浅谈云原生架构的 7 个原则

原创

阿里云云栖号 于 2021-07-19 16:01:22 发布 284 收藏 3
分类专栏： 云栖号技术分享 文章标签： 云原生 数据

简介： 作为一种架构模式，云原生架构通过若干原则来对应用架构进行核心控制。这些原则可以帮助技术主管和架构师在进行技术选型时更加高效、准确，下面将展开具体介绍。

服务化原则

在软件开发过程中，当代码数量与开发团队规模都扩张到一定程度后，就需要重构应用，通过模块化与组件化的手段分离关注点，降低应用的复杂度，提升软件的开发效率，降低维护成本。

如图 1，随着业务的不断发展，单体应用能够承载的容量将逐渐到达上限，即使通过应用改造来突破垂直扩展（Scale Up）的瓶颈，并将其转化为支撑水平扩展（Scale Out）的能力，在全局并发访问的情况下，也依然会面临数据计算复杂度和存储容量的问题。因此，需要将单体应用进一步拆分，按业务边界重新划分成分布式应用，使应用与应用之间不再直接共享数据，而是通过约定好的契约进行通信，以提高扩展性。

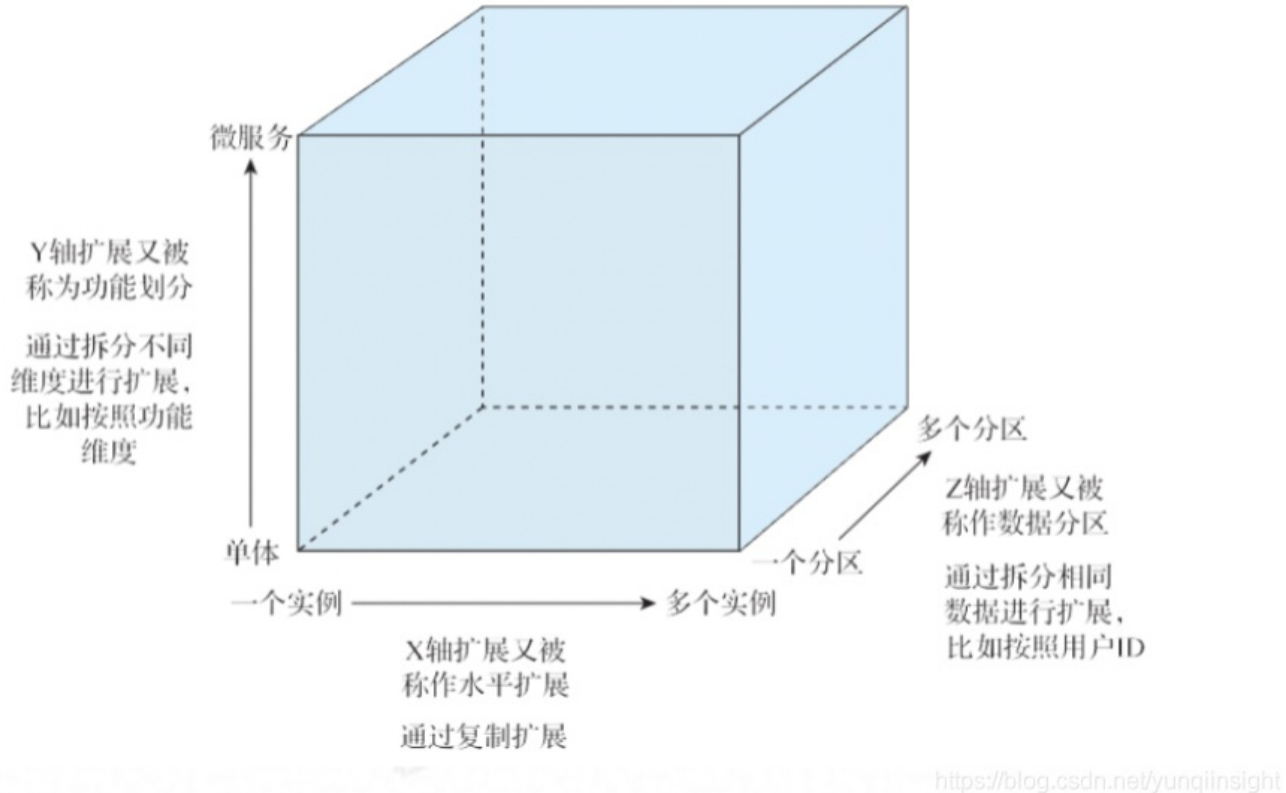

图 1 应用服务化扩展

服务化设计原则是指通过服务化架构拆分不同生命周期的业务单元，实现业务单元的独立迭代，从而加快整体的迭代速度，保证迭代的稳定性。同时，服务化架构采用的是面向接口编程方式，增加了软件的复用程度，增强了水平扩展的能力。服务化设计原则还强调在架构层面抽象化业务模块之间的关系，从而帮助业务模块实现基于服务流量（而非网络流量）的策略控制和治理，而无须关注这些服务是基于何种编程语言开发的。

有关服务化设计原则的实践在业界已有很多成功案例。其中影响最广、最为业界称道的是 Netflix 在生产系统上所进行的大规模微服务化实践。通过这次实践，Netflix 在全球不仅承接了多达 1.67 亿订阅用户以及全球互联网带宽容量 15% 以上的流量，而且在开源领域贡献了 Eureka、Zuul、Hystrix 等出色的微服务组件。

不仅海外公司正在不断进行服务化实践，国内公司对服务化也有很高的认知。随着近几年互联网化的发展，无论是新锐互联网公司，还是传统大型企业，在服务化实践上都有很好的实践和成功案例。阿里巴巴的服务化实践发端于 2008 年的五彩石项目，历经 10 年的发展，稳定支撑历年大促活动。以 2019 年“双 11”当天数据为例，阿里巴巴的分布式系统创单峰值为每秒 54.4 万笔，实时计算处理为每秒 25.5 亿笔。阿里巴巴在服务化领域的实践，已通过 Apache Dubbo、Nacos、Sentinel、Seata、Chaos Blade 等开源项目分享给业界， 同时，这些组件与 Spring Cloud的集成 Spring Cloud Alibaba 已成为 Spring Cloud Netflix 的继任者。

虽然随着云原生浪潮的兴起，服务化原则不断演进、落地于实际业务，但企业在实际落地过程中也会遇到不少的挑战。比如，与自建数据中心相比，公有云下的服务化可能存在巨大的资源池，使得机器错误率显著提高；按需付费增加了扩缩容的操作频度；新的环境要求应用启动更快、应用与应用之间无强依赖关系、应用能够在不同规格的节点之间随意调度等诸多需要考虑的实际问题。但可以预见的是，这些问题会随着云原生架构的不断演进而得到逐一解决。

弹性原则

弹性原则是指系统部署规模可以随着业务量变化自动调整大小，而无须根据事先的容量规划准备固定的硬件和软件资源。优秀的弹性能力不仅能够改变企业的 IT 成本模式，使得企业不用再考虑额外的软硬件资源成本支出（闲置成本），也能更好地支持业务规模的爆发式扩张，不再因为软硬件资源储备不足而留下遗憾。

在云原生时代，企业构建 IT 系统的门槛大幅降低，这极大地提升了企业将业务规划落地为产品与服务的效率。这一点在移动互联网和游戏行业中显得尤为突出。一款应用成为爆款后，其用户数量呈现指数级增长的案例不在少数。而业务呈指数级增长会对企业 IT 系统的性能带来巨大考验。面对这样的挑战，在传统架构中，通常是开发人员、运维人员疲于调优系统性能，但是，即使他们使出浑身解数，也未必能够完全解决系统的瓶颈问题， 最终因系统无法应对不断涌入的海量用户而造成应用瘫痪。

除了面临业务呈指数级增长的考验之外，业务的峰值特征将是另一个重要的挑战。比如，电影票订票系统下午时段的流量远超凌晨时段，而周末的流量相比工作日甚至会翻好几倍；还有外卖订餐系统，在午餐和晚餐前后往往会出现订单峰值时段。在传统架构中，为了应对这类具有明显峰值特征的场景，企业需要为峰值时段的流量提前准备大量的计算、存储及网络资源并为这些资源付费，而这些资源在大部分时间内却处于闲置状态。

因此，在云原生时代，企业在构建 IT系统时，应该尽早考虑让应用架构具备弹性能力，以便在快速发展的业务规模面前灵活应对各种场景需求，充分利用云原生技术及成本优势。

要想构建弹性的系统架构，需要遵循如下四个基本原则。

1. 按功能切割应用

一个大型的复杂系统可能由成百上千个服务组成，架构师在设计架构时，需要遵循的原则是：将相关的逻辑放到一起，不相关的逻辑拆解到独立的服务中，各服务之间通过标准的服务发现（Service Discovery）找到对方，并使用标准的接口进行通信。各服务之间松耦合，这使得每一个服务能够各自独立地完成弹性伸缩，从而避免服务上下游关联故障的发生。

2. 支持水平切分

按功能切割应用并没有完全解决弹性的问题。一个应用被拆解为众多服务后，随着用户流量的增长，单个服务最终也会遇到系统瓶颈。因此在设计上，每个服务都需要具备可水平切分的能力，以便将服务切分为不同的逻辑单元，由每个单元处理一部分用户流量，从而使服务自身具备良好的扩展能力。这其中最大的挑战在于数据库系统，因为数据库系统自身是有状态的，所以合理地切分数据并提供正确的事务机制将是一个非常复杂的工程。不过，在云原生时代，云平台所提供的云原生数据库服务可以解决大部分复杂的分布式系统问题，因此，如果企业是通过云平台提供的能力来构建弹性系统，自然就会拥有数据库系统的弹性能力。

3. 自动化部署

系统突发流量通常无法预计，因此常用的解决方案是，通过人工扩容系统的方式，使系统具备支持更大规模用户访问的能力。在完成架构拆分之后，弹性系统还需要具备自动化部署能力，以便根据既定的规则或者外部流量突发信号触发系统的自动化扩容功能，满足系统对于缩短突发流量影响时长的及时性要求，同时在峰值时段结束后自动缩容系统，降低系统运行的资源占用成本。

4. 支持服务降级

弹性系统需要提前设计异常应对方案，比如，对服务进行分级治理，在弹性机制失效、弹性资源不足或者流量峰值超出预期等异常情况下，系统架构需要具备服务降级的能力，通过降低部分非关键服务的质量，或者关闭部分增强功能来让出资源，并扩容重要功能对应的服务容量，以确保产品的主要功能不受影响。

国内外已有很多成功构建大规模弹性系统的实践案例，其中最具代表性的是阿里巴巴一年一度的“双11”大促活动。为了应对相较于平时上百倍的流量峰值，阿里巴巴每年从阿里云采买弹性资源部署自己的应用，并在“双11”活动之后释放这一批资源，按需付费，从而大幅降低大促活动的资源成本。另一个例子是新浪微博的弹性架构，在社会热点事件发生时，新浪微博通过弹性系统将应用容器扩容到阿里云，以应对热点事件导致的大量搜索和转发请求。系统通过分钟级的按需扩容响应能力，大幅降低了热搜所产生的资源成本。

随着云原生技术的发展，FaaS、Serverless 等技术生态逐步成熟，构建大规模弹性系统的难度逐步降低。当企业以 FaaS、Serverless 等技术理念作为系统架构的设计原则时，系统就具备了弹性伸缩的能力，企业也就无须额外为“维护弹性系统自身”付出成本。

可观测原则

与监控、业务探活、APM（Application Performance Management，应用性能管理）等系统提供的被动能力不同，可观测性更强调主动性，在云计算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 点击所产生的多次服务调用耗时、返回值和参数都清晰可见，甚至可以下钻到每次第三方软件调用、SQL 请求、节点拓扑、网络响应等信息中。运维、开发和业务人员通过这样的观测能力可以实时掌握软件的运行情况，并获得前所未有的关联分析能力，以便不断优化业务的健康度和用户体验。

随着云计算的全面发展，企业的应用架构发生了显著变化，正逐步从传统的单体应用向微服务过渡。在微服务架构中，各服务之间松耦合的设计方式使得版本迭代更快、周期更短；基础设施层中的 Kubernetes 等已经成为容器的默认平台；服务可以通过流水线实现持续集成与部署。这些变化可将服务的变更风险降到最低，提升了研发的效率。

在微服务架构中，系统的故障点可能出现在任何地方，因此我们需要针对可观测性进行体系化设计，以降低 MTTR（故障平均修复时间）。

要想构建可观测性体系，需要遵循如下三个基本原则。

1. 数据的全面采集

指标（Metric）、链路跟踪（Tracing）和日志（Logging）这三类数据是构建一个完整的可观测性系统的“三大支柱”。而系统的可观测性就是需要完整地采集、分析和展示这三类数据。

（1）指标

指标是指在多个连续的时间周期里用于度量的 KPI 数值。一般情况下，指标会按软件架构进行分层，分为系统资源指标（如 CPU 使用率、磁盘使用率和网络带宽情况等）、应用指标（如出错率、服务等级协议 SLA、服务满意度 APDEX、平均延时等）、业务指标（如用户会话数、订单数量和营业额等）。

（2）链路跟踪

链路跟踪是指通过 TraceId 的唯一标识来记录并还原发生一次分布式调用的完整过程，贯穿数据从浏览器或移动端经过服务器处理，到执行 SQL 或发起远程调用的整个过程。

（3）日志

日志通常用来记录应用运行的执行过程、代码调试、错误异常等信息，如 Nginx 日志可以记录远端 IP、发生请求时间、数据大小等信息。日志数据需要集中化存储，并具备可检索的能力。

2. 数据的关联分析

让各数据之间产生更多的关联，这一点对于一个可观测性系统而言尤为重要。出现故障时，有效的关联分析可以实现对故障的快速定界与定位，从而提升故障处理效率，减少不必要的损失。一般情况下，我们会将应用的服务器地址、服务接口等信息作为附加属性，与指标、调用链、日志等信息绑定，并且赋予可观测系统一定的定制能力，以便灵活满足更加复杂的运维场景需求。

3. 统一监控视图与展现

多种形式、多个维度的监控视图可以帮助运维和开发人员快速发现系统瓶颈，消除系统隐患。监控数据的呈现形式应该不仅仅是指标趋势图表、柱状图等，还需要结合复杂的实际应用场景需要，让视图具备下钻分析和定制能力，以满足运维监控、版本发布管理、故障排除等多场景需求。

随着云原生技术的发展，基于异构微服务架构的场景会越来越多、越来越复杂，而可观测性是一切自动化能力构建的基础。只有实现全面的可观测性，才能真正提升系统的稳定性、降低 MTTR。因此，如何构建系统资源、容器、网络、应用、业务的全栈可观测体系，是每个企业都需要思考的问题。

韧性原则

韧性是指当软件所依赖的软硬件组件出现异常时，软件所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 或网卡带宽耗尽）、业务流量超出软件设计承受能力、影响机房正常工作的故障或灾难、所依赖软件发生故障等可能造成业务不可用的潜在影响因素。

业务上线之后，在运行期的大部分时间里，可能还会遇到各种不确定性输入和不稳定依赖的情况。当这些非正常场景出现时，业务需要尽可能地保证服务质量，满足当前以联网服务为代表的“永远在线”的要求。因此，韧性能力的核心设计理念是面向失败设计，即考虑如何在各种依赖不正常的情况下，减小异常对系统及服务质量的影响并尽快恢复正常。

韧性原则的实践与常见架构主要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、多 AZ（Availability Zone，可用区）的高可用、单元化、跨区域（Region）容灾、异地多活容灾等。

下面结合具体案例详细说明如何在大型系统中进行韧性设计。“双 11”对于阿里巴巴来说是一场不能输的战役，因此其系统的设计在策略上需要严格遵循韧性原则。例如，在统一接入层通过流量清洗实现安全策略，防御黑产攻击；通过精细化限流策略确保峰值流量稳定，从而保障后端工作正常进行。为了提升全局的高可用能力，阿里巴巴通过单元化机制实现了跨区域多活容灾，通过同城容灾机制实现同城双活容灾，从而最大程度提升 IDC（Internet Data Center，互联网数据中心）的服务质量。在同一 IDC 内通过微服务和容器技术实现业务的无状态迁移；通过多副本部署提高高可用能力；通过消息完成微服务间的异步解耦以降低服务的依赖性，同时提升系统吞吐量。从每个应用的角度，做好自身依赖梳理，设置降级开关，并通过故障演练不断强化系统健壮性，保证阿里巴巴“双11”大促活动正常稳定进行。

随着数字化进程的加快，越来越多的数字化业务成为整个社会经济正常运转的基础设施，但随着支撑这些数字化业务的系统越来越复杂，依赖服务质量不确定的风险正变得越来越高，因此系统必须进行充分的韧性设计，以便更好地应对各种不确定性。尤其是在涉及核心行业的核心业务链路（如金融的支付链路、电商的交易链路）、业务流量入口、依赖复杂链路时，韧性设计至关重要。

所有过程自动化原则

技术是把“双刃剑”，容器、微服务、DevOps 以及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，也提高了软件技术栈的复杂度，加大了组件规模，从而不可避免地导致了软件交付的复杂性。如果控制不当，应用就会无法体会到云原生技术的优势。通过 IaC、GitOps、OAM、Operator 和大量自动化交付工具在 CI/CD（持续集成 /持续交付）流水线中的实践，企业可以标准化企业内部的软件交付过程，也可以在标准化的基础上实现自动化，即通过配置数据自描述和面向终态的交付过程，实现整个软件交付和运维的自动化。

要想实现大规模的自动化，需要遵循如下四个基本原则。

1. 标准化

实施自动化，首先要通过容器化、IaC、OAM 等手段，标准化业务运行的基础设施，并进一步标准化对应用的定义乃至交付的流程。只有实现了标准化，才能解除业务对特定的人员和平台的依赖，实现业务统一和大规模的自动化操作。

2. 面向终态

面向终态是指声明式地描述基础设施和应用的期望配置，持续关注应用的实际运行状态，使系统自身反复地变更和调整直至趋近终态的一种思想。面向终态的原则强调应该避 免直接通过工单系统、工作流系统组装一系列过程式的命令来变更应用，而是通过设置终态，让系统自己决策如何执行变更。

3. 关注点分离

自动化最终所能达到的效果不只取决于工具和系统的能力，更取决于为系统设置目标的人，因此要确保找到正确的目标设置人。在描述系统终态时，要将应用研发、应用运维、基础设施运维这几种主要角色所关注的配置分离开来，各个角色只需要设置自己所关注和擅长的系统配置，以便确保设定的系统终态是合理的。

4. 面向失败设计

要想实现全部过程自动化，一定要保证自动化的过程是可控的，对系统的影响是可预期的。我们不能期望自动化系统不犯错误，但可以保证即使是在出现异常的情况下，错误的影响范围也是可控的、可接受的。因此，自动化系统在执行变更时，同样需要遵循人工变更的最佳实践，保证变更是可灰度执行的、执行结果是可观测的、变更是可快速回滚的、变更影响是可追溯的。

业务实例的故障自愈是一个典型的过程自动化场景。业务迁移到云上后，云平台虽然通过各种技术手段大幅降低了服务器出故障的概率，但是却并不能消除业务本身的软件故障。软件故障既包括应用软件自身的缺陷导致的崩溃、资源不足导致的内存溢出（OOM）和负载过高导致的夯死等异常问题，也包括内核、守护进程（daemon 进程）等系统软件的问题，更包括混部的其他应用或作业的干扰问题。随着业务规模的增加，软件出现故障的风险正变得越来越高。传统的运维故障处理方式需要运维人员的介入，执行诸如重启或者腾挪之类的修复操作，但在大规模场景下，运维人员往往疲于应对各种故障，甚至需要连夜加班进行操作，服务质量很难保证，不管是客户，还是开发、运维人员，都无法满意。

为了使故障能够实现自动化修复，云原生应用要求开发人员通过标准的声明式配置，描述应用健康的探测方法和应用的启动方法、应用启动后需要挂载和注册的服务发现以及配置管理数据库（Configuration Management Data Base，CMDB）信息。通过这些标准的配置，云平台可以反复探测应用，并在故障发生时执行自动化修复操作。另外，为了防止故障探测本身可能存在的误报问题，应用的运维人员还可以根据自身容量设置服务不可用实例的比例，让云平台能够在进行自动化故障恢复的同时保证业务可用性。实例故障自愈的实现，不仅把开发人员和运维人员从烦琐的运维操作中解放了出来，而且可以及时处理各种故障，保证业务的连续性和服务的高可用性。

零信任原则

基于边界模型的传统安全架构设计，是在可信和不可信的资源之间架设一道墙，例如，公司内网是可信的，而因特网则是不可信的。在这种安全架构设计模式下，一旦入侵者渗透到边界内，就能够随意访问边界内的资源了。而云原生架构的应用、员工远程办公模式的普及以及用手机等移动设备处理工作的现状，已经完全打破了传统安全架构下的物理边界。员工在家办公也可以实现与合作方共享数据，因为应用和数据被托管到了云上。

如今，边界不再是由组织的物理位置来定义，而是已经扩展到了需要访问组织资源和服务的所有地方，传统的防火墙和 VPN 已经无法可靠且灵活地应对这种新边界。因此，我们需要一种全新的安全架构，来灵活适应云原生和移动时代环境的特性，不论员工在哪里办公，设备在哪里接入，应用部署在哪里，数据的安全性都能够得到有效保护。如果要实现这种新的安全架构，就要依托零信任模型。

传统安全架构认为防火墙内的一切都是安全的，而零信任模型假设防火墙边界已经被攻破，且每个请求都来自于不可信网络，因此每个请求都需要经过验证。简单来说，“永不信任，永远验证”。在零信任模型下，每个请求都要经过强认证，并基于安全策略得到验证授权。与请求相关的用户身份、设备身份、应用身份等，都会作为核心信息来判断请求是否安全。

如果我们围绕边界来讨论安全架构，那么传统安全架构的边界是物理网络，而零信任安全架构的边界则是身份，这个身份包括人的身份、设备的身份、应用的身份等。要想实现零信任安全架构，需要遵循如下三个基本原则。

1. 显式验证

对每个访问请求都进行认证和授权。认证和授权需要基于用户身份、位置、设备信息、服务和工作负载信息以及数据分级和异常检测等信息来进行。例如，对于企业内部应用之间的通信，不能简单地判定来源 IP是内部 IP 就直接授权访问，而是应该判断来源应用的身份和设备等信息，再结合当前的策略授权。

2. 最少权限
**对于每个请求，只授予其在当下必需的权限，且权限策略应该能够基于当前请求上下文自适应。例如，HR 部门的员工应该拥有访问 HR相关应用的权限，但不应该拥有访问财务部门应用的权限。

3. 假设被攻破

假设物理边界被攻破，则需要严格控制安全爆炸半径，将一个整体的网络切割成对用户、设备、应用感知的多个部分。对所有的会话加密，使用数据分析技术保证对安全状态的可见性。

从传统安全架构向零信任架构演进，会对软件架构产生深刻的影响，具体体现在如下三个方面。

第一，不能基于 IP 配置安全策略。在云原生架构下，不能假设 IP 与服务或应用是绑定的，这是由于自动弹性等技术的应用使得 IP 随时可能发生变化，因此不能以 IP 代表应用的身份并在此基础上建立安全策略。

第二，身份应该成为基础设施。授权各服务之间的通信以及人访问服务的前提是已经明确知道访问者的身份。在企业中，人的身份管理通常是安全基础设施的一部分，但应用的身份也需要管理。

第三，标准的发布流水线。在企业中，研发的工作通常是分布式的，包括代码的版本管理、构建、测试以及上线的过程，都是比较独立的。这种分散的模式将会导致在实际生产环境中运行的服务的安全性得不到有效保证。如果可以标准化代码的版本管理、构建以及上线的流程，那么应用发布的安全性就能够得到集中增强。

总体来说，整个零信任模型的建设包括身份、设备、应用、基础设施、网络、数据等几个部分。零信任的实现是一个循序渐进的过程，例如，当组织内部传输的所有流量都没有加密的时候，第一步应该先保证访问者访问应用的流量是加密的，然后再逐步实现所有流量的加密。如果采用云原生架构，就可以直接使用云平台提供的安全基础设施和服务，以便帮助企业快速实现零信任架构。

架构持续演进原则

如今，技术与业务的发展速度都非常快，在工程实践中很少有从一开始就能够被明确定义并适用于整个软件生命周期的架构模式，而是需要在一定范围内不断重构，以适应变 化的技术和业务需求。同理，云原生架构本身也应该且必须具备持续演进的能力，而不是一个封闭式的、被设计后一成不变的架构。因此在设计时除了要考虑增量迭代、合理化目标选取等因素之外，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规范以及业务自身的特点，特别是在业务高速迭代的情况下，更应该重点考虑如何保证架构演进与业务发展之间的平衡。

1. 演进式架构的特点和价值

演进式架构是指在软件开发的初始阶段，就通过具有可扩展性和松耦合的设计，让后续可能发生的变更更加容易、升级性重构的成本更低，并且能够发生在开发实践、发布实践和整体敏捷度等软件生命周期中的任何阶段。

演进式架构之所以在工业实践中具有重要意义，其根本原因在于，在现代软件工程领域达成的共识中，变更都是很难预测的，其改造的成本也极其高昂。演进式架构并不能避免重构，但是它强调了架构的可演进性，即当整个架构因为技术、组织或者外部环境的变化需要向前演进时，项目整体依然能够遵循强边界上下文的原则，确保领域驱动设计中描述的逻辑划分变成物理上的隔离。演进式架构通过标准化且具有高可扩展性的基础设施体系，大量采纳标准化应用模型与模块化运维能力等先进的云原生应用架构实践，实现了整个系统架构在物理上的模块化、可复用性与职责分离。在演进式架构中，系统的每个服务在结构层面与其他服务都是解耦的，替换服务就像替换乐高积木一样方便。

2. 演进式架构的应用

在现代软件工程实践中，演进式架构在系统的不同层面有着不同的实践与体现。

在面向业务研发的应用架构中，演进式架构通常与微服务设计密不可分。例如，在阿里巴巴的互联网电商应用中（例如大家所熟悉的淘宝和天猫等），整个系统架构实际上被精细地设计成数千个边界划分明确的组件，其目的就是为希望做出非破坏性变更的开发人员 提供更大的便利，避免因为不适当的耦合将变更导向难以预料的方向，从而阻碍架构的演 进。可以发现，演进式架构的软件都支持一定程度的模块化，这种模块化通常体现为经典的分层架构及微服务的最佳实践。

而在平台研发层面，演进式架构更多地体现为基于“能力”的架构(Capability Oriented Architecture，COA)。在 Kubernetes 等云原生技术逐渐普及之后，基于标准化的云原生基础设施正迅速成为平台架构的能力提供方，而以此为基础的开放应用模型(Open ApplieationModel，OAM)理念，正是一种从应用架构视角出发，将标准化基础设施按照能力进行模块化的 COA 实践。

3. 云原生下的架构演进

当前，演进式架构还处于快速成长与普及阶段。不过，整个软件工程领域已经达成共识，即软件世界是不断变化的，它是动态而非静态的存在。架构也不是一个简单的等式，它是持续过程的一种快照。所以无论是在业务应用还是在平台研发层面，演进式架构都是一个必然的发展趋势。业界大量架构更新的工程实践都诠释了一个问题，即由于忽略实现架构，且保持应用常新所要付出的精力是非常巨大的。但好的架构规划可以帮助应用降低新技术的引入成本，这要求应用与平台在架构层面满足:架构标准化、职责分离与模块化。而在云原生时代，开发应用模型(OAM)正在迅速成为演进式架构推进的重要助力。

结语

我们可以看到云原生架构的构建和演进都是以云计算的核心特性(例如，弹性、自动化、韧性)为基础并结合业务目标以及特征进行的，从而帮助企业和技术人员充分释放云计算的技术红利。随着云原生的不断探索，各类技术不断扩展，各类场景不断丰富，云原生架构也将不断演进。但在这些变化过程中。典型的架构设计原则始终都有着重要意义，指导我们进行架构设计以及技术落地。

原文链接