Alibaba Group
阿里巴巴集团

# 阿里巴巴云原生技术与实践

## – KubeCon 2020 经典演讲集锦

Alibaba Group
阿里巴巴集团

阿里云开发者钉钉交流群

阿里云开发者社区

阿里巴巴云原生公众号

KubeCon 2020 视频回看

# CONTENT





# 前言

十年云原生实践经验，带给阿里巴巴的不止有全球最大规模成功落地的云原生样板间，还有对云原生技术进化趋势的精准把控。实际上，智能、互联、信任三位一体的创新基础设施，将成为数字时代新基建的“底色”，阿里巴巴将持续以自身成功经验回馈广大用户，始终以前瞻性视角引领云原生进化，帮助企业找到数字化转型“最短路径”。

阿里巴巴云原生技术与实践

-KubeCon 2020 经典演讲集锦

开篇

# 云原生-数字经济技术创新基石

阿里云容器服务负责人 易立

云原生技术不但可以最大化云的弹性，帮助企业实现降本提效，而且还有着更多创新的想象空间。云原生将和 AI、边缘计算、机密计算等新技术相结合，进入新的进化阶段，为数字经济构建智能、互联、信任三位一体的创新基础设施。

首届 KubeCon 2020 线上峰会圆满结束，阿里云作为 CNCF 最重要的云原生布道伙伴，此次携 37 位云原生专家与广大开发者分享了 27 场演讲，云原生话题丰富度在本场活动中位居首位。作为峰会的压轴主题演讲，CNCF 理事会成员、阿里云容器服务负责人易立发表演讲《云原生，数字经济技术创新基石》，重点介绍了阿里云如何通过开源云原生技术与开放云原生产品，帮助企业提升面向疫情的应变能力，拥抱数字经济发展的新机遇。与此同时，与大家分享了阿里云对云原生技术普惠和云原生操作系统的思考，并详解阿里云 ACK Pro、ASM、ACR EE、ACK@Edge 等四款企业级容器新品。

## 疫情当前，云原生为数字经济按下快进键

2020 年，疫情突发之后，政府、企业和学校对在线协作的需求猛增。**“上班用钉钉，上学云课堂，出门健康码，订菜送到家”**成了我们日常生活的一部分，数字经济发展势头空前高涨。这背后是一系列云计算、云原生技术支撑的业务创新，**云原生正在打通数字化落地的“最后一公里”。**





**钉钉扛住了有史以来最大的流量洪峰。**春节后第一天复工，钉钉瞬间迎来了远程办公流量高峰：阿里云 2 小时内支撑了钉钉业务 1 万台云主机的扩容需求。基于云服务器和容器化的应用部署方案，让应用发布扩容效率大大提升，为全国用户提供线上工作的流畅体验。

**希沃课堂顺利积累超过 30 万教师开设 200 万节课程。**面对指数级增长的流量，希沃课堂通过容器服务 ACK 高效管理神龙裸金属服务器和 Serverless 弹性容器实例，整个平台兼具高性能、高弹性、低成本、免维护的优势。整体业务性能提升 30%，运维成本降低 50%，给全国教师和学生提供了更好服务。

**支付宝 3 天实现三省精准防控全覆盖。**支付宝加速研发全国统一的疫情防控健康信息码，短短 3 天时间，浙江、四川、海南三省陆续实现健康码全覆盖。云原生大数据平台提供了强大的数据统一、汇集和计算能力。基于云原生应用架构的码引擎系统支持了业务需求的快速迭代，具备弹性、韧性和安全的特点，平稳支撑每日亿次调用，成为广大市民日常息息相关的“健康通行证”。

**盒马全程保障疫情期间居民日常供应。**这背后是盒马整个数字化的供应链体系发挥了重要作用。基于阿里云边缘容器服务 ACK@Edge 底座，盒马团队快速构建了人、货、场数字化全链路融合，云、边、端一体化协同的天眼 AI 系统。结合了云原生技术体系良好的资源调度和应用管理能力，与边缘计算就近访问，实时处理的优势，轻松实现全方位的降本提效，门店计算资源成本节省 50%，新店开服效率提升 70%。

## 三位一体 阿里云打下数字时代新基建的“底色”

云原生架构与技术一定是开放的，需要全行业共同定义与建设。阿里巴巴作为云原生领域的先行者、实践者，基于自身累积多年的最佳实践，持续对社区赋能，为企业提供普惠的云原生产品服务，与开发者共建云原生生态，帮助更多用户分享时代技术红利。

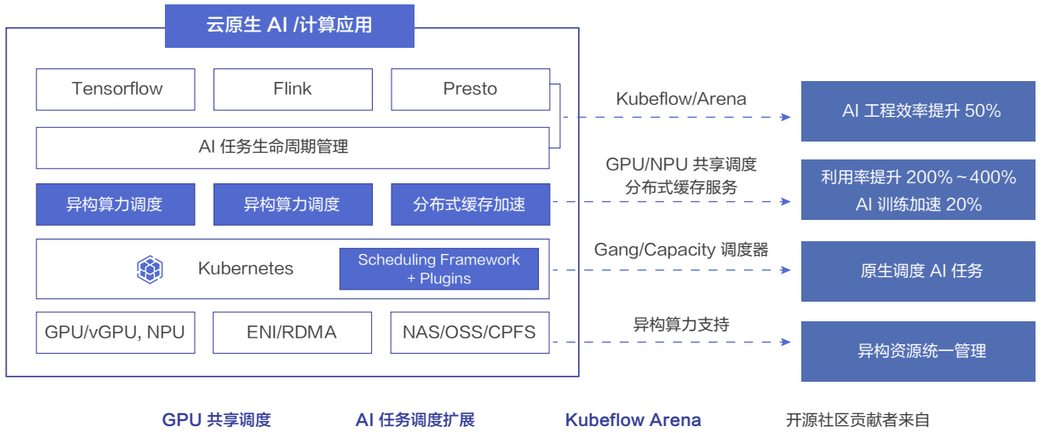
阿里巴巴致力于为数字经济构建智能、互联、信任三位一体的创新基础设施，引领云原生进化新阶段。反观阿里云容器服务团队近期在 AI、边缘、机密计算三个领域的开源新动态，与智能、互联、信任的方向一一对应。

## 1、AI-智能

企业在 IT 转型的大潮中对数字化和智能化的诉求越来越强烈，最突出的需求是如何能快速、精准地从海量业务数据中挖掘出新的商业机会和模式创新，才能更好应对多变、不确定性的业务挑战。

阿里云提供云原生 AI 解决方案从底层异构计算资源，到上层计算框架进行全栈优化。主要特性有统一资源管理、高效共享隔离、统一调度器架构、分布式数据缓存、全生命周期赋能。同时阿里云也将这些成果分享到社区，目前已有来自苹果、IBM、微博等贡献者和我们在开源社区合作共建。

### 加速数字化、智能化升级




开源项目



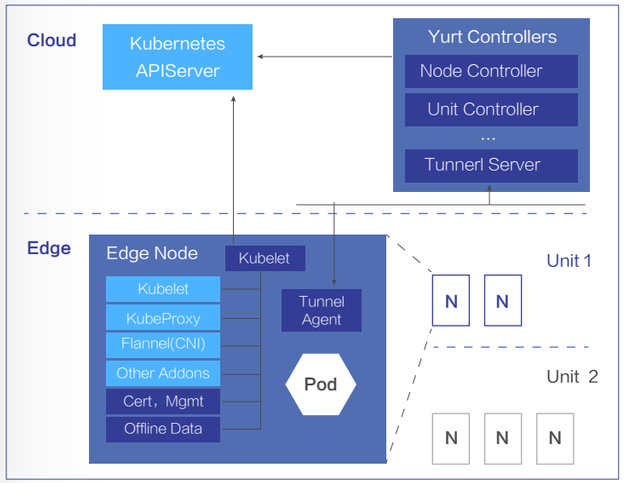


## 2、边缘-互联

阿里云发布边缘容器服务 ACK@Edge，短短一年内，已经应用于音视频直播、云游戏、工业互联网、交通物流、城市大脑等应用场景中，并服务于盒马、优酷、阿里视频云和众多互联网、新零售企业。近期，阿里巴巴正式对外宣布将其核心能力开源，并向社区贡献完整的云原生边缘计算项目—— OpenYurt。它深度挖掘了“边缘计算+云原生落地实施”诉求。

Kubernetes 有强大的容器编排、资源调度能力，但在边缘/ IoT 场景中，对低功耗、异构资源适配、云边网络协同等方面有独特的需求。OpenYurt 主打“云边一体化”概念，通过对原生 Kubernetes 进行扩展的方式支持边缘计算场景需求。未来，OpenYurt 将采用开源社区开发模式，逐步将产品完整能力开源，并持续推动 K8s 上游社区兼顾边缘计算的需求。

### 云边一体迎接万物智联




- 支持原生 K8s：提供完整的 Kubernetes 兼容性，支持原生 K8s 的所有工作负载，以及 Operator、CNI、CSI等
- 无缝转换：为开发者提供了转换工具，可以轻松实现原生 Kubernetes 集群一键转化为 OpenYurt 集群
- 节点自治：具备云边弱网或断网环境下的边缘节点自治能力，保障边缘业务连续性
- 多场景覆盖：支持 x86/ARM 等多种 CPU 体系架构，广泛适用于 工业大脑、城市大脑、音视频、云游戏、IoT 等业务场景

OpenYurt 欢聚时代 YY Inc. 盒马 鲜 美 生活

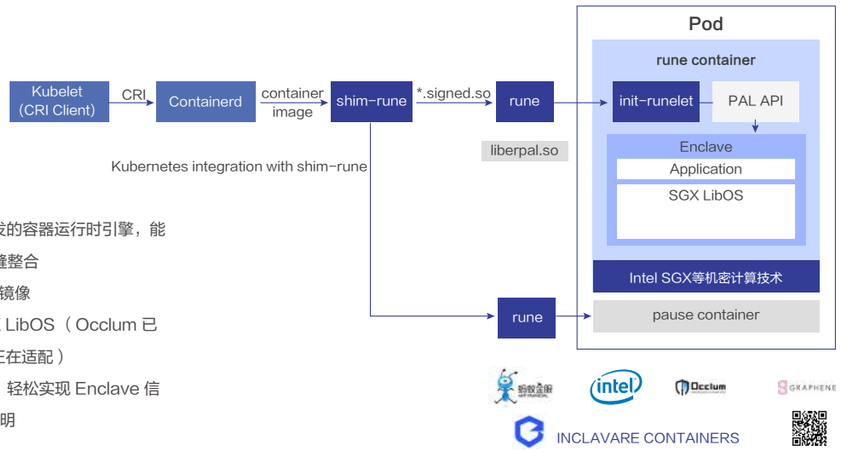
## 3、机密计算-信任

随着数字经济的发展，企业的数据资产成为新的石油。大量的数据需要在云端进行交换、处理。如何保障数据的安全、隐私、可信成为了企业上云的最大挑战。我们需要用技术手段，建立数字化信任以保护数据，帮助企业创建可信任的商业合作关系，促进业务增长。

基于 Intel SGX 等加密计算技术，阿里云为云上客户提供了可信的执行环境。但可信应用开发和使用门槛都很高，需要用户对现有应用进行重构，处理大量的底层技术细节，让这项技术落地非常困难。

Inclavare-Containers 是阿里巴巴开源的，业界第一个面向机密计算的容器运行时，只需对传统应用少量修改即可运行在可信执行环境中，极大提高了可信应用的开发、交付效率。目前已经提供了对蚂蚁金服的 Occlum LibOS 支持，此外，得益于和Intel的紧密合作，Inclavare-Containers 会在近期支持 Graphene。通过云原生技术普及数字化信任，将帮助企业更加安全、简单地处理、融合、分享数据资产。目前，机密计算容器已经在蚂蚁区块链、钉钉等业务场景中得到应用。

### 普及数字化信任



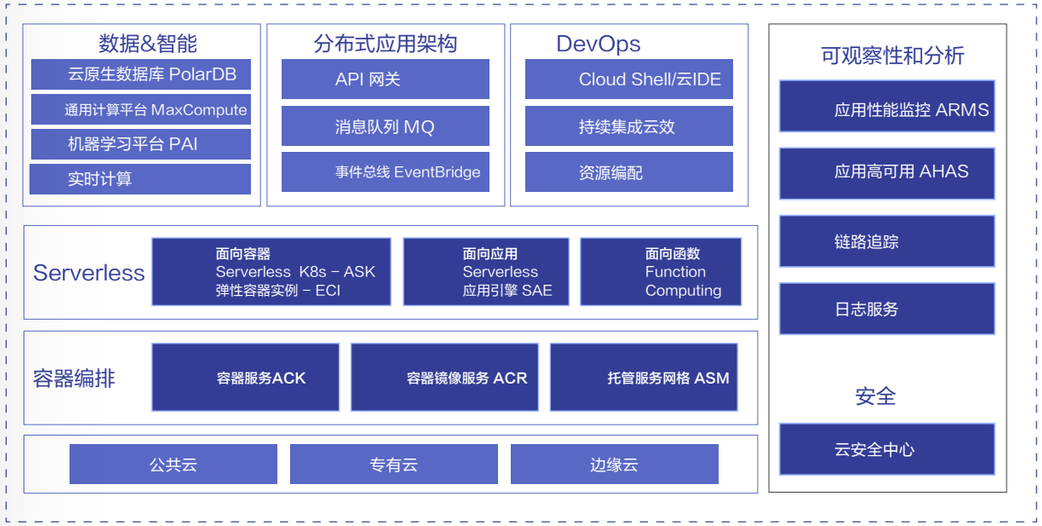

- 基于云原生标准开发的容器运行时引擎，能够与 Kubernetes 无缝整合
- 支持 OCI 标准容器镜像
- 支持业界多款 SGX LibOS（Occlum 已经支持，Graphene 正在适配）
- 提供 runectl 工具，轻松实现 Enclave 信息获取，并进行远程证明





阿里云容器服务连续两年国内唯一进入 Gartner《公有云容器服务竞争格局》报告；在 Forrester 首个企业级公共云容器平台报告中，阿里云容器服务容器服务位列 Strong Performer, 中国第一。此外，阿里云提供了国内最丰富的云原生产品家族，覆盖八大类别 20 余款产品，涵盖底层基础设施、数据智能、分布式应用等，可以满足不同行业场景的需求，帮助企业客户更加简单高效地利用云原生技术。

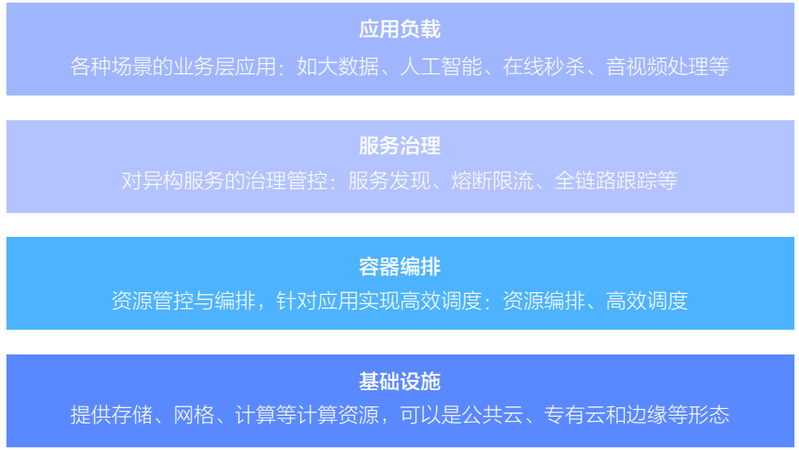
### 国内最丰富的云原生产品家族




十年云原生实践经验，带给阿里巴巴的不止有全球最大规模成功落地的云原生样板间，还有对云原生技术进化趋势的精准把控。智能、互联、信任三位一体的创新基础设施，将成为数字时代新基建的“底色”，阿里巴巴将持续以自身成功经验回馈广大用户，始终以前瞻性视角引领云原生进化，帮助企业找到数字化转型“最短路径”。

## 三位一体 阿里云打下数字时代新基建的“底色”

容器技术的发展揭开了云原生计算的序幕，在易立看来， Kubernetes 为基础的云原生计算也已经成为新的操作系统，云原生操作系统的雏形被越来越多的行业和企业采纳并因此受益：容器应用化、容器编排系统和 Istio 服务网格等技术依次解耦了应用与运行环境、资源编排调度与底层基础设施、服务实现与服务治理等传统架构的依赖关系。



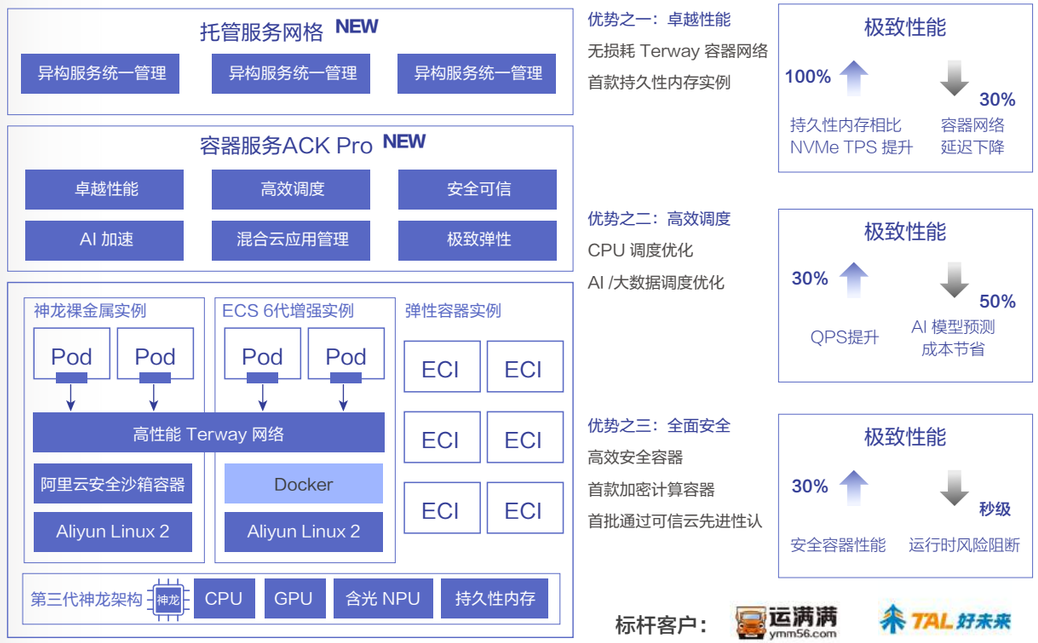

阿里云为用户提供了怎样的云原生操作系统？这个云原生操作系统又有哪些突出特点呢？

首先基础设施层是强大的 IaaS 资源，基于第三代神龙架构的计算资源可以更弹性的扩展，以更加优化的成本提供更高的性能；云原生的分布式文件系统，为容器持久化数据而生；云原生网络加速应用交付能力，提供应用型负载均衡与容器网络基础 设施。





## 云原生操作系统进化




其次在容器编排层，阿里云容器服务自 2015 年上线来，伴随数千家企业客户，共同实践过各行各业大量生产级场景。越来越多的客户以云原生的方式架构其大部分甚至全量应用，随着业务深入发展，为了满足大中型企业对可靠性、安全性的强烈需求，阿里云推出新品可供赔付 SLA 的容器服务企业版 ACK Pro。

## 容器服务企业版 ACK Pro 横空出世：全面安全、高性能，支持新一代神龙架构，SLA 可赔付

容器服务企业版 ACK Pro，是在原容器服务 ACK 托管版集群的基础上发展而来，其继承了原托管版集群的所有优势，例如 Master 节点托管和高可用等。同时，相比原托管版进一步提升了集群的可靠性、安全性和调度性能，并且支持赔付标准的 SLA，高达 99.95%，单集群可支持 5000 节点。ACK Pro 非常适合生产环境下有着大规模业务、对稳定性和安全性有高要求的企业客户。

ACK Pro 支持神龙架构，凭借软硬一体的优化设计，可为企业应用提供卓越的性能；支持无损 Terway 容器网络，相比路由网络延迟下降 30%；为企业提供全面安全防护，支持阿里云安全沙箱容器，满足企业客户对应用的安全、隔离需求，性能相比开源提升 30%。此外，ACK Pro 提供了对异构算力和工作负载优化的高效调度，支持智能 CPU 调度优化，在保障 SLA 和密度的前提下，Web 应用 QPS 提升 30%；支持 GPU 算力共享, AI 模型预测成本节省 50% 以上。

阿里云视频云已在全球十多个区域使用容器服务作为服务基础，有效管理全球万级节点的资源，其中 ACK Pro 切实保障基础设施层大规模计算资源的运维效率和高稳定性，使视频云团队可以将更多时间精力聚焦在视频领域从而为客户提供更多价值。

近日， 阿里云容器服务并成为国内首批通过可信云容器安全先进性认证的企业级容器平台，以 49 个满分项目荣获最高级别先进级认证，特别是在最小化攻击面，二进制镜像验签名，密文的 BYOK 加密等能力上国内领先，达到国际先进水平。

容器服务 ACK Pro 现已正式开启公测，非常适用于互联网、大数据计算、金融政府及跨海业务企业等，欢迎各位感兴趣的客户在官网上申请试用。





## 业内首个全托管 Istio 兼容服务网格 ASM，多种异构应用服务统一治理

在服务网格技术诞生前，应用服务治理的逻辑实现通常都是以代码库的方式嵌入在应用程序中，随着应用复杂度的增长和团队规模的扩大，代码库变更和维护会变地越来越复杂。服务网格通过 Sidecar 代理功能，将服务治理能力与应用程序本身解耦，将服务治理能力标准化、统一化，且更好地支持多种编程语言和技术框架的应用服务。

作为业内首个全托管 Istio 兼容的服务网格产品 ASM，已经正式商业化发布，用户可以在多个地域部署使用。阿里云一开始从架构上就保持了与社区、业界趋势的领先性，控制平面的组件托管在阿里云侧，与数据面侧的用户集群独立。ASM 在托管的控制面侧提供了用于支撑精细化的流量管理和安全管理的组件能力。通过托管模式，解耦了 Istio 组件与所管理的 K8s 集群的生命周期管理，使得架构更加灵活，提升了系统的可伸缩性。

商米科技使用服务网格 ASM 之后，充分享受了 Service Mesh 带来的好处，解决了 GRPC-HTTP 2.0 多路复用引起的负载不均衡的问题，得以分离控制平面和数据平面，受益于 ASM 的可视化方式对控制平面进行管理，对规则配置一目了然。同时还无缝结合链路监控（Tracing Analysis）解决服务化体系下调用链排查追踪问题。

作为多种类型计算服务统一管理的基础设施，ASM 提供了统一的流量管理能力、统一的服务安全能力、统一的服务可观测性能力、以及统一的数据面可扩展能力, 并提出了相应的实践之成熟度模型。针对用户不同的场景, 逐步采用, 分别为一键启用、可观测提升、安全加固、多种基础设施的支持以及多集群混合管理。

## 全球镜像同步加速，ACR EE 保障云原生制品的安全托管及高效分发

容器镜像服务企业版 ACR EE 面向安全及性能需求高，业务多地域大规模部署的企业级客户，提供了公共云首个独享实例模式的企业版服务。

在制品托管部分，ACR EE 除了支持多架构容器镜像，还支持多版本 Helm Chart 等符合 OCI 规范制品托管。在分发及安全治理部分，ACR EE 加强了全球多地域分发和大规模分发支持，提供网络访问控制、安全扫描、安全加签、安全审计等多维度安全保障，助力企业从 DevOps 到 DevSecOps 的升级。目前已有数百家企业线上环境大规模使用，保障企业客户云原生应用制品的安全托管及高效分发。

某国际零售巨头是全球多地域业务形态，存在多地研发协作及多地域部署的场景。采用 ACR EE 之后，客户只需配置实例同步规则，在业务迭代更新容器镜像后，可实现分钟级自动同步至全球指定地域，自动触发 ACK 集群业务部署。完美应对跨海网络链路不稳定、分发不安全的问题，极大提高业务研发迭代效率和部署稳定性，保障企业业务的全球化部署。

除了业务镜像全球自动化部署，也有很多企业通过 ACR EE 的云原生应用交付链功能，通过全链路可追踪、可观测、可自主配置等实践容器化 DevSecOps。

## 业界首创“云边一体化”理念，边缘容器服务 ACK@Edge 正式商业化

与此同时，阿里云深度挖掘了“边缘计算+云原生落地实施”诉求，在去年 KubeCon 上，重磅发布了边缘容器（ACK@Edge），时隔一年，宣布 ACK@Edge 正式商用。此外，ACK@Edge 也将其核心能力开源，并向社区贡献完整的云原生边缘计算项目——OpenYurt。

在过去短短一年的时间里，该款产品已经应用于音视频直播、云游戏、工业互联网、交通物流、城市大脑等应用场景中，并服务于盒马、优酷、阿里视频云和众多互联网、新零售企业。

YY 使用 ACK@Edge 之后，可以 API 统一管理、统一运维边缘容器集群和中心容器集群，边缘算力快速接入并实现边缘节点自治，同时也可以无缝接入 Prometheus 服务实现监控数据上报，总体上运维效率和资源使用率都得到显著改善。

ACK@Edge 适用于丰富的应用场景，包括边缘智能、智慧楼宇、智慧工厂、音视频直播、在线教育、CDN。

除此之外，阿里巴巴近期推出了《云原生架构白皮书》，是国内第一本全方位介绍云原生架构从规划到落地实践的指导书。其中有很多实操性的技术内容和极具借鉴意义的案例参考。希望每一位阅读本书的同学，能够有所收获，成为云原生技术的受益者和创造者。





## 重磅发布：云原生架构白皮书

**业界首本 深度解析云原生架构落地实践**


去年，CNCF 与 阿里云联合发布了《云原生技术公开课》已经成为了 Kubrnetes 开发者的一门“必修课”。今天，阿里云再次集结多位具有丰富云原生实践经验的技术专家，正式推出《云原生实践公开课》。课程内容由浅入深，专注讲解“ 落地实践”。还为学习者打造了真实、可操作的实验场景，方便验证学习成果，也为之后的实践应用打下坚实基础。课程已经正式上线，欢迎大家观看。

# 1/1 Kubernetes + OAM 让开发者更简单

阿里云资深技术专家　李响

作者简介：李响，阿里云智能资深技术专家，负责阿里巴巴大规模集群调度与管理系统，帮助阿里巴巴通过云原生技术初步完成了基础架构的转型，实现了资源利用率与软件的开发和部署效率的大幅提升，并同步支撑了云产品的技术演进。中国首个 CNCF TOC，etcd 作者，阿里巴巴开源开放委员会 TOC。

本次演讲的内容主题是如何让开发者更简单，更方便的使用 Kubernetes。

**来自应用开发者的“灵魂拷问”**

**“ Kubernetes 让 Devops 更复杂了！”**

近年来，阿里巴巴一直致力于推动 Kubernetes 以及云原生生态普及，期间遇到的比较大的挑战来自于云原生应用的开发者。他们提出的诉求在于，阿里巴巴在推动 Kubernetes 普及的过程，似乎让 DevOps 变得更加复杂了。应用开发人员要了解 Kubernetes 的诸多细节，才能够开始尝试使用 Kubernetes。使用过程之中，他们还需要不停的补充和丰富很多边边角角的知识，才能顺畅的使用库。综合来说，Kubernetes 确实让自动化运维的操作和资源分配的流程变得更为顺畅了，但同时，也给开发者增加了非常多的心智负担，开发者需要去学习原子化的低抽象程度的 API，才能够完成日常的工作。所以总体来讲，开发者会认为不管是 Kubernetes，还是云原生，反而让 DevOps 变得更加复杂了。





## Kubernetes 对于应用开发复杂在哪里

**1. 关注点不同**

业务研发


代码、应用、CICD 流水线

**2. 语义与抽象程度不同**

业务运维

扩容策略

当 RT 上升 10% 时，自动扩容 100 个实例

发布策略

当金丝雀实例通过 99% 的测试时，按每小时切 10% 流量的节奏进行发布

**3. 交互与使用习惯不同**

业务研发、运维

图形化界面　命令行工具　IaC 配置语言



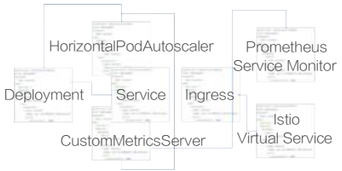


容器　Pod

调度　Controller

Sidecar　Node

CNI　CSI





基于此，我们也一直在思考 Kubernetes 对于应用开发者来讲，到底复杂在什么地方？哪里是可以改进或者增强？在过去的两年里，我们总结出了 3 点不同：

第一点是关注点的不同。Kubernetes 诞生之初是用于对基础设施的管理。比如说 Kubernetes 有一些管理的基础单元，像容器，pod，节点，节点上的网络和存储，都是面向资源的管理和 provision 的操作，向下抽象了 IaaS 层，抽象了云这一层。然而，从业务研发的角度来看，他们更关心的是代码如何进行 deploy？应用在日常生活中如何被稳定的运维？如何连接好 CI/CD 的 pipeline 。对于业务研发来说，他们的关注点更多的是面向应用的维度，而不是面向基础设施，面向资源的关注点。

第二点是语义与抽象程度的不同。Kubernetes 对于基础设施进行管理，提供了大量的原子化能力。比如说 Deployment 抽象一次最为原子的部署，Service 抽象对于不同副本的访问，Ingress 抽象从外部向内部的访问。这些都是最基础的原子性能力，好处是这些能力可以非常容易被组装在一起，形成更高级的抽象。因为每个能力都是原子化的，所以使用起来非常的灵活多变。然而，站在最终用户的角度，他们希望得到的是一个更 high level 的抽象，或者是策略。比如说我们去定义一个扩容策略，具体描述为当 RT 上升 10% 时，自动扩容 100 个实例，再比如金丝雀发布策略。这些策略往往都是由一些原子化的 Kubernetes 所提供的基础设施层能力所组装起来的。所以就语义和抽象程度而言，最终用户想要的与技术层面提供的之间，还有一个 gap 需要去弥补。

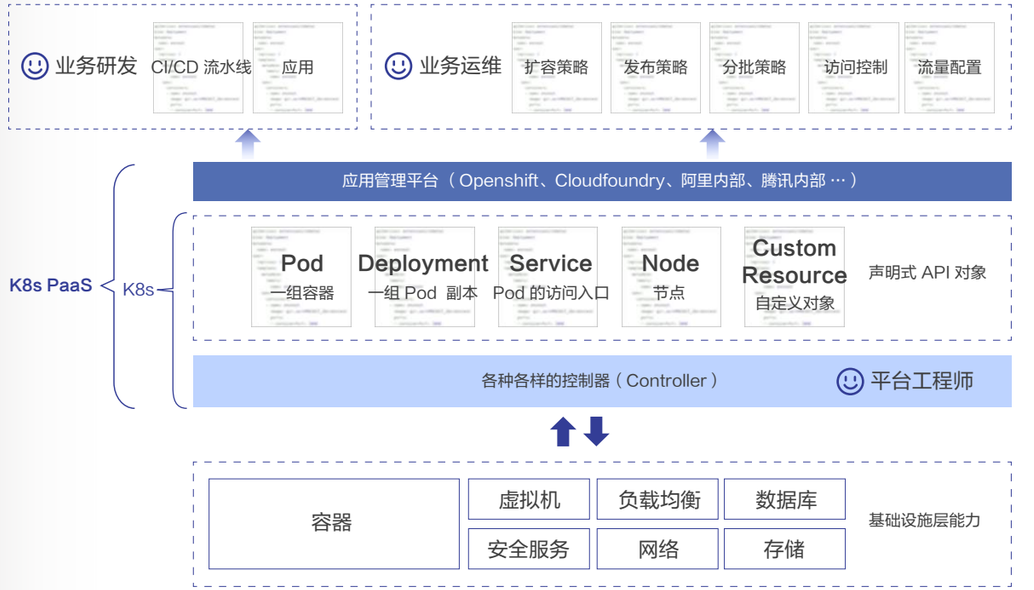
第三点是交互和使用习惯不同。比如说 Kubernetes 其实是基于 YAML 文件面向终态的 API 系统资源的设计。这样设计的好处主要是可以提高自动化程度和描述式程度，然后通过管理所有能力，事无巨细的进行描述，就能够提供最大的灵活度和扩展度。YAML 文件也是非常的机器友好的，可以在机器不同的环境之间来去交互，因为它本质上来讲都是一些数据。另外，YAML 文件也具有一定的可读性。但是，从最终用户的角度来看，还是存在一定的 gap 。因为最终业务研发更希望能够是以最简化的方式进行配置，以代码的方式来进行配置。比如说写 infrastructures code 的语言，来生成这些 YAML 文件，或者使用命令行进行一些简单的操作，然后转换成YAML 文件。在日常的管理工作中，我们需要监控系统状态，看一看运行情况，可能也需要一个图形化界面来去观察。这几点其实也是 Kubernetes 并不具备，也没有提供的。

所以综上三点，其实 Kubernetes 与应用开发之间其实存在一定的 gap ，需要更好的弥补，才能让最终应用研发或者应用运维的人员喜欢上 Kubernetes 云原生体系。





## Alluxio – 分布式缓存的领导者




因此，是不是每一个研发都要做 Kubernetes 专家，都需要自己去弥补鸿沟，才能去在 Kubernetes 上方便合理的运维管理自己的应用？

## 但是，K8s PaaS 正面临着“能力困境”


一方面，现在 Kubernetes 正面临着叫做能力困境的局面。整个云原生体系在快速蓬勃的发展，不管是哪一个专有的 PaaS ，如果 PaaS 是一个有限的能力集合，是一个提供出不可扩展的 API 与能力，总会或多或少的限制研发运维人员对于应用管理日益增长的诉求，尤其是已经有一些已经被云原生生态提供的能力。如何便捷自主的扩展 pass 的平台? 这是我们面临一个巨大挑战。我们不希望被 PaaS 专有管理，不希望被管理的 API 们限制用户想去使用的一些能力。





## 而且，PaaS 还面临着严重分化


另一方面，我们看到基于 Kubernetes 的 PaaS 还面临着严重分化的问题。在同一个公司内部可能就有数套 PaaS ，在公司外部或者在开源社区中也已经有多个基于Kubernetes的 PaaS 。对于一个研发运维人员来讲，他在公司内部需要学习好几个 PaaS 的能力，这将提高学习成本，也降低了运维的效率。尤其是有一些 PaaS 互相之间可能有类似的这些概念，但是有不同的实现激励不同的 behavior ，这可能还会造成稳定性的风险。所以，我们也要解决这样的问题，尽量避免 PaaS 的分化，尽量保证有一些专有的场景中，同样的概念在不同的 PaaS 中行为是一致的，定义是一致的。

## 但是，K8s PaaS 正面临着“能力困境”

**思考：**

1. 基于 Kubernetes　2. 用户友好、高可扩展　3. 统一、标准化

因此，提出了一个我们认为理想的应用管理平台。第一，基于 Kubernetes 充分利用好云原生构建起来活跃且庞大的生态。第二，我们也希望它能够是用户友好，有非常高扩展性的一个 PaaS。第三，我们也希望它能够是一个统一的标准化，大家只需要学习一次就可以把知识在很多场景下复用的一个 PaaS。

## 目标一：一个面向用户，应用为中心

关键词：用户友好，应用层语义和抽象


以下，我们将理想中的 PaaS 拆分为几个目标。

目标一：面向用户，以应用为中心，能够做到用户友好，应用级别的语义和抽象，定义出一些标准化的 CI/CD 应用，以及其他的一些运维相关策略的更高级别的 API，并且是一个统一的可扩展的 API 供大家作为使用。



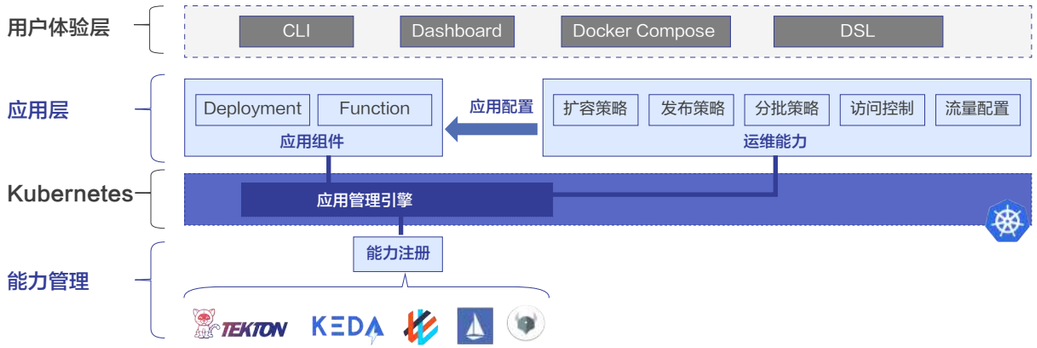


## 目标二：一个高可扩展的应用管理平台

关键词：可插拔，可扩展，模块化，没有抽象程度锁定


目标二：不同于今天一些封闭的 PaaS 管理平台，我们希望当前的 PaaS 平台是可插拔的，可扩展的，模块化的。这就意味着作为一个 PaaS 的用户或者是一个 PaaS 平台的供应商，大家可以自主的去增加或者减少一些能力，在抽象程度上没有锁定，既可以把原生 Kubernetes 或者原生体系中的一些比较低抽象级别的这些能力暴露出来，也可以把一些更高抽象层级别的东西暴露出来。另外，生态体系的能力能够一下就安装起来，进入当前这套 PaaS 平台中，不需要做更多改动。

## 目标三：一个统一、标准化的应用管理引擎




目标三：我们希望所有这一切都是基于同一个框架体系来进行构建的，这样才能做到统一标准，降低大家的心智负担。所以，我们希望能够去定义一个对应用模型，应用运维，应用架构的标准。大家都能够 follow 标准，在框架中定义自己的定制需求。当然在标准中，我们也会提供一些标准的定义，一些标准运维的策略以及标准应用的组件。但在实践层面，我们也希望有一个实现的框架，大家可以基于框架来进行拓展，把一些共有的功能下沉到应用引擎中。这样一来，我们才能够不停的去做大生态，容纳更多的能力。从用户的角度，我们能够看到一个统一比较友好的一致的抽象，降低了心智的负担。

# Open Application Model（OAM）

## 一个用来构建云原生应用管理平台的标准规范与核心框架

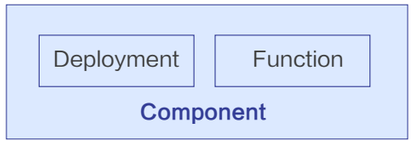
这也是为什么阿里巴巴在去年的 10 月份联合微软发布的一个叫做 OAM 的项目。该项目就是一个用来构建云原生应用管理平台的这种标准规范与核心框架，也是希望去做到以上提到的三个目标。





## OAM + OAM Platform


从 high level 上来讲，OAM 主要分为两个部分，第一个是 OAM 对于应用的抽象和定义。我们把一个应用抽象为两部分，第一部分是应用的组成，是一些 Component 应用的组件。比如你有一个 web 典型的网站应用，它可能包含前端，后端，数据库，每一个都是一个 Component。另外，我们也会定义一组对于应用日常运维或者说运维需求的一些 API 叫做 trait。这些trait都可以去绑定在一个或者多个应用中的 component 上。所以这一组的 components 加上一组的 trait，就完整的定义出来了一个应用。我们把完整定义好的应用标准提交到任意一个 OAM Platform 上，就可以把应用部署并且自动化运维。我们需要是一个 OAM compatible 的实现，我们也会在 Kubernetes 上面构建这样一个实现，能够跟 Kubernetes 对接，充分的利用起来 Kubernetes 生态中非常好的系统，也充分利用整个云原生生态的社区，能够跟大家一起去进行共建。




**Component：**应用中的一个组成部分，例如容器、 Function 或者云服务等

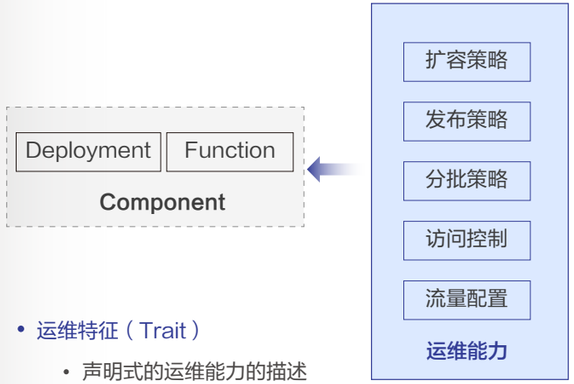
```
$ kubectl get components
NAME        WORKLOAD
frontend    deployment.apps.k8s.io

$ kubectl get deployment
NAME                  REVISION   AGE
frontend-c8bb659c5    1          2d15h
```

```
apiVersion: core.oam.dev/v1alpha2
kind: Component
metadata:
  name: frontend
  annotations:
    description: Container workload
spec:
  workload:
    apiVersion: apps/v1
kind: Deployment
spec:
  template:
    spec:
      containers:
        - name: web
          image: 'php:latest'
          env:
            - name: OAM_TEXTURE
              value: texture.jpg
          ports:
            - containerPort: 8001
              name: http
              protocol: TCP
```

第一点，详细介绍一下什么是 Component。Component 可以是应用中的一个组成部分，比如说是一个容器，一个 function ，或者是一些云的服务。例如，如图所示定义的一个 component ，是一个前端，真正包含的是一个容器，我们把它包含的内容，或者说包含真正运行的东西，我们叫做 workload 。workload 在这里面就是deployment ，熟悉的朋友可能就知道这是 Kubernetes 的一个 deployment，当然也有可能是一个 function 或者是一个 MySQL 的 database 。总而言之，workload 都会被 components 标准抽象的应用组件所封装起来。大家是可以自行拓展 workload，OAM Platform 也会提供一些标准的 workload ，也可以根据自己需求增加更多。








- 运维特征（ Trait ）
  - 声明式的运维能力的描述

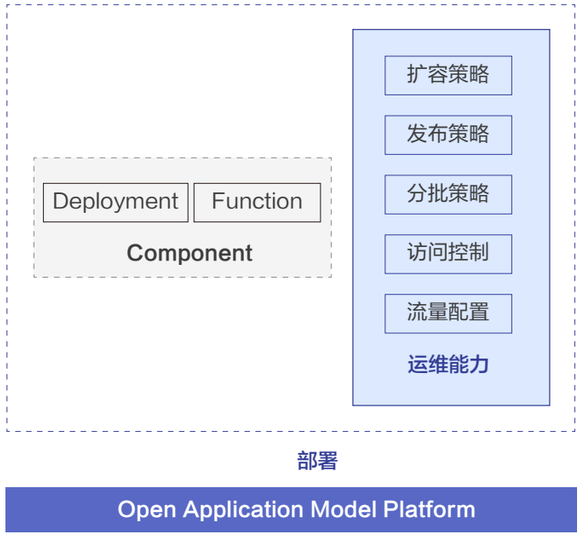
```
    - componentName: frontend
      traits:
        - trait:
            apiVersion: autoscaling/v2beta2
            kind: HorizontalPodAutoscaler
            spec:
              minReplicas: 1
              maxReplicas: 10
        - trait:
                apiVersion:  networking.aliba-
ba-inc.com/v1
            kind: APIGateway
            spec:
              hostname: app.alibaba.com
              path: /
              service_port: 8001
```

第二点，详细介绍一下T rait 。Trait 是一些运维能力，比如扩容发布，访问控制等。我们以描述式的方法来表达这些运维能力，然后绑定在上述 component 应用组件上。图中例子里面有一个Trait 就是 autoscaling，我们可以把 autoscaling 的 Trait 绑定在上述 component，也就是 frontend 上。

当我们定义好 autoscaling 策略，Trait 就会生效。所以大家也可以看到，在这个过程中，我们很好的把应用研发所需要定义的 components 以及应用运维人员所需要定义的 Trait 运维能力做了解耦，分别定义好应用在两方面所需要的不同诉求。

应用配置（Application Configuration）

面向应用维度配置运维能力与组件



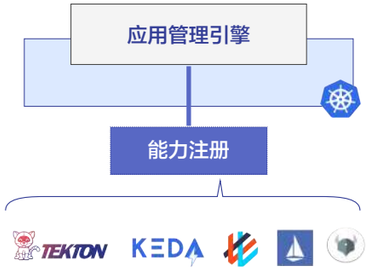

```
apiVersion: core.oam.dev/v1alpha2
kind: ApplicationConfiguration
metadata:
  name: helloworld
spec:
  components:
    # 1st component
    - componentName: frontend
      traits:
        - trait:
            apiVersion: autoscaling/v2beta2
            kind: HorizontalPodAutoscaler
            spec:
              minReplicas: 1
              maxReplicas: 10
        - trait:
            apiVersion: networking.alibaba-inc.com/v1
            kind: APIGateway
            spec:
              hostname: app.alibaba.com
              path: /
              service_port: 8001
    # 2nd component
    - componentName: redis
```

对于一个应用来讲，我们最终会把一些应用组件以及这些应用组件相应的运维能力，通过一个叫做应用配置的 YAML 文件所描述出来。最终把应用配置文件提交给 OAM Planform，就可以去部署应用，并且在日常生活中自动化运维应用。总体来看，OAM 还是一个比较干净简单概念的。





Workload 与 Trait 注册与发现机制




```
apiVersion: core.oam.dev/v1alpha2
kind: TraitDefinition
metadata:
  name: virtualservices.networking.istio.io
  annotations:
    alias: traffic
spec:
  appliesTo:
    - apps.k8s.io
  conflictsWith:
    - services.k8s.io
  definition:  virtualservices.networking.istio.io
```

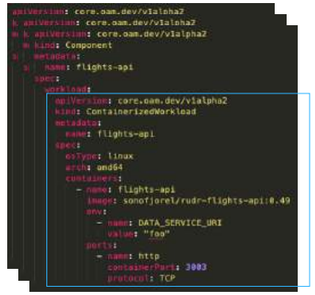
```
# 示例：将 Istio VirtualService
注册为平台的流量管理能力
```

```
$ kubectl get traits
NAME   DEFINITION                          APPLIES TO     CONFLICTS WITH
traffic  virtualservices.networking.istio.io  apps.k8s.io    services.k8s.io
route    route.core.oam.dev                  apps.k8s.io
tls      tls.core.oam.dev                    apps.k8s.io
```

为了让大家能够去自定义 Workload 和 Trait ，我们在 OAM 的平台层也做了一个项目，帮助大家能够更好的去定义以及插入这些能力。图中示例是如何自定义 Trait，利用 Istio 里面所定义的 VirtualService。

大家也可以定义自己的 Workload，比如说自己可以实现一个分布式数据库，像 TiKV 的 workload ，或者大家可能使用自己公司内部的 MySQL ，可以去定义不同的 workload ，注册到 OAM 平台中进行使用。

## 示例: 使用 OAM 模型管理应用



示例：容器化工作负载

示例：手动扩容策略

查看完整演示

**1. 创建应用组件** **2. 绑定运维特征** **3. 应用部署成功**

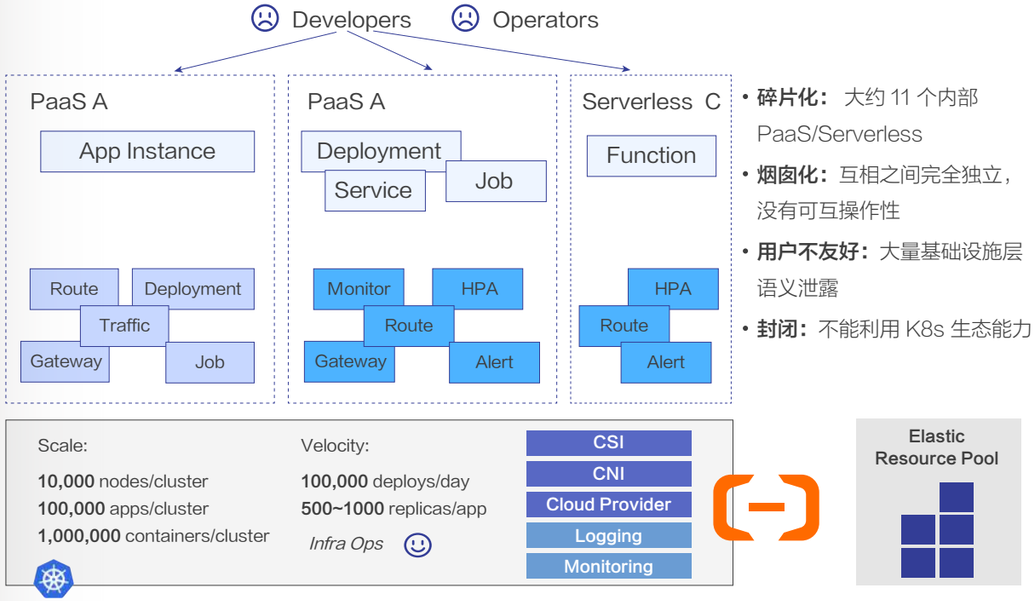
接下来，依托于 PPT 中的简单例子为大家演示如何使用 OAM 。

首先会由应用研发人员去创建这些组件，比如前端会先定义好应用组件，可以是前端需要的 FaaS，或者后端定义一个应用组件，可能是一个 Container 。定义好组件以及组件之间的关联关系之后，会由研发运维人员或者是 DevOps 的同学，去定义好运维所需要的一些能力。最终把运维所需要的能力以及应用所需要的组件，我们以 ApplicationConfiguration 方式包装起来，最终提交给 OAM Planform ，就可以完成一个应用的部署工作。真正这些 dirty work 其实都是 OAM Planform 帮助完成的。





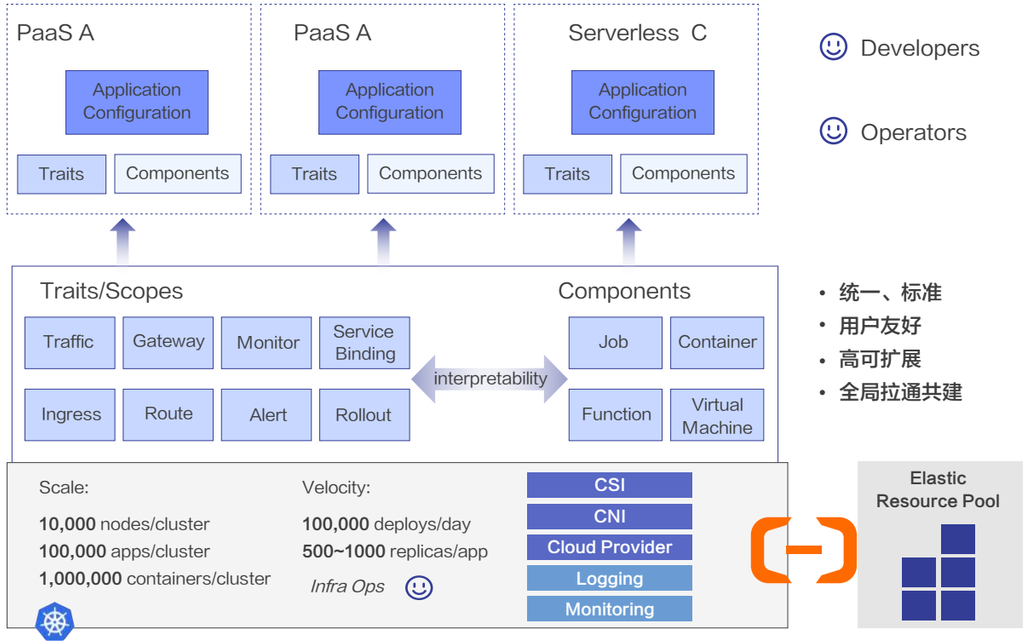
## 案例：过去的阿里巴巴应用管理平台




- **碎片化：** 大约 11 个内部 PaaS/Serverless
- **烟囱化：** 互相之间完全独立，没有可互操作性
- **用户不友好：** 大量基础设施层语义泄露
- **封闭：** 不能利用 K8s 生态能力

在没有 OAM 之前，阿里巴巴实际上一直也在推广 Kubernetes。我们的 Kubernetes 中管理着超过数万个 Application，在上面也是有很多个 PaaS 平台的。为了解决这个问题，我们开始思考怎么提出像 OAM 这样的概念，来让这些平台有一定的 consolidation ，让我们的 Developer 应用开发和运维人员能够变得更高兴。我们也希望阿里巴巴不同的 PaaS 平台之间能够有更好的共建，能够降低 PaaS 平台开发整体的复杂度，也让每一个不同业务上面的 PaaS 有自己专属的能力。

## 示例: 使用 OAM 模型管理应用




- **统一、标准**
- **用户友好**
- **高可扩展**
- **全局拉通共建**





时至今日，我们在一些典型的场景中就在实践像 OAM 的理念。OAM 通过提供一些标准化的Trait 、一些标准的 Components 、以及底层基础引擎层的实现，帮助阿里巴巴内部的应用管理平台做一系列的 consolidation ，这样一来，我们还可以 enable 不同的 PaaS 之间去定义专属的一些能力。这样带来的好处是，不管是 developer ，还是 operator 。它其实对于一些通用能力不用学习多遍，而只需要学习一遍，也可以带来灵活性。不同的业务方还是可以去定义属于自己的一些能力，也满足了最开始我们讲的第一点——抽象程度，都是在应用级别进行抽象，就会非常的用户友好。

总而言之，在最近的半年到一年的时间内阿里巴巴实践了 OAM，并且对 OAM 相当满意。

## OAM 社区欢迎大家!

- **The OAM spec:**
  - https://github.com/oam-dev/spec#community
- **OAM Kubernetes 官方插件（CNCF Sandbox 项目）:**
  - https://github.com/crossplane/oam-kubernetes-runtime
  - A join effort with Crossplane
- **OAM Kubernetes 云原生应用基础平台 – 9 月即将开源**
  - 早期项目成员与合作伙伴招募中！

云原生应用管理交流千人钉钉群

**We Are Hiring!**

**y.zhang@alibaba-inc.com**

**x.li@alibaba-inc.com**

最后，我们的 OAM 社区还是一个相对比较年轻的社区，从 2019 年 10 月份成立到现在将近 10 个月。OAM 的主要分为几部分，第一部分是 OAM 的核心，也就是 OAM 的 spec 标准部分。这部分我们一直是在与微软共建，后续也有 upbound 的同学加入。我们希望能够跟更多的同学一起去共建最核心的spec，希望去看一看我们的 Trait、Components、以及 ApplicationConfiguration 这些概念是不是能够满足大家的需求，是不是能够提供足够的灵活性和方便性。我们最近也在 CNCF 中跟 upbound 微软一起捐赠了一个沙箱项目，叫做 Crossplane，目的是希望能够帮大家抽象基础设施的能力，并且以描述式的方式，以一个 OAM workload 方式表达出来。我们在这里面也有一个项目叫 OAM Kubernetes runtime ，是用来管理 Crossplane 所支持的一些描述式的，以 YAML 文件方式形成的 workload，并且 OAM runtime 也能够让 OAM 和 Kubernetes 之间的交互形成一个桥梁，而不用每个人自己去实现一套。这个“桥梁”项目我们是跟Crossplane大项目一起去共建的。我们也非常希望对 OAM、Kubernetes 以及云上服务集成感兴趣的同学，一起参与到项目中来，跟我们一起共建。

最后，我们有了 OAM spec ，有了一个 OAM 跟 Kubernetes 的核心集成方式，以及 OAM 跟云服务核心集成方式之后，会基于这套推出另外一个项目，叫做云原生基础平台。这个项目将会是一个比较完整的项目，一个大家能够开箱即用的 PaaS 平台。我们希望大家在上面也能够以这个项目为内核去构建出不一样的能力，这个项目会是一个更为核心能力的提供者。不管是阿里巴巴的一些内部 PaaS 体系，还是在阿里云上的一些开放服务，后续都会基于我们即将发布的平台进行拓展。所以，我们最终会是一个内外一致的一个开源，希望这些我们开源东西都是在阿里巴巴内部经过大规模验证的，也希望能够跟社区和用户一起去共建平台。



阿里云原生实践

# 1/2 Alluxio 助力 Kubernetes，加速云端深度学习

阿里云高级技术专家　车漾　|　Alluxio 创始成员，开源社区副总裁　范斌

## 演讲者介绍

范斌：是 Alluxio 的创始成员，曾经就职于 Google，早期负责 Alluxio 的架构设计，现在关注于 Alluxio 开源社区的运营。

车漾：就职于阿里云容器服务团队，关注于云原生技术与AI，大数据场景的结合

## 为什么要加速云端深度学习

人工智能是近几年非常火热的技术领域，而推动这个领域快速演进的原动力包括以英伟达 GPU 为代表的异构算力，以 TensorFlow，Pytorch 为代表的的机器学习框架，以及海量的数据集。除此之外我们也发现了一个趋势，就是以 Kubernetes 和 Docker 为代表的容器化基础架构也成为了数据科学家的首选，这主要有两个因素：分别是标准化和规模化。比如 TensorFlow，Pytorch的软件发布过程中一定包含容器版本，这主要是仰仗容器的标准化特点。另一方面以 Kubernetes 为基础的集群调度技术使大规模的分布式训练成为了可能。

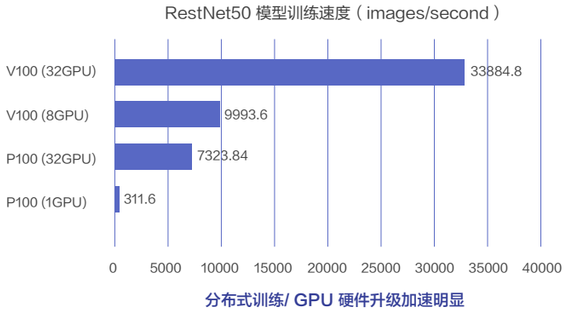
## 背景


首先我们观察下图，这是模拟数据下的深度学习模型训练速度，所谓模拟数据的意思就是这个测试中没有 IO 的影响。从这个图中我们可以得到两个发现：

1.GPU 硬件升级的加速效果显著。从单卡的算力看，pascal 架构为代表的 P100 一秒钟只能处理 300 张图片，而 volta 架构的 v100一秒钟可以处理 1200 张图片，提升了 4 倍

2.分布式训练的也是有效加速的方式。从单卡 P100 到分布式 32 卡 v100，可以看到训练速度提升了 300 倍

## 模拟数据训练速度




分布式训练/ GPU 硬件升级加速明显

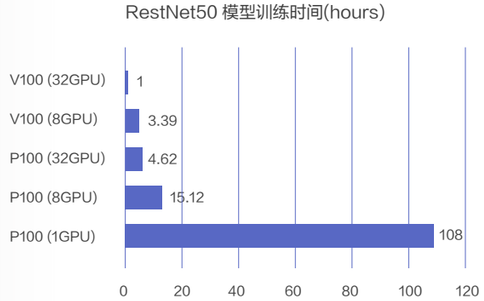




而从训练时间来看，同样的数据，同样的训练目标，单卡 P100 需要 108 个小时，4 天半的时间。而 V100 的 32 卡分布式训练只需要 1 小时。而从成本上来看，单卡 P100 的成本是接近 1400 元，而 8 卡V 100 是 600 元，不到一半。

可以发现，更新的 GPU 硬件不但会更高效，实际上也会更省钱。这也许就是黄教主说的买的越多，省的越多。从云资源的角度来说还是有道理的。

## 模拟数据训练时间




V100 8 卡：157.9 元/小时 x 4 = 631.6 元

P100 1 卡：12.78 元/小时 x108 = 1380.24 元

但是之前的测试结果实际上是做了一些前提假设，就是没有数据延时的影响。而真实的情况下，模型训练是离不开海量数据的访问。而实际上

1.强大的算力需要与之匹配的数据访问能力，不论是延时还是吞吐，都提出了更高的需求。下面的图可以看到，在云盘的数据读取的情况下，GPU 的训练速度直接降为了原来的三分之一。GPU 的使用率也很高。

2.在云环境下，计算和存储分离后，一旦没有了数据本地化，又明显恶化了 I/O 影响

3.此时如果能够把数据直接加载到计算的节点上，比如ossutil把数据拷贝到 GPU 机器是不是可以满足计算的需求呢。实际上也还是不够的，因为一方面数据集无法全集控制，另一方面AI场景下是全量数据集，一旦引入驱逐机制，实际上性能影响也非常显著。因此我们意识到在 k8s 下使用分布式缓存的意义

## Alluxio是什么

Alluxio 是一个面向AI以及大数据应用，开源的分布式内存级数据编排系统。在很多场景底下， Alluxio 非常适合作为一个分布式缓存来加速这些应用。这个项目是李浩源博士在加州大学 Berkeley 分校的 AMPLab 攻读博士的时候创立的，最早的名字 Tachyon。AMPLab 也是孵化出了 Spark 和 Mesos 等优秀开源项目的功勋实验室。2015 年，由顶级的风险投资 Andreessen Horowitz 投资，Alluxio 项目的主要贡献者在旧金山湾区成立了 Alluxio 这家公司。

### Alluxio – 分布式缓存的领导者

ALLUXIO

面向大数据和 AI 应用的内存级数据编排系统

amplab
2013

开源项目由李浩源博士(Alluxio公司CEO)在加州大学 Berkeley 分校 AMPLab 就读期间创立

ANDREESSEN HOROWITZ
2015

由硅谷著名投资公司 Andreessen Horowitz 投资，公司在 2015 年在旧金山湾区成立，致力于推动开源项目和社区以及商业化

## Alluxio 的简介

简单看一下在大数据和 AI 生态圈里， Alluxio 处于什么位置。在大数据软件栈里，Alluxio 是新的一层，我们称之为数据编排层。它向上对接计算应用，比如Spark， Presto，Hive，Tensorflow，向下对接不同的存储，比如阿里巴巴的 OSS，HDFS。我们希望通过这一层新加入的数据编排层，可以让计算和存储之间的强关联解耦。从而让计算和存储都可以独立而更敏捷的部署和演进。数据应用可以不必关心和维护数据存储的具体类型，协议，版本，地理位置等。而数据的存储也可以通过数据编排这一层更灵活更高效的被各种不同应用消费。



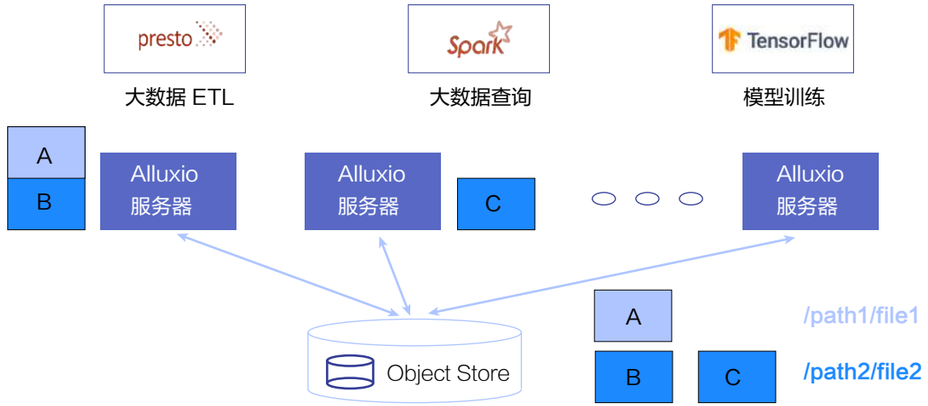



## Alluxio的核心功能

### 1.分布式数据缓存

下面介绍一下 Alluxio 的核心功能。Alluxio 最核心的服务就是提供一个分布式的数据缓存用来加速数据应用。对于 Spark，Presto，Tensorflow 等数据密集型的应用，当读取非本地的数据源时，Alluxio 可以通过加载原始数据文件，将其分片以及打散，并存储在靠近应用的 Alluxio 服务器上， 增强这些应用的数据本地性。

比如在这个例子里， 文件1和文件 2 分别被分片后存储在不同的 Alluxio 服务器上，应用端可以就近从存储了对应的数据分片的服务器读取。当应用需要的读入有明显的热数据时， 添加缓存层可以显著的节省资源以及提升效率。



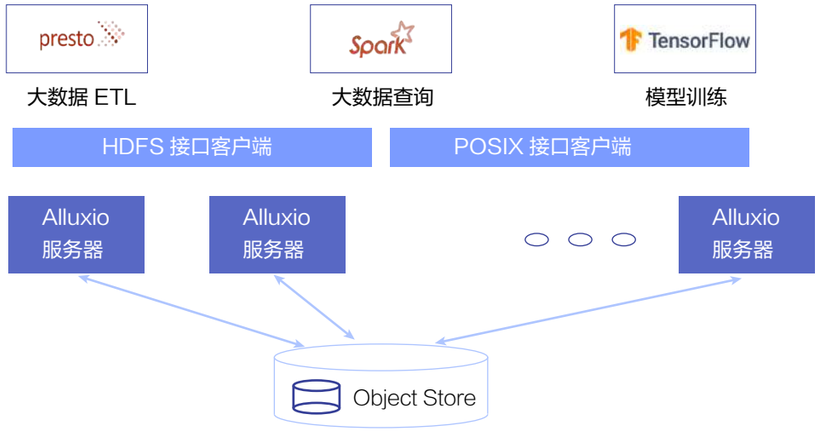

## 2.灵活多样的数据访问 API

Alluxio 的第二个核心应用，是对应用提供不同类型的数据接口，包括在大数据领域最常见的 HDFS 接口，以及在 ai 和模型训练场景下常用的 POSIX 标准文件系统接口。

这样同样的数据一旦准备完毕， 可以以不同的形式呈现给应用，而不用做多次处理或者 ETL



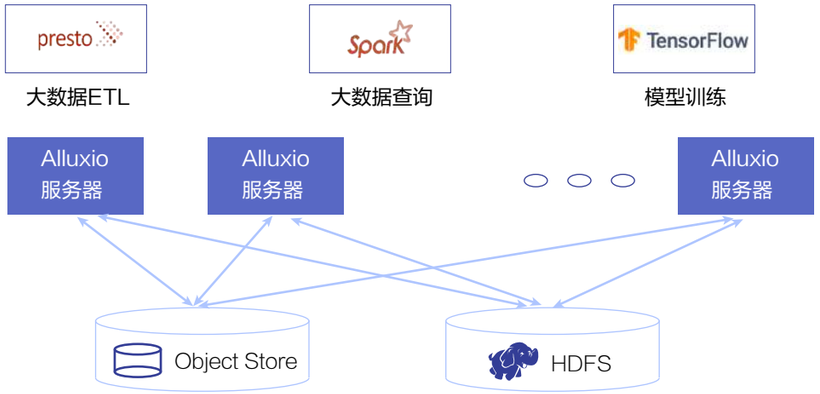





## 3.统一文件系统抽象

Alluxio 的第三个核心功能，是把多个不同的存储系统，以对用户透明的方式，统一接入一个文件系统抽象中。这样使得复杂的数据平台变得简单而易于维护。数据消费者，只需要知道数据对应的逻辑地址，而不用去关心底层对接的时候什么存储系统。

举个例子， 如果一家公司同时有多个不同的 HDFS 部署，并且在线上接入了 Alibaba 的 OSS 服务， 那么我们完全可以使用 Alluxio 的挂载功能，把这些系统接入一个统一的逻辑上的 Alluxio 文件系统抽象中。每一个 HDFS 会对应到不同的 Alluxio 目录。




## Alluxio 在云端 AI 训练场景的性能好处

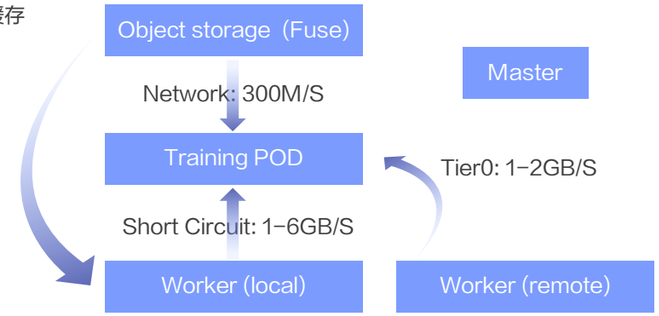
介绍完了 Alluxio 的核心功能，让我们聚焦在云端AI训练场景下，再来回顾一下 Alluxio 可能带来的好处。在模型训练场景下内存加速才能满足 GPU 需要的高吞吐。如果通过普通的网络从 Object store 传输数据， 大约能支撑 300MB/s， 这远远不能达到充分使用训练资源特别是 GPU 高吞吐的特性。但是一旦利用 Alluxio 构建了一层分布式的数据缓存，负责训练的容器进程和 alluxio worker容器进程就可以以很高的速率交换数据。比如当两者在同一物理主机上的时候， 可以达到 1-6GB 每秒。从其他 alluxioworker 处读取也可以通常达到 1-2GB/s 。

此外，通过 Alluxio 可以实现非常简单便捷的分布式缓存管理，比如设置缓存替换策略，设置数据的过期时间，预读取或者驱逐特定目录下的数据等等操作。这些都可以给模型训练带来效率的提升和管理的便捷





- 支持大规模的数据缓存
- 本地内存加速
- 支持数据预热
- LRU 缓存管理

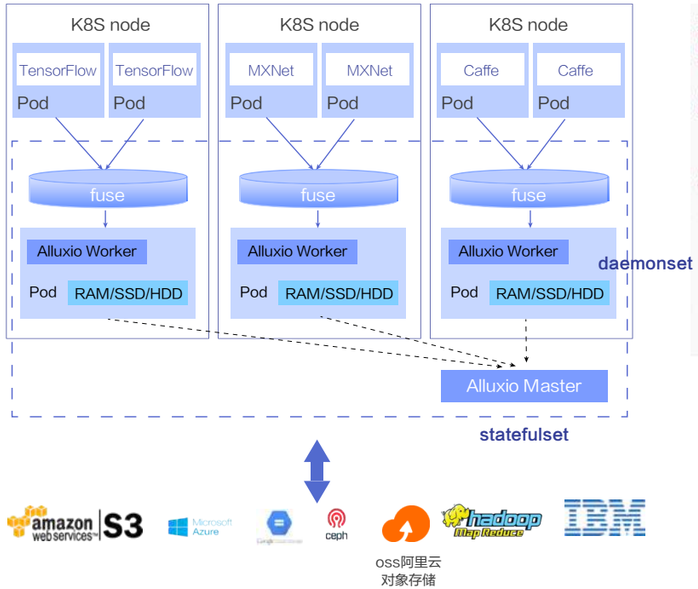



## Alluxio 在 Kubernetes 上的架构

要在 Kubernetes 中原生的使用 Alluxio，首先就要把它部署到 K8s 中，因此我们的第一步工作和 Alluxio 团队一起提供一个 Helmchart，可以统一的配置用户身份，参数以及分层缓存配置。

从左图中看，这里 Alluxio 的 master 以 statefulset 的模式部署，这是因为 Alluxiomaster 首先需要稳定，唯一的网络 id，可以应对容灾等复杂场景。而 worker 和 Fuse 以 daemonset 的模式部署，并且二者通过 podaffinity 绑定，这样可以使用到数据亲和性。

## KubeNode – Remedy Operator




```
# Security Context
user: 1000
group: 1000
fsGroup: 1000

# Site properties for all the components
properties:
  alluxio.user.metrics.collection.enabled: 'true'
  alluxio.security.stale.channel.purge.interval: 365d

#
tieredstore:
  levels:
  - level: 0
    mediumtype: MEM
    quota: 2GB
    high: 0.95
  - level: 1
    mediumtype: SSD
    quota: 10GB
    high: 0.99
```

通过将应用完成 helm 化之后，部署它就变成了非常简单的事情，只需要编写 cong.yaml,执行 helminstall 就可以一键式在 Kubernetes 中部署 Alluxio。大家感兴趣的话可以查看 alluxio 文档，或者借鉴阿里云容器服务的文档。

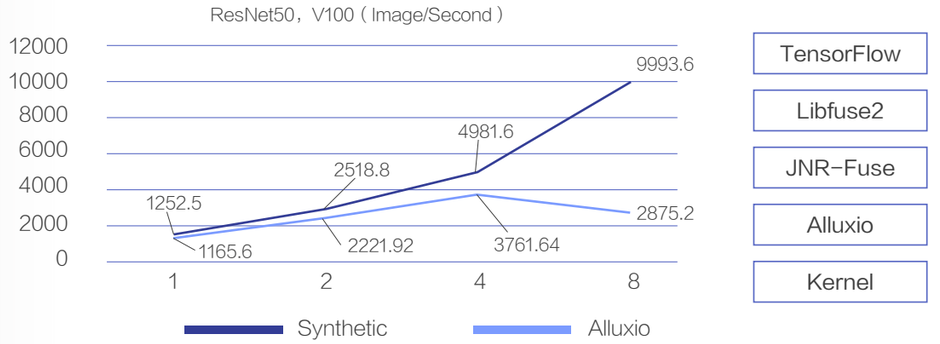




## Alluxio 支持 AI 模型训练场景的挑战

在性能评估中，我们发现当 GPU 硬件从 NVidia P100 升级到 NVidia V100 之后，单卡的计算训练速度得到了不止 3 倍的提升。计算性能的极大提升给数据存储访问的性能带来了压力。这也给 Alluxio 的 I/O 提出了新的挑战。

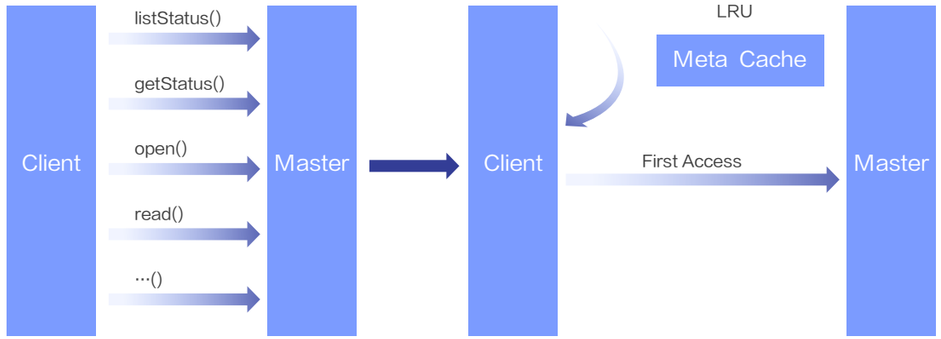
下图是在分别在合成数据 (Synthetic Data) 和使用 Alluxio 缓存的性能对比，横轴表示 GPU 的数量，纵轴表示每秒钟处理的图片数。合成数据指训练程序读取的数据有程序自身产生，没有 I/O 开销，代表模型训练性能的理论上限; 使用 Alluxio 缓存指训练程序读取的数据来自于 Alluxio 系统。在 GPU 数量为 1 和 2 时，使用 Alluxio 和合成数据对比，性能差距在可以接受的范围。但是当 GPU 的数量增大到4时，二者差距就比较明显了，Alluxio 的处理速度已经从 4981 images/second 降到了3762 images/second。 而当 GPU 的数量达到 8 的时候，Alluxio 上进行模型训练的性能不足合成数据的 30%。而此时通过系统监控，我们观察到整个系统的计算、内存和网络都远远没有达到瓶颈。这间接说明了简单使用 Alluxio 难以高效支持 V100 单机 8 卡的训练场景。



# 调优策略

## 1. 缓存元数据减少 gRPC 交互

Alluxio 不只是一个单纯的缓存服务。它首先是一个分布式虚拟文件系统，包含完整的元数据管理、块数据管理、UFS 管理（UFS 是底层文件系统的简称）以及健康检查机制，尤其是它的元数据管理实现比很多底层文件系统更加强大。这些功能是 Alluxio 的优点和特色，但也意味着使用分布式系统带来的开销。例如，在默认设置下使用 Alluxio 客户端来读一个文件，即便数据已经缓存在本地的 Alluxio Worker 中，客户端也会和 Master 节点有多次 RPC 交互来获取文件元信息以保证数据的一致性。完成整个读操作的链路额外开销在传统大数据场景下并不明显，但是深度面对学习场景下高吞吐和低延时的需求就显得捉襟见肘了。因此我们要提供客户端的元数据缓存能力。








## 2. Alluxio 缓存行为控制

由于深度学习训练场景下，每次训练迭代都是全量数据集的迭代，缓存几个 TB 的数据集对于任何一个节点的存储空间来说都是捉襟见肘。而 Alluxio 的默认缓存策略是为大数据处理场景（例如，查询）下的冷热数据分明的需求设计的，数据缓存会保存在 Alluxio 客户端所在的本地节点，用来保证下次读取的性能最优。具体来说

1. alluxio.user.ufs.block.read.location.policy 默认值为 alluxio.client.block.policy.LocalFirstPolicy，这表示 Alluxio 会不断将数据保存到 Alluxio 客户端所在的本地节点，就会引发其缓存数据接近饱和时，该节点的缓存一直处于抖动状态，引发吞吐和延时极大的下降，同时对于 Master 节点的压力也非常大。因此需要 location.policy 设置为 alluxio.client.block.policy.LocalFirstAvoidEvictionPolicy 的同时，指定 alluxio.user.block.avoid.eviction.policy.reserved.size.bytes 参数，这个参数决定了当本地节点的缓存数据量达到一定的程度后，预留一些数据量来保证本地缓存不会被驱逐。通常这个参数应该要大于节点缓存上限 X（100%-节点驱逐上限的百分比）。

2. alluxio.user.file.passive.cache.enabled 设置是否在Alluxi的本地节点中缓存额外的数据副本。这个属性是默认开启的。因此，在Alluxio客户端请求数据时，它所在的节点会缓存已经在其他 Worker 节点上存在的数据。可以将该属性设为 false，避免不必要的本地缓存。

3. alluxio.user.file.readtype.default 默认值为 CACHE_PROMOTE。这个配置会有两个潜在问题，首先是可能引发数据在同一个节点不同缓存层次之间的不断移动，其次是对数据块的大多数操作都需要加锁，而 Alluxio 源代码中加锁操作的实现不少地方还比较重量级，大量的加锁和解锁操作在并发较高时会带来不小的开销，即便数据没有迁移还是会引入额外开销。因此可以将其设置为 CACHE 以避免 moveBlock 操作带来的加锁开销，替换默认的 CACHE_PROMOTE

**策略：1.优先本地加载缓存 2.避免数据震荡 3.避免数据冗余**

| | 参数 | 取值 | 含义 |
|---|---|---|---|
| 1 | alluxio.user.ufs.block.read.location.policy | LocalFirstAvoidEvictionPolicy | Alluxio 读取的数据块优先保存到本地，但是当本地空间不足时，不会驱逐本地数据块而是将读取数据缓存到邻近节点 |
| 2 | alluxio.worker.tieredstore.level0.dirs.quota | 100GB | 缓存存储的容量上限。 |
| | alluxio.worker.tieredstore.level0.watermark.high.ratio | 0.99 | 后台驱逐任务启动条件，本例子中条件本地空间超过 100 x 0.99=99 GiB 触发驱逐 |
| | alluxio.user.block.avoid.eviction.policy.reserved.size.bytes | 1056MB | 当本地节点的空间少于 1056MB 时，数据缓存的调度器不会选择该节点；转而选择其他节点。 |
| 3 | alluxio.user.file.passive.cache.enabled | false | 当从 Alluxio 远程 worker 读文件时，是否缓存文件到 Alluxio 的本地 worker。 |
| | alluxio.user.file.readtype.default | CACHE | 默认的 CACHE_PROMOTE 会带来显著的性能开销 |

## 3. Fuse性能调优

1. 延长 FUSE 元数据有效时间

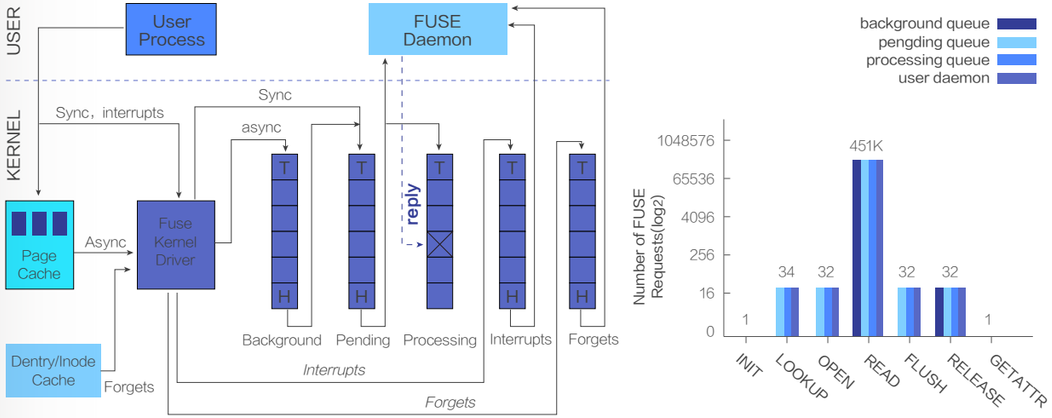
Linux 中每个打开文件在内核中拥有两种元数据信息：`struct dentry`和`struct inode`，它们是文件在内核的基础。所有对文件的操作，都需要先获取文件这两个结构。所以，每次获取文件/目录的 inode 以及 dentry 时，FUSE 内核模块都会从 libfuse 以及 Alluxio 文件系统进行完整操作，这样会带来数据访问的高延时和高并发下对于 Alluxio Master 的巨大压力。可以通过配置 `-o entry_timeout=T -o attr_timeout=T` 进行优化。





2. 配置 `max_idle_threads` 避免频繁线程创建销毁引入 CPU 开销。

这是由于 FUSE 在多线程模式下，以一个线程开始运行。当有两个以上的可用请求，则 FUSE 会自动生成其他线程。每个线程一次处理一个请求。处理完请求后，每个线程检查目前是否有超过`max_idle_threads` (默认10)个线程；如果有，则该线程回收。而这个配置实际上要和用户进程生成的 I/O 活跃数相关，可以配置成用户读线程的数量。而不幸的是 `max_idle_threads` 本身只在 libfuse3 才支持，而 AlluxioFUSE 只支持 libfuse2，因此我们修改了 libfuse2 的代码支持了 `max_idle_threads` 的配置。



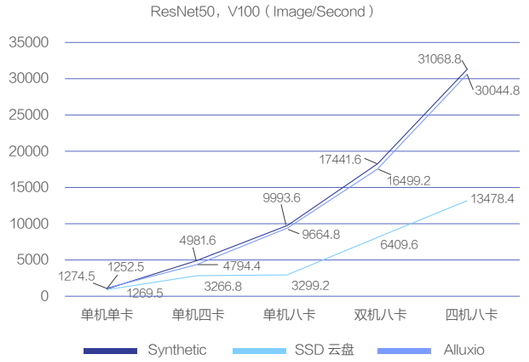

**建议**

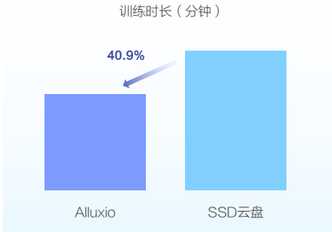
- 选择更高版本的 kernel
- 设置 max_read=131072
- 选定制 libfuse2 代码，支持配置 Libfuse 线程池
- 延长元数据缓存时间

## 总结

在优化 Alluxio 之后，ResNet50 的训练性能单机八卡性能提升了 236.1%，并且扩展性问题得到了解决，训练速度在不但可以扩展到了四机八卡，而且在此场景下和合成数据相比性能损失为 3.29%(31068.8images/s vs 30044.8 images/s)。相比于把数据保存到 SSD 云盘，在四机八卡的场景下，Alluxio 的性能提升了 70.1% (云 SSD 17667.2 images/s vs 30044.8 images/s)。

## 端到端的优化方案




如果您对通过 Alluxio 在 Kubernetes 中加速深度学习感兴趣，欢迎钉钉扫码加入中国社区大群。我们一起来讨论您的场景和问题。

Alluxio 中国社群大群





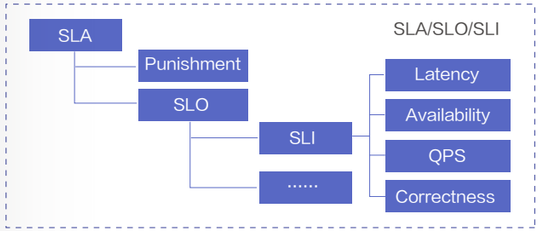
# 1/3 在大规模 Kubernetes 集群上实现高 SLO 的方法

蚂蚁金服技术专家　姚菁　|　蚂蚁金服高级开发工程师　范康

随着 Kubernetes 集群的规模和复杂性的增加，集群越来越难以保证高效率，低延迟的交付 pod。在本次演讲中，我们将分享蚂蚁金服在设计 SLO 架构和实现高 SLO 的方法和经验。

## Why SLO?

SLO (Service-Level Objective). Within service-level agreements (SLAs), SLOs are the objectives that must be achieved — for each service activity, function and process — to provide the best opportunity for service recipient success. — Gartner




SLI defines an indicator, which can represent user experience.

SLO is the object that try to meets all SLIs in a period of time.

SLA = SLO + Punishment.

Gartner 对 SLO 的定义：在SLA框架下，SLO 是系统必须要达到的目标；需要尽可能地保障调用方的成功。有些人可能会对 SLI, SLO, SLA 有困惑，可以先来看下下面三者的关系图。

SLI 定义一个指标，来描述一个服务有多好算达到好的标准。比如 Pod 在 1min 内交付。我们通常从迟延，可用性，吞吐率，成功率这些角度来制定 SLI。

SLO 定义了一个小目标，来衡量一个 SLI 指标在一段时间内达到好的标准的比例。比如说，99% 的 Pod 在1min内交付。当一项服务公布了其 SLO 的以后，用户方就会对该服务的质量有了期望。

SLA 是 SLO 衍生出来的协议，常用于 SLO 定义的目标比例没有完成时，服务方要赔多少钱。通常来说，SLA的协议会具体白纸黑字形成有法律效率的合同，常用于服务供应商和外部客户之间（例如阿里云和阿里云的使用者）。一般来说对于内部服务之间的 SLO 被打破，通常不会是经济上的赔偿，可能更多的是职责上的认定。

所以，我们在系统内部更多关注的是 SLO。

## What we concern about Large k8s Cluster

### Is the cluster healthy

1 Are all software components working fine

2 How many failures occurred on the cluster

### What happened about the cluster

1 Is there something unexpected happened in the cluster

2 What end users did in the cluster

### How to locate failure

1 Which component is going wrong

2 Which component that leads delivery of the pod to failure





随着生产环境不断发展，K8s 集群越来越复杂，集群规模不断增大。如何保障大规模环境 K8s 集群的可用性，是摆在众多厂家面前的一个难题。对于 K8s 集群，我们通常关心以下问题。

第一个问题就是集群是否健康，所有组件是否正常工作，集群中 Pod 创建的失败数量有多少；这是一个整体指标的问题；

第二个问题就是集群中发生了什么，集群中是否有异常发生了，用户在集群中做了些什么事情；这是一个追踪能力的问题；

第三个问题就是有了异常后，是哪个组件出了问题导致成功率降低。这是一个原因定位的问题。

那么，我们该如何解决上面的问题呢。

首先，我们要定义一套 SLO，来描述集群的可用性；

接着，我们必须有能力对集群中 Pod 的生命周期进行追踪，对于失败的 Pod，还需要分析出失败原因，以快速定位异常组件；

最后，我们要通过优化手段，消除集群的异常。

## SLIs on Large k8s Cluster

### 1. Cluster health state

A combination value indicates the risk in the cluster. Currently Healthy, Warning and Fatal are the possible value.

### 2. Success rate

A rate value indicates the rate of success about creating/upgrading pod.

### 3. Number of Terminating Pod

A number value indicates the count of pods that can not be deleted in a certain period.

### 4. Centralized Components Availability

A ratio value indicates the time in which the cluster is available. It is used to evaluate the master components.

### 5. Nodes Availability

A number value indicates the number of unhealthy node in the cluster. Pods scheduled to unhealthy nodes may not be delivered in time, success rate would decrease consequently.

我们先来看下集群的一些指标。

第一项，集群健康度，目前有 Healthy，Warning，Fatal 三个值来描述。Warning 和 Fatal 对应着告警体系，比如 P2 告警发生，那集群就是 Warning；如果 P0 告警发生，那集群就是 Fatal，必须进行处理。

第二项指标，成功率，这里的成功率是指 Pod 的创建成功率。Pod 成功率是一个非常重要的指标，蚂蚁一周 Pod 创建量是百万级的，成功率的波动会造成大量的 Pod 的失败；而且 Pod 成功率的下跌，是集群异常的最直观反应。

第三项指标，残留 Terminating Pod 的数量。为什么不用删除成功率呢，因为在百万级别的时候，即使 Pod 删除成功率达到 99.9%，那么 Terminating Pod 的数量也是千级别的。残留如此多的 Pod，会占着应用的容量，在生产环境中是不可接受的。

第四项指标，服务在线率，服务在线率是通过探针来衡量的，探针失败，意味着集群不可用。服务在线率是会对 Master 组件来设计的。

最后一项，故障机数量，这是一个节点维度的指标。故障机通常是指那些无法正确交付 Pod 的物理机，可能是磁盘满了，可能是 load 太高了。集群故障机并须做到“快速发现，快速隔离，及时修复”，毕竟故障机会对集群容量造成影响。

## The success standard and reason classification

### The success standard:

| Pod | Feature | Time limit | Success condition |
|---|---|---|---|
| Pod | RestartPolicy=Always | 1min (example value) | the status of {.Status.Conditions. “type==Ready”} is "True" |
| Job Pod | RestartPolicy= Never/OnFailure | 1min | {.Status.Phase} == Running/Succeeded/Failed |
| Terminating Pod | | 1min | pod is removed from etcd |
| Unhealthy Node | Taint/Degrade | 1min | Node has taints or is degraded |
| | Processing | Base on the failure reason | Unhealth node is healed or removed. |





### Reason classification:

| Source | Feature | Example |
|---|---|---|
| System | Failure caused by cluster itself | RuntimeError, ImageFailed, Unscheduled, KubeletDelay... |
| End Users | Failure caused by end users | ContainerCrashLoopBackOff, FailedPostStartHook, Unhealthy… |

有了集群的指标后，我们需要把这些指标进行细化，定义出成功的标准。

先来看 Pod 创建成功率指标。我们把 Pod 分为了普通 Pod 和 Job 类 Pob。普通 Pod 的 RestartPolicy 为 Never，Job 类 Pod 的 RestartPlicy 为 Never 或 OnFailure。两者的都设定有交付时间，比如必须在 1 分钟以内完成交付。普通 Pod 的交付标准是 1min 内 Pod 已经 Ready；Job 类 Pod 的交付标准是 1min 内 Pod 的状态已达 Running，Succeeded，或 Failed。当然创建的时间需要把 PostStartHook 执行时间排除。

对于 Pod 的删除，成功的标准为在规定时间内，Pod 从 ETCD 内删除。当然，删除的时间需要把 PreStopHookPeriod 时间排除。

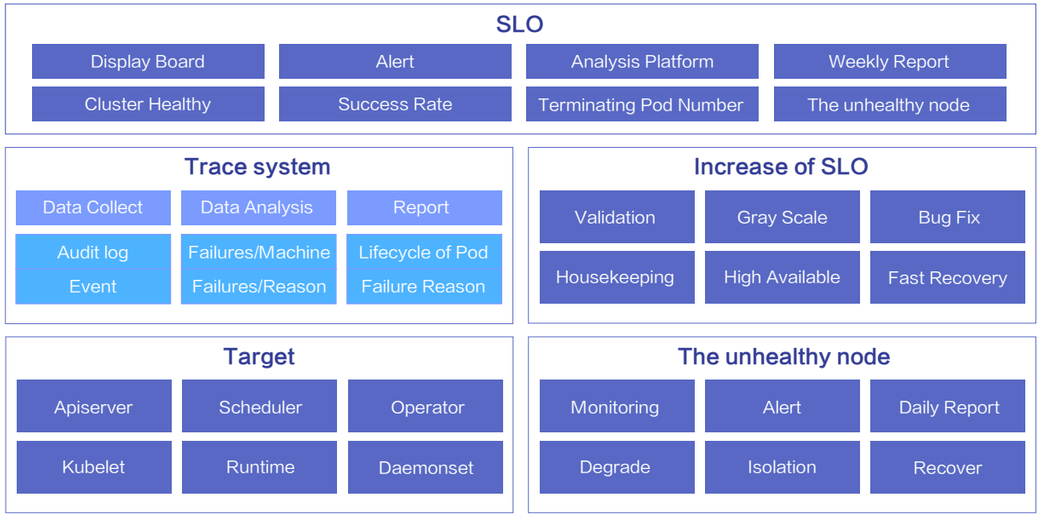
对于故障机，要尽快的发现并进行隔离和降级。比如物理机磁盘只读，那必须在 1min 内完成对该 Pod 打 taint。至于故障机的恢复时间，需要按不同的故障原因，制定不同的恢复时间。比如系统故障需要重要安装系统，那恢复时间就会长些。

有了这些标准后，我们也对 Pod 失败的原因进行了整理，有些失败原因是系统引起的，我们需要关心的；有些失败原因是用户引发的，是我们不需要关心的。

比如 RuntimeError，就是一个系统错误，底层 Runtime 有问题了；ImagePullFailed，Kubelet 下载镜像失败，由于蚂蚁有 Webhook 对镜像准入做了校验，所有镜像下载失败一般都是系统原因造成的。

对于用户原因，在系统侧无法解决，我们只把这些失败原因以接口查询的方式提供给用户，让用户自己解决。比如 ContainerCrashLoopBackOff，通常是由于用户容器退出引起；

## The infrastructure




**SLO:**

Indicate the cluster is healthy or there is something unexpected happened.

**Trace system:**

Collect and analyze logs in cluster. So we can known what happened about the cluster.

**Increase of SLO:**

Get the weakness of the cluster by analyzing the failure reasons and effective actions should be taken to increase the success rate.

**The unhealthy node:**

Detect the unhealthy machines timely and fix problems automatically。





围绕 SLO 目标，我们构建了一整套体系，一方面用于向终端用户、运维人员展示当前集群各项指标状，另一方面，各个组件相互协作，通过分析当前集群状态，得到影响 SLO 的各项因素，为提升集群 pod 交付成功率提供数据支持。

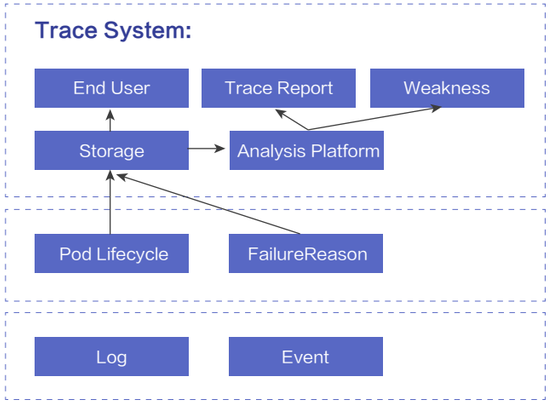
自顶向下而看，顶层组件主要面向各种指标数据，如集群健康状态，pod创建、删除、升级成功率，残留 pods 数量，不健康节点数量等指标。其中 Display Board 就是我们常说的监控大盘。

我们同样构建了 Alert 告警子系统，支持灵活的配置方式，可以为不同的指标，根据指标的下跌百分比，指标下跌绝对值等配置多种告警方式，如电话，短信，邮件等。

Analysis System 通过分析指标历史数据以及采集到的节点metrics和master组件指标，给出更详细的集群运营报告。其中 Weekly Report 子系统给出当前集群本周pod创建/删除/升级的数据统计，以及失败案例原因汇总。Terminating Pods Number 给出一段时间内集群内新增的无法通过k8s机制删除的 pods 列表和 pods 残留原因。Unhealthy Nodes 则给出一个周期内集群所有节点的总可用时间占比，每个节点的可用时间，运维记录，以及不能自动恢复，需要人工介入恢复的节点列表。

为了支撑上述这些功能，我们开发了 Trace System，用来分析展示单个 pod 创建/删除/升级失败的具体原因。其中包含日志和事件采集、数据分析，pod 生命周期展示三个模块。日志和事件采集模块采集各 master 组件以及节点组件的运行日志和 pod，node 事件，分别以 pod/node 为索引存储日志和事件。数据分析模块分析还原出 pod 生命周期中各阶段用时，以及判断 pod 失败原因，节点不可用原因。最后，由 Report 模块向终端用户暴露接口和 UI，向终端用户展示 pod 生命周期以及出错原因。

## The trace system




**Trace Result:**


```
"SLO数据" : {
    "ContainersReady时间" : "                    ",
    "Pod Running时间" : "                    ",
    "Pod类型" :                    ,
    "ReadyAt" : "                    ",
    "创建开始" : "                    ",
    "创建结果" : "FailedMount",
    "调度时间" : "                    "
},
```


**Data Collect:**

Collect Audit log for the whole cluster.

**Data analysis:**

Analyze failure reason if pod is failed.

**Reason analysis:**

Analyze the failure reasons. Try to find something abnormal in the cluster.

**We can get:**

It is failed to deliver the pod，and the fail reason is FailedMount.





接下来，以一个 pod 创建失败案例为例，向大家展示下 tracing 系统的工作流程。

用户输入 pod uid 之后，tracing system通过 pod 索引，查找到 pod 对应生命周期分析记录，交付成功与否判定结果。当然，storage 存储的数据不仅为终端用户提供基础数据，更重要的是通过对集群内 pods 生命周期，分析出周期内集群的运营状况及每个节点的运营状况。比如说集群内太多 pods 调度到热点节点，不同 pods 的交付引起节点上资源竞争，导致节点负载太高，而交付能力却在下降，最终表现为节点上 pods 交付超时。再举个例子，通过历史统计数据，分析出 pods 生命周期中各阶段的执行时间基线，以基线为评估标准，比较组件不同版本的平均用时，用时分布，给出组件改进建议。另外，通过整体的 pods 生命周期中各组件负责的步骤时间占比，找出占比较多的步骤，为后续优化 pod 交付时间提供数据支持。

## Node Metrics

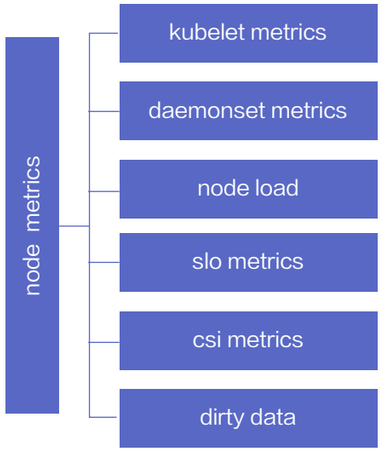
**With huge amount of metrics data collected, statistical methods can be used to check whether the node is healthy or not.**

**Besides, node delivery capacity can also be evaluated via historical data.**

With dirty data metrics which consists of

- escaped/zombie/uninterruptible process
- orphaned containers
- orphaned pod directories/volumes
- orphaned cgroups
- orphaned net device

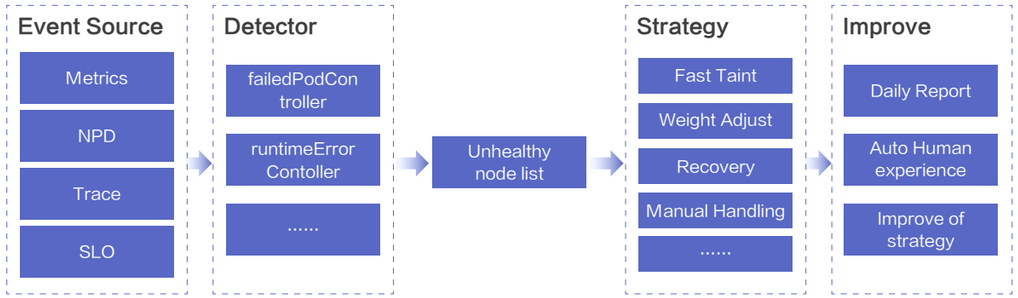
**and so on, node recovery system can cleanup those dirty data or alert cluster admins to process dirty data manually.**



一个运行状况良好的集群，不仅需要 master 组件保持高可用，节点稳定性也不容忽视。如果把 pod 创建比作是 rpc 调用，则每个节点就是一个 rpc 服务提供者，集群的总容量等于每个节点能处理的 pod 创建请求的总和。每多一个不可用的节点，都代表着集群交付能力的下降，也代表着集群可用资源的下降，这就要求尽量保证集群内节点保证高可用。每一次 pod 交付/删除/升级失败，也意味着用户使用成本上升，体验下降。这就要求集群节点保证良好的健康度，调度到节点上的 pods才能成功交付。换句话说，不仅要尽早发现节点异常，也要尽快修复节点。通过分析各组件在 pod 交付链路上的功能，我们补充了各种不同类型的组件的 metrics，以及将 host 运行状态转换为 metrics，一并采集到数据库之后，结合每个节点上 pod 交付结果，构建模型预测节点可用性，分析节点是否存在不可恢复异常，适当调整节点在调度器中比重，从而提升 pod 交付成功率。

Pod 创建/升级失败，用户可以通过重试来解决，但 pod 删除失败，虽然有着 k8s 面向终态的理念，组件会不断重试，但终究也会存在脏数据，如 pod 在 etcd 上删除，但是节点上还残留着脏数据。我们设计实现了一个巡检系统，通过查询 apiserver 获取调度到当前节点上的 pods，通过对比，找到节点上残留的进程/容器/ volumes 目录/cgroup /网络设备等，通过其他途径尝试释放残留资源。

## Alluxio 支持 AI 模型训练场景的挑战








1. Collect data from metrics **NPD**, Trace system and Log.
2. Analyze the problem of the node, such as DiskRO, critical Daemonset is not ready.
3. Processing unhealthy node: **Heal, Degrade or Isolate.** With scoring mechanism and historical operation records, unhealthy nodes can be recovered automatically. Otherwise manual intervention is required.
4. Generate daily report to show what happens, and programmers can improve the system with the reports continuously.

接下来描述故障机的处理流程。

故障机判断的数据来源有很多，主要有节点的监控指标，比如某类 Volume 挂载挂载失败；NPD(Node Problem Detector)，这是社区的一个框架；Trace 系统，比如某个节点上 Pod 创建持续报镜像下载失败；SLO，比如单机上残留大量 Pod。

我们计划多个 Controller 对这些某类故障进行巡检，形成故障机列表。一个故障机可以有好几项故障。对于故障机，会按照故障进行不同的操作。主要的操作有：打 Taint，防止 Pod 调度上去；降低 Node 的优先级；直接自动处理进行恢复。对于一些特殊原因，比如磁盘满，那就需要人工介入排查。

故障机系统每天都会产生一个日报，来表明故障机系统今天做了哪些事情。开发人员可以通过不断地添加 Controller 和处理规则完善整个故障机处理系统。

## Tips on increasing SLO

### Case 1: Image Download

Image lazyload technology provides the ability to run a container without downloading image.

### Case 2: Retry

Pod should be recreate when the previous pod is failed. The previous node should be excluded.

### Case 3: Critical Deamonset

Node should be tainted when critical Daemonset is unhealthy.

### Case 4: Plugin registry

Registration of plugin such as CSI plugin should be checked.

### Case 5: Capacity

The QPS Limit and Capacity Limit should be applied.

接下来，我们来分享下达到高 SLO 的一些方法。

第一点，在提升成功率的进程中，我们面临最大的问题就是镜像下载的问题。要知道，Pod 必须在规定时间内交付的，而镜像下载通常需要非常多的时间。为此，我们通过计算镜像下载时间，还专门设置了一个 ImagePullCostTime 的错误，即镜像下载时间太长，导致 Pod 无法按时交付。

还好，阿里镜像分发平台蜻蜓支持了 Image lazyload 技术，就也是支持远程镜像。在 Kubelet 创建容器时，不用再下载镜像。所以，这大大加速了 Pod 的交付速度。有关 Image lazyload 技术，大家可以看下阿里蜻蜓的分享。

第二点，对于提升单个 Pod 成功率，随着成功率的提升，难度也越来越难。可以引入一些 workload 进行重试。在蚂蚁，paas 平台会不断重试，直到 Pod 成功交付，或者超时。当然，在重试时，之前的失败的节点需要排除。

第三点，关键的 Daemonset 一定要进行检查，如果关键 Daemonset 缺失，而 把 Pod 调度上去，那非常容易出问题，从而影响创建/删除链路。这需要接入故障机体系。

第四点，很多 Plugin，如 CSI Plugin，是需要向 Kubelet 注册的。可能存在节点上一切正常，但向 Kubelet 注册的时候失败，这个节点同样无法提供 Pod 交付的服务，需要接入故障机体系。

最后一点，由于集群中的用户数量是非常多的。所以隔离非常重要。在权限隔离的基础上，还需要做到 QPS 隔离，及容量的隔离，防止一个用户的 Pod 把集群能力耗尽，从而保障其他用户的利益。





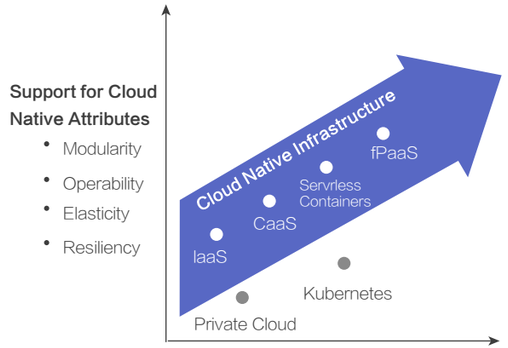
# 1/4 Knative Serverless 架构分析

阿里云技术专家　牛秋霖（冬岛）

作者简介：冬岛，就职于阿里云容器服务团队，本次分享的主题是《Knative Serverless 架构剖析》，本次分享从 Serverless 的发展历程开始讨论，到 Kubernetes 为首的云原生理念再到如何在 Kubernetes 之上构建云原生 Serverless 应用这三个阶段介绍 Knative 的价值，最后通过一个 Demo 演示如何用 Knative 管理 Serverless 应用。

## Serverless 万众期待

- Gartner 认为 Serverless 将在 2019-2022 年成熟，预计到 2020 将超过 20% 企业使用 Serverless 计算服务
- Forrester 报告显示 49% 的公司正在使用或在未来1年内将使用 Serverless 计算
- O’Reilly Serverless Survey 2019 1500 名受访者的调研结果显示 40% 的受访者所在公司已经在使用 Serverless 技术，超过三分之二的受访者认为他们的组织对 Serverless 的采用至少大部分是成功的
- Serverless 已经是万众期待，未来可期的状态



Serverless 已经是万众期待，未来可期的状态。各种调查报告显示企业及开发者已经在使用 Serverless 构建线上服务，而且这个比例还在不断增加。

在这个大趋势下咱们看一下 IaaS 架构的演进方向。最初企业上云都是基于 VM 的方式在使用云资源，企业线上服务都是通过 Ansible、Saltstack、Puppet 或者 Chef 等工具裸部在 VM 中的。直接在 VM 中启动应用，导致线上服务对 VM 的环境配置有很强的依赖，而后伴随着容器技术的崛起大家开始通过容器的方式在 VM 中部署应用。但如果有十几个甚至几十个应用需要部署，就需要在成百上千的 VM 快速部署、升级应用，这是一件非常头疼的事儿。而 Kubernetes 很好的解决了这些问题，所以现在大家开始通过 Kubernetes 方式使用云资源。随着 Kubernetes 的流行，现在各大云厂商都提供了 Serverless Kubernetes 服务，用户无需维护 Kubernetes 集群，即可直接通过 Kubernetes 语义使用云的能力。

## 为什么需要 Knative

既然 Kubernetes 已经很好了，为什么还需要 Knative 呢？要回答这个问题咱们先梳理一下 Serverless 应用都有哪些共同特质

- 自动弹性

按需使用云资源，业务量上涨的时候自动扩容，业务量下降的时候自动缩容，所以需要自动弹性能力

- 灰度发布

要能支持多版本管理，应用升级的时候可以使用各种灰度发布策略上线新的版本

- 流量管理

能够管理南北流量，可以按照流量百分比对不同版本进行灰度

- 负载均衡、服务发现

应用弹性过程中自动增加或者减少实例数量，流量管理需要具备负载均衡和服务发现的功能

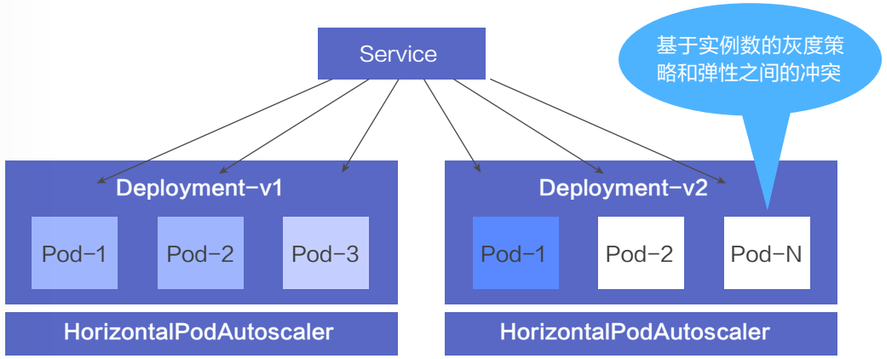
- Gateway

多个应用部署在同一个集群中，需要一个接入层网关对多个应用以及同一个应用的不同版本进行流量的管理



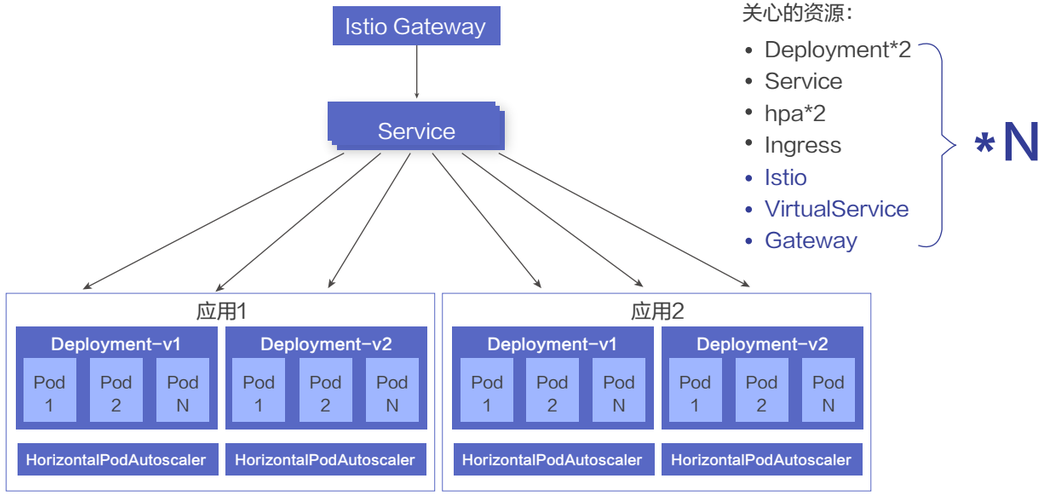


随着 Kubernetes 和云原生概念的崛起，第一直觉可能是直接在 Kubernetes 之上部署 Serverless 应用。那咱们来看一下，如果要在原生的 Kubernetes 上部署 Serverless 应用咱们可能会怎么做。




首先需要一个 Deployment 来管理 Workload，还需要通过 Service 对外暴露服务和实现服务发现的能力。应用有重大变更，新版本发布时可能需要暂停观察，待观察确认没问题之后再继续增加灰度的比例。这时候就要使用两个。

Deployment 才能做到。v1 Deployment 代表旧版本，灰度的时候逐一减少实例数，v2 Deployment 代表新版本，灰度的时候逐一增加实例数。hpa 代表弹性能力，每一个 Deployment 都有一个 hpa 管理弹性配置。这里面其实是有冲突的：假设 v1 Deployment 原本有三个 pod，灰度的时候升级一个 pod 到 v2，此时其实是 1/3 的流量会打到 v2 版本上。但当业务高峰上来后，因为两个版本都配置了 hpa，所以 v2 和 v1 会同时扩容，最终 v1 和 v2 的 pod 数量就不是最初设置的 1/3 的比例了。所以传统的这种按照 Deployment 实例数发布的灰度策略和弹性配置天然是冲突的。而如果按照流量比例进行灰度就不会有这个问题，要安装流量比例进行灰度可能就要引入 Istio 的能力。



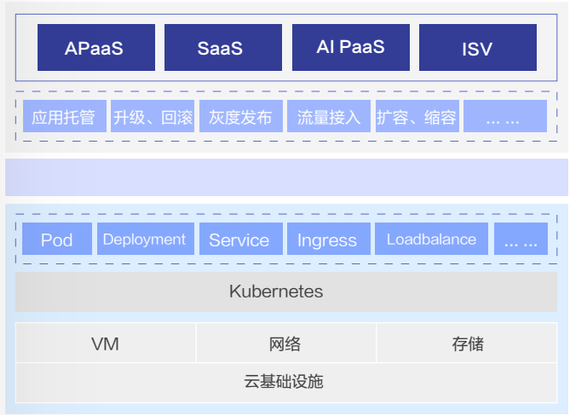

引入 Istio 作为 Gateway 组件，Istio 除了管理同一个应用的流量灰度，还能对不同的应用进行流量管理看起来很好，咱们再仔细分析一下看看存在什么问题。咱们现在梳理一下在原生 K8s 之上手动管理 Serverless 应用都需要做什么：

· Deployment · Service · HPA · Ingress · Istio · VirtualService · Gateway

这些资源是每一个应用维护一份，如果是多个应用就要维护多分。这些资源散落在 K8s 内，根本看不出来应用的概念，其实管理起来也是非常的繁琐。








### 现状

- 用户使用云正在向面向服务的方式转变，弹性越来越重要
- 通过 k8s API 实现服务全生命周期管理比较复杂
- 需要面向 Serverless 应用的抽象，而不是面向底层资源的抽象

### 提出问题?

如何才能让用户以及上层 PaaS 平台以面向服务的方式使用云的能力?

Serverless 应用需要的是面向应用的管理动作，比如应用托管、升级、回滚、灰度发布、流量管理以及弹性等功能。而 Kubernetes 提供的是 IaaS 的使用抽象。所以 Kubernetes 和 Serverless 应用之间少了一层应用编排的抽象。而 Knative 就是建立在 Kubernetes 之上的 Serverless 应用编排框架。除了 Knative 以外，社区也有好几款 FaaS 类的编排框架，但这些框架编排出来的应用没有统一的标准，每一个框架都有一套自己的规范，而且和 Kubernetes API 完全不兼容。不兼容的 API 就导致使用难度高，可复制性不强。云原生的一个核心标准就是 Kubernetes 的 API 标准，Knative 管理的 Serverless 应用保持 Kubernetes API 语义不变。和 Kubernetes API 良好的兼容性这就是 Knative 的云原生特性所在。

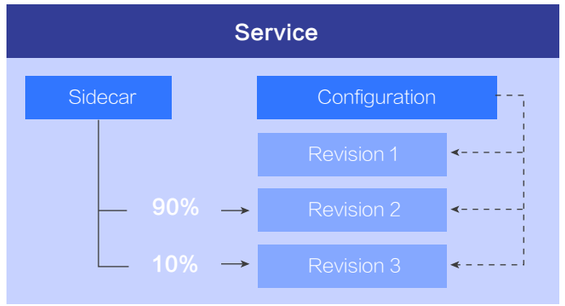
## Knative 是什么

**Kubernetes**-based platform to **build, deploy,**
and **manage** modern **serverless** workloads.

基于 **Kubernetes** 平台，用于**构建、部署**和
管理现代 **Serverless** 工作负载

Knative 主要解决的问题域就是在 Kubernetes 之上提供通用的 Serverless 编排、调度服务，给上层的 Serverless 应用提供面向应用层的原子操作。并且通过 Kubernetes 原生 API 暴露服务 API，保持和 Kubernetes 生态工具链的完美融合。Knative 有 Eventing 和 Serving 两个核心模块，今天就主要介绍一 Serving 的核心架构。

## Knative Serving 架构




**Service**
对应用 Serverless 编排的抽象，通过 Service 管理应用的生命周期

**Configuration**
当前期望状态的配置。每次更新 Service 就会更新 Configuration

**Revision**
Service 的每次更新都会创建一个快照，一个快照就是一个 Revision

**Route**
将请求路由到 Revision，并可以向不同的 Revision 转发不同比例的流量





Serving 核心是 Knative Service，Knative Controller 通过 Service 的配置自动操作 Kubernetes Service 和 Deployment 从而实现简化应用管理的目标。

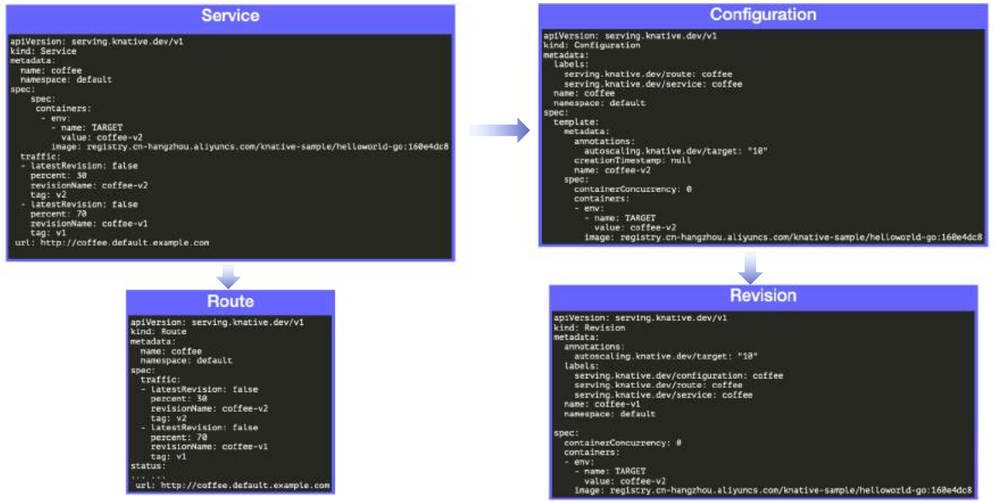
Knative Service 对应一个叫做 Configuration 的资源，每次 Service 变化如果需要创建新的 Workload 就更新 Configuration，然后每次 Configuration 更更新都会创建一个唯一的 Revision。Revision 可以认为是 Configuration 的版本管理机制。理论上Revision 创建完以后就不会修改的。

Route 主要负责 Knative 的流量管理，Knative Route Controller 通过 Route 的配置自动生成 Knative Ingress 配置，Ingress Controller 基于 Ingress 策略实现路由的管理。

Knative Serving 对应用 Workload 的 Serverless 编排是从流量开始的。流量首先达到 Knative 的 Gateway，Gateway 根据 Route 的配置自动把流量根据百分比拆分到不同的 Revision 上，然后每一个 Revision 都有一个自己独立的弹性策略。当过来的流量请求变多的时候当前 Revision 就开始自动扩容。每一个 Revision 的扩容策略都是独立的，相互不影响。

基于流量百分对对不同的 Revision 进行灰度，每一个 Revision 都有一个自己独立的弹性策略。Knative Serving 通过对流量的控制实现了流量管理、弹性和灰度三者的完美结合。接下来具体介绍一下 Knative Serving API 细节。

## Knaive Serving API

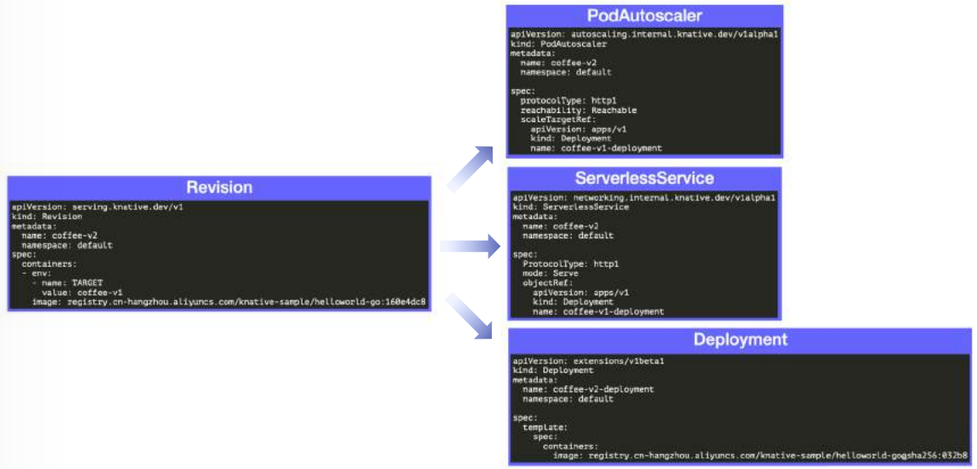



这是 Knative Service CR 关系示意图。Knative Service 的变动会生成 Configuration，Configuration 的每次变动会生成一个对应的 Revision。Knative Service 除了生成 Configuration 以外还会对应一个唯一的 Route，Route 负责流量管理。





## Revision API

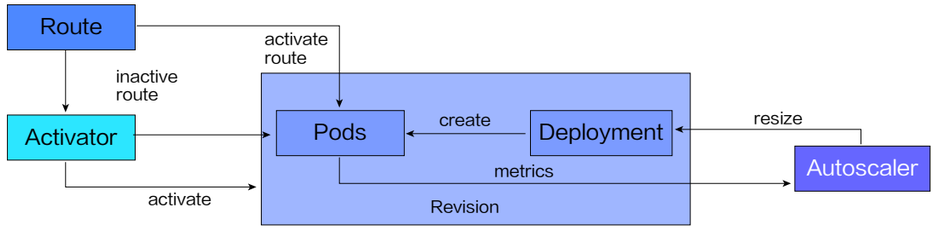



Revision 代表一次版本发布，每次版本都对应一个 PodAutoscaler、ServerlessService 和 Deployment 三个资源。PodAutoscaler 主要是弹性策略的配置，业务量高峰的时候自动扩容，低谷的时候自动缩容。ServerlessService 主要服务发现，管理 pod EndPoint。Deployment 是 Workload 管理，弹性扩容或者缩容都是通过修改 Deployment 副本数实现的。

## Route API


Route 负责流量管理，Route 不直接管理流量，是通过 Ingress 实现了解耦。Ingress 需要 Ingress Controller 来具体实现流量管理的功能。社区默认的 Ingress Controller 是基于 Istio 实现的，其实 Istio 只是其中一种，除了 Istio 还有 Ambassador、Contour、Gloo、Kong 以及 Kourier 等 Gateway。如果在云上和云服务 ALB 结合也许是更好的选择。

## Autoscaler 详解








上图展示了 Knative Autoscaler 的工作机制，Route 负责接入流量，Autoscaler 负责做弹性伸缩。当没有业务请求的时候会缩容到零，缩容到零后 Route 进来的请求会转到 Activator 上。当第一个请求进来之后 Activator 会保持住 http 链接，然后通知 Autoscaler 去做扩容。Autoscaler 把第一个 pod 扩容完成以后 Activator 就把流量转发到 Pod ，从而做到了缩容到零也不会损失流量的目的。

到此 Knative Serving 的核心模块和基本原理都介绍完了，你应该对 Knative 已经有了初步的了解。在介绍原理的过程中你可能也感受到了，要想把 Knative 用起来其实还是需要维护很多 Controller 组件、Gateway 组件 (比如 Istio) 的，而要做到这些使用持续之处 IaaS 成本和运维成本。

## Knative 运维难度

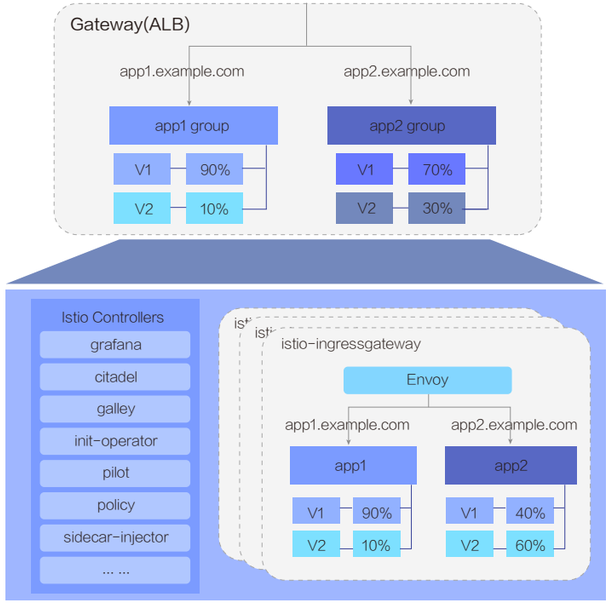
- Gateway 成本
- Knative Controller 成本
- 缩容到零的冷启

Gateway 组件假设使用 istio 实现的话，Istio 本身就需要十几个 Controller，如果要做高可用可能就二十几个 Controller 了。Knative Serving Controller 全都高可用部署也需要十几个。这些 Controller 的 IaaS 成本和运维成本都比较多。另外冷启动问题也很明显，虽然缩容到零可以降低业务波谷的成本，都是第一批流量也可能会超时。

## Knative 和云的完美结合

为了解决上述问题，我们把 Knative 和阿里云做了深度的融合。用户还是按照 Knative 的原生语义使用，但底层的 Controller 、Gateway 都深度的嵌入到阿里云体系内。这样既保证了用户可以无厂商锁定风险的以 Knative API 使用云资源，还能享受到阿里云基础设施的已有优势。

## Gateway 和云的融合




**降成本：**减少了十几个组件，大大降低运维成本和 IaaS 成本

**更稳定：**SLB 云产品服务更稳定、可靠性更高，易用性也更好

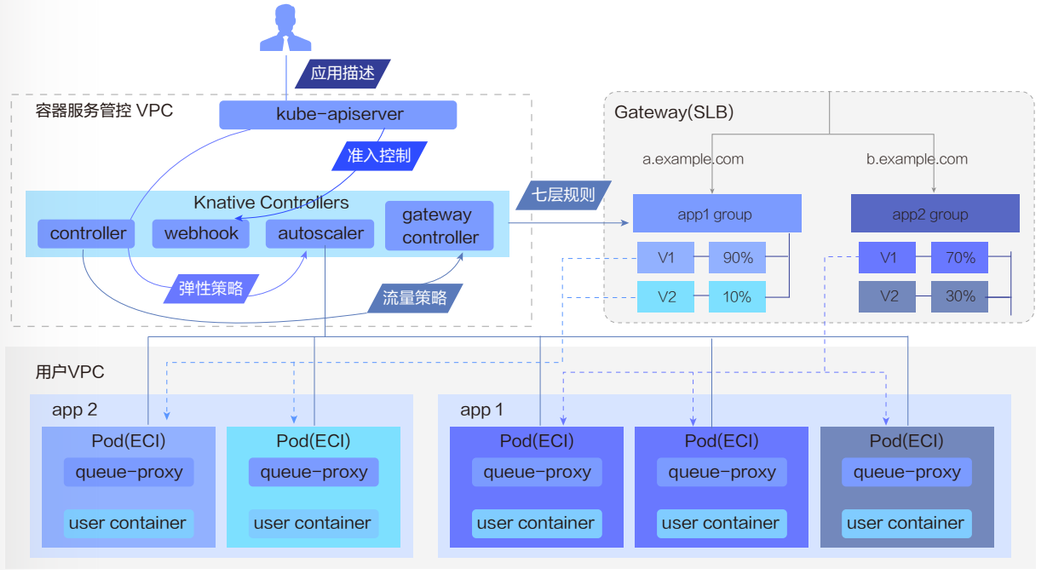
首先是 Gateway 和云的融合。直接使用阿里云 SLB 作为 Gateway。使用云产品 SLB 的好处有：

- 云产品级别的支撑，提供 SLA 保障
- 按需付费，不需要出 IaaS 资源
- 用户无需承担运维成本，不用考虑高可用问题，云产品自带高可用能力





## 管控组件下沉



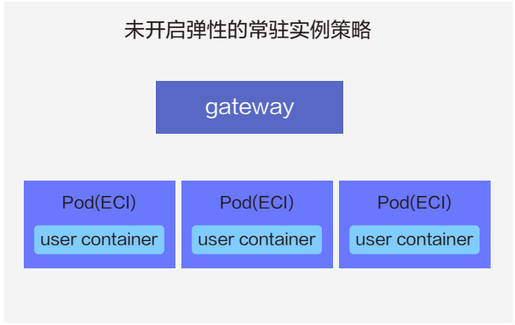

**开箱即用：**用户直接使用 Serverless Framework，不需要自己安装

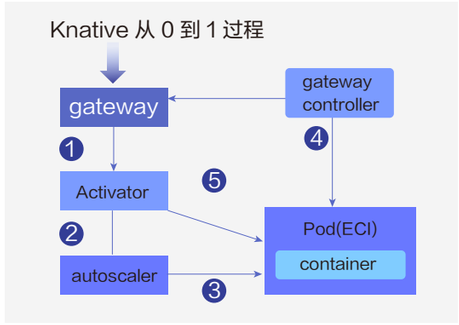
**免运维、低成本：**Knative 组件和 K8s 集群进行融合，用户没有运维负担，也无需承担额外的资源成本

**高管控：**所有组件都在管控端部署，升级和迭代更容易

除了 Gateway 组件以外，Knative Serving Controller 也需要一定的成本，所以我们把 Knative Serving Controller 和阿里云容器服务也进行了融合。用户只需要拥有一个 Serverless Kubernetes 集群并开通 Knative 功能就可以基于 Knative API 使用云的能力，并且用户无需为 Knative Controller 出任何成本。

## 弹性效率&成本



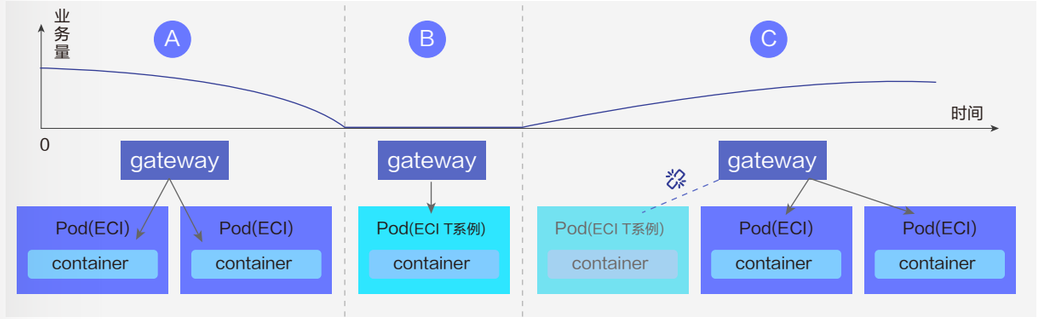



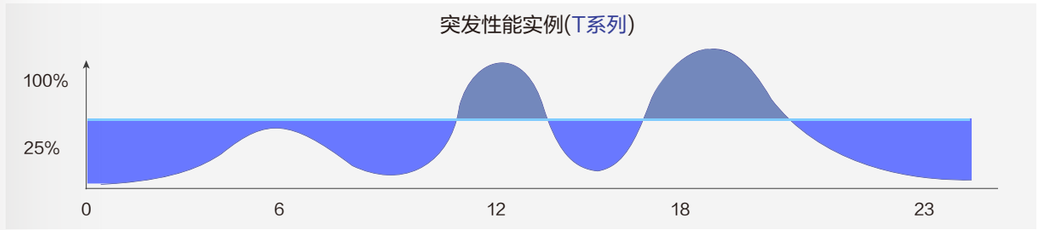


接下来咱们再分析一下冷启动问题。传统应用在没开启弹性配置的时候实例数是固定的，Knative 管理的 Serverless 应用默认就有弹性策略，在没有流量的时候会缩容到零。传统应用在流量低谷的时候即便没有业务请求处理，实例数还是保持不变，这其实是浪费资源的。但好处就是请求不会超时，什么时候过来的请求都可以会很好的处理。而如果缩容到零，第一个请求到达以后才会触发扩容的过程。Knative 的模型中从零到一扩容需要 5 个步骤串行进行，这 5 个步骤都完成以后才能开始处理第一个请求，而此时往往都会超时。所以 Knative 缩容到零虽然降低了常驻资源的成本，但第一批请求的冷启动问题也非常明显。可见弹性其实就是在寻找成本和效率的一个平衡点。








**免冷启动：**通过保留规格消除了从 0 到 1 的 30 秒冷启动时间

**成本可控：**突发性能实例成本比标准规格实例降低 40% 的成本，如果和 Spot 实例结合还能再进一步降低成本

为了解决第一个实例的冷启动问题，我们推出了保留实例的功能。保留实例是阿里云容器服务 Knative 独有的功能。社区的 Knative 默认在没有流量的时候缩容到零，但是缩容到零之后从零到一的冷启动问题很难解决。冷启动除了要解决 IaaS 资源的分配、Kubernetes 的调度、拉镜像等问题以外还涉及到应用的启动时长。应用启动时长从毫秒到分钟级别都有。应用启动时间这完全是业务行为，在底层平台层面几乎无法控制。

ASK Knative 对这个问题的解法是通过低价格的保留实例来平衡成本和冷启动问题。阿里云 ECI 有很多规格，不同规格的计算能力不一样价格也不一样。如下所示是对 2c4G 配置的计算型实例和突发性能型实例的价格对比。

| 规格族 | 实例规格 | VCPU | 内存 | 参考价格 元/时 |
| --- | --- | --- | --- | --- |
| 计算型 c5 | ecs.c5.large | 2 vCPU | 4 GiB | 0.62 |
| 突发性能实例 t5 | ecs.t5-lc1m2.large | 2 vCPU | 4 GiB | 0.333 |
| 计算型 c6 | ecs.c6.large | 2 vCPU | 4 GiB | 0.39 |
| 突发性能实例 t6 | ecs.t6-c1m2.large | 2 vCPU | 4 GiB | 0.236 |

通过上面的对比可知突发性能实例比计算型便宜 46%，可见如果在没有流量的时候使用突发性能实例提供服务不单单解决了冷启动的问题还能节省很多成本。

突发性能实例除了价格优势以外还有一个非常亮眼的功能就是 CPU 积分。突发性能实例可以利用 CPU 积分应对突发性能需求。突发性能实例可以持续获得 CPU 积分，在性能无法满足负载要求时，可以通过消耗积累的 CPU 积分无缝提高计算性能，不会影响部署在实例上的环境和应用。通过 CPU 积分，您可以从整体业务角度分配计算资源，将业务平峰期剩余的计算能力无缝转移到高峰期使用(简单的理解就是油电混动啦⊠⊠)。突发性能实例的更多细节参见这里。

所以 ASK Knative 的策略是在业务波谷时使用突发性能实例替换标准的计算型实例，当第一个请求来临时再无缝切换到标准的计算型实例。这样可以帮助您降低流量低谷的成本，并且在低谷时获得的 CPU 积分还能在业务高峰到来时消费掉，您支付的每一分钱都没有浪费。

使用突发性能实例作为保留实例只是默认策略，您可以指定自己期望的其他类型实例作为保留实例的规格。当然您也可以指定最小保留一个标准实例，从而关闭保留实例的功能。





更多云 Knative 实践的案例请移步这里：https://knative-sample.com/40-cloud-native-practice/

**最后才能通过视频体验一下通过 Knative 管理 Serverless 应用的便利性。**

（扫码观看动手实践）

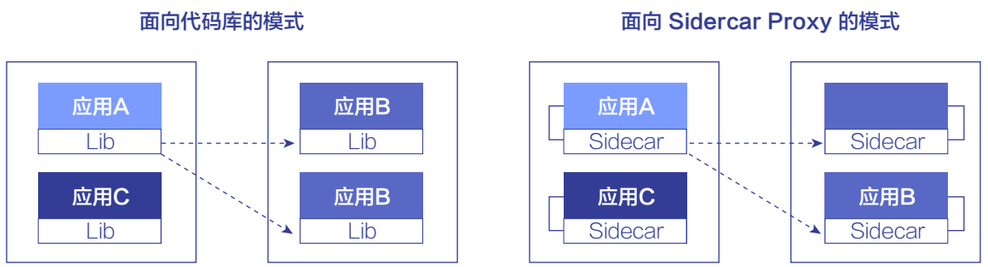
Knative 钉钉群

# 1/5 托管式服务网格：多种类型计算服务统一管理的基础设施

阿里云高级技术专家　王夕宁

在服务网格技术使用之前，为了更快更灵活地进行业务创新，我们常常会把现有应用进行现代化改造，把单体应用程序分拆为分布式的微服务架构。通常来说，在微服务架构模式的变迁过程中，最初都是面向代码库的模式。

## 架构模式的变迁



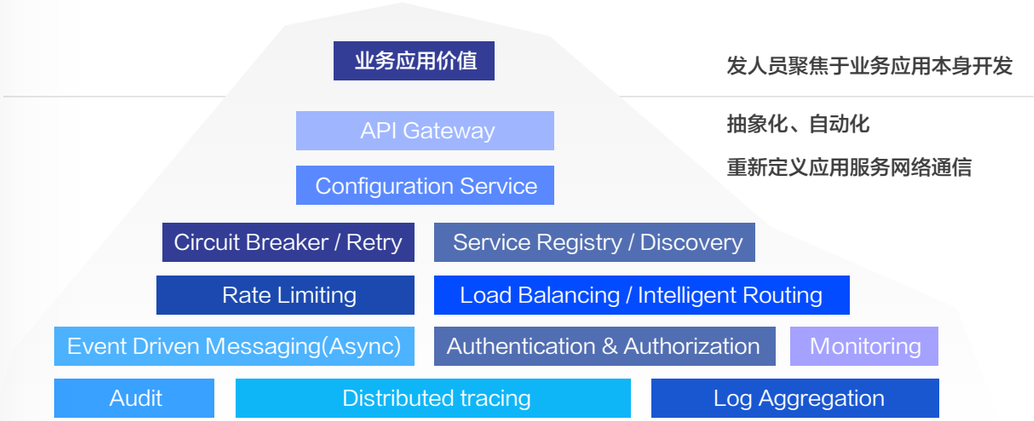





对这些微服务治理的实现，往往是以代码库的方式把这些服务治理的逻辑构建在应用程序本身中，这些代码库中包括了流量管理、熔断、重试、客户端负载均衡、安全以及可观测性等这样的一些功能。这些代码库随着功能的不断增强，版本也随之变更，因为版本不同导致的冲突问题处处可见。此外，库的版本一旦变更，即使你的应用逻辑并没有任何变化，整个应用也要随之全部变更。由此可见，随着应用的增长和团队数量的增加，跨服务一致地使用服务治理功能会变得非常复杂。

而通过把这些服务治理的能力 Sidecar 化，就能够把服务治理的能力与应用程序本身进行解耦，可以较好地支持多种编程语言、同时这些 Sidecar 能力不需要依赖于某种特定技术框架。这就是我们常说的面向 sidecar proxy 的架构模式。

## 抽象应用服务通信能力到基础设施

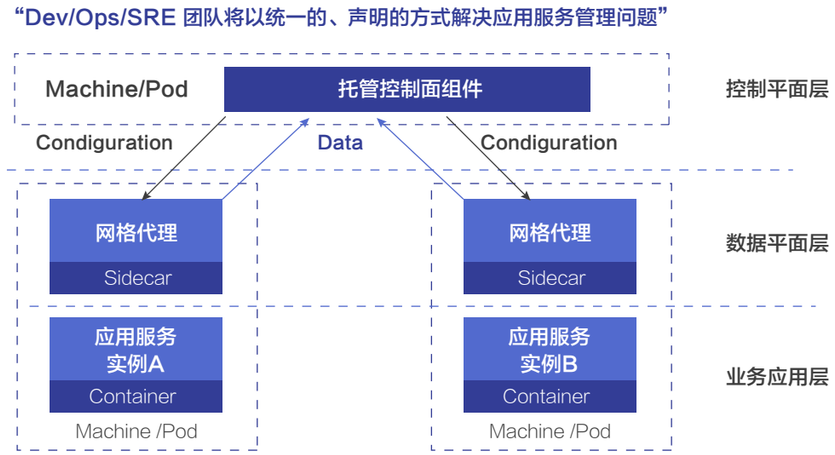


随着这些 Sidecar 代理功能的增强，原本需要在代码库中实现的服务治理功能被抽象化为一个个通用组件，并被逐渐地下沉到代理中。这些服务治理能力的标准化、统一化，可以解决复杂系统微服务实现中面临的差异大、缺少共性的问题，可以很好地支持不同的编程语言、不同的框架。

通过把应用服务通信能力抽象下沉到基础设施， 使得开发人员可以更加聚焦于业务应用本身开发，而让基础设施来提供这些通用的能力。

与此同时， 容器编排技术的更加成熟，也加速了 Sidecar 代理的普及与使用的便捷。Kubernetes 作为一个出色的容器部署和管理平台、Istio 作为应用服务治理的平台，两者的结合成为了将这些应用服务通信能力下沉到基础设施的载体。

## 启用服务网格



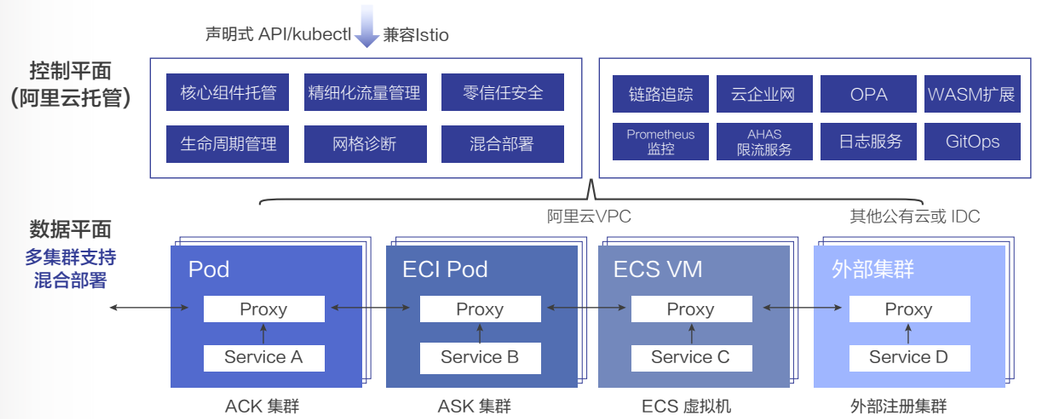





在云原生应用模型中，一个应用程序可能会包含数百个服务，每个服务又有数百个实例构成，那么这些成百上千个应用程序的 Sidecar 代理如何统一管理，这就是服务网格中定义的控制平面部分要解决的问题。作为代理，Envoy 非常适合服务网格的场景，但要发挥 Envoy 的最大价值，就需要使它很好地与底层基础设施或组件紧密配合。

这些Sidecar代理形成一个网状的数据平面，通过该数据平面处理和观察所有服务间的流量。数据平面扮演了一个用来建立、保护和控制通过网格的流量的角色。负责数据平面如何执行的管理组件称为控制平面。控制平面是服务网格的大脑，并为网格使用人员提供公开 API，以便较容易地操纵网络行为。启用服务网格之后，开发人员、运维人员以及 SRE 团队将以统一的、声明的方式解决应用服务管理问题。

## 托管的服务网格ASM产品架构



图中是服务网格 ASM 产品当前的架构。作为业内首个全托管Istio兼容的服务网格产品 ASM，一开始从架构上就保持了与社区、业界趋势的一致性，控制平面的组件托管在阿里云侧，与数据面侧的用户集群独立。ASM 产品是基于社区开源的 Istio 定制实现的，在托管的控制面侧提供了用于支撑精细化的流量管理和安全管理的组件能力。通过托管模式，解耦了 Istio 组件与所管理的K8s集群的生命周期管理，使得架构更加灵活，提升了系统的可伸缩性。

在深入分析服务网格方面，提供了网格诊断能力，把过去一年多来客户遇到的问题以及如何解决这些问题的手段变成产品能力，帮助用户快速定位遇到的问题；在扩展与集成方面，ASM产品整合阿里云服务包括可观测性服务链路追踪/日志服务/ Prometheus 监控等、跨 VPC 网络互连 CEN 能力等，同时也优化整合了社区开源软件包括 OPA 的支持、授权服务的定制化能力、限流服务等。

此外，随着Isito新架构的优化，将 WebAssembly 技术引入服务网格，解决代理扩展的问题。 这样一来，ASM 架构就变成了“托管的高可用弹性控制平面 + 可扩展的插件式的数据平面“的模式。

在数据平面的支持上，ASM 产品可以支持多种不同的计算基础设施，这包括了阿里云提供的公有云 ACK 集群（其中包括了托管的 K8s 集群和专有 K8s 集群）、也包括对无服务器 Kubernetes 容器服务 ASK 集群的支持。同时，对非容器化应用，例如运行在ECS虚拟机上的应用服务网格化的支持。

此外，ASM 也推出了一个支持多云混合云的能力，能够针对外部的非阿里云 K8s 集群进行支持，不论这个集群是在用户自建的 IDC 机房，还是在其他的公有云之上，都可以通过 ASM 进行统一的服务治理。

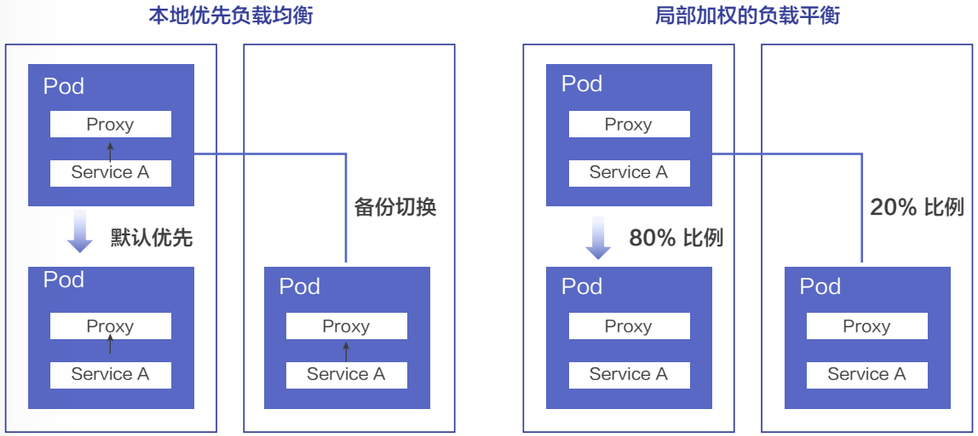
接下来，将介绍托管式服务网格在成为多种类型计算服务统一管理的基础设施中，如何提供了统一的流量管理能力、统一的服务安全能力、统一的服务可观测性能力、以及如何基于 WebAssembly 实现统一的代理可扩展能力。





## KubeNode – Machine Operator

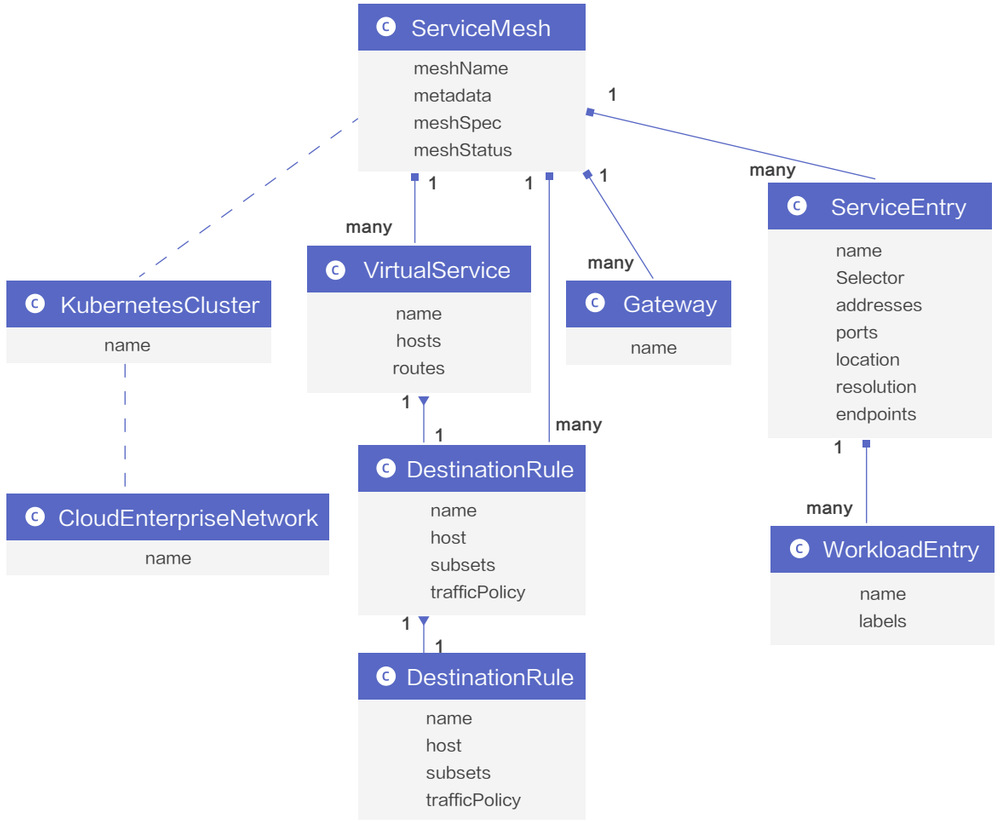
**相比于 Kubernetes 默认提供的轮询方式进行负载均衡， 服务网格技术提供的基于位置的路由能力，可以将流量路由到最靠近的容器。这样可以确保服务调用的低延迟，并能尽量保持在同一区域内降低流量费用。**




首先，关于统一的流量管理能力方面，重点讲述2个方面。

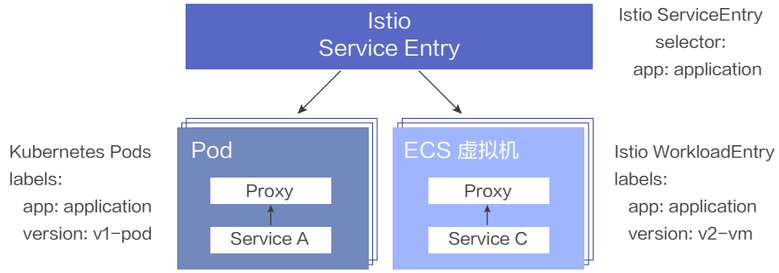
第一个方面，是基于位置实现流量路由请求。在大规模服务场景下，成千上万个服务运行在不同地域的多种类型计算设施上，这些服务需要相互调用来完成完整的功能。为了确保获得最佳性能，应当将流量路由到最近的服务，使得流量尽可能在同一个区域内,而不是只依赖于 Kubernetes 默认提供的轮询方式进行负载均衡。服务网格应当提供这样的基于位置的路由能力，一方面，可以将流量路由到最靠近的容器，实现本地优先的负载均衡能力，并在主服务出现故障时可以切换到备用服务。另一方面，提供局部加权的负载平衡能力，能够根据实际需要，将流量按比例拆分到不同的地域。

## 非 K8s 工作负载的网格化统一管理








第二个方面，是关于非 K8s 工作负载的网格化统一管理。在一个托管的服务网格实例中，我们可以添加若干个 K8s 集群，并在控制面定义路由规则的配置，也可以定义网关服务等。为了能够统一地管理非K8s工作负载，我们通过一个 WorkloadEntry 的 CRD 来定义工作负载的标签以及该工作负载运行的 IP 地址等信息。然后通过 ServiceEntry CRD 将这个工作负载注册到服务网格中，并提供类似于 K8s Pod 的处理机制来处理这些非 K8s 工作负载。譬如可以通过 selector 机制路由到对应的 Pod 或者这个非容器应用上。

## 统一的服务安全与审计

| TLS/JWT认证 | RBAC授权 | 审计 | 策略管理 |
| --- | --- | --- | --- |
| • 以渐进方式逐步实现 mTLS<br>• 支持细粒度，包括 namespace 与 workload 级别<br>• 简单易用 JWT 认证 | • 灵活设置RBAC规则<br>• 支持细粒度，包括 namespace /service/port 级别 | • 灵活开启网格审计功能<br>• 查看审计报表<br>• 查看日志记录<br>• 设置告警规则 | • 开放策略代理（OPA）<br>• 自定义授权服务 external_auth grpc |

主子账户支持/RAM授权支持

关于统一的服务安全能力,托管服务网格为多种不同计算基础设施上的应用服务提供统一的主子账户支持/RAM 授权支持。在此基础上,提供统一的 TLS 认证与 JWT 认证,支持启用与禁用 TLS 认证的简易切换、支持以渐进方式逐步实现双向 TLS 认证;支持以细粒度的认证范围,包括namespace与 workload 级别。此外,服务网格也提供对 JWT 认证能力的支持,使得这种 TOKEN 认证机制不再依赖于某种特定实现框架就可以统一透出。

在 RBAC 授权方面,针对不同协议提供了统一的授权策略,可以在不同粒度上支持,包括 namespace/service/port 级别的授权。在审计方面,可以灵活开启网格审计功能,并可以查看审计报表、查看日志记录以及设置告警规则等。在策略管理方面,提供了开放策略代理(OPA)的集成, 用户可以使用描述性策略语言定义相应的安全策略。 此外也提供了自定义授权服务 external_auth grpc 的对接。只要遵循这一接口定义,任意授权服务都可以集成到服务网格中。

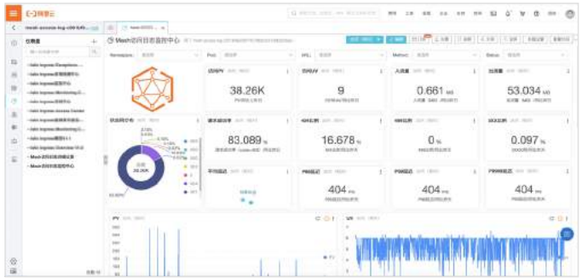
统一的服务可观测性,分为3个方面。

第一个可观测性能力是日志分析,通过对数据平面的 AccessLog 采集分析,特别是对入口网关日志的分析,可以分析出服务请求的流量情况、状态码比例等,从而可以进一步优化这些服务间的调用。

## 统一的服务可观测性 - 日志分析

**日志**

- 数据平面的AccessLog
- 入口网关日志
- 日志分析报表





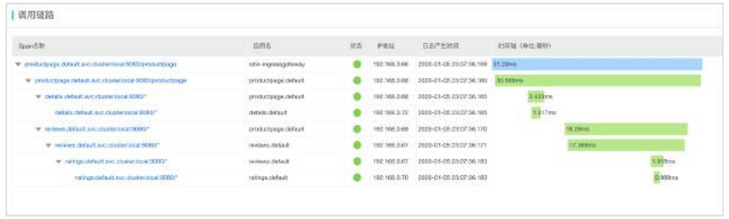



第二个可观测性能力是分布式追踪能力。 为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等工具,可以帮助开发者快速分析和诊断分布式应用架构下的性能瓶颈,提高微服务时代下的开发诊断效率。

## 统一的服务可观测性 - 追踪

**追踪**

- 简易使用阿里云链路追踪
- 集成自定义 zipkin 服务

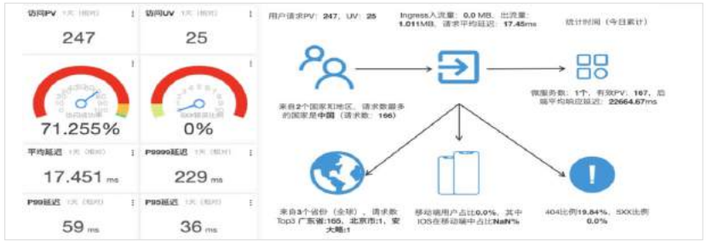



第三个可观测性能力是监控能力。根据监视的四个维度(延迟,流量,错误和饱和度)生成一组服务指标, 来了解、监视网格中服务的行为。

## 统一的服务可观测性 - 监控

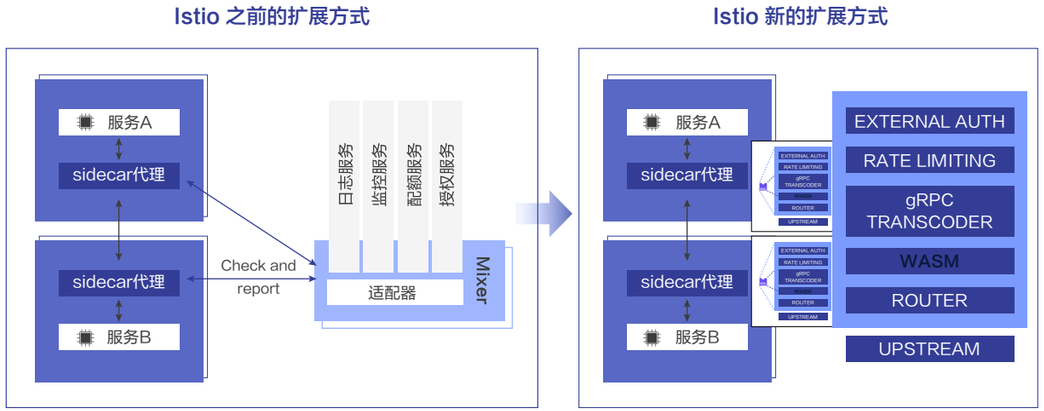
**监控**

- 4个维度:延迟、流量、错误和饱和度
- 简易使用阿里云托管 Prometheus
- 集成社区可观性 Grafana/Kiali 等



关于 Sidecar 代理的可扩展性。尽管 Sidecar 代理已经把服务治理过程中常用的一些功能进行了封装实现，但它的可扩展能力一定是必须具备的，譬如如何与已有的后端系统做对接，如何解决用户的一些特定需求。这个时候，一个 Sidecar 代理的可扩展性显得尤为重要，而且在一定程度上会影响 Sidecar 代理的普及。

## 统一的基于WASM的代理可扩展能力




在 Istio 之前的架构中，对 Sidecar 能力的扩展主要集中在Mixer组件上。Sidecar 代理的每个服务到服务连接都需要连接到 Mixer，以进行指标报告和授权检查，这样会导致服务之间的调用延迟更长，伸缩性也变差。同时，Envoy 要求使用代理程序的编程语言 C++编写，然后编译为代理二进制文件。对于大多 Istio 用户而言，这种扩展能力具有一定的挑战性。





然而，在采用了新架构之后，Istio 把代理的扩展能力从Mixer下移到了数据平面的 Envoy 本身中，并使用 WebAssembly 技术将其扩展模型与 Envoy 进行了合并。WebAssembly 支持几种不同语言的开发，然后将扩展编译为可移植字节码格式。这种扩展方式既简化了向 Istio 添加自定义功能的过程，又通过将决策过程转移到代理中而不是将其种植到 Mixer 上来减少了延迟。使用 WebAssembly（WASM）实现过滤器Filter的扩展, 可以获得以下好处：

**敏捷性：**过滤器Filter可以动态加载到正在运行的Envoy进程中，而无需停止或重新编译。

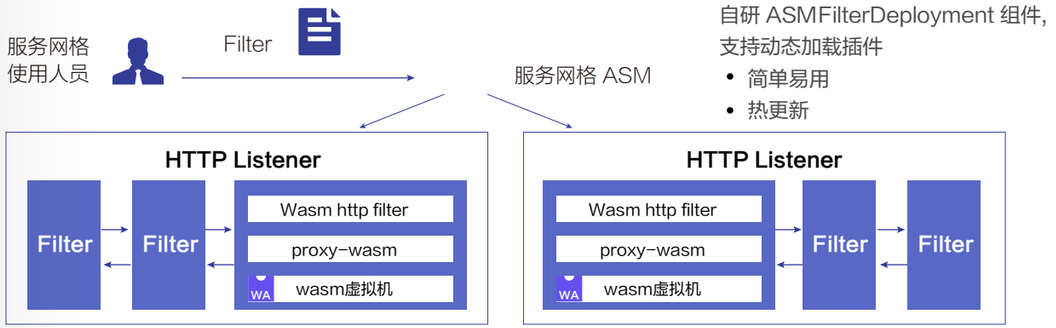
**可维护性：**不必更改Envoy自身基础代码库即可扩展其功能。

**多样性：**可以将流行的编程语言（例如C / C ++和Rust）编译为 WASM，因此开发人员可以使用他们选择的编程语言来实现过滤器Filter。

**可靠性和隔离性：**过滤器Filter会被部署到VM沙箱中，因此与 Envoy 进程本身是隔离的；即使当 WASM Filter 出现问题导致崩溃时，它也不会影响 Envoy 进程。

**安全性：**过滤器Filter通过预定义 API 与 Envoy 代理进行通信，因此它们可以访问并只能修改有限数量的连接或请求属性。

## 统一的代理扩展插件的生命周期管理



阿里云服务网格 ASM 产品中提供了对 WebAssembly（WASM）技术的支持，服务网格使用人员可以把扩展的 WASM Filter 通过 ASM 部署到数据面集群中相应的 Envoy 代理中。通过自研的 ASMFilterDeployment 组件，可以支持动态加载插件、简单易用、以及支持热更新等能力。通过这种过滤器扩展机制，可以轻松扩展 Envoy 的功能并将其在服务网格中的应用推向了新的高度。

## 服务网格实践之成熟度模型

| 01 | 02 | 03 | 04 | 05 |
|---|---|---|---|---|
| **一键启用** | **可观测提升** | **安全加固** | **多种基础设施** | **多集群混合管理** |
| 流量管理<br>灰度发布<br>入口网关 | 日志采集与分析<br>分布式追踪<br>监控指标采集与分析 | mTLS 认证<br>Identity 身份识别 | 任意的 Kubernetes 集群<br>ECS 虚拟机非容器应用<br>裸机服务器非容器应用 | 多集群跨地域<br>非容器化应用 |

服务网格作为应用服务通信的统一基础设施，可以（并且应该）逐步采用。 在此，我们推出了它的实践之成熟度模型，分为了 5 个层次，分别为一键启用、可观测提升、安全加固、多种基础设施的支持，以及多集群混合管理。这 5 个方面分别涵盖了前面讲述的统一流量管理、统一可观测性、统一服务安全以及支持不同的计算基础设施和多集群非容器化应用的混合管理。





# 2/1 KubeNode：阿里巴巴云原生容器基础设施运维实践

**阿里云技术专家　周涛**

作者简介：2017 年加入的阿里，过去三年一直负责阿里巴巴数十万规模集群节点管控系统的研发工作，参与了历年的双十一大促。随着 2018 年底开始的集团上云项目，管理的节点从云下的物理机覆盖到了阿里云公有云上的神龙裸金属服务器，并支撑双 11 大促实现了交易核心系统的全面云原生化改造。

## 阿里巴巴节点运维的挑战

**规模大**

- 数百ASI 集群
  （Ali Serverless Infra,ACK+Ali addon）
- 数十万节点
  （单集群节点最多~10K 台）
- 数万应用
- 数百万容器

**环境复杂**

- X86 / ARM / GPU / FPGA
- 在线（应用类型差异大）、混部、安全容器

**环境复杂**

- X86 / ARM / GPU / FPGA
- 在线（应用类型差异大）、混部、安全容器

**稳定性要求高**

- 在线业务延迟、抖动敏感
- 宕机、夯机业务无感知

**稳定性要求高**

- 在线业务延迟、抖动敏感
- 宕机、夯机业务无感知

在阿里巴巴的场景下，做节点运维面临的挑战主要来自于这几个方面：规模、复杂性、稳定性。

首先是规模大，从 18 年第一个集群的搭建，到现在线上共运行着数百个 ASI 集群、数十万个节点，其中单集群的节点数最多有超过1万台。在这之上，运行着经济体数万个不同的应用，比如，大家熟知的淘宝、天猫等，总容器实例数有数百万规 aba Serverless Infrastructure）也就是阿里巴巴 Serverless 基设设施。ASI 的概念同时包含集群的管控面和节点两方面。每一个 ASI 集群都是调用 Aliyun 标准的 OpenAPI 创建的 **ACK 托管版集群**，然后在此之上，我们自研了调度器、开发部署了很多的 addon，进行功能增强、性能优化，并对接集团的各个系统，并做到节点全托管，不需要应用开发运维人员关心底层的容器基础设施。

其次是环境非常复杂。目前 IaaS 层运行着很多种异构的机型，有x86服务器，也有 ARM 国产化机型，同时为了服务新计算、AI 的业务，也有 GPU、FPGA 的机型。线上的内核版本也非常多，4.19 是今年开始规模上线的内核版本，同时 3.10 / 4.9 内核版本上的节点问题也需要继续支撑，不同的内核版本演进也需要具备规模化轮转的运维能力。目前上万个在线应用包含了整个阿里经济体像淘宝、天猫、菜鸟、高德、饿了么、考拉、盒马 等各种不同的业务，同时跟在线应用一起运行着混部、安全容器的业务，像大数据、离线计算、实时计算、搜索等这些业务类型也都跟在线业务运行在同一个主机环境中。

最后是对稳定性的高要求。在线业务的特点是对延迟和抖动非常敏感，单节点的抖动、夯机、宕机等故障都可能会影响某个用户在淘宝的下单付款，引发用户的吐槽和投诉，所以整体对稳定性的要求非常高，要求对单节点故障的处理有很高的及时性和有效性。

## KubeNode：云原生节点运维底座介绍





### What & hy

- 以云原生方式管理节点生命周期及节点组件
- 申明式、面向终态

### 组成

- 中心端：

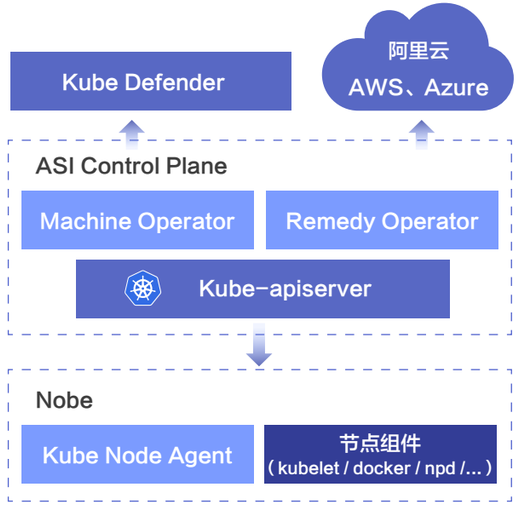
Machine Operator：节点及组件管理
Remedy Operator：节点故障自愈

- 节点侧：

Kube Node Agent：单机 agent

- 配套组件

Kube Defender 统一风控
NPD：单节点故障检测




KubeNode，是阿里巴巴自研的基于云原生方式来管理和运维节点的底座项目，相比于传统的过程式的运维手段，KubeNode 通过 k8s 扩展 CRD 以及对应的一组 Operator，能提供完整的节点生命周期和节点组件生命周期管理，通过申明式、面向终态的方式，把节点及节点组件的管理变得跟管理 k8s 中的一个应用一样简单，并实现节点的高度一致性以及自愈修复的能力。

上图右侧是 KubeNode 一个简化的架构，整体是由这几个部分组成的：

中心端有 Machine Operator 负责节点和节点组件管理，Remedy Operator 负责节点的故障自愈修复。节点侧有 Kube Node Agent，这个单机 agent 组件负责 watch 中心 Machine Operator、Remedy Operator 生成的 CRD 对象实例，并执行相应的操作，像节点组件的安装、故障自愈任务的执行等。

配合着 KubeNode，阿里巴巴还使用 NPD 进行单机的故障检测，以及对接 Kube Defender (阿里自研组件)进行统一风控。当然社区版本的 NPD 提供的故障检测项是比较有限的，阿里巴巴在社区的基础上进行了扩展，结合阿里巴巴多年节点、容器运维的实践，加入了很多节点的故障检测项，大大丰富了单机故障检测的能力。

## KubeNode 和社区项目关系

**github.com / kube-node**

不相关，该项目 2018 年初已停止

**ClusterAPI**

KubeNode 可以作为 ClusterAPI 节点终态的补充

功能对比：

| | Cluster API | KubeNode |
|---|---|---|
| 集群 Provision | Yes | No |
| 节点 Provision | Yes | Yes |
| 节点组件终态 | No | Yes |
| 节点故障自愈 | Yes(simple) | Yes(full，rule，based) |





这里解释下阿里巴巴自研的 KubeNode 项目跟社区项目的关系。大家看到 kube-node 这个名字的时候，可能会觉得有点似曾相识，在 github 上是有一个同名的项目 github.com/kube-node，但其实这个项目早在 2018 年初的时候就已经停止了，所以只是名字相同，两者并没有关系。

另外社区的 ClusterAPI 是一个创建并管理 k8s 集群和节点的项目，这里对比了两个项目的关系：

1. 集群创建：ClusterAPI 负责集群的创建，KubeNode 不提供此功能。
2. 节点创建：ClusterAPI 和 KubeNode 都可以创建节点。
3. 节点组件管理和终态维持：ClusterAPI 没有提供相应的功能，KubeNode 可以管理节点组件并保持终态。
4. 节点故障自愈：ClusterAPI 主要提供基于节点健康状态重建节点的自愈能力；KubeNode 提供了更丰富的节点组件自愈功能，能对节点上的各种软硬件故障进行自愈修复。

总的来说，KubeNode 可以和 ClusterAPI 一起配合工作，是对 ClusterAPI 的一个很好补充。

这里提到的节点组件，是指运行在节点上的 kubelet，Docker 的软件，阿里内部使用 Pouch 作为我们的容器运行时。除了 kubelet，Pouch 这些调度必须的组件外，还有用于分布式容器存储、监控采集、安全容器、故障检测等总共十多个组件。

通常安装升级 kubelet，Docker 是通过类似 Ansible 等面向过程的一次性动作来完成的。在长时间运行过程中，软件版本被意外修改或是碰到 bug 不能工作的问题是很常见的，同时这些组件在阿里巴巴的迭代速度非常快，往往一两周就需要发布推平一个版本。为了满足组件快速迭代又能安全升级、版本一致的需求，阿里巴巴自研了 KubeNode，通过将节点以及节点组件通过 k8s CRD 的方式进行描述，并进行面向终态的管理，从而确保版本一致性，配置一致性以及运行态正确性。

## KubeNode - Machine Operator

### CRDs

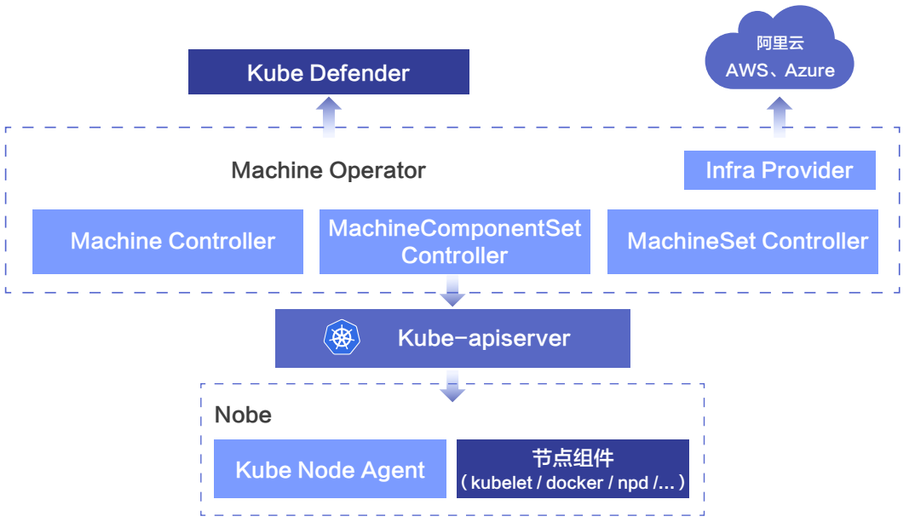
- Machine：节点元信息
- MachineSet（MS）：节点集合
- MachineComponentSet（MCS）：节点组合集合
- MachineComponentSet（MC）：节点组件

### Controllers

- MS controller：节点 Provision
- MCS controller：节点组件分批安装、升级
- Infra Provider：对接云厂商 OpenAPI

### Kube Node Agent

- 单机组件安装、升级、终态维持




图中是 Machine Operator 的架构，一个标准的 Operator 设计：扩展的一组 CRD 再加上中心的控制器。

CRD 定义包括：跟节点相关的 Machine、MachineSet，以及跟节点组件相关的 MachineComponent、MachineComponentSet。





中心端的控制器包括：Machine Controller、MachineSet Controller、MachineComponentSet Controller，分别用来控制节点的创建、导入，节点组件的安装、升级。

Infra Provider 具有可扩展性，可以对接不同的云厂商，目前只对接了阿里云，但是也可通过实现相应的 Provider 实现对接 AWS、Azure 等不同的云厂商。

单机上的 KubeNode 负责 watch CRD 资源，当发现有新的对象实例时，则进行节点组件的安装升级，定期检查各个组件是否运行正常，并上报组件的运行状态。

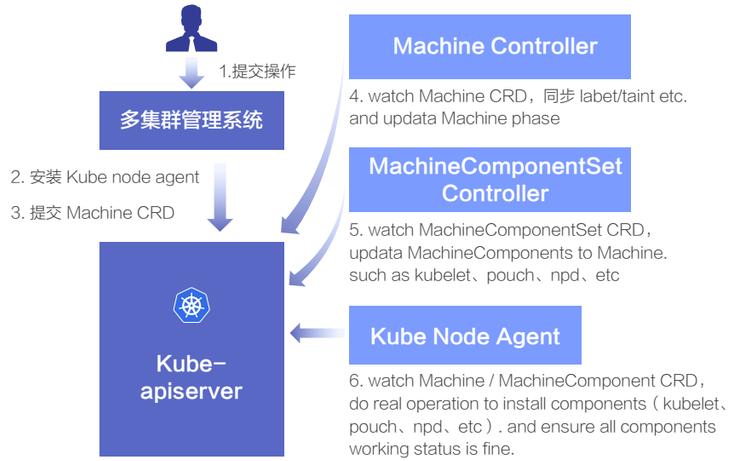
## Use Case：节点导入

### k8s 扩展 CRD 描述节点及组件

- Machine
- MachineComponent
- MachineComponentSet

### 节点组件确保终态一致

- version
- config
- status



以下分享基于 KubeNode 对已有节点的导入过程。

首先用户会在我们的多集群管理系统中提交一个已有节点的导入操作，接下来系统先下发证书，并安装 Kube Node Agent，等 agent 正常运行并启动之后，第3步会提交 Machine CRD，接下来 Machine Controller 会修改状态为导入 phase，并等 Node ready 之后从 Machine 上同步 label / taint 到 Node。第 5 步是 MachineComponentSet，根据 Machine 的信息确定要安装的节点组件，并同步到 Machine 上。最后 Kube Node Agent 会 watch 到 Machine、MachineComponent 的信息，完成节点组件的安装，并等所有组件运行正常后，节点导入操作完成。整个过程跟用户提交一个 Deployment 并最终启动一个业务 Pod 是类似的。

节点组件的终态一致主要包含了软件版本、软件配置和运行状态的正确、一致。

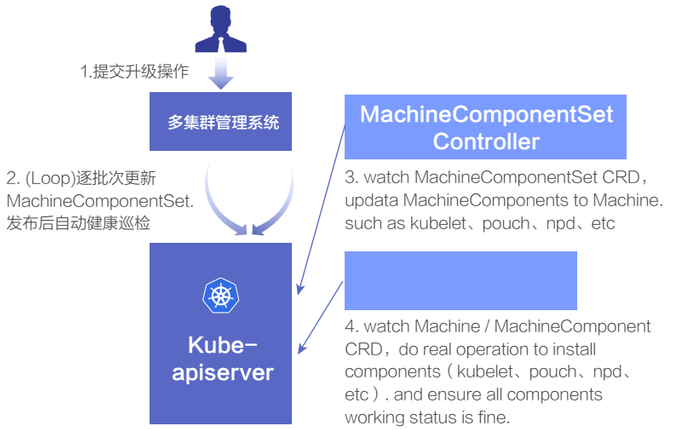
## Use Case：组件升级

### ASIOps

- ASI 组件变更统一 CD 平台
- 上百集群 Pipeline 自动流水线发布
  测试 -> 测法 -> 正式
- 变更后自动触发健康巡检

### KubeNode 组件升级

- 逐批次灰度、暂停升级
- 单机 watch 变化触发升级, 高并发高效率
- 健康巡检异常状态上报、暂停自动变更








这里介绍下组件的升级过程，主要依赖的是 MachineComponentSet Controller 提供的分批次升级的能力。

首先用户在多集群管理系统上面提交一个组件升级操作，然后就进入一个逐批次循环升级的过程：先更新 MachineComponentSet 一个批次升级的机器数量是多少，之后 MachineComponentSet Controller 会计算并更新相应数量的节点上组件的版本信息。接下来 Kube Node Agent watch 到组件的变化，执行新版本的安装，并检查状态正常后上报组件状态正常。当这个批次所有的组件都升级成功后，再开始下一个批次的升级操作。

上面描述的单集群单个组件的升级流程是比较简单的，但对于线上十多个组件、上百个集群，要在所有的集群都完成版本推平工作就不那么简单了，我们通过 ASIOps 集群统一运维平台进行操作。在 ASIOps 系统中，将上百个集群配置到了有限的几个发布流水线，每条发布流水线都按照：测试 -> 预发 -> 正式 的顺序编排。一次正常的发布流程，就是选定一条发布流水线，按其预先设定好的集群顺序进行发布，在每个集群内部，又按照 1/5/10/50/100/... 的顺序进行自动发布，每一个批次发布完成，会触发健康巡检，如果有问题则暂停自动发布，没问题的话等观察期结束，则自动开始下一个批次的发布。通过这样的方式，做到了既安全又高效的完成一个组件新版本发布上线的过程。

## KubeNode – Remedy Operator

### CRDs

- NodeRemedier: 节点故障修复规则
- RemedyOperationJob: 节点自愈修复任务

### Host Doctor

- 中心故障诊断, 对接主动运维事件

### Kube Node Agent

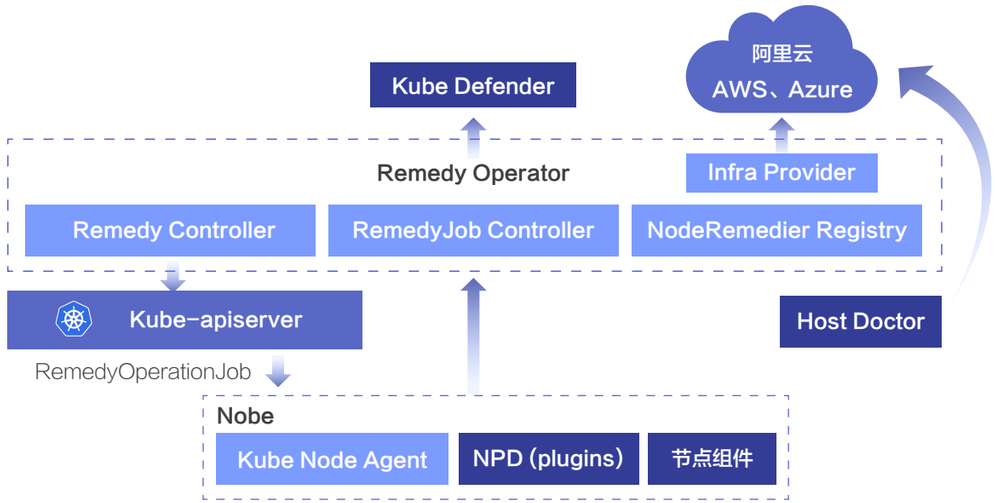
- 单机自愈修复任务执行

### Controllers

- Remedy controller: 自愈控制
- RemedyJob controller: 自愈任务控制
- NodeRemedier Registry: 自愈规则注册中心

### NPD

- 节点故障检测
  (插件式 : kernel/kubelet/docker/...)




接下来分享 KubeNode 里面的 Remedy Operator，也是一个标准的 Operator，用来做故障自愈修复。

Remedy Operator 也是由一组 CRD 以及对应的控制器组成。CRD 定义包括：NodeRemedier 和 RemedyOperationJob，控制器包括：Remedy Controller、RemedyJob Controller，同时也有故障自愈规则的注册中心，在单机侧有 NPD 和 Kube Node Agent。

Host Doctor 是在中心侧的一个独立的故障诊断系统，对接云厂商获取主动运维事件并转换为节点上的故障 Condition。在阿里云公有云上，ECS 所在的物理机发生的硬件类的故障或是计划中的运维操作，都会通过标准 OpenAPI 的形式获取，对接后就可以提前感知节点的问题，并提前自动迁移节点上的业务来避免故障。





## Use Case：夯机自愈

**故障自愈**

- NPD -> Node Condition -> Remedy

**Remedy 自愈优势**

- 云原生自闭环自愈链路
- 覆盖广：硬件、OS、组件
- 秒级故障发现、分钟级故障自愈
- 对接风控，防止自愈操作引发二次故障


这里以夯机自愈的案例来介绍一个典型的自愈流程。

首先我们会在多集群管理系统 ASI Captain 上配置下发 CRD 描述的自愈规则，并且这些规则是可以灵活动态增加的，针对每一种 Node Condition 都可以配置一条对应的修复操作。

接下来节点上的 NPD 会周期性的检查是否有发生各种类型的故障，当发现内核日志中有出现 "task xxx blocked for more than 120 seconds" 的异常日志之后，判定节点夯机，并给 Node 上报故障 Condition，Remedy Controller watch 到变化后，就触发自愈修复流程：首先会调用 Kube Defender 风控中心接口判断当前自愈操作是否允许执行，通过后就生成 RemedyOperationJob 自愈任务，Kube Node Agent watch 到 job 后执行自愈操作。

可以看到整个自愈流程不依赖于外部第三方系统，通过 NPD 做故障检测，Remedy Operator 执行自愈修复，以云原生的方式完成了整个的自愈修复流程，最快可以做到分钟级的故障发现并修复。同时，通过对 NPD 检测规则的增强，处理的故障范围覆盖了从硬件故障、OS 内核故障、到组件故障的全链路修复。值得强调的是，所有的自愈操作都会对接 Kube Defender 统一风控中心，进行分钟级、小时级、天级别的自愈修复流控，防止当出现 Region / Zone 级别断网、大规模 io hang、或是其他大面积软件 bug 的情况下，触发对整个 Region 所有节点的故障自愈，引发更严重的二次故障。

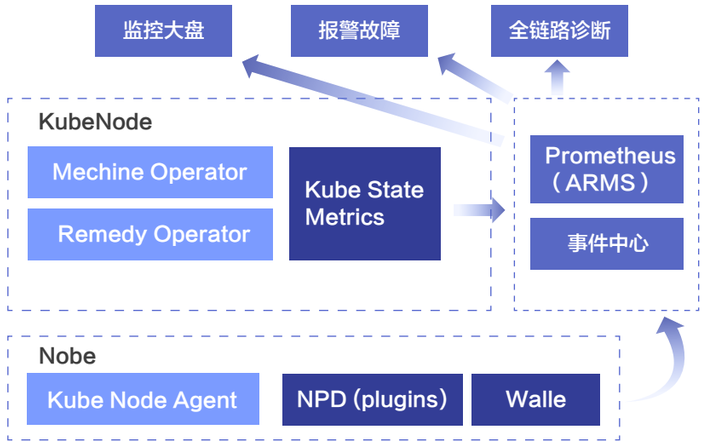
## KubeNode 数据体系

**数据体系**

- 数据采集链路
- 统一数据采集和存储

**数据体系**

- 资源利用率分析统计
- 实时监控报警
- 整体故障分析统计
- 节点组件覆盖度、一致率分析
- 节点自愈效率分析
- 全链路诊断








KubeNode 数据体系建设的好坏对整体度量并提升 SLO 起着非常重要的作用。

在节点侧，NPD 会检测故障并上报事件中心，同时 walle 是单机侧的指标采集组件，会采集节点以及容器的各种指标项信息，包括像 CPU / Memory / IO / Network 等常见的指标，以及很多其他的像内核、安全容器等的指标项。中心侧 Promethes **(阿里云公有云上的 ARMS 产品)** 会采集所有节点的指标并进行存储，同时也采集扩展过的 Kube State Metrics 的数据，获取 Machine Operator 和 Remedy Operator 的关键指标信息。 在获取到这些数据的基础之上，再在上层面向用户，配置监控大盘、故障报警、以及全链路诊断的能力。

通过数据体系的建设，我们就可以用来做资源利用率的分析统计，可以提供实时的监控报警，进行故障分析统计，也可以分析整体 KubeNode 中的节点以及节点组件的覆盖率、一致率、节点自愈的效率，并提供针对节点的全链路诊断功能，当排查节点问题时，可以查看该节点上历史发生过的所有的事件，从而帮助用户快速定位原因。

## 未来展望

目前 KubeNode 已经覆盖了阿里巴巴集团的所有的 ASI 集群，接下来，将随着阿里巴巴经济体“统一资源池”的项目，推进 KubeNode 覆盖更大的范围、更多的场景，让云原生的容器基础设施运维架构发挥更大的价值。

另外阿里巴巴也计划在 2020 年下半年开源 KubeNode，届时欢迎大家一起参与共建。

# 2/2 一种基于硬多租的大数据 Serverless 解决方案

阿里云开发工程师　庄清惠　|　蚂蚁金服高级开发工程师　范康

**摘要：**随着 kubernetes 的不断成熟，大数据向 kubernetes 迁移已经成为了一种趋势。由于大数据应用本身的复杂性，目前大数据 on kubenetes 通常采用 operator + crd 的方式，以独占集群的视角来管理计算任务。独占集群的任务管理模式使得在构建 SaaS 服务时，大数据应用对 kubenetes 集群的隔离要求与 kubenetes 集群的利用率成为了难以调和的矛盾。

- 阿里云实时计算服务在 Serverless 领域的最新实践成果，如何结合 virtualcluster 和 多租网络技术提供更有竞争力的解决方案
- virtualcluster 如何实现完整的 kubernetes 控制面多租和集群利用率提升
- 在基于云环境的 vpc 网络和安全容器基础上如何构建多租 kubernetes 的服务发现

## 一、大数据Serverless平台需求

在这里先对本次讨论的场景做一个简单的定义，本文提到的大数据 Serverless 平台是什么？阿里云作为一个公有云提供商，提供丰富多样的云服务，其中也包括大数据服务，比如基于 flink 的实时计算服务。同时，为了进一步帮助用户专注于自身业务，提供潜在的灵活性和成本节约的能力，我们希望能够以 Serverless 的形式提供相应服务。而本文提到的大数据 Serverless 平台，就是支撑 Serverless 大数据服务的底座。





首先，对于一个 Serverless 服务，对用户提供的最重要的能力就资源弹性的能力，相较于用户自建集群或者购买独立集群的模式，大数据 Serverless 平台需要为用户提供极致的资源弹性，真正让用户能按照自己使用的资源付费。同时用户不需要关注底层的资源容量，集群环境等信息，进一步降低运维开销。最后，一个成熟的大数据 Serverless 平台还应提供完备的管控工具和各种增值服务，进一步提升平台的易用性。而对于大数据平台的技术人员，他们关注的是如何安全高效的将大数据平台运行在云上。目前很多大数据框架都提供了基于 k8s 的部署模式，比如支持 nativeonk8s 或者提供配套的 operator。如果底层平台能够提供标准的 k8s 接口能力，那么在大数据框架上云时，能够充分复用现有技术。

同时，作为服务提供方，如何降低成本也是非常重要的。这里降低成本通常有两个方向，分别是资源利用率的提升，和运维成本的降低。

最后也是最重要的一点，就是无论什么情况下，作为公共服务的提供方，一定要保证整个系统的安全性。

- 极致的资源弹性，降低使用成本
- 无需关心任何底层资源信息，降低运维开销
- 简单易用

- 完全兼容 kubernetes 接口，服务一键上云
- 提升资源利用率、降低运维开销
- 安全性

## 平台架构模式的选择

下面来看看目前大数据云服务的主流形式。目前主流的大数据服务架构通常有独立集群模式和共享集群模式。

独立集群模式就是为单一用户购买一批虚拟机，组成一个用户独占的集群运行相应任务。这种模式最大的好处就是由于用户独占集群，隔离性强，不同租户间隔离性有保障。但是缺陷也很明显。对于用户来说，独占集群的资源粒度是以 ECS 为粒度的。ECS 作为资源的最小粒度，弹性较差，带来更高的硬件成本。同时对于平台运维人员来说，不同小集群需要管理不同的配置，配置碎片化问题较为严重。这些问题导致独立集群模式不适合作为大数据 Serverless 产品的底层平台。

另一种模式是共享模式，共享模式指的是多个用户共享底层集群的资源，此时用户的最小资源粒度为 pod 粒度，用户无需感知机器信息，资源粒度更小，弹性更大，启停速度也更快，更容易做到按需使用，随用随走，更契合 Serverless 的理念。同时对于运维同学来说，数个大集群的资源、配置管理代价要小于数百个小集群，因此共享集群模式也能很大程度上降低服务提供方的运维成本。但是共享集群模式有一个很大的问题，就是其隔离性较差。用户共享一个集群，无论是数据平面还是管控平面都存在一定程度的共享，如果基于共享集群模式提供大数据 Serverless 服务，一定要解决隔离性的问题。

**租户独立集群模式**

- **优势**
  - 更好的隔离性
    (control plane/ network/ runtime)
- **劣势**
  - 资源粒度大，弹性差
  - 硬件成本高
  - 运维代价大，配置碎片化严重

**共享集群模式**

- **优势**
  - 资源粒度小，弹性好
  - 硬件成本低
  - 运维代价小
- **劣势**
  - 隔离性差

## 共享集群模式基于 namespace 的多租方案及其缺陷

下面展开讨论下大数据应用在现有的共享集群多租方案下有哪些缺陷。namespace 是 k8s 内置的租户隔离机制，但是基于 namespace 的多租方案不同用户间是共享 control plane的，这会带来一系列的限制。





首先，共享 control plane 会使得用户之间访问 apiserver 互相干扰，用户 a 频繁访问 apiserver 会导致用户 b 访问 apiserver 缓慢。而大数据应用物理 operator 还是 native 模式都必须访问 apiserver，这个问题无法避免。同时多租户共享数据面有信息泄露的风险，需要精细的 rbac 授权，而 rbac 权限管理本身也是一个复杂的问题。第三，在基于 namespace 的多租场景下，部分 k8s 资源用户是无法安装的，如 crd、webhook 和 clusterrole。目前开源社区的大数据 on k8s 解决方案大多以 operator 为基础，强依赖上述三类 k8s 资源。这使得用户无法直接将社区大数据 on k8s 方案迁移至共享模式的 k8s 集群，需要很大的改造成本。最后，除了上述控制面的共享带来的问题，在数据层，不同用户的 pod 会运行在同一个物理机上，如何保证用户的运行时环境隔离、网络隔离和存储隔离也是很重要的问题。

综上所述，基于 namespace的多租方案需要用户间共享管控面和数据平面，会带来各种问题。而目前大数据 on k8s 的方案通常以独占集群视角使用 k8s，在一定程度上放大了上述问题，使得大数据 Serverless 服务直接基于 namespace 隔离的共享集群方案难以实施。

## Kubernetes 如何更好地支持大数据 Serverless 形态产品

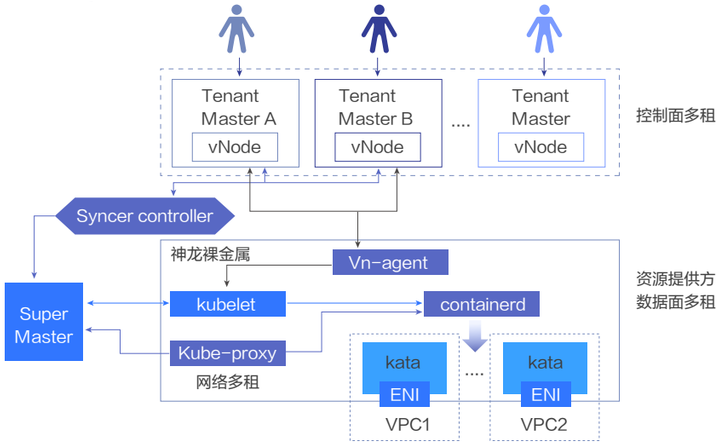
基于前面的讨论，我们可以总结出，如果希望在 k8s 上能支持大数据 Serverless 产品，需要满足以下能力。

首先，k8s 层需要提供更好的隔离性，保证不同用户间无法相互影响。因为安全性是一个云产品的根本。其次，需要能够提供 Serverless 级别的资源使用体验：对于用户侧，提供更小的资源粒度和更加的资源弹性；对于平台侧，提供统一的资源池和资源调度，节约资源成本和运维开销。最后，由于大数据应用的复杂性和社区大数据 on k8s 的现状，要想方便的支持大数据框架上云，需要为大数据框架提供原生的 k8s 体验，包括接口和权限，降低应用迁移的成本。基于以上需求，我们提供了一种全新的 k8s 多租方案，用于更好的支持大数据 Serverless 这一业务场景。

| 更好的隔离性 | Serverless 资源使用体验 | 原生的 kubernetes 体验 |
| --- | --- | --- |
| • Control plane 隔离<br>• Runtime 隔离<br>• 网络隔离<br>• 存储隔离 | • 统一的资源调度<br>• 以 pod 为粒度的资源弹性<br>• 用户侧无需运维资源，资源管理下沉至平台层 | • 提供原生 kubernetes 接口<br>• 提供用户视角内 kubernetes 的全部权限 |
| 安全可靠 | Pay As You Go | 一键上云 |

## 二、阿里云大数据 Serverless 技术方案

接下来将介绍数据 Serverless 解决方案底层“硬多租”架构的技术细节。硬多租架构的核心组件已经开源至 kubernetes 社区的 multi-tenancy 小组孵化，https://github.com/kubernetes-sigs/multi-tenancy/tree/master/incubator/virtualcluster。







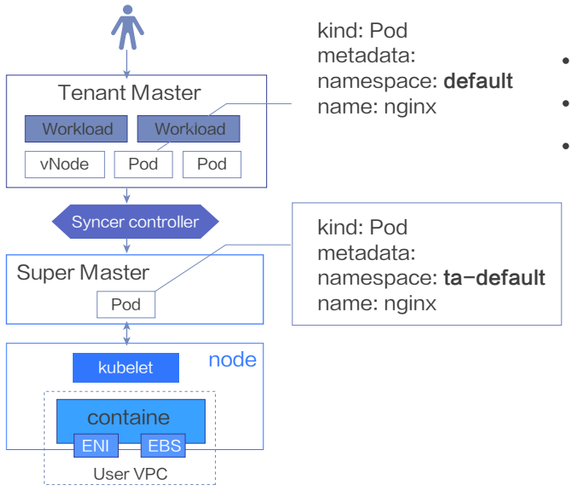

在这个架构里首先引入两个概念，一个是超级集群（super master），是实际管理以及生产物理集群的集群；另一个是租户集群（tenant master），是在超级集群之上虚拟出来的 k8s 集群，控制平面是独立，但实际上使用的是超级集群的物理资源。每个租户集群会给独立、真实存在的 kubernetes apiserver, controller manager, etcd 作为租户独占的 k8s 控制面。租户集群的生命周期由一个独立的 operator，vc manager 来管理。平台层通过提交 VC CR 即可创建出一个租户集群。租户在使用kubernetes的时候，入口是租户独立的 apiserver，租户与租户之间的 k8s 核心组件从物理层面就隔离开了。

实际的资源生产是如何实现的呢？租户在向自己的apiserver 提交pod的时候，会由一个叫 syncer controller 的组件将 pod 相关对象做多租转换后提交至超级集群，由超级集群实际生产出资源后，再将状态反向告知给租户集群。

目标就是基于这种模式之下，让每个租户使用 k8s 就像使用一个真实的k8s集群一样。这就是所谓的控制面多租。除了计算资源的生命周期管理，VC 还支持 kubelet server api，即允许租户登录容器和查看日志等操作，VC 通过添加一个名为 vn-agent 的组件来做 kubelet server api 的代理支持这一功能。

在 virtual cluster 架构了，为了提升资源利用率，会将不同租户的容器混跑在同一台神龙物理机上。这里引出了数据面隔离的概念，单机上不同租户的容器必须是强隔离的，不会互相干扰，同时也无法侵入其他容器。

## 控制面隔离




kind: Pod
metadata:
namespace: **default**
name: nginx

kind: Pod
metadata:
namespace: **ta-default**
name: nginx

- 同步 12 种必要资源至 super 集群
- 不修改任何 k8s api，与上游 k8s 100% 兼容
- 租户集群通过 99% 的一致性认证测试

**已开源，在 multi-tenancy 小组孵**

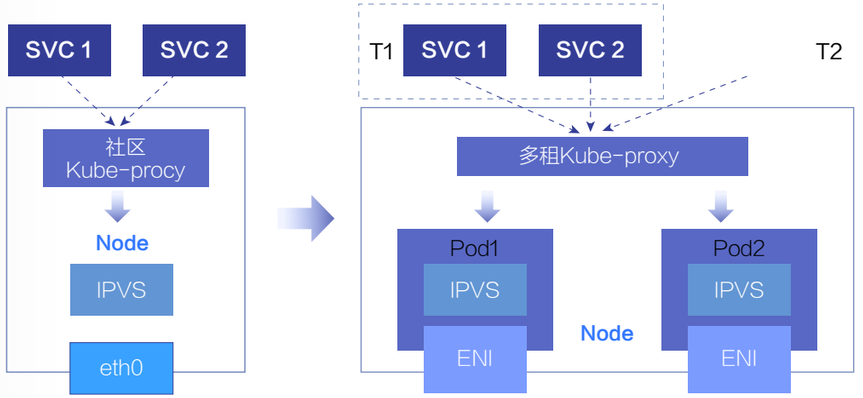
前面简单介绍了控制面多租的工作模式。控制面多租的设计有这么几点原则：只同步必要的资源至集群；不修改 k8s 原生组件，比如 apiserver, controller manager, kubelet 等，与上游的 k8s 完全兼容；最大限度让租户集群成为一个标准的 kubernetes 集群，也即通过 kubernetes conformance test。

首先，并不是所有类型的资源都需要同步，k8s 自身管理平面的资源依旧留在租户独立的 apiserver 里运转，比如 deployment, RBAC role 等逻辑。真正需要同步的是计算类型的资源。而在 k8s 里，计算资源以 pod 的形式呈现，所以我们真正需要同步的是与 Pod 运行相关的资源。总共有 12 种，分别是 ConfigMap, Endpoints, Event, Namespace, Node, PersistentVolume, PersistentVolumeClaim, Pod, Service, ServiceAccount, Secret, StorageClass. Pod 在同步到 super 集群前会做多租化的转换，包括 ns 名称，dns, env, service account token 等。在同步至 super 集群之后，这个容器从物理上属于 super 集群，但从逻辑上归属于租户集群。目前的实现下，租户集群能通过 199/204 的 conformance test case.





## 数据面隔离




**网络隔离**

- 不同租户间通过 vpc 强隔离
- 租户集群支持 service 服务发现

**计算隔离**

- 阿里云安全沙箱

**存储隔离**

- 弹性云盘

前面提到数据面隔离的背景是，单机上会运行不同租户的容器。包括计算、存储、网络三部分的隔离。首先计算隔离，这一块现有的方案比较成熟，使用的是阿里云安全沙箱。存储隔离则使用的是云盘。本文将重点讨论网络的隔离。

网络隔离的基础是每个租户的容器都运行在自己的 VPC 内，这样从底层网络实现的角度，不同租户间容器网络就已经是强隔离的了。又因为使用的是安全容器，在这个解决方案里，是不会使用类似 bridge 的网络的，这样会让安全容器的网络性能特别差。这里通过直通ENI至安全容器的方式来让安全容器的网络性能几乎与宿主机持平。在解决完隔离性和性能问题后，引申出了另一个问题，租户集群内部的服务发现能力，也即支持 k8s 的 service。先看左图，这是社区版 kube-proxy 的实现原理，它会将 LVS 规则写到宿主机网络空间，那么对于普通的 runc，它的网络包是会经过宿主机网络协议栈，这个 LVS 的规则自然就生效了。但是在安全容器场景下，使用社区版的 kube-proxy 是无法工作的，因为安全容器内部的进程，它的网络包经过 guest os 的网络协议栈之后，直接走直通ENI 出去了，是不会经过宿主机网络协议栈，也就是说写在宿主机网络空间的路由规则根本就不生效。硬多租的解决方案是通过对 CRI 接口的拓展，将 LVS 规则灌注至安全容器内部，来支持 k8s service 的路由联通。同时，kube-proxy 也是要做多租改造的，在更新路由规则的时候，得按租户维度分组来刷新，比如租户 A 的容器只能刷租户 A 的规则，不能刷到租户 B 内，租户与租户之间相互是不可见的。从底层网络强隔离，租户内部又是支持 k8s 原生服务发现机制的，这就是硬多租中的网络隔离。

对以上两点做一个简单的小结，硬多租解决方案在控制面通过独立的 master 组件做到了物理层面的隔离，在数据平面基于阿里云安全沙箱做计算、网络、存储隔离。这就是所谓的“硬多租”。

## 三、阿里云实时计算产品 Flink 全托管产品的大数据 Serverless 实战

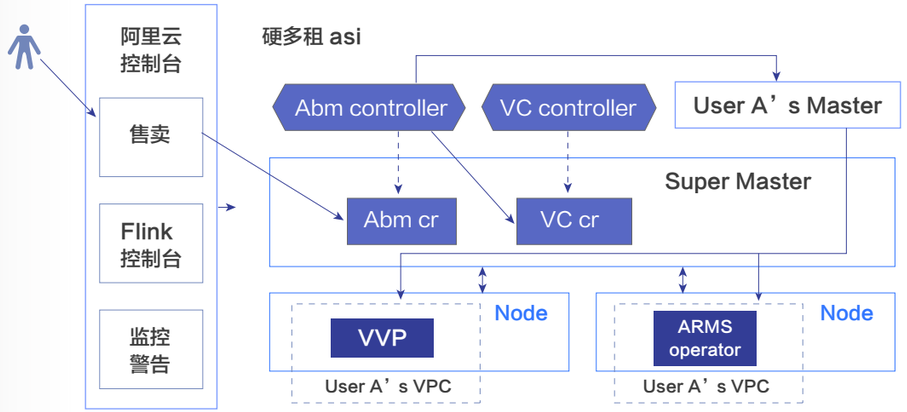
阿里云实时计算产品基于硬多租 k8s 架构搭建，用户通过阿里云控制台进行服务购买，flink 任务管理等操作。同时该产品自动对接阿里云 arms 服务，为用户提供监控报警能力。下图展示的就是整个产品的架构，每个用户购买产品后会拥有一个独立的 k8smaster，用户通过flink控制台向租户 k8s 提交任务。提交的任务会被同步到 super 集群运行，用户完全不感知底层机器信息和资源情况，只需要关心任务本身的运行状态。同时产品自动对接了阿里云的 arms 监控报警，用户的每个任务都可以直接从控制台跳转到 arms 查看监控信息并配置报警。





在整个架构中，我们额外提供了两个组件： ABM controller 和 VC controller 来管理整个集群。ABM 全称是 Alibaba Bigdata Manager, 是一个大数据系统通用的管理组件，他提供一套通用的定义能力用于在 k8s 上安装丰富的大数据服务。而 VC controller 负责管理整套系统中所有的 virtual cluster。下面以一个新用户购买流程为例，展示一下如何利用这两个组件对平台进行自动化管理。

### 集群开服流程




在开服的流程中，用户会在阿里云控制台发起对产品的购买，这个购买逻辑最终会在 super 集群上提交一个 ABM的CR(custom resource)。在 CR 中会声明用户购买的大数据服务的类型，比如说 flink 服务。之后 ABM controller 会 watch 到这个 CR，并开始为用户开通服务。ABM 本身管理着各个组件的安装模板，并且支持以一个工作流的模式，为用户管理一个全托管大数据产品的整个流程。如上图所示，在 flink 服务开通过程中， ABM controller watch 到 abm CR 信息，对应的是如下工作流。

1. 为用户创建一个 vc
2. 为用户开通对应 vc 的监控报警服务
3. 为用户拉起 flink 任务控制台 vvp

拉起 VC 这一步是由ABM提交一个 VC CR，通过 VC controller 来创建这个虚拟集群。用户下线则是相反的操作，当用户结束服务购买时，系统会删除一个 ABM CR，反向一步步删除用户的所有资源，包括 tenant master 资源和 super 集群上的 pod 资源。

值得一提的是，整套架构不仅能支持 flink 服务，对于其他的大数据服务，我们只需要配置对应的 abm 工作流，就可以将服务提供给终端用户，无需对大数据框架进行任何更改

## VVP: 企业级 Flink 控制台

接下来简单介绍一下 VVP (Ververica Platform)，vvp 是阿里云推出的企业级 flink增强版本的控制台，它主要围绕 flink 提供以下能力

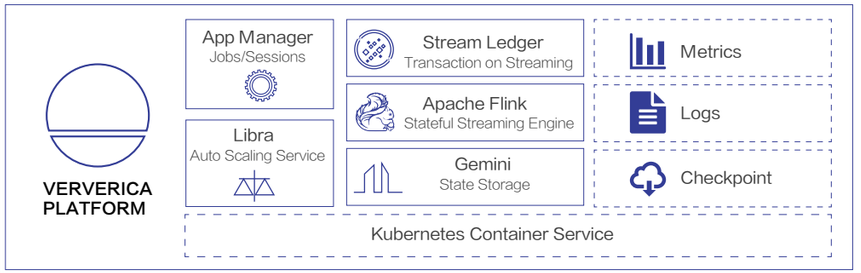
- Appmanager: 生命周期管理
- Libra service: 自动调优
- Ledger: 分布式跨行跨机器解决方案
- Gemini: 状态存储插件，2-3 倍性能提升

而最右侧，metrics 部分由系统帮用户自动对接阿里云 arms；logs 部分用户可选是否开通阿里云 SLS 服务；checkpoint 部分用户可选对接到阿里云OSS服务。

整套服务对用户来讲开箱即用，并且有各种增值服务和运维服务的集成，解放用户人力完全投入业务开发，不需要关注任何运维相关的问题








## 四、总结

**用户**

- 按量付费，pay as you go
- 全链路作业开发、管理系统
- 无缝对接阿里云运维监控体系
- 节约成本

**大数据开发**

- 提供社区原生 k8s 接口，其他产品可以无缝对接
- 安全性

**平台运维**

- 无需运维多个小集群，集群配置统一
- 大资源池，资源水位更加平滑，资源管理更加容易
- 统一的监控大盘、作业诊断能力

综上所述，我们提出了一套基于硬多租 k8s 的大数据 Serverless 的解决方案，并基于该方案推出了 flink 全托管服务。该解决方案对于用户，大数据平台开发人员和平台运维同学都有很多的益处。

对于用户来讲，他能享受到以 pod 为粒度的资源弹性，并且能够以按量付费的方式使用资源。同时可以利用 vvp 提供的强大增值能力，方便地管理作业，利用阿里云的运维监控能力，实现高效运维，节约运维成本。

对于大数据平台开发同学来讲，该解决方案提供了和社区一致的 k8s语义，社区开源的大数据 on k8s 方案可以无缝对接，对外提供 Serverless 服务，并且不用担心多租环境下的安全性问题。

对于运维同学来讲，无需运维多个小集群，避免了无数配置问题带来的烦恼。而统一的资源池更够更方便的进行资源水位预测与资源管理。最后，在大集群上运维同学也能够以更低成本的实现统一的监控大盘和作业诊断能力。

阿里新技术方案

# 2/3 Kubernetes 异常配置检测的DSL框架

阿里云容器服务技术专家　邓隽　|　阿里云容器服务开发工程师　顾静

## 1. Kubernetes 典型异常

应用部署、集群扩容、组件升级等都是 Kubernetes 常见运维操作。当这些操作异常时，往往很难定位原因，尤其是集群层面的异常。

常见的 K8s 异常如访问集群出现概率性失败。从客户端发起请求到 apiserver pod 处理请求并返回结果，其中牵涉多个环节。可能是客户端网络问题，可能是SLB访问控制问题，可能是 service 配置问题，也可能是因为 apiserver pod 异常导致。如果对 Kubernetes 没有深入了解，往往很依靠人工经验定位问题。

集群网络异常也是在运维工作中经常出现的异常。Kubernetes 网络是多个层次，不仅依赖于 Node 的网络，比如 IAAS 层的安全组，路由表打通等，还依赖 Node 上的 Linux路由、iptables、bridge等虚拟网络。因此，当集群网络出现问题后，排查链路非常长，极度依赖运维同学的网络经验。此外，网络异常原因也难以定位，有时可能是因为一个毫不相关的操作导致了集群异常。例如我们遇到的一个真实场景：如图1所示，某客户在集群的某个节点上使用 yum 安装了一个自动运维工具，这个自动运维工具依赖了 firewalld 防火墙服务。而 firewalld 启动后自动设置了一些 iptables 规则，拦截了 Kubernetes 的 service 和 pod 的地址，从而导致了容器网络不通。对于这类问题，排查耗时耗力。





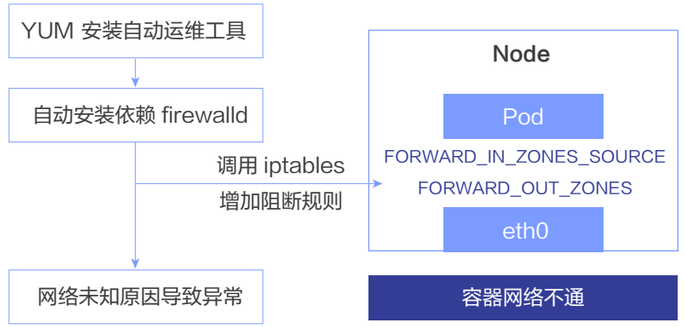



图1 集群网络异常示例图

## 2. Kubernetes 异常检测工具

近年来，开源领域涌现了许多用于 Kubernetes 异常检测的工具。

- **NPD(Node Problem Detector)**

NPD 一般以 DaemonSet 的方式部署在 Kuberentes 集群中，常用于检测硬件错误（CPU、内存、磁盘）、操作系统异常（NTP、内核死锁、文件系统异常）、Docke r异常（后台进程无响应）等问题。检测结果透出依赖于APIServer 及 Prometheus。

- **Sonobuoy**

Sonobuoy 采用 Collector pod及DaemonSet的方式进行部署。DaemonSet 用于确保每个节点上都有一个检测pod， Collector pod 用于收集检测pod上报的结果。Sonobuoy 常用于 Kubernetes 兼容性检查及节点自定义检查。检测结果的采集和分析需要依赖用户自行处理。

- **其他**

| 工具 | 适用场景 | 局限性 |
| --- | --- | --- |
| kube-bench | 在集群中运行 CIS Benchmark<br>检测项依赖于 CIS Benchmark 内容 | 能发现集群核心组件配置错误<br>无法发现如 Flannel 组件异常<br>增加检查项流程较复杂 |
| kuberhealthy | 在集群中运行 CronJob 实现检查<br>可以自定义检查项 | 无法检测集群核心组件配置<br>集群异常时无法进行检测 |
| kube-hunter | 适用于集群安全检测 | 仅能检测集群安全性 |
| kubectl-trace | 在集群中运行 bpftrace 检查 Kernel | 仅能检测 Kernel 相关问题<br>要求熟悉 bpftrace 语言 |

## 3. Kubernetes 异常配置检测框架

现有的异常检查工具只能发现部分 Kubernetes 异常，难以很好地支持如集群升级、组件升级等场景。因此，我们提出一种 Kubernetes 异常检测框架，支持集群多纬度异常检测能力，同时也支持集成开源检测组件，具有较好的可扩展性及通用性。

### 3.1 Version 1.0

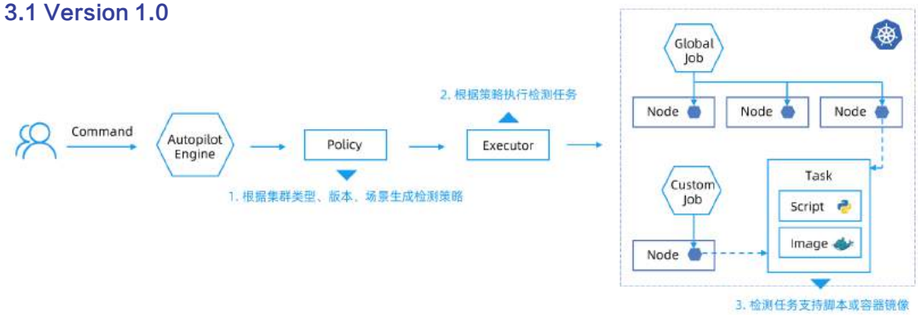



图 2 Kubernetes 异常配置检测框架 Ver1.0 架构图





异常配置检测框架 1.0 的主要目标是实现检测自动化。图 2 为框架 1.0 的架构图，如图所示，异常配置检测主要分为三步。

1、根据集群类型、集群版本及场景（集群升级、组件升级、安全检查等）生成检测策略

2、Executor 根据检测策略下发检测任务

3、执行检测任务。检测任务分为两类，一类为 Global Job，采用 DaemonSet 的形式部署，常用于检测节点相关配置如 Kernel 参数、Docker 参数等；另一类为 Custom Job，运行在部分节点上，常用于检测 Kubernetes 资源配置、集群资源清单等。

### 3.2 Version 2.0

随着 Kubernetes 版本的快速迭代、集群类型不断增加（托管版集群、Serverless 集群、边缘集群、GPU 集群等）、检测场景不断丰富，导致了检查项数量爆炸式增长，新增和维护检查项都比较困难。因此，我们提出了 Kuberentes 异常检测框架 2.0 版本，可以实现检查项的动态定制和动态扩展，其架构图如图 3 所示。

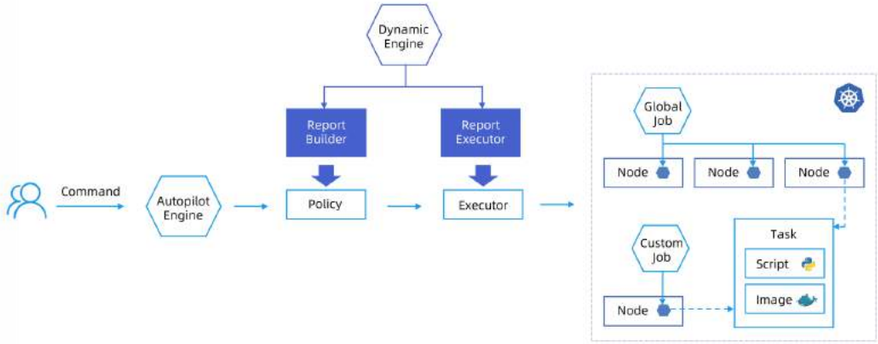



图 3 Kubernetes 异常配置检测框架 Ver2.0 架构图

与检测框架 1.0 相比，检测框架 2.0 增加了 Report Builder 和 Report Executor 部分。

### 3.2.1 Report Builder

Reprot Builder 负责根据不同集群版本、类型、场景动态定制检测报告，同时根据不同检测点动态定制检测策略，如图 4 所示。


图 4 Report Builder 模块示意图

### 3.2.2 Report Executor

Report Executor 负责根据不同检测策略动态扩展检测任务，如图 5 所示。用户可以使用 KCQL（Kubernetes Cluster Query Language）自定义检测任务。检测任务的数据来源多种多样，不仅可以为 Kubernetes 组件，还可以是开源的异常检测工具。

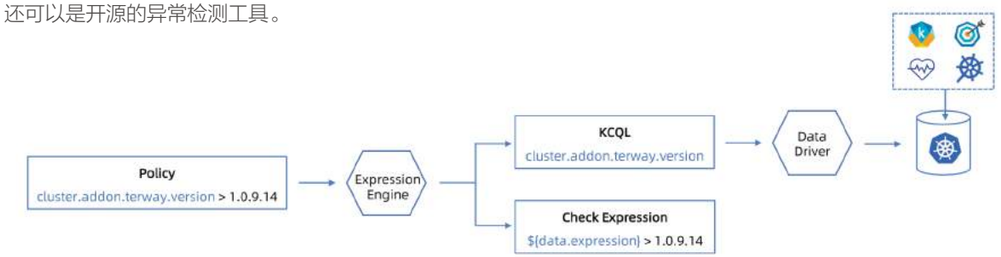



图 5 Report Executor 模块示意图

## 4. 实践

### 4.1 集群巡检

某客户 Node 节点新增安全组，一段时间后发现 Node 节点状态异常。经排查，根因是 Master 节点和 Node 节点处于不同的安全组中且两个安全组并未相互放开，新建安全组仅对新建连接生效，不会拦截已有连接，因此一段时间后问题才会浮现。





针对这种场景，客户在配置安全组后可通过集群巡检功能检测集群安全组配置是否符合预期，提前发现集群异常配置，避免线上故障。

### 4.2 集群网络

为了解决复杂的网络问题的检查，我们基于异常配置检测的引擎，支持了集群网络的自动问题诊断，如图6所示，其中包含了网络配置静态检测和网络动态异常巡检。

- **网络配置静态检测**

网络配置静态检测常用于检测一些静态的固定配置，比如安全组，路由表，内核参数，网卡配置等等是否跟集群中网络相关的配置是一致的，从而发现一些典型的配置被修改的情况，比如上文提到的安装防火墙软件导致的问题。

- **网络动态异常巡检**

当一些问题通过静态检测不能排查出时，例如一种未知的网络异常，通过动态的全链路的网络事件的采集，比如 IAAS 层的 Flow 采集，Node 中通过 ebpf 注册到内核的各个网络转发阶段的采集，最终自动发现网络问题出现的地方，大大降低网络链路手动梳理排查的成本。而动态异常巡检排查出的问题也会逐渐沉淀到静态检测中。

通过网络静态检测和网络动态异常巡检覆盖了白盒和黑盒网络检测的能力，大幅度的降低了运维人员排查网络问题的成本，提高运维效率，避免线上问题。

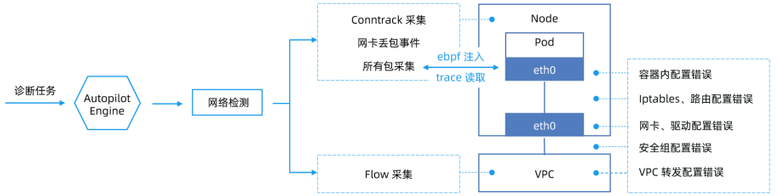



图6 集群网络诊断架构图

### 4.3 集群升级

为进一步提高集群升级成功率和集群稳定性，我们针对集群升级场景，不仅设置了集群升级前置检测，在升级完成后还设置了集群升级后置检测，从而形成完整的升级检测闭环，集群升级过程如图 7 所示。


图 7 集群升级示意图

## 5. 总结

随着 Kubernetes 版本不断迭代，集群产品类型不断丰富、检查场景不断增加，Kubernetes 异常配置检测变得十分困难。现有的开源Kubernetes异常检测工具仅能检测部分问题，且可扩展性差，因此我们提出一种基于 DSL 的 Kubernetes 集群异常检测框架。

- 框架已支撑了阿里云上万 Kubernetes 集群常态运行、集群升级、集群巡检、组件升级等场景
- 框架具有强通用性和扩展性，适用于多种集群版本、类型、场景
- 框架可实现零代码定制集群检查报告
- 框架可实现低代码扩展、集成多种异常检测能力





# 2/4 Serverless Kubernetes – 理想，现实和未来

阿里云Serverless Kubernetes产品TL　贤维

阿里云容器服务产品线负责人　易立

## 从 Serverless 容器到 Serverless Kubernetes

Serverless（无服务器）容器是让用户无需购买和管理服务器直接部署容器应用的产品、技术形态。Serverless 容器可以极大提高容器应用部署的敏捷度和弹性能力，降低用户计算成本；让用户聚焦业务应用而非底层基础设施管理，极大地提高应用开发效率，降低运维成本。

### Serverless 容器：关注应用而非基础设施


目前 Kubernetes 已经成为业界容器编排系统的事实标准， 基于 Kubernetes 的云原生应用生态（Helm, Istio, Knative, Kubeflow, Spark on k8s 等）更是让 Kubernetes 成为云操作系统。Serverless Kubernetes 得到了云厂商的高度关注。一方面通过 Serverless 方式根本性解决 K8s 自身的管理复杂性，让用户无需受困于 K8s集群容量规划、安全维护、故障诊断；另一方面进一步释放了云计算的能力，将安全、可用性、可伸缩性等需求由基础设施实现，这样可以形成差异化竞争力。

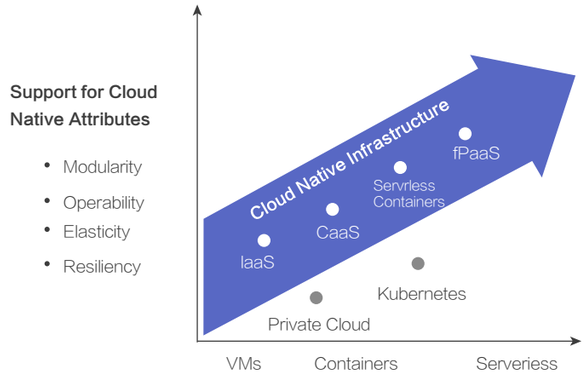
## 行业趋势

Gartner 预测到 2023 年，70% AI 任务会通过容器、Serverless 等计算模型构建。在 AWS 的调研中，在 2019 年 40% 的 ECS（AWS弹性容器服务）新客户采用 ECS on Fargate的Serverless Container 形态。

Serverless 容器是现有 Container as a Service 的进化方向之一，可以和 fPaaS/FaaS(Function as a Service)形成良好的互补。FaaS 提供了事件驱动的编程方式，用户只需实现函数的处理逻辑，比如当接收用户上传一个视频时，对视频进行转码和加水印。FaaS 方式开发效率很高，也可以将弹性发挥到极致，但需要用户改变现有的开发模式来进行适配。而 Serverless Container 应用的载体是容器镜像，灵活性很好，配合调度系统可以支持各种类型应用，比如无状态应用、有状态应用、计算任务类应用等等。用户大量现有的应用无需修改即可部署在 Serverless Container 环境中。








Gartner 在 2020 年 Public Cloud Container Service Market 评估报告中把 Serverless 容器作为云厂商容器服务平台的主要差异化之一，其中将产品能力划分为 Serverless 容器实例和Serverless Kubernetes两类。

Gartner 报告中也谈到 Serverless 容器业界标准未定，云厂商有很多空间通过技术创新提供独特的增值能力，其对云厂商的建议是：

- 扩展 Serverless 容器应用场景和产品组合，迁移更多普通容器 workload 到 serverless 容器服务。
- 推进 Serverless 容器的标准化，减轻用户对云厂商锁定的担忧。

## Competitive Landscape of Public Cloud Container Service Market

Generally Available　Preview

| Analysis Criteria | | Alibaba | AWS | Google | IBM | Microsoft | Oracle |
|---|---|---|---|---|---|---|---|
| Container Orchestration Services | Kubernetes | Alibaba Cloud Container Service for Kubernetes (ACK) | Amazon Elastic Kubernetes Service (EKS) | Google Kubernetes Engine (GKE) | IBM Cloud Kubernetes Service (IKS) | Azure Kubernetes Service (AKS) | Oracle Container Engine for Kubernetes (OKE) |
| | Other Orchestration Technologies | Alibaba Cloud Container Service[c] | Amazon Elastic Container Service (ECS) | | | | |
| Serverless Container Services | Serverless Kubernetes | Alibaba Cloud Serverless Kubernetes (ASK) | AWS EKS on Fargate | | | AKS Virtual Nodes | |
| | Serverless Container Instance | Alibaba Cloud Elastic Container | AWS ECS on Fargate | Google Cloud Run<br>Google Cloud Run on GKE | | Azure Container Instances (ACI) | |
| Kubernetes-Based Managed aPaaS | Red Hat OpenShift | | Red Hat OpenShift Dedicated[b] | Red Hat OpenShift Dedicated[b] | Red Hat OpenShift on IBM Cloud | Azure Red Hat OpenShift | |
| Container-Related Other Services | Service Mesh (Managed) | Alibaba Cloud Service Mesh | AWS App Mesh | Anthos Service Mesh | OpenShift Service Mesh<br>IKS Istio | | |
| | Registry | | | | | | |
| | Marketplace | | | | | | |
| Hybrid and Multicloud Solutions | On-Premises Deployable Container Solutions | ACK on Apsara Stack | ECS/EKS on AWS Outposts | Anthos | IBM Cloud Paks | AKS Engine on Azure Stack[d] | Oracle Linux Cloud Native Framework |
| | Multicloud Container Management Solutions | ACK | | Anthos (for the Other Clouds)[e] | IBM Cloud Pak Multicloud Management | Azure Arc | |
| General Criteria | First Production Support Date | May 2016 | April 2015 | August 2015 | March 2017 | March 2018 | May 2018 |
| | Pricing Model for Kubernetes Master Services[a] | Free | 10 Cents per Hour per Cluster (EKS)/Free (ECS) | 10 Cents per Hour per Cluster[f, g] | Free | Free | Free |

Source: Gartner

## 典型场景与客户价值

自各大公有云厂商全部布局 Serverless 容器领域，我们非常高兴的看到 Serverless 容器的价值逐渐被客户认可。典型客户场景包括：



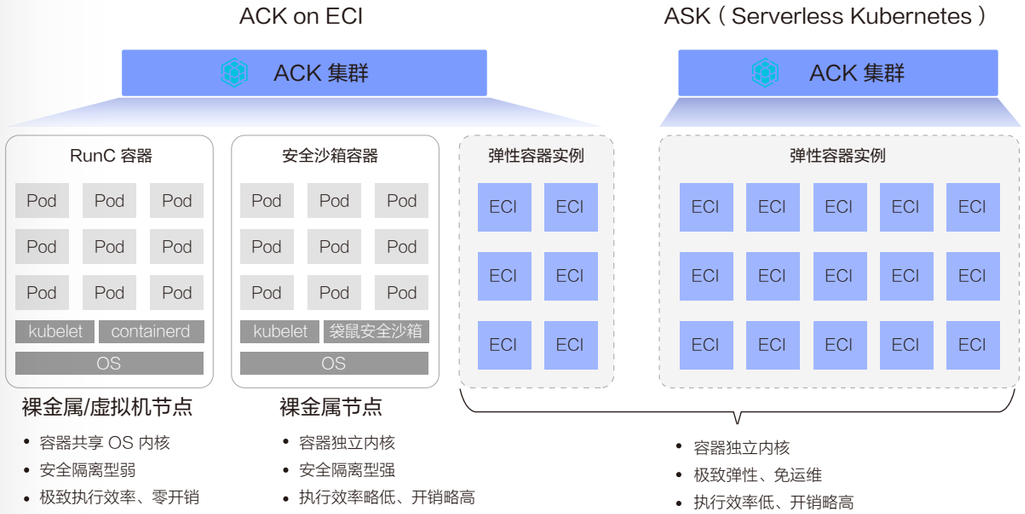


**在线业务弹性扩容：**支撑在线业务弹性扩容，在短时间内可以极速扩容上千个应用实例，轻松应对预期和非预期突发流量。比如此次疫情时期多个在线教育客户基于阿里云 ASK/ECI 超强弹性能力轻松面对业务高峰。

**免运维 Serverless AI 平台：**基于 Serverless 容器平台开发的智能、免运维的AI应用平台，可以让开发者创建自己的算法模型开发环境，而平台则会按需弹性伸缩，极大减少了系统维护和容量规划复杂性。

**Serverless 大数据计算：**通过 Serverless 化的 Spark, Pres to 等数据计算应用，灵活满足企业中不同业务部门快速成长过程中对多种计算计算任务，高弹性，强隔离、免维护的业务需求。

我们可以看到在 Serverless 容器领域有着不同的细分形态，第一种是把 Serverless 容器作为已有 Kubernetes 的弹性能力补充，第二种是在一个完全无节点管理和免运维的 Kubernetes 集群中使用 Serverless 容器。二者可以实现互补，覆盖满足不同客户的的诉求。




裸金属/虚拟机节点
- 容器共享 OS 内核
- 安全隔离型弱
- 极致执行效率、零开销

裸金属节点
- 容器独立内核
- 安全隔离型强
- 执行效率略低、开销略高

- 容器独立内核
- 极致弹性、免运维
- 执行效率低、开销略高

## Serverless 容器架构思考

不同于标准 K8s，Serverless K8s 与 IaaS 基础设施深度整合，其产品模式更利于公有云厂商通过技术创新，提升规模、效率和能力。在架构层面我们将 Serverless 容器分成容器编排和计算资源池两层，下面我们将对这两层进行深度剖析，分享我们对 Serverless 容器架构和产品背后的关键思考。

## Kubernetes 的成功秘诀

Kubernetes 在容器编排的成功不止得益于 Google 的光环和 CNCF（云原生计算基金会）的努力运作。背后是其在 Google Borg 大规模分布式资源调度和自动化运维领域的沉淀和升华。

其中几个技术要点：

**声明式API：**由于 Kubernetes 采用了声明式的 API。开发者可以关注于应用自身，而非系统执行细节。比如 Deployment, StatefulSet, Job 等不同资源类型，提供了对不同类型工作负载的抽象。对 Kubernetes 实现而言，基于声明式API的 “level-triggered” 实现比 “edge-triggered” 方式可以提供更加健壮的分布式系统实现。

**可扩展性架构：**所有 K8s 组件都是基于一致的、开放的 API 实现、交互。三方开发者也可通过 CRD（Custom Resource Definition）/Operator 等方法提供领域相关的扩展实现，极大提升了 K8s 的能力。

**可移植性：**K8s 通过一系列抽象如 Loadbalance Service, Ingress, CNI, CSI，帮助业务应用可以屏蔽底层基础设施的实现差异，灵活迁移。





## Serverless Kubernetes 的设计原则

Serverless Kubernetes 必须要能兼容 Kubernetes 生态，提供 K8s 的核心价值，此外要能和云的能力深度整合。

1. 用户可以直接使用 Kubernet es的**声明式 API**，兼容 Kubernetes 的应用定义，Deployment, StatefulSet, Job, Service等无需修改

2. 全兼容 Kubernetes 的**扩展机制**，这个很重要，这样才能让 Serverless Kubernetes 支持更多的工作负载。此外 Serverless K8s 自身的组件也是严格遵守 K8s 的状态逼近的控制模式。

3. Kubernetes 的能力尽可能充分利用云的能力来实现，比如资源的调度、负载均衡、服务发现等。根本性简化容器平台的设计，提升规模，降低用户运维复杂性。同时这些实现应该是对用户透明的，保障**可移植性**，让用户现有应用可以平滑部署在 Serverless K8s 之上，也应该允许用户应用混合部署在传统容器和 Serverless 容器之上。

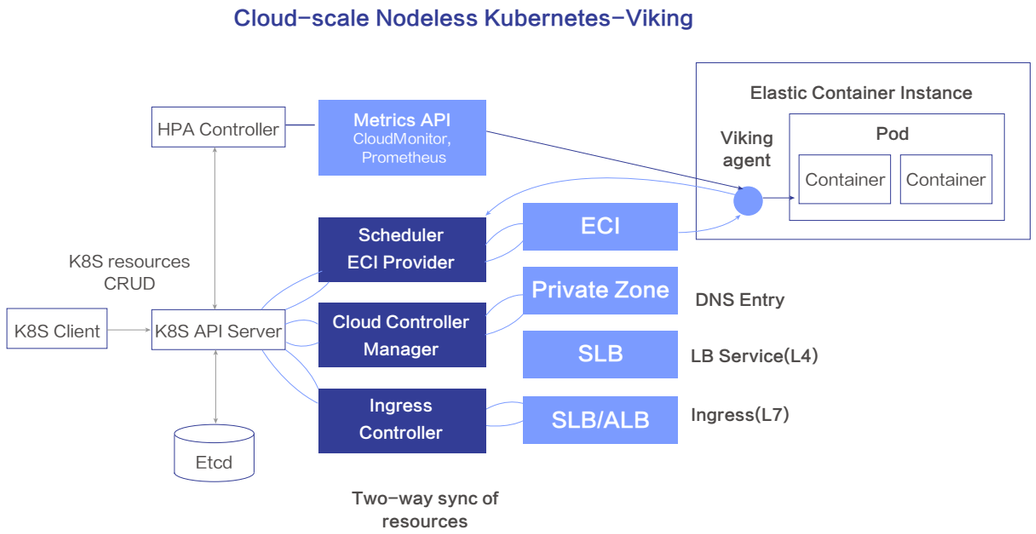
## 从 Node Centric 到 Nodeless

传统的 Kubernetes 采用以节点为中心的架构设计：节点是 pod 的运行载体，Kubernetes 调度器在工作节点池中选择合适的 node 来运行 pod，并利用 Kubelet 完成 对 Pod 进行生命周期管理和自动化运维；当节点池资源不够时，需要对节点池进行扩容，再对容器化应用进行扩容。

对于 Serverless Kubernetes 而言，最重要的一个概念是将容器的运行时和具体的节点运行环境解耦。只有如此，用户无需关注 node 运维和安全，降低运维成本；而且极大简化了容器弹性实现，无需按照容量规划，按需创建容器应用 Pod 即可；此外 Serverless 容器运行时可以被整个云弹性计算基础设施所支撑，保障整体弹性的成本和规模。

在 2017 年底，我们启动 Serverless Kubernetes 项目的时候，我们一直在思考，如果 Kubernetes 天生长在云上，它的架构应该如何设计。我们在现有 Kubernetes 的设计实现上进行了扩展和优化，构建了 Cloud Scale 的 Nodeless K8s 架构。

## 数据面隔离




**Scheduler：**传统 K8s scheduler 的主要功能是从一批节点中选择一个合适的 node 来调度 pod，满足资源、亲和性等多种约束。由于在 Serverless K8s 场景中没有 node 的概念，资源只受限于底层弹性计算库存，我们只需要保留一些基本的 AZ 亲和性等概念支持即可。这样 scheduler 的工作被极大简化，执行效率极大提升。此外我们可以基于新的 scheduler 对 serverless workload 进行更多的编排优化，可以在保证应用可用性的前提下充分降低了计算成本。





**可伸缩性：**K8s 的可伸缩性收到众多因素的影响，其中一个就是节点数量。为了保障 Kubernetes 兼容性，AWS EKS on Fargate 采用 Pod 和 Node 1:1 模型（一个虚拟节点运行一个 Pod），这样将严重限制了集群的可扩展性，目前单集群最多支持 1000 个 pod。我们认为，这样的选择无法满足大规模应用场景的需求。在 ASK 中我们在保持了 Kubernetes 兼容性的同时，解决了集群规模受限于 Node 影响，单集群可以轻松支持 10K Pod。此外传统 K8s 集群中还有很多因素会影响集群的可伸缩性，比如部署在节点上的 kube-proxy，在支持 clusterIP 时，任何单个 endpoint 变更时就会引起全集群的变更风暴。在这些地方 Serverless K8s 可以使用更创新的方法限制变更传播的范围。

**基于云的控制器实现：**通过云的各种能力，可以有效降低系统复杂性，比如：

- 利用 DNS 服务PrivateZone，为容器实例动态配置 DNS 地址解析，支持了 headless service。
- 通过 SLB 提供了提供负载均衡能力。
- 通过 SLB/ALB 提供的 7 层路由来实现 Ingress 的路由规则。

**面向工作负载的深度优化：**未来充分发挥 Serverless 容器的能力，我们需要针对工作负载的特性进行深度优化。

- Knative：Knative 是 Kubernetes 生态下一种 serverless 应用框架，其中 serving 模块支持根据流量自动扩缩容和缩容到 0 的能力。基于 Serverless k8s 能力，我们可以将应用自动缩容到最低成本的容器实例规格，这样可以在保障冷启动时间的 SLA 并有效降低计算成本。此外通过 SLB/ALB 实现了 Ingress Gateway，有效地降低了系统复杂性并降低成本。
- 在 Spark 等大规模计算任务场景下，也通过一些垂直优化手段提高大批量任务的创建效率。

## Serverless 容器基础设施

对于 Serverless 容器而言，我们梳理的客户重点诉求是：

1. 更低的计算成本：弹性成本要低于普通 VM 虚拟机，long run 应用成本要接近虚拟机包年包月。
2. 更高的弹性效率：扩容速度要远高于普通 VM 虚拟机。

3. 更大的弹性规模：与传统节点扩容不同，一个大规模容器应用动辄需要数万核的弹性算力。
4. 持平的计算性能：计算效能需要和同规格虚拟机有一致的性能表现。
5. 更低的迁移成本：与现有容器应用生态完美集成。
6. 更低的使用成本：全自动化安全和运维能力。

## 我们的关键技术选择如下：

### 基于轻量化 Micro VM 的安全容器运行时

对于云产品而言，首先的考虑就是安全性。为此，我们选择基于云原生容器引擎和轻量化 Micro VM 来实现安全、隔离的容器运行时。除了运行时的资源隔离之外，不同客户之间的网络、存储、quota、弹性 SLO 等一系列能力也基于阿里云基础设施，实现了严格的多租隔离。

### 基于 Pod 的基本调度单位和标准、开放的 API 接口

选择基于 Pod 作为 Serverless 容器的基本调度和运行单位，这样可以更加简单地结合上层 Kubernetes 编排系统。此外标准化的 API 屏蔽了底层资源池的具体实现，可以同时可以容纳底层不同形态、不同架构、不同的资源池和生产调度实现。底层架构做的优化和迭代对上层应用和集群编排是无感的。

此外 API 需要拥有足够的通用性支持在多个场景中使用，让用户在云上 k8s、自建 k8s 和混合云场景中都可以充分利用 Serverless 容器的优势和价值。

### 与 IaaS 并池的架构

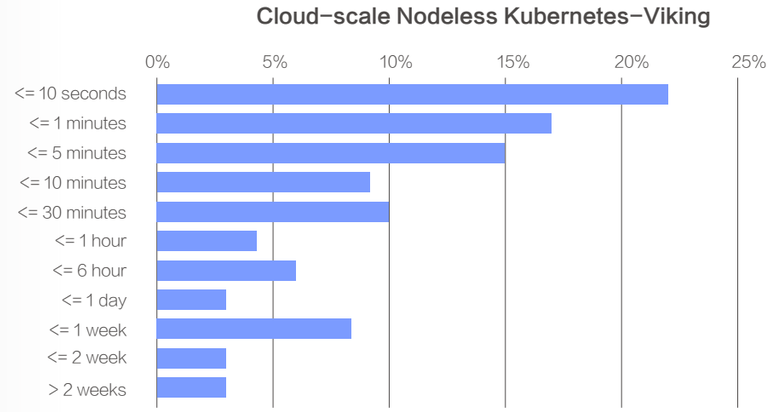
通过与 IaaS 并池我们有能力充分整合云厂商底层弹性计算资源池的算力，包括多种售卖模式（按量，spot，RI，Saving Plan 等），多种机型供应（GPU/vGPU, 新 CPU 架构上线），多样化的存储能力（ESSD，本地盘）等，这样的 Serverless 容器产品在功能，成本和规模上会更有优势，满足客户对计算成本和弹性规模的强诉求。





### Serverless 容器的挑战

Serverless 容器资源创建过程首先是一个计算资源的创建装配过程，是计算、存储、网络多个基础 IaaS 资源的协同装配过程。然而与 VM 虚拟机不同，Serverless 容器有很多独立的挑战。




1. 根据 Sysdig 2019 年容器的调研报告, 超过 50% 的容器生命周期小于 5 分钟。Serverless 容器需要**具备秒级的启动速度**才能满足用户对 Serverless 容器的启动诉求。Serverless 容器自身的启动速度主要受如下因素影响：
2. 底层虚拟化资源的创建和组装：通过端到端管控链路的优化可以将资源准备时间优化到亚秒级。
3. Micro VM 操作系统启动时间。通过对 OS 的**大量剪裁和优化，极大减少了 OS 启动时间。**

镜像下载时间：从 Docker 镜像仓库下载镜像并在本地解压缩是一个非常耗时的操作。下载时间取决于镜像大小，通常在 30 秒到数分钟不等。在传统 Kubernetes 中， worker 节点会在本地缓存已下载过的镜像，这样下次启动不会重复下载和解压。而在 IaaS 的计算存储分离架构下，我们不可能通过传统方式利用本地盘来做容器镜像的缓存。为此我们实现了一个创新的方案：可以将容器镜像制作成一个数据盘快照。当容器实例启动时，如果镜像快照存在，可以直接基于快照创建一个只读数据盘，并随着实例启动自动挂载，容器应用直接利用挂载数据盘作为 rootfs 进行启动。

此外， Serverless 容器的调度相比虚拟机**更关注资源弹性供给的确定性。**Serverless 容器强调按需使用，而虚拟机更多是提前购买预留。在大规模容器创建场景下，单用户单 AZ 弹性 SLO 保障是一个很大的挑战。在电商大促、跨年活动和最近的突发疫情保障的一系列客户支持中，客户对于云平台是否能够提供弹性资源供给确定性 SLO 是极度重视的。此外，结合 Serverless K8s，上层调度器和底层容器实例弹性供给策略配合，我们可以给客户更多对弹性资源供给的控制能力，平衡弹性的成本，规模和持有时间等不同需求维度。

Serverless 容器的**并发创建效率也至关重要。**在高弹性场景，客户要求 30s 内启动 500 pod 副本来支持突发流量，类似 Spark/Presto 等计算业务对并发启动的诉求更大。

为了有效减低计算成本，Serverless 容器应该**提升部署密度。**由于采用 MicroVM 技术，每个容器实例拥有独立的 OS 内核。此外为了兼容 K8s 语义，还有一些辅助进程运行在容器实例内部。这些进程会占用一些资源开销，从而会降低 Serverless 的部署密度。我们需要将共性的开销下沉到基础设施之上，甚至通过软硬一体的方式来进行卸载。

## 未来展望

在预期范围内各大云厂商将会持续投入 Serverless 容器方向来加大容器服务平台的差异化。上面已经提到成本、兼容性、创建效率和弹性供给保障是 serverless 容器技术重要的硬核能力。

我们将持续努力建设云原生 IaaS 基础设施， 一方面进一步优化现有 Serverless 容器产品形态，给客户一个更低成本、更好的使用体验，更佳的兼容性的 Serverless K8s 产品。一方面基于 Serverles 容器，未来也可以帮助上层应用有更多发展和创新空间。Cloud Native First! Serverless First！是我们前进的方向。





# 2/5 Serverless 场景下 Pod 创建效率优化

**阿里云技术专家　张翼飞**

作者简介：张翼飞 就职于阿里云容器服务团队，主要专注 Serverless 领域的产品研发。

众所周知，Kubernetes 是云原生领域的基石，作为容器编排的基础设施，被广泛应用在 Serverless 领域。弹性能力是 Serverless 领域的核心竞争力，本次分享将重点介绍基于 Kubernetes 的 Serverless 服务中，如何优化 Pod 创建效率，提升弹性效率。

## Serverless 计算简介

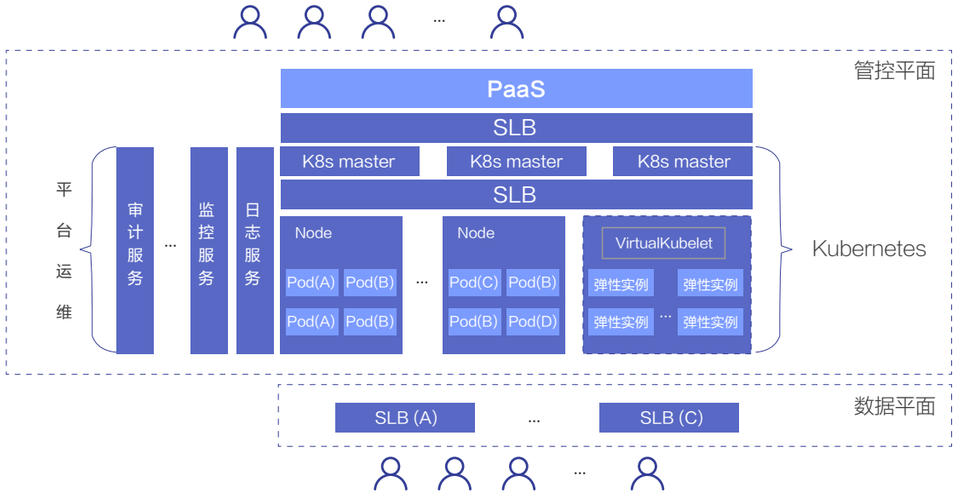
在进入主题之前，先简单回顾下 Serverless 计算的定义。

从维基百科可以了解到，Serverless 计算是云计算的一种形态，由云厂商管理服务器，向用户动态分配机器资源，基于实际使用的资源量计费。

用户构建和运行服务时，不用考虑服务器，降低了用户管理服务器的负担。在业务高峰期通过平台的弹性能力自动扩容实例，在业务低峰期自动缩容实例，降低资源成本。

## Serverless 计算平台

下述是当前常见的 Serverless 计算产品的架构。




整个产品架构通常会有管控平面和数据平面两层，管控平面服务开发者，管理应用生命周期，满足开发者对应用管理的需求，数据平面服务应用的访问方，如开发者业务的用户，满足应用的流量管理和访问诉求。

管控平面通常采用 Kubernetes 做资源管理和调度，master 通常是 3 节点，满足对高可用的需求，节点通过内网 SLB 访问 K8s master。

在节点层面，通常会有两种类型的节点：





- 一种是运行 kubelet 的节点，如裸金属服务器、虚拟机等，这类节点上会运行安全容器作为 Pod 运行时，每个 Pod 拥有独立的 kernel，降低共享宿主机 kernel 带来的安全风险。同时会通过云产品 VPC 网络或其他网络技术，在数据链路层隔离租户的网络访问。通过 安全容器+二层网络隔离，单个节点上可以提供可靠的多租运行环境
- 还有一种是虚拟节点，通过 VirtualKubelet 衔接 K8s 和弹性实例。弹性实例是云产品中类似虚拟机的一种轻量资源形态，提供无限资源池的容器组服务，该容器组的概念对应 K8s 中的 Pod 概念。AWS 提供有 Fargate 弹性实例，阿里云提供有 ECI 弹性实例

Serverless 产品会提供基于 K8s 的 PaaS 层，负责向开发者提供部署、开发等相关的服务，屏蔽 K8s 相关的概念，降低开发者开发、运维应用的成本。

在数据平面，用户可通过 SLB 实现对应用实例的访问。PaaS 层也通常会在该平面提供诸如流量灰度、A/B 测试等流量管理服务，满足开发者对流量管理的需求。

弹性能力是 Serverless 计算平台的核心竞争力，需要满足开发者对 Pod 规模 的诉求，提供类似无限资源池的能力，同时还要满足 创建 Pod 效率 的诉求，及时响应请求。

Pod 规模可通过增加 IaaS 层资源来满足，接下来重点介绍提升 Pod 创建效率的技术。

## Pod 创建相关场景

先了解下 Pod 创建相关的场景，这样可以更有效通过技术满足业务诉求。

业务中会有两种场景涉及到 Pod 创建：

- 第一种是创建应用，这个过程会先经过调度，决策最适合 Pod 的节点，然后在节点上创建 Pod
- 第二种是升级应用，在这个过程中，通常是不断进行 创建新 Pod 和 销毁旧 Pod

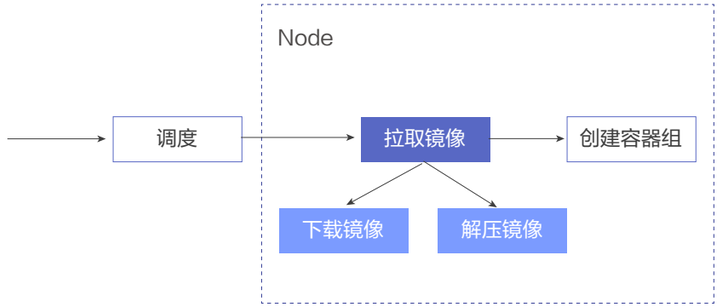
Serverless 服务中，开发者关心的重点在于应用的生命周期，尤其是创建和升级阶段，Pod 创建效率会影响这两个阶段的整体耗时，进而影响开发者的体验。面对突发流量时，创建效率的高低会对开发者服务的响应速度产生重要影响，严重者会使开发者的业务受损。

面对上述业务场景，接下来重点分析如何提升 Pod 创建效率。

## 创建 Pod 流程

整体分析下 Pod 创建的阶段，按照影响 Pod 创建效率的优先级来依次解决。

这是简化后的创建 Pod 流程：




当有 Pod 创建请求时，先进行调度，为 Pod 选取最合适的节点。在节点上，先进行拉取镜像的操作，镜像在本地准备好后，再进行创建容器组的操作。在拉取镜像阶段，又依次分为下载镜像和解压镜像两个步骤。





**我们针对两种类型的镜像进行了测试，结果如下：**

| 镜像 | 解压前大小 | 解压后大小 | (内网) 拉取镜像总耗时 | 下载镜像耗时 | 解压镜像耗时 |
|---|---|---|---|---|---|
| golang:1.10 | 248.76 MB | 729.90 M | 16.97s | 3.90s (占比 22.98% ) | 13.07s (占比 77.02%) |
| bde2020/hadoop-namenode:2.0.0-hadoop3.1.1-java8 | 506.87 MB | 1.29 GB | 38.87s | 23.26s (占比 59.84%) | 15.61s (占比 40.16%) |

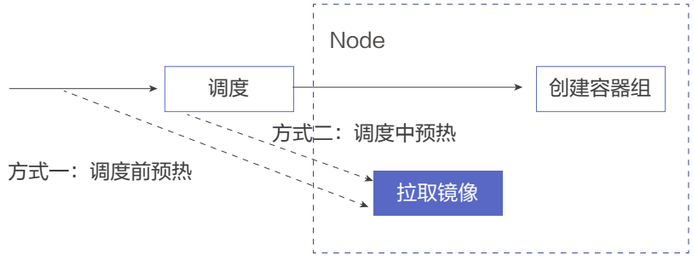
从测试结果可看到，解压镜像耗时在整个拉取镜像过程中的占比不容忽视，对于解压前 248MB 左右的 golang:1.10 镜像，解压镜像耗时竟然占到了拉取镜像耗时的 77.02%，对于节解压前 506MB 左右的 hadoop namenode 镜像，解压镜像耗时和下载镜像耗时各占 40% 和 60% 左右，即对于拉取镜像过程的总耗时也不容忽视。

接下来就分别针对上述过程的不同节点进行优化处理，分别从上述整个流程、解压镜像、下载镜像等方面进行探讨。

## 拉取镜像效率提升

### 镜像预热

可以快速想到的方法是进行镜像预热，在 Pod 调度到节点前预先在节点上准备好镜像，将拉取镜像从创建 Pod 的主链路中移除，如下图：



可以在调度前进行全局预热，在所有节点上行提前拉取镜像。也可以在调度过程中进行预热，在确定调度到的节点后，在目标节点上拉取镜像。

两种方式无可厚非，可根据集群实际情况进行选择。

社区里 OpenKruise 项目即将推出镜像预热服务，可以关注下。下述是该服务的使用方式：

```yaml
apiVersion: apps.kruise.io/v1alpha1
kind: ImagePullJob
metadata:
  name: golang
spec:
  completionPolicy:
    activeDeadlineSeconds: 300
    ttlSecondsAfterFinished: 600
  image: registry-vpc.cn-beijing.aliyuncs.com/test/golang:1.10
  parallelism: 10
  pullSecrets:
  - test
  selector:
    names:
    - i-2zed6ptcmrk9que37wq0
    - i-2zed6ptcmrk9que37wq1
    - i-2zed6ptcmrk9que37wq2
```

通过 ImagePullJob CRD 下发镜像预热任务，指定目标镜像和节点，可配置拉取的并发度、Job 处理的超时时间以及 Job Object 自动回收的时间。若是私有镜像，可指定拉取镜像时的 secret 配置。ImagePullJob 的 Events 会提镜任务的状态信息，可考虑适当增大 Job Object 自动回收的时间，便于通过 ImagePullJob Events 查看任务的处理状态

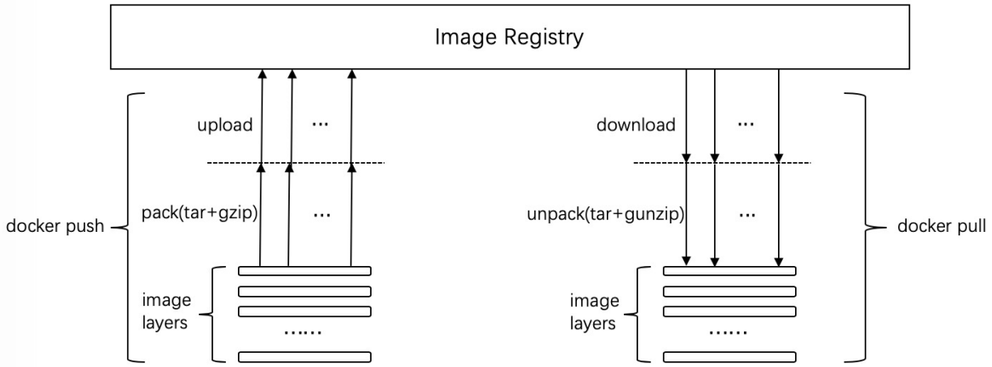




## 提升解压效率

从刚才看到的拉取镜像的数据来看，解压镜像耗时会占拉取镜像总耗时很大的比例，测试的例子最大占比到了77%，所以需要考虑如何提升解压效率。

先回顾下 docker pull 的技术细节：




在 docker pull 时，整体会进行两个阶段：

1. 并行下载 image 层
2. 拆解 image 层

在解压 image 层时，默认采用的 gunzip。

再简单了解下 docker push 的过程：

1. 先对 image 层进行打包操作，这个过程中会通过 gzip 进行压缩
2. 然后并行上传

gzip/gunzip 是单线程的压缩/解压工具，可考虑采用 pigz/unpigz 进行多线程的压缩/解压，充分利用多核优势。

containerd 从 1.2 版本开始支持 pigz，节点上安装 unpigz 工具后，会优先用其进行解压。通过这种方法，可通过节点多核能力提升镜像解压效率。

这个过程也需要关注 下载/上传 的并发度问题，docker daemon 提供了两个参数来控制并发度，控制并行处理的镜像层的数量，--max-concurrent-downloads 和 --max-concurrent-uploads。默认情况下，下载的并发度是 3，上传的并发度是 5，可根据测试结果调整到合适的值。

使用 unpigz 后的解压镜像效率：

| 镜像 | 解压前大小 | 解压后大小 | (内网) 拉取镜像总耗时 | 下载镜像耗时 | 解压镜像耗时 | 解压效率提升 |
|---|---|---|---|---|---|---|
| golang:1.10 | 248.76 MB | 729.90 M | 16.97s | 3.90s | Gunzip: 13.07s | 35.88% |
| | | | 11.98s | 3.60s | Unpigz: 8.38s | |
| bde2020/hadoop-namenode:2.0.0-hadoop3.1.1-java8 | 506.87 MB | 1.29 GB | 38.87s | 15.60s | Gunzip:23.27s | 16.41% |
| | | | 29.35s | 9.90s | Unpigz 19.45s | |

在相同环境下，golang:1.10 镜像解压效率提升了 35.88%，hadoop namenode 镜像解压效率提升了 16.41%。

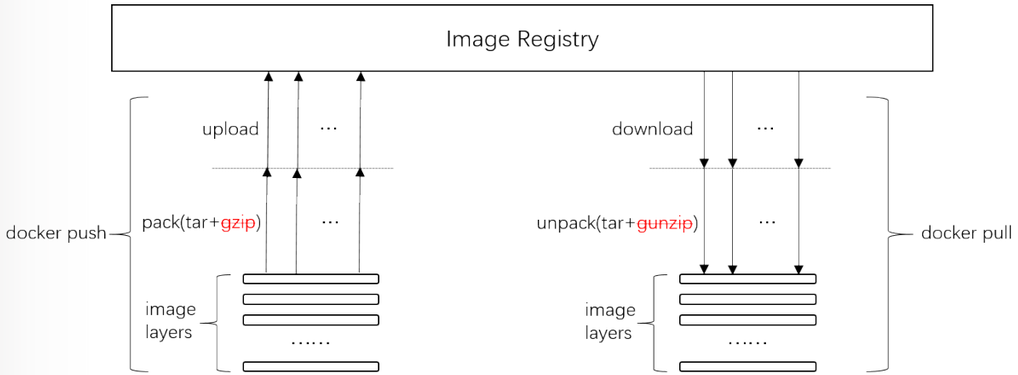
## 非压缩镜像

通常内网的带宽足够大，是否有可能省去 解压缩/压缩 的逻辑，将拉取镜像的耗时集中在下载镜像方面？即适量增大下载耗时，缩短解压耗时。

再回顾下 docker pull/push 的流程，在 unpack/pack 阶段，可以考虑将 gunzip 和 gzip 的逻辑去掉：








对于 docker 镜像，若 docker push 时的镜像是非压缩的，则 docker pull 时是无需进行解压缩操作，故要实现上述目标，就需要在 docker push 时去掉压缩逻辑。

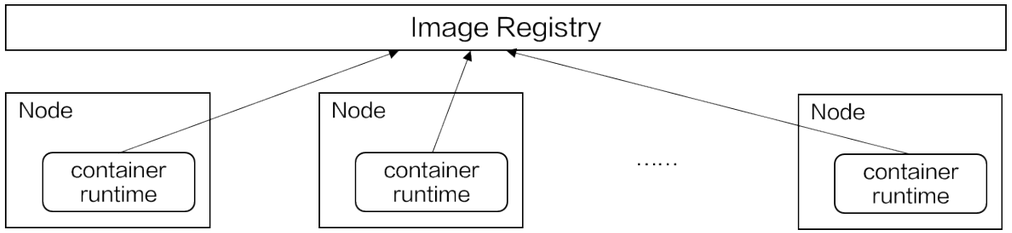
docker daemon 暂时不支持上述操作，我们对 docker 进行了一番修改，在上传镜像时不进行压缩操作，测试结果如下：

| 镜像 | 解压前大小 | 解压后大小 | (内网) 拉取镜像总耗时 | 下载镜像耗时 | 解压镜像耗时 | 解压效率提升 |
|---|---|---|---|---|---|---|
| golang:1.10 | 248.76 MB | 729.90 M | 16.97s | 3.90s | 13.07s | 49.81% |
| | | | 13.16s | 6.60s | 6.56s | |
| bde2020/hadoop-namenode:2.0.0-hadoop3.1.1-java8 | 506.87 MB | 1.29 GB | 38.87s | 15.60s | 23.27s | 27.89% |
| | | | 33.28s | 16.50s | 16.78s | |

这里重点关注解压镜像耗时，可看到 golang:1.10 镜像解压效率提升了 50% 左右，hadoop namenode 镜像解压效率替身挂了 28% 左右。在拉取镜像总耗时方面，该方案有一定的效果。

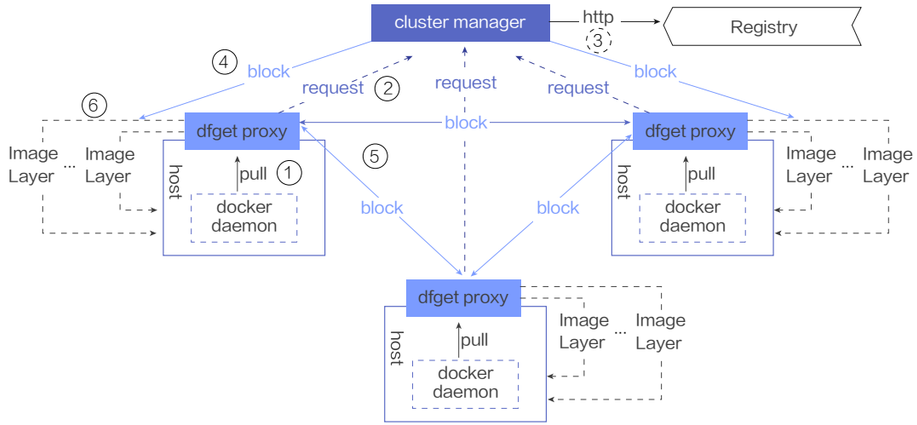
## 镜像分发

小规模集群中，提升拉取镜像效率的重点需要放在提升解压效率方面，下载镜像通常不是瓶颈。而在大规模集群中，由于节点数众多，中心式的 Image Registry 的带宽和稳定性也会影响拉取镜像的效率，如下图：




下载镜像的压力集中在中心式的 Image Registry 上。

这里介绍一种基于 P2P 的镜像分发系统来解决上述问题，以 CNCF 的 DragonFly 项目为例：








这里有几个核心组件：

### ClusterManager

- 它本质上是一个中心式的 SuperNode，在 P2P 网络中作为 tracker 和 scheduler 协调节点的下载任务。同时它还是一个缓存服务，缓存从 Image Registry 中下载的镜像，降低节点的增加对 Image Registry 带来的压力

### Dfget

- 它既是节点上下载镜像的客户端，同时又充当向其他节点提供数据的能力，可以将本地已有的镜像数据按需提供给其他节点

### Dfdaemon

- 在每个节点上有个 Dfdaemon 组件，它本质上是一个 proxy，对 docker daemon 的拉取镜像的请求实现透明代理服务，使用 Dfget 下载镜像

通过 P2P 网络，中心式的 Image Registry 数据被缓存到 ClusterManager 中，ClusterManager 协调节点对镜像的下载需求，将下载镜像的压力分摊到集群节点上，集群节点既是镜像数据的拉取方，又是镜像数据的提供方，充分利用内网带宽的能力进行镜像分发。

## 按需加载镜像

除了上述介绍到的方法，是否还有其他优化方法?

当前节点上创建容器时，是需要先把镜像全部数据拉取到本地，然后才能启动容器。再考虑下启动虚拟机的过程，即使是几百 GB 的虚拟机镜像，启动虚拟机也通常是在秒级别，几乎感受不到虚拟机镜像大小带来的影响。

那么容器领域是否也可以用到类似的技术?

再看一篇发表在 usenix 上的题为《Slacker: Fast Distribution with Lazy Docker Containers》的 paper 描述:

Our analysis shows that pulling packages accounts for 76% of container start time, but only 6.4% of that data is read.

该 paper 分析，在镜像启动耗时中，拉取镜像占比 76%，但是在启动时，仅有 6.4% 的数据被使用到，即镜像启动时需要的镜像数据量很少，需要考虑在镜像启动阶段按需加载镜像，改变对镜像的使用方式。

对于「Image 所有 layers 下载完后才能启动镜像」，需要改为启动容器时按需加载镜像，类似启动虚拟机的方式，仅对启动阶段需要的数据进行网络传输。

但当前镜像格式通常是 tar.gz 或 tar，而 tar 文件没有索引，gzip 文件不能从任意位置读取数据，这样就不能满足按需拉取时拉取指定文件的需求，镜像格式需要改为可索引的文件格式。

Google 提出了一种新的镜像格式，stargz，全称是 seeable tar.gz。它兼容当前的镜像格式，但提供了文件索引，可从指定位置读取数据。

传统的 .tar.gz 文件是这样生成的: Gzip(TarF(file1) + TarF(file2) + TarF(file3) + TarFooter))。分别对每个文件进行打包，然后对文件组进行压缩操作。

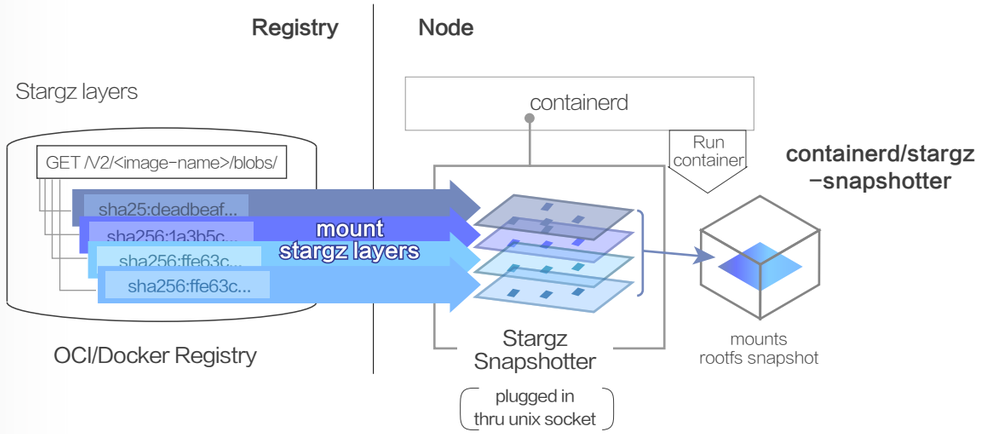
stargz 文件做了这样的创新：Gzip(TarF(file1)) + Gzip(TarF(file2)) + Gzip(TarF(file3_chunk1)) + Gzip(F(file3_chunk2)) + Gzip(F(index of earlier files in magic file), TarFooter)。针对每个文件进行打包和压缩操作，同时形成一个索引文件，和 TarFooter 一起进行压缩。

这样就可以通过索引文件快速定位要拉取的文件的位置，然后从指定位置拉取文件。

然后在 containerd 拉取镜像环节，对 containerd 提供一种 remote snapshotter，在创建容器 rootfs 层时，不通过先下载镜像层再构建的方式，而是直接 mount 远程存储层，如下图所示:







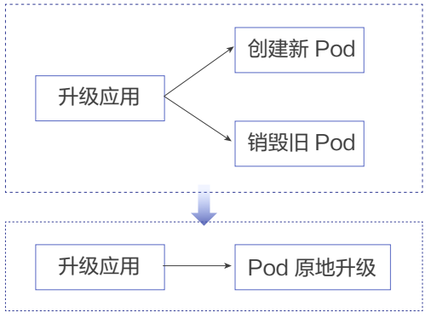

要实现这样的能力，一方面需要修改 containerd 当前的逻辑，在 filter 阶段识别远程镜像层，对于这样的镜像层不进行 download 操作，一方面需要实现一个 remote snapshotter，来支持对于远程层的管理。

当 containerd 通过 remote snapshotter 创建容器时，省去了拉取镜像的阶段，对于启动过程中需要的文件，可对 stargz 格式的镜像数据发起 HTTP Range GET 请求，拉取目标数据。

阿里云实现了名为 DADI 的加速器，类似上述的思想，目前应用在了阿里云容器服务，实现了 3.01s 启动 10000 个容器，完美杜绝了冷启动的漫长等待。感兴趣的读者也参考该文章：https://developer.aliyun.com/article/742103

## 按需加载镜像

上述都是针对创建 Pod 过程提供的技术方案，对于升级场景，在现有的技术下，是否有效率提升的可能性？是否可以达到下述效果，即免去创建 Pod 的过程，实现 Pod 原地升级？




在升级场景中，占比较大的场景是仅升级镜像。针对这种场景，可使用 K8s 自身的 patch 能力。通过 patch image，Pod 不会重建，仅目标 container 重建，这样就不用完整经过 调度+新建 Pod 流程，仅对需要升级的容器进行原地升级。

在原地升级过程中，借助 K8s readinessGates 能力，可以控制 Pod 优雅下线，由 K8s Endpoint Controller 主动摘除即将升级的 Pod，在 Pod 原地升级后加入升级后的 Pod，实现升级过程中流量无损。

OpenKruise 项目中的 CloneSet Controller 提供了上述能力：





```yaml
apiVersion: apps.kruise.io/v1alpha1
kind: CloneSet
metadata:
  labels:
    app: sample
  name: sample
spec:
  replicas: 5
  selector:
    matchLabels:
      app: sample
  template:
    metadata:
      labels:
        app: sample
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
```

开发者使用 CloneSet 声明应用，用法类似 Deployment。在升级镜像时，由 CloneSet Controller 负责执行 patch 操作，同时确保升级过程中业务流量无损。

## 小结

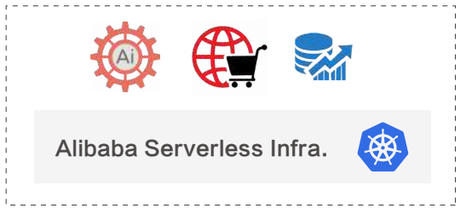
从业务场景出发，我们了解了提升 Pod 创建效率带来收益的场景。然后通过分析 Pod 创建的流程，针对不同的阶段做相应的优化，有的放矢。

通过这样的分析处理流程，使得可以有效通过技术满足业务需求。

# 2/6 面向 SLO 的资源估计调度以优化利用率

阿里云技术专家　余英豪　|　阿里云技术专家　王双

随着阿里经济体的上云，越来越多的业务进行了云原生的改造，运行在基于 Kubernetes 原生语义的阿里巴巴 Serverless 基础设施上。集团内部的泛交易在线电商业务和越来越多的离线作业框架包括大数据和机器学习负载进行了云原生的混部



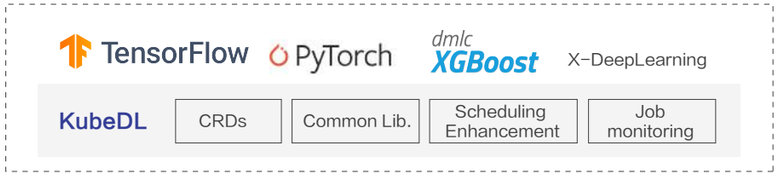

由于业务属性的不同，不同作业的服务水平目标（SLO）有很大的差异。例如，直接面向终端用户的电商业务对服务时延十分敏感，而离线作业在意则是吞吐量和完成基线。如何在满足不同业务的 SLO 的基础上，最大化的优化资源效率，是云原生混部的一个重要议题。





## 统一的 AI 作业 Operator -- KubeDL

在分享我们在阿里巴巴场景中的解决方案之前，先介绍一下我们从内部孵化并已开源的 AI 作业统一管理框架，KubeDL。AI 作业是混部在 Serverless Kubernetes 集群中的重要增量业务之一，阿里使用了业界流行的 AI 框架如 Tensorflow，以及自研的新型框架来支持丰富的业务场景。由于作业框架的多样性、生命周期管理的复杂性，如何将种类繁多的 AI 框架统一的管理起来，并且在 Kubernetes 上面方便的迭代升级，需要有一个通用的 job 层 Operator 来进行统一的管理，这便是 KubeDL 设计的初衷。




不同于 Kubeflow per-framework-per-operator 的形式，KubeDL 将多种主流框架的AI作业的调度管理逻辑抽象为可复用的通用库，做到一次开发、一次升级，减少运维和部署的压力，统一作业监控管理。除此之外，我们还开发了更多的能力增强，例如解耦镜像和训练代码，集成了作业元数据持久化功能以及监控作业全生命周期的 Dashboard，进一步降低在 Kubernetes 集群中运行 AI 作业的使用门槛。

欢迎对 AI on Kubernetes 感兴趣的小伙伴关注我们的开源社区：https://github.com/alibaba/kubedl。

## QoS 和优先级等级交付 SLO 差异化的资源保障

业务 SLO 的差异性直接影响了资源调度和保障的不同。例如，对于追求低时延而流量实时变化的在线服务，除了运行时要给与充分的资源保障降低干扰之外，还会配置一定的冗余资源来应对突发的流量。大数据和 AI 相关的离线作业等，在不破坏完成基线的前提下，可以接受一定程度的超卖资源及必要时刻的资源退让，来提升整体效率。

为此，我们通过 QoS 等级和优先级的设置，交付不同 SLO 等级的资源。

- **QoS 等级：**QoS 等级决定了业务在运行时的资源保障程度。在社区定义中，QoS 等级影响了资源的 limit 和 request 的设置，本质上其实是影响了是否可以使用超卖的资源。在这个基础上，我们进一步约束了 CPU 绑核的要求，例如，对于 CPU 竞争敏感型的业务，会设置一个独占 CPU 核不进行共享的 QoS 等级。
- **优先级等级：**优先级等级决定了业务获得资源的机会。在社区定义中，优先级等级影响了调度、抢占顺序，以及在资源紧张时的驱逐顺序，而在内存不足时 OOM kill 的优先顺序则和 QoS 等级绑定，这导致了资源不足（out-of-resource）时行为的不确定性。我们重新设计了 OOM 的用户态打分逻辑（oom-score-adj），和优先级进行了挂钩，并保证了驱逐和 OOM 顺序的一致性。

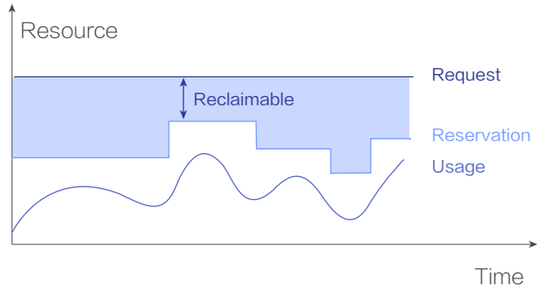
## 资源水位的实时估计

通过 QoS 和优先级等级的设置，我们明确了哪些业务可以使用超卖资源，以及超卖后发生资源紧张时的保障顺序。不过，我们仍然需要尽量避免出现资源紧张的情况，导致频繁地压制和驱逐低优先级的超卖任务，以保障超卖的稳定性。

混部方案通常是将已分配给高优先保障业务的资源中尚未使用的部分，超卖出来给低保障业务来使用。为了超卖的稳定性，我们需要准确地估计可以从高优先保障业务的资源中稳定超卖出去的资源。我们根据过去一段时间内业务的资源使用数据，来预测未来一段时间的水位；扣除这部分的预估需求之外的部分，是可以稳定超卖的资源。下图展示了这个逻辑。







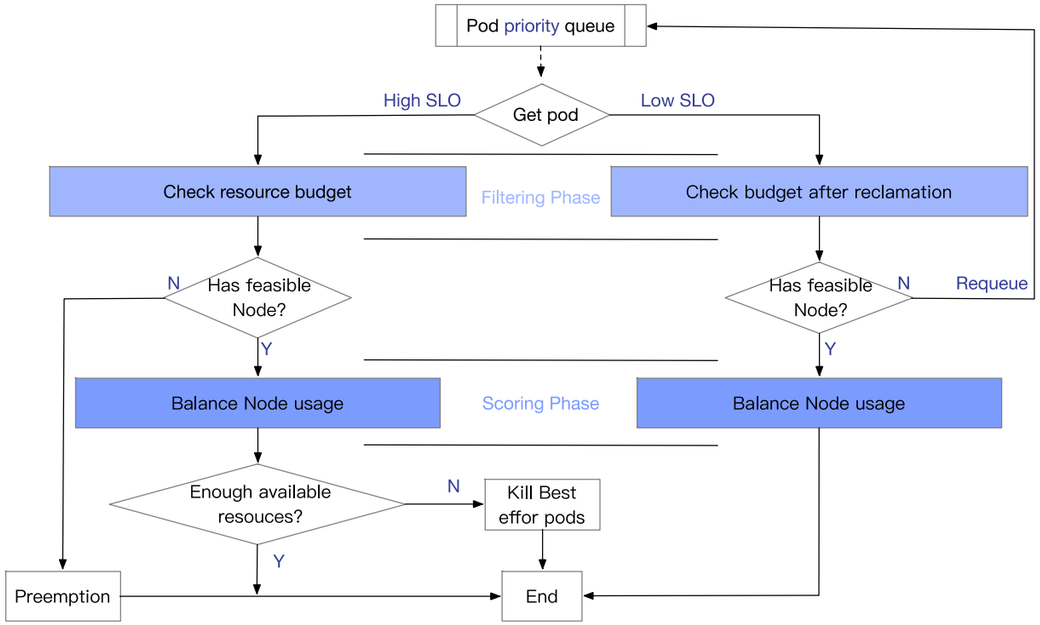

为了保障水位估计的稳定性，我们会实时收集业务的资源真实使用量，基于历史的数据去预估未来一段时间内业务的资源水位。当然，基于历史数据的估计无法准备预测可能发生的突发资源申请。为了防止不准确的估计导致超卖不稳定，我们会在实时水位估计值的基础上增加一定的 buffer，并根据实时变化及时调整估计值。

## 均衡水位的动态调度

在实时水位预估的基础上，我们进行均衡水位的动态调度，防止热点出现。下图展示了我们的动态调度逻辑。

在调度账本上，高 SLO 保障pod走正常非超卖调度账本，低 SLO 保障的 pod 则会考虑用回收的资源来进行调度。在选择节点的打分阶段，我们统一增加一个均衡实时水位的策略。因为在账本扣减阶段，高 SLO 的 pod 并没有感知低 SLO 业务的账本，这个时候如果不考虑低 SLO 业务的水位，可能会调度到利用率平均水位较高的节点上，导致不必要的热点和竞争。

在整体集群水位比较高是，不可避免可能会需要把高 SLO 的 pod 调度到一个水位比较高的节点上。此时，我们增加了一个主动驱逐的逻辑，预先驱逐低 SLO 的 pod，这样可以避免 pod 真正拉起后，造成资源竞争甚至 OOM。

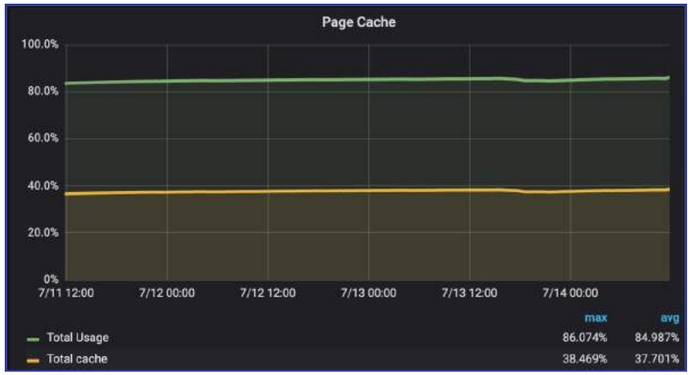



## 冷内存的扫描和回收扩大超卖空间

在动态资源调度的实践中，我们发现内存往往会成为超卖的瓶颈，阻止我们进一步提高资源的利用率。主要的原因是 Linux 内存管理的策略：尽量缓存所有使用过的页，加快访问速度，通过这种空间换时间的方式来解决性能问题。这样带来的影响是资源利用率的降低：有大量不活跃的 page cache 占据着内存，而且无法超卖出去。下图展示了我们一个线上集群的内存使用情况，可以看到内存使用量一直是处在一个比较高的位置，其中黄线是 page cache 的总量，几乎占据了全部内存使用量的一半，其中又有相当的部分，是不再活跃的 page。








因此，我们需要一个机制去识别冷的 page cache，并进行回收，以在不影响业务性能的基础上扩大超卖的收益。社区有一个比较成熟的技术，idle page tracking，但是由于扫描系统开销太大，无法大规模落地使用。为此，我们的内核团队开发了一个内核模块，可以做到轻量级扫描和鉴定内存页的活跃程度，并释放不活跃的 page cache。上线后，整体冷内存回收效果可以达到整机可用内存的 10-20%。

## 总结

本文介绍了我们在阿里巴巴的云原生 Serverless 基础设施上，如何支撑类型多样、SLO 差异化的业务。从上到下，我们开发了一个通用的 AI job operator - KubeDL， 来简化AI作业的部署和管理；基于社区定义增强 QoS 和优先级等级的设计，来交付不同级别的资源 SLO；建设了一条实时水位估计的链路，结合动态均衡的调度策略来稳定超卖资源；同时依赖冷内存的回收来扩大超卖收益。

整套方案上线后，我们将一个较大规模集群的资源利用率提升了 10%。整体性能比较稳定，低 SLO 业务并没有对高 SLO 业务造成影响。我们遇到的主要挑战是如何安全地超卖不可压缩的内存资源。未来，我们将继续探索，做到比较大规模的稳定内存超卖，进一步提高资源效率，扩大端到端超卖收益。

# 3/1 Dragonfly：在云原生高效、安全的进行镜像分发

阿里云开发工程师　刘裕惺　|　阿里云高级技术专家　马介悦

首先欢迎大家来到 Dragonfly 的 Introduction，我是来自阿里云的刘裕惺，是 Dragonfly 的 Maintainer 之一。今天我将和来自蚂蚁集团的马介悦一起为大家带来分享。

相信今天大家来到这个房间，一定也遇到过如下的困扰：在云原生生产环境中，随着业务规模的扩大，机器数量在不断的增加，业务镜像 的 size 以及数量也在逐渐的增加。随着带来的，镜像分发的时间开始逐渐的影响业务的发展，甚至成为瓶颈。如果对以上的场景，你深有体感，那么恭喜你，你今天来对了地方。

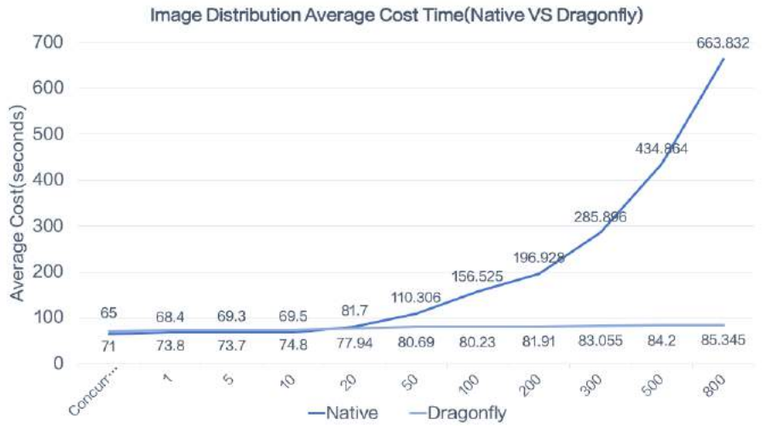





我们先来看一下今天分享的 Agenda。首先，会由我简单的分享一下 Dragonfly 项目技术以及社区的现状和发展，让您初步了解一下这个项目；然后，会由马介悦分享一下我们在 OCIV2 背景下的一些思考以及探索。

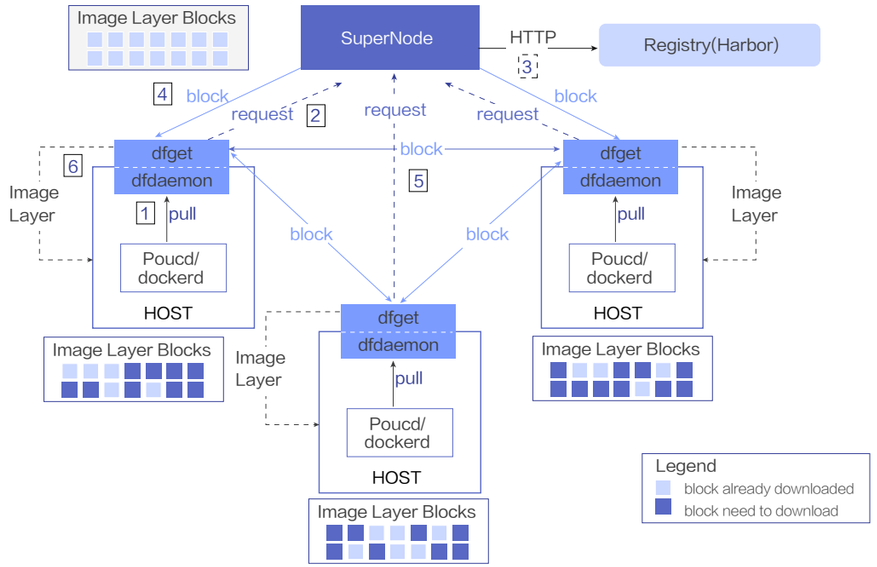
## Part One – Intro

### What & Why




首先，我们先看一下 Dragonfly 是什么？这里我抽取了几个关键词。Dragonfly 是一个基于 P2P 技术的，高可靠的，高性能的镜像分发基础设施。其所要解决的主要问题就是在大规模场景下镜像分发的瓶颈问题。我们可以看一下 PPT 中右边这张图，这张图针对不同节点规模下，通过传统镜像分发模式与通过 Dragonfly 进行分发的模式，其分发时间的变化以及对比。其中，横轴表示节点数量，纵轴表示分发的总时间。我们可以看到，在传统镜像分发模式下，随着节点数量的增加，分发完成的总时间，在指数形式的增长；而通过 Dragonfly 进行分发，随着节点数量的增加，分发完成的时间并没有发生剧烈的变化，其相对于传统镜像分发模式的优势愈加明显。

## P2P Mechanism




1. Pull image from node proxy
2. Send pulling requests to SuperNode
3. Cache image from Registry if non-exist
4. Reply to node peers which have blocks
5. Transport blocks among all peers
6. Finish whole pulling when all blocks downloaded





那么 Dragonfly 是如何做到这样的优化呢？这里简单的介绍一下 Dragonfly 的分发流程。首先，我们明确一点，Dragonfly 存在两种系统组件，第一，server 组件，也就是图中的 supernode 组件；第二，proxy 组件，也就是途中的 dfget+dfdaemon。那么我们从一个多节点下拉镜像的流程来给大家展示一下 Dragonfly 的设计。

1. 容器引擎将下拉镜像的请求转发到每个节点上的 proxy 组件。
2. proxy 组件将请求转发给 supernode（supernode 会进行统一调度），并期待返回可用的 Peer 以及接下来需要下载的分片。
3. supernode 组件拥有两个职责：1. P2P 调度决策 2. CDN 缓存。因此，supernode 收到一个文件下载请求的时候，会首先判断本地是否有该文件的缓存，如果没有，则从源下拉源文件，并在本地进行缓存。
4. 在 supernode 有了缓存之后，就会开始进行 P2P 的调度策略执行，并返回给 peer 节点去哪个 peer 上下载哪个 block。这里的 block 指得是一个镜像文件被切分后的一个分片，这么做的原因是为了一个镜像文件可以被更快的进行并发下载以及 P2P 分发。
5. 接下来，在 supernode 的调度下，所有的文件分片将在所有节点之间进行共享和传输。
6. 最后，就可以通过 P2P 的方式，快速的完成多台机器针对同一个镜像的下载。

## How to Use

1. Deploy SuperNode
2. Deploy dfclient on every client
3. Config Container Engine Daemon
4. Pull images with Dragonfly

- Docker Registry Mirrors

```
--registry-mirror=https://<your awesome mirror>
```

- Containerd CRI Registry Mirrors

```
[plugins.cri.registry.mirrors]
  [plugins.cri.registry.mirrors."docker.io"]
    endpoint = ["https://registry-1.docker.io"]
  [plugins.cri.registry.mirrors."test.secure-registry.io"]
    endpoint = ["https://HostIP1:Port1"]
  [plugins.cri.registry.mirrors."test.insecure-registry.io"]
    endpoint = ["http://HostIP2:Port2"]
```

- Add redirection in your registry

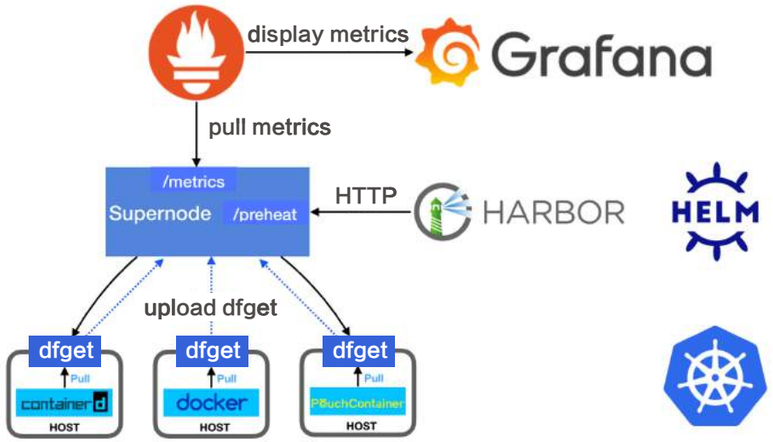
在了解了 Dragonfly 的基本流程以及其性能优势后，我们应该如何使用 Dragonfly 的快速分发能力呢?

1. 部署 supernode 组件
2. 在每台机器上部署 proxy 组件
3. 配置容器引擎，不同的容器引擎有不同的配置（如下图中的 Docker/Containerd 等），配置的作用是通知容器引擎，将下拉镜像的请求转发到机器上启动的 proxy 进程上；
4. 这样通过容器引擎发出的下拉镜像的请求就可以通过 Dragonfly 进行快速分发了。

# Part Two – Status

接下来，我将简单为大家介绍一下 Dragonfly 社区的一些情况以及进展。

## Ecology







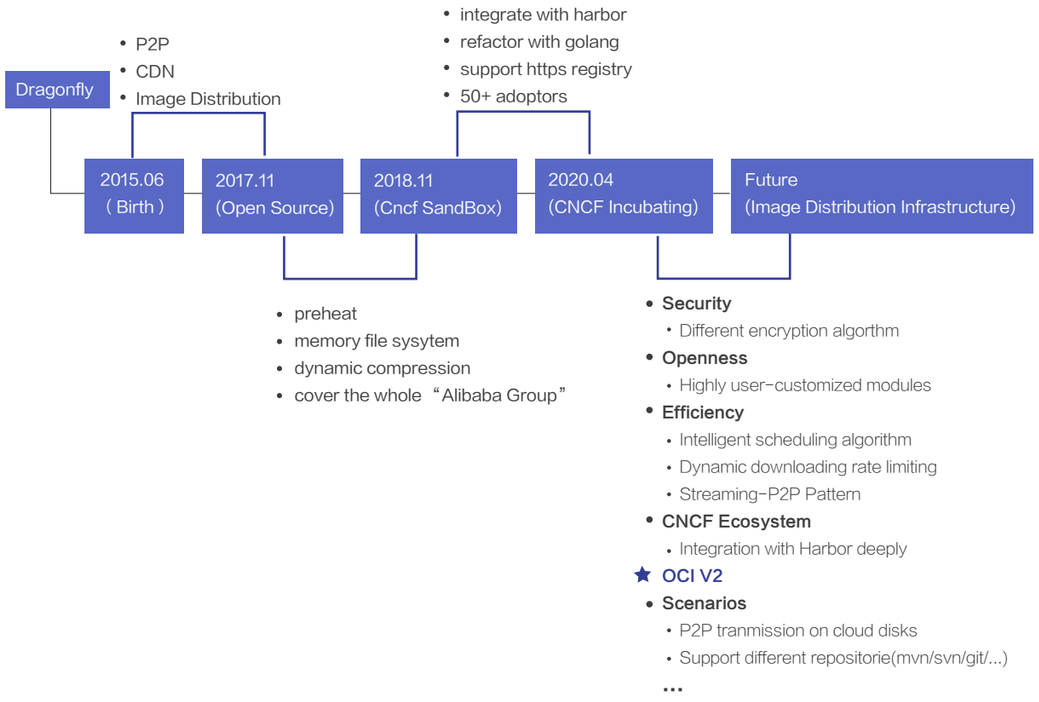

首先Dragonfly 作为云原生生态的一部分，其与社区中很多项目也进行了深度的集成。

- 面向下游，容器引擎方面，Dragonfly 无侵入支持Docker，Containerd，PouchContainer 等容器引擎
- 面向上游，镜像仓库方面，Dragonfly 通过提供 Preheat（即预热）的能力与 Harbor 进行了深度的集成
- 在可观测性方面，Dragonfly 支持 Prometheus，Grafana 的集成
- 在运维部署能力方面，Dragonfly 支持通过 helm 的方式在 k8s 集群上一键化的安装 Dragonfly 集群

## Community Situation

Dragonfly 目前已经经过了很多的实践验证。其中，在阿里巴巴，中国移动，bilibili 等都已经有了上千生产节点的规模。

## Milestone








最后，这里给大家介绍一下 Dragonfly 社区发展过程中几个比较重要的里程碑：

1. Dragonfly 在2 015 年诞生于阿里巴巴，其初衷是作为一个具备 P2P 以及 CDN 能力的分发工具。
2. 随着容器技术的普及，Dragonfly 加入了镜像分发的能力。2017 年，Dragonfly 正式在社区开源
3. 开源之后，随着云原生大环境的发展，Dragonfly 在功能上不断迭代，并在阿里巴巴大规模场景下进行验证。Dragonfly 得到社区的认可，在 2018 年，Dragonfly 项目被贡献给 CNCF 基金会，并成为 CNCF Sandbox 阶段的项目。
4. 在加入 CNCF SandBox 后，Dragonfly 往云原生的道路上进一步靠拢。如在社区生态集成度，与 Harbor，Helm 等的深度集成；在代码层面，通过 Golang 进行重构; 在能力方面，支持 HTTS registry；在社区，支持更多的厂商在生产环境中落地 Dragonfly 等。
5. 未来，Dragonfly 将朝着作为镜像分发基础设施的方向不断演进。目前，我们也在不断的朝着这个方向进行探索，包括在安全性，可扩展性，性能，场景化等。同时，我们更希望协同 OCI 社区，打造并制定镜像/文件在云原生领域的分发标准。目前，基于社区热论的 OCI V2 方面，我们也积极的参与其中。

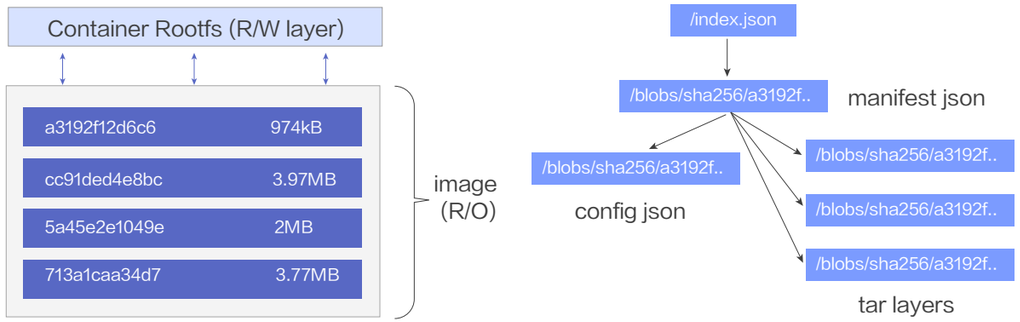
那么接下来马介悦会给我们分享一下在 OCI V2 方面做出的一些探索以及思考，欢迎马介悦！

# Part Three – OCI V2

OCI 组织一共推动了三类标准：运行时标准（runtime spec），分发标准（distribution spec），以及镜像标准（image spec）。我们今天主要讨论镜像标准，当前的镜像标准 OCIv1 版本目前有很多问题，因此 OCI 社区才会开始推进对当前的镜像标准做一些改进，形成新的 OCIv2 标准。

## Container Image Background

Container image is a lightweight, standalone, executable package of software including everything needed to run an application, e.g. code, system tools and libraries




我们先来介绍一下镜像的背景知识，在当前的 OCIv1 的标准下，镜像是由多个 blob 文件(一般是 tar+gzip，也可以是 tar 或者 tar+zstd 等不同的压缩方式)组成的一组数据的bundle集合，每个 blob 文件在 OCI 的定义中叫做镜像层(image layer)，从下往上，每一层都可以看做是针对下一层的一个数据 diff 集合。除了 tarball 数据 blob 之外，还有配置文件 config.json 以及一个包含镜像整个元数据的 manifest.json，所有这些文件最终组合成为一个 OCIv1 标准的镜像。





## Why need OCIv2 format spec

**The OCI organization is now discussing the new OCIv2 format, which focuses on these issues in OCIv1 format**

- **Lazyload**

While only a small fraction of image might be used by application, we waste a lot of disk space, image pulling time, network bandwidth and cpu cycles for the entire image downloading & decompressing procedure

- **Deduplication**

Metadata or small part of modified data will lead to a total file copy in the upper layer, while deleted files still downloaded in the lower layers.

- **Security**

After decompressed, image files are not verifiable, and the targz lacks of signature verification, vulnerable to man-in-the-middle attacks. How to verify data in the lazyload mode is still a problem.

目前业界对于镜像中基于 tarball 的 blob 层的设计有很多异议，主要在三个方面。首先是按需加载方面，目前的镜像加载方式需要把所有镜像数据都拉取到本地，但实际生产上统计只有不到 10% 的镜像数据会被用到，全量下载造成了大量的 cpu，网络带宽以及本地存储的浪费。同时全量下载也大大拖慢了容器的启动时间，大部分启动时间被耗费在了永远不会用到的数据下载上。因此我们第一个目标是能否做到镜像数据的按需加载。

第二个方面在于去重，目前的 OCIv1 的格式中，blob 层之间只保留了 diff 的部分，可以提供一定的去重能力，但是也存在很多漏洞，首先如果一个在 lower layer 中的文件在 upper layer 中被删除，但是镜像下载时，还是会把这个被删除的文件下载下来，其次如果一个文件只是元数据产生了变化，那么在 upper layer 中还是会全量被复制一份，这样同一个文件在镜像下载过程中会被下载多次，最后按照层粒度进行去重也无法解决同一层内部文件的去重需求。我们的目标是能够做到整个镜像里面文件粒度，甚至于文件 chunk 粒度的去重能力。

最后一个方面是安全，目前 OCIv1 的标准中，对每一层的 tarball 做了 digest(一般使用 SHA256 算法)，但是这个 digest 只是保证了 tarball 本身的防篡改，但无法防护被解压到本地文件系统上的文件数据的非法篡改问题，同时当前的镜像标准里没有签名扩展，无法防范网络中间人攻击等行为

## Why need OCIv2 format spec

- P2P based image distribution: Dragonfly, Kraken etc.
- Leveraging and rely on storage level snapshot capability: e.g. Slacker, Ceph rbd solutions
- Layered userspace filesystem: CRFS, but it needs a new stargz file format, which is not a standard
- Leveragin file level blob and local file cache: NTT-filegrain, CERN-vmfs, umoci, but these images package too many blobs, and umoci also rely on local filesystem reflink capability

**All these solutions lacks of data chunk level on-demand loading(except CRFS), data chunk in-flight verification, or real digital signature preventing from man-in-the-middle attacks.**

我们来看一下社区目前已有的工作：dragonfly 和 kraken 通过 p2p 的方式加快了镜像全量下载的速度，并解决了网络带宽的瓶颈。slacker，ceph rbd 则利用了后端存储的 snapshot 能力，e.g. NFSv4.2 snapshot，rbd snapshot。CRFS 提出了新的 stargz 文件格式，从而支持按需加载并向后兼容。filegrain, umoci 则是把 layer的粒度细化到了文件的粒度，但这也造成了 layer blob 文件爆炸，给 registry 和镜像元数据管理造成巨大的开销。

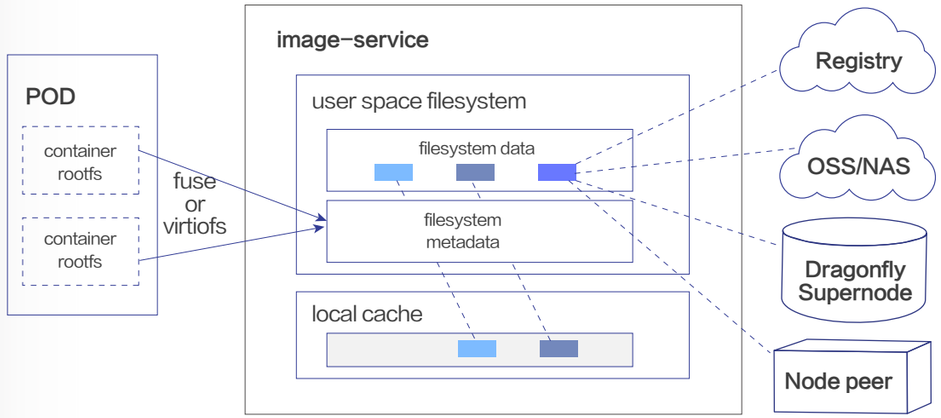




# Part Four – Nydus

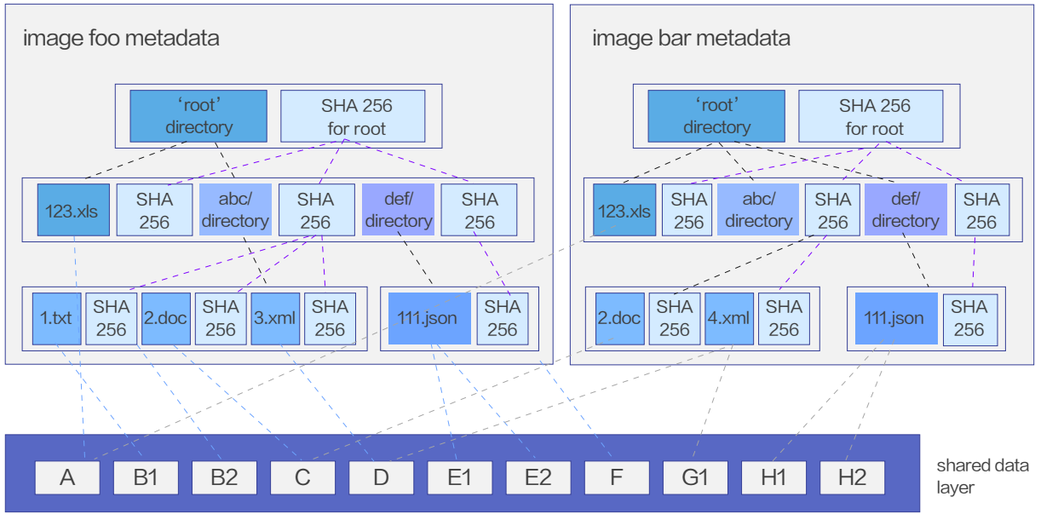
## Nydus: The Dragonfly Image Service

We introduces Nydus, a new container image storage and distribution solution, which targets on current image problems and highly meets the ongoing OCIv2 image format requirement




针对上面社区已有工作的不足，我们提出了一个全新的基于用户态文件系统的实现方案：nydus。nydus 以一个用户态文件系统进程的方式运行，对外暴露 fuse 或者 virtiofs 的文件系统接口。文件系统元数据在 nydus 启动时就被加载，而实际的文件数据则是以 load-on-demand 的方式按需加载，从而可以快速准备好镜像目录，大大缩短容器的冷启动时间。

## Nydus: The Dragonfly Image Service




nydus 针对的是镜像只读文件系统场景，在镜像制作的过程中，我们会把文件元数据的树形结构制作成一个独立元数据 blob 文件，而文件树下每个普通文件则按照固定大小切分为不同的 chunk 分块，任何在分块范围内的io请求会触发整个分块被完整下载。我们还针对每个分块，文件，以及整个目录树都计算了 SHA256 的 digest 值，并形成一个 merkle 树的结构体，这对于 nydus 文件校验，签名认证等功能都是必要的元数据信息。





## Why need OCIv2 format spec

- Open source as a reference implementation for incoming OCIv2 image specification
- Contribute to a new dragonfly p2p distribution service mechanism for image fast-speed lazyloading
- Support more compression/decompression algo, flexible data chunk size and data dedup levels
- Be compatible with different CRI runtimes

我们未来将会把 nydus 作为 OCIv2 的标准实现向社区推广，同时捐献给 dragonfly 社区作为 dragonlfy 的镜像服务子项目开源。我们还会去适配不同的CRI runtime，并支持更多的压缩算法，动态的分块大小和更精细的去重。谢谢大家。


Alibaba Group
阿里巴巴集团