阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

阿里云云原生应用平台

云原生大规模应用落地指南

2020阿里巴巴双11云原生技术实践

2020 年的“全面云原生化”革命性地重构了双 11 “技术引擎”。我们在不断发展的过程中看到云原生的方向，又通过自身实践，证明这是一条正确的道路，从而更加坚定。

藏经阁

阿里云开发者电子书系列

关注阿里巴巴云原生
了解更多技术资讯

阿里云开发者“藏经阁”
海量免费电子书下载

目录

序言

推荐序 1

电刚刚被发明时，人类只用它来照明，未曾想过真正广泛应用后，由电带来的革新将远远超出人们的想象力。

数字技术也是类似的，商业世界仅停留在过去“见招拆招”的思考层面是远远不够的。现在，我们需要重新理解一个新技术带来的数字原生世界。阿里巴巴经过各类技术不断地迭代发展到今年，已经为构建这样一个世界完成了初步的技术准备。

以今年的双 11 为例，我们将“光棍节”升级为“双节棍”。过去双 11 的模式是提早把业务需求固定下来，严阵以待，调试到位，来迎接那一个晚上的峰值节点，这就像打固定靶。但今年改成了移动靶，我们需要在双 11 进程中变更系统，为消费者与商家持续创新。从固定靶到移动靶，这显然是一个巨大的挑战。我们正在挑战以前很难实现的事情。

尽管今年我们准备了更高的峰值能力，但交易峰值不再是主要技术看点了。我们希望把过去 11 年支撑双十一的技术创新融会贯通，形成一个「数字原生商业操作系统」，帮助全社会在产业链各环节的数字创新。去年双 11 的主题是核心系统「全面上云」，今年双 11 的主题是「云原生」，是在「云上」实现核心系统全面云原生化的的第一年，今年的云和去年的云有很大不同，有些技术是第一次使用。

我们看到，云的定义在不断变化，它成为了商业领域数字化的底座和基础，不再单指传统云计算了，而是将未来的方向指向云原生。这种云计算的再升级让商业效率提升和技术创新变得更加简单。某种程度上，恰恰是因为云原生，我们才能从过去的束缚中解放出来。不迈出这一步，把业务的创新空间打开，我们的综合能力很难迎来下一次突破。

阿里巴巴的使命是「让天下没有难做的生意」，技术愿景是「技术创造新商业」，沿着这个方向，数字原生商业操作系统将是阿里巴巴技术接下来的一个重点。这样的「操作系统」，在数字基础科技与数字商业创新之间架起一座桥梁，一方面，我们让商业要素的资源调度更高效，另一方面，我们让应用创新变得更简单，两者结合将产生巨大潜力。无论是打造这个新的基础平台，还是基于它来贴身为客户创造价值，都是用创新在推进全社会的商业数字化进步，值得我们所有技术人为之兴奋而奋斗不已。

阿里合伙人、阿里巴巴集团首席技术官

程立

推荐序 2

如今，企业上云已经成为一种必然趋势。与此同时，作为诞生于云计算时代的新技术理念，云原生让企业用云的方式发生着从“上云”到“云上”的转化。

云原生拥有传统 IT 无法比拟的优势，它能从技术理念、核心架构、最佳实践等方面，帮助企业 IT 平滑、快速、渐进式地落地上云之路。可以预测,在未来企业加快数字化转型的过程中,云原生一定会变成现代业务的基础应用。

通过云原生，企业可以最大化使用云的能力，聚焦于自身业务发展，开发者也可以基于云原生的技术和产品，提升开发效率，并将精力更多地聚焦于业务逻辑的实现。与此同时，云原生正在成为新基建落地的重要技术抓手。只有提前拥抱基础设施，才不会被时代淘汰。这个现象在今年体现得非常明显。疫情之下，各行各业都受到了不同程度的影响，但那些基础设施能力健全的企业抵御风险的能力更强。

比如今年疫情期间，基于阿里云容器解决方案，钉钉 2 小时内扩容 1 万台云主机，支撑 2 亿上班族在线开工；申通快递将核心系统搬到阿里云上，并进行应用容器化和微服务改造，在日均处理订单 3000 万的情况下，IT 成本降低 50%；采用了阿里云原生 PaaS 平台的中国联通号卡应用，开卡业务效率提升了 10 倍，需求响应时间缩短了 50%，支撑访问量由 1000 万上升至 1.1 亿。

经过此次疫情，越来越多的企业坚定了上云和实现数字化转型的信念和步伐，而云原生技术则是实现数字化转型的最短路径。但不得不说，今天云原生技术的应用和普及显然仍处于早期发展阶段。

如果说在过去传统的工作方式下，一家企业想使用云原生的技术或产品，需要花费大量的精力研究一些开源项目，自己做运维和管理，还需要考虑集成、稳定性保障等问题，这样才能建立一个云原生平台；今天，为了方便企业和开发者更便利地使用云原生的技术和产品，更好地接受云原生的理念，阿里云在帮助国内企业了解云原生、使用云原生上做了很多工作。

一方面，我们在内部积极去推进云原生技术的使用，阿里巴巴内部有非常丰富的、大规模的使用场景，通过这些场景可以充分打磨云原生技术；另一方面，阿里云已经拥有国内最丰富的云原生产品家族、最全面的云原生开源贡献、最大规模的云原生应用实践，去为最大

的云原生客户群体赋能。在容器、服务网格和 Serverless 等领域均为企业提供丰富的技术和产品体系，覆盖八大类别 20 余款产品，涵盖底层基础设施、数据智能、分布式应用等，可以满足不同行业场景的需求，极大地降低企业在云计算方面的部署成本，能从技术理念、核心架构、最佳实践等方面，帮助企业 IT 平滑、快速、渐进式地踏上落地上云之路。最后，也正是经历过这样的具体实践，阿里云才有底气在技术成熟以后，将其回馈到社区，帮助云原生社区提高技术质量和发展水平。

在 2020 年云栖大会期间，阿里云宣布成立“云原生技术委员会”。除了承担推动阿里巴巴全面云原生化的职责，委员会更加重要的一个责任是将阿里巴巴已经沉淀 10 多年的云原生实践对外赋能数百万家企业，帮助他们进行云原生改造，提升 30% 研发效率的同时降低 30% IT 成本，携手客户迈入数字原生时代。

云原生的核心是创新，硬核技术要创新，服务客户的模式要创新。今天我们讲云原生是阿里云的再升级，其实，云原生也是阿里云的 DNA。相信在阿里云原生的助推下，“云”也将成为“日用品”，让企业业务“生于云，长于云”，帮助企业实现全面数字化，享受云原生时代由技术带来的红利。

阿里云高级研究员、阿里云基础产品事业部负责人
蒋江伟

推荐序 3

云原生已经成为云计算的再升级，通过重塑整个软件生命周期，成为释放云价值的最短路径，加速企业数字化创新。

如果说过去企业在使用云服务时需要先采购虚拟机，再通过中间件开发应用，那么云原生就是更进一步的云计算形态体现，追求最大化利用其技术模式，充分发挥云计算的生产力。利用容器、Kubernetes、微服务、Serverless 等能力，可以让虚拟化、中间件实现进一步封装。基于云原生架构，企业只需关注自己的应用开发、交付即可，实现生于云长于云。

在这样的模式下，应用从设计、开发、交付、到管理的思维方式与最佳实践有机结合，大幅降低了 IT 实施和运维成本，让应用可以最快地创造价值，进而提升业务创新效率，这就是我们所说的“最短路径”。

今天，阿里巴巴作为一家科技公司，我们不仅要对内释放云原生技术红利，实现研发效率、资源效率和迭代创新效率的整体提升；同时，云原生也升级为阿里云的技术战略，为更多的客户和开发者赋能。为此，阿里云云原生提出了“三位一体”的策略，将“自研技术”、“开源项目”、“商业产品”形成统一的技术体系，最大化技术价值。

在阿里巴巴我们常说，没有经过双 11 检验的技术不是成熟的技术。2020 年双 11，我们实现了核心系统全面云原生化的重大技术突破，带来资源效率、研发效率、交付效率的三大提升，万笔交易的资源成本 4 年间下降 80%，研发运维效率平均增效 10% 以上，规模化应用交付效率提升了 100%。可以说，阿里巴巴在 2020 双 11 完成了全球最大规模的云原生实践。

与 2019 年全面云化相比，2020 年的“全面云原生化”革命性地重构了双 11“技术引擎”。从产品和技术两方面来看，产品侧，阿里云通过提供容器服务 ACK、云原生数据库 PolarDB/Redis、消息队列 RocketMQ、企业级分布式应用服务 EDAS、微服务引擎 MSE、应用监控服务 ARMS 等百款云原生产品全面支撑双 11。技术侧，云原生四大核心技术实现规模和创新的双重突破，成为从技术能力向业务价值成果转变的样本：

1. 支持全球最大容器集群、全球最大 Mesh 集群，神龙架构和 ACK 容器的组合，可

以实现 1 小时扩容 1 百万个容器，混部利用率提升 50%，万笔交易成本 4 年下降 80%。

2. 国内最大计算平台、顶级实时计算能力。大数据平台批处理单日计算数据量达到 1.7 EB，实时计算峰值每秒 30 亿条记录；PolarDB 读写性能提高 50%+，计算资源利用率提高 60%+。
3. 云原生中间件首次实现自研、商用、开源的"三位一体"，通过阿里云服务全球客户。云原生中间件服务框架峰值调用量超百亿 QPS。
4. 核心业务规模实践 Serverless，弹性伸缩能力会提升 10 倍，大幅提升压测支撑效率和稳定性。

云原生技术不仅在阿里内部大规模普及，也正通过阿里云服务全社会的双 11。大促期间，阿里云原生还支撑了中国邮政、申通快递、完美日记、世纪联华等客户，稳定高效应对双 11 大促的流量。以物流行业为例，申通快递将核心系统搬到云上，采用阿里云容器服务，亿级包裹过境，系统稳如泰山，IT 成本还降低了 30%；以大型商超为例，世纪联华基于阿里云函数计算(FC)弹性扩容，业务峰值 QPS 超过 2019 年双 11 的 230%，研发效率交付提效超过 30%，弹性资源成本减少 40% 以上。

回顾阿里巴巴云原生的发展历程，我们就比别人更早一些下定决心。 从 2008 年落地分布式、互联网中间件，到 2011 年落地容器化，我们在不断发展的过程中看到云原生的方向，又通过自身实践，证明这是一条正确的道路，从而更加坚定。

当然，任何一家企业，特别是规模越大、历史沉淀越多的企业，一定会有一些历史包袱。在云原生落地的过程中，即便是阿里也不可能百分百所有的技术全都云原生，全都使用阿里云的产品。云原生的动作会先从核心系统开始，因为核心系统人员更充沛，并对技术有更极致的要求。我们将今年在阿里巴巴双 11 核心系统全面云原生化过程中积累的经验沉淀成为这本电子书，希望帮助更多企业和研发人员去更好地做新技术的尝试、迭代和落地。

未来十年，云计算将无处不不在，真正地融入到我们生活的方方面面。而云原生则让云计算变得标准、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。

阿里云研究员、阿里云云原生应用平台负责人

丁宇

前言

4985 亿交易额的背后，全面揭秘阿里巴巴双 11 的云原生支撑力

在新冠肺炎疫情催化下，数字化生活方式渐成新常态。2020 天猫双 11 全球狂欢节（简称：天猫双 11）如约而至，更直观展现了数字经济的先发优势和巨大潜能。11 月 11 日零点零分 26 秒，天猫双 11 的订单创建峰值就达到 58.3 万笔/秒，阿里云又一次扛住全球最大规模流量洪峰。与此前不同的是，继去年天猫双 11 核心系统上云后，阿里巴巴基于数字原生商业操作系统，实现了全面云原生化，底层硬核技术升级带来了澎湃动力和极致效能。以支撑订单创建峰值为例，每万笔峰值交易的 IT 成本较四年前下降了 80%。

这次全球最大规模的云原生（Cloud Native）实践也引发了业界新的思考，在企业积极进行数字化转型，全面提升效率的今天，几乎无人否认云原生代表着云计算的“下一个时代”，IT 大厂们都不约而同的将其视为未来云应用的发展方向。当企业技术能力突破瓶颈，将带来业务价值转换，而云原生正是助力企业提升技术竞争力的最佳途径。那么，在双十一到来的第 12 个年头，为何“全面云原生化”可以保障顾客在流量峰值也能拥有丝般润滑的购物体验？

当大促场景成为企业业务日常，这个问题的答案非常值得借鉴。

从“双 11 上云”到“云上双 11 云原生重构双 11“技术引擎”

十二年来，双 11 交易峰值屡创新高，2020 年以 58.3 万笔/秒再创世界纪录，是 2009 年第一次双 11 的 1458 倍。阿里巴巴集团首席技术官程立表示，因为技术，双 11 才成为可能。2010 年“去 IOE”，阿里发展出互联网海量交易支付的架构与技术；2015 年提出“中台”战略，阿里迈向移动化、数据化新征程；2017 年达摩院成立，开启阿里双 11 智能时代；2019 年核心系统 100%上云，让积累多年的双 11 技术，通过阿里云实现技术红利大迸发。

从 2019“双 11 上云”到 2020“云上双 11”，阿里核心系统在 2020 年实现了全面云原生化，革命性重构了双 11"技术引擎"。具体实现可以从产品和技术两方面来看。产品侧，阿里云通过提供容器服务 ACK、云原生数据库 PolarDB/Redis、消息队列 RocketMQ、企业级分布式应用服务 EDAS、微服务引擎 MSE、应用监控服务 ARMS 等数十款云原生产品全面支撑双 11。技术侧，云原生四大核心技术实现规模和创新的双重突破，成为从技术能力向业务价值成果转变的样本：

- **极致弹性：**神龙架构和 ACK 容器的组合，可以实现 1 小时扩容 1 百万个容拥有国内最大计算平台、顶级实时计算能力，支持全球最大容器集群、全球最大 Mesh 集群。
- **极致算力：**大数据平台批处理单日计算数据量达到 1.7EB，实时计算峰值每秒 30 亿条记录；PolarDB 读写性能提高 50%+，计算资源利用率提高 60%+。
- **三位一体：**云原生中间件首次实现自研、商用、开源的"三位一体"，通过阿里云服务全球客户。云原生中间件服务框架峰值调用量超百亿 QPS。
- **Serverless 大规模实践：**核心业务首次大规模实践 Serverless，弹性伸缩能力会提升 10 倍，大幅提升压测支撑效率和稳定性。

与其他云计算公司不同的是，阿里云支撑着阿里巴巴的核心业务，阿里云原生应用平台研究员丁宇表示，"云原生是释放云计算红利的最短路径，也将成为全面上云的新底座。云原生是云计算的再升级，是真正意义的云技术革命，推动从 Cloud Hosting 演进到 Cloud Native，从基于自有、封闭的技术体系，走向标准、开放公共的云技术体系。除了支撑双 11 之外，这些双 11 的同款技术也通过阿里云支撑全社会，成为数字新基建的基础设施。"

云原生普惠技术红利 支撑每一个行业的“双 11”

如今双 11 已经成为“购物节”的代名词，任何一个行业都开始打造自己的“双 11”。普惠科技的价值已经不仅仅服务于狭义的电商双 11，而是支撑全社会的双 11 场景。云原生技术不仅在阿里内部大规模普及，也正通过阿里云服务全社会的双 11。大促期间，阿里云原生还支撑了中国邮政、申通快递、完美日记、世纪联华等客户，稳定高效应对双 11 大促的流量。

以物流行业为例，申通快递将核心系统搬到云上，采用阿里云容器服务，亿级包裹过境，系统稳如泰山，IT 成本还降低了 30%；以大型商超为例，世纪联华基于阿里云函数计算(FC)弹性扩容，业务峰值 QPS 超过 2019 年双 11 的 230%，研发效率交付提效超过 30%，弹性资源成本减少 40% 以上。在自动驾驶领域，创业公司图森利用通过阿里云的 ASK（Alibaba Cloud Serverless Kubernetes）容器服务灵活调度 AI 模型训练时的计算资源，可缩短了 60% 的模型测试时间，并在完成测试之后可以快速释放算力，极大节约了成本。

继 2020 年 9 月云栖大会上阿里巴巴宣布成立云原生技术委员会，云原生升级为阿里技术新战略。2020 双 11 核心系统全面云原生化，成为云原生技术委员会推动阿里巴巴全面云原生化的重要里程碑。阿里巴巴集团首席技术官程立表示，云原生带来最大的不同是让阿里真正实现了自研、商用、开源的"三位一体"，双 11 的核心技术可以直接给到客户使用，省略了经过云上沉淀再输出的过程，降低了客户获取"双 11 同款技术引擎"的门槛和成本。

从云计算到云原生——彻底拥抱数字时代的业务架构方式，开启云上商业时代。

正如阿里云创始人王坚博士所言，「核心系统上云让阿里巴巴和客户真正坐上了同一架飞机」，阿里云也将云原生的实践经验与各个行业分享，让「这架飞机上的每个乘客」都享受技术创新带来的红利。

第一章 技术体系升级

本章主要作者：李响、汤志敏、黄涛、司徒放、许晓斌、杨皓然

注：作者姓名按文章顺序排列

阿里云原生中间件首次实现自研、开源、商用“三位一体”，技术飞轮效应显现

对于阿里的技术同学来说，每年的 双 11 都是一场“盛宴”。为了让顾客有顺滑的购物体验，给商户提供更多样化的让利活动，阿里电商平台对于效率、可靠性、规模性的要求在 双 11 的驱动下成倍提高，激发着技术人的潜力。作为基础技术核心之一，阿里中间件也会在每年 双 11 迎来一次技术的全面演进和升级。

阿里在 2019 年完成了全站的核心系统上云，对于阿里中间件来讲，这是一个意义非凡的机遇和挑战。实际上，从 2011 年 Dubbo 开源开始，阿里中间件就已经尝试在云产品和开源方面努力探索，希望让支持阿里核心业务的中间件系统从封闭走向开放，服务更广泛的用户。过去几年，阿里云推出了 EDAS 产品线，希望能够把阿里在微服务和应用托管体系的实践经验分享给用户；与此同时，阿里云还在开源社区中推出了 Dubbo、RocketMQ、Nacos、Seata 等多个为人熟知的开源项目，鼓励广大开发者共建中间件生态体系。

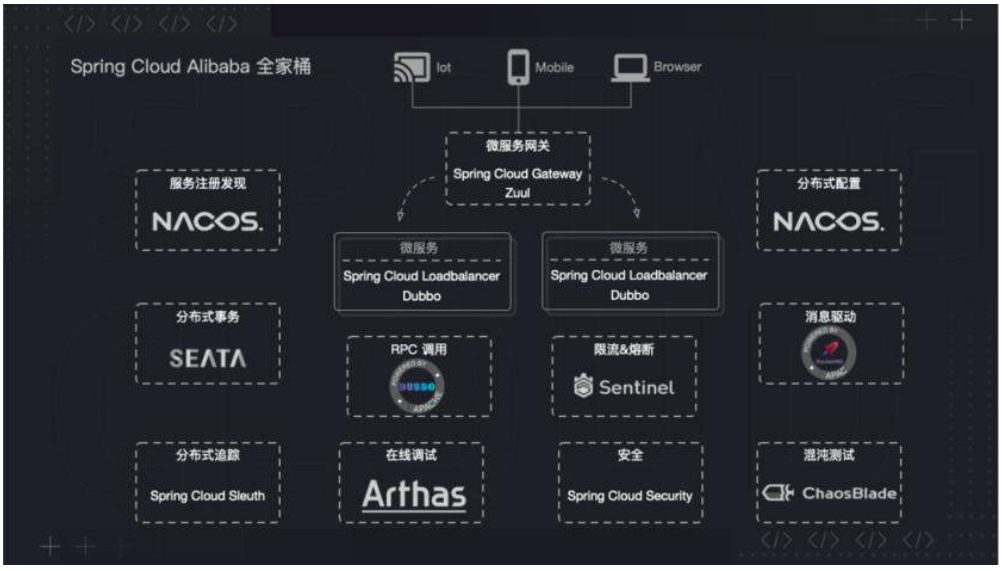
阿里云在探索中一直存在的苦恼，是内部的自研体系、商业化的产品技术与开源的项目，三方的技术路线一直没有机会融为一体。然而，就在今年阿里云提出了“三位一体”理念，即将“自研技术”、“开源项目”、“商业产品”形成统一的技术体系，最大化技术的价值。随着阿里自研体系的上云，这个机遇终于到来了。今年，让阿里云中间件技术人最兴奋的，除了支持 双 11 大促的再一次成功，更是能用这些技术持续赋能阿里云上数以万计的企业、机构、开发者以及他们的用户，把 双 11 的技术红利发挥到极致。

基于集团场景，沉淀 Spring Cloud Alibaba 全家桶，形成微服务领域最佳实践

在考拉入淘过程中，集团基于开源核心预研的下一代服务框架 Dubbo 3.0，完美融合了内部 HSF 的特性。考拉基于 Dubbo 以及 MSE 提供的服务发现和流量治理能力，轻松实现了与集团核心电商业务的接入。在今年 双 11 大促中，考拉核心链路上的数百个应用运行在 Dubbo 3.0 这个版本上。

Nacos 与 Dubbo/Spring Cloud Alibaba 生态完成无缝整合。2018 年，随着阿里开源战略的推进，阿里云以 10 年 双 11 沉淀的注册中心和配置中心为基础开源了 Nacos，以简单易用、性能卓越、高可用、特性丰富等核心竞争力快速成为领域首选。并且跟阿里 Dubbo/Spring Cloud Alibaba 生态完成无缝整合，形成微服务领域最佳实践。2020 年，随着阿里全站上云的全面推进，阿里云将阿里巴巴内部注册中心和配置中心用 Nacos 重构完成，并以云产品 MSE 支撑了淘宝、饿了么、考拉等核心 BU 平稳度过双 11。

阿里微服务体系通过阿里内部场景锻炼出高性能和高可用的核心竞争力，通过开源构建了生态和标准，凭借 MSE、EDAS 等云产品完成产品化和能力输出。基于此，阿里云中间件完成了三位一体的正向循环，通过标准持续输出阿里巴巴的核心竞争力，让外部企业快速享有阿里微服务能力，加速企业数字化转型！



Spring Cloud Alibaba 全家桶

阿里云 Prometheus 监控服务，提供了水平扩展能力，平均查询性能比开源提升 30% 以上

基础设施的自动化是云原生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。Prometheus 是 CNCF 下第二个毕业的项目，已成为云原生可观测领域的事实标准之一。如何将开源 Prometheus 的优秀生态与技术架构与阿里云原生基础设施进行整合，提供一个监、管、控一体化的自动化运维平台，提升业务系统的交付效率与在线稳定性，是阿里云这一年多来不断探索的目标。在今年的 双 11 期间我们见证了这一目标的实现，阿里云 Prometheus 服务成功为众多大规模在线业务保驾护航，帮助业务系统顺利度过洪峰。

相比于自研的监控体系，阿里云 Prometheus 服务与云生态有更紧密的集成，实现了与托管类产品底层 API 的深度集成与联动。外部用户也无需顾虑运维 Prometheus 服务，只需一键开启一组资源开销极小的无状态采集组件，即可实现自动服务发现、高可靠的数据采集与上报，以极低的迁移成本将自建 Prometheus 迁移到阿里云的 Prometheus 服务上。相比于开源版本的 Prometheus，阿里云的 Prometheus 为了应对阿里的大规模体量，提供了水平扩展能力，能够应对超大规模的指标写入，其优化后的查询引擎，针对高维查询、正则查询、长时间线查询等场景做了特定优化，平均查询性能比开源版本提升 30% 以上。

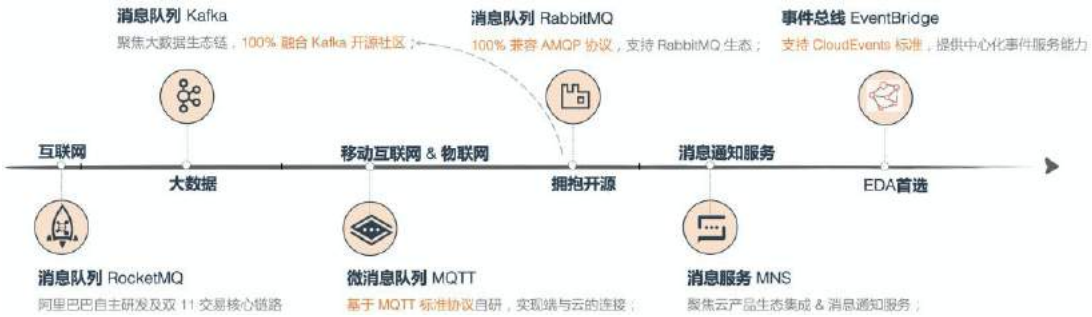
钉钉视频会议在今年基于 ASK 实现了全球系统的全量容器化，采用云原生 Serverless 技术，使得整体业务架构变得更加轻量、易运维，能够更好地应对音视频领域流量特征所带来的特殊资源弹性诉求。阿里云 Prometheus 服务针对 ASK 集群特性做了一系列定制，实现了无损的 Serverless 指标采集能力，以及钉钉视频会议整个 Serverless 架构的全局可观测能力。与此同时，我们开始在无状态工作负载下探索，基于 Prometheus 指标数据的自动弹性能力。

基于 RocketMQ 的消息产品家族无缝快速上云，拥抱标准，引领标准

RocketMQ 是阿里巴巴在 2012 年开源的第三代分布式消息中间件，并在 2017 年正式成为 Apache 顶级开源项目。在阿里巴巴内部，RocketMQ 一直承载着阿里巴巴

所有核心链路的消息流转，历经多年 双 11 万亿级消息洪峰的严苛考验。随着阿里全站上云战略的推进，阿里云消息团队打造了三位一体的技术融合架构，克服了微内核抽象统一、商业化差异性打造等难关，实现了自研、开源、商用三方技术的平滑兼容，以同一消息体系支撑阿里巴巴、阿里云产品以及开源社区需求。通过三种截然不同场景的打磨，RocketMQ 可以帮助用户无缝快速上云。

今年 双 11，菜鸟、饿了么、考拉等阿里巴巴核心部门将其消息系统迁移到云上消息产品，相比于原有需要提前预算规划的使用方式，云消息产品为其提供了快速按需扩缩容的弹性能力，不仅节省了成本，也消除了其对容量预估失准的担忧。除此之外，三位一体技术融合也为内部用户带来了诸多便利。RocketMQ 开源社区中的一大批生态项目可以快速在阿里巴巴内部以及云上得以复用，不仅节省了开发成本，也使得开发模式从依赖阿里巴巴内部组件的封闭方式，走向与社区开源生态协同的开放方式，在拥抱标准的同时引领标准。而这些生态项目通过阿里巴巴内部严苛场景的打磨，也正在变的愈发成熟，吸引着越来越多的开发者。



消息队列产品家族

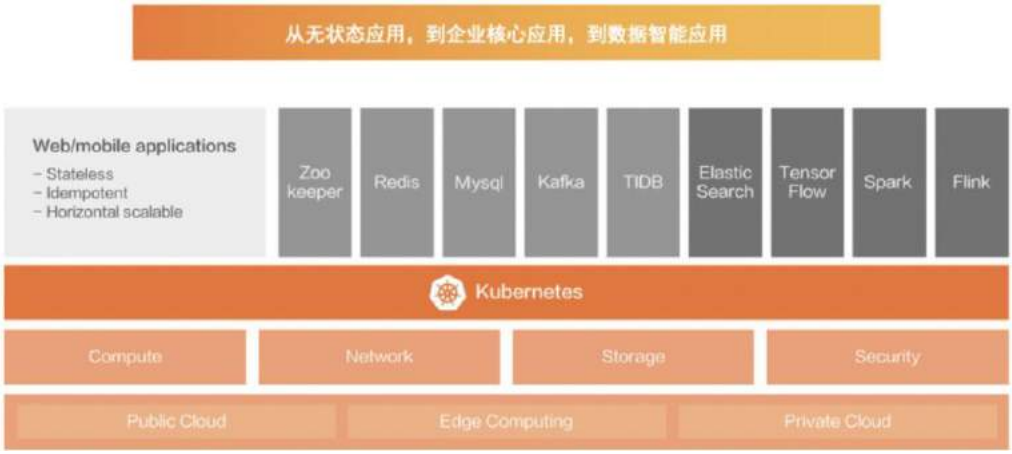
目前，阿里云消息产品已服务于数千家付费企业用户，为其提供开箱即用，稳定可靠的消息服务。三位一体技术融合使得 RocketMQ 不仅让阿里成熟稳定的技术能够服务外部客户，造福无数企业和开发者，也通过开源与开放的共赢方式，消除了用户被厂商锁定的担忧。继今年 9 月云栖大会上阿里巴巴宣布成立云原生技术委员会，云原生升级为阿里技术新战略。2020 双 11 核心系统全面云原生化，成为云原生技术委员会推动阿里巴巴全面云原生化的重要里程碑。阿里巴巴集团首席技术官程立表示，“云原生带来最大的不同是让阿里真正实现了自研、商用、开源的“三位一体”，双 11 的核心技术可以直接给到客户使用，省略了经过云上沉淀再输出的过程，降低了客户获取 “双 11 同款技术引擎” 的门槛和成本，可帮助客户快速迈入数字原生时代。”我们坚信驱动技术演进的背后一定是复杂的业务场景、严格的稳定性和挑战以及来自于用户的的信任和支持。

以 Kubernetes 为代表的容器技术，已成为云计算的新界面

2020 年 双 11，阿里核心系统实现了全面云原生化，扛住了史上最大流量洪峰，向业界传达出了“云原生正在大规模落地”的信号。这里包含着诸多阿里 "云原生的第一次”，其中非常关键的一点是 80% 核心业务部署在阿里云容器 ACK 上，可在 1 小时内扩展超百万容器。

可以说，以 Kubernetes 为代表的容器技术正成为云计算新界面。容器提供了应用分发和交付标准，将应用与底层运行环境进行解耦。Kubernetes 作为资源调度和编排的标准，屏蔽底层架构差异性，帮助应用平滑运行在不同基础设施上。CNCF Kubernetes 的一致性认证，进一步确保不同云厂商 Kubernetes 实现的兼容性，这也让更多的企业愿意采用容器技术来构建云时代的应用基础设施。

云原生容器新界面的崛起



作为容器编排的事实标准，Kubernetes 支持 IaaS 层不同类型的计算、存储、网络等能力，不论是 CPU、GPU、FPGA 还是专业的 ASIC 芯片，都可以统一调度、高效使用异构算力的资源，同时完美支撑各种开源框架、语言和各类型应用。

伴随着 Kubernetes 成为新操作系统的事实，以云原生容器为主的技术，已经成为云计算的新界面。

（一）云原生容器界面特征

云原生容器界面具有以下三个典型特征：

- 向下封装基础设施，屏蔽底层架构的差异性。
- 拓展云计算新边界，云边端一体化管理。
- 向上支撑多种工作负载和分布式架构。

1）向下封装基础设施，屏蔽底层差异性

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，通过云原生提倡的 DevOps 文化和工具集的使用有效提升技术迭代速度，因此整体上可以降低人力成本。

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% 的 AI 任务运行在容器和 Serverless 上”，而 AI 模型训练和大数据计算类工作负载更加需要 Kubernetes 提供更低的调度延迟、更大的并发调度吞吐和更高的异构资源利用率。

加速数据服务的云原生化：由于计算存储分离具备巨大的灵活性和成本优势，数据服务的云原生化也逐渐成为趋势。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，也可以大大提升数据计算类和 AI 任务的计算效率。

安全能力进一步加强：随着数字经济的发展，企业的数据资产成为新“石油”，大量数据需要在云端进行交换、处理。如何保障数据的安全、隐私、可信成为了企业上云的最大挑战。我们需要用技术手段，建立数字化信任基础，保护数据，帮助企业创建可信任的商业合作关系，促进业务增长。比如基于 Intel SGX 等加密计算技术，阿里云为云上客户提供了可信的执行环境。不过，可信应用开发和使用门槛都很高，需要用户对现有应用进行重构，处理大量的底层技术细节，让这项技术落地非常困难。

2）拓展云计算新边界，云边端一体化管理

随着边缘计算的场景和需求不断增加，“云边协同”、“边缘云原生”正在逐渐成为新的技术焦点。Kubernetes 具有强大的容器编排、资源调度能力，可以满足边缘/IoT 场景中，对低功耗、异构资源适配、云边网络协同等方面的独特需求。为了推动云原生和边缘计

算交叉领域的协同发展，阿里巴巴于 2020 年 5 月正式对外开源边缘计算云原生项目 OpenYurt，推动“云边一体化”概念落地，通过对原生 Kubernetes 进行扩展的方式完成对边缘计算场景需求的支持，其主要特性有：

- “零”侵入的边缘云原生方案：提供完整的 Kubernetes 兼容性，支持所有原生工作负载和扩展技术（Operator/CNI/CSI 等）；可以轻松实现原生 Kubernetes 集群一键转化为 OpenYurt 集群。
- 节点自治：具备云边弱网或断网环境下的边缘节点自治、自愈能力，保障业务连续性。
- 针对海量边缘节点的应用交付，可提供高效、安全、可控的应用发布和管理方式。

2019 年 KubeCon 上阿里云发布边缘容器服务 ACK@Edge，OpenYurt 正是其核心框架。短短一年，ACK@Edge 已经应用于音视频直播、云游戏、工业互联网、交通物流、城市大脑等场景中，并服务于盒马、优酷、阿里视频云和众多互联网、新零售企业。同时，作为 ACK@Edge 的开源版本 OpenYurt，已经成为 CNCF 的沙箱项目，推动 Kubernetes 上游社区兼顾边缘计算的需求，欢迎各位开发者一起共建，迎接万物智联的新时代。

3）向上支撑多种工作负载和分布式架构

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

Kubernetes 可以向上支持众多开源主流框架构建微服务、数据库、消息中间件、大数据、AI、区块链等各种类型应用。从无状态应用、到企业核心应用、再到数字智能应用，企业和开发者都可以基于 Kubernetes 顺利地自动部署、扩展和管理容器化应用。

（二）阿里巴巴如何理解云原生容器界面

阿里巴巴将云原生看作未来重要的技术趋势，为了更快加速、更好协同，制定了清晰的阿里巴巴云原生技术路线，举集团之力统筹推动云原生。

在云原生容器界面的指引下，阿里巴巴集团以基础设施、运维及其周边系统作为切入点，掀起全面云原生化的浪潮，陆续将系统改造为适配云原生架构的新方案，推动集团内部使用

的技术框架、工具等被云可接受的标准产品或云产品替代；进一步转变运维思路和工作方式，兼容适配新的运维模式。例如：DevOps 需要改变传统虚拟机时代的运维思想，容器运行时的组件要改为支持 Kubernetes Pod 下的新模式，容器内日志、监控等各类运维组件都需要变化、运维模式也随之变化。

在计算、网络、存储方面，用户通过 Kubernetes 的统一管理，可以充分利用阿里云的 IaaS 能力，让每个业务拥有自己独立的弹性网卡和云盘，对网络和存储性能有着高低不同需求的业务，也有能力部署在同一台宿主机上，并保证互相不被干扰的隔离性。传统非云物理机机型决定业务部署类型，会导致的弹性不足问题，也得到了很好的解决。因此，用户在提升资源利用率、降低成本的同时，也极大提升了业务的稳定性。

在节点资源层，用户可充分利用 Kubernetes 的底座扩展能力，让节点管理实现云原生化；在架构层面，通过节点生命周期控制器、自愈控制器和组件升级控制器等，可实现节点自愈、流转、交付、环境组件变更的节点生命周期的完全闭环，让容器层完全屏蔽底层节点感知，完全改变了节点的运维管理模式。基于强大的云原生节点管理模式，阿里巴巴内部将集团之前相对割裂的节点资源整合为一体，真正实现了资源池从点形成面，将内核、环境组件、机型规格等进行统一标准整合，资源池的大一统并结合统一调度形成巨大的弹性能力，这也是云原生节点管理中的『书同文，车同轨，度同制，行同伦，地同域』，让节点资源从诸侯格局变成了大一统的云原生资源池。

新兴的生态和业务，基于 ACK（阿里云容器服务）提供的云原生土壤，如 Service Mesh、Serverless、Faas 等，也非常快速地在集团内落地，并得到蓬勃发展。

在应用 PaaS 层，云原生的应用交付模式走向了更为彻底的容器化，充分利用了 Kubernetes 的自动化调度能力，基于 OAM Trait 的标准定义来构建集团内统一的 PaaS 运维能力，基于 GitOps 研发模式让基础设施和云资源代码化、可编程。

阿里集团向云原生容器界面的演进

为了支撑阿里集团庞大而复杂的业务，十年之间，众多技术工程师走出了一条深深浅浅的容器之旅。 那么，在阿里集团内部，容器界面是如何演进的呢?

在过去十年，阿里集团内的容器技术，经历了从自研 LXC（Linux Container）容器 T4，到富容器，再到 Kubernetes 云原生轻量级容器的演进历程。每一次转变升级，都是基于不同时期的业务背景，所做出的技术迭代和自我革新。

第一阶段：基于 LXC 的容器 T4 尝试

受困于虚拟机 KVM 的巨大开销，以及 KVM 编排管理的复杂度，阿里集团在 2011 年时发起对 LXC 和 Linux Kernel 的定制，在内部上线了基于 LXC 的 T4 容器。但相比后面出现的 Docker，T4 容器在技术上存在一些不足，比如没有实现镜像提取和应用描述。T4 诞生后的多年，阿里持续尝试在 T4 之上构建复杂的基线定义，但屡屡遭遇问题。

第二阶段：引入容器镜像机制的 AliDocker，实现大规模分发

2015 年，阿里引入 Docker 的镜像机制，将 Docker 和 T4 的功能取长补短互相整合，即：让 T4 具备 Docker 镜像能力，同时又让 Docker 具备了 T4 对内部运维体系的友好性，并在此基础上形成内部产品 AliDocker。

过程中，阿里引入 P2P 镜像分发机制，随着电商核心应用逐步全面升级到 AliDocker，通过宿主机的环境隔离性和移植性，屏蔽了底层环境差异，为云化/统一调度/混部/存储计算分离等后续基础架构变革打下了基础，镜像机制的优势得以体现。其中，孵化的 P2P 镜像分发是 2018 年 10 月加入 CNCF 的 Dragonfly。

第三阶段：完全自主产权的容器 Pouch，阿里内部全面容器化

随着容器技术的规模化铺开，AliDocker 化的优势得以体现，阿里完全自主产权的 Pouch 得以展开并逐渐替代 AliDocker。同时，阿里集团 100% Pouch 化也一直在快速推进，2016 年 双 11 前，已经实现了全网的容器化。

Pouch 寓意是一个神奇的育儿袋，为里面的应用提供贴心的服务。因为 Pouch 统一了集团在线应用的运行时，应用开发人员就无需关注底层基础设施的变化。接下来的数年，底层基础设施发生了云化、混部、网络 VPC 化、存储无盘化、内核升级、调度系统升级等各种技术演进，但 Pouch 容器运行时使绝大部分底层变化对应用无感知，屏蔽了对上

层应用的影响。Pouch 自身也把运行时从 LXC 切换到了 runC，并将其核心技术反哺给开源社区，同时集团也逐步将过去的存量 AliDocker 实例无缝切换为开源的 Pouch 实现。

过程中富容器模式的存在，一方面让用户和应用能够无缝平滑地切换到容器化，另一方面应用依赖的各种运维、监控、日志收集等运维系统，基于富容器模式也得以跟随容器化平滑迁移。

但富容器也有着较多缺点。由于富容器中可以存在多个进程，并且允许应用开发和运维人员登录到容器中，这违反了容器的“单一功能”原则，也不利于不可变基础设施的技术演进。例如：Serverless 演进过程中，调度插入的代理进程实际上是与应用无关的，一个容器中有太多的功能也不利于容器的健康检查和弹性。

容器化是云原生的必经之路。阿里集团正是通过这种方式，快速走完了容器化这一步，极大加速了云原生的进一步演进。全面容器化后，云原生的大势已经不可阻挡，越来越多新的理念和应用架构在容器生态中成长起来，基于容器和镜像的应用打包、分发、编排、运维的优势被越多的人看到、接受和拥抱，各种运维系统开始适配云原生架构。

第四阶段：调度系统兼收并蓄及 ACK 的演进

随着以 Kubernetes 为代表的容器技术成为云计算的新界面，阿里自研的 Sigma 也在持续探索 Kubernetes 的落地实践，并借助集团全面上云的契机，最终实现了从 Sigma 管控到 ACK 的全面迁移。

2018 年，集团调度系统开始了从内部定制的 Sigma 到 ACK 的逐步演进，容器轻量化成为一个重要的演进目标。云原生浪潮下，集团内部的运维生态也随之快速演进。轻量化容器的解决思路是用 Kubernetes 的 Pod 来拆分容器，剥离出独立的运维容器，并将众多与应用无关的运维进程逐个转移至运维容器。

Sigma 诞生之初致力于将阿里集团众多割裂的在线资源池整合统一，在此基础上，不断探索新的资源混跑形态，包括在离线混部、离在线混部、Job 调度、CPUShare、VPA 等众多技术。通过提升阿里集团数据中心整体资源利用率，带来巨大的成本节约效益。基于全托管免运维 Sigma Master、公共大资源池、应用额度服务，提供 Serverless 的资源

交付和最佳的用户体验。Sigma 调度也加速了 T4 到 Pouch 的全面容器化进程，通过应用研发自定义的 Dockerfile 标准化容器，以及透明化基础设施的 Sigma 调度引擎，业务研发已无需关心底层运维，工作重心得以聚焦于业务本身。

从 Sigma 到 ACK 的升级，是希望 ACK 领先的云产品能力得以赋能阿里集团，使得 Sigma 可以加速享受云计算的能力，包括异构资源的统一管理、面向全球化的安全合规等。但实际上，迁移 ACK 的过程并非一帆风顺：

首先，围绕着核心管控链路，阿里原有的规模和复杂场景能力、原有的庞大存量容器如何迁移到新的平台，以及容器界面如何兼容并影响现有的庞大生态体系升级，实际上都会成为演进中的包袱和劣势。实现在高速飞行中换引擎并解决存量迁移问题的难度，这在业界都有共鸣。

其次，性能、多集群运维、安全防御、稳定性等众多问题，都是全面迁移 ACK 的挑战。围绕着性能，阿里基于原生 Kubernetes 做了非常多的优化并回馈给社区，如 Cache Index、Watch Bookmark 等，并建设了一整套 Kubernetes 规模化设施，包括安全防御组件、OpenKruise、多集群组件发布等能力等。

围绕 “阿里巴巴调度 = ACK + 阿里巴巴扩展” 的总体思路，阿里集团内部迁移至 ACK 过程中的积累又能沉淀给云，丰富产品能力，帮助客户形成云上的竞争力。至此，阿里集团内部、阿里云、开源社区形成了非常好的技术合力，自研、商用、开源，三位一体融合互补。

自研、商用、开源，三位一体融合互补

技术和业务是相辅相成的，业务为技术提供场景促进技术进步；技术的进步反过来带动业务更好的发展。复杂而丰富的场景，提供了一个天然肥沃的土壤，进一步推动阿里技术的发展。阿里集团的技术一直持续保持先进。在过去，业内一直非常领先的中间件、容器、调度等各类技术，阿里都率先应用于业务，并将能力沉淀到云产品再输送给客户，助力企业加速数字化转型，产生了广泛的引领者影响力。

但在新云原生时代，如何在云原生标准下持续保持这份影响力，我们看到了更多挑战。上述的阿里容器界面演进简史记录了一线阿里工程师们如何应对这些挑战。更抽象地讲，这

些得益于阿里巴巴云原生技术体系自研、商用、开源三位一体的战略决策。

（一）阿里云侧的挑战

阿里云过去面对的用户大部分是普适性用户，而阿里集团内部场景的诉求是需要解决大规模、超高性能等问题，阿里云产品能否很好地兼顾和支撑是非常大的挑战。进一步考虑，如果我们能很好地抽象出大众用户的诉求，阿里集团对阿里云来说又是一个非常好的“试炼场”。

（二）集团内部的挑战

船小好调头，而船大就没那么灵活了。过去业界独有的阿里集团内部庞大规模场景，现在反而是迈向云原生的包袱。问题的根本在于如何让阿里集团的技术能够快速融合和贡献云原生标准，而不是形成技术孤岛。

（三）开源侧的挑战和机遇

开源侧的挑战和机遇：阿里云在云原生开源项目贡献中有着持续投入，推出了 OpenKruise、联合微软推出 OAM、KubeVela 等开源项目，这些都源于阿里巴巴在云原生领域的沉淀，并且通过开源社区用户的反馈，完善在阿里云原生落地的解决方案。以 OpenKruise 为例，该项目是阿里巴巴打造的一个基于 Kubernetes 的、面向大规模应用场景的通用扩展引擎，它的开源使每一位 Kubernetes 开发者和阿里云上的用户都能便捷地使用阿里巴巴内部云原生应用统一的部署发布能力。当社区用户或外部企业遇到了 Kubernetes 原生 workload 不满足的困境时，企业内部不需要重复造一套相似的“轮子”，而是可以选择使用 OpenKruise 的成熟能力。而且，阿里集团内部使用的 OpenKruise 和开源社区版本中有 95% 以上的代码是完全一样的。我们希望和每一位参与 OpenKruise 建设的云原生爱好者，共同打造了这个更完善、普适的云原生应用负载引擎。

云原生操作系统的进化

如今，在云原生应用架构界面层，阿里集团的技术体系正在全面切向云原生技术、云产品。

阿里云为客户提供的云原生操作系统，首先基础设施层是强大的 IaaS 资源，基于第三代神龙架构的计算资源可以更弹性的扩展，以更加优化的成本提供更高的性能；云原生的分布式文件系统，为容器持久化数据而生；云原生网络加速应用交付能力，提供应用型负载均衡与容器网络基础设施。其次在容器编排层，阿里云容器服务自 2015 年上线来，伴随数千家企业客户，共同实践过各行各业大量生产级场景。越来越多的客户以云原生的方式架构其大部分甚至全量应用，随着业务深入发展，为了满足大中型企业对可靠性、安全性的强烈需求，阿里云推出新品可供赔付 SLA 的容器服务企业版 ACK Pro，并同样支撑了阿里集团内部的众多产品的落地。容器服务 ACK Pro 版，针对金融、大型互联网、政企客户的需求，支持更大规模集群，更高性能和更加全面的安全防护。

首先，基于神龙架构，软硬一体化优化设计，提供卓越性能：

- 无损 Terway 容器网络，简化数据链路，相比路由网络延迟下降 30%。
- 支持全球首款持久性内存实例，相比 NVMe，I/O 密集应用 TPS 提升 100%。

其次，提供对异构算力和工作负载优化的高效调度：

- 智能 CPU 调度优化，在保障 SLA 和密度的前提下，Web 应用 QPS 提升 30%。
- 支持 GPU 算力共享，AI 模型预测成本节省 50% 以上。

最后，为企业提供全面安全防护：

- 支持阿里云安全沙箱容器，满足企业客户对应用的安全、隔离需求，性能相比开源提升 30%。

- 国内首批通过可信云容器安全先进性认证，支持运行时风险秒级阻断。

同时，阿里云全托管托管服务网格 ASM 正式商用化，这是业内首个全托管 Istio 兼容服务网格 ASM。ASM 可以实现多种异构应用服务统一治理，提供了对云上虚拟机，容器，弹性容器实例，和 IDC 应用等异构服务的统一管理，提供全链路可观测性和端到端安全防护。帮助您加速企业应用的现代化改造，轻松构建混合云 IT 架构。

阿里云容器服务连续两年国内唯一进入 Gartner《公有云容器服务竞争格局》报告；在 Forrester 首个企业级公共云容器平台报告中，阿里云容器服务位列 Strong Performer，中国第一。

展望

云计算的未来是云原生，容器新界面是进化中的关键一小步。向下，容器新界面带来的高密度、高频度的能力要求会进一步催熟云计算的端到端优化；向上，基于容器新界面的 Serverless、新一代的中间件、新一代的应用 PaaS 方兴未艾。

云原生技术正成为释放云价值的最短路径，未来，阿里巴巴将会持续在云原生上进行投入，而阿里的云原生技术不仅会在内部大规模普及，也通过阿里云服务全社会。

Serverless 如何落地？揭秘阿里核心业务大规模落地实现

2020 年，新冠肺炎疫情催化数字化生活方式渐成常态。在企业积极进行数字化转型，全面提升效率的今天，几乎无人否认背负“降本增效”使命诞生的 Serverless 即将成为云时代新的计算范式。Serverless 将开发者从繁重的手动资源管理和性能优化中解放出来，正在引发云计算生产力的新变革。然而，Serverless 的落地问题却往往很棘手，例如传统项目如何迁移到 Serverless，同时保障迁移过程业务连续性，在 Serverless 架构下如何提供完善的开发工具、有效的调试诊断工具，如何利用 Serverless 做更好的节约成本等，每一个都是难题。尤其涉及到在主流场景大规模的落地 Serverless ，更是并非易事。正因为这样，业界对于 Serverless 核心场景规模化落地最佳实践的呼唤更加迫切。总交易额 4982 亿元，订单创建峰值 58.3 万笔/秒，2020 年天猫双 11 又一次创造记录。对于阿里云来说，今年的双 11 还有另一个意义，阿里云实现了国内首例 Serverless 在核心业务场景下的大规模落地，扛住了全球最大规模的流量洪峰，创造了 Serverless 落地应用的里程碑。

Serverless 落地之痛

挑战一：冷启动耗时长

快弹是 Serverless 天然自带的属性，但是快弹的条件是要有极致的冷启动速度去支撑。在非核心的业务上，毫秒级别的延时，对业务来说几乎不受影响。但是，对于核心业务场景，延时超过 500 毫秒已经会影响到用户体验。虽然 Serverless 利用轻量化的虚拟技术，不断的降低冷启动，甚至某些场景能降低到 200 毫秒以下。但这也只是理想的独立运行场景，在核心业务链路上，用户不仅是运行自己的业务逻辑，还要依赖中间件、数据库、存储等后端服务，这些服务的连接都要在实例启动的时候进行建连，这无形中加大了冷启动的时间，进而把冷启动的时间加长到秒级别。对于核心在线业务场景来说，秒级别的冷启动是不可接受的。

挑战二：与研发流程割裂

Serverless 主打的场景是像写业务函数一样去写业务代码，简单快速即可上线，让开发者在云上写代码，轻松完成上线。然而在现实中，核心业务的要求把开发者从云上拉回到现实，面对几个灵魂拷问：如何做测试？如何灰度上线？如何做业务的容灾？如何控制权限？当开发者回答完了这些问题，就会变的心灰意冷，原来在核心业务上线中，“函数正常运行”只占了小小的一环，离上线的距离还有长江那么长。

挑战三：中间件的连通问题

核心在线业务不是独立函数孤立运行的，需要连接存储、中间件、数据中后台服务，获取数据后再计算，进而输出返回给用户。传统中间件客户端需要打通和客户的网络、初始化建连等一系列操作，往往会使函数启动速度下降很多。Serverless 场景下实例生命周期短、数量多，会导致频繁建连、连接数多的问题，因此针对在线核心应用常用的中间件的客户端进行网络连通优化，同时对调用链路进行监控数据打通，帮助 SRE （Site Reliability Engineer ）从业者更好的评估函数的下游中间件依赖情况，对于核心应用迁移上 Serverless 非常重要。

挑战四：可观测性差

用户大多数的核心业务应用多采用微服务架构，看核心业务应用的问题也就会带有微服务的特性，比如用户需要对业务系统的各种指标进行非常详尽的检查，不仅需要检查业务指标，还需要检查业务所在系统的资源指标，但是在 Serverless 场景中没有机器资源的概念，那这些指标如何透出？是否只透出请求的错误率和并发度，就可以满足业务方的需求？实际上，业务方的需求远不止这些。可观测性做的好坏还是源于业务方是否信任你的技术平台。做好可观测性是赢得用户信任的重要前提。

挑战五：远程调试难度高

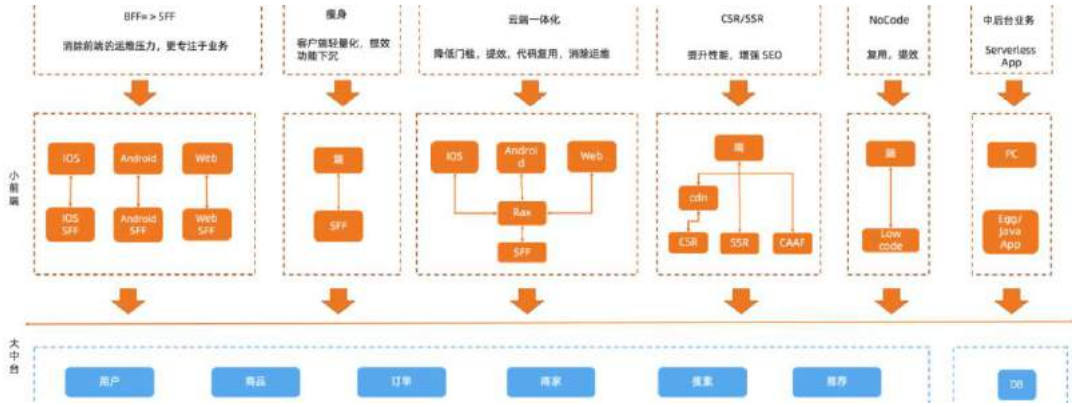
当核心业务出现线上问题时，需要立即进入调查，而调查的第一要素就是：现场的保留，然后登陆进行调试。而在 Serverless 场景中没有机器层面的概念，所以如果用户想登陆机器，在现有的 Serverless 基础技术之上是很难做到的。当然原因不仅限于此，比如 Vendor-lockin 的担心等。上面几大类痛点的概括，主要是针对开发者的开发体验，对于实际的开发场景中，是否真的是"提效"，而不是新瓶装旧酒。目前仍有大部分核心应用开发者对 Serverless 还是持有观望状态，当然也不乏一些质疑观点，“FaaS 只适合小业务场景以及非核心业务场景”。

Serverless 的 双 11 “大考”

2019 年 12 月咨询公司 O'Reill 发布 Serverless 使用调研中，已有 40% 的受访者所在的组织采用了 Serverless。2020 年 10 月，中国信息通信研究院发布的《中国云原生用户调研报告》指出：“Serverless 技术显著升温，近 30%的用户已在生产环境中应用。”2020 年，越来越多企业选择加入 Serverless 阵营，翘首以待更多 Serverless 规模化落地核心场景的案例。面对 Serverless 开发者数量的稳步增长的现状，阿里巴巴年初就制定了“打造 Servrelss 双 11”的策略，目的不只是单纯的去抗流量、打峰值，而是切实的降成本，提高资源利用率，通过“双 11 技术炼金炉”把阿里云 Serverless 打造成更安全、更稳定、更友好的云产品，帮助用户实现更大的业务价值。与过去 11 年的双 11 都不同的是，继去年天猫双 11 核心系统上云后，阿里巴巴基于数字原生商业操作系统，实现了全面云原生化，底层硬核技术升级带来了澎湃动力和极致效能。以支撑订单创建峰值为例，每万笔峰值交易的 IT 成本较四年前下降了 80%。Serverless 也迎来了首次在双 11 核心场景下的规模化落地。

场景一：前端多场景

2020 双 11，阿里巴巴集团前端全面拥抱云原生 Serverless，淘系、飞猪、高德、CBU、ICBU、优酷、考拉等十数 BU ，共同落地了以 Node.js FaaS 在线服务架构为核心的云端一体研发模式。今年 双 11 在保障稳定性、高资源利用率的前提下，多 BU 的重点营销导购场景实现了研发模式升级。前端 FaaS 支撑的云端一体研发模式交付平均提效 38.89%。依托 Serverless 的便利性和可靠性，淘宝、天猫、飞猪等 双 11 会场页面快捷落地 SSR 技术，提高了用户页面体验，除了保障大促以外，日常弹性下也较以往减少 30% 计算成本。



场景二：个性化推荐场景

Serverless 天然的弹性伸缩能力，是“个性化推荐业务场景”选择由 Serverless 实现的最重要原因，数以千计的异构应用运维成本一直是这个场景下的痛点。通过 Serverless 化进一步释放运维，让开发者专注于业务的算法创新。目前这个场景的应用范围越来越广，已经覆盖了几乎整个阿里系 APP：淘宝，天猫，支付宝，优酷，飞猪等等，因此我们可以对机器资源利用率方面做更多的优化，通过智能化的调度，在峰值时的机器资源利用率达到了 60%.

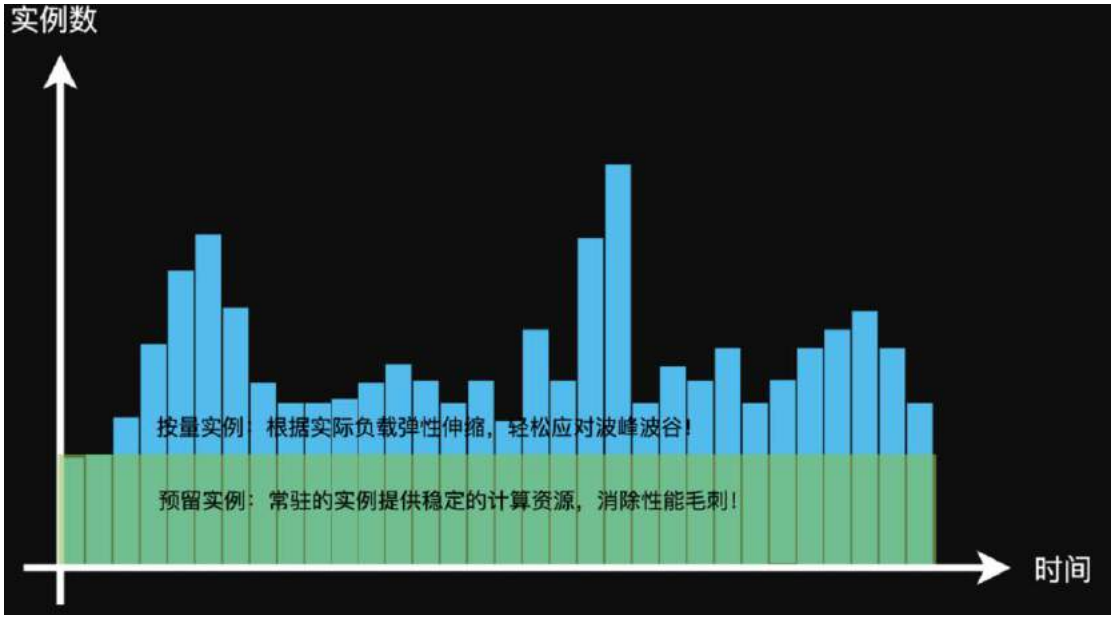
场景三：中、后台场景

2020 年，世纪联华 双 11 基于阿里云函数计算(FC)弹性扩容，在大促会场 SSR、线上商品秒杀、优惠券定点发放、行业导购、数据中台计算等多个场景进行应用，业务峰值 QPS 超过 2019 年 双 11 的 230%，研发效率交付提效超过 30%，弹性资源成本减少 40% 以上。

当然，适用于 Serverless 的场景还有很多，需要更多行业的开发者们共同丰富。总的来说，今年 FaaS 的成绩单非常耀眼，在 双 11 大促中，不仅承接了部分核心业务，流量也突破新高，帮助业务扛住了百万 QPS 的流量洪峰。

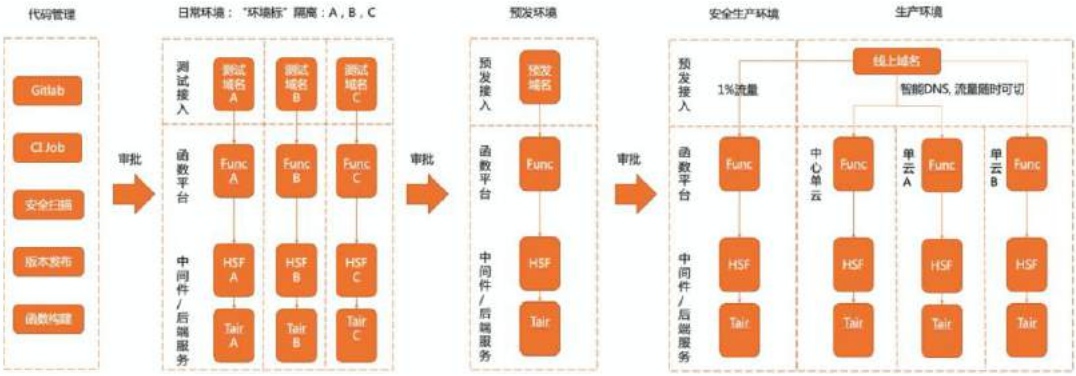
阿里云如何击破 Serverless 痛点?

那么，面对行业共有的 Serverless 落地之痛，阿里云是如何克服的呢？预留模式 + 按量模式消除冷启动在 2019 年的 Serverless 2.0 重大升级中，阿里云函数计算率先支持了预留模式，接着 AWS Lambda 几个月后，也上线了类似的功能。为什么阿里云会率先提出这个问题？阿里云 Serverless 团队不断探索真实业务的需求，按量模式的按需付费模式，虽然非常的诱人，但是冷启动时间过长，因此把核心在线业务拒之门外。接下来阿里云着重分析了核心在线业务的诉求：延时小，保证资源弹性。那如何解决呢？请看下图，一个非常典型的业务曲线图，用预留模式方式满足底部固定的量，用弹性能力去满足 burst 的需求。针对 burst 扩容，我们利用两种扩容方式结合进行满足：按资源扩容 与按请求扩容，比如用户可以只设置 CPU 资源的扩容阈值为 60%，当实例的 CPU 达到阈值后，就会触发扩容。此时的新请求并没有立即到扩容实例，而是等待实例准备好后再导流，从而避免了冷启动。同理，如果只设置了并发度指标的扩容阈值为 30（每一个实例承载的并发度），同样满足这个条件后，也会触发同样流程的扩容。如果两个指标都进行了设置，那么先满足的条件会先触发扩容。通过丰富的伸缩方式，阿里云函数计算解决了 Serverless 冷启动的问题，很好的支撑了延时敏感业务。



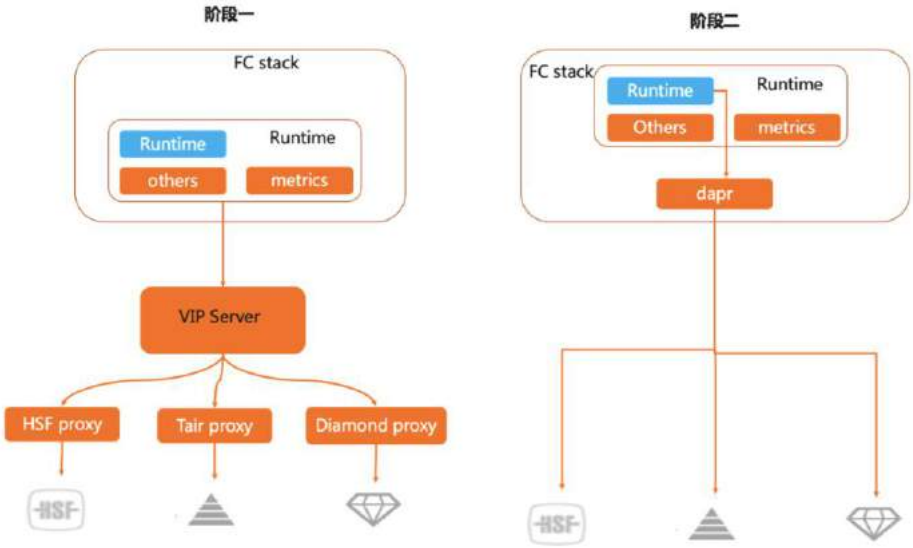
核心业务研发提效 38.89%

“提升效率”本应该是 Serverless 的优势，但对于核心应用来说，"快" = "风险大"，用户需要经过 CI 测试，日常测试，预发测试，灰度部署等几个流程验证，才能确保函数的质量。这些流程是阻碍核心应用使用 FaaS 的绊脚石。针对于这个问题，阿里云函数计算的策略是" 被集成“，把研发平台的优势与阿里云函数计算进行结合，既能满足用户的 CI/CD 流程，又能享受到 Serverless 的红利，帮用户跨过使用 FaaS 的鸿沟。阿里集团内部通过暴露标准的 OpenAPI 与各个核心应用的研发平台进行集成，经过双十一业务研发的验证，研发效率大大提高了 38.89 %。在公有云上我们与云效平台集成，把研发流程 与 FaaS 结合的更紧密、更顺畅，帮助集团外的业务提高人效。



中间件连通

核心应用离不开上下游的配合，一旦核心应用使用了函数计算，又该如何与中间件相配合？传统应用开发需要集成各类中间件的 SDK，进行打包上线，但对于 Serverless 的函数来说，代码包的大小就是一个硬伤，这个问题将将直接影响冷启动的时间。阿里云函数计算经过两个阶段的发展，第一个阶段我们通过搭建中间件 Proxy，通过 Proxy 去打通中间件，函数只用单一的协议与 Proxy 进行交互，从而 offload 掉中间件的 SDK 的包袱。第二个阶段：随着中间件能力的下沉，一些控制类型的需求也被提上了议程，比如：命令下发，流量管理，配置拉取等等，期间阿里云拥抱了开源组件 Dapr，利用 Sidecar 的方式 Offload 中间的交互成本。上述的方案，是基于阿里云函数计算的 Custom Runtime，以及 Custom Container 功能完成的。



极致的开发体验

远程调试，日志查看，链路追踪，资源利用率 ，以及完善周边工具链是开发者的必备能力。阿里云函数计算同时启动了不同的攻关小组，首先与 Tracing/ARMS 结合，打造清晰的链路追能力，与 SLS 集成打造了全面的业务数据监控。因此，业务可以根据需求进行自定义，并且拥抱开源产品 Prometheus 暴露出资源利用率，支持远程调试能力的 WebIDE。再加上阿里云近期刚开源的重磅武器：Serverless-devs ，一个无厂商绑定的、帮助开发者在 Serverless 架构下实现开发/运维效率翻倍的开发者工具。开发者可以简单、快速的创建应用、项目开发、项目测试、发布部署等，实现项目的全生命周期管理。

Serverless 初始的痛点有很多，为什么阿里云却能把 Serverless 落地到各行各业？首先，阿里云提供了所有云厂商中最完整的 Serverless 产品矩阵，包括函数计算 FC、Serverless 应用引擎 SAE、面向容器编排的 ASK、以及面向容器实例的 ECI。丰富的

产品矩阵能够覆盖不同的场景，比如针对事件触发场景，函数计算提供了丰富的事件源集成能力和百毫秒伸缩的极致弹性；而针对微服务应用，Serverless 应用引擎能做到零代码改造，让微服务也能享受 Serverless 红利。其次， Serverless 是一个快速发展的领域，阿里云在不断拓展 Serverless 的产品边界。例如函数计算支持容器镜像、预付费模式、实例内并发执行多请求等多个业界首创的功能，彻底解决了冷启动带来的性能毛刺等 Serverless 难题，大大拓展了函数计算的应用场景。最后，阿里巴巴拥有非常丰富的业务场景，可以进一步打磨 Serverless 的落地实践。今年阿里巴巴的淘系、考拉、飞猪、高德等多个 BU 的双 11 核心业务场景均使用了阿里云函数计算，并顺利扛住了双 11 的高峰。

Serverless 引领下一个十年

“劳动生产力的最大激进，以及运用劳动时所表现的更大熟练、技巧和判断力，似乎都是劳动分工的结果” 这是摘自《国富论》的一段话，强调的是“劳动分工” 的利害关系，任何一个行业，市场规模越大，分工将会越细，这也是著名的“斯密定理”。同样，这一定理也适用于软件应用市场行业，随着传统行业进入了互联网化阶段，市场规模越来越大，劳动分工越来越细，物理机托管时代已经成为了历史，被成熟的 IaaS 层取代，随之而来的是容器服务，目前也已经是行业的标配。那么，接下来的技术十年是什么呢？答案是：Serverless，抹平了研发人员在预算、运维经验上的不足，在对抗业务洪峰的情况下，绝大多数研发也能轻易掌控处理，不仅极大地降低了研发技术门槛，同时大规模提升了研发效率，线上预警、流量观测等工具一应俱全，轻松做到了技术研发的免运维，可以说 Serverless 是更细粒度的分工，让业务开发者不再关注底层运维，只关注于业务创新，以此大大提高了劳动生产力，这就是“斯密定理”效应，也是 Serverless 成为未来必然趋势的内在原因。当下，整个云的产品体系已经 Serverless 化，70% 以上的产品都是 Serverless 形态。对象存储、消息中间件、API 网关、表格存储等 Serverless 产品已经被广大开发者熟知。

下一个十年， Serverless 将重新定义云的编程模型，重塑企业创新的方式。

第二章 技术能力突破

本章主要作者：庞永健、赵奕豪、张振、汤志敏、王思宇、黄涛、汪萌海、孙琦

注：作者姓名按文章顺序排列

七年零故障支撑双 11 的消息中间件 RocketMQ,2020 有何不同?

2020 年双十一交易峰值达到 58.3W 笔/秒，消息中间件 RocketMQ 继续 0 故障丝般顺滑地完美支持了整个集团大促的各类业务平稳。今年双十一大促中，消息中间件 RocketMQ 发生了以下几个方面的变化：

- 云原生化实践：完成运维层面的云原生化改造，实现 kubernetes 化。
- 性能优化：消息过滤优化交易集群性能提升 30%。
- 全新的消费模型：对于延迟敏感业务提供新的消费模式，降低因发布、重启等场景下导致的消费延迟。

云原生化实践

背景

Kubernetes 作为目前云原生化技术栈实践中重要的一环，其生态已经逐步建立并日益丰富。目前，服务于集团内部的 RocketMQ 集群拥有巨大的规模以及各种历史因素，因此在运维方面存在相当一部分痛点，我们希望能够通过云原生技术栈来尝试找到对应解决方案，并同时继续实现降本提效，达到无人值守的自动化运维。

消息中间件早在 2016 年，通过内部团队提供的中间件部署平台实现了容器化和自动化发布，整体的运维比 16 年前已经有了很大的提高，但是作为一个有状态的服务，在运维层面仍然存在较多的问题。

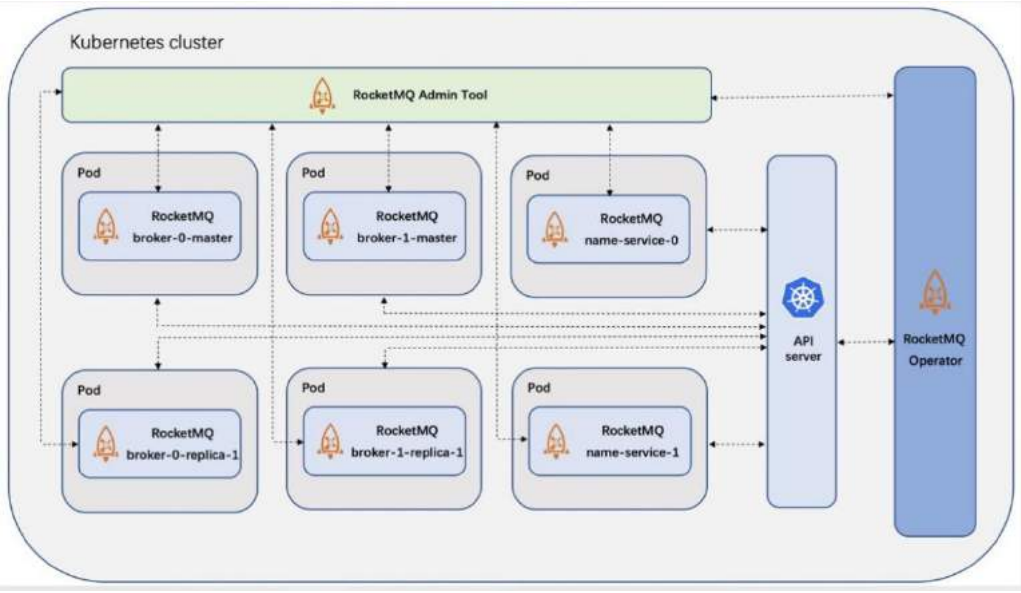
中间件部署平台帮我们完成了资源的申请，容器的创建、初始化、镜像安装等一系列的基础工作，但是因为中间件各个产品都有自己不同的部署逻辑，所以在应用的发布上，就是各应用自己的定制化了。中间件部署平台的开发也不完全了解集团内 RocketMQ 的部署过程是怎样的，因此在 16 年的时候，部署平台需要我们去亲自实现消息中间件的应用发布代码。

虽然部署平台大大提升了我们的运维效率，甚至还能实现一键发布，但是这样的方案也有不少的问题。比较明显的就是，当我们的发布逻辑有变化的时候，还需要去修改部署平台对应的代码，需要部署平台升级来支持我们，用最近比较流行的一个说法，就是相当不云原生。

同样在故障机替换、集群缩容等操作中，存在部分人工参与的工作，如切流，堆积数据的确认等。我们尝试过在部署平台中集成更多消息中间件自己的运维逻辑，不过在其他团队的工程里写自己的业务代码，确实也是一个不太友好的实现方案，因此我们希望通过 Kubernetes 来实现消息中间件自己的 operator。我们同样希望利用云化后云盘的多副本能力来降低我们的机器成本并降低主备运维的复杂程度。

经过一段时间的跟进与探讨，最终再次由内部团队承担了建设云原生应用运维平台的任务，并依托于中间件部署平台的经验，借助云原生技术栈，实现对有状态应用自动化运维的突破。

实现



整体的实现方案如上图所示，通过自定义的 CRD 对消息中间件的业务模型进行抽象，将原有的在中间件部署平台的业务发布部署逻辑下沉到消息中间件自己的 operator 中，托管在内部 Kubernetes 平台上。该平台负责所有的容器生产、初始化以及集团内一切线上环境的基线部署，屏蔽掉 Iaas 层的所有细节。

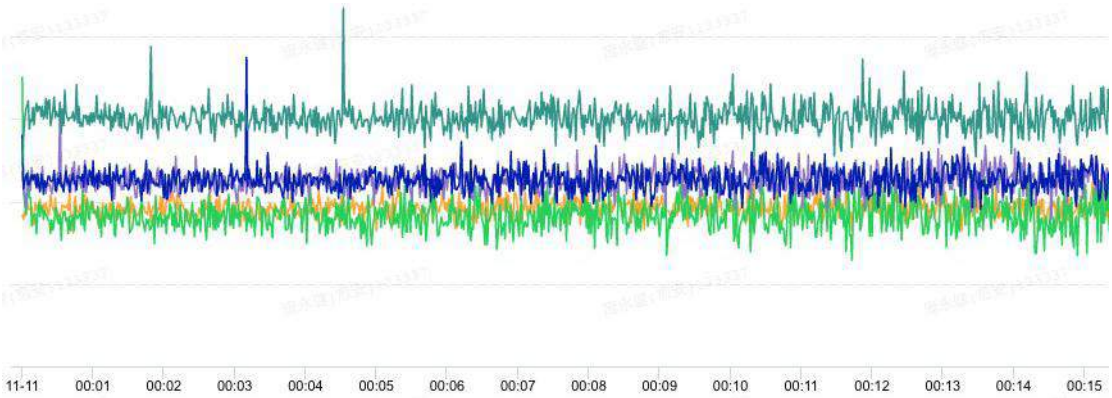
Operator 承担了所有的新建集群、扩容、缩容、迁移的全部逻辑，包括每个 pod 对应的 brokerName 自动生成、配置文件，根据集群不同功能而配置的各种开关，元数据的同步复制等等。同时之前一些人工的相关操作，比如切流时候的流量观察，下线前的堆积数据观察等也全部集成到了 operator 中。当我们有需求重新修改各种运维逻辑的时候，也再也不用去依赖通用的具体实现，修改自己的 operator 即可。

最后线上的实际部署情况去掉了图中的所有的 replica 备机。在 Kubernetes 的理念中，一个集群中每个实例的状态是一致的，没有依赖关系，而如果按照消息中间件原有的主备成对部署的方案，主备之间是有严格的对应关系，并且在上下线发布过程中有严格的顺序要求，这种部署模式在 Kubernetes 的体系下是并不提倡的。若依然采用以上老的架构方式，会导致实例控制的复杂性和不可控性，同时我们也希望能更多的遵循 Kubernetes 的运维理念。

云化后的 ECS 使用的是高速云盘，底层将对数据做了多备份，因此数据的可用性得到了保障。并且高速云盘在性能上完全满足 MQ 同步刷盘，因此，此时就可以把之前的异步刷盘改为同步，保证消息写入时的不丢失问题。云原生模式下，所有的实例环境均是一致性的，依托容器技术和 Kubernetes 的技术，可实现任何实例挂掉（包含宕机引起的挂掉），都能自动自愈，快速恢复。

解决了数据的可靠性和服务的可用性后，整个云原生化后的架构可以变得更加简单，只有 broker 的概念，再无主备之分。

大促验证



上图是 Kubernetes 上线后双十一大促当天的发送 RT 统计，可见大促期间的发送 RT 较为平稳，整体符合预期，云原生化实践完成了关键性的里程碑。

性能优化

背景

RocketMQ 至今已经连续七年 0 故障支持集团的双十一大促。自从 RocketMQ 诞生以来，为了能够完全承载包括集团业务中台交易消息等核心链路在内的各类关键业务，复用了原有的上层协议逻辑，使得各类业务方完全无感知的切换到 RocketMQ 上，并同时充分享受了更为稳定和强大的 RocketMQ 消息中间件的各类特性。

当前，申请订阅业务中台的核心交易消息的业务方一直都在不断持续增加，并且随着各类业务复杂度提升，业务方的消息订阅配置也变得更加复杂繁琐，从而使得交易集群的进行过滤的计算逻辑也变得更为复杂。这些业务方部分沿用旧的协议逻辑（Header 过滤），部分使用 RocketMQ 特有的 SQL 过滤。

主要成本

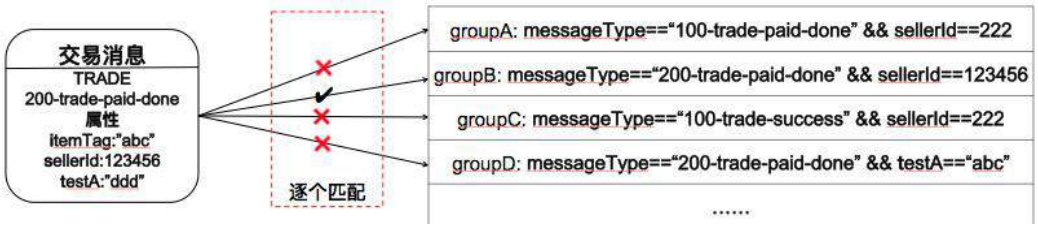
目前集团内部 RocketMQ 的大促机器成本绝大部分都是交易消息相关的集群，在双十一零点峰值期间，交易集群的峰值和交易峰值成正比，叠加每年新增的复杂订阅带来了额外 CPU 过滤计算逻辑，交易集群都是大促中机器成本增长最大的地方。

优化过程

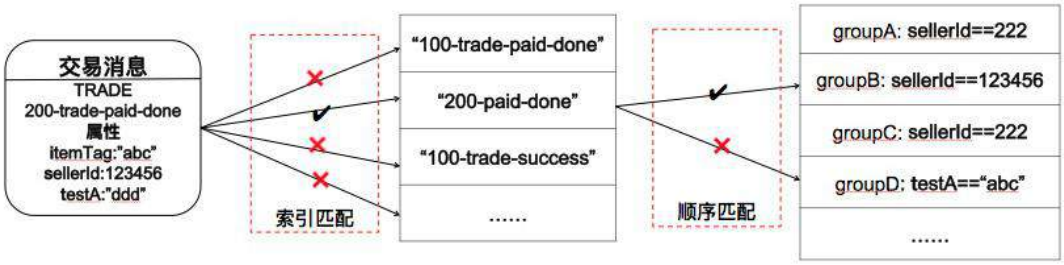
由于历史原因，大部分的业务方主要还是使用 Header 过滤，内部实现其实是 aviator 表达式（ https://github.com/killme2008/aviatorscript ）。仔细观察交易消息集群的业务方过滤表达式，可以发现绝大部分都指定类似 MessageType == xxxx 这样的条件。翻看 aviator 的源码可以发现这样的条件最终会调用 Java 的字符串比较 String.compareTo()。

由于交易消息包括大量不同业务的 MessageType，光是有记录的起码有几千个，随着交易业务流程复杂化，MessageType 的增长更是繁多。随着交易峰值的提高，交易消息峰值正比增长，叠加这部分更加复杂的过滤，持续增长的将来，交易集群的成本极可能和交易峰值指数增长，因此决心对这部分进行优化。

原有的过滤流程如下，每个交易消息需要逐个匹配不同 group 的订阅关系表达式，如果符合表达式，则选取对应的 group 的机器进行投递。如下图所示：



对此流程进行优化的思路需要一定的灵感，在这里借助数据库索引的思路：原有流程可以把所有订阅方的过滤表达式看作数据库的记录，每次消息过滤就相当于一个带有特定条件的数据库查询，把所有匹配查询（消息）的记录（过滤表达式）选取出来作为结果。为了加快查询结果，可以选择 MessageType 作为一个索引字段进行索引化，每次查询变为先匹配 MessageType 主索引，然后把匹配上主索引的记录再进行其它条件(如下图的 sellerId 和 testA)匹配，优化流程如下图所示：



以上优化流程确定后，要关注的技术点有两个：

1. 如何抽取每个表达式中的 MessageType 字段?
2. 如何对 MessageType 字段进行索引化?

对于技术点 1，需要针对 aviator 的编译流程进行 hook，深入 aviator 源码后，可以发现 aviator 的编译是典型的 Recursive descent（http://en.wikipedia.org/wiki/Recursive_descent_parser）。同时需要考虑到提取后父表达式的短路问题。

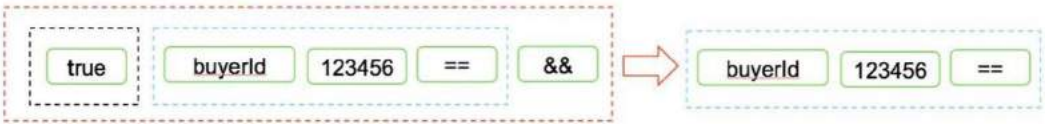
在编译过程中针对 messageType==XXX 这种类型进行提取后，把原有的 message==XXX 转变为 true/false 两种情况，然后针对 true、false 进行表达式的短路即可得出表达式优化提取后的情况。例如：

```
表达式：
messageType=='200-trade-paid-done' && buyerId==123456
提取为两个子表达式：
子表达式 1（messageType==200-trade-paid-done）：buyerId==123456
子表达式 2（messageType!=200-trade-paid-done）： false
```

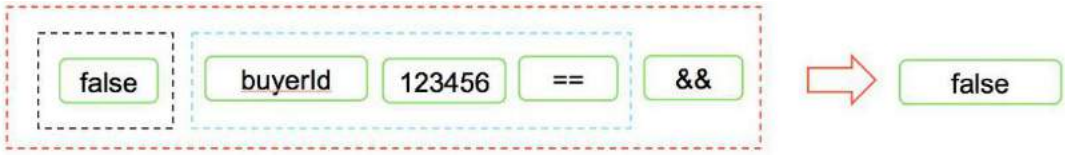
具体到 aviator 的实现里，表达式编译会把每个 token 构建一个 List，类似如下图所示(为方便理解，绿色方框的是 token，其它框表示表达式的具体条件组合)：

提取了 messageType，有两种情况：

情况一：messageType == '200-trade-paid-done'，则把之前 token 的位置合并成 true，然后进行表达式短路计算，最后优化成 buyerId==123456，具体如下：



情况二：messageType != '200-trade-paid-done'，则把之前 token 的位置合并成 false，表达式短路计算后，最后优化成 false，具体如下：



这样就完成 messageType 的提取。这里可能有人就有一个疑问，为什么要考虑到上面的情况二，messageType != '200-trade-paid-done'，这是因为必须要考虑到多个条件的时候，比如：

```
(messageType=='200-trade-paid-done' && buyerId==123456) || (messageType=='200-trade-success' && buyerId==3333)
```

就必须考虑到不等于的情况了。同理，如果考虑到多个表达式嵌套，需要逐步进行短路计算。但整体逻辑是类似的，这里就不再赘述。

说完技术点 1，我们继续关注技术点 2，考虑到高效过滤，直接使用 HashMap 结构进行索引化即可，即把 messageType 的值作为 HashMap 的 key，把提取后的子表达式作为 HashMap 的 value，这样每次过滤直接通过一次 hash 计算即可过滤掉绝大部分不适合的表达式，大大提高了过滤效率。

优化效果

该优化最主要降低了 CPU 计算逻辑，以三个交易集群为例，优化前后的平均 cpu 使用率对比如下（机器配置 24c128g）：

	TRADE	TRADE-SUB	TRADE-SUB2
优化前	42%	72%	70%
优化后	31%	40%	38%

三个交易集群 TRADE，TRADE-SUB，TRADE-SUB2 的消息峰值一致，但由于每个交易集群的业务订阅方复杂度不同（TRADE-SUB > TRADE-SUB2 > TRADE），理论上只要订阅关系约复杂优化效果越好，因此可以看到 TRADE-SUB 的优化最大，有 32% 的提升。

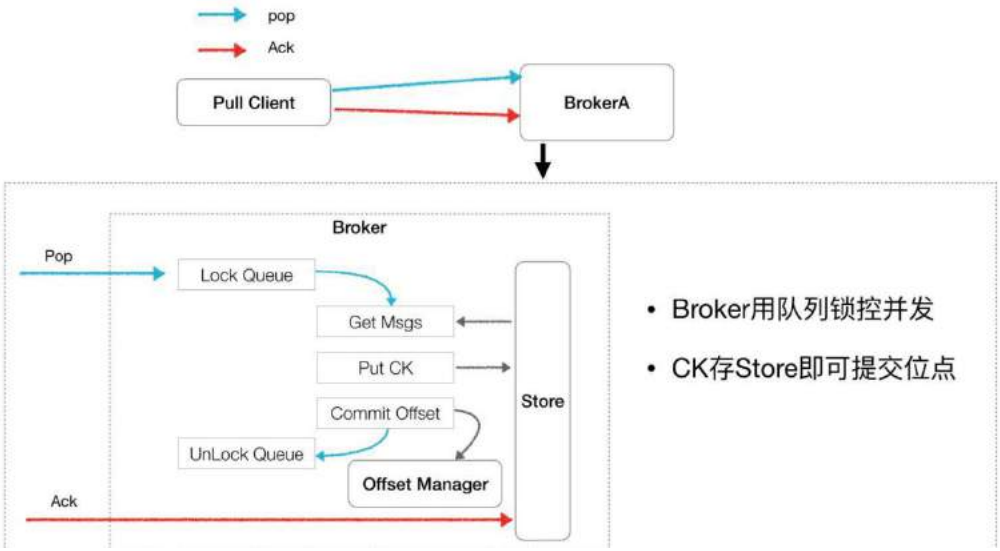
全新的消费模型 —— POP 消费

背景

RocketMQ 的 PULL 消费对于机器异常 hang 时并不十分友好。如果遇到客户端机器 hang 住，但处于半死不活的状态，与 broker 的心跳没有断掉的时候，客户端 rebalance 依然会分配消费队列到 hang 机器上，这样会导致消费堆积。另外类似还有服务端 Broker 发布时，也会由于客户端多次 rebalance 导致消费延迟影响等无法避免的问题。对于此，我们增加了一种新的消费模型——POP 消费，能够解决此类稳定性问题。

实现

POP 消费和原来 PULL 消费对比，最大的一点就是弱化了队列这个概念，PULL 消费需要客户端通过 rebalance 把 broker 的队列分配好，从而去消费分配到自己专属的队列，新的 POP 消费中，客户端的机器会直接到每个 broker 的队列进行请求消费，broker 会把消息分配返回给等待的机器。随后客户端消费结束后返回对应的 Ack 结果通知 broker，broker 再标记消息消费结果，如果超时没响应或者消费失败，再会进行重试。



POP 消费的架构图如上图所示。Broker 对于每次 POP 的请求，都会有以下三个操作：

1. 对应的队列进行加锁，然后从 store 层获取该队列的消息；
2. 然后写入 CK 消息，表明获取的消息要被 POP 消费；
3. 最后提交当前位点，并释放锁

CK 消息实际上是记录了 POP 消息具体位点的定时消息，当客户端超时没响应的时候，CK 消息就会重新被 broker 消费，然后把 CK 消息的位点的消息写入重试队列。如果 broker 收到客户端的消费结果的 Ack，删除对应的 CK 消息，然后根据具体结果判断是否需要重试。

从整体流程可见，POP 消费并不需要 reblance，可以避免 rebalance 带来的消费延时，同时客户端可以消费 broker 的所有队列，这样就可以避免机器 hang 而导致堆积的问题。

阿里 双 11 同款流控降级组件 Sentinel Go 正式 GA，助力云原生服务稳稳稳

前言

微服务的稳定性一直是开发者非常关注的话题。随着业务从单体架构向分布式架构演进以及部署方式的变化，服务之间的依赖关系变得越来越复杂，业务系统也面临着巨大的高可用挑战。

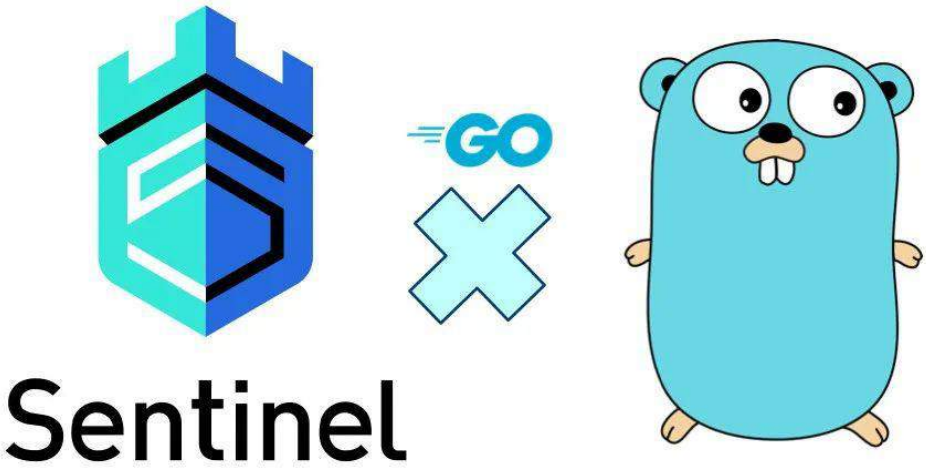
在生产环境中大家可能遇到过各种不稳定的情况，比如：

- 大促时瞬间洪峰流量导致系统超出最大负载，load 飙高，系统崩溃导致用户无法下单。
- “黑马”热点商品击穿缓存，DB 被打垮，挤占正常流量。
- 调用端被不稳定第三方服务拖垮，线程池被占满，调用堆积，导致整个调用链路卡死。

这些不稳定的场景可能会导致严重后果，但很多时候我们又容易忽视这些与流量/依赖相关的高可用防护。大家可能想问：如何预防这些不稳定因素带来的影响？如何针对流量进行高可用的防护？如何保障服务“稳如磐石”？这时候我们就要请出阿里双十一同款的高可用防护中间件 —— Sentinel。在今年刚刚过去的天猫 双 11 大促中，Sentinel 完美地保障了阿里成千上万服务 双 11 峰值流量的稳定性，同时 Sentinel Go 版本也在近期正式宣布 GA。下面我们来一起了解下 Sentinel Go 的核心场景以及社区在云原生方面的探索。

Sentinel 介绍

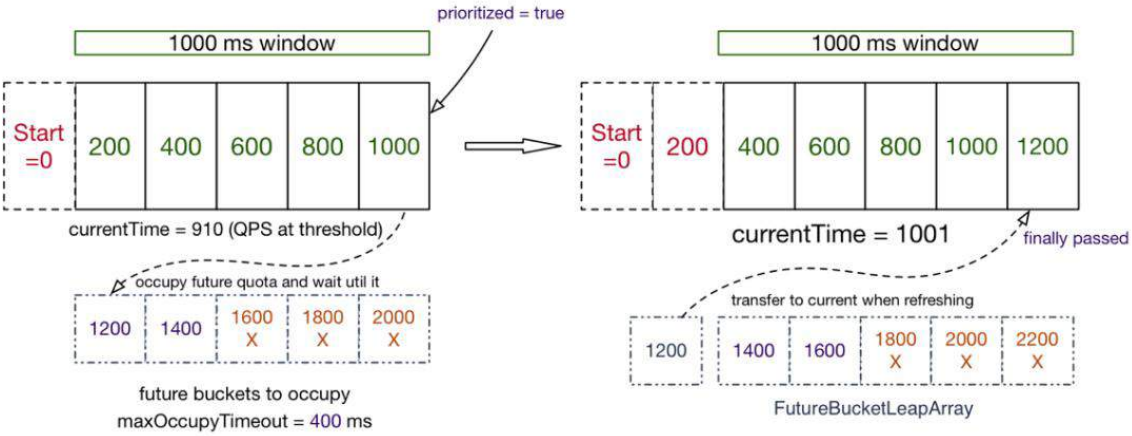
Sentinel 是阿里巴巴开源的，面向分布式服务架构的流量控制组件，主要以流量为切入点，从限流、流量整形、熔断降级、系统自适应保护等多个维度来帮助开发者保障微服务的稳定性。Sentinel 承接了阿里巴巴近 10 年的 双 11 大促流量的核心场景，例如秒杀、冷启动、消息削峰填谷、集群流量控制、实时熔断下游不可用服务等，是保障微服务高可用的利器，原生支持 Java/Go/C++ 等多种语言，并且提供 Istio/Envoy 全局流控支持来为 Service Mesh 提供高可用防护的能力。



今年年初，Sentinel 社区宣布了 Sentinel Go 版本的发布，为 Go 语言的微服务和基础组件提供高可用防护和容错能力的原生支持，标志着 Sentinel 朝着多元化与云原生迈出了新的一步。在这半年的时间内，社区推出了近 10 个版本，逐步对齐了核心高可用防护和容错能力，同时也在不断扩充开源生态，与 dubbo-go、蚂蚁 MOSN 等开源社区进行共建。

就在近期，Sentinel Go 1.0 GA 版本正式发布，标志着 Go 版本正式进入生产可用阶段。Sentinel Go 1.0 版本对齐了 Java 版本核心的高可用防护和容错能力，包括限流、流量整形、并发控制、熔断降级、系统自适应保护、热点防护等特性。同时 Go 版本已覆盖主流开源生态，提供了 Gin、gRPC、go-micro、dubbo-go 等常用微服务框架的适配，并提供了 etcd、Nacos、Consul 等动态数据源扩展支持。Sentinel Go 也在朝着云原生的方向不断演进，1.0 版本中也进行了一些云原生方面的探索，包括 Kubernetes CRD data-source, Kubernetes HPA 等。对于 Sentinel Go 版本而言，我们期望的流控场景并不局限于微服务应用本身。云原生基础组件中 Go 语言生态占比较高，而这些云原生组件很多时候又缺乏细粒度、自适应的保护与容错机制，这时候就可以结合组件的一些扩展机制，利用 Sentinel Go 来保护自身的稳定性。

Sentinel 底层通过高性能的滑动窗口进行秒级调用指标统计，结合 token bucket, leaky bucket 和自适应流控算法来透出核心的高可用防护能力。



那么我们如何利用 Sentinel Go 来保证我们微服务的稳定性？下面我们来看几个典型的应用场景。

高可用防护的核心场景

（一）流量控制与调配

流量是非常随机性的、不可预测的。前一秒可能还风平浪静，后一秒可能就出现流量洪峰了（例如 双11 零点的场景）。然而我们系统的容量总是有限的，如果突然而来的流量

超过了系统的承受能力，就可能会导致请求处理不过来，堆积的请求处理缓慢，CPU/Load 飙高，最后导致系统崩溃。因此，我们需要针对这种突发的流量来进行限制，在尽可能处理请求的同时来保障服务不被打垮，这就是流量控制。流量控制的场景是非常通用的，像脉冲流量类的场景都是适用的。

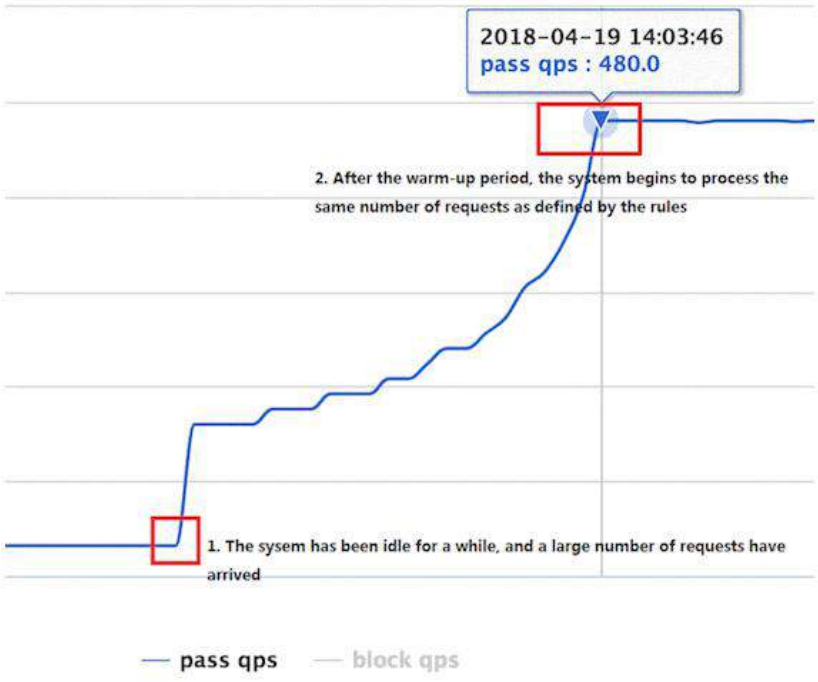
通常在 Web 入口或服务提供方（Service Provider）的场景下，我们需要保护服务提供方自身不被流量洪峰打垮。这时候通常根据服务提供方的服务能力进行流量控制，或针对特定的服务调用方进行限制。我们可以结合前期压测评估核心接口的承受能力，配置 QPS 模式的流控规则，当每秒的请求量超过设定的阈值时，会自动拒绝多余的请求。
下面是最简单的一个 Sentinel 限流规则的配置示例：

```
    _, err = flow.LoadRules([]*flow.Rule{
    {
            Resource:               "some-service", // 埋点资源名
    Count:                  10, // 阈值为 10，默认为秒级维度统计，即该请求单机每秒不超过 10 次
            ControlBehavior:    flow.Reject, // 控制效果为直接拒绝，不控制请求之间的时间间隔，不排队
        },
    })
```

（二）Warm-Up 预热流控

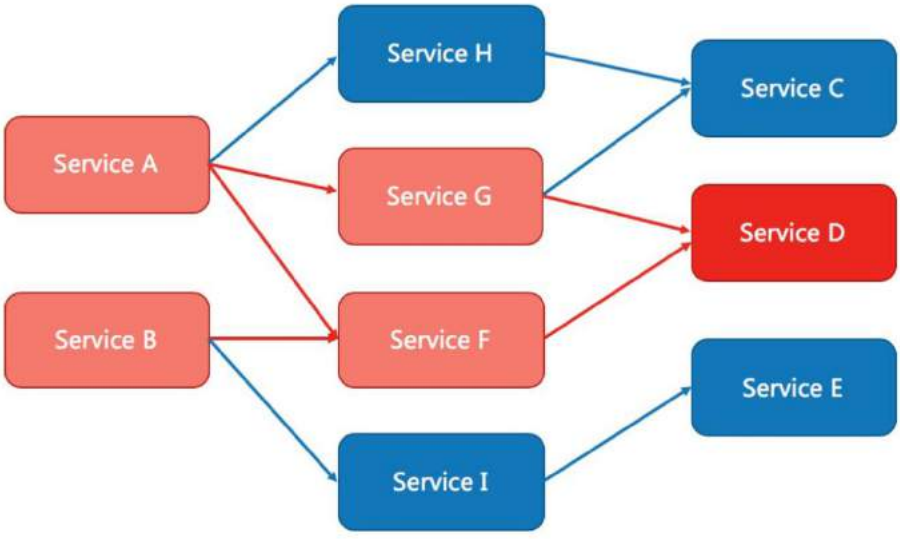
当系统长期处于低水位的情况下，流量突然增加时，直接把系统拉升到高水位可能瞬间把系统压垮。比如刚启动的服务，数据库连接池可能还未初始化，缓存也处于空的状态，这时候激增的流量非常容易导致服务崩溃。如果采用传统的限流模式，不加以平滑/削峰限制，其实也是有被打挂的风险的（比如一瞬间并发很高）。

针对这种场景，我们就可以利用 Sentinel 的 Warm-Up 流控模式，控制通过的流量缓慢增加，在一定时间内逐渐增加到阈值上限，而不是在一瞬间全部放行，同时结合请求间隔控制+排队的控制效果 来防止大量请求都在同一时刻被处理。这样可以给冷系统一个预热的时间，避免冷系统被压垮。



（三）并发控制与熔断降级

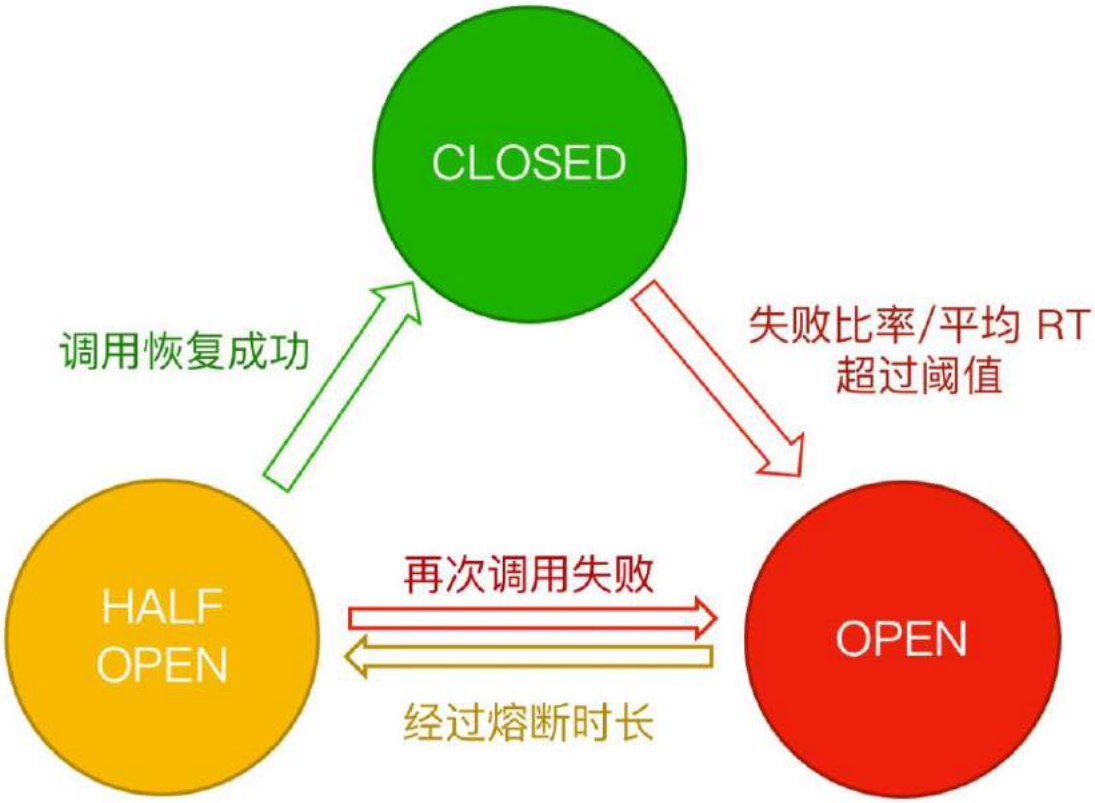
一个服务常常会调用别的模块，可能是另外的一个远程服务、数据库，或者第三方 API 等。例如，支付的时候，可能需要远程调用银联提供的 API；查询某个商品的价格，可能需要进行数据库查询。然而，这个被依赖服务的稳定性是不能保证的。如果依赖的服务出现了不稳定的情况，请求的响应时间变长，那么调用服务的方法的响应时间也会变长，线程会产生堆积，最终可能耗尽业务自身的线程池，服务本身也变得不可用。



现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。Sentinel Go 提供以下的能力避免慢调用等不稳定因素造成不可用:

- 并发控制（isolation 模块）：作为一种轻量级隔离的手段，控制某些调用的并发数（即正在进行的数目），防止过多的慢调用挤占正常的调用。
- 熔断降级（circuitbreaker 模块）：对不稳定的弱依赖调用进行自动熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。

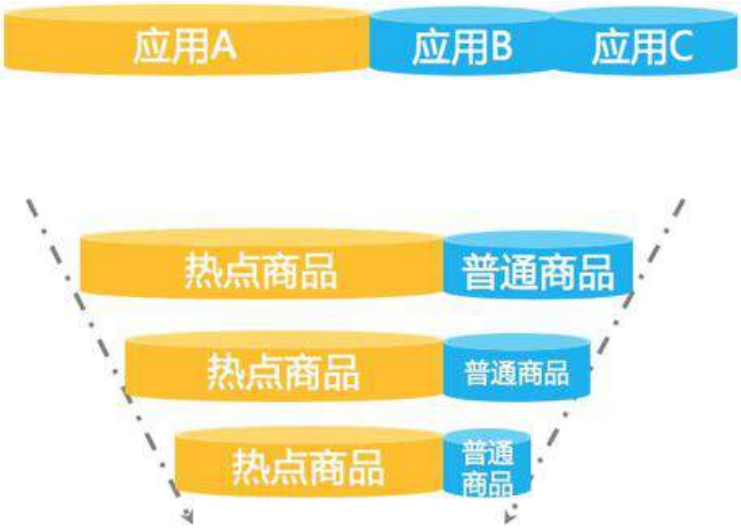
Sentinel Go 熔断降级特性基于熔断器模式的思想，在服务出现不稳定因素（如响应时间变长，错误率上升）的时候暂时切断服务的调用，等待一段时间再进行尝试。一方面防止给不稳定服务“雪上加霜”，另一方面保护服务的调用方不被拖垮。Sentinel 支持两种熔断策略：基于响应时间（慢调用比例）和基于错误（错误比例/错误数），可以有效地针对各种不稳定的场景进行防护。



注意熔断器模式一般适用于弱依赖调用，即降级后不影响业务主流程，开发者需要设计好降级后的 fallback 逻辑和返回值。另外需要注意的是，即使服务调用方引入了熔断降级机制，我们还是需要在 HTTP 或 RPC 客户端配置请求超时时间，来做一个兜底的防护。

（四）热点防护

流量是随机的，不可预测的。为了防止被大流量打垮，我们通常会对核心接口配置限流规则，但有的场景下配置普通的流控规则是不够的。我们来看这样一种场景——大促峰值的时候，总是会有不少“热点”商品，这些热点商品的瞬时访问量非常高。一般情况下，我们可以事先预测一波热点商品，并对这些商品信息进行缓存“预热”，以便在出现大量访问时可以快速返回而不会都打到 DB 上。但每次大促都会涌现出一些“黑马”商品，这些“黑马”商品是我们无法事先预测的，没有被预热。当这些“黑马”商品访问量激增时，大量的请求会击穿缓存，直接打到 DB 层，导致 DB 访问缓慢，挤占正常商品请求的资源池，最后可能会导致系统挂掉。这时候，利用 Sentinel 的热点参数流量控制，自动识别热点参数并控制每个热点值的访问 QPS 或并发量，可以有效地防止过“热”的参数访问挤占正常的调用资源。



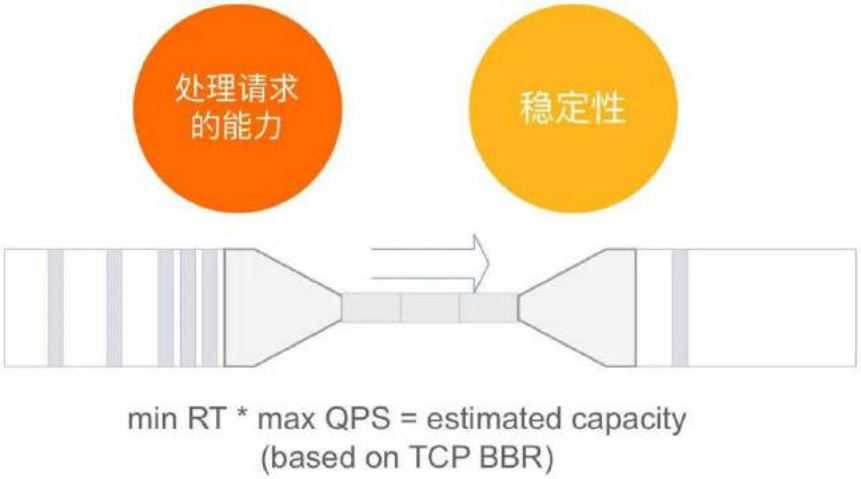
再比如有的场景下我们希望限制每个用户调用某个 API 的频率，将 API 名称 +userId 作为埋点资源名显然是不合适的。这时候我们可以在给 API 埋点的时候通过 WithArgs(xxx) 将 userId 作为参数传入到 API 埋点中，然后配置热点规则即可针对每个用户分别限制调用频率；同时，Sentinel 也支持针对某些具体值单独配置限流值，进行精细化流控。像其他规则一样，热点流控规则同样支持通过动态数据源进行动态配置。

Sentinel Go 提供的 RPC 框架整合模块（如 Dubbo、gRPC）均会自动将 RPC 调用的参数列表附带在埋点中，用户可以直接针对相应的参数位置配置热点流控规则。注意如果需要配置具体值限流，受类型系统限制，目前仅支持基本类型和 string 类型。

Sentinel Go 的热点流量控制基于缓存淘汰机制+令牌桶机制实现。Sentinel 通过淘汰机制（如 LRU、LFU、ARC 策略等）来识别热点参数，通过令牌桶机制来控制每个热点参数的访问量。目前的 Sentinel Go 版本采用 LRU 策略统计热点参数，社区也已有贡献者提交了优化淘汰机制的 PR，在后续的版本中社区会引入更多的缓存淘汰机制来适配不同的场景。

（五）系统自适应保护

有了以上的流量防护场景，是不是就万事无忧了呢？其实不是的，很多时候我们无法事先就准确评估某个接口的准确容量，甚至无法预知核心接口的流量特征（如是否有脉冲情况），这时候靠事先配置的规则可能无法有效地保护当前服务节点；一些情况下我们可能突然发现机器的 Load 和 CPU usage 等开始飚高，但却没有办法很快的确认到是什么原因造成的，也来不及处理异常。这个时候我们其实需要做的是快速止损，先通过一些自动化的兜底防护手段，将濒临崩溃的微服务“拉”回来。针对这些情况，Sentinel Go 提供了一种系统自适应保护规则，结合系统指标和服务容量，自适应动态调整流量。



Sentinel 系统自适应保护策略借鉴了 TCP BBR 算法的思想，结合系统的 Load、CPU 使用率以及服务的入口 QPS、响应时间和并发量等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐

量的同时保证系统整体的稳定性。系统规则可以作为整个服务的一个兜底防护策略，保障服务不挂，对 CPU 密集型的场景会有比较好的效果。同时，社区也在结合自动化控制理论和强化学习等手段，持续完善自适应流控的效果和适用场景。在未来的版本中，社区会也推出更多试验性的自适应流控策略，来满足更多的可用性场景。

云原生探索

云原生是 Sentinel Go 版本演进最为重要的一环。在 GA 的过程中，Sentinel Go 社区也在 Kubernetes 和 Service Mesh 等场景下进行了一些探索。

（一）Kubernetes CRD data-source

在生产环境中我们一般都需要通过配置中心来动态管理各种规则配置。在 Kubernetes 集群中，我们可以天然利用 Kubernetes CRD 的方式来管理应用的 Sentinel 规则。在 Go 1.0.0 版本中社区提供了基本的 Sentinel 规则 CRD 抽象以及相应的数据源实现。用户只需要先导入 Sentinel 规则 CRD 定义文件，接入 Sentinel 时注册对应的 data-source，然后按照 CRD 定义的格式编写 YAML 配置并 kubectl apply 到对应的 namespace 下即可实现动态配置规则。以下是一个流控规则的示例：

```yaml
apiVersion: datasource.sentinel.io/v1alpha1
kind: FlowRules
metadata:
  name: foo-sentinel-flow-rules
spec:
  rules:
    - resource: simple-resource
      threshold: 500
    - resource: something-to-smooth
      threshold: 100
      controlBehavior: Throttling
      maxQueueingTimeMs: 500
    - resource: something-to-warmup
      threshold: 200
      tokenCalculateStrategy: WarmUp
      controlBehavior: Reject
      warmUpPeriodSec: 30
      warmUpColdFactor: 3
```

Kubernetes CRD data-source 模块地址：
https://github.com/sentinel-group/sentinel-go-datasource-k8s-crd

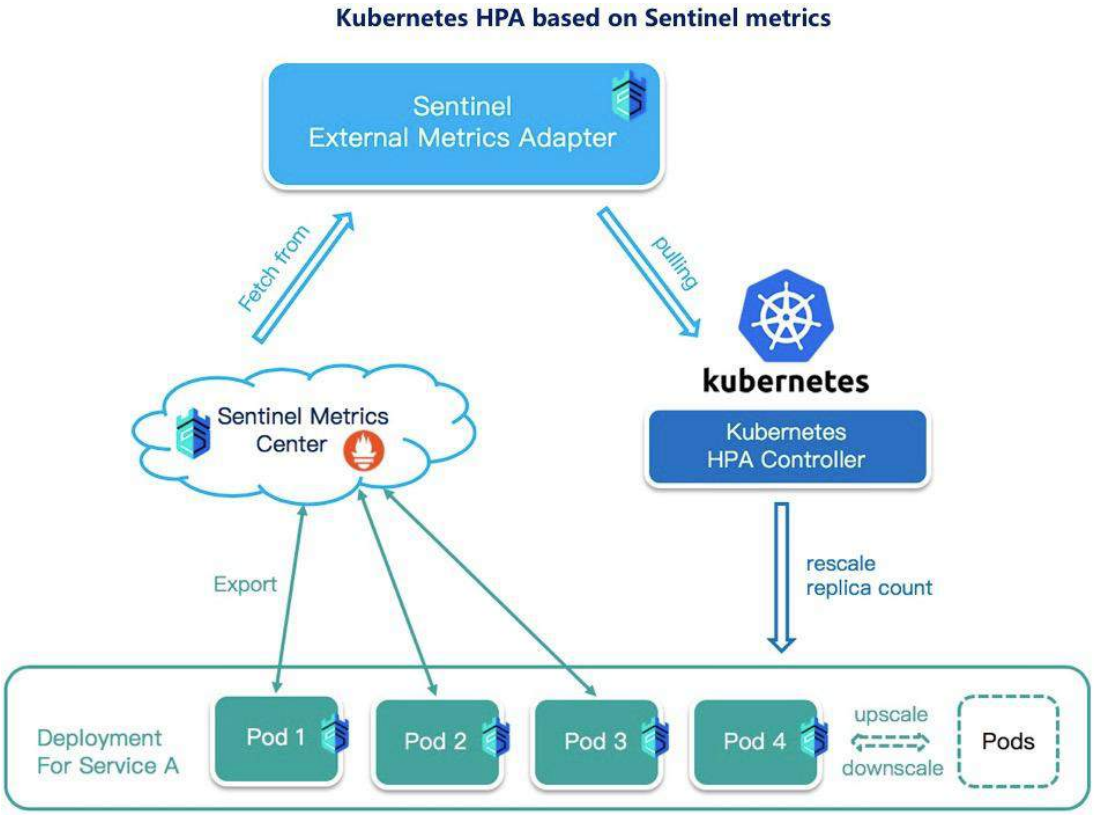
后续社区会进一步完善 Rule CRD 定义并与其它社区一起探讨高可用防护相关的标准抽象。

（二）Service Mesh

Service Mesh 是微服务向云原生演进的趋势之一。在 Service Mesh 架构下，一些服务治理和策略控制的能力都逐渐下沉到了 data plane 层。去年 Sentinel 社区在 Java 1.7.0 版本里面做了一些尝试，提供了 Envoy Global Rate Limiting gRPC Service 的实现 ——Sentinel RLS token server，借助 Sentinel 集群限流 token server 来为 Envoy 服务网格提供集群流量控制的能力。今年随着 Sentinel Go 版本的诞生，社区与更多的 Service Mesh 产品开展合作、整合。我们与蚂蚁的 MOSN 社区进行共建，在 MOSN Mesh 中原生支持了 Sentinel Go 的流控降级能力，同时也已在蚂蚁内部落地。社区也在探索更为通用的方案，如利用 Istio 的 Envoy WASM 扩展机制实现 Sentinel 插件，让 Istio/Envoy 服务网格可以借助 Sentinel 原生的流控降级与自适应保护的能力来保障整个集群服务的稳定性。

（三）Kubernetes HPA based on Sentinel metrics

保障服务稳定性的方法多种多样，除了各种规则对流量进行“控制”之外，“弹性”也是一种思路。对于部署在 Kubernetes 中的应用来说，可以利用 Kubernetes HPA 能力进行对服务进行水平扩缩容。HPA 默认支持多种系统指标，并且支持自定义指标统计。目前我们已经在阿里云 Kubernetes 容器服务上结合AHAS Sentinel支持基于服务的平均 QPS、响应时间等作为条件进行弹性伸缩。社区也正在这一块做一些尝试，将一些 Sentinel 的服务级别的指标统计（通过量，拒绝量，响应时间等）通过 Prometheus 或 OpenTelemetry 等标准方式透出，并适配到 Kubernetes HPA 中。



当然基于 Sentinel 的弹性方案不是万灵药，它只适用于一些特定的场景，比如适用于启动速度快的无状态服务（Serverless 场景）。对于启动较慢的服务，或非本服务容量问题的场景（如依赖的 DB 容量不够），弹性的方案不能很好地解决稳定性的问题，甚至可能会加剧服务的恶化。

Let's start hacking!

了解了以上的高可用防护的场景，以及 Sentinel 在云原生方向的一些探索，相信大家对微服务容错与稳定性的手段有了新的体会。欢迎大家动手玩一下 demo，将微服务接入 Sentinel 来享受高可用防护和容错的能力，让服务“稳如磐石”。同时 Sentinel Go 1.0 GA 版本的发布离不开社区的贡献，感谢所有参与贡献的小伙伴们。

本次 GA 我们也新加入了两位给力的 committer ——@sanxun0325 和 @luckyxiaoqiang，两位在 1.0 版本的演进带来了 Warm Up 流控、Nacos 动态数据源以及一系列功能改进和性能优化，非常积极地帮助社区答疑解惑以及 review 代码。恭喜两位！社区在未来版本中也会朝着云原生和自适应智能化的方向不断探索和演进，也欢迎更多的同学加入贡献小组，一起参与 Sentinel 未来的演进，创造无限可能。我们鼓励任何形式的贡献，包括但不限于：

- bug fix
- new features/improvements
- dashboard
- document/website
- test cases

开发者可以在 GitHub 上面的 good first issue 列表上挑选感兴趣的 issue 来参与讨论和贡献。我们会重点关注积极参与贡献的开发者，核心贡献者会提名为 Committer，一起主导社区的发展。我们也欢迎大家有任何问题和建议，都可以通过 GitHub issue、Gitter 或钉钉群（群号：30150716）等渠道进行交流。Now start hacking!

- Sentinel Go repo:https://github.com/alibaba/sentinel-golang
- 企业用户欢迎进行登记：https://github.com/alibaba/Sentinel/issues/18
- Sentinel 阿里云企业版：https://ahas.console.aliyun.com/

「更高更快更稳」，看阿里巴巴如何修炼容器服务「内外功」

11 月 11 日零点刚过 26 秒，阿里云再一次抗住了全球最大的流量洪峰。今年 双 11 是阿里巴巴核心系统全面云原生化的一年，相比去年核心系统的上云，云原生化不仅让阿里享受到了云计算技术成本优化的红利，也让阿里的业务最大化获得云的弹性、效率和稳定性等价值。

为应对 双 11，阿里云原生面临怎样的挑战?

为了支持阿里这样大规模业务的云原生化，阿里云原生面临怎么样的挑战呢?

（一）集群多、规模大

基于对业务稳定性和系统性能等方面的综合考虑，大型企业往往需要将业务集群部署到多个地域，在这样的场景下，支撑多集群部署的容器服务能力非常必要。同时，为了简化多集群管理的复杂度，以及为不能实现跨集群服务发现的业务提供支持，还需要关注容器服务中单个集群对大规模节点的管理能力。另外，大型企业的业务复杂多样，因此一个集群内往往需要部署丰富的组件，不仅包括主要的 Master 组件， 还需要部署业务方定制的 Operator 等。集群多、规模大，再加上单个集群内组件众多， 容器服务的性能、可扩展性、可运维性都面临着很大的挑战。

（二）变化快、难预期

市场瞬息万变，企业，特别是互联网企业，如果仅凭借经验、依靠规划来应对市场变化，越来越难以支撑业务发展，往往需要企业快速地进行业务迭代、部署调整以应对市场的变化。这对为业务提供应用交付快速支持、弹性伸缩性能和快速响应支撑的容器服务提出了很大的要求。

（三）变更多、风险大

企业 IT 系统的云原生化过程不仅要面临一次性的云迁移和云原生改造成本，还将持续应对由于业务迭代和基础设施变更、人员流动带来的风险。而业务的迭代、基础设施的变更会无法避免地为系统引入缺陷，严重时甚至造成故障，影响业务正常运行。最后，这些风险都可能会随着人员的流动进一步加剧。

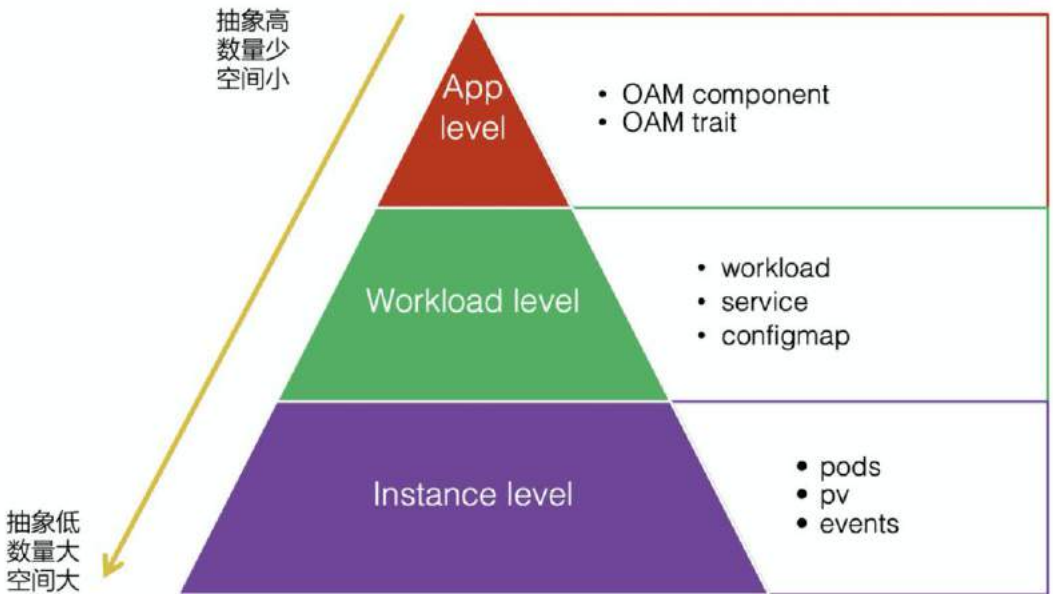
阿里云容器服务大规模实践

（一）高扩展性

为了提高容器服务的可扩展性，需要自底向上、联动应用和服务一起优化。

1）接口最小化

针对上层 PaaS，容器服务依托 OAM（Open Application Model）和 OpenKruise Workload 提供了丰富的应用交付能力抽象。对于 PaaS 层来说，只需要读取汇总的应用部署统计数值即可，极大减少了上层系统需要批量查询 pod、event 等信息的工作量，进而减小了对容器服务的管控压力；同时，通过把数量多、占用空间大的 pod 及 events 信息转存到数据库中， 并根据数据库中的内容提供一个统一的、可内嵌的部署状态查看和诊断页面的方式，可以使 PaaS 层避免直接访问容器服务来实现相关功能。



2）分化而治之

“分化而治之”是指要对业务做出合理的划分，避免因为所有业务和应用都集中在少数几个命名空间中，导致容器服务管控（控制器和 APIServer）在查询命名空间下所有资源时产生过大消耗的情况。目前在阿里内部最广泛的实践方式是使用“应用名”做为命名空间。一方面应用是业务交付的基本单位，且不受组织调整的影响；另一方面，容器服务的组件以及业务自己的 Operator，在处理时往往会 list 命名空间下的资源，而命名空间默认在控制器和 APIServer 中都实现有索引，如果使用应用作为命名空间可以利用默认的索引加速查询操作。

3）苦修内外功

对于容器服务的核心管控，需要扎实的内功做基础。针对大规模的使用场景，阿里云的容器服务进行了大量的组件优化，比如通过索引、Watch 书签等的方式，避免直接穿透 APIServer 访问底层存储 ETCD，并通过优化 ETCD 空闲页面分配算法、分离 event 和 lease 存储至独立 ETCD 的方法，提升 ETCD 本身的存储能力，其中不少已经反馈给了社区，极大降低了 ETCD 处理读请求的压力。详情可查看：https://4m.cn/JsXsU。其次，对于核心管控本身，要做好保护的外功。特别是安全防护，需要在平时就做好预案，并且常态化地执行演练。例如， 对于容器服务 APIServer， 需要保证任何时候的 APIServer 都能保持可用性。除了常见的 HA 方案外， 还需要保证 APIServer 不受异常的 operator 和 daemonset 等组件的影响。为了保证 APIServer 的鲁棒性，可以利用官方提供的 max-requests-inflight 和 max-mutating-requests-inflight 来实现， 在紧急情况下阿里云还提供了动态修改 infight 值配置的功能，方便在紧急情况下不需要重启 APIServer 就可以应用新的配置进行限流。对于最核心的 ETCD，要做好数据的备份。考虑到数据备份的及时性，不仅要进行数据定期的冷备，还需要实时地进行数据的异地热备，并常态化地执行数据备份、灰度的演练，保证可用性。

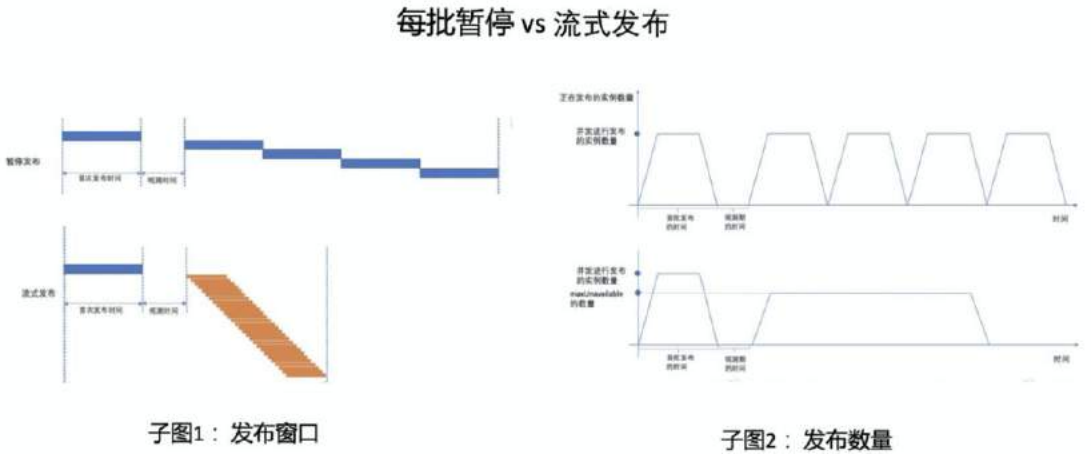
（二）快速

在应对业务变化多、基础设施变化多带来的不可预期问题，可采取以下方式。

1）负载自动化

为了高效地进行应用交付，研发需要权衡交付效率和业务稳定性。阿里大规模地使用 OpenKruise 进行应用的交付，其中 cloneset 覆盖了阿里上百万的容器。 在 cloneset 中可以通过 partition 来控制暂停发布从而进行业务观察，通过 maxunavailable 来控

制发布的并发度。通过综合设置 partition 和 maxunavailable 可以实现复杂的发布策略。实际上，大多数情况下 PaaS 层在设计分批发布的功能时，往往选取了每批暂停的方式（仅通过 cloneset partition 字段来控制分批），如下图。然而，每批暂停的方式往往因为应用每批中个别机器的问题而产生卡单的问题，严重影响发布效率。



子图1： 发布窗口　　子图2： 发布数量

因此推荐使用流式发布的配置：

```yaml
apiVersion: apps.kruise.io/v1alpha1
kind: CloneSet
spec:
  updateStrategy:
  # 首批partition设置，  非首批置0
    partition: 0
  # 控制发布的并发度，  实现为1/分批数
    maxUnavailable: 20%
```

2）以快制动

为了应对突发的业务流量， 需要能够快速的进行应用的部署，并从多方面保障紧急场景的快速扩容。

一方面，通过推进业务使用方的 CPUShare 化，让应用能够原地利用节点额外计算资源，从而给紧急扩容争取更多的反应时间。

其次，通过镜像预热的方式来预热基础镜像，通过 P2P 和 CDN 的技术加速大规模的镜像分发，通过按需下载容器启动实际需要的镜像数据，使业务镜像下载时间极大地降低。

3）打好提前量

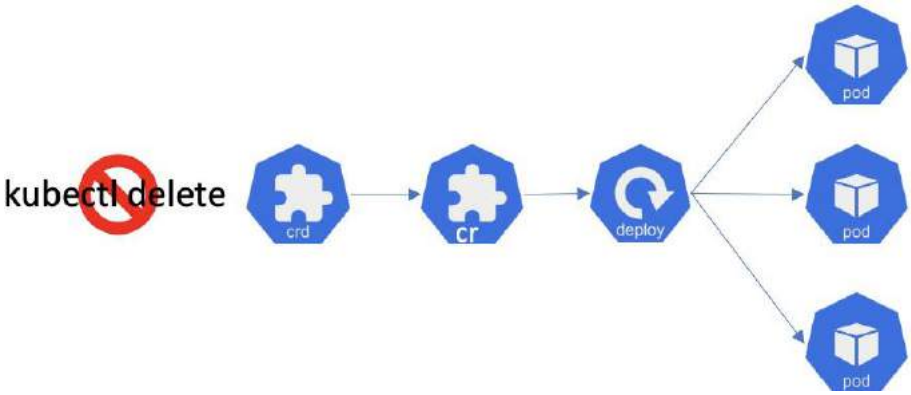
俗话说，"不打无准备之仗" 。要实现高效率的部署，做好准备是必需的。 首先是对于资源的准备， 如果集群内没有为服务准备好一定的容量，或者没有为集群准备好额外节点额度的话，就无法在必要时实现弹性。因为阿里不同业务有着不同的流量峰值，我们会结合实际情况定期对部分业务缩容，并对另外一种业务进行扩容的方式实现计划资源的伸缩。 当然，寻找这样可以匹配较好的业务组会比较困难，对于采用函数等 Serverless 形态的应用， 阿里构建一个公共预扩容的 Pod 池，使得不同业务紧急扩容时能够完全规避调度、网络和储存的开销， 达到极致的扩容速度。

（三）稳定

为了确保使用容器服务的业务稳定，阿里在具体实践中遵循了以下几个原则。

1）信任最小化

俗话说，"常在河边走，哪有不湿鞋"。为了确保频繁进行集群运维工作的同学不因为疏忽而犯错，就要保证业务可操作的权限最小化，对于授权的写和删除操作，还要增加额外的保护。近一步来讲，为了防止容器服务用户的误操作，我们对 Namespace、CRD 和 Workload 的资源都增加了级联删除的保护，避免用户因为误删除还在运行 Pod 的 Namespace、CRD 和 Workload 而引发灾难性的后果。下图展示了删除一个 CRD 可能造成的级联删除故障，实际应用中，许多 operator 中都会设置 CR 为关联 Workload 的 Owner， 因此一旦删除了 CRD（例如非预期的 Operator 升级操作），极有可能会导致级联删除关联的所有 Pod，引起故障。



另外， 对于业务运行依赖的如日志上传、微服务调用、安全审计等基础设功能，需要进行资源的隔离。因此，阿里在对应用进行大量的轻量化容器改造过程中，采取了把基础设施功能从应用的富容器中剥离到 Sidecar 或者 Daemonset 中的方式，并对 Sidecar 或者 Daemon 的容器进行资源的隔离， 保证即使基础设施功能发生内存泄露等异常也不会直接影响业务的正常运行。

2）默认稳定性

指保证所有应用都具备基本的稳定性保障配置，包括默认的调度打散策略、Pod 中断预算、应用负载发布最大不可用数量，让稳定性成为像水、电、煤一样的基础设施。这样能够避免应用因为宿主机故障、运维操作、应用发布操作导致服务的完全不可用。保障可以通过 webhook 或者通过全局的应用交付模板来实现，应用 PaaS 也可以根据业务的实际要求来定制。

3）规范化应用

在进行云原生改造时，需要制定启停脚本、可用和探活探针规范，帮助业务把自愈能力内置到应用中去。这包括推动应用配置相应的存活探针，保证应用在异常退出后能够被自动拉起；保证应用启动和退出时执行优雅上下线的操作等。配合这些规范，还需要设置相关探针的准入、监测机制，防止业务研发同学在对 K8s 机制不完全了解的情况下编写错误的探针。我们常常见到很多研发同学直接复用已有的健康检查脚本来作为探活探针，但是这些脚本往往存在调用开销大（例如执行了解压缩）、存在副作用（例如会修改业务流量开启状态）、以及执行不稳定（例如调用涉及下游服务）的问题，这些对业务的正常运行都会产生非常大的干扰，甚至引发故障。

4）集中化处理

对于探活失败的自动重启、问题节点的驱逐操作， 阿里云容器服务把 Kubelet 自主执行的自愈操作，改为了中心控制器集中触发，从而可以利用应用级别的可用度数据实现限流、熔断等安全防护措施。这样，即使发生了业务错配探活脚本或者运维误操作执行批量驱逐等操作状况，业务同样能得到保护；而在大促峰值等特殊的业务场景下，可以针对具体需求设计相应的预案，关闭相应探活、重启、驱逐等操作，避免在业务峰值时因为探活等活动引起应用资源使用的波动，保证业务短期的极致确定性要求。

5）变更三板斧

首先，要保证容器服务自身的变更可观测、可灰度、可回滚。对于 Controller 和 Webhook 这类的中心管控组件，一般可以通过集群来进行灰度，但如果涉及的改动风险过大，甚至还需要进行 Namespace 级别细粒度的灰度；由于阿里部分容器服务是在节点上或者 Pod 的 Sidecar 中运行的，而官方 K8s 中欠缺对于节点上 Pod 和 Sidecar 中容器的灰度发布支持，因此阿里使用了 OpenKruise 中的 Advance Daemonset 和 Sidecarset 来执行相关的发布。

使用阿里云容器服务

轻松构建大规模容器平台

阿里云容器服务 ACK（Alibaba Cloud Container Service for Kubernetes）是全球首批通过 Kubernetes 一致性认证的服务平台，提供高性能的容器应用管理服务，支持企业级 Kubernetes 容器化应用的生命周期管理。ACK 在阿里集团内作为核心的容器化基础设施，有丰富的应用场景和经验积累，包括电商、实时音视频、数据库、消息中间件、人工智能等场景，支撑广泛的内外部客户参加 双 11 活动；同时，容器服务将阿里内部各种大规模场景的经验和能力融入产品，向公有云客户开放，提升了更加丰富的功能和更加突出的稳定性，容器服务连续多年保持国内容器市场份额第一。

在过去的一年，ACK 进行了积极的技术升级，针对阿里的大规模实践和企业的丰富生产实践，进一步增强了可靠性、安全性，并且提供可赔付的 SLA 的 Kubernetes 集群 - ACKPro 版。ACK Pro 版集群是在原 ACK 托管版集群的基础上发展而来的集群类型，继承了原托管版集群的所有优势，例如 Master 节点托管、Master 节点高可用等。同时，相比原托管版进一步提升了集群的可靠性、安全性和调度性能，并且支持赔付标准的 SLA，适合生产环境下有着大规模业务，对稳定性和安全性有高要求的企业客户。

9 月底，阿里云成为率先通过信通院容器规模化性能测试的云服务商，获得最高级别认证—“卓越”级别，并首先成功实现以单集群 1 万节点 1 百万 Pod 的规模突破，构建了国内最大规模容器集群，引领国内容器落地的技术风向标。此次测评范围涉及：资源调度效率、满负载压力测试、长稳测试、扩展效率测试、网络存储性能测试、采集效率测试等，

客观真实反映容器集群组件级的性能表现。目前开源版本 Kubernetes 最多可以支撑 5 千节点及 15 万 Pod，已经无法满足日益增长的业务需求。作为云原生领域的实践者和引领者，阿里云基于服务百万客户的经验，从多个维度对 Kubernetes 集群进行优化，率先实现了单集群 1 万节点 1 百万 Pod 的规模突破，可帮助企业轻松应对不断增加的规模化需求。

在应用管理领域，OpenKruise 项目已经正式成为了 CNCF 沙箱项目。OpenKruise 的开源也使得每一位 Kubernetes 开发者和阿里云上的用户，都能便捷地使用阿里内部云原生应用统一的部署发布能力。阿里通过将 OpenKruise 打造为一个 Kubernetes 之上面向大规模应用场景的通用扩展引擎，当社区用户或外部企业遇到了 K8s 原生 Workload 不满足的困境时，不需要每个企业内部重复造一套相似的“轮子”，而是可以选择复用 OpenKruise 的成熟能力。阿里集团内部自己 OpenKruise；而更多来自社区的需求场景输入，以及每一位参与 OpenKruise 建设的云原生爱好者，共同打造了这个更完善、普适的云原生应用负载引擎。

在应用制品管理领域，面向安全及性能需求高的企业客户，阿里云推出容器镜像服务企业版 ACR EE，提供公共云首个独享实例的企业级服务。ACR EE 除了支持多架构容器镜像，还支持多版本 Helm Chart、Operator 等符合 OCI 规范制品的托管。在安全治理部分，ACR EE 提供了网络访问控制、安全扫描、镜像加签、安全审计等多维度安全保障，助力企业从 DevOps 到 DevSecOps 的升级。在全球分发加速场景，ACR EE 优化了网络链路及调度策略，保障稳定的跨海同步成功率。在大镜像规模化分发场景，ACR EE 支持按需加载，实现镜像数据免全量下载和在线解压，平均容器启动时间降低 60%，提升 3 倍应用分发效率。目前已有众多企业生产环境模使用 ACR EE，保障企业客户云原生应用制品的安全托管及多场景高效分发。

我们希望，阿里云上的开发者可以自由组合云上的容器产品家族，通过 ACK Pro+ ACR EE+OpenKruise 等项目，像搭积木一样在云上打造众多企业自有的大规模容器平台。

OpenKruise：阿里巴巴 双 11 全链路应用的云原生部署基座

OpenKruise 是由阿里云于 2019 年 6 月开源的云原生应用自动化引擎，本质是基于 Kubernetes 标准扩展出来一个的应用负载项目，它可以配合原生 Kubernetes 使用，并为管理应用容器、sidecar、镜像分发等方面提供更加强大和高效的能力，从而在不同维度上通过自动化的方式解决 Kubernetes 之上应用的规模化运维和规模化建站问题，包括部署、升级、弹性扩缩容、Qos 调节、健康检查、迁移修复等等。

OpenKruise 官网：https://openkruise.io/
GitHub 项目地址：https://github.com/openkruise/kruise

Kruise 是 Cruise 的谐音，'K' for Kubernetes，寓意 Kubernetes 上应用的航行和自动巡行，它满载着阿里巴巴多年在大规模应用部署、发布与管理最佳实践，以及阿里云 Kubernetes 服务数千客户的需求沉淀。

OpenKruise: 阿里巴巴 双 11 全链路应用的云原生部署基座

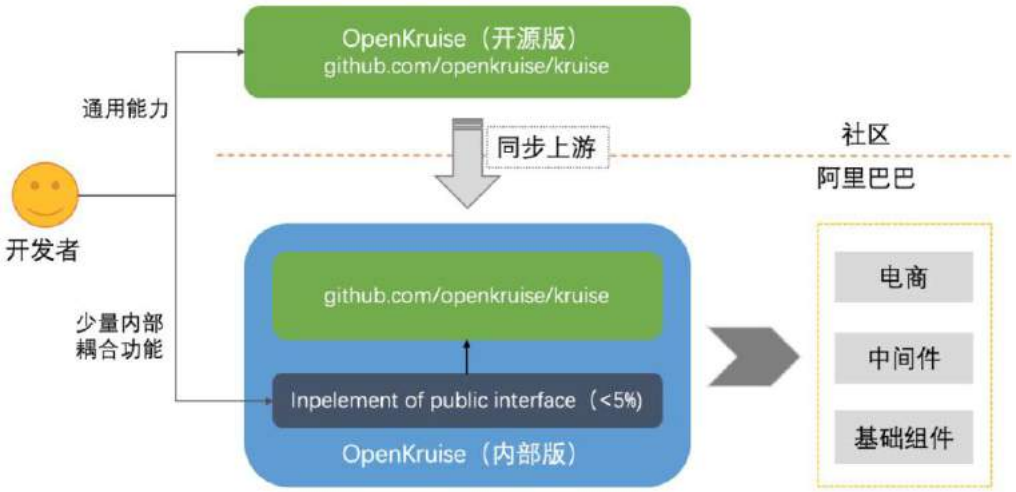
在阿里巴巴的整体云原生化过程当中，阿里的技术团队逐渐沉淀出了一套紧贴上游社区标准、适应互联网规模化场景的技术理念与最佳实践。这其中，最重要的无疑是如何对应用进行自动化的发布、运行和管理。于是，阿里云容器团队将这些能力通过

OpenKruise 反哺社区，以期指引业界云原生化最佳实践，少走弯路。

今年 双 11，阿里巴巴实现了核心系统的全面云原生化。截至 2020 年 双 11，阿里巴巴内部已运行近十万 OpenKruise 的 workload、管理着上百万容器。

（一）内部运行的 OpenKruise 版本代码超 95% 来自社区仓库

下图展示了阿里巴巴内部运行的 OpenKruise 版本与开源版本之间的关系：

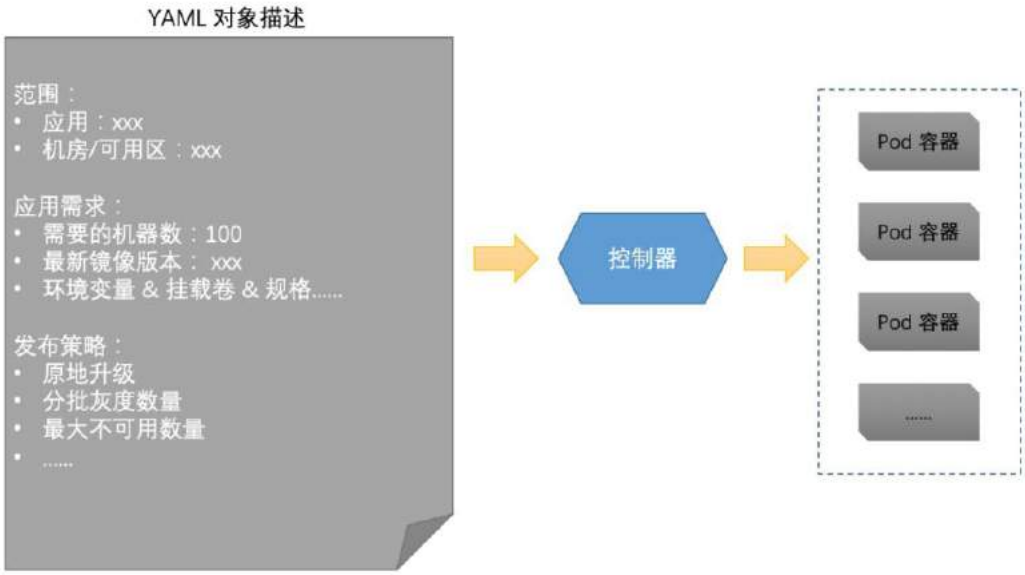


从上图可以看出：Github 上的 OpenKruise 就是我们主体的上游仓库，而内部的下游仓库只基于公共接口实现了极少数内部耦合功能（这部分代码只占据了不到 5%），也就是说阿里巴巴内部运行的 OpenKruise 其中 95% 以上的代码完全来自于社区仓库。有两点值得说明：

- 所有通用能力，我们都会直接基于开源仓库开发和提交，然后再同步到内部环境。
- 社区成员为 OpenKruise 贡献的每一行代码，都将运行在阿里内部环境中。

（二）在 Kubernetes 上自动化应用程序工作负载管理

做上层业务的同学可能对 “应用负载（workload）” 缺乏概念，这里先简单做个介绍。不知道你是否有好奇过，每一次应用扩缩容、发布操作的背后是如何实现的呢？在云原生的环境下，我们都是通过面向终态的方式去描述应用的部署需求（需要的机器数、镜像版本等等），见下图：



应用负载（workload）主要指的就是这个 YAML 定义和对应的控制器：

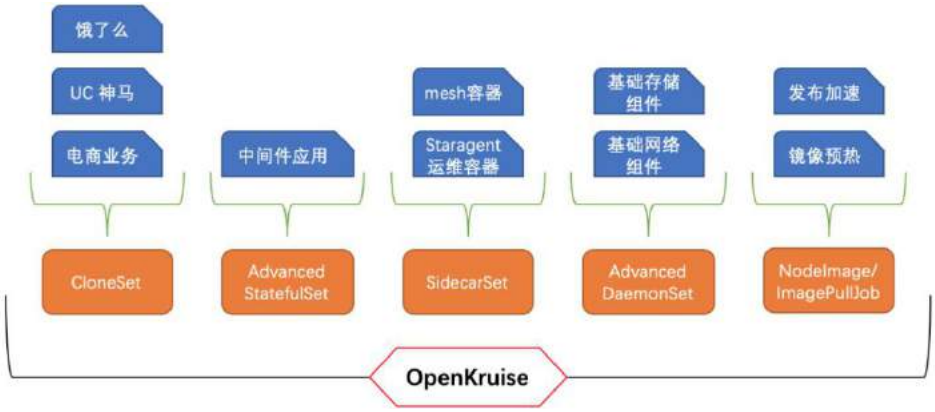
当应用扩缩容时，PaaS（运维平台）会修改上述 YAML 对象中的需求机器数（比如扩容 10 台改为 110，再缩容 5 台则改为 105），然后控制器就会按照 workload 期望的数量来调整实际运行的 Pod（容器）数量。

当应用触发发布或回滚时，PaaS（运维平台）则会修改上述 YAML 对象中的镜像版本和发布策略，控制器就会按照给指定的发布策略来将所有管理的 Pod（容器）重建为期望版本（这只是一些便于理解的简化描述，实际工作机制会复杂得多）。

也就是说，应用负载（workload）管理着应用所有容器的生命周期。不仅应用扩容、缩容、发布都依赖于 workload 的工作，workload 还负责持续维持应用运行时的 Pod（容器）数量，来保证持续有符合期望数量的实例在跑着。如果有宿主机发生故障、上面的应用实例被驱逐，那么 workload 会立即再为应用扩出新的容器。

（三）双 11 核心应用全面基于 OpenKruise 部署

随着阿里巴巴上云，双 11 主体相关的电商类业务、以及中间件等应用都迁移到了云原生环境下，统一使用 OpenKruise 的应用负载能力做部署。OpenKruise 提供了多种不同类型的 workload 来支持不同的部署方式：



- CloneSet：（无状态应用）这是规模最大的部分，绝大部分泛电商业务都是通过 CloneSet 来部署发布。
- Advanced StatefulSet:（有状态应用）目前主要是用于中间件在云原生环境的部署。
- SidecarSet：（sidecar 生命周期管理）可以将定义好的 sidecar 容器动态注入到新建的 Pod 中，云上的运维容器、 mesh 容器都是通过这种机制加入到业务 Pod 中。
- Advanced DaemonSet：将宿主机级别的守护进程部署到所有节点上，包括各种用于给业务容器配置网络、存储的基础组件。

因此，我们看到从上层电商业务到中间件，再到运维容器、基础组件，整个上下游链路都是依赖于 OpenKruise 提供的 workload 做部署和运行。不管是应用运行时的机器数量、版本管理，还是紧急扩容、发布等操作，都有 OpenKruise 无时无刻在维护着。

可以想象，如果 OpenKruise 出现了故障会发生什么?

如果只是控制器挂了，则应用扩缩容、发布操作全量失败。

而如果控制器逻辑存在重大 bug，比如数量或版本号计算错误，甚至可能引起业务容器大规模误删或是升级为错误的版本。

当然，针对以上高危情况，我们做了很多重的防护措施，务必保障业务的稳定可用。

这就是阿里巴巴的云上部署基座，OpenKruise 几乎承载了全量 双 11 业务的部署管理与运维职责。

（四）主要能力

OpenKruise 从何而来？或者说什么问题或需求促使了 OpenKruise 的诞生呢？

当上云成为大势、当云原生逐渐成为标准，我们却发现 Kubernetes 原生提供的 workload 能力根本无法满足阿里巴巴超大规模业务场景的需求：

- 应用发布时，所有容器都要飘移重建：对于我们来说几乎无法接受，在发布高峰期如果阿里巴巴的大规模应用都在大规模重建，这是不管对于业务自身还是其他调度器、中间件、网络/存储组件都是一种灾难性的压力。
- 无状态应用负载（Deployment）无法灰度升级容器。
- 有状态应用负载（StatefulSet）无法并行升级容器。
- 还有很多，这里不一一列举……

在这种背景下，OpenKruise 出现了。我们通过或是全新开发（CloneSet、SidecarSet），或是兼容性增强（Advanced StatefulSet、Advanced DaemonSet），来使得上层业务终于可以顺利落地云原生。

OpenKruise 首推的功能就是“原地升级”。通过这种能力，我们终于可以使应用发布不需要将容器飘移重建，而是在原地、原 Pod 上只升级需要更新的镜像。这样带来的好处太多了：

- 发布效率大大提升。根据不完全统计数据，在大部分业务场景下原地升级至少比完全重建升级提升了 80% 以上的发布速度：不仅省去了调度、分配网络、分配远程盘的耗时，连拉取新版本镜像的时候都得益于 node 上已有旧镜像、只需要拉取较少的增量 layer。
- 发布前后 IP 不变、升级过程 Pod 网络不断，并且 Pod 中除了正在升级容器之外的其他容器都一直保持正常运行。
- Volume 不变，完全复用原容器的挂载设备。
- 确保了集群确定性，使全链路压测通过后的集群拓扑为大促提供保障。

当然，除此之外我们还增强了许多其他的高级能力，满足了许多种面向大规模场景下的业务诉求，本文不做一一介绍，但下图可以看到 OpenKruise 与 Kubernetes 原生应用

负载针对无状态、有状态应用的功能对比：

功能	Deployment	CloneSet	StatefulSet	Advanced StatefulSet
Pod 名字	Generated	Generated	Ordered	Ordered
先扩再缩（主动迁移）	No	Yes	No	No
流式扩容	No	Yes	No	No
Pod重建升级	Yes	Yes	Yes	Yes
Pod原地升级	No	Yes	No	Yes
分批（灰度）发布	No	Yes	Yes	Yes
最大不可用数量	Yes	Yes	No	Yes
最大弹性数量	Yes	Yes	No	No
发布顺序 可配置 (优先级、打散)	No	Yes	No	Yes
Pod升级前后hook (preUpdate/postUpdate)	No	Yes	No	No

橙色：开源中
绿色：已开源

OpenKruise 已正式进入 CNCF Sandbox

2020 年 11 月 11 日，在这个特殊的时点，阿里巴巴技术人又迎来一件大事：经 CNCF 技术监督委员会全体成员投票，一致同意将阿里云开源的 OpenKruise 正式晋级为 CNCF 托管项目。

正如开篇所说，OpenKruise 已经完成了社区开源，并且内外的版本做到几乎完全一致。除此之外，我们还将 OpenKruise 提供到了阿里云容器服务的应用目录中，公有云上的任意客户都可以一键安装和使用 OpenKruise，真正实现 OpenKruise 在阿里集团内部业务、云产品、开源社区中的“三位一体”。目前在 ACK 上使用 OpenKruise 的客户主要包括斗鱼 TV、申通、有赞等，而开源社区中携程、Lyft 等公司也都是 OpenKruise 的用户和贡献者。

OpenKruise 将基于阿里巴巴超大规模场景锤炼出的云原生应用负载能力开放出来，不仅在云原生社区中补充了扩展应用负载的重要板块，还为云上客户提供了阿里巴巴多年应用部署的管理经验和云原生化历程的最佳实践成果。从正式开源之日起，OpenKruise 项目已经建立多个关键里程碑：

- Maintainer 5 位成员来自阿里巴巴、腾讯、Lyft
- 44 位贡献者
- 国内：阿里云、蚂蚁集团、携程、腾讯、拼多多...
- 国外：微软、Lyft、Spectro Cloud、Dsicord...
- 1900+ GitHub Stars
- 300+ Forks

后续，OpenKruise 的重点包括但不限于以下几个目标：

继续将阿里巴巴内部沉淀的通用云原生应用自动化能力输出，走可持续的三位一体发展战略。

深度挖掘细分领域的应用负载需求，比如我们正在探索针对 FaaS 场景的池化能力。

与其他相关领域的开源产品做更紧密的结合，如 OAM/KubeVela 等，打造更完整的云原生应用体系。

欢迎加入 OpenKruise 大家庭

如果你对 OpenKruise 的发展有任何建议，欢迎发表在下方评论区。另外，你可以通过扫码进入 “OpenKruise 社区交流钉钉群”，我们衷心欢迎每一位开源爱好者来参与 OpenKruise 的建设，共同打造云原生领域最成熟、面向最大规模的应用自动化引擎。

揭开阿里巴巴复杂任务资源混合调度技术面纱

作为更进一步的云计算形态，云原生正在成为云时代的技术新标准，通过重塑整个软件生命周期，成为释放云价值的最短路径。

在企业内部，将云原生基础架构作为企业内部的统一架构已成为大势所趋。与此同时，也必然带来了由各种基础平台整合带来的兼容性问题，特别是规模越大、历史沉淀越多的企业，这种“技术债务”体现得越明显。

本文分享的经验来自于阿里巴巴过去数年来在混合调度方面积累的生产实践积累，具有很强的生产实用价值。内容由浅入门，逐渐深入到调度器内幕，讲述在大规模容器调度场景下，阿里巴巴针对云原生应用设计的统一基础设施 ASI（Alibaba Serverless infrastructure）调度器如何管理阿里巴巴如此复杂、繁忙的资源调度任务；并尝试通过一些具体的案例让您得以充分理解，相信会为有类似问题的读者打开设计思路，并提供落地借鉴。通过本文，相信您将系统地理解阿里巴巴复杂任务场景下的资源混合调度。

调度器总览

ASI 在阿里集团内部引领着容器全面上云的实施，承担了包括阿里集团内部轻量级容器架构演进、运维体系云原生化等职责，并进一步加速促进了新兴的技术包括 Mesh、Serverless、Faas 等在阿里集团内的落地；支撑了包括淘宝、天猫、优酷、高德、饿了么、UC、考拉等几乎所有阿里巴巴内部业务、阿里云云产品众多场景及生态。

ASI 的核心基于 Kubernetes，并提供完整的云原生技术栈支持。如今的 ASI 也已成功实现与阿里云容器服务 ACK（Alibaba Cloud Container Service for Kubernetes）的会师；而 ACK 既保留了云上的各种能力，也能成功应对阿里集团复杂的业务环境。

ASI 调度器是 ASI 云原生的核心组件之一。在 ASI 云原生的发展历程中，调度器的作用至关重要。最直观的认知是：阿里巴巴规模庞大的在线电商交易容器，例如购物车、订单、淘宝详情等，每一台容器的分布，包括容器编排、单机计算资源、内存资源，均由调度器分配和调度；特别是在 双 11 零点峰值场景下，少数容器编排错误都有可能给业务带来致命影响，调度器需负责把控峰值时每一台容器计算的质量，其重要性可想而知。

ASI 调度器起源于 2015 年在线电商交易容器调度，这一年最早的调度器仅涵盖在线交易 T4（阿里早期基于 LXC 和 Linux Kernel 定制的容器技术）和 Alidocker 场景，出生即责任重大，并在 2015 年扛住 双 11 流量峰值中发挥作用。

ASI 调度器的演进之路也伴随着云原生发展的全过程。它经历了最早期的在线交易容器调度器、Sigma 调度器、Cerebulum 调度器、ASI 调度器；到如今我们正在建设的下一代调度器 Unified-Scheduler，它将进一步吸收并融合过去数年阿里巴巴 ODPS（伏羲）、Hippo 搜索、在线调度在各个领域的先进经验。各阶段的调度解读如下图：

在 ASI 调度器的演进过程中有非常多的挑战需要解决，主要体现在：

- **调度器调度的任务种类丰富多样，**有海量的在线长生命周期容器和 POD 实例、Batch 任务、众多形态的 BestEffort 任务等不同 SLO 等级的任务；有计算型、存储型、网络型、异构型等众多不同资源类型的任务，不同任务的诉求和场景又千差万别。
- **调度之上的宿主资源各异。**调度管理着阿里集团内部数量庞大的宿主资源，包括众多机型的存量非云物理机、云上神龙、ECS、异构机型如 GPU/FPGA 等。
- **调度器服务场景广泛。**例如：最典型的泛交易场景；最复杂的中间件场景；Faas/Serverless/Mesh/Job 等众多新兴计算场景；饿了么、考拉、神马等新兴生态场景；公共云上伴随着多租安全隔离的调度诉求；还有全球范围内都极具有挑战性的 ODPS（伏羲）、Hippo、蚂蚁、ASI 统一调度场景。

- **在基础设施层的职责众多。**调度器部分承担着基础设施机型定义、计算存储网络资源整合、收敛硬件形态、透明化基础设施等众多职责。

关于阿里云原生详细的发展历程，有兴趣的同学可以通过《一个改变世界的“箱子”》这篇文章来了解。下面，我们重点来分享 ASI 调度器是如何管理着阿里如此庞大规模、如此复杂繁忙的计算资源调度任务。

调度器初探

（一）调度器是什么

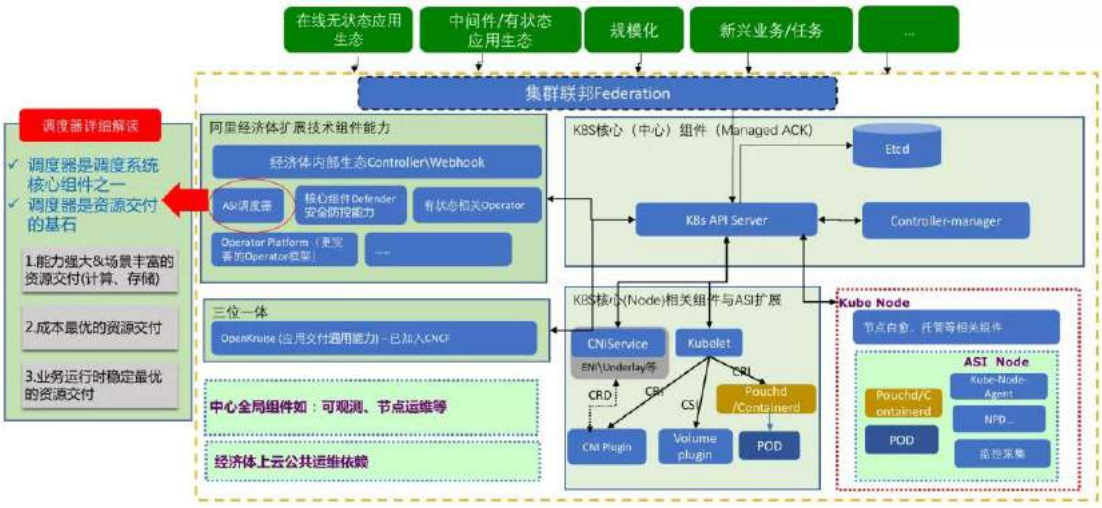
调度器在 ASI 众多组件中的作用非常核心。调度器是云原生容器平台调度系统内众多核心组件之一；调度器是资源交付的基石。调度器是 ASI 云原生的大脑。调度器的价值主要体现在：

- 能力强大 & 场景丰富的资源交付(计算、存储)
- 成本最优的资源交付
- 业务运行时稳定最优的资源交付

更通俗地讲，调度器要做的是：

- 一次作业调度的最优：在集群内选择一台最合适的宿主机，并在这台宿主机上，以最佳的资源使用姿势，做到最少的相互干扰（如 CPU 分布、IO 争抢）来运行用户提交的计算作业。
- 集群全局调度的最优：确保全局资源编排最优（如碎片等）、资源运行最稳定、全局成本最优。

在 ASI 云原生体系中，中心调度器所在的位置如下图（其中标红的框框所示）：



（二）广义的调度器

在大部分时候，业界讲到调度器指的是“中心调度器”，例如社区的 K8s kube-scheduler。但真实的调度场景复杂，每一次调度都是一个复杂又灵活的综合体协同。作业提交后，它需要中心调度器、单机调度、内核调度多层次调度共同协调，并进一步在 K8s 组件 kubelet、controller 等配合下完成执行；在线调度场景，又有着批量编排调度器；而重调度下的多次调度则确保集群总是保持最优。

ASI 广义的调度器理解为：中心调度器、单机调度、内核调度、重调度、规模化编排调度、多层调度 一体的综合体。

1）中心调度器

中心调度器负责计算每一个（或一批）作业的资源编排计算，它确保一次调度最优。中心调度器为这个具体的任务计算确定诸如集群、地域、执行节点（宿主机）等信息，更进一步细化节点上的 CPU 分配、存储、网络资源分配。

中心调度器在 K8s 生态组件协同下，管理着大部分任务的生命周期。

ASI 云原生的演进之路中，中心调度器即上文描述的 Sigma 调度器、Cerebulum 调度器、ASI 调度器等等。

2）单机调度

主要负责两类职责：

第一类职责：统筹协同单机内多 POD 的最佳运行。ASI 在接受到中心调度器的节点选择指令后，将任务调度到具体的节点上执行，单机调度即开始工作：

- 单机调度将立刻、或周期性、或运维式 动态确保单机内多 POD 的工作最优，这意味着它将在单机内统筹协同资源，例如：每一个 POD 的 CPU 核分配的最佳调整。
- 实时根据 POD 的运行指标如负载、QPS 等，针对部分运行时资源执行单机内的 VPA 扩缩容、或对低优先级的任务执行驱逐等操作。例如：动态扩展 POD 的 CPU 容量。

第二类职责：单机资源信息的采集、上报、汇聚计算，为中心调度提供决策依据。在 ASI 内，单机调度组件主要指 SLO-Agent、Kubelet 的部分增强能力；在正在建设的 Unified-Scheduler 调度内，单机调度主要指 SLO-Agent、Task-Agent、以及 Kubelet 的部分增强能力。

3）内核调度

单机调度从资源视角统筹单机内多 POD 的最佳运行，但任务的运行态实际由内核控制。这就需要内核调度。

4）重调度

中心调度器确保了每次任务的最佳调度，即一次性调度问题；但中心调度器并不能实现集群维度的全局最优，这就需要重调度。

5）规模化编排调度

规模化编排调度是阿里巴巴大规模在线调度的特有场景，自 17 年开始建设，现在已经非常成熟，并仍在持续增强中。

利用规模化编排能力，我们可以一次性调度数万、数十万容器，一次性确保集群维度所有容器的全局最佳编排。它非常巧妙地弥补了一次性中心调度的弊端，规避了大规模建站场景下需反复重调度的复杂度。

关于内核调度、重调度、规模化编排调度，我们将在下面的章节中详细展开。

6）调度分层

另一个维度，我们也会定义调度分层，包括 一层调度、二层调度、三层调度...等；Sigma 在离线混部场景甚至引入了零层调度的概念。每个调度系统对调度分层的理解和定义会不一样，并都有各自的概念。例如，在过去的 Sigma 体系内，调度分为 0 层、1 层 和 2 层调度：

- 0 层调度器负责的职责是负责全局资源视图和管理，并承接各个 1 层调度间的调度仲裁，以及具体执行；1 层调度主要是对应 Sigma 调度器、伏羲调度器 [也可以包含其它调度器]。
- 在 Sigma 体系中，Sigma 调度器作为 1 层调度，负责资源层的分配。
- 2 层调度交由不同的接入业务各自实现（例如电商交易、广告 Captain、数据库 AliDB 等）。2 层调度充分贴近和理解各自业务，并站在业务全局视角做众多优化，建设调度能力，如业务驱逐、有状态应用故障自动运维等，做到贴心服务。

Sigma 的调度分层体系的致命弊端是，各个二层调度的技术能力和投入参差不齐；例如广告的二层调度系统非常优秀，但并不是所有的二层调度对业务贴心到极致。ASI 吸取教训，将众多能力下沉至 ASI 内，并进一步标准化上层 PAAS，简化上层的同时增强上层能力。

而今天正在建设的下一代调度器概念内，也分为多层，例如：计算负载层（主要指 Workload 的调度管理）、计算调度层（如 DAG 调度、MR 调度等）、业务层（同 Sigma 2 层的概念）。

（三）调度资源类型

我尝试用正在建设的 Unified-Scheduler 调度器来让大家更好地理解。在 Unified-Scheduler 调度器内，调度着 Product 资源、Batch 资源、BE 计算资源三种分等级的资源形态。

不同调度器对分等级的资源形态有不同的定义，但本质上大同小异。为了让大家更好地理解这一精髓，我在后续的章节对 ASI 调度器也做了详细讲解。

1）Product（在线）资源

有 Quota 预算的资源，且调度器需保障其最高级别的资源可用性。典型代表是在线电商核心交易的长生命周期 POD 实例。最经典的例子就是 双 11 核心链路上的购物车（Cart2）、订单（tradeplatform2）等交易核心的业务 POD。这些资源要求算力的严格保障、高优先级、实时性、响应低延时、不可被干扰等等。

举例来说，在线交易的长生命周期 POD 的存在时间很长，数天、数月、甚至数年。大部分应用研发同学申请的应用，在构建完毕后需要申请数台长生命周期实例，这些都是 Product 资源。淘宝、天猫、聚划算、高德、友盟、合一、菜鸟、国际化、闲鱼....等等众多业务研发同学申请的 POD（或容器）实例，相当大部分都是 product 资源。

Product 资源不仅仅指在线长生命周期的 POD；凡是符合上述定义的资源请求，都是 Product 资源。但并不是所有的长生命周期 POD 都是 Product 资源。例如阿里内部“Aone 实验室”用于执行 CI 构建任务的 POD，可以长生命周期存在，但它可以被低成本驱逐抢占。

2）Batch 资源

在线业务使用的 Product 资源的 Allocate 和 Usage 之间的 Gap 在一段时间内是比较稳定的，这个 Gap 和 Prod 未分配的资源就作为 BE 资源，售卖给针对 latency 不那么敏感和但是对资源稳定性有一定需求的业务。Batch 有 quota 预算，但是保证一段时间内（例如 10 分钟）的一定概率（例如 90%）的资源可用性。

也就是说，Product（在线）资源申请走了账面上的资源，但实际上从负载利用率指标来看可能有众多算力未被使用；此时将发挥调度器的差异化 SLO 分等级调度能力，将那些未跑满的部分，作为超发资源充分使用，售卖给 Batch 资源。

3）Best Effort(BE)资源

指没有 Quota 预算，不保障资源可用性，随时可以被压制和抢占；节点上已分配在节点的 Usage 低于一个水位的时候，调度器认为这部分 Gap 是一个“比较不稳定/不记账”的资源，那么这个 Gap 就称为 BE 资源。

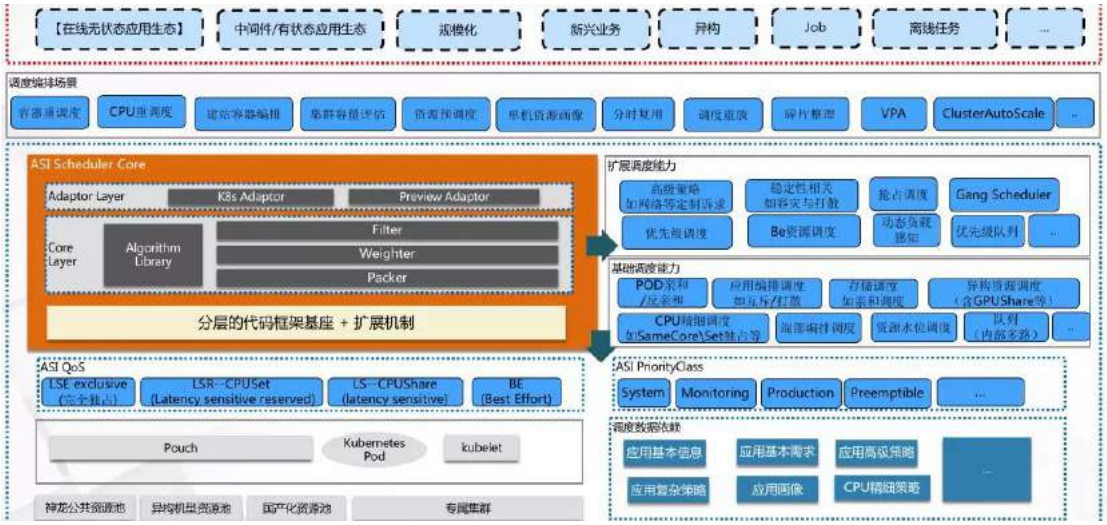
我们可以这样来比方：Product、Batch 资源负责大块吃肉，BE 资源则负责消费 Product 和 Batch 不要的残渣。例如：日常开发工作中，研发需要跑很多 UT 测试任务，这类计算任务对计算资源的质量要求并不高，时间延时的容忍度也比较高，也不大好评估额

度预算，针对这类场景去购买大量的 Product 或者 Batch 资源，将非常不划算；但如果使用最廉价的 BE 资源，收益将相当可观。此时，BE 资源就是 Product/Batch 运行中没有用到的资源。

很容易理解到，正是通过这种分等级的资源调度能力，从技术层面，Unified-Scheduler 调度器可以将一台物理节点的资源使用，发挥到极致。

调度器能力总览

下图是 ASI 围绕着广义调度需覆盖的职责，并对应不同资源等级诉求、以及服务的丰富业务场景，构建的调度能力总览。通过这张图，大家可以理解到 ASI 调度器的技术全貌。



四、典型在线调度能力

（一）在线调度的业务诉求

在 ASI 云原生容器平台上，在线部分服务着交易、导购、直播、视频、本地生活、菜鸟、高德、合一、友盟、海外等数十个 BU 的各类调度场景。其中最高等级的“Product 资源”的调度占比最为庞大。

在线业务的调度与离线调度、众多 JOB 型调度相比较，有着典型的差异（描述在线场景时，大家可以想象到，离线调度的世界同样精彩）。

1）生命周期

- Long Running：在线应用的容器生命周期普遍都比较长。少则数天，大部分以月计，部分长尾应用甚至存活数年。
- 启动时间长：应用的镜像体积大，下载镜像时间较长，服务启动内存预热等等，这导致应用启动时间在几秒、数十分钟都有。

长生命周期的特性，与一些典型的短生命周期任务调度（如 FaaS 函数计算），在任务特征上有着本质的不同，背后的技术挑战也截然不同。例如：相对短生命周期的函数计算场景的挑战是：最极致的调度效率、百毫秒的执行效率、快速的调度吞吐、POD 运行时性能等。而长生命周期 POD 带来的差异化挑战是：全局最优的调度必须依赖重调度来持续迭代优化；运行时的最佳调度必须依赖单机重调度持续优化保障。可以想象，在过去非云原生时代，众多业务不可迁移，对调度而言简直是噩梦；这意味着调度器不仅仅面对调度能力的技术问题，还需面对难度巨大的存量业务治理推动；在线应用的启动时间长，又更加剧降低了重调度的灵活度，带来更多的复杂度。

2）容器运行时

- 容器运行时需要支持业务的实时交互、快速响应、低业务 RT 等诉求。在线容器运行时，大部分系统需承担实时交互职责，并对延迟异常敏感，稍大的延迟将带来明显糟糕的业务体感。
- 资源特征明显：如网络消耗型、IO 消耗型、计算消耗型等等。相同特征的实例共存时，极易发生彼此间的明显资源争抢。

在线容器的运行时由于对业务和算力都非常敏感，因此对调度质量提出了非常苛刻的挑战。

3）应对阿里在线应用特有的复杂业务模型

- 流量高低峰特征：在线业务的服务一般都会有比较明显的高低峰，例如饿了么的高峰是在中午和晚上、淘宝的高峰也有明显的波谷和峰值。
- 突发流量：业务的复杂度，导致这些突发流量并不一定能呈现一定规律；例如直播业务可能因为某个突发的事件导致流量激增。突发流量背后的技术诉求往往是弹性，最经典的案例是 2020 年疫情期间的钉钉弹性诉求。
- 资源冗余：在线业务从出生的那一刻开始，就定义了冗余资源；这主要是出于容灾的考虑。但站在阿里巴巴全局视角，相当多的长尾应用因规模小而对成本和利用率不敏感，积少成多，背后是巨大的算力浪费。

4）特有的规模化运维诉求

- 复杂的部署模型：例如：需要支持应用单元化部署，多机房容灾，小流量、灰度、正式多环境部署的复杂调度需求。
- 大促 & 秒杀的规模化峰值特征：阿里巴巴的各种大促贯穿全年，比如大家熟悉的 双 11、双 12、春节红包等等。整个链路上的应用压力、资源消耗都会随着大促峰值流量的增长成倍增加，这需要调度器强大的规模化调度能力。
- 大促建站：大促的时间是计划性的，为了节约云资源的采购成本，必须尽可能降低云资源的保有时间。调度器需最快速度完成大促前的建站，并在大促后快速归还资源到阿里云。这意味着极其严苛的规模化调度效率诉求，并把更多的时间留给业务。

（二）一次调度：调度基本能力

下表详细描述了在线调度最常见的调度能力：

调度规则	解读
应用基本诉求	业务对一些基本必要项，如规格、OS的要求
容灾与打散	同一个应用的多个容器，务必按网络核心、ASW（一种网络单元）、机架、宿主机充分打散，确保好的容灾
高级策略（数十项）	应用对物理资源、网络、容器的一些定制要求
应用编排	其中最核心的是： 1. 应用对宿主机的的独占要求，以确保绝对的防干扰。 2. 多个应用部署在同一台宿主机上的互斥要求
CPU精细编排	最大化使用宿主机物理核资源应用之间；最小的CPU干扰隐患
重调度	1.应用重调度 2.CPU重调度能力

- 应用基本诉求对应的是：应用扩容对应的基本诉求，例如 POD 规格、OS 等。在 ASI 调度器内，它被抽象为普通的 label 匹配调度。
- 容灾与打散：locality 调度，ASI 已经通过各种手段获取到诸多详细信息，例如上图中的 网络核心、ASW 等。
- 高级策略：ASI 会尽可能标准化、通用化业务诉求，但仍然不可避免地存在一些业务，对资源、运行时有众多特定要求，例如特定的基础设施环境如硬件等、容器能力的特定诉求如 HostConfig 参数、内核参数诉求等。

- 关于调度规则中心：业务对策略的特定要求，决定了调度背后还将对应一个强大的调度策略中心，它指导着调度器使用正确的调度规则；调度规则中心的数据来自于学习，或专家运维经验。调度器采纳这些规则，并应用于每一个 POD 的扩容分配中。

（三）应用间编排策略

因集群节点数量有限，彼此潜在干扰的众多应用，不得已需在同一节点并存时，这时就需要应用间编排策略，来确保每一个宿主节点和每一个 POD 运行时最优。

实际的调度生产实践中，“业务稳定性”永远是排在第一位，但资源却总是有限的；我们很难平衡“资源成本最优”和“业务稳定性”。大部分情况下，应用间编排策略都能完美地解决这一平衡；通过定义应用之间（如 CPU 消耗密集型、网络消耗型、IO 密集型、峰值模型特征等）的并存策略，集群内充分打散，或同一节点并存时又有充分的策略约束保护，进而做到不同 POD 间的干扰概率最小。

更进一步，调度器在运行时辅以更多技术手段优化，例如：通过网络优先级控制、CPU 精细编排控制等策略，来尽可能规避应用间运行时的潜在影响。

应用间编排策略带来的其它挑战是：调度器在建设好本职的应用间编排能力外，还需充分理解其上运行的每一个业务运行特征。

（四）CPU 精细化编排

CPU 精细编排在“在线调度领域”内是非常有意思的话题，它包括 CpuSet 调度、CpuShare 调度。其它场景的调度领域，例如离线调度领域，它并没有这么重要，甚至不可被理解；但在线交易场景下，无论是理论推断、实验室场景、还是无数次大促压测数据，都证明了精准的 CPU 调度就是这么重要。

CPU 精细编排的一句话解读是：调核，确保 CPU 核最大化、最稳定地被使用。

CPU 精细编排如此重要，以至于 ASI 在过去的数年，已经将这一规则吃透并使用到了极致。相信您在看到下表后（仅含 CpuSet 精细编排调度），您也会感叹 ASI 甚至已经将它玩出了花样。

归类	调度规则项	解读
CpuSet精细编排复杂策略 （尽可能规避争抢隐患、最大化使用CPU	CPU物理核堆叠规避	同一容器|同一分组的不同容器，尽量避免共用相同物理核
	CPU独占	非核心应用允许逻辑核重叠的同时，重保指定核心应用独占CPU。（将带来极大的成本节约）
	同一容器尽量不要跨Socket	尽量减少跨socket带来的L3 cache问题
	CPU物理核互斥	效果：峰值时避免交易核心链路应用彼此争抢，并尽可能提高CPU利用效率。 在CPU物理核堆叠SLO符合期望的基础上，交易核心类应用之间也尽量不要共用物理核。
CpuSet编排（简单策略）	CPU预期外超卖	不允许出现；如出现需立刻监控到并重调度。
	CPU逻辑核预期外重叠规避	如出现CPU逻辑核预期外重叠，必然会发生资源争抢
在离线、离在线混部CpuSet编排	在离线混部在线使用所有core	离线使用S30资源（S30特征：CpuShare值最低，随时被noise clean在线抢占。但离线可削谷填峰使用CPU核）
	离在线混部在线仅使用同一物理核上的1个逻辑核	离线退水，在线拉起后，在线最大化使用CPU 整个物理core。

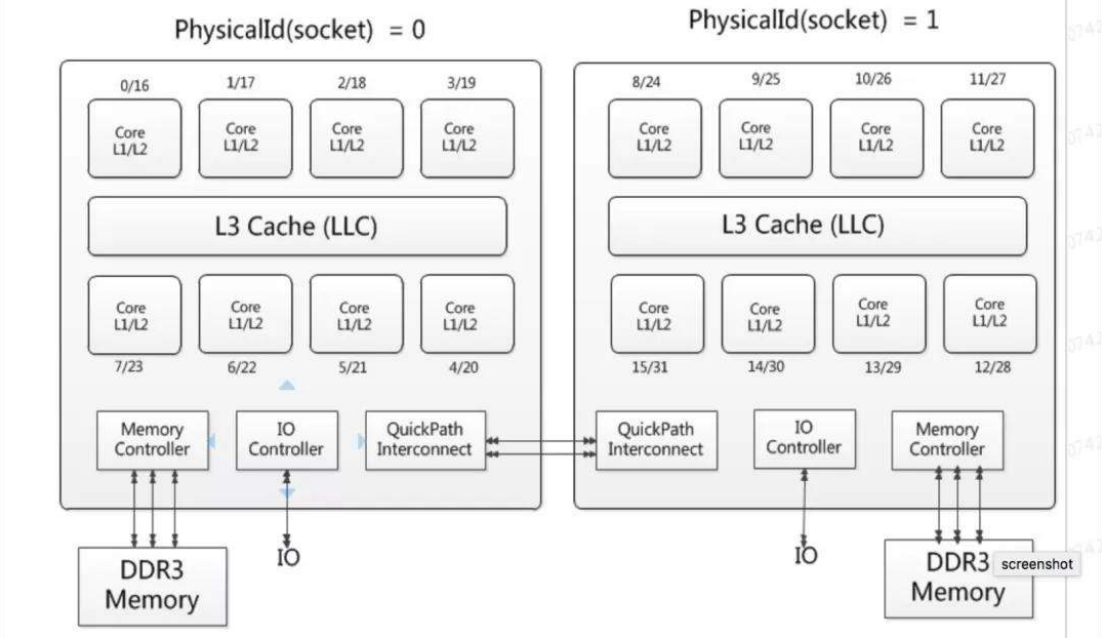
科普：以一台 96 核（实际上我们说的都是 96 个逻辑核）的 X86 架构物理机或神龙为例，它有 2 个 Socket，每个 Socket 有 48 个物理核，每个物理核下有 2 个逻辑核。【当然，ARM 的架构又与 X86 不同】。

由于 CPU 架构的 L1 L2 L3 Cache 设计，最理想的分配是：同一个物理核下的 2 个逻辑核，其中一个核 分配给最核心的在线交易应用如 Carts2（购物车业务），另一个核分配给另一个不繁忙的非核心应用；这样在日常、或 双 11 零点峰值时，Carts2 可以占尽便宜。这种用法，在实际的生产环境、压测演练环境中均屡试不爽。

假如我们将同一个物理核上的两个逻辑核，都同时分配给 Carts2 时，由于业务峰值的相同（尤其是同一个 POD 实例），资源的最大化使用就会大打折扣。

理论上我们也应该尽量规避两个同样都是交易核心的应用，例如 Carts2（购物车业务）、tradePlatform2（订单），使其不要去共用这两个逻辑核。但实际上在微观层面，Carts2 和 tradePlatform2 的峰值会有差异，所以实际上影响还小。尽管这样的 CPU 分配看起来有些“将就”；但物理资源总归是有限的，也只能保持这份“将就”了。

而在 numa-aware 开启时，为了最大化使用 L3 Cache 来提升计算性能，同一个 POD 的更多核，又应确保尽量规避跨 Socket。

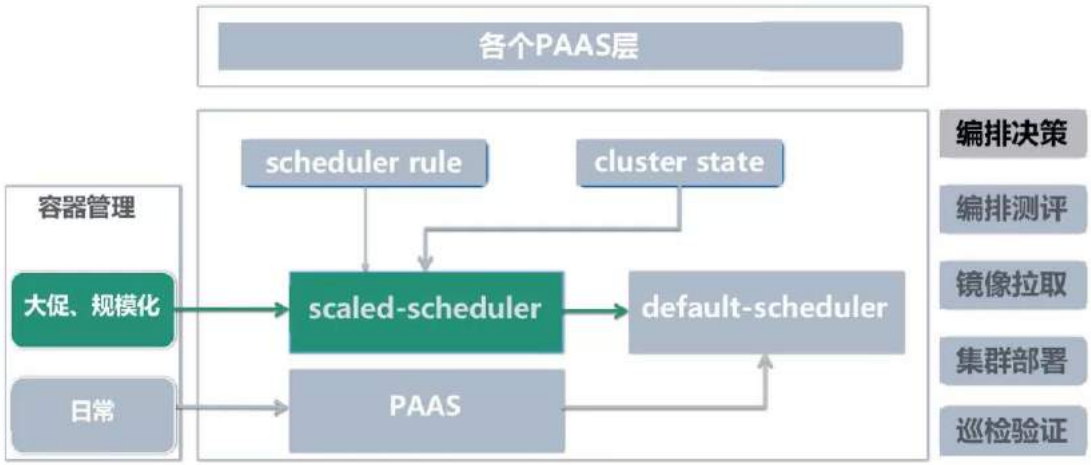


而当使用 CPUShare 时，Request 和 Limit 如何分配，也非常有学问；CPUSet 和 CPUShare 并存时，调度将更加复杂（例如：CpuSet 容器的新扩容、或下线，潜在的诉求是整机所有 POD 的 CPU 重调度）；而在新兴的 GPU 异构调度场景下，CPU 与 GPU 的并存分配也具备一定的技巧。

（五）规模化编排调度

规模化编排主要应用于建站、搬站或规模化迁移场景，例如阿里巴巴频繁的大促建站、机房迁移诉求下的超大规模搬站等。基于成本考量，我们需要在尽可能短的时间，以极少的人力成本快速创建数十万级别 POD。

多个任务依次请求的随机性和不可预见性，决定了中心调度在规模化领域存在诸多弊端。在没有规模化编排能力之前，阿里巴巴大规模站点的建设，往往需经历复杂的 “业务自扩容 -> 反复重调度” 的过程，这将耗费大量人力数周的精力。好在我们有了规模化编排调度，在小时级规模化交付效率的同时，又能确保 99% 以上的资源分配率。



通用调度能力

（一）重调度

中心调度器做到了一次性最优调度；但与最终想要的集群维度全局调度最优，是两个完全不一样的概念。重调度也包括全局中心重调度和单机重调度。

为什么一定需要中心重调度作为一次性调度的补偿呢？我们举几个例子：

- ASI 调度集群内存在众多长生命周期的 POD 实例；随着时间的积累，集群维度必将产生众多资源碎片、CPU 利用率不均等问题。
- 大核 POD 的分配需要动态的腾挪式调度能力（实时驱逐某些小核 POD 并空闲出资源）、或基于提前规划的全局重调度，在众多节点上预空闲一些大核。
- 资源供给总是紧张的。对某个 POD 做一次性调度时，可能存在一定“将就”，这意味着某种瑕疵和不完美；但集群资源又是动态变化的，我们可以在其后的某个时刻，对该 POD 发起动态迁移，即重调度，这将带来业务更佳的运行时体验。

中心重调度的算法、实现上往往非常复杂。我们需要深入理解各类重调度场景并充分覆盖，定义清晰的重调度 DAG 图，动态执行并确保执行的成功率。

众多场景也需要单机重调度。例如：CPU 精细编排的 SLO 优化、基于 OQS 数据驱动的单机重调度优化等。

需要强调的是，单机重调度的执行，必须先解决安全风控的问题，规避不可控的爆炸半径。在单机侧风控能力不足前，我们建议您暂不要采用节点自治方式，而是改为严格保护控制下的中心统一触发。实际上在 K8s 域内，会出现非常多不可避免的节点自治场景（例如 pod yaml 变化时，Kubelet 将执行相应变更），过去 ASI 花费数年持续梳理每一个潜在的风控点，并迭代建设了分等级风控管理（核按钮、高危、中危等）的 Defender 系统；针对潜在风险项，执行单机侧动作前，与中心的 Defender 交互，通过安全防控规避灾难事件的发生。我们建议调度器必须同样做到严密的安全防御等级，才允许节点自治操作。

（二）内核调度

内核调度存在的背景是：一台并行运行着众多任务的繁忙宿主机，即使中心调度 & 单机调度，已共同确保其最佳资源分配（如 CPU 分配、IO 打散等），但实际运行时，多任务间不可避免地进行内核态的资源争抢，在大家熟知的在离线混部场景中竞争尤为激烈。这就需要中心调度、单机调度、内核调度 通过众多协同，例如统筹任务的各类资源优先级，并交由内核调度控制执行。

这也对应众多内核隔离技术。包括 CPU：调度优先级 BVT、Noise Clean 机制等；内存：内存回收、OOM 优先级等；网络：网络金银铜优先级、IO 等等。

在今天我们还有了安全容器。基于安全容器的 Guest Kernel 和 Host Kernel 隔离机制，我们可以更优雅地规避内核运行态的部分争抢问题。

（三）弹性调度、分时调度

弹性和分时的逻辑都是更好的资源复用，只是维度不一样。

ASI 调度器与阿里云基础设施层充分协同，利用 ECS 提供的强大弹性能力，在饿了么场景，低峰期向云归还资源，高峰期重新申请相应资源。

我们可以使用 ASI 大资源池（注：ASI 资源池的宿主资源均来自于阿里云云资源）的内置弹性 Buffer，也可以直接使用阿里云 IaaS 层的弹性技术。二者的平衡是一个很有争议的话题，也是一个比较艺术的过程。

ASI 的分时调度更是将资源复用做到了极致，并带来了巨大的成本优化。通过每天晚上大规模停用在线交易 POD 实例，释放的资源用于 ODPS 离线任务使用，每天早上离线任务退水并重新拉起在线应用。这个经典场景更是将在离线混部技术的价值发挥到最大。

分时的精髓是资源复用，以及依赖的大资源池建设管理，这是资源运营 & 调度技术的综合。这需要调度器积累多多益善的丰富形态作业、以及多多益善的海量作业任务。

（四）垂直伸缩调度/X+1/VPA/HPA

垂直伸缩调度是一种秒级交付技术，非常完美地部分解决了突发流量问题。垂直伸缩调度也是大促零点峰值压力风险的杀手锏，通过对存量 POD 的资源垂直调整、准确可靠的 CPU 调度和洗牌算法来做到计算资源的秒级交付。垂直伸缩调度、VPA 技术一脉相承，垂直伸缩调度也是 VPA 的场景之一。

“X+1” 水平扩容调度在某种意义上也可以理解为 HPA 场景之一，只不过 “X+1” 水平扩容调度由手动触发。“X+1” 侧重于强调极致的资源交付效率，这背后是研发效率的极大提升：在线 POD“X（若干）”分钟可启动完毕提供业务服务；除应用启动外的所有其它操作，务必在 “1” 分钟内全部完成。

垂直伸缩调度与 “X+1” 水平扩容调度互为补充，共同为各类峰值保驾护航。ASI 也正在实施更多的 VPA 和 HPA 场景。例如，我们可以通过 VPA 技术，额外提供蚂蚁春节红包更多数量的免费算力，这将是非常大的成本节约。

VPA/HPA 等调度技术更多场景的极致实施，也是阿里巴巴未来将继续追求完善的地方。

（五）分等级[差异化 SLO]的资源调度

差异化 SLO 调度是调度器的精髓之一；这节内容与上文中的【调度资源类型】章节存在一定的重复。鉴于差异化 SLO 的复杂度，所以有意将它放在本章的最后一节来讲述。ASI 调度器内，也非常精确地定义了 SLO(服务质量目标)、QoS 和 Priority。

1）SLO

SLO 描述的是服务质量目标。ASI 通过不同的 QoS 和 Priority 来提供差异化 SLO，不同的 SLO 有不同的定价。用户可以根据不同的业务特性来决定"认购"哪类 SLO 保障的资源。如：离线的数据分析任务，则可以使用低等级的 SLO 以享受更低的价格。而对于重要业务场景则可以使用高等级的 SLO，当然价格也会更高。

2）QoS

QoS 描述了资源保障质量。K8s 社区定义的 QOS 包括 Guaranteed、Burstable、BestEffort。ASI 中定义的 QOS，与社区并没有进行完全映射（社区完全用 Request / Limit 来映射）。为了能将集团的场景（如 CPUShare，混部等）描述清晰，ASI 从另外一个维度定义了 QOS，它包括 LSE / LSR / LS / BE，清晰地划分出不同的资源保障，不同的业务根据自身的延迟敏感度可以选择不同的 QOS。

Qos	描述
LSE: Latency Sensitive Exclusive	CPU 核是 CPUSet 模式，且不和任何其他任务共享，包括 BE。
LSR: Latency Sensitive Reserved	CPU 核是 CPUSet 模式，不和其他的非 Best Effort 共享。
LS ： Latency Sensitive	Shared CPU，使用 LSE / LSR 之外的 CPU。
BE：Best Effort	使用所有除了 LSE 之外的核。

3）PriorityClass

PriorityClass 和 QoS 是两个维度的概念。PriorityClass 描述的则是任务的重要性。

资源分配策略和任务的重要性（即 PriorityClass 和 QoS）会有不同的组合，当然也需要存在一定的对应关系。例如，我们可以定义一个名为 Preemptible 的 PriorityClass，其大部分任务对应到 BestEffort 的 QoS。

各个调度系统对 PriorityClass 有不同的定义。例如：

- 在 ASI 中，ASI 的 priority 定义，目前定义了 System、Production、Preemptible、Production、Preemptible。这里不详细解读每个等级的细节。
- 搜索 Hippo 中定义的种类和粒度更细，包括：System、ServiceHigh、ServiceMedium、ServiceLow、JobHigh、JobMedium、JobLow 等。这里不详细解读每个等级的细节。

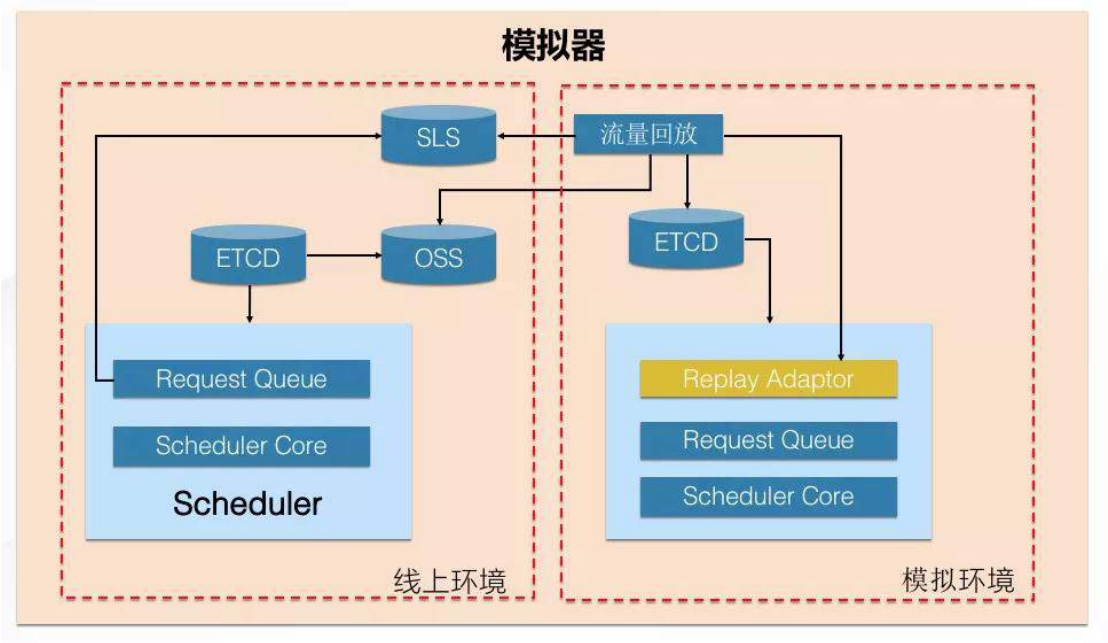
全局最优的调度

（一）调度模拟器

调度模拟器有点类似于阿里巴巴的全链路压测系统，通过线上真实的流量回放、或模拟流量回放，在模拟环境验证调度新的能力，进而不断地锤炼各种调度算法，优化各类指标。

调度模拟器的另一个常见的用途是，是对线上疑难杂症问题做线下模拟，做到无害、高效地定位各类问题。

一定程度上，调度模拟器是全局调度最优的基础。有了调度模拟器，我们得以在模拟环境，反复锤炼各种算法、技术框架、技术链路，进而做到全局指标的优化，例如：全局分配率、不同场景下的调度性能、调度稳定性等等。



（二）Elastic Scheduling Platform（ESP 平台）

为了做到全局最优的调度，围绕着调度器，ASI 构建了一套全新的 ElasticScheduling Platform（ESP 平台），旨在围绕调度器，打造基于调度数据指导 & 核心调度能力 & 产品化调度运营 的一站式自闭环调度效能系统。

在过去，我们已经建设了诸多类似模块，例如调度 SLO 巡检、众多调度工具、不同场景的二层调度平台等；而基于 ESP 平台，集合更多的二层调度能力，带给 ASI 全局最优的调度质量，并围绕 业务稳定性、资源成本、用户效能提升，带给客户更贴心的服务。

更多调度能力

本文尝试着系统地讲解 ASI 调度器的基本概念、原理和各种场景，并带领您走进调度器美丽精彩的世界。调度器博大精深，遗憾的是，受限于篇幅，也不得不控制篇幅，非常多的内容点到为止并未深入展开。在调度器内，还有更多更深层次的调度内幕，如异构机型调度、调度画像、公平性调度、优先级调度、腾挪调度、抢占调度、磁盘调度、Quota、CPU 归一化、GANG Scheduling、调度 Tracing、调度诊断等众多调度能力，本文均未予阐述。受限于篇幅，本文也未讲述 ASI 强大的调度框架结构及优化、调度性能优化等更多更深层次的技术内幕。

早在 2019 年，ASI 已优化 K8s 单集群至业界领先的万级节点规模，并得益于阿里云 ACK 强大的 K8s 运维体系，阿里集团内持续保有着数量众多的大规模计算集群，同时也积累了业界领先的 K8s 多集群生产实践。正是在这些大规模 K8s 集群内，ASI 基于完善的容器调度技术，持续为众多复杂的任务资源提供着计算资源算力。

在过去的数年，借助于集团全面上云的契机，阿里集团在调度领域，已实现了从 ASI 管控到阿里云容器服务 ACK 的全面迁移和进化。而阿里集团内复杂、丰富、规模化的业务场景，未来也将持续输出、增强并锤炼云的技术能力。

云原生趋势下的迁移与容灾思考

导读：下一个云原生颠覆的领域会不会是在传统的容灾领域呢？在云原生的趋势下，如何构建应用系统的迁移与容灾方案？

趋势

（一）云原生发展趋势

云原生（Cloud Native）是最近几年非常火爆的话题，在 2020 年 7 月由信通院发布的《云原生发展白皮书（2020）年》明确指出：云计算的拐点已到，云原生成为驱动业务增长的重要引擎。我们不难发现云原生带给 IT 产业一次重新洗牌，从应用开发过程到 IT 从业者的技术能力，都是一次颠覆性的革命。

在此基础上，出现了基于云原生平台的 Open Application Model 定义，在云原生平台基础上进一步抽象，更加关注应用而非基础架构。同时，越来越多的公有云开始支持 Serverless 服务，更加说明了未来的发展趋势：应用为核心，轻量化基础架构层在系统建设过程中的角色。但是无论如何变化，IT 整体发展方向，一定是向着更有利于业务快速迭代、满足业务需求方向演进的。

2020 年 9 月，Snowflake 以每股 120 美金 IPO，创造了今年规模最大的 IPO，也是有史以来最大的软件 IPO。Snowflake 利用云原生方式重构了数据仓库，成功颠覆了行业竞争格局。这正是市场对云原生发展趋势的最佳认可，所以下一个云原生颠覆的领域会不会是在传统的容灾领域呢？

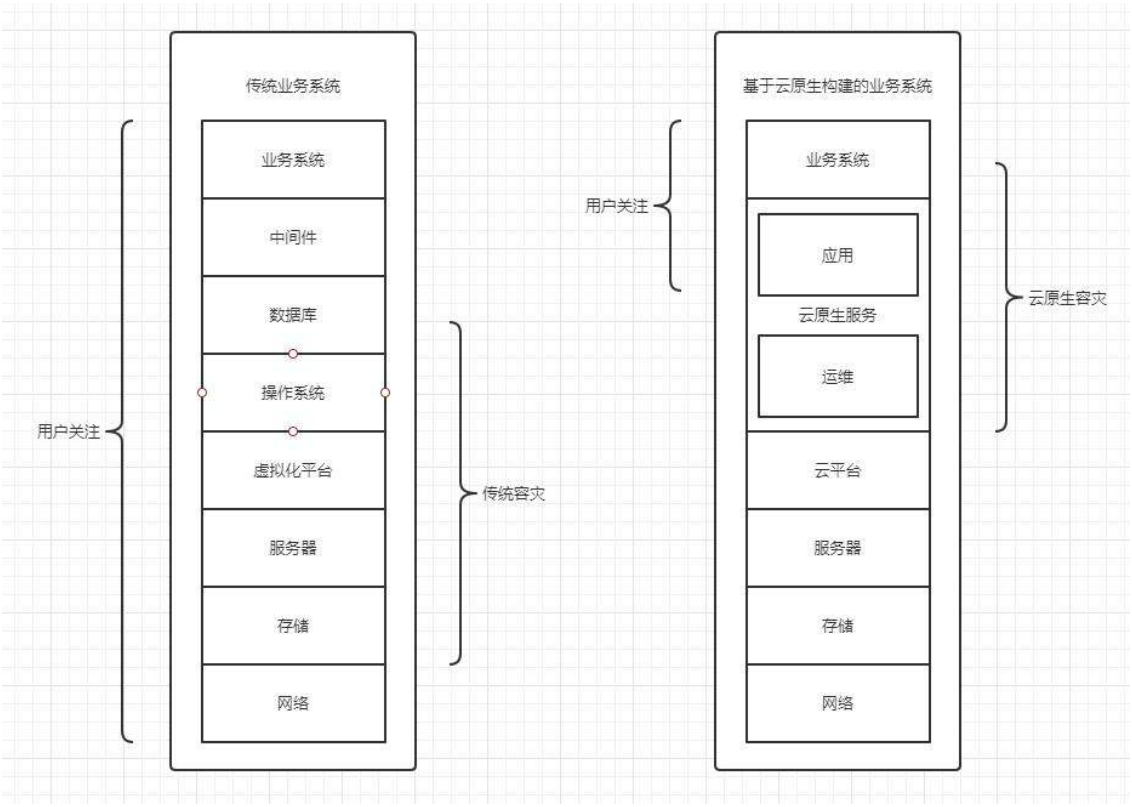
（二）为什么云上需要全新的迁移和容灾？

1）传统方案的局限性

在这种大的趋势下，传统的迁移和容灾仍然停留在数据搬运的层次上，而忽略了面向云的特性和用户业务重新思考和构建。云计算的愿景是让云资源像水、电一样按需使用，所以基于云上的迁移和容灾也理应顺应这样的历史潮流。Snowflake 也是通过这种商业模式的创新，成功打破旧的竞争格局。

为什么传统容灾的手段无法满足云原生需求呢？简单来说，二者关注的核心不同。传统的容灾往往以存储为核心，拥有对存储的至高无上的控制权。并且在物理时代，对于计算、存储和网络等基础架构层也没有有效的调度方法，无法实现高度自动化的编排。而基于云原生构建的应用，核心变成了云原生服务本身。当用户业务系统全面上云后，用户不再享有对底层存储的绝对控制权，所以传统的容灾手段，就风光不在了。

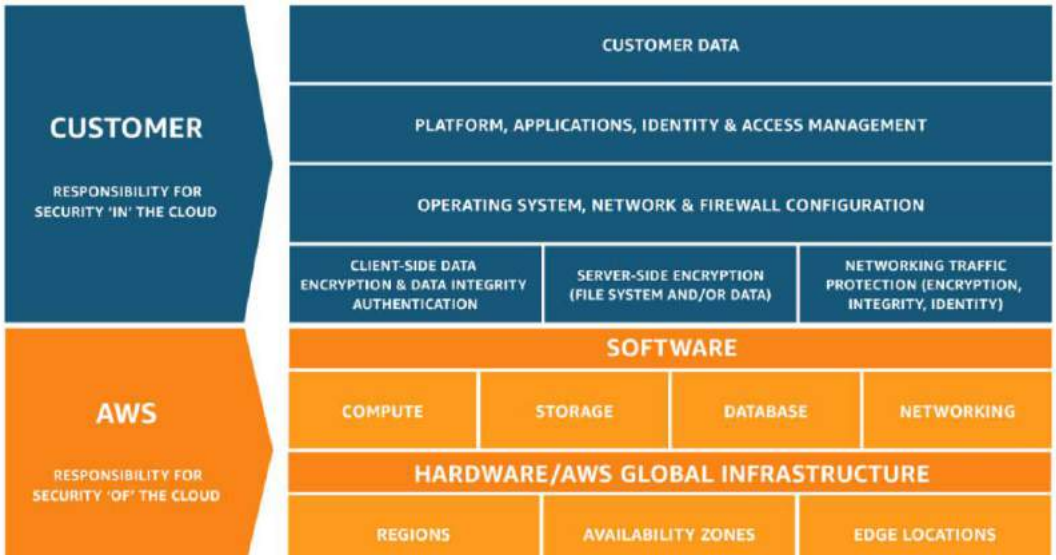
、



我认为在构建云原生容灾的解决方案上，要以业务为核心去思考构建方法，利用云原生服务的编排能力实现业务系统的连续性。

2）数据安全性

AWS CTO Werner Vogels 曾经说过：Everything fails, all the time。通过 AWS 的责任共担模型，我们不难发现云商对底层基础架构负责，用户仍然要对自身自身数据安全性和业务连续性负责。

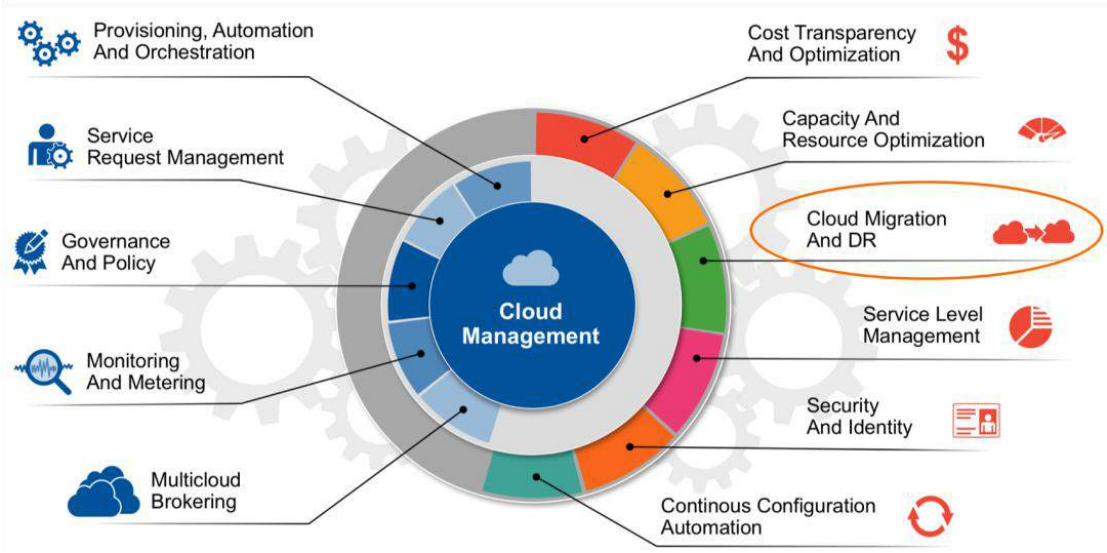


我认为在云原生趋势下，用户最直接诉求的来自数据安全性即备份，而迁移、恢复、高可靠等都是基于备份表现出的业务形态，而备份能力可能是由云原生能力提供的，也有可能是第三方能力提供的，但最终实现业务形态，是由编排产生的。

用户上云并不等于高枕无忧，相反用户要学习云的正确打开方式，才能最大程度来保证业务的连续性。虽然云在底层设计上是高可靠的，但是仍然避免不了外力造成的影响，例如：光缆被挖断、断电、人为误操作导致的云平台可用区无法使用，所以才有了类似“蓝翔决定了中国云计算稳定性”的调侃。我认为用户决定将业务迁移到云上的那一刻开始，备份、迁移、恢复、高可靠是一个连续的过程，如何合理利用云原生服务的特性实现业务连续性，同时进行成本优化，降低总体拥有成本（TCO）。

3）防止厂商锁定

某种意义上说，云原生的方向是新一轮厂商锁定，就像当年盛极一时的 IOE 架构一样，只不过现在换成了云厂商作为底座承载应用。在 IOE 时代，用户很难找到完美的替代品，但是在云时代，这种差异并不那么明显。所以大部分的客户通常选用混合云作为云建设策略，为了让应用在不同云之间能够平滑移动，利用容灾技术的迁移一定是作为一个常态化需求存在的。Gartnar 也在多云管平台定义中，将迁移和 DR 作为单独的一项能力。充分说明迁移与容灾在多云环境的的常态化趋势。



云迁移与云容灾的关系

（一）云迁移需求的产生

在传统环境下，迁移的需求并不十分突出，除非是遇到机房搬迁或者硬件升级，才会想到迁移，但这里的迁移更像是搬铁，迁移工具化与自动化的需求并不明显。当 VMware 出现后，从物理环境到虚拟化的迁移需求被放大，但由于是单一的虚拟化平台，基本上虚拟化厂商自身的工具就完全能够满足需求了。在虚拟化平台上，大家突然发现原来只能人工操作的物理环境一下子轻盈起来，简单来说，我们的传统服务器从一堆铁变成了一个文件，并且这个文件还能够被来回移动、复制。再后来，进入云时代，各家云平台风生水起，国内云计算市场更是百家争鸣，上云更是成为了一种刚性需求。随着时间的推移，出于对成本、厂商锁定等诸多因素的影响，在不同云之间的互相迁移更是会成为一种常态化的需求。

（二）底层技术一致

这里提到的云迁移和容灾，并不是堆人提供的迁移服务，而是强调的高度自动化的手段。目标就是在迁移过程中保证业务连续性，缩短停机时间甚至不停机的效果。这里就借助了容灾的存储级别同步技术来实现在异构环境下的的“热迁移”。现有解决方案里，既有传统物理机搬迁时代的迁移软件，也有基于云原生开发的工具。但无论何种形式，都在不同程度上都解决了用户上云的基本诉求。最大的区别在于人效比，这一点与你的利益直接相关。

从另外一个角度也不难发现，所谓的迁移在正式切换之前实质上就是容灾的中间过程。同时，业务系统迁移到云平台后，灾备是一个连续的动作，这里既包含了传统的备份和容灾，还应该包含云上高可靠的概念。这样，用户业务系统在上云后，才能摆脱传统基础架构的负担，做到“零运维”，真正享受到云所带来的的红利。所以，我认为在云原生状态下，云迁移、云容灾、云备份本质上就是一种业务形态，底层采用的技术手段可以是完全一致的。

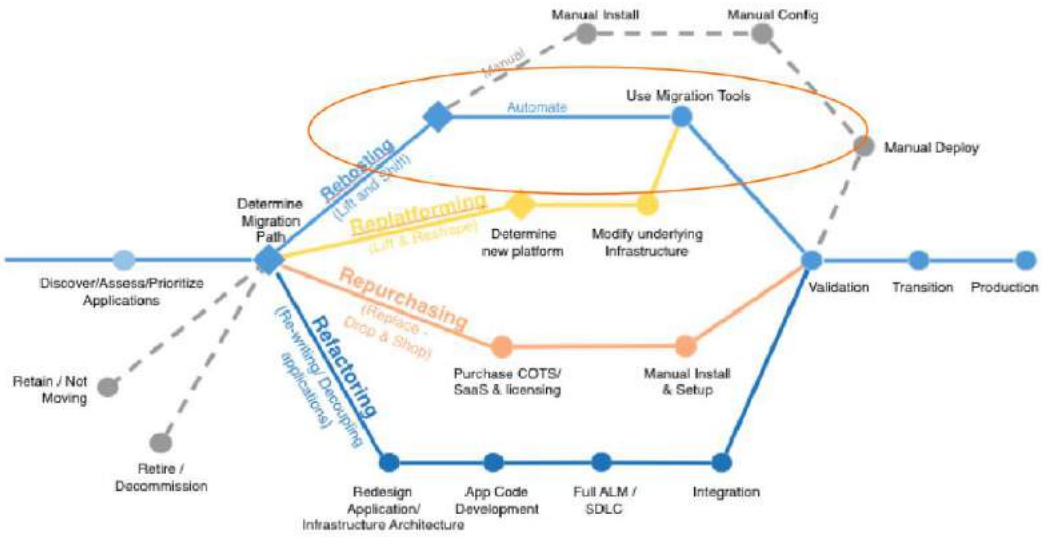
（三）发展方向

在上述的痛点和趋势下，必然会出现一种全新的平台来帮助客户解决数据的安全性和业务连续性问题，今天就从这个角度来分析一下，在云原生的趋势下如何构建应用系统的迁移与容灾方案。

云迁移发展趋势

（一）云迁移方式

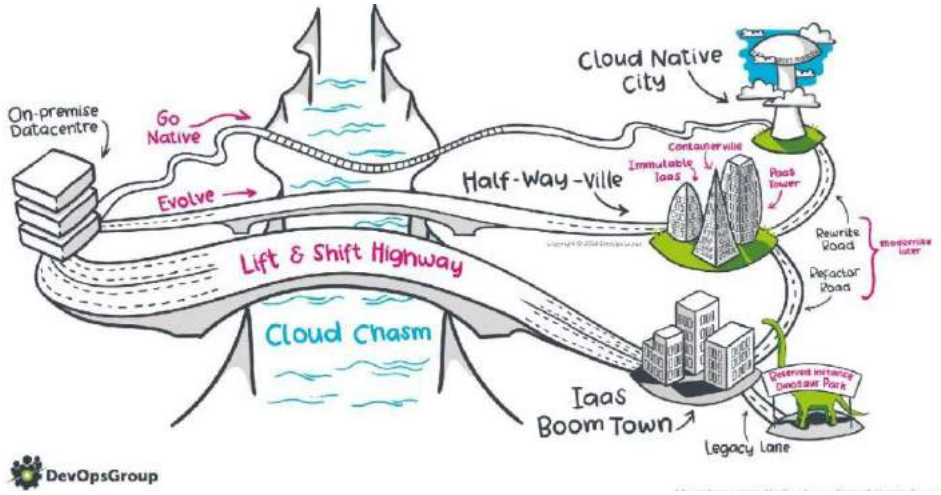
迁移是一项重度的咨询业务，网上各家云商、MSP 都有自己的方法论，其实看下来差别都不大，之前也有很多人在分享相关话题，本文就不再赘述。这里我们重点讨论，在实际落地过程中到底该采用哪种工具，哪种方式的效率最高。所谓云迁移工具，就是将源端迁移至目标端，保证源端在目标端正确运行。常见的方式包括：物理机到虚拟化、虚拟化到虚拟化、物理机到云平台、虚拟化到云平台等。



这是经典的 6R 迁移理论（现在已经升级为了 7R，多了 VMware 出来搅局），在这个图中与真正迁移相关的其实只有 Rehosting、Replatforming、Repurchasing 和 Refactoring，但是在这 4R 中，Refactoring 明显是一个长期的迭代过程，需要用户和软件开发商共同参与解决，Repurchasing 基本上与人为重新部署没有太大的区别。所以真正由用户或 MSP 在短期完成的只剩下 Rehosting 和 Replatofrming。

与上面这张经典的迁移理论相比，我更喜欢下面这张图，这张图更能反应一个传统应用到云原生成长的全过程。与上述的结论相似，我们在真正拥抱云的时候，路径基本为上述的三条：

- Lift & Shift 是 Rehost 方式的另一种称呼，这种方式路面最宽，寓意这条路是上云的最短路径，应用不需要任何改造直接上云使用。
- Evolve 和 Go Native 都属于较窄的路径，寓意为相对于 Rehost 方式，这两条路径所消耗的时间更久，难度更高。
- 在图的最右侧，三种形态是存在互相转换的可能，最终演进为彻底的云原生，寓意为迁移并不是一蹴而就，需要循序渐进完成。



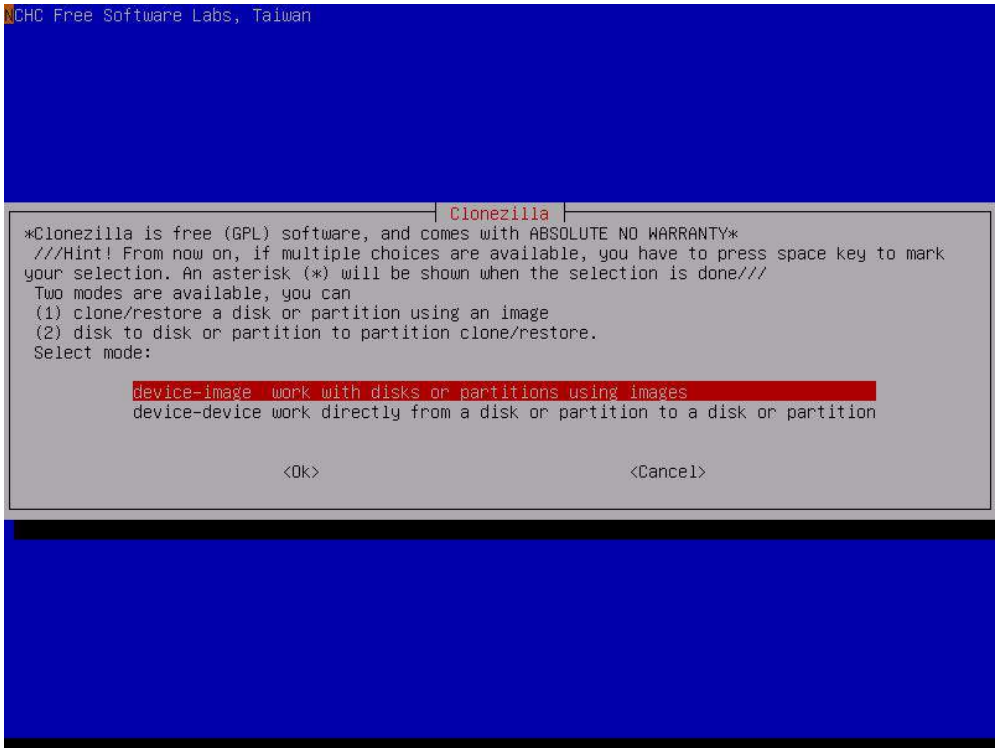
（二）重新托管（Rehost）方式

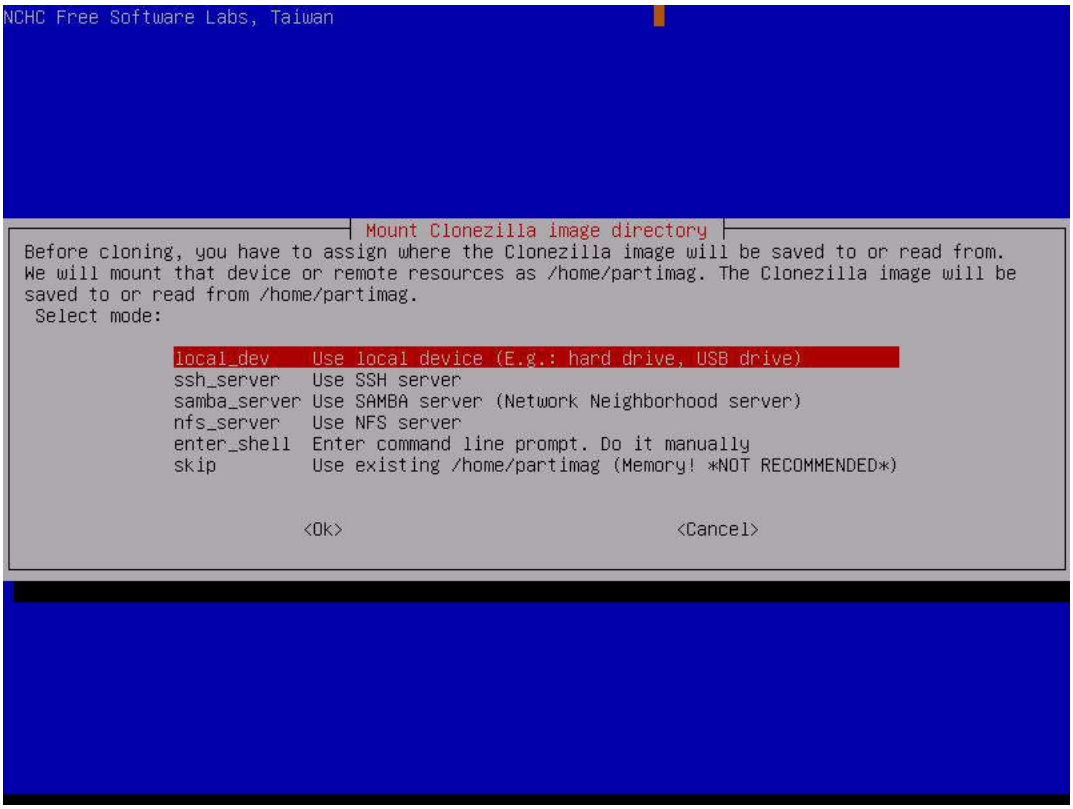
常用的重新托管方式为冷迁移和热迁移，冷迁移往往涉及到步骤比较繁琐，需要大量人力投入，并且容易出错效率低，对业务连续性有较大的影响，不适合生产系统迁移。而热迁移方案基本都是商用化的解决方案，这里又分为块级别和文件级别，再细分为传统方案与云原生方案。

1）冷迁移

我们先来看一下冷迁移的手动方案，以 VMware 到 OpenStack 为例，最简单的方式就是将 VMware 虚拟机文件（VMDK）通过 qemu-img 工具进行格式转换，转换为 QCOW2 或者 RAW 格式，上传至 OpenStack Glance 服务，再重新在云平台上进行启动。当然这里面需要进行 virtio 驱动注入，否则主机无法正常在云平台启动。这个过程中最耗时的应该是虚拟机文件上传至 OpenStack Glance 服务的过程，在我们最早期的实践中，一台主机从开始迁移到启动完成足足花了 24 小时。同时，在你迁移这段时间的数据是有增量产生的，除非你将源端关机等待迁移完成，否则，你还要将上述步骤重新来一遍。所以说这种方式真的不适合有业务连续性的生产系统进行迁移。

那如果是物理机的冷迁移方案怎么做呢？经过我们的最佳实践，这里为大家推荐的是老牌的备份工具 CloneZilla，中文名为再生龙。是一款非常老牌的备份软件，常用于进行整机备份与恢复，与我们常见的 Norton Ghost 原理非常相似。CloneZilla 从底层的块级别进行复制，可以进行整盘的备份，并且支持多种目标端，例如我们将磁盘保存至移动硬盘，实际格式就是 RAW，你只需要重复上述的方案即可完成迁移。但是在使用 CloneZilla 过程中，需要使用 Live CD 方式进行引导，同样会面临长时间业务系统中断的问题，这也是上面我们提到的冷迁移并不适合生产环境迁移的原因。





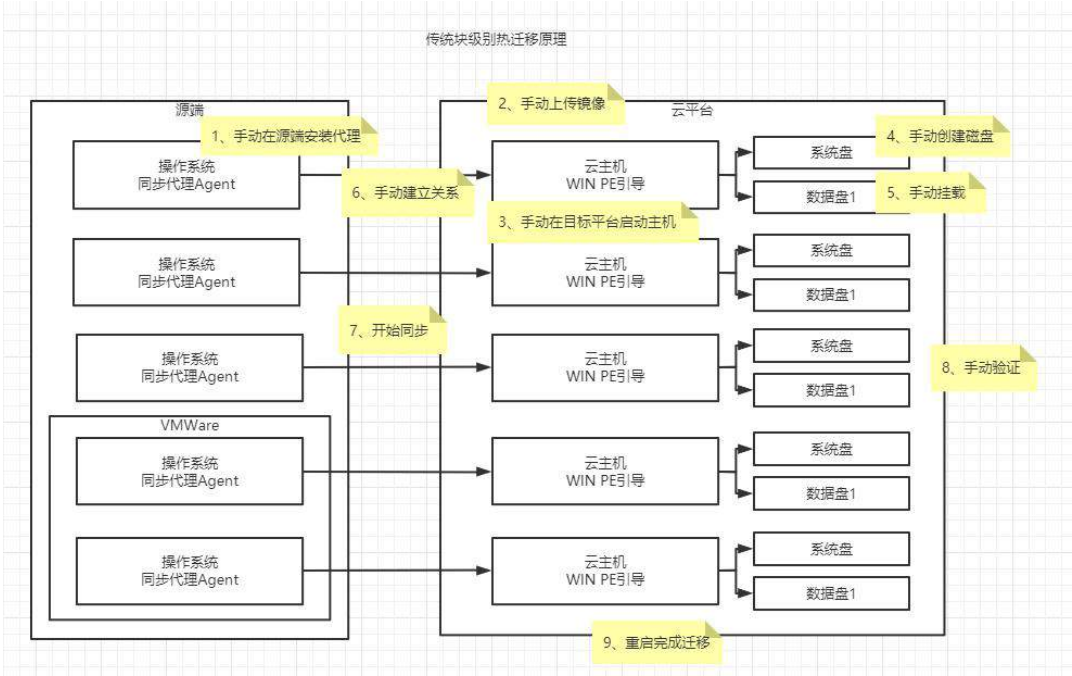
2）传统热迁移方案

传统的热迁移方案基本分为块级别和文件级别，两者相似之处都是利用差量同步技术进行实现，即全量和增量交叉同步方式。

文件级别的热迁移方案往往局限性较大，并不能算真正的 ReHost 方式，因为前期需要准备于源端完全一样的操作系统，无法实现整机搬迁，从操作的复杂性更大和迁移的稳定性来说都不高。我们在 Linux 上常用的 Rsync 其实可以作为文件级别热迁移的一种解决方案。

真正可以实现热迁移的方案，还要使用块级别同步，降低对底层操作系统依赖，实现整机的搬迁效果。传统的块级别热迁移方案基本上来自于传统容灾方案的变种，利用内存操作系统 WIN PE 或其他 Live CD 实现，基本原理和过程如下图所示。从过程中我们不难发现这种方式虽然在一定程度解决了迁移的目标，但是作为未来混合云常态化迁移需求来说，仍然有以下几点不足：

- 由于传统热迁移方案是基于物理环境构建的，所以我们发现在整个过程中人为介入非常多，对于使用者的技能要求比较高。
- 无法满足云原生时代多租户、自服务的需求。
- 安装代理是用户心中永远的芥蒂。
- 一比一同步方式，从成本角度来说不够经济。
- 最好的迁移验证方式，就是将业务系统集群在云端完全恢复，但是手动验证的方式，对迁移人力成本是再一次增加。

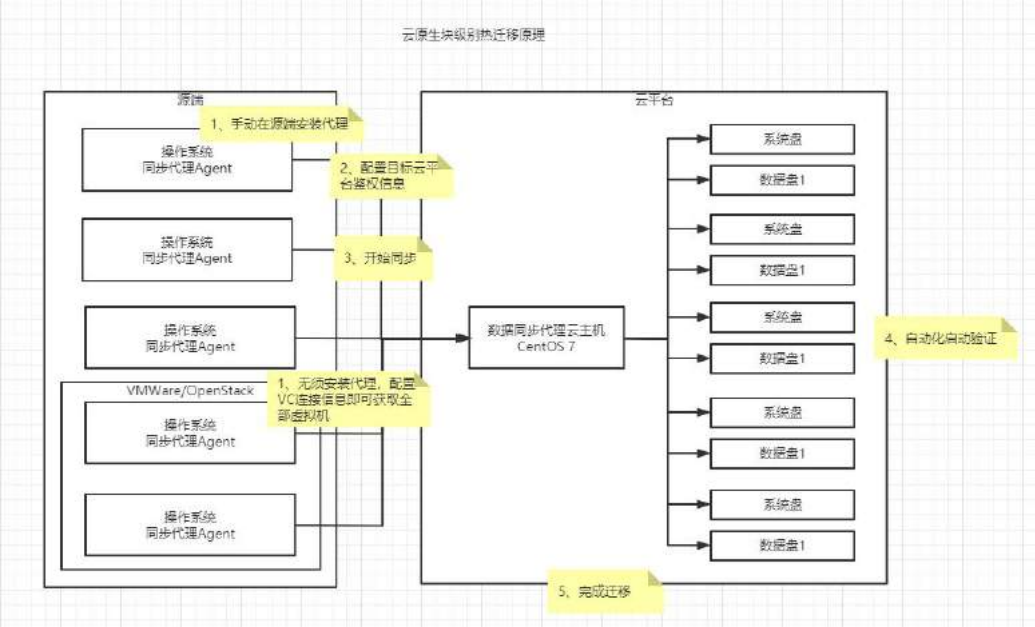


3）云原生热迁移方案

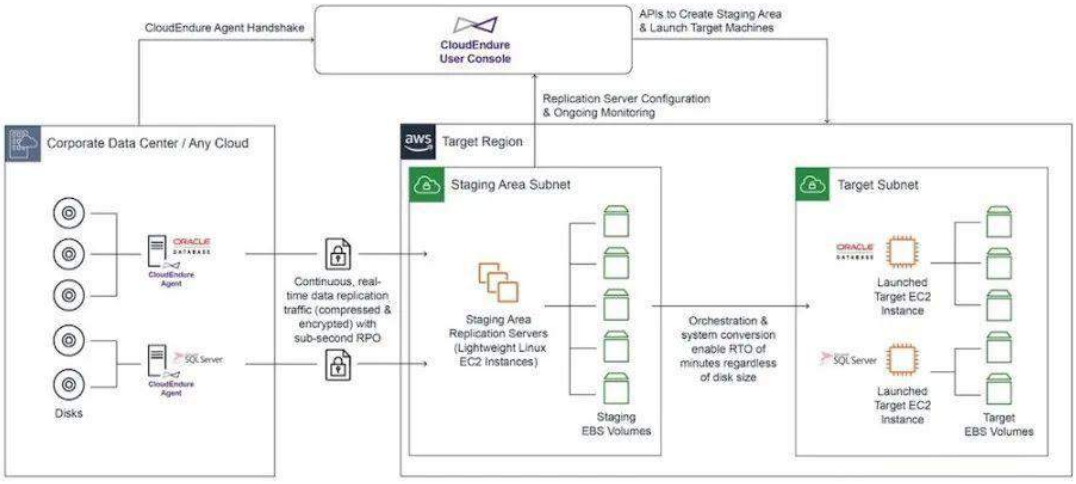
正是由于传统迁移方案的弊端，应运而生了云原生的热迁移方案，这一方面的代表厂商当属 AWS 在 2019 年以 2.5 亿美金击败 Google Cloud 收购的以色列云原生容灾、迁移厂商 CloudEndure。

云原生热迁移方案是指利用块级别差量同步技术结合云原生 API 接口和资源实现高度自动化迁移效果，同时提供多租户、API 接口满足混合云租户自服务的需求。我们先从原理角度分析一下，为什么相对于传统方案，云原生的方式能够满足高度自动化、用户自服务的用户体验。通过两个方案对比，我们不难发现云原生方式的几个优势：

- 利用云原生 API 接口和资源，操作简便，完全取代了传统方案大量繁琐的人为操作，对使用者技术要求降低，学习陡峭程度大幅度降低。
- 由于操作简便，迁移效率提高，有效提高迁移实施的人效比。
- 一对多的同步方式，大幅度降低计算资源使用，计算资源只在验证和最终切换时使用
- 能够满足多租户、自服务的要求。
- 源端也可以支持无代理方式，打消用户疑虑，并且适合大规模批量迁移。
- 高度自动化的验证手段，在完成迁移切换前，能够反复进行验证。



这是 CloudEndure 的架构图，当然你也可以利用 CloudEndure 实现跨区域的容灾。



不过可惜的一点是由于被 AWS 收购，CloudEndure 目前只能支持迁移至 AWS，无法满足国内各种云迁移的需求。所以这里为大家推荐一款纯国产化的迁移平台——万博智云的 HyperMotion，从原理上与 CloudEndure 非常相似，同时支持了 VMware 及 OpenStack 无代理的迁移，更重要的是覆盖了国内主流的公有云、专有云和私有云的迁移。

（三）平台重建（Replatforming）方式

随着云原生提供越来越多的服务，降低了应用架构的复杂度，使得企业能够更专注自己的业务本身开发。但是研发侧工作量的减少意味着这部分成本被转嫁到部署及运维环节，所以 DevOps 成为在云原生运用中比不可少的一个缓解，也让企业能够更敏捷的应对业务上的复杂变化。

正如上面所提到的，用户通过少量的改造可以优先使用一部分云原生服务，这种迁移方式我们成为平台重建（Replatforming），目前选择平台重建方式的迁移，多以与用户数据相关的服务为主。常见的包括：数据库服务 RDS、对象存储服务、消息队列服务、容器服务等。这些云原生服务的引入，降低了用户运维成本。但是由于云原生服务自身封装非常严密，底层的基础架构层对于用户完全不可见，所以无法用上述 Rehost 方式进行迁移，必须采用其他的辅助手段完成。

以关系型数据库为例，每一种云几乎都提供了迁移工具，像 AWS DMS，阿里云的 DTS，腾讯云的数据传输服务 DTS，这些云原生工具都可以支持 MySQL、MariaDB、PostgreSQL、Redis、MongoDB 等多种关系型数据库及 NoSQL 数据库迁移。以 MySQL 为例，这些服务都巧妙的利用了 binlog 复制的方式，实现了数据库的在线迁移。

再以对象存储为例，几乎每一种云都提供了自己的迁移工具，像阿里云的 ossimport，腾讯云 COS Migration 工具，都可以实现本地到云端对象存储的增量迁移。但是在实际迁移时，还应考虑成本问题，公有云的对象存储在存储数据上比较便宜，但是在读出数据时是要根据网络流量和请求次数进行收费的，这就要求我们在设计迁移方案时，充分考虑成本因素。如果数据量过大，还可以考虑采用离线设备方式，例如：AWS 的 Snowball，阿里云的闪电立方等。这部分就不展开介绍，以后有机会再单独为大家介绍。

上云场景	方案	迁移过程产生的费用情况	适用场景
IDC到云上	直接传输	对象存储容量和使用时间 对象存储请求次数	小规模上云
	数据传输设备	对象存储容量和使用时间 对象存储请求次数 每个作业的服务费 每个作业超出时限后的每天费用（10天内免费） 数据传输（免费）	大批量迁移
	普通存储设备 + CK网络	对象存储容量和使用时间 对象存储请求次数 设备租赁费用 CK网络费用	大批量迁移
同云不同区域	跨区域复制	对象存储容量和使用时间 对象存储请求次数 跨区域复制流量	原生方式，稳定性和效率有保障
同云同区域跨租户	内网直接传输	目标租户对象存储容量和使用时间 对象存储请求次数	均适用
不同云间迁移/云上到IDC	直接传输	对象存储容量和使用时间 对象存储请求次数 外网流出流量	小规模上云
	在ECS内安装迁移工具	对象存储容量和使用时间 对象存储请求次数 ECS费用 ECS网络费用	大批量迁移
	普通存储设备 + CK网络	对象存储容量和使用时间（源端 + 目标端） 对象存储请求次数 设备租赁费用 CK网络费用	大批量迁移

如果选择平台重建方式上云，除了要进行必要的应用改造，还需要选择一款适合你的迁移工具，保证数据能够平滑上云。结合上面的 Rehost 方式迁移，能够实现业务系统的整体上云效果。由于涉及的服务较多，这里为大家提供一张迁移工具表格供大家参考。

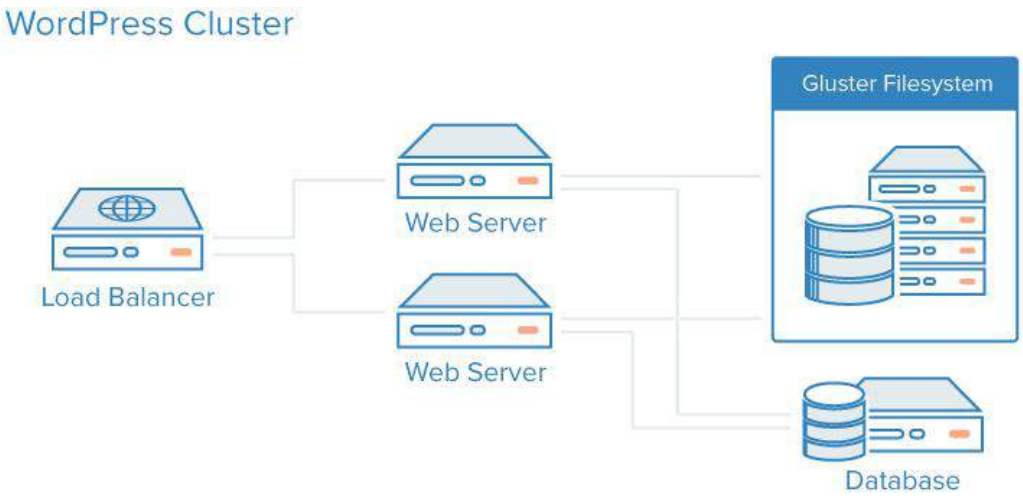
工具类型	名称	说明
开源	mysqldump/mongodump/pg_dump	MySQL/MongoDB/PostgreSQL数据库的备份工具
开源	gh-ost https://github.com/github/gh-ost 项目活跃	MySQL在线Schema迁移工具，无Tirgger，使用了binary log stream
开源	canal https://github.com/alibaba/canal 项目活跃	阿里开源项目，canal [kə'næl]，译意为水道/管道/沟渠，主要用途是基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费
开源	yugong https://github.com/alibaba/yugong 不活跃	愚公移山，阿里巴巴去Oracle数据迁移同步工具(全量+增量,目标支持MySQL/DRDS)
开源	datax https://github.com/alibaba/DataX 项目活跃	DataX 是阿里巴巴集团内被广泛使用的离线数据同步工具/平台，实现包括 MySQL、Oracle、SqlServer、Postgre、HDFS、Hive、ADS、HBase、TableStore(OTS)、MaxCompute(ODPS)、DRDS 等各种异构数据源之间高效的数据同步功能。
云商	AWS DMS	AWS Database Migration Service 支持同构迁移（例如从 Oracle 迁移至 Oracle），以及不同数据库平台之间的异构迁移（例如从 Oracle 或 Microsoft SQL Server 迁移至 Amazon Aurora）。
云商	阿里云DTS	数据传输服务DTS（Data Transmission Service）是阿里云提供的实时数据流服务，支持RDBMS、NoSQL、OLAP等，集数据迁移/订阅/同步于一体，为您提供稳定安全的传输链路。支持MySQL、SQL Server、Oracle、PostgreSQL、MongoDB、Redis、DB2
云商	阿里云ADAM	是一款把数据库和应用迁移到阿里云（公共云或专有云）的产品，显著地降低了上云的技术难度和成本，尤其是Oracle数据库应用。ADAM全面评估上云可行性、成本和云存储选型，内置实施脚助，数据、应用迁移等工具，确保可靠、快速上云。

云原生下的容灾发展趋势

目前为止，还没有一套平台能够完全满足云原生状态下的统一容灾需求，我们通过以下场景来分析一下，如何才能构建一套统一的容灾平台满足云原生的需求。

（一）传统架构

我们以一个简单的 Wordpress + MySQL 环境为例，传统下的部署环境一般是这样架构的：



如果为这套应用架构设计一套容灾方案，可以采用以下的方式：

1）负载均衡节点容灾

负载均衡分为硬件和软件层面，硬件负载均衡高可靠和容灾往往通过自身的解决方案实现。如果是软件负载均衡，往往需要安装在基础操作系统上，而同城的容灾可以使用软件高可靠的方式实现，而异地的容灾往往是通过提前建立对等节点，或者干脆采用容灾软件的块或者文件级别容灾实现。是容灾切换（Failover）很重要的一个环节。

2）Web Server 的容灾

Wordpress 的运行环境无非是 Apache + PHP，由于分离了用于存放用户上传的文件系统，所以该节点几乎是无状态的，通过扩展节点即可实现高可靠，而异地容灾也比较简单，传统的块级别和文件级别都可以满足容灾的需求。

3）共享文件系统的容灾

图中采用了 Gluster 的文件系统，由于分布式系统的一致性通常由内部维护，单纯使用块级别很难保证节点的一致性，所以这里面使用文件级别容灾更为精确。

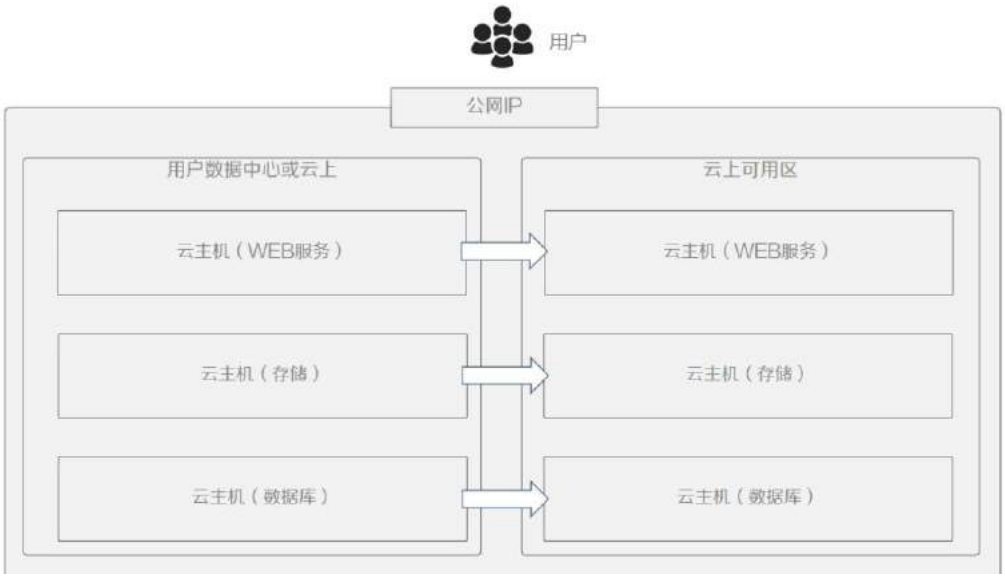
4）数据库的容灾

单纯依靠存储层面是无法根本实现数据库 0 丢失数据的，所以一般采用从数据库层面实现，当然如果为了降低成本，数据库的容灾可以简单的使用周期 Dump 数据库的方式实现，当然如果对可靠性要求较高，还可以使用 CDP 方式实现。

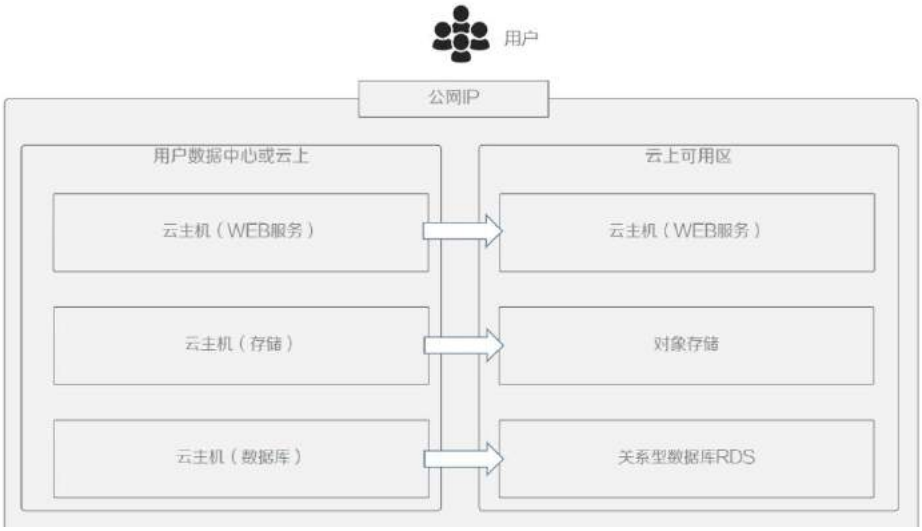
从以上的案例分析不难看出，传统基础架构下的容灾往往以存储为核心，无论是磁盘阵列的存储镜像，还是基于 I/O 数据块、字节级的捕获技术，结合网络、数据库和集群的应用级别技术完成高可靠和容灾体系的构建。在整个容灾过程的参与者主要为：主机、存储、网络和应用软件，相对来说比较单一。所以在传统容灾方案中，如何正确解决存储的容灾也就成为了解决问题的关键。

（二）混合云容灾

这应该是目前最常见的混合云的方案，也是各大容灾厂商主推的一种方式。这里我们相当于将云平台当成了一套虚拟化平台，几乎没有利用云平台任何特性。在恢复过程中，需要大量人为的接入才能将业务系统恢复到可用状态。这样的架构并不符合云上的最佳实践，但的确是很多业务系统备份或迁移上云后真实的写照。

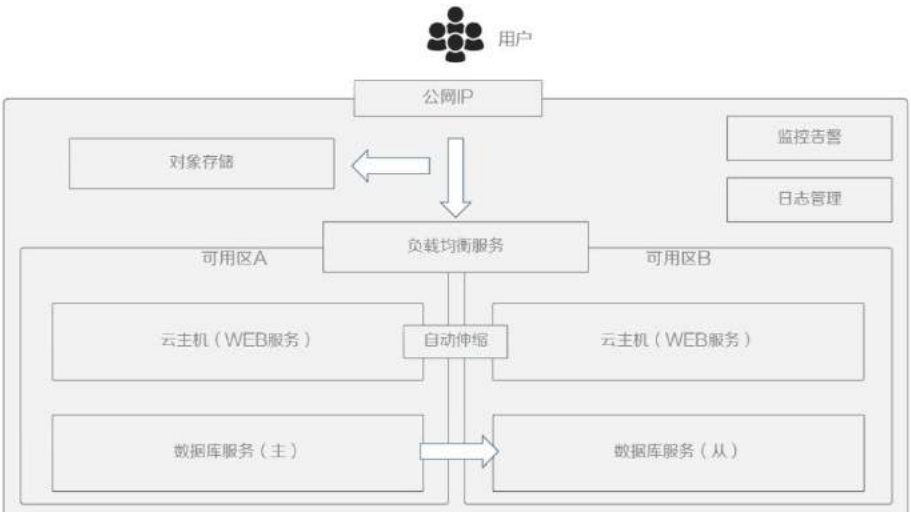


这样的架构确实能解决容灾的问题，但是从成本上来说很高，现在我们来换一种方式。我们利用了对象存储和数据库进行一次优化。我们将原有存储服务存放至对象存储中，而使用数据传输服务来进行实时的数据库复制。云主机仍然采用传统的块级别进行同步。一旦出现故障，则需要自动化编排能力，重新将备份进行恢复，在最短时间内根据我们预设的方案进行恢复，完成容灾。



（三）云上同城容灾架构

上述的备份方式，实质上就是利用平台重建的方式进行的迁移，既然已经利用迁移进行了备份，那完全可以对架构进行如下改造，形成同城的容灾架构。我们根据云平台的最佳实践，对架构进行了如下调整：



这个架构不仅实现了应用级高可靠，还能够支撑一定的高并发性，用户在最少改造代价下就能够在同城实现双活的效果。我们来分析一下在云上利用了多少云原生的服务：

- 域名解析服务
- VPC 服务
- 负载均衡服务
- 自动伸缩服务
- 云主机服务
- 对象存储服务
- 关系型数据库 RDS 服务

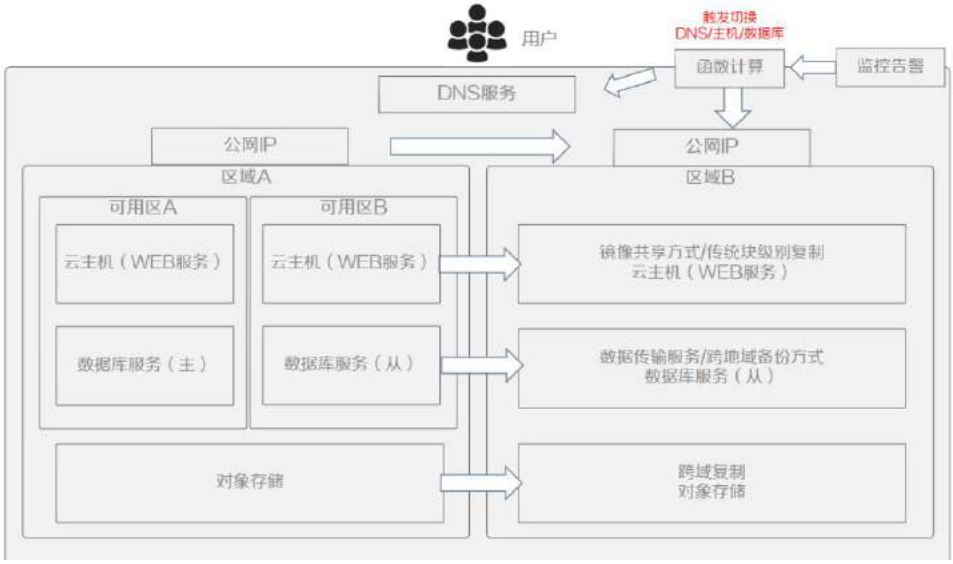
除了云主机外，其他服务均是天然就支持跨可用区的高可用特性，对于云主机我们可以制作镜像方式，由自动伸缩服务负责实例的状态。由于云上可用区就是同城容灾的概念，这里我们就实现了同城的业务系统容灾。

经过调整的架构在一定程度上满足了业务连续性的要求，但是对于数据的安全性仍然缺乏保障。近几年，勒索病毒横行，大量企业为此蒙受巨大损失，所以数据备份是上云后必须实施的。云原生服务本身提供了备份方案，例如云主机的定期快照等，但往往服务比较分散，不容易统一进行管理。同时，在恢复时往往也是只能每一个服务进行恢复，如果业务系统规模较大，也会增加大量的恢复成本。虽然云原生服务解决了自身备份问题，但是将备份重新组织成应用是需要利用自动化的编排能力实现。

（四）同云异地容灾架构

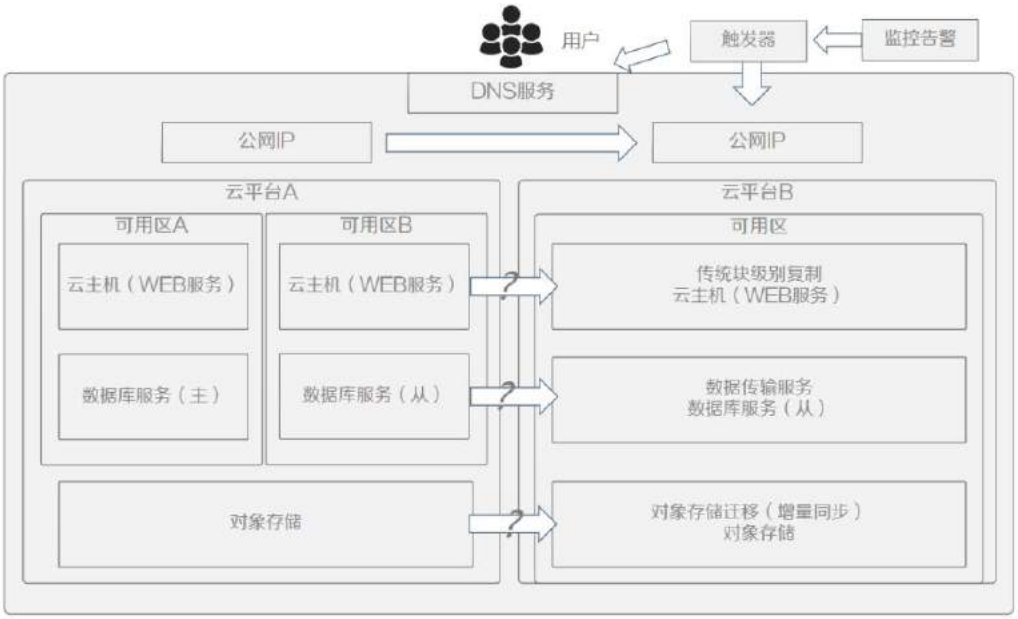
大部分的云原生服务都在可用区内，提供了高可靠能力，但是对于跨区域上通常提供的是备份能力。例如：可以将云主机变为镜像，将镜像复制到其他区域内；关系型数据库和对象存储也具备跨域的备份能力。利用这些组件自身的备份能力，外加上云自身资源的编排能力，我们可以实现在容灾可用域将系统恢复至可用状态。那如何触发切换呢？

这里我们根据业务系统的特点，在云原生的监控上定制告警，利用告警平台的触发能力触发函数计算，完成业务系统的跨域切换，形成异地容灾的效果。



（五）跨云容灾

但跨云容灾不像同云容灾时，在不同的可用区之间至少服务是一致的，那么此时，在同云上使用的方法基本失效，完全需要目标云平台的能力或者中立的第三方的解决方案。这里除了数据的备份，还有一点是服务配置的互相匹配。才能完全满足跨云容灾恢复的需求。另外需要考虑的一点就是成本为例，以对象存储为例，是典型的的“上云容易下云难”。所以如何利用云原生资源特性合理设计容灾方案是对成本的极大考验。



总结

云原生容灾还处于早期阶段，目前尚没有完整的平台能够支持以上各种场景的容灾需求，是值得持续探索的话题。云原生容灾以备份为核心，以迁移、恢复和高可靠为业务场景，实现多云之间的自由流转，最终满足用户的业务需求。

所以，作为面向云原生的容灾平台要解决好三方面的能力：

- 以数据为核心，让数据在多云之间互相流转。数据是用户核心价值，所以无论底层基础架构如何变化，数据备份一定是用户的刚醒需求。对于不同云原生服务如何解决好数据备份，是数据流转的必要基础。

- 利用云原生编排能力，实现高度自动化，在数据基础上构建业务场景。利用自动化编排能力实现更多的基于数据层的应用，帮助用户完成更多的业务创新。

- 灵活运用云原生资源特点，降低总体拥有成本。解决传统容灾投入巨大的问题，让用户的成本真的能像水、电一样按需付费。

第三章 双 11 云原生实践

本章主要作者： 覃柳杰、王炳燊、周金龙、丁超、刘佳旭、何以然、朱鹏

注：作者姓名按文章顺序排列

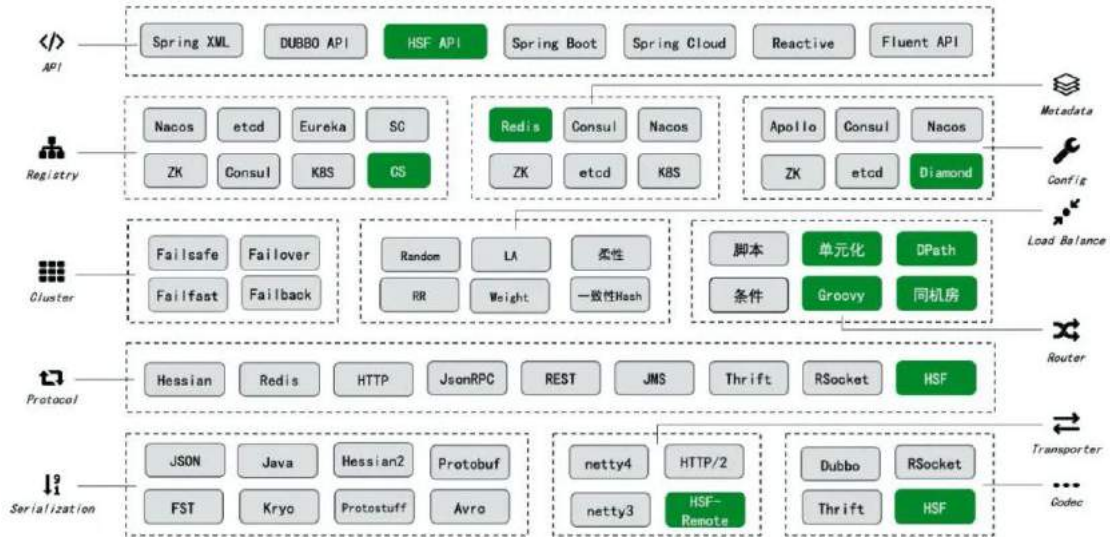
2020 双 11，Dubbo3.0 在考拉的超大规模实践

外部开发者一直以来好奇的是：阿里自己有没有在用 Dubbo？会不会用 Dubbo？在刚刚结束的 双 11，我们了解到阿里云今年提出了“三位一体”的理念，即将“自研技术”、“开源项目”、“商业产品”形成统一的技术体系，最大化技术的价值。终于，在 2020 的 双 11，阿里巴巴也用上了 Dubbo！本文是阿里 双 11 在考拉大规模落地 Dubbo3.0 的技术分享，系统介绍了 Dubbo3.0 在性能、稳定性上对考拉业务的支撑。

HSF 是阿里内部的分布式的服务框架，作为集团中间件最重要的中间件之一，历经十多届 双 11 大促，接受万亿级别流量的锤炼，十分的稳定与高效。另外一方面，Dubbo 是由阿里中间件开源出来的另一个服务框架，并且在 2019 年 5 月以顶级项目身份从 Apache 毕业，坐稳国内第一开源服务框架宝座，拥有非常广泛的用户群体。

在集团业务整体上云的大背景下，首要挑战是完成 HSF 与 Dubbo 的融合，以统一的服务框架支持云上业务，同时在此基础上衍生出适应用下一代云原生的服务框架 Dubbo 3.0，最终实现自研、开源、商业三位一体的目标。今年作为 HSF&Dubbo 融合之后的 Dubbo 3.0 在集团 双 11 落地的第一年，在兼容性、性能、稳定性上面都面临着不少的挑战。可喜的是，在今年 双 11 在考拉上面大规模使用，表现稳定，为今后在集团大规模上线提供了支撑。

Dubbo 3.0 总体方案



Dubbo 核心整体方案

在上面的方案中，可以看出来我们是以 Dubbo 为核心，HSF 作为功能扩展点嵌入到其中，同时保留 HSF 原有的编程 API，以保证兼容性。为什么我们选择以 Dubbo 为核心基础进行融合，主要基于以下两点的考量：

- Dubbo 在外部拥有非常广泛的群众基础，以 Dubbo 为核心，符合开源、商业化的目标。
- HSF 也经历过核心升级的情况，我们拥有比较丰富的处理经验，对于 Dubbo 3.0 新核心内部落地也是处于可控的范围之内。

选定这个方案之后，我们开始朝着这个方向努力，由于 Dubbo 开源已久，不像 HSF 这样经历过超大规模集群的考验验证，那么我们是如何去保证它的稳定性呢？

稳定性

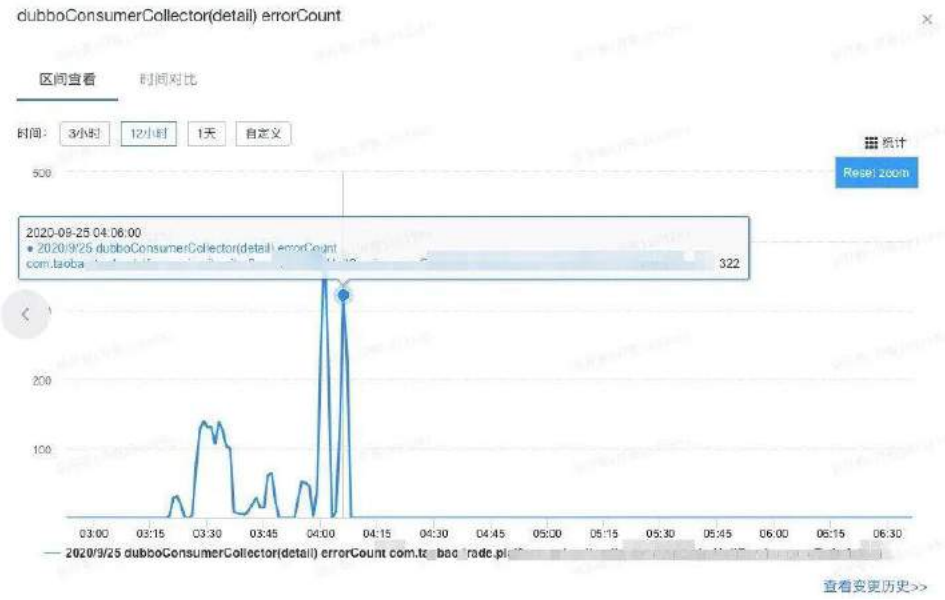
为了保证新核心的稳定，我们从各个方面进行巩固，保证万无一失。

（一）功能测试

HSF3 共有集成用例数百个，100% 覆盖到了 HSF 的核心功能；HSF3 的单测共有上千个，行覆盖率达到了 51.26%。

（二）混沌测试

为了面对突发的异常情况，我们也做了相应的演练测试，例如 CS 注册中心地址停推空保护测试、异常注入、断网等情况，以此验证我们的健壮性；例如，我们通过对部份机器进行断网，结果我们发现有比较多的异常抛多。

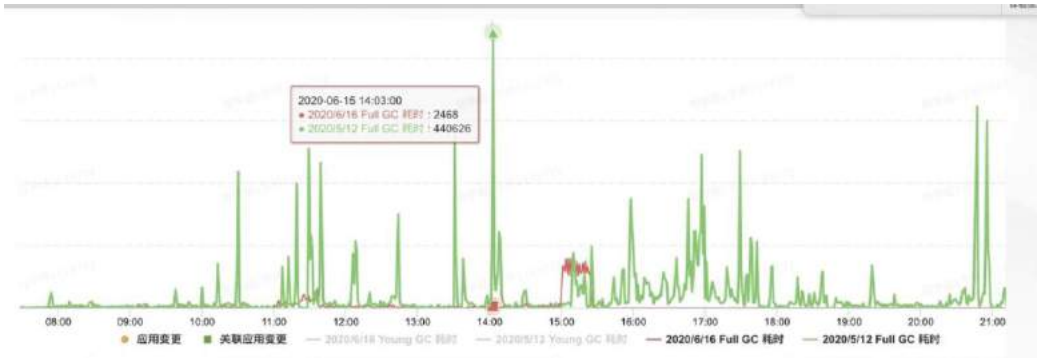


原因是 Dubbo 对异常服务端剔除不够及时，导致还会调用到异常服务器，出现大量报错。同时，我们也构建了突发高并发情况下的场景，发现了一些瓶颈，例如：

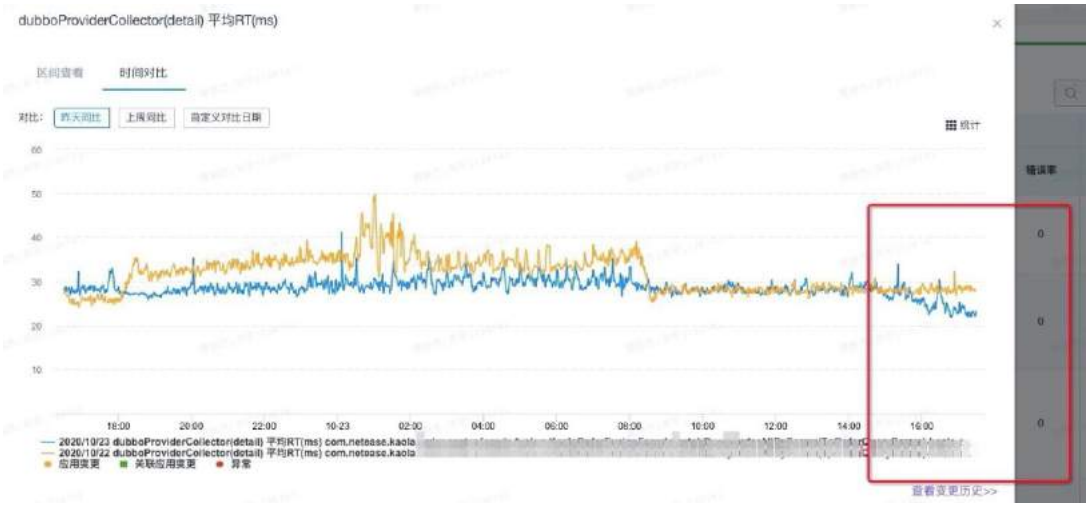
瞬间大并发消耗掉绝大部分 CPU 。我将在下一篇文章中重点解读我们通过混沌测试发现这些问题后是如何解决这些性能瓶颈的。

（三）性能优化

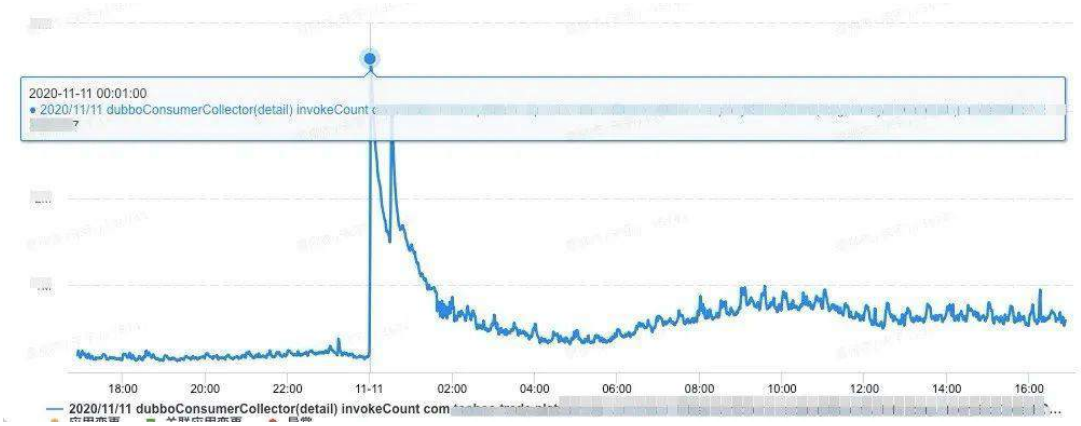
Dubbo 核心之前未经历过超大规模集团的考验，性能上面必将面临着巨大的挑战；对于 Dubbo 来说，优化主要从地址推送链路和调用服务链路两个链路来进行。对于地址推送链路，主要是减少内存的分配，优化数据结构，减少静态时地址占用内存对应用的影响，从而减少 ygc/fgc 造成的抖动问题。我们利用测试同学提供的风暴程序，模拟了反复推送海量地址的场景，通过优化，120 万个 Dubbo 服务地址常态内存占用从 8.5G 下降到 1.5G，有效降低 GC 频率。



另外一方面，在调用链路上，我们主要对选址过程、LoadBalancer、Filter 等进行优化，总体 CPU 下降达到 20%，RT 也有一个比较明显的下降。



成果



在 双 11 考拉零点高峰，某个 Dubbo 接口，总的流量达到了数百万次/每分钟 ,全程稳定顺滑，达到了预定的目标，Dubbo 3.0 是至重启开源以来，首次在这么大规模的场景进行验证，充分证明了 Dubbo 3.0 的稳定性。

结语

在本次 双 11 考拉落地 Dubbo 3.0 只是在阿里内部全面落地 Dubbo 3.0 的第一步，现在 Dubbo 3.0 云原生的新特性也在如火如荼的进行开发与验证，如应用级服务发现、新一代云原生通信协议 Triple 等已经开始在集团电商应用开始进行 Beta 试点。阿里微服务体系完成了通过开源构建生态和标准，通过云产品 MSE、EDAS 等完成产品化和能力输出，通过阿里内部场景锻炼高性能和高可用的核心竞争力。从而完成了三位一体的正向循环，通过标准持续输出阿里巴巴的核心竞争力，让外部企业快速享有阿里微服务能力，加速企业数字化转型！

Dubbo 3.0 sample@GitHub：
https://github.com/apache/incubator-dubbo-samples/tree/3.x

申通快递 双 11 云原生应用实践

一年一度的 “双 11” 大促中，交易额每年都在刷新，承接这些交易商品的快递包裹的数量也在成倍增长。这些快速的增长对物流系统带来了巨大的挑战，让物流管理更加敏捷来应对 “双 11” 成为了必须解决的问题。

申通是目前国内最大的物流公司之一，为了解决 “双 11” 的技术挑战，申通在物流场景引入 IOT、大数据和 AI 等先进和创建的技术手段，通过不断的技术快速迭代，使得物流资源得到有效的配置，推动了物流行业的发展。

快速的技术迭代带来了对 IT 基础设施的挑战，申通近年来全面将传统应用切换使用云原生架构，通过云原生架构实现应用的快速迭代、稳定的高可用、资源的动态扩缩容来支撑起快速的技术创新。

上云前，申通使用线下机房作为计算及数据存储平台，一到 双 11 资源需求就膨胀，大促之后则闲置浪费；上云和云原生化后，几乎全部的资源都是按量购买，使用云原生技术快速扩缩容，双 11 前快速扩容，双 11 释放，真正做到了开箱即用，不产生一天浪费。与去年 双 11 当天相比，今年 11 月 1 到 3 日，在业务量大幅提升的情况下，IT 投入反而降低了 30%。上云的成效显著。

申通云原生化架构的背景

目前申通正在把业务从 IDC 机房往云上搬迁，核心业务系统目前已经在云上完成流量承接。原有 IDC 系统帮助申通早期业务快速发展，但也暴露了不少问题，传统 IOE 架构，各系统架构的不规范，稳定性，研发效率等都限制了业务发展需求。

随着云计算在国内的越发成熟，更多的企业在把自己核心系统往云上搬，享受云计算带来的技术红利。在跟阿里云多次技术交流之后最终确定阿里云为唯一合作伙伴，为申通提供稳定的计算、数据处理平台。

采用云原生应用架构的诉求/驱动力

快递公司是非常典型的云边一体架构，实操环节很重。大量的业务逻辑下沉到边缘，所以申通在上云改造过程中，也在尝试做云边一体化的架构升级。通过云边一体，可以让开发在同一个平台上面完成云上业务及边缘侧的业务开发。同时快递公司还有典型的大数据处理场景，全网每天会新增几亿条扫描数据，需要对这些数据进行实时分析，对数据的处理要求非常高。

之前使用线下机房作为计算及数据存储平台的方式，使申通在业务增长过程中遇到了一些瓶颈，比如软件交付周期过长、大促保障对资源的要求、系统稳定性挑战等等。而云原生技术就是来解决传统应用升级缓慢、架构臃肿、不能快速迭代等方面的问题。正是看中了云原生技术能够给我们带来的价值才驱使我们转为使用公有云作为主要计算资源。

站在企业的角度来看，在这样一个快速多变的时代，云原生给我们带来的价值也正是企业最需要的：

- 唯快不破。这里的“快”有两层含义，一是业务应用快速上线，通过云原生技术可以做到快速上线部署；二是在业务爆发式增长的时候，对资源的需求要做到开箱即用。
- 稳中求变。业务稳定性永远是第一位。通过监控埋点，业务日志收集，链路监控等手段保证了在快速迭代过程中业务系统的稳定性。
- 节省资源。通过对计算资源的水位监测，结合业务的峰值情况，当发现资源利用率偏低采用降配规格及数量，降低整个资源的费用。相比于一次性投入租建机房及相应的维护费用，使用公有云成本投入更低。

- 开拓创新。采用微服务架构，将之前臃肿的架构进行合理拆分，再结合容器编排的能力做到持续交付。让企业成功转型成为一家 DevOps 驱动的公司。

申通云原生架构历程

（一）云原生化技术改造

原架构是基于 Vmware+Oracle 数据库的架构，通过上阿里云全面转型基于 Kubernetes 的云原生架构体系。应用服务架构重构主要分两部分：

1）程序代码改造升级

- 应用容器化

  跟虚拟机比起来，容器能同时提供效率和速度的提升，让其更适合微服务场景。所以我们引入容器技术。通过应用容器化解决了环境不一致的问题，保证应用在开发、测试、生产环境的一致性。

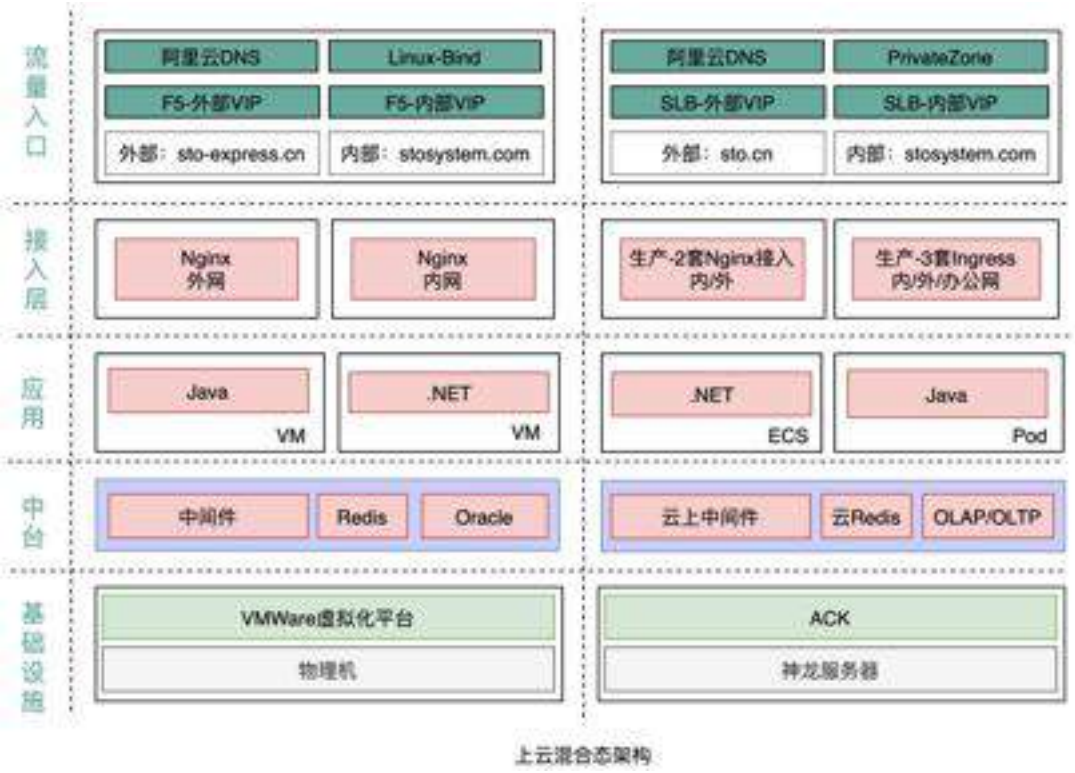
- 微服务改造

  原先很多业务是基于 Oracle 的存储过程及触发器完成的，系统之间的服务依赖也是走的数据库 OGG 同步完成。带来的问题就是系统非常难维护，也非常不稳定。通过引入 Kubernetes 的服务发现来做微服务方案，按业务域进行拆分，让整个系统更易于维护。

2）引入云原生数据库方案

通过引入 OLTP 跟 OLAP 型数据库，将在线数据与离线分析逻辑拆到两种数据库中，取代之前完全依赖 Oracle。特别是在处理历史数据查询场景下解决了 Oracle 支持不了的业务需求。

（二）云原生技术框架设计



整体架构

架构阐述：

- 基础设施

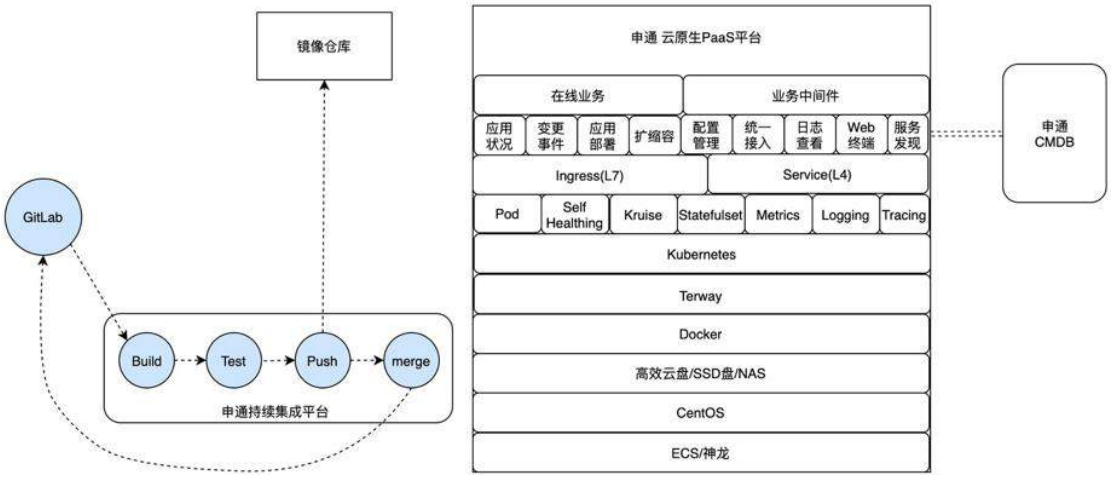
  全部的计算资源取自阿里云的神龙裸金属服务器，Kubernetes 搭配神龙服务器能够获得更佳性能及更合理的资源利用率，云上资源可以按量付费，特别适合大促场景，大促结束之后资源使用完释放。相比于线下自建机房和常备机器，云上资源操作更方便，管理成本也更低。

- 流量接入

  共有 2 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 能力来做统一的域名转发，这样可以节省公网 SLB 的数量便于运维管理。

（三）平台选择

整体的云原生 PaaS 平台基于阿里云容器服务 Kubernetes 版（ACK）打造：



平台特点：

- 测试、集成、预发、生产统一环境，打通 DevOps 闭环
- 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

（四）应用服务层设计

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

（五）运维管理

线上 Kubernetes 集群都是采用了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。同时上面跑的业务系统均通过我们的 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 集群带来的风险。

申通云原生应用服务特点

（一）API 接口

我们的应用场景主要有 2 块：

- 封装 Kubernetes 管控 API

  包括创建 StatefulSet、修改资源属性、创建 Service 资源等等，通过封装这些管控 API 方便通过一站式的 PaaS 平台来管理在线应用。

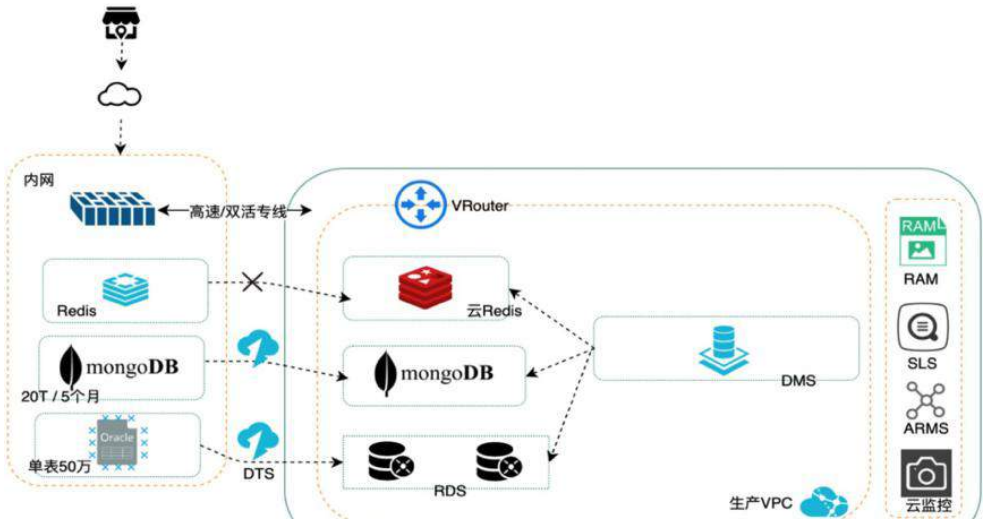
- 云原生业务系统

  我们云上的业务系统封装了各类云资源的 API，比如：封装 SLS 的 API、将在线数据写入 SLS 再跟 Maxcompute 或 Flink 集成。封装 OSS 的 API，方便在应用程序中将文件上传。

（二）应用和数据迁移

我们云上的业务系统及业务中间件都是通过镜像的方式部署，应用的服务通过 Service 发现，全部在线应用对应的 Pod 及 Service 配置均保存 PaaS 平台里面，每个应用历史版本对应的镜像版本都保存到系统中，可以基于这份配置快速构建一套业务生产环境。

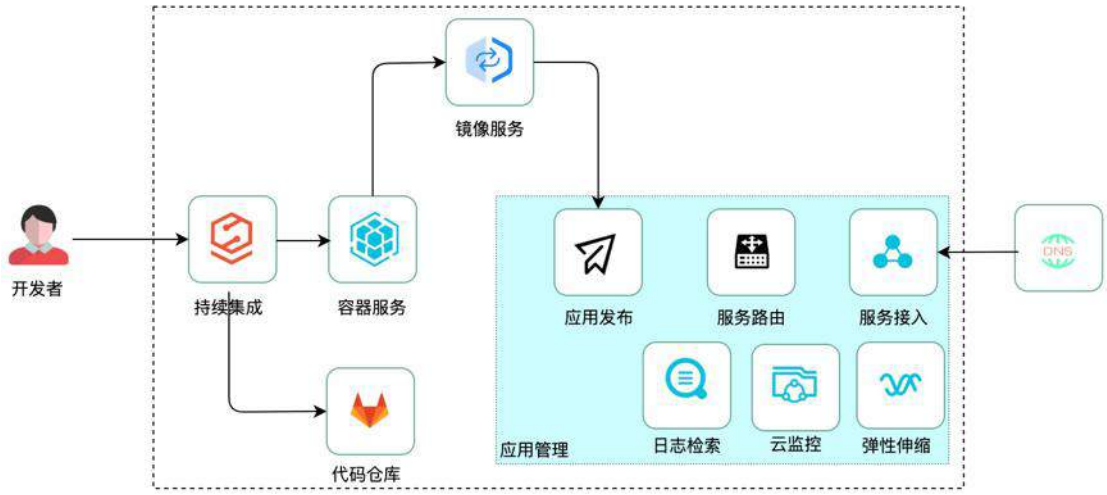
数据迁移示意图：



通过 DTS 工具将业务系统的数据从 IDC 存储及增量迁移到云上。在线数据稳定地存储在云原生的数据库上面，如 OLTP 类型的 RDS、PolarDB 支撑高并发的实时处理，OLAP 类型的 ADB 支持海量数据分析。同时对于小文件存储保存在 OSS 上面。引入 NAS 做共享存储介质，通过 Volume 直接挂载到神龙节点来实现应用数据共享。

（三）服务集成

以云原生 PaaS 示意:

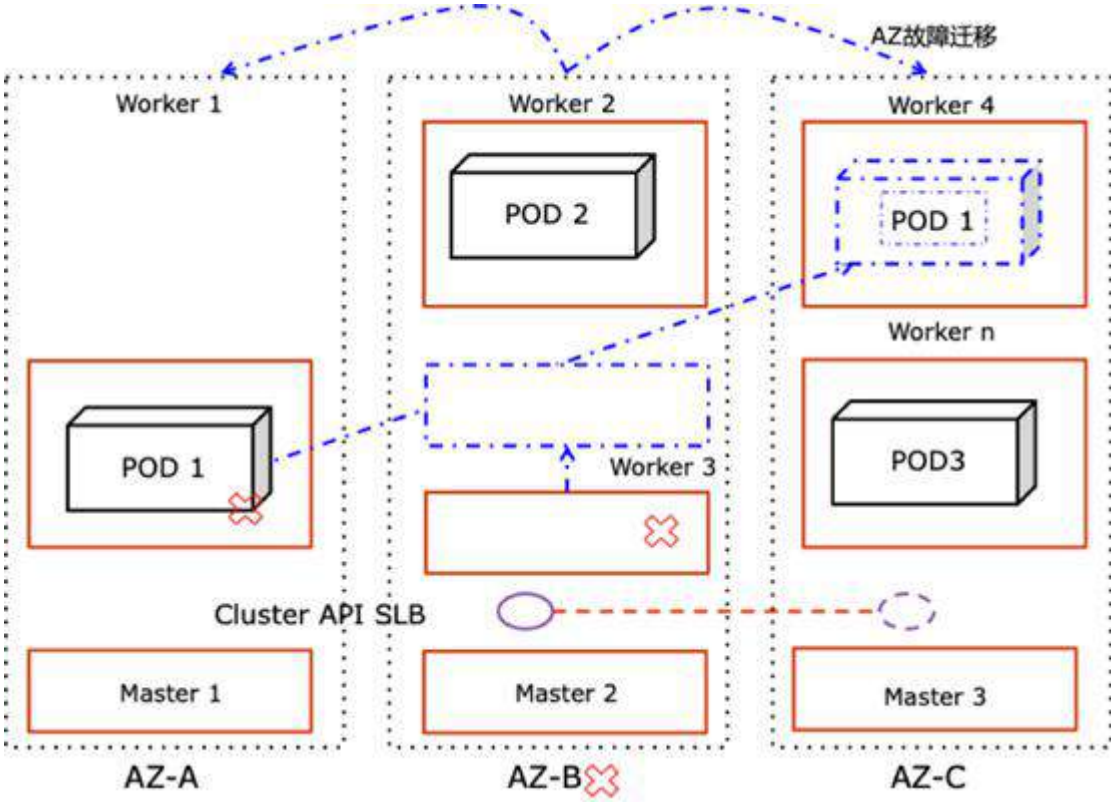


服务集成阐述

持续集成通过 Git 做版本控制，利用云效的持续集成功能实现了云原生应用的构建、编译及镜像上传，全部的业务镜像均保存在云端的镜像服务仓库。底层是 Kubernetes 集群作为整个业务的计算资源。其他集成的服务包括:

- 日志服务，通过集成日志服务方便研发人员方便定位业务及异常日志。
- 云监控，通过集成监控能力，方便运维研发人员快速发现故障。
- 服务接入，通过集成统一的接入，整个应用流量可做到精细化管理。
- 弹性伸缩，借助 ESS 的能力对资源进行动态编排，结合业务高低峰值做到资源利用率最大化。

（四）服务高可用



ACK 集群多层级高可用示意图

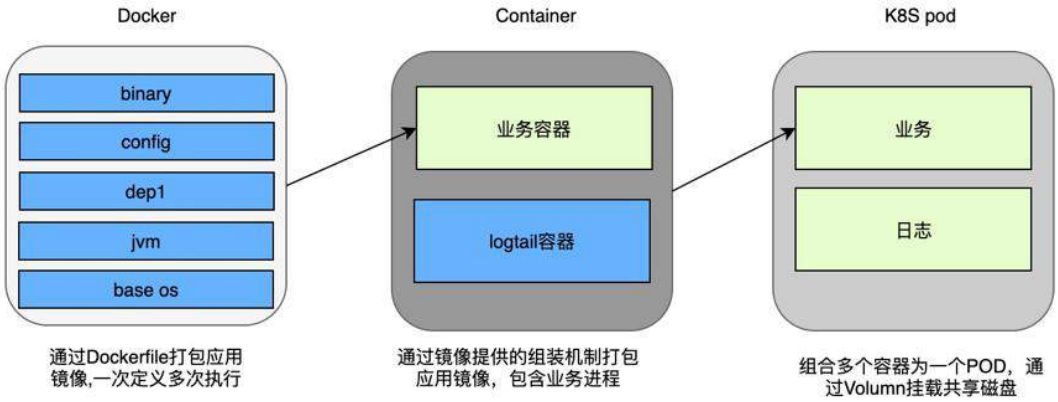
架构说明：

- 支持多可用区部署架构，由用户自定义分配比例
- 容器集群内故障迁移
- AZ 故障整体容器迁移

Kubernetes 集群通过控制应用的副本数来保证集群的高可用。当某个 Pod 节点出现当机故障时，通过副本数的保持可以快速在其他 worker 节点上再启新的 Pod。

（五）监控

主动发现业务故障，通过引入监控体系主动发现业务问题，快速解决故障。



监控采集示意图

在同一个 POD 里面部署了两个容器：一个是业务容器；一个是 Logtail 容器。应用只需要按照运维定的目录将业务日志打进去，即可完成监控数据采集。

技术/应用服务创新点

（一）从虚拟机到 Kubernetes

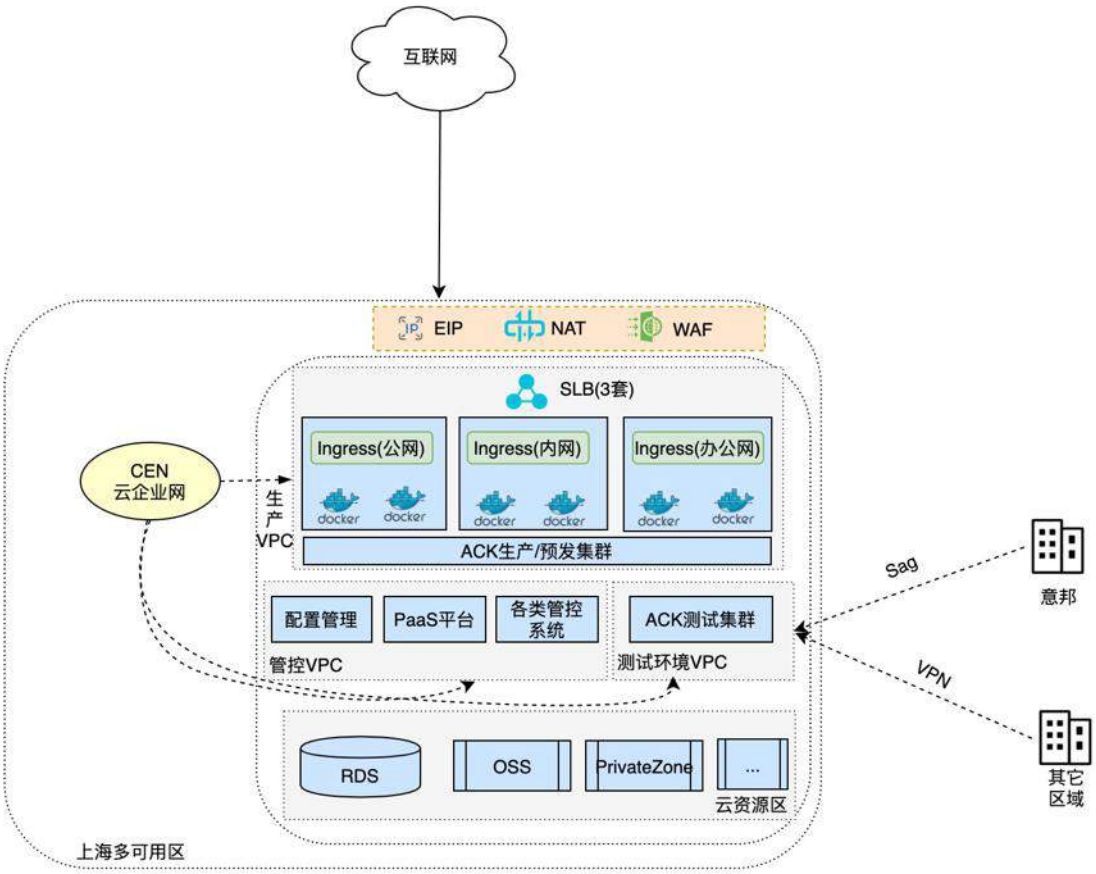
相比于通过虚拟机来运维应用，Kubernetes 可以将各类资源定义成描述文件，整个应用环境通过容器的方式统一，避免环境不一致的风险。通过修改副本数即可轻松完成应用容器的扩缩容操作。

（二）基于 Terway 让 Pod 和 ECS 网络处于同等地位

优势：

- 不依赖 VPC 路由表，就能打通网络，节点规模不受路由表 Quota 限制
- 不需要额外为 Pod 规划 Overlay 的网段
- 混合云专线打通也无需额外配置路由
- 可以直接将 POD 挂到 SLB 后端
- 性能高，相比于社区的 Flannel 提升至少 20%

（三）定义三套接入环境及三套业务环境



架构示意图

1）三套接入环境

- 公网接入：适合于跟外部客户对接，通过统一的证书卸载，收敛公网 IP。
- 办公网接入：适合于有敏感接口的对接，只允许指定源 IP 的请求，通过网络 ACL 让整个应用访问更安全。
- 内网接入：适合于业务之间及混合云架构下 IDC 的业务调用云上应用，内部调用性能更高也更安全。

2）三套业务环境

- 测试环境：全部的云资源都是给测试环境使用，可以采用低配资源来满足功能回归测试。
- 预发环境：准上线环境，连接生产环境的资源进行发布前最后一次功能验证。
- 生产环境：实际运行环境，接收真实流量处理业务请求。

应用效益

（一）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的需求，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是非常方便。用完即销毁，按量付费。另外云产品都是免运维自行托管在云端，可以节省人工运维成本，让企业更专注于做核心业务。

（二）稳定性方面

云上产品都是提供至少 5 个 9 以上的 SLA 服务，而自建的话稳定性差不少。另外有些开源的软件可能还存在部分功能上的 bug 影响了业务。另外在数据安全方面云上数据可以做到异地备份，阿里云数据类产品的归档高可靠、低成本、安全性、存储无限等特点，让企业数据更安全。

（三）效率方面

借助跟云产品的深度集成，方便研发一站式完成研发、运维工作。从业务需求立项到拉分支开发，再到测试环境功能回归验证，再部署到预发验证及最后上线，整个持续集成可以做到分钟级。排查问题方面，研发直接选择所负责的应用通过集成的 SLS 日志控制台快速检索程序的异常日志，定位问题。免去了登录机器查日志的麻烦。赋能业务：

（四）赋能业务

云上组件有 300 多种，涵盖了计算、AI、大数据、IOT 等等诸多领域，这样可以节省业务创新带来的技术成本。

总结和后续展望

目前申通已经使用云原生技术快速的将基础设施迁移到云上，使用云原生技术解决了双十一成本预算问题，服务监控问题，服务接入和负载均衡等问题，让 双 11 的快递高峰能够更低成本、更稳的方式应对。

对于类似于申通这样的传统企业数字化转型和上云来说，使用云原生技术内置的弹性、监控、负载均衡和服务发现等能力，可以大幅降低企业研发和运维人员的迁云的成本，让企业的研发和运维人员只需要关心业务研发和迁移，而无需管理大量的基础设施迁移成本。可以说是企业上云的最佳路径。

将来申通还在持续的利用最新的云原生技术继续优化基础设施和业务系统，下一步将会结合云原生技术进一步的降低成本和提升业务稳定性：

（一）实时的弹性调度

申通的快递业务是非常典型周期性业务，使用云原生技术的实时的弹性调度能力可以让每天的业务高低峰都能自动弹性伸缩。可能再节省一大笔的资源成本。

（二）智能运维和安全生产

结合云原生的细粒度的监控能力，结合 AIOPS 技术，对系统和业务的指标做到自动分析诊断，从而让异常事件做到及时发现和处理。

（三）服务网格

引入服务网格来优化目前的微服务架构，统一微服务调用的协议，实现全链路追踪和监控，提升研发和运维的效率。

「云原生上云」后的聚石塔是如何应对 双 11 下大规模应用挑战的

云原生被认为是云计算的重要发展趋势，并且已经成为数字新基建必不可少的一个组成部分。每年的阿里巴巴 双 11 都是考验各种前沿技术的最佳“实战场”，而今年云原生技术在 双 11 中的大规模应用，充分证明了云原生技术的动力与前景。

本文会系统讲解聚石塔在 2019 年 7 月份以阿里云容器服务为基础设施，进行云原生实践的探索和经验、带来的价值与改变，及其在 双 11 中的应用，希望对云原生化树立可以借鉴使用的案例和样板。

聚石塔业务背景与介绍

聚石塔最早上线于 2012 年，是阿里集团为应对电商管理和信息化挑战，帮助商家快速解决信息化建设与管理的瓶颈打造的一款开放电商云工作平台。它的价值在于汇聚了整个阿里系的各方资源优势，包括阿里集团下各个子公司的平台资源，如淘宝、天猫、阿里云等，通过资源共享与数据互通来创造无限的商业价值。

依托于聚石塔，各种业务类型（如 ERP、CRM、WMS 等业务）的服务商为淘宝、天猫等淘系的商家提供服务，服务商需要遵循聚石塔平台的发布规则、数据安全、稳定性建设等要求。这种关系决定了聚石塔平台的技术和架构，更直接决定服务商的技术演进方向和先进性。

聚石塔业务痛点

聚石塔承接的业务大类可以分为两个部分：

- 传统电商链路中，订单服务链路上的系统：比如 ERP 系统，CRM 系统，WMS 系统。
- 电商行业中直接面向客户的小程序场景，比如手淘与千牛客户端的小程序业务。

综合聚石塔的的业务类型，存在如下的业务痛点：

（一）标准化和稳定性问题

对于订单服务链路的系统而言，稳定性永远是最前面的“1”，一个系统抖动可能就会导致订单履约链路中断甚至造成资损以及客诉。稳定性和标准化是不分家的，相信大家对此对有强烈的感受。而此类系统的开发语言不统一，有 Java、PHP、.Net 等；技术栈复杂，涉及 Windows 系统、Linux 系统、单体应用、分布式应用，可谓五花八门。因此需要一套跨语言、跨平台的通用 PaaS 平台来解决应用的标准化、运维的标准化问题，并提供通用的链路问题观测手段，来帮助服务商和平台规范发布运维操作，发现链路问题，消除稳定性隐患。

（二）突发流量下的弹性问题

对于应用小程序的业务系统而言，最大的挑战就是需要应对突发流量以及流量的不确定性。尤其在 双 11 期间，手淘端各类小程序插件会面临比平时多十倍甚至百倍的流量。面对这种不确定性的流量洪峰，聚石塔需要一套可以实现流量预估、流量观测、流量控制以及标准应用快速扩缩容的 PaaS 平台。对于订单服务链路的系统而言，弹性能力也是关键点，在原来的架构下扩容需要经历创建虚拟机资源、部署并配置应用等诸多环节，服务商普遍感觉流程长、效率低。以上我们都总结为弹性能力的挑战。

（三）效率和成本的问题

聚石塔在云原生之前的应用部署基本都是基于 VM 直接部署进程，这种方式缺乏进程间的资源隔离。同时当 ECS 数量变多，资源的统一管理就变得非常复杂，很容易造成资源争抢导致应用稳定性问题以及资源浪费导致的多余成本开销。同时，在传统的 VM 部署模式中，应用的扩缩容不仅仅需要处理应用的代码包启动，还需要处理应用的端口冲突，应用所关联的存储资源分配，应用流量在 SLB 的挂载和摘除，应用配置的分发以及持久化，整个部署过程会变得非常耗时且容易出错。

聚石塔云原生落地方案

针对上述的业务痛点，聚石塔开始探索技术演进方向以及系统性的解决方案，以便为淘系服务商带来服务上质的提升。云原生带来的技术红利，比如应用环境标准化、DevOps 思想、弹性伸缩、跨语言的服务化能力以及运维自动化等，都不仅可以帮助聚石塔的服务商解决现存架构中的稳定性和成本问题，同时也可以帮助我们引导服务商实现技术架构的升级。

因此，聚石塔进行了重新定位，主动拥抱云原生。整体来看，目前的聚石塔云原生技术底座基于阿里云容器服务，平台目标是赋能服务商应用架构的稳定性升级，打造基于云原生的、面向业务链接的 DevOps PaaS 平台。

那么，为什么聚石塔会选择阿里云容器服务作为云原生基础设施呢?

阿里云容器服务 ACK（Alibaba Cloud Container Service for Kubernetes）是全球首批通过 Kubernetes 一致性认证的服务平台，提供高性能的容器应用管理服务，支持企业级 Kubernetes 容器化应用的生命周期管理。作为国内云计算容器平台的领军者，从 2015 年上线后，一路伴随并支撑 双 11 发展。

ACK 在阿里巴巴集团内作为核心的容器化基础设施，有丰富的应用场景和经验积累，包括电商、实时音视频、数据库、消息中间件、人工智能等场景，支撑广泛的内外部客户参加 双 11 活动；同时，容器服务将阿里巴巴内部各种大规模场景的经验和能力融入产品，向公有云客户开放，提升了更加丰富的功能和更加突出的稳定性，容器服务连续多年保持国内容器市场份额第一。

ACK 在今年 双 11 期间稳如磐石，深度参与了 双 11 众多领域的支撑：在阿里巴巴集团内部支撑电商后台系统 ASI，在零售云支撑聚石塔，在物流云支撑菜鸟 CPAAS，在中间件云原生支撑 MSE，在边缘云支持支撑 CDN 和边缘计算，并首度支持了数据库云原生化和钉钉音视频云原生化。

在过去的一年，ACK 进行了积极的技术升级，升级红利直接运用到了 双 11 中，进一步提升了 ACK 的技术竞争力和稳定性，升级包括：高性能云原生容器网络 Terway 相比于社区社区提升 30%，高性能存储 CSI 引入 BDF 等的支持支撑数据库 5K 台神龙主机对数十 PB 数据实现高效卷管理，极致弹性 ASK，Windows 容器等首次在 双 11 活动中参战并应用等等。规模化调度方面，ACK 高效稳定的管理了国内最大规模的数万个容器集群，是国内首个完成信通院大规模认证（1 万节点、1 百万 Pod）的厂商。规模化管理的技术和经验在 双 11 中支撑了全网客户集群 APIServer 数十万的峰值 QPS。

因此，聚石塔选择 ACK 容器服务，不论是从技术角度，还是业务角度，是非常合理的，双方的合作也是强强联合、相互赋能、共同成长。

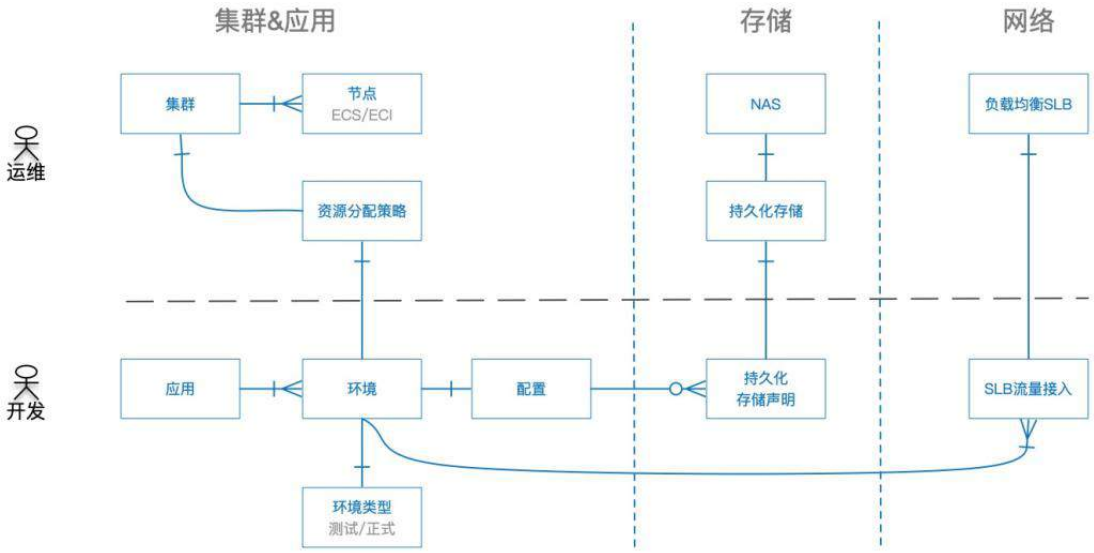
下面内容会讲到聚石塔如何使用云原生中的技术能力来解决实际的业务痛点和问题。

（一）应用和发布标准化

聚石塔上聚集了上万的开发者，但是不论规模还是技术能力都参差不齐。因此，聚石塔亟需为用户提供一套可以屏蔽 Kubernetes 复杂性的、易用、灵活高效的发布系统，进而降低用户使用 Kubernetes 的门槛，帮助用户享用云原生的技术红利，同时满足作为管控方对于稳定性的要求。为了应对各种不同业务形态的差异化，聚石塔通过标准化来解决，设计了一套通用的应用模型以及对应的发布规范。

1）应用模型

Kubernetes 的一个重要思想就是面向应用的 DevOps，聚石塔 DevOps 平台一开始就是以应用为中心的 Paas 平台，在应用模型的设计上，整体的设计原则是让开发者人员关注应用本身，让运维人员关注基础设施运维，从而让应用管理和交付变得更轻松和可控。

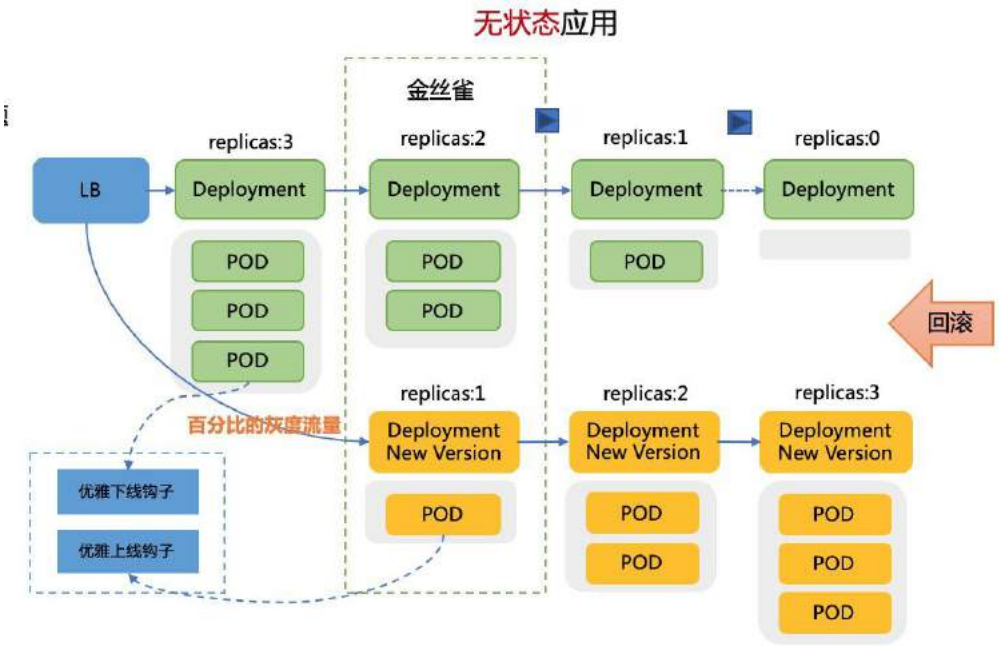


同时，为了满足客户需求的多样性，比如对于同一个应用，SaaS 服务商为不同商家提供不同规模的应用实例数量，或部署有差异性的代码，聚石塔在应用下支持了多环境，并支持对测试和正式环境的隔离。

2）发布

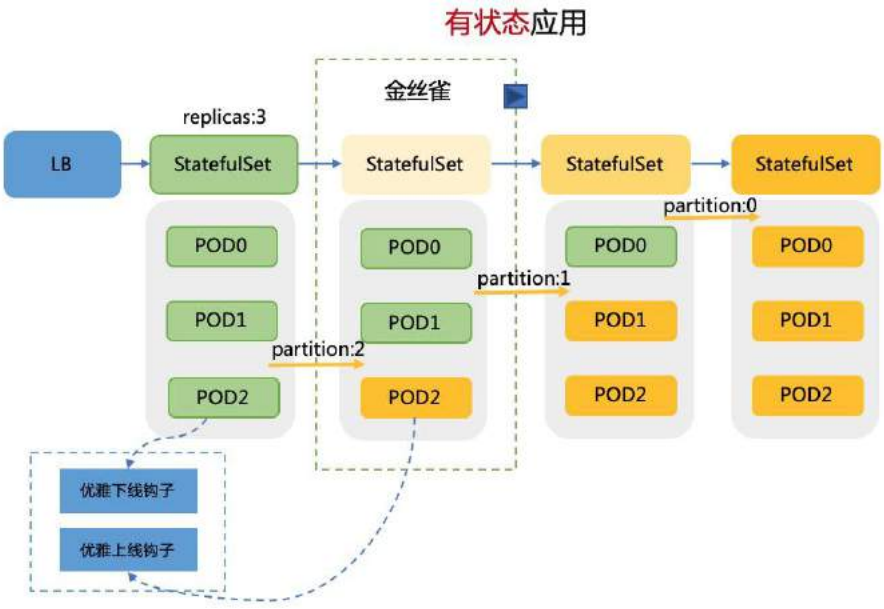
稳定性的风险很大程度上来源于变更，Kubernetes 自带的滚动发布，虽然标准化了发布环节，但在发布期间无法暂停，即使服务起来了，可能功能和业务存在问题，最终也会随着滚动不断扩大影响面；对此，聚石塔设计了支持暂停的分批发布，通过提前批的“金丝雀”验证，从而提升系统的稳定性。

以一个“无状态”的应用为例，主要实现原理是，通过控制两个版本的 Deployment 的滚动，来做到可暂停和金丝雀验证。

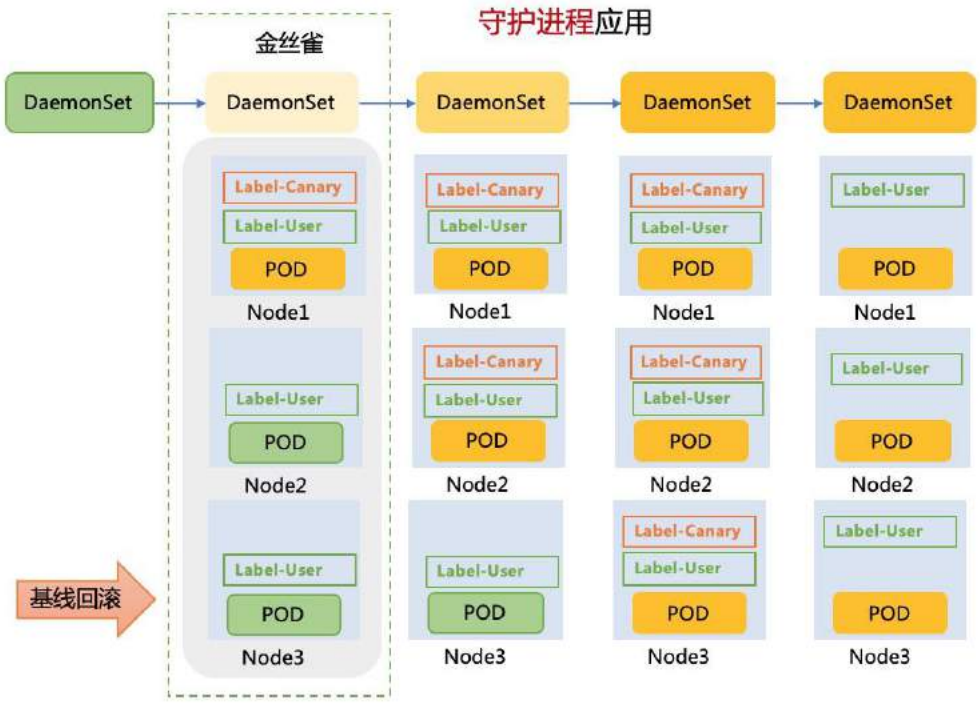


同时，相对于其他 Paas 平台，聚石塔需要支撑更多的客户应用场景，还支持有状态应用（像 Zookeeper）以及守护进程应用（比如日志采集）的分批发布，以满足不同客户基于成本和防锁定层面的需求。

而对于有状态应用，主要原理则是通过控制 Kubernetes StatefulSet 的 Partition 分区来做到分批和暂停。



另外，对于守护进程应用，则是通过对 DaemonSet 进行 Label 的调度来实现：



从而做到不同类型的应用，得到统一的操作体感和变更稳定性保障。

（二）弹性：ACK/ASK + HPA

随着集群规模的增大，资源成本的消耗越发明显，尤其对于流量波动较大的场景（例如电商场景），问题更加突出。用户不可能一直把集群资源保持在高水位上，大多数情况下用户都会把集群资源维持在足以应对日常流量的规模，并稍微冗余一部分资源，在流量高峰在来临前进行集群资源的扩容。

对于 Kubernetes 集群来说，启动一个 Pod 是很快的，但是对于上述场景，启动 Pod 前需要提前扩容 ECS，使之加入集群后，才能进行扩容，而扩容 ECS 则需要数分钟的时间。

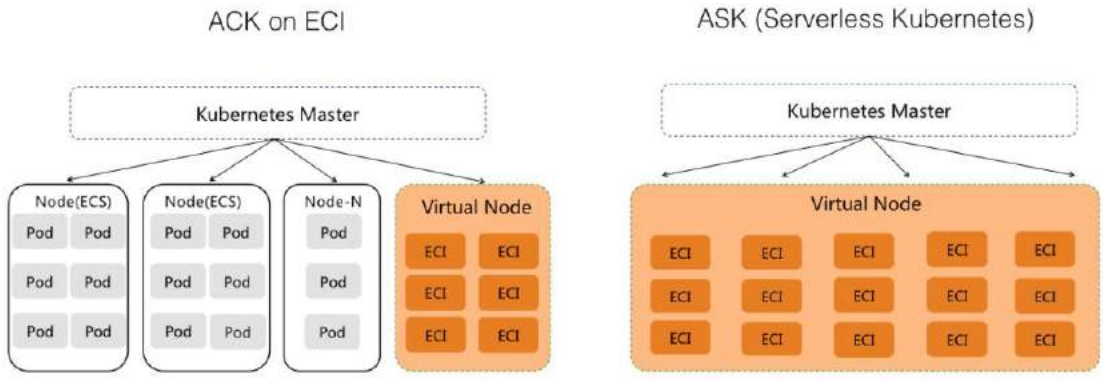
以上的弹性能力比较弱，操作繁琐，耗时过长，无法及时响应流量变化，并且依然会存在很多的资源浪费，消耗成本。

1）ACK/ASK 与 ECI

阿里云弹性容器实例（Elastic Container Instance），旨在用户无需管理底层服务器，也无需关心运行过程中的容量规划，只需要提供打包好的 Docker 镜像，即可运行容器，并仅为容器实际运行消耗的资源付费。ECI 是 ACK 底层的一种资源形态，一个 ECI 可以看做一个 Pod，无需提前扩容 ECS，便可直接启动。

ECI 的价格与同等规格按量付费的 ECS 价格相近，且为按秒付费，ECS 为小时级别。如果用户仅需要使用 10 分钟，理论上 ECI 的成本是使用 ECS 的 1/6。

如下图所示，Kubernetes 集群通过 Virtual Node 使用 ECI，该技术来源于 Kubernetes 社区的 Virtual Kubelet 技术，无需提前规划节点容量，不占用已购买的 ECS 资源。



2）聚石塔结合 ECI 与 HPA

为了带给客户更好的体验，聚石塔基于阿里云 ACK/ASK，结合底层的 ECI 资源，以及原生 HPA 能力，为客户带来了更加灵活优质的弹性能力。

通过使用污点来控制一个应用的某一个环境是否可是使用 ECI，并且会优先使用集群中 ECS 资源，资源不足时才会使用 ECI，从而为用户解决额外成本。

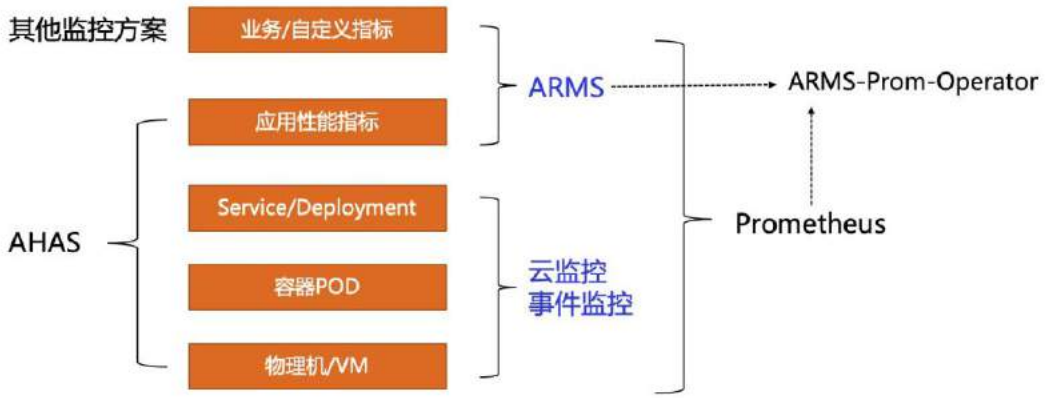
同时聚石塔提供了标准的 HPA 能力，以及 cronhpa 能力，帮助用户实现根据指标的自动伸缩（例如根据 CPU 的使用情况自动伸缩 Pod 数量），以及定时伸缩的能力。

并且通过两者的结合，在流量动态变化的过程中，无需手动购买 ECS，以及手动扩容 Pod，实现 0 运维成本。

以上能力在今年 618 前开始小范围推广，完美通过了 618 以及 双 11 的考验，用户在 双 11 期间单个应用使用 ECI 占比最高达到 90%，为用户节约了一半的计算成本。在双 11 期间 ECI 在电商业务场景下整体运行稳定，平均弹升时间在 15s 以内，相比 ECS 扩容至少需要 10 分钟，大大减少了扩容的时间成本，保证业务应对峰值流量的稳定性。

（三）应用监控

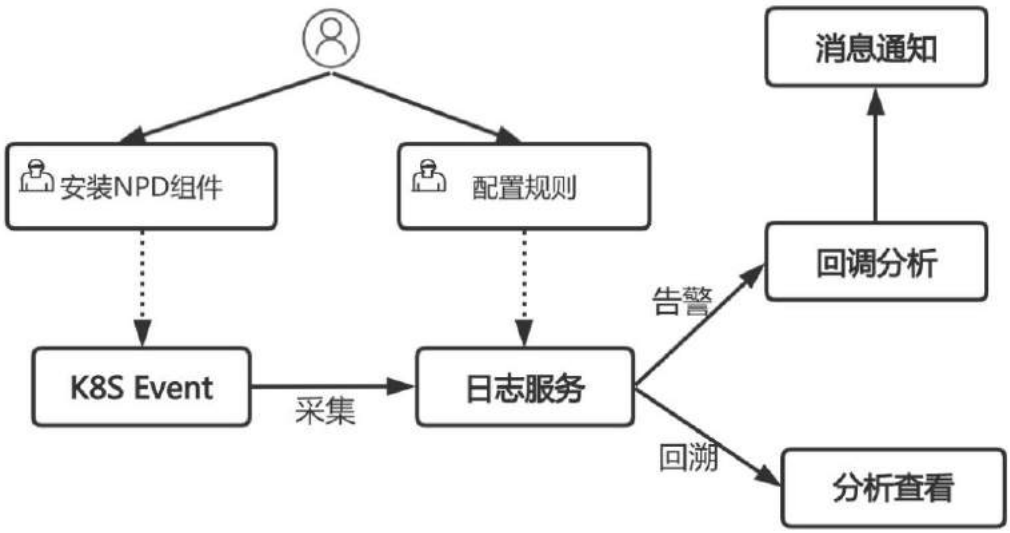
在享受 Kubernetes 云原生技术带来快速发布、弹性伸缩等便利的同时，如何做到可监控可运维也是聚石塔的核心挑战。一个合格的监控系统需要具体准确性、实时性、可用性，提供分析和解决问题的洞察力。传统的监控方案，大部分是自顶向下的，配置一个监控的任务、采集端点，应用的生命周期与监控任务生命周期一致，采集目标也是固定的，无论应用如何重启、变化，对于采集任务而言只要采集端点没有变化，那么任何的变化都是生命周期中的正常现象。与传统应用相比，基于 Kubernetes 的云原生应用容器实例是动态调度的、生命周期短，容器上层更是难以监控的如 Deployment、负载均衡 Service 等抽象，此外，还需要底层 ECS 计算资源、实例生命周期、Kubernetes 集群自身以及集群核心组件的各个维度的监控。基于上述考虑，聚石塔充分打通阿里云云原生监控中间件产品，为聚石塔云应用提供了分层一体化监控解决方案。



以事件监控为例，介绍下阿里云事件中心如何在聚石塔 PaaS 平台落地。

事件中心

聚石塔借助阿里云日志服务提供的集群事件采集、索引、分析、告警等能力，将集群和云应用上的各种事件进行监控和告警。事件的范围涵盖所有 Kubernetes 原生 event、集群组件 event 以及其他各种定制的检查。通常比较关心的是会影响云应用正常运行的一些异常情况，比如 Node 节点不可用、资源不足、OOM 等，比如 Pod 实例容器重启、驱逐、健康检查失败、启动失败等。PaaS 平台上云应用实践过程。



- 用户为集群一键安装 NPD 组件，为集群和应用分别配置告警规则，设置关注的事件类型和通知方式即可。
- 集群上所有事件自动采集到 SLS 日志服务，日志服务上的告警规则由我们根据事件类型和用途自动配置。
- 日志服务告警回调之后，由我们统一分析处理后进行消息推送。

事件监控和告警功能，不仅能在日常帮助用户排查和定位自身应用问题，也为平台对各个应用的运行情况有较好的了解，从而制定个性化的优化方案以及大促保障方案。此外，对于集群底层的组件，也可以通过配置相应的规则进行告警通知，此次 双 11 大促，聚石塔为核心集群自动配置了集群 DNS 组件的告警，以便及时扩容或者切换更高可用的 DNS 方案，从而确保业务稳定。

（四）DNS 生产环境优化实践

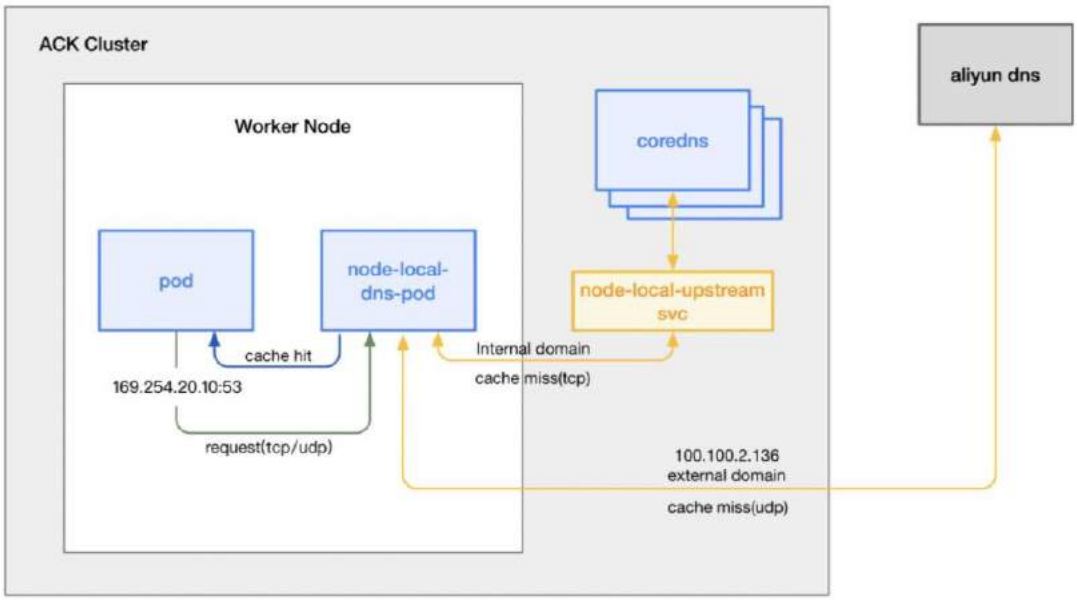
由于聚石塔的用户大多是电商或小程序的场景，流量波动明显，并且开发语言多样化，有些语言没有很好的连接池，导致每一次请求都需要进行一次 DNS 解析。Kubernets 默认的 CoreDNS 在高并发时会遇到性能瓶颈，部分用户在日常活动中，已经遇到了 DNS 性能的问题，更不要说 双 11 期间，因此聚石塔对 DNS 的性能做了深入的优化，确保 双 11 的稳定性。

1）Node Local DNS

默认的 CoreDNS 方式是采用的 Deployment 方式部署的无状态服务，集群中的 Pod，通过 service 对其进行访问。通过上述流程可以看到，DNS 解析需要跨节点请求，性能不是很高，在高并发的场景下，容易达到瓶颈。

为了进一步提高性能，阿里云提供了 ack-node-local-dns。原理如下，通过 DaemonSet 的方式，将 DNS 解析的服务在每个节点上启动一个实例，同时设置相应的转发规则，将 Pod 发出的 DNS 的请求转发到各自节点上对应的 DNS 解析服务，从而避免了跨节点的访问，很大程度上提高了性能。

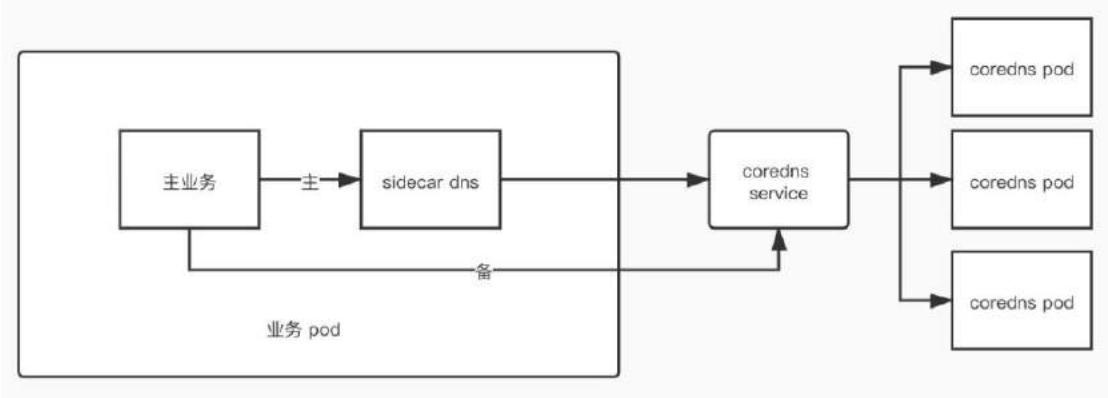
聚石塔对该插件进行了相应封装，可以使用户无感知的在两种方式间进行切换。



2）Sidecar DNS

对于 ECI 的场景，由于不存在真实的宿主机，因此无法采用上述 Node Local DNS 的方案提高 DNS 性能，同时各个节点上的服务数量不同，并且不同应用对的 dns 解析并发量的不同，在极端的场景下可能会出现 DNS 解析并发量分布不均，从而导致部分节点上 DNS 解析出现问题。

基于以上场景，聚石塔结合 Kubernetes 提供的 sidecar 能力，提供了 sidecar dns。原理如下图所示。



通过在 Pod 容器组中加入 DNS 解析服务容器。实现容器组中，其他容器所有的 DNS 解析请求均通过 sidecar dns 获取。sidecar dns 可以看做是业务容器的缓存，只为自身所在的 Pod 提供服务，不会存在负载分配不均的问题，并且可以为 ECI 提供相应的 dns 解决方案。

（五）Windows 场景落地

全球范围内来看，Windows 的市场份额还不容小觑，在 StackOverflow 2020 最新的开发者调研中，在最受欢迎的的框架、开发包和工具中，.Net 的占比超过了 35%；在最受欢迎的编程语言中，C# 虽然有所下滑，仍保持在 30% 以上。随着云原生的接受度越来越高，越来越多的传统行业也对容器等云原生技术的呼声越来越高，迫切需要更好的支持。

1）限制

Kubernetes 最先是在 1.14 版本 GA 实现了 Windows Server 2019 上进程容器的调度，随着后面的不断迭代，Windows 容器在调度、安全、网络、存储和镜像管理等

模块上都有了很大的进步，正在慢慢对齐 Linux 容器的功能并尽最大努力保持对异构容器管理的统一性。但我们还是能看到 Windows 容器有很多的限制，这种限制更多的是来自于操作系统层面的。

隔离不彻底

目前 Windows 容器的实现模式分为："进程容器"和"Hyper-V 容器"。"进程容器"是通过任务名管理来模拟资源隔离的，所以这种隔离是不彻底的，最常见的限制就是没法 OOM kill。而"Hyper-V 容器"是通过内核隔离技术来实现，因此这种隔离是最彻底的。

Kubernetes 默认支持的是"进程容器"，其实说"只支持"都不为过，为什么呢？因为目前能支持"Hyper-V 容器"的运行时只有 Dokcer EE，而又受限于其实现，"Hyper-V 容器"不能支持 Multiple Container one Pod 的场景，再加上微软把节点上可以跑"Hyper-V 容器"的数目也 License 化，这就极大的阻碍了"Hyper-V 容器"Kubernetes 化的进程。

升级难平滑

为了提高迭代效率，加速功能沉淀，微软在 2017 年推出一种新的系统发布里程："半年通道版"（Semi-Annual Channel）。相比于"长通道版"（Long-Term Servicing Channel），"半年通道版"是按照半年一次来进行 release 的，所 release 的内核是完全兼容当前所支持的 "长通道版"的操作系统。比方说，Windows Server 1809 SAC，Windows Server 1903 SAC ，Windows Server 1909 SAC 等都属于 Windows Server 2019 LTSC。

比起"长通道版"，显然"半年通道版"来得更加敏捷高效，但是转而来的是更复杂的升级管理。

严格内核版本要求的"进程容器"：由于进程容器是通过任务名模拟的，这就导致了容器的 base 镜像内核版本必须等于节点内核版本。换句话说，1903 SAC 的容器是没法跑在 1809 SAC 的节点上，反之亦然。

有限兼容的"Hyper-V 容器"："Hyper-V 容器"的向后兼容是有限制的。比方说，在 2004 SAC 的容器上，通过 Hyper-V 技术创建的容器，只能兼容 2014 SAC，1909 SAC 和 1903 SAC，而无法运行 1809 SAC。

安全管理困境：当碰到安全补丁的时候，问题会变得更麻烦，除了节点要打安全补丁以外，容器也要重新 re-package。详情可以参看：https://support.microsoft.com/en-us/help/4542617/you-might-encounter-issues-when-using-windows-server-containers-with-t。

文件难管理

目前的 Windows 容器的文件管理比较 tricky。由于 Docker EE 只实现了主机目录级别的挂载，这就导致 Windows 要读写某个文件的时候，必须把其整个目录挂载进来。于此同时，由于主机的 ACL 管理不能被投影到容器内，这就导致了 Windows 容器对文件的权限修改是不能被主机所接收的。

在 Secret 资源的管理上，在 Linux 上请求挂载的 Secret 资源会被暂存到内存里面，但由于 Windows 不支持挂载内存目录，所以 Secret 资源的内容会被放置在节点上。这就需要一些额外的机制来对 Secret 资源做控制。

2）展望

不过随着这几年的努力，我们正迎来一个更加稳定和成熟的 Kubernetes Windows 解决方案。

在运行层面，ContainerD 会加速替代 Docker EE。目前社区在 ContainerD 上集成了 HCS v2，实现了对单独文件的挂载和容器优雅退出的管理，在 v1.20 上将会实现对 GPU 和 GMSA 的支持。另外，我们也可以看到实现"Hyper-V 容器"和"特权容器"也已位列 roadmap。

在镜像层面，微软在努力的削薄基础镜像层，从 1809 SAC 到 2004 SAC，整个 Windows Server Core 减少了将近一半的大小，但是功能却越来越丰富，这是让人非常惊喜的。这也为后期的"Hyper-V 容器"打下来良好的基础。

在功能层面，随着 privileged proxy 模式的兴起，很多之前没法容器化运行的方案都找到了合适的解决方式。比方说，CSI Windows 的实现方案，就是借力于 https://github.com/kubernetes-csi/csi-proxy 的实现。

在运维层面，借助于 Kubernetes 的能力，完全可以打造一套 CICD 流来解决掉升级平滑的问题。

聚石塔上的传统电商链路中，订单类的各类 ERP、CRM、WMS 等系统，使用 Windows 技术栈开发的占了一半以上。如何帮助 Windows 场景下的系统进行云原生升级改造，对我们也是一个全新的挑战。半年来，聚石与阿里云 ACK 技术团队一起做了许多尝试，使用阿里云 ACK 集群 + Windows 节点池作为底层基础设施，聚石塔 PaaS 已经能够提供完整的发布部署、负载均衡、基础监控、扩缩容等功能。在今年的 双 11 大促中，已经有不少客户的系统运行在 Windows 容器之上，平稳渡过 双 11 的业务高峰。

当然，Windows 场景下还有其他一些特性暂不支持，比如 NAS 共享存储、日志采集、弹性伸缩、事件监控等，双 11 之后，聚石塔与与阿里云 ACK 会继续一起努力，为 Windows 容器的成熟和市场化贡献更大的技术和业务价值。

云原生带来的业务和技术价值

在正在进行中的 2020 年 双 11 大促中，聚石塔云应用 PaaS 已经落地开花，聚石塔的核心服务商的核心系统也基本完成了云原生化的改造。

在 双 11 第一波高峰中，构建在阿里云 ACK 之上的聚石塔云原生规模达到了上万核 CPU，上千个集群，承载 2 万个 Pod。基于 Kubernetes，聚石塔 PaaS 封装的应用与运行环境的标准化，发布部署以及流量变更流程标准化已经成为聚石塔服务商日常软件交付流程中必不可少的一部分，通过这些努力聚石塔最大限度的减少了 双 11 期间不必要的应用变更，将聚石塔的变更数量减少为日常的十分之一，保证线上系统稳定性；同时我们推动聚石塔 PaaS 上的核心应用完成了基于阈值（CPU，内存，load）以及基于集群事件（比如 Pod 驱逐，节点不可用）的监控告警，实现了线上问题先于客户发现，及时解决止损；对于小程序的突发流量应用场景，聚石塔在普通 Kubernetes 集群内引入了 vnode 和 ECI 的技术，保证集群资源不足的情况下，可以实现 Pod 的秒级快速应急弹缩，应对突发的流量洪峰。

云原生带来的价值不止于此，基于聚石塔的用户反馈，云应用 PaaS 普遍给开发者带来了 30% 以上的研发效能提升，应用的扩缩容甚至实现了小时级到秒级的时间缩短；同时基于容器的运行环境以及计算资源标准化抽象，给大多数用户带来了 30% 以上的计算资源成本节约。基于云原生的架构与运维规范也推动了服务商自身的软件架构升级，比如从单体应用向分布式应用的升级，从垂直扩缩的有状态应用向可横向扩缩的无状态应用的升级。

云原生在电商垂直业务下的实践才刚刚起航，后续聚石塔还将持续在云原生技术领域深耕，在应用运维标准化 OAM，应用链路的可观测性，基于 Mesh 的应用流量监测和控制，基于业务指标的弹性伸缩，分布式应用压测与容灾等领域继续探索，将云原生的技术价值赋能给客户和业务，同时将电商垂直领域的云原生技术实践反哺云原生社区。

高德最佳实践：Serverless 规模化落地有哪些价值?

导读：曾经看上去很美、一直被观望的 Serverless，现已逐渐进入落地的阶段。今年的"十一出行节"，高德在核心业务规模化落地 Serverless，由 Serverless 支撑的业务在流量高峰期的表现十分优秀。传统应用也能带来同样的体验，那么 Serverless 的差异化价值又是什么呢？本文分享高德 Serverless 规模化落地背后的实践总结。

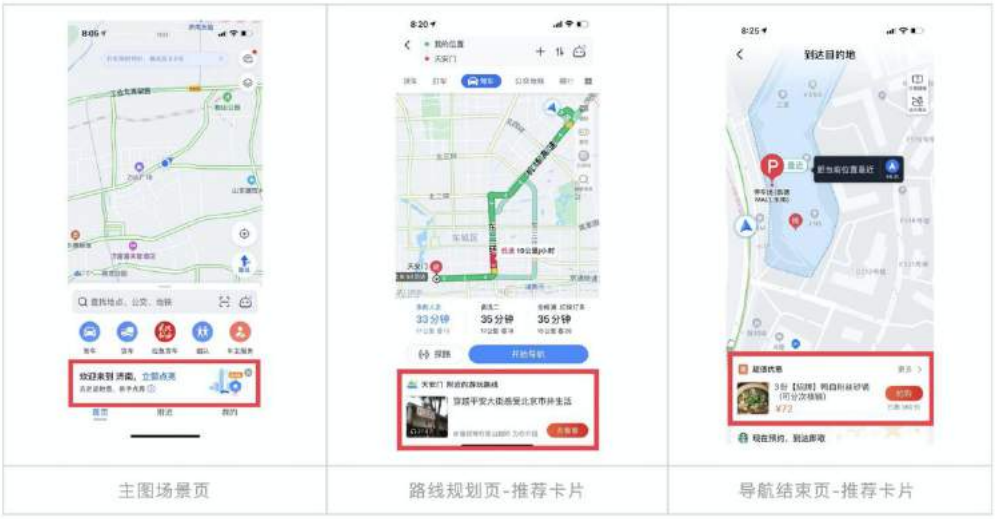
随着 Serverless 概念的进一步普及，开发者已经从观望状态进入尝试阶段，更多的落地场景也在不断解锁。“Serverless 只适合小场景吗？”、“只能被事件驱动吗？”这些早期对 Serverless 的质疑正在逐渐消散，用户正在更多的核心场景中，开始采用 Serverless 技术达到提效、弹性、成本优化等目的。作为地图应用的领导者，高德为带给用户更好的出行体验，不断在新技术领域进行探索，在核心业务规模化落地 Serverless，现已取得显著成效。

2020 年的“十一出行节”期间，高德地图创造了记录 ——截止 2020 年 10 月 1 日 13 时 27 分 27 秒，高德地图当日活跃用户突破 1 亿，比 2019 年 10 月 1 日提前 3 时 41 分达成此记录。

期间，Serverless 作为其中一个核心技术场景，平稳扛住了流量高峰期的考验。值得一提的是，由 Serverless 支撑的业务在流量高峰期的表现十分优秀，每分钟函数调用量接近两百万次。这再次验证了 Serverless 基础技术的价值，进一步拓展了技术场景。

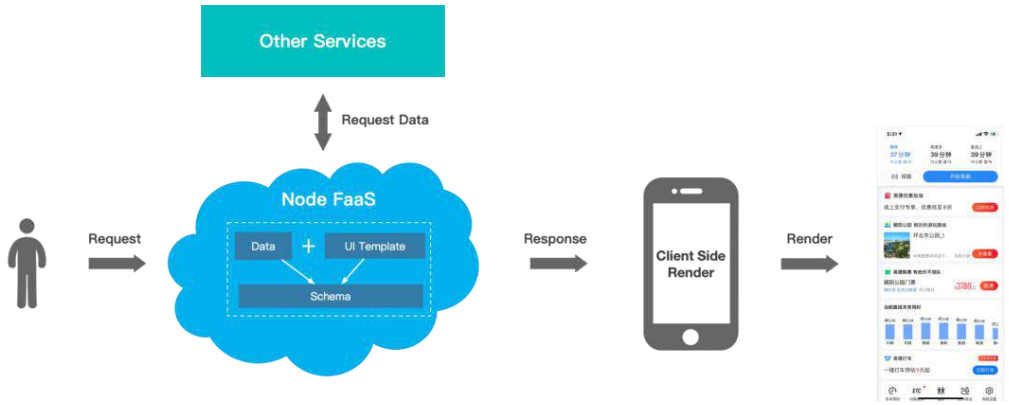
业务场景

自主出行是高德地图的核心业务，涉及到用户出行相关的功能诉求，承载了高德地图 APP 内最大的用户流量。下图为自主出行核心业务中应用 Node FaaS 的部分场景，从左至右依次为：主图场景页、路线规划页、导航结束页。



随着功能的进一步拓展，高德地图从导航工具升级为出行服务平台和生活信息服务入口，进一步拓展了出行相关的生活信息服务场景，带给用户更全面的用户体验。上图功能为场景推荐卡片，旨在根据用户出行意图推荐信息，提升用户出行体验。此功能需具备快速迭代，样式调整高灵活性的能力。因此，将卡片样式模版存放于云端，通过服务下发的形式渲染至客户端无疑为最优选择，可以满足业务快速灵活迭代的目的。

经过方案评估判断，此场景类型属于无状态服务，基于阿里云 Serverless 成熟的生态，高德最终选择接入 Node FaaS（阿里云函数计算）服务能力，出行前端搭建了场景推荐卡片服务。卡片的 UI 模版获取、数据请求聚合&逻辑处理、拼接生成 Schema 的能力均在 FaaS 层得到实现，客户端根据服务下发的 Schema 直接渲染展示，达到更加轻便灵活的目标。



那么，Serverless 场景在“十一出行节”峰值场景中的具体表现如何？

整体服务成功率均大于 99.99%，总计 100W+ 次触发/分钟，QPS 2W+，各场景的服务平均响应时间均在 60ms 以下，服务稳定性超出预期。

业务价值

从对以上业务场景的支撑中，我们可以看出 Serverless 的表现非常优秀。当然你也会问，传统的应用也能带来同样的体验，那么 Serverless 的差异化价值又是什么呢?

（一）简单提效

传统 BFF（Back-end For Front-end）层应用会随着时间推移，以及业务需求的增加，其 BFF 层逐渐变 “富”，冗余的代码逐渐变多，最后变成开发者的噩梦——“牵一发而动全身”。随着人员迭代变化，模块的开发者也会变化，BFF 层就会慢慢变成一个无人知晓，无人敢动的模块。

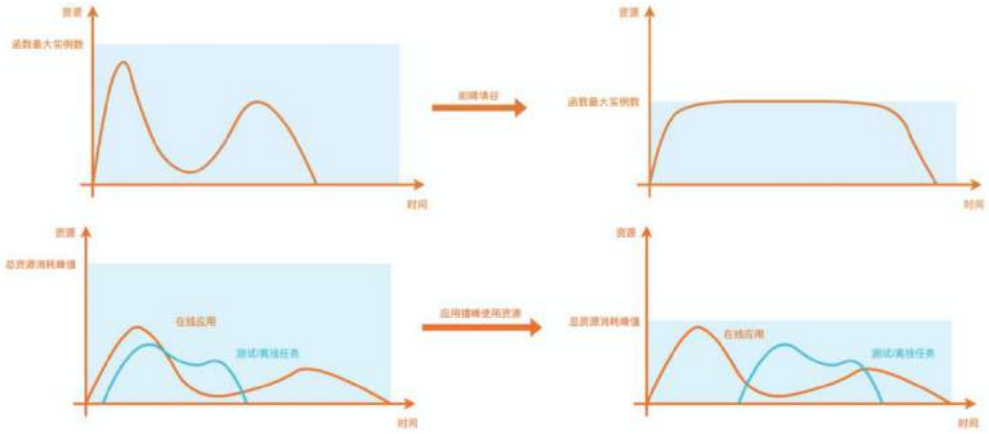
当 BFF 层转换成 SFF（Serverless For Front-end）层之后，会有什么变化? SFF 的职责会变的单一、零运维、成本更低，这些是 Serverless 本身自带的能力，而这些能力可以帮助前端进一步释放生产潜能。开发者不再需要一个富 BFF 层，而只需一个接口或一个 SFF 就可以实现功能，天然解决了“牵一发而动全身”的问题。如果接口停服或者没有流量，那么所用的实例会自动缩零，也就很容易分辨出是哪一个接口函数，后期就可以删掉此接口的函数，有效提升资源利用率。

高德在 Serverless 应用上非常先进，实现了 FaaS 层与研发体系的完全对接，因此，应用从开发、测试、灰度、上线的全生命周期，到具备流控、弹性、容灾等标准化能力，所用的时间较以前缩短了 40%，大大提高了人效。

（二）弹性以及成本

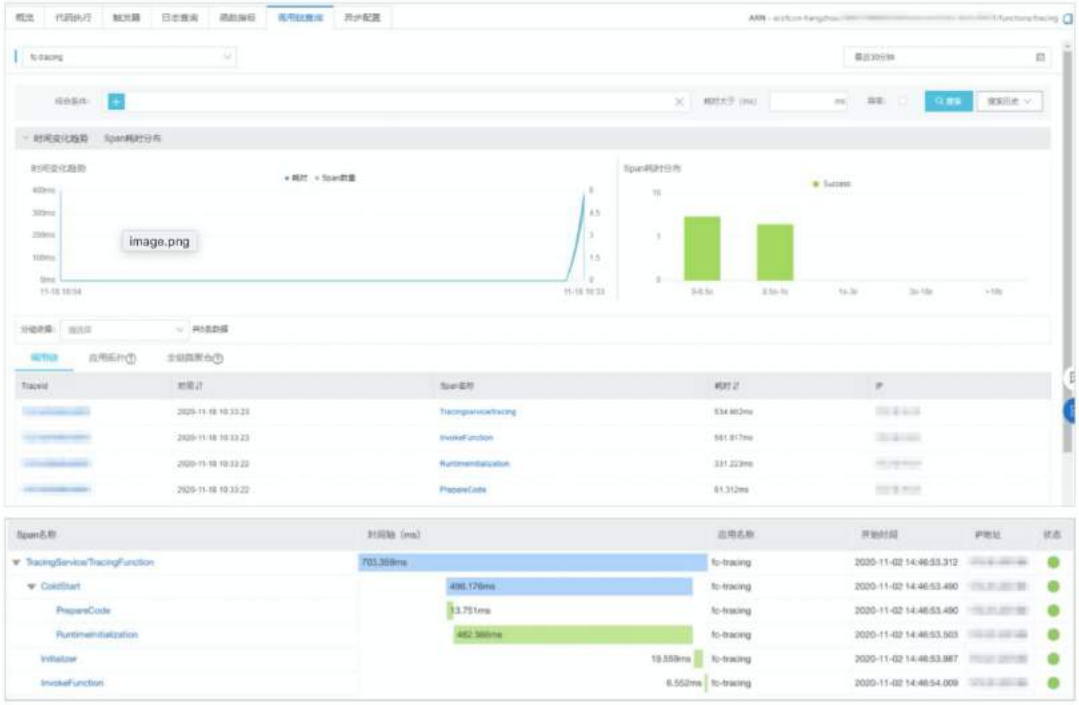
通过流量趋势数据，我们可以观察到地图场景流量特点——高峰与低峰的落差十分明显。按照传统应用的资源准备，我们需要根据最高峰的流量进行资源准备，所以到了流量低峰期，多准备的机器会有很多冗余，这就造成了成本的浪费。

针对以上情况，高德使用了阿里云函数计算，可以根据流量变化自动扩缩容。然而，提升扩缩容速度的复杂性较大，一直是大企业的专属，但函数计算可以通过毫秒级别的启动优势，将快上快下的扩缩容能力普及给用户，轻松帮助用户实现了计算资源的弹性利用，并且大大降低了成本。



（三）可观测性

可观测性是应用上线诊断平台的必备属性，要让用户观察到 RT 变化、资源的使用率、系统应用的全链路调用，从而快速诊断出系统应用的瓶颈问题。阿里云函数计算率先与日志服务、云监控、tracing 平台以及函数工作流编排做了完美的融合，用户只需要配置一次，就可以完完整整的享受到以上这些功能，大大降低了用户的学习成本，实现了对应用程序的快速诊断。



Serverless 规模化落地的序幕已经拉开， 更多场景正在各行各业中解锁。

Serverless 在高德的规模化落地，对于业务方来而言，业务迭代更快更灵活了，为业务创新创造了前提条件；对于前端开发者而言，进一步激活了开发者的生产潜能，提升了极大的能力自信。高德出行业务从 2020 年初的能力试点到“十一出行节”的自主出行核心场景，期间接入了阿里云函数计算，积累了非常宝贵的云原生落地经验，为未来业务整体上云打下了良好基础。

订单峰值激增 230%，Serverless 如何为世纪联华降本超 40%?

导读：2020 年 双 11，世纪联华基于阿里云函数计算（FC）弹性扩容，应用于大促会场 SSR、线上商品秒杀、优惠券定点发放、行业导购、数据中台计算等多个场景，业务峰值 QPS 较去年提升 230%，研发效率交付提效超过 30%，弹性资源成本减少 40% 以上。当 双 11 走过 11 个年头，传统企业正在凭借云原生技术悄然逆势崛起，参与到这场全球购物狂欢节中。

联华华商旗下的世纪联华超市近期迎来了一年一度的 “双 11” 大促活动。

时光回到 2014 年的 双 12，支付宝联合杭州多家线下品牌和商场门店，推出五折优惠促销活动。在这一次消费狂欢中，滨江区世纪联华店所有的柜员 POS 机前排起了长队，队伍移动速度异常缓慢，商场工作人员表示，服务器出现异常导致支付出了问题。

6 年后的今天，视线再次回到世纪联华超市 双 11 大促现场，人潮涌动，却井然有序。参与大促的人数是 6 年前的几倍，但现场支付却稳定又顺滑，这种天壤之别的转变，源于世纪联华对云原生技术的大胆尝试。

世纪联华大促现场

技术架构演进之路

我们有幸采访到世纪联华的技术人员，了解到世纪联华在技术架构演进一路走来的不容易。

2014 年及以前：物理单机架构的灾难

过去所有 POS 机及会员卡支付机器全部部署在各个门店，这套架构持续了近十年之久。

这套架构的最大好处是不受网络影响，在当年网络基础建设还不完善的情况下，商家可以尽最大可能地保障单个门店的交易稳定性不受外界影响。这套架构的最大问题是：当门店机器出现故障时，专业的技术管理员很难第一时间赶到现场及时修复，系统维护工作变得十分困难。

2014 年世纪联华的 双 12 活动中，因为业务遭遇爆炸性流量，多个门店支付时好时坏，短时间也无法维护，导致用户体验差，这让世纪联华的技术人决心改进这套使用了十多年的老系统。

2014~2018 年：中央机房部署架构的演进

在 2014 年经历了 双 12 大促活动的问题后，联华技术人决心改进各项系统，于是将交易系统和会员系统陆续迁移到自建的中央物理机房，商品系统也改迁为中央下发机器，在浙江省各个门店的 POS 机，通过互联网连接至中央机房。
相比 2014 年以前的架构，新架构主要解决了三个问题：

- 问题修复可集中维护处理
- 商品调整价格下发全走网络
- 数据可以集中查询统计

然而新架构遗留的最大的问题是：

- 管理人员需要掌握所有机器细节
- 运维过程中可能出现宕机、断网等事件，调查相对困难，应急处理方案薄弱

2018 年~2019 年年中：全面上云

随着国内公共云建设的进一步发展，世纪联华也开始全面使用阿里云产品，将本地业务包括 MySQL 等全部迁移到了阿里云 ECS 上。

全面上云很好地应对了宕机、计算节点断网等事件的发生，这也坚定了世纪联华使用阿里云的决心。

然而业务急剧的扩展，数据库的查询写入越来越多，全面上云的架构在 2019 年中促销活动中，某台 16 核 32G 的 MySQL 数据库所在的 ECS ，因为会员查询业务实现未做好弹性扩容准备，定时业务陡增导致请求延迟巨大，严重影响了用户体验。

世纪联华开始探索新的业务架构方式，琢磨如何使用简洁的架构实现出更高可用的业务系统。

2019 年年中至 2019 年 双 11：Serverless 的探索和尝试

随着线上业务不断发展，世纪联华的业务量不断扩大。在经历了 2019 年中数据库事件后，世纪联华经过尝试探索后，发现阿里云的表格存储服务比较适合自己的弹性业务：陡升陡降的业务不需要提前做预算来准备购买多少台存储服务器。这让联华的技术人很兴奋，因为很难算准突发的定时抢购业务量，这就导致很难预估准备多少机器来应付相应的业务量。表格存储的出现让联华技术人了解到一个词：Serverless！

在探索 Serverless 的道路上，联华的技术人偶然接触到了阿里云的函数计算，在紧张的测试验证后，技术人员发现函数计算的优异表现很契合联华的业务高度弹性的会员查询系统。

从 2019 年 7 月开始，联华技术人在不到 3 个月的时间里，将原有的会员数据全部副本镜像迁移到表格存储，并将所有渠道商的 API 全面迁移到阿里云 API 网关做分发，会员查询业务的计算业务也全面迁移到阿里云函数计算。

2019 年的 双 11，函数计算作为计算模块，表格存储作为存储模块，顺利地帮助世纪联华渡过大促，扛住高峰流量的同时确保了应对业务的弹性。而未使用 Serverless 的业务因为预估不足，出现了一些异常。

Serverless 给世纪联华带来新的曙光：

- 无需人工干预，瞬时弹性扩容很好地解决了流量的爆发带来的服务全面不可用；
- 运维管理简单，一键部署更新函数，不需要了解网络分发架构，部署流程变得更简单，无需特聘运维人员，普通研发即可操作；
- 不再需要提前做费用预算，用多少资源是多少钱，极大解决了技术和财务的沟通烦恼；

因为定时秒杀场景请求波峰波谷明显，不需要预留大量机器，从而节省了大量费用。

遗留的问题：

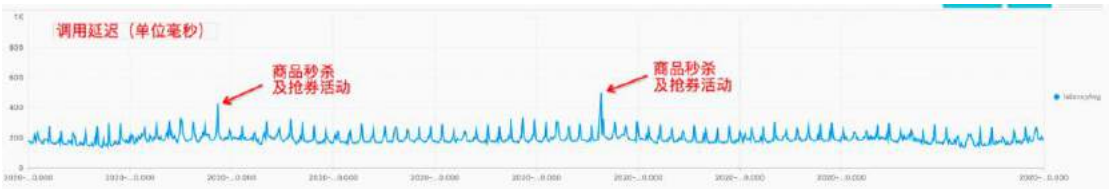
- 部分请求因为冷启动延时高导致用户体感不好，这也是当时 Serverless 开发人员普遍遇到的问题。

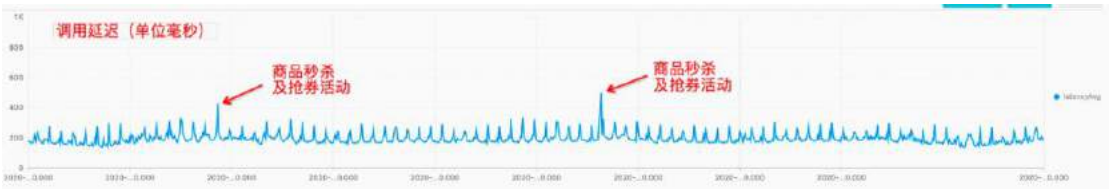
双 11 中 Serverless 的表现让联华技术人很振奋。在顺利渡过大促活动后，世纪联华很快宣布：将在所有业务中全面使用函数计算及表格存储！

2020 年的 双 11：函数计算 2.0 及全面拥抱 Serverless

2019 年下半年，阿里云函数计算宣布推出 2.0，支持预留模式，全面解决冷启动延迟大的问题；推出单实例多请求问题，较少实例支持重 IO 高并发类型请求调用；支持自定义运行时，支持一键迁移传统 Web 架构服务器。2.0 的出现让函数计算在业务和规模上实现了巨大升级。

在经历了过去的线下场景考验后，世纪联华将各渠道商的业务及旗下的“联华鲸选 APP”，以及线上交易、定时抢优惠券、秒杀业务也全部从 ECS 迁移到了函数计算 2.0，在开启预留模式调整好单实例多并发的模式后，顺利地扛过了是平时数十倍的洪峰流量请求。





比较上述的“时间-流量图”及“时间-延迟”两图可以看到，急剧上升的突发流量对用户造成的延迟变化影响非常小，从实际用户反馈来看确实也证实了用户体验非常顺滑。

所有的数据和业务上云，减轻的不只是研发人员的心理压力，还有工作量。联华华商技术负责人楼杰表示，“阿里云函数计算省了我们技术人员好多工作，我们不用管理服务器这些基础设施，只要编写代码上传，系统就会准备好计算资源，还提供日志查询、性能监控、报警等功能。这要是放在以前，超市搞 双 11 大促，我们技术团队都睡不着觉，只靠扩展机器支撑大体量的流量和业务，谁心里都没谱。现在扩容的问题交给阿里云，水位远远高于我们储备能力的极限。”

站在风口上超车：传统企业何以更需关注 Serverless？

2020 年是国内 Serverless 的技术元年，为了降低技术研发成本、提升运维效率，越来越多的企业开始选择使用 Serverless 作为基础研发底座，大力发展业务。

在近期 CNCF Serverless 研究报告中，阿里云函数计算以 46% 的占比占据国内榜首。报告同时显示，大量的国内开发人员正在将传统架构往 Serverless 上做迁移。Serverless 的出现给传统企业数字化转型带了更多机遇。

在今年的疫情当中，线上教育已经成为广大学生群体不可或缺的基础设施，阿里云函数计算也为企业提供了强大的计算力，助力企业实现视频转码成为国内在线教育 TOP 级企业。

除了新浪微博、芒果 TV、石墨文档等互联网企业，在过去的一年里，越来越多的传统企业正在以意想不到的速度接触、尝试、大规模使用函数计算。

传统企业何以更关注 Serverless 呢? 现如今，大量尖端技术人才更偏向在互联网公司就业，但传统企业又面对着大量技术升级和重构技术架构的刚需，人才缺口和技术升级之间产生了对云原生技术的需求。Serverless 的出现抹平了研发人员在预算、运维经验上的不足。在帮助企业对抗业务洪峰的情况下，研发人员能轻易掌控处理，不仅极大地降低了研发技术门槛，而且大规模提升了研发效率。对于开发者而言，线上预警、流量观测等工具一应俱全，关键是免去了运维负担，切实为广大开发者提供了普惠技术红利。对传统企业而言，Serverless 缩短了互联网公司与传统企业之间技术竞争力的距离。

由于业务场景、用户习惯迅速变化，许多行业数字化业务出现急速增长，加快数字化业务发展成为传统企业的必然选择。云原生是企业数字化最短路径，越来越多的传统企业正在拥抱云原生，借助更加快速、灵活的开发和交付模式，满足市场快速变化的需求，进而加速业务创新。世纪联华借助 Serverless 保证了一次次大促的成功，正是这一趋势的最好证明。

关注阿里巴巴云原生
了解更多技术资讯

阿里云开发者“藏经阁”
海量免费电子书下载