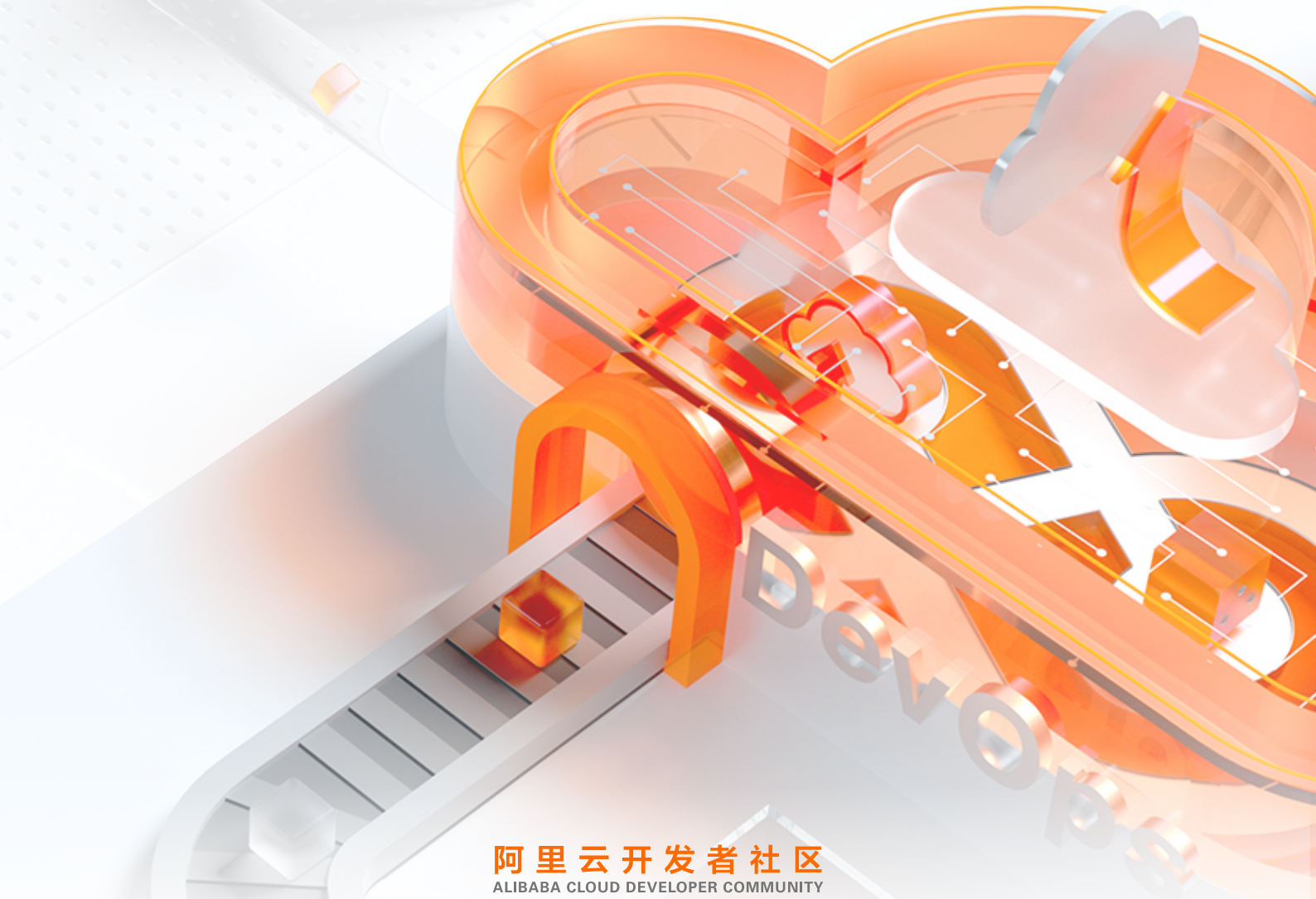
阿里云云效
云原生时代新DevOps平台

ALPD系列丛书

阿里巴巴DevOps实践指南（2021）

从 DevOps 到 BizDevOps



阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY



钉钉扫码加入
阿里巴巴 DevOps 交流群



扫码了解
阿里云云效



阿里云开发者“藏经阁”
海量电子书免费下载

作者团

编写团队

阿里云云效团队

编写委员会

何勉、陈鑫、张裕、黄心懿、崔力强

编写组成员

何勉、陈鑫、张裕、崔力强、邢凯航、胡飞虎、陶彬贤、
喻煜阳、胡晓宇、王蒙、魏洪波、张小菲、张伟、吴丰、
周新生、陆叶平、胡作政、何想、周楠、王特

目录

01 DevOps概述

1.1 DevOps 的起源 010

1.2 DevOps 的目标是支持业务敏捷 015

1.3 阿里巴巴 DevOps 实施的价值主张 020

02 业务驱动的协作和产品导向的交付

2.1 需求的层次结构 026

2.2 业务驱动的协作 030

2.3 产品导向的交付 038

2.4 规模化实施路径 044

03 建设和提升持续交付的工程能力

3.1 以特性为核心的持续交付 051

3.2 本地开发 055

3.3 云端开发 065

3.4 代码评审 069

3.5 代码检测 073

3.6 测试提效 082

3.7 测试环境与路由 091

3.8 应用环境能力 097

3.9 基于应用和变更的交付模式 103

3.10 提升构建的效率 109

3.11 基于制品元数据提升交付效率 115

04 基于应用的持续运维

4.1 监管控一体化运维 124

4.2 业务系统安全工程 129

4.3 全景监控 137

4.4 发布策略 146

4.5 编排运维 156

4.6 智能运维 163

05 阿里巴巴 DevOps 工具体系 173

06 DevOps 能力提升模型 180

序

数字化是一次产业革命，DevOps 是其中的重要组成部分。

每一次产业变革都从革命性的技术开始，围绕它建立新的基础设施，形成新的技术和管理范式，最终彻底变革整个产业。那些及早建立新基础设施，并引领新技术和管理范式迁移的经济体将从一轮产业革命中崛起。

第一次工业革命建立了蒸汽动力和机器生产的基础设施，形成了工业化的基本范式，英国从中崛起。第二次工业革命建立了内燃机和电气化的基础实施，形成了科学管理的基本范式，美国和德国实现超越。

数字化革命发端于 1970 年代的美国硅谷，它以信息化为肇始，带来互联网在全球蓬勃发展，而这只是开始。今天，数字化变革正深入每一个具体产业，驱动深层次的产业变革，让企业对客户、市场和环境的反应更精准、灵活、即时和高效。数字化和产业将发生化学反应，客户体验和产业效率都必须发生质的提升，这是数字化变革给每个产业的命题和挑战。

以云计算、大数据、人工智能、IoT 等为代表，新的数字化基础设施已经就绪，并将继续演进。更重要的是与之配套的技术和管理范式的转变，如：数字化转型路径、面向数字化的组织、数字化的全链路协同、数字化时代的 IT 研发模式。其中 DevOps 已成为数字化时代 IT 研发和管理范式，并成为企业数字化转型重要的组成部分。

DevOps 是什么？企业该如何建设 DevOps 能力？我们必须在数字化转型的背景下加以考察，具体来说包含业务交付和系统运行两个方面。

第一：持续顺畅高质量地交付有效价值。它的目的是缩短从业务想法的提出到实现和交付的时长，使这一过程更加顺畅和精准。数字化的组织，要围绕这一目标构建技术工程体系和协作模式，消除业务需求交付过程中的一切阻碍和等待，让 IT 交付节奏，跟上业务发展的需要。

第二：极致弹性和韧性的系统运行。IT 系统必须满足业务运营的要求，具备极致的弹性和韧性。弹性是指它随业务负载自动、实时的扩缩容，以精准的弹性和合理的成本满足业务；韧性指的是确保系统安全、合规和稳定的运行，实现系统运行的连续可用性和安全稳定。

以上两者，都要求打破 IT 开发部门和 IT 运维部门之间的隔阂，包括技术、流程和组织上的隔阂，也就是所谓开发运维一体化（DevOps）。一体化是手段，最终的成果必须体现为顺畅的业务交付和连续稳定的系统运行。而这一切都必须以业务为源头，打通从业务到开发再到运维的整个流程，实现业务、产品和运维的有机融合，形成高效的业务交付、运行和反馈闭环。严格意义上 DevOps 应该是包含业务在内的 BizDevOps，延续业界的习惯，我们还是称其为 DevOps，不过其内涵一定是包含业务，且以业务为根本驱动的。

DevOps 是数字化时代的研发新范式，需要协作模式和工程技术两个方面的变革，这两者相辅相成，又相互制约。阿里巴巴 DevOps 实践指南，将分别从协作模式和工程技术两个方面展开。工程技术对应研发敏捷，协作模式对应组织敏捷，帮助组织顺应数字化变革的要求，让技术发挥其核心作用，加速业务发展，引领业务创新。

在数字化变革的浪潮中，中国作为产业规模最大和门类最齐全的经济体，迎来百年未有的崛起机会。拥抱数字基础设施，探索符合数字化时代要求的技术和管理范式，将帮助我们切实把握机会，实现伟大的复兴。而把握这一机会的组织，将在数字化变革的浪潮中脱颖而出。DevOps 是其中不可或缺的环节，希望我们的实践总结对你有所启发。

推荐语

数字化转型是从社会到企业的整体转型，需要顶层设计和统一规划。数字生态最终是一个高度连接的社会而非一座座割裂的竖井。阿里巴巴 DevOps 实践指南说明，一体化的开发和运维要支持的目标正是全链路、全生命周期的业务，这是技术发展的方向，也是数字生态的必然要求。

——IBM 副合伙人、《银行数字化转型》作者、极客时间《说透数字化转型》专栏作者 付晓岩

企业的数字化转型必然绕不过 DevOps，核心点还是 IT 如何赋能业务，创造价值。该本指南书体系思路非常清晰，个人理解是从交付态和运行态两个视角去阐述，并完全以应用为视角。交付态也就是今天 DevOps 里面提到的核心工程实践：持续交付。持续交付是对过去割裂的 IT 组织交付模式的一次革命；运行态，是从连续性运维的角度去探讨，其中又引入了多个不同的最新实践，如智能化运维。难得的是，书中的很多实践都来自于阿里实际，实践完善了理论，理论才可以更好地指导实践。的确是一本难得的实践指南书！

——优维科技创始人&CEO 王津银

DevOps 与云化开发平台的结合实现了软件开发工具平台的一次飞跃，不仅实现了软件开发工具的集成化和流水线化，而且使得基于大数据分析的软件开发质量与效能提升成为可能。《阿里巴巴 DevOps 实践指南》的推出为我们了解业界 DevOps 和云化开发实践、开展软件开发大数据分析研究工作提供了重要指导。围绕相关话题，学术界和工业界有望开展更多的交流与合作，共同推进软件工程研究与实践的发展。

——复旦大学计算机科学技术学院副院长、软件学院副院长、教授 彭鑫

过去 10 多年，阿里云在 IT 基础设施方面做了非常多的探索和努力，得到了客户和社会的认可。云、大数据、AI、IoT 等，已成为新一代数字化基础设施。如何让这些基础设施发挥更大作用，推动产业数字化变革，这也是我们持续努力的方向，DevOps 能力是其中的核心环节之一，希望《阿里巴巴 DevOps 实践指南》对你有所启发，成为你在数字化转型道路上的伙伴。

—— 蒋江伟 阿里巴巴合伙人

技术创造新商业，技术已成为数字化时代业务创新和发展的核心环节。提升技术的业务响应和交付能力，并保障系统运行的连续性和稳定性，已成为数字化组织的共同挑战。《阿里巴巴 DevOps 实践指南》源自阿里巴巴多年的一线实践，并做了系统性的沉淀。不管是工程、协作、运维还是工具，希望你能有所收获，并用以指导实际工作。

—— 刘国华 阿里巴巴研究员、混合云平台负责人

从 B2B 到淘宝、从跨境电商到本地生活，面对如此丰富、如此大规模的业务交付需求，阿里巴巴在软件交付效率上一直走在行业的前列。为此，公司从技术文化、技术架构、软件基础设施及平台、以及流程上做了全面和深入的设计和优化。《阿里巴巴 DevOps 实践指南》总结了其中精华的部分，并详细解释了实践背后的思考，指南的内容主要是由工作在一线的资深工程师撰写，读来体感强烈，推荐给大家。

——许晓斌 阿里巴巴高级技术专家

DevOps概述

01

阿　里　云　云　效

01

1.1 DevOps 的起源

谈到 DevOps，就必须从敏捷和精益开发说起。DevOps 在它们基础上发展而来，借鉴了其中的方法、理念，并发展和完善了它们的实践体系。

■ 敏捷软件开发的兴起

敏捷软件开发的实践最早出现在上世纪 90 年代。当时，一批轻量的软件工程方法和框架相继诞生，它们共同的特点是，相对传统软件工程，都遵循演进和迭代的模型，过程更加轻量灵活。其中 Scrum 和极限编程（ExtremeProgramming）在实践上最为成功，影响最大。它们都是迭代和增量的软件开发框架，区别是 Scrum 只包含管理实践，而极限编程则同时涵盖工程和管理实践。

上世纪 90 年代，PC 软件流行和第四代编程语言的出现，面向对象和设计模式运动的兴起，让小型项目的开发蓬勃发展。同时，互联网应用和开源社区兴起，有别于传统的开发模式不断涌现，优秀个人在程序开发中的作用得到凸显。

这些因素都让非传统开发方法有了实验的土壤。其结果是，一方面质量问题层出不穷，这部分促使了源自全面质量管理体系的 CMM/CMMI 在这一时间的繁荣和推广；另一方面也产生了许多不同于传统方法的有效实践，让业界看到了新的可能。敏捷运动这时已经呼之欲出，它既是对传统的反叛，也是对野蛮生长的规范。

2001 年 2 月，17 位轻量级软件工程方法的代表人物，齐聚美国犹他州的雪鸟滑雪胜地，其中也包括 Scrum 和极限编程的几位发明人。在两天的会议之后，发布了后来产生巨大影响的敏捷宣言（参见 http://agilemanifesto.org/），敏捷宣言陈述了他们共同认可的关于软件的开发方法的理念，同样重要的是他们找到了敏捷这个词来总领这些理念。

敏捷概念在 2001 年出现，可以说适逢其时。当时，一方面传统方法变得越来越臃肿笨重，却没有解决软件危机；另一方面，人类正在进入互联网时代，软件业对响应变化和创新的要求迅速升级，这是更根本的原因，毕竟需求才是行业发展的最好助推剂。

敏捷宣言发布后，敏捷成为了一场运动，被迅速地推广和应用。但是，早期的敏捷专注的还是研发交付阶段，站在业务的角度，它的目标是帮助产品和研发团队提升敏捷响应能力，也就是：“更早地交付价值，更灵活地应对变化”。然而，在企业数字化转型的背景之下，IT 不仅要保证产品的开发和交付，系统部署和运行同样重要。DevOps 继承了敏捷开发的理念，又补上了运维的部分，但 DevOps 绝不是开发和运维的简单叠加，这个我们后面还会说到。

精益产品开发的出现

精益思想最早源自生产制造领域，根源于丰田在产品制造中管理和工程实践。1988 年《斯隆管理评论》的一篇论文《精益生产系统的胜利》比较了西方的生产方式和丰田生产方式在效率和质量上巨大差距，挑战了规模生产带来效益的神话。从此，精益开始进入西方的视野，逐渐成为现代管理学的重要组成部分。

《精益思想》一书将精益定义为：有效组织人类活动的一个新的思维方法，目标是消除浪费，以更多地交付有用的价值。书中进一步总结了精益的 5 个原则，同时也是精益的 5 个实施步骤：

1. 定义价值：站在用户的视角定义什么是价值，并把它描述为具体产品或服务；
2. 识别价值流：识别和映射创造价值的流程步骤，消除不增加用户价值的步骤和活动；
3. 让价值持续流动：让用户价值在流程步骤中流动起来，使它们持续、顺畅地流向最终用户；
4. 用户价值拉动：由用户价值拉动流动，避免不带来用户价值的浪费；
5. 精益求精：不断重复 1 到 4 步。追求完美的价值和价值流动，消除过程中所有浪费。

在这个抽象层次上，精益思想超越了其诞生的制造业，深刻影响了各个行业，如精益政务、精益医院、精益领导力、精益服务业、精益供应链、精益教育等，这其中也包含产品开发。事实上，主流的敏捷开发方法都直接受到了精益思想的影响，遵循精益的基本原则。

与此同时，精益产品开发作为独立的实践体系也快速发展，它聚焦两个方面的目标——价值交付过程和价值本身。

第一，关注价值交付过程。其中比较有代表性的是“精益看板方法”，它由 David Anderson 在 2006 年左右基于自己的实践发展而来，并在 2010 年出版的《看板方法》一书中加以系统总结。“看板方法”是精益思想在软件开发中具体应用。它从可视化需求交付端到端的价值流开始，通过系统的实践提升需求的流动效率，并确保流动的过程质量，从而实现端到端的系统改进。

“看板方法”为代表的这一类精益实践的本质改变是：从关注资源的使用效率，转变为关注价值的流动效率。这也带动使用者从过去的局部优化转向端到端的全局优化。

第二，关注价值本身。其中比较有代表性的是“精益创业”。精益创业的实践最初由 Steve Blank 基于自己和其他硅谷的创业实践发展而来，Eric Ries 在《精益创业》一书中对精益创业的理念和实践，做了系统的总结，并让精益创业的理念迅速普及。

精益创业认为创业是一个巨大不确定的过程，其最大的浪费是交付没用的（不能解决用户问题，或带来业务成功）的东西。为此它把价值的探索和发现融入产品交付过程，提出了著名的“开发–测量–学习”循环。循环从关于市场和产品的待检验概念开始。接下来，循环的第一步是开发用以验证这一概念的最小可行产品（MVP，Minimal Viable Product）；第二步：基于最小可行产品收集市场、用户的反馈，并获得测量数据；第三步：用数据验证假设，证实或证伪它们，并加以调整，产生经实证的认知。然后，进入下一循环，持续探索商业模式和产品功能设计。

精益创业的影响远超初创公司，事实上“精益创业”一书中把“创业”定义为在不确定的环境下，开创新的业务和产品。而“不确定性”似乎已成为今天 IT 领域身处环境的共性，也因此，MVP、“开发–测量–学习”循环等理念已成为 IT 创新领域公认的实践，并且围绕精益创业发展出一套完整的创新实践体系，如精益数据分析、精益客户开发、精益交付设计等。

探索和发现有效的价值，并让价值顺畅流动。围绕这两个目标，并遵循精益思想，精益产品开发已经发展成为系统的实践。精益思想对 DevOps 的影响也非常根本，DevOps 三原则就完全遵循了精益思想。

DevOps 的诞生

最初，敏捷和精益社区都还只是关注开发侧的实践和行为，运维并没有成为他们关注的重点。但是，光有系统开发是不够的，开发完的系统必须即时顺利部署，并连续稳定运行才能够实现价值。而传统上，这部分是由运维负责的。

从价值的角度，开发加运维才构成相对完整的 IT 价值链。当然更完整的还应该包含业务，这是后话了，这

还不是早期 DevOps 社区关注的重点。DevOps 诞生之初，解决的问题还是开发和运维之间的问题，这是影响 IT 价值链的最突出问题。

在传统的 IT 组织下，开发团队（Dev）和运维团队（Ops）之间诉求不同——开发团队（尤其是敏捷团队）追求变化，运维团队追求稳定。双方往往存在利益的冲突，比如，精益和敏捷的团队把持续交付作为目标，而运维团队则为了线上的稳定而强调变更控制。部门墙由此建立起来，这当然不利于 IT 价值的最大化。

2009 年，在美国举行的第二届 Velocity 大会上，来自 Flickr 公司的 John Allspaw 和 Pauk Hammond 发表了一个演讲《10+ Deploys Per Day: Dev and Ops Cooperation at Flickr》。在这个演讲中，Allspaw 和 Hammond 以角色扮演的方式，生动地表现了开发与运维之间的各种冲突。演讲中出现很多金句，如“It's not my code, it's your machines!”，这深刻反映了 Dev 和 Ops 关系的现状。接着，他们又展示了如何通过消除开发团队（Dev）和运维团队（Ops）的壁垒，双方通力合作，借助工具和文化的改变让软件的发布和运维变得持续和高效。

这次演讲是 DevOps 发展历程中的标志性事件。它提出了正确的问题——为了更快交付和实现价值，必须弥合开发和运维之间的鸿沟，并给出了解决方案——为了弥合开发和运维之间的鸿沟，需要在文化、工具和实践方面的系列变革。

同一年，比利时独立 IT 咨询师 Patrick Debois 看到这个演讲，受到启发，组织了第一届 DevOpsDays，DevOps 正式登上舞台，DevOps 的理念开始流行，其相关的工具和实践也快速发展。期间以容器化和自动编排调度为代表的云原生技术的出现也极大加速了这一进程。今天，DevOps 已成为企业数字化的核心能力之一，是对 IT 交付和运行的基本要求。

其后，在《凤凰项目》和《DevOps 实践指南》两本书中，Gene Kim 等人总结了 DevOps 实施的三步工作法，它们分别是:

- **流动原则**：聚焦 IT 系统的整体价值流，全局优化，保证价值从左（上游）到右（下游）的快速流动。
- **反馈原则**：创建从左到右的反馈循环，并缩短反馈周期和放大反馈效果。这样，就可以更快的响应和理解内外部客户，并即时获取改进所需要的知识。
- **持续的实验和学习原则**：创建承担风险、持续实验并从错误中学习的文化，在不断的尝试中精进能力，并提高系统的韧性。

Kim 等人认为，这三步工作法是其他一切 DevOps 流程、实践的价值和哲学根基，所有 DevOps 模式都可以从这三个原则派生而来。

稍作探究，就能够觉察，DevOps 三步工作法是精益原则的翻版。更确切的讲，是精益原则在 IT 开发和运行上下文中的具体实例。事实上，DevOps 的基础部分，体现了精益原则的影响和应用。

总结

回顾敏捷、精益和 DevOps 的发展，我们可以得出如下两个结论。

第一，DevOps 是敏捷开发实践的自然发展。敏捷开发的目标是“更早地交付价值，更灵活地应对变化”。敏捷运动始于开发侧，但运维侧如果不做改变，就一定会成为瓶颈，最终敏捷的目标还是无法达成。为了让敏捷实践发挥真正的价值，开发运维的联动就势在必行了。

第二，DevOps 是精益思想应用在 IT 领域的必然结果。精益产品开发的目标是：“顺畅的交付有效价值”，精益思想则要求端到端的系统优化和持续的改进。而开发和运维是系统的两个重要组成部分，缺一不可。DevOps 三原则，正是精益思想在 IT 开发运维领域的具体实例。

最后，从精益思想出发，我们可以看到 DevOps 的必然发展方向，那就是向业务侧延伸。业务是产品开发和运维的源头，完整的价值流必须从源头开始。这不是预测，而是正在发生的事实，大部分 DevOps 的实施都已经将业务侧包含在内，成为 BizDevOps，只不过 DevOps 的称谓已经深入人心，我们也将延续 DevOps 的说法，但缺省情况下，它包含了业务在内。

DevOps 发展的同时，数字化转型也成为了企业界的共识，大部分企业数字化框架都把 DevOps 作为最核心的能力之一，DevOps 的影响范围也不断扩大，成为力求提升数字化能力的企业必然选择。下一节我们将在数字化转型这一背景下，分析 DevOps 要解决的根本问题。

01

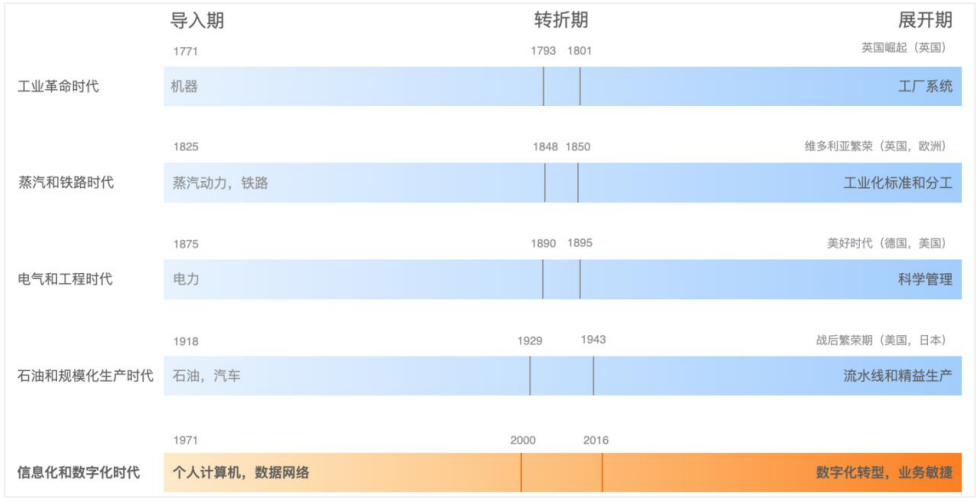
1.2 DevOps 的目标是支持业务敏捷

数字化时代，技术已成为产业发展的核心，技术的交付速度和质量，直接关系业务的发展和创新。大部分企业在规划数字化转型战略时，都会把 DevOps 作为重要一环。

DevOps 的实施必须服务于数字化转型的目标。在实施 DevOps 时，我们首先要理解数字化转型对它的期望和要求。

■ 数字化转型：从信息化到数字化

数字化正在深入改变各个产业。就影响的广度和深度，毫无疑问数字化是一次技术革命。



工业革命以来的 5 次技术革命

《技术革命与金融资本》一书回顾了工业革命以来的 5 次技术革命的历程，总结了其中的共性。每一次技术革命大致都可以分为三个时期：导入期、转折期和展开期。数字化革命也是如此：

01 导入期（1971–2000）：信息化阶段，互联网从无到有

数字化革命的导入期开始于 1971 年。

第一台个人计算机和第一个微处理器都诞生在 1971 年。而此前，1969 年 ARPANET 诞生，成为后来互联网的原型；1968 年 NATO 第一次软件工程会议，标志着软件工程诞生。

微处理器、PC、互联网、软件工程，数字化的基础技术元素就绪。其后 30 年是数字化革命的导入期，互联网从无到有，让全球范围内信息获取、传播和处理都有了若干数量级的提高。

02 转折期（2000–2016）：基础设施就绪，技术和管理范式准备

技术变革不可能一蹴而就，它需要基础设施的全面升级，以及行业认知的深层次转变。而这往往会让技术变革遭遇低潮， 2000 年的互联网泡沫，标志着数字化革命进入了转折期。

转折期影响的是业务发展，但技术发展本身并没停滞。这期间包括云、大数据、AI 和 IoT 等大力发展，奠定了数字化的基础设施。数字化理念也不断深化，并开始在各个行业播种。

今天业界公认转折期已经结束。对中国 2016 是一个标志性的年份，那一年中国的互联网网民超过 7 亿，互联网人口首次过半，再也不可能翻番了。信息产业不可能再单纯依靠互联网人口红利来维持增长。数字革命必须向纵深发展，与各个产业结合。

03 展开期（2016–至今）：数字化阶段，互联网从有到“无”

今天，数字化革命正处于其展开期，它的关键标志是：数字化转型成为各个产业共识。人们关心的不是要不要数字化，而是数字化如何与具体的产业结合。

2016 年是中国产业数字化转型的元年，正如马云在 2016 乌镇的互联网大会上所说，过去 30 年互联网从无到有，未来 30 年互联网从有到“无”，这里的无指的是无处不在，互联网产业和传统产业的界限正在消失。

总结数字化革命已经走过的历程，我们正在从信息化向数字化转变。信息化和数字化有什么不同?

信息化面向的是信息及互联网产业，包括企业内的信息部门。它关注的对象是信息，解决信息互联互通的

问题。它最终目标是提升信息获取、存储、处理和传递的效率。就这一目标而言，信息化做得非常出色。

数字化面向的是全社会的所有产业，关注的对象是具体的业务，深入具体的业务流程；解决的问题是如何精准、实时地响应用户需求。而最终目标是，同时实现最佳的客户体验，和最高的运营效率。



信息化 vs. 数字化

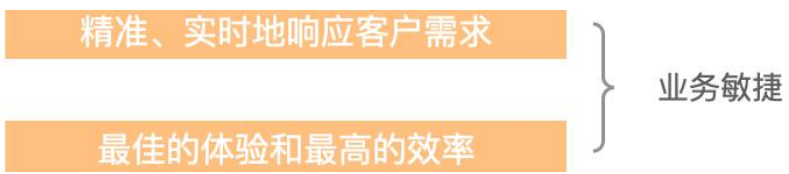
如上图所示，相对信息化，数字化的要求要高许多——广度更广、深度更深、问题更复杂、而目标也更高。这对 IT 技术当然也提出了更高的要求。

■ 从信息化到数字化，DevOps 必须以支持业务敏捷为目标

数字化变革的结果也必须体现在业务上。数字化要做的是赋能业务，带来用户体验和运作效率的同步提升。

在规模化生产时代，体验（尤其是个性化的体验）与运营效率本来就是一对矛盾。我们以牺牲个性化需求体验为代价，带来规模化和标准化的效率。

数字化将有机会为彻底化解体验和效率的矛盾，精准和实时的响应多样化的用户需求，同时还要提高组织的运作效率，实现最佳体验和最高效率的统一。我们把这样的能力称为业务敏捷。

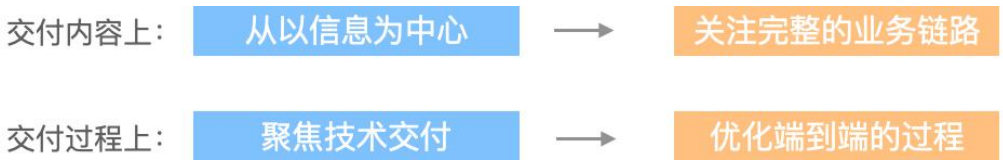


业务敏捷的目标

业务敏捷要求组织全方位的转变，比如构建产业的全量、多维度和实时的数据采集和处理体系；建立符合数字化时代快速反应的组织结构等；实现线上和线下业务的的融合；IT 交付、运行能力的升级等。

数字化转型，是一个系统的变革，DevOps 是数字化转型的一个重要组成部分。数字化时代，更多的业务创新和发展，通过数字化技术才能落地。IT 技术交付和运行的效率，成为决定数字化转型成败的关键，而 DevOps 要解决的问题正在于此。

与聚焦信息本身的消费互联网不同，面向产业的数字化转型，提出了更高的挑战。



数字化时代的业务敏捷挑战

第一：交付内容上，从以信息为中心到关注完整的业务链路。在信息化时代，我们更多关注的还是信息的获取，分享和处理，在商业系统中更多关注的是交易。面对具体的产业，我们必须关注从客户获取、供应链、生产制造、财务支持、市场运营、内部协同到服务交付等完整的链路。

第二：交付过程上，从聚焦技术交付，到优化端到端的过程。我们讲敏捷时，再也不能仅仅是敏捷软件开发，而是要关注从业务、开发、运维在内的全链路的流程，实现端到端的快速响应、交付和稳定的运行。

上面所说的两个变化，是对 IT 技术部门的巨大挑战，需要能力的全面升级，包括组织和协同能力，以及技术和工程能力。下一节介绍阿里巴巴 DevOps 的价值主张时，我们会具体介绍这两个能力。

总结

数字化转型的核心是产业变革。中国有全球最齐全的产业门类，最大的产业规模，是全世界唯一拥有联合国产业分类中所列全部工业门类的国家。同时，我们建立了强大数字化基础设施，有强有力的政策支持和技术人才红利。

改革开放 40 年，我们迅速工业化，实现了规模化制造能力。而，数字化将引领我们进入规模化定制时代，实现体验和效率的同步提升，我们称之为业务敏捷。它比较带动中国产业能力的跃升，从产业大国走向产业强国。

数字化转型是信息技术与产业的结合。需要转型的不仅仅是各个传统的产业，也包含信息产业本身，如互联网公司。DevOps 是数字化转型的重要组成部分，DevOps 的体系和实践也必须服务于数字化转型的需求，这是互联网和传统产业公司的共同挑战和使命。

1.3 阿里巴巴 DevOps 实施的价值主张

数字化转型是对互联网公司和产业内公司的共同挑战。产业公司要应用数字化能力，提升用户体验和运作效率；互联网公司要将数字化能力与具体的产业结合，带来更广更深的创新。共同点是，它们都需要升级 IT 的交付和运行模式，都离不开 DevOps 的能力。

DevOps 实施的定位和目标

下图展示阿里巴巴对 DevOps 实施的理解。



对 DevOps 的理解和价值主张

DevOps 的实施构建在云原生的基础设施之上，并以一站式的 DevOps 工具平台为支撑。通过 DevOps 实施，构建以下两个能力：

第一，持续地顺畅高质量交付有效价值。持续优化协作模式和工程体系，消除业务需求交付过程中的一切阻碍和等待，让交付节奏跟上业务发展的需要，同时保障交付的质量和交付效能的可持续性。

第二，极致弹性和韧性的系统运行。IT 系统必须满足业务运营的要求，具备极致的弹性和韧性。弹性是指它随业务负载自动、实时的扩缩容，以精准的弹性和合理地成本满足业务；韧性指的是确保系统安全、合规和稳定的运行，实现系统运行的连续可用性和安全稳定。

为了建设这两个能力，我们明确提出了 DevOps 实施的价值主张。它们分别是：1）业务驱动的协作模式；2）产品导向的交付模式；3）特性为核心的持续交付；4）应用为核心的运维。接下来将分别作一概述。

■ 阿里巴巴 DevOps 实施的价值主张

01 业务驱动的协作模式

IT 系统的交付是一个协作过程，涉及交付链路上的不同职能，如：业务、产品、开发、测试和运维等；涉及不同功能团队，如前端、后端、中台的不同产品、基础技术组件等。

如何让协作更高效，从而更快地响应和交付业务需求？我们实施 DevOps 的第一个价值主张就是，业务驱动的协作模式。它要求：1）通过业务需求拉通端到端的交付过程，包括：业务、产品、开发、测试、运维等职能的工作；2）通过业务需求对齐各个功能开发的工作，如：前端，后端，中台的交付节奏等。

业务驱动的协作模式寻求系统优化，确保各个局部的工作转化为业务可见的交付效能。我们将在第二部分介绍业务驱动的协作模式相关的实践。

02 产品导向的交付模式

业务需求的满足最终必须落地到产品上才能够交付。产品的交付有两种模式，分别是产品导向的和项目导向的。项目关注短期的交付，而产品关注的是长期的价值。我们主张产品导向的交付模式，是为了长期的效率和业务的价值。

产品导向的交付模式把技术交付团队看成利润中心（而非成本中心），面向产品和业务建设跨功能和相对稳定的产品交付团队，以业务价值和业务响应来衡量和激励产品交付团队。团队面向业务价值，持续地迭代和学习，并积累软件资产、工程和技术资产，提升自己的响应和交付能力。

产品导向的交付模式实践也会在第二部分介绍。

03 特性为核心的持续交付

“业务驱动的协作模式”以及“产品导向的交付模式”，这两者都离不开工程能力的支持，尤其是持续交付工程能力。

我们将持续交付分解为持续部署和持续发布这两个能力。其中，部署（deployment）是技术概念，指的是将软件安装到一个特定的环境；发布（release）是业务概念，指让一个或一组需求对用户可用。

建设持续交付能力，首先要做到两个解耦。1）需求之间的解耦，让各个需求的开发和发布能够独立进行；2）部署和发布之间的解耦，让部署的动作更加灵活，发布能够随需进行。



单应用部署、单需求发布是持续交付的最理想状态

这两个解耦要达成的是：单应用持续变更和单需求持续发布的能力。这是响应业务最敏捷的方式，也是我们对持续交付能力的追求。上图反映了这一状态，一个业务需求经过拆解，对应多个应用的变更，每个应用独立开发、测试和部署，当该需求涉及的所有变更部署完成，这个需求就可以自动发布，或通过特性开关按业务需要发布。

为了做到单应用的持续部署和单需求的按需发布，需要一系列机制、能力和工具体系的支持，如：环境的管理，持续交付流水线的构建、开发联调的手段、质量的保障体系。

我们将在第三部分介绍持续交付的实践。

04 应用为核心的运维体系

运维的目标是在快速响应业务的同时，保障业务系统运行的弹性和韧性。弹性指的是随业务的规模自动和精准伸缩的能力，韧性指的是系统运行的稳定、安全，并保障业务的连续性。

应用视角是连接系统和业务的必然选择，也是连接开发和运维的必然选择。以电商系统为例，购物车、商品详情，下单系统都是独立应用。众多应用构成了淘宝、天猫、支付宝等业务系统。阿里巴巴 DevOps 的开发、交付、运维工作都是围绕应用展开的。每个应用有独立的负责人，对应独立的代码库，有自己独立的资源集、预算，故障定责都是以应用为维度展开的。

阿里巴巴运维体系是构建在应用这个基础单元之上的。基于应用，我们可以进行各种精细化管理，推动完成技术升级，资源成本优化，以及各种稳定性治理工作，实现监、管、控一体化的运维。

我们将在第四部分介绍应用运维的实践。

一站式的 DevOps 工具体系和 DevOps 能力提升模型

DevOps 的实施离不开工具的支持。好的工具能够沉淀原则和方法，贯彻正确的价值主张，让 DevOps 的实施事半功倍。

在研发协作和交付过程，以及 DevOps 的实施过程中，我们面临诸多挑战，如：1）工作流程自动化、标准化问题，如何让业务、产品、技术三种角色高效和有效地协同；2）资产管理问题。如何有效的管理代码、文档、应用、资源等关键资产；3）透明化与数字化问题。如何通过全局的信息透明促进协同，并通过数据洞察为团队改进指明方向。

为了应对这些挑战，阿里逐渐形成了自己特色的 DevOps 实践，并将这些实践落地到一套完整的 DevOps 工具体系中，以适应业务研发的诉求。它有如下特点：

1. **一站**搞定需求、开发、测试、部署、运维的所有诉求。

2. **松管控**、**强卡点**，在工具设计上弱化人为管控，把操作权限下放到一线研发，同时，又提供了全局和特定范围内的卡点能力（如安全检测），保证发布的质量符合要求。

3. **可定制**、**可复用**，**可扩展**，允许开发者根据应用特征和开发习惯定制自己的使用方式以及扩展组件，同时又能方便地复用他人的优秀成果。

我们将在第五部分介绍阿里巴巴的 DevOps 工具体系。

为了给组织 DevOps 实施指明方向，并协助规划清晰的路径，我们设计了 DevOps 能力提升模型，它分为 4 个方面 10 个能力。基于它组织可以评估当前的状态和不足，并规划 DevOps 实施和提升路径。

我们将在最后部分提供 DevOps 的能力提升模型。

业务驱动的协作
和产品导向的交付

02

阿　里　云　云　效

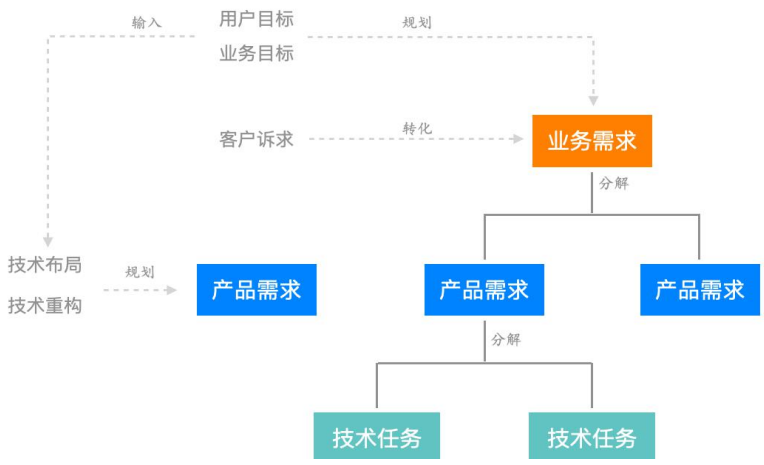
02

2.1 需求的层次结构

解决交付问题的第一步，是搞清交付的是什么。否则，交付对象是模糊的，明确的过程就无从谈起，更无法保证过程效率。

搞清楚交付的对象，具体包括它来源于哪里，为什么服务，有怎样的层次结构以及各个层次承载的价值。接下来，我们将基于最典型的需求层次结构，介绍这部分内容。

■ 典型的需求层次



典型的需求来源及需求层次关系

上图展示了典型的需求层次结构，它包含三个层次：业务需求、产品需求和技术任务。

01 业务需求

需求协作和交付过程，最终必须为业务服务——交付业务价值，并保障业务成功。业务需求来源或转化自原始的用户诉求或业务和产品的初始想法。这些原始的需求经过过滤、归类和分析转化为正式业务需求。

不同特征的组织，业务需求的习惯名称不同。比如：强调用户驱动的可能称之为“用户需求”，对外以产品形式售卖的可能称之为“解决方案”或“产品特性”。不管名称是什么，它们的共同特点是都服务于业务的目标，且最终都要落地为产品功能。在本篇中，我们统称其为“业务需求”，强调这个层次的业务属性。

业务需求承载业务价值，它是系统验收的基本单元，也是运营和发布（Release）的基本单元，需要时可以被独立地发布和运营。它一般由业务人员（如业务运营或业务分析师）负责收集、创建和维护。

02 产品需求

产品需求可以分为两类。第一类拆分自业务需求，直接服务于今天的业务，它一般由业务负责人或产品经理基于业务需求拆分而来；第二类源自产品的规划，它们不直接服务当前业务，而是为未来的业务做基础和提前的准备，典型的包括：产品的基础功能、提前的技术布局、技术重构等。它们通常由产品经理和技术团队创建和维护。

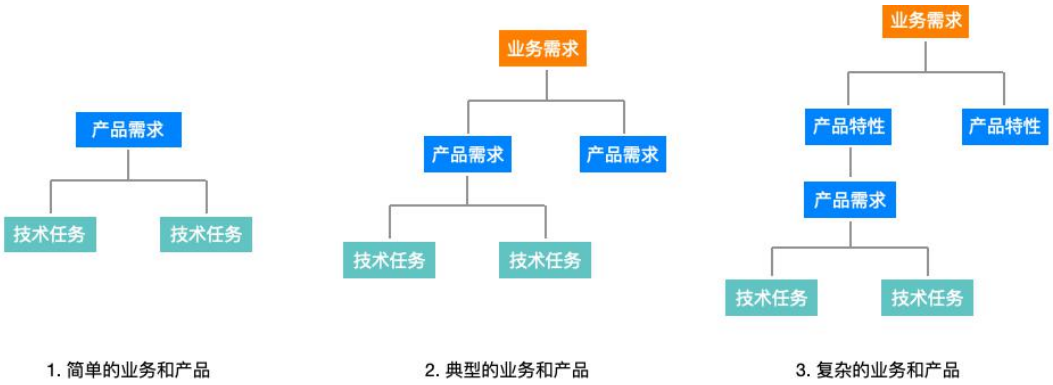
产品需求承载产品的具体功能，是产品集成和测试的基本单元，通常也是系统部署（Deploy）的基本单元。一个业务需求设计多个产品的功能时，它会被拆分到多个产品，对应多个产品需求。例如，业务需求是“在供应链中支持预先锁定库存”。为了实现它，需要供应链计划、库存管理、履约服务、财务结算等多个产品的功能改动，就对应多个产品需求。

03 技术任务

技术任务承载具体的实现工作，它是工作分配（Assign）的基本单元，也是实现的基本单元。技术任务分解自产品需求，它包括不同职能的任务，如前端、后端、算法等。技术任务的拆分通常由技术团队完成。

■ 不同上下文中的层次结构变体

现实中，业务和产品复杂度不同，其层次结构也不同。下图是三个不同变体。从左到右分别适用于简单、典型和复杂的业务和产品。



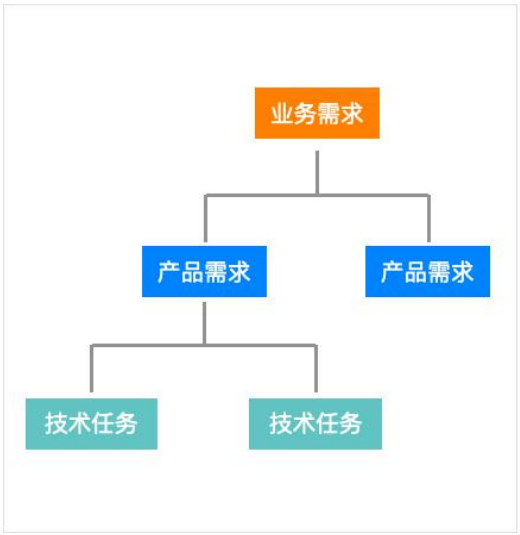
基于不同的业务和产品复杂度的需求层次结构调整

图中的第一种模式适用于简单的业务。这时，业务与产品一一对应，业务需求与产品需求也合并为一层。这一模式下，原始需求直接转化为产品需求，产品需求分解为技术任务。这一模式适用于简单的互联网业务、企业应用或工具产品，业务在初创时都适合这一模式。

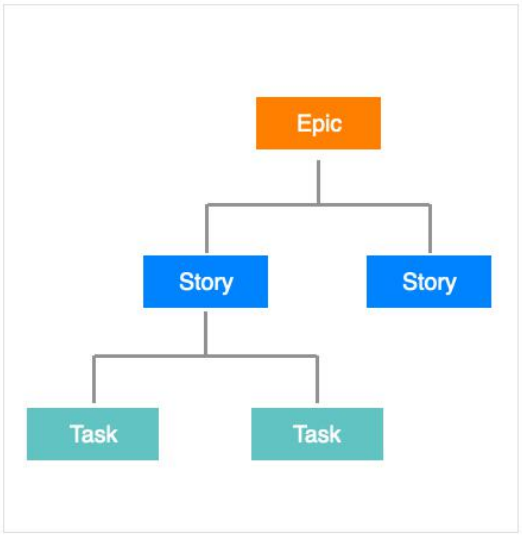
对于复杂的产品和业务，如果产品需求只有一个层次，可能会让产品需求过大。这时，可以将产品需求分解为两个层次——产品特性和产品需求两层，也就是图中的第三种模式，它适用于电信产品、基础技术产品、复杂的企业应用等复杂的业务领域。不过，对于大部分场景，第二种典型中的三个层次就已经足够。

■ 赋予每个层次明确的意义和目的

不同方法体系，其需求层次结构的定义的依据不同，结果也不同，如：Scrum 中常用 Epic、Story 和 Task 来描述需求的层次关系。这一层次结构的划分依据是规模。其中，Story（用户故事）是从用户角度对需求的描述，它要求能够在一个迭代完成；Epic（史诗）是巨型故事或故事集，需要被进一步分解为 Story；Task（任务）是对 Story 的进一步拆分，要求能跟踪和更新日常进展，通常可以由单个人完成，工作量不超过 20 工作人时（Person Hours）。



以承载的功能来区分需求的层次，带来落地过程的明确性



Scrum 以需求大小来区分需求的层次，带来普适性

需求类型的不同命名方式

作为通用方法，Scrum 追求普适性，其推荐的需求层次刻意回避了业务属性。但是，它在提高普适性的同时，让协作过程的业务意义模糊，导致无法定义精准和有效的协作流程。

我们认为，只有明确需求层次以及每个层次承载的价值，我们才能够定义有意义的协作过程，过程相关的人员的职责和活动，并判断这一过程是否与所承载的价值匹配。因此，我们在设计需求的层次结构时，要求明确每个层次的意义和价值。

总结

以上，我们分析了需求的来源、需求的层次结构、以及每个层次所承载的价值。它是一个标准参考模型，本身具备较好的普适性。基于业务的特征不同，也可以加以定制。比如：因业务的复杂性不同，而增减层次；或者业务交付方式不同，而将业务需求改成其他名称。

清晰定义需求的层次结构，明确各个层次的价值。基于它们，我们就可以定义协作过程，实现并交付这些价值，保障协作和交付的效率和效果。下一节我们会介绍业务驱动的协作。

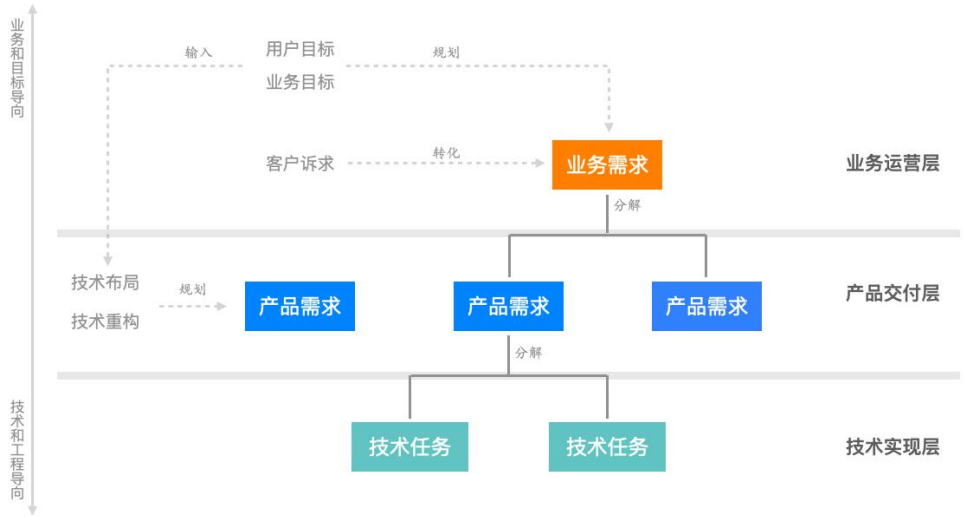
02

2.2 业务驱动的协作

明确需求层次以及每个层次承载的价值之后，接下来要做的是定义每个层次的协作过程，最终服务于“顺畅高质量地交付业务需求”这一目标。如何组织各个层次的协作，来达成这一最终目标？我们将从两个方面分别讲述：1）协作的层次以及各层次的定位和目标；2）以业务为源头驱动整体协作。

■ 协作的层次以及各层次的定位和目标

如下图作所示，基于需求的层次关系，我们将协作自上而下分成三个层次。分别是业务运营层、产品交付层和技术实现层。这三个层次中，越向上越是业务和目标导向的，越向下越是技术和工程导向的。



业务需求交付的三个层次

接下来，我们将分别讲述这三个层次的协作要点和目标。

01 业务运营层

业务运营层处于最上层，是驱动整体协作的源头核心。我们要建立的是业务驱动的协作模式——通过业务拉通和协调整体产品交付和技术工作。

业务运营层由业务人员负责，他们管理的核心对象是业务需求，目标是业务需求交付的效率和效果。

在业务运营层，首先要管理好业务需求的端到端的交付流程，包括：创建或接收业务需求；规划业务需求的发布和版本；管理跟踪业务需求交付；验收和发布业务需求；反馈和改进业务的设计和规划等。

向上，业务需求或者是向目标对齐，基于目标创建和规划业务需求；或者是响应客户诉求，将客户的诉求转化为业务需求。业务需求发布后，要能够基于其对接的目标和客户诉求形成反馈闭环。

向下，业务需求会拆分为产品需求，并分配至对应的产品交付团队。同时，还要协调相关产品需求的交付过程，对齐它们的进度，及时发现瓶颈问题，加速业务需求的实现。

为了保障业务的交付效率和有效性，业务运营层需要三个核心能力。

- 业务规划能力：业务规划指的是基于业务目标、客户诉求、市场情况，结合资源约束等，做出取舍，决定做什么不做什么以及何时做；
- 交付协调能力：基于业务需求拆分和分配产品需求，并协调产品需求的交付过程，发现和处理问题和瓶颈，最终保证业务需求的顺畅交付；
- 业务反馈调整能力：在需求发布后即时收集用户和市场的定性及定量的反馈，并基于反馈调整后续的规划和业务演进，保证业务进化和迭代的有效性。

这三个能力形成完整的规划、交付、调整闭环。加速这一闭环并提高它的有效性，将帮助组织提升业务交付的效率和效果，更好地满足客户的诉求和达成业务目标，这也是业务运营层的长期目标。

02 产品交付层

产品交付层由产品交付团队负责，产品交付团队包括产品经理和开发团队。他们关注和管理的核心对象是产品需求，目标是提高产品需求的持续交付能力，包括速度、可预测性和质量，最终服务业务需求的交付。

在产品交付层，首先要管理好产品需求的端到端的交付流程，包括：接受拆分自业务需求的产品需求，或

者规划产品需求；对产品需求进行排期或做出迭代计划；跟踪和管理产品需求交付过程；反馈和改进产品的交付效率。产品交付层，向上对接业务交付，向下对接技术和工程活动。

向上，产品交付要向业务交付对齐。首先，产品需求的内容、计划及进度向进行中的业务需求对齐，保障业务的交付速度和质量。其次，为了保障长期效率，产品还要面向未来规划和演进，如：面向未来业务发展的提前产品布局和准备；或者为了长期效率进行和消除技术债务的产品重构和沉淀。

向下，产品需求会被拆分为技术任务，由不同的人负责。产品交付层协调技术任务的进度，及时发现瓶颈和问题，确保产品需求的顺畅和高质量交付。

为了提高产品需求的交付效率，业务运营层需要建设三个核心能力：

- **规划和演进能力**：产品需求要响应即时的业务需求，还要关注长期的效率。
- **排期和交付能力**：决定需求的交付顺序和节奏，并管理交付过程，保证产品需求的快速交付。
- **交付效能改进能力**：改善团队的协作，产品的架构和设计，工程和技术实践等。

这三个能力构成从规划、交付到改进的闭环，确保产品交付的持续效率。

03 技术实现层

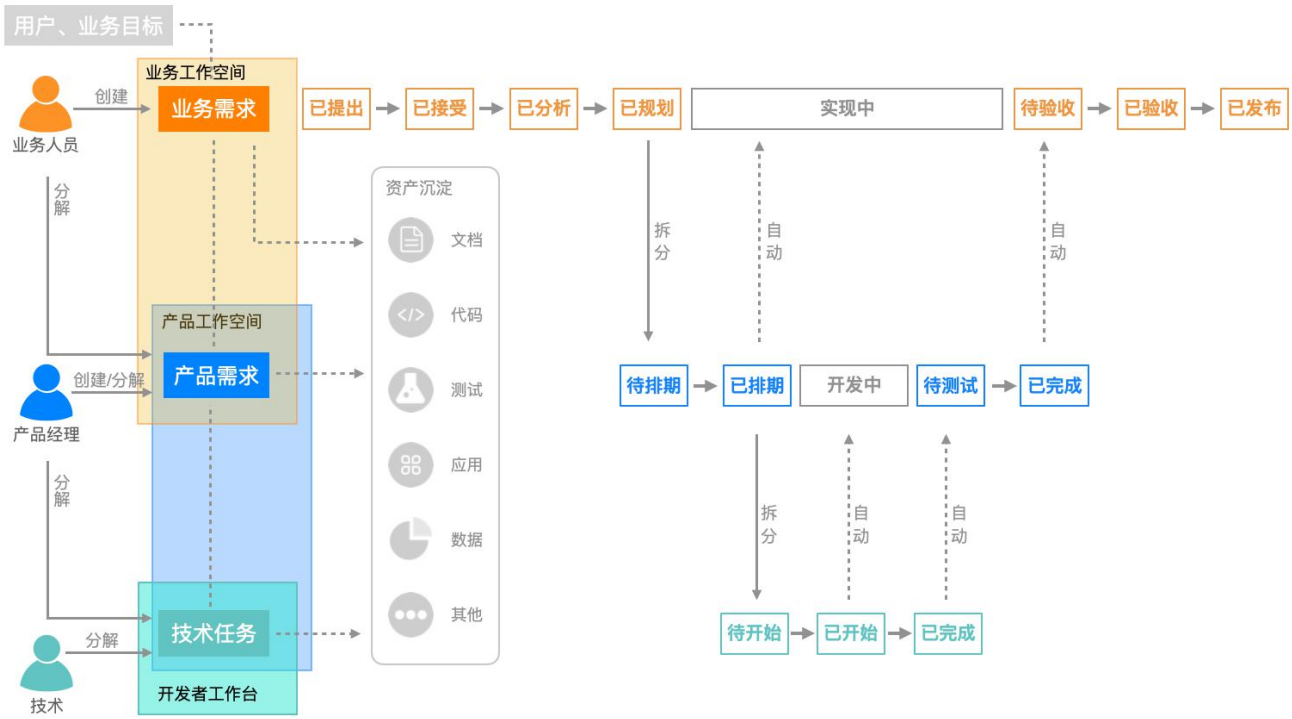
技术实现层由技术人员负责，包括开发、测试等。技术实现层关注的核心对象是系统（应用）的变更，他们基于产品需求拆分任务或创建变更，并完成开发、测试、联调和变更发布。

在技术实现层，首先应该为开发者提供良好的工具、环境和机制，让他们可以高效的完成工作，并保障交付的质量。以此为基础，技术实现层的工作要向产品交付对齐，对齐各自的工作，尽快完成和交付产品需求。

综上所述，基于需求的结构，我们将需求的交付工作分为三个层次。每个层次都有负责角色，他们要专注完成自己的工作，同时更要能够相互协同，实现业务需求的顺畅和高质量交付。其中，业务既是驱动这一协作过程的源头，也是协作的目标。下一节我们将关注怎样让这一协作过程更高效和有效。

■ 以业务为源头驱动整体协作

上面我们分析了协作的层次。每个层次都有自己的目标，有专门的负责角色，他们的管理对象以及工作任务不同，为了让他们能更高效的工作，他们应该有自己专属的协作空间。



业务驱动的分层协作

上图展示了业务驱动协作方式，它按需求的层次将协作也分为三层，各个层相互连接构成整体。接下来我们将从三个方面加以阐述，分别是：1）为每个层次创建专属工作空间；2）连接各个层次，实现高效协同；3）以需求连接和结构化过程资产。

01 为每个层次的创建专属工作空间

	负责角色	目标	管理对象	工作职责	工作空间
业务层	业务人员	响应和满足业务诉求 达成业务目标	以业务需求为核心，同时关注其下属产品需求	• 创建或接收业务需求 • 规划业务需求的交付计划或版本 • 拆分产品需求 • 管理跟踪业务需求交付过程 • 验收和发布业务需求 • 反馈和改进业务的设计和规划	业务工作空间

	负责角色	目标	管理对象	工作职责	工作空间
产品层	产品交付团队（含产品经理及技术团队）	规划和交付产品需求满足业务交付的短期和长期需求	以产品需求为核心，同时关注其下属的技术任务	• 接受或创建产品需求 • 管理和跟踪产品需求的交付过程 • 对产品需求进行排期或做出迭代计划 • 拆分产品需求的技术任务 • 反馈和改进产品的交付过程	产品交付空间
技术层	技术开发	完成技术任务并高质量和即时的交付需求	系统变更及其它个人任务	• 接收、管理工作任务 • 开发、联调和验证 • 集成需求并变更系统	开发者工作台
总体目标	拉通端到端的交付过程，确保过程质量，对齐各个层次的工作，最终实现业务需求的顺畅和高质量交付。				

针对不同角色的工作职责的工作空间

上表列举了各个层次特征。每个层次的负责人（或团队）不同，他们的目标、管理对象和工作职责不同。为了提高每个层次工作的效率和质量，他们应该有自己专属工作空间，具体包括：业务工作空间、产品交付空间和开发者工作台。这些空间可以在不同的数字化工具中配置。不同的研发管理工具的实现方式不同，它可能是一个独立的项目；也可以是逻辑的视图。

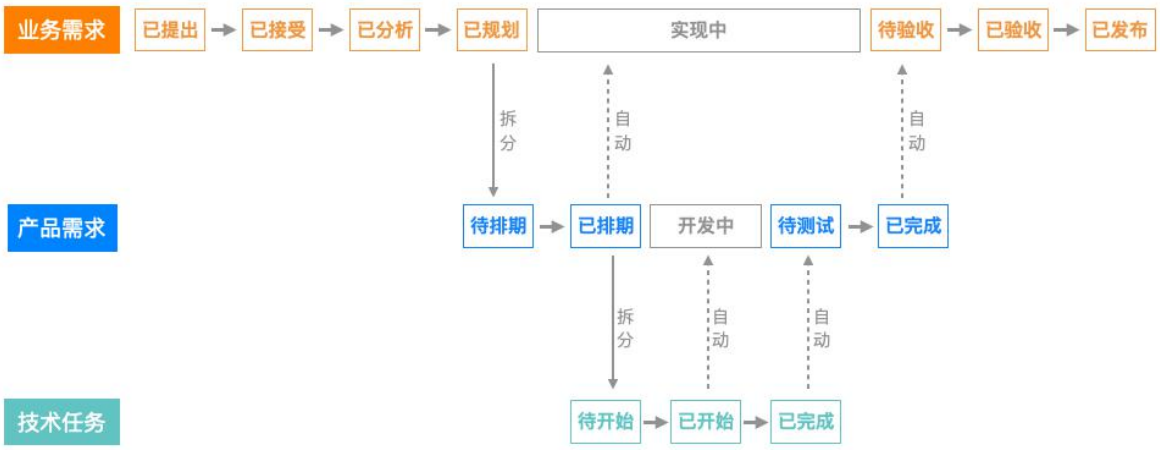
在数字化工具中，区分各个空间的是：1）支持不同的需求类型；2）不同需求类型有不同工作流（需求从开始到完成所要经历的阶段）；3）各个空间应用不同的功能服务（如版本规划、迭代排期等）。

业务工作空间支持的需求类型是业务需求和产品需求，其中以业务需求为主，并基于业务需求拆分产品需求。在业务工作空间中清晰定义业务需求的工作流（参见图二示例）。业务工作空间中应用的典型功能服务有需求拆分、业务规划和业务复盘等。

产品交付空间支持的需求类型是产品需求和工作任务，其中以产品为主，并支持从产品需求拆分任务；在产品交付空间中要清晰定义业务需求的工作流（参见图二示例）。产品交付空间中应用的典型功能服务有迭代排期，任务拆分、进度统计等。

02 连接各个层次需求和需求流程，实现高效协同

以上各个层次的工作空间，可以专注于工作。同时，各个层次又是互相连接，构成有机整体的，下图反映其中的连接关系。下面将基于最典型的场景介绍各个层次的协作流程，以及它们之间的连接。



各个层次的连接

业务人员工作在业务工作空间，管理需求的端到端的交付过程。他们创建、接受、分析和规划需求。规划好的业务需求会被拆分为产品需求，这通常由业务人员与各产品的产品经理共同完成。拆分的产品需求，被分配到对应的产品交付团队。

产品交付团队工作在产品交付空间，他们接受拆分的产品需求或者自主创建需求、进行产品设计和需求澄清、完成需求排期、拆分技术任务、开发并集成需求、完成部署。

技术开发人员工作在个人工作空间，他们接受或拆分技术任务、创建变更、进行系统开发和集成联调、提交代码、完成持续集成验证，发布变更。

业务工作空间和产品交付空间的联动关系是：1）业务需求拆分的产品需求会被分配到对应的产品交付空间，此时产品需求与业务需求维持归属关系；2）当下属的产品需求完成排期后，对应的业务需求会进入开发中状态；3）当业务需求所有下属产品需求完成后，业务需求进入“待验收”状态。

产品交付空间和个人工作台的联动关系是：1）产品需求会分配到个人，需求拆分的任务也会分配到人。这些需求和任务都会自动出现在个人工作空间；2）个人对需求和任务的更新会自动体现到相应的产品和业务空间；3）当下属任务开始开发时，对应的产品需求会进入开发中状态；4）当产品需求所有下属的任务开发完成后，

产品需求进入“待测试”状态。

这三个层次相互解耦，不同角色可以专注于各自的职责和目标，高效完成自己的工作。它们又相互连接，构成一个有机整体。通过这个有机的整体，我们以业务为驱动拉通端到端的交付过程，确保过程质量，对齐各个层次的工作，最终实现业务需求的顺畅和高质量交付。

03 以需求连接和结构化过程资产

需求交付是知识工作，过程中有大量的产出物，如：讨论的记录、临时或正式文档、UE/UI 的设计、代码、测试用例、应用变更记录等，这些都是重要的知识资产。

如何管理好这些资产的创建过程，并结构化的组织和索引它们？这是一个重要的挑战。它事关沟通和协作的效率，更关系到组织的智力资产的价值。



以层次化的需求为线索，连接过程产出物。

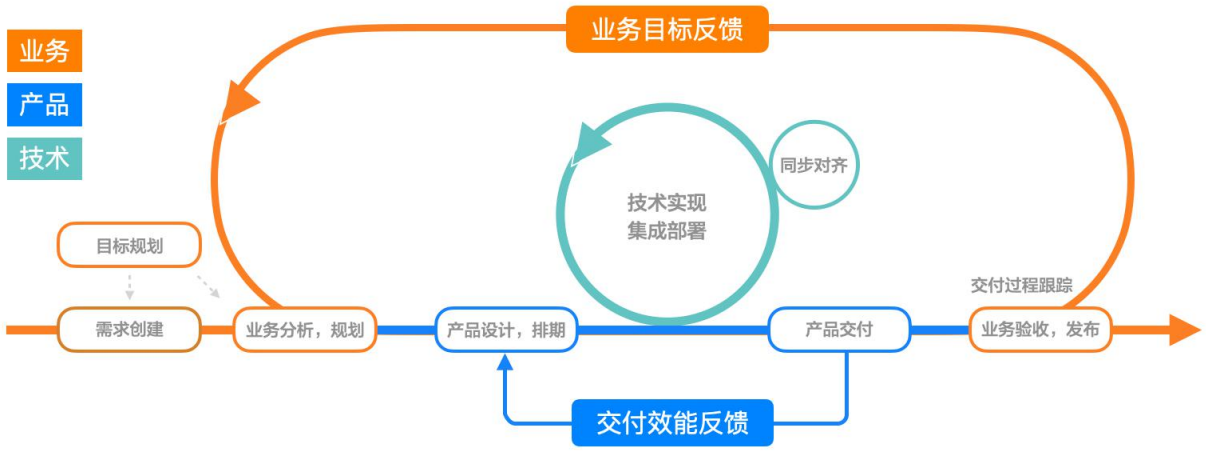
确保信息沟通的效率和知识资产的有效组织和沉淀。

交付过程中产出物与需求的关联

如上图所示，资产应该与研发过程中的价值单元相关联，所谓价值单元指的是前文介绍的需求的层次结构。在需求交付的价值流中，我们创建并与相关资产关联。如：业务需求创建时关联原始业务诉求的描述，业务需求的分析则关联相关的业务分析文档；产品设计过程中关联相关文档和测试用例；技术任务交付过程中关联相关的应用、变更和代码。

这一方面让研发的数字资产直接与客户价值打通，另一方面让它与价值的创造活动关联。打通这些信息间的连接，避免了信息孤岛，不仅提高沟通和协作效率，更让资产沉淀有序和结构化，便于信息的检索和使用，充分发挥研发过程数字资产的价值。

总结：整体的交付流程和反馈闭环

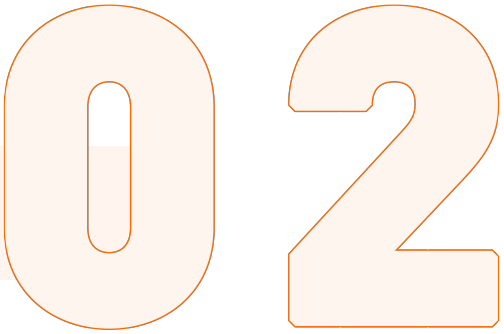


整体交付流程和反馈闭环

以上我们定义并介绍了业务驱动的协作过程。如上图所示，这个过程包含业务、产品和技术三个方面。

第一是业务交付和反馈闭环（图中橙色部分），它的核心关注点是业务的快速响应和交付，以及业务目标的达成；第二个是产品交付和反馈闭环（图中蓝色部分），它的核心关注点是产品需求的快速交付，产品的演进，以及产品交付效能的持续提升；第三个是技术和工程实践（图中绿色部分），它的核心关注点是基础设置和技术及工程能力的持续改进，从而奠定产品和业务效能的基础。

这一交付过程既分层解耦又有机连接，帮助我们打通交付的各个环节和层次，它让整个组织协调一致，发挥最大的效率，实现对业务的精准响应和快速迭代进化。



2.3 产品导向的交付

IT 的发展的历程，也是一部技术和业务相互接近和融合的历程，IT 的交付模式也需要随之进化。

最初 IT 是专业公司及专业人员的工作，IT 开发了什么，业务就用什么。再后来，IT 公司向业务靠拢，更多与客户互动和共创，很多业务公司也有了自己的 IT 开发部门。但，即使在组织内部，IT 也是成本中心，扮演乙方的角色。此时，IT 与业务是分离的，大部分 IT 交付是以项目的形式进行的。

项目的内核是：在特定时间内，以相对确定的预算和人力，交付预先规定的内容。项目追求确定性，注重计划和计划执行。在 IT 与业务分离的时代，这样划清边界，对双方都有好处。相应的，组织也可以按项目来预先规划、分配和执行 IT 预算，而这通常是以年度或至少以季度为单位的。

进入数字化时代，IT 成为业务的一部分。以项目为单位来规划和交付，越来越不能满足即时响应业务的需要。项目面向一次性交付的特点，也不利于资产（特别是软件资产）的长期积累和优化，不利于工程能力及基础设施的持续优化。

	项目导向的交付模式	产品导向的交付模式
基本假设	追求工程的确定性。面向单次项目交付，注重计划和按计划的执行	承认不确定性。面向长期业务价值，注重迭代演进和能力的积累
成功标准定义	把 IT 当着成本中心。衡量能否在确定的预算和时间内交付确定的内容	把 IT 当成利润中心，衡量业务价值的创造能力和业务响应速度及效率
工作分配模式	把人作为资源分配到工作中，需要较大批量的计划和安排，趋向于批量式的交付	把工作分配给团队，持续的响应、紧密协作、持续交付
团队组织方式	临时团队，为事情（项目）组建团队	跨职能和功能的长期和相对稳定的团队
资产沉淀和优化	项目交付物，项目管理流程的过程资产	软件资产，工程和技术能力，基础设施、高效能的交付团队

上图总结了项目导向的交付模式与产品导向的交付模式的区别。在 DevOp 是实施中，我们主张用尽量采取产品导向的交付模式。

在产品导向的交付模式下，组织应该把技术交付团队看成利润中心而非成本中心，面向产品和业务建设跨功能和相对稳定的产品交付团队，以业务价值和业务响应来衡量和激励产品交付团队。而团队则面向业务价值，持续地迭代和学习，并积累软件资产、工程和技术资产，提升自己的响应和交付能力。

下面我们将介绍产品导向的交付模式的基本要求。

关注长期效率和业务价值

产品导向的交付模式下，产品为长期的业务结果负责。相应的，衡量产品交付团队的标准应该支持业务的效果和效率。我们把产品开发的目标分为 3 个大类：

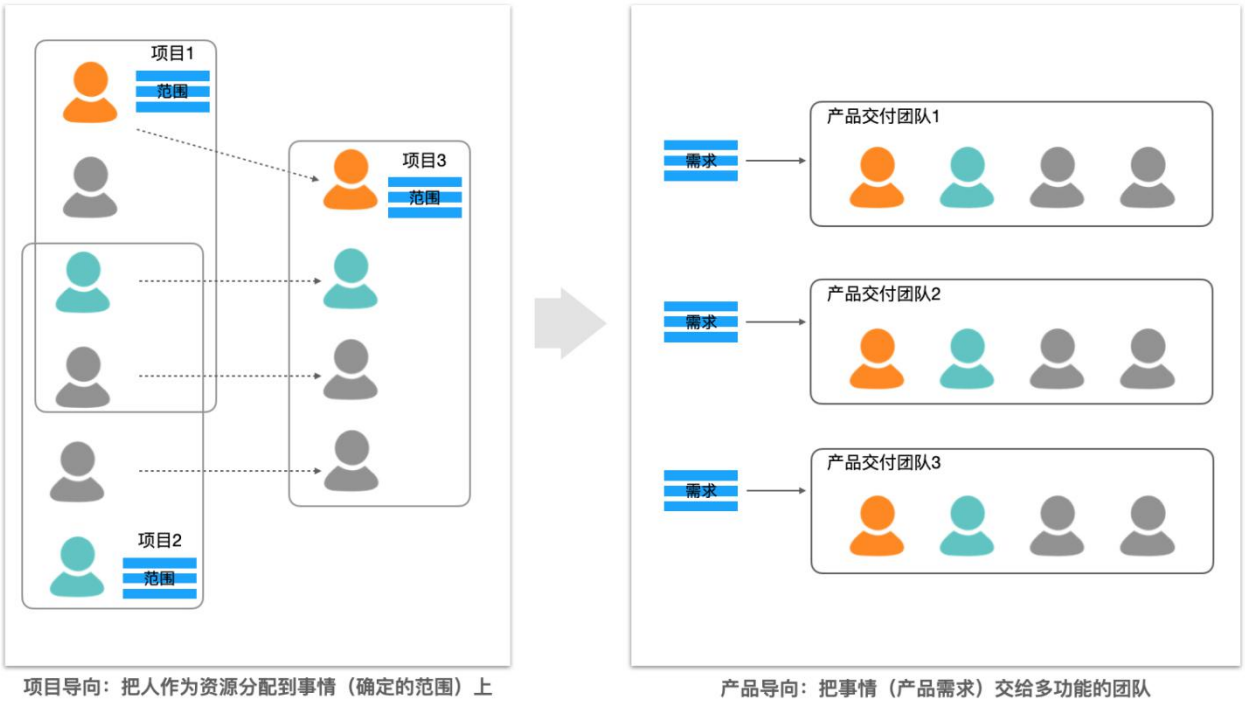
- 第一：直接服务于当下的业务。这一类目标最容易理解，技术的价值体现为直接的业务成果。此时，产品需求通常来自具体的客户诉求或业务的规划。以下是一些典型的例子：1）构建新的客户服务交付流程，以提高用户交付的效率；2）支持内容敏感的缓存和路由模式，以提高视频观看体验并减低带宽消耗；
- 第二：提前的产品布局。这一类目标并不直接服务当下的业务。它是基于对未来业务和技术方向的判断，做出的提前布局。比如：基于未来业务增长的预测，提前升级分布式架构；为 IoT 战略而做的预研和技术准备。这类目标也是业务驱动的，不过驱动它的是未来的业务。
- 第三：改进交付的效率和效能。除了产品交付本身，团队还要为交付效能负责，包括对业务需求的响应速度、交付的质量、创新的有效性等。比如，通过平台化的能力或业务中台的建设，以更小的工作量和更短的时间交付业务需求。比如通过工程能力的建设，建立系统集成和质量保障体系。

与项目交付不同，衡量一个产品交付团队不再是完成一次性的交付，而是长期的业务价值和效率。它体现为以上三个目标的综合。在设定产品交付团队的目标时，我们也需要按这三个类别来组织。比如团队的 OKR（objective key results 一种目标设定和管理的方法）应该从这三个方面来展开和设定。

把工作分给相对稳定的多功能团队

在人员管理上，项目管理会先规划预算、确定范围、做出计划，再按计划把人分配到项目中。项目结束时，人员被释放，准备进入下一个项目。这一模式下，人被当成可替换的资源。对于确定性高且响应速度要求低的业务，项目的人力配置方式有它的优势，尤其是加大了人员的使用效率。

数字化时代业务的不确定性越来越高，对 IT 的响应速度要求却越来越高，对产品的持续健康演进要求也越来越高。项目管理中的团队组建模式带来以下问题：1）由于每次都要先确定内容，再做出计划、组建团队，必然会延长需求的响应和交付时间，满足不了即时响应业务的需要；2）由于项目团是临时组建的，只对短期的交付负责，很难保证长期的业务结果，同时业务发对产品的长期演进和持续的交付效能负责



从把人分到事情上到把事情交给团队

在 DevOps 实施中，我们倡导产品导向的交付。就团队组织而言，它的核心变化是：从把“人分配到事情上”转化为“把事情交给人”。更详细的描述是：项目导向的交付模式，把人作为可替换的资源，分配到预先确定范围的事情上；产品导向的交付模式，把事情（也就是动态产生的产品需求）分配给跨职能的交付团队。它的核心变化有两点：

第一，需求的处理方式。项目交付会预先和批量的确定项目的范围，不能即时灵活的响应业务的变化。产

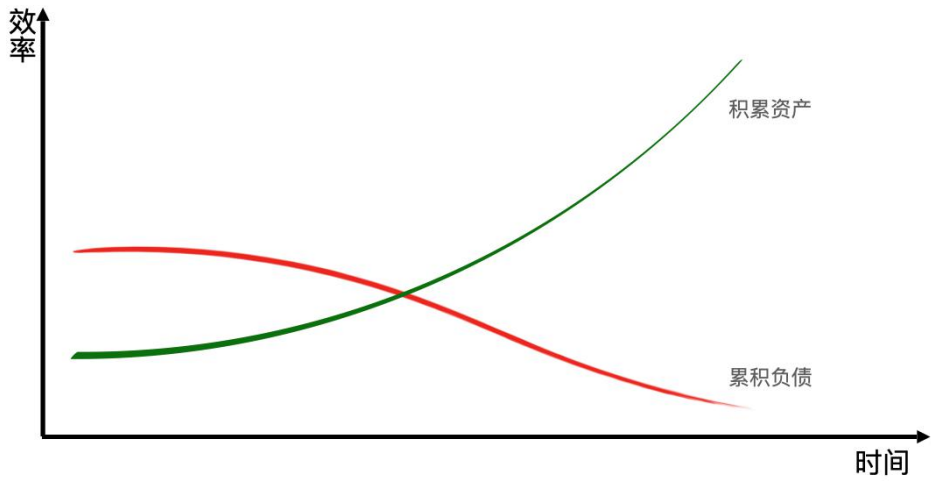
品导向的交付，应该持续的响应业务的输入，迭代的进行业务规划，并通过不同优化团队的需求交付价值流，做到持续和快速的需求交付。

第二， 团队的组织方式。产品交付团队是相对稳定的，对长期的产品演进、交付效能、业务贡献负责。为此，这个团队应该是 1）跨功能的，包含产品、开发、测试等角色。只有这样才能直接面向业务，完成产品需求的加氟，才能集体改进交付效能；2）被充分授权的，比如：在满足外部业务需要的情况下，被授权演进自己的协作方式、技术和工程实践。只有这样才能够；3）相对长期稳定的。长期的团队才能对长期的结果负责。

产品导向的交付模式中，产品交付团队是组织的核心，它是跨功能的、被充分授权，并相对稳定的。长期效能是产品交付团队的核心关注，而它很大程度依赖于上产品的演进和工程能力的建设，这是下一节的内容。

■ 持续积累高质量软件资产和团队工程技术能力

理想情况下，随着产品开发的演进，产品提供的基础功能越来越完备，更多的原子功能被封装为软件的接口和服务，成为可复用的组件。这时，满足新的业务需求的成本应该更低。然而，很多时候事实却正好相反，随着技术开发的演进，软件的复杂度越来越高，可扩展和可维护性越来越差，团队对业务的响应越来越慢。



减少负责，积累高质量资产

上图反映了效率随时间变化的两种趋势。决定效率是持续提升还是下降，是我们在开发过程究竟是在积累资产，还是累积债务。

资产指的是今天积累的，会为未来收益的东西。在 IT 产品交付中，良好的资产主要包含：软件资产、工程资产和组织资产，以下是常见的例子：

- 丰富和设计良好的原子服务；
- 持续逼近业务本质，且具备持续演进能力的领域模型；
- 与概念领域模型一致，且维护良好且有测试守护的设计及编码；
- 良好的工程实践和质量保障体系，如：持续交付流水线以及基于持续交付流水线的质量保障体；
- 良好的团队组织方式和协作机制及实践；

而债务指的是，今天造成的，将来要位置付出的东西。在 IT 产品交付中，以下是常见的债务：

- 糟糕的接口和服务设计，导致重用过程中的麻烦；
- 软件复杂度增加、设计欠佳，造成扩展和维护困难；
- 缺乏有效的测试守护的代码，导致回归测试工作量不断增加，软件质量劣化；
- 组织规模膨胀，协同复杂度增加；
- 缺少有效的工程实践，导致系统的集成、验证成本变大，质量却难以保证；

对待资产和债务的态度，是区分卓越和糟糕的重要因素。一个卓越的产品交付团队，应该不断的积累资产，并消除累积的债务。当然，这里说的并非 100%的避免债务。在特殊情况下，如响应紧急业务诉求是，引入再无在所难免。但，那必须是有意识的权衡，并且将来会有计划的去消除债务。

只有不断的积累资产，以及避免和消除债务，才能让交付效率持续提升，提高产品对业务的响应能力，并提升产品的业务价值。

■ 业务驱动的协作和产品导向的交付相互契合

适配数字化时拥抱不确定性和持续创新的要求，IT 的协作模式和交付模式也需求要演进，我们提倡并践行业务驱动的协作模式和产品导向的交付模式，这两者是天然契合并可以无缝连接的。

协作模式应该是业务驱动的：

- IT 的工作一切源自业务，业务驱动各个职能和功能的协作和快速交付；
- IT 的工作必须回到业务，形成有效的学习闭环，支持业务的创新。

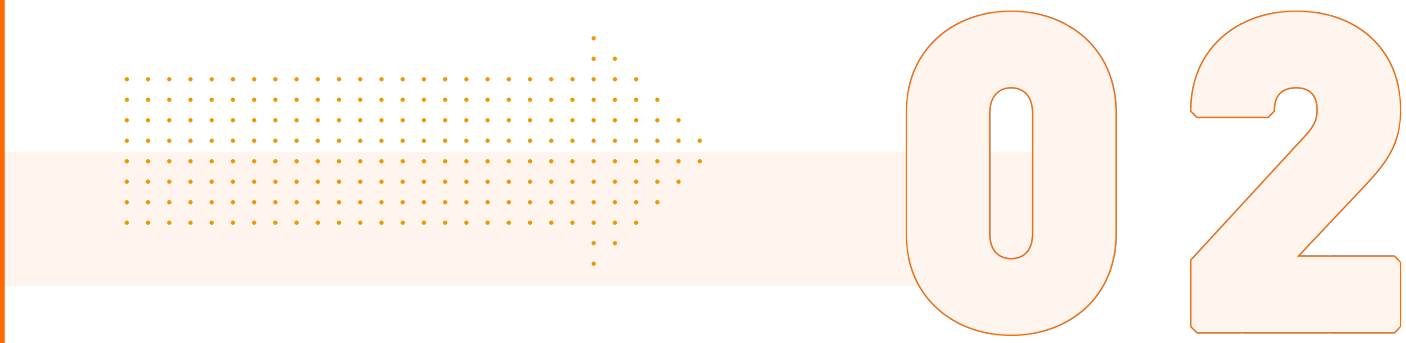
交付模式应该是以产品导向的：

- 从把 IT 看作成本中心，关注资源使用效率，到把 IT 看作利润中心，关注长期和整体的效益；
- 从面向短期的一次批量式性交付，到长期主义的持续交付和持续产品演进；
- 从把人看作可替换资源，为项目组建临时团队，到持续改进的相对固定的多功能团队
- 从主要关注项目管理的流程资产，到关注架构设计、软件代码、工程能力、团队效率在内全方位的软件、工程和组织资产；

在上一章介绍协作模式时，业务层分解出的产品需求，直接交给产品交付团队，产品交付团队快速的完成产品需求并持续交付，这两个层次相互衔接，完成业务的快速响应、交付和反馈闭环，赋能数字化时代的高效创新。

总结

业务驱动和产品导向是适应数字化时代要求的协作和交付方式，是我们对 DevOps 实施的核心价值主张。同时，它们的有效实施离不开工程实践和能力的支撑，下一章我们将讨论 DevOps 的另一核心要素——持续交付的工程能力。



2.4 规模化实施路径

今天，大部分企业不再怀疑 DevOps 实施的必要性，并开始行动。问题是，小范围实施效果不错，但一扩大范围就会遇到各种问题，无法达成预期。DevOps 的规模化实施存在诸多陷阱。

要想成功规模化，首先要理解规模化的真实含义，其次是选择有效的规模化路径。接下来我们会从这两个方面为规模化实施提供策略性的指导。

规模化的真实含义

不同人讲规模化时，背后的意思可能千差万别。比如：有人关注人数规模；有人更关注深度；而另一些人则关注规模化方法框架的推广。

为了有效的规模化，首先必须对规模化达成一致的理解。而要达成一致的理解，就必须回到规模化的目标这一本源上来，也就是为什么要规模化。我们从业务出发，认为规模化的目标有两层：

第一是“系统”——延伸至端到端的系统。DevOps 的启动可能会从某个或某几个环节开始，比如从部署环节开始。但，我们必须逐步延展实施范围，打通从业务、研发到运维的整个链路。只有这样，才能带来系统性的改变，实现快速的交付、稳定的运行和即时有效的闭环。

第二是“普惠”——让更多的业务或同一业务各个组成部分都受惠。DevOps 是为了服务企业的数字化，它涉及企业不同的独立业务，或同一业务的不同职能，如：一个业务线中的客户管理、内部运营、生产制造、服务交付等。DevOps 的实施要让更多的业务部门从中受惠，这样才能支持组织的整体数字化转型。

综合以上两点，规模实施的目标就是：让更多业务，更系统性地受惠。DevOps 的规模化实施就是要寻找平滑和可持续的路径来实现这一目标。

设计和选择规模化的路径的原则

DevOps 的规模化实施是一个转型过程。它涉及到技术、流程和组织的变革，甚至是权力和利益的再分配。

转型从来不易，“以复杂对规模”是 DevOps 转型最常见的误区。

规模带来复杂性，直觉上复杂的状况需要复杂的解决方案。于是乎，就出现了各种复杂的规模化框架，比如各种复杂的规模化敏捷框架。然而，照搬规模化框架很少带来好结果，反而带来各种问题。

首先，任何方案的实施都要以对现状的理解为基础，比如说现有的组织结构、对新方法的接受度、具体的业务特征、技术成熟度等。脱离具体的上下文，照搬“普适”的框架，会带来巨大的适配成本。前期花费大量精力在组织和流程上，却不直接改善交付效能，特别是业务可见的效能。这非常容易带来怀疑和抵制，让实施效果大打折扣，甚至半途而废。

其次，为了普适，各种规模化框架都会定义统一组织结构、角色职责、流程机制，引入复杂机制，却无法适配现实的多样性。结果是，老的模式无法被替代，又叠加一套新的机制，复杂度不减反增。复杂的组织结构、流程机制以及更多的角色，这些会加大实施难度，降低实施效果，即使取得暂时的成果，也很难维持。

照搬规模化的框架不行，那该如何做呢？我们必须回到规模化实施的目标——“系统”和“普惠”上，并在具体的上下文中，选择自己的规模化路径。为此我们提出了 DevOps 规模化三原则：

原则一：业务驱动原则。以业务的视角衡量 DevOps 的实施效果，并驱动 DevOps 的实施。

业务驱动是 DevOps 规模化的基本原则。判断方案的好坏，必须要有标准，但它不应该是个人喜好或各个部门的利益。只有业务的成功能成为大家的共同标准。

以业务视角来衡量和选择方案，才能凝聚各个角色的共识；以业务目标驱动 DevOps 实施，更能拉通各职能的协作，保障实施的系统性；以业务为单元来实施 DevOps，也能让实施的边界和节奏更加清晰，尽早取得可感知的业务结果。

原则二：去规模化原则。面对复杂的组织，先“去规模化（Scale Down）”再水平扩展（Scale Out）。

为了更好的规模化，我们绝不应该以复杂来应对复杂、以规模应对规模。恰恰相反，我们首先要做的是“去

规模化”。

再大组织，也是由一个个业务和产品交付团队构成的。相对闭环的业务和潜在可独立交付的产品，这是实施的基本单元。我们应该解耦复杂的组织，识别可独立实施单元，把复杂性尽量控制在实施单元以内，而在实施单元之间维持尽可能松的耦合，如：不同业务单元之间通过定期的目标对齐，而不是统一的交付规划来连接；关联的产品之间以业务规划来拉通，而减少执行上的耦合。

以去规模化（Scale Down）为基础，在相对独立的单元以尽可能简单的方式实现系统性的转型。我们就可以基于成功的经验复制和水平扩展（Scale Out）出去。水平扩展是源自系统设计的原则，如：互联网网络和云计算基础设施都遵循水平扩展原则，让它们能走到了今天的规模。它同样适用于 DevOps 的实施。

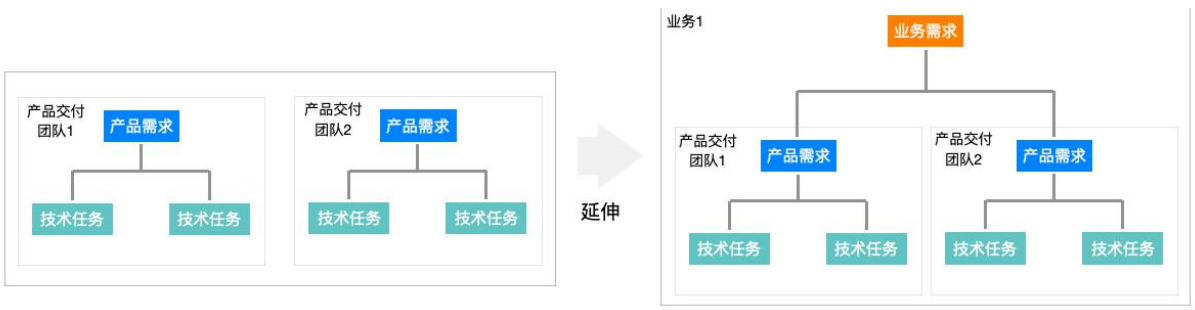
原则三：渐进原则。从现状出发，寻求持续且渐进的改进。在系统改进目标的牵引下，以小步前进，每个小步都带来改进，最终实现系统和全面的转型。

渐进不是妥协。恰恰相反，它是在明确且坚定的最终目标下选择更坚实和可持续的路径。对于转型，现状是我们要改变的东西。但改变必须从理解和尊重现状开始。识别 DevOps 实施的基本单元的基础之上，或者从局部环节开始到拉通整体价值流，或者是先拉通整体价值流，再在各个环节深入，最终实现整体的目标。

■ 规模化转型的实例和策略

介绍完 DevOps 规模化三原则，我们看两个具体的演进路径实例。

01 路径一：自下而上，从产品交付到业务



从独立的产品交付团队落地　　连接业务到产品交付团队的规模化实施

从产品交付团队到端到端业务

上图所示的是一个最典型的路径，它采取的模式是从局部延伸至整体。

首先，从产品交付团队开始，建立产品交付团队的持续交付能力，比如：引入精益和敏捷需求实践，梳理需求交付的端到端的价值流，建设持续集成和持续交付的工程能力，形成持续或迭代的交付模式。

接下来，以产品交付团队的持续交付能力为基础，通过业务拉通和对齐各个产品，将价值流从产品交付延伸至端到端的业务，实现持续顺畅地业务价值交付，建立快速的反馈闭环，保证价值的有效性。

这一模式的优点是启动相对容易，但是取得显著的业务效果会相对滞后。对于技术部门主导的 DevOps 转型可以尝试这一模式。重要的是，技术部门的改进，最终一定要拉通并体现到业务效能上。

02 路径二：自上而下，从业务到产品交付



从业务开始落地　　连接业务到产品交付团队的规模化实施

从端到端业务到产品交付团队

上图是另一个实用的路径，它采取的模式是从整体拉通开始，再寻求局部深化。

首先，从业务价值出发，梳理业务需求交付的端到端流程并形成反馈闭环。此时我们追求的是整体拉通，从一开始就关注整个系统。通过价值流的可视化和优化，就已经能够带来初步业务交付的改进。当然，更深度的改进，还需要深入各个局部。

接下来，在整体拉通的基础上，识别各个环节中的瓶颈和效率问题，驱动各个产品交付团队的变革，引入敏捷和精益的协作方式，和持续交付的工程实践。

这一模式的优点，以系统的优化驱动整个变革过程，更容易保证实施的业务结果和系统性的改进。但是，它要求组织，特别是业务层面对 DevOps 的高度认可和决心，并且要求实施者对系统改进有深入的认知。对于业务或高层主导的 DevOps 实施，可以尝试这一模式。

以上两种路径，不管是自上而下还是自下而上，它们有两个共同点。

首先，解耦各个业务线，把业务线当成衡量 DevOps 实施效果的基本单元，也就是我们前面所说的“去规模化”。通过去规模化实现更加平滑和可持续的规模化，这是一对矛盾的统一。

其次，最终都要走向以业务驱动的端到端的优化，实现 DevOps 规模化的目标，让更多业务，更系统性地受惠，这也是殊途同归。

现实中的转型路径往往是以上两者的混合体，是自上而下和自下而上同步进行，只不过不同时间的侧重不同。

DevOps 规模化三原则和实施三步工作法配合，相得益彰

以上我们介绍了 DevOps 规模化的三原则，并基于它给出了建议的规模化路径。

在概述部分，我们还介绍了 Kim 等人提出的 DevOps 实施三步工作法，它是 DevOps 的实施的实用且有效的指导，具体包含：

- **流动原则**：聚焦 IT 系统的整体价值流，全局优化，保证价值从左（上游）到右（下游）的快速流动。
- **反馈原则**：创建从左到右的反馈循环，并缩短反馈周期和放大反馈效果。这样，就可以更快的响应和理解内外部客户，并即时获取改进所需要的知识。
- **持续的实验和学习原则**：创建承担风险、持续实验并从错误中学习的文化，在不断的尝试中精进能力，并提高系统的韧性。

DevOps 实施三步工作法，可以应用在于各个层面的实施，如：业务层面、交付团队层面、持续交付工程实践的落地等。

DevOps 规模化三原则为 DevOps 规模化路径选择提供建议；DevOps 实施三步工作法则为具体的实施提供指导。它们相辅相成、相得益彰。

总结

为了支持组织的全面数字化转型，DevOps 的规模化应用是必然选择。但，规模化实施绝不等于实施复杂的规模化框架。恰恰相反，在规模化的路径选择上，我们应该以业务为驱动，寻求简单、可持续的方案，而“系统”和“普惠”是 DevOps 规模化的最终目标。

建设和提升持续交付的工程能力

03

阿 里 云 云 效

03

3.1 以特性为核心的持续交付

随着微服务架构和云原生技术的成熟，持续交付的理念也深入人心。持续交付要求开发团队持续、高频地向生产系统交付软件。然而，不断增多的服务数量，给企业交付流程管理带来了巨大挑战。同时，在 DevOps 落地的过程中，逐步开放生产环境的发布权限给到开发人员，这种松管控与企业安全生产存在冲突，多应用版本之间的协同问题也逐渐凸显。如何解决企业在新技术转型和 DevOps 落地中的痛点，拿到技术变革带来的效能红利，是每个企业包括阿里巴巴都必须面对和解决的难题。

■ 软件交付挑战

阿里巴巴 2008 年对淘宝巨型服务进行拆分以后，逐步形成了一套适用于服务化、分布式架构的中间件体系，解决了复杂系统性能和稳定性在业务高速扩张中的瓶颈问题。随之而来的是应用变多、架构依赖复杂、人员数量高速膨胀、技能参差不齐等等问题。总结下来有以下几个方面：

01 应用数量多

微服务架构被广泛应用以后，首先面临的就是应用数量的快速膨胀。原有研发流程也必须从批量发布模式向持续交付模式转型，否则会导致发布软件的风险和回滚的复杂度不可控。另一方面，测试和运维的工作量因为应用的膨胀而倍增，变成整个研发团队的效率瓶颈。打破这种瓶颈的方法就是 DevOps 的全面落地，把整个软件交付过程交给开发来主导，从而解除瓶颈提升效率。

02 架构依赖复杂

微服务架构让应用内依赖变为了应用间依赖，变更过程无法做到原子化，因此需要很好的模块拆分和接口设计。一方面减少单特性覆盖的应用数量，变更顺序可控、回滚风险可控；另一方面单元测试能覆盖的场景需要集成测试来覆盖，导致开发过程对测试环境的使用频度和依赖度变高，需要稳定、可靠的环境来保障所有开发人员都可以并行工作。

03 测试资源成本大

测试环境受到资源成本和运维成本双重制约。在业务发展初期，可以采用全链路完整部署加上多套环境的方式来满足研发团队的要求。但是随着业务的快速发展和研发团队的快速扩张，不断地增加环境在成本上已经无法负担。因此需要一套运维高度自动化，高度弹性随用随取，并且可以实现局部隔离的测试环境方案来满足多版本部署需求。

04 研发协同难

研发环节的协同分为开发间协同和测试，开发、运维多角色间协同两种。前者主要解决并行开发、按需上线的问题。后者解决的是在一个交付流程中各司其职、互相约束，确保软件能高质量、安全交付的问题。在 DevOps 场景下软件交付过程由开发人员主导，而测试和运维角色则需要承担流程守护、门禁卡点、提供自动化工具的责任。为了提升协同的效率，需要一个能够满足以上要求的工具平台来将团队的约定固化下来，确保团队各个角色可以高效率的完成工作。

05 线上风险大

线上的风险来自于两方面，一方面越来越高频的线上迭代意味着出错的概率也在变大，另一方面随着系统规模变大，传统人防人治的手段已不可能满足风控要求。因此必须从出错可能性和出错影响面两个方面系统性地去解决问题，前者关注能否在出错之前对风险进行拦截，而后者关注系统变更影响的用户数量和频度。这两种主动和被动防御措施的相结合，可以有效的解决风险控制的投入产出比问题，从而达到一个比较优的状态。

解决思路

为了解决以上在企业规模增长和新技术应用中的种种交付痛点，阿里巴巴不断探索和尝试，逐步摸索出一种适合这种业务发展快、软件迭代快、架构依赖复杂场景的交付方法和实践，我们称之为“以特性为核心的持续交付”。它有三个特点：

01 以特性为核心

特性是一个用户能体验到的产品能力的最小单元，其代码可能涉及到多个应用，因此特性也是协同多个开发团队完成一个能力的最小单元。以特性为核心的交付过程管理可以有效地将开发、测试等角色连接起来并统一推进，比如组织隔离测试环境、运行自动化测试、编写测试用例、做好测试验收等等。

02 以应用为载体

应用可以直接对应一个服务，是提供一种业务能力的最小单元，也是软件交付和运维的最小单元。可以通过应用串联代码、流水线、环境、测试和资源，以及外围工具链比如监控、数据库、运维、中间件等。开发人员可以在工具平台上定义他的应用，及应用的交付运维过程，比如配置流水线、规划环境、创建资源、设置部署策略等。以独立应用为载体的交付流程可以实现软件交付的原子化，并强迫开发降低应用间的耦合性，同时避免系统级集中式交付模式的惯性。

03 松管控与强卡点

在软件高速迭代下需要兼顾质量和效率，DevOps 模式需要给开发人员足够的自由度来完成软件的线上变更，阿里巴巴结合自身业务特点，在实践上采用了松管控和强卡点结合的方式。“松管控”表现在有多种流水线可以供开发选择，应用负责人可以完整定义这个应用的各种规则，比如如何部署、如何测试、资源环境如何配置。在技术可控的前提下，还可以开放线上测试，比如全链路压测和全链路灰度。轻发布，重恢复，在每一个应用维度，开发可以随时使用流水线来交付代码，不主张过多的人为限制，重点需要思考的是如果出问题，如何控制影响面，如何快速恢复。“强卡点”是一种软件质量底线思维的体现，比如代码审核和质量红线，规约检查，发布窗口，安全检查，线上灰度卡点等。这些卡点是为了保障集团所有开发工程师步调统一，交付合格的产品。

持续交付的核心是快速高质量地交付价值，给与开发人员最大自由度，负责开发和运维全部过程。在监控、故障防控工具、功能开关的配合下，可以在保障用户体验和快速交付价值之间找到平衡点。

总结

今天，基于云的开发已成为主流，这是效能提升的巨大机会，同时又对工程实践提出了前所未有的要求。比如，云原生基础设施、云原生中间件和新一代的云软件编程方法等等，都要求有与之适配的实践和工具。在适配新的技术发展趋势过程中，阿里形成了以特性为核心的持续交付工程实践，并且将其内建到 DevOps 工具体系中，以保障实践准确、有效地落地。

接下来的我们将按照软件开发和交付过程逐一介绍，具体包括：开发、调试、测试、集成、交付。

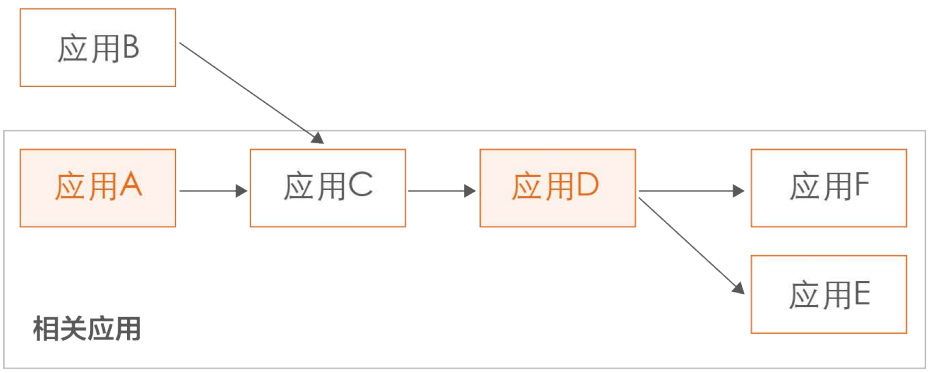
3.2 本地开发

开发一个需求，需要先进行代码的编写和个人验证，验证功能符合预期之后，再提交代码，并进入到集成环境，进行进一步的验证及验收。而这个编码和验证的过程占据了整个需求交付的大部分时间，因此提高这部分工作的效率就显得至关重要。

问题

有什么因素降低了开发调试的效率呢?

给定下面一个系统，其中为了开发某个需求，修改了 A 和 D 这两个应用（这里的应用指的是一个可提供服务的一组独立进程加上可选的负载均衡，比如一个 kubernetes 下的 service 及其后端的 deployment）。



接下来看看为了本地调测这两个应用，会遇到什么问题。

本地难以启动整个系统

我们通常都在开发一个复杂系统中的一个应用，这个应用可能在系统的最前端，也可能在系统的中间位置，有时候为了端到端验证整个流程，需要把相关的应用都启动起来。

比如上图中的应用 A 为最前端应用，应用 D 处在中间位置，而黑框中部分是为了完整的测试这个需求而涉及到的应用，如果是 Java 应用，开发机上启动这样 5 个进程，就已经不堪重负了，而很多时候需要完整启动的应用数量会远大于这个数字。

01 依赖系统不稳定

既然不能把整个系统都在本地启动起来，那么本地就会一部分依赖于公共测试环境。虽然前面提到应该本地测试符合预期之后再把代码部署到测试环境，但不可避免的还是会出现一些 bug，导致测试环境不可用（这也是测试环境的价值所在，尽早的发现问题）。一旦依赖系统不可用，就无法正常的进行测试。

02 云原生开发模式下的测试环境的连通性

在基于 Kubernetes 的基础设施下，整个系统中大部分的应用通常不需要通过 Ingress 暴露到公网。如果你的测试环境是独立的 K8S 集群，那就意味着无法从本地无法访问到集群内的应用，那么依赖公共测试环境这件事情都无法进行，比如上图中 A->C，D->E，D->F 的依赖。

还有另外一种依赖，即上游应用对本地应用的依赖，比如 C->D 的依赖。但因为 C 是公共测试环境，不可以将所有的 C 对 D 的请求都打到本地来，这就需要某种机制来保证只有特定规则的请求会路由到开发本地的 D 应用。

外部依赖系统到开发环境的连通性

有一些测试链路需要接受一些外部依赖系统的回调，比如微信或者支付宝的回调等。而本地应用通常没有公网地址，这也给调试带来了一些困难。

中间件的隔离

分布式系统中经常会用到 RocketMQ 等消息中间件，如果使用了公共测试环境，就意味着 MQ 也是共用的，那么 MQ 的消息到底是应该被测试环境消费，还是某个个人的开发环境消费呢，这也是需要解决的问题。

高效本地开发

为了进行全流程的高效开发，应该尽量使用反馈比较快的验证方式，并及早发现问题，逐步进行更加集成，更加真实的测试。

一般来讲，一个开发过程可以经过下面的三个阶段：

1. 编码+单元测试。在小的逻辑单元的层面保证正确性。
2. 针对单个应用的集成测试，可能需要对依赖的应用进行 HTTP 级别的 mock。
3. 结合公共测试环境进行完整的集成测试。

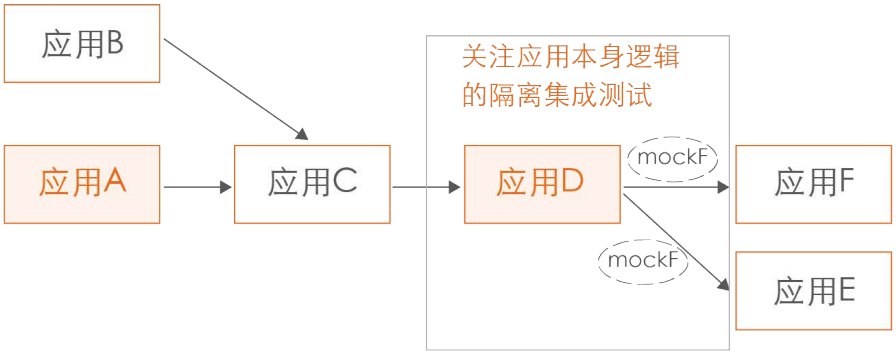
基于上面的三个阶段，可以使用以下的方式来解决前面提到的几个问题

1. 使用各个语言相应的测试工具（比如 JUnit）来进行单元测试
2. 使用 moco 等 HTTP Mock 工具来解决本地隔离验证的问题，完成单个应用的集成测试。
3. 使用 kt-connect 和 virtual-environment 等工具来解决云原生基础设施下，本地和测试环境的互相连通性问题，及 http 请求链路的染色和路由。
4. 使用 ngrok 等工具解决外部依赖调用本地应用的问题
5. 使用“主干稳定环境”作为公共测试环境，提高其稳定性
6. 使用中间件的染色隔离能力保证 http 请求之外的其它链路（比如消息）的染色和路由。

其中第 1、4 是成熟的技术，这里不再赘述。第 5、6 点会在后面的测试环境相关的章节中我们详细讲解。本文主要就第 2、3 点展开讲解。

单应用的集成测试方案

比如对应用 D 而言，测试范围如下图的橙色框所示：



应用本身的持久化等依赖使用真实的（一般使用本地 DB），但外部应用（应用 E、F）使用基于 HTTP 协议的测试替身。这样就可以保证所有的依赖都是稳定的。并且也可以很方便的修改测试替身的行为，以进行特定场景的测试。

应用 D 依赖了两个应用：

1. org-service（应用 F）：提供查询组织信息等能力
2. user-service（应用 E）：提供查询用户信息等能力

这两个应用的访问地址配置在应用 D 的配置项中：

```
...
org-service-host: org-service
user-service-host: user-service
...
```

我们使用 docker compose + moco 的方案来讲解如何使用本地测试替身。

首先创建如下的目录结构：

```
├── Dockerfile
├── docker-compose.yml
├── moco-runner.jar
└── services
```

```
    ├─── org-service
    │    └─── config.json
    └─── user-service
         └─── config.json
```

Dockerfile:

```
FROM openjdk:8-jre-slim

ARG SERVICE

ADD moco-runner.jar moco-runner.jar
COPY services/${SERVICE}/config.json config.json

ENTRYPOINT ["java", "-jar", "moco-runner.jar", "http", "-c", "config.json", "-p",
 "8080"]
```

docker-compose.yml:

```
version: '3.1'
services:
  service-f:
    ports:
    - 8091:8080
    build:
      context: .
      dockerfile: Dockerfile
      args:
        SERVICE: org-service
  service-e:
    ports:
    - 8092:8080
    build:
      context: .
      dockerfile: Dockerfile
      args:
        SERVICE: user-service
```

services/org-service/config.json：

```
[
  {
    "request": {
      "uri": "/"
    },
    "response": {
      "text": "org service stub"
    }
  },
  {
    "request": {
      "uri": {
        "match": "/orgs/[a-z0-9]{24}"
      }
    },
    "response": {
      "json": {
        "name": "some org name",
        "logo": "http://xx.assets.com/xxx.jpg"
      }
    }
  }
]
```

services/user-service/config.json：

```
[
  {
    "request": {
      "uri": "/"
    },
    "response": {
      "text": "user service stub"
    }
  },
  {
    "request": {
      "uri": {
        "match": "/users/[a-z0-9]{24}"
      }
```

```
    },
    "response": {
      "json": {
        "name": "somebody",
        "email": "somebody@gmail.com"
      }
    }
  }
]
```

然后使用如下命令来启动两个依赖的应用：

```
docker-compose up --build
```

验证下本地测试替身的行为：

```
$ curl http://localhost:8092/users/11111111111111111111111
{"name":"somebody","email":"somebody@gmail.com"}
$ curl http://localhost:8091/orgs/11111111111111111111111
{"name":"some org name","logo":"http://xx.assets.com/xxx.jpg"}
```

然后再把应用 D 的依赖配置改成本地测试替身即可进行测试：

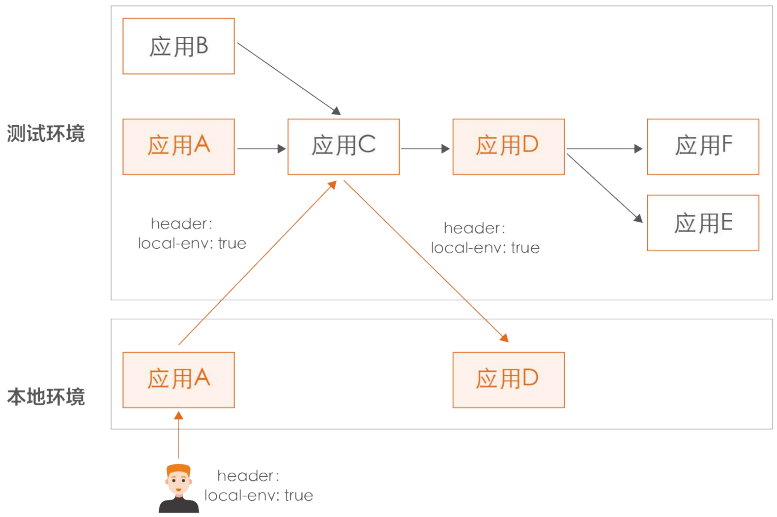
```
...
org-service-host: localhost:8091
user-service-host: localhost:8092
...
```

至此，我们得到了一个稳定的单应用的集成测试环境。当需要修改依赖的行为时，只需要修改相应应用的 config.json 即可。

使用 docker-componse 和 moco 是一种实现单应用集成测试的方式，你可以根据项目的具体情况选择合适的工具和方案。

本地和公共测试环境的互访及链路隔离

完成单应用的集成测试之后，可以获得单个应用级别的质量信心，但更大范围的验证还是需要和真实的依赖集成在一起进行。



如上图所示，为了能够在本地按需启动应用（A 和 D），并复用测试环境的其他应用（C），就需要解决两个问题：

1. 本地如何调用到公共测试环境的应用，即 A 如何调用到 C

2. 公共测试环境如何调用到本地，即 C 如何调用到本地的 D

关于第一点，如果本地环境和测试环境的网络是直接可达的，则直接修改本地应用 A 的配置项即可。如果你使用了云原生的基础设施，那么就需要类似云效 kt-connect 之类的工具来进行打通，这里不再展开，有需求要的可以参看 kt-connect 的 connect 部分。

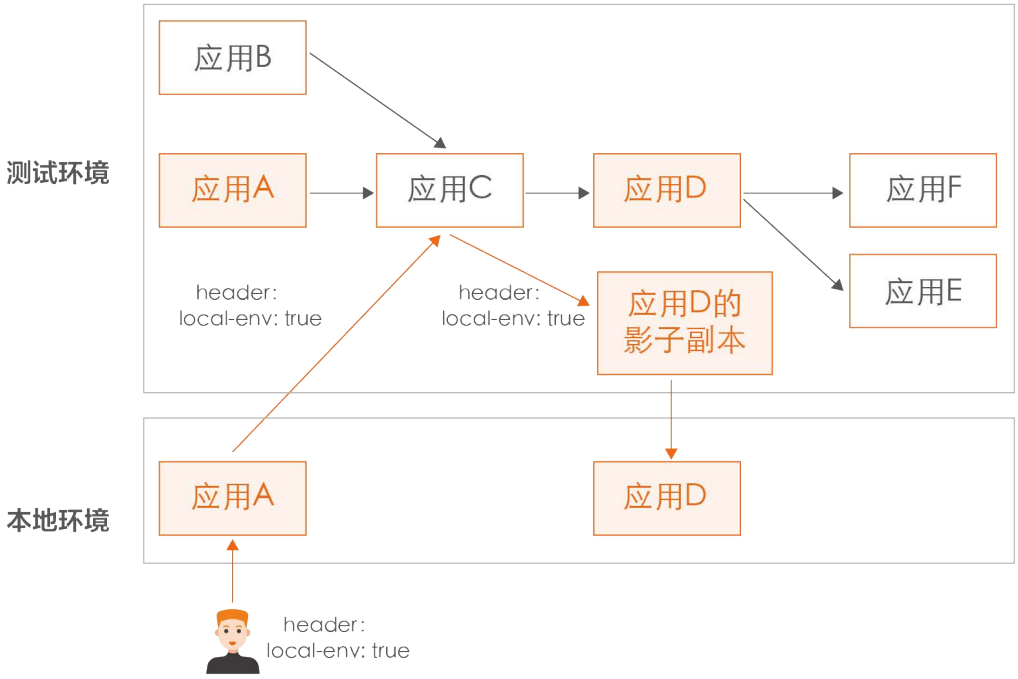
关于第二点，需要解决三个问题：

1. 从测试环境的 A 发起的调用链，应该最终访问到测试环境的 D，而从本地环境的 A 发起的调用链，应该最终访问到本地环境的 D，互不影响。为了能够对这两种调用进行区分，需要对调用链进行“染色”，这里采用的染色的方式是在请求中加入一个额外的 header。

2. 根据这个染色的标志，即“染色标”，进行路由。

3. 一个调用链会贯穿多个应用，要保证在调用到不同的应用时，染色标要能够自动的传递下去。

关于第一点和第二点，在阿里巴巴内部有一套完整的方案进行染色和路由，这套方案不仅仅适用于 HTTP 链路，也适用于 RPC，异步消息等。而在开源领域，也有基于云原生基础设施的 kt-connect 可以用，使用 kt-connect 的 mesh 功能就可以针对特定染色规则的调用链进行路由。



kt-connect 基于 istio 的 VirtualService 和 DestinationRule 来进行路由。其基本原理是在集群内新建一个影子副本的 service 和 deployment，然后提交一个应用 D 的 DestinationRule 资源，使得包含“local-env: true”header 的请求被路由到应用 D 的影子副本，然后应用 D 的影子副本再把请求转发到本地。在这个过程里，除了提交和更新 istio 相关资源的操作需要手动进行之外，其他的事情都可以使用 ktctl mesh 命令来完成，详情请参看 mesh 最佳实践。

接下来解决第三点，染色标传递。即需要保证当本地的应用 A 把含有“local-env: true”header 的请求打到测试环境的应用 C 后，应用 C 继续访问应用 D 时候，请求中也应该包含这个 header。

一般的思路是在 Web 层的入口加一个 Interceptor，将染色标记录下来到一个 ThreadLocal 中，然后再出口的 HttpClient 层再从 ThreadLocal 中把这个染色标取出来，并填充到 Request 对象中。这里有一个需要注意的问题，因为染色是放在 ThreadLocal 中的，因此在一个 web 请求的处理中一旦遇到多线程的情况，就需要小心的把这个 ThreadLocal 的值传递到相应的子线程中。所有的应用都正确的将染色标传递下去，就

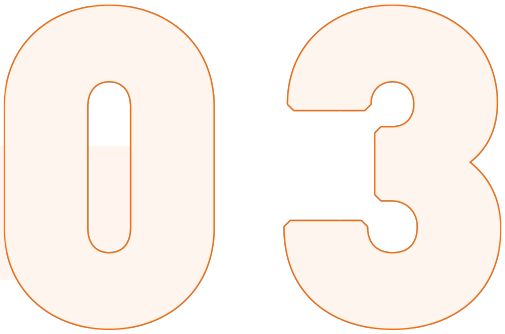
可以保证染色标在全链路进行传递。

使用 kt-connect 的 mesh 方案加上全链路染色标的方案，就可以轻松的在本地按需启动应用，并进行开发调测。

总结

1. 使用单元测试、单应用集成测试、端到端集成测试结合的方式进行本地调测，提高获得反馈的效率。

2. 本地按需启动应用进行端到端集成测试的关键技术是：全链路染色和路由。在不同的基础设施下可以有不同的实现方式。

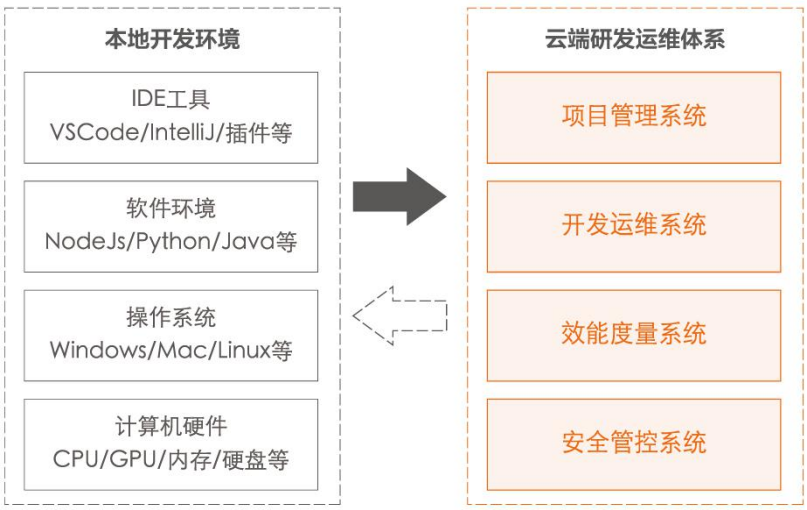


3.3 云端开发

云端开发指开发者可基于云平台完成编码、测试、发布等研发流程。一个完整的云端开发平台不仅是提供了一个云端的编码环境，还提供了一整套研发工具和配套设施，让开发者做到在云端即可完成应用程序的需求、编码、测试和运维的全生命周期管理。

■ 传统的本地开发的问题

如下图所示，在传统的开发模式中，企业研发人员通常在本地完成代码的编写和测试，然后把代码推送到远端服务器，通过一系列的构建和集成，最终发布到生产环境，并持续利用线上的运维体系完成线上系统的监控和运维；同时，企业也会采集部分研发过程中的关键数据，用来度量团队及个人的效能。



随着各种软硬件技术逐渐更替，公司规模也越来越大，为了适应这种变化：

- 公司需要不断为企业研发人员配备合适的本地研发工具（如：多核高内存的计算机设备、Mac 笔记本电脑），这些设备可能价值不菲，而且需要定期的更新换代；
- 新加入的员工，在正式开始开发前，需要配置复杂的本地开发环境，安装特定的软件及插件，并熟悉项目的研发流程及各个线上系统；部分项目因为网络配置等问题，可能第一时间还无法在本地启动，还会耽误不少额外的配置及调试时间；
- 公司需要投入较多的资源，才能构建起匹配管理者需求的效能度量系统和安全管控系统，并且因为云端体系天生对开发者本地环境的弱管控性，效果只能差强人意；

阿里巴巴也不例外，随着近些年各项业务的飞速发展，人员的快速扩充，如何解决发展过程中带来的类似问题变得迫在眉睫。而云端研发作为一种新兴的技术形式，其独特的优势恰好可以用来解决上述问题。

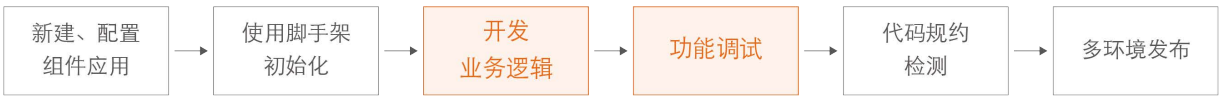
云端开发的典型应用案例

01 案例 1：前端组件的开发

在阿里内部，存在大量的基于 Node.js 构建的前端工程，这些前端工程普遍采用模块化的组织方式，在开发过程中会随着需求迭代产生众多的模块（或组件）。同时，有些前端工程会邀请业务方参与共建，即由提出功能需求的团队在大的标准下自行开发组件，并发布上线，在平台中集成自己的场景。

在这样的背景下，组件的开发会被高度的抽象，大部分的步骤都可以由工具辅助完成（如下图中，业务开发人员只需要关注自己的业务逻辑即可），这样既提升了研发效率，又提升了组件的开发质量。

前端组件开发过程：



云端开发的开箱即用，恰好可以解决类似的问题。开发者打开浏览器就是一个配置好的环境，实现零配置上手；而环境配置可以由项目组的资深同事维护，配置好针对某个项目的系统版本、程序运行时、SDK 和 IDE

插件集合。相比使用本地的研发工具，云端开发可以实现：

- 研发流程的产品化，从组件的新建到最终的发布一气呵成，不用再在多个平台工具上来回跳转；
- 屏蔽用户操作系统的差异，提供统一的研发环境，不用再解决 Windows/Mac 的差异，不用担心本地 Node.js 的版本问题；
- 所有环境都会预装好必要的开发提效工具，如：规约扫描和修复工具、预览调试工具、各环境发布工具等；
- 充分释放本地磁盘空间，不用担心磁盘被 node_modules 占满；

02 案例 2. 代码安全管控与研发过程数字化度量

在政务、金融以及部分高科技企业的研发场景中，对代码的安全管控要求极其严格。但近几年，公司内部源代码泄露的事件时有发生，有的被明码标价进行出售，标价数十万甚至上百万美元；有的直接被公开在网络上，任何人都可以访问下载。一旦发生类似事件，将会直接或间接造成商业信息泄露及公司声誉受损。

当使用本地开发时，源码的传输环境、本地的持久化介质不可控，对于员工有意泄露源码的行为似乎无可奈何。当使用云端开发时，一切都迎刃而解：开发者可以从代码库或需求直接打开网页开始云端开发，研发过程中代码不落本地磁盘，既能减少传输风险，又避免了员工本地环境被植入木马、从而在不知情的情况下泄露源码的可能；同时，在云端开发环境中可以对用户的浏览、拷贝行为做不同程度的管控，结合告警和系统自动拦截，可有效降低源码泄露的风险。

在阿里内部，当涉及到对保密性要求极高的项目，或者当企业外部成员参与对代码保密性有要求的项目时，我们会推荐项目团队使用云端进行研发，从而有效防止源码的泄露。

此外，随着越来越多的企业进入到数字化转型阶段，管理者期望能更加全面的看到企业员工的投入与产出，并且针对项目人员分布与研发过程效率做出更加及时的调整与改进。在过去，所有的数字化信息都依赖人工的反馈和统计，反馈的是否准确、统计过程中是否有纰漏都会直接影响管理层的判断。但如果把研发过程搬到云上，所有的研发过程数据就能生在云上、用在云上，想要借助数字化提升研发效率变得更加容易。

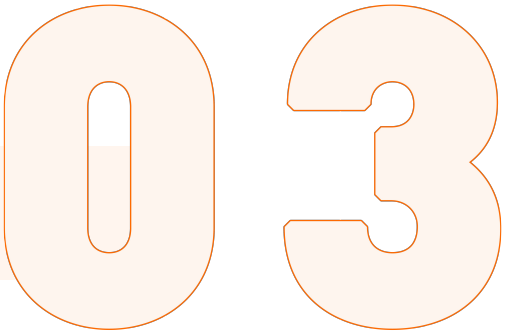
在阿里内部，团队中经常会出现一名正式员工带领多名企业外部成员完成项目的情况。在需要对企业外部成员的工作进行绩效评定时，传统的评定方式通常是参考需求完成数量、代码缺陷率等指标，但实际工作中需求有大有小、有难有易，完全基于结果指标进行评定很难做到公平公正，让优秀的员工脱颖而出。借助云端开发，可以让所有研发过程中的数据也透明出来，如各需求的编码时长、临时版本发布次数、过程代码与最终有效代码的比例、单位时间代码产出量等。通过结合研发过程数据，也可以让绩效评定更加透明公正。

总结

云端开发具备灵活定制、开箱即用的特点，借助好这两个特性，就可以创新性的解决掉传统本地开发过程中的顽疾。除了上述两个案例外，我们认为，当前适合云端开发落地的场景还可以是：

1. 云原生场景中的轻量代码开发，如 Serverless 场景，这类场景中研发人员只需要集中式的编写业务逻辑，大量的框架类代码已被默认隐藏，并且调试、部署方式有别于传统研发过程，更适合云端开发的落地。

2. 各类垂直化的场景，这类场景通常需要有针对性的定制，与特定的线上系统进行打通，只要能利用好云端开发灵活定制的特性，就有望实现开发阶段 10 倍效能的提升。



3.4 代码评审

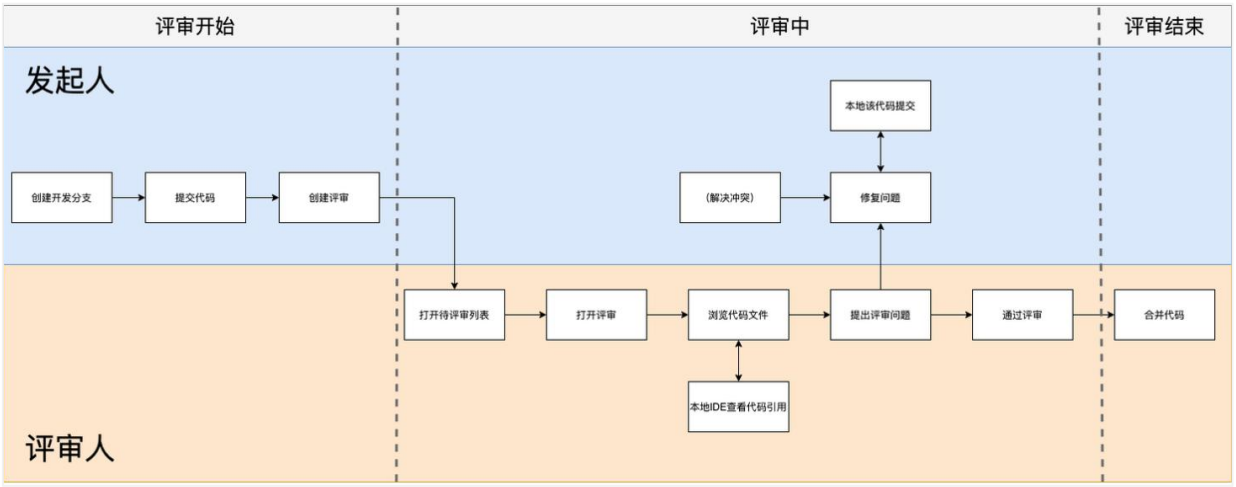
代码评审，英文名是 Code Review，简称 CR，它是结对编程相互切磋相互学习的方式。严肃地讲，CR 能够提升代码质量、促进人才成长、培养技术情怀。

首先，代码也是一种资产且具有“流通性”，通常会需要 3 到 5 年的维护。过程中将面临维护人员更替、其他人参考引用的情况，谁都不希望自己接手或参考的是一份“天书”一样的代码。所以通过 CodeReview 能够发现除程序逻辑以外的设计、性能、安全、规范等多方面的问题。

其次，CodeReview 是一个互相检查错误，互相学习代码用法的机会。如果团队的核心骨干人员，能参与到团队日常的架构评审、设计评审以及代码评审中，一定可以切切实实地帮助到其他研发人员的成长，不论是新人的融入，还是处于瓶颈期的其他研发人员。我们期望看到 CodeReview 可以促使团队"战斗力"的提升。同时 CodeReview 因为涉及到研发之间就某一个具体的问题或场景交互式的讨论，所以也成为了工程师们提升编程能力的最佳途径。

最后，CodeReview 是一种文化，通过一个个小团队内部的评审实践逐渐形成为 IT 企业的一种开发者文化。当一个企业拥有了特定的技术文化，这对人才的吸引和成长是至关重要的。

传统评审流程的问题



如上图所示，代码评审主要分为三个阶段：评审开始、评审中、评审结束。

1. 评审开始阶段，发起人需要从大量库成员中选择最熟悉改动代码的若干人作为评审人，选择开发分支和目标分支，填写评审描述信息，建成评审，评审通知会发送给受邀的评审人。一个评审人通常会收到多个评审邀请，他需要安排工作的间隙选择适合的评审各个击破或者用固定的时段集中评审。评审人点开评审，依次浏览代码逻辑，对于比较复杂的嵌套或调用依赖，需要去本地 IDE 查看内部逻辑和影响范围。

2. 评审过程中，负责的评审人会基于漏洞异常，代码风格，性能缺陷，重复代码等维度提出评论。评审发起人收到评论通知后，会重新审视问题代码，修改或重构局部问题行，再次提交并更新评审。如此迭代，直到问题被解决。

3. 评审结束后，评审人点击通过评审，如果源分支和目标分支有代码冲突的话，需要评审发起人去解决冲突，最后合并代码。

综上所述，传统的代码评审流程主要有以下几个问题：

1. 创建速度慢，评审的创建需要手动填写大量信息：选择分支，填写描述，选择评审人，关联工作项等。

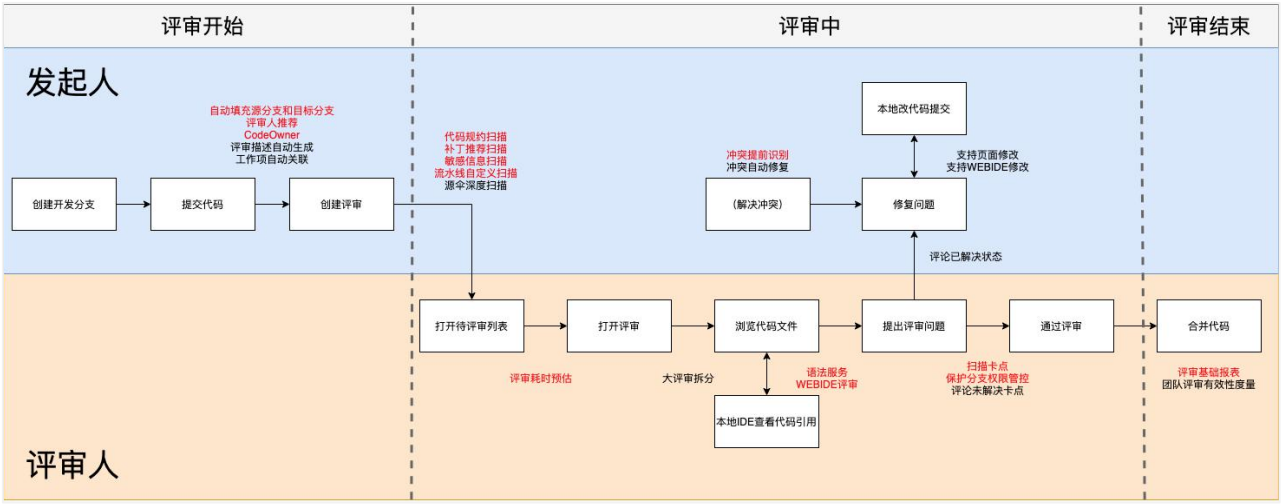
2. 人力投入大，最传统的代码评审是结对编程，以及团队圆桌评审，人力的投入显而易见。

3. 评审内容杂，不规范的评审会存在变更文件过多，变更版本错乱，多个需求混合等现象，导致评审的代码难以理解，评审过程无从下手，文件间逻辑难以关联，文件跨度过长忘记前文逻辑等问题。

4. 效果评估难，作为管理者常常在评审问题上犯难，他们认为评审是一个重要且正向的流程，促进代码质量和团队协作。但是团队的成员是否能真正践行评审保证评审质量，而不是随意通过走个流程，是管理者困惑且急需知道的。同时管理者也想知道评审流程的投入是否能带来足够的回报。

智能代码评审

针对传统评审流程面临的问题，阿里巴巴的代码平台团队借助智能化和流程管控的手段，提升评审效率，如下图所示：



具体从以下几个方面进行优化：

1. 通过对代码库维度的评审数据进行智能分析，自动完成创建评审过程中的信息填充从而提升评审创建的效率，同时还提供以客户端命令行的方式快速创建评审。

2. 通过丰富的代码自动化检测工具链，尽可能减少人工 review 的内容。例如通过阿里巴巴 Java 规约检测工具自动发现不符合代码规范的代码并给出修复建议。

3. 通过大评审拆分，针对包含文件数过多的评审按文件相关度进行智能拆分，将一个超大的评审拆分成若干个小评审，并根据评审的内容和当前评审人的工作饱和度智能选择合适的评审人进行评审。这样通过精简评审内容来提高评审的有效性，避免走马观花的形式评审。

4. 评审虽然依靠人工，但整个流程是数字化的，提供了丰富的代码报表数据，帮助管理者更好的管理和优化评审流程，指导管理者对评审带来的效果进行有效的评估。

01 评审人推荐

通过智能算法，自动推荐最合适的评审人。对于大型应用，和跨团队共建的中台应用，自动推荐可以节约大量的沟通成本，快速推进评审流程。

02 评审耗时预估

通过提交数据分析当前一次评审预计需要多长时间可以完成，帮助评审人根据当前评审所需的时间来合理安排自己的工作。此外，对于一个评审人同时拥有多个代码评审的场景，还可以根据评审耗时来安排评审任务的优先级。

03 git-repo 评审工具

git-repo 是由阿里巴巴开发的一款客户端工具，可以从客户端直接发起代码评审，实现在客户端快速创建并在页面端快速打开浏览。git-repo 并不会改变 git 用户的使用习惯，而是提供了对 git 命令的扩展。

传统的代码评审工作模式，代码贡献者要将代码推送到个人/特性分支，再通过在浏览器页面上发起创建合并请求，整个过程要经历多个步骤，开发者要切换到不同的工具才能完成。而使用 git-repo，一个用户只要拥有仓库的读取权限，就可以在本地工作区中执行下面的一条命令，将代码以合并请求的方式贡献到服务端。

总结

在阿里巴巴内部，从上自下都在推行代码评审文化，要求每个开发者不仅要对自己的代码负责，也要作为代码评审人帮助其他同事一起把控代码的发布质量。想要做好一个评审，除了需要好用的工具辅助之外，还需要开发者注意以下几点：

- 做好提交，即将提交做小做好。写小提交就是将问题解耦：“Do one thing and do it well”。每个提交控制在只修改一个到几个文件，每次只修改几行到几十行。每个提交应该是一个完整的功能，可能是修改某个 bug 或完成某个小需求的开发。
- 充分描述，对于代码评审描述应介绍本次改动的需求背景，改动范围及影响点。一段清晰的评审描述能让 reviewer 充分理解需求背景，快速开始评审，降低沟通成本。
- 数字化度量，通过数据洞察获得团队质量情况，有策略的提升团队技术能力。



03

3.5 代码检测

随着业务演进和团队扩张，软件规模和调用链路越来越复杂。如若没有良好的代码检测机制，只依靠功能性验证，团队技术债会越累越高，开发团队往往要花费大量的时间和精力发现并修改代码缺陷，最终拖垮迭代进度、协作效率，甚至引发严重的安全问题。

通过观察近几年业内暴露出的问题，以及长期以来阿里巴巴内部的研发经验，通过合理使用代码检测分析工具可以大量减少常见缺陷问题。同时，代码检测工具能够帮助开发同学快速定位并纠正代码缺陷、帮助代码设计同学更专注于分析和解决代码设计缺陷、减少在代码人工检查上花费的时间、提高软件可靠性并节省开发成本。

基于这些优势，将代码检测工具集成进持续集成体系，就可以在前置流程中提早暴露代码中隐藏错误和缺陷。通过设置卡点流程，可以进一步保障项目代码质量和可维护性。

■ 面临问题

在日常研发过程中，我们通常面临的代码资产问题主要分为两大类：代码质量问题和代码安全漏洞。

01 代码质量问题

代码质量其实是一个老生常谈的话题，但问题是大家都知道它很重要，却又不知道如何去提升和维护这一团队的共同财产。一方面开发人员可能为了功能及时上线，疏忽了对质量的把控，另一方面开发人员编码习惯和程序理解风格各异。长期下来代码质量下降通常会自成因果，因为业务压力大而趋于下降，又因此开发效率

下降，进一步加大业务压力，导致恶性循环。

在实践过程中，我们通过代码评审、集成测试、代码检测、代码规范等一系列手段保障代码质量和可维护性，从流程上保障开发团队能高效协作。

02 代码安全问题

安全类问题往往隐藏在缺乏安全意识的编码逻辑和未经检测或维护的开源依赖组件中，难以在日常开发和代码评审中被及时察觉。

代码安全问题也可以分两个方面进行分析：

1. 编码安全问题，即：安全规范类问题，通过避免不符合规范的代码进入企业代码库，减少隐私数据泄露、注入类风险、安全策略漏洞的出现。

2. 依赖安全问题，即：开源依赖三方组件引入的安全漏洞。根据 Synopsys 2020 开源安全报告显示，99%以上的组织使用了开源技术。使用开源组件本身带来的技术交流和站在巨人肩膀上协作、降低开发成本、加快迭代周期、提高软件质量等优势已经不必赘述，但是，开源软件带来一系列便利的同时，也暗藏大量安全风险，据审计，75%的代码库存在安全漏洞，其中 49%包含高危问题，另外 82%的代码库仍在使用超过 4 年的 outdated 组件。

因此，代码安全类问题，一方面也需要进行准入性检查，根据业务场景和规范配置安全编码规范检测和卡点。另一方面需要定期维护，对于新爆发出的安全漏洞进行及时察觉并修复。

■ 解决方案

01 代码质量检测

● Java 代码规约检测

阿里巴巴在实践过程中，各个组织由于历史隔阂和业务风格差异，工程结构差别很大，代码风格迥异，规范不一，沟通成本大，合作效率低，维护成本高。集团发展到现在的规模，需要专业化的技术集团军迭代式、集约式发展，而不是动辄重复造轮，而真正专业化的团队一定会有统一的开发规约，这代表效率、共鸣、情怀、可持续。

基于上述背景，阿里制定了《阿里巴巴 Java 开发手册》，作为阿里内部 Java 工程师所遵循的开发规范，涵盖编程规约、单元测试规约、异常日志规约、MySQL 规约、工程规约、安全规约等。这是近万名阿里 Java 技术精英的经验总结，并经历了多次大规模一线实战检验及完善。

制订交通法规表面上是要限制行车权，实际上是保障公众的人身安全。试想如果没有限速，没有红绿灯，没有靠右行驶条款，谁还敢上路。同理，对软件来说，开发规约绝不是消灭代码内容的创造性、优雅性，而是限制过度个性化，推行相对标准化，以一种普遍认可的方式一起做事。

因此，代码规约的目标是：

1. 码出高效：标准统一，提升沟通效率和研发效能。

2. 码出质量：防患未然，提升质量意识和系统可维护性，降低故障率。

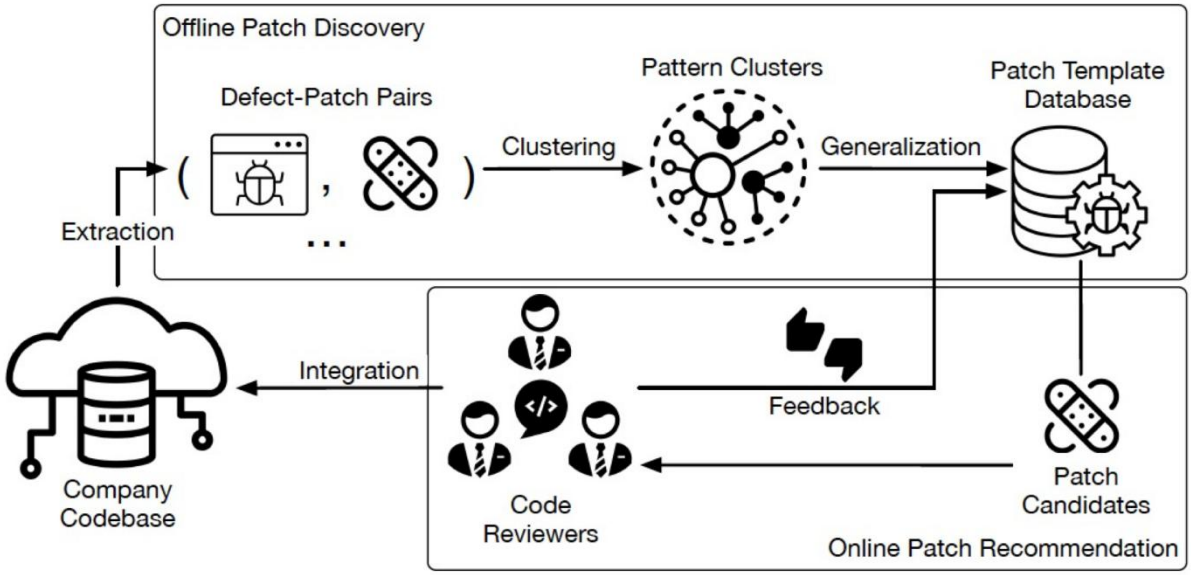
3. 码出情怀：工匠精神，追求极致的卓越精神，打磨精品代码。

代码规约通过 IDE 检测插件、流水线集成测试、代码评审集成等工具手段，深度融入了阿里巴巴的各种开发活动中。与此同时，在云效代码托管平台 Codeup 中，也内置集成了 Java 代码规约检测能力，为开发人员在代码提交和代码评审阶段提供更为方便的快速检查。

- **代码智能补丁推荐**

缺陷检测和补丁推荐几十年来一直是软件工程领域的难题，又是研究者和一线开发者最为关心的问题之一，这里讲的缺陷不是网络漏洞、系统缺陷，而是隐藏在代码中的缺陷。帮助开发者识别这些缺陷，并进行修复，能够大幅提升软件质量。

基于业界和学术界较为流行的缺陷检测手段，并分析和规避其局限性，阿里巴巴 Codeup 的算法工程师们提出了一种新的算法，实现更加精准和高效的分析代码缺陷并推荐优化方案，该算法已被国际软件工程大会（ICSE）收录。



1. 根据 commit message 中的关键词找出修复型的 commit，只取涉及文件小于 5 个的 commit（涉及文件过多的 commit 可能会稀释修复行为）。这个步骤十分依赖开发者良好的 commit 习惯，希望开发者能用好 commit，写好 message。

2. 从这些修复型 commit 中以文件级别提取删除内容和新增内容，即 Defect and Patch pairs(DP Pair)，这一步难免会有很大的噪声。

3. 利用改进的 DBSCAN 方法对 buggy 和 patch 对同时聚类，将相似的缺陷和补丁代码聚在一起。（也可以做片段级的聚类）通过将相似的缺陷和修复做聚类，减少了上一步留下的大量噪声，同时历史代码提交中大家共同犯过的错具有很强的借鉴意义。

4. 利用自研的模板提取方法总结缺陷代码和补丁代码，并根据不同的变量来适配上下文。

代码补丁推荐服务目前应用于合并请求的代码自动扫描场景，在代码评审过程中检测可优化代码片段并给出优化建议，将历史评审中的人工经验沉淀下来持续提升企业代码质量。

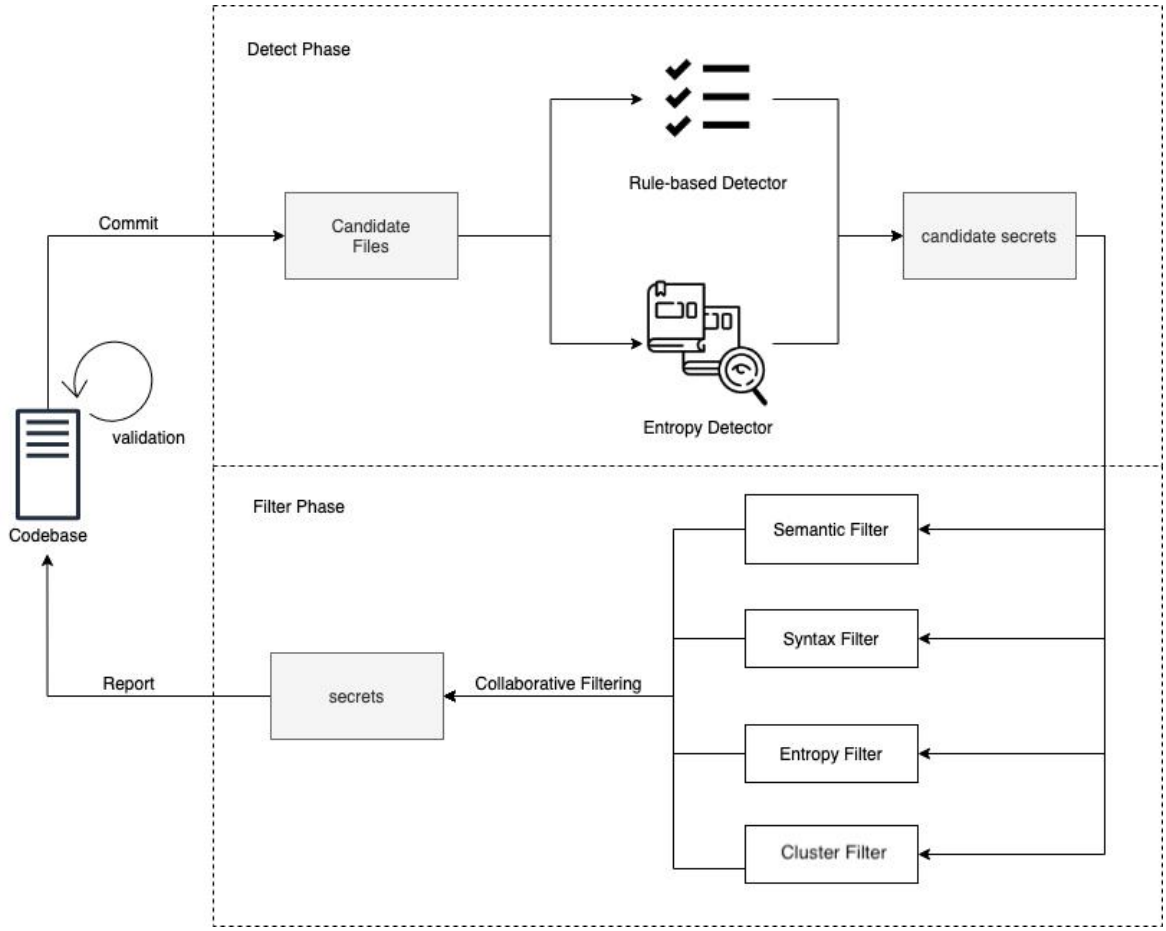
02 代码安全检测

● 敏感信息检测

近年来，业内发生多起敏感信息（API Key、Database credential、OAuth token 等）通过某些站点被无意识地泄露出去的事件，给企业带来了安全风险，甚至直接的经济损失。

在我们的实践过程中，我们也面临了类似的问题，硬编码问题出现非常高频，而又缺乏有效的识别机制。因此开发者和企业管理者亟需一款稳定健全的敏感信息检测方法和系统。通过调研我们了解到，目前已有的敏

感信息检测工具大多单纯使用规则匹配或信息熵技术，导致其召回率或准确率难以满足预期。因此我们在规则匹配和信息熵技术的基础上，结合上下文语义，提出了一款采用多层检测模型的敏感信息检测工具——SecretRadar。



SecretRadar 的技术实现思路主要分为三层，第一层采用规则匹配这种传统敏感信息识别技术，规则匹配具有良好的准确度和扩展性，但是非常依赖比较固化的长度、前缀、变量名，难以应对不同开发者的不同编码风格，容易造成漏报。针对难以固定规则捕捉的场景，在第二层我们采用了信息熵算法。信息熵算法用于衡量代码行混乱程度，对随机生成型密钥和随机身份信息识别效果良好。但信息熵算法也有其局限性，召回率提升的同时误报也增多了。因此在第三层我们采用了模板聚类和上下文语义分析等方法进行过滤优化，针对信息熵结果聚合提取常见关键字，结合上下文语义和当前语法结构提升模型准确率。

敏感信息检测工具不止服务我们内部开发同学，在云效平台上也支持了超过 2 万代码库、3 千家企业，帮助开发者解决了超过 9 万硬编码问题。

源码漏洞检测

阿里巴巴采用 Sourcebrella Pinpoint 源伞检测引擎，进行源码漏洞检测，主要涉及到注入类风险和安全策略类风险检测。

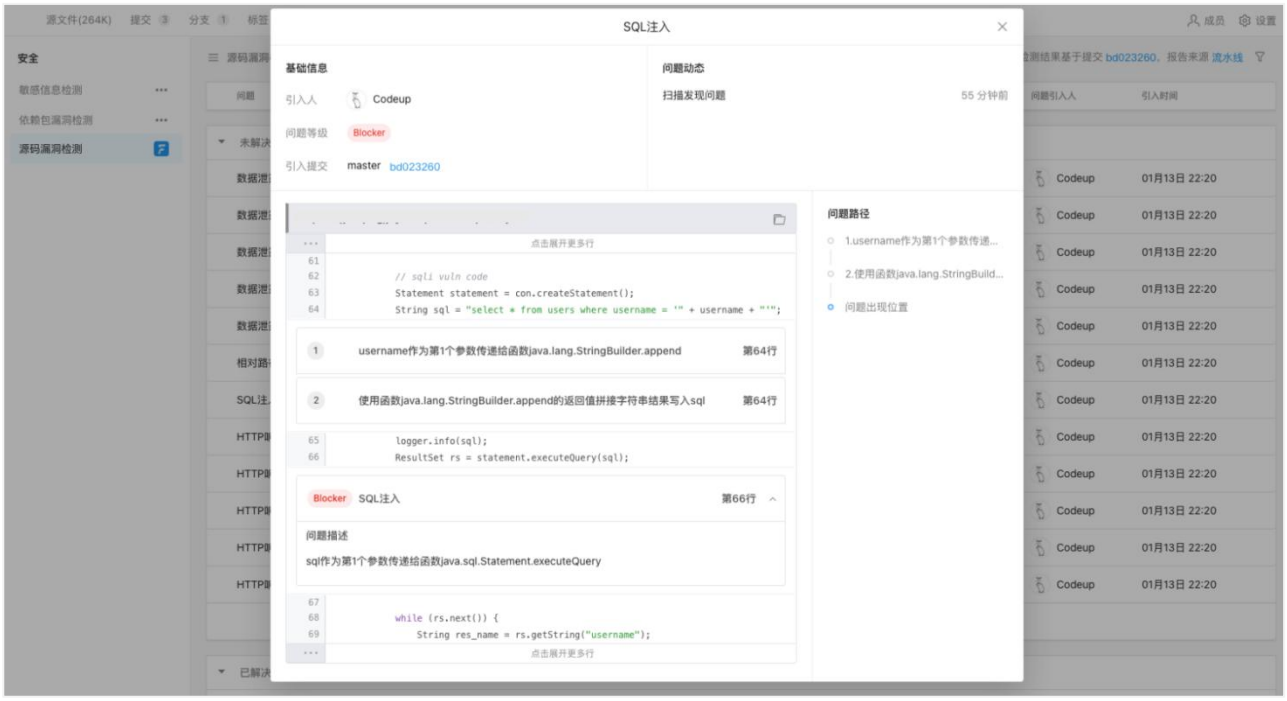
源伞检测引擎是香港科技大学 Prism 研究组在过去十年的时间内的技术研究成果。该引擎吸收了国际上近十年的软件验证技术研究成果，并且加以改进和创新，独立设计和实现了一套技术领先的软件验证系统。其主要验证方式是将编程语言翻译成一阶逻辑和线性代数等数学表达，通过形式化验证技术推理缺陷成因。迄今为止，一共发表了四篇核心技术相关的论文，一篇 PLDI 和三篇 ICSE，研究型的同学可以点击下方链接阅读：

- Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code [1]
- SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code [2]
- Pipelining Bottom-up Data Flow Analysis [3]
- Conquering the Extensional Scalability Problem for Value-Flow Analysis Frameworks [4]

源伞检测引擎能够在活跃度比较高的大型开源项目中发现隐藏超过 10 年的缺陷，以 MySQL 检测 [5] 为例，这些缺陷都是市面上其他检查工具无法扫描出来的，并且能够在 1.5 小时完成 200 万行大型开源项目的检测。在保持扫描的高效率同时，还能够将误报率控制在 15%左右。对于复杂且体量庞大的分析项目来说，源伞检测引擎所表现的扫描效率和误报率在业内也处于领先水平。

「源码漏洞检测」集成了源伞检测引擎的安全分析能力，能够在分析精度、速度、深度等方面均衡得到较好的分析结果，具备的核心优势：

1. 支持分析字节码，二三方包的代码逻辑都不会遗漏；
2. 擅长跨函数长调用链路的逻辑分析；
3. 可以处理引用、指针等带来的间接数据修改；
4. 精度高，相比于同类工具，如 Clang、Infer，在精度和有效问题识别上表现更佳；
5. 性能好，目前单应用平均 5 分钟左右分析完毕；



源伞检测引擎可以精确的追踪代码中的数据流向，拥有高深度高精度的函数调用链分析能力，可以找到跨越多层函数的深度问题。在发现缺陷的同时还能给出问题触发的过程，完整展示相关的控制流以及数据流，这样可以辅助开发者快速理解和修复问题，在软件开发早期更低成本地提高软件质量，大幅减少生产成本，提高研发效能。

● 依赖包漏洞检测

我们期望对于开源组件的安全可信程度，为开发者建立一种有效的检测和管理机制，因此我们实现了依赖包漏洞检测服务和依赖包安全问题报表。在实践过程中，开发者普遍反映依赖包漏洞修复成本大多高于修复自身编码漏洞，从而不愿意或难于处理此类问题。究其原因，一方面是大部分漏洞并非直接引入，而是依赖的第三方组件又间接依赖了其它组件，另一方面是不确定具体哪个版本是干净可用且兼容的。



为了降低开发者的修复难度，我们对依赖项的引用关系进行了进一步识别分析，清晰的标注出直接依赖和间接依赖，并定位到具体的依赖包引入文件，方便开发者快速找到关键问题位置。同时，通过对漏洞数据的聚合，智能推荐修复漏洞的版本升级建议，因为一个依赖可能对应多个漏洞问题，开发者可以针对建议评估是否接受采用。通过分析不同版本间的 API 变更和代码调用链路，衡量版本升级成本，为开发人员自动创建修复评审，最大程度的帮助开发人员更高效地维护代码安全。

03 检测服务应用

● 代码提交

检测服务的最直接的应用便是在代码提交场景中，企业可以根据业务场景和规范，制定和配置不同项目的检查方案。当开发者将代码变更推送至服务端，自动触发当前代码库配置的检测服务，可以为开发者检查当前 commit 版本中的全量问题，帮助开发者及早发现新增问题，并确认存量问题的解决情况。通过接入上述检测服务，可以从代码规范、代码质量、代码安全等多个维度进行测试左移，在开发人员刚完成编码时就进行快速检测和反馈。

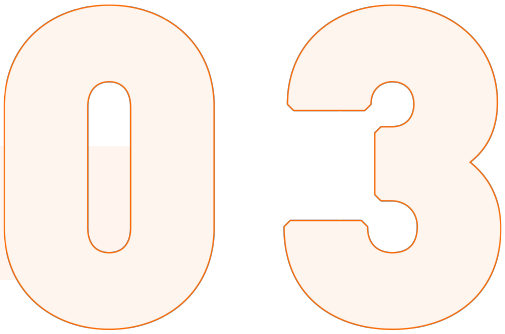
● 代码评审

在企业项目协同中，开发者多以合并请求的方式将特性分支代码合入主干分支，合并请求过程需要项目开发负责人或模块负责人进行代码评审和人工检查。一方面人工审查代码需要投入较大精力，另一方面人工审查

难以覆盖代码各个维度的潜在问题。因此，通过合理配置检测服务，可以极大地减少人工评审的工作量，加速代码评审的工作过程。同时，通过丰富、筛选、沉淀检测规则集和人工经验，检测服务可以更加贴合企业的业务场景进行卡点，避免不符合规范或存在风险的代码进入企业代码库。

● 代码度量

检测服务除了在代码提交和代码评审阶段帮助开发者及早发现问题并解决问题，还可以帮助管理者进行企业代码质量度量和风险可视化。通过建设企业级报表服务和项目任务管理，可以更为直观地度量项目演进过程中的安全问题和质量问题。



3.6 测试提效

在任何业务发展的过程中都会不可避免的面临服务的膨胀，应用复杂度的增加，可持续测试的难度不断增加。一方面，用例集会不断的膨胀，一次 CI 验证要数十分钟，用例的维护成本越来越高，开发效率开始降低。另一方面，我们花了精力写了很多自动化用例，希望能够提高投入产出比，也就是测试的有效性。

■ 提升测试速度

01 分布式测试

分布式测试的核心思想是通过增加计算资源，并发的对 case 进行执行，并在执行后对测试产生的结构化结果进行解析合并，进而提升单次测试的执行速度。

整个测试的执行过程可以划分为以下三个阶段:

- 测试用例解析与分发
- 分组用例的执行
- 分组测试结果合并



以阿里云某云产品核心团队的某工程为例，该工程拥有 1 万多个单元测试用例，在没有采用分布式测试方案之前，一次 CI 的验证时长超过 4 小时，导致问题发现、修复时长拉长，影响了日常的迭代速度。该团队后采取了分布式测试的方式进行，平均的执行时长被优化到了半小时以内。

分布式测试的本质就是用执行资源的堆叠，去换取更快的执行速度。理论上我们把每一个测试用例拆分到一个容器内执行，可以获得极致的反馈速度。但是并不是所有场景下都适合采用分布式测试，例如用例之间存在依赖，这些用例不能无差别的分布在不同的执行分组。

02 精准测试

分布式测试很大程度上解决了测试执行的速度问题。但是如果在任何情况下都无差别的执行全量的用例，会存在一些问题：

- 对计算资源的浪费
- 引入了大量的无效执行
- 用例本身稳定性问题导致排查时间浪费
- 在探索测试有效性的过程中，我们引入了精准测试的方案

什么是精准测试?

通过建立测试用例与业务方法的关联关系，在代码发生变化时，精准的推荐出需要运行的用例，进行测试执行与结果反馈。通过精准的圈定测试范围，可以带来效率和速度的双重收益。

精准测试的基本要素

- 测试用例（Case）与应用代码方法（CodeMethod）关联关系的建立。这种关联关系，我们定义为**基线**。
- 代码发生变更，根据**基线**中用例与应用代码方法的关联关系，准确推荐出**变化的方法**、**关联的测试用例**和**变化的测试用例**，并进行运行。

如何建立测试基线

- 基线的建立方式有多种，在阿里巴巴内部使用过以下两种方法去建立用例与代码的关联。
- 通过字节码注入的方式，埋入 trace 调用，并在调用中传入用例与业务方法的签名。通过采集 trace 的日志，拿到所有的测试用例与方法调用链路，建立起用例与方法的关联关系。
- 通过 AST 解析的方式。

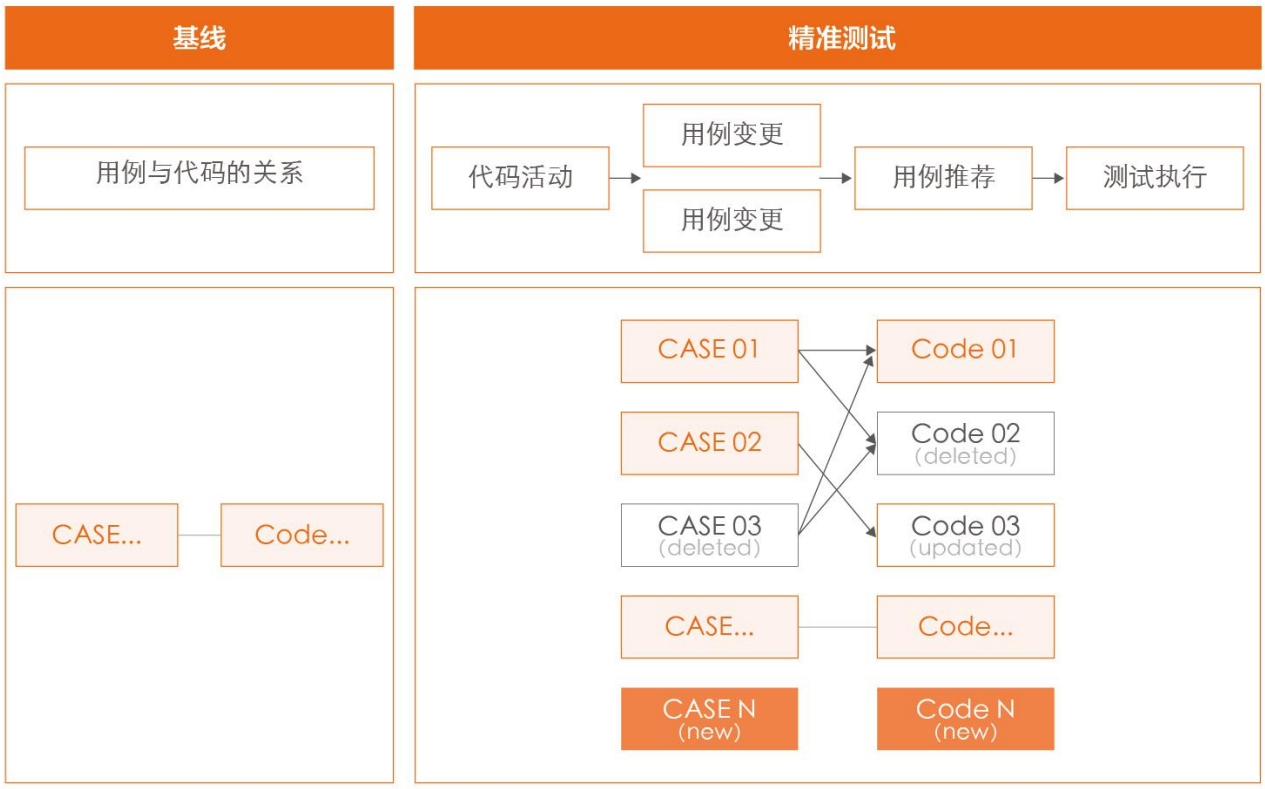
如何进行用例推荐

在代码发生变更后，会基于代码变更解析出变化的测试用例与变化的业务方法。方法的变化通常是**新增**、**删除**、**更新**。

用例的代码变更情况比较简单，所有新增和更新的用例都会纳入到回归范围。

被测应用代码变更，要分情况考虑：

- **新增**的方法：匹配不到任何用例，针对此次变更，开发或测试可能补充了测试用例，也可能未补充，需要进行提示。
- **更新**的方法：更新的方法分两种情况，第一种，更新的方法有关联的用例，需要推荐出来；第二种，更新的方法原本就没有关联任何用例，需要提示用户补充测试用例进行覆盖。
- **删除**的方法：删除的方法也分两种情况，第一种，该方法本身就不关联任何测试用例，不做任何提示；第二种，该方法关联了一些测试用例，那么考虑到方法有可能只是重构或者更名的情况，这部分关联的测试用例在没有被删除的情况下，也是需要进行回归的。



阿里云某核心云产品，通过使用精准测试的方案，一个迭代了一周左右的代码变更，从原先每次 CI 需要全量执行 3700+用例，到每次 CI 可以精准的执行变更影响的用例范围。速度提升了近一倍，测试范围缩小到不到原先的 1/6。





■ 提升测试有效性

我们希望编写和运行的测试用例能够有效的覆盖代码的逻辑，其中重要的一个着手点是测试覆盖率，通过测试覆盖率来暴露问题，并促进问题的解决。

01 测试覆盖率

测试覆盖率，在本文中，指的是代码的覆盖率。即行覆盖率，分支覆盖率等。

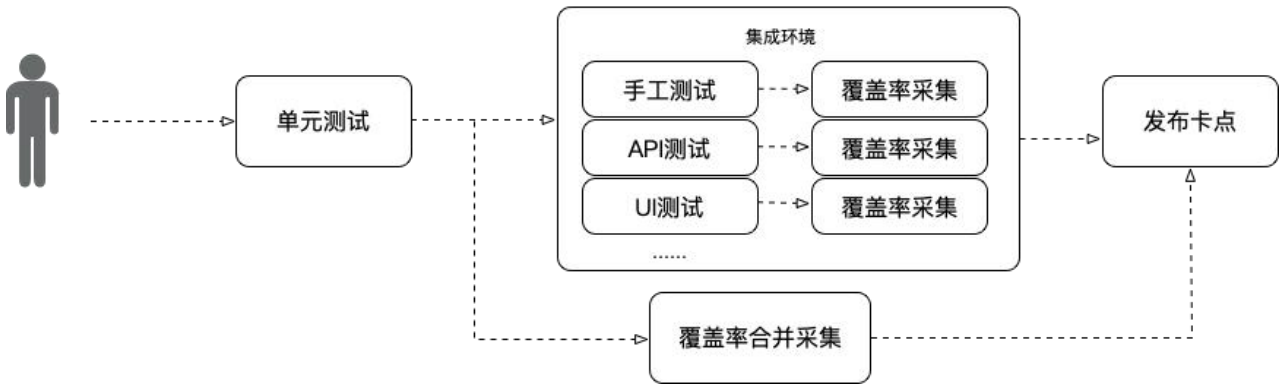
此外，本文中所有涉及覆盖率采集的示例都以 Java 为例。

02 如何收集完整的覆盖率

一个应用下通常存在多种不同类型的自动化测试集

- 单元测试
- 手工测试
- API 测试
- WebUI 测试
- 其他

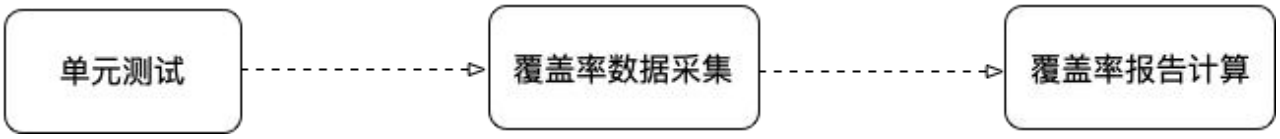
为了能够准确的反映一个应用的完整覆盖率，需要对上述多种自动化测试的结果进行聚合。在阿里，每一个测试的运行都会关联到相应的应用，从而可以对测试结果进行聚合。



为了能够收集所有类型的测试覆盖率，我们做了以下事情：

● 单元测试

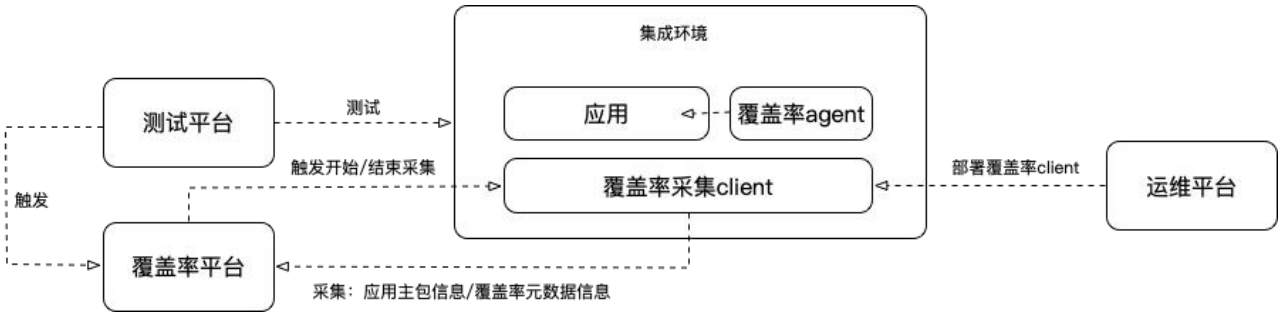
对于单元测试来说，覆盖率数据产生在单测执行的机器上，我们会根据执行机上的原始代码信息，编译后的 class 信息，单测执行后产生的覆盖率数据原始文件，以及变更的代码信息，计算出单元测试的覆盖率报告。



手工/自动化测试

我们实现了一个覆盖率采集客户端，和一个覆盖率采集/报告计算解析的覆盖率平台。通过运维平台将覆盖率采集客户端部署到应用的集成环境，在应用启动时会挂载一个 javaagent 进程。当我们在任意测试平台触发任意类型的自动化测试时，会通知覆盖率平台与覆盖率采集客户端进行交互，完成覆盖率计算原始数据的采集与解析。

另外，在进行发布卡点时，我们会合并相应的单测覆盖率形成完整的覆盖率报告。



测试覆盖率能给研发过程带来哪些价值

- 分析未覆盖部分的代码，从而反推在前期测试设计是否充分，没有覆盖到的代码是否是测试设计的盲点，为什么没有考虑到？需求/设计不够清晰，测试设计的理解有误，工程方法应用后的造成的策略性放弃等等，之后逐步补充测试用例。
- 代码覆盖率高不能说明代码质量高，但是反过来看，代码覆盖率低，代码质量不会高到哪里去，可以作为测试自我审视的重要工具之一。
- 分析变更代码的覆盖情况，从而保证对变更的测试充分，增强发布成功率与信心。

除了上述的“测试覆盖率”，还可以使用相同的技术来统计“线上业务对代码的覆盖情况”。也就是统计出来有哪些代码是被真正的线上业务所用到的，哪些是从来没有被调用到的。从而检测出程序中的废代码，可以逆向反推在代码设计中思维混乱点，提醒设计/开发人员理清代码逻辑关系，提升代码质量。

03 增量覆盖率

有了完整的测试覆盖率数据之后，就可以让它发挥作用了。但脱离了具体的场景，单独追求一个较高的测试覆盖率数值是没有意义的。如果一味的追求较高的覆盖率的数值，往往带来的的是用例的过度设计。如何健康有效的让项目的测试覆盖率稳步的提升，在阿里巴巴内部，我们更多的是采用关注增量覆盖率的方式来提升整体的测试覆盖率。

● 什么是增量覆盖率

增量覆盖率是指，某一次测试过程中，变化的代码的测试覆盖情况。

变化的代码=被测分支的代码与目标对比分支的 diff（通常目标对比分支是我们最终会合入的分支）。

增量覆盖率=变化的被覆盖的代码行/变化的代码行。

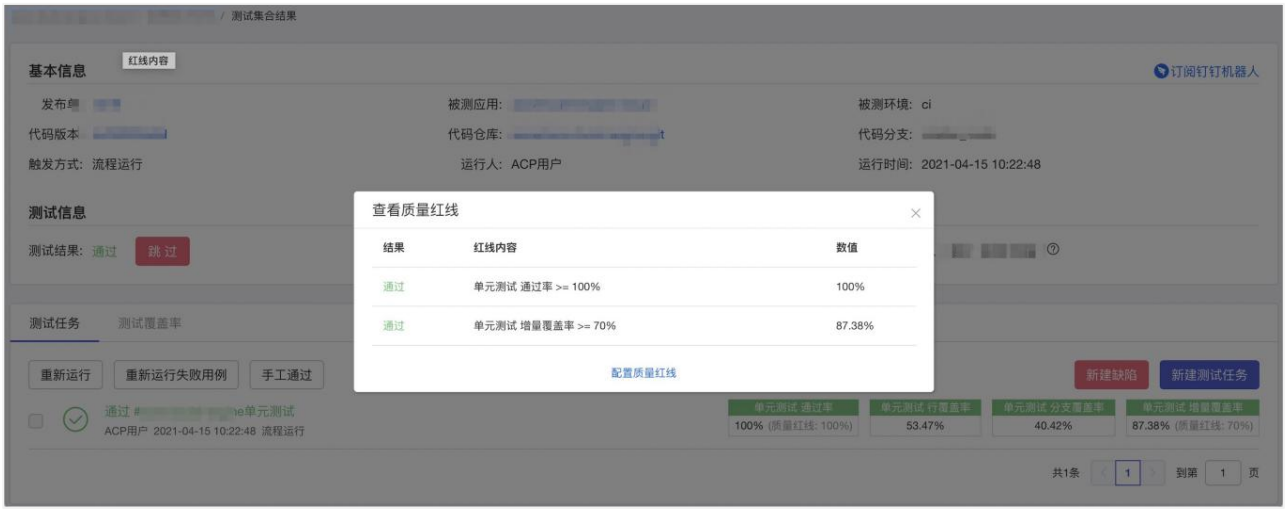


● 增量覆盖率的价值

- 发布之前是否存在漏测
- 针对漏测完善用例集
- 增强变更发布的成功率与发布信心
- 通过追求增量覆盖率进而提高被测应用的整体测试充分度

● 增量覆盖率的应用

在单元测试的时候，会有单元测试的增量覆盖率，在测试/预发等环境进行各种接口自动化/UI 自动化/流量回放/手工测试时，也会有增量覆盖率产生。当增量覆盖率与持续交付流水线进行结合时，能够有效的保障项目质量是在往好的趋势发展。



在阿里巴巴内部的实践中，我们通常会在 CICD 流水线中的关键阶段设置各类卡点，保障在多人协作的场景下，每个开发同学提交进入集成的代码经过充分的自测。在上线前的集成阶段，进入的代码在集成环境中被充分验证后，才会被允许发布上线。

通过增量覆盖率的反馈，开发/测试同学可以针对性的去补充各类测试用例，尽可能的保障在各阶段不存在测试遗漏。同时，在不断的完善测试集的情况下，项目的整体测试覆盖率也会得到健康有效的提升。

04 线上覆盖率

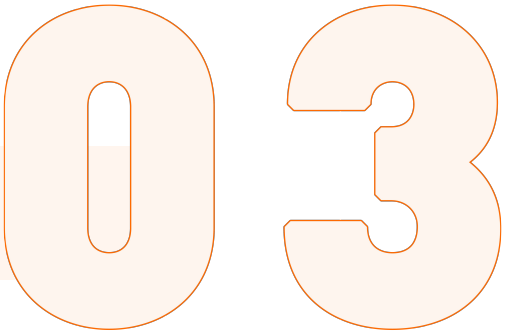
覆盖率的采集我们通常不会应用到线上环境中，但是如果将覆盖率采集放到线上环境，又能演化出应用瘦身的场景，帮助我们从另外一个角度提升效率。

通常在线业务的服务都会部署多个副本，为了减少风险，我们会在其中的少量副本上进行覆盖率采集。经过一个较长的采集周期后，会生成线上覆盖率报告。在这个时候可以认为被覆盖到的代码都是有效代码，而剩下的那些长时间没有流量覆盖的代码，需要谨慎的考虑删除/重构。通过这样的方式去精简我们的代码，从而降低维护成本。

在阿里巴巴内部，当我们决定对某些维护困难或者腐化明显的应用进行重构时，都会进行一段时间的线上覆盖率采集，并根据报告去指导我们进行代码重构，代码瘦身。

小结

分布式测试为测试速度插上了翅膀，精准测试有效的识别出了测试的范围，增量覆盖率又为测试的不断完备提供了有利的指引，线上覆盖率帮助我们有效的进行应用瘦身。充分利用好这些技术手段进行测试提效，可以让持续交付的过程更加的顺畅。



3.7 测试环境与路由

阿里巴巴内部的测试环境治理方案数年来不断演进，其中也经历过不少试错的过程。早期测试环境治理的迭代过程是比较漫长的，究其原因，是因为测试环境解决方案的更新换代牵扯面比较广。这里面不仅仅涉及全链路微服务的整体策略调整，而且涉及影响路由隔离能力的多种中间件协议（HTTP、RPC、消息等）的整体变更，一次测试环境解决方案的整体规模化更新落地时间需要以年计。

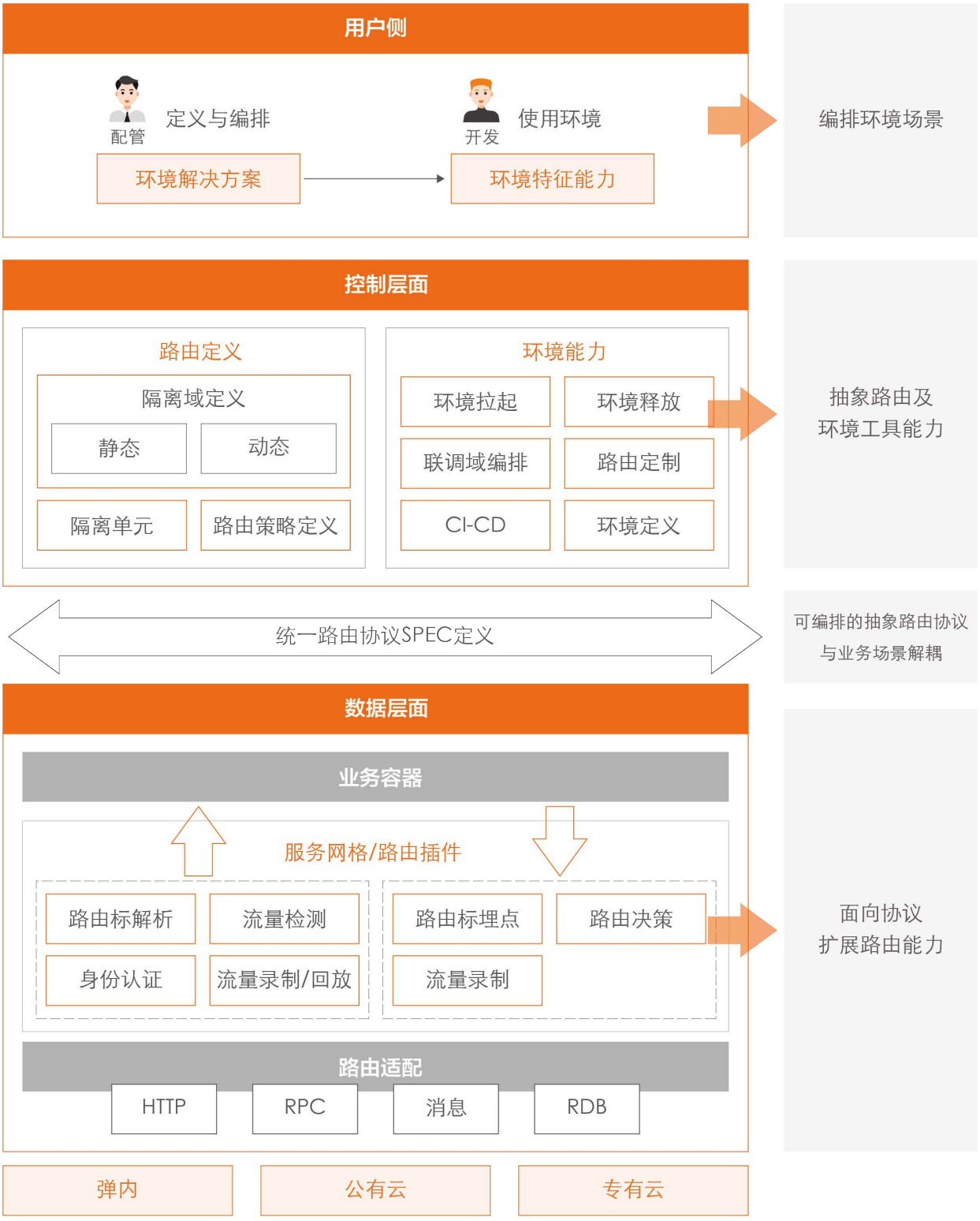
同时随着新技术发展给测试环境拉起带来便利性的红利，业务侧针对测试环境的使用也逐步向多职能、多样性的解决方案快速演化。

为跟上业务测试环境解决方案演进步伐，加快测试环境治理迭代速度同时减少试错成本，有必要进行测试环境流量路由领域抽象，以适配和满足各种使用场景的诉求。

解决方案

需要针对环境使用场景以及路由进行分层抽象，增加路由协议层定义，协议支持路由编排，并与业务场景解耦。数据层面与控制层面面向统一路由协议进行扩展和实现，数据层面提供路由能力的扩展（如 HTTP、RPC、消息、DB 等），控制层面提供环境和路由产品化工具，在用户侧提供面向配管的环境解决方案编排能力，以及提供面向开发者的环境使用能力。

环境路由平台


用户侧
配管
定义与编排
开发
使用环境
环境解决方案
环境特征能力
编排环境场景
控制层面
路由定义
隔离域定义
静态
动态
隔离单元
路由策略定义
环境能力
环境拉起
环境释放
联调域编排
路由定制
CI-CD
环境定义
抽象路由及
环境工具能力
统一路由协议SPEC定义
可编排的抽象路由协议
与业务场景解耦
数据层面
业务容器
服务网格/路由插件
路由标解析
流量检测
身份认证
流量录制/回放
路由标埋点
路由决策
流量录制
面向协议
扩展路由能力
路由适配
HTTP
RPC
消息
RDB
弹内
公有云
专有云

路由与测试环境

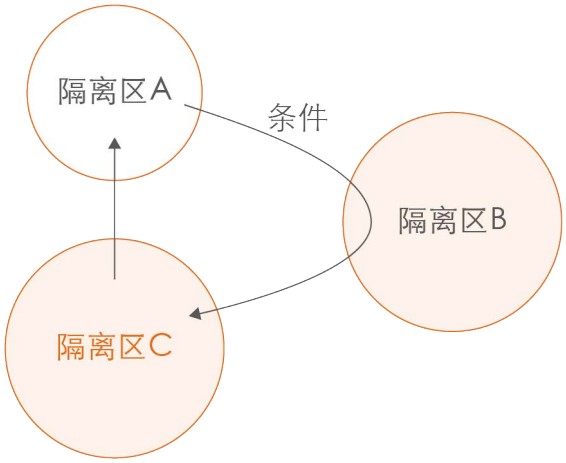
01 概念

- 隔离域：提供相同层面服务的集合，服务间调用优先选择在隔离域内进行，隔离域间默认路由隔离，可以从路由层面定义隔离域间的 fallback 降级调用策略。
- 隔离单元（环境）：隔离域内最小辨识单位，为隔离域内的某一应用的资源集合，通常 DevOps 平台面向开发者会使用某应用下环境来承载隔离单元。

02 路由如何领域建模

流量路由能力是测试场景可以并行的基础，能够为开发、测试等多角色开辟一块私有隔离域，自有服务调试流量在私有隔离域内部调用，隔离域之间流量隔离互不干扰。

接下来我们透过具体的业务使用场景，来看下什么是路由模型。



路由模型由隔离域、降级策略及降级条件三部分组成：

- 隔离域定义了隔离域内的隔离单元组成
- 隔离域降级策略定义了隔离域间的 fallback 机制
- 降级条件包括多种场景，如有隔离单元无实例、有实例无服务以及无隔离单元等

通过上述三者的组合，即可编排表述出面向具体业务场景的路由定义。

03 环境如何领域建模

● 环境分类

分析阿里巴巴以往的测试环境使用场景，环境可以归纳为两大类：静态环境和动态环境。

● 静态环境

静态环境类型有固定的隔离域，隔离域一经定义不再改变隔离域属性，通常只存在隔离单元的加入和删除，这种隔离域一般规模比较大，提供兜底的服务支撑，对稳定性有比较高的要求，需要开发者维护环境的稳定性，比如基础环境（微服务场景下提供支撑开发联调的兜底服务的环境）等。

静态环境的核心指标为稳定性 SLA，可以通过全链路的持续测试流量、监控、工单及故障机制来保障，其中静态环境的故障对比生产环境影响因子如下：

生产环境故障影响因子	静态环境故障影响因子
用户舆情、资损、下跌比例等	应用重要度（被依赖）、研发舆情、上游研发的活跃度、影响的联调流量等

● 动态环境

动态环境归属隔离域不稳定，经常会按照联调对象的不同发生聚合或者分裂，这种隔离域一般规模比较小，通常用作临时开发调试及测试使用，秉承“按需申请、用完即弃”的原则。

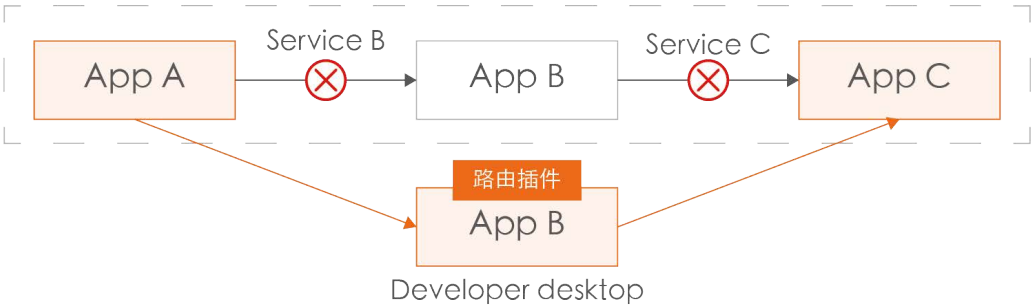
动态环境的核心指标为一键拉起能力、拉起的成功率和拉起的速度。

■ 如何实现端的一键接入

测试环境拉起多版本、多隔离域的同时，也给开发者的调用接入带来不小的代价，因此必须要让开发者能够从开发机或者端（PC 浏览器、APP 端）一键接入隔离域。

01 开发机接入测试

开发机通过对云上资源的替换标记，实现一键替换云端资源，将调用流量路由到本地，满足开发者本地联调的需求，联调完毕可以一键将开发机移出，联调流量重新回到云端资源。

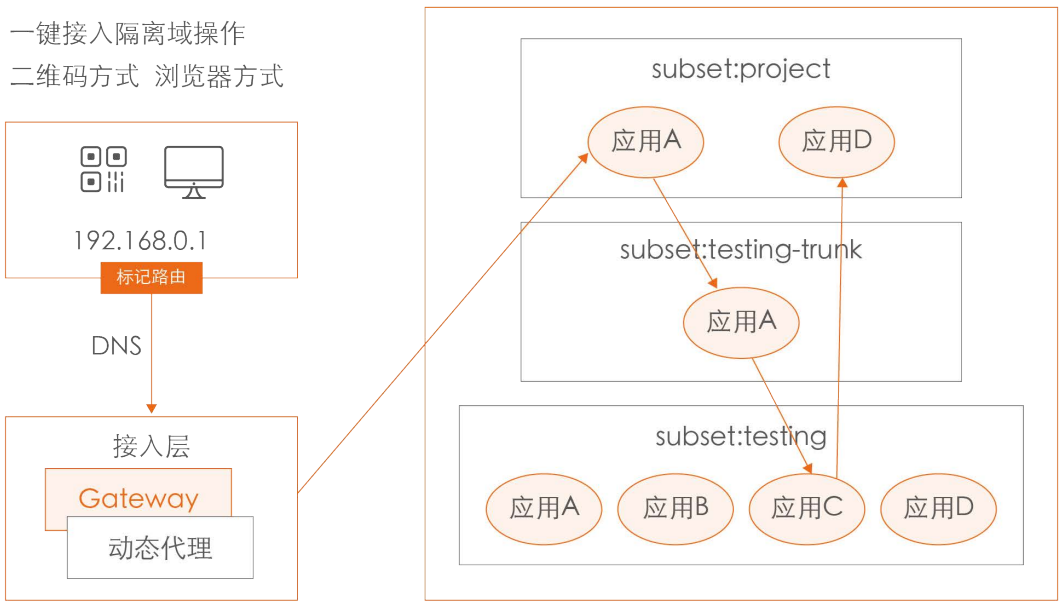


适用：办公网与测试环境互通
方式：通过路由规则定义，从隔离域移除原服务，替换成本地服务

02 端接入测试

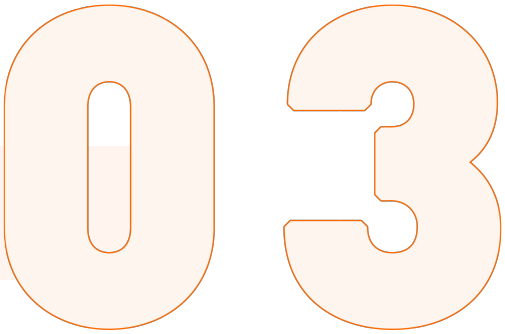
测试环境多隔离域场景下，实现端（浏览器、移动端）无配置化一键接入隔离域，摆脱大量如域名本地绑定 hosts 等配置化工作，提升研发效率。

移动端可以通过扫描目标隔离域二维码的方式标记路由，实现一键接入对应隔离域。PC 端浏览器可以通过打开对应隔离域网页地址的方式标记路由，实现一键接入对应隔离域。



总结

在阿里巴巴内部，随着业务规模和技术栈的拓展和更新，业务侧对测试环境的使用也逐步打破原固有模式，快速向多场景、多样化、多职能方向发展，如何能够跟上业务发展速度，及时满足业务侧对测试环境新场景的诉求，基于环境和路由模型的测试环境解决方案是解决问题的关键。



3.8 应用环境能力

每个软件都无法离开其依赖的运行环境。从代码的编写、调试、测试、上线、运维，每个步骤都离不开对应环境的支持。对于测试环境的争用、环境各阶段需要满足不同应用场景、软件质量的守护、成本与效率优化等诸多诉求，是环境治理和基于环境的变更交付流程规范化的原始需求。

软件行业对效率的要求是非常高的，如何能在现有条件下，提高开发测试的效率是一个很有意义的问题。而在这个问题中，环境，又是一个避不开的话题。如果每个开发都能在自己专属的环境里进行开发调试，不受外部人和物的因素干扰，自然会比较高效。然而，在微服务大行其道的今天，在同个软件模块多人多项目并行开发的情况下，为每个开发都分配一整套包含所有服务的环境，从硬件成本和维护成本上说，都不是一个明智的选择，有效的对环境进行隔离和编排，可以在保证开发效率的同时，优化硬件成本和维护成本。

软件开发通常需要经历多阶段的反复测试验证，是从无到有，从简到丰，从不稳定到稳定的一个过程。正是由于这样的特点，不同阶段中对应的环境特性也会不一样，例如最开始的开发环境，用途就是个人测试；线下集成环境，用于线下的集成测试；预发环境，用于上线前的回归测试及验收；正式环境，用于真正给用户提供服务等等。

用流程将不同的环境串联，能将原来松散凌乱的开发流程变得规范化，再加上合适的测试、验证及准入卡点等手段，实现满足准入条件（如质量标准等）才能交付下一阶段环境部署。通过系统流程化的方式引导开发者来完成安全、高效、可信的变更交付，避免交付过程无规则导致的混乱及安全隐患，同时可以让整个研发流程做到可视化、可追溯、可度量。

总而言之，基于应用环境的变更交付，需要具备两大能力：

1. 单应用多开发者并行开发互不干扰的测试环境隔离能力

2. 基于环境的变更交付过程规范化的流程管理能力

解决方案

在微服务架构的大背景下，服务的数量大大增加，使得原本大应用内部模块的复杂性转化为了多个小应用服务之间的调用复杂性。在实际业务场景下，一个整体业务通常会由多个小的应用服务组成，因此，在开发整体业务的过程中，通常涉及到上下游一系列的微服务应用的修改，并且在多个不同业务的开发过程中，会导致同个微服务应用有不同的服务版本，大大增加了微服务之间的联调复杂性，给开发测试过程带来了除开发业务本身外的额外成本。

如何减轻联调的系统成本，让研发专注于自身的业务，是环境治理亟待解决的问题。为了更好的支持研发工作，通过将环境进行专职分类、编排隔离域、交付过程规范化，帮助研发更高效的交付软件产品，阿里沉淀出了一系列在测试环境治理方面的最佳实践。这些实践包括：

01 代码变更在环境上的递进机制

环境分为两大类：线上环境和线下环境。线上环境是那些操作和数据会真正影响到实际用户服务的环境，例如预发、beta、灰度、生产等。线下环境是主要提供给研发进行开发测试的，目前按使用方式主要分为项目环境、集成环境、基础环境三大类，本篇着重介绍线下环境，其中：

1. 项目环境：单个应用的单个开发中特性独占的环境，不受其他开发中特性的影响，也不会影响到其他环境的使用。用户可以在这个环境上进行任意的占用调试或破坏性测试。

2. 集成环境：一个应用的多个开发中特性共享的环境。这个环境主要用来验证多开发中特性在集成以后是否会在业务上产生新的问题或是引入新的冲突。

3. 基础环境：提供上述项目环境和集成环境的服务依赖，通过在生产环境部署后立即部署基础环境，来保证基础环境提供的服务都是最新的生产环境运行版本，从而保障上述项目环境和集成环境能够稳定的进行开发测试。

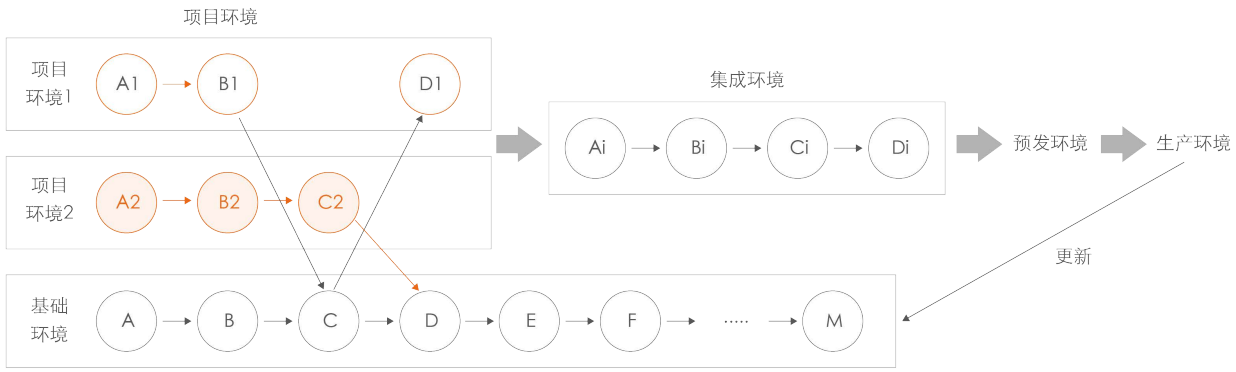


流程上，从拉取特性分支开始，自动分配项目环境，在特性分支线下测试基本完成后，将项目环境转为集成环境，同其他特性分支一起测试，在集成环境测试通过后，部署至线上环境，最后在特性分支合并至主干后，用最新的生产版本更新基础环境。

02 编排隔离域

随着业务的不断增长，当前的测试环境需要支持数万开发工程师的同时使用，对于某些核心业务，有数百个开发中的业务特性同时进行开发测试，而大部分的业务特性都涉及到多个不同微服务的修改，如何保证多个并行业务之间能相互独立，不会相互影响？解决方案是隔离。将同个业务特性的多个不同微服务的环境圈定为一个隔离域，保证相关调用在隔离域内进行，这样就能隔绝其他的业务特性提供的服务对这个业务特性开发测试的干扰，在用户看起来，这个业务特性仿佛拥有了一套完整的环境。

然而，在微服务大行其道的今天，一个业务特性的运行通常依赖着许许多多其他的服务，如果每个隔离域内都将这些服务全部部署作为支撑，是一件成本非常高的事情。我们的解决方案是共享。从路由维度出发，所有隔离域共享基础环境，而保留隔离域自身的项目和集成环境，来达到尽可能的复用公共服务的目的，大量减少了资源成本和环境维护成本。



如图，项目环境 1 的用户和项目环境 2 用户分别拉起隔离域进行联调，相互之间流量隔离互不干扰，隔离域不存在的服务都复用基础环境进行服务兜底，待特性开发分支经过集成、预发验证并发布生产环境后，会自动更新基础环境的基线版本，所以基础环境会持续保持跟生产环境相同的稳定版本，以提供稳定的支撑服务。

03 基础环境建设

微服务架构场景下从端发起的一次调用，会涉及到调用链路上多个应用提供的服务，但在实际的开发联调中，一条链路上只有少部分的应用需要变更，如果面向开发者需要拉起整个链路进行开发联调，那么会带来低效和成本的问题，所以其中非变更应用可以采用基础环境作为服务支撑。如何保证基础环境的稳定可用就是隔离环境建设的重点。为了保证基础环境的可用性，主要做了三方面的工作：

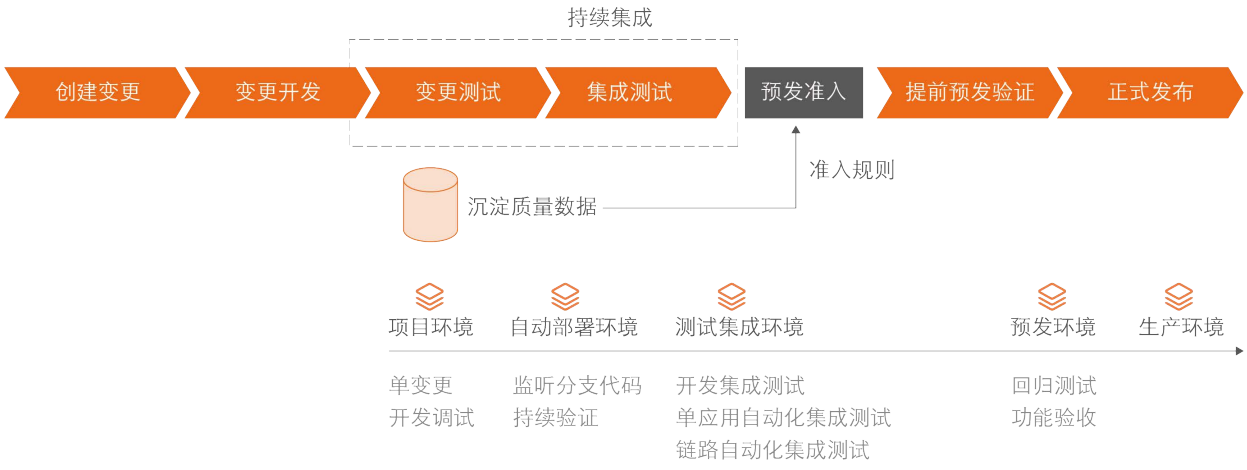
1. 降低接入基础环境成本：创建环境通常是一个比较让开发头疼的工作，通常涉及到上下游一系列配置工作，例如修改发布流程、增加 Dockerfile、下发环境隔离文件、准备测试域名、申请相应资源等，为了实现基础环境解决方案的落地，需要实现批量的基础环境搭建，过程中需要具备用户一键搭建基础环境及相应配套的能力，降低基础环境的接入成本。

2. 代码层面保证服务稳定：基础环境只部署最新的主干代码版本，而且通过系统流程保证每当线上部署完成，发布特性分支代码合入主干分支后，将基线版本自动部署到基础环境，用户无法直接部署开发中的特性到基础环境上，从代码版本管理层面来保证基础环境服务的稳定性。

3. 可持续流量与自愈、监控保障：为了保证基础环境稳定服务的可持续性，基础环境需要稳定的全链路测试流量及监控，实时监测基础环境的可用性，在发现基础环境服务不可用时，首先通过无人值守的系统手段自愈，如果还是不可用，通过自动工单机制通知并跟踪基础环境的恢复进程。

04 基于环境的交付流程

从拉取变更代码特性分支，到最终特性分支交付正式发布大致分为几个阶段：创建变更特性分支、变更特性分支功能开发、分支功能调试及自动化验证、多分支集成验证、准入卡点、提交预发验证、正式发布上线（如下图所示）。



其中预发准入卡点是为了保障达到一定质量要求的代码才能进入预发验收，把低质量的代码拦在测试环境持续验证，而自动部署环境则是为了减轻准入卡点给开发者带来的负担，通过自动化手段来对特性分支代码持续验证，收集并沉淀质量数据，为准入卡点提供判断依据。

● 自动部署环境

当用户通过系统创建特性分支时，会自动为用户的新创建分支申请一个项目环境，该项目环境的配置与基础环境相同，并自动进行资源分配、部署操作。同时，这个项目环境的调用和消息同其他环境相互隔离，它的服务并不会影响到其他环境的调用。

每当用户所创建的特性分支提交了新的代码，都会自动触发系统去进行一次部署及自动化测试验证，通过持续的变更提交+反馈，缩短特性分支变更缺陷的反馈周期，帮助开发尽早修复代码缺陷。

● 分支版本准入卡点

在研发交付的过程中，通过开放配置流程的能力，可以让业务方灵活的根据业务需要，配置所需的流程步骤，通过流水线步骤的自动推进，完成从测试到上线的整个交付过程。流水线是由一个个组件组成的，每个流水线都可以串联一到多个组件，在阿里巴巴的最佳实践中，在某些环境的部署组件后配置代码质量管控的组件，可以让环境在部署后，马上自动触发测试组件，来验证最新的部署版本是否符合质量要求，从而倒推出最新变化的代码是否符合质量要求。

使用中，为了保证线上环境的安全和稳定，在提交预发环境集成部署之前，会判断所要发布的分支的最新版本是否都通过了准入质量要求，没有达到质量要求的代码变更会被拦在测试环境继续修复验证以保障生产环境的安全。

05 环境与测试技术的结合

自动化测试一直作为有效的验证手段被广泛使用，然而，在实际使用过程中，用户对于自动化测试用例的诟病并不少，例如：

1. 测试用例本身有问题，而不是被测试的代码有问题，导致花费了大量时间排查后，在确定本身代码没有问题，才能反查测试用例的正确性，排查成本较高。

2. 破窗效应明显，一个开发的懒惰导致测试用例失败，其他开发人员并没有足够的动力将测试用例修复，同时由于他人的原因导致测试用例无法通过，也就降低了自身对测试用例的要求，几次下来就会导致测试用例的情况越来越糟糕。

3. 责任排查难，大多数测试用例只会告诉你这次失败还是成功，但是无法关联比较多次测试用例之间的代码变动情况，当测试用例失败时，用户并不知道上次测试用例成功到这次测试用例失败之间，变动的是哪些代码，带来高昂的定位成本。

针对这样的情况，我们需要从更高纬度的视角去关联代码版本与测试用例，主要分为两点：

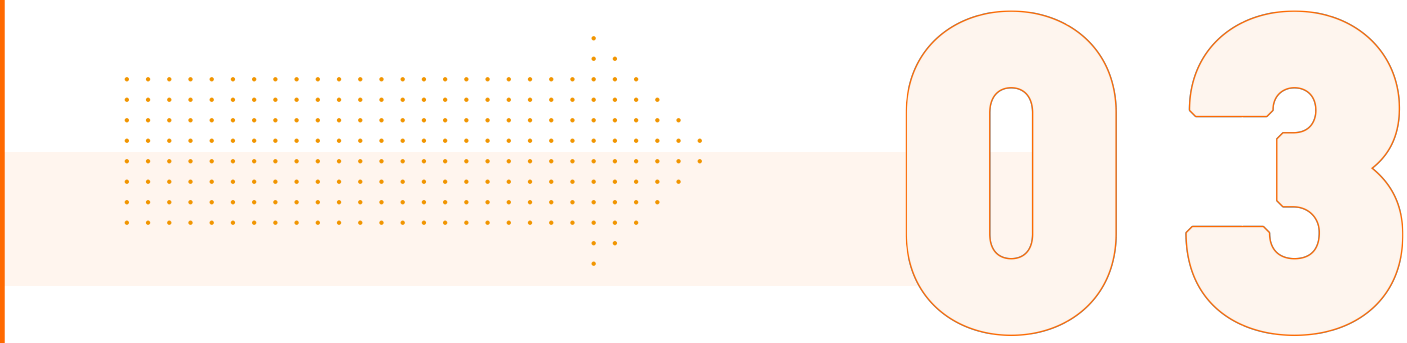
1. 常态化代码主干测试用例回归：在每天夜里系统自动根据主干代码创建应用环境并隔离、部署，然后跑对应的测试用例，由于是主干代码，因此受到的代码干扰较小，如果通过，表示该测试用例有效释放环境，如果不通过，则保留现场，并自动创建发送消息到用例负责人。通过常态化的跑测试，能对比上一天的结果来快速定位问题，并能保证“测试用例本身有问题”时能尽快得到解决。

2. 测试用例对比迅速定位代码：系统将每次跑测试用例的各个分支代码版本、集成分支代码版本都保存下来并进行关联，当测试用例失败时，能直接定位到上次这些分支测试通过时候的代码版本，将提交记录直接展示给用户，帮助用户定位测试用例失败的原因。

上述过程需要结合环境的一键拉起和流量隔离能力，通过这两点措施，可以让测试用例失败时做到结果可对比、原因可回溯，防止测试用例失败的破窗效应出现。

总结

应用环境解决方案并不仅仅是将应用的开发环境、基础环境搭建起来即可，还涉及到环境的稳定性如何保证，基于环境如何规范变更的流程，基于环境如何提升开发效率等等。环境治理需要站在更高的角度，综合看待上述问题，否则就会陷入环境问题年年治理、年年被吐槽的怪圈。



3.9 基于应用和变更的交付模式

让我们进入到阿里巴巴在交付阶段的一些实践，包括：

- 以应用和变更为核心的交付流程
- 基于变更的检查项和卡点
- 针对应用特征选择研发模式

■ 应用与变更

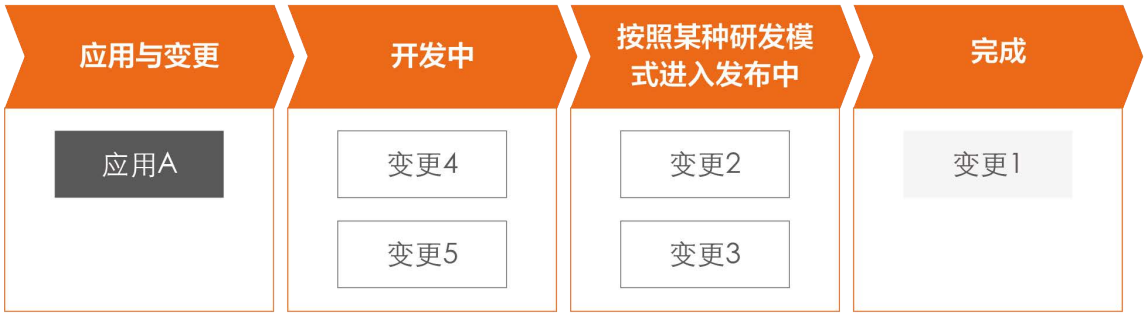
进入到具体的实践之前，我们先介绍下阿里巴巴的应用和变更模型。

应用，是研发活动的载体，包括了代码库、部署环境、配置项、变更管理、发布流水线、运维等一系列要素。整个需求的开发交付过程都可以在应用中完成。应用上可以设置不同的角色，这些角色信息在研发的各个环节中会起到作用。比如谁能发布线上环境，谁有权限修改测试配置等。

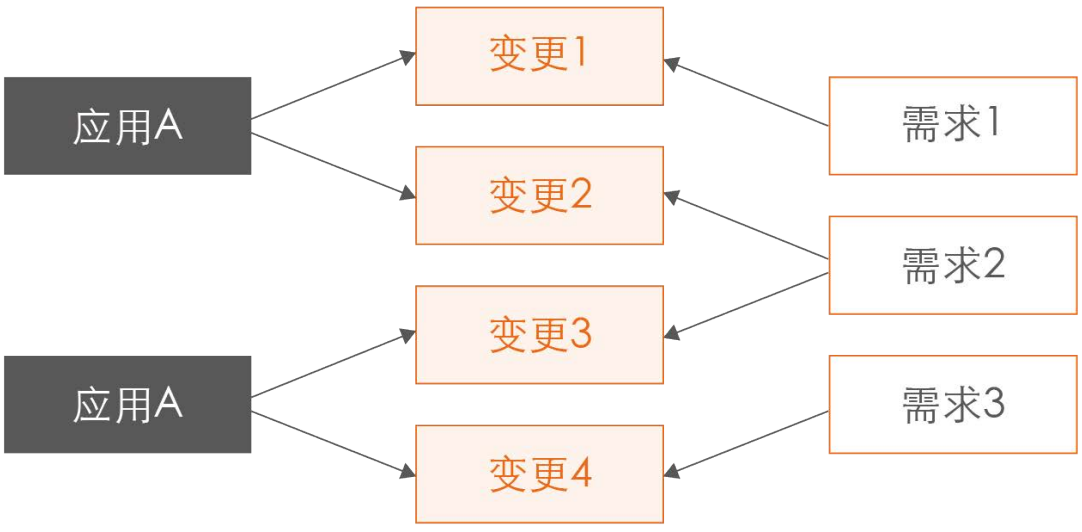


变更，作为应用的重要组成部分，是一个抽象的概念。所有对线上的行为的修改都可以称之为变更，变更属于某一个应用。变更中可以包含一个或者多个不同类型的变更内容，最常见的变更内容类型就是**代码变更**。

变更有相应的生命周期，一个典型的变更生命周期为：开发中->发布中->完成。当一个应用有多个变更同时在开发和发布时，需要有一定的协调机制，比如是在一个临时的集成分支上进行集成，还是在一个常驻的 develop 分支上进行集成。我们称这种分支协作模式为“研发模式”。



创建变更通常是为了实现需求和修复缺陷。一个需求可能需要修改多个应用，也就是需要在多个应用上创建变更，而简单的缺陷修复可能修改一个应用就可以解决问题，也就是一个变更。所以工作项（需求和 bug）和变更是一对多的关系。



我们从两个视角来看一下：

- 应用部署视角：每次应用部署上线会包含一个或者多个该应用的变更，可能涉及多个工作项。
- 工作项发布视角：一个工作项的发布可能需要多个应用的先后部署。
- 这两个视角是有交叉的。作为平台和工具，应该更关注哪个视角呢?
- 如果偏应用，则要完成一个工作项的发布，开发团队需要自行协调多个应用的部署顺序，会产生一定的沟通成本。
- 如果偏工作项，则会出现一个工作项的一组变更在部署某个验收环境，其他的工作项的变更需要等待，影响了整体交付吞吐率。
- 如果想要兼顾这两个视角，则不可避免的需要使用大版本制，或者叫火车制，拉长了单工作项的交付周期。

阿里巴巴的 DevOps 工具选择偏应用的视角，主要保证单个应用的部署上线流程和质量，将部署的节奏交给开发团队灵活安排。

比如一个工作项的发布涉及三个应用的三个变更，这个工作项的开发可以选择在周一部署第一个变更，周二部署第二个变更，而这两个变更不会造成任何线上可见的修改；到了周四，产品希望该功能上线时，再部署第三个变更，需求正式发布。这种将部署应用行为与发布功能行为分离的模式，可以降低集中部署带来的风险。当然能够这样做的前提是需要有比较完善的自动化质量保障及卡点机制。

■ 基于变更的检查项和卡点

变更是交付的载体，因此可以通过在变更上承载检查项来保证交付质量。最常见的检查项包括：

- 是否通过代码评审
- 是否通过安全扫描
- 是否通过单元测试/UI 测试
- 基于这些检查项会进行质量卡点，一般在如下的生命周期节点上：
- 从开发中进入发布中：需要满足一定的质量标准（比如单元测试通过率，覆盖率等）之后，才允许进入到发布中的状态，能进入测试环境进行验证。
- 发布到某个环境上：这时变更已经进入了集成流水线，完成了构建等，但如果要继续部署到某个环境，需要满足更多的验证条件。比如安全审核需要通过，才能发布正式等。

至于在哪些节点设置哪些卡点，应用负责人可以自行决定。多个检查项建议尽早同时开始，以提高效率。

从另一个角度来看，发布平台也可以对所有应用进行全局的卡点设置，针对某些卡点自动开启，且不允许取消。比如扫描到你的代码中依赖了有安全问题的 JAR 包，则要求必须进行修复，否则无法部署上线。这样就可以控制一些全局性的风险。

变更和应用中的发布流水线，提供了一个进行检查和卡点的框架。具体的检查工作由其它的专门的平台提供，比如测试平台、安全扫描平台、与特定业务相关的合规扫描等。

研发模式

如前文提到，为了能够让一个应用的多个变更协同开发和发布，我们需要采用不同的研发模式。研发模式的主要差别在分支合并策略，目前阿里巴巴主要采用下面三种研发模式：

- 主干模式
- Aone-Flow 分支模式
- Git Flow 模式

应用有大有小，在其上进行需求开发的并行度也不相同。

阿里巴巴有大量的的应用没有并行进行的变更发布。这类应用适合使用主干模式或者 Git Flow 模式。也有一些应用变更的并发度比较高，会出现一些发布节奏不同的问题。业界一般使用功能开关的方式解决这个问题。阿里内部采用名为 Aone-Flow 的分支模式来解决。

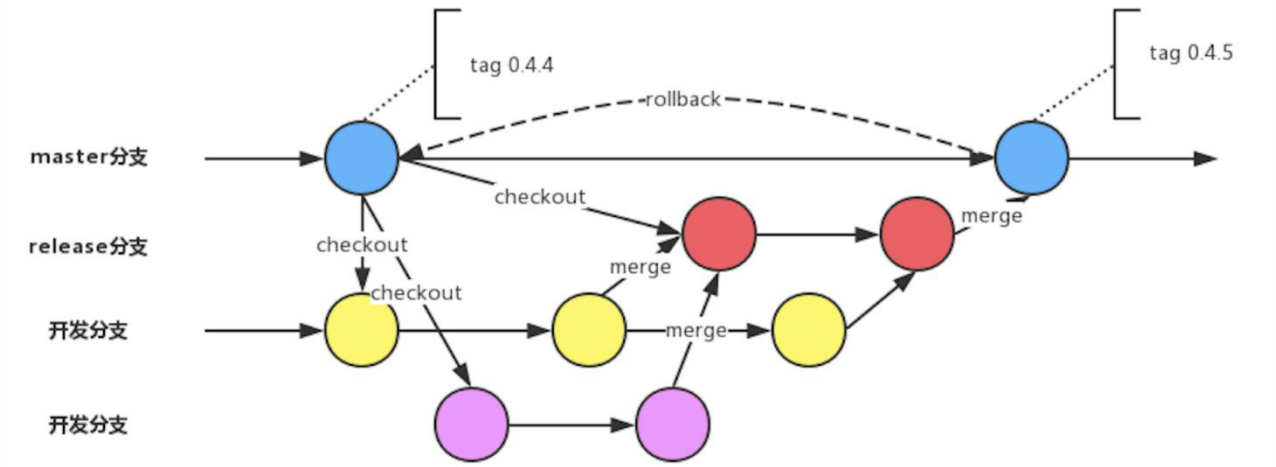
在 Aone-Flow 的分支模式下，开发人员的典型工作流程为：

1. 提交变更，Aone-Flow 会创建一个集成分支，或者复用已有分支，自动将你的变更合并到这个集成分支。

2. 当你发现你的变更有问题不能发布，可以“退出发布”，Aone-Flow 会自动创建一个新的集成分支，把剩余的需要继续发布的变更再合并进去。

3. 当集成分支完成正式部署之后，会合并到 master 主干上。这个集成分支上的变更都会被标记“已完成”，并打上 Git Tag。

4. 当需要回滚时，可以根据系统的记录同时将线上的版本和代码一起回滚掉（git revert）。避免出现无意把有问题的 master 代码发布上线的情况。



关于 Aone-Flow 的详细描述可以参看：https://www.infoq.cn/article/EaC4c6yiJrzZ_Gtaf9Ne

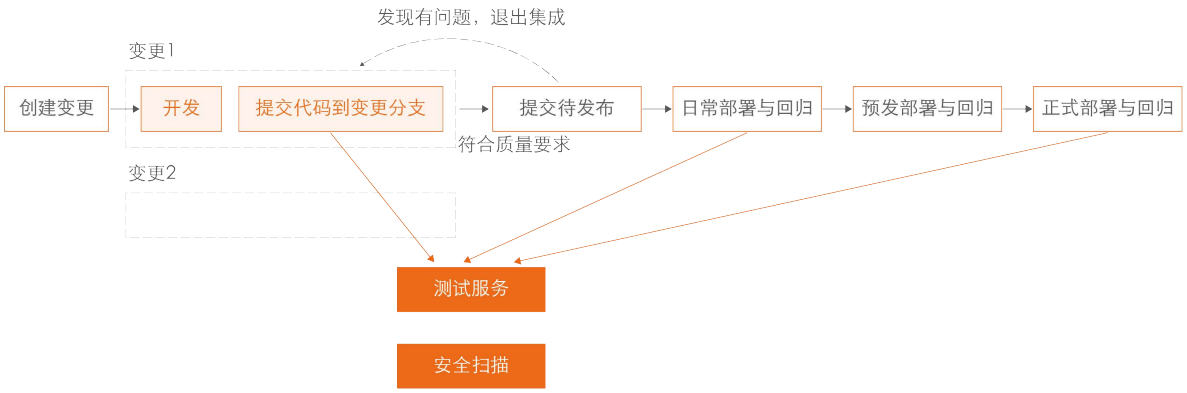
这里仅就主干模式和 Aone-Flow 进行一个简单的对比。

	主干模式/Git Flow 模式	Aone-Flow
对并行功能开发的支持	功能开关，成本高	进入/退出 发布，有可能出现重复解决分支之间的冲突的问题
对重构的友好程度	友好，修改了之后 push 上去即可	重构容易产生冲突，在 Aone-Flow 中会出现重复解决冲突的问题。解决方案就是尽快的把重构的变更发布到线上，这样才能合并到 master
对“快速上线”的友好程度	通过自动化测试的代码就会合并到公共分支（比如 master）。当需要进行发布时候，就会把这些功能都发布上去，如果自动化测试不能给你足够的信心，那么发布的风险就会提升，从而可能造成了大家不愿意去频繁发布，或者成为一个火车版本模式。	由于发布时候可以选择变更，每个变更可以不受其他变更的约束，单独尽快进行发布上线，发布风险比较小。 另一方面，前面提到的“重复解决分支冲突”问题也会督促开发者尽快把变更发布上线。

可以看到主干模式和 Aone-Flow 各有利弊。在阿里巴巴，为了能够更加快速独立的交付特性，有一多半的团队采用了 Aone-Flow。

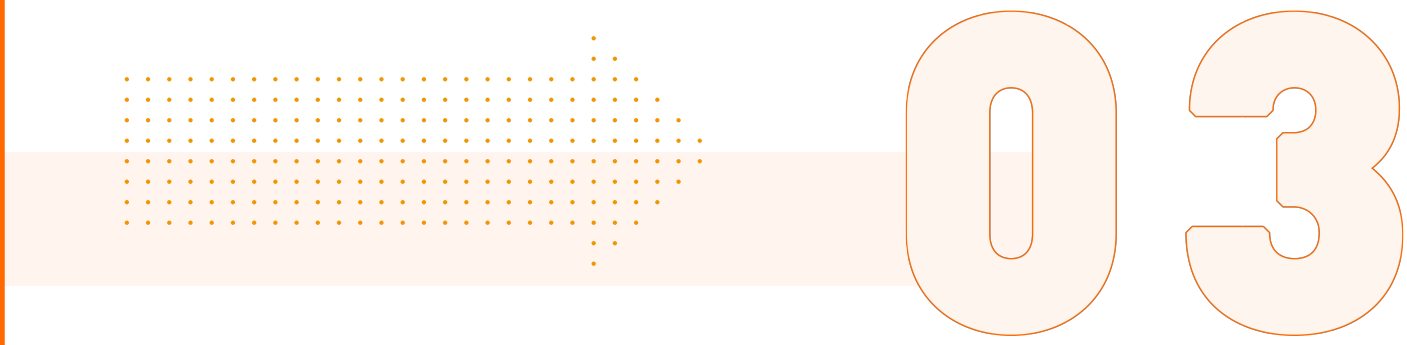
总结

了解了上述的过程，以 Aone-Flow 为例，我们对一个需求的上线过程要经过的阶段进行一个图解：



这里只画了日常、预发和正式三个环境，在实际的流程中，还会有灰度，安全生产环境等，来尽量避免出现线上问题和故障。

另外在变更的开发阶段，阿里内部还采用了一些隔离测试环境的技术来帮助开发者进行特性级别的隔离测试联调，详见前序的隔离测试环境章节。通过上述的管控和流程，我们就得到一个比较安全的需求交付流程。



3.10 提升构建的效率

构建是将源码变成制品的过程。构建包括编译，但不等同于编译。即使对于不需要编译的解释型语言，也要构建成一个压缩包或 Docker 镜像再去部署。无论是在开发阶段还是 CICD 阶段，都离不开构建过程，构建的质量和效率对持续交付影响很大。影响构建效率的因素，包括源码以及构建的依赖。

背景

在阿里巴巴，Java 是被最多人使用的编程语言。在 Java 的构建工具中，由于迁移成本，生态等原因，Maven 一直是服务端应用的最主要构建工具。

Maven 构建性能问题主要有二方面原因。一是外因，从应用角度看，一些 Java 应用历史悠久，依赖不断增加，并因为清理有风险导致累积太多，典型应用有 3000 个 Jar 包依赖，而且这些应用研发人员多，且是跨团队的，导致依赖管理成本高，依赖变得复杂混乱。二是内因，从 Maven 本身来看，它对复杂依赖的处理考虑欠妥。

对于外因，需要业务团队建立依赖梳理管理机制。针对内因，我们对 Maven 进行了重新实现，推出了 AMaven。

另一方面，阿里巴巴有 15%的 C/C++应用。C/C++应用与 Java 的最大区别是：Java 应用构建次数频繁，但每次构建时间短；C/C++应用构建次数少，但一次构建时间长。如某些软件的构建长达 10 多个小时。所以 C/C++开发同学的痛点，除了构建慢外，最不能忍受的还是等了 10 多个小时，最后几分钟居然构建失败了。

那为什么 C/C++构建慢且容易失败呢?

原因主要有两个：

1. 现有框架无法保证 C/C++编译和链接的严谨性，导致编译结果不确定，运行时也不稳定。举个例子，在编译阶段，同名的头文件在不同时间或不同机器上会可能内容不同，所以编译容易失败。

2. 公司内编译框架众多，有 scons，cmake 等，造成团队之间业务转接成本高，编译问题排查成本高。从平台角度看，也无法触达用户真正的编译逻辑，无法统一性能调优，更进一步无法统一升级底层编译器 gcc，即无法享受新技术红利。所以构建慢。

相比 AMaven 是重新实现底层构建工具的 Maven，在 C/C++领域我们主要是建设上层的编译框架，推出了 Alimake，因为底层的 make 已经很优秀了。

接下来，我们具体来看看 AMaven 和 Alimake。

■ 方案

01 AMaven

Maven 构建带来的性能问题，会严重影响持续交付的效率。主要体现在以下几点：

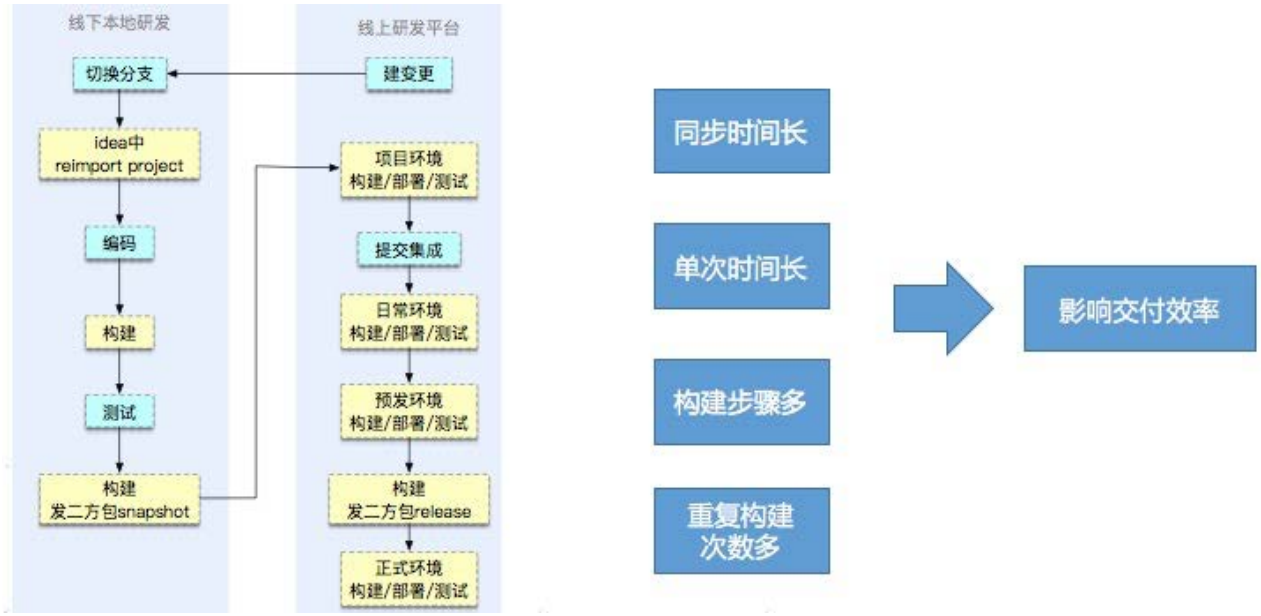
1. 在 idea 中同步时间长，如典型应用需要 10 分钟左右。

2. 单次编译时间长，如典型应用也需要 10 分钟左右。

3. 构建步骤多，在一次交付过程中，不同环境都要构建。

4. 在同一环境中往往会构建多次。

同时，构建性能问题也影响了开发同学的成就感。只写了几行代码，但一刷新工程，一本地编译，就一个上午过去了。一天下来，写代码的时间没等编译的时间多。



构建问题影响软件交付效率

一线研发同学，他的研发工作远不止体现在研发协同平台上的操作，在一个分支能进入集成前，有大部分研发工作都是在线下本地完成的。所以线下本地的提效也很重要。从新的 Java 构建工具 AMaven 开始，我们也将提效的视角范围从线上研发协同平台延伸到了一线研发本地。

基于 Maven 协议，遵循缓存，增量等思想，重新实现了一个工具：AMaven。同时在使用 AMaven 过程中，为保证它构建结果的准确性，在后台也会使用 Apache Maven 进行构建，并比较二者编译结果。同时又是基于 Maven 协议，对用户透明 ，从而做到"无成本，无风险"。



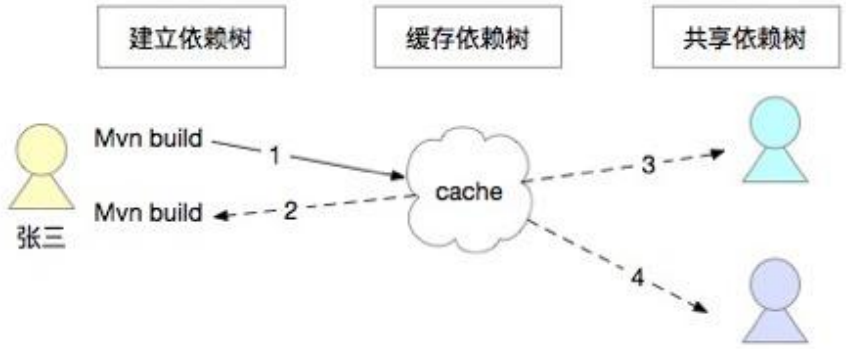
无成本，无风险



结果对比，保证无风险

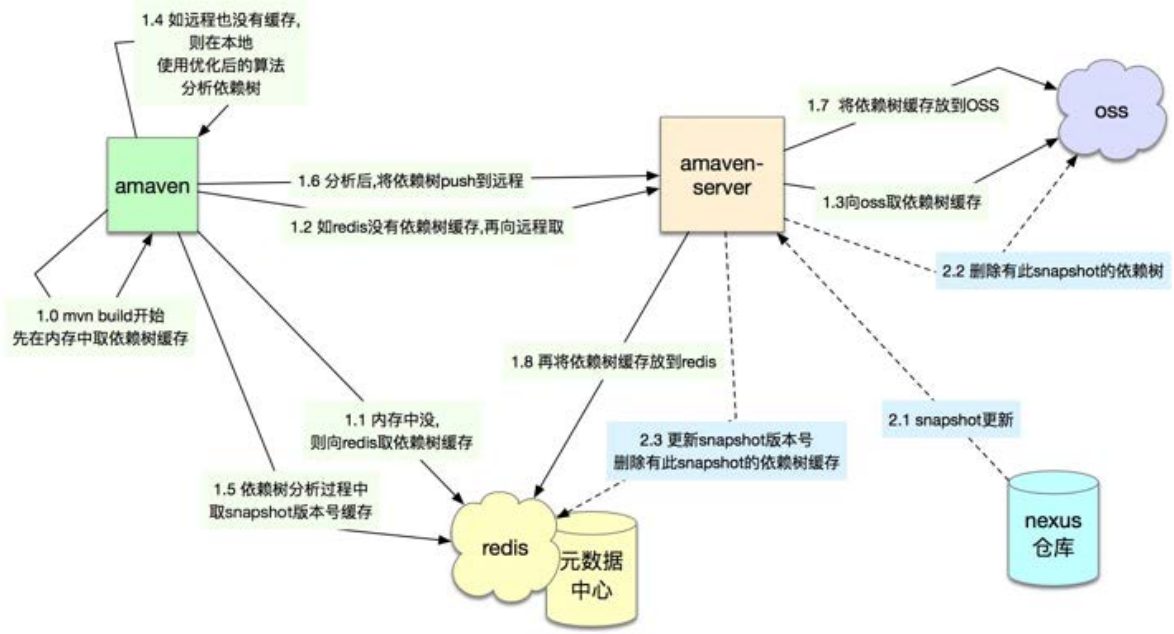
AMaven 对症下药，主要通过建立依赖树，缓存依赖树，共享依赖树来解决依赖复杂问题。一个依赖树缓存是否有效，除了与源码中的 pom 文件有关，还与依赖的 snapshot 变化等有关。当 snapshot 变化时，依赖树缓存就会失效，而需要重新生成依赖树。为提高依赖树的生成效率，AMaven 还通过依赖遍历算法对依赖树生成进行了优化，并开放了遍历深度参数让用户来微调。

这些依赖树信息同时也会保存为制品的重要元数据，在"制品元数据"章节中会有详述。



依赖缓存与共享

除了利用缓存，增量等思想，AMaven 还做了 C/S 化。即将部分计算能力移到 server 端。使用 AMaven 后，一次构建的流程变成如下图所示:



一次典型的 AMaven 构建

AMaven 还增加了循环依赖检测，动态执行插件等能力，虽不能直接提升构建速度，但加快了研发同学对依赖等构建问题的排查速度。

我们再来看看 AMaven 给用户带来的收益与效果。从线上 CICD 平台数据来看：AMaven 实现了 Java 秒级构建，阿里巴巴的 Java 构建中有 44%在 30 秒内完成。其次，从依赖庞大的典型应用来看，提效可达 10 倍。

从线下研发本地数据来看：AMaven 无论是在命令行还是在 IDEA 中使用，都能将构建耗时降至 50%。特别是在 IDEA 中切换分支后刷新工程时，最快能在 10 秒内完成。

02 Alimake

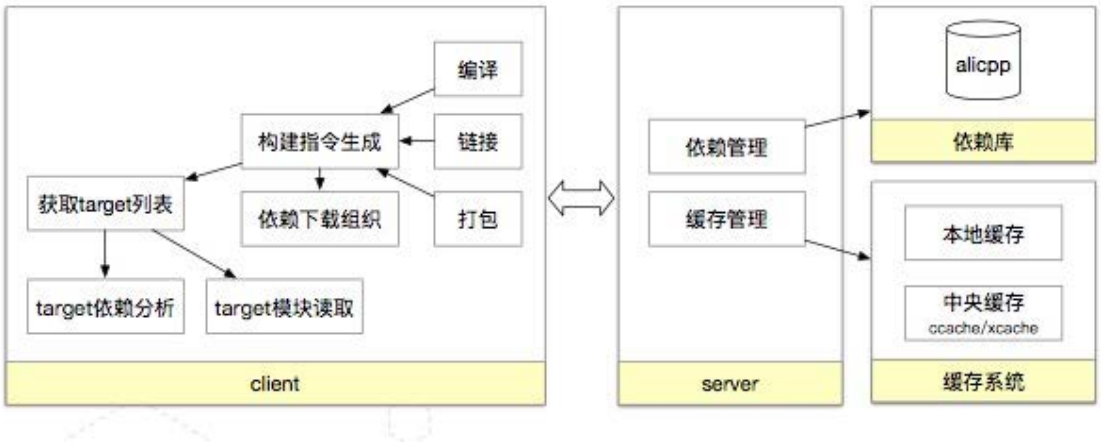
Alimake 主要从两方面入手。

1. 从"规范"入手。首先，建立全新的严格的配置文件 target，所有编译入参必须严格清晰的定义，它会被翻译成严谨的 makefile，它也同时在一定程度上培养了工程师严谨自律的文化。接着，建立了一个全新的仓库 alicpp，它统一存放了原本在编译机器上的依赖，从而保证编译的环境无关性，保障编译结果的强一致性。

2. 从"效率"入手。它不但集成了业界优秀技术 ccache 和 distcc，而且还自研了分布式链接等技术。因为 Alimake 统一了入口，所以它不但能让专家经验规模化，统一调优编译参数，而且还能将升级编译器 gcc 机制化，让我们可以随心所

欲，一直跟上编译器的进步。

Alimake 的架构思想与 AMaven 类似，也是将部分计算能力移到 server 端。利用 server 端一来能减轻 client 端的资源消耗，解决 client 端由于硬件及配置带来的性能瓶颈，二来也能实现资源共享，如依赖缓存，中间产物缓存。

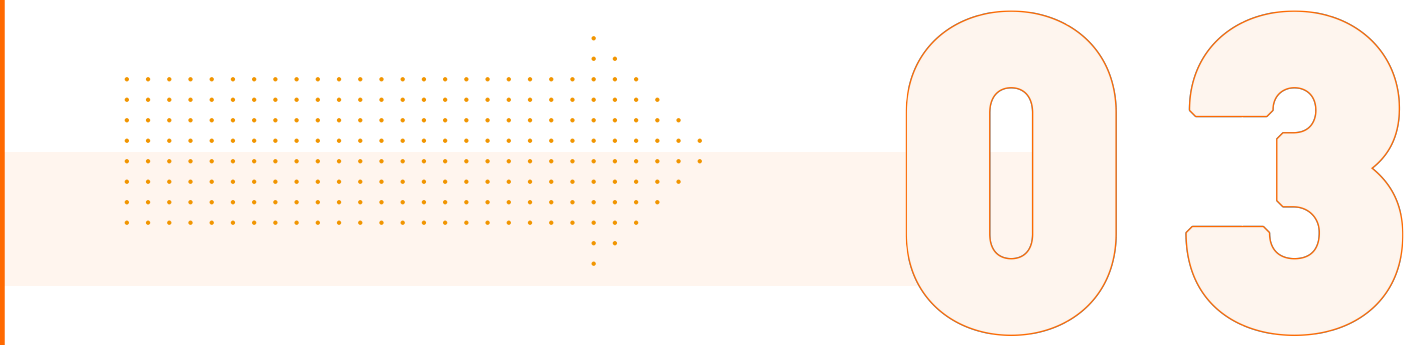


Alimake 架构

Alimake 覆盖阿里巴巴多个产品，包括：钉钉、阿里云存储、OSS、盘古、伏羲、蚂蚁人工智能等，平均提升构建效率 30%，最优情况下可以提效 70%。

小结

为提高构建效率，我们从空间（产物大小），及时间（通过工具来提升构建速度）二方面进行了优化。除此之外，还利用构建过程中产生的原生数据（如依赖关系），赋能于持续交付与安全生产等方面，以实现高效的可信的发布。



3.11 基于制品元数据提升交付效率

为保证软件交付的质量，我们对交付物有功能和性能上的要求。这些要求体现在交付过程中产生的数据上，包括：代码评审数据、安全扫描数据、回归测试结果等。这些数据以交付物（制品）为载体。我们把这些数据称作制品的元数据。

■ 什么是元数据？

元数据指一经产生就不会变化的数据。元数据是由系统产生，具有不可篡改和可回溯的特点，因而成为发布过程中的必要基础数据。

■ 元数据为何重要？

为说明元数据的重要性，先举个例子。阿里中台应用在架构上依赖很多业务团队的二方包，这些二方包质量往往难以把控。那怎么来解决呢？

一种改进方法就是从单个应用维度，到应用依赖树维度更”全景”更”立体”展示数据。以代码评审为例，在中台应用的制品中，包含很多业务团队开发的二方包。而在评审中台应用时，只会看到 pom 文件中的二方包的版本号变了，看不到具体的代码变化。对于评审者，需要看到这些版本号背后的代码变化，以及与这些代码变化相关的信息，如相关的需求、有没有通过代码检测、单元测试结果等。

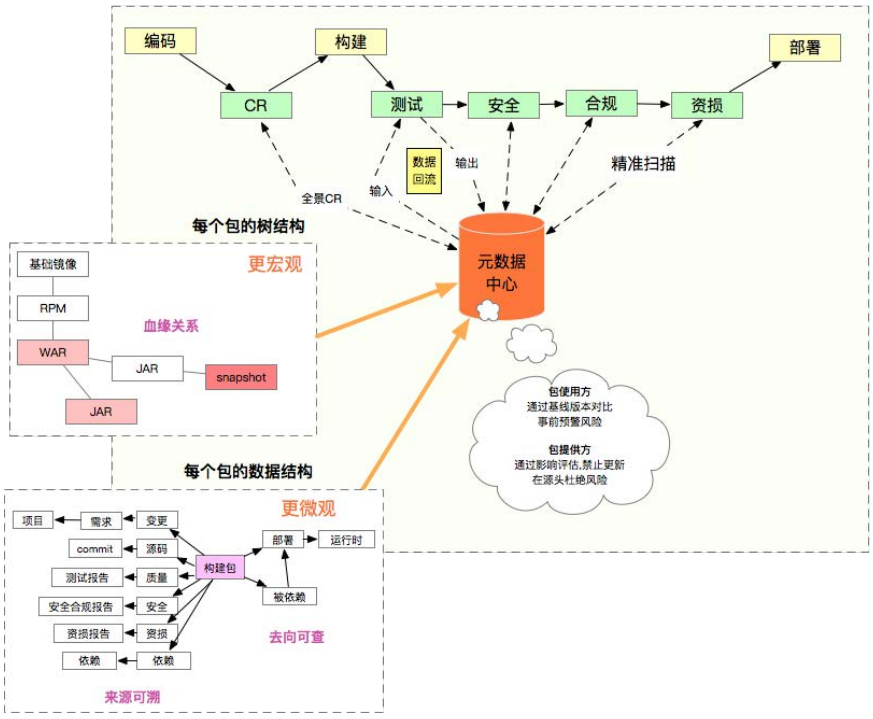
一个应用运行时的依赖大部分在构建时就决定了。运行时问题很多是由依赖引起的。让构建依赖(树)产生新价值，从而实现风险左移。

元数据主要有哪些?

除了在构建阶段取到的依赖树，我们还有其它数据，如测试产生的质量数据，安全扫描产生的安全数据。这些数据我们都会存放到"元数据中心"，再通过"管控策略中心"，利用这些数据对交付过程做自动化的卡点。这一流程通常包括可见、可控、可信三个阶段。

- "可见"是给用户展现元数据中心的数据，给用户透出交付过程的风险与瓶颈。
- "可控"是给用户根据看到的问题后，再设置规则来实现自动化检测与门禁。
- "可信"则是结合"合规安全扫描服务"与"制品仓库"能力，完成数据与规则的融合，实现交付安全。

从可见到可控，再到可信，最后达到提升交付效率的目标。与此同时，元数据与规则不断演进，慢慢沉淀成知识库，成为公司最重要的资产之一。



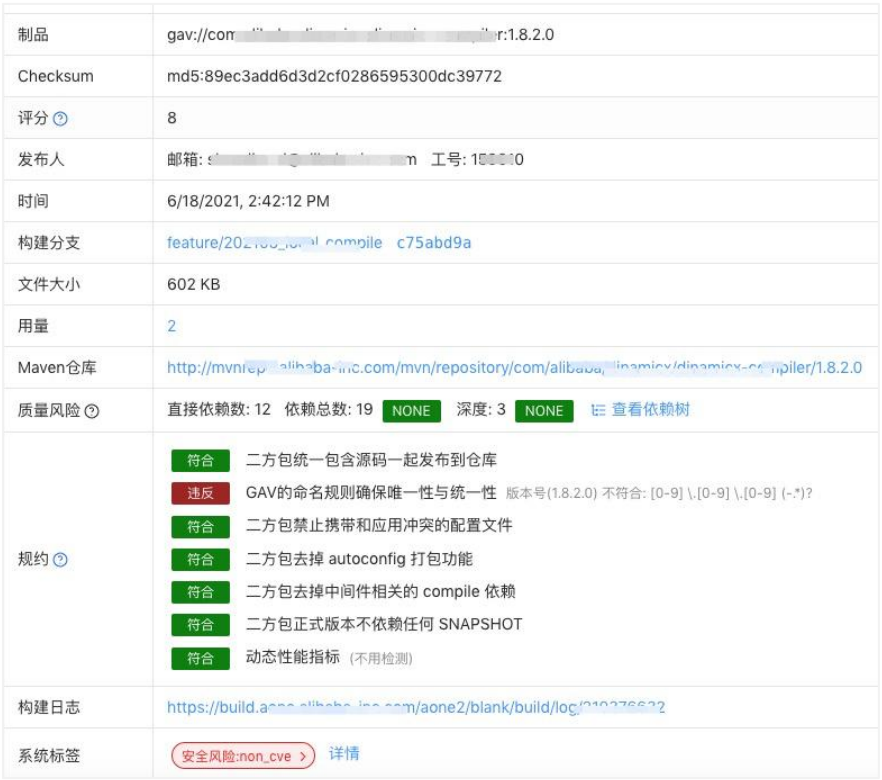
以元数据为基础实现可信发布

上图展示了可信发布的架构，可以看到可信发布以元数据为中心，基于制品，持续生产和消费元数据，配合各类自动化的门禁规则，做到持续、可靠、安全地发布。

元数据分为两类：包本身的元数据、包的血缘关系数据。

01 包本身的元数据

包本身的元数据包括包的基本信息，如：构建信息、质量数据、安全扫描数据等。除了这些基本信息外，还包括通过元数据协议写入的三方数据。



元数据详情

评分体系

一个制品，如一本书或一部电影一样，可以评分。以二方库 jar 包为例，评分可以指导 jar 包的"使用方"引用最佳质量版本的 jar 包。

评分包括：1. 系统评分，2. 用户评分。评分是针对制品的某个版本的。

其中，系统评分满分 10 分，主要根据以下几点来判断:

1. 是否符合代码规约，每有一条不满足就扣一分；

2. 是否符合核心指标，每有一条不满足就扣一分；

3. 是否通过官方平台发布，如不是，直接扣分；

4. 是否有安全漏洞，如有，则扣分；

5. 运行时的动态指标，如启动耗时，启动时内存消耗等；

6. 业务测试用例指标，如单元测试覆盖率/成功率；

7. 被引用数，一定时间段被越多的应用使用，会加分。

用户评分是开发者对这个制品的评价，某些制品系统评分很高，但是接口设计不合理，依赖多，使用方可以给较低的用户评分，推动制品的持续改进。

02 血缘关系的元数据

一个大制品，是由多个小制品组成的。制品与制品存在依赖与组合等多种关系。制品的血缘关系，也就是制品的依赖数，评分高的依赖版本会被推荐使用。



自动推荐版本升级

■ 元数据如何有效使用？

元数据除了在交付流程中用于各节点的准入与准出外，还可以帮助进行制品治理。制品治理基于元数据的“洞察”和“态势”两大能力。

- 洞察：告诉团队主管如何治理，帮助团队主管找到哪些制品，或哪些自己的子团队需要优先治理。
- 态势：经过治理后，制品质量的趋势是什么样的，帮助团队主管看治理力度与治理效果。

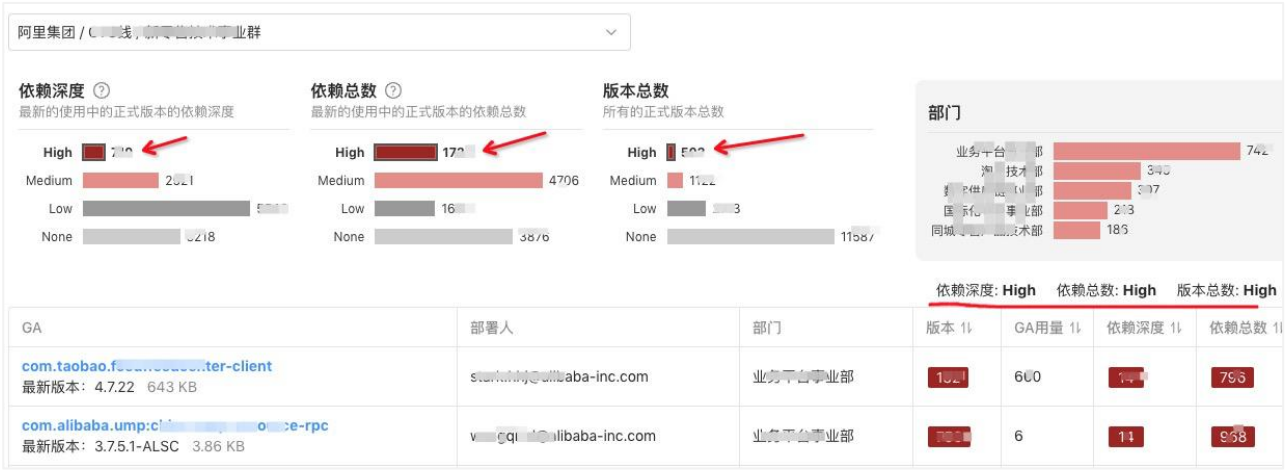
以 jar 包(二方包)治理为例。

01 洞察

二方包核心指标包括：依赖深度，依赖总数，版本总数。团队主管可以在"洞察报表"中选择自己的团队，找到最需要优化的二方包，进行针对性重点优化。

● 如何选择"三高"对象

重点治理对象一般是"三高"对象，即依赖深度等级过高，依赖总数等级过高，版本总数等级过高的二方包。如下图中，"com.tao**d:fee**nt"是典型的三高对象，需要优先，重点治理。



发现三高制品

● 如何找到"元凶"

因依赖深度，依赖总数是对 GA 最后一个 release 版本且有被应用依赖的二方包，所以要选择符合这条件的二方包版本，如下图:



确定"元凶"

接下来，我们查看"依赖树详情"，再分析，因为有可能自己的二方包也是因为引用了一个"三高"二方包，而导致自己"三高"的，在"依赖树详情"中，就可以找到它，而它可能就是"元凶"。

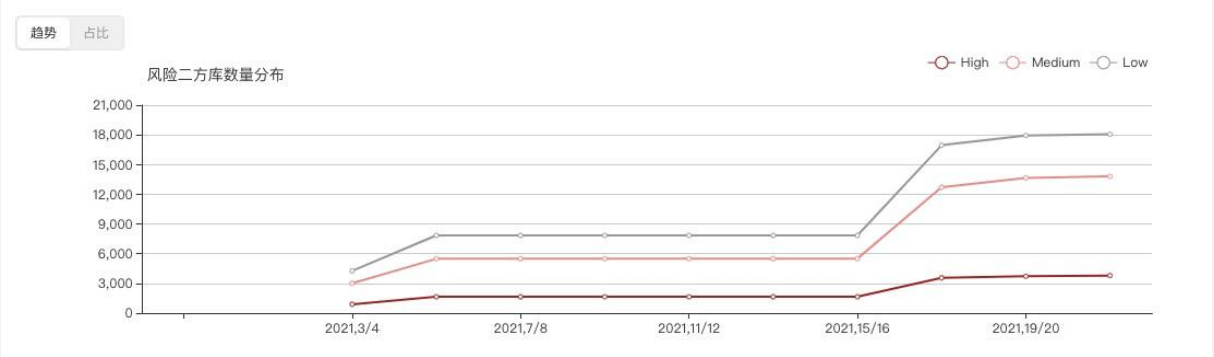
在找到"元凶"后，一般这样处理:

1. 联系"元凶"二方包的负责同学，进行优化。
2. 先去掉它的一些没使用到的间接依赖。

02 态势

态势报表主要体现以下两个方面:

1. 如对制品没治理，那包括风险等质量问题会如何恶化?
2. 如对制品进行了治理，那治理的效果如何。如效果不好，则再引申出是治理的方法有问题，还是治理的力度不够?



态势分析

03 元数据在持续交付流程中的应用

我们会根据历史出现过的故障，及安全、测试质量等要求，对一次发布中的制品及它的依赖进行扫描。并基于制品的元数据分析，将发布的风险进行分类，并提示用户如何修复。



流程中风险提示



风险详情

风险等级

风险按照严重等级分为高危、中危和低危，其中高危和中危需要重点关注。等级定义如下：

高危

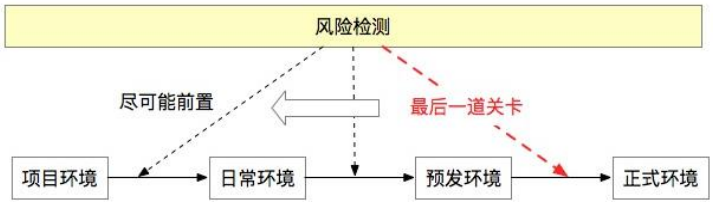
- 本地发布的 snapshot 更新
- release 版本号降低
- 新增的 release 包有依赖 snapshot
- 包含有 x86 的 so 的 jar 包用于非 x86 的服务器上

中危

- 新增相同 ga，不同 version 的依赖
- 新增了 snapshot
- 前后二个版本代码删除过多

风险检测流程

首先会在"正式发布"时卡最后一道关，然后基于风险修复成本，将风险检测尽可能提前。



最后一道关卡

小结

从基于代码的交付到基于制品的交付，其核心区别在于制品是完整和不可变的。这样基于制品及其元数据构建的持续交付体系，可以做到可信发布，极大地提升发布效率、降低发布风险。

基于应用的持续运维

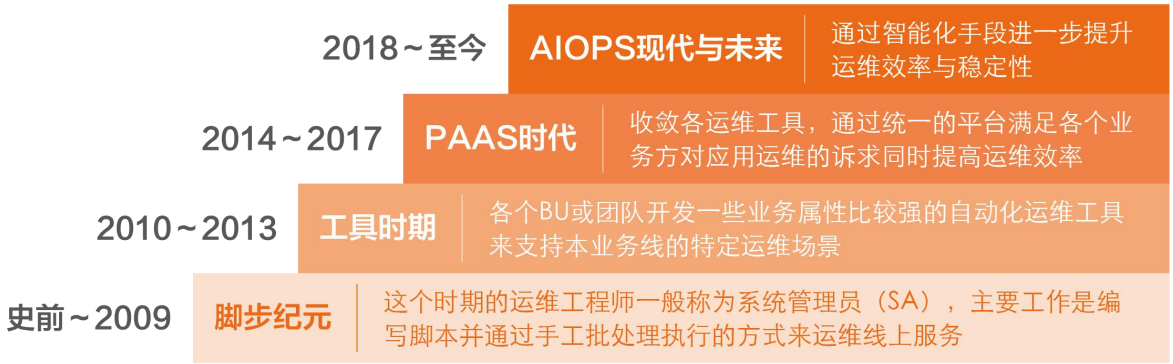
04

阿　里　云　云　效

04

4.1 监管控一体化运维

阿里巴巴的运维体系经历了脚本时代、工具时代和 DevOps 时代，目前正在实现自动化运维并探索智能化运维阶段。在 2008-2009 年，阿里巴巴的运维还处于脚本时代，大量的运维工作需要通过脚本来实现。随着业务规模扩大和复杂度的提高，脚本的方式越来越难以维护，因此阿里巴巴开始引入运维工具。在运维工具时代，阿里巴巴的运维体系经历了：从工具团队和运维团队并行的阶段，到为了更好地保障工具质量统一的工具团队阶段，再到逐渐有 DevOps 思想和职能的偏软件的工具团队阶段。最后，阿里巴巴应用运维团队迎来了一场大变革，以前的应用运维团队全被打散，合并到各业务的软件开发团队中去，全面践行 DevOps 思想。



进入 DevOps 阶段后，成熟的流程化运维工具虽然提升了一部分运维效率，但是各个工具之间实际是独立割裂的，例如监控工具和运维工具是割裂的，巡检工具和快恢工具也是割裂的，这导致日常应用持续运维过程中，从监控发现、定位并快速恢复问题的链路很长而且低效。对运维开发来说，期望的状态是业务应用上线后可以“No Ops”，监控及运维系统能自行发现异常并自动解决，把应用及业务带回正常状态，处理结束后，发一个消息通知下即可。朝着“No Ops”方向努力，阿里巴巴应用运维开始了“监管控一体化”的体系建设。

新挑战

随着阿里巴巴业务的持续发展及技术架构地不断演进，新的场景和问题不断出现，这些都给以应用为中心的监控运维带来了新的挑战。

01 超大规模

阿里巴巴不但拥有众多形态各异的业务，而且体量大，特别是每年天猫双 11 大促，需要超大规模的 IAAS 资源支撑。2015 年之前，阿里巴巴每年都要花费巨额费用来购买服务器，建设一代又一代的 IDC 数据中心；2015 年至 2019 年，阿里巴巴走向全面云化的过程。在这个时期，阿里巴巴的基础设施一部分在云下数据中心，另一部分在阿里云上的数据中心，还需要支持同城多活到异地多活，所以必须要有强大的云上云下一体化超大规模资源管理的能力；2019 年阿里巴巴实现全面云化之后，又开始面对一个新的超大规模资源管理场景：混合云。

02 运维效率

业务发展瞬息万变，特别是公司的重要业务，迭代变更的速度非常快。在超大规模集群管理的前提下，为了保障业务的连续性和快速迭代，我们需要有能力持续高效地对应用进行发布、部署、变配等运维变更。这也就是 DevOps 的持续运维领域要去解决的问题。

03 运维安全

安全性是任何行业的基础，在 IT 运维领域更是如此。系统宕机、数据异常、数据丢失、删库跑路等运维故障和事件层出不穷，这可能给企业带来致命的打击，甚至关乎业务的生死存亡。因此，防范和杜绝高危运维故障是 DevOps 一直不懈追求的目标。在当代众多业务形态和云技术架构下，如何保障企业 IT 运维的安全运行至关重要。

04 业务连续性

在阿里巴巴传统的监控和运维模式下，应用的运维开发需要在监控系统上配置一些监控项和预警规则。当监控项触发预警规则时，运维开发会收到预警通知。紧接着运维开发需要打开电脑，在运维工具平台上创建相应的处理工单。当运维系统工单执行完成后，运维开发要持续观察监控项是否回归正常。若遇到节假日或休假

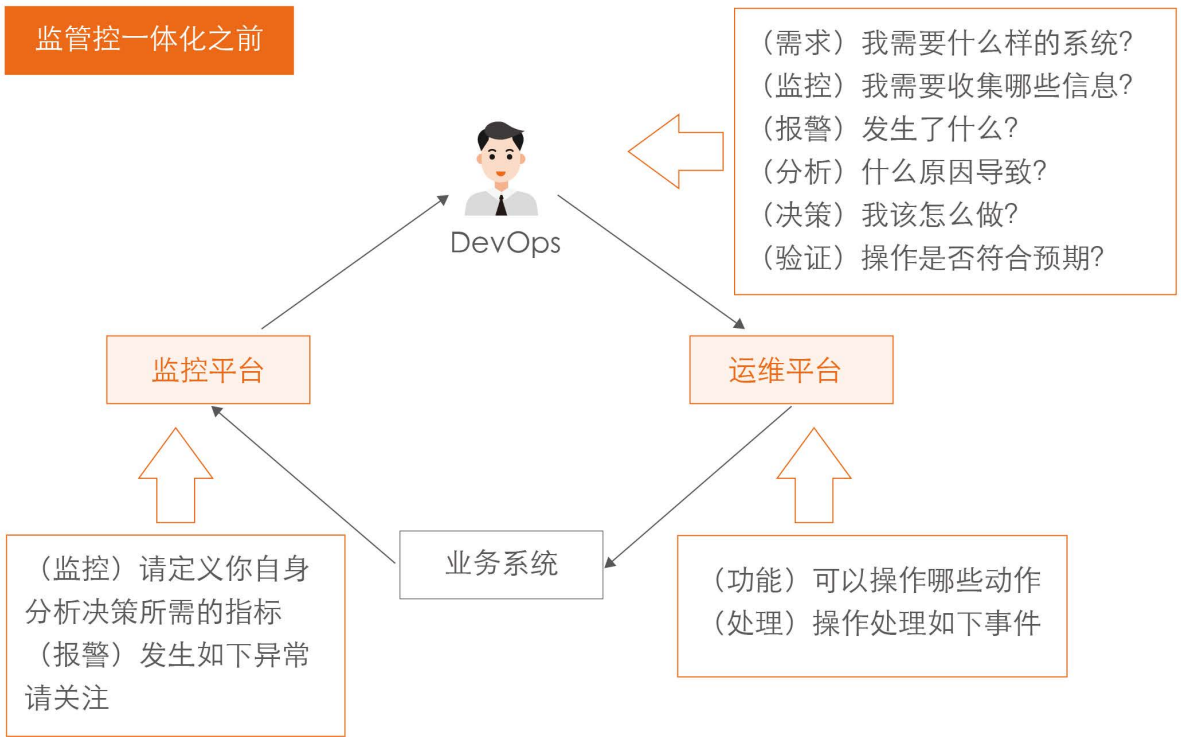
期间接到预警通知，不能及时上线查看情况时还需要联系团队其他同学上线处理；若在半夜睡梦中接到预警通知，需要立马清醒下自己的大脑，打开电脑上线处理。整个预警异常处理过程持续时间较长，并且需要人为参与的工作很多，人力成本大，这使得运维开发的工作幸福感很低。

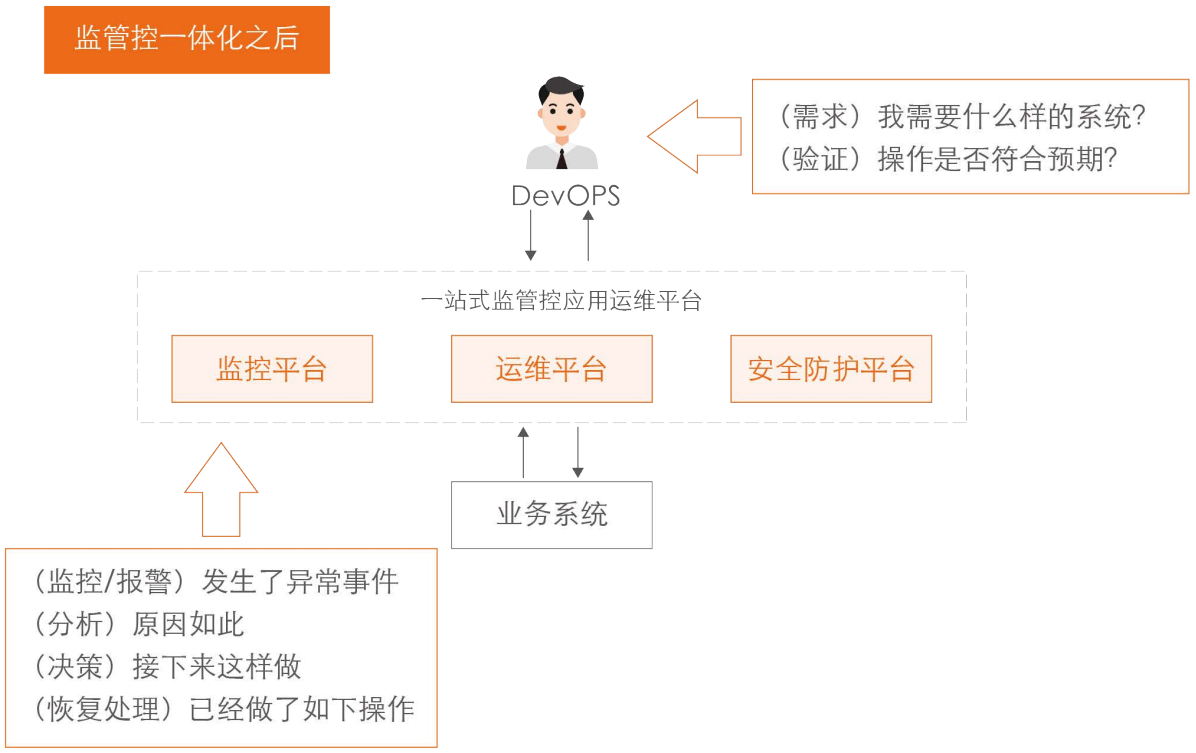
另一方面，随着业务地不断发展，系统也在不断增加，监控项和预警也急速增多，慢慢地运维开发就会对预警信息变得麻木或轻视，容易错失一些重要的报警信息，进而导致线上业务故障。近年来，淘宝直播、盒马线下门店、饿了么外卖、钉钉在线教育等新业务形态蓬勃发展，这些业务基本上对生产故障都是零容忍，原来系统最佳的 99.99%可用性已经不能满足新业务的要求，而传统的监控、运维各自为战、单打独斗的模式更无法满足新业务 100%业务连续性的要求。

解决思路

为了保证生产业务的连续运行，提高业务系统从异常预警到异常恢复的整体效率，解放人力成本的同时又能确保安全，我们考虑将监控预警和运维执行联动起来，视为一体，从而实现异常自动发现、自动快速定位以及自动快速恢复的目的，达到一种“No Ops”状态的应用运维。

在应用监管控一体化建设之前，传统的监控和运维是分开的状态，运维开发想要在应用迭代变更期间关注系统运行态势，需要事前在监控平台上定义和配置好这些应用所要关注的各项指标。在应用变更期间需要不断主动查看应用监控指标的变化，或者为每个指标设置预警规则，通过订阅接收配置好的监控报警来及时获取应用的运行异常。当应用变更出现异常报警后，运维开发需要看监控、应用日志、应用调用链路等信息分析异常原因，决策需要到运维平台上执行什么任务才能恢复，最后验证任务执行结果是否符合预期。因此，明确需求->配置监控指标和报警->分析异常原因->决策处理方式->执行任务->验证执行结果，整个过程都需要运维开发的介入。


监管控一体化之前
（需求）我需要什么样的系统?
（监控）我需要收集哪些信息?
（报警）发生了什么?
（分析）什么原因导致?
（决策）我该怎么做?
（验证）操作是否符合预期?
DevOps
监控平台
运维平台
业务系统
（监控）请定义你自身分析决策所需的指标
（报警）发生如下异常请关注
（功能）可以操作哪些动作
（处理）操作处理如下事件


监管控一体化之后
（需求）我需要什么样的系统?
（验证）操作是否符合预期?
DevOPS
一站式监管控应用运维平台
监控平台
运维平台
安全防护平台
业务系统
（监控/报警）发生了异常事件
（分析）原因如此
（决策）接下来这样做
（恢复处理）已经做了如下操作

解决方案

以保障业务持续性为源动力，在逐步推进监管控一体化建设过程中，阿里巴巴从实战经验沉淀出一套业务系统安全工程标准，实现了业务异常故障提前发现、自动定位、快速恢复地自动联动，在监控、运维、安全防护领域探索出了多样化的解决方案。

安全防护

在推进 DevOps 的过程中，我们要求的底线是不能对既有的现状带来更多不可控的因素，特别是高危场景的防护，不能因为运维工作移交到运维开发人员而造成全局系统性风险，因此安全防护方案孕育而生。

全景监控

监控是运维的基础，传统的资源监控或者应用监控模式已经不能满足运维开发快速发现生产故障的需求。基于阿里的大规模实践，我们发展出了以应用为中心，从上层业务到 PaaS 直至底层资源的全链路监控解决方案，为业务异常发现和定位提供了强有力的支撑。

多样化运维

为了实现监管控一体化，促进业务异常能快速自动恢复，应用运维从原来的单事件执行模式，探索出以应用为中心的可编排运维、智能化运维、ChatOps 等运维模式，打开运维领域新视角。

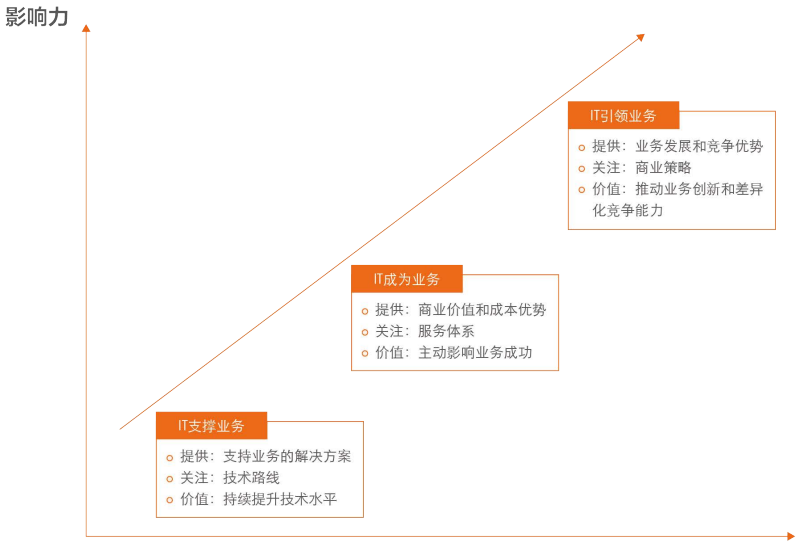
总结

阿里巴巴应用运维监管控一体化的建设随着业务形态和技术架构还在不断地探索和发展，本文主要介绍了应用运维监管控一体化建设的背景和思路。我们以应用为中心，从应用监控管角度出发，通过全视角监控实时掌握应用的运行状态，通过高效发布部署和灵活的运维编排对应用进行安全变更，通过智能化运维和安全防护实现应用的高级防护，我们将在下面的章节为你详细展开。

04

4.2 业务系统安全工程

随着企业的数字化演进，信息系统在业务中的地位越来越高。最早的信息系统主要是为了支撑业务发展，提升业务运作效率，后来逐渐发展成为业务，提供商业价值和成本优势。随着国家数字化政策的推进，信息系统的地位愈发提高，逐步引领业务发展，提供竞争优势。



■ 面临的挑战

随着业务越来越强依赖信息系统，信息系统的稳定和安全问题逐步凸显，主要表现为以下三大痛点：

系统故障问题频发

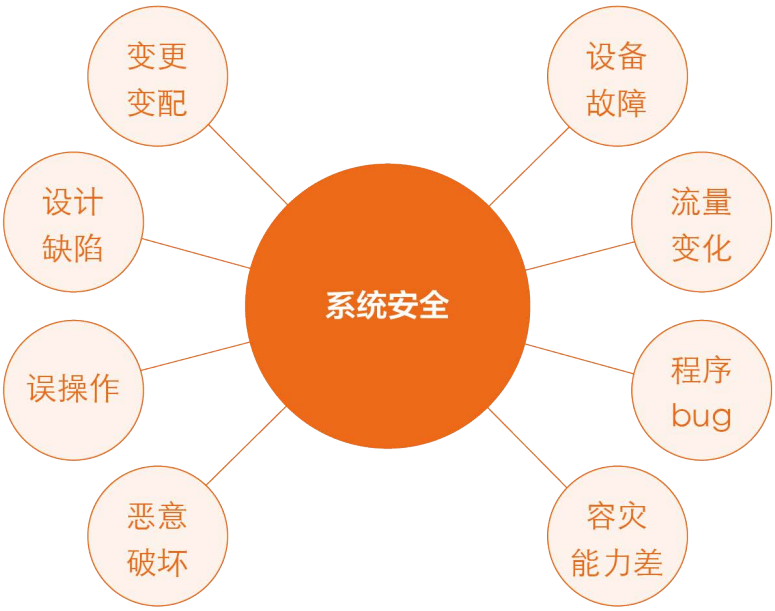
由于越来越多的用户通过信息系统获取服务，一旦系统出现故障，造成的影响和损失都是巨大的。每隔一段时间，市面上都会出现一些大规模性故障，对业务带来严重影响。仅仅 2020 一年，就发生过这些故障：

- 2 月 23 日，微盟数据库被恶意删除，停止服务 7 天，市值蒸发超 10 亿元。
- 5 月 13 日，特斯拉服务系统宕机，市值一夜间蒸发 2800 亿。
- 6 月 3 日苹果 iCloud 云存储服务器故障，用户无法登录。
- 8 月 27 日，思科员工删除虚拟机，导致思科损失 1600 万。
- 12 月 25 日，谷歌服务全球性宕机。

频发的故障不仅会导致经济损失，影响用户的使用和体验，同时也会伤害企业的公信力，特别是在一些对于安全性要求较高的行业。

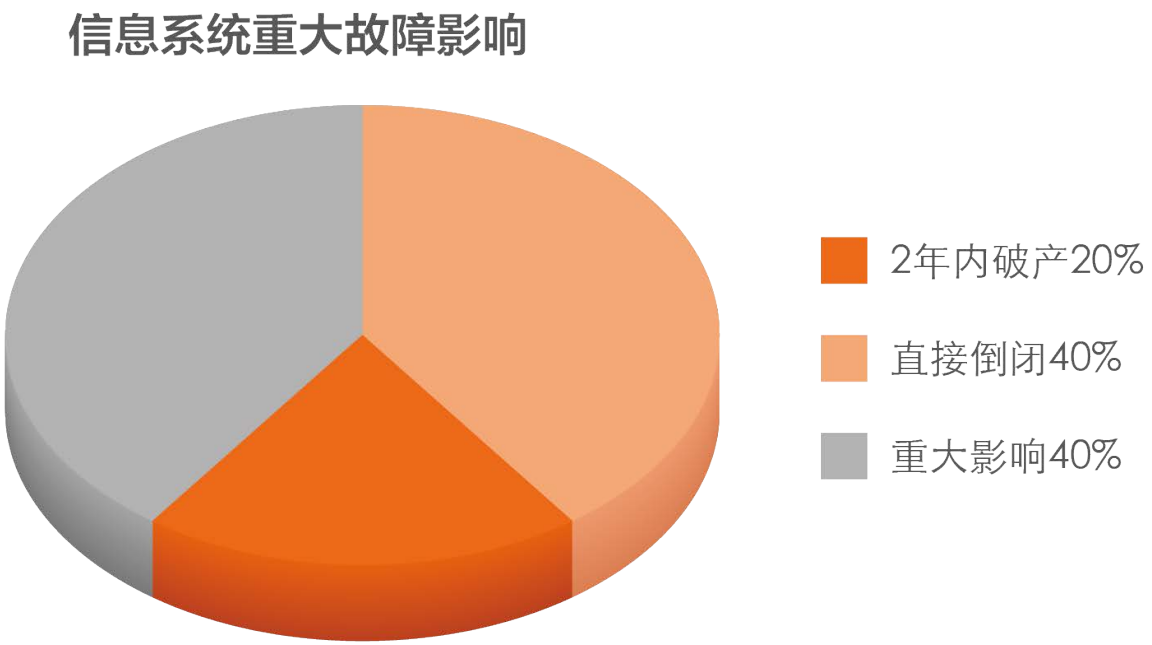
诱因复杂，控制难度大

随着业务规模的扩大，支撑业务的系统变得越来越复杂。这个复杂一方面体现在系统逻辑上，另一方面表现在研发团队的扩大带来的人员上的复杂性，例如研发运维人员变更变配、设备故障、误操作、恶意破坏、程序 Bug 等。这些都导致故障控制的难度越来越大，甚至出现屡控不住，越控越多的情况。



● 管理者缺乏安全感

据 Gartner Group 的调查，在信息系统经历重大故障后，至少有 40%的公司没有继续运营，而剩下的企业中，有 1/3 在两年内破产。系统的安全、稳定已经成为企业安全的重要命门之一。各种故障、以及众多潜在的系统威胁，也让企业管理者极度缺乏安全感。



■ 解决方案实践

系统故障的诱因复杂，这导致单点的控制很难解决问题，需要一个系统化的解决方案。第一届天猫双十一，开发和运维人员需要整夜保障，随时解决出现的问题。即便这样，也会出一些意想不到的故障。2020 年，双十一的用户数量和销售规模和第一届双十一已经不可同日而语，系统也更加的复杂，但双十一大促的系统保障过程却越来越流畅，保障的人数也在持续降低。这背后是一个系统化的解决方案。

● 组织的顶层设计

组织设计是从组织层面设置专门的组织机构来负责系统的稳定和安全，包括最高层的安全生产委员会和各个研发部门的稳定性负责人。安全生产委员会的职能包括负责全局稳定性决策、安全生产规则制定、整体应急协同、安全文化培养、全局管控系统的规划与管理。当故障发生时，由相关人员负责故障应急与统筹。各研发

部门稳定性负责人负责各系统的风险治理和稳定性保障，在研发、运维的过程中避免系统故障的出现。

● 事前的风险预防

防患于未然是安全的最高能力。事前风险预防包括事前分析系统的各个组成要素以及组成要素可能面临的威胁和存在的脆弱性，并将分析结果作为安全治理的输入。对于威胁，需要制定相应的措施避免或减少威胁的发生。对于脆弱性，需要针对性的进行巩固。比如对于经常会导致系统故障的系统变配操作，通过统一的变更平台集中管理各种变配申请，从而实现对变配操作的集中管控。其次，通过最小权限原则，限制操作人的操作权限，包括操作时间的限制、操作对象的限制和操作范围的限制。另外，每一次的变配操作，系统可以根据操作人、操作对象、操作类型等要素，计算操作过程中存在的风险。一旦发现过程中存在确定风险，则会直接阻断当前操作。如果是高风险，则会发起交叉确认流程。如果是低风险，则会直接放行。这种方式，既实现了对风险的实时管控，防止由于人为失误导致的故障，同时又平衡了研发效率与安全生产间的关系。

● 事中的实时监控

快速发现是避免损失扩大的重要手段。在系统运行的过程中，通过业务指标监控、应用程序监控、云资源监控相结合的方式，能够及时地发现系统存在的问题。一旦发现故障，按照事先制定的预案，系统会通知相关人员进行处理。其次，基于大数据和人工智能的算法，平台会实时预测相关指标的变化趋势，将故障预警的时间再次提前。

● 事后的快速恢复

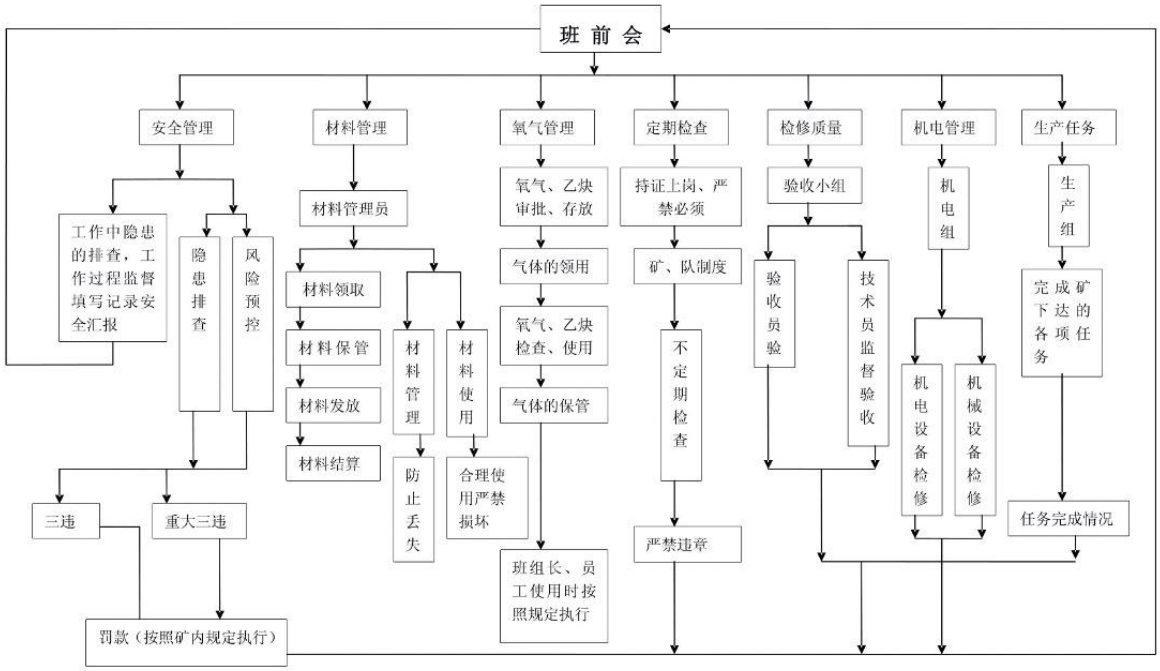
尽管事前事中制定了详尽的方案，但是还是很难避免故障的发生。一旦故障发生，如何快速地进行故障恢复就是首先需要做的事情。按照故障的不同类型，可以使用的故障恢复手段有限流、拦截、熔断、快恢、降级、扩容、切流、重启等。不同的恢复方式都需要有相应的系统支持和日常的演练测试。

故障恢复后，安全生产委员会还需要组织相关人员排查和分析故障原因，制定整改方案，确定故障责任人，推进和落实整改方案，防止相同故障的再次发生。

■ 业务系统安全工程

从以上的实践过程可以看出，企业很难依靠单一手段解决系统故障，而需要通过系统化的手段，从顶层的组织设计、事前的风险分析和策略制定、事中的持续监测和预警、日常的演练和事后的应急响应等多方面进行控制。

在传统行业中，为了保证生产经营活动能够正常运行，国家制定了一系列的措施使生产过程在符合规定的物质条件和工作秩序下进行，从而有效消除或控制危险和有害因素，减少人身伤亡和财产损失，保障人员安全与健康、设备和设施免受损坏、环境免遭破坏。在建筑、石油化工、交通运输、航空航天等行业，安全生产已相对成熟和完备，但在互联网领域还是空白。以下图采矿业安全生产流程为例，我们可以看出安全生产的管理要求已经落实到了作业的各个过程和环节。

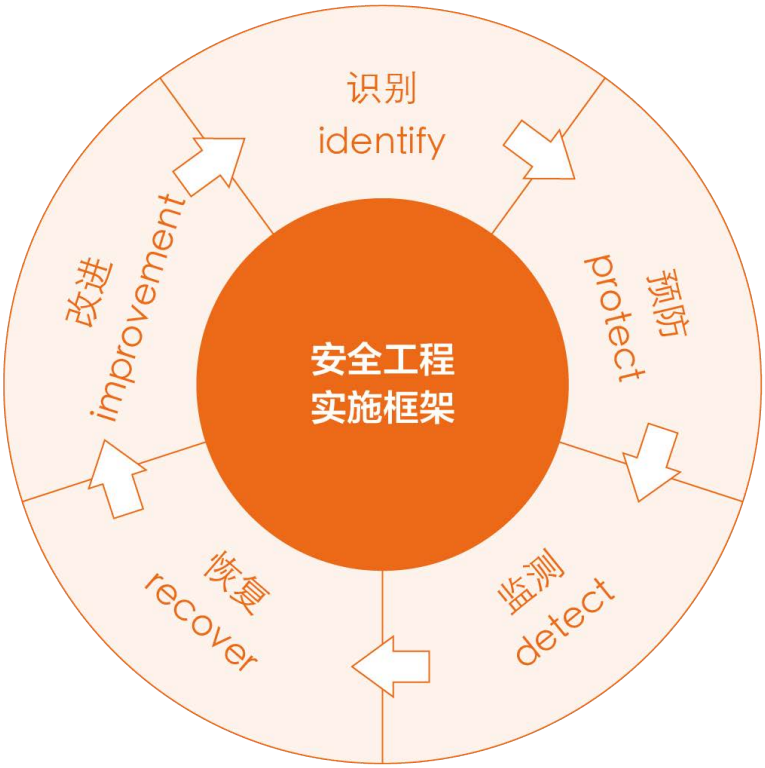


图：采矿业安全生产流程

参考传统行业中的安全生产解决方案，同时结合阿里巴巴内部的最佳实践，我们提出了业务系统安全工程解决方案，该方案是指导业务系统防范故障的安全指南，其目标是通过预防、监测预警、应急响应等手段，减少业务系统故障，保障业务系统稳定、可用和可靠，防范由于业务系统故障导致的资产损失和用户影响。

业务系统安全工程框架

由于业务系统以及故障原因的复杂性，单纯的从一个或多个点出发很难解决问题。业务系统安全工程以控制论和系统论为指导，以风险控制方法为工具，形成了自己的实施框架 IPDRI，即识别（identify)、预防（protect）、监测（detect）、恢复（recover）和改进（improvement）五个环节。从事前、事中、事后进行风险的控制，形成闭环的反馈网络。

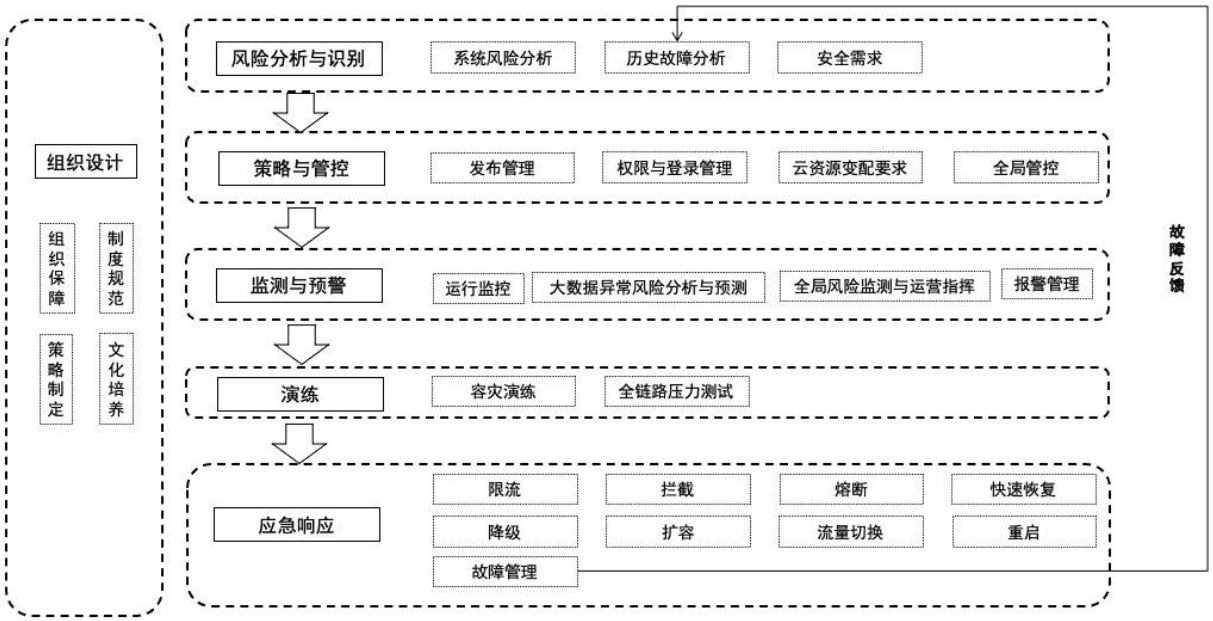


其中，识别包括资产分析、威胁识别、脆弱性识别等。预防是为了避免风险的发生而采取的一定的预防措施。监测是监测系统和保护措施是否在正常的运行。恢复是在故障出现时快速的采取措施恢复系统的运行。改进是查找故障原因，制定改进方案避免相同故障的再次发生。

业务系统安全工程标准

在此背景下，阿里云联合国家信通院牵头起草了《基于云计算的数字化业务安全工程标准》，该标准是国内首部聚焦于保护系统持续正常运行的行业标准。标准的核心目标是保护业务系统能够持续正常运行，防范由于业务系统故障导致的资产损失和用户影响，保证系统的可用性、稳定性和可靠性。

标准规定了企业实现业务系统持续正常运行需要具备的各项能力，包括组织设计能力、风险分析与识别能力、策略与管控能力、监测与预警能力以及应急响应能力。



其中：

- 组织设计能力规定企业应设立顶层安全生产委员会，下辖公司安全生产部门，用技术手段提升风险控制能力，保障业务稳定；打造安全生产文化，确保人人重视、有持续性的提升；明确行为准则，用机制保护人，减少犯错，降低损失，以此快速推进稳定性治理，大幅收敛公司全局性故障和重大影响故障。
- 风险分析与识别模块帮助企业通过对系统的脆弱性、业务安全生产需求、系统已发生故障的分析，寻找影响信息系统安全生产的潜在风险。
- 策略与管控模块是针对已经分析发现的风险制定安全生产管控策略，通过降低、预防威胁的发生，提前巩固、消除脆弱性等手段预防风险的发生。
- 监测与预警模块是通过业务状态监控、云资源状态监控、大数据风险分析与预警以及预警管理等能力，快速发现风险。
- 应急响应模块规定了企业缩短故障时间、快速恢复故障应该具备的响应和快恢能力，包括容灾演练、切流、限流、降级、重启、拦截、扩容等能力。

总结

系统的安全受内部和外部双重影响，在防止企业系统受外部影响上，信息安全目前相关的理论研究和产品建设已经较为完善。当前系统故障的更多原因是由于企业内部问题导致的，信息系统安全工程作为降低系统故障的体系化解决方案，未来的相关理论研究、产品服务也将得到快速发展。

04

4.3 全景监控

随着云原生技术的发展与演进，微服务和容器化技术成为大型分布式 IT 架构的必然选择。新技术在让 IT 系统变得更敏捷、健壮、高性能的同时，也带来了更高的技术架构复杂度，给应用监控带来了前所未有的挑战。

■ 传统监控挑战

01 系统监控爆炸式增长

传统监控重点关注应用、主机、网络等系统层面的监控。随着微服务、云原生等新技术的大量运用，系统架构越来越复杂，应用数量呈爆炸式增长。当一个故障发生时，大量系统报警会产生报警风暴，使得技术人员无法快速定位问题。同时，大量的系统级监控会产生大量误报，技术人员被迫花费大量的精力去处理这些误报，最终对报警产生麻木。

02 监控结果和业务目标之间欠缺联系

传统监控缺少业务视角的监控，以及业务与 IT 系统之间的联系。这导致用户、业务人员和技术人员之间无法形成统一视角。很多故障往往是用户已经反馈，技术人员却在用系统监控指标证明系统没有问题。或者业务已经受损，技术人员依然无法确定是哪个系统的问题，恢复时间被大大拉长。

03 监控工具之间数据割裂，数据缺乏分析

以往阿里巴巴采用多种监控工具，用于监控网络、物理机、应用、客户端等不同的对象，但不同工具之间无法共享数据，监控数据缺乏统一的分析，更加难以跟业务场景相结合，造成大量的故障只能依靠技术人员的经验，在多个工具之间不断地切换和逐步排查，大大增加了故障恢复时长。

04 监控维护成本高，报警准确率低

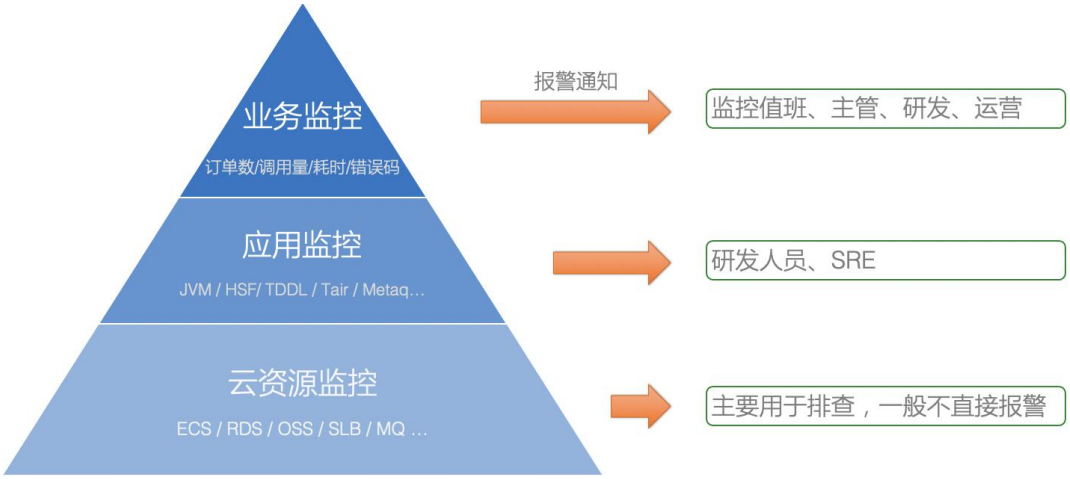
传统监控都需要大量的配置工作，整个过程费事费力，缺乏自动化、智能化监控的手段，这也是造成各系统监控能力参差不齐的重要原因。一些新业务因为无力投入大量精力配置监控，导致业务监控能力缺失。同时，随着业务的发展，技术人员需要不断地调整报警规则，又常常因不及时的调整造成误报和漏报。



业务驱动的监控理念

为了适应 DevOps 研发模式，解决传统监控的问题，阿里巴巴总结了一套以业务驱动的自顶向下的全景监控体系，主要包括：业务监控、应用监控和云资源监控三层：

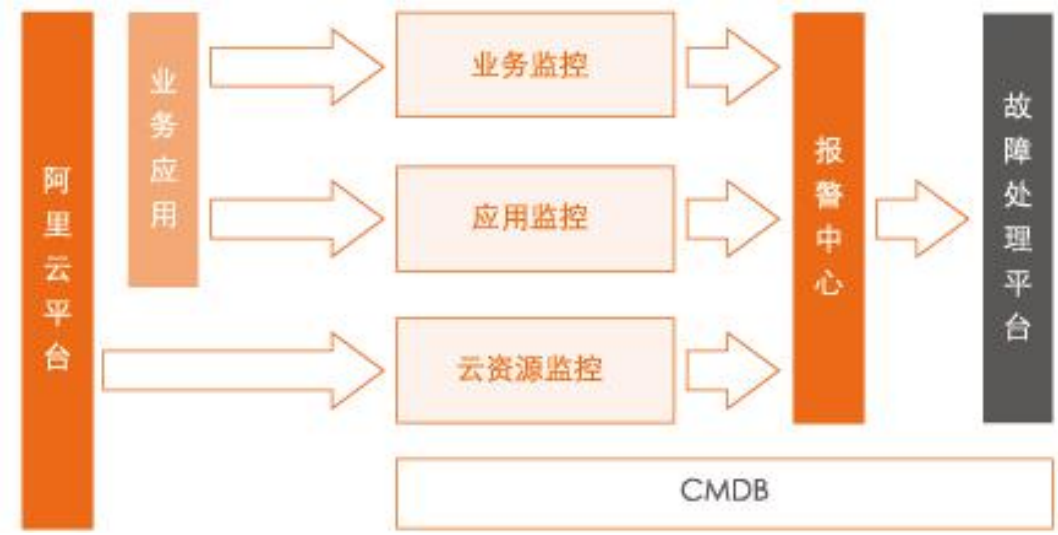
- **业务监控**是整个监控体系的“顶层”，能够反映用户使用业务的真实情况，可以与业务结果直接挂钩，能够被不同部门、不同角色的人员所理解。
- **应用监控**提供应用中服务和系统层监控能力，能够直接反应系统运行状态，帮助研发人员全面了解应用中服务和中间件的健康状态，快速定位系统问题。
- **云资源监控**提供应用依赖的各类云资源(如 RDS、OSS、SLB 等)的基本监控，在故障排查中能够为研发人员提供实例级别的明细监控数据，快速确定是应用系统的问题，还是云基础实施的问题。



监控分层以后，各层的监控指标和报警规则会按重要程度分成严重、警告、普通等多个等级，不同层次、不同级别的监控报警会被分派给不同的角色来处理，比如集团的安全生产团队只关注全集团的核心业务指标，事业部的稳定性负责人关注部门级的核心业务监控，每个团队的研发人员则接收自己负责业务和应用报警，而云资源实例的监控一般不发送告警消息，主要用于故障排查和定位。这样充分发挥了 DevOps 的优势，传统模式中少数运维人员成为故障排查瓶颈的问题也得以解决。同时，每个人需要处理的报警数量大幅减少，也解决了在故障发生时因为报警风暴，而将重要的业务监控报警淹没的问题。

统一的监控架构

基于全景监控的理念，阿里巴巴探索出了一套统一监控的架构，该架构不追求大一统的监控平台模式，而是采用分层建设，抽象出了云资源，应用，业务这 3 种监控系统，每种监控都专注发现相关领域的故障发现，再通过统一 CMDB 解决监控元数据相互不统一的问题，通过智能算法平台，报警中心和故障平台集中管理事件，故障以及提升准确率。

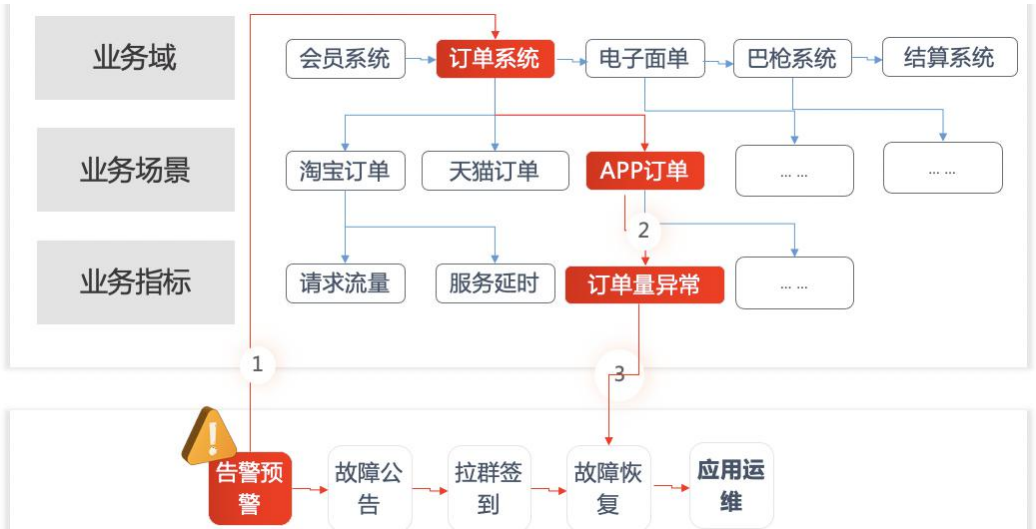


01 业务监控

阿里巴巴“业务监控”采用专为监控自研的日志采集&计算框架，通过页面配置即可实时提取日志中的监控指标，具有简单易用、自定义能力强、响应速度快、对业务无侵入等特点；提供了完整的业务监控领域模型，引导用户完成监控覆盖。

业务监控的领域模型包括：

- 业务域：一个完整的业务或产品称为“业务域”，如电商的“交易域”、“营销域”、“支付域”等。
- 业务场景：业务域中的核心业务用例叫做“业务场景”，如交易域的“下单确认”、“创建订单”等，业务场景是整个监控模型的核心。
- 业务指标：体现每个业务场景的特有指标，如每分钟的订单数量、交易成功率、错误码等。



在业务指标的选择上，传统的运维人员喜欢使用穷举式的手段配上所有可观测性的指标，各种告警加上，显得有“安全感”。实际当故障来临时，满屏出现指标异常、不断增加的告警短信，这样的监控看上去功能强大，实际效果却适得其反。

通过对阿里巴巴历年故障的仔细梳理，阿里巴巴集团内的核心业务的常见故障（非业务自身逻辑问题）都可以通过流量、时延、错误等 3 类指标反应出来，我们称之为黄金指标：

- **流量**：业务流量跌零 OR 不正常大幅度上涨下跌，中间件流量如消息提供的服务跌零等，都可能触发重大故障；
- **延时**：系统提供的服务 OR 系统依赖的服务，时延突然大幅度飙高，基本都是系统有问题的前兆；
- **错误**：服务返回错误的总数量，系统提供服务 OR 依赖服务的成功率。

业务监控平台提供了“黄金指标”插件，通过单次配置就可以生成一组黄金指标，是目前业务监控使用范围最广的指标模型。

业务监控报警直接与故障关联，对监控数据的质量有很高的要求，同时需要具备很好的灵活性（既满足不同技术实现的监控需求，又不能对被监控的业务系统造成性能影响）。阿里巴巴的“业务监控”通过日志作为数据来源，能最大程度地保证业务监控的灵活性，能够适应几乎所有的技术栈。日志采集采用了无压缩增量采集、zero-copy 等技术，降低监控采集对业务系统的性能影响；采用拉模式架构、重试机制、数据齐全度模型保证了数据采集的可靠性和完整性；完全白屏化的配置能力、完善的调试功能，最大程度降低了用户的配置难度和配置成本。

02 应用监控

阿里巴巴应用监控采用标准化和组件化的方式搭建，与阿里巴巴技术栈相结合，提供常用的系统和中间件层面的监控组件；运维同学无需修改程序代码，整个监控过程自动化，应用上线、扩容后自动开启应用监控，无需人为操作，大幅降低了监控的维护成本。

当运维系统执行应用上线、扩容等操作后会将变更信息写入到 CMDB，CMDB 会将变更信息推送到 MQ，应用监控平台订阅 MQ 实时获取应用的配置变化生成新的监控任务，监控任务下发到指定的目标服务器（容器）的 Agent 端，Agent 根据任务的配置信息发送对应的采集请求，从业务应用提供的 Exporter 等 EndPoint 中获取监控数据，并上传到监控集群进行计算和存储；同时异常检测模块也会根据应用配置的变化生成报警检测任务，从时序数据库中拉取监控数据进行异常检测，并将异常事件发送给报警中心。



03 云资源监控

阿里巴巴云资源监控直接对接阿里云平台的“云监控” API 获取各类云资源的指标数据和报警事件，再将这些数据与 CMDB 中应用与云资源的关系信息连接，最终形成应用视角的云资源健康状态视图，解决了云基础设施监控与上层应用监控相互隔离的问题。依托云平台的监控能力和 CMDB 的数据积累，整个云资源监控也是自动化完成的，无需用户人工配置。

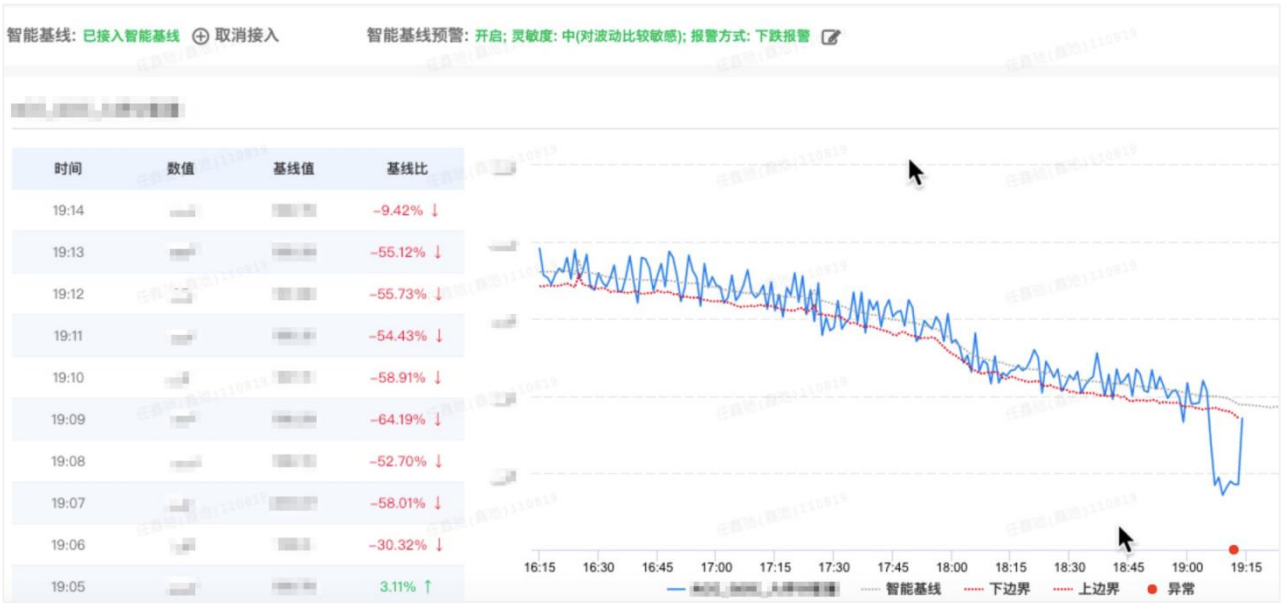
● 智能检测平台

为了解决报警准确率低、配置维护成本高的问题，阿里巴巴建设了智能检测平台，通过 AI 算法精确地发现线上业务和应用的异常，并且在此过程中不需要任何人工配置报警阈值。针对业务和应用监控数据的不同特点，采取不同的异常检测策略：

1. 智能基线

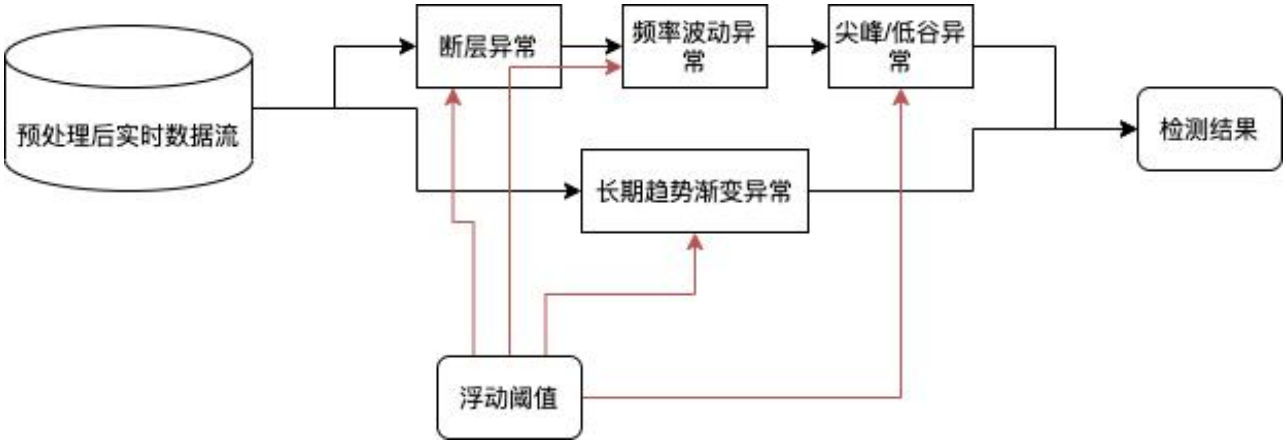
业务监控对报警的准确性要求极高，同时数据会随着业务周期不断波动，高峰和低谷之间数据可能相差几十倍甚至上百倍。传统的阈值或同环比报警往往需要有经验的运维专家不断地调整规则，极易引发误报。对此，阿里巴巴采用智能基线算法通过历史趋势自动学习数据曲线的周期规律。当业务指标超出基线容忍范围时，就会立即触发业务报警。为了优化算法对数据的兼容性，智能基线算法通过在线预测的功能（即算法对往后一段

时间的数据进行逐点预测），完成了对长期历史规律与近期历史规律较好地折衷；基于对阿里巴巴内部大量多样化业务指标的训练和专家经验标注，让平台对不同类型的业务波动能优雅地在算法中体现出来，算法可以很好地适配数据曲线中的波动毛刺及随业务产生的高低起伏，可一键接入各类业务监控数据；算法长期经受各种外部攻击及爬虫的内部压测干扰的历练，目前已具备了对干扰攻击较好的抵抗能力；算法可以很好的支持秒级与分钟级计算，无需任何人工监控配置，无需随业务变化对算法进行调参，算法可自己通过对规律的学习适应业务的变化。



2. 应用指标异常检测

应用监控指标数量众多，传统的人工配置阈值的方式成本极高。企业往往会采用报警模板的方式对大量应用配置相同的报警阈值，但由于不同应用系统的差异性较大，很难定义一个准确的阈值，这就容易产生“小了漏报，大了误报”的问题。系统指标的场景与业务指标不同，它的周期性更不确定，各个指标的浮动相对来说会比较大且没有周期特性。针对应用指标的特性，阿里巴巴又针对性开发了应用指标异常检测算法，通过断层检测、频率波动异常检测、尖峰/低谷异常检测、长期趋势渐变检测、浮动阈值等多种算法联合进行检测；同时由于应用监控指标数量众多，为了能够大范围使用，所有检测方式都采用了轻量化算法，大幅降低异常检测服务的资源消耗。



报警中心：统一对接各监控平台的报警事件，实现报警事件地统一记录、统一处理（合并、降噪、抑制等），最后发送到相关处理人。

故障管理平台：用于定义故障等级并管理整个故障生命周期的平台，命中故障等级定义的重要报警将升级成故障，进入故障管理流程。

CMDB：运维统一 CMDB 是整个阿里巴巴应用运维体系的元数据中心，维护着整个阿里巴巴的产品、应用、实例（容器、VM、云资源）、机房、单元、环境等运维对象，以及对象之间的关联关系，各层的监控系统都与 CMDB 模型中的对象关联。

通过这样体系化的建设，我们不但能在故障发生时快速准确的发出告警，还具备了从业务入口下钻分析故障链路上各个关键应用，资源状态，甚至基础设施的能力。因此开发运维人员可以在一个监控界面上逐步排除故障发生时的可疑点，快速定界到故障发生的原因

这里以某订单系统调用延时突增故障为例，介绍一下全景监控的故障排查过程：

1. 线上问题发生的第一时间，负责该订单系统的研发 Owner 会收到报警中心的调用延时报警（如果线上问题达到故障等级标准，故障台将自动发送对应等级故障通告给相关人员）。

2. 研发人员通过报警信息中的链接直接打开对应业务监控页面查看交易总调用量、延时、成功率等黄金指标数据，发现延时大幅升高，成功率有所下降，调用量没有明细下跌，排除上游流量徒增造成系统容量问题的情况。

3. 通过业务监控的多维下钻功能，查看交易的错误码详情可以发现“超时错误码”大量增加，可以排除是业务逻辑类问题。

4. 继续从业务监控下钻到应用监控，根据订单指标对应的调用链路发现，某下单应用的数据库调用成功率大幅下降，调用延时暴增。

5. 再从应用监控下钻到云资源监控，查看该应用关联的数据库的 CPU、调用时长、慢 SQL 指标都出现异常，查看慢 SQL 列表和应用的变更记录发现与上一次发布的定时任务相关，通过运维系统回滚后解决问题。

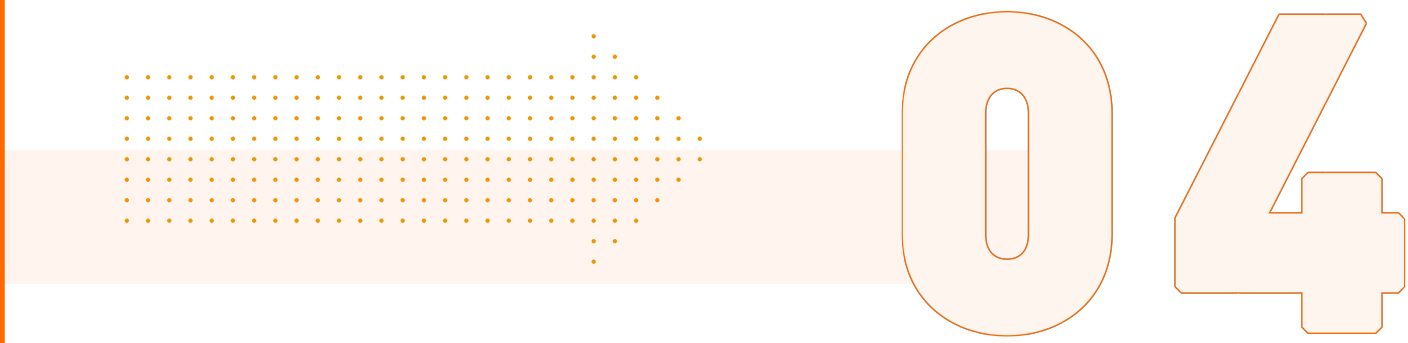
监控管理体系

好的监控体系除了有优秀的监控功能以外，还必须有与之配套的管理系统。阿里巴巴采用了故障管理驱动的监控管理体系，为各个部门、团队制定了严格的量化故障等级定义。故障等级定义直接与业务监控指标关联，明确不同故障等级对应的指标触发规则。安全生产团队会与各业务部门梳理核心业务场景、业务指标和故障等级定义。梳理完成的“业务监控”配置和“故障等级定义”需要通过评审达成一致，使得业务团队（运营、产品、客户）与研发团队之间形成统一的监控标准，明确各方的职责，降低沟通成本，实现监控结果与业务目标的强关联。

整个故障定义的过程都是线上化和结构化的，当业务指标超出故障定义的范围时，故障台会自动触发故障通告，并将通告信息及时发送给相关团队的技术人员。技术人员通过故障通告快速查看业务监控数据，通过全景监控的纵向拓扑联动能力，从业务指标下钻分析到关联应用状态，再从应用状态下钻分析到云资源状态，实现快速故障定位。然后技术人员根据故障排查的信息，确定故障恢复方案，并通过运维平台执行回滚、降级、切流等操作快速恢复故障。整个过程都在线上完成，故障排查的进展会自动推送给相关人员，所有操作都会被记录。最后由安全生产团队组织故障复盘，制定改进措施、完善监控覆盖，实现业务安全生产的正向反馈。

总结

全景监控不仅是业务、应用、资源等分层监控能力的简单集成，更重要的是具备通过业务指标下钻分析到应用状态，及从应用状态下钻分析到资源状态的纵向拓扑联动能力，也是各层指标的智能化健康检查能力的一体化监控。全景监控直击传统监控平台缺失业务监控能力、各层监控数据及报警分散、监控配置成本较高等痛点，基于阿里巴巴强大的监控技术积累和应急故障处理的最佳实践，为阿里巴巴经济体提供一体化、一站式的监控解决方案，是阿里巴巴安全生产管理的最佳实践。



4.4 发布策略

DevOps 追求更短的迭代周期、更高频的发布。但发布的次数越多，引入故障的可能性就越大。更多的故障将会降低服务的可用性，进而影响到客户体验。所以，为了保证服务质量，守好发布这个最后一道关，阿里逐步发展出了适应 DevOps 要求的发布策略。

在开始讲述阿里的实践之前，我们先简单介绍下几种常见发布策略，以及它们适用的场景和优缺点。

■ 常见发布策略

01 停机发布

停机发布会在发布以前关闭服务，停止用户访问，然后一次性的升级所有服务。这种发布策略的发布频率往往比较低，且需要在发布之前做好充足的测试。

停机发布的特点有：

- 所有需要升级的组件被整合到一次发布中
- 一个项目中的大部分应用都会被更新
- 发布之前的研发流程和测试流程往往需要花很长的时间
- 发布时如果出现问题，修复和回滚的成本很高

- 完成一次停机发布，需要花费很久的时间，且需要很多团队在一起才能完成
- 往往需要客户端和服务器端同步升级

停机发布并不适合互联网公司，因为两次发布的间隔很久，从功能特性提出到进入市场的时间太长，对市场反应不敏感，会在充分竞争的市场里处于下风。每次发布因为要停机，也会带来经济损失。

优势：

1. 简单，不太需要考虑新旧版本共存时的兼容性问题

劣势：

1. 发布过程中，服务不可用
2. 只能在业务低峰期（往往是夜间)发布，并且需要很多团队在一起工作
3. 出现故障后很难回滚

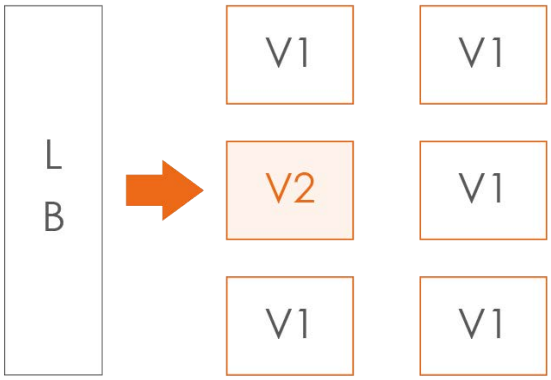
适合场景：

1. 开发测试环境
2. 非关键应用，用户影响面小
3. 兼容性比较难管控的场景

02 金丝雀发布

金丝雀发布这个术语源自 20 世纪初期，当时英国的煤矿工人在下井采矿之前，会把笼养的金丝雀携带到矿井中，如果矿井中一氧化碳等有毒气体的浓度过高，在影响矿工之前，金丝雀相比人类表现的更加敏感快速，金丝雀中毒之后，煤矿工人就知道该立刻撤离。金丝雀发布是在将整个软件的新版本发布给所有用户之前，先发布给部分用户，用真实的客户流量来测试，以保证软件不会出现严重问题，降低发布风险。

在实践中，金丝雀发布一般会先发布到一个小比例的机器，比如 2% 的服务器做流量验证，然后从中快速获得反馈，根据反馈决定是扩大发布还是回滚。金丝雀发布通常会结合监控系统，通过监控指标，观察金丝雀机器的健康状况。如果金丝雀测试通过，则把剩余的机器全部升级成新版本，否者回滚代码。



优势：

1. 对用户体验影响较小，在金丝雀发布过程中，只有少量用户会受影响

2. 发布安全能够得到保障

劣势：

1. 金丝雀的机器数量比较少，有一些问题并不能够暴露出来

适用场景：

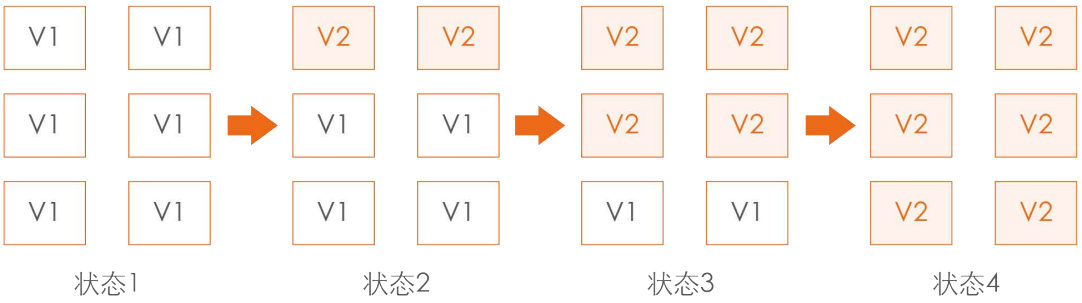
1. 监控比较完备且与发布系统集成

03 灰度/滚动发布

灰度发布是金丝雀发布的延伸，是将发布分成不同的阶段/批次，每个阶段/批次的用户数量逐级增加。如果新版本在当前阶段没有发现问题，就再增加用户数量进入下一个阶段，直至扩展到全部用户。

灰度发布可以减小发布风险，是一种零宕机时间的发布策略。它通过切换线上并存版本之间的路由权重，逐步从一个版本切换为另一个版本。整个发布过程会持续比较长的时间，在这段时间内，新旧代码共存，所以在开发过程中，需要考虑版本之间的兼容性，新旧代码共存不能影响功能可用性和用户体验。当新版本代码出现问题时，灰度发布能够比较快的回滚到老版本的代码上。

结合特性开关等技术，灰度发布可以实现更复杂灵活的发布策略。



优势：

1. 用户体验影响比较小，不需要停机发布

2. 能够控制发布风险

劣势：

1. 发布时间会比较长

2. 需要复杂的发布系统和负载均衡器

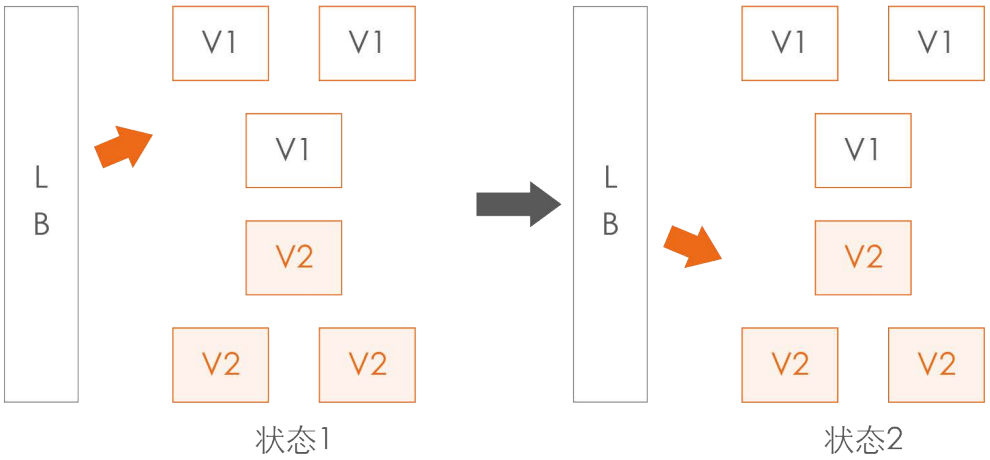
3. 需要考虑新旧版本共存时的兼容性

适用场景：

1. 适合可用性较高的生产环境发布

04 蓝绿发布

蓝绿部署是指有两个完全相同的、互相独立的生产环境，一个叫做“**蓝环境**”，一个叫做“**绿环境**”。其中，绿环境是用户正在使用的生产环境。当要部署一个新版本的时候，先把这个新版本部署到蓝环境中，然后在蓝环境中运行冒烟测试，以检查新版本是否正常工作。如果测试通过，发布系统更新路由配置，将用户流量从绿环境导向蓝环境，蓝环境就变成了生产环境。这种切换通常在一秒钟之内就能搞定。如果出了问题，把路由切回到绿环境上，再在蓝环境中调试，找到问题的原因。因此，蓝绿部署可以做到仅仅一次切换，立刻就向所有用户推出新版本，新功能对所有用户立刻生效可见。



优势：

1. 升级切换和回退速度非常快

2. 零停机时间

不足：

1. 一次性的全量切换，如果发布出现问题，会对用户产生比较大的影响

2. 需要两倍的机器资源

3. 需要中间件和应用自身支持热备集群的流量切换

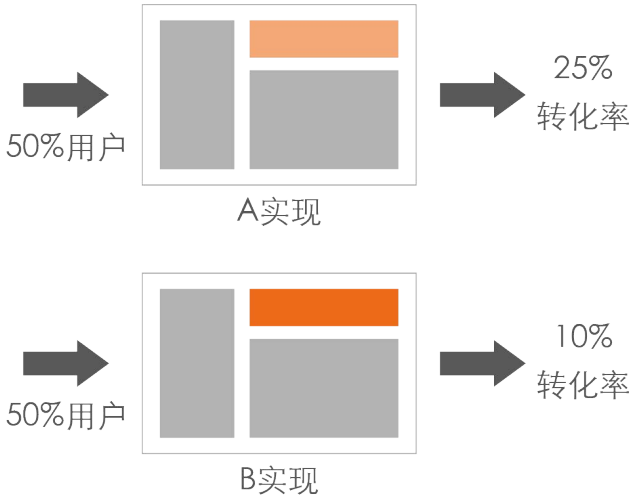
适用场景：

1. 机器资源比较富余或者按需分配（背靠云厂商）

05 A/B 测试

A/B 测试和灰度发布非常像，可以从发布的目的上进行区分。AB 测试侧重的是根据 A 版本和 B 版本的差异进行决策，最终选择一个版本进行部署。和灰度发布相比，AB 测试更倾向于去决策，和金丝雀发布相比，AB 测试在权重和流量的切换上更灵活。

举个例子，某功能有两个实现版本 A 和 B，通过细粒度的流量控制，把 50% 的用户总是引导到 A 实现上，把剩下的 50% 用户总是引导到 B 实现上，通过比较 A 实现和 B 实现的转化率，最终选择转化率较高的 A 实现作为功能的最终版本。



优势：

1. 快速实验能力
2. 用户体验影响小
3. 可以使用生产环境流量做测试
4. 可以针对某些特定用户做测试

不足：

1. 需要较为复杂的业务流量识别和控制能力
2. 需要考虑较为复杂的新旧版本兼容性问题

适用场景：

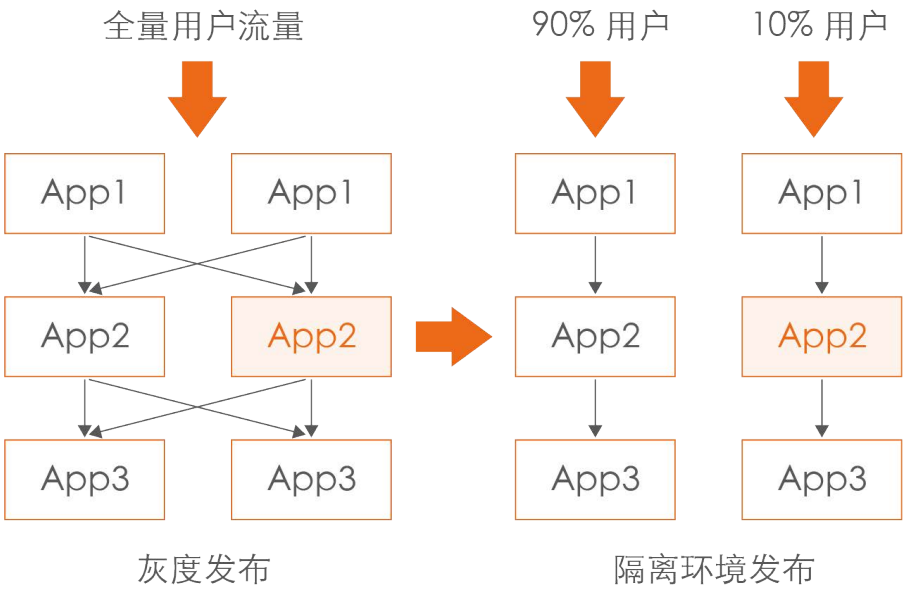
1. 用来做业务探索和创新测试
2. 需要对多个方案进行决策

06 流量隔离环境发布

在上述的发布策略中，发布的单位都是应用，但是一个功能模块往往是由多个应用组合在一起提供的服务，即使当前发布的应用出现了异常，这个异常也未必体现在当前应用中，在复杂的情况下，异常会延迟到它的下

游应用才体现出来，如何发现此类问题并且不影响用户体验是非常重要的。此外，我们有时候还希望新版本的代码上线以后，只影响到一小部分用户。而传统的灰度发布，因为无法识别业务流量，所以即使某个应用只有一台机器出现了问题，也可能会影响到所有的用户。

如下图左侧的灰度发布，App1 的所有机器都有一定概率会路由到出现问题的红色 App2 机器上。 而右侧的隔离环境发布中，新版本的代码会先发布在全链路隔离环境中，即使发布中出现问题，也只会影响少量用户。



优势：

1. 能够发现一些复杂的，涉及到多应用的问题

2. 出现故障时，只会影响很小一部分用户

不足：

1. 需要对流量隔离环境进行独立监控

2. 系统设计复杂，需要中间件和链路上的所有应用能够识别业务流量

适用场景：

1. 较为核心的生产业务场景

阿里巴巴发布最佳实践

我们将按照发布的过程来介绍阿里巴巴发布的最佳实践。

01 发布计划

发布前要对待发布功能模块做充分验证，同时要思考假如本次发布引入故障该如何止血。所以在发布之前写出本次发布的计划清单是非常重要的，一个典型的发布计划如下：

a. 本次发布参与人

开发人

测试人

代码 Review 人

b. 发布内容

c. 测试过程

d. 风险描述

e. 线上验证方案

f. 线上出现问题的止血方案

g. 发布步骤

分 x 批发布，前 x 批发布后暂停 x 小时

02 不同环境使用不同的发布策略

前面介绍的几种发布策略都有各自的优缺点，要根据自己的场景特点和需求选择合适的发布策略。

一般来说，测试环境是用来做初步功能测试，所以会频繁的更新代码和发布，如果采用灰度发布的方式且发布的批次设置的比较大，则开发效率会大打折扣。这个时候单机或多机的单批次停机发布其实是一个不做的选择。

对于预发环境，不仅要考虑自己测试的需要，还要考虑上下游其他开发者的测试需求，所以单批次停机发布就不再合适，可以设置两批发布。

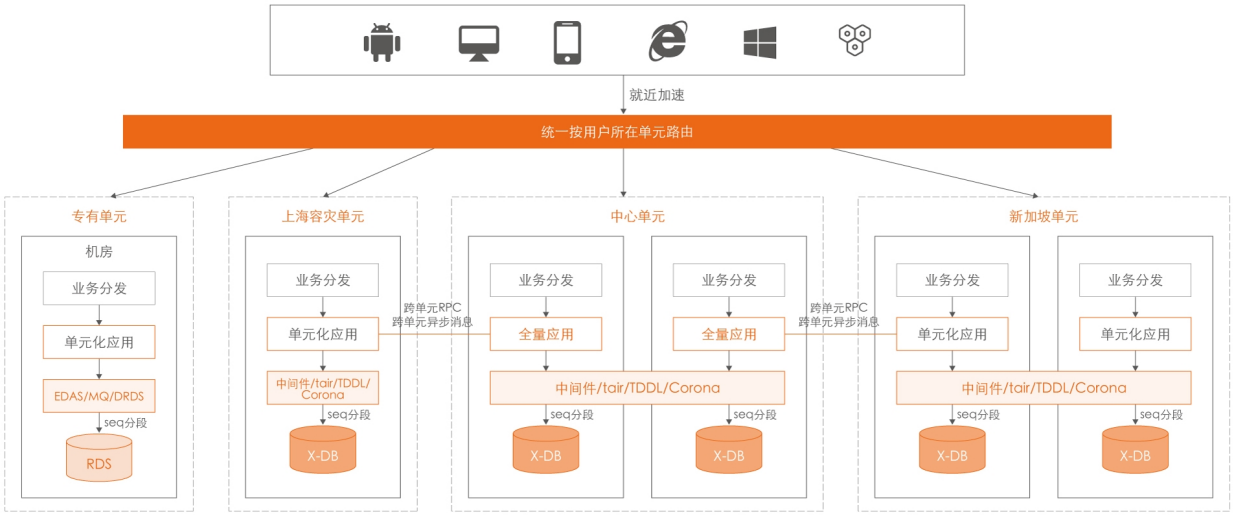
对于线上环境，可以先发布隔离流量环境，再多批次发布线上环境。

03 发布中关注监控报警

仅靠发布策略是无法避免故障的发生的，在发布中和发布后仔细的观察应用的监控数据非常重要。应用的核心指标监控数据，比如 QPS、RT、成功率和报错数，能够帮助用户尽可能早的发现故障。此外，在生产环境中，如果批次数量设置的比较小，每批发布机器数量比较少，那么即使某些监控指标出现了问题，因为数据量比较小，可能会被淹没在整体的监控数据中，所以配置已发布机器的独立监控也是非常重要的。

04 金丝雀发布和无人值守

阿里内部绝大部分应用会在多机房/单元部署，可能存在一种场景，同一份代码和配置在某些机房/单元正常，在其他的的单元/机房下就会出现故障，所以有必要在分批发布的时候，把所有机房/单元的组合都在第一批发布时出现，这样问题可以及早暴露。此外研发人员往往会重点关注前几批发布，如果后面批次才出现问题，研发人员可能无法快速响应。



单元化是为了解决容灾和扩展性问题，上图是阿里巴巴的单元化部署架构.

此外，应用的监控项一般都很多，在发布周期比较长的情况下，不能要求研发人员时刻专注每一个监控项，需要一定的智能化方案帮助研发找出那些需要重点关注的监控项。

为了解决上面两个问题，阿里设计并实现了自己的金丝雀发布策略。金丝雀发布从应用的每个机房/单元下抽取 10% 的机器放到首批，无人值守智能监控系统会对这部分机器设置独立的监控，对于每个监控项，无人值守会对比已发布和未发布机器的监控指标数据，同时对比发布前和发布后的监控数据，如果发现异常，会推送给研发人员做进一步的判断。

这种金丝雀发布策略可以帮助研发尽可能早的发现问题，并且减少研发人员的工作量，提高研发效率。

05 持续集成和发布

合理的选择发布策略，按照上面所述的最佳实践来发布，发布的风险可以被控制在很小的范围内，甚至比停机发布的风险还要小。实际上，发布周期短，每次发布仅包含少量代码是一个很好的发布实践。因为部署间隔时间长，将会导致每次的部署包含更多的代码变更，结果就是出现更多缺陷和宕机的风险。这种情况下，人们为了降低发布风险，会倾向于增加更多的评审，事实上这除了大大增加部署时间外，对降低发布风险的影响微乎其微。这是一个越来越差的增强回路，我们需要通过高频的持续部署，来颠覆这个恶性循环。

总结

敏捷开发能够缩短产品走向市场的时间，让消费者更快的获得想要的功能，也能让产品团队更快的拿到消费者的反馈并据此对产品做出迭代。为了解决敏捷开发下频繁发布带来的发布风险，本文介绍了多种发布策略，包括各个发布策略的优缺点，适用场景，在不同场景下综合应用这些模式可以在更快速地交付高质量的产品。

04

4.5 编排运维

阿里巴巴应用运维平台已经发展了 6 年有余，支撑了公司绝大部分应用的上线部署、扩缩容、资源管理以及各种运维变更操作，并逐渐沉淀出一套丰富且稳定的运维原子服务。为了最大化这些原子服务的价值并打造应用运维平台的中台能力，我们提出了一种面向编排的运维解决方案。

面向编排的运维是指用户（PaaS 服务以及开发、运维、运营等角色）根据实际业务需要，对多个原子组件通过简单编排的方式进行灵活装配，构造出不同的业务流程以便完成一个完整的运维需求。运维编排可以帮助我们更好地规范、管理和执行自动化运维操作，以模板的方式定义所需要进行的操作，然后再通过系统运行，从而提高整体运维操作的效率、增强运维操作的安全性，并避免人工运维的错误。

主要痛点

在应用运维领域，大部分的做法都是基于工作流以及工单管理来实现对应的运维变更操作，而传统的运维工作流在维护成本及可扩展性上都存在一定的不足，缺乏有效的流程生命周期管理手段。

这些问题可以归结为以下三类：

- 随着业务的不断发展和业务场景的愈发丰富，运维业务自身也变得越来越复杂，经常会出现一些非通用的个性化需求，比如在扩容流程中新增一个第三方数据同步的步骤，或者针对同一变更类型，不同环境需要执行不同的运维流程。这些需求导致平台实现成本以及维护成本越来越大。
- 依赖的底层流程引擎在运维领域的支持有限，组件编排和流程管控等能力不易进行扩展，同时在规模化场景下，性能、稳定性以及安全性等方面也很难得到有效保证。

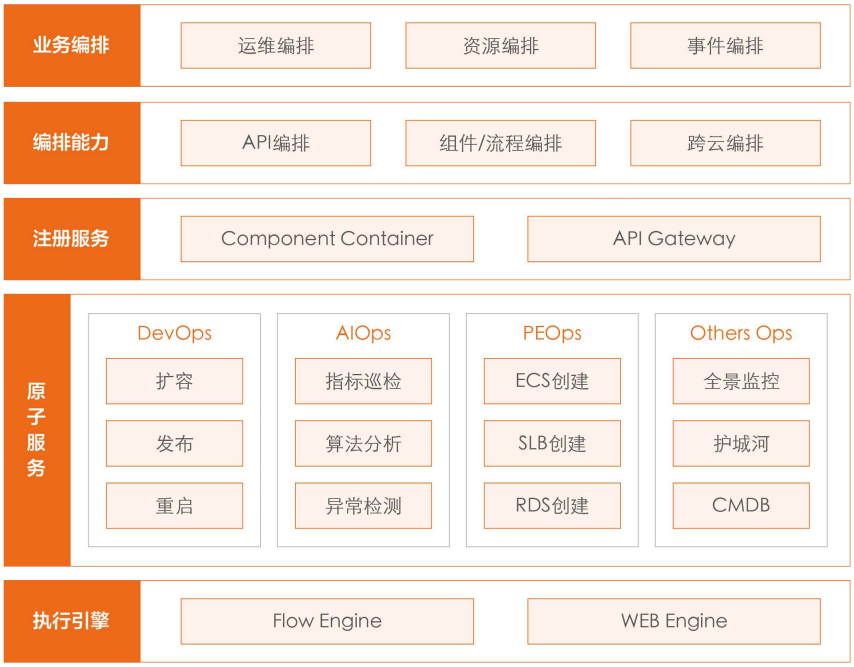
- 传统运维平台不具备统一且标准化的集成与被集成能力，难以赋能其他运维 PaaS 产品，中台能力欠缺，价值渗透有限，同时开发或运维人员缺乏设计和管理定制化运维操作的手段。

01 核心理念

运维编排的核心理念是服务组件化、运维编排化。我们把运维原子服务按照平台规范注册为组件，并托管到统一的组件池中进行维护和管理，用户按需从组件池中选择对应组件，并采用合适的编排方式装配成运维业务流程，最后触发执行即可完成期望的运维变更任务。运维编排的最终目的是打造一款高效、稳定、安全的运维业务构建平台。

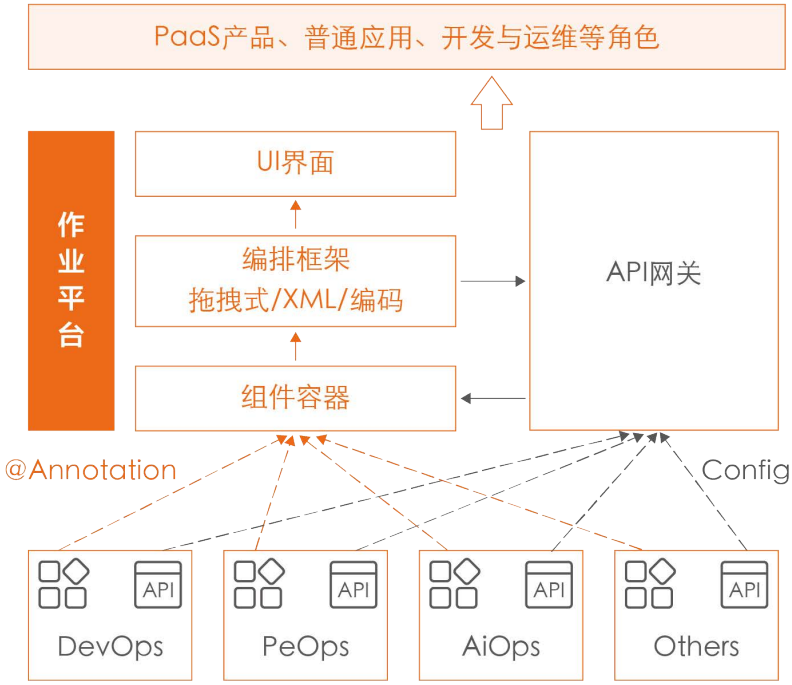
02 技术思路

业务架构



架构一共有五层，从下到上，第一层是流程引擎以及容器引擎，作为原子服务的执行者；第二层用于定义各种不同的运维原子服务，是原子服务的定义者；第三层则主要用于注册原子服务为组件，作为组件的注册者；第四层是提供核心的编排能力，作为流程的编排者；第五层主要提供场景化编排能力，针对不同的场景有一些额外的特性支持。

技术架构



被集成服务可以向 API Gateway 注册 Rest API，从而通过统一的网关对外暴露服务。网关本身需要实现标准的鉴权/授权策略以及 API 生命周期管理、熔断和限流等能力，同时注册到网关的 API 还要能够进一步注册到作业平台的组件池当中；如果被集成服务还引入了流程引擎，那么对应的原子组件也要能够直接注册到远程的组件池，最终通过作业平台完成所有原子组件的收敛和统一管理。基于此可以让业务方按需从组件池中选择对应组件并进行装配，同时通过自定义表单功能设置流程输入，最后触发流程。流程执行时由作业平台的执行引擎子系统进行远程调度并驱动最终的服务提供者运行对应的功能组件。

核心功能组件

- **编排引擎**：通过流程引擎、表单引擎、规则引擎以及脚本引擎等驱动运维业务的制作与执行。
- **中台网关**：规范组件接入标准，同时通过统一的服务网关集成丰富多样的运维原子组件，提供给第三方或者平台编排使用。
- **安全保障**：由编排生成的业务流程默认集成审批流、安全风控、无人值守以及多种巡检能力，为运维变更提供全方位的安全保障机制。
- **支撑服务**：提供企业主数据、消息中心、通知中心、任务中心、权限中心等业务支持服务。

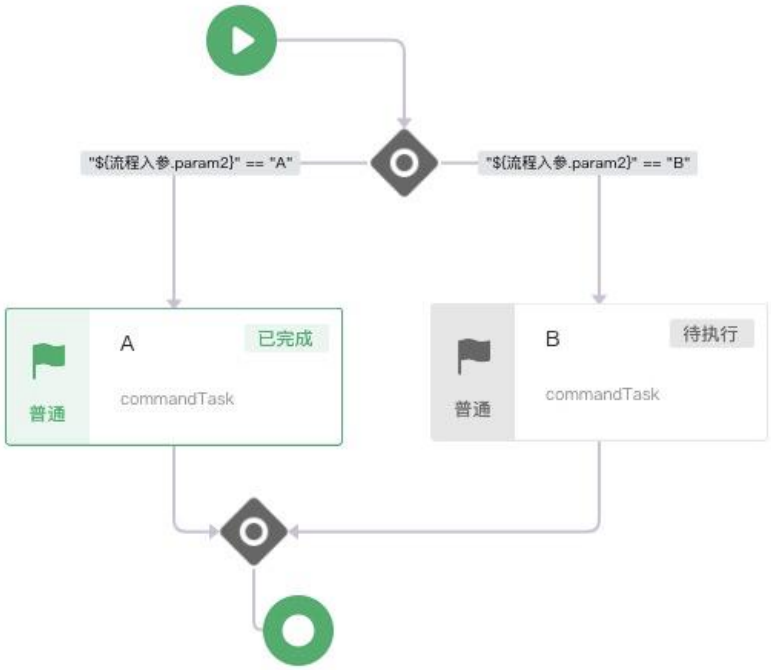
使用步骤



关键能力

主要包括以下关键能力：

- **快速功能扩展构建能**力：运维编排提供丰富的运维基础组件，以及常见运维场景的公共模板，用户可以通过复制公共模板并对其修改，快速地构建模板，实现特定的运维需求，降低模板编写的难度，提高整体运维的效率。
- **快速集成第三方运维能**力：用户可以把第三方运维能力通过 API 网关，包装成运维编排的功能组件，在运维编排中使用，实现第三方运维能力的快速集成。
- **被第三方平台集成的能力**：第三方平台可以通过运维编排中心的 API，进行模板和流程的管理，通过订阅流程事件来监听执行过程，实现运维编排被集成的能力。
- **管理运维脚本/文件**：运维平台统一对用户的运维脚本或文件进行管理，包括脚本或文件的上/下线、版本管理、授权管理等。
- **可视化的执**行**过程和执**行**结果**：通过提供可视化的执行过程，用户可以看到完整的执行过程和执行结果，具体包括：
  - 直观地看到各个任务的执行详情；
  - 清晰地看到执行的流程、顺序和错误跳转。



03 适用场景

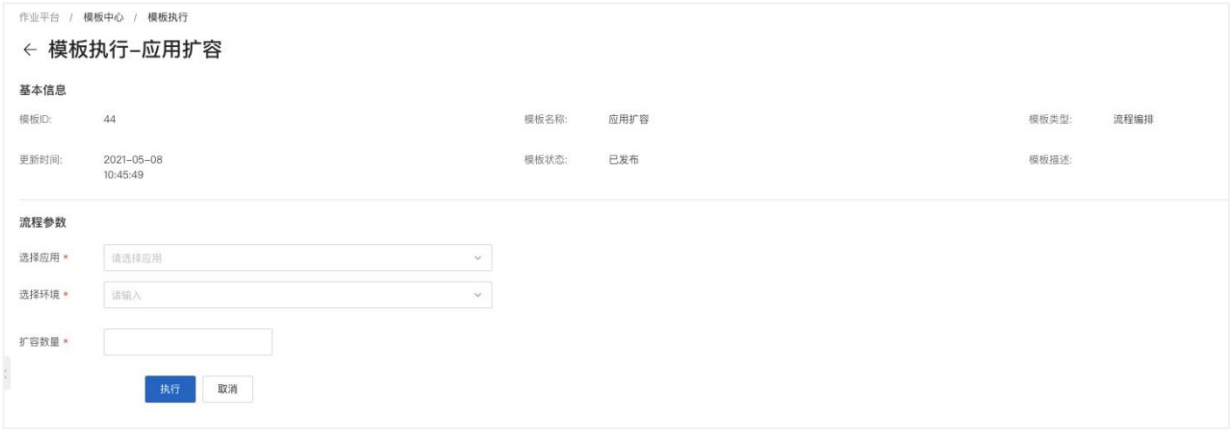
- **扩展现有运维变更业务**：针对运维平台现有的运维变更操作进行调整，以便满足业务方所在部门的特定需求。
- **定义全新的变更类型**：针对运维平台当前并未提供的变更类型（比如 IoT 场景的运维），业务方可以根据自身需要把相关的原子组件注册到平台中，然后通过编排方式构建出全新类型的运维操作流程。
- **批量主机运维**：选定一批主机并按照编排的顺序执行一系列的运维脚本或者命令，以达到批量操作主机的目的。
- **定时巡检任务**：通过定时组件结合自定义流程可以对线上资源或者服务进行各种不同维度的数据巡检以及结果报告。
- **运维编排器**：用户使用该平台，把自己的 API 以自定义 HTTP 组件的形式进行编排，进而快速编排出所需的运维功能，减少开发工作量。
- **主机运维**：用户通过主机运维组件，实现对主机的日志清理、组件管理等。

以应用扩容为例

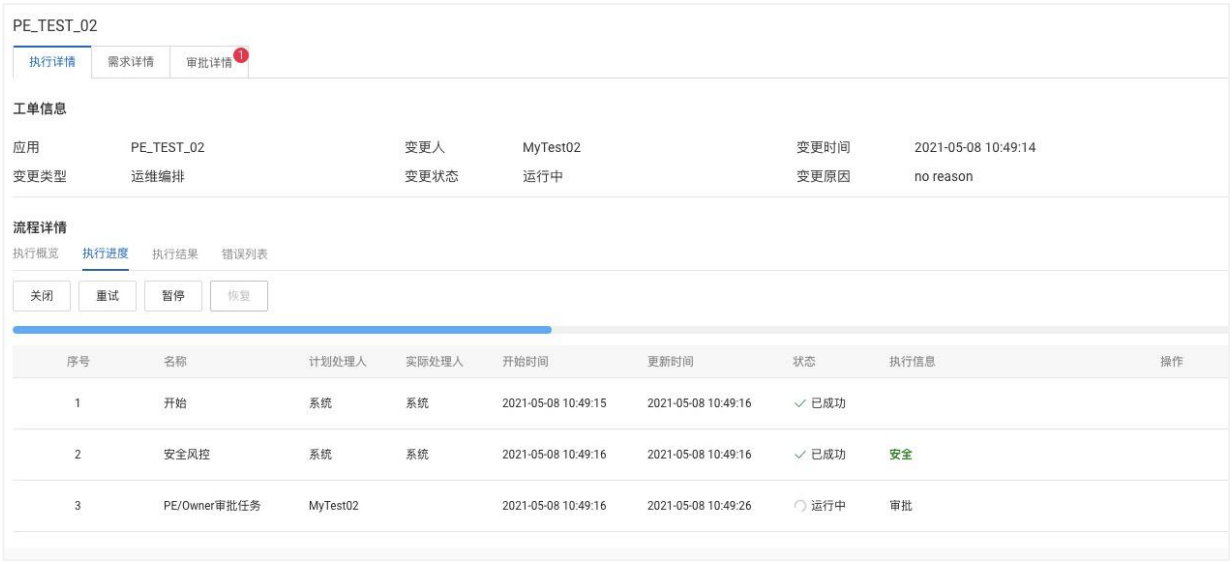
1. 可视化编排出应用扩容模板



2. 提交表单，执行应用扩容模板



3. 查询执行进度及结果



总结

面向编排的运维业务构建模式，可以高效、灵活、稳定地支持企业中的各种运维业务场景。围绕企业业务管理需求，通过可视化的用户编排界面、控件元素和成熟稳定的模块组件，面向编排的运维工具可以支持团队快速搭建轻资产、高效能、个性化的 IT 运维工具，助力传统运维转型，加速企业数字化进程。

04

4.6 智能运维

阿里巴巴的运维团队致力于打造无人值守的运维平台，用智能化推动高效率、低成本的应用运维。智能运维是运维平台实现信息化和数字化之后的自然发展，利用扎实的技术基础，把机器学习、优化算法和各个专业领域方面的知识完美结合起来，针对具体运维场景提供令人满意的解决方案。

智能运维（ AIOps ）是依托于阿里巴巴 DevOps 经验沉淀而来的智能化运维平台，通过运维大数据的积累，以及算法团队多种算法的校对，我们将运维提升到新的高度，通过 AI 来帮我们查看数据、判断异常、决策运维操作，形成监、管、控一体化的运维平台。

运维体系面临的挑战

DevOps 运维时代，阿里巴巴运维体系面临如下挑战：

第一，规模化。阿里巴巴的基础设施规模呈指数级增长，在服务器数量是千级别、万级别的时候还勉强可以通过人为操作来运维，但发展到百万级别的时候，任何一个步骤依赖纯人为操作都是不现实的。服务器规模百万级的时候，如何保证整体运维安全、高效的进行是第一个挑战。

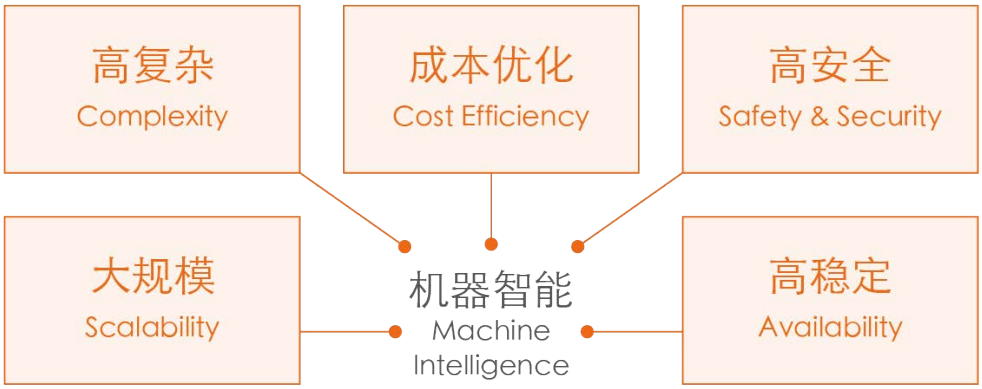
第二，高复杂。阿里巴巴业务的多样性及高速发展也对系统稳定性提出了更高的要求，对运维体系带来更大的挑战。曾经我们考核系统可用率 7 个 9，存储方面达到 6 个 9 就很好，但是盒马鲜生这样的业务是要求 100% 可用。作为线下业务，在盒马店有半小时不能支付是无法接受。我们要从全链路视角出发，关注每个环节的稳定性建设。

第三，成本优化。成本是门槛，做不到一定的门槛，进入这个市场的机会都没有。除了固定资产投入，运

营成本也是很重要的一部分。利用技术进行流程优化，降低各个部分的成本，是提高业务的核心竞争力的关键。

第四，安全。云计算最关心的是安全。系统越来越大，变化越来越快，所面临的内部和外部的风险也越来越大。每天无数变更升级同时进行，如何在系统变更时保持稳定，是需要面对的另一个巨大挑战。

阿里巴巴基础设施的体量和复杂性，显然都超过了人脑的处理能力，需要从新的视角，应用机器智能来解决这些复杂的问题。



■ 智能运维实践

基于上面的挑战，我们在阿里巴巴集团各个业务场景落地了无人值守发布、无人介入运维的解决方案。

01 无人值守发布（Unmanned Deploy）

全新一代发布平台支持滚动、蓝绿、金丝雀等多种发布模式。通过算法，机器学习方法对应用发布过程进行异常检测，从而避免由于代码变更导致的故障。基于大量监控数据、日志数据的积累，并有算法的加持，我们推出了无人值守发布系统。

无人值守发布 riskfree 系统上线以来，从探索到实现再到优化经历了将近三年的时间。目前业务范围定义在应用发布时故障预防。接入无人值守发布的应用在提交发布单后，系统会对整个发布过程中的监控数据进行分析，如果有异常会自动暂停发布，并提示异常指标和拦截原因，开发确认有问题则可以选择关闭或回滚，没有问题则继续发布。

线上发布之痛

以往线上发布的时候，工程师们一般做了如下“精心”工作：

- 发布前

测试人员对代码进行全方位的单元测试、集成测试，如果发现 Bug，会让开发人员返工。这里有两个问题：第一，有些业务团队由于人员问题，根本就没有测试人员，自己既是开发也是测试；第二，不是所有的 Bug 都能通过测试发现，难免有漏网之鱼。

- 发布中

进行预发、灰度、分批发布、金丝雀发布。在每一个环境缓慢发布过程中，要到监控平台，查看各个监控，甚至登录到机器上“刷”日志，通过自己的“火眼金睛”，期望能在众多的日志中，找到某个特殊模式的异常日志；另外，如果是多方依赖的应用，还要查看上下游的应用监控有没有问题。

- 发布后

检查一下应用的机器是否都正常启动，将失败的机器下线或者置换掉，看看故障系统有没有报警，看看上下游团队有没有“叫”起来，如果有，得马上回滚。总之，这个过程是既耗时又耗力，而且还不能保证没有漏掉一处细节，并且不同发布人员的经验不一样，熟手和新手对一个发布的稳定性保障程度有巨大差别。

我们的解法

我们设计了一套无人值守发布系统



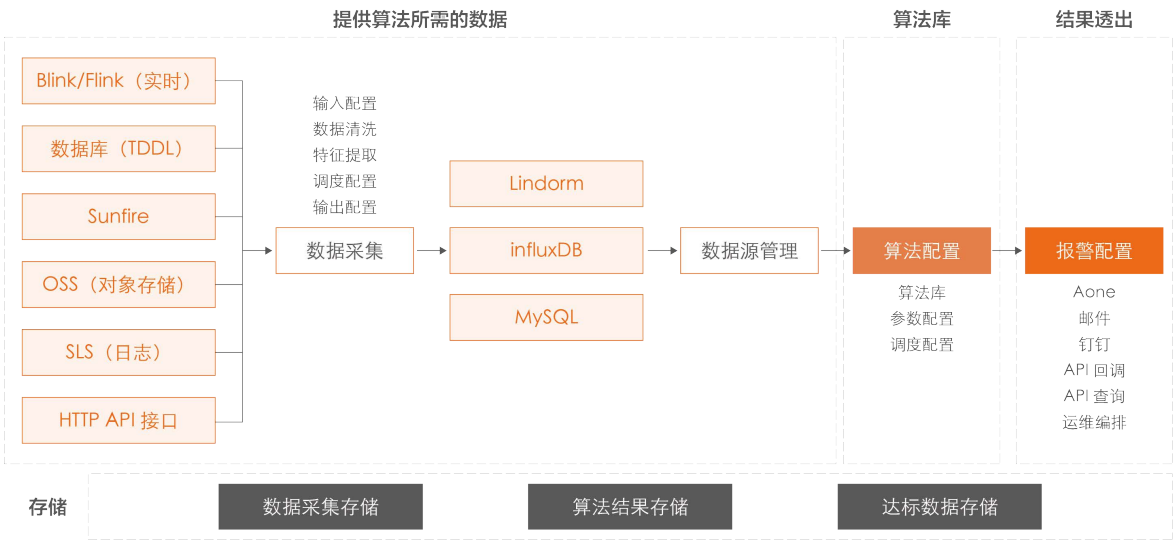
系统分为两大部分：

1. 在线分析，无人值守发布系统会对系统监控、业务监控、日志监控、调用链路等维度进行异常检测，检测到异常后，会对发布单进行拦截或回滚。当用户认为无异常时，会进行反馈，继续发布。

2. 离线分析，在第一步中用户反馈后，这个反馈数据对我们的算法非常有用，可以对我们的算法进行自动的调整。当反馈数据积累一段时间之后，异常检测的准确率就非常高。

● 算法平台

在发布的过程中，系统会采集各个监控源的数据，对数据的采集、清洗、存储要求很高，我们设计了算法平台来承接各个平台的数据源、算法检测、算法验证、算法上线等流程，系统架构如下图所示。



主要包含三大部分：

1. 数据采集存储：对各个监控数据源的数据进行采集，包含系统监控、业务监控、中间件监控、日志监控、数据库监控、云监控等。数据采集后，根据不同数据的特性，存在时序数据库或者关系数据库中。

2. 算法结果存储：对于每次检测的结果都会进行存储，以方便结果排查和效果评估。

3. 数据打标：对每次异常检测结果，都可以打标，利用打标数据来重新训练算法，形成正向循环，检测的结果也可以通过邮件、钉钉实时通知给发布者，且可以自动对接前面介绍的运维编排自愈流程，比如，将异常的机器直接自动置换掉。

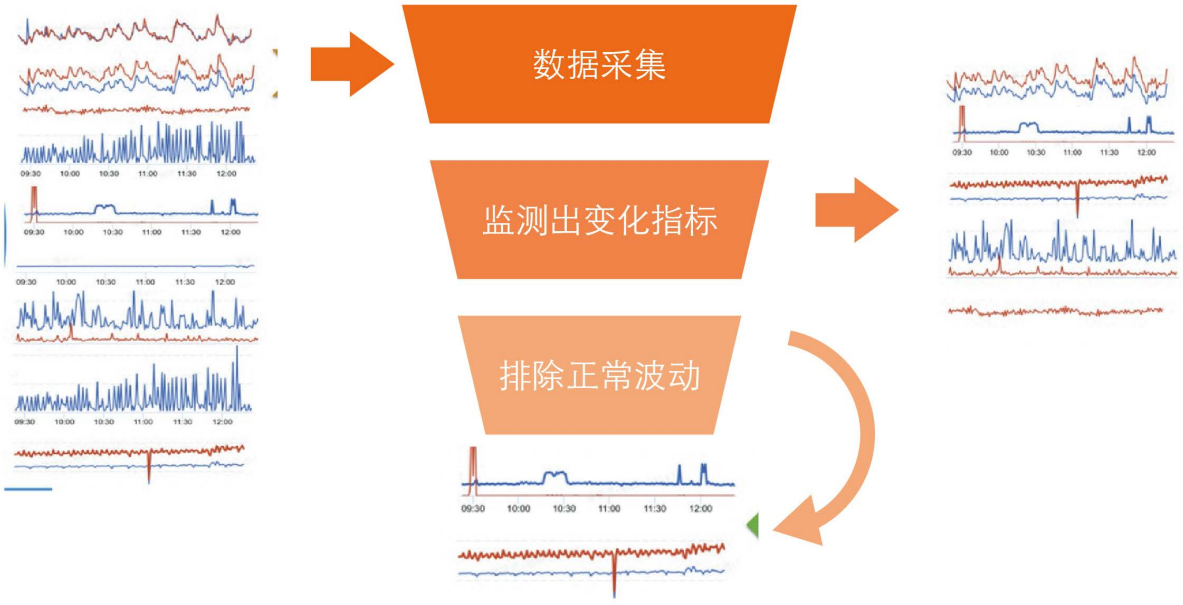
智能算法

在上面的算法平台中，我们设计了众多异常检测算法。异常检测在无人值守发布系统中有着举足轻重的地位，主要分为三个部分：

1. 数据采集：我们综合了各个维度的监控数据、调用链路分析等，在观测的广度上是人工盯屏所不能相比的。

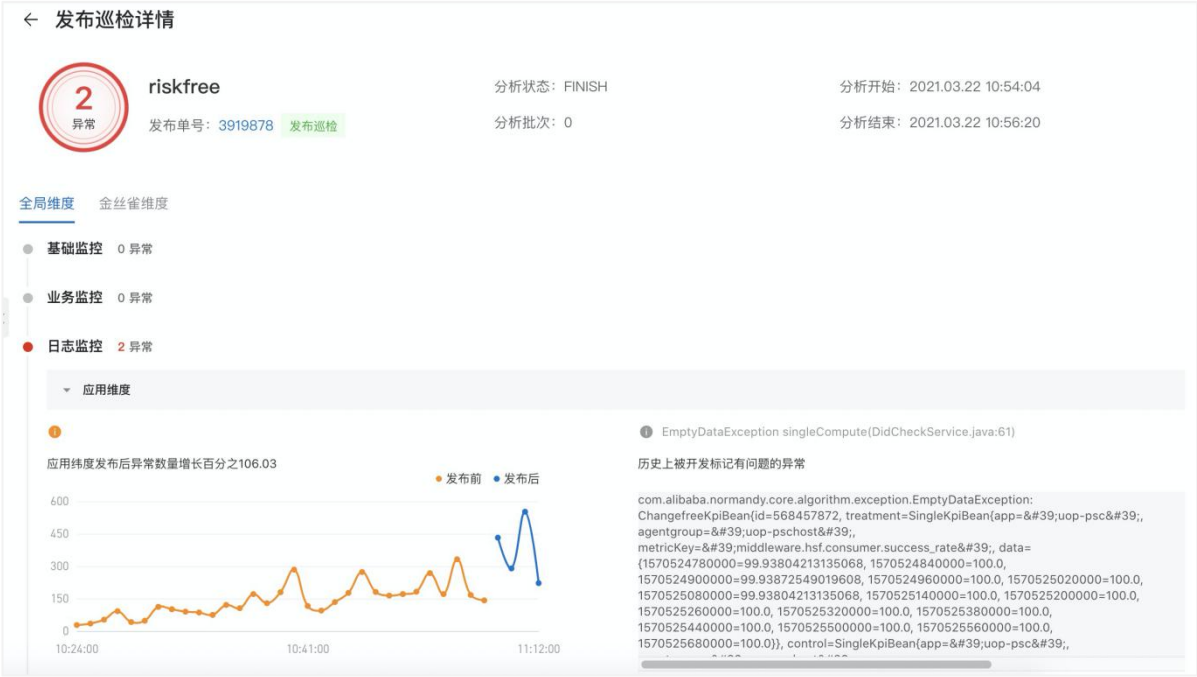
2. 异常检测：我们精心调校的异常检测算法，完全不依赖于传统的基于阈值、3Sigma 等检测算法，全部自动判定，泛化能力好，支持单指标检测、多指标检测、前后对比检测、已发布未发布对比检测等多种模式，检测算法包含 ArimaKSigma、BoxplotDetect 方法（Tukey 方法）、GrubbsTest 方法、Donat 等。

3. 排除正常波动：通过历史数据、用户反馈数据，精准过滤正常的波动，让用户得到精准的异常检测结果，示意图如下：



实践效果

无人值守发布自上线以来，覆盖了阿里巴巴集团所有的应用发布过程，为发布的安全稳定保驾护航，异常检测结果如下图所示：



至此，接入无人值守发布后，开发可以在点击发布后专注别的事情，不需要时时关注发布过程。如果发布过程出现异常，系统会通过钉钉消息、邮件通知到开发，再介入即可，如果过程是机器异常则自动替换异常机器，开发无需人工介入，发布将继续。

简单总结一下，无人值守发布是一个智能化变更故障检测和异常推荐系统。通过对变更执行过程中的多维度监控数据进行分析，判断当前变更是否会造成故障，在发布出现异常的情况下进行拦截和智能推荐。

02 无人介入运维-ChatOps (Unmanned Operations)

日常运维有很多类别，目前我们专注于其中两类运维工作的“无人介入”：1. 用户接到告警或事件而发起的运维操作；2. 日常运维答疑或咨询。

针对第一种情况，通过“运维诊断”，给应用来一次 360 度全方位“体检”，找到异常点并一键修复；针对第二种情况我们发布了 ChatOps 机器人来加强 DevOps 之间沟通与合作，帮助研发完成一些“脏活”、“累活”、“机械式”任务，目标是达到“0”人工介入的咨询和答疑。

ChatOps 简介

运维小蜜(简称“维蜜”)，是 chatbot 在运维领域的实践，也是 ChatOps 的具体实现，是 DevOps 的重要工具。维蜜的定位是面向应用的智能 DevOps 服务助理，那么摊开来讲：

- 面向应用：将应用的开发、测试、运维的同学集合起来，加强沟通与合作，缩短产品上线时间，降低人力成本，在产品出现问题时能够快速检测并修复，减少甚至消灭产品服务中断可能性，保证开发和运维的同学时刻处于同一个上下文中，时刻了解应用所处的状态。
- DevOps：强调快速迭代，持续交付，力求信息共享、技术学习与合作、加快信息反馈周期。
- 智能：理解用户输入的指令，根据命令槽位的元信息和用户自身的信息确定命令各个参数的值，通过自然语言处理和理解用户指令。

维蜜就是希望通过一触即达，秒级响应的体验，把服务做到极致，让研发、测试、运维同学幸福地工作，是我们的终极目标。

ChatOps 优势

我们再来看运维小蜜有哪些价值：

第一，从员工个人的角度来看，能够提升员工的工作效率。运维小蜜可以帮助用户处理简单、重复、枯燥的工作，例如日志查看、命令执行、开关报警、查看机器状态、查看监控、运维事件推送等。

第二，从团队沟通的角度来看，能够降低协作成本。在团队内部，ChatOps 是一种透明、合作、会话驱动的开发模式，群里所有人都知道 what/when happening & who/how fixing it ，也就实现了事件发生场景完整、透明，事件解决过程共享、可查询、可记录，便于其他同学对同类事件处理的学习参考，即所谓“Teach by doing”。

ChatOps 也是一种会话驱动的运维模式，通过聊天机器人对接各种系统后台，将软件开发、交付过程中涉及的开发、测试、运维人员、工具、环境、自动化进程等串联起来，使得聊天室里的所有人能够围绕某个特定话题进行信息共享、技术学习与合作，加快应用的测试、发布、监控、诊断，整个工作的展开全员可见。

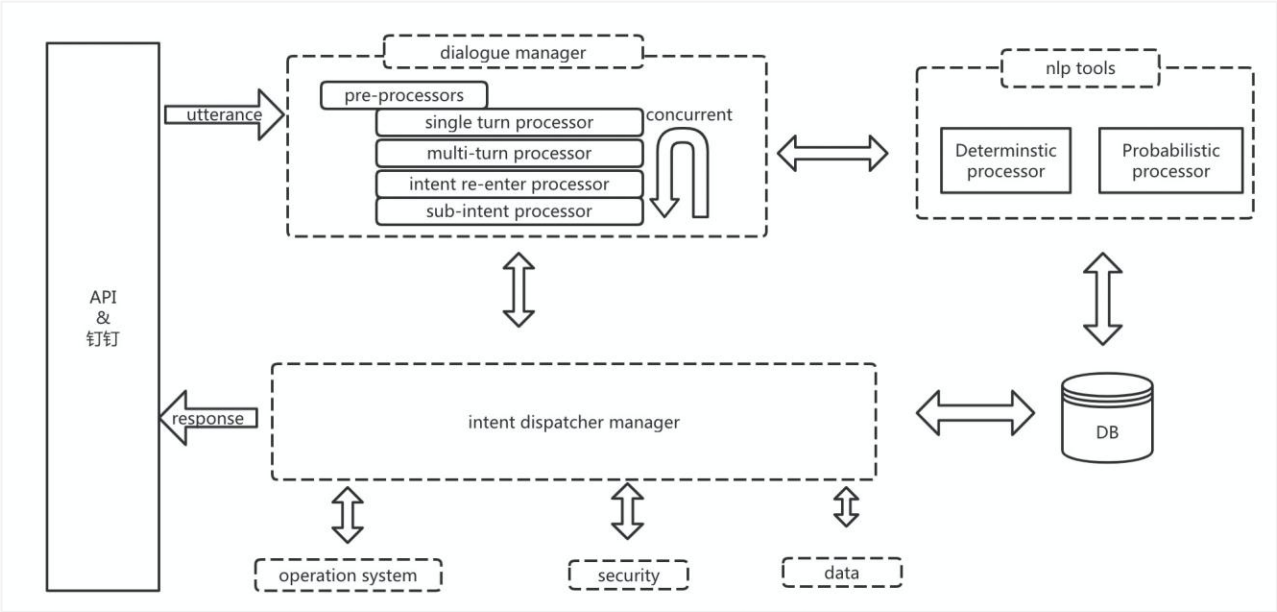
运维机器人带来的好处包括：

1. 方便，把很多系统的常用操作聚合到机器，就不用登陆多个系统找信息。
2. 协作，事件发生的全部信息推送至聊天室，所有成员均能够了解你这儿发生了什么。
3. 快速，定位问题时，能够让大家都看到所有的信息，不必让每个人重复的搜索资料。



ChatOps 实现

我们再来看运维小蜜的实现架构图：



主要包含三个模块，分别是 dialogue manager 、nlp tools 和 intent dispatcher manager 。其中 dialogue manager 用于判断用户的 utterance 的意图是什么，是发起一轮新的对话还是承接上面已有的意图，它调用 nlp tools 的处理器辅助判断。intent dispatcher manager 负责对接具体的业务系统，dialogue manager 处理后的结果传到它调用具体的业务逻辑触发任务的执行。

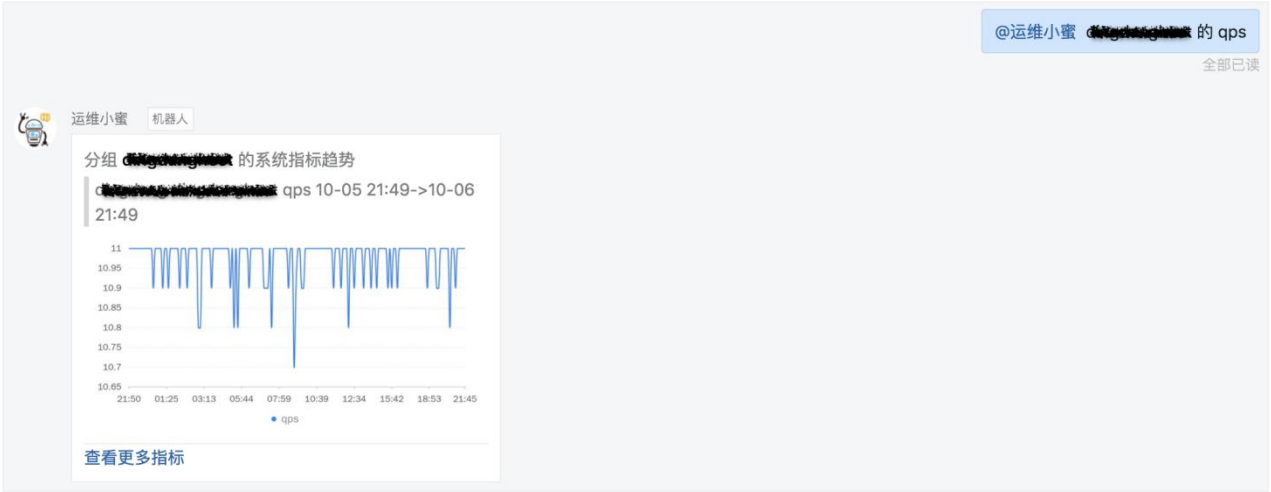
ChatOps 实践

我们再来看运维小蜜在阿里巴巴集团的几个落地场景：

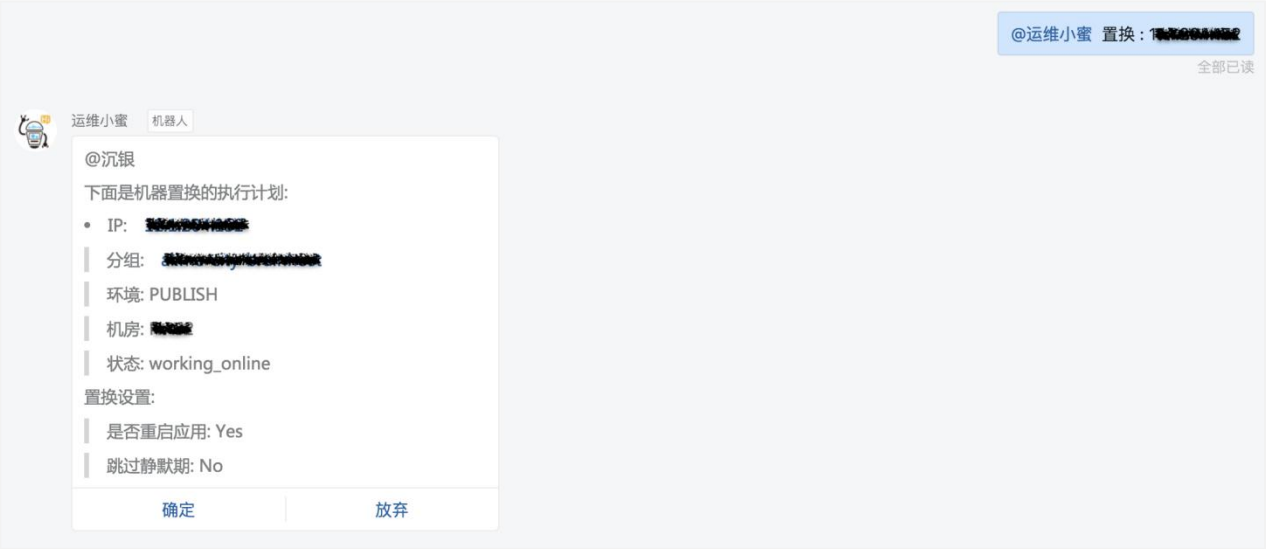
1. 智能问答



2. 查询应用的监控信息



3. 机器置换



总之，ChatOps 可以帮助我们提升开发效率，提升开发幸福感。

总结

随着智能化算法的成熟和大量运维数据的积累，智能化在运维场景的落地也会越来越多，阿里巴巴运维从阿里集团大量研发场景出发，打磨出一系列智能化运维产品，并赋能中小企业。我们的理念是把复杂留给自己，把简单留给用户。智能化是运维的终极状态，未来我们将在自动化、无人化和智能化上做更大的投入，打造世界级的智能运维平台。

阿里巴巴DevOps
工具体系

05

阿　里　云　云　效

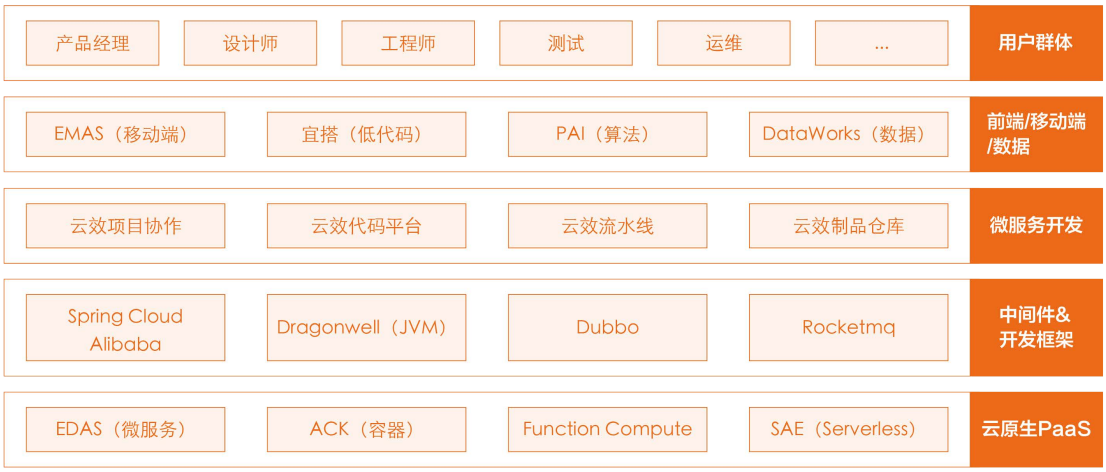
05

5.1 阿里巴巴 DevOps 工具体系

随着阿里巴巴多元化业务 20 多年的高速发展，技术体系经历了 web 时代、移动化时代、数据智能时代、云计算时代等多个重大变革。在这些变革中，开发者面对的技术体系、工具体系、知识体系也在不断进化。研发工具在其中起到了技术规模化和降本提效的关键作用。

工具体系总览

通常企业中技术人员会按照技术工种分为前端、移动端、服务端、数据、算法、测试、运维等多个角色，这也代表着当前软件工程领域的几大技术分工。每种技术栈都有自己独有的技术发展路径和配套工具集，在阿里巴巴除了这种纵向的技术维度切分以外，还存在按照用户感知路径从前往后的横向切分。比如偏向业务侧的 no-code/low-code 编程，偏向通用侧的 pro-code 编程等。



研发工具体系发展大体分为：技术栈标准化、工具流程平台一体化、细分场景技术多样化三个主要阶段。

在一种特定技术领域发展初期或者公司刚成立之时，会出现技术框架百家争鸣，多种研发流程并行的情况，通常主流技术栈收敛是提升研发效率的第一选择。比如阿里开发中 Java 技术栈人员占比超过 50%，基于 Java 技术栈演进出的中间件、编程框架、配套工具，以及研发流程会高度耦合，形成统一研发解决方案。

解决方案的产品化会诞生一体化的工具流程平台，而此平台对企业的核心收益在于将固有流程标准化和自动化，抬升了所有技术员工的技能底线，从而提升平均人效。另一方面工具平台可以帮助企业积累可用资产，并将过程数据进行汇总分析，为管理者提供决策依据。

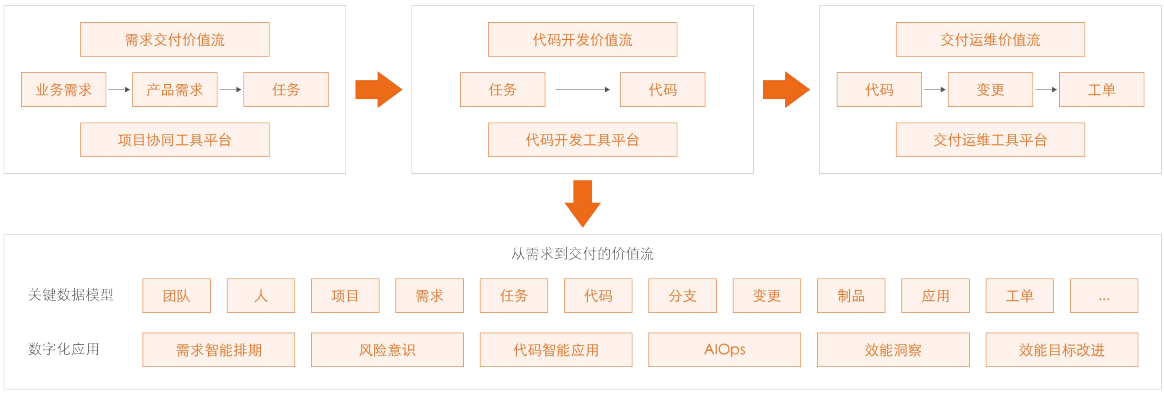
研发工具发展的第三阶段是与企业业务深度耦合和定制后的场景化，实现特定领域的效能突破。比如 OA 领域的无代码编程、前端智能化 P2C、服务端函数编程等。

阿里巴巴 DevOps 平台

我们通常所说的 DevOps 是计划、代码、开发、测试、发布、运维、监控的全流程，分为三大阶段：需求分析阶段、代码开发阶段、交付运维阶段，分别对应以需求为中心、以代码为中心、及以应用为中心的三个工具平台。

平台首先需要解决的是如何管理企业研发类资产的问题，通常分为知识类资产（需求、文档、设计图等）、代码类资产（程序、配置、数据等）、应用与资源类资产（实现对外服务的逻辑单元以及背后的物理资产）。其次需要记录研发过程所产生的数据，用于分析寻找提升效率的路径。

工具平台会将资产数据和过程数据沉淀到统一的数据中台之上。而串联数据的正是 DevOps 从计划到监控的标准化流程。在阿里我们称之为价值流，代表着一个业务价值从定义到实现的全过程，而这种价值交付的速度正是研发效能。



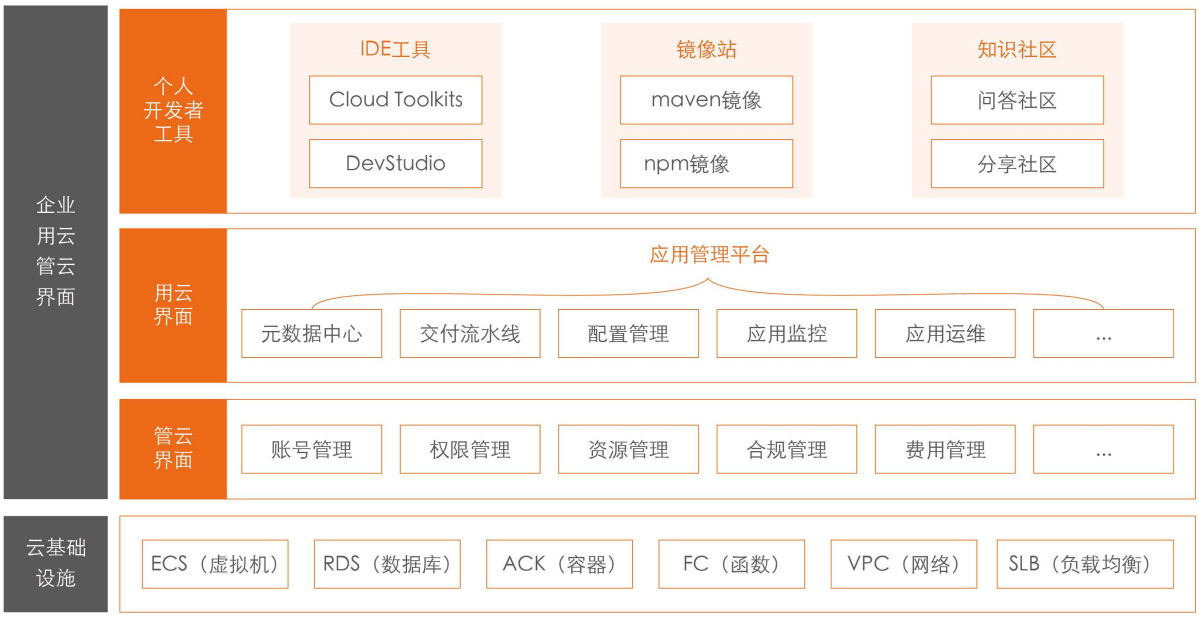
基于“云”的 DevOps 体系

当前企业上云几乎成为必选，建立 DevOps 体系的时候必须要考虑“用好云”的问题。从阿里巴巴的经验来看，“用好云”的关键是给开发和运维两种角色分别建立用云的工具切面。

运维或者 SRE 这个角色是基础设施的创建和维护者，他所关注的是大量零散的 IT 资产，如何管理这些资产，控制其生产和运维流程是最重要的。我们会选择一个基于 ITIL 或者 ITSM 的“云资源管理平台”来帮助运维人员提升管理效率，因此称之为面向“资源”的管云界面。

开发和测试所关注的是如何快速安全的将业务需求转变为线上可以被使用的服务。一个或多个服务的组合我们称之为“应用”，而应用可以运行在一系列云资源之上，因此它会变成一系列资源的逻辑归组。我们会建立应用的开发、测试、运维流程，并将这些流程配置到一个“应用管理平台”之上，这就是面向“应用”的用云界面。

在阿里巴巴，我们通过“云资源管理平台”和“应用管理平台”实现了产研人员与云的有效连接，并通过平台的流程抽象，实现了对云技术细节的屏蔽，提升了各角色用云的效率，并将企业“资源”与“应用”两种最重要的资产沉淀下来。

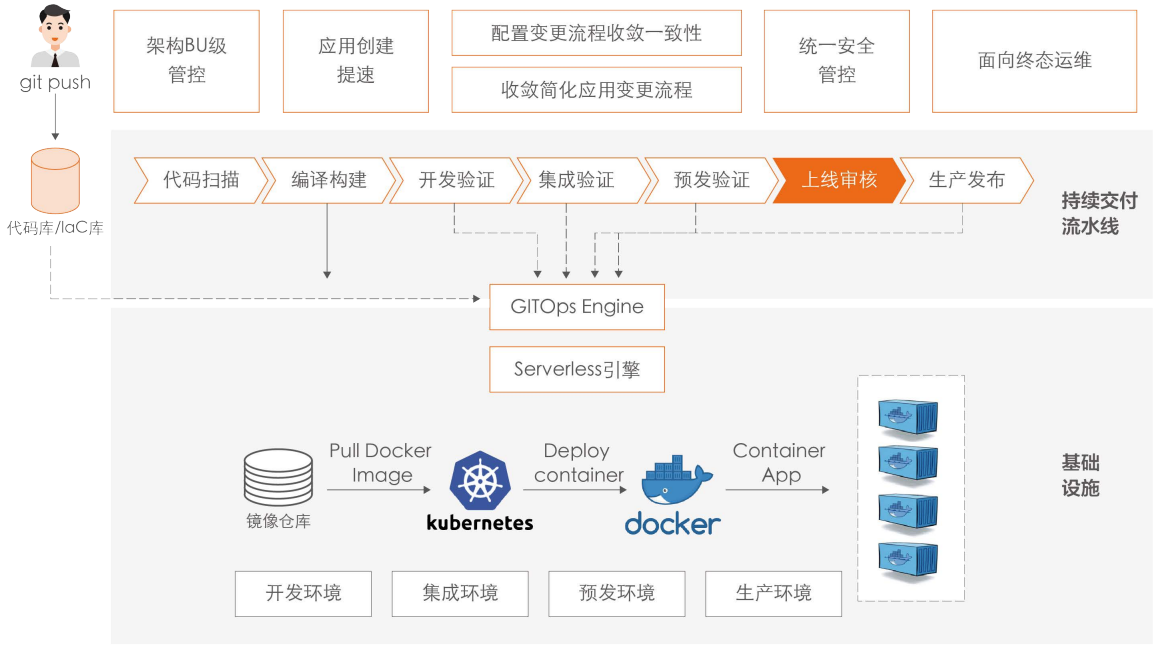


DevOps 工具的云原生趋势

随着 kubernetes、容器化、Serverless、Service Mesh 等完全基于云的技术体系逐步成为业界事实标准，云原生化成为了众多企业技术升级的目标。DevOps 工具体系需要进行升级以适应云原生的发展趋势。

Kubernetes 是云原生的代表技术，首先它从容器编排能力开始不断演化，不但实现了对底层物理资源的有效屏蔽，还发展出非常强的可编程的扩展能力。基于此能力发展出了一些列中间件、运维工具，甚至是编程框架；其次它具有面向终态的特性，这种声明式的资源运维模式与传统面向过程的运维模式有着本质区别，有机会彻底摆脱人的控制，实现无人值守的变更。因此云原生的 DevOps 工具不但需要适配云原生的技术和产品，而且要能够继承面向终态的思想，来进一步提升研发运维效率。

阿里巴巴将 GitOps/IaC 理念与云原生技术相结合，并融合传统应用管理经验产生了新一代云原生研发运维平台。相比传统模式，新平台具备以下几个特点。



应用终态运维

开发人员可以通过代码去描述应用的交付过程和运行时状态，系统根据变更内容自主决定执行策略，将应用状态逐步逼近终态。在此过程中系统可以接收用户指令或者监控数据的变化，来自主改变变更路径，确保系统安全可靠。

分层定义和管控

架构师、SRE、测试工程师、安全工程师都可以对应用的描述代码进行模块化定义，在代码上实现 import 功能，引入各个角色的预定义内容和管控规则。应用负责人可以在规则允许的范围内对应用进行细节定义。这样的分层设计一方面可以减少应用定义的复杂度，另一方面可以满足企业分层管控的要求。

配置收敛统一

将包括交付流程、规则配置、配置项、资源配置等方方面面的内容，通过代码这个唯一形式来定义，可以实现运维定义的收敛，大幅降低开发理解各种云产品的复杂度。而且可以形成统一操作界面，防止不同系统不同权限策略带来的不一致风险。

变更流程一致性

任何配置的变更都简化为代码变更后，可以通过统一的 CICD 流程安全可靠地推进到生产环境。这种流程一致性可以最大程度上保障质量和控制风险，甚至可以为运维变更准备自动化测试用例。

总结

阿里巴巴的业务仍在高速发展，云的技术尤其是云原生领域也在快速成熟，不管是软件开发方法还是工具体系都需要及时应对挑战，不断地降低技术门槛，不断地提升效率，不断地降低风险。同时云原生带来的标准化和开放性，也让阿里云云研发团队有机会不断地将内部的实践产品化，通过“云效平台”对外输出，服务于广大云上开发者。期待我们的这些工具和实践可以让大家“用好云”，并与我们一起分享云所带来的效能红利。

DevOps能力提升模型

06

阿　里　云　云　效

06

6.1 DevOps 能力提升模型

DevOps 能力反映的是技术研发响应业务变化的能力。随着组织规模的增加和业务复杂性增长，DevOps 能力会变得越来越重要。持续提升 DevOps 的能力成为技术研发的共同挑战。

为了给组织的 DevOps 能力提升指明方向，并规划清晰的路径。我们定义了 DevOps 能力成熟度模型，它提供两个价值：1）知道我们今天在哪里；2）如何规划提升路径。

我们将 DevOps 的实施，分解为 4 大类，10 个能力，分别是：

基础能力：包括系统的服务化水平和基础设施水平两块，它是研发和交付的基础。其中，服务化水平跟应用架构紧密关联，最理想的情况是无服务化架构，比较低级的状况是整个系统耦合在一起的巨石架构；基础设施水平体现为研发对基础设施所需要的关注度，需要的关注度越高，研发花在基础设施上的成本越高，效率越低，而且稳定性反而难以保障。

交付能力：包括工具化水平、测试自动化水平和部署发布水平三大块，它是工程能力水平的主要体现。其中，工具化水平指的是研发全流程中涉及到的各类工具的整体水平，包括单点能力（如项目协作工具、构建工具、依赖管理工具、环境管理工具等）和协同能力（如需求与发布的系统、缺陷与测试的打通等）；测试自动化水平指测试的反馈效率和自动化程度，测试自动化是工程能力的重要组成部分，也是提升部署发布能力的基础；部署发布水平是指把制品上线到生产环境并提供服务的能力，包括发布的自动化程度、稳定性（如平滑的灰度发布）和适应性（即面向不同情况的处理能力及出现问题后的自愈能力）。好的交付应该是持续、快速、高质量和低风险的。

运维能力：包括系统的可观测性、应用的运维水平和基础设施的运维水平，是系统运行时弹性和韧性水平的体现。其中，可观测性是运维能力中最重要的一环，主要体现在能否站在系统的角度看到全局的运行情况以及其中的问题，通常体现在监控水平和链路分析及问题定位能力上；应用运维是指对应用进行的运维操作，包括配置项的修改、调整应用运行时参数、对应用进行调整如扩缩容等；基础设施运维是指对系统的基础设施部分的运维操作，包括虚拟机、容器平台、基础服务（如域名、配置中心等），这是整个系统的基础部分。最好的运维就是自运维。

协同能力：包括业务和技术间的协同，以及开发与运维的协同能力，它是整体协同和业务响应能力的体现。其中，开发和运维协同是为了交付过程更加顺畅和高效，以提高技术的响应速度，同时保障系统运行的弹性和韧性；技术和业务协同是为了让从业务到技术的价值传递和交付更加精准、高效，反馈更加即时，以提高业务发展和创新的效率和效果。

成熟度模型

基于这 4 大类 10 个能力，我们给出了一个包含 5 个级别的成熟度模型，从 L0 到 L4 成熟度依次递增。

分类	能力	L0	L1	L2	L3	L4
基础能力	服务化水平	低（单体或刚开始进行服务化）	中（基于特定框架下的业务开发）	中（基于特定框架下的业务开发）	高（仅关注业务开发，语言相关）	高（仅关注业务开发，语言无关）
	基础设施关注度	高（非云原生基础设施）	高（非云原生基础设施）	中（云原生基础设施）	较低（主要 serverless）	低（完全 serverless）
交付能力	工具化水平	低（无 DevOps 工具链）	中（部分、孤岛式的工具）	较高（持续交付工具链）	较高（持续交付工具链）	高（端到端 DevOps 工具链）
	测试自动化水平	低（无自动化，反馈时间长）	低（无自动化，反馈时间长）	中（部分自动化）	较高（主要自动化）	高（完全自动化）
	部署发布能力	低（手工发布）	较低（自动化工具发布）	中（自动化声明式发布）	较高（自动化声明式发布，有干预的灰度能力）	高（自动化声明式发布，自动灰度能力）
运维能力	可观测性	低（只能观测散点式的基础资源）	较低（散点式的基础资源和服务接口可观测）	中（服务整体能力可观测）	高（全景业务能力和服务能力可观测可追溯）	高（全景业务能力和服务能力可观测可追溯）
	应用运维能力	低（人工运维）	较低（自动化工具运维）	中（声明式运维）	中（声明式运维）	高（服务自运维）
	基础设施运维能力	低（单点、人工运维）	较低（单点、自动化工具运维）	中（集群、自动化工具运维）	中（集群、自动化工具运维）	高（集群、自运维）

分类	能力	L0	L1	L2	L3	L4
协同能力	开发、运维协同能力	低（开发、运维分离批量部署、独立运维）	较低（开发、运维协作部署，加大部署频次）	中（开发自主部署开发协助应用运维）	较高（开发团队持续交付开发、运维团队融合）	高（开发团队持续交付、自运维）
	业务、技术协同能力	低（业务与技术相互独立、抛接需求、很少同步）	较低（业务与技术定期同步，批量开发、部署、发布）	中（以业务需求为单位持续开发、部署、发布）	中（以业务需求为单位持续开发、部署、发布）	高（业务需求的快速响应和交付，业务需求的即时反馈、调整闭环）

L0：手工批量交付、手工运维，这是零能力的 DevOps 阶段，其服务能力完全取决于开发者个人，业务交付质量普遍不高，随着业务的发展和团队规模的变大会遇到各类问题，通常会首先寻求工具的帮助。

L1：手工为主、工具辅助的批量交付和运维，这个阶段开始引入自动化工具来辅助进行运维、发布等工作，通常已经有了服务化的基础，基础设施已经部分上云，并通过引入开源工具或自建搭建了一些完成特定诉求的工具，但这些工具往往还是孤岛，没有联系起来，业务、开发、运维间采用定期同步的方式，需求的交付还是批量式的。

L2：基于业务需求的部分自动化的交付运维，这个阶段能基于业务需求进行持续发布，已经采用了声明式运维，通常已经使用云原生的基础设施，并且使用云上的资源管理服务状态，大部分工具链已经能串联起来，实现一定程度的持续交付，服务开始具有中间件级别的抽象和治理能力，但一般还无法做到自运维，回滚等操作还需要依赖人来判断和处理。

L3：基于业务需求的端到端自动化交付和有限制的自运维，这个阶段的业务需求交付频率和交付质量有了明显提高，服务化水平已经相当高，针对特定的技术栈可以做到大部分情况下关注业务开发，主要服务以 serverless 方式发布运行。发布过程可以做到自动化和声明式，只在灰度处理上需要少量的干预，服务已经可以做到大部分情况下的自运维和自治理。

L4：业务需求的端到端持续交付和调整闭环、完全自运维，这个阶段开发者只需关注业务开发，且业务需求能够做到快速的交付和调整，服务化水平与技术栈解耦，做到完全 serverless 化。整个交付过程完全自动化，服务能够完全自治理。这个阶段是我们追求的目标。

阿 里 云 云 效

yunxiao.aliyun.com