

The Cloud-native  
Architecture  
White Paper by  
Alibaba Cloud

阿里云

# 云原生 架构

白皮书

生于云 长于云 爆发于云  
企业数字化转型最短路径

## 编写说明

编写单位：阿里云计算有限公司

## 顾问组成员

阿里云

蒋江伟 贾扬清 丁宇

蚂蚁集团

何征宇

## 编写组成员（按目录排序）

阿里云

李小平 易立 司徒放 杨浩然 李响

罗毅 张磊 李艳林 张璦珩 赵燕标

赵林 谢纯良 邱戈川 员海滨 崔飞飞

张勇 付宇轩 许晓斌 雷卷

特步

王海能

## 定义贡献者（征集活动优选）

陈启涛、莫哈韦、郑富林、王炜、杨树宇、李皓伟

## 云原生架构

### 3 序

## 1

### 为什么需要云原生架构？

## 2

### 云原生架构的定义

- 5 云原生架构定义
- 7 云原生架构原则
- 9 主要架构模式
- 13 典型的云原生架构反模式

## 3

### 主要云原生技术

- 15 容器技术
- 18 云原生微服务
- 23 Serverless
- 27 开放应用模型（OAM）
- 31 Service Mesh 技术
- 33 DevOps
- 38 云原生中间件

## 4

### 阿里巴巴云原生架构设计

- 40 ACNA（Alibaba Cloud Native Architecting）架构设计方法
- 41 企业战略视角
- 41 业务发展视角
- 42 组织能力视角
- 42 云原生技术架构视角
- 43 架构持续演进闭环
- 44 云原生架构成熟度模型

## 5

### 阿里云云原生产品介绍

- 45 云原生产品家族
- 45 容器产品家族
- 46 微服务产品家族
- 47 Serverless 产品家族
- 47 Service Mesh 产品家族
- 48 消息产品家族
- 48 云原生数据库产品家族
- 49 云原生数仓产品家族

## 6

### 云原生架构实践案例

- 50 案例一：申通快递核心业务系统云原生上云案例
- 53 案例二：完美日记电商业务案例
- 55 案例三：特步业务中台案例（零售、公共云）
- 57 案例四：中国联通号卡业务云化案例（传统业务，专有云）
- 60 案例五：Timing App 的 Serverless 实践案例

## 7

### 云原生架构未来发展趋势

- 63 容器技术发展趋势
- 67 基于云原生的新一代应用编程界面
- 68 Serverless 发展趋势

# 序

# Cloud-native Architecture

The Cloud-native Architecture  
White Paper by Alibaba Cloud

回顾过去十年，数字化转型将科技创新与商业元素不断融合、重构，重新定义了新业态下的增长极。商业正在从大工业时代的固化范式进化成面向创新型商业组织与新商业物种的崭新模式。随着数字化转型在中国各行业广泛深入，不管是行业巨头，还是中小微企业都不得不面对数字化变革带来的未知机遇与挑战。

数字化转型的十年，也是云计算高速发展的十年，这期间新技术不断演进、优秀开源项目大量涌现，云原生领域进入“火箭式”发展阶段。通过树立技术标准与构建开发者生态，开源将云计算实施逐渐标准化，大幅降低了开发者对于云平台的学习成本与接入成本。这都让开发者更加聚焦于业务本身并借助云原生技术与产品实现更多业务创新，有效提升企业增长效率，爆发出前所未有的生产力与创造力。

可以说，当云计算重构整个IT产业的同时，也赋予了企业崭新的增长机遇。正如集装箱的出现加速了贸易全球化进程，以容器为代表的云原生技术作为云计算的服务新界面加速云计算普及的同时，也在推动着整个商业世界飞速演进。上云成为企业持续发展的必然选择，全面使用开源技术、云服务构建软件服务的时代已经到来。作为云时代释放技术红利的新方式，云原生技术在通过方法论、工具集和最佳实践重塑整个软件技术栈和生命周期，云原生架构

对云计算服务方式与互联网架构进行整体性升级，深刻改变着整个商业世界的IT根基。

虽然云原生概念的产生由来已久，但对于云原生的定义、云原生架构的理解却众说纷纭。到底什么是云原生？容器就代表云原生吗？云原生时代互联网分布式架构如何发展？云原生与开源、云计算有什么关系？开发者和企业为什么一定要选择云原生架构？面对这些问题，每个人都有着不同回答。鉴于此，阿里云结合自身云原生开源、云原生技术、云原生产品、云原生架构以及企业客户上云实践经验，给出了自己的答案，并通过这本白皮书与社会分享自己的思考与总结，旨在帮助越来越多的企业顺利找到数字化转型最短路径。

未来十年，云计算将无处不在，像水电煤一样成为数字经济时代的基础设施，云原生让云计算变得标准、开放、简单高效、触手可及。如何更好地拥抱云计算、拥抱云原生架构、用技术加速创新，将成为企业数字化转型升级成功的关键。

云计算的下一站，就是云原生；

IT架构的下一站，就是云原生架构；

希望所有的开发者、架构师和技术决策者们，共同定义、共同迎接云原生时代。

# 1



## 为什么需要云原生架构？

### 发展背景

计算机软件技术架构进化有两大主要驱动因素，一个是底层硬件升级，另一个是顶层业务发展诉求。正如伴随着 x86 硬件体系的成熟，很多应用不再使用昂贵、臃肿的大中型机，转而选择价格更为低廉的以 x86 为主的硬件体系，也由此诞生了包括 CORBA、EJB、RPC 在内的各类分布式架构；后由于互联网业务飞速发展，人们发现传统 IOE 架构已经不能满足海量业务规模的并发要求，于是又诞生了阿里巴巴 Dubbo & RocketMQ、Spring Cloud 这样的互联网架构体系。

云计算从工业化应用到如今，已走过十五个年头，然而大量应用使用云的方式仍停滞在传统 IDC 时代：虚拟机代替了原来的物理机；使用文件保存应用数据，大量自带的三方技术组件，没有经过架构改造（如微服务改造）的应用上云，传统的应用打包与发布方式等等。对于如何使用这些技术，没有绝对的对与错，只是在云的时代不能充分利用云的强大能力，不能从云技术中获得更高的可用性与可扩展能力，也不能利用云提升发布和运维的效率，是一件非常遗憾的事情。

回顾近年来商业世界的发展趋势，数字化转型的出现使得企业中越来越多的业务演变成数字化业务，数字化对于业务渠道、竞争格局、用户体验等诸多方面都带来更加严苛的要求，这就要求技术具备更快的迭代速度，业务推出速度从按周提升到按小时，每月上线业务量从“几十 / 月”提升到“几百 / 天”。大量数字化业务重构了企业的业务流水线，企业要求这些业务不能有不可接受的业务中断，否则会对客户体验以及营收可能造成巨大影响。

对于企业的 CIO 或者 IT 主管而言，原来企业内部 IT 建设以“烟囱”模式比较多，每个部门甚至每个应用都相对独立，如何管理与分配资源成了难题。大家都基于最底层 IDC 设施独自向上构建，都需要单独分配硬件资源，这就造成资源被大量占用且难以被共享。但是上云之后，由于云厂商提供了统一的 IaaS 能力和云服务，大幅提升了企业 IaaS 层的复用程度，CIO 或者 IT 主管自然而然想到 IaaS 以上层的系统也需要被统一，使资源、产品可被不断复用，从而能够进一步降低企业运营成本。

所有这些问题都指向一个共同点，那就是云的时代需要新的技术架构，来帮助企业应用能够更好地利用云计算优势，充分释放云计算的技术红利，让业务更敏捷、成本更低的同时又可伸缩性更灵活，而这些正好就是云原生架构专注解决的技术点。

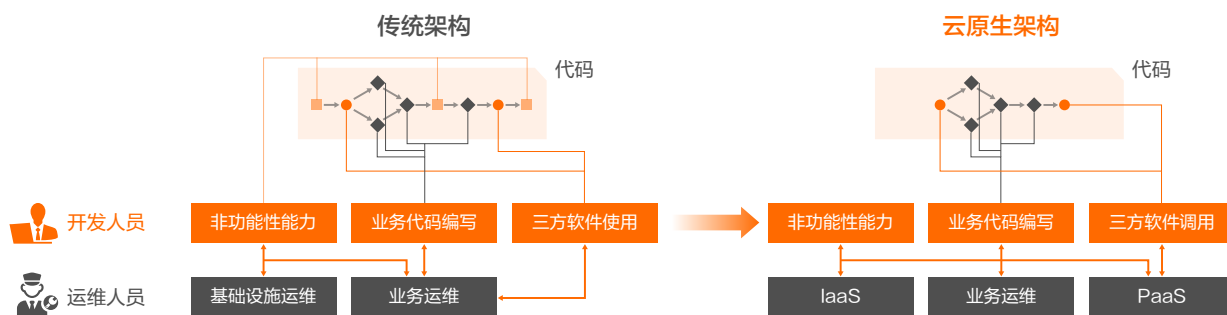
# 2 云原生架构的定义

今天云原生的定义有众多版本，云原生架构的理解也不尽相同，阿里将根据自身的云原生技术、产品和上云实践，给出自己的理解。

## 1 云原生架构定义



从技术的角度，云原生架构是基于云原生技术的一组架构原则和设计模式的集合，旨在将云应用中的非业务代码部分进行最大化的剥离，从而让云设施接管应用中原有的大量非功能特性(如弹性、韧性、安全、可观测性、灰度等)，使业务不再有非功能性业务中断困扰的同时，具备轻量、敏捷、高度自动化的特点。



云原生架构与传统架构的对比

上图展示了在代码中通常包括三部分：业务代码、三方软件、处理非功能特性的代码。其中“业务代码”指实现业务逻辑的代码；“三方软件”是业务代码中依赖的所有三方库，包括业务库和基础库；“处理非功能性的代码”指实现高可用、安全、可观测性等非功能性能力的代码。

三部分中只有业务代码是核心，是对业务真正带来价值的，另外两个部分都只算附属物，但随着软件规模的增大、业务模块规模变大、部署环境增多、分布式复杂性增强，使得今天的软件构建变得越来越复杂，

对开发人员的技能要求也越来越高。云原生架构相比较传统架构进了一大步，从业务代码中剥离了大量非功能性特性（不会是所有，比如易用性还不能剥离）到 IaaS 和 PaaS 中，从而减少业务代码开发人员的关注范围，通过云厂商的专业性提升应用的非功能性能力。

此外，具备云原生架构的应用可以最大程度利用云服务和提升软件交付能力，进一步加快软件开发：

## 1 代码结构发生巨大变化

云原生架构产生的最大影响就是让开发人员的编程模型发生了巨大变化。今天大部分的编程语言中，都有文件、网络、线程等元素，这些元素为充分利用单机资源带来好处的同时，也提升了分布式编程的复杂性；因此大量框架、产品涌现，来解决分布式环境中的网络调用问题、高可用问题、CPU 争用问题、分布式存储问题 ……

在云的环境中，比如“如何获取存储”变成了若干服务，包括对象存储服务、块存储服务和没有随机访问的文件存储服务。云不仅改变了开发人员获得这些存储能力的界面，还在于云产品在这些 OpenAPI 或者开源 SDK 背后把分布式场景中的高可用挑战、自动扩缩容挑战、安全挑战、运维升级挑战等都处理了，应用开发人员就不用在其代码中处理节点宕机前如何把本地保存的内容同步到远端的问题，也不用处理当业务峰值到来时如何对存储节点进行扩容的问题，而应用的运维人员不用在发现 zero day 安全问题时紧急对三方存储软件进行升级 ……

云把三方软硬件的能力升级成了服务，开发人员的开发复杂度和运维人员的运维工作量都得到极大降低。显然，如果这样的云服务用得越多，那么开发和运维人员的负担就越少，企业在非核心业务实现上从必须的负担变成了可控支出。在一些开发能力强的公司中，对这些三方软硬件能力的处理往往是交给应用框架（或者说公司内自己的中间件）来做的；在云的时代云厂商提供了更具 SLA 的服务，使得所有软件公司都可以由此获益。

这些使得业务代码的开发人员技能栈中，不再需要掌握文件及其分布式处理技术，不再需要掌握各种复杂的网络技术 …… 简化让业务开发变得更敏捷、更快速！

## 2 非功能性特性的大量委托

任何应用都提供两类特性，功能性特性和非功能性特性。功能性特性是真正为业务带来价值的代码，比如如何建立客户资料、如何处理订单、如何支付等等；即使是一些通用的业务功能特性，比如组织管理、业务字典管理、搜索等等也是紧贴业务需求的。非功能性特性是没有给业务带来直接业务价值，但通常又是必不可少的特性，比如高可用能力、容灾能力、安全特性、可运维性、易用性、可测试性、灰度发布能力等等。

不能说云计算解决了所有非功能性问题，但确实大量非功能性特性，特别是分布式环境下复杂非功能性问题，被云产品处理掉了。以大家最头疼的高可用为例，云产品在多个层面为应用提供了解决方案：

**虚机：**当虚机检测到底层硬件异常时，自动帮助应用做热迁移，迁移后的应用不需重新启动而仍然具备对外服务的能力，应用对整个迁移过程都不会有任何感知；

**容器：**有时应用所在的物理机是正常的，只是应用自身的问题（比如 bug、资源耗尽等）而无法正常对外提供服务。容器通过监控检查探测到进程状态异常，从而实施异常节点的下线、新节点上线和生产流量的切换等操作，整个过程自动完成而无需运维人员干预；

**云服务：**如果应用把“有状态”部分都交给了云服务（如缓存、数据库、对象存储等），加上全局对象的持有小型化或具备从磁盘快速重建能力，由于云服务本身是具备极强的高可用能力，那么应用本身会变成更薄的“无状态”应用，因为高可用故障带来的业务中断会降至分钟级；如果应用是 N-M 的对等架构架构模式，那么结合 Load Balancer 产品可获得几乎无损的高可用能力！

### 3 高度自动化的软件交付

软件一旦开发完成，需要在公司内外部各类环境中部署和交付，以将软件价值交给最终客户。软件交付的困难在于开发环境到生产环境的差异（公司环境到客户环境之间的差异）以及软件交付和运维人员的技能差异，填补这些差异的是一大堆安装手册、运维手册和培训文档。容器以一种标准的方式对软件打包，容器及相关技术则帮助屏蔽不同环境之间的差异，进而基于容器做标准化的软件交付。

对自动化交付而言，还需要一种能够描述不同环境的工具，让软件能够“理解”目标环境、交付内容、配置清单并通过代码去识别目标环境的差异，根据交付内容以“面向终态”的方式完成软件的安装、配置、运行和变更。

基于云原生的自动化软件交付相比较当前的人工软件交付是一个巨大的进步。以微服务为例，应用微服务化以后，往往被部署到成千上万个节点上，如果系统不具备高度的自动化能力，任何一次新业务的上线，都会带来极大的工作量挑战，严重时还会导致业务变更超过上线窗口而不可用。

## 2 云原生架构原则



云原生架构本身作为一种架构，也有若干架构原则作为应用架构的核心架构控制面，通过遵从这些架构原则可以让技术主管和架构师在做技术选择时不会出现大的偏差。



## 1 服务化原则

当代码规模超出小团队的合作范围时，就有必要进行服务化拆分了，包括拆分为微服务架构、小服务（Mini Service）架构，通过服务化架构把不同生命周期的模块分离出来，分别进行业务迭代，避免迭代频繁模块被慢速模块拖慢，从而加快整体的进度和稳定性。同时服务化架构以面向接口编程，服务内部的功能高度内聚，模块间通过公共功能模块的提取增加软件的复用程度。

分布式环境下的限流降级、熔断隔仓、灰度、反压、零信任安全等，本质上都是基于服务流量（而非网络流量）的控制策略，所以云原生架构强调使用服务化的目的还在于从架构层面抽象化业务模块之间的关系，标准化服务流量的传输，从而帮助业务模块进行基于服务流量的策略控制和治理，不管这些服务是基于什么语言开发的。

## 2 弹性原则

大部分系统部署上线需要根据业务量的估算，准备一定规模的机器，从提出采购申请，到供应商洽谈、机器部署上电、软件部署、性能压测，往往需要好几个月甚至一年的周期；而这期间如果业务发生了变化了，重新调整也非常困难。弹性则是指系统的部署规模可以随着业务量的变化自动伸缩，无须根据事先的容量规划准备固定的硬件和软件资源。好的弹性能力不仅缩短了从采购到上线的时间，让企业不用操心额外软硬件资源的成本支出（闲置成本），降低了企业的 IT 成本，更关键的是当业务规模面临海量突发性扩张的时候，不再因为平时软硬件资源储备不足而“说不”，保障了企业收益。

## 3 可观测原则

今天大部分企业的软件规模都在不断增长，原来单机可以对应用做完所有调试，但在分布式环境下需要对多个主机上的信息做关联，才可能回答清楚服务为什么宕机、哪些服务违反了其定义的 SLO、目前的故障影响哪些用户、最近这次变更对哪些服务指标带来了影响等等，这些都要求系统具备更强的可观测能力。可观测性与监控、业务探查、APM 等系统提供的能力不同，前者是在云这样的分布式系统中，主动通过日志、链路跟踪和度量等手段，让一次 APP 点击背后的多次服务调用的耗时、返回值和参数都清晰可见，甚至可以下钻到每次三方软件调用、SQL 请求、节点拓扑、网络响应等，这样的能力可以使运维、开发和业务人员实时掌握软件运行情况，并结合多个维度的数据指标，获得前所未有的关联分析能力，不断对业务健康度和用户体验进行数字化衡量和持续优化。

## 4 韧性原则

当业务上线后，最不能接受的就是业务不可用，让用户无法正常使用软件，影响体验和收入。韧性代表了当软件所依赖的软硬件组件出现各种异常时，软件表现出来的抵御能力，这些异常通常包括硬件故障、

硬件资源瓶颈（如 CPU/ 网卡带宽耗尽）、业务流量超出软件设计能力、影响机房工作的故障和灾难、软件 bug、黑客攻击等业务不可用带来致命影响的因素。

韧性从多个维度诠释了软件持续提供业务服务的能力，核心目标是提升软件的 MTBF（Mean Time Between Failure，平均无故障时间）。从架构设计上，韧性包括服务异步化能力、重试 / 限流 / 降级 / 熔断 / 反压、主从模式、集群模式、AZ 内的高可用、单元化、跨 region 容灾、异地多活容灾等。

## 5 所有过程自动化原则

技术往往是把“双刃剑”，容器、微服务、DevOps、大量第三方组件的使用，在降低分布式复杂性和提升迭代速度的同时，因为整体增大了软件技术栈的复杂度和组件规模，所以不可避免地带来了软件交付的复杂性，如果这里控制不当，应用就无法体会到云原生技术的优势。通过 IaC（Infrastructure as Code）、GitOps、OAM（Open Application Model）、Kubernetes operator 和大量自动化交付工具在 CI/CD 流水线中的实践，一方面标准化企业内部的软件交付过程，另一方面在标准化的基础上进行自动化，通过配置数据自描述和面向终态的交付过程，让自动化工具理解交付目标和环境差异，实现整个软件交付和运维的自动化。

## 6 零信任原则

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。

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。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。

## 7 架构持续演进原则

今天技术和业务的演进速度非常快，很少有一开始就清晰定义了架构并在整个软件生命周期里面都适用，相反往往还需要对架构进行一定范围内的重构，因此云原生架构本身也应该和必须是一个具备持续演进能力的架构，而不是一个封闭式架构。除了增量迭代、目标选取等因素外，还需要考虑组织（例如架构控制委员会）层面的架构治理和风险控制，特别是在业务高速迭代情况下的架构、业务、实现平衡

关系。云原生架构对于新建应用而言的架构控制策略相对容易选择(通常是选择弹性、敏捷、成本的维度),但对于存量应用向云原生架构迁移,则需要从架构上考虑遗留应用的迁出成本/风险和到云上的迁入成本/风险,以及技术上通过微服务/应用网关、应用集成、适配器、服务网格、数据迁移、在线灰度等应用和流量进行细颗粒度控制。

## 3 主要架构模式



云原生架构有非常多的架构模式,这里选取一些对应用收益更大的主要架构模式进行讨论。

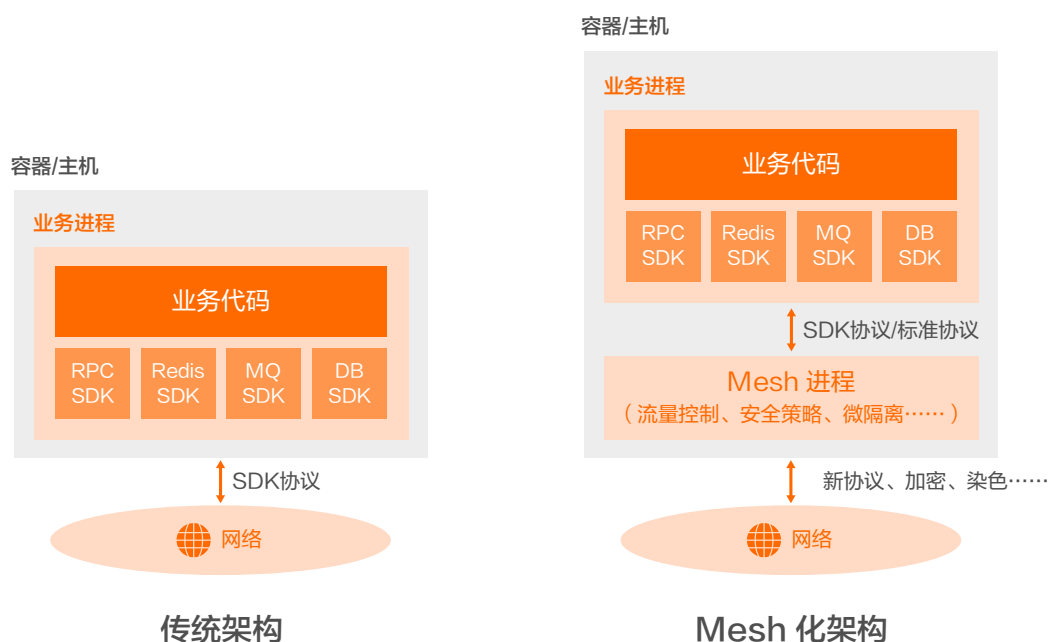
### 1 服务化架构模式

服务化架构是云时代构建云原生应用的标准架构模式,要求以应用模块为颗粒度划分一个软件,以接口契约(例如 IDL)定义彼此业务关系,以标准协议(HTTP、gRPC 等)确保彼此的互联互通,结合 DDD(领域模型驱动)、TDD(测试驱动开发)、容器化部署提升每个接口的代码质量和迭代速度。服务化架构的典型模式是微服务和小服务(Mini Service)模式,其中小服务可以看做是一组关系非常密切的服务的组合,这组服务会共享数据,小服务模式通常适用于非常大型的软件系统,避免接口的颗粒度太细而导致过多的调用损耗(特别是服务间调用和数据一致性处理)和治理复杂度。

通过服务化架构,把代码模块关系和部署关系进行分离,每个接口可以部署不同数量的实例,单独扩缩容,从而使得整体的部署更经济。此外,由于在进程级实现了模块的分离,每个接口都可以单独升级,从而提升了整体的迭代效率。但也需要注意,服务拆分导致要维护的模块数量增多,如果缺乏服务的自动化能力和治理能力,会让模块管理和组织技能不匹配,反而导致开发和运维效率的降低。

### 2 Mesh 化架构模式

Mesh 化架构是把中间件框架(比如 RPC、缓存、异步消息等)从业务进程中分离,让中间件 SDK 与业务代码进一步解耦,从而使得中间件升级对业务进程没有影响,甚至迁移到另外一个平台的中间件也对业务透明。分离后在业务进程中只保留很“薄”的 Client 部分,Client 通常很少变化,只负责与 Mesh 进程通讯,原来需要在 SDK 中处理的流量控制、安全等逻辑由 Mesh 进程完成。整个架构如下图所示。



实施 Mesh 化架构后，大量分布式架构模式（熔断、限流、降级、重试、反压、隔仓……）都由 Mesh 进程完成，即使在业务代码的制品中并没有使用这些三方软件包；同时获得更好的安全性（比如零信任架构能力）、按流量进行动态环境隔离、基于流量做冒烟 / 回归测试等。

### 3 Serverless 模式

和大部分计算模式不同，Serverless 将“部署”这个动作从运维中“收走”，使开发者不用关心应用在哪里运行，更不用关心装什么 OS、怎么配置网络、需要多少 CPU …… 从架构抽象上看，当业务流量到来 / 业务事件发生时，云会启动或调度一个已启动的业务进程进行处理，处理完成后云会自动会关闭 / 调度业务进程，等待下一次触发，也就是把应用的整个运行时都委托给云。

今天 Serverless 还没有达到任何类型的应用都适用的地步，因此架构决策者需要关心应用类型是否适合于 Serverless 运算。如果应用是有状态的，云在进行调度时可能导致上下文丢失，毕竟 Serverless 的调度不会帮助应用做状态同步；如果应用是长时间后台运行的密集型计算任务，会得不到太多 Serverless 的优势；如果应用涉及到频繁的外部 I/O（网络或者存储，以及服务间调用），也因为繁重的 I/O 负担、时延大而不适合。Serverless 非常适合于事件驱动的数据计算任务、计算时间短的请求 / 响应应用、没有复杂相互调用的长周期任务。

### 4 存储计算分离模式

分布式环境中的 CAP 困难主要是针对有状态应用，因为无状态应用不存在 C（一致性）这个维度，因此可以获得很好的 A（可用性）和 P（分区容错性），因而获得更好的弹性。在云环境中，推荐把各

类暂态数据（如 session）、结构化和非结构化持久数据都采用云服务来保存，从而实现存储计算分离。但仍然有一些状态如果保存到远端缓存，会造成交易性能的明显下降，比如交易会话数据太大、需要不断根据上下文重新获取等，则可以考虑通过采用 Event Log + 快照（或 Check Point）的方式，实现重启后快速增量恢复服务，减少不可用对业务的影响时长。

## 5 分布式事务模式

微服务模式提倡每个服务使用私有的数据源，而不是像单体这样共享数据源，但往往大颗粒度的业务需要访问多个微服务，必然带来分布式事务问题，否则数据就会出现不一致。架构师需要根据不同的场景选择合适的分布式事务模式。

- 传统采用 **XA 模式**，虽然具备很强的一致性，但是性能差；
- 基于消息的**最终一致性（BASE）**通常有很高的性能，但是通用性有限，且消息端只能成功而不能触发消息生产端的事务回滚；
- **TCC 模式**完全由应用层来控制事务，事务隔离性可控，也可以做到比较高效；但是对业务的侵入性非常强，设计开发维护等成本很高；
- **SAGA 模式**与 TCC 模式的优缺点类似但没有 try 这个阶段，而是每个正向事务都对应一个补偿事务，也是开发维护成本高；
- 开源项目 **SEATA 的 AT 模式**非常高性能且无代码开发工作量，且可以自动执行回滚操作，同时也存在一些使用场景限制。

## 6 可观测架构

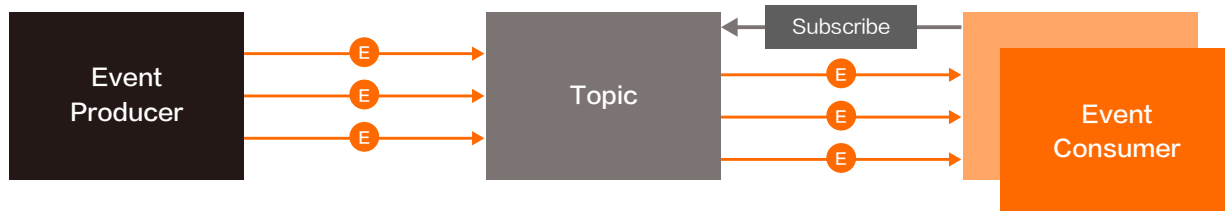
可观测架构包括 Logging、Tracing、Metrics 三个方面，其中 Logging 提供多个级别（verbose/debug/warning/error/fatal）的详细信息跟踪，由应用开发者主动提供；Tracing 提供一个请求从前端到后端的完整调用链路跟踪，对于分布式场景尤其有用；Metrics 则提供对系统量化的多维度度量。

架构决策者需要选择合适的、支持可观测的开源框架（比如 OpenTracing、OpenTelemetry），并规范上下文的可观测数据规范（例如方法名、用户信息、地理位置、请求参数等），规划这些可观测数据在哪些服务和技术组件中传播，利用日志和 tracing 信息中的 span id/trace id，确保进行分布式链路分析时有足够的信息进行快速关联分析。

由于建立可观测性的主要目标是对服务 SLO（Service Level Objective）进行度量，从而优化 SLA，因此架构设计上需要为各个组件定义清晰的 SLO，包括并发度、耗时、可用时长、容量等。

## 7 事件驱动架构

事件驱动架构（EDA，Event Driven Architecture）本质上是一种应用 / 组件间的集成架构模式，典型的事件驱动架构如下图：



事件和传统的消息不同，事件具有 schema，所以可以校验 event 的有效性，同时 EDA 具备 QoS 保障机制，也能够对事件处理失败进行响应。事件驱动架构不仅用于（微）服务解耦，还可应用于下面的场景中：

- **增强服务韧性**：由于服务间是异步集成的，也就是下游的任何处理失败甚至宕机都不会被上游感知，自然也就不会对上游带来影响；
- **CQRS (Command Query Responsibility Segregation)**：把对服务状态有影响的命令用事件来发起，而对服务状态没有影响的查询才使用同步调用的 API 接口；结合 EDA 中的 Event Sourcing 可以用于维护数据变更的一致性，当需要重新构建服务状态时，把 EDA 中的事件重新“播放”一遍即可；
- **数据变化通知**：在服务架构下，往往一个服务中的数据发生变化，另外的服务会感兴趣，比如用户订单完成后，积分服务、信用服务等都需要得到事件通知并更新用户积分和信用等级；
- **构建开放式接口**：在 EDA 下，事件的提供者并不关心有哪些订阅者，不像服务调用的场景——数据的产生者需要知道数据的消费者在哪里并调用它，因此保持了接口的开放性；
- **事件流处理**：应用于大量事件流（而非离散事件）的数据分析场景，典型应用是基于 Kafka 的日志处理；
- **基于事件触发的响应**：在 IoT 时代大量传感器产生的数据，不会像人机交互一样需要等待处理结果的返回，天然适合用 EDA 来构建数据处理应用。

## 4 典型的云原生架构反模式



技术往往像一把双刃剑，阿里在自己和帮助客户做云原生架构演进的时候，会充分考虑根据不同的场景选择不同的技术，下面是我们总结的一些典型云原生架构反模式。

### 1 庞大的单体应用

庞大单体应用的最大问题在于缺乏依赖隔离，包括代码耦合带来的责任不清、模块间接口缺乏治理而带来变更影响扩散、不同模块间的开发进度和发布时间要求难以协调、一个子模块不稳定导致整个应用都变慢、扩容时只能整体扩容而不能对达到瓶颈的模块单独扩容……因此当业务模块可能存在多人开发



的时候，就需要考虑通过服务化进行一定的拆分，梳理聚合根，通过业务关系确定主要的服务模块以及这些模块的边界、清晰定义模块之间的接口，并让组织关系和架构关系匹配。

阿里巴巴在淘宝业务快速发展阶段，就遇到过上百人维护一个核心单体应用，造成了源码冲突、多团队间协同代价高的严重问题。为了支撑不断增长的流量，该应用需要不断增加机器，很快后端数据库连接很快就到达了上限。在还没有“微服务”概念的 2008 年，阿里巴巴决定进行服务化架构拆分，当时思路就是微服务架构，第一个服务从用户中心开始建设，后续交易、类目、店铺、商品、评价中心等服务陆续从单体中独立出来，服务之间通过远程调用通信，每个中心由独立团队专门维护，从而解决了研发协同问题，以及按规模持续水平扩展的问题。

## 2 单体应用“硬拆”为微服务

服务的拆分需要适度，过分服务化拆分反而会导致新架构与组织能力的不匹配，让架构升级得不到技术红利，典型的例子包括：

- **小规模软件的服务拆分**：软件规模不大，团队人数也少，但是为了微服务而微服务，强行把耦合度高、代码量少的模块进行服务化拆分，一次性的发布需要拆分为多个模块分开发布和维护；
- **数据依赖**：服务虽然拆分为多个，但是这些服务的数据是紧密耦合的，于是让这些服务共享数据库，导致数据的变化往往被扇出到多个服务中，造成服务间数据依赖；
- **性能降低**：当耦合性很强的模块被拆分为多个微服务后，原来的本地调用变成了分布式调用，从而让响应时间变大了上千倍，导致整个服务链路性能急剧下降。

## 3 缺乏自动化能力的微服务

软件架构中非常重要的一个维度就是处理软件复杂度问题，一旦问题规模提升了很多，那就必须重新考虑与之适应的新方案。在很多软件组织中，开发、测试和运维的工作单位都是以进程为单位，比如把整个用户管理作为一个单独的模块进行打包、发布和运行；而进行了微服务拆分后，这个用户管理模块可能被分为用户信息管理、基本信息管理、积分管理、订单管理等多个模块，由于仍然是每个模块分别打包、发布和运行，开发、测试和运维人员的人均负责模块数就直线上升，造成了人均工作量增大，也就增加了软件的开发成本。

实际上，当软件规模进一步变大后，自动化能力的缺失还会带来更大的危害。由于接口增多会带来测试用例的增加，更多的软件模块排队等待测试和发布，如果缺乏自动化会造成软件发布时间变长，在多环境发布或异地发布时更是需要专家来处理环境差异带来的影响。同时更多的进程运行于一个环境中，缺乏自动化的人工运维容易给环境带来不可重现的影响，而一旦发生人为运维错误又不容易“快速止血”，造成了故障处理时间变长，以及使得日常运维操作都需要专家才能完成。所有这些问题导致软件交付时间变长、风险提升以及运维成本的增加。

# 3

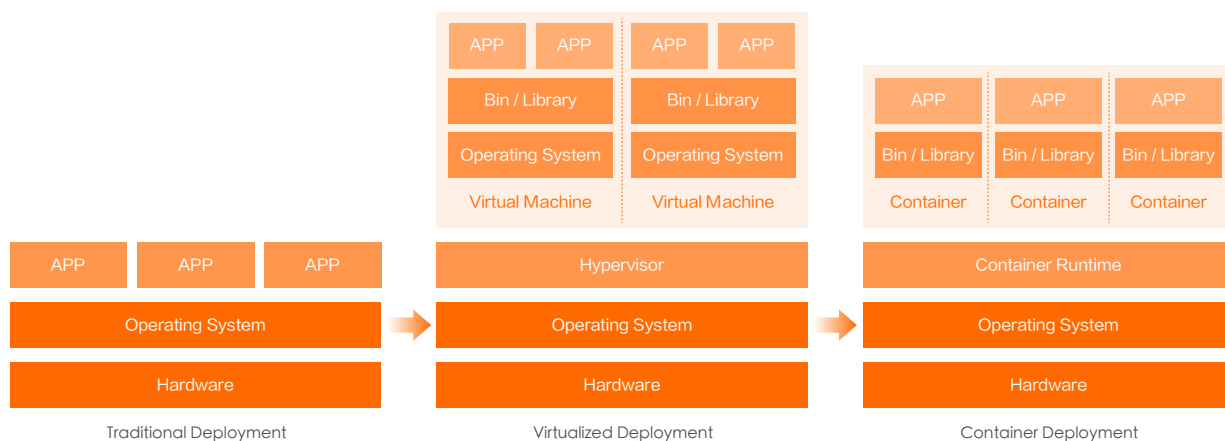
## 主要云原生技术

### 1 容器技术



#### 1 容器技术背景与价值

容器作为标准化软件单元，它将应用及其所有依赖项打包，使应用不再受环境限制，在不同计算环境中快速、可靠地运行。



传统、虚拟化及容器部署模式比较

虽然 2008 年 Linux 就提供了 Cgroups 资源管理机制、Linux Namespace 视图隔离方案，让应用得以运行在独立沙箱环境中，避免相互间冲突与影响；但直到 Docker 容器引擎的开源，才很大程度上降低了容器技术的使用复杂性，加速了容器技术普及。Docker 容器基于操作系统虚拟化技术，共享操作系统内核、轻量、没有资源损耗、秒级启动，极大提升了系统的应用部署密度和弹性。更重要的是，Docker 提出了创新的应用打包规范 —— Docker 镜像，解耦了应用与运行环境，使应用可以在不同计算环境间一致、可靠地运行。借助容器技术呈现了一个优雅的抽象场景：**让开发所需要的灵活性、开放性和运维所关注的标准化、自动化达成相对平衡**。容器镜像迅速成为了应用分发的工业标准。



随后开源的 Kubernetes，凭借优秀的开放性、可扩展性以及活跃开发者社区，在容器编排之战中脱颖而出，成为分布式资源调度和自动化运维的事实标准。Kubernetes 屏蔽了 IaaS 层基础架构的差异并凭借优良的可移植性，帮助应用一致地运行在包括数据中心、云、边缘计算在内的不同环境。企业可以通过 Kubernetes，结合自身业务特征来设计自身云架构，从而更好支持多云 / 混合云，免去被厂商锁定的顾虑。伴随着容器技术逐步标准化，进一步促进了容器生态的分工和协同。基于 Kubernetes，生态社区开始构建上层的业务抽象，比如服务网格 Istio、机器学习平台 Kubeflow、无服务器应用框架 Knative 等。

在过去几年，容器技术获得了越发广泛的应用的同时，三个核心价值最受用户关注：

#### • 敏捷

容器技术提升企业 IT 架构敏捷性的同时，让业务迭代更加迅捷，为创新探索提供了坚实的技术保障。比如疫情期间，教育、视频、公共健康等行业的在线化需求突现爆发性高速增长，很多企业通过容器技术适时把握了突如其来的业务快速增长机遇。据统计，使用容器技术可以获得 3~10 倍交付效率提升，这意味着企业可以更快速的迭代产品，更低成本进行业务试错。

#### • 弹性

在互联网时代，企业 IT 系统经常需要面对促销活动、突发事件等各种预期内外的爆发性流量增长。通过容器技术，企业可以充分发挥云计算弹性优势，降低运维成本。一般而言，借助容器技术，企业可以通过部署密度提升和弹性降低 50% 的计算成本。以在线教育行业为例，面对疫情之下指数级增长的流量，教育信息化应用工具提供商希沃 Seewo 利用阿里云容器服务 ACK 和弹性容器实例 ECI 大大满足了快速扩容的迫切需求，为数十万名老师提供了良好的在线授课环境，帮助百万学生进行在线学习。

#### • 可移植性

容器已经成为应用分发和交付的标准技术，将应用与底层运行环境进行解耦；Kubernetes 成为资源调度和编排的标准，屏蔽了底层架构差异性，帮助应用平滑运行在不同基础设施上。CNCF 云原生计算基金会推出了 Kubernetes 一致性认证，进一步保障了不同 K8s 实现的兼容性，这也让企业愿意采用容器技术来构建云时代应用基础设施。

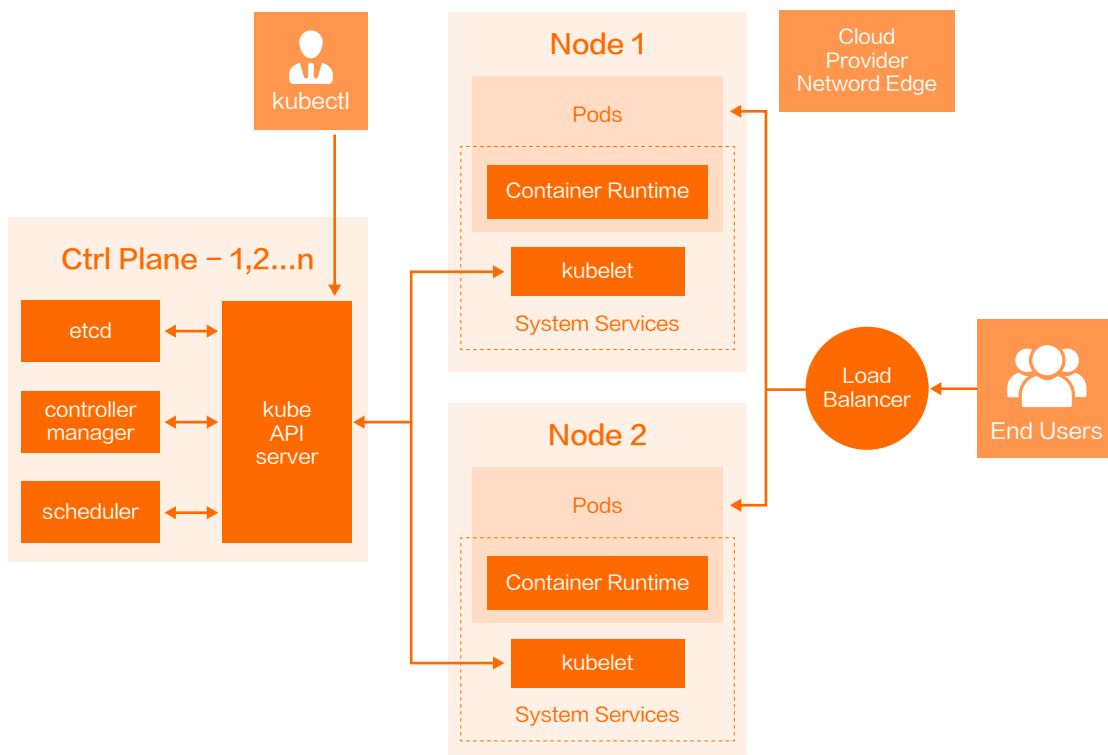
## 2 容器编排

Kubernetes 已经成为容器编排的事实标准，被广泛用于自动部署，扩展和管理容器化应用。Kubernetes 提供了分布式应用管理的核心能力：

- **资源调度**：根据应用请求的资源量 CPU、Memory，或者 GPU 等设备资源，在集群中选择合适的节点来运行应用；
- **应用部署与管理**：支持应用的自动发布与应用的回滚，以及与应用相关的配置的管理；也可以自动化存储卷的编排，让存储卷与容器应用的生命周期相关联；

- **自动修复**: Kubernetes 可以监测这个集群中所有的宿主机, 当宿主机或者 OS 出现故障, 节点健康检查会自动进行应用迁移; K8s 也支持应用的自愈, 极大简化了运维管理的复杂性;
- **服务发现与负载均衡**: 通过 Service 资源出现各种应用服务, 结合 DNS 和多种负载均衡机制, 支持容器化应用之间的相互通信;
- **弹性伸缩**: K8s 可以监测业务上所承担的负载, 如果这个业务本身的 CPU 利用率过高, 或者响应时间过长, 它可以对这个业务进行自动扩容。

Kubernetes 的控制平面包含四个主要的组件: API Server、Controller、Scheduler 以及 etcd。如下图所示:



Kubernetes 在容器编排中有几个关键设计理念:

- **声明式 API**: 开发者可以关注于应用自身, 而非系统执行细节。比如 Deployment (无状态应用)、StatefulSet (有状态应用)、Job (任务类应用) 等不同资源类型, 提供了对不同类型工作负载的抽象; 对 Kubernetes 实现而言, 基于声明式 API 的 “level-triggered” 实现比 “edge-triggered” 方式可以提供更加健壮的分布式系统实现。
- **可扩展性架构**: 所有 K8s 组件都是基于一致的、开放的 API 实现和交互; 三方开发者也可通过 CRD (Custom Resource Definition) / Operator 等方法提供领域相关的扩展实现, 极大提升了 K8s 的能力。

- **可移植性:** K8s 通过一系列抽象如 Loadbalance Service（负载均衡服务）、CNI（容器网络接口）、CSI（容器存储接口），帮助业务应用可以屏蔽底层基础设施的实现差异，实现容器灵活迁移的设计目标。

## 2 云原生微服务



### 1 微服务发展背景

过去开发一个后端应用最为直接的方式就是通过单一后端应用提供并集成所有的服务，即单体模式。随着业务发展与需求不断增加，单体应用功能愈发复杂，参与开发的工程师规模可能由最初几个人发展到十几人，应用迭代效率由于集中式研发、测试、发布、沟通模式而显著下滑。为了解决由单体应用模型衍生的过度集中式项目迭代流程，微服务模式应运而生。

微服务模式将后端单体应用拆分为松耦合的多个子应用，每个子应用负责一组子功能。这些子应用称为“微服务”，多个“微服务”共同形成了一个物理独立但逻辑完整的分布式微服务体系。这些微服务相对独立，通过解耦研发、测试与部署流程，提高整体迭代效率。此外，微服务模式通过分布式架构将应用水平扩展和冗余部署，从根本上解决了单体应用在拓展性和稳定性上存在的先天架构缺陷。但也要注意微服务模型也面临着分布式系统的典型挑战：如何高效调用远程方法、如何实现可靠的系统容量预估、如何建立负载均衡体系、如何面向松耦合系统进行集成测试、如何面向大规模复杂关联应用的部署与运维……

在云原生时代，云原生微服务体系将充分利用云资源的高可用和安全体系，让应用获得更有保障的弹性、可用性与安全性。应用构建在云所提供的基础设施与基础服务之上，充分利用云服务所带来的便捷性、稳定性，降低应用架构的复杂度。云原生的微服务体系也将帮助应用架构全面升级，让应用天然具有更好的可观测性、可控制性、可容错性等特性。

### 2 微服务设计约束

相较于单体应用，微服务架构的架构转变，在提升开发、部署等环节灵活性的同时，也提升了在运维、监控环节的复杂性。在结合实践总结后，我们认为设计一个优秀的微服务系统应遵循以下设计约束：

#### 微服务个体约束

一个设计良好的微服务应用，所完成的功能在业务域划分上应是相互独立的。与单体应用强行绑定语言和技术栈相比，这样做的好处是不同业务域有不同的技术选择权，比如推荐系统采用 Python 实现效率可能比 Java 要高效得多。从组织上来说，微服务对应的团队更小，开发效率也更高。“一个微服务团队一顿能吃掉两张披萨饼”、“一个微服务应用应当能至少两周完成一次迭代”，都是对如何正确划分微服务在业务域边界的隐喻和标准。总结来说，微服务的“微”并不是为了微而微，而是按照问题域对单体应用做合理拆分。

进一步，微服务也应具备正交分解特性，在职责划分上专注于特定业务并将之做好，即 SOLID 原则中单一职责原则（SRP, Single Responsibility Principle）。如果当一个微服务修改或者发布时，不应该影响到同一系统里另一个微服务的业务交互。

### 微服务与微服务之间的横向关系

在合理划分好微服务间的边界后，主要从微服务的可发现性和可交互性处理服务间的横向关系。

微服务的可发现性是指当服务 A 发布和扩缩容的时候，依赖服务 A 的服务 B 如何在不重新发布的前提下，如何能够自动感知到服务 A 的变化？这里需要引入第三方服务注册中心来满足服务的可发现性；特别是对于大规模微服务集群，服务注册中心的推送和扩展能力尤为关键。

微服务的可交互性是指服务 A 采用什么样的方式可以调用服务 B。由于服务自治的约束，服务之间的调用需要采用与语言无关的远程调用协议，比如 REST 协议很好的满足了“与语言无关”和“标准化”两个重要因素，但在高性能场景下，基于 IDL 的二进制协议可能是更好的选择。另外，目前业界大部分微服务实践往往没有达到 HATEOAS 启发式的 REST 调用，服务与服务之间需要通过事先约定接口来完成调用。为了进一步实现服务与服务之间的解耦，微服务体系中需要有一个独立的元数据中心来存储服务的元数据信息，服务通过查询该中心来理解发起调用的细节。

伴随着服务链路的不断变长，整个微服务系统也就变得越来越脆弱，因此面向失败设计的原则在微服务体系中就显得尤为重要。对于微服务应用个体，限流、熔断、隔仓、负载均衡等增强服务韧性的机制成为了标配。为进一步提升系统吞吐能力、充分利用好机器资源，可以通协程、Rx 模型、异步调用、反压等手段来实现。

### 微服务与数据层之间的纵向约束

在微服务领域，提倡数据存储隔离（DSS, Data Storage Segregation）原则，即数据是微服务的私有资产，对于该数据的访问都必须通过当前微服务提供的 API 来访问。如若不然，则造成数据层产生耦合，违背了高内聚低耦合的原则。同时，出于性能考虑，通常采取读写分离（CQRS）手段。

同样的，由于容器调度对底层设施稳定性的不可预知影响，微服务的设计应当尽量遵循无状态设计原则，这意味着上层应用与底层基础设施的解耦，微服务可以自由在不同容器间被调度。对于有数据存取（即有状态）的微服务而言，通常使用计算与存储分离方式，将数据下沉到分布式存储，通过这个方式做到一定程度的无状态化。

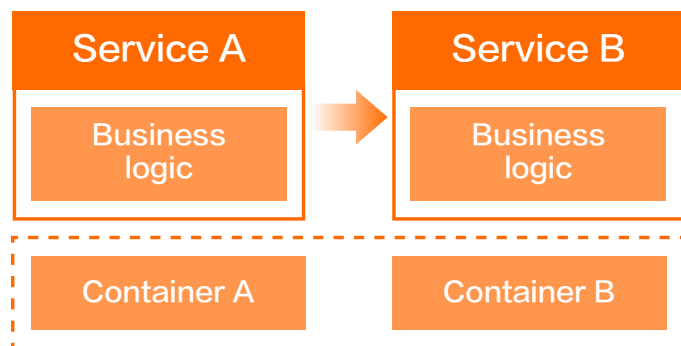
### 全局视角下的微服务分布式约束

从微服务系统设计一开始，就需要考虑以下因素：

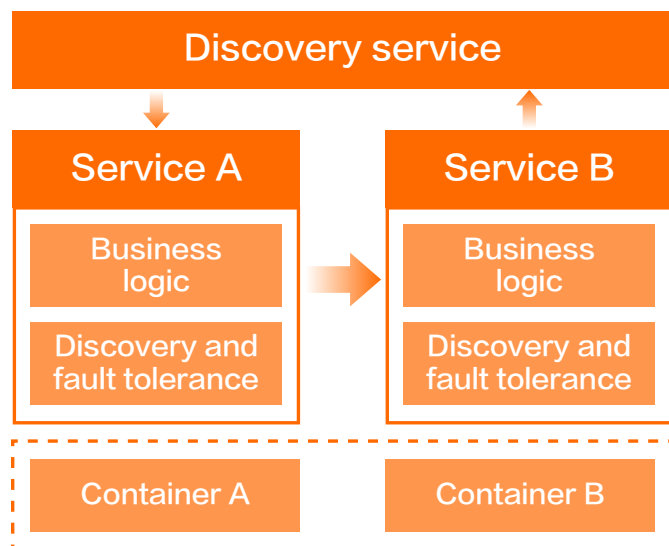
- 高效运维整个系统，从技术上要准备全自动化的 CI/CD 流水线满足对开发效率的诉求，并在这个基础上支持蓝绿、金丝雀等不同发布策略，以满足对业务发布稳定性的诉求。
- 面对复杂系统，全链路、实时和多维度的可观测能力成为标配。为了及时、有效地防范各类运维风险，需要从微服务体系多种事件源汇聚并分析相关数据，然后在中心化的监控系统中进行多维度展现。伴随着微服务拆分的持续，故障发现时效性和根因精确性始终是开发运维人员的核心诉求。

### 3 云原生微服务典型架构

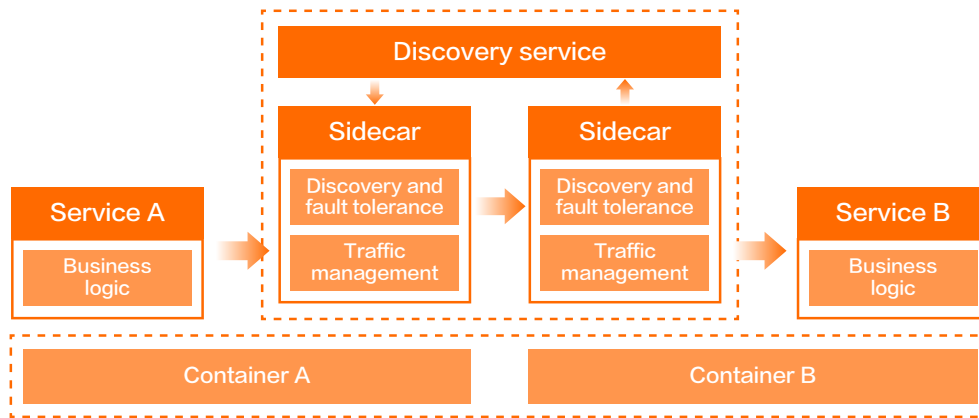
自从微服务架构理念在 2011 年提出以来，典型的架构模式按出现的先后顺序大致分为四代。



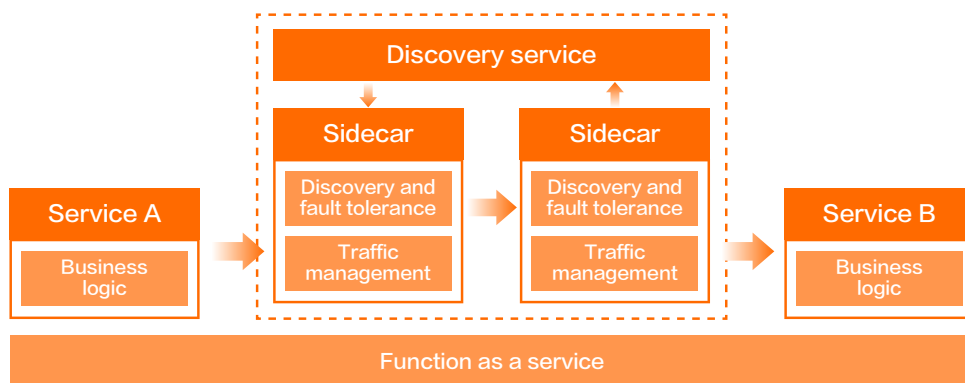
第一代微服务架构中，应用除了需要实现业务逻辑之外，还需要自行解决上下游寻址、通讯，以及容错等问题。随着微服务规模扩大，服务寻址逻辑的处理变得越来越复杂，哪怕是同一编程语言的另一个应用，上述微服务的基础能力都需要重新实现一遍。



在第二代微服务架构中，引入了旁路服务注册中心作为协调者来完成服务的自动注册和发现。服务之间的通讯以及容错机制开始模块化，形成独立服务框架。但是随着服务框架内功能日益增多，用不同语言的基础功能复用显得十分困难，这也就意味着微服务的开发者被迫被绑定在某种特定语言上，从而违背了微服务的敏捷迭代原则。



2016 年出现了第三代微服务架构 – 服务网格，原来被模块化到服务框架里的微服务基础能力，被进一步的从一个 SDK 演进成为一个独立进程 – Sidecar。这个变化使得第二代架构中多语言支持问题得以彻底解决，微服务基础能力演进和业务逻辑迭代彻底解耦。这个架构就是在云原生时代的微服务架构 – Cloud Native Microservices，边车（Sidecar）进程开始接管微服务应用之间的流量，承载第二代中服务框架的功能，包括服务发现、调用容错，到丰富的服务治理功能，例如：权重路由、灰度路由、流量重放、服务伪装等。



近两年开始，随着 AWS Lambda 的出现，部分应用开始尝试利用 Serverless 来架构微服务，这种方式被称之为第四代微服务架构。在这个架构中，微服务进一步由一个应用简化为微逻辑（Micrologic），从而对边车模式提出了更高诉求，更多可复用的分布式能力从应用中剥离，被下沉到边车中，例如：状态管理、资源绑定、链路追踪、事务管理、安全等等。同时，在开发侧开始提倡面向 localhost 编程的理念，提供标准 API 屏蔽掉底层资源、服务、基础设施的差异，进一步降低微服务开发难度。这个也就是目前业界提出的多运行时微服务架构（Multi-Runtime Microservices）。

## 4 主要微服务技术

Apache Dubbo 作为源自阿里巴巴的一款开源高性能 RPC 框架，特性包括基于透明接口的 RPC、智能负载均衡、自动服务注册和发现、可扩展性高、运行时流量路由与可视化的服务治理。经过数年发展已是国内使用最广泛的微服务框架并构建了强大的生态体系。为了巩固 Dubbo 生态的整体竞争力，2018 年阿里巴巴陆续开源了 Spring-Cloud Alibaba( 分布式应用框架 )、Nacos( 注册中心 & 配置中心 )、Sentinel( 流控防护 )、Seata( 分布式事务 )、Chaosblade( 故障注入 )，以便让用户享受阿里巴巴十年沉淀的微服务体系，获得简单易用、高性能、高可用等核心能力。Dubbo 在 v3 中发展 Service Mesh，目前 Dubbo 协议已经被 Envoy 支持，数据层选址、负载均衡和服务治理方面的工作还在继续，控制层目前在继续丰富 Istio/Pilot-discovery 中。

Spring Cloud 作为开发者的主要微服务选择之一，为开发者提供了分布式系统需要的配置管理、服务发现、断路器、智能路由、微代理、控制总线、一次性 Token、全局锁、决策竞选、分布式会话与集群状态管理等能力和开发工具。

Eclipse MicroProfile 作为 Java 微服务开发的基础编程模型，它致力于定义企业 Java 微服务规范，MicroProfile 提供指标、API 文档、运行状况检查、容错与分布式跟踪等能力，使用它创建的云原生微服务可以自由地部署在任何地方，包括 Service Mesh 架构。

Tars 是腾讯将其内部使用的微服务框架 TAF ( Total Application Framework ) 多年的实践成果总结而成的开源项目，在腾讯内部有上百个产品使用，服务内部数千名 C++、Java、Golang、Node.js 与 PHP 开发者。Tars 包含一整套开发框架与管理平台，兼顾多语言、易用性、高性能与服务治理，理念是让开发更聚焦业务逻辑，让运维更高效。

SOFAStack ( Scalable Open Financial Architecture Stack ) 是由蚂蚁金服开源的一套用于快速构建金融级分布式架构的中间件，也是在金融场景里锤炼出来的最佳实践。MOSN 是 SOFAStack 的组件，它一款采用 Go 语言开发的 Service Mesh 数据平面代理，功能和定位类似 Envoy，旨在提供分布式、模块化、可观测、智能化的代理能力。MOSN 支持 Envoy 和 Istio 的 API，可以和 Istio 集成。

Dapr ( Distributed Application Runtime，分布式应用运行时 ) 是微软新推出的，一种可移植的、Serverless 的、事件驱动的运行时代，它使开发人员可以轻松构建弹性，无状态和有状态微服务，这些服务运行在云和边缘上，并包含多种语言和开发框架。



## 3 Serverless



### 1 技术特点

随着以 Kubernetes 为代表的云原生技术成为云计算的容器界面，Kubernetes 成为云计算的新一代操作系统。面向特定领域的后端云服务（BaaS）则是这个操作系统上的服务 API，存储、数据库、中间件、大数据、AI 等领域的大量产品与技术都开始提供全托管的云形态服务，如今越来越多用户已习惯使用云服务，而不是自己搭建存储系统、部署数据库软件。

当这些 BaaS 云服务日趋完善时，Serverless 因为屏蔽了服务器的各种运维复杂度，让开发人员可以将更多精力用于业务逻辑设计与实现，而逐渐成为云原生主流技术之一。Serverless 计算包含以下特征：

- **全托管的计算服务**，客户只需要编写代码构建应用，无需关注同质化的、负担繁重的基于服务器等基础设施的开发、运维、安全、高可用等工作；
- **通用性**，结合云 BaaS API 的能力，能够支撑云上所有重要类型的应用；
- **自动的弹性伸缩**，让用户无需为资源使用提前进行容量规划；
- **按量计费**，让企业使用成本得到有效降低，无需为闲置资源付费。

函数计算（Function as a Service）是 Serverless 中最具代表性的产品形态。通过把应用逻辑拆分多个函数，每个函数都通过事件驱动的方式触发执行，例如当对象存储（OSS）中产生的上传 / 删除对象等事件，能够自动、可靠地触发 FaaS 函数处理且每个环节都是弹性和高可用的，客户能够快速实现大规模数据的实时并行处理。同样的，通过消息中间件和函数计算的集成，客户可以快速实现大规模消息的实时处理。

目前函数计算这种 Serverless 形态在普及方面仍存在一定困难，例如：

- 函数编程以事件驱动方式执行，这在应用架构、开发习惯方面，以及研发交付流程上都会有比较大的改变；
- 函数编程的生态仍不够成熟，应用开发者和企业内部的研发流程需要重新适配；
- 细颗粒度的函数运行也引发了新技术挑战，比如冷启动会导致应用响应延迟，按需建立数据库连接成本高等。

针对这些情况，在 Serverless 计算中又诞生出更多其他形式的服务形态，典型的就和容器技术进行融合创新，通过良好的可移植性，容器化的应用能够无差别地运行在开发机、自建机房以及公有云环境中；基于容器工具链能够加快解决 Serverless 的交付。云厂商如阿里云提供了弹性容器实例（ECI）以及更上层的 Serverless 应用引擎（SAE），Google 提供了 CloudRun 服务，这都帮助用户专注于容器化应用构建，而无需关心基础设施的管理成本。此外 Google 也开源了基于 Kubernetes 的 Serverless 应用框架 Knative。

相对函数计算的编程模式，这类 Serverless 应用服务支持容器镜像作为载体，无需修改即可部署在 Serverless 环境中，可以享受到 Serverless 带来的全托管免运维、自动弹性伸缩、按量计费等优势。下面是传统的弹性计算服务、基于容器的 Serverless 应用服务和函数计算的对比：



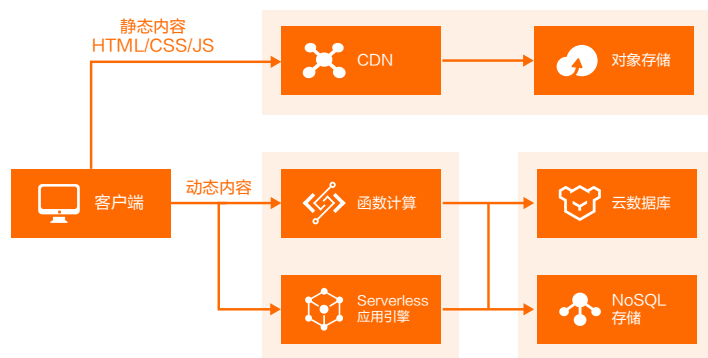
服务分类	弹性计算	Serverless 应用	FaaS
代表产品	弹性计算 ECS	阿里云 SAE Google Knative	函数计算
虚拟化	硬件虚拟化	安全容器	安全容器或应用运行时
交付模式	虚拟机镜像	容器镜像	函数
应用兼容性	高	中	低
扩容单位	虚拟机	容器实例	函数实例
弹性效率	分钟级	秒级	毫秒级
计费模式	实例运行时长	实例运行时长或 请求处理时长	请求处理时长

## 2 常见场景

近两年来 Serverless 近年来呈加速发展趋势，用户使用 Serverless 架构在可靠性、成本和研发运维效率等方面获得显著收益。

### 小程序 /Web/Mobile/API 后端服务

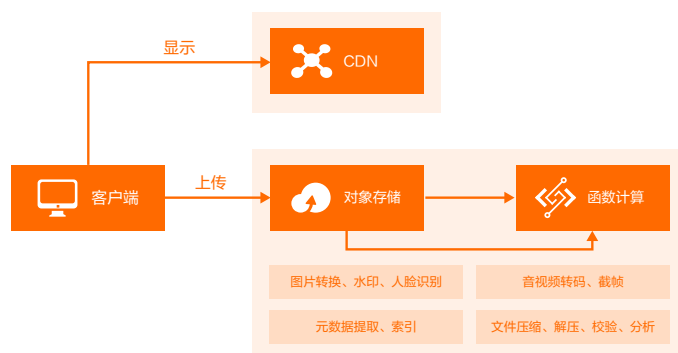
在小程序、Web/Mobile 应用、API 服务等场景中，业务逻辑复杂多变，迭代上线速度要求高，而且这类在线应用，资源利用率通常小于 30%，尤其是小程序等长尾应用，资源利用率更是低于 10%。Serverless 免运维，按需付费的特点非常适合构建小程序 /Web/Mobile/API 后端系统，通过预留计算资源 + 实时自动伸缩，开发者能够快速构建延时稳定、能承载高频访问的在线应用。在阿里内部，使用 Serverless 构建后端服务是落地最多的场景，包括前端全栈领域的 Serverless For Frontends，机器学习算法服务，小程序平台实现等等。



## 大规模批处理任务

在构建典型任务批处理系统时，例如大规模音视频文件转码服务，需要包含计算资源管理、任务优先级调度、任务编排、任务可靠执行、任务数据可视化等一系列功能。如果从机器或者容器层开始构建，用户通常使用消息队列进行任务信息的持久化和计算资源分配，使用 Kubernetes 等容器编排系统实现资源的伸缩和容错，自行搭建或集成监控报警系统。而通过 Serverless 计算平台，用户只需要专注于任务处理逻辑的处理，而且 Serverless 计算的极致弹性可以很好地满足突发任务下对算力的需求。

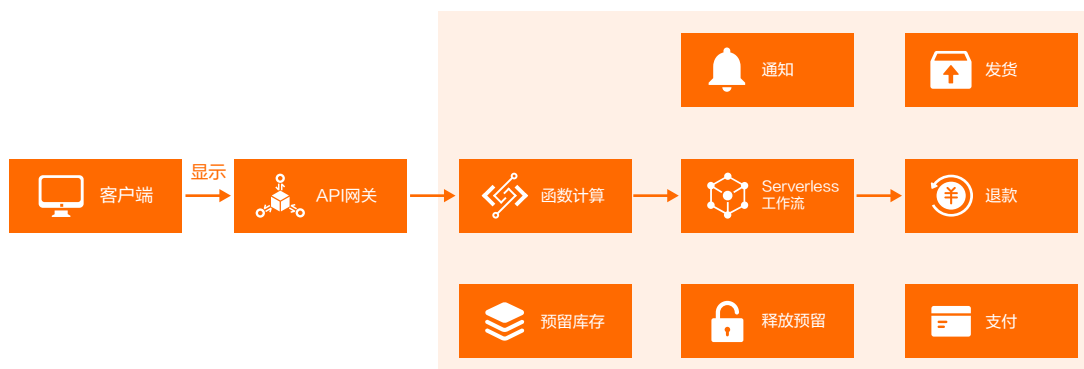
通过将对象存储和 Serverless 计算平台集成的方式，能实时响应对象创建、删除等操作，实现以对象存储为中心的大规模数据处理。用户既可以通过增量处理对象存储上的新增数据，也可以创建大量函数实例来并行处理存量数据。



## 基于事件驱动架构的在线应用和离线数据处理

典型 Serverless 计算服务通过事件驱动的方式，可以广泛地与云端各种类型服务集成，用户无需管理服务器等基础设施和编写集成多个服务的“胶水”代码，就能够轻松构建松耦合、基于分布式事件驱动架构的应用。

通过和事件总线的集成，无论是一方 BaaS 云服务，还是三方的 SaaS 服务，或者是用户自建的系统，所有事件都可以快速便捷地被函数计算处理。例如通过和 API 网关集成，外部请求可以转化为事件，从而触发后端函数处理。通过和消息中间件的事件集成，用户能快速实现对海量消息的处理。



## 开发运维自动化

通过定时触发器，用户用函数的方式就能够快速实现定时任务，而无须管理执行任务的底层服务器。通过将定时触发器和监控系统的时间触发器集成，用户可以及时接收机器重启、宕机、扩容等 IaaS 层服务的运维事件，并自动触发函数执行处理。

## 3 技术关注点

### 计算资源弹性调度

为了实现精准、实时的实例伸缩和放置，必须把应用负载的特征作为资源调度依据，使用“白盒”调度策略，由 Serverless 平台负责管理应用所需的计算资源。平台要能够识别应用特征，在负载快速上升时，及时扩容计算资源，保证应用性能稳定；在负载下降时，及时回收计算资源，加快资源在不同租户函数间的流转，提高数据中心利用率。因此更实时、更主动、更智能的弹性伸缩能力是函数计算服务获得良好用户体验的关键。通过计算资源的弹性调度，帮助用户完成指标收集、在线决策、离线分析、决策优化的闭环。

在创建新实例时，系统需要判断如何将应用实例放置在下层计算节点上。放置算法应当满足多方面的目标：

- **容错**：当有多个实例时，将其分布在不同的计算节点和可用区上，提高应用的可用性。
- **资源利用率**：在不损失性能的前提下，将计算密集型、I/O 密集型等应用调度到相同计算节点上，尽可能充分利用节点的计算、存储和网络资源。动态迁移不同节点上的碎片化实例，进行“碎片整理”，提高资源利用率。
- **性能**：例如复用启动过相同应用实例或函数的节点、利用缓存数据加速应用的启动时间。
- **数据驱动**：除了在线调度，系统还将天、周或者更大时间范围的数据用于离线分析。离线分析的目的在于利用全量数据验证在线调度算法的效果，为参数调优提供依据，通过数据驱动的方式加快资源流转速度，提高集群整体资源利用率。

### 负载均衡和流控

资源调度服务是 Serverless 系统的关键链路。为了支撑每秒近百万次的资源调度请求，系统需要对资源调度服务的负载进行分片，横向扩展到多台机器上，避免单点瓶颈。分片管理器通过监控整个集群的分片和服务器负载情况，执行分片的迁移、分裂、合并操作，从而实现集群处理能力的横向扩展和负载均衡。

在多租户环境下，流量隔离控制是保证服务质量的关键。由于用户是按实际使用的资源付费，因此计算资源要通过被不同用户的不同应用共享来降低系统成本。这就需要系统具备出色的隔离能力，避免应用相互干扰。

### 安全性

Serverless 计算平台的定位是通用计算服务，要能执行任意用户代码，因此安全是不可逾越的底线。系统应从权限管理、网络安全、数据安全、运行时安全等各个维度全面保障应用的安全性。轻量安全容器等新的虚拟化技术实现了更小的资源隔离粒度、更快的启动速度、更小的系统开销，使数据中心的资源使用变得更加细粒度和动态化，从而更充分地利用碎片化资源。

## 4 开放应用模型（OAM）



### 1 开放应用模型的背景

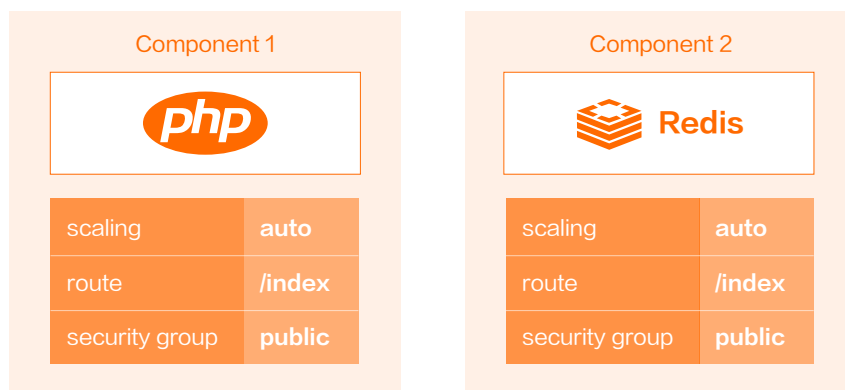
随着 Kubernetes 技术体系逐渐成熟，整个云原生生态正在积极思考与探索如何达成云原生技术理念——“以应用为中心”的终极愿景。2019 年末，阿里云联合微软共同发布了 Open Application Model (OAM) 开源项目，其主要目标是解决从 Kubernetes 项目到“以应用为中心”的平台之间关键环节——标准化应用定义。

### 2 开放应用模型的定义

容器技术以“彻底改变了软件打包与分发方式”迅速得到大量企业的广泛使用。不过软件打包与分发方式的革新，并没有能够让软件本身的定义与描述发生本质变化；基于 Kubernetes 的应用管理体验，还没有让业务研发与运维的工作变得足够简单。最典型的例子，Kubernetes 至今都没有“应用”这个概念，它提供的是更细粒度的“工作负载”原语，比如 Deployment 或者 DaemonSet。在实际环境中，一个应用往往由一系列独立组件组成，比如一个“PHP 应用容器”和一个“数据库实例”组成电商网站；一个“参数服务节点”和一个“工作节点”组成机器学习训练任务；一个由“Deployment + StatefulSet + HPA + Service + Ingress”组成微服务应用。

OAM 的第一个设计目标就是补充“应用”这一概念，建立对应用和它所需的运维能力定义与描述的标准规范。换言之，OAM 既是标准“应用定义”同时也是帮助封装、组织和管理 Kubernetes 中各种“运维能力”。在具体设计上，OAM 的描述模型是基于 Kubernetes API 的资源模型（Kubernetes Resource Model）来构建的，它强调一个现代应用是多个资源的集合，而非一个简单工作负载。例如在 PHP 电商网站的 OAM 语境中，一个 PHP 容器和它所依赖的数据库以及它所需要使用的各种云服务，都是一个“电商网站”应用的组成部分。同时，OAM 把这个应用所需的“运维策略”也认为是应用的一部分，比如这个 PHP 容器所需的 HPA（水平自动扩展策略）：

Application



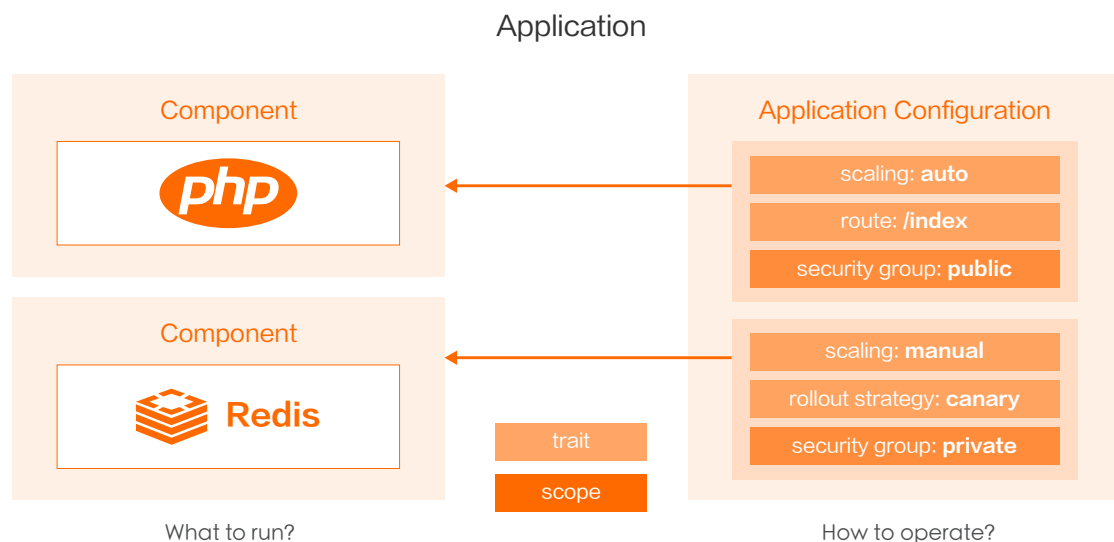
OAM 项目的第二个设计目标就是提供更高层级的应用层抽象和以应关注点分离的定义模型。Kubernetes 作为一个面向基础设施工程师的系统级项目，主要负责提供松耦合的基础设施语义，这就使得用户编写 Kubernetes YAML 文件时，往往会感觉这些文件里的关注点非常底层。实际上，对于业务研发人员和运维人员而言，他们并不想配置这些如此底层的资源信息，而希望有更高维度的抽象。这就要求一个真正面向最终用户侧的应用定义，一个能够为业务研发和应用运维人员提供各自所需的应用定义原语。

## 开放应用模型的核心概念

OAM 主要定义了三个具体的概念和对应的标准，它们包括应用组件依赖、应用运维特征和应用配置：

- **组件依赖**：OAM 定义和规范了组成应用的组件（Component）。例如，一个前端 Web Server 容器、数据库服务、后端服务容器等；
- **应用运维特征**：OAM 定义和规范了应用所需的运维特征（Trait）的集合。例如，弹性伸缩和 Ingress 等运维能力；
- **应用配置**：OAM 定义和规范了应用实例化所需的配置机制，从而能够将上述这些描述转化为具体应用实例。具体来说，运维人员可以定义和使用应用配置（Application Configuration）来组合上述的组件和相应的特征，以构建可部署的应用交付实例。

例如，一个由 PHP 容器和 Redis 实例组成的应用，在 OAM 的框架和规范下，就可以用如下的示意图来表达出来：



而在上述模块化的应用定义基础上，OAM 模型还强调整个模型的关注点分离特性。即业务研发人员负责定义与维护组件（Component）来描述服务单元，而运维人员定义运维特征（Trait）并将其附加到组件上，构成 OAM 可交付物 - Application Configuration。这种设计使 OAM 在能够无限接入 Kubernetes 各种能力同时，为业务研发与运维人员提供最佳使用体验和最低心智负担。


## 在 Kubernetes 上使用开放应用模型

在上述应用定义模型规范基础上，OAM 提供了标准的 OAM Kubernetes 插件（即：OAM Kubernetes 核心依赖库项目），从而能够“一键式”的为任何 Kubernetes 集群加上基于 OAM 模型的应用定义与应用管理能力。

具体来说，这个插件的功能包括：

### 功能一：无缝对接现有 K8s API 资源

OAM Kubernetes 核心依赖库支持将任何现有 CRD 被声明为 Workload 或者 Trait，而无需做任何改动。这也意味着任何 Kubernetes 原生 API 资源也可以被声明为 Workload 或者 Trait。通过这种设计，现有 Kubernetes 集群里所有能力进行 OAM 化变得非常容易。




## Component

**组件：**  
完全从研发视角定义的应用模块

**工作负载：** 应用模块的具体描述

**举例：** 通过OAM定义和部署OpenFaaS Function

Component A



```

apiVersion: core.oam.dev/v1alpha2
kind: Component
metadata:
  name: web-function
  annotations:
    description: OpenFaaS workload
spec:
  workload:
    apiVersion: openfaas.com/v1
    kind: Function
    spec:
      name: nodeinfo
      handler: node main.js
      image: function/nodeinfo
      environment:
        http_proxy: http://proxy1.corp.com:3128
        no_proxy: http://gateway/
          
```

### 功能二：Workload 与 Trait 标准化交互机制

OAM Kubernetes 插件保证了 OAM 可以模块式接入、部署和管理任何 Kubernetes 工作负载和运维能力。而这些工作负载和运维能力之间的交互需要标准化、统一化，Workload 与 Trait 标准化交互机制应运而生。比如 Deployment 和 HPA（自动水平扩展控制器）的协作关系中，Deployment 在 OAM 模型中就属于 Workload，而 HPA 则属于 Trait。在 OAM 当中，Application Configuration 里引用的 Workload 和 Trait 也必须通过协作方式来操作具体 Kubernetes 资源。

在 OAM Kubernetes 核心依赖库中，过 DuckTyping（鸭子类型）机制，在 Trait 对象上自动记录与之绑定的 Workload 关系，从而实现了工作负载（Workload）和运维能力（Trait）之间的双向记录关系：

- 给定任何一工作负载（Workload），系统可以直接获取到同它绑定的所有运维能力（Trait）；

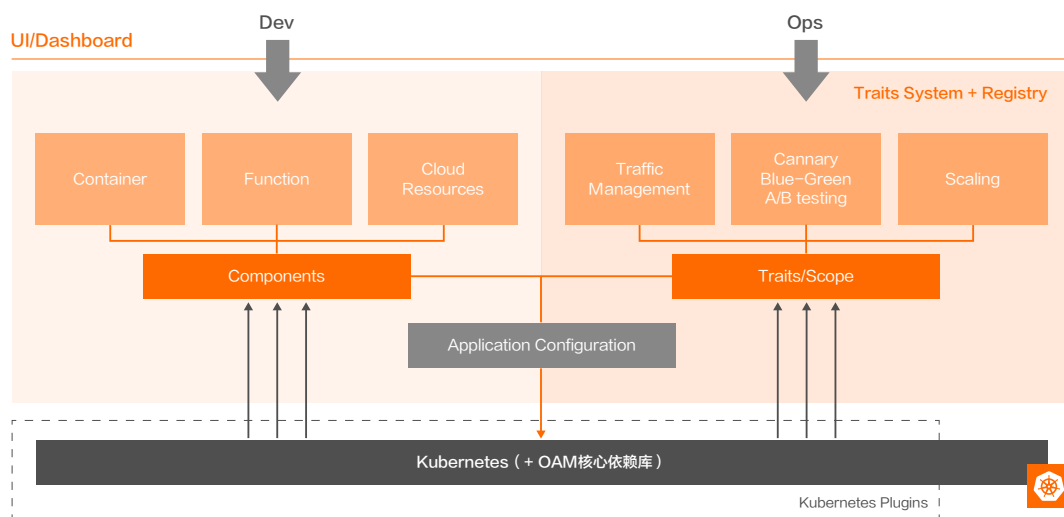
- 给定任何一个运维能力（Trait），系统可以直接获取到它所作用于的所有工作负载（Workload）。这种双向记录关系对于在一个大规模的生产环境中保证运维能力的可管理性、可发现性和应用稳定性是至关重要的。

除此之外，OAM Kubernetes 插件还提供几个非常重要的基础功能，包括：

- **Component 版本管理**：对于任何一次 Component 变更，OAM 平台都将记录下来其变更历史，从而允许运维通过 Trait 来进行回滚、蓝绿发布等运维操作。
- **Component 间依赖关系与参数传递**：该功能将解决部署亟需的组件间依赖问题，包括 Component 之间的依赖和传输传递，以及 Trait 与 Component 之间的依赖和参数传递。
- **Component 运维策略**：该功能将允许研发在 Component 中声明对运维能力的诉求，指导运维人员或者系统给这个 Component 绑定和配置合理的运维能力。

## 开放应用模型的未来

相比于传统 PaaS 封闭、不能同“以 Operator 为基础的云原生生态”衔接的现状，基于 OAM 和 Kubernetes 构建的现代云原生应用管理平台的本质是一个“以应用为中心”的 Kubernetes，保证应用平台能够无缝接入整个云原生生态。同时，OAM 进一步屏蔽掉容器基础设施的复杂性和差异性，为平台使用者带来低心智负担的、标准化的、一致化的应用管理与交付体验，让一个应用描述可以完全不加修改的在云、边、端等任何环境下直接交付运行起来。



一个基于 OAM 构建的 Kubernetes 应用管理平台

此外，OAM 还定义了一组核心工作负载 / 运维特征 / 应用范畴，作为应用程序交付平台的基石。当模块化的 Workload 和 Trait 越来越多，就会形成组件市场。而 OAM 就像是这个组件市场的管理者，通过处理组件之间的关系，把许多组件集成为一个产品交付给用户。OAM 加持下的 Kubernetes 应用拼图，可以像乐高积木一样灵活组装底层能力、运维特征以及开发组件。

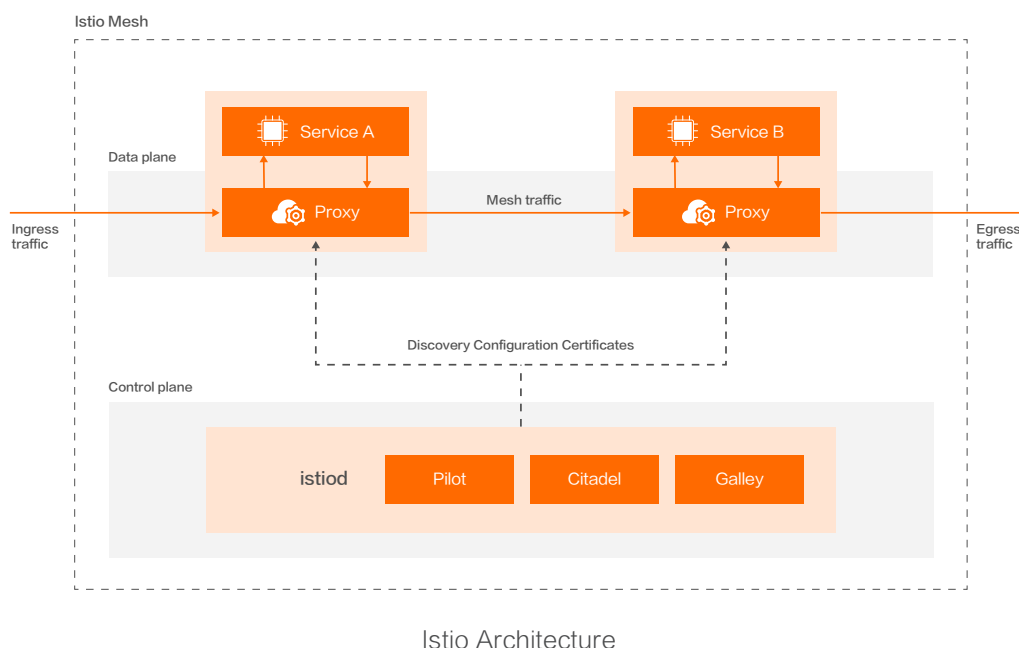
OAM 社区还同混合云管理项目 Crossplane 建立了深度合作，从而保证符合 OAM 规范的待运行程序、运维能力和它所依赖的云服务，可以组成一个整体，在混合云环境中无缝漂移。这种无关平台的应用定义范式，使得应用研发人员只需通过 OAM 规范来描述应用程序，应用程序就可以在任何 Kubernetes 群集或者 Serverless 应用平台甚至边缘环境上运行而无需对应用描述做任何修改。OAM 社区与整个云原生生态一起，正在将标准应用定义和标准化的云服务管理能力统一起来，使“云端应用交付”的故事真正成为现实。

## 5 Service Mesh 技术



### 1 技术特点

Service Mesh 是分布式应用在微服务软件架构之上发展起来的新技术，旨在将那些微服务间的连接、安全、流量控制和可观测等通用功能下沉为平台基础设施，实现应用与平台基础设施的解耦。这个解耦意味着开发者无需关注微服务相关治理问题而聚焦于业务逻辑本身，提升应用开发效率并加速业务探索和创新。换句话说，因为大量非功能性从业务进程剥离到另外进程中，Service Mesh 以无侵入的方式实现了应用轻量化，下图展示了 Service Mesh 的典型架构：



在这张架构图中，Service A 调用 Service B 的所有请求，都被其下的 Proxy（在 Envoy 中是 Sidecar）截获，代理 Service A 完成到 Service B 的服务发现、熔断、限流等策略，而这些策略的总控是在 Control Plane 上配置。



从架构上, Istio 可以运行在虚拟机或容器中, Istio 的主要组件包括 Pilot( 服务发现、流量管理 )、Mixer( 访问控制、可观测性 )、Citadel ( 终端用户认证、流量加密 ); 整个服务网格关注连接和流量控制、可观测性、安全和可运维性。虽然相比较没有服务网格的场景多了 4 个 IPC 通讯的成本, 但整体调用的延迟随着软硬件能力的提升而并不会带来显著的影响, 特别是对于百毫秒级别的业务调用而言可以控制在 2% 以内。从另一方面, 服务化的应用并没有做任何改造, 就获得了强大的流量控制能力、服务治理能力、可观测能力、4 个 9 以上高可用、容灾和安全等能力, 加上业务的横向扩展能力, 整体收益仍然是远大于额外 IPC 通讯支出。

服务网格的技术发展上数据平面与控制平面间的协议标准化是必然趋势。大体上, Service Mesh 的技术发展围绕着“事实标准”去展开——共建各云厂商共同采纳的开源软件。从接口规范的角度: Istio 采纳了 Envoy 所实现的 xDS 协议, 将该协议当作是数据平面和控制平面间的标准协议; Microsoft 提出了 Service Mesh Interface ( SMI ), 致力于让数据平面和控制平面的标准化做更高层次的抽象, 以期 Istio、Linkerd 等 Service Mesh 解决方案在服务观测、流量控制等方面实现最大程度的开源能力复用。UDPA ( Universal Data Plane API ) 是基于 xDS 协议而发展起来, 以便根据不同云厂商的特定需求便捷地进行扩展并由 xDS 去承载。

此外数据平面插件的扩展性和安全性也得到了社区的广泛重视。从数据平面角度, Envoy 得到了包括 Google、IBM、Cisco、Microsoft、阿里云等大厂的参与共建以及主流云厂商的采纳而成为了事实标准。在 Envoy 的软件设计为插件机制提供了良好扩展性的基础之上, 目前正在探索将 Wasm 技术运用于对各种插件进行隔离, 避免因为某一插件的软件缺陷而导致整个数据平面不可用。Wasm 技术的优势除了提供沙箱功能外, 还能很好地支持多语言, 最大程度地让掌握不同编程语言的开发者可以使用自己所熟悉的技能去扩展 Envoy 的能力。在安全方面, Service Mesh 和零信任架构天然有很好的结合, 包括 POD Identity、基于 mTLS 的链路层加密、在 RPC 上实施 RBAC 的 ACL、基于 Identity 的微隔离环境 ( 动态选取一组节点组成安全域 )。

## 2 行业应用情况

根据 Gartner 研究报告, Istio 有望成为 Service Mesh 的事实标准, 而 Service Mesh 本身也将成为容器服务技术的标配技术组件。即便如此, Service Mesh 目前在市场仍处于早期采用 (Early adoption) 阶段。

除 Istio 以外, Google 与 AWS 分别推出了各自的云服务 Traffic Director、App Mesh。这两个 Service Mesh 产品与 Istio 虽有所不同, 但与 Istio 同样地采纳了 Envoy 作为数据平面。此外, 阿里云、腾讯云、华为云也都推出了 Service Mesh 产品, 同样采用 Envoy 技术作为数据面并在此基础上提供了应用发布、流量管控、APM 等能力。

## 3 主要技术

2017 年发起的服务网格 Istio 开源项目, 清晰定义了数据平面 ( 由开源软件 Envoy 承载 ) 和管理平面 ( Istio 自身的核心能力 )。Istio 为微服务架构提供了流量管理机制, 同时亦为其它增值功能 ( 包括安全性、监控、路由、连接管理与策略等 ) 创造了基础。Istio 利用久经考验的 Lyft Envoy 代理进行构建, 可在无需对应用程序代码作出任何

发动的前提下实现可视性与控制能力。2019 年 Istio 所发布的 1.12 版已达到小规模集群上线生产环境水平，但其性能仍受业界诟病。开源社区正试图通过架构层面演进改善这一问题。由于 Istio 是建构于 Kubernetes 技术之上，所以它天然地可运行于提供 Kubernetes 容器服务的云厂商环境中，同时 Istio 成为了大部分云厂商默认使用的服务网格方案。

除了 Istio 外，也有 Linkerd、Consul 这样相对小众的 Service Mesh 解决方案。Linkerd 在数据平面采用了 Rust 编程语言实现了 linkerd-proxy，控制平面与 Istio 一样采用 Go 语言编写。最新的性能测试数据显示，Linkerd 在时延、资源消耗方面比 Istio 更具优势。Consul 在控制面上直接使用 Consul Server，在数据面上可以选择性地使用 Envoy。与 Istio 不同的是，Linkerd 和 Consul 在功能上不如 Istio 完整。

Conduit 作为 Kubernetes 的超轻量级 Service Mesh，其目标是成为最快、最轻、最简单且最安全的 Service Mesh。它使用 Rust 构建了快速、安全的数据平面，用 Go 开发了简单强大的控制平面，总体设计围绕着性能、安全性和可用性进行。它能透明地管理服务之间的通信，提供可测性、可靠性、安全性和弹性的支持。虽然与 Linkerd 相仿，数据平面是在应用代码之外运行的轻量级代理，控制平面是一个高可用的控制器，然而与 Linkerd 不同的是，Conduit 的设计更加倾向于 Kubernetes 中的低资源部署。

## 6 DevOps



### 1 概述

DevOps 就是为了提高软件研发效率，快速应对变化，持续交付价值的的一系列理念和实践，其基本思想就是持续部署（CD），让软件的构建、测试、发布能够更加快捷可靠，以尽量缩短系统变更从提交到最后安全部署到生产系统的时间。

要实现持续部署（CD），就必须对业务进行端到端分析，把所有相关部门的操作统一考虑进行优化，利用所可用的技术和方法，用一种理念来整合资源。DevOps 理念从提出到现在，已经深刻影响了软件开发过程。DevOps 提倡打破开发、测试和运维之间的壁垒，利用技术手段实现各个软件开发环节的自动化甚至智能化，被证实对提高软件生产质量、安全，缩短软件发布周期等都有非常明显的促进作用，也推动了 IT 技术的发展。

### 2 DevOps 原则

要实施 DevOps，需要遵循一些基本原则，这些原则被简写为 CAMS，是如下四个英文单词的缩写：

- 文化（Culture）
- 自动化（Automation）
- 度量（Measurement）
- 共享（Sharing）

下面我们分别来谈一谈这几个方面。

## 文化

谈到 DevOps，一般大家关注的都是技术和工具，但实际上要解决的核心问题是和业务、和人相关的问题。提高效率，加强协作，就需要不同的团队之间更好的沟通。如果每个人能够更好的相互理解对方的目标和关切的对象，那么协作的质量就可以明显的提高。

DevOps 实施中面对的首要矛盾在于不同团队的关注点完全不一样。运维人员希望系统运行可靠，所以系统稳定性和安全性是第一位。而开发人员则想着如何尽快让新功能上线，实现创新和突破，为客户提供更大价值。不同的业务视角，必然导致误会和摩擦，导致双方都觉得对方在阻挠自己完成工作。要实施 DevOps，就首先要让开发和运维人员认识到他们的目标是一致的，只是工作岗位不同，需要共担责任。这就是 DevOps 需要首先在文化层面解决的问题。只有解决了认知问题，才能打破不同团队之间的鸿沟，实现流程自动化，把大家的工作融合成一体。

## 自动化

DevOps 的持续集成的目标就是小步快跑，快速迭代，频繁发布。小系统跑起来很容易，但一个大型系统往往牵涉几十人到几百人的合作，要让这个协作过程流畅运行不是一件容易的事情。要把这个理念落实，就需要规范化和流程化，让可以自动化的环节实现自动化。

实施 DevOps，首先就要分析已有的软件开发流程，尽量利用各种工具和平台，实现开发和发布过程的自动化。经过多年发展，业界已经有一套比较成熟的工具链可以参考和使用，不过具体落地还需因地制宜。

在自动化过程中，需要各种技术改造才能达到预期效果。例如如果容器镜像是为特定环境构建的，那么就无法实现镜像的复用，不同环境的部署需要重新构建，浪费时间。这时就需要把环境特定的配置从镜像中剥离出来，用一个配置管理系统来管理配置。

## 度量

通过数据可以对每个活动和流程进行度量和分析，找到工作中存在的瓶颈和漏洞以及对于危急情况的及时报警等。通过分析，可以对团队工作和系统进行调整，让效率改进形成闭环。

度量首先要解决数据准确性、完整性和及时性等问题，其次要建立正确的分析指标。DevOps 过程考核的标准应该鼓励团队更加注重工具的建设，自动化的加速和各个环节优化，这样才能最大可能发挥度量的作用。

## 共享

要实现真正的协作，还需要团队在知识层面达成一致。通过共享知识，让团队共同进步：

- **可见度 visibility**：让每个人可以了解团队其它人的工作。这样可以知道是否某一项工作会影响另一部分。通过相互反馈，让问题尽早暴露。

- **透明性 transparency**：让每个人都明白工作的共同目标，知道为什么要干什么。缺乏透明性就会导致工作安排失调。
- **知识的传递 transfer of knowledge**：知识的传递是为了解决两个问题，一个是为了避免某个人成为单点，从而导致一个人的休假或离职，就导致工作不能完成。另一个是提高团队的集体能力，团队的集体能力要高于个人的能力。

要实现知识共享有很多方法。在敏捷开发中，通过日站会来共享进度。在开发中，通过代码、文档和注释来共享知识。ChatOps 则是让在一个群里的人都看到正在进行的操作和操作的结果。其它的所有会议、讨论和非正式的交流等当然也是为了知识共享。从广义上讲，团队协作就是知识不断积累和分享的过程。落实 DevOps 要努力建设一个良好的文化氛围并通过工具支持让所有的共享更加方便高效。

文化、自动化、度量和共享四个方面相辅相成，独立而又相互联系，所以要落实 DevOps 时，要统一考虑。通过 CAMS 也认识到，CI/CD 仅仅是实现 DevOps 中很小的一部分。DevOps 不仅仅是一组工具，更重要是代表了一种文化，一种心智。

### 3 理性的期待

对于 DevOps 要有合乎理性的期待。DevOps 提供了一套工具，工具能够起多大的作用，其最重要的影响有两个，一是使用工具的人，另外就是对于需要解决的问题本身复杂性的掌握。虽然 DevOps 已经被广泛接受和认可，但其在实际应用中的成熟度还待进一步提高。在一份关于 DevOps 的调查报告中，把 DevOps 进化分为三个级别，80% 被调查团队处于中等级级，而处于高级阶段和低级阶段的都在 10% 左右。报告分析，这是因为通过实现自动化以达到中等级级是相对比较容易，而达到高等级则需要文化和共享方面的努力，这相对来说更加难以掌握和实施。这也符合组织文化变革中的 J 型曲线。在经过了初期的效率增长之后，自动化的深化进入了瓶颈期。更进一步的效率提升需要对组织、架构进行更加深入的调整。这段期间可能还会有效率下降、故障率攀升的问题，从这一方面讲，实际 DevOps 落地和深化还有很长路要走。

#### 转型的J型曲线



前面比较抽象的讲了一下 DevOps 的理念和原则，下面我们从更加技术的层面来分析 DevOps。

#### 4 IaC 和 GitOps

##### IaC 和声明式运维

前面说过，DevOps 所面对的矛盾就是开发和运维团队之间的矛盾。因为两个团队的关注点完全不同，或者说是冲突的。在这种背景下，IaC 提出系统建设的核心理念，兼顾高效和安全，让运维系统的建设更加有序。

运维平台一般都经历过如下几个发展阶段：手工、脚本、工具、平台、智能化运维等。现有运维平台虽然很多实现方式，但总体来说分为两类：

- 指令式
- 声明式

两者的特点如下：



采用声明式接口的方式相比较与指令式多一个执行引擎，把用户的目标转化为可以执行的计划。

最开始的运维系统一般都是指令式的，通过编写脚本来完成于运维动作，包括部署、升级、改配置、缩扩容等。脚本的优点就是简单、高效、直接等，相对于更早之前的手工运维，这是一种极大的效率提高。在分布式系统和云计算起步阶段，采用这种模式进行运维是完全合理的。基于这个方法，各个部门都会相应建立一些系统和工具来加速工具的开发和使用。

不过随着系统复杂性逐步提高，指令式的运维方式的弊端也逐渐显现出来。简单高效的优点同时也变成了最大缺点。因为方式简单，所以无法实现复杂的控制逻辑；因为高效，如果有 bug，那么在进行破坏时也同样高效。往往一个小失误就会导致大面积服务瘫痪：一个变更脚本中的 bug，可能会导致严重事故。

在复杂的运维场景下，指令式的运维方式具有变更操作副作用：不透明、指令性接口一般不具有幂等性、难以实现复杂的变更控制、知识难以积累和分享、变更缺乏并发性等缺点。针对这种情况，人们提出了声明式的编程理念。这是一个很简单的概念，用户仅仅通过一种方式描述其要到达的目的，而并不具体说明如何达到目标。声明式接口实际上代表了一种思维模式：把系统的核心功能进行抽象和封装，让用户在一个更高的层次上进行操作。

声明式接口是一种和云计算时代相契合的思维范式。前面列出的指令式的缺点都可以由声明式接口来弥补。

- **幂等性**：运维终态被反复提交也不会具有任何副作用。
- **声明式最明显的优点是变更审核简单明了**。配置中心会保存历史上的所有版本的配置文件。通过对比新的配置和上一个版本，可以非常明确看到配置的具体变更。一般来说，每次变更的范围不是很大，所以审核比较方便。通过审核可以拦截很多人为失误。通过把所有的变更形式都统一为对配置文件的变更，无论是机器的变更、网络的变更还是软件版本和应用配置的变更等。在人工审核之外，还可以通过程序来检测用户配置是否合乎要求，从而捕捉用户忽略掉的一些系统性的限制，防患于未然。
- **复杂性抽象**：系统复杂性越来越高，系统间相互依赖和交互越来越广泛，以及由于操作者无法掌握所有可能的假设条件、依赖关系等而带来的运维复杂性。解决这个问题的唯一思路，就是要把更多逻辑和知识沉淀到运维平台中，从而有效降低用户使用难度和操作风险。

## GitOps

GitOps 作为 IaC 运维理念的一种具体落地方式，就是使用 Git 来存储关于应用系统的最终状态的声明式描述。GitOps 的核心是一个 GitOps 引擎，它负责监控 Git 中的状态，每当它发现状态有改变，它就负责把目标应用系统中的状态以安全可靠的方式迁移到目标状态，实现部署、升级、配置修改、回滚等操作。

Git 中存储有对于应用系统的完整描述以及所有修改历史。方便重建的同时，也便于对系统的更新历史进行查看，符合 DevOps 所提倡的透明化原则。同时，GitOps 也具有声明式运维的所有优点。

和 GitOps 配套的一个基本假设是不可变基础设施，所以 GitOps 和 Kubernetes 运维可以非常好的配合。



GitOps 引擎需要比较当前态和 Git 中的终态间的差别，然后以一种可靠的方式把系统从任何当前状态转移到终态，所以 GitOps 系统的设计还是比较复杂的。对于用户的易用，实则是因为围绕 GitOps 有一套完整的工具和平台的支持。

## 5 云原生时代的 DevOps

相对于传统 IT 基础设施，云具有更加灵活的调度策略，接近无限的资源、丰富的服务供用户选择、使用，这些都极大方便了软件的建设。而云原生开源生态的建设，基本统一了软件部署和运维的基本模式。更重要的是，云原生技术的快速演进，技术复杂性不断下沉到云，赋能开发者个体能力，不断提升了应用开发效率。



首先是容器技术和 Kubernetes 服务编排技术的结合，解决了应用部署自动化、标准化、配置化问题。CNCF 打破了云上平台的壁垒，使建设跨平台的应用成为可能，成为事实上的云上应用开发平台的标准，极大简化了多云部署。

一个完整开发流程涉及到很多步骤，而环节越多，一次循环花费的时间越长，效率就越低。微服务通过把巨石应用拆解为若干单功能的服务，减少了服务间的耦合性，让开发和部署更加便捷，可以有效降低开发周期，提高部署灵活性。Service Mesh 让中间件的升级和应用系统的升级完全解耦，在运维和管控方面的灵活性获得提升。Serverless 让运维对开发透明，对于应用所需资源进行自动伸缩。FaaS 是 Serverless 的一种实现，则更加简化了开发运维的过程，从开发到最后测试上线都可以在一个集成开发环境中完成。无论哪一种场景，后台的运维平台的工作都是不可以缺少的，只是通过技术让扩容、容错等技术对开发人员透明，让效率更高。

## 7 云原生中间件



在云原生时代，传统中间件技术也演化升级为云原生中间件，云原生中间件主要包括网格化的服务架构、事件驱动技术、Serverless 等技术的广泛应用，其中网格化（Service Mesh）、微服务和 Serverless 前面都讲过了，这里主要讲云原生技术带来的不同点。

云原生中间件最大的技术特点就是中间件技术从业务进程中分离，变成与开发语言无关的普惠技术，只与应用自身架构和采用的技术标准有关，比如一个 PHP 开发的 REST 应用也会自动具备流量灰度发布能力、可观测能力，即使这个应用并没有采用任何服务化编程框架。

微服务架构一般包含下列组件：服务注册发现中心、配置中心、服务治理、服务网格、API 管理、运行时监控、链路跟踪等。随着 Kubernetes 的流行，Kubernetes 提供的基础部署运维和弹性伸缩能力已经可以满足多数中小企业的微服务运维要求。微服务与 Kubernetes 集成会是一个大趋势。

服务注册发现和配置中心的功能主要致力于解决微服务在分布式场景下的服务发现和分布式配置管理两个核心问题。随着云原生技术的发展，服务发现领域出现了两个趋势，一个是服务发现标准化（Istio），一个是服务下沉（CoreDNS）；配置管理领域也有两个趋势，一个是标准化（ConfigMap），一个是安全（Secret）。

提到事件驱动就必须先讲消息服务，消息服务是云计算 PaaS 领域的基础设施之一，主要用于解决分布式应用的异步通信、解耦、削峰填谷等场景。消息服务提供一种 BaaS 化的消息使用模式，用户无需预先购买服务器和自行搭建消息队列，也无需预先评估消息使用容量，只需要在云平台开通即用，按消息使用量收费。例如阿里云 MaaS（Messaging as a Service）提供的是一站式消息平台，提供业界最主流的开源消息队列托管服务，包括 Kafka、RabbitMQ、RocketMQ 等；也支持多种主流标准协议规范，包括 AMQP、MQTT、JMS、STOMP 等，以满足微服务、IoT、EDA、Streaming 等各种细分领域的需求。

事件驱动架构：由于 IoT、云计算技术的快速发展，事件驱动架构在未来将会被越来越多的企业采纳，通过事

件的抽象、异步化，来提供业务解耦、加快业务迭代。在过去事件驱动架构往往是通过消息中间件来实现，事件用消息来传递。进入云计算时代，云厂商提供更加贴近业务的封装，比如 Azure 提供了事件网格，把所有云资源的一些运维操作内置事件，用户可以自行编写事件处理程序实现运维自动化；AWS 则把所有云资源的操作事件都用 SNS 的 Topic 来承载，通过消息做事件分发，用户类似实现 WebHook 来处理事件。由于事件是异步触发的，天然适合 Serverless，所以现在很多云厂商都采用自身的 Serverless 服务来运行事件负载，包括阿里云 Function Compute、Azure Function、AWS Lambda 都和事件处理集成。今年阿里云也发布了最新产品 EventBridge，作为事件驱动和无服务器架构的核心承载产品。



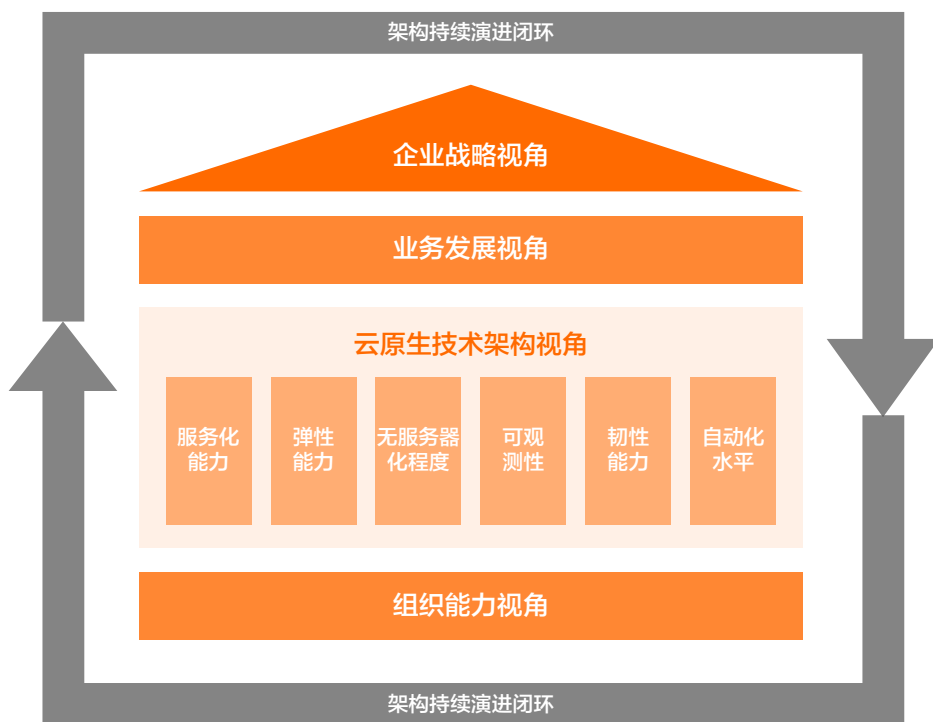
# 4

## 阿里巴巴云原生架构设计

### 1 ACNA ( Alibaba Cloud Native Architecting ) 架构设计方法

阿里巴巴为大量各行各业的企业客户提供基于阿里云服务的解决方案和最佳实践，帮助企业基于阿里云特别是云原生技术完成数字化转型，并积累了大量的方案和经验教训。阿里巴巴将企业的核心关注点、企业组织与 IT 文化、工程实施能力等方面与架构技术等相结合，形成了阿里巴巴独有的云原生架构设计方法——ACNA ( Alibaba Cloud Native Architecting ) 。

ACNA 是一个「4+1」的架构设计流程，「4」代表架构设计的关键视角，包括企业战略视角、业务发展视角、组织能力视角和云原生技术架构视角；「1」表示云原生架构的架构持续演进闭环。4 个架构视角和一个闭环的关系如下图所示：



ACNA 架构设计方法

ACNA 除了是一个架构设计方法，也包含了对云原生架构的评估体系、成熟度衡量体系、行业应用最佳实践、技术和产品体系、架构原则、实施指导等。

## 2 企业战略视角



任何架构都必须服务于企业战略，云原生架构也不例外。云原生架构和以往的架构升级不同，云原生架构不仅是一个技术升级，更是一个对企业核心业务生产流程（即通过软件开发和运营构建数字化业务）的重构，如果打一个比方的话，就像工业时代用更自动化的流水线替换手工作坊一样深刻。

企业必须规划清楚战略中业务战略与 IT 战略之间的关系，即 IT 战略只是服务好业务战略进行必要的技术支撑呢，还是 IT 战略本身也是业务战略的一部分。通常高科技公司本身会对云计算有更高的需求，比如通过大量使用云厂商提供的 AI 技术为用户提供智能化的用户体验，也会使用 IoT 和音视频技术为用户建立更广泛和生动的连接。实际上我们发现，在数字化转型的今天，越来越多的企业认为 IT 战略应该在企业战略中扮演技术赋能业务创新的重要角色，CTO（Chief Technology Officer）、CIO（Chief Information Officer）、CDO（Chief Data Officer）、CISO（Chief Information Security Officer）等岗位的设计也从一个层面表明了这些技术在企业战略中的位置。

## 3 业务发展视角



阿里巴巴总结为企业提供云服务和咨询的过程，发现数字化业务对技术架构的主要诉求是业务连续性、业务快速上线、成本以及科技赋能业务创新。业务连续性诉求主要包括了数字化业务必须能够持续为用户提供服务，不能因为软硬件故障或者 bug 导致业务不可用，也能够抵御黑客攻击、数据中心不可用、自然灾害等意外事故。此外，当业务规模快速增长时，不能因为软硬件资源来不及购买或者部署而导致不能拓展新用户。

市场瞬息万变，数字化业务由于比传统实体业务更灵活可变而被要求更快的推向市场能力，包括新业务快速构建的能力、现有业务快速更新的能力。这些能力诉求被云原生架构深刻理解并在产品、工具、流程等层面得到不同程度的处理，需要注意的是这个诉求同时对组织结构带来了新的要求，也可能要求应用进行彻底的重构（比如微服务化）。

云计算作为新的技术必须为企业释放成本红利，帮助企业从原来的 CAPEX 模式转变为 OPEX 模式，不用事先购买大批软硬件资源，而是用多少付多少；同时大量采用云原生架构也会降低企业开发和运维成本，有数据展示通过采用容器平台技术就降低了 30% 以上的运维支出。

传统模式下如果要使用高科技技术赋能业务，有一个冗长的选型、POC、试点和推广的过程，而大量使用云厂商和三方的云服务，由于这些云服务具备更快的连接和试错成本，且在不同技术的集成上具备统一平台和统一技术依赖的优势，从而可以让业务更快速的应用新技术进行创新。

## 4 组织能力视角



云原生架构涉及到的架构升级对企业中的开发、测试、运维等人员都带来了巨大的影响，技术架构的升级和实现需要企业中相关的组织匹配，特别是架构持续演进需要有类似“架构治理委员会”这样的组织，不断评估、检查架构设计与执行之间的偏差。

此外前面提到云原生服务中重要的架构原则就是服务化（包括微服务、小服务等），这个领域典型的一个原则就是“康威定律”，要求企业中让技术架构与企业沟通架构保持一致，否则会出现畸形的服务化架构实现。

## 5 云原生技术架构视角



### 1 服务化能力

用微服务或者小服务构建业务，分离大块业务中具备不同业务迭代周期的模块，并让业务以标准化 API 等方式进行集成和编排；服务间采用事件驱动的方式集成，减小相互依赖；通过可度量建设不断提升服务的 SLA 能力；

### 2 弹性能力

自动根据业务峰值、资源负载扩充或者收缩系统的规模；

### 3 无服务器化程度

在业务中尽量使用云服务而不是自己持有三方服务，特别是自己运维开源软件的情况；并让应用的设计尽量变成无状态的模式，把有状态部分保存到云服务中；尽量采用 FaaS、容器 / 应用无服务器的云服务；

### 4 可观测性

IT 设施需要被持续治理，任何 IT 设施中的软硬件发生错误后能够被快速修复，从而不会让这样的错误对业务带来影响，这就需要系统有全面的可观测性，从传统的日志方式、监控、APM 到链路跟踪、服务 QoS 度量；

## 5 韧性能力

除了包括服务化中常用的熔断、限流、降级、自动重试、反压等特性外，还包括高可用、容灾、异步化的特性；

## 6 自动化水平

关注整个开发、测试和运维三个过程的敏捷，推荐使用容器技术自动化软件构建过程、使用 OAM 标准化软件交付过程、使用 IaC ( Infrastructure as Code ) /GitOps 等自动化 CI/CD 流水线和运维过程；

## 7 安全能力

关注业务的数字化安全，在利用云服务加固业务运行环境的同时，采用安全软件生命周期开发，使应用符合 ISO27001、PCIDSS、等级保护等安全要求。这个维度作为基础维度，要求在架构评测中关注但是并不参与评分。

## 6 架构持续演进闭环



云原生架构演进是一个不断迭代的过程，每一次迭代都是从企业战略、业务诉求到架构设计与实施的一个完整闭环，整体关系如下图：



其中，

**关键输入：**企业战略视角、业务发展视角；

**关键过程：**识别业务痛点和架构债务、确定架构迭代目标、评估架构风险、选取云原生技术、制定迭代计划、架构评审和设计评审、架构风险控制、迭代验收和复盘。

## 7 云原生架构成熟度模型



由于云原生架构包含了 6 个关键架构维度（简称为 SESORA，Service + Elasticity + Serverless + Observability + Resilience + Automation），因此我们先定义关键维度的成熟度级别：

指标维度	ACNA-1 (0 分)	ACNA-2 (1 分)	ACNA-3 (2 分)	ACNA-4 (3 分)
服务化能力 (Service)	无 (单体应用)	部分服务化 & 缺乏治理 (自持技术，初步服务化)	全部服务化 & 有治理体系 (自持技术，初步服务化)	Mesh 化的服务体系 (云技术，治理最佳实践)
弹性能力 (Elasticity)	全人工扩缩容 (固定容量)	半闭环 (监控 + 人工扩缩容)	非全云方式闭环 (监控 + 代码伸缩，百节点规模)	基于云全闭环 (基于流量等多策略，万节点规模)
无服务器化程度 (Serverless)	未采用 BaaS	无状态计算委托给云 (计算、网络、大数据等)	有状态存储委托给云 (数据库、文件、对象存储等)	全无服务器方式运行 (Serverless/FaaS 运行全部代码)
可观测性 (Observability)	无	性能优化 & 错误处理 (日志分析、应用级监控、APM)	360 度 SLA 度量 (链路级 Tracing、Metrics 度量)	用户体验持续优化 (用观测大数据提升业务体验)
韧性能力 (Resilience)	无	十分钟级切流 (主备 HA、集群 HA、冷备容灾)	分钟级切流 (熔断、限流、降级、多活容灾等)	秒级切流、业务无感 (Serverless、Service Mesh 等)
自动化能力 (Automation)	无	基于容器的自动化 (基于容器做 CI/CD)	具备自描述能力的自动化 (提升软件交付自动化)	基于 AI 的自动化 (自动化软件交付和运维)

云原生架构成熟度模型：关键指标维度

由于云原生架构包含了 6 个关键架构维度（简称为 SESORA，Service + Elasticity + Serverless + Observability + Resilience + Automation），因此我们先定义关键维度的成熟度级别：

云原生架构 成熟度	零级	基础级	发展级	成熟级
级别和定义	完全传统架构 (未使用云计算或者 云的技术能力)	小于等于 10 分	大于 10 且小于 16 分 且无 ACNA-1 级	大于等于 16 分 且无 ACNA-2 级



云原生架构成熟度模型

# 5

## 阿里巴巴云原生产品介绍

### 1 云原生产品家族



阿里巴巴云原生产品家族包括容器产品家族、微服务产品家族、Serverless 产品家族、Service Mesh 产品家族、消息产品、云原生数据库家族、云原生大数据产品家族等。

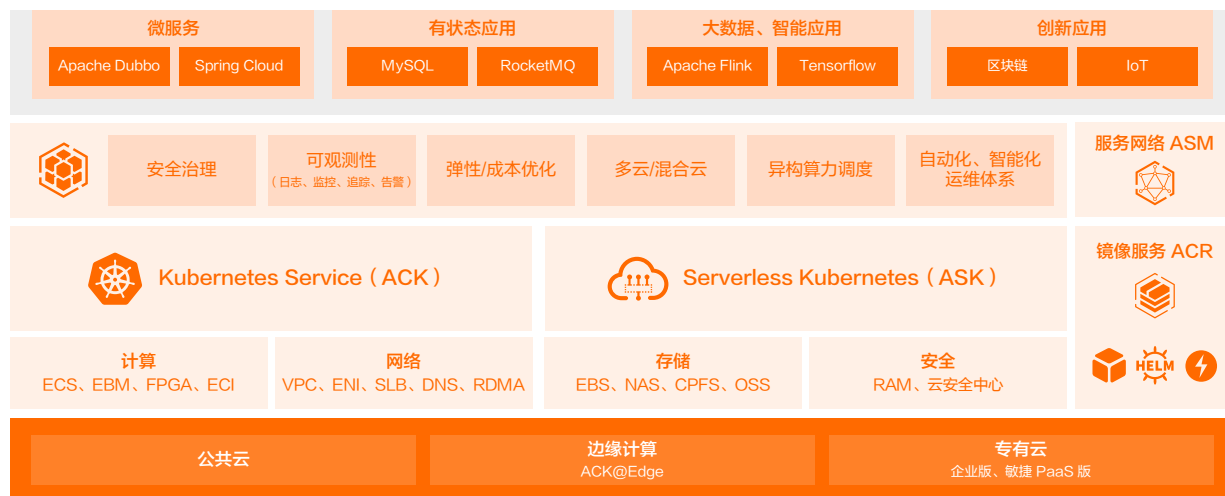


阿里巴巴云原生产品家族

### 2 容器产品家族



阿里云容器服务自从 2016 年 5 月正式推出，历经 4 年时间，服务了全球上万家企业客户。阿里云容器服务产品家族可以在公共云、边缘计算和专有云环境提供企业容器平台。阿里云容器产品以容器服务 Kubernetes 版 (ACK) 和 Serverless Kubernetes (ASK) 为核心，它们构建在阿里云基础设施



阿里云容器产品大图

之上，计算、存储、网络、安全等，并为企业客户提供标准化接口、优化的能力和简化的用户体验。通过的 CNCF Kubernetes 一致性兼容认证的 ACK 为企业提供了一系列业务所需的必备能力，如安全治理、端到端可观测性、多云混合云等。

**镜像服务（ACR）** 作为企业云原生应用资产管理的核心，企业可以借之高效管理 Docker 镜像、Helm Chart 等应用资产，并与 CI/CD 工具结合，组成完整 DevSecOps 流程。

### 3 微服务产品家族



**EDAS（企业分布式应用服务）** 是一个面向微服务应用的应用全生命周期 PaaS 平台，产品全面支持 HSF、Dubbo、Spring Cloud 技术体系，提供 ECS 集群和 K8s 集群的应用开发、部署、监控、运维等全栈式解决方案。

**MSE（微服务引擎）** 是一个面向业界主流开源微服务框架 Spring Cloud、Dubbo 的微服务平台，包含治理中心、托管注册 / 配置中心，一站式的解决方案帮助用户提升微服务的开发效率和线上稳定性。

**ACM（应用配置管理）**，是一款应用配置中心产品，实现在微服务、DevOps、大数据等场景下的分布式配置服务，保证配置的安全合规。

**CSB Micro Gateway（微服务网关服务）** 针对微服务架构下 API 开放的特点，提供能与微服务环境的治理策略无缝衔接的网关服务，实现高效的微服务 API 开放。



**GTS（全局事务服务）**用于实现分布式环境下特别是微服务架构下的高性能事务一致性，可以与多种数据源、微服务框架配合使用，实现分布式数据库事务、多库事务、消息事务、服务链路级事务及各种组合。

**ARMS（应用实时监控服务）**是一款应用性能管理产品，包含前端监控、应用监控和 Prometheus 监控三大子产品，涵盖了浏览器、小程序、APP、分布式应用和容器环境等性能管理，实现全栈式性能监控和端到端全链路追踪诊断。

**链路追踪(Tracing Analysis)**为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等工具，能够帮助开发者快速分析和诊断分布式应用架构下的性能瓶颈，提高微服务时代下的开发诊断效率。

**PTS（Performance Testing Service）**是一款云化测试工具，提供性能测试、API 调试和监测等多种能力，紧密结合监控、流控等产品提供一站式高可用能力，高效检验和管理业务性能。

## 4 Serverless 产品家族



**FC（函数计算）**是一个事件驱动的全托管 Serverless 计算服务，用户无需管理服务器等基础设施，只需编写代码并上传，函数计算会准备好计算资源，并以弹性、可靠的方式运行业务代码。

**SAE（Serverless 应用引擎）**实现了 Serverless 架构 + 微服务架构的完美融合，真正按需使用、按量计费，节省闲置计算资源，同时免去 IaaS 运维，有效提升开发运维效率；SAE 支持 Spring Cloud、Dubbo 和 HSF 等流行的微服务架构。

**Serverless 工作流**是一个用来协调多个分布式任务执行的全托管 Serverless 云服务，致力于简化开发和运行业务流程所需要的任务协调、状态管理以及错误处理等繁琐工作，让用户聚焦业务逻辑开发。用户可以用顺序、分支、并行等方式来编排分布式任务，服务会按照设定好的顺序可靠地协调任务执行，跟踪每个任务的状态转换，并在必要时执行用户定义的重试逻辑，以确保工作流顺利完成。

## 5 Service Mesh 产品家族



**托管服务网格（ASM）**提供全托管的微服务应用流量管理平台，兼容 Istio 的同时，支持多个 Kubernetes 集群中应用的统一流量管理，为容器和虚拟机中应用服务提供一致的通信、安全和可观测能力。

**AHAS（应用高可用服务）**是专注于提高应用及业务高可用的工具平台，目前主要提供应用架构探测感知，故障注入式高可用能力评测和流控降级高可用防护三大核心能力，通过各自的工具模块可以快速低成本的在营销活动场景、业务核心场景全面提升业务稳定性和韧性。

## 6 消息产品家族



**消息队列 RocketMQ 版**是阿里云基于 Apache RocketMQ 构建的低延迟、高并发、高可用、高可靠的分布式消息中间件。该产品最初由阿里巴巴自研并捐赠给 Apache 基金会，服务于阿里集团 13 年，覆盖全集团所有业务，支撑千万级并发、万亿级数据洪峰，历年刷新全球最大的交易消息流转记录。

**消息队列 Kafka 版**是阿里云基于 Apache Kafka 构建的高吞吐量、高可扩展性的分布式消息队列服务，广泛用于日志收集、监控数据聚合、流式数据处理、在线和离线分析等，是大数据生态中不可或缺的产品之一，阿里云提供全托管服务，用户无需部署运维，更专业、更可靠、更安全。

**消息队列 AMQP 版**由阿里云基于 AMQP 标准协议自研，完全兼容 RabbitMQ 开源生态以及多语言客户端，打造分布式、高吞吐、低延迟、高可扩展的云消息服务。

**微消息队列 MQTT 版**是专为移动互联网 (MI)、物联网 (IoT) 领域设计的消息产品，覆盖互动直播、金融支付、智能餐饮、即时聊天、移动 Apps、智能设备、车联网等多种应用场景；通过对 MQTT、WebSocket 等协议的全面支持，连接端和云之间的双向通信，实现 C2C、C2B、B2C 等业务场景之间的消息通信，可支撑千万级设备与消息并发，真正做到万物互联。

**阿里云消息服务 MNS** 是一种高效、可靠、安全、便捷、可弹性扩展的分布式消息服务，能够帮助应用开发者在他们应用的分布式组件上自由的传递数据、通知消息，构建松耦合系统。

**事件总线 EventBridge** 是阿里云提供的一款无服务器事件总线服务，支持阿里云服务、自定义应用、SaaS 应用以标准化、中心化的方式接入，并能够以标准化的 CloudEvents 1.0 协议在这些应用之间路由事件，帮助用户轻松构建松耦合、分布式的事件驱动架构。

## 7 云原生数据库产品家族



**PolarDB** 是阿里巴巴自主研发的下一代关系型分布式云原生数据库，目前兼容三种数据库引擎：MySQL、PostgreSQL、高度兼容 Oracle 语法；计算能力最高可扩展至 1000 核以上，存储容量最高

可达 100T。PolarDB 经过阿里巴巴双十一活动的最佳实践，让用户既享受到开源的灵活性与价格，又享受到商业数据库的高性能和安全性。

**PolarDB-X**（原 DRDS 升级版）是由阿里巴巴自主研发的云原生分布式数据库，融合分布式 SQL 引擎 DRDS 与分布式自研存储 X-DB，专注解决海量数据存储、超高并发吞吐、大表瓶颈以及复杂计算效率等数据库瓶颈难题，历经各届天猫双 11 及阿里云各行业客户业务的考验，助力企业加速完成业务数字化转型。

## 8 云原生大数据产品家族



**云原生数据仓库 AnalyticDB MySQL 版**（简称 ADB，原分析型数据库 MySQL 版）是一种支持高并发低延时查询的新一代云原生数据仓库，全面兼容 MySQL 协议以及 SQL:2003 语法标准，可以对海量数据进行即时的多维分析透视和业务探索，快速构建企业云上数据仓库。产品规格按需可选，基础版成本最低，适合 BI 查询应用；集群版提供高并发数据实时写入和查询能力，适用于高性能应用；弹性模式版本存储廉价按量计费，适用于 10TB 以上数据上云场景。

**云原生数据仓库 AnalyticDB PostgreSQL 版**，支持标准 SQL 2003，兼容 PostgreSQL / Greenplum，高度兼容 Oracle 语法生态；具有存储计算分离，在线弹性平滑扩容的特点；既支持任意维度在线分析探索，也支持高性能离线数据处理；是面向互联网，金融，证券，保险，银行，数字政务，新零售等行业有竞争力的数据仓库方案。

# 6



## 各个行业面临的挑战及解决方案

### 云原生解决方案

随着云计算的普及与云原生的广泛应用，越来越多的从业者、决策者清晰地认识到「云原生化将成为企业技术创新的关键要素，也是完成企业数字化转型的最短路径」。因此，具有前瞻思维的互联网企业从应用诞生之初就扎根于云端，谨慎稳重的新零售、政府、金融、医疗等领域的企业与机构也逐渐将业务应用迁移上云，深度使用云原生技术与云原生架构。面对架构设计、开发方式到部署运维等不同业务场景，基于云原生架构的应用通常针对云的技术特性进行技术生命周期设计，最大限度利用云平台的弹性、分布式、自助、按需等产品优势。借助以下几个典型实践案例，我们来看看企业如何使用云原生架构解决交付周期长、资源利用率低等实际业务问题。

#### 1 案例一：申通快递核心业务系统云原生上云案例



##### 1 背景和挑战

作为发展最为迅猛的物流企业之一，申通快递一直积极探索技术创新赋能商业增长之路，以期达到降本提效目的。目前，申通快递日订单处理量已达千万量级，亿级别物流轨迹处理量，每天产生数据已达到 TB 级别，使用 1300+ 个计算节点来实时处理业务。

过往申通快递的核心业务应用运行在 IDC 机房，原有 IDC 系统帮助申通安稳度过早期业务快速发展期。但伴随着业务体量指数级增长，业务形式愈发多元化。原有系统暴露出不少问题，传统 IOE 架构、各系统架构的不规范、稳定性、研发效率都限制了业务高速发展的可能。软件交付周期过长，大促保障对资源的特殊要求难实现、系统稳定性难以保障等业务问题逐渐暴露。

在与阿里云进行多次需求沟通与技术验证后，申通最终确定阿里云为唯一合作伙伴，采用云原生技术和架构实现核心业务搬迁上阿里云。2019 年开始将业务逐步从 IDC 迁移至阿里云。目前，核心业务系统已经在阿里云上完成流量承接，为申通提供稳定而高效的计算能力。

##### 2 云原生解决方案

申通核心业务系统原架构基于 Vmware+Oracle 数据库进行搭建。随着搬迁上阿里云，架构全面转型为基于 Kubernetes 的云原生架构体系。其中，引入云原生数据库并完成应用基于容器的微服务改造是整个应用服务架构重构的关键点。

#### • 引入云原生数据库

通过引入 OLTP 跟 OLAP 型数据库，将在线数据与离线分析逻辑拆分到两种数据库中，改变此前完全依赖 Oracle 数据库的现状。满足在处理历史数据查询场景下 Oracle 数据库所无法支持的实际业务需求。

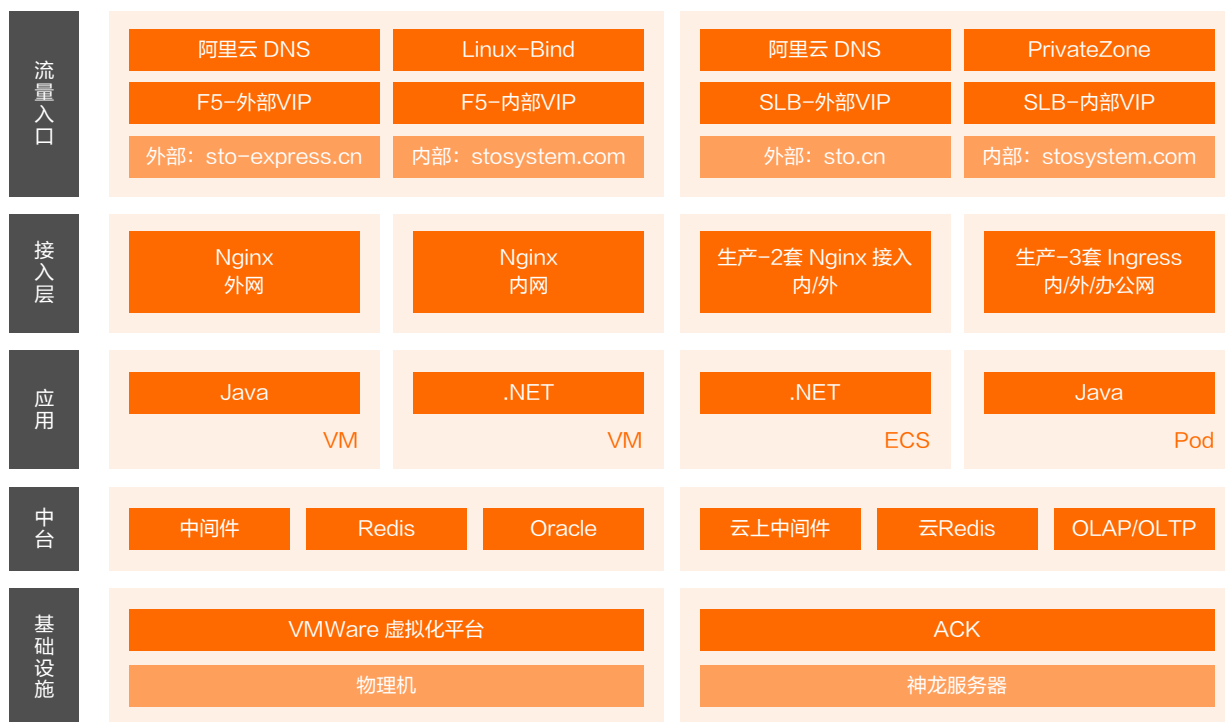
#### • 应用容器化

伴随着容器化技术的引进，通过应用容器化有效解决了环境不一致的问题，确保应用在开发、测试、生产环境的一致性。与虚拟机相比，容器化提供了效率与速度的双重提升，让应用更适合微服务场景，有效提升产研效率。

#### • 微服务改造

由于过往很多业务是基于 Oracle 的存储过程及触发器完成的，系统间的服务依赖也需要 Oracle 数据库 OGG 同步完成。这样带来的问题就是系统维护难度高且稳定性差。通过引入 Kubernetes 的服务发现，组建微服务解决方案，将业务按业务域进行拆分，让整个系统更易于维护。

综合考虑申通实际业务需求与技术特征，最终选择了「阿里云 ACK+ 神龙 + 云数据库」的云原生解决方案，从而实现核心应用迁移上阿里云。



上云混合态架构

申通核心业务上云架构示意图

## 2.1 架构阐述

基础设施，全部计算资源取自阿里云的神龙裸金属服务器。相较于一般云服务器（ECS），Kubernetes 搭配神龙服务器能够获得更优性能及更合理的资源利用率。且云上资源按需取量，对于拥有大促活动等短期大流量业务场景的申通而言极为重要。相较于线下自建机房、常备机器，云上资源随取随用。在大促活动结束后，云上资源使用完毕后即可释放，管理与采购成本更低，相应效率。

流量接入，阿里云提供两套流量接入，一套是面向公网请求，另外一套是服务内部调用。域名解析采用云 DNS 及 PrivateZone。借助 Kubernetes 的 Ingress 能力实现统一的域名转发，以节省公网 SLB 的数量，提高运维管理效率。

## 2.2 平台层

基于 Kubernetes 打造的云原生 PaaS 平台优势明显突出。

- 打通 DevOps 闭环，统一测试，集成，预发、生产环境；
- 天生资源隔离，机器资源利用率高；
- 流量接入可实现精细化管理；
- 集成了日志、链路诊断、Metrics 平台；
- 统一 ApiServer 接口和扩展，天生支持多云跟混合云部署。

## 2.3 应用服务层

每个应用都在 Kubernetes 上面创建单独的一个 Namespace，应用跟应用之间实现资源隔离。通过定义各个应用的配置 Yaml 模板，当应用在部署时直接编辑其中的镜像版本即可快速完成版本升级，当需要回滚时直接在本地启动历史版本的镜像快速回滚。

## 2.4 运维管理

线上 Kubernetes 集群采用阿里云托管版容器服务，免去了运维 Master 节点的工作，只需要制定 Worker 节点上线及下线流程即可。同时业务系统均通过阿里云的 PaaS 平台完成业务日志搜索，按照业务需求投交扩容任务，系统自动完成扩容操作，降低了直接操作 Kubernetes 集群带来的业务风险。

## 3 应用效益

- 成本方面：使用公有云作为计算平台，可以让企业不必因为业务突发增长需求，而一次性投入大量资金成本用于采购服务器及扩充机柜。在公共云上可以做到随用随付，对于一些创新业务想做技术调研十分便捷。用完即释放，按量付费。另外云产品都免运维自行托管在云端，有效节省人工运维成本，让企业更专注于核心业务。

稳定性方面：首先，云上产品提供至少 5 个 9 以上的 SLA 服务确保系统稳定，而自建系统稳定性相去甚远。其次，部分开源软件可能存在功能 bug，造成故障隐患。最后，在数据安全方面云上数据可以轻松实现异地备份，阿里云数据存储体系下的归档存储产品具备高可靠、低成本、安全性、存储无限等特点，让企业数据更安全。

效率方面：借助与云产品深度集成，研发人员可以完成一站式研发、运维工作。从业务需求立项到拉取分支开发，再到测试环境功能回归验证，最终部署到预发验证及上线，整个持续集成流程耗时可缩短至分钟级。排查问题方面，研发人员直接选择所负责的应用，并通过集成的 SLS 日志控制台快速检索程序的异常日志进行问题定位，免去了登录机器查日志的麻烦。

赋能业务：阿里云提供超过 300 余种的云上组件，组件涵盖计算、AI、大数据、IOT 等等诸多领域。研发人员开箱即用，有效节省业务创新带来的技术成本。

## 2 案例二：完美日记电商业务案例



### 1 背景和挑战

作为美妆届的璀璨新星，完美日记（Perfect Diary）上线不到两年即成为天猫彩妆销冠，截至 2020 年 4 月，品牌 SKU 超过 700 个，全网用户粉丝数量超过 2500 万，月曝光量 10 亿+。

伴随着公司业务高速发展，技术运维对于面临着非常严峻的挑战。伴随着“双 11”电商大促、“双 12”购物节、小程序、网红直播带货呈现爆发式增长趋势，如何确保微商城系统稳定顺畅地运行成为完美日记面对的首要难题。其中，比较突出几个挑战包含以下几点：

- 系统开发迭代快，线上问题较多，定位问题耗时较长；
- 频繁大促，系统稳定性保障压力很大，第三方接口和一些慢 SQL 存在导致严重线上故障的风险；
- 压测与系统容量评估工作相对频繁，缺乏常态化机制支撑；
- 系统大促所需资源与日常资源相差较大，需要频繁扩缩容。

### 2 云原生解决方案

完美日记与阿里云一起针对所面临问题以及未来业务规划进行了深度沟通与研讨。通过阿里云原生应用稳定性解决方案以解决业务问题。引入阿里云容器服务 ACK、Spring Cloud Alibaba、PTS、AHAS、链路追踪等配套产品，对应用进行容器化改造部署，优化配套的测试、容量评估、扩所容等研发环节，提升产研效率。





方案的关键点是：

- 通过容器化部署，利用阿里云容器服务的快速弹性应对大促时的资源快速扩容。
- 提前接入链路追踪产品，用于对分布式环境下复杂的服务调用进行跟踪，对异常服务进行定位，帮助客户在测试和生产中快速定位问题并修复，降低对业务的影响。
- 使用阿里云性能测试服务（PTS）进行压测，利用秒级流量拉起、真实地理位置流量等功能，以最真实的互联网流量进行压测，确保业务上线后的稳定运营。
- 采集压测数据，解析系统强弱依赖关系、关键瓶颈点，对关键业务接口、关键第三方调用、数据库慢调用、系统整体负载等进行限流保护。
- 配合阿里云服务团队，在大促前进行 ECS/RDS/ 安全等产品扩容、链路梳理、缓存 / 连接池预热、监控大屏制作、后端资源保障演练等，帮助大促平稳进行。

### 3 应用收益

**高可用：**利用应用高可用服务产品（AHAS）的限流降级和系统防护功能，对系统关键资源进行防护，并对整体系统水位进行兜底，确保大促平稳进行，确保顺畅的用户体验。

**容量评估：**利用性能测试服务（PTS）和业务实时监控（ARMS）对系统单机能力及整体容量进行评估，对单机及整体所能承载的业务极限量进行提前研判，以确保未来对业务大促需求可以做出合理的资源规划和成本预测。

**大促保障机制：**通过与阿里云服务团队的进行多次配合演练，建立大促保障标准流程及应急机制，达到大促保障常态化。

### 4 客户声音

“使用 ACK 容器服务可以帮助我们快速拉起测试环境，利用 PTS 即时高并发流量压测确认系统水位，结合 ARMS 监控，诊断压测过程中的性能瓶颈，最后通过 AHAS 对突发流量和意外场景进行实时限流降级，加上阿里云团队保驾护航，保证了我们每一次大促活动的系统稳定性和可用性，同时利用 ACK 容器快速弹性扩缩容，节约服务器成本 50% 以上。” ——完美日记技术中台负责人

### 3 案例三：特步业务中台案例（零售、公共云）



#### 1 背景和挑战

成立于 2001 年的特步，作为中国领先的体育用品企业之一，门店数 6230 家。2016 年，特步启动集团第三次战略升级，打造以消费者体验为核心的“3+”（互联网+、体育+和产品+）的战略目标，积极拥抱云计算、大数据等新技术，实现业务引领和技术创新，支撑企业战略变革的稳步推进。在集团战略的促使下，阿里云中间件团队受邀对特步 IT 信息化进行了深度调研，挖掘阻碍特步战略落地的些许挑战：

- 商业套件导致无法满足特步业务多元化发展要求，例如多品牌拆分重组所涉及的相关业务流程以及组织调整。对特步而言，传统应用系统都是紧耦合，业务的拆分重组意味着必须重新实施部署相关系统。
- IT 历史包袱严重，内部烟囱系统林立。通过调研，阿里云发现特步烟囱系统多达六十三套，仅 IT 供应商就有三十余家。面对线上线下业务整合涉及到的销售、物流、生产、采购、订货会、设计等不同环节及场景，想要实现全渠道整合，需要将几十套系统全部打通。
- 高库存、高缺货问题一直是服装行业的死结，特步同样被这些问题困扰着。系统割裂导致数据无法实时在线，并受限传统单体 SQLServer 数据库并发限制，6000 多家门店数据只能采用 T+1 方式回传给总部，直接影响库存高效协同周转。
- IT 建设成本浪费比较严重，传统商业套件带来了“烟囱式”系统的弊端，导致很多功能重复建设、重复数据模型以及不必要的重复维护工作。

#### 2 云原生解决方案

阿里云根据特步业务转型战略需求，为之量身打造了基于云原生架构的全渠道业务中台解决方案，将不同渠道通用功能在云端合并、标准化、共享，衍生出全局共享的商品中心、渠道中心、库存中心、订单中心、营销中心、用户中心、结算中心。无论哪个业务线、哪个渠道、哪个新产品诞生或调整，IT 组织都能根据业务需求，基于共享服务中心现有模块快速响应，打破低效的“烟囱式”应用建设方式。全渠道业务中台遵循互联网架构原则，规划线上线下松耦合云平台架构，不仅彻底摆脱传统 IT 拖业务后腿的顽疾并实现灵活支撑业务快速创新，将全渠道数据融通整合在共享服务中心平台上，为数据化决策、精准营销、统一用户体验奠定了良好的产品与数据基础，让特步真正走上了“互联网+”的快车道。

2017 年 1 月特步与阿里云启动全渠道中台建设，耗时 6 个月完成包括需求调研、中台设计、研发实施、测试验证等在内的交付部署，历经 4 个月实现全国 42 家分公司、6000+ 门店全部上线成功。以下是特步全渠道业务中台总体规划示意图，



下面是基于云原生中间件的技术架构示意图



架构的关键点：

- 应用侧：新技术架构全面承载面向不同业务部门的相关应用，包括门店 POS、电商 OMS、分销商管理供销存 DRP、会员客户管理 CRM。此外，在全渠道管理方面也会有一些智能分析应用，比如库存平衡，同时可以通过全渠道运营平台来简化全渠道的一些配置管理。所有涉及企业通用业务能力比如商品、订单等，可以直接调用共享中心的能力，让应用“更轻薄”。

- 共享中心：全渠道管理涉及到参与商品品类、订单寻源、共享库存、结算规则等业务场景，也涉及与全渠道相关的会员信息与营销活动等。这些通用业务能力全部沉淀到共享中心，向不同业务部门输出实时 / 在线 / 统一 / 复用的能力。直接将特步所有订单 / 商品 / 会员等信息融合、沉淀到一起，从根本上消除数据孤岛。
- 技术层：为了满足弹性、高可用、高性能等需求，通过 Kubernetes/EDAS/MQ/ARMS/PTS 等云原生中间件产品，目前特步核心交易链路并发可支撑 10w/tps 且支持无线扩容提升并发能力。采用阿里历经多年双 11 考验的技术平台，稳定性 / 效率都得到了高规格保障，让开发人员能够更加专注在业务逻辑实现，再无后顾之忧。
- 基础设施：底层的计算、存储、网络等 IaaS 层资源。
- 后台系统：客户内部的后台系统，比如 SAP、生产系统、HR/OA 等。

### 3 应用收益

全渠道业务中台为特步核心战略升级带来了明显的变化，逐步实现了 IT 驱动业务创新。

经过中台改造后，POS 系统从离线升级为在线化。包括收银、库存、会员、营销在内的 POS 系统核心业务全部由业务中台统一提供服务，从弱管控转变为集团强管控，集团与消费者之间真正建立起连接，为消费者精细化管理奠定了坚实的基础。

中台的出现，实现了前端渠道的全局库存共享，库存业务由库存中心实时处理。借助全局库存可视化，交易订单状态信息在全渠道实时流转，总部可直接根据实时经营数据对线下店铺进行销售指导，实现快速跨店商品挑拨。中台上线后，售罄率提升 8%，缺货率降低 12%，周转率提升 20%，做到赋能一线业务。

IT 信息化驱动业务创新，通过共享服务中心将不同渠道类似功能在云端合并共享，打破低效的“烟囱式”应用建设方式，吸收互联网 DDD 领域驱动设计原则，设计线上线下松耦合云平台架构，不仅彻底摆脱了传统 IT 拖业务后腿的顽疾并灵活支撑业务快速创新。全渠道数据融通整合在共享服务中心平台上，沉淀和打造出特步的核心数据资产，培养出企业中最稀缺的“精通业务，懂技术”创新人才，使之在企业业务创新、市场竞争中发挥核心作用。截止 2019 年初，业务部门对 IT 部门认可度持续上升，目前全渠道业务支撑系统几乎全部自主搭建，80% 前台应用已经全部运行在中台之上，真正实现技术驱动企业业务创新。

## 4 中国联通号卡业务云化案例（传统业务，专有云）



### 1 背景和挑战

一直以来，作为国内三大运营商之一的中国联通始终以新理念新模式促进信息基础设施升级。推动网络资源优化演进升级，持续提升网络竞争力。中国联通 BSS 核心系统随着用户规模增长，个性化业务复杂性日趋提升、竞争压力不断加大，以 cBSS1.0 为核心的集中业务系统，在系统架构、支撑能力、业务运营等方面无法满足一体化运营发展要求的问题日益突出，主要体现在：

- 难以满足个性化业务模式和新业务场景需求，客户与一线营业人员体感较差。
- 日常运营面临的新问题数量、种类多，待优化提升的需求压力大。
- 传统系统架构制肘了开放性、扩展性、体验性、稳定性均的提升，支撑能力与支撑模式无法满足集团 IT 集中一体化运营的业务需要。
- 外围对接系统众多。作为全国唯一的号卡资源管理方和服务提供方，号卡资源集中共享系统需要与包括 cBSS、商城网厅、省分 BSS、北六 ESS、总部 CRM、沃易购等在内的四十余种外围系统对接。因为要向全国所有销售系统提供号卡资源服务，对号卡资源集中共享系统提出了非常严苛的性能要求，进而也提出了更高规格的扩展性要求。
- 面向全渠道，号卡资源集中共享系统除了要向目前已存自有渠道（营业厅、电子渠道）、社会渠道提供高效稳定的服务外，也要考虑未来面向互联网渠道等新兴渠道开放服务能力。这都对号卡资源集中共享系统的服务化、能力开放能力提出了很高的要求。
- 高性能挑战，满足全国要求的性能要求估算：面向未来的互联网营销模式以及物联网等新业务领域可能带来的更高要求。传统架构存在无法横向扩展、同步调用流程长的问题。引入分布式架构成为迫在眉睫的需求。
- 高可用挑战，号卡资源作为中国联通的核心资源的同时，号卡资源管理同样是销售流程的关键环节，号卡资源管理系统直接关系到销售业务是否可正常开展全国集中的号卡资源管理系统、直接关系到全国销售业务是否能正常开展，引入分布式架构的同时必须确保高可用。

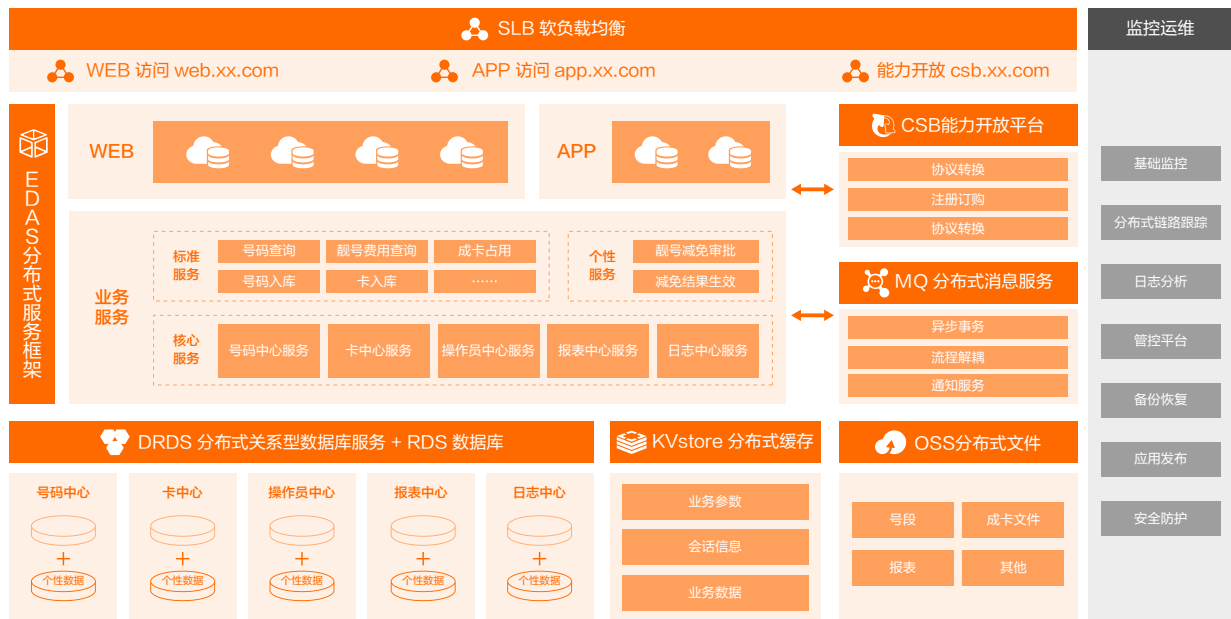
## 2 云原生解决方案

中国联通与阿里云结合阿里云原生 PaaS、阿里飞天操作系统、阿里云原生数据库以及中国联通天宫平台，共同研发运营商级专有云平台“天宫云”，支撑中国联通核心业务应用。合作过程中，阿里云从互联网架构，业务中台架构，云原生技术等方面出发，基于阿里云业务中台，互联网技术上的最佳实践，云原生完整技术产品的整体架构方案。阿里云原生 PaaS 平台为业务能力层和核心能力层及能力开放管理平台提供包括：分布式服务框架、分布式消息服务、分布式数据服务、分布式监控等云原生技术服务、管理业务能力、技术组件、云原生运维管理服务在内的基本技术支持。

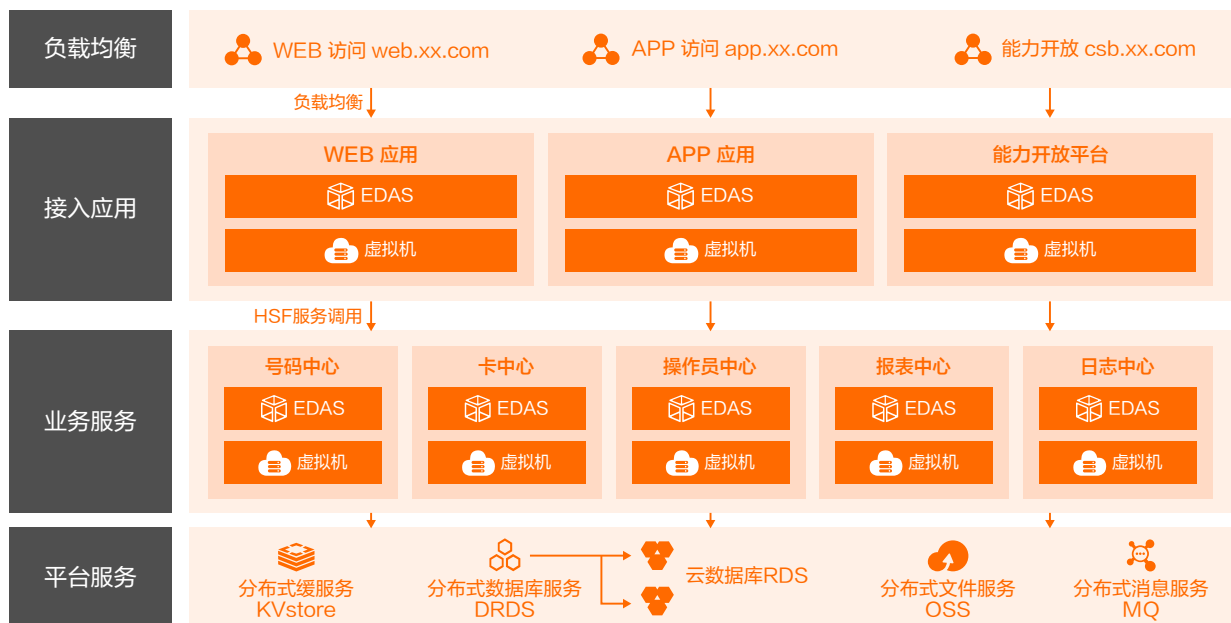
全面云化的系统：基于“平台 + 应用”的三层分布式架构模式，采取高内聚、低耦合、易扩展、服务化的设计原则，由去中心化的服务框架为应用提供服务，由能力开放平台提供能力集成和能力开放。

服务化设计：充分发挥平台线性扩展能力和服务化设计优势，纵向分多个中心、中心分多个模块，横向分层设计，从而实现业务需求的灵活响应和快速支撑。

下面是中国联通号卡应用的技术架构示意图。



下面是中国联通号卡应用的部署架构示意图。



部署架构的优势：

- 共享平台服务：平台组件服务单独部署，自身分布式设计，全局应用、服务共享使用；
- 分层分中心部署：应用、业务服务分层、分中心、分区部署，高度灵活、稳定、可扩展。

### 3 应用收益

彻底解决号码资源在各省、多渠道、多系统共同管理问题，过去号码状态同步问题多，一号多卖现象严重，有效降低业务冲突。

解决销售过程中由于传统架构瓶颈造成的各类业务问题，开卡业务效率提升了 10 倍，单日选号访问量提升了 3 倍，需求响应时间缩短了 50%，上层业务效率显著提升。

支撑业务快速创新，比如：支撑微信红点业务推广，由推广前每天访问量 1000 万 +，到推广一周后快速上升到 1.1 亿 +，完全满足业务部门销售策略支持需求。

实现集中管控，资源合理分配，彻底解决数据分散、号码利用率低问题。

## 5 Timing App 的 Serverless 实践案例



### 1 背景和挑战

作为广受好评的学习应用，Timing App 专注于帮助社区用户提升学习凝聚力，达成学习目标。目前已有超过 700 万人通过 Timing 进行高效学习。与传统在线学习应用不同，Timing app 提供了 Timing 自习室、图书馆学习、视频打卡、学习日记、契约群、学习服务等多类具有社交性质的在线教育服务，帮助用户找到自己的学习节奏，找到坚持学习的一万种理由。Timing 业务本身具有潮汐特性，用户访问主要集中在晚间和节假日。受疫情影响，春节期间峰值流量暴增 4 倍，公司面临较大的运维成本压力。在用户、流量爆发式增长背景下，Timing App 不得不直面以下四大痛点：

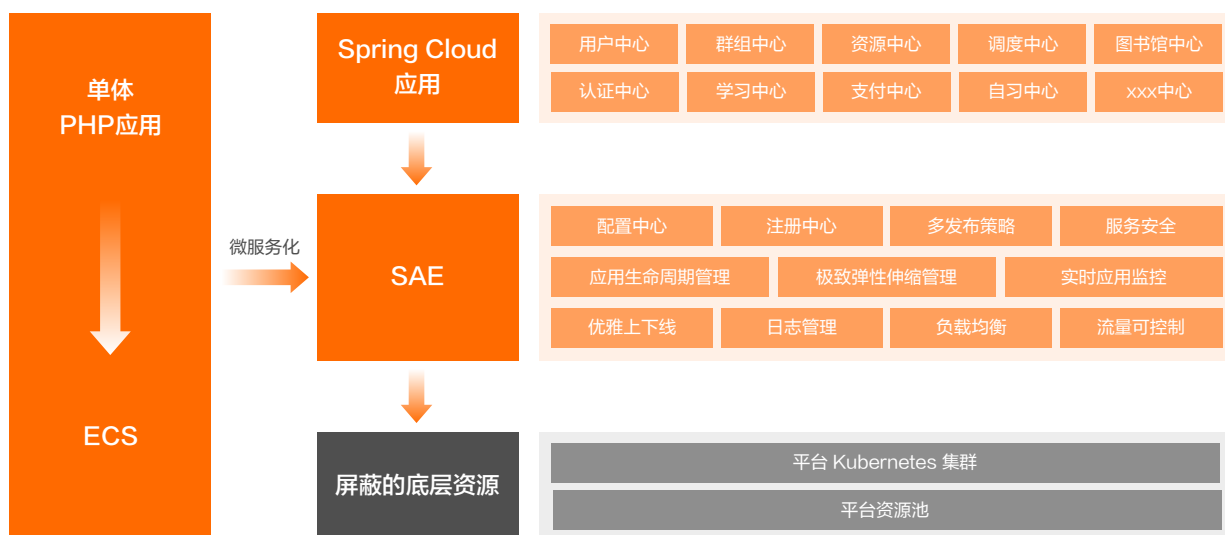
- 系统稳定性差。原有 PHP 单应用架构系统无法做到线性快速扩容，在业务高峰时段，系统问题频繁发生，严重影响用户体验。
- 产品迭代缓慢。随着业务的高速发展，原有单体架构对于产品的迭代力不从心，没法快速响应研发需求。
- 资源使用浪费。由于业务具有非常强的流量潮汐特征，需要按照业务高峰阶段进行资源保有配置，造成资源的浪费。
- 技术成本昂贵。以前的团队除了技术负责人及少数团队新成员外，基本缺乏微服务架构实战经验。想要实现微服务改造，急需能够快速上手的平台支撑，需要最大限度降低底层 IaaS、容器以及常用微服务套件的学习成本。

### 2 云原生解决方案



阿里云应用引擎 Serverless (SAE)，基于 Serverless 架构，屏蔽了底层 IaaS 运维和 K8s 细节，区别于 FaaS 形态的 Serverless 产品，用户无需修改编程模型，零代码改造就能直接使用。同时，完美结合 Spring Cloud/Dubbo 等微服务架构，提供应用发布、管理和服务治理等应用全生命周期的服务，完美贴合 Timing 的技术需求：极限弹性伸缩，应用生命周期灵活管理，完美支持主流微服务架构。

下图的方案架构示意图。



方案的关键优势是：

- 利用弹性伸缩，应对不确定突发流量。提供秒级自动弹性 & 定时弹性能力，帮助应用轻松应对大促峰值流量，保证 SLA 的同时也节省机器保有成本。多适用于互联网、游戏、在线教育行业。
- 应用环境按需灵活启停，节约成本。提供了一键启停开发测试环境的能力，即开即用，节省成本，方便运维。适用于对成本敏感、云上有多套环境但部分环境闲置率较高的企业型客户（不限行业）。
- 中小企业快速构建云上微服务应用。帮助用户屏蔽底层 IaaS 购买和运维细节、底层 K8s 细节，低门槛部署微服务应用。适用于初创型 / 上升期的公司（不限行业），业务增长很快，对增长有较高预期，但人员配置跟不上。
- 整体技术架构更为清晰，每个服务相互独立且职责明确。加之阿里云应用引擎 Serverless（SAE）加持，让客户只关注在业务层，做好产品。

### 3 应用收益

Timing App 在较短时间内和阿里云共同梳理业务结构，将之前的 PHP 单体应用转型为了微服务架构，包括用户中心、群组中心、资源中心、调度中心、图书馆中心、认证中心、学习中心、支付中心、自习中心等九大业务模块。方案实施后，Timing App 实现了降本增效，护航系统稳定性，为提升用户体验，增加用户粘性作出了重要贡献。

**降成本：**大幅节省自建微服务架构的云服务器成本。基于秒级弹性能力，无需长期保有固定资源，按需自动弹、按分钟计费，极大提升了资源利用率。

**提效率：**屏蔽了底层 IaaS 购买、底层 Kubernetes 细节和运维的烦恼，低门槛部署 Dubbo/Spring Cloud 等微服务应用，支撑新业务快速上线。此外，还提供了 QPS、RT、接口调用量、错误数等实时监控功能，帮助研发人员快速定位问题，提升诊断效率，让企业聚焦于业务本身。

**业务稳定：**基于 阿里云应用引擎 Serverless（SAE）的定时弹性能力和基于监控指标弹性（CPU/Memory 等），无须容量规划，秒级弹性，便可轻松应对流量暴增，保障 SLA。

# 7

## 云原生架构未来发展趋势

### 1 容器技术发展趋势



**在云端，Serverless 技术逐渐融入主流**  
 极致弹性，免运维，强安全，高效开发  
 - 与云深度融合，复杂性下沉，无限弹性  
 - 新的应用开发、交付范形开始形成

**Serverless**  
 Serverless 容器  
 函数计算

**分布式云**  
 从多云/混合云，到  
 分布式云

**动态、混合、分布式的云环境将成为新常态**  
 统一技术栈，统一应用界面，统一管理界面  
 - 公共云服务能力将延伸到边缘计算和IDC  
 - 云原生架构推进无边界云计算，促成云边端应用  
 一体协调

**无处不在的计算催生新一代容器实现**  
 针对计算场景优化、安全、轻重、高效  
 - 基于 MicroVM 的安全容器  
 - 基于 WebAssembly 的可移植、轻量容器  
 - OS 虚拟化创新，如 cgroup v2 提升隔离性

**新一代的计算单元**

**云原生操作系统**

**云原生操作系统开始浮现**  
 定义开放标准，向下封装资源，向上支撑应用  
 - 计算、存储、网络、安全的云原生进化  
 - 多种工作负载、海量计算任务，多种异构算力的  
 高效调度和编排  
 - 标准化、自动化、可移植、安全可控的应用交付、  
 管理体系

#### 1 趋势一：无处不在的计算催生新一代容器实现

随着互联网的发展到万物智联，5G、AIoT 等新技术的涌现，随处可见的计算需求已经成为现实。针对不同计算场景，容器运行时会有不同需求。KataContainer、Firecracker、gVisor、Unikernel 等新的容器运行时技术层出不穷，分别解决安全隔离性、执行效率和通用性三个不同维度的要求。OCI ( Open Container Initiative ) 标准的出现，使不同技术采用一致的方式进行容器生命周期管理，进一步促进了容器引擎技术的持续创新。其中，我们可以预见以下几个细分方向的未来趋势：

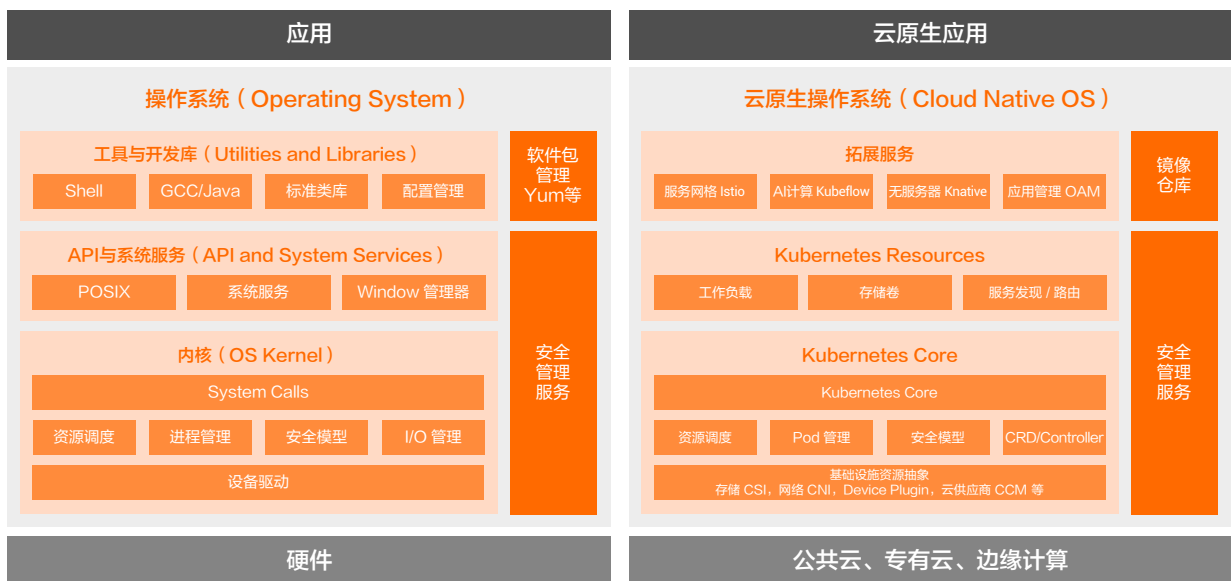
基于 MicroVM 的安全容器占比将逐渐增加，提供更强的安全隔离能力。虚拟化和容器技术的融合，已成为未来重要趋势。在公共云上，阿里云容器服务已经提供了对基于 KataContainer 的阿里云的袋鼠容器引擎支持，可以运行不可信的工作负载，实现安全的多租隔离。

基于软硬一体设计的机密计算容器开始展露头角。比如阿里云安全、系统软件、容器服务团队以及蚂蚁金服可信原生团队共同推出了面向机密计算场景的开源容器运行时技术栈 `inclavare-containers`，支持基于 Intel SGX 机密计算技术的机密容器实现，如蚂蚁金服的 Occlum、开源社区的 Graphene 等 Library OS。它极大降低了机密计算的技术门槛，简化了可信应用的开发、交付和管理。

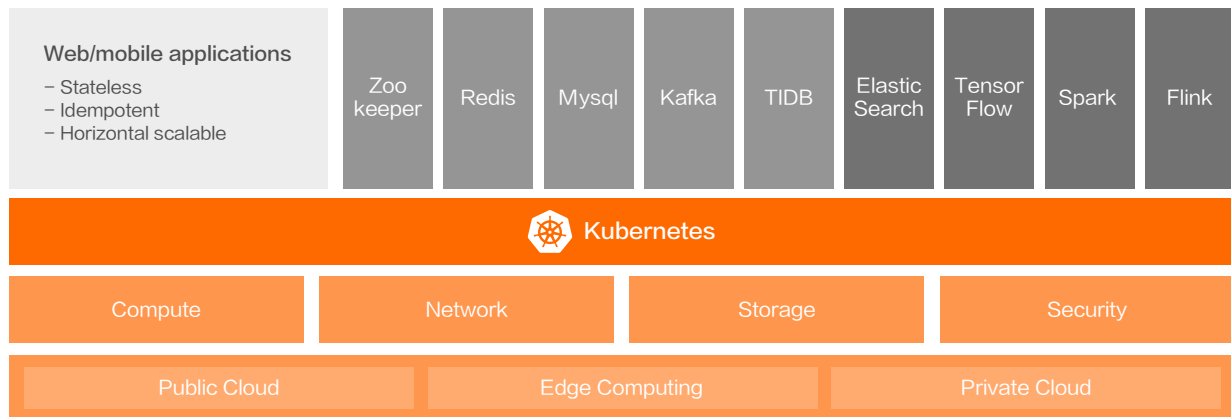
WebAssembly 作为新一代可移植、轻量化、应用虚拟机，在 IoT，边缘计算，区块链等场景会有广泛的应用前景。WASM/WASI 将会成为一个跨平台容器实现技术。近期 Solo.io 推出的 WebAssembly Hub 就将 WASM 应用通过 OCI 镜像标准进行统一管理和分发，从而更好地应用在 Istio 服务网格生态中。

## 2 趋势二：云原生操作系统开始浮现

Kubernetes 已经成为云时代的操作系统。对比 Linux 与 Kubernetes 的概念模型，他们都是定义了开放的、标准化的访问接口；向下封装资源，向上支撑应用。



它们都提供了对底层计算、存储、网络、异构计算设备的资源抽象和安全访问模型，可以根据应用需求进行资源调度和编排。Linux 的计算调度单元是进程，调度范围限制在一台计算节点。而 Kubernetes 的调度单位是 Pod，可以在分布式集群中进行资源调度，甚至跨越不同的云环境。



从无状态应用，到企业核心应用，到数据智能应用

过往 Kubernetes 上主要运行着无状态的 Web 应用。随着技术演进和社区发展，越来越多有状态应用和大数据 /AI 应用负载逐渐迁移到 Kubernetes 上。Flink、Spark 等开源社区以及 Cloudera、Databricks 等商业公司都开始加大对 Kubernetes 的支持力度。

统一技术栈提升资源利用率：多种计算负载在 Kubernetes 集群统一调度，可以有效提升资源利用率。Gartner 预测“未来 3 年，70% AI 任务运行在容器和 Serverless 上。” AI 模型训练和大数据计算类工作负载需要 Kubernetes 提供更低调度延迟、更大并发调度吞吐和更高异构资源利用率。阿里云在和 Kubernetes 上游社区共同合作，在 Scheduler V2 framework 上，通过扩展机制增强 Kubernetes 调度器的规模、效率和能力，具备更好的兼容性，可以更好的支撑多种工作负载的统一调度。

统一技能栈降低人力成本：Kubernetes 可以在 IDC、云端、边缘等不同场景进行统一部署和交付。云原生提倡的 DevOps 文化和工具集可以有效提升技术迭代速度并降低人力成本。

加速数据服务的云原生：由于计算存储分离具备巨大的灵活性和成本优势，数据服务的云原生也逐渐成为趋势。容器和 Serverless 的弹性可以简化对计算任务的容量规划。结合分布式缓存加速（比如 Alluxio 或阿里云 Jindofs）和调度优化，大大提升数据计算类和 AI 任务的计算效率。

### 3 趋势三：Serverless 容器技术逐渐成为市场主流

Serverless 和容器技术也开始融合得到了快速的发展。通过 Serverless 容器，一方面根本性解决 Kubernetes 自身复杂性问题，让用户无需受困于 Kubernetes 集群容量规划、安全维护、故障诊断等运维工作；一方面进一步释放云计算能力，将安全、可用性、可伸缩性等需求下沉到基础设施实现。

### 4 趋势四：动态、混合、分布式的云环境将成为新常态

上云已是大势所趋，但对于企业客户而言，有些业务出于对数据主权、安全隐私的考量，会采用混合云架构。一

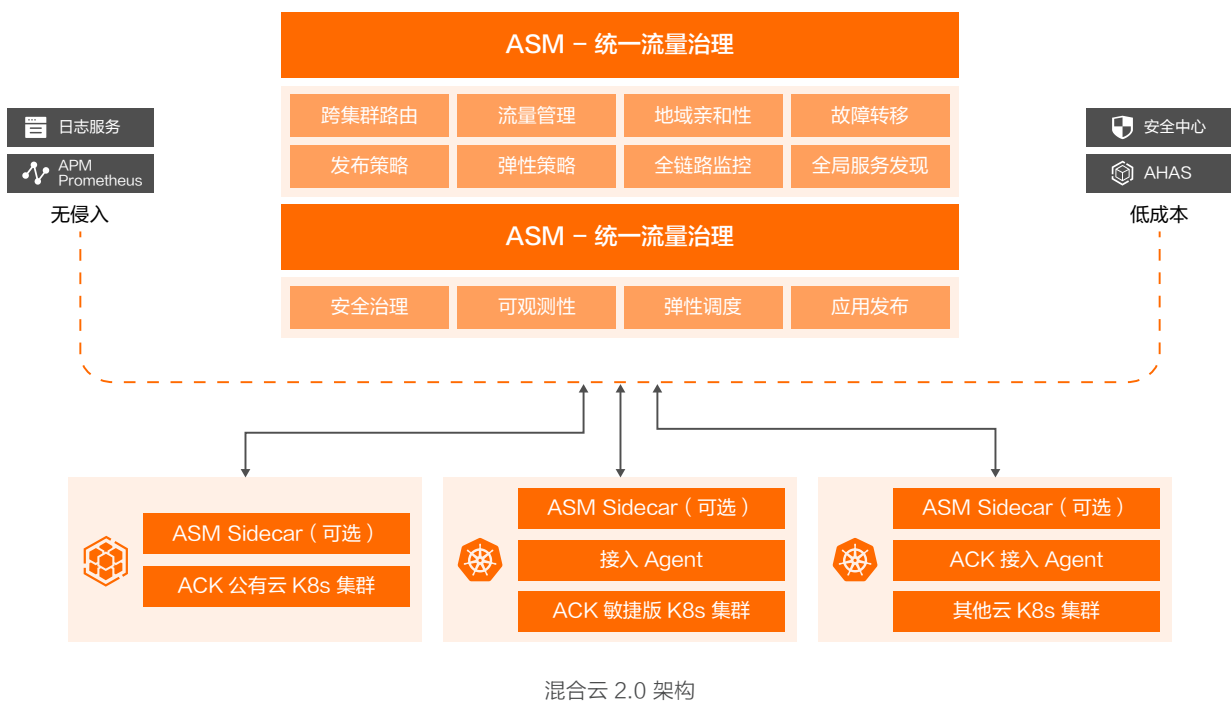
些企业为了满足安全合规、成本优化、提升地域覆盖性和避免云厂商锁定等需求，会选择多个云厂商。混合云 / 多云架构已成为企业上云新常态。Gartner 指出，" 到 2021，超过 75% 的大中型组织将采用多云或者混合 IT 战略。"

此外边缘计算将成为企业云战略的重要组成部分，为应用提供更低网络延迟，更高网络带宽和更低网络成本。我们需要有能力将智能决策、实时处理能力从云延展到边缘和 IoT 设备端。

随着云平台已经成为企业数字化转型的创新平台，一个变化随之产生——云正在靠近它们。在分布式云中，公有云的服务能力可以位于不同的物理位置，而公共云平台提供者会负责服务的运维、治理、更新和演变。

然而不同环境的基础设施能力、安全架构的差异会造成企业 IT 架构和运维体系的割裂，加大云战略实施复杂性，增加运维成本。在云原生时代，以 Kubernetes 为代表的云原生技术屏蔽了基础设施差异性，推动了以应用为中心的混合云 / 分布式云架构的到来。从而更好地支持不同环境下应用统一生命周期管理和统一资源调度。Kubernetes 已经成为企业多云管理的事实基础，

阿里云容器服务 ACK 去年 9 月份发布了混合云 2.0 架构，提供了完备的混合云 Kubernetes 管理能力。



ACK 提供了统一集群管理能力，除了可以管理阿里云 Kubernetes 集群之外，还可以纳管用户在 IDC 的自有 Kubernetes 集群和其他云的 Kubernetes 集群。利用统一的控制平面实现多个集群的统一的的安全治理、可观测性、应用管理、备份恢复等能力。比如利用日志服务、托管 Prometheus 服务，可以无侵入的方式帮助用户对线上、线下集群有一个统一的可观测性大盘。利用云安全中心，AHAS 可以帮助客户在混合云的整体架构中发现并解决安全和稳定性风险。

托管服务网格 ASM 提供统一的服务治理能力，结合阿里云云企业网 CEN、智能接入网关 SAG 提供的多地域、混合云网络能力，可以实现服务就近访问，故障转移，灰度发布等功能。

ACK 也提供了统一的应用交付能力，通过 GitOps 方式可以将应用安全、一致、稳定地发布在多个不同的云环境中。配合网格 ASM 提供的流量管理能力，可以支持云容灾、异地多活等应用场景，提升业务连续性。

## 2 基于云原生的新一代应用编程界面



Kubernetes 已经成为了云原生的操作系统，而容器成为了操作系统调度的基本单元，同时定义了应用交付的标准。但是对于应用开发者来说，这些还远没有深入到应用的架构，改变应用的编程界面。但是这种变革已经在悄然发生了，而且有不断加速之势。

- **Sidecar 架构彻底改变了应用的运维架构。**由于 Sidecar 架构支持在运行时隔离应用容器与其他容器，因此原本在虚拟机时代和业务进程部署在一起的大量运维及管控工具，都被剥离到独立的容器里进行统一管理。对于应用来说，仅仅是按需声明使用运维能力，能力的实现成为云平台的职责。
- **应用生命周期全面托管。**在容器技术基础上，应用进一步描述清晰自身状态（例如通过 Liveness Probe），描述自身的弹性指标以及通过 Service Mesh 和 Serverless 技术将流量托管给云平台。云平台将能够全面管理应用的生命周期，包括服务的上下线、版本升级、完善的流量调配、容量管理等，并保障业务稳定性。
- **用声明式配置方式使用云服务。**云原生应用的核心特点之一就是大量依赖云服务（包括数据库、缓存、消息等）构建，以实现快速交付。而这些服务的配置实际上是应用自身的资产，为了能够让应用无缝地运行在混合云的场景，应用逐渐开始以基础设施即代码的方式使用云服务。正因如此，相关的产品如 Terraform、Pulumi 越来越受到应用开发者的拥抱。
- **语言无关的分布式编程框架成为一种服务。**为了解决分布式带来的技术挑战，传统中间件需要在客户端 SDK 编写大量的逻辑管理分布式状态。今天，还是基于 Sidecar 架构，我们看到很多项目在把这些内容下沉到 Sidecar 中，并通过语言无关的 API（基于 gRPC/HTTP）提供给应用。这一变化进一步简化应用代码的逻辑和应用研发的职责，例如配置的绑定，身份的认证和鉴权都可以在 Sidecar 被统一处理。在这个方面 dapr.io 和 cloudstate.io 是先行者。

综上，包括生命周期管理、运维管理、配置范围和扩展和管理、以及语言无关的编程框架，一起构成了崭新的应用与云之间的编程界面。这一变革的核心逻辑还是把应用中和业务无关的逻辑和职责，剥离到云服务，并在这个过程中形成标准，让应用开发者能够在专有云、公有云、或者混合云的场景中，都能有一致的研发运维体验。



### 3 Serverless 发展趋势



近年来，Serverless 一直在高速发展，呈现出越来越大的影响力。在这样的趋势下，主流云服务商也在不断丰富云产品体系，提供更便捷的开发工具，更高效的应用交付流水线，更完善的可观测性，更丰富的产品间集成。

#### 1 趋势一：Serverless 将无处不在

任何足够复杂的技术方案都可能被实现为全托管、Serverless 化的后端服务。不只是云产品，也包括来自合作伙伴和三方的服务，云及其生态的能力将通过 API + Serverless 来体现。事实上，对于任何以 API 作为功能透出方式的平台型产品或组织，例如钉钉、微信、滴滴等等，Serverless 都将是其平台战略中最重要的部分。

#### 2 趋势二：Serverless 将通过事件驱动的方式连接云及其生态中的一切

通过事件驱动和云服务连接，Serverless 能力也会扩展到整个云的生态。无论是用户自己的应用还是合作伙伴的服务，无论是 on-premise 环境还是公有云，所有的事件都能以 Serverless 的方式处理。云服务及其生态将更紧密地连接在一些，成为用户构建弹性、高可用应用的基石。

#### 3 趋势三：Serverless 计算将持续提高计算密度，实现最佳的性能功耗比和性能价格比

虚拟机和容器是两种取向不同的虚拟化技术，前者安全性强、开销大，后者则相反。Serverless 计算平台一方面要求兼得最高的安全性和最小的资源开销，另一方面要保持对原有程序执行方式的兼容，比如支持任意二进制文件，这使得适用于特定语言 VM 的方案不可行。以 AWS FireCracker 为例，其通过对设备模型的裁剪和 kernel 加载流程的优化，实现百毫秒的启动速度和极小的内存开销，一台裸金属实例可以支持数以千计的实例运行。结合应用负载感知的资源调度算法，虚拟化技术有望在保持稳定性能的前提下，将超卖率提升一个数量级。

当 Serverless 计算的规模与影响力变得越来越大，在应用框架、语言、硬件等层面上根据 Serverless 负载特点进行端对端优化就变得非常有意义。新的 Java 虚拟机技术大幅提高了 Java 应用启动速度，非易失性内存帮助实例更快被唤醒，CPU 硬件与操作系统协作对高密环境下性能扰动实现精细隔离，所有新技术正在创造崭新的计算环境。

实现最佳性能功耗比和性能价格比的另一个重要方向是支持异构硬件。长期以来，x86 处理器的性能越来越难以提升。而在 AI 等对算力要求极高的场景，GPU、FPGA、TPU (Tensor Processing Units) 等架构处理器的计算效率更具优势。随着异构硬件虚拟化、资源池化、异构资源调度和应用框架支持的成熟，异构硬件的算力也能通过 Serverless 的方式释放，大幅降低用户使用门槛。

