级虚拟机”）和**水平扩展**（在数据中心可用的许多不同机器上部署“轻量级虚拟机”）

如何实现轻量级虚拟化？



Linux
cgroups

Linux
namespace

这两项技术起初是为了增强进程间的隔离性，与虚拟化无关

然而，它们却可被用与创建一种新形式的轻量级虚拟化，同时避免了全虚拟化所需的开销

可控制的资源

- ✓ CPU and Disk quotas
- ✓ I/O rate and Memory limit
- ✓ Network isolation
- ✓ Check-pointing and live migration
- ✓ File system isolation
- ✓ Root privilege isolation



Linux cgroups

Linux cgroups (Control Groups) 是一项内核功能，供了对一组进程的资源限制、控制和统计的能力。这些资源包括 CPU、内存、存储、网络等。

资源限制：cgroups 可以对任务是要的资源总额进行限制。比如设定任务运行时使用的内存上限，一旦超出就发 OOM。

优先级分配：通过分配的 CPU 时间片数量和磁盘 IO 带宽，实际上就等同于控制了任务运行的优先级。

资源统计：cgroups 可以统计系统的资源使用量，比如 CPU 使用时长、内存用量等。这个功能非常适合当前云端产品按使用量计费的方式。

任务控制：cgroups 可以对任务执行挂起、恢复等操作。



Linux cgroups例子 – CPU资源

1. Top查看资源使用情况

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
30167	root	20	0	4220	732	663	R	100.0	0.0	1:23.32	test

一个死循环程序

```
#include <stdio.h>

int main(){
    long i=0;

    while(1){
        i++;
    }

    return 0;
}
```

2. 使用cpu.cfs_period_us和cpu.cfs_quota_us控制cpu

```
mkdir /sys/fs/cgroup/cpu/test
cd /sys/fs/cgroup/cpu/ test
ls
cgroup.clone_children  cgroup.procs  cpuacct.stat  cpuacct.usage  cpuacct.usage64
cat cpu.cfs_quota_us
-1
echo 100000 > cpu.cfs_period_us
echo 20000 > cpu.cfs_quota_us
```

cpu.cfs_period_us分配可用CPU资源的时间周期
cpu.cfs_quota_us控制某个cgroup中所有任务可运行的时间总量

3. Top查看资源使用情况

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
30167	root	20	0	4220	732	663	R	20.0	0.0	1:24.12	test

Linux namespaces



Linux namespaces 是一项内核功能，用于为运行中的进程提供**隔离的环境**（包括网络、文件系统等）。它允许在同一台机器上运行多个独立的进程，同时限制它们之间的互相影响。



Linux namespaces

Namespace	Constant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name
Cgroup	CLONE_NEWCGROUP	Control groups

IPC : 独立的 System V IPC 和 POSIX 消息队列，防止进程间的 IPC 资源冲突

Network : 独立的网络栈，包括网络接口、路由表和防火墙规则

Mount : 独立的文件系统挂载点视图

PID : 进程有独立的进程 ID 空间

User : 独立的用户和组 ID 空间

UTS : (UNIX 时间共享) 独立的主机名和域名

Cgroup : 独立的 cgroup 根目录视图



Linux namespaces例子

```
root@tryit-eminent:~# ps -ef
```

UID	PID	PPID	C	S	TIME	TTY	TIME	CMD
root	1	0	0	08:34	?		00:00:00	/sbin/init
root	73	1	0	08:34	?		00:00:00	/usr/bin/lxd --g
root	154	0	0	08:34	pts/1		00:00:00	bash
root	157	154	0	08:35	pts/1		00:00:00	ps -ef

通常情况下，linux系统中的init或systemd进程的pid为1

```
root@tryit-eminent:~# unshare --fork --pid --mount-proc /bin/bash  
root@tryit-eminent:~# ps -ef
```

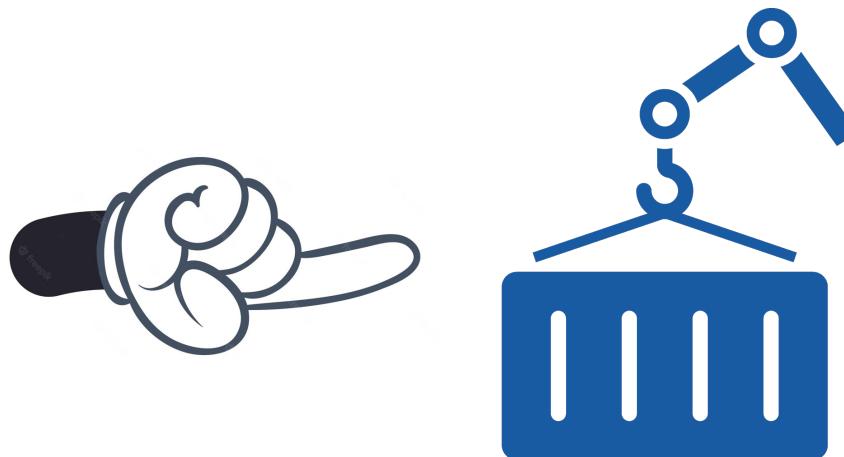
UID	PID	PPID	C	S	TIME	TTY	TIME	CMD
root	1	0	0	08:35	pts/1		00:00:00	/bin/bash
root	3	1	0	08:35	pts/1		00:00:00	ps -ef
root@tryit-eminent:~#								

在这个例子中，我们创建了一个新的pid namespace，但是没有启动init或systemd进程，因此PID为1的进程非二者之一

cgroups和namespaces的局限



- 非常灵活，但使用起来却很困难，需要很多命令/脚本去启动一个容器
- 无法保证应用的可移植性，不像虚拟机可以整体打包
- 它们可用于实现单个服务器上的虚拟化，却无法对整个数据中心进行统一管理
- cgroups和namespaces是很好的开始，但需要扩展到能够胜任数据中心级别的虚拟化管理

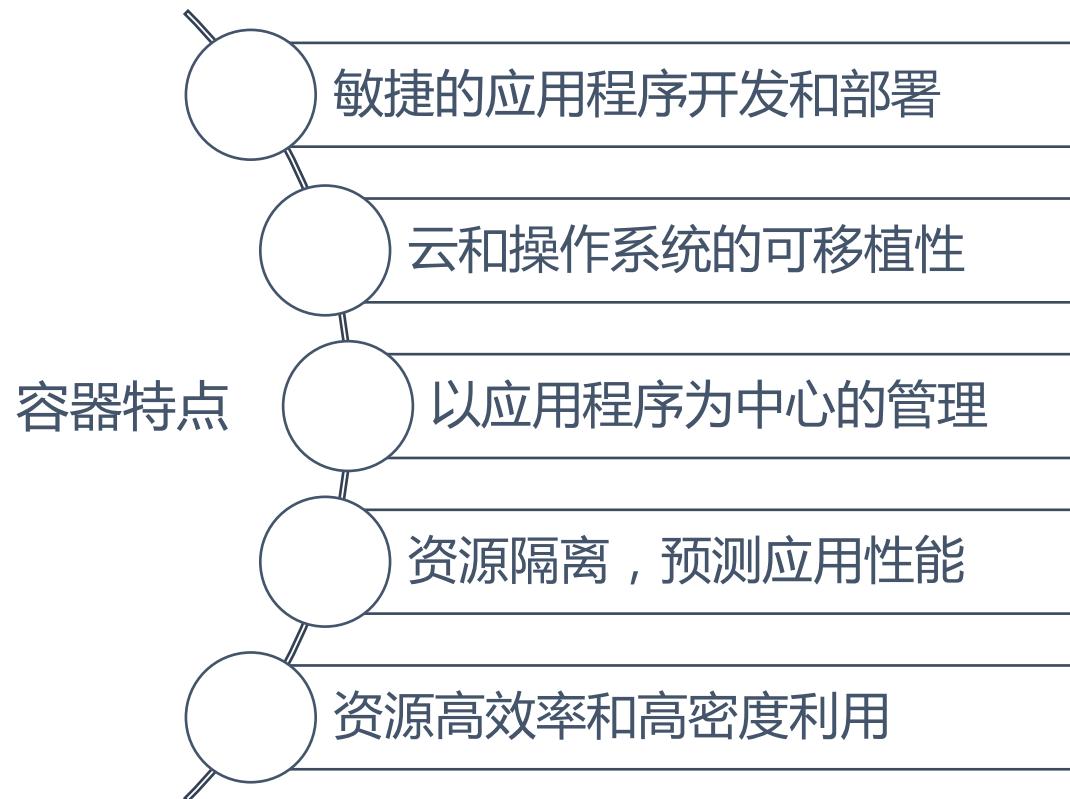


容器 (container)

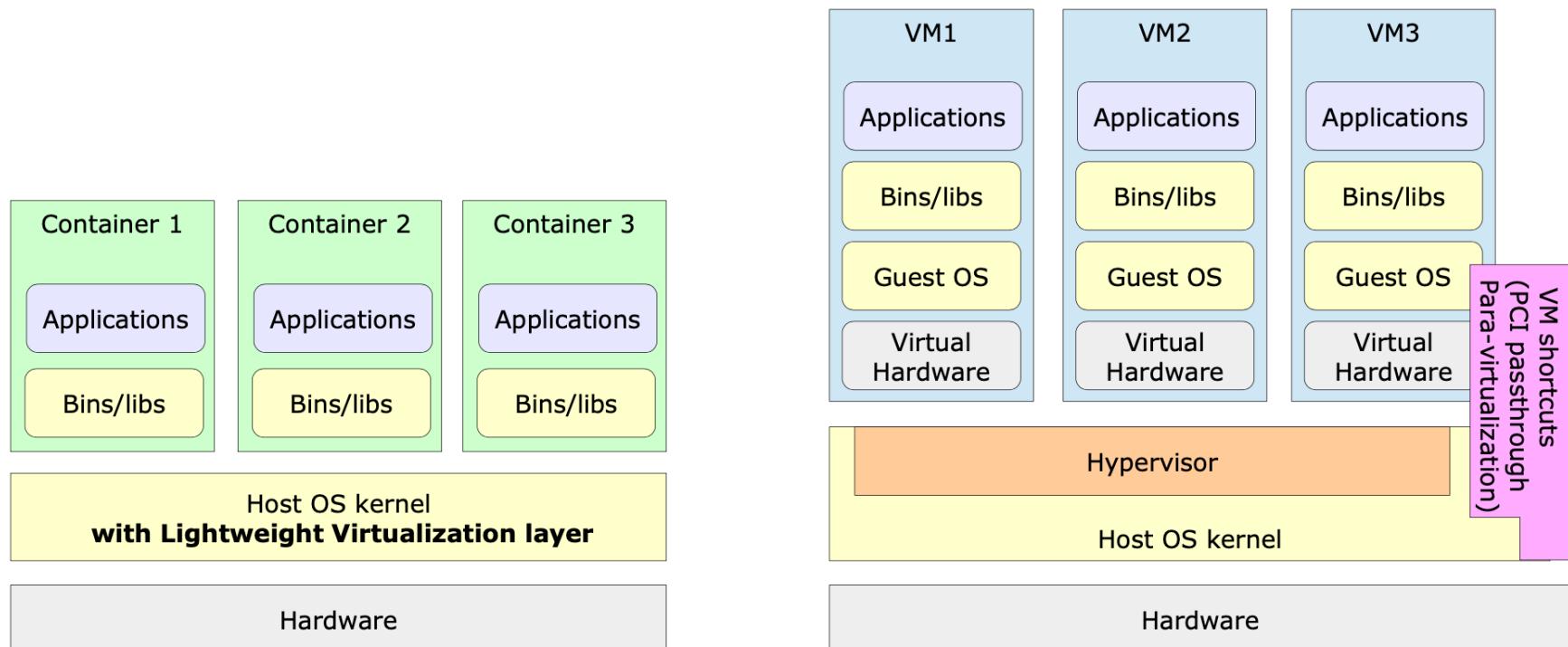
容器概述



容器提供轻量级虚拟化，允许隔离进程和各种计算资源，同时没有了完全虚拟化所具有的复杂性



Containers vs. Hypervisors





容器做什么

- ❖ 将进程分组在隔离的容器中（可以为其分配不同的资源）
- ❖ 与主机共享一个操作系统
- ❖ 从里面看，它们像VM
- ❖ 从外面看，它们是普通的进程

容器不做什么

- ❖ 不模拟硬件
- ❖ 不运行不同的Linux内核或操作系统
 - ✓ 可以运行不同的Linux版本 (?)，只要它们与主机有相同的内核
- ❖ 安全性不是一种开箱即用的功能，不能视为理所当然
 - ✓ 仍然可以保证适当程度的安全级别

Containers vs. VMs



Containers

- ❖ 比VM快 (重启、阻塞、管理和编排)
- ❖ 比VM轻量：更少的CPU、Memory，没有虚拟化开销（如指令模式）
 - ✓ 比VM更密集：资源消耗更低
 - ✓ 容器虚拟化技术实际上可以实现与本机相同的性能

VMs

- ❖ 更好的隔离性（比如可以防止内核级的问题）
- ❖ 更好的安全性，这是由于VM具有更少的攻击点（hypervisor通常很小）
- ❖ 可运行更多的操作系统

Containers vs. VMs (cont' d)



从不同角度看，容器可以同时是：

- ❖ 基础设施基元 (infrastructure primitives)，构成底层基础设施的基本构建模块
- ❖ 应用程序管理和配置系统 (Application management and configuration systems)



基础设施工程师认为是第一种



应用开发者认为是第二种

不同虚拟对象的比喻



Unique servers, Virtual Machines



Replaceable servers and VMs



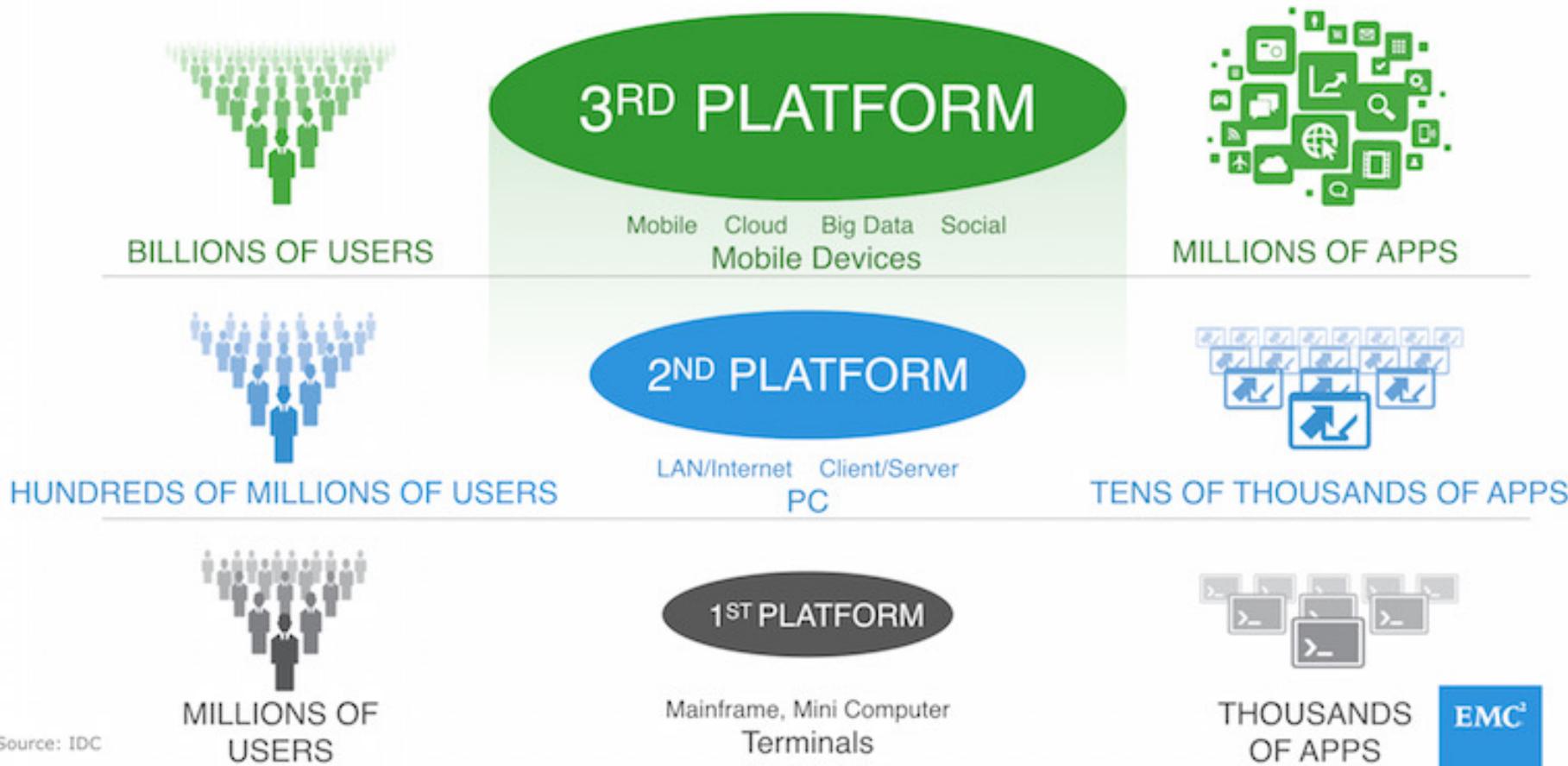
Even more dense and lightweight containers

Extremely short-lived functions (serverless)

计算平台的演进过程



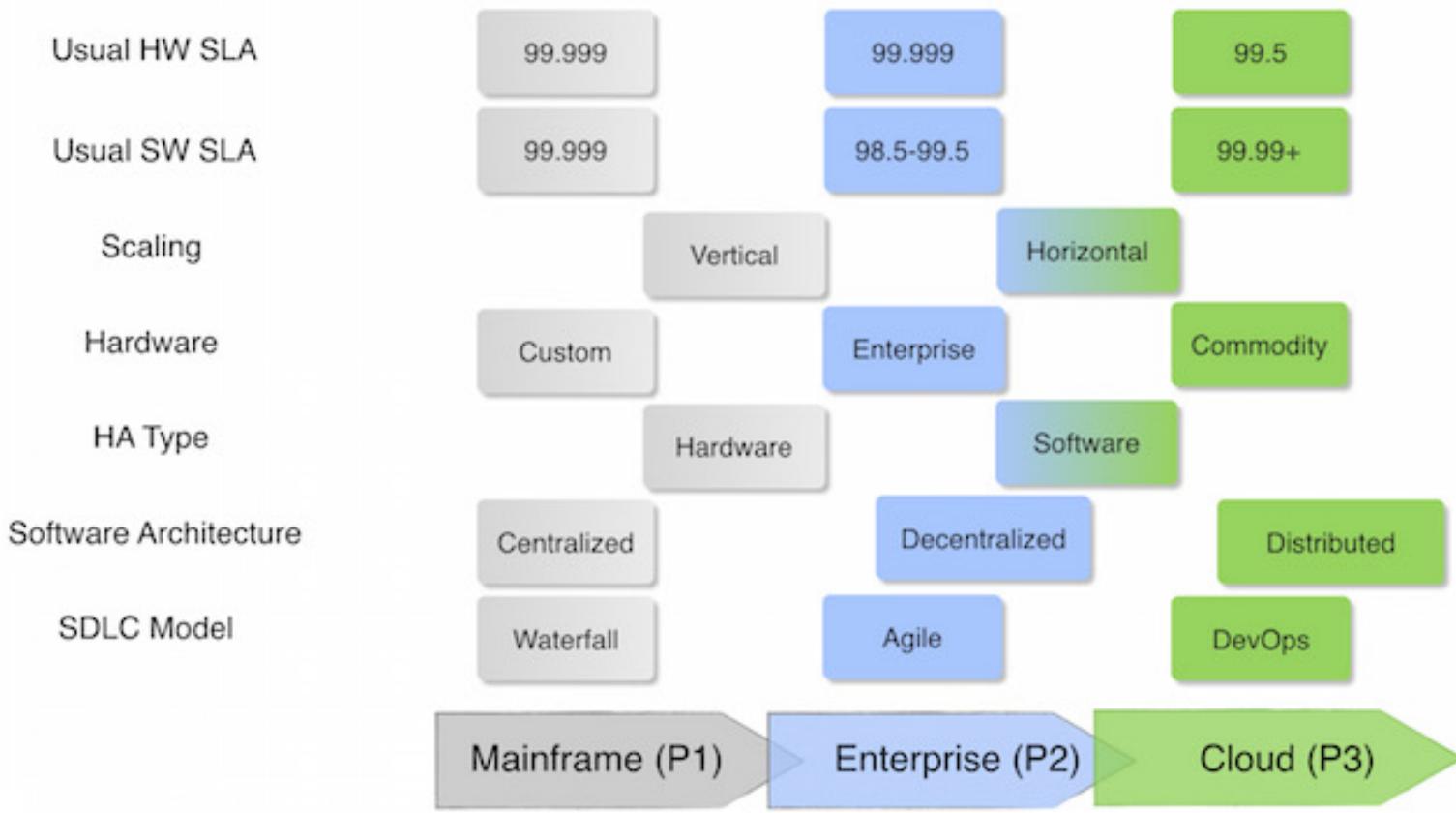
The Third Platform aka “Cloud Native”



不同计算平台的比较



Cloud Computing Patterns Are Different



EMC²

18

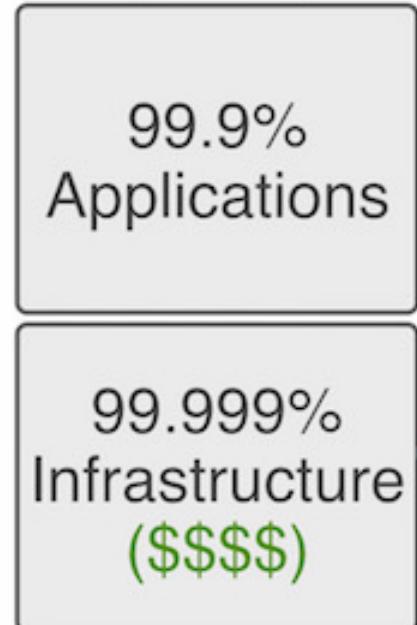
43

第二和第三平台间的差异

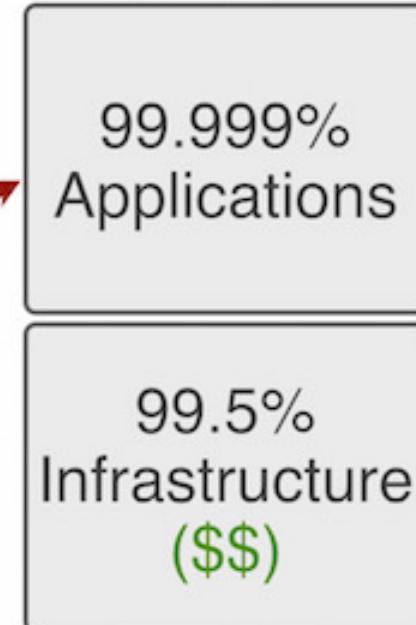


The Fundamental Difference from P2 -> P3

Second Platform Model
(inelastic)



Third Platform Model
(elastic)



DESIGN
FOR FAILURE!

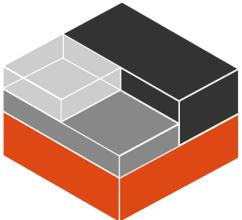


EMC²

20

44

Linux containers (LXC)

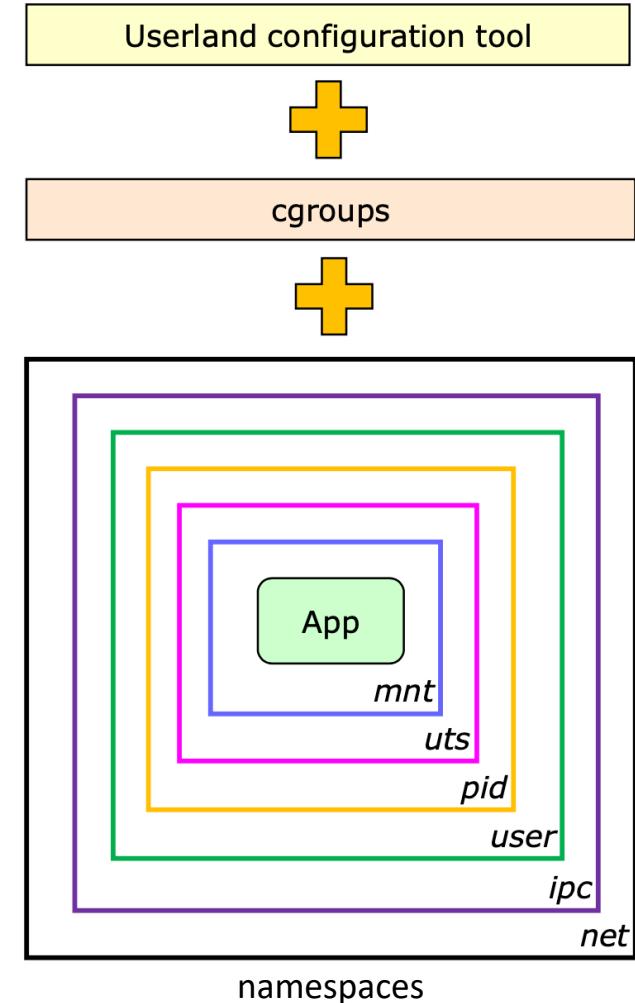


LXC是一个内核功能的集合，可以用来以不同的方式隔离进程，也是一个用户空间工具，可以使用所有这些功能来创建完整的容器。

LXC = cgroups + namespaces (+ user friendly config)

LXC features

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





LXC优点

- LXC本身只提供资源管理，其他特性需要手动添加，如
 - ✓ 需要隔离性，使用namespaces
 - ✓ 需要安全性，使用Apparmor或/和SELinux
- LXCI以一种用户友好的方式让用户定制需要的容器特性
 - ✓ LXC允许以配置文件的方式指定某个container的特征
 - ✓ 提供预定义的模版创建不同类型的容器

```
# Network configuration
lxc.net.0.type = veth
lxc.net.0.link = lxcbr0
lxc.net.0.flags = up
lxc.net.0.hwaddr = 00:16:3e:xx:xx:xx

# Limit memory to 512MB
lxc.cgroup.memory.limit_in_bytes = 512M

# Limit CPU usage to 2 cores
lxc.cgroup.cpuset.cpus = 0,1
```

配置文件部分

1. `ubuntu`: 用于创建基于 Ubuntu 的容器。
2. `debian`: 用于创建基于 Debian 的容器。
3. `centos`: 用于创建基于 CentOS 的容器。
4. `fedora`: 用于创建基于 Fedora 的容器。
5. `opensuse`: 用于创建基于 openSUSE 的容器。
6. `archlinux`: 用于创建基于 Arch Linux 的容器。
7. `gentoo`: 用于创建基于 Gentoo 的容器。
8. `alpine`: 用于创建基于 Alpine Linux 的容器。

```
sudo lxc-create -t ubuntu -n my-container
```

预定义模版及使用



LXC缺点

- 检查点（checkpointing）
 - ✓ LXC在检查点方面的支持相对较弱，依赖于CRIU (Checkpoint/Restore In Userspace) 技术来实现检查点和恢复功能，需要手动配置
- 迁移（migration）
 - ✓ LXC更注重操作系统级别的虚拟化，容器在不同的主机之间迁移可能会受到内核版本和底层系统差异的影响
- 资源隔离（resource isolation）
 - ✓ 资源隔离不够彻底，容易收到其他容器影响
- 可移植性（portability）
 - ✓ 镜像管理和分发机制不够成熟，可移植性较差



服务技术

- ❖ 云原生的概念
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

Docker



Build, Ship, Run, Any App Anywhere

From Dev



To Ops



Any App



Any OS



Windows



Linux

Docker概述



Docker专注于应用程序，简化了应用程序的部署和执行

Docker目标

- ❖ 应用隔离
 - ✓ 已定义资源的沙盒（通过cgroups和namespaces完成）
 - ✓ 进程拥有明确隔离的运行环境
- ❖ 统一的环境支撑应用的整个生命周期
 - ✓ 所有应用有相同的命令(run, stop, etc)和环境
- ❖ 透明及无缝的网络功能
 - ✓ Docker bridge, DHCP, transparent NAT, etc

Docker概述



Docker专注于应用程序，简化了应用程序的部署和执行

Docker的“不”

- ❖ 不是虚拟化引擎
 - ✓ 利用cgroups和namespaces实现虚拟化
- ❖ 不支持不同内核
- ❖ 不使用硬件基元 (hardware primitives)
 - ✓ 如CPU extensions



Docker专注应用

- 一直以来，打包应用程序进行部署很困难
 - ✓ 应用需要打包成多种格式 (e.g., for debian, redhat, etc)
 - ✓ 为每个操作系统版本创建多个包 (e.g., Ubuntu 19.10, Ubuntu 19.04) , 因为依赖关系可能不同，并且所需的库可能存在于不同的版本中
 - ✓ 当OS有新的版本出来就需要再创建包
 - ✓ 用户可能依旧会抱怨，因为打包过程容易出错
- 这甚至还没考虑不同的操作系统 (Windows, MacOS等)

Docker专注于应用程序，通过创建一个可在任何地方运行的**轻量级、可移植、自包含**的包来简化应用程序的部署和执行。



Docker vs. LXC

- Docker针对**应用程序**的部署进行了优化，而不是**机器**
- LXC专注于将容器作为**轻量级机器**，基本上是启动速度更快、需要更少RAM的服务器
- Docker容器可以跨机器移植，而LXC必须在目标机器上重建
- Docker只需一次build就可以在“任何地方”运行（尽管CPU架构必须匹配），并且与主机隔离
- Docker具有简化的（命令行）界面和更强大的管理工具
- Docker支持资源管理和隔离、版本控制、组件重用、自动构建和共享

Docker vs. LXC (cont' d)



Docker: additional features compared to LXC

- Application portability
- Union file system
- Automatic build
- Focus on running applications
- Versioning
- Component re-use
- Sharing (Docker server)
- Better documentation and ecosystem
- Integration with OpenStack, Kubernetes and all cloud vendors
- Resource management and isolation (already available also in LXC)

Image and containers



Docker Image

- ❖ 容器的不可变模板
- ❖ 可以上传至registry或从中下载
- ❖ Image的命名格式
 - ❖ [registry/] [user/] name [:tag]
 - ❖ 默认的tag是*latest*

Docker Container

- ❖ Image的一个实例
- ❖ 可以启动，停止或重启
- ❖ 能保持文件系统内的修改
- ❖ 新的Image可以根据当前的容器状态创建 (尽管使用Dockerfile是更好的方式)

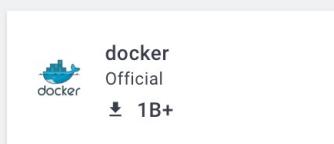
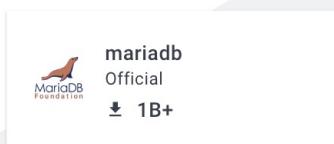
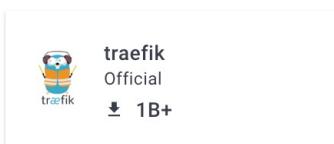
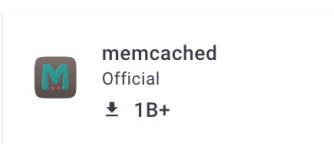
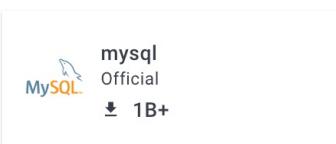
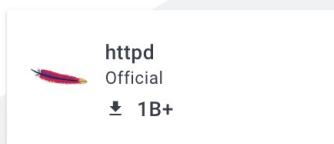


Docker Registry

类似于software repository，用于保存可用的Docker Image。
可以是公共的，也可以是私有的。

<https://hub.docker.com/>

Access the world's largest library of container images





Docker Registry (cont' d)

- 常用命令

- ✓ 在Registry中查找

```
docker search <term>
```

- ✓ 下载或更新本地的image

```
docker pull <image>
```

- ✓ 上传image到Registry

```
docker push <image>
```

```
[root@CentOS8 ~]# docker search java
```

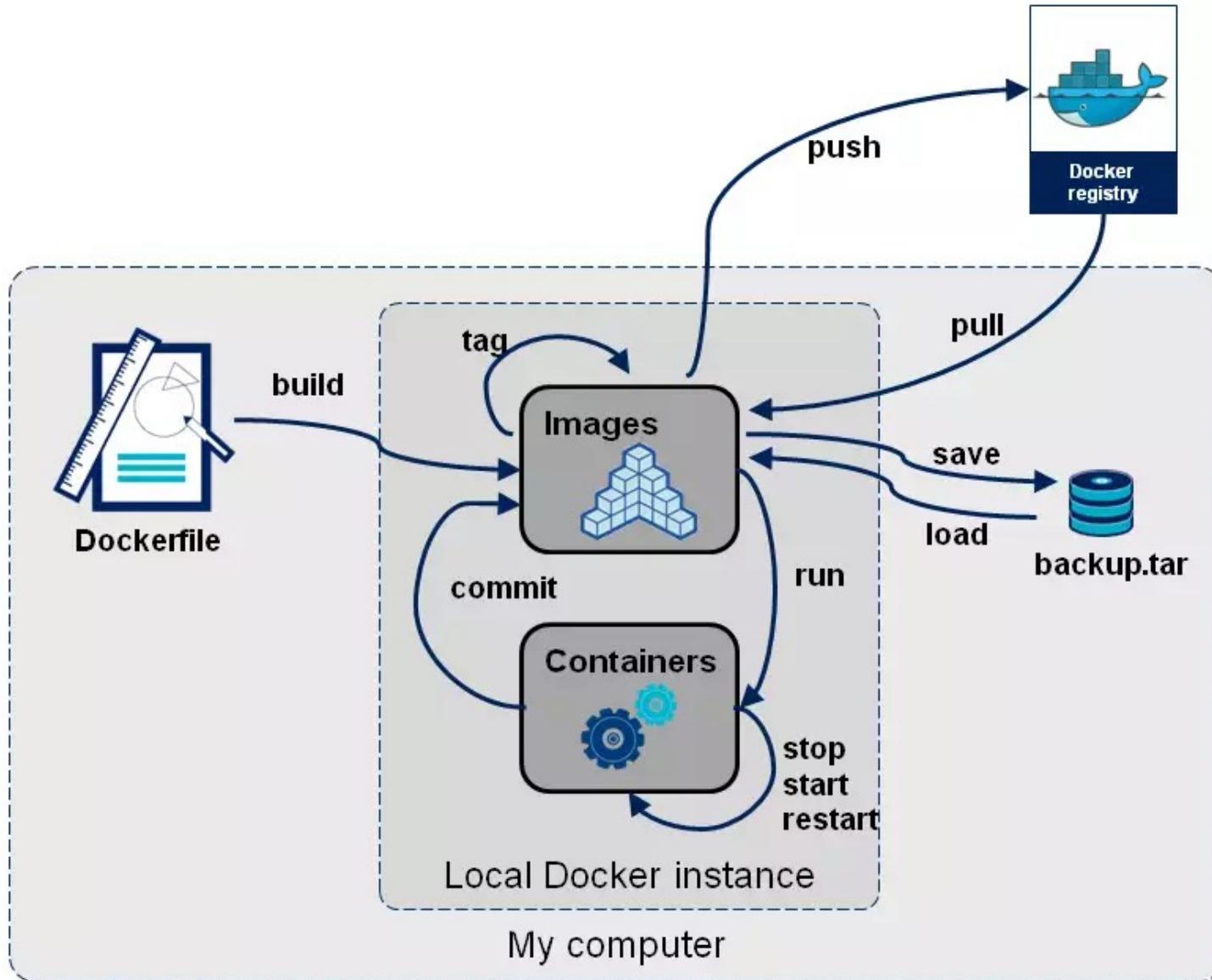
NAME	DESCRIPTION	STARS	OFFICIAL
node	Node.js is a JavaScript-based platform for s...	10810	[OK]
tomcat	Apache Tomcat is an open source implementati...	3184	[OK]
openjdk	OpenJDK is an open-source implementation of ...	3038	[OK]
java	DEPRECATED; use "openjdk" (or other JDK impl...	1976	[OK]
ghost	Ghost is a free and open source blogging pla...	1447	[OK]
lwieske/java-8	Oracle Java 8 Container – Full + Slim – Base...	50	
nimmis/java-centos	This is docker images of CentOS 7 with diffe...	42	
fabric8/java-jboss-openjdk8-jdk	Fabric8 Java Base Image (JBoss, OpenJDK 8)	29	



Docker其他常用命令

- Show running containers:
 - docker ps
- Show all containers (shows also terminated containers):
 - docker ps -a
- Show metadata of a running container
 - docker inspect <container>
 - It returns a JSON with all the information about the container (e.g., current IP/MAC address, the log path, the image name, etc)
 - docker inspect --format='{{.Image}}' <container> (to show a specific metadata)
 - Shows only a specific metadata from the above JSON
- Control your containers:
 - docker run <container..> (by far the most important command; it starts a new container)
 - docker start <container..> (start stopped container)
 - docker stop <container..> (stop gracefully a container with SIGTERM signal)
 - docker kill <container..> (kill running container by sending a SIGKILL signal)
 - docker rm <container..>

Docker生命周期





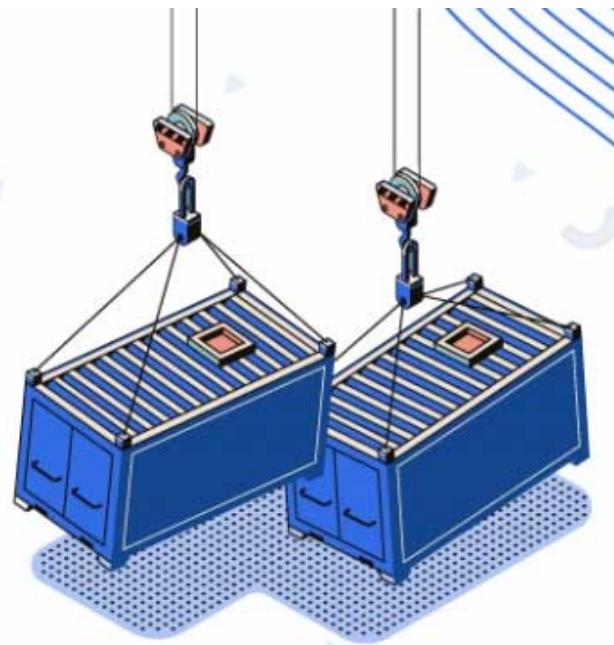
服务技术

- ❖ 云原生
- ❖ Devops
- ❖ Microservices and Serverless
- ❖ Container
- ❖ Docker
- ❖ Kubernetes

容器编排 (Orchestration)



- ✓ 编排是指对一组容器的**自动化管理、协调和控制**，以便在分布式环境中实现应用的**高效部署、扩展和运行**。
- ✓ 容器编排能够**处理容器的生命周期管理**，包括创建、部署、扩展、更新和终止等操作。
- ✓ 通过容器编排，开发人员和运维人员可以在复杂的分布式系统中**轻松地管理和维护多个容器实例**。



部署和编排容器



方式一：手动编排

- ❖ SSH进入机器并运行Docker
- ❖ Pros : 简单，随处可用，不需要特殊工具，易于理解
- ❖ Cons : 不自动化、不易复制、不可扩展、不可自我修复
- ❖ 不过，并不是每个人都需要“大规模”操作，因此这种方法也没那么糟糕

部署和编排容器 (cont' d)



方式二：通过脚本

- ❖ Puppet , Chef , Ansible , Salt或者人工写的脚本
- ❖ Pros : 与现有环境集成，结果易于理解，可重复
- ❖ Cons : 手动调度，不可扩展，无法自我修复，通常不可移植



方式三：特定的容器编排器

- ❖ Mesos , Kubernetes , Docker Swarm等
- ❖ Pros : 自动化、可复制、自我修复、可扩展、一般可移植
- ❖ Cons : 一些开销，需要新的工具和培训，更复杂的结果



Docker组件

- Docker Swarm

- ✓ Docker自带的容器编排器 (orchestrator)
- ✓ 它将Docker主机池变成一个虚拟的单个主机，处理整个数据中心的资源

- Docker Compose

- ✓ Compose是一个用于定义和运行多容器Docker应用程序的工具
- ✓ 利用YAML文件去配置应用的各个服务组件
 - 比如web frontend , database engine , storage



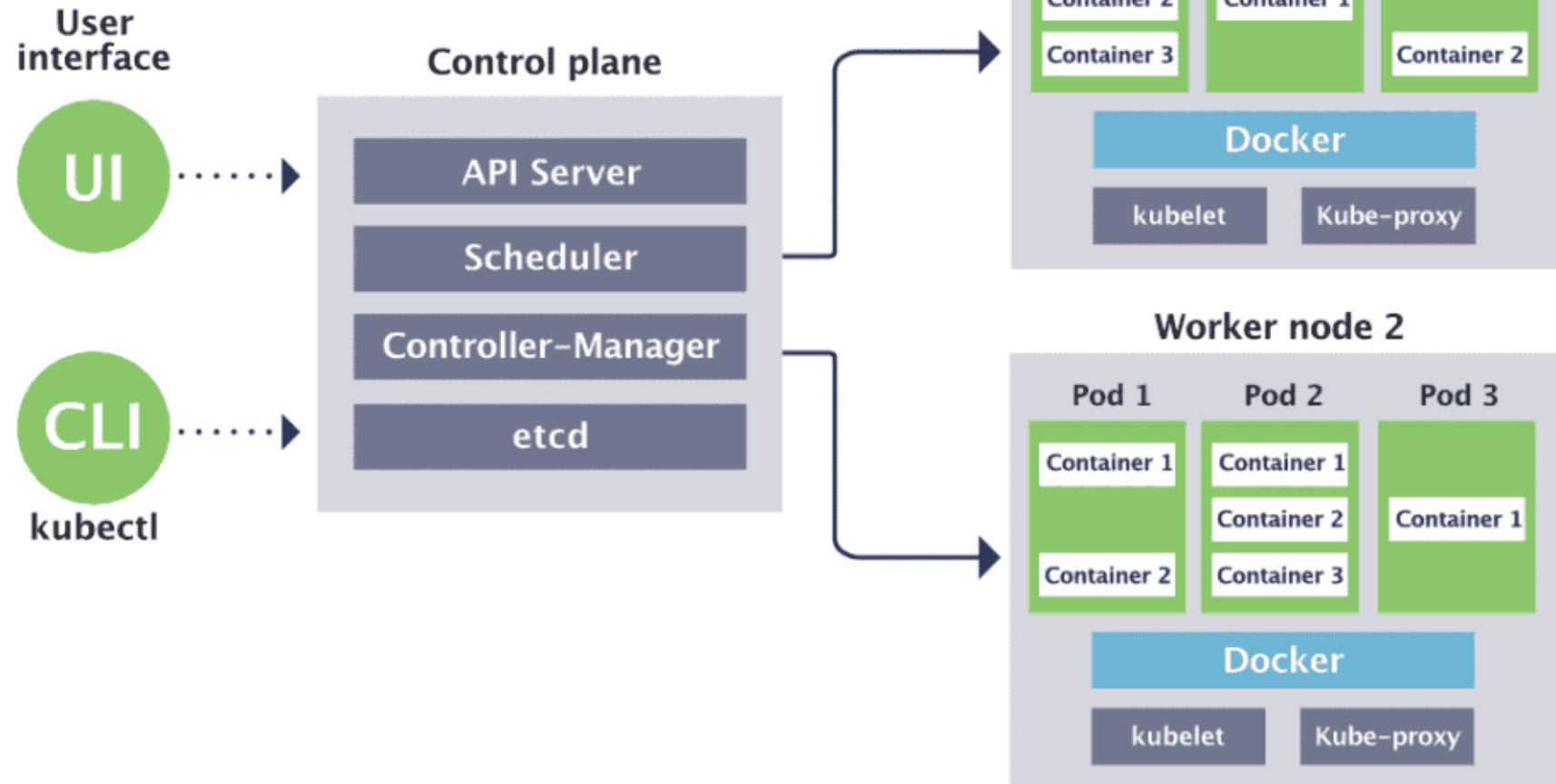
kubernetes

上述Docker自带的两个组件
已几乎被kubernetes替代

Kubernetes 架构



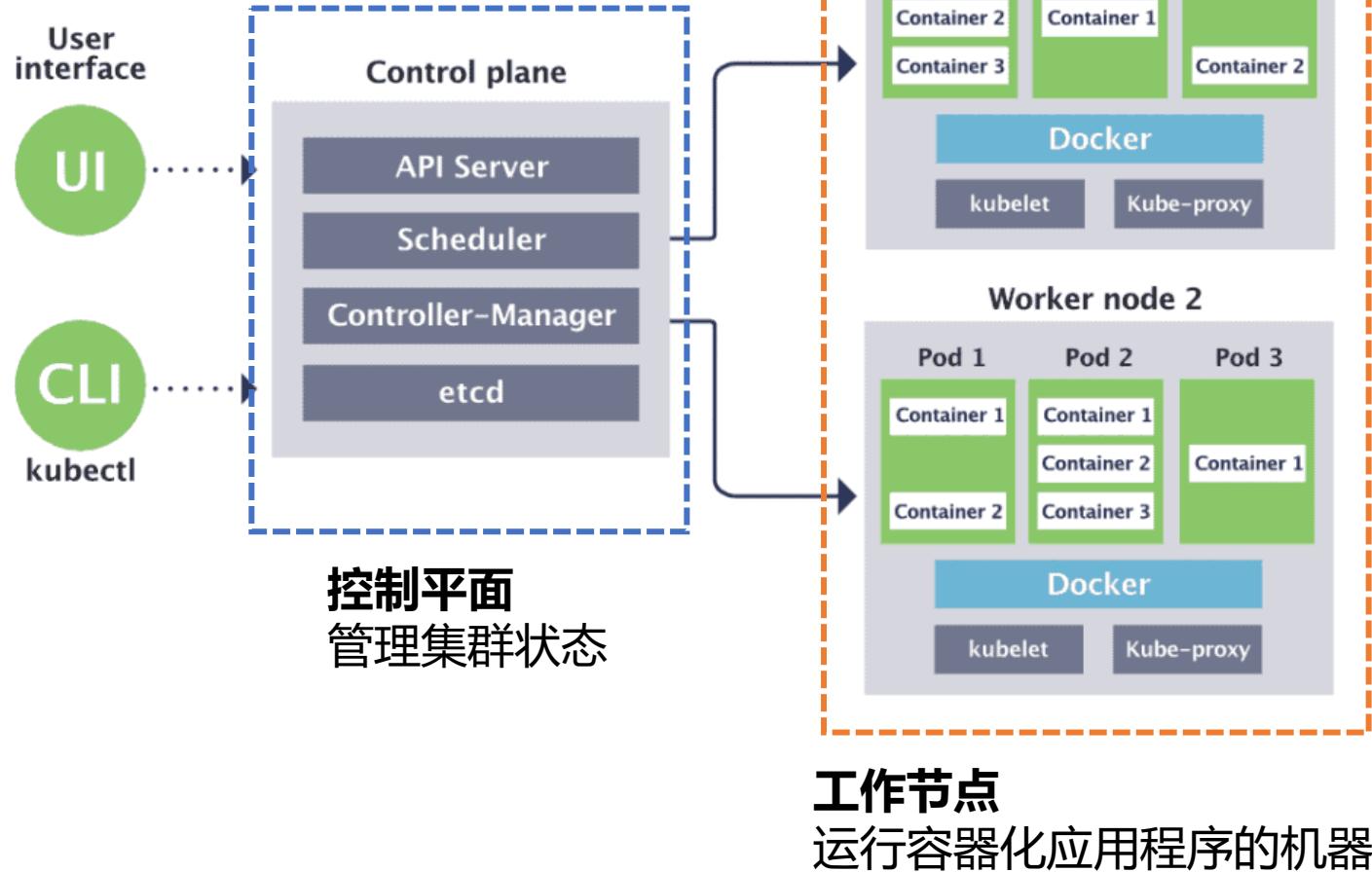
Kubernetes architecture



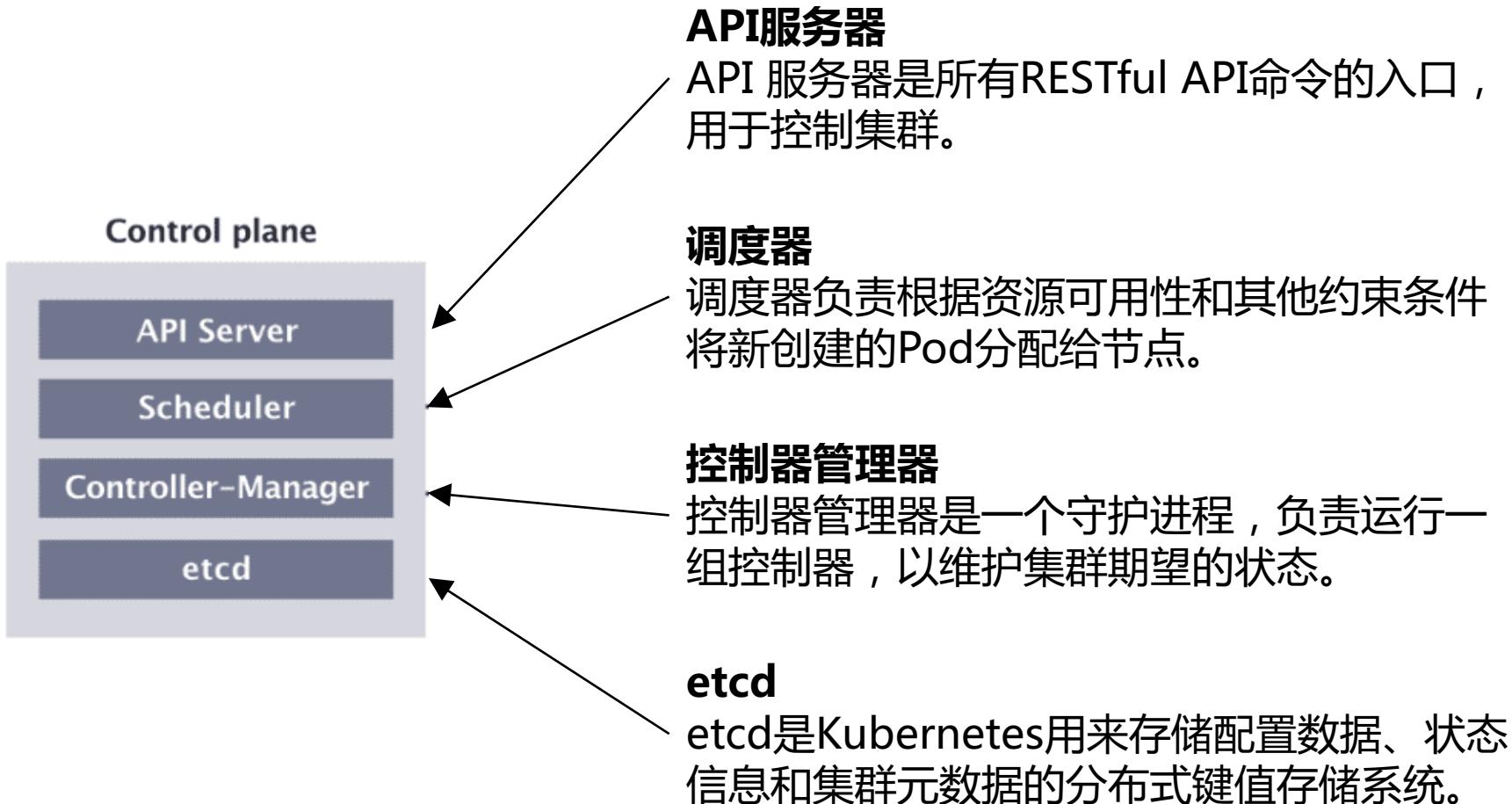
控制平面和工作节点



Kubernetes architecture



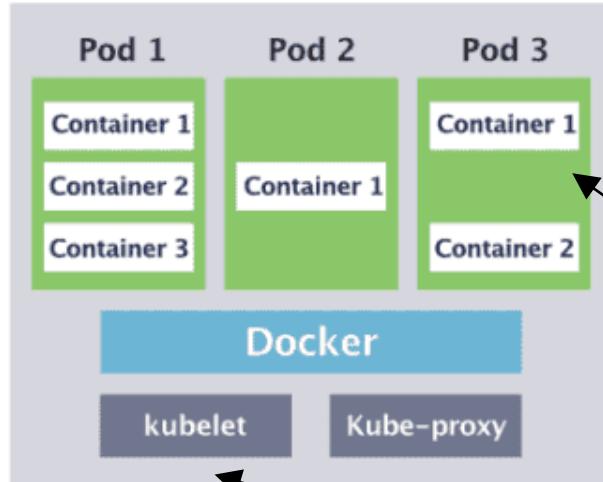
控制平面组件





工作节点组件

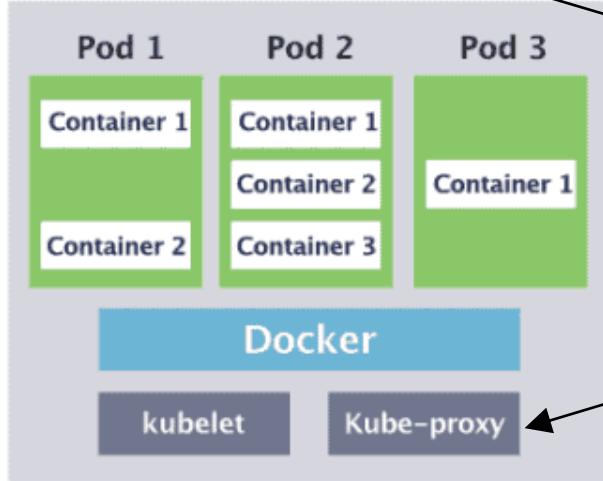
Worker node 1



Pod

Pod是K8s中**最小的部署单元**，一个Pod可以包含一个或多个紧密关联的容器，它们之间可以方便进行**通信和数据共享**。

Worker node 2



Kubelet

Kubelet是一个在每个工作节点上运行的代理，它确保容器按照API服务器的指示在Pod中正确运行。

Kube-proxy

Kube-proxy是一个网络代理，它实现了K8s服务在每个节点上的**网络抽象**。它负责处理工作节点上的**网络规则**和**负载均衡**。



中山大學 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬
软件工程学院

<https://zbchern.github.io/sse316.html>