阿里巴巴 Alibaba.com

仅限内部传阅

全球化技术架构白皮书 1.0

GLOBAL TECHNOLOGY ARCHITECTURE WHITE PAPER

国际化中台事业部 平台架构团队

INTERNATIONAL MIDDLE PLATFORM BU

PLATFORM ARCHITECTURE TEAM

编写说明

出品单位

国际化中台 - 平台架构团队

编写组成员

毕啸 宇升 亦同 江泉 封渊
勰进 沉禹 行禅 附离 三轩
鲍鱼 箫仪 阵图 明门 鸿迪

顾问组成员

毕啸 骆杰 勿乞 北岩
荀彧 麟皓 玄宰

白皮书交流群

推荐寄语

全球化作为阿里巴巴核心战略之一，我们希望用阿里巴巴沉淀 20 年的技术和经验来帮助全世界中小卖家实现卖全球，也让全世界消费者可以方便的买全球，以及背后的全球付和全球运，全球化基础设施分散和长尾，业务模式复杂和多样，系统链路涉及面广，技术挑战极大，本白皮书综合从 10 年前 AliExpress 的全球化探索，到 2017 年 Voyager 项目全面升级 lazada 电商系统，再到 2019 年国际化中台正式成立以来我们在全球化技术征途上的种种探索和经验，希望能够让所有关注和参与全球化技术的同学有一个更加系统和全面的了解，感谢参与本白皮书编写的所有同学。

——在宽

"电商技术"可以说是过去 20 年飞速发展的互联网技术之集大成者。随着阿里业务的快速发展，阿里的技术也在不断的演进升级，创造了诸多的成绩，日趋成熟。然而当我们将视野拓展到全球时，希望能够依赖已有的技术能力快速在海外市场开疆扩土时，却又发现我们做的还远远不够，有非常多新的挑战。200 多个国家，不同的风土人情和用户习惯，极大差异化的网络条件以及监管环境让原本已经非常复杂的问题又上升了几个难度。但技术人永远是不服输的，过去的几年间，一批批全球化技术同学迎难而上，对阿里全球化技术做了非常彻底的升级，一项项关键能力与技术指标逐步与业界和国内先进水平拉齐甚至反超，为全球化业务打下了坚实的基础。非常高兴看到阿里全球化技术人这些年所挥洒的汗水能够凝聚成这本《全球化技术架构白皮书 1.0》是对过去的总结，更是对外来的开启！希望更多的同学加入到我们的全球化征程，我们的征程是星辰大海，刚刚起步！

——龙髯

一个团队对架构的重视程度一定程度上就是这个团队对技术的重视程度，全球化技术体系历经数年发展，从 17 年开始成为集团核心技术战略，一路走来，架构驱动始终是全球化技术体系演进的核心推动力。现在作为架构驱动策略下的基础产物 ----《全球化技术架构白皮书 1.0》的发表，代表着全球化技术体系在演进道路中的重要一步，即经历了野蛮生长后我们开始阶段性审视自己，希望通过这版白皮书以及后续多版白皮书的发表，持续见证全球化技术架构的不断演进和发展。

——骆杰

阿里电商全球化技术，随着阿里电商业务在多国家多区域市场深入和多业态商业模式展开，不仅是支撑交易规模和峰值持续攀升，更体现在技术对多国家复杂业务支撑能力在不断提升，更体现在快速成功将并购公司技术栈和人员融入阿里体系积累的经验沉淀，这本白皮书是这几年阿里全球化技术发展一个总结和见证，希望里面的经验能对行业同仁有帮助和启发。

——天施

17 年以来，伴随着全球化正式成为阿里巴巴集团的核心战略，上层业务和技术底层都以飞快的速度迭代发展，我们难得有机会从全局的角度总结、审视全球化技术架构解决的具体问题和面临的挑战。 从「淘系有什么」，到「全球化要什么」，架构设计更有针对性，技术底座也在不断夯实。《全球化技术架构白皮书 1.0》在介绍具体的架构方案时，都附带了当时的具体场景以及思考路径，对于新加入的同学以及已经参与其中的技术同学更加系统化地了解全局提供了很大的帮助。期望这份白皮书后续伴随着架构演进能够持续更新，也期望 2.0 版本产出时能够为海外技术同学附带英文版。

——梁楷

全球化电商相较国内，仍然处于蓝海阶段，但是其门槛和壁垒是非常高的。第一层挑战，是全球买、卖的背景下，面临性能、成本、容灾、合规方面的挑战；第二层挑战是面临全球本地模式、纯本地模式、跨境模式等多种形态的竞争对手，这些对手都从不同层面满足差异化的购物需求，建立竞争优势；第三层挑战是本地化文化、品牌心智、LTV 建立方面的挑战。面对这些挑战，全球化团队建立了面向全球的部署架构来应对第一层挑战；建立中台化开源模式来实现阿里基础电商能力复用、以及在未来将极大的实现高效的研发效率，这个效率将是我们面对全球竞对的核心优势，从而应对第二层挑战；中台模式下的 AER SAAS 化走出了本地化研发团队融合的第一步，势必在本地化方面让技术走得更深更远，以应对第三层挑战。本白皮书在对三个层面的核心思考、技术架构、核心技术方案做出了详细的阐述，是极其宝贵的技术素材，不容错过。

——验钞

对于全球化业务而言，多时区多语言多币种多区域是最基本的能力需要，所以基础业务架构天然是面向多 MKTP 的模型，围绕跨境与本对本两种业态交付多区域差异化服务能力，这是跟国内有很大差别。全球化技术架构的核心在我看来是单元化超远距的一种延伸，延伸的影响因素包括了海外物理硬件差异、区域用户隐私保护政策、地缘政治冲突风险等，很多在国内的确定性在全球化会形成不确定性的存在，比如红包多区域动态库存、数据跨区同步等。很高兴看到全球化架构团队能基于过往的成功实战，总结出这样一份白皮书，相信阅读了这份白皮书的同学不单能了解「全球化技术架构」模式，也能在思维上形成「技术架构全球化」的改变。

——颜垣

阿里巴巴走向世界实现全球化的梦想，需要面对政治经济体制差异、文化差异、经营理念差异所带来的复杂性以及技术实现的挑战，本白皮书是这一代全球化技术人在经历了 Voyager 开始的一系列全球化战役之后，基于对全球化的深度理解，经验总结，第一次体系化的沉淀并定义了阿里巴巴全球化技术架构，无论是全球化部署、全球融灾、云原生、数据合规等技术能力首先实现的是阿里巴巴技术的全球化，其次是为整个阿里巴巴全球化业务的快速发、为未来技术高效赋能业务奠定了扎实的基础。非常感谢参与本白皮书撰写的同学，相信这不仅仅是总结与分享更是一种传承，可以让不断参与进来的全球化技术人全面系统的了解全球化技术架构的细节，并基于此不断的迭代创新，共同实现阿里巴巴全球梦。

——木咚

从 Voyager 到 Apollo，伴随阿里全球化业务的突飞猛进，阿里国际化中台从无到有完成一次次迭代升级。在支持业务快速发展的过程中，技术团队有意识地思考清晰架构理念，在过程中不仅仅只是实践，而是不断总结问题，利用文档沉淀经验，逐步构建了国际化中台的架构体系，并最终产出这份白皮书。白皮书从技术视角入手，比较详细地阐述了国际化中台的基础架构、数据架构与基于云原生的应用架构的应用规范与设计指导。每一个章节，除包含了丰富的细节知识和过程指导，也有对于整体全局的思考，在了解 Know How 的过程中更加让人思考 Know Why，这是非常难能可贵的。这份白皮书也是在集团电商第一次真正形成的架构白皮书，非常具有指导与参考意义。相信对于每一个架构师来说，通读一定获益匪浅。

——**陌铭**

全球化是对阿里技术架构的终极挑战，因为它面临的不单单是一个技术问题，而是包含了政治，经济，和文化差异的复杂的生产关系问题。同时业务的形态也千差万别，从阿里完全控股，到合资，到纯技术输出以及财务投资，各种形态下对技术的要求也有极大的差异性。所以全球化技术架构最大的特点是它的包容性和可扩展性。架构总是在不断的演化，僵化的架构是没有生命力的。如何抓住变化中相对不变的要素，定义清楚关键的技术内核和协议，构建一个百花齐放，不断创新的技术新生态，是最最核心的东西。本书谈到的云原生，多租户架构，API 优先，中台开源等，都是在往开放，扩展的道路上的一些尝试。非常感谢国际化架构组的同学，把这些宝贵的经验总结，沉淀，分享给大家。全球化也是对阿里的文化，组织，和人才的大挑战。我们也希望能够有更多胸怀天下，具有包容心，勇于探索和创新的同学加入我们，一起开创全球化技术的新篇章！

——**一戒**

让天下没有难做的生意是阿里巴巴的使命，而全球化就是阿里巴巴在这个使命下的核心战略之一。全球化不仅仅是业务的全球化，也是阿里巴巴把基于数字商业的基础设施和解决方案，在全球化的过程当中推向全球，成为全球化战略的重要基石。在这个过程中，阿里巴巴耶面临了多样化的监管合规诉求，用户隐私保护政策差异，多时区多语言多币种，以及性能 / 成本 / 容灾等全新的挑战。《全球化技术架构白皮书 1.0》综合了阿里巴巴在过去 10 年的全球化征途中，在这些全新的挑战下积累的各种经验和总结，希望分享给更多有志于参与全球化建设进程的同学们。

——**得路**

2020 年 1 月初，横跨阿里巴巴、菜鸟、Lazada 等多个 BU 的全球化物流 APOLLO 2020 Battle，是经济体全球化重要战役之一，将阿里巴巴和菜鸟领先的物流数智化能力赋能全球化，助力全球本地化电商业务的高速发展，让物流成为未来商业核心竞争力之一。一年后，APOLLO 不仅在业务指标上，如单包裹成本 CPP、物流时效、OM1、NPS、PSAT 等都达到战役目标，有些甚至提前完成全年目标，而且在稳定性上，从系统割接、到 D9、D11、D12 Campaign 中全年 0 故障。这样的成绩，其中之一是离不开国际中台的全球化技术的顶层架构，包括多租户、多语言、多币种、多时区、云原生、全球合规等全球化的技术难题，都得到很好的实践和解答。更令人兴奋的是，这样领先的全球化技术架构点点滴滴，汇成一本书，功在当代、利在千秋，值得大家好好品味。

——**伟才**

马老师说：我们必须要坚持开放，必须要坚持全球化，全球化的核心是在其他国家和地区创造价值，全球化是一种服务世界的能力，我们应当坚定地走向全球。在这种理念的指导下，我们开始了国际化的征程。从 AE 到 LZD 到 Daraz 再到天猫海外，我们的系统遍布全球，覆盖了几乎世界上每一个国家。这其中，离不开阿里云和咱们电商业务系统的架构、部署和运维。在这个过程中，国际化团队完成了机房规划、系统的物理架构设计、系统稳定性和容灾能力的建设，并成功的实施了合规、成本、云原生三大战役，为阿里国际电商系统稳定运行打下了坚实的基础。这本书详细记录了从开始到现在的全部过程，有经验也有反思，非常值得大家看一看。

——**清城**

全球化征程从 Voyager 开始，我们一路高歌猛进，从未停止，踩过了无数的坑，也沉淀下来了无数的最佳实践，而本书则是回顾我们这过去几年，是怎么一步步理解全球化，一步步理解中台，一步步构建我们心目中的架构的大成之精华；希望所有阅读此书的读者，都能给到你一些关于我们在全球化板块的新的输入，帮助你了解整个全球化板块，也希望我们在整个国际化中台演进过程中的一些好的经验能带给你们不一样的思考。

——**勿乞**

从 10 年前 AE 技术开始，全球化技术分别经历了 Voyager、Sirius 和 Polaris 等重大技术架构升级，这一路走来，全球化技术团队承载着集团的战略期望，伴随着业务的飞速发展走出了一条独特的全球化技术之路。近两年，从全球化业务到全球化中台，我们不断的总结、梳理、沉淀具有全球化特色的架构标准和技术体系，所有的知识与传承都浓缩在这本宝典里面。同时，面向技术的未来，过去一段时间全球化技术团队在云原生技术体系、新的研发架构模式、国际合规等方面的探索积累也都本书里有完整定义和总结，希望这本书能够帮助读者全面了解全球化技术过去、现在和未来，在后续的全球化技术之路上少踩一些坑，少走一些弯路，能给大家的技术思维打开一扇窗，也希望有更多的同道中人一起来建设全球化技术更广阔的未来。

——**北岩**

前言

全球化是阿里巴巴集团的核心战略之一，有很多非常独特的业务特点和技术挑战。近几年，我们通过在国际化中台技术架构的探索和实践，来应对全球化带来的挑战。

这里的《全球化技术架构白皮书》旨在面向国际化业务和技术同学分享我们在全球化技术架构相关的一些思考，内容涉及多个主题，如全局架构、全球化部署、全球稳定性、全球容灾、上云、本地化、云原生、全球合规、成本优化、弹性、区域化部署、多租户等主题，每个主题包括相关的背景原则、架构方案、以及国际化相关案例，希望可以对同学们有所帮助。

面向未来，我们的业务模式趋于多元化，竞争也愈发激烈，基础设施的趋势趋于云原生，技术底盘核心主题中的稳定、性能、成本、体验、合规等给我们提出了更高的要求。如何面向未来，构建国际化的平台架构的底盘，是我们非常关键和重要的核心使命。

目录

TABLE OF CONTENT

内部资料，请勿外传

内部资料，请勿外传

【第一篇】
全球化技术架构综述

一、综述

本章作为开篇，将从整体上介绍全球化、中台和架构的相关背景知识，并简要介绍国际化中台架构的发展历程，进而展开白皮书的各章节内容。

1.1 全球化概述

1.1.1 什么是全球化

全球化（Globalization）是人类社会发展的现象过程，是全世界的人、组织、公司等交互及整合的过程。全球化发展了数十年，很多大型跨国企业奠定了一些理论体系、技术标准和实践框架，特别是随着近几年国际贸易、交通运输及信息技术的发展，全球化进程正在进一步加快。

本质上，全球化是对多样性的包容，是将平等的用户体验带给来自不同国家、不同地区、不同背景的全球用户的过程。同时全球化是循环迭代的，对企业来讲也有一定的门槛，比如需要有一定规模且有足够竞争壁垒和值得长期投入的业务，有可以快速响应及全球化专业能力的团队，以及支撑这个业务和团队的文化与技术体系。我们在讲全球化的时候，也经常提到国际化和本地化，这里讨论一下三者之间的区别和关系，如图 1-1 所示。

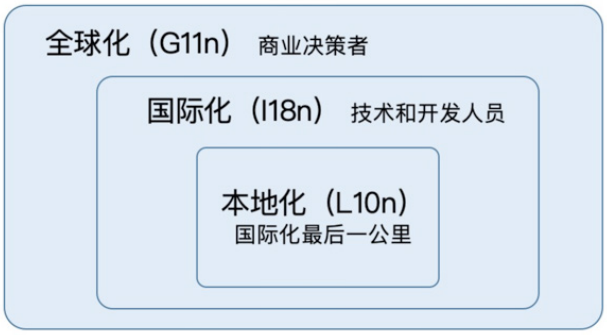


图 1-1 全球化、国际化与本地化

全球化：（Globalization，G11n，字母 G 与 n 之间有 11 个字母，下同），是面向商业决策者，为产品和服务走向全球市场而制定战略安排与规划，包括市场推广，国际品牌战略，合规运营等一系列全球范围内的商务活动。

国际化：（Internationalization，I18n），是面向技术和产品人员，将产品和服务与特定的地区解耦，

仅限内部传阅

通过标准统一的形态适用于任何地方的潜力。一些国际化相关的技术比如全球化部署、高可用、容灾处理、安全生产、代码部署分离等。

本地化:（Localization，L10n），也是面向技术和产品人员，让产品和服务更适合于特定地方的使用，“Do as the Romans do”，让用户感觉这款产品是为他们量身定制的，是全球化和国际化的“最后一公里”。一些本地化的工作比如多语言、多租户、本地化内容等。

总的来说，全球化制定了商业战略方向，国际化构建整体产品和技术能力，本地化针对不同的地区进行量身定制和落地。国际化是本地化的前提和基础，需要心怀全球用户和全局视野。

1.1.2 阿里巴巴全球化战略

阿里巴巴集团的使命是让天下没有难做的生意。阿里巴巴的业务包括核心商业、云计算、数字媒体及娱乐以及创新业务。阿里巴巴 CEO 张勇在 2020 年致股东信中提到了阿里巴巴面向未来的三大战略“全球化、内需与云计算大数据”。2024 年的战略目标是服务中国 10 亿消费者，在阿里平台上实现人民币 10 万亿元的消费规模，并基于此全面走向全球化。到 2036 年，阿里巴巴希望能够服务全球 20 亿消费者，创造 1 亿就业机会，帮助超过 1000 万中小企业盈利。从中可以看出，全球化战略作为阿里巴巴的长期之战，对整个集团有着重要的作用。尽管受到疫情挑战，阿里巴巴依然完成了五年前既定的战略目标，实现了 2020 财年（截至 2020 年 3 月 31 日止 12 个月期间）1 万亿美元的交易总额，服务了年度活跃消费者 7.80 亿，而这其中，超过 1.8 亿消费者来自海外。

图 1-2 列出了阿里巴巴经济体相关的主要业务及提供的服务。阿里巴巴围绕买家和卖家构筑起一系列交易平台，如中国零售商业、中国批发商业、跨境及全球零售商业、跨境及全球批发商业、物流服务及生活服务等。同时，在这些交易平台之上还生长着数字媒体与娱乐、创新业务、数字媒体、营销服务、支付及金融服务等一系列的生态系统。底层的阿里云等技术及系统基础设施提供了强大的技术解决方案。在这些核心业务中，我们可以看出与全球化业务相关的有跨境及全球零售商业（B2C，如 AliExpress、Lazada、天猫国际等），和跨境及全球批发商业（B2B，如 Alibaba.com）。

图 1-2 阿里巴巴主要业务及生态系统

仅限内部传阅

阿里巴巴国际站（Alibaba.com）是阿里巴巴集团最先创立的业务，根据 2020 年财报来看是中国最大的综合型外贸线上批发交易平台。Alibaba.com 为来自中国和全球的供应商提供与海外批发买家之间的询盘、线上交易、数字化营销、数字化供应链履约和金融等服务。海外批发买家一般是贸易代理商、批发商、零售商、制造商和开展进出口业务的中小企业。2020 财年，来自于约 190 个国家的超过 2,000 万买家在阿里巴巴国际站寻求商机或完成交易。

全球速卖通（AliExpress.com，文中简称 AE）创立于 2010 年，是一个全球交易市场，使全球消费者得以直接从中国乃至全球的制造商和经销商购买商品，其主要消费者市场包括俄罗斯、美国、巴西、西班牙和法国。除了全球英文版网站，全球速卖通平台还有 17 种其他语言的版本，包括俄语、葡萄牙语、西班牙语和法语等。消费者可以通过速卖通 APP 或者网站购物。

Lazada 创立于 2012 年，是东南亚领先且快速成长的电子商务平台，于印尼、马来西亚、菲律宾、新加坡、泰国和越南六个国家经营业务，为东南亚中小企业、本地及国际品牌提供服务。2020 财年，Lazada 帮助消费者接触品类丰富多样的产品，拥有超过 7,000 万消费者。上述期间内，超过 75% 的 Lazada 包裹经由其自营物流设施或“最初一公里”物流团队处理。

天猫淘宝海外作为中文版电子商务平台，使海外华人消费者得以便捷地采购来自中国的商品。在进口商业方面，2020 财年，天猫国际帮助海外品牌和零售商触达中国消费者，是中国最大的进口电商平台。此外，我们还运营了土耳其领先电商平台 Trendyol，以及主要市场在巴基斯坦和孟加拉国的南亚领先电商平台 Daraz。

1.1.3 全球化的业务特点

技术是为业务服务的，我们先来看看全球化的业务特点，主要体现在多样的市场环境、多地域发展差异、全球化合规要求、以及全球化用户体验，如图 1-3 所示。

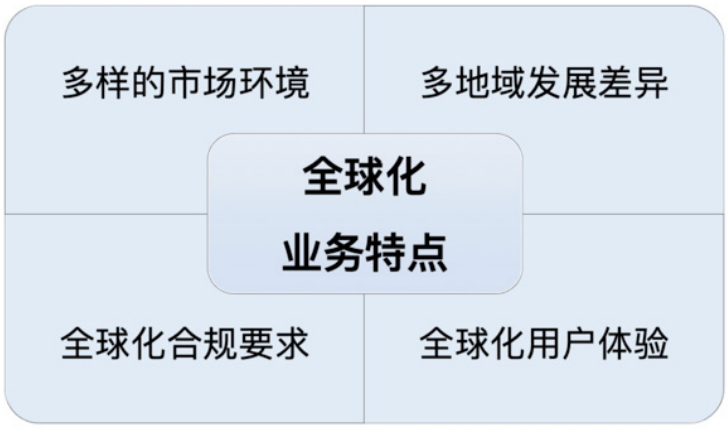


图 1-3 全球化的业务特点

① 多样的市场环境

当前全球化板块服务了 AE，Lazada，Daraz，Trendyol 等多个业务方，面临着多种多样的业务模式。以 AE 为例，核心的商业模式是全球卖和全球买，用户的分布在全球各地，业务模式有 B2B、C2C，也有本地商业、自营、团购等；而 Lazada 虽然也服务多国的用户，但主要采用的是本对本的交易，本国卖家卖给本国买家，业务模式与 AE 不完全一样。这些多样的业务模式也面临着多样的市场竞争环境，同时也带来不同的业务场景，比如第三方服务对接场景，有些中小服务商只具备服务本国家的能力，在接入全球化版图后，需要综合考虑全球的多样市场环境。

② 多地域发展差异

不同的业务方服务于不同的地域，比如 AE 服务于全球，主要面向俄罗斯、欧洲、北美和南美；Lazada 服务于东南亚 6 个国家；Daraz 服务于南亚 5 个国家。不同的地域有着不一样的时区、语言和文化，比如商品描述的度量衡、时间展示、搜索推荐排序等。不同的地域有着独特的组织阵型，通过相对独立的区域 / 国家阵型进行业务差异化发展，比如 Country team 负责运营，Regional team 负责收集诉求并制定策略，产品打法以及跟进技术落地等。同时不同地域之间除了互相独立发展之外，也希望能够进行互通和共享，比如商品互通，会员互通等。

③ 全球化合规要求

全球化的另一个重要特点是合规要求，各个国家的法律法规不同，全球化需要充分考虑全球的政治政策因素，例如隐私数据、本地存储、跨境传输等，在 AE 中俄罗斯也对数据合规提出了非常高的要求，比如不能传输到美国等。另外，多个国家也对相关的技术出口提出了相应的管制，需要我们格外注意。同时，在全球化复杂业务的发展中，也有与其他公司不同的合作方式，比如像 Lazada 这种集团全资收购的公司，以及像 AER 这种通过成立合资 JV 公司去孵化本地市场的公司，在合规方面也有所不同。

④ 全球化用户体验

全球化的业务最终服务于用户，用户体验非常关键。不同地区的用户由于语言和文化的差异，如何能够为需求各异的用户提供性能优异的本地化体验本身就是一个巨大的挑战。不同地区网络环境也不一样，还需要考虑基础设施的性能问题，比如机房内网络、公域互联网、网络基建性能等，有时还存在国家间网络几乎不可达的情况，有时两个物理距离非常接近的国家进行网络互联时甚至需要绕地球一周。另外，由于一些突发事件，比如海啸造成的机房断电，需要全球化最好应急处理响应的准备。

1.2 中台概述

1.2.1 什么是中台

了解了全球化，我们再谈谈中台。中台概念起源于阿里，也发展于阿里，2015 年阿里提出了 “大中台，小前台” 战略。其实，阿里早在 2009 年建设 “共享事业部” 开始，就已经开始了中台的探索。可以说，中台的理念已经深入到阿里内部工作的每个环节。

仅限内部传阅

中台是一套结合互联网技术和行业特性，将企业核心能力以共享服务形式沉淀，形成“大中台、小前台”的组织和业务机制，供企业快速低成本的进行业务创新的企业架构。中台又可以进一步细分，比如业务中台，数据中台，IOT 中台等。本质上，都是对企业通用能力在不同层面的沉淀，并对外能力开放。中台将企业的核心能力以数字化形式沉淀为各种服务中心，强调“一切业务数据化，一切数据业务化”，通过业务板块之间的链接和协同，持续提升业务创新效率，确保关键业务链路的稳定高效和经济性兼顾的思想体系，并突出组织和业务机制。

1.2.2 阿里巴巴中台战略

图 1-4 阿里巴巴中台战略

如图 1-4 所示，阿里巴巴超过数几十个业务单元（如淘宝、天猫、聚划算、AE）均不是独立构建的，在后端技术平台和前端业务之间有“共享业务事业部”，将业务中公共、通用的业务沉淀下来，包括用户中心、商品中心、交易中心、评价中心等十几个共享单元，是“厚平台的真正实现“。而后端的阿里云提供计算资源和中间件 PaaS 云服务能力做载体。同时，使用集团近十年的双 11、双 12 的高可靠、可稳定的运维保障能力，对整个系统进行支撑。中台的使命是从下到上逐步完善阿里的整个体系，从阿里云、数据、中间件、算法，到上面支撑的各种业务解决方案，构建阿里自己核心的能力。可以说整个阿里中台的革命也是共享服务中心的革命，各共享服务中心聚焦核心业务单元能力的构建，协助目前集团上百个前台业务的快速创新。

中台是平台化和服务化的自然演进，是包括人，组织，平台，数据，标准，规范，是人和系统的一整套体系。中台可以帮助我们应对诸如系统割裂、数据孤岛、业务响应慢等问题，可以带来激发创新、提高协同、业务能力沉淀等好处。中台建设过程中需要充分考虑整体的战略、业务流程、组织文化、技术体系等多方面，并通过小步迭代的方式持续构建。

提到中台，也会提到前台和后台。一般前台面向不同的业务进行敏捷创新，满足快速变化的市场需求，中台提供通用可复用相对稳定的共享能力，而后台一般是一些稳定的后端内部标准化的支撑系统，由

中台统一协同和对接。这其中，特别是前台与中台如何更好的协同是一个重点话题，比如中台如何提供更加稳定、高效、开发的能力给到前台团队，前台团队如何基于中台能力自主快速地进行复用和扩展，我们将在本章“国际化中台架构”进一步展开讨论。

1.3 架构概述

1.3.1 什么是架构

我们再来看看什么是架构。架构的本质是对事物复杂性的管理。在平时的工作中，我们会经常与各种各样的架构打交道，在 IEEE 这样定义架构：“架构是环境中该系统的一组基础概念（concepts）和属性（properties），具体表现就是它的元素（elements）、关系（relationships），以及设计与演进的基本原则（principles）”。我们一般接触的是软件 IT 架构或者技术架构，不过从整体来看是属于企业架构的范畴，主要包括业务架构、数据架构、应用架构、技术架构，如图 1-5 所示。企业架构是构建业务与 IT 之间的桥梁。企业架构已经经过近 30 多年的发展历史，有很多架构框架理论，比如 Zachman、TOGAF、DoDAF、ETOM 等。

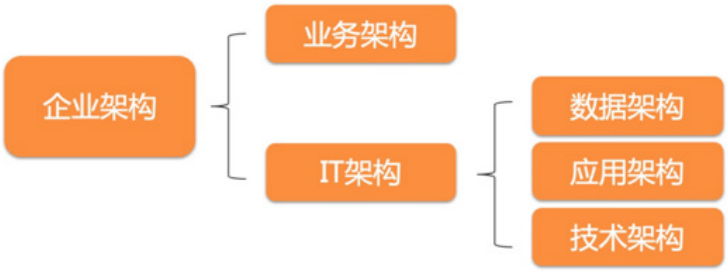


图 1-5 企业架构范围

1.3.2 什么是技术架构

技术架构，也是我们经常说的软件架构，或者系统架构，是将业务需求转变为技术实现的规划过程。技术架构可以帮助我们梳理系统边界、识别系统需求、识别系统风险和问题优先级、确定技术方案和路线、让团队之间达成共识和约束，并反向指引团队来适应业务变化。

技术架构解决的问题包括了如何进行纯技术层面的分层、开发框架选择、语言选择、涉及到各自非功能性需求的技术点。技术架构是确定组成应用系统实际运行的技术组件、技术组件之间的关系，以及部署到硬件的策略。技术架构有多种常见的架构模式，比如分层、事件驱动、微服务、大数据等，我们将在第二章全局架构中对业界主流架构模式进一步分析。这里我们先简单看看国际化中台主要经历的几种架构模式，如图 1-6 所示。

图 1-6 国际化中台架构模式

仅限内部传阅

单体架构: 所有业务逻辑全部在一套系统中，大而全的体系，没有服务化，没有分层，单机集中式数据库，存储过程较多；这种架构适应业务发展早期，在产品的完善性、系统扩展性、部署规模的要求都不高的背景下。

分布式架构：引入中间件和分布式数据库，通过服务化、微服务技术进行系统的拆分，并通过通过服务化以及消息等形式进行交互；这种架构适应业务发展中期，产品的完善性和精细性有了一定的需求，团队也越来越大，扩展性主要依赖服务化的能力，部署层面主要在单一机房，或同城容灾，国际化特性较少。

中台架构：通过中台理念，有对应在各区域的前台团队，以及服务前台构建通用能力的中台团队，在复用和个性扩展之间寻求平衡，通过插件或者扩展点机制进行扩展，同时全球多区域、多机房，需要考虑全球化部署、容灾、合规等全球化特性。这种架构方式适应于复杂的国际化业务场景，同时在云原生时代，国际化中台正在全面拥抱云原生的技术，典型的有国际化云原生样板间，今年双十一中我们有近 10 个核心系统完全升级到云原生架构，覆盖核心交易导购链路。

1.3.3 全球化技术架构挑战

在介绍了全球化、中台、以及架构之后，让我们来看看全球化技术架构面临的诸多挑战，主要体现如下几个方面：

① 性能维度：部署、成本与上云

全球化对性能有更高的要求。首先体现在用户请求时延方面，在全球化的背景下，物理距离变得更长，用户请求时延会进一步放大。比如，美国用户请求美国机房的网络 RTT 往返时间在 10ms 以内，而俄罗斯用户请求美国西部机房的网络 RTT 在 150ms 到 300ms 之间，这直接导致用户的全屏加载时间会多出 1s，会造成交易达成的转化率下降，甚至用户流失。这种性能的要求进一步对部署提出了更高的要求，比如是全球化部署还是本对本部署，机房如何选址，自建 IDC 还是使用云服务，网络基础设施如何选择，使用什么样的中间件和数据库，以及相关的成本计算，资源优化，数据如何同步、用户如何路由等。可以看出，全球化的性能特性与全球化部署、成本优化等紧密相关，同时随着云计算云服务的普及，通过“上云”使用云上服务来进一步提升效率降低成本也成为主流方式。图 1-7 展示了当前全球化基础设施的 IDC 布局，可以看到当前在全球多个区域有多个物理及逻辑机房，包括云上和非云形态。

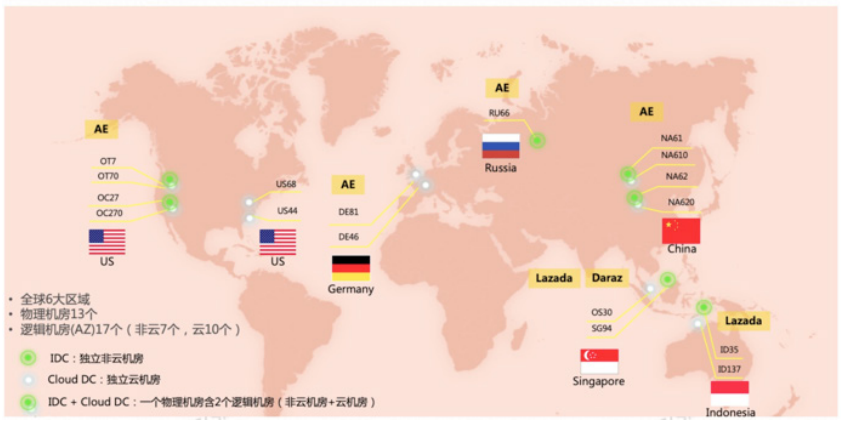


图 1-7 全球化基础设施的 IDC 布局

② 稳定性维度：高可用、容灾与安全生产

全球化的业务会跨越时区，这就要求我们的服务要 7X24 小时稳定可靠。北京的白天，正好是纽约的夜晚，这种对稳定性的要求不仅是对系统的挑战，也是对组织团队的挑战。构建高可用需要多种方法，比如数据的冗余备份、服务的限流降级、熔断隔离等；同时站在全球视角，还需要考虑不同时区的闲置资源，做到全球范围的容灾，将极大地节省成本，比如异地多活，全球容灾，灵活对容灾进行切流。而在保障稳定性和容灾的同时，我们需要重点考虑数据的一致性，比如当全球数据中心之间相互备份、容灾时，可能存在多地读写的情况。同时，基于阿里国际化的特殊业务形态，我们还需要做好各种重要节点的安全生产，比如在双十一大促之前，我们对多个业务进行了细致的大促保障，包括核心链路梳理、容量评估、监控告警、应急预案、全链路压测等。图 1-8 展示了 2020 年全球化双十一全链路示意，通过大促安全生产保障，保证全球 6 区域，16 机房万笔交易。

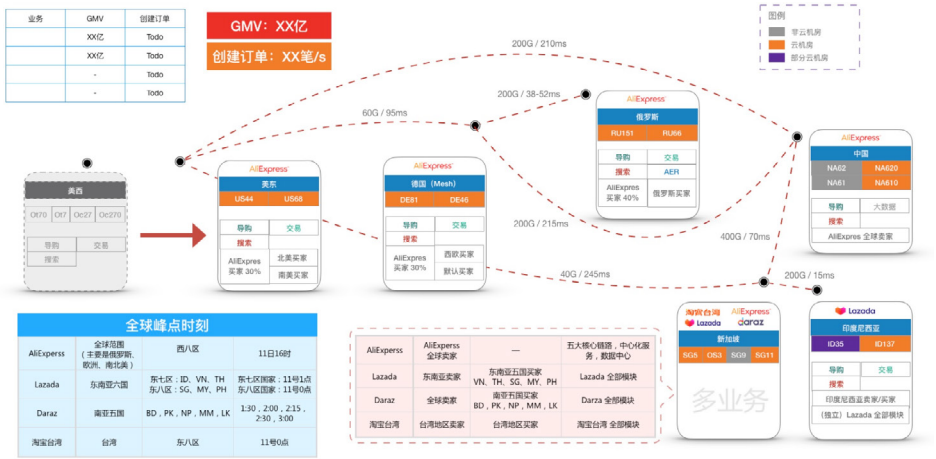


图 1-8 2020 双十一全链路大促保障

③ 隐私保护维度：全球合规

全球化对于隐私保护非常重视，比如 2018 年 5 月份，欧盟就出台了 GDPR（General Data Protection Regulation 通用数据保护条例），这个是继 1995 年之后，更加严格的个人隐私保护的条例。在目前国际的形式下，各个国家对于个人隐私数据也越来越重视，对于知识产权，数据安全，隐私安全，制裁等等，图 1-9 展示了 GDPR 条例的一些概要内容。从趋势上看，全球化的隐私保护越来越需要安全

仅限内部传阅

合规的技术支撑。合规需要考虑的核心是数据是否必须本地存储，是否可以复制出境，隐私政策是否明确说明出境的情况，用户是否需要同意，是否有合同要求，是否需要向监管机构登记或者备案等。对于技术层面，我们需要做好对应的 IDC、数据存储、数据传输、数据脱敏、流量调度等一系列技术准备。

图 1-9 国家合规的架构框架

④ 多地域维度：本地化、区域化与多租户

全球化多地域的业务发展差异，带来了技术上的不同挑战，首先是本地化的挑战，阿里巴巴集团 CEO 张勇曾说过“全球化不是在前搞采购，而是培育当地新生态”。从生态角度阐述了全球化业务与本地化不可分割的关系。而本地化需要从用户的整个交易链条相关的物流、资金流和信息流等能力，比如需要对接本地化的物流、支付体系等，需要技术做好不同的对接能力。同时多地域的技术挑战也体现了区域化方面，比如区域化下的部署和路由策略，数据一致方案等。多地域也对多租户提出了新的要求，如何通过物理多租和逻辑多租使得资源利用达到最优。另外，多地域对多语言多文化也有要求，不同国家有特殊的语言展示方式和习俗，在技术方面需要做好可扩展性，资源方面可以通过水平扩展应用，或者通过服务扩展和编排、以及界面的配置能力提高多地域的快速响应。图 1-10 展示了全球化的应用系统，可以看到目前国际化中台当前覆盖多地的 1000+ 应用系统，9000+ 应用分组，约 20 个 AZ（机房），以及 18+ 单元。

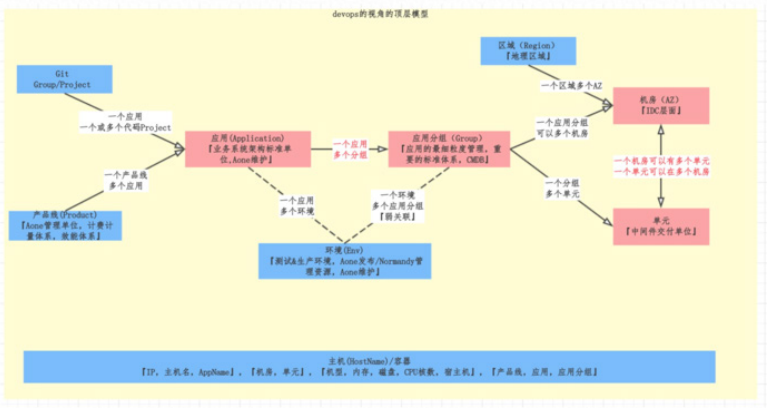


图 1-10 国际化多地域应用分布

1.3.4 全球化技术架构三边战略

面对全球化复杂的业务特点和技术挑战，全球化中台平台架构团队在近几年的技术架构探索中，通过“三

边战略”来指导技术架构的方向，以及保证架构到代码落地的一致性。三边战略指的是在技术架构规划和执行落地的过程中，需要守住三个边，一个边是基础设施，一个边是用户体验，一个边是架构规约和工具，如图 1-11 所示。

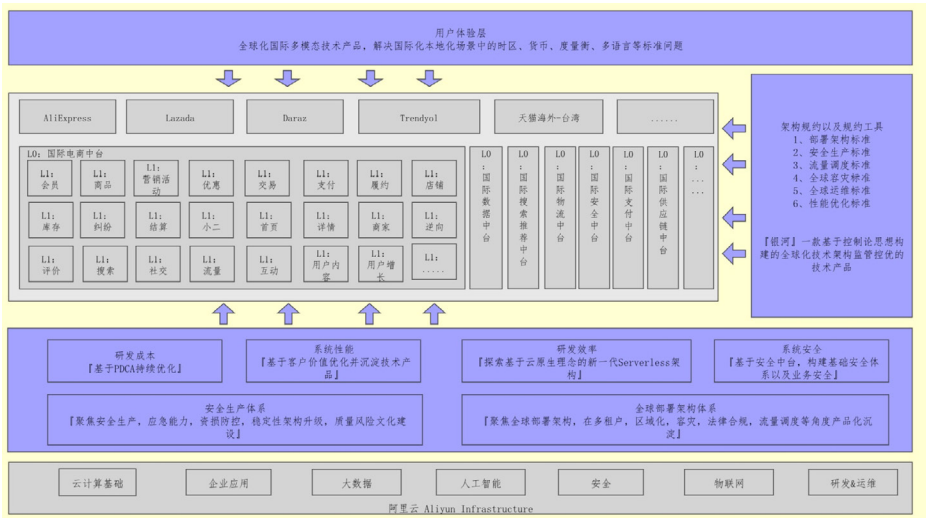


图 1-11 国际化中台技术架构三边战略

① 基础设施边：基础设施一般围绕稳定、性能、效率、成本这些主题展开，同时需要考虑全球化技术挑战中的全球化部署、全球容灾、国际化高可用等多个方面，是技术架构中需要重点解决的基本问题；
② 架构规约和工具边：架构是抽象的，需要一定的原则规约进行指导，比如大促保障规约、全球合规规约等，以及这里讲的全球化技术白皮书；同时架构也是具体的，最终需要落地到具体项目或者代码当中，为了更好的标准化、自动化以及度量，需要通过工具来进行落地，例如部署规则，容灾规则，流量调度规则等。这个边对于全局技术架构的一致性非常重要；
③ 用户体验边：架构是服务于业务的，好的架构需要屏蔽掉复杂度，把简单留给用户。一方面是从用户体验上将前前端体系逐渐收敛，保持一致性，比如国际化小程序体系；另一方面是面向技术同学，从产品的视角，带来研发和运维的一体化产品体验，这个我们也正在积极探索。

三边战略的思考框架借鉴了一些前人的经验，主要从业务分析、基础设施趋势分析、核心问题分析、架构哲学和理念、客户价值分析，再到需要的核心能力分析，进而推导出技术大图，从而细化出技术战役。这个过程秉承着一些外围架构哲学，作为总体的指导原则：
① 全球中台云：三位一体的约束条件，即全球化 + 中台 + 云作为前提，全球化是业务特点，云是未来的基础设施，中台是典型的架构模式；
② 数据驱动 PDCA：全球化的技术跳转需要不停的迭代优化，需要持续采用 PDCA（Plan Do Check Action）的数据驱动理念，对架构进行可视可管可控；
③ 技术微观化管理：从整体用户和系统研发视角，基于开发，测试，发布，运维等全过程，借助清单理念，做微观化管理。

仅限内部传阅

1.4 国际化中台架构

1.4.1 国际化中台架构演进历程

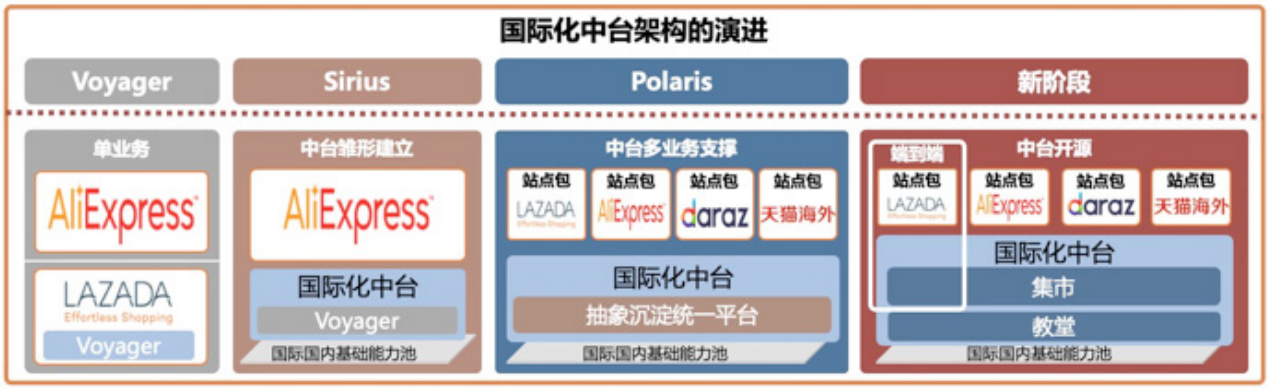


图 1-12 国际化中台架构演技历程

国际化中台事业部是集团新零售技术事业群的一员，为集团全球化战略提供坚强的技术支撑。我们先来简要回顾下国际化中台的架构演进历程，如图 1-12 所示。国际化中台孵化于集团的业务平台，所以在中台建设上继承了业务平台的很多思想和经验，国际化中台的核心系统也直接缘于业务平台。

第一代（单业务）：2017 年 Voyager 项目，基于国内业务平台、淘宝、天猫、搜索等团队的系统在 6 个月的时间搭建了全套支持 Lazada 的新电商系统。2018 年电商核心系统开始和业务平台的系统进行融合，形成国际国内一个基础域，Lazada 和 AE 两套上层实现的架构。构建了国际化单业务电商解决方案，沉淀了全球多中心逻辑隔离部署标准。

第二代（中台雏形建立）：2019 年初 Sirius 项目，以 Voyager 为内核，和 AE 系统进行了深度融合，同时引入了淘宝、天猫等团队的优秀系统解决方案，形成了可同时支持本地 + 跨境交易模式的国际化中台模式。基于 Voyager 技术栈，完成了国际化中台雏形建立，并助力 AE 业务增长。

第三代（中台多业务支撑）：2019 年底 Polaris 项目，以 Sirius 融合版本为基础，合并 Lazada、Daraz、TMOVS。完成了 4 个国际化平台版本统一，形成了 1 个中台支撑 4 个站点的全球化架构，构建了国际化中台业务研发体系，做到了多业务并行研发。

第四代（中台开源新阶段）：2020 年，为了更快地支撑业务敏捷发展，正在开展中台开源模式新阶段，让业务先跑，让平台后置沉淀的方式让业务迭代节奏和平台沉淀节奏解耦，让协同成本更低，并通过开放标准的中台支撑全球化研发效能大幅提升。

1.4.2 国际化中台开源模式

图 1-13 国际化中台开源模式

目前国际化中台架构的关键是开源模式，如图 1-13 所示。最下面是基于云原生的基础设施层，是整个全球化技术的底盘，四个业务完全统一。在此之上，我们把传统的交易、商品、支付等领域在架构上分为了三层，底层是跟国内的业务平台共享的核心领域模型层，这层无论是模型还是基础能力均采用教堂模式，只有国际中台的同学和国内业务平台的同学可以修改这一层的代码，不过改动频次相对较低；中间层是基于模型的流程、节点、扩展点等，这层业务和中台同学可以基于集市模式开源共建；上层是站点包，类似业务平台上面业务包、APP 的模式，这层完全由业务同学独立负责。我们希望当业务提出一个需求之后，70% 的需求可以通过标准的完全自助的站点包的模式解决，剩下 20% 的需求可以通过修改集市层代码完成，剩下 10% 涉及到底层模型或基础能力修改的需求由中台协助开发完成。关于代码分支管理，中台版本迭代里面四个业务可以分别 Fork 自己的分支，独立迭代和发布，不定期地集成到中台主干上，完成业务先赢和中台沉淀的完整迭代。

1.4.3 国际化中台核心思想

国际化中台在建设过程当中，我们坚持有几个核心思想，如图 1-14 所示：

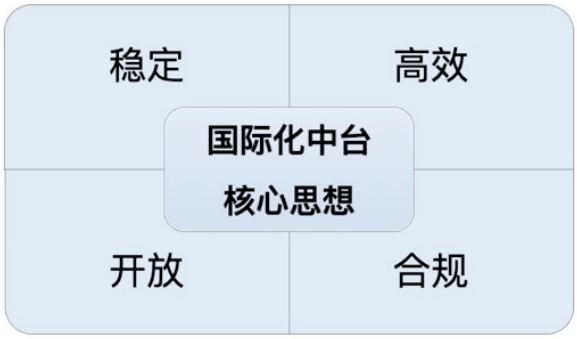


图 1-14 国际化中台核心指导思想

① **稳定**：稳定是一切的基础，特别在多次大促中，中台需要时刻关注可用性，并通过相应机制和体系

仅限内部传阅

保障，比如鼓掌分、峰值稳定性表现等。

② **高效**：需要让中台之上的业务开发团队更高效，比如业务小闭环自助率、站点包自助率、业务需求交付时长、业务变更的前置时长等。

③ **开放**：中台开源之后业务代码可以自发集成中台主干，业务方在贡献能力的同时也能复用别的业务或中台沉淀好的能力，比如中台主干业务方集成比例等。

④ **合规**：全球合规是国际化关键的技术挑战，比如全球部署、流量灵活切换、数据灵活过滤等，需要推动整个电商、支付、物流一体化的合规标准。

1.5 白皮书内容概述

这本《全球化技术架构白皮书》是面向国际化业务和平台的技术同学，目的是与同学们分享国际化中台团队在这几年对于全球化技术架构的一些沉淀和思考，也欢迎各位提供宝贵建议，我们后续进行优化和完善。基于前文关于全球化、中台、架构以及国际化中台架构的介绍，在后面章节中我们将对各个主题进行单独阐述，主要包括：

- **全局架构**：全局架构常见架构风格、设计原则与理论、全局架构图与外化等。
- **基础主题**：涉及全球化部署、全球稳定性、全球容灾、上云、本地化，包含各主题的国际化背景、原则规范、架构方案、以及国际化相关案例等。
- **全球化部署**：基于全球化特性的部署架构形态，比如本地部署、跨境部署、独立部署、中台部署等，介绍全球化部署的方案演进、落地规范、部署分布与数据同步。
  - **全球稳定性**：针对全球化特性的稳定性体系，保障系统的高可用，介绍稳定性通用原则、容灾快恢、灰度发布、监控体系、流量限流、以及大促稳定性保障体系。
  - **全球容灾**：稳定性的一种特殊情况，介绍国际化容灾的定义和背景、挑战和原则、多种容灾部署架构、以及多种国际化容灾方案。
  - **上云**：以云服务为基础设施能力建设系统，介绍国际化上云背景、应用架构、总体原则、上云方法体系、产品选型、以及国际化 ASI 上云案例。
  - **本地化**：让产品和服务更适合于特定地方的使用，是全球化和国际化的“最后一公里”，介绍多语言、多时区、多币种等本地化相关内容。
- **高阶主题**：涉及云原生、全球合规、成本优化、区域化、多租户，包含各主题的国际化背景、原则规范、架构方案、以及国际化相关案例等。
  - **云原生**：云原生是云计算的下一站，介绍云原生背景、架构原则、核心技术、国际化云原生基础架构、国际化云原生应用架构、以及云原生架构演技方向。
  - **全球合规**：国际化需要遵守相应的法律法规和监管规则，比如对用户的隐私数据进行保护，介绍全球合规相关的背景、架构、原则、工具、实施步骤、实践案例。
  - **成本优化**：是指对成本的分析和优化，介绍成本优化背景、基本框架、基本原则、计费规则、优化策略，并突出对组织架构和成本意识的重要性。
  - **弹性**：体现了一个系统是否能够随着压力的增减而动态的对资源进行调整的能力，介绍弹性资源、体系化弹性、容器弹性、国际化弹性产品、弹性稳定性等方面。

- **区域化部署**：在全球的多个区域按需部署应用程序为用户提供服务，介绍区域化部署的背景、整体技术架构、路有体系、以及云原生下的区域化展望。
- **多租户**：多用户的环境下共用相同的系统或程序组件，并保持隔离型，介绍多租户的背景知识、国际化多租的背景和历程、总体原则、相关技术方案等。

仅限内部传阅

二、全局架构

本章聚焦在架构相关的基础话题上，比如全局架构和局部架构，架构师需要具备的基础能力，常见的架构风格，需要关注的架构主题，以及怎么来画好架构图等等。这些话题并没有标准答案，本章以一些比较好的实践为例，希望对大家从全局视角理解架构有所帮助。

首先，什么是全局架构？业界并没有明确的定义，我们认为全局架构是针对一个企业的一条产品线或者 BU 级别产品矩阵而言的架构形态，不仅要考虑技术架构，还需要考虑基础设施、物理架构、网络架构以及各个局部架构之间的互联互通及相互依赖关系。从更大一个层面来看，需要从企业架构的视角，如从业务、数据、应用、技术多个视角来看架构，也需要描述多个视角之间的关系，借助自顶向下，分而治之等方法论辅助，比如图 2-1 所展示的全局架构模型。

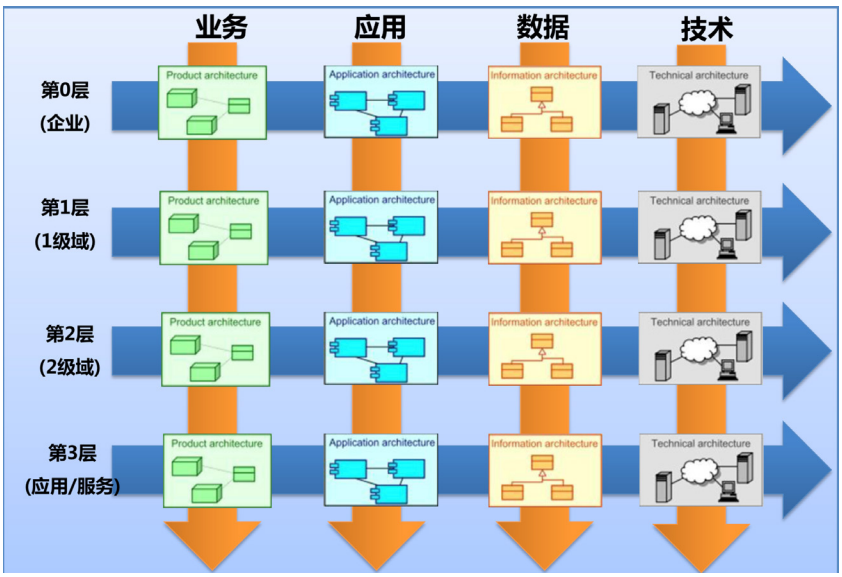


图 2-1：全局架构模型

局部架构，是相对于全局架构而言的，可以简单理解为我们通常讲的架构。局部架构的范围可大可小，最小的架构可以是一个功能的流程图或者相关模块的关系。相对大一点而言，设计一个应用，需要考虑外部依赖、核心服务、数据模型、中间件选型、存储选型等，都需要从架构层面进行设计，某业务的区域化部署架构、稳定性架构的压测模型、云原生样板间的应用站点交付架构等。用经典的架构模型可以这样理解：

- **经典公式 1**：架构 ={ 组件 , 关系 , 原则 / 约束 }
- **经典公示 2**：RUP 的 4+1 视图 (逻辑、进程、物理、开发、用例)

仅限内部传阅

2.1 架构师基础能力

2.2.1 架构思维

架构的本质是管理复杂性，抽象、分层、分治和演化思维是我们工程师 / 架构师应对和管理复杂性的四种最基本武器。

① 抽象思维

抽象 (abstraction) 是对某种事物进行简化表示或描述的过程，抽象让我们关注要素，隐藏额外细节。在系统架构和设计中，抽象帮助我们从大处着眼（get our mind about big picture），隐藏细节（temporarily hide details）。抽象能力的强弱，直接决定我们所能解决问题的复杂性和规模大小。架构师先要在大脑中形成抽象概念，然后是子模块分解，然后是依次实现子模块，最后将子模块拼装组合起来，形成最后系统。这里也提一下抽象层次跳跃问题，这个在开发中是蛮普遍的。有经验的程序员写代码会保持抽象层次的一致性，代码读起来像讲故事，比较清晰易于理解；而没有经验的程序员会有明显的抽象层次跳跃问题，代码读起来就比较累，这个是抽象能力不足造成。

② 分层思维

除了抽象，分层也是我们应对和管理复杂性的基本思维武器，为了构建一套复杂系统，我们把整个系统划分成若干个层次，每一层专注解决某个领域的问题，并向上提供服务。有些层次是纵向的，它贯穿所有其它层次，称为共享层。分层也可以认为是抽象的一种方式，将系统抽象分解成若干层次化的模块。分层架构的案例很多，一个中小型的 Spring Web 应用程序，我们一般会设计成三层架构，操作系统是经典的分层架构，TCP/IP 协议栈也是经典的分层架构。

③ 分治思维

分而治之 (divide and combine 或者 split and merge) 也是应对和管理复杂性的一般性方法。对于一个无法一次解决的大问题，我们会先把大问题分解成若干个子问题，如果子问题还无法直接解决，则继续分解成子子问题，直到可以直接解决的程度，这个是分解 (divide) 的过程；然后将子子问题的解组合拼装成子问题的解，再将子问题的解组合拼装成原问题的解，这个是组合 (combine) 的过程。

④ 演化思维

社区里头经常有人在讨论：架构是设计出来的？还是演化出来的？我们认为，架构既是设计出来的，同时也是演化出来的，对于互联网系统，基本上可以说是三分设计，七分演化，而且是在设计中演化，在演化中设计，一个不断迭代的过程。在互联网软件系统的整个生命周期过程中，前期的设计和开发大致只占三分，在后面的七分时间里，架构师需要根据用户的反馈对架构进行不断的调整。架构师除了要利用自身的架构设计能力，同时也要学会借助用户反馈和进化的力量，推动架构的持续演进，这个就是演化式架构思维。

仅限内部传阅

2.2.3 架构师能力

这里借鉴蚂蚁对架构师能力的定义：

① 首先，需要有业务洞察能力，全局架构师更多的偏业务架构和应用架构，因此对于自己所负责的领域要有独到的理解。就拿支付宝来说，你要理解它做的支付本质是什么？解决什么问题？底层是什么？包括支付、账务、清算、结算、交易平台等等，都需要有自己的理解。

② 其次要保持心态开放，有些问题一线的同学可能感受没那么强烈，比如两个团队之间的一些合作效率上的问题，但架构师要来解决的话可能伤害到同学们，那么到底要不要解决呢？这里就涉及到具体的权衡，而在这个过程中，要保持开放的心态，要能够容忍各种矛盾。

③ 第三个是架构师需要具有全局视野，做架构本质是权衡和取舍，如果决策的时候只能看到一部分，那么结果就是盲人摸象，决策肯定会出问题。要解决局部领域的问题，需要从全局视野出发，否则最后的方向可能与整个公司的方向背道而驰。而这条对于全局架构师更加是必备的素质。

④ 最后是紧盯结果，需要很强的推动力，架构师不是画几张图就完了，还必须把它落实到架构的原则、架构的元素、系分方案的评审中。因为全局架构师独立于业务线之外，在这样一个环境里推动一个事情往前走是非常困难的，涉及很多的团队协作和沟通。需要有鲜明的观点，并且要能说服很多人，才能得到想要的结果。

2.2 架构风格

2.2.1 传统的架构风格

① 分层架构

在软件工程中，分层架构是一种客户端 / 服务器端架构。在该架构中，表现功能、应用处理和数据管理功能物理分离。最常使用的分层架构是三层架构，分层架构提供了一种模型，使得开发者可以建立方便扩展和复用的应用。通过将应用分成多层，开发者拥有修改或增加一个特定层的选择，而不是重写整个应用。每一层都是围绕一种功能的抽象，各负其责，有利于系统开发、测试、管理和维护。

适用于传统业务领域，需求更新比较少。优势是可移植性好，开发学习曲线少，适合从传统应用模型进行演进，对异构环境的适配性强。挑战是会增加额外的延迟，单体式设计阻碍了功能的独立部署，管理 IaaS 应用的工作量更大，在大型系统中管理网络安全比较困难。

② 事件驱动架构

事件驱动架构是使用高解耦、单一用途的事件处理组件来异步接收和处理事件的架构。事件驱动也是十分流行的架构，特别是在分布式、异步系统中。事件驱动模型由于其天然的异步性，是一种相对复杂的模式。由于其异步性，很难保证事务的原子性。如果你的业务逻辑中很少有原子性的事务则可以

选择它。除此之外，事件驱动架构的代码编写、维护、管理都相对困难。尽管事件驱动有如此多麻烦问题（编程要求高），但是其异步特性和事件处理的机制是很多程序的首选。

事件驱动架构的优势是生产者 / 消费者解耦；高度可扩展和分布式；削峰填谷；响应速度快。同时事件驱动架构的挑战是事务性；确保事件不丢（Guaranteed delivery）；按顺序或仅一次处理事件（Exactly once）。

③ 微内核架构

微内核架构（Microkernel Architecture），也被称为插件化架构（Plugin-in Architecture），是一种面向功能进行拆分的可扩展架构。例如 VS Code、Eclipse 这一类 IDE 软件、UNIX 操作系统等等，都是参照微内核架构设计实现的。微内核系统架构模式由两块组成：核心系统和插件系统。应用逻辑分为独立的插件模块和基本核心模块，继而有扩展性、灵活性同时隔离应用程序功能和处理特定事物逻辑。

微内核架构的优势是具有良好的功能延伸性（extensibility），需要什么功能，开发一个插件即可。功能之间是隔离的，插件可以独立的加载和卸载，使得它比较容易部署。可定制性高，适应不同的开发需要。可以渐进式地开发，逐步增加功能。挑战是扩展性（scalability）比较差，内核通常是一个独立单元，不容易做成分布式。开发难度相对较高，因为涉及到插件与内核的通信，以及内部的插件登记机制。

④ 微服务架构

微服务体系结构的每个组件都可以作为一个单独的单元进行部署，允许通过有效且精简的交付管道更容易地进行部署。微服务架构由于其实用性，成为代替单体架构、面向服务架构的选择，并获得了广泛的关注。微服务架构由一组小型自治服务组成， 每个服务都是独立的，实现单个业务功能；微服务是比较小型的、独立的、松耦合的，一个小的开发人员小组可以编写和维护服务；每个服务都是一个单独的代码库，服务可以独立部署，可以单独更新而无须重建和部署整个应用；服务负责持久化自己的数据或者外部状态，而不是像传统分层架构中有单独的数据层处理数据持久化；服务通过使用定义良好的 API 进行通信，每个服务的内部实现细节对于其他服务都是隐藏的；服务不需要共享相同的技术栈、库、或者框架。

微服务架构的优势：敏捷，研发运维效率提升；可扩展性，服务可以独立扩展，同时也易于支持架构演进；支持异构技术；故障隔离，如果单个微服务不可用，可以通过熔断等技术控制故障影响范围；数据隔离，更新数据仅会影单个微服务，相对单体式应用风险小；更小的代码基线（small code base），通过不共享代码或数据存储，微服务架构最大程度地减少了代码依赖关系，添加新功能更加容易。

微服务架构的挑战：复杂度变高；部署和测试困难；缺乏治理；容易网络拥塞和延迟；数据的完整性，数据的一致性可能是一个挑战；管理问题，需要由成熟的 DevOps 文化；版本控制，向后或者向前兼容性的问题；团队技术储备是否足够。

仅限内部传阅

⑤ 大数据架构

大数据架构是用于摄取和处理大量数据的总体系统，因此可以针对业务目的进行分析。该架构可视为基于组织业务需求的大数据解决方案的蓝图。大数据架构旨在处理以下类型的工作，主要包括如下几个部分：批量处理大数据源，实时处理大数据，预测分析和机器学习。

大数据架构优势是应对大规模数据处理，通过并行度提升性能，弹性伸缩，与现有解决方案的互通性（如用于 IoT 处理或者企业 BI 解决方案）。大数据架构挑战在于复杂度提升，团队技能储备是否足够，大数据架构中使用技术的成熟度，以及数据安全性问题。

⑥ 大型计算架构

大型计算一词描述了需要大量内核的大规模工作负载，通常需要数百或数千个内核。场景包括图像渲染，流体动力学，财务风险建模，石油勘探，药物设计和工程应力分析等。典型特征就是可以把任务切割为更小粒度的任务以便于并行化执行。

此架构的优势在于并行化处理带来的高性能，可以解决大型问题。挑战在于管理虚拟机基础架构，需要及时配置数千个内核，对于紧耦合的任务，添加更多内核可能会减少收益，需要进行反复试验。

2.2.2 演进的架构风格

① 云化架构

企业上云今天已经成为广泛接受的必然趋势（Rightscale state of the cloud report 2019 显示，94% 企业已经在使用云，其中公有云 91%）。企业的数字化转型的过程中，利用云的能力的过程也分为不同的阶段，一般来说也会是走过和阿里上云类似的过程：首先是弹性使用云的资源，实现部分业务上云，Cloud-hosting。在此过程中，一般是非核心系统使用云资源。然后涉及到核心系统的云化，这里发生的变化不仅仅是上云的应用的数量，更是底层基础设施整体使用云平台的能力的过程。

云化架构带来的优势：极致的弹性，可有效应对电商大促等流量峰值，按需付费；加速企业 IT 设施发展；降低总体 IT 成本；提升系统稳定性；提升应用开发部署敏捷性；可以有效整合优化设计、生产和市场资源，实现产业链上下游的高效对接与协同创新，重塑生产组织方式和创新机制。云化架构带来的挑战：企业数据安全；团队技术储备是否足够；整个架构升级带来的一系列风险问题；

② 云原生架构

CNCF（云原生计算基金会）在 2018 年对云原生进行了重新定义：云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

云原生本身甚至不能称为是一种架构，它首先是一种基础设施，运行在其上的应用称作云原生应用，只有符合云原生设计哲学的应用架构才叫云原生应用架构。云原生架构是依赖云产品和云原生技术构建的 IT 架构，生于云长于云最大化运用云的能力，让开发者聚焦于业务本身。

云原生系统的设计理念如下：

- **面向分布式设计**（Distribution）：容器、微服务、API 驱动的开发；
- **面向配置设计**（Configuration）：一个镜像，多个环境配置；
- **面向韧性设计**（Resistancy）：故障容忍和自愈；
- **面向弹性设计**（Elasticity）：弹性扩展和对环境变化（负载）做出响应；
- **面向交付设计**（Delivery）：自动拉起，缩短交付时间；
- **面向性能设计**（Performance）：响应式，并发和资源高效利用；
- **面向自动化设计**（Automation）：自动化的 DevOps；
- **面向诊断性设计**（Diagnosability）：集群级别的日志、metric 和追踪；
- **面向安全性设计**（Security）：安全端点、API Gateway、端到端加密；

③ Serverless 架构

按照 CNCF 对 Serverless 计算的定义，Serverless 架构应该是采用 FaaS（函数即服务）和 BaaS（后端服务）服务来解决问题的一种设计。简单理解，Serverless（无服务器架构）是指服务端逻辑由开发者实现，应用运行在无状态的计算容器中，由事件触发，完全被第三方管理，其业务层面的状态则存储在数据库或其他介质中。Serverless 是云原生技术发展的高级阶段，可以使开发者更聚焦在业务逻辑，而减少对基础架构的关注。

AWS 的 Lamada 和阿里云的函数计算都是作为 Serverless 产品对外提供服务。如阿里云函数计算是事件驱动的全托管计算服务。使用函数计算，无需采购与管理服务器等基础设施，只需编写并上传代码。函数计算为您准备好计算资源，弹性地、可靠地运行任务，并提供日志查询、性能监控和报警等功能。

Serverless 架构的优势：无需采购和管理服务器等基础设施，运维成本低；毫秒级别弹性伸缩，快速实现底层扩容以应对峰值压力；按需付费，只需为实际使用的计算资源付费；提高研发效率，技术同学只需专注业务逻辑的开发。Serverless 架构的挑战：延迟：serverless 应用程序是高度分布式、低耦合的，延迟将是一个大问题；迁移成本；成功案例比较少，没有行业标准。

2.3 架构设计原则和理论

2.3.1CAP 定理

CAP 定理：一个分布式系统最多只能同时满足一致性（Consistence）、可用性（Availability）和分区

仅限内部传阅

容错性（Partition tolerance）这三项中的两项。

- **一致性**：所有节点在同一时间的看到的数据相同。
- **可用性**：读、写永远都能成功，即，服务一直可用。
- **分区容错性**：即使系统的某个分区遇到严重的故障，系统能继续提供服务。

根据 CAP 定理，我们无法同时满足一致性、可用性、分区容错性这三个特性。那么，要舍弃哪个呢？对于多数大型互联网应用的场景，主机众多、部署分散，节点故障、网络故障是常态，必须保证 P；应用的目的是提供服务，因此通常也要保证 A。既然要保证 P 和 A，就只能不同程度的舍弃 C，牺牲一些用户体验。严格来讲，部分应用的 A 也不必保证 100%，因此，主流做法是首要保障 P、在 A 和 C 之间取舍、重 A 轻 C。但是，对于金融服务，必须保证 C；大规模金融服务几乎必然涉及网络分区，所以也要保证 P；为了保证 C、P，只能牺牲 A（停止服务）。对于某些特殊的金融服务，需要 7*24 小时提供服务，则改为牺牲部分 P（如单节点主从备份，故障切换），保障 C、A。

2.3.2 BASE 理论

BASE 定理是对 CAP 定理的延伸：即使无法做到强一致性（Strong Consistency），但应用可以采用适合的方式达到最终一致性（Eventual Consitency）。CAP 中提到的一致性是强一致性，所谓“牺牲一致性”指牺牲强一致性保证弱一致性。

BASE 是指基本可用（Basically Available）、软状态（Soft State）、最终一致性（Eventual Consistency）。

- **基本可用**：出现故障的时候，允许损失部分可用性，即，保证核心可用。如电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只提供降级服务
- **软状态**：允许系统存在中间状态，而该中间状态不会影响系统整体可用性。软状态本质上是一种弱一致性，允许的软状态不能违背“基本可用”的要求。如，分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时（某些时刻副本数低于 3）。
- **最终一致性**：系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。软状态的终极目标是最终一致性。如，分布式存储的副本数最终会达到稳定状态。

2.3.3 SOLID

在程序设计领域，SOLID（单一功能、开闭原则、里氏替换、接口隔离以及依赖反转）是由罗伯特·C·马丁在 21 世纪早期引入，指代了面向对象编程和面向对象设计的五个基本原则。当这些原则被一起应用时，它们使得一个程序员开发一个容易进行软件维护和扩展的系统变得更加可能。

- **SRP 单一职责**：该设计原则是基于康威定律的推论，每个软件模块有且只有一个被更改的理由。
- **OCP 开闭原则**：对扩展开放，对修改关闭。
- **LSP 里氏替换原则**：任何基类可以出现的地方，子类一定可以出现。

- **ISP 接口隔离原则**：在设计中需要避免不需要的依赖。
- **DIP 依赖反转原则**：高层策略性代码不应该依赖底层细节的代码，而应该是底层细节代码依赖高层策略。

Martin Fowler 在《架构整洁之道》专门介绍了 SOLID 设计原则。和编程范式相比，设计原则和架构的关系更加紧密，设计原则就是架构设计的指导思想，它指导我们如何将数据和函数组织成类，如何将类链接起来成为组件和程序。反向来说，架构的主要工作就是将软件拆解为组件，设计原则指导我们如何拆解、拆解的粒度、组件间依赖的方向、组件解耦的方式等。

2.3.4 AKF 架构原则

这 15 个架构原则来自《架构即未来 (The Art of Scalability)》一书，作者马丁 L. 阿伯特和迈克尔 T. 费舍尔分别是 eBay 和 PayPal 的前 CTO，他们经历过 eBay 和 PayPal 大规模分布式电商平台的架构演进，在一线实战经验的基础上总结并提炼出 15 条架构原则。

- **N + 1 设计**：永远不要少于两个，通常为三个。比方说无状态的 Web/API 一般部署至少 >=2 个。
- **回滚设计**：确保系统可以回滚到以前发布过的任何版本。可以通过发布系统保留历史版本，或者代码中引入动态开关切换机制 (Feature Switch)。
- **禁用设计**：能够关闭任何发布的功能。新功能隐藏在动态开关机制 (Feature Switch) 后面，可以按需一键打开，如发现问题随时关闭禁用。
- **监控设计**：在设计阶段就必须考虑监控，而不是在实施完毕之后补充。例如在需求阶段就要考虑关键指标监控项，这就是度量驱动开发 (Metrics Driven Development) 的理念。
- **设计多活数据中心**：不要被一个数据中心的解决方案把自己限制住。当然也要考虑成本和公司规模发展阶段。
- **使用成熟的技术**：只用确实好用的技术。商业组织毕竟不是研究机构，技术要落地实用，成熟的技术一般坑都被踩平了，新技术在完全成熟前一般需要踩坑躺坑。
- **异步设计**：能异步尽量用异步，只有当绝对必要或者无法异步时，才使用同步调用。
- **无状态系统**：尽可能无状态，只有当业务确实需要，才使用状态。无状态系统易于扩展，有状态系统不易扩展且状态复杂时更易出错。
- **水平扩展而非垂直升级**：永远不要依赖更大、更快的系统。一般公司成长到一定阶段普遍经历过买更大、更快系统的阶段，即使淘宝当年也买小型机扛流量，后来扛不住才体会这样做不 scalable，所以才有后来的去 IOE 行动。
- **设计时至少要有两步前瞻性**： 在扩展性问题发生前考虑好下一步的行动计划。架构师的价值就体现在这里，架构设计对于流量的增长要有提前量。
- **非核心则购买**：如果不是你最擅长，也提供不了差异化的竞争优势则直接购买。避免 Not Invented Here 症状，避免凡事都要重造轮子，毕竟达成业务目标才是重点。
- **使用商品化硬件**：在大多数情况下，便宜的就是最好的。这点和第 9 点是一致的，通过商品化硬件水平扩展，而不是买更大、更快的系统。

仅限内部传阅

- **小构建、小发布和快试错**：全部研发要小构建，不断迭代，让系统不断成长。这个和微服务理念一致。
- **隔离故障**：实现故障隔离设计，通过断路保护避免故障传播和交叉影响。通过舱壁泳道等机制隔离失败单元 (Failure Unit)，一个单元的失败不至影响其它单元的正常工作。
- **自动化**：设计和构建自动化的过程。如果机器可以做，就不要依赖于人。自动化是 DevOps 的基础。

AKF 的 15 条架构原则之间的关系如图 2-2 所示：

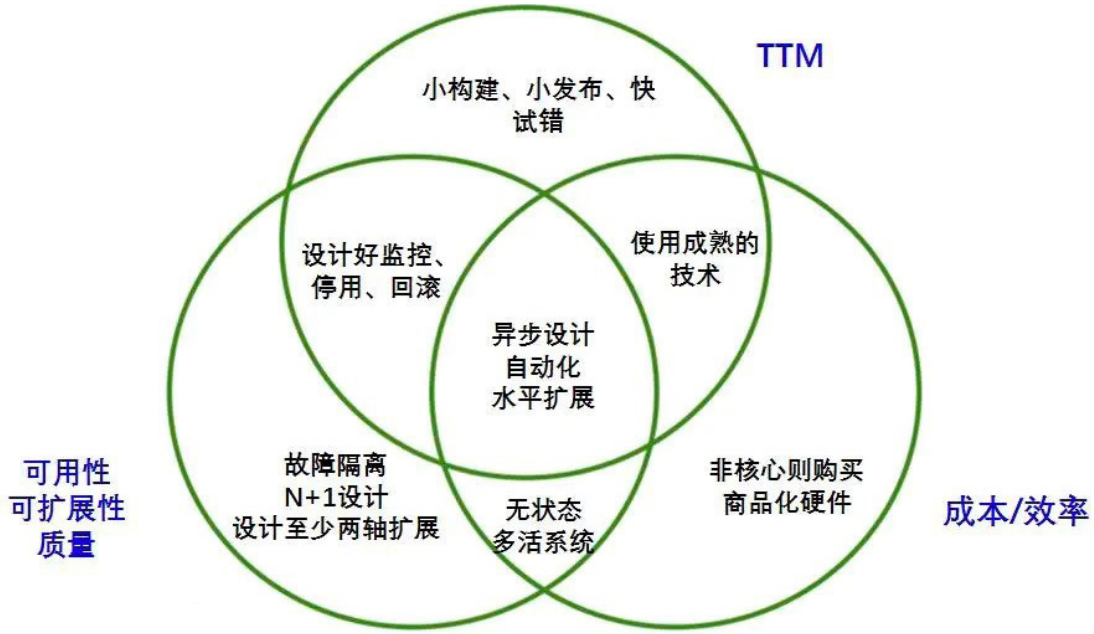


图 2-2 AKF 架构原则

2.3.5 12 要素 (12-factors)

12 要素指的是构建 SaaS 应用的方法论的总结，一共有 12 条。

I. **基准代码**：一份基准代码（Codebase），多份部署（deploy）

II. **依赖**：显式声明依赖关系（dependency）。12-Factor 规则下的应用程序不会隐式依赖系统级的类库。 它一定通过「依赖清单」，确切地声明所有依赖项。此外，在运行过程中通过 依赖隔离 工具来确保程序不会调用系统中存在但清单中未声明的依赖项。这一做法会统一应用到生产和开发环境。

III. **配置**：在环境中存储配置。12-Factor 推荐将应用的配置存储于「环境变量」中。环境变量可以非常方便地在不同的部署间做修改，却不动一行代码；与配置文件不同，不小心把它们签入代码库的概率微乎其微；与一些传统的解决配置问题的机制（比如 Java 的属性配置文件）相比，环境变量与语言和系统无关。

IV. **后端服务**：把后端服务 (backing services) 当作附加资源。12-Factor 应用不会区别对待本地或第三方服务。 对应用程序而言，两种都是附加资源，通过一个 url 或是其他存储在 配置 中的服务定位 / 服务证书来获取数据。12-Factor 应用的任意 部署 ，都应该可以在不进行任何代码

改动的情况下，将本地 MySQL 数据库换成第三方服务（例如 Amazon RDS）。类似的，本地 SMTP 服务应该也可以和第三方 SMTP 服务（例如 Postmark ）互换。上述 2 个例子中，仅需修改配置中的资源地址。

V. **构建，发布，运行**：严格分离构建和运行。12-factor 应用严格区分构建，发布，运行这三个步骤。 举例来说，直接修改处于运行状态的代码是非常不可取的做法，因为这些修改很难再同步回构建步骤。

VI. **进程**：以一个或多个无状态进程运行应用。12-Factor 应用的进程必须无状态且 无共享 。 任何需要持久化的数据都要存储在 后端服务 内，比如数据库

VII. **端口绑定**：通过端口绑定 (Port binding) 来提供服务。12-Factor 应用完全自我加载 而不依赖于任何网络服务器就可以创建一个面向网络的服务。互联网应用 通过端口绑定来提供服务 ，并监听发送至该端口的请求。

VIII. **并发**：通过进程模型进行扩展。在 12-factor 应用中，进程是一等公民。12-Factor 应用的进程主要借鉴于 unix 守护进程模型 。开发人员可以运用这个模型去设计应用架构，将不同的工作分配给不同的 进程类型 。例如，HTTP 请求可以交给 web 进程来处理，而常驻的后台工作则交由 worker 进程负责。

IX. **易处理**：快速启动和优雅终止可最大化健壮性。12-Factor 应用的 进程 是 易处理（disposable）的，意思是说它们可以瞬间开启或停止。 这有利于快速、弹性的伸缩应用，迅速部署变化的 代码 或 配置 ，稳健的部署应用。

X. **开发**：环境与线上环境等价：尽可能的保持开发，预发布，线上环境相同。12-Factor 应用想要做到 持续部署 就必须缩小本地与线上差异。 12-Factor 应用的开发人员应该反对在不同环境间使用不同的后端服务 ，即使适配器已经可以几乎消除使用上的差异。这是因为，不同的后端服务意味着会突然出现的不兼容，从而导致测试、预发布都正常的代码在线上出现问题。这些错误会给持续部署带来阻力。从应用程序的生命周期来看，消除这种阻力需要花费很大的代价。

XI. **日志**：把日志当作事件流。12-factor 应用本身从不考虑存储自己的输出流。 不应该试图去写或者管理日志文件。相反，每一个运行的进程都会直接的标准输出（stdout）事件流。开发环境中，开发人员可以通过这些数据流，实时在终端看到应用的活动。在预发布或线上部署中，每个进程的输出流由运行环境截获，并将其他输出流整理在一起，然后一并发送给一个或多个最终的处理程序，用于查看或是长期存档。这些存档路径对于应用来说不可见也不可配置，而是完全交给程序的运行环境管理。类似 Logplex 和 Fluentd 的开源工具可以达到这个目的。

XII. **管理进程**：后台管理任务当作一次性进程运行。一次性管理进程应该和正常的 常驻进程 使用同样的环境。这些管理进程和任何其他的进程一样使用相同的 代码 和 配置 ，基于某个 发布版本 运行。后台管理代码应该随其他应用程序代码一起发布，从而避免同步问题。

2.3.6 AWS 架构完善的框架原则

① 性能效率

在云中实现性能效率包括五个方面的最佳实践：

仅限内部传阅

- **普及先进技术**：通过将复杂的任务委派给云供应商，降低您的团队实施高级技术的难度。与要求您的 IT 团队学习有关托管和运行新技术的知识相比，考虑将新技术作为服务使用是一种更好的选择。例如，NoSQL 数据库、媒体转码和机器学习都是需要专业知识才能使用的技术。在云中，这些技术会转变为团队可以使用的服务，让团队能够专注于产品开发，而不是资源预置和管理。
- **数分钟内实现全球化部署**：您可以在全球多个 AWS 区域中部署工作负载，从而以最低的成本为客户提供更低的延迟和更好的体验。
- **使用无服务器架构**：借助无服务器架构，您无需运行和维护物理服务器即可执行传统计算活动。例如，无服务器存储服务可以充当静态网站（从而无需再使用 Web 服务器），事件服务则可以实现代码托管。这不仅能够消除管理物理服务器产生的运行负担，还可以借由以云规模运行的托管服务来降低业务成本。
- **提升试验频率**：利用虚拟和可自动化的资源，您可以快速利用各种类型的实例、存储或配置执行对比测试。
- **考虑软硬件协同编程**：了解如何使用云服务，并始终使用最适合您工作负载目标的技术方法。例如，在选择数据库或存储方法时考虑数据访问模式。

② 可靠性

在云中实现可靠性有五个设计原则：

- **自动从故障中恢复**：通过监控工作负载的关键绩效指标 (KPI)，您可以在指标超过阈值时触发自动化功能。这些 KPI 应该是对业务价值（而不是服务运营的技术方面）的一种度量。这包括自动发送故障通知和跟踪故障，以及启动解决或修复故障的自动恢复流程。借助更高级的自动化功能，您可以在故障发生之前预测和修复故障。
- **测试恢复过程**：在本地环境中，经常会通过执行测试来证明工作负载能够在特定场景中正常运作。通常不会利用测试来验证恢复策略。在云中，您可以测试工作负载的故障情况，并验证您的恢复程序。您可以采用自动化方式来模拟不同的故障，也可以重新建立之前导致故障的场景。此方式可以在实际的故障发生以前揭示您可以测试与修复的故障路径，从而降低风险。
- **横向扩展以提高聚合工作负载的可用性**：使用多个小型资源替换一个大型资源，以降低单个故障对整个工作负载的影响。跨多个较小的资源分配请求，以确保它们不共用常见故障点。
- **无需再预估容量**：本地工作负载出现故障的常见原因是资源饱和，即对工作负载的需求超过该工作负载的容量（这通常是拒绝服务攻击的目标）。在云中，您可以监控需求和工作负载利用率，并自动添加或删除资源，以保持最佳水平来满足需求，而不会出现超额预置或预置不足的问题。虽然还有很多限制，但有些配额是可控的，其他配额也可以管理。
- **管理自动化变更**：应利用自动化功能对基础设施进行更改。需要管理的变更包括，对自动化的变更，可对其进行跟踪与审查。

③ 卓越运营

- **执行运营即代码**：在云中，您可以将用于应用程序代码的工程规范应用于整个环境。您可以将

整个工作负载（应用程序、基础设施）定义为代码，并使用该代码进行更新。您可以将运营流程写成代码（脚本），并通过事件触发来自动执行这些脚本。通过以代码形式执行操作，您可以减少人为错误并实现对事件的一致响应。

- **频繁进行可逆的小规模更改**：将工作负载设计为支持组件定期更新。以较小增量进行失败时可逆的更改（尽可能不影响客户）。
- **经常优化运营流程**：在使用运营程序时，要寻找机会改进它们。在改进工作负载的同时，您也要适当改进一下流程。设置定期的实际演练，以检查并验证所有流程是否有效，以及团队是否熟悉这些流程。
- **预测故障**：执行“故障演练”，找出潜在的问题，以便消除和缓解问题。测试您的故障场景，并确认您了解相应影响。测试您的响应流程以确保它们有效，并确保团队能够熟练执行。设置定期的实际演练，以测试工作负载和团队对模拟事件的响应。
- **从所有运营故障中吸取经验教训**：从所有运营事件和故障中吸取的经验教训，推动改进。在多个团队乃至组织范围中分享经验教训。

④ 安全性

在云中实现安全性有七个设计原则：

- **健壮的身份验证体系**：实施最小权限原则，并通过对每一次与 AWS 资源之间的交互进行适当授权来强制执行职责分离。集中进行身份管理，并努力消除对长期静态凭证的依赖。
- **实现可追溯性**：实时监控和审计对环境执行的操作和更改并发送警报。为系统集成日志和指标收集功能，以自动调查并采取措施。
- **在所有层面应用安全措施**：利用多种安全控制措施实现深度防御。应用到所有层面（例如网络边缘、VPC、负载均衡、每个实例和计算服务、操作系统、应用程序和代码）。
- **自动实施安全最佳实践**：借助基于软件的自动化安全机制，您能够以更为快速且更具成本效益的方式实现安全扩展。创建安全架构，包括实施可在版本控制模板中以代码形式定义和管理的控制措施。
- **保护动态数据和静态数据**：将您的数据按敏感程度进行分类，并采用加密、令牌和访问控制等机制（如适用）。
- **限制对数据的访问**：使用相关机制和工具来减少和消除直接访问或人工处理数据的需求。这样可以降低处理敏感数据时数据处理不当、被修改以及人为错误的风险。
- **做好应对安全性事件的准备**：制定符合您组织要求的事件管理和调查策略和流程，做好应对事件的准备工作。开展事件响应模拟演练并使用具有自动化功能的工具来提高检测、调查和恢复的速度。

⑤ 成本优化

在云中实现成本优化有五个设计原则：

- **践行云财务管理**：为获得财务上的成功并加速在云中实现业务价值，需要投资云财务管理 / 成

仅限内部传阅

本优化。您的组织需要投入时间和资源增强自身在这个新的技术和使用情况管理领域中的能力。与安全性或卓越运营能力类似，您的组织需要通过知识构建、计划、资源和流程来培养能力，从而成为一家具有成本效益的组织。

- **采用消费模型**：仅为所需计算资源付费，并可根据业务需求而非复杂的预测增加或减少使用量。例如，开发和测试环境通常只需要在每个工作日运行八个小时。您可以在不需要时停用这些资源，从而实现 75% 的潜在成本节约（40 小时对比 168 小时）。
- **衡量整体效率**：衡量工作负载的业务产出及这些产出的实现成本。使用这种衡量方式了解您通过提高产出和降低成本获得的收益。
- **不再将资金投入到无差别的繁重任务上**：AWS 会负责繁重的数据中心运营任务，例如安装、堆叠和驱动服务器。它还消除了使用托管服务管理操作系统和应用程序的运营负担。因此，您可以集中精力处理客户和业务项目而非 IT 基础设施。
- **对支出进行分析和归因**：使用云，您可以更轻松地确定系统的准确使用量和成本，从而将 IT 成本透明地分摊到各个工作负载拥有者。这有助于衡量投资回报率 (ROI)，并让工作负载拥有者能够据此优化资源和降低成本。

2.3.7 Azure 应用程序的十个设计原则

遵循这些设计原则可以提高应用程序的可伸缩性、复原能力和易管理性。

- **自我修复设计。**在分布式系统中，故障时有发生。设计应用程序以在故障发生时进行自我修复。
- **实现全面冗余。**在应用程序中构建冗余，以避免出现单一故障点。
- **尽量减少协调。**最大程度地减少应用程序服务之间的协调以实现可伸缩性。
- **横向扩展设计。**设计应用程序，使其能够扩大，根据需要添加或删除新实例。
- **通过分区解决限制。**使用分区来解决数据库、网络和计算限制。
- **运营设计。**合理设计应用程序，使运营团队获得所需的工具。
- **使用托管服务。**尽量使用平台即服务 (PaaS) 而不是基础结构即服务 (IaaS)。
- **使用最佳的数据存储完成作业。**选择最适合数据的存储技术，并了解如何使用该技术。
- **演变设计。**所有成功的应用程序会不断变化。 进化型设计是持续创新的关键。
- **根据业务需求构建。**每个设计决策必须与业务要求相称。

2.3.8 云原生模式 Cloud Native Pattern

- **应用设计**
  - 微服务化，通过微服务可以更好的解耦，提高研发效率，提高弹性和容灾能力。而实现微服务的模式可以是 Request/Response 这种同步式的模式，也可以是事件驱动这种异步式的模式，各有优劣，看情况选择。
  - 应用程序无状态化，无状态可以保持简单，避免很多问题，无状态化也无自动弹性提高了可能

- Infrastructure as Code，应将应用程序配置视为源代码：在源代码存储库中进行管理，版本控制和访问控制。

- **冗余服务**
  - 提高弹性和可用性
  - 多实例部署
  - 负载均衡，服务端可以可以通过 Nginx，客户端可以通过 Istio 等技术实现
  - 服务发现，通过域名服务将客户端和服务端松耦合，可以通过 Zookeeper 等经技术实现，也可以托管给 Kubernetes
  - bulkheads 模式：业务分片、用户分片、数据库拆分
- **容错设计**
  - 考虑重试，也靠避免重试风暴，以及控制循环
  - 通过断路器模式进行熔断，保护系统过载
- **可观测性设计**
  - 使用 API 网关或者 Sidecar 代理流量的出入和管理，也可以统一处理可观察性信息，日志，指标和跟踪数据的收集
  - 需要将日志从运行时主动推动到中心化存储中，并且需要对日志做聚合等操作确保可观察性，分布式跟踪技术至关重要
- **数据存储设计**
  - 通过统一的分布式后端存储可以解决数据的隔离问题，提供更好的提供弹性，如 ectd
  - 可以通过 Pub/Sub 模式传递数据，可以使用事件日志作为唯一事实来源
- **持续交付**
  - 滚动升级、蓝绿发布、并行部署

2.4 全局架构的制图与外化

2.4.1 为什么需要架构图

一般而言，架构图和文档应该主要用于团队内部和团队之间的协作、沟通、愿景和指导。它们还应该包含项目的重要设计决策（在某个特定时刻采取）。架构图应该帮助每个人看到大局，了解周围的环境，这应该是创建和维护架构图背后的根本原因。所谓“一图胜千言”。常见的架构图有产品 & 业务架构、系统分层架构、系统链路架构、技术架构、部署架构、数据架构等等，不同的架构图有不同的表现方式。

2.4.2 关于架构制图的思维

很多时候一线开发同学会觉得架构图难以起笔，不知道怎么画才能准确的表达，或者画到一半看着似是而非的架构图就搁置了，造成这种情况的很大一部分原因没有掌握到位架构制图的正确思维，比如换位思维，层次思维，取舍思维，接下来会进行详细阐述。

仅限内部传阅

① 换位思维

画图前，一定要想清楚你画的架构图的目标受众有哪些，不同的目标受众有不同的关注层次。不同的目标受众会有不同的关注点：

- **CTO 或管理者**：他们可能不太会关注非常细的细节，而是从整体上看架构的合理性、边界的清晰性、以及架构的可持续性、可维护性、投入成本的 ROI 等问题。
- **研发**：一般会关注很多实现相关细节，比如技术选型、实现可行性、可维护性等，毕竟他们是架构的最直接消费者。
- **运维**：不太关心应用内的具体技术实现（当成黑盒），但很关心各个应用实例的物理部署方式、网络连通性、可运维性等。
- **安全**：只关注系统是否有安全风险，例如是否可能被注入恶意代码、是否有权限漏洞等；如果经历过安全评审，应该很有体感。
- **产品**：大部分情况下只关心项目能否按期上线，其他方面 ... 可能表面上表示些许关心，实际上要么并不在乎要么真的不懂。

② 层次思维

软件架构发展到当前阶段，不管是整个电商领域，还是电商领域下的某个资源，其技术架构的复杂度也是超乎想象的。因此怎么去解构一个复杂系统呢？最好的办法就是层次化。层次思维的掌握可以从以下 2 个方面来训练：

- **分而治之**：软件领域中，分而治之是控制和应对复杂系统的最有效方法。而层次化拆分，本质上就是一种分而治之手段：将系统按照从粗到细的粒度，一级一级地拆分成多个相对独立和低耦合的元素（子系统、应用、组件等）。
- **金字塔原理**：这本书的核心观点就是，按照自顶向下的方式，先抛出主观点再依次用各个子观点去论证。这样的沟通方式更符合人类的思维逻辑，也更容易让读者接受。简单来说，就是要“先说重点”，帮助读者做归纳总结和划重点，而不是先抛出一大堆细枝末节的零散东西让读者自己去消化和推演。

③ 取舍思维

中国古人说鱼和熊掌不可兼得的理论，其实也在阐释这样一个朴素的道理。而很多人的烦恼往往就是来自于不懂得取舍，什么都想要，什么都不肯放。架构图也是一样，一个图不可能承载方方面面的诉求，因此要想明白重点是什么，非重点是什么，进行取舍。突出一个或几个重点来表现，一些非重点的可以通过其他的图来补充。所有东西都放在一个图里，反而会造成信息爆炸，画的时候复杂度居高，哪怕画出来了，看到人也是云里雾里。

2.4.3 架构制图的方法框架

① UML 建模

UML 是统一建模语言的简称，它是一种由一整套图表组成的标准化建模语言。UML 用于帮助系统开发

人员阐明，展示，构建和记录软件系统的产出。UML 代表了一系列在大型而复杂系统建模中被证明是成功的做法，是开发面向对象软件和软件开发过程中非常重要的一部分。UML 主要使用图形符号来表示软件项目的设计，使用 UML 可以帮助项目团队沟通、探索潜在的设计和验证软件的架构设计。

以下简要地介绍了这 13 个 UML 图表。UML 图表可大致分为结构性图表和行为性图表两种。

结构性图表显示了系统在不同抽象层次和实现层次上的静态结构以及它们之间的相互关系。结构性图表中的元素表示系统中具意义的概念，可能包括抽象的和现实的概念。结构性图表有七种类型：

- **类图** (Class Diagram)：类图是一切面向对象方法的核心建模工具。该图描述了系统中对象的类型以及它们之间存在的各种静态关系。
- **组件图** (Component Diagram)：在统一建模语言中，组件图描绘了组件如何连接在一起以形成更大的组件或软件系统。
- **部署图** (Deployment Diagram)：部署图有助于模拟面向对象软件系统的物理方面。它是一个结构性图表，显示了软件产出于系统架构内如何被分发至指定目标。
- **对象图** (Object Diagram)：对象图是实例 (Instance) 的表达，包括对象和数据值。静态的对象图是类图的一个实例，使用是相当有限的，它常被用作展示数据结构例子。
- **包图** (Package Diagram)：包图是 UML 一种用以显示包和包之间的依赖关系的结构性图表。模型图能显示系统的不同视图，例如多层应用程序。
- **组合结构图** (Composite Structure Diagram)：UML 2.0 中的新的图表之一。它是一种静态结构图，显示了一个类的内部结构和这个结构所实现的协作。
- **轮廓图** (Profile Diagram)：轮廓图能够创建特定于域和平台的原型，并定义它们之间的关系。

行为性图表显示了系统中对象的动态行为 ，可用以表达系统随时间的变化。行为性图表有七种类型：

- **用例图** (Use Case Diagram)：用例模型从用例的角度描述系统的功能需求，它是系统预期功能（用例）及其环境（参与者）的模型。
- **活动图** (Activity Diagram)：活动图用于展示工作流程，描述了目标系统的控制流程，比如探索复杂的业务规则和操作，描述用例和业务流程。
- **状态机图** (State Machine Diagram)：状态图描绘允许的状态和转换以及影响这些转换的事件，它有助于可视化对象的整个生命周期，从而更好地理解以状态主导 (State-based) 的系统。
- **序列图** (Sequence Diagram)：根据时间序列展示对象如何进行协作。它展示了在用例的特定场景中，对象如何与其他对象交互。
- **通讯图** (Communication Diagram)：与序列图类似，通讯图也用于模拟用例的动态行为。与序列图相比，通讯图更侧重于显示对象的协作而不是时间顺序。
- **交互概述图** (Interaction Overview Diagram)：交互概述图侧重于交互控制流程的概述，它是活动图的变体。
- **时序图** (Timing Diagram)：时序图显示了既定时间内对象的行为。时序图是序列图的一种特殊

仅限内部传阅

形式，它俩之间的差异是轴反转。

② 4+1 视角架构模型

1995 年，Philippe Kruchten 在《IEEE Software》上发表了题为《The 4+1 View Model of Architecture》的论文，引起了业界的极大关注。4+1 视角架构模型一文提出了一种用来描述软件系统体系架构的模型，如图 2-3 所示，这种模型是基于使用者的多个不同视角出发。这种多视角能够解决多个 " 利益相关者 " 关心的问题，利益相关者包括：最终用户、开发人员、系统工程师、项目经理等，能够分别处理功能性和非功能性需求。五个视角中每个都是使用符号进行描述。这些视角都是使用以架构为中心场景驱动和迭代开发等方式实现设计的。

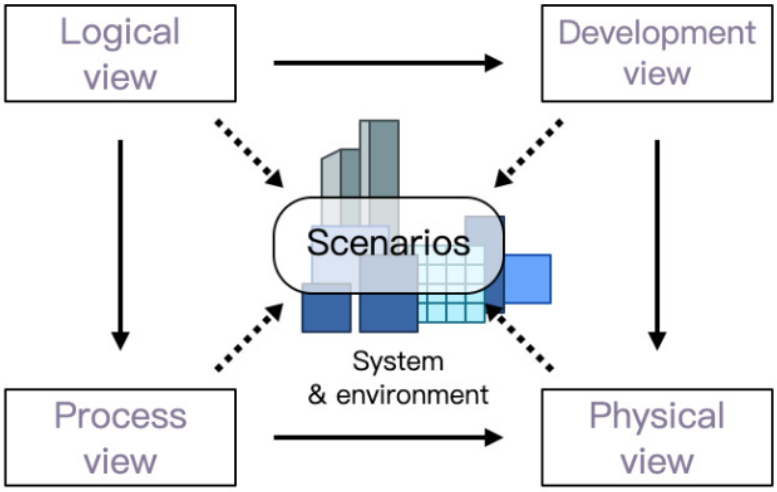


图 2-3 4+1 视图架构模型

- **逻辑视图**（Logical view）：描述系统为终端用户提供的功能，一般会通过 UML 中的类图和状态图来表示。
- **过程视图**（Process view）：描述系统的动态行为，包括流程和交互等，一般会通过 UML 中的时序图、活动图和通讯图来表示。
- **开发视图**（Development view）：从程序员的视角来阐述系统，也被称为“实现视图”，一般会通过 UML 中的组件图和包图来表示。
- **物理视图**（Physical view）：从系统工程师的角度来描述系统，包括系统组件的物理拓扑、各组件之间的物理连接，也被称为“部署视图”，一般会通过 UML 中的部署图来表示。
- **场景**（Scenarios）：通过一小组用例或场景来描述架构，包括系统中各种对象和进程之间的交互时序，也被称为“用例视图”。这些场景会被用于识别架构元素（architectural elements）以及阐述和验证整个架构设计，也可以被作为架构原型的测试起点。

③ C4 模型

C4 模型是来自《Software Architecture for Developers》一书的定义，指的是 context 上下文场景，container 容器，component 组件和 class 类，意思指一个软件架构是由这些模型呈树形结构组成。

关注代码仍然是大多数软件开发生命周期中关注的焦点，这是有一定道理，因为代码是最终交付。但如果你不得不向别人解释关于系统是如何工作的，你会从代码开始解释吗？确实代码并不能讲述系统的整个故事，在缺乏文档的情况下，人们通常会开始在白板上或纸上用图框和线条解释系统的主要构建块是什么，它们是如何连接的。而使用 Microsoft Visio, 等专业工具又比较复杂。更好的办法是在不同的抽象层次创建不同的图 diagram，一个简单的图 diagram 比复杂的图能够更有效表达描述。

C4 代表上下文（Context）、容器（Container）、组件（Component）和代码（Code）——一系列分层的图表，可以用这些图表来描述不同缩放级别的软件架构，每种图表都适用于不同的受众。

- **第 1 层**：系统上下文。它显示了你正在构建的软件系统，以及系统与用户及其他软件系统之间的关系。
- **第 2 层**：容器。将软件系统放大，显示组成该软件系统的容器（应用程序、数据存储、微服务等），技术决策也是该图的关键部分。
- **第 3 层**：组件。将单个容器放大，以显示其中的组件。这些组件映射到代码库中的真实抽象（例如一组代码）。
- **第 4 层**：代码。如果你确实想要，或者说有这个必要，可以放大个别组件，以显示该组件的实现方式。

2.4.4 架构制图的工具

正所谓“工欲善其事，必先利其器”，掌握正确的工具画图，可以事半功倍。一般我们所知的 Microsoft Visio、Rational Rose 这类软件，往往只适用于 windows 系统，对 mac/linux 的用户不太友好，接下来介绍 2 款比较好用的画图工具。

① Draw.io

Draw.io 是一款免费的在线绘图工具 (也有客户端)，可以绘制流程图、结构图、组织架构图、UML 等。

- Draw.io 内置大量丰富的模板，可以直接选取后再加工；
- Draw.io 支持多种格式导出，包括 png、SVG、PDF、HTML 等等；
- 上手比较简单，画出来的图也是比较简洁，画风舒适。

② OminiGraffle

OmniGraffle 是由 The Omni Group 制作的一款绘图软件，其只能于运行在 Mac OS X 和 iPad 平台之上。它曾获得 2002 年的苹果设计奖。OmniGraffle 可以用来绘制图表，流程图，组织结构图以及插图，也可以用来来组织头脑中思考的信息，组织头脑风暴的结果，绘制心智图，作为样式管理器，或设计网页或 PDF 文档的原型。优点是功能强大，缺点是需要付费。

仅限内部传阅

【第二篇】
基础主题

三、全球化部署

全球化业务有多种业务形态，全球化部署架构的模式、架构标准、机房和网络分布、全球数据同步，这些都是全球化部署架构需要考虑的核心问题，本章我们来看看全球化部署的相关话题。

3.1 全球化部署架构演进

业务的部署方案是由业务的形态，用户体验，成本，数据和合规等综合来决定的。总体上，可以分为跨境电商模式和全球化电商模式两大类。

3.1.1 跨境电商模式

以国际化业务 AE（AliExpress，速卖通）为例，AE 业务的场景是跨境电商，将国内的商品售卖给俄罗斯，美国，巴西，欧洲等国家。所以业务的特点就是商品发布是国内，卖家信息都是一个站点发布。但是买家确实多个国家。所以有几个方案可以考虑。

本地部署架构

卖家是国内的，整个电商系统也部署在国内，在每个国家设置 pop 接入点，然后访问国内的系统。如图 3-1 所示。

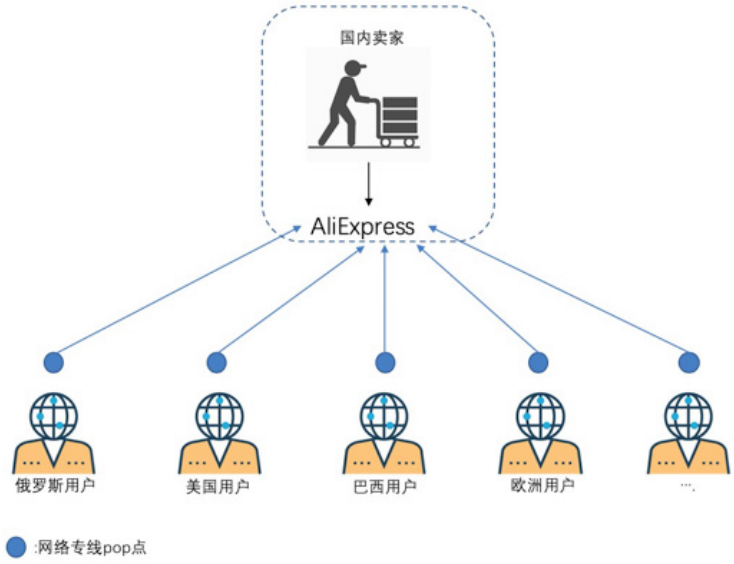


图 3-1 本地部署示意图

本地方案优势

- **部署架构简单**：一套系统服务多个国家；卖家数据库只有一份，不用考虑数据同步和一致性的

仅限内部传阅

问题；买家信息也只有一份，不用考虑区域化的问题，比如俄罗斯的用户去了美国，那么该用户在美国也可以登录系统，购买服务，查看历史订单信息等。

- **部署成本**：部署成本最低，电商系统只需要部署一套，数据库也只需要维护一套。资源集中也方便进行运维成本的优化和控制。

本地方案劣势

- 买家是通过跨地区调用到国内，即便有网络 pop 点加速，但是专线延迟对比直接访问本地系统还是比较高
- 受数据合规严重，很容易受到数据法规政策导致用户不能访问，业务不能开展等问题。

全球化跨境部署

因为新加坡是中立国，对于数据合规的政策增加宽松，比如美国用户数据是不可以落在国内，但是可以落在新加坡。卖家用户群体对比买家数量少，新加坡到国内的专线延迟对比直接访问国内稍高，并且对于网络延迟接受程度是比买家要大的。买家在 top 国家部署本地站点，减少用户跨地区访问带来的专线网络耗时。买家信息也是落在本地站点。因为电商存在库存等中心化的服务，各个站点间是需要进行网络数据同步。如图 3-2 所示。

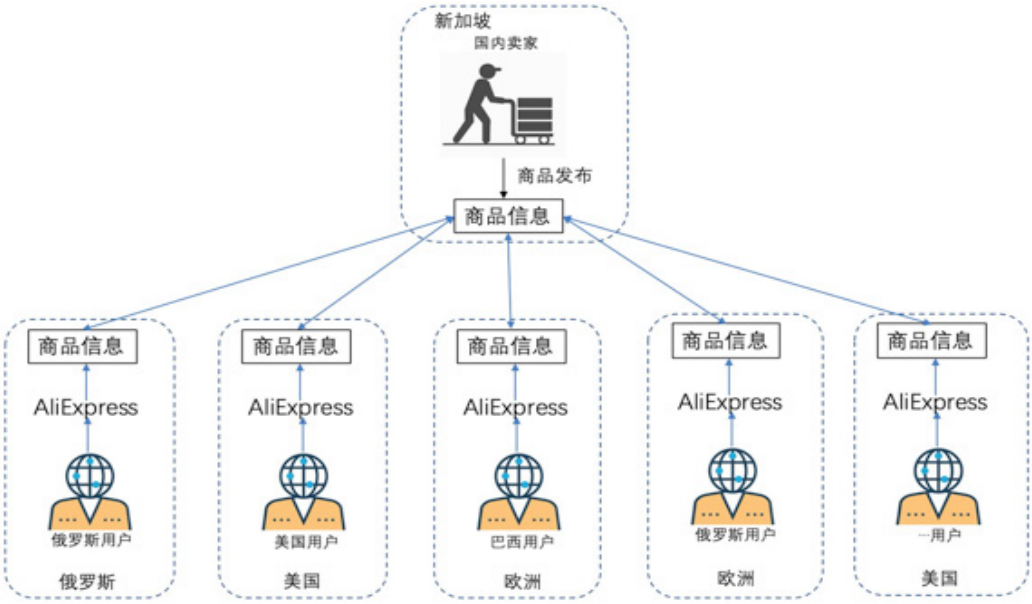


图 3-2 全球化部署示意图

全球化部署方案优势：

- 本地部署，更接近用户，用户体验更好。
- 数据合规的风险更低，避免因为数据合规导致的买家不可访问或者业务不能开展的问题。

全球化部署方案的劣势：

- 卖家跨地区进行商品发布等一系列商家操作，但是卖家数量对比买家少，并且对网络延迟的容忍程度更强，该影响可忽略。

- 部署方案复杂：需要考虑用户区域化访问的问题，比如俄罗斯用户去到了美国，那么因为数据合规，该用户的数据只在俄罗斯本地有，美国没有，那么在美国访问站点，服务就会受影响（更多细节，可以参考“区域化”主题）。部署成本更高，每个国家都需要部署一套电商系统，并且商品库等也需要完整的本地部署一套。需要考虑数据同步的问题，还有数据一致性的问题。

在建站初期，数据合规并没那么严格，业务最开始采用的部署方案是商品发布在国内，买家系统部署在美国，随着业务发展，开始在核心国家进行本地站的建设来提升用户体验，再随着数据合规越来越严格，将卖家迁移到了新家坡进行部署。

3.1.2 全球化电商模式

商家和买家都是本地用户，以阿里巴巴东南亚头部电商 Lazada 为例。Lazada 服务东南亚马来西亚，新加坡，泰国，越南，印尼，菲律宾 6 个国家。为了更好服务好这 6 个国家，遵循当地用户的习惯，风俗等，所以业务最开始的时候我们为每个国家都搭建了一套 Lazada 的电商系统，并且 6 个国家的卖家和买家都是独立的。

全球化独立部署架构

新加坡是一个中立国，在业务开始初期，结合成本和用户体验的综合，将服务新加坡，菲律宾，马来西亚，泰国，越南站点都部署在了新加坡，然后每个站点通过中间件实现了单元隔离（不同的站点通过不同单元标的中间件和 configserver 进行隔离，来保证数据的一致性）。印尼因为本国强硬的法律政策，数据只能落本地，所以为印尼用户单独部署了一套 Lazada 的电商系统，如图 3-3 所示（更多细节，可以参考“全球容灾”主题）。

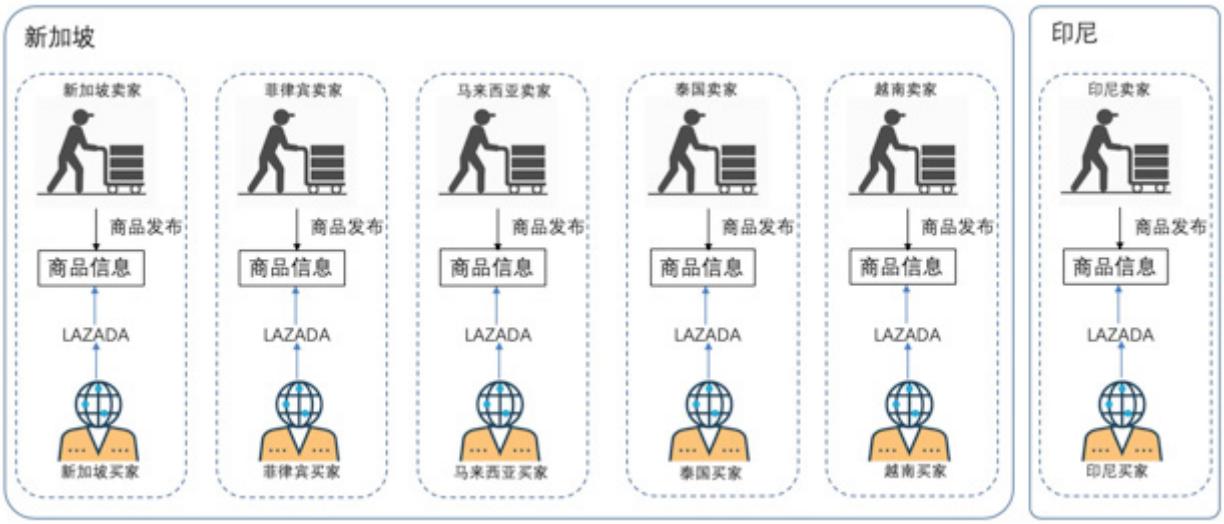


图 3-3 全球化独立部署示意图

全球化独立部署方案优势：

- 可以更贴合用户，不同的站点，独立的系统，独立配置，根据不同地区不同用户进行不同的运营

仅限内部传阅

全球化独立部署方案劣势：

- **部署成本高**：新加坡，菲律宾，马来西亚，泰国，越南 6 个国家都是部署在新加坡，电商系统都是完全一样的功能，部署了 5 套。
- **维护成本高**：因为东南亚 6 个国家，6 套相同的系统，开发代码发布需要 6 个站点都进行发布，而且因为站点太多，环境太多，学习成本和维护成高。

全球化中台部署架构

将 6 套独立的站点，整合成 2 套站点（印尼因为数据合规要求，只能部署印尼本地）。通过国际化中台，大大的降低了业务的部署成本和运维成本，如图 3-4 所示（更多细节，可以参考“多租户”主题）。

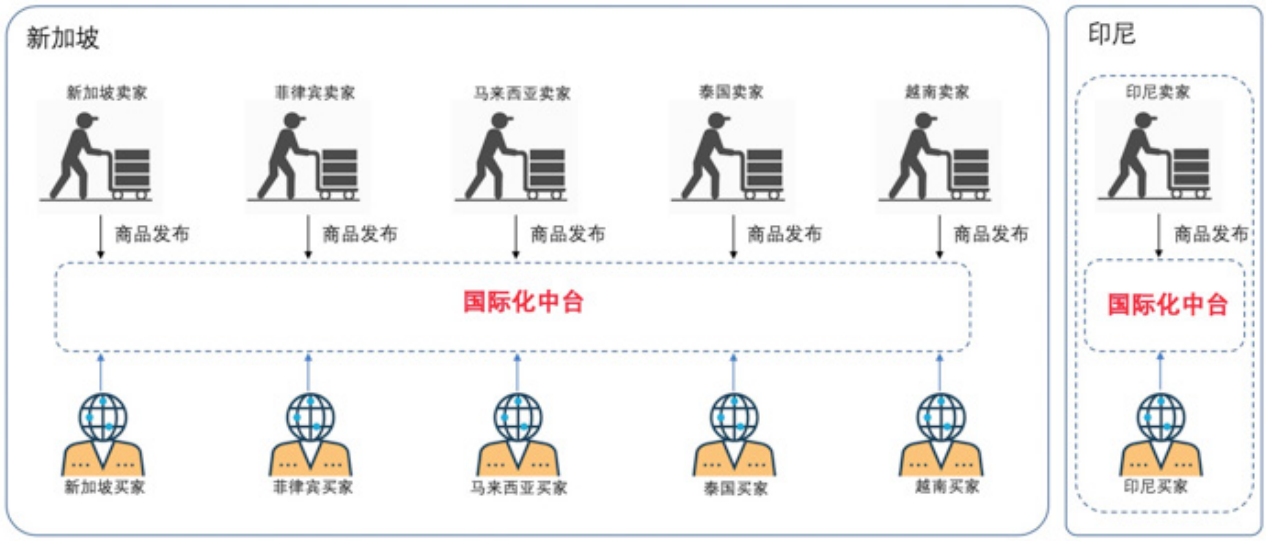


图 3-4 全球化中台部署示意图

全球化中台部署方案优势：

- **继续支持可以更贴合用户**，不同的站点，独立的系统，独立配置，根据不同地区不同用户进行不同的运营。
- **资源成本低**：将之前的 5 套或者更多套系统通过多租户的方案缩减成 1 套系统。大大降低了资源成本。
- **维护成本降低**：一套代码发 2 套系统（印尼因为数据合规，只能单独部署），大大降低了维护成本。
- **复用成本低**：现在 Lazada 业务只服务东南亚 6 国，当有新的站点，比如我们当时做淘宝台湾，淘宝香港站，Daraz 业务也是基于现在的新家坡的这套系统来支持的。实际上现在新加坡的这一套国际化中台业务支撑了 4 个业务 13 个国家

全球化中台部署方案的劣势：

- **对团队的要求比较高**，比如中台组织关系，中台与业务的边界，研发效能的管理。

总体来看，部署架构是一个变化的过程，是和业务的形态，用户体验，数据合规，成本等密切挂钩的，

一个好的架构是逐渐演变过来的。

3.2 全球化部署架构标准

全球化部署架构也需要一个好的标准和规范，根据标准来约束系统，做到更好的统一，下面我们来看下相关的部署架构标准

3.2.1 基础设施的标准

- **容器资源的标准**：我们将容器使用的资源规格根据业务场景和需求做了汇总，然后制定了几个标准个规格。比如 4 核 8G，虽然部分业务可能使用 4 核的资源有一些浪费，这些我们再通过技术手段，比如 cpu share 的技术或者资源超卖，资源抢占的形式来解决。资源的标准确定了，也更加方便我们的服务器选型和资源的分配，避免零碎资源的存在。
- **底层宿主机的标准**: 通过容器资源的标准来确定宿主机的标准，比如业务容器内存和 cpu 比是 2:1，那么在宿主机的选型中，我们就会选择内存：cpu 为 3：1 或者 4：1 的。因为内存是不可超卖资源，所以内存越大，对于我们可提升的空间也就越大。
- **网络的标准**：整个机房的网络布局和架构，包括机架位的选择，都是按照一定的标准来执行，做到更大的适配性。

3.2.2 应用层的标准

- **无状态化**：应用必须无状态化，做到可迁移，可扩容和可缩容。
- **镜像的标准**：因为国际化覆盖了十几个国家，并且一些应用也需要做一些特定的配置。所以我们也将底层的镜像做了标准。大家都使用统一的基础镜像，然后如果有需要特殊配置，在代码中配置相关的 dockerfile 内容。
- **应用名命名标准**：我们对应用的类别和功能，业务域等制定了相应的规范。如图 3-5 所示：

图 3-5 应用名命名标准

比如 global-trade-center-s 的应用名中，global 代表的是国际化中台的业务，trade 标识交易应用，center 是标识交易的核心，s 代表的是该应用是一个 service 应用，如果是已 -w 结尾则表示是一个 web 应用等。

- **应用结构标准**：

仅限内部传阅

缓存数据库基本都在区域机房内做到闭环。

```
                |--------脚本目录(bin)          //必须
                |--------程序目录(target)       //必须
                |--------配置目录(conf)         //必须
应用主目录一    |--------日志目录(logs)         //必须
                |--------数据目录(data)         //可选
                |--------库目录(lib)            //可选
                |--------临时目录(tmp)          //可选
                |.....                          //可选, 其它目录
```

图 3-6 应用结构目录标准

- **启动脚本的标准**：/home/admin/${appname}/bin/appctl.sh {start|stop|restart}
- **其他标准**：包括不限于 jvm 标准，端口标准，war 包的规范，启动时长，健康检查的规范，时区的规范，db 的规范等

3.3 全球化部署与分布

我们再来看看全球化当前的整体机房分布的特点，从图 3-7 中可以看出，全球化机房部署主要有如下特点：

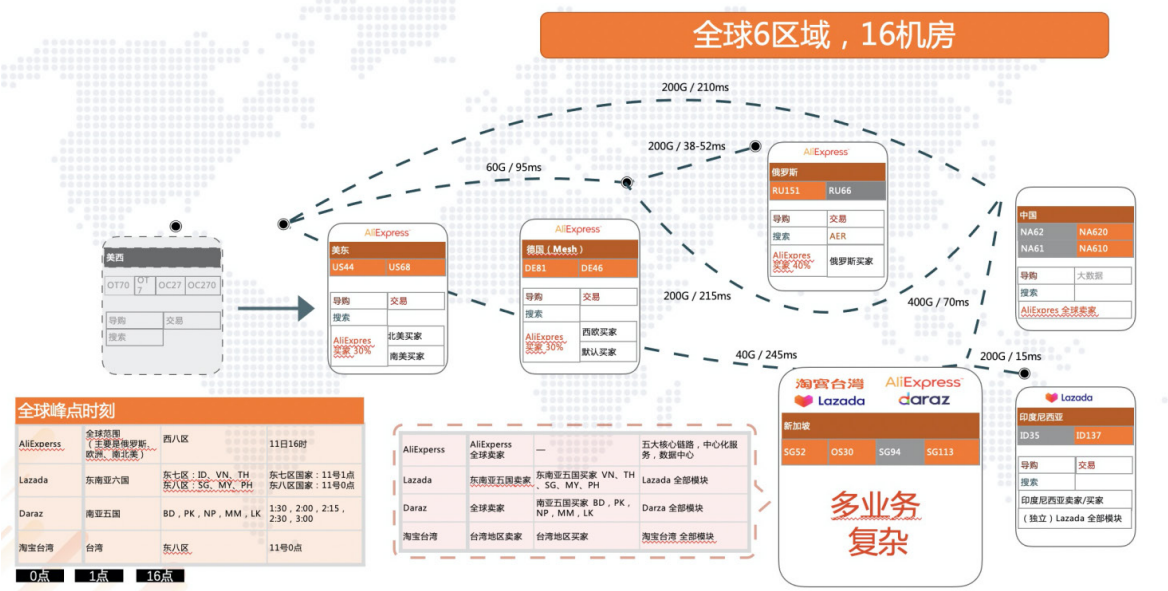


图 3-7 全球化机房分布

多区域，多机房，多业务

全球 6 大区域，物理机房 16 个，逻辑机房 20 个，机房众多分散， 覆盖多个时区。支持业务涵盖 AliExpress, Lazada, Daraz, 淘宝香港站。

云机房与非云机房并存，区域机房业务闭环

每个区域的国际物理机房规模很小，基础设施能力很不平均，再加上部分机房还隔离多个逻辑机房，混合云机房以及非云机房。区域内机房做到业务闭环，涉及的中台产品比如大数据，搜索推荐，容器，

业务多单元混合分布

服务于东南亚 10 国的 lazada 业务，都在新加坡两个机房支撑；再加上支撑国际业务的其他 BU，最终应用众多，存在一个逻辑机房存在 13 单元混合部署情况，相应的基础架构比较复杂，如图 3-8 所示。

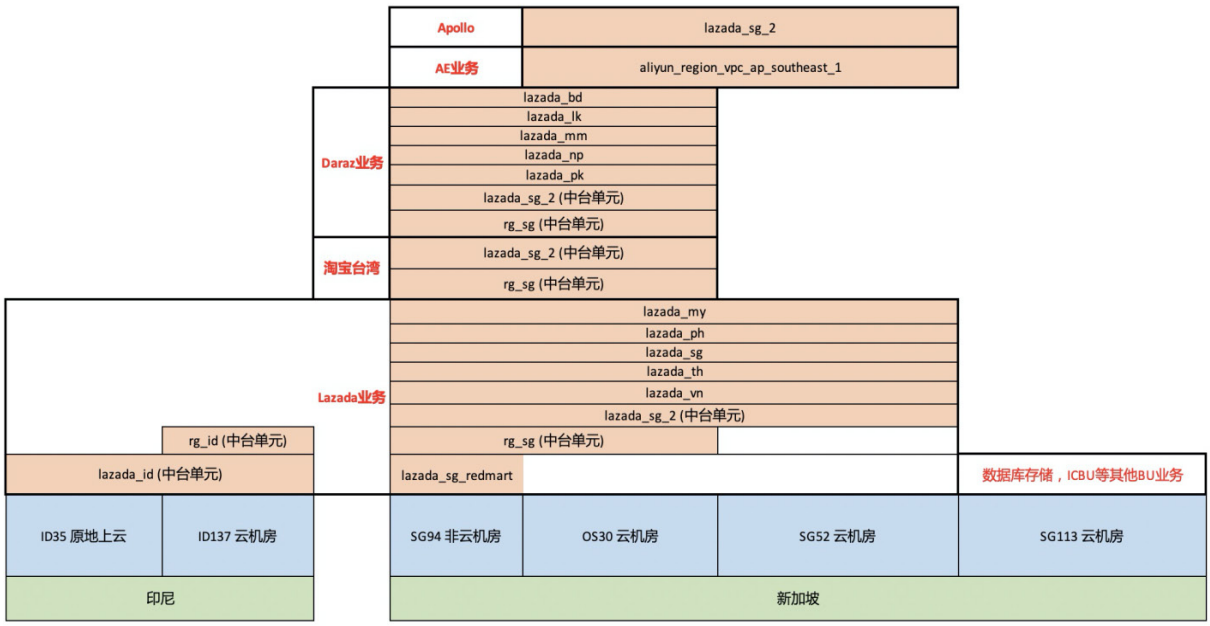


图 3-8 业务多单元混合部署分布

骨干网络铺设全球

从网络分布角度，在全球各大洲重点区域都有核心骨干网络节点，如图 3-9 所示；以 AE 业务覆盖范围，两个区域间延时仅在百毫秒级别，比如张北中北到美东最长链路，延时也就在 177ms 左右。

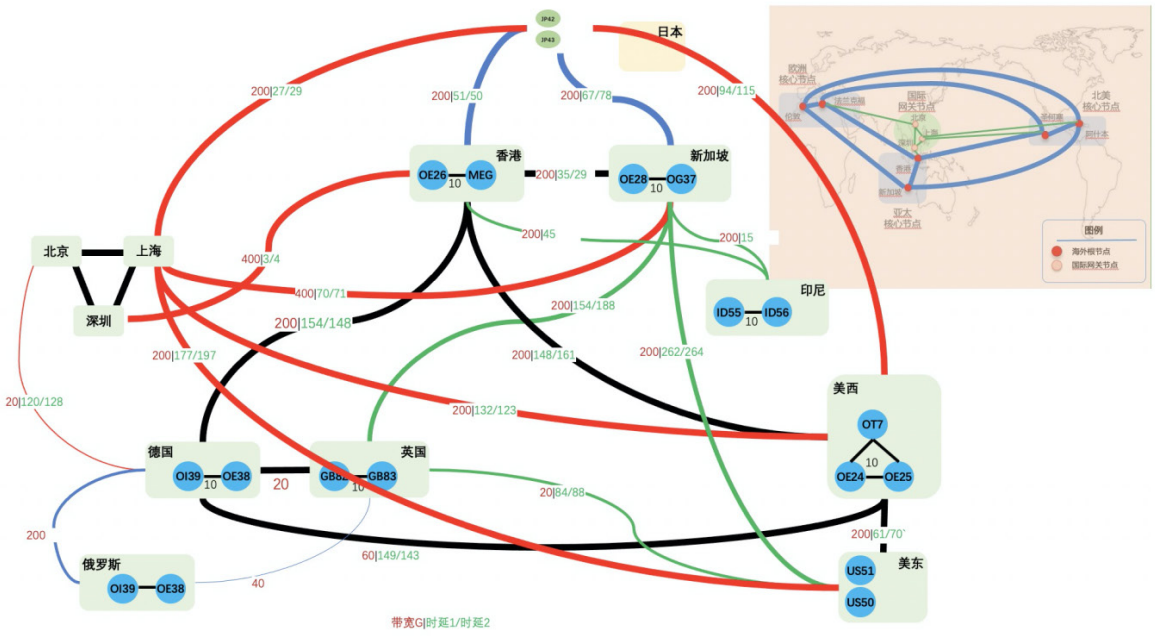


图 3-9 全球化骨干网络分布

3.4 全球数据同步

而在国际化场景中，因为涉及到跨单元、跨区域甚至跨洲的数据传输和使用，所以在数据的处理和使

仅限内部传阅

用上尤其重要，数据主要分为两部分，一个是数据的使用，还有一个是数据的同步，针对单一区域的数据使用这里就不赘述，主要都是偏业务性质的，这里主要讨论下数据同步。在说数据同步前，先简单介绍下数据变更，因为数据同步其实也就是对数据变更的同步。

3.4.1 数据变更

数据变更是最基本的操作，涉及到数据库的 DML，DDL 等。如果是单库操作，那只需要按照正常的变更流程进行就可以了，但在国际化场景中，因为每一次的数据变更可能会涉及多个单元的变更，所以需要特别小心，这些变更会同步到其他单元库中。

一般我们操作变更是指对在线数据库表以及数据的变更，这些变更可以分为两大类，一类是 DDL，也就是针对表结构等元数据的定义变更，例如 CREATE，ALTER 等操作，另一类是 DML，即针对实际数据的操作变更，例如 INSERT，UPDATE 等。针对这两类变更，在多单元情况下的操作，我们总结了一些规范用作参考。

DDL 变更规范

- **CREATE 操作**：必须确认各单元不存在需要创建的表。原因：因历史原因之前存在过这个表，但因某些误 DROP 操作，导致部分单元库丢失这表，部分单元库存在这表。若是中心库丢失该表且操作者只 check 了中心库，而其他单元存在这个表，则会导致 DDL drc 失败。
- **ALTER 操作**：必须确认各单元表均存在，且 ALTER 对象完全同义。原因：若某单元库该表丢失，则导致 DDL drc 失败。若 alter 对象不同义，则可能导致某些未知风险。
- **DROP 操作**：必须确认各单元该表均存在，且数据量 =0。原因：若某单元库该表丢置，则导致 DDL drc 失败；若数据量不为 0，则 DROP 操作无法执行。

DML 变更规范

- DML 操作主要考虑是否涉及大批量数据操作，大型批量数据量 DML 必须考虑分页或限流；无法进行分页或限流，必须避开在线业务的高峰时间。

一些变更建议

- DDL 需要在中心库操作，因为 DRC 无法同步单元到中心的 DDL 操作。
- 涉及到数据的变更需要双人 Check。

3.4.2 数据同步

正常系统中数据库都是主从备份，这种同步对于开发来说基本不感知，且已经被很多数据库支持。但在国际化场景中，因为涉及跨单元、跨区域的数据同步，这里我们主要讨论在线 DB 的数据同步。

数据同步可以分为两种，一种是同单元内的数据同步，这个可以通过精卫来实现，其原理也是通过对 binlog 做解析并同步到目标库中，另一种是跨单元的数据同步，采用的是 DRC 系统，DRC 系统是阿里自主研发的对异构数据源进行实时迁移同步的数据管道基础技术设施。DRC 同步模块主要由 DRC Strore 和 DRC Congo 组成，DRC Store 在原端进行数据的抓取、存储和发送服务，DRC Congo 在目标端顺序读取 Store 中的增量记录，并在保证记录更新时序和事务一致性的前提下，将记录高效并发同步到目的端数据库。

基础的 DRC 同步系统可以按照库或者表这一级别来过滤同步数据，同步过程中即可以开通单向的数据同步，也可以基于双向的数据同步，基于双向的数据同步场景下，这意味着同一张表的同一条记录，假设多地都有可以修改，可能会存在同步冲突。区域化能保证多地永远不会同时修改同一个字段，即便是双向的数据同步，也不会产生数据冲突。

但是基础的 DRC 系统还无法实现有选择的只同步表中的某些记录，因为这需要同步系统实现更加精细化的过滤逻辑，既让我们的同步系统不光要支持库表级别过滤，还需要支持记录级别的过滤，另外还要支持在现有过滤规则中嵌入业务相关的过滤逻辑。详细的同步场景可以参考“区域化部署”中的数据层跨区域路由部分。

仅限内部传阅

四、全球稳定性

4.1 全球稳定性背景

我们先看看几个与全球稳定性相关的故障：

- 『运行态故障』内存泄露导致 OOM-killer，监控没覆盖，切流恢复过程慢
- 『运行态故障』基础设施联动性差，容灾架构极限情况未覆盖，无法容灾切流恢复
- 『编码问题导致资损』一行代码设置错误的价格，导致 E3 资损，核对和监控没有覆盖
- 『运行态故障』预发和线上缓存没有隔离，导致 P1 故障
- 『接口协同导致资损』上下游交互 API 的 Result 不一致，导致资损，长时间没有发现
- 『压测场景遗漏，大促故障』压测缺失平台优惠，导致双 11PH 站点 0 点峰值成功率下跌
- 『数据库内存，大促故障』数据库内存出现雪崩效应，交易库出问题导致成功率下跌
- 『运行态，全局架构』边缘场景增加配置，引发全局单元化规则调用出现问题
- 『Diamond 配置错误』配置大小写问题，解析没有错误保护
- 『Diamond 配置错误』配置少了一个中括号，配置解析错误，用户无法支付
- 『容量问题』线上容量以及限流不足，在流量增加的情况下，无法提供服务
- 『配置错误，预发修改，错误的操作了线上』配置是有问题的，误操作到线上，且没有灰度
- 『全集群 FullGC』整个集群运行态 FullGC，是一个大灾难，且不好恢复
- 『海外网络』海外网络的稳定性差，短期也没有太好的方案
- 『海外网络』印尼的基础设施很薄弱，地震、海啸、断电、网络问题频发
- 『限流保护』应用限流和霸下防护，是保护自己的核心手段，需要时长巡检
- 『配置没有隔离，也没有监控』CEO 周六看数据报表，发现的一个下跌故障
- 『幂等唯一键更改』一行代码，导致的大资损，且没法追回
- 『高危低频变更要慎重』JingWei 控制台误操作，同时应用缺乏限流能力和分组隔离

可以看出，故障是不分时间发生的，每一次故障都是一个黑天鹅，全球稳定性面临了非常多不确定的黑天鹅，我们如何防止黑天鹅发生，或是当黑天鹅发生时系统能够做到反脆弱性，是全球稳定性一直在追求的目标。全球稳定性，也经常叫高可用，通常来描述一个系统经过专门的设计，从而减少停工时间，而保持其服务的高度可用性；关注平均无故障时长，尽量减少停服时间，让服务全年不停顿，最终保持系统服务的高度可用性。

对于国际化场景的特性，稳定性面临着如下挑战：全球底层基础设施复杂，任何一个地方出问题都会导致出现问题，面临高度复杂的环境，随时都可能出现故障，故障发生时不能快速发现，故障发生后

仅限内部传阅

不能快速定位根因，故障发生后恢复时间不可控。

基于此，我们全球化稳定性的目标如下：围绕客户价值驱动，提升用户稳定性体验来展开，通过基础能力建设：告警能力、预警有效性、故障监控能力、变更大脑能力建设、智能定位、区域容灾、单元容灾、DC 容灾、机房 / 单元自愈能力、快恢预案、限流；对以上能力建设的数据度量；通过演练对能力建设的验证来达到最终目标。

为了达到这个目标，我们用如下方法思想：从三个维度来快速将问题进行恢复：通过监控快速发现，通过快速定位系统快速定位问题发生的位置，通过多种方式进行快速恢复，如逃逸、回滚、熔断等方式。这个过程中，我们通过建设、度量、突袭三个方式对上述方法论进行闭环的持续建设：

- **建设**：按照容灾架构体系不停建设；
- **度量**：通过数据化进行建设的数据度量，从数据层面进行展开；
- **突袭**：通过突袭方式进行建设验证，确保建设的是符合预期的。

其实稳定性体系有一套比较通用的完整体系，这里暂时不做完整介绍，这里更聚焦一些国际化这面更有特色的如下方面进行展开：

- **日常稳定性**：如容灾快恢体系、灰度体系、监控体系、限流体系等；
- **大促稳定性**：如大促标准化、链路梳理、压测模型、预热等。

4.2 全球稳定性体系

4.2.1 容灾快恢

目标方向

针对 1-5-10 的发现、定位、恢复三个维度的能力确定最终的国际化容灾快恢方向：

- **针对 1 的快速发现**，从告警有效性、预警有效性、故障监控三个维度进行突破并进行地毯式治理，并最终将三者进行有效的提升；1 的目标是准确，只有准确了才能快速发现，如果不准确，那就会导致狼来了的情况，最终影响开发判断，产生不必要的噪音；
- **针对 5 的能力**，从变更大脑、智能定位两个维度进行快速的定位能力建设，确保有问题时能够判断出问题出现的位置，从而可以进行快速恢复；5 的目标是实时，只有实时定位出来才能对后续的 10 进行快速决策；
- **针对 10 的能力**，从容灾（区域、单元、DC）、快恢预案、自愈、限流，以及针对能力的建设进行容灾演练进行保鲜；10 的目标是保鲜，都说能力建设好了，是否真正 work 还是需要演练来进行最后的验证；

如图 4-1 所示，基础数据对应 1 分钟发现，要求的是准确，需要从告警有效性、预警有效性、故障监控等方面进行突破；诊断对应 5 分钟定位，要求能够实时发现，从变更大脑、智能定位等方面进行突破，

仅限内部传阅

确保诊断的实时性；恢复对应 10 分钟快恢，要求能够快速自动恢复且具备保鲜性，目前从区域容灾、单元容灾、DC 容灾、有损强切、机房 / 单元自愈、异常 VM 熔断、单机自愈、快恢预案等多方面进行建设，确保快恢能力能够快速恢复对应故障；

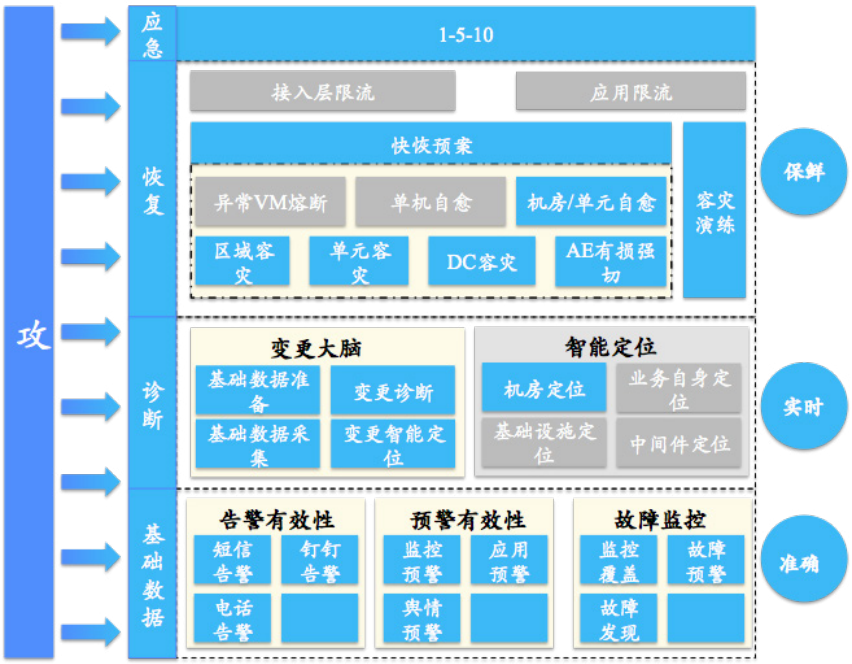


图 4-1 容灾快恢 1-5-10 目标

容灾成果展示

通过前期的建设，结合监控、预警、分机房监控等措施，故障发生前将预警推送，并满足可以通过切流进行容灾时，会在钉钉中推送出对应预案，如图 4-2 所示，并且直接执行。并由此养成了有问题线切流的习惯。

图 4-2 容灾快恢预警

未来的方向：建设、度量、验证三维闭环

容灾快恢当前有了基本的能力，还需要长时间进行验证。面向未来，需要进一步基于三维度的闭环进行展开，确保建设的能力确实有效的，如图 4-3 所示。

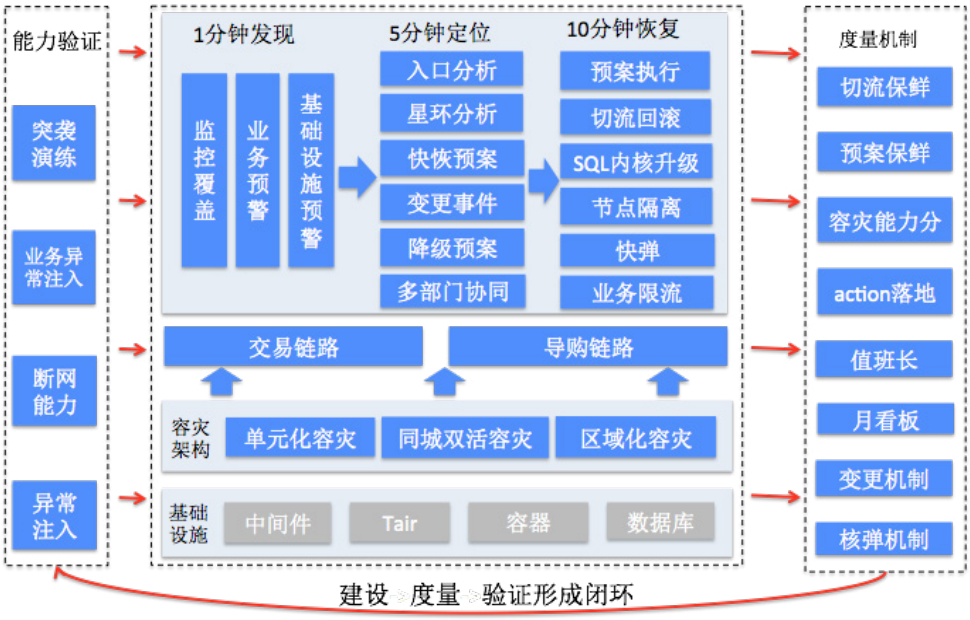


图 4-3 容灾快恢未来方向

4.2.2 灰度发布

背景

灰度提供了一种快速低成本试错的机制，在持续部署的不同的阶段有多种实现形式，有时也叫金丝雀发布。灰度的核心是将修改推广到一小部分用户，验证没有问题后，再推广到全部用户，以降低生产环境引入新功能带来的风险。

谈到灰度，这里再解释一下蓝绿部署，是通过运行两个相同的生产环境来减少停机时间和降低风险，与灰度不同，蓝绿部署本质是非黑即白的部署方式。同时我们在效果验证过程中，经常也会提到 AB 测试，本质上是一种测试效果验证的手段，同时运行多个不同版本的服务，这些服务更多的是用户侧的体验不同，比如页面布局、按钮颜色，交互方式等，通常底层业务逻辑还是一样的。

整体架构

一次业务上线变更完整生命周期包含 CI 和 CD 过程，灰度是处在 CD (continuous delivery) 中的一个阶段，必须要经过灰度之后，才可以在正式环境进行全量发布变更，如图 4-4 所示。

仅限内部传阅

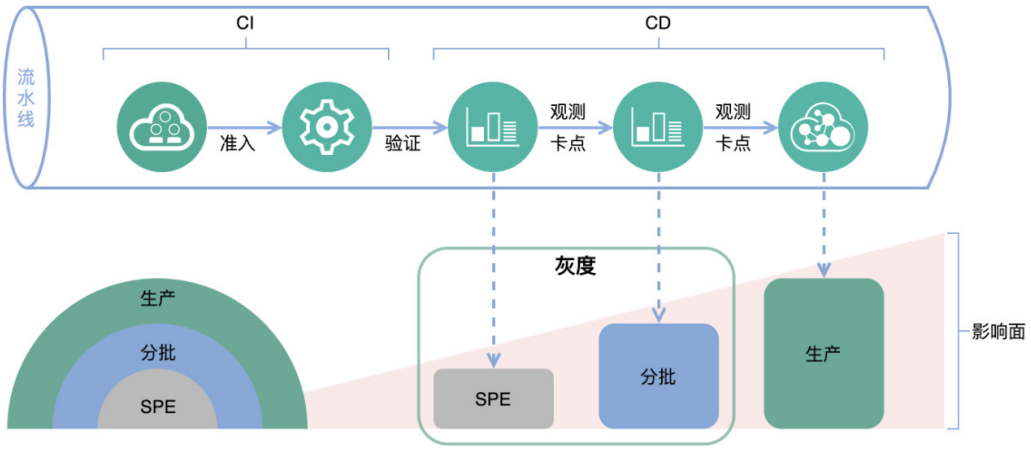


图 4-4 灰度发布示意图

灰度发布我们包含两个阶段：SPE(“安全生产环境”)和分批发布，若当前 BU 或变更系统已被 SPE(“安全生产环境”)所覆盖，则须先在 SPE(“安全生产环境”)内进行发布，此环境针对发布批次无要求。只有在 SPE 验证无问题后，才能在线上生产环境进行灰度发布。

灰度的原则

线上生产环境灰度变更包含以下要求：可分批、可控制分批间隔、可观测 / 可验证、可暂停 / 可回滚。

- **可分批**：目前的分批方式分为三个层面，每个层面中对应的灰度方式如下图所示。此外，图中的簇指的是可以继续拆分的逻辑组，包含但不限于单元、Region、机房、Availability Zone、VPC、Cluster、分组、自定义逻辑区域等。灰度方式必须至少满足灰度分批方式中的一项：簇内分批、簇间串行、簇间打散。确定好灰度方式后，至少需要 2 批进行发布，如图 4-5 所示。

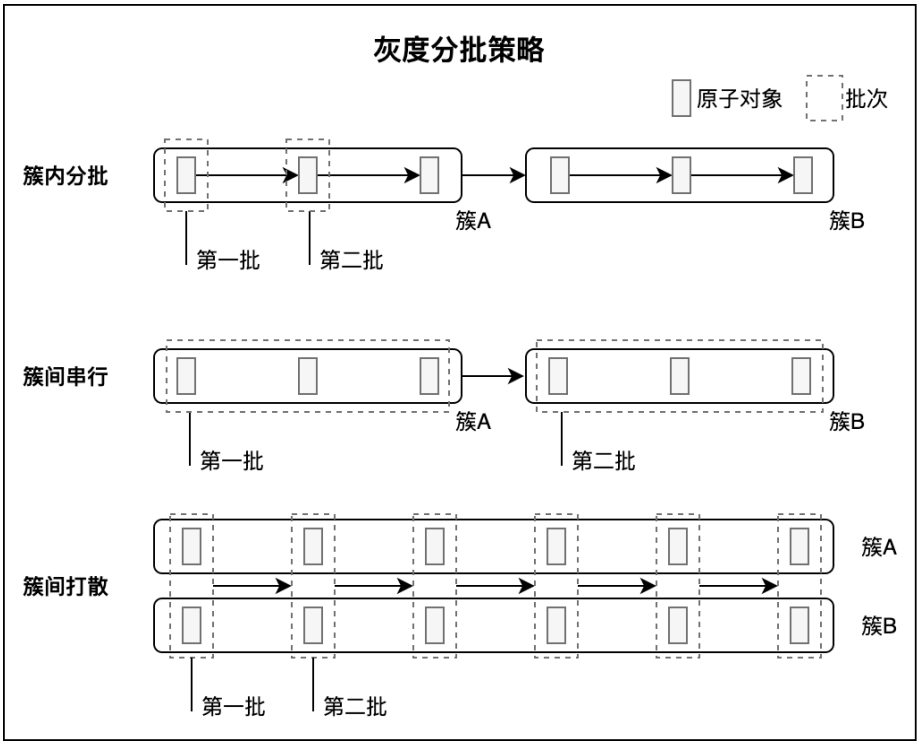


图 4-5 灰度可分批原则

- **可控制间隔**：变更系统必须在系统层面对每批次的发布时间间隔进行控制，不接受文案提示的方式。具体来说有如下几类：**重大风险的变更灰度时长至少一个小时。**集团核心系统的生产环境灰度发布总观测时长不能少于 60 分钟。此外，第一批变更完后至少观测 20 分钟，后面几批发布间隔可自定。BU 核心系统的生产环境灰度发布总观测时长不能少于 30 分钟。此外，第一批变更完后至少观测 20 分钟，后面几批发布间隔可自定。
- **可观测**：变更系统每批次发完后，需要观测并验证本批次发布无问题后才能进行下一批次的发布。观测和验证的手段包括但不限于以下方式：在变更系统里至少记录一项核心反应健康状态的指标（业务监控项、日志文件名等）或记录 double check 人员或采用自动化观测等，并能通过对线上文件验证等方式确定已发布成功。
- **可回滚**：灰度时需具备分批回滚、全量回滚的能力，回滚单要有变更记录并可追溯。

国际的安全生产环境

安全生产环境 (SPE) 通过流量控制和隔离技术实现影响面的控制，目的是隔离生产流量，减少风险影响，在环境内形成调用闭环，方便测试的进行，SPE 环境灰度的时间点位一定要在上线生产之前。SPE 引流集团内网全部流量和线上 1% 流量作为测试覆盖的基础。图 4-6 展示了为 SPE 架构图。

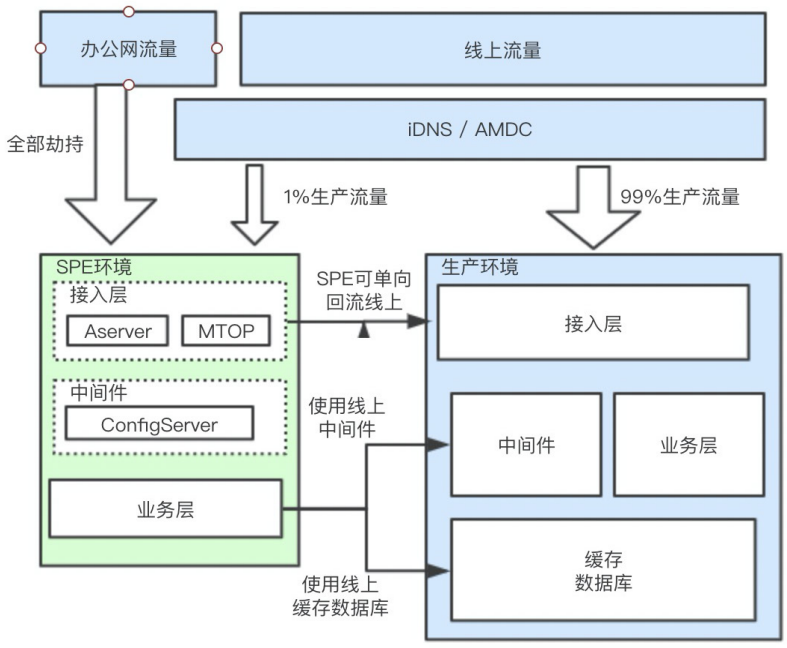


图 4-6 SPE 安全生产环境架构图

- **灰度流量调度**：基于区域化流量调度能力，通过银河平台整体切流。比如下图展示的根据不通业务及不通的流量比例，5% 给到 SPE 环境，95% 给到线上生产环境。
- **流量调度原理**：通过 aserver、软负载的方式进行流量调度。接入层 aserver 基于 vipserver key 获取服务器列表，在 SPE 环境中，aserver 优先使用带业务 SPE 分组机器（spe 环境机器

仅限内部传阅

默认带小流量标可被识别）。如果有 spe 环境分组机器，就转发进入 spe 环境并往下流转。如果不存在 spe 环境分组机器，默认使用生产机器并转入线上生产环境。软负载 configserver 对 SPE 服务有调度优先级，通过服务 consumer 的注册环境标判断 SPE 服务请求，优先分配 SPE 的服务 provider，达到 SPE 环境内服务调度闭环。

- **整体请求流程**：通过 DNS 导流生产流量进入 SPE 专属的 aserver 集群；灰度流量进入 aserer 后，mtop 依据 HSF 的调度优先级进入对应业务网关服务；业务链路在 spe 环境有部署则形成闭环（服务必须注册在 spe 所属 configserver）；链路必须在 SPE 闭环，否则无法形成业务链路验证。

4.2.3 监控体系

背景

在 Google SRE 书中对监控的定义为：收集、处理、汇总，并且显示关于某个系统的实时量化数据，例如请求的数量和类型，错误的数量和类型，以及处理用时，应用服务器的存活时间等。监控一个系统有多个原因：

- **分析长期趋势**：如业务量是否上涨，系统指标是否和之前版本比较有优化或者恶化。
- **w 临时性回溯分析**：如临时发现系统 load 过高，需要分析近期系统使用清康。
- **告警**：监控告警可以让系统在出现故障的时候及时发出通知，或者在系统处于恶化趋势时提前进行预警。
- 除此以外，对于整体系统，可以针对业务特点，进行合理的监控大盘设计，可以让例如测试、横向 SRE 可以清晰了解当前业务系统是否异常。

监控体系框架

一个合理的监控体系，可以让不同角色在不同视角下关注自己应该关心的内容，如 PE 关心物理或者虚拟机资源的情况，业务 SRE 关注业务自己的指标，以及业务自己所强 / 弱依赖组件的各项指标，来进行合理的预警。全局的横向 SRE 则可以通过核心业务的指标大盘等，及时发现当前业务系统是否受损，即时牵头组织针对故障或者风险进行应急。图 4-7 展示了整体的监控体系。

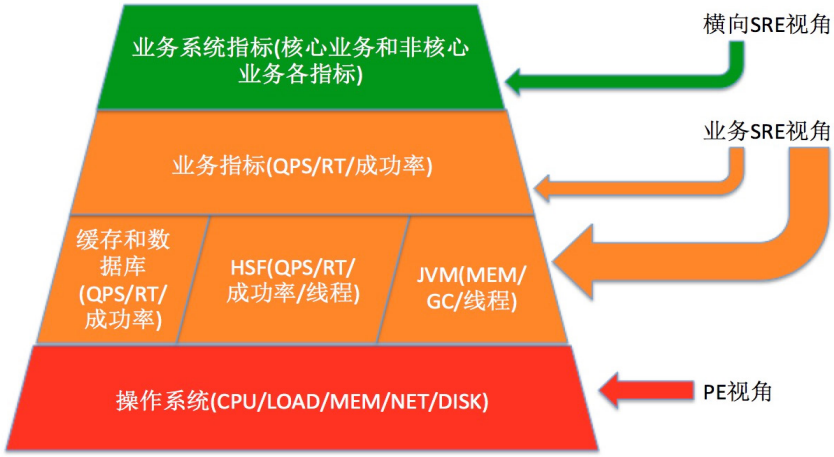


图 4-7 监控体系框架

五大监控原则

- **原则一**：合理的服务监控至少应该包含三个核心指标，QPS、成功率、RT。Google SRE 中定义的四个核心指标为延迟、流量、错误、饱和度，一个合理的服务监控指标，至少应该包含对应 QPS、成功率、RT。
- **原则二**：合理的系统监控应该包含容器基础指标和相关中间件的核心指标。系统最基础指标包括 cpu、load、内存、流量、网络丢包 / 重传率、磁盘 IO 等。基于集团技术栈上，还需要包括一些其他指标，如 jvm 的 gc(包括 young gc 和 full gc) 次数和时间、java 线程数、java 堆内存使用情况、HSF 的成功率、RT、QPS。如果涉及缓存和数据库，则还需要关注如 tair 读写 QPS、RT、成功率、缓存命中率，数据库连接池、读写 QPS、RT、成功率。
- **原则三**：合理设计告警阈值，提高监控系统信噪比。监控系统信噪比应该很高，发出紧急警报的组件需要非常简单而且可靠。产生警报的监控系统规则应该非常容易理解，同时代表一个清晰的故障场景。特别是对于国际化，不同国家所承载的业务量不同，必须细致设计各个站点对应的告警阈值，同时告警规则内容必须清晰，否则会导致无效告警频繁触发，开发人员疲于应对无效告警问题的排查。
- **原则四**：简化监控告警，能反映真实故障的规则应该越简单越好。那些最能反映真实故障的规则应该越简单越好，可预测性强，非常可靠。
- **原则五**：一个优质的监控系统需要长期维护。监控系统需要随着业务系统进行长期维护，当软件重构、业务变化、负载特征发生变化，监控系统也需要相应的修正。

典型案例

案例 1: Lazada 侧媒体大屏

以压测大屏为例，对于交易链路，核心场景为订单创建和支付收单，因此在大屏上核心部分展示了订单创建总量 / 分站点 QPS / 分站点 QPS 趋势 / 成功率 / 错误码及支付收单 QPS / 支付收单成功率。

图 4-8 Lazada 媒体监控大屏

仅限内部传阅

其他核心场景，如商品详情页，购物车，优惠，履约按照分站点展示 QPS。同时展示核心应用的水位计算指标。这样，我们就可以在如图 4-8 上，看到核心链路的业务指标，对应的错误信息，以及其他场景的访问量和基础的应用水位指标。

案例 2: 支付域分站点告警降噪

国际这里一共支持 Lazada ／ Daraz 十一个国家的业务，每个国家的业务请求量不同，而且支付域是整个交易链路比较底层，流量整体比例会更低。早期的支付域预警中，并没有针对各个国家进行区分配置，特别是晚上小流量国家的抖动非常容易触发风险预警。

针对支付域的痛点进行整改，将 11 个国家分为大中小流量型国家，针对故障等级定义中的标准，针对三种类型的国家指标进行区分，例如 (仅为示例，非实际配置，实际还有持续优化)：大流量国家跌幅 10% 并同比持续 2 分钟即告警；中流量国家跌幅 20% 并同比持续 3 分钟即告警；小流量国家跌幅 30% 并同步持续 5 分钟即告警。优化后，支付域的风险预警准确率提升，除大促后的同比异常外，准确率已经高于 90%，误报率低于 10%。

4.2.4 流量限流

背景

在应对如下场景时：秒杀场景，如 Lazada 之前零点 APP 的商品详情自动刷新；大促瞬时高并发场景；超过预期流量，如上游击穿缓存，导致过量请求访问下游服务；上云后的服务器超卖，导致系统负载过高，提供服务异常；针对核心请求和非核心请求进行区分，防止非核心请求过量访问阻断基础业务运行。我们需要依靠限流来对我们自己的系统提供保护，以便更好的保护业务系统的整体稳定运行，如图 4-9 所示。

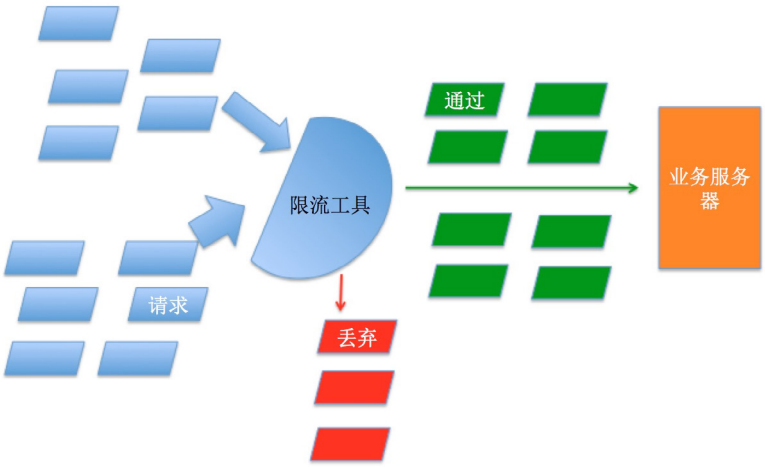


图 4-9 流量限流示意图

限流体系架构

限流体系目前一共有两层：接入层限流（霸下）、应用层限流（Sentinel）以下针对这两种限流展开介绍。

① 霸下限流

霸下是集团通过接入层限流的安全防护工具，其功能不单纯是限流，也包含了安全防护的功能。这里主要介绍限流相关内容。霸下的网关层有三种主要的接入方式：在统一接入层进行霸下配置限流，无需业务改造，大部分业务都是采用这种模式；应用前置一个 Tengine 并配置霸下限流，一般用于没有接入网关，或者是核心的重要应用会额外增加一层；霸下的 Java SDK 接入，业务需要做适量修改。当客户端发起请求后，霸下网关层会判断是否触达限流，没有触达就转发给业务服务器进行处理，同时还会异步转发流量到霸下处置中心，这里会进行一些安全的判断。其架构图如图 4-10 所示。

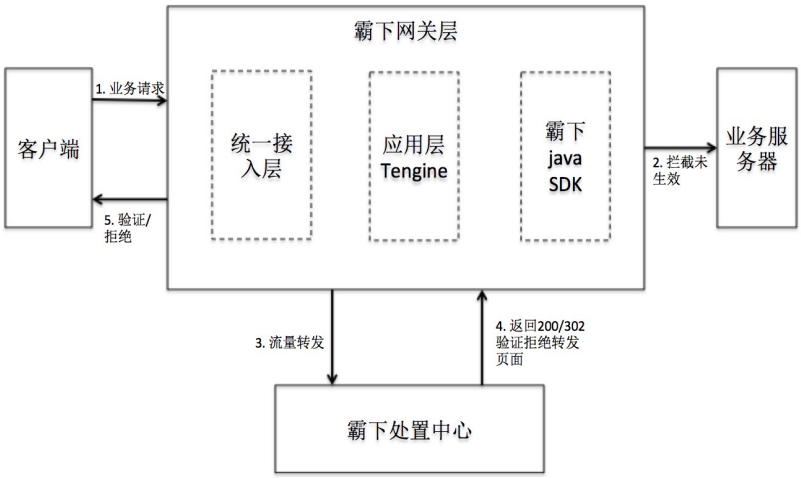


图 4-10 霸下架构示意

② Sentinel 限流

Sentinel 是从集团内诞生，并向外部开源的限流工具。Sentinel 主要由控制台和 Sentinel 的客户端两大部分，控制台可以进行规则配置，并推送到规则中心并下发到 Sentinel 客户端上，客户端根据相关规则控制限流逻辑，可以对上游应用进行入口限流降级，或者对下游进行限流降级，并生成相关流控数据，通过接口发送到控制台上进行运行态监控，也会写入本地日志，可以被其他监控和数据系统采集。Sentinel 针对资源的限流，国际化中台有集群和单机两种限流方式方式，单机限流由本地的 Sentinel 客户端根据限流值对入口的请求进行限制。其简单架构如图 4-11 所示。

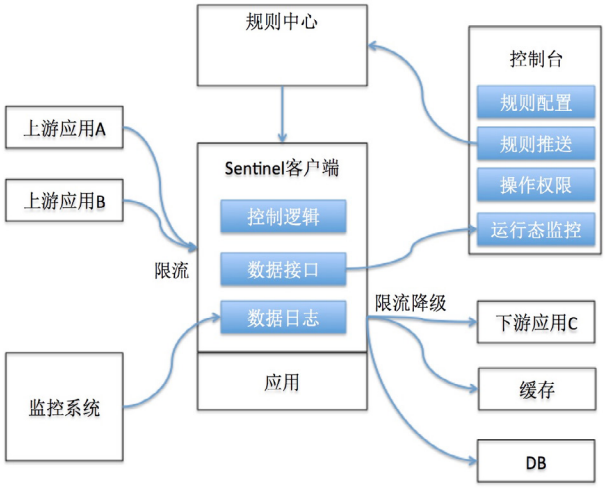


图 4-11 Sentinel 限流架构图

仅限内部传阅

集团一般在统一接入层 aserver 之后再到业务应用，在 aserver 中可以使用霸下进行限流，防止过量请求在 java 的网络 io 层就将业务应用打爆，在业务侧可以使用 sentinel 进行限流。业务入口限流建议按照预估值的 120% 进行配置，非业务入口的按照系统容量支持的 80% 进行限流保护。霸下配置的限流建议为 sentinel 配置的限流值的 150%-200%，如果霸下仅支持单机限流，那么当限流 QPS 小于 aserver 机器数量 * 5 时，建议按照 aserver 机器数量 * 5 进行兜底配置。如图 4-12 所示。

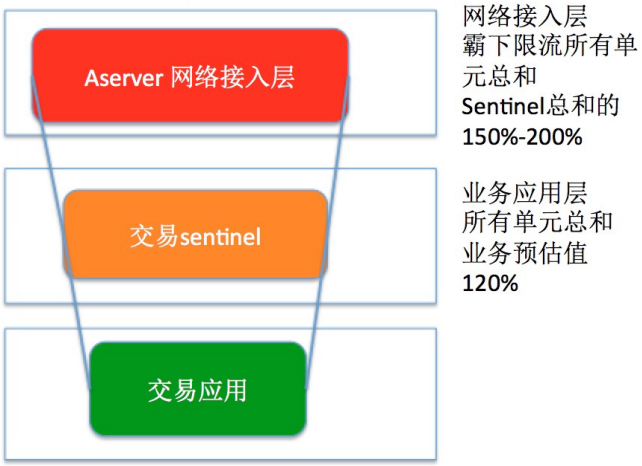


图 4-12 集团 Aserver 限流示意

限流原则

- **原则一**：业务核心入口应该配置限流，防止下游容量不够，未能很好进行自我保护。

对于系统的核心入口，应该严格按照评估值，并在保证容量的情况下按照评估值进行一定比例的限流一般按照预估值的 120% 限流)。在实际系统中，由于业务的变化，无法有人能非常细致的评估每一个请求对下游的放大比，而且也无法清楚的了解下游应用的 owner 是否也已经使用合理的限流或者降级策略来进行自我保护，因此入口应该按照业务预估值进行严格限流，而非按照自己所能承载的量进行限流。

- **原则二**：每个应用应该设置合理限流进行自我保护

每个应用应该根据自己系统及自身依赖中间件 (如 Tair) 容量进行压测评估，明确上游对自身请求 QPS 达到某个阈值会触发自身系统的异常，并配置对应请求的 80%-90% 为限流值。如线程池满，RT 增大。缓存被击穿，大量请求打爆数据库或者下游服务。缓存容量达到瓶颈，缓存限流后大量请求打爆数据库或者下游服务。

- **原则三**：每个应用应该配置系统限流进行兜底保护

系统限流保护主要针对入口请求，根据当前单机系统的 cpu，内存，load，平均 RT，系统线程数对当前系统进行保护，其中最核心的是 cpu 和 load，建议配置是 load 值为当前 CPU 核数的 3-5 倍，cpu 平均值超过 80% 时候拒绝请求。由于当前业务容器有可能跑在超卖的宿主机或者和核心业务混跑，或者宿主机异常导致 load 值异常飙升，此时业务处理可能会因此受到影响，直接影响整体系统的稳定性。

- **原则四**：合理使用单机限流与集群限流，降低误限率

对于高限流值或者流量均衡应该使用单机限流，对于低限流值或者流量不均衡必须使用集群限流，单机限流值小于 5 的都应该使用集群限流。当单机限流值过小时，请求可能在还没有完全处理完成时候出现积累，单机限流对于这种场景的处理非常容易出现误限而影响业务转化率。而且非常小的限流值

在流量稍微不均衡的场景下，也很容易触发限流，因此需要使用集群限流来保障整体限流值。

典型案例

案例 1: Lazada 核心链路限流模型

Lazada 典型的核心链路限流模型，以交易核心链路为例，入口在霸下会进行一个约 2.75 倍预估值的限流值，其中 2 倍为无线的限流值，0.75 倍预估值为 PC 端的限流值。按交易顺序配置购物车加购渲染 / 下单渲染 / 下单进行一个漏斗式限流，如图 4-13 所示。

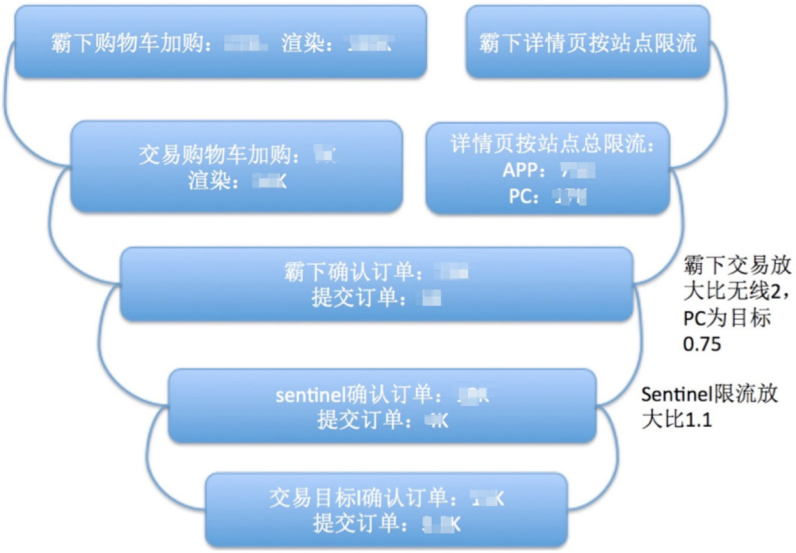


图 4-13 Lazada 核心链路限流模型

案例 2: 大促模型下的限流对业务的影响

大促模型下东七区的下单流量模型输入：在压测的第三阶段，在预估值 110% 下零星限流，保护系统；进而在 150% 预估值下，入口大量限流，进而保护整个系统稳定性。图 4-14 展示了泰国站点限流情况。

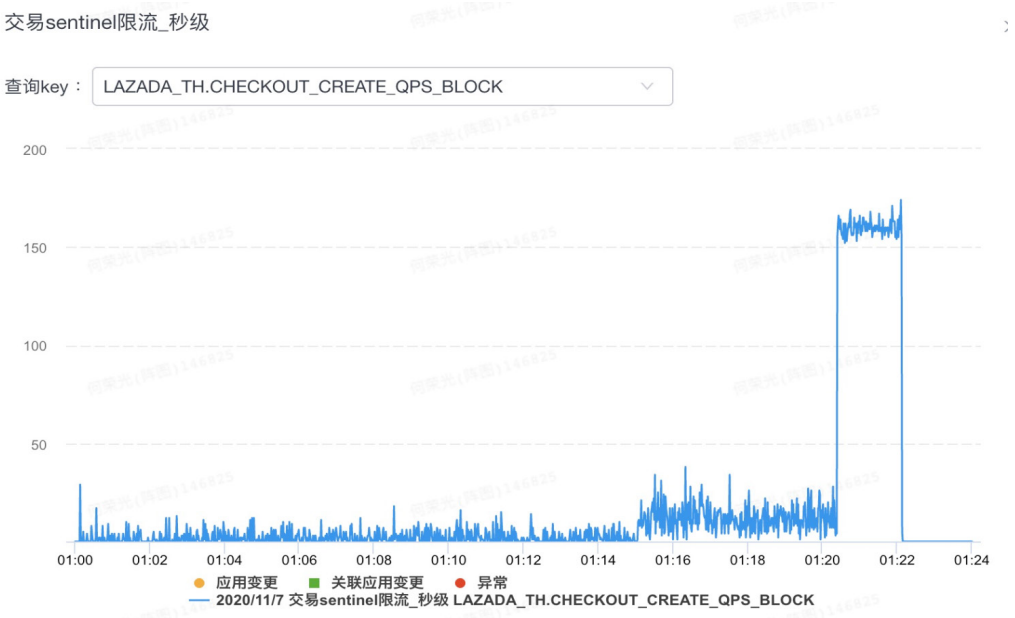


图 4-14 泰国站点限流示例

仅限内部传阅

4.3 大促稳定性保障

4.3.1 大促标准化

① 背景目标

国际化的大促活动日益频繁，尤其体现在各国大促活动需求多样、多国地域协同成本高、资源组织投入不均衡、业务需求快速迭代迫切等，技术侧结合历史大促沉淀，就效率提升、大促传承能力进行了标准化 SOP 的设计，目的是统一技术域内及上下游能力，提升大促效率，平衡效率成本，以期全球化大促活动后续开展能够有借鉴、有继承、有创新能动力。

② 大促活动级别定义

大促活动根据 GMV、组织、资源等方面的因素进行级别定义，便于技术业务的对焦，匹配技术资源投入、保障能力，不同级别的大促保障活动力度也会不同，即进行保障活动的裁剪，高级别重保投入、中低级别轻量级投入。当然，大促活动每年都会有所不同，本文介绍的内容仅作为参考，可以根据具体的大促情况进行定制。

活动保障等级	级别范围	举例说明	技术资源投入参考
S1	GMV导向，集团战略性大促，涉及全品类 （对应业务S级）	双11、328	全员投入，全力保障 （除业务需求，预计技术保障投入1000+人日）
S2	年中、国际化特色重大节日大促 （对应业务S级）	618、828、黑5	核心链路重度参与，其余轻度参与 （除业务需求，预计技术保障投入200+人日）
S3	定期如季节性活动，不限于地域，1+N品类 （对应业务A+级）	3月春上新、双12+圣诞节	核心链路轻度参与，其余可不参与 （除业务需求，预计技术保障投入50+人日）

表 4-1 技术侧大促活动级别定义

事项	适用等级	保障策略
大促业务需求 （营销玩法输入）	全部	根据业务玩法进行对应的链路保障
业务&技术目标	全部	
启动仪式 （立项&KO动员）	S1、S2	S级大促间隔时间<5个月采用增量保障，>5个月采用全量保障
项目管理 （需求、计划、进度、风险等）	全部	可视化项目管理 1、需求管理 2、计划进度管理 3、接口人管理 4、风险管理 5、周会、周报制度
封网&变更管控	S1、S2	预热期应提前对核心应用进行封网限定； 变更需遵循特定审批流，并进行核心应用的变更量化 S1保整个大促，S2保首日
组织&系统保障		1、作战室安排&设备硬件支持，战前中宣传，OC&HR组织协同 2、作战手册&各域手册review 3、问题排查指南 4、问题处理流程&SLA标准 5、战前技术分享&考试 6、值班人员收集&排班
	S1	S1上述1-6需全员现场组织保障；
	S2	S2核心链路现场参与保障，其他链路可选oncall
	S3	S3建议核心链路oncall保障
应急&联动机制	S1	1、战前：GOC应急指挥钉群组建、故障应急演练、值班安排-现场/oncall 2、战中：应急联动、P1P2级故障应急协同、当日问题总结
产品/技术复盘	全部	各业务产品线、国际化技术复盘 （S1） 其他等级需在活动结束后自行组织复盘

表 4-2 全局性活动裁剪

③ 大促组织及协同

大促需要整个部门的各种角色的协同参与，如图 4-15 所示，其中核心的角色比如：总指挥、大队长、副队长、PMO、稳定性负责人、测试负责人、业务 / 产品 PM、GOC、HR 行政等多方面。

图 4-15 大促组织协同

④ 大促活动生命周期

大促的生命周期可参考如图 4-16 所示，根据大促实际情况，有侧重点地评估，具体做哪些系统业务稳定性建设。

图 4-16 大促生命周期

仅限内部传阅

⑤ 大促专题专项

编号	专项内容	内容说明	结果衡量
1. 业务稳定性保障			
1.1	需求管理	制定需求冻结计划、跟踪日常/大促态需求进展、明确版本发布节点，并定期跟踪实施结果，规避线上风险	是否有需求延期，尤其是影响大促相关的需求及时把控风险
1.2	变更管理（审批流机制）	明确需求变更、审批流程，评估并决策变更实施方案，尤其是封网期变更的关注	变更是否符合预设条件，是否有违反红线情况
1.3	功能预演&捉虫	组织全链路的产品功能，做功能预演、产出预演报告且推动问题的解决	大促是否有预演遗漏
1.4	国际化时区（时差）	根据各国大促时间段，制定对应的大促保障、值班、风险对应的策略，确保全量覆盖大促活动上下线周期	大促时间段保障措施是否覆盖全面、无遗漏
2. 稳定性保障			
2.1	压测 & 验收	压测相关的压测模型、压测数据、压测操作、压测复盘以及压测问题跟进	大促问题是否和压测相关
2.2	攻防/故障演练	针对系统薄弱点以及运维能力，做特定问题场景问题注入，练人练系统练流程，提升整体能力	是否由于问题处理能力差引发问题
2.3	资损专项	针对潜在的资损的场景和变化点做梳理，匹配监控、报表、对账三板斧的方法，并处理预警	大促期间是否有资损
2.4	预案准备 & 执行	针对业务以及系统的问题，构建前置预案以及紧急预案，并组织预案的评审、演练以及预案的在线化	关注大促期间预案准确性
2.5	限流降级专项	针对系统限流、降级、鉴权、系统保护、霸下等限流措施做评估以及落地，确保限流有效性	大促是否有限流问题
3. 系统保障			
3.1	性能容量评估	针对容器、数据库、中间件、缓存等体系做全局容量，评估业务领域容器情况，产出扩容计划等	大促是否有容量相关的问题
3.2	核心应用/交易链路	梳理每个核心领域的架构、链路现状，主要是匹配大促保障的潜在优化点	梳理点治理是否都有效闭环跟踪，尖峰是否出现问题
	尖峰链路梳理	针对尖峰时刻的链路，做彻底的分析，专项产出链路上的应用以及缓存和数据库，确保100%准确	
3.3	系统强弱依赖关系梳理	针对链路做强弱依赖的梳理，多少条链路，什么是强依赖，什么是弱依赖，并驱动治理	是否有强弱依赖问题
3.4	监控+中间件	系统监控以及业务监控，能够基于Sunfire等监控系统，产出领域内或者BU内的监控体系，快速做故障定位以及日常盯屏等	是否有中间件相关的问题。关注大促期间监控有效性
		针对中间件涉及到的容量以及预案，做全局的评估，驱动相关的薄弱点治理优化	
4. 汇报&会议			
4.1	战中保障	明确大促日人员到位、区域安排、日程及应急联动机制等保障措施	大促期间各项活动有序开展，衔接不段层
5. GOC专项			
5.1	GOC专项	指定封网变更、应急联动、历史故障传承action落地、工单清零等专项治理及策略指定	大促作战应战期无封网、应急联动等实施活动露点
6. 技术目标管理			
6.1	技术目标管理	技术目标指向性强，具备可度量、可回溯有较大牵引力	目标可在复盘期进行回溯对焦结果

表 4-3 大促稳定性专项参考

⑥ 大促中控视图

大促保障：需要精细化管控、平台化支撑、强健的执行力才能确保各个步骤环环相扣，产出预期的结果。图 4-17 对大促全局视图进行了概览介绍。

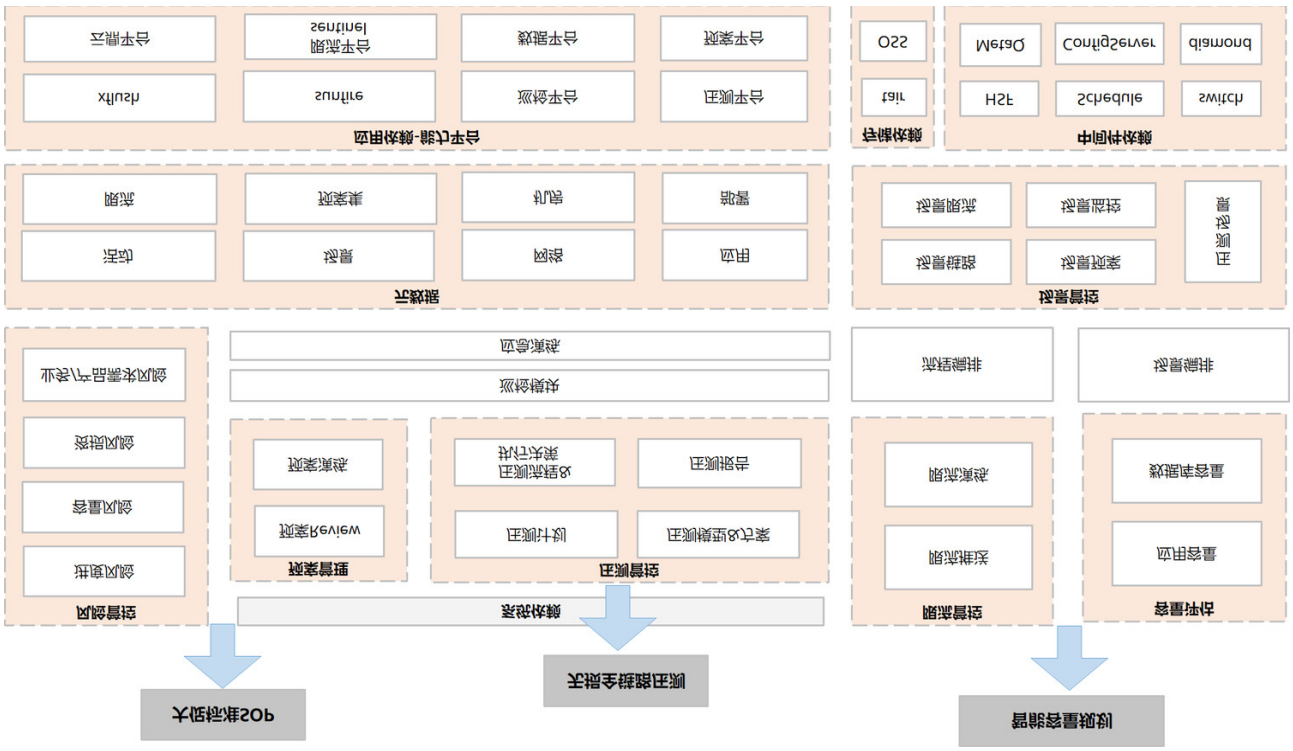


图 4-17 大促中控视图

4.3.2 链路梳理

背景

全球当前面临的一个问题：各个域自己的链路相对清晰，比如下单链路、支付链路，但是没有一个同学能够清楚，一个买家从进入到 APP 的动向是怎么样的，比如进入首页 -> 搜索 ->detail-> 加购（立即购买）-> 购物车 -> 下单 -> 支付；基于从全局视角的维度来看全链路的必要性在大促全局作战的过程中是非常有必要的，基于以上思考进行全链路进行梳理。

方案

简单一句话进行实现：找入口、找基础设施、入口和基础链路进行梳理、根据各域梳理结果产出一张全链路大图。以下分别进行介绍，找核心入口：PDP 详情、购物车加购、购物车渲染 / 修改、下单页渲染、提交下单、创建订单、收银台渲染、提交支付、支付结果查询、支付方式查询、渲染前置收银台、创建履约单；核心基础设施：IC、UIC、SHOP、SELLER；各域进行链路梳理：针对入口流量各业务进行链路梳理，使用工具和人为决策的方式进行梳理，最终产出一张链路梳理图，如图 4-18 所示，每个域都会产出一张类似的图。

仅限内部传阅

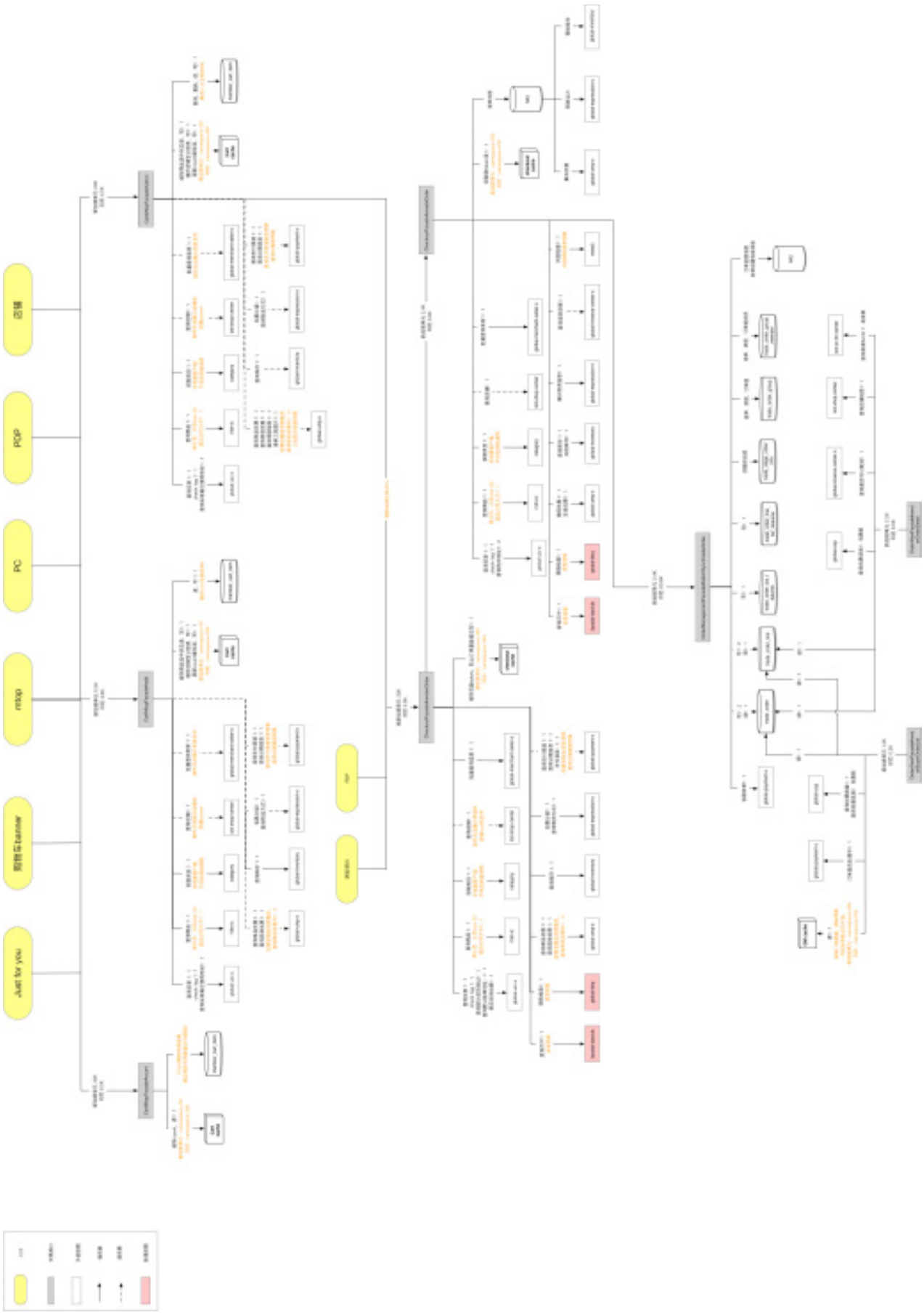


图 4-18 链路梳理示意

基于以上梳理的链路进行全链路融合，最终产出一张全局大图，如图 4-19 所示，大图会包含所有核心入口以及对应下游调用、强弱依赖、预案、核心存储（tair、DB）等。

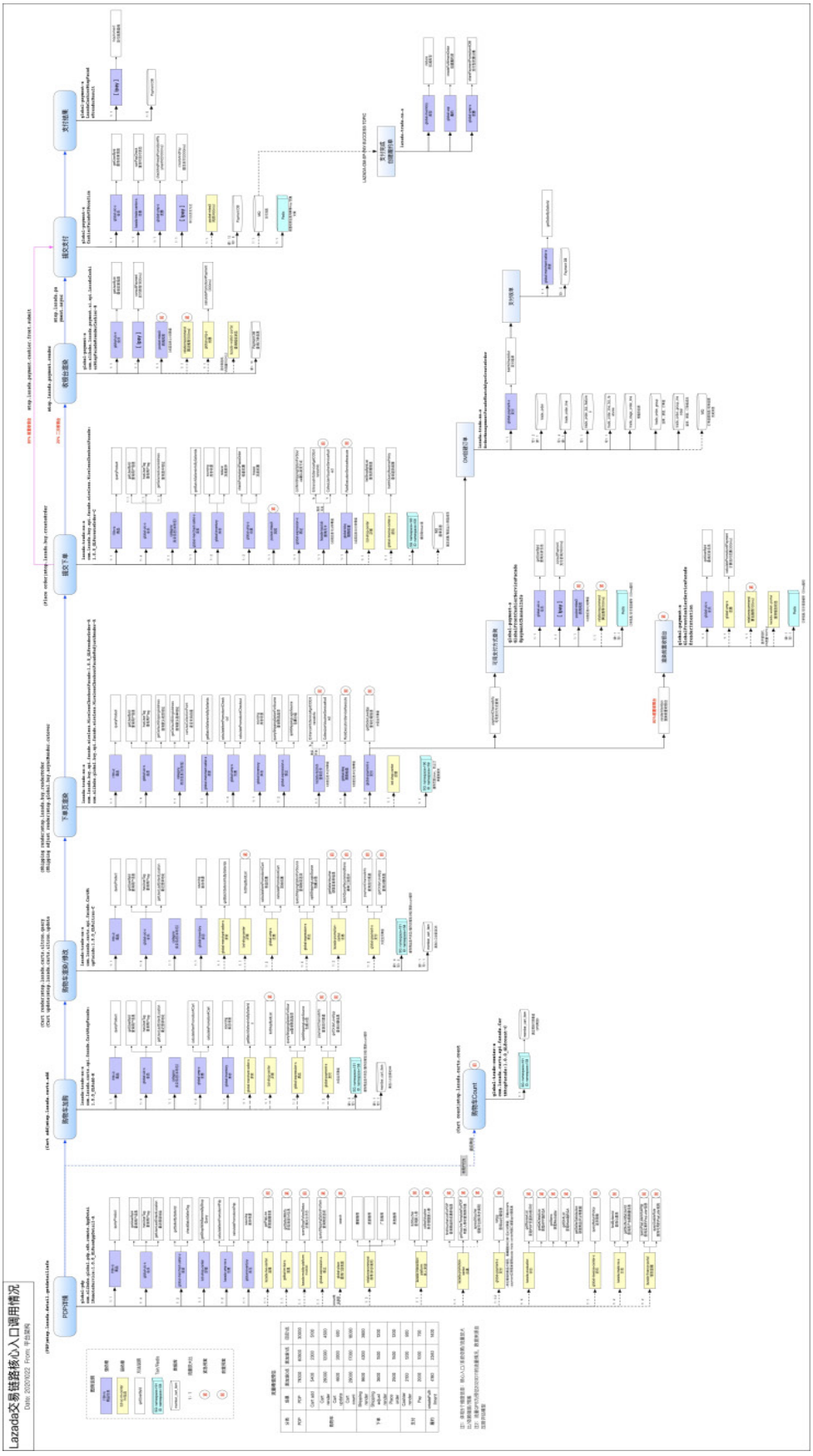


图 4-19 全局链路大图示意

仅限内部传阅

- **强弱依赖**：借助此图进行了完整的梳理，是否是强或者弱依赖可以通过攻防演练的方式进行验证，确保弱依赖有问题后不会影响主链路的流程；
- **放大比**：有了上游对下游的放大比，对后续做压测模型是个非常重要的数据，能够更清晰的对比压测模型跟链路的匹配程度；
- **压测模型**：全链路验收重点保障的一定是核心链路，当有了上述链路图后，核心入口就不会被遗漏，确保链路压测模型的完整性；
- **预案**：通过对预案的梳理，确保核心调用需要有对应的紧急或者提前预案，另外也可以看到预案的合理性，通过对链路的梳理，可以识别出预案是否有遗漏；
- **核心中间件、存储**：通过梳理可以识别到核心对应的中间件，如 MetaQ、Tair、DB 等信息，避免在稳定性保障中遗漏。

未来展望

目前所有梳理还是基于人的方式进行梳理，还是希望未来能将这张大图的产出自动化，我们只需要维护对应的入口即可，从系统上自动将对应的链路进行梳理清楚，减少人的成本。如下图为系统自动采集链路。

4.3.3 压测模型

背景

国际压测时面临了比较大的困局，比如压测场景不全，导致漏掉某些场景，最终导致峰值时出现问题；压测 qps 不准确，评估方式不统一，压测模型的准确性、压测结果对比、正式流量对比的缺失，导致无法看到流量模型的准确性。

方案

① 压测模型评估：基于上述困局，需要进行破局，最重要的一步是全链路梳理，基于上一章内容进行展开，确保满足 MECE 法则，如图 4-20 所示：

- **压测场景不漏**：全链路进行梳理，上一章已说明，此处不做赘述；基于梳理出来的链路进行入口、底层调用梳理，确保不漏；基于特殊场景进行梳理，如：购物车模型、优惠模型等。
- **压测 qps 准确性**：确定历史数据准备，历史数据是最好的参考，如 GMV、DAU、QPS 准备等；根据历史数据制定公式进行入口流量的评估；基于入口流量对基础链路的放大比进行下游评估。
- **准确性**：按照接口维度进行 qps 搜集；将预估出的 qps 与搜集的 qps 进行对比确保数据的准确性。

图 4-20 压测模型构建方法

② 按照压测模型进行数据构造，通过上述评估后，第二步就要进行数据构造，数据构造相对来说是最复杂的部分，目前需要开发同学配合测试一起的方式进行数据构造，最终确保数据构造与评估出来的流量是一样的。

③ 压测模型验证：完成前面两步后，还需要最终对压测数据做验收，需要确保实际压测数据与预估数据是一样的，这里的对比需要逐个接口和模型进行对比，确保实际压测与预估的模型的一致性。

至此从评估、构造、验证三个方面进行了压测模型的准确性校验。剩余的工作就是对压测模型的查缺补漏，确保场景不被遗漏。如图 4-21 所示，为其中一次全链路压测时入口流量与预估流量的对比。

领域	场景	接口API	SG(+8)			MY(+8)			PH(+8)			ID(+7)			VN(+7)			TH(+7)		
			目标	限流	压测	目标	限流	压测	目标	限流	压测	目标	限流	压测	目标	限流	压测	目标	限流	压测
PDP	PDP	http://acs-m.lazada.sg/gw/mtop.lazada.detail.getDetailInfo/2.0/	10000	符合	12327	30000	符合	37739	30000	符合	36477	30000	符合	31224	17900	符合	19256	28600	符合	30914
交易	Carts add	https://acs-m.lazada.sg/gw/mtop.lazada.carts.add/1.0	300	符合	358	950	符合	1118	4000	符合	4042	5100	符合	5167	450	符合	643	600	符合	777
交易	Cart render	https://acs-m.lazada.sg/gw/mtop.lazada.carts.ultron.query/1.0	2200	符合	2597	7500	符合	8370	15000	符合	15314	6000	符合	6460	2200	符合	2175	4200	符合	3960
交易	Shipping render	https://acs-m.lazada.sg/gw/mtop.global.buy.pressure.shipping/1.0	550	符合	964	2400	符合	4066	6300	符合	10609	3600	符合	4435	1100	符合	1406	1000	符合	1406
交易	Place order	https://acs-m.lazada.sg/gw/mtop.global.buy.pressure.placeorder/1.0	250	符合	328	850	符合	854	1300	符合	1634	1200	符合	1253	400	符合	474	500	符合	603
支付	Cashier rednder	https://acs-m.lazada.sg/gw/mtop.global.payment.render/1.0/	250	-	270	850	-	885	900	-	910	800	-	963	300	-	362	400	-	483

100% 压测时，部分场景没有达到预期流量；
130%时，全部场景都能压到高于目标流量
支付本身单机限流，sentinel值比较大，所以没有限流触发
150%期间印尼加购只有很少的限流
150%期间印尼下单零星限流，只有一秒有限流峰值，流量在 150% 左右达到目标值；

图 4-21 全链路压测入口流量对比示意

未来展望

如何通过更好的方式来完成流量评估并确保不遗漏、流量构造、流量验证三维关系如何更好的打通，并通过系统进行全程沉淀，最终将人力成本做到最低，未来的想象还是很大的。如：前的准确性都是基于开发自行搜集，如何更好的系统化；业务只需要评估入口流量，剩余的流量都应该根据线上真实情况进行仿真；压测的准确性与实际链路的覆盖比例，是否出现漏的场景；压测数据的构造如何打通；

仅限内部传阅

4.3.4 预热

背景

预热包含两部分：冷数据预热和冷系统预热。

冷数据预热指系统刚启动，包括缓存、数据库的磁盘等系统处于冷状态，此时容易被击穿，导致业务不可用。冷数据预热如图 4-22 为冷数据预热，指业务流量进入业务服务器时，系统中对应的 key 值并不在缓存系统中，此时业务流量会请求数据库，当业务流量较大且缓存命中率低的时候，瞬间数据库流量过大可能会导致 RT 上涨，宕机，甚至是系统雪崩，因此需要提前预估好数据，通过工具让冷数据变 " 热 "。如 2016 年双十一大促的前两分钟，就由于系统没有预热，冷库导致了大促前两分钟交易成功量下跌。懒加载即使用时再进行加载，当系统启动时，系统处于较冷的状态，此时用到一个加载一个，对整体服务造成的影响不可控，容易影响用户体验。

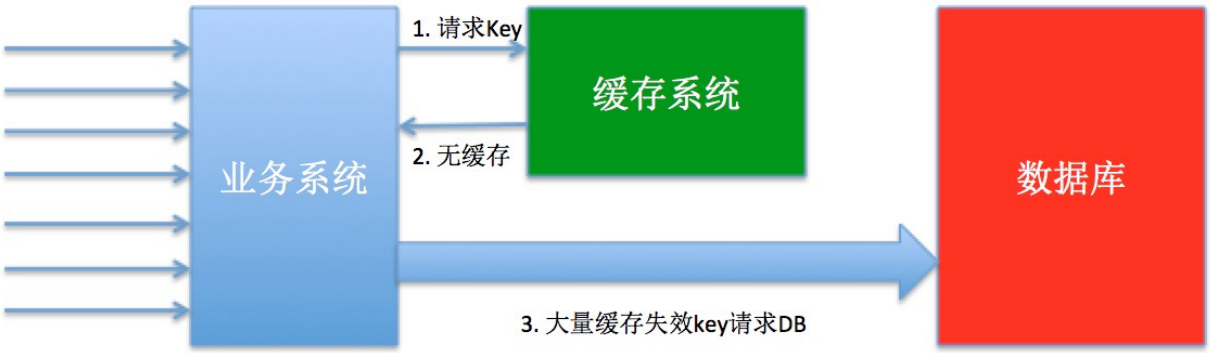


图 4-22 冷数据预热示意图

冷系统预热神器之 Alihot，Alihot 工具是协助业务自行预热的预热平台，以下为 Alihot 的设计和简单流程，如图 4-23 所示。

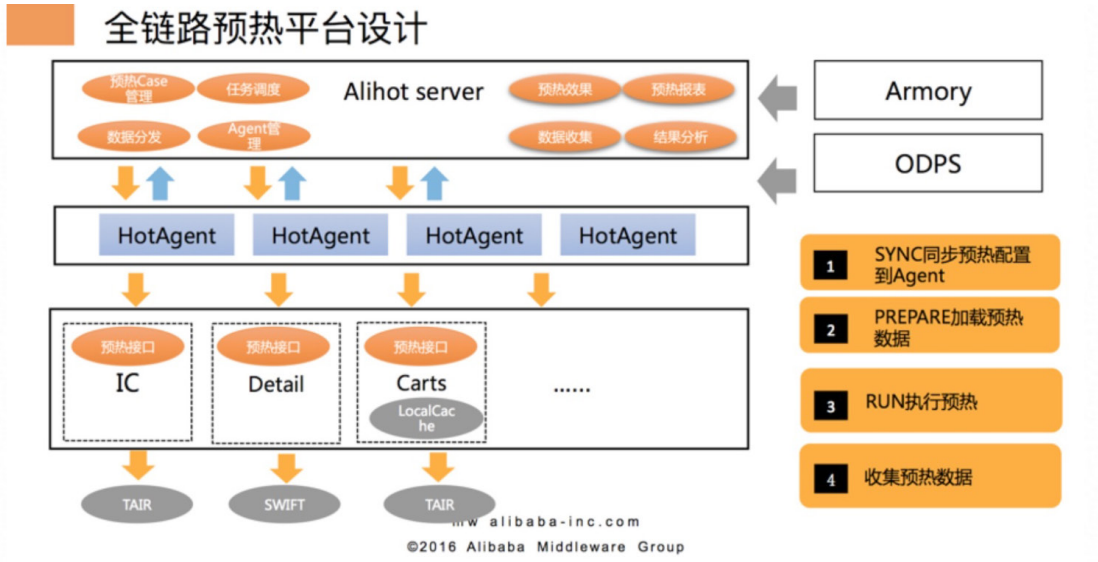


图 4-23 冷系统预热 Alihot 工具

接入预热平台需要以下操作：

- 预热平台上进行相关的配置，简单配置如下图，配置了相关的预热数据，包括应用，请求，预热方式和链路，数据来源、数据等相关信息。预热平台会将这些预热配置下发到预热的 hot agent 中，后续由 hot agent 加载需要预热的数据并对线上业务发起请求，进行预热。
- 业务方需要评估所需要预热的数据，比如购物车，需要评估预热几天 / 月之内的购物车数据，选择合理的预热数据量，并导入到预热平台的 ODPS 中。
- 数据导入后，hot agent 会根据配置，开始执行，将数据从 odps 中读取，然后根据配置的请求速度对线上发起请求，进行数据预热。
- 最后收集预热数据，汇总整体预热的结果，如图 4-24 所示。

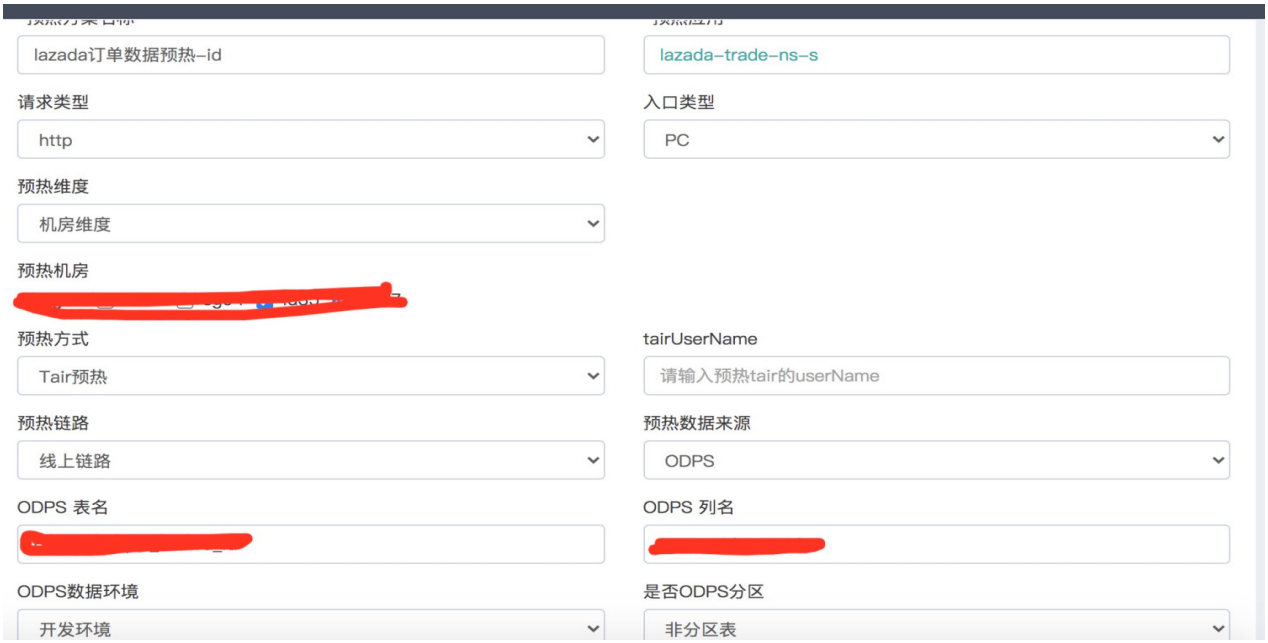


图 4-24 预热数据示意图

原则

- **原则一**：涉及缓存和热点的系统应该提前预热，防止冷数据影响系统稳定性

涉及到使用了缓存 (本地缓存或者 Tair) 的都应该考虑提前预热，可以是业务自然预热，如果效果不明显就需要使用工具对热点进行主动提前预热。

- **原则二**：核心场景尽量主动预热，防止自然预热不到位

核心场景中涉及缓存使用，如购物车、优惠系统，在大促期间都应该主动预热，尽量不要等待业务自然预热，防止过久没有使用的冷数据在大促期间突然访问导致缓存命中率低下，击穿系统。

- **原则三**：新启动的 Java 进程应该尽量进行 JIT 预热

由于大部分应用在大促前一段时间 (一天 / 半天 / 快上) 都会进行重启，重启后的系统不应该马上投入实际业务使用，冷系统会造成业务访问问题。

- **原则四**：选取合适的预热入口和预热数据量，防止过久数据淘汰热点数据，以及影响大促战前

仅限内部传阅

准备

预热的数据量选择需要进行合理的评估，过长的数据 (比如超过一个月)，就有可能出现预热后，冷数据淘汰热点数据导致大促期间缓存一样被击穿。此外，如果数据量过大，单纯一个应用的预热请求会比较有限，如果要加快预热速度，还需要上下游打开相关预案进行提速，一旦下游某个环境预案遗漏，就会在预热过程中出现线上故障。

典型案例

案例 1: Lazada 购物车 / IC 主动预热

2019 年双十一的时候 Lazada 就由于购物车的冷数据出现过问题。今年的大促期间，Lazada 购物车加入了预热环节 (压测预热到后来的 alihot 平台预热)，预热数据为一周内的购物车商品，预热链路包含了下游链路，主要预热到了优惠场景，保障了大促期间的冷数据不再存在。

案例 2: Lazada 购物车预热评估及实现流程

本次 99 大促和双十一大促购物车均加入了预热环境，一方面是预热购物车业务，同时也顺带预热了下游的优惠系统。99 大促最早开始评估的时候，准备采用一个月的购物车预热数据，后来发现数据量过大，在下游不开启预案的情况下，预热时间很长，会影响到大促当天的其他战前准备流程。后来重新评估数据，采用一周的购物车数据，数据量大约 2000W 条，6 站点白天预热完成。在双十一备战的过程中，这个痛点也解决了。购物车预热依然选择一周数据进行预热，同时，拉通下游的业务同学，整体拉通预案执行，整体预热速度提速了两三倍。

五、全球容灾

5.1 容灾背景与定义

为了持续提高可用性级别，系统必须具备强大的容灾能力，以在遭遇灾害时，仍然能够保证系统正常运行，保障关键业务实现连续运行目标。在架构设计中，作为系统高可用性技术的重要组成部分，容灾设计强调的是系统对外界环境影响具备快速响应能力，尤其是当发生灾难性事件并对 IT 节点产生影响时，能够具备节点级别的快速恢复能力，保障系统的持续可用。

在介绍容灾之前先来介绍下 Failover，中文名为故障转移，是指计算技术或网络技术中，当发生故障时，将计算任务或网络请求切换到冗余的或备用的计算机服务器，系统，或网络上。当需要提供类似于 9999 这样的高可用性服务时，系统设计者往往都需要思考如何为自己的系统设计 Failover。Failover 其实无处不在，Raid 技术为磁盘提供 Failover 能力，Load balancer 技术提供为 Web 应用程序提供 Failover 能力，Master 及 Slave 技术提供数据库的 Failover 能力，HSF 或 Dubbo 服务框架也会提供服务的 Failover 能力。

多机房容灾技术就是指多机房间的 Failover，通过将用户请求在冗余机房间转移来避免故障对用户的影响。而容灾技术不仅仅能解决整个机房层面的故障，如地震等自然灾害带来的机房毁灭，或断电带来的整个机房不可用。它还是 Failover 终极解决方案，由于机房内的应用程序隔离性小，相互间很容易产生影响，此时机房内的组件或服务间的 Failover 可能受到同一个原因的影响都出现了故障，此时无法通过机房内 Failover 解决问题；而多机房间往往应用程序是相互隔离的，相互间的影响较小，不容易受到同一问题的影响，因此当出现故障时，基于多机房的容灾技术往往可以绝对解决这个问题。

5.2 国际化经典故障案例

① UIC 接口被下线

UIC 是我们的用户中心服务，所有的用户信息都需要调用 UIC 进行读取，这里面部署了多个微服务接口。工程师的一次变更中，阴差阳错删掉了某个接口，并且进行了灰度发布，然后系统容量充足时，是发现不了这个问题的，因为服务接口调用都被路由到了未发布的实例上，而进行灰度的实例还在提供这个接口的服务。直到当剩余节点数无法支撑当前调用量时，才发现问题。万能的灰度发布呢？

② 网关配置变更

我们使用的一个公共网关服务进行了变更，并且这个变更是定时生效的，大约在北京时间早上 5 点生

仅限内部传阅

效后，变更使得调用时长增加带来了部分上游系统的 OOM，团队快速响应了这一个故障，但是在钉钉群沟通时，有人提到了怀疑是某个发布导致的，而其实这个发布并不是故障的根本原因，却转移了所

有人的目光，大家只盯着这个问题看，有的人干脆事不关我，不作理会了。后果显然是惨重的。确保一个复杂系统的稳定性，并且保障团队大于 20 人时，如何有效协同？

③ 运营商网络故障

我们发现，某 Tier1 的运营商经常出现网络抖动，甚至有的时候是某个国家到某个机房网络都不可达，但是这个运营商一直定位不到问题，这个时候我们怎么办，如果容灾链路都不可用了怎么办？难道要怪运气不好吗？就算不能彻底解决，但是当前有没有优化方案来缓解这一问题呢？我们未来到底要投入多大的成本来避免这种统计意义上很小的概率，但是又在我们身上不巧的经常发生的问题呢？

面对这些典型的故障，我们需要用系统容灾的思路来进行解决，我们进一步来看一下国际化容灾的核心原则。

5.3 国际化容灾原则

国际化的业务系统会综合考虑法律合规、数据一致性、稳定性、性能、Scalability、成本及容量等方面。考虑容灾时，核心原则是：**第一优先级是满足法律合规要求，第二优先级是数据一致性，第三优先级是 Scalability，最后是成本、容量、性能、稳定性方面的考虑。**因为法律合规决定一个业务的生死；数据一致性没有保障就无法保障功能的正确性；Scalability 没有保障，则业务做不大。如图 5-1 所示。

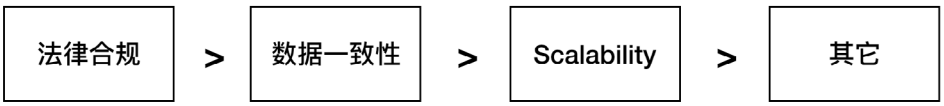


图 5-1 国际化容灾核心原则

通常情况下，我们使用极简单的就近服务作为编排策略，举例俄罗斯用户使用俄罗斯机房进行服务，基本可以满足以上的需求，还能做到性能的最优化。

下面我们来看看通用的容灾方案，比如在全球多个物理区域存在自建机房或使用云厂商机房，在此基础上，我们还希望实现如下能力：

1）每个机房对等部署所有的应用程序、服务、数据库等；
2）对于任何一个用户，可以让任何一个机房为其提供服务；
3）用户所有的操作产生的数据是在为其服务的机房内部封闭的；
4）机房间是可以相互灾备的，在灾备切换的整个过程中确保数据一致性；
5）可以让用户通过任何一个可行的网络链路到达这个机房；

6）可以让任何一个 CDN 厂商为用户就近提供静态内容的访问；

7）用户如果在机房迁移过程中，系统确保数据一致性；

8）用户数据是否可以在机房间复制是可以配置的。

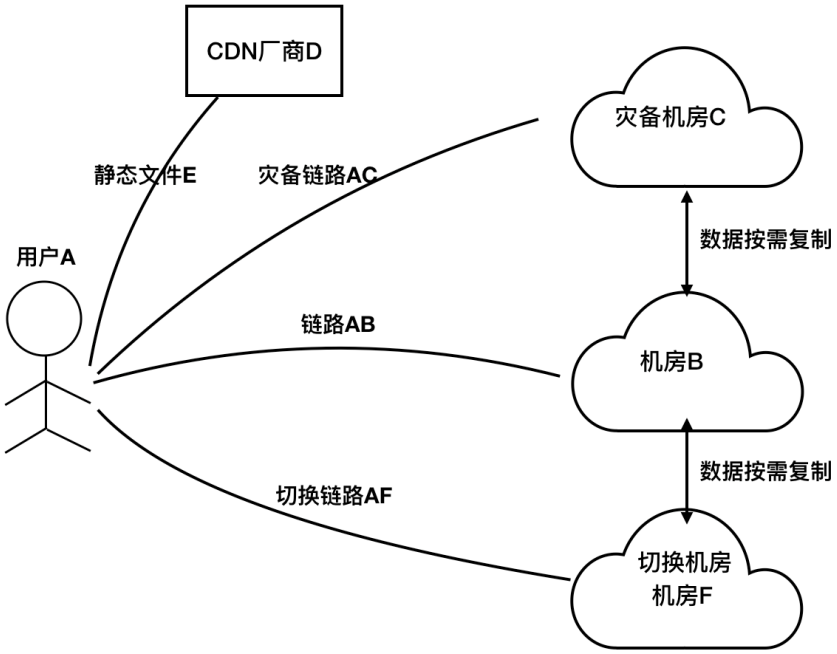


图 5-2 国际化容灾举例

我们以图 5-2 中为例说明以上的几个能力。用户 A 代表网络平台上的任何一个用户，机房 B 代表为其选择的服务机房，可以是任何一个机房，链路 AB 是被选择的用户 A 与机房 B 间的任何一个可达链路，CDN 厂商 D 是被选择的为用户 A 服务的任何一个 CDN 厂商，用户 A 请求的所有静态资源 E 都是通过 CDN 厂商 D 返回，机房 F 是用户可能由于某种原因被重新指定的为其服务的任一机房，链路 AF 是用户 A 到机房 F 的任意网络链路，灾备机房 C 是当机房 B 发生故障时，将用户切换至任一灾备机房 C 机房，链路 AC 就是用户 A 到达 C 机房的任何链路。

整体来看，用户 A 通过网络链路 AB 到达为其服务的机房 B，用户 A 的所有操作所产生的数据都在机房 B 中封闭，即所有的服务调用及数据变更都产生在机房 B 之内；用户数据会被按需复制到所有其它机房，确保满足用户隐私保护及法律合规要求；当发生灾备时，用户 A 通过 AC 链路到到 C 机房，由 C 机房为其提供服务，且整个过程可以保障数据一致性；用户所有的静态资源的请求都会通过 CDN 厂商 D 为其提供服务；如果用户 A 由于移民或长期出差等原因，发现机房 F 对其提供服务收益会更好，则会将用户 A 切换到机房 F 中，并通过链路 AF 到达机房 F，过程中也可以保障数据一致性。

当我们选取距离每个用户最近的机房为其提供服务时，并且方案在数据一致性、容灾、数据按需同步及隐私保护等方面得以实现，则整体的解决方案可以最大化提到的所有的用户价值和平台价值。

仅限内部传阅

5.4 集团容灾部署架构方案

5.4.1 淘宝单元化

淘宝核心的单元化主要为了解决如下业务场景：

- **导购单元化**（只读类的业务，搜索推荐）
- **交易单元化**（有数据一致性要求的数据，交易下单）

整体的解决方案主要通过下面三种方法，如图 5-3 所示：

- **水平扩展**
- **异地容灾**
- **数据一致性**

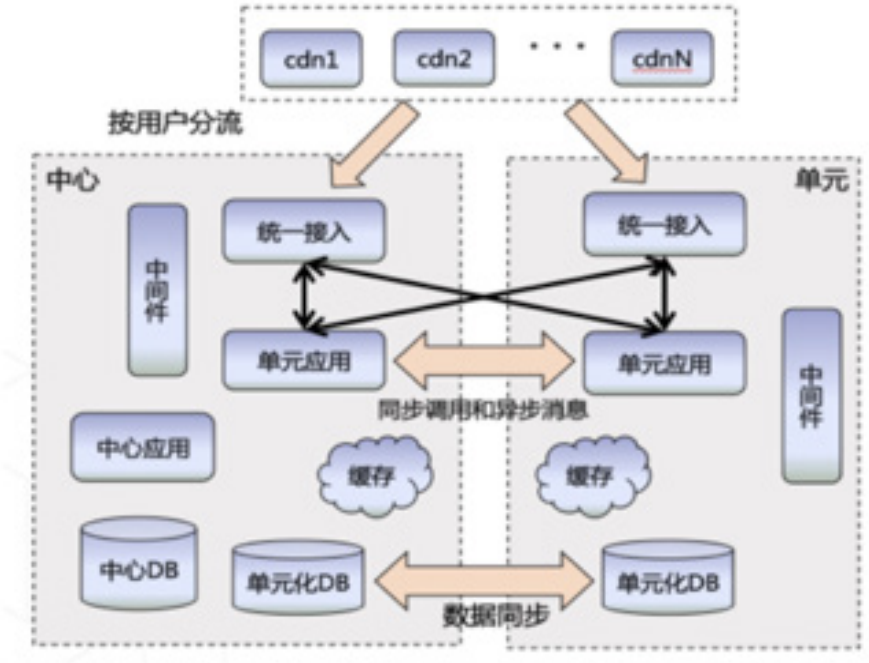


图 5-3 淘宝单元化容灾示意

5.4.2 蚂蚁 LDC 架构

在 LDC 架构中，一个单元被称为一个“zone”，并且单元种类不是唯一的，而是有 3 种不同的 zone。

- **RZone**（Region Zone）：最符合理论上单元定义的 zone，每个 RZone 都是自包含的，拥有自己的数据，能完成所有业务。
- **GZone**（Global Zone）：部署了不可拆分的数据和服务，这些数据或服务可能会被 RZone 依赖。GZone 在全局只有一组，数据仅有一份。
- **CZone**（City Zone）：同样部署了不可拆分的数据和服务，也会被 RZone 依赖。跟 GZone 不同的是，CZone 中的数据或服务会被 RZone 频繁访问，每一笔业务至少会访问一次；而 GZone 被 RZone 访问的频率则低的多。CZone 是为了解决异地延迟问题而特别设计的。

因为 LDC 架构带来的好处，在“两地三中心”IDC 架构的基础上，还实现了“分布式多活”的部署运行能力，具体包括：

- LDC 的 RZone 将可在异地多活部署运行，提供日常业务运行服务；
- 并且，RZone 间具备逻辑灾备切换能力；
- 一个 GZone 将在异地部署，作为杭州主站 GZone 的灾备。

5.4.3 AE 区域化架构

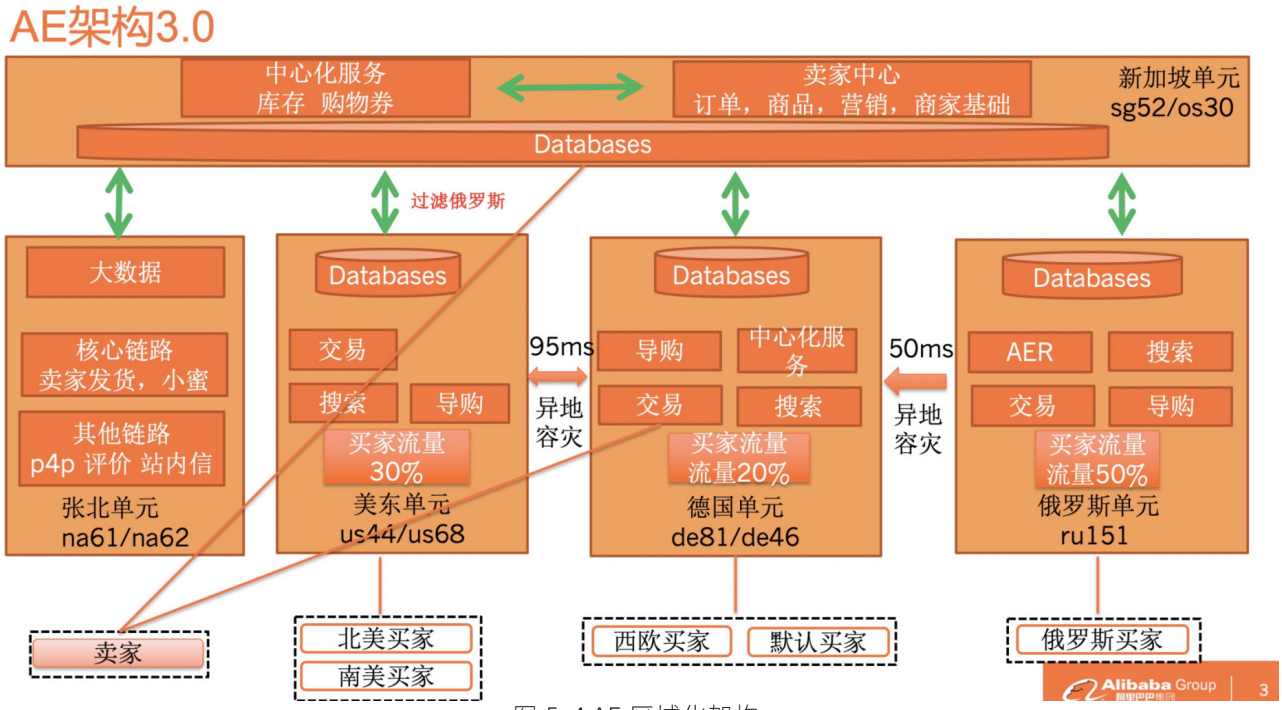


图 5-4 AE 区域化架构

区域是指地理区域，区域化部署就是按需在多个地理区域部署应用程序，全球区域化部署就是在全球的多个区域部署应用程序为用户提供服务，如图 5-4 所示。从应用程序分层视角来看解决方案，会更加清晰。从左至右从接入层到展现层，最后到数据层，在每一层中，我们都需要解决什么样的问题。

1）在 DNS 就近解析层，通过 DNS 与网络层调度编排技术的结合，实现了区域的部署，就近的访问；

2）在接入层路由中，实现了用户归属的一致性，从而保证了单一用户的单一数据 Master，从而确保数据一致性；

3）服务层路由确保共享数据 (全球买、全球卖) 场景下的单一数据 Master 及共享数据一致性；

4）TDDL 写保护在最后一层，当数据入库的最后一道程式中，确保单一数据 Master；

5）数据按需同步实现全球买卖及异地容灾。

整体的系统架构分为机房，管控系统，ZK 变更推送，技术栈 (基础设施、数据库 / 存储 / 数据同步系统、中间件、应用程序)，路由服务等几部分组成。

1）管控系统：区域化部署或机房调度技术的大脑，由管控系统控制所有的调度编排，用户归属等，

仅限内部传阅

核心是依赖大数据技术实现边缘、网络、机房的调度，管控系统就是具体实现及控制机房调度的系统。管控系统会部署在其中两个机房当中，一主一备，并控制所有机房中的用户归属及调度；

2）**Zookeeper**：确定了用户归属及调度逻辑后，如何进行每个机房中的变更，并确保全局一致性；

3）**机房**：分为中心机房及区域机房，为了降低数据同步的复杂度，采用数据同步，所有的机房与同一个机房进行双向数据复制，来实现数据的按需全量复制。而与所有其它建立同步连接的机房，及星型的中心就是中心机房，所有其它机房就是区域机房；

4）**技术栈**：基本需要在技术栈的每一层中进行落实，并且技术栈的每一层都需要依赖路由服务，确定如何进行用户、服务、数据的路由。技术栈分为基础设施如服务器；数据层，包含数据库及关联系统；中间件，实现服务的同步及异步（MQ 消息队列）调用的路由；

5）**路由服务**：核心逻辑就是一个映射表，用于落实管控系统所确定的哪个用户归属到哪个机房的映射。如何保证路由服务中用户归属的全局一致性，如何保证路由服务的高性能。

整个流程就是管控系统确认调度逻辑（用户的机房归属及路由逻辑）；将其通过 ZK 推送到各机房中，并更新机房中的路由服务和路由表；技术栈的各层调用路由服务来实现用户到机房的调度，来支持全球买、全球卖、区域部署、就近服务、异地容灾，并确保数据一致性，具体内容可以参考“区域化部署”章节。

5.4.4 LAZADA 同城双单元架

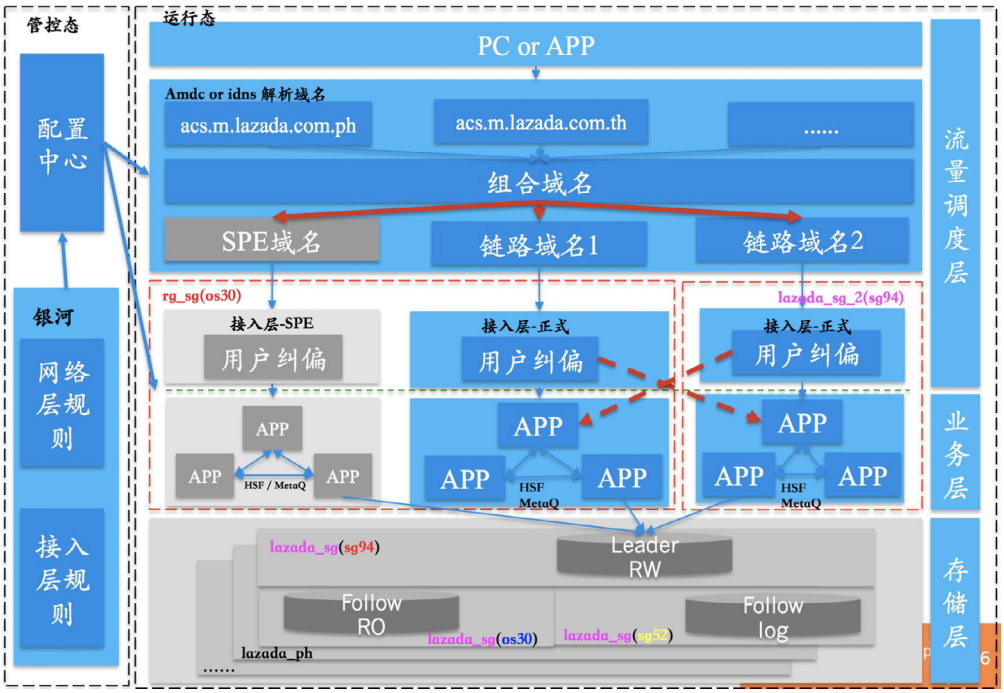


图 5-5 Lazada 同城双单元架构

Lazada 在国际化的背景下，如图 5-5 所示，主要构建了端到端的部署容灾架构，构建了中间件，存储，应用，容灾的方案，并通过技术产品进行运行态、管控态等多维度的管理。

5.5 国际化容灾方案

5.5.1 同城容灾

同城容灾指的是在发生灾难事故时，能够将生产中心的应用和数据通过在本地同城的容灾机房中的备份快速唤起，实现服务可用。在 LDC 架构下，通过消除单点，以及在 PaaS 上实现同城 IDC 一键容灾切换。同城容灾的响应速度相对较快，但是还有不足，一旦发生大灾大难，如杭州出现大的地震或者其他 IDC 灾难，将无法保证本地容灾机房中数据和系统的可用性，可能会导致所有业务不可用。所以，更高级别的高可用设计，还需要使用“两地三中心“+“分布式多活数据中心（Distributed Active Data Center）”的异地容灾部署架构。

5.5.2 用户粒度切流容灾

统一接入层在取得用户 ID 后，会调用路由服务（路由表）获取其归属机房。路由表中保存的其实是逻辑机房，逻辑机房会有一个状态及映射关系，当处于分流状态时，逻辑机房在 Diamond 中的值形如 "(US)(0-500 RU NORMAL|500-1000 US NORMAL)"，第一个括号中的 US 即逻辑机房；第二个括号中的表达式代表分流规则，并用竖线分隔，竖线之间的部分代表到物理机房的分流映射关系。而 "(US)(0-500 RU NORMAL|500-1000 EU NORMAL)" 则代表 US 这个逻辑机房的用户 ID 对 1000 取模，0-500 之间映射到 RU 机房，500-1000 之间映射还是在物理的 US 机房。图 5-6 从架构角度对这个例子进行说明。

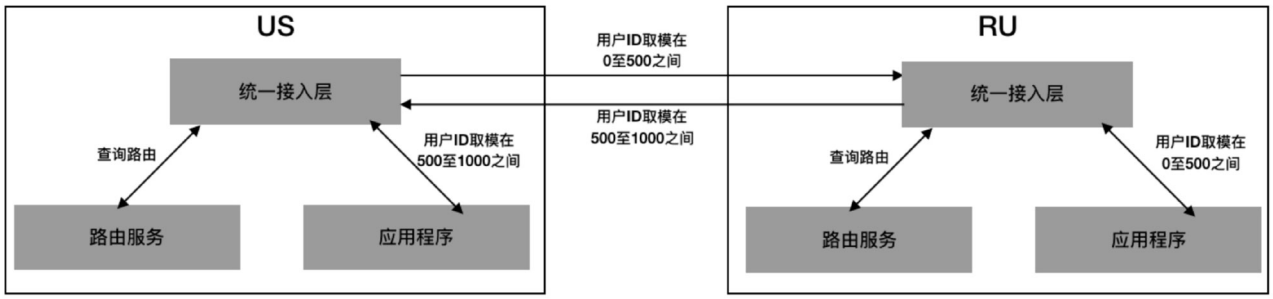


图 5-6 用户路由表切流示例

当我们进行这样的设置时 "(US)(0-500 RU NORMAL|500-1000 US NORMAL)"，代表我们期望将所有 ID 在路由表映射到逻辑机房 US 的用户中，百分之五十流量路由至 RU 机房。而通过大量的实验发现，当用户量足够大时，这个分流是非常精确的。当然不精确时，我们仍然可以通过逐渐切换的方式实现终的分流目标。

我们还会看到在物理机房后面有一个 NORMAL 的字样，代表当前分流的执行状态，为了确保数据一致性，切流过程是先经过 MIGRATING 状态，形如 "(US)(0-500 RU MIGRATING|500-1000 US NORMAL)"，代表将对 1000 取模结果为 500 以内的用户 ID 分流给 RU，但是当前正在变更过程中，还未完成，此时系统做的处理就是落在此区间的用户会被禁写，并且做 DB 复制位点检查，当 DB 复制到的时间点已经超过停写的时间点，则会将状态变更为 NORMAL，此时分流完成，用户不再禁写。整个过程如图 5-7 所示。

仅限内部传阅

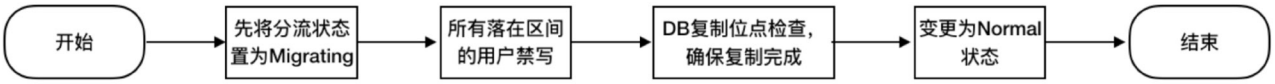


图 5-7 用户路由切流逻辑

5.5.3 异地容灾切流

区域化部署的容灾技术是对上面提到的技术的升级，是一种更加完善的全链路应用的容灾技术。在全球化的需求作用下，区域化部署技术已经将全链路的应用程序部署在了多个国家和区域，并且数据实现了按需复制，这几乎可以理解为用户需要使用到的应用程序和数据都已经完整了。区域化部署的容灾技术同时解决了数据一致性问题，也让容灾生效的时间更快且不受到 DNS 的 TTL 影响。提到区域化就不得不提到路由表和路由服务，当进行容灾切换时，并不用更新整个归属机房对应用户的路由表，而是将用户所对应的逻辑机房在 Diamond 中改为容灾状态，此时路由服务在返回归属机房时，就会自动返回灾备机房，如图 5-8 所示。

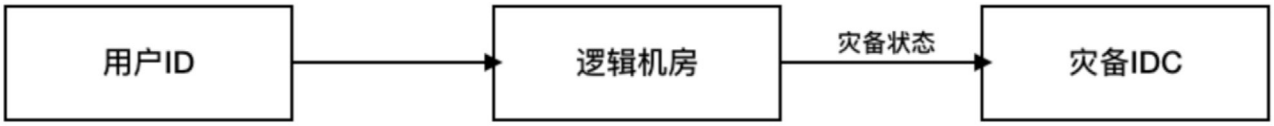


图 5-8 异地容灾切流示意

而基于路由服务对容灾的处理结果，统一接入层路由就可以实现多机房的容灾，这个容灾生效的时间就是机房中 Diamond 配置的生效时间，可以认为是准实时的，不需要等待 DNS 完全生效。而在 DNS 的整个生效过程中，DNS 未生效的那部分用户会仍然接入到原机房中，通过统一接入层路由再访问到备份机房。如图 5-9 所示。

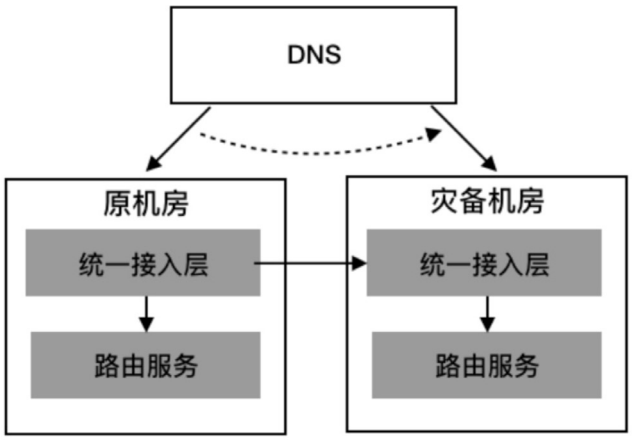


图 5-9 异地容灾机房切流

细心的读者可能会发现，原机房已经出现问题了，统一接入层还能路由吗？事实上我们遇到的大部分场景的问题都发生在统一接入层以下，而统一接入层都没有问题，因此这条链路是有效的。当然，如果故障已经使得原机房的统一接入层也不能正常工作，则此方案只是不执行，并不会带来危害，这个时间的故障恢复时间仍然要依赖到 DNS 在经过 TTL 后的恢复时间。

解决了容灾恢复时间的问题，剩下的问题就是一致性问题了。这里再强调一下，没有既能保证数据一致性，又能绝对快，没有任何损失的一种方案，这永远是一个权衡的结果，但我们有勇气说，我们在一致性的基础上，最大可能的实现了容灾快速切换。如何解决一致性问题呢？如图 5-10 所示。

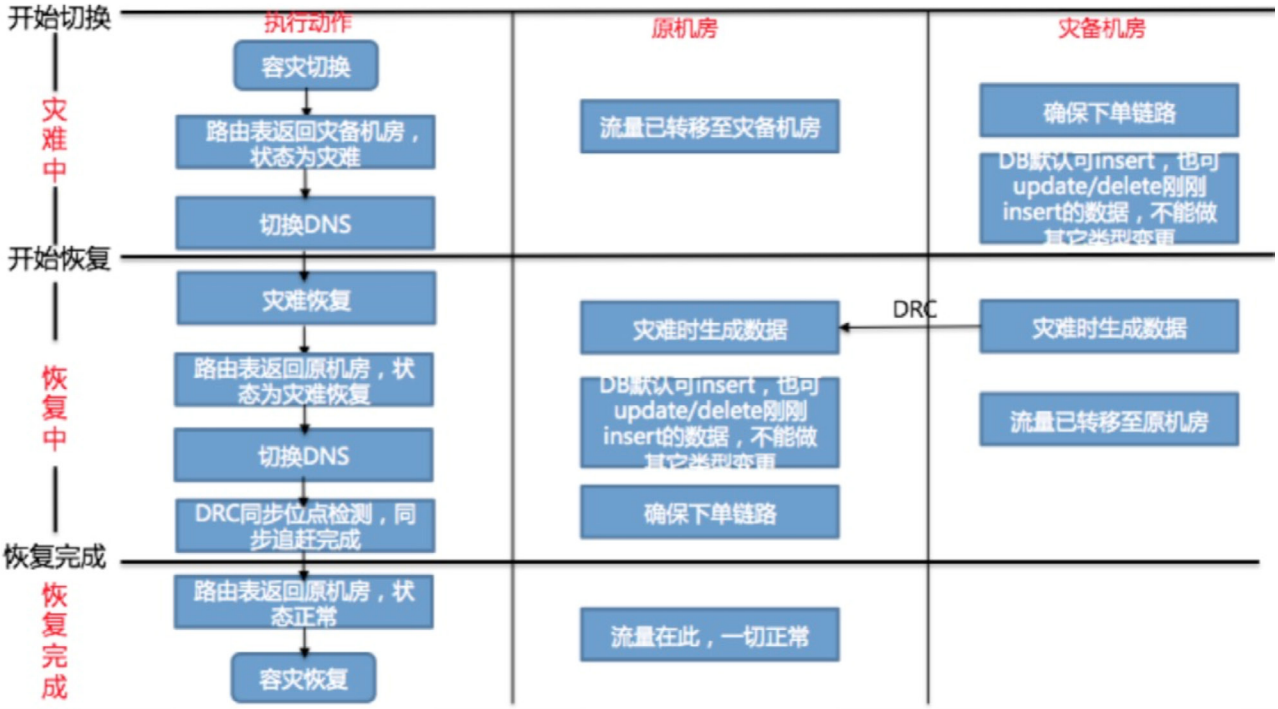


图 5-10 容灾切流一致性

1）开始切换，先将路由表对应的逻辑机房状态改为灾备状态，再进行 DNS 切换（其实是通过网络层路由解决，后续章节介绍，简单理解为 DNS 切换），将原机房的请求解析到灾备机房；此时原机房所有的流量中一部分是通过 DNS 已经切换到灾备机房中，另一部分 DNS 仍解析至原机房，但统一接入层路由会将其重新路由到灾备机房；而此时灾备机房中，通过 TDDL 层的禁写控制，对应的用户只可以执行 insert 操作，也可以 update 或 delete 切换后新生成的数据，但对于切换之前产生的数据不能做任何变更，从而保证了单一数据 Master 原则。为了确保一致性，牺牲了一定的用户体验，但用户仍然可以完成最关键的业务动作即下单。

2）开始恢复，灾难总有结束的一天，当原机房问题已经修复时，还需要将原用户切回。此时进行的操作就是用户对应的归属逻辑机房状态改为灾难恢复，并切换 DNS，再开启数据复制的位点检查，直到数据复制完成的时间点已经超过容灾恢复的时间点，再进行下一步操作。此时部分用户已经通过 DNS 切换回了原机房，剩余用户通过统一接入层路由回到原机房。对应的用户仍然只可以执行 insert 操作，也可以 update 或 delete 切换后新生成的数据，但对于切换为灾难恢复的动作时间点之前产生的数据不能做任何变更。用户下单等关键操作仍然可以进行。同时，DRC 正在将灾备机房的数据复制回原机房。

3）恢复完成，当数据复制时间点已经超过灾难恢复时间点时，就进入了恢复完成状态，将用户于路由表中对应的原机房状态改为正常状态，所有流量恢复到原机房，一切又恢复到正常状态。

仅限内部传阅

5.5.4 网络容灾

对于动态请求，全球网络复杂，之前 PC 端依赖的 akamai 网络链路出现了问题，导致交易下跌，引起可用性故障，但此时如果切换到互联网又会引起极大的性能问题，此时我们需要更多的网络链路可以选择，POP 点加专线可以作为一种候选链路。对于静态资源，现阶段强依赖 akamai，akamai 的边缘节点出现问题，导致部分国家图片下载失败率严重下降，此时我们需要更多的边缘节点。接入越多的 CDN 和海外 POP 点，可以获得更多潜在的网络节点和链路，合理的使用以更低的成本可以获得性能的提升和稳定性保障。集团内的比如网络团队和 ALICDN 团队在海外逐渐发力，逐渐减少外部依赖，可以获得更多集团内的特性的使用。

在国际化的实践中，我们基于 PC 和无线分别进行探测，并基于大数据分析技术，决策 PC 和无线分别最优的网络链路是哪一个，如图 5-11 所示。通过 DNS 技术对 PC 网络链路进行调度的执行，通过 HTTPDNS 和 PUSH 技术的结合，实现无线网络链路的调度执行。最终归属到同一个 IDC。PC 和无线之所以分开，原因是 PC 和无线网络本身具体差异性，而从整体效果来说，其实技术本身并不关心，只关心用户端的性能和稳定性。PC 和无线最终都将归属到同一个 IDC，主要是用户归属必须一致的原因，具体可以参考区域化部署章节。

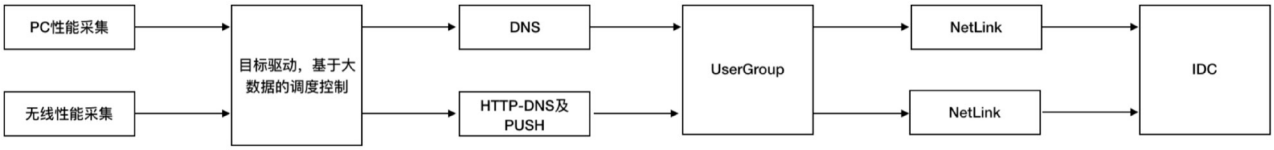


图 5-11 基于大数据的网络调度

而这个调度最终不仅仅输出网络链路，也会输出用户最终归属的机房，如图 5-12 所示。

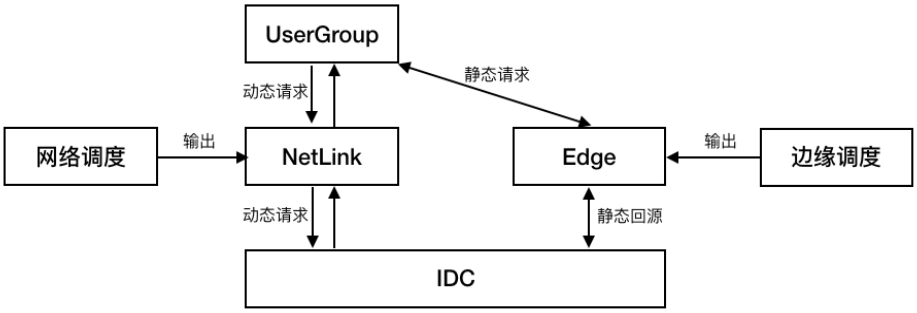


图 5-12 网络容灾的用户机房调度

5.5.5 细粒度自动化容灾

容灾的更进一步的切流粒度可以通过单域名或者单服务层面，图 5-13 展示的基于变更、机房、网络、容量、演练等多个层面进行的容灾操作，其中单域名单元间容灾通过 DNS 切换或者 vipserver 更改进行路由。

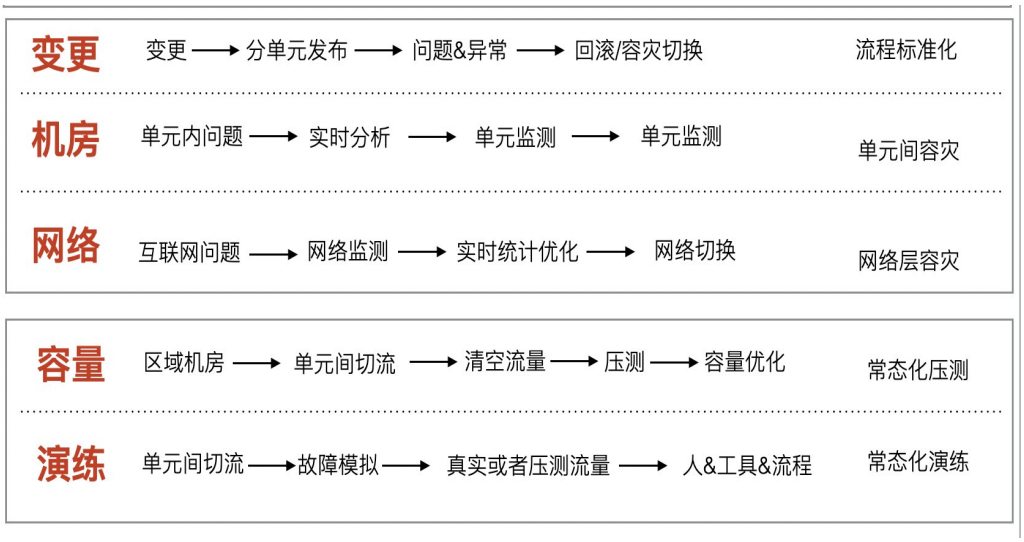


图 5-13 多层次细粒度容灾

仅限内部传阅

六、上云

6.1 国际化上云背景

6.1.1 什么是上云

“上云”是指企业以云计算的服务能力为基础，进行信息化系统以及相关业务应用的建设并对外连接的过程。云计算正如电力这样的公共服务基础设施，可以提供时刻在线的服务能力。上云可以有多种好处，上云可以协助企业解决技术统一、业务统一的问题；可以支持持敏捷开发，加速业务快速创新；同时可以帮助企业降本增效。比如，计算资源池化与虚拟化技术，让应用和底层硬件资源解耦。一方面提升资源利用率，降低计算成本；一方面提升了资源的弹性供给，弹性成为差异化云基础设施和传统 IDC 的关键能力。

企业上云之路一般可分为三个类型：

- **新托管（Rehost）**：简单地通过 lift-and-shfit 方式，将线下物理机替换成为云上虚拟机或者裸金属实例，不改变原有的运维方式。
- **新平台（Re-platform）**： 利用托管的云服务替换线下自建应用基础设施，比如通过 RDS 数据库服务替换自建 MySQL，通过容器服务，比如阿里云 ACK 来取代自建 K8s 集群。托管的云服务通常提供更好的弹性、稳定性和自治运维能力，可以让用户关注于应用而非基础设施管理。
- **重构 / 新架构（Refactor/Re-architect）**：包括单体应用的微服务架构改造，应用的容器化和 Serverless 化整体开发和交付架构改造。

从 Rehost, Re-platform 到 Re-architect，上云的成本和复杂性逐步增加，但是在其他方面，比如敏捷性、弹性、可用性、容错性等收益也在持续增加，所以企业上云需要综合考虑自身的业务和技术特点，选择适合自己的上云节奏。

6.1.2 集团上云背景

如今阿里巴巴集团已经实现了核心系统 100% 上云，这个过程实际上并不是一帆风顺，面临着很多业务和技术方面的挑战，比如极致能力要求，超大规模数据中心网络迁移到云网络，以及大规模云化存储计算分离。回顾集团上云之路的整个过程，如图 6-1 所示，主要分为三个阶段：

- 在 2015 年之前未上云，全部采用内部的基础设施。
- 2015 开始，双 11 期间单元化的交易应用开始通过弹性使用云资源，实现成本节约的目标。通过弹性云化，以及离在线混部等能力，使大促成本持续下降。

仅限内部传阅

- 又经过一年的努力，在 2019 年双 11 前，集团实现了核心系统的全面上云，包括核心电商的中心和单元业务、数据库、中间件等组件。集团的技术栈和云产品融合，应用通过神龙服务器 + 容器和 K8s 上云，数据库接入 PolarDB，中间件采用云中间件和消息产品，负载均衡采用云 SLB 产品等。

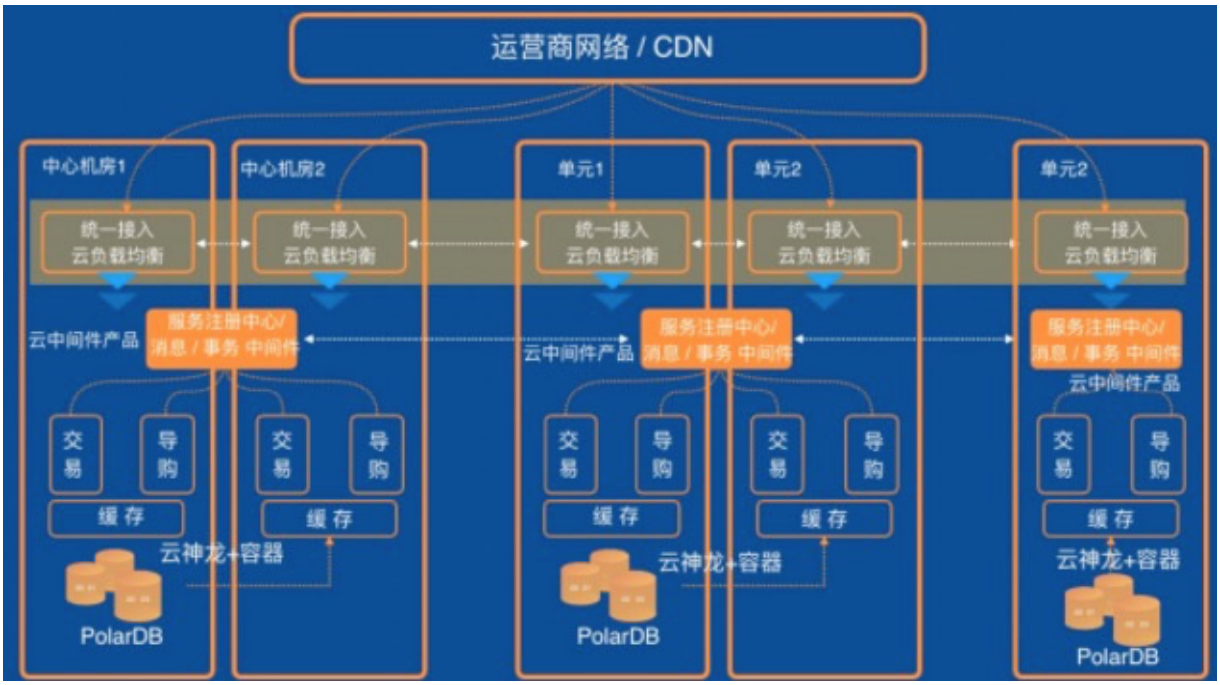


图 6-1 集团核心系统全面上云

6.1.3 国际化上云背景

业务应用	业务服务	用户	商品	活动
		交易	支付	店铺
基础架构	基础服务	中间件	数据库	缓存
		大数据	搜索引擎	文件系统
	基础软件设施	虚拟化	局域网	LVS
	基础物理设施	交换机	路由器	物理机
		机房	机架位	阿里骨干传输网

图 6-2 国际化上云前自建基础设施

国际化上云在集团上云的大背景下，在多变的国际业务背景下，也面临这非常多的技术挑战。在国际化上云之前，我们主要是采用了自建基础设施 IDC，如图 6-2 所示，面临着非常多的痛点：

- **成本高周期长**：AE 服务于全球 200+ 国家和区域，Lazada 服务于东南亚 6 个国家，同时还需要考虑全球多机房部署来实现容灾，如果每个机房都需要自建，成本会很高，周期也非常长，比如一个 IDC 交付可能需要 6-9 个月时间。
- **运维压力大**：机房新旧不一，老机房基础架构落后难以改造；业务全球分布，部分网络覆盖度

仅限内部传阅

和质量不高；基础设施需要大量的运维工程师，拖过多个平台维护多套环境，多个时区，并且需要 7*24 小时 oncall，自建的 IDC 运维成本很高。

- **无序的产品和版本**：不同的 IDC 内部使用的基础设施、中间件、数据库等产品使用不同的产品，多套环境版本不一、代码不一；并有同类功能产品重叠，新老交替问题；导致开发和适配成本增加。
- **资源碎片化**：全球 6 大区域，物理机房 16 个，逻辑机房 20 个，由于海外机房规模较小，资源碎片化，机位触顶后无法扩容，无法享受大单元实例。不同的 IDC 缺少统一的成本管理，很难准确预算。
- **业务复杂，多单元混合**：服务于东南亚 10 国的 lazada 业务，都在新加坡两个机房支撑；再加上支撑国际业务的其他 BU，最终应用众多，单机房资源紧张，比如一个逻辑机房存在 13 单元混合部署情况，业务复杂。

6.2 国际化上云应用架构

6.2.1 上云应用架构

应用架构涵盖应用、服务、应用组件、功能组件、服务接口等，过程中涉及到领域驱动设计、服务化、微服务等相关架构设计能力，是整个 IT 架构中关键阶段，决定了企业为客户提供的具体的应用服务功能。

进行云上的应用架构设计时，我们需要充分考虑到云的特点，比如本地部署的应用通常是一个较大的整体，而云上应用会被拆散成若干个小的服务或组件，通过 API 或其他协议通信。云上的各类资源也是分散的，随时可以新增实例进行水平扩展。表 6-1 总结了云上和本地应用架构部分不同点。

对比项	本地应用	云上应用
存在形式	整体	分散的服务和组件
伸缩性	只能应对预期中的负载	伸缩性强，可以应对波动的负载
数据存储	关系型数据库	NoSQL 等各类数据存储混合使用
操作方式	同步处理	分布式、异步处理
故障处理理念	尽量避免故障	通过高可用设计规避故障发生的风险
更新频度	较低，只为重要特性更新	小幅迭代，频繁更新
可管理性	手动 / 自动管理	智能化管理

表 6-1 本地应用 vs 云上应用

6.2.2 上云总体原则

1）选择好架构模式

要充分结合各种架构模式的优缺点，根据具体的应用场景进行选择，比如在全局架构章节提到的 N 层架构、微服务架构、事件驱动架构、大数据架构或高性能计算架构等。

2）单一还是多云

需要结合企业的实际现状，多个供应商对接的成本更高，但如果企业一定要与多个供应商合作，建议

将相对独立的应用模块放在同一个云中，将其他放在另外的云，而不是试图将单个应用模块部署到多个云中。

3）充分评估转化成本

选择云供应商或者云服务时，需要充分评估转化成本，不同的云以及云产品在功能、性能、成本、可靠性等多方面都有差异，要综合评估。

4）重视非功能性

一个良好设计的上云架构在满足功能需求的同时，需要充分考虑非功能性需求，尽可能利用云服务来简化这些非功能性的实现，比如性能效率、可靠性、兼容性、可移植性、可维护性等：

- **性能效率**：包括响应时间、时延、资源利用率、系统容量、可伸缩性，过程中可以使用更合适的云上数据库产品、使用 CDN 提速或通过缓冲队列 / 缓存来消除性能瓶颈等。
- **可靠性**：包括可用性、容错性和可恢复性，我们在高可用章节介绍了很多提高可靠性的方法，这里建议选择相关的云产品来辅助应用程序设置自我恢复机制、配备容灾系统、瞬态故障处理机制等来提升应用整体可靠性。
- **兼容性**：应用需要考虑应用的共存能力（例如向上或向下兼容）、互操作性（例如不同的系统之间互相下达指令）、可改进性（例如敏捷的应用开发运维模式）、可扩展性（例如服务编排的扩展、应用功能的扩展、数据字段的扩展）。
- **可移植性**：可以在不同的环境下运行，包括可适配性、可安装性、可替代性。为了尽快完成安装部署和上线，建议尽可能使用云上托管的服务（如 IaaS、PaaS 服务），来简化可移植性的达成。
- **可维护性**：模包括块化程度、可管理性、可监控性、可分析性和可测试性。云产品例如 APM 产品可以提供给应用从资源、应用到业务层面，提供了全栈式的性能监控和端到端的全链路追踪诊断能力。

6.2.3 核心系统全面上云

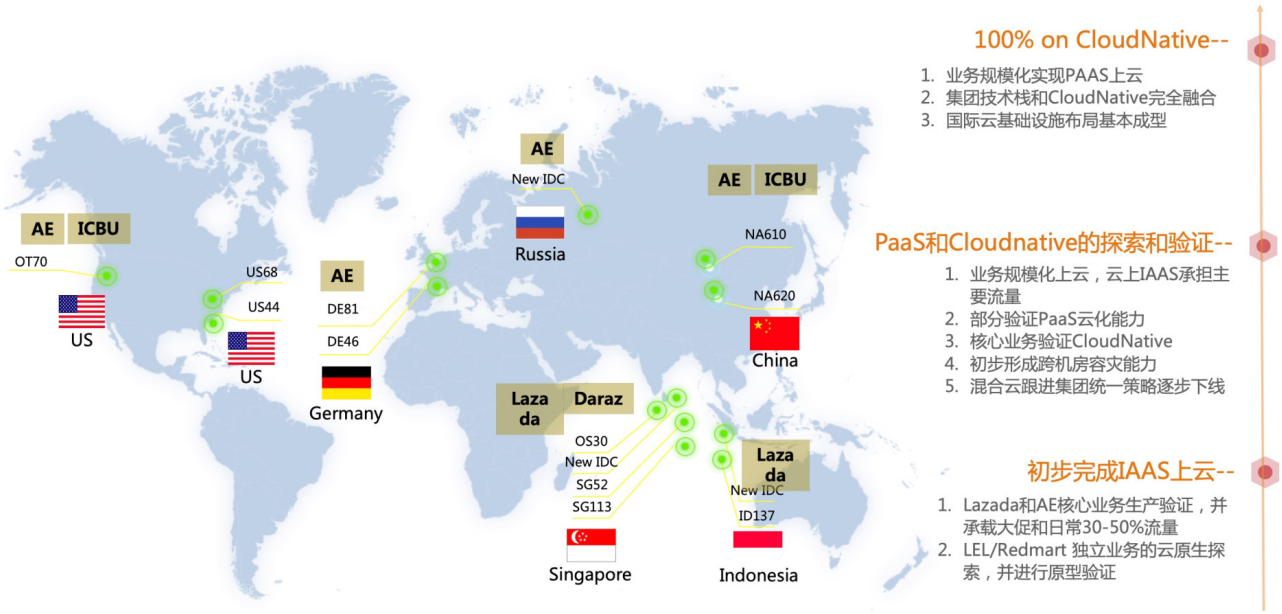


图 6-3 国际上云演进路线

仅限内部传阅

国际化中台的上云路线在集团全面上云的基础上，经过了几年的发展，也实现了核心系统全面上云，同时不只是将机器搬到云上，更重要的是通过平台迁移 Replatforming 以及云应用重构 Refactoring，大体上经历了如图 6-3 所示的三个阶段，目前国际化正在从第二阶段向第三阶段稳步迈进：

- **初步完成 IaaS 上云**：Lazada 和 AE 核心业务生产验证，并承载大促和日常 50% 的流量，一些独立业务比如 LEL/Redmat 进行探索云原生的探索。
- **PaaS 和云原生的探索**：业务规模化上云，云上 IaaS 承载主要流量，验证 PaaS 云化能力，核心业务验证云原生，混合云跟进集团统一策略逐步下线。
- **全面拥抱云原生**：业务规模化上云，技术栈与云原生基础设施完全融合，国际化云基础设施布局全面成型。

6.2.4 国际化上云的价值

国际化上云的过程中，我们收到了很多的上云带来的红利，主要有如下几点：

IDC 升级云机房：核心系统全面上云

目前完成国际化 6 大区域 12 个 IDC 基础设施升级到云机房，如图 6-4 所示，充分利用云的弹性，降低成本，过程中我们使用了 ASI 基础设施、IDC 神龙裸金属服务器和网络、PolarDB 数据库、DMS 数据管理、OSS 对象存储、RocketMQ 分布式消息、ARMS 应用实时监控服务、AHAS 限流管控、SchedulerX 分布式任务调度、CDN 内容分发网络、以及大量的中间件、缓存、搜索引擎相关的基础云服务产品，支撑用户、商品、活动、交易、支付、店铺等核心业务，最终实现了国际化核心系统全面上云。

图 6-4 国际化 IDC 升级云机房

容灾架构升级：提升可靠性

上云的过程中，我们也逐步完善国际化系统容灾架构体系建设，做到系统高可用，比如区域化容灾、同城容灾、单元内容灾，这部分可以具体参考容灾和高可用主题。图 6-5 展示了单元内容灾的场景，更细粒度的同城容灾策略，当某一个机房的某个应用服务出现异常，受影响的入口流量立马切换到另外一个机房。

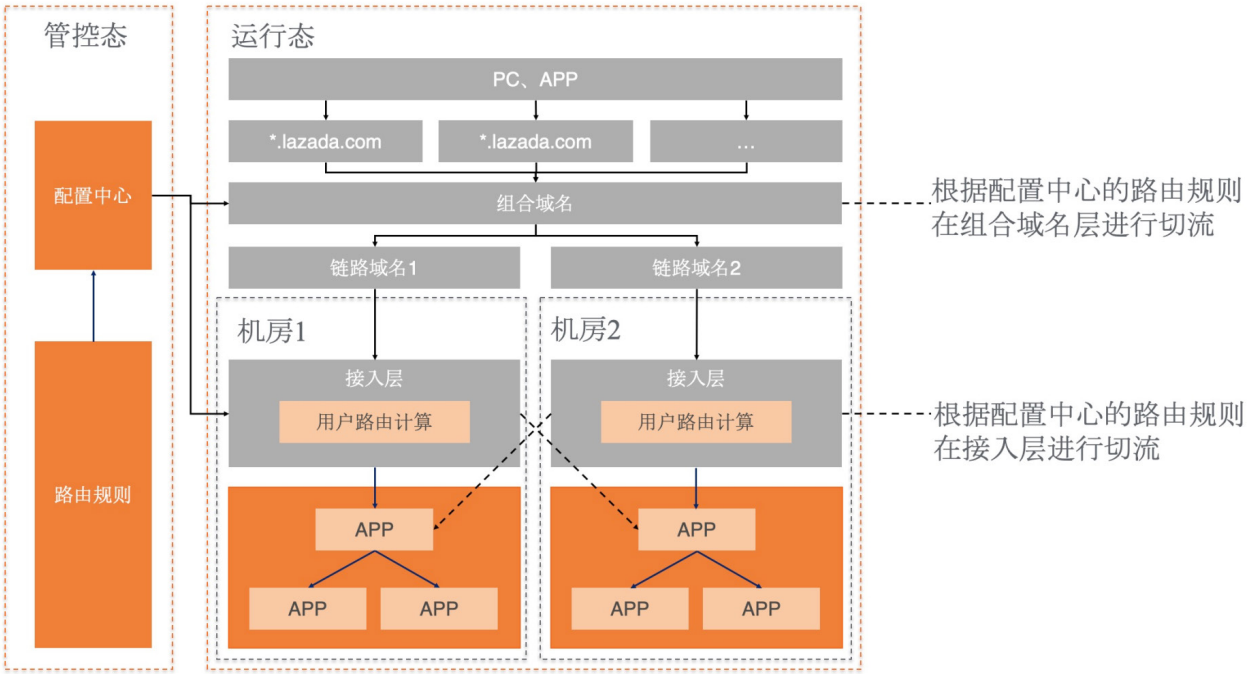


图 6-5 国际化容灾架构升级 - 单元内容灾

容器化重构：提升交付效率

国际化上云过程中，很关键的一点是从虚拟化到容器化的升级，大大提升了国际化整体的交付效率，特别是基于 ASI 的容器化云原生统一基础设施管理。ASI 是 Alibaba Serverless infrastructure 的缩写，是阿里巴巴针对云原生应用设计的统一基础设施，图 6-6 展示了 ASI 的产品能力示意图，国际化容器化重构的过程中得到了 ASI 和 Normandy 团队的大力支持，从 19 年国际化开始接触 ASI 技术，到新加坡 OS30 机房试点，再到当前德国、俄罗斯、美东、印尼云上机房应用 ASI 化基本 100%，双十一国际化交易导购链路的应用几乎 100% 跑在 ASI 上。

图 6-6 ASI 能力示意图

仅限内部传阅

敏捷开发升级：提升研发效能

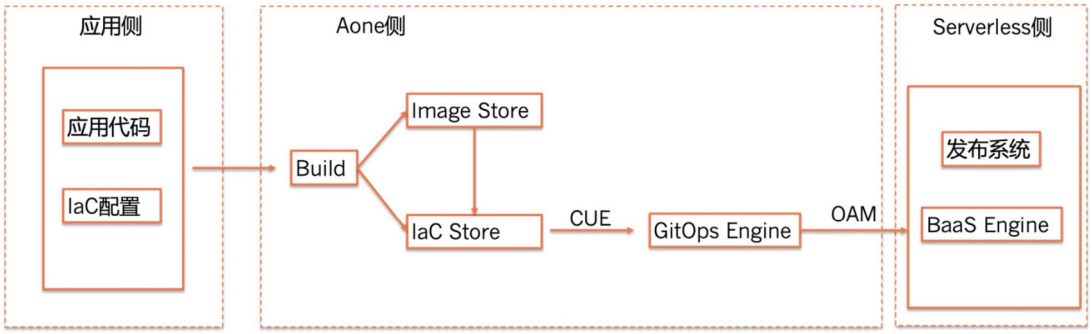


图 6-7 云上敏捷开发运维

国际化上云，从应用架构层面也进行了重构，这里主要强调下云原生带来的研发效能的提升。上云通过云产品的标准使用带来了新的研发模式，进而推进了研发过程中规范化；同时上云也是我们更加优化了研发的组织和配合关系，比如大促的标准化管理运作机制。另外经过更加敏捷的开发模式升级平台架构模式，实现业务和平台迭代更加灵活，业务与业务之间彻底隔离的研发模式；如图 6-7 所示，通过 IaC+GitOps+OAM 的应用，完成了应用和微服务的治理，使得研发、运维效率更高。更多云原生相关内容会在云原生专题进一步展开。

6.3 应用上云方法参考

企业应用上云需要紧密结合自身的业务和技术特点，国际化上云采用了通用的企业上云方法，主要包括如下几个步骤：

① 确认上云需求

首先需要收集和确认应用上云的需求。需求可以通过场景用例的方式展开，需要说清楚通过上云可以达到哪些业务或者技术的效果。过程中可能需要与不同的同学交流和讨论，包括业务方、平台方，也可能涉及研发、运维、产品、测试等多种角色，收集各方的痛点。从上云的视角来看，需要重点考虑工作负载（Workload 对高可用及容灾的需求）、业务增长（如大促峰值的增长）、成本管理（如资源消耗、Quota 等）、时间目标（如恢复时间目标 RTO 等）、安全合规（如数据隐私等）。

② 考虑架构原则

确认了需求之后，需要综合考虑架构模式和原则，明确上云的非功能需求，比如在全局架构中介绍的多种架构模式，以及确定是使用单一还是多云提供商，同时评估上云的转化成本和风险，并重点对性能效率、可靠性、兼容性、可移植性、可维护性等原则进行考虑，确定上云的核心架构原则与相应指标，具体可以参考本章中“上云总体原则”部分。

③ 选择云上服务

使用标准云产品是上云的关键路径，可以帮助我们全面解耦应用与云资源，充分利用云计算带给我们

的技术红利，让云服务来尽最大可能地解决企业非功能性需求。目前云上服务有非常多，比如从基础设施、接入层、中间件、消息、数据存储、大数据、安全、产品研发等多个方面，具体在国际化中采用的云产品请参考本章“上云产品选型”部分。

④ 规划上云方案

进而可以基于需求、架构原则、以及云服务选型进而规划整体的上云方案，建议从全局视角进行总体规划，并将核心的架构原则及云服务选型进行充分阐述，并规划出具体的分步实施计划，并通过公司高层如 CTO/CIO 及架构委员会的评审。

⑤ 实施上云项目

通过项目或者战役的方式进行推进，包括上云的项目管理、组织流程、质量保障、风险识别等，特别是相关的进度和上云前的各种压测，以及一些上云过程中预案处理。同时从国际化角度，还需要考虑国际化的全球部署、区域化部署、本地化、容灾、稳定性等多方面，可以参考白皮书相应的主题。

6.4 上云产品选型

结合国际化的业务和技术特点，我们在上云过程中使用了很多云上产品，包括从基础设施、接入层、中间件、消息、数据存储、大数据、安全、产品研发等多个方面，下面进一步介绍当前典型产品背景和选型。

6.4.1 基础设施

国际化上云的基础设施选型中，核心出发点是需要应对前文提到的国际化自建 IDC 基础设施的痛点：成本高周期长、运维压力大、容器化重构、资源碎片化等。同时需要充分考虑国际化的实际情况，并与国际化云原生的规划方向保持一致。

基于此，我们选择了 ASI 作为国际化基础设施底座。ASI 是 Alibaba Serverless infrastructure 的缩写，是阿里巴巴针对云原生应用设计的统一基础设施。ASI 对应的是标准云产品 Dedicated K8S，加上集团自有的节点管理运维能力、规模化运维等扩展，对集团内部业务方提供 Serverless infra 的基础平台。ASI 实现节点运行环境全托管的标准 K8s 基础设施，实现资源和节点运行环境解耦：在标准 K8S 的基础上，ASI 通过全托管节点计算资源，使业务无须管理服务器节点，降低运维成本。

国际化上云 ASI 的过程中，主要经历了如下几个阶段：

- 我们首先规划国际化的云原生路线需要将自身的 Sigma2.0 升级到 ASI。而国际这边的新项目较多，机房数量相对国内多一些，可以在部分机房部分应用做试点（如新加坡 OS30 机房）。同时当时正处于国际化中台的 Polaris 项目，涉及到应用架构的升级，时机非常合适。
- 过程中，经过 ASI 和 Normandy 的同学的大力支持，加上运行态很稳定和后续 Apollo 项目的

仅限内部传阅

加码，逐步为国际其他机房 ASI 化打下了稳固的基础。

- 截止到目前，德国、俄罗斯、美东、印尼云上机房应用 ASI 化基本 100%，达到了双十一国际化交易导购链路的应用几乎 100% 跑在 ASI 上。

6.4.2 数据存储

数据库主要分为四类，也是云数据库厂商发力的四个方向，即 OLTP 传统的关系型数据库、NoSQL 非结构化或半结构化数据库、OLAP 在线实时分析型数据库、以及数据库相关的服务和管理类工具。

在国际化的云上数据存储选型过程中，我们需要综合考虑相关功能、性能、可维护性、可靠性、安全成本等角度，特别是如下与国际化密切相关的几个方面：

- **业务场景**：数据库需要根据国家隔离，并提供相应的权限隔离能力，同时需要根据合规要求进行拆库
- **研发成本**：需要提供区域化部署能力，成本相关的如资源申请成本、研发变更成本、问题排查成本等需要足够低
- **运维容灾**：在压测、大促、预热的分站点运维压力低，工具支持复杂度低，需要具备机房级别故障等容灾能力

阿里云在数据管理及数据分析领域都拥有深厚的技术积累，在最新公布的 Gartner2020 年度全球数据库魔力象限中，阿里云首次挺进全球数据库第一阵营领导者 Leaders 象限。基于上面的需求，当前国际化上云主要使用了如下阿里云数据库产品：

- **PolarDB 数据库**：阿里巴巴自主研发的关系型分布式云原生数据库，兼容 MySQL、PostgreSQL、Oracle 语法。PolarDB 既解决了传统数据库容量有限、扩缩容时间长等问题，又提供了分钟级扩容、弹性变配、超高并发等能力。计算能力最高可扩展至 1000 核以上，存储容量最 高可达 100T，今年双 11，PolarDB 再度刷新了数据库处理峰值纪录，高达 1.4 亿次 / 秒。
- **DMS 数据管理**：可视化界面实现混合云数据库统一管理、细粒度安全管控、数据库变更业务稳定可回溯，可助力 25 种常见数据库的管理研发，支持数万人每天对海量数据库表进行大量的访问与变更。
- **OSS 对象存储**：海量、安全、低成本、高可靠的云存储服务，提供 12 个 9 的数据持久性。使用 RESTful API 可以在互联网任何位置存储和访问，容量和处理能力弹性扩展，多种存储类型供选择全面优化存储成本。

6.4.3 中间件

国际化在中间件方面，主要面临着上文的几个核心痛点：

- 各个中间件产品部署方式不一，有些产品有 4 种部署模式，无标准化方案
- 同类功能产品重叠，新老交替不彻底，

- 在海外环境复杂度远高于国内，维护难度很高

基于此，当前国际化上云逐步在使用阿里云的中间件产品，目前主要使用了如下产品，未来也在规划更加全面的在微服务治理、动态配置、全链路压测等多方面使用更多的中间件产品：

- **ARMS 应用实时监控服务**：一款应用性能管理 APM 产品，包含前端监控，应用监控和 Prometheus 监控三大功能，涵盖了浏览器，小程序，APP，分布式应用和容器环境等性能管理，帮助实现全栈式的性能监控和端到端的全链路追踪诊断，简化应用运维
- **AHAS 限流管控**：提高应用及业务高可用的工具平台，提供应用架构探测感知，故障注入式高可用能力评测和流控降级高可用防护三大核心能力，通过各自的工具模块可以快速低成本的在营销活动场景、业务核心场景全面提升业务稳定性和韧性。
- **SchedulerX 分布式任务调度**：阿里自研的基于 Akka 架构的新一代分布式任务调度平台，提供定时、任务编排、分布式跑批等功能，具有高可靠、海量任务、秒级调度及可运维等能力。

6.4.4 消息

消息服务也是云服务领域很重要的 PaaS 服务，应对异步通信、解耦、削峰填谷等场景。消息服务提供一种 BaaS 化的消息使用模式，消息服务也是集团支持海量并发的核心，特别在历届大促活动中承担着关键作用。在国际化上云的过程中，在消息服务角度面临的痛点与前文类似。我们逐步统一消息服务的产品，比如将集团自建的 MetaQ 迁移升级到阿里云的 RocketMQ 中，后续也将持续使用其他类型的消息服务，这里介绍下 RocketMQ 的产品能力：

- **RocketMQ 分布式消息**：基于 Apache RocketMQ 构建的低延迟、高并发、高可用、高可靠的分布式消息中间件。最初由阿里巴巴自研并捐赠给 Apache 基金会，服务于阿里集团 13 年，覆盖全集团所有业务。作为双十一交易核心链路的官方指定产品，支撑千万级并发、万亿级数据洪峰，历年刷新全球最大的交易消息流转记录。

6.4.5 大数据

国际化在上云的过程中，在大数据层面主要面临这如下痛点：

- 产品多样性带来的部署复杂度，不同机房使用的大数据产品和版本各不相同
- 大数据资源分配缺乏统一的规划，包括短期和长远规划
- 平台运维缺乏云产品级别的故障处理和相应能力

基于此，我们逐步使用了阿里云相关的大数据产品能力，目前使用的产品如下：

- **Maxcompute 大数据计算服务**：前身是 ODPS，是面向分析的大数据计算服务，以 Serverless 架构提供快速、全托管的在线数据仓库服务，消除传统数据平台在资源扩展性和弹性方面的限制，最小化用户运维投入，经济高效的分析处理海量数据
- **Blink 实时计算**：基于 Apache Flink 构建的企业级、高性能的实时大数据处理系统，由

仅限内部传阅

Apache Flink 创始团队官方出品，拥有全球统一商业化品牌，完全兼容开源 Flink API，提供丰富的企业级增值功能。

- **Dataworks 大数据开发治理平台**：基于 MaxCompute/EMR/MC-Hologres 等大数据计算引擎，提供专业高效、安全可靠的一站式大数据开发与治理平台，自带阿里巴巴数据中台与数据治理最佳实践，承担集团 99% 数据业务构建。

6.5 国际化 ASI 上云案例

接下来，我们进一步展开国际化基于 ASI 上云的过程。整个国际化 ASI 落地计划中，我们划分为两个阶段，先实现基础架构的 ASI 升级，并逐步滚动完成应用粒度的 ASI 升级。

6.5.1 容器资源 ASI 化

对于国际基础设施资源的特点，我们分三个方面去落地国际基础架构的 ASI 升级计划：

云机房 Sigma 滚动升级

我们一开始使用这种比较传统但相对节约成本方式推进 ASI 化。以我们现有弹性容量管理能力，通过我们银河平台算法计算应用日常资源使用量，对头部资源大户弹性扩缩容，实现手术刀般的资源腾挪。国际业务大促频繁，在大促结束之后我们可以用非常有效率的方式，腾挪出大促资源恢复日常态用于资源流转。这种普遍以应用粒度去推进，涉及大量的应用扩容缩容，并且对应用缩容过程中的容器类型还需要设定优先级，优先缩容 Pouch 类型容器，通过频繁置换达到应用最终 100%POD 化。

新建云机房跨机房上云

在俄罗斯以及美国区域，我们同时进行两个上云项目实现机房级 ASI 化

- **俄罗斯 RU151 同城云机房上线**：在俄罗斯新建云机房，在同城双机房实现业务上云
- **美西迁移美东云机房**：在美东新建云机房，业务从美西通过区域化调度，实现跨区域上云

我们在云机房建站阶段直接实现云机房 ASI 化，业务上云直接升级到 ASI 平台，并在上云切流过程逐步对业务实现验证。这种方式相对成本最高，整体项目建设周期也很长，耦合整体上云机房切流能力，但是对基础架构 ASI 化最彻底，新建机房交付大部分都统一交付神龙裸金属物理机资源，相对能有资源规格的保障，确保有更新的资源迭代。

非云机房原地上云

有部分国际物理机房属于非云机房，我们通过隔离多个逻辑区域，对其中逻辑区域改造为云机房达到原地上云。lazada 业务所在印尼 ID35 机房目前属于原地上云方式落地，原地上云对于业务升级 ASI 来说成本是最小的，可以实现无延时感知迁移云机房

6.5.2 国际应用 ASI 化

针对国际化应用方面，主要面临着如下痛点：

- AE 业务经历过微服务拆分改造，很多业务应用拆分出了大量的应用，导致目前国际业务有近 500+ 应用量
- 应用在国际中台化后，根据支持业务单元以及国家划分出很多 AONE 环境分组，比如支持 lazada 业务 12 个单元，AE 业务 4 个国家地区，叠加应用量后产生 7000+ 环境分组需要升级

我们通过平台解决的思路，设定默认路由方案来加以解决：

- 支持按 aone 产品目录树，对应用批量升级路由，支持环境分组的 ASI 路由
- 对机房级别设定默认容器调度路由方案，比如德国 de46/de81 默认以 asi 路由，除非降级为 sigma 路由

进而，经过长期的努力，国际化最终完成了 ASI 的上云过程，如图 6-8 所示：

- **全球化 9 个云上机房基本完成 100%ASI 化，云原生基础设施搭建完成**
  - 覆盖 AE+ 国际化中台 500+ 应用，21000+pod。
  - 基本完成全球 9 个云上机房的应用从 Sigma 集群升级到 ASI 集群

- **全球化海外机房完成核心机房 Cloud Hosting**
  - 俄罗斯 RU151，美东 US44/US68 云机房交付完成业务上云迁移超过 8 万核 15000+POD
  - 印尼 ID35 原地上云，新加坡 SG52 云机房完成业务上云迁移超过 2 万核近 4000POD

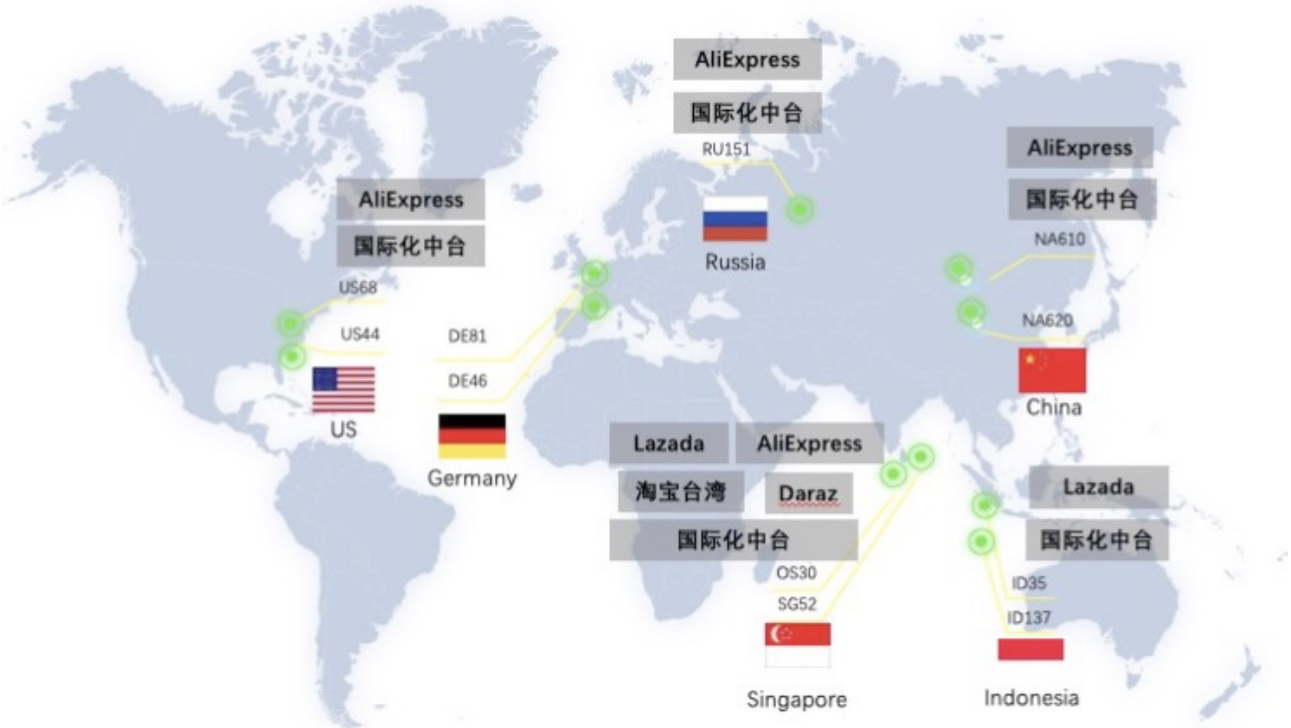


图 6-8 国际化 ASI 上云

仅限内部传阅

6.6 上云展望：云原生基础设施

云计算的下一站是云原生，云原生可以使应用中的非业务代码部分进行最大化的剥离，让开发者聚焦于业务而不是底层技术，让业务应用更加轻量、敏捷、高度自动化。从上云的视角来看，未来企业上云的趋势是**云原生基础设施。**

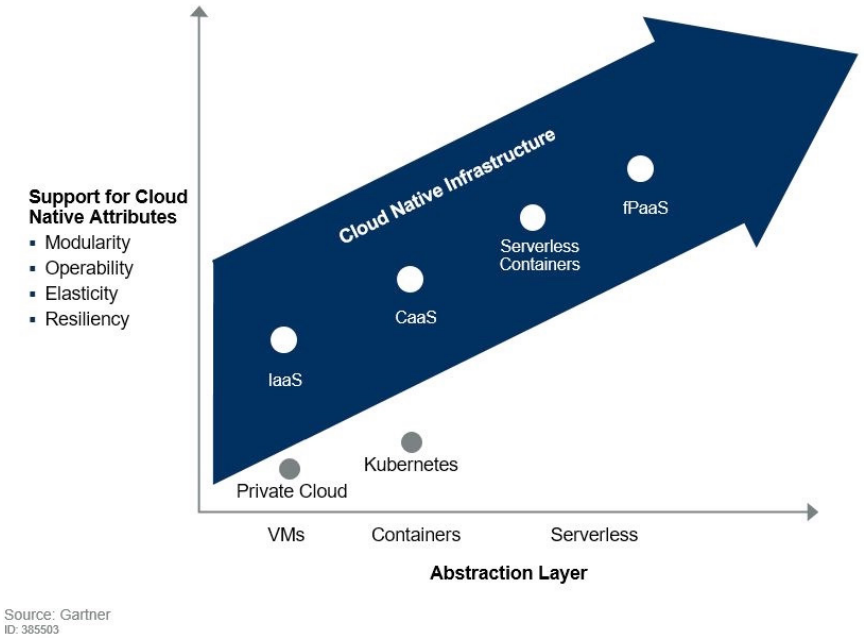


图 6-9 Gartner 对云原生基础设施的分类

Gartner 将云原生基础设施划分成四大类，如图 6-9 所示，主要分为 IaaS、CaaS（Container as a Service）、Serverless 容器（比如 ASI）、FaaS（如函数计算）。可以看到这几种分类，计算单元的粒度越来越细，越来越多体现的云原生的特质，比如模块化程度、自动化运维程度、弹性效率、故障回复能力。可以说，云原生基础设施将应用研发和运维的最佳实践的组合（如容器化、持续集成和交付、不可变基础设施等），使分布式系统更加可靠、易管理和易观测，提升研发运维的效率；同时使云原生的优势和云厂商的传统优势如规模、稳定性和弹性完美结合，并通过开放的社区标准与各大云提供商的商业化服务能力，极大的改变了用户的心智。可以预期，云原生基础设施将大大简化企业上云的过程，并结合新一代计算单元、Serverless 容器、函数计算、分布式云环境等技术的演进更加灵活地适配企业多种上云诉求。

从国际化视角，经过国际化云原生样板间半年时间的尝试，我们有近 10 个核心系统完全升级到云原生架构，技术架构上也将全面拥抱云原生架构，更多细节请参考“云原生”主题。

七、本地化

全球化、国际化和本地化这几个概念听上去很相近，而且的确常常被很多人互换着使用。但它们之间存在着细微的区别。如果要为“全球化”找到一个普遍认同的、明确的定义，那么可能要费点工夫。根据维基百科上的定义，全球化是世界观、产品、概念及其他文化元素的交换，所带来的国际性整合的过程。电信等基础建设的进步，包括电报及之后互联网的兴起，都造成了全球化，以及在文化及经济上互相影响。

在信息技术领域，国际化与本地化（英文：internationalization and localization）是指修改软件使之能适应目标市场的语言、地区差异以及技术需要。根据维基百科的定义，国际化是指在设计软件，将软件与特定语言及地区脱钩的过程。当软件被移植到不同的语言及地区时，软件本身不用做内部工程上的改变或修正。本地化则是指当移植软件时，加上与特定区域设置有关的信息和翻译文件的过程。

国际化和本地化之间的区别虽然微妙，但却很重要。国际化意味着产品有适用于任何地方的“潜力”；本地化则是为了更适合于“特定”地方的使用，而另外增添的特色。用一项产品来说，国际化只需做一次，但本地化则要针对不同的区域各做一次。这两者之间是互补的，并且两者合起来才能让一个系统适用于各地。

在如微软及 IBM 等企业中，则会使用全球化（英文：globalization）来表示此两者的合称。在英文中，也会使用 g11n 做为简称。也有使用缩写 GILT(globalization、internationalization、localization 和 translation)，即“全球化、国际化、本地化和翻译”。

进行本地化时，需要考虑下面几个重要事项：

- 姓名规则（比如，有些文化中可能没有姓氏，而有些文化中可能有多个姓氏）
- 电话号码格式
- 日期和时间格式，例如: 年 / 月 / 日 (YYYY/MM/DD)、日 / 月 / 年 (DD/MM/YYYY)、月 / 日 / 年 (MM/DD/YYYY)
- 币（符号和金额）
- 书写方向（大多数语言是从左向右，但希伯来语和阿拉伯语是从右向左，有些亚洲语言是竖着书写）
- 度量单位，即公制或英制
- 标点符号，例如各语言使用的引号各不相同，在英语中是（“”），在德语中是（„“），在法语中

仅限内部传阅

是 («»)

- 符号和象形图，如勾号、停止标志以及用颜色传达的意思
- 电压、频率和插头
- 法律要求（如，使用欧盟公民的个人数据时应遵守的 GDPR）
- 对于技术人员来讲，可能需要比较多的关注多语言、多时区、多币种这几个方面，接下来会重点介绍。

7.1 多语言

很多人接触到的国际化或者本地化要做的第一件事情就是语言翻译，但是多语言不等于翻译，翻译也不等于本地化。根据某平台的调研数据显示，本地化语言的产品或促进消费者的购买欲望，促成更多的交易 GMV，如图 7-1 所示。

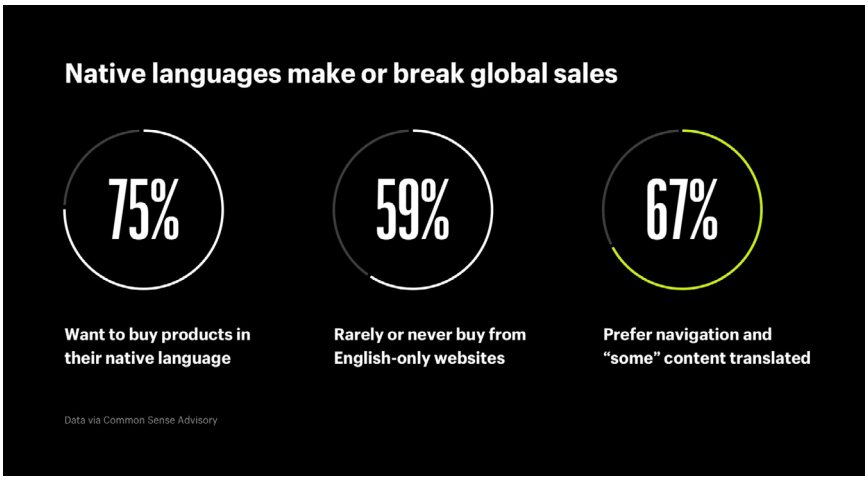


图 7-1 多语言示意

7.1.1 多语言产品设计

从软件工程的角度，产品要出海是要做到软件产品的国际化，用一套代码来适应不同地区、不同语言文化用户的使用需求，而不是针对不同地区语言用户开发不同的代码版本，从而为产品的快速本地化打好基础。

那么问题来了，如何做到软件产品的国际化呢？最基本的做法是在设计中定位需要国际化的部分，在实现中对可国际化的部分进行隔离，在运行时根据地区语言动态加载。这些国际化目标元素包括：

- 数据 (预载数据和用户数据)
- 界面显示文本
- 在线帮助
- 产品文档
- 语言相关的规则，如名词单复数 / 性别变化，动词人称变化等
- 数字格式规则，如小数点，分组，负数，百分号，缩写等

- 时间日期规则，如年月日顺序及分割符，时分秒顺序及分隔符，上下午等
- 日历规则，年、月、日、季节、星期的设置等
- 时区规则，如名称，标识，时间偏移，夏时制等
- 国家 / 地区，名称，标识等
- 地址、邮编人格式
- 电话号码，如国家地区码，格式
- 人名，如姓、名、中间名，格式，缩写
- 字体排版，字形、样式、字重、联合、方向等
- 度量衡，长度、容量、重量单位
- 排序规则，读音符、大小写、拼音、笔画、偏旁部首等

国际化资源隔离是指界面显示文本通常提取到编程框架支持的资源文件中，而在线帮助和产品文档通常需要整体翻译不需要单独提取其中的字符串。国际化运行机制包括：

- 数据的处理中使用 Unicode 字符集
- 识别用户的语言区域设置，可以来自用户的偏好设置，浏览器 / 客户端设备。
- 根据用户语言区域加载相应的语言翻译，包括用于界面显示文本、在线帮助、产品文档、已翻译用户数据或者实时机翻用户数据等。
- 根据用户语言区域格式化数据，包括日期、时间、时区、数字、货币、地址、人名等。

7.1.2 多语言开发规范

- **文本处理**
  - 程序 / 代码内部应只使用 Unicode 字符集进行文本数据的处理。
  - 当支持文本数据的导入导出（如 .csv 文件），程序应该使用 Unicode 相关的编码来处理这些文件。
  - 当程序和系统中的其他部分交换文本数据的时候，应该使用 UTF-8 编码方案。
  - 产品应支持语言学上的排序。比如中文的笔画排序，德文的 DIN 排序。
  - 在文本搜索的时候，程序应该能根据不同的语言来支持不同的分词算法。

- **文案**
  - 应该将资源文件从代码中完全分离出来，并确保文案以字符串的形式存在。
  - 在翻译前，应确保文案的可译性。

- **UI 布局**
  - 切换语言后，用户界面上的文字应该无截断。
  - 切换语言后，用户界面上的文字应该无溢出。
  - 切换语言后，用户界面上的文字应该无重叠。

仅限内部传阅

◦ 切换语言后，用户界面上的文字应该无错位。

7.1.3 多语言引发的故障

- 1103 印尼购物车渲染成功量下跌：美杜莎客户端初始化时获取美杜莎文案失败，而购物车对美杜莎的文案有强依赖，在获取美杜莎文案为 null 的时候，会触发校验报错进而购物车渲染失败，渲染页面异常。
- Lazada 购物车加购成功量下跌：美杜莎客户端拉取数据包，Tair 的缓存服务容量不足从而 Tair 限流，导致客户端拉取数据失败进而影响 PDP 只拉取到部分美杜莎数据（缺少 Add To Cart 文案），最终 PDP 页面展示异常影响购物车“Add To Cart 文案”和“buy now”未透出，购物车成功量下跌。

7.2 多时区

时区是地球上的区域使用同一个时间定义。时区是本地化最基本的一项要求。时区的正确与否，不仅决定了页面中用户体感的好坏，错误的时区甚至有可能会诱导用户做出错误的决策。时区在后端服务中也有着同等的重要性，错误的计算时区、时间差，错误的调度会导致服务运行的混乱，极易造成资损。

以 AliExpress 为例，作为一个服务全球的电商网站，消费者市场主要覆盖了美国、俄罗斯、巴西、西班牙和法国。这些不同的国家有着不同的时区，如图 7-2 所示：

图 7-2 AE 多时区

这些国家不仅有不同的时区，而且时间的本地化表达也是各有不同，如图 7-3 所示：

LANGUAGE	CODE	EXPRESSION
English	en-US	Nov 11, 12:00 a.m. - Nov 12, 12:00 p.m. PST
Russian	ru-RU	11 нояб., 11:00 - 13 нояб., 11:00 MSK
Spanish	es-ES	11 nov. 9:00 - 13 nov. 9:00 (Madrid)
Portuguese	pt-PT	11/11/2019, 8:00:00 da manhã WET - 13/11/2019, 7:59:59 da manhã WET
French	fr-FR	11 nov. à 9:00 - 13 nov. à 9:00 (Paris)
Indonesian	id-ID	11 Nov 2019 3.00.00. PM WIB - 13 Nov 2019 2.59.59. PM WIB
Turkish	tr-TR	11 Kas 2019 ÖÖ 11:00:00 GMT+3 - 13 Kas 2019 ÖÖ 10:59:59 GMT+3
Thai	th-TH	11 พ.ย. 2562 3:00:00 หลังเที่ยง GMT+7 - 13 พ.ย. 2562 2:59:59 หลังเที่ยง GMT+7
Italian	it-IT	11 nov 2019, 9:00:00 AM CET - 13 nov 2019, 8:59:59 AM CET
German	de-DE	11. Nov. 2019, 9:00:00 AM MEZ - 13. Nov. 2019, 8:59:59 AM MEZ
Hebrew	he-HE	11 בנוב׳ 2019, 10:00:00 לפנה״צ GMT+2 - 13 בנוב׳ 2019, 9:59:59 לפנה״צ GMT+2
Japanese	ja-JA	2019年11月11日 午後5:00:00 JST ～ 2019年11月13日 午後4:59:59 JST
Korean	ko-KO	2019년 11월 11일 오후 5시 0분 0초 GMT+9 ~ 2019년 11월 13일 오후 4시 59분 59초 GMT+9
Dutch	nl-NL	11 nov. 2019 9:00:00 a.m. CET - 13 nov. 2019 8:59:59 a.m. CET
Vietnamese	vi-VL	3:00:00 CH GMT+7, 11 thg 11, 2019 - 2:59:59 CH GMT+7, 13 thg 11, 2019
Arabic	ar-AR	ar-AR 11 نوفمبر 2019 11:00:00 ص غرينتش+3 - 13 نوفمبر 2019 10:59:59 ص غرينتش+3
Polish	pl-PL	11 lis 2019, 9:00:00 AM CET - 13 lis 2019, 8:59:59 AM CET
Ukrainian	uk-UK	11 лист. 2019, 10:00:00 дп GMT+2 - 13 лист. 2019, 9:59:59 дп GMT+2

图 7-3 本地化时间表达

7.2.1 基本概念

- **夏时制**（美国及加拿大英语：daylight time）：又称夏令时、日光节约时间（美国及加拿大称为 daylight saving time，简称 DST；英国与其他地区称为 Summer Time），是一种在夏季月份牺牲正常的日出时间，而将时间调快的做法。通常使用夏时制的地区，会在接近春季开始的时候，将时间调快一小时，并在秋季调回正常时间 [1]。实际上，夏时制会造成在春季转换当日的睡眠时间减少一小时，而在秋季转换当日则会多出一小时的睡眠时间 [2][3]。
- **操作系统时区**（/etc/localtime）：虽然我们可以选择 GMT+8 这样的固定时区来设置系统时区，但一般我们都是通过选择以 Area/Location 方式命名的城市时区，比如：Asia/Shanghai 代表的就是中国北京时间，America/Los_Angeles 代表的就是美国洛杉矶时间。为什么会选择以指定城市来命名时区呢？一方面是便于大家理解和设置，还有个因素是这些大城市一般不会像国家名或地区名那样，随着政权的变化而变化，所以对于操作系统来说，以这种城市命名方式来命名时区，即兼顾对用户的友好操作，也尽可能的考虑到历史版本兼容稳定。

7.2.2 多时区开发规范

- **数据存储**：
  - 应该在后端使用 UTC 存储时间数据。产品应有能力部署到非中国时区。此外，为了统一内部处理，所有日期时间数据在程序内部处理时应该使用 UTC 时区。如果不得不用非 UTC 时区，那么必须明确指明相关时区信息。
  - 数据库时间字段禁用 datetime 类型，都要求使用 timestamp 类型。
  - 存储在数据库的时间应该不受 timestamp 的 2038 年限制，应使用 64 位的 bigint，或者

仅限内部传阅

使用 UTC 时区的 datetime。

- **数据展现**：
  - 在前端展示日期，时间相关数据时，使用的是用户的时区。
  - 程序应按需实现对夏时制的支持。一些时区有夏时制（DST）。大部分与此相关的问题可以通过将服务器时区设置为 UTC 解决。

- **对外 API**：
  - 产品各组件之间进行时间数据交换时，应该使用 UTC 时间。
  - 如果 API 接收的参数是时间数据，该时间应为 UTC 时间。
  - 如果 API 的返回值是时间数据，应该有能力接收 UTC 时间。

- **应用设计**：
  - 新建应用容器时区默认为 UTC 时区。
  - 定义日期属性（只包含年月日信息），必须明确时区。推荐使用 String 定义日期属性，不能使用 Date 类型来定义日期属性。
  - 涉及时间相关的转换方法，统一使用 DateUtil 工具类。

7.2.3 多时区引发的故障案例

- 天猫海外自营店会场页面倒计时模块不支持时区，导致展示不正确。活动正式开始后不会自动刷新，用户如果正常刷新，活动可以正常进行。只影响活动时间的展示。
- LZD 活动在多国 (PH/TH) 举行，由于 TH 的机器部署在 SG 机房 (东八区)，正常情况当地到达 0 点时，活动才会被重置清零重新计算，由于代码时区时间获取逻辑存在问题，东八区到达 0 点的时候，TH 站点的机器判断已达到第二天，所以重置了活动，如果 TH 有用户的达到了活动门槛，会导致这个时间差内满足要求的用户有概率重复参与活动成功。

7.3 多币种

多币种问题在本地化中占据着非常重要的位置，不仅仅影响消费者的下单购买意愿以致影响 GMV，也是一个资损和故障问题高发的领域。

来自某平台的调研数据显示，用当地币种展现商品价是为了提升本地用户购买意愿，92.2% 的用户倾向于在展示本地货币的网站上购物，33% 的用户会直接放弃购买以美金标价的商品，如图 7-4 所示。

图 7-4 多币种

币种问题涉及到钱的问题，但是在整个商业活动中，大部分都有涉及到钱的流程。在产品设计、应用设计、数据存储、页面展现、风控等等方面都可能存在信息丢失或者转换错误。

7.3.1 基本概念

- **钱的表述**：钱 = 货币 + 金额
- **币种**：货币的国家属性或者政权属性，不同国家币种不一样不同政权币种可以统一比如欧元，全球目前一共有 278 个币种。
- **单位的精度**：不同的币种，单位精度不一样，单位的精度目前在 ISO4217 里面有 0，2，3，4 四种取值。常用的：人民币 / 美元 / 港币 / 台币支持 2 位小数点（分），新韩元 / 日元不支持小数点（元）。这里用白话来讲就是有些币种最小单位是分，有些币种最小单位是元，还有些币种最小单位是十分之一分。
- **金额同样也需要关注两大属性**：数值和单位，缺一不可。

7.3.2 多币种规范

- **数据存储**
  - 产品存储金额相关数据确保连同币种一起存储。金额数据必须和币种相关联，金额和币种应该同时存储。比如金额数据可以是一个结构，或者，在数据库中有币种这个字段。
  - 对于金额的存储建议按照当前币种的最小单位来存储。

- **数据输入**：用户输入金额时，应该可以选择货币种类。程序应该有能力根据用户的偏好设置来判断默认币种。如果用户选择了默认币种以外的币种，程序也应支持。
- **数据展**现：程序要能够确定金额相应的币种，并显示相关货币符号。程序应该能根据用户的偏好设置来判断币种，来正确判断出相应的货币符号。
- **对外 API**：产品如果提供对外的 API，API 应该能够明确标示钱款相关的币种，并返回相应货币代码。如果程序 API 提供返回货币相关的数据，API 需要返回币种的相关信息。
- **应用设计**：

仅限内部传阅

- 定义统一的数据结构来封装金额、币种、单位等属性。
- 定义统一的工具类来封装计算逻辑， 涉及金额计算的地方统一调用工具类。
- 客户端和前端不做金额的加减乘除的运算，服务端下发的金额直接展示。
- 金额计算禁止使用 float/double，推荐使用 BigDecimal 或金额计算工具类。
- 计算精度舍入规则必须明确（一般舍入规则分 3 种：四舍五入、全部舍、全部入；具体采用哪种舍入规则，由业务 / 产品基于业务特征来决定）。

7.3.3 多币种引发的故障案例

- 因为日元的最小单位是元，美元是分，因为程序 bug 导致日元实际支付的时候比预期扩大了 100 倍。
- 日元、台币没有分，导致优惠价格计算错误。
- 多个系统对币种的处理不一致，导致计算的金额出问题，例如：voyager 期间 vn 站点物流对邮费计算时出现了小数点导致交易链路金额计算错误。
- 全球惠计价配置页面的前端和后端在处理金额单位转换时存在问题，开发误将前端传入的数值乘以 100 后存入数据库，导致兜底金额被扩大了 100 倍 , 造成资损。

【第三篇】
高阶主题

八、云原生

8.1 云原生简介

8.1.1 云原生背景介绍

云原生（Cloud Native）的概念，由来自 Pivotal 的 MattStine 于 2013 年首次提出，被一直延续使用至今。云原生包括 DevOps、持续交付（Continuous Delivery）、微服务（MicroServices）、敏捷基础设施（Agile Infrastructure）和 12 要素（The Twelve-Factor App）等几大主题。采用基于云原生的技术和管理方法，可以更好地把业务生于云，长于云，用好云。

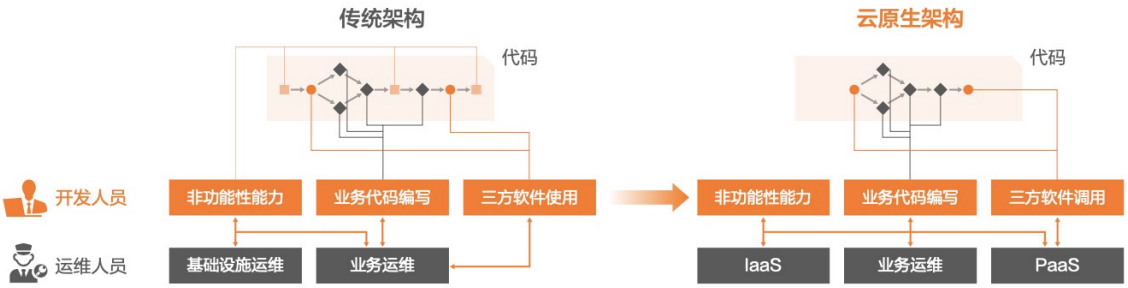
2018 年 CNCF 对云原生进行了重定义：云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。结合可靠的自动化手段，云原生技术可进行频繁并可预测的变更。

随着云原生技术的升级和阿里经济体全面上云，我们国际化的架构设计必然会走向基于云的能力而进行构建应用的时代。在这个过程，我们会实践到：Service Mesh 微服务架构、GitOps 研发模式、Serverless 无服务器基础设施的运维，这必将对研发的体验、交付效率产生深刻的影响。阿里巴巴集团在 2020 年初成立“国际化云原生样板间战役”，通过云原生在国际化团队的试点来为集团全面云原生化做好摸底，从 5 月份的 OK 到现在大半年的时间内，在基础架构域、业务架构域都沉淀出了一套适合阿里架构的标准云原生体系，通过这一体系可以把当前沉淀的技术和方案快速赋能整个集团，我们相信阿里的未来架构就是云原生。

8.1.2 云原生架构定义

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

仅限内部传阅



云原生架构与传统架构的对比

图 8-1 云原生架构与传统架构对比

图 8-1 展示了在代码中通常包括三部分：业务代码、三方软件、处理非功能特性的代码。云原生架构相比较传统架构进了一大步，从业务代码中剥离了大量非功能性特性 (不会是所有，比如易用性还不能剥离) 到 IaaS 和 PaaS 中，从而减少业务代码开发人员的技术关注范围，通过云厂商的专业性提升应用的非功能性能力。此外，具备云原生架构的应用可以最大程度利用云服务和提升软件交付能力，进一步加快软件开发。

8.1.4 云原生架构原则

服务化原则

当代码规模超出小团队的合作范围时，就有必要进行服务化拆分了，包括拆分为微服务架构、小服务 (Mini Service) 架构，通过服务化架构把不同生命周期的模块分离出来，分别进行业务迭代，避免迭代频繁模块被慢速模块拖慢，从而加快整体的进度和稳定性。云原生架构强调使用服务化的目的还在于从架构层面抽象化业务模块之间的关系，标准化服务流量的传输，从而帮助业务模块进行基于服务流量的策略控制和治理。

弹性原则

弹性则是指系统的部署规模可以随着业务量的变化自动伸缩，无须根据事先的容量规划准备固定的硬件和软件资源。在弹性能力的支持下，应用能够更好的体验到云上迅捷、稳定的特性，同时又可以利用各种弹性产品以节省容器等资源的成本。

可观测原则

在云上分布式系统中，主动通过日志、链路跟踪和度量等手段，让一次 APP 点击背后的多次服务调用的耗时、返回值和参数都清晰可见，甚至可以下钻到每次三方软件调用、SQL 请求、节点拓扑、网络响应等，这样的能力可以使运维、开发和业务人员实时掌握软件运行情况，并结合多个维度的数据指标，获得前所未有的关联分析能力，不断对业务健康度和用户体验进行数字化衡量和持续优化。

仅限内部传阅

韧性原则

韧性代表了当软件所依赖的软硬件组件出现各种异常时，软件表现出来的抵御能力，这些异常通常包括硬件故障、硬件资源瓶颈 (如 CPU/ 网卡带宽耗尽)、业务流量超出软件设计能力、影响机房工作的故障和灾难、软件 bug、黑客攻击等对业务不可用带来致命影响的因素。

所有过程自动化原则

通过 IaC(Infrastructure as Code)、GitOps、OAM(Open Application Model)、Kubernetes operator 和大量自动化交付工具在 CI/CD 流水线中的实践，一方面标准化企业内部的软件交付过程，另一方面在标准化的基础上进行自动化，通过配置数据自描述和面向终态的交付过程，让自动化工具理解交付目标和环境差异，实现整个软件交付和运维的自动化。

零信任原则

零信任安全针对传统边界安全架构思想进行了重新评估和审视，并对安全架构思路给出了新建议。其核心思想是，默认情况下不应该信任网络内部和外部的任何人 / 设备 / 系统，需要基于认证和授权重构访问控制的信任基础，诸如 IP 地址、主机、地理位置、所处网络等均不能作为可信的凭证。零信任对访问控制进行了范式上的颠覆，引导安全体系架构从“网络中心化”走向“身份中心化”，其本质诉求是以 身份为中心进行访问控制。

架构持续演进原则

云原生架构本身应该和必须是一个具备持续演进能力的架构，而不是一个封闭式架构。除了增量迭代、目标选取等因素外，还需要考虑组织 (例如架构控制委员会) 层面的架构治理和风险控制，特别是在业务高速迭代情况下的架构、业务、实现平衡关系。

8.1.3 12-Factor 应用原则

12-Factor 应用原则为构建一个优雅的互联网应用，定义了需要遵循的一些基本原则和方法论，被众多微服务应用架构设计广泛参考。我们在第二章“全局架构”中介绍过 12-Factor，这里简要回顾一下：

- **I. 基准代码**：一份基准代码，多份部署
- **II. 依赖**：显式声明依赖关系
- **III. 配置**：在环境中存储配置
- **IV. 后端服务**：把后端服务当作附加资源
- **V. 构建，发布，运行**：严格分离构建和运行
- **VI. 进程**：以一个或多个无状态进程运行应用
- **VII. 端口绑定**：通过端口绑定提供服务
- **VIII. 并发**：通过进程模型进行扩展
- **IX. 易处理**：快速启动和优雅终止可最大化健壮性
- **X. 开发环境与线上环境等价**：尽可能的保持开发、预发布、线上环境相同

- **XI. 日志**：把日志当作事件流
- **XII. 管理进程**：后台管理任务当作一次性进程运行

8.2 云原生核心技术

8.2.1 容器

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

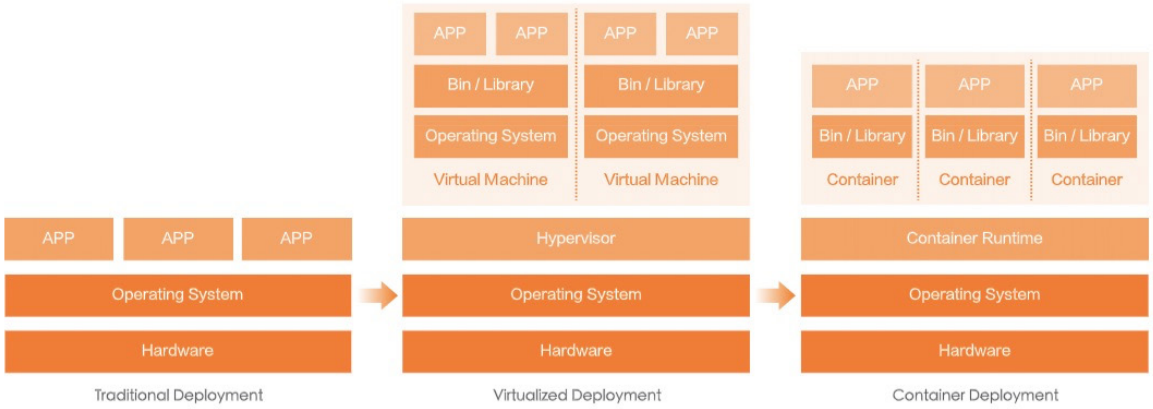


图 8-2 容器演进历程

2008 年 Linux 就提供了 Cgroups 资源管理机制、Linux Namespace 视图隔离方案，让应用得以运行在独立沙箱环境中，如图 8-2 所示。阿里巴巴也通过这一块技术研发了第一代容器 -T4，它的隔离性比普通虚拟机弱但更加轻量化，性能损耗很少，所以一经推出便大受欢迎，并逐渐取代虚拟机，承担了集团整个交易系统的计算资源。后续 Dotcloud 公司开源了其容器创建技术 Docker，Docker 引入了容器新技术 - 镜像，镜像包含应用代码和所依赖的所有文件和目录。将镜像打包上传到镜像库后，无论在何种环境，工程师只需到库里下载镜像，便能重新创造出跟之前一摸一样的容器，有了它可以完美地解决了 DevOps 两大难题——发布成功率低和运维压力，同时更大程度上降低了容器技术的使用复杂性，加速了容器技术普及，同时 Docker 容器也成了业界的事实标准。

8.2.2 ASI

Alibaba Serverless infrastructure 的缩写，是阿里集团针对云原生应用设计的统一容器基础设施，底座基于 Kubernetes。其作为阿里经济体容器全面上云的引领实施者，承担了包括容器全面上云、轻量级容器架构演进、集团运维体系云原生化等职责。图 8-3 展示了 ASI 的架构图。

仅限内部传阅

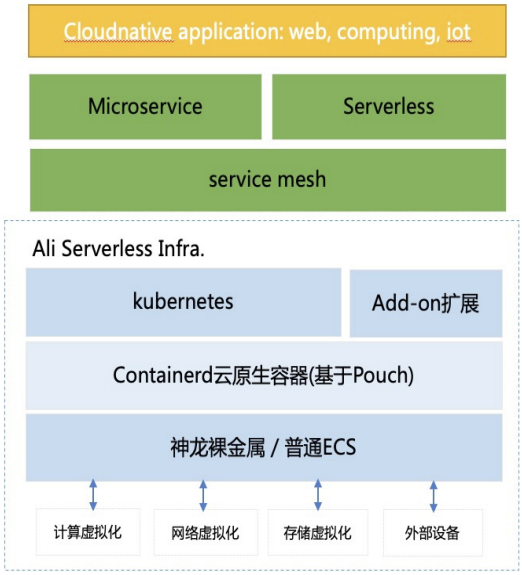


图 8-3 ASI 架构图

ASI 有以下特点，组件图如图 8-4 所示：

- 原生 K8S API，提供完整的云原生技术栈支持
- 以容器为中心，节点运维全托管
- 低运行时和弹性开销，没有虚拟化层开销
- 和阿里内的基础设施、中间件天然集成
- 高效的应用交付效率，支持万级实例数的应用发布
- 企业级的安全保护，完备的限流、风控机制
- 差异化 SLO 任务混合智能调度，提供可抢占容器实例，降低成本

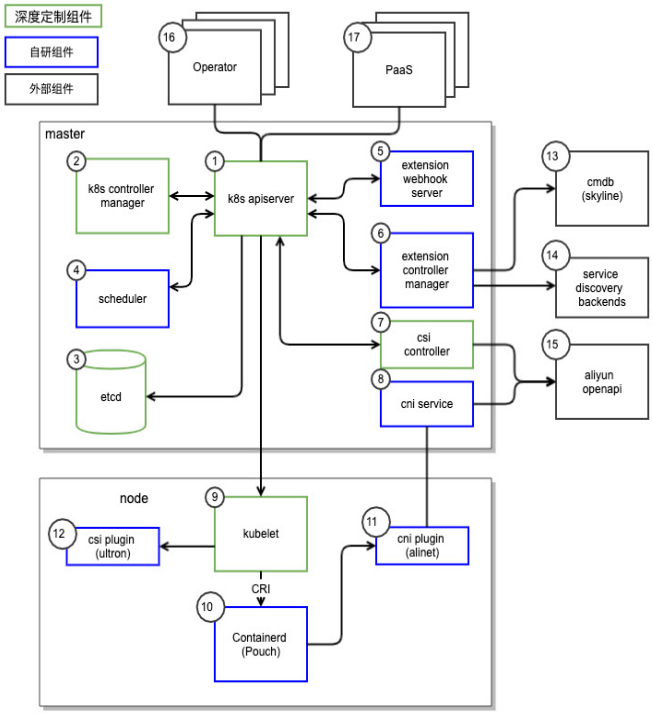


图 8-4 ASI 组件图 8.2.3 ServiceMesh

阿里中间件团队提供了大量丰富的中间件产品，每个中间件产品背后都有一个强大的架构和分布式系统来支撑，例如 HSF、Dubbo、Sentinel 等，这些中间件产品的出现给开发带来了很大的便利性，同时提供了很轻量级的接入方式，只要引入客户端 SDK 即可使用，所有这些好的中间件产品在阿里内部是广泛使用，但虽然我们微服务升级，机器规模的扩大，频繁的 SDK 升级给测试和开发带来了很大的人力成本，服务治理迫在眉睫，所有在 19 年阿里内部就开始向 ServiceMesh 方向做技术探讨，为后续创造了一次以开发者为中心去打造面向未来的分布式应用开发平台的机会。

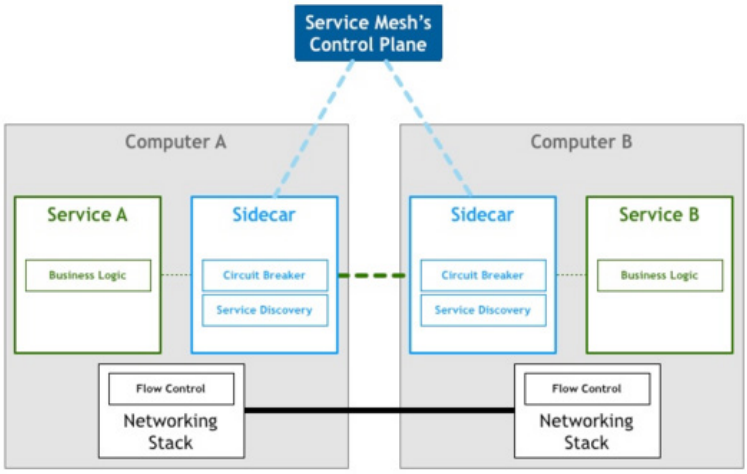


图 8-5 ServiceMesh 示意图

如图 8-5 所示，Service Mesh 是处理服务之间通讯的基础设施层。它的职责是在复杂的拓扑结构下，让应用服务之间进行可靠的数据传送。它是和应用服务容器部署在一个 Pod 里面的轻量级网络代理，这样可以对应用服务进行充分解耦。Service Mesh 的架构思想是建立在云原生基础上的，及基础设施下层，之前流量的路由、限流、安全校验等都是集成到客户端 SDK，现在统一落地到云原生基础设施，让底座越来越厚，业务越来越薄。另外对于多语言场景下 ServiceMesh 天然带来了很大的便利性，在重客户端 SDK 场景下，一个中间件产品如果需要支持多语言那开发和运维成本都很高，在 ServiceMesh 场景下可以通过去 SDK 或轻量级 SDK 来降低成本。

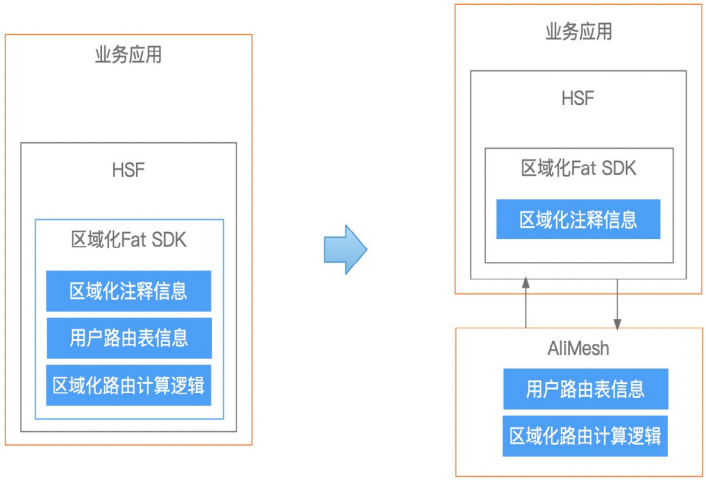


图 8-6 国际化 ServiceMesh 下沉

仅限内部传阅

在国际化云原生架构下，我们主要将国际化特色的区域化路由和多租户路由统一下层到 ServiceMesh 中，由图 8-6 中可以看出我们将区域化 Fat SDK 进行了瘦身，主要将用户路由信息、区域化路由计算逻辑统一下层到 ServiceMesh，区域化路由 Fat SDK 变成了 Thin SDK，Thin SDK 里面的逻辑很简单所以更新频率很低，这样很方便以后的维护。

8.2.4 GitOps

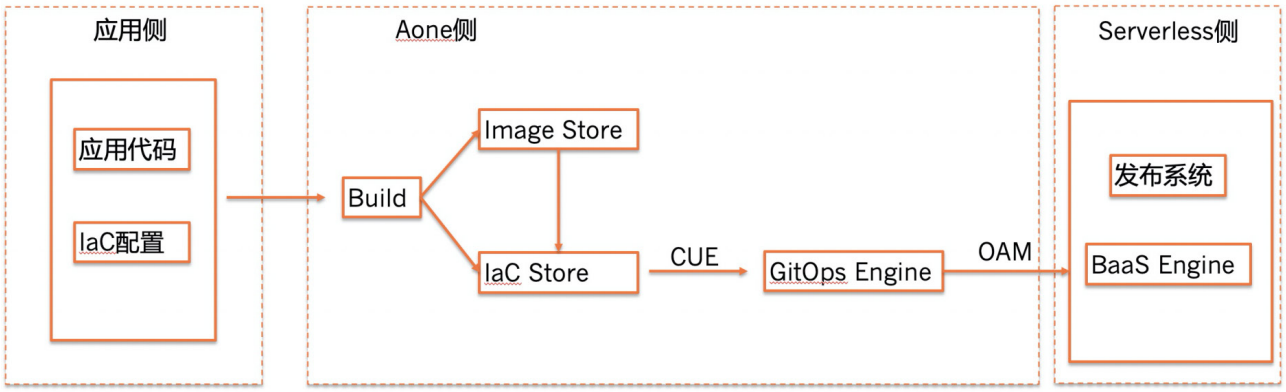


图 8-7 GitOps 持续交付

GitOps 是一种持续交付的方式。它的核心思想是将应用系统的声明性基础架构和应用程序存放在 Git 版本库中，如图 8-7 所示。将 Git 作为交付流水线的核心，每个开发人员都可以提交拉取请求（Pull Request）并使用 Git 来加速和简化 Kubernetes 的应用程序部署和运维任务。通过使用像 Git 这样的简单熟悉工具，开发人员可以更高效地将注意力集中在创建新功能而不是运维相关任务上（例如，应用系统安装、配置、迁移等）。

8.3 国际云原生基础架构

随着阿里经济体全面上云和云原生技术的升级,基础架构必然会从云下走向云上,再从云上走向云原生。从 CloudHosting 走向 CloudNative，给基础架构带来了巨大的变化，从基础设施下沉、无服务器化、可观测、系统的韧性、系统的自动化，无不体现了云原生是面向未来的架构。

8.3.1 基础设施下沉

图 8-8 是业界基础架构的一般演进过程，从开始 IDC 到服务器托管给 IDC，再到后来基础设施上云，我们发现开发人员和运维人员关心的东西越来越少，从之前的黑屏批量到现在控制台上的一个按钮，从一人运维千台服务器就累到不行到现在一人维护万台甚至几十万台都很轻松，同时上云之后变更的安全性大大的提高。随着从 CloudHosting 到 CloudNative 的演进，应用托管以及通过 FaaS 来快速获取业务价值会变得更加容易，开发人员只需关心业务逻辑代码的编写，基础设施加系统的高可用全部由云来承担，让业务和基础设施架构解耦，让开发人员的价值最大化。

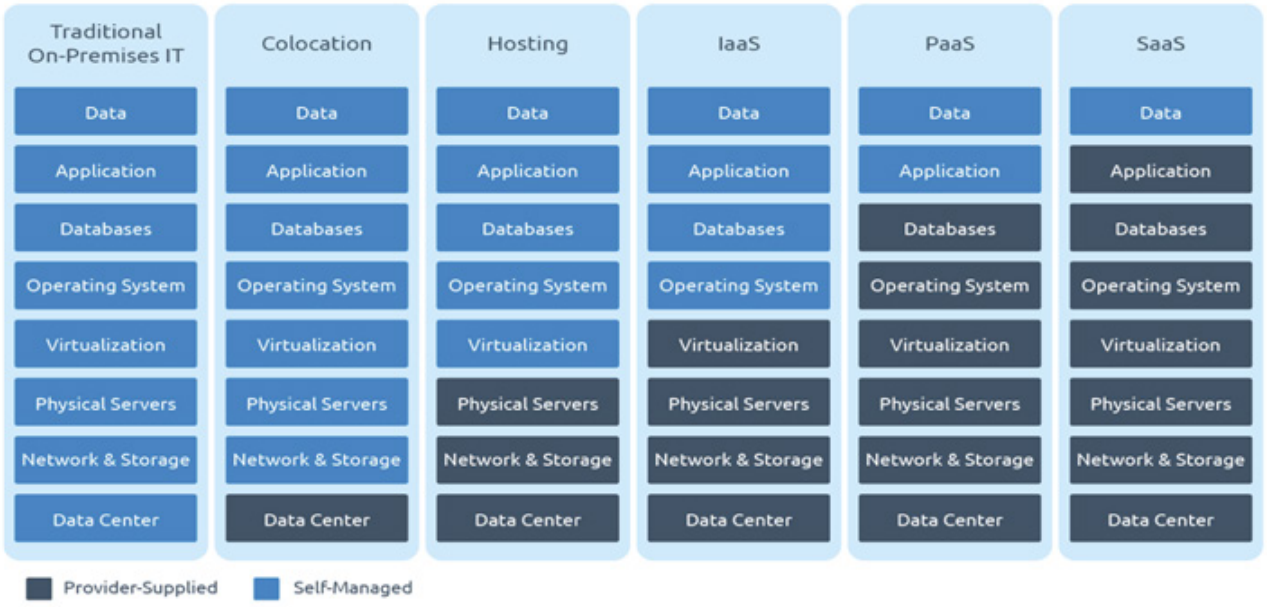


图 8-8 基础架构演进历程

在国际化云原生架构中我们使用云原生操作系统 ASI，ASI 作为阿里经济体容器全面上云的引领实施者，承担了包括容器全面上云、轻量级容器架构演进、集团运维体系云原生化等职责；在此基础之上加速新兴的技术包括 mesh，serverless 在集团内的落地；并进一步作为容器调度基础技术中台支撑了阿里经济体众多业务场景及生态（如函数计算、异构机型、新兴业务等）。而对终端用户，ASI 云原生标准化、声明式、面向终态以及 ASI Defender 防御体系的架构升级，带来了研发效率、交付效能、安全的提升。当前国际全球 7 大云机房已经从 Sigma2.0 升级到了 ASI，在 ASI 架构下我们发现资源的交付速度更快，成功率也上了一个新的台阶。

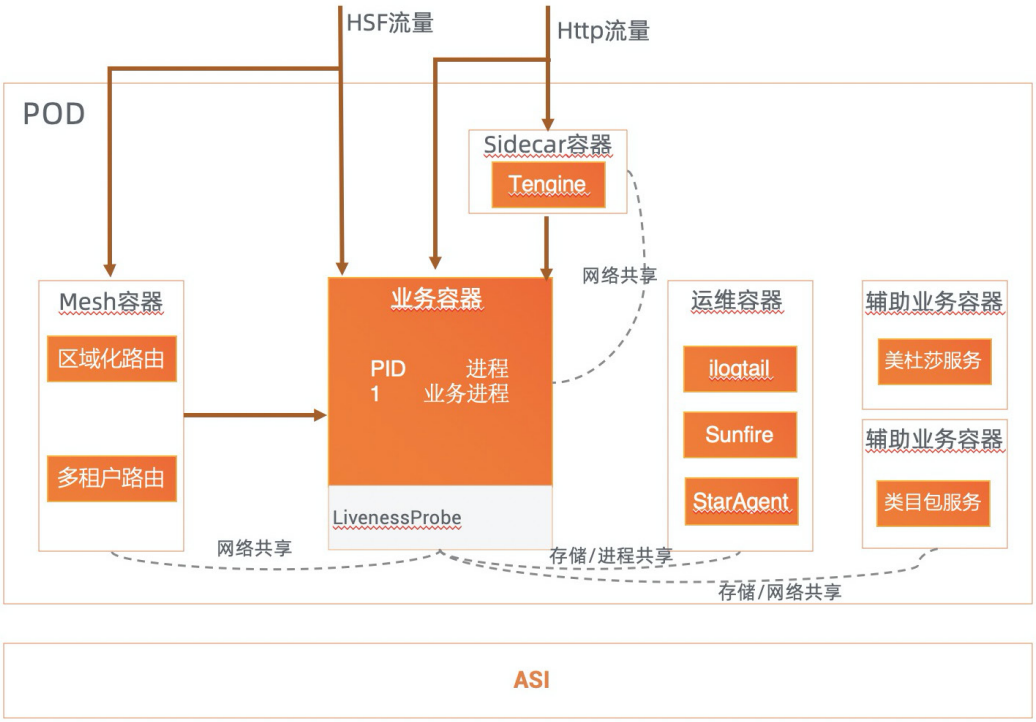


图 8-9 国际化场景下的 ASI

仅限内部传阅

如图 8-9 所示，在 ASI 中交付的最小单元是 Pod，Pod 可以理解为是一组容器的编排，Pod 内的容器之间网络、存储是互相共享，通过这一特性我们将运维进程、Tengine 进程、公共基础包、基础服务下沉到不同 Sidecar 中，这样在业务容器中只有业务应用单一进程，开发人员也只要关心业务容器，其他服务统一由基础设施平台统一维护，带来的好处是从之前的进程隔离到现在的容器隔离，隔离性更好，不会出现进程之前互相影响的情况。

一个部门或者一个公司一般都会有自己的研发框架，例如阿里的 PandoraBoot，统一的框架将程序的启动、健康检查、下线等流程都是标准化的，这就给基础设施的统一性带来了很大的便利，我们可以通过统一的 Dockerfile 来构建基础镜像，这一基础镜像可以无缝覆盖 80%~90% 的应用场景，带来的价值就是研发同学再也不要特别关心 Dockerfile，同时可以让应用镜像拉取速度更快。

8.3.2 弹性

弹性伸缩是云原生中的一大亮点功能，当系统水位高的时候可以对应用进行扩容，提升实例数来应对流量高峰，当系统水位低的时候可以对应用进行缩容，减小实例数以避免资源浪费。我们可以通过对应用实例数进行自动弹性来降低我们的成本。例如直报出现爆款产品的时候，商品出现了流量高峰，此时需要大量的实例来处理访问流量，这个时候就需要自动扩容，而当直播结束时，需要对服务器进行缩容来减少成本，所以我们可以通过自适应弹性来支撑流量不稳定场景，最大化的利用机器资源来降低成本。

在国际化云原生架构中提供了 3 种容器弹性方式：

- **manual-scaler**：手动弹性，类似传统模式，用于用户自己指定到一个固定的实例数，不支持基于 metrics 响应的自动弹性。
- **auto-scaler**：自动弹性，serverless 领域真正关心的自动弹性配置，支持基于按照配置的 metrics 进行弹性的设置。
- **cron-scaler**：基于时间表达式的自动弹性，主要应用于特定提前知道流量模型的场景，可以基于流量模型进行配置，做到提前控制扩缩。

当前弹性系统的设计还是主要面向无状态应用，同时希望开启自动弹性之前对系统的启动速度做下优化，这样可以更快的交付服务。

对于应用依赖的数据库或者中间件尽量使用云服务（比如 BaaS），把有状态部分保存到云服务中，云服务帮我们做统一的运维和弹性。

8.3.3 可观察

如图 8-10 所示，可观测性由日志、指标和追踪三根支柱来构建，可观察性不是云原生带来的新概念，

只是可观测性对云原生很重要，大家都知道云原生中的声明式理念，声明式带来了大量的异步流程，异步过程中需要有很好的可观察性，我们需要通过可观测性分析在面向终态的执行中出现了什么异常，自愈不行的情况下可以让人快速介入。

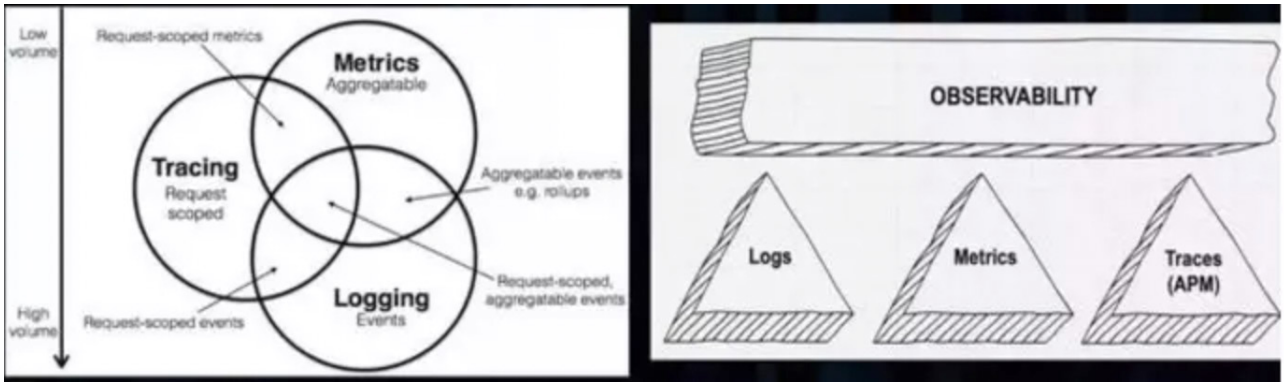


图 8-10 可观察基础理论

指标数据（Metrics Data）

描述具体某个对象某个时间点的值。在 Prometheus 中，指标有四种类型，分别 Counter（计数器）、Gauge（瞬时值）、Histogram（直方图）和 Summary （概要）， 通过这四种类型，可以实现指标的高效传输和存储。在集团内主要对应的产品是 Sunfire，通过 Sunfire 可以看到系统指标数据、应用指标数据、业务指标数据等全维度的指标。

日志数据 (Logging Data)

日志是我们日常排查问题的重要依据，所以日志对业务系统十分重要，当前在云原生中通过在 IaC 中的声明日志采集功能，可以快速的将重要的日志通过 logtail 统一采集到 SLS 后展现，通过 SLS 强大的检索能力可以快速定位出问题，同时我们需要注意日志中不要出现敏感内容，防止数据泄露。未来我设想日志采集也会下层到基础设施，开发同学不需要感知到日志的采集配置，可以直接通过 logtail 采集到 SLS，然后直接在控制台检索日志。

跟踪数据（Tracing Data）

数据跟踪对日常问题及性能的排查比日志分析更加简易，通过整个调用链路可以快速分析出某一个异常服务，这一块还是使用的集团鹰眼，后续也会整体升级到阿里云 ARMS，ARMS 是一款应用性能管理产品，包含前端监控，应用监控和 Prometheus 监控三大子产品，涵盖了浏览器，小程序，APP，分布式应用和容器环境等性能管理，能帮助你实现全栈式的性能监控和端到端的全链路追踪诊断， 让应用运维从未如此轻松高效。

8.3.4 韧性

韧性是系统对于异常的抵御能力，体现了软件持续提供业务服务的能力，核心目标是提升 MTBF（Mean Time Between Failure，平均无故障时间）。从架构设计上，包括高可用、容灾、异步化能力，比如消除单点、服务分级、限流降级、超时重试、容灾单元、主从 / 集群、异地多活等。这是个很大的主题，

仅限内部传阅

不在这详细的展开。

在国际化云原生中在原来基础上加上了多个重量级工具：

- **进程异常退出或夯住的必杀器 - 探活探针。**探活探针主要是周期性探测服务的健康状况，如果发现多次探测失败会触发容器重启，当出现 oom、oom-killer 等情况可以快速恢复服务。
- **支持应用级别、服务级别、服务方法级别切流容灾。**当前我们遇到变更带来的服务异常时，大招就是切流，将异常机房的流量全部切到正常的机房，这样简单粗暴会带来成本和风险问题，成本主要体现在我们必须在单机房内部署更多的服务来提供全站流量，风险是当前能不能将一个机房流量整体切到另外一个机房需要慎重评估。如果我们可以支持应用级别、服务级别、服务方法级别的切流，那对我们来说风险更小，成本更低。

8.3.5 自动化

集团 Aone+Normandy 等基础平台已经帮助我们解决了大部分研发运维自动化问题，例如 CI/CD 自动化流程、日志自动化清理、宿主机故障容器自动腾挪等等，云原生中让自动化上了一个新台阶，主要从两个层面来看。

- **系统层面**：宿主机自愈、容器异常自愈
- **服务层面**：服务异常自愈、自动化 Quota 分配、自动化弹性、自适应限流

8.4 国际云原生应用架构

对于云原生应用架构，我们的核心思想是基于 K8S 的编排能力，重新定义应用的交付过程以及交付形态，打破常规的中心化静态声明式集成，以***模块化独立演进交付，提高迭代的并行度，以及交付的灵活度***，解决应用“中心节点”问题，按照职责划分模块，并且各自能够互不干扰的迭代演进以及交付，***从而最终提高软件研发效率，以及基于云原生重新定义软件研发模式***。所以我们基于当前可以看见的几种场景抽象出几种明确的升级模式，希望能够给大家带来更多对于云原生思考以及启发。

8.4.1 模块化交付模式

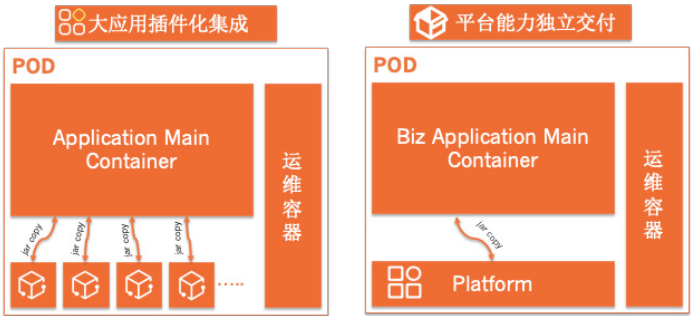


图 8-11 云原生规模化交付模式适用范围：业务研发模式 & 业务架构升

适用范围：业务研发模式 & 业务架构升级 .

场景：Pandora sar 类型的中间件、中台平台包等通过代码集成的方式提供服务的方式。

解决问题：打破传统的静态代码声明式集成，基于云原生编排方式集成，提供模块化独立研发、迭代、交付通道，从而提高业务模块交付确定性 & 效率

优势：

- **终态一致性**：对于业务运行态升级前后终态一致性（所有应用模块都在一个进程中），运行态对升级无感，
- **稳定灵活的交付**：提供了更加细粒度的变更管控、交付能力；提高模块间的交付并行度，提高交付效率；由于交付内容镜像化，使得交付内容的确定性增强。
- **独立迭代**：提供了各自独立的迭代空间
- **独立类加载空间**：对于这种模式为了解决各个 模块最终集成的类冲突问题，我们将整个模块的交付集成过程基于 pandora 来做，所以每个模块拥有独立的 pandora 模块类加载器去进行加载，从而实现类加载隔离。

约束：

- **Metaspace 膨胀**：为了解决各个模块类加载隔离，所以使用了 pandora 的隔离机制，所以如果每个模块之间对于公共包重复度很高，会导致类重复加载，从而导致 metaspace 膨胀，所以在制作模块的时候，一定控制使用的最小范围，避免出现 metaspace oom。
- **模块上线**：模块化交付上线无法做到应用层无感，需要应用进行一次重新部署才能出发一次新的 K8S 编排，才能触发模块升级

8.4.2 独立服务能力交付模式

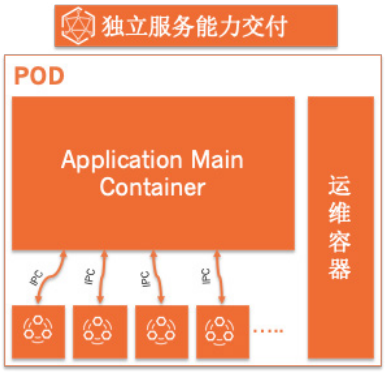


图 8-12 独立服务能力交付模式

适用范围：独立功能组件能力下沉

场景：类目包、美杜莎包等占用很大内存的二方包

解决问题：

- **资源隔离**：解决独立的功能组件对主应用的影响，比如 CPU，内存，存储上对主应用形成干扰，从而引发主应用不可用

仅限内部传阅

- **职能解耦**：将功能组件的交付和应用自身交付解耦，能够提供应用无感的升级迭代

优势：

- **独立迭代**：提供了各自独立的迭代空间
- **资源隔离**：组件运行态在资源上能够实现和主应用隔离，组件自身的基础指标下降，不会触发应用的不可用
- **独立交付**：能够独立于应用升级实现自身组件能力的交付

约束：

- **运行态**：运行态不能以应用形态运行，否则和主应用主体存在冲突和干扰
- **交付内容**: 交付内容是比较独立的功能服务,除了暴露 POD 内容器间的服务,不能暴露外部服务。
- **内存资源消耗**：POD 内 CPU 可以通过 cupshare 实现共享，但是内存出现一定的增加或者压缩主应用内存
- **复杂度增加**：这种模式对于架构复杂度，以及运维复杂度都会有一定增加，从稳定性保障上来说，也增加了一层保障返回

8.4.3 云原生研发交付模式

面对云原生提供的能力以及新的研发理念（不可变的基础设施），我们尝试将平台定义为业务的基础设施，以基础设施的交付形态将平台能力交付给上层的业务实现。为了解决运维权限不清晰、管控权限不隔离、监控告警不隔离等问题，我们基于云原生实现平台能力通过基础设施通道交付，以及应用归属业务实例的改造。**其本质变化就是平台基于镜像交付能力，业务实例以独立应用去进行迭代。**如图 8-13 所示。

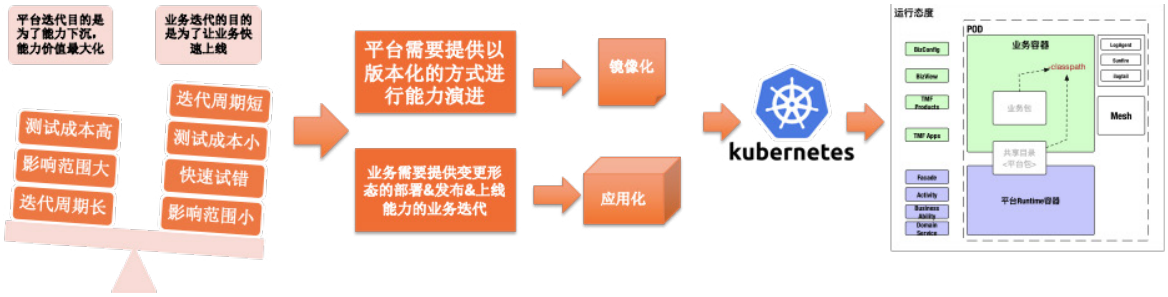


图 8-13 云原生带来的中台研发交付模式演进

对平台视角和业务视角看待变更去分析，会发现，平台的变更节奏、成本以及目的是和业务变更的节奏、成本和目的是不一致的。平台的目的是希望更多的通用模式以及能力被下沉，而业务的目的是实现业务需求的快速交付，针对这两者的差异，两者的迭代演进不能以相同的方式和路径来实施，所以我们整个演进思想是将平台镜像化和站点应用化。平台镜像化主要从以下两点考虑：

- **交付内容的确定性**：因为镜像的不变性，那么只要不使用新的镜像，镜像内容不会随着业务站点应用的重新构建而发生交付内容的不一致。这是当前以二方包模式存在很大的差异，因为二方包的模式依赖 maven 的仲裁策略实现最终交付内容，而 maven 仲裁策略依赖当前应用 maven 依赖上下文，这个不是由二方包的提供方所能控制的，是应用研发控制，可能由于应

用引入一个新的 maven 依赖或者 maven 改动，导致平台二方包的间接依赖的内容发生变化，也导致平台最终交付给业务的内容也发生了变化，这是不符合平台交付预期的。所以这是本次基于云原生的编排能力，将平台镜像化改造的一个初衷。

- **平台脱离业务独立稳定演进**：站在平台角度目标是趋于稳定，所以平台的迭代速度远低于业务迭代速度，所以平台镜像交付可以实现平台交付物脱离业务应用，自己独立交付对应版本的镜像，平台升级只需推动业务编排入最新的平台镜像即可。同时基于镜像平台可以根据自己的节奏将平台能力升级，业务可以根据需求选择对应的镜像版本实现平台能力交付，最终形成两条独立自主的迭代链路，业务按需或者平台按照节奏推进自身的交付和升级。

整体落地我们分为以下几个部分：

- **运维态迁移**：因为本次升级后，运行态的应用发生了变化，那么之前应用上的运维态内容都需要迁移到新的应用上，这里主要包含 diamond 归属、switch 迁移、监控告警迁移。
- **平台镜像交付物标准**：因为本次平台交付物不只是单个二方包，而是平台所有能力集合，所以这里需要提供并且定义构建的产出物标准，这部分是通过自定义 maven 插件完成打包动作，完成交付物产出，以及结构标准。
- **平台变更迭代交付体系**：由于当前集团 aone 的变更体系只有对于二方包和应用交付形态的支撑，而对于镜像能力交付变更没有一个产品体系支撑，所以我们需要提供一套新的变更产品体系去支撑平台的变更迭代演进以及交付。这部分是基于 aone 的构建能力，以及和底层云原生研发体系打通，实现整个产品能力落地。
- 平台和站点应用启动集成过程：由于本次方案最后的集成过程是 jar 包拷贝的方式交付平台能力，那么这个过程需要被统一定义，所以我们定义了统一的应用基础镜像和平台集成镜像，实现这个交付过程。
- **构建站点应用 IaC 规范**：IaC 是用于定义应用对基础设施的依赖，而平台之上的业务对基础设施的依赖不完全来自业务方，也有来自平台侧，而 IaC 只能归属某个应用，所以为了将平台对这些集成设施依赖对业务侧屏蔽，降低平台认知成本，我们本次对站点应用的 IaC 进行了分层抽象，定义了站点应用 IaC 的内容规范（如图 8-14 所示）。

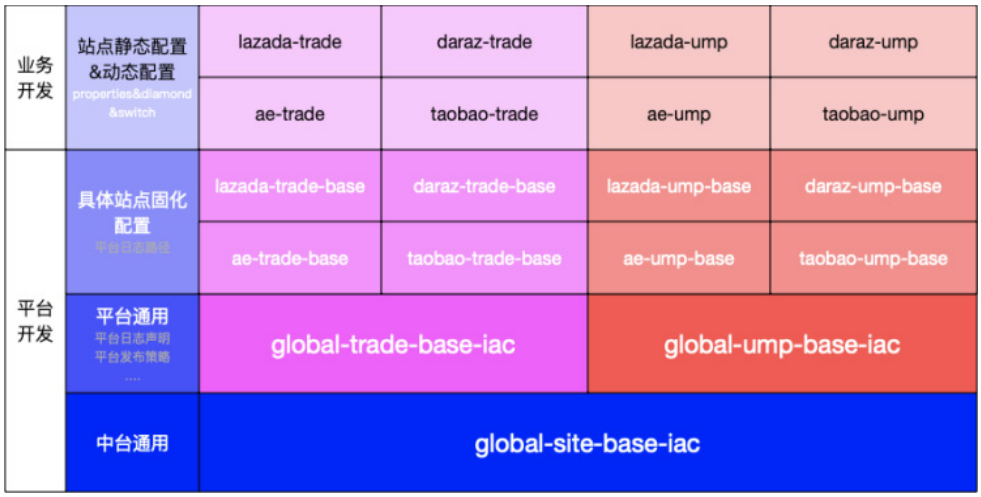


图 8-14 IaC 站点应用分层规范

仅限内部传阅

为了支撑本次架构演进落地，我们沉淀了一套新的平台能力研发迭代产品体系，以及平台交付的容器编排声明，整体实现如图 8-15 所示：

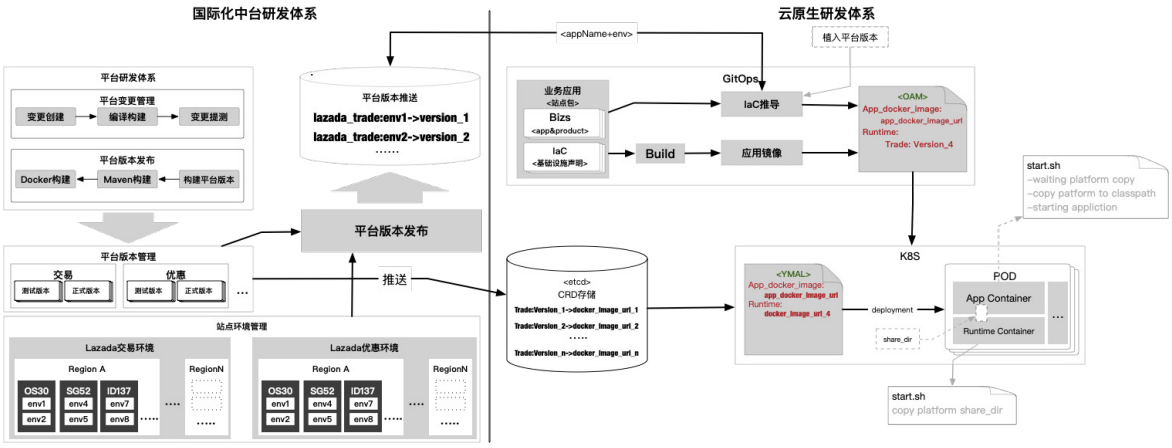


图 8-15 IaC 站点应用分层规范

在完成整体方案落地之后，我们将中台之上的四个业务站点完成了应用化改造，并且运维态也顺利平滑的迁移到新的应用之上，如图 8-16 所示。整体新的架构我们经过一个多月的方案讨论和论证，一个月的基础能力建设研发，两个月的切流，成功在国庆之前完成了四个业务站点两个核心系统（交易、营销）架构升级上下交付，并且顺滑支撑了今年双十一大促，同时也论证了我们当前在基于云原生的架构模式的可行性，基于当前平台镜像化的编排交付模式，可以更加抽象的去看待业务架构如何基于云原生进行演进，以及业务架构升级可基于云原生的实施途径，从而让云原生下的业务架构升级变成普适性。

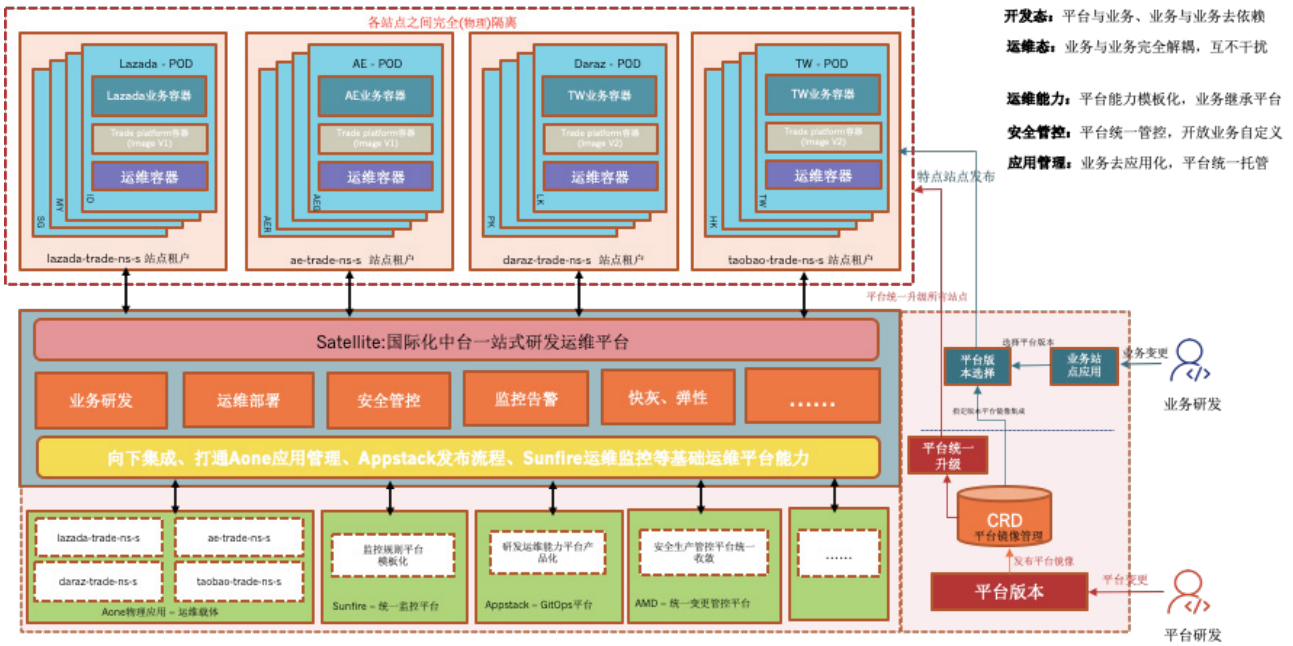


图 8-16 基于云原生的研发运维体系

8.5 云原生未来展望

图 8-17 云原生未来演进方向

云原生有很多可憧憬的未来，云原生技术本身各自也有很多发展趋势，总体上，这里引用@李响在“FY21云原生峰会主论坛”上对云原生的未来展望，如图 8-17 所示：

- **易操控：**云原生将把云带向更容易操控的状态，让用户更简单、更便捷的去使用云，比如配置语言、LowCode 应用开发、应用编排、资源编排、标准 API/SDK；
- **更开放：**云未来一定会变得更为开放，云生态体系会变得更加开放，比如标准开放的云原生应用与服务、自言增强实现、三方厂商跨云；
- **无边界：**从原来的专有云、公共云这种相对独立的模式，发展到混合云的模式，再走向未来个更为分布式的云，能够去把我们的边缘计算纳管起来；
- **无服务：**将更广泛的普及，愿意使用 Serverless 带来的事件驱动、极简开发、按量计费等构建自己的应用；
- **更安全：**让整个应用得到一个端到端的一个安全的，更可靠的一个保障，比如安全可信、统一鉴权、网络数据安全、TPM 可信硬件。

国际化云原生样板间也在云原生路上一直探索着，目前我们已经完成了云原生的基础设施升级和一些业务架构的探索，对于未来的方向一定是 Serverless，Serverless 让不变的“基础设施”逐步下层，这里的基础设施不单单包含 IaaS 层和 PaaS 层，公共业务代码一样属于基础设施，这一些都会沉淀到云原生平台，沉淀到平台后业务只需关心业务，对于机器、扩缩容、运维都由云原生平台自动化完成，这样最大化降低人力成本。特别地，可以面向研发运维效率角度，应用 Serverless 化，后端服务 BaaS 化，实现全托管、免运维；应用与基础设施解耦，站点和平台解耦；通过云原生做技术创新，例如通过 sidecar 做业务之间解耦；通过 Mesh 实现服务路由、服务鉴权、服务限流、服务熔断、服务降级等和流量相关的工具链。同时在面向稳定性角度，通过云原生的演进支撑服务在线，提高系统的稳定。

仅限内部传阅

九、全球合规

9.1 全球合规背景

在互联网时代兴起之前，个人数据就一直伴随着人类社会的发展而发展，广义上来说和个人社会活动相关的数据都可以认为是个人数据，这些数据可能不会引起注意，但切切实实围绕每个人展开，例如个人的地址信息，银行账户信息等。通过这些信息可以定位到具体个人，如果是不法分子恶意使用这些信息的话可能会构成犯罪活动。

进入互联网时代之后，随着数据产生和传输的复杂化，信息获取途径的丰富多样呈指数级增长，对于个人数据的获取和使用也越来越广泛，随之而来的也是对个人数据的误用、滥用等。不过随着社会的不断发展和进步，越来越多的个人和群体意识到了隐私数据的重要性，也有越来越多的国家和组织出台了相关的法律法规来保障互联网时代的个人隐私数据，
传统隐私更多的关注数据使用，防止数据在某一区域或者某一个领域被滥用，互联网时代的合规隐私不仅关注这些，同时也会关注数据跨境传输。

9.1.1 相关法律法规的介绍

随着隐私数据受到越来越多国家和地区的重视，对应的法律法规也逐渐出台，整个的体系也在不断完善中，这里挑选几个典型的国家地区出台的法律法规做个简单介绍。

① 中国网络信息安全法

我国于 2017 年 6 月 1 日正式发布《中华人民共和国网络安全法》。《网络安全法》是我国首部全面规范网络空间安全管理方面问题的基础性法律，包含的内容十分丰富，一共包括 7 章 79 条，包含网络运行安全、关键信息基础设施的运行安全、网络信息安全等内容。《网络安全法》在数据（包括个人信息）安全与保护上也有诸多规定，诸如第四十至四十五条。

② 中国数据安全法

2020 年 7 月 3 日，《中华人民共和国数据安全法（草案）》在中国人大网公布，公开征求意见，征求意见截止日期为 2020 年 8 月 16 日。主要内容包括：确立数据分级分类管理以及风险评估，检测预警和应急处置等数据安全管理各项基本制度；明确开展数据活动的组织、个人的数据安全保护义务，落实数据安全保护责任；坚持安全与发展并重，锁定支持促进数据安全与发展的措施；建立保障政务数据安全和推动政务数据开放的制度措施。

仅限内部传阅

③ 欧盟 GDPR

欧盟于 2018 年 5 月 25 日正式发布 GDPR《通用数据保护条例》，这是一项保护欧盟公民个人隐私和数据的法律，其适用范围包括欧盟成员国境内企业的个人数据、也包括欧盟境外企业处理欧盟公民的个人数据。

④ 美国 CCPA《加州消费者隐私保护法》

2018 年 6 月 28 日美国加州发布 CCPA《加州消费者隐私保护法》。该法案被称为美国“最严厉和最全面的个人隐私保护法案”，将于 2020 年 1 月 1 日生效。

⑤ 德国 BDSG《联邦个人资料保护法》

德国联邦议院于 2018 年 4 月 27 日通过《个人信息保护调整和施行法》，其中包含新的德国 BDSG《联邦个人信息保护法》。在这部新的法案中，已实施 40 年的 BDSG 进行了大幅调整以符合欧盟 GDPR《通用数据保护条例》。在新的 BDSG 法案中德国联邦政府运用了 GDPR 的开放性条款，导致部分新的 BDSG 规范内容超越了 GDPR 的条文规范，与现行欧盟法律不符，很可能被宣布违反欧盟法律。另一方面，旧的 BDSG 仅有 48 条规定，而新的 BDSG 则超过 85 条规定，且更为复杂，提高了法律适用上的难度。

9.1.2 典型合规隐私案例

进入互联网时代后，各互联网公司基于自身业务发展需要，被动或者主动地搜集了大量的用户数据，同时全球化的不断发展，也促使数据在国家地区之间的流动不断加快，但这些数据可能不经意或者故意被各公司用作一些商业目的或其他目的的活动，在不同国家和地区这些活动可能被认为是侵犯了相关法律条文的，下面就几起典型的数据隐私事件做个简单介绍和解析。

① Google 违反 GDPR 罚款 5000 万

2019 年 1 月 21 日，法国数据保护监管机构宣布，针对谷歌公司违反《通用数据保护条例》（“GDPR”）的行为，对其处以 5000 万欧元的罚款。该案作为自 2018 年 5 月 25 日 GDPR 生效后，法国监管机构作出的首个处罚决定，也是截至当时欧盟国家依据 GDPR 开出的最大罚单。

② Facebook 德国数据垄断案解析

2019 年 2 月 7 日德国反垄断监管机构“联邦卡特尔局”裁定 Facebook 在收集、合并和使用用户数据时滥用了其市场支配地位，联邦卡特尔局认定 Facebook 的行为属于滥用支配地位，无限制地从第三方网站收集各种用户信息并将其与用户的 Facebook 账号相连结，要求其在 12 个月内停止这些滥用行为。该案是数据保护规则与竞争规则融合第一案，联邦卡特尔局该裁定将会具有历史性的意义。

③ 美国封禁 Tiktok 事件解析

自去年底以来，美国对于 TikTok 的监管措施不断升压，主要聚焦在数据和内容两个方面，监管事由从

仅限内部传阅

业务合规逐渐上升到国家安全；7 月底，美国通过法案禁止联邦政府雇员在政府设备中下载 TikTok；8 月 3 日，美国政府要求 TikTok 在 9 月 15 日前完全剥离在美业务将公司出售给美资公司；8 月 6 日，特朗普签署行政命令宣布将在 45 天后禁止任何美国个人及企业与 TikTok 母公司字节跳动进行任何交易，对此 8 月 23 日上午字节跳动发布声明表示将在美国时间 8 月 24 日正式起诉特朗普政府。

④ 阿里手淘 APP 隐私合规整改

2019 年 11 月工信部组织开展 App 侵害用户权益专项整治行动工作，对手淘等阿里多款 App 进行检测，不合规问题将在央媒通报并有大众监督整改，会带来较大的监管合规及舆情风险。当时手淘通过快速提供测试体验版的方式挂在了官网上，并承诺尽快将整改项并入主版本才勉强未被公开通报。2020 年 3 月，工信部组织对上轮检查的 App “回头看” 行动，不合规问题可能 315 前直接通报。阿里迅速响应，经过两天紧密的研发、提测、回归、适配等工作，最终得以稳定上线应用市场。

9.2 全球合规架构

9.2.1 全球部署

全球合规前提是全球部署架构，如果没有应用和数据的全球部署那合规也就无从谈起，在全球化架构中，全球部署也是重要的一部分。详细的全球部署细节可以参考《全球化部署》章节，这里就不详细展开，只大致提下目前阿里国际化业务中，AliExpress 业务是一个比较典型的全球部署案例，在合规管控前的部署架构如图 9-1 所示：

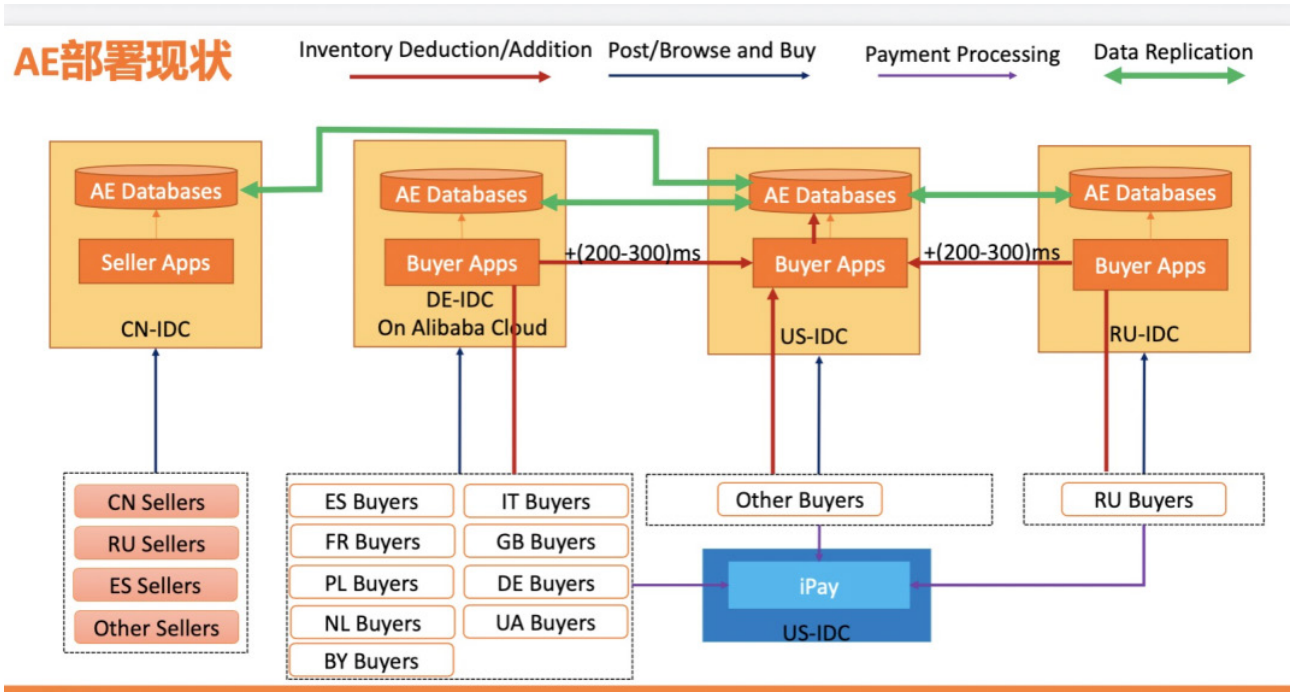


图 9-1 AE 部署现状

可以看到，旧的部署架构在应对一些合规要求时是无法满足的，原因在于数据中心是按照中国和美国来建的，而在一些合规场景中，可能存在这些数据中心禁止传输等风险，所以需要重新选择一个新的

数据中心，且该数据中心在合规角度来看需要尽可能减少风险。基于这个考虑，我们升级了 AE 的全球部署架构，终态的部署架构如图 9-2 所示：

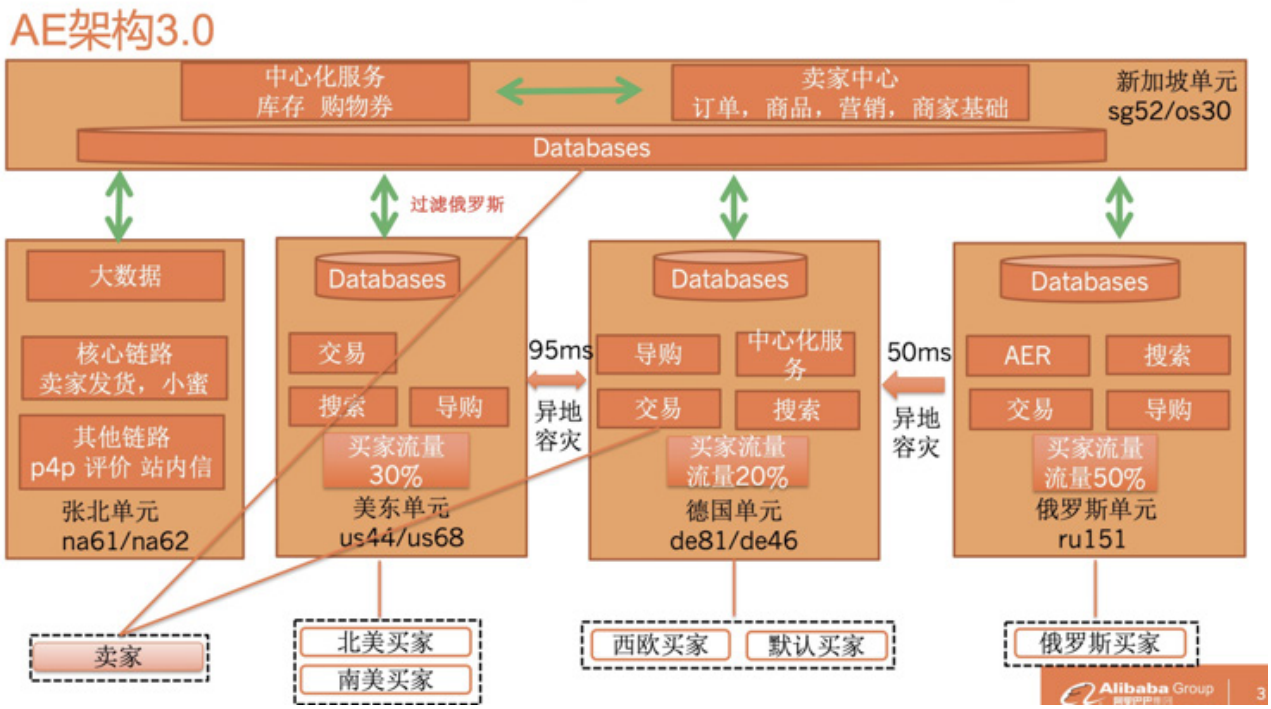


图 9-2 AE 架构 3.0

可以看到，新加坡作为数据中心，承载了业务的全量数据，为什么选择新加坡作为数据中心，除了业务上的决策外，从合规角度来看，新加坡作为世界上数据合规要求比较高且较多处于白名单国家之一，比较适合做数据中心和中转地区，包括俄罗斯、欧盟、美国等国家地区对新加坡都是可以接受的。这也是现在越来越多互联网公司选择新加坡建立数据中心的原因之一。

9.2.2 本地存储 & 跨境传输

在数据流动过程中，整体的一个链路是 **数据产生 -> 数据传输 -> 数据接收**，通常意义上我们说的本地存储是指在数据产生以及接收后的存储形态和介质处于法律条文规定的当地。且更严格要求下是数据首次产生时就要落盘本地存储。与此对应的是跨境传输，也就是数据在中间传输过程中涉及的一系列行为和操作。

我们先来看看什么是个人数据，严格来说并没有官方定义的个人数据，从一些法律法规中可以看到一些说明，例如 GDPR 中对个人数据的定义，是指已识别或可被识别的自然人（数据主体）的所有信息。可被识别的自然人是指其能够被直接或间接通过识别要素得以识别的自然人，如姓名、身份证号码、定位数据、在线身份等识别数据，或者通过该自然人的物理、生理、遗传、心理、经济、文化或社会身份的一项或多项要素予以识别。而在国家标准化委员会的定义中则是以电子或者其他方式记录的能够单独或者其他信息结合识别特定自然人身份或者反映特定自然人活动情况的信息；一旦泄露、非法提供或者滥用可能危害人身和财务安全，极易导致个人名誉、身心健康受到伤害或者歧视待遇的个人

仅限内部传阅

信息。综上，个人数据是指能够直接或者间接识别到具体自然人的相关数据。

本地存储

对于大部分场景来说，合规的第一点要求是本地存储，特别是在一些国家和地区，本地存储是必须满足的第一要求，需要明确的是，这里的本地不是物理意义上的国家和地区，而是指司法管辖区，举个例子,美国不同的州有不同的合规法律,州之间的跨境传输需要根据源和目的地之间的法律来综合判断。

跨境传输

指个人数据从一个司法管辖区传输到另一个司法管辖区。注意这里用的是司法管辖区而不是国家或地区，可能同一个国家或地区的不同司法管辖区的合规政策也有区别。

典型国家和地区的本地存储 & 跨境传输政策

- **俄罗斯个人数据本地化和跨境传输**

对个人数据的定义和欧盟类似，不同的是未提及在线标识 (online identifiers)。但实践中很可能会借鉴GDPR，把在线标识认为个人数据。

本地化数据处理规则包括：俄罗斯境内数据库建成后处理数据前应将地址告知监管机构；且使用俄罗斯境内服务器处理数据；数据处理需要在本地完成。

跨境传输规范包括：对于传输到白名单国家无限制要求，一次传输到非白名单国家，在非履行合同情况下需要数据主体书面同意（门槛较高）；如果一次传输到白名单国家后二次传输到非白名单国家，则数据主体在线同意即可。

- **新加坡个人数据本地化和跨境传输**

对个人数据的定义是值关于一个可被识别的个人的数据（无论真假），若通过某数据可以识别到个人，或者该数据和”组织“很可能可以访问的其他信息结合后可以识别个人，那就是个人数据。不包含商业联系人信息。

本地化数据：一般情况下新加坡不要求个人数据本地化，可综合客户需求，用户体验，监管期待等决定数据是否需要本地化。

跨境传输：一般情况下数据传输需要遵循隐私政策，数据输出和接收方之间签署的符合新加坡法律的数据 保护合同。

- **德国个人数据本地化和跨境传输**

德国或者欧盟的个人数据范围非常广，不仅包括姓名、电话、邮箱等可以直接识别到个人的数据，也

包括可以识别设备的数据和很多其他不能直接、但间接识别个人的数据。例如含有个人联系方式的订单信息、交易数据等。

本地化和跨境传输规范主要分三种情况：一方面，根据《欧盟数据保护法律指令》，德国对欧盟其他成员国的数据保护持信任态度，允许与欧盟成员国之间进行数据交换。另一方面，根据欧盟 2001 年《关于向第三国的处理者传输个人数据的标准合同款的委员会决定》，若接受欧盟以外的运营商和服务商提供数据服务，则须签署欧洲标准合同，以确保数据处理过程中的安全性。第三，欧盟还制定了数据保护的可信任国家清单，列入清单的国家被认为在数据保护方面拥有信任保障，包括加拿大、瑞士、安道尔、挪威等国。

9.3 合规架构原则

① 本地部署 & 跨境传输原则

在设计系统时，需要考虑当地的法律法规政策条文，如果明确规定了数据需要本地存储，特别是首次数据落盘，那需要在当地部署数据中心，以及相关的系统。同时再从跨境传输方面看，如果数据在本地存储后可以跨境传输流出，那么需要考虑数据同步和数据过滤，如果数据在本地存储后且不可以跨境传输，那就需要考虑本对本模式，全部数据和业务系统都需要在当地部署并提供完善的一套服务。

② 最小化原则

最小化原则是指在个人数据收集过程中，数据收集服务应当遵循数据最小原则，不得过度收集、处理个人信息，个人信息的处理包括个人信息的使用、加工、传输、提供、公开等。实践中应当做到，在满足实现 APP 功能所必需的个人信息的数量、种类应尽可能少，自动采集个人信息的频率也应当尽可能小，不以欺骗、误导或者强迫等方式收集个人信息，也不得超出目的范围收集个人信息，即在够用的基础上不收集其他非必要的个人信息。

③ 建站合规原则

一般新开站点时需要考虑合规方面的因素包括当地法律法规、站点承载服务和数据用途、业务链路等。首先需要考虑的就是当地对合规要求是否能满足，如果无法满足可能需要重新评估地址选择。其次要看站点承载的服务是什么，如果是涉及到会员、用户相关数据的使用就要考虑这些会员数据是否需要额外部署到本地还是可以跨站点访问，部署到本地是否是需要首次落盘就在当地，且落盘后允不允许跨机房流出。

9.4 合规工具

在传统法律行业和合规审计中，我们常见的是合规专业人士们手动对账，手动审核风险，对于互联网时代的海量数据，这个效率是很低的，所以需要有一些工具能在一些关键的方面帮助我们解决合规问题，这里结合我们在实际项目中的实践和沉淀，主要讨论一下合规相关能有什么工具帮助我们以及一些设

仅限内部传阅

计原则和思路。首先我们会整体说一下工具相关的设计原则和应对合规的流程管控，其次会针对合规中的关键点进行一些讲解说明。

9.4.1 设计原则

合规工具的设计还是围绕数据流转来的，和数据本地存储跨境传输的流向基本保持一致，首先需要考虑的是如何对数据做产生的审计和管控，只有从源头做好合规评估和管控才能应对后面复杂的合规场景，理想情况下我们需要对源头产生的每一个数据能了解它的产生来源、业务含义、使用方式等，才能对后续数据的传输、在不同国家地区法律政策限制下的使用策略做到比较全面的掌握。其次合规工具重点需要考虑的是跨境传输中数据的管控，业务场景下的数据传输是否符合合规，传输目的是什么，一般对于个人数据来说，经济目的是主要的，但不排除政治或者其他因素。另一个是接受国家或地区对数据的保护水平如何，如果保护水平较低，就需要考虑在工具层面做到数据限制和风险控制。

源头审计

我们主要针对新数据和存量数据的源头审计展开说明。

对于一些新项目或者新流程来说，带来的必然会有一些新数据的产生，这些新数据是如何搜集的，搜集过程中是否涉及隐私合规，数据使用过程中涉及到影响面有哪些，这些都是需要我们考虑的，也是工具需要来解决的。欧盟的 GDPR 中已经对这部分有了比较规范的定义和实践，也就是合规中需要做的 PTA、PIA 以及 DPIA。

- **PTA**：隐私阈值分析。新项目或者新流程前针对隐私阈值的分析，以帮助项目和相关人员明确在隐私方面的影响和风险。
- **PIA**：隐私影响评估。从业务和合规性角度考虑，明确何时启动隐私影响评估（PIA）。同时，明确触发启动 PIA 评估的原因，原因包括该项目是否收集了新的用户信息、是否进行了新的隐私数据处理、是否涉及与其他服务提供商共享数据。
- **DPIA**：数据保护影响评估。依据 PIA 评估的结果，从隐私数据敏感处理的角度考虑，明确何时启动数据保护影响评估（DPIA）。同时，依据 PIA 评估的报告，明确触发 DPIA 评估的原因。

正常的一个评估流程如图 9-3 所示，可以看到，PTA 是整体的一个评估手段，而 PIA 和 DPIA 是这个手段中的两个阶段，且逐渐严厉和精细。针对以上这个流程，我们可以工具化来实现，在新项目和流程中设置卡口接入，进行 PTA 的评估。对于存量数据，相对来说就没有那么完善的流程控制了，目前能做到的就是对存量数据的元信息扫描和识别。

跨境过滤

源头数据有了比较好的管控后，接下来就是对于数据传输过程的管控，这也是传统合规中比较难覆盖的一块，大部分情况下，只能做到对源头数据产生方和目标数据接收方的管控，中间过程传输通常是管控不到的，也就意味着源头最终待传输数据和经过跨境传输到目标端接收到的数据是一致的。这个

特别对于源头和目标数据政策不一致的情况比较难处理，所以我们需要能在中间传输过程进行管控。

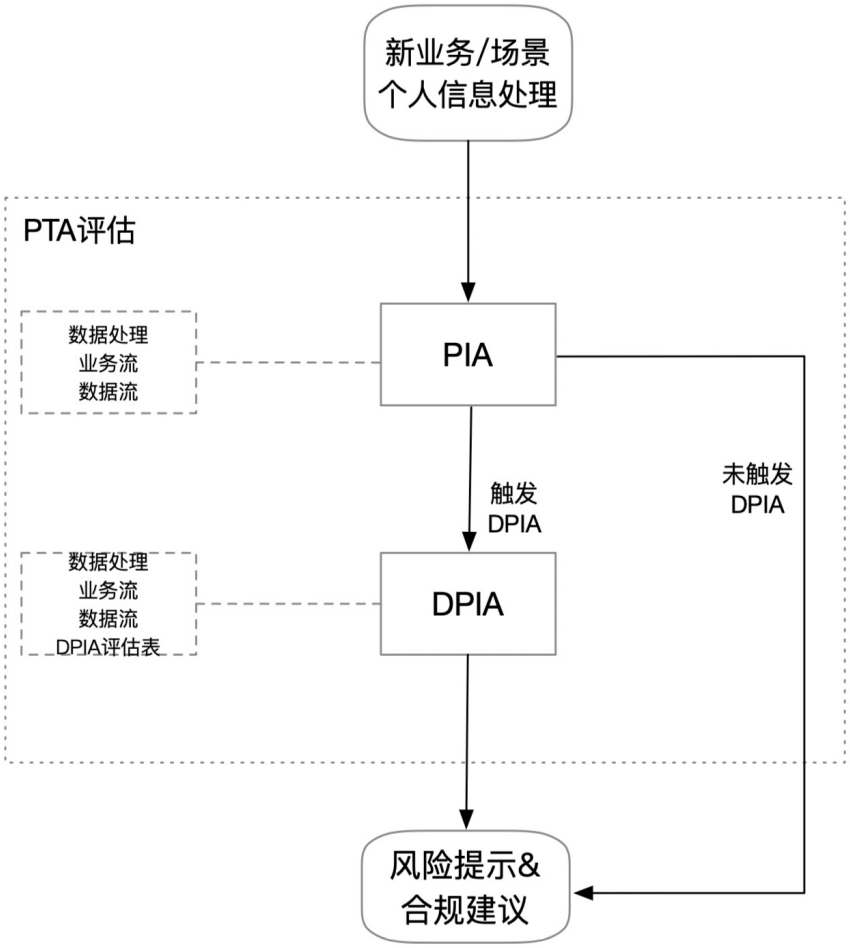


图 9-3 合规审计评估流程

对跨境过滤阶段的工具设计原则是对数据做切面，通过对常见的数据传输渠道进行卡口，具体在下面数据过滤章节会展开说明。如果当前技术受限，对于渠道的卡口无法做到技术上的限制的话（例如封闭的传输渠道无法进行切面卡口），可以考虑从流程管控上做一些卡口。

9.4.2 流程管控

整个合规可以分解为如图 9-4 所示，可以看到在每个环节我们都可以做到一定程度上的管控，不论是工具层面的强管控还是流程审批上弱管控，且是可灰度的。

仅限内部传阅

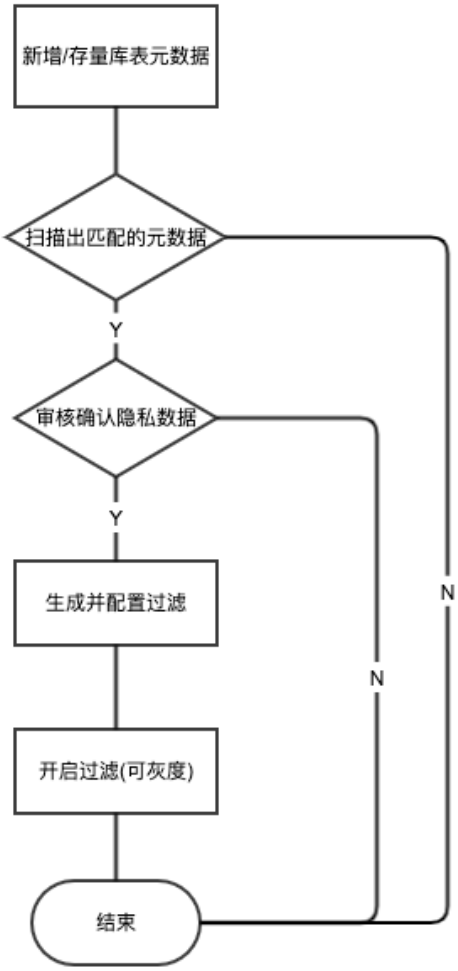


图 9-4 合规流程分解

9.4.3 数据过滤

数据过滤是合规中最重要的一部分，这部分也是工具方面最能助力的一块，对于庞大的数据和复杂的传输链路，如果想像人工审核一样处理，无异于天方夜谭，所以这块我们需要有一个合适的工具来协助我们完成。要做到数据过滤首先需要了解数据跨境传输的几大方式，如下表所示。

数据源	过滤	说明
MySQL	binlog 解析判断	通过 binlog 同步，可以解析 binlog，根据合规条件判断是否接收 binlog
ODPS	copytask 弱管控	copytask 任务流程上卡点管控
MetaQ	消息消费订阅判断	可以考虑加上一个消息消费管道
HSF	调用管控	在 hsf 控制面做分析，管控接口和出入参

9.4.4 数据删除 & 恢复

合规中涉及的另一个要考虑的方面是数据删除和恢复，在数据过滤后，过滤的只是增量新数据，对于存量数据，需要有一个解决方案来处理，否则对于这些存量数据还是会有合规风险，另一方面，对于一些识别为合规风险的数据，可能过了一段时间或者有新的合规政策出来后又是非合规风险数据，需

要对这些被过滤删除掉的数据做一个恢复。

① 数据删除

对于正常业务系统来说，数据删除是一个重要但技术上没那么复杂的事，评估好业务场景和风险，联系 DBA 或者自行删除即可，数据量大的情况分批低峰删除，但对于合规场景下的数据删除就没有那么简单了，前文提到，合规场景必然是关联到全球部署的，对于数据来说，也是全球存储，通过同步来实现数据一致性，这也意味着一个地区机房下的数据删除会连带着同步到其他地区机房，最终数据全部被删除，这显然不是我们的目的。所以数据删除需要解决的是如何在确保指定地区机房的数据删除情况下其他地区机房的数据还保留。

以 mysql 为例，mysql 数据的主从同步和跨地区的数据同步一般都是基于 binlog 来完成的，读取源库的 binlog，解析并写入目标库，这个就给了我们数据删除的一个切入点，如果源库的数据改动不产生 binlog 那就可以避免影响到目标库。基于这个特点，针对 mysql 库的数据删除我们就可以有一个初步的解决方案，就是在对库操作数据删除时不产生 binlog，当然这里只是针对符合合规删除条件的数据，正常数据还是需要开启 binlog 的。

② 数据恢复

数据恢复是另一个场景下的需求，对于那些被识别为具有合规风险需要删除的数据，可能在政策变动，或者一些误识别为合规风险而被删除的数据，我们需要有一个恢复的方案来保证数据能重新传输到目的库中。同样以 mysql 为例，对于过滤掉的数据，如何快速恢复过滤的数据，主要可以考虑如下几种方式：

- **第一**是从其他库同步过来，可以做到行数据级别，但这个需要考虑新老数据的实时性，以及同步后这些数据的后续更新是否正常。
- **第二**是通过 binlog 转存恢复的方式，对于过滤的数据，将这些 binlog 转存，恢复时解析对应的 binlog 并写到目标数据源中。

9.5 全球合规实施步骤

上述主要讲述了全球合规的架构和一些工具设计，有了这些架构指导和工具协助，我们就可以展开具体的合规实施步骤了。接下来我们就按照合规过滤的流程顺序来介绍下全球合规的关键实施步骤和每个步骤中需要关注的点。

9.5.1 隐私数据识别

全球合规第一个需要做的是对隐私数据的识别，特别是对于大量的存量数据，如何能快速识别到潜在的合规风险数据是需要解决的第一个问题。从目前普遍的做法来说，主要有字段正则识别，内容正则识别，内容解析识别等，如下表所示。

仅限内部传阅

识别方法	说明	优缺点
字段正则识别	对字段名进行正则匹配和识别，最简单的方法	优点：简单易配 缺点：无法根据内容来判定，存在误判的可能。
内容正则识别	对数据内容配置正则规则和识别	优点：基于内容识别，准确率相对较高 缺点：需要根据不同库和业务场景来制定正则
内容解析识别	对内容配置精确合规规则	优点：基于内容识别，准确率最高 缺点：计算逻辑复杂，针对内容解析，需要一定的计算和算法成本。

除了确定合规数据识别方法外，还需要对确定判定依据，也就是什么样的数据被认定为合规隐私数据，虽然前文提到了很多法律法规定义相关跨境传输，本地存储的限制，但对于明确的什么是个人数据还没有一个明确的法律法规定义，我们也只是基于这些法律法规给出一个相对符合认知的定义。进一步的，有了个人数据的定义后，哪些数据可以算作个人数据，例如，常见的手机、地址等肯定算个人信息，设备 id、cookie 这些呢，再进一步，业务属性较强的数据，例如订单、退单等数据是否算作个人数据。这些都是需要考虑的问题，特别对于技术来说，思考这些问题有的已经超出了技术范畴，需要法务合规的专业人士来看下了。

集团法务联合蚂蚁法务梳理了一版个人隐私数据分类，目前来看是相对比较完善的一版，部分如图 9-5 所示：

个人信息类别（I类）	个人信息类别（II类）	个人信息类别（III类）	字段编码 0720	字段使用注释	Remarks	个人识别标识	敏感信息标识（GDPR）	地区识别信息标识
A. 个人基本资料	A. 姓名		AA1	姓、名、持卡人姓名均归此类				
A. 个人基本资料	B. 生日		AB1					
A. 个人基本资料	C. 性别		AC1					
A. 个人基本资料	D. 民族		AD1				Y	
A. 个人基本资料	E. 国籍		AE1					Y
A. 个人基本资料	F. 地址	1. 详细地址	AF1	如存在无法具体分类的地址，均放入此类				Y
A. 个人基本资料	F. 地址	2. 国家/地区	AF2	电话号码国家、地址国家均归入此类				Y
A. 个人基本资料	F. 地址	3. 省/州	AF3					Y
A. 个人基本资料	F. 地址	4. 城市	AF4					Y
A. 个人基本资料	F. 地址	5. 经纬度	AF5					Y
A. 个人基本资料	F. 地址	6. 邮编	AF6					Y
A. 个人基本资料	G. 电话	1. 电话号码	AG1	包括手机号码和固定电话号码		Y		
A. 个人基本资料	G. 电话	2. 电话区号	AG2					
A. 个人基本资料	H. 电子邮箱		AH1			Y		
A. 个人基本资料	I. 家庭情况		AI1	家庭ID，配偶、子女及其他家庭成员信息，子女个数等				
A. 个人基本资料	J. 社交媒体		AJ1					
B. 联系人信息	A. 联系人姓名		BA1	通讯录				
B. 联系人信息	B. 联系人电话		BB1			Y		
B. 联系人信息	C. 联系人来源		BC1					

图 9-5 个人隐私数据分类

有了识别规则和依据，那就可以结合我们前文提到的工具来对数据进行扫描和分析。

9.5.2 合规风险确认

在扫描分析前，还需要和专业的法务合规团队再确认下最终的识别规则和数据，防止出现误识别的情况。通常情况下生成的规则和数据是偏技术方面的，需要和法务明确规则的语义、数据的使用场景、命中的合规规则等，减少双方理解和沟通上的差异。

9.5.3 合规扫描 & 规则生成

法务合规确认没问题，接下来就是利用这些定义依据来对数据进行扫描，扫描可以有多种手段，如果有建离线数仓存储数据源的元数据，那是比较方便的方式，可以直接通过离线任务完成扫描，如果是正常的元数据，也可以通过一些技术手段完成扫描。

扫描元数据不是最终目的，一般的合规流程中法务得到命中隐私字段的元数据可能也就目的达成了，但我们需要的不仅仅是这些，数据在跨境传输过程的管控也是我们关注的，所以我们需要根据这些扫描出来的字段生成对应的规则。

9.5.4 合规开启

前面讨论了数据的扫描和规则生成，最终产出确定的合规隐私字段和生成规则后，接下来就是重要的开启阶段了，根据不同的数据源有不同的开启方式，举个例子，对于 MySQL 的数据源，一般是配置好 binlog 的解析插件。插件中读取对应的过滤规则，判断是否需要灰度。对于其他的数据源，同样的操作方式，通过上文提到的数据卡口来设置插件判断。

从软件研发和安全生产经验来看，任何一个变更需要满足可灰度原则，否则很容易出现线上故障影响正常业务使用。合规规则的开启也是一种线上变更行为，所以这块需要满足可灰度原则，在实际操作过程中，可以针对数据过滤的具体库表来设计灰度规则，最常见的做法是因为过滤一般涉及到用户个人，所以我们可以通过对用户来进行灰度，逐步控制灰度比例。

9.5.5 合规验证 & 审计

在实施的最后一步就是针对合规结果的验证和审计，主要是从两方面来进行，一方面是从技术层面的验证，需要看下数据在跨境传输过程中是否正确处理，在接收方的数据是否满足业务和技术需求。另一方面是从法务层面的验证，过滤后的数据以及数据使用形式是否符合接收方的数据合规政策。

9.6 俄罗斯合规实践案例

本章节作为主题的最后一部分，主要结合上面提到的架构设计原则和工具，介绍如何解决我们实际遇到的合规问题。

背景

AE RU 收到俄政府通知，俄监管机构 RosKomNadzor 将对 AER 进行个人隐私数据合规审计。主要以下要求：

- **本地存储**：所有个人隐私数据必须首次在俄罗斯存储。

仅限内部传阅

- **跨境传输**：在本地化存储的基础上，须符合处境合规要求。具体的合规要求可以参考上文讲述的俄罗斯跨境从传输要求。

应对

- **本地存储**：AER 团队负责对数据进行本地化存储改造。已经解决。
- **跨境传输**：分析当前 AE 全球部署架构，调整部署架构满足跨境传输要求，同时采取合规手段过滤数据保证数据安全。

怎么做的

全球部署架构的升级细节我们就不展开讲了，在升级到新架构的基础上，我们针对合规需求，根据上文讲述的几个指导原则和流程，展开了一系列工作，关键的几个步骤如下：

■ **定义隐私字段**

根据法务给出的隐私字段定义，结合实际业务场景，我们定义了符合 AE 的隐私合规定义，如图 9-6 所示：

规则名	规则版本ID	规则引擎类型	规则类型	规则内容	规则分类	描述
个人资料电话规则	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	个人基本资料–电话–电话号码	个人资料电话
个人资料社交媒体	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	个人基本资料–社交媒体–社交媒体	个人资料社交媒体
网络cookie规则	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	网络身份识别信息–Cookie–Cookie	网络cookie规则
设备id规则	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	设备信息–设备识别码–设备识别码	设备id
网络ip地址规则	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	网络身份识别–IP地址–IP地址	网络ip地址规则
会员账号头像规则	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	会员信息–账号头像–账号头像	会员账号头像
会员账号昵称	[redacted]	FIELD_NAME_REGULAR	COLUMN	[redacted]	会员信息–登录账号和密码–账户昵称	会员账号昵称

图 9-6 定义隐私字段

可以看到，这些规则主要集中在个人信息，社交、网络设备等数据上，这些也是电商场景下比较常见的个人隐私数据。

■ **扫描**

有了明确定义好的字段规则后，就是针对元数据的扫描了，因为业务的元数据都存储在离线数仓中，所以我们直接扫描数仓中的元数据，并得到需要过滤的库表字段。部分如图 9-7 所示：

库名称	表名称	字段名称	命中规则
houyi_operation	houyi_eventcenter_data_field	data_source_code	非敏感
houyi_operation	houyi_eventcenter_data_action	data_source_id	非敏感
houyi_operation	houyi_eventcenter_data_source	data_source_name	非敏感
houyi_operation	houyi_eventcenter_data_source	data_source_show_name	账户昵称
houyi_operation	houyi_eventcenter_data_source	data_struct	非敏感
houyi_operation	houyi_eventcenter_data_source	data_type	非敏感
houyi_operation	houyi_eventcenter_template_v1	datasource_id	非敏感
houyi_operation	houyi_sme_instance	device_id	唯一设备识别码
houyi_operation	houyi_sme_instance_index_device	device_id	唯一设备识别码

图 9-7 数据合规扫描

- **开启**

为了保证数据过滤可灰度，我们针对用户级别的数据设计了灰度规则，按照用户 id 百分比开启过滤。

最终合规层面上基本解除了俄罗斯个人数据跨境传输风险，涉及到相关隐私库表数据开启了过滤，确保俄罗斯用户数据不到美国。

仅限内部传阅

十、成本优化

10.1 资源成本优化背景

随着 AE、Lazada、Daraz、淘海外等国际业务的迅速发展，应用规划已经超过 2k+，新 / 旧系统改造、业务融合、组织架构调整等原因，很难简单直观的了解 资源成本使用情况，评估其合理性。与此同时，技术资源费用上很高，同比业界其他国际电商 成本偏高的现状。

在业务高速发展同时，对于某些场景点也有 做一些成本优化项目治理，虽然一波操作在技术资源成本下降上有一定的效果，但是持续性差，随着人员流动 一些策略措施 / 方法 沉淀不足，缺乏有效的架构理论指导。从整体意识上看，也会发现业务研发同学 在高业务增长下，更多关注在业务目标的实现上，面向低成本架构思考不多，资源成本理念不足。

10.2 成本优化基本框架

10.2.1 整体框架

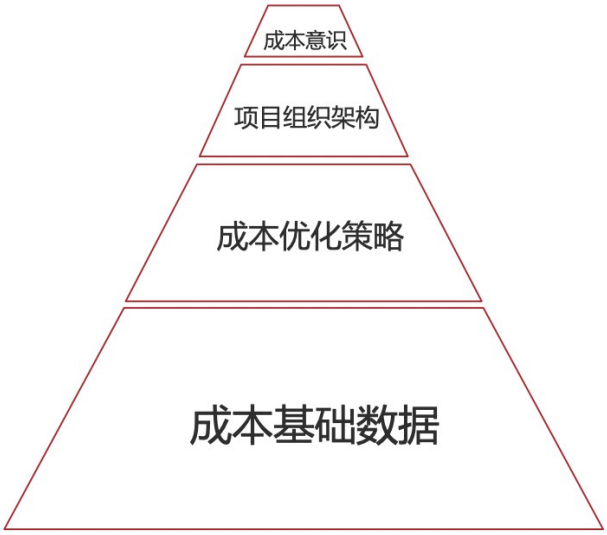


图 10-1 成本优化整体框架

针对业务背景已经相关问题，资源成本架构整体划分为四个部分，如图 10-1 所示：成本基础数据、成本优化策略、项目组织架构、成本意识。前一部分内容，来支撑后续部分的工作。

- **成本基础数据**：能够将资源对于成本花费明确化，帮助业务人快速了解到自身的成本情况；
- **成本优化策略**：如何更好的使用资源，以及相关低成本技术选型方法的沉淀；

仅限内部传阅

- **项目组织架构**：能够有效的落地执行相关成本优化策略；
- **成本意识**：提升整体研发人员对于技术资源成本的感知，和节约意识。

① 成本基础数据

成本基础数据的可视化，其实是在做整个成本架构第一个考虑到的事情。对于一些降本场景，有时候**让业务人员看到成本就是很好的降本措施。**比如，Lazada 业务中有个应用挂了二十台空余的 ECS，由于当时使用完后后续不再需要，但是忽略了释放处理。当成本数据外化后，马上发现了这个问题，处理就非常简单了释放资源即可，当月即产生客观的降本。

基于客观数据的降本目标比主观判断更加有效。有些降本措施上 是根据人的经验 来进行的，一般经验确实都是对的 能够降低成本，但是效果 往往存在偏差；例如 做一个老应用下线 工作量比较大，最终直接降本结果不明显，当然可能存在更长久的价值；但 对一个大流量应用 规范日志 打印，工作量小很多，但降本效果上非常明显。如果想在 有限的资源下，更好的 达到降本效果，客观数据可以有效的协助决策目标与策略。

基于数据驱动的降本 能够发掘出被忽略的地方。业务的重要性 和 资源的使用上，有时候并非正比。举个典型的场景，图片服务的资源型应用，因为服务比较简单、平时稳定，相比与变化多样业务应用关注度一般不高，但由于流量较大资源往往很充足。所以 研发人员主观评估资源合理更多的聚焦在了业务应用，但就降本而言资源应用也是应该重点评估。

综合来说，资源成本基础数据 是整个成本优化的基石，基于数据驱动的降本可以达到更全面、有效的降本优化，让成本可见就是很好的降本措施，如图 10-2 所示。

图 10-2 成本基础数据示例

② 成本优化策略

成本优化策略 是 武林秘籍，这部分要解决的问题是 当发现成本主要花费，有什么方法、手段去进行降本？一般实践而言分为两大类。一类是 直接措施上：比如，优化大流量日志打印来节约 SLS 成本；评估清理大库 DB 的历史数据，降低存储成本；对于容器使用过多，分析 CPU 等资源利用率 合理评估

仅限内部传阅

数量等。二类是 设计架构调整上：是否应该使用存储成本更低的 Lindorm；应用是否架构合理，应该同其他合并；能否减少某些链路 tair 调用降低峰值等。

除开上述手段、方法外，这部分架构设计上 还希望 **对于这些手段 能沉淀出 标准**。举个例子，目前我们在国际化中台容量评估过程中，沉淀了如下 结合 cpu 和 load 使用率，进行阶梯的缩容初步标准。

*cpu 峰值 >40% || load 峰值 >cpu 核规格 *1.5 的分组 (不缩容)*

*cpu 峰值 <=40% && load 峰值 <=cpu 核规格 *1.5 的分组 (缩容 10%)*

*cpu 峰值 <30% && load 峰值 <cpu 核规格 *1.5 的分组 (缩容 20%)*

*cpu 峰值 <15% && load 峰值 <cpu 核规格 1.5*0.7 的分组 (缩容 50%)*

整体来说，能够让这些策略 更客观、更方便的执行，能够让没有经验的人快速学习了解到降本方法。

③ 项目组织架构

如何保障好的技术降本落实，项目组织架构是重要的保障。从国际成本战役的经验看，首先，应该从 KPI、OKR 进行顶层设计，特别对于设计多个 BU、团队协同工作的场景 是 目标一致的重要保障；然后，需要有合理的项目目标设定，这里就需要结合当前业务场景、数据来看，比如有些业务团队 技术主要费用是 DB 甚至 80% 的是某个数据库、表，项目目标更多的围绕这个资源来进行 评估策略可行与改造成本等（TOP 策略），更多的建议目标结合 技术资源花费 来定，而非 直接财务费用。最后，对时间节奏 Deadline 的合理设置。

项目整个推进过程中，更多可以结合 成本基础数据 去监管 是否与预期一致，保障整体目标的达成。同时，也可以对于 成本基础数据的表现做一些调整，比如 SLS 治理上，先前主要围绕存储治理、但发现其实 索引费用比例也很高，因此对于索引也可以进行一波治理，来达到预期 SLS 降本 XXX 项目目标。

④ 成本意识

什么是成本意识？举个例子，一个会议室内，几个研发相互之间对于某个技术架构方案激烈争论。有的说这个服务不应该归属于 XXX 域，这样并不合理；有的提这样链路会增加了两次的调用量 RT 会明显上升；也有的 数据表这样设计 后续 XXX 功能 支持不了了等等；这其实背后隐藏的是研发人员 思考意识，领域意识、性能意识、可扩展性意识等等，这些意识作用在工作中会习惯性的去思考这些方面。如果有人说 这样设计会增加一倍成本，显然这就是成本意识的实际显化。

从当前现状来看，业务研发更多技术方案设计的聚焦在架构、业务功能上，成本上讨论或者考虑的较少；要想解决新方案、新设计成本问题，成本意识可见是不可缺失的一部分。所以在整个基础框架的最顶层设计 是 成本意识。应当 有一定的宣讲、知识传递、培训、产品呈现等手段，提升 业务人员的成本意识，解决未来成本问题以及促进当前成本治理都能够起到关键性的作用。某种角度说，这里的 “成本优化” 章节也是成本意识具体实操的一个措施之一。

10.2.2 持续降本架构

在上一部分中主要阐述成本架构框架各个部分的设计与内容,这里主要要描述各部分之间的关系协助,构成持续降本的架构目标。核心的理念是：**打造基于数据驱动的核心降本闭环。**

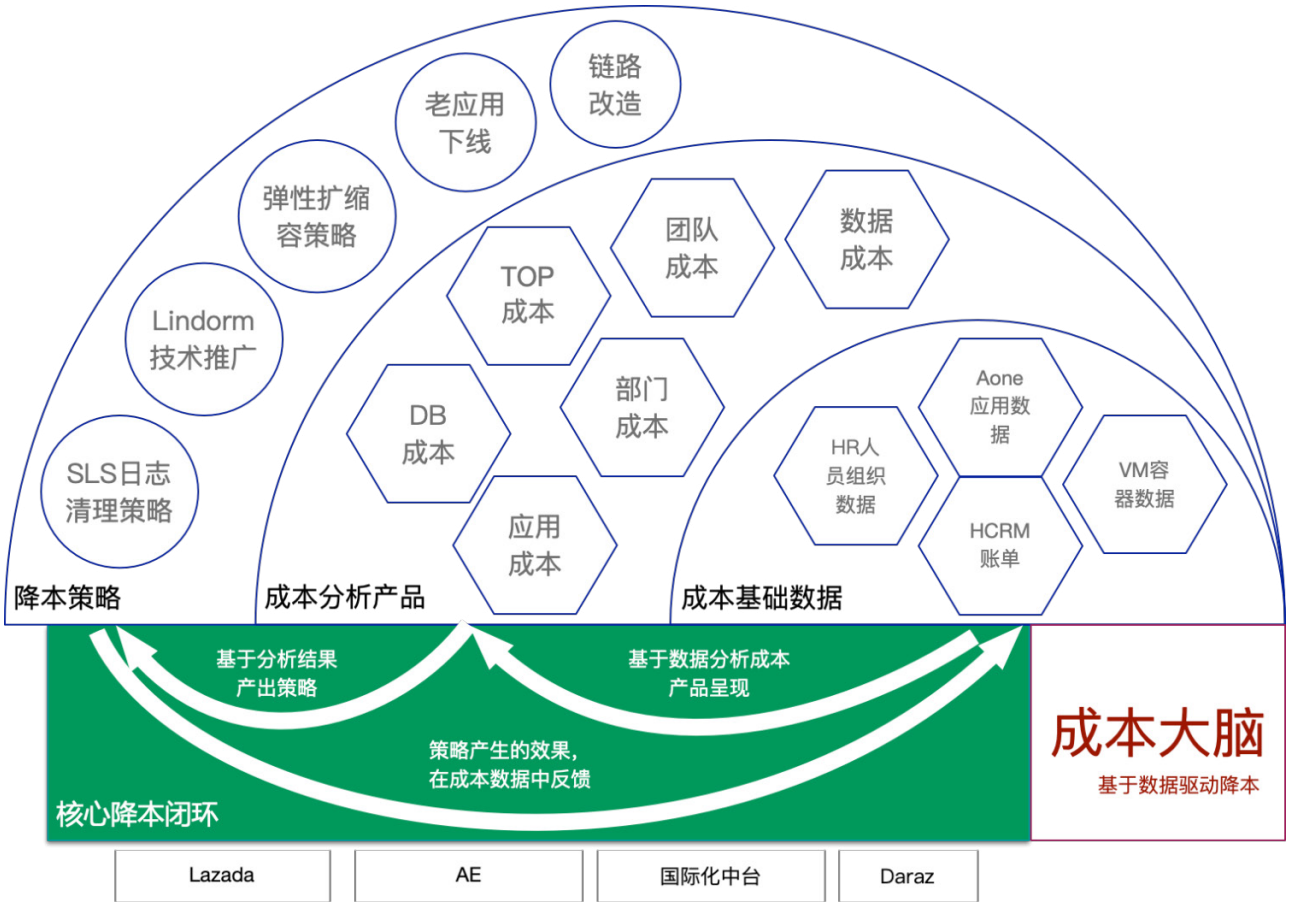


图 10-3 成本优化架构

成本优化架构如图 10-3 所示，根据成本基础数据（包含不限于：成本账单、HR 人员组织架构、应用容器数据等）来打造一系列成本分析产品，比如：以团队维度成本组成分析，用于大团队在选择降本降本项目执行团队上做决策；团队内，以技术资源产品分析 用于明确 主要成本消耗 来抉择相关降本策略；关联应用、Owner 相关数据 用于决策策略执行归属等；根据这些分析产品 明确相关降本策略目标，构建项目。再以项目组织形式 落地实施执行，最终项目执行过程 、最终复盘 也可以在基础数据中反馈出来。从而构成一轮降本闭环，每完成一轮闭环 即 完成一次降本循环体系，持续的进行管控。这一套体系 基于国际化中台、AE、Lazada 实际项目执行推动中，提炼设计而成。

10.2.3 成本基本原则

① 成本分布 TOP 原则

在海外降本的第一阶段实施完成后，我们对涉及 2411 个应用、数十万容器进行了一次应用成本数据分析统计，其中 204 个应用（TOP 8.46%）约使用了 80.04% 成本。即使，对相关数据分 BU 后统计，也同样的获得了类似的分析数据。同样类似的分析云账号资源，也存在相仿的规律。综合看，

仅限内部传阅

TOP10% 的应用占用并使用了 80% 的资源成本。图 10-4 是 TOP 部分应用的成本分布图。

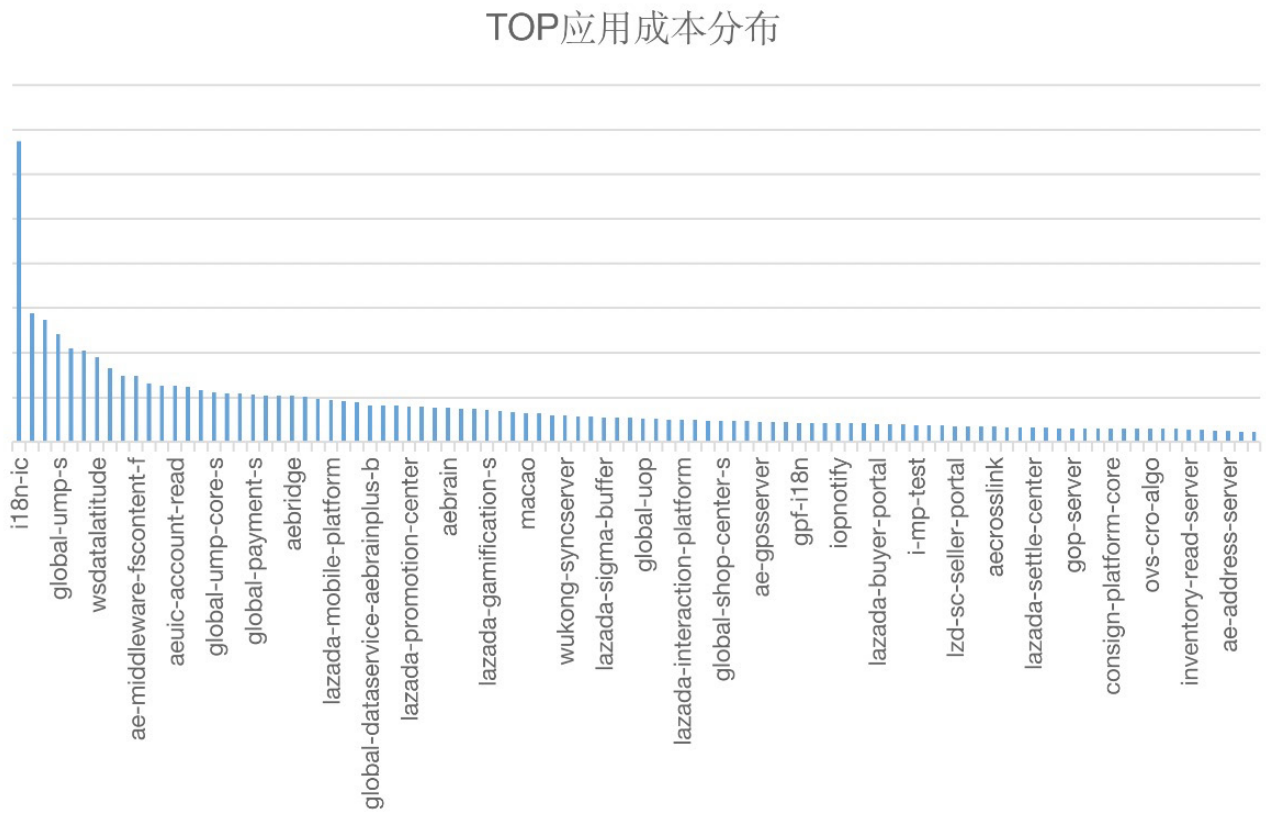


图 10-4 TOP 应用成本分布

所以，对 TOP 10% 的应用进行严格管控，降本也好、资源节流都具备了至关重要的战略地位。这一点其实也容易解释某些场景应用优化后发现，其实对于整个部门来说应用技术资源费用上变化并没有很多，投入产出比不高。如果想少量措施来拿到好的成本优化结果，不妨多关注下 TOP 应用，也许会有效果。

当然，我们也要从另一方面来看待这个规律。虽然应用直接资源的使用上，治理长尾应用 并不能很好的降低部门总资源成本，但是长尾应用 背后带来的 维护成本、稳定性保障、人员成本等，也是值得思考的。以国际平台降本提效经验看，治理长尾应用对于人员提效而言，性价比往往极高。

② 成本与效率关系

成本与效率无必然关系，如果在分布式容器管控的业务场景中，容器越少那么系统单机资源使用率就会越高也就是效率越高，同时成本越低。姑且在不讨论 缩容同时 人力投入、稳定性影响、限流调整复杂性等其他效率体系上，机器使用效率也许越高。换另外一场景，ADB 与 XDB 选型，ADB 性能上具备很多优势 一些复杂场景下效率一般高于 XDB，但资源成本费用上 高于 XDB，这么看来成本越高 带来了 效率的提升。我们更应该的 还是将成本、效率作为两个技术方案指标来独立看，如果不是在特定场景、各种限制条件下 来说 两者的关系并非合适。

③ 成本与稳定性关系

成本与稳定性无必然关系，直观上看 成本越高 稳定性越好，实践看并非如此。

在一个链路优化降本中，通过整合缓存设计 降低 tair 峰值整体 QPS，改造后对于 一个服务 tair 多次调用变为一次调用，改造后受链路简化影响原先监控毛刺有显著降低 稳定性更好，成本也有大幅下降。可见在这场景下，方案链路复杂程度决定稳定性，越简单稳定性越强、成本越低；当然也有另外的例子，大促分布式容器上，与上一个例子认知相反。分布式设计的系统中，容器越多 对于 但单容器故障问题鲁棒性越强，成本也越高。

综合来说，成本与稳定性两者关系并没有盖棺定论。稳定性更多是在技术、部署方案上考虑分析出来的指标，成本同样也是技术、部署方案上综合资源花费的指标。并不适合 两者建立关系来评估，而是应该从技术方案上对两部分进行分别评估。

④ 资源成本与财务结算

关于资源成本与财务结算的关系，这里提到的资源成本指技术上对于生产资料的消耗（CPU、内存、硬盘、带宽等等）和 中间件使用计量；财务结算 指代每个月初财务对技术资源进行对帐、预算核对，然后与关联提供资源 BU 进行结算。比较直观的看，两者之间一定是等价关联的 应该是一致的，这样认识并没有错，但实际情况下，并不完全如此，原因如下：

首先，对于财务结算计费系统而言，Pouch、DB 也好本身就存在市场价格波动的可能性，使用完全相同的资源在不同财年、季度都可能存在费用的波动；同时 计费规则也可能存在调整，所以相同资源消耗，不同时间上财务结算就存在不同，很明显这种成本的 增加 / 降低 与 技术资源上的优化是无关的；然后，财务计费系统本身而言 动态发展的角度看待，目前并没有完全成熟，举个例子 因为某些原因今年 10 月 tair 结算合并到了 11 月账单中，资源成本角度看 10 月明显是有使用 tair 资源的；还有，财务结算计费中 对于一些特殊场景上 可能存在偏差，比如 某个负责人转岗部门，可能在一段时间内 资源挂靠关系没有完成转移 团队资源使用计费上 就存在遗漏等等；综上，可以看到资源成本和财务结算实际场景下无法做到完全一致。但是，从长期价值角度看，无论价格波动也好、特殊场景也好、计费临时调整，最终效果 资源成本上的优化 必然能带来财务结算上的节省。

基于此，结合到实际的 技术资源降本项目目标制定中，如果 直接用财务结算作为目标，会带来很多核对财务数据的工作量，并且 带来了更多的非项目的不确定性。技术资源降本项目，更多的工作应该聚焦在 资源成本优化带来的长期 成本节省效益，技术资源使用策略等方面，对于此 使用资源成本同财务同学核对简单点的 评估公式作为目标更合适。举个例子，降低容器成本，降本项目中 可以按照 A* 减少容器数 来作为 直接降本目标，A 为同财务同学核对的数值，这样更加能将项目目标聚焦到 提高技术资源利用率上来。

仅限内部传阅

另外一方面，对于优化财务计费系统而言，利用实际场景资源成本和财务结算差异特性，可以反向对帐 财务结算数据 让财务结算计费逻辑 规则 更加合理准确。

10.3 技术资源计费规则

10.3.1 容器计费

谈到容器计费必须先了解下容器的计费组成 (项)

① ECS 实例 (CPU+MEM): 以实例规格的形式提供 ， 包括 vCPU 和内存，收取实例规格费用，例如 4C8G、8C16G 等规格。

② 块存储 (DISK): 按云盘容量和使用时长收取费用。

由于国际化下应用的基线规格是不尽相同的，不同应用会出现 (CPU, MEM, DISK) 各种各样的排列组合。且随着 VPA 和 HPA 技术的展开，容器的容量和自身基线规格也可以灵活伸缩扩展，因此当前的容器收费通过 " 按使用量与使用时长 " 收费的标准执行。其通过计算公式如下 :

总费用 = (CPU_SIZE*CPU 单核价 + MEM_SIZE*MEM 单 G 单价 + DISK_SIZE*DISK 单 G 单价)* 使用时长

通常我们是以 4C-8G-60G 的容器规格进行原子化描述单位容器做费用预估。

10.3.2 日志计费

SLS 日志收集服务会按照具体使用到的功能细分为多个收费项 (名称 / 单位):

- **存储空间大小**：存储空间为压缩后原始数据量与索引数据量之和
- **索引流量**：索引流量根据实际索引字段进行计算。在写入时一次性收取存储费用。
- **读写流量**：读写流量根据传输的流量计算，传输流量为压缩后的大小。
- **活跃 Shard 租用**：Shard 租用只统计当前读写 Shard 的数量。已经被 合并 / 分裂 的 Shard 不收取租用费。
- **读写次数**：日志写入日志服务的次数，由您的日志产生速度决定。后台实现机制会尽量减少读写次数。
- **外网流量**：被外网程序读取消费所产生的数据流量。

10.3.3 缓存计费

缓存以 Tair 为例，计费方式主要根据规格计算 :

MDB/LDB 规格数 = MAX（ROUNDUP（已使用容量峰值 / 最小规格容量），ROUNDUP（已使用 QPS

峰值 / 最小规格 QPS），ROUNDUP（已使用吞吐量峰值 / 最小规格吞吐量））

RDB 2.0 规格数 = MAX（ROUNDUP（申请容量 / 最小规格容量），ROUNDUP（申请 QPS/ 最小规格 QPS），ROUNDUP（申请吞吐量 / 最小规格吞吐量））

10.3.4 数据库计费

数据库计费以云 Mysql 为例，主要是实例规格、CPU、内存、连接数、IOPS 的方式进行收费；同时有按月计费、或者按量计费；另外，存储空间也分为不同的版本。

10.3.5 大数据计费

MaxCompute 计费规则

包年包月计算资源包括 SQL、MapReduce、Spark 等类型任务计算所需资源。包年包月的计算资源有两种：

- **预留计算资源**：包括**包年包月标准版**、**包年包月套餐版**中的计算资源，购买后，MaxCompute 将会为您预留购买的资源数量，您将可以完全独享这部分资源。
- **非预留计算资源**：购买后，不预留固定数量的资源而是提供一个数量在 [0，购买量] 范围内的弹性资源池。当任务发起后，MaxCompute 会在总资源池的空闲资源部分进行抢占并分配。若总资源池中空闲资源充足则可能分配给您所购买的最大 CU 量，若总资源池繁忙则可能分配到 0CU，MaxCompute 可以保障您购买的非预留计算资源全天 50% 的计算力（一天计算力 = 资源数量 *24 小时）

10.4 成本优化策略

技术成本一直以来容易受到大家的忽略，经常发生在以下场景：

- 开始一块新业务或者创建新应用时无法准确评估需要承载业务量的技术资源。往往都是靠人为拍脑袋决定，没有相关数据的支撑。
- 正在提供服务的应用，自身的容量、服务能力和系统水位表现是千差万别的。缺乏其合理性的检查与反馈机制。
- 一些技术创新手段、链路与架构优化会直接或间接影响技术资源的供需，但是这个成本优化的过程当前无法量化。
- 国际化电商业务各种促销活动开展频繁，应用维护者通常会将容量保持在较高的大促状态，当大促结束后通常也不回将不必要的机器下线掉，这会造成在日常态大量的浪费。
- 在大部分情况下，容量是存在浪费情况的。一是业务同学的成本意识较为薄弱；二是容量规划体系不完备，存在 " 超用滥用 " 情况。在整体 " 成本高利用率低 " 的背景下，成本优化已经迫在眉睫。

仅限内部传阅

10.4.1 成本数据存储选型

国际化主要选型 Lindorm 作为成本数据存储，主要原因如下：

- Lindorm 是一款适用于任何规模、多种模型的云原生数据库服务，支持海量数据的低成本存储处理和弹性按需伸缩，提供宽表、时序、搜索、文件等多种数据模型。
- Lindorm 融合了阿里巴巴过去十年在大规模宽表、时序存储等领域的技术能力和经验，同时今年重点在云原生、多模融合、低成本等方向创新突破。
- 在 Lindorm 全面拥抱云原生过程中，看到了基于云存储技术多样化的块存储、对象存储等原子存储能力。
- 数据往往呈现出一定的活跃周期规律。将访问频率低，生命周期长的的数据沉淀到低成本存储中，并辅以更高压缩比的压缩策略来降低总体存储成本。

10.4.2 容量评估策略

解决容器的成本，核心是解决容量。基于给应用集群进行合理的容量评估，我们摸索出了多种容量评估策略

应用画像描绘

不同应用在接受同样流量处理时的整体集群性能表现是不尽相同的。应用画像皆在为不同特征的应用进行行为与表象的抽象描绘。

- 对应用历史上各类指标时序数据进行无监督学习，屏蔽上层业务的干扰，获取应用的不同类别特性，这些特性有
  - 时间（早高峰晚低谷）
  - 计算类型 (IO 密集型或 CPU 密集型）
  - 服务能力 (QPS 高、RT 低）
- 提取特征后，将其进行聚类转化为标签，再通过监督学习通过历史指标分析，给应用进行聚类打标，这样每个应用的系统特征就显性出来。

系统水位预估

- 根据不同类别标签的应用，我们会有不同的容量评估策略，例如 CPU 密集型应用，会以当前应用的 cpu 和 load 水位进行评估
  - CPU 利用率 40%
  - LOAD1 为 CPU 核规格的 1.2 倍
- 当系统水位处于较低水平时将对集群容量进行缩容

  cpu 峰值 >40% || load 峰值 >cpu 核规格 *1.2 的分组（不缩容）

  cpu 峰值 <=40% && load 峰值 <=cpu 核规格 *1.2 的分组（缩容 10%）

  cpu 峰值 <30% && load 峰值 <cpu 核规格 *1.2 的分组（缩容 20%）

  cpu 峰值 <15% && load 峰值 <cpu 核规格 *1.2*0.7 的分组（缩容 50%）

- 当系统水位处于较高水位时，将对集群容量进行扩容

10.4.3 无状态应用治理

为了能够快速、安全、稳定地对容量进行变更，前提是应用必须 " 无状态 "，因此必须要先进行无状态应用治理。

无状态应用的定义：在应用集群负载承受能力范围内，如果一个应用集群可以在任意时间执行任意程度的扩缩容动作。并且针对一次请求，可以随机访问到该集群内任意一个实例且处理结果是完全一样的，那么此应用称为 " 无状态应用 "，否则为 " 有状态应用 "。

" 有状态应用 " 特征：应用升级或者灰度时对启停顺序有强要求；存在对机器信息的依赖，如 IP、MAC、SN 等信息；集群内不同实例内置的相关缓存数据不同。

10.5 成本优化展望

结合国际化成本优化的不断探索，对于未来说可以在成本分析、成本监控、成本预期方面进行进一步挖掘。结合过往经验看 部门维度、团队维度的成本分析 还不够，细力度的分析可能更能够发现问题，以容器为例说明：目前还只是分析到单个容器天维度信息，如果细化每天使用时 部分高峰应用 可以考虑峰时自动扩容 策略来降低成本等。当涉及到 数千应用、数十万容器时，全部项目研发成本评估是不现实的，成本监控也变得越发重要 当成本异常增长时 可以主动监控报警出来，更加动态的对成本进行管控。对于 未来成本的估算，面对业务不确定性如何用历史确定性（DAU、GMV 增长率等）的数据进行 推迟预估 是个复杂性命题，需要长期探索，更好的评估预测成本也是 成本可控的重要一方面内容。

仅限内部传阅

十一、弹性

11.1 弹性的核心是资源

弹性能力体现了一个系统是否能够随着压力的增减而动态的对资源进行调整的能力，是技术架构中重要的话题之一。在进一步讨论弹性之前，首先需要说明的就是资源问题。因为国际化业务的多样性、不同业务对机型的特殊需求，底层资源可能有多个；比如专门服务电商业务的资源池，服务中间件的资源池，服务搜索推荐的资源池，服务计算平台的资源池。各个资源池相互独立，导致整体资源的使用率不高，比如双 11，计算平台的业务可能会有降级，但是资源池隔离，并不能给电商业务使用，造成了资源的浪费。

在这样资源的大背景下，期望可以将电商业务，搜索推荐，在线离线业务资源池合并然后做全局的弹性计划，让资源使用更加合理，这就是弹性的核心思想。在这种思想的指导下，弹性的方向主要是在统一资源池与统一调度基础上，做全局的弹性规划，更合理的利用物理资源。

- **统一资源池**：将多个业务的资源池统一到一个资源池，比如统一全部导入到集团的 ASI 资源池（Alibaba Serverless infrastructure）中，如图 11-1 所示。

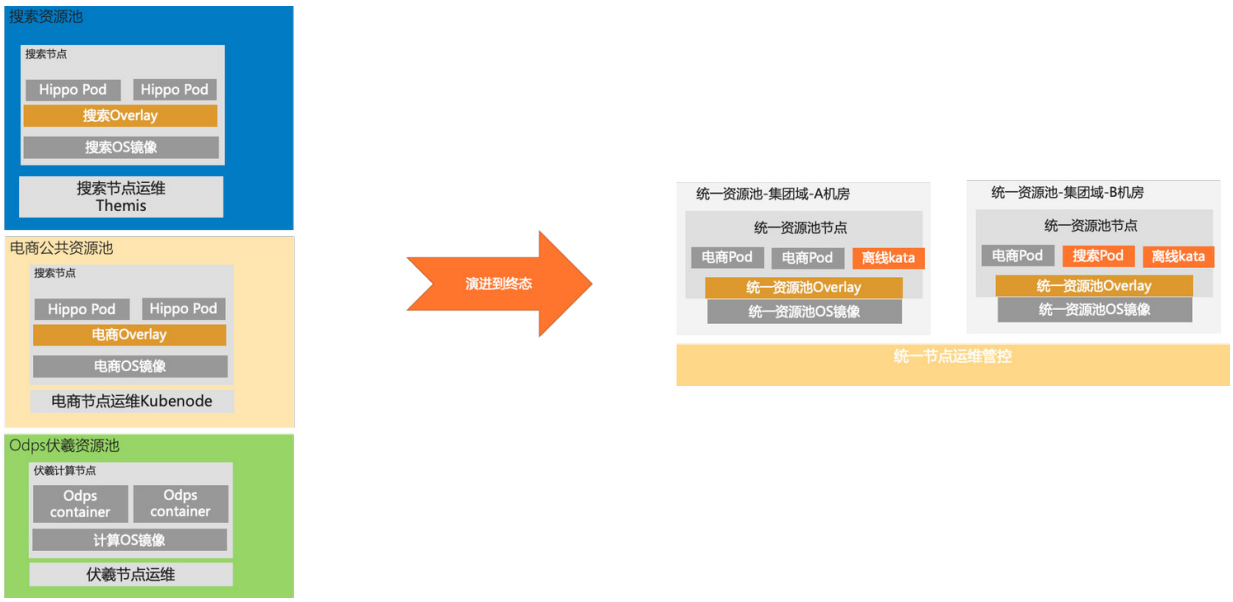


图 11-1 统一资源池示意图

- **统一调度**：将电商，搜索，计算平台的之前各自的调度器进行统一，满足多种场景，然后再根据业务需求，进行相关的资源调度，如图 11-2 所示。

仅限内部传阅

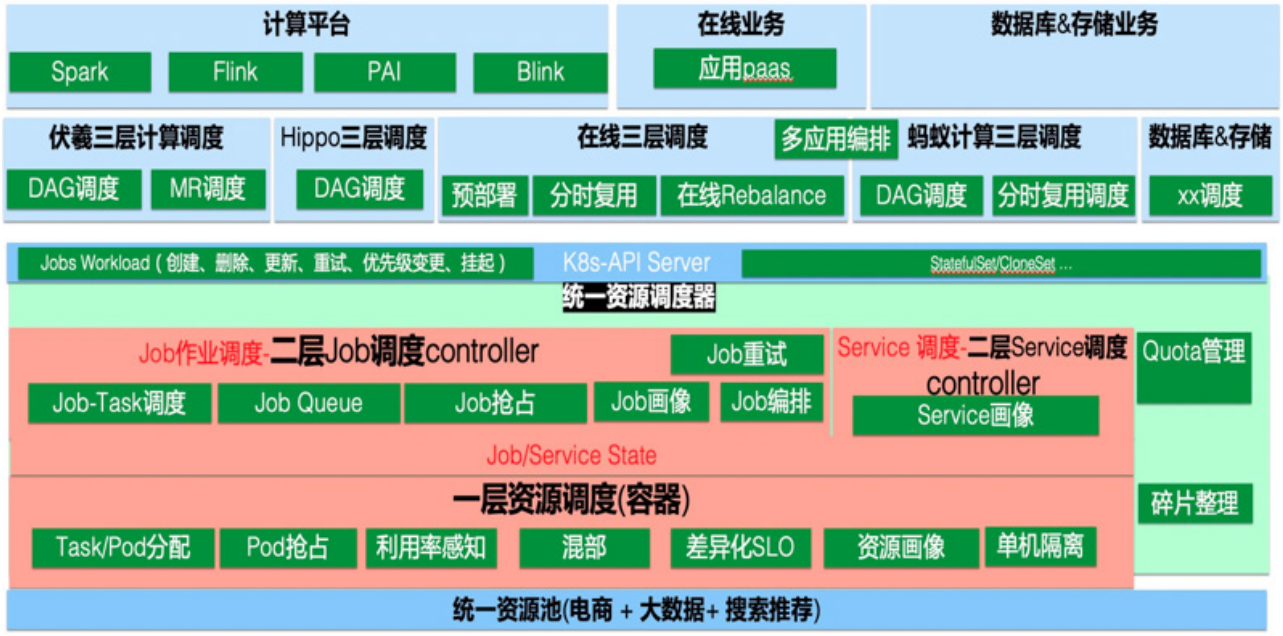


图 11-2 统一调度示意图

11.2 体系化弹性

在国际化场景业务环境中，会发现一些成体系化的扩缩容场景，通过批量应用弹性技术来达到一定的弹性目的，在此定义称之为“体系化弹性”。先举例来看，在德国建站过程中，整体核心应用最小闭环已经完成部署。随着灰度流量日益完善，推动全体应用逐个扩容效率很低。因此结合业务链路、应用信息，应用弹性技术批量进行处理可以极大的节约了建站效率人力。再者，目前整个国际站基于单元部署，受到合规、本地运营等业务、政策影响，可以应用批量规模化弹性的，对站点单元进行快速搭建部署，正在讨论的 Lazada 越南站可能就会使用到。另外，在大促场景中结合入口流量的变化，对于整条线关联业务的应用体系化扩容也是个经典 Case。这些都是体系化弹性具体的例子。

这些体系化弹性应用场景的背后，其实是国际化单元部署架构的支持。**体系化弹性技术落地关键，在于能够清楚梳理出业务应用之间的关系，进行批量扩容的同时兼顾上下游相互之间依赖容量的需求，各种服务资源之间的配比。**其中各种容量、应用、服务量上的数据来源，基于整体站点业务的监控体系数据。同时，稳定性上，国际化中台的区域切流、单元切流的基础能力技术，保障了体系化弹性实施的系统稳定性。

就整体而言，目前基于国际化架构部署体系化弹性也还是处于探索阶段，不过随着业务、合规、大促的发展在未来会变得更加方便与成熟。

11.3 容器弹性

回到弹性的最小资源单位容器来看，用简单公式表示一个应用的容器资源消耗：

仅限内部传阅

容器资源消耗 = 数量 * 规格 (CPU 核数 | 内存量 | 磁盘量)* 在线时间。

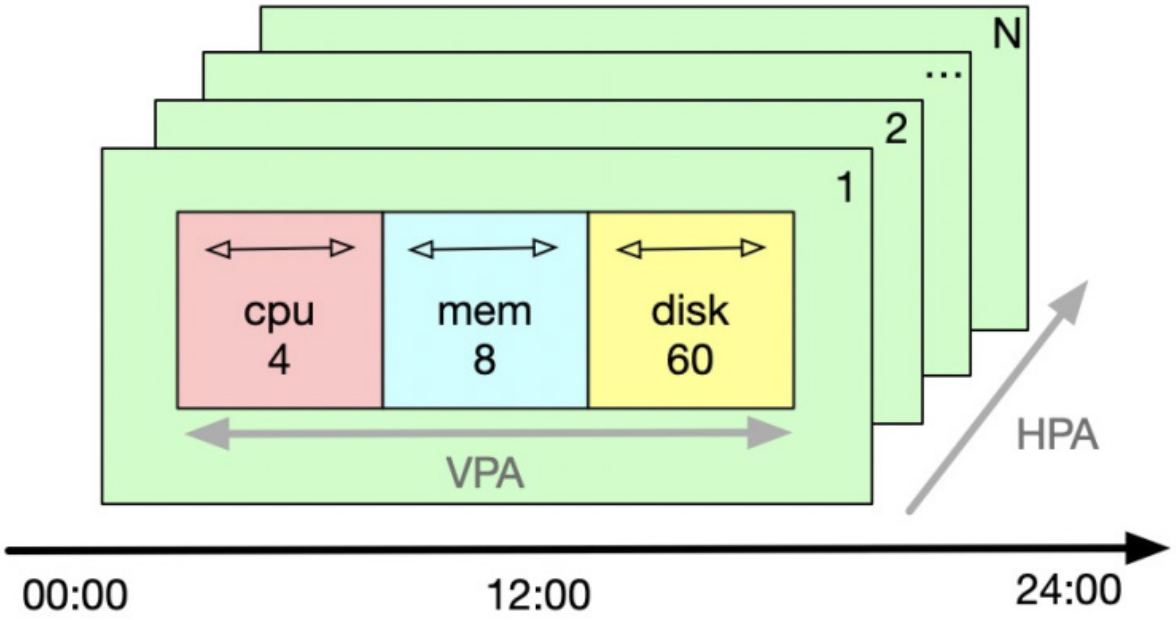


图 11-3 容器弹性示意图

图 11-3 展示了容器弹性的核心概念：

数量：一个应用可以用 10 台机器部署，也可以用 100 台机器部署。理想状态是当应用的系统水位较低时可以用较少的容量承载，当集群系统水位升高时，可以利用弹性扩容到较高容量，这个过程也就是大家常说的 HPA(Horizontal Pod Autoscaler) 模式。

HPA：HPA 解决的是 " 横向 " 的数量问题，这种模式下弹性可以根据集群的整体水位来做规则判定执行扩容还是缩容。常见的系统指标包括 :cpu 利用率、LOAD1、内存利用率、QPS、RT 等。引用很多云厂商以及原生 K8S 对 HPA 的基础实现逻辑：

目标容量 =ceil(当前容量 * 当前指标值 / 目标指标值)

HPA 的优势是控制和调度逻辑较为简单，实现成本较低；可以解决大部分的容量问题，这也符合业务同学对机器资源的期望。水位高了出问题了，扩点机器出来均摊下负载就可以解决。而 HPA 的劣势是扩容操作中，绝大部分情况应用需要冷启动，容量与水位匹配的线性程度不高，容量的变更存在一定的滞后性，这与应用的启动时间与相关依赖环境强相关；HPA 的结果是影响数量，会自然引起应用的 startup 与 shutdown 等生命周期变化，而当前应用健康也跟资源、环境、中间件依赖等很多因素强相关，因此这种个别实例的变化会给集群的稳定性带来不确定性。

规格：应用的容器规格是否合理也是衡量资源使用效率的关键因素。通常意义上，我们将容器规格分为三个方面：占用的 CPU 核数、分配的内存大小以及使用的磁盘大小。

时间：合理的时间规划也是衡量集群资源使用效率以及弹性收益的重要手段之一。因为数量的控制以及规格的分配都是在某一段时间上产生影响，可能到了下一时间段，其数量与规格又会发生较大变化。所以无论是数量还是规格的变化都需要乘上时间维度这个变量才能发挥弹性收益。

11.4 国际化弹性产品

从上面内容我们了解了容器资源消耗的各个因素，国际化的弹性工具也是围绕数量、规格和时间进行思考建设，且在国际化的多种复杂场景下进行验证。

自动弹性产品

利用 cpu 利用率、load 等指标判断执行弹性扩缩容且常为 HPA 模式。优势在于指标配置灵活，规则简单易懂，业务开放程度高。劣势也很明显，首先，集群的整体性能判定过于简单，有时无法描述集群真实的扩缩容目的；其次，准确性严重依赖 metric 监控系统，对于未接入监控的应用，无法发挥作用。

分时弹性产品

通过对应用的历史水位进行分析以及应用画像的描绘，对具有明显稳定分时特性的应用实行的弹性手段。该能力适用于日内存在明显峰谷特性的应用。

定时弹性产品

不同于系统自动的根据水位来判断弹性执行，基于 cron 的定时弹性给了业务更加定制化的将容量分散计划到时间线上。

大促弹性产品

国际化电商业务每年也会存在多场大促活动，大促批量扩缩容工具主要覆盖 " 大促容量预估 "、" 大促压测前备容 "、" 边压边弹 "、" 大促后快速缩容 " 等场景、形成大促态容器容量管理的闭环。

11.5 弹性的稳定性

在稳定性的要求下，国际化业务的弹性策略也需要考虑其部署与容灾架构的特殊性，在此把实践过程中的稳定性建设进行总结：

- **同城双活**：同单元多机房部署，例如一个应用在美东单元部署在 us44 和 us68 两个机房，弹性策略需要考虑到每个机房至少保留两台实例，即使是尾部流量较少的流量
- **异地多活**：国际化核心业务要求很多应用需要在多单元容灾部署。举例 AE 的交易应用需要再德国、俄罗斯、美东等多个国家区域部署，当德国单元出现故障时，接入层流量会将其流量切到俄罗斯或者美东，来消除故障影响。因此弹性的容量操作也需要考虑这种情况，要保证某个单元承受其他一个甚至多个单元的流量时，容量可以扛得住，因此弹性策略也需要和容灾策略进行关联。

仅限内部传阅

- **弹性稳定性检查**：下面列举出多个类型的风险检查项，将作为弹性执行前置与后置检查
  - **VIP 机器挂载异常**：开启 VIP 服务的集群在线机器数与 VIP 挂载数不一致
  - **单分组多部署环境**：一个应用分组下部署了多套环境
  - **机器在线状态异常**：出现很多非 working_online 的机器不放进弹性体系，且应用下线掉这一部分机器
  - **(分组 - 单元机房) 不匹配**：德国分组下存在俄罗斯单元机房的机器，可能是 Normandy 的扩容界面配置太过自由，选择了分组后，扩容目标单元和机房可以随意组合，导致人工手误扩容出错。

11.6 弹性的“故事”

下面列举一些因弹性引发的一些故障，来看看弹性对我们技术架构的重要性。

PDP 成功量下跌超过 10%

背景：618 大促缩容

原因：Pod 扩容不支持自动挂载 VIP, 需人工手动添加，如图 11-4 所示。

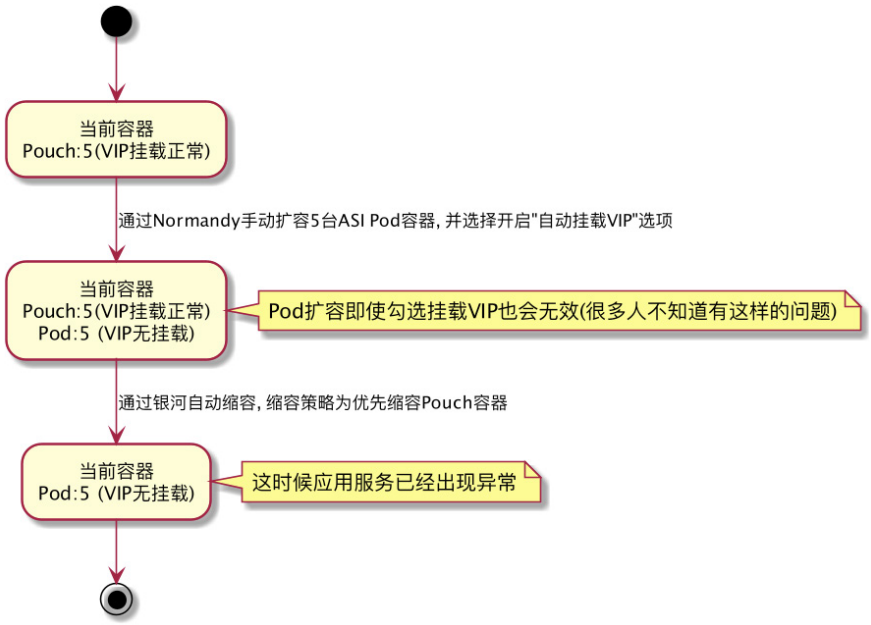


图 11-4 PDP 成功量下跌

措施 Action：

- 联系 Normandy 底层支持 Pod 扩缩容自动挂载 VIP, 目标同 PSP 的 Pouch 扩缩容效果
- 拉取接入 VIP 能力的应用离线数据，加入弹性黑名单，不开启弹性
- 同时我们也需要分析拿到了 VIP 已挂载分组容器和未挂载容器的数据，推动应用 owner 去手动挂载

AE 交易订单状态不一致

背景：日常缩容

原因：银河缩容正好缩掉了 CSB 单元化调用不支持的 11 网段的容器，如图 11-5 所示。

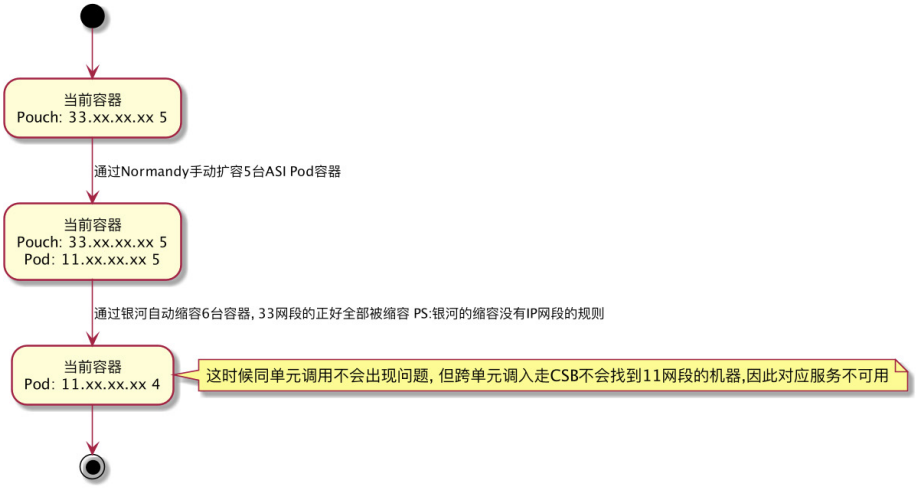


图 11-5 AE 交易订单不一致

措施 Action：

- 缩容策略上不会一次性同时将同 IP 网段的容器缩掉
- 目前将 CSB 的这种 " 有状态 " 靠 ip 网段进行服务发现的功能进行改造难度太大，经过上次 GOC 的复盘

大家也一致认为不可能做到，风险也太大，因此只能说被动地跟 ASI 的网段信息同步，如果有新的网段出现，就加入到 CSB 规则中去

卖家 seller portal 成功量下跌（类似非弹性引起）

背景：业务方自己进行云下到云上机房搬迁

原因：存在业务上的原因 :daraz 的前端调用了 lazada 的后端服务，但是导致问题的最根本原因也是底层 Normandy 不支持 Pod 扩容自动挂载 VIP, 而操作人也忽略了这点，整体故障原理类似于故障。

仅限内部传阅

十二、区域化部署

12.1 背景与定义

从全球化部署这一章节可以了解到，目前的全球化主要有两种方案，一种是类似 Amazon 和阿里集团 Lazada 的本地化站点的商业模式，这种本质上是不同的站点；另一种就是全球买，全球卖，将一个商品同步到全球所有站点进行售卖的全球区域化部署模式。针对这两种模式的异同，在此章节就不详细展开，请参考“全球化部署”这一章节。

那什么是区域化部署呢？区域是指地理区域，区域化部署就是按需在多个地理区域部署应用程序，全球区域化部署就是在全球的多个区域部署应用程序为用户提供服务。本章中出现的“区域”均指在全球化部署中，某一个（多个）机房一起服务的地理上的一块物理空间。简单理解可以使用机房替代，但实际上一个区域内通常存在多个机房。

12.1.1 区域化部署总体原则

区域化的核心需求是电商的全球买、全球卖，此外还要考虑到用户就近访问的网络延时性、容灾场景等。在这种多区域部署的场景以及核心需求的制约下，区域化部署的总体原则就很明确了，即**不同于 Amazon 以及 Lazaada 的本地站点模式，不同区域之间必须保障数据的一致性。**比如当来自不同区域的买卖家进行交易时，需要保障共享数据的一致性；当异地容灾时。用户区域进行迁移后，也需要保障不同区域服务统一用户的一致性。

在这些区域化原则制约下，我们从流量的一次完整漫游来看我们所面临的问题，如图 12-1 所示。

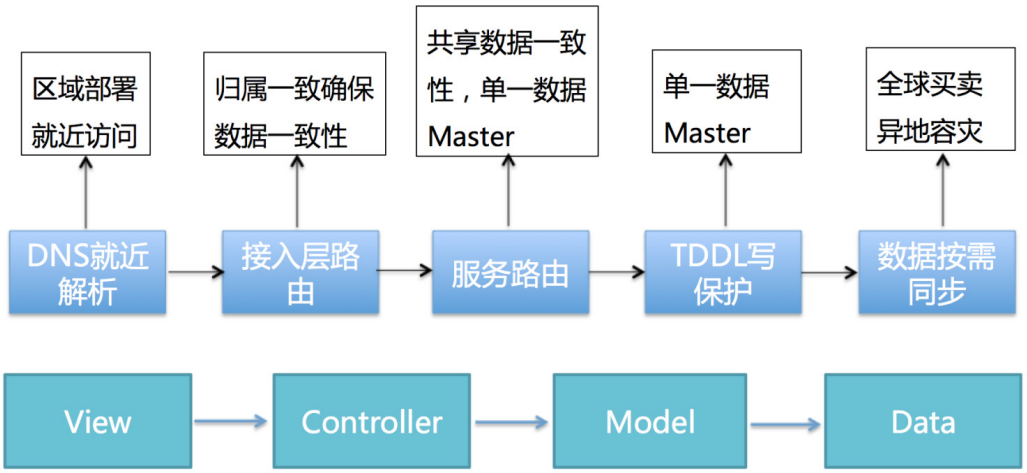


图 12-1 区域化部署缩面临的问题

仅限内部传阅

① 就近解析：在 DNS 解析层，通过 DNS 与网络层调度编排技术的结合，实现了区域的部署以及用户的就近访问；

② 数据一致性：在接入层路由中，实现了用户归属的一致性，从而保证了单一用户的单一数据 Master，从而确保数据一致性；服务层路由确保共享数据 (全球买、全球卖) 场景下的单一数据 Master 及共享数据一致性；TDDL 写保护在最后一层，当数据入库的最后一道程式中，确保单一数据 Master；

③ 异地容灾：由数据层的按需同步来确保异地容灾的数据完整性，由网络层调度编排技术来实现异地容灾的解决方案

12.1.2 区域化的要求与挑战

在明确了区域化部署的定义以及总体原则后，我们所面临如下挑战：
技术基础条件：在全球多个物理区域存在自建机房或使用云厂商机房

① 网络层

- 可以让用户通过任何一个可行的网络链路到达这个机房；
- 可以让任何一个 CDN 厂商为用户就近提供静态内容的访问；

② 机房层

- 每个机房对等部署所有的应用程序、服务、数据库等；
- 对于任何一个用户，可以让任何一个机房为其提供服务；
- 用户所有的操作产生的数据是在为其服务的机房内部封闭的；
- 机房间是可以相互灾备的，在灾备切换的整个过程中确保数据一致性；

③ 数据层

- 用户如果在机房迁移过程中，系统确保数据一致性；
- 用户数据是否可以在机房间复制是可以配置的。

12.1.3 区域化部署关键能力

结合区域化部署总体原则以及技术要求，我们认为完整的区域化部署至少应该具备如下 5 个特征：

① 全球多地建站：区域化部署的第一个核心的特征是支持在全球多地多区域建站，这是一切的基础以及我们的技术基础条件

② 数据按需同步：多个区域之间需要具备数据按需同步的能力，卖家需要将一个商品同步到全球所有区域进行售卖；同时在容灾场景下，在灾备切换的整个过程中也需要保障用户的数据一致性；并且在某些国家区域法律法规的限制下，比如政策不允许数据处境 ，就需要过滤掉对应国家或区域的用户数据。

仅限内部传阅

③ 用户路由机制：即用户经过 DNS 进行就近访问后，考虑到 DNS 的漂移以及容灾场景，我们需要明确将用户路由到一块固定的区域，即用户路由机制，保障用户归属的一致性

④ 数据一致性保障机制：用户归属的一致性保障单一用户的单一数据 Master, 而对于数据强一致性的业务，比如交易支付等，我们还需要在数据入库的最后一道程式中，通过 DB 禁写等手段来进行保障

⑤ 用户网络加速：DNS 就近解析可以确保用户归属在最近的机房，但同时我们还需要有能力让终端用户通过最快的网络路径，例如 CDN 边缘节点来快速访问到最近的区域机房

12.2 区域化部署之整体技术架构

本小节从总体架构和应用部署架构来阐述区域化部署的大致理念

12.2.1 总体部署架构

区域化的总体架构分为机房层，管控系统，ZK 变更推送，技术栈 (基础设施、数据库 / 存储 / 数据同步系统、中间件、应用程序)，路由服务等几部分组成，如图 12-2 所示。

① 管控系统：具体实现及控制机房调度的系统，即控制所有的用户归属、调度编排等。负基于大数据的监控来实现路由的优化，发起路由变更，并且确保各层路由的一致性

② 分布式管理系统：在分布式系统中，确定了用户归属以及调度逻辑后，我们采用 Zookeeper 来确保每个机房中路由变更的一致性。

③ 机房层：分为中心机房及区域机房，所有区域机房与中心机房之间进行双向数据复制，来实现数据的按需全量复制。

④ 技术栈：技术栈分为基础设施和中间件层，基础设施例如服务器、数据库及关联存储同步系统；中间件例如 HSF/Notify/TDDL 等；技术栈的每一层都依赖路由服务，并且从接入层、服务层、数据层来确保用户路由的一致性。

⑤ 路由服务：用户路由表用于确定用户到机房之间的唯一映射关系；并且对外提供服务，用于各层之间查询用户归属； 如何确保各层中用户归属的全局一致性以及高可用方案，是区域化部署中的核心内容。

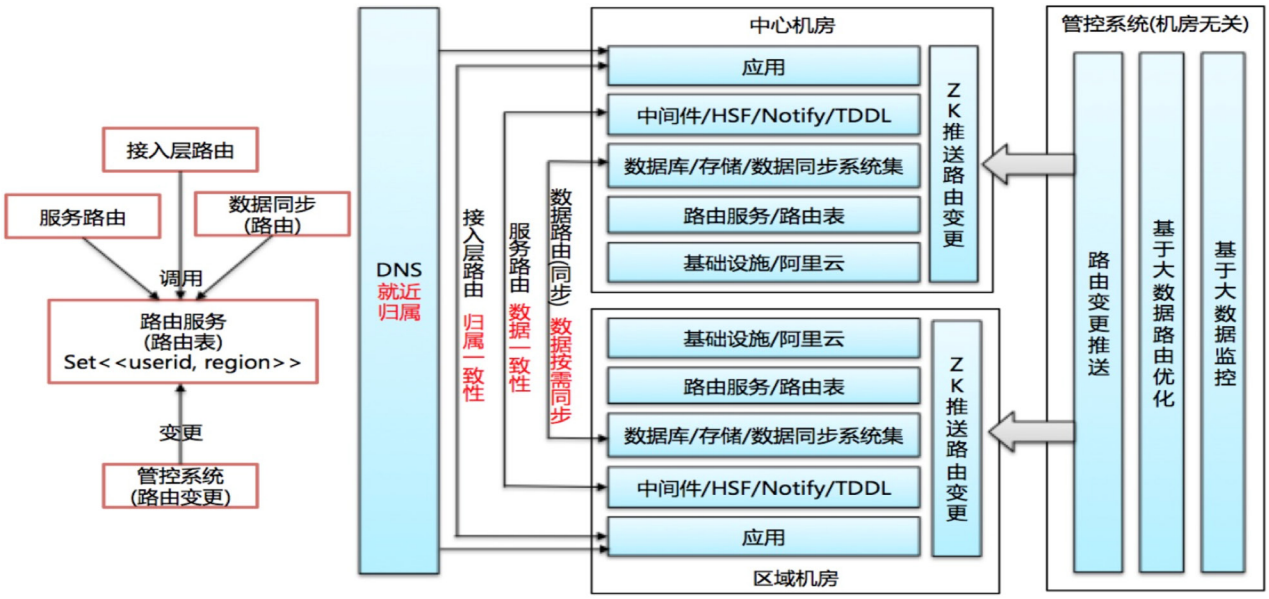


图 12-2 区域化部署总体架构

12.2.2 应用部署架构

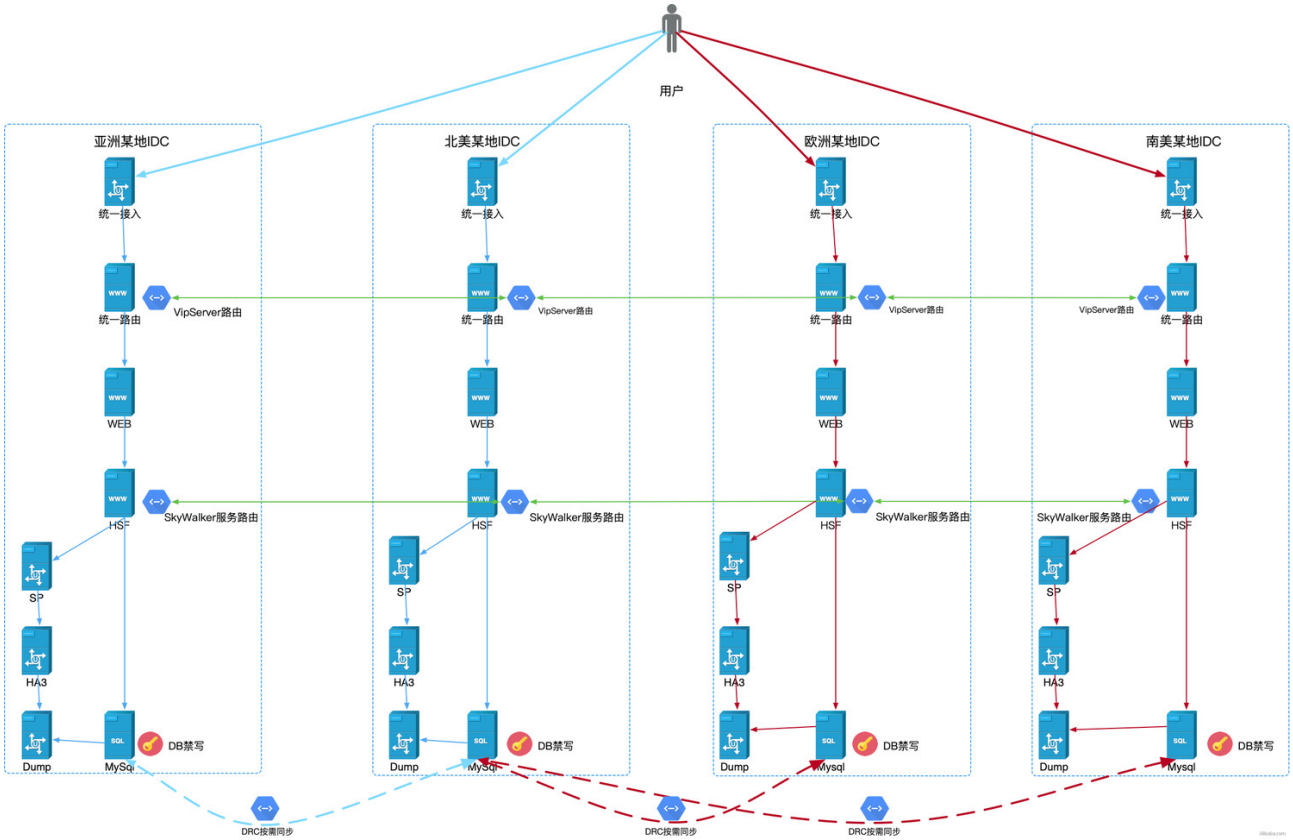


图 12-3 区域化应用部署架构

图 12-3 是区域化部署的应用层架构。首先我们需要在全球多区域实现完整的建站，例如美东、德国、新加坡、中国等 IDC。

- 用户根据 DNS 就近解析到最近的机房 IDC，到达该机房的统一接入层。

仅限内部传阅

- 在用户到达统一接入层之后，需要桥接一个 Nginx 的统一路由层来对用户归属进行强一致性纠偏，即在 Nginx 中调用路由服务，查询用户的归属，通过 VipServer 来实现跨机房调度，来达到用户跨机房跳转的目的。
- 在服务层，对于强一致性数据，例如支付、交易等，需要对统一路由层的用户归属进行保障性兜底，即如果统一路由层路由错误，那么 HSF 层也需要将服务跨机房调用回用户正确归属的机房进行消费；同时针对共享型数据的一致性问题，需要拓展出中心读写的跨机房服务调用功能; 简而言之，在 HSF 层需要实现根据用户归属或者中心机房消费的跨机房调用功能，在这一层，我们借助于 HSF 的 CSB 和 SkyWalker 来进行实现。
- 在分布式数据库层，我们通过扩展其插件，实现了禁写功能，同样是对于用户归属错误和数据强一致性保障的兜底，即用户归属区域如果和实际调用区域不一致，我们将会对其禁写保护，来避免不同区域之间的数据脏写。
- 在数据层，中心机房和区域机房之间数据进行双向同步，保障异地容灾的数据一致性。避免用户区域更改后的数据缺失情况。

12.3 区域化部署之路由体系

12.3.1 用户路由表

为什么需要一张用户路由表？

最根本的原因是数据一致性和按需同步的要求。理论上 DNS 就近解析可以满足需要，但是在数据强一致性的制约下，DNS 的飘逸可能会导致数据的多区域数据双写。并且在数据按需同步的场景下，我们也需要知道用户的归属来判断数据能否同步。放在全球范围内，如果不能做到根据区域就近服务，那么对于用户就会产生很大的性能损耗；同时也不能获取用户归属来判断数据是否合规。因此，在全球区域化部署的背景下，我们需要的是为每个用户指定到特定的归属机房，为此，我们必须实现一张用户 ID 到所属区域的关联表，也就是我们必须保存几亿用户到机房的映射关系表。

如何来设计用户路由表？

当我们需要存储几亿、几十亿的数据时，并且用户一次访问可能需要多次查询路由信息，如何设计路由表是一个极大的挑战。

最好是保存在内存中，可以避免分布式的复杂问题。大数据存储有一个典型的应用案例，即如何用尽可能少的内存消耗来标识一个用户是否存在？比特数组可以很好的解决这个问题，如果用数组的下标来作为用户 ID, 其对应的 bit 位以 1 来标识用户存在，0 则标识该用户不存在。如图 12-4 所示。路由表存储的是用户 ID 到用户归属机房及用户状态的对应关系，因此还需要知道用户归属到哪个机房中，以及用户当前所处的状态。假设我们仅支持最多 8 个机房，用户状态最多两种状态，则我们可以用 4 个比特来表达一个用户，即使这样保存 1 亿用户需要消耗 4*100000000/8/1024/1024~=47M 的内存空间。访问时间复杂度不变。

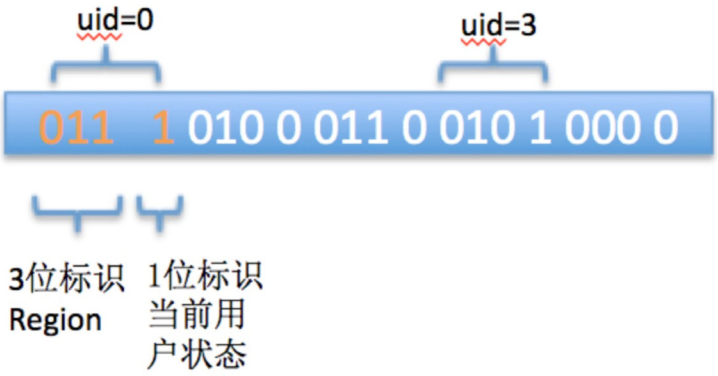


图 12-4 用户路由表

另外，还有一个问题是用户 ID 分布不均匀的。既然分布不均匀，那么我们能否有办法忽略那些空余的用户 ID 段呢？要解决这个问题看来只有引入分段模式了，分段的核心思路是建立一个段索引表，每个索引项用来指向一段比特序列，用来保存该段内部的用户信息（例如我们以 100 万用户为一个段），针对那些索引项中一个用户也没有的段，其索引表就直接指向一个 NULL 段，其对应的比特序列空间也就不用再分配，达到节省空间的目的。综上，我们的方案如图 12-5 所示：

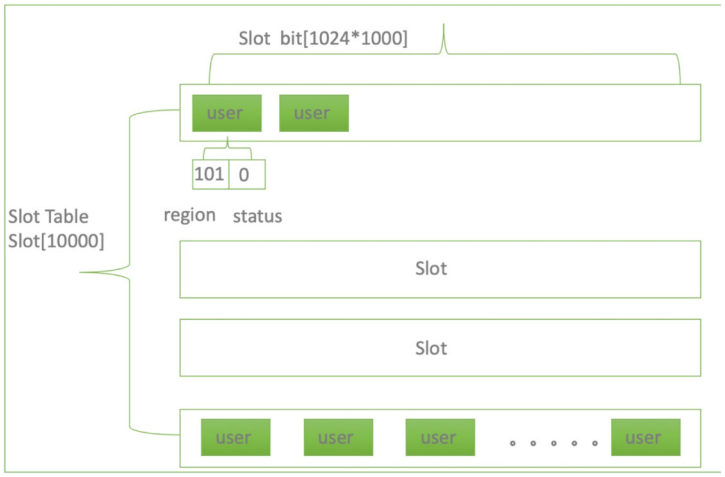


图 12-5 用户路由表设计

整体上看我们是通过一个二维的 bit 数组来存储数据，下面的伪代码简单的描述来如何从二维数组中获取用户路由信息的方法：

```
index = (userId / 1024*1000);
subUserId = (userId % 1024*1000);
User=Slot[index][subUserId]
region=user[0,3]
```

这样存储带来的好处：

① 1 亿用户需要消耗 4*100000000/8/1024/1024 ~= 47M 内存空间。

仅限内部传阅

② 查询的时间复杂度 O(1)

③ 能有效的解决 UserID 分布不均匀的问题，比如：20 亿空间中实际只使用 1 亿的 ID。

用户路由表数据

然后我们继续看如何来构建路由表中的数据

- **首先**实体路由表最核心的需求是我们需要就近服务，所以我们的第一个原则是根据用户的访问 IP 计算国家，然后根据国家分配区域。第二个原则是对于卖家如有需要，我们可以在其认证通过之后，根据其认证的信息进行区域纠正。第三原则是对于特色的有数据保护的国家，根据公民信息进行区域纠正。简单的来说，我们是以用户实际的 IP，作为基础的计算策略，然后通过一系列不同优先级的需求进行纠正即可。
- **第二**个需要注意的问题是路由表更新频率的问题，准确的说是单个用户路由信息更新频率的问题。当然从系统设计上任何时候只要有必要都可以支持其更新，但是考虑到区域变更随后进行的停写和数据迁移，我们有必要控制更新的频率，比如只有当用户稳定在新国家或区域 1 个月以上的时候，我们才会讲用户迁移到新的区域。
- **第三**是新用户的问题，当新用户刚刚注册时，我们通常将其默认路由到中心区域，只有其在网站有足够的行为数据的时候才会进行路由迁移。图 12-6 是用户路由数据的构建过程的一个实例。

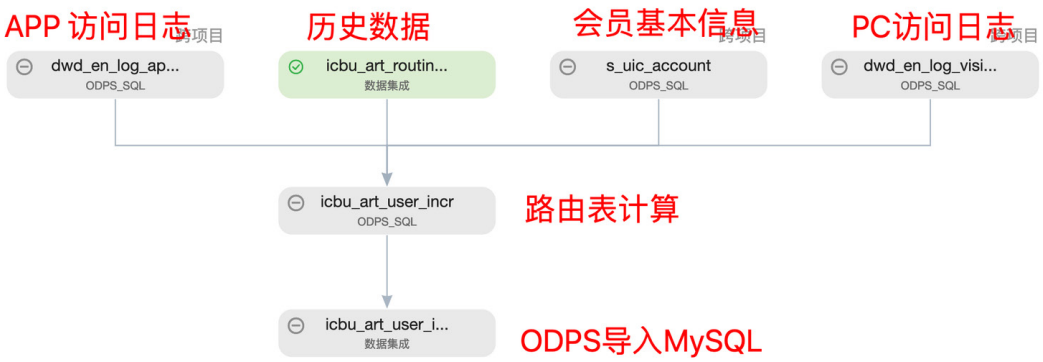


图 12-6 用户路由数据构建

12.3.2 全球路由体系

前面已经介绍过，区域化部署技术本质上就是多层路由。而多层路由中每一层要做的事情，就是基于用户所对应的归属机房进行调用的路由。真实的路由服务的输入是用户 ID，输出是对应的机房以及用户对应的状态信息。状态信息主要是用于记录用户迁移、或者机房容灾中，当前用户所处于哪个阶段，用于确保一数据一致性。

基于路由服务，接入层、服务层、消息层、数据层的路由构成了整体的区域化部署方案。下面将介绍各层路由的实现原理，以及关键技术细节。

① 统一接入层路由

区域化部署技术是一套系统进行多区域对等部署，并将用户归属到不同的区域，而这个过程对用户是

透明的，归属到不同区域的用户访问我们网站时是用相同的域名，看到的是相同的页面。通过 DNS 技术将不同用户解析至其对应的不同机房来实现。但这个实现是基于 IP 地址的，而无论 DNS 技术还是基于 IP 地址技术本身，都存在不确定性，服务大量用户的系统必然会出现一种情况，就是用户被 DNS 解析到了其并不归属的机房，这就会出现一个问题，用户可能刚刚在正确的归属机房中进行了操作并产生了数据，而这个数据还未被复制到当前被解析到的机房中，用户的操作就会产生数据不一致性。

我们通过统一接入层路由技术解决这个问题，如图 12-7 所示。所有应用的域名都会解析到一个 VIP，这些 VIP 可能会不同，但是都对应到统一接入层，统一接入层会基于请求的 URL 或域名来确认是对哪一个应用的请求，并将请求代理转发到对应的应用。

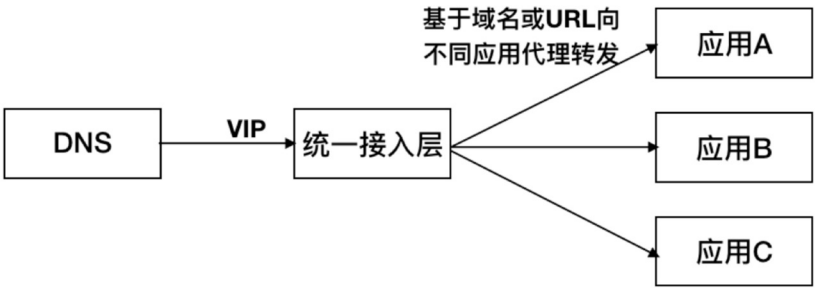


图 12-7 统一接入层路由

那么整个请求处理流程中，Nginx 如何使用路由表呢？再想想本节开始提到的问题，如果 DNS 将用户解析至不正确的机房，应该怎么做。方案很简单，每一个统一接入层的 Nginx 都会从 cookie 中获取用户 ID，并调用路由表模块提供的函数，取得归属机房信息，如果归属是本机房，则基于域名及 URL 路径信息，proxy_pass 至本机房对应的应用程序；如果不是本机房，将用户请求 proxy_pass 至所归属机房的统一接入层，统一接入层接收到请求后，再从头开始执行上面的流程，调用路由表后发现用户是归属到本机房，则 proxy_pass 至本机房中对应的应用程序。

② 服务层路由

统一接入层路由确保了用户归属的一致性，使得用户对自身数据的读写是在同一个机房发生，从而实现单一数据 Master 原则，确保数据一致性，同时结合上文介绍的网络层路由技术，也将在一致性的基础上实现最小的 Latency，最优的性能。如果两个用户共享同一份数据，但是这两个用户归属为不同的机房，应该如保证一致性和性能？如果某数据不归属于任何用户，是一种全局共享的数据，但用户的行为会触发这个数据的更新操作，应该如何保证一致性和性能？

下面我们来看看服务层路由技术将解决这一问题：

- **服务层路由设计原则**

用户优先级原则：就是当不能确保所有用户体验都是最优时，优先保证哪个用户的体验最优，保证顺序如何。如果多个用户共享同一份数据，且数据仅与有限的几个用户相关，非全局共享，而此时如果多个用户归属于不同机房，为了确保单一数据 Master 原则，一定是要有用户需要跨机房调用非其归

仅限内部传阅

属机房的服务。究竟谁应该跨机房调用呢？这里就涉及到用户优先级，在确保单一数据 Master 及数据一致性的前提下，优先保证谁的用户体验。这个优先级依据业务而定，看业务最重要是服务于哪一个或哪一类客户。

在电商的场景中，多用户数据共享最复杂最典型的案例就是买家、卖家、平台运营人员共享同一份数据，比如买家与卖家都可以变更其相互交易的订单，而平台运营可以基于安全或风险控制的原则关闭掉这个订单。优先级显然是买家重要性最高，卖家次之，平台运营最低，有人会质疑为什么买家最高，很简单，买家是卖家的客户，是卖家生意的来源，服务好买家对于卖家的生意有帮助，而买家和卖家都是平台运营的客户。

- **三种服务路由模式**

针对不同的数据类型，以及基于单一数据 Master 及用户优先级原则，服务调用分为三类：

1）Local：本地模式，服务按需部署，但一般情况都是全球部署，服务调用只发生在本地，区域内封闭，这类模式用于支持只读数据、独享数据的场景；

2）Global：全球模式，需要在全球部署，服务的调用进行按需路由。中间件会获取用户 ID，取得用户归属区域后，将服务调用路由至对应区域，这就要求有一种获取用户 ID 的方法，后面会详细介绍；Global 模式用于支持普通的共享数据场景；

3）Center：中心模式，一般只需要在中心部署，为了容灾考虑，仍然全区域部署。但所有的服务调用都会路由到一个区域，就是前面提到的网络条件最好的区域。对于全局共享数据的支持就采用这种模式。这种模式有一个天生的风险，全局共享数据往往是非常重要的核心业务链路的一环，比如电商中的库存扣减，几乎在所有的下单链路都涉及到调用，而采用中心模式的跨机房调用，会让每次服务调用增加 100ms-300ms 之间的延时，延时一方面影响用户体验；但更大风险是由于延时带来的系统稳定性的风险，延时加大会让用户对应的线程所加载的资源如数据库 connection 不能及时释放，造成对资源的争用，让系统下降明显；并且这些资源往往占用较大内存，迟迟不释放，并且不断在申请，会带来 OOM 的风险。

- **服务路由原理及架构**

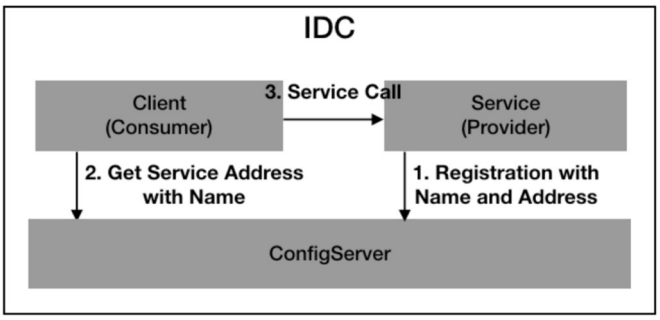


图 12-8 服务路由原理

图 12-8 展示了一般的服务路由原理，适用于 Local 模式中。而对于 Global 和 Center 模式，就是是在这个技术基础上进行演进。除了涉及到 ConfigServer，Consumer，Provider 以外，新增了一个两个

服务组件：

1）SkyWalker，作用就是跨机房的服务目录同步，将 Global 及 Center 模式的服务注册到其它机房的 ConfigServer 中，并带有机房前缀，而每个机房的 Consumer 就可以仍然使用本机房的 ConfigServer 获取跨机房的服务地址；

2）Routing Service，就是前面介绍到的路由服务，用于通过用户 ID 获取其归属机房。为了方便理解，这里单独画出一个 Service，而真正的路由实现仍然是采用统一接入层透传，与失效远程读取相结合。

基于两个新增组件，就可以实现完整的服务路由，通过图 12-9 中的例子，我们可以完整的理解原理和流程：假设有一个 Global 或 Center 模式的服务提供者在 IDC-B 中提供服务，而对于这个服务的跨机房消费者在 IDC-A 中进行调用；服务提供者既会向本地的 ConfigServer 注册自己，也会通过 SkyWalker，将自己注册到 IDC-A 的 ConfigServer 中，并带有机房前缀；IDC-A 的消费者在调用时，会先去路由服务中查找当前用户的归属机房，发现归属机房为 IDC-B，则带着机房前缀去 ConfigServer 中获取服务调用地址，如果此时是 Center 模式的服务，则可以省去路由查找过程，本地的 ConfigServer 会直接将 Center 模式的服务的查找返回对应的跨机房中的地址；IDC-A 中的消费者基于 ConfigServer 返回的地址进行服务调用；如果消费者在 IDC-B 中，实现流程与 IDC-A 前面部分相同，只是在服务调用时不进行跨机房调用。

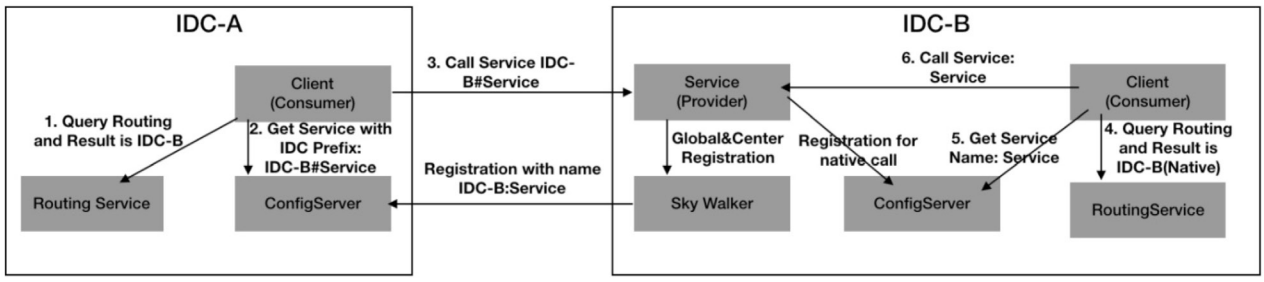


图 12-9 服务路由架构

③ 数据层跨区域路由

在区域化部署的情况下，多个机房之间如何维持数据的一致性？数据同步是实现这种一致性必不可少的重要环节，然而数据同步的场景本身也极其多样化，比如我们会有传统 DB 数据的同步、文件同步、缓存数据同步等等等等，看上去我们似乎应该为每一种场景都有必要设计一套独立的数据同步系统。数据同步除了前面提到的存在不同系统的差异性之外，还存在各个机房数据流向的差异性。比如我们有些数据希望同步到指定的机房，而另外一些数据则不希望同步到这些机房，这些约束规则就需要我们在数据同步系统层设置一套统一的数据流向控制系统，它能够根据我们数据路由规则，决定一条数据的正确流向。

从上面的分析中我们发现，如果每种系统都建立一套自己独立的数据同步系统的情况下，此时假设我们还想控制同步过程中数据的流向的话，我们会发现实现起来会非常麻烦。比较理想的结果是，不管是什么类型的系统，如果其数据同步层都能归结为一套 / 两套数据同步系统，这样对我们整个区域化

仅限内部传阅

部署的同步系统架构复杂性就可以降低很多。

例如对于持久化缓存的数据同步，首先不推荐将其作为一个高可靠的数据化持久层，推荐将其作为一个持久化的只读数据层，这样一来理论上所有存储在缓存中的数据都不会有跨区域数据同步的功能，因为其数据都存储在另外的持久化数据层（比如 DB）中，其数据同步基于 DB 层面的同步即可，最终我们其他的数据同步通过改造都可以基于 DB 的数据同步，所以接下来我们主要看 DB 的数据同步以及改造方案。

DB 数据同步架构

首先我们采取了星型同步的方案，星型同步是一种中心化的方案，首先会建立一个中心机房，所有区域机房与中心机房建立双向数据同步，区域机房间无同步。其优点是简单，所需建立的同步通道随着机房数量呈线性增长，但是缺点是区域数据回到中心后需要广播到其他区域，中心同步压力大。

因为现阶段我们的机房数量相对不多，中心机房压力可以承受住。未来随着我们的机房数量不断增加我们可能会采取交叉同步的方案，接下来我们的同步方案围绕数据星型同步展开，整体数据同步架构如图 12-10 所示。

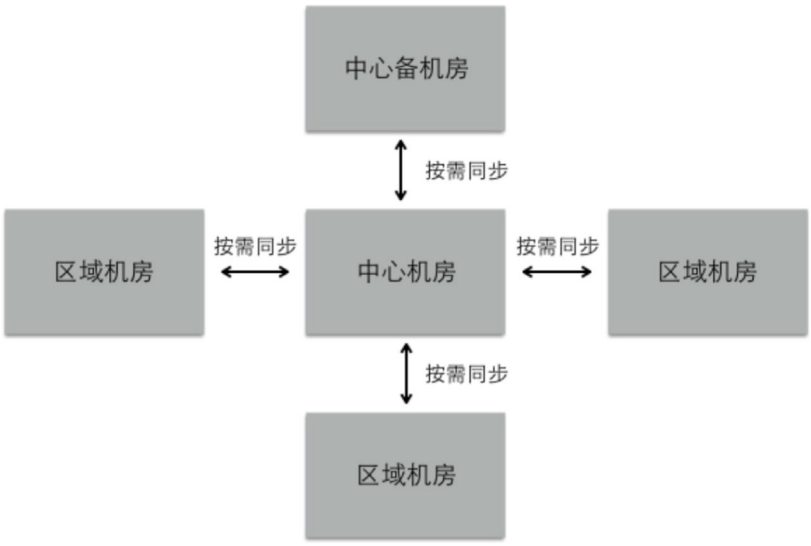


图 12-10 数据同步架构

首先 DB 数据同步采用的是 DRC 系统，DRC 系统是阿里自主研发的对异构数据源进行实时迁移同步的数据管道基础技术设施。DRC 同步模块主要由 DRC Strore 和 DRC Congo 组成，DRC Store 在原端进行数据的抓取、存储和发送服务，DRC Congo 在目标端顺序读取 Store 中的增量记录，并在保证记录更新时序和事务一致性的前提下，将记录高效并发同步到目的端数据库。

基础的 DRC 同步系统可以按照库或者表这一级别来过滤同步数据，同步过程中即可以开通单向的数据同步，也可以基于双向的数据同步，基于双向的数据同步场景下，这意味着同一张表的同一条记录，

假设多地都有可以修改，可能会存在同步冲突。区域化能保证多地永远不会同时修改同一个字段，即便是双向的数据同步，也不会产生数据冲突。但是基础的 DRC 系统还无法实现有选择的只同步表中的某些记录，因为这需要同步系统实现更加精细化的过滤逻辑，既让我们的同步系统不光要支持库表级别过滤，还需要支持记录级别的过滤，另外还要支持在现有过滤规则中嵌入业务相关的过滤逻辑。

接下来我们来看几种精细化的按需同步模式以及其适用的场景。

- 机房间数据同步最朴素的方案是任意机房间采取数据全量同步的方式，优点是在数据库容量充足的情况下，可以快速的进行机房间的切换，且用户数据迁移逻辑简单，但是此方式缺点是区域机房存在数据冗余。
- 在此基础上可以实现中心机房到部分区域机房之间采取过滤同步，优点是在当前分库分表的大趋势下，局部化的业务数据各自独立部署和存储的趋势势不可挡，区域机房只存储本机房用户的数据，可以减少 DB 的容量节省成本。如果用户从区域机房迁移到中心机房等同于全量同步方式，缺点是如果需要从中心或者其他区域用户迁移到被过滤同步的区域需要将此用户历史数据的全量迁移，用户迁移速度较慢。
- 还有一种极端的法律数据保护场景，某一个地区的用户数据不能出境只能保存在某一个区域机房，这时只能采取这部分用户的独享数据被过滤不能同步到其他机房，但是此方式有较大的缺点就是这部分用户丧失了异地容灾的能力，因为没有任何一个异地机房存在这部分用户的数据，只能采取同城容灾的方式。

以上几种同步方式都是各有优缺点，线上会根据业务场景进行组合选择。如果想要依靠路由表来实现基于路由表在 DRC 上扩展实现行级别的精细化的按需同步，请查看“合规”主题相关部分。

12.4 云原生下的区域化展望

目前，阿里巴巴的国际化业务正在全面上云，对于 AE 业务来说，已经在德国区域、美东区域完成了全面云化，并且得到了很多意想不到的惊喜。同样，云原生和区域化也能产生奇妙的化学反应，催生了更多的新技术。例如服务层路由下沉至 serviceMesh，让底座越来越厚，业务越来越薄。具体云原生的细节，可以参照“云原生”这一章节，期待云原生能够孵化出更多轻便性、高可用、高性能的技术。

仅限内部传阅

十三、多租户

13.1 多租户架构简介

13.1.1 什么是多租户架构

关于多租户技术在百科上有非常清晰明确的定义，引用百科的定义如下：多租户技术（英语：multi-tenancy technology）或称多重租赁技术，是一种软件架构技术，它是在探讨与实现如何于多用户的环境下共用相同的系统或程序组件，并且仍可确保各用户间数据的隔离性。

这个定义中明确表示了这是一种软件架构技术，并且指的是在面对多个用户的场景下，软件系统实例能够被复用，并且能够保证各个用户之间的隔离性，对于用户而言就像独占使用一样，所以多租户架构又可以称为单实例架构；通常来说除了保证租户之间的隔离性外，不同的租户有自身个性化的定制需求，所以多租户技术不仅仅在保证最基本的数据 / 配置 / 资源 / 流量的隔离性外，还要能够支持不同租户的个性化的定义以及个性化定义之间的隔离性。

在翻阅多租户的资料时，还有一个名词与之相对应，那就是单租户 SaaS 架构（也被称作多实例架构 Multiple Instance）。单租户架构与多租户的区别在于，单租户是为每个客户单独创建各自的软件应用和支撑环境。单租户 SaaS 被广泛引用在客户需要支持定制化的应用场合，而这种定制或者是因为地域，抑或是他们需要更高的安全控制。通过单租户的模式，每个客户都有一份分别放在独立的服务器上的数据库和操作系统，或者使用强的安全措施进行隔离的虚拟网络环境中；所以从根本上来讲，多租户技术是希望通过技术的手段，保证能具备单租户的隔离性以及安全性的前提下，又能尽可能的复用软件的计算资源，从而达到最大的资源利用率，降低成本；同时又需要面对各个租户个性化定制带来的挑战。

13.1.2 业界的发展历程

多租户技术源于 1960 年代，许多公司为了要使用更多的运算资源，向持有大型主机的供应商租用一部份的运算资源，而这些用户经常会用到相同的应用程序，当时会以用户在登录系统时输入的数据来决定用户的帐户 ID，基于这个 ID，大型主机的供应商即可利用此 ID 来计算运算的资源使用量，包含 CPU，存储器与软盘或磁带等，这个作法也被 SAP 公司用在其 R/1 到 R/3 的产品线。

到了 1990 年代，应用程序服务提供者服务（application service provider）模式出现，它的作法与运作模式与租用大型主机时相同，不过租用的资源是在软件上，除了操作系统以外也包含了其上的应用

仅限内部传阅

程序，例如 ERP 系统或是 CRM 等应用，系统可能会运行在数台不同的机器上，或是在相同的主机但共享不同的数据库，以区分并计算客户的资源使用量，以作为计费的标准，而此技术也有效的缩减供应商的实体机器成本（因为可以在一台电脑上同时运行多个用户所租用的应用程序进程）。到了现代，受欢迎的消费者导向 Web 应用程序（如 Hotmail 或 Gmail 等）也是以单一应用程序平台来支持所有的用户，这已经是多租户技术的自然演化的结果，多租户技术也可以让客户中的一部份用户得以进一步定制化他们的应用程序。

在虚拟化技术的成熟与应用性的扩张之下，多租户技术可以驾驭虚拟化的平台，更强化在用户应用程序与数据之间的隔离，让多租户技术能更加发挥它的特色。图 13-1 是多租户技术发展的几个阶段，在不同的架构层次上都存在不同的多租户技术实现：

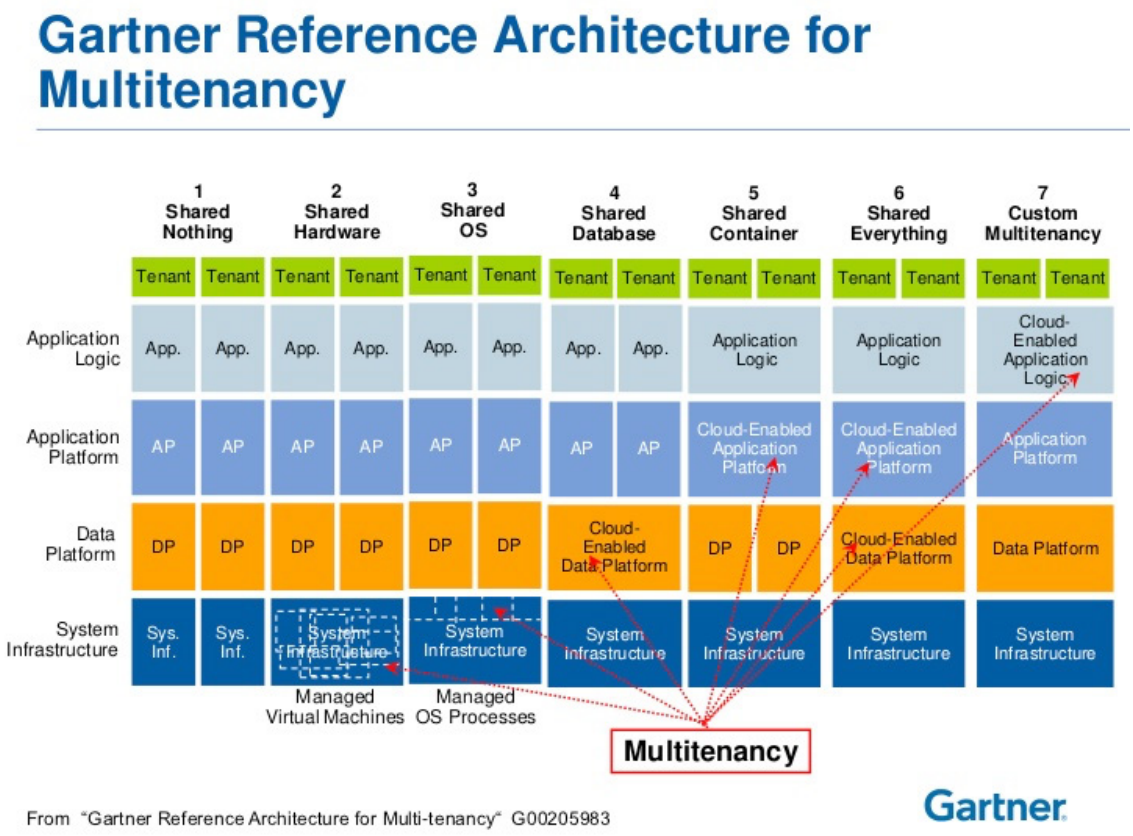


图 13-1 多租户技术发展阶段

回过头来，本文主要探讨的还是 SaaS 层的多租户技术，通常情况下，我们实现多租户的方式有如下几种方式，如图 13-2 所示：

- 操作系统层的简单虚拟化，其中只对硬件进行共享。
- 共享应用程序，对每个租户使用不同的数据库。
- 共享应用程序和数据库（效率最高，真正的多租户）。

仅限内部传阅

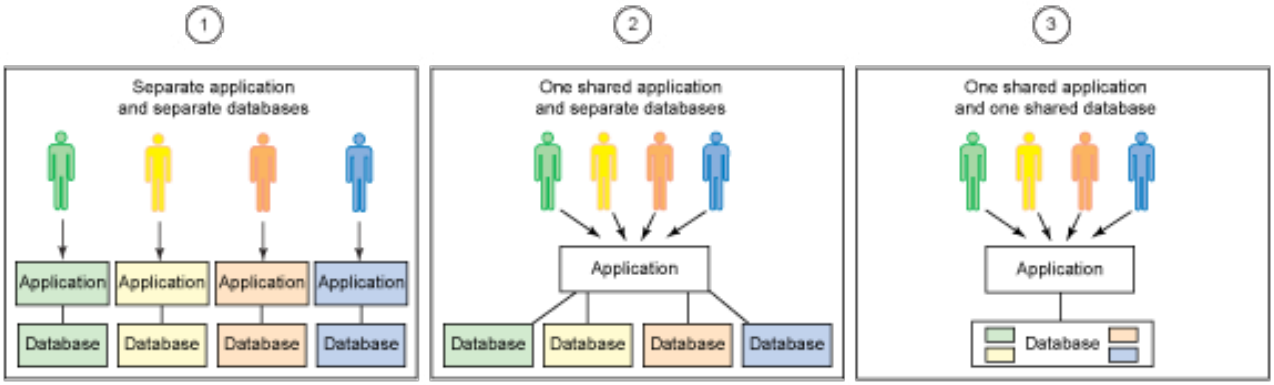


图 13-2 多租户的几种实现方式

对于第一种，基本上靠的就是操作系统的虚拟化技术或者容器化的技术，在应用程序层面基本上不需要做额外的变化就可以支持；对于第二种跟第三种核心的差别在数据库层，这部分在多租户架构上都是存在的，取舍在与对数据安全性以及隔离性的要求上，如果要求非常高要求的隔离性那么建议选择第二种方式；反之，第三种方式是成本最小，也是租户化程度最高的一种架构模式。

13.2 设计背景

13.2.1 国际业务的发展历程

在第一部分，对多租户技术的本质进行了明确的定义，那么在国际化的场景下，我们为什么会提到多租户技术呢？首先来看下国际化业务的发展历程，2017 年随着集团 Voyager 项目的开启，几个月时间用阿里技术栈完整的替换了东南亚电商 LAZADA，并在 2018 年顺利的完成了 LAZADA 割接，同年 9 月又交付了南亚的电商 DARAZ；随后随着 Sirius 项目的开启，完成了 AE 技术栈的融合；2019 年的 930，顺利的交付了天猫海外台湾站；当前整个国际化团队正在完成天猫海外香港站的建站；整体的发展历程如图 13-3 所示：

图 13-3 国际化多租户发展历程

国际化中台的业务从 17 年开始至今，每年都在扩张不同的业务站点，这些站点遍布了全球各地，LAZADA、DARAZ、天猫海外的站点集中部署在新加坡机房，AE 当前是以新加坡为中心，美国，俄罗斯，德国，杭州 5 个区域部署；除了部署形态上的差异，在业务形态上每个站点都有不同的定制诉求，在业务特征上，lazada/daraz 天猫海外的台湾跟香港站主要是本对本（本地招商，本地运营，面向本地买家）模式，AE 主要是跨境模式。从整体的经营模式上的差异如图 13-4 所示：

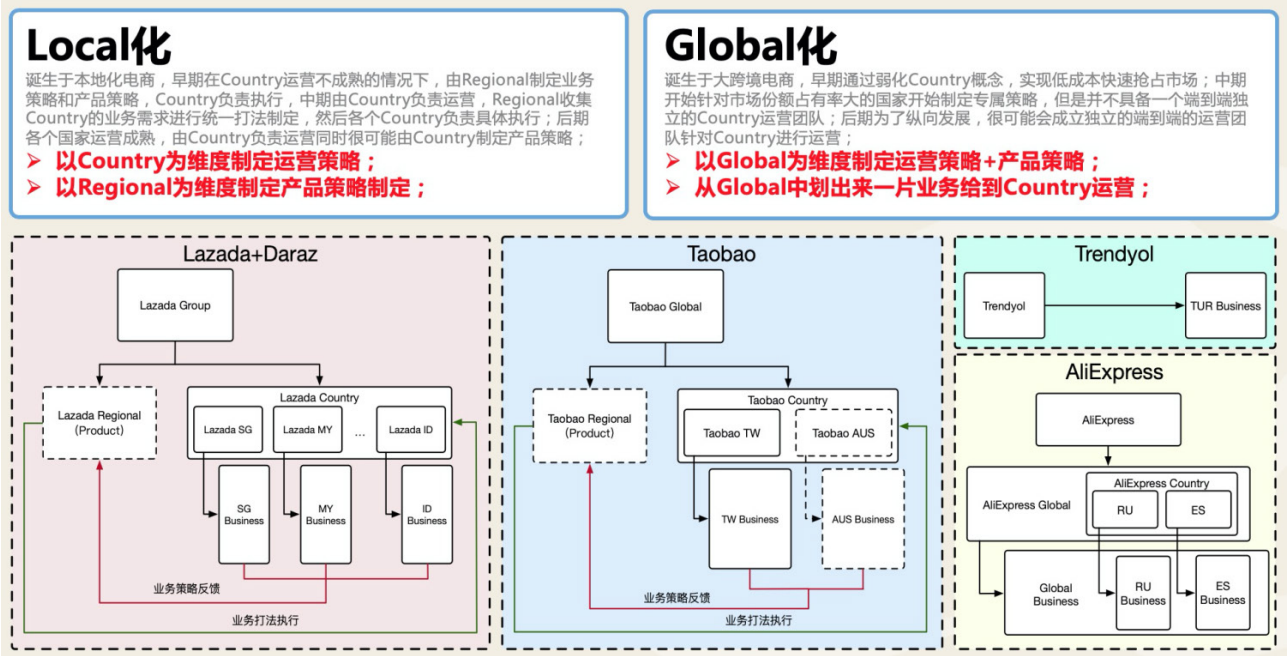


图 13-4 国际化业务经营模式差异

从趋势上看，针对某个区域、具体的人群做精细化运营是大的一个方向，包括目前的 AE 全球站也在往这个方向发展，比如 AE 的俄罗斯站，西班牙站等；相信随着国际化市场的拓展会有更多的国家站的建立覆盖全球的各个区域；另外从整体的业务运营策略上看，我们目前以 Reginal 为维度去制定产品策略，比如 LAZADA 的产品团队会统一看需要上哪些产品需求，先在哪个国家站做试点上线，然后复用到其他的国家站；所以整体上而言，目前的国际化场景下，业务在多个国家站之间是高可复用的。

13.2.2 多租户架构设计背景

在 Voyager 项目的期间，用阿里的技术替换了 LAZADA 的技术体系，LAZADA 从业务上讲是个本对本的电商，总共 6 个站点，每个站点有独立的运营团队，业务上也要求不同的站点数据是相互隔离的，并且不同的站点也存在一定的业务差异，在当时的项目背景下，没有一套完备的逻辑隔离的技术方案能解决站点的隔离性的问题，所以采用了最简单也最直接有效的办法，将不同的站点部署到了不同的业务集群，基于中间件单元标的维度为每一个站点搭建了一整套基础设施，在当时的这种架构模式下，顺利的保障了业务交付；但是当集团收购了南亚电商 Daraz，希望用 LAZADA 的技术体系能赋能 Daraz，对于南亚电商的 5 个站点，同样的事情还要做 5 遍，新开一个站点，需要重复的搭建基础设施，这部分的人力、时间、资源成本还是比较大的，整个 Daraz 项目 IaaS 层和 PaaS 层搭建，包括基础设施功能的验证，总共耗费了 1 个月时间；另外南亚的这些站点流程都比较小，最小的站点一天不到 1000 笔订单，这些重复的基础设施带来了机器成本的严重浪费；同时搞出来这些多套环境之后，对后期的运维成本也带来了不小的挑战，日常需求发布需要发布 11 个集群，大促需要运维 11 个集群的稳定性；另外从未来的发展的趋势来看，国际化场景下会有越来越多的本对本站点需要去建立，如果还是采用这套架构体系，开发、运维、资源、人力的成本会越来越高，并且在这种部署模式下，希望能够实现同城容灾几乎不可能，运维对等的环境还要部署一份，整个成本是巨大的。总结分析下原

仅限内部传阅

因就是我们的站点不具备逻辑隔离能力带来的一系列的成本问题。图 13-5 描述了基于本对本站点场景下的痛点：

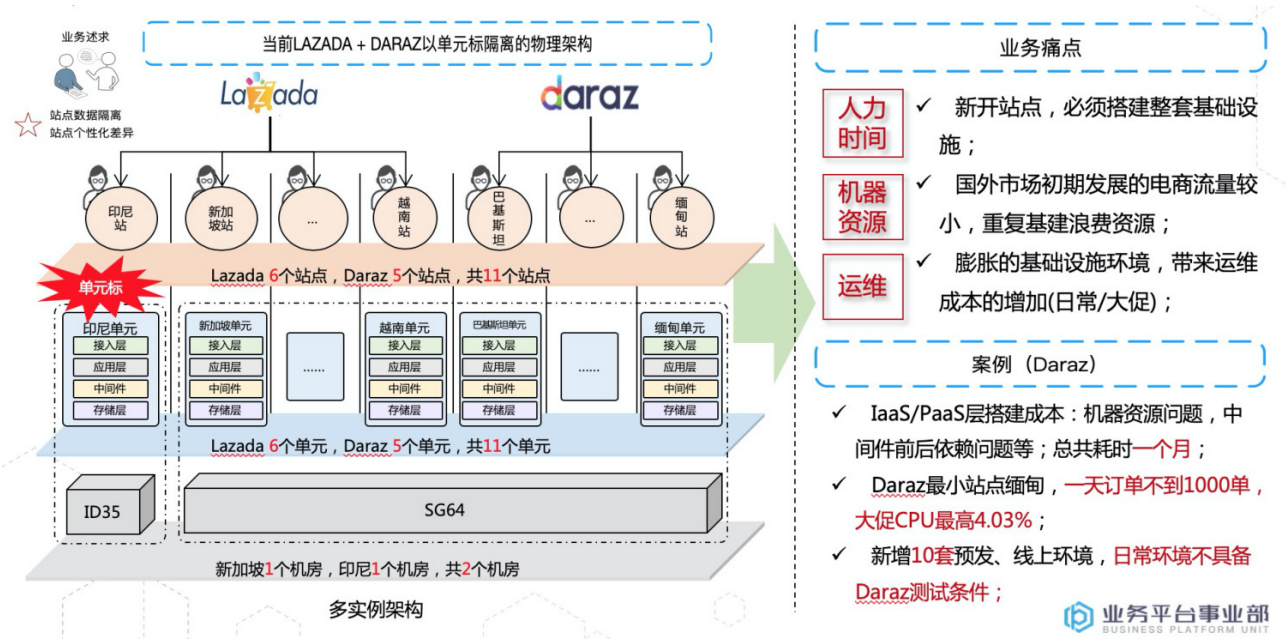


图 13-5 本对本站点痛点

在以 AE 为代表的全球买卖模式下，也呈现出越来越多的本地国家站需要建立，Reginal 的拆分、隔离也是需要讨论解决的问题。

13.2.3 国际场景下租户的定义

前面我们通过梳理国际化中台的业务发展历程，可以看到当前中台支持了 LAZADA，DARAZ，AE，TAOBAO 四大业务。在定义上，我们将这几个业务称之为 " 站点 "，像 LAZADA 又可以分为 6 个子业务站点，分别为新加坡站 / 泰国站 / 马来站 / 印尼站 / 菲律宾站 / 越南站。我们把这个维度的站点称之为 " 国家站 "（随着 HK，TW 站的建立，" 国家 " 这个词已经不太准确，但是这个概念还继续保留）；所以从站点的划分层次上分为站点 -> 国家站，相对准确的站点定义如下，一个站点需要具备以下 3 个明确的条件：

- **端到端独立的运营阵地**：必须要有一个明确的组织为一个明确的地区提供端到端的电商服务，并且这个组织需要具备这个运营阵地的全部经营权
- **全局独立的流量标识**：这个独立的运营阵地必须具备独立的流量标识，并且这个标识必须具备全站统一的识别标准，各个业务域不能独立定义
- **明确的服务人群**：这个独立的运营阵地必须明确其服务的买卖家，在买卖家的信息上也需要有明确的标识属于当前这个阵地

那么基于上述定义，国际化中台当前总共存在 4 个站点，LAZADA 包含 6 个国家站，DARAZ 包含 5 个国家站，AE 当前为一个站点，正在拆分 AER 等国家站，但是对于一个独立国家站的定义第二条，独

立的流量标识，AER 当前还不是很清晰。TAOBAO 当前包含 HK 一个国家站。这些国家站都具备独立的业务定制逻辑 / 数据等隔离的诉求，所以映射到技术实现的概念租户上，正好是一个国家站对应一个租户，比如 LAZADA 新加坡租户，唯一标识符为 LAZADA_SG，AEG 当前的租户标为 AE_GLOBAL。

13.3 实施方案

13.3.1 设计原则

那么怎么设计符合国际化场景的多租户架构呢？在国际化当前的场景下，不仅仅要解决数据的隔离问题，同时还要考虑到不同的租户之间的流量隔离的问题，以及不同租户配置的隔离问题，以及租户间的资源隔离问题 (保证租户间的限制性)；另外租户定制也是多租户架构中比较重要的一个话题，salesforces 以及华为的产品采用了基于元数据驱动的方式来实现定制能力；如何设计一套租户的定制化体系，能满足不同租户的个性化业务以及实现个性化逻辑的隔离？另外，在电商领域，如何去识别租户，以及实现在整个电商全链路进行租户信息的透传也是需要解决的问题，如果做不到全链路透传，那么意味着所有的系统间的交互接口都需要升级加上站点 ID 参数，这样的成本太高了；另外，对于后期整个架构的看护上也需要一些对策去保障。以上都是在设计多租户架构中面临的一些问题跟挑战，总结起来需要解决的问题如图 13-6 所示：

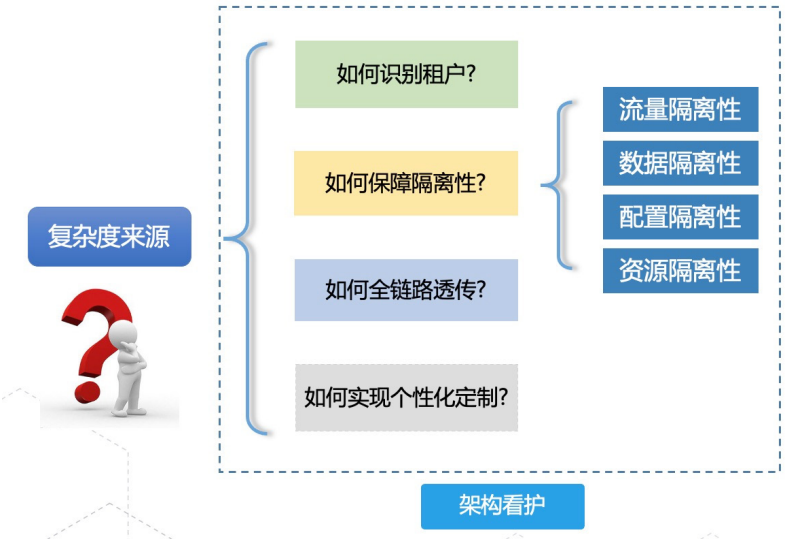


图 13-6 多租户需要解决的核心问题

面对上述问题，在设计多租户架构的过程中，结合实际的情况遵守了以下几个原则，可以作为设计参考：

- 租户的定义要非常清晰，以便于面对新场景的时候具备清晰的判断路径能够决策是否要新增加租户。
- 在设计租户的隔离性上，不仅仅要考虑多个维度的隔离性，还要考虑租户间的限制性以及安全性，防止相互影响，稳定性是首先要考虑的问题。
- 需要设计一套多租户的编码规范，以便于业务系统能够改造完之后具备多租户能力，并且尽可

仅限内部传阅

能少的让业务代码感知，这个编码规范最好能落地成 Maven 插件，以便于在编译期检测编码的合法性。

- 设计上需要考虑一个租户完整的生命周期，从租户的创建到消亡，从租户个性化业务定制研发、上线、运维整套机制。

13.3.2 核心技术

① 技术概览

在上述设计原则基础上，如何在整个国际化电商领域去实现这么一套架构，核心要解决的技术点以及方案如图 13-7 所示：

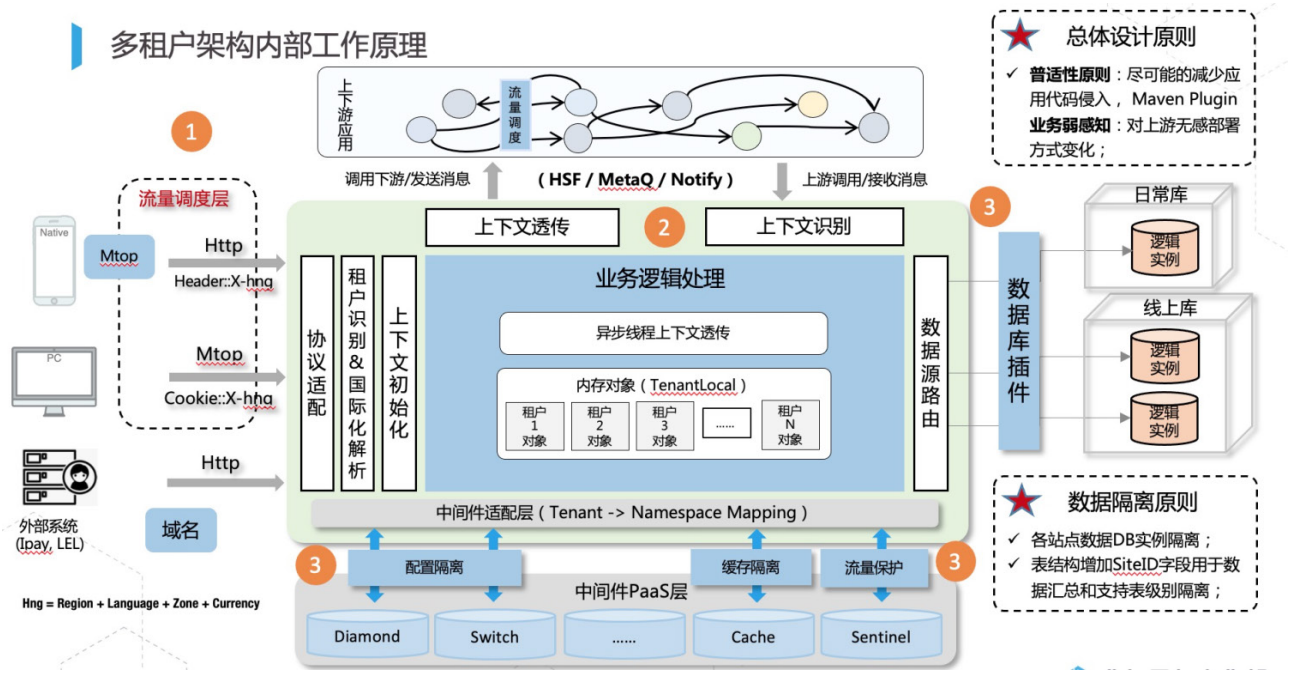


图 13-7 国际化多租户核心技术点

这个是多租户架构的运行原理，中间绿色部分是经过多租户改造之后的后端系统，那么从请求来源来看，主要分为 2 个部分，端的请求和外部系统间的交互，端包括 PC 跟无线，外部系统主要指的是像 IPAY,LEL 等请求交互的协议主要 2 种，HTTP 和 MTOP，那么针对任何一次请求，怎么识别出这是哪个租户呢？对于任何一个独立站点而言，他们的域名都是唯一的，那么完全可以基于域名来识别出租户；同时，框架还跟前端制定了请求协议规范，任何一次请求都携带了国际化特有的区域、语言、货币信息，为的是在入口处就可以解析出国际化信息；那么在完成了租户和国际化信息识别之后，下一步就要是完全整个上下文的全链路透传，那么首先是应用内部，需要解决异步线程的透传问题，在应用间的同步调用和异步调用，需要在请求的协议头上封装完整的交互协议，实现了应用间的透传，这里的透传功能跟鹰眼功能比较像，但是在技术选型的时候放弃了鹰眼的方案，因为鹰眼的整个产品定位跟当前的场景是不符的，租户标的透传需要保证 100% 的可靠性，而鹰眼存在丢失的可能性；在完成全链路透传之后，接下来需要根据链路上的租户信息来隔离租户的资源，这些资源很多跟中间件 PaaS 层相关，中间件每个产品基本上都有能够实现多租户的方案，但是最大的问题是没有形成 PaaS 层多租户的整

体解决方案，所以多租户框架进行了所有中间件的适配，开发了中间件适配层，用于解决 SaaS 层多租户跟 PaaS 层的映射关系，从而屏蔽了 PaaS 层多租户的技术细节。下面围绕多租户框架实现的几个细节进行描述。

② 租户识别 & 透传

一个系统支持多租户后，就需要从流量的进入到系统内部逻辑处理，再到数据的存取都携带租户信息用于唯一标识本次请求。默认情况下，一个国家站都有唯一的顶级域名，比如 lazada.sg 表示 lazada 新加坡站的顶级域名，那么当前台的用户通过 http 访问我们的系统时，在 mtop 等网关层就能根据域名唯一映射出一个租户标，并且在最顶层对当前的流量进行染色。当然业务系统也可以自定义租户的识别规则，比如有些系统就是用了一个相同的域名，但是在 http 参数上做了区分等就可以按自定义租户识别的方式。

租户标识的透传分为三个场景，系统内部的透传 / 系统间的同步调用透传 / 系统间异步调用透传：

- **系统内部透传**

系统内部的透传核心要解决的就是异步线程的问题，多租户框架封装了支持线程上下文透传的线程、线程池，业务系统需要替换相应的异步线程使用方式，同时对于租户标的透传跟保存方式采用了 TTL 的 ThreadLocal，父线程的上下文会被复制到子线程。

- **系统间同步调用透传**

系统的同步调用采用 RPC 的方式，多租户框架会在调用的发起端 (consumer) 端就会初始化透传的上下文信息，这个上下文会设置到 RPC 的上线文头信息中，传递给下一跳的服务提供方，服务的提供方在接收到请求后会根据上下文信息初始化当前线程的租户信息，如果初始化失败，则返回租户不存在的异常。当服务结果返回的时候 consumer 端的 response 接口会清理掉本次的调用上下文，并恢复发起请求前的上下文信息，也就是说没有存在一个全链路服务调用全局的上下文变量，而是在每次发起新请求的过程中临时生成，返回之后销毁的这种方式，这样也可以避免线程上下文被污染的可能性；另外框架也会在鹰眼上设置租户标的信息，以便于问题定位。

- **系统间异步调用透传**

异步透传主要指的是系统间通过消息的方式通信的这种机制，框架会为每一个消息的发送端裂变出每个租户的 TOPIC 并且初始化对应的 producer，也就是说从消息的生产端就是多租户隔离的，那么在消息的消费端框架也是需要裂变订阅不同租户的消息，这样从 TOPIC 维度就是可以识别当前的租户信息，另外框架本身在消息头上也携带了租户信息。

系统之间的交互示意图如图 13-8 所示：

仅限内部传阅

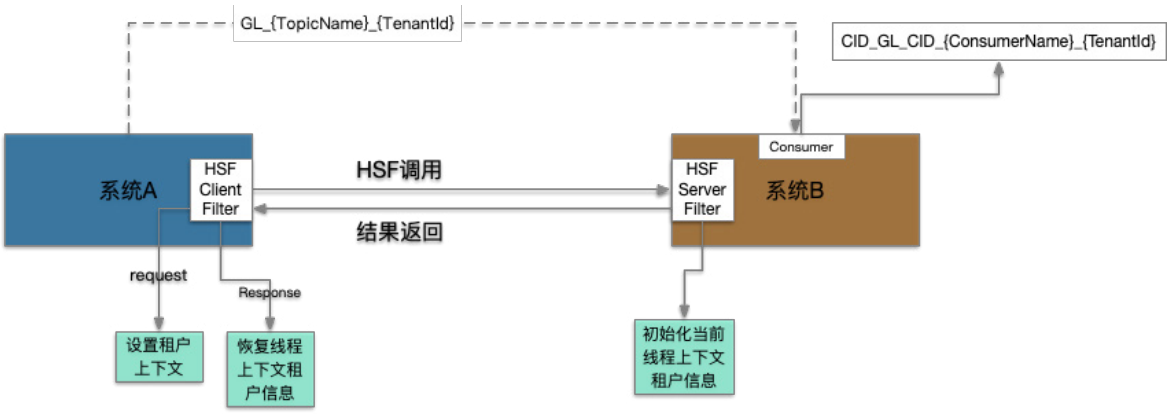


图 13-8 多租户交互示意图

③ 租户隔离性

- **数据隔离**

数据的隔离标准当前提供了 3 种方式，表结构维度的逻辑隔离，表维度的隔离，数据库实例维度的强隔离。

隔离方式	隔离级别	使用情况
表字段隔离	低	日常使用 / 部分业务线上使用
表名称隔离	中	未采用
数据库实例隔离	高	买家核心链路线上使用

- **表结构维度逻辑隔离**

表结构维度的逻辑隔离，隔离性相对比较弱，也就是约定了在表结构设计的时候增加 tenant_id（可自定义）的字段，用于区分不同租户的数据，多个租户的数据存储在一个 DB 实例中，对于数据的安全性保障上并没有这么强，对于安全性要求不是特别高的业务场景下可以使用，并且由于集团 idb 权限的控制是 db 实例维度，如果多个租户的数据存储在一起还要考虑数据权限问题。这种存储方式的成本是最低的，并且具备较好的可扩展性，新增加租户的时候如果容量足够是无需新申请 db 实例的。

在实现原理上，多租户框架提供了基于 MyBatis 和 tddl 的数据库插件，用于拦截每一条执行的 sql 语句，通过对 sql 的语法解析，动态的追加租户标识的查询条件和租户标识的写入，从而避免了在编码过程中需要额外的在 Mapper 文件中写关于多租户的查询条件等逻辑，也避免了存量应用的改造成本；

上述的隔离方案，目前一般的是日常环境中使用，在电商的场景下，还是比较要求数据的安全性的，加上国际化场景下的合规等问题，所以线上买家核心链路并没有运行这套机制，在周边的其他的场景还是有使用；但是基于成本以及未来国际化业务站点扩张的角度去看，这种方式是一种趋势

- **数据库实例维度隔离**

这种方式显而易见，通过为不同的租户分配不同的数据库实例的方式从而达到隔离的效果，这种隔离效果最强，但是也是成本最高的一种方式，有先天的优势，比如面对合规问题需要将数据隔离的场景下天然就是拆开的；

在实现方式上，多租户框架提供代理数据库访问层，在系统启动阶段维护了多个数据源跟租户的映射

关系，在每一次请求的过程中根据上下文的租户信息路由到具体的数据源进行数据的读写操作，这种方式在实操层面也有一定的局限性，连接的数据源过多会导致系统启动较慢等问题，虽然框架本身已经做了异步启动的优化；

- **表维度的隔离性**

表维度的隔离性顾名思义指的是不同的租户存储在不同的表中，这就好比正常的数据存储在正式表，压测流量的数据存储在压测表一个道理，这也是实现多租户数据隔离的一种方式，在隔离性方面介于第一种和第二种之间，但是在国际化场景下并没有尝试这种方案的设计。

- 配置隔离
  - **Diamond**

Diamond 配置隔离的方案是通过区分不同租户的 Group_ID 实现的，不同的租户分别需要增加租户标前缀，多租户框架对原生的 Diamond 进行了封装，可以根据不同 GROUP_ID 解析称对应的租户信息，然后针对当前进行配置信息的监听以及变更。

  - **Switch**

Switch 最基本的原理是通过定义 static 静态属性，并且配合配置平台进行开关的推送，存储在 diamond，并修改内存的值达到变量变更的目的；但是如果多个租户的开关值需要不同，由于 static 静态属性在内存中只能唯一一份存在，所以实现 Switch 的多租户需要通过动态字节码增强技术为每一个租户的每一个注册的开关类动态生成一份当前租户的开关类，在 NameSpace 上做区分并加载到 JVM。

比如原始的 NameSpace：
@NameSpace(nameSpace = "checkout.business.switch")
如果当前租户是 DARAZ_MM，生成的 nameSpace 如下：
@NameSpace(nameSpace = "checkout.business.switch(daraz.mm)")
其他部分的代码会全部拷贝一份。改造发布后，不同的租户会在不同的 namespace 下具备开关推送能力：

  - **配置项**

配置项的隔离性设计参考了 Spring-boot 的 @Value 方式，多租户框架提供了 @TenantValue 的属性注解，用于描述当前的属性不同的租户需要配置不同的值，在属性的定义上需要用 TenantLocal 进行修饰，用于存储不同租户的不同配置项；

- 流量隔离

对于流量隔离的方案，国内的成熟解决方案是 Dpath，针对同步的 HSF 流量、异步的消息流量都提出

仅限内部传阅

了隔离的方案，国际化多租户框架没有直接采用；原因如下：

（1）Dpath 强依赖 default 分组，如果流量上游未被识别，就跑 default 兜底了，这在国际化场景下不可能实现，国际化场景下做不到将所有的国家站都部署在一个 VM 上，由于不同的国家站的类目数据等原因，导致机器的内存等配置跟不上；

（2）异步流量的隔离，metaQ 消息的隔离 Dpath 采用在订阅端做过滤的方式，因为网络开销，流量区分等各方面都有消耗，更重要的是，如果下游不接 Dpath，无法兼容让下游系统增加过滤逻辑；

以上两个致命的问题，导致多租户的流量隔离不能直接用 Dpath，但是 Dpath 的设计思路是可以借鉴的，比如 HSF 的流量隔离方案，通过流量标跟机器分组的绑定关系规则，确定上游到下游调用过程中的机器选址；所以结合国际化多站点的场景，最终流量隔离的方案如下：

（1）HSF 同步调用：基于类似 Dpath 方案，每一次请求都会用租户标唯一标识这部分流量，也就是用租户标实现流量的染色；在 Mtop 层，往下一跳调用的时候，Mtop 集成了多租户 SDK，在这个 SDK 中会首先根据进入到 Mtop 的请求域名跟信息识别出当前请求的租户标识，然后根据租户标识实现目标业务集群的选择；当然这是比较理想的情况，上游都接了多租户 SDK，由于历史包袱原因，总是有一堆的应用没有接多租户框架，框架还提供了 HSF 版本裂变的方案，也就是说不同的租户，通过访问不同的版本号的方式实现流量隔离；

（2）MetaQ 异步逻辑：通过发送端的 Topic 隔离，同时消费端隔离；对于任何一个发布的消息，框架都自动代理增加了 Topic 后缀，不同站点的消息接收到的 Topic 就是隔离的，这样异步消息的流量也就天然隔离了；

图 13-9 展示了一个简单的示意图标识一次请求的隔离过程：

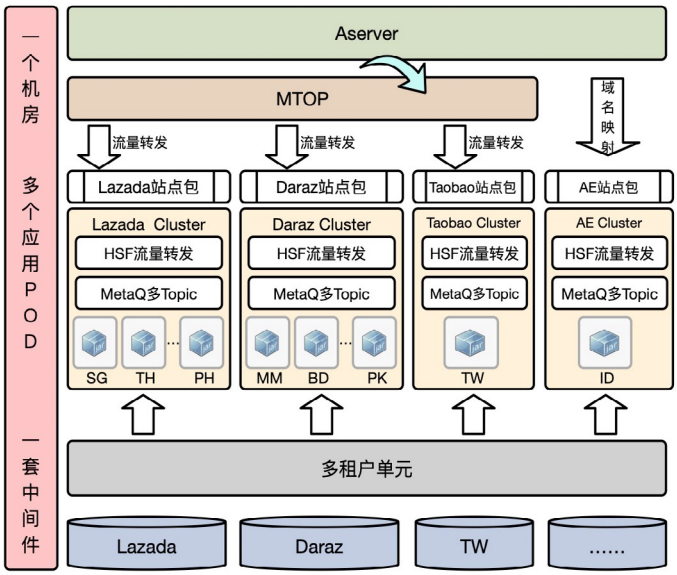


图 13-9 多租户请求隔离过程

- **资源隔离**

资源的隔离一般指的是在运行过程中租户访问的机器资源的隔离，比如 cpu, 内存等。常规的实现手段

有通过 jvm 多租户，为每一个业务租户分配固定的资源配比，这样可以有效的防止某个租户将所有资源耗尽导致整个集群雪崩的问题，但是在实现手段上也较为复杂，在运维等问题上也较为复杂，所以在国际化场景下并没有采用。当前在国际化场景下通过将不同的租户按站点的维度进行了资源隔离，也就是说 lazada/daraz/taobao/ae 之间的资源都是物理上相互隔离的，但是 lazada 多个国家站之间是共享资源的，也就是说存在一定的可能性会因为一个国家站的逻辑上死循环导致整个集群的 cpu 被打爆的问题。未来在隔离性方向上更倾向于通过容器隔离的方式，所以这一块未来还是有一定的发展空间，可以根据业务的需求自由的编排部署的方式。

- **其他隔离**

其他的部分包括 SchedulerX，通过不同租户配置不同的任务 Group 实现隔离，精卫的隔离实现目前需要业务自己完成。

④ 租户定制

不同的租户使用当前的系统软件功能过程中，必定有些个性化的定制逻辑，比如 lazada 有 redmart 生鲜类的业务，有手机充值类的业务等。这部分业务需要在交易系统上实现逻辑的定制，且这部分的定制逻辑只能归属于 lazada 这个站点。这部分定制逻辑同时也是跟别的租户完全隔离的部分，那么如何设计一套租户的定制框架保证整个多租户架构的完整性呢？

在讨论如何实现定制这个问题前，得先讨论技术实现上业务的定义原则。在国内，针对的业务划分原则，相信大家都有个深入人心的概念，就是垂直业务 + 水平业务，垂直业务代表了一个行业的业务特征，比如服饰类业务。水平业务代表了一类共性的业务特征的抽象，比如完成一次预售交易；如果用坐标系来描述，就是一个二维的业务特征，在某个坐标上就是整个业务叠加的效果，比如服务行业的交易通过预售的方式完成，必然有一些叠加后的业务特征；那么海外多站点场景下，业务的表达又发生了哪些变化呢，如图 13-10 所示，我们用一个三维坐标系描述了差异性：

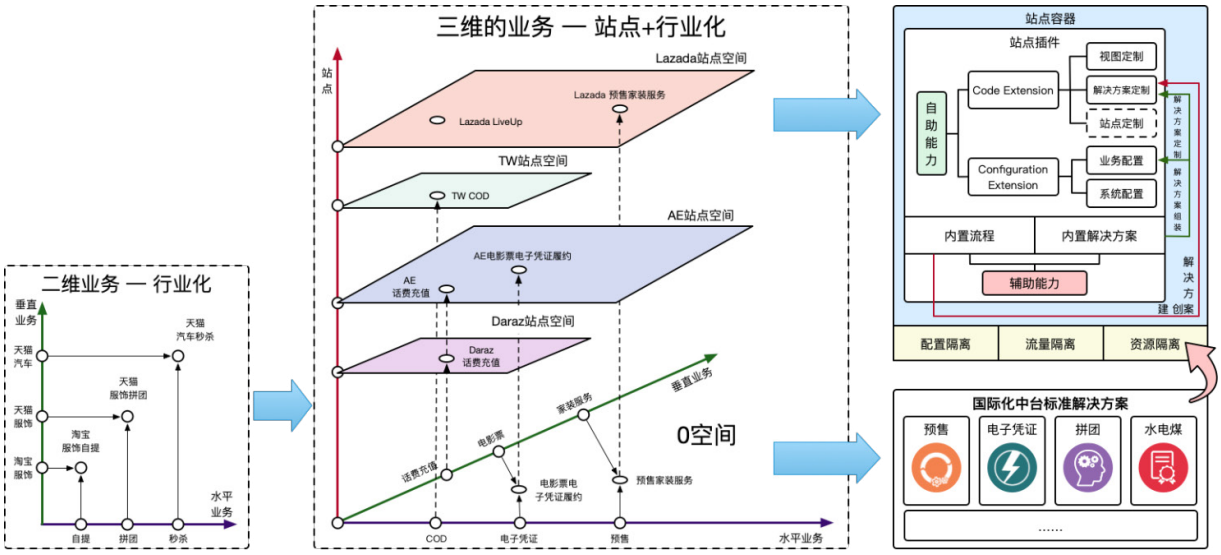


图 13-10 多租户下的的业务差异

仅限内部传阅

新增的第三维就是站点，一个业务真正的经营实体是国家站，也就是这个业务运营的“场”，那么国家站就相当于是一个大的业务空间，涵盖了多种不同的业务形态，业务成为了其中的一些元素。这里首次提到了“空间”这个概念，具体到国家站的空间一定是不相交的，也就是国家站之间的业务是独立运营的，如图所示的 LAZADA TH 站点空间，DARAZ BD 站点空间，处在不同的平面上；但是最开始国际化业务特征部分讲到的 Regional 的模式如何表达呢，从表达上是应该存在 Regional 空间的，这个空间内的业务元素是可以给 Regional 覆盖的国家站复用的，如图所示的 Lazada Regional 站点空间；另外，整个国际化中台团队也存在中台 PD 团队，负责设计面向全球复用的业务解决方案（有点像 App Store 上的 App 的概念），在三维坐标系上我们用站点坐标为 0 的平面去表达，也就是“0 空间”（解决方案）；至此，从抽象概念上，我们利用坐标系上空间平面的概念，描述了什么是一个站点的业务；

那么在工程实践部分怎么去表达这样一种设计呢？在国际化场景下，我们提出了站点包的概念，一个站点会存在一个对应的站点包，用于实现这个站点的个性化业务定制逻辑。一个站点下包含的多个国家站也必然会存在不一样的业务定制逻辑，并且在一定程度上可能也是隔离的，但是由于当前一个站点跟一个业务研发团队的配比是 1：1 的方式，所以站点包的划分维度也是按组织的形态划分的，在一个站点包内可以包含多个国家站的定制逻辑；一个完整站点包的定义如图 13-11 所示：

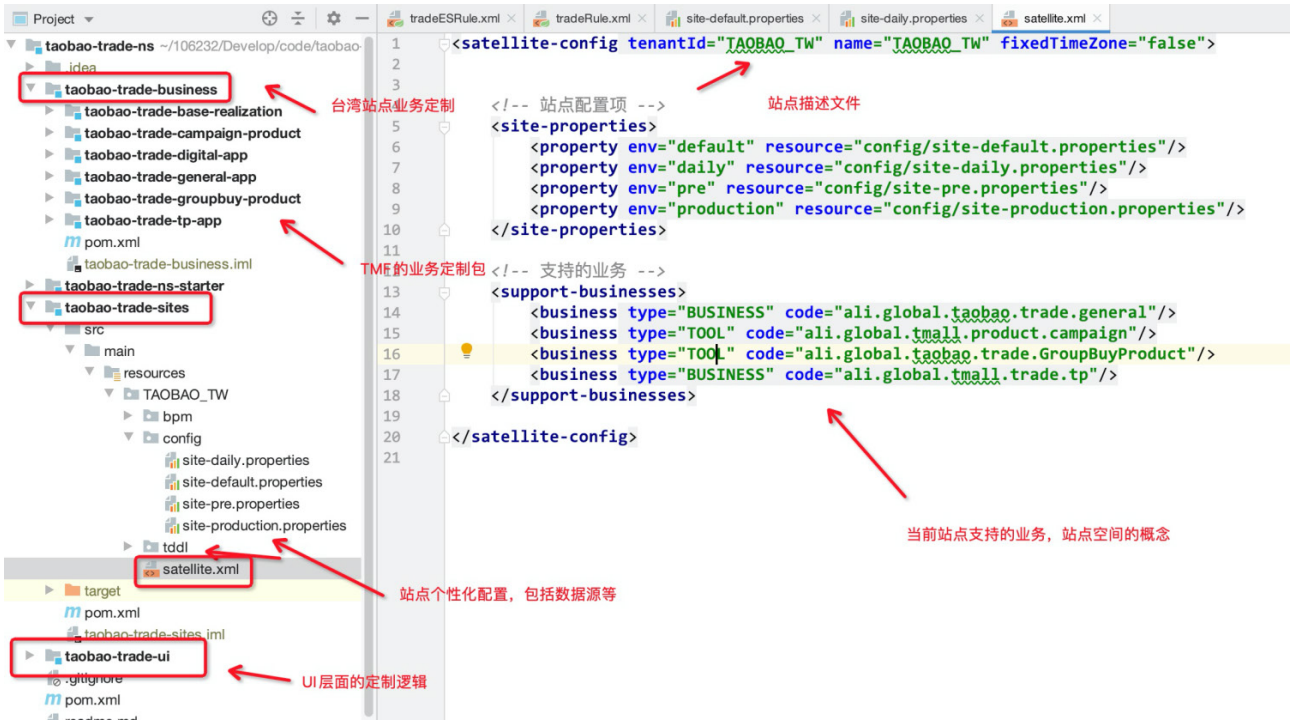


图 13-11 完整站点包定义

- 在研发态上，站点包包含了当前站点所有的定制逻辑，结合平台的逻辑就组成了一个站点的完整工程结构，在定制化的技术实现手段上可以基于 TMF 的插件化方式，也可以基于多租户框架提供的 SPI 的方式。
- 在交付态上，最早期的设计上站点包以插件化的方式进行交付，不同的站点集群会通过 profile 的方式打出不同站点的镜像包，也就是说 LAZADA 的集群只会部署 LAZADA 站点包 + 平台代

码的镜像；云原生之后，为了实现不同站点的运维隔离性，以及以业务为中心的架构理念下，站点包的交付态发生了本质的变化，升级为一个 AONE 应用，平台变更了被站点包集成的方式进行交付；

- 在运维态上，从一个中台大应用集中运维的方式变成了按站点分开运维的方式，彻底的解决了站点间的运维耦合；
- 至此，围绕着租户的概念（对应业务上的国家站），形成了国际化特色的一整套完成架构，从租户的定义，租户的识别跟透传，租户间的隔离，租户的个性化定制能力，并通过一套框架承载了整个设计理念；

13.4 未来多租户架构

整个多租户架构从 18 年开始提出想法，到 19 年第一个版本上线，到现在涵盖了买卖家 200+ 的应用，买家核心链路全部基于多租户架构，并且在此基础上构建起了容灾体系，整体运行平稳。但是还是有如下几个问题需要进一步优化：

- 整个多租户架构体系化程度不够，推行力度不足，多数人理解的只是一个框架而不是一套架构体系。
- 当前只在买家核心链路完整的遵循了多租户的设计规范，架构模式很难全链路推行将老链路迁移完成。
- 流量隔离等方案产品化体系建设不足，导致问题排查比较难，多租户架构答疑成本较高。
- 多租户 SDK 整体设计过于灵活，兼容了不同场景下的使用方式，缺乏最佳实践。
- 未来，随着 Nebula 产品的建设，围绕着站点为中心的主线，整套多租户体系的架构思想在产品化层面更上一层楼，从架构思想到最佳实践被继续完善跟传承。

仅限内部传阅

【第四篇】
未来展望

未来展望

随着 AE、Lazada、Daraz、天猫淘宝海外等国际业务的迅速发展，国际化中台面临着多样的市场环境、多地域的发展差异、全球化合规的要求、以及全球化的用户体验等多方面的业务和技术的挑战。我们从全球化部署、稳定性、容灾、上云、云原生、本地化、全球合规、成本优化、弹性、区域化、多租户等多技术方面来应对这些挑战。这一章，我们简要总结一下国际化技术架构之路，并对未来架构进行展望。

国际化业务发展历程

AE 发展

AE 是一个全球交易市场，使全球消费者得以直接从中国乃至全球的制造商和经销商购买商品，其主要消费者市场包括俄罗斯、美国、巴西、西班牙和法国。AE 特点的业务形态促使着我们技术架构的发展，比如部署架构中通过全球多区域进行部署和流量分发，合规架构中通过新加坡作为数据中心，同时为了进一步提升性能和稳定性，我们通过多租户以及区域化的部署来为全球用户提供服务，比如针对买家和卖家特性的单元化策略，以及从中心单元到区域单元的流量调度等，AE 的架构如图 14-1 所示。

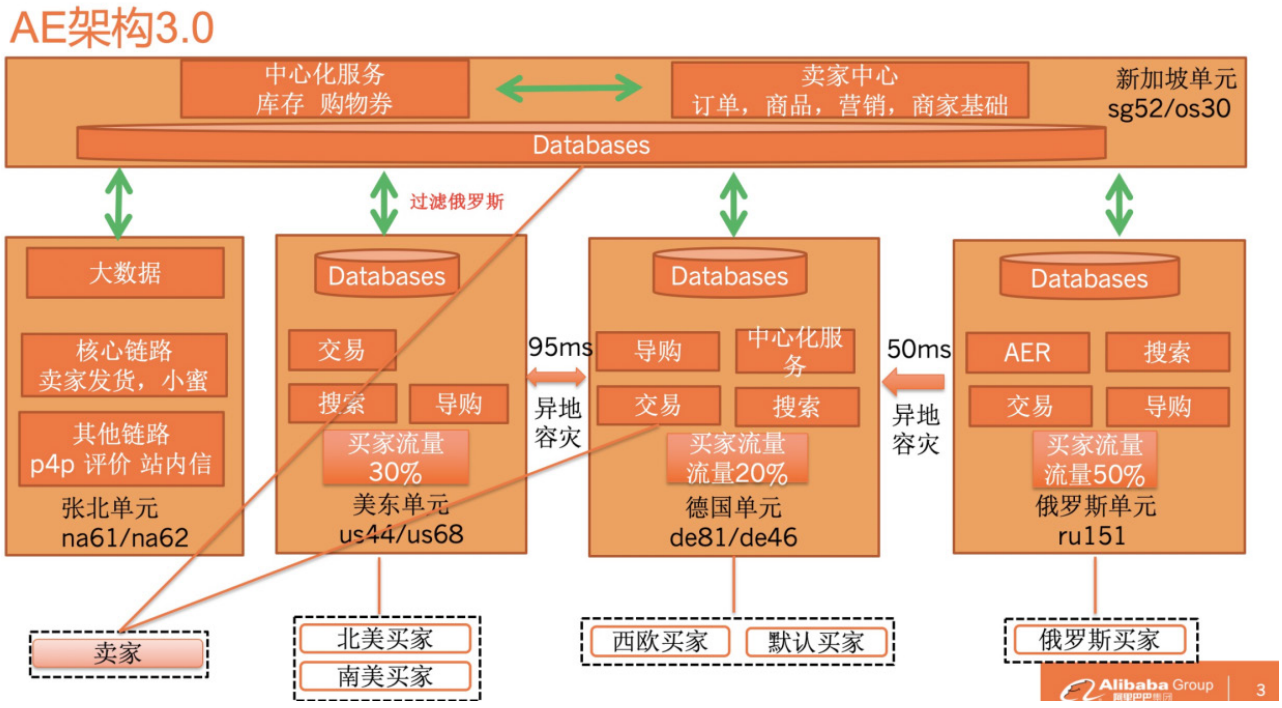


图 14-1 AE 架构示意图

仅限内部传阅

Lazada 发展

Lazada 是东南亚领先且快速成长的电子商务平台，于印尼、马来西亚、菲律宾、新加坡、泰国和越南六个国家经营业务，近几年发展迅速。遵循当地用户的习惯，风俗等，最开始我们为每个国家都搭建了一套 Lazada 的电商系统，并且相互独立，进而我们演进称为全球化的中台部署架构来兼顾成本合规等技术架构要求，并构建了端到端的部署容灾架构，比如图中所示的从中间件、存储到应用容灾架构方案，Lazada 的架构如图 14-2 所示。

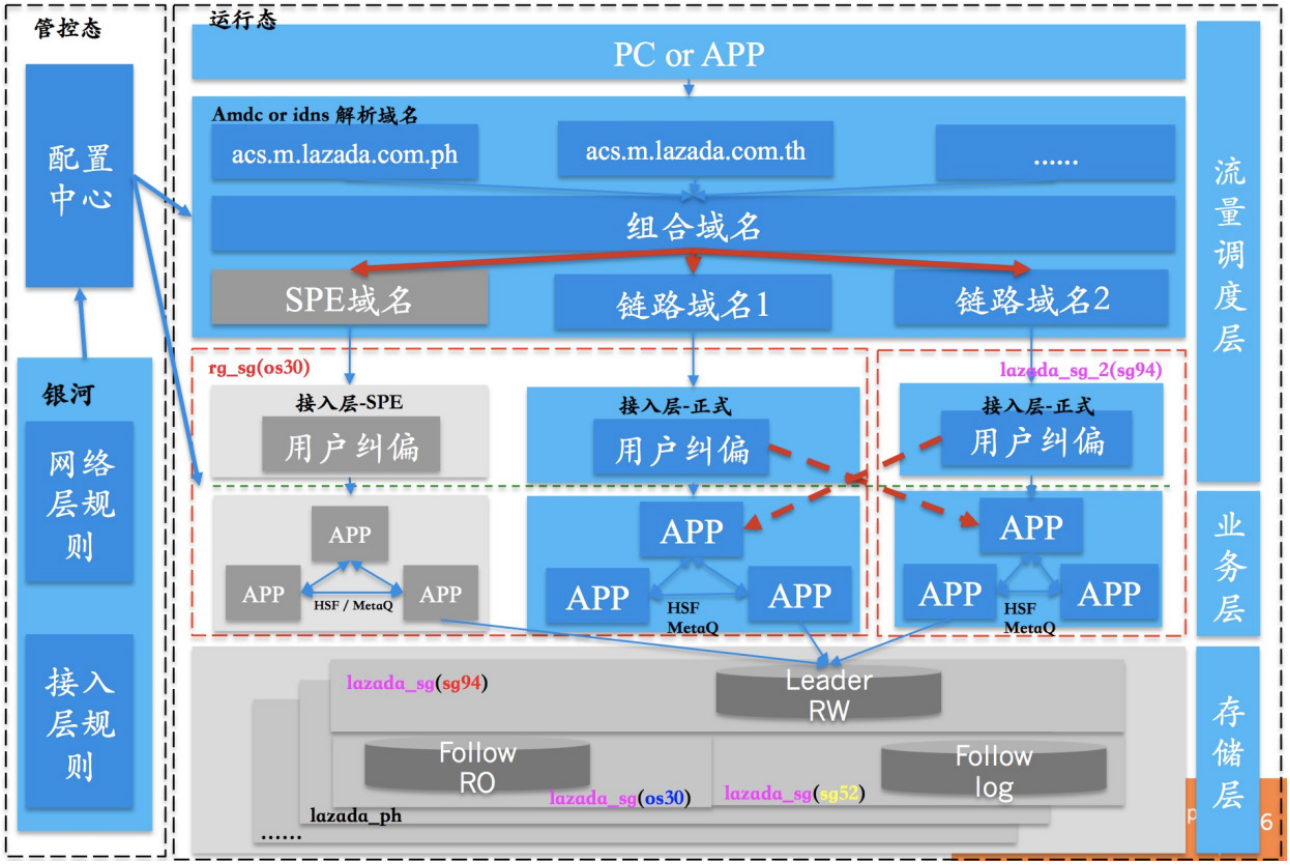


图 14-2 Lazada 架构示意图

国际化技术架构历程

为了适应国际化技术架构的发展，我们从标准架构、云化架构、以及中台架构角度进行简要回顾。

1）标准架构

标准的技术架构体系往往与成本、性能、稳定、安全相关，针对国际化特殊的技术特性，近些年来我们进行了如下的实践：

成本

随着 AE、Lazada、Daraz、天猫淘宝海外等国际业务的迅速发展，应用规划已经超过 2k+，新 / 旧系统改造、业务融合、组织架构调整等原因，很难简单直观的了解资源成本使用情况，评估其合理性。与此同时，技术资源费用上很高，同比业界其他国际电商成本偏高的现状。我们结合过去的探索，完

成了成本数据、成本分析、成本优化等多方面的探索，同时结合沉淀的成本优化框架，提出对成本意识的重要性。面向未来，可以仅以一步在成本分析、成本监控、成本预期进一步完善，做到全球化成本可控。

部署

我们针对不同的业务建立了不同的部署架构：

- **对于 AE**：业务场景是跨境电商，商品发布是国内，卖家信息在一个站点发布；部署架构可以有本地部署方案（一套系统服务多个国家）和全球化部署方案（买家在 top 国家部署本地站点）。
- **对于 LAZADA**：我们为每个国家都搭建了一套系统，进一步演化出可以分为全球化独立部署方案（6 个国家独立部署）和中台部署方案（通过中台进行整合统一部署）。

高可用

针对国际化全球底层基础设施复杂、大促活动多、高可用目标要求高等特点，我们构建了全球化的稳定性保障体系，比如包括

- **日常稳定性**：容灾快恢体系、灰度体系、监控体系、限流体系等
- **大促稳定性**：大促标准化 SOP 体系、链路梳理、压测模型、预热等

合规

合规前，旧的部署架构在应对一些合规要求时无法满足的，主要由于数据中心是按照中国和美国来建的，而在一些合规场景中，可能存在这些数据中心禁止传输等风险。合规后，新加坡作为世界上数据合规要求比较高且较多处于白名单国家之一，比较适合做数据中心和中转地区，所以我们将新加坡作为数据中心，承载了业务的全量数据，现在越来越多互联网公司也选择新加坡建立数据中心。

2）云化架构

上云

国际化中台的上云之路经过了几年的发展，也实现了核心系统全面上云，其中成了全球化 9 个云上机房基本完成 100%ASI 化，云原生基础设施搭建完成，以及全球化海外机房完成核心机房 Cloud Hosting。大体上经历了如下三个阶段：

- 初步完成基础设施上云（Lazada 和 AE 核心业务生产上 ASI，并承载大促和日常 50% 的流量）
- PaaS 和云原生的探索（业务规模化上云，云上 IaaS 承载主要流量，验证 PaaS 云化能力）
- 全面拥抱云原生（技术栈与云原生基础设施完全融合，国际化云基础设施布局全面成型）

云原生架构

随着云原生技术的升级和阿里经济体全面上云，国际化全面拥抱云原生架构，我们进行了如下实践：Service Mesh 微服务架构、GitOps 研发模式、 Serverless 无服务器基础设施的运维。过程中对研发的体验、交付效率产生深刻的影响；比如 2020 年初成立“国际化云原生样板间战役”，通过云原生在

仅限内部传阅

国际化团队的试点来为集团全面云原生化做好摸底，从 5 月份的 OK 到现在大半年的时间内，在基础架构域、业务架构域都沉淀出了一套适合阿里架构的标准国际化云原生体系。同时，我们基于云原生重新定义软件研发模式，将平台定义为业务的基础设施，以基础设施的交付形态将平台能力交付给上层的业务实现，从而提高研发交付效率。

3）中台架构

国际化中台孵化于集团的业务平台，所以在中台建设上继承了业务平台的很多思想和经验，国际化中台的核心系统也直接缘于业务平台。国际化中台主要经历了如下几代演进：

- 单业务（电商核心系统开始和业务平台的系统进行融合）
- 中台雏形建立（同时支持本地 + 跨境交易模式的国际化中台模式）
- 中台多业务支撑（4 个国际化平台版本统一）
- 中台开源新阶段（通过开放标准、共享内核、共享协议支撑全球化研发效能大幅提升）

未来展望

为了集团全面走向全球化的战略目标，面向国际化技术架构的未来，我们将从如下几个方面继续探索前行：

1）全面云原生化

云计算的下一站是云原生，云原生让业务应用更加轻量、敏捷、高度自动化。通过国际化云原生样板间半年时间的尝试，我们已经完成了云原生的基础设施升级和一些业务架构的探索，包括使用 ASI 的 Serverless 能力覆盖核心交易导购链路，区域化路由逻辑下沉到 AliMesh 的广泛尝试，使用 IaC+GitOps+OAM 技术对国际化云原生运维的实践，以及轻量化容器、Dockerfile 统一治理等。接下来，我们也将继续跟阿里云团队深度合作，用云原生的技术进一步提高研发效率，比如进一步的应用容器弹性、Serverless、ServiceMesh 等核心技术的全面云原生演进，逐步构建全面的基于云原生的国际化新一代架构体系。

2）深化大中台战略

继续贯彻业务平台的指导方针，继续深化国际化中台战略，进一步推进 API FIRST + KEEP SIMPLE 的开源模式。对各领域交互协议进行标准化治理，规范各领域对外的接口定义，并构建核心最小集，严格控制内核 API 的升级迭代；同时各域内核精简，将域业务更相关的内容上移到站点包，如流程编排能力上移，平台部分通过集市层和教堂层的分治，进行面向业务语义的模块化改造，内核能力开放灵活，对使用者提供确定性且稳定的技术内核能力。我们希望能够实现 996 和 211 的目标。996 重点是 6 周 3000 人日以内交付一个新的站点，211 是主要的业务需求能够在 2 周内完成交付，其中 1 周内完成研发，1 小时内完成发布。通过基础模型和商业能力的白名单建设，并辅以全球化架构委员会和产品技术小组定期 review，多团队有效协同，将全球化的基础设施、技术框架、基础领域模型、资损敏感度

非常高的领域、系统稳定性要求非常高的领域等方面更好的收口到中台。

3）数据驱动

数据驱动 PDCA 是工业界非常重要的理念，即 plan do check action 的循环迭代机制，非常适合国际化技术架构的演进。过去的几年，我们通过多种方式进行了多维度的数据驱动的尝试，接下来，我们将进一步通过数据驱动的手段，加强对技术架构的优化迭代和度量管控。比如通过成本数据的分析进行成本优化、对运维效率的分析进行运维效能优化、对研发过程的自助率和变更质量、业务和平台指标度量 SLA 体系等驱动，全面通过数据思维进行研发运维效能的全面提升。

4）技术产品升级

我们将逐步推进产品化思维和沉淀，和业务团队一起沉淀更多标准化的产品，打造面向全球化的稳定、高效、合规、开放的中台新标杆。带来面向业务和平台同学，打造面向平台和站点的需求管理、开发设计、测试交付与运维管理工作台。进一步降低业务研发成本，提升业务交付效率。同时在技术层面，通过规范化、自动化的产品工具，以及统一的交互体验，沉淀灵活稳定的国际化基础架构与应用架构标准，安全生产规范和合规标准通用能力，同时基于产品化打造国际化的品牌，并促进全球化的研发效能持续提升，让业务创新更简单。

面向国际化未来的星辰大海，我们仍在路上！

仅限内部传阅

附 1：术语解释

- **全球化（Globalization）**：是面向商业决策者，为产品和服务走向全球市场而制定战略安排与规划，包括市场推广，国际品牌战略，合规运营等一系列全球范围内的商务活动。
- **国际化（Internationalization）**：是面向技术和产品人员，将产品和服务与特定的地区解耦，通过标准统一的形态适用于任何地方的潜力。一些国际化相关的技术比如全球化部署、高可用、容灾处理、安全生产、代码部署分离等。
- **本地化（Localization）**：是面向技术和产品人员，让产品和服务更适合于特定地方的使用，让用户感觉这款产品是为他们量身定制的，是全球化和国际化的“最后一公里”。一些本地化的工作比如多语言、多时区、多币种内容等。
- **架构（Architecture）**：架构的本质是对事物复杂性的管理，是环境中该系统的一组基础概念和属性，具体表现就是它的元素、关系，以及设计与演进的基本原则。
- **企业架构（Enterprise Architecture）**：企业信息管理系统中具有体系的、普遍性的问题而提供的通用解决方案，一般包括业务架构、应用架构、数据架构、技术架构。
- **技术架构（Technology Architecture）**：是将业务需求转变为技术实现的规划过程，确定组成应用系统实际运行的技术组件、技术组件之间的关系，以及部署到硬件的策略。
- **架构风格(Architecture Style)**：解决某一类型问题的架构表现方式，比如单体架构、微服务架构、云原生架构等
- **全球化部署（Globalization Deployment）**：基于全球化的用户体验、成本、数据、合规等因素决定的部署架构形态，包括本地部署、跨境部署、独立部署、中台部署等。
- **全球稳定性（Globalizatin Reliability）**：针对全球化系统专门的设计，从而减少系统停止服务的时间，而保持服务的高可用和稳定性。
- **灰度发布（Gray Release）**：一种平滑过渡的一种发布方式，提供快速低成本试错的机制，变更系统对线上生产环境的发布变更操作建议通过灰度手段进行。
- **系统监控（System Monitoring）**：收集、处理、汇总，并且显示关于某个系统的实时量化数据
- **系统预热（System Warmup）**：系统提供服务进行预先准备，防止瞬时抖动导致系统雪崩，可分为冷数据预热和冷系统预热
- **容灾（Disaster Recovery）**：是指建立两套或多套功能相同的系统，互相之间可以进行健康状态监视和功能切换，当一处系统因意外停止工作时，整个系统可以切换到另一处，使得该系统功能可以继续正常工作。
- **上云（Cloud Hosting）**：是指企业以云计算的服务能力为基础，进行信息化系统以及相关业务

仅限内部传阅

应用的建设并对外连接的过程。

- **云原生架构（Cloud Native Architecture）**：基于云原生技术的一组架构原则和设计模式的集合，将云应用中的非业务代码部分进行最大化的剥离，从而让云设施接管应用中原有的大量非功能特性，使业务不再有非功能性业务中断困扰的同时，具备轻量、敏捷、高度自动化的特点。
- **全球合规（Globalization Compliance）**：是指国际化的经营活动需要遵守法律、法规、监管规则或标准，对用户的隐私数据进行保护，包括本地存储和跨境传输等多方面。
- **成本优化（Cost Optimization）**：是指对资源成本的分析和优化，包括成本基础数据、成本优化策略、项目组织架构、成本意识。
- **弹性（Scalability）**：体现一个系统是否能够随着压力的增减而动态的对资源进行调整的能力。
- **区域化部署（Regional Deployment）**：按需在多个地理区域部署应用程序，全球区域化部署就是在全球的多个区域部署应用程序为用户提供服务。
- **多租户技术（Multi-Tenancy Technoloty）**：也称多重租赁技术，是一种软件架构技术，它是在探讨与实现如何于多用户的环境下共用相同的系统或程序组件，并且仍可确保各用户间数据的隔离性。

仅限内部传阅

附 2：参考阅读

参考文章

- Alibaba.com 全球化技术探索之路
- 阿里云云原生架构白皮书
- 阿里巴巴集团开发规约
- 蚂蚁国际技术架构规范
- AWS Well-Architected Framework
- Microsoft Azure Well-Architected Framework
- 全球化技术架构系列之一『三边战略』
- 架构制图：工具与方法论
- 国际化中台建设思路 - 在宽
- 云上应用架构指南：完整手册
- 从获得 CTO 线最佳子战役想到的
- 全球区域化部署架构
- 面向全球用户的大规模网站架构
- 如何做好一名稳定性 SRE
- 从国际化云原生样板间看云原生下业务架构升级
- 资损防控编码规约
- ASI 用户使用文档
- 云原生多模数据库 Lindorm 2020 双十一总结

推荐阅读

- 全球化技术架构系列之二『基础设施 5+5』
- 从 Voyager 到 Polaris，我们到底做了啥
- 全球化卫星发射站 -- Halo-satellite
- 积沙成塔，造全球梦 - 国际化中台商家技术 2016-2020 技术发展总结
- 架构落地：顺势而为，把无奈变故意
- ASI 调度器 - 系统解析百万级云原生调度内幕

仅限内部传阅