Alibaba Group 阿里巴巴集团 | 阿里云

不一样的双11技术

阿里巴巴经济体云原生实践

2019

微信扫一扫关注

阿里巴巴云原生公众号

微信扫一扫关注

阿里技术公众号

阿里云开发者社区

开发者一站式平台

钉钉扫描二维码，

立即加入 K8s 社区大群

目录

寄语

开篇词

开篇词：上云Cloud Hosting 到 Cloud Native 1

第一章——超大规模 Kubernetes 实践

1、释放云原生价值才是拥抱 Kubernetes 的正确姿势 4

2、阿里云上万个 Kubernetes 集群大规模管理实践 10

3、深入 Kubernetes 的“无人区”：蚂蚁金服双11 的调度系统 21

4、云原生应用万节点分钟级分发协同实践 31

第二章——阿里巴巴云原生化的最佳组合：Kubernetes+容器+神龙

1、阿里巴巴大规模神龙裸金属 K8s 集群运维实践 40

2、PouchContainer 容器技术演进助力阿里云原生升级 46

3、更强，更稳，更高效- 2019 年 双 11 etcd 技术升级分享 53

第三章——Service Mesh 将带来的变革和发展机遇

1、Service Mesh 将带来的变革和发展机遇 61

2、阿里集团 双11 核心应用落地 Service Mesh 所克服的挑战 67

3、蚂蚁金服 双11 Service Mesh超大规模落地揭秘 75

4、服务网格在“路口”的产品思考与实践 86

第四章——Serverless

1、CSE：Serverless 在阿里巴巴 双11 场景的落地 105

2、解密 双11 小程序云背后毫秒级伸缩的 Serverless 计算平台：函数计算 118

第五章——云原生安全及可观测性

1、双11 背后的全链路可观测性：阿里巴巴鹰眼在“云原生时代”的全面升级 123

2、Kubernetes 时代的安全软件供应链 130

3、安全容器：开启云原生沙箱技术的未来 136

4、云原生全链路加密 140

5、Kubernetes 下零信任安全架构分析 148

6、云原生时代的 Identity 157

7、云原生开发者工具——Cloud Toolkit 167

寄语

小邪

阿里巴巴合伙人

阿里云基础产品事业部负责人

许多经过精心设计的架构，平时运行稳定，但在 双11 的演练中都暴露出了大量的细节问题，双11 这个独一无二的场景是阿里技术的试金石。在这个世界级复杂规模单一业务场景下，双11 不断挑战着世界纪录。但所有问题的暴露都是在帮助技术和产品不断完善，这样产品才能更加成熟和稳定，服务更多的客户。今年 双11，我们把“最要命”的系统全都放到了云上，实现了核心系统 100%上云，撑住了 双11 的世界级流量洪峰。这离不开云原生技术的创新，希望本书能对开发者和企业客户产生借鉴意义，共享云计算技术红利。

叔同

阿里云云原生应用平台负责人

在 All in Cloud 的时代企业 IT 架构正在被重塑，而云原生已经成为释放云计算价值的最短路径。2019 年阿里 双 11 核心系统 100%以云原生的方式上云，基于神龙服务器、轻量级容器和兼容 K8s 的调度、集群管理技术，通过云原生应用管理标准化模型 OAM，构建高效和自动化的应用交付体系，大大加快了上云速度；通过升级微服务和服务网格，将服务治理与应用解耦并下沉到基础设施层，提升治理能力和迭代速度，整体向 Serverless 极致弹性、按用计费、无需运维的架构演进，全面实现核心系统上云，让双 11 更稳定让业务创新更敏捷。希望通过将阿里的上云和云原生技术落地最佳实践通过电子书的形式输出，对开发者和企业客户产生借鉴意义，加速技术演进共享云计算技术红利。

START

开篇词

毕玄，阿里巴巴经济体上云总架构师

与客户在同一架“飞机”上

王坚院士曾在很多场合都和阿里的技术人员讲到：阿里云作为一家输出技术的公司，我们需要做到和我们的客户在同一架“飞机”上，而不仅仅是“造飞机”或看着“飞机”在空中飞，阿里经济体云化最重要的就是要做到让我们和客户在同一架“飞机”上。

早在几年前，阿里巴巴经济体就开始借助阿里云的机器资源来支撑 双11 零点的高峰，云的弹性资源优势使得 双11 的机器资源投入成本下降超过 50% 以上，但在这些机器资源上部署的却是我们自己的技术体系，例如容器、中间件、数据库、缓存等，也就意味着我们和客户其实是在不同类型的“飞机”上，而且阿里巴巴经济体在的“飞机”是专为阿里巴巴定制打造的，外部客户是买不到的，这是一个典型的从 Hosting 演进到 Cloud Hosting 的阶段。为了切实做到和客户在同一架“飞机”上，在今年 3 月份，阿里云智能事业群 CTO 张建锋（花名：行癫）正式对外宣布未来一到两年，阿里巴巴百分之百的业务要跑在公共云上，成为“云上的阿里巴巴”。

从 Cloud Hosting 到 Cloud Native

阿里巴巴经济体云化是阿里技术发展史上继之前的分布式架构、异地多活后的又一轮巨大的架构升级，这次架构升级需要把我们从 Cloud Hosting 演进到 Cloud Native，Cloud Native 作为技术圈最火热的名词，不同的人的眼中有不同的定义，我们认为 CloudNative 带来的是一次系统构建方式的巨大变革，Cloud Native 是指业务系统的构建从基于自有、封闭的技术体系，走向基于开放、公共的 Cloud 的技术体系。

在 Cloud Native 时代之前，多数公司随着业务的发展，或多或少都会打造出自有、封闭的技术体系，这一方面造成了巨大的投入，使得公司的技术人才力量没有完全专注的投入在业务上，另一方面也造成了这个行业人才流动的困难，因为知识体系的不同，每到一家新的公司几乎都是全新的一套，这个一定程度上影响了业务创新的速度，尽管很多的开源产品在一定程度上有助于解决这个问题，但还不足以体系化，而在 Cloud Native 时代，我们认为会有两个典型的特征：

1. 对于业务系统端而言，在做系统设计的技术选型上，Cloud 提供了远比自有技术体系更为丰富了多的选择，这使得架构师可以更好的根据业务的状况、阶段等来进行更合理、合适的技术选型，最后表现出来的特征会是业务系统基于 Cloud 的技术体系来搭建，而越来越少的自建或自研，就像 Cloud Hosting 带来的越来越少的公司自己 Hosting 机器的变化一样；
2. 对于云厂商而言，会提供越来越多开放、主流的技术栈的技术产品，从而让客户有更为丰富和自主的选择权，同时云厂商会去做到让这些技术产品的互通性更好，这样客户才能真正做到对于不同类型的业务选择不同的技术产品和体系。

按照这样的思路，阿里巴巴经济体云化在走向 Cloud Native 的道路上，我们的原则是：

1. 业务系统不再采用自有、封闭的技术产品，而是阿里云上对外提供哪些技术产品，我们就基于这些来重构、新建我们的业务系统；
2. 阿里云上提供相应技术领域的主流技术产品，同时根据阿里业务的需求去新增、完善、改造相应的技术产品，并增强不同技术产品的互通性、开放性。

按照这样的原则，随着阿里经济体云化项目的进展，阿里的业务系统就必将完成从基于自有、封闭的自有体系构建，进化到和阿里云的客户一样，基于阿里云上公共的技术产品的体系来构建，从而实现和客户在同一架“飞机”上。

不一样的双11，云原生技术亮点

在这个双11，我们在以下几个方面有了一些不错的进展：

超大规模 Kubernetes 实践

2017 年下半年，阿里集团开始尝试使用 Kubernetes API 来改造内部自研平台，并开始了对应用交付链路的改造，以适配 Kubernetes。2018 年下半年，阿里集团和蚂蚁金服共同投入 Kubernetes 技术生态的研发，力求通过 Kubernetes 替换内部自研平台，实现了小规模的验证，支撑了当年部分双11 的流量。2019 年初，阿里经济体开始进行全面上云改造，阿里集团通过重新设计 Kubernetes 落地方案，适配云化环境，改造落后运维习惯，在 618 前完成了云化机房的小规模验证。2019 年 618 之后，阿里集团内部开始全面推动 Kubernetes 落地，在大促之前完成了全部核心应用运行在 Kubernetes 的目标，并完美支撑了 双11 大考。

阿里巴巴超大规模Kubernetes落地，经受了双11大促真实场景的考验，单集群能支撑万级别 Node、十万级别 POD 的规模。我们推进了三个方面改造：面向终态的改造；自愈能力改造；不可变基础设施改造。相比原有传统的运维链路，扩容效率提升了 50%，集群镜像一致性达到了 99.99% 以上。

阿里巴巴云原生化的最佳组合：Kubernetes+容器+神龙

今年 双11，我们通过 K8s+容器+神龙的最佳组合实现了阿里核心系统 100%以云原生的方式上云，完美支撑了 54.4w 峰值流量以及 2684 亿的成交量。基于 0 虚拟化开销的神龙裸金属，通过使用行业标准的容器与调度、编排、管理技术，推动经济体云原生技术全面升级。容器性能提升 10%、神龙节点可调度率达到 99% 以上、容器稳定性与在线率全面提升。

Service Mesh 超大规模落地

阿里巴巴在 双11 的部分电商核心应用上落地了完整的 Service Mesh 解决方案，借助 双11 的严苛业务场景完成了规模化落地前的初步技术验证；蚂蚁金服也实现了 Service Mesh 的大规模落地。Service Mesh 所带来的变化体现于：服务治理手段从过去的框架思维向平台思维转变；技术平台的建设从面向单一编程语言向面向多编程语言转变。

Service Mesh 创造了一次以开发者为中心去打造面向未来的分布式应用开发平台的机会，给其他技术产品创造了重新思考在云原生时代发展的机会，给技术基础设施如何与业务基础技术更好地协同提供了一次探索机会，并为探索面向未来的异地多活、应用永远在线的整体技术解决方案打开了一扇大门。

期待《不一样的 双11：阿里巴巴经济体云原生实践》会给你带来新的灵感。

01

释放云原生价值才是拥抱 Kubernetes 的正确姿势

张振，花名守辰，阿里云云原生应用平台高级技术专家

Kubernetes 作为云原生的最佳实践，已经成为了事实上的容器编排引擎标准，Kubernetes 在阿里巴巴集团落地主要经历了四个阶段：

研发和探索：2017年下半年阿里集团开始尝试使用Kubernetes api来改造内部自研平台，并开始了对应用交付链路的改造，以适配Kubernetes。
初步灰度：2018年下半年阿里集团和蚂蚁金服共同投入Kubernetes技术生态的研发，力求通过 Kubernetes 替换内部自研平台，实现了小规模的验证，支撑了当年部分双11的流量。
云化灰度：2019年初阿里经济体开始进行全面上云改造，阿里集团通过重新设计 Kubernetes 落地方案，适配云化环境，改造落后运维习惯，在618前完成了云化机房的小规模验证。
规模化落地：2019年618之后，阿里集团内部开始全面推动Kubernetes落地，在大促之前完成了全部核心应用运行在 Kubernetes 的目标，并完美支撑了双11大考。

在这几年的实践中，一个问题始终萦绕在各个架构师的头脑中， 在阿里巴巴这么大体量、这么复杂的业务下， 遗留了大量传统的运维习惯以及支撑这些习惯的运维体系，落地 Kubernetes 到底要坚持什么，要妥协什么，要改变什么？

今天，将分享阿里巴巴这几年对这些问题的思考。答案很明显，就是拥抱 Kubernetes 本身并不是目的，而是通过拥抱 Kubernetes 撬动业务的云原生改造，通过 Kubernetes 的能力，治理传统运维体系下的沉疴顽疾，释放云弹性的能力，为业务的应用交付解绑提速。

在阿里巴巴的 Kubernetes 落地实践中，关注了下面几个关键的云原生改造：

面向终态改造

在阿里巴巴传统的运维体系下，应用的变更都是 PaaS 通过创建操作工单，发起工作流，继而对容器平台发起一个个的变更来完成的。

当应用发布时， PaaS 会从数据库中查到应用所有相关的容器， 并针对每个容器，向容器平台发起修改容器镜像的变更。而每一个变更实际上也是一个工作流，涉及到镜像的拉取，旧容器的停止和新容器的创建。而工作流中一旦发生错误或者超时，都需要 PaaS 进行重试。一般而言，为了保证工单及时的完成，重试仅会执行几次，几次重试失败后，只能依靠人工处理。

当应用缩容时，PaaS 会根据运维人员的输入，指定容器列表进行删除，而一旦其中有容器因为宿主机异常的情况下删除失败或者超时，PaaS 只能反复重试，为了保证工单的结束，在重试一定次数后只能认为容器删除成功。如果宿主机后续恢复正常，被"删除"的容器很有可能依然运行着。

传统的面向过程的容器变更一直存在如下问题无法解决：
1. 单个变更失败无法保证最终成功。例如，一旦容器镜像变更失败，PaaS 无法保证容器镜像的最终一致；一旦删除容器失败， 也无法保证容器最后真的被删除干净。两个例子都需要通过巡检来处理不一致的容器。而巡检任务因为执行较少，其正确性和及时性都很难保证。
2. 多个变更会发生冲突，例如应用的发布和应用的扩容过程需要加锁，否则会出现新扩的容器镜像未更新的情况。而一旦对变更进行加锁，变更的效率又会大幅下降。

自愈能力改造

在阿里巴巴传统的运维体系下，容器平台仅生产资源，应用的启动以及服务发现是在容器启动后由 PaaS 系统来执行的，这种分层的方法给了 PaaS 系统最大的自由，也在容器化后促进了阿里巴巴第一波容器生态的繁荣。但是这种方式有一个严重的问题，即： 容器平台无法独立地去触发容器的扩缩容，而需要和一个个 PaaS 来做复杂的联动，上层 PaaS 也需要做很多重复的工作。这妨碍了容器平台在宿主机发生故障、重启、容器中进程发生异常、卡住时的高效自愈修复，也让弹性扩缩容变得非常复杂。

在 Kubernetes 中通过容器的命令以及生命周期钩子，可以将 PaaS 启动应用以及检查应用启动状态的流程内置在了 pod 中；另外，通过创建 service 对象，可以将容器和对应的服务发现机制关联起来，从而实现容器、应用、服务生命周期的统一。容器平台不再仅仅生产资源，而是交付可以直接为业务使用的服务。这极大地简化了上云之后故障自愈以及自动弹性扩缩容能力的建设， 真正地发挥了云的弹性能力。

另外， 在宿主机发生故障的情况下， PaaS 传统上需要先对应用进行扩容，然后才删除宿主机上的容器。然而在大规模的集群下，我们发现往往会卡在应用扩容这步。应用资源额度可能不够，集群内满足应用调度限制的空闲资源也可能不够，无法扩容就无法对宿主机上的容器进行驱逐，进而也无法对异常的宿主机进行送修，久而久之，整个集群很容易就陷入故障机器一大堆，想修修不了，想腾腾不动的困境之中。

在 Kubernetes 中对于故障机的处理要"简单和粗暴"得多，不再要求对应用先扩容，而是直接把故障机上的容器进行删除，删除后才由负载控制器进行扩容。这种方案乍一听简直胆大妄为，落地 Kubernetes 的时候很多 PaaS 的同学都非常排斥这种方法，认为这会严重影响业务的稳定性。事实上呢，绝大多数核心的业务应用都维护着一定的冗余容量，以便全局的流量切换或者应对突发的业务流量，临时删除一定量的容器根本不会造成业务的容量不足。

我们所面临的关键问题是如何确定业务的可用容量，当然这是个更难的问题，但对于自愈的场景完全不需要准确的容量评估，只需要一个可以推动自愈运转的悲观估计就可以。在 Kubernetes 中可以通过 PodDisruptionBudget 来定量地描述对应用的可迁移量，例如可以设置对应用进行驱逐的并发数量或者比例。这个值可以参考发布时的每批数量占比来设置。假如应用发布一般分 10 批，那么可以设置 PodDisruptionBudget 中的 maxUnavailable 为 10%（对于比例，如果应用只有 10 个以内的实例，Kubernetes 还是认为可以驱逐 1 个实例）。万一应用真的一个实例都不允许驱逐呢，那么对不起，这样的应用是需要改造之后才能享受上云的收益的。一般应用可以通过改造自身架构，或者通过 operator 来自动化应用的运维操作，从而允许实例的迁移。

不可变基础设施改造

Docker 的出现提供了一种统一的应用交付的形式，通过把应用的二进制、配置、依赖统一在构建过程中打到了镜像中， 通过使用新的镜像创建容器，并删除掉旧容器就完成了应用的变更。Docker 在交付应用时和传统基于软件包或者脚本的交付方式有一个重大区别，就是强制了容器的不可变，想要变更容器只能通过新创建容器来完成，而每个新容器都是从应用同一个镜像创建而来，确保了一致性，从而避免了配置漂移，或者雪花服务器的问题。

Kubernetes 进一步强化了不可变基础设施的理念，在默认的滚动升级过程中不但不会变更容器，而且还不会变更pod。每次发布，都是通过创建新的pod，并删除旧的 pod 来完成，这不仅保证了应用的镜像统一，还保证了数据卷、资源规格以及系统参数配置都是和应用模板的 spec 保持一致。另外， 不少应用都有比较复杂的结构，一个应用实例可能同时包含多个团队独立开发的组件。 比如一个应用可能包括了业务相关的应用程序服务器，也包括了基础设施团队开发的日志采集的进程，甚至还包括了第三方的中间件组件。这些进程、组件如果想要独立发布就不能放在一个应用镜像中，为此 Kubernetes 提供了多容器 pod 的能力，可以在一个 pod 中编排多个容器，想要发布单个组件，只需要修改对应容器的镜像即可。

不过，阿里巴巴传统的容器形态是富容器，即应用程序服务器，以及日志采集进程这些相关的组件都部署在一个大的系统容器中，这造成了日志采集等组件的资源消耗无法单独限制，也无法方便地进行独立的升级。因此，在阿里巴巴这次上云中， 开始把系统容器中除业务应用外的其他组件都拆分到独立的 sidecar 容器，我们称之为轻量化容器改造。改造后，一个 pod 内会包括一个运行业务的主容器的，一个运行着各种基础设施 agent 的运维容器，以及服务网格等的 sidecar。轻量化容器之后， 业务的主容器就能以比较低的开销运行业务服务，从而为更方便进行 serverless 的相关改造。

不过，Kubernetes默认的滚动升级过程过于僵硬地执行了不可变基础设施的理念，导致对多容器 pod 的能力支持有严重的缺失。虽然可以在一个 pod 中编排多个容器，但如果要发布 pod 中的一个容器，在实际执行发布时，不仅会重建待发布的容器，还会把整个 pod 都删除，而后重调度、再重建。这意味着假如要升级基础设施的日志采集组件，会导致其他组件，特别是业务的应用服务器被一起删除重启，从而干扰到正常的业务运行。因此，多个组件的变更依然没有解耦。

对业务而言，假如 pod 中有本地缓存的组件，而每次业务的发布都会重启缓存进程，这会导致业务发布期间缓存的命中率大幅下降，影响性能甚至用户的体验；另外，如果基础设施、中间件等团队的组件升级都和业务的组件升级绑定在一起，这会给技术的迭代更新带来巨大的阻碍。假设负责中间件的团队推出了新的 service mesh 版本， 而为了升级 mesh 还需要央求一个个业务发布才能更新 mesh 组件，那么中间件的技术升级就会大大减速。

因此，相比 pod 层次的不可变，我们认为坚持容器级别的不可变原则，更能发挥 Kubernetes 多容器 pod 的技术优势。为此，我们建设了支持应用发布时，只原地修改 pod 中部分容器的能力，特别的，建设了支持容器原地升级的工作负载控制器，并替换 Kubernetes 默认的 deployment 和 stateful-set 控制器作为内部的主要工作负载。另外，还建设了支持跨应用进行 sidecar 容器升级的 side-carset， 方便进行基础设施以及中间件组件的升级。此外，通过支持原地升级还带来了集群分布确定性、加速镜像下载等的额外优势。这部分能力，我们已经通过 OpenKruise 项目开源出来。OpenK-ruise 中的 Kruise 是 cruise的谐音，'K' for Kubernetes, 寓意 Kubernetes 上应用的自动巡航，满载着满载阿里巴巴多年应用部署管理经验和阿里经济体云原生化历程的最佳实践. 目前，OpenK-ruise 正在计划发布更多的 Controller 来覆盖更多的场景和功能比如丰富的发布策略，金丝雀发布，蓝绿发布，分批发布等等。

总结

今年我们实现了Kubernetes 的规模化落地，经受了双11大促真实场景的考验。像阿里这样有着大量存量应用的场景，落地k8s并没有捷径，我们经受住了快速规模化落地的诱惑， 没有选择兼容和妥协落后的运维习惯，而是选择深蹲打好基础，选择深挖云原生价值。接下来，我们将继续推动更多应用的云原生改造，特别是有状态应用的改造，让有状态应用的部署和运维更加高效和；另外，还将推动整个应用交付链路的云原生改造，让应用交付更加高效和标准化。

02

阿里云上万个 Kubernetes 集群大规模管理实践

汤志敏，阿里云容器服务高级技术专家

内容简介

阿里云容器服务从 2015 年上线后，一路伴随并支撑 双11 发展。在 2019 年的双11 中，容器服务 ACK（ACK是Alibaba Cloud Container Service for Kubernetes的缩写） 除了支撑阿里巴巴内部核心系统容器化上云和阿里云的云产品本身，也将阿里巴巴多年的大规模容器技术以产品化的能力输出给众多围绕 双11 的生态公司和 ISV 公司。通过支撑来自全球各行各业的容器云，容器服务已经沉淀了支持单元化架构、全球化架构、柔性架构的云原生应用托管中台能力，管理了超过 1W 个以上的容器集群。本文会介绍容器服务 ACK 在海量 Kubernetes 集群管理上的实践经验。

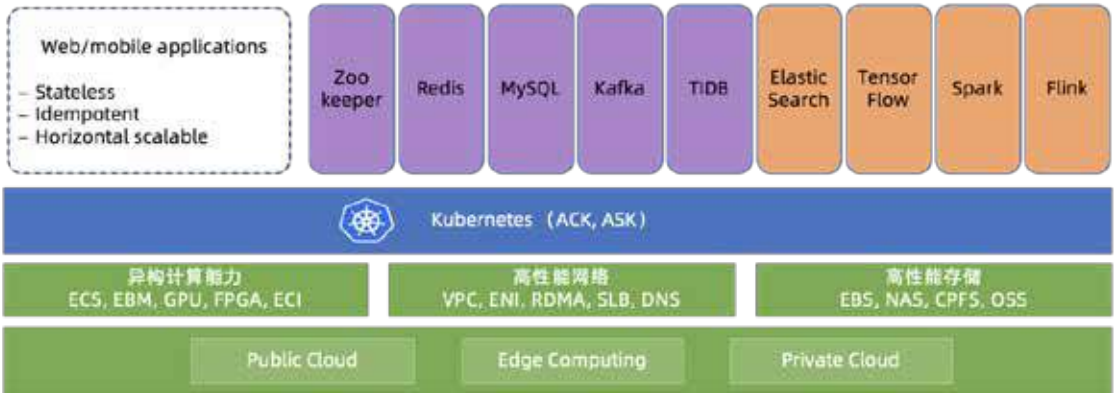
引言

什么是海量 Kubernetes 集群管理？大家可能之前看过一些分享，介绍了阿里巴巴如何管理单集群 1W 节点的最佳实践，管理大规模节点是一个很有意思的挑战。不过这里讲的海量 Kubernetes 集群管理，会侧重讲如何管理超过 1W 个以上不同规格的 Kubernetes 集群。根据我们和一些同行的沟通，往往一个企业内部只要管理几个到几十个 Kubernetes 集群，那么我们为什么需要考虑管理如此庞大数量的 Kubernetes 集群？

- 首先，容器服务 ACK 是阿里云上的云产品，提供了 Kubernetes as a Service 的能力，面向全球客户，目前已经在全球 20 个地域支持；
- 其次，得益于云原生时代的发展，越来越多的企业拥抱 Kubernetes，Kubernetes 已经逐渐成为云原生时代的基础设施，成为 platform of platform。

Kubernetes逐渐成为云原生时代的基础设施

从无状态应用，到企业核心应用，到数据智能应用



背景介绍

海量k8s集群托管的痛点

种类不同：
- 标准K8s集群
- Serverless k8s集群
- Edge k8s集群
- Windows k8s集群

大小不一：
- 单集群节点数从几个到上万
- 单集群POD数从几个到百万
- 单集群Service从几个到几千

安全合规：
- 海外：PCIDSS、欧盟的GDPR
- 中国：金融云、政务云的等保等

持续演进：
- k8s版本的持续迭代和CVE修复
- 系统组件、三方组件的持续更新

首先我们一起来看下托管这些 Kubernetes 集群的痛点：

集群种类不同：有标准的、无服务器的、AI 的、裸金属的、边缘、Windows 等 Kubernetes 集群。不同种类的集群参数、组件和托管要求不一样，并且需要支撑更多面向垂直场景的 Kubernetes；
集群大小不一：每个集群规模大小不一，从几个节点到上万个节点，从几个 service 到几千个 service 等，需要能够支撑每年持续几倍集群数量的增长；
集群安全合规：分布在不同的地域和环境的 Kubernetes 集群，需要遵循不同的合规性要求。比如欧洲的 Kubernetes 集群需要遵循欧盟的 GDPR 法案，在中国的金融业和政务云需要有额外的等级保护等要求；
集群持续演进：需要能够持续的支持 Kubernetes 的新版本新特性演进。

设计目标

1. 支持单元化的分档管理、容量规划和水位管理；
2. 支持全球化的部署、发布、容灾和可观测性；
3. 支持柔性架构的可插拔、可定制、积木式的持续演进能力。

支持单元化的分档管理、容量规划和水位管理

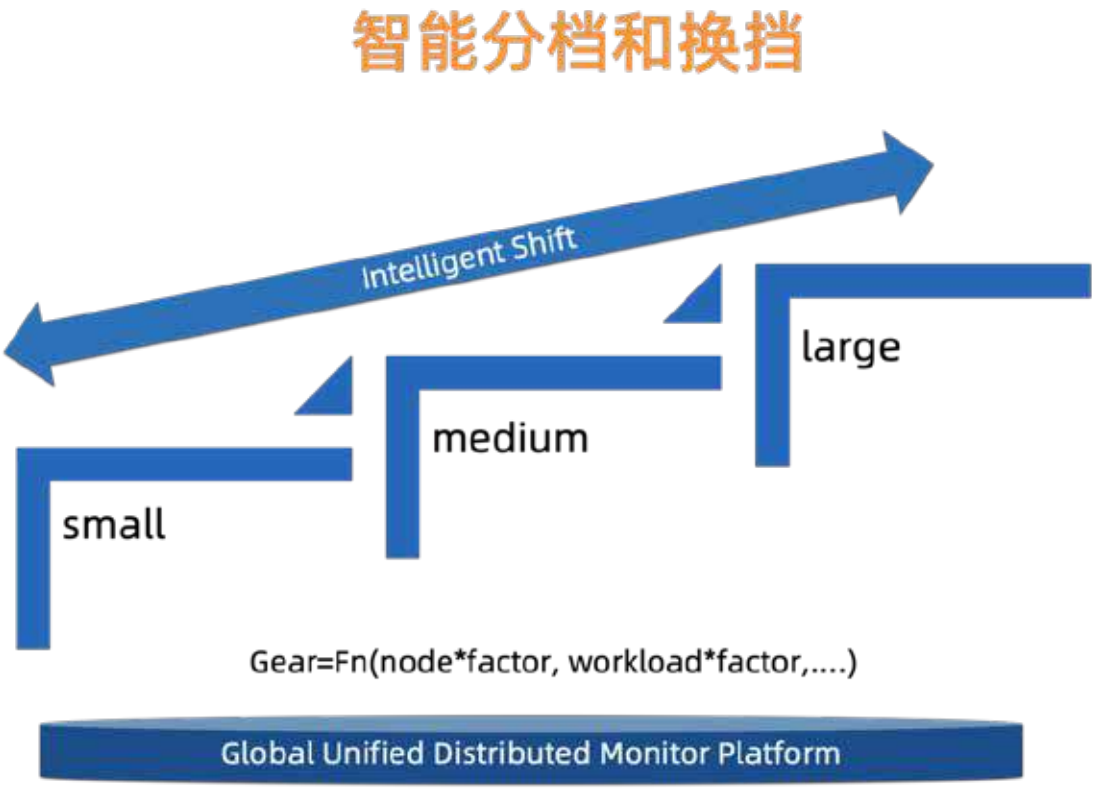
单元化

一般讲到单元化，大家都会联想到单机房容量不够或二地三中心灾备等场景。那单元化和 Kubernetes 管理有什么关系？对我们来说，一个地域（比如：杭州）可能会管理几千个 Kubernetes，需要统一维护这些 Kubernetes 的集群生命周期管理。作为一个 Kubernetes 专业团队，一个朴素的想法就是通过多个 Kubernetes 元集群来管理这些guest K8s master。而一个 Kubernetes 元集群的边界就是一个单元

曾经我们经常听说某某机房光纤被挖断，某某机房电力因故障而导致服务中断，容器服务 ACK 在设计之初就支持了同城多活的架构形态，任何一个用户 Kubernetes 集群的 master 组件都会自动地分散在多个机房，不会因单机房问题而影响集群稳定性；另外一个层面，同时要保证 master 组件间的通信稳定性，容器服务 ACK 在打散 master 时调度策略上也会尽量保证 master 组件间通信延迟在毫秒级。

分档化

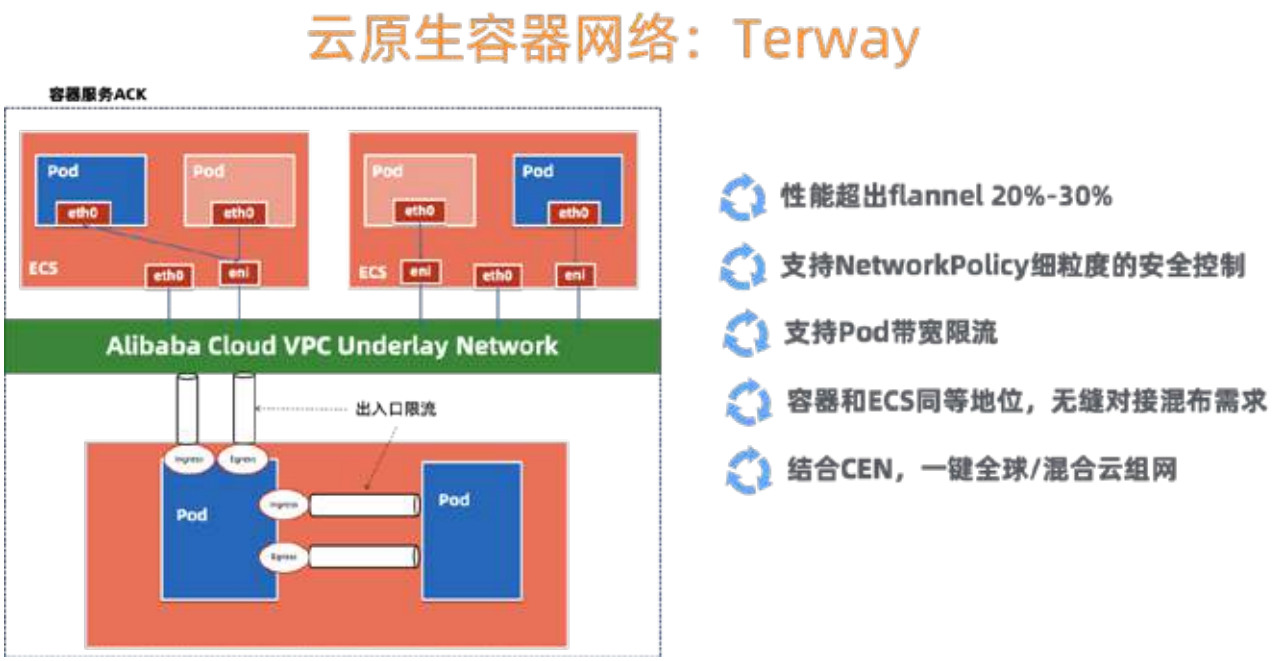
大家都知道，Kubernetes 集群的 master 组件的负载主要与 Kubernetes 集群的节点规模、worker 侧的 controller 或 workload 等需要与 kube-apiserver 交互的组件数量和调用频率息息相关，对于上万个 Kubernetes 集群，每个用户 Kubernetes 集群的规模和业务形态都千差万别，我们无法用一套标准配置来去管理所有的用户 Kubernetes 集群。同时，从成本经济角度考虑，我们提供了一种更加灵活、更加智能的托管能力。考虑到不同资源类型会对 master 产生不同的负载压力，因此我们需要为每类资源设置不同的因子，最终可归纳出一个计算范式，通过此范式可计算出每个用户 Kubernetes 集群 master 所适应的档位。另外，我们也会基于已构建的 Kubernetes 统一监控平台实时指标来不断地优化和调整这些因素值和范式，从而可实现智能平滑换挡的能力。



容量规划

接下来我们看下 Kubernetes 元集群的容量模型，单个元集群到底能托管多少个用户 Kubernetes 集群的 master?

- 首先，要确认容器网络规划。这里我们选择了阿里云自研的高性能容器网络Terway, 一方面需要通过弹性网卡 ENI 打通用户 VPC 和托管 master 的网络，另一方面提供了高性能和丰富的安全策略；
- 接下来，我们需要结合 VPC 内的 ip 资源，做网段的规划，分别提供给 node、pod 和 service。
- 最后，我们会结合统计规律，结合成本、密度、性能、资源配额、档位配比等多种因素的综合考量，设计每个元集群单元中部署的不同档位的 guest Kubernetes 的个数，并预留 40% 的水位。



支持全球化的部署、发布、容灾和可观测性

容器服务已经在全球 20 个地域支持，我们提供了完全自动化的部署、发布、容灾和可观测性能力，接下来将重点介绍全球化跨数据中心的可观测。

全球跨数据中心的可观测性

全球化布局的大型集群的可观测性，对于 Kubernetes 集群的日常保障至关重要。如何在纷繁复杂的网络环境下高效、合理、安全、可扩展的采集各个数据中心中目标集群的实时状态指标，是可观测性设计的关键与核心。我们需要兼顾区域化数据中心、单元化集群范围内可观测性数据的收集，以及全局视图的可观测性和可视化。基于这种设计理念和客观需求，全球化可观测性必须使用多级联合方式，也就是边缘层的可观测性实现下沉到需要观测的集群内部，中间层的可观测性用于在若干区域内实现监控数据的汇聚，中心层可观测性进行汇聚、形成全局化视图以及告警。样设计的好处在于可以灵活的在每一级别层内进行扩展以及调整，适合于不断增长的集群规模，相应的其他级别只需调整参数，层次结构清晰；网络结构简单，可以实现内网数据穿透到公网并汇聚。

针对该全球化布局的大型集群的监控系统设计，对于保障集群的高效运转至关重要，我们的设计理念是在全球范围内将各个数据中心的数据实时收集并聚合，实现全局视图查看和数据可视化，以及故障定位、告警通知。进入云原生时代，Prometheus 作为 CNCF 第二个毕业的项目，天生适用于容器场景，Prometheus 与 Kubernetes 结合一起，实现服务发现和对动态调度服务的监控，在各种监控方案中具有很大的优势，实际上已经成为容器监控方案的标准，所以我们也选择了 Prometheus 作为方案的基础。

针对每个集群，需要采集的主要指标类别包括

- OS 指标，例如节点资源（CPU, 内存，磁盘等）水位以及网络吞吐；
- 元集群以及用户集群 Kubernetes master 指标，例如 kube-apiserver, kube-controller-manager, kube-scheduler 等指标；
- Kubernetes 组件（kubernetes-state-metrics，cadvisor）采集的关于 Kubernetes 集群状态；
- etcd 指标，例如 etcd 写磁盘时间，DB size，Peer 之间吞吐量等等。

当全局数据聚合后，AlertManager 对接中心 Prometheus，驱动各种不同的告警通知行为，例如钉钉、邮件、短信等方式。

监控告警架构

为了合理的将监控压力负担分到多个层次的 Prometheus 并实现全局聚合，我们使用了联邦 Federation 的功能。在联邦集群中，每个数据中心部署单独的 Prometheus，用于采集当前数据中心监控数据，并由一个中心的 Prometheus 负责聚合多个数据中心的监控数据。基于 Federation 的功能，我们设计的全球监控架构图如下，包括监控体系、告警体系和展示体系三部分。

监控体系按照从元集群监控向中心监控汇聚的角度，呈现为树形结构，可以分为三层：

边缘 Prometheus

为了有效监控元集群 Kubernetes 和用户集群 Kubernetes 的指标、避免网络配置的复杂性，将 Prometheus 下沉到每个元集群内

级联 Prometheus

级联 Prometheus 的作用在于汇聚多个区域的监控数据。级联 Prometheus 存在于每个大区域，例如中国区，欧洲美洲区，亚洲区。每个大区域内包含若干个具体的区域，例如北京，上海，东京等。随着每个大区域内集群规模的增长，大区域可以拆分成多个新的大区域，并始终维持每个大区域内有一个级联 Prometheus，通过这种策略可以实现灵活的架构扩展和演进。

中心 Prometheus

中心 Prometheus 用于连接所有的级联 Prometheus，实现最终的数据聚合、全局视图和告警。为提高可靠性，中心 Prometheus 使用双活架构，也就是在不同可用区布置两个 Prometheus 中心节点，都连接相同的下一级 Prometheus。

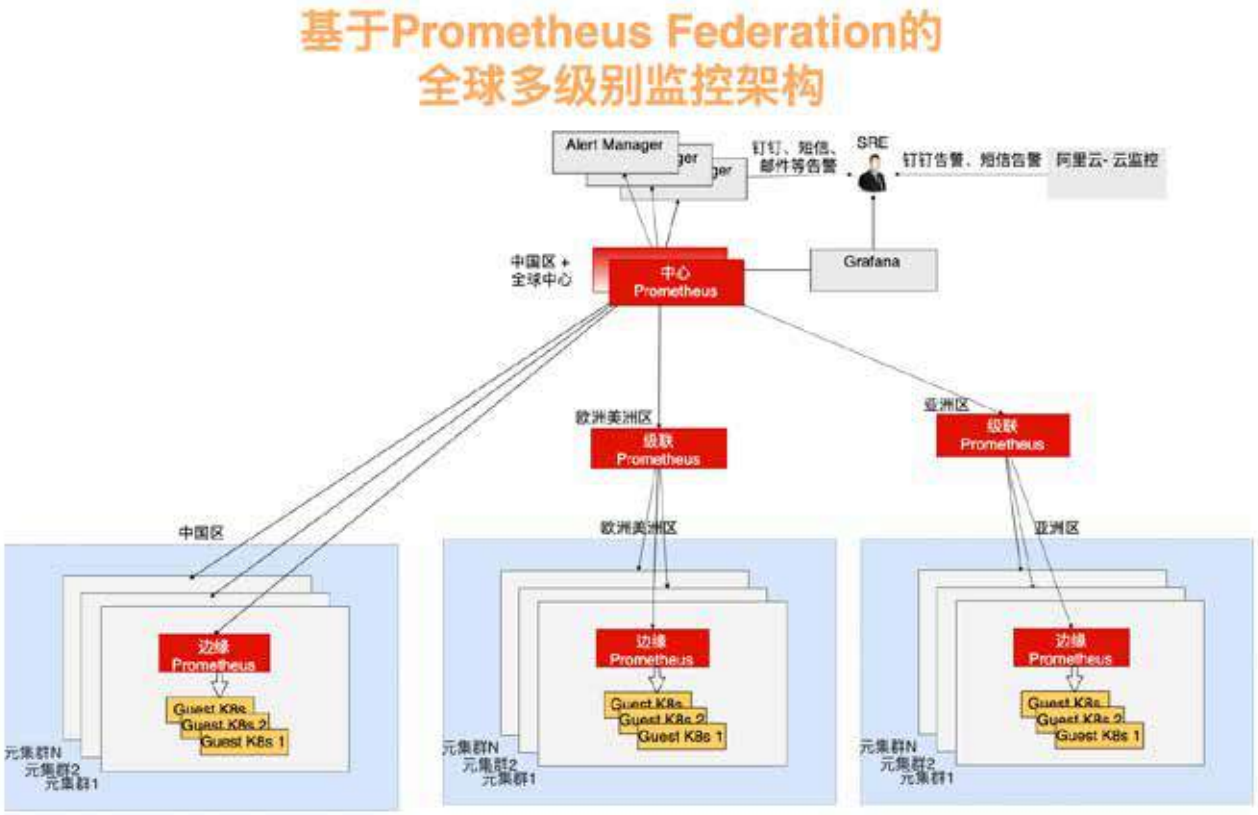


图2-1 基于 Prometheus Federation 的全球多级别监控架构

优化策略

监控数据流量与 API server 流量分离

API server 的代理功能可以使得 Kubernetes 集群外通过 API server 访问集群内的 Pod、Node 或者 Service。

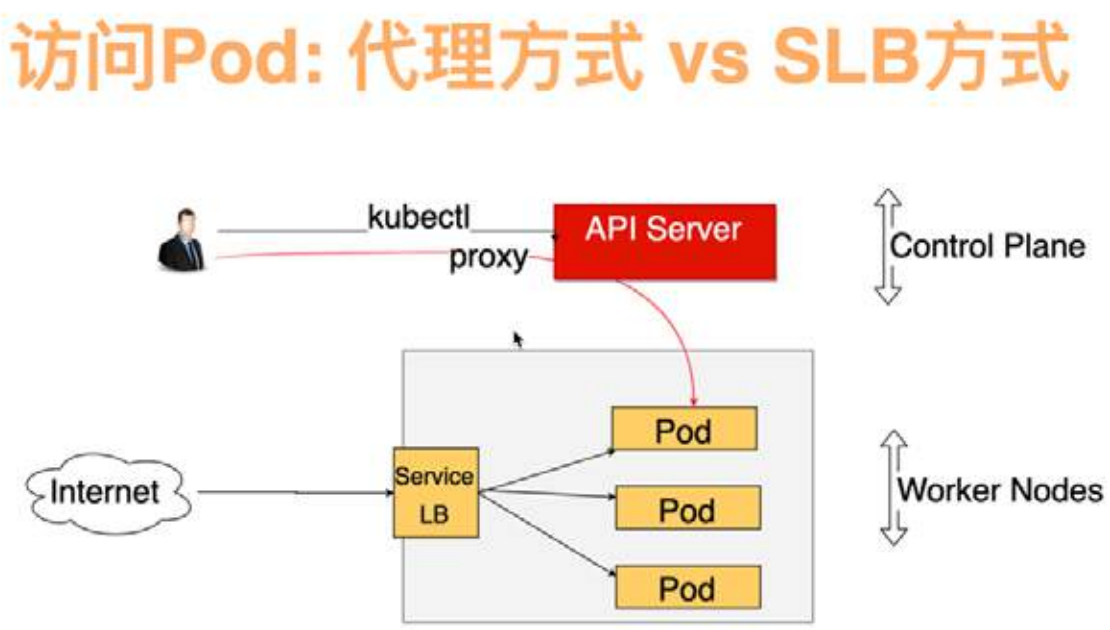


图3-1 通过 API Server 代理模式访问 Kubernetes 集群内的 Pod 资源

常用的透传 Kubernetes 集群内 Prometheus 指标到集群外的方式是通过 API server 代理功能，优点是可以重用 API server 的 6443 端口对外开放数据，管理简便；缺点也明显，增加了 API server 的负载压力。如果使用 API Server 代理模式，考虑到客户集群以及节点都会随着售卖而不断扩大，对 API server 的压力也越来越大并增加了潜在的风险。对此，针对边缘 Prometheus 增加了 LoadBalancer 类型的 service，监控流量完全 LoadBalancer，实现流量分离。即便监控的对象持续增加，也保证了 API server 不会因此增加 Proxy 功能的开销。

收集指定 Metric

在中心 Prometheus 只收集需要使用的指标，一定不能全量抓取，否则会造成网络传输压力过大丢失数据。

Label 管理

Label 用于在级联 Prometheus 上标记 region 和元集群，所以在中心 Prometheus 汇聚是可以定位到元集群的颗粒度。同时，尽量减少不必要的 label，实现数据节省。

支持柔性架构的可插拔、可定制、积木式的持续演进能力

前面两部分简要描述了如何管理海量 Kubernetes 集群的一些思考，然而光做到全球化、单元化的管理还远远不够。Kubernetes 能够成功，包含了声明式的定义、高度活跃的社区、良好的架构抽象等因素，Kubernetes 已经成为云原生时代的 Linux。我们必须要考虑 Kubernetes 版本的持续迭代和 CVE 漏洞的修复，必须要考虑 Kubernetes 相关组件的持续更新，无论是 CSI、CNI、Device Plugin 还是 Scheduler Plugin 等等。为此我们提供了完整的集群和组件的持续升级、灰度、暂停等功能。

组件可插拔

组件检查


容器服务（集群运维） / 升级检查 / 检查报告
← 检查报告
检查时间 2019-11-25 14:09:02
检查状态 已完成
检查结果 正常
集群资源
已完成
集群组件
已完成
集群配置
已完成
集群组件检查结果
全部
待处理
集群组件
Kube Proxy Master
正常 (1)
Kube Proxy Worker
正常 (1)
API Service
正常 (20)
集群节点
节点
正常 (6)

组件升级

2019 年 6 月，阿里巴巴将内部的云原生应用自动化引擎 OpenKruise 开源，这里我们重点介绍下其中的 BroadcastJob 功能，他非常适用于每台 worker 机器上的组件进行升级，或者对每台机器上的节点进行检测。（Broadcast Job 会在集群中每个 node 上面跑一个 pod 直至结束。类似于社区的 DaemonSet，区别在于 DaemonSet 始终保持一个 pod 长服务在每个 node 上跑，而 BroadcastJob 中最终这个 pod 会结束。）

集群模板

此外，考虑不同 Kubernetes 使用场景，我们提供了多种 Kubernetes 的 cluster profile，可以帮助用户提供更方便的集群选择。我们会结合大量集群的实践，持续提供更多更好的集群模板。

总结

随着云计算的发展,以Kubernetes为基础的云原生技术持续推动行业进行数字化转型。容器服务ACK提供了安全稳定、高性能的Kubernetes托管服务已经成为云上运行Kubernetes的最佳载体。在本次双11，容器服务 ACK 在各个场景为双十一作出贡献，支撑了阿里巴巴内部核心系统容器化上云，支撑了阿里云微服务引擎MSE、视频云、CDN等云产品，也支撑了双11 的生态公司和 ISV 公司，包括聚石塔电商云、菜鸟物流云、东南亚的支付系统等等。容器服务ACK会持续前行，持续提供更高更好的云原生容器网络、存储、调度和弹性能力，端到端的全链路安全能力，serverless 和 servicemesh 等能力。对于有兴趣的开发者，可以前往阿里云控制台 https://cn.aliyun.com/product/kubernetes ，创建一个 Kubernetes 集群来体验。对于容器生态的合作伙伴，也欢迎加入阿里云的容器应用市场，和我们一起共创云原生时代。

03

深入 Kubernetes 的“无人区”：蚂蚁金服双11 的调度系统

曹寅，蚂蚁金服 Kubernetes 落地负责人

前言

经过超过半年的研发，蚂蚁金服在今年完成了 Kubernetes 的全面落地，并使得核心链路 100% 运行在 Kubernetes。到今年 双11，蚂蚁金服内部通过 Kubernetes 管理了数以万计的机器以及数十万的业务实例，超过 90% 的业务已经平稳运行在 Kubernetes 上。整个技术切换过程平稳透明，为云原生的资源基础设施演进迈好了关键的一步。

本文主要介绍 Kubernetes 在蚂蚁金服的使用情况，双11 大促对 Kubernetes 带来史无前例的挑战以及我们的最佳实践。希望通过分享这些我们在实践过程中的思考，让大家在应用 Kubernetes 时能够更加轻松自如。

蚂蚁金服的 Kubernetes 现状

发展历程与落地规模

Kubernetes 在蚂蚁金服落地主要经历了四个阶段：

平台研发阶段：2018 年下半年蚂蚁金服和阿里集团共同投入 Kubernetes 技术生态的研发，力求通过 Kubernetes 替换内部自研平台；

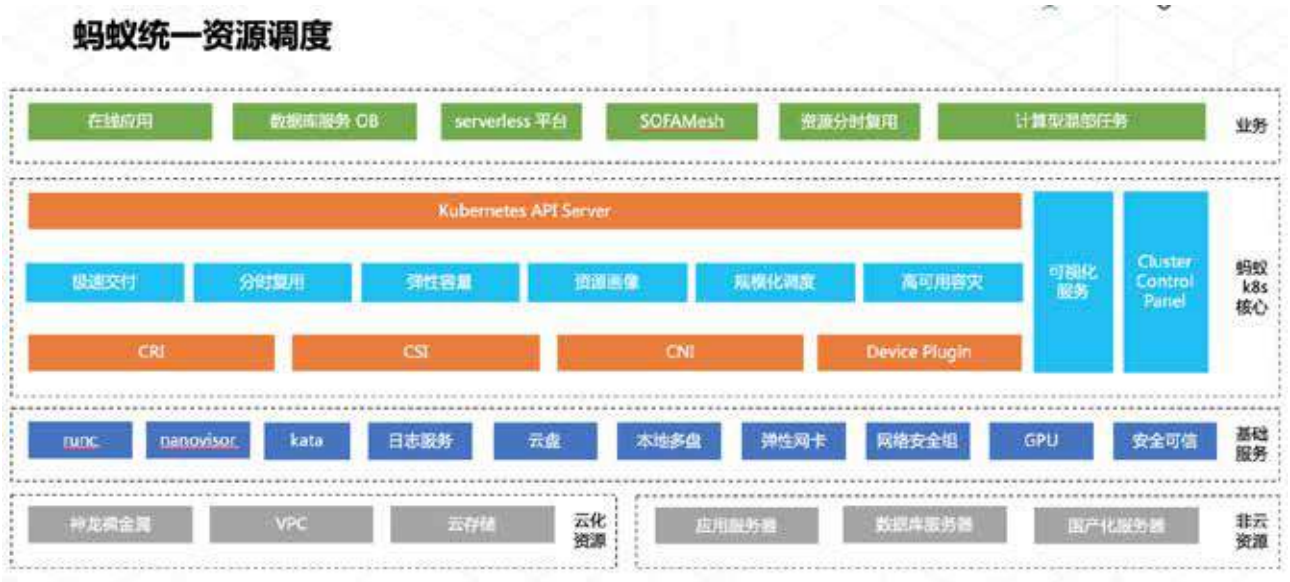
灰度验证：2019 年初 Kubernetes 在蚂蚁金服灰度落地，通过对部分资源集群进行架构升级以及灰度替换生产实例两种方式，让 Kubernetes 得以小规模的验证；

云化落地（蚂蚁金服内部基础设施上云）：2019 年 4 月蚂蚁金服内部完成 Kubernetes 适配云化环境的目标，并于 618 之前完成云化机房 100% 使用 Kubernetes 的目标，这是 Kubernetes 第一次在蚂蚁金服内部得到规模化的验证；

规模化落地：2019 年 618 之后，蚂蚁金服内部开始全面推动 Kubernetes 落地，在大促之前完成核心链路100% 运行在 Kubernetes 的目标，并完美支撑了 双11 大考。

统一资源调度平台

Kubernetes 承载了蚂蚁金服在云原生时代对资源调度的技术目标：统一资源调度。通过统一资源调度，可以有效提升资源利用率，极大的节省资源成本。要做到统一调度，关键在于从资源层面将各个二层平台的调度能力下沉，让资源在 Kubernetes 统一分配。



蚂蚁金服在落地 Kubernetes 实现统一调度目标时遵循了标准化的扩展方式：

- 一切业务扩展均围绕 Kubernetes APIServer，通过 CRD + Operator 方式完成业务功能的适配和扩展；
- 基础服务通过 Node 层定义的资源接口完成个性化适配，有益于形成资源接入的最佳实践。

得益于持续的标准化工作，我们在落地 Kubernetes 的大半年内应用了多项技术，包含安全容器，统一日志，GPU 精细调度，网络安全隔离及安全可信计算等，并通过 Kubernetes 统一使用和管理这些资源服务了大量在线业务以及计算任务型业务。

双11 Kubernetes 实践

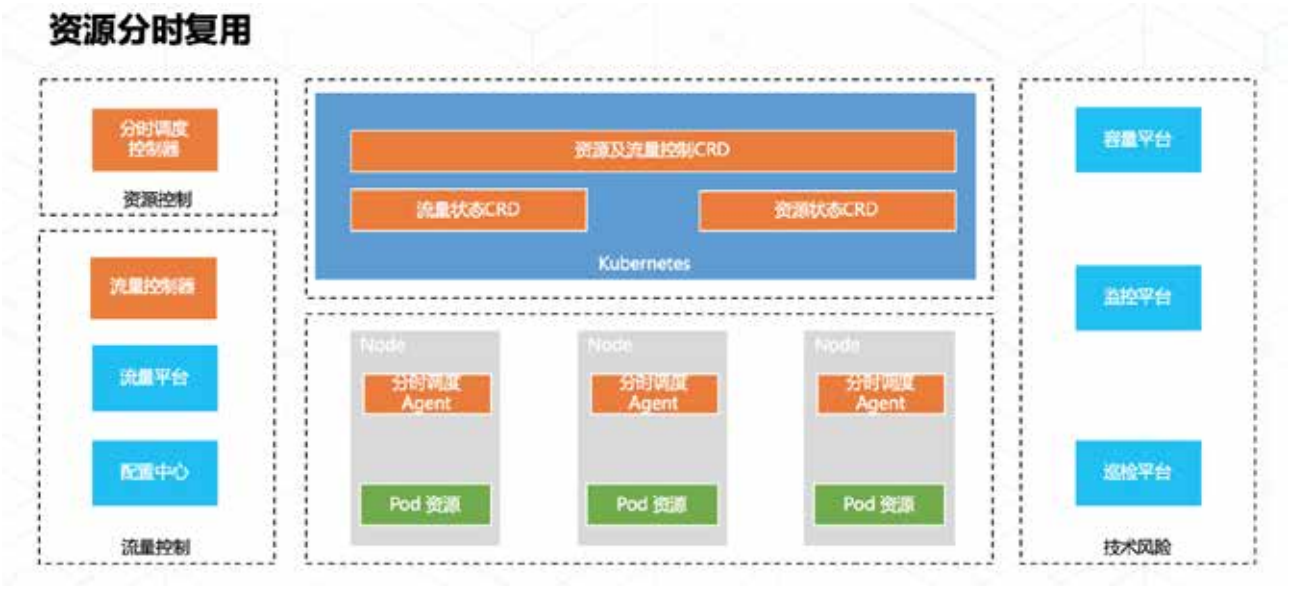
下面我们通过以下几种场景介绍蚂蚁金服内部是如何使用 Kubernetes，以及在此过程中我们面对的挑战和实践之路。

资源分时复用

在大促过程中，不同业务域的洪峰流量通常都是在不同时间段来临，而应对这些不同时间到来的业务流量往往都需要大量的额外计算资源做保障。在以往的几次活动中，我们尝试了通过应用快速腾挪的方式来做到资源上的分时复用，但是服务实例上下线需要预热，腾挪耗时不可控，大规模调度的稳定性等因素都影响了最终腾挪方案的实践效果。

今年 双11 我们采用了资源共享调度加精细化切流的技术以达到资源分时利用的目标，为了达到资源充分利用和极速切换的目标，我们在以下方面做了增强：

- 在内部的调度系统引入了联合资源管理（Union-Resource Management），他可以将波峰波谷不重叠的业务实例摆放在相同的资源集合内，达到最大化的资源利用。
- 研发了一套融合资源更新，流量切换及风险控制的应用分时复用平台，通过该平台 SRE 可以快速稳定的完成资源切换以应对不同的业务洪峰。



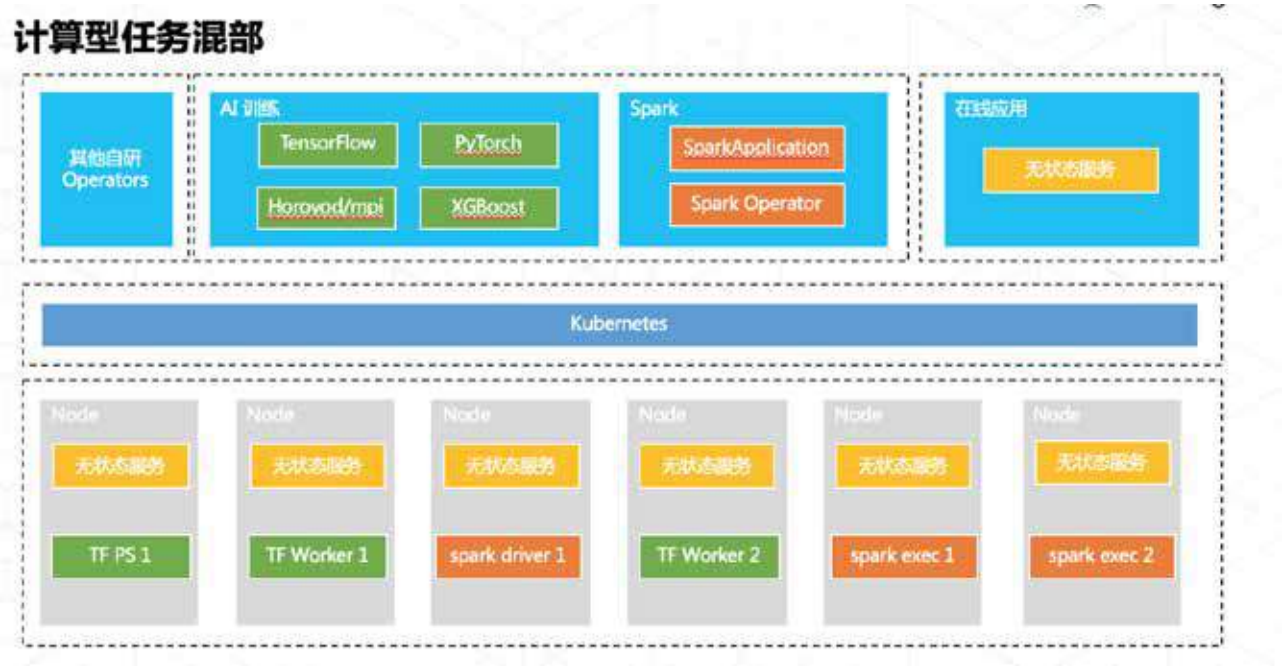
整套平台和技术最终实现了令人激动的成果：蚂蚁金服内部不同业务链路数以万计的实例实现了最大程度的资源共享，这些共享资源的实例可分钟级完成平滑切换。这种技术能力也突破了当下资源水平伸缩能力的效率限制，为资源的分时复用打开了想象空间。

计算型任务混部

Kubernetes 社区的落地案例中，我们往往看到的都是各种各样的在线业务，计算型业务往往通过“圈地”式的资源申请和独立的二层调度跑在 Kuberentes 集群中。但是在蚂蚁内部我们从决定使用 Kubernetes 的第一天起，就将 Kubernetes 融合计算型业务实现资源的统一调度作为我们的目标。

在蚂蚁金服内部我们持续的使用 Kubernetes 支持各类计算业务，例如各类AI 训练任务框架，批处理任务和流式计算等。他们都有一个共同的特点：资源按需申请，即用即走。

我们通过 Operator 模型适配计算型任务，让任务在真正执行时才会调用 Kubernetes API 申请 Pod 等资源，并在任务退出时删除 Pod 释放资源。同时我们在调度引擎中引入了动态资源调度能力和任务画像系统，这为在线和计算的不同等级业务提供了分级资源保障能力，使在线业务不受影响的情况下资源被最大化的利用。

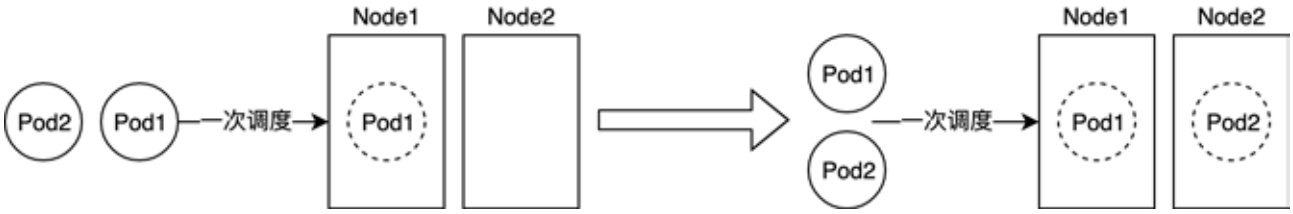


今年 双11 除了洪峰时间段（00：00~02：00），蚂蚁金服 Kubernetes 上运行的任务均未做降级，每分钟仍有数以百计的计算任务在 Kubernetes 上申请和释放。未来蚂蚁金服内部将会持续推动业务调度融合，将 Kubernetes 打造成为资源调度的航空母舰。

规模化核心

蚂蚁金服是目前少数运行了全球最大规模的 Kubernetes 集群的公司之一，单集群机器规模过万，Pods 数量数十万。随着类似计算型任务混部和资源分时复用这类业务的落地，资源的动态使用以及业务自动化运维都对 Kubernetes 的稳定性和性能带来了巨大的挑战。

首先需要面对的挑战是调度性能，社区的调度器在 5k 规模测试中调度性能只有 1~2 pods/s，这显然无法满足蚂蚁金服的调度性能需求。



针对同类业务的资源需求往往相同的特点，我们自研了批量调度功能，再加上例如局部的 filters 性能优化等工作，最终达到了百倍的调度性能提升！

在解决了调度性能问题后，我们发现规模化场景下 APIServer 逐渐成为了影响 Kubernetes 可用性的关键组件，CRD+Operator 的灵活扩展能力更是对集群造成巨大的压力。业务方有 100 种方法可以玩垮生产集群，让人防不胜防。

造成这种现象一方面是由于社区今年以前在规模化上的投入较少 APIServer 能力较弱，另一方面也是由 Operator 模式的灵活性决定。开发者在收益于 Operator 高灵活度的同时，往往为集群管理者带来业务不受控制的风险。即使对 Kubernetes 有一定熟悉程度的开发者，也很难保障自己写出的 Operator 在生产中不会引爆大规模的集群。

面对这种“核按钮”不在集群管理员手上的情况，蚂蚁内部通过两方面入手解决规模化带来的问题：

我们通过持续总结迭代过程中的经验，形成了内部的最佳实践原则，以帮助业务更好的设计 Operator

- CRD 在定义时需要明确未来的最大数量，大量 CR 业务最好采用 aggregate-apiserver 进行扩展；
- CRD 必须 Namespaced scope，以控制影响范围；
- MutatingWebhook + 资源 Update 操作会给运行时环境带来不可控破坏，尽量避免使用这种组合；
- 任何 controllers 都应该使用 informers，并且对写操作配置合理限流；
- DaemonSet 非常高阶，尽量不要采用这类设计，如果必需请在 Kubernetes 专家的辅导下使用

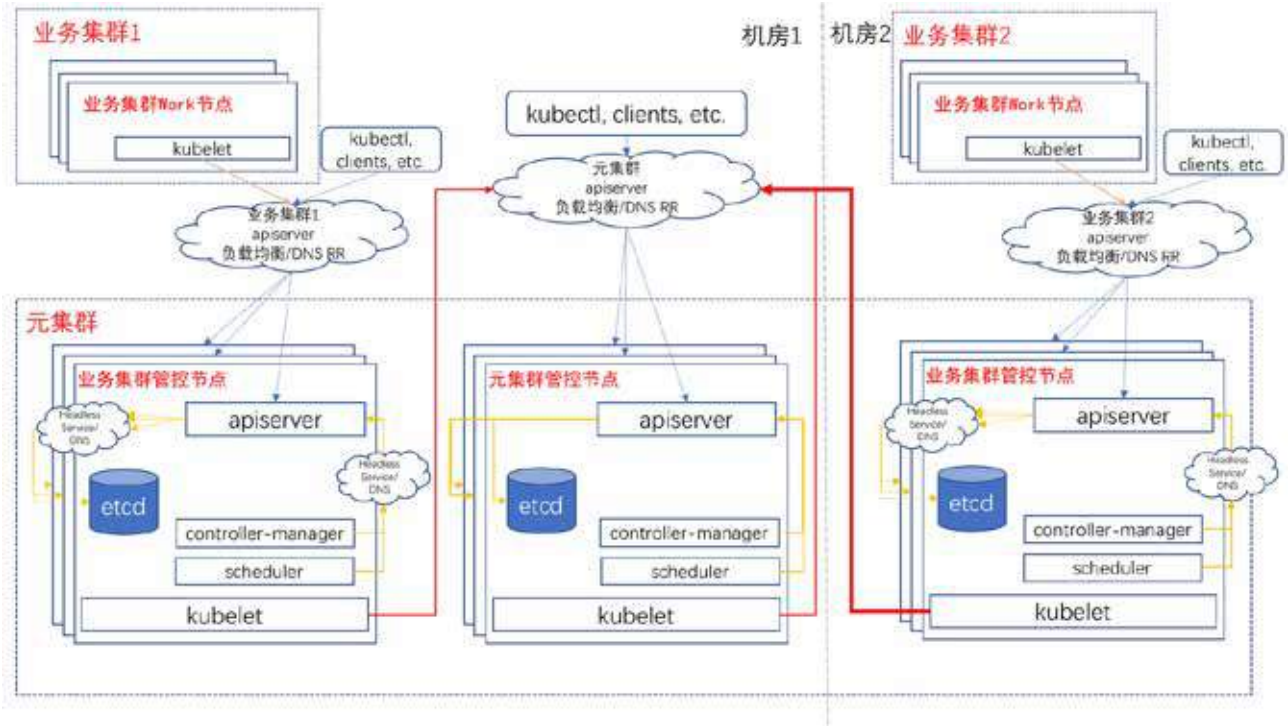
我们已经在 Kubernetes 实施了一系列的优化，包含多维度流量控制，WatchCache 处理全量 List 请求，controller 自动化解决更新冲突，以及 APIServer 加入自定义索引等。

通过规范和优化，我们从 client 到 server 对 API 负载做了整体链路的优化，让资源交付能力在落地的大半年内提升了 6 倍，集群每周可用率达到了 3 个 9，让 Kubernetes 平稳顺滑的支撑了 双11 的大促。

弹性资源建站

近几年大促后动中，蚂蚁金服都会充分利用云化资源，通过快速弹性建站的方式将全站业务做“临时”扩容，并在大促后回收站点释放资源。这样的弹性建站方式为蚂蚁节省了大量的资源开支。

Kubernetes 提供了强大的编排能力，但集群自身的管理能力还比较薄弱。蚂蚁金服从 0 开始，基于 Kubernetes on Kubernetes 和面向终态的设计思路，开发了一套管理系统来负责蚂蚁几十个生产集群的管理，提供面向大促的快速弹性建站功能。



通过这种方式我们可以自动化的完成站点搭建，3 小时内交付可立即使用的包含数千个 Nodes 的 Kubernetes 集群。今年 双11 我们在一天内交付了全部弹性云化集群。随着技术的不断提升和磨练，我们期望未来能够按天交付符合业务引流标准的集群，让蚂蚁金服的站点随时随地可弹。

展望未来，迎接挑战

云原生时代已经到来，蚂蚁金服内部已经通过 Kubernetes 迈出了云原生基础设施建设的第一步。虽然当前在实践中仍然有许多挑战在等着我们去应对，但相信随着我们在技术上持续的投入，这些问题会一一得到解决。

平台与租户

当前我们面对的一大挑战是多租户带来的不确定性。蚂蚁金服内部不可能为每个业务部门都维护一套 Kubernetes 集群，而单个 Kubernetes 集群的多租户能力十分薄弱，这体现在以下两个维度：

- APIServer 和 etcd 缺少租户级别的服务保障能力；
- Namespace 并不能有效的隔离全部资源，并且由于Namespace 只提供了部分资源能力，对平台型的接入方也很不友好。

未来我们会在核心能力如 Priority and Fairness for API Server Requests 以及 Virtual Cluster 上持续的做技术投入和应用，有效保障租户的服务能力保障和隔离。

自动化运维

除了资源调度，Kubernetes 下一阶段的重要场景就是自动化运维。这涉及到应用资源全生命周期的面向终态自行维护，包含但不限于资源自动交付及故障自愈等场景。

随着自动化程度的不断提升，如何有效控制自动化带来的风险，让自动化运维能够真正提升效率而不是任何时刻都需要承担删库跑路的风险是接下来的一个重要难题。

蚂蚁金服在落地 Kubernetes 的过程中经历过类似的情况：从初期高度自动化带来无限的兴奋，到遭遇缺陷不受控制最终爆炸引发故障后的无比沮丧，这些都说明了在 Kubernetes 上做自动化运维仍有很长的路要走。

为此我们接下来一起推动 Operator 的标准化工作。从接入标准，Operator 框架，灰度能力建设和控制治理上入手，让 Kubernetes 上的自动化运维变的更加可视可控。

结束语

今年我们实现了 Kubernetes 由 0-1 的落地，经受了 双11 双大促真实场景的考验。但云原生的基础设施建设仍是刚刚起步，接下来我们将在弹性资源交付，集群规模化服务以及技术风险与自动化运维上持续发力，以技术支撑和推动业务服务完成云原生的落地。

04

云原生应用万节点分钟级分发协同实践

谢于宁，花名予栖，阿里云容器服务高级开发工程师
罗晶，花名瑶靖，阿里云容器服务高级产品经理
邓隽，阿里云容器服务技术专家

引言

2019 年天猫双11，阿里巴巴核心系统首次实现 100% 上云。面对全球最大的交易洪峰，阿里云扛住了每秒 54.4 万笔的交易峰值，这是“云原生”与“天猫全球狂欢节”的一次完美联名。

（图为 2019 年天猫双11 成交额）

容器镜像服务作为阿里巴巴经济体云原生领域的重要基础设施之一，早在 双11 备战期间就已面临大规模分发需求。为了更好地支持这一需求，产品提前进行规划及迭代更新，全面提升了大规模分发场景下的性能、可观测性和稳定性。在新的 双11 来临前，容器镜像服务新增了数 PB 的镜像数据，月均镜像拉取达数亿次。同时产品提供了云原生应用交付链等功能，全面覆盖阿里经济体及云上用户在云原生时代的使用需求。

本文将介绍容器镜像服务如何通过提升产品能力来应对云原生应用万节点分发场景下的新发展和新挑战。

新发展和新挑战

随着云原生技术的迅速普及，Kubernetes 已经成为事实上应用容器化平台的标准，成为了云原生领域的“一等公民”。Kubernetes 以一种声明式的容器编排与管理体系，让软件交付变得越来越标准化。Kubernetes 提供了统一模式的 API，能以 YAML 格式的文件定义 Kubernetes 集群内的资源。这一些 YAML 格式的资源定义使得 Kubernetes 能轻松被上下游系统所集成，完成一系列原本需要用非标准化脚本、人工来完成的操作。同时社区根据应用交付场景及需求，在原生 YAML 格式的资源定义文件之外衍生出了更多系列的云原生应用交付标准，例如 Helm Chart、Opeartor、Open Application Model 等。

（图为云原生应用交付标准演进）

除了云原生应用交付标准推陈出新，用户对交付方式也提出了更高的要求。越来越多的用户期望能以流程化、自动化、更安全的方式交付云原生应用，因此单纯的万节点分发场景已经演化成万节点分钟级多环节协同分发。再加上全球化业务发展，这意味着在分钟级时间内完成各个环节之后，还需再完成全球化分发，这对支撑云生应用分发的平台提出了更高的要求。

新实践

通过控制容器镜像大小、采用 P2P 分发镜像层、优化 Registry 服务端等方式，我们极大优化了大规模分发的性能，最终达成了万节点分钟级分发的目标：

优化容器镜像大小，降低镜像传输成本

- 制作基础镜像，将使用频繁的应用或环境制作成基础镜像复用，尽可能减少镜像的层数，控制每次变更层数
- 采用多阶段镜像构建，将镜像制作过程中的中间产物与最终产物分离，形成最精简的应用镜像

优化服务端处理性能，提高请求响应速率

- 服务端通过识别热点镜像，采用热点数据缓存等多种方式应对大规模镜像 Manifest 并发拉取

优化客户端容器镜像层下载方式，减少镜像传输时间

- 客户端使用蜻蜓下载容器镜像， 基于 P2P 方式大幅减少镜像 Layer 下载时间

（图为镜像大规模分发的优化策略）

为了让拥有同样需求的企业客户能够享受到如上一致的分发能力和体验，容器镜像服务产品在 2019 年 3 月正式推出了容器镜像服务企业版（ACR Enterprise Edition）。容器镜像服务企业版提供了企业级云原生资产托管能力以及云原生应用全球化同步、大规模分发能力，适合有着高安全需求、多地域业务部署、拥有大规模集群节点的企业级容器客户。除此之外，容器镜像服务企业版还在云原生资产托管、交付、分发等几个方面进一步提升云原生应用万节点分钟级分发协同体验。

云原生应用托管

在应用交付物层面，容器镜像服务企业版目前支持容器镜像、Helm Chart 两类云原生应用资产的全生命周期管理。

在访问安全层面，产品提供了独立网络访问控制功能，可以细粒度控制公网及 VPC 网络的访问策略，仅允许符合策略的来源方访问资产，进一步保障云原生资产的访问安全。

在访问体验层面，产品提供容器集群透明拉取插件，支持容器镜像透明拉取，保障业务在弹性场景极速拉取镜像不因凭证配置有误导致业务更新或扩容异常。

（图为容器镜像服务企业版支持云原生应用交付）

云原生应用交付

云原生应用生产环节，用户可以直接上传托管容器镜像、Helm Chart 等云原生资产；也可以通过构建功能自动从源代码（Github、阿里云 Code、GitLab 等来源）智能构建成容器镜像。同时为了解决流程化、自动化、更安全的方式交付云原生应用这一需求，容器镜像服务企业版引入了云原生应用交付链功能。云原生应用交付链以云原生应用托管为始，以云原生应用分发为终，全链路可观测、可追踪、可自主设置。可以实现一次应用变更，全球化多场景自动交付，从流程层面极大地提升了云原生应用万节点分发的效率及安全性。

（图为控制台创建云原生应用交付链）

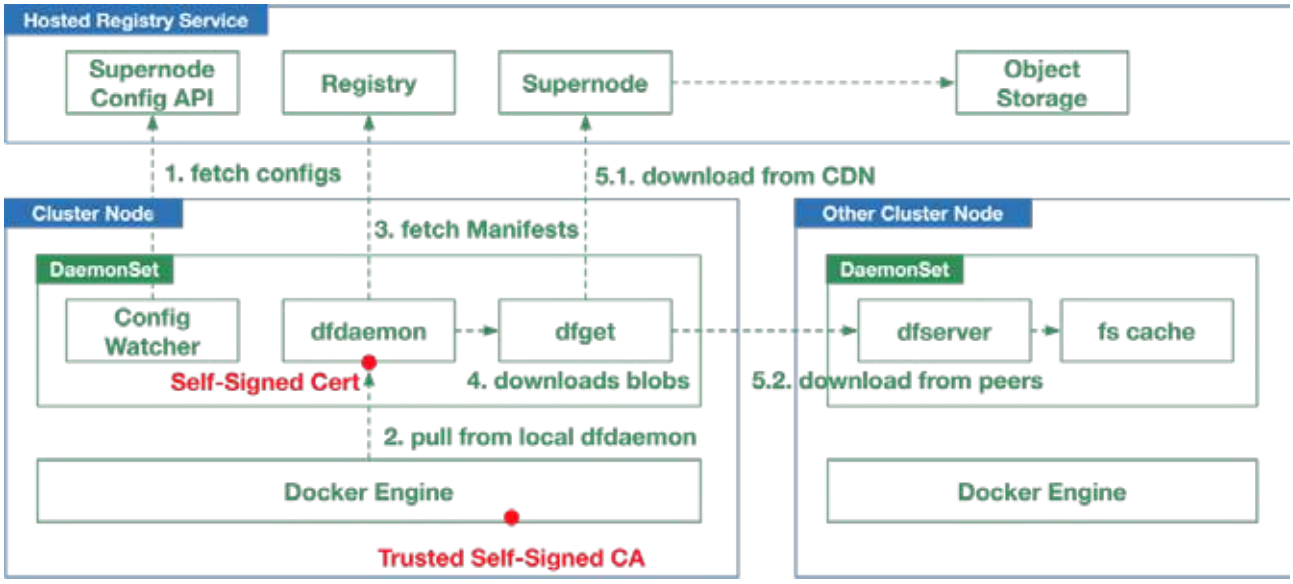
云原生应用交付环节，支持自动发起静态安全扫描并自定义配置安全阻断策略。一旦识别到静态应用中存在高危漏洞后，可自动阻断后续部署链路。用户可基于漏洞报告中的修复建议，更新优化构建成新的镜像版本，再次发起交付。

云原生应用分发

云原生应用分发环节，当前置环节完成无阻断后，云原生应用正式进入全球化分发及大规模分发环节。为了保障万节点分钟级分发协同完成，容器镜像服务联合容器服务、弹性容器实例等云产品提供了端到端的极致分发体验。针对全球化分发，由于基于细粒度同步策略调度、同步链路优化等优化手段，云原生应用的全球同步效率相比手动同步提升了 7 倍。

（图为控制台创建云原生应用交付链）

在 P2P 大规模分发方面，产品针对云环境多次优化基于 Dragonfly 的分发方案，最终通过多个创新技术解决了大规模文件下载以及跨网络隔离等场景下各种文件分发难题，大幅提高大规模容器镜像分发能力。平均镜像大规模分发效率比普通方式提高数倍，适用于容器集群单集群节点数达 100 及以上的场景。



（图为基于 P2P 的分发流程示意）

除了 P2P 大规模分发手段外，为了更好地满足特定场景下的大规模分发需求，产品还支持基于镜像快照的大规模分发方式。基于镜像快照的分发方式，可避免或减少镜像层的下载，极大提高弹性容器实例创建速度。在容器集群（ASK）及弹性容器实例（ECI）的联合使用场景下，产品可以支持 500 节点秒级镜像拉取，实现业务突发场景下极速扩容。

（图为基于镜像快照的分发流程示意）

新平台

在功能及性能指标满足云原生应用万节点分钟级分发协同需求外，容器镜像服务还对平台能力进行了提升和优化，保障了分发过程的可观测性及稳定性。同时平台提供了集成能力，进一步延展云原生应用分发的使用场景和价值。

稳定性

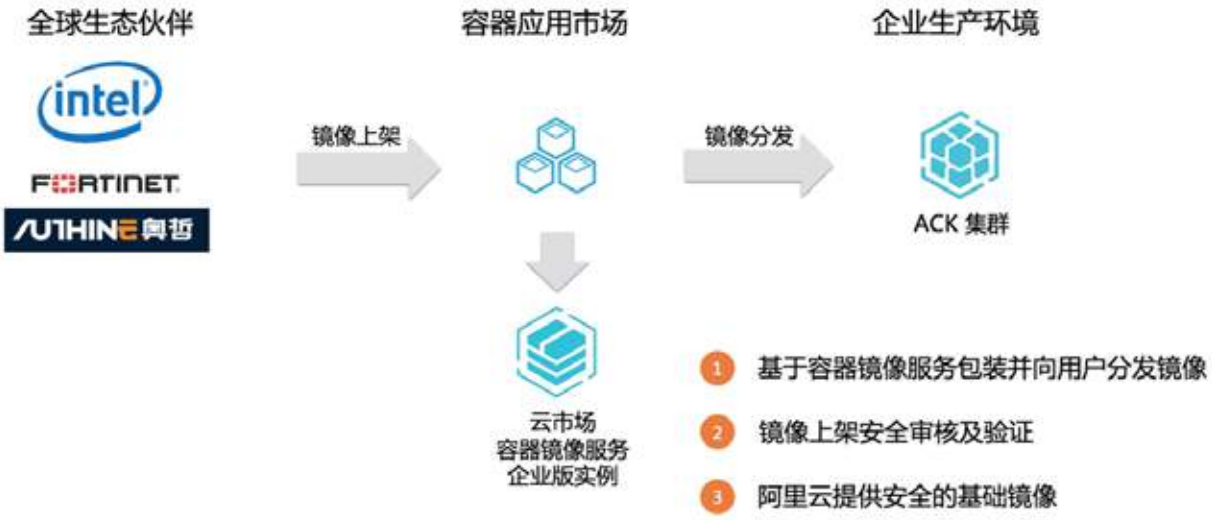
稳定性层面的具体提升及优化工作从监控报警、容错容灾、依赖治理、限流降级、容量规划等几个方面展开。

- 在依赖治理方面，平台对云原生应用交付链中的相关重点环节及外部依赖进行统一管理，提升交付链整体交付能力，帮助用户识别热点仓库及追踪交付链执行结果；
- 在限流降级方面，平台分析识别云原生应用分发核心环节的主次业务功能，优先保障主要业务逻辑完成，次要业务逻辑可降级延后处理；
- 在容量规划方面，平台根据上下游业务变化情况，对资源进行按需扩容，确保云原生应用正常交付完成。

（图为平台的稳定性保障策略）

生态集成

基于平台提供的丰富的集成能力，用户还可以将容器镜像服务企业版作为云原生资产托管及分发的基础设施，为他们的用户提供云原生应用分发能力。其中，容器镜像服务企业版支撑阿里云云市场构建容器应用市场，支撑容器应用市场的容器商品托管及商业化分发，构建云上云原生生态闭环。ISV 服务商，例如 Intel、Fortinet、奥哲，将容器化商品以容器镜像或者 Helm Chart 的形式在云市场快速上架，实现标准化交付、商业化变现。市场客户也可以从容器应用市场获取到优质的阿里云官方及 ISV 容器镜像，快速部署至容器服务容器集群，享受到阿里云丰富的云原生生态。



（图为容器应用市场流程示意）

写在最后

从支持阿里巴巴 双11 大规模分发需求，到全面覆盖阿里经济体及云用户的云原生资产托管及分发需求，再到支撑构建云上容器生态闭环，阿里云容器镜像服务已成为了云原生时代的核心基础设施之一，释放云原生价值的重要加速器。容器镜像服务也将持续为用户带来更加优异的云原生应用分发功能、性能及体验。

01

阿里巴巴大规模神龙裸金属 K8s 集群运维实践

姚捷，花名喽哥，阿里云容器平台集群管理高级技术专家

值得阿里技术人骄傲的是 2019 年阿里双11 核心系统 100% 以云原生的方式上云，完美支撑了 54.4w 峰值流量以及 2684 亿的成交量。背后的承载海量交易的计算力就是来源于容器技术与神龙裸金属的完美融合。

集团上云机器资源形态

阿里巴巴 双11 采用三地五单元架构，除 2 个混部单元外，其他 3 个均是云单元。神龙机型经过 618、99 大促的验证，性能和稳定性已大幅提升，可以稳定支撑 双11。今年 双11 的 3 个交易云单元，已经 100% 基于神龙裸金属，核心交易电商神龙集群规模已达到数万台。

神龙架构

阿里云 ECS 虚拟化技术历经三代、前二代是 Xen 与 KVM，而神龙是阿里自研的第三代 ECS 虚拟化技术产品，它具备以下四大技术特征：

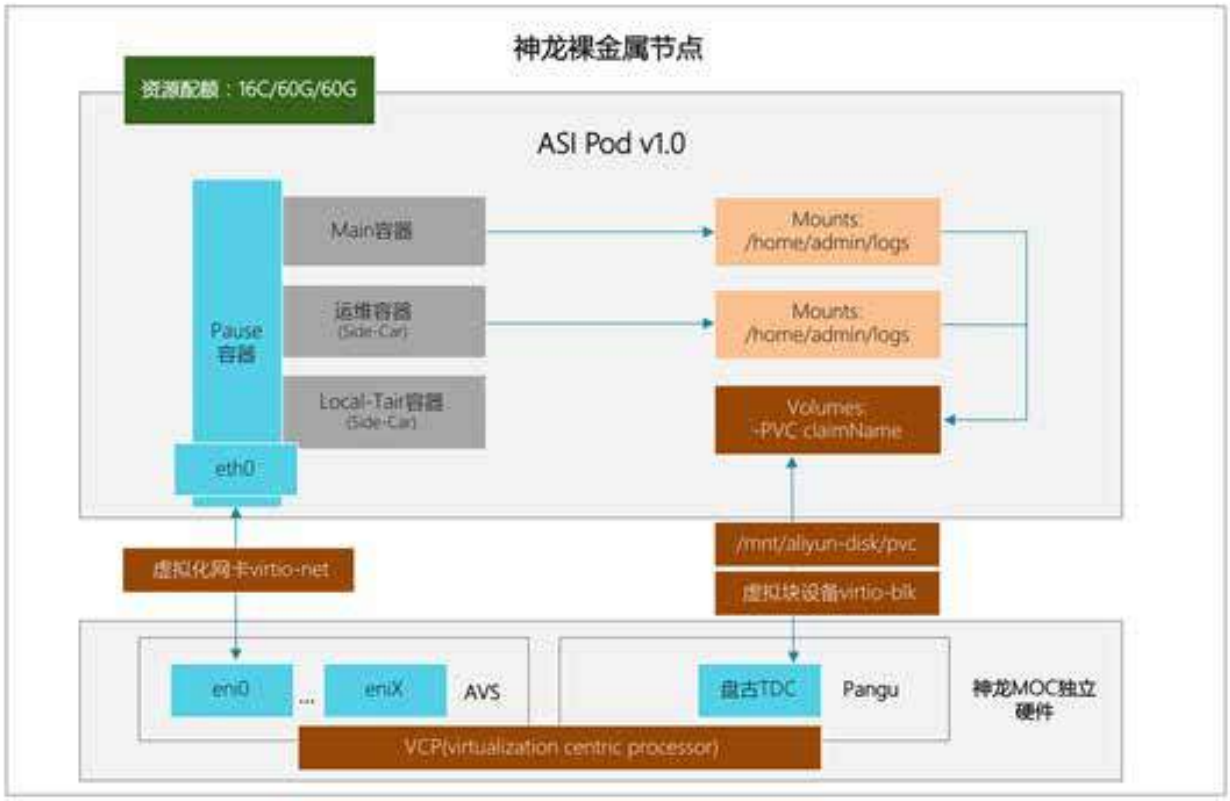
- 存储和网络 VMM 以及 ECS 管控，和计算虚拟化分离部署；
- 计算虚拟化进一步演化至 Near Metal Hypervisor；
- 存储和网络 VMM 通过芯片专用 IP 业务加速；
- 并池支持弹性裸金属和 ECS 虚拟机生产。

简而言之，神龙将网络/存储的虚拟化开销 offload 到一张叫 MOC 卡的 FPGA 硬件加速卡上，降低了原 ECS 约 8% 的计算虚拟化的开销，同时通过大规模 MOC 卡的制造成本优势，摊平了神龙整体的成本开销。神龙类物理机特性，可进行二次虚拟化，使得对于新技术的演进发展留足了空间，对于采用一些多样的虚拟化的技术，像 Kata、Firecracker 等成为了可能。

在阿里 双11 大规模迁移到神龙架构前，通过在 618/99 大促的验证，我们发现集团电商的容器运行在云上神龙反而比非云物理机的性能要好 10%～15%，这令我们非常诧异。经过分析，我们发现主要的原因是因为虚拟化开销已经 offload 到 MOC 卡上，神龙的 CPU/Mem 是无虚拟化开销的，而上云后运行在神龙上的每个容器都独享 ENI 弹性网卡，性能优势明显。同时每个容器独享一块 ESSD 块存储云盘，单盘 IOPS 高达 100 万，比 SSD 云盘快 50 倍，性能超过了非云的 SATA 和 SSD 本地盘。这也让我们坚定了大规模采用神龙来支撑 双11 的决心。

神龙+容器+Kubernetes

在 All in Cloud 的时代企业 IT 架构正在被重塑，而云原生已经成为释放云计算价值的最短路径。2019 年阿里 双11 核心系统 100% 以云原生的方式上云，基于神龙服务器、轻量级云原生容器以及兼容 Kubernetes的调度的新的ASI（alibaba serverless infra.）调度平台。其中 Kubernetes Pod 容器运行时与神龙裸金属完美融合，Pod 容器作为业务的交付切面，而其运行在神龙实例上。下面是 Pod 运行在神龙上的形态：



ASI Pod 运行在神龙裸金属节点上，将网络虚拟化和存储虚拟化 offload 到独立硬件节点 MOC 卡上，并采用 FPGA 芯片加速技术，存储与网络性能均超过普通物理机和 ECS；MOC 有独立的操作系统与内核，可为 AVS（网络处理）与 TDC（存储处理）分批独立的 CPU 核；

ASI Pod 由 Main 容器（业务主容器），运维容器（star-agent side-car 容器）和其它辅助容器（例如某应用的 Local 缓存容器）构成。Pod 内通过 Pause 容器共享网络命名空间，UTS 命名空间和 PID 命名空间（ASI 关闭了 PID 命名空间的共享）；

Pod 的 Main 容器和运维容器共享数据卷，并通过 PVC 声明云盘，将数据卷挂载到对应的云盘挂载点上。在 ASI 的存储架构下，每一个 Pod 都有一块独立的云盘空间，可支持读写隔离和限制磁盘大小；

ASI Pod 通过 Pause 容器直通 MOC 卡上的 ENI 弹性网卡；

ASI Pod 无论内部有多少容器，对外只占用独立的资源，例如 16C（CPU）/60G（内存）/60G（磁盘）。

大规模神龙运维

2019 年 双11 云上核心交易的神龙集群规模巨大，管理和运维如此大规模神龙集群极具挑战，这其中包括云上各类业务的实例规格的选择、大规模集群弹性扩缩容、节点资源划分与管控、核心指标统计分析、基础环境监控、宕机分析、节点标签管理、节点重启/锁定/释放、节点自愈、故障机轮转、内核补丁升级、大规模巡检等能力建设。下面就几个领域细化展开：

实例规格

首先需要针对不同类型业务规划不同的实例规格，包括入口层、核心业务系统、中间件、数据库、缓存服务这些不同特性的基础设施和业务，有些需要高性能的计算、有些需要高网络收发包能力，有些需要高性能的磁盘读写能力。在前期需要系统性整体规划，避免实例规格选择不当影响业务性能和稳定性。实例规格的核心配置参数包括 vcpu、内存、弹性网卡数，云盘数、系统盘大小、数据盘大小、网络收发包能力(PPS)。

电商核心系统应用的主力机型为 96C/527G，每个 Kubernetes Pod 容器占用一块弹性网卡和一块 EBS 云盘，所以弹性网卡和云盘的限制数非常关键，此次电商上云神龙将弹性网卡和 EBS 云盘的限制数提高到 64 与 40，有效避免了 CPU 与内存的资源浪费。另外对于不同类型的业务，核心配置也会略有差异，例如入口层 aserver 神龙实例由于需要承担大量的入口流量，对 MOC 的网络收发包能力的要求极高，为避免 AVS 网络软交换 CPU 打满，对于神龙 MOC 卡里的网络和存储的 CPU 分配参数的需求不同，常规计算型神龙实例的 MOC 卡网络/存储的 CPU 分配是 4+8，而 aserver 神龙实例需要配置为 6:6；例如对于云上混部机型，需要为离线任务提供独立的 nvme 本盘实例。为不同类型业务合理规划机型和规格，会极大程度的降低成本，保证性能和稳定性。

资源弹性

双11 需要海量的计算资源来扛住洪峰流量，但这部分资源不可能常态化持有，所以需要合理划分日常集群与大促集群，并在 双11 前几周，通过大规模节点弹性扩容能力，从阿里云弹性申请大批量神龙，部署在独立的大促集群分组里，并大规模扩容 Kubernetes Pod 交付业务计算资源。在 双11 过后，立即将大促集群中的 Pod 容器批量缩容下线，大促集群神龙实例整体销毁下线，日常只持有常态化神龙实例，通过大规模集群弹性扩缩容能力，可大幅节约大促成本。另外从神龙交付周期而言、今年上云后从申请到创建机器，从小时/天级别缩短到了分钟级，上千台神龙可在 5 分钟内完成申请，包括计算、网络、存储资源，并在 10 分钟完成创建并导入 Kubernetes 集群，集群创建效率大幅度提高，为未来常态化弹性资源池奠定基础。

核心指标

对于大规模神龙集群运维，有三个非常核心的指标可以来衡量集群整体健康度，分别是宕机率、可调度率、在线率。云上神龙宕机原因通常分为硬件问题和内核问题。通过对日宕机率趋势统计和宕机根因分析，可量化集群的稳定性，避免出现潜在的大规模宕机风险出现。可调度率是衡量集群健康度的关键指标，集群机器会因为各种软硬件原因出现容器无法调度到这些异常机器上，例如 Load 大于 1000、磁盘出现压力、docker 进程不存在，kubelet 进程不存在等，在 Kubernetes 集群中，这批机器的状态会是 notReady。2019 年 双11，我们通过神龙宕机重启与冷迁移特性，极大提升了故障机轮转效率，使神龙集群的可调度率始终维持在 98% 以上，大促资源备容从容。而 双11 神龙的宕机率维持在 0.2‰ 以下，表现相当稳定。

标签管理

随着集群规模的增加，管理难度也随之变大。例如如何能筛选出“cn-shanghai”Region 下生产环境，且实例规格为“ecs.ebmc6-inc.26xlarge”的所有机器。我们通过定义大量的预置标签来实现批量资源管理。在 Kubernetes 架构下，通过定义 Label 来管理机器。Label 是一个 Key-Value 健值对，可在神龙节点上使用标准Kubernetes的接口打Label。例如机器实例规格的label可定义"sigma.ali/machine-model":"ecs.ebmc6-inc.26xlarge", 机器所在的 Region 可定义为"sigma.ali/ecs-region-id":"cn-shanghai"。通过完善的标签管理系统，可从几万台神龙节点中快速筛选机器，执行诸如灰度分批次服务发布、批量重启、批量释放等常规运维操作。

宕机分析

对于超大规模集群，日常宕机是非常普遍的事情，对宕机的统计与分析非常关键，可以甄别出是否存在系统性风险。宕机的情况分为很多种，硬件故障会导致宕机，内核的 bug 等也会导致宕机，一旦宕机以后，业务就会中断，有些有状态应用就会受到影响。我们通过 ssh 与端口 ping 巡检对资源池的宕机情况进行了监控，统计宕机历史趋势，一旦有突增的宕机情况就会报警；同时对宕机的机器进行关联分析，如根据机房、环境、单元、分组 进行归类，看是否跟某个特定的机房有关；对机型、CPU 进行分类统计，看是否跟特定的硬件有关系；同时 OS 版本，内核版本进行归类，看是否都发生在某些特定的内核上。对宕机的根因，也进行了综合的分析，看是硬件故障，还是有主动运维事件。内核的 kdump 机制会在发生 crash 的时候生成 vmcore，我们也对 vmcore 里面提取的信息进行归类，看某一类特定的 vmcore 关联的宕机数有多少。内核日志有些出错的信息，如 mce 日志，soft lockup 的出错信息等，也能发现系统在宕机前后的是否有异常。通过这一系列的宕机分析工作，把相应的问题提交给内核团队，内核专家就会分析 vmcore，属于内核的缺陷的会给出 hotfix 解决这些导致宕机的问题。

节点自愈

运维大规模神龙集群不可避免遇到软硬件故障，而在云上技术栈更厚，出现问题会更为复杂。如果出问题单纯依靠人工来处理是不现实的，必须依赖自动化能力来解决。1-5-10 节点自愈可提供 1 分钟异常问题发现，5 分钟定位，10 分钟修复的能力。主要的神龙机器异常包括宕机、夯机、主机 load 高、磁盘空间满、too many openfiles、核心服务（Kubelet、Pouch、Star-Agent）不可用等。主要的修复动作包括宕机重启、业务容器驱逐、异常软件重启、磁盘自动清理，其中 80% 以上问题可通过重启宕机机器与将业务容器驱逐到其他节点完成节点自愈。另外我们通过监听神龙 Reboot 重启与 Redepoly 实例迁移二个系统事件来实现 NC 侧系统或硬件故障的自动化修复。

未来展望

2020 年 双11，阿里巴巴经济体基础设施将会 100% 基于 Kubernetes，基于 runV 安全容器的下一代混部架构将会大规模落地，轻量化容器架构会演进到下一阶段。在此大背景下，一方面 Kubernetes 节点管理将会朝向阿里巴巴经济体并池管理方向发展，打通云库存管理，提升节点弹性能力，根据业务特性错峰资源利用，进一步降低机器持有时间从而大幅降低成本。在技术目标上，我们会采用基于 Kubernetes Machine-Operator 的为核心引擎，提供高度灵活的节点运维编排能力，支持节点运维状态的终态维持。另一方面，基于完整的全域数据采集和分析能力，提供极致的全链路监控/分析/内核诊断能力，全面提升容器基础环境的稳定性，为轻量化容器/不可变基础设施架构演进提供极致性能观测与诊断的技术保障。

02

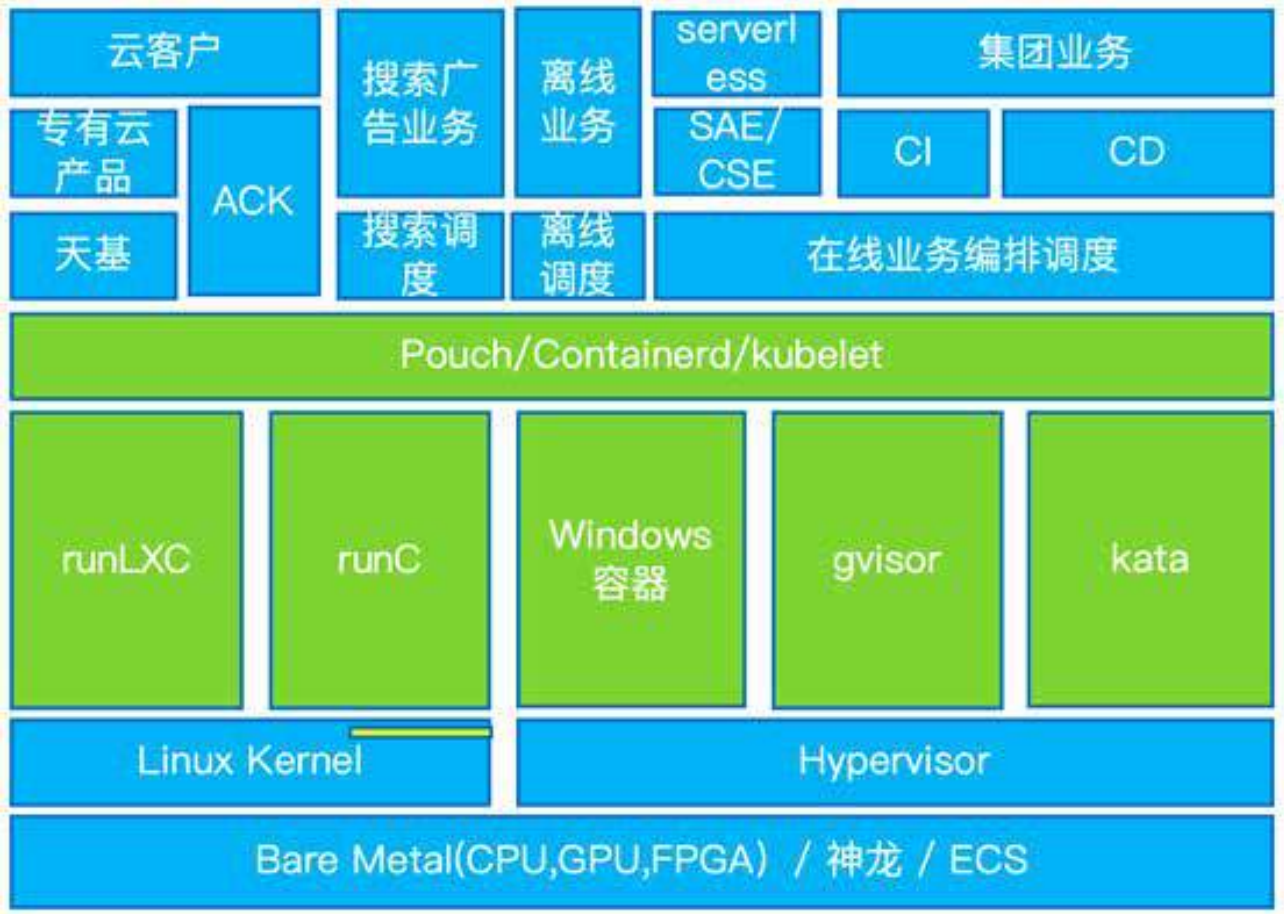
PouchContainer 容器技术演进助力阿里云原生升级

杨育兵，花名沈陵，阿里云基础技术中台高级技术专家

我们从 2016 年开始在集团推广全面的镜像化容器化，今年是集团全面镜像化容器化后的第四个 双11，PouchContainer 容器技术已经成为集团所有在线应用运行的运行时底座和运维载体，每年双十一都有超过百万的 PouchContainer 容器同时在线，提供电商和所有相关的在线应用平稳运行的载体，保障大促购物体验的顺滑。

我们通过 PouchContainer 容器运行时这一层标准构建了应用开发和基础设施团队的标准界面，每年应用都有新的需求新的变化，同时基础设施也有上云/混部/神龙/存储计算分离/网络变革这些升级，两边平行演进，互不干扰。技术设施和 PouchContainer 自身都做了很大的架构演进，这些很多的架构和技术演进对应用开发者都是无感知的。

在容器技术加持的云原生形成趋势的今天，PouchContainer 容器技术支持的业务方也不再只有集团电商业务和在线业务了，我们通过标准化的演进，把所有定制功能做了插件化，适配了不同场景的需要。除了集团在线应用，还有运行爱离线调度器上面的离线job类任务、跑在搜索调度器上面的搜索广告应用，跑在 SAE/CSE 上面的 Serverless 应用、专有云产品、公有云（ACK+CDN）等场景都使用 PouchContainer 提供的能力。



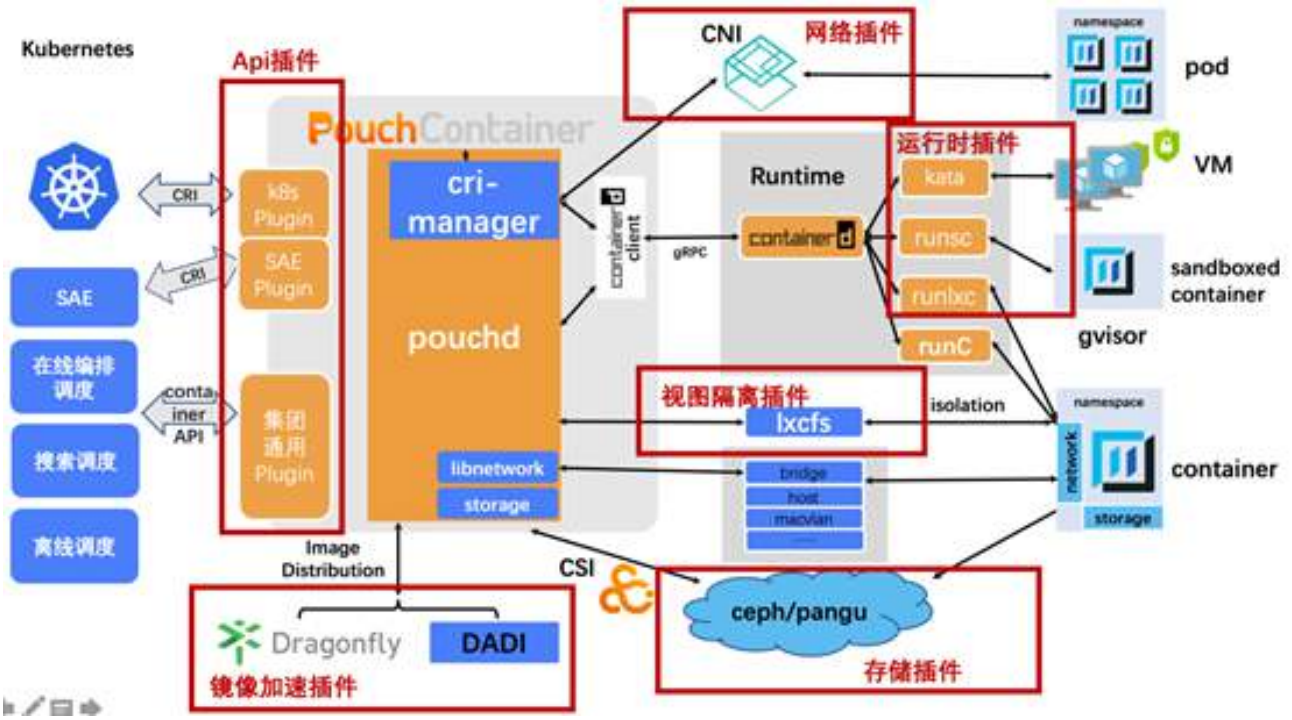
运行时的演进

2015 年之前，我们用的运行时是 LXC，PouchContainer 为了在镜像化后能够平滑接管原来的 T4 容器，在 LXC 中支持新的镜像组装方式，并支持交互式的 exec 和内置的网络模式（PouchContainer 支持 libnetwork 和 CNI 标准的网络插件是后来运行时升级到 runc 阶段做的）。随着云原生的进程，我们在用户无感知的情况下对运行时做了 containerd+runc 的支持，用标准化的方式加内部功能插件，实现了内部功能特性的支持和被各种标准化运维系统无缝集成的目标。

无论是 LXC 还是 runc 都是让所有容器共享 Linux 内核，利用 cgroup 和 namespace 来做隔离，对于强安全场景和强隔离场景是不适用的。为了容器这种开发和运维友好的交付形式能给更多场景带来收益，我们很早就开始探索这方面的技术，和集团 os 创新团队以及蚂蚁 os 虚拟化团队合作共建了 kata 安全容器和 gvisor 安全容器技术，在容器生态嫁接，磁盘网络系统调用性能优化等方面都做了很多的优化。在兼容性要求高的场景我们优先推广kata 安全容器，已经支持了 SAE 和 ACK 安全容器场景。在语言和运维习惯确定的场景我们也在 618 大促时上线了一些合适的电商使用了 gvisor 的运行时隔离技术，稳定性和性能都得到了验证。

为了一部分专有云场景的实施，我们今年还首次支持了 Windows 容器运行时，在容器依赖相关的部署、运维方面做了一些探索，帮助敏捷版专有云拿下了一些客户。

除了安全性和隔离性，我们的运行时演进还保证了标准性，今年最新版本的 PouchContainer 把 diskquota、lxcfs、dragonfly、DADI 这写特性都做成了可插拔的插件，不需要这些功能的场景可以完全不受这些功能代码的影响。甚至我们还对一些场景做了 containerd 发行版，支持纯粹的标准 CRI 接口和丰富的运行时。



镜像技术的演进

镜像化以后必然会引入困难就是镜像分发的效率，一个是速度另一个是稳定性，让发布扩容流程不增加太多时间的情况下，还要保证中心节点不被压垮。

PouchContainer 在一开始就支持了使用 Dragonfly 来做 P2P 的镜像分发，就是为了应对这种问题，这是我们的第一代镜像分发方案。在研发域我们也对镜像分层的最佳实践做了推广，这样能最大程度的保证基础环境不变时每次下载的镜像层最小。镜像加速要解决的问题有：build 效率/push 效率/pull 效率/解压效率/组装效率。第一代镜像加速方案，结合 Dockerfile 的最佳设计解决了 build 效率和 pull 效率和中心压力。

第一代镜像分发的缺点是无论用户启动过程中用了多少镜像数据，在启动容器之前就需要把所有的镜像文件都拉到本地，在很多场景都是又浪费的，特别影响的是扩容场景，所以第二代的镜像加速方案，我们调研了阿里云的盘古，盘古的打快照、mount、再打快照完美匹配打镜像和分发的流程。能做到秒级镜像 pull，因为 pull 镜像时只需要鉴权，下载镜像 manifest，然后 mount 盘古。也能做到镜像内容按需读取。2018 年 双11，我们小规模上线了盘古远程镜像，也验证了我们的设计思路，这一代的镜像加速方案结合新的 overlay2 技术在第一代的基础上有解决了PouchContainer 效率/pull 效率/解压效率和组装效率。

但是也存在一些问题，首先镜像数据没有存储在中心镜像仓库中，只有 manifest 信息，这样镜像的分发范围就受限，在哪个盘古集群做的镜像，就必须在那个盘古集群所在的阿里云集群中使用这个镜像，其次没有 P2P 的能力，在大规模使用时对盘古后端的压力会很大，特别是离线场景下由于内存压力导致很多进程的可执行文件的 page cache 被清理然后需要重新 load 这种场景，会给盘古后端带来更大的压力。基于这两个原因，我们和 ContainerFS 团队合作共建了第三代镜像分发方案：DADI（基于块设备的按需p2p加载技术，后面有计划开源）。

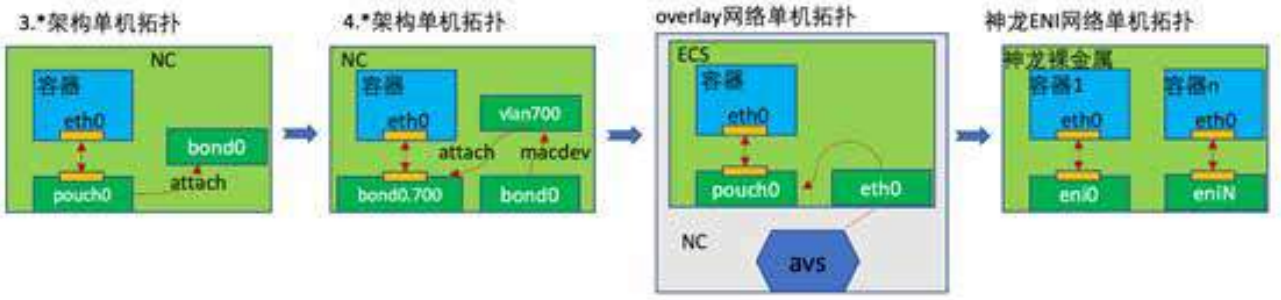
daidi在构建阶段保留了镜像的多层结构，保证了镜像在多次构建过程中的可重用性，并索引了每个文件的在每层的offset 和 length，推送阶段还是把镜像推送到中心镜像仓库中，保证在每个机房都能拉取到这个镜像。在每个机房都设置了超级节点做缓存，每一块内容在特定的时间段内都只从镜像仓库下载一次。如果有时间做镜像预热，像双十一这种场景，预热阶段就是从中心仓库中把镜像预热到本地机房的超级节点，后面的同机房的数据传输会非常快。镜像 pull 阶段只需要下载镜像的 manifest 文件（通常只有几 K大小），速度非常快，启动阶段大地会给每个容器生成一个快设备，这个块设备的 chunk 读取是按需从超级节点或临近节点 P2P 的读取内容。这样就保证了容器启动阶段节点上只读取了需要的部分内容。为了防止容器运行过程中出现 iohang，我们在容器启动后会在后台把整个镜像的内容全部拉到 node 节点，享受超快速启动的同时最大程度的避免后续可能出现的 iohang。

使用 DADI 镜像技术后的今年双十一高峰期，每次有人在群里面说有扩容任务，我们值班的同学去看工单时，基本都已经扩容完成了，扩容体验做到了秒级。

网络技术演进

PouchContainer 一开始的网络功能是揉合在 PouchContainer 自己的代码中的，用集成代码的方式支持了集团各个时期的网络架构，为了向标准化和云原生转型，在应用无感知的情况下，我们在 Sigma-2.0 时代使用 libnetwork 把集团现存的各种网络机架构都统一做了 CNM 标准的网络插件，沉淀了集团和专有云都在用的阿里自己的网络插件。在在线调度系统推广期间 CNM 的网络插件已经不再适用，为了不需要把所有的网络插件再重新再实现一遍，我们对原来的网络插件做了包装，沉淀了CNI的网络插件，把 CNM 的接口转换为 CNI 的接口标准。

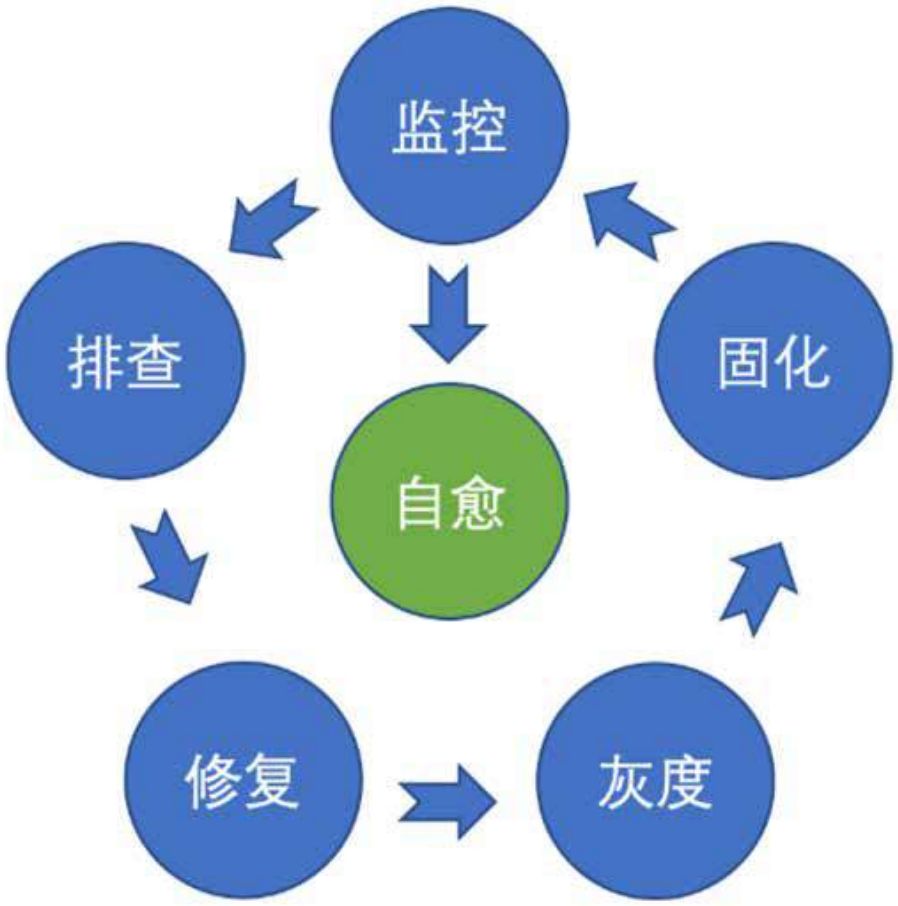
内部的网络插件支持的主流单机网络拓扑演进过程：



从单机拓扑能看出来使用神龙 eni 网络模式可以避免容器再做网桥转接，但是用神龙的弹性网卡和CNI网络插件时也有坑需要避免，特别是 eni 弹性网卡是扩容容器时才热插上来的情况时。创建 eni 网卡时，udevd 服务会分配一个唯一的 id N，比如 ethN，然后容器 N 启动时会把 ethN 移动到容器 N 的 netns，并从里面改名为 eth0。容器 N 停止时，eth0 会改名为 ethN 并从容器 N 的 netns 中移动到宿主机的 netns 中。这个过程中，如果容器 N 没有停止时又分配了一个容器和 eni 到这台宿主机上，udevd 由于看不到 ethN 了，他有可能会分配这个新的 eni 的名字为ethN。容器 N 停止时，把 eth0 改名为 ethN 这一步能成功，但是移动到宿主机根 netns 中这一步由于名字冲突会失败，导致 eni 网卡泄漏，下一次容器N启动时找不到它的 eni 了。可以通过修改 udevd 的网卡名字生成规则来避免这个坑。

运维能力演进

PouchContainer 容器技术支持了百万级的在线容器同时运行，经常会有一些问题需要我们排查，有很多都是已知的问题，为了解决这个困扰，我还写了 PouchContainer 的一千个细节以备用户查询，或者重复问题问过来时直接交给用户。但是 PouchContainer 和相关链路本身稳定性和运维能力的提升才是最优的方法。今年我们建设了 container-debugger 和 NodeOps 中心系统，把一些容器被用户问的问题做自动检测和修复，任何修复都做了灰度筛选和灰度部署能力，把一些经常需要答疑的问题做了用户友好的提示和修复，也减轻了我们自身的运维压力。



1. 内部的中心化日志采集和即时分析
2. 自带各模块的健康和保活逻辑
3. 所有模块提供 Prometheus 接口，暴露接口成功率和耗时
4. 提供常见问题自动巡检修复的工具
5. 运维经验积累，对用户问题提供修复建议
6. 提供灰度工具，任何变更通过金丝雀逐步灰度
7. 剖析工具，流程中插入代码的能力
8. Pouch 具备一键发布能力，快速修复

容器使用方式演进

提供容器平台给应用使用，在容器启动之前必然有很多平台相关的逻辑需要处理，这也是我们以前用富容器的原因。

1. 安全相关：安全路由生成、安全脚本配置
2. cpushare 化相关配置：tsar 和 nginx 配置
3. 运维agent 更新相关：运维agent 更新相对频繁，基础镜像更新特别慢，不能依赖于基础镜像更新来更新运维agent
4. 配置相关逻辑：同步页头页尾，隔离环境支持， 强弱依赖插件部署
5. SN 相关：模拟写 SN 到/dev/mem，保证 dmidecode 能读到正确的 SN
6. 运维相关的的 agent 拉起，很多运维系统都依赖于在节点上面有一个 agent，不管这个节点是容器/ecs 还是物理机
7. 隔离相关的配置：比如 nproc 这个限制是在用户上面的，用统一个镜像的容器不能使用统一 uid 不然无法隔离 nproc

现在随着基于 K8s 的编排调度系统的推广，我们有了 Pod 能力，可以把一些预置逻辑放到前置 hook 中去执行，当然富容器可以瘦下来还要依赖于运维 agent 可以从主容器中拆出来，那些只依赖于 volume 共享就能跑起来的 agent 可以先移动到 sidecar 里面去，这样就可以把运维容器和业务主容器分到不同的容器里面去，一个 Pod 多个容器在资源隔离上面分开，主容器是 Guaranteed 的 QOS，运维容器是 Burstable 的 QOS。同时在 kubelet 上支持 Pod 级别的资源管控，保证这个 Pod 整体是 Guaranteed 的同时，限制了整个 pod 的资源使用量不超过应用单实例的申请资源。

还有一些 agent 不是只做 volume 共享就可以放到 sidecar 的运维容器中的，比如 monkeyking、arthas 需要能 attach 到主容器的进程上去，还要能 load 主容器中非 volume 路径上面的 jar 文件才能正常工作。对于这种场景 PouchContainer 容器也提供了能让同 Pod 多容器做一些 ns 共享的能力，同时配合 ns 穿越来让这些 agent 可以在部署方式和资源隔离上是和主容器分离的，但是在运行过程中还可以做它原来可以做的事情。

容器技术继续演进的方向

可插拔的插件化的架构和更精简的调用链路在容器生态里面还是主流方向，kubelet 可以直接去调用 containerd 的 CRI 接口，确实可以减少一调，不过 CRI 接口现在还不够完善，很多运维相关的指令都没有，logs 接口也要依赖于 container API 来实现。还有运行环境和构建环境分离，这样用户就不需要到宿主机上面执行 build。所有的运维系统也不再依赖于 container API。在这些约束下我们可以做到减少一跳，直接用 kubelet 去调用 containerd 的 CRI 接口。

现在每个应用都有很多的 Dockerifle，怎么让 Dockerfile 更有表达能力，减少 Dockerfile 数量。构建的时候并发构建也是一个优化方向，buildkit 在这方面是可选的方案，Dockerfile 表达能力的欠缺也需要新的解决方案，buildkit 中间态的 LLB 是 go 代码，是不是可以用 go 代码来代替 Dockerfile，定义更强表达能力的 Dockerfile 替代品。

容器化是云原生的关键路径，容器技术在运行时和镜像技术逐渐趋于稳定的情况下，热点和开发者的目光开始向上层转移，K8s 和基于其上的生态成为容器技术未来能产生更多创新的领域，PouchContainer 技术也在向着更云原生，更好适配 K8s 生态的方向发展，网络/diskquota/试图隔离等 PouchContainer 的插件在 K8s 生态系统中适配和优化也我们后面的方向之一。

03

更强，更稳，更高效：2019 年双11 etcd 技术升级分享

陈星宇，花名宇慕，阿里云基础技术中台技术专家

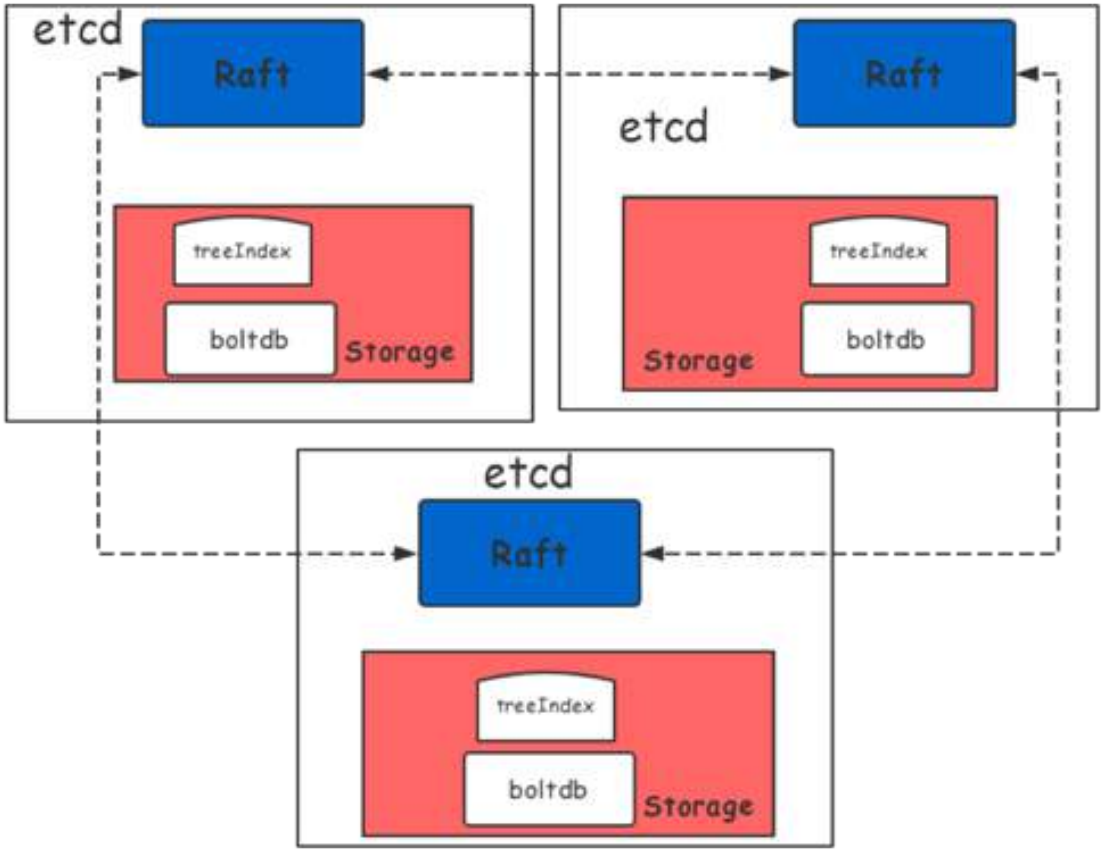
etcd 是阿里巴巴内部容器云平台用于存储关键元信息的组件。阿里巴巴使用 etcd 已经有 3 年的历史，在今年 双11 过程中它又一次承担了关键角色，接受了 双11 大压力的检验。除了在阿里巴巴内部使用，今年我们也将内部的一些优秀资源，如：优化代码、最佳实践推广到了社区，获得了不错的反响。为了让更多同学了解到 etcd 的最佳实践和阿里巴巴内部的使用经验，这里分享一下阿里巴巴是如何把 etcd 升级得更强，更稳，更高效的，在文章结尾，将进一步展望未来更智能的工作。希望可以通过这篇文章让更多人了解 etcd, 让更多的人享受云原生技术带来的红利。

让 etcd 变得更强

本节主要介绍 etcd 在性能方面的升级工作。首先我们来理解一下 etcd 的性能背景。

性能背景

这里先庖丁解牛，将 etcd 分为如下几个部分，如下图所示：



每一部分都有各自的性能影响，让我们逐层分解：

1. raft 层: raft 是 etcd 节点之间同步数据的基本机制，它的性能受限于网络 IO, 节点之间的 rtt 等，WAL 受到磁盘 IO 写入延迟。
2. 存储层：负责持久化存储底层 kv, 它的性能受限于磁盘 IO，例如：fdatasync 延迟，内存 treeIndex 索引层锁的 block, boltdb Tx 锁的 block 以及 boltdb 本身的性能。
3. 其他还有诸如宿主机内核参数, grpc api 层等性能影响因子。

服务端优化

了解完背景后，这里介绍一下性能优化手段，主要由两个方面服务端和客户端组成，这里先介绍服务端优化的一些手段：

硬件部署

etcd 是一款对cpu、内存、磁盘要求较高的软件。随着内部存储数据量的增加和对并发访问量的增大，我们需要使用不同规格的硬件设备。这里我们推荐 etcd 至少使用 4 核 cpu, 8GB 内存，SSD 磁盘，高速低延迟网络，独立宿主机部署，具体硬件的配置信息。在阿里巴巴，由于有超大规模的容器集群，因此我们运行 etcd 的硬件也较强

软件优化

etcd 是一款开源的软件，集合了全世界优秀软件开发者的智慧。最近的一年在软件上有很多贡献者更新了很多性能优化，这里分别从几个方面介绍几个优化，最后介绍一个由阿里巴巴贡献的 etcd 存储优化。

1. 内存索引层

由于索引层大量使用锁机制同步对性能影响较大，通过优化锁使用，提升了读写性能
具体参考：github pr；

2. lease 规模化使用

lease 是 etcd 支持 key 使用 ttl 过期的机制。在之前的版本中 scalability 较差，当有大量 lease 时性能下降的较为严重，通过优化 lease revoke 和过期失效的算法，解决了 lease 规模性的问题
具体参考：github pr；

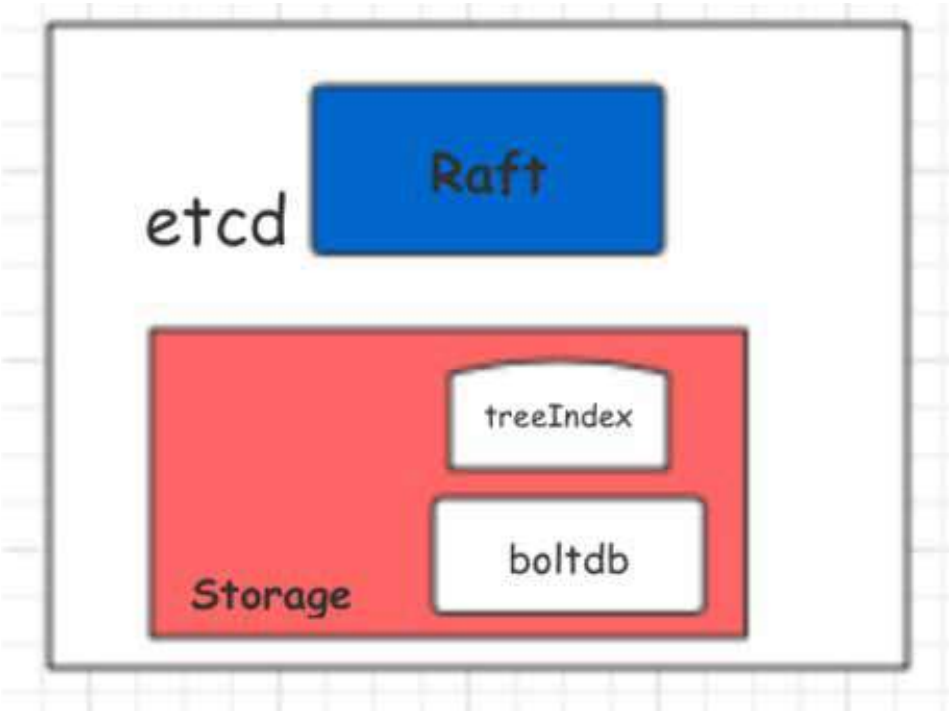
3. 后端 boltdb 使用优化

etcd 使用 boltdb 作为底层数据库存储 kv, 它的使用优化对整体性能影响很大。
通过调节不同的 batch size 和 interval， 使我们可以根据不同硬件和工作负载优化性能
具体参考：github pr。

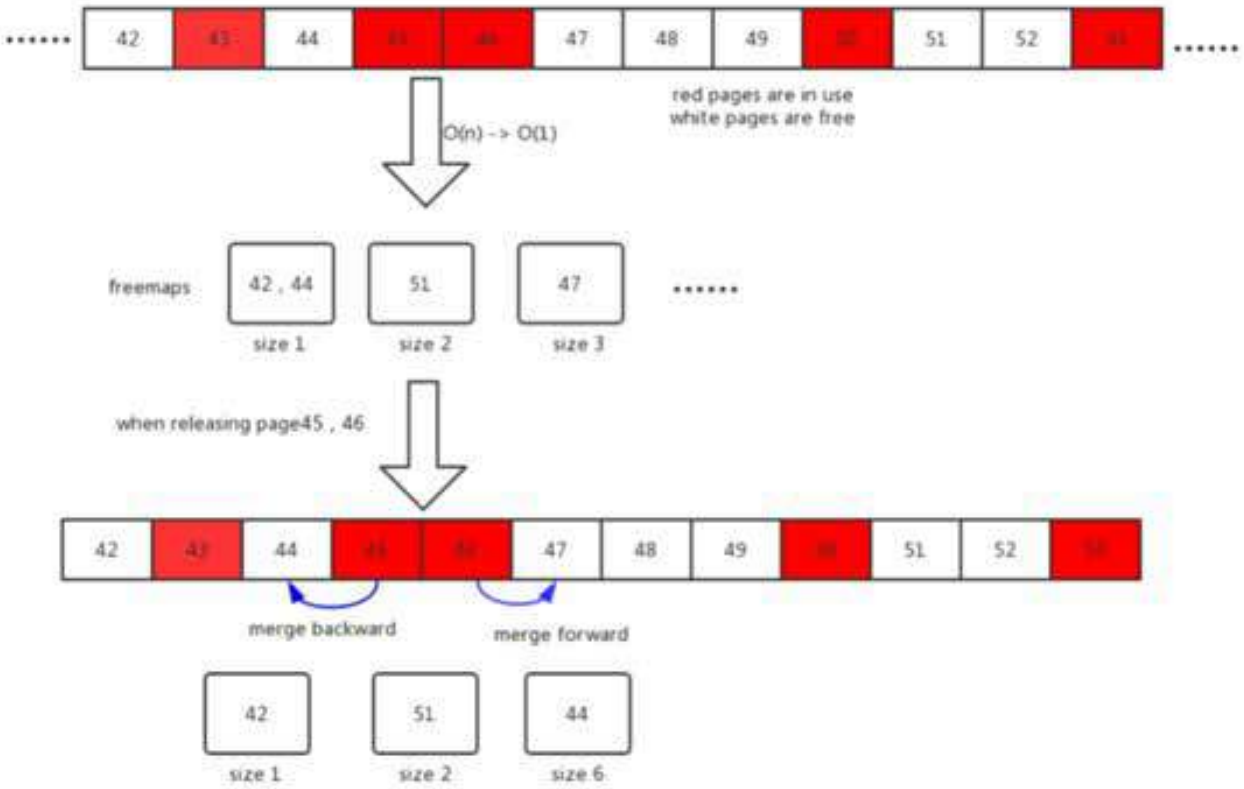
除此之外，新的完全并发读特性也优化了 boltdb tx 读写锁性能，大幅度的提升了读写性能
具体参考：github pr。

最后介绍一个由阿里巴巴自主研发并贡献开源社区的优化：基于 segregated hashmap 的 etcd 内部存储 freelist 分配回收算法。

下图是一个 etcd 节点的架构，etcd 使用 boltdb 持久化存储所有 kv，他的性能好坏对 etcd 性能起着非常重要的作用。



在阿里巴巴内部大规模使用 etcd 用于存储元数据，在使用中我们发现了 boltdb 的性能问题。这里给大家分享一下：



上图是 etcd 内部存储分配回收的核心算法。etcd 内部默认以 4kB 为一个页面大小存储数据。图中的数字表示页面 id, 红色表示该页面正在使用, 白色表示没有。当用户删除数据时 etcd 不会把存储空间还给系统，而是内部先留存起来维护一个页面池，以提升再次使用的性能，这个页面池专业术语叫 freelist。当 etcd 需要存储新数据时，普通 etcd 会线性扫描内部 freelist，时间复杂度 o(n)，当数据量超大或是内部碎片严重的情况下，性能会急剧下降。

因此我们重新设计并实现了基于 segregated hashmap 的 etcd 内部存储 freelist 分配回收新算法，该优化算法将内部存储分配算法时间复杂度从 o(n) 降为 o(1), 回收从 o(nlgn) 也降为 o(1), 使 etcd 性能有了质的飞跃，极大地提高了 etcd 存储数据的能力，使得 etcd 存储容量从推荐的 2GB 提升到 100GB，提升 50 倍，读写性能提升 24 倍。CNCF 官方博客收录了此次更新，感兴趣的读者可以读一下。
https://www.cncf.io/blog/2019/05/09/performance-optimization-of-etcd-in-web-scale-data-scenario/

客户端优化

性能优化除了服务端要做的事情外，还需要客户端的帮助。保持客户端使用最佳实践将保证 etcd 集群稳定高效地运行，这里我们分享 3 个最佳实践：

1. put 数据时避免大的 value, 大的 value 会严重影响 etcd 性能，例如：需要注意 Kubernetes 下 crd 的使用；
2. 避免创建频繁变化的 key/value, 例如：Kubernetes 下 node 数据上传更新；
3. 避免创建大量 lease 对象，尽量选择复用过期时间接近的 lease, 例如 Kubernetes 下 event 数据的管理。

让 etcd 管理更高效

作为基于 raft 协议的分布式键值数据库，etcd 是一个有状态的应用。管理 etcd 集群状态，运维 etcd 节点，冷热备份，故障恢复等过程均有一定复杂性，且需要具备 etcd 内核相关的专业知识，想高效地运维 etcd 有不小的挑战。

目前在业界里已经有一些 etcd 运维的工具，例如开源的 etcd-operator 等，但是这些工具往往比较零散，功能通用性不强，集成度比较差，学习这些工具的使用也需要一定的时间，关键是这些工具不是很稳定，存在稳定性风险等。

面对这些种种的问题，我们根据阿里巴巴内部场景，基于开源 etcd-operator 进行了一系列修改和加强，开发了 etcd 运维管理平台 Alpha。利用它，运维人员可以高效地运维管理 etcd，之前要前后操作多个工具完成的任务，现在只要操作它就可以完成，一个人就可以管理成百上千的 etcd 集群。下图展示了 Alpha 的基础功能。

如上图所示，Alpha 分为 etcd 生命周期管理和数据管理两大部分。

其中生命周期管理功能依托于 operator 中声明式的 CustomResource 定义，将 etcd 的集群创建、销毁的过程流程化、透明化，用户不再需要为每个 etcd 成员单独制定繁琐的配置，仅需要指定成员数量、成员版本、性能参数配置等几个简单字段。除此之外，我们还提供了 etcd 版本升级、故障节点替换、集群实例启停等功能，将 etcd 常用的运维操作自动化，同时也在一定程度上保证了 etcd 变更的稳定性。

其次，数据作为 etcd 的核心内容，我们也开发了一系列功能进行重点保障。在备份上，数据管理工具支持定期冷备及实时热备，且保持本地盘和云上 OSS 两类备份，同时也支持从备份上快速恢复出一个新的 etcd 集群。此外，数据管理工具支持对 etcd 进行扫描分析，发现当前集群的热点数据键值数和存储量，弥补了业界无法提供数据管理的空白，同时该拓展也是 etcd 支持多租户的基础。最后，数据管理工具还支持对 etcd 进行垃圾数据清理、跨集群数据腾挪传输等功能。

这些丰富的功能为上层 Kubernetes 集群的管理提供了很多灵活的帮助，例如用户 A 原来在某云厂商或自建 Kubernetes 集群，我们可以通过迁移 etcd 内部的账本数据的功能，将用户的核心数据搬移至另外一个集群，方便地实现用户的k8s集群跨云迁移。

利用 Alpha，我们可以做到透明化、自动化、白屏化，减少人肉黑屏操作，让 etcd 运维管理更高效。

让 etcd 变得更稳

让 etcd 变得更稳这节主要介绍一些 etcd 稳定建设的技巧，大家知道 etcd 是容器云平台的底层依赖核心，他的服务质量、稳定程度决定了整个容器云的稳定程度，它的重要性无需赘述。这里先介绍一下 etcd 常见的问题和风险分析如下图，主要分如下三个方面：

- etcd 自身: 例如 OOM、代码 bug、panic 等；
- 宿主机环境: 例如宿主机故障，网络故障，同一台宿主机其他进程干扰；
- 客户端: 例如客户端 bug、运维误操作、客户端滥用 ddos 等

针对这些风险点，我们从如下几个方面入手：

1. 建立完善的监控告警机制，覆盖客户端输入，etcd 自身以及宿主机环境状态；
2. 客户操作审计，高危操作如删除数据做风控限流；
3. 数据治理，分析客户端滥用 引导最佳实践；
4. 定期数据冷备，通过热备实现异地多活，保证数据安全；
5. 常态化故障演练，做好故障恢复预案。

总结展望：让 etcd 变得更智能

本文已分别从性能、稳定性、生态工具三个部分享了 etcd 变得更强、更快、更高效的技巧。在未来我们还将为让 etcd 变得更智能努力。如何让 etcd 变得更智能是一个比较高级的话题，这里简单做一下展望。更智能意思是指可以使 etcd 的管理更加地聪明，更少的人为干预，例如遇到一些故障，系统可以自行修复等。

01

Service Mesh 带来的变化和发展机遇

李云，花名至简，阿里云中间件技术部高级技术专家

在即将过去的 2019 年，Service Mesh 开源产品的成熟度虽在全球范围内没有发生质的变化，但在国内仍出现了一些值得特别关注的事件。比如，蚂蚁金服在 双11 大规模落地了 SOFAMosn；阿里巴巴在 双11 的部分电商核心应用上落地了完整的 Service Mesh 解决方案，借助 双11 的严苛业务场景完成了规模化落地前的初步技术验证。随着 Service Mesh 在规模化落地的探索案例的增多，相信对于该技术所带来的变革和发展机遇的理解也将更为深刻。本文结合作者在阿里巴巴落地实践Service Mesh 的过程中的那些观察和思考，与同行分享自己的一点看法。

Service Mesh 是新瓶装旧酒吗?

新技术出现之时所主张的价值一定会引发相应的探讨，Service Mesh 也不例外。以往，怀疑 Service Mesh 价值的观点主要有两大类。第一类是应用的数量并没有达到一定的规模，在 Service Mesh 增加运维和部署复杂度的情形下，认为所带来的成本和复杂度高于所获得的收益。根本上，这一类并非真正怀疑 Service Mesh 的价值，而是主张在 Service Mesh 还没有完全成熟和普及的情形下在未来合适的时机再考虑采纳。当然，我在与外部客户交流时也碰到一些特例，他们即便在应用数很少的情形下，仍希望通过 Service Mesh 去解决非 Java 编程语言（比方说Go）的分布式链路追踪等服务治理问题，虽说这些能力在 Java 领域有相对成熟的解决方案，但在非 Java 领域确实偏少，所以很自然地想到了采用 Service Mesh。

第二类怀疑 Service Mesh 价值的，是应用的数量具有相当的规模且对分布式应用的规模问题也有很好的认知，但由于在发展的过程中已经积累了与 Service Mesh 能力相当的那些（非体系化的）技术，造成初识 Service Mesh 时有“老酒换新瓶”的感觉而不认可其价值。阿里巴巴过去也曾属于这一阵营。

阿里巴巴在分布式应用的开发和治理方面的整体解决方案的探索有超过十年的历程，且探索过程持续地通过 双11 这样的严苛场景做检验和孵化，采用单一的 Java 语言打造了一整套的技术。即便如此，应对分布式应用的规模问题依然不轻松，体现于因为缺乏顶层设计而面临体系性不足，加之对技术产品自身的用户体验缺乏重视，最终导致运维成本和技术门槛都偏高。在面临这些阵痛之际，云原生的概念逐渐清晰地浮出了水面。

云原生在主张技术产品在最为严苛的场景下仍能提供一定质量的服务而体现良好弹性的同时，也强调技术产品本身应当具有良好的易用性，以及为将来企业需要多云和混合云的IT基础设施提供支撑（即，帮助实现分布式应用的可移植性）。云原生的概念不仅很好地契合了阿里巴巴集团在技术发展上亟待解决的阵痛，也迎合了阿里巴巴将云计算作为集团战略、让云计算普惠社会的初衷。在这一背景下，阿里巴巴做出了全面云原生化的决定，Service Mesh 作为云原生概念中的关键技术之一，当然也包含其中。

Service Mesh 给阿里巴巴带来的价值

Service Mesh 所带来的第一个变化体现于，服务治理手段从过去的框架思维向平台思维转变。这种转变并非后者否定前者，而是前后者相结合去更好地发挥各自的优势。两种思维的最大区别在于，平台思维不仅能实现应用与技术基础设施更好的解耦，也能通过平台的聚集效应让体系化的顶层设计有生发之地。

框架思维向平台思维转变在执行上集中体现于“轻量化”和“下沉”两个动作。轻量化是指将那些易变的功能从框架的 SDK 中移出，结果是使用了 SDK 的应用变得更轻，免除了因易变功能持续升级所带来的打扰而带来的低效，也因为彻底让应用的开发者无需关心这些功能而让他们能更好地聚焦于业务逻辑本身。从框架中移出的功能放到了 Service Mesh 的 Sidecar 中而实现了功能下沉。Service Mesh 作为平台性技术将由云厂商去运维和提供相应的产品，通过开源所构建的全球事实标准一旦被所有云厂商采纳并实现产品化输出，那时应用的可移植性问题就能水到渠成地解决。

功能下沉在阿里巴巴落地 Service Mesh 的过程中也看到了相应的价值。阿里巴巴的电商核心应用基本上都是用 Java 构建的，在 mesh 化之前，RPC 的服务发现与路由是在 SDK 中完成的，为了保证双11 这样的流量洪峰场景下的消费者用户体验，会通过预案对服务地址的变更推送做降级，避免因为频繁推送而造成应用进程出现 Full GC。Mesh 化之后，SDK 的那些功能被放到了 Sidecar（开发语言是 C++）这一独立进程中，这使得 Java 应用进程完全不会出现类似场景下的 Full GC 问题。

软件设计的质量主要体现在“概念”和“关系”两个词上。同样功能的一个系统，不同的概念塑造与切分将产生完全不同的设计成果，甚至影响到最终软件产品的工程质量与效率。当概念确定后，关系也随之确立，而关系的质量水平体现在解耦的程度上。Service Mesh 使得应用与技术基础设施之间的关系变得更松且稳定，通过流量无损的热升级方案，使得应用与技术基础设施的演进变得独立，从而加速各自的演进效率。软件不成熟、不完善并不可怕，可怕的是演进起来太慢、包袱太重。

阿里巴巴在落地 Service Mesh 的过程中，体会到了松耦合所带来的巨大工程价值。当应用被 mesh 化后，接下来的技术基础设施的升级对之就透明了，之前因为升级工作所需的人力配合问题可以通过技术产品化的手段完全释放。另外，以往应用进程中包含了业务逻辑和基础技术的功能，不容易讲清楚各自对计算资源的消耗，Service Mesh 通过独立进程的方式让这一问题得以更好地隔离而实现量化，有了量化结果才能更好地对技术做优化。

Service Mesh 所带来的第二个变化在于，技术平台的建设从面向单一编程语言向面向多编程语言转变。对于初创或小规模企业来说，业务应用的开发采用单一的编程语言具有明显优势，体现于因为个体掌握的技术栈相同而能带来良好的协作效率，但当企业的发展进入了多元化、跨领域、规模更大的更高阶段时，多编程语言的诉求就随之产生，对于阿里巴巴这样的云厂商来说更是如此（所提供的云产品不可能过度约束客户所使用的编程语言）。多编程语言诉求的背后是每种编程语言都有自己的优势和适用范围，需要发挥各自的优势去加速探索与创新。

从技术层面，这一转变意味着二点：

- 第一，技术平台的能力需要尽可能地服务化，避免因为服务化不彻底而需要引入 SDK，进而带来多编程语言问题（即因为没有相应编程语言的 SDK 而无法使用该编程语言）
- 第二，在无法规避 SDK 的情形下，让 SDK 变得足够的轻且功能稳定，降低平台化和多编程语言化的工程成本。支持多编程语言 SDK 最好的手段是采用 IDL

从组织层面，这一转变意味着平台技术团队的人员技能需要多编程语言化。一个只有单一编程语言的团队是很难做好面向多编程语言的技术平台的，不只是因为视角单一，还因为无法“吃自己的狗食”而对多编程语言问题有切肤之痛。

Service Mesh 带来的发展机遇

在这两个变化之下，我们来聊一聊 Service Mesh 所带来的发展机遇。

首先，Service Mesh 创造了一次以开发者为中心去打造面向未来的分布式应用开发平台的机会。在 Service Mesh 出现之前，各种分布式服务治理技术产品的发展缺乏强有力的抓手去横向拉通做体系化设计和完成能力复用，因而难免出现概念抽象不一致和重新造轮子的局面，最终每个技术产品有自己的一套概念和独立的运维控制台。当多个运维控制台交到开发者手上时，他们需要做大量的学习去理解每个运维控制台的概念和理解它们之间的关系，背后所带来的困难和低效是很容易被人忽视的。

本质上，Service Mesh 的出现是解决微服务软件架构之下那些藏在应用与应用之间的复杂度的。它的出现使得所有的分布式应用的治理问题被放到了一起去考虑。换句话说，因为 Service Mesh 的出现，我们有机会就分布式应用的治理做一次全局的设计，也有机会将各种技术产品整合到一起而避免重复建设的问题。

未来的分布式应用开发平台一定是基于 Service Mesh 这一基础技术的。为此，需要借这个契机从易用性的角度重新梳理应给开发者塑造的心智。易用性心智的确立，将使得开发者能在一个运维控制台上做最少的操作，通过为他们屏蔽背后的技术实现细节而减轻他们在使用时的脑力负担，以及降低操作失误而带来安全生产事故的可能性。理论上，没有 Service Mesh 之前这些工作也能做，但因为没有具体的横向技术做抓手而无法落地。

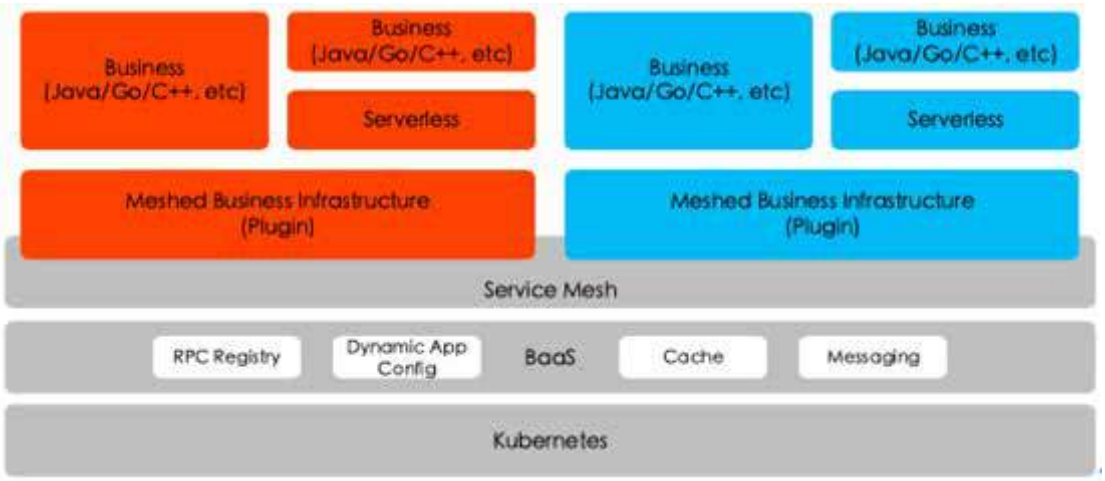
其次，Service Mesh 给其他技术产品创造了重新思考云原生时代的发展机会。有了 Service Mesh 后，以前好些独立的技术产品（比如，服务注册中心、消息系统、配置中心）将变成 BaaS（Back-end as a Service）服务，由 Service Mesh 的 Sidecar 负责与它们对接，应用对这些服务的访问通过 Sidecar 去完成，甚至有些 BaaS 服务被 Sidecar 终结而完全对应用无感。这些变化并非弱化了那些 BaaS 服务的重要性。相反，因为其重要性而需要与 Service Mesh 做更好的整合去为应用提供服务，与些同时探索做一定的能力增强。比方说，Service Mesh 所支持的应用版本发布的灰度功能（包括蓝绿发布、金丝雀发布、A/B 测试），并非每一个 BaaS 服务在 mesh 化后就能很好地支持这一功能，而是需要做相应的技术改造才行。请注意，这里主要讲的是应用的灰度能力，而非BaaS服务自身的灰度能力。当然，并不妨碍探索通过 Service Mesh 让 BaaS 服务自身的灰度工作变得简单且低风险。

未来好些技术产品的竞争优势将体现于它能否与 Service Mesh 形成无缝整合。无缝整合的核心驱动力源于用户对技术产品的易用性和应用可移植性的需要。基于这一认识，阿里巴巴正在将 RocketMQ/MetaQ 消息系统的客户端中的重逻辑剥离到 Envoy 这一 Sidecar 中（思路依然是“下沉”），同时基于 Service Mesh 所提供的能力做一定的技术改造，以便 RocketMQ/MetaQ 能很好地支撑应用的灰度发布。类似这样的思考与行动相信未来会在更多的技术产品上出现。

再次，Service Mesh 给技术基础设施如何与业务基础技术更好地协同提供了一次探索机会。每一种业务（比如电商）都会构建基于所在领域的基础技术，这类技术我们称之为业务基础技术。当阿里巴巴希望将某一业务的基础技术搬到外部去服务客户时，面临业务基础技术如何通过服务化去满足客户已选择的、与业务基础技术不同的编程语言的问题，否则会出现基于 Java 构建的业务基础技术很难与 Go 所编写的应用协同。

在 Service Mesh 致力于解决服务化问题的过程中，能否通过一定的技术手段，让业务基础技术的能力通过插件的形式“长”在 Service Mesh 之上是一个很值得探索的点。当业务基础技术以插件的形式存在时，业务基础技术无需以独立的进程存在而取得更好的性能，且这一机制也能被不同的业务复用。阿里巴巴的 Service Mesh 技术方案所采用的 Sidecar 开源软件 Envoy 正在积极地探索通过 Wasm 技术去实现流量处理的插件机制，将该机制进一步延延成为业务基础技术插件机制是值得探索的内容。

下图示例说明了业务基础技术的插件机制。图中两个彩色分别代表了不同的业务（比如一个代表电商，另一个代表物流），两个业务的基础技术并非开发了两个独立的应用（进程）然后做发布和运维管理，而是基于 Wasm 所支持的编程语言实现了业务技术插件，这一点可以理解为用多编程语言的方式解决业务服务化问题，而非强制要求采用与 Sidecar 一样的编程语言。插件通过 Service Mesh 的运维平台进行管理，包含安装、灰度、升级、监控等能力。



由于插件是“长”在 Service Mesh 之上的，插件化的过程就是业务技术服务化的过程。另外，Service Mesh 需要提供一种选择能力，让业务的应用开发者或运维者选择自己的机器上需要哪些插件（可以理解为插件市场）。另一个值得关注的点是，插件的运维和管理能力以及一定的质量保证手段由 Service Mesh 平台提供，但运维、管理和质量保证的责任由各插件的提供者承担。这种划分将有效地独绝所由插件的质量由 Service Mesh 平台去承担而带来的低效，分而治之仍是改善很多工程效率的良方。

最后，Service Mesh 给探索面向未来的异地多活、应用永远在线的整体技术解决方案打开了一扇大门。服务之间的互联互通，服务流量的控制、观测和安全加固是微服务软件架构下所要解决的关键问题，这些问题与规模化下的服务可用性和安全性紧密相关。未来，通过 Service Mesh 的流量控制能力能做很多改善应用发布和运维效率的文章，那时才能真正看到一个灵动、称手的云平台。

Service Mesh 的“三位一体”发展思路

阿里巴巴作为云计算技术的供应商，在探索 Service Mesh 技术的道路上，不只是考虑如何让云原生技术红利在阿里巴巴内部兑现，还同时思考着如何将技术红利带给更多的阿里云客户。基于此，阿里巴巴就 Service Mesh 的整体发展思路遵循“三位一体”，即阿里巴巴内部、阿里云上的相应商业产品和开源软件将采用同一套代码。

就我们与阿里云客户交流的经验来看，他们乐于尽最大可能采用非云厂商所特有的技术方案，以免被技术锁定而在未来的发展上出现掣肘。另外，他们只有采纳开源的事实标准软件才有可能达成企业的多云和混合云战略。基于客户的这一诉求，我们在Service Mesh的技术发展上特别重视参与开源事实标准的共建。在 Istio 和 Envoy 两个开源项目上，我们都会致力于将内部所做的那些优化反哺给开源社区。

未来，我们将在 Service Mesh 领域坚定而扎实地探索，也一定会将探索成果和思考持续地分享给大家。

02

阿里集团 双11 核心应用落地 Service Mesh 所克服的挑战

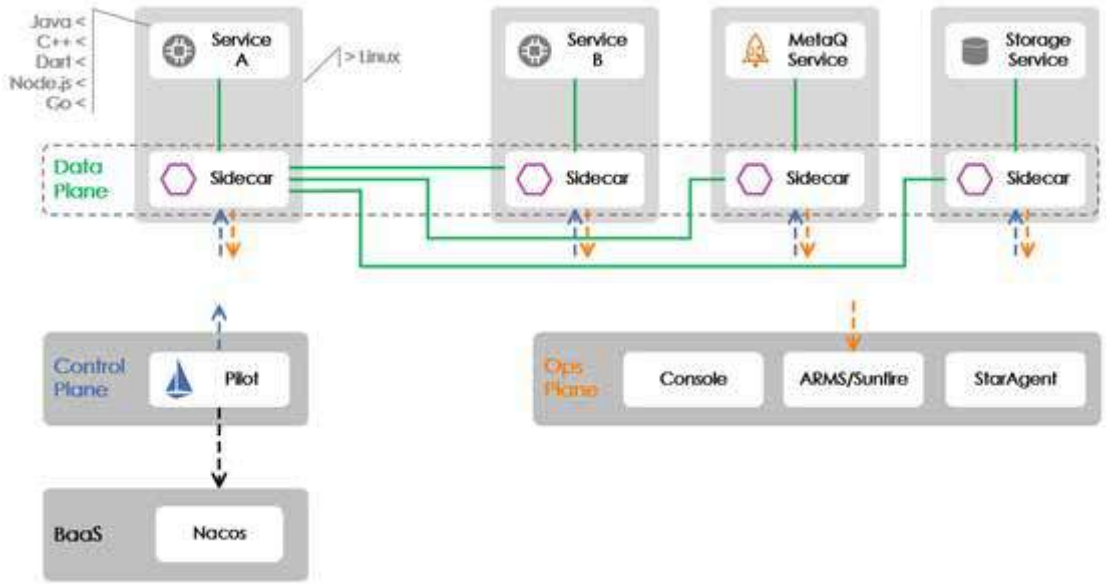
方克明，花名溪翁，阿里云中间件技术部技术专家

前言

2019 年，云原生已成为整个阿里巴巴经济体构建面向未来的技术基础设施。Service Mesh 作为云原生的关键技术之一，当仁不让地在云原生战略中占有一席之地。为了更快地催熟技术和兑现云原生的技术红利，阿里巴巴定下了在 双11 核心应用上落地 Service Mesh 的 目标。借助 双11 的严苛而复杂的场景，去检验 Service Mesh 技术方案的成熟度。这篇文章将与大家分享在完成这一目标的过程我们所面临和克服的挑战。

部署架构

在切入主题前，需要交代一下在 双11 核心应用上落地的部署架构，如下图所示。在这篇文章中，我们主要聚焦于 Service A 和 Service B 之间 RPC 协议的 mesh 化。



图中示例说明了 Service Mesh 所包含的三大平面，即数据平面（Data Plane）、控制平面（Control Plane）和运维平面（Operation Plane）。数据平面我们采用的是开源的 Envoy（上图中的 Sidecar，请读者注意这两个词在本文中可以互换使用），控制平面采用的是开源的 Istio（目前只使用了其中的 Pilot 组件），运维平面则完全自研。

与半年前落地时不同，这次 双11 核心应用上落地我们采用了 Pilot 集群化部署的模式，也即 Pilot 不再与 Envoy 一起部署到业务容器中，而是搭建了一个独立的集群。这一变化使得控制平面的部署方式演进到了 Service Mesh 应有的终态。

挑战

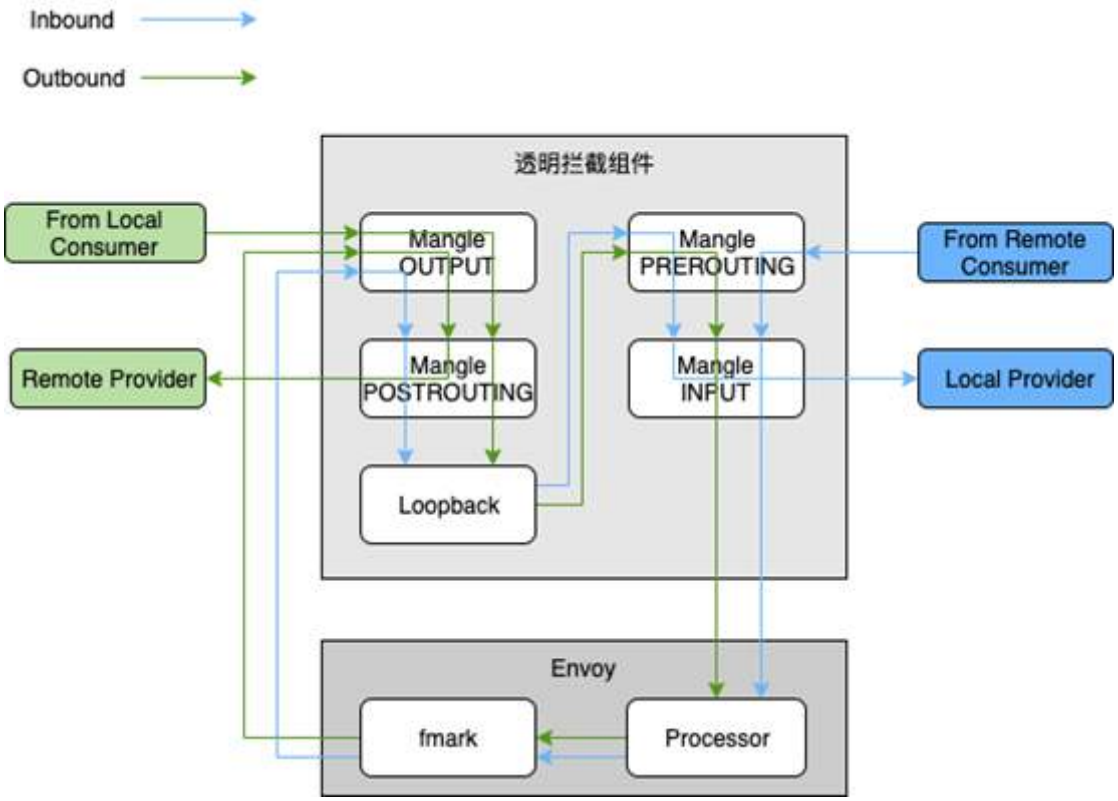
落地所选择的 双11 核心应用都是采用 Java 编程语言实现的，在落地的过程中我们面临了以下挑战。

挑战一：在 SDK 无法升级的情形下如何实现应用的 mesh 化

在决定要在 双11 的核心应用上落地 mesh 时，Java 应用依赖的 RPC SDK 版本已经定稿，为了 mesh 化完全没有时间去开发一个适用于 mesh 的 RPC SDK 并做升级。那时，摆在团队面前的技术问题是：如何在不升级 SDK 的情形下，实现 RPC 协议的 mesh 化?

熟悉 Istio 的读者想必清楚，Istio 是通过 iptables 的 NAT 表去做流量透明拦截的，通过流量透明拦截可在应用无感的情形下将流量劫持到 Envoy 中而实现 mesh 化。但很不幸，NAT 表所使用到的 nf_contrack 内核模块因为效率很低而在阿里巴巴的线上生产机器中被去除了，因此无法直接使用社区的方案。好在年初开始不久我们与阿里巴巴 OS 团队达成了合作共建，由他们负责承担 Service Mesh 所需的流量透明拦截和网络加速这两块的基础能力建设。经过两个团队的紧密合作，OS团队探索了通过基于 userid 和 mark 标识流量的透明拦截方案，基于 iptables 的 mangle 表实现了一个全新的透明拦截组件。

下图示例说明了存在透明拦截组件的情形下，RPC 服务调用的流量走向。其中，Inbound 流量是指调进来的流量（流量的接受者是 Provider 角色），而 Outbound 是指调出去的流量（流量的发出者是 Consumer 角色）。通常一个应用会同时承担两个角色，所以有 Inbound 和 Outbound 两股流量并存。



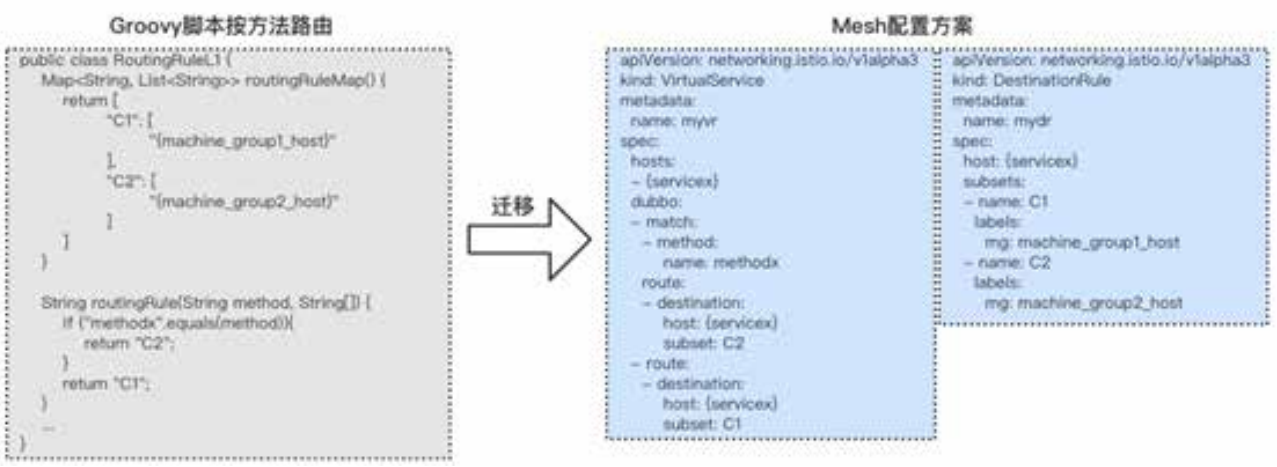
有了透明拦截组件之后，应用的 mesh 化完全能做到无感，这将极大地改善 mesh 落地的便利性。当然，由于 RPC 的 SDK 仍存在以前的服务发现和路由逻辑，而该流量被劫持到 Envoy 之后又会再做一次，这将导致 Outbound 的流量会因为存在两次服务发现和路由而增加 RT，这在后面的数据部分也将有所体现。显然，以终态落地 Service Mesh 时，需要去除 RPC SDK 中的服务发现与路由逻辑，将相应的 CPU 和内存开销给节约下来。

挑战二：短时间内支持电商业务复杂的服务治理功能

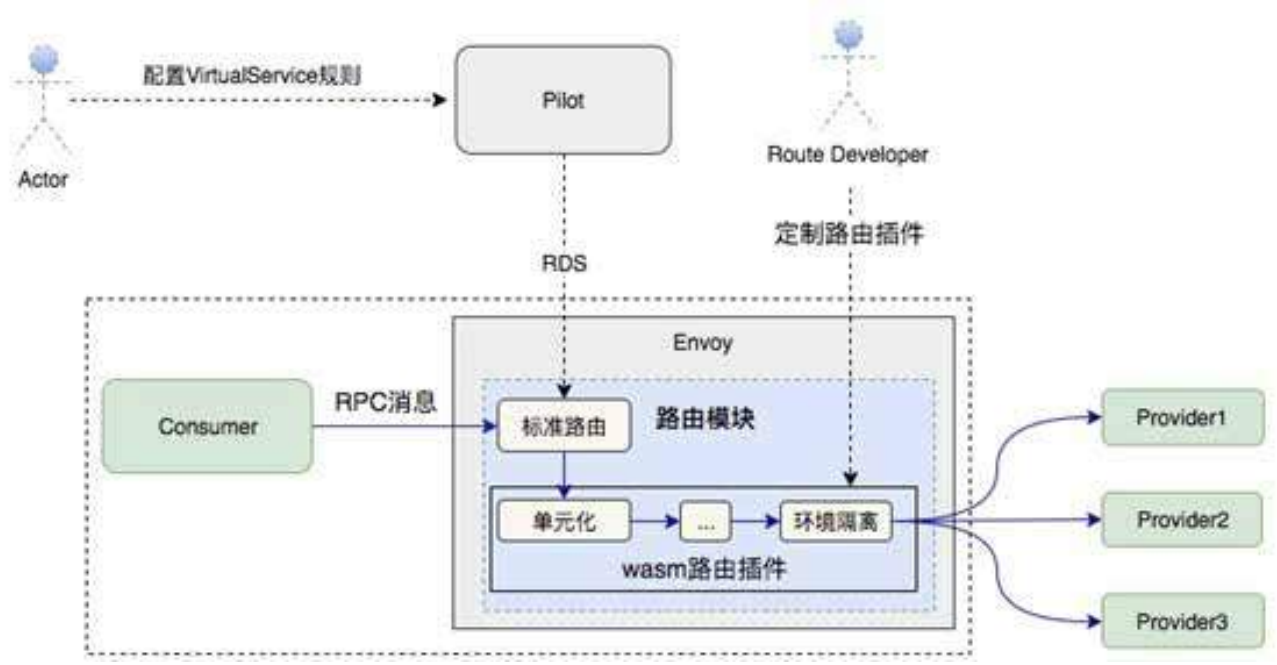
路由

在阿里巴巴电商业务场景下的路由特性丰富多样，除了要支持单元化、环境隔离等路由策略，还得根据 RPC 请求的方法名、调用参数、应用名等完成服务路由。阿里巴巴内部的 Java RPC 框架是通过嵌入 Groovy 脚本来支持这些路由策略的，业务方在运维控制台上配置 Groovy 路由模板，SDK 发起调用时会执行该脚本完成路由策略的运用。

未来的 Service Mesh 并不打算提供 Groovy 脚本那么灵活的路由策略定制方案，避免因为过于灵活而给 Service Mesh 自身的演进带去掣肘。因此，我们决定借 mesh 化的机会去除 Groovy 脚本。通过落地应用所使用 Groovy 脚本的场景分析，我们抽象出了一套符合云原生的解决方案：扩展 Istio 原生的 CRD 中的 VirtualService 和 DestinationRule，增加 RPC 协议所需的路由配置段去表达路由策略。

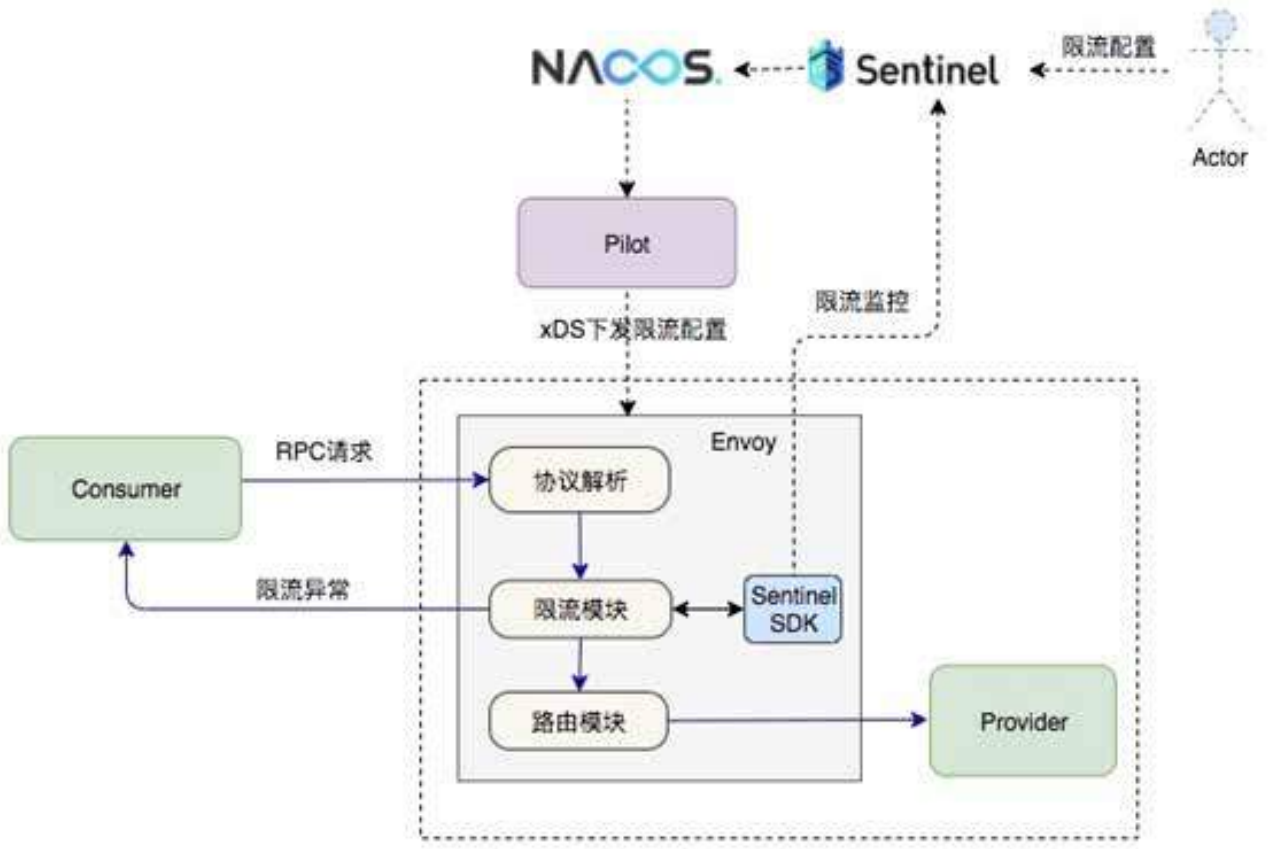


目前阿里巴巴环境下的单元化、环境隔离等策略都是在 Istio/Envoy 的标准路由模块内做了定制开发，不可避免地存在一些 hack 逻辑。未来计划在 Istio/Envoy 的标准路由策略之外，设计一套基于 Wasm 的路由插件方案，让那些简单的路由策略以插件的形式存在。如此一来，既减少对标准路由模块的侵入，也在一定程度上满足业务方对服务路由定制的需要。设想的架构如下图所示：



限流

出于性能考虑，阿里巴巴内部落地的 Service Mesh 方案并没有采用 Istio 中的 Mixer 组件，限流这块功能借助阿里巴巴内部广泛使用的 Sentinel 组件来实现，不仅可以与已经开源的 Sentinel 形成合力，还可以减少阿里巴巴内部用户的迁移成本（直接兼容业务的现有配置来限流）。为了方便 Mesh 集成，内部多个团队合作开发了 Sentinel 的 C++版本。整个限流的功能是通过 Envoy 的 Filter 机制来实现的，我们在 Dubbo 协议之上构建了相应的 Filter（Envoy 中的术语，代表处理请求的一个独立功能模块），每个请求都会经过 Sentinel Filter 做处理。限流所需的配置信息则是通过 Pilot 从 Nacos 中获取，并通过 xDS 协议下发到 Envoy 中。



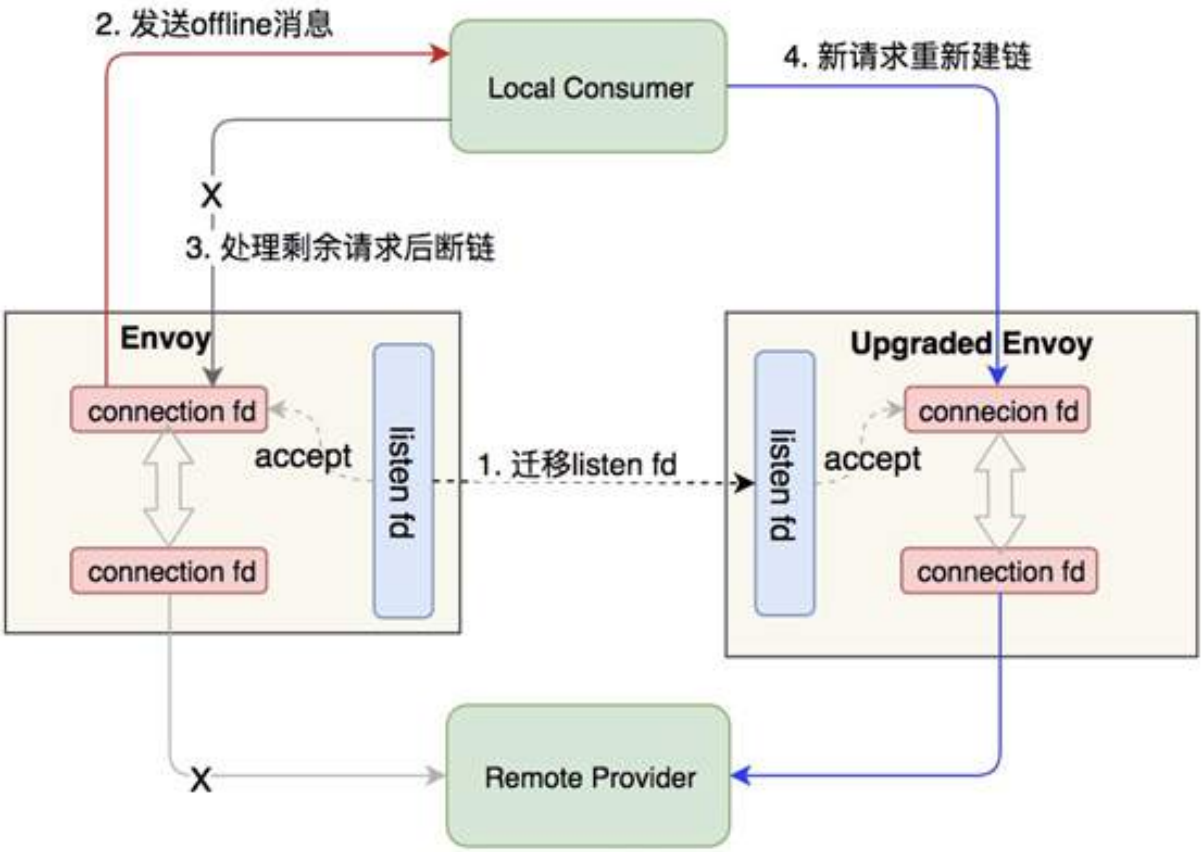
挑战三：Envoy 的资源开销过大

Envoy 诞生之初要解决的一个核心问题就是服务的可观测性，因此 Envoy 一开始就内置了大量的 stats（即统计信息）以便更好地对服务进行观测。Envoy 的 stats 粒度很细，甚至细到整个集群的IP 级别，在阿里巴巴环境下某些电商应用的 Consumer 和 Provider 服务加起来达到了几十万之多的 IP（每个 IP 在不同的服务下携带的元信息不同，所以不同的服务下的相同 IP 是各自独立的）。如此一来，Envoy 在这块的内存开销甚是巨大。为此，我们给 Envoy 增加了 stats 开关，用于关闭或打开 IP 级别的 stats，关闭 IP 级别的 stats 直接带来了内存节约 30% 成果。下一步我们将跟进社区的 stats symbol table 的方案来解决 stats 指标字符串重复的问题，那时的内存开销将进一步减少。

挑战四：解耦业务与基础设施，实现基础设施升级对业务无感

Service Mesh 落地的一项核心价值就是让基础设施与业务逻辑完全解耦，两者可以独立演进。为了实现这个核心价值，Sidecar 需要具备热升级能力，以便升级时不会造成业务流量中断，这对方案设计和技术实现的挑战还是蛮大的。

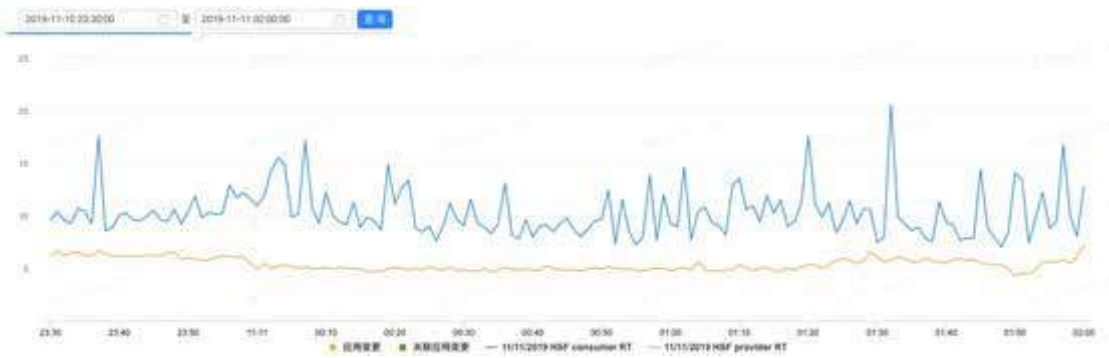
我们的热升级采用双进程方案，先拉起新的 Sidecar 容器，由它与旧的 Sidecar 进行运行时数据交接，在新的 Sidecar 准备发接管流量后，让旧的 Sidecar 等待一定时间后退出，最终实现业务流量无损。核心技术主要是运用了 Unix Domain Socket 和 RPC 的节点优雅下线功能。下图大致示例了关键过程。



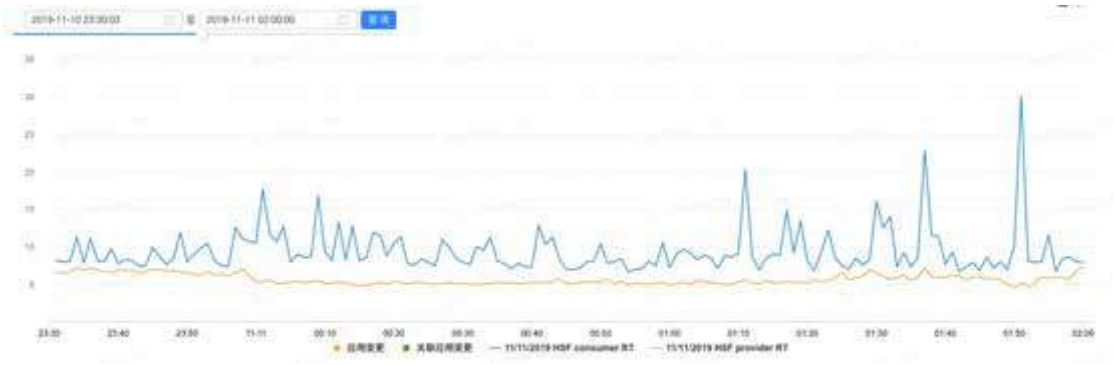
数据表现

公布性能数据一不小心就会引发争议和误解，因为性能数据的场景存在很多变量。比如，并发度、QPS、payload 大小等对最终的数据表现将产生关键影响。也正因如此，Envoy 官方从来没有提供过本文所列出的这些数据，背后的原因正是其作者 Matt Klein 担心引发误解。值得强调，在时间非常紧迫的情形下，我们所落地的 Service Mesh 并非处于最优状态，甚至不是最终方案（比如，Consumer 侧存在两次路由的问题）。我们之所以选择分享出来，是希望让更多的同行了解我们的进展和状态。

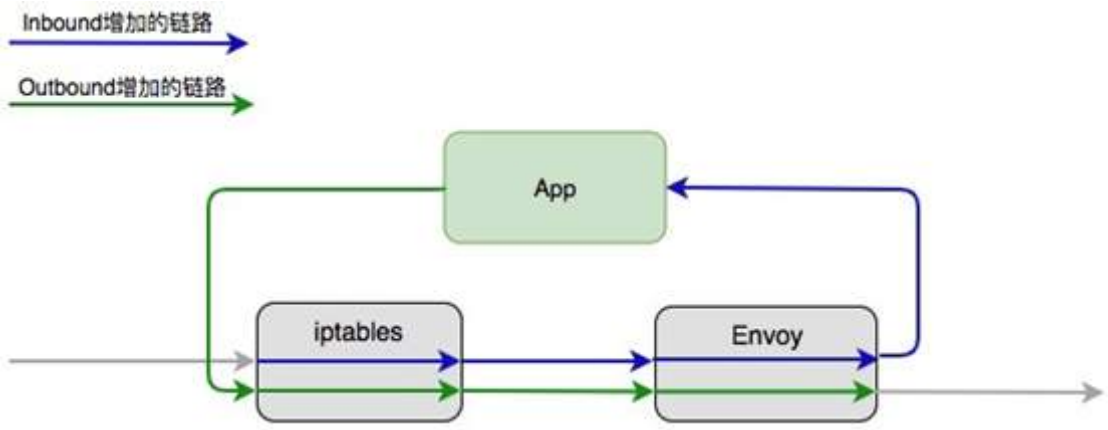
本文只列出了 双11 所上线核心应用中某一个的数据。从单机 RT 抽样的角度，部署了 Service Mesh 的某台机器，其 Provider 侧的 RT 均值是 5.6ms，Consumer 侧的是 10.36ms。该机器在 双11 零点附近的 RT 表现如下图所示：



没有部署 Service Mesh 的某台机器，Provider 侧的均值为 5.34ms，Consumer 侧的则是 9.31ms。下图示例了该机器在 双11 零点附件的 RT 表现。



相比之下，Provider 侧的 RT 在 mesh 化前后增加了 0.26ms，Consumer 侧则增加了 1.05 毫秒。注意，这个 RT 差是包含了业务应用到 Sidecar，以及 Sidecar 处理的所有时间在内的，下图示例说明了带来时延增加的链路。



整体上，该核心应用所有上线了 Service Mesh 的机器和没有上线 Service Mesh 的机器在某个时间段的整体均值数据做了对比。Provider 侧 Mesh 化后的 RT 增加了 0.52 毫秒，而 Consumer 侧增加了 1.63 毫秒。

在 CPU 和内存开销方面，Mesh 化之后，Envoy 所消耗的 CPU 在所有核心应用上都维持在 0.1 核左右，会随着 Pilot 推送数据而产生毛刺。未来需要借助 Pilot 和 Envoy 之间的增量推送去对毛刺做优化。内存的开销随着应用的服务和集群规模不同而存在巨大差异，目前看来 Envoy 在内存的使用上仍存在很大的优化空间。

从所有双11 上线的核心应用的数据表现来看，Service Mesh 的引入对于 RT 的影响和带来的 CPU 开销是基本一样的，而内存开销则因为依赖服务和集群规模的不同而有相当大的差异。

展望

在云原生的浪潮下，阿里巴巴借这拨技术浪潮致力于打造面向未来的技术基础设施。在发展的道路上将贯彻“借力开源，反哺开源”的发展思路，通过开源实现技术普惠，为未来的云原生技术在更大范围的普及做出自己的贡献。接下来，我们的整体技术着力点在于：

- 与 Istio 开源社区共同增强 Pilot 的数据推送能力。在阿里巴巴具备 双11 这种超大规模的应用场景下，我们对于Pilot　的数据推送能力有着极致的要求，相信在追求极致的过程中，能与开源社区一道加速全球事实标准的共建。从阿里巴巴内部，目前我们拉通了与 Nacos 团队的共建，将通过社区的 MCP 协议与 Nacos 对接，让阿里巴巴所开源的各种技术组件能体系化地协同工作。
- 以 Istio 和 Envoy 为一体，进一步优化两者的协议以及各自的管理数据结构，通过更加精炼、更加合理的数据结构去减少各自的内存开销。
- 着力解决大规模 Sidecar 的运维能力建设。让 Sidecar 的升级做到可灰度、可监控和可回滚。
- 兑现 Service Mesh 的价值，让业务与技术设施能以更高的效率彼此独立演进。

03

蚂蚁金服 双11 Service Mesh 超大规模落地揭秘

黄挺，花名鲁直，蚂蚁金服微服务以及云原生方向负责人，主导蚂蚁金服的云原生落地。
雷志远，花名碧远，蚂蚁金服 RPC 框架负责人

引言

Service Mesh 是蚂蚁金服下一代架构的核心，本主题主要分享在蚂蚁金服当前的体量下，我们如何做到在奔跑的火车上换轮子，将现有的 SOA 体系快速演进至 Service Mesh 架构。聚焦 RPC 层面的设计和改造方案，分享蚂蚁金服 双11 核心应用如何将现有的微服务体系平滑过渡到 Service Mesh 架构下并降低大促成本。

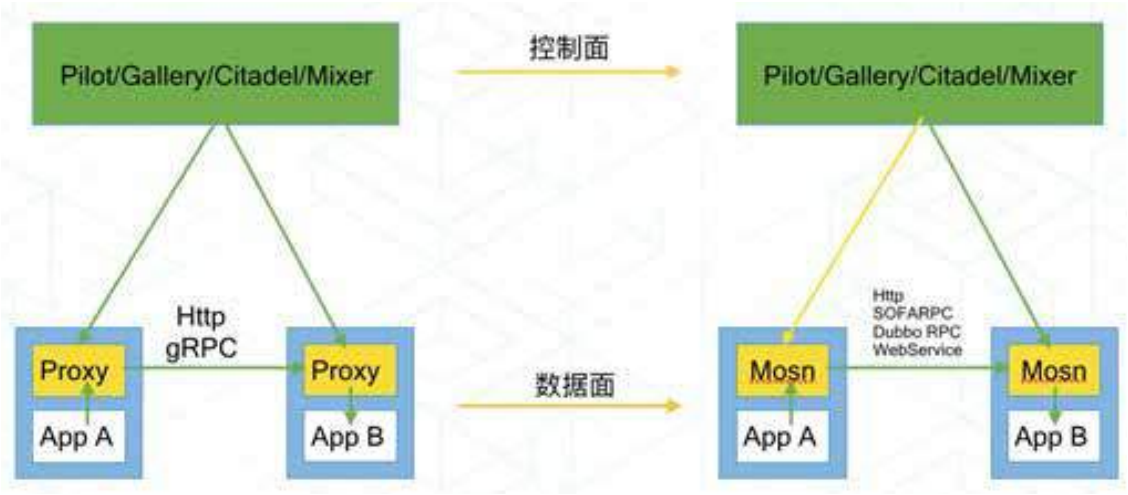
蚂蚁金服每年 双11 大促会面临非常大的流量挑战，在已有 LDC 微服务架构下已支撑起弹性扩容能力。本次分享主要分为 4 部分：

1. Service Mesh 简介；
2. 为什么要 Service Mesh；
3. 我们的方案是什么；
4. 分时调度案例；

Service Mesh 简介

在讲具体的蚂蚁金服落地之前，想先和大家对齐一下 Service Mesh 的概念，和蚂蚁金服对应的产品。

这张图大家可能不陌生，这是业界普遍认可的 Service Mesh 架构，那么对应到蚂蚁金服，蚂蚁金服的 Service Mesh 也分为控制面和数据面，分别叫做 SOFAMesh 和 SOFAMosn，其中 SOFAMesh 后面会以更加开放的姿态参与到 Istio 里面去。



今天我们讲的实践主要集中在 SOFAMosn 上，以下我的分享中提到的主要就是集中在数据面上的落地，这里面大家可以看到，我们有支持 HTTP/SOFARPC/Dubbo/WebService。

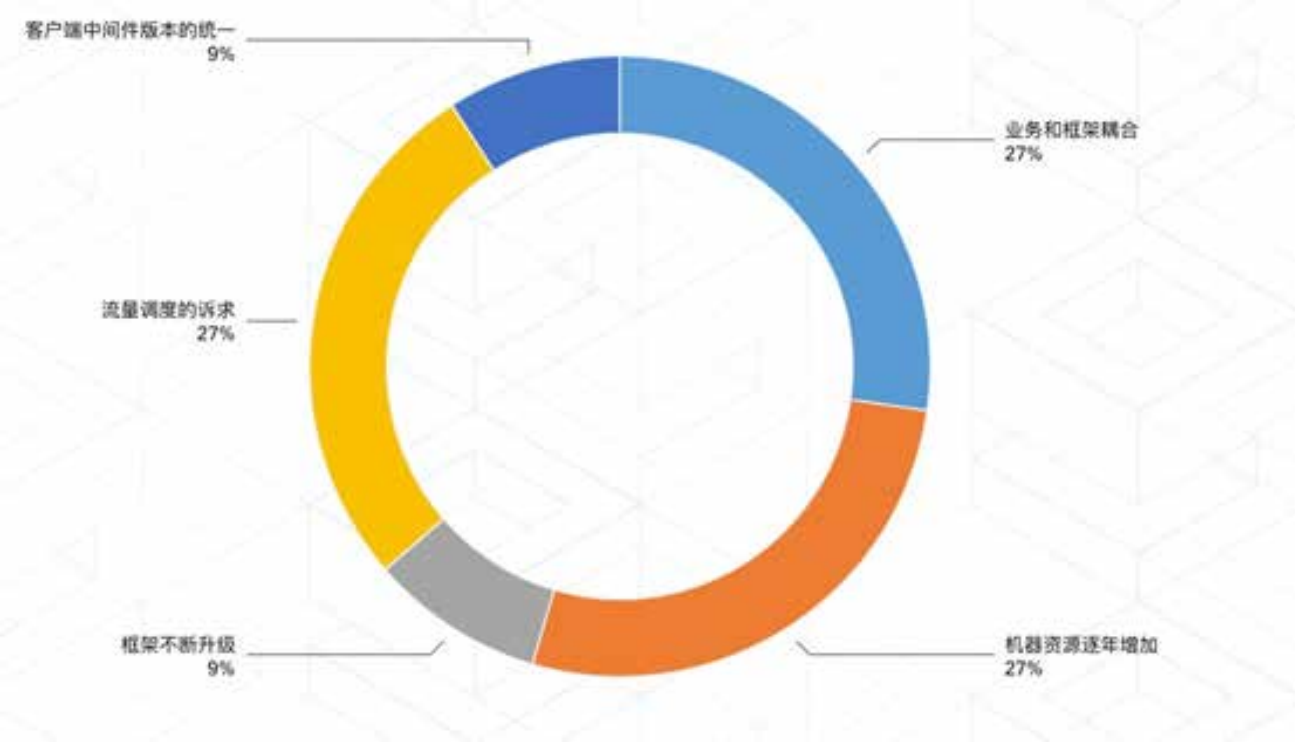
为什么我们要 Service Mesh

有了一个初步的了解之后，可能大家都会有这样一个疑问，你们为什么要 Service Mesh，我先给出结论：

因为我们要解决在 SOA 下面，没有解决但亟待解决的：基础架构和业务研发的耦合，以及未来无限的对业务透明的稳定性与高可用相关诉求。

那么接下来，我们一起先看看在没有 Service Mesh 之前的状况。

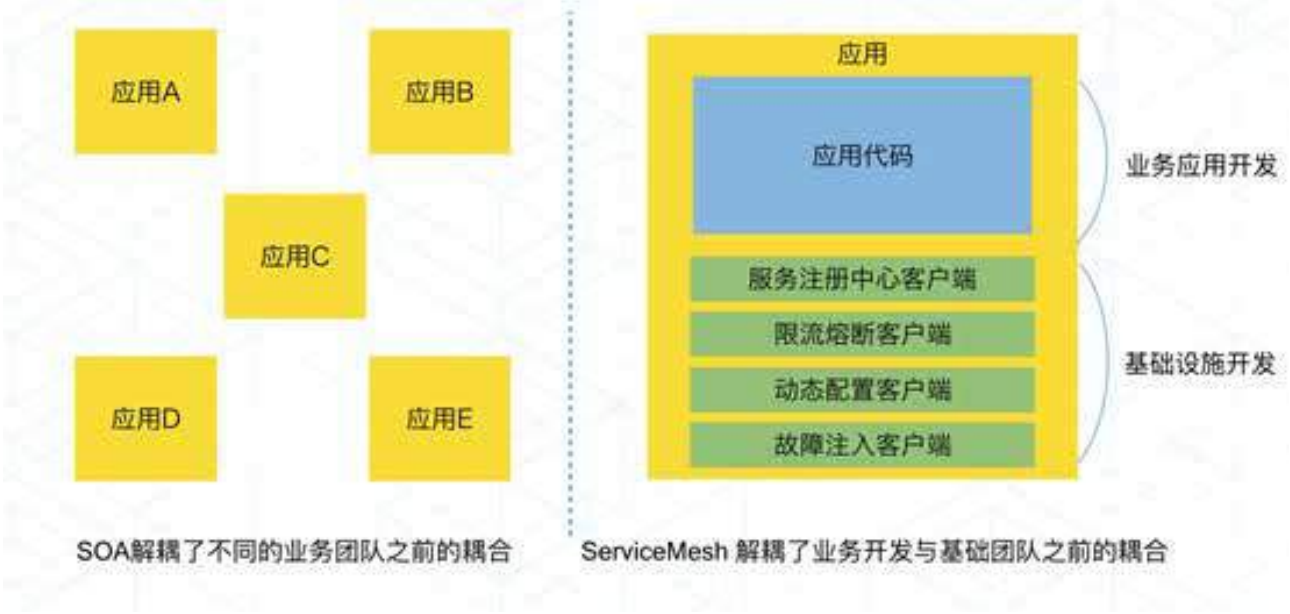
在没有 Service Mesh 之前，整个 SOFAStack 技术演进的过程中，框架和业务的结合相当紧密，对于一些 RPC 层面的需求，比如流量调度，流量镜像，灰度引流等，是需要在 RPC 层面进行升级开发支持，同时，需要业务方来升级对应的中间件版本，这给我们带来了一些困扰和挑战。如图所示：



1. 线上客户端框架版本不统一；
2. 业务和框架耦合，升级成本高，很多需求由于在客户端无法推动，需要在服务端做相应的功能，方案不够优雅；
3. 机器逐年增加，如果不增加机器，如何度过 双11；
4. 在基础框架准备完成后，对于新功能，不再升级给用户的 API 层是否可行；
5. 流量调拨，灰度引流，蓝绿发布，AB Test 等新的诉求；

这些困扰着我们，我们知道在 SOA 的架构下，负责每个服务的团队都可以独立地去负责一个或者多个服务，这些服务的升级维护也不需要其他的团队的接入，SOA 其实做到了团队之间可以按照接口的契约来接耦。但是长期以来，基础设施团队需要推动很多事情，都需要业务团队进行紧密的配合，帮忙升级 JAR 包，基础设施团队和业务团队在工作上的耦合非常严重，上面提到的各种问题，包括线上客户端版本的不一致，升级成本高等等，都是这个问题带来的后果。

而 Service Mesh 提供了一种可能性，能够将基础设施下沉，让基础设施团队和业务团队能够解耦，让基础设施和业务都可以更加快步地往前跑。

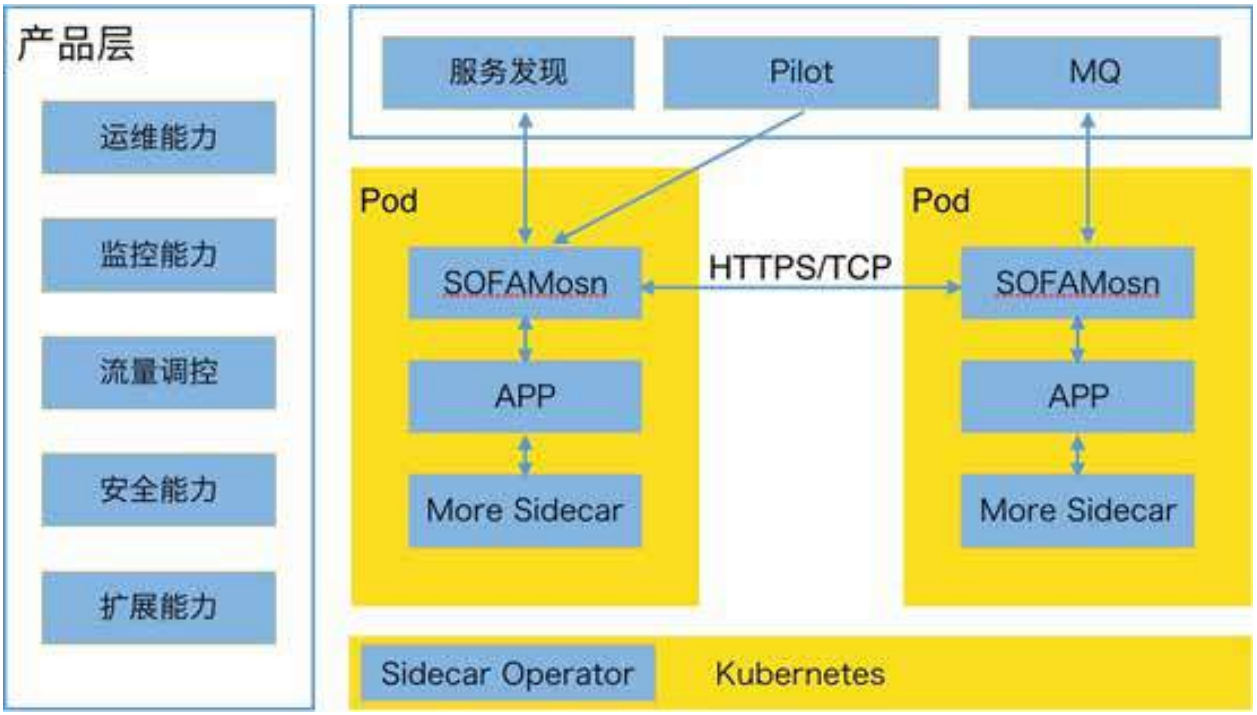


我们的方案

说了这么多，那我们怎么解决呢？

方案一：全部迁移到 Envoy？不现实，自有协议+历史负担。
方案二：透明劫持？性能问题，且表达能力有限，运维和可监控性，风险不太可控。
方案三：自研数据面，最终我们给出的答案是 SOFAMosn。

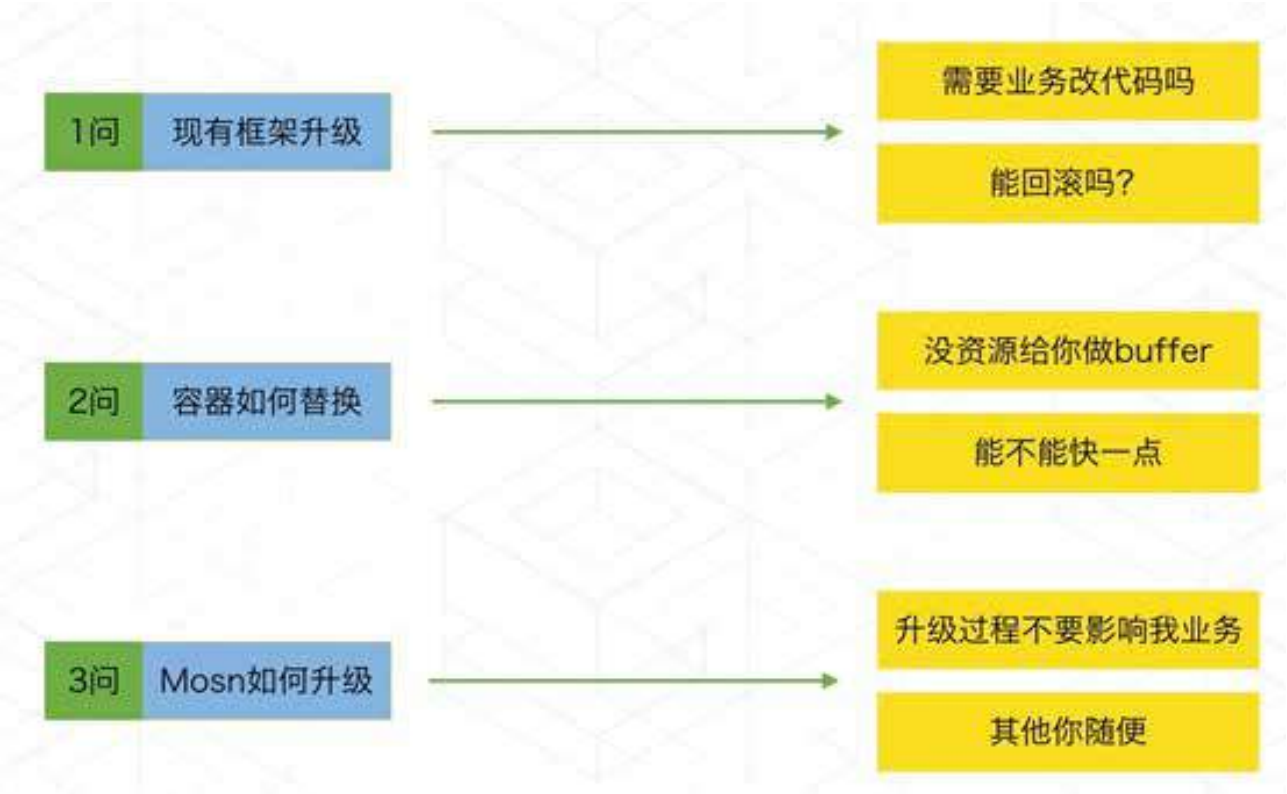
总体



我们的 SOFAMosn 支持了 Pilot ，自有服务发现 SOFARegistry，和自有的消息组件，还有一些 DB 的组件。在产品层，提供给开发者，不同的能力，包括运维，监控，安全等能力，这个是目前我们的一个线上的状态。

SOFARegistry 是蚂蚁金服开源的具有承载海量服务注册和订阅能力的、高可用的服务注册中心，在支付宝/蚂蚁金服的业务发展驱动下，近十年间已经演进至第五代。

看上去很美好，要走到这个状态，我们要回答三个问题。

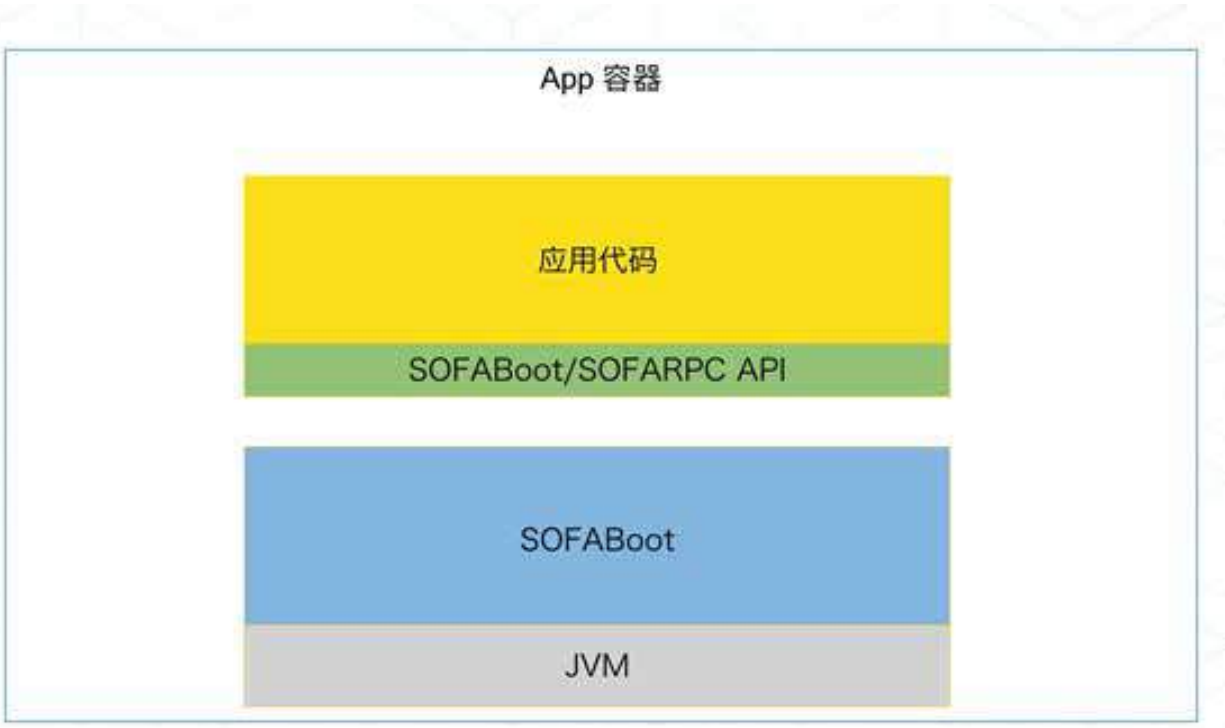


这三个问题后面，分别对应着业务的几大诉求，大家做过基础框架的应该比较有感触。

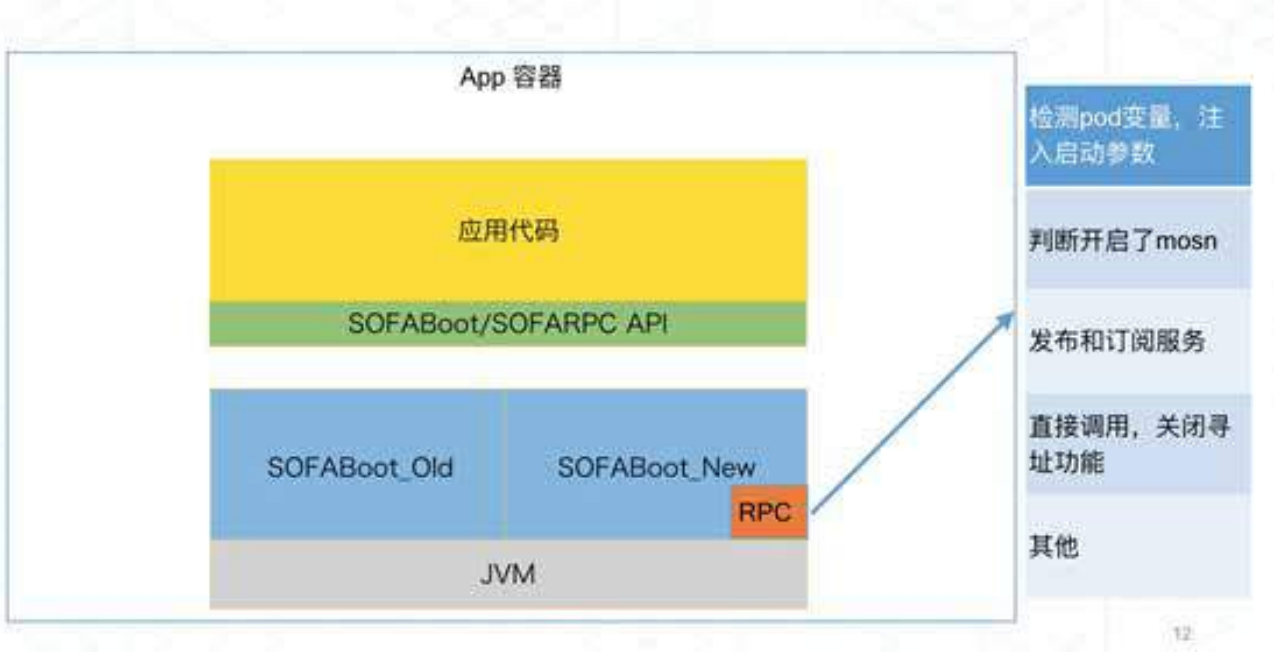
框架升级方案

准备开始升级之后，我们要分析目前我们的线上情况，而我们现在线上的情况，应用代码和框架有一定程度的解耦，用户面向的是一个 API，最终代码会被打包，在 SOFABoot 中运行起来。

SOFABoot 是蚂蚁金服开源的基于 SpringBoot 的研发框架，它在 SpringBoot 的基础上，提供了诸如 Readiness Check，类隔离，日志空间隔离等能力。在增强了 SpringBoot 的同时，SOFA-Boot 提供了让用户可以在 SpringBoot 中非常方便地使用 SOFA 中间件的能力。



那么，我们就可以在风险评估可控的情况下，直接升级底层的 SOFABoot，在这里，我们的 RPC 会检测一些信息，来确定当前 Pod 是否需要开启 SOFAMosn 的能力。然后我们完成如下的步骤。



这里，我们通过检测 PaaS 传递的容器标识，知道自己是否开启了 SOFAMosn，则将发布和订阅给 SOFAMosn，然后调用不再寻址，直接完成调用。

可以看到，我们通过批量的运维操作，直接修改了线上的 SOFABoot 版本，以此，来直接使得现有的应用具备了 SOFAMosn 的能力，有些同学可能会问，那你一直这么做不行吗？不行，因为这个操作时要配合流量关闭等操作来运行的，也不具备平滑升级的能力。而且直接和业务代码强相关。不适合长期操作。

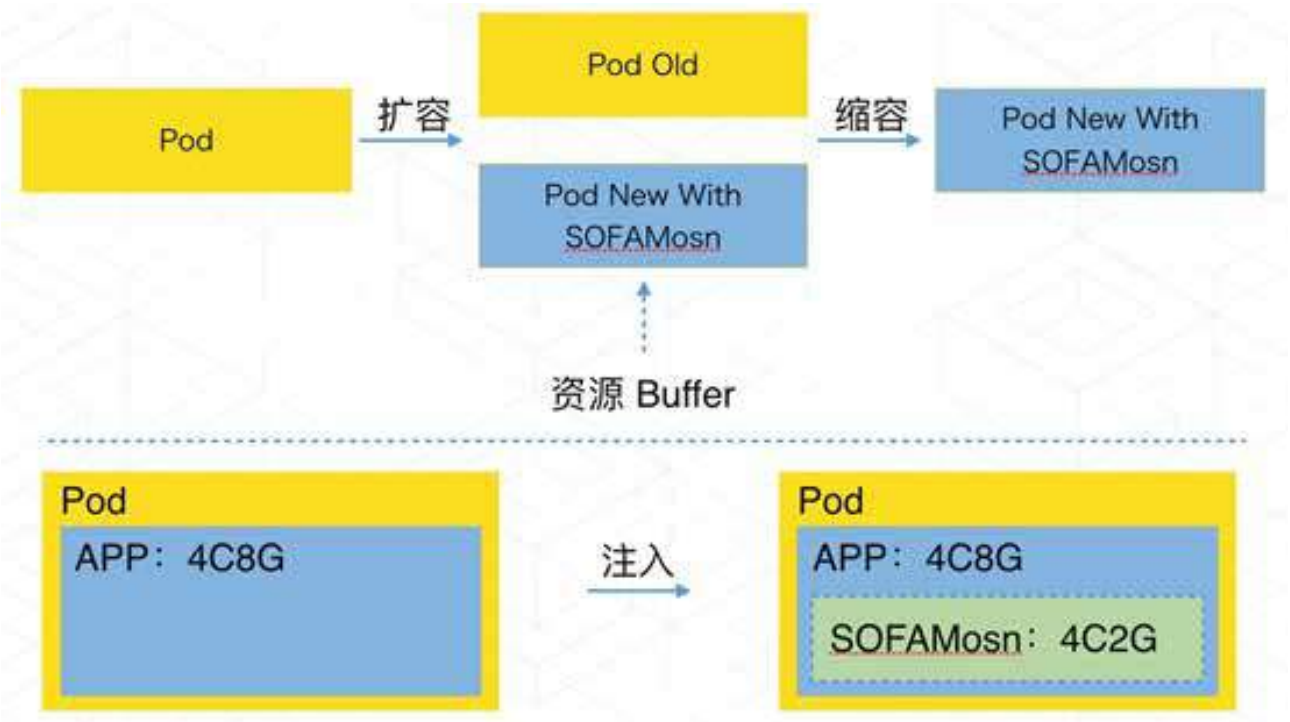
这里我们来详细回答一下，为什么不采用社区的流量劫持方案?

主要的原因是一方面 iptables 在规则配置较多时，性能下滑严重。另一个更为重要的方面是它的管控性和可观测性不好，出了问题比较难排查。而 Service Mesh 从初期就把蚂蚁金服现在线上的系统全量切换 Mesh 作为目标，并不是只是跑跑 demo，所以我们对性能和运维的要求是非常高的，毕竟，技术架构升级，如果是以业务有损或者资源利用率大幅度上升，这是无论如何都不能接受的。

容器替换方案

解决了刚刚提到的第一个难题，也只是解决了可以做，而并不能做得好，更没有做得快，面对线上数十万，带着流量的业务容器， 我们如何立刻开始实现这些容器的快速稳定接入?

这么大的量，按照传统的替换接入显然是很耗接入成本的事情，于是我们选择了原地接入，我们可以来看下两者的区别



在之前，我们做一些升级操作之类的，都是需要有一定的资源 Buffer，然后批量的进行操作，替换 Buffer 的不断移动，来完成升级的操作。这就要求 PaaS 层留有非常多的 Buffer，但是在 双11 的情况下，我们要求不增加机器，并且为了一个接入 SOFAMosn 的操作，反而需要更多的钱来买资源，这岂不是背离了我们的初衷。有人可能会问，不是还是增加了内存和 CPU 吗，这是提高了 CPU 利用率。以前业务的 CPU 利用率很低。并且这个是一个类似超卖的方案。看上去分配了。实际上基本没增加。

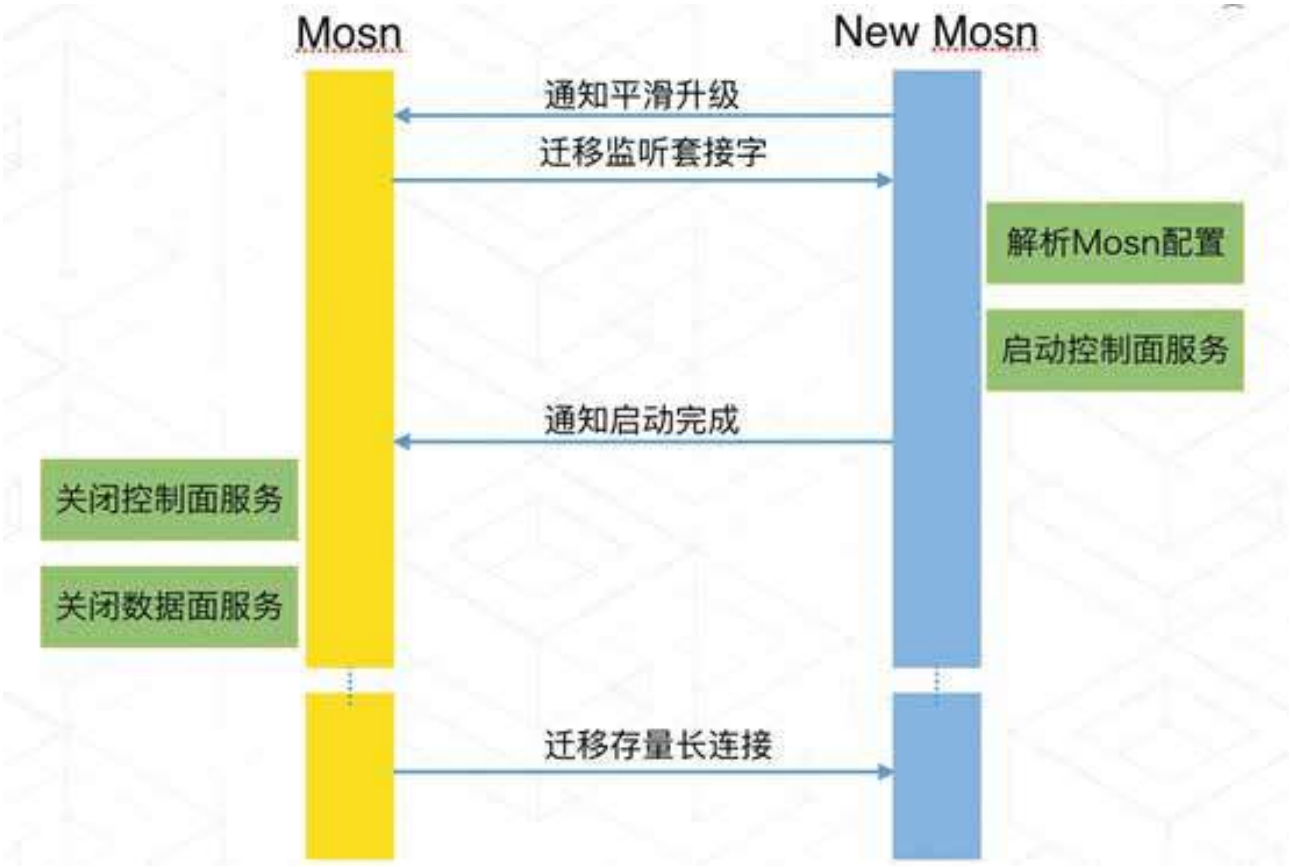
可以看到， 通过 Paas 层，我们的 Operator 操作，直接在现有容器中注入，并原地重启，在容器级别完成升级。升级完成后，这个 Pod 就具备了 SOFAMosn 的能力。

SOFAMosn 升级方案

在快速接入的问题完成后，我们要面临第二个问题，由于是大规模的容器，所以 SOFAMosn 在开发过程中，势必会存在一些问题，出现问题，如何升级，要知道，线上几十万容器要升级一个组件的难度是很大的，因此，在版本初期，我们就考虑到 SOFAMosn 升级的方案。

能想到的最简单的方法，就是销毁容器，然后用新的来重建，但是在容器数量很多的时候，这种运维成本是不可接受的。如果销毁容器重建的速度不够快，就可能会影响业务的容量，造成业务故障。因此，我们在 SOFAMosn 层面，和 PaaS 一起，开发了无损流量升级的方案。

在这个方案中。SOFAMosn 会感知自己的状态，新的 SOFAMosn 启动会通过共享卷的 Domain Socket 来检测是否已有老的 SOFAMosn 在运行，如果有，则通知原有 SOFAMosn 进行平滑升级操作。



具体来说，SOFAMosn 启动的时候查看同 Pod 是否有运行的 SOFAMosn (通过共享卷的domain socket)，如果存在，需要进入如下流程：

1. New Mosn 通知 Old Mosn，进入平滑升级流程
2. Old Mosn 把服务的 Listen Fd 传递给 New Mosn，New Mosn 接收 Fd 之后启动， 此时 Old 和 New Mosn 都正常提供服务。
3. 然后 New Mosn 通知 Old Mosn，关闭 Listen Fd，然后开始迁移存量的长链接。
4. Old Mosn 迁移完成， 销毁容器。

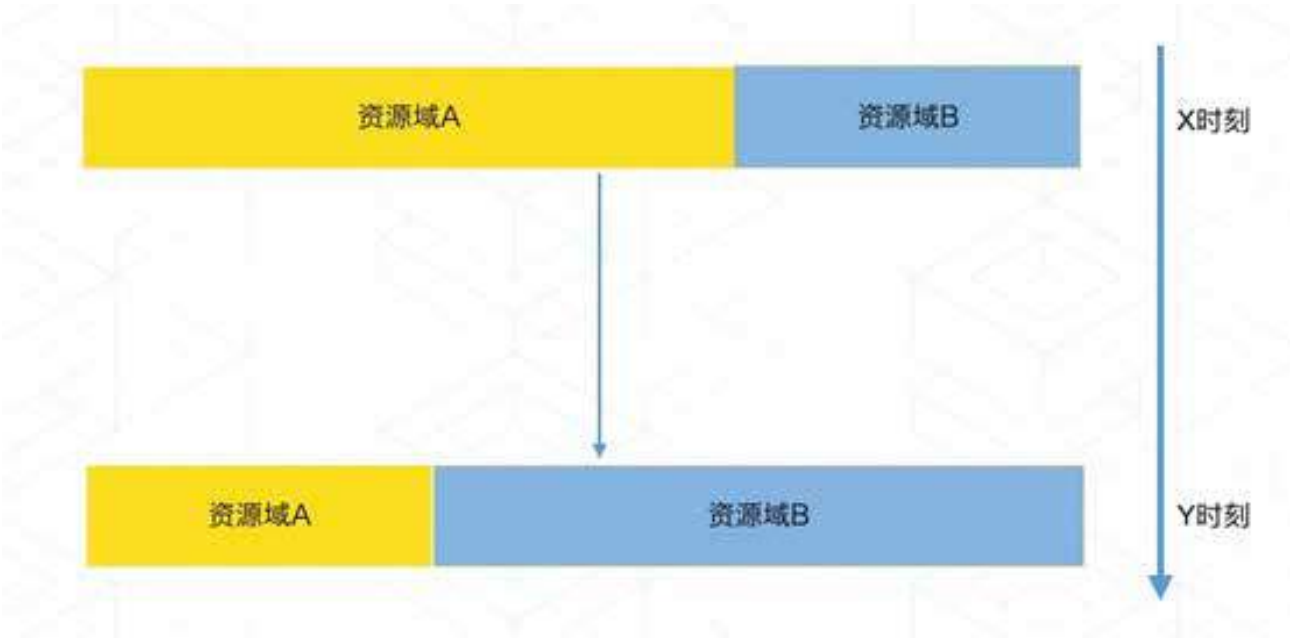
这样，我们就能做到，线上做任意的 SOFAMosn 版本升级，而不影响老的业务，这个过程中的技术细节，不做过多介绍，之后，SOFAStack 会有更详细的分享文章。

分时调度案例

技术的变革通常一定不是技术本身的诉求，一定是业务的诉求，是场景的诉求。没有人会为了升级而升级，为了革新而革新，通常，技术受业务驱动，也反过来驱动业务。

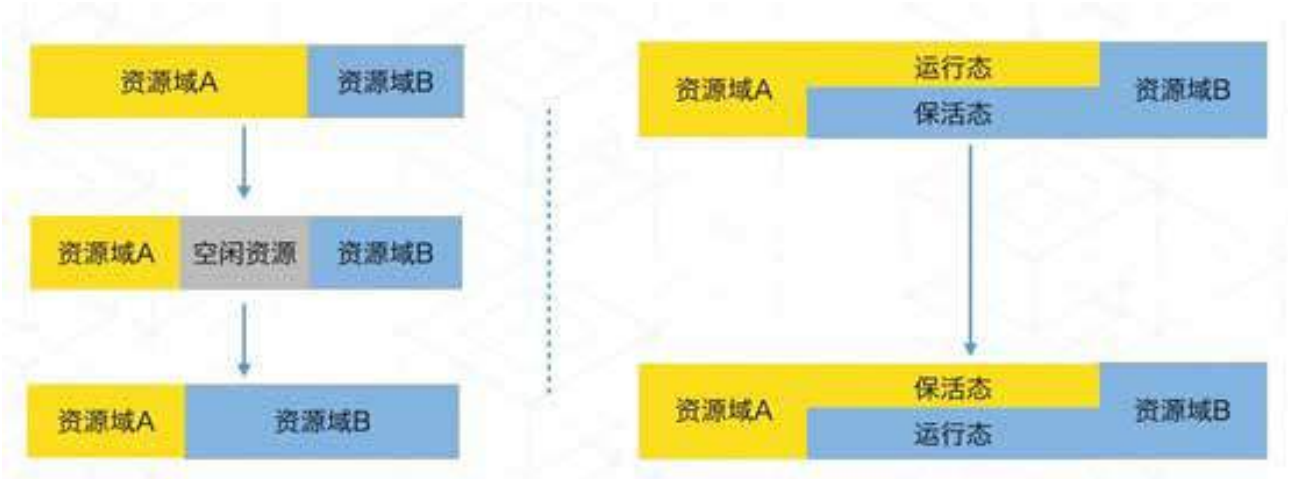
在阿里经济体下，在不断的淘宝直播，实时红包，蚂蚁森林，各种活动的不断扩张中，给技术带了复杂了场景考验。

这个时候，业务同学往往想的是什么？我的量快撑不住了，我的代码已经最优化了，我要扩容加机器，而更多的机器付出更多的成本，面对这样的情况，我们觉得应用 Service Mesh，是一个很好的解法。通过和 JVM， 系统部的配合，利用进阶的分时调度实现灵活的资源调度，不加机器。这个可以在资源调度下有更好的效果。



首先，我们假设有两个大的资源池的资源需求情况，可以看到在 X 点的时候，资源域 A 需要更多的资源，Y 点的时候，资源域 B 需要更多的资源，总量不得增加。那当然，我们就希望能借调机器。就像下面这样。

请大家看左图。



在这个方案中， 我们需要先释放资源，然后销毁进程，然后开始重建资源，然后启动资源域 B 的资源。这个过程对于大量的机器，是很重的，而且变更就是风险，关键时候，做这种变更，很有可能带来衍生影响。

而在 SOFAMosn 中，我们有了新的解法。如右图所示，有一部分资源，一直通过超卖，运行着两种应用，但是 X 点的时候，对于资源域 A，我们通过 SOFAMosn 来将流量全部转走，这样，应用的 CPU 和内存就被限制到非常低的情况，大概保留 1% 的能力。这样，机器依然可以预热，进程也不停。

在这里。我们可以看这张图。

当需要比较大的资源调度的时候，我们推送一把开关，则资源限制打开，包活状态取消。资源域 B 瞬间可以满血复活。而资源域 A 此时进入上一个状态，CPU 和内存被限制。在这里，SOFAMosn 以一个极低的资源占用。完成流量保活的能力。使得资源的快速借调成为可能。

我们对 Service Mesh 的思考与未来

Service Mesh 在蚂蚁金服经过 2 年的沉淀，最终经过 双11 的检验，在 双11 ，我们覆盖了 200+ 的双11 交易核心链路，Mosn 注入的容器数量达到了数十万，双11 当天处理的 QPS 达到了几千万，平均处理 RT<0.2 ms，SOFAMosn 本身在大促中间完成了数十次的在线升级，基本上达到了我们的预期，初步完成了基础设施和业务的第一步的分离，见证了 Mesh 化之后基础设施的迭代速度。

不论何种架构，软件工程没有银弹。架构设计与方案落地总是一种平衡与取舍，目前还有一些 Gap 需要我们继续努力，但是我们相信，云原生是远方也是未来，经过我们两年的探索和实践，我们也积累了丰富的经验。

我们相信，Service Mesh 可能会是云原生下最接近“银弹”的那一颗，未来 Service Mesh 会成为云原生下微服务的标准解决方案，接下来蚂蚁金服将和阿里集团一起深度参与到 Istio 社区中去，与社区一起把 Istio 打造成 Service Mesh 的事实标准。

04

服务网格在“路口”的产品思考与实践

宋顺，花名齐天，蚂蚁金服高级技术专家

引言?

Service Mesh 是蚂蚁金服下一代架构的核心，经过了 2 年的沉淀，我们探索出了一套切实可行的方案并最终通过了 双11 的考验。本文主要分享在当下“路口”，我们在产品设计上的思考和实践，希望能给大家带来一些启发。

为什么需要 Service Mesh?

微服务治理与业务逻辑解耦

在 Service Mesh 之前，微服务体系的玩法都是由中间件团队提供一个 SDK 给业务应用使用，在 SDK 中会集成各种服务治理的能力，如：服务发现、负载均衡、熔断限流、服务路由等。

在运行时，SDK和业务应用的代码其实是混合在一个进程中运行的，耦合度非常高，这就带来了一系列的问题：

升级成本高

每次升级都需要业务应用修改 SDK 版本号，重新发布。

在业务飞速往前跑的时候，是不太愿意停下来做这些和自身业务目标不太相关的事情的。

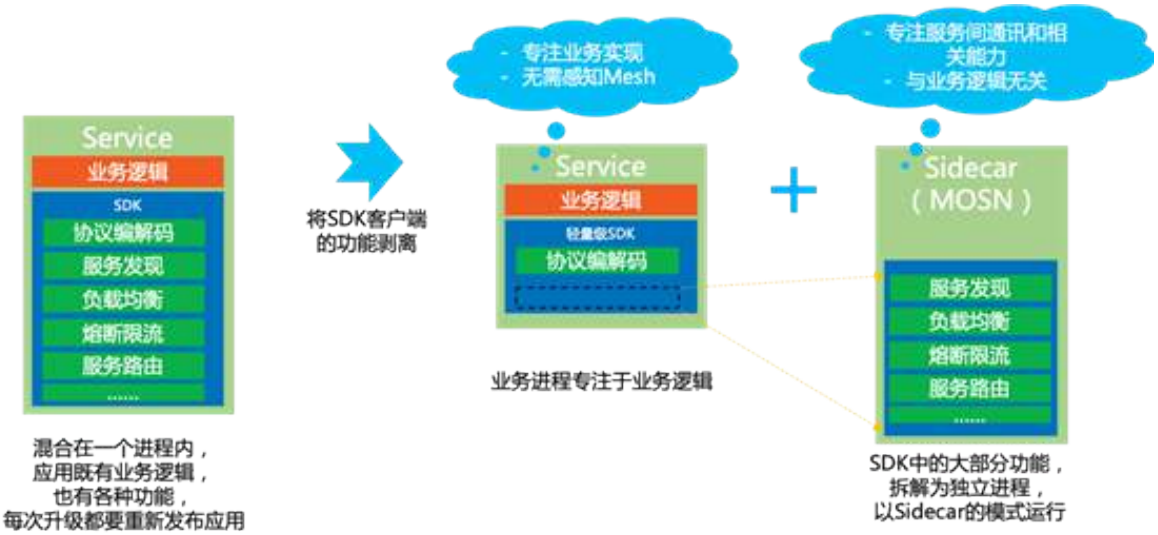
版本碎片化严重

由于升级成本高，但中间件还是会向前发展，久而久之，就会导致线上 SDK 版本各不统一、能力参差不齐，造成很难统一治理

中间件演进困难

由于版本碎片化严重，导致中间件向前演进过程中就需要在代码中兼容各种各样的老版本逻辑，戴着“枷锁”前行，无法实现快速迭代

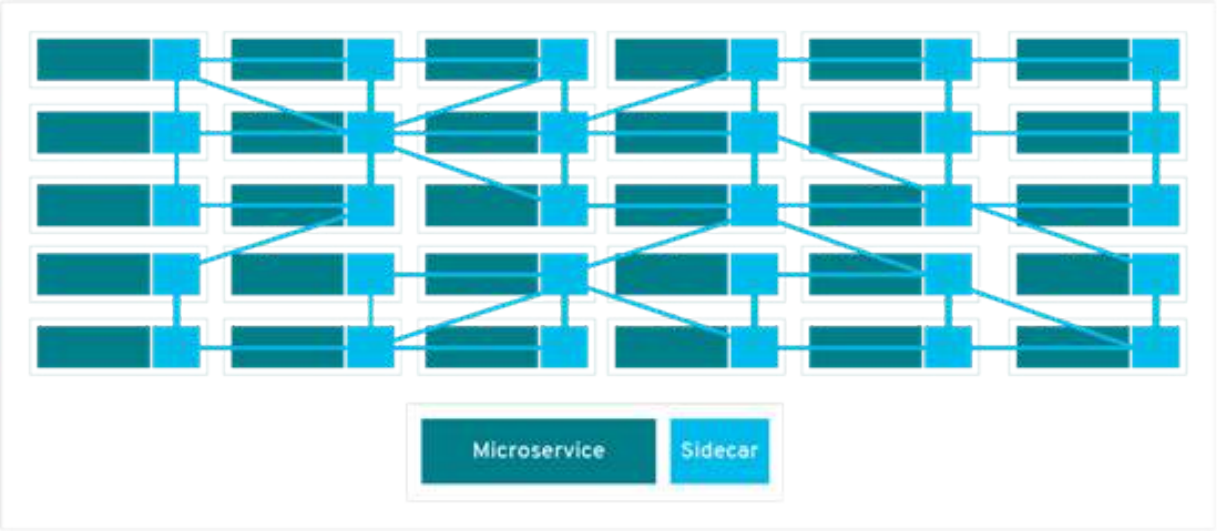
有了 Service Mesh 之后，我们就可以把 SDK 中的大部分能力从应用中剥离出来，拆解为独立进程，以 Sidecar 的模式部署。通过将服务治理能力下沉到基础设施，可以让业务更加专注于业务逻辑，中间件团队则更加专注于各种通用能力，真正实现独立演进，透明升级，提升整体效率。



异构语言统一治理

随着新技术的发展和人员更替，在同一家公司中往往会出现使用各种不同语言的应用和服务，为了能够统一管控这些服务，以往的做法是为每种语言都重新开发一套完整的SDK，维护成本非常高，而且对中间件团队的人员结构也带来了很大的挑战。

有了 Service Mesh 之后，通过将主体的服务治理能力下沉到基础设施，多语言的支持就轻松很多了，只需要提供一个非常轻量的 SDK、甚至很多情况都不需要一个单独的 SDK，就可以方便地实现多语言、多协议的统一流量管控、监控等治理需求。



图片来源https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

金融级网络安全

当前很多公司的微服务体系建设都建立在“内网可信”的假设之上，然而这个原则在当前大规模上云的背景下可能显得有点不合时宜，尤其是涉及到一些金融场景的时候。

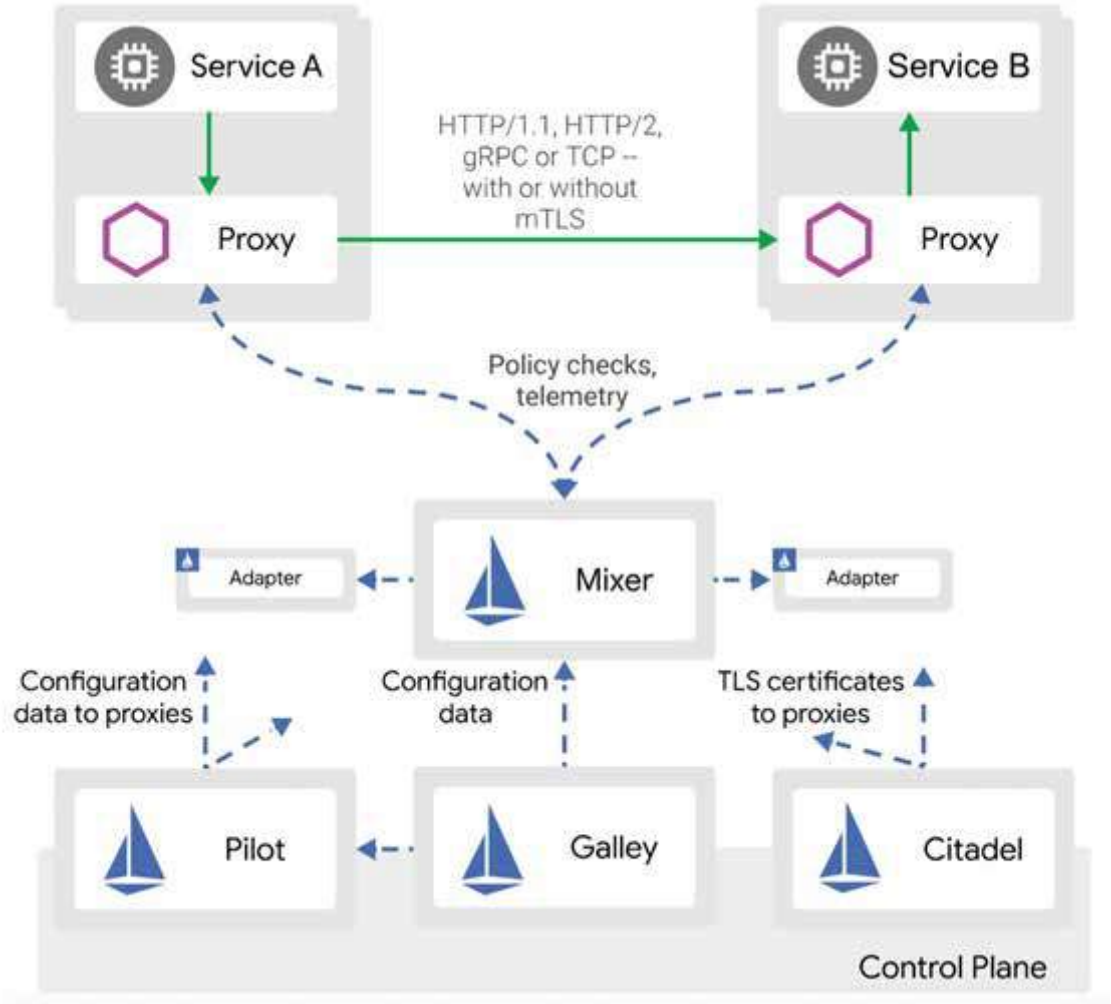
通过 Service Mesh，我们可以更方便地实现应用的身份标识和访问控制，辅之以数据加密，就能实现全链路可信，从而使得服务可以运行于零信任网络中，提升整体安全水位。

在当下“路口”的思考

云原生方案?

正因为 Service Mesh 带来了上述种种的好处，所以这两年社区中对 Service Mesh 的关注度越来越高，也涌现出了很多优秀的 Service Mesh 产品，Istio 就是其中一款非常典型的标杆产品。

Istio　以其前瞻的设计结合云原生的概念，一出现就让人眼前一亮，心之向往。不过深入进去看了之后发现，在目前阶段要落地的话，还是存在一些 gap 的。



图片来源https://istio.io/docs/concepts/what-is-istio/

Greenfield vs Brownfield

在正式展开讨论之前，我们先来看一副漫画。

图片来源https://faasandfurious.com/90

上面这幅漫画描绘了这样一个场景：

- 有两个工人在工作，其中一个在绿色的草地（Greenfield）上，另一个在棕色的土地（Brown-field）上
- 在绿色草地上的工人对在棕色土地上的工人说：“如果你没有给自己挖这么深的坑，那么你也可以像我一样做一些很棒的新东西”
- 然后在棕色土地上的工人回答道：“你倒是下来试试！”

这是一幅很有意思的漫画，我们可以认为在绿色草地上的工人是站着说话不腰疼，不过其实本质的原因还是两者所处的环境不同。

在一片未开发过的土地上施工确实是很舒服的，因为空间很大，也没有周遭各种限制，可以使用各种新技术、新理念，我们国家近几十年来的一些新区新城的建设就属于这类。而在一片已经开发过的土地上施工就大不一样了，周围环境会有各种限制，比如地下可能有各种管线，一不小心就挖断了，附近还有各种大楼，稍有不慎就可能把楼给挖塌了，所以做起事来就要非常小心，设计方案时也会受到各种约束，无法自由发挥。

对于软件工程，其实也是一样的，Greenfield 对应着全新的项目或新的系统，Brownfield 对应着成熟的项目或遗留系统。

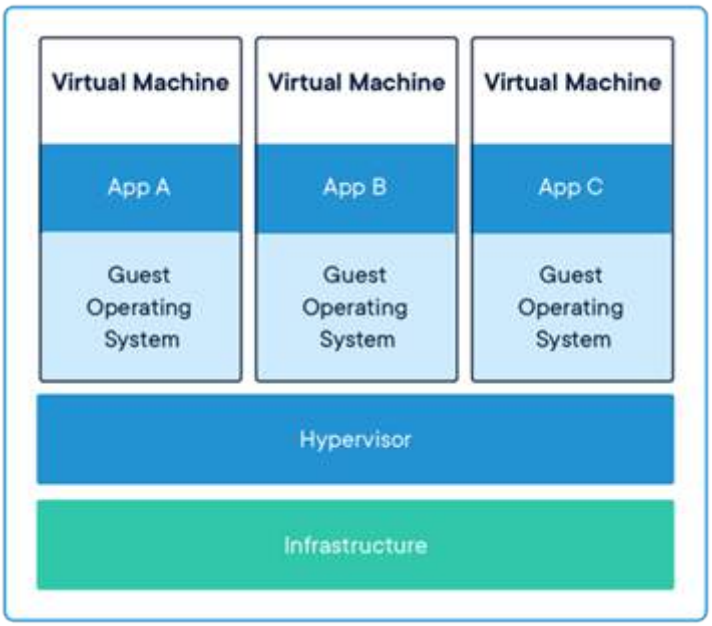
我相信大部分程序员都是喜欢做全新的项目的，包括我自己也是一样。因为可以使用新的技术、新的框架，可以按照事物本来的样子去做系统设计，自由度很高。而在开发/维护一个成熟的项目时就不太一样了，一方面项目已经稳定运行，逻辑也非常复杂，所以无法很方便地换成新的技术、新的框架，在设计新功能时也会碍于已有的架构和代码实现做很多妥协，另一方面前人可能不知不觉挖了很多坑，稍有不慎就会掉进坑里，所以行事必须要非常小心，尤其是在做大的架构改变的时候。

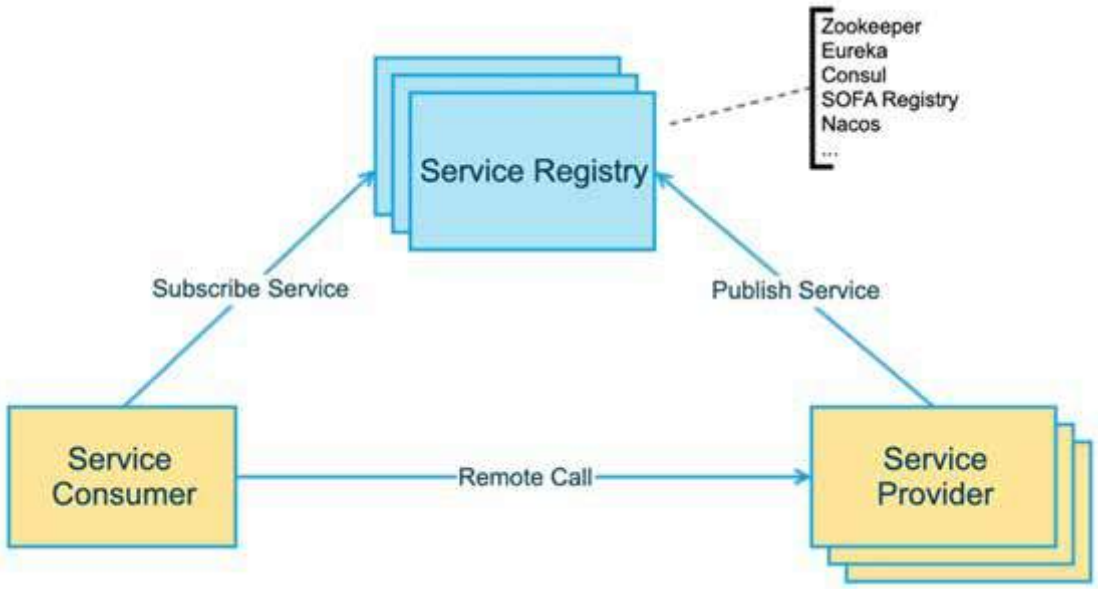
现实场景

Brownfield 应用当道

在现实中，我们发现目前大部分的公司还没有走向云原生，或者还刚刚在开始探索，所以大量的应用其实还跑在非 K8s 的体系中，比如跑在虚拟机上或者是基于独立的服务注册中心构建微服务体系。

虽然确实有少量 Greenfield 应用已经在基于云原生来构建了，但现实是那些大量的 Brownfield 应用是公司业务的顶梁柱，承载着更大的业务价值，所以如何把它们纳入 Service Mesh 统一管控，从而带来更大的价值，也就成了更需要优先考虑的话题。





独立的服务注册中心

图片来源https://medium.com/next-level-german-engineering/comparison-of-two-different-approaches-towards-container-management-4e5298736d42

云原生方案离生产级尚有一定距离

另一方面，目前 Istio 在整体性能上还存在一些有待解决的点，以下引述熬小剑老师在蚂蚁金服 Service Mesh 深度实践中的观点）：

Mixer

Mixer 的性能问题，一直都是 Istio 中最被人诟病的地方。

尤其在 Istio 1.1/1.2 版本之后引入了 Out-Of-Process Adapter，更是雪上加霜。

从落地的角度看，Mixer V1 糟糕至极的性能，已经是“生命无法承受之重”。对于一般规模的生产级落地而言，Mixer 性能已经是难于接受，更不要提大规模落地……

Mixer V2 方案则给了社区希望：将 Mixer 合并进 Sidecar，引入 web assembly 进行 Adapter 扩展，这是我们期待的 Mixer 落地的正确姿势，是 Mixer 的未来，是 Mixer 的“诗和远方”。然而社区望穿秋水，但Mixer V2 迟迟未能启动，长期处于 In Review 状态，远水解不了近渴。

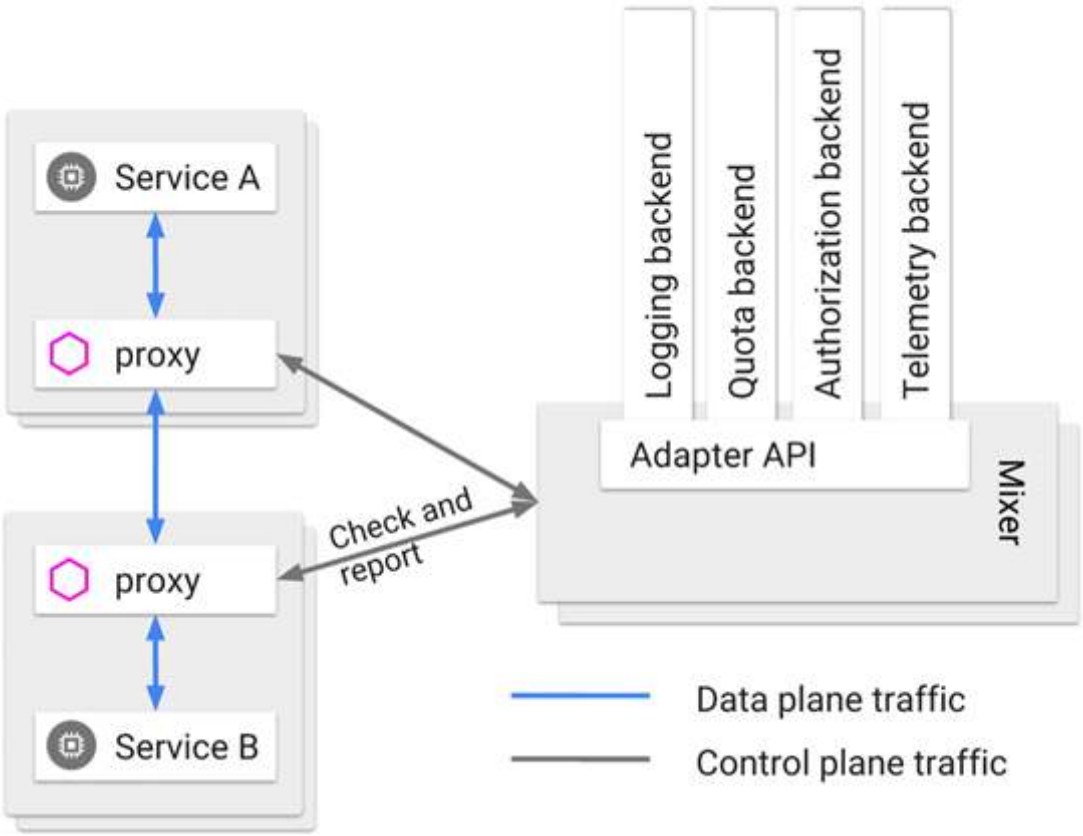
Pilot

Pilot 是一个被 Mixer 掩盖的重灾区：长期以来大家的性能关注点都在 Mixer，表现糟糕而且问题明显的Mixer 一直在吸引火力。但是当选择放弃 Mixer（典型如官方在 Istio 新版本中提供的关闭 Mixer 的配置开关）之后，Pilot 的性能问题也就很快浮出水面。

我们实践下来发现 Pilot 目前主要有两大问题：

1）无法支撑海量数据

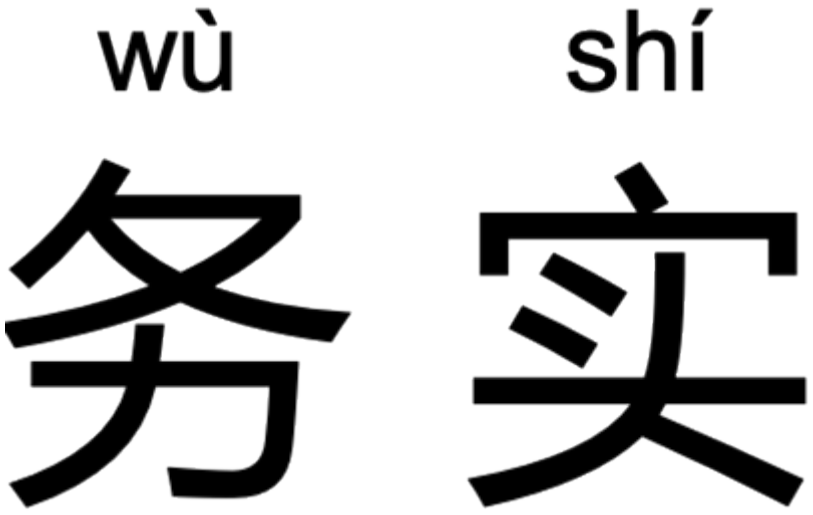
2）每次变化都会触发全量推送，性能较差



当下“路口”我们该怎么走?

我们都非常笃信云原生就是未来，是我们的“诗和远方”，但是眼下的现实情况是一方面 Brownfield 应用当道，另一方面云原生的 Service Mesh 方案自身离生产级还有一定的距离，所以在当下这个“路口”我们该怎么走?

我们给出的答案是:

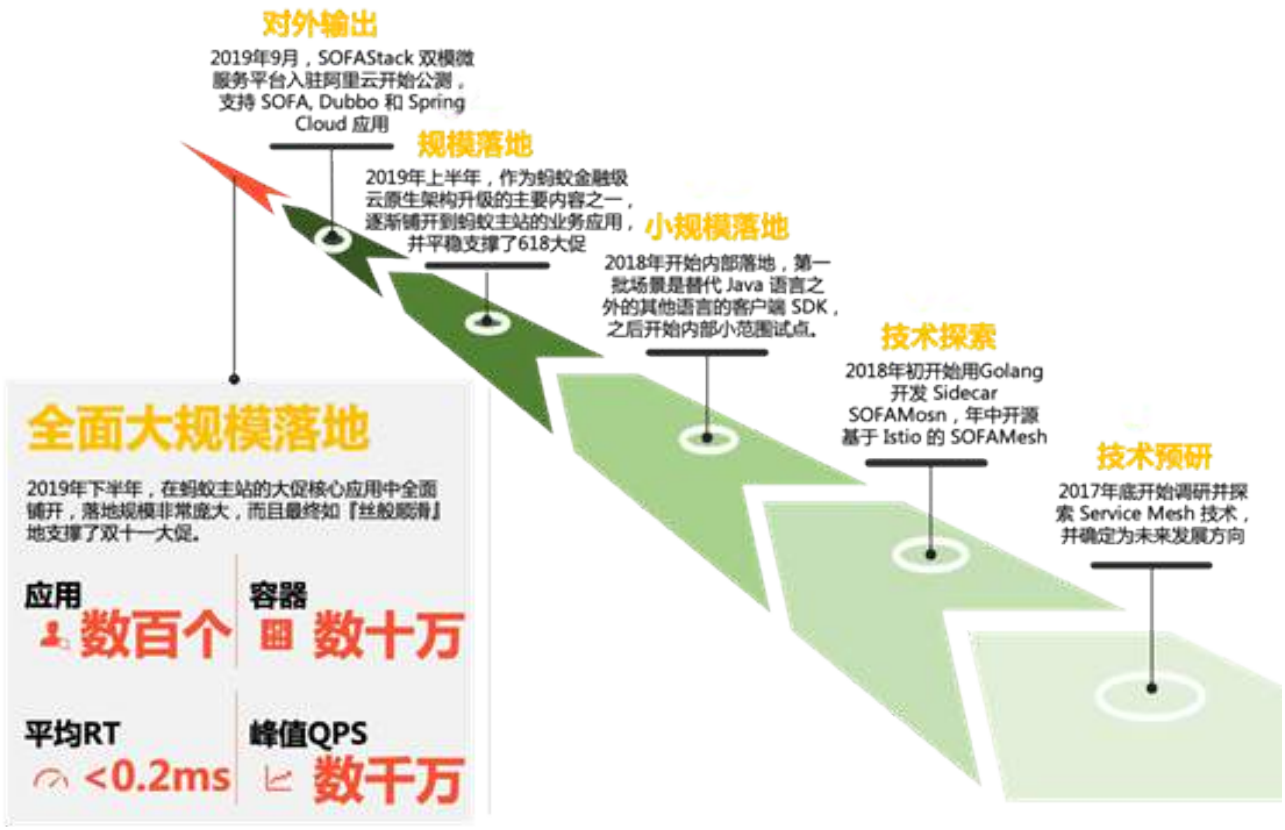


其实如前面所述，我们采用 Service Mesh 方案的初心是因为它的架构改变可以带来很多好处，如：服务治理与业务逻辑解耦、异构语言统一治理、金融级网络安全等，而且我们相信这些好处不管对 Greenfield 应用还是 Brownfield 应用都是非常需要的，甚至在现阶段对 Brownfield 应用产生的业务价值会远远大于 Greenfield 应用。

所以从“务实”的角度来看，我们首先还是要探索出一套现阶段切实可行的方案，不仅要支持 Greenfield 应用，更要能支持 Brownfield 应用，从而可以真正把 Service Mesh 落到实处，产生业务价值。

蚂蚁金服的产品实践

发展历程和落地规模



Service Mesh 在蚂蚁金服的发展历程，先后经历过如下几个阶段：

技术预研阶段：2017 年底开始调研并探索 Service Mesh 技术，并确定为未来发展方向。
技术探索阶段：2018 年初开始用 Golang 开发 Sidecar SOFAMosn ，年中开源基于 Istio 的 SOFAMesh 。
小规模落地阶段：2018 年开始内部落地，第一批场景是替代 Java 语言之外的其他语言的客户端 SDK，之后开始内部小范围试点。
规模落地阶段：2019 年上半年，作为蚂蚁金融级云原生架构升级的主要内容之一，逐渐铺开到蚂蚁金服内部的业务应用，并平稳支撑了 618 大促。
对外输出阶段：2019 年 9 月，SOFAStack 双模微服务平台入驻阿里云开始公测，支持 SOFA, Dubbo 和 Spring Cloud 应用
全面大规模落地阶段：2019 年下半年，在蚂蚁主站的大促核心应用中全面铺开，落地规模非常庞大，而且最终如“丝般顺滑”地支撑了 双11 大促。

在今年 双11，Service Mesh 覆盖了数百个交易核心链路应用，SOFAMosn 注入的容器数量达到了数十万（据信是目前全球最大的 Service Mesh 集群），双11 当天处理的 QPS 达到了几千万，平均处理响应时间<0.2 ms.

SOFAMosn 本身在大促中间完成了数十次的业务无感升级，达到了我们的预期，初步完成了基础设施和业务的第一步的分离，见证了 Mesh 化之后基础设施的迭代速度。

SOFAStack 双模微服务平台

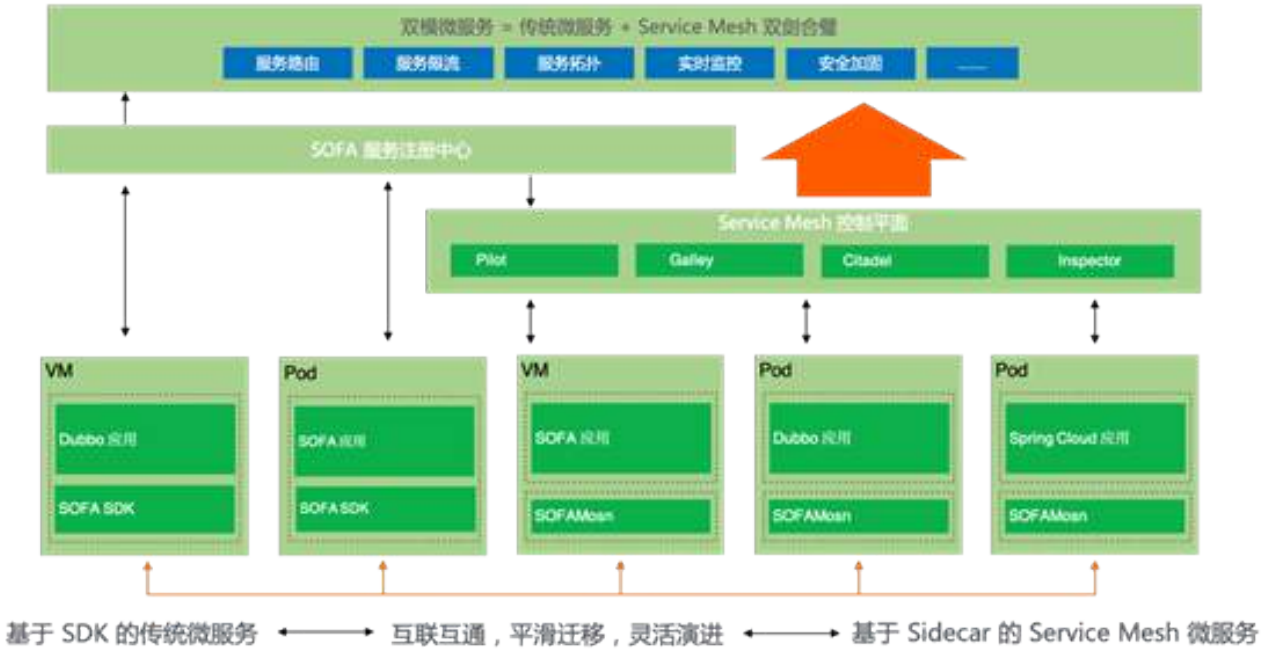
我们的服务网格产品名是 SOFAStack 双模微服务平台，这里的“双模微服务”是指传统微服务和 Service Mesh 双剑合璧，即“基于 SDK 的传统微服务”可以和“基于 Sidecar 的 Service Mesh 微服务”实现下列目标：

- 互联互通：两个体系中的应用可以相互访问；
- 平滑迁移：应用可以在两个体系中迁移，对于调用该应用的其他应用，做到透明无感知；
- 异构演进：在互联互通和平滑迁移实现之后，我们就可以根据实际情况进行灵活的应用改造和架构演进；

在控制面上，我们引入了Pilot实现配置的下发（如服务路由规则），在服务发现上保留了独立的 SOFA 服务注册中心。

在数据面上，我们使用了自研的 SOFAMosn，不仅支持 SOFA 应用，同时也支持 Dubbo 和 Spring Cloud 应用。

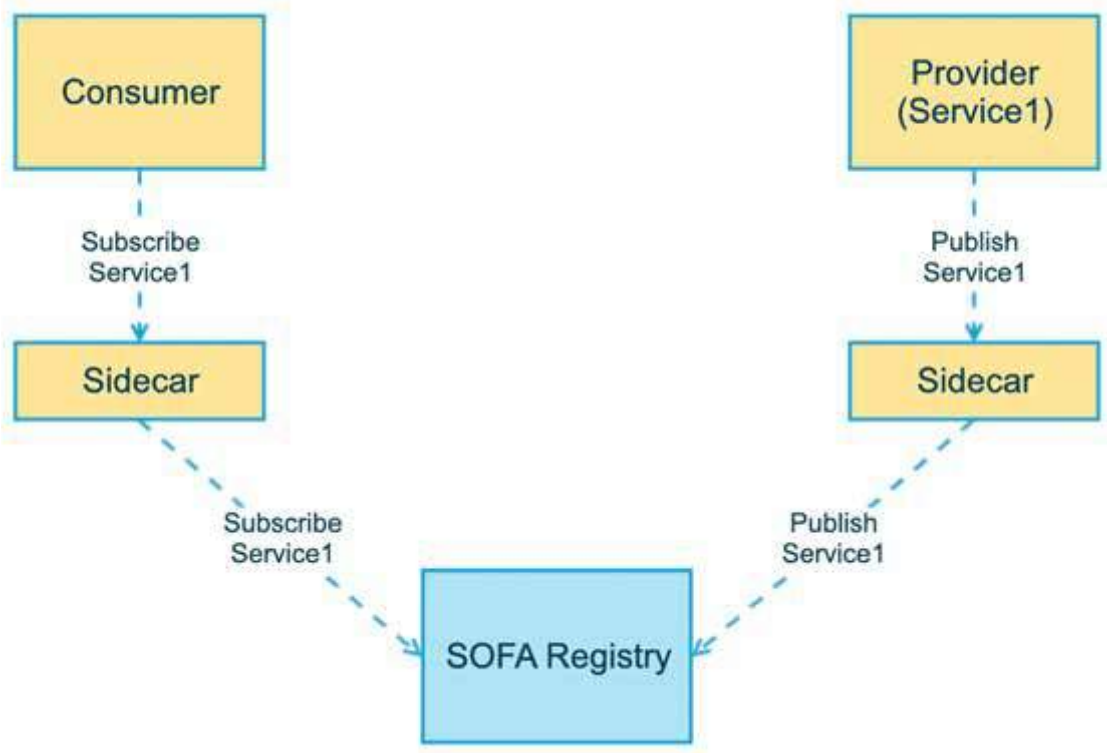
在部署模式上，我们不仅支持容器/K8s，同时也支持虚拟机场景。



大规模场景下的服务发现

要在蚂蚁金服落地，首先一个需要考虑的就是如何支撑 双11 这样的大规模场景。前面已经提到，目前 Pilot 本身在集群容量上比较有限，无法支撑海量数据，同时每次变化都会触发全量推送，无法应对大规模场景下的服务发现。

所以，我们的方案是保留独立的 SOFA 服务注册中心来支持千万级的服务实例信息和秒级推送，业务应用通过直连 Sidecar 来实现服务注册和发现。

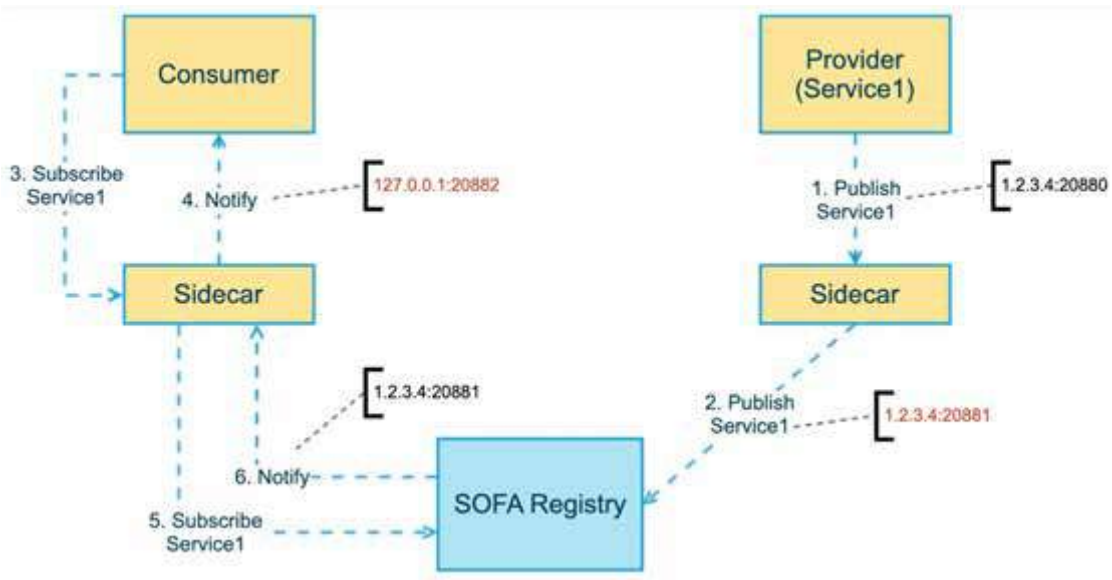


流量劫持

Service Mesh 中另一个重要的话题就是如何实现流量劫持：使得业务应用的 Inbound 和 Outbound 服务请求都能够经过 Sidecar 处理。

区别于社区的 iptables 等流量劫持方案，我们的方案就显得比较简单直白了，以下图为例：

1. 假设服务端运行在 1.2.3.4 这台机器上，监听 20880 端口，首先服务端会向自己的 Sidecar 发起服务注册请求，告知 Sidecar 需要注册的服务以及 IP + 端口(1.2.3.4:20880)；
2. 服务端的 Sidecar 会向 SOFA 服务注册中心发起服务注册请求，告知需要注册的服务以及 IP + 端口，不过这里需要注意的是注册上去的并不是业务应用的端口(20880)，而是 Sidecar 自己监听的一个端口(例如：20881)；
3. 调用端向自己的 Sidecar 发起服务订阅请求，告知需要订阅的服务信息；
4. 调用端的 Sidecar 向调用端推送服务地址，这里需要注意的是推送的 IP 是本机，端口是调用端的 Sidecar 监听的端口(例如：20882)；
5. 调用端的 Sidecar 会向 SOFA 服务注册中心发起服务订阅请求，告知需要订阅的服务信息；
6. SOFA 服务注册中心向调用端的 Sidecar 推送服务地址(1.2.3.4:20881)；



经过上述的服务发现过程，流量劫持就显得非常自然了：

1. 调用端拿到的“服务端”地址是127.0.0.1:20882，所以就会向这个地址发起服务调用；
2. 调用端的 Sidecar 接收到请求后，通过解析请求头，可以得知具体要调用的服务信息，然后获取之前从服务注册中心返回的地址后就可以发起真实的调用(1.2.3.4:20881)；
3. 服务端的Sidecar接收到请求后，经过一系列处理，最终会把请求发送给服务端(127.0.0.1:20880)；

可能会有人问，为啥不采用 iptables 的方案呢？主要的原因是一方面 iptables 在规则配置较多时，性能下滑严重，另一个更为重要的方面是它的管控性和可观测性不好，出了问题比较难排查。

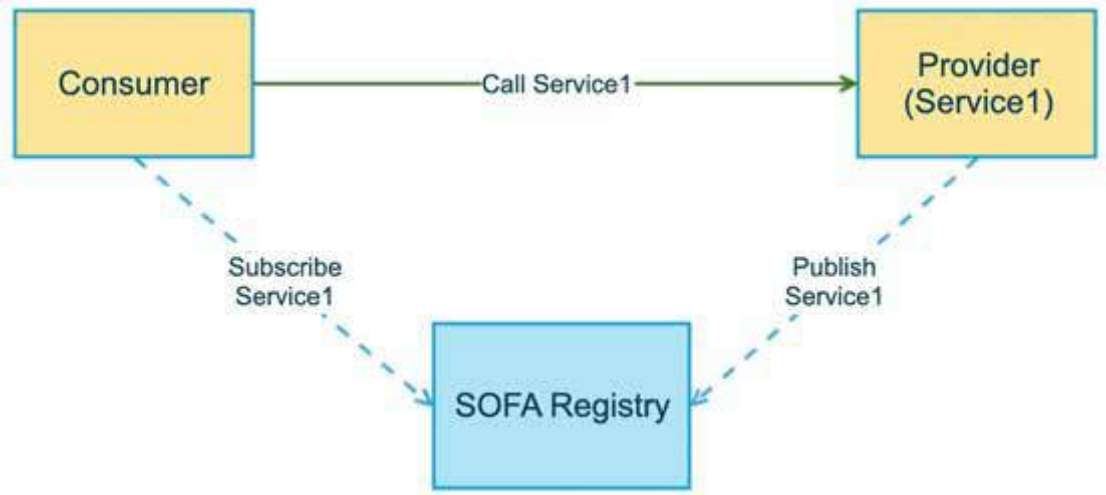
平滑迁移

平滑迁移可能是整个方案中最为重要的一个环节了，前面也提到，在目前任何一家公司都存在着大量的 Brownfield 应用，它们有些可能承载着公司最有价值的业务，稍有闪失就会给公司带来损失，有些可能是非常核心的应用，稍有抖动就会造成故障，所以对于 Service Mesh 这样一个大的架构改造，平滑迁移是一个必选项，同时还需要支持可灰度和可回滚。

得益于独立的服务注册中心，我们的平滑迁移方案也非常简单直白：

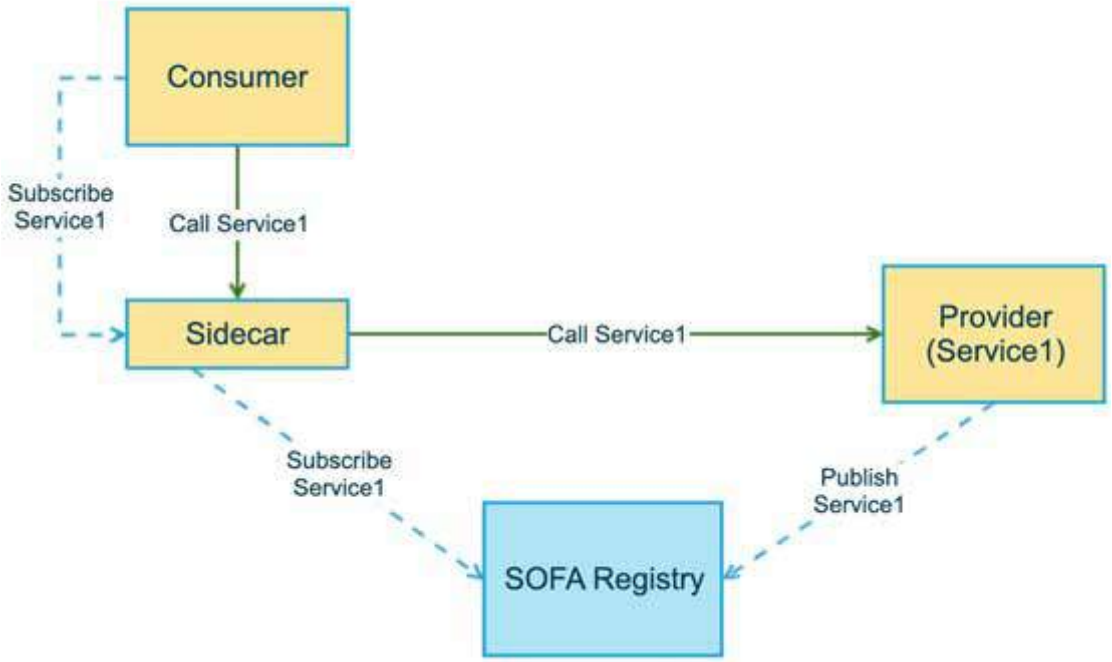
初始状态

以一个服务为例，初始有一个服务提供者，有一个服务调用者。



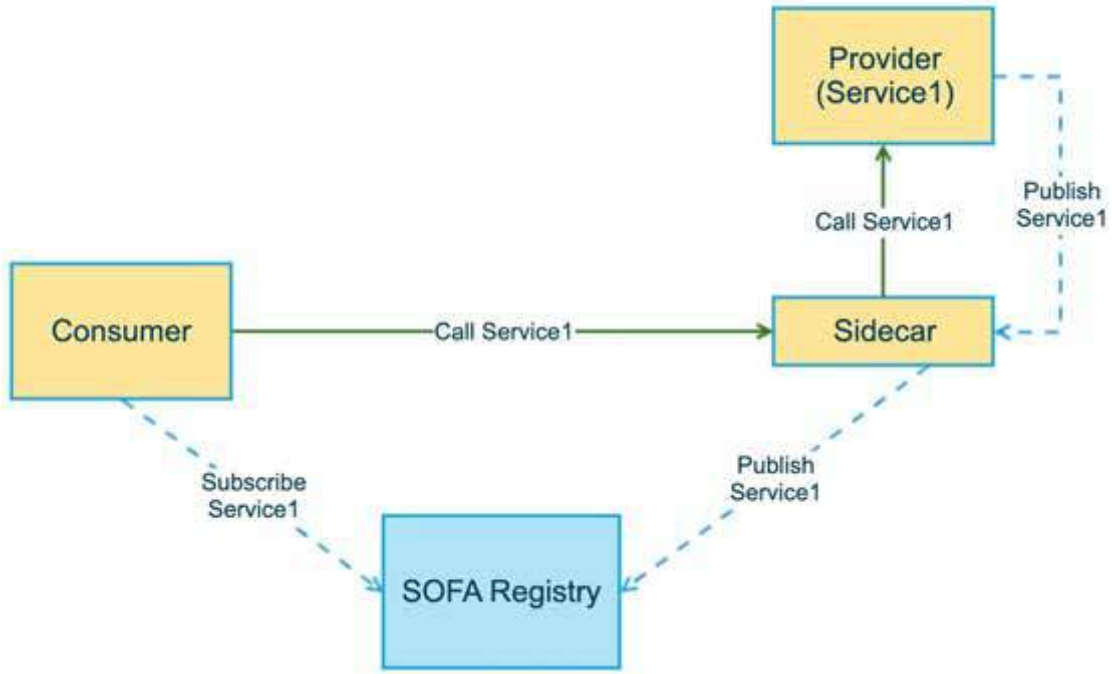
透明迁移调用方

在我们的方案中，对于先迁移调用方还是先迁移服务方没有任何要求，这里假设调用方希望先迁移到 Service Mesh 上，那么只要在调用方开启 Sidecar 的注入即可，服务方完全不感知调用方是否迁移了。所以调用方可以采用灰度的方式一台一台开启 Sidecar，如果有问题直接回滚即可。

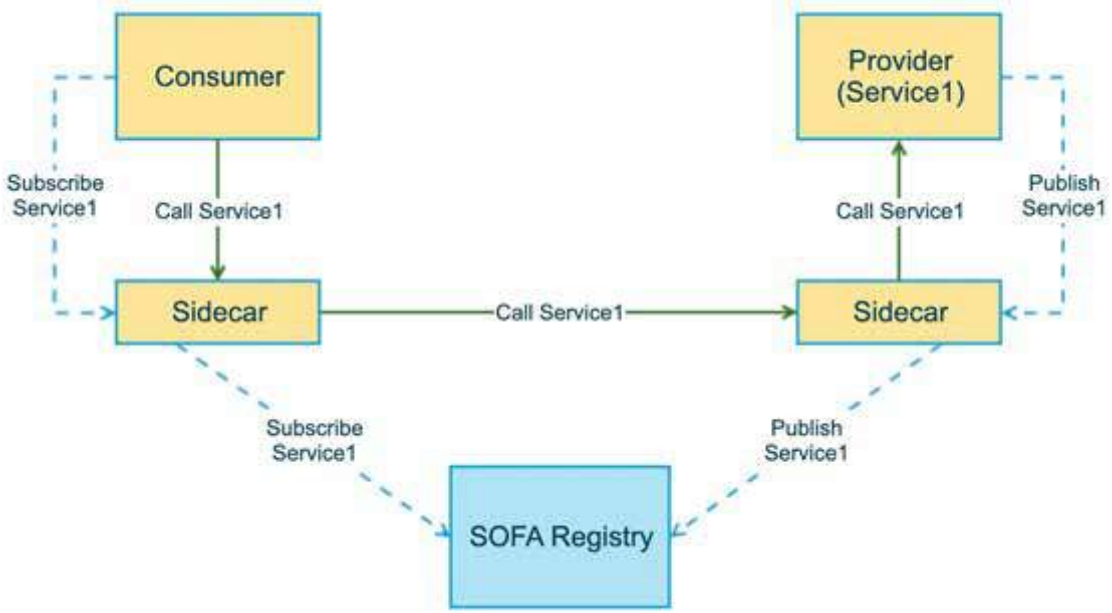


透明迁移服务方

假设服务方希望先迁移到 Service Mesh 上，那么只要在服务方开启 Sidecar 的注入即可，调用方完全不感知服务方是否迁移了。所以服务方可以采用灰度的方式一台一台开启 Sidecar，如果有问题直接回滚即可。



终态

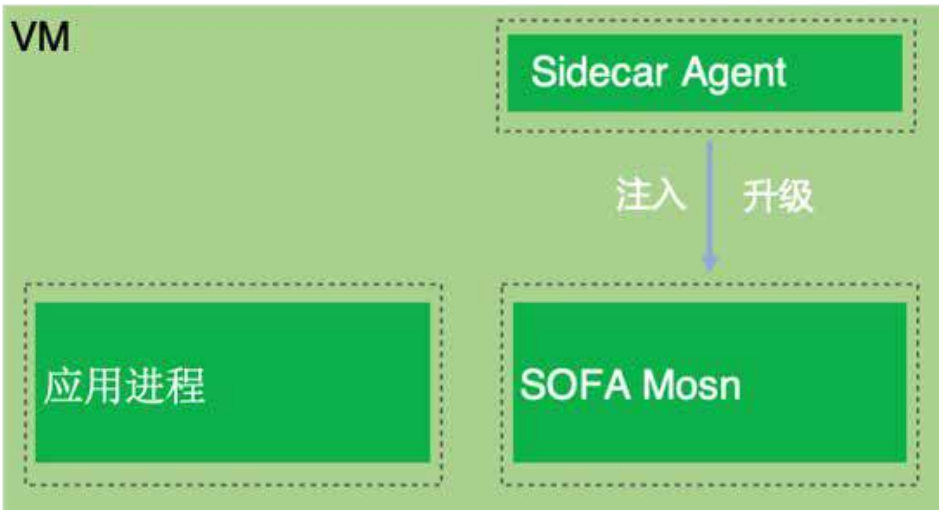


多协议支持

考虑到目前大部分用户的使用场景，除了 SOFA 应用，我们同时也支持 Dubbo 和 Spring Cloud 应用接入SOFAStack 双模微服务平台，提供统一的服务治理。多协议支持采用通用的 x-protocol，未来也可以方便地支持更多协议。

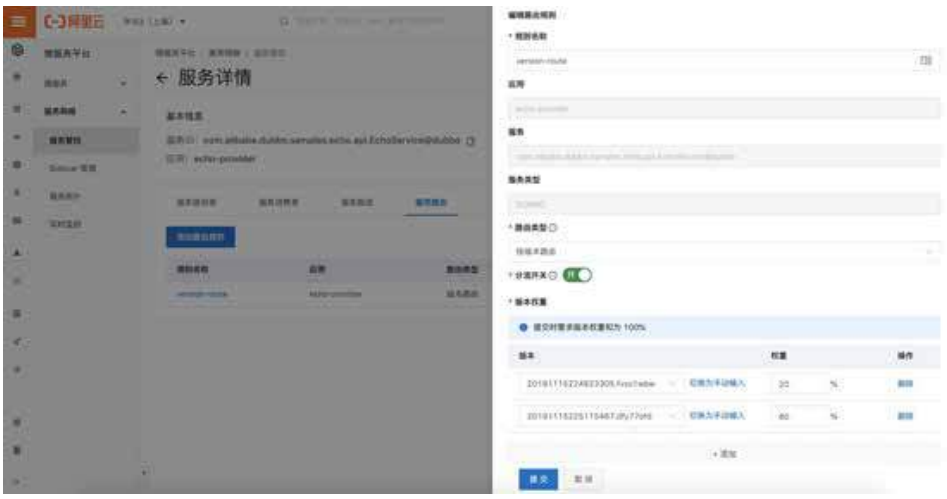
虚拟机支持

在云原生架构下，Sidecar 借助于 K8s 的 webhook/operator 机制可以方便地实现注入、升级等运维操作。然而大量系统还没有跑在 K8s 上，所以我们通过 agent 的模式来管理 Sidecar 进程，从而可以使 Service Mesh 能够帮助老架构下的应用完成服务化改造，并支持新架构和老架构下服务的统一管理。

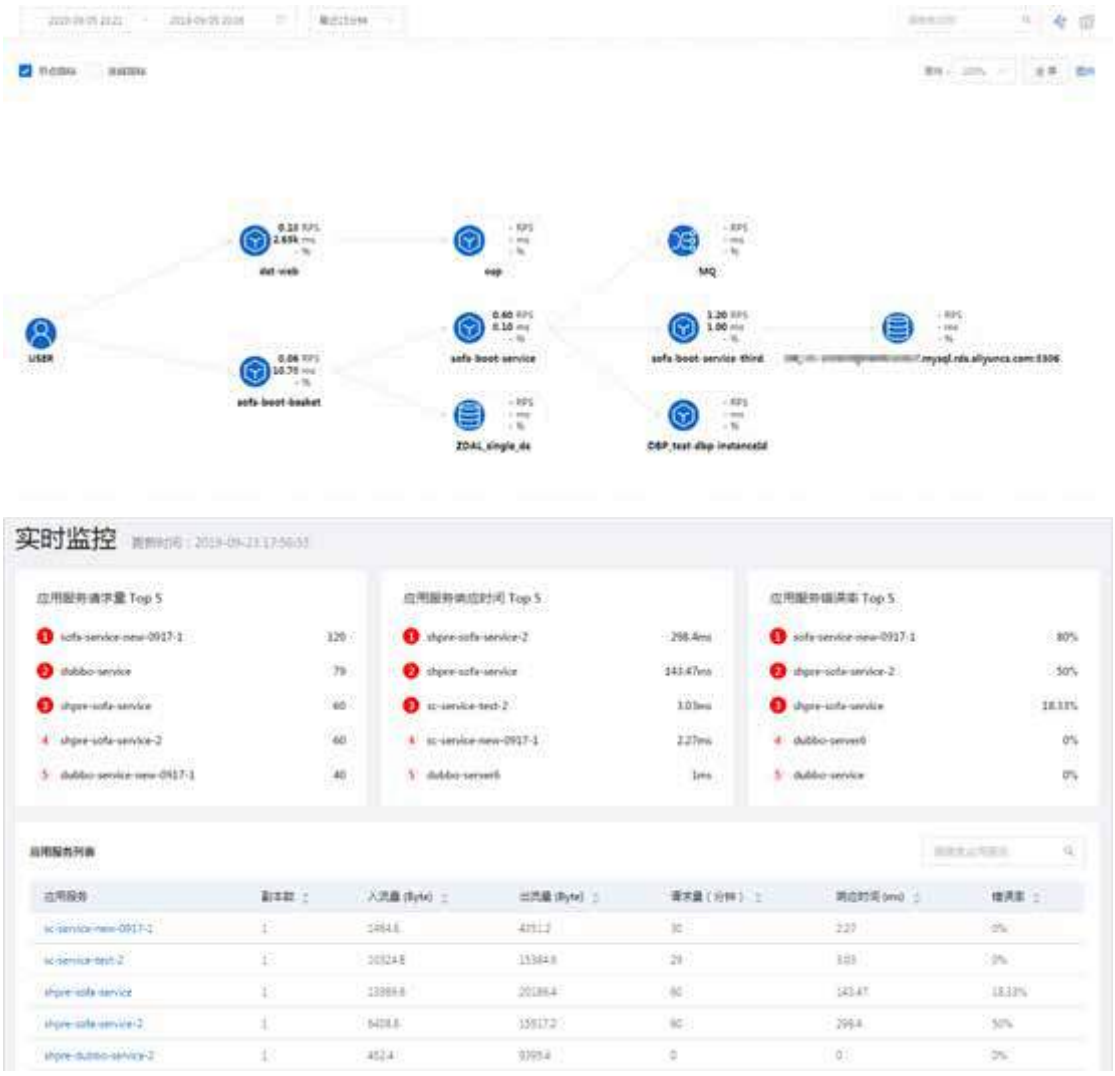


产品易用性

在产品易用性上我们也做了不少工作，比如可以直接在界面上方便地设置服务路由规则、服务限流等，再也不用手工写 yaml 了：



也可以在界面上方便地查看服务拓扑和实时监控：



展望未来

拥抱云原生

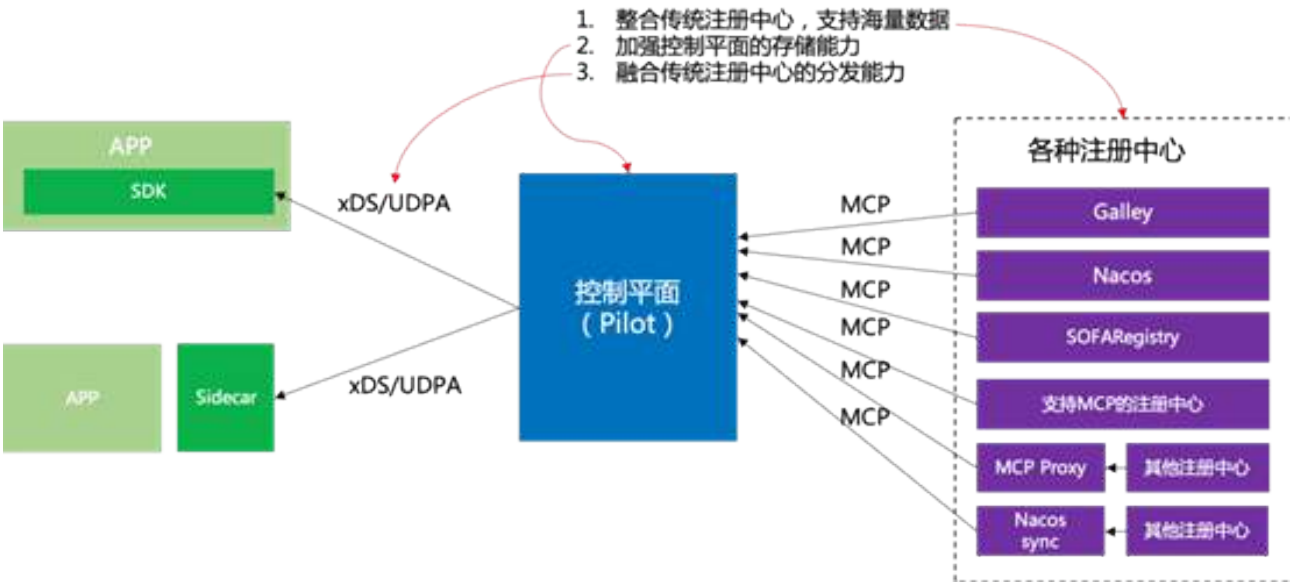
目前已经能非常清楚地看到整个行业在经历从云托管(Cloud Hosted)到云就绪(Cloud Ready)直至云原生（ Cloud Native ）的过程，所以前面也提到我们都非常笃信云原生就是未来，是我们的“诗和远方”，虽然眼下在落地过程中还存在一定的 gap，不过相信随着我们的持续投入，gap 会越来越小。

另外值得一提的是我们拥抱云原生其根本还在于降低资源成本，提升开发效率，享受生态红利，所以云原生本身不是目的，而是手段，切不可本末倒置了。

持续加强 Pilot 的能力

为了更好地拥抱云原生，后续我们也会和 Istio 社区共建，持续加强 Pilot 的能力。

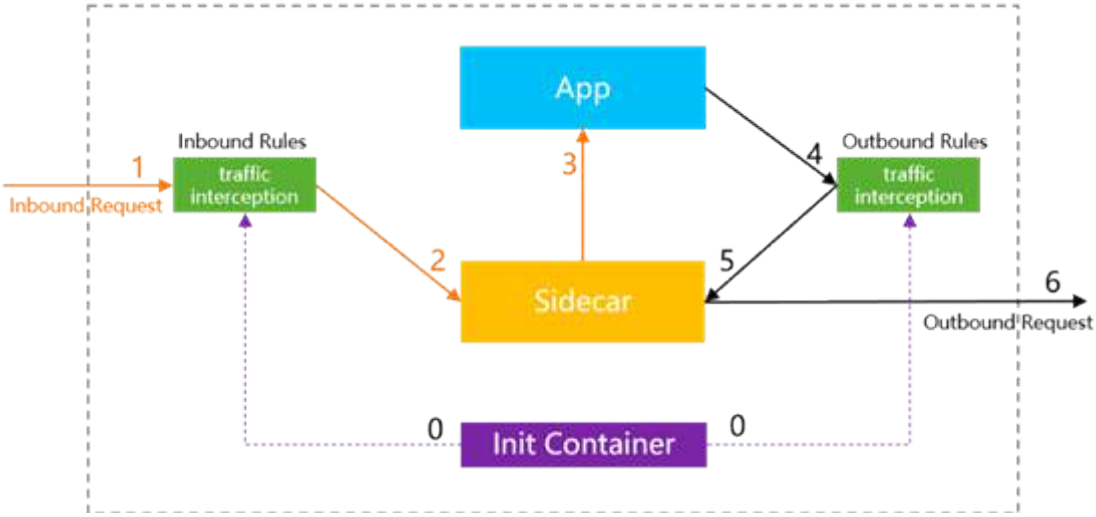
而就在最近，在综合了过去一年多的思考和探索之后，蚂蚁金服和阿里巴巴集团的同事们共同提出了一套完整的解决方案，来融合控制平面和传统注册中心/配置中心，从而可以在保持协议标准化的同时，加强 Pilot 的能力，使其逐步具备生产级的能力。



支持透明劫持

前面提到在蚂蚁金服的实践中是基于服务注册中心来实现流量劫持的，该方案不管是在性能、管控能力还是可观测性方面都是不错的选择，不过对应用存在一定的侵入性（需要引入一个轻量的注册中心 SDK）。

考虑到很多用户对性能要求没那么敏感，同时有大量遗留系统希望通过 Service Mesh 实现统一管控，所以后续我们也会支持透明劫持，同时在管控性和可观测性方面会做增强。



结语

基于“务实”的理念，Service Mesh 在蚂蚁金服经过了 2 年的沉淀，我们探索出了一套现阶段切实可行的方案并最终通过了 双11 的考验。在这个过程中，我们也愈发体验到了 Service Mesh 带来的好处，例如 SOFAMosn 在大促中间完成了数十次的业务无感升级，见证了 Mesh 化之后基础设施的迭代速度。

我们判断，未来 Service Mesh 会成为云原生下微服务的标准解决方案，所以我们也会持续加大对 Service Mesh 的投入，包括接下来蚂蚁金服将和阿里巴巴集团一起深度参与到 Istio 社区中去，和社区一起把 Istio 打造成 Service Mesh 的事实标准。

分档化

01

CSE：Serverless 在阿里巴巴 双11 场景的落地

周新宇，花名尘央，阿里云中间件技术部技术专家
史明伟，花名世如，阿里云中间件技术部技术专家
王川，花名弗丁，阿里云中间件技术部技术专家
夏佐杰，花名刘禅，阿里云中间件技术部高级开发工程师
陶宇田，花名宇拓，阿里云中间件技术部技术专家
许晓斌，阿里云中间件技术部高级技术专家

云计算时代，Serverless 作为云原生重要技术组成部分，一开始便承载了太多的使命 —— 承诺了云计算时代最典型并极具挑战的多维度服务指标： 无服务运维、极速弹性伸缩、按量付费等。 这些极具挑战并富有吸引力的服务指标带给了行业极大的想象空间，但任何技术红利的普及并不是一蹴而就，或者像宣称的那样美好。Serverless 在阿里巴巴落地的时候，同时面临了两类诉求不同的用户：

- 第一类是积极拥抱 Serverless 愿意将其代码修改成 FaaS 形态的用户，这类用户以 BFF（Backend For Frontend）层为典型，这类用户的代码天生具备良好的轻量、无状态等特性。
- 第二类则是存量应用，存量应用作为目前商业系统的核心组成部分，是最需要享受 Serverless 服务红利，但也是最难享受到这些服务红利的群体， 面对 Serverless 严格的无状态要求，以及客户对系统稳定性、业务正确性的担忧，存量应用的改造代码设计的难度和成本太高，因此他们希望能够在不更改代码结构的前提下享受 Serverless 红利。

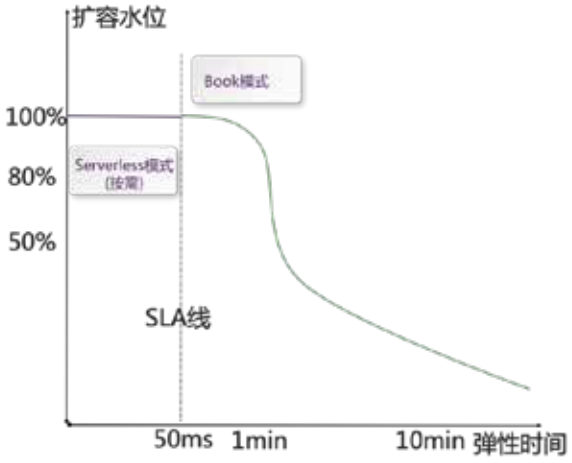
此外，由于是严苛的电商场景，而且会经历 双11，稳定性、性能、可观测性等要求一条也不能少。CSE（Cloud Service Engine）作为阿里巴巴内部孵化的一款 Serverless 产品，满足了不同用户的各种严苛需求。本文将深入介绍 CSE 的相关技术。

CSE 面临的难题

阿里巴巴内部业务一般分为在线业务和离线业务，在线业务拥有一个庞大的资源池，且资源平均利用率并不高(未考虑混部)，采用 Book 模式进行业务部署是无法进一步提高利用率、降低成本的关键原因。

Book 模式会按照应用的峰值进行部署，且业务 Owner 在进行容量评估时往往会留有余地，导致资源池利用率低。CSE 采用 Serverless 按需模式进行部署，通过快速弹缩、分时复用和高密度部署来大幅度降低成本。

Serveless 按需模式与 Book 模式两者的区别在于业务的启动时间：



应用扩容水位跟弹性的关系图例

假设业务单次请求的 SLA 在 50ms，如果业务启动时间能做到 50ms 以下，就可以等待请求到来时再按需进行扩容，反之则只能根据水位进行 Book。启动时间越短，扩容时机可推迟到越晚，缩容时机可提前至越早，成本越低。

但现实情况是，大多数在线业务的启动时间都在分钟级，甚至有的达到 30 分钟。在线业务一般基于 HSF 和潘多拉框架进行研发，同时会依赖大量的中间件富客户端以及其它部门提供的二方包，启动过程中包含了大量的初始化工作，比如网络连接、缓存加载、配置加载等。

因在线业务的启动瓶颈在于业务自身的代码，像 AWS Lambda、函数计算等 Function 场景对 IaaS 或 Runtime 框架的弹性优化手段，在在线业务的场景下基本没有用武之地。在无法侵入业务代码的前提下，如何将业务启动时间从分钟级优化至秒级甚至毫秒级是 CSE 研发阶段重点攻坚的方向。

CSE Zizz：一种针对存量应用的 Serverless 方案

阿里巴巴 CSE 团队一直在寻找一种通用的以及低成本的弹性方案，来支撑阿里巴巴在线业务大规模地透明演进至 Serverless 架构。今年团队提出了一种新的弹性方案（CSE Zizz），并于 双11 在核心应用上验证了该能力。我们能够以不到原有 1/10 的资源保有低功耗态的实例，并在有需要的时候在秒级别恢复到在线态，相比冷启动的分钟级时间，提升了两个数量级。

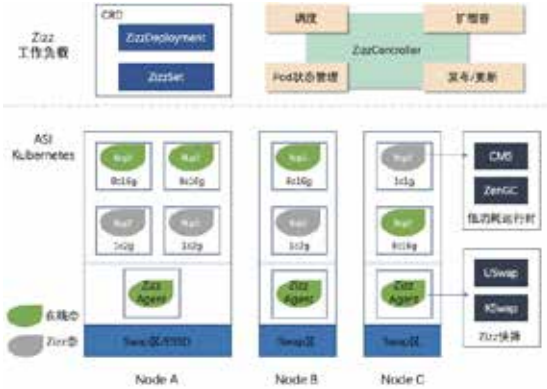
Zizz 方案基于一个核心和一个假设：

- 核心是热备（Standby）：既然应用的冷启动时间如此长且很难优化，那么提前启动一批实例进行 Standby，在流量峰值临时进行上线接流，在流量低谷时进行离线处理（服务发现层面的离线）；
- 假设是离线的实例功耗非常低：如果热备的实例需要占据全部的物理资源，那么与 Book 模式并没有本质的区别。但理论上，处于离线状态的实例没有前端流量的驱动，应该仅存在少量的后台任务在运行。所以对于 Standby 的实例，可以将其换入一个低功耗的状态，将 CPU 和内存的规格降至很低，以极低的成本进行 Standby，为分时复用和高密度部署带来更高的灵活性。

基于这两点，CSE Zizz 研发了几种核心技术：

- 一种基于弹性堆的低功耗技术；
- 基于内核态和用户态的 Swap 能力的内存弹性技术；
- 基于 Inplace Update 的实例规格动态升降配技术。

同时，将这几种核心技术进行包装，通过 Kubernetes 自定义 CRD 的方式以 K8s 工作负载的方式进行能力输出，整体架构如下：



CSE Zizz 的核心技术

在嵌入式领域， 低功耗运行模式被广泛接受， 包括 OS 设计和应用设计。 针对 OS 在设计阶段都会考虑到系统低功耗运行的情况，而对于应用，也会在设计和开发阶段考虑到低功耗运行的情况，通常应用会注册系统低功耗运行事件，在应用收到系统的低功耗运行事件之后，会进行一系列逻辑处理使自己进入低功耗运行状态。 基于这样的启发，在未来云计算环境下，应用的运行模式应该也需要支持类似的设计，这也是在 CSE Zizz 方案中引入应用低功耗运行时的概念，从而使整个 Zizz 方案更好地适配 Serverless 场景需求。 Zizz 低功耗运行时核心技术主要包括下面三个部分。

CPU 低功耗

按照 Zizz 架构设计，进入低功耗模式，我们会对实例的资源配置进行降级； 针对 CPU 资源，我们维护一个低功耗 CPU Pool（CPU Set），CPU Pool 初始值是 1C，会随着加入这个 CPU Pool 中低功耗实例的数量，增加其 CPU Core 的数量，这样同一个 Node 上的多个低功耗实例争抢一个 CPU Pool 中的资源。从 CPU 资源利用角度，多个低功耗实例构成了一个分时复用 CPU 资源的低功耗 Serverless 场景。

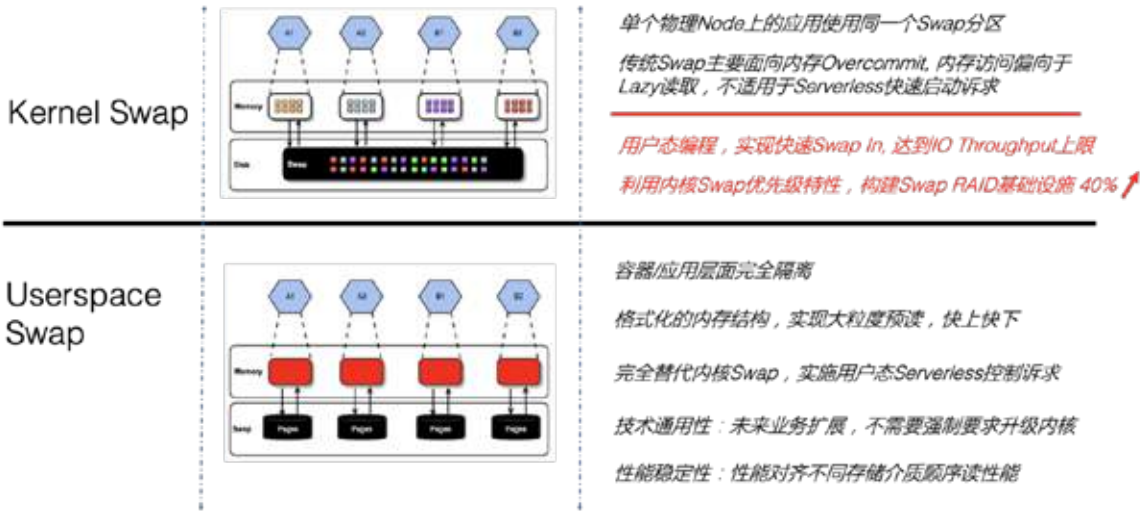
内存低功耗

Zizz 方案设计初衷是不绑定特定的进程运行时，希望为所有类型应用实例提供通用的低功耗解决方案，这种低功耗方案能够赋予实例运行时根据请求流量弹性伸缩内存，而在没有任何流量请求的情况下，能够提供一种极致的低成本运行时。

经过探索和论证，我们认为 Linux Swap 技术能够完成这样的使命。Linux Swap 向系统提供了一种透明的与进程运行时无关的内存扩展能力：根据系统当前内存压力，将系统的匿名内存按照 LRU 访问活跃度从低到高依次换出到外部低速存储介质，从而达到扩展系统可用内存的目的；在具体换出过程中，系统会根据 Swap 子系统的其它设置，根据当前内存 Anonymous 和 Filecache 的占比，将不活跃的 Anonymous 内存 Page 换出到外部低速存储设备上；从系统内存的角度来看，Swap 是一个扩展低速内存 Pool；从成本的角度，Swap 提供了一种低成本的进程运行时上下文； 面向未来，我们希望提供一种不限于内存和磁盘的多级存储架构，承载实例低功耗运行需要的上下文。

结合 Serverless 弹性场景，要求运行实例能够快速进入和退出低功耗模式，要求能够对实例的运行时状态根据流量完全可控，而目前 Linux swap 设计和实现更多的是面向系统内存 Overcommit，其内存换入的实现更多偏向于 Lazy 方式基于 Page 粒度的内存访问形式，这种内存换入的方式在真实的业务系统中会造成 RT 抖动，在商业系统中是不可接受的。这要求我们必须基于目前的 Linux Swap 技术提供一套用户态完全可控的 Swap 换入换出实现。

下图展示了我们在 Swap 基础上做的一些系统定制和创新实现。 针对传统 Linux Swap，我们利用用户态编程，并发实现快速 Swap In， 在不同存储介质上，能够达到其最大 IO Throughput。



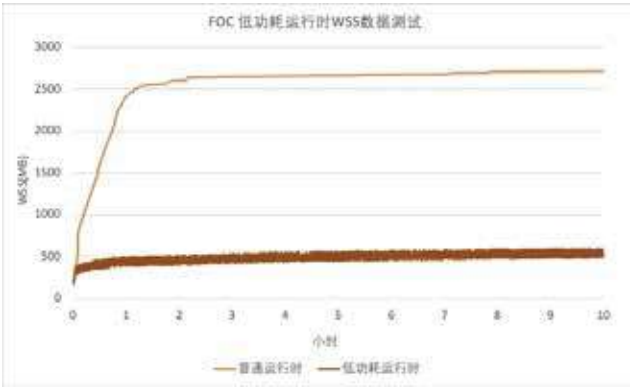
考虑到未来大规模使用场景， 以及目前 Linux Kernel Swap 存在的一些问题，我们开发了 User-space swap， 实现了 Per Process Swap 隔离存储，基于此，能够实现大粒度快速 Swap In，性能对齐不同存储介质顺序读性能。

Elastic Heap

考虑到目前阿里巴巴内部普遍以 Java 技术栈为基础的运行时现状，存量应用尤为如此。针对 Java 应用，CSE Zizz 方案利用 AJDK Elastic Heap 技术，使其作为 Java 应用实例低功耗运行的基础；Elastic Heap 技术能够有效避免 Full GC 对 JVM Heap 区域及关联内存区域的大面积访问，只对局部 Heap 区域的快速内存回收实现了低功耗运行，只使用相对较小的 Working Set Size 内存，避免了传统 Java 运行时堆区内存快速扩张对 Swap 子系统造成的压力，有效减轻了系统 IO 负载。

CSE Zizz 方案利用 AJDK 提供 Elastic Heap 的特性和相关命令，在没有请求流量的情况下， 使运行实例进入低功耗状态，使运行实例在低功耗状态下保持极低运行 Memory Footprint。

下面是针对阿里巴巴内部 4Core8G 内存规格典型应用的运行时 WSS(Working Set Size) 评估，在使用传统固定堆的情况下，内存回收采用 CMS，该应用实例的 WSS 会一路上涨至 2.7G 左右并稳定下来；使用基于 Elastic Heap 调参后的运行时 Working Set Size 内存如下所示：



AJDK Elastic Heap 技术能够保障目标评估应用实例在长达 32 小时的测试中，WSS 稳定在 500~600M。使用 Elastic Heap 技术，结合 Swap 可以为 Java 应用实例提供非常稳定的低功耗运行时、能够使运行实例在低功耗状态下维持极低运行 Memory Footprint 和 WSS，有效降低了存量应用低功耗运行的内存和 IO 成本。

在 Zizz 研发过程中，考虑到未来云计算的场景，我们做出了一个极具挑战的决策：抛弃传统的本地数据盘， Zizz 方案 Swap 存储全面使用阿里云 ESSD 存储服务。

目前 CSE Zizz 使用的 ESSD 规格为：PL1 级别，最大吞吐量 350M/s ESSD 存储服务。借助快速并发换入实现：目前换入 IO Throughput 达到 ESSD 上限，300M-350M/s。（具体可以参考阿里云 ESSD 云盘规格）

IO 层面，目前计划和 CPU 一样，希望能够利用 Cgroup 为多个低功耗运行实例构造一个抢占式动态的 IO 复用 Pool， 这样既能从系统层面控制低功耗实例 IO 请求对系统 IO 的影响，同时能最大限度的动态利用分配给每一个低功耗实例的 IO。目前实现并未涉及 Zizz IO 低功耗特殊实现，作为 Zizz IO 低功耗下一阶段的工作。

CSE Fn：基于 RSocket Broker 架构统一 FaaS 体系

前文所介绍的是针对应用启动时间所做的极速启动创新，下文我们再介绍一下针对 FaaS 场景，我们引入的 RSocket Broker 架构。

举个例子，娱乐圈的明星通常会将宝贵的时间投入到活动和演出中，商业洽淡以及各种琐事，通常交给职业经纪人去打理。可以说经纪人是明星与外界的桥梁，两者各司其职。而应用的核心价值通常在其业务领域，服务发现、负载均衡、加密、可观测、可追踪、安全和流控等通用基础服务通常不是应用的核心价值，然而它们又是必须的。假如将明星比做业务领域的话，那么经纪人就是 Broker。Broker 它将通用基础设施服务从应用中剥离，使得后者只专注业务领域，可以说，这个架构是目前最适合函数体系的新架构。

架构整体表述如图所示：

CSE Fn 架构里的所有组件（triggers, functions and proxies）相互间的网络通信经过处于枢纽位置的 Broker。

- Broker 负责服务发现（Service discovery）、路由（Routing）、负载均衡（LB）、追踪（Tracing）、流量调度（Traffic shifting）、流控（Circuit breaker）等应用基础设施服务；
- Trigger 组件（如 HSF gateway）负责接收外部请求产生事件，将网络协议转化成 RSocket，再转发给 Broker；
- Proxy 组件（如 Tair Proxy）负责代理中间件的服务，将 RSocket 协议翻译成中间件原始的协议；
- 用户编写的每一个函数（如 Fn1.v1）都运行在独立的容器内。它们被来自 Trigger 的事件触发，并可调用 Proxy 来使用中间件的服务。

Broker 的性能优势

Broker 作为网络中枢，对性能有极高要求。为此，CSE Fn 架构里的所有组件相互间的网络通信使用了 RSocket 协议：它是新一代、跨语言且基于 Reactive 编程模型的开源通讯框架，阿里巴巴（Reactive foundation 成员）为其主要贡献者之一。压测显示 1 台普通规格配置的 Broker 约可支持数万连接（每个函数 1 个连接）及数万QPS。

- 全异步无阻塞: RSocket 协议基于 Reactive 编程模型，全异步无阻塞。HSF、HTTP、Tair、MetaQ 等协议在 CSE Fn 中也适配成无阻塞的实现；
  payload 高效转发: RSocket 的网络 payload 分为 metadata 和 data（可以类比为 HTTP 的 header 和 body）两部分。Broker 在路由 Payload 时只解析较小 size 的 metadata，不解析 data。同时，Broker 转发 payload 时内存零拷贝；
- 单连接多路复用: 所有组件只跟每个 Broker 建立 1 个长连接，在 1 个连接中双向通信、多路复用——长连接免去每次调用的新建 TCP 连接的开销；单连接使得网络最优化（如 MTU 利用率提升，增加吞吐量）。

Broker 的能力

- 轻量方式使用中间件: 对于 Java 语言，使用中间件原本需要依赖重客户端，由此会带来启动慢、有状态、依赖冲突和升级等弊端；对于 Node.js 和 Python 语言，部分中间件没有客户端或非官方版，无法充分利用中间件的能力。通过 Broker 架构，CSE Fn 所有语言的函数都可以通过轻量的方式使用集团主流的中间件；
- 服务透明路由、负载均衡: 服务路由和负载均衡完全由 Broker 负责，函数进程无需关心；

- 追踪：所有函数请求自带分布式追踪透传功能，工程师无需关心，同时可以快速融入现有集团生态，简化了排查问题的成本；
- 流量调度：CSE Fn 自研了基于别名的流量调度规则支持，用户可使用多版本、蓝绿部署、金丝雀发布等部署策略——回滚秒级完成（不需要重新部署）、可在函数多个版本间按百分比精细化分配流量；
- 流控：函数自带打通阿里巴巴开源的 Sentinel（https://github.com/alibaba/Sentinel），用户仅需去 Sentinel 配置限流规则即可；
- 自动伸缩：基于并发指标的自动弹性伸缩，帮助用户基于系统的实际负载需求自动申请和回收资源，高效响应用户请求，做到 Serverless 的轻松运维体验。

成果

经过一年多的努力，以及合作伙伴、早期用户的不断尝试和反馈，CSE Fn 在 2019 年 双11 顺利通过大考。服务了包括淘宝导购、飞猪导购、ICBU、CBU 和 UC 等部门在内的几十个业务，涉及的函数数量已经初具规模。

零实例

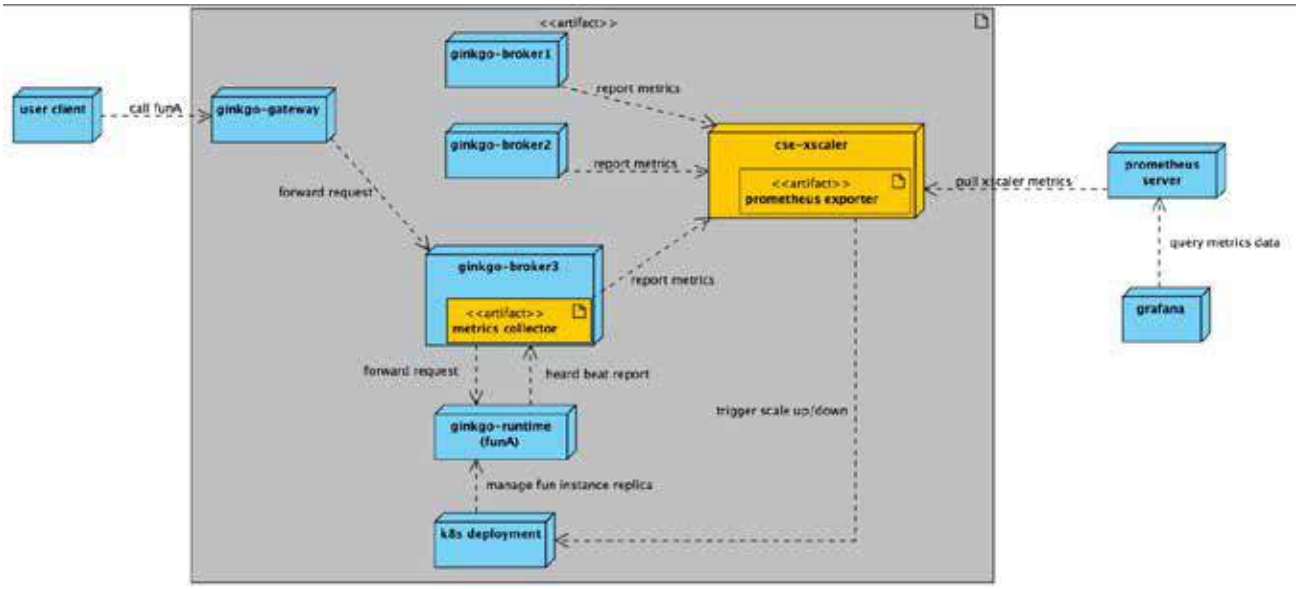
早期 Serverless 自动弹性只要做到用户体感上的这种体验（基于流量的自适应能力）就可以了，在底层基础设施能够保证用户所需要资源的基础上（当然底层也是需要提供足够的稳定性），多数用户是很高兴不再关心自己底下到底部署了多少这个服务器的，作为底层的基础设施，在用户的低流量期，“偷偷”保留好一个用户实例，以备用户不时之需使用即可。

随着技术进步以及工程师们追求的极致信念，价值的体现某种程度上也是在已有能力的基础上改进和构建出更为先进和合理的基础设施。另一方面，随着按需付费基础需求的不断涌现，真正闲时全数回收资源的能力更突显其重要性。

事实上，在一个理想运维调度模型下，在一个有限资源池内，合理利用缩容至零的能力，可以很好地高效利用资源，以下会以预发场景使用零实例的例子具体阐述这个具体应用。

设计介绍

基于 CSE Fn 自身在阿里巴巴落地的广泛场景和大量用户这一基础，CSE 团队很快就将 CSE Fn 的零实例能力提到了比较高的优先级。以下为结合 CSE Fn 自身架构的零实例架构：



注：图中 ginkgo 为 CSE Fn 的内部开发代号。

从以上架构图中可以看出，用户 Fn（图中的 funA）缩零和扩容 (0->1) 的过程都是一个完整的数据闭环，具体：

1.1->0 过程

- 1.用户 Fn 流量归零之后，CSE 的弹性模块会基于 ginkgo broker 上报的 metrics 数据，计算出当前用户 Fn 已无存在必要，可以进入缩零流程，在缩零静默期内（缩零静默期主要用于解决短时流量波动造成的不必要反复缩零拉起的过程），Fn 至少保留 1 个实例，期间任何流量进入都会触发退出缩零流程；
- 2.缩零静默期满后，CSE 的弹性模块会下发缩零指令，触发 K8s 底层 pod 资源全数释放。

2.0 ->1 过程

- 1.任何对缩零 Fn 的请求，仍然会发往 ginkgo broker，ginkgo broker 对 CSE 弹性模块的 metrics 上报会立刻反映出对 Fn 的“需求”；
- 2.CSE 的弹性模块基于最新的弹性数据，直接下发扩容指令（多数情况下，流量并没有突然出现并且暴涨很高，所以这里的 N 多数情况下是 1）。

从上面描述的扩缩过程中，我们可以看到这里结合了 ginkgo 自身架构特点几个核心组件的作用：

- 1.ginkgo-broker：CSE Fn 的调用流量入口，具备双重身份，既承担平时的指标收集职责，也肩负实例缩零后的流量承接。广义上说，这两个职责都是流量收集的范畴（只是处在一个 Fn 生命周期的不同阶段）；
- 2.cse-xscaler：CSE 弹性的核心决策组件，基于 metrics 计算 Fn 需要的实例数量，结合缩零静默期等配置，完成优雅缩零和实例拉起的重任。

应用场景

作为平台类产品，CSE 有众多的用户需要在预发布环境体验、验证和使用 Serverless 的功能。但本着艰(zhen)苦(de)朴(hen)素(qiong)的原则，CSE 在预发只有有限的物理资源，如果所有 Fn 实例均在上面运行，cpu、memory、eni 等资源都将很快成为瓶颈，造成新 pod 的调度直接 pending 卡住。

早期的解决方案是：CSE 维护一个暴力脚本，每天运行一次，清理 N 天前在预发环境创建的 Fn。这个方式简单粗暴，好处是确实有效，清理一次过后立马空出大量资源，但缺陷也比较明显：
1.没法规避误杀。因为清理条件简单，这个 N 无论怎么调整都没法规避误杀（干掉了 N 天之前但用户其实还是要用的 Fn），N 过大对清理几乎无效，N 过小会扩大误杀范围；
2.随着预发环境试用业务方的增多，每天一次的清理周期都已经显得捉襟见肘。

为了更好地解决预发资源的利用率问题，充分发挥预发公共资源池的错峰利用作用，开启 CSE 的零实例能力是一个非常不错的选择。为了降低预发用户的缩零抖动波动，目前，CSE 的预发环境设置了默认 6 小时的缩零静默期。这样在用户不再使用 Fn 的 6 小时之后，系统会自动全数回收 Fn 占据的物理资源，为其他玩家空出场地继续玩耍。

在一个集群内，我们做到了用 3 个 Node 支撑 400+ 函数的使用，其中的秘诀就在于零实例的应用。

零实例本身的建设也是一项系统工程，这里 CSE 团队也在结合自身的技术积累不断打磨和优化零实例的用户体验，后面我们会不断完善包括但不限于以下的方面：

1.提升零实例拉起的速度，整合 CSE Zizz 极速启动技术，逐步向极速启动方向靠拢；

2.多协议完备支持：除了目前在 CSE Fn 的运用场景，逐步发展支撑集团内的非 FaaS 应用场景；

3.探索零实例的更多应用场景，比如应用零实例到压测平台的压力机资源回收中等等。

富容器与轻容器

Docker 使得容器镜像成为软件及运行环境交付的事实标准，从而使 docker 容器成为一种事实标准的、轻量的、可以细粒度限制资源的沙箱。轻量的沙箱就意味着 docker 容器天生适合少量进程，那么在独占一个 pid namespace 的容器中，1 号进程自然就是 Dockerfile 或者启动容器的命令行中指定的进程。 然而，当年第一批吃螃蟹尝 docker 的人很快发现，没有 init system 的 docker 容器会带来两个令人无法忽视的问题：

- 僵尸进程
- 优雅退出（SIGTERM）

解决这两个问题的方法有很多，包括让容器内只有一个进程；容器内的 1 号进程负责 wait 所有子进程以避免出现僵尸进程，同时负责转发 SIGTERM 信号使得子进程都能够优雅退出；或者让 docker 支持 init system 等等。

阿里巴巴在容器化的过程中，通过 Pouch（https://github.com/alibaba/pouch） 的富容器提供了与 VM 极其相似的使用体验，这样开发人员只依赖过往的经验就能够顺利地迁移及运维自己的应用。得益于此，阿里巴巴的应用从 VM 向容器迁移，其实就是迁移到了 Pouch 的富容器中。然而 Docker 社区一直认为容器应当是轻量的，一个容器只应该负责一项具体的事情。

Each container should have only one concern. Decoupling applications into multiple containers makes it easier to scale horizontally and reuse containers. For instance, a web application stack might consist of three separate containers, each with its own unique image, to manage the web application, database, and an in-memory cache in a decoupled manner.

Kubernetes 则更是提供了一个用轻量容器编排多个进程/服务的标准。然而阿里巴巴应用的镜像基本上都是针对富容器模式的产物，可以看到运行的实例中跑着一大票进程，有业务进程、日志采集进程、监控采集进程、staragent 进程，林林总总。一个常见的做法是，业务应用使用某个囊括了 staragent 等一系列运维基础组件的镜像作为基础镜像，再 add 自己的程序后打包成最后的镜像用于发布。不可否认，在从 VM 向容器迁移时，这种做法使得我们快速享受到了镜像发布的便利：一方面可以保证运行环境的一致性；另一方面还能让我们基于 VM 的运维经验得以继续发挥。

SideCar

然而 CSE 的落地过程中就很快遇到了富容器模式的副作用。我们曾遇到 CSE Fn 业务方反馈，新扩容的实例在启动后 30-60s 内 cpu 飙高，超时错误骤增。排查后可以确认是运维组件 staragent 在启动后更新插件的行为抢占了 cpu，导致业务进程无法及时响应请求。

一般基础镜像中打包的运维组件版本都很老旧，不得不在启动时进行更新，如果实例规格较大，应用启动较慢，这好像也不是什么问题。但在 CSE Fn 场景下，应用启动速度快，实例规格小（比如 1 core），这种问题就被放大了，并且是不可接受的。

问题原因搞清楚了之后，解决思路也有很多，但我们很自然地选择了最云原生的那个：剥离运维组件到 sidecar 容器，利用容器的资源隔离能力来保证业务资源不被争抢。

在剥离运维 sidecar 的过程中，我们解决了如下问题：
1.资源隔离及 QoS：给 sidecar 分配合适的资源并保证与业务容器资源隔离；
2.CMDB 集成：将原本业务容器内汇报 cmdb 数据的逻辑剥离到 sidecar；
3.系统监控：将原本基于系统 metrics 日志的监控转换到 K8s 的 Node 上报采集；
4.日志采集：剥离日志采集 agent 到 sidecar，打通业务容器和 sidecar 的日志文件共享；
5.Web Terminal：将 Web Terminal 功能剥离到 sidecar。

价值和未来

在 Serverless 场景下，为了让实例根据负载变化自动扩缩，就需要应用尽可能快速地启动。影响应用启动速度的因素包括但不限于以下几点：

- 镜像分发
- 容器启动
- 进程启动

容器轻量化显然是云原生化过程中绕不开的一步，除了能够实实在在地提升应用启动速度以外，还有很多额外的好处：

- 解耦业务应用与运维组件
- 更好的资源隔离
- 更小的镜像（层少文件少）
- 镜像复用
    - 运维镜像因为能跨多个应用复用而不必重复分发
- 更快的分发速度（体积小）
    - 应用镜像因体积减小而能够显著提升分发效率
- 更快的启动速度（进程少）
    - 镜像瘦身能够提高容器启动速度
- 更简单可靠的运维管理（readiness/liveness probe，迁移等）
- 更高的集群整体资源利用率（容器资源需求小）

容器轻量化只有从应用开发侧提供轻量的镜像才能够实际应用，而应用侧在基础设施没有 ready 的情况下是不可能有任何变化的，但应用侧不去改变的话，基础设施就必须按照既有方式继续支持和适配，这看起来像是个死循环。CSE 努力推动应用侧往更轻量的镜像去发展，同时推动底层基础设施不断演进，继而收获容器轻量化的价值。

小结

Serverless 落地是一个极具挑战的系统工程，它涉及了非常广泛的技术，在基础设施层面我们从阿里巴巴的容器及调度团队到了大量极具价值的帮助；在系统和 JVM 层面，我们为了解决应用极速启动的挑战，在相关团队帮助下做了非常多的创新探索；在中间件和运维管控层面，我们在不断重新思考那些以前认为理所应当的方法。

阿里巴巴 双11 对性能、质量、稳定性方面严苛的要求，也使得我们在各个细节上都要做到精益求精，2019 年的 双11，CSE 顺利通过了第一次大考，我们明白这只是个开始，Serverless 能兑现的价值远超我们现在已经做到的，Serverless 所能带来的技术创新也必将远远比本文介绍的内容丰富，处在创新技术的浪潮之巅，我们都感到无比的幸运。

02

解密 双11 小程序云背后毫秒级伸缩的 Serverless 计算平台：函数计算

吴天龙，花名木吴，阿里云函数计算技术专家

自 2017 年第一批小程序上线以来，越来越多的移动端应用以小程序的形式呈现。小程序拥有触手可及、用完即走的优点，这大大减少了用户的使用负担，使小程序得到了广泛的传播。在阿里巴巴小程序也被广泛地应用在淘宝/支付宝/钉钉/高德等平台上，例如今年 双11 大家在淘宝/天猫上参加的活动，大部分都是通过小程序提供的。

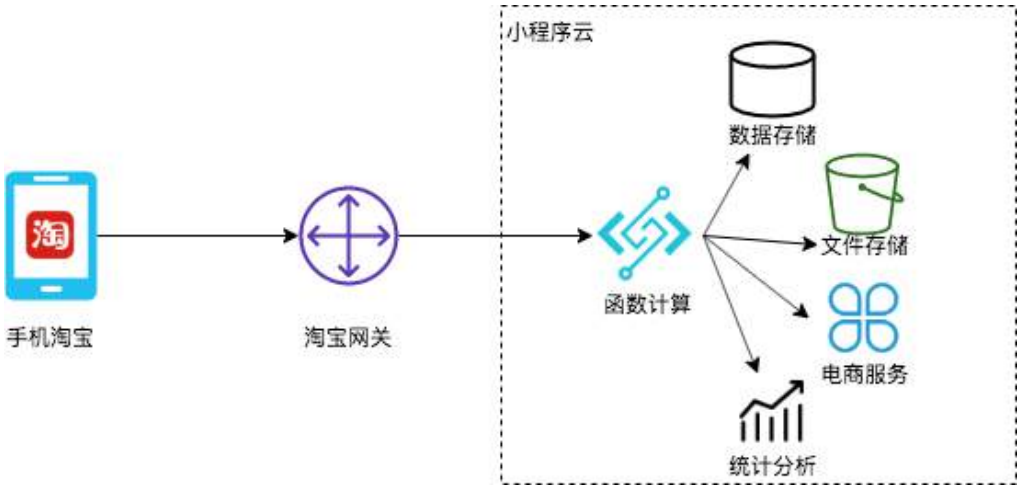
一个小程序可以分为客户端和服务端：客户端包括界面的展示和交互逻辑，服务端则包括数据的处理和分析。为了支撑大量的小程序，平台在服务端面临的挑战有：

1.大量的小程序是不活跃的，传统的至少一台服务器的方式会造成资源浪费
2.在活动高峰期小程序的调用量激增，要求服务端能够快速进行弹性伸缩

针对小程序场景，阿里云提供了完整的小程序解决方案：小程序云。资源的有效利用和弹性伸缩，是小程序云提供的核心能力之一，而这背后依托的，就是阿里云函数计算服务。函数计算是一个全托管 Serverless 计算服务，让开发者无需管理服务器等基础设施，只需编写和上传代码，就能够构建可靠、弹性、安全的服务。下面就以 双11 小程序场景为例，解析函数计算在弹性伸缩上的核心技术。

小程序架构

让我们先来看一下淘宝小程序的技术架构是什么样的：

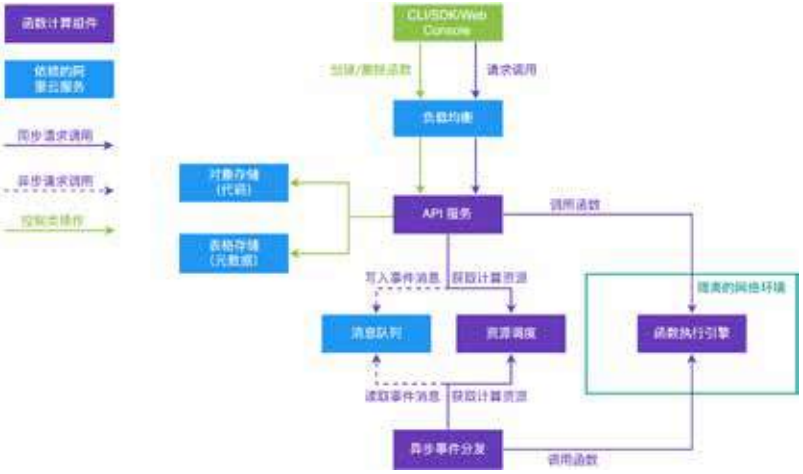


1.用户在手机淘宝点击店铺活动，就进入了小程序。界面及交互由小程序客户端提供
2.在用户参与活动过程中，需要向服务端请求或者发送数据时，由客户端发起函数调用
3.函数调用先经过淘宝接入网关，进行必要的鉴权认证，然后调用到小程序云
4.用户的函数代码执行在小程序云中，用户可以实现自定义的业务逻辑。利用小程序云提供的丰富的扩展能力，用户可以方便地构建完整的电商应用

- 数据存储：存储结构化的数据
- 文件存储：存储文本/图片/视频等文件
- 电商服务：获取用户信息/创建支付交易
- 统计分析：自动统计小程序的使用信息及用户分析，支撑商业决策

可以看到，函数是整个小程序的业务逻辑的核心，它将云端的基础能力组合串联起来，对客户端提供服务能力。如果函数能力成为瓶颈，将影响整个小程序的运行。在这样的架构下，要支撑大量的小程序，需要函数能够做到：一是随时在线以支持小程序即开即用，二是弹性伸缩以应对小程序访问突增。为了做到以上两点，让我们看一下函数计算的技术架构：

函数计算架构

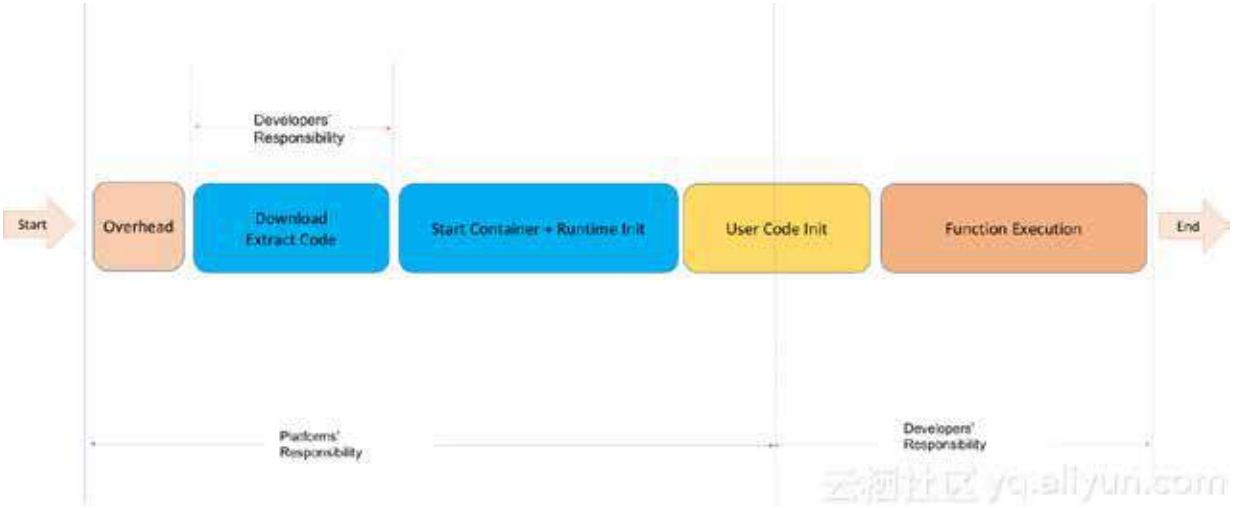


其中几个核心组件的功能如下：
1.API 服务：函数计算的网关，实现鉴权/流控等功能
2.资源调度：为函数调用分配管理计算资源，负责调度效率和性能
3.函数执行引擎：执行函数代码的环境，做到安全和隔离

基于这个架构，函数计算是如何解决上面提到的小程序平台的挑战呢？接下来我们逐一分析。

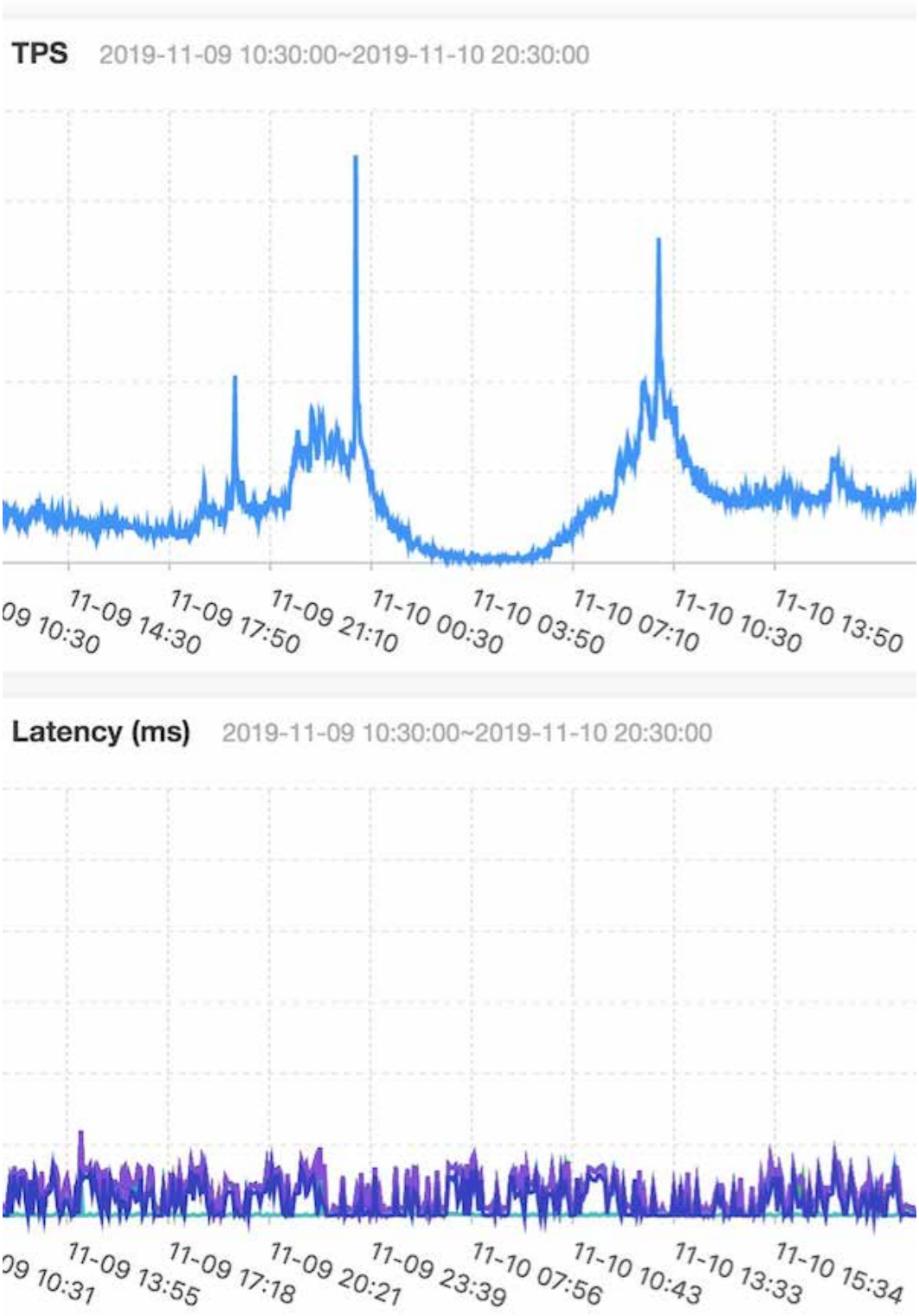
1. 冷启动

当用户创建函数上传代码时，函数计算只是将代码包保存到 OSS，并没有分配计算资源，因此函数计算可以支撑海量的小程序。当函数第一次被调用时，函数计算会分配计算资源、下载函数代码、加载并执行代码。这一过程称为冷启动，函数计算通过大量的优化，将系统侧的冷启动时间优化到 200ms 以内。因此即使是冷的小程序，在初次调用时也能够做到快速的即开即用。



2. 弹性伸缩

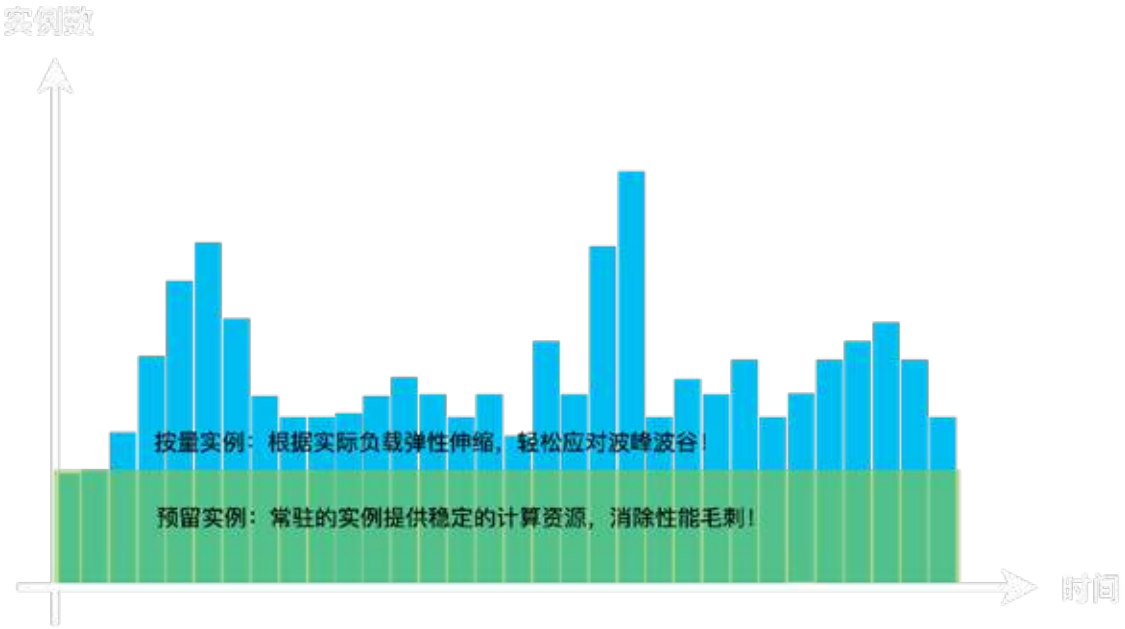
当小程序持续调用过程中，负载逐渐上升或者突然升高的情况下，函数计算是如何应对的？函数计算的“资源调度”模块，会精确管理每个实例的状态，当请求到来时，它首先检查是否有空闲的实例可以服务，如果没有请求就会进入等待队列，当有空闲的实例释放出来时，请求就能够被及时处理。同时，调度器还会在后台创建新的实例，当新的实例准备好后，也能够服务请求。在这种策略下，能够做到在负载以 2 倍的速度增长情况下，请求的 P95 延时是稳定的。优化细节可以参考我们的博客文章。https://yq.aliyun.com/articles/625148



上面是某个淘宝小程序的调用量和延时监控数据。可以看到，在整点活动时，TPS 出现了瞬间的高峰，但是 P95 延时却没有明显的波动。这是因为函数计算在请求高峰来临时，能够快速地弹性伸缩创建新的实例，同时利用已有资源做一定的缓冲，使得整个攀升的过程能够比较平滑。

3. 预留实例

对于一些“秒杀”的场景，要求瞬间提供大量的计算资源。此时靠实时的弹性伸缩是不够的：一是冷启动的时间即使是　200ms，对于秒杀场景也太慢了；二是底层的计算资源在扩容时也会有流控。针对这种场景，函数计算提供了预留实例的功能。使用预留实例，用户可以为一些可预测的活动提前预留好资源，彻底消除冷启动。



和传统的基于服务器的做法不同，用户不需要按峰值来预留资源，而是可以结合预留实例和按量实例的混合模式：请求先被预留实例处理，当预留实例用满时，会自动弹性伸缩出更多的按量实例来处理请求。由于有一定的资源基础，结合调度优化，按量实例的冷启动所产生的影响就被大大减小了。这就是利用函数计算的弹性伸缩能力，在性能和成本之间达到很好的平衡。

总结

小程序是轻量级的快速迭代的移动应用，对小程序开发者的开发效率有很高的要求。小程序上线后随着访问量的增加或者活动期间的访问突增，对后端服务的稳定和弹性也是一个很大的考验。函数计算上传代码即可运行，极大地提高了后端服务的开发效率；混合模式的弹性伸缩，轻松应对负载变化。这些特点使得函数计算成为支撑小程序平台的很好的选择。

参考

函数计算与冷启动的那些事
函数计算性能福利篇(一) —— 系统冷启动优化

01

双11 背后的全链路可观测性：阿里巴巴鹰眼在“云原生时代”的全面升级

周小帆，花名承嗣，阿里云中间件技术部高级技术专家
王华锋，花名水彧，阿里云中间件技术部技术专家
徐彤，花名绍宽，阿里云中间件技术部技术专家
夏明，花名涯海，阿里云中间件技术部技术专家

云原生与可观测性

在刚刚过去的 2019 年 双11，我们再次见证了一个技术奇迹：这一次，我们花了一整年的时间，让阿里巴巴的核心电商业务全面上云，并且利用阿里云的技术基础设施顶住了 54 万笔/秒的零点交易峰值；我们的研发、运维模式，也正式步入了云原生时代。

作为一支深耕多年链路追踪技术 (Tracing) 与性能管理服务 (APM) 的团队，阿里巴巴中间件鹰眼团队的工程师们见证了阿里巴巴基础架构的多次升级，每一次的架构升级对于系统可观测性能力 (Observability) 都会带来巨大挑战，而这次的“云原生”升级，给我们带来的新挑战又是什么？

云原生所倡导的新范式，给传统的研发和运维模式带来巨大冲击：微服务、DevOps 等理念让研发效率变得更高，但带来的却是海量微服务的问题排查、故障定位的难度变得更大；容器化、Kubernetes 等容器编排技术的逐渐成熟让规模化软件交付变得容易，但带来的挑战是如何更精准地评估容量、调度资源，确保成本与稳定性的最好平衡。今年阿里巴巴所探索的 Serverless、Service Mesh 等新技术，未来将彻底地从用户手中接管运维中间件以及 IaaS 层的工作，对于基础设施的自动化程度则是一个更加巨大的挑战。

基础设施的自动化（Automation）是云原生的红利能够被充分释放的前提，而可观测性是一切自动化决策的基石。

如果每个接口的执行效率、成败与否都能被精准统计、每一个用户请求的来龙去脉都能被完整追溯、应用之间以及应用与底层资源的依赖关系能被自动梳理，那我们就能基于这些信息自动判断业务的异常根因在哪？是否需要对影响业务的底层资源做迁移、扩容或是摘除？我们就能根据 双11 的峰值自动推算出每一个应用所需准备资源是否充分且不浪费。

可观测性≠监控

许多人会问，“可观测性”是否就是“监控”换个说法，而业界对这两件事的定义其实大相径庭。

不同于“监控”，监控更加注重问题的发现与预警，而“可观测性”的终极目标是为一个复杂分布式系统所发生的一切给出合理解释。监控更注重软件交付过程中以及交付后（Day 1 & Day 2），也就是我们常说的“事中与事后”，而“可观测性”则要为全研发与运维的生命周期负责。

回到“可观测性”本身，依旧是由老生常谈的“链路(Tracing)”、“指标(Metric)”和“日志(Logging)”构成，单独拉出来看都是非常成熟的技术领域。只不过这三样东西与云基础设施的如何整合？它们之间如何更好地关联、融合在一起？以及他们如何更好地和云时代的在线业务做结合？是我们团队这一两年来努力探索的方向。

我们今年做了什么

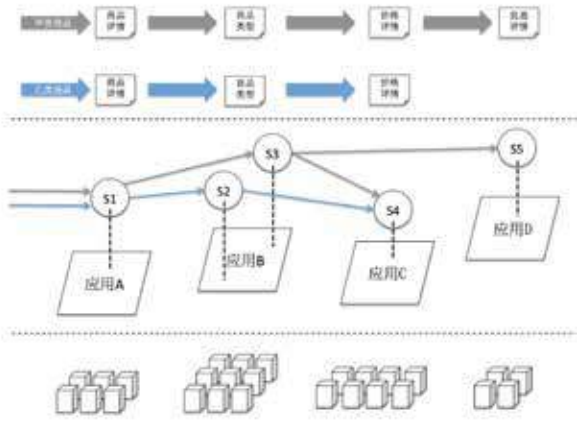
今年的 双11，鹰眼团队的工程师们在四个新方向的技术探索，为集团业务全面上云、双11 的自动化备战与全局稳定性提供了强有力的保障：

面向场景化的业务可观测性

随着阿里巴巴电商业态不断的复杂与多元化，大促备战也不断趋向于精细化与场景化。

以往的备战方式，是各个微服务系统的负责人根据所负责系统的自身以及上下游情况，各自为战。这样分而治之的做法虽然足够高效，却难免有疏漏，根本原因在于中台应用与实际业务场景的错位关系。以交易系统为例，一个交易系统会同时承载天猫、盒马、大麦、飞猪等多种类型的业务，而每种业务的预期调用量、下游依赖路径等均不相同，作为交易系统的负责人，很难梳理清楚每种业务的上下游细节逻辑对自身系统的影响。

今年鹰眼团队推出了一种场景化链路的能力，结合业务元数据字典，通过无侵入自动打标的手段实现流量染色，将实际的流量业务化，打通了业务与下游各中间件的数据，从以往的以应用为中心的视图，转变成了以业务场景为中心，也因此更加贴近于真实的大促模型。



如上图所示，这是一个查询商品的案例，四个系统 A、B、C、D 分别提供“商品详情”、“商品类型”、“价格详情”和“优惠详情”的查询能力。入口应用 A 提供了一个商品查询的接口 S1，通过鹰眼，我们可以很快地发现应用 B、C、D 属于应用 A 的依赖，同时也是接口 S1 的下游，对于系统稳定性的治理而言，有这样一份链路数据已经足够。

但其实这样的视角并不具备业务的可观测性，因为在这样的一个依赖结构中包含着两种业务场景，而这两种场景所对应的链路也是完全不同的，甲类商品所对应的链路是 A->B->C-D，而乙类商品对应的链路是A->B->C。假设日常态这两类商品的占比是 1:1，而大促态的占比是 1:9，那么仅从系统的角度或者业务的角度去梳理链路，是无法得到一个合理的流量预估模型的。

所以，如果我们能在系统层通过打标的方式把两种流量染色，就能很方便梳理出两种业务场景所对应的的链路，这样一份更加精细化的视角对于保证业务的稳定性、以及更加合理的依赖梳理和限流降级策略的配置显得尤为重要。

这样的业务场景化能力在今年的 双11 备战中发挥了巨大的价值，很多业务系统都基于这样的能力梳理出了自己核心的业务链路，备战更加从容且不会有遗漏；同时，一系列的服务治理工具，在鹰眼的赋能下，进行了全面的场景化升级，例如针对场景化的流量录制和回放，场景化的故障演练工具，场景化的精准测试回归等等；配合这些更加贴合业务场景的服务治理工具，帮助整个 双11 备战的可观测性颗粒度走进了“高清时代”。

基于可观测性数据的智能根因定位

云原生时代，随着微服务等技术的引入，业务规模的增长，应用的实例数规模不断增长，核心业务的依赖也变得愈加复杂。一方面我们享受着开发效率的指数提升的红利，同时也在承受着故障定位成本居高不下的痛苦。特别是当业务出现问题的时候，如何快速发现问题和止血变得非常困难。鹰眼团队作为集团内应用性能的“守护神”，如何帮助用户快速完成故障定位成为今年的新挑战。

要完成故障定位，首先要回答，什么是你认为的故障？这背后需要运维人员对业务深层次的理解，很多维护人员喜欢使用穷举式的手段配上所有可观测性的指标，各种告警加上，显得有“安全感”，实际上当故障来临时，满屏出现指标异常、不断增加的告警短信，这样的“可观测性”看上去功能强大，实际效果却适得其反。

团队对集团内的历年故障做了一次仔细梳理，集团内的核心应用通常有四类故障(非业务自身逻辑问题)，资源类、流量类、时延类、错误类，再往下细分：

1.资源类： 比如 cpu、load、mem、线程数、连接池；
2.流量类：业务流量跌零 OR 不正常大幅度上涨下跌，中间件流量如消息提供的服务跌零等；
3.时延类：系统提供的服务 OR 系统依赖的服务，时延突然大幅度飙高了，基本都是系统有问题的前兆；
4.错误类：服务返回的错误的总数量，系统提供服务 OR 依赖服务的成功率。

有了上面这些故障分类作为抓手后，后面要做的就是“顺藤摸瓜”，可惜随着业务的复杂性，这根“藤”也来越长，以时延突增这个故障为例，这背后就隐藏着很多可能的根因，可能的原因多种多样：有可能是上游业务促销导致请求量突增导致，有可能是应用自身频繁 GC 导致应用整体变慢，也有可能是下游数据库负载过大导致响应变慢导致，还有数不胜数的其他各种原因。鹰眼以前仅仅提供了这些指标信息，维护人员光看单条调用链数据，鼠标就要滚上好几番才能看完一条完整的 tracing 数据，更别说跨多个系统之间来回切换排查问题，效率也就无从谈起。

回到故障定位的本质就是一个不断排查、否定、再排查的过程，是一个“排除掉所有的不可能，剩下的就是真相”的过程。仔细想想可枚举的可能+可循环迭代的过程，这个不就是计算机最擅长的处理动作吗？故障定位智能化项目在这样的背景下诞生了。

提起智能化，很多人第一反应是把算法关联在一起，把算法过度妖魔化。其实了解机器学习的同学应该都知道：数据质量排第一，模型排第二，最后才是算法。数据采集的可靠性、完整性与领域模型建模才是核心竞争力。只有把数据化这条路走准确后，才有可能走智能化。

故障定位智能化的演进路线也是按照上面的思路来逐步完成的，但在这之前我们先得保障数据的质量：得益于鹰眼团队在大数据处理上深耕多年，数据的可靠性已经能得到非常高质量的保障，否则出现故障还得先怀疑是不是自己指标的问题。接下来就是数据的完备性和诊断模型的建模，这两部分是智能化诊断的基石，决定了故障定位的层级。同时这两部分也是相辅相成的，通过诊断模型的构建可以对可观测性指标查漏补缺，通过补齐指标也可以增加诊断模型的深度。

主要通过三方面结合来不断的完善这两部分：第一，历史故障推演，历史故障相当于已经知道标准答案的考卷，通过部分历史故障+人工经验来构建最初的诊断模型，然后迭代推演其余的历史故障，但是这一步出来的模型容易出现过拟合现象；第二，利用混沌工程模拟常见的异常，不断修正模型；第三，线上人为打标的方式，来继续补齐可观测性指标、修正诊断模型。

经过了以上三个阶段之后，这块基石基本建立完成了。接下来就要解决效率问题，从上面几步迭代出来的模型其实并不是最高效的，因为人的经验和思维是线性思维，团队内部针对现有模型做了两方面的工作：边缘诊断和智能剪枝。将定位的过程部分下沉到各个代理节点，对于一些可能对系统造成影响的现象自动保存事发现场关键信息同时上报关键事件，诊断系统自动根据各个事件权重进行定位路径智能调整。

智能根因定位上线后，累计帮助数千个应用完成故障根因定位，并取得了很高的客户满意度，基于根因定位结论为抓手，可观测性为基石，基础设施的自动化能力会得到大大提升。今年的 双11 大促备战期间，有了这样的快速故障定位功能，为应用稳定性负责人提供了更加自动化的手段。我们也相信在云原生时代，企业应用追求的运行的质量、成本、效率动态平衡不再是遥不可及，未来可期！

最后一公里问题定位能力

什么是“最后一公里”的问题定位？“最后一公里”的问题有哪些特点，为什么不是“最后一百米”、“最后一米”？

首先，我们来对齐一个概念，什么是“最后一公里”？在日常生活中，它具备以下特点：

- 走路有点远，坐车又太近，不近不远的距离很难受；
- 最后一公里的路况非常复杂，可能是宽阔大道，也可能是崎岖小路，甚至是宛如迷宫的室内路程（这点外卖小哥应该体会最深）。

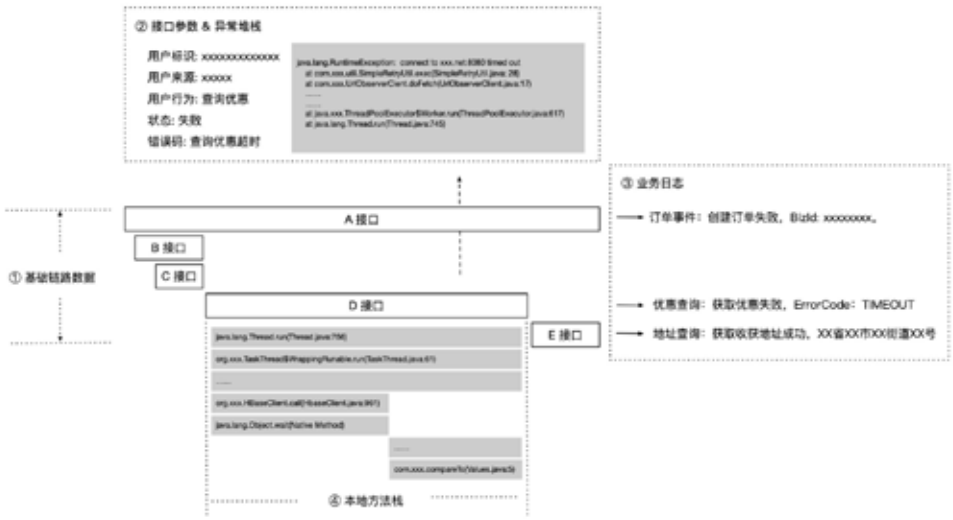
那么分布式问题诊断领域的最后一公里又是指什么呢，它又具备哪些特征？

- 在诊断流程上，此时已经离根因不会太远，基本是定位到了具体的应用、服务或节点，但是又无法
- 确定具体的异常代码片段；
- 能够定位根因的数据类型比较丰富，可能是内存占用分析，也可能是 CPU 占用分析，还可能是特

通过上面的分析，我们现在已经对最后一公里的概念有了一些共识。下面，我们就来详细的介绍，如何实现最后一公里的问题定位？

首先，我们需要一种方法，可以准确的到达最后一公里的起点，也就是问题根因所在的应用、服务或是机器节点。这样可以避免根源上的无效分析，就好像送外卖接错了订单。那么，如何在错综复杂的链路中，准确的定界根因范围？这里，我们需要使用 APM 领域较为常用的链路追踪（Tracing）的能力。通过链路追踪能够准确的识别、分析异常的应用、服务或机器，为我们最后一公里的定位指明方向。

然后，我们通过在链路数据上关联更多的细节信息，例如本地方法栈、业务日志、机器状态、SQL 参数等，从而实现最后一公里的问题定位，如下图所示：



单链路诊断示意图

- 核心接口埋点： 通过在接口执行前后插桩埋点，记录的基础链路信息，包括 TraceId、RpcId（SpanId）、时间、状态、IP、接口名称等。上述信息可以还原最基础的链路形态；
- 自动关联数据： 在调用生命周期内，可以自动记录的关联信息，包括 SQL、请求出入参数、异常堆栈等。此类信息不影响链路形态，但却是某些场景下，精准定位问题的必要条件；
- 主动关联数据： 在调用生命周期内，需要人为主动记录的关联数据，通常是业务数据，比如业务日志、业务标识等。由于业务数据是非常个性化的，无法统一配置，但与链路数据主动关联后，可以大幅提升业务问题诊断效率；
- 本地方法栈： 由于性能与成本限制，无法对所有方法添加链路埋点。此时，我们可以通过方法采样或在线插桩等手段实现精准的本地慢方法定位。

通过最后一公里的问题定位，能够在日常和大促备战态深度排查系统隐患，快速定位根因，下面举两个实际的应用案例：

- 某应用在整体流量峰值时出现偶发性的 RPC 调用超时，通过分析自动记录的本地方法栈快照，发现实际耗时都是消耗在日志输出语句上，原因是 LogBack 1.2.x 以下的版本在高并发同步调用场景容易出现“热锁”，通过升级版本或调整为异步日志输出就彻底解决了该问题；

- 某用户反馈订单异常，业务同学首先通过该用户的 UserId 检索出下单入口的业务日志，然后根据该日志中关联的链路标识 TraceId 将下游依赖的所有业务流程、状态与事件按实际调用顺序进行排列，快速定位了订单异常的原因（UID 无法自动透传到下游所有链路，但 TraceId 可以）。

监控告警往往只能反映问题的表象，最终问题的根因还需要深入到源码中去寻找答案。鹰眼今年在诊断数据的“精细采样”上取得了比较大的突破，在控制成本不膨胀的前提下，大幅提升了最后一公里定位所需数据的精细度与含金量。在整个双11 漫长的备战期中，帮助用户排除了一个又一个的系统风险源头，从而保障了大促当天的“丝般顺滑”。

全面拥抱云原生开源技术

过去的一年，鹰眼团队拥抱开源技术，对于业界主流的可观测性技术框架做了全面集成。我们在阿里云上发布了链路追踪（Tracing Analysis）服务，兼容 Jaeger(OpenTracing)、Zipkin、Skywalking 等主流的开源 Tracing 框架，已经使用了这些框架的程序，可以不用修改一行代码，只需要修改数据上报地址的配置文件，就能够以比开源自建低出许多的成本获得比开源 Tracing 产品强大许多的链路数据分析能力。

鹰眼团队同时也发布了全托管版的 Prometheus 服务解决了开源版本部署资源占用过大、监控节点数过多时的写入性能问题，对于长范围、多维度查询时查询速度过慢的问题也做了优化。优化后的 Prometheus 托管集群在阿里巴巴内全面支持了 Service Mesh 的监控以及几个重量级的阿里云客户，我们也将许多的优化点反哺回了社区。同样，托管版的 Prometheus 兼容开源版本，在阿里云的容器服务上也可以做到一键迁移到托管版。

可观测性与稳定性密不可分，鹰眼的工程师们今年将这些年来可观测性、稳定性建设等相关一系列的文章、工具做了整理，收录在了 Github 上，也欢迎大家一起来进行共建。

02

Kubernetes 时代的安全软件供应链

汤志敏，阿里云容器服务高级技术专家
汪圣平，花名木樵，阿里云云平台安全高级安全专家

“没有集装箱，就不会有全球化”。在软件行业里，Docker 和 Kubernetes 也扮演了类似的角色，加速了软件行业的社会化分工和交付运维的效率。2013 年，Docker 公司提出了容器应用打包规范 Docker Image，帮助开发者将应用和依赖打包到一个可移植的镜像里。2015 年，Google 将 Kubernetes 捐献给 CNCF，进一步普及了大规模容器编排调度的标准。Kubernetes 以一种声明式的容器编排与管理体系，屏蔽了底层基础架构的差异，让软件交付变得越来越标准化。随着 K8s 为代表的云原生技术的大规模运用，越来越多的容器化应用被分发到 IDC、公共云、边缘等全球各地。

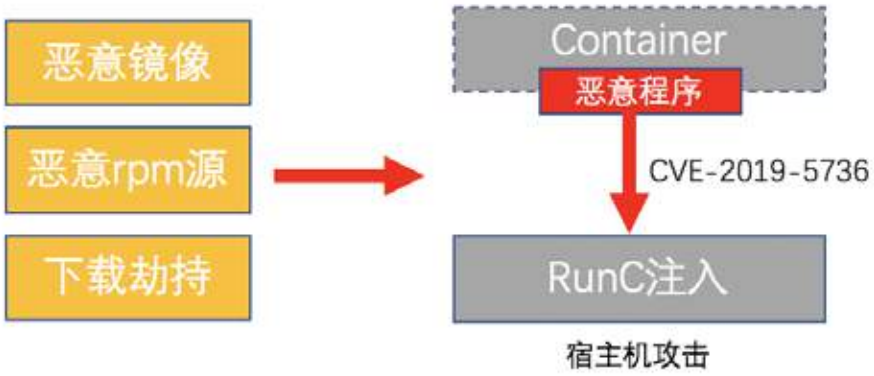
在 2019 年，阿里云容器镜像服务 ACR 的月镜像下载量超过了 3 亿次。同年 10 月，阿里云云市场的容器镜像类目发布，越来越多的企业将其软件以容器的方式进行上架和销售。11 月，天猫 双11 的所有核心系统 100% 上云。容器镜像服务 ACR 除了支持 双11 的内部镜像托管以外，也将内部的能力在云上透出，支持更多的 双11 生态公司。接下来我们看下如何保证容器和 Kubernetes 下的软件供应链安全，并先熟悉下软件供应链的常见攻击场景。

软件供应链攻击面和典型攻击场景

软件供应链通常包括三个阶段-软件研发阶段，软件交付阶段和软件使用阶段，在不同阶段的攻击面如下：

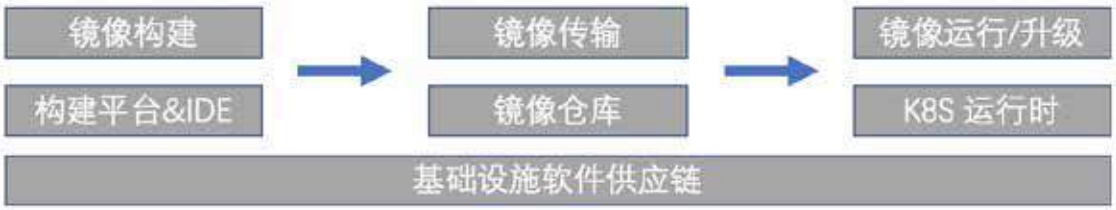
历史上著名的 APPLE Xcode IDE 工具攻击则是发生在软件研发阶段的攻击，攻击者通过向 Xcode 中注入恶意后门，在非官方网站提供下载，所有使用此 Xcode 的开发者编译出来的 APP 将被感染后门。同样著名的还有美国的 XX 门事件，亦是在大量的软件中植入后门程序，进行数据获取等恶意操作。

Kubernetes 中的软件供应链攻击面也包括在以上范围之中，以软件使用阶段为例，今年 RunC 漏洞 CVE-2019-5736，漏洞本身与 RunC 的运行设计原理相关，Container 之外的动态编译 Runc 被触发运行时会引用 Conainer 内部的动态库造成 RunC 自身被恶意注入从而运行恶意程序，攻击者只需要在一个 Container 镜像中放入恶意的动态库和恶意程序，诱发受攻击者恶意下载运行进行模糊攻击，一旦受攻击者的 Container 环境符合攻击条件，既可完成攻击。



同样的攻击面还存在于 Kubernetes 自身服务组件之中，前段时间爆出的 HTTP2 CVE-2019-9512、CVE-2019-9514 漏洞就是一个非常好的软件研发阶段脆弱性的例子，漏洞存在于 GO 语言的基础 LIB 库中，任何依赖 GO 版本（<1.2.9）所编译的 KubernetesHTTP2 服务组件都受此漏洞影响，因此当时此漏洞影响了 K8s 全系列版本，攻击者可以远程 DOS Kubernetes API Server。同时在 Kubernetes 组件的整个交付过程中也存在着攻击面，相关组件存在被恶意替换以及植入的可能性。

不同于传统的软件供应链，镜像作为统一交付的标准如在容器场景下被大规模应用，因此关注镜像的使用周期可以帮助攻击者更好的设计攻击路径。



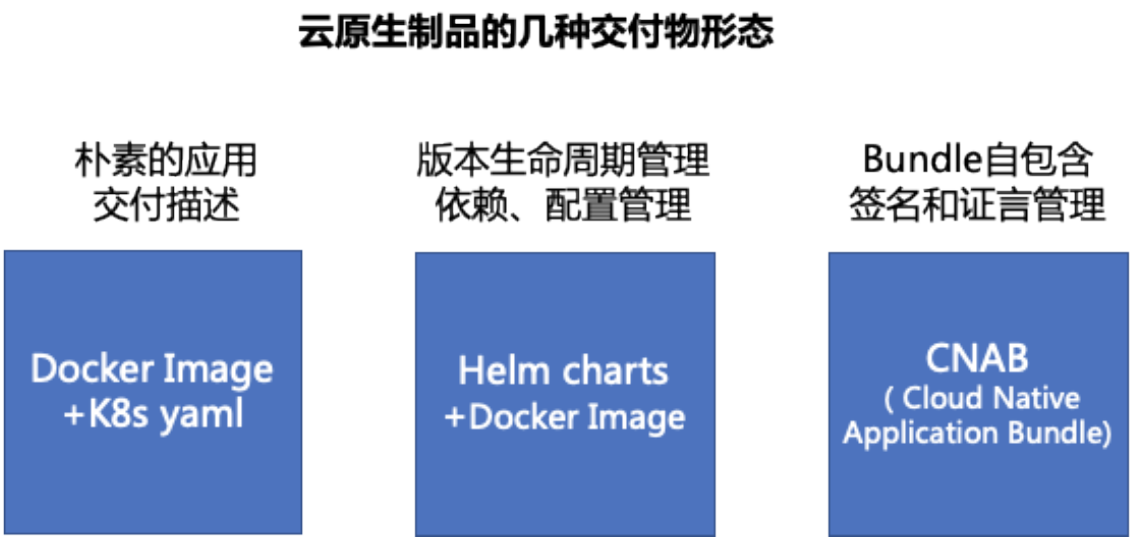
攻击者可以在镜像生命周期的任何一个阶段对镜像进行污染，包括对构建文件的篡改，对构建平台的后门植入，对传输过程中的劫持替换和修改，对镜像仓库的攻击以替换镜像文件，对镜像运行下载和升级的劫持攻击等。整个攻击过程可以借助云化场景中相关的各种依赖如 Kubernetes 组件漏洞，仓库的漏洞，甚至基础设施底层漏洞。对于防御者来说，对于镜像的整个生命周期的安全保障，是容器场景中攻击防范的重中之重。

云原生时代的应用交付标准演进（从 Image 到 Artifacts）

在云原生时代之前，应用交付的方式比较多样化，比如脚本、RPM 等等。而在云原生时代，为了屏蔽异构环境的差异，提供统一的部署抽象，大家对应用交付标准的统一也迫切起来。

Helm: Helm 是 Kubernetes 的包管理工具，它提出了 Chart 这个概念。
1.一个 Chart 描述了一个部署应用的多个 Kubernetes 资源的 YAML 文件，包括文档、配置项、版本等信息；
2.提供 Chart 级别的版本管理、升级和回滚能力。

CNAB：CNAB 是 Docker 和微软在 2018 年底联合推出平台无关的 Cloud Native Application Bundle 规范。相比于 Helm，有额外几个定义：
1. 在 thick 模式时，CNAB 的 bundle 可以包含依赖的镜像二进制，从而不需要额外去镜像仓库下载。作为一个整体打包。
2.CNAB 定义了扩展的安全标准，定义了 bundle 的签名（基于 TUF ）和来源证明（基于 In-Toto ）描述；
3.CNAB 的部署是平台无关性，可以部署在 K8s 之上，也可以通过 terraform 等方式来部署。
CNAB 的这些特性，可以在可信软件分发商与消费者之间进行跨平台（包括云和本地 PC ）的应用打包和分发。

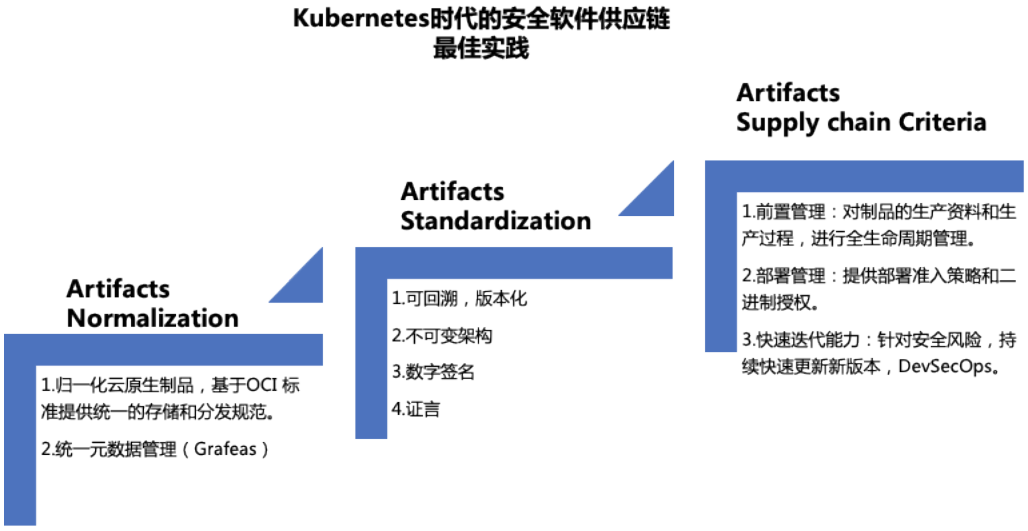


OCI Artifacts：2019 年 9 月，开放容器标准组织（OCI）在 OCI 分发标准之上，为了支持更多的分发格式，推出了 OCI Artifacts 项目来定义云原生制品的规范。我们可以通过扩展 media-type 来定义一种新的 Artifacts 规范，并通过标准的镜像仓库来统一管理。

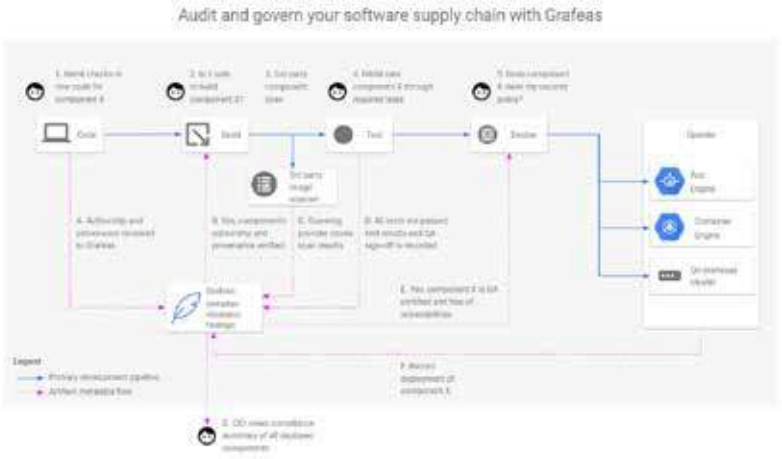
Kubernetes 时代的安全软件供应链

在之前章节也提到过，相对于传统软件的安全软件供应链管理，容器和 Kubernetes 的引入使得：
1.发布和迭代更加频繁，容器的易变性也使得安全风险稍纵即逝；
2.更多的不可控三方依赖，一旦一个底层基础镜像有了安全漏洞，会向病毒一样传递到上层；
3.更大范围的全球快速分发，在分发过程中的攻击也会使得在末端执行的时造成大规模安全风险；

在传统的软件安全和安全准则之上，我们可以结合一些最佳实践沉淀一个新的端到端安全软件供应链：



我们来看一些和安全软件供应链相关的社区技术进展：
Grafeas：2017 年 10 月，Google 联合 JFrog、IBM 等公司推出了 Grafeas。Grafeas（希腊语中的" scribe"）旨在定义统一的方式来审核和管理现代软件供应链，提供云原生制品的元数据管理能力。可以使用 Grafeas API 来存储，查询和检索有关各种软件组件的综合元数据，包括合规和风险状态。



In-toto:　In-toto 提供了一个框架或策略引擎来保护软件供应链的完整性。通过验证链中的每个任务是否按计划执行（仅由授权人员执行）以及产品在运输过程中未被篡改来做到这一点。In-toto 要求项目所有者创建布局(Layout)。

布局列出了软件供应链的步骤（Step）顺序，以及授权执行这些步骤的工作人员。当工作人员执行跨步操作时，将收集有关所使用的命令和相关文件的信息，并将其存储在链接(Link)元数据文件中。通过在完整的供应链中定义每个 Step，并对 Step 进行验证，可以充分完整的完整整个供应链的安全。

Kritis：为强化 Kubernetes 的安全性，Google 引入了二进制授权（Binary Authorization），确保使用者只能将受信任的工作负责部署到 Kubernetes 中。二进制授权可以基于 Kubernetes 的 Admission Controller 来插入部署准入检测，让只有授权后的镜像在环境中运作。下图为一个策略示例：

Kritis: ImageSecurityPolicy example

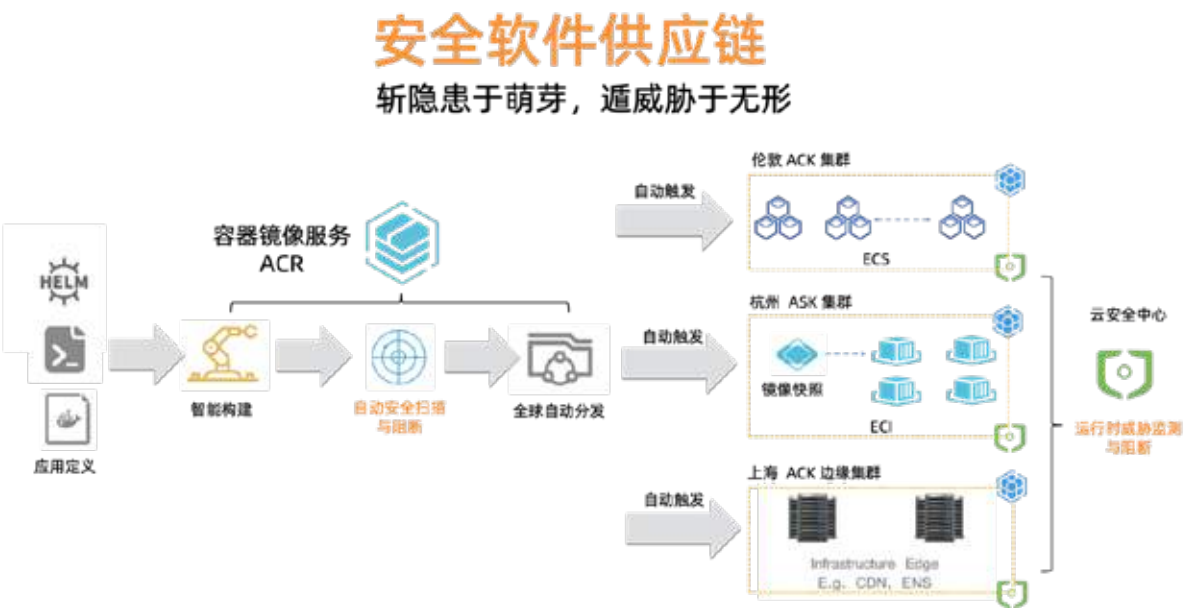
```
apiVersion: kritis.grafeas.io/v1beta1
kind: ImageSecurityPolicy
metadata:
 name: my-isp
spec:
 imageWhitelist:
  - gcr.io/kritis-int-test/nginx-digest-whitelist:latest
  - gcr.io/kritis-int-test/nginx-digest-whitelist\
@sha256:56e0af16f4a9d2401d3f55bc8d214d519f070b5317512c87568603f315a8be72
 packageVulnerabilityRequirements:
   maximumSeverity: HIGH # BLOCKALL|LOW|MEDIUM|HIGH|CRITICAL
   whitelistCVEs:
     - providers/goog-vulnz/notes/CVE-2017-1000082
     - providers/goog-vulnz/notes/CVE-2017-1000081
```

同时对于在安全软件供应链中占比很大的第三方软件需要有完善的基线机制和模糊安全测试机制来保障分发过程中的安全风险，避免带已知漏洞上线，阿里云正在与社区积极贡献帮助演进一些开源的工具链。

关于基础镜像优化、安全扫描、数字签名等领域也有非常多的工具和开源产品，在此不一一介绍。

云端的安全软件供应链最佳安全实践

在阿里云上，我们可以便捷地基于容器服务 ACK、容器镜像服务 ACR、云安全中心打造一个完整的安全软件供应链。安全软件供应链全链路以云原生应用托管为始，以云原生应用分发为终，全链路可观测、可追踪、可自主设置。可以帮助安全需求高、业务多地域大规模部署的企业级客户，实现一次应用变更，全球化多场景自动交付，极大提升云原生应用交付的效率及安全性。



在云原生应用的安全托管阶段，容器镜像服务 ACR 支持容器镜像、Helm Chart 等云原生资产的直接上传托管；也支持从源代码（Github、Bitbucket、阿里云 Code、GitLab 来源）智能构建成容器镜像。在安全软件供应链中，支持自动静态安全扫描并自定义配置安全阻断策略。一旦识别到静态应用中存在高危漏洞后，可自动阻断后续部署链路，通知客户失败的事件及相关漏洞报告。客户可基于漏洞报告中的修复建议，一键更新优化构建成新的镜像版本，再次触发自动安全扫描。

在云原生应用的分发阶段，当安全漏洞扫描完成且应用无漏洞，应用将被自动同步分发至全球多地域。由于使用了基于分层的调度、公网链路优化以及免公网入口开启的优化，云原生应用的全球同步效率相比本地下载后再上传提升了 7 倍。云原生应用同步到全球多地域后，可以自动触发多场景的应用重新部署，支持在 ACK、ASK、ACK@Edge 集群中应用自动更新。针对集群内大规模节点分发场景，可以实现基于镜像快照的秒级分发，支持 3 秒 500 Pod 的镜像获取，实现业务在弹性场景下的极速更新。

在云原生应用运行阶段，可实现基于云安全中心的应用运行时威胁检测与阻断，实时保障每个应用 Pod 的安全运行。云安全中心基于云原生的部署能力，实现威胁的数据自动化采集、识别、分析、响应、处置和统一的安全管控。利用多日志关联和上下文分析方案，实时检测命令执行、代码执行、SQL 注入、数据泄露等风险，覆盖业务漏洞入侵场景。结合 K8s 日志和云平台操作日志实时进行行为审计和风险识别，实现容器服务和编排平台存在的容器逃逸、AK 泄露、未授权访问风险。

03

安全容器：开启云原生沙箱技术的未来

王旭，花名循环，蚂蚁金服资深技术专家
刘奖，阿里云基础软件部资深技术专家
秦承刚，花名承刚，蚂蚁金服资深技术专家

前不久，我们的同事艾竞说“安全所带来的信任，是一种无形的产品，支撑着所有金融业务”，我深以为然，作为蚂蚁金服的基础设施部门，我们既要努力推进基础设施云原生化，也要保证全链路的安全，而安全容器，就是整个安全链路中最终“落地”的一环。无独有偶，基础设施云服务，作为云原生应用的发源地，也在和云原生应用的互动中前行，安全容器沙箱作为云服务的演化方向之一也在扮演日益重要的角色。本文中，蚂蚁金服和阿里云的容器开发者们会共同向大家介绍安全容器的前世今生、它们在双11 中的应用、以及未来的发展方向。

容器与安全容器

如果把一套云原生的系统想成一个有机体，那么，容器无疑就是其中的细胞，它们每个都是个独立的个体，又协同工作，它们生生不息让整个生命体健康运转。每个细胞都有它自己的隔离系统，正如容器，它的名字（Container，Jail，Zone...）天生就带着隔离的味道，但是，容器的安全性究竟够不够呢？年初 Tripwire 的一份调查揭示了人们对容器安全的种种担心【1】：

- 42% 的受访者因为安全顾虑推迟或限制了容器化进程；
- 58% 的在生产环境中使用容器的受访者承认它们的容器中存在安全缺陷，另外还有 13% 的人觉得自己的容器有安全问题，但不确定，而非常决绝地表示自己的容器非常安全的比例只有 7%。

面对这样的威胁，目前可用的容器层的解决方案就是 Linus Torvalds 在 2015 年北美 LinuxCon/-ContainerCon 上所说的——“安全问题的唯一正解在于允许那些（导致安全问题的）Bug 发生，但通过额外的隔离层来阻挡住它们。” 这也就是当今的安全容器技术的基本原理。（注：Linus 所说的思路是安全容器的思路，但安全容器不是从这里开始的，事实上，主流开源安全容器项目 Kata Containers 【2】的两个前身——runV 和 Clear Containers 都是在 Linus 上述访谈之前三个月开源的）。

隔离性不仅是安全

Kata Containers 在官方网站上展示的格言就是——

The speed of containers, the security of VMs

而且，这类技术最广为接受的名字也是“安全容器”。然而，隔离性所带来的好处远不止安全性这一点，在阿里巴巴和蚂蚁金服的大规模实践中，它至少还意味着四个方面的优势：

- 分层内核可以改善节点调度效率，容器的进程在自己的内核上被调度，对于主机来说，就只是一个进程，这在节点上运行着大量容器的时候，对节点的调度效率和稳定性都有极大的帮助；
- 良好的封装可以降低运维的压力，如果可以把应用进程乃至数据全部放入安全容器中，不对主机展示，那么对于主机的运维来说，将变得更加简单；
- 有利于系统的 QoS 保障和计费系统的准确工作，如果所有用户的相关的计算和数据流量都放在一个完全封装的安全容器的中，那么，对于主机管理网的 QoS 保障、对用户的 SLA 保障、对用户的计费都将是可操作且集中的；
- 有利于保障用户的数据隐私，普通容器的情况下，容器的文件系统数据和进程数据是完全暴露给主机的，对于云服务场景，这就要求用户授权服务方来访问用户数据，否则无异于掩耳盗铃，但如果可以通过安全容器的沙箱把这些都放在容器中，那么对于用户的授权要求也就可以最小化了。

显而易见，云原生基础设施确实是需要更好的隔离技术的，安全容器是大有可为的。

Kata Containers 与阿里云安全沙箱

上面已经提到，目前主流的开源安全容器项目是 Kata Containers，核心是用虚拟化技术来作为容器（确切地说是 Kubernetes Pod）的沙箱，从而提供虚拟机级别的隔离性，这个安全级别是在过去多年以来的云服务中被广泛接受的。该项目在两年前的 2017 年 12 月，由 runV 和 Intel 的 Clear Containers 项目合并而成，本文作者之一王旭就是 runV 项目的发起者，也是 Kata Containers 的创立者之一。

在过去两年中，Kata Containers 在保证了对标准容器镜像的兼容性和对 Kubernetes 生态的兼容性的同时，不断致力于与相关社区一起，降低沙箱技术的开销，提升安全容器技术的弹性。引入/支持/推动了 containerd shim-v2 API【3】，Kubernetes RuntimeClass，Rust-VMM，等来降低开销、减少间接层、改善集成 Kubernetes 效果。

在阿里巴巴和蚂蚁金服内部，基于 Kata Containers 等安全沙箱技术也被应用在内部不同应用等混合部署、阿里云 Serverless 服务等场景，支持着各种业务。

gVisor 与进程级虚拟化

Google 在 2018 年发布了基于进程级虚拟化技术的沙箱产品“gVisor”。gVisor 很轻薄，利用了基本的硬件虚拟化技术提供进程级别的抽象，无需 VMM 支撑。gVisor 的运行就像 Linux 进程一样，拉起速度快，对内存等资源的占用也较小。蚂蚁金服在 gVisor 开源之后不久，就开始投入开发。目前已经是 gVisor 社区中，Google 以外的最大贡献者。同时，蚂蚁针对自己的业务场景进行了大量的定制优化。现在已经成功将其应用于生产环境，进行了初步的生产验证。

gVisor 是安全容器技术，更是全新的操作系统。它实现了一个用 Go 语言写的“安全内核”。这个内核实现了 Linux Kernel 的绝大部分功能。由于 Go 语言在内存安全与类型安全上的优势，gVisor 的“安全内核”被视为一道重要的防御纵深。我们在开发中会 review 每一行代码的“安全性”。为了进一步加强安全，蚂蚁搭建了一套 fuzz 测试系统，一刻不停的测试。为了弥补 Go 语言在性能上的不足，蚂蚁重构了网络协议栈，在获取安全的同时，也大幅提升了性能。

面向云原生，gVisor 的最大优势是“资源可伸缩”。它不会预留 CPU 与内存资源，跟普通的 Linux 进程并无太大差异。在 gVisor 容器中，不用的资源会立刻还给宿主机。我们在尝试基于 gVisor 的进程级虚拟化技术，打破传统容器的资源边界。在 Serverless 等场景下，实现超高密度的容器部署。

未来：从安全容器到云原生沙箱

就目前来看，虽然云原生应用期待沙箱有更好的隔离性，但目前的沙箱技术尚有一些需求没有完全满足，所以，我们在规划 2.0 乃至未来的 Kata 的时候，就将演进方向定为：

- 在保持沙箱边界清晰的同时，可以跨沙箱共享一部分资源，极致降低开销；
- 更加按需、弹性、即时地根据应用的需求来提供资源，而不是硬性地切分；
- 让用户空间工具、沙箱、容器应用的内核联合为应用提供服务，因为应用的服务边界是内核 ABI 而不是模拟的硬件。

可以看到，这也喻示着两类安全容器技术实际上是彼此靠近的，虚拟化容器也与传统虚机渐行渐远，发展成为”云原生虚拟化“。蚂蚁金服和阿里云的正在这些方面共同合作，其中的部分工作已经展开，并在推动社区。

开源：从社区来，到社区去

目前，蚂蚁金服和阿里巴巴的安全容器技术都是基于开源社区而开发的，重要的开源社区包括 Kata Containers，gVisor 和 RustVMM。我们已经将一些成果推向社区，并在持续贡献。并且，秉承开放设计、开放开发的理念，我们也在将自己实践中的经验和问题，总结成为对未来的需求，不仅是对社区的修补和改进，更在推动社区在的设计理念、架构、协议方面前进。

未来正在到来，不论是加入社区做贡献还是直接加入我们，都非常欢迎。

参考链接

【1】The New Stack: Security Worries Rise as Container Adoption Increases, 2019-01
【2】Kata Containers, https://katacontainers.io
【3】Containerd Runtime V2 (shim-v2) API

04

云原生全链路加密

李鹏，花名壮怀，阿里云容器服务高级技术专家
黄瑞瑞，阿里云技术架构部资深技术专家

什么是云原生全链路加密

对于云上客户而言，其云上数据被妥善的安全保护是最重要的安全需求，也是云上综合安全能力的最具象体现。数据安全在云上的要求，可以用信息安全基本三要素 “CIA”来概括，即机密性（Confidentiality）、完整性（Integrity）和可用性（Availability）。机密性专指受保护数据只可以被合法的（或预期的）用户可访问，其主要实现手段包括数据的访问控制，数据防泄露，数据加密和密钥管理等手段；完整性是保证只有合法的（或预期的）用户才能修改数据，主要通过访问控制来实现，同时在数据的传输和存储中可以通过校验算法来保证用户数据的完整性；数据的可用性主要体现在云上环境整体的安全能力，容灾能力，可靠度，以及云上各个相关系统（存储系统、网络通路，身份验证机制和权限校验机制等等）的正常工作保障。

在三要素中，第一要素机密性（Confidentiality）的最常见也是最常被要求的技术实现手段就是数据加密。具体到云原生维度，需要实现的就是云原生的全链路加密能力。“全链路”指的是数据在传输（in Transit，也叫in-motion），计算（Runtime，也叫in-process），存储（in storage，也叫at-rest)的过程，而“全链路加密”指的是端到端的数据加密保护能力，即从云下到云上和云上单元之间的传输过程，到数据在应用运行时的计算过程（使用/交换），和到数据最终被持久化落盘的存储过程中的加密能力。在数据的传输，计算，和存储过程中，云原生的全链路加密能力即要满足云环境下的数据安全传输，又引入了容器网络，微服务，区块链场景，对云原生数据安全传输提出了进一步的要求。既有云安全虚拟化安全运行的要求，又有容器安全沙箱，可信安全沙箱的需求。既有云安全对云存储加密的需求，又有对容器镜像存储加密，审计日志，应用日志加密和三方集成的需求。

- 数据传输 （数据通信加密，微服务通信加密，应用证书和密钥的管理)
- 数据处理（运行时安全沙箱 runV, 可信计算安全沙箱 runE）
- 数据存储 （云原生存储的 CMK / BYOK 加密支持， 密文 / 密钥的存储管理，容器镜像的存储加密，容器操作 / 审计日志安全）

在这篇文章中我们会从云安全体系，到云数据安全，到云原生安全体系对全链路加密进行一次梳理，从而回答在云原生时代，全链路加密需要做什么，如何做到，以及未来要做什么。本文中的技术描述针对的是在云原生全链路加密中已有的和未来需要实现的技术目标。

云安全 > 云数据安全 > 云原生全链路加密

云安全

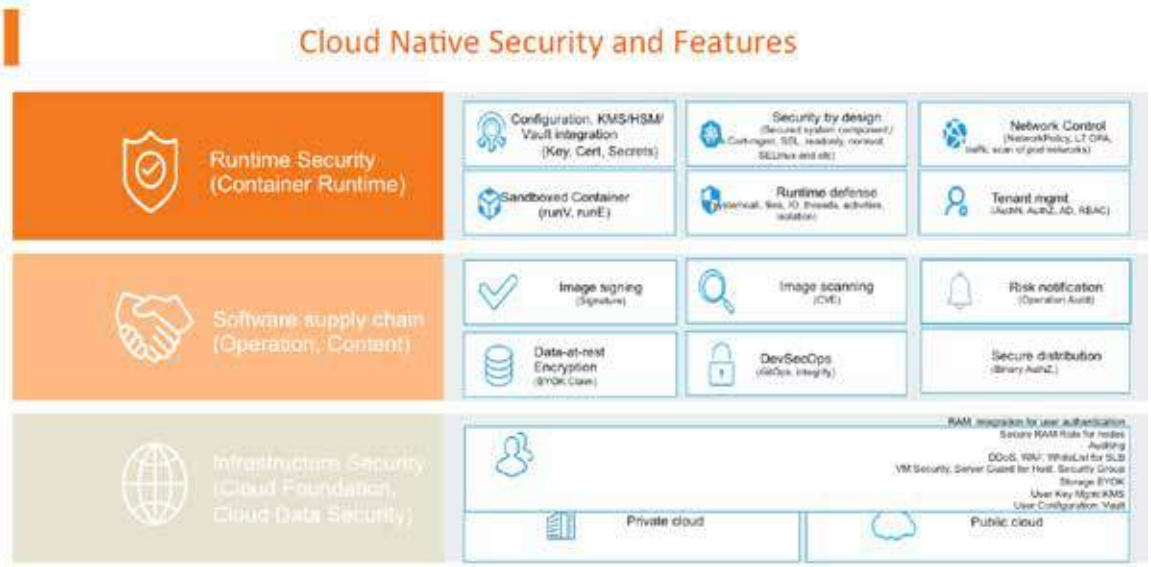
针对用户的群体的不同，对安全链路有不同的层次定义，云安全涵盖了云客户安全和云厂商安全在 IaaS 的软件，硬件以及物理数据中心等的安全。

1.云原生客户（Cloud Native Customer）安全

a.应用安全
b.操作安全
c.商业安全
d.容器网络安全
e.容器数据安全
f.容器运行时安全

2.云客户（Cloud Customer）安全
3.云厂商（Cloud IaaS DevOps）安全

云原生安全



云原生安全首先需要遵循云数据安全标准，在复用了云基础架构安全能力的前提下，同时在安全运行时，软件供应链上有进一步的安全支持。

云原生存储是通过声明式 API 来描述了云数据的生命周期，并不对用户透出底层 IaaS 的数据加密细节。不同的云原生存储一般作为云数据的载体，复用了云 IaaS 基础安全能力，还需要包括软件供应链中的镜像安全，和容器运行时 root 文件系统安全和容器网络安全。

1.云原生安全的运行时 = 数据处理过程中的计算安全，内存安全，文件系统安全和网络安全
2.云原生软件供应链安全 = 可执行文件 / 用户代码安全
3.云原生基础架构的安全 = 云数据存储安全

云数据安全

云用户数据安全包括以下的三个方面的工作

1.数据保护

a.RAM ACL 控制细粒度的数据的访问权限；
b.敏感数据保护（Sensitive Data Discovery and Protection，简称 SDDP），数据脱敏，数据分级分类。

2.数据加密

a.CMK 加密数据能力；
b.BYOK 加密数据能力。

3.密钥 / 密文管理，

a.KMS/HSM 等云服务；
b.三方 Vault 服务。

数据安全的生命周期

为了更好的理解数据保护，需要对数据安全的生命周期有一个了解，因为数据保护贯穿于整个的数据生命周期：

1.数据收集
2.数据传输
3.数据处理
4.数据交换
5.数据存储
6.数据销毁

云原生数据生命周期，以ACK（容器服务 Kubernetes）挂载阿里云云盘为例：

1.云盘 PV 的申明和创建定义了数据，云盘数据的加密需要在申明定义中就需要体现，对密钥匙选择，加密算法选择都可以申明式支持，RAM 权限细粒度遵循最小权限；
2.云盘挂载到虚拟机通过 PVC 在容器组 Pod 引用得以触发和实现；
3.云盘数据的解密通过用户 CMK / BYOK 在块设备上实现透明加密解密；
4.Pod 生命周期的变化导致 PVC 关联云盘在不同宿主 ECS 上的 Detach / Attach；
5.对 PV 的 Snapshot 生命触发了云盘 Snapshot 的创建；
PV 的删除可以通过 OnDelete 关联到云盘的中止和数据的删除。

全链路的数据安全

在狭义上来说是对数据的端到端的加密，主要集中在了数据生命周期中的三个阶段：
1.数据传输
2.数据处理
3.数据存储

数据传输阶段

安全通信设计，密文 / 密钥的安全管理和传输，既要满足云环境下的安全传输，云原生引入的容器网络、微服务、区块链场景，又对云原生数据安全传输提出了进一步的要求。

云安全传输

1.在云环境下 VPC / 安全组的使用，密文 / 密钥的安全管理 KMS 南北向流量通过 SSL 证书服务获取可信有效的 CA，对南北流量实现 HTTPS 加密和卸载，以及对 RPC/gRPC 通信使用 SSL 加密，减小 VPC 的攻击面，通过 VPN/SAG Gateway 来实现安全访问链路。

云原生安全传输

1.云原生场景，单一集群允许多租户的同时共享网络，系统组件权限控制，数据通信加密，证书轮转管理。多租场景下东西流量的网络隔离，网络清洗；
2.云原生微服务场景，应用 / 微服务间通信加密，和证书管理；
3.云原生场景下密钥，密文的独立管理和三方集成。KMS 与 Vault CA， fabric-ca，istio-cert-manager 等的集成。

数据处理阶段

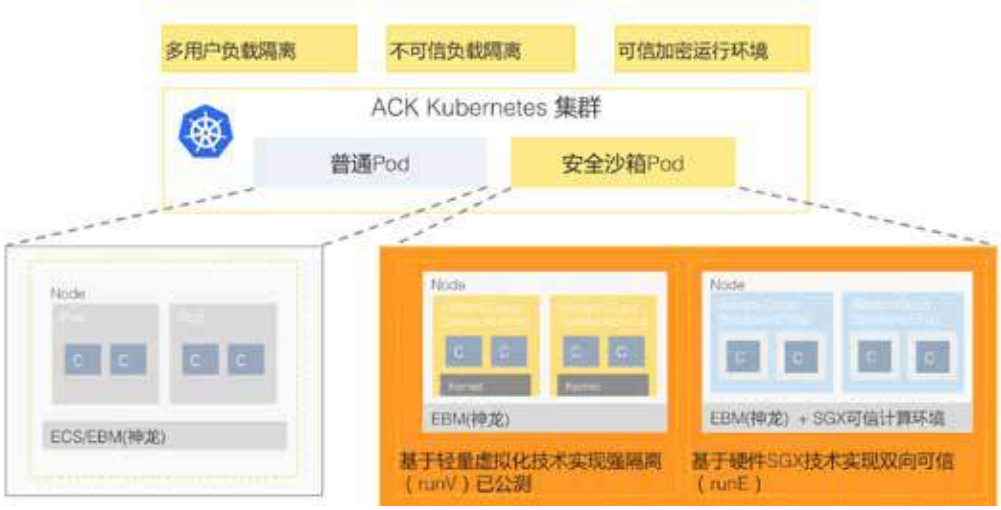
数据处理阶段，对内存级的可信计算，既有云安全虚拟化安全运行的要求，又有容器安全沙箱和可信安全沙箱的需求。

云安全虚拟化可信计算

1.TEE SGX
2.ARM Trust Zone

云原生容器安全沙箱

1.runV Kata 安全容器沙箱
2.runE Graphane / Occlum 可信安全沙箱



数据存储阶段

既有云安全对云存储加密，云数据服务加密需求，又有对容器镜像存储加密，审计日志，应用日志加密和三方集成的需求，以及对密文密码的不落盘存储的支持。

云存储加密方式

数据 + 加密算法 + 用户密钥或主密钥
客户端加密 / 服务端加密

1.云存储数据，以服务端加密为主
2.安全的加密算法，全面支持国产算法以及部分国际通用密码算法，满足用户各种加密算法需求。
a.对称密码算法：支持　SM1、SM4、DES、3DES、AES
b.非对称密码算法：支持　SM2、RSA（1024-2048）
c.摘要算法：支持　SM3、SHA1、SHA256、SHA384
3.安全的密钥管理 KMS / HSM
阿里云只能管理设备硬件，主要包括监控设备可用性指标、开通、停止服务等。密钥完全由客户管理，阿里云没有任何方法可以获取客户密钥。

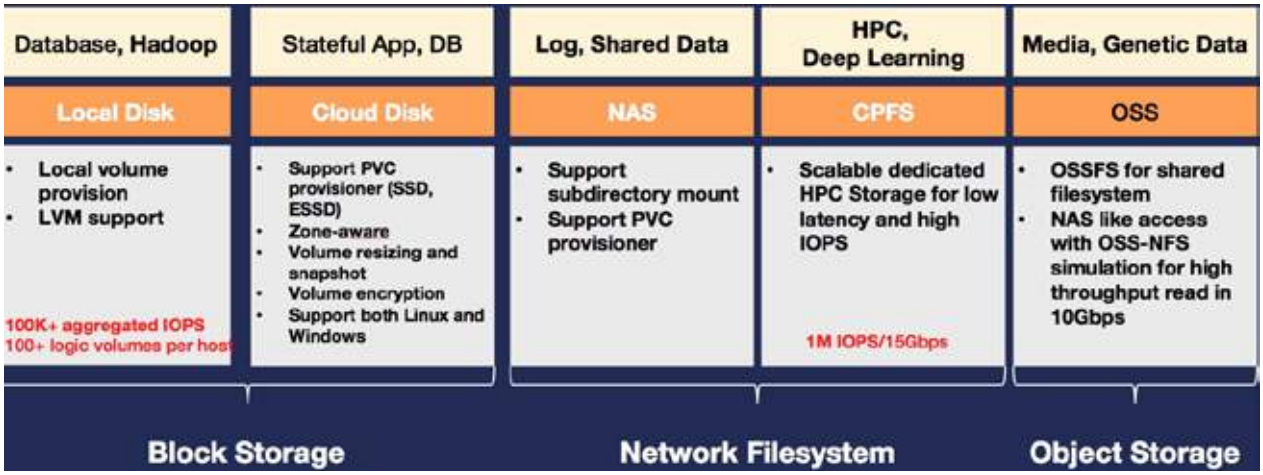
云存储加密支持

1.块存储　EBS 云盘：支持虚拟机内部使用的块存储设备（即云盘）的数据落盘加密，确保块存储的数据在分布式系统中加密存放，并支持使用服务密钥和用户自选密钥作为主密钥进行数据加密。
2.对象存储　OSS：支持服务端和客户端的存储加密能力。在服务端的加密中，支持使用服务密钥和用户自选密钥作为主密钥进行数据加密。在客户端的加密中，支持使用用户自管理密钥进行加密，也支持使用用户　KMS　内的主密钥进行客户端的加密。
3.RDS　数据库的数据加密：RDS　数据库的多个版本通过透明加密（Transparent Data Encryption，简称　　TDE）或云盘实例加密机制，支持使用服务密钥和用户自选密钥作为主密钥进行数据加密。
4.表格存储　OTS：支持使用服务密钥和用户自选密钥作为主密钥进行数据加密。
5.文件存储　NAS：支持使用服务密钥作为主密钥进行数据加密。
6.MaxCompute 大数据计算：支持使用服务密钥作为主密钥进行数据加密。
操作日志，审计日志的安全存储，以及三方日志系统集成。

云原生存储加密

目前阿里云容器服务 ACK 可以托管的主要以块存储，文件存储和对象存储为主，其他类型的 RDS，OTS 等数据服务是通过 Service Broker 等方式支持。

1.用户容器镜像 / 代码（企业容器镜像服务，OSS CMK / BYOK 加密）
2.云原生存储卷 PV（申明式支持云存储的 CMK / BYOK 以及数据服务层的加密支持）
3.操作日志和审计日志（ActionTrail OpenAPI, Kubernetes AuditLog: SLS 日志加密）
4.密文密码（KMS，Vault 对密文的三方加密支持和内存存储，非 etcd 持久化）



结论

云原生全链路的数据安全，云安全体系下的全链路加密已经成为了基础配置，新的容器化基础架构和应用架构的变化，结合云原生技术体系的特征，在数据传输，数据处理，数据存储阶段都需要增加相应云原生环境对网络，运行时，存储的全链路加密需求。既要满足云环境下的安全传输，云原生引入的容器网络，微服务，区块链场景，又对云原生数据安全传输提出了进一步的要求。既有云安全虚拟化安全运行的要求，又有容器安全沙箱，可信安全沙箱的需求。既有云安全对云存储加密，云数据服务加密需求，又有对容器镜像存储加密，审计日志，应用日志加密和三方集成的需求，以及对密文密码的不落盘存储的支持。

05

Kubernetes 下零信任安全架构分析

杨宁，花名麟童，阿里云基础产品事业部高级安全专家
刘梓溪，花名寞白，蚂蚁金服大安全基础安全安全专家
李婷婷，花名鸿杉，蚂蚁金服大安全基础安全资深安全专家

内容简介

零信任安全最早由著名研究机构 Forrester 的首席分析师约翰.金德维格在 2010 年提出。零信任安全针对传统边界安全架构思想进行了重新评估和审视，并对安全架构思路给出了新的建议。

其核心思想是，默认情况下不应该信任网络内部和外部的任何人/设备/系统，需要基于认证和授权重构访问控制的信任基础。诸如 IP 地址、主机、地理位置、所处网络等均不能作为可信的凭证。零信任对访问控制进行了范式上的颠覆，引导安全体系架构从“网络中心化”走向“身份中心化”，其本质诉求是以身份为中心进行访问控制。

目前落地零信任概念包括 Google BeyondCorp、Google ALTS、Azure Zero Trust Framework 等，云上零信任体系，目前还是一个新兴的技术趋势方向，同样的零信任模型也同样适用于 Kubernetes，本文重点讲解一下 Kubernetes 下零信任安全架构的技术分析。

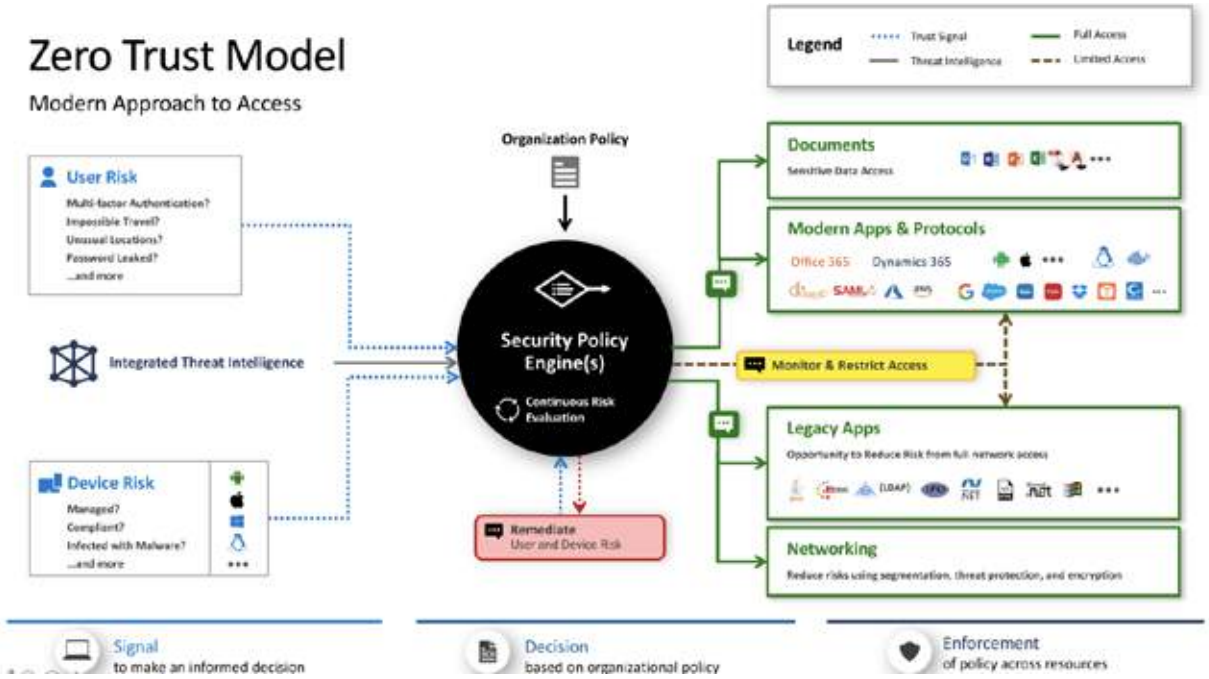
传统零信任概念和目前落地情况

Microsoft Azure

Azure的零信任相对来说还是比较完善的，从体系角度来看涵盖了端、云、On-Permises、SaaS 等应用，下面我们分析一下相关的组件：

- 用户 Identity：然后通过 Identity Provider（创建、维护和管理用户身份的组件）的认证，再认证的过程中可以使用账号密码，也可以使用 MFA（Multi Factor Auth）多因素认证的方式，多因素认证包括软、硬 Token、SMS、人体特征等；

- 设备 Identity：设备包含了公司的设备以及没有统一管理的设备，这些设备的信息，包含 IP 地址、MAC 地址、安装的软件、操作系统版本、补丁状态等存储到 Device Inventory；另外设备也会有相应的 Identity 来证明设备的身份；设备会有对应的设备状态、设备的风险进行判定；
  Security Policy Enforcement：通过收集的用户 Identity 以及状态、设备的信息，状态以及 Identity 后，SPE 策略进行综合的判定，同时可以结合 Threat Intelligence 来增强 SPE 的策略判定的范围和准备性；策略的例子包括可以访问后面的 Data、Apps、Infrastructure、Network；
- Data：针对数据（Emails、Documents）进行分类、标签、加密的策略；
- Apps：可以自适应访问对应的 SaaS 应用和 On-Permises 应用；
  Infrastructure：包括 IaaS、PaaS、Container、Serverless 以及 JIT（按需开启访问）和 GIT 版本控制软件；
- Network：针对网络交付过程以及内部的微隔离进行策略打通。

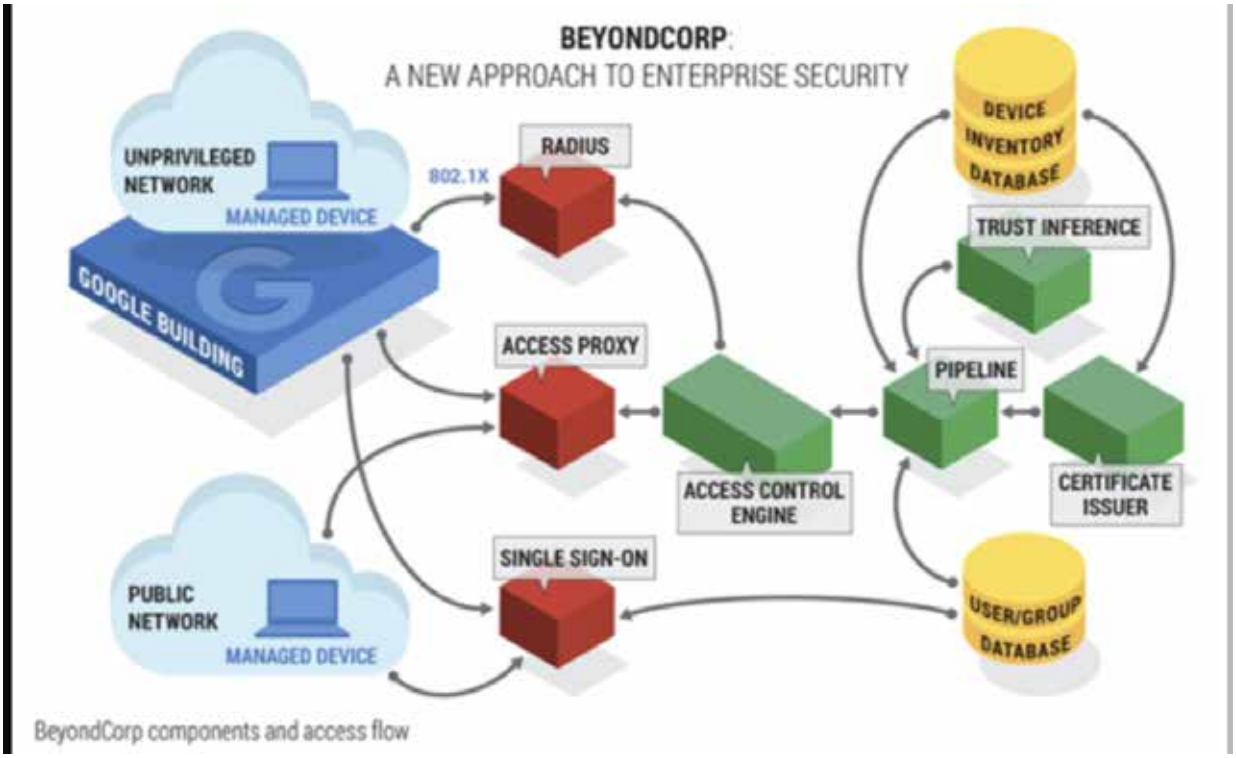


下面这张微软的图进行了更加细化的讲解，用户（员工、合作伙伴、用户等）包括 Azure AD、ADFS、MSA、Google ID 等，设备（可信的合规设备）包括 Android、iOS、MacOS、Windows、Windows Defender ATP，客户端（客户端 APP 以及认证方式）包括浏览器以及客户端应用，位置（物理和虚拟地址）包括地址位置信息、公司网络等，利用 Microsoft 的机器学习 ML、实时评估引擎、策略等进行针对用户、客户端、位置和设备进行综合判定，来持续自适应的访问 On-Permises、Cloud SaaS Apps、Microsoft Cloud，包含的策略包括 Allow、Deny，限制访问、开启 MFA、强制密码重置、阻止和锁定非法认证等；从下图可以看出 Azure 已经打通了 On-Permises、Cloud、SaaS 等各个层面，构建了一个大而全的零信任体系。

Google BeyondCorp

Google BeyondCorp 是为了应对新型网络威胁的一种网络安全解决方案，其实 Google BeyondCorp 本身并没有太多的技术上的更新换代，而是利用了持续验证的一种思路来做的，去掉了 VPN 和不再分内外网。Google 在 2014 年之前就预测到互联网和内网的安全性是一样危险的，因为一旦内网边界被突破的话，攻击者就很容易的访问企业的一些内部应用，由于安全意识的问题导致企业会认为我的内部很安全，我就对内部的应用进行低优先级别的处理，导致大量内部的安全问题存在。现在的企业越来越多的应用移动和云技术，导致边界保护越来越难。所以 Google 干脆一视同仁，不分内外部，用一样的安全手段去防御。

从攻防角度来看一下 Google 的 BeyondCorp 模型，例如访问 Google 内部应用http://blackberry.corp.google.com ，它会跳转到https://login.corp.google.com/ 也就是 Google Moma 系统，首先需要输入账号密码才能登陆，这个登录的过程中会针对设备信息、用户信息进行综合判定，账号密码正确以及设备信息通过规则引擎验证之后，会继续跳转到需要 YubiKey 登录界面，每个 Google 的员工都会有 Yubikey，通过 Yubikey 来做二次验证。Yubikey 的价值，Google 认为是可以完全杜绝钓鱼攻击的。另外类似的就是 Amazon 的 Midway-Auth 方式 （ https://midway-auth.amazon.com/login?next=%2F ）。



Kubernetes 下容器零信任模型

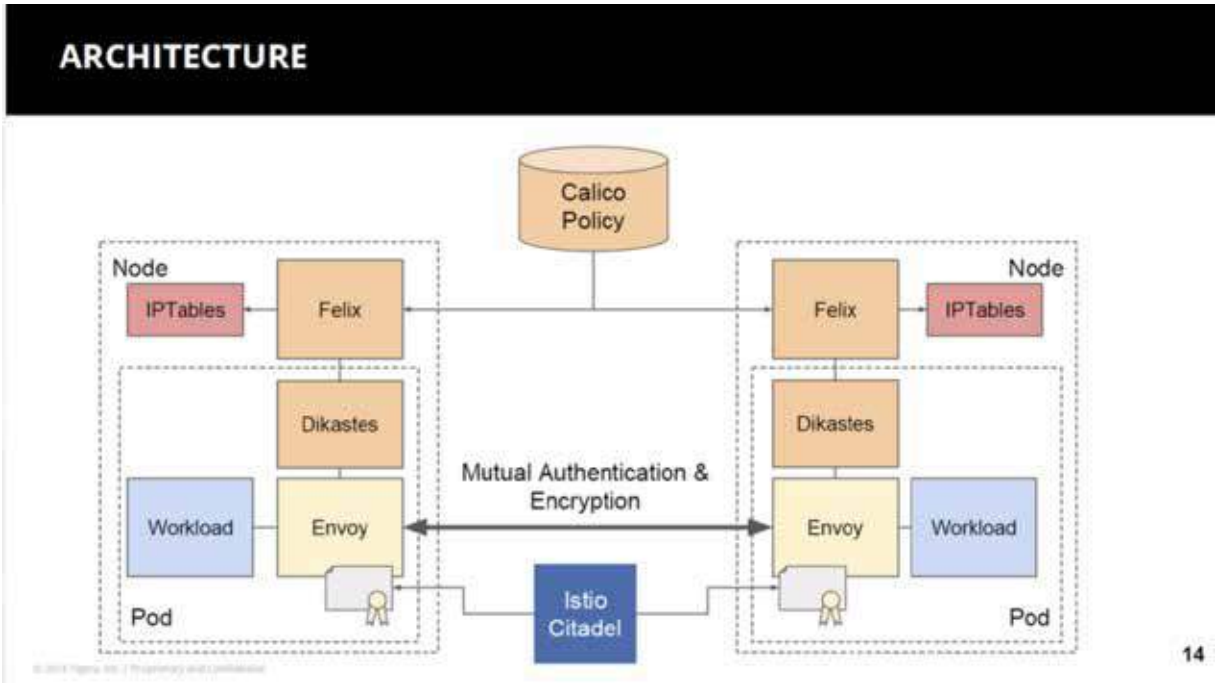
容器下网络零信任

首先介绍一下容器下的网络零信任组件 Calico，Calico 是针对容器，虚拟机和基于主机的本机 Workload 的开源网络和网络安全解决方案产品。Calico 支持广泛的平台，包括 Kubernetes、OpenShift、Docker EE、OpenStack 和裸金属服务。零信任最大的价值就是即使攻击者通过其他各种手法破坏应用程序或基础架构，零信任网络也具有弹性。零信任架构使得使攻击者难以横向移动，针对性的踩点活动也更容易发现。

在容器网络零信任体系下，Calico+Istio 目前是比较热的一套解决方案；先来看看两者的一些差别，从差别上可以看到 Istio 是针对 Pod 层 Workload 的访问控制，以及 Calico 针对 Node 层的访问控制：

	Istio	Calico
Layer	L3-L7	L3-L4
实现方式	用户态	内核态
策略执行点	Pod	Node

下面重点讲解一下 Calico 组件和 Istio 的一些技术细节，Calico 构建了一个 3 层可路由网络，通过 Calico 的 Felix（是运行在 Node 的守护程序，它在每个 Node 资源上运行。Felix 负责编制路由和 ACL 规则以及在该 Node 主机上所需的任何其他内容，以便为该主机上的资源正常运行提供所需的网络连接）运行在每个 Node 上，主要做路由和 ACL 的策略以及搭建网络；通过运行在 Node 上的 Iptables 进行细粒度的访问控制。可以通过 Calico 设置默认 Deny 的策略，然后通过自适应的访问控制来进行最小化的访问控制策略的执行，从而构建容器下的零信任体系；Dikastes/Envoy：可选的 Kubernetes sidecars，可以通过相互 TLS 身份验证保护 Workload 到 Workload 的通信，并增加相关的控制策略；



Istio

再讲解 Istio 之前先讲一下微服务的一些安全需求和风险分析：

1、微服务被突破之后通过 Sniffer 监控流量，进而进行中间人攻击，为了解决这种风险需要对流量进行加密；

2、为了针对微服务和微服务之间的访问控制，需要双向 TLS 和细粒度的访问策略；

3、要审核谁在什么时候做了什么，需要审计工具；

分析了对应的风险之后，下面来解释一下 Istio 如何实现的零信任架构。首先很明显的一个特点就是全链路都是双向 mTLS 进行加密的，第二个特点就是微服务和微服务之间的访问也可以进行鉴权，通过权限访问之后还需要进行审计。Istio 是数据面和控制面进行分离的，控制面是通过 Pilot 将授权策略和安全命名信息分发给 Envoy，然后数据面通过 Envoy 来进行微服务的通信。在每个微服务的 Workload 上都会部署 Envoy，每个 Envoy 代理都运行一个授权引擎，该引擎在运行时授权请求。当请求到达代理时，授权引擎根据当前授权策略评估请求上下文，并返回授权结果 ALLOW 或 DENY。

Istio 安全架构

微服务下的 Zero Trust API 安全

42Crunch（ https://42crunch.com/ ）将 API 安全从企业边缘扩展到了每个单独的微服务，并通过可大规模部署的超低延迟微 API 防火墙来进行保护。 42Crunch API 防火墙的部署模式是以 Kubernetes Pod 中以 Sidecar 代理模式部署，毫秒级别的性能响应。 这省去了编写和维护单个 API 安全策略过程，并实施了零信任安全体系结构，提升了微服务下的 API 安全性。42Crunch 的 API 安全能力包括：审核：运行 200 多个 OpenAPI 规范定义的安全审核测试，并进行详细的安全评分，以帮助开发人员定义和加强 API 安全；扫描：扫描实时 API 端点，以发现潜在的漏洞；保护：保护 API 并在应用上部署轻量级，低延迟 Micro API Firewall。

蚂蚁零信任架构体系落地最佳实践

随着 Service Mesh 架构的演进，蚂蚁已经开始在内部落地 Workload 场景下的服务鉴权能力，如何建设一套符合蚂蚁架构的 Workload 间的服务鉴权能力，我们将问题分为一下三个子问题：

1、Workload 的身份如何定义，如何能够实现一套通用的身份标识的体系；

2、Workload 间访问的授权模型的实现；

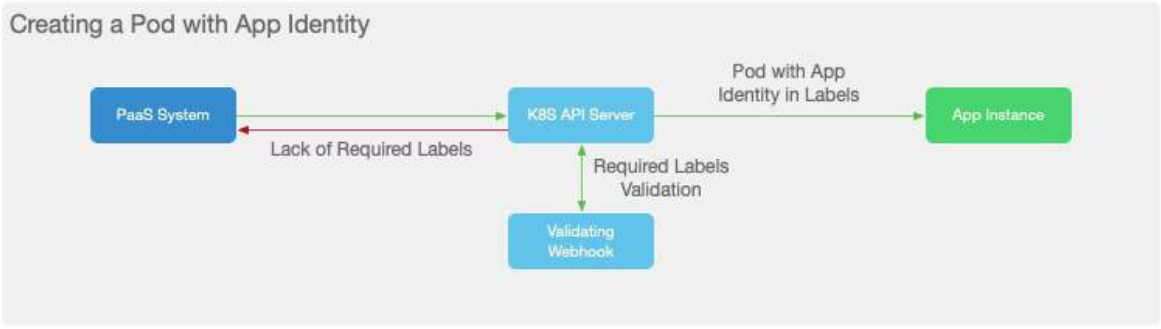
3、访问控制执行点如何选择。

Workload 身份定义 & 认证方式

蚂蚁内部使用 SPIFFE 项目中给出的 Identity 格式来描述 Workload 的身份，即：

spiffe://<domain>/cluster/<cluster>/ns/<namespace>

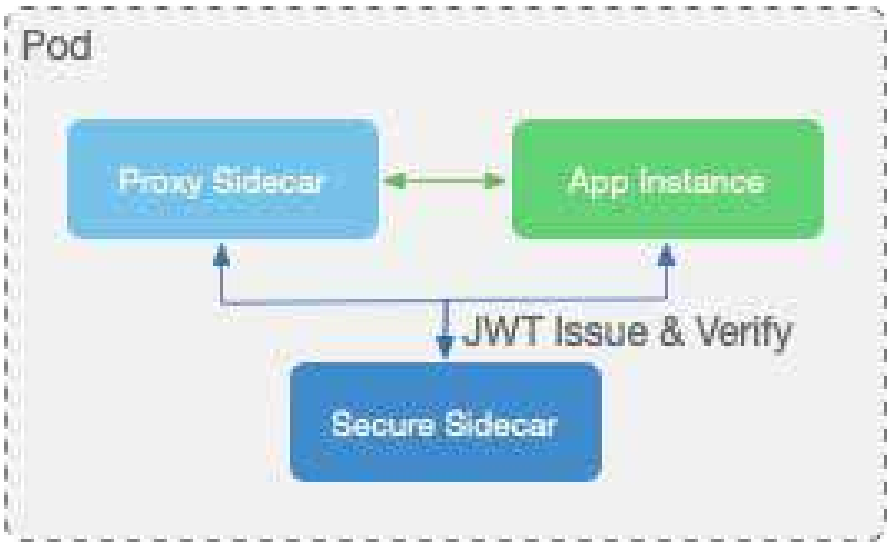
不过在工程落地过程中发现，这种维度的身份格式粒度不够细，并且与 K8s 对于 namespace 的划分规则有较强的耦合。蚂蚁的体量较大，场景较多，不同场景下 namespace 的划分规则都不完全一致。因此我们对格式进行了调整，在每一场景下梳理出能够标识一个 Workload 示例所须要的一组必备属性（例如应用名，环境信息等），并将这些属性携带在 Pod 的 Labels 中。调整后的格式如下：
spiffe://<domain>/cluster/<cluster>/<required_attr_1_name>/<required_attr_1_value>/<required_attr_2_name>/<required_attr_2_value>



在解决了 Workload 身份定义的问题后，剩下的就是如何将身份转换成某种可校验的格式，在 Workload 之间的服务调用链路中透传。为了支持不同的使用场景，我们选择了 X.509 证书与 JWT 这两种格式。

对于 Service Mesh 架构的场景，我们将身份信息存放在 X.509 证书的 Subject 字段中，以此来携带 Workload 的身份信息。如下图所示：

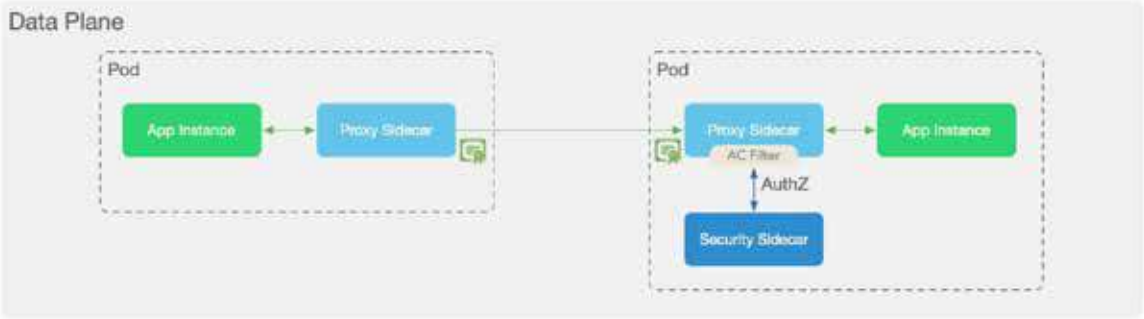
对于其他场景，我们将身份信息存放在 JWT 的 Claims 中，而 JWT 的颁发与校验，采用了 Secure Sidecar 提供服务。如下图所示：



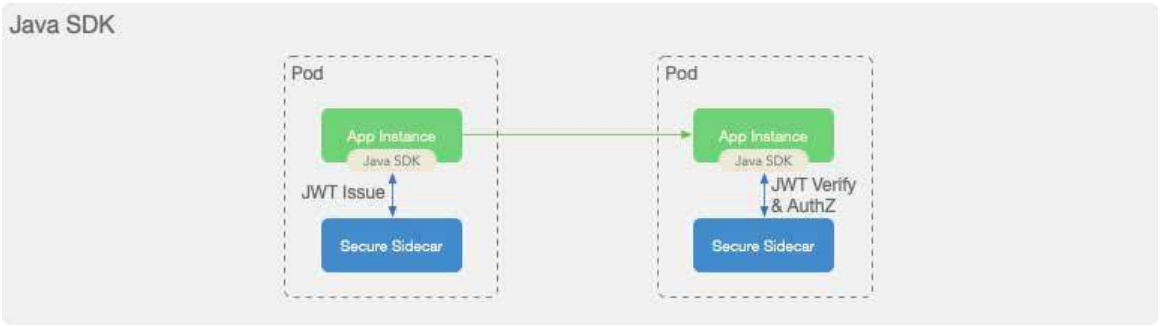
授权模型

在项目落地的初期，使用 RBAC 模型来描述 Workload 间服务调用的授权策略。举例描述，应用 A 的某一个服务，只能被应用 B 调用。这种授权策略在大多数场景下都没有问题，不过在项目推进过程中，我们发现这种授权策略不适用于部分特殊场景。

我们考虑这样一个场景，生产网内部有一个应用 A，职责是对生产网内部的所有应用在运行时所需要使用的一些动态配置提供中心化的服务。这个服务的定义如下：
A 应用 - 获取动态配置的 RPC 服务：
```message FetchResourceRequest {
// The appname of invoker
string appname = 1;
// The ID of resource
string resource_id = 2;

在此场景下，如果依然使用 RBAC 模型，应用 A 的访问控制策略将无法描述，因为所有应用都需要访问 A 应用的服务。但是这样会导致显而易见的安全问题，调用方应用 B 可以通过该服务获取到其它应用的资源。因此我们将 RBAC 模型升级为 ABAC 模型来解决上述问题。 我们采用 DSL 语言来描述 ABAC 的逻辑，并且集成在 Secure Sidecar 中。

### 访问控制执行点的选择

在执行点选择方面，考虑到 Service Mesh 架构推进需要一定的时间，我们提供了两不同的方式，可以兼容 Service Mesh 的架构，也可以兼容当前场景。

在 Service Mesh 架构场景下，RBAC Filter 和 ABAC Filter（Access Control Filter）集成在 Mesh Sidecar 中。
```



在当前场景下，我们目前提供了 JAVA SDK，应用需要集成 SDK 来完成所有认证和授权相关的逻辑。与 Service Mesh 架构场景类似，所有 Identity 的颁发、校验，授权与 Secure Sidecar 交互，由 Secure Sidecar 完成。



结语

零信任的核心是“Never Trust， Always Verify”，未来会继续深化零信任在整个阿里巴巴的实践，赋予不同的角色不同的身份，例如企业员工、应用、机器，并将访问控制点下沉到云原生基础设施的各个点，实现全局细粒度的控制，打造安全防护的新边界。本文从业界的零信任体系的落地最佳实践，到基于 Kubernetes 的零信任落地方式进行了简单的描述，本文只是抛砖引玉，希望能引发更多关于 Cloud Native 下的零信任架构体系的更多讨论和能看到更多的业界优秀的方案和产品能出现。

06

云原生时代的 Identity

艾竞，花名庭坚，蚂蚁金服系统部研究员
李婷婷，花名鸿杉，蚂蚁金服平台安全资深安全专家

背景

随着轻量级容器（container）的兴起，应用（applications）逐渐由单体（monolithic）向微服务（micro-service）演进。相应的，应用的微服务化也带来了如下的变化：

- 软件研发、运维人员组织围绕着微服务下的软件模块重新架构：研发、运维人员分化成更小的组织以专注于一个或多个软件模块（以容器的形式交付）；在微服务 API 稳定的情况下，各个模块独立开发，测试和部署可以进行更高速的迭代
- 应用中软件模块之间由函数调用的“强”耦合变成网络远程调用的“松”耦合；维持原先各个模块之间的信任需要有额外的安全机制

另一方面，除了台式机（desktop），其它可以接入应用的物理设备（device），例如笔记本电脑（laptop），手机（mobile phone），也已经大规模普及。因此，企业员工通常拥有多个设备来访问企业应用以完成日常工作。个人多设备以及设备的移动性对于企业传统的安全边界模型（perimeter model）提出了挑战。

可以观察到，在应用由单体向微服务，设备由单一、静态向多种、移动的演化下，系统中各个 Entity（实体，例如，人，设备，服务等）的 Identity（身份），由单一的网络标识，即 IP 或者 IP:Port，向多种形态（X.509 certificate, token, service account 等）分化。Identity 的分化会对研发、运维效能和资源效能也会产生深刻的影响。

在这些背景下，传统基于 IP、域名的管控方式临着巨大的挑战，静态的规则让安全策略变得无法维护；同时，在微服务架构下，安全人员很难快速找到一条边界，将不同的风险域隔离开来。因此，急需一种新 Access Control 模式来应对。

Forrester 的首席分析师约翰·金德维格在 2010 年提出了零信任（Zero Trust）的理念针对传统边界安全架构思想进行了重新评估和审视，并对安全架构思路给出了新的建议，其核心思想是 "Never Trust，Always Verify" – All network traffic is untrusted. There is no "trusted traffic"。阿里巴巴已经在内部开始推进零信任架构的落地，包括企业内部接入层的零信任、生产网内部机器、应用间的零信任，并已经开始在 双11 中发挥了重要的作用。

零信任第一个核心问题就是 Identity，赋予不同的 Entity 不同的 Identity，解决是谁在什么环境下访问某个具体的资源的问题。因此，本文将从 Identity 与微服务和 Identity 与企业应用访问两个方面来阐述 Identity 在生产应用和企业应用中的思考，同时也会介绍在不同的场景下如何利用 Identity 解决谁在什么环境可以访问某个资源的具体实践。

Identity 与微服务

随着用户需求在信息化时代的日益增长，软件也变得越来越复杂。为了应对软件复杂度的提升，软件的架构也由简单的单体应用逐渐演变成为复杂的微服务应用。2017 年，Kubernetes 开始成为事实上的标准容器编排平台。同一年，谷歌联合 IBM，Lyft 推出的 Istio 主打 SideCar 模式下的微服务治理。至此，在云原生时代，应用得以在 Kubernetes 和 Istio 构成的底座之上以容器的形式微服务化。

在云原生的转型中，软件研发、运维组织架构需要围绕微服务来重新构建：如果把一组在逻辑、功能上耦合紧密的软件模块看作一门“功夫”的话，它们背后的研发、运维人员就是一个“武林门派”；各个“武林门派”相对独立的“修炼”自己的“功夫”；配合起来，又能满足软件系统整体的演进。然而，这套适配微服务的组织架构对于研发、运维模式提出了很大的挑战。归结到一点：就是从每个研发、运维者的角度，如何在关注本门“功夫”的同时保证软件系统整体的可用性。

Identity 由单一网络地址变得更多元化以标识不同的对象（例如，用户，用户组，服务等）为应对上述挑战提供了可能性。接下来，我们会看到 Identity 是如何覆盖整个软件开发周期各个场景为每个研发、运维者创造相对隔离的“环境”的。

Identity 与 Kubernetes 集群

首先，我们来看 Identity 是如何依据用户/用户组提供所需 Kubernetes 集群资源的。Kubernetes 集群是用于容器生命周期管理的。细分一下，其用户大致可以分为如下两类：

- Kubernetes 集群的管理者：关注集群本身，主要的职责包括并不限于：
  - Kuberentes 组件的维护及升级
  - Kuberentes 集群中 Node 的治理
  - Kuberentes 集群中 Namespace 的治理
  - Kuberentes 集群中各类 policy（例如，Role-Based Access Control（RBAC） for AuthZ）的治理
  - Kuberentes 集群中 Pod 调度管理
  - Kuberentes 集群中 C[X]I（例如，CRI，CNI，CSI 等）的治理
  - Kuberentes 集群资源容量管理

    。。。
- Kubernetes 集群的使用者：关注如何在集群之上运行容器，依据角色（例如，开发/测试/运维）的不同，主要的使用方式有：
  - Kuberentes 集群中 Service Account 的治理
  - 将 Kubernetes 集群作为开发环境
  - 将 Kubernetes 集群作为线下测试环境，运行 CI/CD
  - 将 Kubernetes 集群作为线上发布环境，进行灰度，回滚等操作

对于第一类 Kubernetes 集群用户，可以通过 Kubernetes RBAC 机制下 ClusterRoleBinding 赋予集群级别权限；并且需要“垄断”集群和 Namespace 级别策略的制定；为了防止敏感信息泄露，还需要将集群和 Namespace 级别的信息以及敏感信息（例如 Secret）设置为仅为 Kubernetes 集群管理者可见。

对于第二类 Kubernetes 集群用户，可以进一步按照对应的微服务中软件模块来进行细分成不同的组（Group）。另一方面，Kubernetes Namespace 提供了虚拟集群的抽象。因此，当 Kubernetes API Server 认证用户身份之后，可以按照组的粒度将用户映射到 Kubernetes 集群某个 Namespace 的角色（Role）而这个角色在此 Namespace 中拥有相应权限。具体来说，用户到 Namespace 的映射可以通过 Kubernetes RBAC 机制下 RoleBinding 完成。考虑到系统策略的可维护性，以组 Identity 的粒度制定策略而以用户 Identity 用于审计（Audit）是合理的。

Kubernetes 中并没有代表用户的资源，并且企业一般都会拥有多个 Kubernetes 集群。因此，提供组织架构信息（例如，用户（User），组（Group））的 Identity Provider（IdP）应该独立于 Kubernetes 集群之外并可以服务多个 Kubernetes 集群。如果对多个 Kubernetes 集群采用相同的配置，用户可以收获跨集群的一致性用户体验。

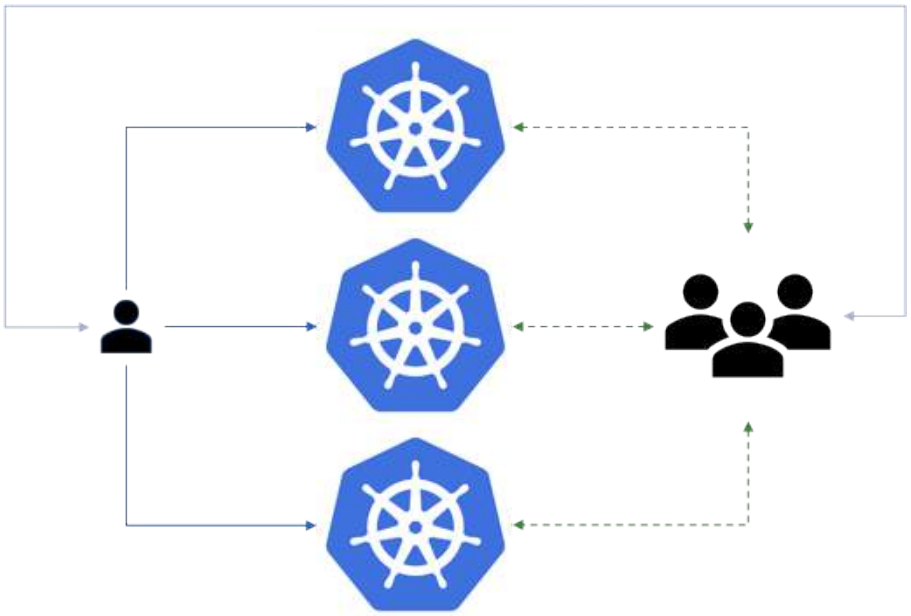


图 1：Identity Provider 与 Kubernetes 集群

在 IdP 中的组织架构，一个组可以包含用户，也可以包含其他组；与此同时，一个用户也可以直接棣属于多个组。因此可以根据用户操作（用户执行的 yaml 文件中指定了具体的 Namespace）来更加灵活映射到不同的角色来操作相应的 Kubernetes Namespace：

例子一：一个用户按架构域划分属于 A 组，按角色划分属于 dev 组。当要运行容器时，Kubernetes 集群的 RBAC 策略会允许他在 Namespace A-dev 中操作。

例子二：一个用户按架构域划分既属于 A 组也属于 B 组，按角色划分属于 test 组。当要运行容器时，Kubernetes 集群的 RBAC 策略会允许他在 Namespace A-test 和Namespace B-test 中操作。

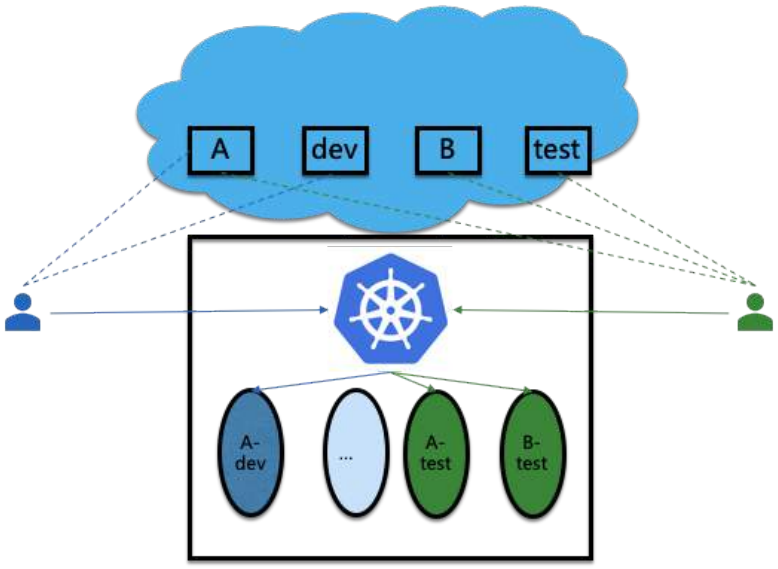


图 2：企业组织架构与 Kubernetes 集群

值得注意的是，IdP 中的组织架构代表 Kubernetes 管理者和使用者的组还是要严格分开以防止权限泄露。此外，当有任何组织架构变动时（例如，用户从 Kubernetes 管理者变成使用者），Kubernetes 集群需要能够及时捕捉。

Identity 与 Service Mesh

其次，我们来看 Identity 是如何依据用户/用户组和相应的配置创造出不同的软件运行环境的。当用户通过 Kubernetes 开始运行容器后，容器所代表的微服务就交由 Service Mesh 来治理。如图 3 所示，在 Istio 1.1+，在启动 Pod 的时候，Pod Identity 会通过 SDS（Secret Discovery Service）获取。

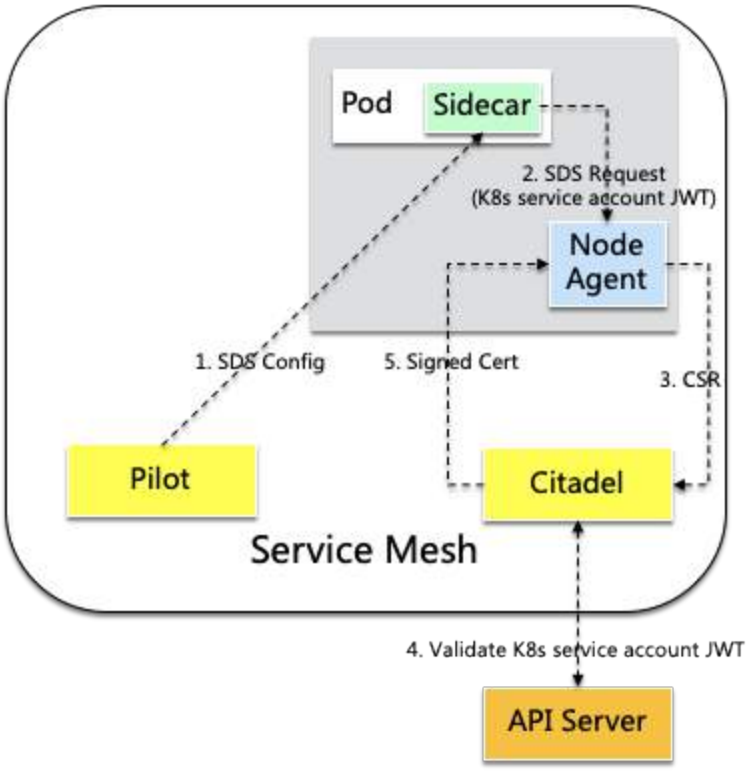


图 3：Pod Identity 获取流程

Pod 的 Identity 的格式会采取源自谷歌 LOAS（Low-Overhead Authentication Service）的 SPIFFE（Secure Production Identity Framework for Everyone）标准，spiffe://<cluster>/ns/<namespace>/sa/<service account>以全局唯一标识相应的 Workload；并且 Identity 相应的证书也会由 CA（Certificate Authority）来签发。有了证书背书，当 Pod 可以与其它 Pod 通信时，就可以通过 mTLS 完成对通信链路加密。

更为重要的是，Service Mesh 可以通过 Istio RBAC 实现基于 Identity 的 RPC（Remote Procedure Call）ACL（Access Control List）。因此，当由有不同角色（例如，开发/测试/运维）的用户来运行/部署同一个微服务时，因为所在组不同而获取不同的 SPIFFE ID，即使在同一个 Kubernetes 集群中，只要设置合适的 Istio RBAC Policy，他们能运行的微服务也处于彼此隔离的环境中。

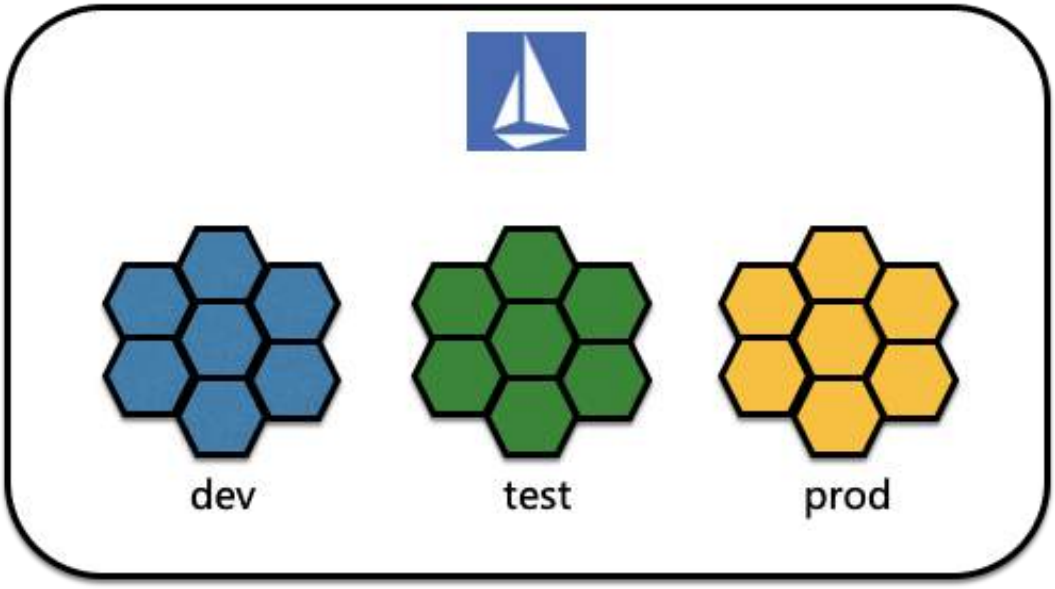


图 4：Istio 下基于 Identity 隔离的环境

此外，一个组通常会拥有微服务的多组软件模块。当映射到 Kubernetes Namespace 时，对不同组的软件模块最好采用不同的 Service Account。这是因为，Pod 的 SPIFFE ID 的信任来自于 Kubernetes Service Account；一旦Service Account 被 compromise，它所造成的损失有限。

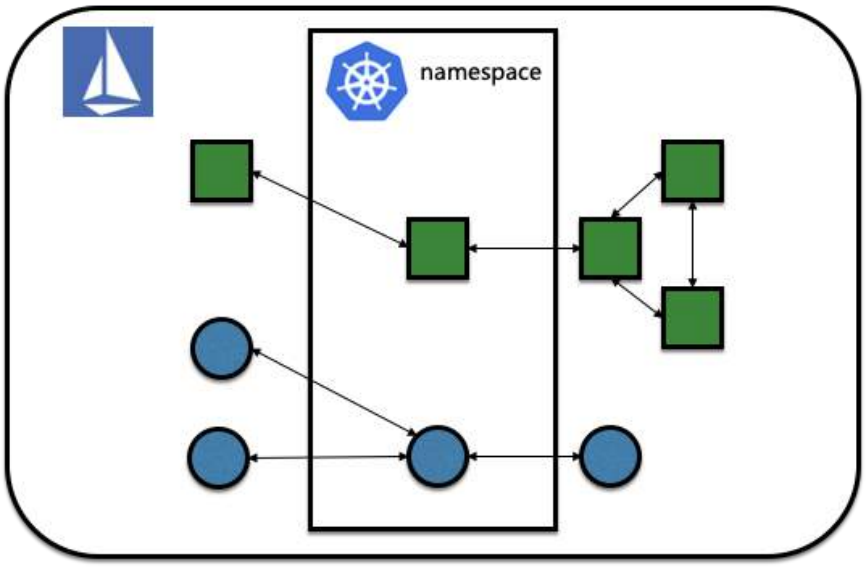


图 5：Service Account 的隔离

Workload Identity 的实践

阿里巴巴正在全面推行 Service Mesh 的架构演进，这为我们在此场景下快速落地 Workload 的 Identity 及访问控制提供了很好的基础。

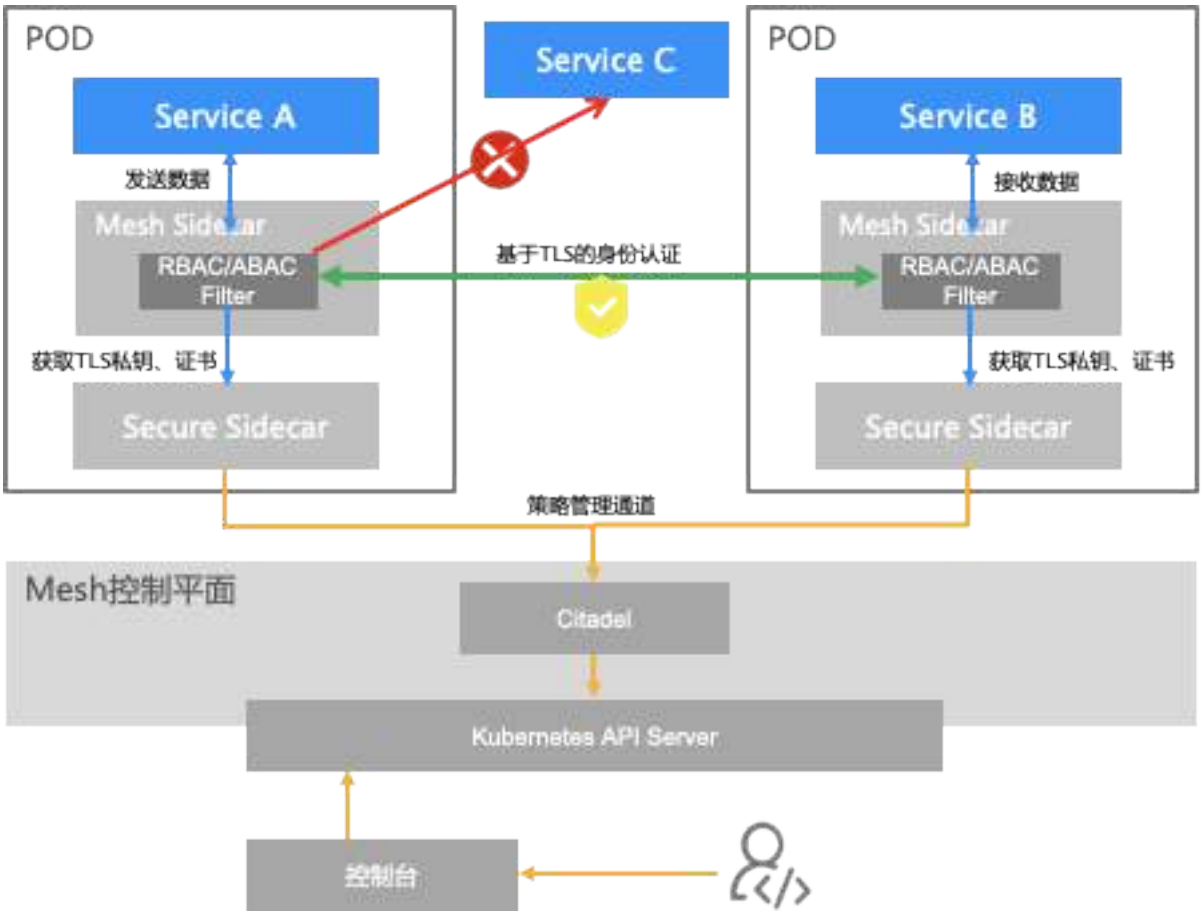


图 6：Service Mesh 架构下应用间访问控制

应用的身份以及认证统一由安全 Sidecar 统一提供，授权模型我们采用了 RBAC 模型，在 Sidecar 中集成了 RBAC Filter。RBAC 模型在大部分场景下都能满足需求，但在项目推进过程中我们发现一种比较特殊的场景，比如 A 应用需要被所有的应用访问，会导致策略无法描述。因此在这种场景下，我们考虑升级为 ABAC（Attribute-Based Access Control）模型，采用 DSL（Domain Specific Language）语言来描述，并集成在安全 Sidecar 中。

Identity 与软件交付

此外，Identity 不仅仅体现在软件研发、测试及运维，它还覆盖其他的方面。

镜像管理

在云原生范式下，代码终将以镜像的形式来使用。在关注构成镜像本身代码的安全性同时，我们还关心镜像的来源（即相应代码库 URL），制作者（组）和标签。在线上生产环境（其它环境可以放松这个限制）以容器运行镜像时，我们需要确保镜像是来自已知代码库，由合法的用户（来自拥有代码所有权的组）制作并有合适的标签（例如，release）。因此，我们需要在镜像的 metadata 中植入相关的 Identity 信息以便应用相关的策略。

微服务部署

在云原生时代，微服务部署所需的 yaml 配置往往是庞大繁琐的。如前所述，Identity 结合 Kubernetes/Istio RBAC Policies 可以创建出不同的隔离环境以便于开发，测试和运维，但是也需要有相应的 yaml 配置来适配。然而，yaml仅仅只是对一组可嵌套<key: value>的描述，自身没有任何复用，重载的计算机编程语言的特性。因此，我们需要使用云原生场景下的 DSL 来方便的刻画同一个微服务不同（例如，研发/测试/生产）环境下的配置。举例来说，针对不同的环境，主要区别在于：1）Kubernetes Namespace 不同；2）所需的计算资源不同。所以，DSL 可以将各个环境相同的配置沉淀下来形成模版，在此基础之上，对与环境相关的配置进行定制或重载以适配不同环境。

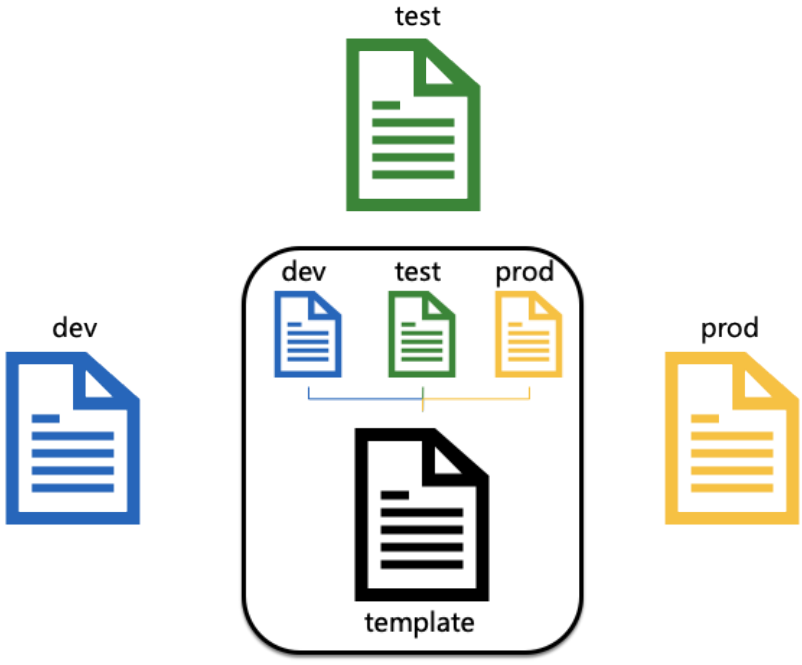


图 7：DSL 下的微服务配置

通常来讲，相对于微服务代码变更，微服务配置变更属于低频操作。DSL 将微服务配置接管过来使得配置也像代码一样可重用、可验证，大大降低因为配置及其变更带来的错误。因此，所有的开发者可以对代码进行更高速的迭代从而把代码缺陷暴露在更早的阶段。

Identity 与企业应用访问

阿里巴巴已经内部已部分地落地了基于零信任的企业应用访问的架构，从以前简单以员工的身份作为唯一且静态的授权依据，扩展为员工身份、设备、以持续的安全评估达到动态安全访问控制的新架构。

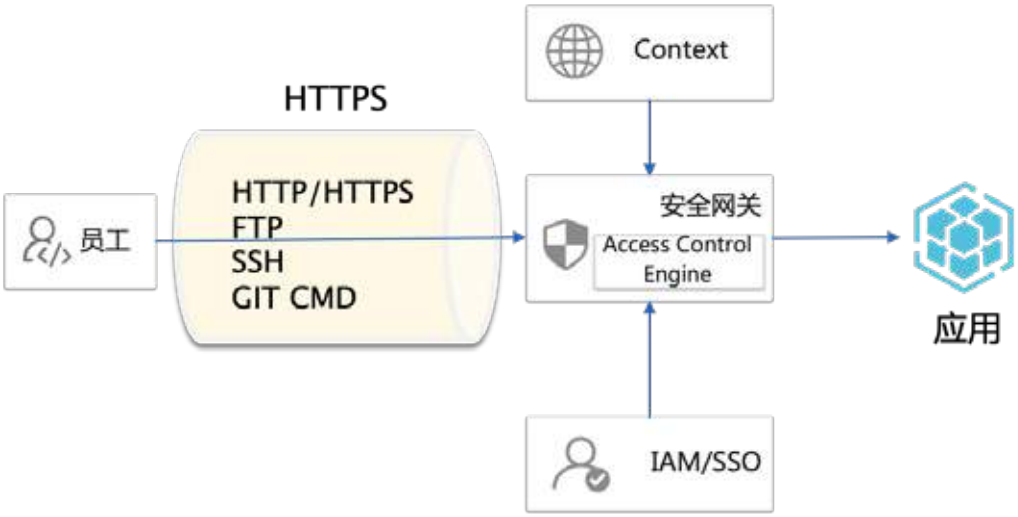


图 8：企业应用的访问控制

在这种新的架构下，首先，我们选择安全网关作为控制点，与阿里巴巴 IAM（Identity Access Management）结合，对员工的身份以及接入的设备进行认证，知道是否从对应的人和其对应的设备请求接入；协议方面，员工设备与安全网关之前，采用 HTTPS 对各种协议进行封装，统一由安全网关进行代理访问后端的应用；其次，安全网关内置的 Access Control Engine 结合流量中的上下文（Context）对访问的请求做持续的建模，对来自非可信、非认证设备、或者异常的访问行为做实时的阻断。

结语

随着互联网应用由单体向微服务架构转型，加之访问互联网应用的移动设备的繁荣，网络标识（例如，IP，Port等）已经不能作为稳定的 Identity 来标识分布式系统中的实体，建立在此之上的边界安全模型应对各种场景已经显得力不从心。因此，对于不同实体（例如，人，设备，服务等），Identity 需要由更能凸显其本质的、有密码学背书的稳定凭证（credential）来担当。相应的，建立在此类 Identity 基础之上安全策略有着丰富的语义来适配不同的场景：在研发、测试和运维微服务场景下，Identity 及其相关策略不仅是安全的基础，更是众多（资源，服务，环境）隔离机制的基础；在员工访问企业内部应用的场景下，Identity 及其相关策略提供了灵活的机制来提供随时随地的接入服务。无论那种场景下，Identity 对分布式系统中不同实体划分了有效的安全边界，并提供隔离机制来提高资源效能和用户体验。

07

云原生开发者工具——Cloud Toolkit

倪超，花名银时，阿里云中间件&容器产品专家

Cloud Toolkit 是本地 IDE 插件，帮助开发者更高效地开发、测试、诊断并部署应用。在 双11 紧张的大促备战中，让我们的开发人员更便捷、更高效地进行开发和部署，降低大促准备的成本，减少开发者繁琐复杂的操作，极大地提高工作效率和迎战的信心。

通过插件，开发者可以：

- 一键部署本地 IDE 内项目到任意远程服务器
- 一键部署本地 IDE 内项目到阿里云 EDAS、SAE 和 Kubernetes
- 本地 Docker Image 打包和仓库推送工具
- 远程服务器实时日志查看
- 阿里云小程序开发工具
- 阿里云函数计算开发工具
- 阿里云 RDS 内置 SQL 执行器
- 内置 Terminal 终端
- 文件上传
- Apache Dubbo 框架项目模板&代码生成
- Java 程序诊断工具
- RPC 服务端云联调

本文，将重点介绍如何通过 Cloud Toolkit 插件，帮助开发者 一键部署到镜像服务 ACR 和 容器服务 ACK 和 管理函数计算。

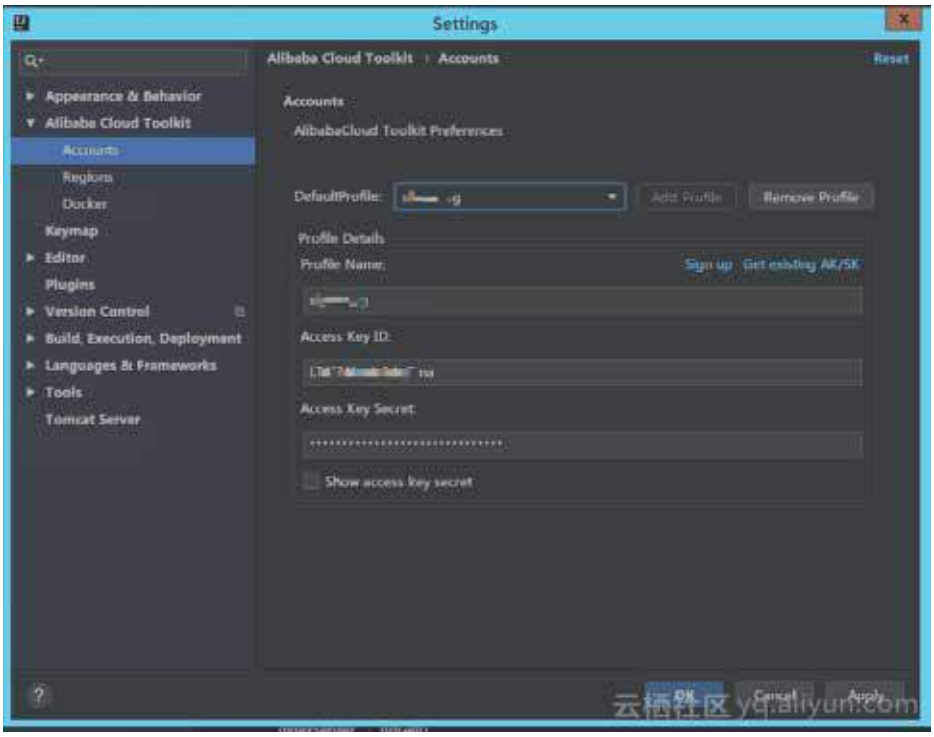
Docker应用利器：一键打通 ACR，并快速部署到 ACK

Docker 使得容器镜像成为软件及运行环境交付的事实标准，从而使 docker 容器成为一种事实标准的、轻量的、可以细粒度限制资源的沙箱。轻量的沙箱就意味着 docker 容器天生适合少量进程，那么在独占一个 pid namespace 的容器中，1 号进程自然就是 Dockerfile 或者启动容器的命令行中指定的进程。 然而，当年第一批吃螃蟹尝 docker 的人很快发现，没有 init system 的 docker 容器会带来两个令人无法忽视的问题：

开发者的部署包从形成镜像到镜像仓库，手动操作的话，每一次都需要经历上图4个步骤：登录阿里云 Docker Registry --> 从 Registry 中拉取镜像 --> 将镜像推送到 Registry --> 选择合适的镜像仓库地址，但是，使用 Cloud Toolkit ，开发者可以实现在本地 IDE 就能一键部署到镜像仓库。

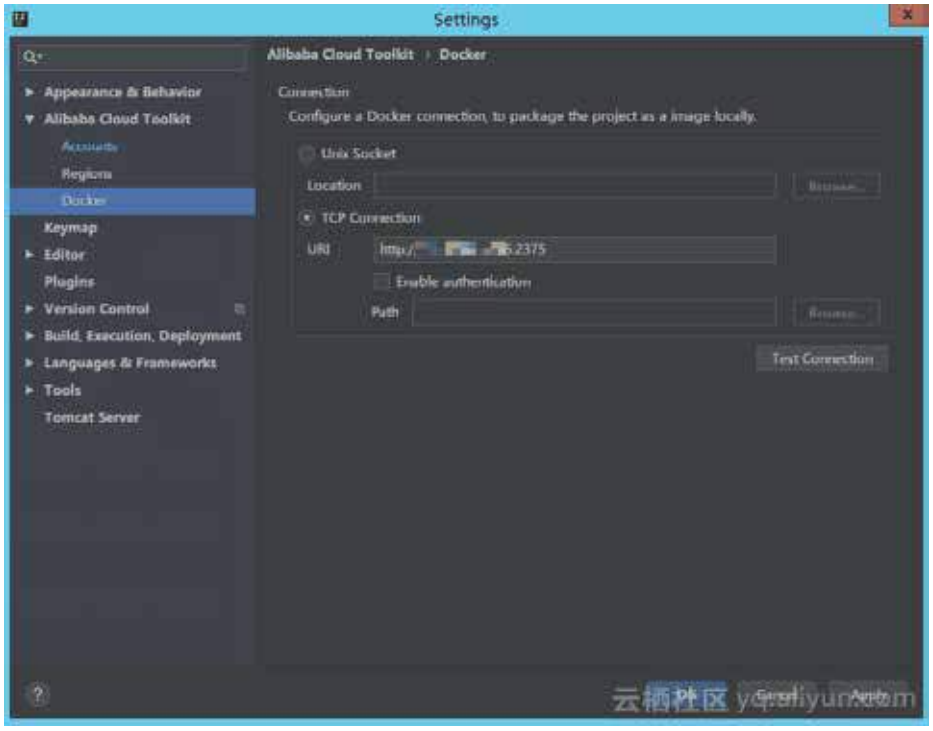
（一）配置插件首选项

点击：顶部菜单 Tools --> Alibaba Cloud Toolkit --> Preferences-->左边列表的 Alibaba Cloud Toolkit--> Accounts ，出现如下界面，配置阿里云账号的 AK 和 SK，即可完成首选项配置。
（如果是子账号，则填写子账号的 AK 和 SK）



（二）设置本地 Docker 镜像打包

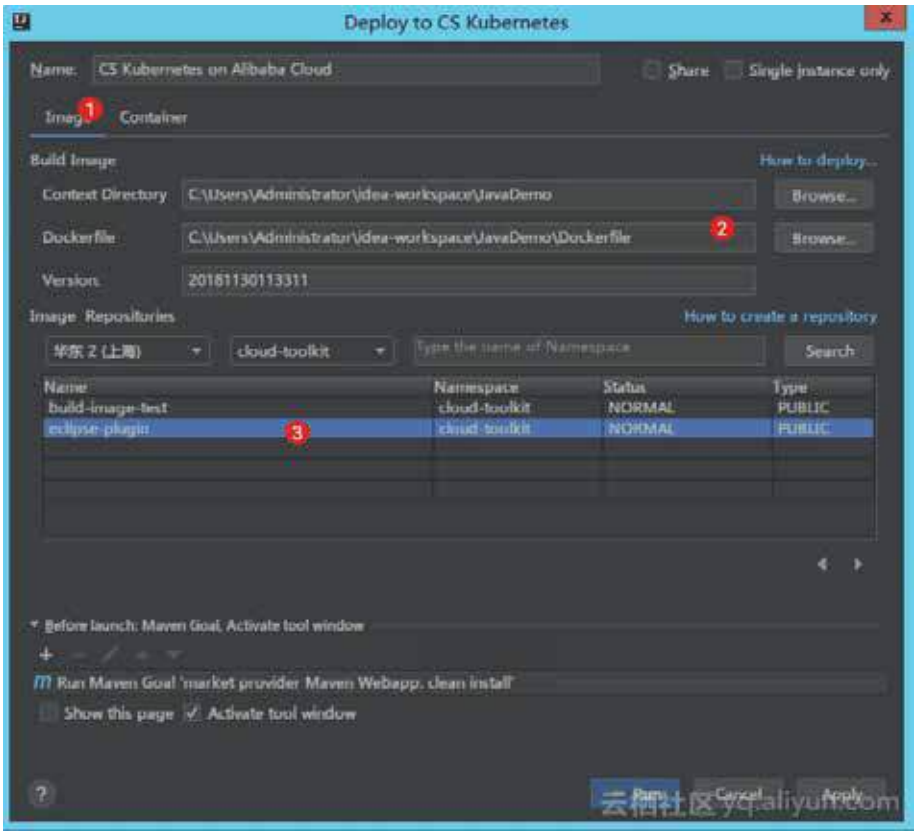
点击：顶部菜单 Tools --> Alibaba Cloud Toolkit --> Preferences --> 左边列表的 Alibaba Cloud Toolkit --> Docker，如下图，设置本地 Docker 镜像打包。



（三）部署应用

在 Intellij IDEA 中，鼠标右键项目工程名，在出现的菜单中点击 Alibaba Cloud --> Deploy to CS Kubernetes...，出现如下部署窗口：

第一步：设置 Image

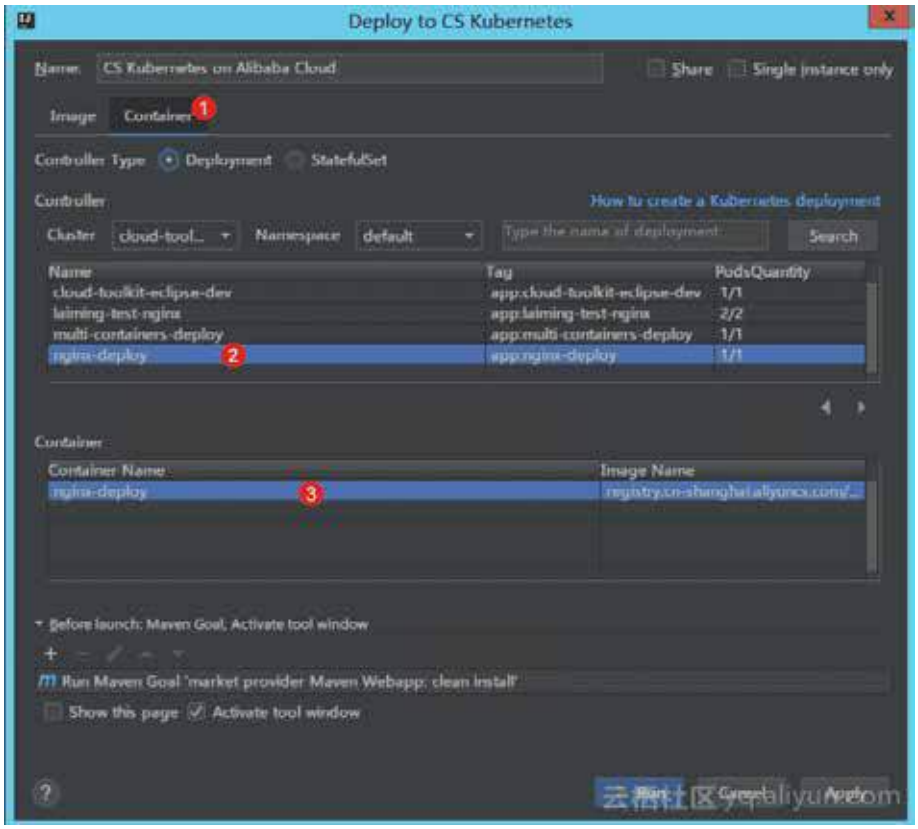


在 Image 标签页中，选择本地应用程序的 Context Directory 和 Dockerfile (通常会根据您本地的应用工程自动识别并设置)。

选择容器镜像服务的地域、命名空间和镜像仓库，然后单击 Container 标签页。

（说明：如果您还没有镜像仓库，在对话框右上角单击 Create a new repository 跳转到容器镜像仓库创建镜像仓库，创建步骤请参考容器镜像仓库文档。）

第二步：设置 Container



在 Container 标签页，选择容器服务 Kubernetes 的 Deployment（部署）、Clusters（集群）和Namespace（命名空间）。

选择指定的 Container（容器）。

（说明：如果您还没有创建容器服务 Kubernetes 的 Deployment，在对话框右上角单击 Create a new Kubernetes deployment，跳转到容器服务 Kubernetes 控制台创建 Deployment，创建步骤请参考容器服务 Kubernetes 版文档）

第三步：执行部署

点击 Run 按钮之后，即可完成本地应用程序向容器服务 Kubernetes 的部署。

管理函数计算

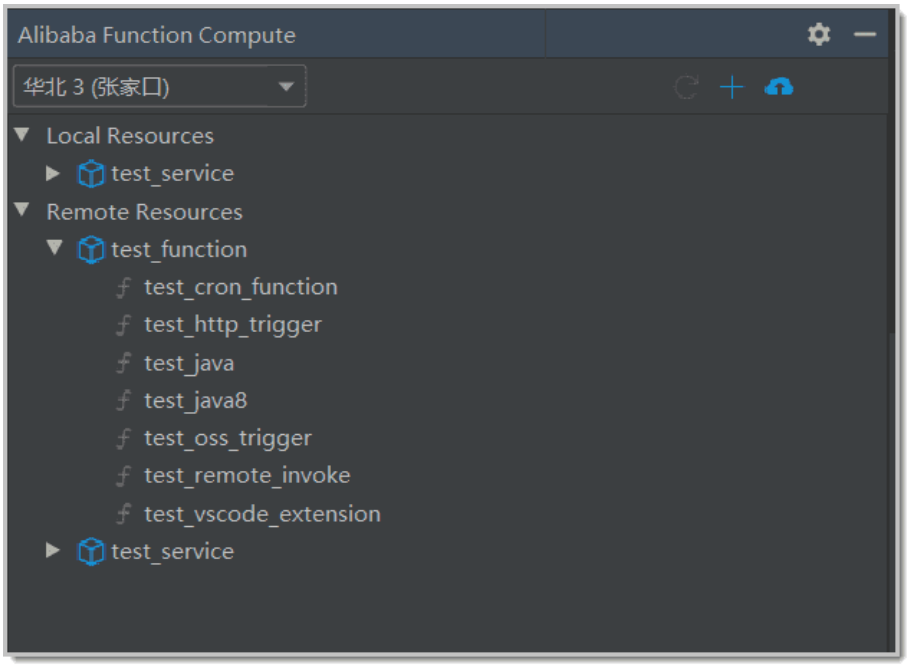
Cloud Toolkit 与函数计算实现了数据打通，安装被配置 Cloud Toolkit 后可以在本地 IDE 中查看该账号下的函数信息。安装 Fun 工具后可以在本地 IDE 中创建、运行、调试和部署函数，还可以运行和下载云端的函数。

第一步：选择地域

1.打开 IntelliJ IDEA。

2.在右侧导航栏中单击 Alibaba Function Compute 页签。

3.在 Alibaba Function Compute 页签中选择地域，然后按需对本地资源或云端资源进行操作：



第二步：展开 Local Resources进行本地资源操作：

- 创建资源：在 Alibaba Function Compute 右上角单击加号，在 Creat Function 对话框中配置创建信息，然后单击 Add。
- 运行本地函数：右键单击目标函数，选择 Local Run。
- 调试本地函数：右键单击目标函数，选择 Local Debug。
  部署 Service 中的所有函数：右键单击目标 Service，选择 Deploy Service。
- 部署单个函数：右键单击目标函数，选择 Deploy Function。

第三步：展开 Remote Resources 进行云端资源操作：

- 运行远端函数：右键单击目标函数，选择　Remote Run。
  下载 Service 中的所有函数：右键单击目标 Service，选择 Import To Local。
- 下载单个函数：右键单击目标函数，选择 Import To Local。
  查看 Service 性能：右键单击目标 Service，选择 Properties。
- 查看函数性能：右键单击目标函数，选择　Properties。

第四步：结果验证

执行下载、运行和部署等操作时，IntelliJ IDEA 的 Console 区域会打印操作日志，请根据日志信息检查部署结果。

阿里云 CLOUD NATIVE COMPUTING FOUNDATION

首个为中国开发者量身打造的云原生课程
《CNCF x Alibaba 云原生技术公开课》

由阿里云与 CNCF（Cloud Native Computing Foundation）共同开发的《CNCF x Alibaba 云原生技术公开课》与2019 年 4月正式上线。来自全球“云原生”技术社区的亲历者和领军人物，齐聚“课堂”为每一位中国开发者讲解和剖析关于“云原生”的方方面面，一步步揭示这次云计算变革背后的技术思想和本质。

《CNCF x Alibaba 云原生技术公开课》，也是 CNCF 旗下首个为中国开发者量身打造的云原生课程。这门课程完全免费且无需注册，旨在让广大中国开发者可以近距离聆听世界级技术专家解析云原生技术，让“云原生”技术真正触手可及。

微信或钉钉扫描二维码
直接开始学习

适合人群

- 计算机科学、软件工程等领域的软件工程师和大学生
- 使用/尝试使用容器和 Kubernetes 技术的应用程序开发者
- 具有基本服务器端知识、正在探索容器技术的软件开发者和技术管理者
- 希望理解云原生技术栈基本原理的技术管理者和开发者

你可以收获什么？

- 完善的知识体系，打造属于自己的云原生技能树
- 理解云原生技术背后的思想与本质
- 与知识体系相辅相成的动手实践
- 一线技术团队云原生技术最佳实践
- CNCF 与阿里云联合颁发课程结业证书

Alibaba Group
阿里巴巴集团

微信扫一扫关注
阿里巴巴云原生公众号

微信扫一扫关注
阿里技术公众号

阿里云开发者社区
开发者一站式平台

钉钉扫描二维码，
立即加入 K8s 社区大群