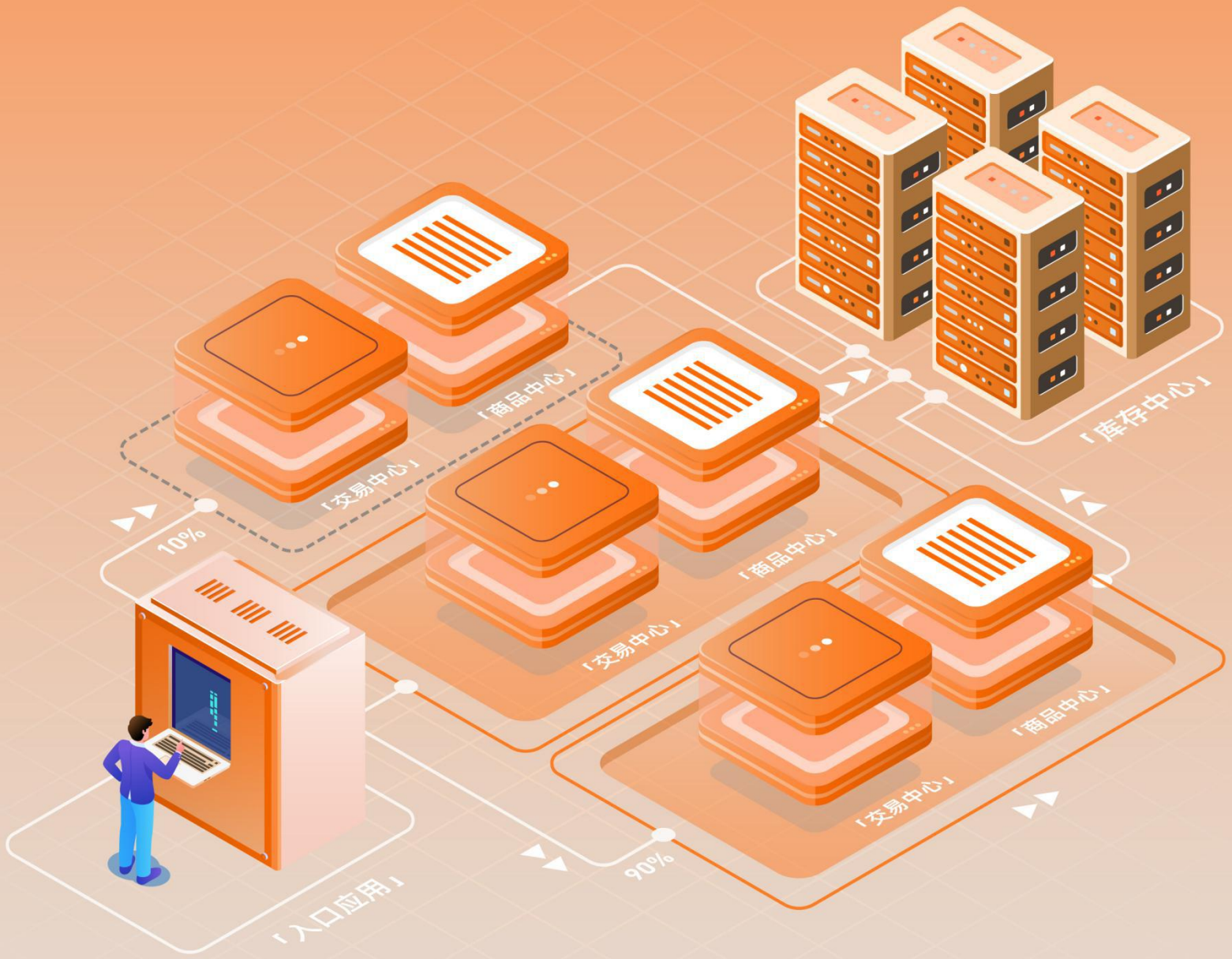
微服务治理技术白皮书

White Paper on Microservices Governance Technology

「高效构建完整的微服务治理体系，
提升开发效率和线上稳定性」



阿里云



电子书中涉及的部分服务治理技术和最佳实践，已经通过 OpenSergo 对外进行开源，该项目由阿里云、bilibili、字节跳动、Apache Dubbo/dubbogo 社区、Nacos 社区、Spring Cloud Alibaba 社区共同维护，相关微服务治理技术已被来自各行各业的数千家企业所采用，特别鸣谢上海三菱电梯、来电科技、Saleforce 中国为此书撰写推荐序，欢迎更多企业加入，共建服务治理开放社区。





阿里云开发者“藏经阁”
海量电子手册免费下载

推荐序

Alibaba Cloud

- 阿里云研究员，云原生应用平台负责人 - 丁宇（叔同）

在阿里巴巴微服务架构 10 余年的演进历程中，服务部署量不断扩大，已经迈入百万节点规模，如此庞大的微服务体系必须要通过服务治理进行精细化管控，提升线上业务稳定性。阿里集团的服务治理框架从无到有，经历了服务框架提供治理 SDK、轻量级隔离容器 Pandora 、无侵入式的 Java Agent 以及针对异构微服务的 Service Mesh 等架构迭代历程，这个过程中沉淀了丰富的服务治理能力，涵盖了开发、测试、线上运维、高可用等多个方面。这本技术白皮书系统化的阐述了服务治理的演进路线以及落地最佳实践，相信对企业开发者和架构师在采纳微服务架构时能有所帮助。

- 阿里云资深技术专家，云原生中间件负责人 - 胡伟琪（白慕）

阿里巴巴作为国内微服务的先行者，有着丰富的实践经验和技术积累，近几年阿里巴巴中间件团队推行三位一体的技术战略，把内部业务支持、云产品、开源进行了技术和架构的统一，并开源了一系列优秀的项目，包括 Apache Dubbo，Apache RocketMQ、Nacos、Spring Cloud Alibaba、Seata 等，为大量企业进行微服务化架构升级提供了助力。本书是阿里巴巴中间件团队在微服务领域的又一力作，详尽介绍了阿里巴巴针对大规模微服务场景下，进行高效治理的方案和思想，推荐目前有计划或已经完成了微服务改造、对微服务领域技术有兴趣的技术人阅读。

- 阿里云资深技术专家，云原生 Serverless 及可观测产品线负责人 - 司徒放（姬风）

在微服务架构已经大行其道的今天，为什么我们要谈微服务治理？就像马斯洛需求模型所述，人类的需求是生理、安全、社交、尊重和自我成就的逐步实现。类似的，企业实施微服务架构的过程也遵循着同样的道理。微服务架构，第一阶段要解决服务间的发现问题和相互通信问题，这是微服务框架所覆盖的基本功能。第二阶段要解决微服务应用的交付和规模化运维问题，这些是容器和 K8s 所擅长的领域。第三阶段随着微服务架构复杂化，分布式场景下排查和诊断效率急剧下降开始成为开发者主要痛点，因此又催生了分布式链路跟踪和可观测性技术。正所谓“乱极必治”，微服务架构是一把双刃剑，大规模之下掩盖的问题很多，从开发联调到发布上线，从流量防护到故障恢复再到容灾，如果不引入恰当的治理手段，很可能会积重难返、万劫不复。因此微服务治理是微服务演进的第四个必然阶段，微服务治理得到重视是恰逢其时。

阿里巴巴从 2008 年开始践行微服务，阿里中间件团队也一路伴随着走过了上述几个阶段，是微服务架构发展的亲历者。这本白皮书来自中间件团队里面对微服务最有经验的那几个人之手，也是阿里微服务开源 Dubbo、Spring Cloud Alibaba、Nacos 等项目的主要创作者。他们结合了阿里自身的微服务实践，以及中间件上云之后面服务外部企业客户的第一手案例，我认为本书无论从内容的丰富度，还是经验的普适性来看，都是国内最有参考价值的一份微服务架构材料。将本书诚挚推荐给每一个希望在企业 内应用微服务架构的架构师、以及对微服务有兴趣的爱好者阅读。

- 阿里云高级技术专家，云原生解决方案负责人 - 邱戈川（了哥）

软件技术的发展，从单体的应用，逐渐演进到分布式应用， 特别是微服务理念的兴起，让大规模、高并发、低延迟的分布式应用成为可能。但是微服务架构不是银弹， 其维护的复杂度，以及管理、治理的难度超过以往的任何系统架构。而阿里巴巴/阿里云中间件团队是这个领域的先行者，通过多年的阿里巴巴内部演进与支持，以及与大量的阿里云外部客户的共同努力，沉淀出在业界领先的微服务领域特别是服务治理上的一套方法论以及最佳实践。这本白皮书，正是微服务治理方法论以及最佳实践的归纳与总结，可以帮助大家在微服务架构设计与管理上提供很好的借鉴与参考。特别是其独特的类似 Mesh 的理念在容器下结合 Java Agent 模式，很好的解决了 Java 领域提供无侵入式增强治理能力如限流降级、全链路灰度、可观测性增强等问题，同时又巧妙的规避了升级、版本不一致等等维护的代价。如果你正好采用微服务架构来完善自己的应用体系，这个白皮书绝对值能帮助你完善你的架构设计，为你的业务稳定运行提供技术架构保障，强烈推荐！

- 上海三菱电梯，系统开发总监 - 谢璟

云原生时代，软件架构，软件框架都经历飞速的发展。从最早期 Eureka，到目前国内以 Nacos 为首的微服务生态，阿里巴巴为此作出了卓越贡献。在和阿里云和阿里巴巴合作过程中，深深感受到阿里云的技术氛围和底蕴，通过无侵入的 Java Agent ，全面代码零侵入，提供了无缝切换阿里微服务治理技术方案。阿里云的服务治理的模型，依托阿里云坚实的底座，结合阿里云云上资源，使得我们能够快速融入阿里云服务治理，形成统一，标准，最佳的微服务治理方案。微服务治理技术，思想，解决方案，最佳实践都在本书有详细的阐述，是值得每个技术人员认真品读的好书。

- 来电科技 CTO - 罗昌明

来电科技拥有百万级的物联网终端，中后台微服务化后拥有数百个微服务组件，管理这些微服务是个极其复杂的工程。阿里云 MSE 微服务治理技术帮助来电无侵入地实现了服务预热、无损上下线、全链路灰度等微服务治理能力，大大提升了服务的稳定性。如果你正准备深入微服务治理相关知识或者遇到相同问题，那么这本书正好可以提供深入学习的机会。

- Salesforce 中国，技术总监 - 叶文帅

随着业务的发展和团队扩大，微服务架构成为了许多大规模开发团队的不二选择。在微服务架构发展的过程中，我们经历了从传统分布式微服务框架治理到云原生微服务治理的演变。伴随着服务数量的增多以及对服务稳定性要求的提高，社区上和公有云上都诞生了许多优秀的微服务治理工具。阿里云中间件团队针对开发者们在微服务开发中遇到的痛点，结合阿里云生态，在本书讲述了云原生下微服务治理架构的设计以及微服务治理的最佳实践。相信这本白皮书能给企业中微服务架构师，以及对微服务架构有兴趣的开发者带来帮助。

CONTENT

Alibaba Cloud

第一章：综述 **1**

1.1 业务发展离不开微服务治理保驾护航 1

1.2 微服务治理在云原生场景下的挑战 7

1.3 微服务治理的发展趋势 11

1.4 微服务治理的区分 18

第二章：微服务治理技术原理介绍 **21**

2.1 微服务治理技术概述 21

2.2 通过 SDK 方式进行微服务治理 23

2.3 通过 Java Agent 方式进行微服务治理 25

2.4 通过 Service Mesh 来进行微服务治理 28

第三章：微服务治理在云原生场景下的解决方案 **33**

3.1 线上发布稳定性解决方案 33

3.2 微服务全链路灰度解决方案 43

3.3 微服务可观测增强解决方案 56

3.4 微服务应用配置解决方案 69

3.5 微服务限流降级解决方案 76

3.6 微服务开发测试提效解决方案 85

3.7 微服务敏捷开发解决方案 90

3.8 微服务无缝迁移上云解决方案 95

3.9 线上故障紧急诊断、排查与恢复 103

3.10 微服务注册发现高可用解决方案 113

3.11 微服务应用安全解决方案 118

3.12 异构微服务互通解决方案 122

3.13 微服务 Serverless PaaS 解决方案 125

第四章：基于 MSE 的微服务治理最佳实践 128

4.1 线上发布稳定性解决方案最佳实践 129

4.2 全链路灰度最佳实践 146

4.2.1 Ingress-nginx 全链路灰度 165

4.2.2 Zuul 和 Spring Cloud Gateway 全链路灰度 183

4.2.3 MSE 云原生网关全链路灰度 199

4.2.4 全链路灰度之 RocketMQ 灰度 221

4.2.5 使用Jenkins CI/CD 实现金丝雀发布 245

4.3 微服务应用配置最佳实践 262

4.4 微服务限流降级最佳实践 272

4.5 微服务开发测试提效最佳实践 277

4.6 微服务敏捷开发最佳实践 289

4.7 微服务无缝迁移上云实践 306

4.8 如何快速构建服务发现的高可用能力 319

4.9 如何在 Serverless 模式下快速使用服务治理能力 330

第五章：微服务治理客户案例 339

5.1 物流行业：菜鸟 Cpaas 平台微服务治理实践 339

5.2 互联网行业：来电科技微服务治理实践 346

5.3 机器智能行业：云小蜜 Dubbo3 微服务治理实践 357

5.4 游戏行业：广州小迈微服务治理实践 363

第六章：总结与展望 368

总结与展望 368

作者

Alibaba Cloud

白壮丽 - 竹达

曹玲微 - 陌微

曹茵茵 - 蔓铃

陈飞 - 麒汀

陈涛 - 毕衫

陈昕 - 捌哥

范扬 - 扬少

鲁严波 - 卜比

泮圣伟 - 十眠

饶子昊 - 铖朴

苏宇 - 流士

汤长征

唐慧芬 - 黛忻

王桐 - 瑾涵

肖京 - 亦盏

叶辰超 - 楚寰

张海彬 - 古琦

张乎兴 - 望陶

张伟 - 柑橘

张哲 - 溪岳

赵俊阳 - 墨昀

赵奕豪 - 宿何

方剑 - 洛夜

徐靖峰 - 岛风

作者按照姓名拼音升序排序

第一章：综述

1.1 业务发展离不开微服务治理保驾护航

微服务开发不简单

随着微服务技术的发展，微服务(MicroServices) 的概念早已深入人心，也越来越多的公司开始使用微服务架构来开发业务应用。

如果采用得当，微服务架构可以带来非常大的优势。微服务架构的最大好处是它可以提升开发效率和系统整体的稳定性：

- 开发和部署相对简单：单个微服务的功能可以更快地更改，因为可以独立部署，影响范围更小，启动和调试单个微服务的时间成本相比于单体应用也大大减少。
- 横向扩展简单：根据业务的高峰低谷周期快速的横向扩展非常简单，因为单个微服务通常很小，可以随着系统整体负载的变化更快地启动和停止。
- 架构升级灵活：单个微服务的内部架构可以迅速升级，因为微服务之间松散耦合的，只面向定义好的通讯接口进行编程。这使开发团队能够基于自身的技术背景和偏好灵活选择，而不会直接影响其他应用程序、服务或团队。
- 更好的容错性：微服务之间可以实现更好的故障隔离，单个服务内的内存泄露等故障不容易影响其他服务，相比单体应用一个组件故障会拖垮整个系统。

但是微服务在实施过程中，也很容易遇到一些难点。如果微服务治理得不恰当，反而有可能适得其反，不仅不能享受到微服务架构带来的好处，反而会因为微服务带来的系统复杂性，造成开发、运维部署的复杂度增加，进而影响开发迭代的速度，甚至影响系统的整体稳定性。

我们总结了一些微服务开发实施过程中常见的问题：

- 服务之间使用远程调用进行通讯，这比进程内的直接调用复杂很多。由于通讯链路的复杂性，可能会出现很多不确定的问题，会出现远程调用不可用或者延迟较高的情况，开发人员需要能够处理这些偶发问题，避免影响业务。

- 随着业务的发展，微服务之间的拓扑图开始变得复杂，排查问题变得困难，搭建完整的开发测试环境成本也越来越大。

- 当功能涉及到多个微服务模块时，迭代时需要谨慎地协调多个开发团队的迭代排期，才能保证迭代能够按时交付，达到敏捷开发的效果。

一个微服务成功落地的典型案例

观察了阿里云众多客户之后，我们总结抽象了一个微服务成功落地的典型案例，叙述了某公司是如何借助于微服务架构的红利，实现了业务快速增长的。案例详细说明了在业务发展的不同阶段，该公司在微服务实施过程遇到的问题，也描述了如何通过微服务治理来解决这些问题，从而享受到了微服务带来的开发效率和业务稳定性提升的红利，进而促进业务快速发展。

业务孵化期

初创公司在刚起步时，虽然业务量比较小、业务模式比较简单，但是为了在后续的快速发展中能够快速迭代，同时公司的人才储备也满足微服务开发的条件，于是在初创期就选择了使用微服务架构进行业务开发。

在技术选型方面，如果公司创始员工是技术出身，那么会比较倾向于使用自己擅长的微服务框架，比如创始人是 Dubbo 的 contributor ，或者是 Spring Cloud 社区的大咖或者 Spring Cloud Alibaba 的 contributor，又或者是 Service Mesh 社区的大咖，那么一般都会选用自己所擅长的微服务框架类型。还有一种选择是选用市面上最流行的微服务框架，比如 Java 体系，会选择 SpringCloud 和 Dubbo。这样的话，在目前的招聘市场上也容易招聘到熟悉相关领域的人，从初学者到专家级的候选人都能很容易招聘到。

选定了技术选型后，基于开源的框架，很容易就能开发好最初的业务应用系统，跑通业务流程。这一阶段组件也很简单。简单的两三个应用，用户系统、业务系统、支持系统，再加上注册中

心、数据库、缓存，就可以开发完全部的应用。

对于一个生产级别的系统来说，在将整套系统部署上线之前，还需要建设监控报警系统。监控报警系统能够帮助我们实时监控机器和应用的状态。我们可以配置预设的报警规则，在出现机器资源不足，业务出现异常报错时这些情况时主动报警，提醒我们尽快处理系统中的风险和异常。保留问题的现场，并帮助我们快速地定位和排查问题也同样重要，阿里云应用实时监控服务 ARMS 和日志服务 SLS 能够在这些场景提供很大的帮助。

采用 ARMS + SLS 完成监控报警系统建设之后，将业务系统部署上线，完成第一次成功上线，业务开始正常运行起来了。

但是业务的开发和运营从来都不是一帆风顺的，在这个过程中，肯定也会遇到很多问题，我们先总结一下这个阶段常见的问题及应对方案：

1.因为代码本身逻辑出错导致业务异常：这时可以通过 SLS 查询日志，结合 ARMS 分析错误堆栈找到根因，修改完代码后，重新发布来修复问题。

2.遇到性能瓶颈：则直接扩容操作，比如水平扩容应用，或者升级数据库。

3.发布新版本影响到用户：因为用户数和请求数都不算多，很容易就可以找到业务低峰期，在业务低峰期进行发布。

4.如何确保新版本的正确性，因为业务场景不复杂，内部测试就能覆盖所有的场景，测试通过就可以直接上线。

那如何识别自身的业务是否处于这个阶段呢？有一系列典型的特征：应用不超过 4 个，应用节点总数不超过 10 个，最高峰时候的 QPS 不超过 10。

业务快速发展期

在活过初创期之后，公司的业务很受用户和市场的欢迎，注册的用户越来越多，用户使用的时长和功能点也越来越多，日活数越来越大，甚至市场中还出现了其他竞争者开始抄袭公司的业务，一些巨头还亲自下场参与竞争。公司业务发展得非常顺利，这也意味着系统进入了快速发展时期。

市场发展很迅速，注册用户数、日活这些数据也是节节攀升，所有统计报表的数字都是一片向好。但是带来的挑战也越来越大，这个时期也是最危险的时期：在业务快速发展中，既要保证好已有业务的稳定性，又要快速地迭代新功能，还要克服团队招聘节奏跟不上业务发展的问题。

这个阶段典型的特征是应用个数在 5 到 50 个，QPS 在 10 到 1000。这个时期经常会遇到的问题概括起来就是两个：稳定性问题和开发效率问题。

稳定性的问题：用户数多起来之后，系统的稳定性就显得比较重要，无论是用户在某段时间遇到异常报错增多，还是某一个功能点持续性地报错，再大到系统有一段时间完全不可用，这些都会影响产品在用户中的口碑，最后这种完全不可用的场景，甚至还可能成为微博等社交网络上的舆论热点。

开发效率的问题：随着用户的增多，相应的需求也越来越多，业务场景也越来越复杂，在这个时候测试可不是内部测试就能覆盖所有的场景，需要加大在测试上的投入。虽然功能需求越来越多，但是迭代的速度却要求越来越快，因为市场中已经出现了竞争者，如果他们抄得快，新功能也上得快，业务有可能会竞争不过，特别是当巨头亲自下场的时候，更需要跑得更快，开发节奏要快，测试节奏要快，发版节奏也要快。

那么如何去解决这两个场景的痛点呢，这时候可以要借助微服务治理的能力来解决。

1.开发测试提效

a.【开发环境隔离】传统的多套开发环境，需要使用多套的物理环境，才能实现多套环境各自独立互不干扰，但是多套物理环境的隔离的机器成本是很高的，基本上不大能接受。但通过全链路灰度这种逻辑隔离的方式实现开发环境隔离，可以在不增加成本的情况下增加多套开发测试环境，助你实现敏捷开发。

b.【自动化回归测试】自动化回归测试功能，可以将多个测试用例串联成测试用例集，将上一条测试的返回值作为下一跳测试入参，串联成具体的业务场景并沉淀到自动化回归测试中，在每一次的发版之前都跑一次自动化回归来验证功能的正确性，这样就可以大大节省测试的人力成本。更进一步，还可以通过流量录制回放功能，将线上的真实流量录制下来，并沉淀成自动化回归用例集，在测试环境进行流量回放，更进一步地提升测试 case 的覆盖率。

c.【服务契约】功能越来越多，迭代越来越快，API 越来越复杂，团队之间沟通的效率越来越低，API 文档严重过期。如果能自动生成服务契约，可以有效地避免文档腐化造成的开发效率低下的问题。

使用上面三点之后，可以在低成本的条件下支持多套开发测试环境，实现自动化回归测试，实现开发节奏和测试节奏的大大提效。

2.安全发布

a.【无损下线】无损下线问题的根本原因是开源的微服务体系没有确保应用提供者节点在停止服务前确保已经通知到所有消费者不再调用自己，也无法确保在处理完所有请求之后再停止应用。所以新发版的应用，即使业务代码没有任何问题，也可能在发布过程影响用户的体验。

b.【无损上线】无损上线问题出现的原因是因为在某些场景下服务提供者，需要经过一段时间才能正常地接收大流量的请求并成功返回。同时在 K8s 场景下，还需要和 K8s 中的 readiness 、滚动发布等生命周期紧密结合，才能确保应用发布过程中能不出现业务报错。

c.【全链路灰度】新功能上线之后，可以通过灰度规则控制哪些用户可以使用。这样可以先选择让内部用户使用，测试新功能的正确性。当内部用户验证通过后，再渐渐地扩大灰度范围，确保每个功能都经过充分验证后再全量开放给客户，屏蔽掉发布新功能的风险。而且当出现问题时，可以通过修改灰度规则来实现快速回滚，做到新版本发版时几乎无风险。

使用以上几点之后，可以确保新版本的发布不出问题，而且可以做到白天在大流量场景下也轻松发布，在实现白天轻松发布之后，一天就可以发布多次，提升发布时候的稳定性和发版的效率。

3.屏蔽偶发异常导致的风险

a.【离群实例摘除】对于这些偶发的异常问题，离群摘除功能可以智能判断应用中的服务提供者某个出现了问题，智能地在一段时间内屏蔽掉这个服务提供者，保证业务的正常，等这个服务提供者恢复过来之后再进行调用。可以在应用节点出现偶发异常时，智能屏蔽掉此节点，以免影响业务，等此节点恢复后再继续提供服务，从而屏蔽偶发异常导致的风险。

据统计数据显示，有将近 90%的线上故障是由于发版过程中出现的，剩下的 10% 左右的线上

问题，可能是由于一些偶然的原因导致的。比如偶然的网络故障、机器 I/O 出现问题、或者是某台机器负载过高等。在解决了发布时候的稳定性问题和偶发异常导致的风险后，基本能够确保线上业务不会出现灾难性的问题。

在业务快速发展的生死存亡期，您需要借助于这些微服务治理能力，才能确保业务能够又快又稳地增长，度过这段生死存亡期，成为这个领域的重要玩家。

业务成熟期

当业务进入成熟期之后，业务的开发进入了新的阶段，这时候，虽然快速发展过程中的问题仍旧存在，但是会因为业务量上来之后，遇到新的问题。

低成本创新：虽然发展不像原来那么迅速，但是业务创新探索的诉求仍旧存在，由于业务规模的扩大，创新的成本也在增加。这个时候不仅是需要快速开发迭代，更大的需求是用尽可能小的成本进行创新探索测试，有时候还需要使用上 AB 测试的手段进行实验比较。

容灾多活：由于业务规模已经很大了，治理中的稳定性的诉求更加强烈，而且随着业务范围的扩大，应用也开始在多个地域、多个云产品中进行部署。同城容灾、异地多活这类需求也开始出现。

问题定位：出现任何问题都必须彻查。虽然出现问题时，业务恢复仍旧排查第一位，但是业务恢复之后的问题根因定位也是不能少的，因为如果不彻查，难免后续出现同样的问题。

风险预案：紧急预案、风险预防也变得非常重要，需要提前做好业务的保护和降级的埋点演练，在遇到绝大多数可预见问题可以紧急修复，出现不可控问题时，可以通过预案手段执行预案，确保整体业务的可控性。

从这个典型的案例中，我们可以看到，微服务的成功落地和业务的快速发展，离不开微服务治理的支持。在业务发展的不同阶段，会遇到不同的微服务问题，需要借助于治理的能力，为业务的又快又稳发展保驾护航。

1.2 微服务治理在云原生场景下的挑战

企业上云的四个阶段

随着云原生时代的到来，越来越多的应用生在云上，长在云上，且随着越来越多的企业开始上云，云原生也是企业落地微服务的最佳伴侣。

我们分析了阿里云典型客户的实践经历，业务上云通常划分为 4 个阶段：云上部署、云原生部署、微服务化、服务治理。



云上部署

这一阶段我们解决的问题，如何把传统业务，原来是跑在自建 IDC 机房的业务，能够原封不动的迁移到云上。通常云厂商提供了丰富的计算，存储，网络等资源可供选择，以虚拟化技术，神龙裸金属服务为代表的硬件可以满足企业客户上云搬迁的丰富需求，这一阶段关注的焦点是资源，对于业务并无任何的改造，只需要从本地原样搬迁到云上即可。

云原生部署

云原生是释放云计算价值的最短路径，以容器技术为代表，云原生提供了强大的调度，弹性等能力，极大的降低了上云的成本。这一阶段我们关注的目标主要是业务进行云原生化改造，随着 Kubernetes 作为容器编排市场的事实标准，我们需要把业务从原来的的虚拟机部署方式改造成容器化方式，部署并运行在 K8s 之上，最大限度享受到云原生带来的技术红利。这一阶段核心关注目标以容器为核心。

微服务化

当我们的业务规模逐步扩大，传统单体应用很难进一步支撑业务的发展，业务的迭代速度已经无法满足业务的增长，此时我们就需要进行微服务化的改造，降低业务的耦合度，提升开发迭代的效率，让开发更加敏捷。这一阶段我们聚焦以应用为核心。

服务治理

当微服务的规模也越来越大的时候，如果对微服务不加以规范和整治，很容易出现问题。例如，每个微服务都有独立的团队来维护，他们之间如果依赖没有整理清楚，可能会出现架构上循环依赖等问题。从我们的数据观察来看，当微服务的节点数超过数十个的情况下，我们通常就需要引入服务治理，通常需要关注的是开发，测试，线上运维，安全等多方面考虑，这一阶段我们聚焦以业务为核心，核心目标是进一步提高开发效率，提高线上业务的稳定性。

微服务治理在云原生下的挑战

随着企业微服务化进程的逐渐深入，微服务的云原生化逐步进入深水区，在这个微服务深化的过程中，我们逐步会面临一系列的挑战，总的而言，我们将这些挑战分为三个大的层面，他们分别是效率，稳定和成本。

我们进行微服务化，本身的使命是让业务的迭代更加高效，但当我们的微服务数量逐步增多，链路越来越长，如果不进行进一步的治理，那么引发的效率问题可能会大于微服务架构本身带来的架构红利。在上一章节中我们提到过在微服务实施的不同阶段，遇到的问题也不尽相同。目前阿里巴巴内部的微服务节点数量是在百万级别，在这个过程我们也积累的非常多的治理经验。



在效率上面临的挑战

在效率方面需要追求的目标是，在开发，线上运维，SDK 升级等方面更加高效。

- 在开发阶段，我们需要考虑的是，业务应用上云之后，如何让本地开发的应用，很好的部署云上的业务进行联调？通常我们的微服务不可能在本地完整的部署一整套系统，所以本地开发的应用只是整个微服务链路的一小部分，这包括我们的流量需要能够轻松的从云上，引导到本地，便于我们做开发调试，或者我们在本地能够很方便的调用云上部署的微服务进行联调。这在微服务上云之后，变的比原来在自身机房进行开发联调更加困难。

- 在线上运维方面，我们通常需要频繁的对微服务进行变更，这些变更通常就会引发一系列的问题，例如在白天高峰期做发布，通常都会导致业务流量出现损失，我们的研发人员不得不选择在晚上业务低峰期做变更，这大大降低了研发人员的幸福指数，因为他们不得不面临熬夜加班的困境。如果能在白天大流量高峰期也能进行流量无损的变更，那么这对于研发人员来说将是大大提升研发效率的事情。

- 微服务框架通常会引入服务治理的逻辑，而这些逻辑通常会以 SDK 的方式被业务代码所依赖，而这些逻辑的变更和升级，都需要每一个微服务业务通过修改代码的方式来实现，这样的变更造成了非常大的升级成本。以阿里巴巴为例，阿里内部一个中间件 SDK 的升级，如果要在整个集团铺开，通常需要消耗的时间以年为单位进行统计，这里面也会消耗每个微服务应用的研发，测试等庞大的资源，效率非常低下，如果能够以无侵入的方式实现中间件 SDK 的升级，那么将会是一件非常高效的事情。

- 进入云原生体系之后，以 K8s 为主的云原生体系强调集群之间的灵活调度型，以 POD 为单位任意的调度资源，在被调度后 POD 的 IP 也将相应的发生变化，传统的服务治理体系，

通常以 IP 为维度进行治理策略的配置，例如黑白名单策略等，但是当进入云原生场景后，这些传统的治理策略都会面临失效的问题，因为 POD 一旦被重新调度，原来的治理策略都将不再使用，如何能让服务治理体系更加适应云原生体系，也是我们要面临的一大挑战。

在稳定上面临的挑战

稳定大于一切，在微服务上云之后，业务高可用是我们必须要解决的问题，因此通常会在同一个地域的多个可用区内进行部署，在多可用区部署的情况下，跨可用区的延时就是不可忽视的问题，我们需要思考的是业务流量需要能够尽量在同一个可用区内进行流转，同时也需要考虑的是如果一个可用区出现问题，业务流量能够尽可能快的流转到正常的可用区，这就对我们的微服务框架的路由能力提出了挑战。

当然，我们的业务不仅需要在同一个地域里保证高可用，也需要考虑一个地域出问题的时候，保证业务的高可用，这时我们就需要考虑业务实现同城双活，甚至是异地多活，这对我们来说也是一大挑战。

第三，微服务之间的调用也需要更加的安全可信，近期层出不穷的安全漏洞，一定程度上也反应出当前上云阶段在安全方便暴露出的问题还是非常多，每次安全漏洞出现之后，中间件 SDK 的升级也是困扰业务多年的问题；同时，一些敏感的数据，即使在数据库层做了非常多的权限管控，由于微服务被授予了数据访问的较高权限，如果微服务的调用被恶意攻击，也可能会造成敏感数据的泄露。微服务之间的调用需要更加可靠可信。

在成本上面临的挑战

首先，在成本方面，业务上云遇到的最大问题就是如何最低成本的把业务迁移上云，对于一个在线业务，如果要进行停机迁移，那么迁移的成本会显得非常高，对于客户的体验也会收到影响，要在不中断业务的情况下，实现平滑迁移上云，还是有非常大的挑战的。

其次，当我们在业务面临极速增长的流量时，迫切的需要快速的弹性，补充更多的资源以承载业务的高峰，当我们进入业务低峰的时候，又希望能够缩小容量，节省资源，因此云产品提供的快速灵活的弹性机制，是微服务上云之后一项急需的能力。

1.3 微服务治理的发展趋势

背景

云原生时代下，我们看到云原生微服务作为核心的技术一直保持着 20%左右的高速增长，微服务技术也渗透到各行各业。在云原生微服务技术逐渐趋于成熟的今天，我们以阿里集团微服务发展与阿里云微服务产品发展的历史为镜子再一次展望微服务发展的趋势。

阿里集团微服务发展历史

服务框架就像铁路的铁轨一样，是互通的基础，只有解决了服务框架的互通，才有可能完成更高层的业务互通，所以用相同的标准统一，合二为一并共建新一代的服务框架是必然趋势。

Dubbo3 是 Dubbo2 与 HSF 融合而来，是阿里巴巴集团面向内部业务、商业化、开源的唯一标准服务框架。

Dubbo 和 HSF 在阿里集团的实践

Dubbo 则在 2011 年开源后，迅速成为业界广受欢迎的微服务框架产品，在国内外均有着广泛应用。Dubbo 项目诞生于 2008 年，起初只在一个阿里内部的系统使用；2011 年，阿里 B2B 决定将整个项目开源，仅用了一年时间就收获了来自不同行业的大批用户；2014 年，由于内部团队调整，Dubbo 暂停更新；2017 年 9 月，Dubbo 重启开源，在 2019 年 5 月由 Apache 孵化毕业，成为第二个由阿里巴巴捐献至 Apache 毕业的项目。



HSF 在阿里巴巴使用更多，承接了内部从单体应用到微服务的架构演进，支撑了阿里历年双十一的平稳运行；自 2008 年 5 月发布第一个版本 1.1 后，经历数年迭代，HSF 从一个基础的 RPC 框架逐渐演变成为日支撑十万亿级别调用的易于扩展的微服务框架。内部场景中，用户既可以选择少量配置轻松接入微服务体系，获取高性能的稳定服务调用。也可以按照自身业务需求对 HSF 进行扩展，获取整条链路的能力增强。

HSF 一统天下

对于集团内的需求而言，稳定和性能是核心，因此，当时选型了在电商这种高并发场景久经考验的 HSF 做为新一代服务框架核心。随后，HSF 推出了 2.0 的版本，并针对 HSF 之前版本的主要问题进行重构改造，降低了维护成本，进一步提高了稳定性和性能。HSF2.0 解决了通讯协议支持不透明，序列化协议支持不透明等框架扩展性问题。基于 HSF2.0 的 Java 版本，集团内也演进出了 CPP/NodeJs/PHP 等多语言的客户端。由于 HSF 还兼容了 Dubbo 的协议，原有的 Dubbo 用户可以平滑地迁移到新版本上，所以 HSF 推出后很快就在集团全面铺开，部署的 server 数量达到数十万，基本完成了阿里巴巴内部微服务框架的统一，并经历了多年双十一零点流量洪峰的验证。

Pandora 与 HSF 黄金搭档

HSF 的顺利落地离不开其丰富的周边生态，服务注册中心、配置中心、限流降级、流量调度、功能开关、分布式事务、预案平台这些都是 HSF 的好伙伴，除了一起完成基本的微服务调用功能之外，也在多次的双十一中进行保驾护航。

这么多的组件，他们的 SDK 之间的升级和依赖管理是一个很难规避的问题。如果没有进行很好的管理，就可能出现两种组件之间的三方依赖互相冲突的情况。也有可能出现某些组件需要特定的版本组合才能正确使用，这些对于开发者来说，分辨起来是一个不小的成本。如果某一个版本的组件出现高危 bug，需要推动全量升级，这些开发、构建、发布的成本，更是难以预估。

为了解决这些问题，Pandora 孕育而生。Pandora 是一个轻量级的隔离容器，它用来隔离 Webapp 和中间件的依赖，也用来隔离中间件之间的依赖。Pandora 会在运行时通过类隔离的方式，将各个中间件之间的三方依赖隔离开来，有效地避免了三方依赖互相冲突的情况。同时，Pandora 还会在运行时导出中间件的类，来替换 SDK 中所引入的中间件类，这样就可以实现运行时的中间件版本和开发时的中间件版本分离。应用升级 SDK 只需升级 Pandora 容器即可，只有在大版本升级时才需要修改 BOM 和重新打包。

三位一体战略下服务框架与服务治理的最终选择 Dubbo3

随着业务的发展，阿里集团也因为同时存在 HSF 与 Dubbo 框架而导致的不少问题。原有部门或公司的技术栈如何更快地融入到现有技术体系是一个绕不开的问题。

一个典型的例子就是 2019 年加入阿里巴巴的考拉。考拉之前一直使用 Dubbo 作为微服务框架，基于 Dubbo 构建了大规模的微服务应用，迁移的成本高，风险也大。需要集团和考拉的基础架构部门耗费较长的时间进行迁移前调研、方案设计，确保基本可行后再开始改动。从分批灰度上线，再到最终全量上线。这种换血式的改动不仅需要耗费大量人力，时间跨度也很长，会影响到业务的发展和稳定性。

同时由于历史原因，集团内部始终存在着一定数量的 Dubbo 用户。为了更好的服务这部分用户，HSF 框架对 Dubbo 进行了协议层和 API 层的兼容。但这种兼容仅限于互通，随着 Dubbo 开源社区的多年发展，这种基础的兼容在容灾、性能和可迭代性方面，都有着较大的劣势，同时很难对齐 Dubbo 的服务治理体系。在稳定性方面也存在风险，更无法享受到集团技术发展和 Dubbo 社区演进的技术红利。

随着云计算的不断发展和云原生理念的广泛传播，下一代微服务也带来了其他的挑战和机遇，微服务的发展有着以下趋势：

1.K8s 成为资源调度的事实标准，Service Mesh 从提出到发展至今已经逐渐被越来越多用户所接受。屏蔽底层基础设施成为软件架构的一个核心演进目标，无论是阿里巴巴还是其他企业用户，所面临的问题都已经从是否上云变为如何平滑稳定地低成本迁移上云。

2.由于上云路径的多样以及由现有架构迁移至云原生架构的过渡态存在，部署应用的设施灵活异变，云上的微服务也呈现出多元化的趋势。跨语言、跨厂商、跨环境的调用必然会催生基于开放标准的统一协议和框架，以满足互通需求。

3.端上对后台服务的访问呈爆炸性的趋势增长，应用的规模和整个微服务体系的规模都随之增长。

这些趋势也给 HSF 和 Dubbo 带来了新的挑战。产生这些问题的根本原因是闭源的 HSF 无法直接用于广大云上用户和外部其他用户，而开源产品对闭源产品的挑战会随着开源和云的不断发展愈演愈烈。越早解决这个问题，阿里巴巴和外部企业用户的云原生迁移成本越低，产生的价值也就越大。

因此，HSF 和 Dubbo 的融合是大势所趋。为了能更好的服务内外用户，也为了两个框架更好发展，Dubbo3 和以 Dubbo3 为内核适配集团内基础架构生态的 HSF3 应运而生。

阿里云微服务产品发展历史

我们站在现在开始回顾的时候，惊奇地发现阿里云微服务产品的发展历程跟阿里集团微服务发展的历史也是非常的相似，或许跟我们最开始创新的源泉来自于阿里集团的实践是分不开的。

HSF + Pandora 时代

阿里云上最早输出微服务治理的产品是 EDAS，定位于分布式应用服务一站式解决方案，最初主推的微服务方案是 HSF + Pandora 的方式，直接将阿里内部已经经过双十一验证的高性能框架在云上输出，同时还输出了 HSF 周边的生态组件，如注册中心、配置管理、链路跟踪、限流降级等，在一经推出之后，非常受正在数字化转型的企业欢迎，服务了非常多的政府服务、事业单位、银行客户以及新零售转型的头部企业，也取得了不错的成绩。

随着业务的深入，我们的客户除了头部政企、数字化转型的团队，也越来越多的互联网客户开始使用我们的产品。微服务框架是基础组件，大部分公司在早期选型或业务发展到一定规模的时候都需要确定使用某一个框架。而一个稳定高效的自研框架通常需要较长时间的迭代来打磨优化。所以大部分公司初期都会倾向于使用开源组件。

对当时阿里云微服务产品而言，这就带来了一个问题：内部使用的是 HSF 框架，而云上的用户大部分都是使用的开源 Dubbo 框架，两种框架在协议、内部模块抽象、编程接口和功能支持上都存在差异。客户在上云的时候，不得不对自己的业务代码进行改造，开发和迁移成本非常大。另外，由于代码不开源，对于许多用户而言，整个服务框架对于他们来说是一个黑盒的组件，排查问题是一个非常头疼的问题，用户会担心稳定性得不到保证，也担心被云厂商技术绑定。

我们发现这并不是一个很好的产品化方向，调研之后发现客户大多数的微服务框架都会选择开源的 Dubbo/Spring Cloud，于是阿里云也选择了拥抱开源的方式，主推 Dubbo 与 Spring Cloud 框架。

拥抱开源，无侵入支持 Dubbo 与 Spring Cloud

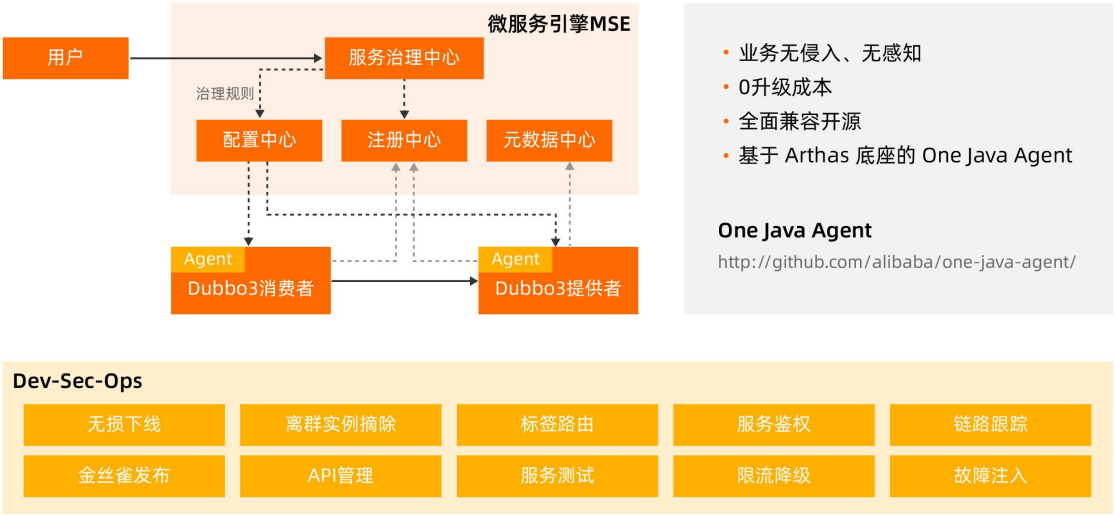
对于微服务框架来说，由于关联到客户的业务代码，要做商业化还是有非常大的挑战的。即使是已经在微服务治理能力上支持了开源的 Spring Cloud 和 Dubbo,但是用户使用起来却不是

那么容易。

首先，迁移成本上来说，希望把降低迁移成本为 0。一个已经运行起来的微服务业务，都会有自建的注册中心，如果要上云还需要把自己的注册中心迁移到 MSE 注册中心上。这个过程需要用户对代码做改造才行，一般来说会采用双注册的方案，通过实现在两个注册中心同时注册和订阅的方案，来实现新老应用可以互相调用，做到平滑迁移上云。

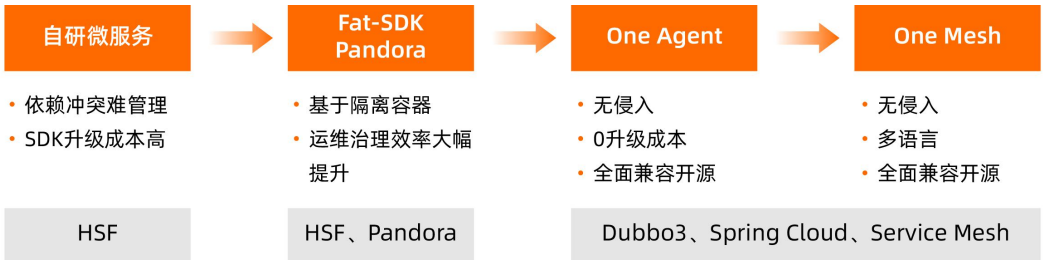
但是我们发现推动客户做代码改造，包括 SDK 的升级是一件非常难的事情，很多客户的 Dubbo 版本还停留在 4-5 年的版本，不仅需要研发、测试、运维都来关注，还需要排期支持，这个动作会耗费大量的人力资源，同时面临许多稳定性的挑战，光是这一步就会阻挡住绝大多数的的客户。

为了解决客户 SDK 升级的问题，我们在想，能不能不要迁移注册中心呢，最好是代码一行也不用改，部署到云上来，就能完整地使用我们的微服务治理能力。在调研了多方技术实现后，我们开始基于 Java Agent 字节码增强的技术来开发微服务治理的能力，帮助用户不改一行代码使用云产品，真正做到了迁移和使用的成本为 0。同时，由于不修改任何代码，客户也可以做到随时可上可下，没有绑定，比较放心的上云产品。



对于商业化中微服务框架的选择，我们选择的态度一直是拥抱开源。并在开源微服务框架的基础上提供差异化的服务治理能力，传统开源微服务框架在 k8s 上的问题在上云的过程中逐步暴露出来。通过一系列和客户的交流，我们总结出了客户的云原生下进行微服务治理的几大痛点，主要包括微服务发布过程中的无损上下线，标签路由，服务鉴权，离群实例摘除，全链路灰度等等，我们通过 Java Agent 技术实现了在用户不改代码的情况下，解决了上述问题，通过客户交流，收集需求，落到产品，给客户 demo 验证这个模式跑通了正向的循环，功能不断丰富

中。除了 Java Agent，对于多语言的场景我们使用Service Mesh、WASM 等技术，同样支持客户无需修改一行代码，就具备于 Java 应用一致的服务治理能力与体验。



云原生下微服务治理发展趋势

随着云原生技术的不断发展，微服务治理也仍旧在快速的发展和变革中。

我们亲身经历了阿里集团的微服务的发展与选择，也参与了阿里云微服务产品的发展。我们观察到微服务技术的主要 3 个趋势：后端服务 BaaS 化；服务治理下沉，业务透明化；部署形态混合云化。



下面将分别分析这几个趋势

后端服务 BaaS 化

BaaS，即 Backend as a service 的简称，微服务架构下，微服务应用通常会依赖多个后端服务，典型的如数据库，缓存，消息队列等等，过去我们搭建一个微服务体系通常使用开源软件自行搭建这些后端的依赖，但是维护这些后端组件需要大量的人力资源和成本，一旦出问题风险非常大。随着业务的云原生上云，我们发现越来越多的云厂商通过云服务的形式，提供这些

后端依赖的托管服务，免去了业务开发人工运维这些软件的烦恼。由此使得企业越来越聚焦在自己的业务应用上，而更少的去关注底层的中间件依赖。过去我们从数据库开始，再到消息队列，缓存，几乎所有的云厂商都提供了托管的服务供选择，然而随着微服务化进一步深入，我们认为微服务的注册中心，配置中心，微服务网关，服务治理中心，分布式事务等也逐步由云厂商提供，通过标准服务的形式，企业上云之后可以灵活的选择，并且没有厂商的锁定。

服务治理下沉，业务透明化

随着 Service Mesh 技术概念的兴起，结合阿里巴巴内部架构演进的实践，我们发现，服务治理技术将逐步的和业务解耦，对业务更加的透明，无论是以 sidecar 为首的 service mesh，还是 Java Agent 为主的新兴治理技术，他们解决的核心问题，就是让业务摆脱对服务治理能力的依赖，让基础设施的迭代可以脱离业务发展独立进行。同时，微服务各个语言之间通常会有不同的框架，这些跨语言的微服务框架将形成统一的服务治理控制面，进一步降低客户选择的成本，一个基于开源微服务框架实现的微服务系统，将能够不改一行代码，能够部署到云上，享受云厂商提供的 BaaS 服务。

部署形态混合云化

第三个趋势是企业上云部署形态不再会选择单一云厂商，无论是从避免厂商锁定，还是从稳定性来讲都会考虑跨多个云厂商提供服务，因此云厂商提供的服务会趋向于标准化，否则企业很有可能选择开源自建来避免厂商的锁定；同时，企业上云也非一蹴而就，通常大量的业务仍部署在自建 IDC 的业务，新兴业务逐步上云。如何针对跨多云，跨自建机房和云上等多种部署形态，对业务进行统一的管理，也是摆在企业面前的难题。未来的云服务，应该是可以提供跨多个云统一纳管，并且一致体验的云服务。

总结

随着云计算的迅速发展，微服务将无处不在。云原生微服务治理的标准也在逐渐成型。相信在不远的将来，我们观察到的三个趋势：后端服务 BaaS 化；服务治理下沉，业务透明化；部署形态混合云化，会逐渐成为云原生下微服务治理的标准，也相信标准的形成会进一步助力云原生微服务治理的蓬勃发展，实现无数云上企业的业务永远在线。

1.4 微服务治理的区分

我们基于应用开发、测试、运维的不同阶段，对服务治理的概念做一个区分，分为开发态、测试态和运行态。其中运行态又分为发布态、安全态和高可用。

开发态Dev	测试态Test	运行态Ops		
		发布态	安全态Sec	高可用
• 服务元信息 • 服务契约管理 • 服务调试 • 服务Mock • 开发环境隔离 • 端云互联	• 服务压测 • 自动化回归 • 流量录制 • 流量回放	• 无损下线 • 无损上线 • 金丝雀发布 • A/B Test • 全链路灰度	• 服务鉴权 • 漏洞防护 • 配置鉴权	• 离群实例摘除 • 限流降级 • 同AZ优先路由 • 就近容灾路由

开发态服务治理

开发态服务治理的目标是为了提升研发效率，让开发更快捷，主要功能包括：

- 服务契约：业务开服能够清晰的了解应用定义了哪些接口、每个接口的参数、以及接口的业务说明；便于开发者迅速了解应用。
- 服务调试：在微服务开发和运行时快速地对某个接口进行调试，而不需要经过手动编写测试代码，也不需要关心网络打通流程。
- 服务 Mock：当某个接口尚未开发完成时，可以通过配置 Mock 此接口的请求行为，返回预设的值，使得开发时不需要依赖于下游接口开发完成。
- 开发环境隔离：通过逻辑隔离的方式，为每一个正在开发的功能特性隔离出一个独立的环境，在低成本的前提下，划分出多个完整的独立环境，使得各功能特性的开发调试不会互相影响，提升开发迭代的效率。
- 端云互联：本地开发的微服务可以快速的访问云上的服务，云上的服务也能调用到本地开发的微服务。

测试态服务治理

测试态的微服务治理的目标是为了提升测试效率，让测试更快更准更全面。

- 服务压测：微服务上线前快速发起压测，迅速了解微服务的容量是否偏离基线，确保新版的性能。
- 自动化回归：通过自动化的方式进行回归测试，自动发起测试并自动比对结果进行验证，无需人工重复测试，保障业务代码逻辑的正确性。
- 流量录制：将线上流量录制下来，自动生成测试用例进行回归测试，通过真实的请求丰富测试覆盖率，保障业务代码逻辑的正确性。
- 流量回放：将录制好的流量重新运行，验证当前的业务运行结果是否和录制好的请求的结果匹配。

运行态服务治理

运行态通常又分为 3 个部分：发布态，安全态，高可用。

发布态

发布态通常解决的是业务发布的时候的流量治理问题，他主要包括：

- 无损下线：确保应用在发布、停止、扩容时，所有请求都不会被影响，确保微服务下线的过程中业务无损。
- 无损上线：应用刚启动时可能会存在一些资源未初始化完成、未预热完毕的情况，无损上线功能可以确保在这个场景下不影响业务。
- 金丝雀发布：满足特定流量特征的请求才会进入微服务的灰度节点，通过小流量验证微服务新版的逻辑是否符合预期。
- 全链路灰度：一个迭代的多个应用同时发布，希望经过灰度的上游流量只能达到下游的灰度节点，确保灰度流量只在灰度环境中流转。

安全态

- 服务鉴权：保护敏感微服务，确保敏感服务只能被已授权的应用发起访问。
- 漏洞防护：开源框架通常会陆陆续续被发现许多漏洞，整体的升级成本很高，需要通过不升级框架的方式实现漏洞的防护。
- 配置鉴权：某些配置比较敏感，不希望任何微服务都有权限访问，控制只有受限的微服务才能访问。

高可用

- 限流：针对超过阈值的流量进行限流控制，保障机器和整体业务的稳定性。
- 降级：在资源有限的情况下，针对某些不重要的请求返回预设的降级结果，把有限的资源让给重要的请求。
- 熔断：客户端访问后端服务不可用的情况下，返回固定异常或预定义的结果，避免引起业务异常，甚至雪崩。
- 离群实例摘除：在单个服务提供者节点持续不可用的情况下，在消费者侧摘除这个异常节点，保障业务的高可用。
- 同可用区优先路由：微服务多可用区部署的情况下，确保流量优先在同一个可用区内流转，降低业务的整体时延。
- 就近容灾路由：当某个可用区发生故障，可以把流量尽快的切到正常的可用区，让业务以最快速度恢复。

第二章：微服务治理技术原理介绍

2.1 微服务治理技术概述

正如第一章所说，服务治理从趋势上来说正在向无侵入，和业务解耦的方向发展。首先介绍一下阿里巴巴内部在服务治理技术形态上的演进路线。

阿里巴巴服务治理技术演进路线



第一阶段： 自研微服务

阿里巴巴的微服务拆分实践进行的很早，从 2008 年就开始了，当时的单体应用已经无法承载业务迭代的速度，由五彩石项目开始了微服务化的改造，在这个改造过程中，也逐步诞生了服务框架，消息队列，数据库分库分表等三大中间件。在这个阶段的服务治理能力是通过 SDK方式直接依赖在框架里面的。每个中间件都有自己独立的 SDK 依赖，服务治理能力的升级需要借助框架 SDK 的升级来解决，升级成本是很高的。

第二阶段：Fat-SDK

随着中间件接入数量的增加，业务升级成本不断攀升，从 2013 年起诞生了代号 “Pandora” 的项目，主要有 2 个目标，一是解决中间件和业务依赖的冲突问题，二是解决服务治理升级效率的问题。同一个组件，业务和中间件的可能依赖不同的版本，最常见的例如日志，序列化组件等等，如果大家共享一个版本则会出现中间件的升级影响到业务，或者出现不兼容的情况。Pandora 提供了一个轻量的隔离容器，通过类加载器隔离的方式，将中间件和业务的依赖互相隔离，而中间件和中间件之间的依赖也能互相隔离。另外，通过 Fat-SDK 的方式，将所有中间件一次性打包交付给业务方升级。这一点和 Maven 引入的 bom 的思路类似，但是相比 bom 来说每个 Pandora 的插件都可以享有独立的依赖。通过这种方式，业务不再需要单独升级某个中间件，而是一次性把所有的中间件完成升级，从而大幅提升了中间件升级的效率。

第三阶段：One Java Agent

随着业务的进一步发展，中间件的数量逐步增加，Pandora 的方式也遇到了相当多的问题，也就是如果要把一个 Pandora 的版本在全集团内全部推平，需要长达 1 年的时间才能完成。这是因为即使是 Pandora 的方式，也需要业务修改代码，升级，验证，发布，这些并非业务真正关心，业务更希望专注于自身业务的发展。通常借助双十一大促这样的机会，才有可能完成中间件的升级。这也给服务治理的形态带来新的挑战。2019 年，阿里推出了 One Java Agent 的形态，把服务治理的能力下沉到 Java Agent 的形式，通过无侵入的方式，实现了中间件的迭代升级，进一步提升了升级效率。

第四阶段：One Mesh

Java Agent 通常只能解决 Java 语言构建的微服务，针对非 Java 语言构建的微服务体系，阿里也借助 Service Mesh 的方式，把服务治理能力下沉到 sidecar，实现了和业务的解耦。通过 sidecar 的方式，不同语言的能力无需重复开发，sidecar 的升级也可以做到透明，对业务无感。

值得注意的是，无论是 SDK 的形态，还是 Agent 的形态，还是 sidecar 的形态，对于服务治理的控制台来说，都需要统一的控制面对多种数据面进行控制。

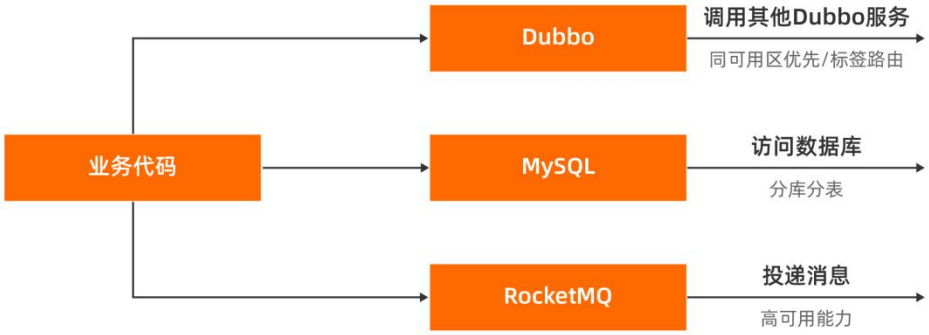
2.2 通过 SDK 方式进行微服务治理

独立 SDK 模式

阿里巴巴通过五彩石项目开始了微服务化的改造，逐步诞生了服务框架，消息队列，数据库分库分表三大中间件。这些中间件都是通过 SDK 的方式被业务项目直接依赖。

在这个阶段，由于基础设施团队对部署、开发模式比较熟悉，业务接入的模式也比较单一。业务方单独对接每个 SDK 是比较合适的。

我们以 Java 应用为例，理论上的依赖应该是这样的：



在这个阶段的服务治理，主要依赖于各个 SDK 提供的能力。比如 HSF/Dubbo 提供了同可用区优先、标签路由能力；消息队列提供了高可用能力；数据库分库分表提供了读写分离、动态分库能力。

Fat-SDK 模式

通过单一 SDK 提供服务治理能力后，各个 SDK 的提供方需要对提供出去的 SDK 进行支持，需要及时替换、升级旧版本的 SDK，并及时督促 SDK 使用方进行升级。

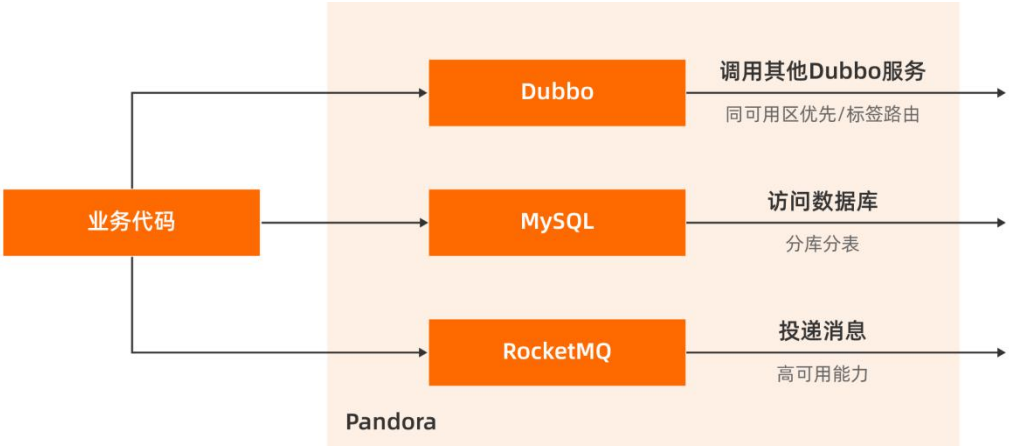
在这个过程中，总是发现 SDK 之间会有依赖冲突，常常遇到 sdk 冲突导致应用无法启动、sdk 被第三方包降级导致功能丢失；督促使用者升级也越来越耗费了大量的时间和精力。

业务研发同学作为 SDK 的使用方，需要关注各个 SDK 提供的新功能、旧版本是否已经不再支持等。业务研发同学的时间和精力越来越多的被中间件组件消耗了。

为了解决这个痛点，2013 年诞生了代号 “Pandora” 的项目。Pandora 借助类加载器隔离能力，让各个中间件组件能够有自己独立依赖且互不冲突；Pandora 也作为一个应用容器，给业务开发同学提供了一整套中间件解决方案。

业务同学只需要使用指定版本的 Pandora，后续的维护、升级只需要跟随 Pandora 版本就可以。

中间件维护同学，也只需要提供 SDK 与 Pandora 集成，并通过统一的发版周期交付给业务同学，极大地减轻了支持维护成本。



2.3 通过 Java Agent 方式进行微服务治理

中间件作为提供方，苦于业务方不能及时升级中间件到最新版。业务方作为使用方，苦于升级成本比较高。

Java Agent 技术，能够在运行时动态修改 Java 字节码，动态的改变 Java 程序的行为，能够很好的满足这种需求。

Java Agent 技术

在 JVM 启动的时候，可以通过 `-javaagent:/path/to/agent.jar` 的方式来加载 Java Agent。在 Java Agent 中，实现 ClassFileTransformer 接口，并调用Instrumentation.addTransformer 将 Transformer 添加到系统中。

接下来加载类的时候，或者通过 retransformClasses 触发类重新加载的时候，Transformer 就可以修改类的字节码，比如去除掉某一段代码，在原来的方法前后执行额外逻辑等等。

基于 ClassFileTransformer 能力，中间件只需要提供一个标准的轻量级 SDK，提供简单实现和接口；剩余的比较重的逻辑，就可以写在 Java Agent 中，通过 ClassFileTransformer 的方式动态插入到 Java 代码中。

业务方也只需要依赖一个中间件 SDK 作为接口，由于主要逻辑都在 Java Agent 中，业务方就避免了频繁升级 SDK 的成本，专注于业务研发。以开源的 Java 应用为例，业务方只需要依赖一个版本的 Apache Dubbo 或者 Spring Cloud 来作为微服务框架，专注于写业务逻辑。在部署的时候，通过 Java Agent 技术加载 Java Agent，实现服务治理功能。

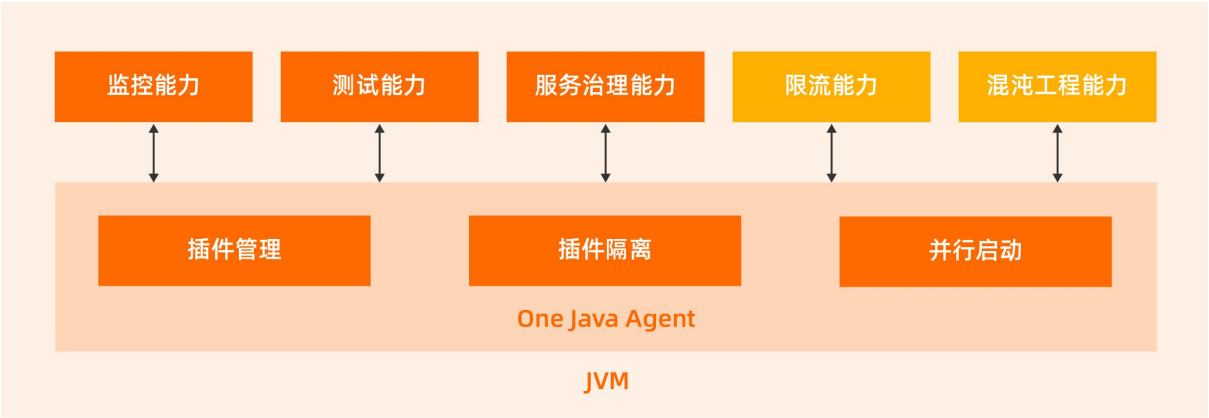
如果后续中间件提供更多能力，只需要升级 Java Agent 即可，业务开发人员不用改一行代码。

One Java Agent

中间件分为很多不同的部分，比如 RPC、消息、数据库等。中间件内部也需要一些机制来保证 Java Agent 中各个中间件的代码能够独立开发、部署，且尽可能做到互不影响。

所以阿里云内部将各个中间件的 Java Agent 作为插件(plugin)，组装成一个统一的 Java Agent，称为 One Java Agent。

- 每个 plugin 可以由启动参数来单独控制是否开启。
- 各个 plugin 的启动是并行的，将 Java Agent 的启动速度由 O(n+m+...)提升至O(n)。
- 各个 plugin 的类，都由不同的类加载器加载，最大限度隔离了各个 plugin。
- 每个 plugin 的状态都可以上报到服务端，可以通过监控来检测各个 plugin 是否有问题。



目前 One Java Agent 项目已经开源：https://github.com/alibaba/one-java-agent。

云原生场景下如何自动注入 Java Agent

对于 Java Agent，需要业务容器启动的时候给 Java 命令行添加启动参数 `-javaagent:<jarpath> [=<options>]` ，这需要重新构建容器镜像。对于大规模的业务统一接入，需要重新构建全部容器镜像。这对于运维来说是及其繁琐的事情。因此我们需要一种方式，能够将自动注入 Java Agent 与生成容器镜像进行解耦。

而 Java 提供了 `JAVA_TOOL_OPTIONS` 环境变量：在 JVM 启动时，JVM 会读取并应用此环境变量的值，这样我们就可以通过在容器镜像中设置环境变量：`JAVA_TOOL_OPTIONS=-javaagent:/path/to/agent.jar` ，从而实现不用修改镜像，就可以加载 Java Agent 了。

在 Kubernetes 环境中，我们可以借助 webhook 能力，来实现自动注入 `JAVA_TOOL_OPTIONS` ，从而达到自动开启服务治理的功能。

配置 Webhook，然后根据 Pod 或者 Namespace 中的 Labels，来判断是否要挂载 Java Agent。如果需要挂载，则就对 Pod 的声明文件做出如下修改：

- 获取并添加环境变量 JAVA_TOOL_OPTIONS，用于加载 Java Agent。
- 给业务容器添加 Volume，用于存储 Java Agent 的文件内容。
- 给 Pod 添加 Init container，用于在业务容器启动前下载 Java Agent。

最终，我们将此功能模块提供为 Helm 包，运维人员可以一键安装。如果要接入服务治理，也只需要给对应资源添加 label 即可接入。

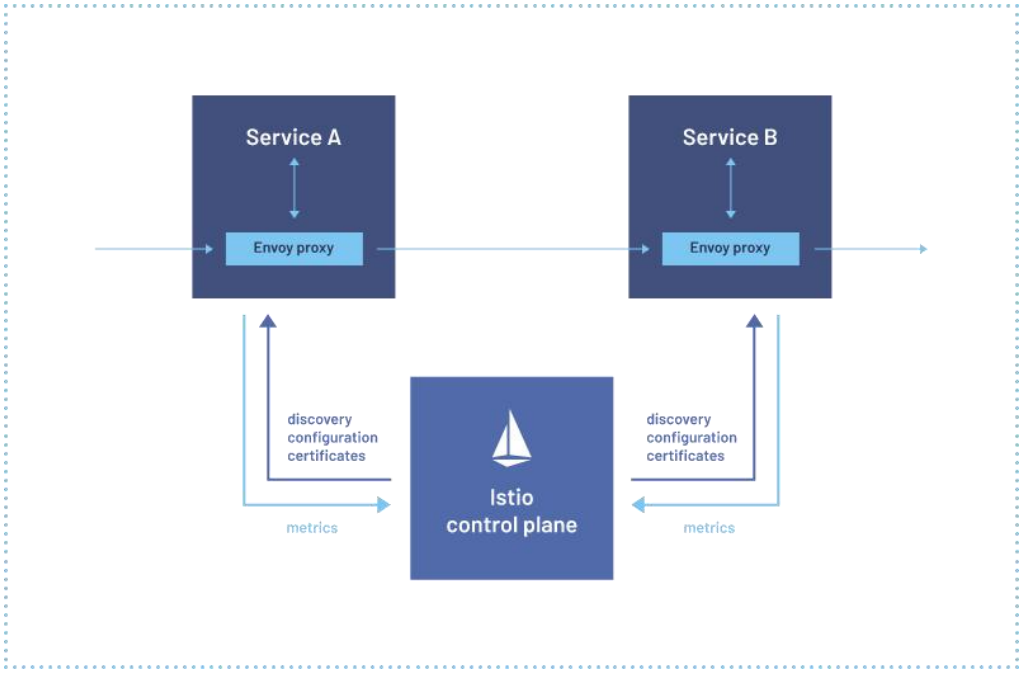
2.4 通过 Service Mesh 来进行微服务治理

Service Mesh 的出现，为异构微服务体系治理打开了一种全新的思路。

- 2016 年可以说是 Service Mesh 的元年，Buoyant 公司 CEO William Morgan 率先发布 Linkerd ，成为业界首个 Service Mesh 项目，同年 Lyft 发布 Envoy ，成为第二个 Service Mesh 项目。
- 2017 年，Google、IBM、Lyft 联手发布了 Istio，它与 Linkerd / Envoy 等项目相比，它首次给大家增加了控制平面的概念，提供了强大的流量控制能力。经过多年的发展 Istio，已经逐步成为 控制平面的事实标准。
- 1.0 版本的问世标志着 Istio 进入了可以生产可用的时代,越来越多的企业将服务网格应用于生产中。
- 1.5 版本开始将原有的多个组件整合为一个单体结构 istiod；同时废弃了被诟病已久的 Mixer 组件，统一为 Istiod 服务，方便部署和运维。

Istio 的架构

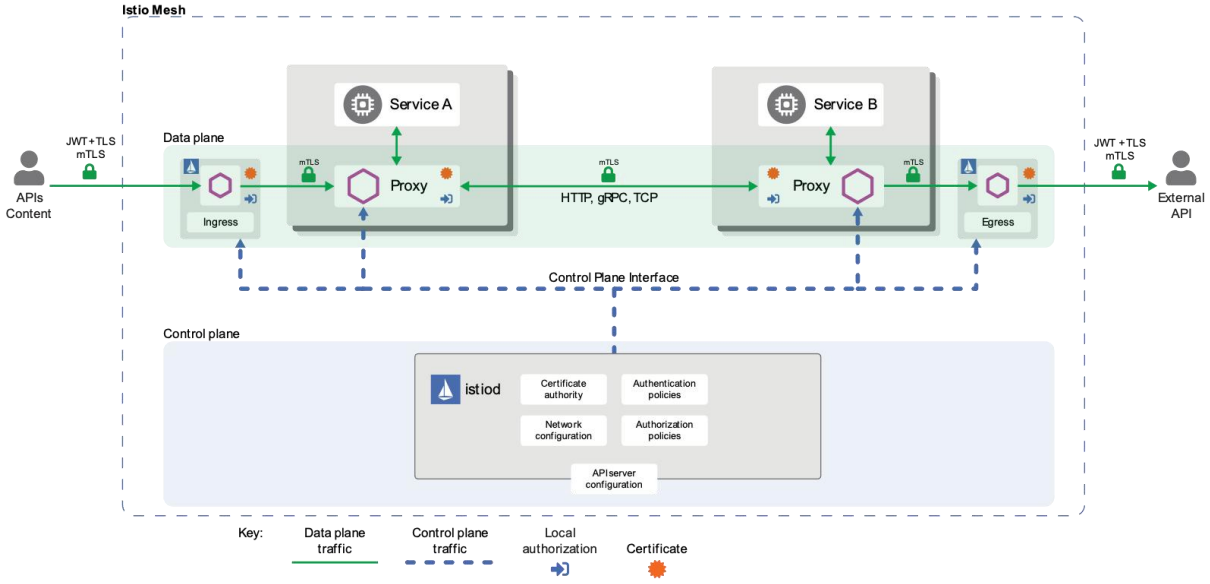
下图是一个 Istio 的典型架构：



出自：https://istio.io/latest/about/service-mesh/

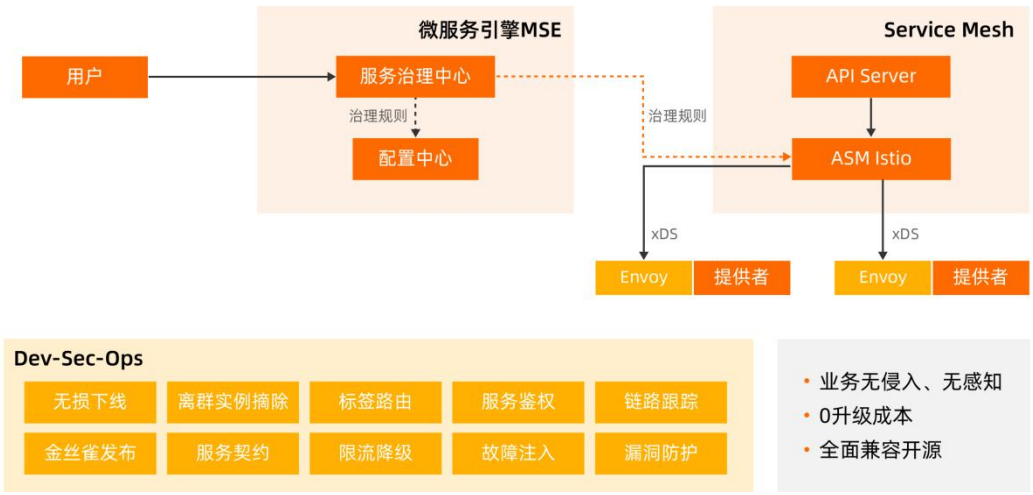
Istiod 作为控制面的统一组件，负责对接服务注册发现、路由规则管理、证书管理等能力，Envoy Proxy 作为数据面Sidecar 代理业务流量，Istio 和 Envoy Proxy 之间通过 XDS 接口完成服务发现、路由规则等数据的传递，同时 Istio 也提供了 MCP Over XDS 接口对接外部注册中心，如 Nacos。

Istio 除了支持东西向的流量代理之外，还支持南北流量的代理，通过 Istio Ingress Gateway 作为入口的网关，通过 Istio Egress Gateway 作为出口网关，这样 Istio 将可以对全域流量进行治理。



出自：https://istio.io/latest/docs/concepts/security/

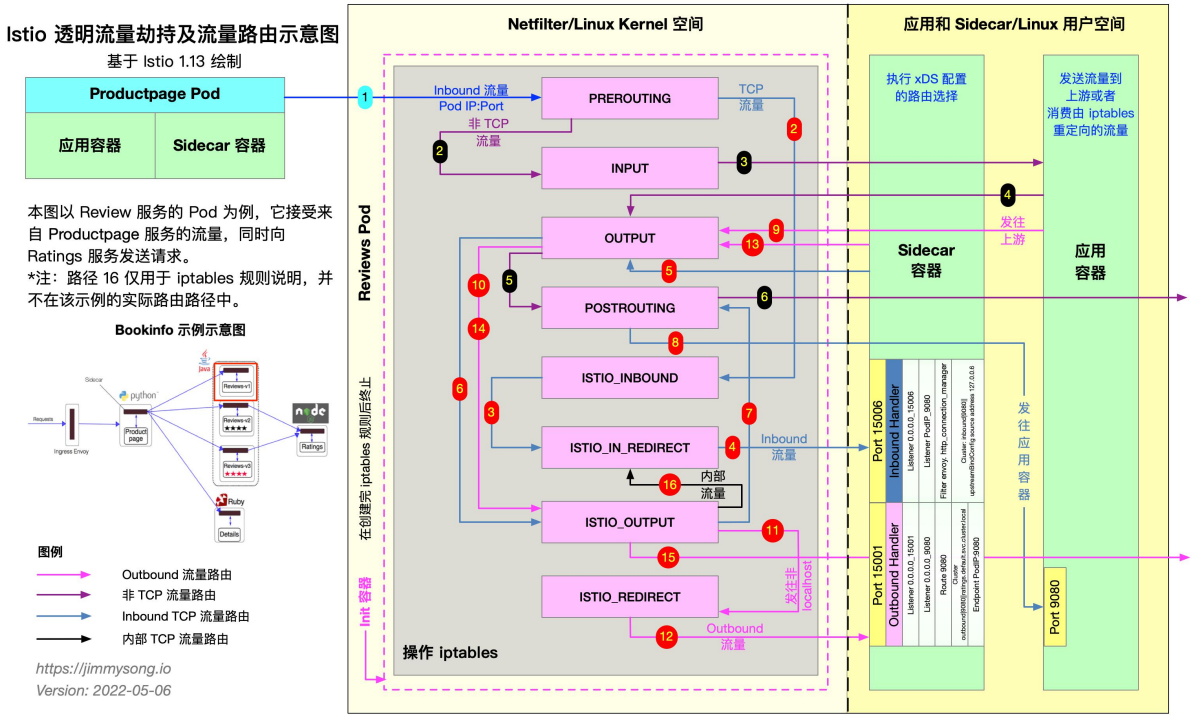
在云原生微服务治理中提供了基于 ASM Istio 的服务治理能力，包含标签路由、服务鉴权、故障注入、金丝雀发布、服务测试、无损上下线等。



流量劫持能力

iptables 是 Linux 内核中的防火墙软件 netfilter 的管理工具，位于用户空间，同时也是 netfilter 的一部分。Netfilter 位于内核空间，不仅有网络地址转换的功能，也具备数据包内容修改、以及数据包过滤等防火墙功能。

在 kubernetes 的方案中，Istio 会在开启了 Sidecar 注入标记的 Pod 中注入一个 Init Container 和一个普通容器 istio-proxy，Init Container 容器会执行一段 iptables 脚本，根据脚本的参数，将 Inbound 和 OutBound 流量都导入到 Envoy 指定的端口上，Istio-Proxy 容器由 2 个进程组成，Pilot Agent 会用来获取 Envoy 的启动配置，然后创建一个 Envoy 的进程，同时会对 Envoy 进程进行健康检查。



出自：https://jimmysong.io/blog/sidecar-injection-iptables-and-traffic-routing/

基于 eBPF 流量劫持能力

由于 eBPF 技术的兴起，在可观测性和网络包的处理上有了不少优秀的实践，像 Cilium 被大家所熟知，使用 eBPF 的 sockops 和 redir 等能力，可以高效地处理数据包。

使用 iptables 的技术，需要对出入口都拦截，会让原本只需在内核态处理两次的链路，变成四次，造成大量的性能损失，这对一些性能要求高的场景有明显的影响。相比于基于 iptables 的流量劫持技术，基于 eBPF 的技术可以将应用发出的包直接转发到对端的 socket，加速包在内核中的处理流程。

eBPF 在做流量劫持的过程，针对出口流量，需要修改连接的目的地址，发送到新的端口，同时要记住之前的目的地址，这样便于 Envoy 将流量进行转发，针对入口流量也是类似的，只是需要将劫持到的 Envoy 端口修改为 15006 端口。

流量路由过程

流量路由分为 Inbound 和 Outbound 两个过程，Inbound 即为进入Pod 的流量，OutBound 即为 Pod 访问出去的流量，以下将描述流量的处理流程：

Inbound handler 的作用是将 iptables 拦截到的 downstream 的流量转交给 localhost，与 Pod 内的应用程序容器建立连接，可以通过访问 15000 的 admin 接口查看业务 Pod 的 Listener 列表，从中可以看到 0.0.0.0:15006/TCP 的 Listener（其实际名字是 virtualInbound）监听所有的 Inbound 流量，Inbound handler 的流量被 virtualInbound Listener 转移到 Pod 的指定端口的 Listener，然后找到对应的 Cluster 和 Endpoint 进行转发，这样流量顺利到达业务容器指定的端口。

Outbound handler 的作用是将 iptables 拦截到的本地应用程序发出的流量，经由 sidecar 判断如何路由到 upstream。应用程序容器发出的请求为 Outbound 流量，被 iptables 劫持后转移给 Outbound handler 处理，然后经过 virtualOutbound Listener、服务端对应端口的 Listener，然后通过 Route 指定的端口号找到 upstream 的 cluster，进而通过 EDS 找到 Endpoint 执行路由动作。

基于 Envoy Filter 的服务治理

Istio 通过 Sidecar 将服务治理的能力进行了标准化和统一化，比如故障注入、金丝雀发布、负载均衡、服务鉴权，同时还提供了可观测的能力，如 Metrics、日志。

Envoy 是 Istio 中的 Sidecar 官方标配，是一个面向服务架构的高性能网络代理，由 C++ 语言实现，拥有强大的定制化能力。Envoy 是面向服务架构设计的 L7 代理和通信总线，核心是一

个 L3/L4 网络代理。可插入 filter 链机制允许开发人员编写 filter 来执行不同的 TCP 代理任务并将其插入到主体服务中。Envoy 还支持额外的 HTTP L7 filter 层。可以将 HTTP filter 插入执行不同任务的 HTTP 连接管理子系统，可以查看每个 Envoy 的 Config Dump 看到 Http filter 配置，Envoy 根据这些配置决定如何执行对应的流量处理流程。



Envoy 提供的过滤器包括侦听器过滤器（Listener Filters）、网络过滤器（Network Filters）、HTTP 过滤器（HTTP Filters）三种类型，它们共同组成了一个层次化的过滤器链，侦听器过滤器在初始连接阶段访问原始数据并操作 L4 连接的元数据，网络过滤器访问和操作 L4 连接上的原始数据，即 TCP 数据包，HTTP 过滤器在 L7 上运行，由网络过滤器（即 HTTP 连接管理器，HTTP Connection Manager） 创建。这些过滤器用于访问、操作 HTTP 请求和响应。

第三章：微服务治理在云原生场景下的解决方案

3.1 线上发布稳定性解决方案

绝大多数的软件应用生产安全事故发生在应用上下线发布阶段，尽管通过遵守业界约定俗成的可灰度、可观测和可滚回的安全生产三板斧，可以最大限度的规避发布过程中由于应用自身代码问题对用户造成的影响。但对于高并发大流量情况下的短时间流量有损问题却仍然无法解决。因此，本节将围绕发布过程中如何解决流量有损问题实现应用发布过程中的无损上下线效果相关内容展开方案介绍。

无损上下线背景

据统计，应用的事故大多发生在应用上下线过程中，有时是应用本身代码问题导致。但有时我们也会发现尽管代码本身没有问题，但在应用上下线发布过程中仍然会出现短时间的服务调用报错，比如调用时出现 Connection refused 和 No instance 等现象。相关问题的原因有相关发布经历的同学或多或少可能有一定了解，而且大家发现该类问题一般在流量高峰时刻尤为明显，半夜流量少的时候就比较少见，于是很多人便选择半夜三更进行应用发布希望以此来规避线上发布事故。本节将就这些问题出现的背后真实原因以及业界对应的设计方案展开介绍。常见的流量有损现象出现的原因包括但不限于以下几种：

- **服务无法及时下线：**服务消费者感知注册中心服务列表存在延时，导致应用特定实例下线后在一段时间内服务消费者仍然调用已下线实例造成请求报错。

- **初始化慢：**应用刚启动接收线上流量进行资源初始化加载，由于流量太大，初始化过程慢，出现大量请求响应超时、阻塞、资源耗尽从而造成刚启动应用宕机。

- **注册太早：**服务存在异步资源加载问题，当服务还未初始化完全就被注册到注册中心，导致调用时资源未加载完毕出现请求响应慢、调用超时报错等现象。

- **发布态与运行态未对齐：**使用Kubernetes 的滚动发布功能进行应用发布，由于 Kubernetes 的滚动发布一般关联的就绪检查机制，是通过检查应用特定端口是否启动作为应用就绪的标志来触发下一批次的实例发布，但在微服务应用中只有当应用完成了服务注册才可对外提供服务调用。因此某些情况下会出现新应用还未注册到注册中心，老应用实例就被下线，导致无服务可用。

接下来，将就具体的下线和上线过程中如何避免流量损耗问题进行分别介绍。

无损下线

由于微服务应用自身调用特点，在高并发下，服务提供端应用实例的直接下线，会导致服务消费端应用实例无法实时感知下游实例的实时状态因而出现继续将请求转发到已下线的实例从而出现请求报错，流量有损。

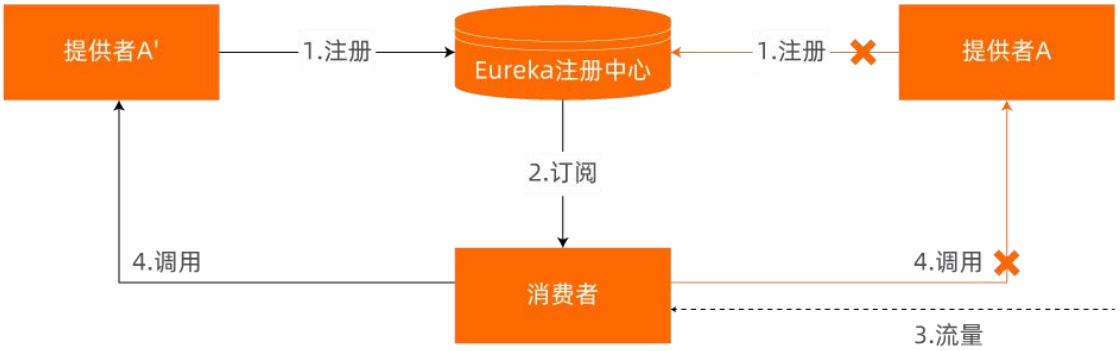


图 1 Spring Cloud 应用消费者无法及时感知提供者服务下线

例如对于 Spring Cloud 应用如上图 1 所示，当应用的两个实例 A'和 A 中的 A 下线时，由于 Spring Cloud 框架为了在可用性和性能方面做平衡，消费者默认是 30s 去注册中心拉取最新的服务列表，因此 A 实例的下线不能被实时感知，流量较大时，消费者会继续通过本地缓存调用已下线的 A 实例导致出现流量有损。基于上述背景，业界提出了相应的无损下线（也叫优雅下线）的技术方案来应对上述问题。本节将对业界主流的一些无损下线技术方案进行介绍。

针对该类问题，业界一般的解决方式是通过将应用更新流程划分为手工摘流量、停应用、更新重启三个步骤。由人工操作实现客户端避免调用已下线实例，这种方式简单而有效，但是限制

较多：不仅需要借助流控能力来实现实时摘流量，还需要在停应用前人工判断来保证在途请求已经处理完毕。这种需要人工介入的方式运维复杂度较高，只适用于规模较小的应用，无法解决当前云原生架构下，自动化的弹性伸缩、滚动升级等场景中的实例下线过程中的流量有损问题。本节将对业界应用于云原生场景中的一些无损下线技术方案进行介绍。

主动通知

一般注册中心都提供了主动注销接口供微服务应用正常关闭时调用，以便下线实例能及时更新其在注册中心上的状态。主动注销在部分基于事件感知注册中心服务列表的微服务框架比如 Dubbo 中能及时让上游服务消费者感知到提供者下线避免后续调用已下线实例。但对于像 Spring Cloud 这类微服务框架服务消费者感知注册中心实例变化是通过定时拉取服务列表的方式实现。尽管下线实例通过注册中心主动注销接口更新了其自身在注册中心上的应用状态信息但由于上游消费者需要在下一次拉取注册中心应用列表时才能感知到，因此会出现消费者感知注册中心实例变化存在延时。在流量较大、并发较高的场景中，当实例下线后，仍无法实现流量无损。既然无法通过注册中心让存量消费者实例实时感知下游服务提供者的变化情况，业界提出了利用主动通知解决该类问题。主动通知过程如下图 2 所示：

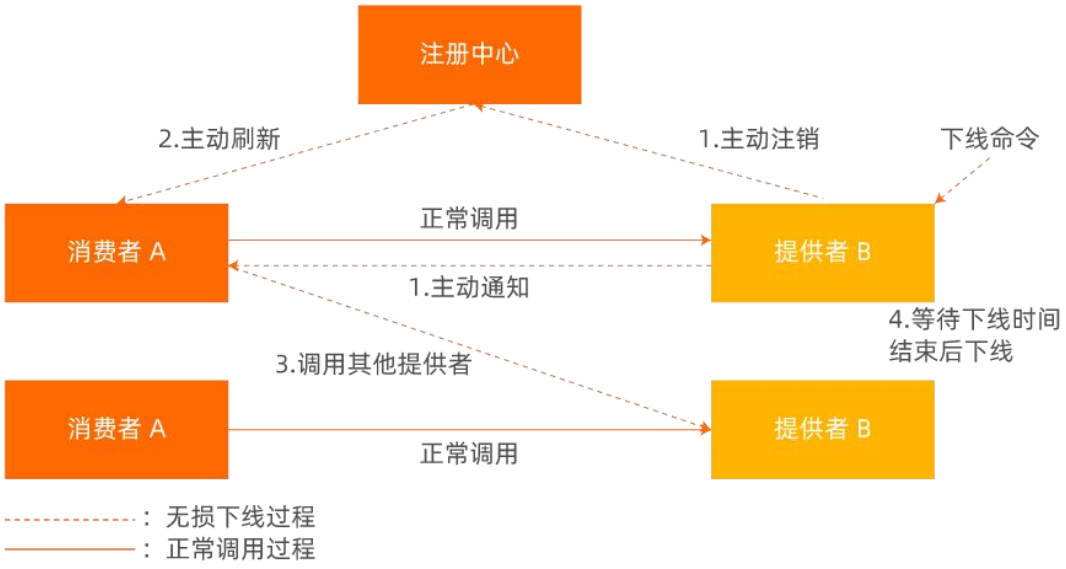


图 2 无损下线方案

如图 2 所示，服务提供者 B 中某个实例在下线时为避免主动在注册中心中注销的服务实例状态无法实时被上游消费者 A 感知到，从而导致调用已下线实例的问题。在接收到下线命令即将下线前，提供者 B 对于在等待下线阶段内收到的请求，在其返回值中都增加上特殊标记让服务消费者接收到返回值并识别到相关标志后主动拉取一次注册中心服务实例从而实时感知 B 实例最新状态，从而达到服务提供者的下线状态能够被服务消费者实时感知。

自适应等待

在并发度不高的场景下，主动通知方法可以解决绝大部分应用下线流量有损问题。但对于高并发大流量应用下线场景，如果主动通知完，可能仍然存在一些在途请求需要待下线应用处理完才能下线否则这些流量就无法正常被响应。为解决该类在途请求问题，可通过给待下线应用在下线前通过自适应等待机制在处理完所有在途请求后，再下线以实现流量无损。

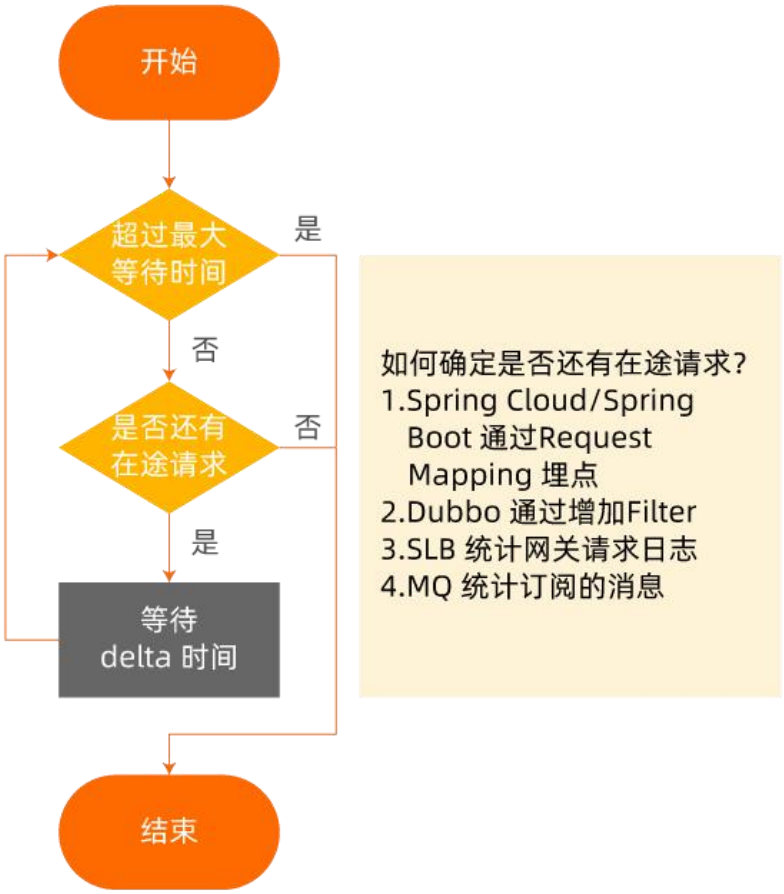


图 3 自适应等待机制

如上图 3 所示，自适应等待机制是通过待下线应用统计应用中是否仍然存在未处理完的在途请求，来决定应用下线的时机，从而让待下线应用在下线前处理完所有剩余请求。

无损上线

延迟加载是软件框架设计过程中最常见的一种策略，例如在 Spring Cloud 框架中 Ribbon 组件的拉取服务列表初始化默认都是要等到服务的第一次调用时刻，例如下图 4 是 Spring Cloud 应用中第一次和第二次通过调用RestTemplate 调用远程服务的耗时对比情况：

```
[arthas@37035]$ trace com.alibaba.mse.consumer.TestController eurekaRest  -n 5
 --skipJDKMethod false
Press Q or Ctrl+C to abort.
Affect(class count: 1 , method count: 1) cost in 105 ms, listenerId: 1
`---ts=2022-02-14 21:28:02;thread_name=http-nio-18099-exec-1;id=39;is_daemon=t
rue;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbedde
dWebappClassLoader@60e5272
    `---[464.275852ms] com.alibaba.mse.consumer.TestController:eurekaRest()
        `---[464.018509ms] org.springframework.web.client.RestTemplate:getForO
bject() #50

`---ts=2022-02-14 21:28:08;thread_name=http-nio-18099-exec-3;id=3b;is_daemon=t
rue;priority=5;TCCL=org.springframework.boot.web.embedded.tomcat.TomcatEmbedde
dWebappClassLoader@60e5272
    `---[8.46028ms] com.alibaba.mse.consumer.TestController:eurekaRest()
        `---[8.402525ms] org.springframework.web.client.RestTemplate:getForObj
ect() #50
```

图 4 应用启动资源初始化与正常运行过程中耗时情况对比

由图 4 结果可见，第一次调用由于进行了一些资源初始化，耗时是正常情况的数倍之多。因此把新应用发布到线上直接处理大流量极易出现大量请求响应慢，资源阻塞，应用实例宕机的现象。

业界针对上述应用无损上线场景提出如下包括延迟注册、小流量服务预热以及就绪检查等一系列解决方案，详细完整的方案如下图 5 所示：

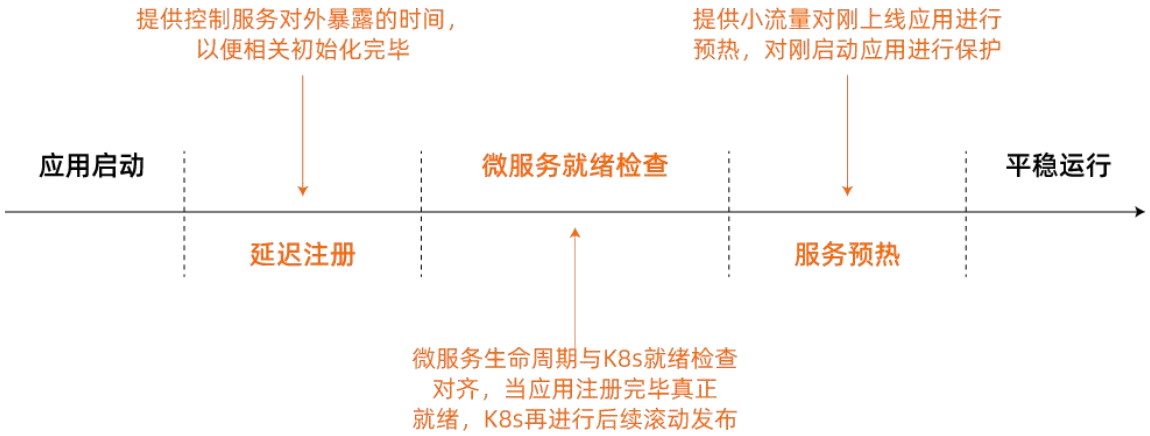


图 5 无损上线整体方案

延迟注册

对于初始化过程需要异步加载资源的复杂应用启动过程，由于注册通常与应用初始化过程同步进行，从而出现应用还未完全初始化就已经被注册到注册中心供外部消费者调用，此时直接调

用由于资源未加载完成可能会导致请求报错。通过设置延迟注册，可让应用在充分初始化后再注册到注册中心对外提供服务。例如开源微服务治理框架 Dubbo 原生就提供延迟注册功能[1]。

小流量服务预热

在线上发布场景下，很多时候刚启动的冷系统直接处理大量请求，可能由于系统内部资源初始化不彻底从而出现大量请求超时、阻塞、报错甚至导致刚发布应用宕机等线上发布事故出现。为了避免该类问题业界针对不同框架类型以及应用自身特点设计了不同的应对举措，比如针对类加载慢问题有编写脚本促使 JVM 进行预热、阿里巴巴集团内部 HSF（High Speed Framework）使用的对接口分批发布、延迟注册、通过 mock 脚本对应用进行模拟请求预热以及小流量预热等。本节将对其中适用范围最广的小流量预热方法进行介绍。

相比于一般场景下，刚发布微服务应用实例跟其他正常实例一样一起平摊线上总 QPS。小流量预热方法通过在服务消费端根据各个服务提供者实例的启动时间计算权重，结合负载均衡算法控制刚启动应用流量随启动时间逐渐递增到正常水平的这样一个过程帮助刚启动运行进行预热，详细 QPS 随时间变化曲线如图 6 所示：

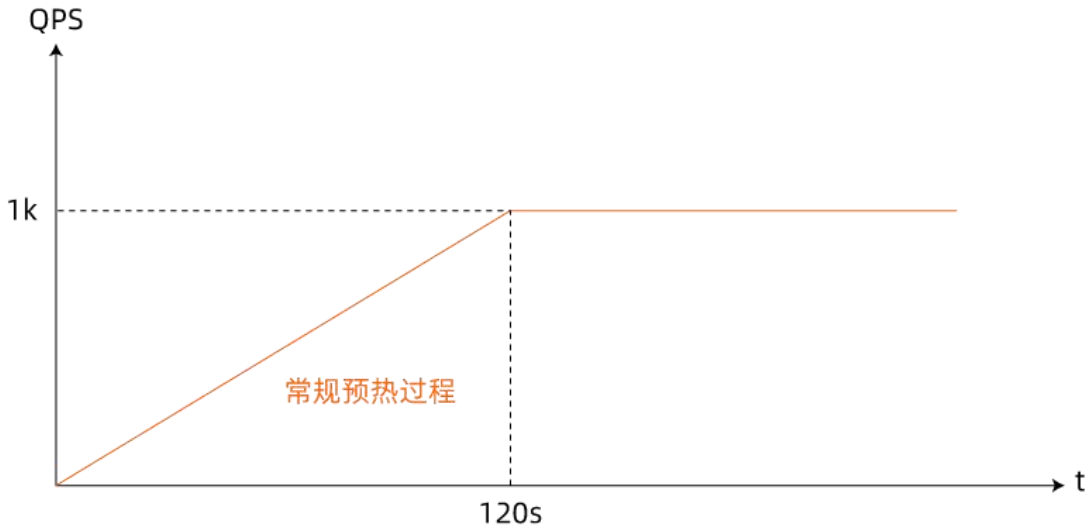


图 6 应用小流量预热过程 QPS 曲线

开源 Dubbo 所实现的小流量服务预热过程原理如下图 7 所示：



图 7 应用小流量预热过程原理图

服务提供端在向注册中心注册服务的过程中，将自身的预热时长 WarmupTime、服务启动时间 StartTime 通过元数据的形式注册到注册中心中，服务消费端在注册中心订阅相关服务实例列表，调用过程中根据 WarmupTime、StartTime 计算个实例所分批的调用权重。刚启动 StartTime 距离调用时刻差值较小的实例权重下，从而实现对刚启动应用分配更少流量实现对其进行小流量预热。

开源 Dubbo 所实现的小流量服务预热模型计算如下公式所示：

$$f(x) = k * \frac{(x - startTime)}{warmupTime}$$

模型中应用 QPS 对应的 f(x) 随调用时刻 x 线性变化，x 表示调用时刻的时间，startTime 是应用开始时间，warmupTime 是用户配置的应用预热时长，k 是常数，一般表示各实例的默认权重。

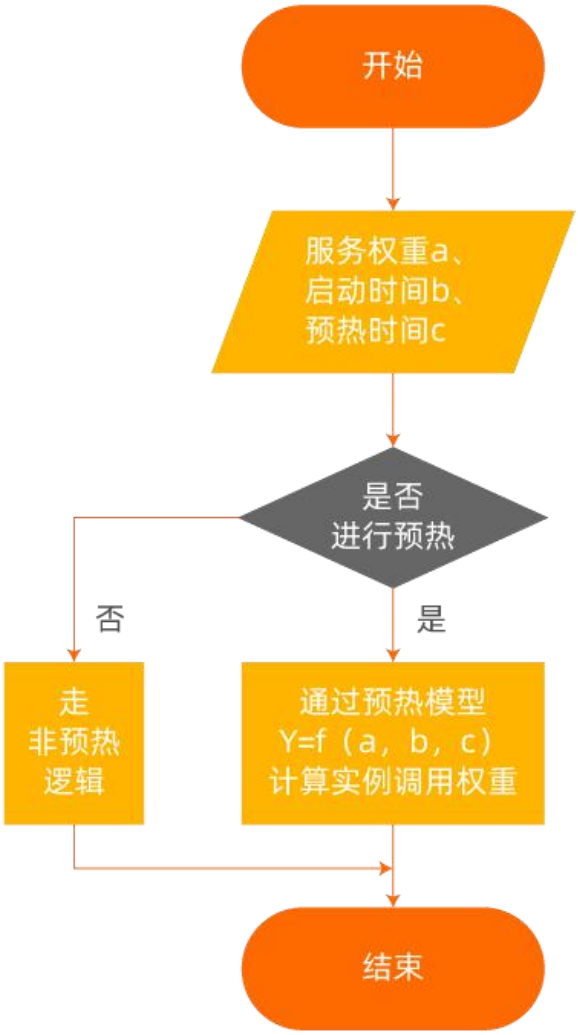


图 8 应用小流量预热权重计算

通过小流量预热方法，可以有效解决，高并发大流量下，资源初始化慢所导致的大量请求响应慢、请求阻塞，资源耗尽导致的刚启动应用宕机事故。

微服务就绪检查

在介绍微服务就绪检查之间，先简单介绍一下相关的 Kubernetes 探针技术作为技术背景，以便更好的理解后文内容：

Kubernetes 探针技术

在云原生领域，Kubernetes 为了确保应用 Pod 在对外提供服务之前应用已经完全启动就绪或者应用Pod长时间运行期间出现意外后能及时恢复，提供了探针技术来动态检测应用的运行情况，为保证应用的无损上线和长时间健康运行提供了保障。

存活探针

Kubernetes 中提供的存活探测器来探测什么时候进行容器重启。例如，存活探测器可以捕捉到死锁（应用程序在运行，但是无法继续执行后面的步骤）。在这样的情况下重启容器有助于让应用程序在有问题的情况下更可用。

就绪探针

Kubernetes 中提供的就绪探测器可以知道容器什么时候准备好了并可以开始接受请求流量，当一个 Pod 内的所有容器都准备好了，才能把这个 Pod 看作就绪了。这种信号的一个用途就是控制哪个 Pod 作为 Service 的后端。在 Pod 还没有准备好的时候，会从 Service 的负载均衡器中被剔除的。

启动探针

Kubernetes 中提供的启动探测器可以知道应用程序容器什么时候启动了。 如果配置了这类探测器，就可以控制容器在启动成功后再进行存活性和就绪检查，确保这些存活、就绪探测器不会影响应用程序的启动。 这可以用于对慢启动容器进行存活性检测，避免它们在启动运行之前就被杀掉。

探针使用小结

1.当需要在容器已经启动后再执行存活探针或者就绪探针检查，则可通过设定启动探针实现。

2.当容器应用在遇到异常或不健康的情况下会自行崩溃，则不一定需要存活探针，Kubernetes 能根据 Pod 的 restartPolicy 策略自动执行预设的操作。

3.当容器在探测失败时被 Kill 并重新启动，则可通过指定一个存活探针，并指定 restartPolicy 为 Always 或 OnFailure。

4.当希望容器仅在探测成功时 Pod 才开始接收外部请求流量，则可使用就绪探针。

Kubernetes 探针技术使用实例：
https://kubernetes.io/zh/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

当前容器+Kubernetes 的应用运维部署方式已经成为了业界的事实标准，相关技术为微服务应用运维部署带来巨大便利的同时，在某些特殊的应用部署场景中也有一些问题需要解决。比如，使用Kubernetes 的滚动发布功能进行应用发布，由于 Kubernetes 的滚动发布一般关联的就绪检查机制，是通过检查应用特定端口是否启动作为应用就绪的标志来触发下一批次的实例发布，但在微服务应用中只有当应用完成了服务注册才可对外提供服务调用。因此某些情况下会出现新应用还未注册到注册中心，老应用实例就被设置下线，导致无服务可用。

针对这样一类微服务应用的发布态与应用运行态无法对齐的问题导致的应用上线事故，当前业界也已经有相关解决方案进行应对。比如阿里云微服务引擎 MSE 就通过就绪检查关联服务注册的方法，通过字节码技术植入应用服务注册逻辑前后，然后在应用中开启一个探测应用服务是否完成注册的端口供 Kubernetes 的就绪探针进行应用就绪态探测进而绑定用户的发布态与运行态实现微服务的就绪检查，避免出现相关状态不一致导致的应用发布上线流量有损问题。

参考资料

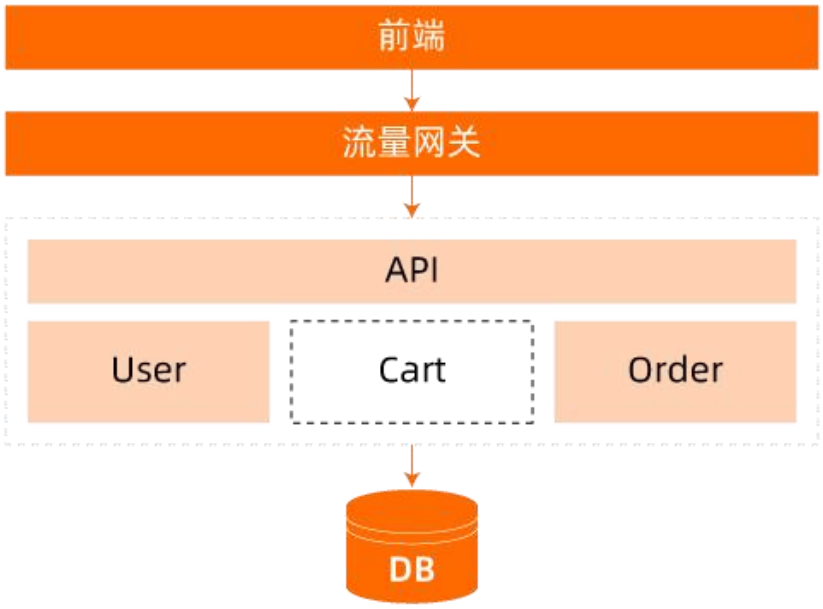
[1] Dubbo 延迟注册：https://dubbo.apache.org/zh/docs/advanced/delay-publish/

3.2 微服务全链路灰度解决方案

什么是全链路灰度

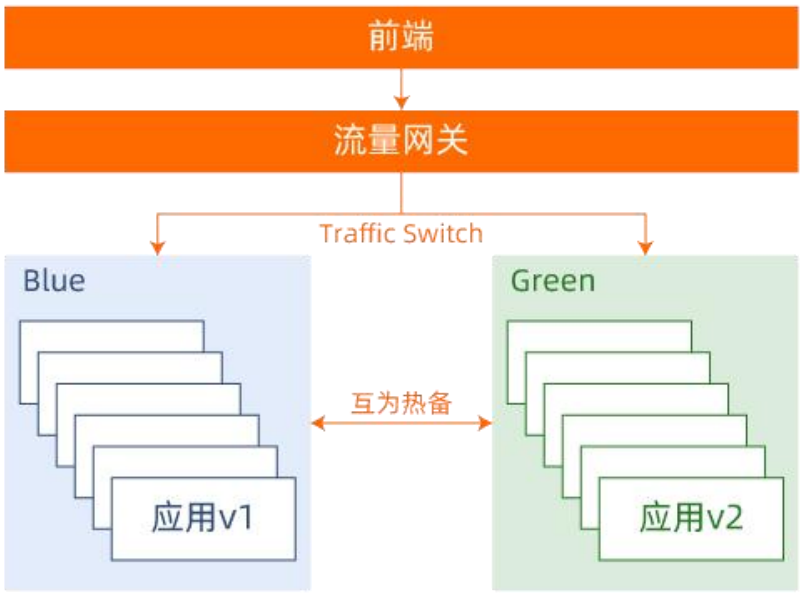
单体架构下的服务发布

首先，我们先看一下在单体架构中，如何对应用中某个服务模块进行新版本发布。如下图，应用中的 Cart 服务模块有新版本迭代：

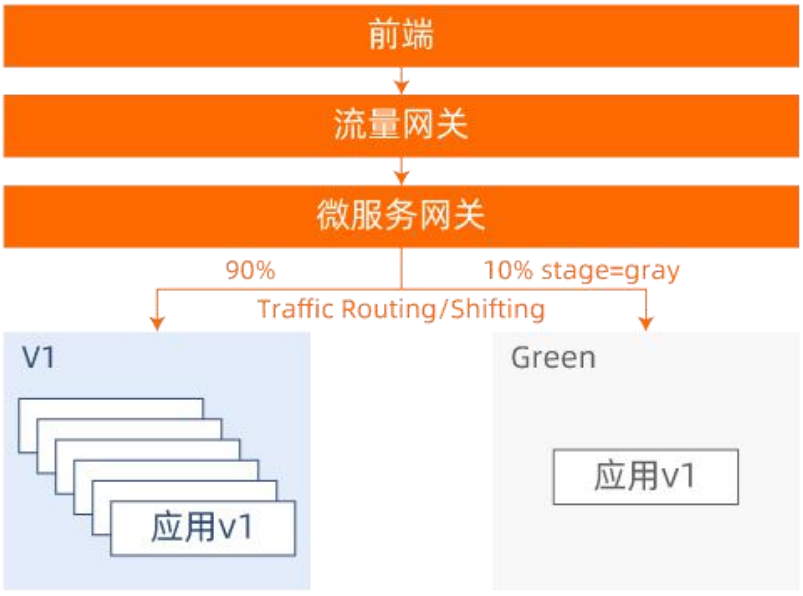


由于 Cart 服务是应用的一部分，所以新版本上线时需要对整个应用进行编译、打包以及部署。服务级别发布问题变成了应用级别的发布问题，我们需要对应用的新版本而不是服务来实施有效的发布策略。

目前，业界已经有非常成熟的服务发布方案，例如蓝绿发布和灰度发布。蓝绿发布需要对服务的新版本进行冗余部署，一般新版本的机器规格和数量与旧版本保持一致，相当于该服务有两套完全相同的部署环境，只不过此时只有旧版本在对外提供服务，新版本作为热备。当服务进行版本升级时，我们只需将流量全部切换到新版本即可，旧版本作为热备。我们的例子使用蓝绿发布的示意图如下，流量切换基于四层代理的流量网关即可完成。



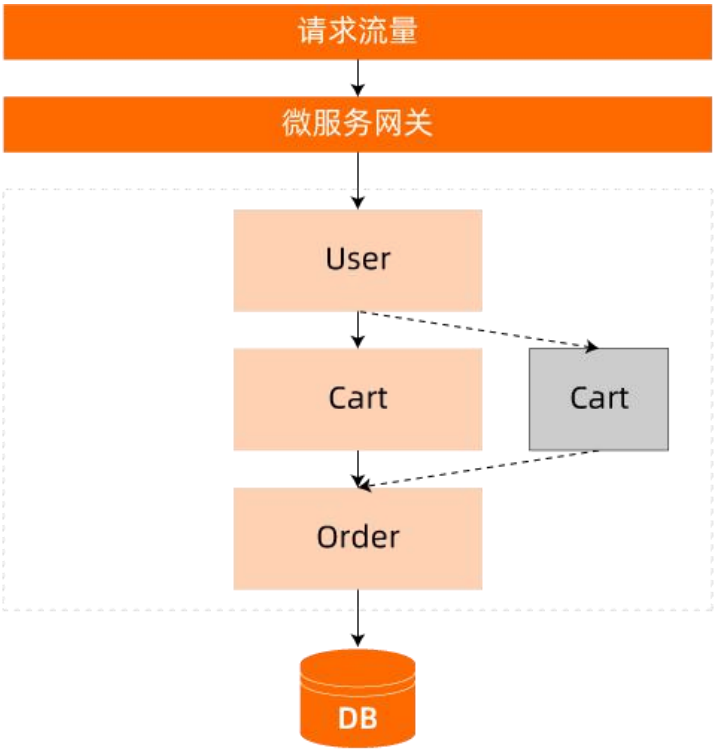
在蓝绿发布中，由于存在流量整体切换，所以需要按照原服务占用的机器规模为新版本克隆一套环境，相当于要求原来 1 倍的机器资源。灰度发布的核心思想是根据请求内容或者请求流量的比例将线上流量的一小部分转发至新版本，待灰度验证通过后，逐步调大新版本的请求流量，是一种循序渐进的发布方式。我们的例子使用灰度发布的示意图如下，基于内容或比例的流量控制需要借助于一个七层代理的微服务网关来完成。



其中，Traffic Routing 是基于内容的灰度方式，比如请求中含有头部 stag=gray 的流量路由到应用v2 版本；Traffic Shifting 是基于比例的灰度方式，以无差别的方式对线上流量按比重进行分流。相比蓝绿发布，灰度发布在机器资源成本以及流量控制能力上更胜一筹，但缺点就是发布周期过长，对运维基础设施要求较高。

微服务架构下的服务发布

在分布式微服务架构中，应用中被拆分出来的子服务都是独立部署、运行和迭代的。单个服务新版本上线时，我们再也不需要对应用整体进行发版，只需关注每个微服务自身的发布流程即可，如下：



为了验证服务 Cart 的新版本，流量在整个调用链路上能够通过某种方式有选择的路由到 Cart 的灰度版本，这属于微服务治理领域中流量治理问题。常见的治理策略包括基于 Provider 和基于 Consumer 的方式。

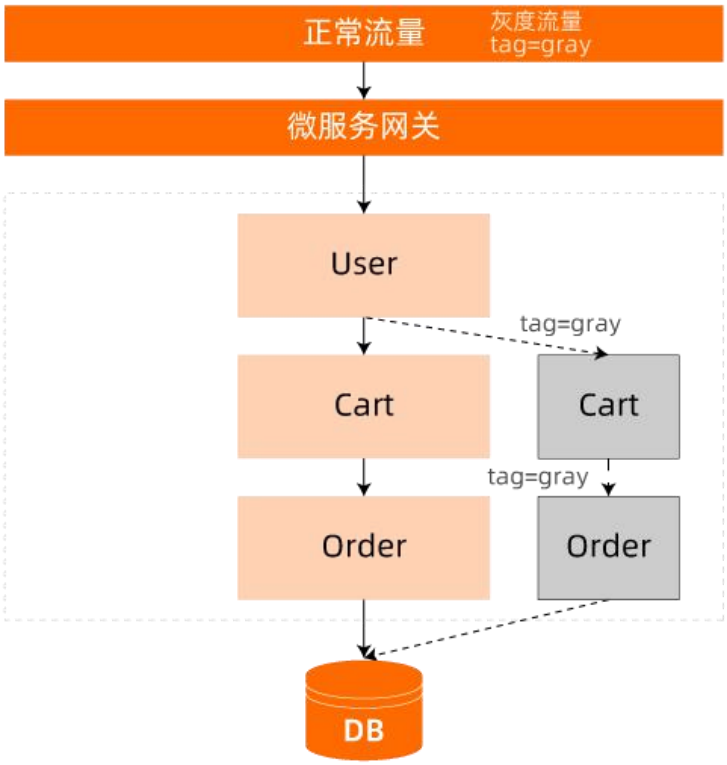
- 基于 Provider 的治理策略。配置 Cart 的流量流入规则，User 路由到 Cart 时使用Cart 的流量流入规则。

- 基于 Consumer 的治理策略。配置 User 的流量流出规则， User 路由到 Cart 时使用User 的流量流出规则。

此外，使用这些治理策略时可以结合上面介绍的蓝绿发布和灰度发布方案来实施真正的服务级别的版本发布。

全链路灰度

继续考虑上面微服务体系中对服务 Cart 进行发布的场景，如果此时服务 Order 也需要发布新版本，由于本次新功能涉及到服务 Cart 和 Order 的共同变动，所以要求在灰度验证时能够使得灰度流量同时经过服务 Cart 和 Order 的灰度版本。如下图：



按照上一小节提出的两种治理策略，我们需要额外配置服务 Order 的治理规则，确保来自灰度环境的服务 Cart 的流量转发至服务 Order 的灰度版本。这样的做法看似符合正常的操作逻辑，但在真实业务场景中，业务的微服务规模和数量远超我们的例子，其中一条请求链路可能经过数十个微服务，新功能发布时也可能会涉及到多个微服务同时变更，并且业务的服务之间依赖错综复杂，频繁的服务发布、以及服务多版本并行开发导致流量治理规则日益膨胀，给整个系统的维护性和稳定性带来了不利因素。

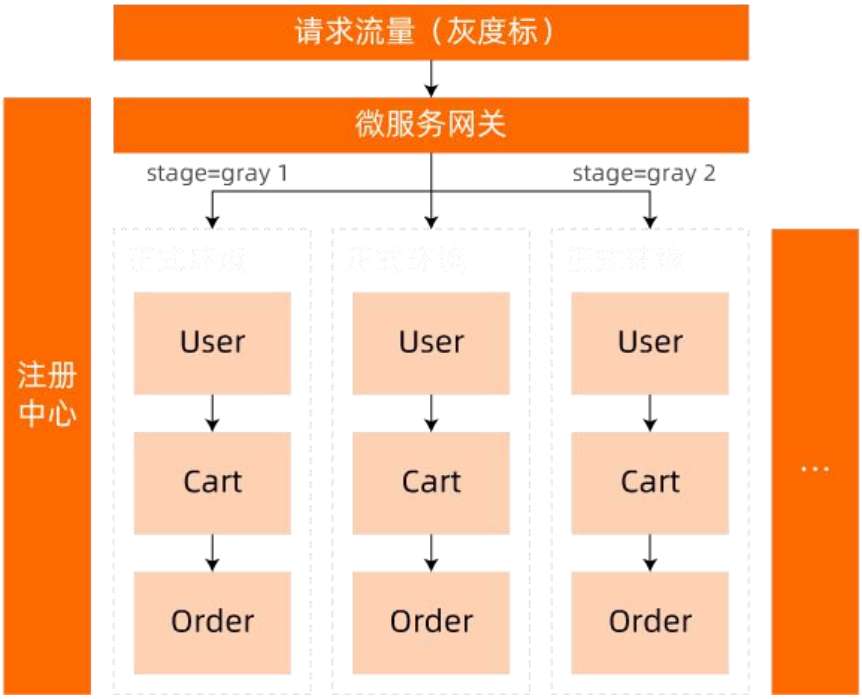
对于以上的问题，开发者结合实际业务场景和生产实践经验，提出了**一种端到端的灰度发布方案，即全链路灰度**。全链路灰度治理策略主要专注于整个调用链，它不关心链路上经过具体哪些微服务，流量控制视角从服务转移至请求链路上，仅需要少量的治理规则即可构建出从网关到整个后端服务的多个流量隔离环境，有效保证了多个亲密关系的服务顺利安全发布以及服务多版本并行开发，进一步促进业务的快速发展。

全链路灰度的解决方案

如何在实际业务场景中去快速落地全链路灰度呢？目前，主要有两种解决思路，基于物理环境隔离和基于逻辑环境隔离。

物理环境隔离

物理环境隔离，顾名思义，通过增加机器的方式来搭建真正意义上的流量隔离。

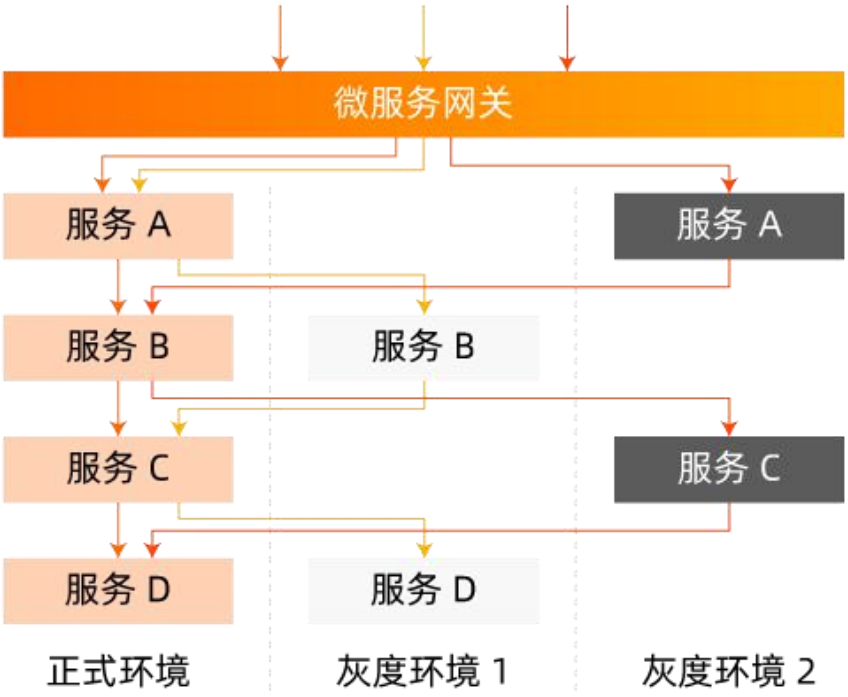


这种方案需要为要灰度的服务搭建一套网络隔离、资源独立的环境，在其中部署服务的灰度版本。由于与正式环境隔离，正式环境中的其他服务无法访问到需要灰度的服务，所以需要在灰度环境中冗余部署这些线上服务，以便整个调用链路正常进行流量转发。此外，注册中心等一些其他依赖的中间件组件也 需要冗余部署在灰度环境中，保证微服务之间的可见性问题，确保获取的节点 IP 地址只属于当前的网络环境。

这个方案一般用于企业的测试、预发开发环境的搭建，对于线上灰度发布引流的场景来说其灵活性不够。况且，微服务多版本的存在在微服务架构中是家常便饭，需要为这些业务场景采用堆机器的方式来 维护多套灰度环境。如果您的应用数目过多的情况下，会造成运维、机器成本过大，成本和代价远超收益；如果应用数目很小，就两三个应用，这个方式还是很方便的，可以接受的。

逻辑环境隔离

另一种方案是构建逻辑上的环境隔离，我们只需部署服务的灰度版本，流量在调用链路上流转时，由流经的网关、各个中间件以及各个微服务来识别灰度流量，并动态转发至对应服务的灰度版本。如下图：



上图可以很好展示这种方案的效果，我们用不同的颜色来表示不同版本的灰度流量，可以看出无论是微服务网关还是微服务本身都需要识别流量，根据治理规则做出动态决策。当服务版本发生变化时，这个调用链路的转发也会实时改变。相比于利用机器搭建的灰度环境，这种方案不仅可以节省大量的机器成本和运维人力，而且可以帮助开发者实时快速的对线上流量进行精细化的全链路控制。

那么全链路灰度具体是如何实现呢？通过上面的讨论，我们需要解决以下问题：

1.链路上各个组件和服务能够根据请求流量特征进行动态路由。
2.需要对服务下的所有节点进行分组，能够区分版本。
3.需要对流量进行灰度标识、版本标识。
4.需要识别出不同版本的灰度流量。

接下来，会介绍解决上述问题需要用到的技术。

标签路由

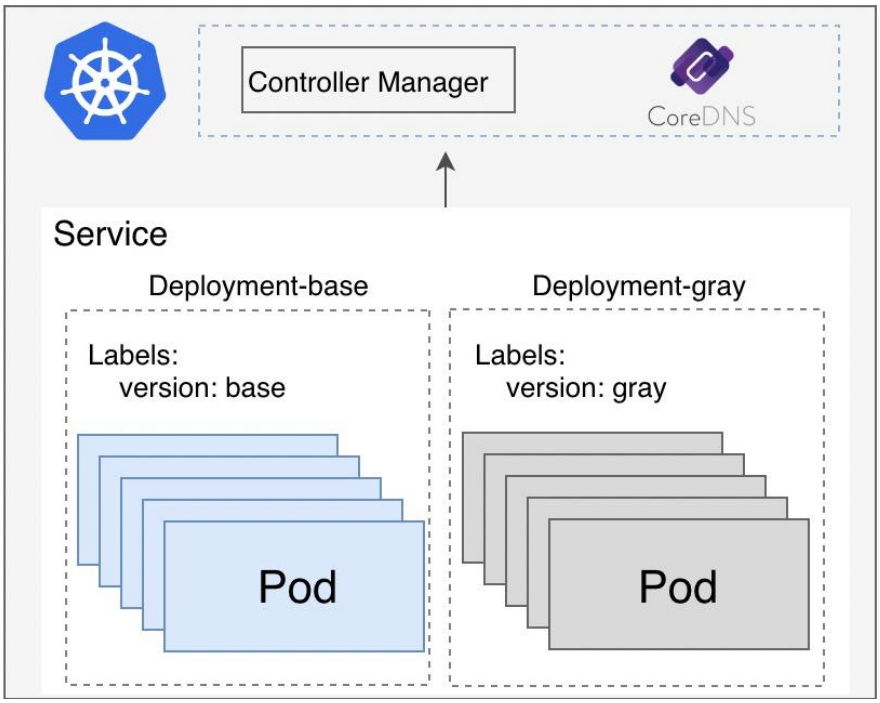
标签路由通过对服务下所有节点按照标签名和标签值不同进行分组，使得订阅该服务节点信息的服务消费端可以按需访问该服务的某个分组，即所有节点的一个子集。服务消费端可以使用服务提供者节点上的任何标签信息，根据所选标签的实际含义，消费端可以将标签路由应用到更多的业务场景中。



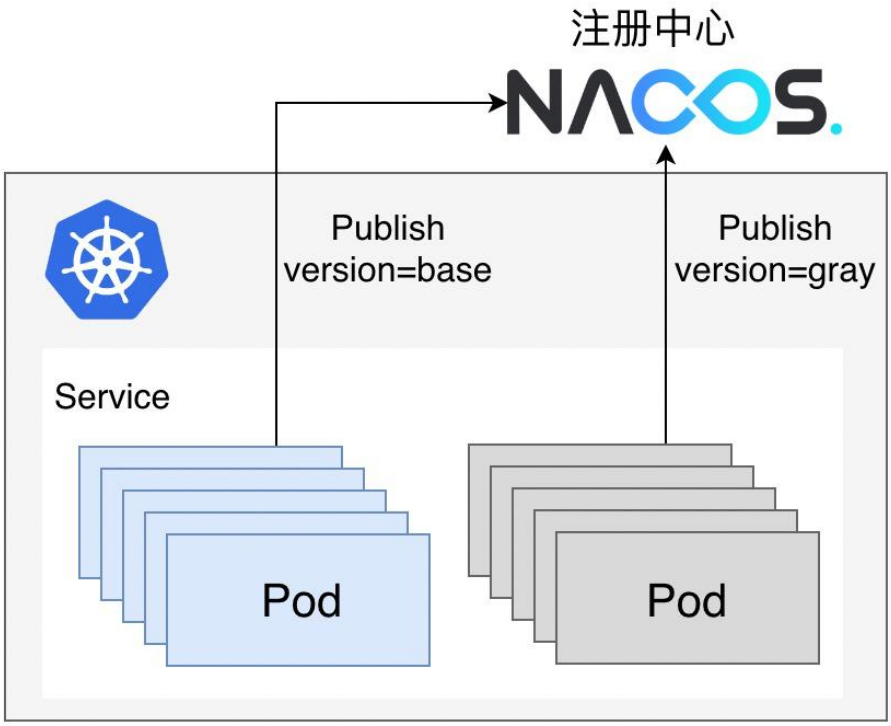
节点打标

那么如何给服务节点添加不同的标签呢？在如今火热的云原生技术推动下，大多数业务都在积极进行容器化改造之旅。这里，我就以容器化的应用为例，介绍在使用Kubernetes Service 作为服务发现和使用比较流行的 Nacos 注册中心这两种场景下如何对服务 Workload 进行节点打标。

在使用Kubernetes Service 作为服务发现的业务系统中，服务提供者通过向 ApiServer 提交 Service 资源完成服务暴露，服务消费端监听与该 Service 资源下关联的 Endpoint 资源，从 Endpoint 资源中获取关联的业务 Pod 资源，读取上面的 Labels 数据并作为该节点的元数据信息。所以，我们只要在业务应用描述资源 Deployment 中的 Pod 模板中为节点添加标签即可。



在使用Nacos 作为服务发现的业务系统中，一般是需要业务根据其使用的微服务框架来决定打标方式。 如果 Java 应用使用的 Spring Cloud 微服务开发框架，我们可以为业务容器添加对应的环境变量来完成标签的添加操作。比如我们希望为节点添加版本灰度标，那么为业务容器添加`spring.cloud.nacos.discovery.metadata.version=gray`，这样框架向 Nacos 注册该节点时会为其添加一个标签`verison=gray`。



流量染色

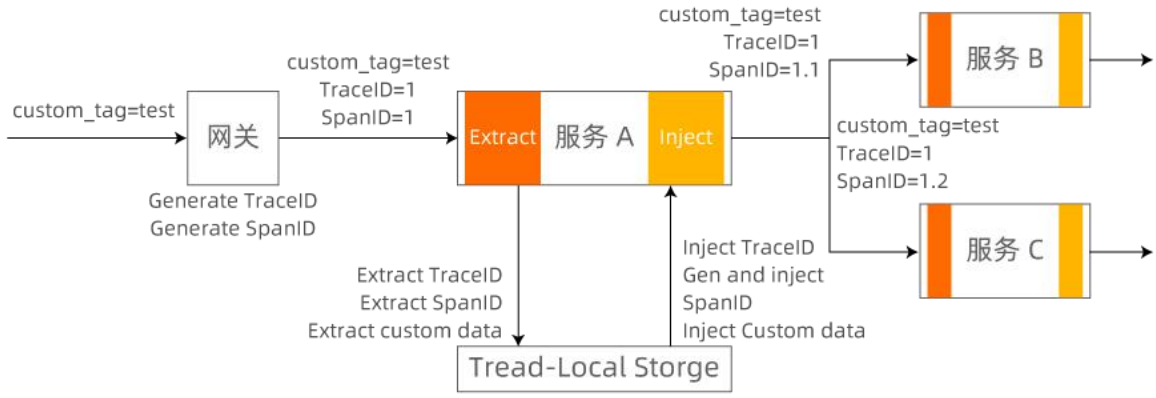
请求链路上各个组件如何识别出不同的灰度流量？答案就是流量染色，为请求流量添加不同灰度标识来方便区分。我们可以在请求的源头上对流量进行染色，前端在发起请求时根据用户信息或者平台信息的不同对流量进行打标。如果前端无法做到，我们也可以在微服务网关上对匹配特定路由规则的请求动态 添加流量标识。此外，流量在链路中流经灰度节点时，如果请求信息中不含有灰度标识，需要自动为其染色，接下来流量就可以在后续的流转过程中优先访问服务的灰度版本。

分布式链路追踪

还有一个很重要的问题是如何保证灰度标识能够在链路中一直传递下去呢？如果在请求源头染色，那么请求经过网关时，网关作为代理会将请求原封不动的转发给入口服务，除非开发者在网关的路由策略中实施请求内容修改策略。接着，请求流量会从入口服务开始调用下一个微服务，会根据业务代码逻辑形成新的调用请求，那么我们如何将灰度标识添加到这个新的调用请求，从而可以在链路中传递下去呢？

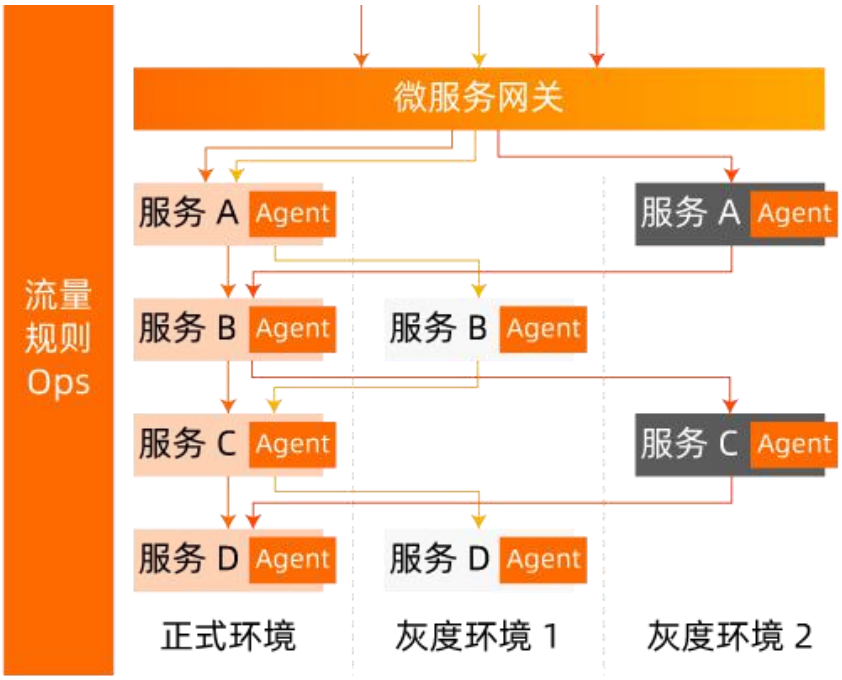
从单体架构演进到分布式微服务架构，服务之间调用从同一个线程中方法调用变为从本地进程的服务调用远端进程中服务，并且远端服务可能以多副本形式部署，以至于一条请求流经的节点是不可预知的、不确定的，而且其中每一跳的调用都有可能因为网络故障或服务故障而出错。分布式链路追踪技术对大型分布式系统中请求调用链路进行详细记录，核心思想就是通过一个全局唯一的 traceid 和每一条的 spanid 来记录请求链路所经过的节点以及请求耗时，其中 traceid 是需要整个链路传递的。

借助于分布式链路追踪思想，我们也可以传递一些自定义信息，比如灰度标识。业界常见的分布式链路追踪产品都支持链路传递用户自定义的数据，其数据处理流程如下图所示：



逻辑环境隔离

首先，需要支持动态路由功能，对于 Spring Cloud、Dubbo 开发框架，可以对出口流量实现自定义 Filter，在该 Filter 中完成流量识别以及标签路由。同时需要借助分布式链路追踪技术完成流量标识链路传递以及流量自动染色。此外，需要引入一个中心化的流量治理平台，方便各个业务线的开发者定义自己的全链路灰度规则。如下图所示：



实现全链路灰度的能力，无论是成本还是技术复杂度都是比较高的，以及后期的维护、扩展都是非常大的成本。

当然您可以选择阿里云 MSE 服务治理产品，该产品就是一款基于 Java Agent 实现的无侵入式企业生产级服务治理产品，您不需要修改任何一行业务代码，即可拥有不限于全链路灰度的治理能力，并且支持近 5 年内所有的 Spring Boot、Spring Cloud 和 Dubbo。

MSE 微服务治理全链路灰度

全链路灰度作为 MSE 服务治理专业版中的拳头功能，具备以下六大特点：

- 可通过定制规则引入精细化流量

除了简单地按照比例进行流量引入外，我们还支持 Spring Cloud 与 Dubbo 流量按规则引入，Spring Cloud 流量可根据请求的 cookie、header、param 参数或随机百分比引入流量，Dubbo 流量可按照服务、方法、参数来引入。





- 全链路隔离流量泳道

1)通过设置流量规则对所需流量进行'染色'，'染色'流量会路由到灰度机器。
2)灰度流量携带灰度标往下游传递，形成灰度专属环境流量泳道，无灰度环境应用会默认选择未打标的基线环境。

- 端到端的稳定基线环境

未打标的应用属于基线稳定版本的应用，即稳定的线上环境。当我们将发布对应的灰度版本代码，然后可以配置规则定向引入特定的线上流量，控制灰度代码的风险。

- 流量一键动态切流

流量规则定制后，可根据需求进行一键停启，增删改查，实时生效。灰度引流更便捷。

- 低成本接入，基于 Java Agent 技术实现无需修改一行业务代码

MSE 微服务治理能力基于 Java Agent 字节码增强的技术实现，无缝支持市面上近 5 年的所有 Spring Cloud 和 Dubbo 的版本，用户不用改一行代码就可以使用，不需要改变业务的现有架构，随时可上可下，没有绑定。只需开启 MSE 微服务治理专业版，在线配置，实时生效。

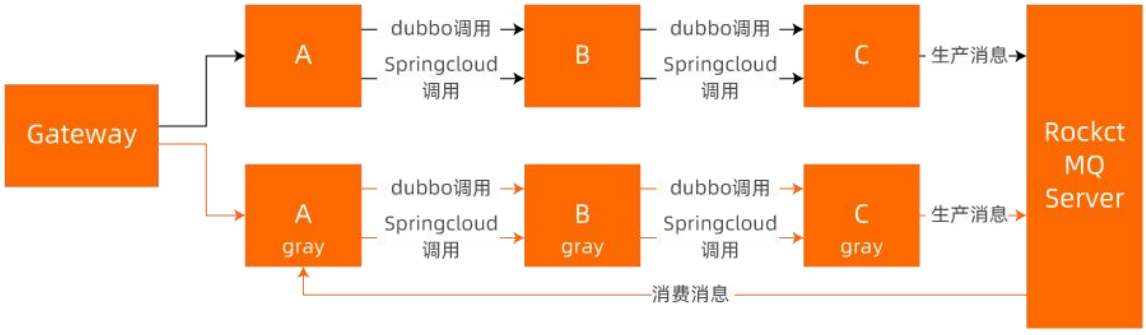
- 具备无损上下线能力，使得发布更加丝滑

应用开启 MSE 微服务治理后就具备无损上下线能力，大流量下的发布、回滚、扩容、缩容等场景，均能保证流量无损。

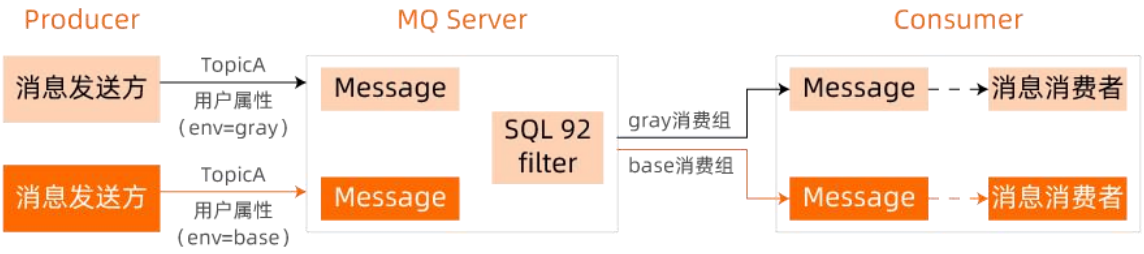
RPC 流量的全链路灰度方案

基于消息队列 RocketMQ 的全链路灰度方案

详细可以了解 MSE 服务治理的帮助文档《基于消息队列 RocketMQ 版实现全链路灰度》



技术方案如下：



·支持 RocketMO 版本：4.2.0+
·支持 Pull 和 Push 两种模式
·需要在服务端配置 enablePropertyFilter=true 并重启服务端

尾

MSE 服务治理以无侵入的方式提供了全链路灰度、离群实例摘除、金丝雀发布、微服务治理流量可观测等核心能力，以更经济的方式、更高效的路径帮助我们的系统在云上快速构建起完整微服务治理体系。

3.3 微服务可观测增强解决方案

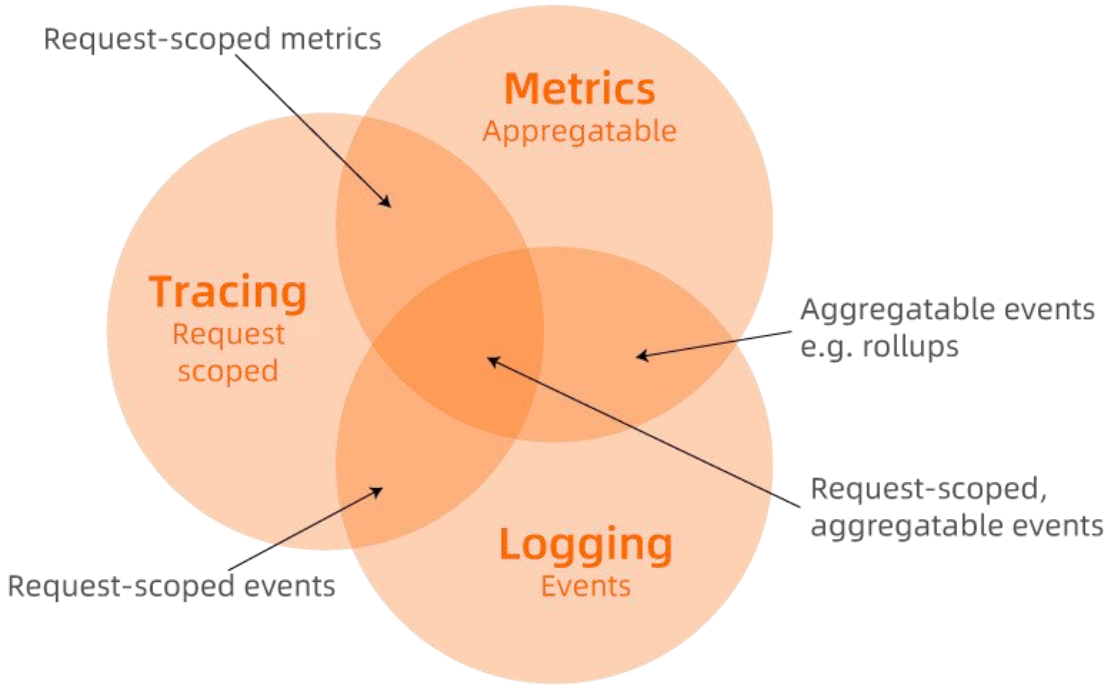
背景

彼得·德鲁克曾经说过：“如果你无法量化它，你就无法管理它。” 可观测性（Observability）是帮助微服务稳健运行的重要一环。“我们的系统是否还是正常的？”，“终端用户的体验是否符合预期？”，“我们如何在系统快要出问题之前主动发现系统的风险？”。如果说监控可以告诉我们系统出问题了，那么可观测就可以告诉我们系统哪里出问题了，什么原因导致的问题。可观测不但可以判断系统是否正常，还可以在系统出现问题之前，主动发现系统风险。



从系统的角度来讲，监控以 Ops 为主，聚焦在发现，确保系统稳定性。可观测性的目标是白盒化，注重 Recall+Precision，贯穿 Dev/Tester/Ops 等环节，通过多种观测手段，确保找到根因，防患于未然。

可观测性支撑数据大致分为三大类：Metrics，Tracing 和 Logging。以下是社区中常用的一张图：



- **Metrics:** 是一种聚合的度量数值，提供量化的系统内/外部各个维度的指标，一般包括 Counter、Gauge、Timer、Histogram 等度量指标。

- **Tracing:** 提供了一个请求从接收到处理完毕整个生命周期的跟踪路径，面向的是请求，可以轻松分析出请求中异常点，通常请求都是在分布式的系统中处理。比如一次请求的范围，也就是从浏览器或者手机端发起的任何一次调用，我们根据轨迹去追踪，经过哪些组件、微服务等，所以也叫做分布式链路追踪。

- **Logging:** 应用运行过程中产生的事件或者程序执行过程中产生的一些日志，可以给我们提供一些系统运行过程中的精细化信息，例如某个关键变量、事件、访问记录等。

我们可以将指标、追踪和日志数据的进行进一步关联、分析，推导出当前微服务系统所处的状况。

云原生下微服务应用可观测的挑战

目前，常见的微服务框架包括 Spring Cloud 和 Dubbo 等多语言微服务，并具备服务注册发现、服务配置、负载均衡、API 网关、分布式微服务等基本能力。其中，服务治理包括无损下线，服务容错，服务路由等能力。可观测性包括应用监控，链路追踪，日志管理，应用诊断等。



随着云原生到来，微服务架构得到越来越多的应用。由最初以机器为核心的云服务器 ECS 上云，到以容器为核心的容器化云原生部署；为了更加敏捷，阿里云开始以应用为核心的微服务化。如今，当微服务发展到一定应用规模，阿里云开始围绕业务核心，以提效稳定为目的的服务治理。



在云原生下的微服务可观测主要面临三个挑战：

- 发现难

从云服务器 ECS 到 Kubernetes，微服务架构复杂度提升，观测对象复杂度提升，监测数据覆盖不全。

- 定位难

随着多种治理能力深入，可观测要求高，服务框架复杂度增加，技术门槛提升，数据本身复杂度提升，数据关联性差。

- 协作差

随着组织角色变化，可观测不只是运维工作。



应用实时监控服务 ARMS 作为阿里云可观测产品，支持自动检测部分产品问题。目前已经覆盖五十多个故障场景，包括应用变更、大请求、QPS 突增等，诊断报告认可率高达 80%。

ARMS能自动检测的部分问题场景

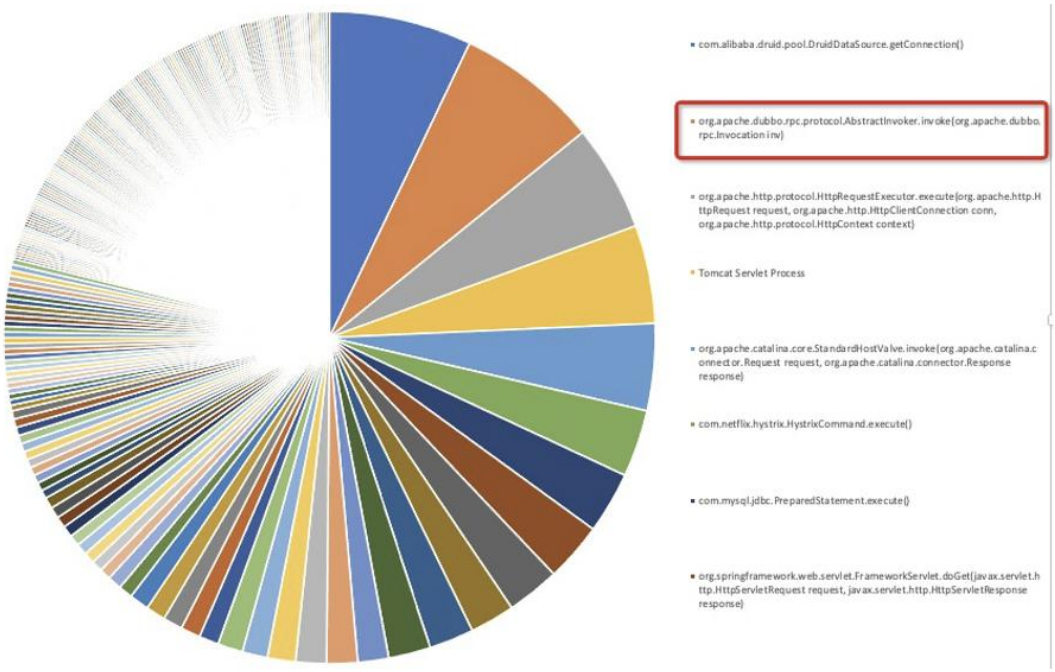
阿里云

覆盖50+故障场景，目前诊断报告认可率高达80%.

ARMS故障诊断支持场景列表

诊断大类	诊断小类	诊断结论(summary)
应用变更	应用发布	应用事件{事件内容}变更导致
	配置变更	应用变更时间{事件内容}导致
大请求	单个大请求	出现大请求
qps突增	第三方导致qps突增	特定上游请求量突增
下游问题	下游问题	下游在{时间}出现故障，异常值{值}
宿主机	宿主机问题(K8s下定位到具体Node)	应用{应用名}的主机{ip}的宿主机{ip}在{事件}负载高
单机问题	fullgc	应用{应用名}主机{ip}出现fullgc耗时高
	线程池满	应用{应用名}主机{ip}hsf 线程池满
	OOM	应用{应用名}主机{ip}出现OOM
	慢SQL	应用{应用名}主机{ip}出现慢SQL，慢SQL内容
	线程Block	应用{应用名}主机{ip}出现线程Block，线程堆栈
	网络重传	应用{应用名}主机{ip}出现网络重传
	单机磁盘使用率高	应用{应用名}主机{ip}磁盘利用率高
中间件问题	redis指标异常	应用{应用名}中间件{中间件名}指标{指标名}在{时间}突增至{值}
	mysql指标异常	应用{应用名}中间件{中间件名}指标{指标名}在{时间}突增至{值}
	MQ指标异常	应用{应用名}中间件{中间件名}指标{指标名}在{时间}突增至{值}
单服务问题	单服务问题	服务{服务名}的响应时间在{时间}出现突增，异常值{值}
业务代码问题	业务方法	服务{服务名}的{方法名}的执行时间在{时间}出现突增，异常值{值}
	N+1问题	出现对于下游依赖服务（具体SQL/具体接口）的大量重复调用
	日志打印问题	服务{服务名}的打印日志方法{方法名}的执行时间在{时间}出现突增，异常值{值}
JVM参数	JVM参数	应用{应用名}配置的JVM参数不合理
异常	新增异常突增	应用{应用名}在{时间}内新增异常突增
...	...	...

如下图所示，线上 7%的应用都在 Dubbo 的 RPC 上耗时，并由于埋点问题，无法定位出根因。



阿里云在为客户服务过程中，发现了很多问题。

- 服务发现

当前一些监测工具无法实现服务框架服务发现层面的问题诊断，导致遗留了许多服务调用问题难以排查，单看监控使得客户根本无从下手。因此，我们希望通过提供以下方面服务发现监控诊断能力，帮助客户及时排查服务发现领域出现问题导致的应用运行异常。

1、监控客户端出现 no provider 问题；

2、微服务应用连接的是哪个注册中心，服务发现链路调用示例图，大块内容有 Provider、Consumer、注册中心，点击对应组件可以看到详细相关地址；

3、应用服务是否注册成功；

4、应用最近一次拉下来的地址数量 & 内容；

5、应用与注册中心的心跳是否健康；

6、注册中心状态信息，如 CPU、内存等运行硬件状态信息，注册服务数目、订阅服务数以及服务内容等信息。

- 微服务生命周期

微服务启动慢，一个服务器花 3 分钟，5 个服务器花 30 分钟。

- 调用链路

Consumer 调用超时 Provider 却快速返回。



除此之外，还有微服务配置混乱，不好梳理；微服务应用上 Kubernets 之后，出现线程池满，却找不到原因等一系列问题。

那么，当站在微服务视角思考如何进行体系建设时，我们提出的微服务可观测性增强解决方案。站在传统监测方案之上还能再做哪些事情？

微服务可观测增强包含哪些方面

服务发现

当前的一些监控工具无法实现服务框架服务发现层面的问题诊断，因此遗留许多服务调用的问题难以排查，单看监控会客户根本无从下手的窘境。因此我们希望通过提供以下方面服务发现监控诊断能力帮助客户及时排查服务发现领域出现问题导致的应用运行异常：

1、监控客户端出现 no provider 问题。

2、微服务应用连接的是哪个注册中心，服务发现链路调用示例图，大块内容有 Provider、Consumer、注册中心，点击对应组件可以看到详细相关地址。

3、应用服务是否注册成功。

4、应用最近一次拉下来的地址数量&内容。

5、应用与注册中心的心跳是否健康。

6、注册中心状态信息（CPU、内存等运行硬件状态信息，注册服务数目、订阅服务数以及服务内容等信息）。

调用链路

当前市面上的开源以及竞品的探针在 RPC 层面仅仅埋点在客户端跟服务端的服务调用上，埋点过于简单与粗粒度，无法实现服务框架层面的问题诊断，因此遗留许多服务调用的问题难以排查，单看监控客户根本无从下手。因此我们希望通过增加以下几个维度的监控帮助用户及时了解调用链路状态信息：

1、Consumer 调用耗时很大 ，Provider 却快速返回（问题排查）

2、上了新的治理功能后，应用性能影响有多大（性能诊断）

3、某条请求调用非常耗时，但是服务端逻辑缺很简单，到底问题出在哪（问题排查）

4、业务传入参数过大/有问题导致序列化/反序列化耗时长（问题排查），如何选择合适的序列化框架？（性能诊断）提供请求超时时间超过 10s 提醒、响应体超过 2M 报警（事件告警）

5、中间件业务逻辑问题，如何一眼看出接入的中间件存在问题？
a、增加 AHAS 后请求 rt 增加
b、使用标签路由能力后性能下降
c 、使用自研路由能力后性能下降（ 性能诊断 ）

6、网络异常如何一眼看出来而不是跟服务框架混在一起？（问题排查）

7、微服务连接层面的监控，参考 hsf 的 remoting.log，netty 的事件/网络心跳/活跃连接数等

8、业务参数的监控

9、超时/截止时间监控与告警

微服务生命周期

我们希望应用启动过程中，从 Spring bean 加载、链接池连接的监测、微服务的服务注册、Kubernetes 的监测检查就绪；应用下线过程中，服务注册、在途请求的停止、定时任务/MQ 等取消、服务停机；例如：Spring bean 初始化异常，卡在哪个 bean 的加载上，哪个 bean 初始化耗时特别长。帮助用户分析启动慢的原因，自动给出修复建议。然而，目前整体过程是缺少相关观测能力。

微服务治理相关

无损上下线、主动下线、金丝雀发布、同 AZ、标签路由、服务鉴权、离群摘除增加观测性、容量评估、健康度打分等。

微服务场景下可观测的探索与实践

微服务可观测增强解决了什么问题?一句话概括就是：全面增强微服务场景下的可观测能力。

让一线运维人员具备微服务诊断基本能力，可以排查 80% 的微服务常见问题，快速进行性能分析诊断。

ARMS 微服务可观测增强方案回答了以下问题：

- 为什么服务启动很慢

从 pod 创建到应用初始化再到服务注册应用启动，端到端分析出应用启动慢的根因，补齐应用启动生命周期的可观测能力。

- 依赖是否存在隐患

为 SpringCloud/Dubbo 依赖的 Jar 包进行分析，定位是否存在 Jar 包依赖冲突等问题。

- 配置分析

微服务场景下配置分散且冗余，提供应用运行时配置可观测能力以及配置优化的专家经验。

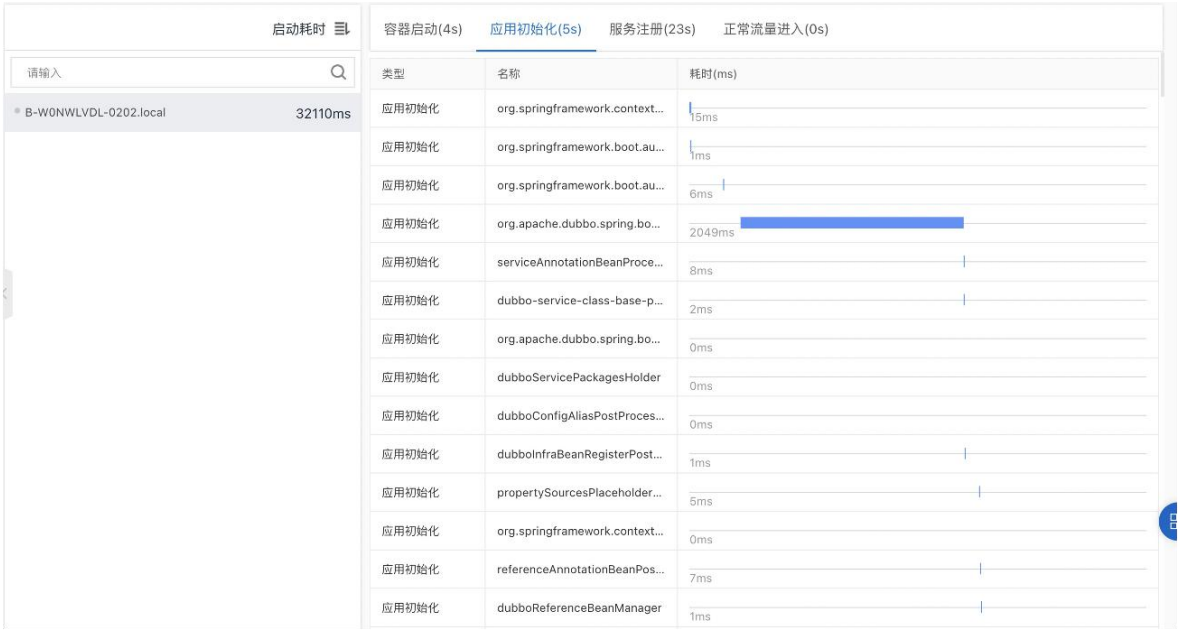
- Dubbo 调用链增强

覆盖寻址，序列化，网络等阶段的埋点，一眼看出 Dubbo 调用的时间都去哪儿了。

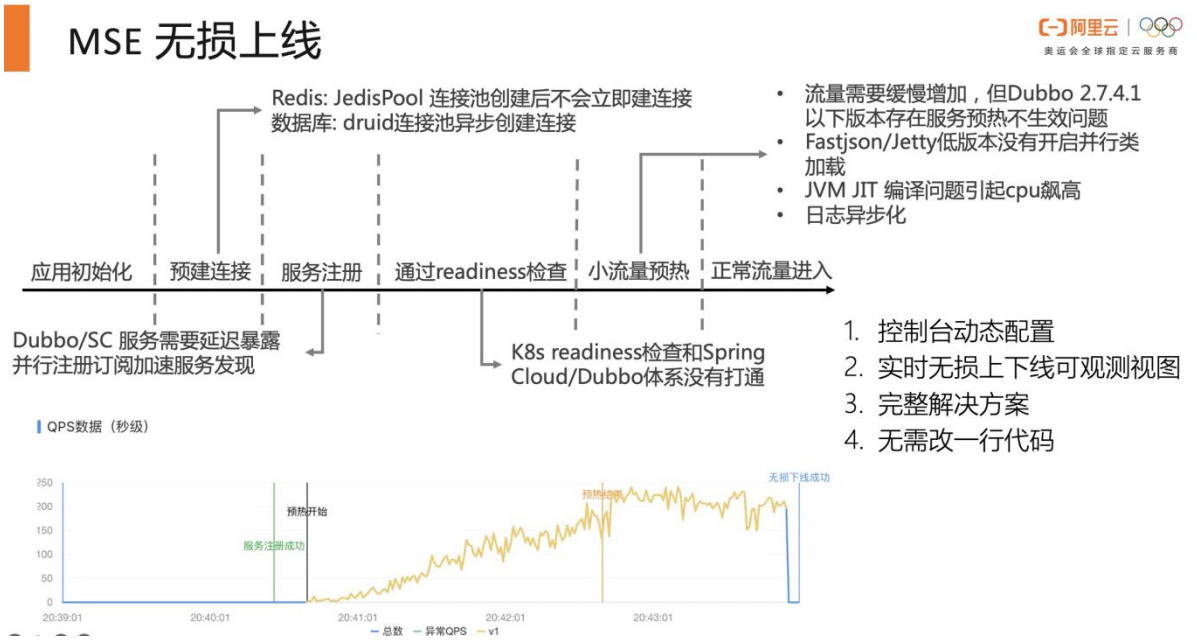
为什么服务启动慢？通过从 Pod 创建到应用初始化再到服务注册应用启动，端到端分析出应用启动慢的根因，补齐应用启动生命周期的可观测能力。



通过将整个流程串联，实时观察到每个点的耗时，可观测视图把问题剖析出来。上图是 ARMS 容器启动分析功能。左边是服务启动，系统将启动过程中的每一块时间拆分出来，从而清晰看到微服务启动慢在了哪一步，增强了其可观测性。



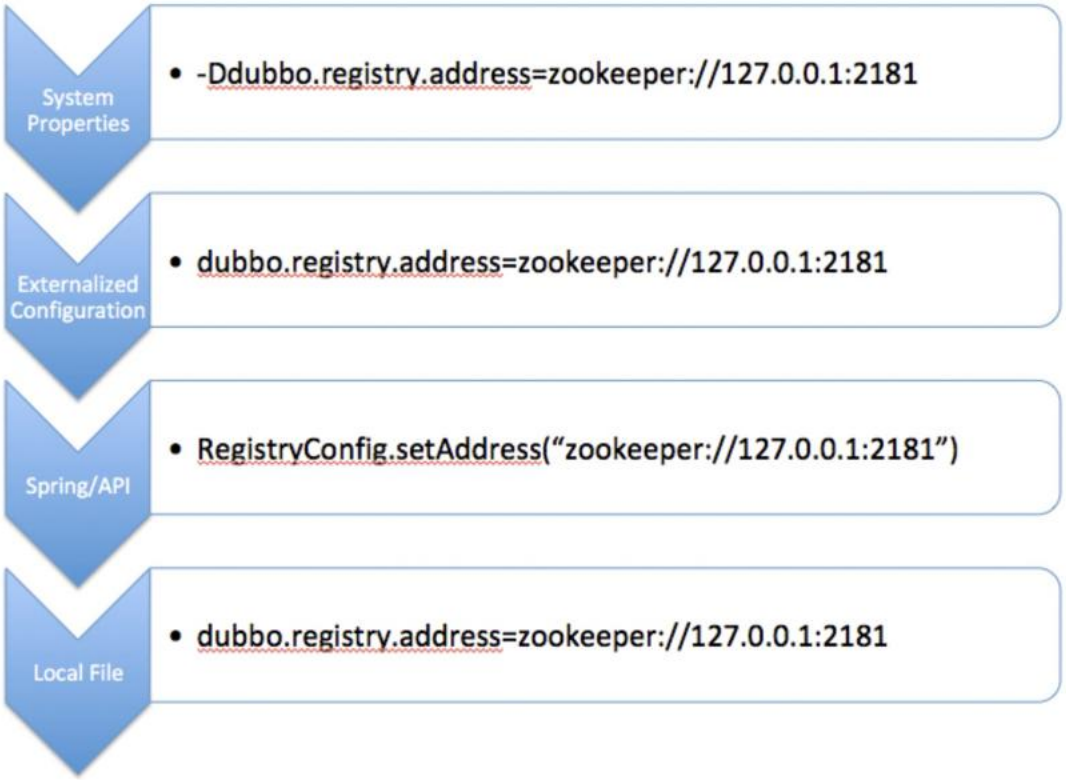
微服务引擎提供了无损上线的能力。控制台动态配置，实时无损上下线可观测视图，完整解决方案无需改一行代码。在微服务启动的全流程进行各种方案的保护与治理：预建连接阶段通过提前异步创建连接保证不会阻塞在连接建立的过程中；服务注册发现阶段通过并行注册与订阅能力进一步提升应用的启动速度；在小流量预热阶段通过调整客户端的负载均衡能力保证新起的实例中流量缓慢增长。



由于微服务配置的覆盖关系较为复杂，需要进行配置分析。

覆盖关系

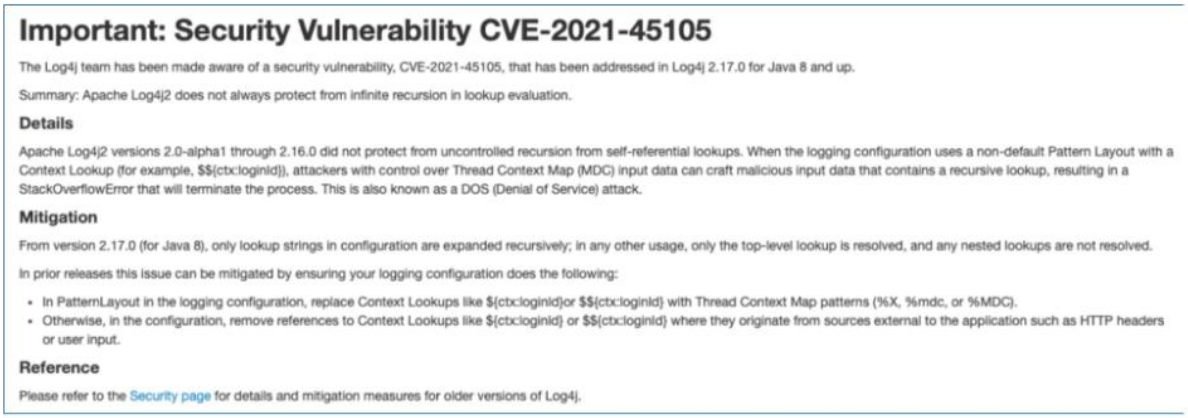
下图展示了配置覆盖关系的优先级，从上到下优先级依次降低：



请参考相关内容：属性覆盖。

上图为 Dubbo 官方提供的配置覆盖关系，可以看到其具有一定顺序先后性。很多时候很难判断配置是否配错地方，是否生效，是否被覆盖。微服务场景下配置分散且冗余，我们提供应用运行时配置可观测能力以及配置优化的专家经验。

我们提供了为 SpringCloud/Dubbo 依赖的 Jar 包进行分析的能力，帮助定位是否存在 Jar 包依赖冲突、依赖的 Jar 是否存在安全、性能风险等问题。



Important: Security Vulnerability CVE-2021-45105

The Log4j team has been made aware of a security vulnerability, CVE-2021-45105, that has been addressed in Log4j 2.17.0 for Java 8 and up.

Summary: Apache Log4j2 does not always protect from infinite recursion in lookup evaluation.

Details

Apache Log4j2 versions 2.0-alpha1 through 2.16.0 did not protect from uncontrolled recursion from self-referential lookups. When the logging configuration uses a non-default Pattern Layout with a Context Lookup (for example, $${ctx:loginId}), attackers with control over Thread Context Map (MDC) input data can craft malicious input data that contains a recursive lookup, resulting in a StackOverflowError that will terminate the process. This is also known as a DOS (Denial of Service) attack.

Mitigation

From version 2.17.0 (for Java 8), only lookup strings in configuration are expanded recursively; in any other usage, only the top-level lookup is resolved, and any nested lookups are not resolved.

In prior releases this issue can be mitigated by ensuring your logging configuration does the following:

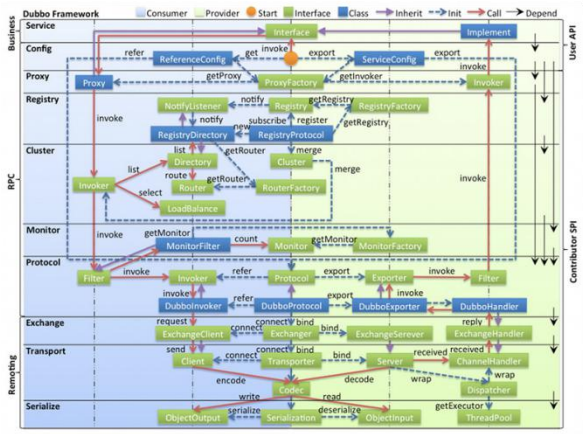
- In PatternLayout in the logging configuration, replace Context Lookups like ${ctx:loginId}or $${ctx:loginId} with Thread Context Map patterns (%X, %mdc, or %MDC).
- Otherwise, in the configuration, remove references to Context Lookups like ${ctx:loginId} or $${ctx:loginId} where they originate from sources external to the application such as HTTP headers or user input.

Reference

Please refer to the Security page for details and mitigation measures for older versions of Log4j.

一次 RPC 调用时间到底去哪儿了？一次 RPC 调用有路由、限流降级、序列化、网络等各个环节。从客户端来讲，需要经过路由、filter、invoker、serialize、remote。从服务端来讲，需要经过 serialize、Proxy Invoke、filter、impleme。





客户端：
路由->filter->invoker->serialize->remote

服务端
serialize -> Proxy Invoke -> filter -> implement

路由、限流降级、序列化、网络...

上图是一次 RPC 调用流程图。其中包括寻址、负载均衡的建立连接时间，打包的序列化时间，解包的返印值反序列化时间，服务端处理时间和等待服务端处理返回的时间。



如上则是我们给出的答案，将调用链在 RPC 框架内进一步细分，一眼看出路由、序列化、网络、代理、服务端处理等耗时的细节。

总结

微服务可观测增强方案站在传统的可观测性方案之上我们进一步从微服务的视角出发，扩展传统可观测覆盖的 Tracing、Logging、Metrics 等数据，结合微服务专家的诊断经验。

从前端、应用至底层机器，应用实时监控服务 ARMS 实时监测应用服务的每次运行、每个慢 SQL、每个异常。与此同时，提供完整数据大盘监控，展示请求量、响应时间、FullGC 次数、慢 SQL 和异常次数、应用间调用次数与耗时等重要的关键指标，时刻了解应用程序的运行状况，确保向用户提供最优使用体验。



3.4 微服务应用配置解决方案

背景介绍

微服务架构之应用配置

应用配置可解决的问题主要包括以下方面：

问题项	问题描述
本地静态配置	采用本地静态配置，导致运行时无法动态修改
配置格式不统一	散乱，难以管理，有的用XML 格式，有的用properties，有的用DB 等
生产事故	容易将非生产配置带到生产环境，引发事故
配置修改困难	部署多台节点时，修改配置费时费力，周期长
缺少安全审计和版本控制能力	无法追溯责任人，无法得知修改内容，无法确定修改时间，无法及时回滚

针对上述问题，应用配置中心应运而生，但市场上存在的配置中心产品侧重点及适用场景各不相同，下面对现有产品进行对比分析。

典型使用场景

动态调整日志级别

此为开箱即用功能。

在特定的场景下，您可以针对性地动态调整日志级别，以便输出更多的日志信息排查线上问题，或是减少日志打印带来的性能消耗。功能开关提供了在应用运行时动态修改日志级别的功能，在不同的应用场景下，您可以随时调整日志的级别，得到更有效的日志信息。

在开发 Java 程序时，我们经常会用到各种各样的日志框架。为了避免在程序正常运行时输出不必要的信息，我们在使用日志框架时会设置默认的日志级别。而程序在线上运行时，我们需要在特定的场景下针对性地动态调整日志级别。

进入目标应用的**开关列表**页面。在**开关列表**页面搜索到 **SYSTEM_LOG_CONFIG** 开关，即日志级别开关。



单击**操作**列的**全局推送**或**单机推送**，按照 **<loggerName,loggerLevel>** 格式填写日志运行的配置，然后单击全局推送或单机推送。即可修改全部机器或是单台机器的日志运行级别。

推送值格式：**Key** 为 **LoggerName**，**Value** 为日志级别。如需修改全局日志级别，**LoggerName** 为 **root**，如下所示。

```
{
 "root": "ERROR"
}
```

注：支持的日志框架：Log4j、Log4j2、Logback。

配置项组合更新

可按不同场景批量更新组合配置项，所谓组合配置项指具有一组相互关联业务语义的配置项，如页面公告中时间、标题、内容等，商品特殊优惠配置中价格、优惠折扣等。

下图以'商品优惠配置'为例进行说明。

'商品优惠配置'在不同场景下优惠对象、优惠折扣及价格等各不相同，将'商品优惠配置'涉及的

配置项组合，在不同场景下设置不同内容，可在不同场景下快速切换，同时省去繁琐校验过程，避免出错。

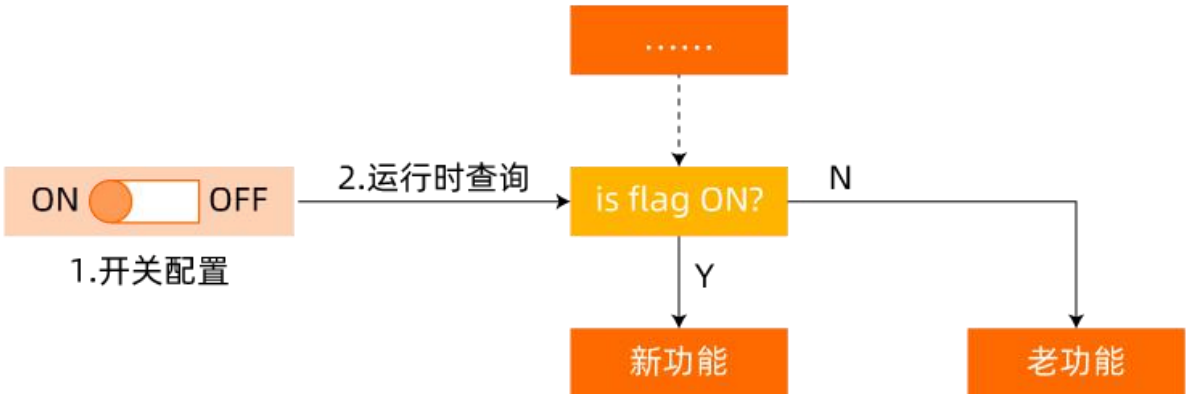


开关驱动开发

以开关方式控制代码执行逻辑，用于新功能快速验证，在出现问题时可及时回退。相比复杂的系统发布，投入成本较低，可结合 DevOps 机制进行实践。

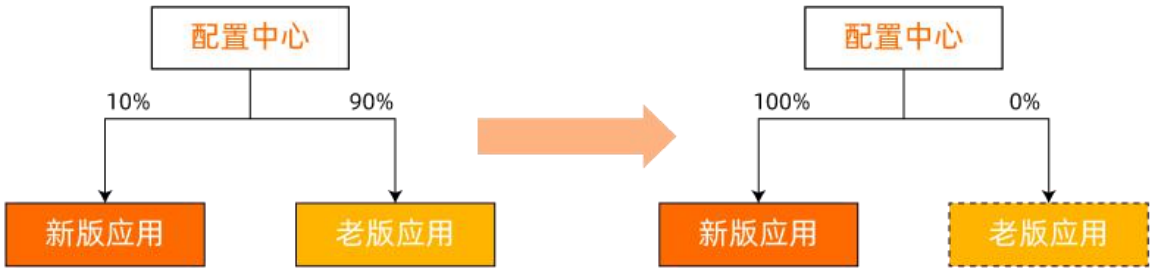
如下图所示，当执行逻辑触发时访问对应的开关配置查看配置是否打开，从而决定是否执行新功能。

可用于 A/B 测试、环境隔离等场景。



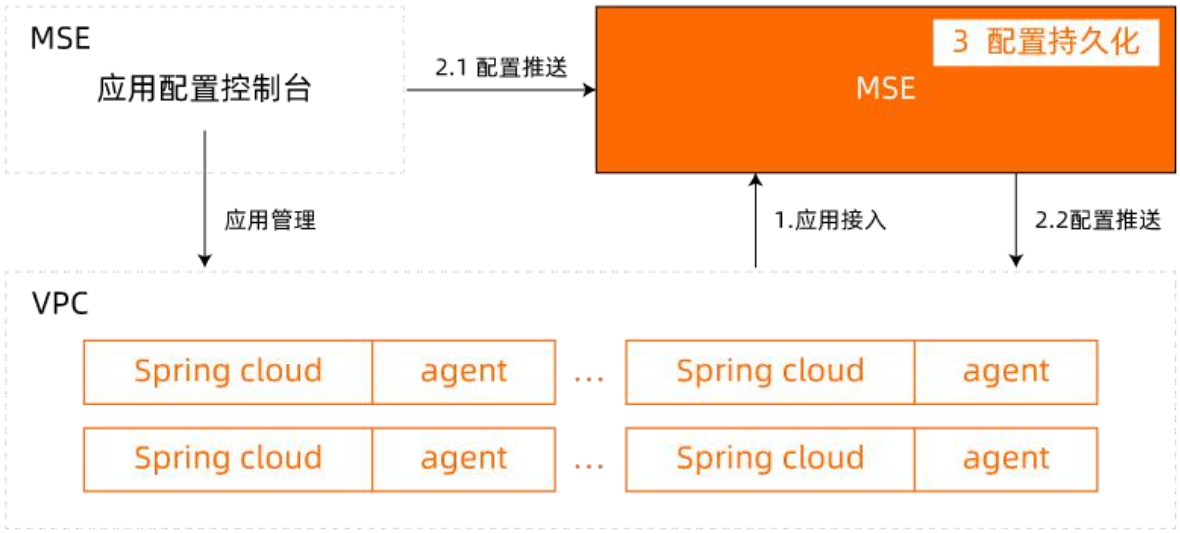
金丝雀(灰度)发布

通过配置项控制应用接收流量，在灰度验证后，可及时回退或全线切流。



应用配置解决方案介绍

用户在无需改变 MSE 使用习惯前提下,通过新应用接入或版本升级方式即可使用应用配置能力。MSE 后端与应用配置服务(ahas-switch)整合而复用应用配置相关功能,解耦应用配置具体实现。控制台推送配置项由应用配置服务处理，配置项通过 ACM 组件持久化，应用重启或扩容阶段可读取持久化配置。



下文首先通过对比现有产品，分析竞品优势与劣势，进而引出本应用配置解决方案核心设计要点，突出差异化优势，为用户提供优秀的应用配置解决方案。

现有产品对比分析

选取较为有代表性且具有一定市场规模的产品进行对比，分析产品的适用场景，为用户产品选型及 AHAS 功能开关应用配置推广提供依据。

1.开源组件：

- Togglz
- Switch 社区版本
- Apollo

2.商业化产品：

- AWS AppConfig: Configure, validate, and deploy application configuration

产品对比分析如下表所示：

产品	Switch 社区版	Togglz	AWS AppConfig	Apollo	AHAS 功能开关（MSE 应用配置）
配置时效性	动态配置，需自行实现可靠推送	动态配置，需自行实现可靠推送	动态配置，实时生效	动态配置，实时生效	动态配置，开箱即用，可靠推送，实时生效
配置覆盖能力	仅支持单机推送	需自行实现持久化	多节点覆盖	多节点覆盖	快速覆盖上千服务实例
配置灰度能力	不支持	支持	支持	支持	支持
简单配置（文本、开关）	支持	场景有限，仅可以当做 bool 类型的开关	支持	支持	支持
复杂类型的业务配置	支持	不支持	需用户手写规则校验，使用成本高	不支持	自动校验，保证类型安全
可观测能力	无控制台，不支持	支持弱	弱，业务生效的实时值无法直接观 测	支持	白屏化观测能力，控制台可观测各接入节点的实时生效值和分布情况

产品	Switch 社区版	Togglz	AWS AppConfig	Apollo	AHAS 功能开关（MSE 应用配置）
微服务生态支持	无直接支持	无开箱即用支持	无直接支持	支持 SpringCloud 微服务家族 配置项	开箱即用的 Spring Cloud @Value 及 @ConfigurationProperties 配置动态管理能力
开箱即用的运维管控能力	不支持	不支持	不支持	支持	支持，如动态日志级别调整
代码侵入性	有侵入	有侵入	有侵入	SDK 方式低侵入；Java Agent方式无侵入，但功能存在问题	Java Agent方式无侵入，SDK方式低侵入

综合比较可得出结论：在应用配置领域，AHAS 提供的应用配置支持的场景及功能完备性方面，相对而言，对用户使用更加友好。

核心设计要点

- 应用接入

应用通过 Agent 方式接入 MSE，连接应用配置服务，无需对应用做任何改造，真正做到无侵入。

- 配置推送

在 MSE 控制台即可对应用配置进行管理，按需推送配置项，支持按节点推送与全局推送方式。

- 配置持久化

应用配置服务通过 ACM 组件持久化配置项，保障配置项高可靠性。应用在重启或扩容阶段可读取持久化配置。

差异化优势

在产品对比分析基础上，对 AHAS 功能开关具备的差异化能力简要概括为以下三点：

- 强类型校验

用户无需在业务层面对接收到的配置进行类型及格式的校验，校验工作由平台承担，应用仅需关注业务。

- 无侵入式接入

对 SpringCloud 应用支持一键接入，自动识别应用中配置项，可通过控制台实时修改并进行持久化等操作。

- 复杂配置项支持

在复杂数据类型支持方面较为完善，无需遵守较为繁琐的配置项规则。

- 开箱即用功能

支持日志级别动态调整，获取不同级别日志，方便问题分析、故障定位。

3.5 微服务限流降级解决方案

背景介绍

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

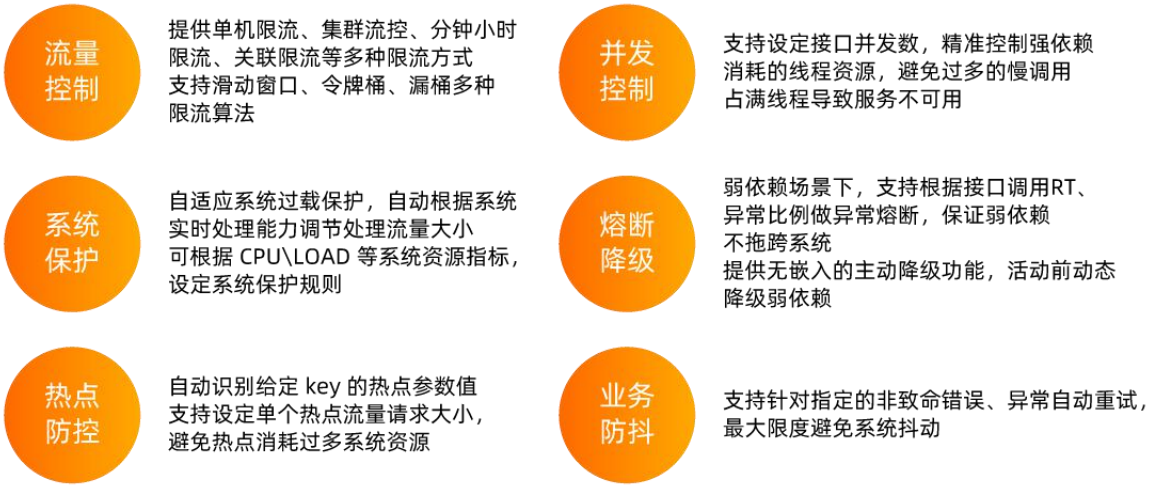
- 演唱会抢票瞬间洪峰流量导致系统超出最大负载，load 飙高，用户无法正常下单；
- 在线选课时同一时刻提交选课的请求过多，系统无法响应；
- 页面服务中某一块内容访问很慢，一直加载不出来，导致整个页面都卡住，无法正常操作。

影响微服务可用性的因素有非常多，而这些不稳定的场景可能会导致严重后果。我们从微服务流量的视角来看，可以粗略分为两类常见的场景：

- 服务自身流量超过承载能力导致不可用。比如激增流量、批量任务投递导致服务负载飙高，无法正常处理请求。
- 服务因依赖其他不可用服务，导致自身连环不可用。比如我们的服务可能依赖好几个第三方服务，假设某个支付服务出现异常，调用非常慢，而调用端又没有有效地进行预防与处理，则调用端的线程池会被占满，影响服务自身正常运转。在分布式系统中，调用关系是网状的、错综复杂的，某个服务出现故障可能会导致级联反应，导致整个链路不可用。



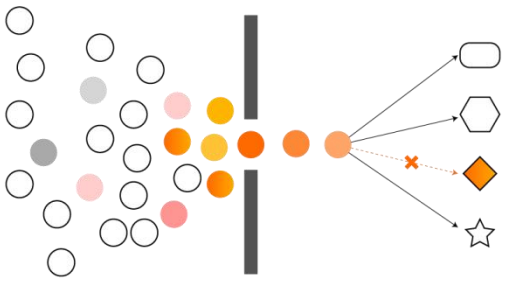
针对这些不稳定的场景，MSE 服务治理依托 AHAS 提供全方位的高可用流量防护能力。AHAS 基于阿里限流降级组件 Sentinel 的稳定性防护能力，以流量为切入点，从流量控制、并发控制、熔断降级、热点防护、系统自适应保护等多个维度来帮助保障服务的稳定性，覆盖微服务、云原生网关、Service Mesh 等几大场景。AHAS Sentinel 不仅在阿里内部淘宝、天猫等电商领域有着广泛的应用，在互联网金融、在线教育、游戏、直播行业和其他大型政央企行业也有着大量的实践。



微服务流量防护手段

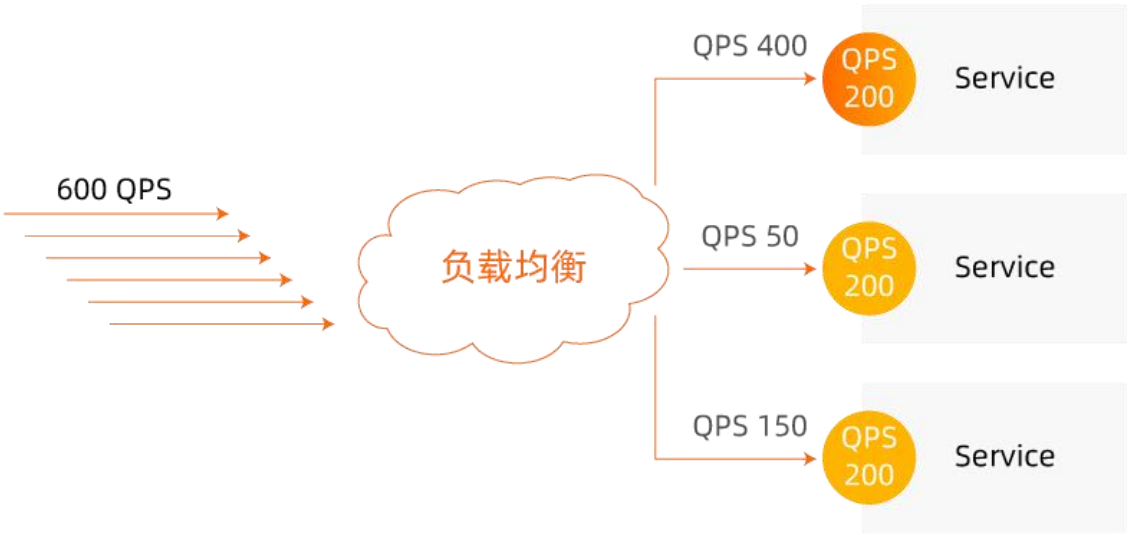
流量控制

流量是非常随机性的、不可预测的。前一秒可能还风平浪静，后一秒可能就出现流量洪峰了（例如双十一零点的场景）。然而我们系统的容量总是有限的，如果突然而来的流量超过了系统的承受能力，就可能会导致请求处理不过来，堆积的请求处理缓慢，CPU/Load 飙高，最后导致系统崩溃。因此，我们需要针对这种突发的流量来进行限制，在尽可能处理请求的同时来保障服务不被打垮，这就是流量控制。流量控制的场景是非常通用的，像脉冲流量类的场景都是适用的。



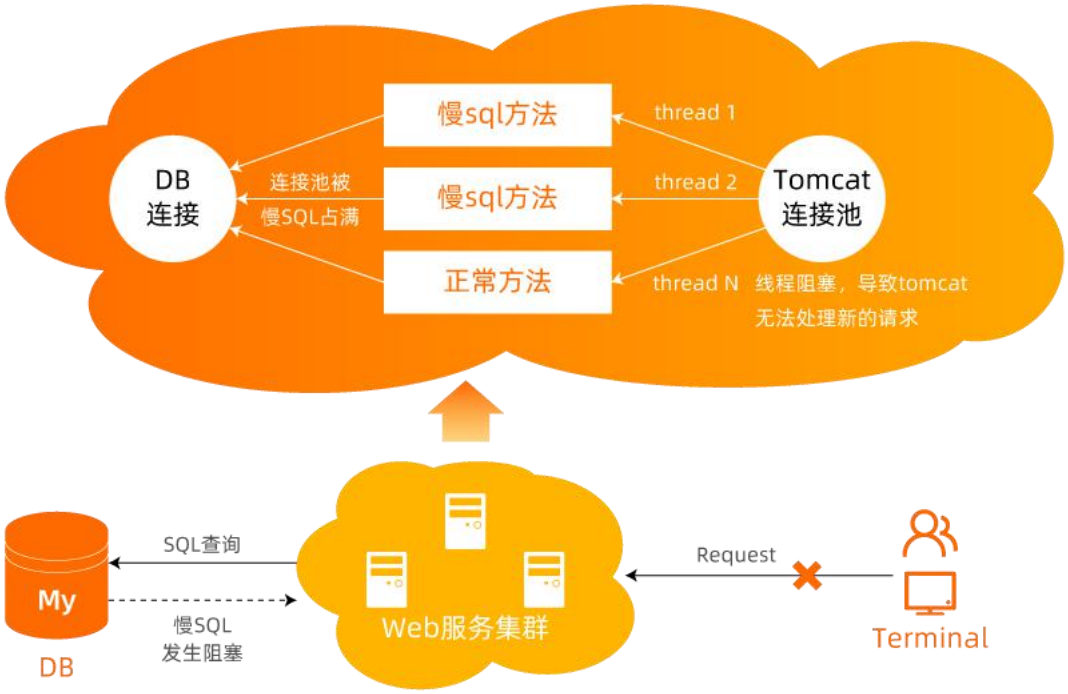
AHAS 基于毫秒级滑动窗口精确统计与令牌桶、漏桶、WarmUp 等流量控制算法，提供包括秒级精准流控、集群总量流控、匀速排队等在内的多种维度的流量控制场景。通常在 Web 入口或服务提供方（Service Provider）的场景下，我们需要保护服务提供方自身不被流量洪峰打垮。这时候通常根据服务提供方的服务能力进行流量控制，或针对特定的服务调用方进行限制。我们可以结合前期压测评估核心接口的承受能力，配置 QPS 模式的流控规则，当每秒的请求量超过设定的阈值时，会自动拒绝多余的请求。

同时，针对单机流控因流量不均匀、机器数频繁变动、均摊阈值太小导致限流效果不佳的问题，AHAS 还提供集群流控能力，可以精确地控制某个服务接口在整个集群的实时调用总量，结合单机流控兜底，更好地发挥流量防护的效果。集群流控既可以支撑数十万 QPS 大流量控制，也支持分钟小时级业务维度细粒度流量控制。

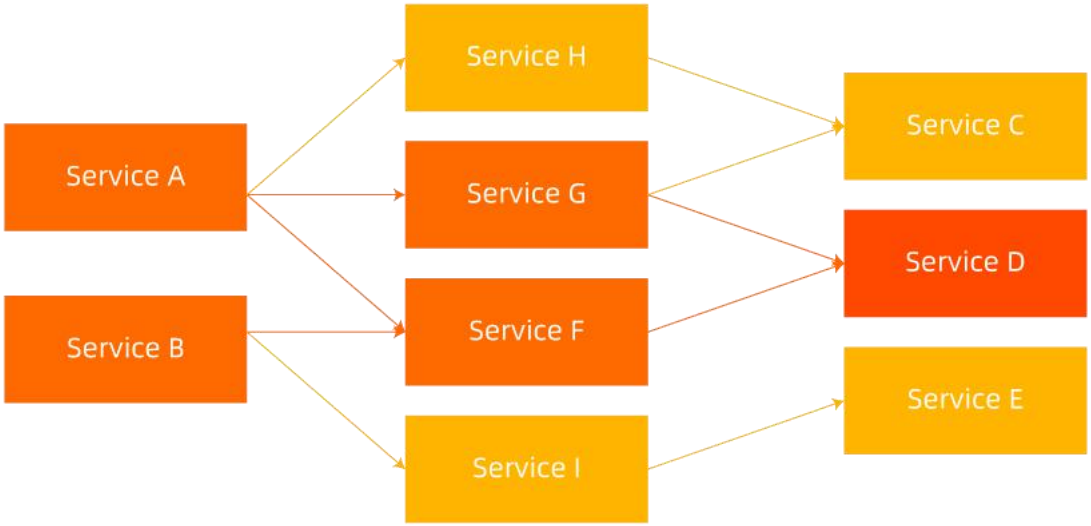


并发控制与熔断降级

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



现代微服务架构都是分布式的，由非常多的服务组成。不同服务之间相互调用，组成复杂的调用链路。以上的问题在链路调用中会产生放大的效果。复杂链路上的某一环不稳定，就可能会层层级联，最终导致整个链路都不可用。



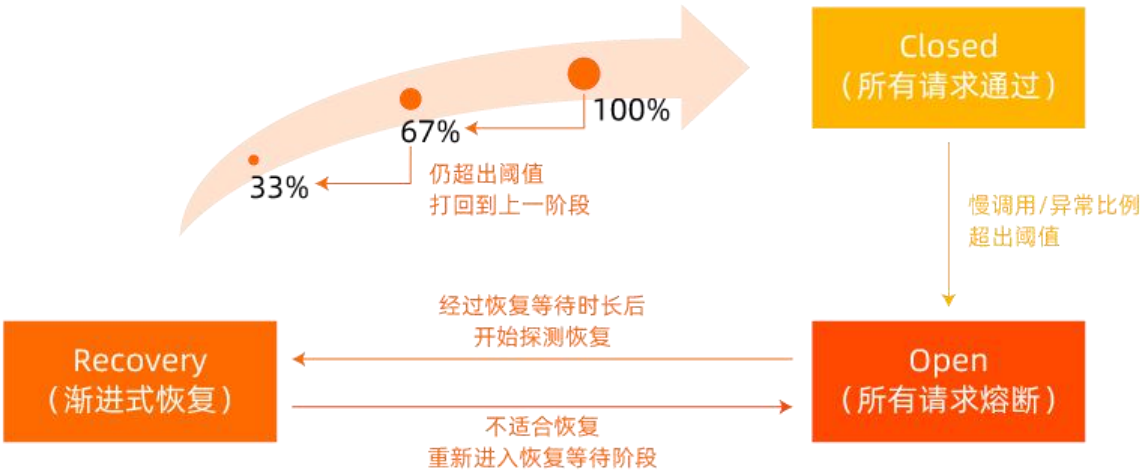
AHAS Sentinel 提供以下的能力避免慢调用等不稳定因素造成服务不可用:

- 并发控制（隔离规则）：作为一种轻量级隔离的手段，控制某些调用的并发数（即正在进行的数目），防止过多的慢调用占满线程池造成整体不可用。并发控制规则可作为一种重要的保底手段，防止服务被大量慢调用拖垮。

- 不稳定调用熔断：对不稳定的弱依赖调用进行自动熔断降级，暂时切断不稳定调用，避免局部不稳定因素导致整体的雪崩。

- 提前降级：对于一些弱依赖的服务（非关键链路依赖），可以在活动前或资源紧张时进行动态降级，以优先保障重要服务的稳定性。被降级的服务将直接返回给定的 mock 值而不会触发实际调用。

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



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

热点防护

流量是随机的，不可预测的。为了防止被大流量打垮，我们通常会对核心接口配置限流规则，但有的场景下配置普通的流控规则是不够的。我们来看这样一种场景——大促峰值的时候，总是会有不少“热点”商品，这些热点商品的瞬时访问量非常高。一般情况下，我们可以事先预测一波热点商品，并对这些商品信息进行缓存“预热”，以便在出现大量访问时可以快速返回而不会都打到 DB 上。但每次大促都会涌现出一些“黑马”商品，这些“黑马”商品是我们无法事先预

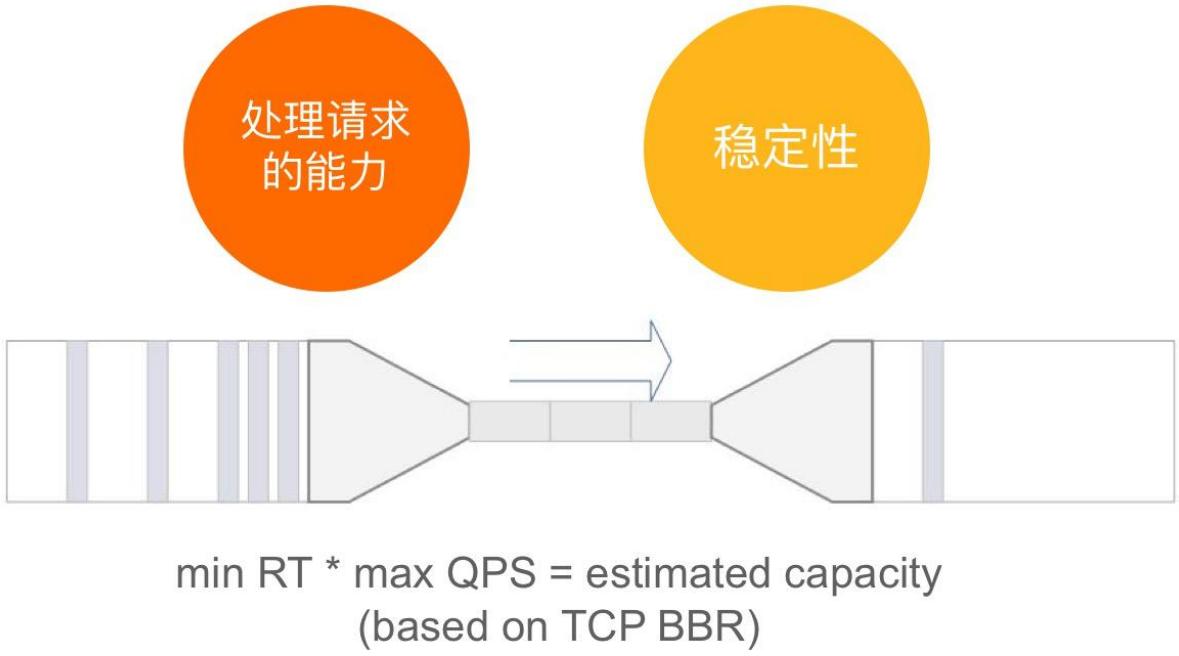
测的，没有被预热。当这些“黑马”商品访问量激增时，大量的请求会击穿缓存，直接打到 DB 层，导致 DB 访问缓慢，挤占正常商品请求的资源池，最后可能会导致系统挂掉。这时候，利用 Sentinel 的热点参数流量控制能力，自动识别热点参数并控制每个热点值的访问 QPS 或并发量，可以有效地防止过“热”的参数访问挤占正常的调用资源。



热点流量防护广泛使用于商品防刷、请求 IP 防刷、业务热点控制等场景，对于任意具有热点属性的请求均可适用。

系统自适应过载保护

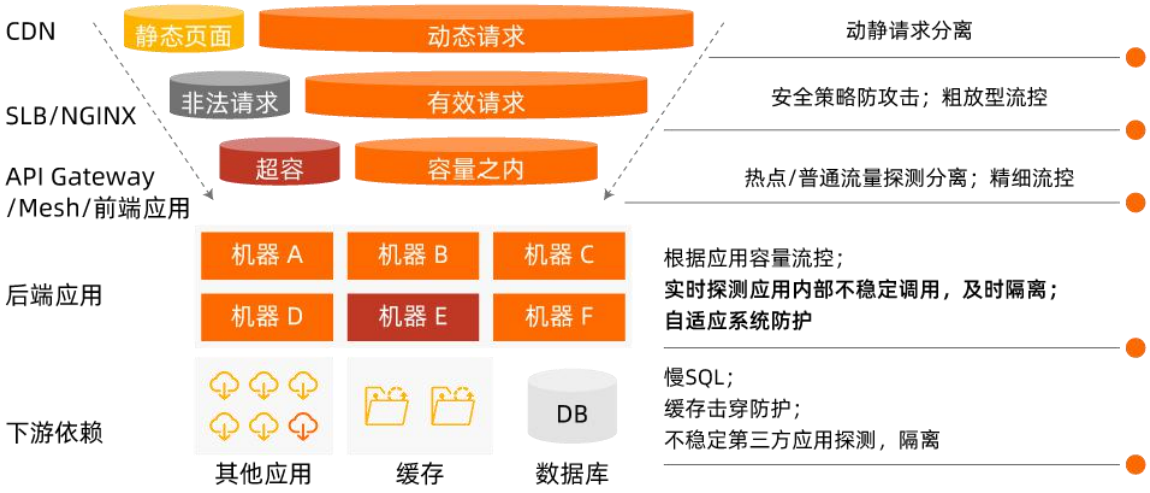
有了以上的流量防护场景，是不是就万事无忧了呢？其实不是的，很多时候我们无法事先就准确评估某个接口的准确容量，甚至无法预知核心接口的流量特征（如是否有脉冲情况），这时候靠事先配置的规则可能无法有效地保护当前服务节点；一些情况下我们可能突然发现机器的 Load 和 CPU usage 等开始飚高，但却没有办法很快的确认到是什么原因造成的，也来不及处理异常。这个时候我们其实需要做的是快速止损，先通过一些自动化的兜底防护手段，将濒临崩溃的微服务“拉”回来。针对这些情况，我们提供了一种系统自适应过载保护机制，结合系统指标和服务水位，动态预估系统最大容量，自适应动态调整流量。



AHAS 系统自适应保护策略借鉴了控制理论以及 TCP BBR 算法的思想，结合系统的 Load、CPU 使用率以及服务的入口 QPS、响应时间和并发量等几个维度的监控指标，通过自适应的流控策略，让系统的入口流量和系统的负载达到一个平衡，让系统尽可能跑在最大吞吐量的同时保证系统整体的稳定性。系统自适应过载保护可以作为整个服务的一个兜底防护策略，保障服务不挂，对 CPU 密集型的场景会有比较好的效果。

流量漏斗防护原则

在分布式系统架构中，每个请求都会经过很多层处理，比如从入口网关再到 Web Server 再到服务之间的调用，再到服务访问缓存或 DB 等存储。在高可用流量防护体系中，我们通常遵循流量漏斗原则进行高可用流量防护。在流量链路的每一层，我们都需要进行针对性的流量防护与容错手段，来保障服务的稳定性；同时，我们要尽可能地将流量防护进行前置，比如将一部分 HTTP 请求的流量控制前置到网关层，提前将一部分流量进行控制，这样可以避免多余的流量打到后端，对后端造成压力同时也造成资源的浪费。



高可用防护流程

通用的流量防护配置流程一般会分为几步：

1.事前评估，基于性能测试、强弱依赖梳理等手段，梳理核心接口及其容量，结合业务指标与系统监控数据，为限流降级配置提供参考；

2.结合评估的容量数据、性能数据，进行限流降级配置；

3.配置防护生效后的触发行为，如对于 RPC 服务，可以 mock 返回值或异常；

4.为了可以在服务出现流量飙高、慢调用的情况下业务方可以及时感知，我们还建议结合防护规则配置预警与告警规则，这样在临近阈值/触发阈值时可以快速感知并作出调整。



总结

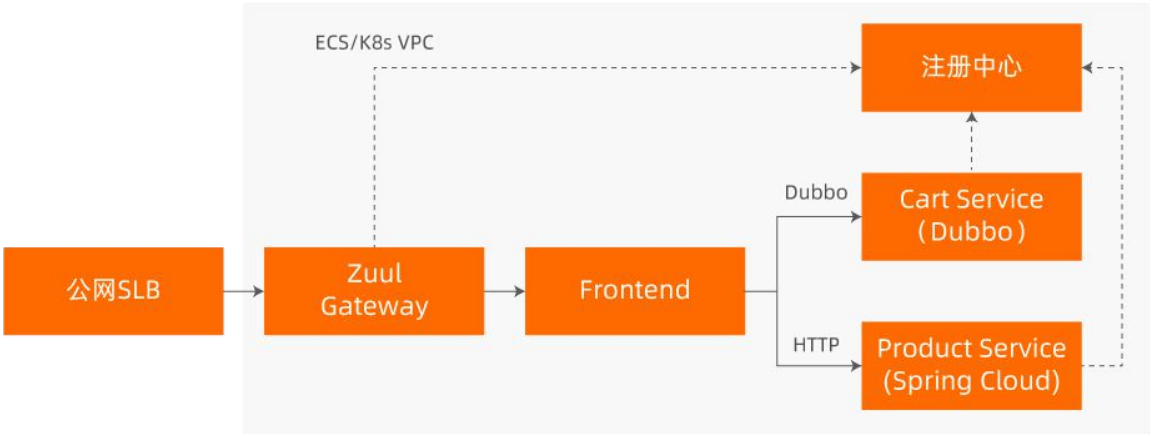
通过流控、熔断降级、系统自适应过载保护、热点防护等一系列的微服务流量防护手段，我们可以从微服务网关入口，到微服务，再到中间件依赖，这样全链路全方位地为微服务集群提供可用性保障。

3.6 微服务开发测试提效解决方案

前言

随着云原生时代的到来，越来越多的应用生在云上，长在云上，且随着越来越多的企业开始上云，云原生也是企业落地微服务的最佳伴侣。但云上应用易测性受到了很大的挑战，如何提高云上应用易测性，增强 DevOps 能力，是微服务测试要解决的核心问题。

在详细讲述微服务测试之前，先给大家讲一个场景。



上图是一个典型的企业微服务应用架构图，为了考虑安全性，云上应用通常部署在云上虚拟局域网内，统一通过网关对外暴露服务。对于负责 Product Service 应用的同学来说，我只想测试一下该应用对应的服务是否可用，他会怎么做呢？

方案一

进入该应用部署所在的机器（ECS）或者容器（Pod），通过 curl 命令验证该服务是否可用。

方案二

将该应用暴露给公网访问，通过本地命令行工具或者 Postman 工具验证该服务是否可用。

方案三

拉一条网络专线，打通云上专有网络 VPC 与办公网网络，通过本地命令行工具或者 Postman 工具验证该服务是否可用。

从以上场景，我们可以总结出云上微服务测试几点问题：

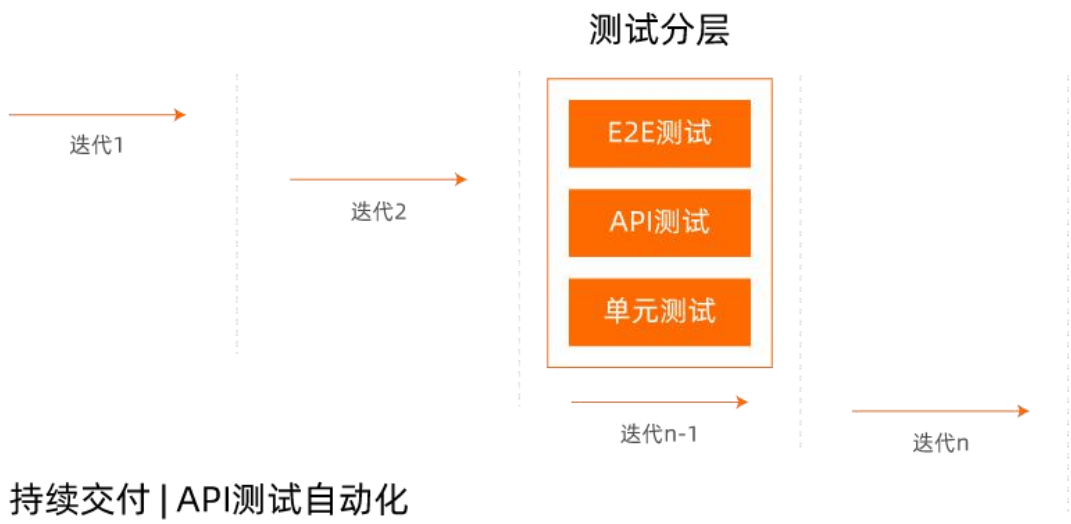
- **云上网络拓扑复杂**
- **暴露公网访问，会出现黑客攻击，引发安全风险**
- **拉一条网络专线，浪费资源成本**

明明只想要一个简单的测试能力，成本却如此之高。**上述场景还仅仅是一个简单的调试功能，如果是压测、自动化回归、巡检、服务 Mock 等其他测试及稳定性保障手段，不仅仅要解决上述场景遇到的问题，还需要自建工具，脑补一下，都觉得成本太高。**

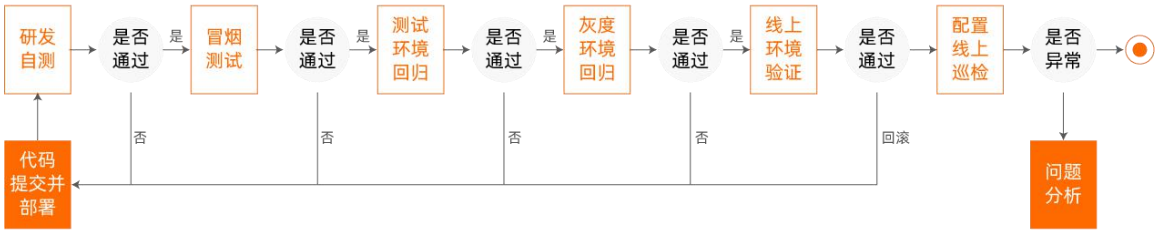
云原生时代下微服务开发测试

测试策略

以 API 为测试对象，进行开发自测，功能回归，性能测试，线上巡检等测试活动，完成高质量交付。



微服务应用测试，主要以 API 测试为主，占比达 60%-80%。微服务应用的测试策略，以 API 为测试载体，进行冒烟测试，回归测试，压力测试等，同时根据业务功能，对多个 API 进行编排，实现自动化测试，提高回归测试的效率，尤其在 DevOps文化盛行下，成为持续交付的质量保障利器。

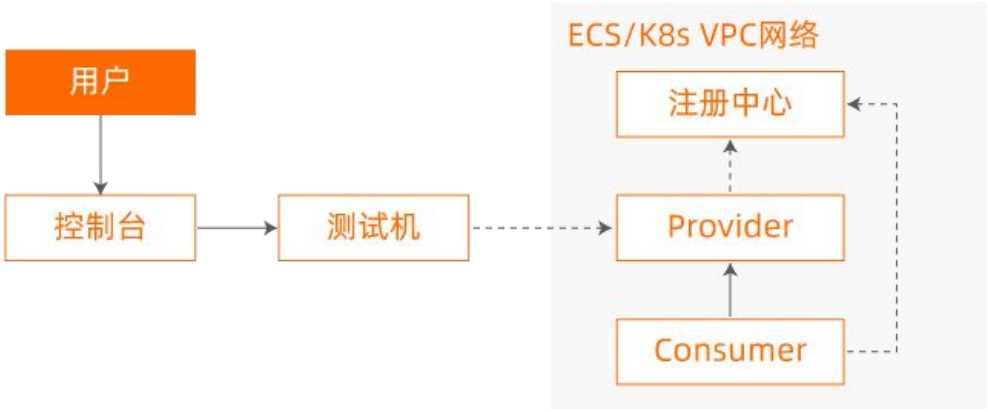


试想一下，研发同学提交代码并部署，可以使用测试工具，验证服务逻辑正确性；可以使用压测工具，验证服务性能指标；验证通过后，开始进行冒烟测试，可以使用自动化回归工具，编写冒烟用例；冒烟通过后，开始进行历史功能回归，可以使用自动化回归工具，编写回归用例；回归通过后，提交测试验收，测试只需要验证新功能，新功能验证通过后，即可提交发布。发布后，进行线上环境验证，需要回归历史功能主流程，可以使用自动化回归工具，编写主流程回归用例，新功能手工验证；主流程回归通过且新功能验证通过，代表发布完成；研发同学，可以使用巡检工具，配置线上巡检；一旦巡检告警，即可先于用户发现问题，并解决问题。云原生时代，我们希望有一个围绕 API 测试展开的测试工具，提高测试效率。

网络方案

安全可靠，拥有在办公网下的测试体验；开箱即用，无需关注专有网络 VPC 下的网络拓扑。

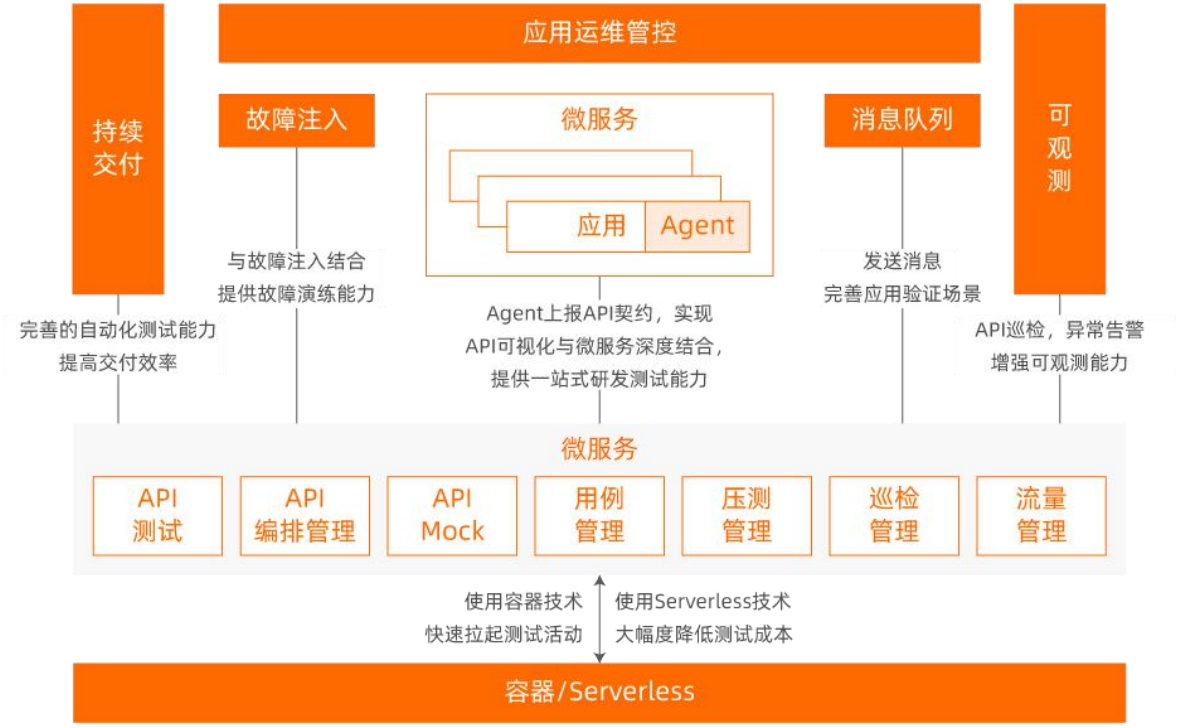
试想一下，企业为了安全隔离，研发环境、测试环境、预发环境、生产环境部署在不同的专有网络 VPC 内，如果用户自建测试工具，需要解决测试工具到不同环境的网络互通问题，企业 IT 人员明明只想要一个简单的测试工具，却因为上云之后，要解决复杂的云上网络拓扑，远远没有结束，为了能够在办公网使用该测试工具，还需要保证该测试工具能够被办公网访问，此时又面临着网络安全的考验。云原生时代，我们希望有一个能够开箱即用且安全可靠的方案，能够让上云的企业 IT人员拥有在办公网测试体验的测试工具。



测试能力

一方面借助云原生的能力，降本增效，另一方面为云原生应用测试提供基础能力，简单易用。

云原生微服务测试能力



业界存在很多 API 测试工具，相对优秀的工具，例如 ApiFox，Postman，提供了 API 管理，API 调试，自动化测试，测试管理，Mock 等能力，仍然是上一代测试工具，云原生时代，我们需要什么样的测试能力？试想一下，如何测试一个微服务接口，需要了解接口入参和出参，如果是研发同学-服务提供者，可能比较熟悉该接口，如果是测试同学，甚至是其他研发同学，可能就需要文档，甚至是口口相传，微服务治理已经可视化应用的 API 契约信息，结合 API 契约信息，即可进行快速调试。云原生时代，增强了 API 测试工具具备的基础能力，甚至在流量管

理，API Mock，压测管理等方面做了极大的增强，流量管理，支持 springCloud、dubbo 等框架的微服务应用流量录制，从而0 编码成本完成 API自动化测试和性能压测，同时能更全面的测试到业务操作场景对应的 API，防止测试遗漏。API Mock，主要解决了在依赖第三方API 未就绪的情况下，通过 Mock 功能，自动根据请求参数返回不同的结果，支持随机生成返回数据，能够真实地模拟后端服务，支持系统联调，及自动化测试的落地。压测管理，可以直接将自动化测试用例一键转化为压测用例，用户无需关心压测机的准备，即可快速了解单个 API 或者用例的并发数和 TPS。

云原生时代微服务测试以 API 为测试对象，屏蔽网络复杂性，提供一套简单易用的测试能力，提供研发测试效率，加速软件交付。结合上述 3 点，测试同学只需负责用例编写+测试验收，API 调试、API 性能水位、用例自动化均可赋能给研发同学，就像早期 DevOps一样，降低研发运维之间的反馈回路，提高软件交付效率，

DevTest，降低研发测试之间的反馈回路，在保证交付质量的前提下，进一步提升软件交付效率，同时主动创建巡检任务，定时监控线上服务可用率，先于用户发现问题，解决问题。

3.7 微服务敏捷开发解决方案

微服务敏捷开发不简单

安得环境千万套，大庇开发小哥俱欢笑

微服务给大家带来了敏捷开发的特性，基于敏捷开发的带来的便利，让我们可以在同一个时间内多个迭代/feature 并行开发。但是，微服务架构本身也给开发环境带来了一定的复杂性：每个 feature 的修改点都可能会被分散在多个应用中，需要多个应用互相配合才能完成整体的逻辑。这些应用既需要互相配合好，又不能让他们互相影响，敏捷开发有时候也不是那么容易。

相信实践过微服务敏捷开发的同学都曾经遇到过以下情况：

1.开发接口时，应用无法独立地联调测试，需要依赖于下游的返回，所以一般都需要一个完整的开发环境，这个环境需要包含所有的其他应用。

2.A 同学辛辛苦苦，终于开发好了一个接口，但是部署到开发环境后，发现返回值一直是错的，就是不符合预期，百思不得其解。最终根据日志、arthas 层层跟踪下去，发现原来是另一个同事更新了下游应用的代码，导致原有逻辑发生了变更。

3.A 同学准备开始联调测试了，这时候他要找到开发 B 和 C 吼一嗓子确认：**“我要开始测试了哈兄弟们，你们都别动环境，不要重启和 debug 哈”**。B 同学 和 C 同学一脸懵逼：**“我自己这还有个逻辑没理清楚呢，刚改完代码准备测一发，你这一测试联调我就不能动环境了，我这功能得等到什么时候才能开发好”**。

4.排查问题好麻烦啊，要不直接 debug 一下吧，这 IDEA 远程 debug 刚连上去呢，立马就传来了同事的声音：**“谁 XX 又在瞎动环境啊，怎么刚刚还能跑的接口现在就出错了”**。

以上这些问题显然会影响项目的进度，非常容易造成项目延期。**对于此刻的开发小哥哥而言，拥有一套属于自己的独立环境，带来的幸福感也许比有一套属于自己的小房子还大。**

流量闭环是微服务敏捷开发的基础

上文中提到的问题，其实都是因为没有在开发环境中，精准地控制流量在 feature 环境内流转。

为什么精准地控制流量如此重要？举个最简单的微服务架构图来说明，这里假设应用的调用链路为 A ---> B ---> C ---> D ，现在同时开发两个 feature，feature1 和 feature2 。feature1 需要修改 A 和 C 的代码，feature2 需要修改 B、C 和 D 的代码。

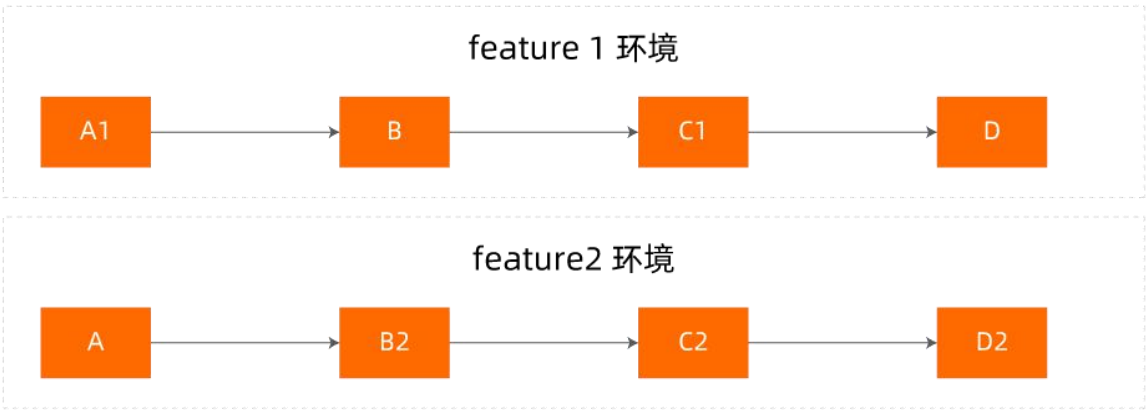
为了方便表述，我们用 A、B、C、D 来代指 A、B、C、D 的线上稳定版本，也叫做基线版本；A1、C1 来代指 feature1 环境中的 A 和 C ；B2、C2、D2 来来代指 feature2 环境中 B、C、D。

那么开发测试 feature1 的同学会要求他的请求，准确地在 A1 ---> B ---> C1 ---> D 中流转。为什么一定要这样，我们来简单分析一下：

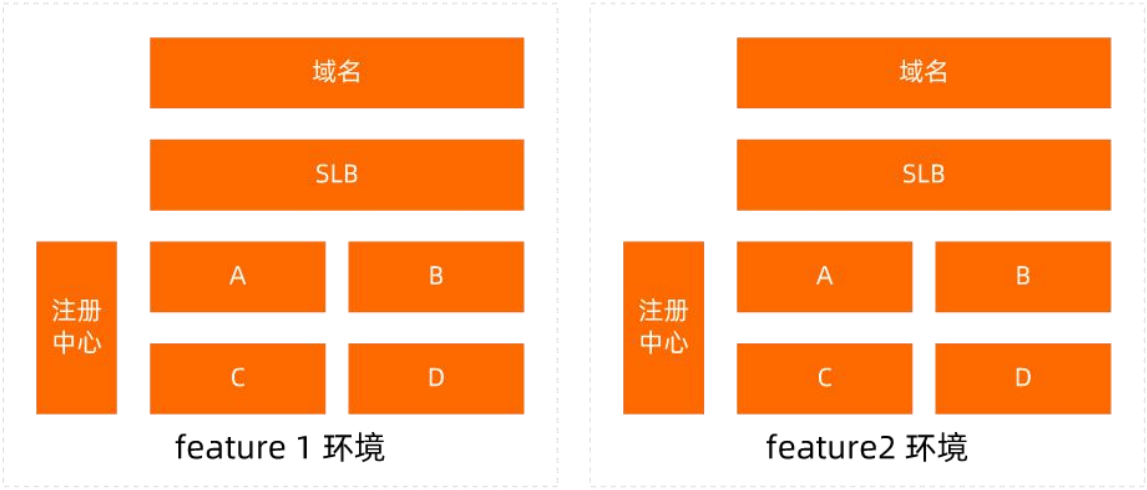
1.如果流量走到 A 或者 C 的基线环境，因为他们都没有包含 feature1 相关的代码，所以肯定是无法正常测试和联调 feature1 对应的功能。

2.如果流量走到 B2、D2 环境，大多数情况下是可以正常工作的，因为正常情况下 B2 和 D2 中的修改是不会影响 feature1 的。但是因为 feature1 和 feature2 可能是由不同的同学开发的，或者有不同的开发排期和节奏，他们有自己的开发、重启、debug 节奏，所以大概率还是会出现上文中提到的场景。

综上所述，让流量在 feature 环境内流转非常重要，是微服务敏捷开发的基础。



如何准确地让请求在 feature 环境内流转呢？最简单的办法是每个迭代/Feature 的都享有一套独立的完整环境，这套独立的环境包含了整个微服务应用集所有的应用，包含注册中心和接入层，这样就能确保流量在 feature 环境里闭环，不用担心应用之间互相影响。



这个解决方案虽然简单，但是问题也很显而易见，成本比较大。我们假设微服务应用有 10 个，每个应用只部署一台，以 java 为例，部署一个 java 应用按 2C4G 的 共享标准型 ECS 进行计算，维护一套环境一年的成本是 10 × 140 × 12 = 16800 元，**如果同时有 4 套环境，即只支持两个迭代并行开发，每个迭代只有 2 个 feature，这样一年的成本就是 67200 元，而且我们可以发现，这里面计算公式使用的是乘法，当应用增加和环境增加时，成本的增加是成倍的。**

注意，这里只是单纯地计算了应用使用的 ECS 的成本，其他周边的配套设施我们还没有计算，因为我们的开发、联调、测试是需要确保端到端的全流程都是 OK 的，那这里就还会涉及到域名/SLB/网关/注册中心这些资源，这些资源一般比较固定，不会需要进行大的修改，但是在多套环境的方案下这些资源也需要维护多套，成本还会进一步上升。

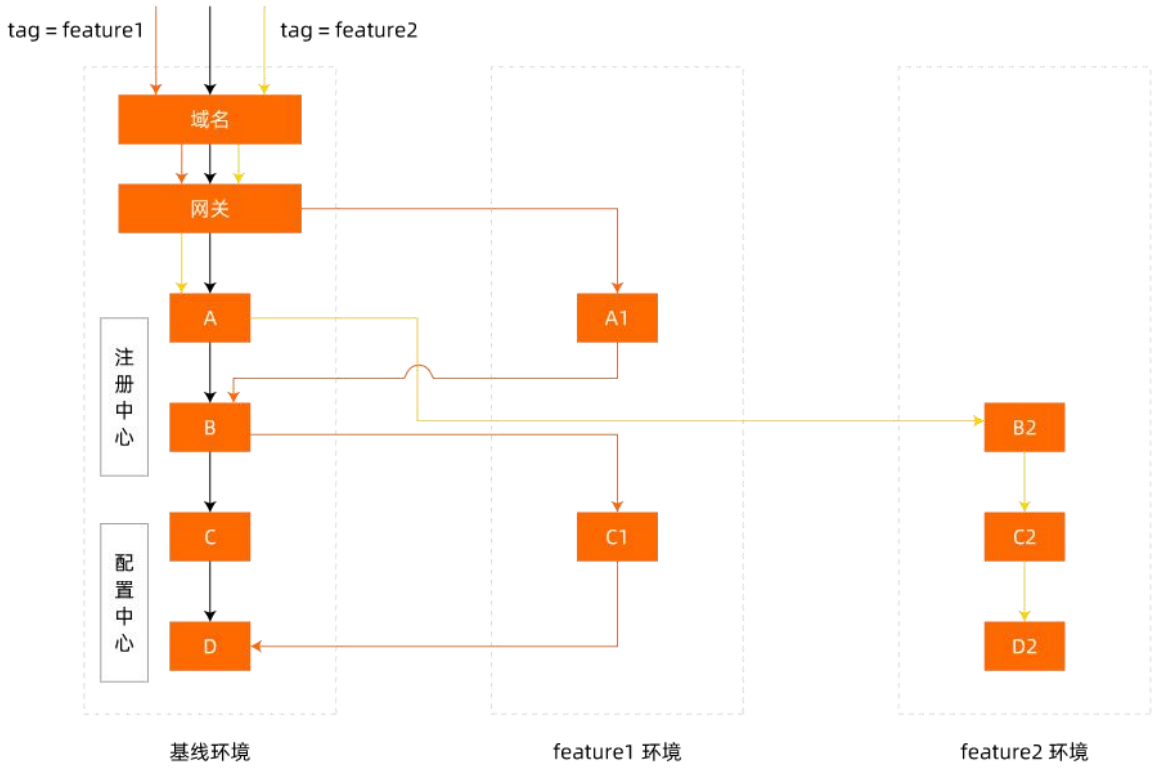
那么，有没有一个比较优雅地方式，既能享受到微服务架构带来的敏捷开发的便利，又不会给日常开发环境的搭建带来很大的成本呢？基于 MSE 标签路由功能使用开发环境隔离方案是您的不二之选。

如何低成本玩转敏捷开发

排除了物理环境隔离的方式之后，我们很自然地去想到，逻辑隔离。简单地说，表面上看起来有很多套环境，每个环境都有一套完整的微服务应用。但是这些环境内的有些应用节点不是只属于某一个环境的，是被多个环境共享的，这样就能减少很多机器的成本。理想中的逻辑隔离

可以做到这样，只需要维护一套完整的基线环境，在增加 feature 环境时，只需要单独部署这个 feature 所涉及到改动的应用即可，而不需要在每个 feature 环境都部署整套的微服务应用及其配套设施。

我们称之这唯一的一套完整的环境为基线环境。基线环境包含了所有微服务应用，也包含了服务注册中心、域名、SLB、网关等其他设施，而 feature 环境中只包含了这个 feature 中需要修改的应用。这样维护 n 套 feature 环境的成本，就变成了加法，而不是原来的乘法，由 n × m 变成了 n + m。**差不多相当于零成本增加 feature 环境，这样我们就可以放心地扩容出多套 feature 环境，每个开发小哥哥都可以轻松拥有属于自己的独立环境，尽情地享受微服务敏捷开发。**



从上图中我们可以看到，feature1 对应的流量，在发现 feature1 中存在 A1 应用时，一定会去往 A1 节点，A1 在调用B 的时候发现 feature1 环境中不存在 B1 ，则会将请求发到 基线版本的 B 中；B 在调用C 时，发现 feature1 环境存在 C1 应用，又会返回到 feature1 环境中，依次类推，确保了流量会在 feature1 环境中闭环。

逻辑隔离的方式确实比物理隔离强大了不少，但是实施起来的复杂度也高了很多。

从开源的角度来看，实现逻辑隔离有个比较通用的方式。服务提供者将环境标相关的信息注册到注册中心里，于是消费者从注册中心中获取的服务提供者列表就包含了环境相关的信息。消费者在路由的过程中，对请求的内容进行计算，选择出与之对应的目标环境，最后再与服务提供者列表中的元数据信息进行匹配，找到正确的目标节点进行路由。

描述起来比较简单，但是这个过程中，也有一些需要注意的地方。

1.在请求的链路中，调用中包含了参数千变万化，源头的 header 信息不一定能被带到下一跳，如何使得入口处配置好的规则，能够在整条链路都生效，这里是一个问题。

2.逻辑隔离的过程中，会存在某个应用不存在 feature 环境的情况，如何在请求被降级到 base 环境后，能够重新回到 feature 环境，这里需要借助于全链路流量透传去传递。

3.微服务的版本比较多，需要对于不同的版本进行适配，同时保证在出现流量规则冲突时，有兜底的措施。

这一些逻辑实现起来还是有些困难，但是基本原理，都是类似的。

当然也可以使用阿里云微服务引擎 MSE 提供的方案，开箱即用，在使用的过程中，您不需要修改任何代码和配置，直接接入 MSE 微服务治理即可使用，不会给您增加任何开发成本。

3.8 微服务无缝迁移上云解决方案

背景介绍

微服务系统架构之注册中心

相比于传统的单体应用，使用微服务系统架构虽然能带来如各微服务之间可独立开发、运行以及部署等优点。但如何对一个微服务系统中的众多微服务进行注册与管理是微服务系统设计过程中的核心内容之一。当前主流的微服务架构中，主要通过设计一个叫作注册中心的组件来提供对系统中所有服务进行状态注册与发现。注册中心本质上是一个数据库，其需要实时存储系统中所有服务的有关状态以及对应的实例列表信息，由于应用在分布式系统中，其还需要提供一定容错、高可用等相关能力。下图是微服务之间通过服务注册中心来提供服务注册与发现能力实现服务调用的流程图。

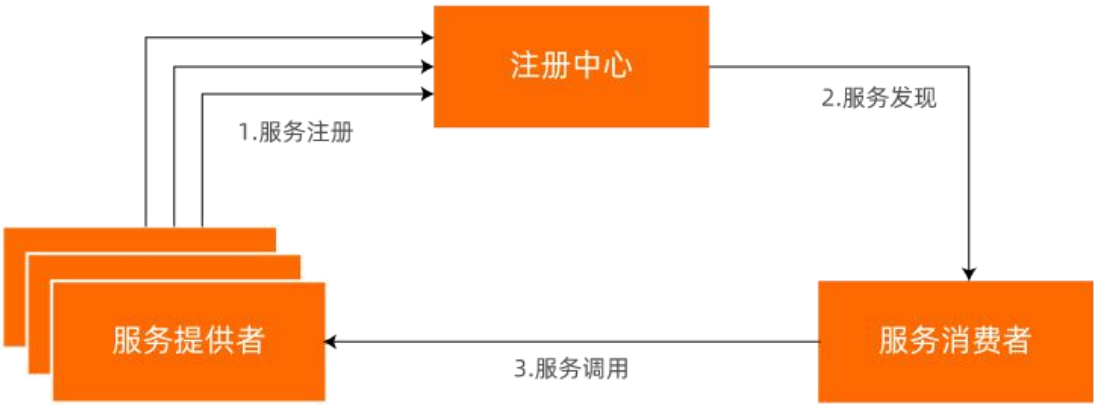


图 1 基于注册中心的微服务系统调用

如上图所示，在微服务系统中，所有的微服务实例启动时都会根据配置信息将服务有关的如服务名称，服务地址以及端口号等信息发送到系统的注册中心保存，注册中心后续过程中也会实时通过心跳等机制探测服务实例的健康程度并及时更新服务状态。服务注册后，调用方通过服务标识到注册中心中获取对应服务的可用实例列表。最后再根据特定的客户端负载均衡机制从服务实例列表中选取特定实例发起服务调用。

当前业界较为知名的注册中心主要有 ZooKeeper、Eureka、Consul 以及 Nacos 等。

ZooKeeper

ZooKeeper 由雅虎公司基于 Google 的 Chubby 论文[1]进行实现 。ZooKeeper 集群采用 Master/Slave 架构，利用原子广播（Zookeeper Atomic Broadcast，ZAB）协议来进行 Master 选举与保证数据强一致性同步。在被广泛应用于服务注册与发现之前，其主要应用在为大型分布式系统提供服务配置管理和分布式协同等能力。

Eureka

Eureka 由 Netflix 开源[2]，作为一款 Spring Cloud 官方推荐使用的注册中心，其是一款较早并与 Spring Cloud 结合较好的注册中心组件。Eureka 集群采用非Master/Slave 架构，集群中所有节点角色一致，数据写入集群任意一个节点后，再由该节点向集群内其他节点进行复制实现弱一致性同步。

Consul

Consul 是一款由 HashiCorp 公司开源的基于 Go 语言实现的注册中心组件，提供了服务发现、配置 和分段等功能[3]。其也是采用Master/Slave 架构来进行集群节点管理，底层基于 Raft 协议来实现 Master 节点选举与数据强一致性同步。

Nacos

Nacos 是一款由阿里巴巴在 2018 年 7 月开源的微服务注册中心组件[4]，除了正常注册中心所具备的核心功能外，其还提供了微服务配置中心的相关能力。 Nacos 集群架构也属于非 Master/Slave 模型，采用了阿里巴巴内部自研的 Distro 协议来实现数据弱一致性同步。其中所有节点都将存储集群内所有服务元数据，因此都可以提供数据读取服务，但每个节点只负责一部分客户端的数据写服务。

由以上介绍可知，以上各注册中心的特性如下：

	ZooKeeper	Eureka	Consul	Nacos
集群架构	Master/Slave	非Master/Slave	Master/Slave	非Master/Slave
一致性协议	ZAB	~	Raft	Distro+Raft
CAP 模型	CP	AP	CP	AP
雪崩保护	无	有	无	有
协议访问	TCP	HTTP	HTTP/DNS	HTTP/DNS
SpringCloud 集成	支持	支持	支持	支持
Dubbo 集成	支持	不支持	不支持	支持

注册中心迁移上云

得益于过去一二十年的互联网技术的飞速发展，业界出现了适用于不同场景具备不同特性的多种注册中心可供用户选择。但因为各种注册中心架构以及形态各异应用的场景不一，不管系统在设计之初采用什么注册中心解决方案，随着系统业务的发展演进，或多或少都可能遭遇原有注册中心难以继续满足当前系统对注册中心服务能力的要求，以下是一些常见的注册中心迁移原因：

1.早期注册中心技术方案有缺陷，例如注册中心的可用性要求强于一致性，因此早先配合 Dubbo 框架使用较多的强一致性注册中心 ZooKeeper，后来被证实并不适合作为大规模微服务系统的注册中心解决方案。相关分析可参见文章[5]。

2.技术前瞻性不够，随着系统业务流量快速增长，早期采用的注册中心解决方案成为了系统发展的瓶颈。

除了以上技术设计造成的原因外，对于广大中小企业来说，自建注册中心技术难度大、成本高、稳定性得不到保障的等诸多原因近年来促使一大批中小企业放弃了自建注册中心想法，转而采用了云厂商所提供的免运维、免托管、性能好、稳定性强以及成本低的云注册中心方案。

注册中心方案介绍

要想完成微服务应用迁移上云，注册中心迁移上云是其中的关键。目前业界主流的注册中心迁

移上云方案主要分为停机迁移和非停机迁移两大类：

- **停机迁移**

停机迁移是最容易想到、最朴素的一种实现注册中心迁移上云的方案，其是指将应用进行停机，然后逐个将应用中的自建注册中心配置修改成云上注册中心配置，最后通过对应用进行重新发布以实现注册中心的迁移上云。该种方式特点简单，但所带来的劣势是工作量大、涉及人员较多、流程繁琐耗时，导致注册中心迁移成本高、难度大、影响面广。

- **非停机迁移**

非停机迁移是相对于停机迁移的一种方案，其在注册中心迁移过程中不要求应用立即停机修改代码，通过切流迁移、自建注册注册中心与云上注册中心数据实时同步或者双注册双订阅的方式来实现在不影响业务正常运行的过程中心注册中心的平滑迁移。

接下来，本章主要对业界主流的几种非停机注册中心迁移方案进行介绍；

切流迁移

切流迁移作为非停机迁移中较为容易实现的一种方式，其主要通过开发一套新的应用，在应用中使用新注册中心，然后通过 SLB 和域名配置来进行线上切流。该方案虽然能保证迁移过程的用户无感知，但需要重新部署一套已有系统，代价较高。

Nacos Sync

Nacos Sync 是 Nacos 社区提供了一种注册中心迁移方案，可以从一个注册中心服务端，同步注册的数据到另一个服务端。Nacos Sync 注册中心迁移方案原理图如图 2 所示：

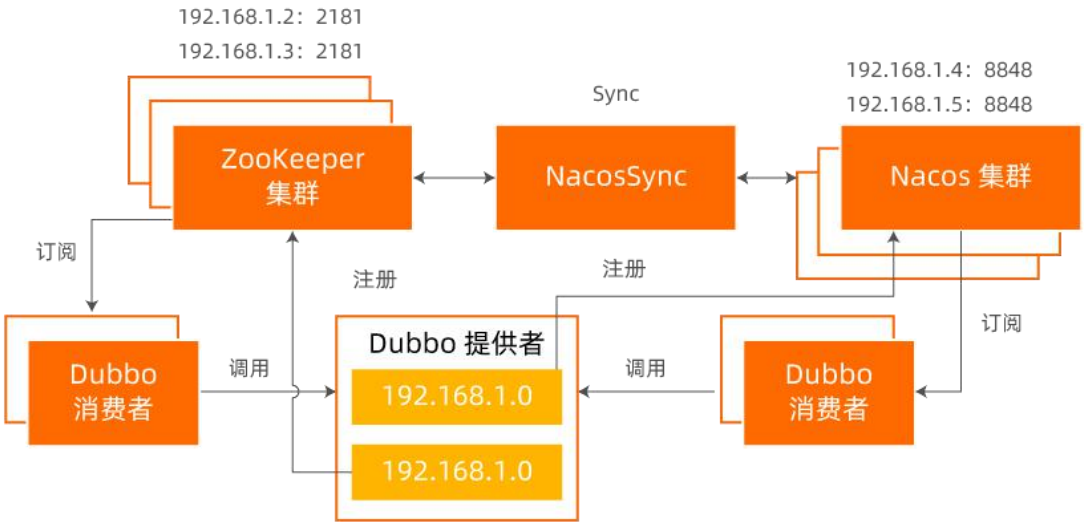


图 2 Nacos Sync 注册中心迁移原理图

下图是 Nacos Sync 系统的概念图，Nacos Sync 通过从各个注册中心拉取注册的服务实例数据同步到 Nacos，左右两边是不同的注册中心，绿色代表目前是可以进行双向同步的，蓝色代表暂时只能进行单向同步。在启动完数据库，并配置好数据库连接串并启动 Nacos Sync 之后，该方案的整体的使用流程如图 3 所示：

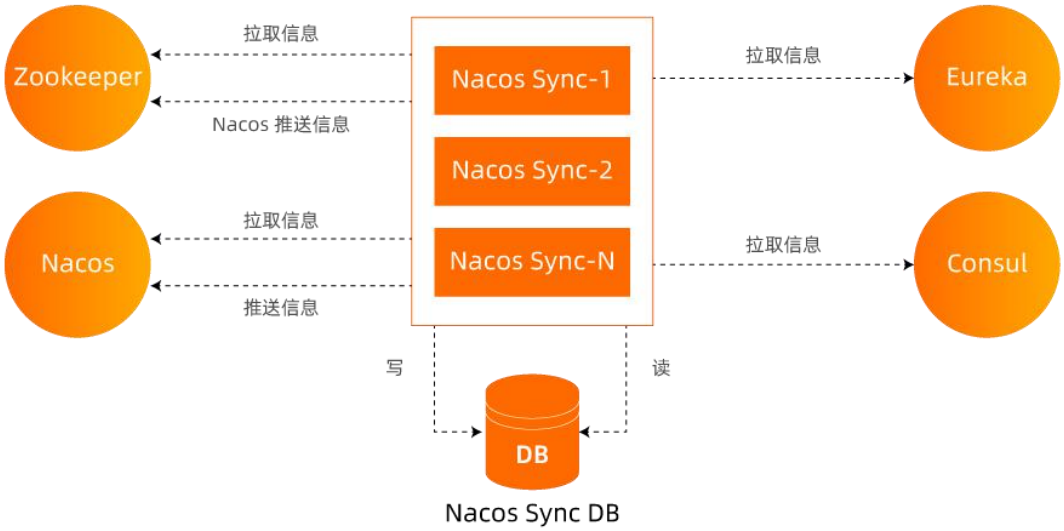


图 3 Nacos Sync 概念图

使用Nacos Sync 进行注册中心迁移的步骤如下：

1.通过 Web 控制台添加相关注册中心，一般都必须配置两个注册中心，一个源注册中心，另外一个是目标注册中心，注册中心相关数据会写入到数据库进行持久化存储；

2.添加完注册中心后，需要增加一个同步任务来添加需要同步的服务(对于 Dubbo 来说就是 RPC 接口);

3.Nacos Sync 会每隔 3s 从数据库捞取同步任务，并通过异步事件的方式进行发布;

4.同步服务管理组件监听到定时任务发布的事件，目前有同步/删除这两种事件;

5.同步服务管理根据不同的策略选择相关的同步服务进行真正同步逻辑处理;

总结来说，在使用该方案进行注册中心迁移过程中，首先需要部署 Nacos Sync 应用，并配置数据库连接串相关信息，然后将需要同步的服务一个个输入到控制台，才能实现注册中心同步迁移。从该方案使用流程可知，其存在以下不足:

1.需要一个独立的数据库，以及启动一个 Nacos Sync 进程，对资源的消耗比较大。

2.需要一个个输入服务名，才能使劲服务的同步，操作起来复杂繁琐，注册中心迁移工作量大。

3.同步的时效性不好，只能控制在**秒**级别。

双注册双订阅

双注册双订阅非停机迁移注册中心方案是通过一些技术手段，如给应用动态注入Sidecar 或者利用字节码技术动态修改应用代码，在应用中动态加入新注册中心依赖包和配置信息，使其在仅需要一次重启就能实现对自建注册中心和新注册中心的双注册以及双订阅，该方案典型代表是阿里云微服务治理团队提出的基于 Java Agent 技术的注册中心迁移[6]。该方案原理图如图 4 所示:

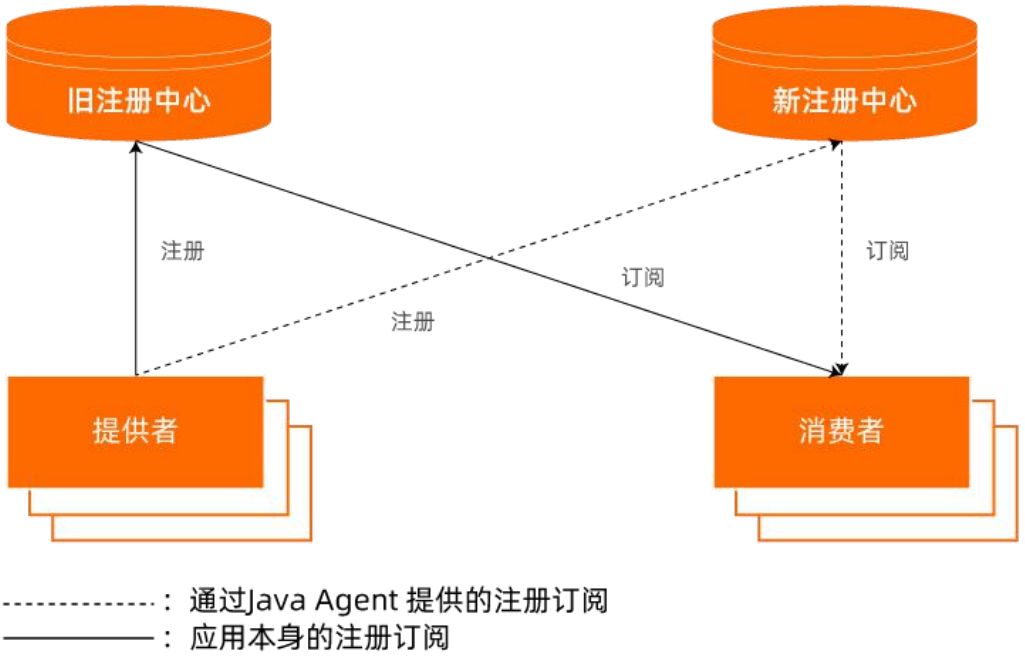


图 4 双注册双订阅实现注册中心迁移上云

无论是通过 SideCar 还是字节码技术实现双注册双订阅来实现注册中心迁移其基本步骤如下：

1.快速重启应用挂载 Sidecar 或者 Agent 来实现应用的双注册和双订阅。

2.将系统中下次需要进行发布升级的应用中的自建注册中心地址改成新注册中心地址，然后发布上线。尽管新发布的应用不会注册或者订阅老注册中心，但因为其他服务在新注册中心都有注册或者订阅一次，所以不影响其订阅其他服务或者被其他服务消费。

3.随着一段时间后，系统中的所有应用都完成了注册中心地址修改，即可下线老注册中心以完成整个系统的注册中心全部迁移工作。

由上述方案介绍内容可知，基于双注册双订阅的非停机注册中心迁移方案具有迁移过程便捷、迁移成本低等诸多优势。

参考资料

[1]Mike Burrows . The Chubby Lock Service for Distributed Systems.
https://static.googleusercontent.com/media/research.google.com/zh-CN//archive/chubby- osdi06.pdf
[2]https://github.com/Netflix/eureka
[3]https://www.consul.io/docs/intro
[4]https://nacos.io/zh-cn/docs/what-is-nacos.html
[5]阿里巴巴为什么不用 ZooKeeper 做服务发现
https://developer.aliyun.com/article/601745
[6]注册中心迁移
https://help.aliyun.com/document_detail/314237.html

3.9 线上故障紧急诊断、排查与恢复



B 站崩溃，让年轻人无心睡觉...

富途证券服务中断，创始人发 2000 字硬核长文解释技术故障...

西安“一码通”半个月崩溃两次...

百年不遇的故障，一年三次。诸如此类的大型 IT 系统故障每隔一段时间都会出来一个。

概述

当前大型互联网系统架构日趋复杂，稳定性风险也在升高，系统中一定会有一些黑天鹅潜伏着，只是还没被发现。然而墨菲定律告诉我们“该出错的终究会出错”。因此需要有更有效的方式避免线上故障，在不可避免发生故障情况下，希望能够快速修复，减少线上影响，基于此提出了 1-5-10 的快恢目标，1-5-10 的目标是要我们对于线上问题能够做到 1 分钟定位，5 分钟定位，10 分钟修复。

1 分钟发现

监控

监控的作用一句话概括就是：发现应用中的问题，并将问题及时告警给技术人员进行处理。监控类型可以分为系统问题的监控与业务问题的监控，系统问题：常见的软硬件相关问题，比如程序异常，内存 fullGC 等，由于没有业务特征，监控策略可适用于各个应用。业务问题：在特定业务场景下定义的问题，比如商品无优惠券，权益超发问题等，需要根据业务特征来定制监控策略。

阿里云实时应用监控服务 ARMS 能够自动发现和监控应用代码中常见的 Web 框架和 RPC 框架，并统计接口的调用量、响应时间、错误数等指标。同时可以进一步获取接口的慢 SQL、MQ 堆积分析报表或者异常分类报表，对错、慢等常见问题进行更细致的分析。

ARMS 还提供了业务监控的能力，以代码无侵入的方式，可视化定义业务请求，提供贴合业务的丰富性能指标与诊断能力。从业务视角衡量应用性能和稳定性的新方式，对业务的关键交易进行全链路的监控。业务监控通过追踪并采集应用程序中的业务信息，实时展现业务级的指标，例如业务的响应时长、次数和错误率，解决了应用程序和业务表现之间无法映射关联的难题。

对于监控的要求有以下三点。实时：要求对问题的发现和预警是实时的，缩短问题产生和发现的时延；准确：要求监控和预警是准确的，包括对监控问题的定义，对预警阀值，预警等级，责任人的配置，避免误报；全面：要求预警信息是全面的，能够帮助排查和解决问题。

“不论应用出现任何问题，ARMS 都可以清楚地展示问题出在哪一行代码。ARMS 对于我们非常重要，大大缩短了修复故障的时间，显著提升了用户体验。自从用了 ARMS，我们能及时发现和修复问题，再也不会被用户投诉所困扰。” -- 华润万家

告警

当监控发现有问题的时候，就需要通过不同等级的告警将问题及时告警给技术人员进行处理。ARMS 告警管理能从以下几点来提升系统的运维效率。

- 集成事件后管理更高效。
  - 告警管理默认支持一键化集成阿里云常见的监控工具，并支持更多的监控工具手动接入，方便统一维护。
  - 事件接入模块稳定，能提供 7x24小时的无间断事件处理服务。
  - 处理海量事件数据时可以保证低延时。

- 及时准确地将告警通知给联系人。
  - 配置通知规则，对事件合并后再发送告警通知，减少运维人员出现通知疲劳的情况。
  - 根据告警的紧急程度选择邮件、短信、电话、钉钉等不同的通知方式，来提醒联系人处理告
  - 警。
  - 通过升级通知对长时间没有处理的告警进行多次提醒，保证告警及时解决。

- 帮助您快速便捷地管理告警。
  - 联系人能通过钉钉随时处理告警。
  - 使用通用告警格式，联系人能更好的分析告警。
  - 多个联系人通过钉钉协同处理。

- 统计告警数据，实时分析处理情况，改进告警处理效率。

5 分钟定位故障

Offline 一键保留现场

当问题发生的时候，我们可以通过在某个 Pod 上执行Offline 命令将该微服务进程从注册中心中进行地址下线，从而实现停止微服务流量的作用，使得流量不再访问执行过 Offline 的实例，从而一键保留现场，使得我们排查问题的过程中不会影响线上的正常流量。

Arthas 诊断

Arthas 是诊断 Java 领域线上问题的利器，利用字节码增强技术，可以在不重启 JVM 进程的情况下，查看程序的运行情况。

JVM 概览

JVM 概览支持查看应用的 JVM 相关信息，包括 JVM 内存、操作系统信息、变量信息等，帮助我们了解 JVM 的总体情况。

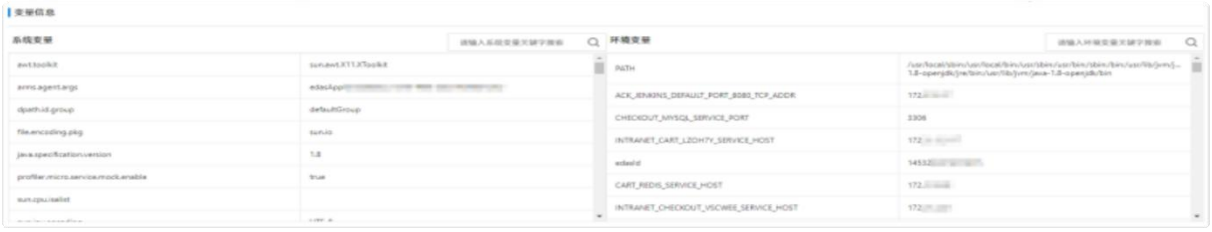
- JVM 内存：JVM 内存的相关信息，包括堆内存使用情况、非堆内存使用情况、GC 情况等。



- 操作系统信息：操作系统的相关信息，包括平均负载情况，操作系统名称、操作系统版本、Java 版本等。



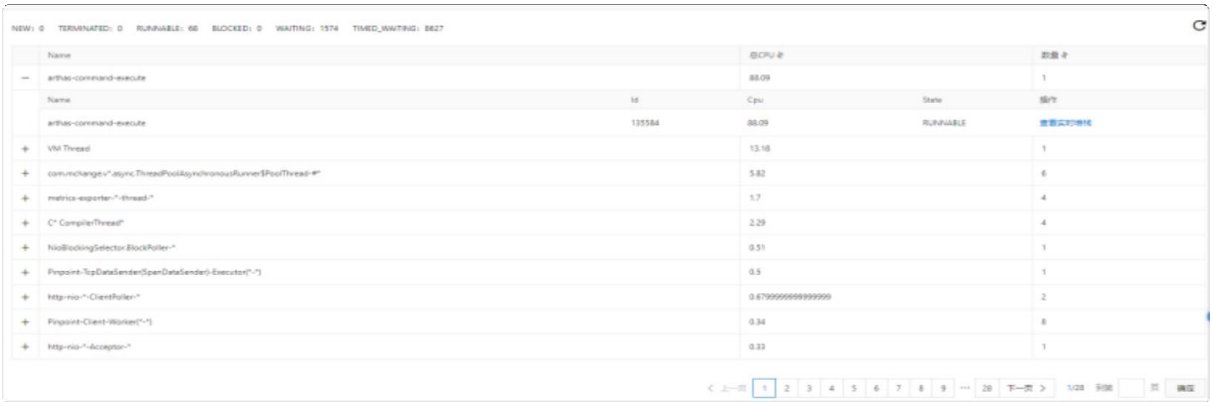
- 变量信息：变量的相关信息，包括系统变量和环境变量。



线程耗时分析

线程耗时分析支持显示该应用的所有线程和查看线程的堆栈信息，帮助我们快速定位耗时较高的线程。

1.线程耗时分析页签会实时获取当前 JVM 进程的线程耗时情况，并将相似线程聚合。可以查看线程的 ID、CPU 使用率和状态。

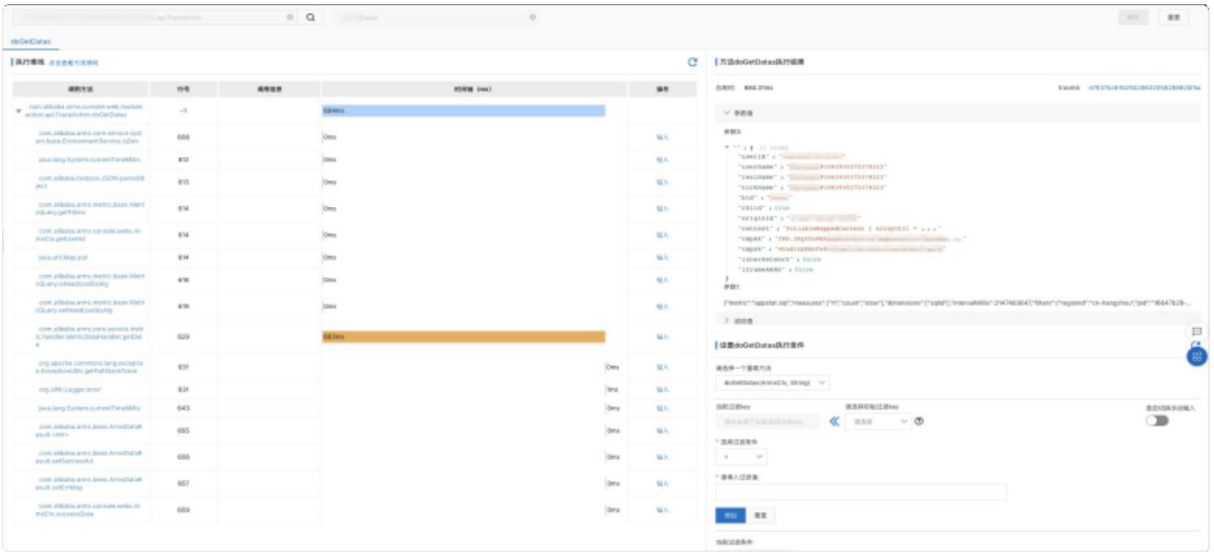


2.我们可以在目标线程右侧的操作列，单击查看实时堆栈。

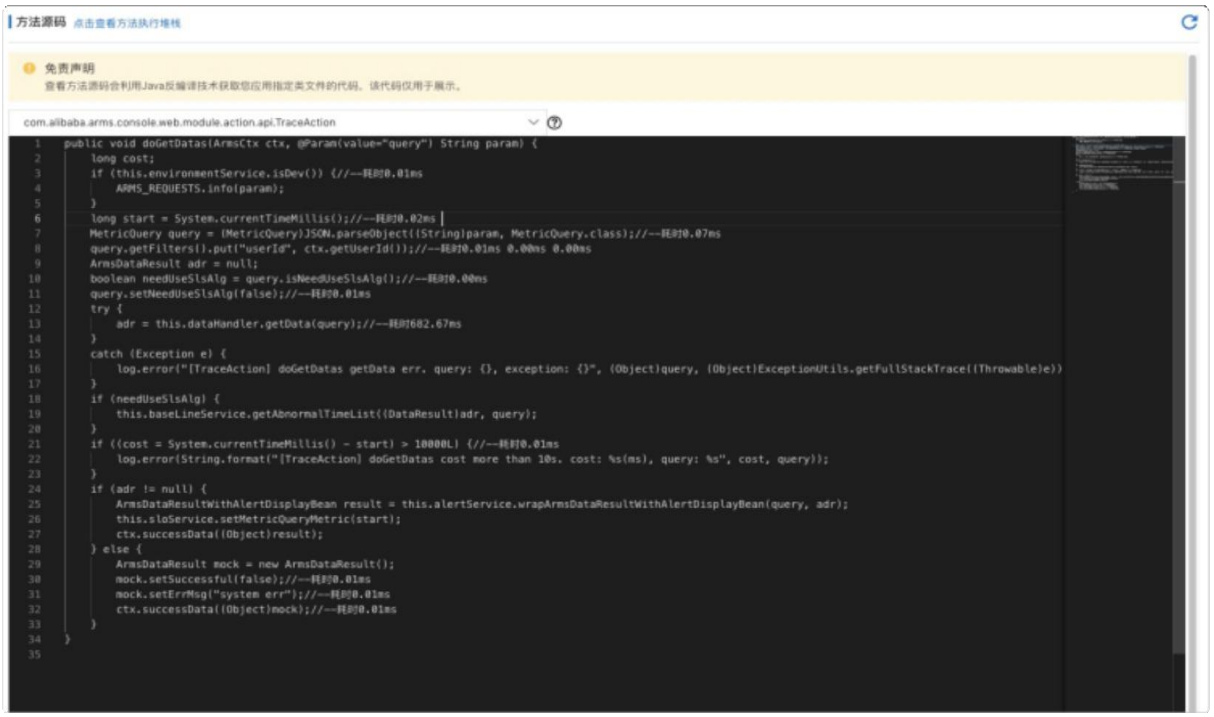


方法执行分析

方法执行分析支持抓取方法的某一次执行的耗时、入参、返回值等信息和钻入，帮助您快速定位导致慢调用的根本原因，以及问题线下无法复现或日志缺失等场景。

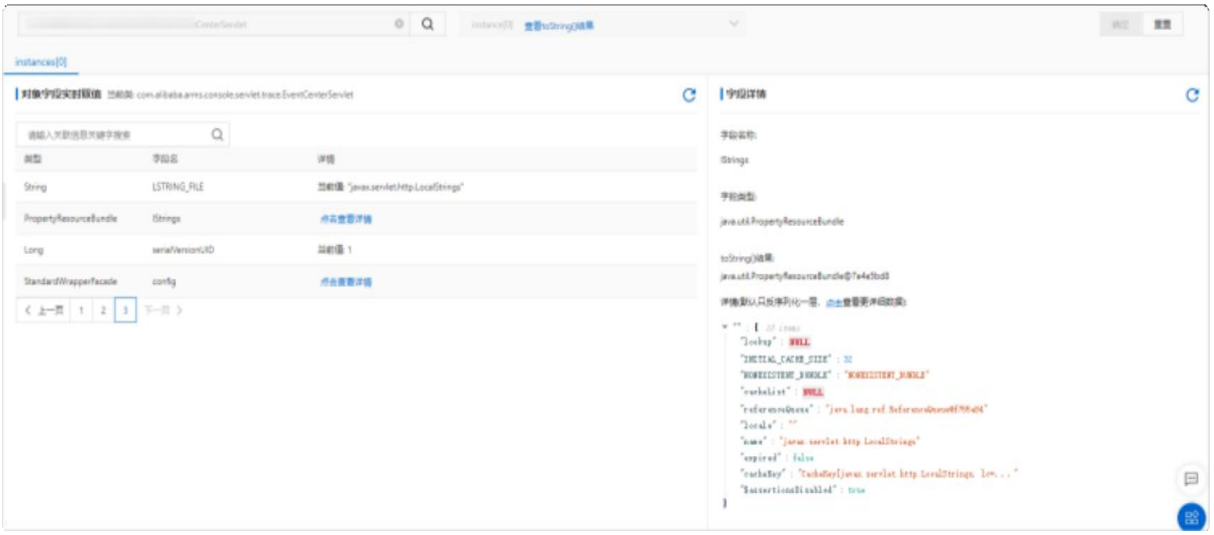


- 如下图所示，每一次内部方法的执行耗时都会以注释的方式显示在源代码中。



对象查看器

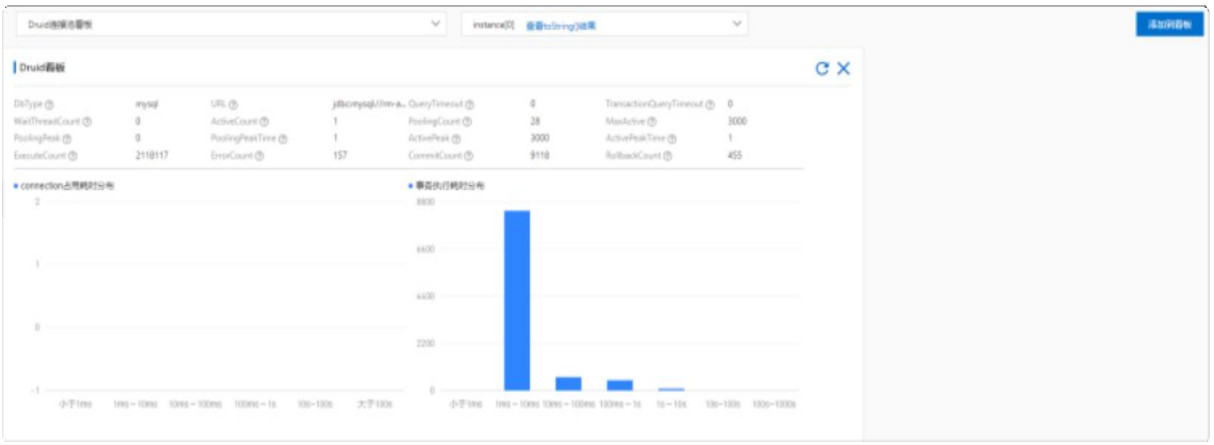
对象查看器用于查看一些单例对象当前的状态，用于排查应用状态异常问题，例如应用配置、黑白名单、成员变量等。



实时看板

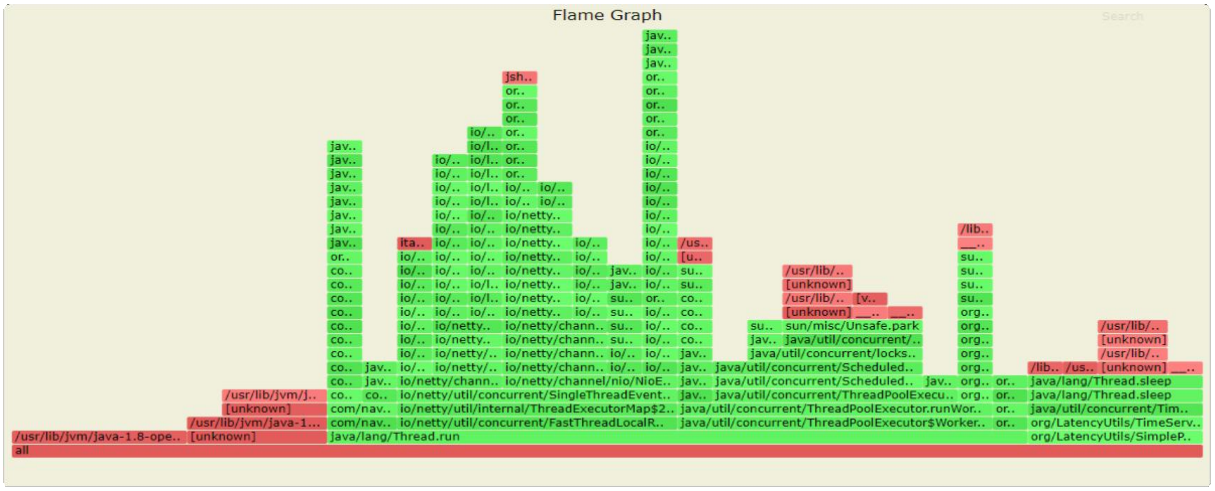
实时看板用于查看系统中用到的关键组件的实时状态，例如查看数据库连接池的使用情况、HTTP 连接池的使用情况等，有利于排查资源类型的问题。

1.如下图显示为一个 Druid 连接池的实时状态信息，包括基础配置、连接池状态、执行耗时分布等。



性能分析

性能分析支持对 CPU 耗时、内存分配等对象进行一定时间的采样并生成相应的火焰图，帮助您快速定位应用的性能瓶颈。



内存快照

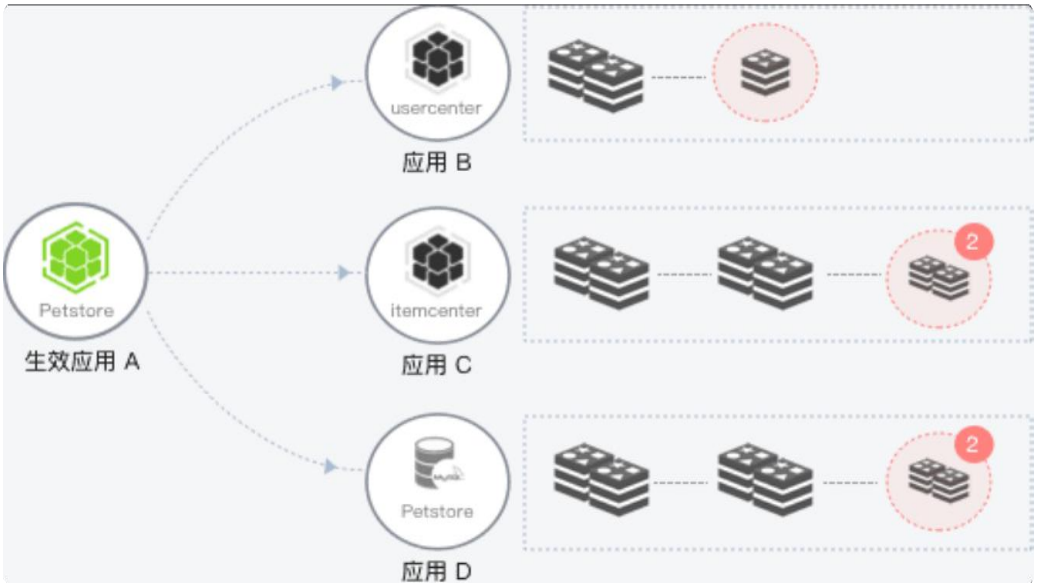
JVM 监控可以直观展示指定时间段内的多项内存指标，虽然图表能体现出内存使用量过大的情况，但无法显示具体信息，因此不能帮助您排查问题产生的原因。我们可以通过创建内存快照，通过详细的日志查看内存占用的详细信息，从而进一步排查内存泄漏和内存浪费等内存问题。



10 分钟恢复

离群实例摘除

在微服务架构中，当服务提供者的应用的某些实例出现异常，而服务消费者无法感知时会影响服务的正常调用，并影响消费者的服务性能甚至可用性。离群实例摘除功能会检测应用实例的可用性并进行动态调整，以保证服务成功调用，从而提升业务的稳定性和服务质量。



服务降级

当应用遇到业务高峰期，发现下游的服务提供者遇到性能瓶颈，甚至即将影响业务时。我们可以对部分的服务消费者进行服务降级操作，让不重要的业务方不进行真实地调用，直接返回 Mock 的结果或甚至异常返回，将宝贵的下游服务提供者资源保留给重要的业务调用方使用，从而提升整体服务的稳定性。我们把这个过程叫做：服务降级。当应用依赖的下游服务出现不可用的情况，导致业务流量损失。您可以通过配置服务降级能力，当下游服务出现异常时，服务降级使流量可以在调用端 "fail fast"，有效防止雪崩。

离群实例摘除与服务降级主要是体现在两点：

1.自动完成：服务降级是一种运维动作，需要通过控制台进行配置，并且指定对应的服务名才能做到相应的效果；而离群实例摘除能力是会主动探测上游节点的存活情况，在这条链路上整体做降级。

2.摘除粒度：服务降级降级的是（服务+节点 IP），以 Dubbo 举例子，一个进程会发布以服务接口名（Interface）为服务名的微服务，如果触发到这个服务的降级，下次将不再调用这个节点的此服务，但是还是会调用其他服务。但是离群实例摘除是整个节点都不会去尝试调用。

多活容灾

多活容灾 MSHA（Multi-Site High Availability），是在阿里巴巴电商业务环境演进出来的多活容灾架构解决方案，可以将业务恢复和故障恢复解耦，有基于灵活的规则调度、跨域跨云管控、数据保护等能力，保障故障场景下的业务快速恢复，助力企业的容灾稳定性建设。多活，顾名思义就是分布在多个站点同时对外提供服务。与传统的灾备的最主要区别就是多活里的所有站点同时在对外提供服务，不仅解决了容灾本身问题，还提升了业务连续性，并且实现了容量的扩展。

多活容灾解决的问题

- 故障快速恢复秉承先恢复，再定位的原则，MSHA 在各种灾难场景下均具备快速恢复业务的能力在数据保护的前提下让业务恢复时间和故障恢复时间解耦合，保障业务连续性。

- 容量异地扩展业务高速发展，受限于单地有限资源，也存在数据库瓶颈等问题。在 MSHA水平拓展能力支撑下，业务具备其它机房或者其它地域快速扩建的特性，减少成本浪费。

- 新技术试验田MSHA 本质上是提供了自上而下的一种流量隔离能力，在最小隔离单元内，业务可灵活进行风险可控的技术演进，例如基础设施升级、新技术验证等，甚至可以驱动在商业上应用。

- 爆炸半径可控基于单元间隔离能力，故障爆炸半径可控制在一个单元格内。

- 性能快捷优化 MSHA 流量在各单元自闭环，可有效降低 RT。

限流、扩容、重启、回滚

- 限流：根据流量、并发线程数、响应时间等指标，把随机到来的流量调整成合适的形状，即流量塑形。避免应用被瞬时的流量高峰冲垮，从而保障应用的高可用性。

- 扩容：水平横向扩容提升集群可用性。
- 重启：重新启动 JVM 进程，从而暂时消除长时间运行累积的问题如内存泄露等。
- 回滚：消除变更引入的问题。

1-5-10 故障快恢，故障 1 分钟响应、5 分钟定位、10 分钟恢复；只有不断地面向失败地设计、基于故障应急方式演练，那么在真正遇到线上故障的时候我们才可以更加从容地面对故障。我们希望新一代的云原生微服务能更多地具备系统自愈能力，微服务架构内部可以自动感知外部组件的失效，自动切换至备用链路，真正地把故障扼杀在摇篮之中。

3.10 微服务注册发现高可用解决方案

背景

注册中心作为承担服务注册发现的核心组件，是微服务架构中必不可少的一环。在 CAP 的模型中，注册中心可以牺牲一点点数据一致性（C），即同一时刻每一个节点拿到的服务地址允许短暂的不一致，但必须要保证的是可用性（A）。因为一旦由于某些问题导致注册中心不可用，或者服务连不上注册中心，那么想要连接它的节点可能会因为无法获取服务地址而对整个系统出现灾难性的打击。

一个真实的案例

本文从一个真实的案例说起，某客户在阿里云上使用K8s 集群部署了许多自己的微服务，由于某台 ECS 的网卡发生了异常，虽然网卡异常很快恢复了，但是却出现了大面积持续的服务不可用，业务受损。

我们来看一下这个问题链是如何形成的？

1.ECS 故障节点上运行着 K8s 集群的核心基础组件 CoreDNS 的所有 Pod，且低版本 K8s 集群缺少 NodeLocal DNSCache 的特性，导致集群 DNS 解析出现问题。

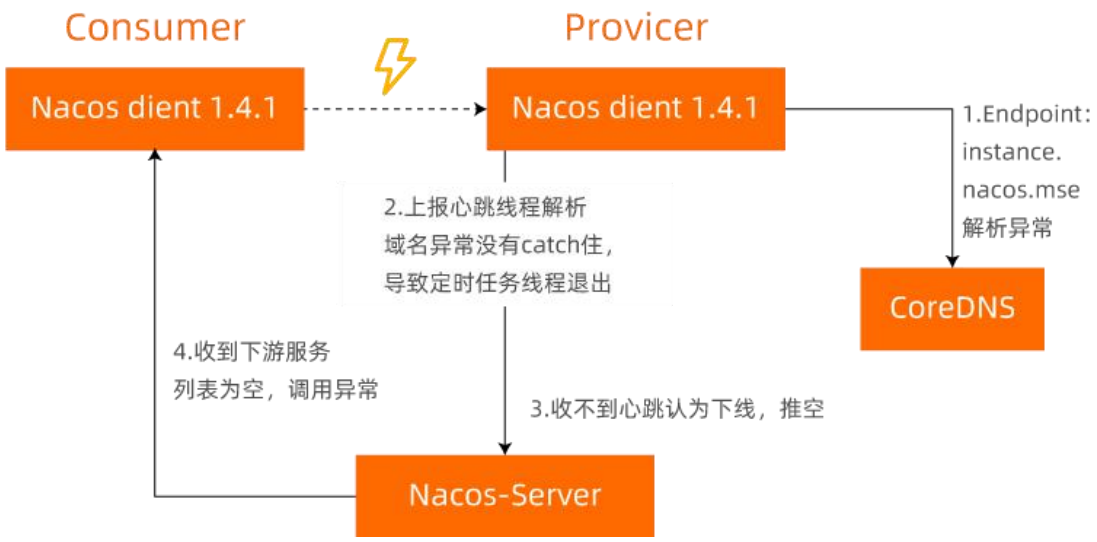
2.该客户的服务发现使用了有缺陷的客户端版本（nacos-client 的 1.4.1 版本），这个版本的缺陷就是跟 DNS 有关——心跳请求在域名解析失败后，会导致进程后续不会再续约心跳，只有重启才能恢复。

3.这个缺陷版本实际上是已知问题，阿里云在 5月份推送了 nacos-client 1.4.1 存在严重 bug 的公告，但客户研发未收到通知，进而在生产环境中使用了这个版本。



风险环环相扣，缺一不可。

最终导致故障的原因是服务无法调用下游，可用性降低，业务受损。下图示意的是客户端缺陷导致问题的根因：



1.Provider 客户端在心跳续约时发生DNS 异常；

2.心跳线程未能正确地处理这个 DNS 异常，导致线程意外退出了；

3.注册中心的正常机制是，心跳不续约，30 秒后自动下线。由于 CoreDNS 影响的是整个 K8s 集群的 DNS 解析，所以 Provider 的所有实例都遇到相同的问题，整个服务所有实例都被下线；

4.在 Consumer 这一侧，收到推送的空列表后，无法找到下游，那么调用它的上游（比如网关）就会发生异常。

回顾整个案例，每一环每个风险看起来发生概率都很小，但是一旦发生就会造成恶劣的影响。服务发现高可用是微服务体系中很重要的一环，当然也是我们时常忽略的点。在阿里内部的故障演练中，这一直是必不可少的一个环节。

面向失败的设计

由于网络环境的抖动比如 CoreDns 的异常，或者是由于某些因素导致我们的注册中心不可用等情况，经常会出现服务批量闪断的情况，但这种情况其实不是业务服务的不可用，如果我们的微服务可以识别到这是一种异常情况（批量闪断或地址变空时），应该采取一种保守的策略，以免误推从而导致全部服务出现"no provider"的问题，会导致所有的微服务不可用的故障，并

且持续较长时间难以恢复。

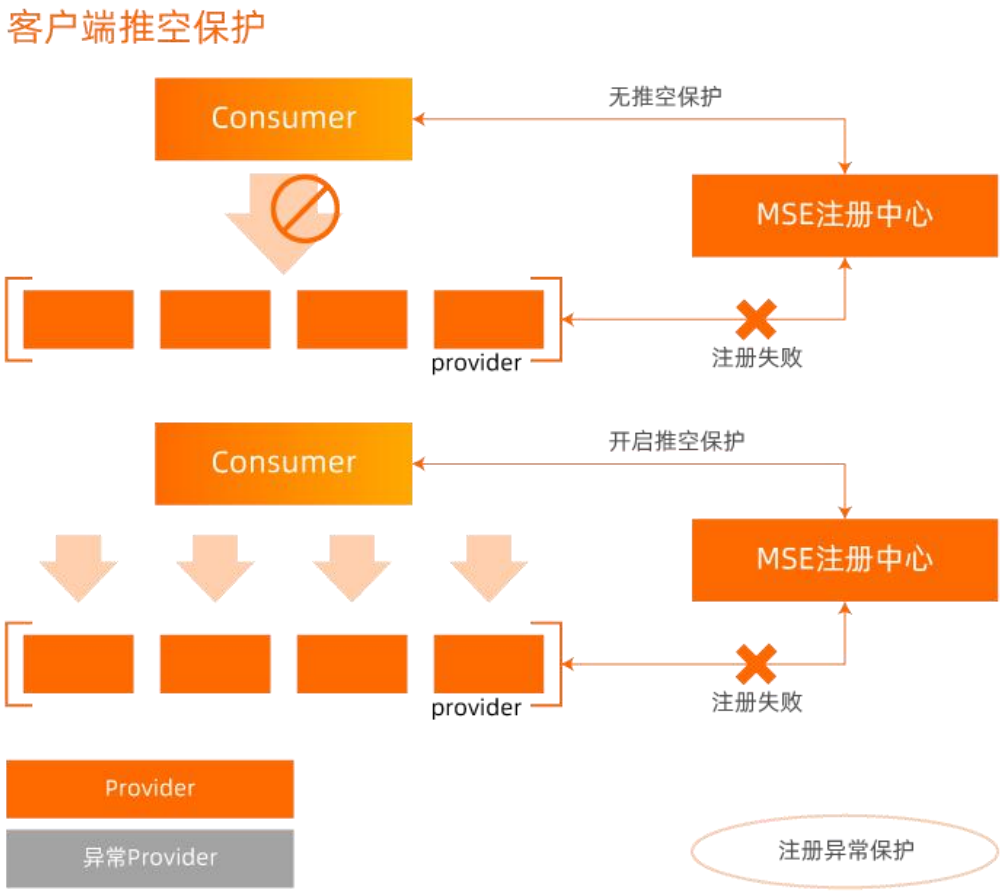
站在微服务角度上考虑，我们如何可以切段以上的问题链呢？以上的案例看起来是 Nacos-client 低版本 造成的问题，但是如果我们用的是 zookeeper、eureka 等注册中心呢？我们能拍着胸脯说，不会发生以上的问题吗？面向失败的设计原则告诉我们，如果注册中心挂掉了，或者我们的服务连不上注册中心了，我们需要有一个方式保证我们的服务正常调用，线上的业务持续不断。

本文介绍的是服务发现过程中的高可用的机制，从服务框架层面思考如何彻底解决以上的问题。

服务发现过程中的高可用原理解析

服务发现高可用 -- 推空保护

面向失败的设计告诉我们，服务并不能完全相信注册中心的通知的地址，当注册中心的推送地址为空时候，服务调用肯定会出 no provider 错误，那么我们就忽略此次推送的地址变更。



微服务治理中心提供推空保护能力：

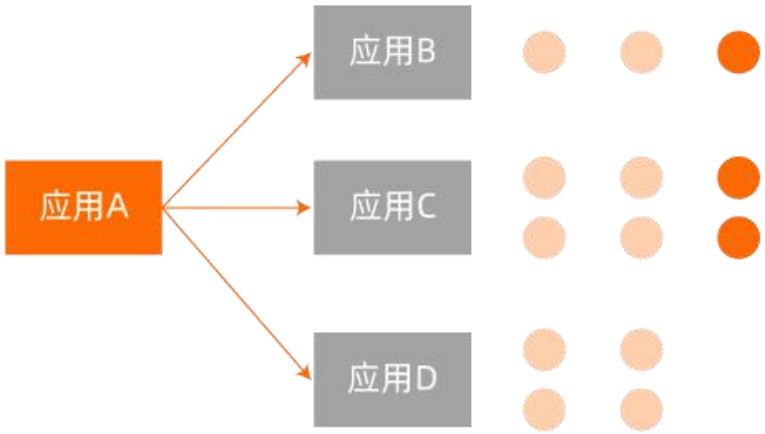
- 默认无侵入支持市面上近五年来的 Spring Cloud 与 Dubbo 框架。
- 无关注册中心实现，无需升级 client 版本。

服务发现高可用 -- 离群实例摘除

心跳续约是注册中心感知实例可用性的基本途径。但是在特定情况下，心跳存续并不能完全等同于服务可用。

因为仍然存在心跳正常，但服务不可用的情况，例如：

- Request 处理的线程池满
- 依赖的 RDS 连接异常或慢 SQL



此时服务并不能完全相信注册中心的通知的地址，推送的地址中，可能存在一些服务质量低下的服务提供者，因此客户端需要自己根据调用的结果来判断服务地址的可用性与提供服务质量的好坏，来定向忽略某些地址。

微服务治理中心提供离群实例摘除：

- **基于异常检测的摘除策略：包含网络异常和网络异常 + 业务异常（HTTP 5xx）。**
- **设置异常阈值、QPS 下限、摘除比例下限。**
- **摘除事件通知、钉钉群告警。**

离群实例摘除的能力是一个补充，根据特定接口的调用异常特征，来衡量服务的可用性。

尾

没有任何系统是百分百没有问题的，风险是无处不在的，尽管有很多发生概率很小很小，却都无法完全避免，所以面对失败（风险）的设计是必不可少的。

服务发现的高可用是我们时常容易忽视的点，但是它又是非常关键的点，一旦我们的系统出现大面积服务发现的问题，并且由于微服务依赖的复杂度，导致相关的问题也很难快速恢复。为了避免对整个系统出现灾难性的打击，我们需要对服务发现进行面向失败的设计与演练，才能做到心中有数。

3.11 微服务应用安全解决方案

为什么需要微服务安全

层出不穷的基础组件的安全漏洞需要及时修复

随着微服务的深入，微服务的安全问题日益成为一个企业关注的重点，微服务所依赖的基础框架，操作系统内核，如果出现安全漏洞，随时可能成为一个深水炸弹，对业务系统造成灾难性的破坏。比较典型的例子是，层出不穷的 fastjson 安全漏洞，Dubbo 安全漏洞，以及近期破坏性较大的 log4j 安全漏洞，给企业客户的生产，业务迭代造成了巨大的影响，所有依赖该基础组件的二方包，三方包，都需要通过升级 SDK 的方式修复，微服务规模越大，升级的成本越高。

- 2021 年 12月10日，国家信息安全漏洞共享平台（CNVD）收录了 Apache Log4j2 远程代码执行漏洞（CNVD-2021-95914），此漏洞是一个基于 Java 的日志记录工具，为 Log4j 的升级。作为目前最优秀的 Java日志框架之一，被大量用于业务系统开发。此次危机由 Lookup 功能引发，Log4j2 在默认情况下会开启 Lookup 功能，提供给客户另一种添加特殊值到日志中的方式。此功能中也包含了对 于 JNDI 的 Lookup，但由于 Lookup 对于加载的 JNDI 内容未做任何限制，使得攻击者可以通过 JNDI 注入实现远程加载恶意类到应用中，从而造成 RCE。

- 2021 年 7月1日，Apache Dubbo 社区爆出如下安全漏洞列表：CVE-2021-25641，等级【高】。Hessian2 协议反序列化漏洞，攻击者以篡改协议的方式绕过反序列化黑/白名单；CVE-2021- 30179，等级【高】。Generic Filter 远程代码执行漏洞，启用泛化调用的用户，可能会受到恶意伪造的参数的攻击；CVE-2021-32824，等级【高】。Telnet handler 远程代码执行漏洞，恶意攻击者可能通过 Telnet 接口构造相关请求进行攻击；CVE-2021-30180，等级【低】。YAML 规则加载造成的远程代码执行漏洞，攻击者在攻破配置中心后，可通过上传恶意的 yaml 规则等触发反序列化漏洞；CVE-2021-30181，等级【低】。Nashorn 脚本远程代码执行漏洞，攻击者在攻破配置中心后，可通过上传恶意的 script 规则等来触发 Nashorn 脚本执行攻击。

微服务之间的调用需要安全可信

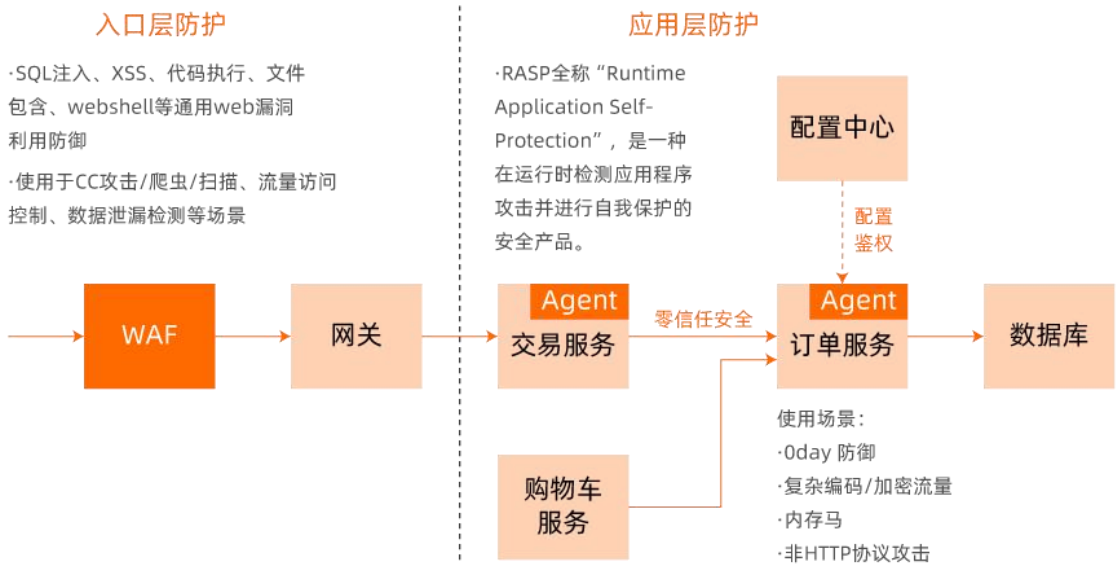
我们对于数据库通常都会做严格的权限控制，但是由于我们的微服务对数据库是拥有完全的访问权限的，所以即使数据库层面做了非常严格的权限控制，一旦微服务层面突破了，也会对数据库造成灾难性的破坏，例如一个黑客，假设具备了微服务的访问权限，可能会发生拖库等严重问题。因此微服务之间的调用也需要严格的而控制安全可信。

业务配置安全

我们对于一些重要的配置，通常会放在统一的配置中心，这里面就包含了一些敏感数据，例如数据库的访问用户名和密码，然后通常的配置中心的配置，大家是可以随意访问的，如果让无权访问改数据库的用户拿到了这些敏感数据，则他也可能会获取到数据库中的敏感数据。因此业务配置也需要进行安全的授权访问。

微服务安全解决方案

漏洞防护



通常我们在入口处通过 WAF 提供入口层的防护，可以提供 SQL 注入、XSS、代码执行、文件包含、webshell 等通用 web 漏洞利用防御能力，入口层安全防护更加适用于 CC 攻击/爬虫/扫描、流量访问控制、数据泄露检测等场景，但入口层防护也有一定的局限性，入口层防护完

全基于流量特征进行检测，容易产生大量无效报警或因担心误报规则不敢做太严格，这给安全运维会带来一些负担，某用户在在线文档中上传一段 SQL 语句，容易产生误报；再如，利用 php 漏洞的报文打到了 Java 环境中，实际 Java 应用不受影响，这也容易产生虚报。

再者，基于流量特征的防护只能看到流量内容，即用户的原始请求，并不能感知应用最终会怎样执行这条请求，有一些比较隐蔽的攻击，可能会通过变形的请求绕过流量特征的检测。

在应用层我们推荐使用应用层防护 RASP 技术来进行防护，RASP 全称 Runtime Application Self Protection，是一种在运行时检测应用程序攻击并进行自我保护的安全产品。RASP 通过 Java Agent 方式挂载到微服务系统中，无需修改任何业务代码，对业务侵入性较低。RASP 能看到应用的上下文，理解应用最终执行了什么动作和命令，不管原始请求怎样变形，最终应用执行的动作是不变的，例如要执行 `cat /etc/passwd` 命令，无论流量特征如何变形，最后都会落到这个执行动作上。RASP 还能理解是什么应用做了什么动作（身份+行为），只要身份和行为不匹配，就可以检测到异常。对于一些加密马等手段，本质上也是对输入内容做变形以绕过基于特征的检测，RASP 同理也能都抵抗。

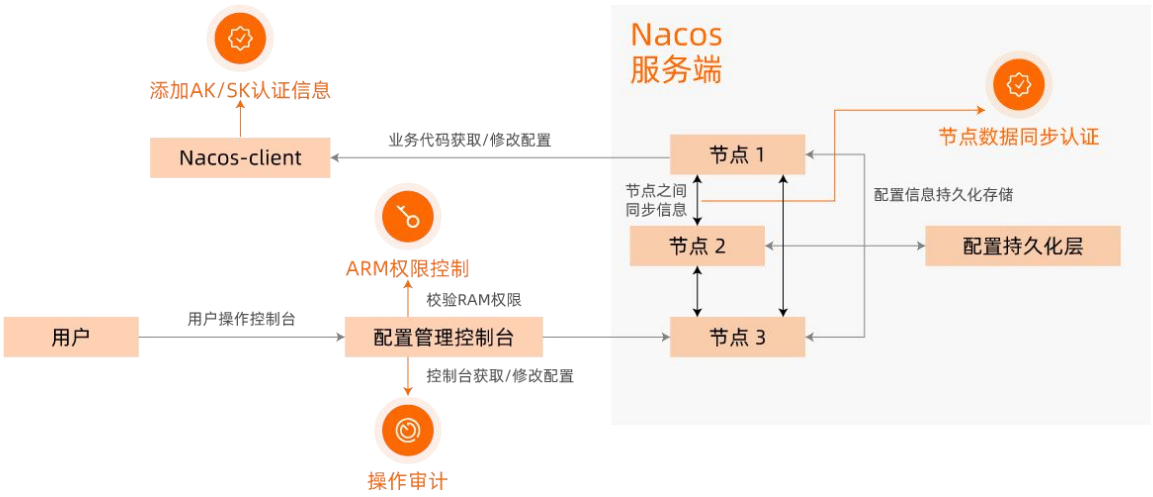
零信任安全

针对 Java 微服务应用，基于字节码增强的方案，提供无侵入的 0 信任方案，接入统一治理平台后，每个微服务的调用方都具有身份，微服务调用过程中，每次请求都会带上该应用的身份，而通过服务治理平台，可以灵活控制针对每个应用配置访问策略：

- 从访问方式上，可以通过黑白名单的方式来进行配置。例如，可以配置只允许来自某些特定应用的 请求访问自身提供的接口，通常我们称为白名单，也可以配置除了某些应用，其他应用都可以访问，通常我们成为黑名单。

- 从访问粒度上，针对 Spring Cloud 类型的微服务，可以控制访问微服务的某一个具体的 URL，针对 Dubbo 类型的接口维度的微服务框架，可以支持控制访问微服务的某一个具体的接口。

配置鉴权



在配置中心的鉴权方面，提供了三重防护方案，本文以 Nacos 配置中心为例：

- 在客户端，开源的 nacos-client 支持通过 AK/SK 方式进行配置的访问，针对开启配置鉴权的配置中心，如果以未授权的 AK/SK 或者没有带上 AK/SK 进行方案，则服务器端会直接拒绝该请求。

- 用户在配置管理控制台，提供基于阿里云 RAM 的权限控制台策略，配置中心鉴权功能可以按照实例、命名空间、group、dataId 设置访问权限，降低某个实例被恶意用户非法获取、修改的风险；同时，所有的操作都提供了审计能力，方便进行溯源，查询历史的操作等动作。

- 针对 Nacos 服务器，节点之间信息同步，我们也添加了认证机制，防止未经授权的数据在集群之间同步。

3.12 异构微服务互通解决方案

背景介绍

什么是异构微服务

异构微服务本质上指的是不同类型的微服务框架，比如当下比较火热的主流微服务框架 Spring Cloud、Dubbo 以及 Istio 服务网格等等，这些微服务框架对开发者屏蔽了底层的细节，为微服务应用提供了服务发现、负载均衡等服务治理的解决方案。

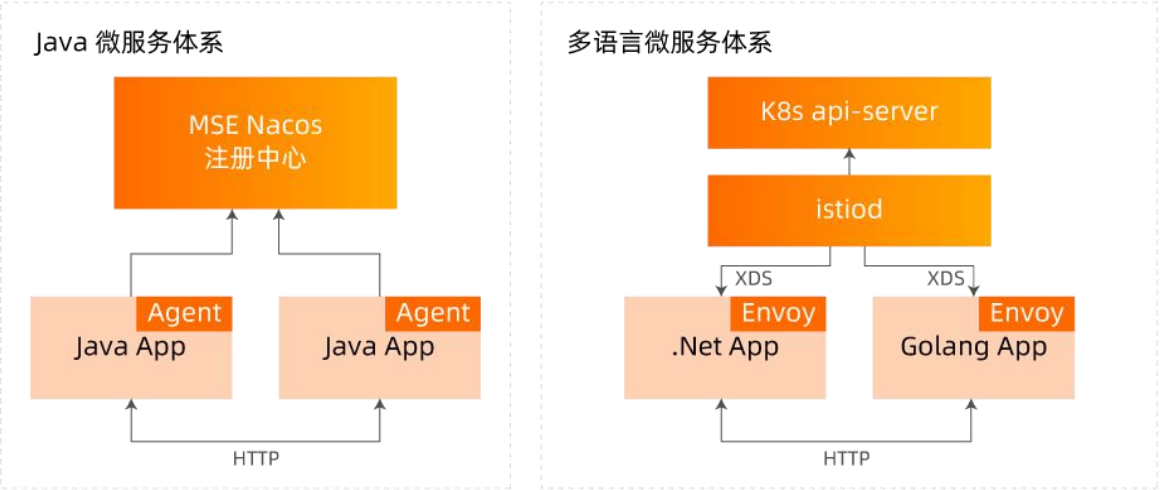
一个企业内部所开发的微服务有可能都是基于同一个微服务框架完成的，对于这样的架构，我们称之为是同构微服务体系；当然也有可能一个企业内的微服务是基于多种不同的微服务框架建立的，这样的架构我们称之为是异构的微服务体系。多种不同类型微服务框架的共存在大型企业内还是比较普遍的，形成这种现象的原因有很多，比如：可能是历史遗留、难以改造的系统；也可能是企业正在做技术栈迁移；又或者是企业内不同业务部门为了满足各自的特殊需求而做的独立选型。这也就意味着异构微服务 体系很有可能需要长期共存。

本小节中的异构微服务特指以 SpringCloud 框架为代表的 Java 微服务体系，和以 istio 服务网格为载体的多语言微服务体系。

为什么需要异构微服务互通

不同类型的微服务框架往往使用了不同的通信协议、进行服务发现的方式也不尽相同。比如说，SpringCloud 框架直接基于 HTTP 协议进行通信，而 Dubbo 则使用自定义报文的 TCP 协议，并且序列化使用定制 Hessian2 框架，二者进行服务发现的方式一般是对接 Nacos、Eureka、Zookeeper 等注册中心；而对于 istio 服务网格来说，服务间通信一般基于 HTTP 协议，服务发现方式则是由 Envoy 通过 XDS 协议向控制面进程 Pilot 获取服务信息。面对这种场景，要如何透明地实现异构微服务体系之间的服务发现和服务调用？如果我们什么都不做，那么每个微服务体系就只能感知到自己所在体系内的微服务的状态，流量也只能在各自的体系内封闭流转。

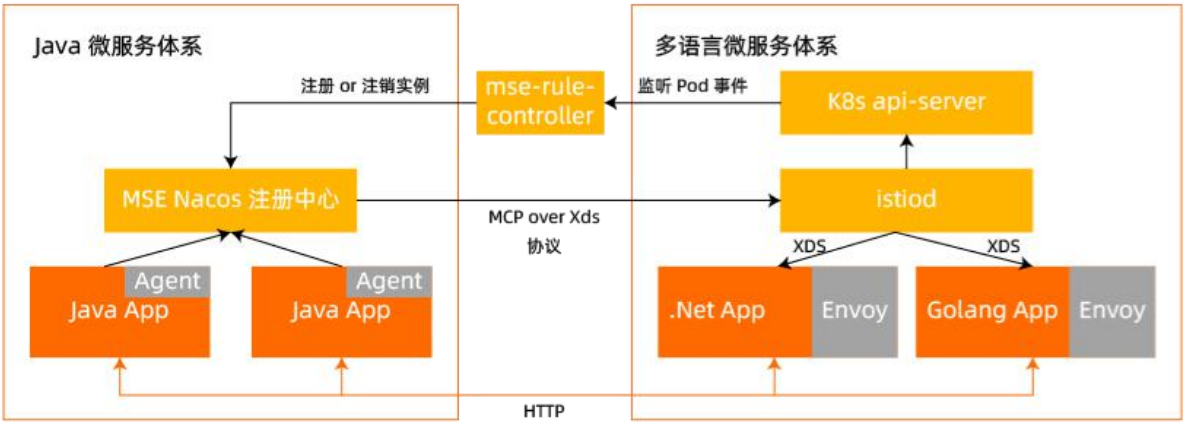
以下图为例，左侧展示的是 SpringCloud Java 微服务体系，右侧展示的是 Istio 多语言微服务体系。对于 Java 微服务体系而言，服务提供者将自身的服务信息注册到 Nacos 注册中心，然后服务消费者从注册中心进行订阅，根据获取到的提供者信息完成 HTTP 服务调用；对于 Istio 多语言微服务体系而言，控制面进程 Pilot（即图中的 istiod）从注册源获取服务信息，然后通过 XDS 协议下发给数据面的 Envoy，istio 默认的注册源是 Kubernetes 集群，使用的是 Kubernetes 的服务发现机制。两个微服务体系下的应用是无法做到相互访问的，因为对于 java 应用来说，注册中心中并不包含多语言应用的相关信息，消费者无法完成服务发现，而对于多语言应用来说，首先 Java 应用并不一定部署在 Kubernetes 集群中，其次即使部署在 Kubernetes 集群中但极有可能没有创建对应的 Service，这就会导致多语言微服务体系下的服务发现机制无法对其生效。



因此，要做到从体系 A 平滑的迁移到体系 B，或者想要长期地保持企业内部多个微服务体系的共存，那么解决异构微服务体系间的互联互通，实现流量的透明调度将是非常重要的环节。

异构微服务互通解决方案

异构微服务互通解决方案示意图如下所示，该方案从两个方向解决了 Java 微服务体系和多语言微服务体系互相进行服务发现的问题，从而能够达到异构微服务体系的互联互通。



从 Java 微服务体系到多语言微服务体系

上文提到，Java 应用有可能没有部署在 Kubernetes 集群中或者没有创建对应的 Service，导致 Istio 的服务发现机制对其失效，实际上，Istio 还支持通过配置来添加额外的注册源进行服务发现。从 Istio 1.9 版本开始，官方移除了原有的 MCP 协议的相关代码，而全部转为 MCP-over-XDS 协议实现，第三方注册中心（比如 Nacos、Consul 等）需要自行实现基于此协议的 XDS 服务器，将自身存储的服务信息按照协议约定传输给 Istio 的控制面进程 Pilot，目前业界中，阿里巴巴的 Nacos 注册中心已经率先支持了 MCP-over-XDS 协议。在具备这样一个前提之后，只需要在 Istio 的配置中添加第三方注册中心的地址和端口即可完成对接，多语言微服务体系下的应用就可以通过 HTTP 协议进行服务调用。

从多语言微服务体系到 Java 微服务体系

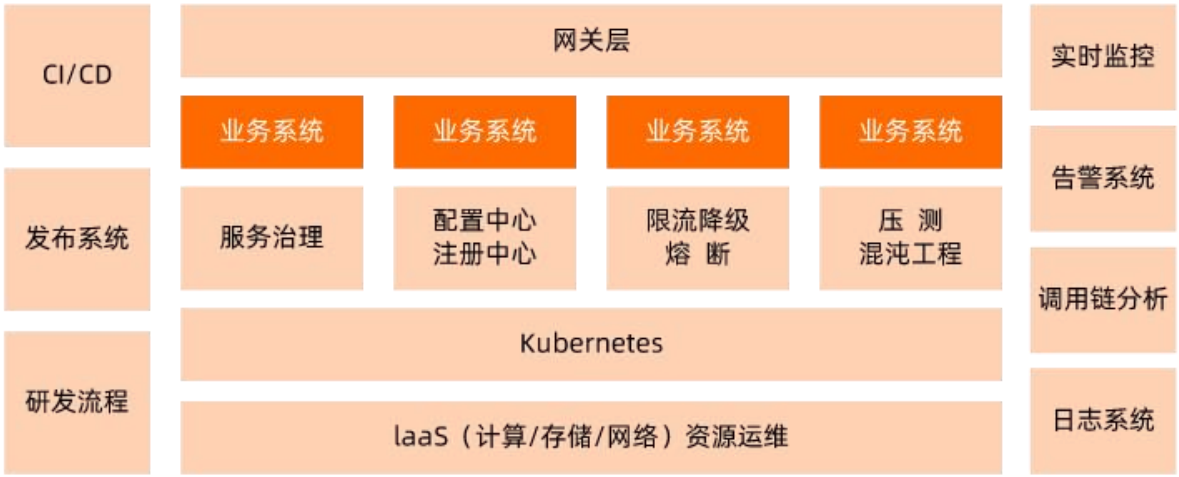
Java 微服务体系下的应用无法对多语言应用进行服务发现是因为注册中心中不包含对应的服务信息，其本质是多语言应用并没有向 Nacos 注册中心进行服务注册。所以要解决这个问题，只需要添加一个工具组件来辅助多语言应用进行服务注册即可，在该方案中，这部分功能由组件 mse-rule-controller 来承载，mse-rule-controller 是阿里云 MSE 中用于多语言微服务治理的一个组件。只不过在这个过程中，有一些需要额外注意的点，比如说，Java 微服务体系下的应用在进行负载均衡时，会参考服务提供者元数据中的某些字段进行选择，那么在 mse-rule-controller 对多语言应用进行辅助注册的过程中，就必须按照相同的约定生成服务元数据，我们解决异构微服务互联互通的目的并不单单是为了不同体系下的微服务能够互相进行调用，还希望将来能够以一种统一的方式进行服务治理。

3.13 微服务 Serverless PaaS 解决方案

背景

基于微服务架构带来了很多好处，包括解耦系统，降低爆炸半径，解耦团队，提升迭代效率等等。但是现实世界中没有银弹，根据“复杂性守恒定律”，我们解决了一部分复杂性，但是会有另一部分的复杂性，需要解决。

典型的，拆分成微服务以后，应用数增多了，迭代的频率也增加了，假设你有 3 套环境（测试、预发、生产），5 个应用，那么你实际需要维护 15 个应用，不同的规格，不同的实例数，不同的高可用等级、不同的弹性策略等等。如果你关注云原生，可能你会觉得需要容器，需要 kubernetes 等等。



或多或少，很多开发者都会发现企业里面都会有类似于上图的一套系统，但是得付出相应的维护成本，这就是微服务带来的复杂性的另一面，那么有没有更简单的解决方案呢？

微服务 Serverless PaaS 解决方案

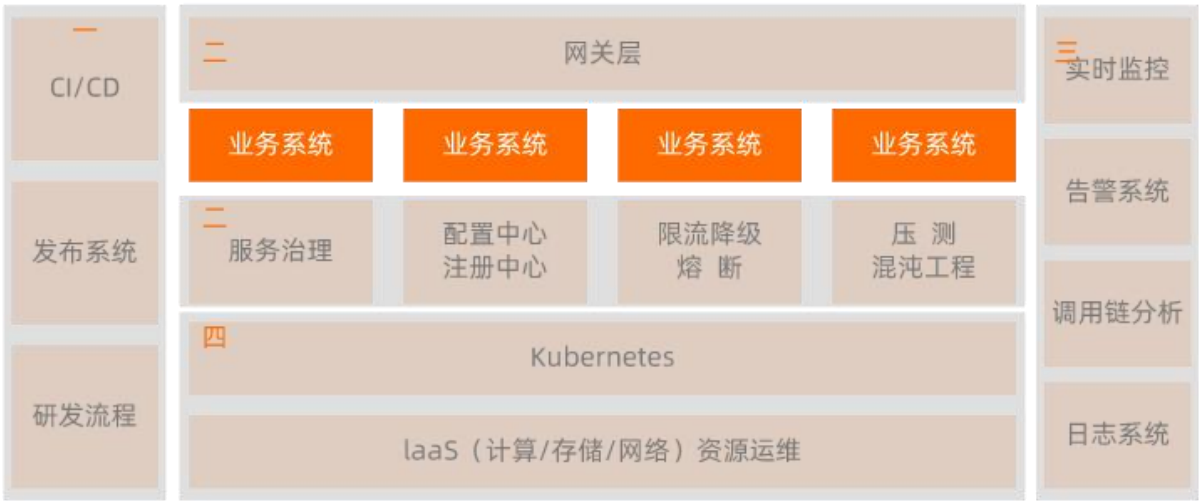
进击的 Serverless

提到简单，低负担，我想大家都会想到 Serverless，它的本质就是希望开发者不再需要去处理 Server 运维的复杂性，当然 Server 并没有消失，而是交给了云平台，这也就是 Less 的部分。

经过这么多年的发展，Serverless 也在不断地演进，"Less"的范围在不断扩大，不仅仅包含 Server 部分，还有 0 改造、可观测、弹性、应用托管、微服务、CI/CD 等等各个方面。结合我们前面的微服务架构下复杂性讨论，Serverless 其实是微服务架构下降低复杂度，减少维护成本的最好的一个解决方案之一。

Serverless PaaS 解决方案

在阿里云最新的 Serverless 体系中，已经不局限仅仅关注 “Server” 部分，而是围绕应用整体需要，开发或者融合各种 PaaS 能力，真正对开发者做到 “将复杂留给平台，简单留给客户” 。以下为阿里云 SAE 产品为例：



首先是第一部分，SAE 给用户提供了一个白屏化的界面，极大的降低用户的使用门槛，甚至可以说 0 门槛，它的交互符合大多数开发者心中 PaaS 的心智，还有 CLI、插件、OpenAPI 等等丰富的被集成能力。另外也有很多企业级特性，比如命名空间隔离，细粒度的权限控制等等，这些多试企业需要面对的实实在在的问题

第二部分，在微服务能力上，SAE 通过注入 Agent，整体上实现了一个无侵入，业务无感的解决方案，提供了丰富的微服务治理能力，包括无损下线，灰度发布、链路跟踪等等。

第三部分，在微服务架构下，应用数量较多，定位问题困难，可观测就需要具备非常高的要求，SAE 结合阿里云的 ARMS、云监控、SLS、Prometheus 等产品，在 Metrices、Tracing、Logging 等方面都提供了相对完整的解决方案，切实解决开发者在可观测方面的痛点，包括基础监控、调用链、实时日志、事件等等

第四部分，SAE 不仅仅让开发者不需要运维 IaaS，而且也不需要运维 kubernetes，免去了开发者运维 kubernetes 的苦恼，同时又可以利用 kubernetes 的各种能力，包括健康检查、弹性等等。

提到云原生，大家都会想到容器、微服务、不可变基础设施等等技术，但是我们回到为什么会出现云原生，本质也是希望能借助云原生相关的技术构建出“容错性好、易于管理和便于观察的松耦合系统”，所以它也是为了降低复杂性和门槛。相信微服务和 Serverless PaaS 结合的解决方案是实现这个目标的最佳手段之一

第四章：基于 MSE 的微服务治理最佳实践

本章基本都是使用一组包含了 spring-cloud-zuul、spring-cloud-a、spring-cloud-b、spring-cloud-c 的 Demo 来演示微服务治理的最佳实践。这些应用都是基于 Spring Cloud 和 Dubbo 框架的标准用法开发的，您可以直接在：https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/mse-simple-demo 项目上查看源码。

本章中主要使用阿里云容器服务 ACK 进行服务的部署，部署到 K8s 中的 yaml 文件可以在这里找到：https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/microservices-materials/white-paper

4.1 线上发布稳定性解决方案最佳实践

本文是阿里云微服务引擎 MSE 在应用发布时提供的无损上下线和服务预热能力最佳实践介绍。假设应用的架构由 Zuul网关以及后端的微服务应用实例（Spring Cloud）构成。具体的后端调用链路有购物车应用A，交易中心应用B，库存中心应用C，这些应用中的服务之间通过 Nacos 注册中心实现服务注册与发现。

前提条件

开启 MSE 微服务治理

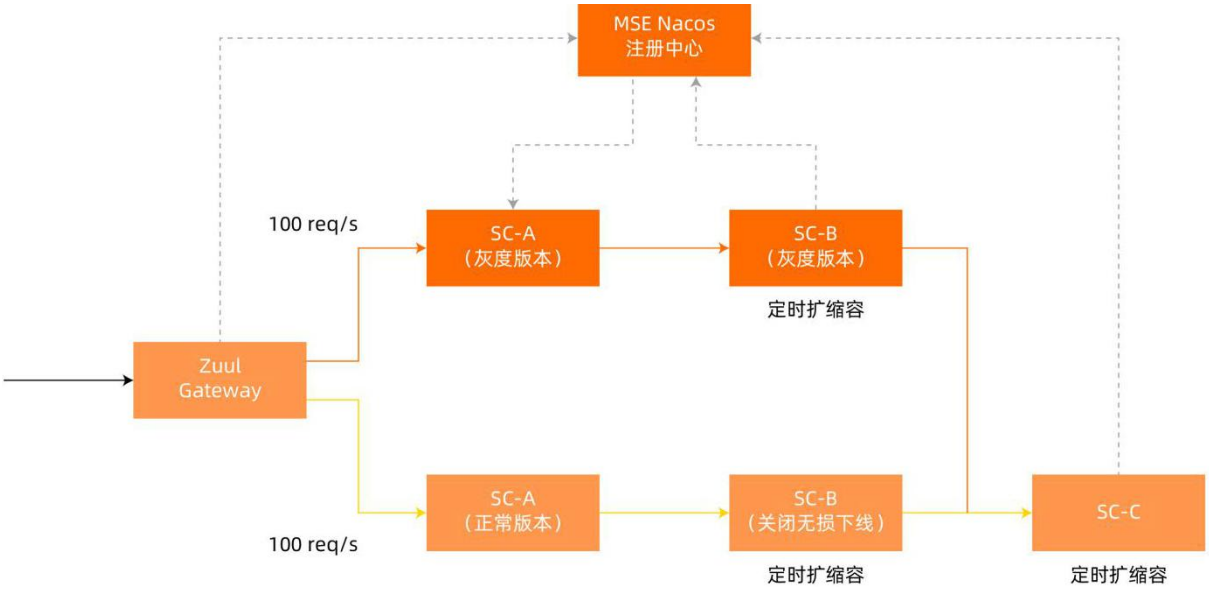
- 已创建 Kubernetes 集群，请参见创建 Kubernetes 托管版集群。
- 已开通 MSE 微服务治理专业版，请参见开通 MSE 微服务治理。

背景信息

很多用户量大并发度高的应用系统为了避免发布过程中的流量有损一般选择在流量较小的半夜发布，虽然这样做有效果，但不可控导致背后的研发运维人员经常因为发布问题时常搞得半夜胆战心惊，心力疲惫。基于此，阿里云微服务引擎 MSE 通过在应用发布过程中，通过应用下线主动实时注销，应用上线健康就绪检查与生命周期对齐以及服务预热等技术手段所提供的微服务应用无损上下线发布功能，让研发运维人员即使是在白天发布应用，也能风轻云淡。

准备工作

应用部署流量架构图



流量压力来源

在 spring-cloud-zuul 应用中，每个 pod 具备并发为 10 的访问本地 zuul 端口的 127.0.0.1:2 0000:/A/a 的 http 请求流量。可以通过环境变量 demo.qps 配置并发数。

部署 Demo 应用程序

将下面的内容保存到一个文件中，假设取名为 mse-demo.yaml ，并执行 kubectl apply -fms e-demo.yaml 以部署应用到提前创建好的 Kubernetes 集群中（注意因为 demo 中有 CronHPA 任务， 所以请先在集群中安装 ack-kubernetes-cronhpa-controller 组件，具体在容器服务- Kubernetes->市场->应用目录中搜索组件在测试集群中进行安装），这里我们将要部署 Zuul，A，B 和 C 三个应用，其中 A、B 两个应用分别部署一个基线版本和一个灰度版本，B 应用的基线版本关闭了无损下线能力，灰度版本开启了无损下线能力。C 应用开启了服务预热能力，其中预热时长为 120 秒。

```
# Nacos Server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nacos-server
```

```
  name: nacos-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      annotations:
        msePilotCreateAppName: nacos-server
        msePilotAutoEnable: "on"
      labels:
        app: nacos-server
    spec:
      containers:
      - env:
        - name: MODE
          value: standalone
        image: registry.cn-shanghai.aliyuncs.com/yizhan/nacos-server:latest
        imagePullPolicy: Always
        name: nacos-server
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
      dnsPolicy: ClusterFirst
      restartPolicy: Always

# Nacos Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
```

```
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
  selector:
    app: nacos-server
  type: ClusterIP

#入口 zuul 应用
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-zuul
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-cloud-zuul
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-zuul
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-zuul
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: LANG
              value: C.UTF-8
```

```
            image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-zuul:1.0.1
            imagePullPolicy: Always
            name: spring-cloud-zuul
            ports:
              - containerPort: 20000

# A 应用 base 版本,开启按照机器纬度全链路透传
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-a
  name: spring-cloud-a
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-a
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: profiler.micro.service.tag.trace.enable
```

```
            value: "true"
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a
          ports:
          - containerPort: 20001
            protocol: TCP
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30

# A 应用 gray 版本,开启按照机器纬度全链路透传
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-a-gray
  name: spring-cloud-a-gray
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a-gray
  strategy:
  template:
    metadata:
      annotations:
```

```
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-a
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-a-gray
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: profiler.micro.service.tag.trace.enable
          value: "true"
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a-gray
        ports:
        - containerPort: 20001
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
        livenessProbe:
          tcpSocket:
            port: 20001
          initialDelaySeconds: 10
          periodSeconds: 30

# B 应用 base 版本，关闭无损下线能力
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  labels:
    app: spring-cloud-b
  name: spring-cloud-b
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b
  strategy:
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-b
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: micro.service.shutdown.server.enable
          value: "false"
        - name: profiler.micro.service.http.server.enable
          value: "false"
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b
        ports:
        - containerPort: 8080
          protocol: TCP
```

```
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
          livenessProbe:
            tcpSocket:
              port: 20002
            initialDelaySeconds: 10
            periodSeconds: 30

# B 应用 gray 版本,默认开启无损下线功能
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-b-gray
  name: spring-cloud-b-gray
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-b
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-b-gray
    spec:
      containers:
      - env:
```

```
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b-gray
        ports:
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
        lifecycle:
            preStop:
              exec:
                command:
                  - /bin/sh
                  - '-c'
                  - >-
                    wget http://127.0.0.1:54199/offline 2>/tmp/null;sleep
                    30;exit 0
        livenessProbe:
          tcpSocket:
            port: 20002
          initialDelaySeconds: 10
          periodSeconds: 30

# C 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  labels:
    app: spring-cloud-c
  name: spring-cloud-c
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
        msePilotAutoEnable: "on"
      labels:
        app: spring-cloud-c
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-c
        ports:
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
        livenessProbe:
          tcpSocket:
```

```
              port: 20003
            initialDelaySeconds: 10
            periodSeconds: 30

#HPA 配置
---
apiVersion: autoscaling.alibabacloud.com/v1beta1
kind: CronHorizontalPodAutoscaler
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: spring-cloud-b
spec:
  scaleTargetRef:
      apiVersion: apps/v1beta2
      kind: Deployment
      name: spring-cloud-b
  jobs:
  - name: "scale-down"
    schedule: "0 0/5 * * * *"
    targetSize: 1
  - name: "scale-up"
    schedule: "10 0/5 * * * *"
    targetSize: 2
---
apiVersion: autoscaling.alibabacloud.com/v1beta1
kind: CronHorizontalPodAutoscaler
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: spring-cloud-b-gray
spec:
  scaleTargetRef:
      apiVersion: apps/v1beta2
```

```
        kind: Deployment
        name: spring-cloud-b-gray
    jobs:
    - name: "scale-down"
      schedule: "0 0/5 * * * *"
      targetSize: 1
    - name: "scale-up"
      schedule: "10 0/5 * * * *"
      targetSize: 2
---
apiVersion: autoscaling.alibabacloud.com/v1beta1
kind: CronHorizontalPodAutoscaler
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: spring-cloud-c
spec:
    scaleTargetRef:
        apiVersion: apps/v1beta2
        kind: Deployment
        name: spring-cloud-c
    jobs:
    - name: "scale-down"
      schedule: "0 2/5 * * * *"
      targetSize: 1
    - name: "scale-up"
      schedule: "10 2/5 * * * *"
      targetSize: 2


# zuul 网关开启 SLB 暴露展示页面
---
apiVersion: v1
kind: Service
```

```
metadata:
  name: zuul-slb
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 20000
  selector:
    app: spring-cloud-zuul
  type: ClusterIP

# a 应用暴露 k8s service
---
apiVersion: v1
kind: Service
metadata:
  name: spring-cloud-a-base
spec:
  ports:
    - name: http
      port: 20001
      protocol: TCP
      targetPort: 20001
  selector:
    app: spring-cloud-a

---
apiVersion: v1
kind: Service
metadata:
  name: spring-cloud-a-gray
spec:
  ports:
    - name: http
```

```
      port: 20001
      protocol: TCP
      targetPort: 20001
  selector:
    app: spring-cloud-a-gray

# Nacos Server SLB Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-slb
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
  selector:
    app: nacos-server
  type: LoadBalancer
```

结果验证一：无损下线功能

由于我们对 spring-cloud-b 跟 spring-cloud-b-gray 应用均开启了定时 HPA，模拟每 5 分钟进行一次定时的扩缩容。

名称	任务名称	状态	调度周期	目标副本数	最近调度	创建时间	操作
spring-cloud-b	scale-down	Succeed	0 0/5 * * * *	1	2022-02-14 16:25:00	2022-02-09 14:19:32	任务添加或编辑 | 删除
	scale-up	Succeed	10 0/5 * * * *	2	2022-02-14 16:25:10	2022-02-09 14:19:32	

名称	任务名称	状态	调度周期	目标副本数	最近调度	创建时间	操作
spring-cloud-b-gray	scale-down	Succeed	0 0/5 * * * *	1	2022-02-14 17:40:00	2022-02-14 17:39:33	任务添加或编辑 | 删除
	scale-up	Succeed	10 0/5 * * * *	2	2022-02-14 17:40:10	2022-02-14 17:39:33	

登录 MSE 控制台，进入微服务治理中心->应用列表->spring-cloud-a->应用详情，从应用监控曲线，我们可以看到 spring-cloud-a 应用的流量数据：



gray 版本的流量在 pod 扩缩容的过程中请求错误数为 0，无流量损失。未打标的版本由于关闭了无损下线功能，在 pod 扩缩容的过程中有 20 个从 spring-cloud-a 发到 spring-cloud-b 的请求出现报错，发生了请求流量损耗。

结果验证二：服务预热功能

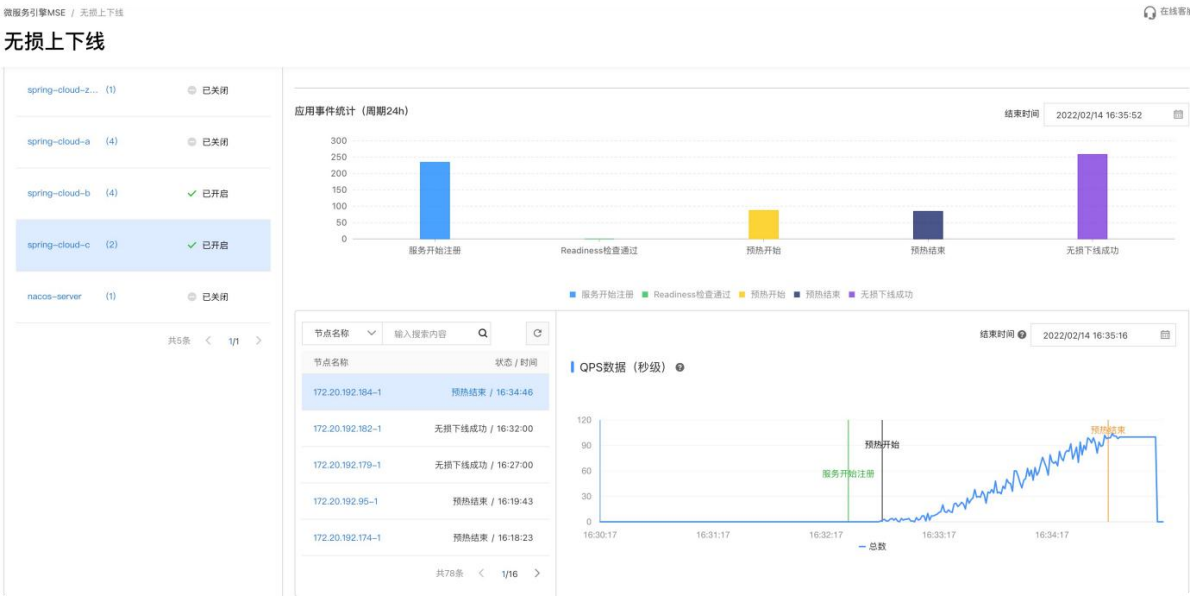
我们在 spring-cloud-c 应用开启了定时 HPA 模拟应用启动的过程，每隔 5 分钟做一次伸缩，在第 2 分钟第 0 秒缩容到 1 个节点，在第 2 分钟第 10 秒扩容到 2 个节点。

名称	任务名称	状态	调度周期	目标副本数	最近调度	创建时间	操作
spring-cloud-c	scale-down	Succeed	0 2/5 * * * *	1	2022-02-14 17:07:00	2022-02-14 13:56:14	任务添加或编辑 | 删除
	scale-up	Succeed	10 2/5 * * * *	2	2022-02-14 17:07:10	2022-02-14 13:56:14	

在预热应用的服务提供端 spring-cloud-c 开启服务预热功能。预热时长配置为 120 秒。



观察节点的流量，发现节点流量缓慢上升。并且能看到节点的预热开始和结束时间，以及相关的事件。



从上图可以看到开启预热功能的应用重启后的流量会随时间缓慢增加，在一些应用启动过程中需要预建连接池和缓存等资源的慢启动场景，开启服务预热能有效保护应用启动过程中缓存资源有序创建保障应用安全启动并做到流量无损。

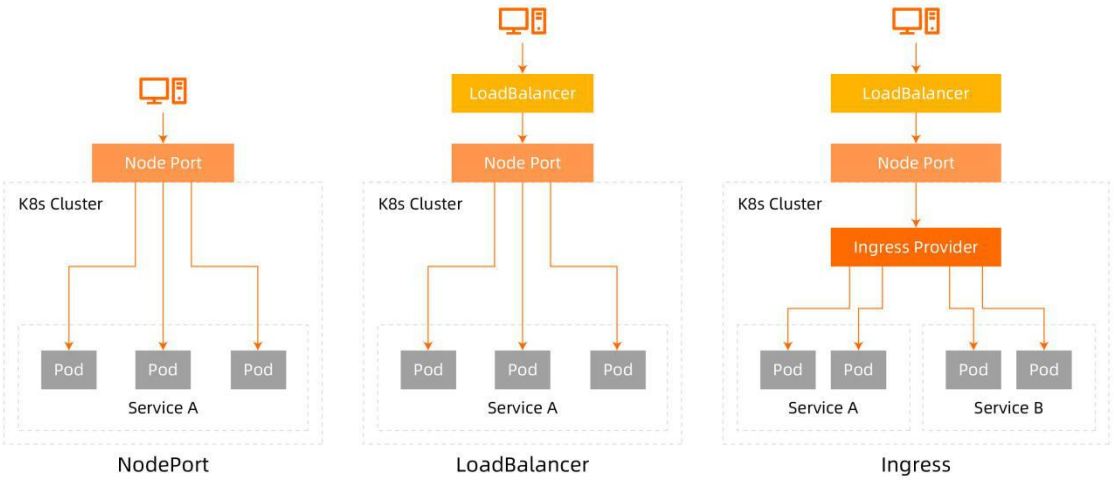
4.2 全链路灰度最佳实践

如何落地可灰度、可观测、可回滚的安全生产三板斧能力，满足业务高速发展情况下快速迭代和小心验证的诉求，是企业在微服务化深入过程中必须要面对的问题。在云原生流行的当下，这个问题又有了一些新的思路与解法。

K8s Ingress 网关

我们先从 Ingress网关谈起，聊一下通过 Ingress 配置路由转发。

K8s 集群内的网络与外部是隔离的，即在 K8s 集群外部无法直接访问集群内部的服务，如何让将 K8s 集群内部的服务提供给外部用户呢？K8s 社区有三种方案：NodePort、LoadBalancer、Ingress，下图是对这三种方案的对比：



注：Ingress ≠ Ingress Provider, Ingress是K8s定义路由规则的Resource, Ingress Provider是Ingress规则的实现者与执行者

通过对比可以看到 Ingress 是更适合业务使用的一种方式，可以基于其做更复杂的二次路由分发，这也是目前用户主流的选择。

随着云原生应用微服务化深入，用户需要面对复杂路由规则配置、支持多种应用层协议（HTTP、HTTPS 和 QUIC 等）、服务访问的安全性以及流量的可观测性等诉求。K8s 希望通过 Ingress

来标准化集群入口流量的规则定义，但实际业务落地时需要的功能点要远比 Ingress 提供的多，为了满足不断增长的业务诉求，让用户轻松应对云原生应用的流量管理，各类 Ingress-Provider 也都在 Ingress 的标准下进行各种扩展。

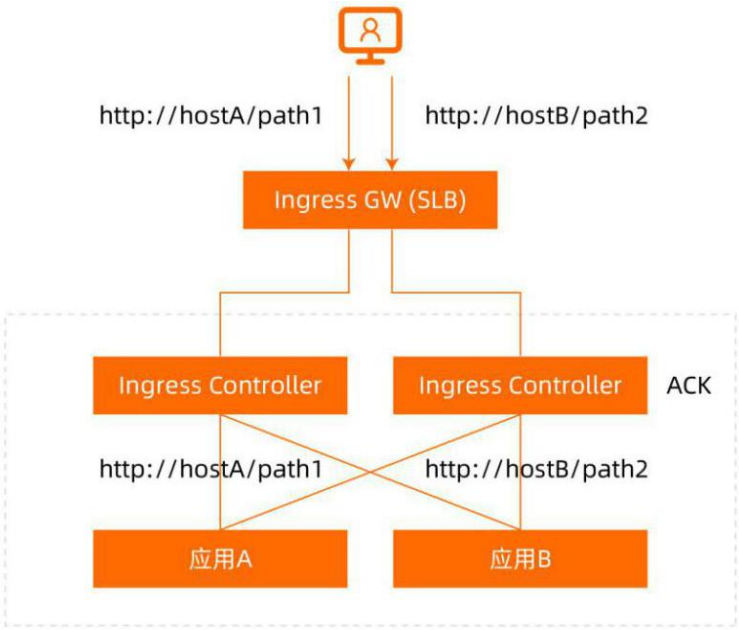
各种 Ingress-Provider 如何路由转发

下面我会简单介绍 K8s 下的各种 Ingress网关的实现，以及如何配置路由转发规则等。

Nginx Ingress

Nginx Ingress 由资源对象 Ingress、Ingress Controller、Nginx 三部分组成，Ingress Controller用以将 Ingress 资源实例组装成 Nginx 配置文件（nginx.conf），并重新加载 Nginx 使变更的配置生效。使用Nginx Ingress Controller 是官方提供的介入Ingress 的方式，最容易部署，但是受性能限制，功能较为单一，且更新 Nginx 配置需要 reload。

基于 Nginx Ingress Controller 配置路由转发



基于部署了 Nginx Ingress Controller 组件的 K8s 集群，可以实现路由转发功能，能够根据域名、路径进行路由转发，也能够支持基于 Ingress 的 Annotations 进行简单规则的灰度流量管理，如权重、Header 等。在当下趋势中，Nginx Ingress 依旧是使用最广泛的。

ALB Ingress

ALB 产品介绍

应用型负载均衡 ALB（Application Load Balancer）是阿里云推出的专门面向 HTTP、HTTPS 和 QUIC 等应用层负载场景的负载均衡服务，具备超强弹性及大规模七层流量处理能力。



ALB 特性

- 弹性自动伸缩：ALB 同时提供域名与 VIP（Virtual IP address），支持对多台云服务器进行流量分发以扩展应用系统的服务能力，通过消除单点故障来提升应用系统的可用性。ALB 允许您自定义可用区组合，并支持在可用区间弹性缩放，避免单可用区资源瓶颈。

- 高级的协议：支持 ALB支持应用传输协议 QUIC，在实时音视频、互动直播和游戏等移动互联网应用场景中，访问速度更快，传输链路更安全可靠。ALB 同时支持 gRPC 框架，可实现海量微服务间的高效 API 通信。

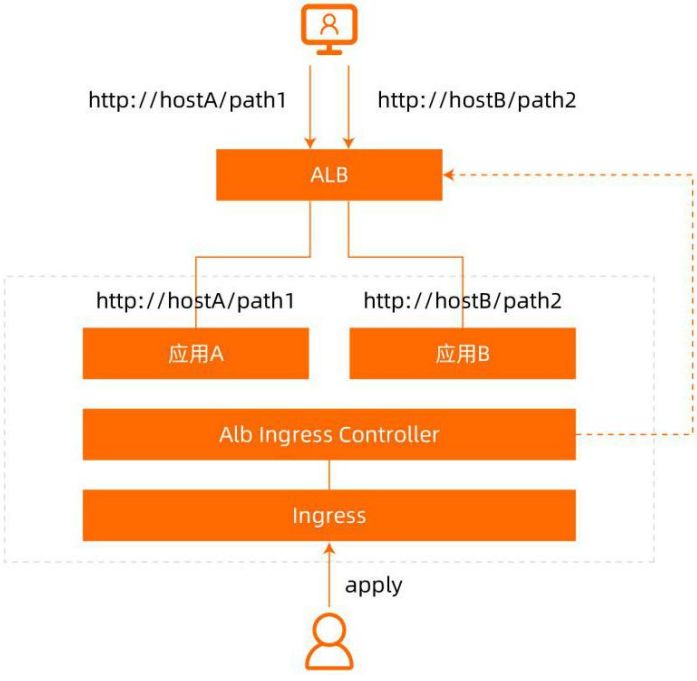
- 基于内容的高级路由：ALB支持基于 HTTP 标头、Cookie、HTTP 请求方法等多种规则来识别特定业务流量，并将其转发至不同的后端服务器。同时 ALB 还支持重定向、重写以及自定义 HTTPS 标头等高级操作。

- 安全加持“ALB自带分布式拒绝服务 DDoS（Distributed Denial of Service）防护，一键集成 Web 应用防火墙（Web Application Firewall，简称 WAF）。同时 ALB支持全链路 HTTPS 加密，可以实现与客户端或后端服务器的 HTTPS 交互；支持 TLS 1.3 等高效安全的加密协议，面向加密敏感型业务，满足Zero-Trust 新一代安全技术架构需求；支持预制的安全策略，您可以自定义安全策略。

- 云原生应用：在云原生时代，PaaS 平台将下沉到基础设施，成为云的一部分。随着云原生逐步成熟，互联网、金融、企业等诸多行业新建业务时选择云原生部署，或对现有业务进行云原生化改造。ALB 与容器服务 Kubernetes 版（Alibaba Cloud Container Service for Kubernetes，简称 ACK）深度集成，是阿里云的官方云原生Ingress网关。

- 弹性灵活的计费：ALB 通过弹性公网IP（Elastic IP Address，简称 EIP）和共享带宽提供公网能力，实现公网灵活计费；同时采用了更先进的、更适合弹性业务峰值的基于容量单位（LCU）的计价方案。

基于 ALB Ingress Controller 配置路由转发



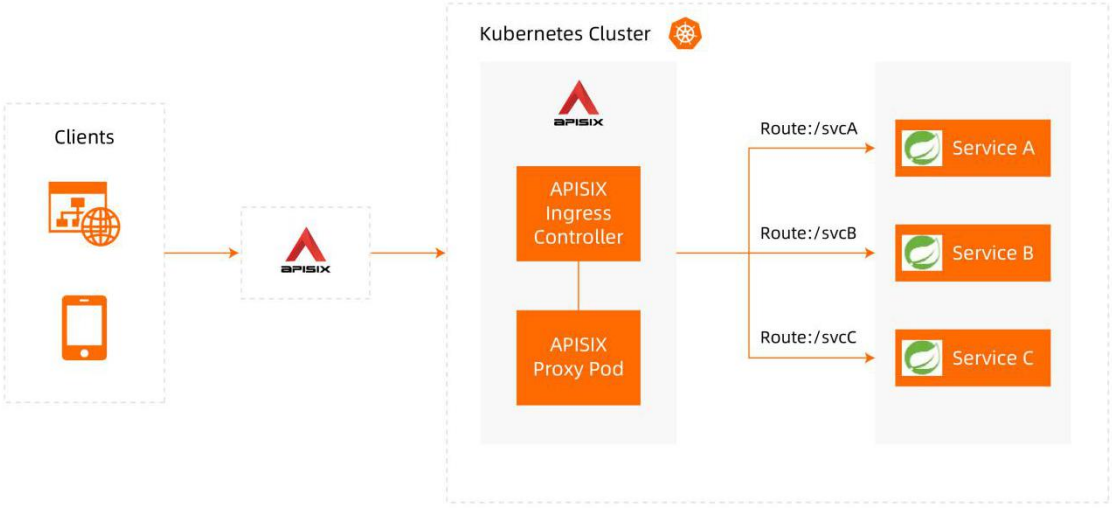
部署了 Alb Ingress Controller 组件的 K8s 集群，组件提供了自定义资源 AlbConfig，该资源实际是一个 ALB 实例模版，可以配置已经存在的 ALB 实例或可以在属性中配置虚拟交换机的 ID 来由 ACK 集群代购 ALB。创建 AlbConfig 后，需要配置 IngressClass用于关联 Ingress 和 Albconfig，用户在创建 Ingress 资源时指定 Spec.IngressClassName 即可将 Ingress 资源关联

到 AlbConfig 模版。alb-ingress-controller 监听到 Ingress 变化后，根据配置的路由规则能够调用ALB 接口进行ALB 实例监听及后端的动态配置。

该方案将应用路由管理面与业务面分离，请求经过 ALB 实例后会直接转发到后端应用，降低了集群的负载，同时基于 ALB 实现了证书自动发现、高可靠、超大弹弹性、大规模流量特性支持。

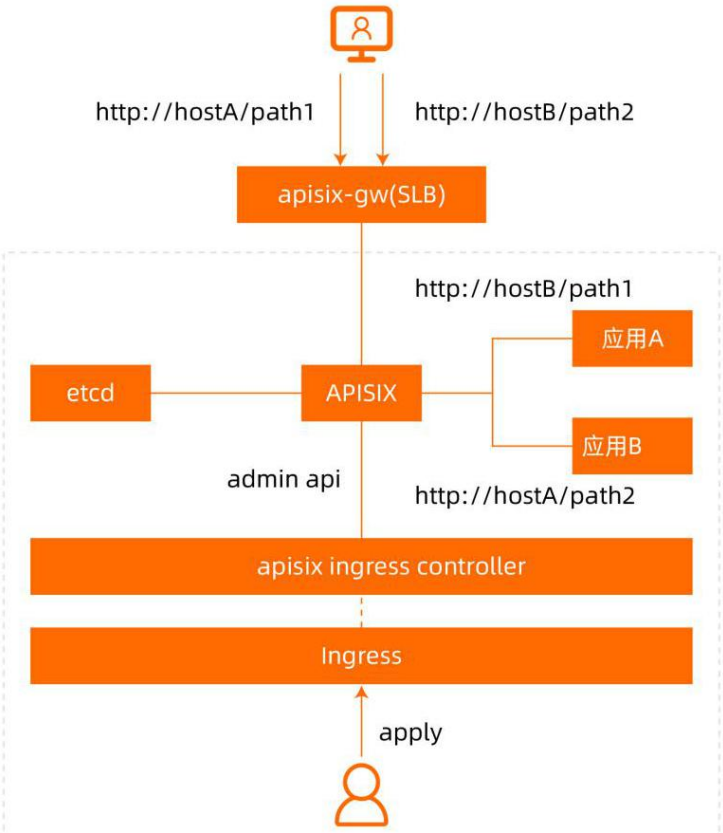
APISIX Ingress

APISIX Ingress 跟 Kubernetes Ingress Nginx 的区别主要在于 APISIX Ingress 是以 ApacheAPISIX 作为实际承载业务流量的数据面。如下图所示，当用户请求到具体的某一个服务/API/网页时，通过外部代理将整个业务流量/用户请求传输到 K8s 集群，然后经过 APISIX Ingress 进行后续处理。



从上图可以看到，APISIX Ingress 分成了两部分。一部分是 APISIX Ingress Controller，作为控制面它将完成配置管理与分发。另一部分 APISIX Proxy Pod 负责承载业务流量，它是通过 CRD(Custom Resource Definitions) 的方式实现的。Apache APISIX Ingress 除了支持自定义资源外，还支持原生的 K8s Ingress 资源。

基于 APISIX Ingress Controller 的应用路由



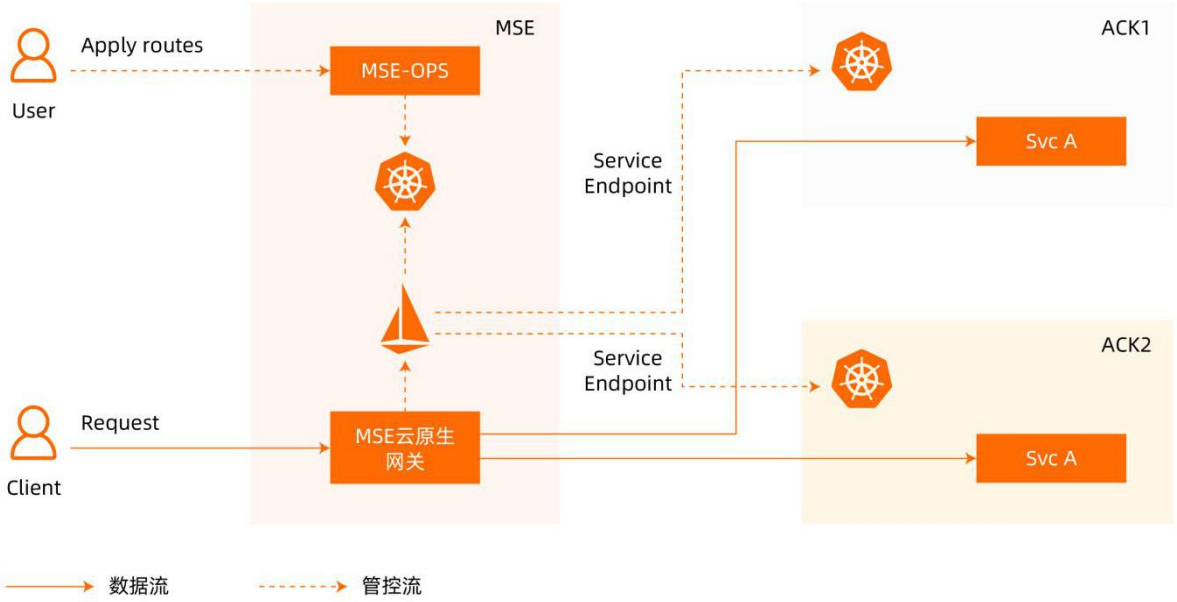
如上图所示，我们部署了 APISIX Ingress Controller 组件的集群，能够实现基于 Ingress 资源和自定义资源 ApisixRoute 进行路由配置，控制器监听资源的变更事件，调用apisix-admin api 进行规则的持久化存储。流量经过 APISIX 配置的 LoadBalancer 类型 Service网关从 ETCD 中同步配置，并将请求转发到上游。

APISIX Ingress Controller 基于 Apache APISIX，支持 K8s 中的 Ingress 资源进行路由配置，也支持通过自定义资源对接到 APISIX 的路由、插件、上游等资源配置。支持动态配置路由规则、热插拔插件、更丰富 的路由规则支持，APISIX 云原生网关也能够提供可观测、故障注入、链路追踪等能力。使用高可靠 ETCD 集群作为配置中心，进行apisix 配置的存储和分发。

MSE 云原生网关 Ingress

MSE 云原生网关是阿里云推出的下一代网关，将传统的流量网关和微服务网关合并，在降低 50% 资源成本的同时为用户提供了精细化的流量治理能力，支持 ACK 容器服务、Nacos、Eureka、固定地址、FaaS 等多种服务发现方式，支持多种认证登录方式快速构建安全防线，提供全方面、多视角的监控体系，如指标监控、日志分析以及链路追踪。

基于 MSE 云原生网关 Ingress Controller 的应用路由



上图是 MSE 云原生网关在多集群模式下对业务应用进行流量管理的应用场景。MSE 云原生网关与阿里云容器服务 ACK 深度集成，可以做到自动发现服务以及对应的端点信息并动态秒级生效。用户只需简单在 MSE 管控平台关联对应的 K8s ACK 集群，通过在路由管理模块中配置路由来对外暴露 ACK 中服务即可，同时可以按集群纬度进行流量分流以及故障转移。此外，用户可以为业务路由实施额外的策略，如常见的限流、跨域或者重写。

MSE 云原生网关提供的流量治理能力与具体的服务发现方式解耦，无论后端服务采用何种服务发现方式，云原生网关以统一的交互体验来降低上手门槛，满足用户业务日益增长的流量治理诉求。

上文介绍了 Nginx Ingress、ALB Ingress、APISIX Ingress 以及 MSE 云原生网关 Ingress 这五种 Ingress 的路由转发与配置，我们可以按照自己的业务需求与复杂度按需选择合适的 Ingress 实现。

假设我们已经配好了 Ingress 的路由转发，那么在多应用环境下，我们该如何简单地玩转全链路灰度呢？

基于 Ingress 实现全链路流量灰度

我们基于全链路灰度的实践，提出了“泳道”这一个概念，当然这一概念在分布式软件领域并非新词。

名词解释

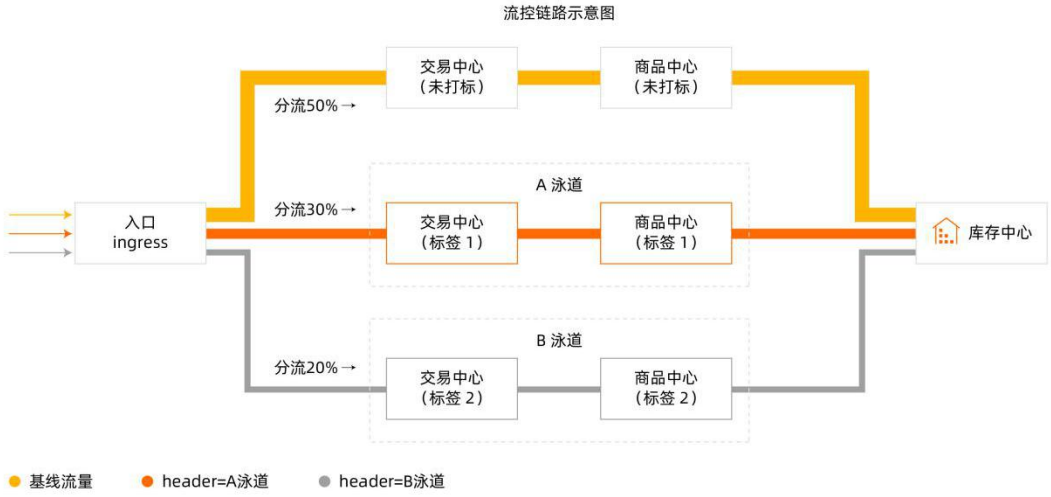
- 泳道：为相同版本应用定义的一套隔离环境。只有满足了流控路由规则的请求流量才会路由到对应泳道里的打标应用。一个应用可以属于多个泳道，一个泳道可以包含多个应用，应用和泳道是多对多的关系。

- 泳道组：泳道的集合。泳道组的作用主要是为了区分不同团队或不同场景。

- 基线：指业务所有服务都部署到了这一环境中。未打标的应用属于基线稳定版本的应用，我们这里指稳定的线上环境。

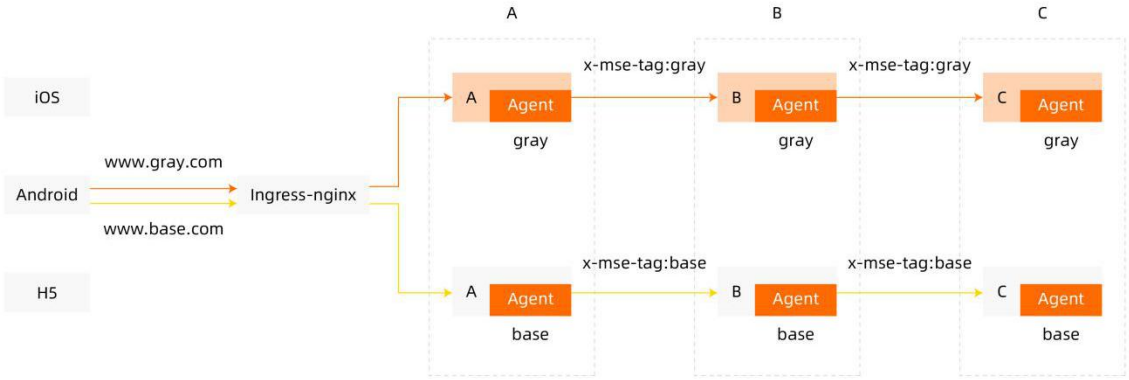
- 入口应用：微服务体系内的流量入口。入口应用可以是 Ingress网关、或者是自建的 Spring Cloud Gateway、Netflix Zuul Gateway 引擎类型网关或者 Spring Boot、Spring MVC、Dubbo 应用等。

为什么要区分出入口应用？因为全链路灰度场景下，我们只需对入口应用进行路由转发的规则配置，后续微服务只需要按照透传的标签进行染色路由（实现“泳道”的流量闭环能力）。



从上图中可以看到，我们分别创建了泳道 A 与泳道 B，里面涉及了交易中心、商品中心两个应用，分别是标签标签 2，其中 A 泳道分流了线上 30%的流量，B 泳道分流了线上 20%的流量，基线环境（即未打标的环境）分流了线上 50%的流量。

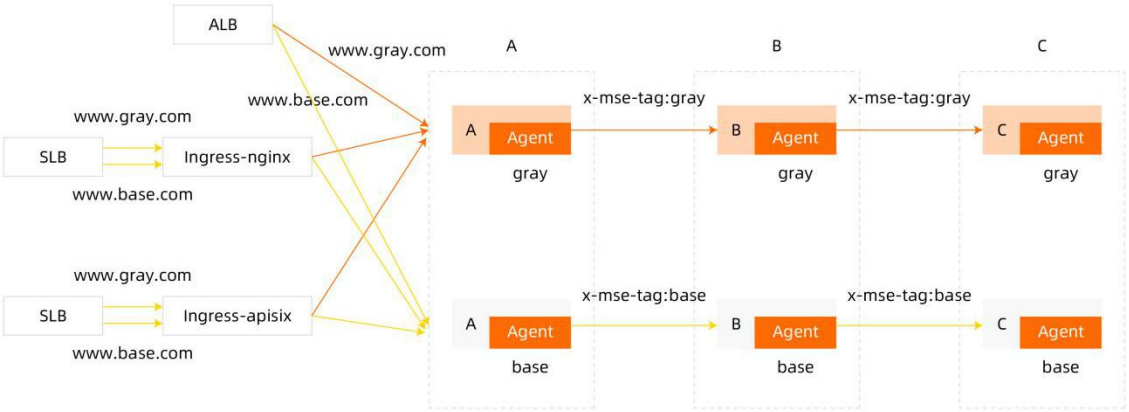
全链路灰度的技术解析



我们通过在 deployment 上配置注解 alicloud.service.tag：gray 标识应用灰度版本，并带标注册到注册中心中，在灰度版本应用上开启全链路泳道(经过机器的流量自动染色)，支持灰度流量自动添加灰度 x- mse-tag：gray 标签，通过扩展 consumer 的路由能力将带有灰度标签的流量转发到目标灰度应用。

基于各种 Ingress 网关的全链路灰度

基于全链路灰度能力搭配合适的入口路由网关即可实现全链路流量灰度，如上述使用Ingress-Nginx、Ingress-APISIX、ALB、MSE 云原生网关均可以作为流量入口。



全链路灰度的产品实现

功能性与易用性是产品化必须思考的点，我们需要从用户的视角出发，端到端思考全流程该如何去实践、如何去落地。在阿里云上关于微服务全链路灰度能力有以下两款产品都提供了完整的全链路灰度解决方案。

MSE 全链路灰度方案

全链路灰度作为 MSE 微服务治理专业版中的核心功能，具备以下六大特点：

- 全链路隔离流量泳道
  - 通过设置流量规则对所需流量进行'染色'，'染色'流量会路由到灰度机器。
  - 灰度流量携带灰度标往下游传递，形成灰度专属环境流量泳道，无灰度环境应用会默认选择未打标的基线环境。

- 端到端的稳定基线环境
  - 未打标的应用属于基线稳定版本的应用，即稳定的线上环境。当我们将发布对应的灰度版本代码，然后可以配置规则定向引入特定的线上流量，控制灰度代码的风险。

- 流量一键动态切流
  - 流量规则定制后，可根据需求进行一键停启，增删改查，实时生效。灰度引流更便捷。

- 低成本接入，基于 Java Agent 技术实现无需修改一行业务代码
  - MSE 微服务治理能力基于 Java Agent 字节码增强的技术实现，无缝支持市面上近 5 年的所有 Spring Cloud 和 Dubbo 的版本，用户不用改一行代码就可以使用，不需要改变业务的现有架构，随时可上可下，没有绑定。只需开启 MSE 微服务治理专业版，在线配置，实时生效。

- 可观测能力
  - 具备泳道级别的单应用可观测能力。



- 具备全链路应用的可观测能力，可以从全局视角观察流量是否存在逃逸情况。灰没灰到，一目了然。



- 具备无损上下线能力，使得发布更加丝滑
- 应用开启 MSE 微服务治理后就具备无损上下线能力，大流量下的发布、回滚、扩容、缩容等场景，均能保证流量无损。

创建流量泳道组

需要将泳道中涉及的应用添加到泳道组涉及应用中。



创建流量泳道

在使用泳道功能前，我们默认已经存在了一个包含所有服务在内的基线（未达标）环境。



我们需要去部署隔离版本的应用Deployment，同时去给他们打上标签，并且给泳道选择对应一一关联的标签（test 泳道关联 gray 标签）。



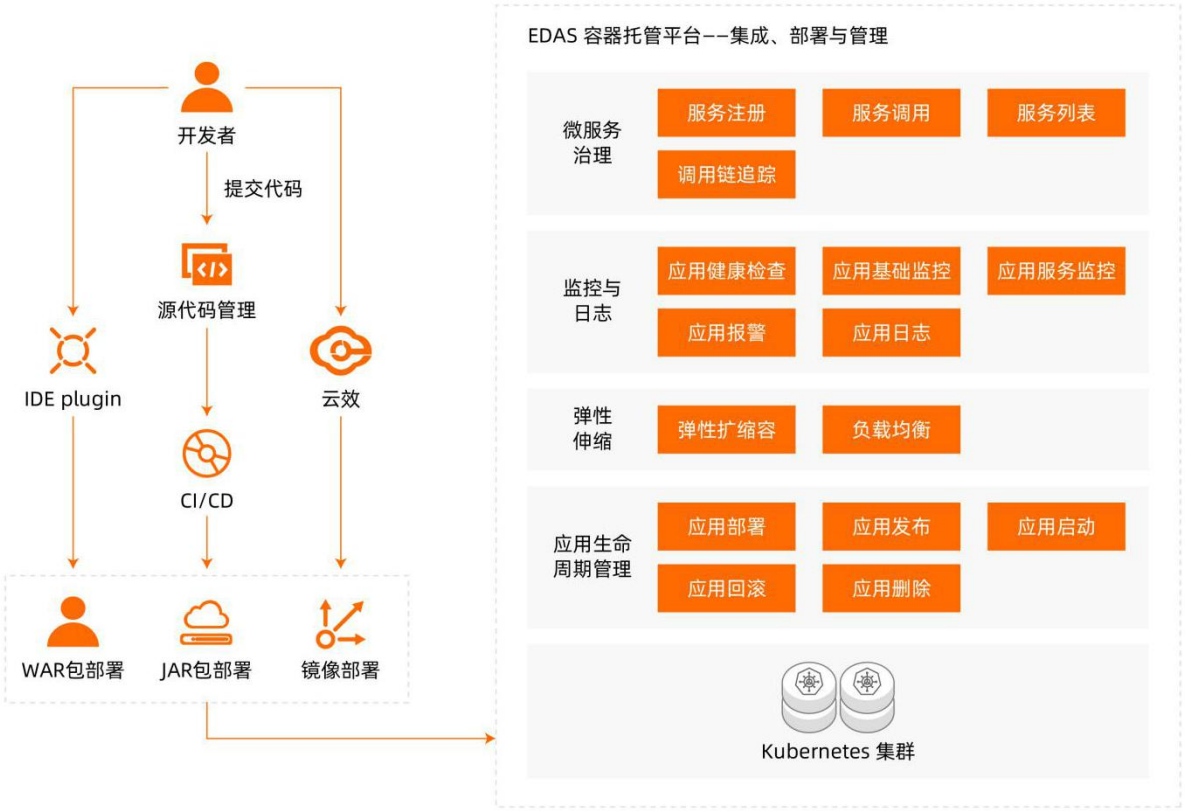
然后需要去配置流量入口的 Ingress 规则，比如访问 www.base.com 路由到 A 应用的 base 版本，访问 www.gray.com 路由到 A 应用的 gray 版本。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
 name: spring-cloud-a-base
spec:
 rules:
 - host: www.base.com
   http:
     paths:
     - backend:
         serviceName: spring-cloud-a-base
         servicePort: 20001
       path: /
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
 name: spring-cloud-a-gray
spec:
 rules:
 - host: www.gray.com
   http:
     paths:
     - backend:
         serviceName: spring-cloud-a-gray
         servicePort: 20001
       path: /
```

EDAS 全链路灰度方案

EDAS 产品介绍

EDAS 是应用托管和微服务管理的云原生PaaS 平台，提供应用开发、部署、监控、运维等全栈式解决方案，同时支持 Spring Cloud 和 Apache Dubbo 等微服务运行环境，助力您的应用轻松上云。



在 EDAS 平台中，用户可以通过 WAR 包、JAR 包或镜像等多种方式快速部署应用到多种底层服务器集群，免集群维护，能够轻松部署应用的基线版本和灰度版本。EDAS无缝接入了 MSE 微服务治理能力，部署在 EDAS 的应用无需额外安装 Agent 即可零代码入侵获得应用无损上下线、金丝雀发布、全链路流量控制等高级特性。

目前 EDAS支持微服务应用为入口的全链路灰度能力，下面简单介绍在 EDAS 中的配置全链路灰度流量的方法。

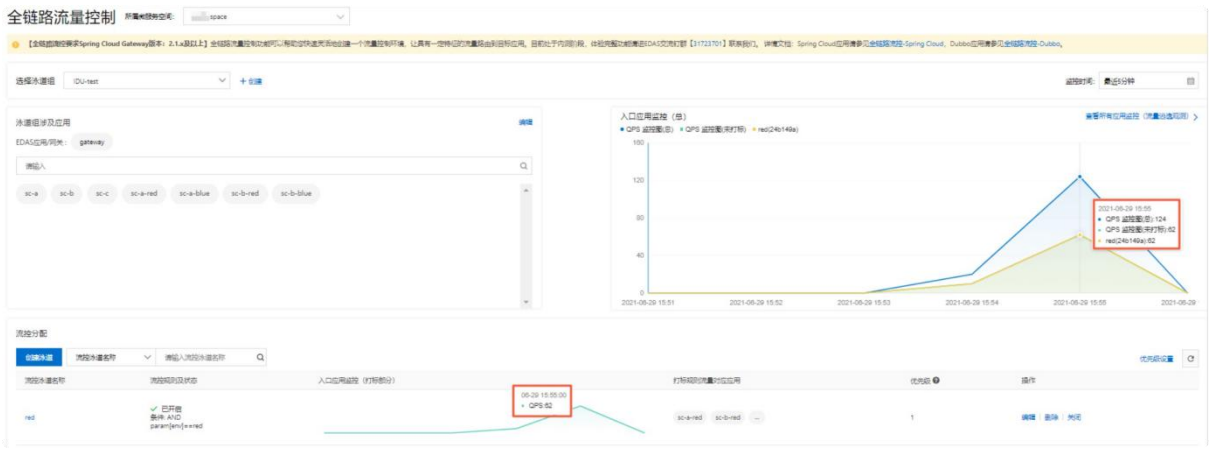
创建流量泳道组和泳道



在创建泳道时需要选择入口类型，目前仅支持部署在 EDAS 中的入口应用作为泳道入口应用，需要将泳道中涉及的基线版本和灰度版本添加到泳道组涉及应用中。



在创建泳道时支持基于路径配置规则定向引入特定的线上流量，配置打标流量应用形成灰度环境链路。泳道支持基于 Cookie、Header、Parameter 进行流量控制。



配置泳道成功后，可以在全链路流量控制界面选择目标泳道组进行流量观测，包括入口应用总的监控图、未打标部分监控图和打标部分监控图，及泳道组内所有应用的监控图。

应用路由作为流量入口实现全链路灰度

EDAS 平台支持用户为 K8s 应用基于 Nginx Ingress 创建应用路由，结合 EDAS 对全链路流控的支持，用户能够直接在 EDAS 控制台实现使用Nginx Ingress 作为流量入口网关的全链路灰度。

在 EDAS 平台中部署基线版本应用、灰度版本应用、入口应用后，根据上述创建流量泳道组和泳道的步骤创建灰度泳道后，可以为入口应用绑定 K8s 的 Service 资源以提供流量入口。可以为入口应用配置 LoadBalancer 类型 Service，以提供入口应用的对外访问，也可以基于 K8s 集群中已有的 Nginx Ingress网关，为入口应用配置 ClusterIP 类型 Service 并配置应用路由，以避免额外分配公网IP。

下面简单介绍为入口应用配置应用路由作为流量入口的方法。



在部署完成的应用详情页面，支持进行应用的访问方式配置，在这里可以选择为入口应用绑定 LoadBalancer 类型 Service 以直接对外访问，也可以创建 ClusterIP 类型 Service，如下图：



配置成功后，可以在 EDAS 应用路由页面点击创建应用路由，选择该入口应用所在的集群，命名空间，应用名称，上述步骤创建的服务名称及端口以配置 Ingress 资源，如下图：



配置成功后，即可使用该 Ingress 资源配置的域名和路径并结合在泳道中配置的灰度流量规则实现全链路灰度。

更进一步

EDAS 中的全链路流量控制目前仅支持部署在 EDAS 中的入口应用作为流量入口，在 K8s 主导的云原生化趋势下，EDAS 将支持使用Ingress 作为流量入口，用户不需要额外部署网关应用，直接使用Ingress 资源配置转发规则即可作为流量入口。不仅如此，EDAS 也将支持 ALB Ingress、APISIX Ingress 以及 MSE 云原生网关 Ingress，并且在这个基础上全链路灰度能力也会进一步升级，支持基于各种 Ingress-Provider 网关的全链路灰度能力。

尾

我们发现在云原生抽象的 Ingress 下，再谈全链路灰度，一切问题都变得更加标准化与简单起来。本文通过介绍各种 Ingress网关实现的路由转发能力，再配合上基于“泳道”实现的 Ingress 全链路灰度方案，企业可以快速落地全链路灰度这个微服务核心能力。

4.2.1Ingress-nginx 全链路灰度

背景

微服务架构下，有一些需求开发，涉及到微服务调用链路上的多个微服务同时发生了改动，通常每个微服务都会有灰度环境或分组来接受灰度流量，我们希望通过进入上游灰度环境的流量，也能进入下游灰度的环境中，确保 1 个请求始终在灰度环境中传递，即使这个调用链路上有一些微服务没有灰度环境，这些应用请求下游的时候依然能够回到灰度环境中。通过 MSE 提供的全链路灰度能力，可以在不需要修改任何您的业务代码的情况下，能够轻松实现上述能力。

本文主要介绍通过 Ingress-nginx 来实现全链路灰度功能。我们假设应用的架构由 Ingress-nginx 以及后端的微服务架构（Spring Cloud）来组成，后端调用链路有 3 跳，购物车（a），交易中心（b），库存中心（c），客户端通过 客户端或者是 H5 页面来访问后端服务，他们通过 Nacos 注册中心做服务发现。

前提条件

安装 Ingress-nginx 组件

访问容器服务控制台，打开应用目录，搜索 ack-ingress-nginx ，选择命名空间 kube-system ，点击创建，安装完成后，在 kube-system 命名空间中会看到一个 deployment ack-ingress-nginx-default-controller ，表明安装成功。

```
$ kubectl get deployment -n kube-system
NAME                                 READY   UP-TO-DATE   AVAILABLE   AGE
ack-ingress-nginx-default-controller   2/2     2            2           18h
```

开启 MSE 微服务治理

- 点击开通 MSE 微服务治理专业版以使用全链路灰度能力。
- 访问容器服务控制台，打开应用目录，搜索 ack-onepilot ，点击创建。

- 在 MSE 服务治理控制台，打开 K8s 集群列表，选择对应集群，对应命名空间，并打开微服务治理。

部署 Demo 应用程序

将下面的文件保存到 ingress-gray-demo-deployment-set.yaml 中，并执行`kubectl apply -f ingress-gray-demo-deployment-set.yaml` 以部署应用，这里我们将要部署 A，B，C 三个应用，每个应用分别部署一个基线版本和一个灰度版本。

```
# A 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a
```

```
        ports:
        - containerPort: 20001
        livenessProbe:
          tcpSocket:
            port: 20001
          initialDelaySeconds: 10
          periodSeconds: 30

# A 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a-new
  strategy:
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a-new
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: profiler.micro.service.tag.trace.enable
          value: "true"
```

```
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a-new
        ports:
        - containerPort: 20001
        livenessProbe:
          tcpSocket:
            port: 20001
          initialDelaySeconds: 10
          periodSeconds: 30

# B 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-b
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b
  strategy:
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
```

```
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 20002
          initialDelaySeconds: 10
          periodSeconds: 30

# B 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-b-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b-new
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b-new
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
```

```
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b-new
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 20002
          initialDelaySeconds: 10
          periodSeconds: 30

# C 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-c
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
```

```
        imagePullPolicy: Always
        name: spring-cloud-c
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 20003
          initialDelaySeconds: 10
          periodSeconds: 30

# C 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-c-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-c-new
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c-new
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
```

```
          imagePullPolicy: IfNotPresent
          name: spring-cloud-c-new
          ports:
          - containerPort: 8080
          livenessProbe:
            tcpSocket:
              port: 20003
            initialDelaySeconds: 10
            periodSeconds: 30

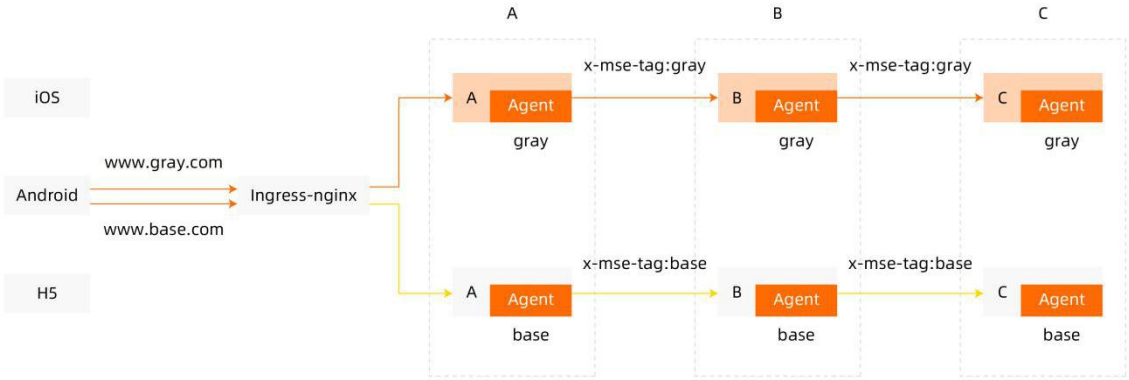
# Nacos Server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nacos-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      labels:
        app: nacos-server
    spec:
      containers:
      - env:
        - name: MODE
          value: standalone
        image: nacos/nacos-server:latest
        imagePullPolicy: Always
        name: nacos-server
      dnsPolicy: ClusterFirst
```

```
      restartPolicy: Always

# Nacos Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
  selector:
    app: nacos-server
  type: ClusterIP
```

场景一：对经过机器的流量进行自动染色，实现全链路灰度

有时候，我们可以通过不同的域名来区分线上基线环境和灰度环境，灰度环境有单独的域名可以配置，假设我们通过访问 www.gray.com 来请求灰度环境，访问 www.base.com 走基线环境。



调用链路 Ingress-nginx -> A -> B -> C ，其中 A 可以是一个 spring-boot 的应用。

注意：入口应用 A 的 gray 和 A 的 base 环境，需要增加 profiler.micro.service.tag.trace.

enable=true 这个环境变量，表示开启向后透传当前环境的标签的功能。这样，当 Ingress-nginx 路由 A 的 gray 之后，即使请求中没有携带任何 header，因为开启了此开关，所以往后调用的时候会自动添加 x-mse-tag:gray 这个 header，其中的 header 的值 gray 来自于 A 应用配置的标签信息。如果原来的请求中带有 x-mse-tag:gray 则会以原来请求中的标签优先。

针对入口应用 A ，配置两个 K8s service，spring-cloud-a-base 对应 A 的 base 版本，spring-cloud- a-gray 对应 A 的 gray 版本。

```
apiVersion: v1
kind: Service
metadata:
  name: spring-cloud-a-base
spec:
  ports:
    - name: http
      port: 20001
      protocol: TCP
      targetPort: 20001
  selector:
    app: spring-cloud-a

---
apiVersion: v1
kind: Service
metadata:
  name: spring-cloud-a-gray
spec:
  ports:
    - name: http
      port: 20001
      protocol: TCP
      targetPort: 20001
  selector:
    app: spring-cloud-a-new
```

配置入口的 Ingress 规则，访问 www.base.com 路由到 a 应用的 base 版本，访问 www.gray.com 路由到 a 应用的 gray 版本。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-base
spec:
  rules:
  - host: www.base.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-base
          servicePort: 20001
        path: /

---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-gray
spec:
  rules:
  - host: www.gray.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-gray
          servicePort: 20001
        path: /
```

此时，访问 www.base.com 路由到基线环境。

```
curl -H"Host:www.gray.com" http://106.14.155.223/a
Agray[172.18.144.160] -> Bgray[172.18.144.57] -> Cgray[172.18.144.157]%
```

此时，访问 www.gray.com 路由到灰度环境。

```
curl -H"Host:www.gray.com" http://106.14.155.223/a
Agray[172.18.144.160] -> Bgray[172.18.144.57] -> Cgray[172.18.144.157]%
```

进一步的，如果入口应用 A 没有灰度环境，访问到 A 的 base 环境，又需要在 A -> B 的时候进入灰度环境，则可以，通过增加一个特殊的 header `x-mse-tag` 来实现，header 的值是想要去的环境的标签，例如 `gray` 。

```
curl -H"Host:www.base.com"  -H"x-mse-tag:gray" http://106.14.155.223/a
A[172.18.144.155] -> Bgray[172.18.144.139] -> Cgray[172.18.144.8]%
```

可以看到第一跳，进入了 A 的 base 环境，但是 A->B 的时候又重新回到了灰度环境。

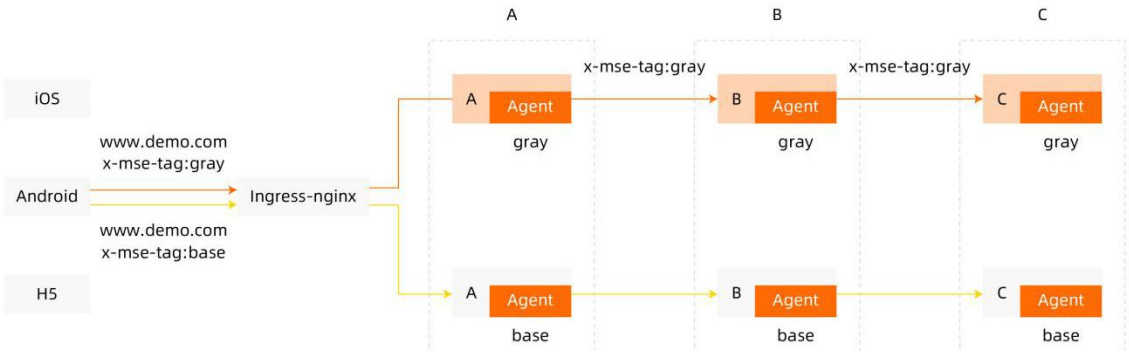
这种使用方式的好处是，配置简单，只需要在 Ingress 处配置好规则，某个应用需要灰度发布的时候，只需要在灰度环境中部署好应用，灰度流量自然会进入好灰度机器中，如果验证没问题，则将灰度的镜像发布到基线环境中；如果一次变更有多个应用需要灰度发布，则把他们都加入到灰度环境中即可。

最佳实践

1.给所有灰度环境的应用打上 gray 标，基线环境的应用默认不打标。
2.线上常态化引流 2%的流量进去灰度环境中。

场景二：通过给流量带上特定的 header 实现全链路灰度

有些客户端没法改写域名，希望能访问 www.base.com 通过传入不同的 header 来路由到灰度环境。例如下图中，通过添加 `x-mse-tag:gray` 这个 header，来访问灰度环境。



这个时候 base 的 Ingress 规则如下，注意这里增加了 nginx.ingress.kubernetes.io/canary 相关的多条规则。

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-base
spec:
  rules:
  - host: www.base.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-base
          servicePort: 20001
        path: /
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-gray
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-by-header: "x-mse-tag"
    nginx.ingress.kubernetes.io/canary-by-header-value: "gray"
    nginx.ingress.kubernetes.io/canary-weight: "0"
spec:
```

```
rules:
- host: www.base.com
  http:
    paths:
    - backend:
        serviceName: spring-cloud-a-gray
        servicePort: 20001
      path: /
```

此时，访问 www.base.com 路由到基线环境。

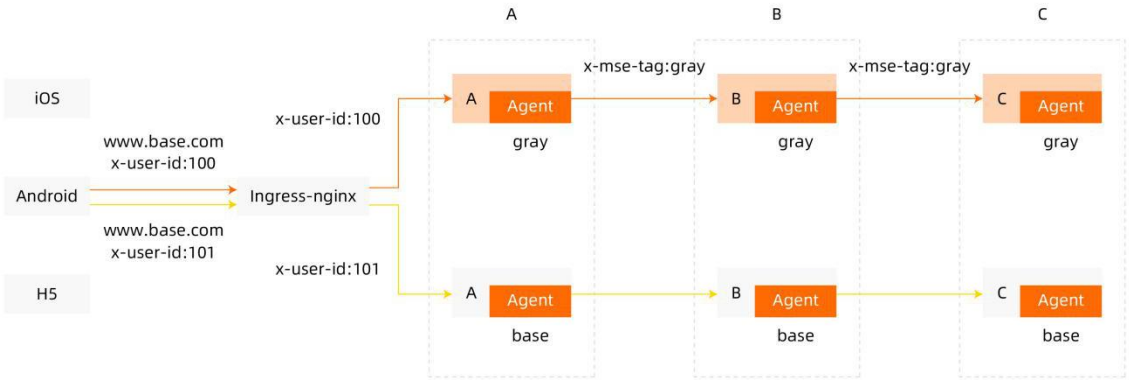
```
curl -H"Host:www.base.com" http://106.14.155.223/a
A[172.18.144.155] -> B[172.18.144.56] -> C[172.18.144.156]%
```

如何访问灰度环境呢？只需要在请求中增加一个 header `x-mse-tag:gray` 即可。

```
curl -H"Host:www.base.com"  -H"x-mse-tag:gray" http://106.14.155.223/a
Agray[172.18.144.82] -> Bgray[172.18.144.57] -> Cgray[172.18.144.8]%
```

可以看到 Ingress 根据这个 header 直接路由到了 A 的 gray 环境中。

更进一步的，还可以借助 Ingress 实现更复杂的路由，比如客户端已经带上了某个 header，想要利用现成的 header 来实现路由，而不用新增一个 header，例如下图所示，假设我们想要 x-user-id 为 100 的请求进入灰度环境。



只需要增加下面这 4 条规则：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-base
spec:
  rules:
  - host: www.base.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-base
          servicePort: 20001
        path: /
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-base-gray
  annotations:
    nginx.ingress.kubernetes.io/canary: "true"
    nginx.ingress.kubernetes.io/canary-by-header: "x-user-id"
    nginx.ingress.kubernetes.io/canary-by-header-value: "100"
    nginx.ingress.kubernetes.io/canary-weight: "0"
spec:
  rules:
  - host: www.base.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-gray
          servicePort: 20001
        path: /
```

访问的时候带上特殊的 header ， 满足条件进入灰度环境。

```
curl -H"Host:www.base.com" -H"x-user-id:100" http://106.14.155.223/a
Agray[172.18.144.93] -> Bgray[172.18.144.24] -> Cgray[172.18.144.25]
```

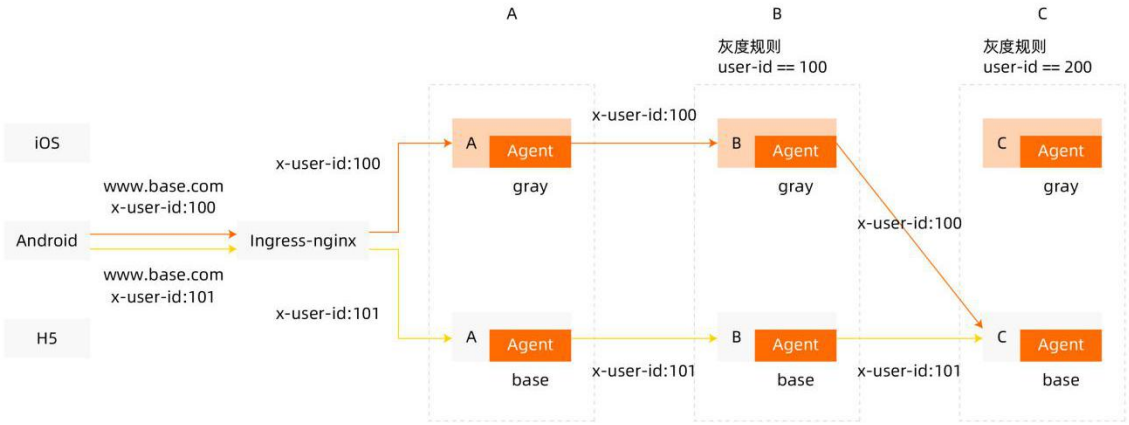
不满足条件的请求，进入基线环境：

```
curl -H"Host:www.base.com" -H"x-user-id:101" http://106.14.155.223/a
A[172.18.144.91] -> B[172.18.144.22] -> C[172.18.144.95]
```

相比场景一来说这样的好处是，客户端的域名不变，只需要通过请求来区分。

场景三：通过自定义路由规则来进行全链路灰度

有时候我们不想要自动透传且自动路由，而是希望微服务调用链上下游上的每个应用能自定义灰度规则，例如 B 应用希望控制只有满足自定义规则的请求才会路由到 B 应用这里，而 C 应用有可能希望定义和 B 不同的灰度规则，这时应该如何配置呢，场景参见如下图：



注意，最好把场景 1 和 2 中配置的参数清除掉。

第一步，需要在入口应用 A 处（最好是所有的入口应用都增加该环境变量，包括 gray 和 base）增加一个环境变量：`alicloud.service.header=x-user-id` ， `x-user-id` 是需要透传的 header，它的作用是识别该 header 并做自动透传。

注意这里不要使用 x-mse-tag，它是系统默认的一个 header，有特殊的逻辑。

第二步，在中间的 B 应用处，在 MSE 控制台配置标签路由规则（C 应用需要配置规则同理）



第三步，在 Ingress 处配置路由规则，这一步参考场景二，并采用如下配置：

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: spring-cloud-a-base
spec:
  rules:
  - host: www.base.com
    http:
      paths:
      - backend:
          serviceName: spring-cloud-a-base
          servicePort: 20001
        path: /
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/canary: 'true'
    nginx.ingress.kubernetes.io/canary-by-header: x-user-id
    nginx.ingress.kubernetes.io/canary-by-header-value: '100'
    nginx.ingress.kubernetes.io/canary-weight: '0'
  name: spring-cloud-a-gray
```

```
spec:
  rules:
    - host: www.base.com
      http:
        paths:
          - backend:
              serviceName: spring-cloud-a-gray
              servicePort: 20001
            path: /
```

第四步，测试验证，访问灰度环境，带上满足条件的 header，路由到 B 的灰度环境中。

```
curl 120.77.215.62/a -H "Host: www.base.com" -H "x-user-id: 100"
Agray[192.168.86.42] -> Bgray[192.168.74.4] -> C[192.168.86.33]
```

访问灰度环境，带上不满足条件的 header，路由到 B 的 base 环境中。

```
curl 120.77.215.62/a -H "Host: www.base.com" -H "x-user-id: 101"
A[192.168.86.35] -> B[192.168.73.249] -> C[192.168.86.33]
```

如果仅仅需要灰度对应的应用，不需要 Ingress 根据 Header 路由，那么可以去掉 Ingress Canary 配置。

访问 base A 服务（基线环境入口应用需要加上 `alicloud.service.header` 环境变量），带上满足条件的 header，路由到 B 的灰度环境中。

```
curl 120.77.215.62/a -H "Host: www.base.com" -H "x-user-id: 100"
A[192.168.86.35] -> Bgray[192.168.74.4] -> C[192.168.86.33]
```

访问 base 环境，带上不满足条件的 header，路由到 B 的 base 环境中。

```
curl 120.77.215.62/a -H "Host: www.base.com" -H "x-user-id: 101"
A[192.168.86.35] -> B[192.168.73.249] -> C[192.168.86.33]
```

4.2.2 Zuul 和 Spring Cloud Gateway 全链路灰度

当新版本发布的时候，我们希望能够控制一部分用户来使用新的版本，待验证通过后再发布给所有的用户进行使用。其中部分用户使用新版本的过程我们叫做“金丝雀发布”。

在微服务体系中一个功能可能同时需要多个应用发布，需要保证有且仅有目标用户访问新版本。下面我们介绍基于 MSE 的全链路金丝雀发布。

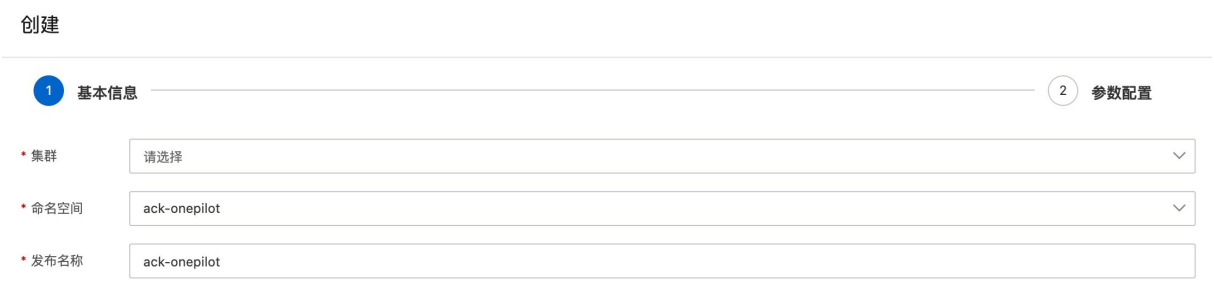
前提条件

1.开通 MSE 专业版，请参见开通 MSE 微服务治理专业版。
2.创建 ACK 集群，请参见创建 Kubernetes 集群。

操作步骤

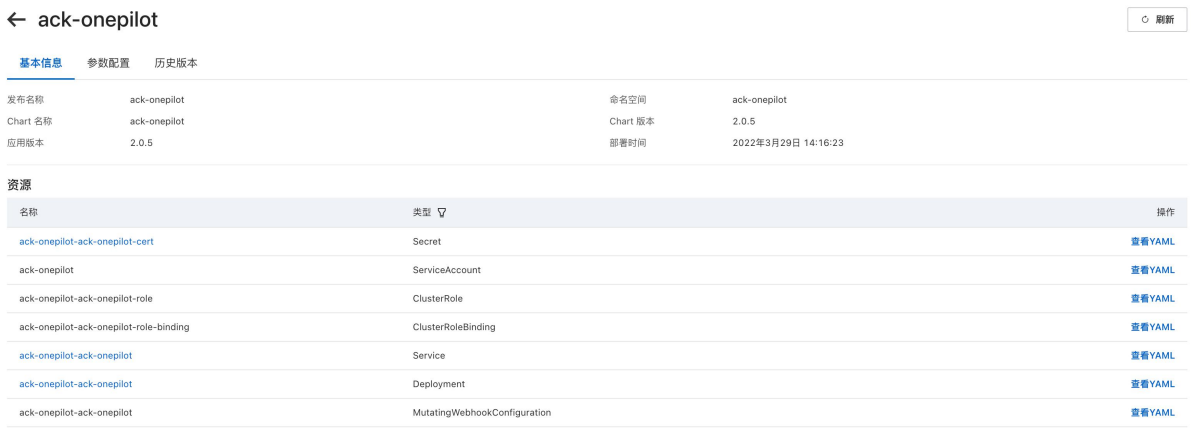
步骤一：接入 MSE 微服务治理

1.安装 ack-onepilot
a.登录容器服务控制台。
b.在左侧导航栏单击**市场 > 应用目录**。
c.在应用目录页面点击**阿里云应用**，选择**微服务**，并单击 **ack-onepilot**。
d.在 ack-onepilot页面右侧集群列表中选择集群，然后单击创建。



安装 MSE 微服务治理组件大约需要 2 分钟，请耐心等待。

创建成功后，会自动跳转到目标集群的 **Helm**页面，检查安装结果。如果出现以下页面，展示相关资源，则说明安装成功。



2.为 ACK 命名空间中的应用开启 MSE 微服务治理

a.登录 MSE 治理中心控制台，如果您尚未开通 MSE 微服务治理，请根据提示开通。

b.在左侧导航栏选择微**服务治理中心 > K8s 集群列表**。

c.在 K8s 集群列表页面搜索框列表中选择集群名称或集群 ID，然后输入相应的关键字，单击搜索图标。

d.单击目标集群操作列的**管理**。

e.在集群详情页面命名空间列表区域，单击目标命名空间操作列下的**开启微服务治理**。

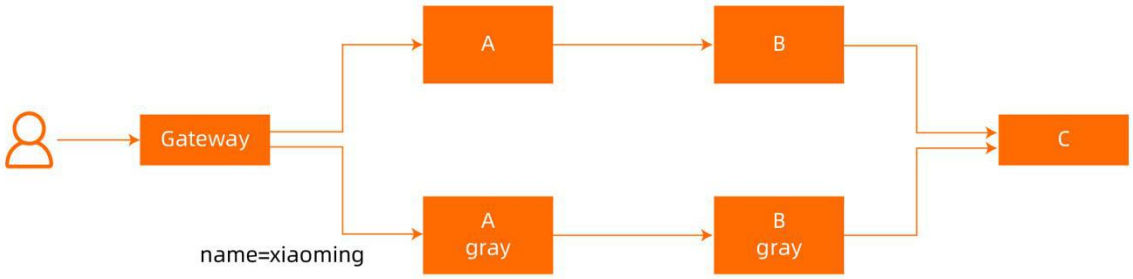
f.在开启微服务治理对话框中单击确认。



步骤二：还原线上场景

首先，我们将分别部署 spring-cloud-zuul、spring-cloud-a、spring-cloud-b、spring-cloud-c 这四个业务应用，以及注册中心 Nacos Server，模拟出一个真实的调用链路。

Demo 应用的结构图下图，应用之间的调用，既包含了 Spring Cloud 的调用，也包含了 Dubbo 的调用，覆盖了当前市面上最常用的两种微服务框架。这些应用都是最简单的 Spring Cloud 和 Dubbo 的标准用法，您也可以直接在 https://github.com/aliyun/alibabacloud-microservice- demo/tree/master/mse-simple-demo 项目上查看源码。



您可以使用 kubectl 或者直接使用 ACK 控制台来部署应用。部署所使用的 yaml 文件如下，您同样可以直接在 https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/mse-simple- demo 上获取对应的源码。

```
# 部署 Nacos Server

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nacos-server
spec:
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      annotations:
      labels:
```

```
        app: nacos-server
    spec:
      containers:
        - env:
            - name: MODE
              value: "standalone"
          image: registry.cn-shanghai.aliyuncs.com/yizhan/nacos-server:latest
          imagePullPolicy: IfNotPresent
          name: nacos-server
          ports:
            - containerPort: 8848

---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
spec:
  type: ClusterIP
  selector:
    app: nacos-server
  ports:
    - name: http
      port: 8848
      targetPort: 8848

# 部署业务应用
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-zuul
spec:
  selector:
```

```
    matchLabels:
      app: spring-cloud-zuul
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-zuul
      labels:
        app: spring-cloud-zuul
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-zuul:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-zuul
          ports:
            - containerPort: 20000


---
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-spec: slb.s1.small
    service.beta.kubernetes.io/alicloud-loadbalancer-address-type: internet
  name: zuul-slb
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 20000
  selector:
    app: spring-cloud-zuul
```

```
  type: LoadBalancer
status:
  loadBalancer: {}


---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a
spec:
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a
          ports:
            - containerPort: 20001
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-b
spec:
  selector:
    matchLabels:
      app: spring-cloud-b
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-b
          ports:
            - containerPort: 20002
          livenessProbe:
            tcpSocket:
              port: 20002
            initialDelaySeconds: 10
            periodSeconds: 30

---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-c
spec:
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-c
          ports:
            - containerPort: 20003
          livenessProbe:
            tcpSocket:
              port: 20003
            initialDelaySeconds: 10
            periodSeconds: 30
```

安装成功后，示例如下：

```
~ kubectl get svc,deploy
NAME                          TYPE           CLUSTER-IP       EXTERNAL-IP
PORT(S)          AGE
service/kubernetes        ClusterIP      192.168.0.1      <none>
443/TCP          4h40m
service/nacos-server      ClusterIP      192.168.152.138  <none>
8848/TCP         99m
service/zuul-slb          LoadBalancer   192.168.75.144   47.100.193.91
80:30767/TCP     25m
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nacos-server          1/1     1            1           99m
deployment.apps/spring-cloud-a        1/1     1            1           26m
deployment.apps/spring-cloud-b        1/1     1            1           25m
deployment.apps/spring-cloud-c        1/1     1            1           25m
deployment.apps/spring-cloud-zuul     1/1     1            1           25m
```

步骤三：部署新版本的 spring-cloud-a 应用

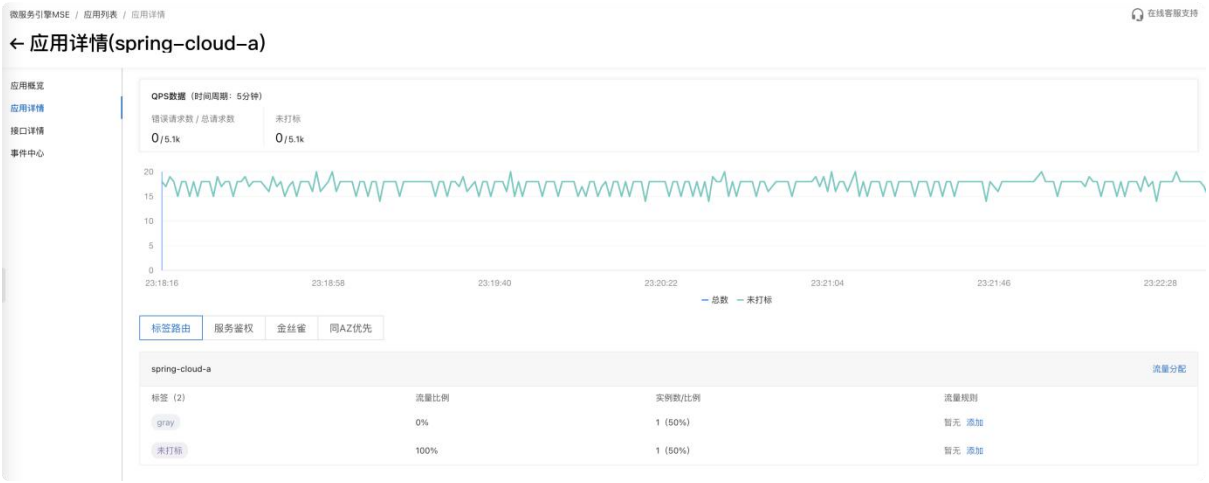
1.现在我们部署一个新版本的 spring-cloud-a 应用，对应的 yaml 文件如下：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a-gray
spec:
  selector:
    matchLabels:
      app: spring-cloud-a-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-a
      labels:
```

```
        app: spring-cloud-a-gray
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a-gray
          ports:
            - containerPort: 20001
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30
```

2.部署完成之后，登录 MSE 治理中心控制台，选择**应用列表**。

3.单击应用 spring-cloud-a **应用详情**菜单，此时可以看到，所有的流量请求都是去往 spring-cloud-a 应用未打标的版本，即稳定版本。



步骤四：配置应用 spring-cloud-a 的灰度规则并验证

1.点击页面下方的**标签路由**中的**添加**按钮，为 spring-cloud-a 应用的 gray 版本设置灰度规则。



参数	描述
框架类型	即 springcloud 还是 dubbo
服务方法	具体的接口类和方法名
条件模式	下面条件列表里多个条件间是“ 与”还是“ 或”的关系
条件列表	具体的条件，该例子中为参数 name 等于 xiaoming 字符串
是否链路传递	是否把当前应用里识别出的灰度流量的标签传递下去，本例中选择继续传递下去完成全链路的金丝雀能力的验证

更多信息请参见标签路由。

2.验证规则生效：



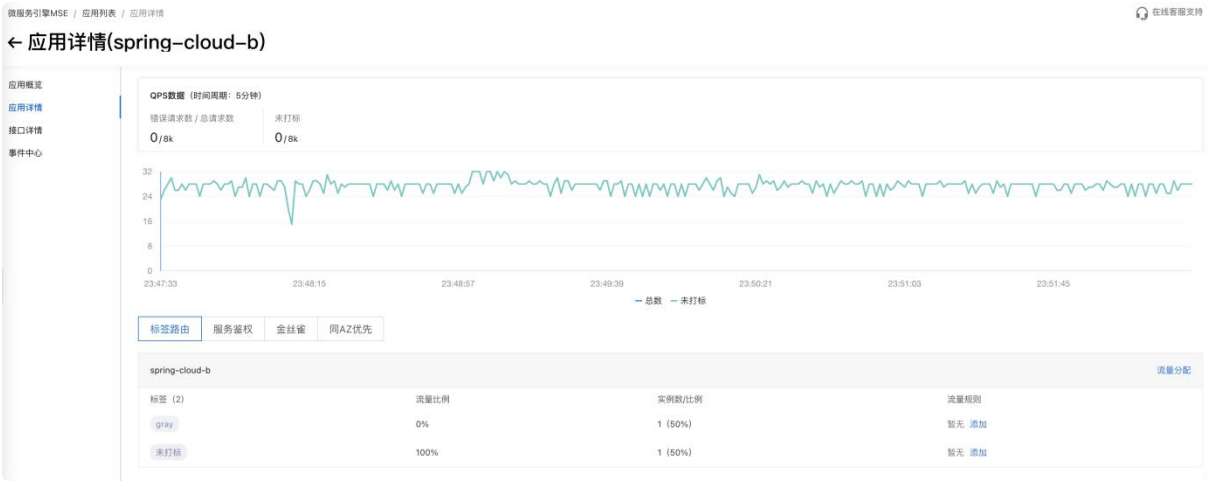
步骤五：部署新版本的 spring-cloud-b ，验证全链路金丝雀发布

1.现在我们部署一个新版本的 spring-cloud-b 应用，对应的 yaml 文件如下：

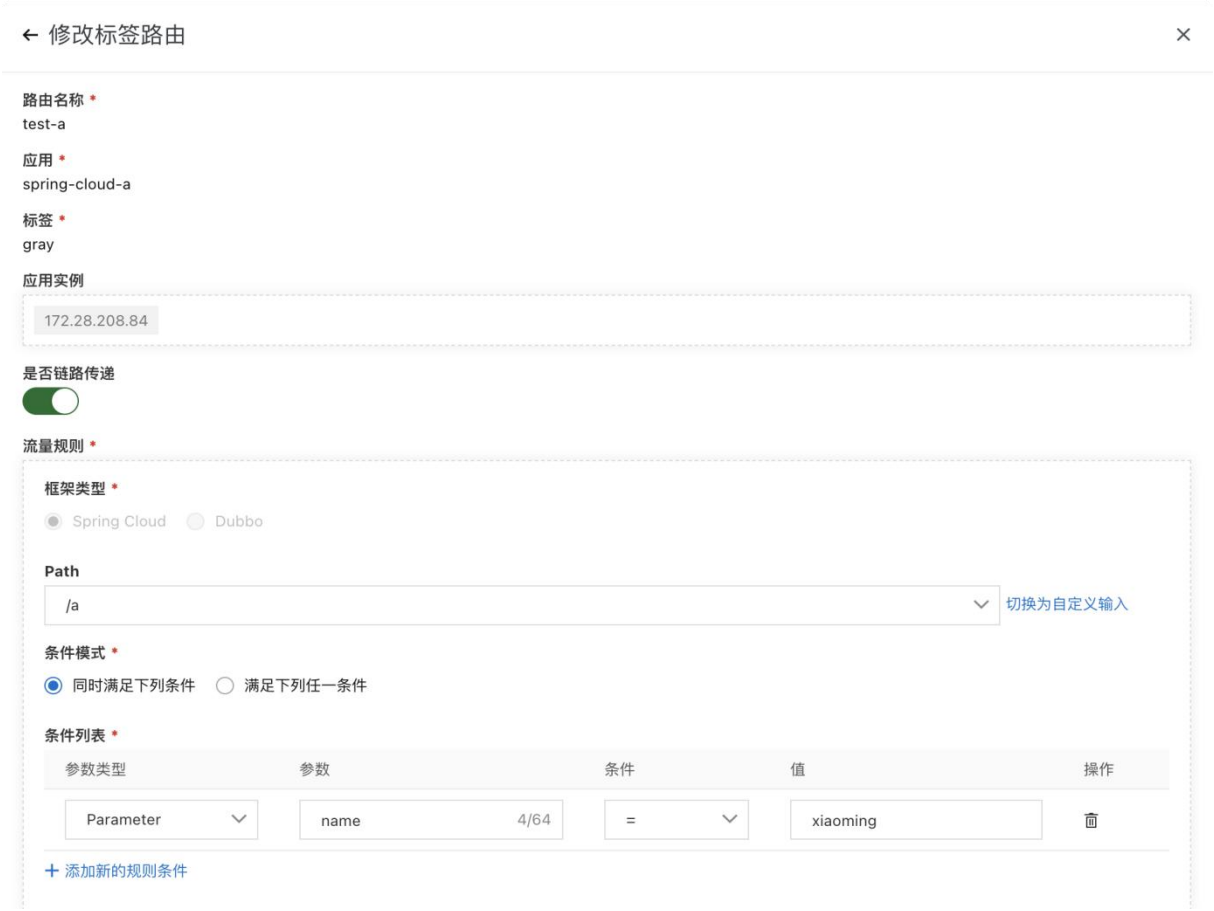
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-b-gray
spec:
  selector:
    matchLabels:
      app: spring-cloud-b-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b-gray
    spec:
      containers:
        - env:
```

```
    - name: JAVA_HOME
      value: /usr/lib/jvm/java-1.8-openjdk/jre
  image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
  imagePullPolicy: Always
  name: spring-cloud-b-gray
  ports:
    - containerPort: 20002
  livenessProbe:
    tcpSocket:
      port: 20002
    initialDelaySeconds: 10
    periodSeconds: 30
```

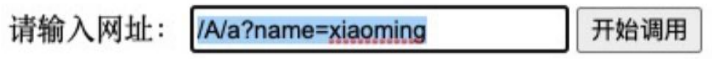
2.单击应用 spring-cloud-b **应用详情**菜单，此时可以看到，所有的流量请求都是去往 spring-cloud-b 的未打标版本，即稳定版本。



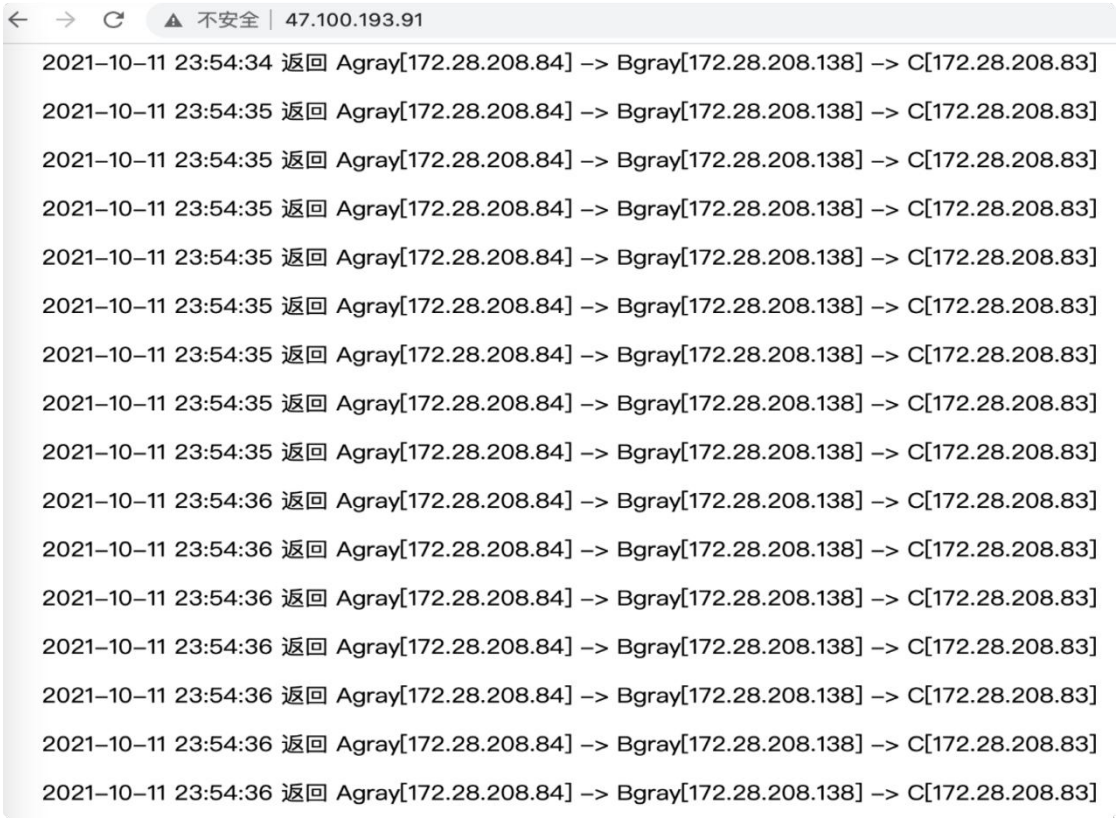
3.为了让 name=xiaoming 的灰度流量能在全链路里进行透传，且无需重复地配置规则，需在修改一下之前配置的应用 A 的灰度规则，打开**链路透传**的开关。



4.在容器服务控制台中，选择服务，找到 zuul-gateway 所对应的 slb 地址，刷新一下页面，输入 /A/a?name=xiaoming 点击开始调用。



观察发现，全链路金丝雀发布已经生效。





步骤六：验证通过后，完成发布

1.新版本的镜像验证通过之后，将 spring-cloud-a 和 spring-cloud-b 这两个 deployment 的镜像版本更新成最新的镜像版本。

2.将 spring-cloud-a-gray 和 spring-cloud-b-gray 这两个 deployment 的副本数量改成 0，灰度规则可以保留下来，这样就不用反复地配置规则。因为当找不到对应的 tag 节点时，兜底的逻辑会请求到未打标的版本，即稳定版本。

操作总结

1.整个过程是不需要修改任何代码和配置的。

2.只需要在入口应用设置规则，该流量的标签是可以“传递”下去。

3.在每个应用的调用过程中，符合金丝雀条件的流量会优先调用对应的“金丝雀”版本，如果没有“金丝雀”版本则会自动切换回“生产”版本。

4.2.3 MSE 云原生网关全链路灰度

背景

微服务架构下，有一些需求开发，涉及到微服务调用链路上的多个微服务同时发生了改动，通常每个微服务都会有灰度环境或分组来接受灰度流量，我们希望通过进入上游灰度环境的流量，也能进入下游灰度的环境中，确保 1 个请求始终在灰度环境中传递，即使这个调用链路上有一些微服务没有灰度环境，这些应用请求下游的时候依然能够回到灰度环境中。通过 MSE 提供的全链路灰度能力，可以在不需要修改任何您的业务代码的情况下，能够轻松实现上述能力。

本文主要介绍通过 MSE 云原生网关来实现全链路灰度功能。我们假设应用的架构由 MSE 云原生网关以及后端的微服务架构（Spring Cloud）来组成，后端调用链路有 3 跳，购物车（a），交易中心（b），库存中心（c），客户端通过 客户端或者是 H5 页面来访问后端服务，它们通过 Nacos 注册中心做服务发现。

前提条件

购买 MSE 云原生网关

点击购买云原生网关，购买成功后，在 MSE 控制台查看购买好的云原生网关实例：



购买 mse nacos 注册中心



开启 MSE 微服务治理

- 点击开通 MSE 微服务治理以使用全链路灰度能力。
- 访问容器服务控制台，打开应用目录，搜索 ack-onepilot ，选择命名空间 ack-onepilot ，点击创建。

部署 Demo 应用程序

将下面的文件保存到 ingress-gray.yaml 中，并执行 kubectl apply -f ingress-gray.yaml 以部署应用，这里我们将要部署 A，B，C 三个应用，每个应用分别部署一个基线版本和一个灰度版本。

有以下注意点：

1.因为需要使用mse Nacos 注册中心，所以需要将 spring.cloud.nacos.discovery.server-addr 换成业务自己的 Nacos 注册中心地址。

2.接入云原生网关的服务，如果需要使用灰度发布，需要在发布服务时在元数据信息增加版本标。在我们的例子，服务 A 是需要暴露给网关，所以发布时为基线版本添加 spring.cloud.nacos.discovery.metadata.version=base，为灰度版本添加 spring.cloud.nacos.discovery.metadata.version=gray。

```
# A 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-a
  name: spring-cloud-a
spec:
  replicas: 2
  selector:
    matchLabels:
```

```
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
        - name: spring.cloud.nacos.discovery.metadata.version
          value: base
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a
        ports:
        - containerPort: 20001
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi

# A 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  labels:
    app: spring-cloud-a-new
  name: spring-cloud-a-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a-new
  strategy:
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a-new
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: profiler.micro.service.tag.trace.enable
          value: "true"
        - name: spring.cloud.nacos.discovery.server-addr
          value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
        - name: spring.cloud.nacos.discovery.metadata.version
          value: gray
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a-new
        ports:
        - containerPort: 20001
```

```
            protocol: TCP
          resources:
            requests:
              cpu: 250m
              memory: 512Mi

# B 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-b
  name: spring-cloud-b
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b
  strategy:
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
```

```
            value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-b
          ports:
          - containerPort: 8080
            protocol: TCP
          resources:
            requests:
              cpu: 250m
              memory: 512Mi

# B 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-b-new
  name: spring-cloud-b-new
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b-new
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b-new
    spec:
      containers:
```

```
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b-new
        ports:
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi

# C 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-c
  name: spring-cloud-c
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
```

```
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-c
        ports:
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi

# C 应用 gray 版本
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: spring-cloud-c-new
  name: spring-cloud-c-new
spec:
  replicas: 2
  selector:
```

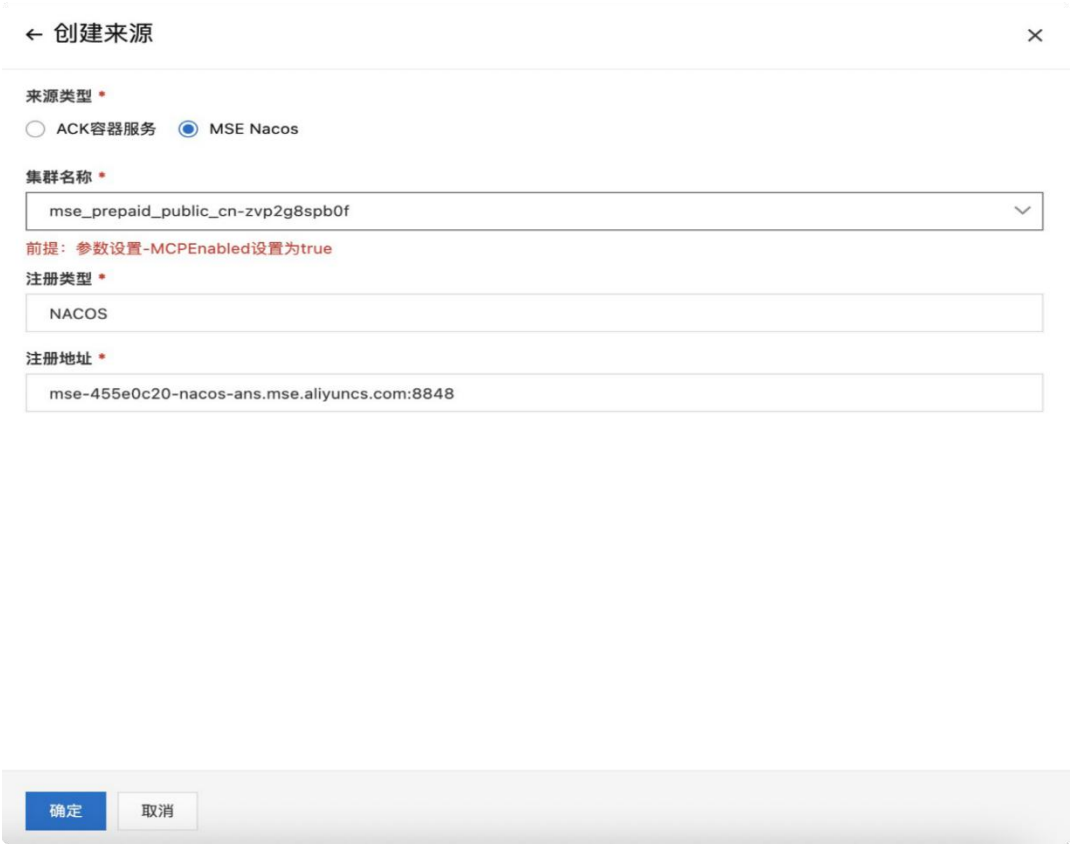
```
    matchLabels:
      app: spring-cloud-c-new
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c-new
    spec:
      containers:
      - env:
        - name: LANG
          value: C.UTF-8
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: mse-455e0c20-nacos-ans.mse.aliyuncs.com:8848
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-c-new
        ports:
        - containerPort: 8080
          protocol: TCP
        resources:
          requests:
            cpu: 250m
            memory: 512Mi
```

完成云原生网关初步配置

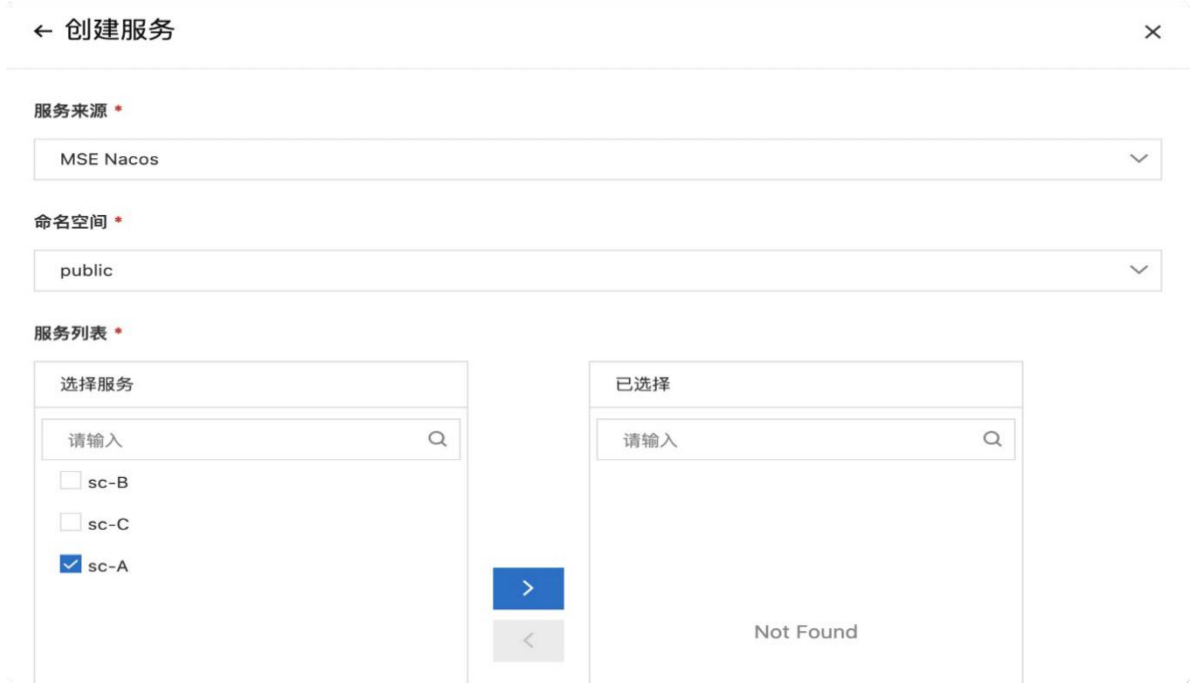
第一步，为云原生网关添加 Nacos 服务来源，服务管理，来源管理，点击创建来源。



选择 MSE Nacos 服务来源，选择需要关联的 Nacos 注册中心，点击确定。



第二步，导入要通过云原生网关暴露给外部的服务。选择服务管理，服务列表，点击创建服务。选择服务来源为 MSE Nacos，选择服务 sc-A。



点击服务 A 的策略配置，为入口服务 A 创建多版本，版本划分依据服务注册时所带的元数据信息 version（注意，这里可以是任意可以区分服务版本的标签值，取决于用户注册服务时所采用的元数据信息），创建以下两个版本 base 和 gray。

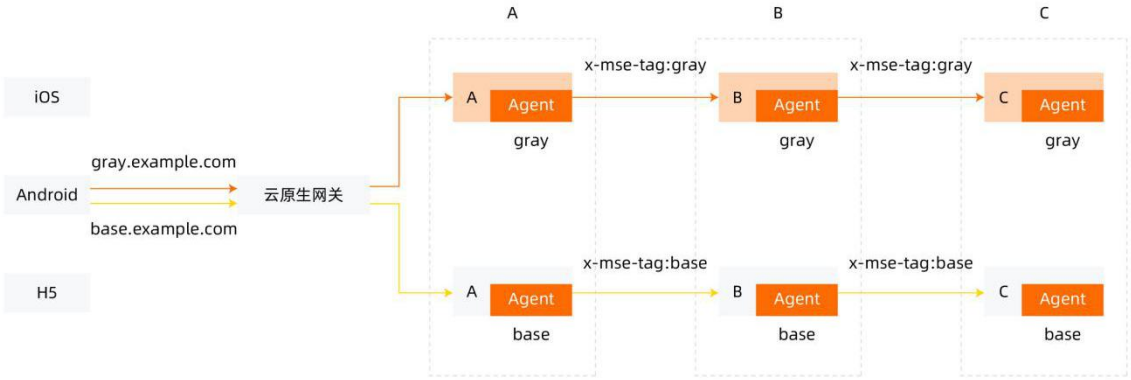


第三步，创建相关的域名，选择域名管理，创建域名，将 `www.base.example.com` 和 `www.gray. example.com` 创建好。



场景一：对经过机器的流量进行自动染色，实现全链路灰度

有时候，我们可以通过不同的域名来区分线上基线环境和灰度环境，灰度环境有单独的域名可以配置，假设我们通过访问 gray.example.com 来请求灰度环境，访问 base.example.com走基线环境。



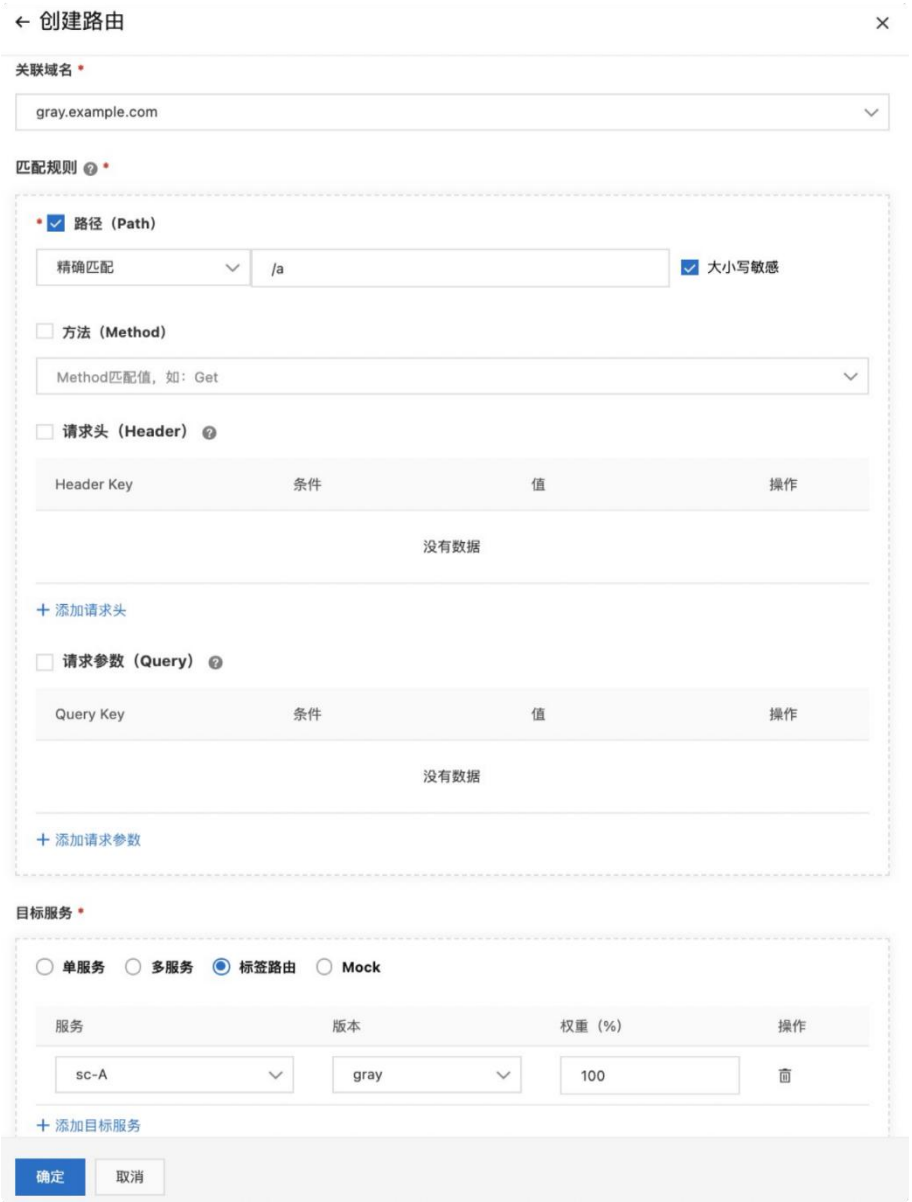
调用链路 云原生网关 -> A -> B -> C ，其中 A 可以是一个 spring-boot 的应用。

注意：入口应用 A 的 gray 和 A 的 base 环境，需要增加 profiler.micro.service.tag.trace.enable=true 这个环境变量，表示开启向后透传当前环境的标签的功能。这样，当 Ingress-nginx 路由 A 的 gray 之后，即使请求中没有携带任何 header，因为开启了此开关，所以往后调用的时候会自动添加 x-mse-tag:gray 这个 header，其中的 header 的值 gray 来自于 A 应用配置的标签信息。如果原来的请求中带有 x-mse-tag:gray 则会以原来请求中的标签优先。

在 MSE 云原生网关中创建路由规则，关联域名 base.example.com ，路由到服务 sc-A 的 base 版本中。



在 MSE 云原生网关中创建另一条路由规则，关联域名 `gray.example.com` ，路由到服务 `sc-A` 的 `gray` 版本中。



这时，我们有了如下两条路由规则。



此时，访问 base.example.com 路由到基线环境。

```
curl -H "Host: base.example.com" http://118.31.118.69/a
A[172.21.240.36] -> B[172.21.240.143] -> C[172.21.240.86]
```

此时，访问 gray.example.com 路由到灰度环境。

```
curl -H "Host: gray.example.com" http://118.31.118.69/a
Agray[172.21.240.98] -> Bgray[172.21.240.26] -> Cgray[172.21.240.144]
```

进一步的，如果入口应用 A 没有灰度环境，访问到 A 的 base 环境，又需要在 A -> B 的时候进入灰度环境，则可以，通过增加一个特殊的 header x-mse-tag 来实现，header 的值是想要去的环境的标签，例如 gray 。

```
curl -H "Host: base.example.com" -H "x-mse-tag: gray"
http://118.31.118.69/a
A[172.21.240.99] -> Bgray[172.21.240.26] -> Cgray[172.21.240.87]
```

可以看到第一跳，进入了 A 的 base 环境，但是 A->B 的时候又重新回到了灰度环境。

这种使用方式的好处是，配置简单，只需要在 MSE 云原生网关中配置好规则，某个应用需要灰度发布的时候，只需要在灰度环境中部署好应用，灰度流量自然会进入好灰度机器中，如果验证没问题，则将灰度的镜像发布到基线环境中；如果一次变更有多个应用需要灰度发布，则把他们都加入到灰度环境中即可。

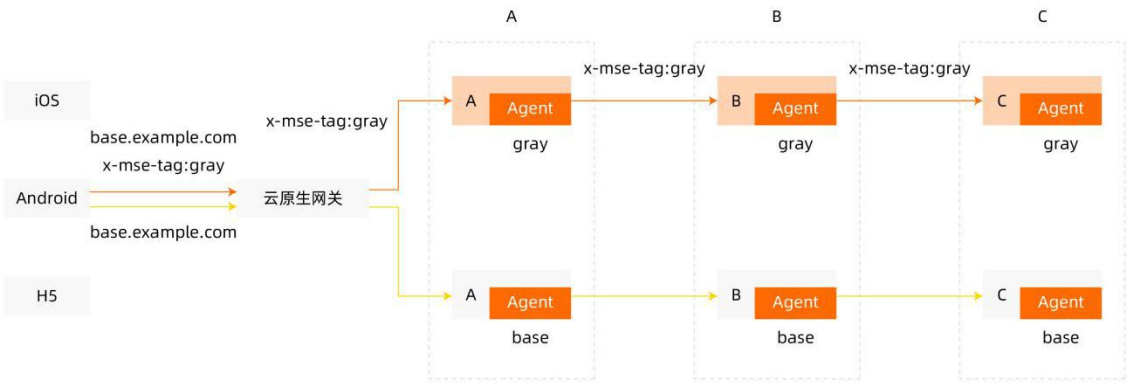
最佳实践

1.给所有灰度环境的应用打上 gray 标，基线环境的应用默认不打标。注意，对于入口服务，即暴露给云原生网关的服务，在发布服务到注册中心时需要为其添加能够表示版本的元数据信息。在我们的例子中，我们使用的是 Nacos 注册中心，所以可以通过环境变量 spring.cloud.nacos.discovery. metadata.version 设置版本。

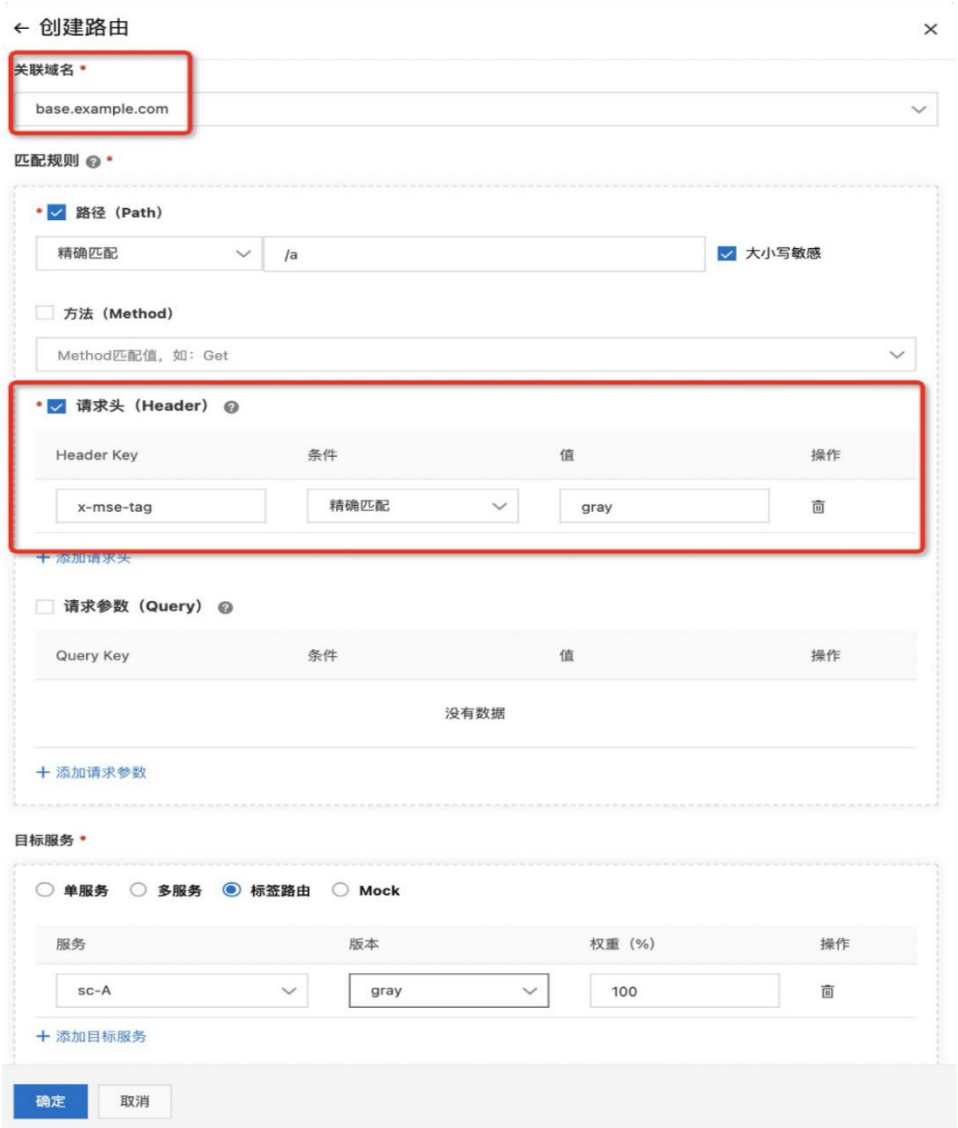
2.线上常态化引流 2%的流量进去灰度环境中。

场景二：通过给流量带上特定的 header 实现全链路灰度

有些客户端没法改写域名，希望能访问 base.example.com 通过传入不同的 header 来路由到灰度环境。例如下图中，通过添加 x-mse-tag: gray 这个 header，来访问灰度环境。



实现这个需求可以创建如下的路由规则，关联的域名与基线环境保持一致，注意此处增加了请求头相关的配置。



这时，我们有了如下两条路由规则。



此时，访问 base.example.com 路由到基线环境。

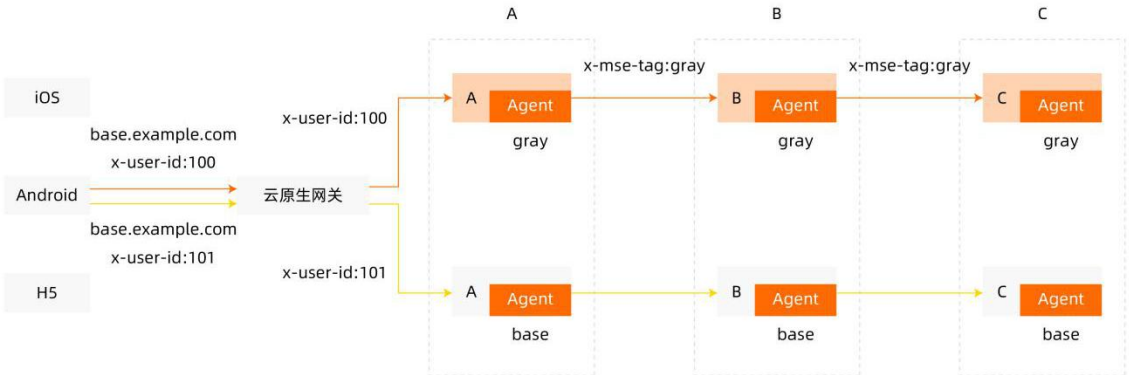
```
curl -H "Host: base.example.com" http://118.31.118.69/a
A[172.21.240.99] -> B[172.21.240.25] -> C[172.21.240.86]
```

如何访问灰度环境呢？只需要在请求中增加一个 header x-mse-tag: gray 即可。

```
curl -H "Host: base.example.com" -H "x-mse-tag: gray"
http://118.31.118.69/a
Agray[172.21.240.37] -> Bgray[172.21.240.26] -> Cgray[172.21.240.144]
```

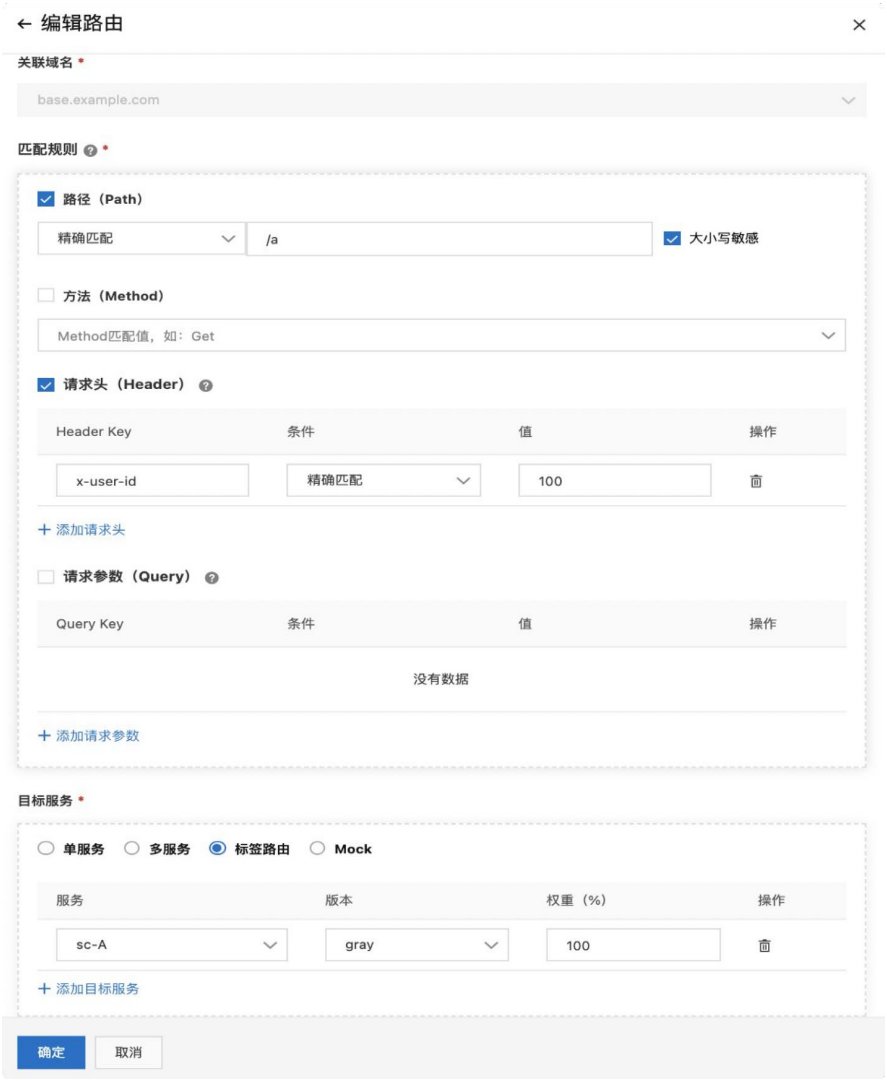
可以看到**云原生网关**根据这个 header 直接路由到了 A 的 gray 环境中。

更进一步的，还可以借助 MSE 云原生网关实现更复杂的路由，比如客户端已经带上了某个 header，想要利用现成的 header 来实现路由，而不用新增一个 header，例如下图所示，假设我们想要 x-user-id 为 100 的请求进入灰度环境。



首先，要确保入口的 A 应用添加了 profiler.micro.service.tag.trace.enable=true 这个配置。

然后，只需要修改灰度的路由配置（sc-a-gray）中的请求头部：



访问的时候带上特殊的 `header x-user-id: 100` ，满足条件进入灰度环境。

```
curl -H "Host: base.example.com" -H "x-user-id: 100"
http://118.31.118.69/a
Agray[172.21.240.98] -> Bgray[172.21.240.88] -> Cgray[172.21.240.144]
```

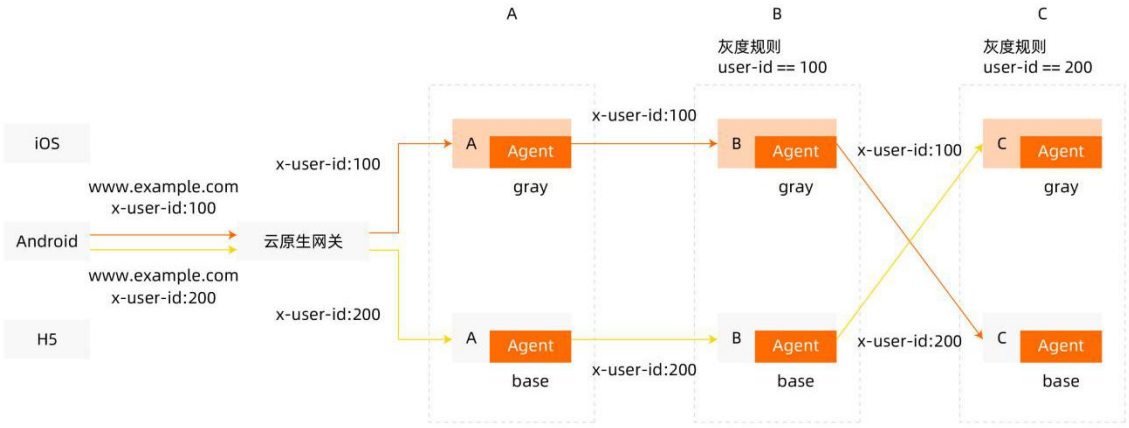
不满足条件的请求，进入基线环境:

```
curl -H "Host: base.example.com" -H "x-user-id: 101"
http://118.31.118.69/a
A[172.21.240.36] -> B[172.21.240.25] -> C[172.21.240.27]
```

这样的好处是，客户端的域名不变，只需要通过请求来区分。

场景三：通过自定义路由规则来进行全链路灰度

有时候我们不想要自动透传且自动路由，而是希望微服务调用链上下游上的每个应用能自定义灰度规则，例如 B 应用希望控制只有满足自定义规则的请求才会路由到 B 应用这里，而 C 应用有可能希望定义和 B 不同的灰度规则，这时应该如何配置呢，场景参见如下图：



- 对于入口服务 A，我们希望 x-user-id：100 的请求进入灰度版本，其他进入基线版本
- 对于后端服务 B，我们希望 x-user-id：100 的请求进入灰度版本，其他进入基线版本
- 对于后端服务 C，我们希望 x-user-id：200 的请求进入灰度版本，其他进入基线版本

第一步，删除我们在场景一和场景二为入口服务 A 添加的透传设置的环境变量 `profiler.micro.servi ce.tag.trace.enable=true` 。另外，我们需要在入口应用 A 处（最好是所有的入口应用都增加该环境变量，包括 gray 和 base） 增加一个环境变量：`alicloud.service.header=x-user-id`，`x-u ser-id` 是需要透传的 header，它的作用是识别该 header 并做自动透传。

注意这里不要使用 x-mse-tag，它是系统默认的一个 header，有特殊的逻辑。

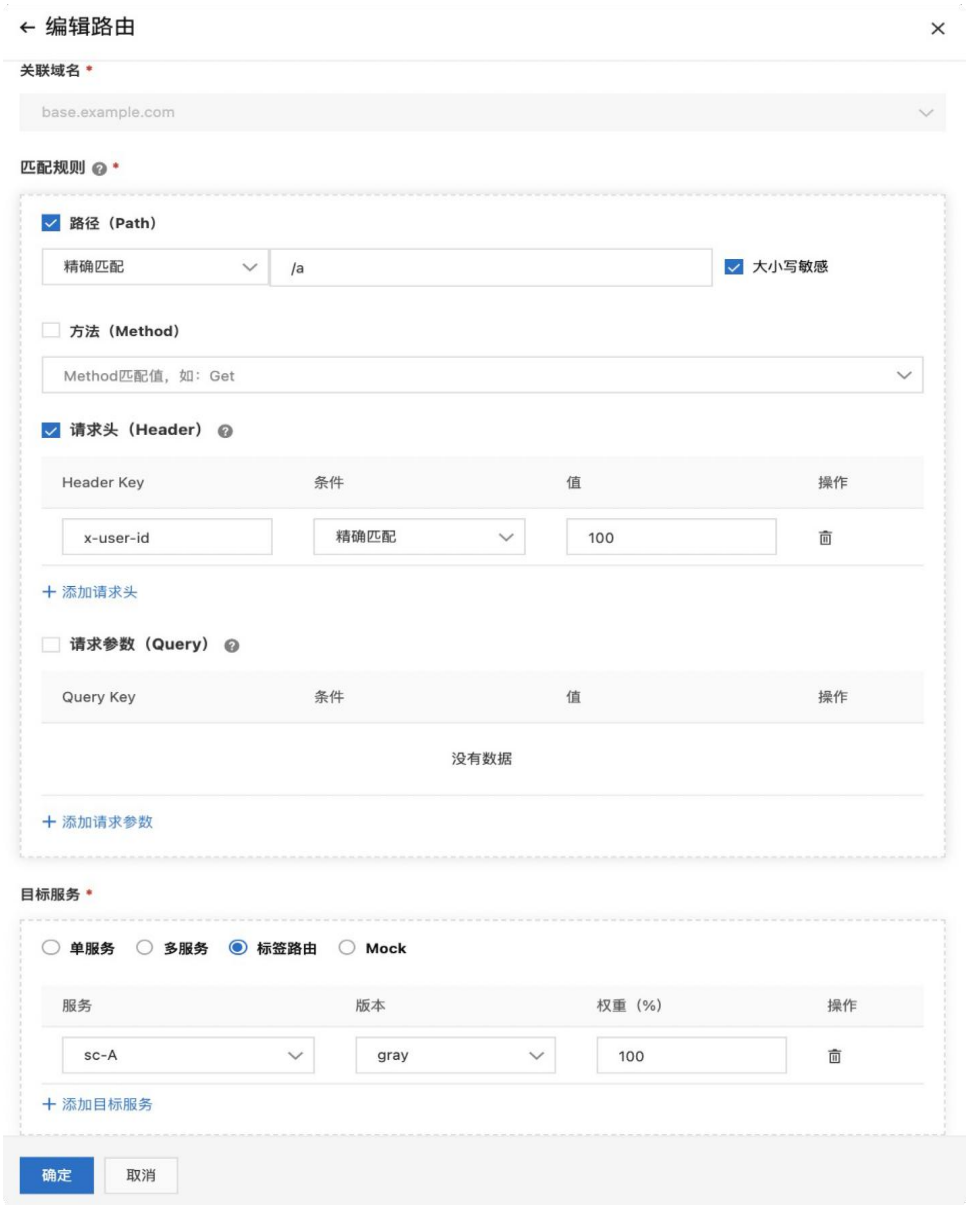
第二步，在 MSE 控制台配置为服务 B（即 sc-B）添加标签路由规则，设置只有 x-user-id:100 的请求进入灰度版本。



在 MSE 控制台配置为服务 C（即 sc-C） 添加标签路由规则，设置只有 x-user-id：200 的请求进入灰度版本。



第三步，请确保有以下的路由规则。



第四步，测试验证，带上满足入口服务的灰度条件的 header 访问网关，整条访问链路会经过 A 的灰度环境、B 的灰度环境和 C 的基线环境。

```
curl -v -H "Host: base.example.com" -H "x-user-id: 100"
http://118.31.118.69/a
Agray[10.43.0.51] -> Bgray[10.43.0.94] -> C[10.43.0.92]
```

如果仅仅需要灰度对应的应用，不需要通过 MSE 云原生网关实现 Header 路由，那么可以去掉 sc-a-gray 路由配置。接下来，访问基线（base）环境 A 应用（基线环境入口应用需要加上 alicloud.service.header 环境变量），带上满足条件的 Header，路由到 B 应用的灰度（gray）环境，C 应用的基线（base）环境。

```
curl -v -H "Host: base.example.com" -H "x-user-id: 100"
http://118.31.118.69/a
A[172.21.240.38] -> Bgray[172.21.240.88] -> C[172.21.240.27]
```

访问基线(base)环境 A 应用(基线环境入口应用需要加上 alicloud.service.header 环境变量)，带上满足条件的 Header，路由到 B 应用的基线（base）环境，C 应用的灰度（gray）环境中。

```
curl -v -H "Host: base.example.com" -H "x-user-id: 200"
http://118.31.118.69/a
A[172.21.240.101] -> B[172.21.240.25] -> Cgray[172.21.240.87]
```

访问基线（base）环境 A 应用（基线环境入口应用需要加上 alicloud.service.header 环境变量，带上不满足B、C 灰度条件的 Header，路由到 B 应用的基线（base）环境，C 应用的基线（gray）环境中。

```
curl -v -H "Host: base.example.com" -H "x-user-id: 101"
http://118.31.118.69/a
A[172.21.240.38] -> B[172.21.240.143] -> C[172.21.240.86]
```

4.2.4 全链路灰度之 RocketMQ 灰度

之前我们在动手实践中，已经通过全链路金丝雀发布这个功能来介绍了 MSE 对于全链路流量控制的场景，我们已经了解了 Spring Cloud 和 Dubbo 这一类 RPC 调用的全链路灰度应该如何实现，但是没有涉及到消息这类异步场景下的流量控制，今天我们将以上次介绍过的《全链路金丝雀发布》中的场景为基础，来进一步介绍消息场景的全链路灰度。

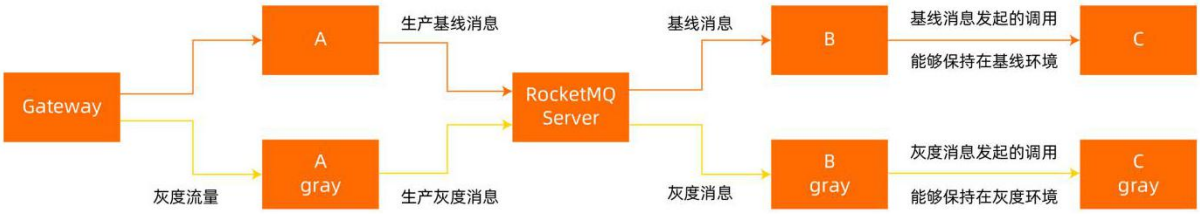
虽然绝大多数业务场景下对于消息的灰度的要求并不像 RPC 的要求得这么严格，但是在以下两个场景下，还是会对消息的全链路有一定的诉求的。

1.第一种场景是在消息消费时，可能会产生新的 RPC 调用，如果没有在消息这一环去遵循之前设定好的全链路流量控制的规则，会导致通过消息产生的这部分流量“逃逸”，从而导致全链路灰度的规则遭到破坏，导致出现不符合预期的情况。

从下图中，我们可以看到，灰度和基线环境生产出来的消息，在消费时候都是随机消费的，但是在消费过程中，产生的新的调用，还是能够回到原来所属的环境。



2.第二种场景是当消息的消费逻辑进行了修改时，这时候希望通过小流量的方式来验证新的消息消费逻辑的正确性，也会对消息的灰度有诉求。



今天我们就来实操一下第二种场景消息的全链路灰度，目前 MSE 仅支持 RocketMQ，开源版本需要在 4.5.0 及以上，阿里云商业版需要使用铂金版，且 Ons Client 在 1.8.0.Final 及以上。

如果只是想使用第一种场景，只需要给 B 应用开启全链路灰度的功能即可，不需要做额外的消息灰度相关的配置。

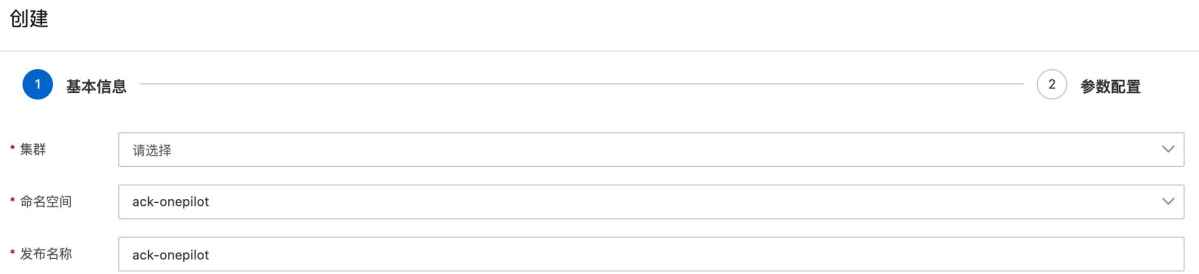
前提条件

1.开通 MSE 专业版，请参见开通 MSE 微服务治理专业版。
2.创建 ACK 集群，请参见创建 Kubernetes 集群。

操作步骤

步骤一：接入 MSE 微服务治理

1.安装 ack-onepilot
a.登录容器服务控制台。
b.在左侧导航栏单击**市场 > 应用目录**。
c.在应用目录页面点击**阿里云应用**，选择**微服务**，并单击 **ack-onepilot**。
d.在 ack-onepilot页面右侧集群列表中选择集群，然后单击创建。



安装 MSE 微服务治理组件大约需要 2 分钟，请耐心等待。

创建成功后，会自动跳转到目标集群的 **Helm**页面，检查安装结果。如果出现以下页面，展示相关资源，则说明安装成功。

2.为 ACK 命名空间中的应用开启 MSE 微服务治理

a.登录 MSE 治理中心控制台，如果您尚未开通 MSE 微服务治理，请根据提示开通。

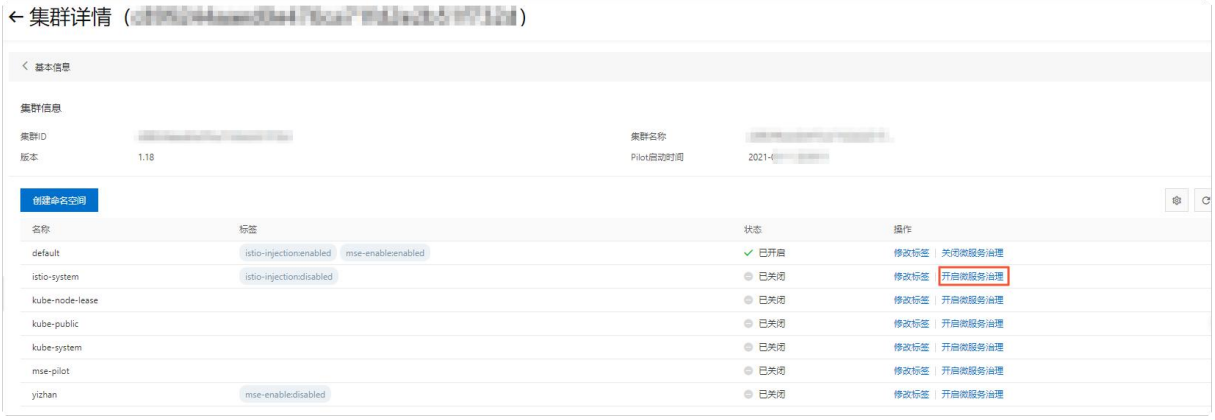
b.在左侧导航栏选择**微服务治理中心 > K8s 集群列表**。

c.在 K8s 集群列表页面搜索框列表中选择集群名称或集群 ID，然后输入相应的关键字，单击搜索图标。

d.单击目标集群操作列的**管理**。

e.在集群详情页面命名空间列表区域，单击目标命名空间操作列下的**开启微服务治理**。

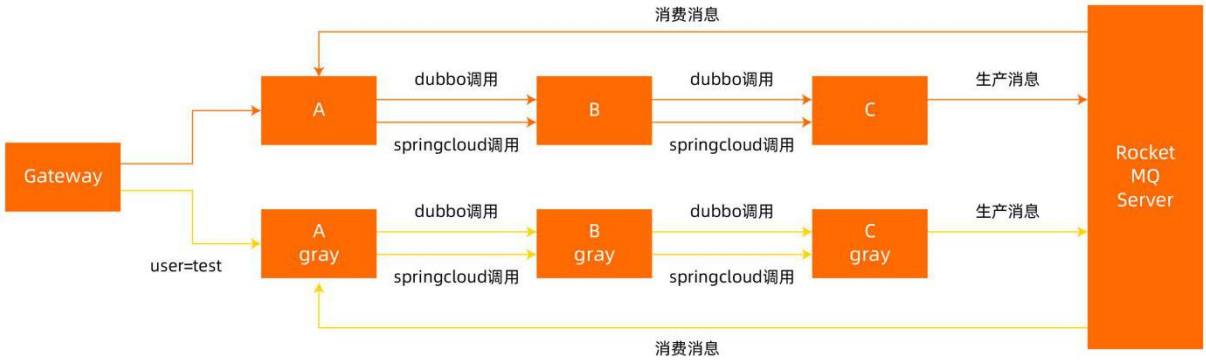
f.在开启微服务治理对话框中单击确认。



步骤二：还原线上场景

首先，我们将分别部署 spring-cloud-zuul、spring-cloud-a、spring-cloud-b、spring-cloud-c 这四个业务应用，以及注册中心 Nacos Server 和 消息服务 RocketMQ Server，模拟出一个真实的调用链路。

Demo 应用的结构图下图，应用之间的调用，既包含了 Spring Cloud 的调用，也包含了 Dubbo 的调用，覆盖了当前市面上最常用的两种微服务框架。其中 C 应用会生产出 RocketMQ 消息，由 A 应用进行消费，A 在消费消息时，也会发起新的调用。这些应用都是最简单的 Spring Cloud 、Dubbo 和 RocketMQ 的标准用法，您也可以直接在 https://github.com/aliyun/alibabacloud-microservice- demo/tree/master/mse-simple-demo 项目上查看源码。



部署之前，简单介绍一下这个调用链路。

spring-cloud-zuul 应用在收到“/A/dubbo”的请求时，会把请求转发给 spring-cloud-a ，然后 spring-cloud-a 通过 dubbo 协议去访问 spring-cloud-b，spring-cloud-b 也通过 dubbo 协议去访问 spring-cloud-c，spring-cloud-c 在收到请求后，会生产一个消息，并返回自己的环境标签和 ip。这些生产出来的消息会由 spring-cloud-a 应用消费，spring-cloud-a 应用在消费消息的时候，会通过 spring cloud 去调用 B，B 进而通过 spring cloud 去调用 C，并且将结果输出到自己的日志中。

```
当我们调用 /A/dubbo 的时候
返回值是这样 A[10.25.0.32] -> B[10.25.0.152] -> C[10.25.0.30]
同时，A 应用在接收到消息之后，输出的日志如下
2021-12-28 10:58:50.301 INFO 1 --- [essageThread_15]
c.a.mse.demo.service.MqConsumer :
topic:TEST_MQ,producer:C[10.25.0.30],invoke result:A[10.25.0.32] ->
B[10.25.0.152] -> C[10.25.0.30]
```

熟悉了调用链路之后，我们继续部署应用。

您可以使用 kubectl 或者直接使用 ACK 控制台来部署应用。部署所使用的 yaml 文件如下，您同样可以直接在 https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/mse-simpledemo 上获取对应的源码。

```
# 部署 Nacos Server

apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: nacos-server
spec:
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      annotations:
      labels:
        app: nacos-server
    spec:
      containers:
        - env:
            - name: MODE
              value: "standalone"
          image: registry.cn-shanghai.aliyuncs.com/yizhan/nacos-server:latest
          imagePullPolicy: IfNotPresent
          name: nacos-server
          ports:
            - containerPort: 8848

---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
spec:
  type: ClusterIP
  selector:
    app: nacos-server
  ports:
    - name: http
      port: 8848
```

```
      targetPort: 8848

# 部署业务应用
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-zuul
spec:
  selector:
    matchLabels:
      app: spring-cloud-zuul
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-zuul
      labels:
        app: spring-cloud-zuul
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: enable.mq.invoke
              value: 'true'
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-zuul:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-zuul
          ports:
            - containerPort: 20000

---
apiVersion: v1
kind: Service
```

```
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-spec: slb.s1.small
    service.beta.kubernetes.io/alicloud-loadbalancer-address-type: internet
  name: zuul-slb
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 20000
  selector:
    app: spring-cloud-zuul
  type: LoadBalancer
status:
  loadBalancer: {}


---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a
spec:
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
        - env:
```

```
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a
          ports:
            - containerPort: 20001
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30


---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-b
spec:
  selector:
    matchLabels:
      app: spring-cloud-b
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
```

```
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b
        ports:
          - containerPort: 20002
        livenessProbe:
          tcpSocket:
            port: 20002
          initialDelaySeconds: 10
          periodSeconds: 30

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-c
spec:
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-c
```

```
            ports:
              - containerPort: 20003
            livenessProbe:
              tcpSocket:
                port: 20003
              initialDelaySeconds: 10
              periodSeconds: 30
---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: rockectmq-broker
spec:
  selector:
    matchLabels:
      app: rockectmq-broker
  template:
    metadata:
      labels:
        app: rockectmq-broker
    spec:
      containers:
        - command:
            - sh
            - mqbroker
            - '-n'
            - 'mqnamesrv:9876'
            - '-c /home/rocketmq/rocketmq-4.5.0/conf/broker.conf'
          env:
            - name: ROCKETMQ_HOME
              value: /home/rocketmq/rocketmq-4.5.0
          image: registry.cn-shanghai.aliyuncs.com/yizhan/rocketmq:4.5.0
          imagePullPolicy: Always
```

```
            name: rockectmq-broker
            ports:
              - containerPort: 9876
                protocol: TCP
              - containerPort: 10911
                protocol: TCP
              - containerPort: 10912
                protocol: TCP
              - containerPort: 10909

---

apiVersion: apps/v1
kind: Deployment
metadata:
  name: rocketmq-name-server
spec:
  selector:
    matchLabels:
      app: rocketmq-name-server
  template:
    metadata:
      labels:
        app: rocketmq-name-server
    spec:
      containers:
        - command:
            - sh
            - mqnamesrv
          env:
            - name: ROCKETMQ_HOME
              value: /home/rocketmq/rocketmq-4.5.0
          image: registry.cn-shanghai.aliyuncs.com/yizhan/rocketmq:4.5.0
          imagePullPolicy: Always
```

```
            name: rocketmq-name-server
            ports:
              - containerPort: 9876
                protocol: TCP
              - containerPort: 10911
                protocol: TCP
              - containerPort: 10912
                protocol: TCP
              - containerPort: 10909
                protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
  name: mqnamesrv
spec:
  type: ClusterIP
  selector:
    app: rocketmq-name-server
  ports:
    - name: mqnamesrv-9876-9876
      port: 9876
      targetPort: 9876
```

安装成功后，示例如下：

```
➜ ~ kubectl get svc,deploy
NAME                          TYPE              CLUSTER-IP         EXTERNAL-IP
PORT(S)          AGE
service/kubernetes        ClusterIP         192.168.0.1          <none>
443/TCP          7d
service/mqnamesrv         ClusterIP         192.168.213.38     <none>
```

```
9876/TCP          47h
service/nacos-server     ClusterIP          192.168.24.189     <none>
8848/TCP          47h
service/zuul-slb          LoadBalancer     192.168.189.111     123.56.253.4
80:30260/TCP     47h
NAME                                        READY     UP-TO-DATE     AVAILABLE
AGE
deployment.apps/nacos-server                1/1       1              1
4m
deployment.apps/rockectmq-broker            1/1       1              1
4m
deployment.apps/rocketmq-name-server     1/1       1              1
5m
deployment.apps/spring-cloud-a              1/1       1              1
5m
deployment.apps/spring-cloud-b              1/1       1              1
5m
deployment.apps/spring-cloud-c              1/1       1              1
5m
deployment.apps/spring-cloud-zuul           1/1       1              1
5m
```

同时这里我们可以通过 zuul-slb 来验证一下刚才所说的调用链路。

```
➜ ~ curl http://123.56.253.4/A/dubbo
A[10.25.0.32] -> B[10.25.0.152] -> C[10.25.0.30]
```

步骤三：开启消息灰度功能

现在根据控制台的提示，在消息的生产者 spring-cloud-c 和消息的消费者 spring-cloud-a 都开启消息的灰度。我们直接通过 mse 的控制台开启，点击进入应用的详情页，选择“消息灰度”标签。



以看到，在未打标环境忽略的标签中，我们输入了 gray，这里意味着，带着 gray 环境标的消息，只能由 spring-cloud-a-gray 消费，不能由 spring-cloud-a 来消费。

这里需要额外说明一下，因为考虑到实际场景中，spring-cloud-c 应用和 spring-cloud-a 应用的所有者可能不是同一个人，不一定能够做到两者同时进行灰度发布同步的操作，所以在消息的灰度中，未打标环境默认的行为是消费所有消息。这样 spring-cloud-c 在进行灰度发布的时候，可以不需要强制 spring-cloud-a 应用也一定要同时灰度发布。
我们把未打标环境消费行为的选择权交给 spring-cloud-a 的所有者，如果需要实现未打标环境
不消费 c-gray 生产出来的消息，只需要在控制台进行配置即可，配置之后实时生效。

除此之外，在这里，我们也看一下控制台中的文字说明，了解在使用消息灰度时需要注意的其他事项。

- 使用此功能您无需修改应用的代码和配置。
- 消息的生产者和消息的消费者，需要同时开启消息灰度，消息的灰度功能才能生效。
- 消息类型目前只支持 RocketMQ，包含开源版本和阿里云商业版。

- 如果您使用开源 RocketMQ，则 RocketMQ Server 和 RocketMQ Client 都需要使用4.5.0及以上版本。
- 如果您使用阿里云 RocketMQ，需要使用铂金版，且 Ons Client 使用1.8.0.Final 及以上版本。

- 开启消息灰度后，MSE 会修改消息的 Consumer Group。例如原来的 Consumer Group 为 group1，环境标签为 gray，开启消息灰度后，则 group 会被修改成 group1_gray，如果您使用的是阿里云 RocketMQ ，请提前创建好 group。

- 默认使用SQL92 的过滤方式，如果您使用的开源 RocketMQ，需要在服务端开启此功能(即在 broker.conf 中配置 enablePropertyFilter=true)。

- 默认情况下，未打标节点将消费所有环境的消息，若需要指定 未打标环节点 不消费 某个标签环境生产出来的消息，请配置“未打标环境忽略的标签”，修改此配置后动态生效，无需重启应用。

步骤四：重启节点，部署新版本应用，并引入流量进行验证

首先，因为开启和关闭应用的消息灰度功能后都需要重启节点才能生效，所以首先我们需要重启一下 spring-cloud-a 和 spring-cloud-c 应用，重启的方式可以在控制台上选择重新部署，或者直接使用 kubectl 命令删除现有的 pod。



然后，继续使用 yaml 文件的方式在 K8s 集群中部署新版本的 spring-cloud-a-gray、spring-cloud-b-gray 和 spring-cloud-c-gray

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a-gray
spec:
  selector:
    matchLabels:
      app: spring-cloud-a-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a-gray
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a-gray
          ports:
            - containerPort: 20001
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: spring-cloud-b-gray
spec:
  selector:
    matchLabels:
      app: spring-cloud-b-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b-gray
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-b-gray
          ports:
            - containerPort: 20002
          livenessProbe:
            tcpSocket:
              port: 20002
            initialDelaySeconds: 10
            periodSeconds: 30

---

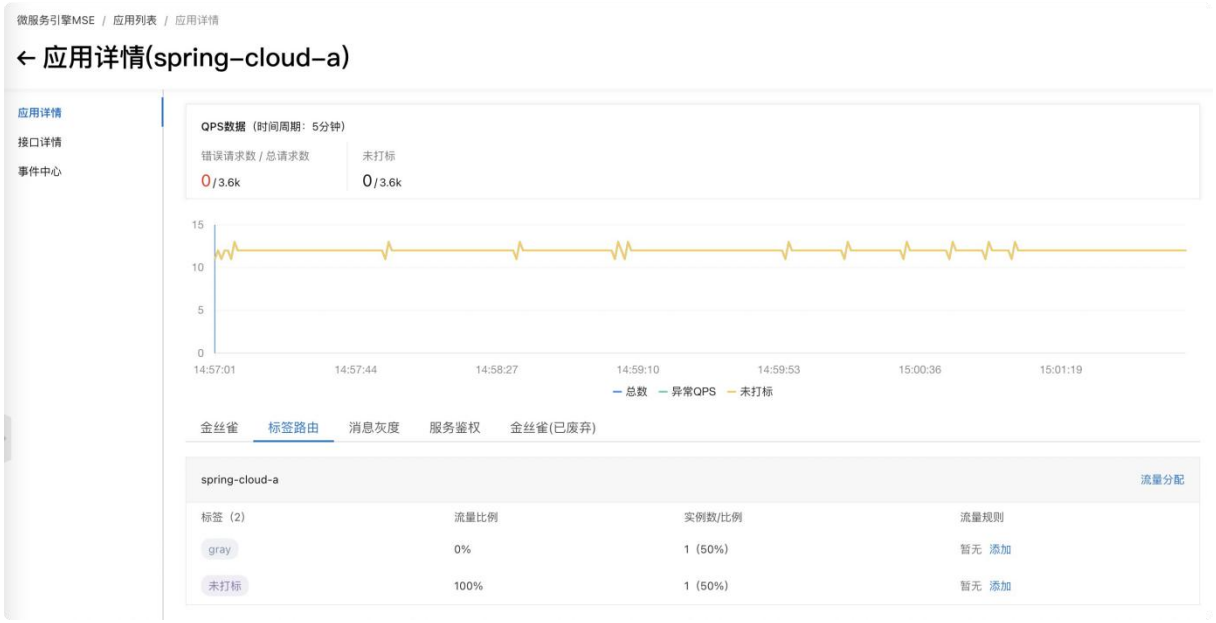
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-c-gray
```

```
spec:
  selector:
    matchLabels:
      app: spring-cloud-c-gray
  template:
    metadata:
      annotations:
        alicloud.service.tag: gray
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c-gray
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-c-gray
          ports:
            - containerPort: 20003
          livenessProbe:
            tcpSocket:
              port: 20003
            initialDelaySeconds: 10
            periodSeconds: 30
```

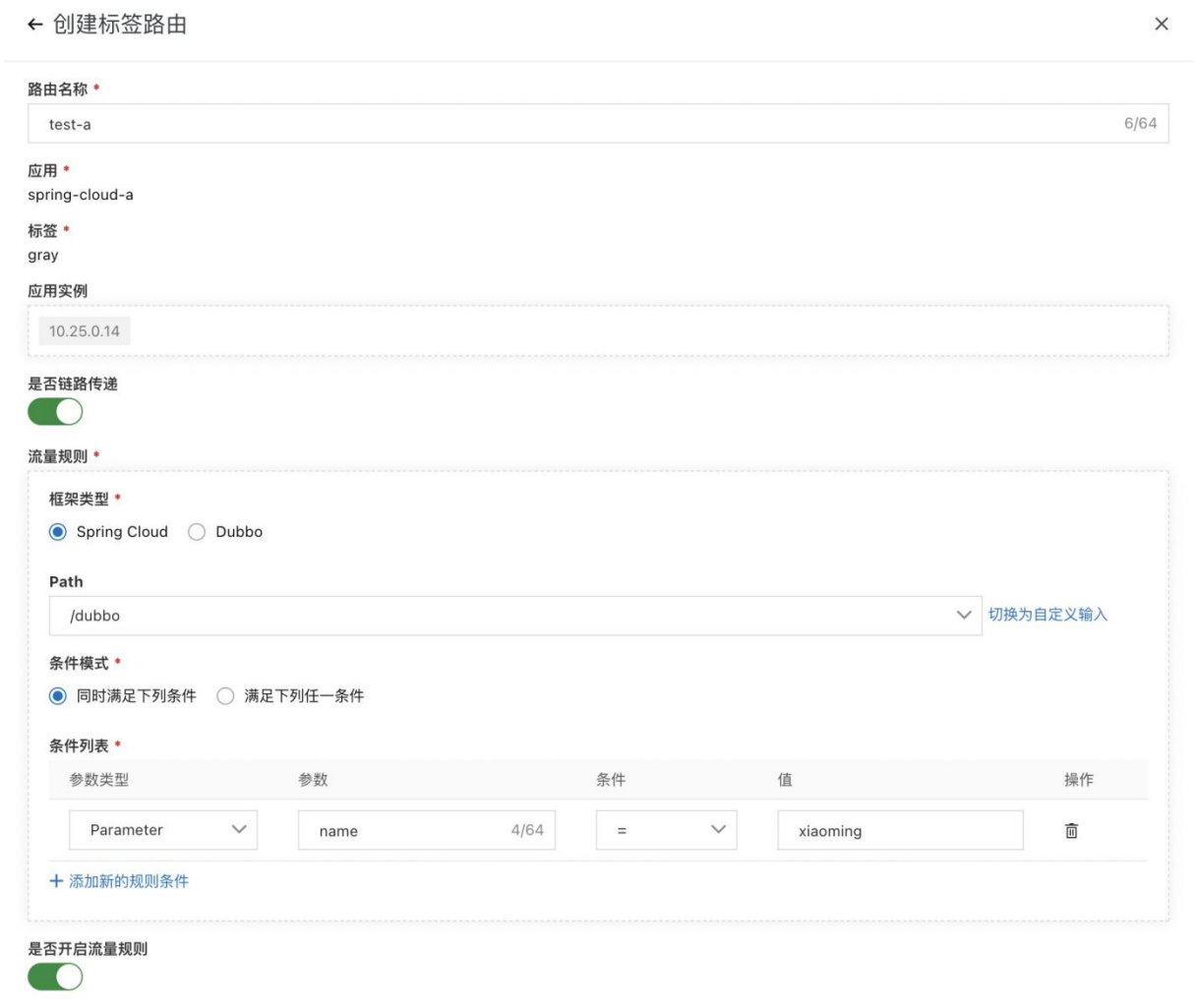
部署完成之后，我们引入流量，并进行验证

1. 登录 MSE 治理中心控制台，选择**应用列表**。

2. 单击应用 spring-cloud-a **应用详情**菜单，此时可以看到，所有的流量请求都是去往 springcloud-a 应用未打标的版本，即稳定版本。



3.点击页面下方的标签路由中的添加按钮，为 spring-cloud-a 应用的 gray 版本设置灰度规则。



参数	描述
框架类型	即 springcloud 还是 dubbo
服务方法	具体的接口类和方法名
条件模式	下面条件列表里多个条件间是“与”还是“或”的关系
条件列表	具体的条件，该例子中为参数 name 等于 xiaoming 字符串
是否链路传递	是否把当前应用里识别出的灰度流量的标签传递下去，本例中选择继续传递下去完成全链路的金丝雀能力的验证

4.发起流量调用，我们通过 zuul-slb，分别发起流量调用，并查看灰度的情况。



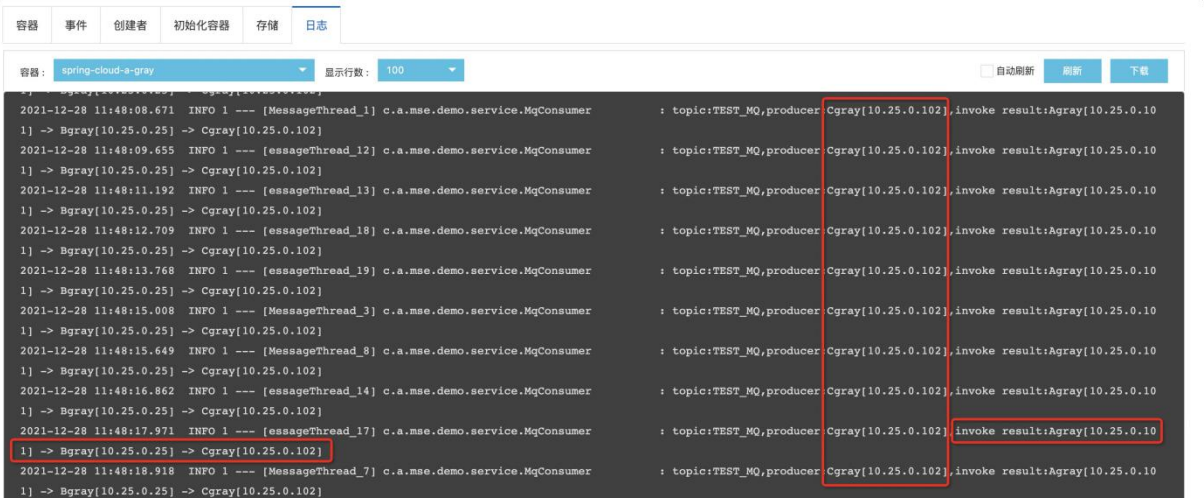


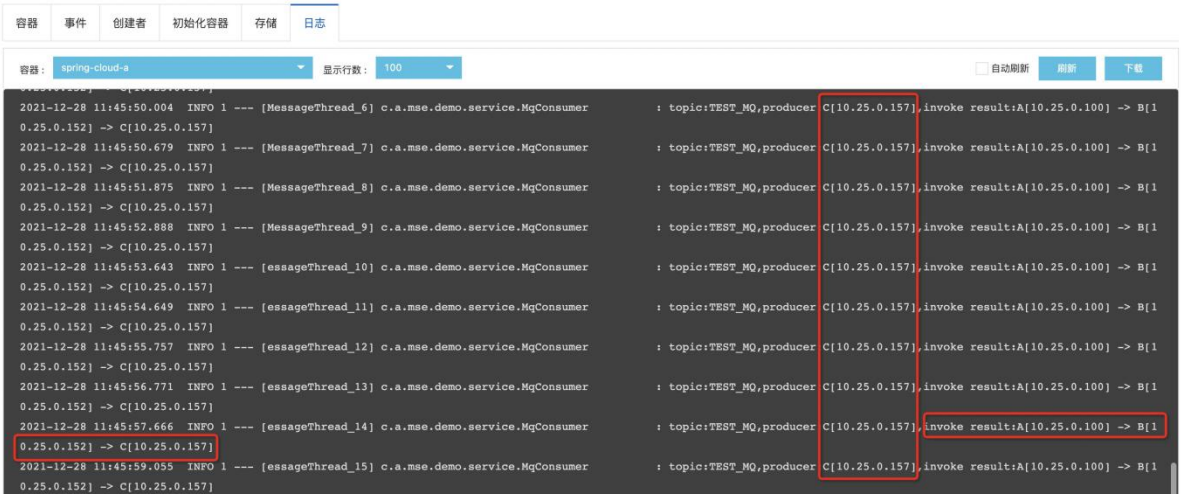


我们通过 spring-cloud-a 和 spring-cloud-a-gray 的日志去查看消息消费的情况。可以看到，消息的灰度功能已经生效，spring-cloud-a-gray 这个环境，只会消费带有 gray 标的消息。spring-cloud-a 这个环境，只会消费未打标的流量生产出来的消息。

在截图中我们可以看见，spring-cloud-a-gray 环境输出的日志 topic:TEST_MQ，producer:Cgray [10.25.0.102] ，invoke result: Agray[10.25.0.101] -> Bgray[10.25.0.25] -> Cgray[10.25.0.102]，spring-cloud-a-gray 只会消费 Cgray 生产出来的消息，而且消费消息过程中发起的 Spring Cloud 调用，结果也是 Agray[10.25.0.101] -> Bgray[10.25.0.25] ->Cgray[10.25.0.102]，即在灰度环境闭环。

而 spring-cloud-a 这个环境，输出的日志为 topic:TEST_MQ,producer:C[10.25.0.157],invoke result:A[10.25.0.100] -> B[10.25.0.152] -> C[10.25.0.157]，只会消费 C 的基线环境生产出来的消息，且在这个过程中发起的 Spring Cloud 调用，也是在基线环境闭环。



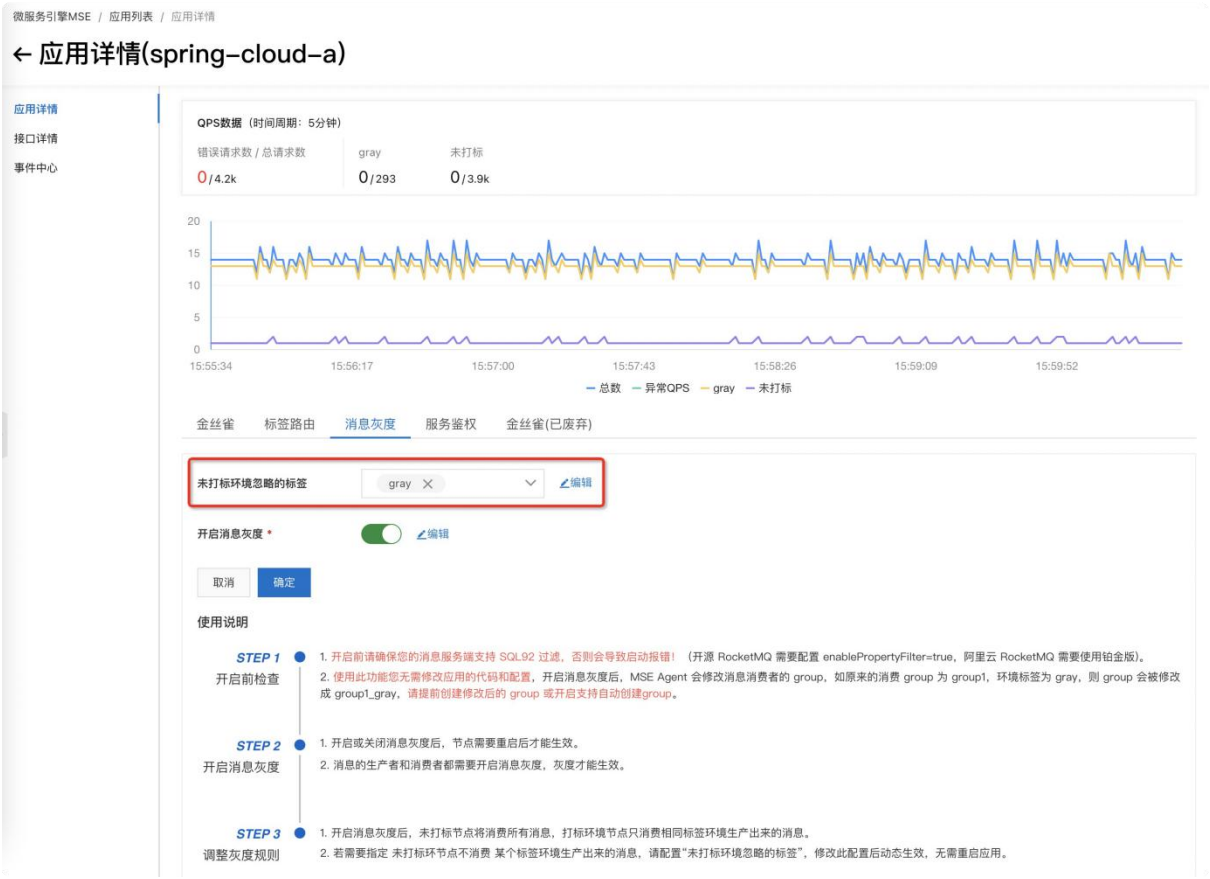


步骤五：调整消息的标签过滤规则，并进行验证

因为考虑到实际场景中，spring-cloud-c 应用和 spring-cloud-a 应用的所有者可能不是同一个人，不一定能够做到两者同时进行灰度发布同步的操作，所以在消息的灰度中，未打标环境默认的行为是消费所有消息。这样 spring-cloud-c 在进行灰度发布的时候，可以不需要强制 spring-cloud-a 应用也一定要同时灰度发布，且使用相同的环境标。

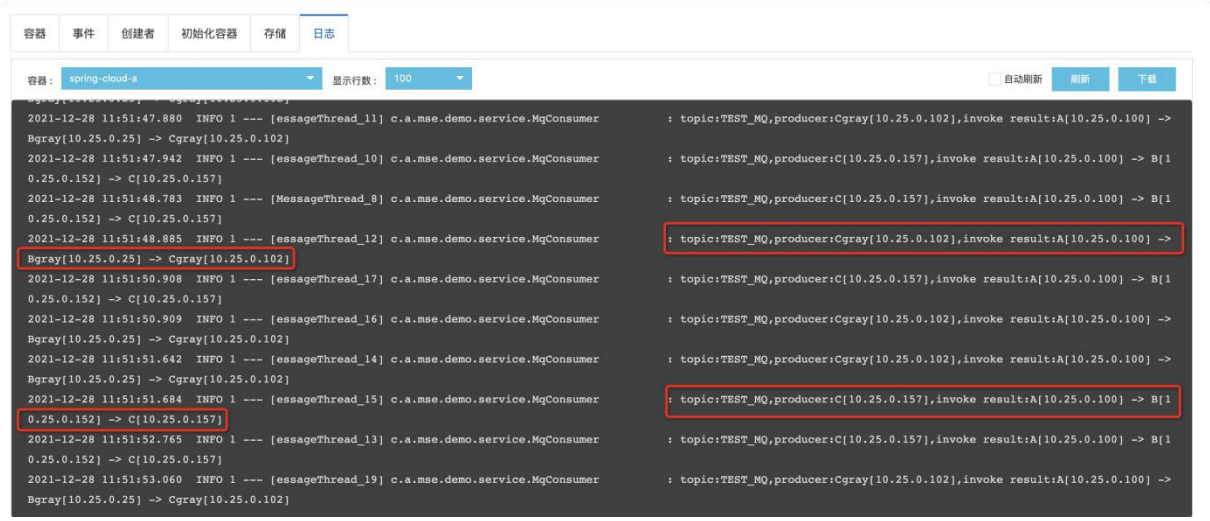
spring-cloud-a 在消费时候，未打标环境的行为的选择权是交给 spring-cloud-a 的所有者，如果需要实现未打标环境不消费 c-gray 生产出来的消息，只需要在控制台进行配置即可，配置之后实时生效。

1.调整 spring-cloud-a 未打标环境的过滤规则。比如这里我们要选择未打标环境不再消费 gray 环境生产出来的消息，只需要在“未打标环境忽略的标签”里面选择 gray，然后点击确定即可。



2.调整规则之后，规则是可以动态地生效，不需要进行重启的操作，我们直接查看 spring-cloud-a 的日志，验证规则调整生效。

从这个日志中，我们可以看到，此时基线环境可以同时消费 gray 和基线环境生产出来的消息，而且在消费对应环境消息时产生的 Spring Cloud 调用分别路由到 gray 和基线环境中。



操作总结

1.全链路消息灰度的整个过程是不需要修改任何代码和配置的。

2.目前仅支持 RocketMQ，Client 版本需要在 4.5.0 之后的版本。RocketMQ Server 端需要支持 SQL92 规则过滤，即开源 RocketMQ 需要配置 enablePropertyFilter=true，阿里云 RocketMQ 需要使用铂金版。

3. 开启消息灰度后，MSE Agent 会修改消息消费者的 group，如原来的消费 group 为 group1，环境标签为 gray，则 group 会被修改成 group1_gray，如果使用的是阿里云 RocketMQ，需要提前创建好修改后的 group。

4.开启和关闭消息灰度后，应用需要重启才能生效；修改未打标环境忽略的标签功能可以动态生效，不需要重启。

4.2.5 使用Jenkins CI/CD 实现金丝雀发布

前提条件

安装 Jenkins

- 本文假设您已经完成安装 Jenkins，要在 ACK 集群中安装 Jenkins，请参考快速搭建 Jenkins 环境并完成流水线作业这篇文章。

开启 MSE 微服务治理

- 点击开通 MSE 微服务治理专业版以使用金丝雀发布能力。
- 访问 MSE 控制台，在 K8s 集群列表中选择相应集群，点击管理，选择 default 命名空间，点击开启服务治理能力。
- 访问容器服务控制台，打开应用目录，搜索 ack-onepilot ，选择对应集群，点击创建。

部署 Demo 程序

将下面的文件保存到 mse-canary-demo-deployment-set.yaml 中，并执行 kubectl apply -f mse-canary-demo-deployment-set.yaml 以部署应用，这里我们将要部署 A，B，C 三个应用，以及注册中心 nacos，以及压力来源 spring-cloud-zuul，流量从 spring-cloud-zuul -> A -> B -> C，其中 spring-cloud-zuul 中默认有 100 qps 正常流量，而另外有 10 qps 带有 x-mse-tag:gray 这个特殊 header，这个 header 是 MSE 内置的灰度 header，只要带上 x-mse-tag:gray 这个 header，在开启 MSE 微服务治理并且 安装 Agent 之后会自动路由到下游带有 gray 标签的节点上。根据需要，这个 gray 的值也可以替换成其他机器上打的标签。

```
#入口 zuul 应用
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-zuul
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-cloud-zuul
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-zuul
      labels:
        app: spring-cloud-zuul
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: LANG
              value: C.UTF-8
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-zuul:1.0.1
          imagePullPolicy: Always
          name: spring-cloud-zuul
          ports:
            - containerPort: 20000


# A 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: spring-cloud-a
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-a
        ports:
        - containerPort: 20001
        livenessProbe:
          tcpSocket:
            port: 20001
          initialDelaySeconds: 10
          periodSeconds: 30


# B 应用 base 版本
---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: spring-cloud-b
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-b
  strategy:
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-b
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 20002
          initialDelaySeconds: 10
          periodSeconds: 30


# C 应用 base 版本
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: spring-cloud-c
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
        imagePullPolicy: Always
        name: spring-cloud-c
        ports:
        - containerPort: 8080
        livenessProbe:
          tcpSocket:
            port: 20003
          initialDelaySeconds: 10
          periodSeconds: 30


# Nacos Server
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nacos-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      labels:
        app: nacos-server
    spec:
      containers:
      - env:
        - name: MODE
          value: standalone
        image: nacos/nacos-server:latest
        imagePullPolicy: Always
        name: nacos-server
      dnsPolicy: ClusterFirst
      restartPolicy: Always

# zuul 网关开启 SLB 暴露展示页面
---
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-spec: slb.s1.small
  name: zuul-slb
spec:
  ports:
    - port: 80
```

```
    protocol: TCP
    targetPort: 20000
  selector:
    app: spring-cloud-zuul
  type: LoadBalancer
```

部署成功后，在 MSE 控制台中观察 spring-cloud-a 应用的流量，确认流量都打到未达标的节点中，并没有灰度节点的流量。



配置镜像仓库的推送权限

本实践需要将源码打包后执行镜像推送，请确保 Jenkins 有权限推送到镜像仓库中，配置方法请参考快速搭建 Jenkins 环境并完成流水线作业这篇文章中的“使用kaniko 构建和推送容器镜像”部分。简单而言，需要创建一个 secret：

```
kubectl create secret generic jenkins-docker-cfg -n jenkins --from
file=/root/.docker/config.json
```

其中上述的 demo 中 config.json 内容为：

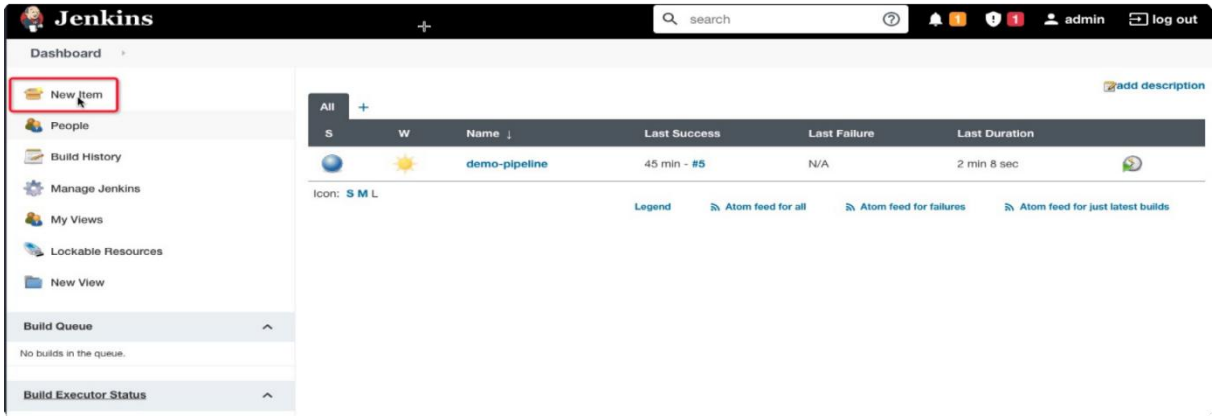
```
{
        "auths": {
                "registry.cn-shanghai.aliyuncs.com": {
                        "auth":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

```
                }
        }
}
```

动手实践

在 Jenkins 中创建金丝雀发布流水线

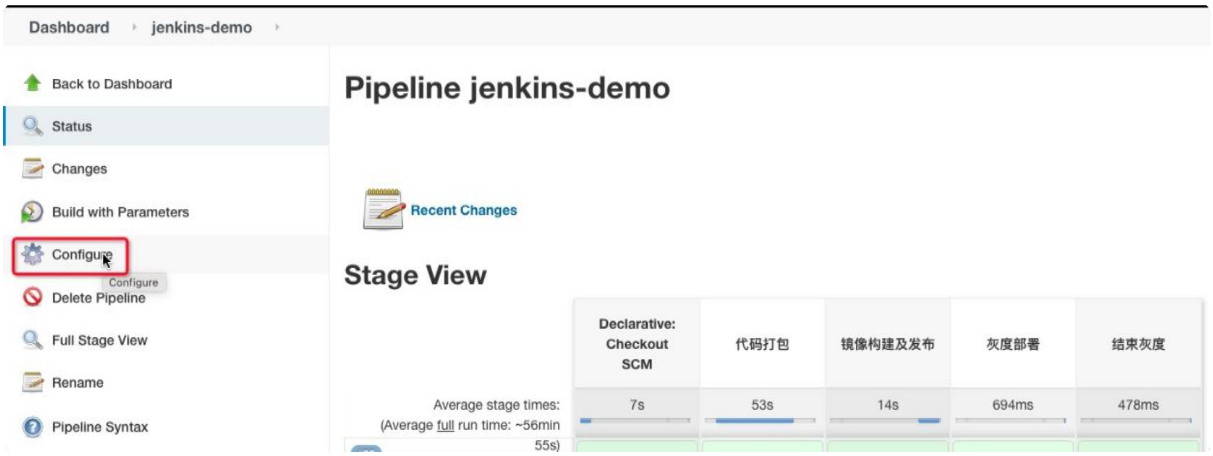
点击 New Item。



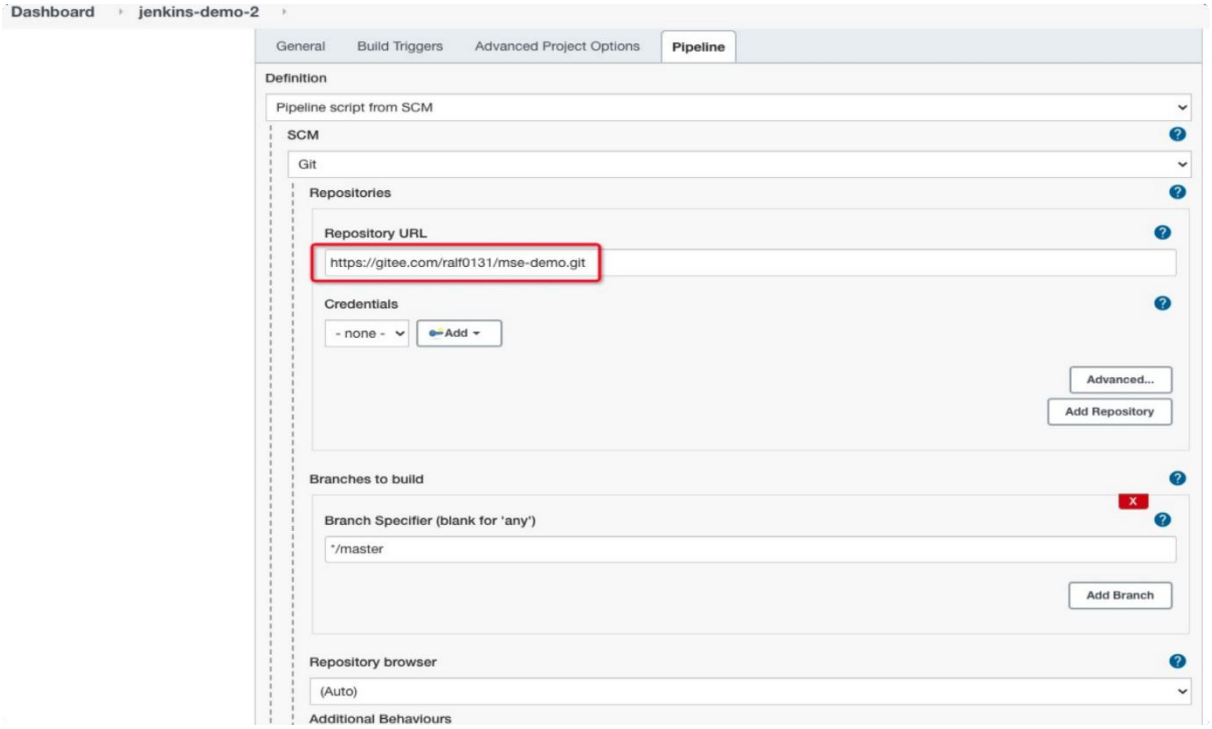
填写名字，选择 Pipeline。



创建成功后，点击配置。



点击 Pipeline，选择 Pipeline script from SCM。



填写 Git 仓库的 URL，这里选择 Gitee 作为样例，代码仓库的地址是：https://gitee.com/ralf0131/mse-demo。

注意：阿里云机器无法拉取 Github 中的源码，暂时使用 Gitee 做为替代。

同时 Script Path 填写为 Jenkinsfile，这个 Jenkinsfile 是 Jenkins 流程编排文件，存放在 Git 仓库中，Jenkins 会读取这个文件创建相应的 Pipeline，点击 Save 保存。
实际的 Jenkinsfile 内容如下：

```
#!groovy
pipeline {

    // 定义本次构建使用哪个标签的构建环境，本示例中为 “slave-pipeline”
    agent{
        node{
          label 'slave-pipeline'
        }
    }

    //常量参数，初始确定后一般不需更改
    environment{
        IMAGE = sh(returnStdout: true,script: 'echo registry.$image_region.aliyuncs.co
m/$image_namespace/$image_reponame:$image_tag').trim()
        BRANCH =  sh(returnStdout: true,script: 'echo $branch').trim()
    }
    options {
        //保持构建的最大个数
        buildDiscarder(logRotator(numToKeepStr: '10'))
    }

    parameters {
        string(name: 'image_region', defaultValue: 'cn-shanghai')
        string(name: 'image_namespace', defaultValue: 'yizhan')
        string(name: 'image_reponame', defaultValue: 'spring-cloud-a')
        string(name: 'image_tag', defaultValue: 'gray')
        string(name: 'branch', defaultValue: 'master')
        string(name: 'number_of_pods', defaultValue: '2')
    }

    //pipeline 的各个阶段场景
    stages {

        stage('代码打包') {
            steps{
```

```
                container("maven") {
                    echo "镜像构建....."
                    sh "cd A && mvn clean package"
                }

            }
        }


        stage('镜像构建及发布'){
          steps{
              container("kaniko") {
                  sh "kaniko -f `pwd`/A/Dockerfile -c `pwd`/A --destination=${IMAGE}
 --skip-tls-verify"
              }
          }
        }



        stage('灰度部署') {
            steps{
                container('kubectl') {
                    echo "灰度部署....."
                    sh "cd A && sed -i -E \"s/${env.image_reponame}:.+/${env.ima
ge_reponame}:${env.image_tag}/\" A-gray-deployment.yaml"
                    sh "cd A && sed -i -E \"s/replicas:.+/replicas: ${env.number_of_
pods}/\" A-gray-deployment.yaml"
                    sh "kubectl apply -f A/A-gray-deployment.yaml -n default"
                }
            }
        }

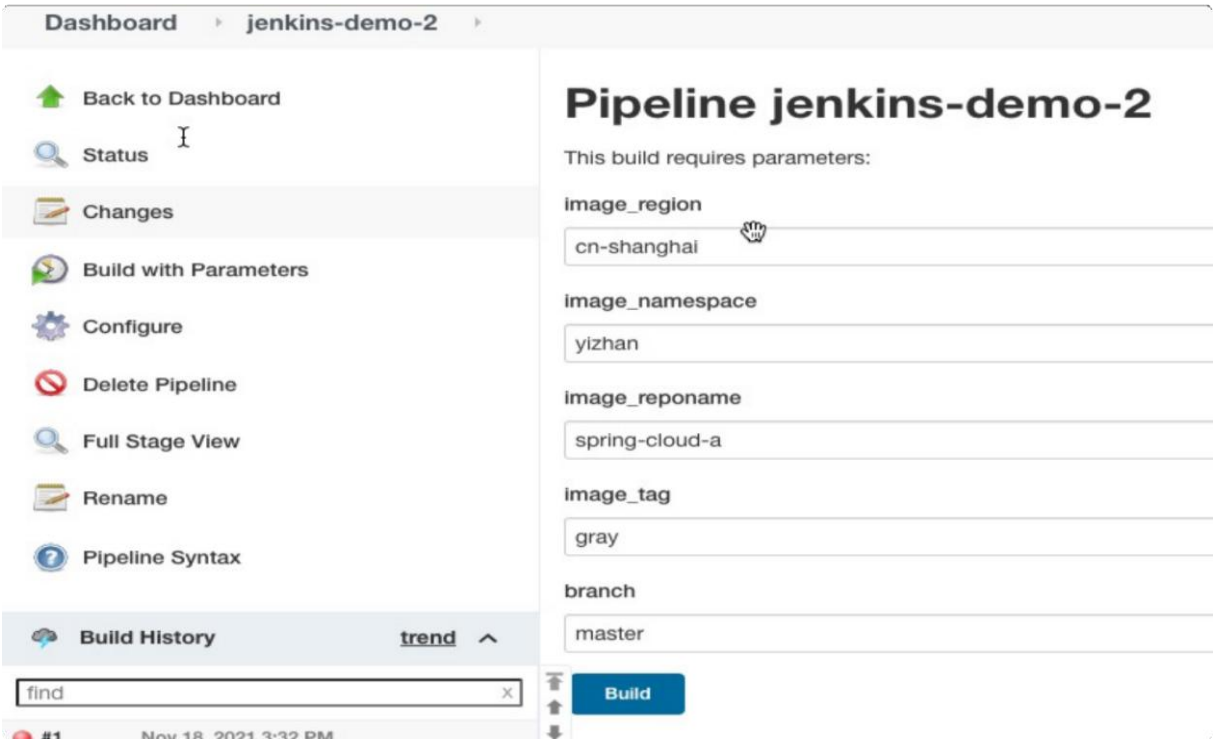
        stage('结束灰度') {
            input {
```

```
                    message "请确认是否全量发布"
                    ok "确认"
                    parameters {
                        string(name: 'continue', defaultValue: 'true', description: 'true 为全量发布，其他为回滚')
                    }
                }
                steps{
                    script {
                        env.continue = sh (script: 'echo ${continue}', returnStdout: true).trim()
                        if (env.continue.equals('true')) {
                            container('kubectl') {
                                echo "全量发布....."
                                sh "cd A && sed -i -E \"s/${env.image_reponame}:.+/${env.image_reponame}:${env.image_tag}/\" A-deployment.yaml"
                                sh "cd A && sed -i -E \"s/replicas:.+/replicas: ${env.number_of_pods}/\" A-deployment.yaml"
                                sh "kubectl apply -f A/A-deployment.yaml -n default"
                            }
                        } else {
                            echo '回滚'
                        }
                        container('kubectl') {
                            sh "kubectl delete -f A/A-gray-deployment.yaml -n default"
                        }
                    }
                }
            }
        }
```

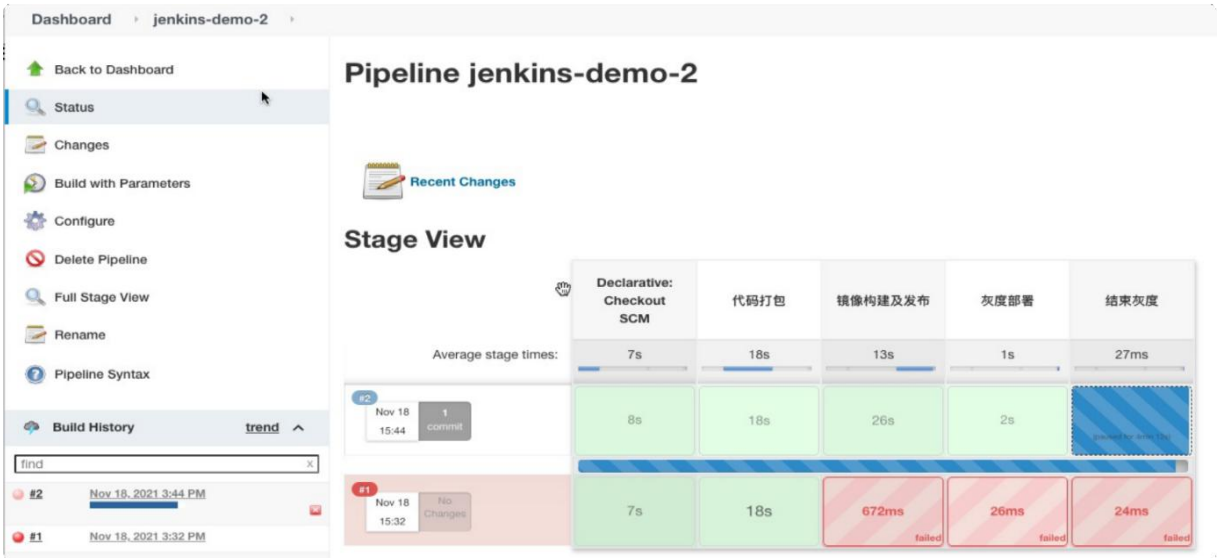
您可以参考以上的文件填写好指定的参数，当然您可以根据您的需求编写 Jenkinsfile ，并上传至git 的指定路径下（流水线中指定的脚本路径）。

执行 Jenkins 构建

点击流水线的 build 执行第一次构建，第一次 build 因为需要从 Git 仓库拉取配置并初始化流水线，所以会报错，再次执行 Build with Parameters，这时候会生成相关的参数，填写相关的参数，再次执行构建。



查看部署状态，发现代码打包，镜像构建，灰度部署阶段都已经完成，在结束灰度阶段等待手工确认。

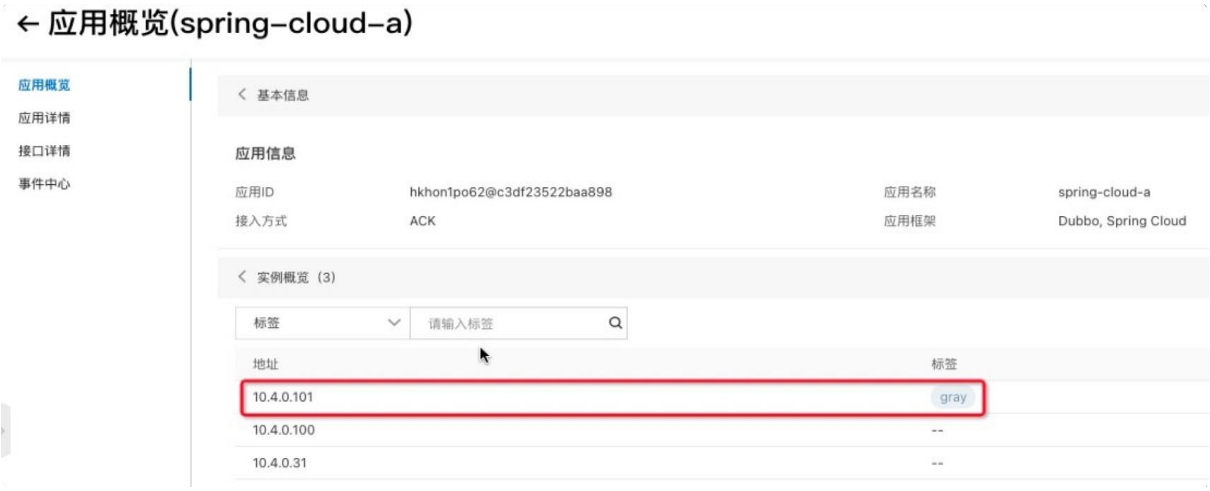


结果验证

在容器服务控制台，发现 spring-cloud-a-gray 这个 Deployment 已经自动创建，它的镜像已经替换为 spring-cloud-a:gray 版本。



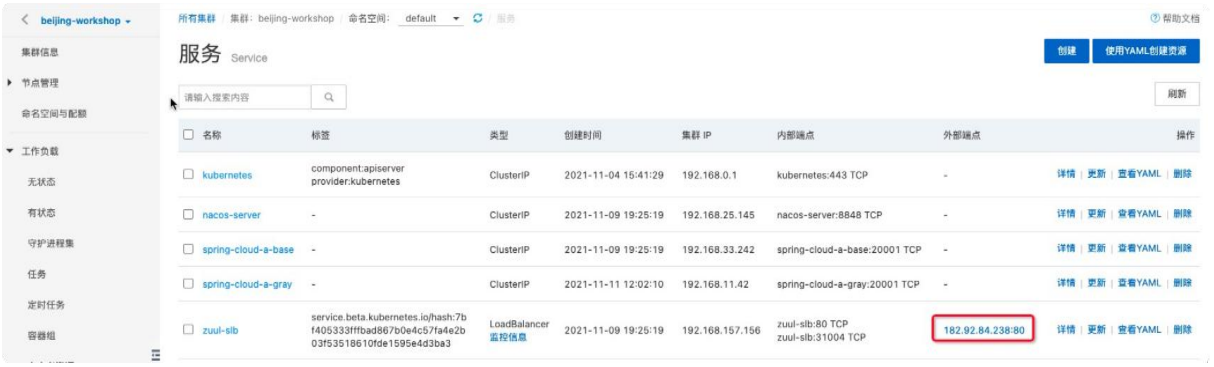
在 MSE 控制台，发现新增了带上 gray 标签的节点。



在 MSE 控制台，在应用列表中选择 spring-cloud-a ，点击应用详情，看到灰度流量已经进入到灰度的机器中。



查看真实的调用情况，首先在容器服务控制台中，找到 zuul 网关对外暴露的 SLB。



不带灰度 header 进行调用，发现路由到 A 的正常节点。

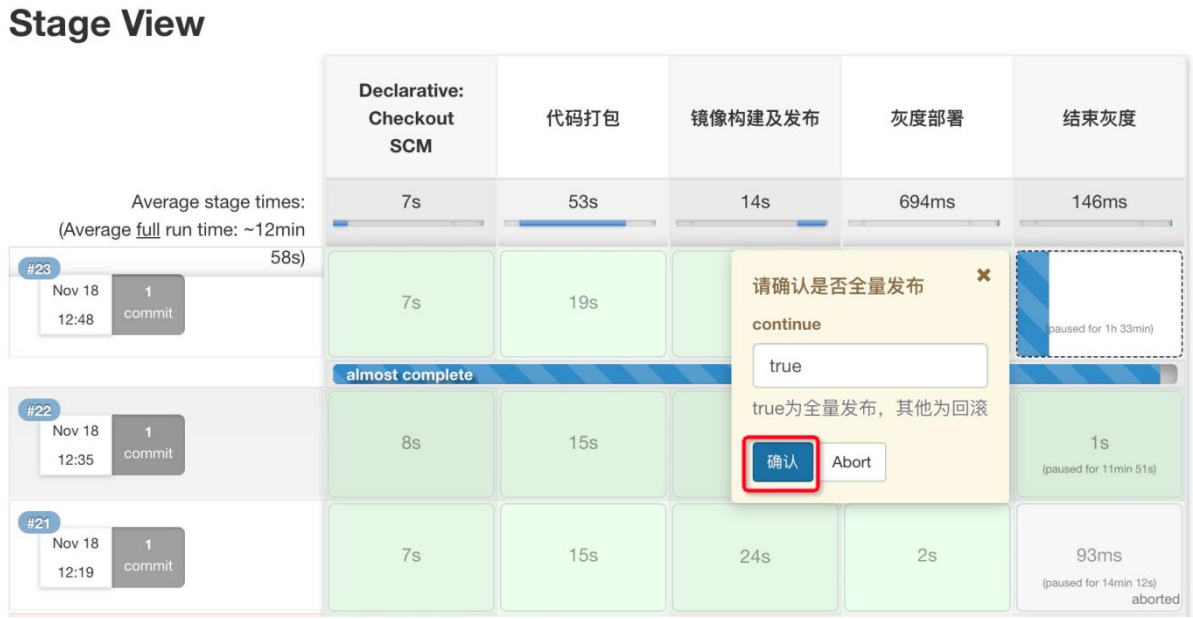
```
curl http://182.92.84.238/A/a
A[10.4.0.85] -> B[10.4.0.94] -> C[10.4.0.57]%
```

带上默认的灰度 header 进行访问，路由到 A 的灰度节点中。

```
curl -H"x-mse-tag:gray" http://182.92.84.238/A/a
Agray[10.4.0.98] -> B[10.4.0.95] -> C[10.4.0.57]%
```

全量发布

结果验证通过之后，点击确认全量发布。



在容器服务控制台，发现 spring-cloud-a-gray 这个 Deployment 已经被删除，而 spring-cloud-a 的镜像已经替换为 spring-cloud-a:gray 版本。



查看 MSE 控制台，发现灰度流量已经消失。

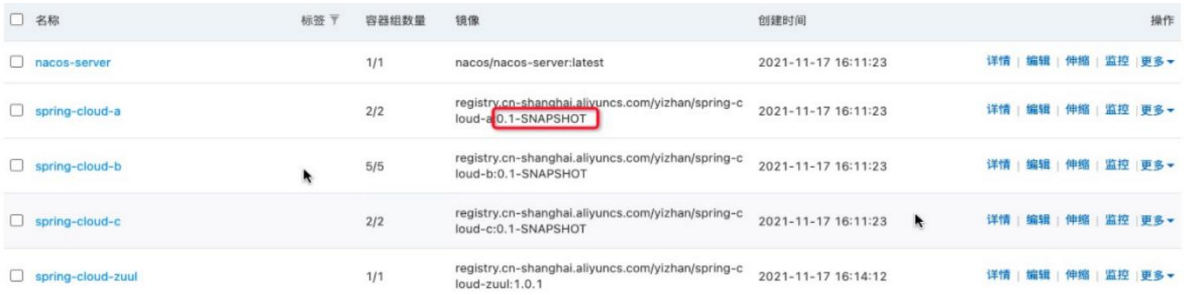


回滚

当发现验证不符合预期，则输入 `false` 执行回滚。



在容器服务控制台，发现 `spring-cloud-a-gray` 这个 Deployment 已经被删除，而 `spring-cloud-a` 的镜像仍然是老版本。



查看 MSE 控制台，发现灰度流量已经消失。



4.3 微服务应用配置最佳实践

前提条件

MSE 微服务应用配置最佳实践

- 已开通 MSE 微服务治理专业版，请参见开通 MSE 微服务治理。
- 以 ECS方式接入，参考 ECS 微服务应用接入，或
- 开源 K8s 环境方式接入，参考开源 K8s 环境中的应用接入MSE 治理中心，或
- ACK 微服务应用接入，参考 ACK 微服务应用接入MSE 治理中心

其他接入方式

- 应用配置 SDK方式接入，参考使用SDK 接入
- 应用配置 Spring Boot Starter 方式接入，参考使用 Spring Boot Starter 接入

Spring Demo 文件

@Value 配置 UserController 类对象，用于测试@PostConstruct 注解，此阶段可识别持久化值，部分内容如下：

```
@RestController
public class UserController {

    @Value("${project.name}")
    public String name;

    @Value("${user.ahas}")
    public boolean ahas;

    @Value("${user.number}")
```

```
    public int num;

    @Value("${destination}")
    public String destinationStr;



    @Autowired
    private Destination destination;

    @PostConstruct
    private void init(){
        System.out.println("[UserController] init()  value: "+ destinationStr +" , " + num + " , "+ ahas + " , " + name);
        System.out.println("[UserController] init()  configuration: "+destination.getAvg()+" , " + destination.getMax() + " , "+ destination.getMin());
    }
```

DemoConfig 类对象，用于测试 InitializingBean，afterPropertiesSet 函数初始化阶段可读取到持久化值。

```
@Configuration
public class DemoConfig implements InitializingBean {
    @Autowired
    private RequestProperties requestProperties;

    @Override
    public void afterPropertiesSet() {
        System.out.println("[DemoConfig] init()  port: " + requestProperties.getPort() + " ,interface: " + requestProperties.getInter());
    }

}
```

RequestProperties 类对象，@ConfigurationProperties 配置，value 模式。

```
@Component
@ConfigurationProperties(value = "request")
public class RequestProperties {
    private int port;

    private String inter;

    public int getPort() {
        return port;
    }

    public void setPort(int port) {
        this.port = port;
    }

    public String getInter() {
        return inter;
    }

    public void setInter(String inter) {
        this.inter = inter;
    }
}
```

Destination 类对象，@ConfigurationProperties 配置，prefix 模式。

```
@Component
@ConfigurationProperties(prefix = "property.destination")
public class Destination {
    private int max;
    private int min;
    private int avg;

    public int getMax() {
```

```
        return max;
    }

    public void setMax(int max) {
        this.max = max;
    }

    public int getMin() {
        return min;
    }

    public void setMin(int min) {
        this.min = min;
    }

    public int getAvg() {
        return avg;
    }

    public void setAvg(int avg) {
        this.avg = avg;
    }
}
```

application.properties 配置内容。

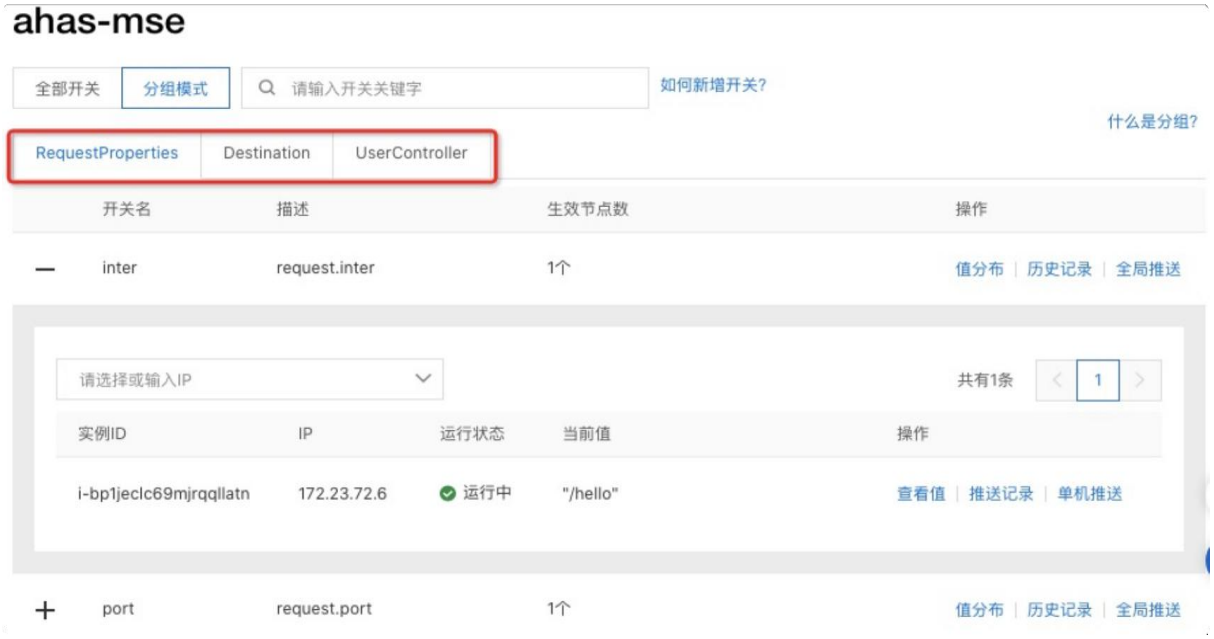
```
user.ahas=false
user.number=123
request.port=8081
request.inter=/hello
destination=sun
property.destination.max=300
property.destination.min=10
property.destination.avg=100
```

配置注册

微服务启动后可在‘微服务治理中心>应用列表中’菜单看到接入的 ahas-mse 应用。

点击应用进入后可看到应用配置菜单，对应的配置项可以分组方式展示，分组名为识别出的类名。

开关名为类中字段名，描述内容在 @Value 方式为注解中内容，@ConfigurationProperties方式为 Spring 配置文件中配置项。

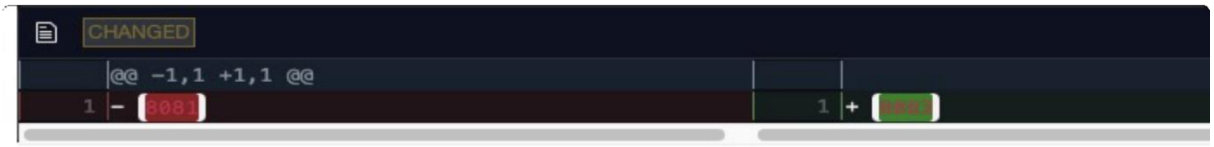


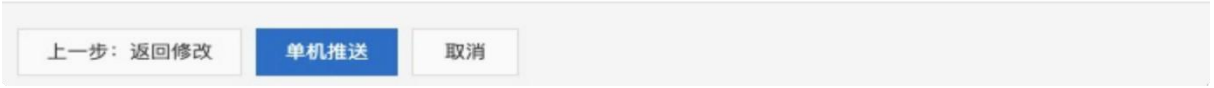
配置修改

分为单机推送与全局推送两种方式，单机推送的值不会持久化，仅当前微服务实例生命周期有效，全局推送方式会进行值持久化，微服务实例再次启动后可看到持久化值。

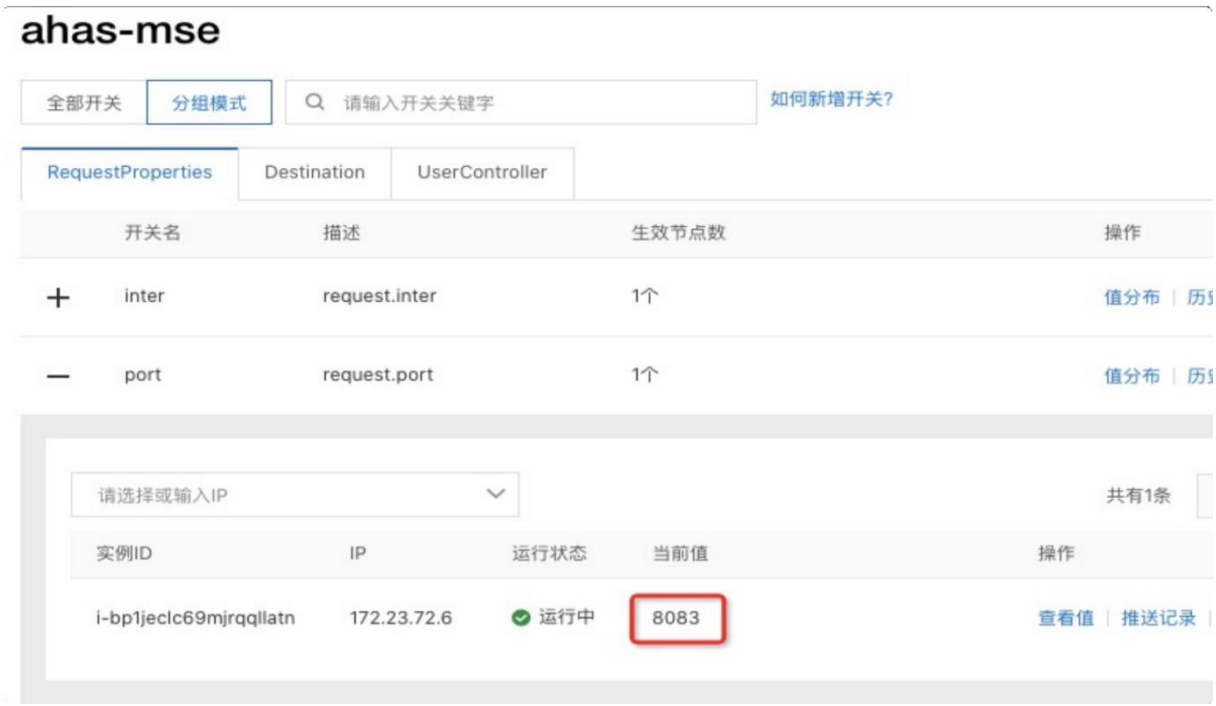
单机推送

对 RequestProperties，port 字段执行单机推送。

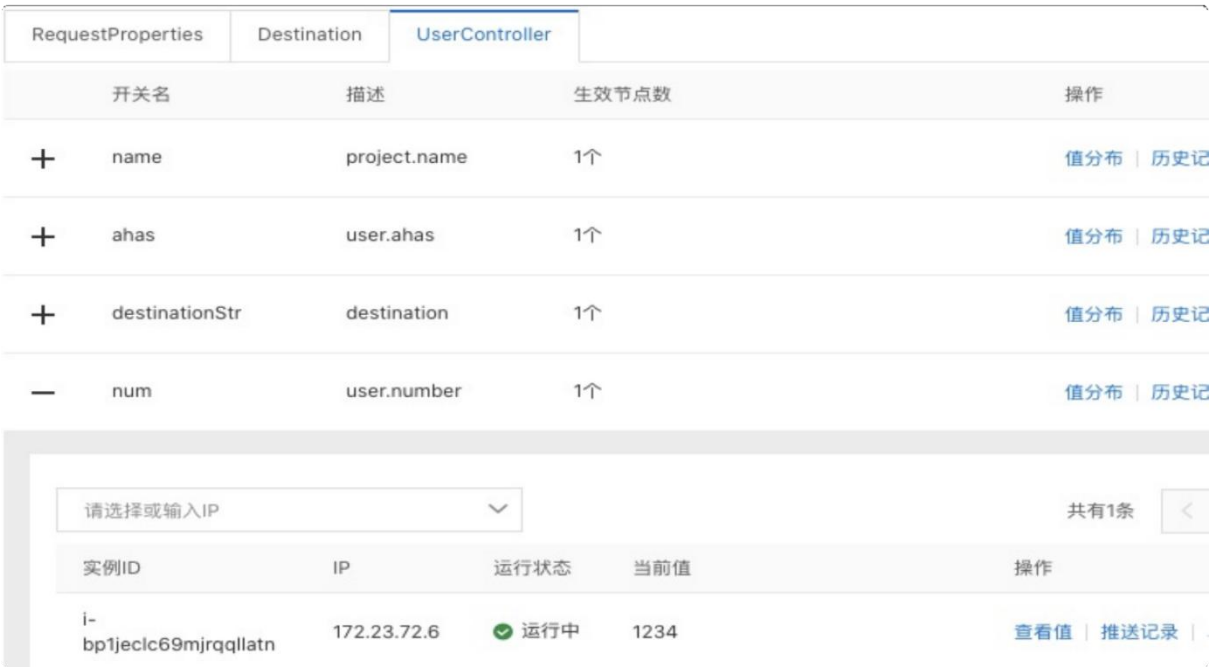




此时应用中 port 字段即被修改为'8083'，可在控制台查看。



类似的可对 UserController 中 num 字段进行单机推送。



可在推送记录中查看历史操作

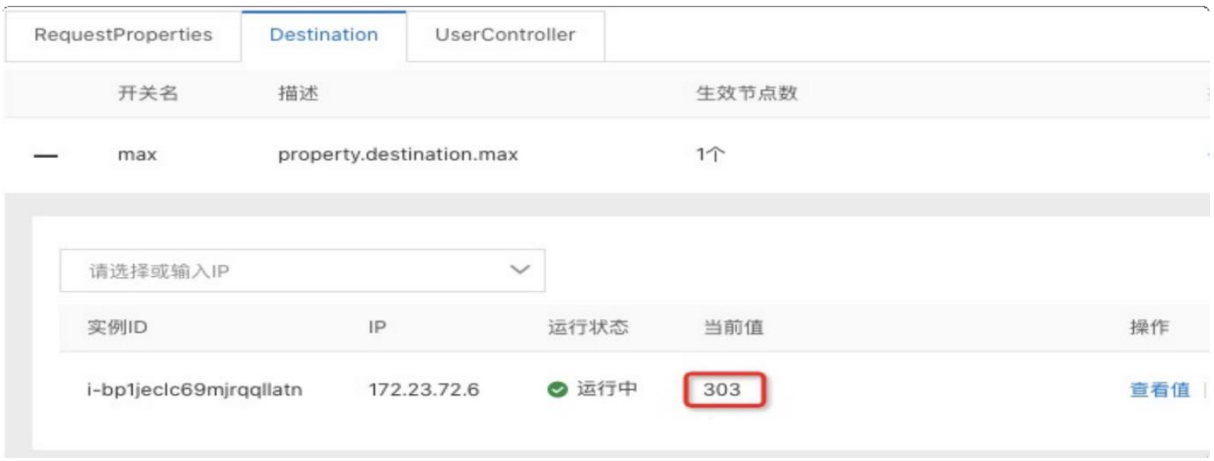


全局推送

对 Destination，max 字段执行全局推送。



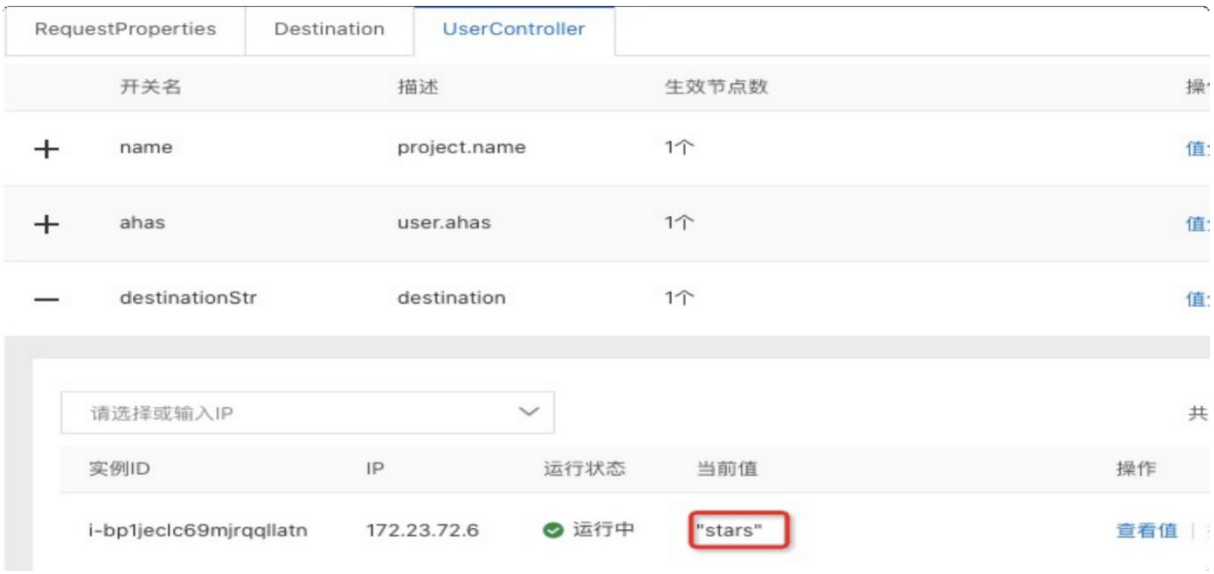
推送后可见修改后的值，同时数据已经持久化。



类似的可对 UserController 中 destinationStr 字段进行全局推送。



推送后内存值变更可在控制台查看，同时开关值已持久化。



配置持久化

初始化阶段

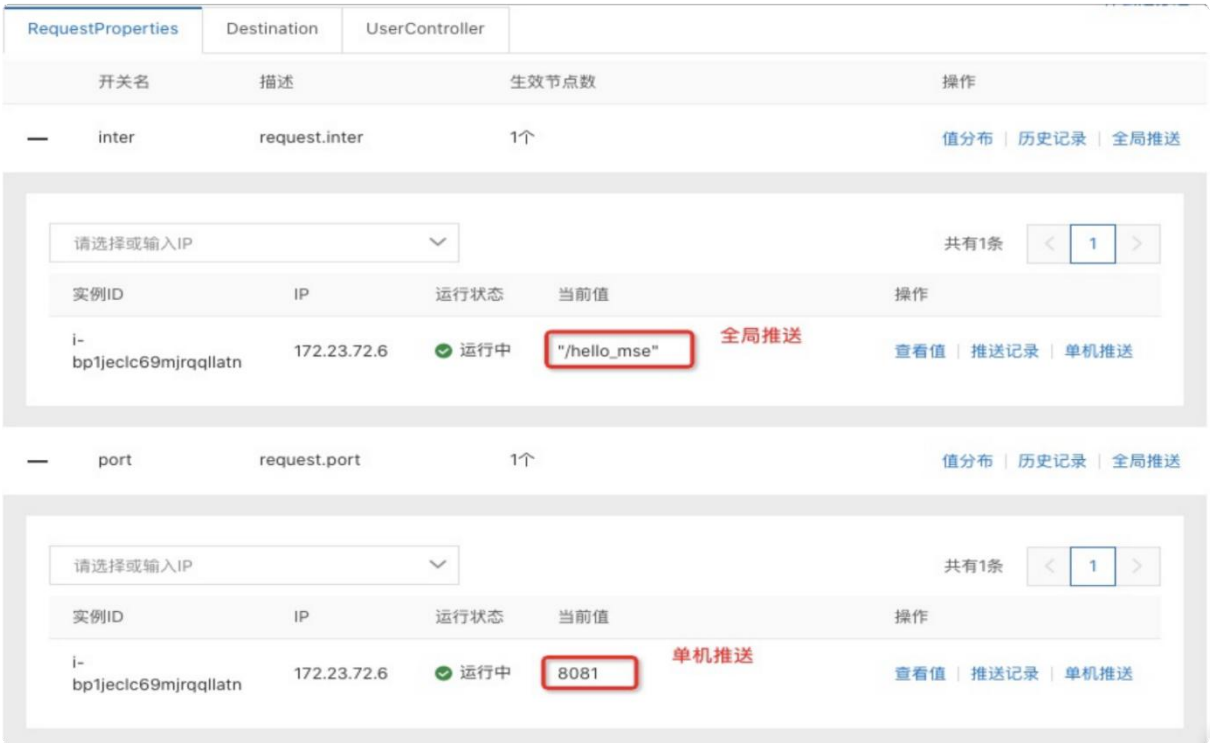
重启应用，在 InitializingBean，afterPropertiesSet 函数初始化阶段与@PostConstruct 初始化阶段均可被读取到已持久化的值。

可通过测试 demo 中的启动日志查看，内容如下：

```
2022-02-23 15:51:16.754 [INFO ] Root WebApplicationContext: initialization completed in 2482 ms
[DemoConfig] init()  port: 8081  interface: /hello_mse
[UserController] init()  value: stars , 123 , false , ahas-mse
[UserController] init()  configuration: 100 , 303 , 10
2022-02-23 15:51:17.489 [INFO ] Initializing ExecutorService 'applicationTaskExecutor'
```

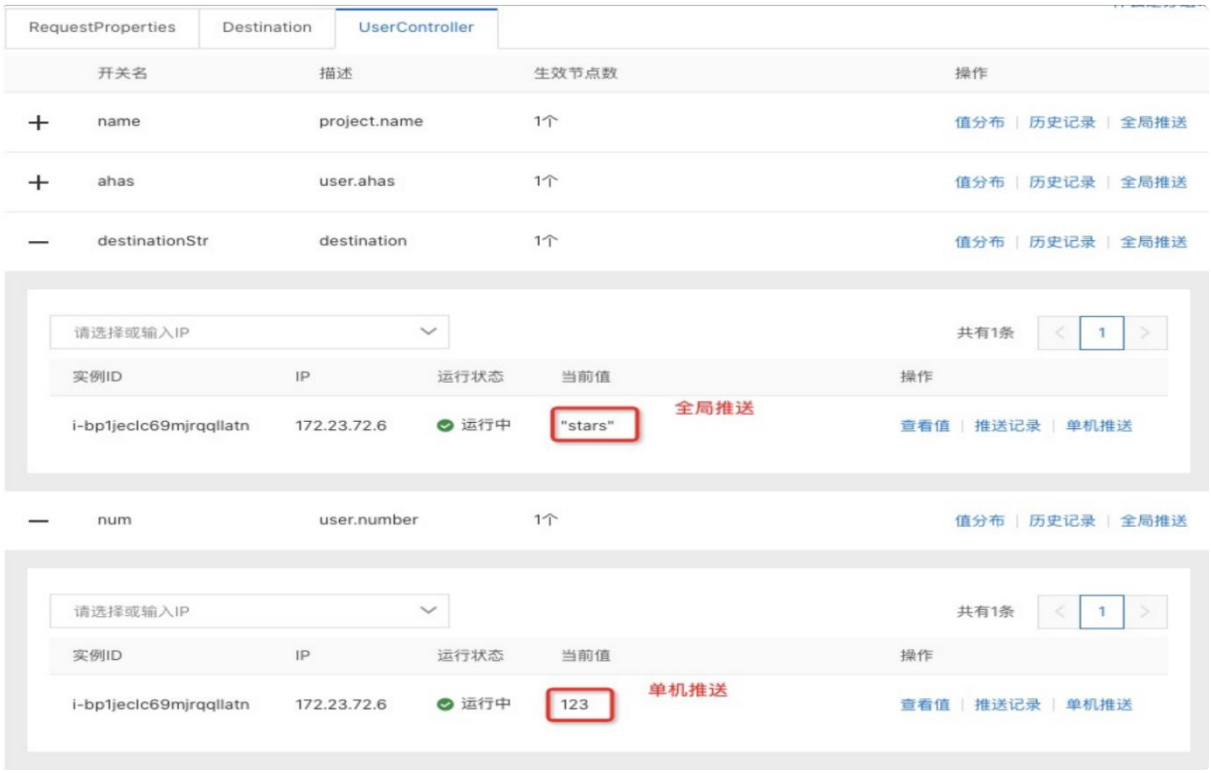
@ConfigurationProperties 配置

在控制台观察 RequestProperties 可看到全局推送方式推送的‘inter’配置项为持久化值，而单机推送的‘port’配置项仍为 Spring 原始配置内容。



@Value 配置

在控制台观察 UserController 可看到全局推送方式推送的‘destinationStr’配置项为持久化值，而单机推送的‘num’配置项仍为 Spring 原始配置内容。



历史记录

可在‘历史记录’菜单查看推送记录。

开关名	推送类型	namespace	推送值	操作时间	生效IP数	操作
inter	全局推送	com.karl.pre.config.RequestProperties	"/hello_mse"	2022-02-23 15:50:47	1	查看
destinationStr	全局推送	com.karl.pre.controller.UserController	"stars"	2022-02-23 15:44:34	1	查看
num	单机推送	com.karl.pre.controller.UserController	1234	2022-02-23 15:41:36	1	查看
max	全局推送	com.karl.pre.config.Destination	303	2022-02-23 15:37:15	1	查看
port	单机推送	com.karl.pre.config.RequestProperties	8083	2022-02-23 15:31:22	1	查看

4.4 微服务限流降级最佳实践

以 WEB 流量防护场景为例说明限流降级最佳实践。

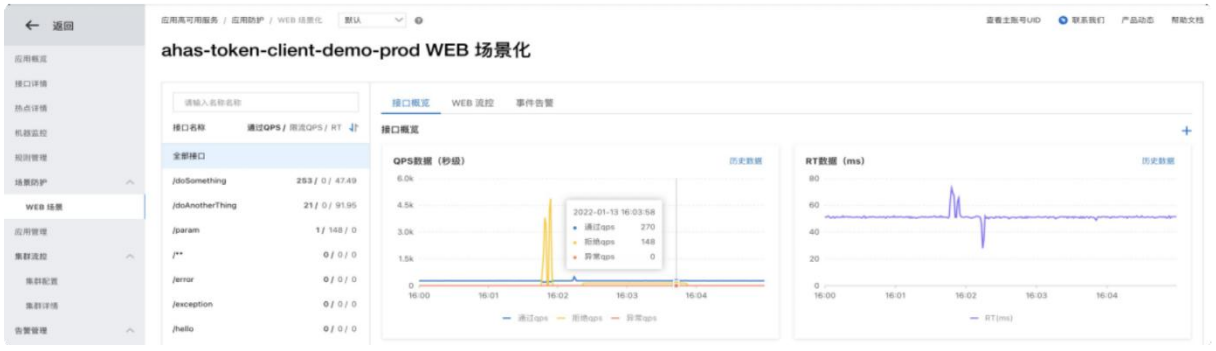
快速玩转 AHAS Web 场景防护

应用接入

第一步，我们将 Web 服务接入 AHAS 流量防护。可参考：https://help.aliyun.com/document_detail/131232.html，AHAS 提供多种快速便捷的无侵入接入手段：



以普通 Spring Boot Web 服务为例，接入 AHAS 成功后，只要触发服务调用/接口访问，即可在 AHAS 控制台看到自己的服务，并可以在 Web 场景页面看到自己的 Web 接口。默认 Spring Boot 应用接入会提取 controller 中的 URL path 作为资源名称：



我们可以在监控页面非常直观地观测各个接口的实时流量与响应时间情况，以便于我们评估系统的稳定性。

配置防护规则

第二步，我们给其中一个接口配置 Web 流控规则。在我们的示例中，我们针对 /hello 这个 API，配置热点参数流控。流控的请求属性为 name 这个请求参数（URL params），流控策略为每个热点参数分别限制每秒访问量最多 1 次。AHAS 会自动提取每个 /hello 请求中的 name 参数对应的值，自动分析出其中 top K 频次的热点值，然后分别对每个热点值进行控制，不超过规则中的访问次数。

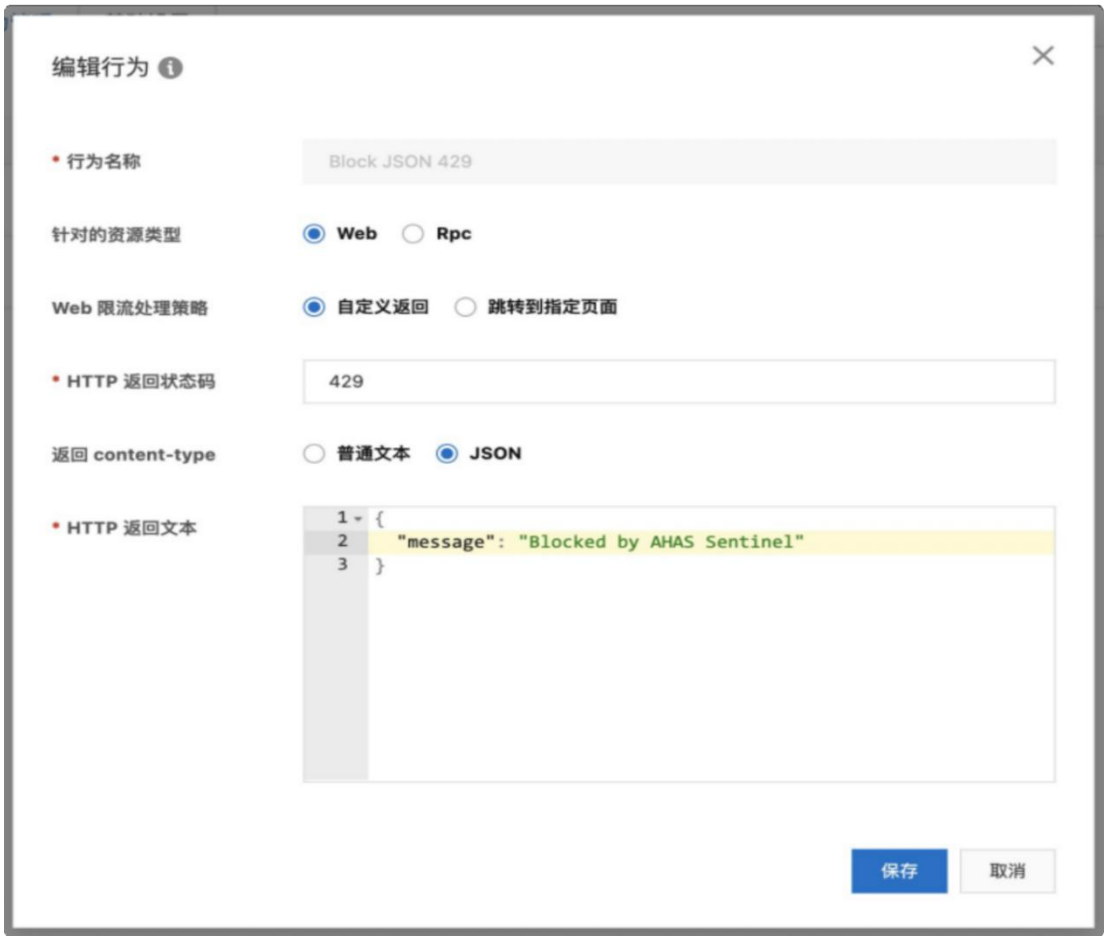


效果示例：假设请求中有许多针对 /hello 接口的访问，其中 /hello?name=A 和 /hello?name=B 两个 name 参数的访问频次非常高，被 AHAS 统计统计为热点值。那么上面的流控规则会针对 name=A 和 name=B 这两个参数值的请求，分别限制每秒访问量不超过 1 次。

绑定行为

第三步，我们给上面的 Web 流控规则绑定一个 fallback 行为，即指定触发该规则后，Web 接口的返回内容。我们可以在页面中自定义 Web 返回行为。以下是一个返回 429 状态码的 JSON 返回配置示例：


新增 Web 防护规则
Fallback行为定义某个埋点资源触发了某种规则（如流控、熔断、降级）后的处理行为，例如某个Web接口触发流控规则后返回Blocked by Sentinel的提示文本。
请求分组：/hello
防护类型：WEB 防护
接口类型
Web
选择防护场景
配置防护规则
3
配置限流行为
* 关联行为
Block JSON 429
行为管理
如无需自定义限流后处理行为，选择默认行为即可。
取消关联
新增行为
行为描述：ContentType = JSON字符串，HTTP status code = ContentType = JSON string, HTTP status code = 429，返回内容 = { "message": "Blocked by AHAS Sentinel" }
上一步
下一步
取消


编辑行为
* 行为名称
Block JSON 429
针对的资源类型
Web
Rpc
Web 限流处理策略
自定义返回
跳转到指定页面
* HTTP 返回状态码
429
返回 content-type
普通文本
JSON
* HTTP 返回文本
1 {
2 "message": "Blocked by AHAS Sentinel"
3 }
保存
取消

配置完成后，我们可以在流程页面选择我们创建的 fallback 行为，然后保存，这样我们的规则就会实时生效了。

流控效果

第四步，我们发起针对 /hello 接口的流量，并给 name 参数带上不同的值，可以从控制台的监控上面看到请求触发了流控：



同时被流控的热点请求返回值也与我们在控制台定义的 fallback 返回相对应：

```
[root@ahas-token-client-demo-prod-8469477696-mpvv5 demo]# ./flow.sh "localhost:8088/hello?name=ahas"
Hello, ahas  一个请求通过
{
  "message": "Blocked by AHAS Sentinel"
}
{
  "message": "Blocked by AHAS Sentinel"
}
```

多余的请求返回给定的内容和状态码

热点防护

在生产环境，我们这些参数的值可能非常多，比如商品 ID 可能有上千万个。在热点请求被流控后，我们通常希望能够知道有哪些 top 参数被限制，方便我们了解业务的请求情况。AHAS 近期新上线热点监控功能，提供热点参数可观测的能力，结合 top 参数热力图，可以非常直观地在控制台了解实时业务情况：



以上就是一个简单的 Spring Boot Web 场景流控的示例，欢迎大家在 AHAS 控制台体验，有关 Web 场景防护的更多信息可以参考文档：

https://help.aliyun.com/document_detail/337922.html

4.5 微服务开发测试提效最佳实践

本文是阿里云微服务引擎 MSE 在微服务开发测试提效最佳实践介绍。

前提条件：微服务应用已接入MSE

下面我们来体验一下，MSE 上如何使用微服务测试的能力。

操作步骤

步骤一：接入 MSE 微服务治理

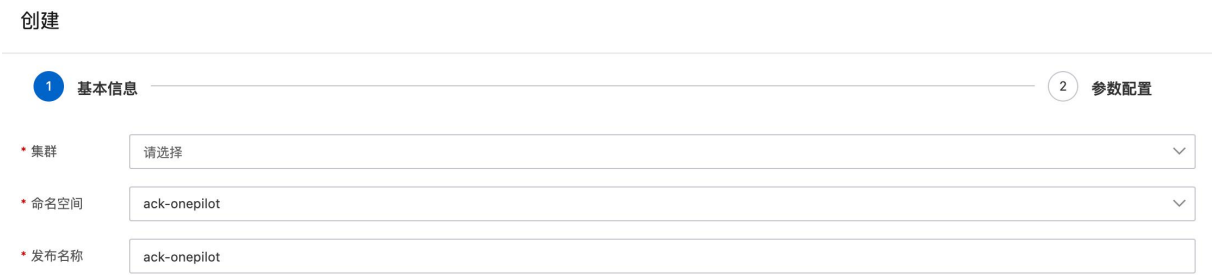
1.安装 ack-onepilot

a.登录容器服务控制台。

b.在左侧导航栏单击**市场 > 应用目录**。

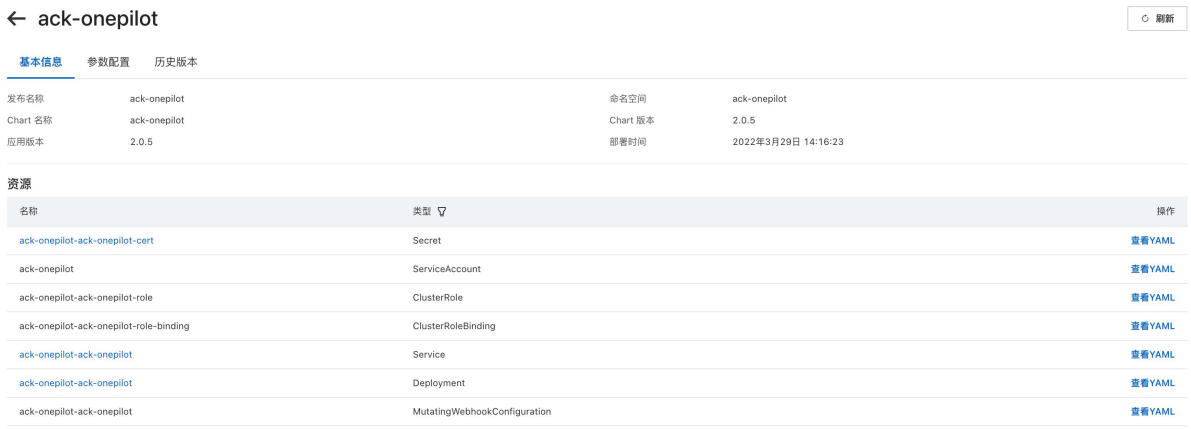
c.在应用目录页面点击**阿里云应用**，选择**微服务**，并单击 **ack-onepilot**。

d.在 ack-onepilot页面右侧集群列表中选择集群，然后单击创建。



安装 MSE 微服务治理组件大约需要 2 分钟，请耐心等待。

创建成功后，会自动跳转到目标集群的 **Helm**页面，检查安装结果。如果出现以下页面，展示相关资源，则说明安装成功。



2.为 ACK 命名空间中的应用开启 MSE 微服务治理

a.登录 MSE 治理中心控制台，如果您尚未开通 MSE 微服务治理，请根据提示开通。

b.在左侧导航栏选择**微服务治理中心 > K8s 集群列表**。

c.在 K8s 集群列表页面搜索框列表中选择集群名称或集群 ID，然后输入相应的关键字，单击搜索图标。

d.单击目标集群操作列的**管理**。

e.在集群详情页面命名空间列表区域，单击目标命名空间操作列下的**开启微服务治理**。

f.在开启微服务治理对话框中单击确认。



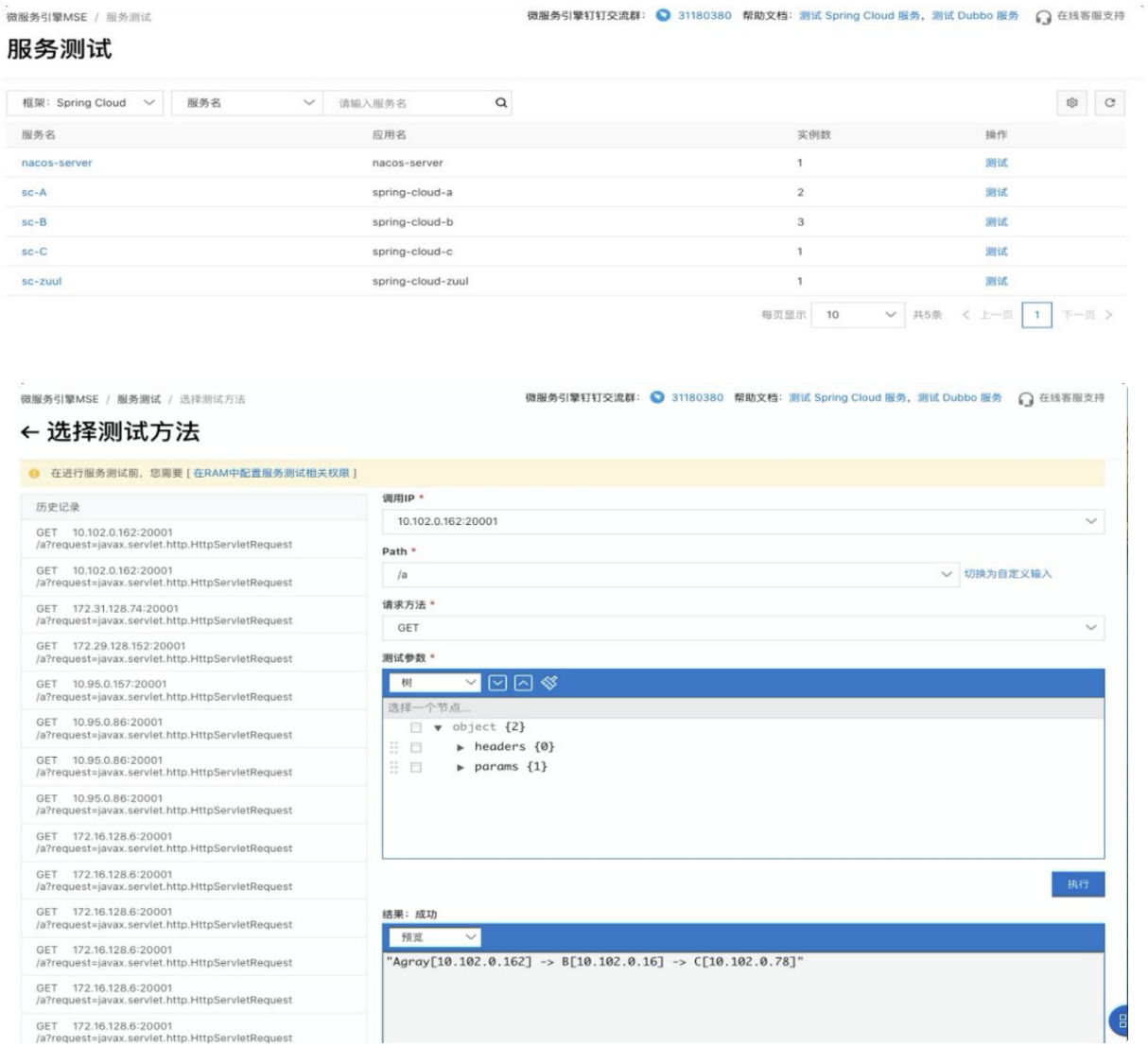
步骤二：服务测试

1.登录 MSE 治理中心控制台，在页面左上角选择地域；

2.左侧导航栏选择：微服务治理 -> 微服务测试 -> 服务测试；

3.选择服务名为“sc-A”的服务，点击测试，进入测试详情页；

4.选择 Path 为“/a”的请求，填写测试参数，点击执行 -> 查看返回结果，同时左侧可以展示历史记录；



步骤三：服务压测

1.登录 MSE 治理中心控制台，在页面左上角选择地域；

2.左侧导航栏选择：微服务治理 -> 微服务测试 -> 服务压测 -> 创建场景；

3.添加步骤

a.选择应用 -> 选择框架 -> 选择服务 -> 选择方法；

b.填写参数；

c.断言/出参提取；

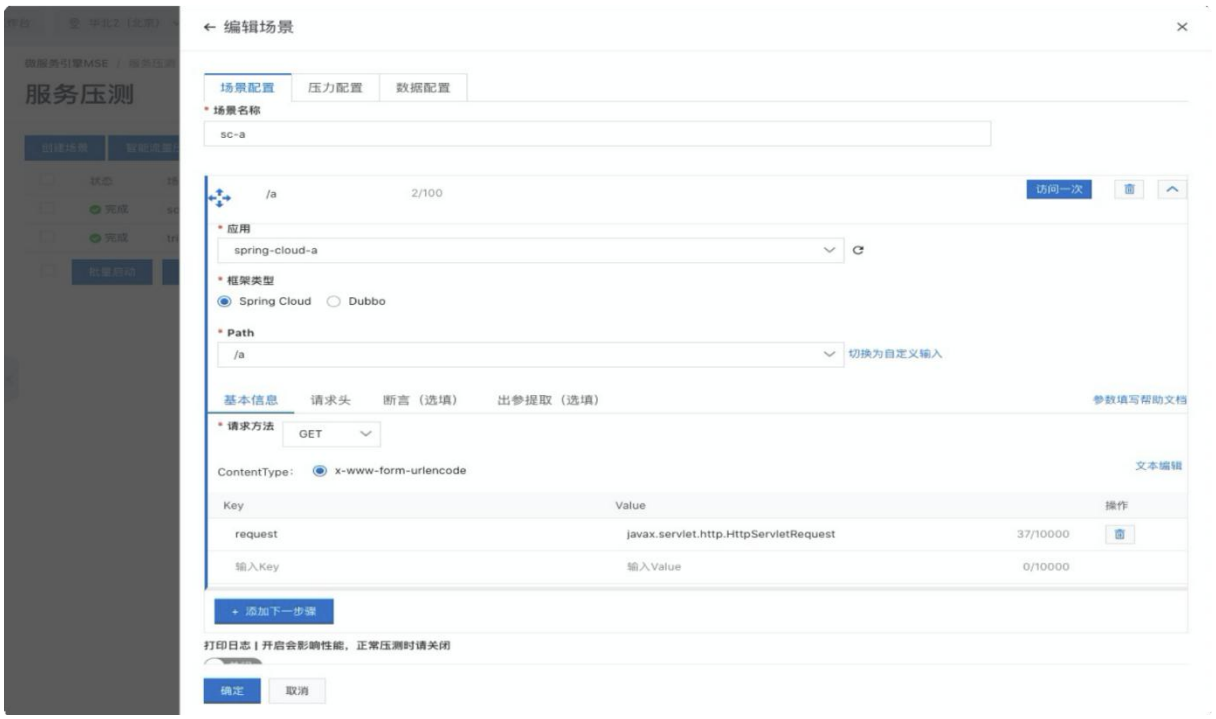
4.填写压测参数，点击确认；

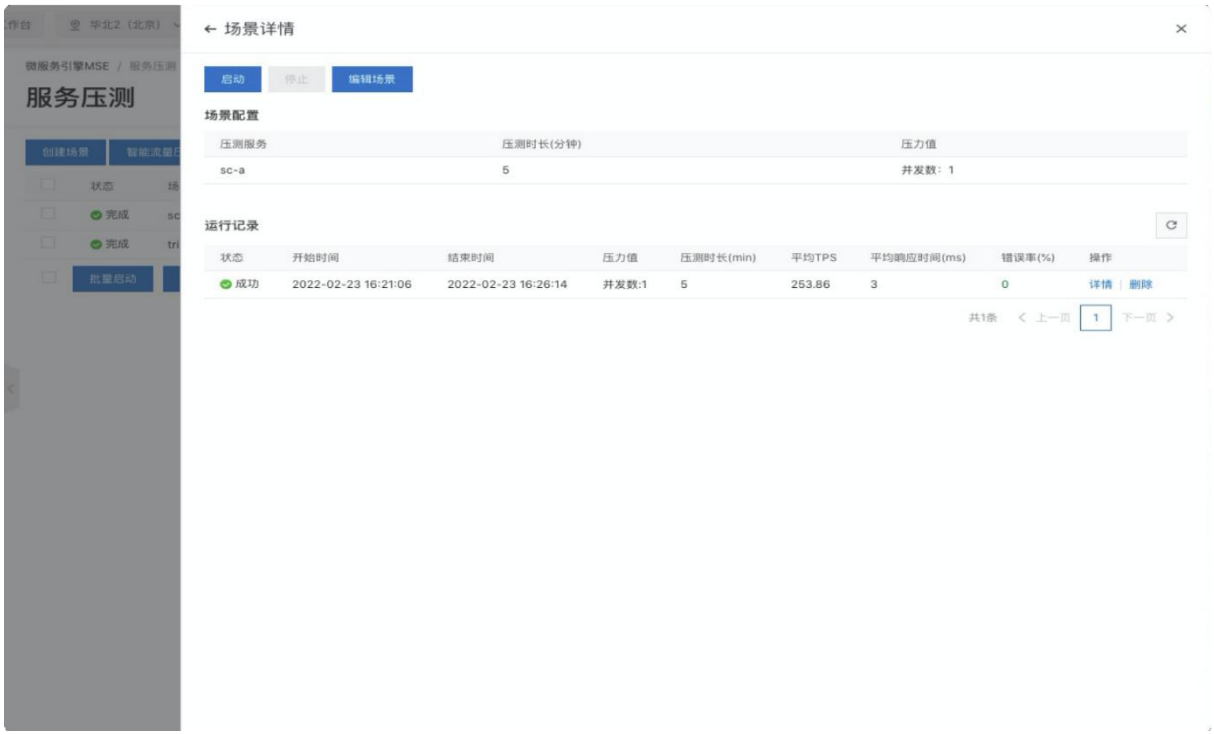
5.进入压测场景列表页，点击详情；

6.进入压测详情页，点击启动，等待施压机准备就绪；

7.点击详情，进入压测性能数据报告页，实时查看性能数据；









步骤四：自动化回归（用例管理）

1.登录 MSE 治理中心控制台，在页面左上角选择地域；

2.左侧导航栏选择：微服务治理 -> 微服务测试 -> 自动化回归（用例管理） -> 创建用例；

3.添加步骤

a.选择应用 -> 选择框架 -> 选择服务 -> 选择方法；

b.填写参数；

c.断言/出参提取；

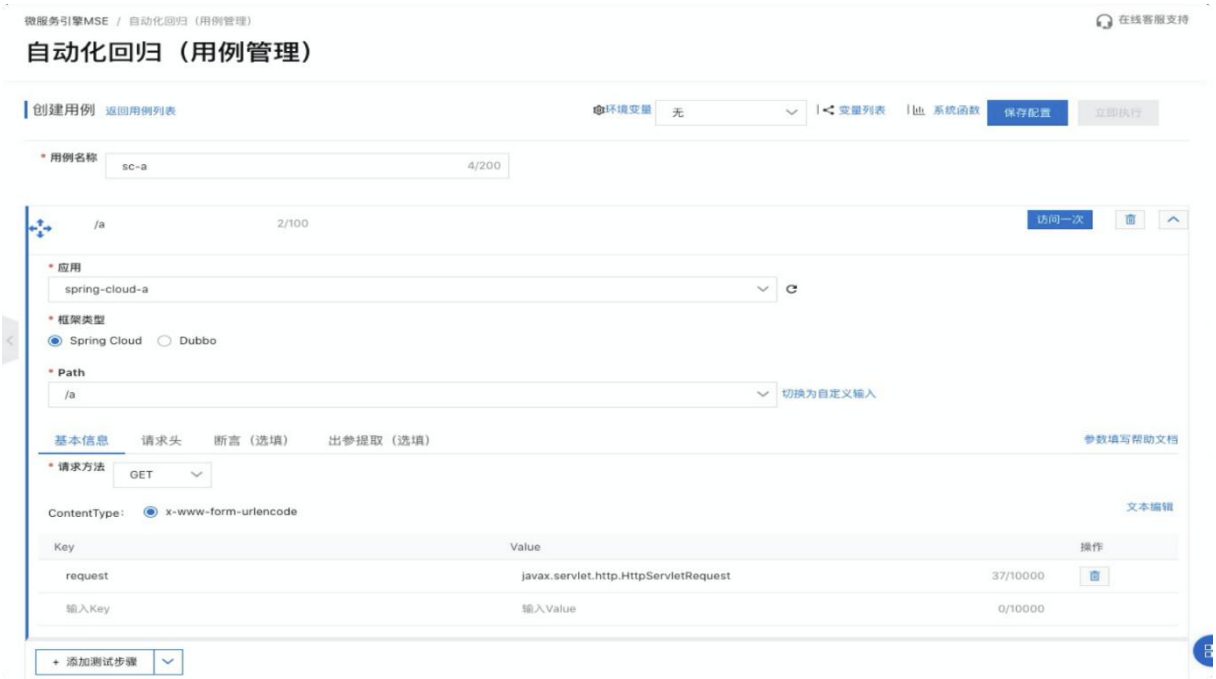
4.可以添加多个步骤；

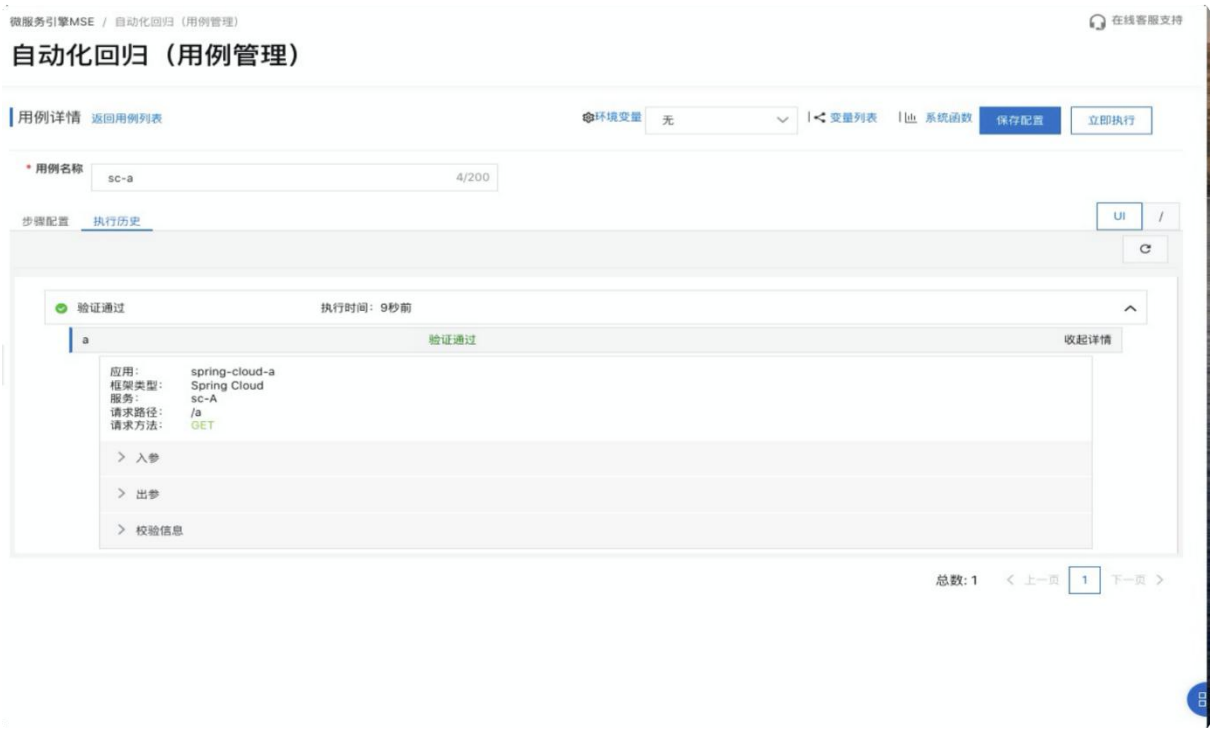
5.保存用例；

6.点击执行；

7.通过执行历史，查看用例是否通过；







步骤五：自动化回归（用例集）

1.登录 MSE 治理中心控制台，在页面左上角选择地域；

2.左侧导航栏选择：微服务治理 -> 微服务测试 -> 自动化回归（用例集） -> 创建用例集；

3.选择已创建的用例集 sc-a，点击详情，进入用例集详情页；

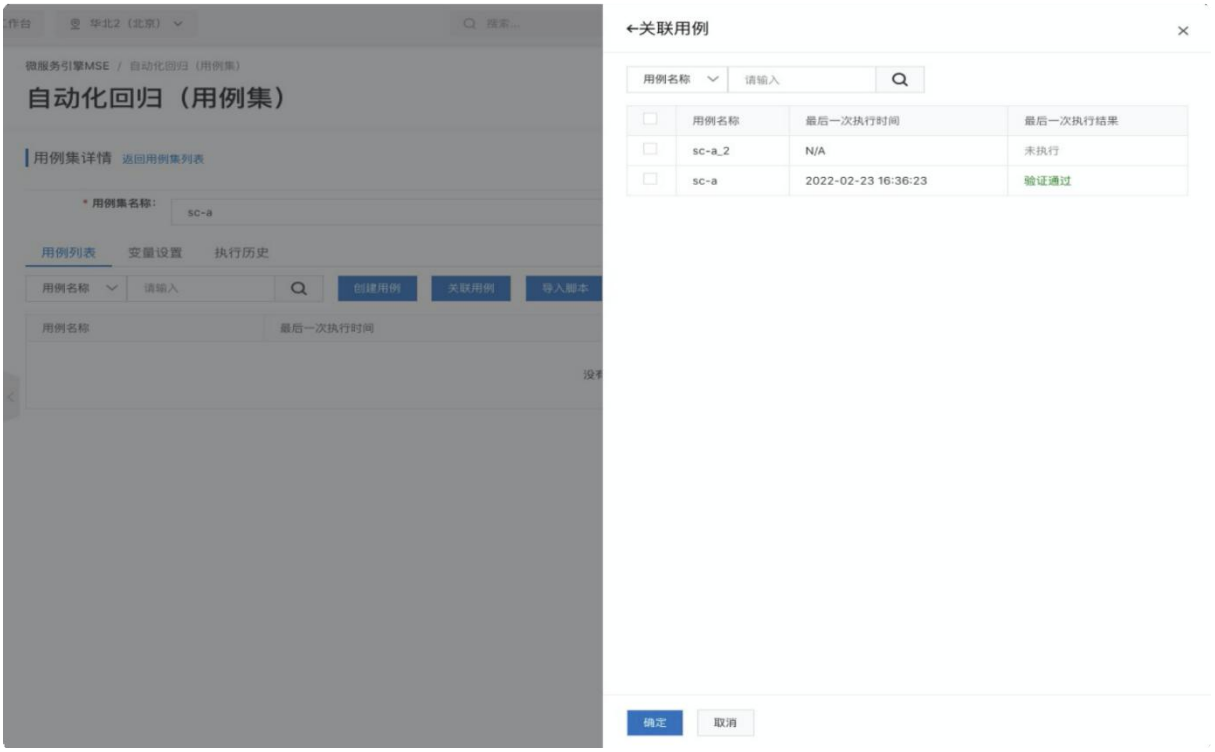
4.点击关联用例，选择已创建的用例，进行关联；

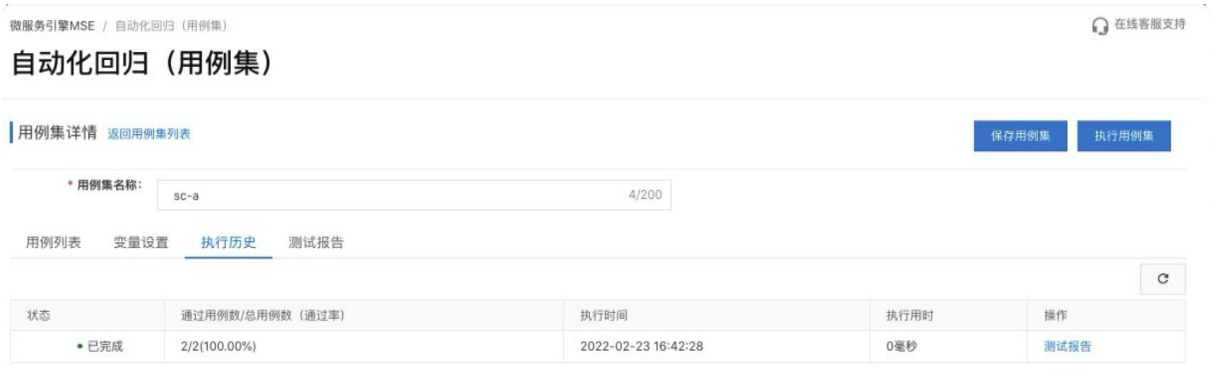
5.点击执行用例集，进入执行历史，查看用例集执行记录；

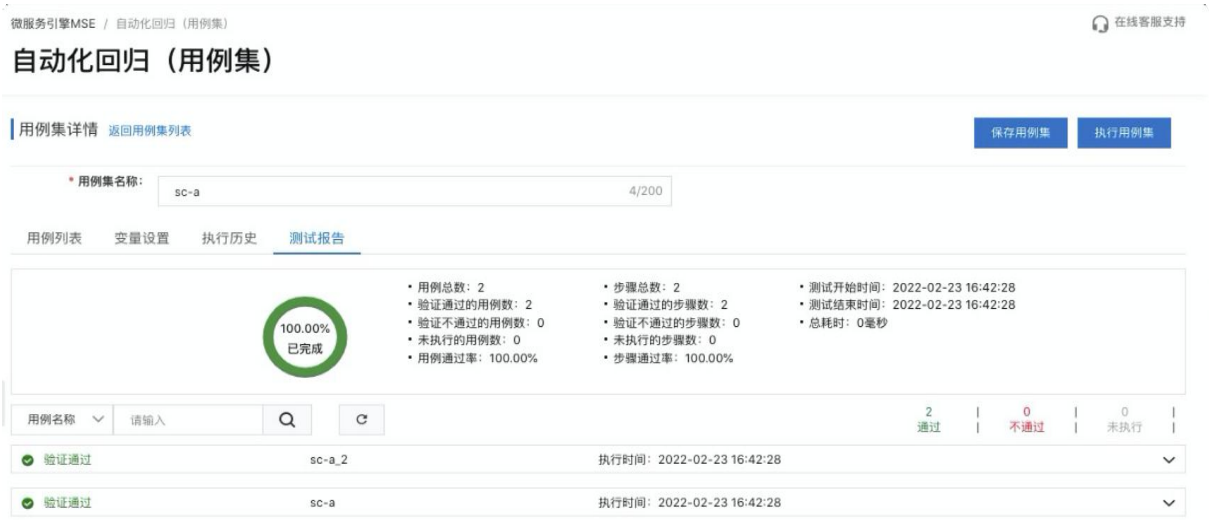
6.点测试报告，查看测试报告详情；




微服务引擎MSE / 自动化回归（用例集）
在线客服支持
自动化回归（用例集）
用例集详情 返回用例集列表
保存用例集
执行用例集
* 用例集名称:
sc-a
4/200
用例列表
变量设置
执行历史
用例名称
请输入
创建用例
关联用例
导入脚本
导出脚本
用例名称
最后一次执行时间
最后一次执行结果
操作
没有数据


←关联用例
用例名称
请输入
用例名称
最后一次执行时间
最后一次执行结果
sc-a_2
N/A
未执行
sc-a
2022-02-23 16:36:23
验证通过
确定
取消


微服务引擎MSE / 自动化回归（用例集）
在线客服支持
自动化回归（用例集）
用例集详情 返回用例集列表
保存用例集
执行用例集
* 用例集名称:
sc-a
4/200
用例列表
变量设置
执行历史
测试报告
状态
通过用例数/总用例数（通过率）
执行时间
执行用时
操作
已完成
2/2(100.00%)
2022-02-23 16:42:28
0毫秒
测试报告



步骤六：服务巡检

1.登录 MSE 治理中心控制台，在页面左上角选择地域；

2.左侧导航栏选择：微服务治理 -> 微服务测试 -> 服务巡检 -> 创建巡检任务；

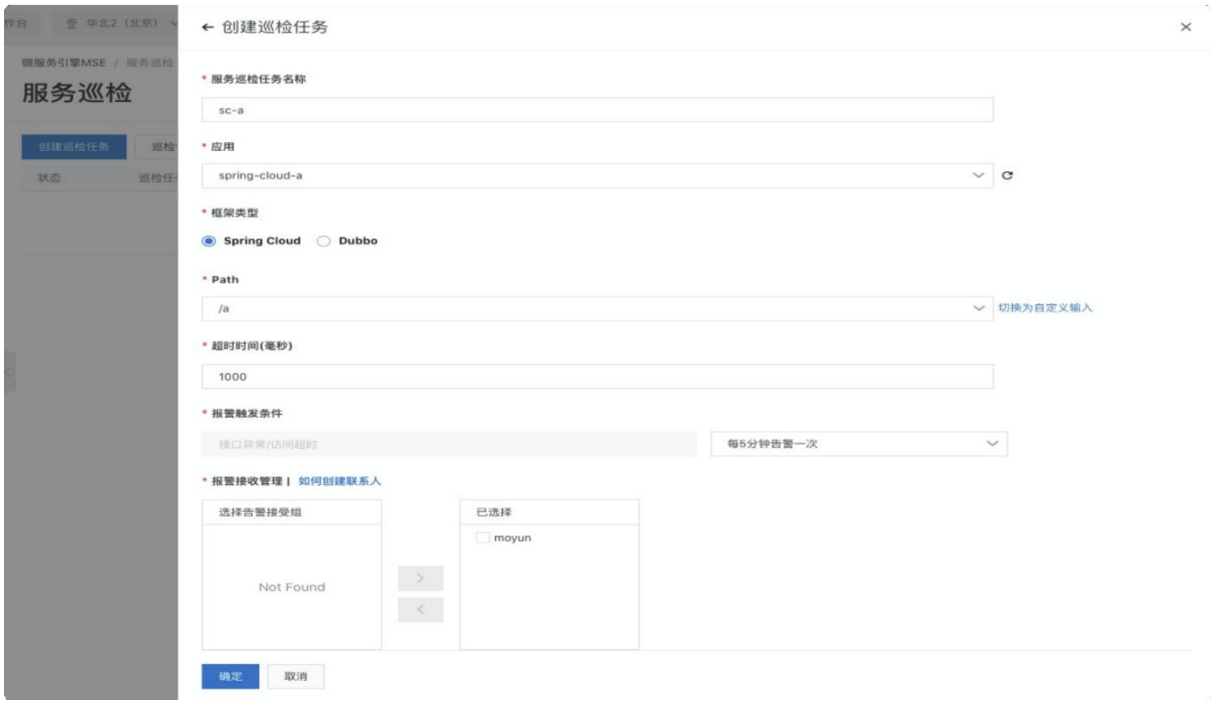
3.选择需要巡检的应用 -> 选择框架 -> 选择服务 -> 选择方法；

4.填写巡检参数及断言内容，点击确认；

5.进入巡检任务列表页，点击启动，即开始巡检；

6.巡检失败时，可以通过失败记录进行查看，也可以添加告警，通过钉钉、短信、邮件的方式告警；







步骤七：智能流量测试

1.登录 MSE 治理中心控制台。

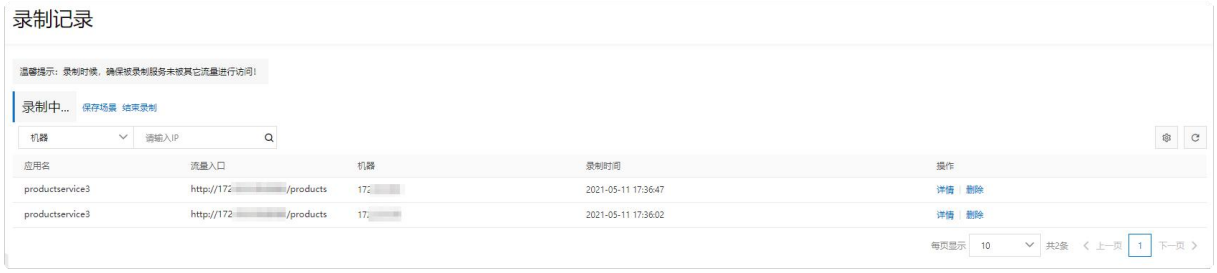
2.在左侧导航栏选择**微服务治理中心 > 微服务测试 > 智能流量测试**。

3.在顶部菜单栏选择**地域**，然后在**应用名**文本框中输入应用名称，单击 图标。

4.在应用列表中单击目标应用**操作**列的**自动化回归录制**。

5.在**录制流量**对话框中选择路径，然后单击**确认**。

6.在**录制记录**页面，您可查看当前流量信息，包括应用名、流量入口、机器、录制时间等。



当前流量正在**录制中**，您可在**录制记录**页面执行以下操作：

- 单击目标录制流量**操作**列下的**详情**，可在**流量详情**面板中查看请求信息、响应信息等。
- 单击目标录制流量**操作**列下的**删除**，可删除该流量数据。

7.**可选**：在**录制记录**页面单击左上角**保存场景**，在**保存操作场景**对话框中输入场景名，单击**确定**。当前流量录制结果会自动保存至管理页面。

8.**可选**：在**录制记录**页面单击左上角**结束录制**。自动返回**智能流量测试**页面。

步骤八：服务 Mock

1.登录 MSE 治理中心控制台。

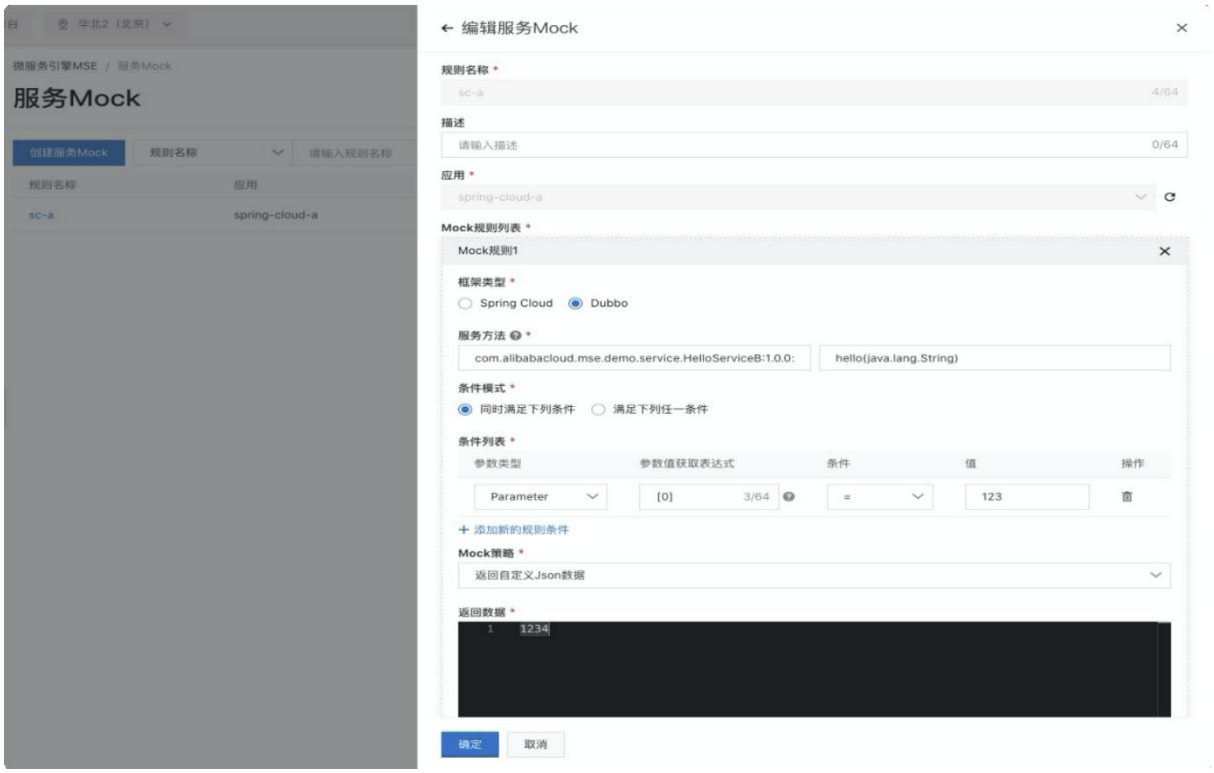
2.在左侧导航栏选择**微服务治理中心 > 服务 Mock**。

3.在顶部菜单栏选择**地域**，在**服务 Mock**页面单击**创建服务 Mock**。

4.在**创建服务 Mock**面板中填入相关参数，然后单击**确定**。

5.直接访问依赖该 Mock 规则的 API，判断 Mock 是否生效。



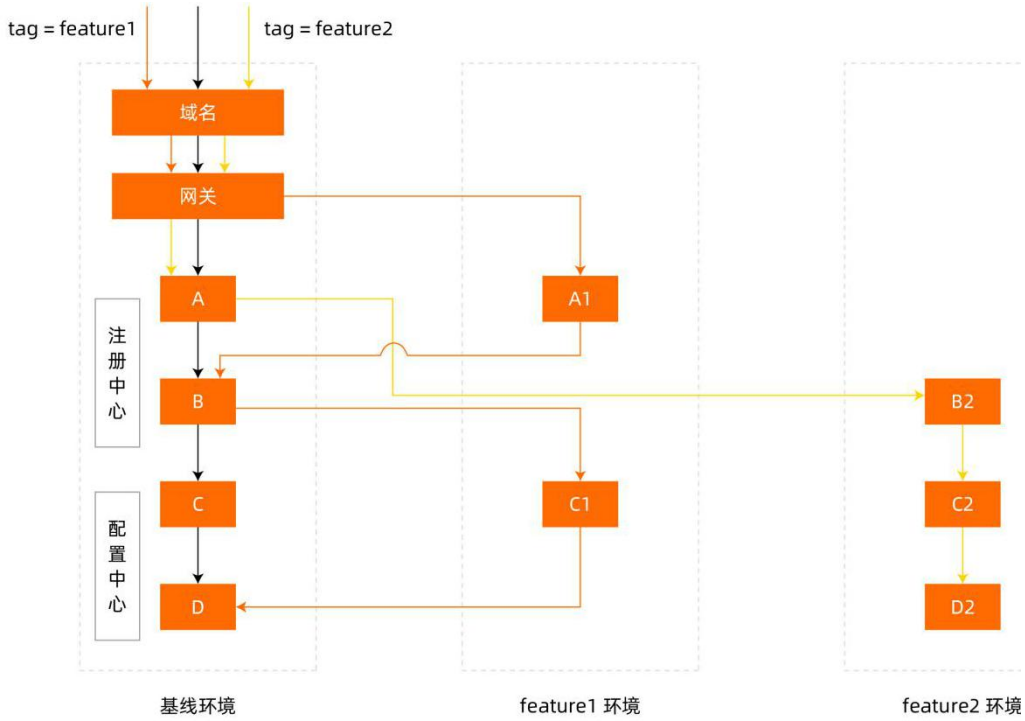

华北2（北京）
微服务引擎MSE / 服务Mock
服务Mock
创建服务Mock
规则名称
请输入规则名称
规则名称
应用
sc-a
spring-cloud-a
← 编辑服务Mock
规则名称 *
sc-a
4/64
描述
请输入描述
0/64
应用 *
spring-cloud-a
Mock规则列表 *
Mock规则1
框架类型 *
Spring Cloud
Dubbo
服务方法 *
com.alibabacloud.mse.demo.service.HelloServiceB:1.0.0:
hello(java.lang.String)
条件模式 *
同时满足下列条件
满足下列任一条件
条件列表 *
参数类型
参数值获取表达式
条件
值
操作
Parameter
[0]
3/64
=
123
添加新的规则条件
Mock策略 *
返回自定义Json数据
返回数据 *
1
1234
确定
取消

4.6 微服务敏捷开发最佳实践

如何低成本玩转敏捷开发

什么是 MSE 开发环境隔离，简单地说就是将 feature 环境的隔离方式从简单的物理隔离转为逻辑隔离。借助于 MSE 提供的逻辑隔离，您只需要维护一套完整的基线环境，在增加 feature 环境时，只需要单独部署这个 feature 所涉及到改动的应用即可，而不需要在每个 feature 环境都部署整套的微服务应用及其配套设施。

我们称之这唯一的一套完整的环境为基线环境。基线环境包含了所有微服务应用，也包含了服务注册中心、域名、SLB、网关等其他设施，而 feature 环境中只包含了这个 feature 中需要修改的应用。这样维护 n 套 feature 环境的成本，就变成了加法，而不是原来的乘法，由 n × m 变成了 n + m。**差不多相当于零成本增加 feature 环境，这样我们就可以放心地扩容出多套 feature 环境，每个开发小哥哥都可以轻松拥有属于自己的独立环境，尽情地享受微服务敏捷开发。**



从上图中我们可以看到，feature1 对应的流量，在发现 feature1 中存在 A1 应用时，一定会

去往 A1 节点，A1 在调用B 的时候发现 feature1 环境中不存在 B1 ，则会将请求发到 基线版本的 B 中；B 在调用C 时，发现 feature1 环境存在 C1 应用，又会返回到 feature1 环境中，依次类推，确保了流量会在 feature1 环境中闭环。

这里我们也可以计算一下这套方案的成本，10 个节点使用 MSE 专业版，一年的成本是 0.4 * 24 * 365 * 10 = 3504，总成本就是 16800 + 3504，差不多是 2 万元，使用 MSE 能节省 4 万多的成本，而且你的开发环境和应用越多，节省的成本越多。

而且，在这个过程中，您不需要修改任何代码和配置，直接接入 MSE 微服务治理即可使用，不会给您增加任何开发成本。

如何使用 MSE 开发环境隔离

场景分析

在描述如何使用开发环境隔离之前，我们先分析一下目前常用的开发环境具体的场景，这里选了三种典型的场景。

场景一

所有的开发环境都在本地，或者说公司内自建的 IDC ，这类场景下开发环境的所有应用都部署在本地的机房。

场景二

公司通过专线打通了办公网与阿里云上的 VPC ，两边的网络实现了互联互通。

开发、测试环境主要部署在阿里云，但是正在开发的工程，有一部分是跑在本地办公网的，甚至是直接跑在开发同事的个人电脑上。

场景三

公司内并没有专线来打通了办公网与阿里云上的 VPC。

但是希望能让本地启动的应用，连接上阿里云上的开发测试集群，并且实现精准的流量隔离。

首先说一下结论，这上面的三个场景，目前都是可以完整的支持的。其中场景一和场景二都不涉及到网络打通这部分的内容,其实只需要根据基本的 MSE 接入方式和 MSE 打标方式即可直接使用起来。

场景三比场景一和场景二多了一个网络打通的功能，所以会多一个端云互联的步骤，这里面会要求注册中心需要使用 MSE 提供的 Nacos。

在这个最佳实践中，我们会以场景三来实操。如果您的使用场景不是场景三，那么您可以忽略端云互联部分的步骤和相关的前置操作。

一、开通 MSE 微服务治理专业版

开通 MSE 治理

登录 MSE 治理中心控制台，如果您尚未开通 MSE 微服务治理，请根据提示开通专业版。如果您已经开通了 MSE 微服务治理基础版，请根据概览页中右侧的提示，升级到专业版。

二、完成基线环境接入

1.接入 MSE 治理

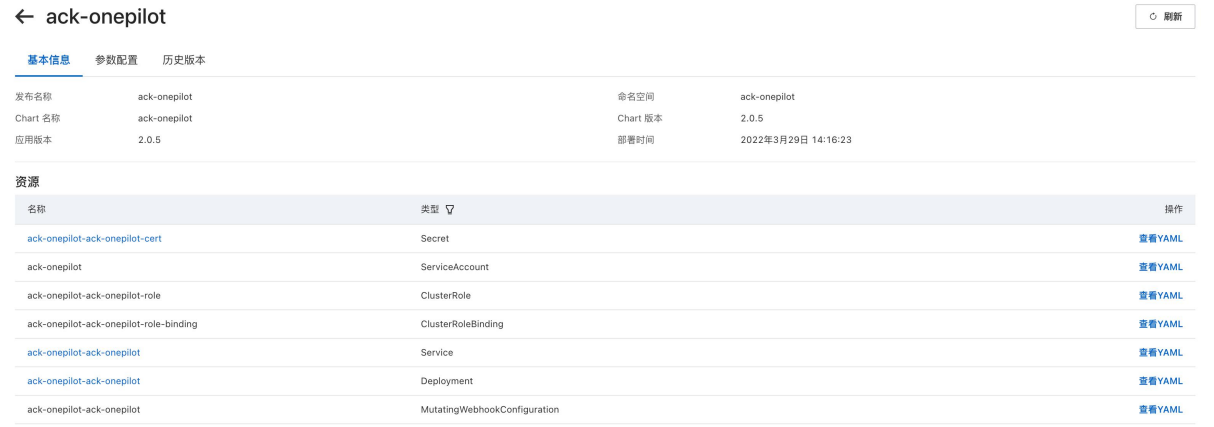
首先，您需要将基线环境的所有应用接入到 MSE 中，接入方式与您开发环境中应用部署环境有关。这里我们使用的是阿里云容器服务 ACK。更多接入场景请参考：MSE 帮助文档 MSE 微服务治理快速入门。

1.在 ACK 中安装 MSE 治理中心组件
a.登录容器服务控制台。
b.在左侧导航栏单击**市场 > 应用目录**。
c.在**应用目录**页面搜索并单击 ack-onepilot。
d.在 **ack-onepilot**页面右侧**集群**列表中选择集群，然后单击**创建**。



安装 MSE 微服务治理组件大约需要 2 分钟，请耐心等待。

创建成功后，会自动跳转到目标集群的**发布**页面，检查安装结果。如果出现以下页面，展示相关资源，则说明安装成功。



2.为 ACK 命名空间中的应用开启 MSE 微服务治理

a.登录 MSE 治理中心控制台。

b.在左侧导航栏选择**微服务治理中心 > K8s 集群列表**。

c.在 **K8s 集群列表**页面搜索框列表中选择**集群名称**或**集群 ID**，然后输入相应的关键字，单击 图标。

d.单击目标集群**操作**列的**管理**。

e.在**集群详情**页面命名空间列表区域，单击目标命名空间**操作**列下的**开启微服务治理**。(如果您的基线环境部署在 default 这个 namespace 中，则目标命名空间为 default)

f.在**开启微服务治理**对话框中单击**确认**。

2.部署基线应用

首先，我们将分别部署 spring-cloud-zuul、spring-cloud-a、spring-cloud-b、spring-cloud-c 这四个业务应用，模拟出一个真实的调用链路。

Demo 应用的结构图下图，应用之间的调用，既包含了 Spring Cloud 的调用，也包含了 Dubbo 的调用，覆盖了当前市面上最常用的两种微服务框架。这些应用都是最简单的 Spring Cloud 、Dubbo 的标准用法，您也可以直接在 https://github.com/aliyun/alibabacloud-microservice-demo/tree/master/mse-simple-demo 项目上查看源码。



您可以使用 kubectl 或者直接使用 ACK 控制台来部署应用。部署所使用的 yaml 文件如下。

注意，上文中提到，使用端云互联的前提是注册中心使用的是 MSE 中的 Nacos，所以请您在部署之前修改 yaml 文件中的 spring.cloud.nacos.discovery.server-addr 和 dubbo.registry.address 为您自己购买的 MSE Nacos 地址，否则应用是无法正常运行的。若您使用的是 MSE Nacos 域名为公网域名，还需要确保开启了白名单。

```
# 部署业务应用
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-zuul
spec:
  selector:
    matchLabels:
      app: spring-cloud-zuul
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-zuul
      labels:
        app: spring-cloud-zuul
    spec:
      containers:
        - env:
            - name: JAVA_HOME
```

```
                value: /usr/lib/jvm/java-1.8-openjdk/jre
              - name: spring.cloud.nacos.discovery.server-addr
                value: 'mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
            image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-zuul:1.0.0
            imagePullPolicy: Always
            name: spring-cloud-zuul
            ports:
              - containerPort: 20000

---
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/alibaba-cloud-loadbalancer-spec: slb.s1.small
    service.beta.kubernetes.io/alicloud-loadbalancer-address-type: internet
  name: zuul-slb
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 20000
  selector:
    app: spring-cloud-zuul
  type: LoadBalancer
status:
  loadBalancer: {}

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-cloud-a
spec:
```

```
  selector:
    matchLabels:
      app: spring-cloud-a
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-a
      labels:
        app: spring-cloud-a
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: spring.cloud.nacos.discovery.server-addr
              value: 'mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
            - name: dubbo.registry.address
              value: 'nacos://mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-a:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-a
          ports:
            - containerPort: 20001
          livenessProbe:
            tcpSocket:
              port: 20001
            initialDelaySeconds: 10
            periodSeconds: 30


---
apiVersion: apps/v1
kind: Deployment
metadata:
```

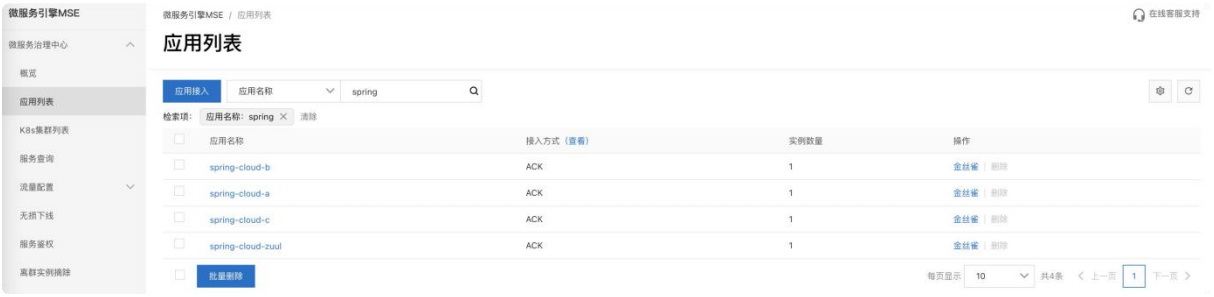
```
  name: spring-cloud-b
spec:
  selector:
    matchLabels:
      app: spring-cloud-b
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-b
      labels:
        app: spring-cloud-b
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: spring.cloud.nacos.discovery.server-addr
              value: 'mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
            - name: dubbo.registry.address
              value: 'nacos://mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-b:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-b
          ports:
            - containerPort: 20002
          livenessProbe:
            tcpSocket:
              port: 20002
            initialDelaySeconds: 10
            periodSeconds: 30

---
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: spring-cloud-c
spec:
  selector:
    matchLabels:
      app: spring-cloud-c
  template:
    metadata:
      annotations:
        msePilotCreateAppName: spring-cloud-c
      labels:
        app: spring-cloud-c
    spec:
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: spring.cloud.nacos.discovery.server-addr
              value: 'mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
            - name: dubbo.registry.address
              value: 'nacos://mse-xxxxxxx-nacos-ans.mse.aliyuncs.com:8848'
          image: registry.cn-shanghai.aliyuncs.com/yizhan/spring-cloud-c:1.0.0
          imagePullPolicy: Always
          name: spring-cloud-c
          ports:
            - containerPort: 20003
          livenessProbe:
            tcpSocket:
              port: 20003
            initialDelaySeconds: 10
            periodSeconds: 30
```

3.验证基线环境接入成功

完成上述步骤后，您的基线环境就已经部署好了。您可以在 MSE 控制台中找到对应的 Region 查看应用列表，以及应用详情页的节点情况。



在本地配置好 K8s 集群对应的 kubeconfig 文件，执行命令，结果如下：

```
➜ ~ kubectl get svc,deploy
NAME                          TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE
service/kubernetes    ClusterIP      192.168.0.1     <none>
443/TCP          7d
service/zuul-slb      LoadBalancer   192.168.87.95   47.94.143.53
80:31983/TCP     9m30s
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/spring-cloud-a        1/1     1            1
9m30s
deployment.apps/spring-cloud-b        1/1     1            1
9m30s
deployment.apps/spring-cloud-c        1/1     1            1
9m30s
deployment.apps/spring-cloud-zuul     1/1     1            1
9m30s
```

在这里我们执行一下 curl http://47.94.143.53:80/A/a 发起调用，并查看返回结果。

```
➜ ~ curl http://47.94.143.53:80/A/a
A[10.242.0.90] -> B[10.242.0.91] -> C[10.242.0.152]%
```

三、IDEA 启动的应用接入 feature 环境

在这一步中，我们将演示如何在网络没有打通的情况下，将你本机启动的应用接入到 feature 环境。首先您需要将您 K8s 集群的 kubeconfig 文件保存到本机，并进行如下操作。

1.下载源码

本工程所有源码都在：https://github.com/aliyun/alibabacloud-microservice-demo 中，将代码 git clone 到本地，并且找到 mse-simple-demo 文件夹中的 A、B、C、gateway 四个应用，就是本次最佳实践所使用的公测。

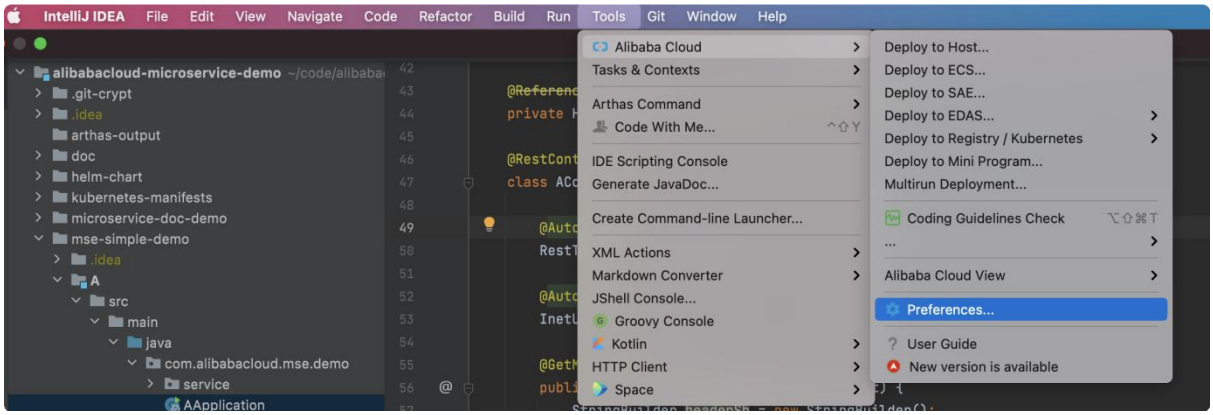
2.安装 CloudToolkit 插件

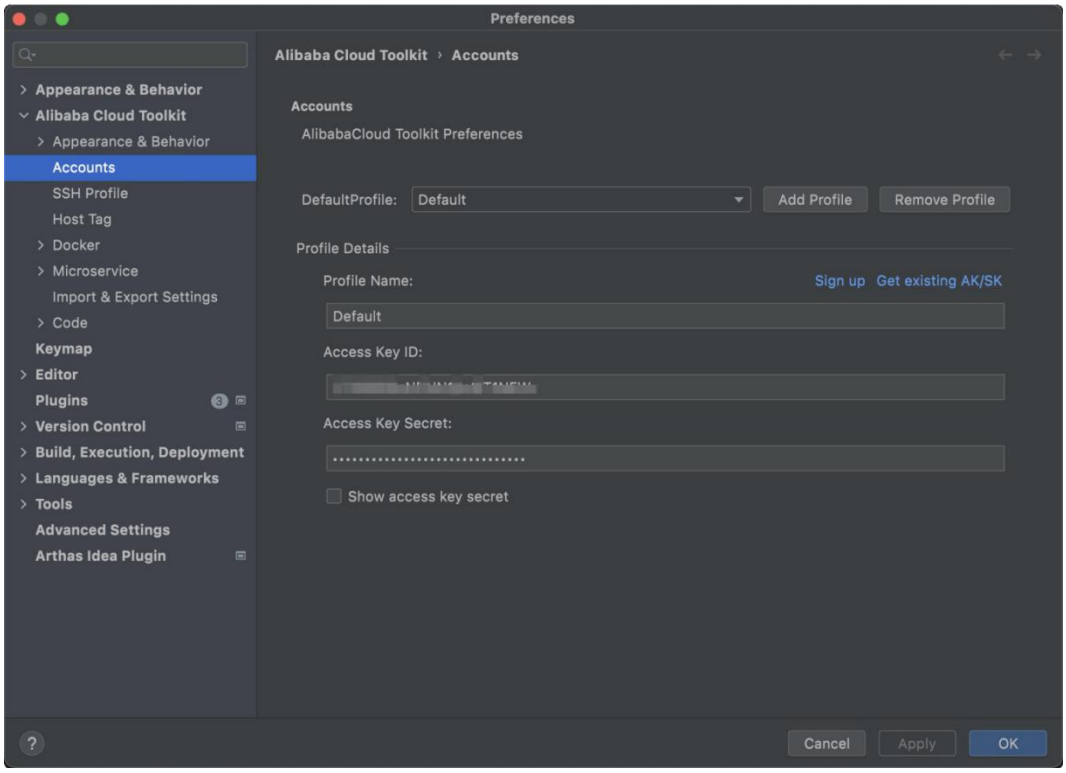
安装最新版本的 Cloud Toolkit，安装详情请参考官网：
https://www.aliyun.com/product/cloudtoolkit

3.填写阿里云 AK、SK

由于使用端云互联功能的时候，需要访问您 MSE 的资源，所以这里需要您填写您的 AK、SK，并确保此 AK、SK 拥有访问 MSE 资源的权限。

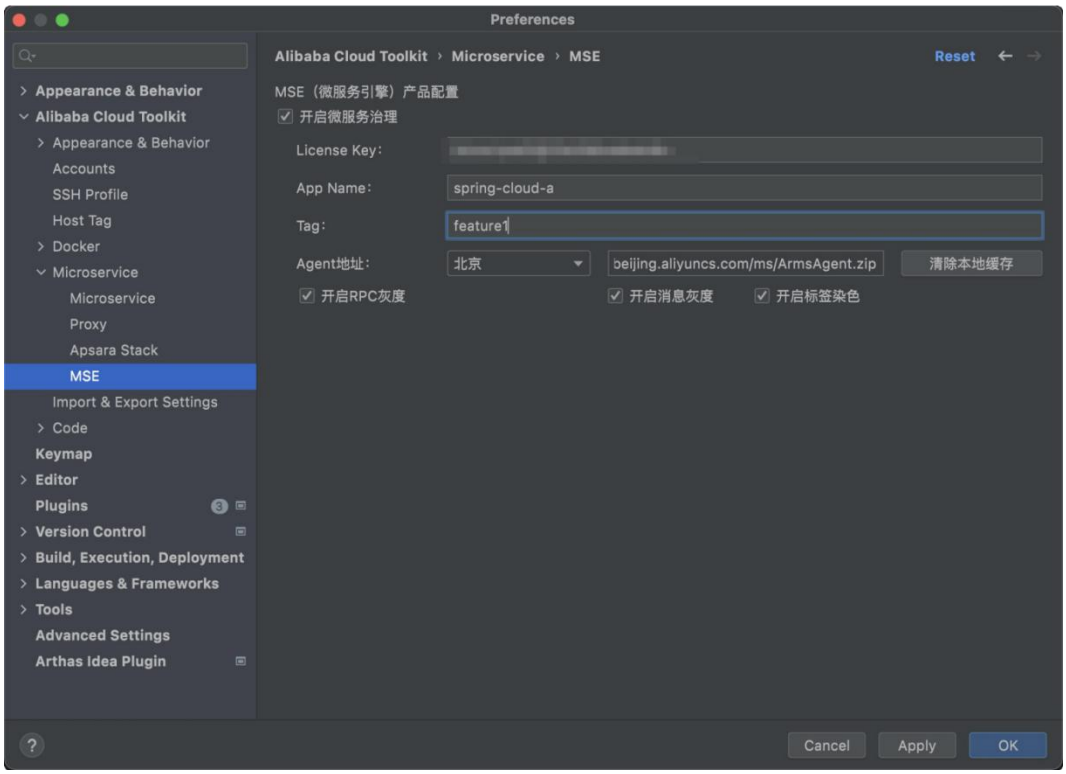
点击 IDEA 的 **Tools** 中找到 **Preference** ，找到 **Alibaba Cloud Toolkit** 中 **Accounts** ，配置 Access Key ID 和 Access Key Secret 信息，并点击保存。





4.配置 MSE 参数

点击 IDEA 的 **Tools** 中找到 **Preference** ，找到 **Alibaba Cloud Toolkit** 中 **Microservice** 下的 **MSE**，点击**开启微服务治理**，并安装下图的方式进行配置即可。



对图中的几个参数做一下说明：

1.LicenseKey

您阿里云账号对应的 MSE 产品的 LicenseKey ，请在 https://mse.console.aliyun.com/#/msc/app/accessType 中的选择 ECS 集群，在**安装 MSEAgent** 章节找到 LicenseKey 的值。
注意：请您做好 LicenseKey 的保密工作。
注意：各个 Region 的 LicenseKey 值可能不一致，请选择对应的 Region，并和基线环境接入的 Region 保持一致。

2.App Name

应用在接入 MSE 时所使用的应用名，请根据实际业务情况进行配置，注意这个值需要和本次所启动的应用保持一致。

3.Tag

此应用所属的环境 Tag，基线不用填，其他请根据实际业务情况进行填写。如果此应用属于 feature1 环境，请填写 feature1。

4.Agent 地址

选择自己应用所在的地域，需要和 LienseKey 所在的地域、以及基线环境接入的地域 都保持一致。

5.开启 RPC 灰度 ✓

支持对 Spring Cloud 和 Dubbo 近 5 年内的所有版本的流量进行精准控制。默认情况下请开启，除非您明确知道关闭此选项的使用场景，否则请勿关闭此选项。

6.开启标签染色 ✓

推荐开启，开启后经过此应用的流量就只会在对应的 Tag 环境中流转。

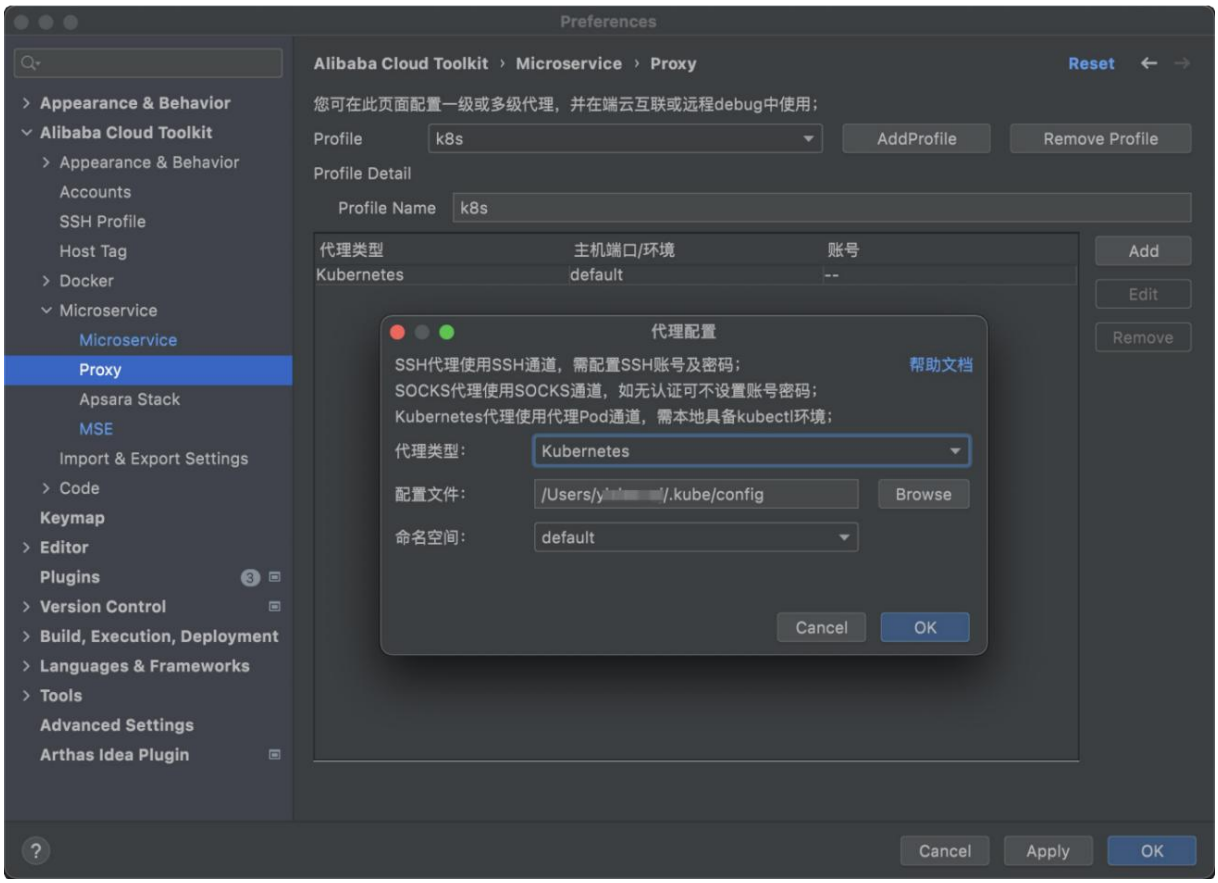
7.开启消息灰度 ✓

请根据业务实际情况选择是否开启，目前仅支持 RocketMQ 4.5 及以上版本。更多消息灰度相关的信息，请参考全链路灰度，消息部分。

5.配置端云互联参数

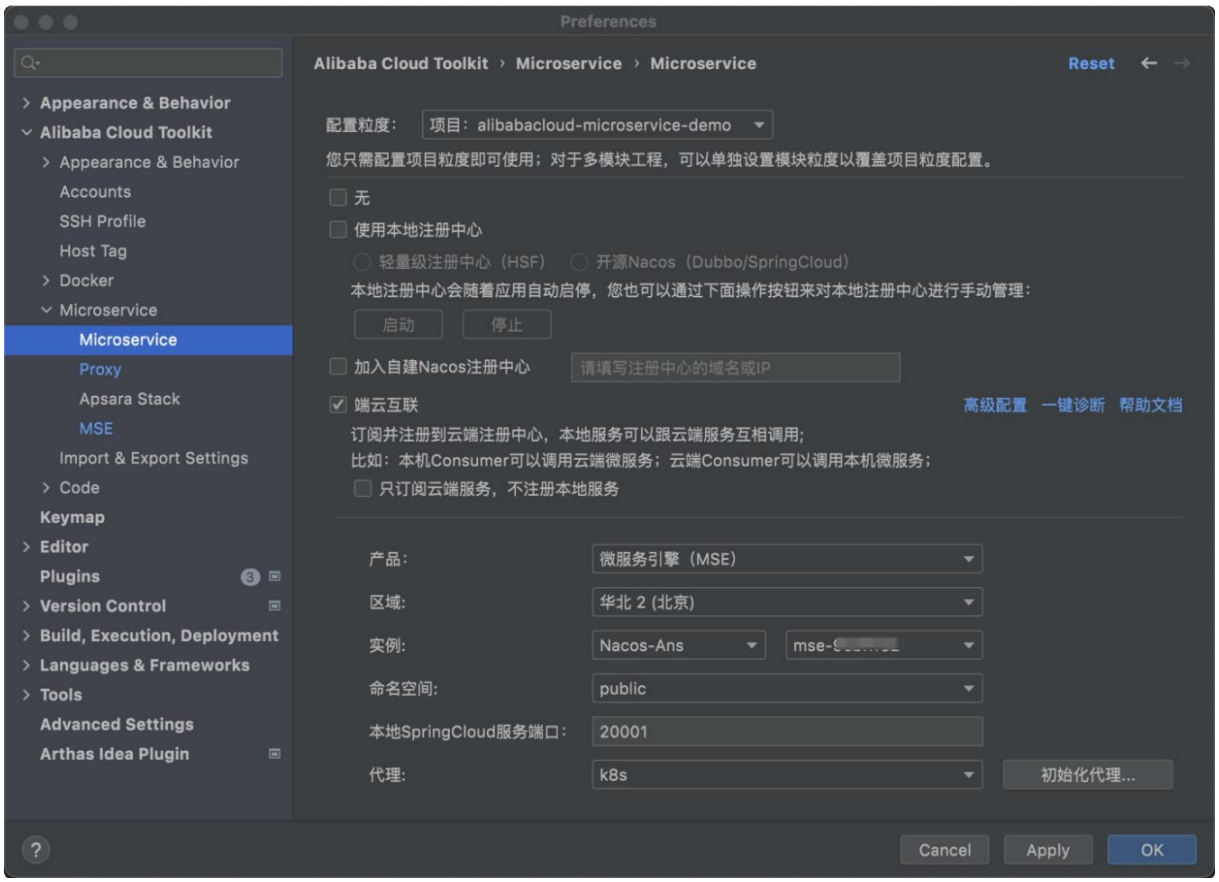
我们已经知道，在网络未打通的时候，联通本机环境和云上 K8s 集群需要使用 端云互联功能，所以这里我们需要配置一下端云互联。

首先需要配置一下代理模式为 K8s，点击 IDEA 的 **Tools** 中找到 **Preference** ，找到 **Alibaba Cloud Toolkit** 中 **Microservice** 下的 **Proxy** ，点击 **AddProfile** 增加一个名称为 k8s 的代理。然后点击右侧的 **Add** 按钮，选择**代理类型**为 Kubernetes，并选择正确的**配置文件地址**和**命名空间**。



然后，点击 IDEA 的 **Tools** 中找到 **Preference** ，找到 **Alibaba Cloud Toolkit** 中 **Microservice** 下的 **MicroService** ，找到**端云互联**功能打 ✓。选择产品为**微服务引擎 MSE**，并选择

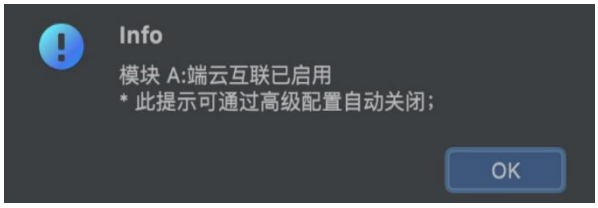
与上面部署时一致的 **Region** 、**实例**和**命名空间**，代理选择刚刚配置好的 **K8s**，如果您的应用是 Spring Cloud 应用，还必须在**本地 Spring Cloud 服务端**口中配置 Tomcat 的启动端口。

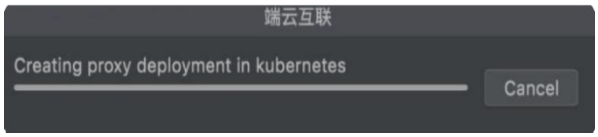


四、启动应用，轻松开始联调和测试

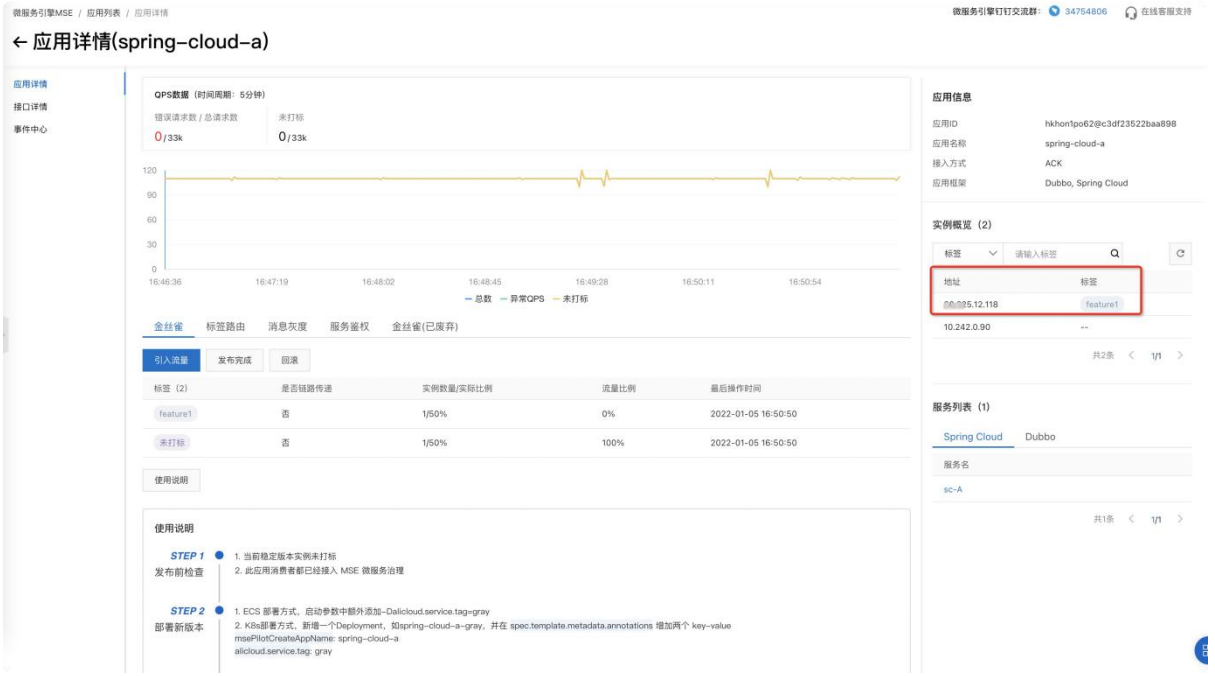
1.验证接入成功

完成上述配置之后，我们启动应用。首先，启动应用的时候，您会看到这两个提示，证明端云互联功能已经生效。





同时为了验证 MSE 微服务治理是否接入成功,我们可以登录 MSE 控制台的应用列表界面进行查看。在控制台中我们也可以看到，我的本机应用已经成功接入到 MSE，并且成功打上了 feature1 的标签。



2.发起流量调用

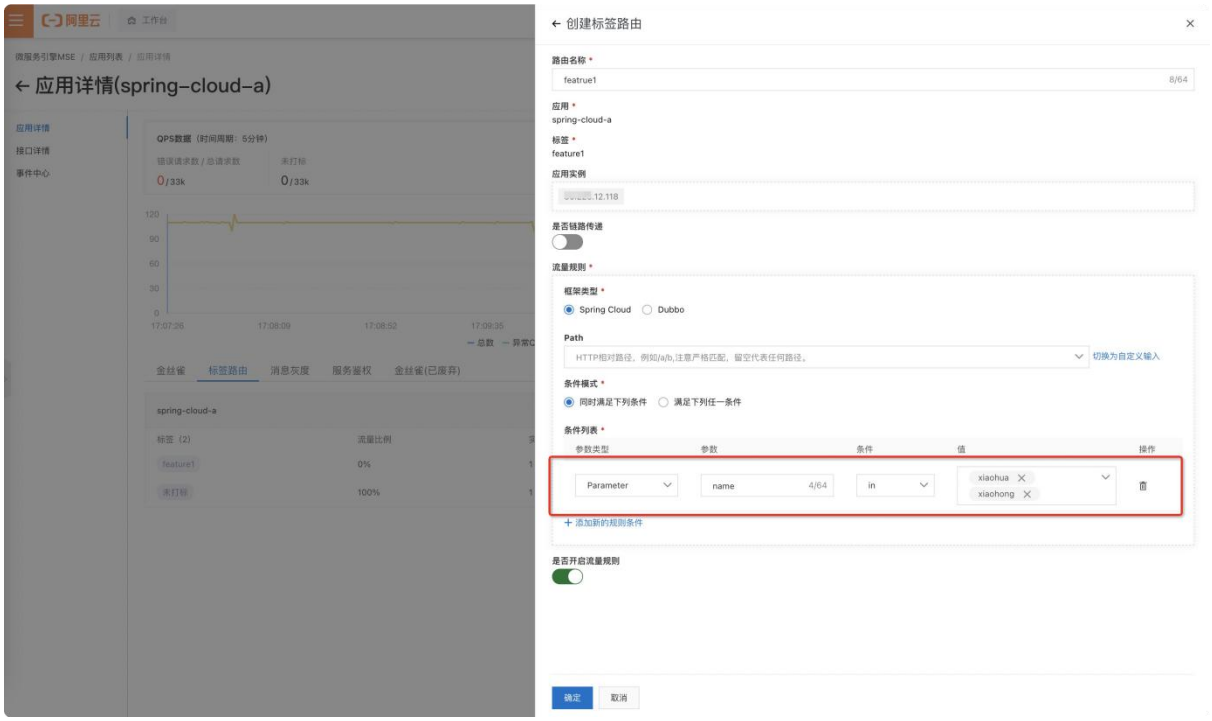
假设我们发往网关的请求是 http 请求，希望这个请求再 feature1 中完成闭环，只需要在请求的 header 中添加 x-mse-tag=feature1 即可，流量会自动在 feature1 环境内完成闭环。注意这里的 key 为 x-mse-tag 为固定值，feature1 则需要和环境的标签(即上文中配置的 alicloud.service.tag)保持一致。

如果您的请求来源为 Dubbo，则需要在 RpcContext 中增加 Attachment ，内容也是 x-mse-tag=feature1。

```
➜ ~ curl http://47.94.143.53:80/A/a
A[10.242.0.90] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl http://47.94.143.53:80/A/a
A[10.242.0.90] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl -H"x-mse-tag:feature1" http://47.94.143.53:80/A/a
Afeature1[xx.xxx.12.118] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl -H"x-mse-tag:feature1" http://47.94.143.53:80/A/a
```

```
Afeature1[xx.xxx.12.118] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl -H"x-mse-tag:feature1" http://47.94.143.53:80/A/a
Afeature1[xx.xxx.12.118] -> B[10.242.0.91] -> C[10.242.0.152]%
```

如果您不方便在请求中增加 header 的话，还可以在 MSE 控制台配置规则，比如设置成 name=xiaohong 或 xiaohua 的流量进入到 feature1 环境。如下图所示。



开启流量规则之后，我们继续验证，可以看到，满足条件的请求会被转发到 feature1 环境。

```
➜ ~ curl http://47.94.143.53:80/A/a
A[10.242.0.90] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl http://47.94.143.53:80/A/a\?name\=xiaohong
Afeature1[30.225.12.118] -> B[10.242.0.91] -> C[10.242.0.152]%
➜ ~ curl http://47.94.143.53:80/A/a\?name\=xiaohua
Afeature1[30.225.12.118] -> B[10.242.0.91] -> C[10.242.0.152]%
```

特别地，如果您的网关应用不属于 Java 体系，则需要在网关层配置一下规则，目前已经支持 MSE 云原生网关，Nginx，K8s Ingress 等，详细的接入方式可以参考 4.2 中相关章节的内容。

4.7 微服务无缝迁移上云实践

本节是阿里云微服务引擎 MSE 在所提供的应用无缝迁移上云方案最佳实践介绍。接下来，将通过 3 个应用演示 MSE 提供的基于 Java Agent 技术利用双注册双订阅实现注册中心平滑迁移功能。

前提条件

开启 MSE 微服务治理

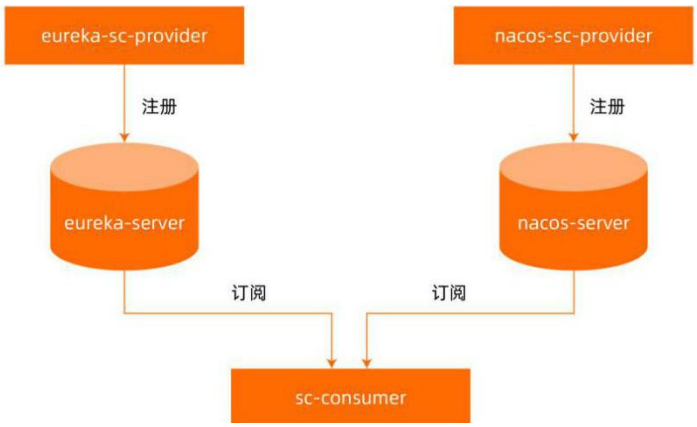
- 已创建 Kubernetes 集群，请参见创建 Kubernetes 托管版集群。
- 已开通 MSE 微服务治理专业版，请参见开通 MSE 微服务治理。

准备工作

注意，本实践所使用的 Agent 目前还在灰度中，需要对应用 Agent 进行灰度升级，升级文档：https://help.aliyun.com/document_detail/392373.html

本次实践所使用的Agent下载地址请到文档：https://help.aliyun.com/document_detail/421680.html，中 Spring Cloud 应用实现 Eureka 和 Nacos 双注册栏目中获得，实践请根据应用部署 Region 进行选择对应地址。

应用部署架构图



部署 Demo 源代码解析

本次实践的应用除了两个独立的注册中心eureka-server 和 nacos-server 以外，另外包含三个应用，分别是服务消费者应用sc-consumer 以及服务提供者应用eureka-sc-provider 和 nacos-sc-provider。

sc-consumer 通过接入MSE 添加双注册和双订阅相关配置的同时注册并从 eureka-server 和 nacos-server 中订阅服务，其核心源代码如下：

```
//ConsumerApplication.java文件
    @FeignClient(name = "eureka-service-provider")
     public interface EurekaProvider {
          @RequestMapping(value = "/user", method = RequestMethod.GET)
          String user();
    }
     @FeignClient(name = "nacos-service-provider")
     public interface NacosProvider {
          @RequestMapping(value = "/user", method = RequestMethod.GET)
          String user();
    }
     @Bean
     @LoadBalanced
     public RestTemplate restTemplate(){
          return new RestTemplate();
    }
//TestController.java
    /**
      * 通过 FeignClient方式调用注册在 Eureka 上服务提供者的 user 接口
      * @return
      */
     @RequestMapping(value = "/eureka/feign", method = RequestMethod.GET)
     public String eurekaFeign() {
          return eurekaProvider.user();
    }
```

```
    /**
     * 通过 RestTemplate方式调用注册在 Eureka 上服务提供者的 user 接口
     * @return
     */
    @RequestMapping(value = "/eureka/rest", method = RequestMethod.GET)
    public String eurekaRest() {
        return restTemplate.getForObject("http://eureka-serviceprovider/user/", String.
class);
    }
    /**
     * 通过 FeignClient方式调用注册在 Nacos 上服务提供者的 user 接口
     * @return
     */
    @RequestMapping(value = "/nacos/feign", method = RequestMethod.GET)
    public String nacosFeign() {
  return nacosProvider.user();
    }
    /**
     * 通过 RestTemplate方式调用注册在 Nacos 上服务提供者的 user 接口
     * @return
     */
    @RequestMapping(value = "/nacos/rest", method = RequestMethod.GET)
    public String nacosRest() {
        return restTemplate.getForObject("http://nacos-serviceprovider/user/", String.c
lass);
    }
```

eureka-provider 和 nacos-provider 的是一样的，只是配置的注册中心信息不同，其都是提供了一个如下所示的服务接口：

```
@RestController
    @RefreshScope
    class SampleController {
        @Value("${server.port}")
```

```
        String serverPort;
        @RequestMapping("/user")
        public String simple() {
            System.out.println("========================== " +
sdf.format(System.currentTimeMillis()));
            return "Hello from [" + this.serverPort + "], this is service
in Nacos!";
        }
    }
```

部署 Demo 应用程序

将下面的内容保存到一个文件中，假设取名为 `immigrate.yaml` ，并执行 `kubectl apply -f immigrate.yaml` 进行应用部署。

```
# eureka-sc-consumer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sc-consumer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sc-consumer
  template:
    metadata:
      annotations:
        msePilotAutoEnable: 'on'
        msePilotCreateAppName: sc-consumer
      labels:
        app: sc-consumer
    spec:
```

```
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: LANG
              value: C.UTF-8
            - name: additional_nacos_address
              value: nacos-server:8848
          image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-consumer:migr
ate-eureka-1.0
          imagePullPolicy: Always
          name: sc-consumer
      imagePullSecrets:
        - name: mse-demo-hz

# eureka-sc-provider
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: eureka-sc-provider
spec:
  replicas: 1
  selector:
    matchLabels:
      app: eureka-sc-provider
  template:
    metadata:
      annotations:
        msePilotAutoEnable: 'on'
        msePilotCreateAppName: eureka-sc-provider
      labels:
        app: eureka-sc-provider
    spec:
```

```
      containers:
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: LANG
              value: C.UTF-8
          image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-provider:migrat
e-eureka-1.0
          imagePullPolicy: Always
          name: eureka-sc-provider
      imagePullSecrets:
        - name: mse-demo-hz


# nacos-sc-provider
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nacos-sc-provider
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nacos-sc-provider
  template:
    metadata:
      annotations:
        msePilotAutoEnable: 'on'
        msePilotCreateAppName: nacos-sc-provider
      labels:
        app: nacos-sc-provider
    spec:
      containers:
```

```
        - env:
            - name: JAVA_HOME
              value: /usr/lib/jvm/java-1.8-openjdk/jre
            - name: LANG
              value: C.UTF-8
          image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-provider:migrat
e-nacos-1.0
          imagePullPolicy: Always
          name: nacos-sc-provider
      imagePullSecrets:
        - name: mse-demo-hz


# Nacos Server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nacos-server
  name: nacos-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      labels:
        app: nacos-server
    spec:
      containers:
      - env:
        - name: MODE
```

```
            value: standalone
          image: nacos/nacos-server:latest
          imagePullPolicy: Always
          name: nacos-server
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
        dnsPolicy: ClusterFirst
        restartPolicy: Always

# Eureka Server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: eureka-server
  name: eureka-server
spec:
  replicas: 1
  selector:
    matchLabels:
      app: eureka-server
  template:
    metadata:
      labels:
        app: eureka-server
    spec:
      containers:
      - env:
        image: springcloud/eureka:latest
        imagePullPolicy: Always
        name: eureka-server
```

```
          resources:
            requests:
              cpu: 250m
              memory: 512Mi
        dnsPolicy: ClusterFirst
        restartPolicy: Always


# Nacos Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
  selector:
    app: nacos-server
  type: ClusterIP

# Nacos Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-slb
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
```

```
  selector:
    app: nacos-server
  type: LoadBalancer


# Eureka Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: eureka-server
spec:
  ports:
  - port: 8761
    protocol: TCP
    targetPort: 8761
  selector:
    app: eureka-server
  type: ClusterIP

# Eureka Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: eureka-slb
spec:
  ports:
  - port: 8761
    protocol: TCP
    targetPort: 8761
  selector:
    app: eureka-server
  type: LoadBalancer
```

```
# sc-consumer Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: consumer-slb
spec:
  ports:
  - port: 18099
    protocol: TCP
    targetPort: 18099
  selector:
    app: sc-consumer
  type: LoadBalancer
```

结果验证一：应用部署

通过 `kubectl apply -f immigrate.yaml` 命名进行测试 demo 部署，正常会拉镜像部署如下图所示 5 个应用。

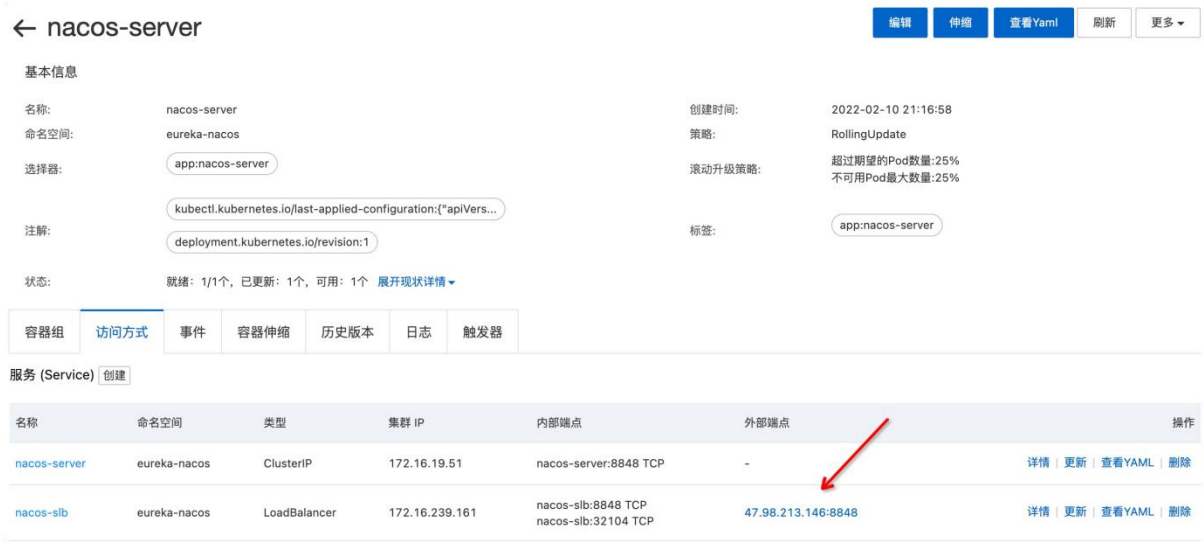


名称	标签	容器组数量	镜像	创建时间	操作
eureka-sc-provider		1/1	registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-provider:migrate-eureka-1.0	2022-02-10 21:16:57	详情 | 编辑 | 伸缩 | 监控 | 更多
eureka-server	app:eureka-server	1/1	springcloud/eureka:latest	2022-02-10 21:16:58	详情 | 编辑 | 伸缩 | 监控 | 更多
nacos-sc-provider		1/1	registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-provider:migrate-nacos-1.0	2022-02-10 21:16:57	详情 | 编辑 | 伸缩 | 监控 | 更多
nacos-server	app:nacos-server	1/1	nacos/nacos-server:latest	2022-02-10 21:16:58	详情 | 编辑 | 伸缩 | 监控 | 更多
sc-consumer		1/1	registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/sc-consumer:migrate-eureka-1.0	2022-02-10 21:16:57	详情 | 编辑 | 伸缩 | 监控 | 更多

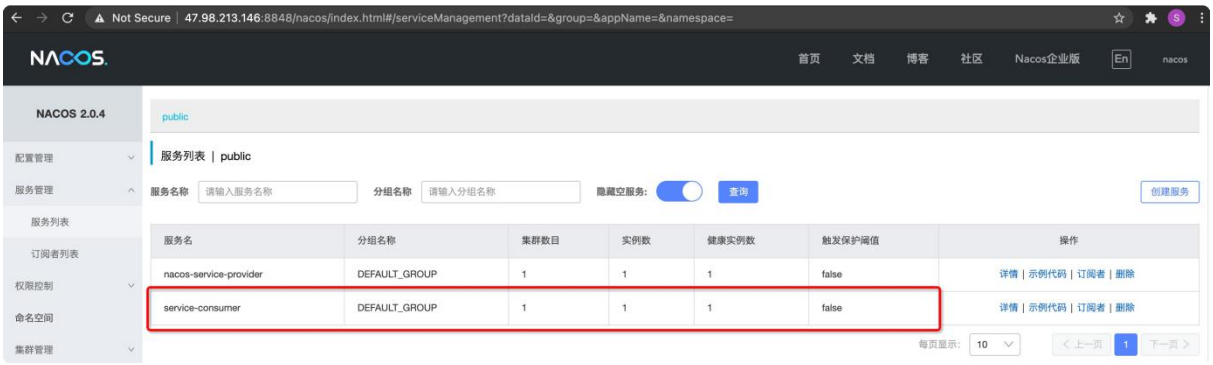
结果验证二：双注册

由于仅在 sc-consumer 应用中配置了双注册配置，可通过登录 nacos-server 和 eureka-server 控制台查看相关服务是否在两边都完成了注册。

通过如下图所示在 nacos-server 的外部端点访问 nacos 控制台，用户名密码都为默认的：nacos。

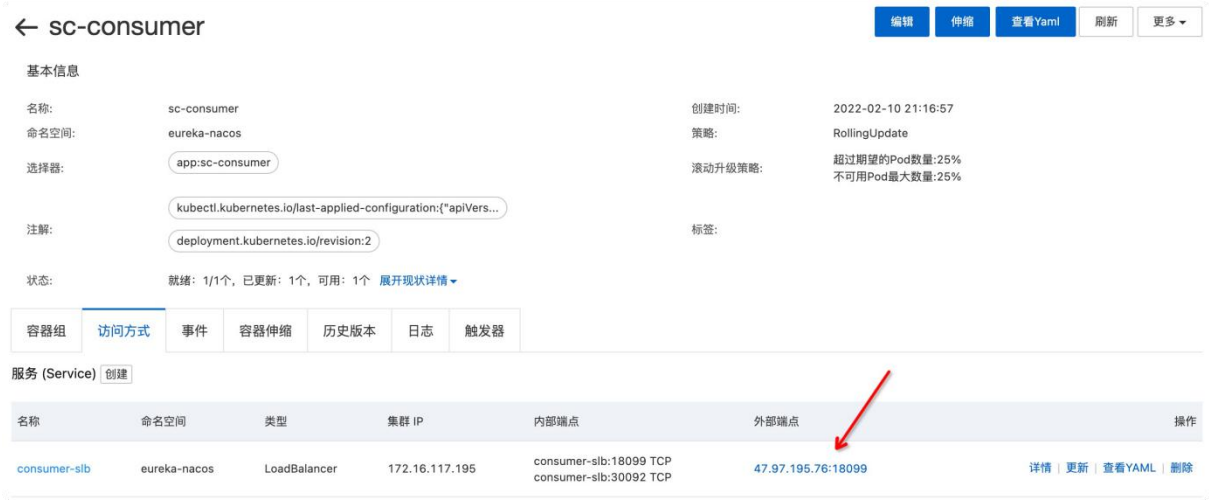


通过控制台，如下图所示可见 sc-consumer 已经完成了在 nacos-server 上的注册，sc-consumer 在 eureka-server 上的注册效果通过类似的方法可查看。



结果验证三：双订阅

通过如下图所示在 sc-consumer 的外部端点对 eureka-sc-provider 和 nacos-sc-provider 分别发起调用验证双订阅效果。



通过 RestTemplate 客户端调用 eureka-sc-provider 的 user 方法，如下图所示可正常返回：



通过 FeignClient 客户端调用 eureka-sc-provider 的 user 方法，如下图所示可正常返回：



通过 RestTemplate 客户端调用 nacos-sc-provider 的 user 方法，如下图所示可正常返回：



通过 FeignClient 客户端调用 nacos-sc-provider 的 user 方法，如下图所示可正常返回：



4.8 如何快速构建服务发现的高可用能力

背景

本文是阿里云微服务引擎 MSE 在服务发现高可用的最佳实践介绍。演示的应用架构是由后端的微服务应用实例（Spring Cloud）构成。具体的后端调用链路有 Spring Cloud Consumer 调用 Spring Cloud Provider,这些应用中的服务之间通过 Nacos 注册中心实现服务注册与发现。

动手实践

前提条件

- 已创建 Kubernetes 集群，请参见创建 Kubernetes 托管版集群。
- 已开通 MSE 微服务治理专业版，请参见开通 MSE 微服务治理。

准备工作

开启 MSE 微服务治理

1.开通微服务治理专业版：
a.单击开通 MSE 微服务治理。
b.微服务治理**版本**选择**专业版**，选中**服务协议**，然后单击**立即开通**。关于微服务治理的计费详情，请参见价格说明。

2.安装 MSE 微服务治理组件：
a.在容器服务控制台左侧导航栏中，选择**市场 > 应用目录**。
b.在**应用目录**页面搜索框中输入**ack-onepilot**，单击搜索图标，然后单击组件。
c.在**详情**页面选择开通该组件的**集群**，然后单击**创建**。安装完成后，在命名空间 **ack-onepilot** **找到 ack-onepilot** 应用，表示安装成功。

3.为应用开启微服务治理：

a.登录 MSE 治理中心控制台。

b.在左侧导航栏选择**微服务治理中心 > K8s 集群列表**。

c.在 **K8s 集群列表**页面搜索目标集群，单击搜索图标，然后单击目标集群**操作**列下方的**管理**。

d.在**集群详情**页面命名空间列表区域，单击目标命名空间**操作**列下方的**开启微服务治理**。

e.在**开启微服务治理**对话框中单击**确认**。

部署 Demo 应用程序

1.在容器服务控制台左侧导航栏中，单击**集群**。

2.在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。

3.在集群管理页左侧导航栏中，选择**工作负载 > 无状态**。

4.在**无状态**页面选择**命名空间**，然后单击**使用YAML 创建资源**。

5.对模板进行相关配置，完成配置后单击**创建**。本文示例中部署 sc-consumer、sc-consumer-empty、sc-provider，使用的是开源的 Nacos。

部署示例应用（springcloud）

YAML：

```
# 开启推空保护的 sc-consumer
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sc-consumer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sc-consumer
```

```
  template:
    metadata:
      annotations:
        msePilotCreateAppName: sc-consumer
      labels:
        app: sc-consumer
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: nacos-server:8848
        - name: profiler.micro.service.registry.empty.push.reject.enable
          value: "true"
        image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/demo:sc-consumer-
0.1
        imagePullPolicy: Always
        name: sc-consumer
        ports:
        - containerPort: 18091
        livenessProbe:
          tcpSocket:
            port: 18091
          initialDelaySeconds: 10
          periodSeconds: 30
# 无推空保护的 sc-consumer-empty
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sc-consumer-empty
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: sc-consumer-empty
  template:
    metadata:
      annotations:
        msePilotCreateAppName: sc-consumer-empty
      labels:
        app: sc-consumer-empty
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: nacos-server:8848
        image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/demo:sc-consumer-
0.1
        imagePullPolicy: Always
        name: sc-consumer-empty
        ports:
        - containerPort: 18091
        livenessProbe:
          tcpSocket:
            port: 18091
          initialDelaySeconds: 10
          periodSeconds: 30
# sc-provider
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sc-provider
spec:
  replicas: 1
  selector:
```

```
    matchLabels:
      app: sc-provider
  strategy:
  template:
    metadata:
      annotations:
        msePilotCreateAppName: sc-provider
      labels:
        app: sc-provider
    spec:
      containers:
      - env:
        - name: JAVA_HOME
          value: /usr/lib/jvm/java-1.8-openjdk/jre
        - name: spring.cloud.nacos.discovery.server-addr
          value: nacos-server:8848
        image: registry.cn-hangzhou.aliyuncs.com/mse-demo-hz/demo:sc-provider-0.
3
        imagePullPolicy: Always
        name: sc-provider
        ports:
        - containerPort: 18084
        livenessProbe:
          tcpSocket:
            port: 18084
          initialDelaySeconds: 10
          periodSeconds: 30
# Nacos Server
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nacos-server
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: nacos-server
  template:
    metadata:
      labels:
        app: nacos-server
    spec:
      containers:
      - env:
        - name: MODE
          value: standalone
        image: nacos/nacos-server:latest
        imagePullPolicy: Always
        name: nacos-server
      dnsPolicy: ClusterFirst
      restartPolicy: Always

# Nacos Server Service 配置
---
apiVersion: v1
kind: Service
metadata:
  name: nacos-server
spec:
  ports:
  - port: 8848
    protocol: TCP
    targetPort: 8848
  selector:
    app: nacos-server
  type: ClusterIP
```

我们只需在 Consumer 增加一个环境变量 `profiler.micro.service.registry.empty.push.reject.enable=true` ，开启注册中心的推空保护（无需升级注册中心的客户端版本，无关注册中心的实

现，支持 MSE 的 nacos、eureka 以及自建的 nacos、eureka、console）

分别给 Consumer 应用增加 SLB用于公网访问

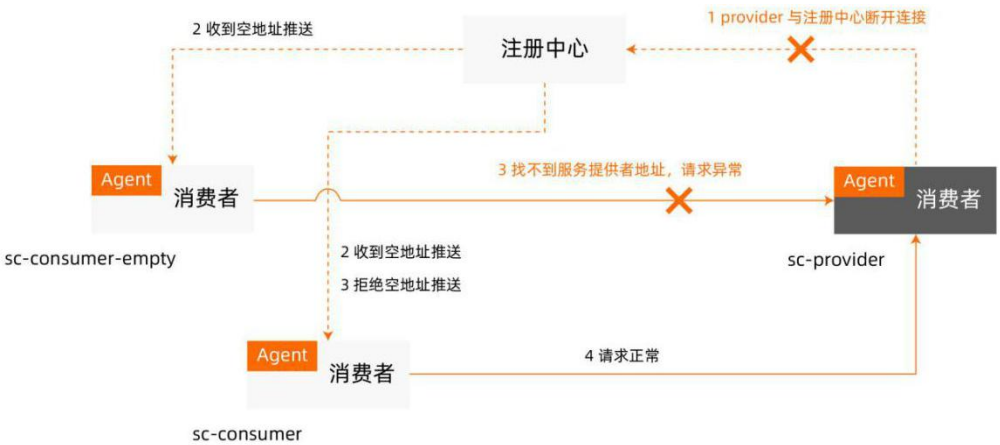




以下分别使用{sc-consumer-empty}代表 sc-consumer-empty 应用的 slb 的公网地址，{sc-consumer}代表 sc-consumer 应用的 slb 的公网地址。

应用场景

下面通过上述准备的 Demo 来分别实践以下场景：



- 编写测试脚本

```
vi  curl.sh
```

```
while :
do
        result=`curl $1 -s`
        if [[ "$result" == *"500"* ]]; then
                echo `date +%F-%T` $result
        else
                echo `date +%F-%T` $result
        fi
        sleep 0.1
done
```

- 测试，分别开两个命令行，执行如下脚本，显示如下：

```
% sh curl.sh {sc-consumer-empty}:18091/user/rest
```

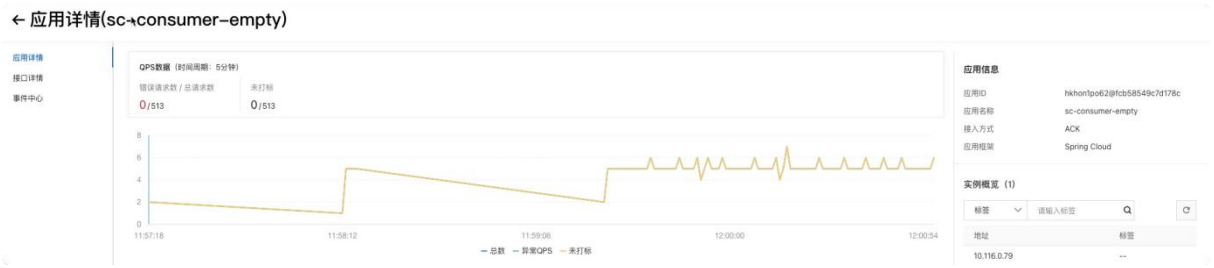
```
2022-01-19-11:58:12 Hello from [18084]10.116.0.142!
2022-01-19-11:58:12 Hello from [18084]10.116.0.142!
2022-01-19-11:58:12 Hello from [18084]10.116.0.142!
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
```

```
% sh curl.sh {sc-consumer}:18091/user/rest
```

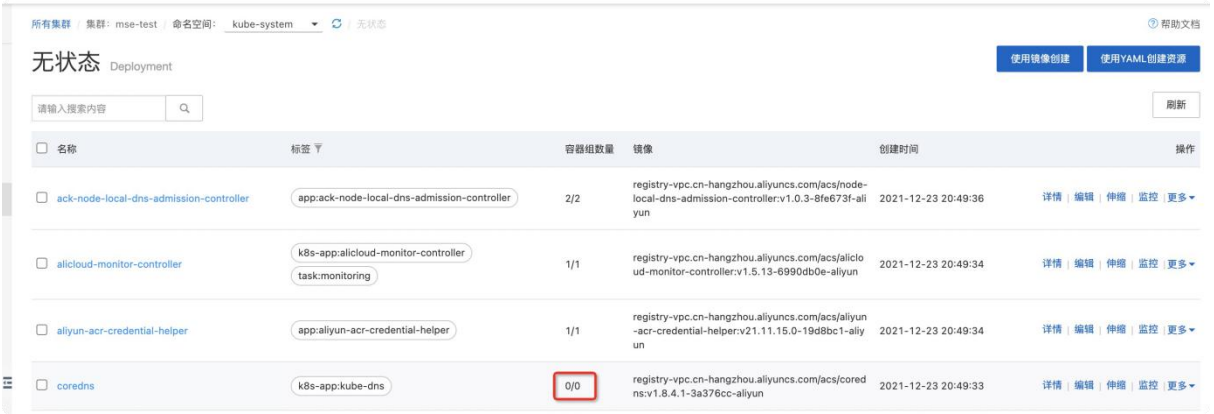
```
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
2022-01-19-11:58:13 Hello from [18084]10.116.0.142!
2022-01-19-11:58:14 Hello from [18084]10.116.0.142!
2022-01-19-11:58:14 Hello from [18084]10.116.0.142!
2022-01-19-11:58:14 Hello from [18084]10.116.0.142!
```

并保持脚本一直在调用，观察 mse 控制台分别看到如下情况：





- 将 coredns 组件缩容至数量 0，模拟 dns 网络解析异常场景。



发现实例与 nacos 的连接断开且服务列表为空

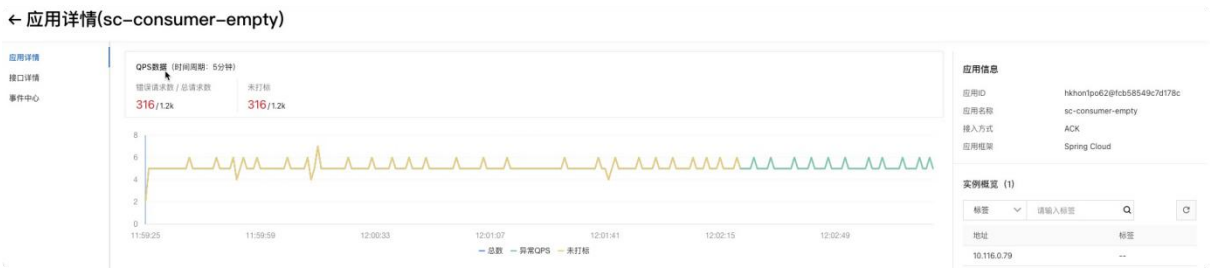


- 模拟 dns 服务恢复，将其扩容回数量 2。

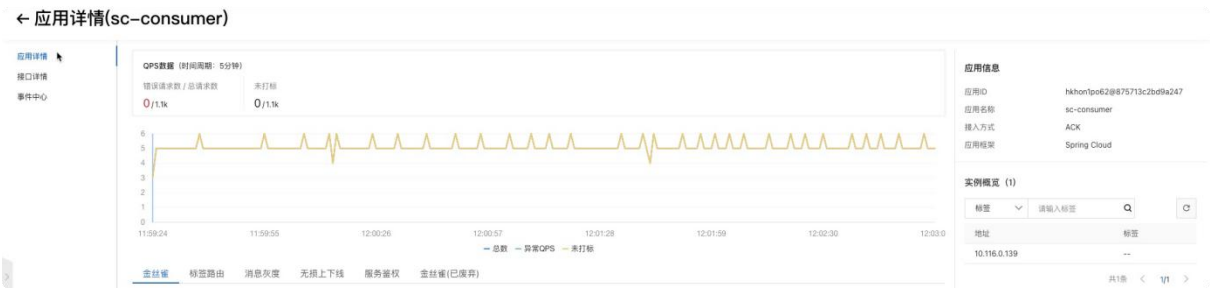


结果验证

在以上过程中保持持续的业务流量，我们发现 sc-consumer-empty 服务出现大量且持续的报错 2022-01-19-12:02:37 {"timestamp":"2022-01-19T04:02:37.597+0000","status":500,"error":"Internal Server Error","message":"com.netflix.client.ClientException: Load balancer doesnot have available server for client: mse-service-provider","path":"/user/feign"} 2022-01-19-12:02:37 {"timestamp":"2022-01-19T04:02:37.799+0000","status":500,"error":"Internal Server Error","message":"com.netflix.client.ClientException: Load balancer does not have available server for client: mse-service-provider","path":"/user/feign"} 2022-01-19-12:02:37 {"timestamp":"2022-01-19T04:02:37.993+0000","status":500,"error":"Internal Server Error","message":"com.netflix.client.ClientException: Load balancer does not have available server for client: mse-service-provider","path":"/user/feign"}



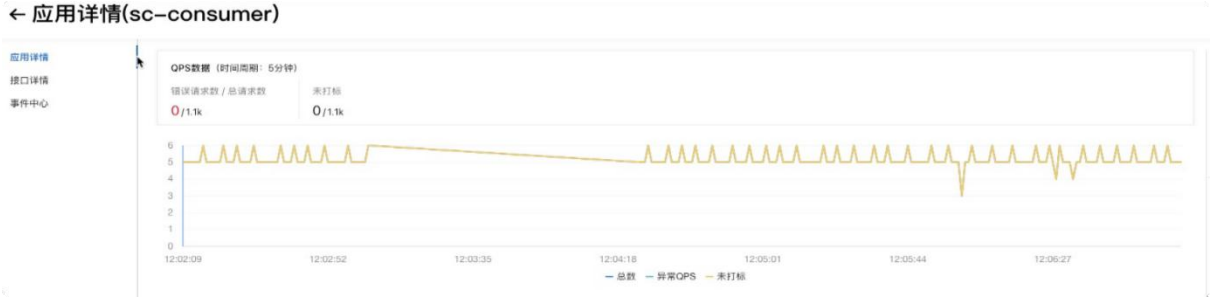
相比之下，sc-consumer 应用全流程没有任何报错。



- 只有重启了 Provider，sc-consumer-empty 才恢复正常。



相比之下，sc-consumer 应用全流程没有任何报错。



保障云上业务的永远在线，是 MSE 一直在追求的目标，本文通过面向失败设计的服务发现高可用能力的分享，以及 MSE 的服务治理能力快速构建起服务发现高可用能力的演示，模拟了线上不可预期的服务发现相关异常发生时的影响以及我们如何预防的手段，展示了一个简单的开源微服务应用应该如何构建起服务发现高可用能力。

4.9 如何在 Serverless 模式下快速使用服务治理能力

背景

Serverless 应用引擎（SAE）是面向应用的 Serverless PaaS 平台，帮助 PaaS 层用户免运维 IaaS、按量计费、低门槛微服务上云，并且提供了开箱即用的历经双 11 考验的微服务治理能力，将 Serverless 架构和微服务架构的完美结合。本文是 SAE 服务治理相关的最佳实践介绍，详细描述了 Spring Cloud 微服务应用如何部署在阿里云 Serverless 应用引擎 SAE 上，并且使用 SAE 提供的一些服务治理能力。

准备工作

前提条件

- 开通 MSE 微服务治理专业版，请参见开通 MSE 微服务治理
- 开通 SAE 服务，请参见开通 SAE

Demo 应用

Demo 代码：
服务提供者：请参见 nacos-service-provider
服务消费者：请参见 nacos-service-consumer

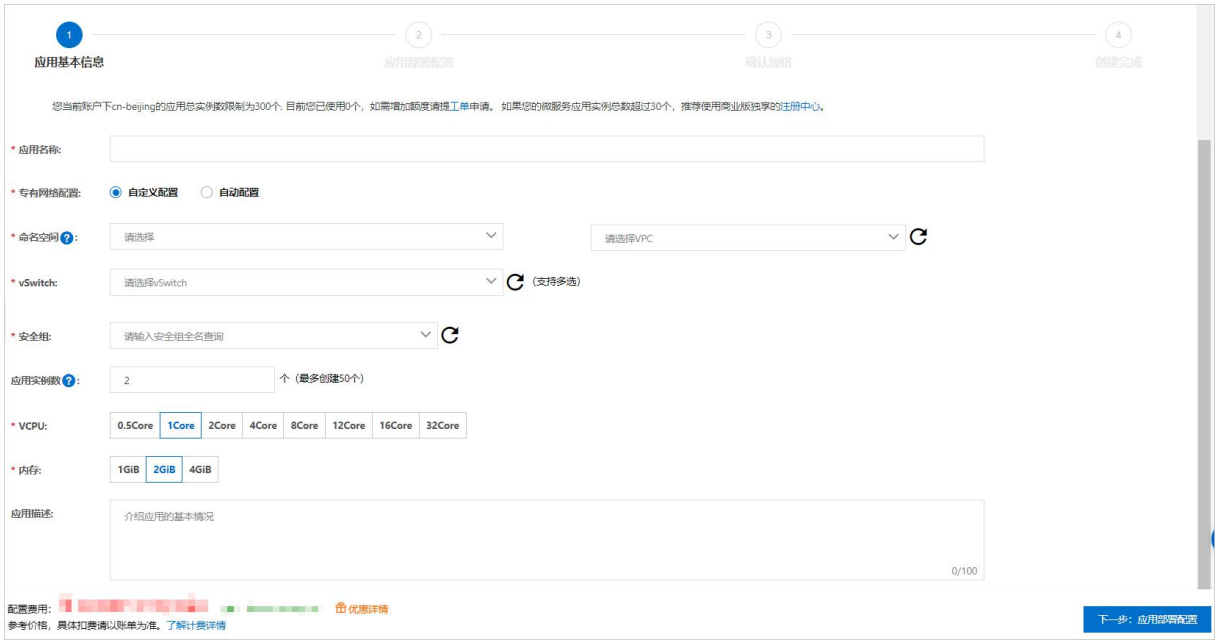
SAE 上部署应用

在本地完成应用的开发和测试后，便可将应用打包并部署到 SAE。具体步骤，请参见部署应用到 SAE

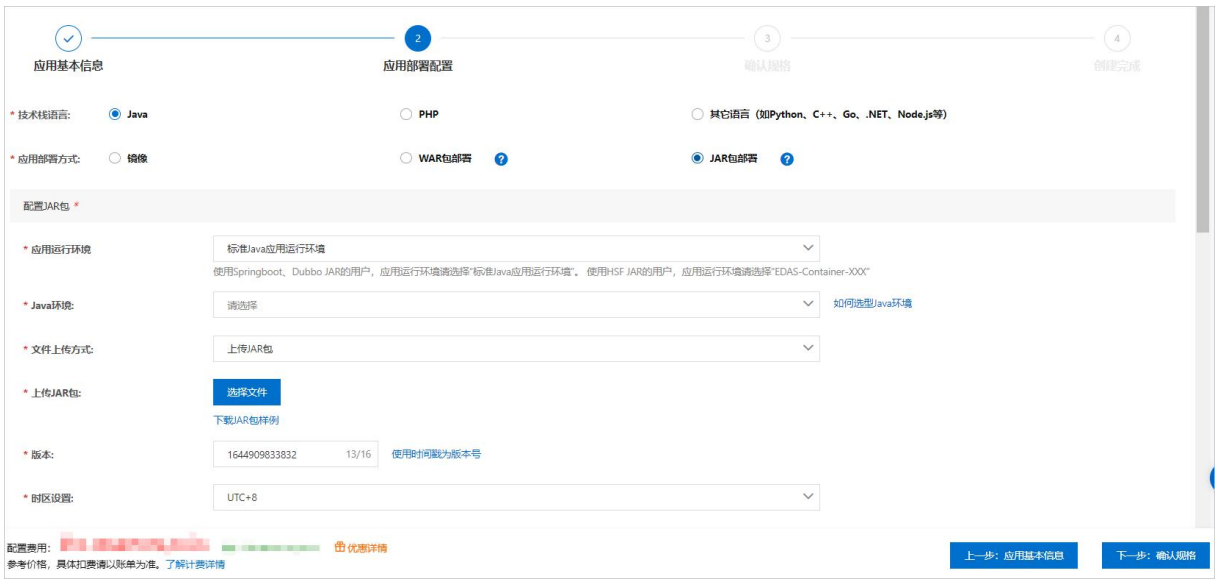
1.登录 SAE 控制台。

2.在左侧导航栏单击**应用列表**，在顶部菜单栏选择地域，单击**创建应用**。

3.在**应用基本信息**页签，设置应用相关信息，配置完成后单击**下一步：应用部署配置**。



4.在**应用部署配置**页签，配置相关参数。



验证部署是否成功

您可以通过提供者和消费者的服务列表页面判断部署是否成功。

1.登录 SAE 控制台。

2.在顶部菜单栏，选择地域。

3..在左侧导航栏单击**应用列表**，在顶部菜单栏选择地域，单击具体应用名称。

4.在左侧导航栏选择**微服务治理 > 服务列表**，查看服务信息。

- 提供者应用：如果**发布的服务**页签上可看到提供者所发布的服务，则说明提供者应用部署成功。

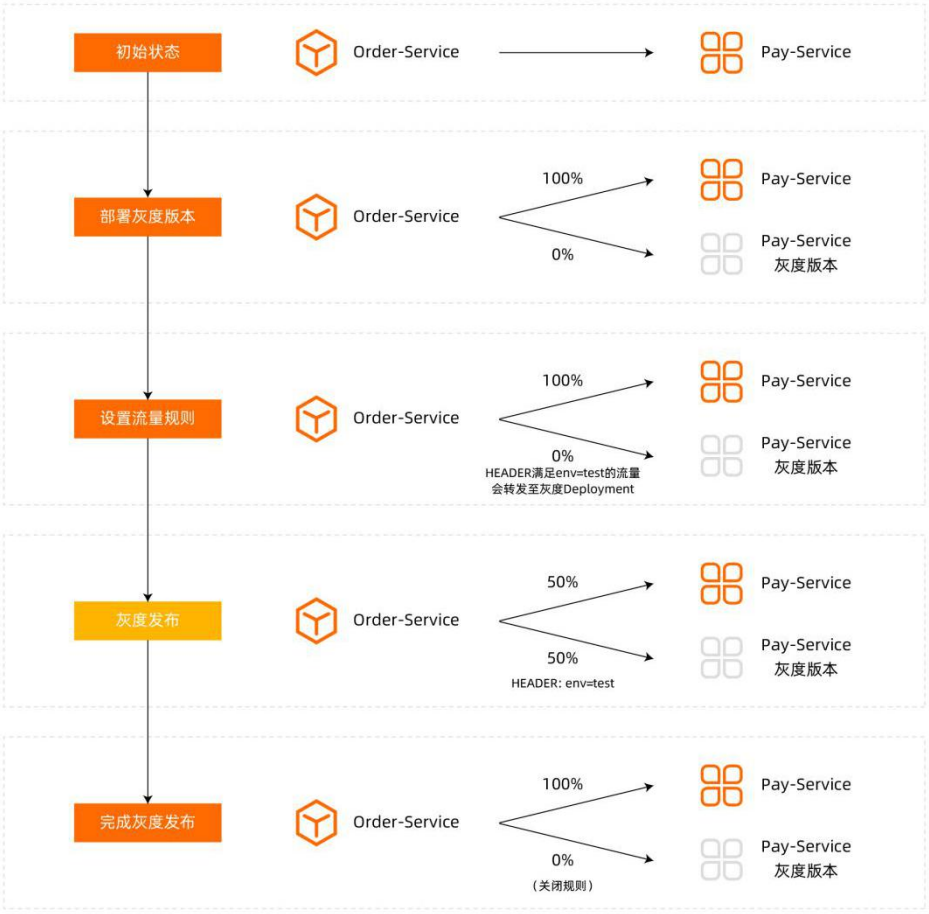


- 如果**消费的服务**页签上可看到提供者所发布的服务，则说明消费者应用部署成功。



动手实践 —— 灰度发布

灰度发布流程显意图：



步骤一：创建灰度规则

1.登录 SAE 控制台。

2.在左侧导航栏单击**应用列表**，在顶部菜单栏选择地域，单击具体应用名称。

3.在左侧导航栏，选择**微服务治理 > 灰度规则**。

4.在**灰度规则**页面，单击**新建灰度规则**。

5.在**新建灰度规则**面板，执行以下操作，然后单击**确定**。

6.按需配置以下参数。

参数	说明
规则名称	灰度规则的名称。支持以小写字母开头，以数字或小写字母结尾，仅可包含小写字母、中文字符、数字及短划线（-）。不超过 64 个字符。
规则描述	灰度规则的描述信息。不超过 64 个字符。
框架类型	配置灰度规则应用的框架类型，支持以下框架： **Spring Cloud**：需要设置 **Path**。 **Dubbo**：需要选择**服务方法**。
条件模式	选择**同时满足下列条件**或**满足下列任一条件**。

7.单击**添加新的规则条件**，按需配置灰度规则条件。

参数	说明
条件列表	**Spring Cloud**：根据 **Cookie**、**Header** 或 **Parameter** 参数类型，设置相应的**参数**、**条件**以及**值**。 **Dubbo**：根据应用实际情况，设置**参数**、**参数值获取表达式**、**条件**以及**值**。

8.**可选：**单击**新建流量规则**创建多个入口流量规则，多个规则可以同时生效。以 Spring Cloud 应用灰度规则示例



在本示例中，**参数类型**选择 Header。

创建成功后，您将在**灰度规则**页面查看到刚创建的灰度规则。同时，**新建灰度规则**按钮已置灰。

步骤二：查看规则详情

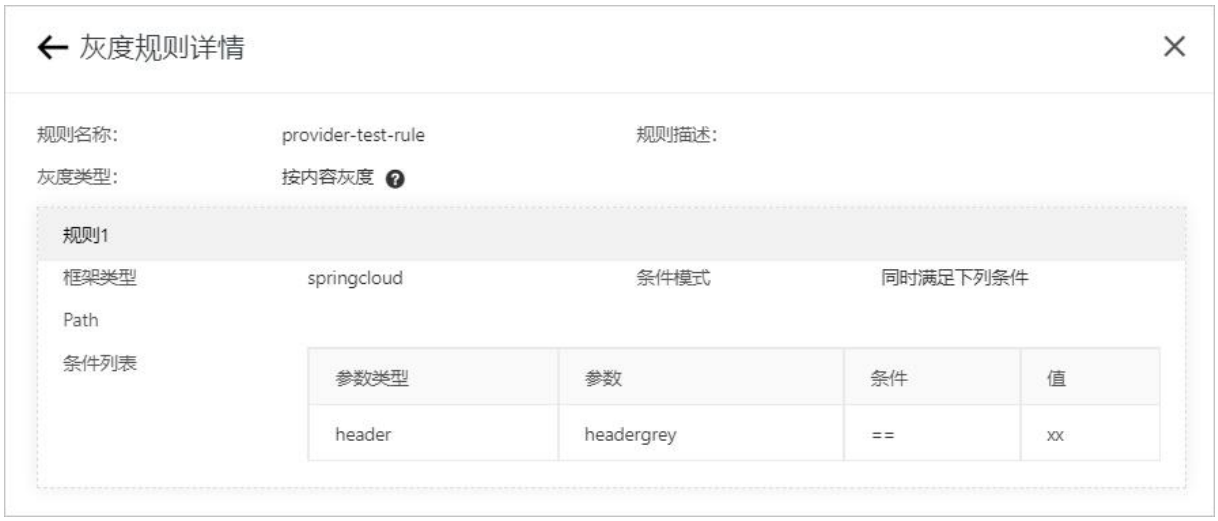
1.登录 SAE 控制台。

2.在左侧导航栏单击**应用列表**，在顶部菜单栏选择地域，单击具体应用名称。

3.在左侧导航栏，选择**微服务治理 > 灰度规则**。

4.在**灰度规则**页面，找到您需查看的灰度规则，单击规则名称。

5.在**灰度规则详情**面板，查看规则内容。



步骤三：配置灰度发布策略

1.登录 SAE 控制台。

2.在左侧导航栏单击**应用列表**，在顶部菜单栏选择地域，单击具体应用名称。

3.在**基本信息**页面的右上角，单击**部署应用**。

4.配置部署参数。

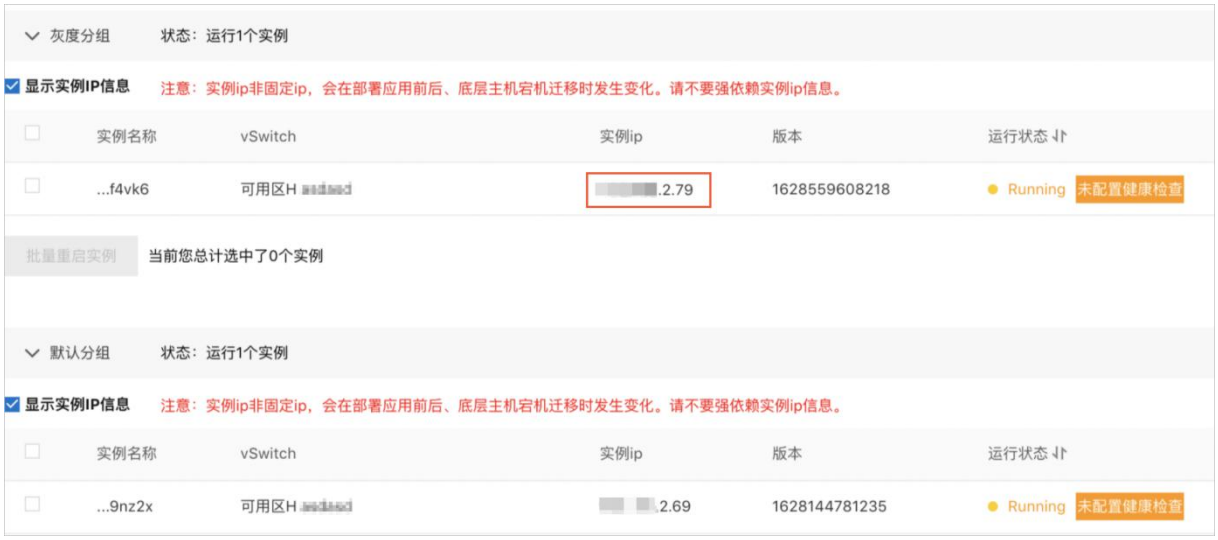
在**发布策略设置**区域内配置灰度发布参数。灰度发布参数说明如下：

配置	是否必选	说明
发布策略	是	选择**灰度发布**。
灰度数量	是	设置首先需要进行灰度发布的应用实例数量。
灰度后剩余批次	是	灰度发布后，剩余的应用实例按照设定的批次完成发布。
最小存活实例数	是	每次滚动升级最小存活的实例数。 **按个数**：输入最小存活实例数。您也可以选中**使用系统推荐值**，SAE 将根据您的需求为应用设置合理的最小存活实例数。 **按比例**：输入百分比。向上取整。例如设置为 50%，如果当前为 5 个实例，则最小存活实例数为 3。
启用微服务灰度规则	否	您为 Spring Cloud 或 Dubbo 应用创建的灰度规则。

说明：每次滚动部署最小存活的实例数建议≥1，保证业务不中断。如果设置为 0，应用在升级过程中将会中断业务。

步骤四：结果验证

在应用**基本信息**页面的**实例部署信息**页签查看实例的运行状态。如果**执行状态**显示为 Running，且实例的**版本**已变更，表示应用部署成功。



通过请求 Consumer 应用访问 Provider 应用，验证流量灰度发布到了指定实例。具体操作，请参见使用 Webshell 诊断应用。

```
sh-4.2# curl --location --request GET 'http://127.0.[redacted]/echo-rest/rest-res
t1?pa=10' \--header 'headergrey: xx' \
> --header 'Cookie: Cookie_2=value; cookie2=xx'
provider:[redacted].2.79:Tue Aug 10 09:47:04 CST 2021:79588a50-f6f4-4146-b521-68a49
5bc[redacted]:(rest-rest1,10,xx,Cookie_2=value; cookie2=xx,xx)sh-4.2# curl --location
t1?pa=10' \--header 'headergrey: xx' --header 'Cookie: Cookie_2=value; cookie2=x
x'
provider:[redacted].2.79:Tue Aug 10 09:47:07 CST 2021:79588a50-f6f4-4146-b521-68a49
5bc[redacted]:(rest-rest1,10,xx,Cookie_2=value; cookie2=xx,xx)sh-4.2# curl --location
t1?pa=10' \--header 'headergrey: xx' --header 'Cookie: Cookie_2=value; cookie2=x
x'
provider:[redacted].2.79:Tue Aug 10 09:47:08 CST 2021:79588a50-f6f4-4146-b521-68a49
5bc[redacted]:(rest-rest1,10,xx,Cookie_2=value; cookie2=xx,xx)sh-4.2# curl --location
t1?pa=10' \--header 'headergrey: xx' --header 'Cookie: Cookie_2=value; cookie2=x
x'
provider:[redacted].2.79:Tue Aug 10 09:47:10 CST 2021:79588a50-f6f4-4146-b521-68a49
5bc[redacted]:(rest-rest1,10,xx,Cookie_2=value; cookie2=xx,xx)sh-4.2#
```

第五章：微服务治理客户案例

5.1 物流行业：菜鸟 Cpaas 平台微服务治理实践

背景

CPaaS（cainiao platform as a service）是以公有云为基座，结合先进的云原生理建设的企业级 DevOps 的 PaaS 平台，CPaaS 主要目前主要支持的场景：菜鸟生态的云上研发运维、菜鸟公有云 SaaS 化的能力透出、菜鸟商业化输出支撑，部署到客户的公有云、专有云环境。

在服务了菜鸟多家生态公司及部分商业化输出的产品过程中，深入客户业务场景，解决业务研发及部署痛点的过程中，积累了一些宝贵的经验。这里我们主要对规范云上研发流程，提升研发效率为目标建设的环境治理（云上项目环境）及减少线上版本发布风而险建设的灰度平台的实现过程进行展开介绍。

目标

1、通过项目环境，为多分支并行开发场景提供流量隔离及快速联调的能力。
2、生产环境实现服务灰度发布（金丝雀发布），降低变更风险。
3、微服务应用具备优雅上下线能力，避免启停过程中带来的服务调用出错问题。

调研阶段

微服务流量管控

我们首先调研了开源自建的方案。在调研时我们发现，开发和维护开源 SDK 方案的成本非常大。需要对 Spring Cloud 和 Dubbo 这些微服务框架以及 RockeMQ 这类消息中间件非常了解，

才能准确地找到各个框架的增强点进行定制化开发。

除此之外，业务方使用的微服务框架版本也是跨度很大，维护这些不同版本的微服务框架适配，也需要投入大量的精力。

最重要的一点是，使用开源 SDK 自建的方案，开发业务的同事，需要在应用的开发和部署运维的流程都感知到 SDK 的存在，对开发、构建、运维的侵入性很大，很难进行推广。

后来我们也找到了阿里集团负责中间件的同事寻求支持，了解到中间件团队已经推出了一款面向公有云的微服务治理产品 MSE，于是我们进行了调研。

MSE 作为公有云的微服务治理产品，具备云上的**服务管控、微服务测试、标签路由、离群摘除、优雅上下线等**能力，与我们的诉求完全吻合，并且通过 agent方式实现，对应用代码无侵入，更适合 PaaS 平台对业务应用增加扩展。



图 1.1

经过与 MSE 团队同时的几次电话和会议沟通之后，逐渐对 MSE 产品有了一些功能上的认知。其中在微服务治理中，我们结合实际的业务需求，落地实现了下面MSE 的部分能力。



图 1.2

落地场景

项目环境

在多分支开发的场景下，我们通常需要同时部署多个分支。但是多个分支同时部署之后，如何将开发自测流量与日常环境的测试流量分开，以及如何让各个分支拥有自己独立的流量，都是需要解决的问题。

流量隔离

调研和验证 MSE 的标签路由的能力之后，实现思路：通过标签路由能力，将流量进行隔离。

相同应用的不同分支，使用不同的 deployment 实现对版本和容器标签的管理。图 2.1 中 core 应用项目环境 c1 和项目环境 c2 均使用独立于日常环境的容器单独部署，各自的路由标签为 joint1 和 joint2。通过 对流量携带流量标记的方式，完成项目环境流量的控制。

接入层应用则通过 K8S-Ingress 实现流量路由，只需要在请求流量的请求头中携带 x-mse-tag 的标签，则可以将流量路由到对应标签的入口层应用。入口应用设置标签传递的开发，可将流经此容器的流量打标传递到上有服务。如此反复，则实现流量在整个调用链路的封闭。

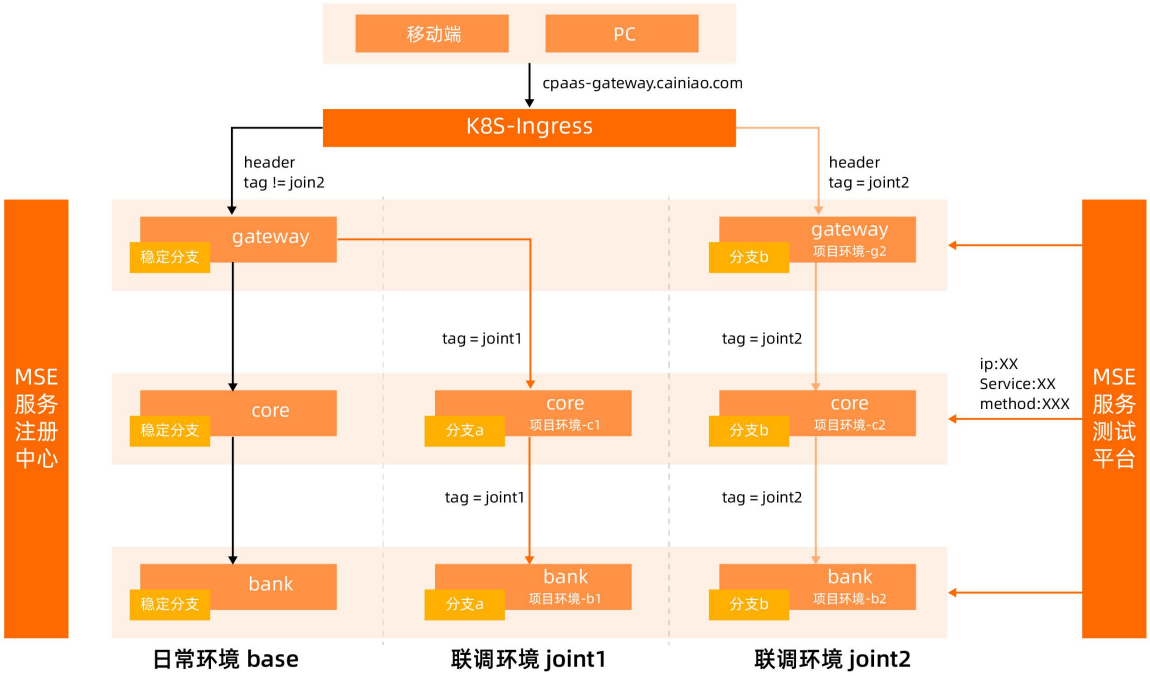


图 2.1

服务测试

在研发过程中，除了分支之间的相互干扰需要隔离之外，还需要从业务侧研发的角度解决服务测试的效率问题。MSE 平台提供了微服务测试平台，可以快速的帮助开发者实现服务自测，并且我们通过集成方式集成到自身的 paas 平台中，免去自身重建的痛苦。

测试平台支持按照服务提供者 IP 的维度进行服务测试，刚好与流量隔离相互契合，可以对自身关注的项目环境的应用容器发起服务测试完成服务自测。

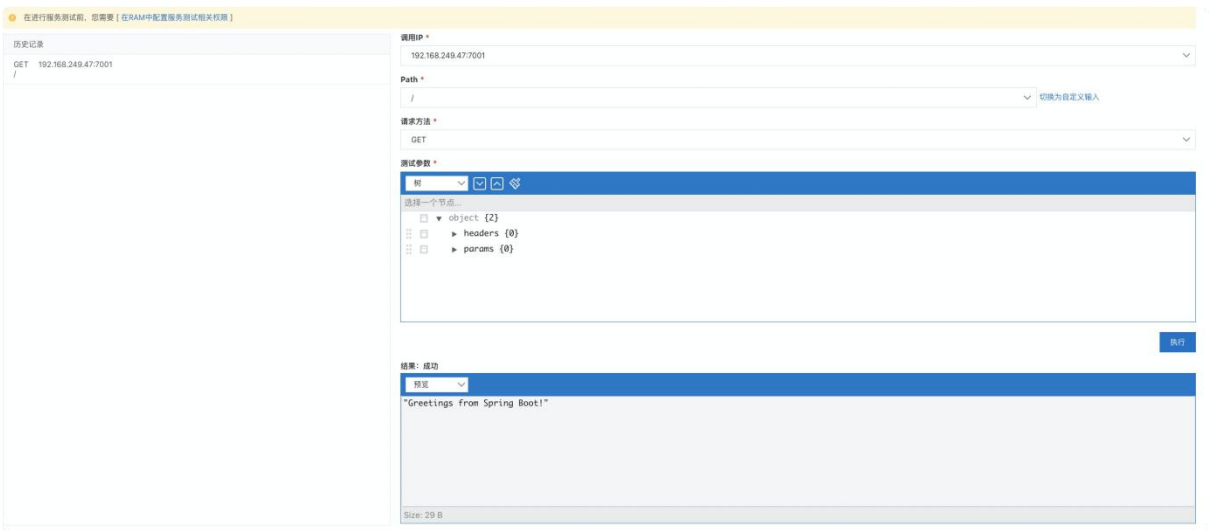
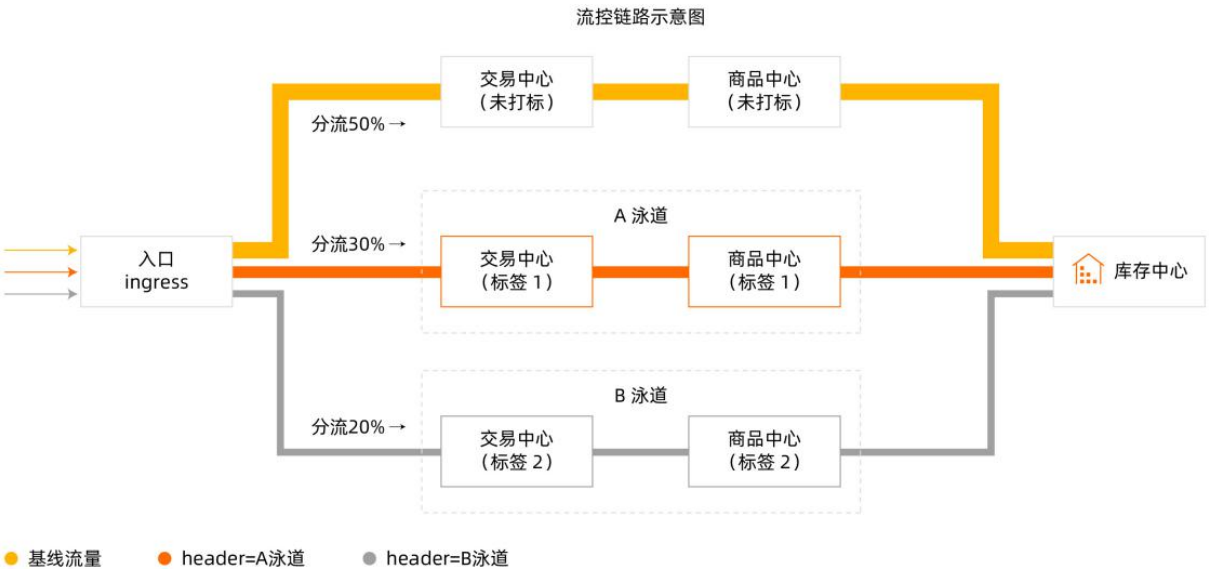


图 2.2

灰度环境

灰度环境的目的是为了降低线上应用版本发布的风险，减小问题爆炸半径。同样可以通过 MSE 提供的标签路由功能基础上完成对流量灰度的实现。



想要流量实现灰度，那么需要对流量的所有入口实现流量路由。通常我们所感知的流量入口包括：HTTP 接入层、RPC 下游调用、MQ 服务消费、Task 任务调度。当前在 MSE 的灰度能力支持范围内，微服务云网关、金丝雀发布、MQ 灰度均可以组合实现。

当然，这里我们只实现了 HTTP 接入层+RPC 的灰度能力，因为历史的原因，在接入层使用了另一个接入层（MSHA 的 MSFE，本质上是一个 tengine）实现接入层的灰度。但这里丝毫不影响与 MSE 的灰度能力完成流量的串联，这得益于 mse 产品自身拥有良好的对其他产品的兼容性。我们只需要在入口层的应用上设置流量经过此容器时带标，即可完成灰度流量的传递。

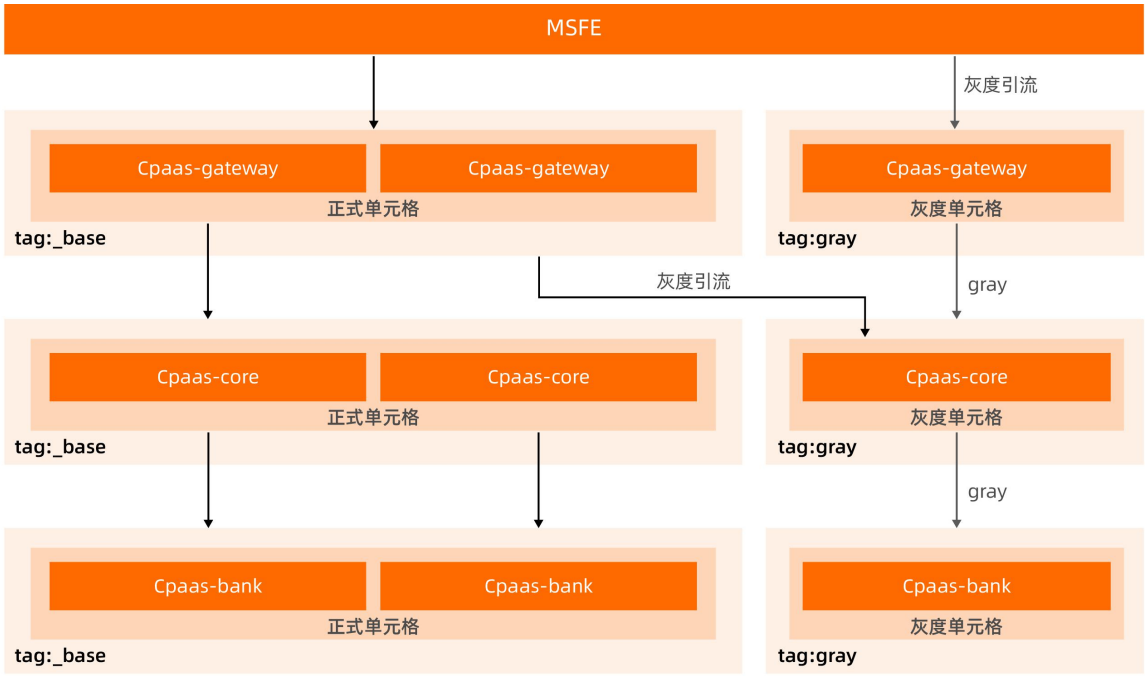


图 2.3

在实际的业务灰度场景中，我们总结了常见的四种灰度场景，均通过 MSE 完成并实现。



图 2.4

未来规划

在使用 mse 的云产品之后，对 paas 平台层来说，避免很多重复功能的建设。在我们业务侧实际落地的远不止如上列举的场景，比如：服务优雅停机、注册中心等能力，均解决了业务侧的微服务治理上的难点问题。

在实现了对项目环境及灰度发布的能力开发之后，我们接下来对服务离群摘除、应用服务列表透出、服务鉴权、本地联调部署等能力做重点关注，在降低业务侧服务运维成本、微服务可观测、服务可用性方面与 MSE 团队加强合作，帮助业务侧解决微服务治理中的痛点。

5.2 互联网行业：来电科技微服务治理实践

MSE 服务治理帮助我们系统以很低的成本无侵入的方式快速实现了全链路灰度能力，进一步提升了我们系统的稳定性，让我们新需求的迭代上线更加地安心。 -- 来电科技架构师 汤长征

来电科技自2014 年起开始进入共享充电领域，定义并开创了行业，属于行业内最早的共享充电企业。主要业务覆盖充电宝自助租赁、定制商场导航机开发、广告展示设备及广告传播等服务。来电科技拥有业内立体化产品线，大中小机柜以及桌面型，目前全国超过 90%的城市实现业务服务落地，注册用户超 2 亿人，实现全场景用户需求。

来电科技的业务场景丰富且系统众多，在技术架构上已完成容器化以及微服务化改造，微服务框架使用的是 Spring Cloud 与 Dubbo。随着近年来的高速发展，充电宝设备节点以及业务量都在快速增加。来电科技的整体应用架构也随着业务的高速发展，持续不断地进化。微服务治理是微服务化深入的必经之路，今天我来和大家分享一下来电科技在微服务化深入过程中探索的这一历程：

缘起：回顾来电科技当时的业务、架构现状和痛点。
初见：分享在技术选型之路上我们为什么选择 MSE 。
落地：我们是怎么一步步落地、在短时间内低成本落地全链路灰度能力以及无损上下线等能力。
展望：MSE 与来电科技携手进一步深化微服务化之路。

缘起

来电科技内部技术趋势满足如下三点：

- 微服务全面落地
- 全面接入K8s
- 快速迭代，稳定发布的诉求

来电科技在 2019 年 10月开始，服务开始全面进行微服务改造，容器化改造完成；在 20 年 12 月，此时来电科技已经全面微服务化，全面接入K8s。

可以看到随着来电微服务化进程的逐渐深入，在这个微服务深化的过程中，我们逐步会面临一系列的挑战，总的而言，我们讲这些挑战分为三个大的层面，他们分别是效率，稳定，和成本。我们进行微服务化，本身的使命是让业务的迭代更加高效，但当我们的微服务数量逐步增多，链路越来越长，如果不进行进一步的治理，那么引发的效率问题可能会大于微服务架构本身带来的架构红利。



因此在 2021 年 6月，来电科技对微服务进行了可观测建设；21 年 9月开始进行微服务治理能力构建。

全面云原生化的优势

云原生化总结来说有以下这些优势：

- 部署方便，发布效率大大提升
- 弹性扩缩容
- 大大节约服务器成本
- 运维成本降低

简单讲一下全面云原生化给来电科技系统带来的好处，首先就是应用部署变得非常方便，同时由于 K8s 的标准化使得 CI/CD 也变得简单，整体的发布效率大大提升；同时部署在 K8s 上的应用天然具备弹性扩缩容的能力，可以有效应对流量洪峰；同时由于上了 K8s 后，服务按需使用资源，相比原先按照峰值长期固定保有服务器，资源利用率相对比较低，现在可以大大节约服务器成本。相比传统集群运维非常繁琐，同时对运维人员技能要求也非常高：既要精通 lua / ansible 脚本等，又要懂云产品网络配置和监控运维，系统的运维成本非常高，阿里云 K8s 的标准化界面能很好解决高密部署以及系统运维的问题，极大降低成本。

稳定发布三板斧的诉求

日常发布中，我们常常会有如下一些错误的想法：

- 这次改动的内容比较小，而且上线要求比较急，就不需要测试直接发布上线好了。
- 发布不需要走灰度流程，快速发布上线即可。
- 灰度发布没有什么用，就是一个流程而已，发布完就直接发布线上，不用等待观察。
- 虽然灰度发布很重要，但是灰度环境很难搭建，耗时耗力优先级并不高。

这些想法都可能让我们进行一次错误的发布。

阿里巴巴内部有安全生产三板斧概念：可灰度、可观测、可回滚。所有研发同学必须要掌握发布系统的灰度、观测和回滚功能如何使用。

互联网频繁发布是常态，对于来电科技来说也是如此，系统具备灰度、观测、回滚的能力是微服务系统必须具备的能力，灰度可以说是发布之前的必备流程，也是提升线上稳定性的关键因素。当服务有新版本要发布上线时，通过引流一小部分流量到新版本，可以及时发现程序问题，有效阻止大面积故障的发生。业界上已经有比较成熟的服务发布策略，比如蓝绿发布、A/B 测试以及金丝雀发布，这些发布策略主要专注于如何对单个服务进行发布。

来电科技的微服务数目众多，服务之间的依赖关系错综复杂，如果采用多套环境的硬隔离，会使得成本大幅升高，发布方式变得复杂。有时某个功能发版依赖多个服务同时升级上线。希望可以对这些服务的新版本同时进行小流量灰度验证，通过构建从 Ingress网关到整个后端服务的环境隔离来对多个不同版本的服务进行灰度验证，这就是微服务治理中的全链路灰度的能力。

自研的挑战

来电科技有考虑过自研微服务治理，来电科技架构师 汤长征 同学也参与过 Dubbo 的开源社区，微服务治理研发对于来电科技来说并不是非常困难的事情，但是自研微服务治理组件还是存在以下必不可少的成本问题。

- 接入成本高
- 维护成本高
- 功能单一，不灵活，可扩展性低

- 可定位性变差

考虑到对生产应用进行微服务治理，微服务框架通常会引入服务治理的逻辑，而这些逻辑通常会以 SDK 的方式被业务代码所依赖，而这些逻辑的变更和升级，都需要每一个微服务业务通过修改代码的方式来实现，这样的变更造成了非常大的接入与升级成本。同时需要对开源的服务框架做治理功能的研发，就意味着需要出人力对微服务治理的组件进行管理与运维，同时自建会使得功能非常贴近业务，也意味着功能将会做得比较薄比较单一，未来的可扩展性就显得比较弱。同时全链路灰度的实现技术细节也是非常之多，动态路由，节点打标，流量打标，分布式链路追踪，技术的实现成本高。由于 Dubbo、Spring Cloud 等服务框架本身的复杂性，同时随着微服务数量逐步增多，链路越来越长，相关的微服务治理问题的定位与解决也成了让人头疼的问题，如果有 Spring Cloud Alibaba、Dubbo 等专业的团队支持，微服务化深入也会变得更加从容。

初见

第一次接触 MSE 服务治理这块产品，就有许多的点命中我们的诉求，以下几点对我们微服务治理改造来说都是很吸引的点。

- 无侵入

MSE 微服务治理能力基于 Java Agent 字节码增强的技术实现，无缝支持市面上近 5 年的所有 Spring Cloud 和 Dubbo 的版本，用户不用改一行代码就可以使用。只需开启 MSE 微服务治理专业版，在线配置，实时生效。

- 接入简单

只需在阿里云容器的应用市场安装 mse-pilot，然后在 mse 控制台开启命名空间级别的服务治理，重启应用即可接入，当然卸载服务治理也是非常容易的，只需在控制台关闭服务治理，卸载 mse-pilot 即可，不需要改变业务的现有架构，随时可上可下，没有绑定。

- 功能强大，持续发展

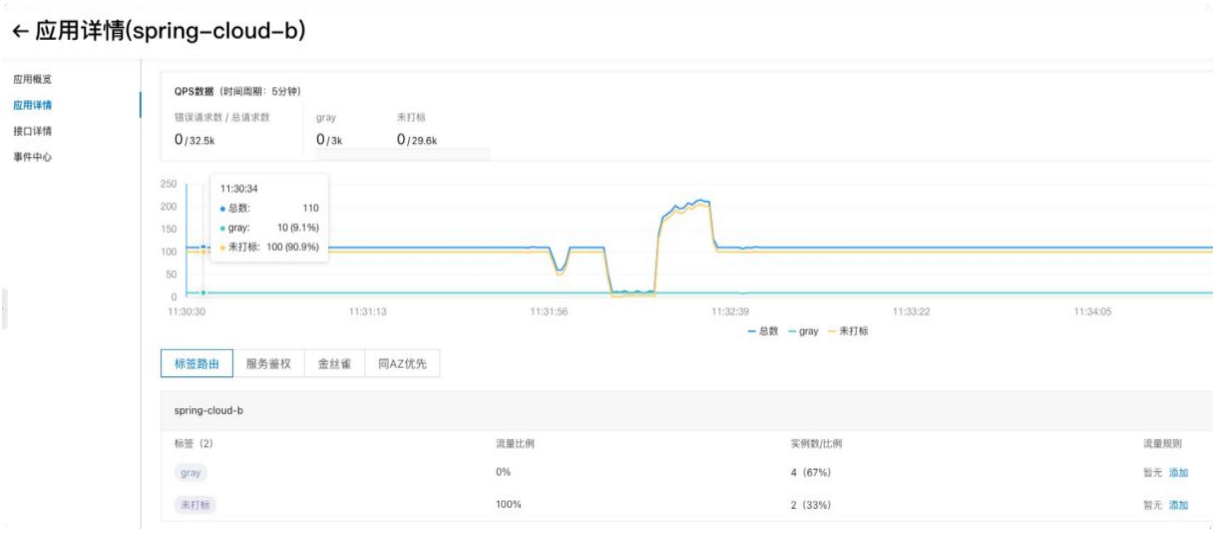
从开发态、测试态到运行态全生命周期的服务治理覆盖，使得研发同学可更加专注于业务本身。



MSE 微服务治理还提供了以下解决方案，解决微服务治理难点，快速提升企业的微服务治理能力。稳定性领域：线上故障紧急诊断排查与恢复、线上发布稳定性解决方案、微服务全链路灰度解决方案。降本增效领域：日常测试环境降本隔离解决方案、微服务无缝迁移上云解决方案、微服务开发测试提效解决方案。

- 可视化

MSE 服务治理专业版提供了微服务治理流量的可视化视图



对于灰度流量灰没灰到，灰了多少，配完路由规则后流量实时生效，做到一眼可见，心中有数。



同时对于无损上下线的场景，MSE 提供端到端全生命周期的防护，一眼可以看出流量有无损失，损失在什么部分。

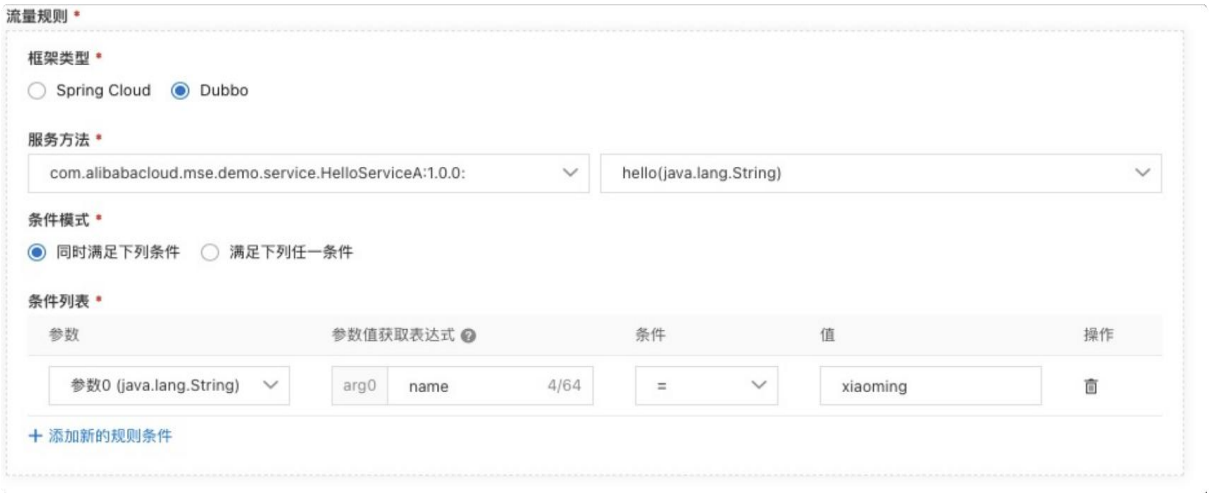
- 拥抱云原生

进入云原生体系之后，以 k8s 为主的云原生体系强调集群之间的灵活调度型，以 POD 为单位任意的调度资源，在被调度后 POD 的 IP 也将相应的发生变化，传统的服务治理体系，通常以 IP 为维度进行治理策略的配置，MSE 使用更加云原生的方式使用标签为维度进行微服务治理策略的配置。

同时在 K8s 环境下与 K8s 的体系深度集成，推出多种完整解决方案，无损上下线使得应用在弹性伸缩的过程中保持流量无损，通过 Jenkins 构建 CI/CD 实现在 K8s 环境下的金丝雀发布，基于 Ingress 实现全链路灰度等。

当时 MSE 的一些局限

当然在 21 年 9月刚接触 MSE 微服务治理的过程中发现 MSE 为服务治理的全链路能力还是存在一些局限性的，首先就是只支持微服务网关 Spring Cloud Gateway 与 Zuul 以及云网关，当时并不能支持自建的 Nginx网关，同时在 Dubbo 的场景下只支持按照接口参数维度的路由，对于运维同学来说还需要了解业务接口的实现，过于精细化控制，上生产的成本过高；同时全链路灰度的入口仅仅支持 Http网关或者应用作为灰度流量的入口，并不能支持 TCP网关作为流量的入口。

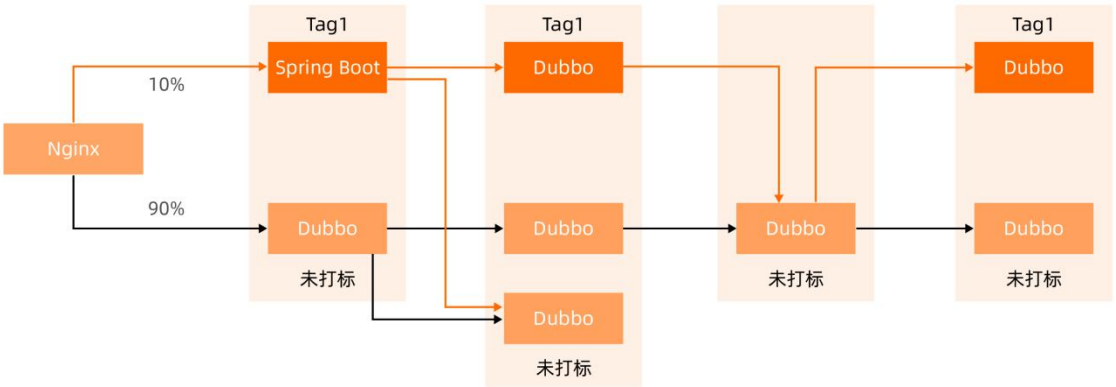


落地

我们与来电科技的架构师深入了解后，对用户的灰度场景进行了进一步的抽象与总结，只有深入到业务中去才能更加了解客户的需求。我们总结出如下三个场景：

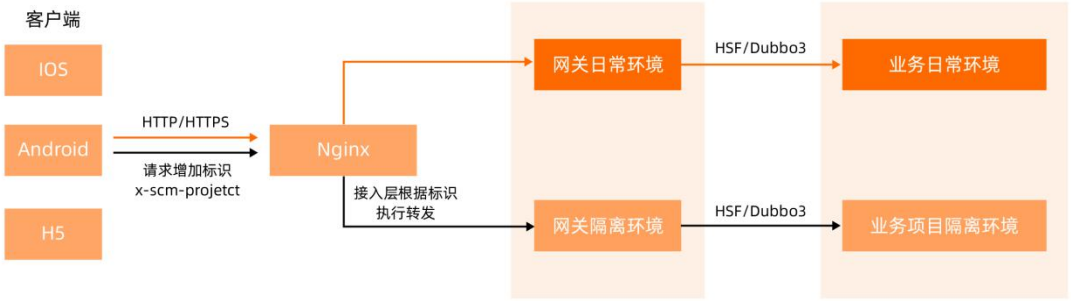
MSE 全链路灰度场景

场景一：对经过机器的流量进行自动染色，实现全链路灰度



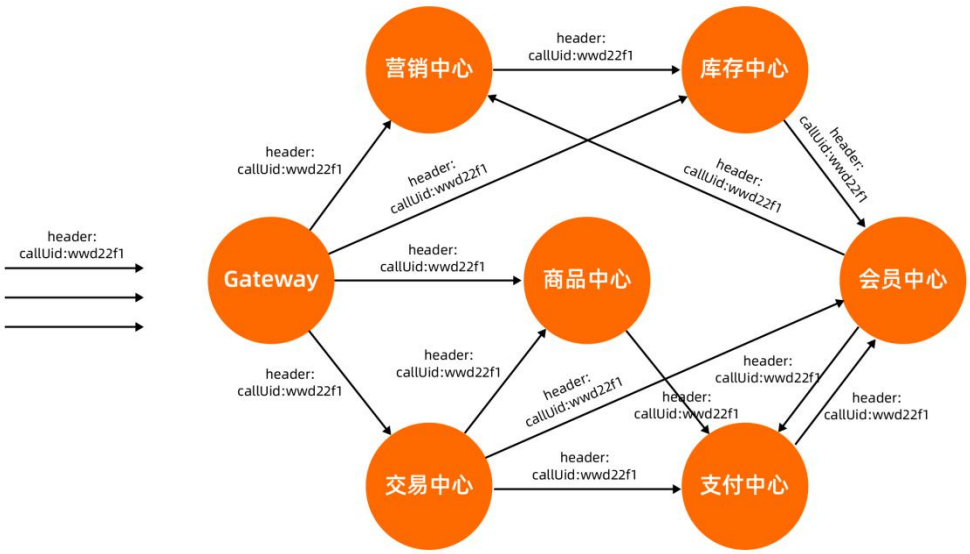
- 进入带 tag 的节点后续调用优先选择带有相同 tag 的节点，即对经过 tag 节点的流量进行"染色"。
- 有 tag 的调用链路上找不到相同 tag 的节点，则 fallback 到无tag 的节点。
- 有 tag 的调用链路经过无tag 的节点，如果链路后续调用有 tag 的节点，则恢复 tag 调用的模式。

场景二：通过给流量带上特定的 header 实现全链路灰度



客户端通过在请求中增加制定环境的标识，接入层根基表示进行转发至表示对应环境的网关，对应环境的网关通过隔离插件调用标识对应的项目隔离环境，请求在业务项目隔离环境中闭环。

场景三：通过自定义路由规则来进行全链路灰度



通过在灰度请求中增加指定的 header，且整条调用链路会将该 header 透传下去，只需在对应的应用配置该 header 相关的路由规则，带指定 header 的灰度请求进入灰度机器，即可按需实现全链路流量灰度。



我们考虑到场景一其实就能完美满足来电科技全链路灰度的场景，同时它也是绝大部分云上客户的诉求，场景二和三可以作为更加高阶的玩法。

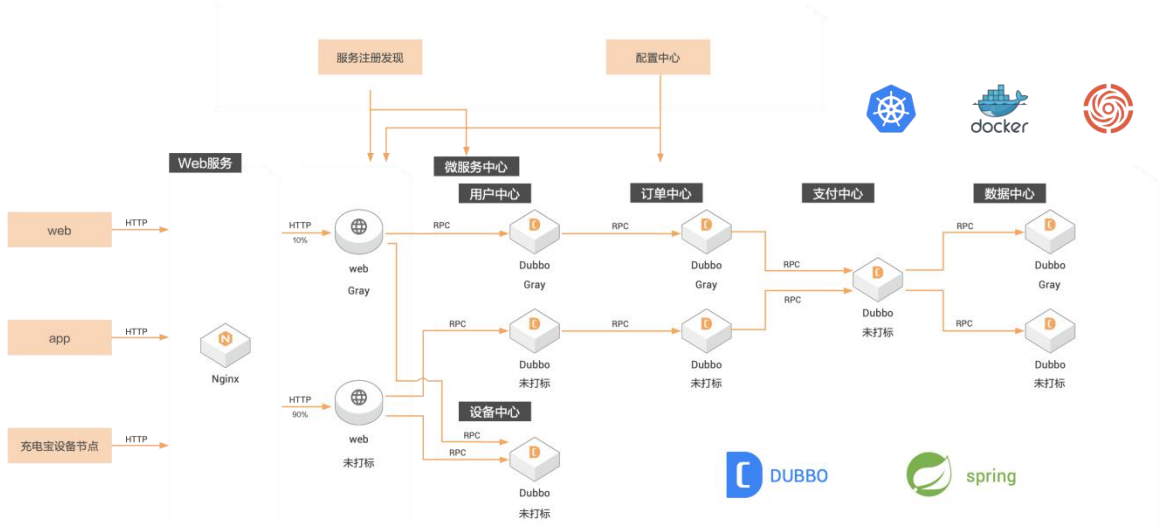
由于对经过应用的流量打标染色并进行全链路灰度，所以我们支持了任意的流量入口，也支持Ingress、自建网关的灰度，在支持应用级别的灰度的同时兼容自定义的路由，更加灵活的方式满足了来电科技全链路灰度的场景。

来电全链路灰度落地方案

来电的业务架构如下，最上层是移动端等用户界面，自建的 Nginx网关作为接入层，服务层就是各种服务，使用的是 Spring Cloud 与 Dubbo 作为服务框架。



来电科技全链路灰度落地的架构如下：



在 Nginx 层配置流量分流的配置，10%的流量进入灰度环境，90%的流量进入未打标即线上正式环境，然后经过灰度环境的流量会自动被 MSE 染上对应环境的颜色，从而进行全链路的灰度路由，保证流量在灰度环境中闭环，如果没有灰度环境的机器，比如支付中心只有线上的机器，那么流量会走线上环境，当我们数据中心又存在灰度环境的机器，那么灰度流量还会重新回到数据中心的灰度环境中。

MSE 服务预热能力

当我们在白天高峰期做发布，通常都会导致业务流量出现损失，我们的研发人员不得不选择在晚上业务低峰期做变更，这大大降低了研发人员的幸福指数，因为他们不得不面临熬夜加班的困境。如果能在白天大流量高峰期也能进行流量无损的变更，那么这对于研发人员来说将是大大提升研发效率的事情。

来电科技也遇到类似的问题，当业务流量过大的场景下，进行应用发布，系统服务刚启动阶段，应用由于存在冷启动的过程，此时的应用容量往往会比正常情况下低，但是线上的流量是无法区分当前的服务是否是刚启动的，依旧会有大流量持续涌入，此时就会导致系统过载而崩溃，出现流量损失。如果我们的微服务应用具备服务预热的能力，使得流量按照一定的曲线进行缓慢增长，从而保证服务进行充分的预热，即使是在高并发大流量场景中，保护应用在安全启动。

MSE 提供的一种基于 Agent 的无侵入预热微服务应用的方法能有效让用户在不修改任何代码的情况下即可为应用提供服务预热能力。

未来

MSE 服务治理专业版以无侵入的方式提供了全链路灰度、离群实例摘除、金丝雀发布、微服务治理流量可观测等核心能力，以更经济的方式、更高效的路径帮助来电科技在云上快速构建起完整微服务治理体系，有效提升线上稳定性，保证服务 99.9%的可用率。

随着来电科技微服务化的深入，除了全链路灰度、无损上下线还有更多的场景逐渐出现，微服务全生命周期的治理将覆盖从发布、运行、故障排查、故障恢复以及全链路流量的治理，MSE 微服务治理将携手帮助来电科技持续提升微服务研发效能与服务的高可用率。

5.3 机器智能行业：云小蜜 Dubbo3 微服务治理实践

前言

阿里云-达摩院-云小蜜对话机器人产品基于深度机器学习技术、自然语言理解技术和对话管理技术，为企业提供多引擎、多渠道、多模态的对话机器人服务。17 年云小蜜对话机器人在公共云开始公测，同期在混合云场景也不断拓展。为了同时保证公共云、混合云发版效率和稳定性，权衡再三我们采用了 1-2 个月一个大版本迭代。

经过几年发展，为了更好支撑业务发展，架构升级、重构总是一个绕不过去的坎，为了保证稳定性每次公共云发版研发同学都要做两件事：

1.梳理各个模块相较线上版本接口依赖变化情况，决定十几个应用的上线顺序、每批次发布比例；

2.模拟演练上述 1 产出的发布顺序，保证后端服务平滑升级，客户无感知；

上述 1、2 动作每次都需要 2-3 周左右的时间梳理、集中演练，但是也只能保证开放的 PaaS API 平滑更新；控制台服务因为需要前端、API、后端保持版本一致才能做到体验无损（如果每次迭代统一升级 API 版本开发、协同成本又会非常大），权衡之下之前都是流量低谷期上线，尽量缩短发布时间，避免部分控制台模块偶发报错带来业务问题。

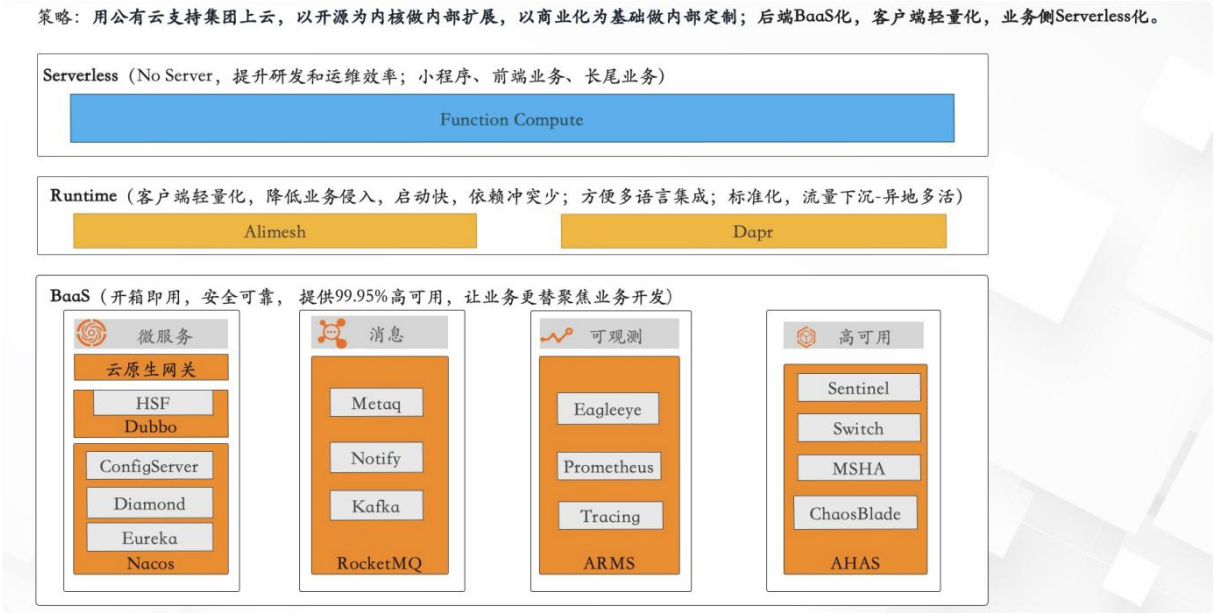
针对上面问题，很早之前就考虑过用蓝绿发布、灰度等手段解决，但是无奈之前对话机器人在阿里云内部业务区域，该不再允许普通云产品扩容，没有冗余的机器，流量治理完全没法做。

迁移阿里云云上

带着上面的问题，终于迎来的 2021 年 9月份，云小蜜将业务迁移至阿里云云上。

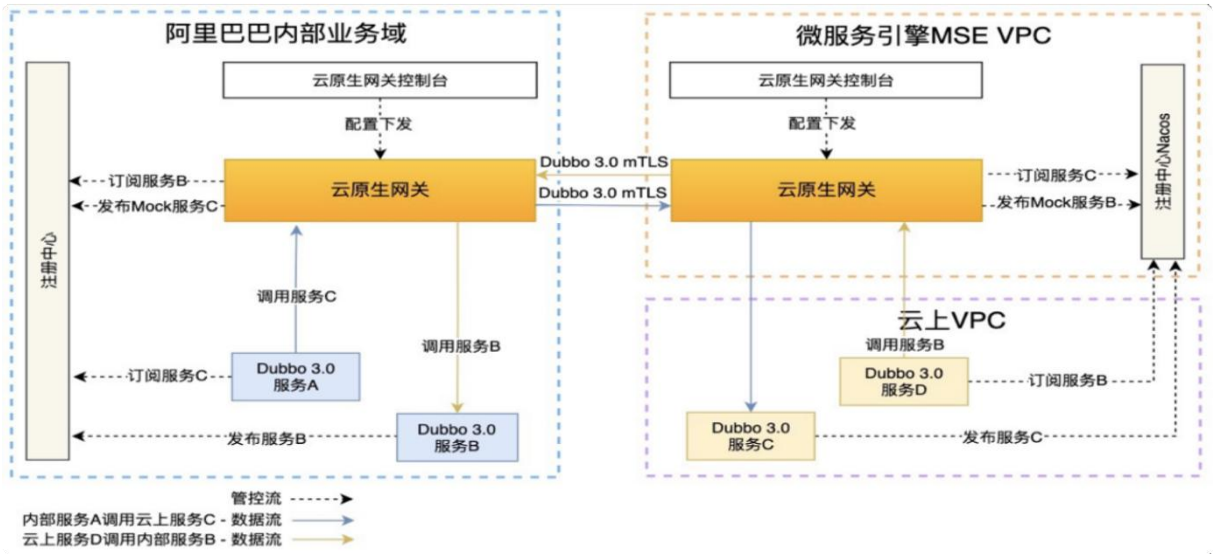
Dubbo3.0 的实践

“当时印象最深的就是这张图，虽然当时不知道中间件团队具体要做什么事情，但是记住了两个关键词：三位一体、红利。没想到在 2021 年底，真真切切享受到了这个红利。”



云小蜜使用的是集团内部的 HSF 服务框架，需要迁移至阿里云云上，并且存在阿里云云上与阿里内部业务域的互通、互相治理的诉求。云小蜜的公共服务部署在公有云 VPC，部分依赖的数据服务部署在内部，内部与云上服务存在 RPC 互调的诉求，其实属于混合云的典型场景。

简单整理了下他们的核心诉求，概括起来有以下三点吧：希望尽可能采用开源方案，方便后续业务推广；在网络通信层面需要保障安全性；对于业务升级改造来说需要做到低成本。



在此场景下，经过许多讨论与探索，方案也敲定了下来：

- 全链路升级至开源 Dubbo3.0，云原生网关默认支持 Dubbo3.0，实现透明转发，网关转发 RT小于 1ms。
- 利用 Dubbo3.0 支持 HTTP2 特性，云原生网关之间采用 mTLS 保障安全。
- 利用云原生网关默认支持多种注册中心的能力，实现跨域服务发现对用户透明，业务侧无需做任何额外改动。
- 业务侧升级 SDK 到支持 Dubbo3.0，配置发布 Triple 服务即可，无需额外改动。

解决了互通、服务注册发现的问题之后，就是开始看如何进行服务治理方案了。

阿里云云上流量治理

迁移至阿里云云上之后，流量控制方案有非常多，比如集团内部的全链路方案、集团内单元化方案等等。

设计目标和原则

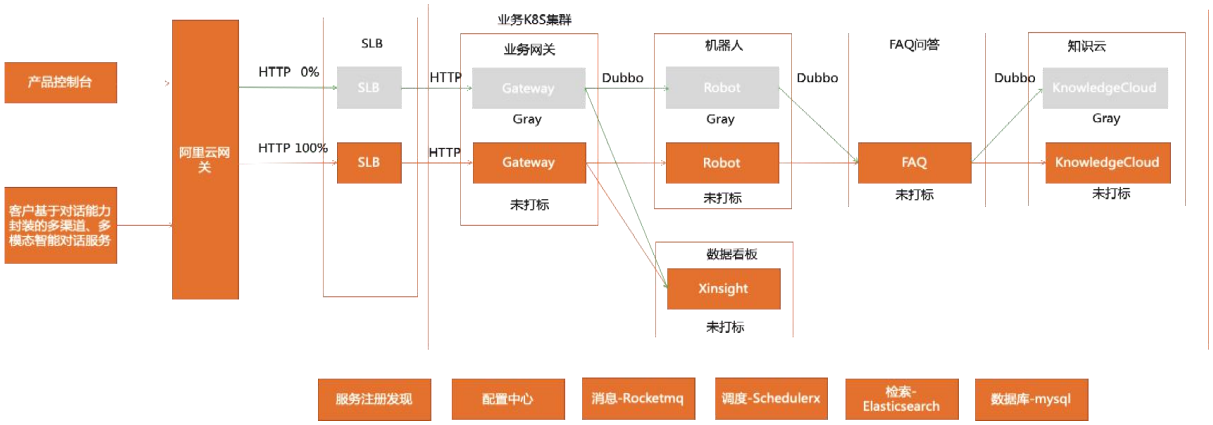
1.要引入一套流量隔离方案，上线过程中，新、旧两个版本服务同时存在时，流量能保证在同一个版本的“集群”里流转，这样就能解决重构带来的内部接口不兼容问题。
2.要解决上线过程中控制台的平滑性问题，避免前端、后端、API 更新不一致带来的问题。
3.无上线需求的应用，可以不参与上线。
4.资源消耗要尽量少，毕竟做产品最终还是要考虑成本和利润。

方案选型

1.集团内部的全链路方案：目前不支持阿里云云上。
2.集团内单元化方案：主要解决业务规模、容灾等问题，和我们碰到的问题不一样。
3.搭建独立集群，版本迭代时切集群：成本太大。
4.自建：在同一个集群隔离新、老服务，保证同一个用户的流量只在同版本服务内流转以 RPC 为例：
方案一：通过开发保证，当接口不兼容升级时，强制要求升级 HSF version，并行提供两个版本的服务；缺点是一个服务变更，关联使用方都要变更，协同成本特别大，并且为了保持平滑，新老接口要同时提供服务，维护成本也比较高。
方案二：给服务（机器）按版本打标，通过 RPC 框架的路由规则，保证流量优先在同版本内流转。

全链路灰度方案

就当 1、2、3、4 都觉得不完美，一边调研一边准备自建方案 5 的时候，兜兜绕绕拿到了阿里云 MSE 微服务治理团队《20 分钟获得同款企业级全链路灰度能力》，方案中思路和准备自建的思路完全一致，也是利用了 RPC 框架的路由策略实现的流量治理，并且实现了产品化（微服务引擎-微服务治理中心），同时，聊了两次后得到几个“支持”，以及几个“后续可以支持”后，好像很多事情变得简单了...



从上图可以看到，各个应用均需要搭建基线(base)环境和灰度(gray)环境，除了流量入口-业务网关以外，下游各个业务模块按需部署灰度（gray）环境，如果某次上线某模块没有变更则无需部署。

各个中间件的治理方案

1.Mysql、ElasticSearch：持久化或半持久化数据，由业务模块自身保证数据结构兼容升级；
2.Redis：由于对话产品会有多轮问答场景，问答上下文是在 Redis里，如果不兼容则上线会导致会话过程中的 C 端用户受影响，因此目前 Redis 由业务模块自身保证数据结构兼容升级；
3.配置中心：基线(base)环境、灰度(gray)环境维护两套全量配置会带来较大工作量，为了避免人工保证数据一致性成本，基线(base)环境监听 dataId，灰度(gray)环境监听 gray.dataId 如果未配置 gray.dataId 则自动监听 dataId；（云小蜜因为在 18 年做混合云项目为了保证混合云、公共云使用一套业务代码，建立了中间件适配层，本能力是在适配层实现）
4.RPC 服务：使用阿里云 one agent 基于 Java Agent 技术利用Dubbo 框架的路由规则实现，无需修改业务代码；应用只需要加一点配置：
1）linux 环境变量
alicloud.service.tag=gray 标识灰度，基线无需打标

profiler.micro.service.tag.trace.enable=true 标识经过该机器的流量，如果没有标签则自动染上和机器相同的标签，并向后透传
2）JVM 参数，标识开启 MSE 微服务流量治理能力SERVICE_OPTS="${SERVICE_OPTS} -Dmse.enable=true"

流量管理方案

流量的分发模块决定流量治理的粒度和管理的灵活程度。

对话机器人产品需要灰度发布、蓝绿发布目前分别用下面两种方案实现：

1.灰度发布：
部分应用单独更新，使用POP 的灰度引流机制，该机制支持按百分比、UID 的策略引流到灰度环境。

2.蓝绿发布：
1）部署灰度(gray)集群并测试：测试账号流量在灰度(gray)集群，其他账号流量在基线(base)集群。
2）线上版本更新：所有账号流量在灰度(gray)集群。
3）部署基线(base)集群并测试：测试账号流量在基线(base)集群，其他账号流量在灰度(gray)集群。
4）流量回切到基线(base)集群并缩容灰度(gray)环境：所有账号流量在基线(base)集群。

全链路落地效果

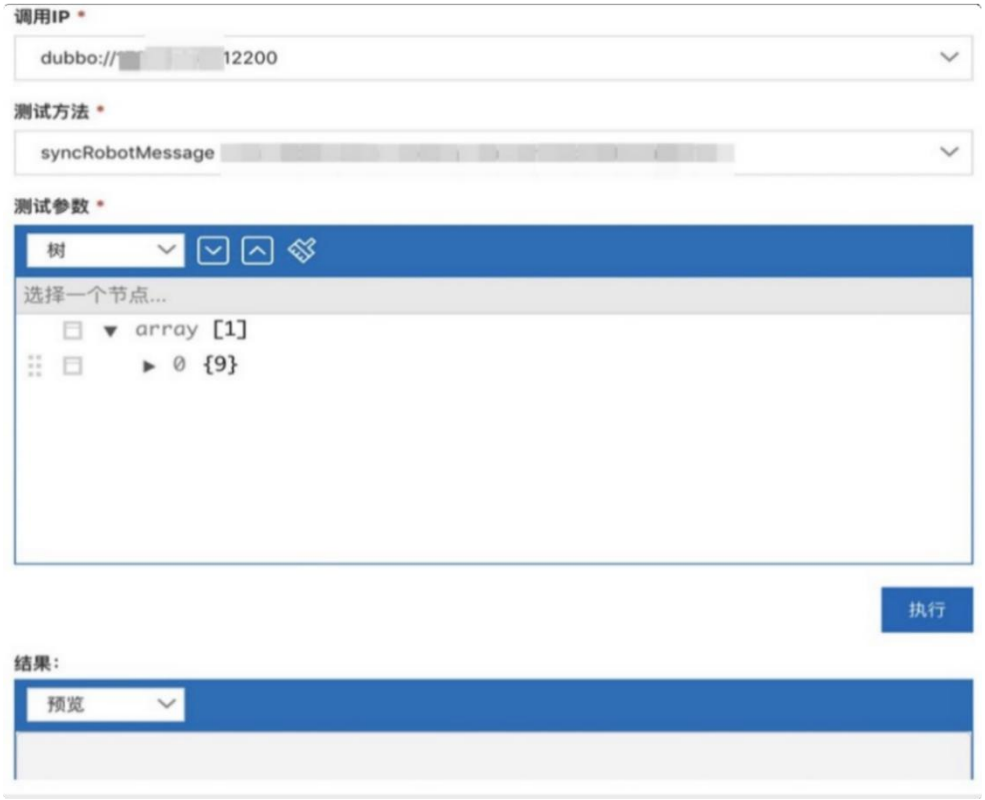
上线后第一次发布的效果：“目前各个模块新版本的代码已经完成上线，含发布、功能回归一共大约花费 2.5小时，相较之前每次上线到凌晨有很大提升。”

MSE 微服务治理全链路灰度方案满足了云小蜜业务在高速发展情况下快速迭代和小心验证的诉求，通过 JavaAgent 技术帮助云小蜜快速落地了企业级全链路灰度能力。

流量治理随着业务发展会有更多的需求，下一步，我们也会不断和微服务治理产品团队，扩充此解决方案的能力和使用场景，比如：rocketmq、schedulerx 的灰度治理能力。

更多的微服务治理能力

使用 MSE 服务治理后，发现还有更多开箱即用的治理能力，能够大大提升研发的效率。包含服务查询、服务契约、服务测试等等。这里要特别提一下就是云上的服务测试，服务测试即为用户提供一个云上私网 Postman ，让我们这边能够轻松调用自己的服务。我们可以忽略感知云上复杂的网络拓扑结构，无需关系服务的协议，无需自建测试工具，只需要通过控制台即可实现服务调用。支持 Dubbo 3.0 框架，以及 Dubbo 3.0 主流的 Triple 协议。



结束语

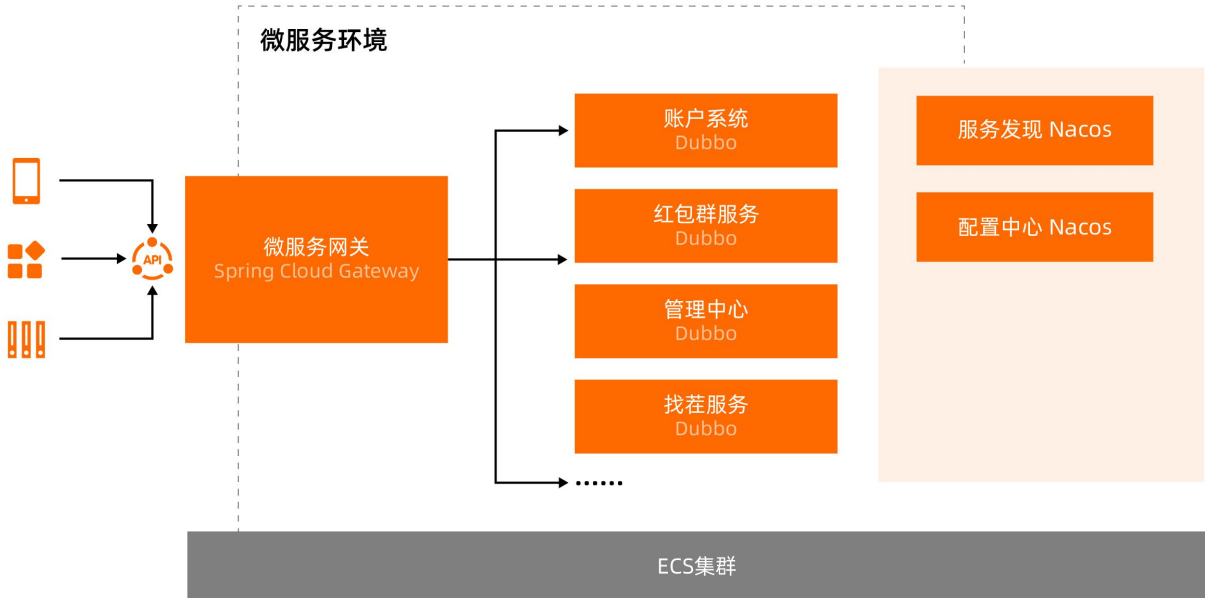
最终云小蜜对话机器人团队成功落地了全链路灰度功能，解决了困扰团队许久的发布效率问题。在这个过程中我们做了将部分业务迁移至阿里云云上、服务框架升级至Dubbo3.0、选择 MSE 微服务治理能力等等一次次新的选择与尝试。“世上本没有路，走的人多了便成了路”。经过我们工程师一次又一次的探索与实践，能够为更多的同学沉淀出一个个最佳实践。我相信这些最佳实践将会如大海中璀璨的明珠般，经过生产实践与时间的打磨将会变得更加熠熠生辉。

5.4 游戏行业：广州小迈微服务治理实践

背景

广州小迈于 2015 年 1 月成立，是一家致力以数字化领先为优势，实现业务高质量自增长的移动互联网科技公司。始终坚持以用户价值为中心，以数据为驱动，为用户开发丰富的工具应用、休闲游戏、益智、运动等系列的移动应用。累计开发 400 余款产品，累计用户下载安装量破七亿。

众所周知，游戏行业的主要特点：种类繁多、推广昂贵、依赖爆款、高性能&更新快、极致稳定、弹缩明显、架构简洁（比较流行烟囱式的部署）。基于这些特性，游戏企业需要一套高可用、易扩展、具备自动弹性的架构来支撑业务。我们小迈的业务架构也是随着业务发展不断演进，从单体到微服务架构，也计划往容器化方向演进。



但整个落地过程中最大的问题是：

1.因游戏推广涉及到买量，经常出现容量评估不准，资源利用率低。

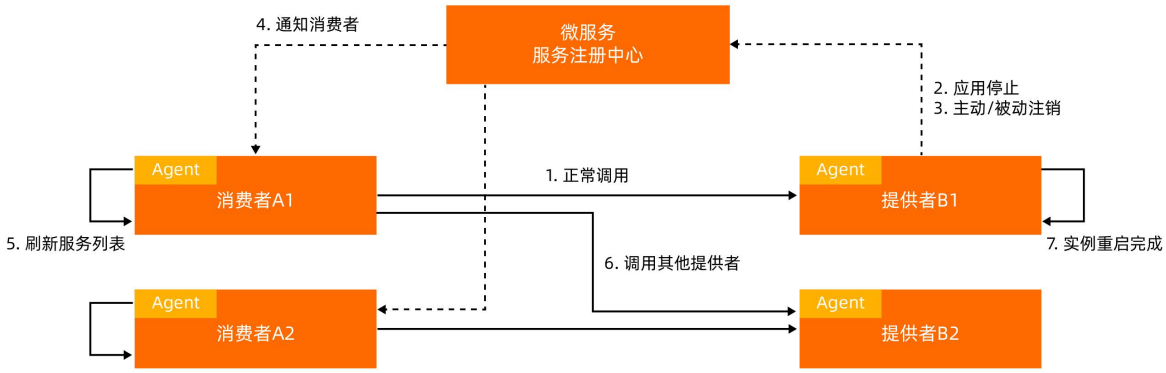
2.传统服务器部署架构，不能快速自动弹缩。

3.缺少微服务治理，发布不稳定：无服务监控，尤其应用发版/扩缩容时流量有损，对终端游戏客户是不能接受的。发布之后，瞬间一批流量被打到新实例，导致部分请求超时/报错。

问题分析

问题 1 和问题 2 均属于容量规划和弹性的范畴，原因均和业务侧的流量有关。

问题 3 服务治理就稍微麻烦一点了，我们仔细分析了应用发版时流量有损的原因，本质上还是服务无法及时下线导致的。



图中有服务消费者 A1-A2 和提供者 B1-B2。B1-B2 注册到注册中心，A1-A2 从注册中心刷新服务列表。提供者 B 发新版本时，先对 B1 操作，首先停止 Java 进程，服务停止过程又分为主动销毁和被动销毁，主动销毁是准实时的，被动销毁的时间由不同的注册中心决定，最差的情况可能需要一分钟。

如果应用正常停止，Dubbo 框架的 ShutdownHook 能正常被执行，耗时基本上可忽略不计。如果应用非正常停止，比如直接 Kill-9 或者是 Docker 镜像构建时，Java 进程不是 1 号进程，且没有把 Kill 信号传递给应用的话，服务提供者是不会主动去注销节点，它会等待注册中心去发现、被动地去感知服务下线的过程。从 B1 提供者下线到 Nacos 注册中心感知并通知消费者，整个耗时最差需要 50 秒。这段时间内消费端所有请求都可能会出现问题，所以发布时会出现各种报错。



Eureka 在最差的情况下需要耗时2分钟，Nacos 在最差的情况下需要耗时50秒

方案选型

针对这些问题，我们当时内部讨论了几种方案，但都存在一些弊端：

方案一：开源微服务定制+自建 K8s

在开源微服务基础上做一些定制，解决发布时流量有损和新实例流量分配等服务治理问题。比如消费端通过轮询注册中心的提供者列表来尽量避免流量有损，虽然有一定效果，但无法保证极极短有损，其他的一些高级治理能力也需挨个定制开发，比较耗费开发人力且存在稳定性风险。容量和弹性的问题考虑过自建 K8s，但学习曲线太陡峭，短期内无法组建专业团队。

方案二：开源微服务定制+ESS 弹性

和方案一的差异是通过 ESS 来做容量规划和自动弹性，虽然能缓解弹性备容问题，但弹性效率还是有点慢，且资源利用率的问题还是没有得到解决。

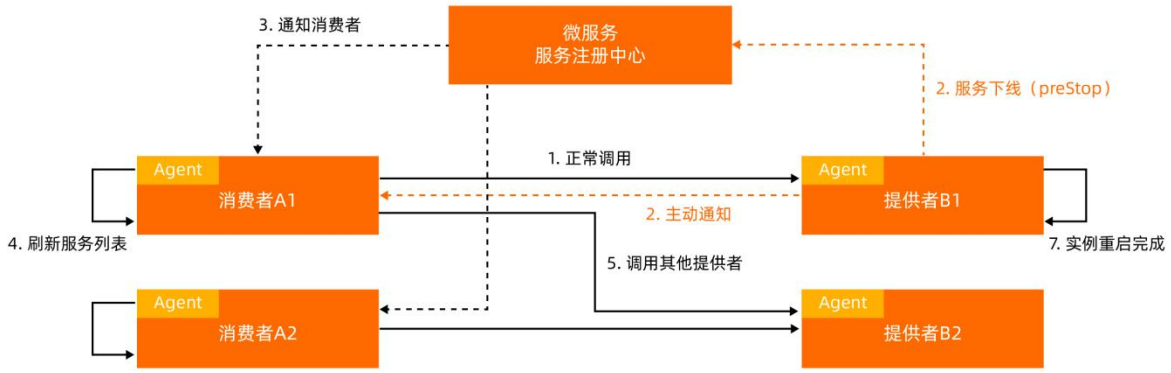
后来，经过阿里云同事介绍，**很快又有了方案三** —— 使用 Serverless 应用引擎（简称 SAE）+ MSE，也是最终落地的方案。我们主要看中的是：该方案提供了一整套开箱即用的微服务治理能力，比开源更丰富稳定。接入简单，war/jar 直接部署，不需要学习容器知识。极致的弹性能力，想弹多少就弹多少，想什么时候弹就什么时候弹，无需人工介入。

落地实施

整个 SAE 的落地过程我们是先在新业务中试点，并且采用 ECS+SAE 混部逐渐切流的方式渐进式推进的。在解决业务侧核心痛点的同时，我们也完善了一些新能力。

1.解决痛点问题 —— 应用发版时服务无法及时下线

对我们用户侧来说，只需要在 SAE 控制台启用无损下线功能，配置微服务应用 Prestop 脚本，即可做到白天也能无损发布，详细操作参考官方文档。我们也找阿里云同学了解到了实现原理：在应用创建部署时内置了 MSE agent，绕过了注册中心，由服务提供者直接通知消费端最新的服务列表的。



SAE 做了两件事情，第一，服务提供者在应用发布前，会主动向服务注册中心注销应用，并将应用标记为已下线状态，将原来停止进程阶段的注销变成了 preStop 阶段注销进程。在接收到服务消费者的请求时，首先会正常处理本次请求，并且通知服务消费者此节点已经下线，在此之后消费者收到通知后，会立即刷新自己的服务列表，在此之后服务消费者就不会再把请求发到服务提供者 B1 的实例上。

基于该方案，下线感知时间大大缩短，从原来的分钟级别做到准实时的，确保应用发布时流量无损，我们的游戏客户再也没有投诉。比开源定制方案更实时、更稳定（经过双 11 考验）、无需投入研发。

2.解决痛点问题 —— 新实例上线流量有损

一般情况下，新实例业务就绪需要一个过程，如果还没就绪就把它加到注册中心让消费端去访问，或者新实例一启动就承载过大流量，极易出现大量请求响应慢，资源阻塞，应用实例宕机的现象。SAE 集成了 MSE 的无损上线能力，提供了自定义延迟到注册中心、小流量服务预热，微服务 readniess 探针检查的功能，很好的解决了新实例流量有损的问题。虽然 Dubbo 原生也有类似的解决思路，但需要开发者自研，SAE + MSE 提供的是产品化的方案、简单易用的同时还提供了可观测能力。

3.完善微服务治理周边配套

在迁移到 SAE 的过程中，偶尔也会碰到一些应用状态异常的问题。我们之前没有应用监控的能力，遇到此类问题非常痛苦，也计划投入人力自建 APM。但发现 SAE 天然就集成了 Arms 应用监控，能够看到应用的调用关系拓扑图，可以定位到慢 SQL、慢服务、方法的调用堆栈、进而定位到代码级别的问题。也能查看各种维度的 TopN 关注应用，实现 1 人轻松运维成千上百个应用，十分方便，极大的提升了我们的排查效率。

后续我们还会继续评估 MSE 的微服务网关产品，如果合适也会考虑替换我们的开源网关。

4.SAE 极致弹性

SAE 提供了丰富的弹性指标和策略，我们主要用的是 cpu，mem、QPS、RT 指标来自动弹性，发现 SAE 真的能在峰值时秒级自动扩容，峰谷时按需自动缩容，使用 SAE 之后比以往 ECS 长期保有方式节省了 40% 左右的硬件成本。

落地效果

通过和 SAE 平台不断的磨合验证，我们游戏团队的应用都已经全部迁移到 SAE。整个迁移过程平滑，无任何改造成本，零故障，并且只投入了 1 个研发人员。



后续，我们也会在公司层面加大宣传 SAE+MSE 提供的微服务 on Serverless 的解决方案，让整个公司都能低门槛充分享受云原生带来的技术红利。

第六章：总结与展望

总结与展望

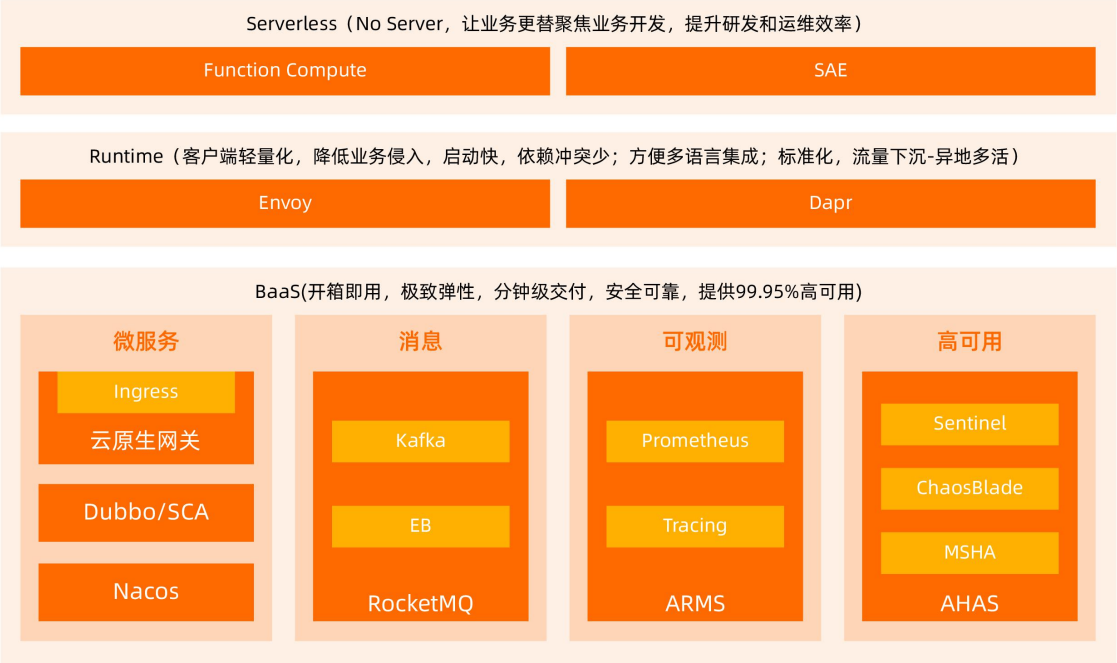
经过十多年的发展微服务，微服务架构已经进入了大众采纳阶段，已经基本满足了企业业务敏捷开发的诉求，随着微服务实践的深入，微服务治理必将成为企业下一阶段微服务架构演进的重要阶段。本白皮书从技术发展趋势，技术原理，解决方案，企业实践等多个角度阐释了微服务治理的原理，方案，和实践，希望能够帮助企业在下一阶段采纳微服务治理方案的时候有所参考。

展望未来，微服务治理的发展趋势，让业务迭代更加高效，让业务和治理更加透明，更加解耦，永远是技术发展的亘古不变的目标。

微服务架构分层逐渐清晰

微服务架构分层逐渐形成，后端 BAAS 化，客户端轻量化，业务侧 Serverless 化，让业务更加聚焦业务开发，进一步提升研发效率。

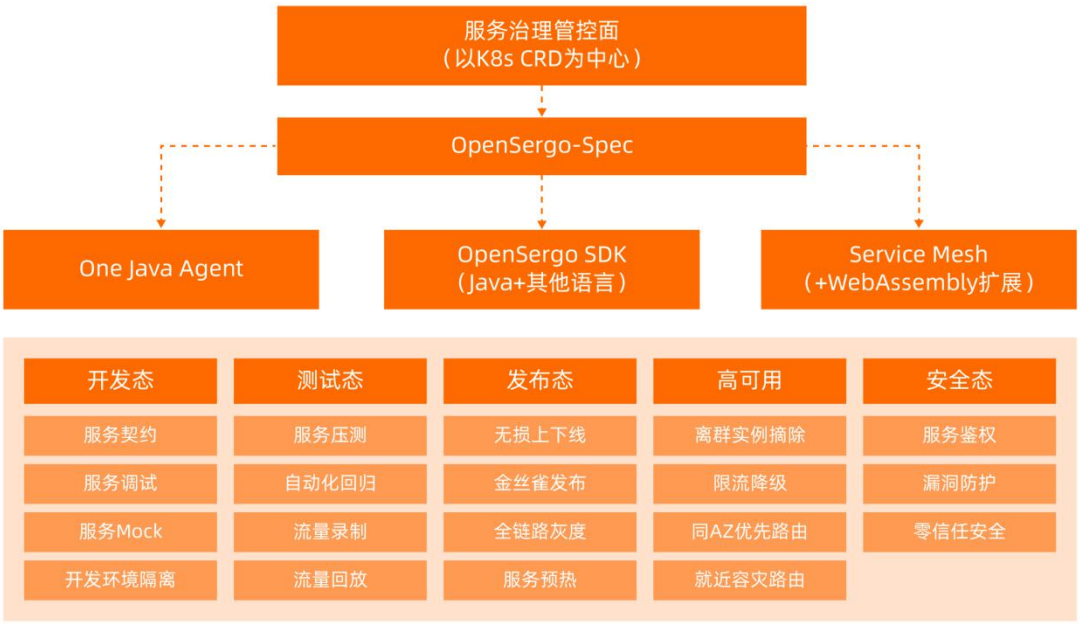
- 后端 BAAS 化，开箱即用，极致弹性，分钟级交付，安全可靠，提供 99.95% 高可用。
- 客户端轻量化：降低业务侵入，多语言标准集成，治理能力下沉。
- 业务侧 Serverless 化：让业务无需感知服务器，更加聚焦业务开发，提升研发和运维效率。



服务治理数据面透明化，控制面标准化

服务治理数据面将会逐步下沉，与业务逻辑逐步解耦，透明的实现治理技术的演进和升级。在数据面的形态上来看，存在多种形态并存，针对 Java 语言，以 Java Agent 为形态的的服务治理技术正在兴起并逐步成为趋势，针对非 Java 语言，基于 Sidecar 的 Service Mesh 技术正在被越来越多的企业采用，而在控制面，以一套控制面去控制不同数据面的形态成为主流，将逐步统一到以 K8s CRD 为中心的服务治理控制面中。服务治理的范围扩展到以开发，测试，发布，运维，安全等多场景的全生命周期。

观察到了这些趋势，我们和 bilibili、字节跳动、Apache Dubbo 社区、Nacos 社区、Spring Cloud Alibaba 社区共同发起了 OpenSergo 项目，致力于建立一套开放的、语言无关的云原生服务治理规范，目前已经开源：https://github.com/opensergo/opensergo-specification。


服务治理管控面
（以K8s CRD为中心）
OpenSergo-Spec
One Java Agent
OpenSergo SDK
（Java+其他语言）
Service Mesh
（+WebAssembly扩展）
开发态
服务契约
服务调试
服务Mock
开发环境隔离
测试态
服务压测
自动化回归
流量录制
流量回放
发布态
无损上下线
金丝雀发布
全链路灰度
服务预热
高可用
离群实例摘除
限流降级
同AZ优先路由
就近容灾路由
安全态
服务鉴权
漏洞防护
零信任安全