法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，讲师及小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

关注 小象学院

从Borg到Kubernetes

难易.钟成

https://github.com/HardySimpson

自我介绍

Github ID： HardySimpson

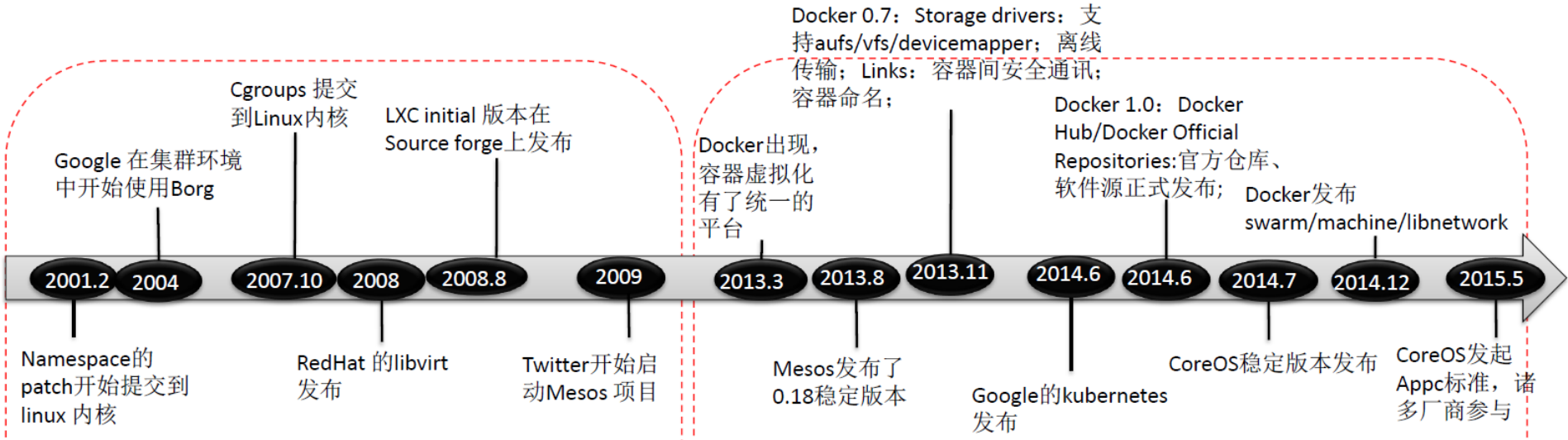
个人博客：难易@开源中国

毕业于复旦大学物理系，开源项目zlog/pilotage作者，从2014年开始致力于容器、PaaS平台的软件设计及开发。精通kubernetes/etcd项目，对云计算的产品及未来发展非常感兴趣

1. 容器的起源和历史
2. Borg在Google的发明和使用
3. Kubernetes简单介绍
4. 云原生应用、微服务、PaaS、DevOps、Serverless等相关概念关系

1. 容器的起源和历史
2. Borg在Google的发明和使用
3. Kubernetes简单介绍
4. 云原生应用、微服务、PaaS、DevOps、Serverless等相关概念关系

1.容器：一段简史



阶段1：聚焦IaaS层价值，把容器当作更轻量级的虚拟机来用

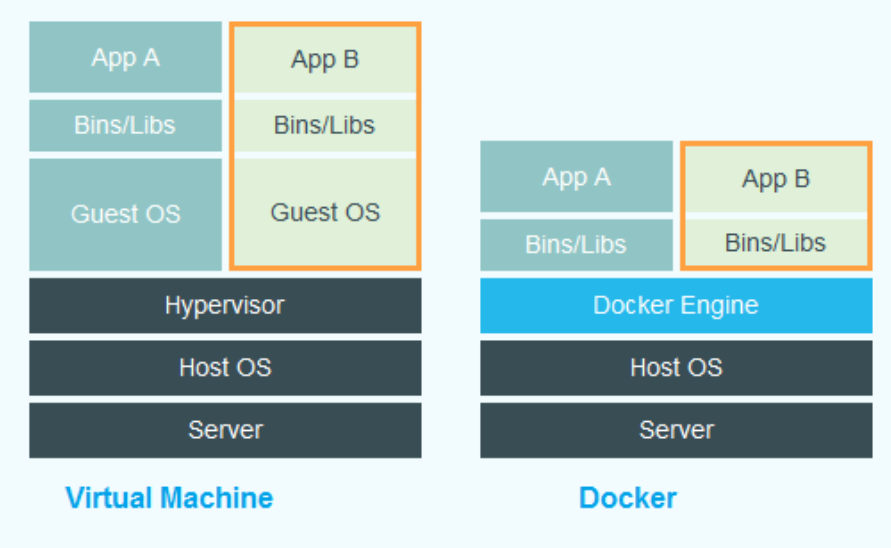
Linux版本 的LXC能力解决了**应用运行时进程级资源资源隔离**的问题，仍然是以机器为中心的传统思路，业界未大规模使用

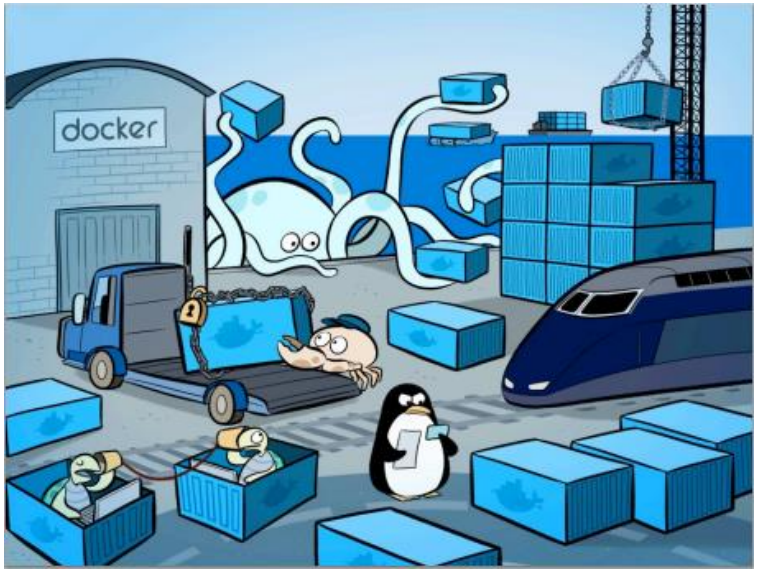
阶段2：聚焦PaaS价值，以应用为中心，应用分发标准化，升级与 DevOps更易实现

Docker解决了**App+运行环境(Bins/Libs)整体打包**的问题，确定了分层镜像格式，简化了Devops流程

K8S/Mesos解决了**应用生命周期管理框架**的问题，使得应用分布式运行更加容易，资源分配更加灵活，管理方式更加便捷

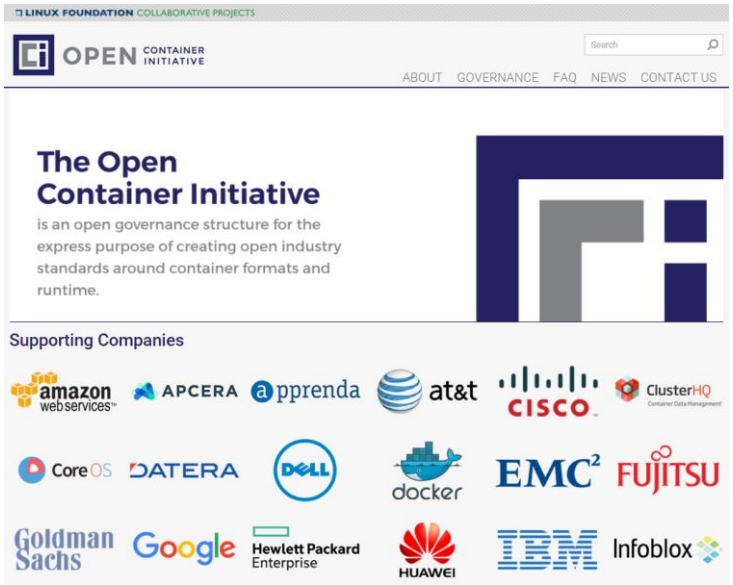
1.容器是什么？

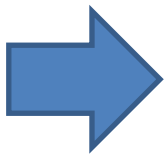


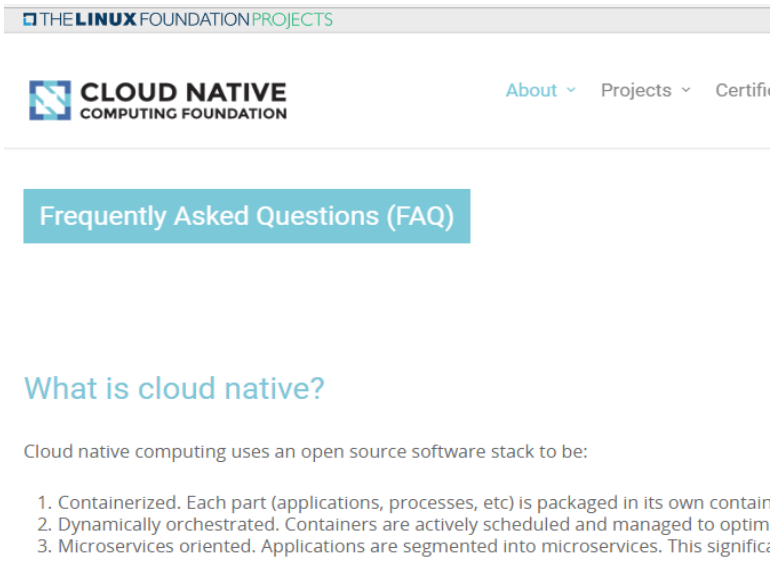


- 容器是操作系统内核自带能力，早已存在。基于linux内核已实现的轻量级高性能资源隔离机制(cgroups, namespace, lxc)
- 虚拟机是操作系统级别的资源隔离，容器本质上是进程级别的资源隔离，所以容器可以秒级启动，比VM要轻量很多
- Docker并没有发明容器，它的核心思路在于实现应用和运行环境的整体打包，统一镜像格式，**成为事实标准**
- Docker只是容器的一种，还有rkt, warden, pouch等，几乎每个大厂都会自己搞一套

1. 容器为何这么重要？又为何不再重要？



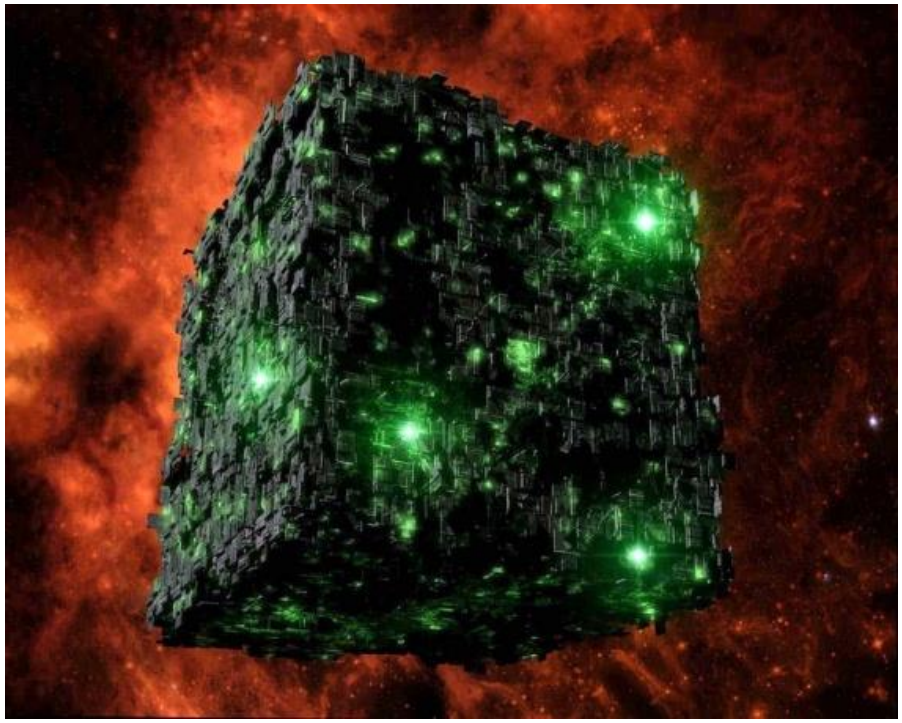




- Open Container 组织 → Cloud Native 组织
- 单机 → 多机
- 小规模，简单应用 → 大规模，复杂应用

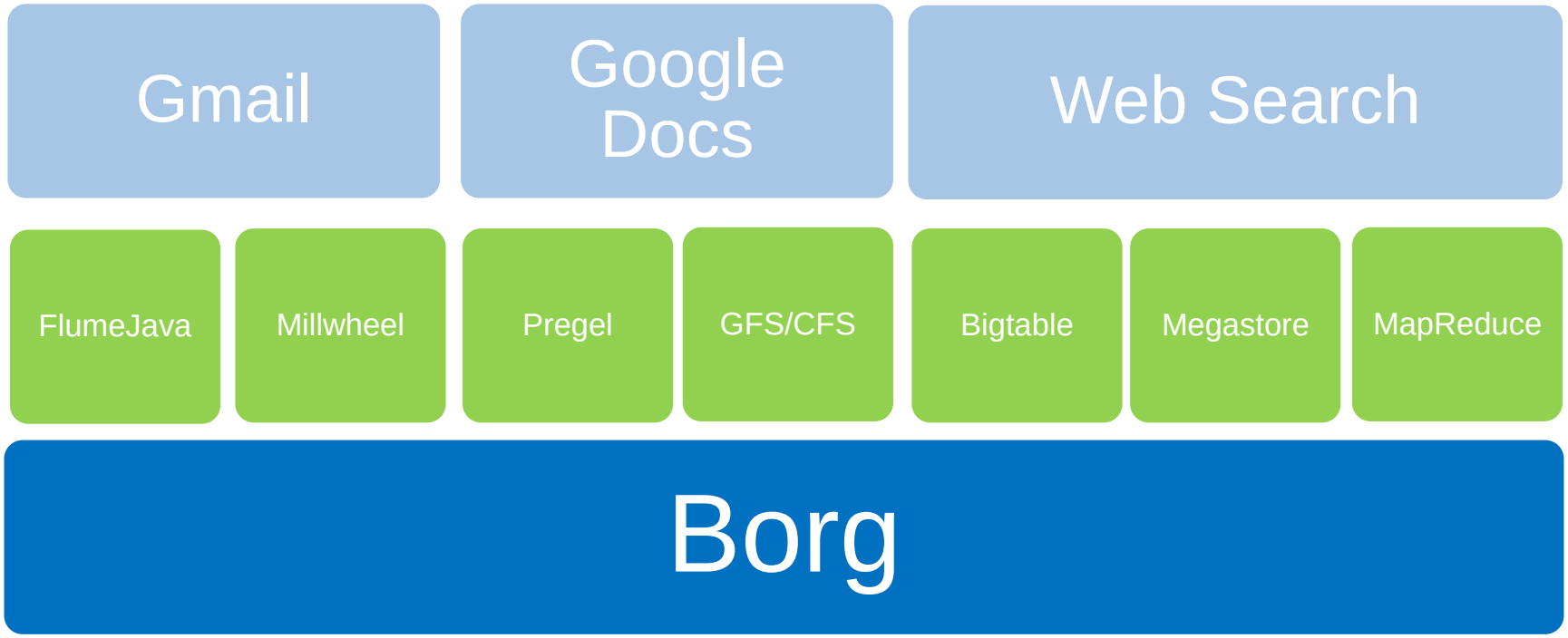
集群的价值超过单机，所以我们要从Borg看看什么是集群

2. What is Borg？



——We are the Borg. Lower your shields and surrender your ships. We will add your biological and technological distinctiveness to our own. Your culture will adapt to service us. **Resistance is futile.**

2. What is Borg in Google?



*The **cluster management** system we internally call Borg*

admits, schedules, starts, restarts, and monitors

***the full range** of applications that Google runs.*

2.Borg的架构及工作流

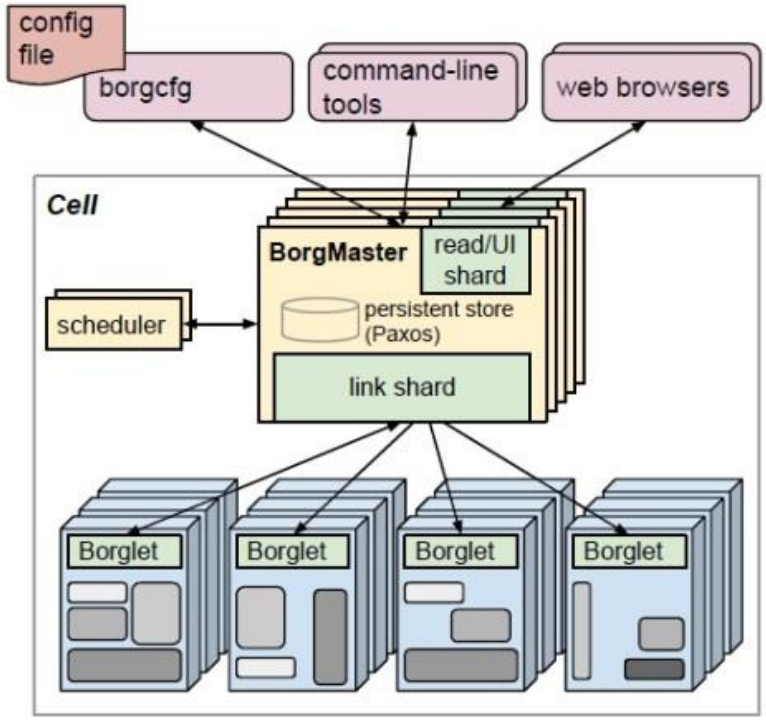
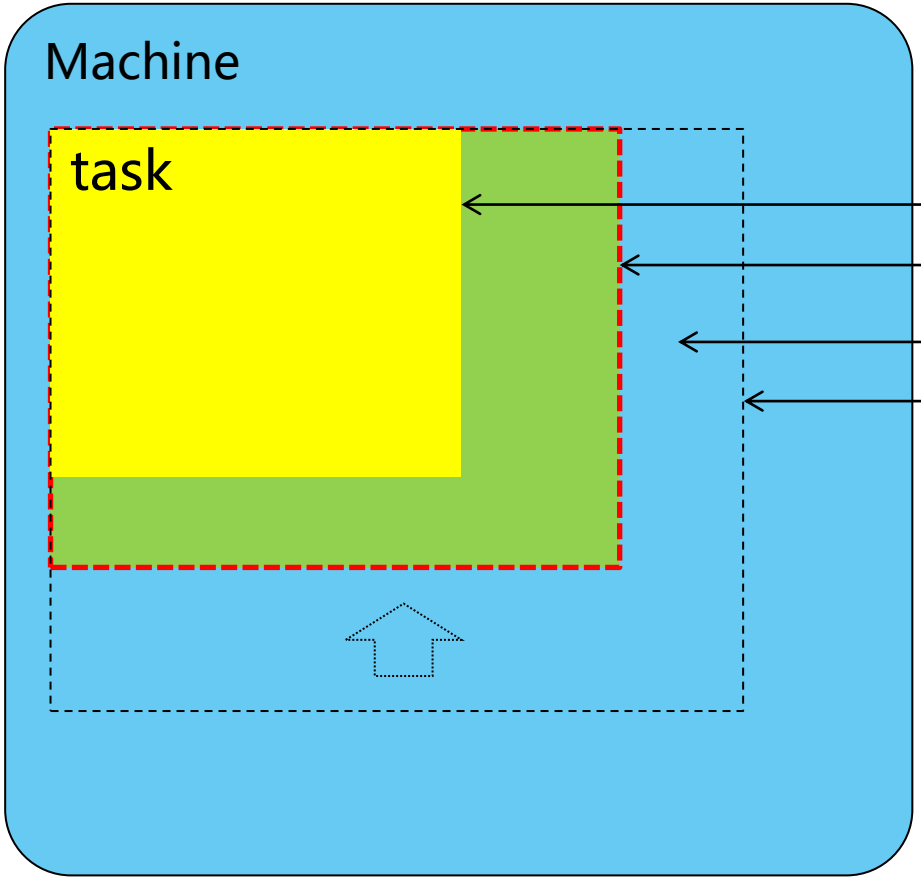


Figure 1: The high-level architecture of Borg. *Only a tiny fraction of the thousands of worker nodes are shown.*

1. 用户使用Borgcfg或者Web UI提交需要跑的应用：例如一个跑100个副本的web服务，或一个批处理任务（Task）
2. Borgmaster接受这个请求，放入队列内
3. Scheduler扫描队列，查看这个应用的资源需求，在集群中寻找匹配的机器
4. Borgmaster通知Borglet，在响应机器上启动应用

提交应用 → 应用启动 ≈ 25秒

2. 调度策略之资源计算



实际使用资源(actual)，进程消耗

保留资源(reservation)，调度计算用

回收资源(reclamation)，超卖用

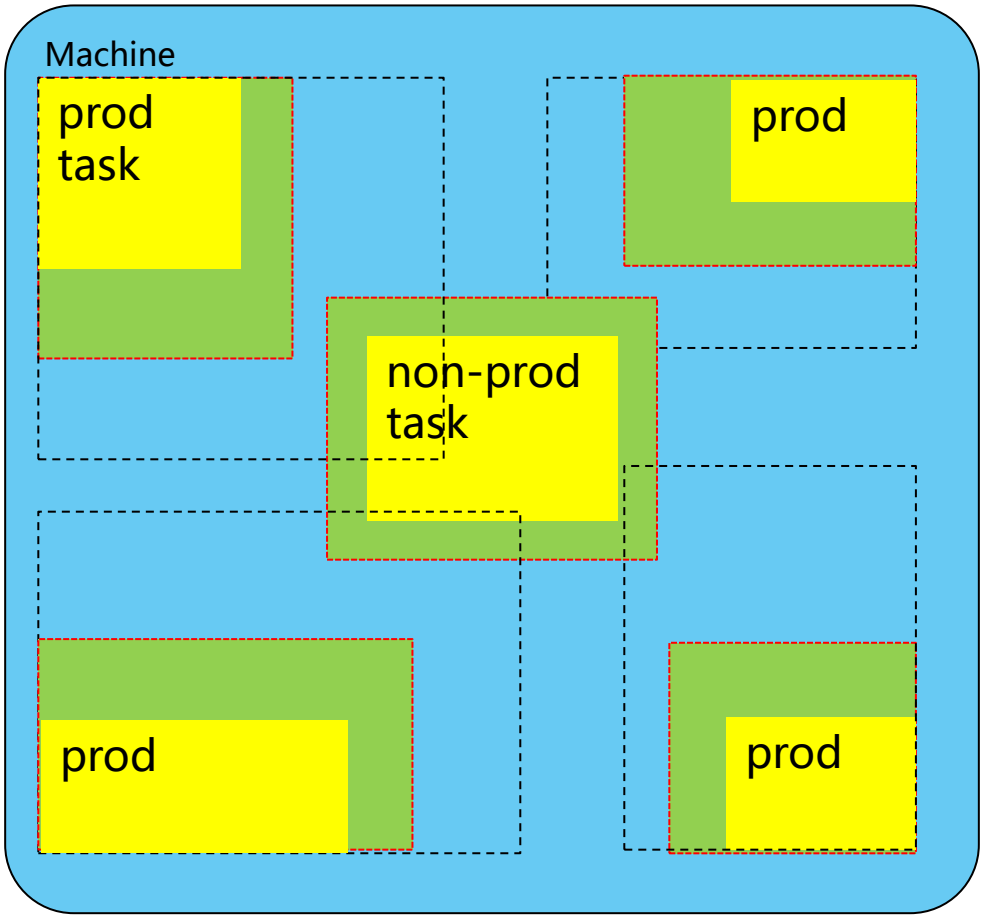
限制资源(limit)：上限，强制不超

在task启动300s后，进行资源回收工作，逐渐把保留资源设置为实际使用资源+固定余量，并每过几秒再重新计计算一次。所以图中的红线是随着时间而波动的。

2. 对负载的分类

- prod task
  - 永不停止，面向用户（Gmail，Google Search，Google Docs）
  - 单请求响应几微秒到几百毫秒
  - 短期性能波动敏感
- non-prod task
  - 批处理任务，不面向用户（Map Reduce）
  - 运行几秒到几天
  - 短期性能波动不敏感

2. 两种负载的调度策略



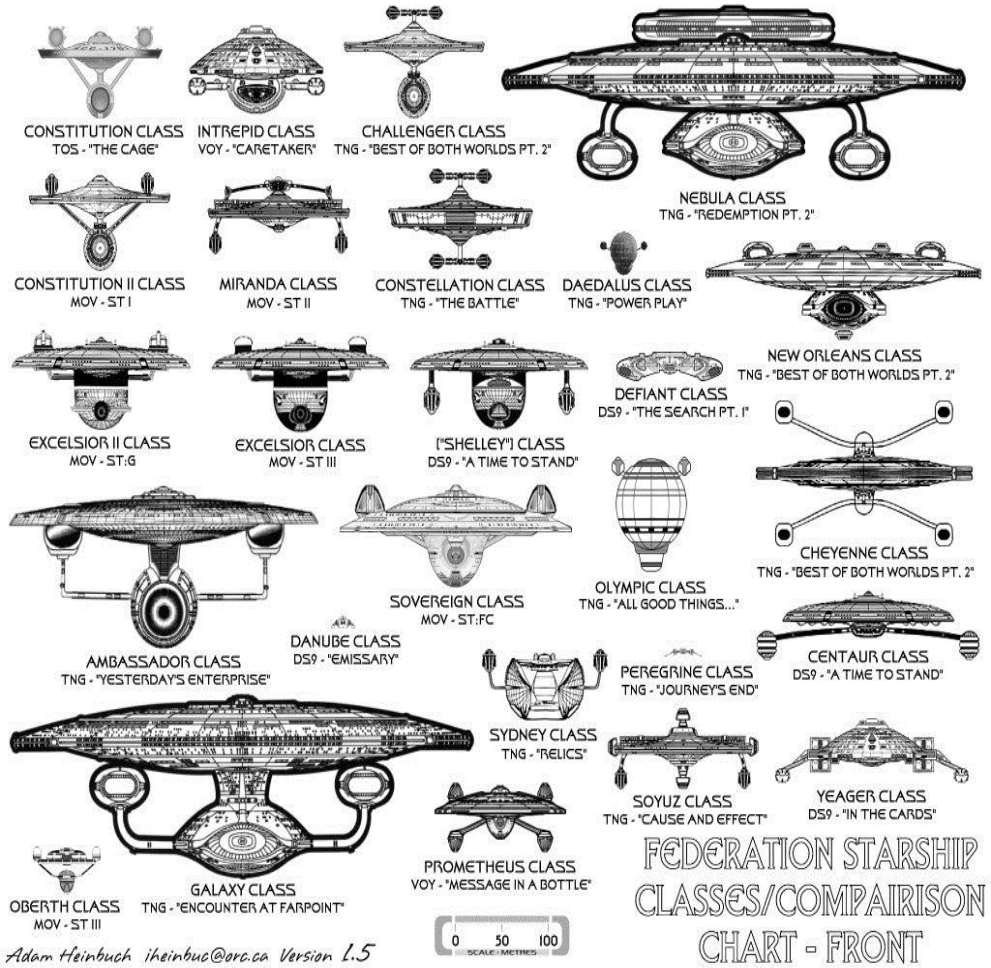
- 使用限制资源计算prod task的可用性
- prod task可以抢占non-prod task的资源，从而导致non-prod task被杀死而重调度
- prod task不能互相抢占资源而驱逐对方

20%的工作负载跑在回收资源上

2. 调度效果

- 即使Borgmaster或Borglet挂了，task继续运行
- 99.99%可用性
- 10k 机器/Cell
- 10k task/分钟
- 99% UI < 1s
- 95% borglet poll < 10s

2. 优化 = 金钱



- 如何去定义一个异构集群的效率？
- 把多个用户、prod和non-prod的task混合会提升还是降低效率？
- 资源回收和调度策略怎么样才是最佳的？
- 如何划分资源粒度？
- Cell是越大越好吗？

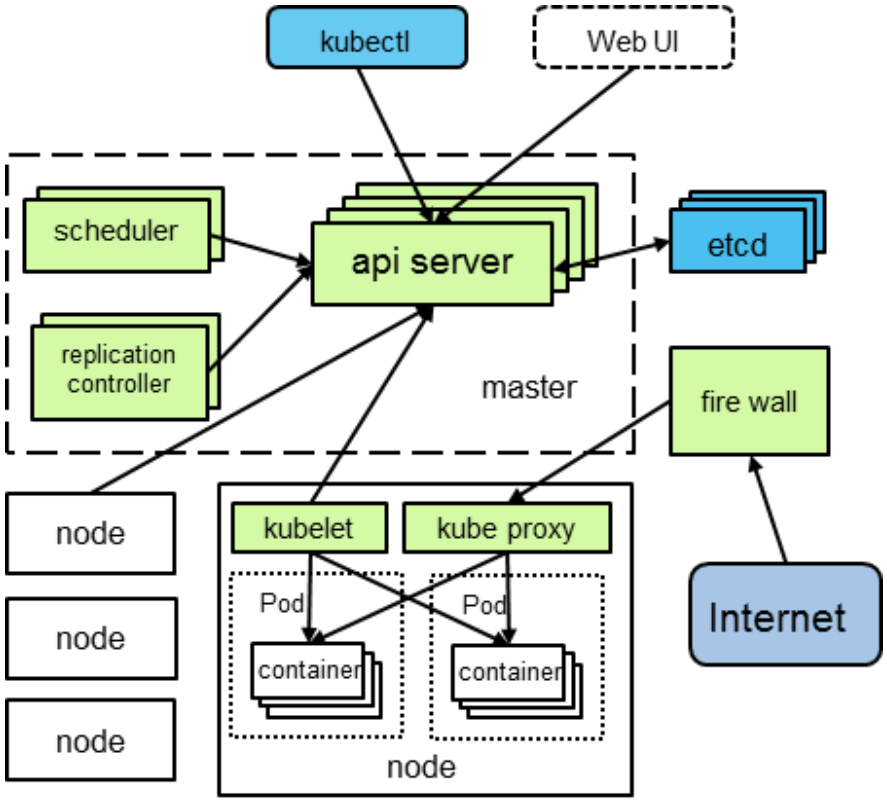
2. 一些优化指标和经验

- 压缩率，给定一个负载，部署到可以运行这个负载的最小Cell里面去
- prod和non-prod task混合运行，会降低3%-20%的CPU速度，但会节省20%-50%的机器
- task请求的资源粒度小(0.001核,byte计数内存)能提升压缩率

2. Borg所带来的……

- 隐藏资源管理和故障处理细节，使用户可以专注于应用开发
- 本身提供高可靠性和高可用性的操作，并支持应用程序做到高可靠高可用
- 在数以万计的机器上高资源利用率运行
- 详见论文《在Google使用Borg进行大规模集群的管理》—— *翻译 By 难易*

3. Kubernetes入门



1. 用户通过kubectl提交需要运行的docker container(pod)
2. api server把请求存储在etcd里面
3. scheduler扫描，分配机器
4. kubelet找到自己需要跑的container，在本机上运行

1. 用户提交ReplicaSet描述，replication controller监视集群中的容器数量并保持数量
2. 用户提交service描述文件，由kube proxy负责具体的工作流量转发

3. Kubernetes vs Borg

Kubernetes应用	Borg应用
Docker容器构建方式	静态编译，包括可执行程序和数据文件
使用容器的标准生命周期	接受SIGTERM信号，用于清理保存状态
支持挂载外部的各种持久层来维持状态(GCEPersistentDisk, AWSElasticBlockStore, NFS, iSCSI.....)	被kill之后能够在其他机器上重启，无状态
从容器中读取监控信息，从多个层面检查应用性能	一般内置http服务，用于获取健康信息
支持在Pod中包含日志处理容器	数据和日志一般都存储在分布式存储上

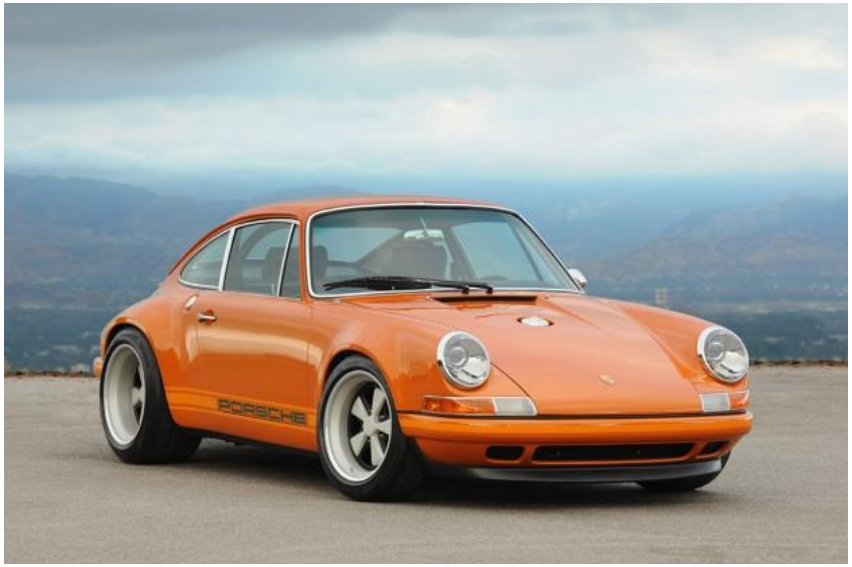
云原生应用，Cloud Native　　应用在设计期就是分布式的

3. Kubernetes vs Borg

Kubernetes	Borg
开源	闭源
2014.7开始第一次提交，发展较快	在Google的数以百万计的集群上运行超过十年
使用Docker容器，或者任何实现CRI标准的容器实现	使用lxc作为运行时
Go语言编写	C++编写
目前还没有做很多性能优化	对集群调度性能要求非常苛刻
目前单集群支持上千节点这个数量级	单集群能调度超过上万台机器

成为云计算领域的Andriod　　支撑Google自身的业务

3. Kubernetes vs Borg



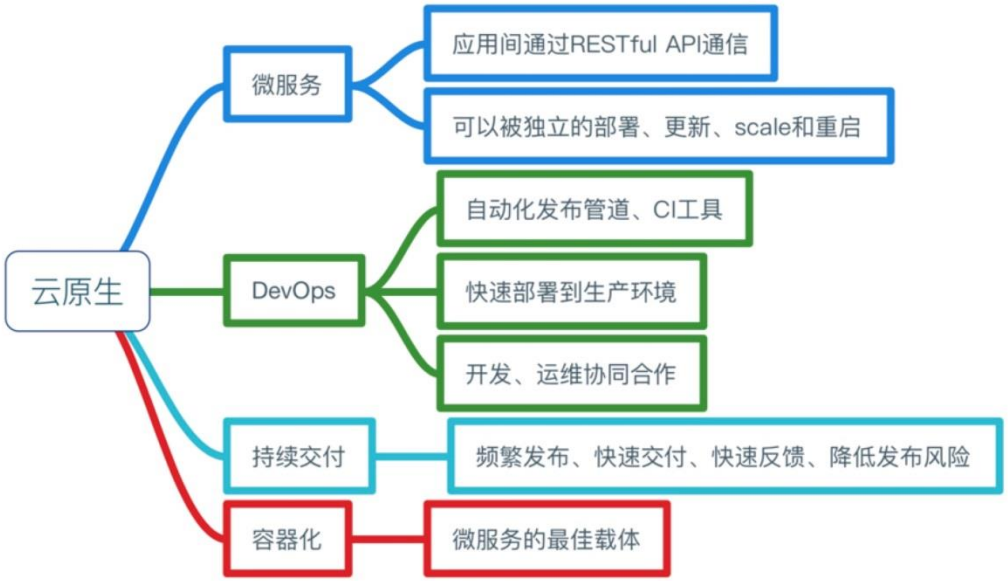
Kubernetes

Borg

4. 单体 vs 分布式

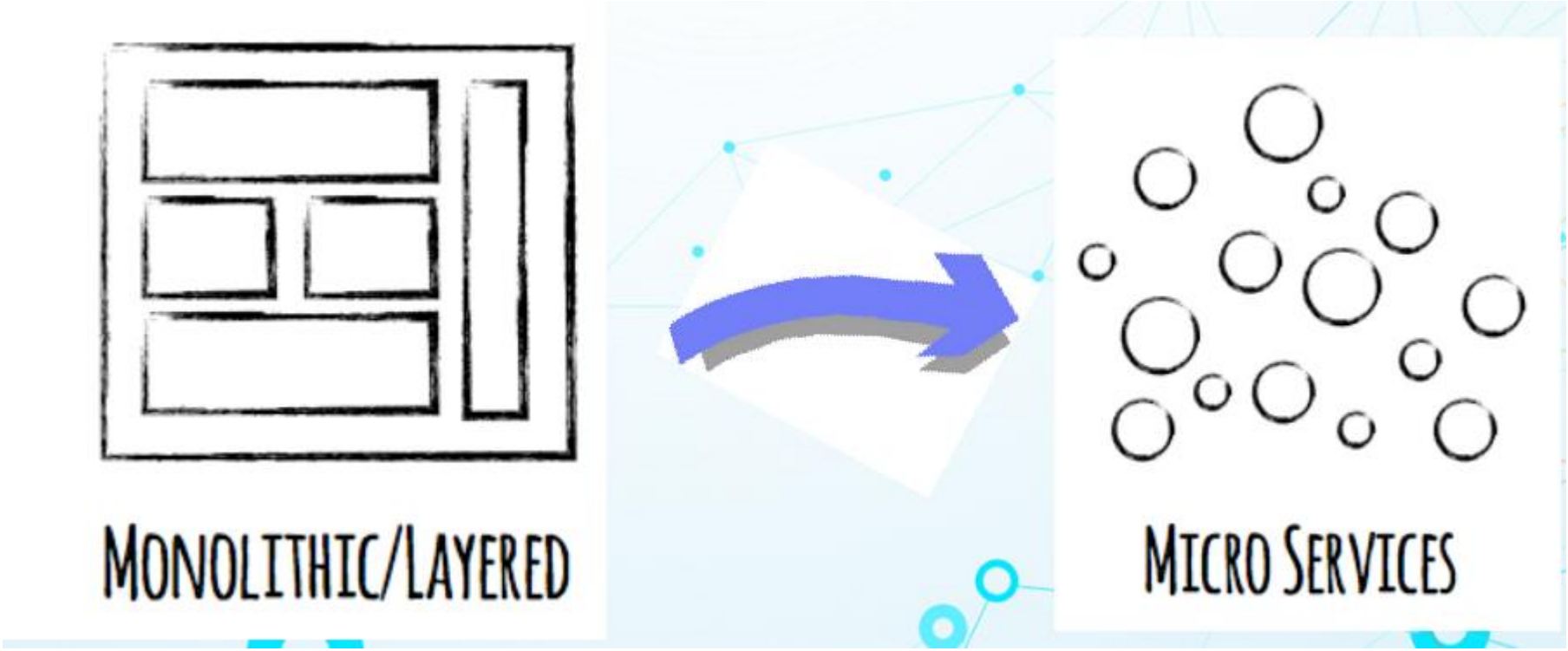
	传统单体架构	分布式服务化架构
新功能开发	需要时间	容易开发和实现
部署	不经常且容易部署	经常发布，部署复杂
隔离性	故障影响范围大	故障影响范围小
架构设计	难度小	难度级数增加
系统性能	响应时间快，吞吐量小	响应时间慢，吞吐量大
系统运维	运维简单	运维复杂
新人上手	学习曲线大（应用逻辑）	学习曲线大（架构逻辑）
技术	技术单一且封闭	技术多样且开放
测试和查错	简单	复杂
系统扩展性	扩展性很差	扩展性很好
系统管理	重点在于开发成本	重点在于服务治理和调度

4.云原生应用的理念



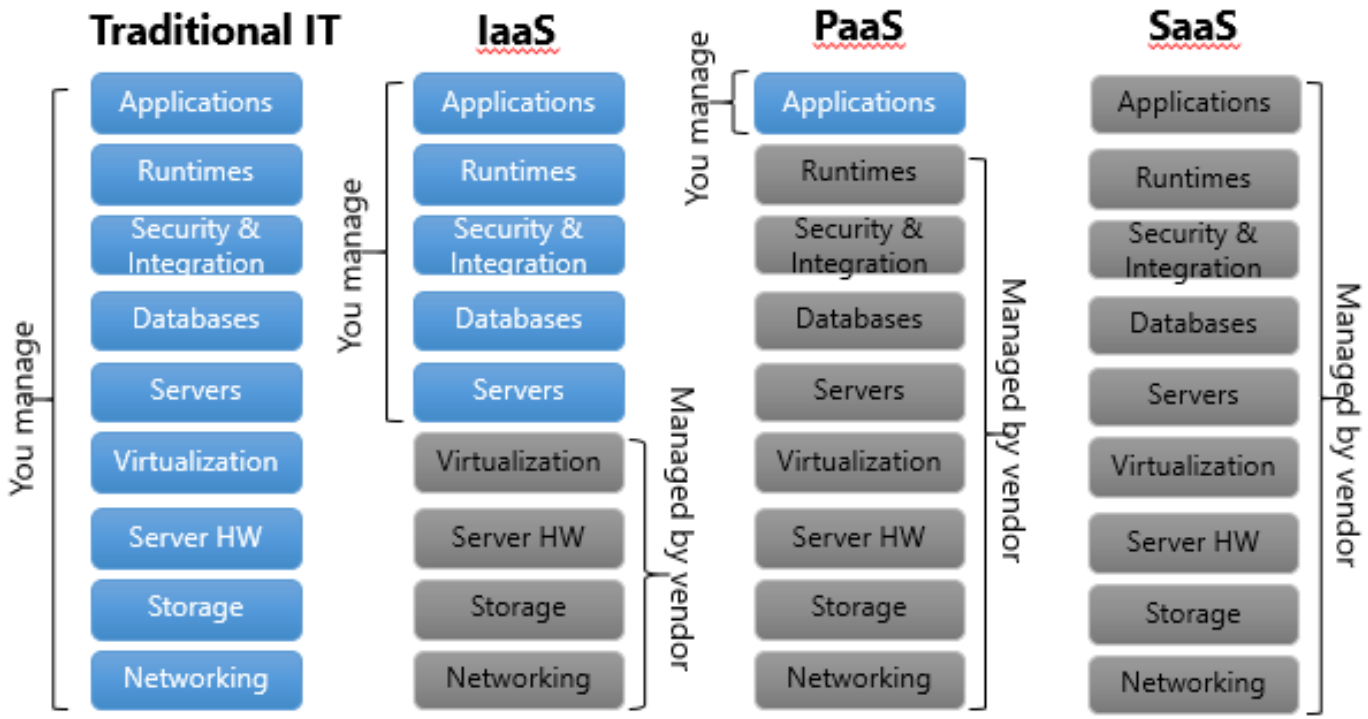
- Containerized. Each part (applications, processes, etc) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.
- Dynamically orchestrated. Containers are actively scheduled and managed to optimize resource utilization.
- Microservices oriented. Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications.

4.微服务



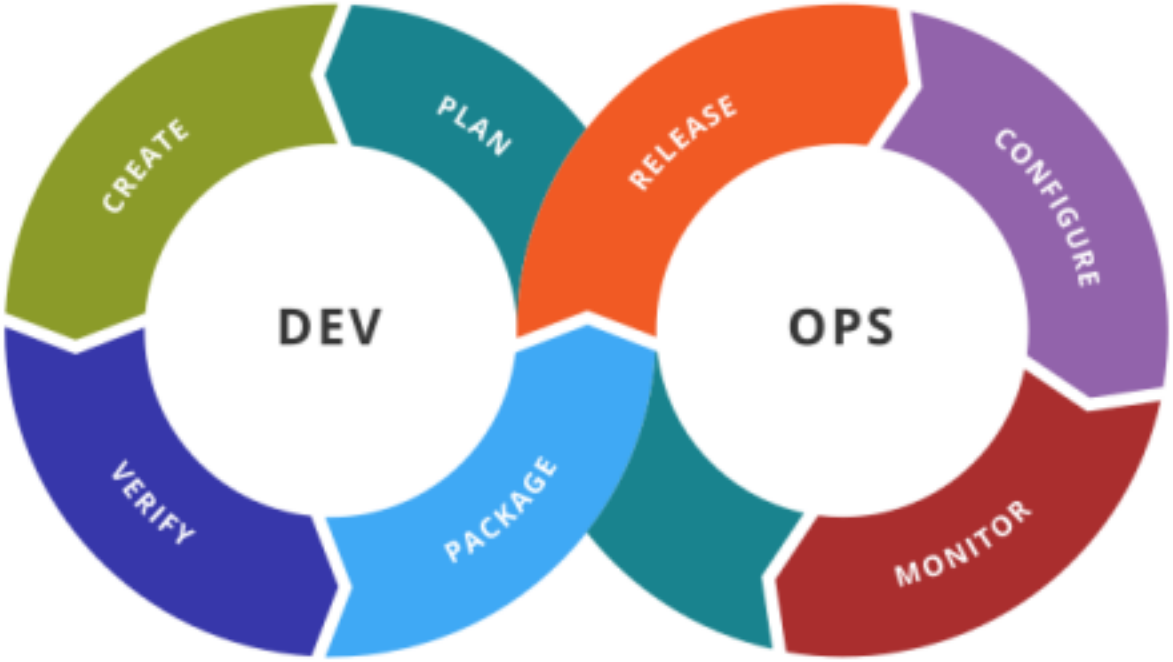
SOA概念的继承者，在亚马逊等电商公司得到成功的实践，详见《STEVEY对AMAZON和GOOGLE平台的吐槽》

4.PaaS



和IaaS, SaaS相对，从软件栈的角度描述云计算提供者和消费者的关系

4. DevOps



从IT的整个流程，面对运维复杂度超过开发的情况所作出的努力

4. Serverless

目标在于一步打通开发者和生产可用云服务，实现了以函数为单元运行在分布式集群上

5. 简单总结

- 理念不同，殊归同途
- 在云时代降低软件开发运行维护的成本
- Kubernetes作为云时代多机的开放式kernel，其机制正在慢慢成为标准
- 无论是想使用，还是想颠覆Kubernetes的大规模应用流行，都可以从Kubernetes所做的工作着手

联系我们

小象学院：互联网新技术在线教育领航者

– 微信公众号：**小象学院**