携程容器云弹性能力构建之路

乐鸿辉

高级研发经理

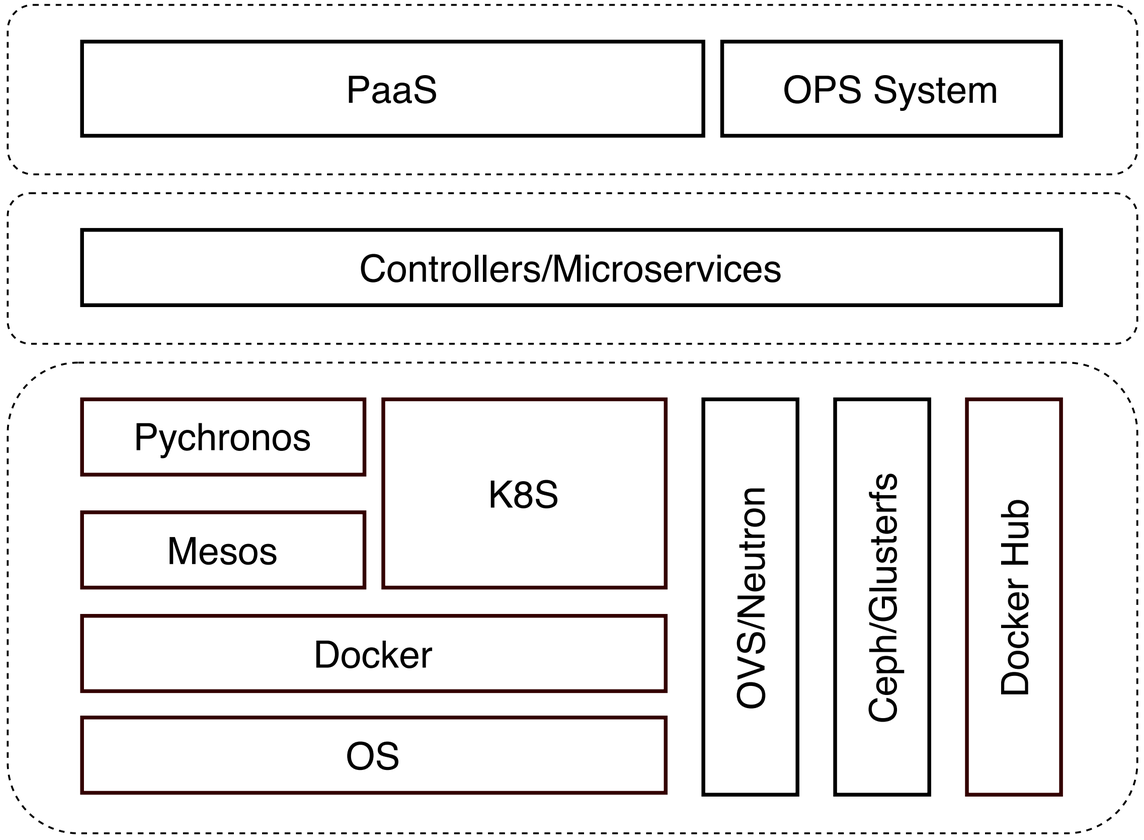
目录

- 历史与架构
- Mesos迁移K8S
- 资源池与弹性
- 踩坑经验

历史


Mesos+Job
极致发布体验
JAVA在线应用容器化
弹性能力
Mesos迁移K8S
2017
Container Trip
2018
Elastic Trip
2019

架构



用户产品层：PaaS
- CI/CD、Build
- 应用生命周期管理
- IaaS资源管理
- 统一VM、Docker

中间层：微服务及Controllers
- 提供编排服务抽象(App、Redis、MySQL...)
- 支持多Region/Available Zone
- 透明支持异构K8S、Mesos Cluster

基础层：云基础设施
- Mesos+Pychronos、K8S
- OVS，Docker/CNI插件
- 存储自定义FlexVolume/Agent，提供Local/Network Storage
- 对接监控、SLB外部系统

选型

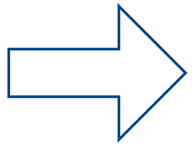
Mesos or K8S ?

愿景

只支持在线应用、JOB
只提供容器实例，无编排
发布系统需要负责网络、实例的状态管理
用户关注实例、IP

服务化更多基础设施(Redis、MySQL...)
提供更高层次的抽象
屏蔽IaaS层的资源管理
用户关注服务、策略

资源提供者



服务提供者

选型

	坚持Mesos	迁移K8S
优势	简单 熟悉 稳定	已有插件与经验大部分可迁移 设计良好的声明式编程范式 丰富的服务编排抽象 技术栈统一，Go语言 生态 未来技术路线清晰(Service Mesh、Serverless)
劣势	生产版本老旧，自研调度器Pychronos 耦合Mesos二进制接口 升级成本不亚于迁移 技术栈繁杂，难以定制优化 编排能力和范式弱 生态趋弱	复杂 团队需要时间熟悉 短期内稳定性有风险 短期有存量迁移成本

短期来看，坚持Mesos成本低；长远来看，迁移K8S更有价值

选型

Deployment or Statefulset ?

选型

现实情况

- 全部固定IP
- 部分应用有存储需求
- 存储需求模糊，且可能发生变化
- 兼容同一套原有发布习惯

Statefulset与Operator定位取舍

- 在线应用使用Statefulset，固定标识、支持存储
- 对编排、Fail Over等有特殊需求的，Statefulset无法完全满足，应该使用CRD、Operator

选型

Deployment很美好

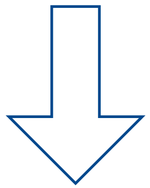
但是我们选了Statefulset

架构需求

存量Mesos上应用迁移

单个Cluster容量不足时扩容

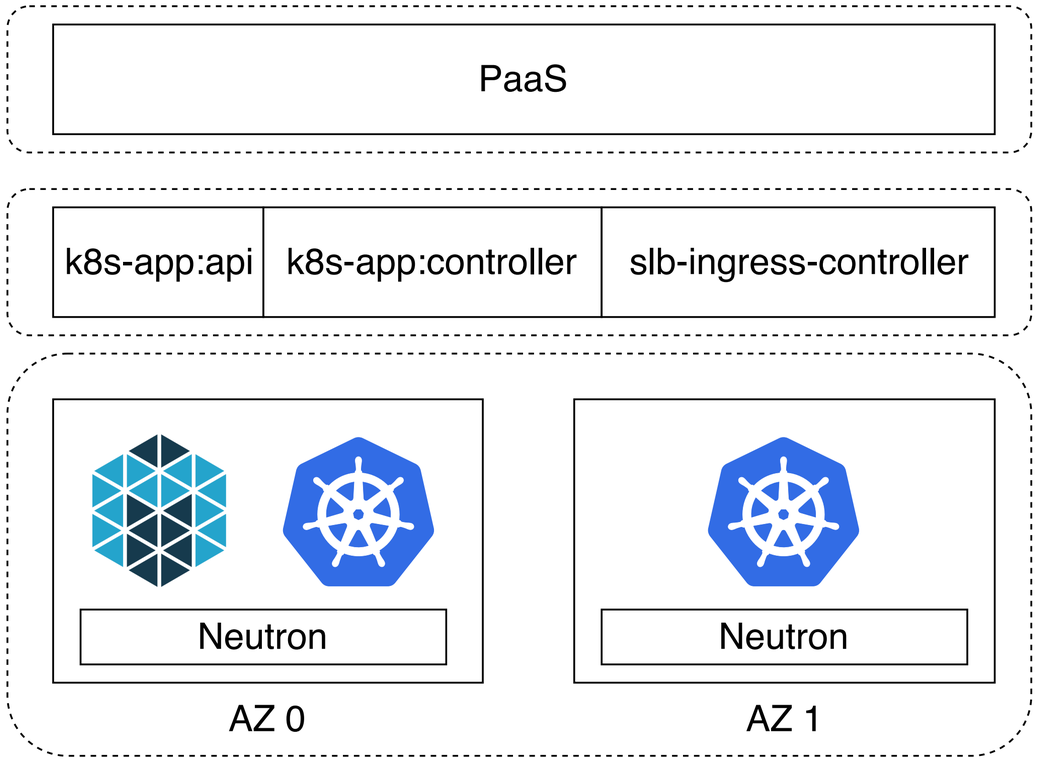
K8S可用性短期有风险

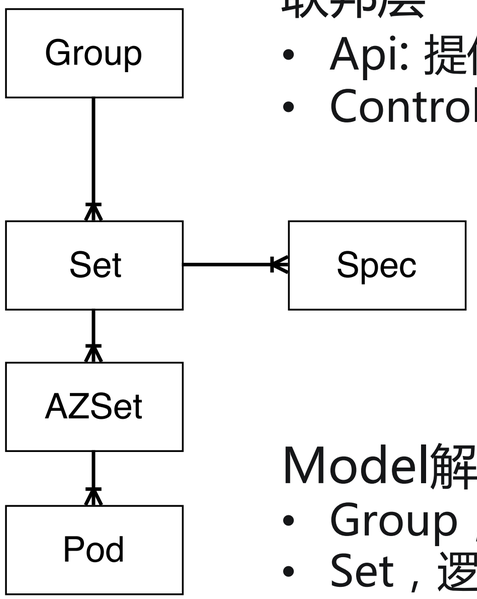


同时支持Mesos、K8S

单个应用透明跨多Cluster

K8S-App架构与模型





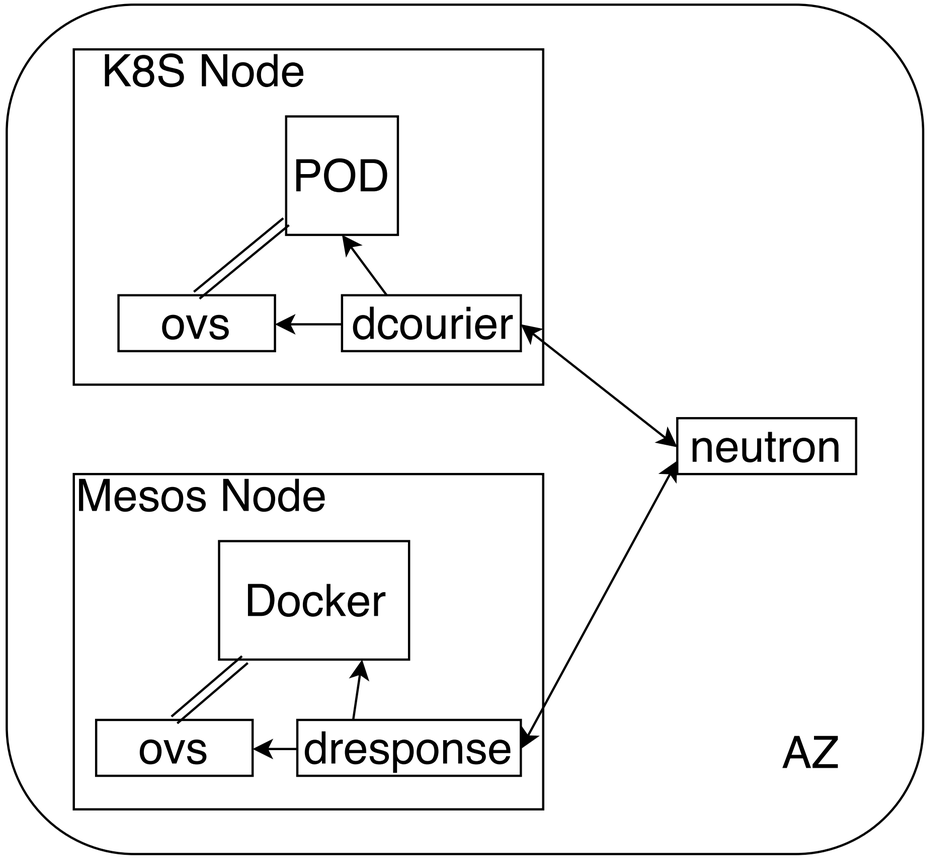
联邦层

- Api: 提供Model Rest Api服务
- Controller: 状态机，管理IP、存储、Mesos、K8S状态

Model解释

- Group，逻辑类似于Deployment
- Set，逻辑类似于跨Cluster的RS/Statefulset
- AZSet，K8S上使用Statefulset，Mesos上实现简版Statefulset
- Spec，版本信息

K8S-App网络



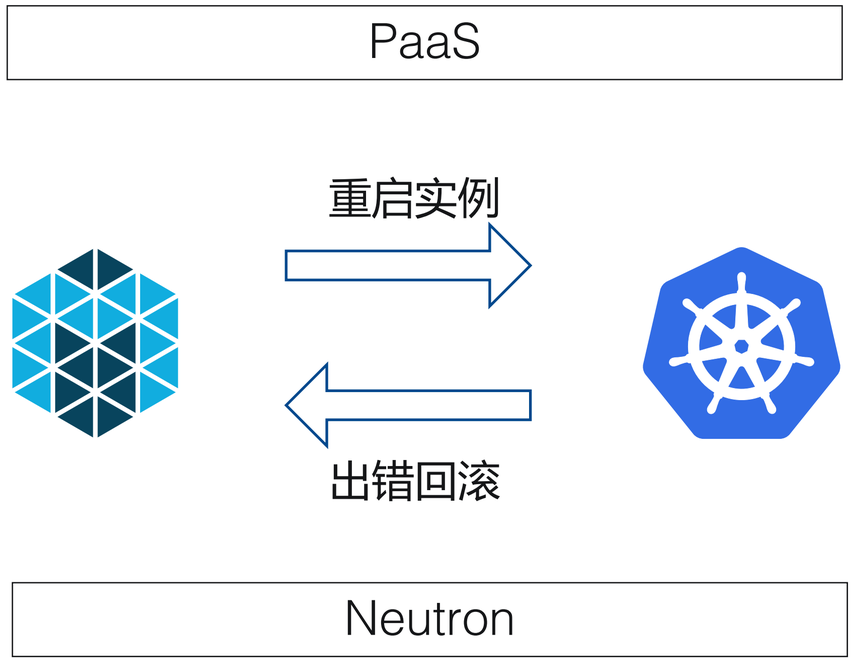
PortId	Name	DeviceOwner	IP
xxxxx	podname:ns:cluster-domain	dcourier/k8s-app	xx.xx.xx.xx

dcourier、dresponse
- K8S及Mesos下OVS网络插件

POD IP
- neutron管理的VLan IP
- Mesos、K8S、VM IP互通

Service & Ingress
- 没有使用K8S的Service IP及Ingress Controller
- 自定义slb-ingress-controller监听 Service&Endpoint 对接公司原有负载均衡

存量迁移



克服挑战
- 无需用户参与，IP、元信息不变
- 灰度、资源池
- 跨系统操作事务性
- 自动化
- 用户沟通

要点
- 同AZ的Mesos到K8S迁移按物理机维度迁移
- 拆解迁移过程为多个checkpoint，支持回滚
- 使用stackstorm自动化pipeline
- 宣讲与邮件通知

正在迁移生产最后一个Mesos集群

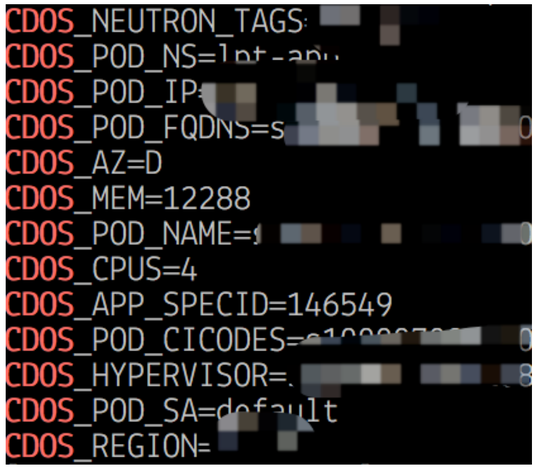
功能扩展

调度
- 带阈值的资源分散与资源集中策略
- 资源维度平衡策略
- 粘性(sticky)调度

存储插件
- 带project quota的系列FlexVolume，规范化宿主机mount point，路径可推导，方便宿主机agent操作volume
- 增加与Statefulset生命周期绑定的FlexVolume

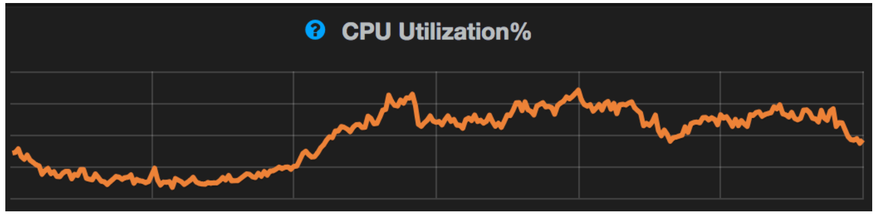
网络
- 自定义CNI插件，对接OVS及AWS，实现固定IP
- 可根据自定义Network Policy分配IP
- CNI插件通过Pod Annotation，配置OVS的网络限速

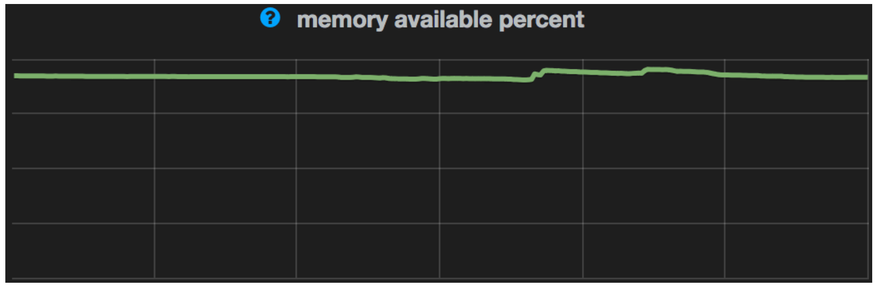
外部Agent自发现
- 制定Pod Env规范，修改cadvisor导出Env，实现宿主机外部Agent对POD的自发现



资源池困境

	CPU	MEM
分配率	>97%	>97%
利用率	~10%	~60%

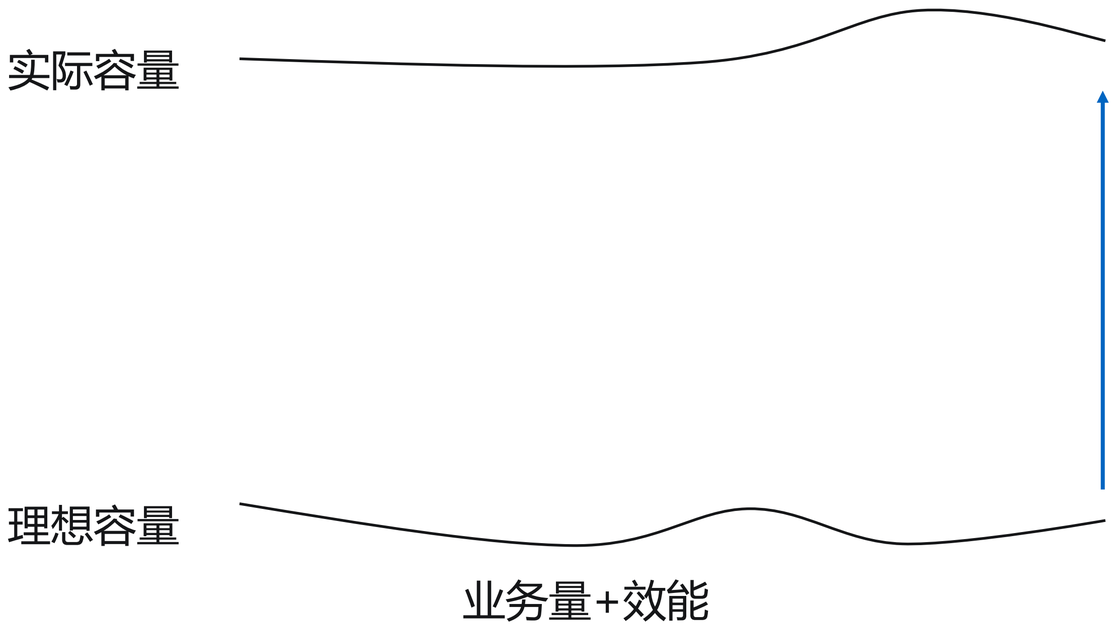




CPU利用率低
- CPU利用率低
- CPU利用率与MEM(RSS)利用率失衡
- 不利于超分混部

MEM利用率周期性差
- CPU有较好周期性
- MEM周期性很差，JAVA堆内掩盖了实际利用率(xmx=xms)
- 不利于低峰期超分混部

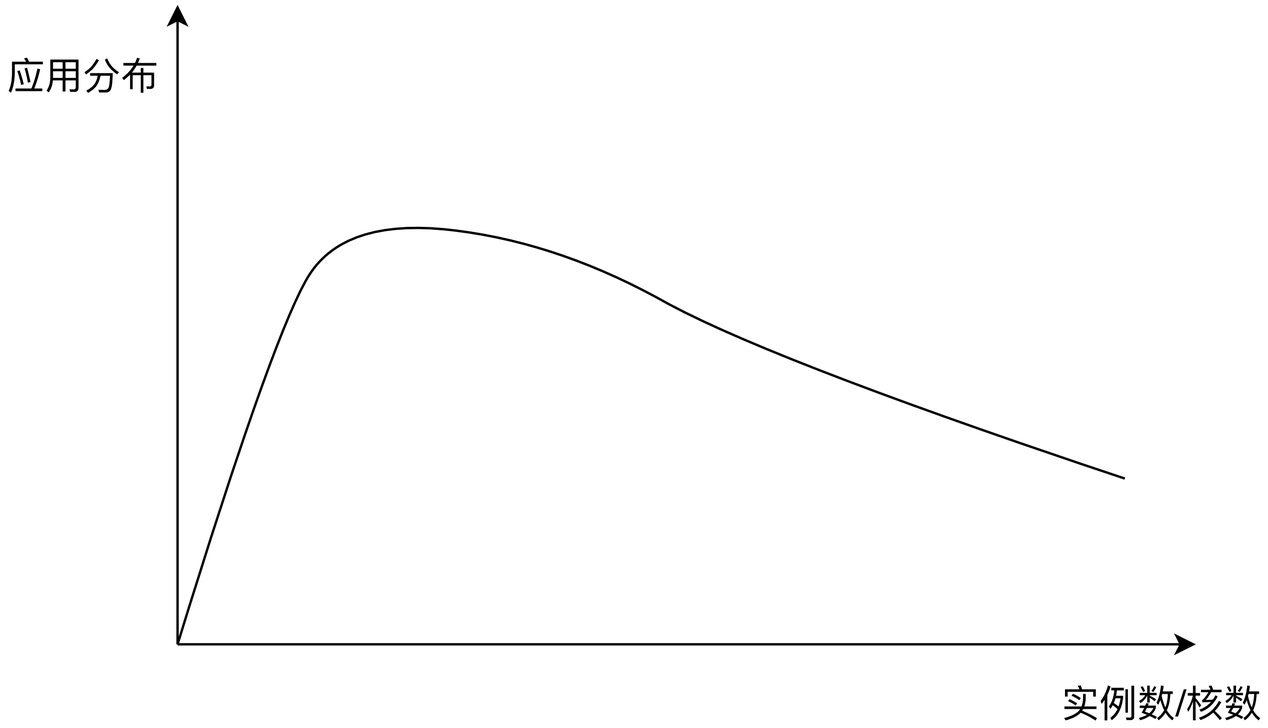
理解容量



浪费

- 资源池割裂
- 应用配置过大
- 应用实例过多
- 分配碎片
- 资源配比不均
- …

分析-长尾与热点

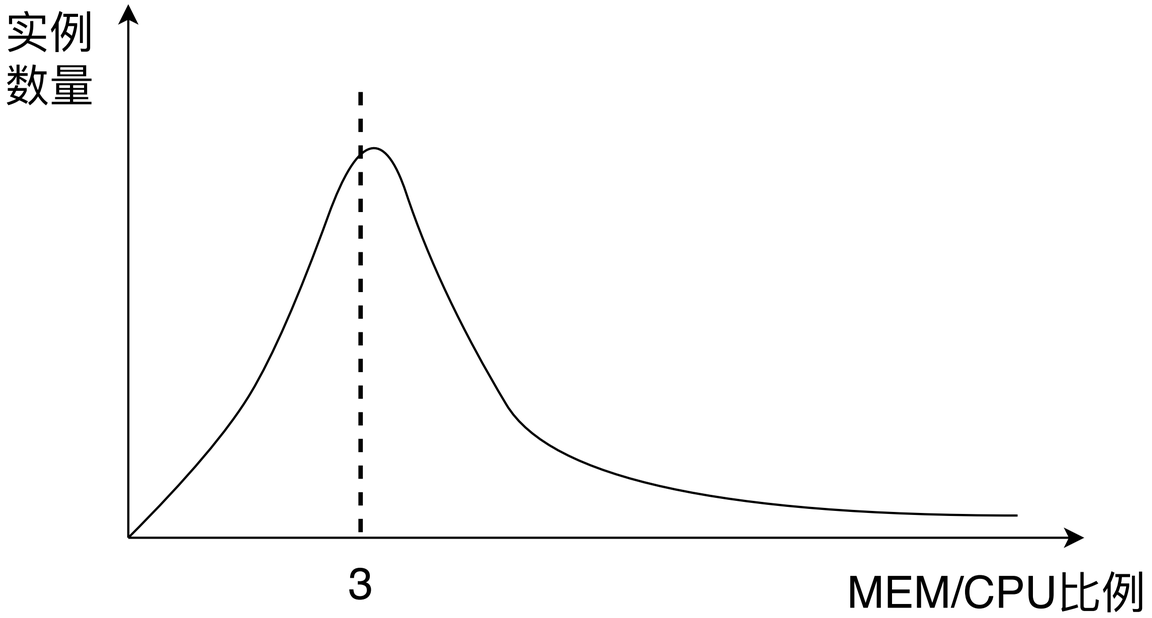


长尾应用
- 实例数少
- 应用数量超过一半
- 利用率极低

热点应用
- 实例数多
- 应用数量不多
- 利用率较低

微服务及单实例单应用趋势
长尾会越来越多

分析-资源配比



大量在线应用实例
MEM(G)/CPU(Core)
大于3

JVM堆内比较空闲，内存给多了

分析-资源配比

越来越贵了
都快买不起内存了

策略

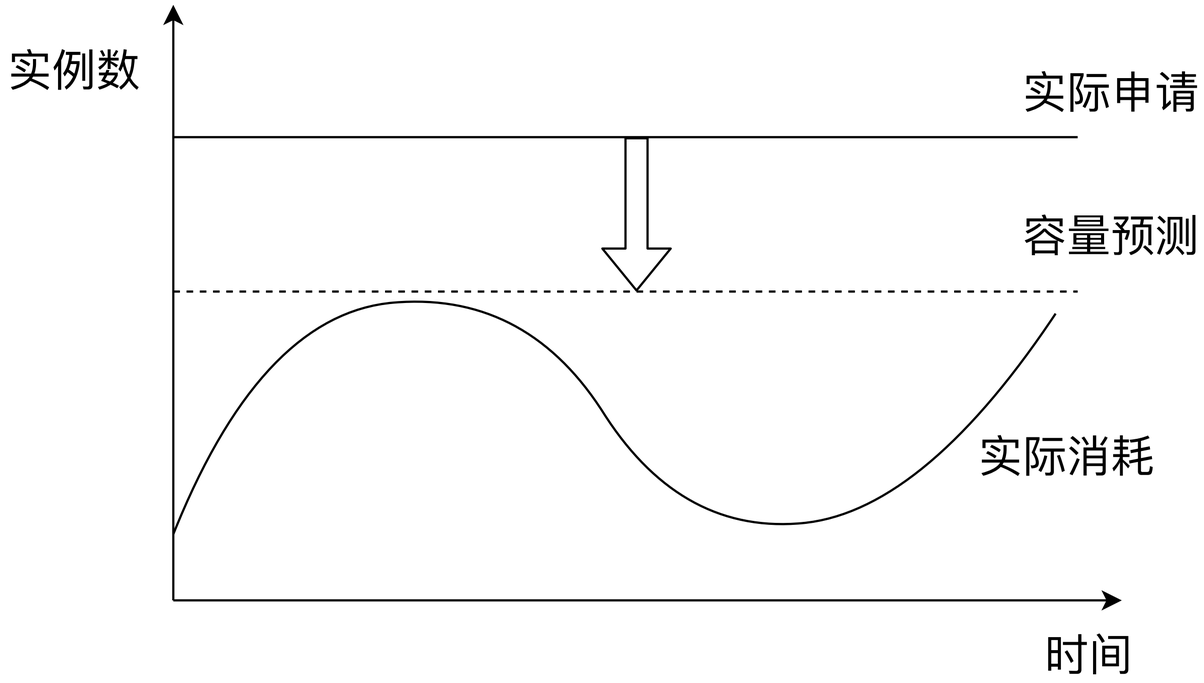
热点应用
- 横向缩容

实际利用率低
- 超分混部

长尾应用
- 纵向缩容
- 调整xms
- 超分

内存浪费
- 压缩MEM:CPU到2：1

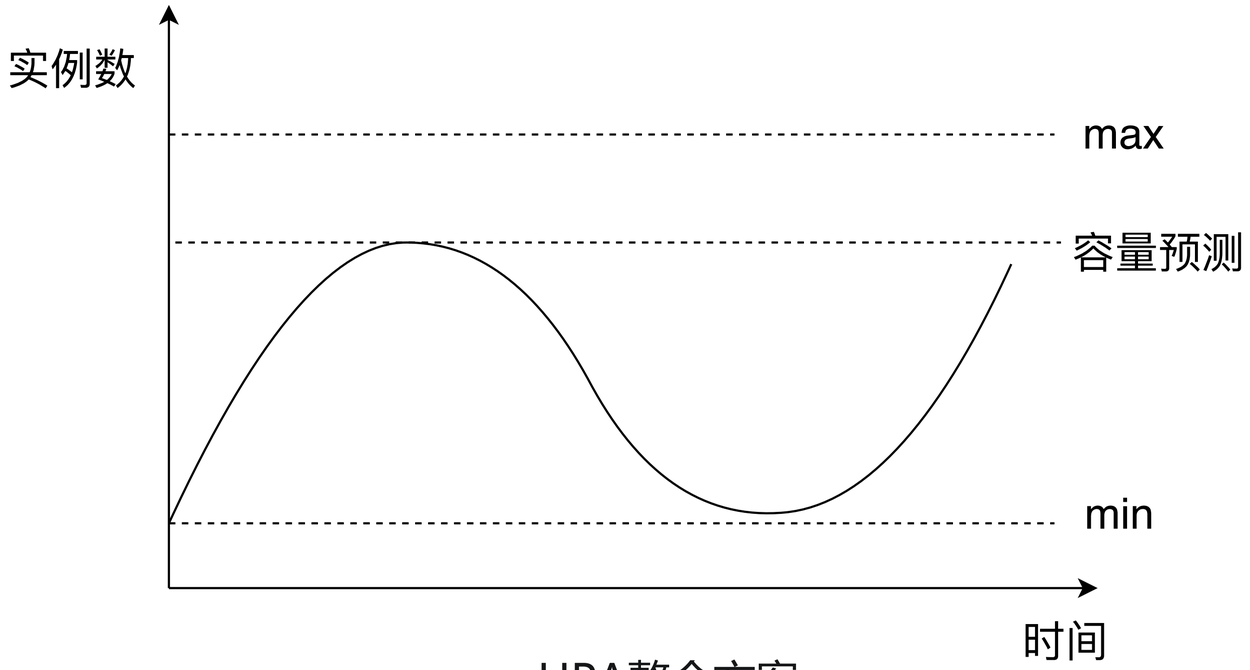
热点-横向缩容



容量规划
- T+n 计算应用合理容量
- 提前通知
- 横向缩容、异常检测、回退或继续

例外
- 白名单
- 跳过异常状态应用

热点-HPA整合



HPA整合方案

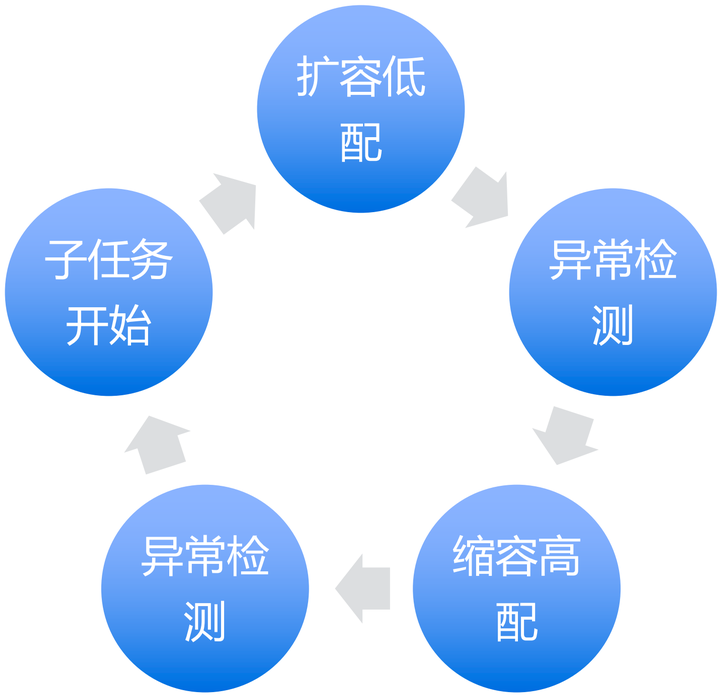
价值
- 公有云OnDemand模式并不省钱
- 低峰期配合混部

min与max确定
- min一般取2
- max=倍数 * 容量预测

例外
- 例外名单
- 手动扩容
- 计划规则

风险
- 一天内周期性扩缩，依赖扩容链路的可用性，min和预测容量间扩容失败有较高服务风险
- App透明多Cluster部署，HPA支持多Cluster

长尾-纵向缩容



目的
- 降配
- MEM:CPU 比例逼近到 2:1

注意
- 提前通知
- 根据CPU、MEM、JVM heap数据计算目标配置
- 分多个Step降配
- 例外名单(Cache类型，特殊用户)
- 先易后难

XMS调优

调整前

$$xmx = \alpha * MEM$$

$$xms = xmx$$

后果

- 堆内闲时JVM仍然hold住内存不释放
- CPU、MEM实际利用不均衡

调整后

$$xmx = \alpha * MEM$$

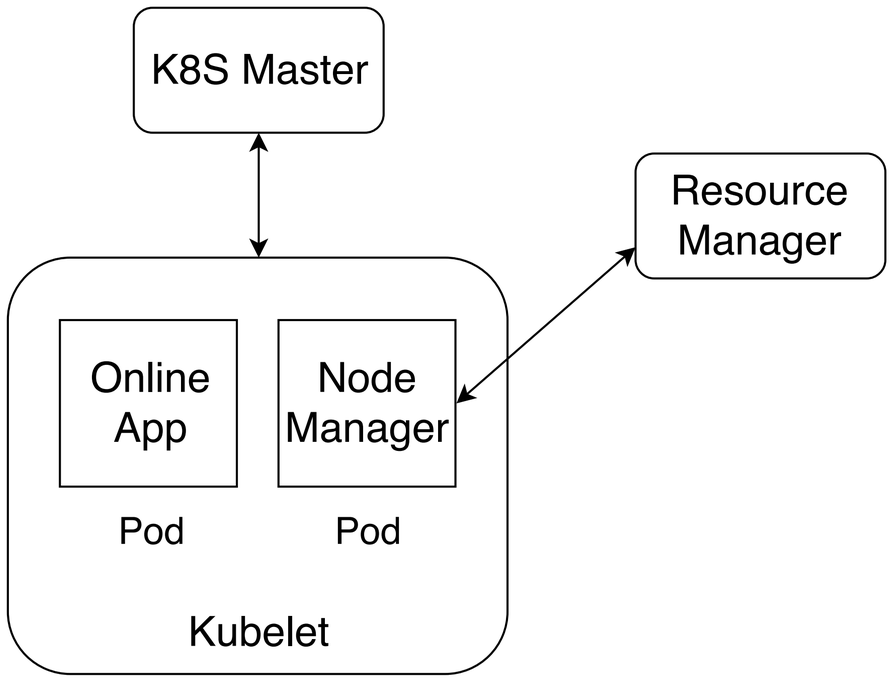
$$xms = \beta * xmx$$

方法

- 配合容器内启动脚本，通过环境变量动态注入β及*MinHeapFreeRatio*、*MaxHeapFreeRatio*
- 测试环境全部调整，内存超分比从1.3提升到2
- 考虑到潜在性能影响，生产环境只调整长尾应用

JDK 1.7无效

超分混部



网络
- 提前申请IP池和域名，注册DNS反解
- 10G网络
- 定制的CNI插件，读取Pod Annotation配置OVS的inbound与outbound limit

存储
- 扩展的cEmptydir，限制磁盘大小，对接监控

超分与驱逐
- limit=(8C,24G)，requests=(10m, 10M)
- oomScore → 1000
- 物主机内存驱逐配置(soft=24G, hard=8G)

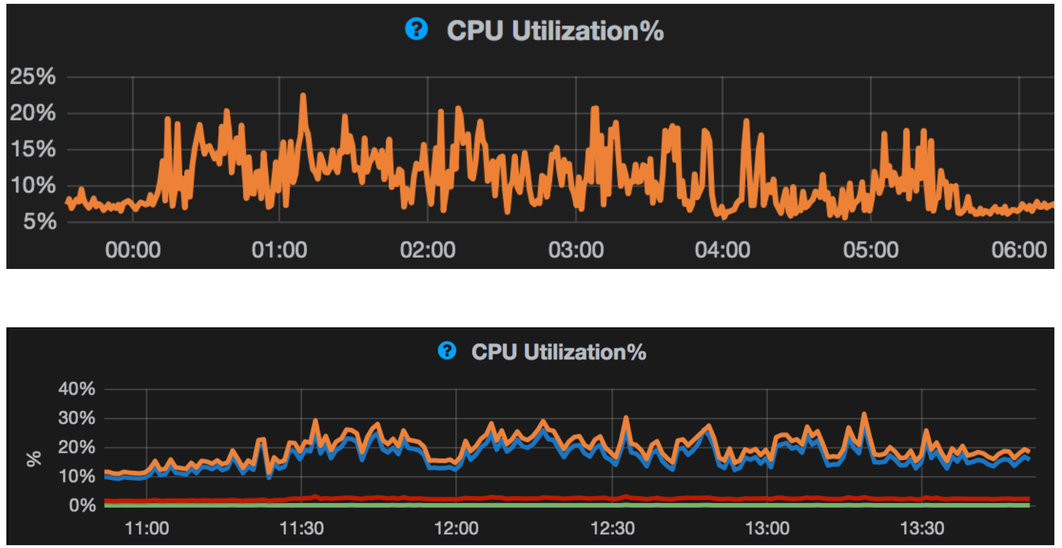
调度策略
- Preferred Pod Antiaffinity
- 每个物理Node上根据实际利用率控制Pod数量

超分混部

投放策略
- JOB，小任务到大任务(网络IO与计算时长)
- 混部容量，从低峰期定时扩缩容，到全天根据容量曲线(CPU、MEM、Network)计算混部容量，动态扩缩容

效果与瓶颈
- CPU利用率(PerCore)绝对值提升8个点
- 受限于离线在线集群架构，跨机房网络瓶颈，无法充分挖掘闲时资源

未来规划
- 在线离线集群部署架构改造，混部放量
- 更精细的数据收集与分析
- 在线到离线混部

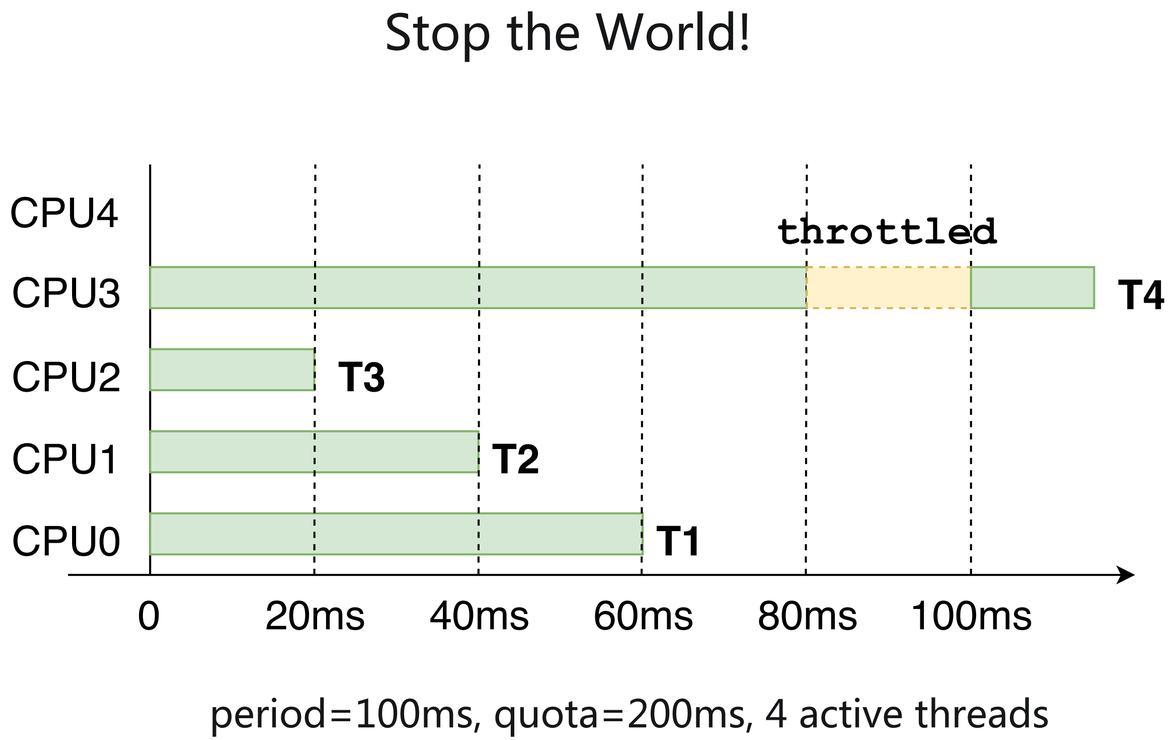


低峰期到全天混部

坑-cpu throttle

危害
- JVM启动时，Throttled严重，易导致请求超时
- Latency敏感型应用，可能引入长尾延迟抖动
- 依赖线程的程序(Java)较易受到影响

原因
- cpu-quota缺陷，限制了quota，却没有限制并行度
- cpu-bound线程太多，超过了分配的逻辑核数



坑-cpu throttle

调整JVM参数，适配分配的逻辑核数
- CICompilerCount
- ConcGCThreads
- ParallelGCThreads
- java.util.concurrent.ForkJoinPool.common.parallelism

Runtime.getRuntime().availableProcessors()
- libsysconfcpus
- 只mock SC_NPROCESSORS_ONLN不够，还需要mock SC_NPROCESSORS_CONF
- cpuset时(本来也无需mock)，需要关闭SC_NPROCESSORS_CONF，高版本JDK(8u144)，取核数代码发生变化，大概率获取核数为0
- 最完美办法，升级到最新JDK，但不太现实

放宽cpu-quota到逻辑核的2倍(cpu密集型应用除外)
应用和框架配合改造，将较重初始化动作放到流量接入前的预热阶段。cpu-bound独立线程池。

还是不行？上cpuset白名单

坑-K8S稳定性

https://github.com/kubernetes/kubernetes/issues/55159
ControllerRevision重复创建

内测环境多次发现该BUG，该BUG触发时，ETCD的DBSIZE会直线增长，直到打爆，破坏力十足！

关键，我们很长时间都没有找到root cause！

多次演练
监控报警
修复预案和脚本
故障修复能在10分钟左右完成

ETCD DB Size监控

坑-K8S稳定性

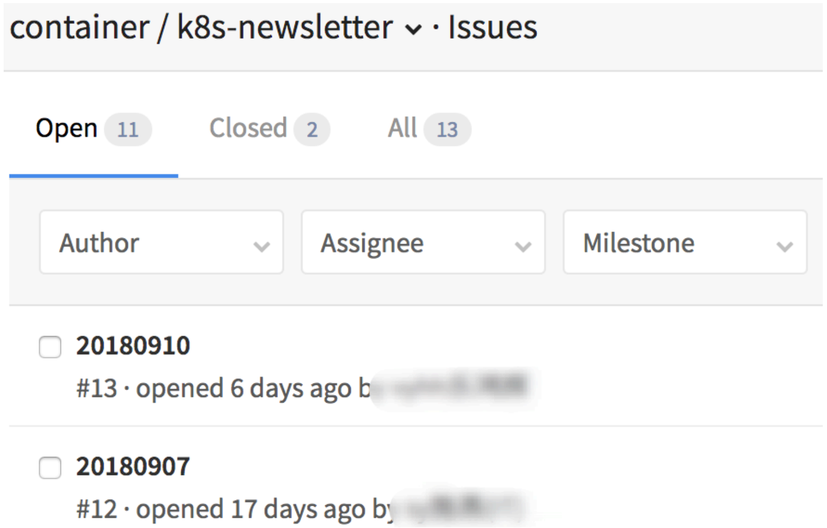
如何尽早发现与控制风险？

坑-K8S稳定性

测试环境
- 1套破坏性测试环境，模拟网络抖动故障、分区、磁盘满等场景，可用脚本快速重建集群
- 2套内测环境，用烂机器，跑内部自有应用及各种测试性的feature

持续社区跟踪
- k8s newsletter机制，收集k8s issue及社区文章动向，每周周会集体review
- 跟踪高版本bug fix与feature，merge回内部版本

防御性架构
- 多Region多AZ，一个AZ一个k8s cluster
- Region内应用透明部署到多个AZ上



THANKS!

更多技术细节：携程技术中心@微信公号