ALIBABA CLOUD

阿里云

金融分布式架构

中间件

文档版本：20210628

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.中间件产品总览 06
2.概述 08
3.获取 AccessKey 10
4.微服务平台 11
4.1. 微服务 11
4.2. 服务网格 11
4.2.1. 简介 11
4.2.1.1. 什么是服务网格 11
4.2.1.2. 产品优势 13
4.2.1.3. 产品架构 13
4.2.1.4. 部署方案 15
4.2.1.5. 应用场景 16
4.2.1.6. 使用限制 16
4.2.2. 快速入门 17
4.2.2.1. 准备工作 17
4.2.2.2. 服务发布 19
4.2.2.3. 服务管控和治理 24
4.2.3. SDK 版本说明 25
4.2.4. 连接 SOFA 服务注册中心 27
4.2.5. 控制台用户指南 32
4.2.5.1. 服务管控 32
4.2.5.2. 服务限流 33
4.2.5.3. 服务路由 36
4.2.5.4. 服务熔断 40
4.2.5.5. 故障注入 44
4.2.5.6. 服务鉴权 46

4.2.5.7. 故障隔离 47
4.2.5.8. 透明劫持 49
4.2.5.9. Sidecar 管理 53
4.2.5.10. Sidecar 监控 54
4.2.5.11. 服务拓扑 54
4.2.5.12. 实时监控 57
4.2.6. 日志说明 59
4.2.6.1. 管理 MOSN 日志级别 59
4.2.6.2. MOSN 内部日志 60
4.2.6.3. Pilot 日志 65
4.2.6.4. Pilot 详细日志说明 66
4.2.6.5. MOSN Trace 日志 68
4.2.7. 最佳实践 80
4.2.7.1. 服务网格落地之核心篇 80
4.2.7.2. 服务网格最佳实践之 RPC 90
4.2.7.3. 服务网格最佳实践之控制面 100
4.2.7.4. 服务网格最佳实践之控制面质量 107
4.2.7.5. 服务网格最佳实践之数据面质量 114
4.2.7.6. 服务网格最佳实践之网关 123
4.2.7.7. 服务网格最佳实践之消息 128
4.2.7.8. 服务网格最佳实践之 Operator 134
4.2.7.9. 服务网格最佳实践之运维 139

1.中间件产品总览

SOFA（Scalable Open Financial Architecture，以下简称 SOFA 中间件）作为蚂蚁科技自主研发的金融级分布式中间件，被广泛应用在包括支付、借贷、信用、基金、保险等全金融场景。
SOFA 中间件包含如下产品：

SOFABoot

- 基于 Spring Boot 自研的开发框架
- 集成 SOFA 中间件，且中间件可插拔

服务网格

- 将传统微服务和 Service Mesh 进行融合
- 对接经典应用服务和容器应用服务
- 提供丰富的服务治理功能

消息队列

- 基于 Apache RocketMQ 构建的分布式消息中间件
- 提供高可用消息云服务

任务调度

- 提供分布式任务调度框架
- 提供自动化任务调度服务

分布式事务

- 金融级分布式事务中间件
- 支持跨数据库、跨服务以及混合分布式事务

数据访问代理

- 通过 MySQL 协议与 RDS\OceanBase 通信
- 通过触发 DDL 任务管理数据库与数据表

API 统一网关

- 提供高性能、高安全、高可靠的 API 托管服务
- 通过 API 网关实现 API 发布与调用

分布式链路追踪

- 面向分布式、微服务架构的一种可观察性的金融级解...
- 通过 TFAT racer 进行透视化网络调用

数据同步服务

- 提供数据流传输与转换平台

2.概述

SOFAStack 提供了一套用于快速构建金融级分布式架构的中间件，是在严苛的金融场景里锤炼出来的最佳实践。要使用 SOFA 中间件，例如微服务或消息队列，推荐使用 SOFABoot 框架进行开发。

SOFABoot

SOFABoot 是蚂蚁金服基于 Spring Boot 自研的开发框架，在 Spring Boot 的基础上不仅进行了能力增强，并提供了 SOFA 中间件的轻量集成，每一个中间件均是一个可插拔的组件。开发者在集成了 SOFABoot 框架后，只需引入对应中间件的 starter，SOFABoot 即会自动导入所需的依赖并完成必要的配置，也能自动解决后续的健康检查、运维监控等问题，开发者能够更加专注于业务逻辑，有效节约了开发时间及后期维护的成本。

使用 SOFABoot 开发框架，就相当于是在进行“Spring Boot 应用 + SOFA 中间件”开发。您可以从 Github 直接下载 SOFABoot 工程包，也可以在本地通过 Maven 快速创建一个 SOFABoot 工程，参见 SOFABoot 快速入门。

微服务平台

SOFAStack 微服务平台主要提供分布式应用常用解决方案。使用微服务框架开发应用，在应用托管后启动应用，微服务会自动注册到服务注册中心，您可以在微服务平台控制台进行服务管理和治理的相关操作。

微服务平台通过微服务和服务网格，提供了既支持 SOFA 框架又支持 Service Mesh 架构的微服务管理和治理能力。微服务提供了 SOFA 框架的微服务，包含 RPC 服务、动态配置、限流熔断。服务网格通过 Service Mesh 技术支持原生 Dubbo 、Spring Cloud、SOFA 框架，无侵入地提供了对 Dubbo、Spring Cloud、SOFA 应用的服务管理和治理能力。

更多微服务平台的详情，参见 什么是微服务平台。

消息队列

SOFAStack 消息队列是基于 Apache RocketMQ 构建的分布式消息中间件，并与金融分布式架构 SOFAStack 深度集成，为分布式应用系统提供异步解耦和削峰填谷的能力，支持事务消息、顺序消息、定时消息等多种消息类型，并具备高可靠、高吞吐、低延时等金融级特性。

更多消息队列的详情，参见 什么是消息队列。

分布式链路跟踪

SOFAStack 分布式链路跟踪是面向分布式架构、微服务架构与云原生架构的应用可观察性（Observability）的金融级解决方案。通过分布式链路跟踪，运维人员、开发人员和架构师能看清楚复杂的大规模微服务架构下的应用及服务之间的复杂调用关系、性能指标、出错信息与关联日志，从而实现故障根因分析、服务治理、应用开发调试、性能管理、性能调优、架构管控、故障定责等运维开发工作。

更多分布式链路跟踪的详情，参见 什么是分布式链路跟踪。

任务调度

SOFAStack 任务调度提供分布式任务调度框架，实现任务的分布式处理，并能规范化、自动化、可视化和集中化对金融企业不同业务系统的任务进行统一的调度和全方位监控运维管理，达到所有任务有序、高效运行的目的，极大降低开发和运维的成本。

更多任务调度的详情，参见 什么是任务调度。

数据访问代理

SOFAStack 数据访问代理是金融级分布式数据库中间件，用于解决海量请求下数据访问的瓶颈及数据库的容灾问题，提供水平拆分、平滑扩缩容、读写分离的在线分布式数据库服务。数据访问代理兼容 MySQL 协议和语法，支持分库分表、平滑扩容、服务升降配、透明读写分离和分布式事务等特性，具备分布式数据库全生命周期的运维管控能力。

更多数据访问代理的详情，参见 什么是数据访问代理。

数据同步服务

SOFAStack 数据同步服务是蚂蚁金融科技提供的数据流传输与转换平台。它具备多种数据库的日志解析能力，包括 MySQL、RDS、Oracle、OceanBase、HBase 等。下游可通过数据订阅功能订阅到这些数据库的实时数据。同时，还可以将实时数据同步到 OceanBase、Metaq、MaxCompute（原 ODPS） 等多种不同的存储。

更多数据同步服务的详情，参见 什么是数据同步服务。

分布式事务

SOFAStack 分布式事务是蚂蚁金服自主研发的金融级分布式事务中间件，用来保障在大规模分布式环境下业务活动的最终一致性。在蚂蚁金服内部被广泛地应用于交易、转账、红包等核心资金链路，服务于亿级用户的资金操作。

更多分布式事务的详情，参见 什么是分布式事务。

API 网关

SOFAStack API 网关是一个 API 管理平台，帮助企业统一管理对内外开放的 API，为网络隔离的系统间提供高性能、高安全、高可靠的通信，同时保障内部系统的安全性。用于满足企业对外部合作伙伴开放业务，企业自身混合云互通、企业内网应用集成异构系统间通信的需求，帮助客户更好的进行场景和业务的创新。

更多 API 网关的详情，参见 什么是 API 统一网关。

3.获取 AccessKey

阿里云的所有的权限都是通过访问控制 RAM（Resource Access Management）进行管理，SOFAStack 中间件作为阿里云产品，也使用 RAM 作为权限的统一管理。本文介绍如何获取账号的 AccessKey。

前提条件

已创建 RAM 用户。操作步骤请参见 创建 RAM 用户。

背景信息

AccessKey 包括 AccessKey ID 和 AccessKey Secret。

- AccessKey ID：用于标识用户。
- AccessKey Secret：是于验证用户的密钥。AccessKey Secret 必须保密。

对接中间件时，因考虑安全问题，您必须做好应用的身份认证，避免非法访问，因此您需要在代码配置文件 `application.properties` 中配置账号的 AccessKey 信息：

```
com.antcloud.mw.access=<AccessKey ID>
com.antcloud.mw.secret=<AccessKey Secret>
```

> **注意**
>
> - 禁止使用阿里云账号 AccessKey，因为阿里云账号 AccessKey 泄露会威胁您所有资源的安全。请使用 RAM 用户 AccessKey 进行操作，可有效降低 AccessKey 泄露的风险。
> - 生产环境中，建议单独为中间件创建一个 RAM 用户，以便于您控制权限，方便管理。

操作步骤

1. 使用阿里云账号登录 RAM 控制台。
2. 选择 **人员管理 > 用户**，然后单击目标用户名称。
3. 单击 **创建 AccessKey**。
   首次创建时需通过手机校验码进行认证。
4. 保存生成的 AccessKey ID 和 AccessKey Secret，然后单击 **关闭**。

> **警告**
>
> - AccessKey Secret 只在创建时显示，不提供查询，请妥善保管。
> - 如果 AccessKey 泄露或丢失，请及时删除，并创建新的 AccessKey 。每个 RAM 用户最多可以创建 2 个 AccessKey。

4.微服务平台

4.1. 微服务

微服务（SOFAStack Microservices，简称 SOFAStack MS）主要提供分布式应用常用解决方案。使用微服务框架开发应用，在应用托管后启动应用，微服务会自动注册到服务注册中心，您可以在微服务控制台进行服务管理和治理的相关操作。微服务主要提供分布式应用常用解决方案，包含 RPC 服务、动态配置、限流熔断等。

关于微服务详细的功能介绍和使用指南，请参见 SOFAStack 微服务。

4.2. 服务网格

4.2.1. 简介

4.2.1.1. 什么是服务网格

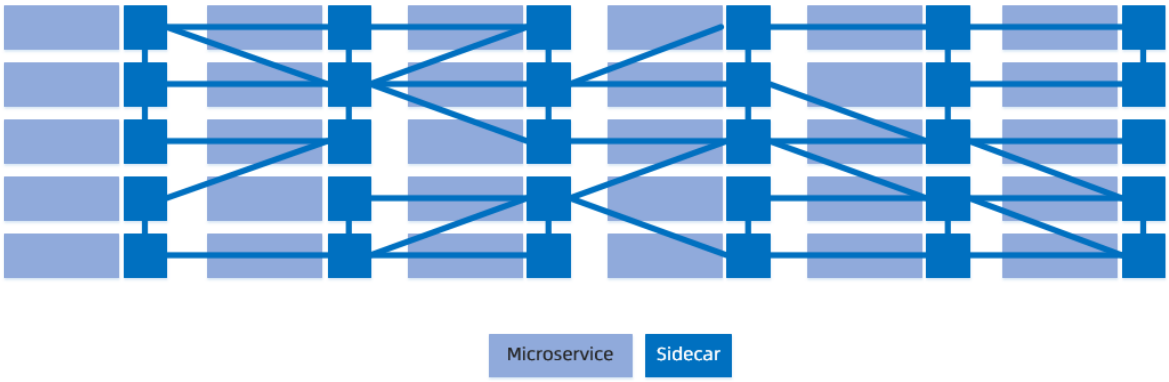
服务网格（SOFAStack Mesh）是蚂蚁集团自主研发的基于金融级生产实践的增强版服务网格平台，将传统微服务和 Service Mesh 技术进行了深度融合，其核心技术经过了蚂蚁集团的大规模生产实践验证。它深度、无缝对接了 SOFAStack 经典应用服务和容器应用服务，为客户提供了简单易用的 Service Mesh 架构的支撑平台。

SOFAStack 服务网格概述

SOFAStack 服务网格（简称服务网格）= 传统微服务 + Service Mesh。

服务网格旨在提供与平台无关、语言无关、轻量无侵入的云原生分布式架构解决方案，在兼容 Kubernetes 和 Istio 生态的基础上还能支持虚拟机场景，同时支持 SOFA、Dubbo、SpringCloud 微服务框架的服务治理能力。在整个接入过程中，无需修改业务代码，帮助企业以最小的成本完成云原生落地，加速数字化转型。

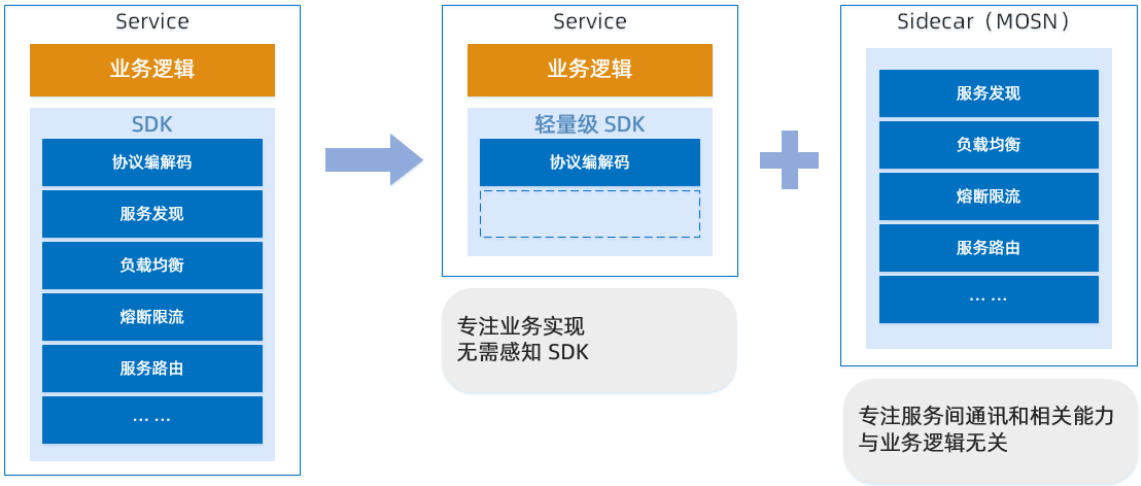
Service Mesh 图解



从一个全局视角来看，Sidecar 连接成网，组成 Service Mesh。

TCP 协议催生了分布式系统，分布式系统催生了微服务，Service Mesh 就是下一代微服务技术的代名词，是微服务时代的 TCP 协议。Service Mesh 以 Sidecar 形式，将服务治理从业务逻辑中剥离，并拆解为独立进程，实现异构系统的统一治理和网络安全。

Sidecar 模式图解



Service Mesh 将 SDK 拆解为单独进程，并以 Sidecar 模式进行部署，使得业务进程专注于业务逻辑。Sidecar 是一个轻量级的网络代理，它们与应用程序部署在一起，对所有流入与流出的网络请求进行拦截，实现各种网络策略，例如：服务发现与负载均衡、流量拆分、故障注入、熔断器以及分阶段发布等功能。

功能一览

服务网格包含以下子产品和功能，可以进行各种服务管理和监控操作。

- 服务管控
  支持异构服务的统一注册与服务详情查看，包括其基本信息、服务提供者以及消费者信息等。
- 服务治理
  支持服务限流、服务路由、服务熔断降级、服务鉴权，提供了精细化调配的服务路由与服务限流功能，保证应用高可用，保障业务持续运行。
- 透明劫持
  应用开启透明劫持功能后，出入应用的业务流量将会被 Sidecar Proxy 自动拦截，继而按照您在控制台配置的规则进行观测与治理。
- Sidecar 管理
  支持查看及管理当前工作空间中的 Sidecar 实例，提供了 Sidecar 状态、Pod 状态、Sidecar 注入时间等信息。
- 服务拓扑
  可视化展示了不同应用服务之间的调用关系和依赖关系，以及各节点的实时监控信息，包括请求量、响应时间及错误率等。
- 实时监控
  提供了所有应用以及各项性能指标的总体统计数据等。支持实时监控应用服务的吞吐量、响应时间、RPS、状态，及时发现应用服务异常。

4.2.1.2. 产品优势

在传统单体式架构向微服务架构迁移的过程中，随着应用微服务数量的增加，微服务间的通信、监控以及安全性的管理成为新的挑战。服务网格作为应用与基础设施的桥梁，突破传统的 SDK 接入方式，以对应用透明的方式处理服务之间、服务与基础设施间的通信，实现应用研发和基础设施最大程度地解耦，让应用研发只需专注于业务开发。

自研自用

- 蚂蚁内部 1500+ 应用，40万+ 容器的规模
- 业界最大规模的 Mesh 集群
- 支持内部聚石塔、IoT 等业务场景

开源开放

- 中国最大的社区 Service Mesh
- 底层技术开源开放
- 先驱布道者和引领者

生产级优化

解决 Istio + Envoy 无法生产落地的问题

- Mixer 性能问题，out of process 到 in proxy 模式的升级
- 突破 k8s 服务注册模型，支持接口级别+应用级别的服务注册增量下发机制，
- 解决 xds 全量下发引起的性能问题

产品特性

- **多协议**：适配业界通用同步、异步通信协议，具备可扩展性和协议转换能力
- **多平台**：容器、虚机
- **多注册中心**：适配业界通用的注册中心
- **安全**：mtls、国密、rbac等安全能力

产品矩阵

- RPC 能力下沉的 Service Mesh
- 异步通讯的消息 Mesh
- 数据代理能力的 DB Mesh
- Mesh 化网关

客户案例

- 网商银行
- 江西农信
- 中信银行
- 工商银行
- 华夏银行
- 浙商证券
- 华瑞银行

技术业务分离

在之前，微服务体系都是由中间件团队提供一个 SDK 给业务应用使用，在 SDK 中会集成各种服务治理的能力，如：服务发现、负载均衡、熔断限流、服务路由等。有了服务网格之后，我们就可以把 SDK 中的大部分能力从应用中剥离出来，拆解为独立进程，以 Sidecar 的模式部署。通过将服务治理能力下沉到基础设施，可以让业务更加专注于业务逻辑，中间件团队则更加专注于各种通用能力，真正实现独立演进，透明升级，提升整体效率。

异构服务治理

既能借助蚂蚁集团久经考验的微服务框架 SOFA 在云上构建微服务应用，也可以支持原生 Dubbo 和 Spring Cloud 上云，无需构建 ZooKeeper、Eureka、Consul 等微服务依赖的自建服务，极大降低运维成本。

轻量无侵入接入

业务应用系统通过 Service Mesh 技术架构轻量级接入，实现对应用无侵入的服务注册与服务治理方案，减少改造成本。

跨平台支持

该方案支持容器平台、虚拟机平台，能够满足企业用户未容器化的场景对 Service Mesh 架构转型的需求。

网络安全

通过网格安全，可以更方便地实现应用的身份标识和访问控制，辅之以数据加密，就能实现全链路可信，从而使得服务可以运行于零信任网络中，提升整体安全水位。

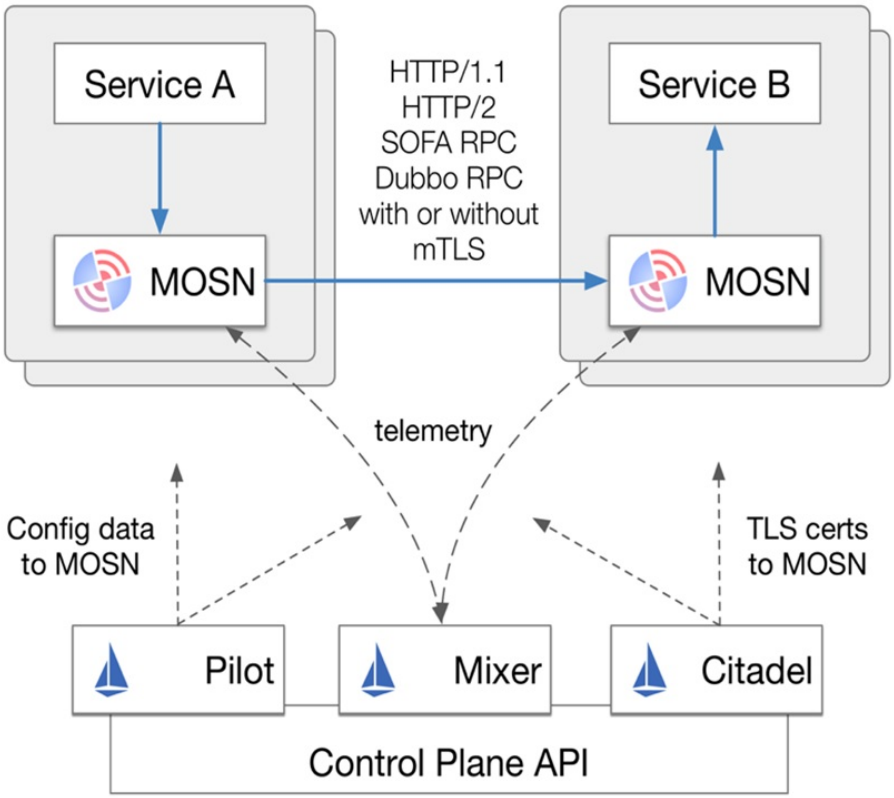
简单易用易维护

SOFAStack 服务网格提供集中式图形化易操作的管理平台，支持多租户管理能力，满足企业级高级特性需求，简化分布式应用的服务管理、服务治理、可观察性等能力，让用户便捷的对应用服务统一管理和治理。

4.2.1.3. 产品架构

服务网格的产品架构逻辑上分为控制层面和数据层面。

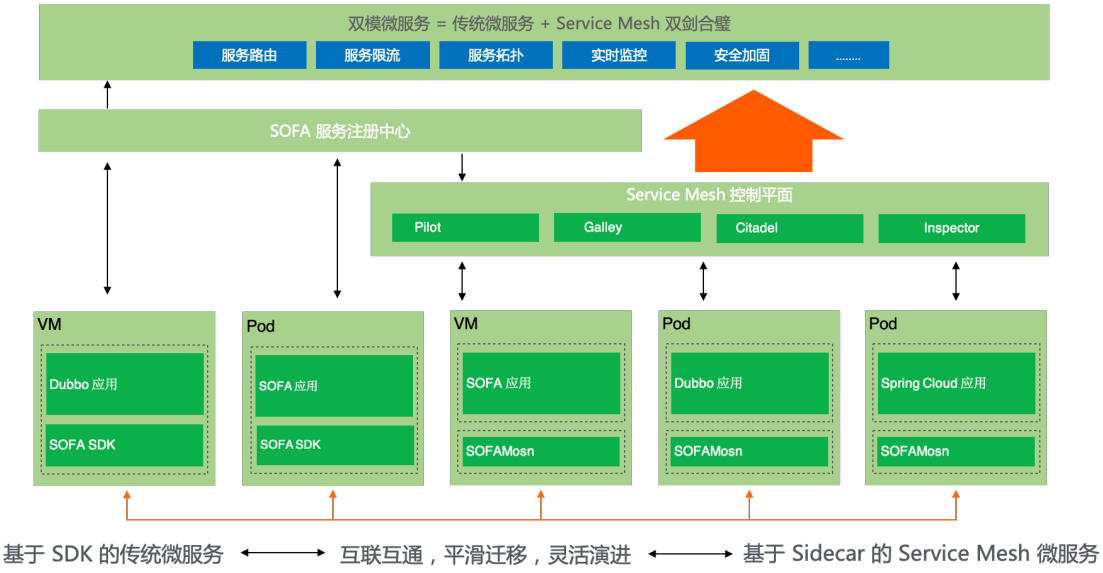
- 控制层面：管理代理（蚂蚁自研 MOSN），用于管理流量路由、运行时策略执行等。
- 数据层面：由一系列代理（蚂蚁自研 MOSN）组成，用于管理和控制服务之间的网络通信。



系统架构

SOFAStack 服务网格结合了 SOFAStack 微服务的优势，将 SOFAStack 微服务和 Service Mesh 双剑合璧，即基于 SDK 的传统微服务可以和基于 Sidecar 的 Service Mesh 微服务实现下列目标：

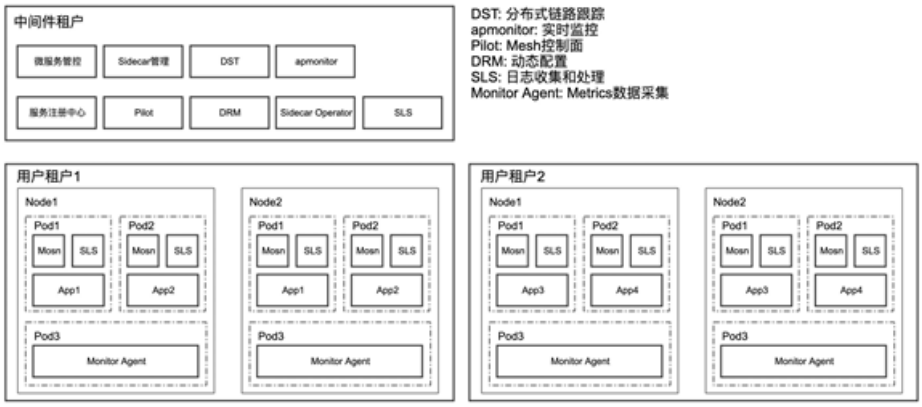
- 互联互通：两个体系中的应用可以相互访问。
- 平滑迁移：应用可以在两个体系中迁移，对于调用该应用的其他应用，做到透明无感知。
- 灵活演进：在互联互通和平滑迁移实现之后，我们就可以根据实际情况进行灵活的应用改造和架构演进。



在控制面上，引入了 Pilot 实现配置的下发（如服务路由规则），在服务发现上保留了独立的 SOFA 服务注册中心。
在数据面上，使用了自研的 SOFA MOSN，不仅支持 SOFA 应用，同时也支持 Dubbo 和 Spring Cloud 应用。
在部署模式上，我们不仅支持容器/k8s，同时也支持虚拟机场景。

4.2.1.4. 部署方案

服务网格部署分为中间件租户部署和用户部署。
部署方案图如下：



在中间件租户会部署以下组件：

- 微服务管控：服务治理后台，用户可以在微服务管控系统上查看服务实例信息、配置服务限流、路由规则等。
- Sidecar 管理：边车管理后台，用户可以在该系统上查看 Sidecar 信息进行运维操作等。
- DST：分布式链路跟踪后台，用户可以在该系统上查看服务调用链信息。
- Apmonitor：实时监控后台，用户可以在该系统上查看实时的服务调用信息统计，如调用次数，平均响应时间，错误率等。
- 服务注册中心：服务信息都会注册到该系统，同时也提供服务发现接口供调用者做服务寻址。
- Pilot：Mesh 控制面，把用户配置以 xds 的形式下发给 Sidecar。
- DRM：动态配置中心，用于动态下发用户的配置。

- Sidecar Operator：边车运维系统，负责注入、升级 Sidecar。
- SLS：日志收集系统，用于收集、处理应用日志。

在用户租户会部署以下组件：

- SLS：在用户应用的每个 Pod 中会注入 SLS 组件（logtail），用于采集 Pod 的日志信息，并上报给 SLS 服务端，从而可以分析链路追踪信息。
- Monitor Agent：在用户租户的每台 ECS 节点上部署，用于采集该节点上所有容器的 metrics。
- Mosn：在用户应用的每个 Pod 中会注入 MOSN，用于拦截服务的流量，从而提供服务治理、流量调拨、通信加密等服务网格的能力。

4.2.1.5. 应用场景

依据服务网格特点和优势，服务网格适用于以下应用场景。

服务治理与业务逻辑解耦

在运行时，SDK 和业务应用的代码是混合在一个进程中运行的，耦合度非常高，会产生如下问题：

- 升级成本高：每次升级都需要业务应用修改 SDK 版本号，重新发布。在业务飞速发展的时候，更倾向于专注业务，不希望分散精力做升级等事情。
- 版本碎片化严重：由于升级成本高，但中间件还是会向前发展，久而久之，就会导致线上 SDK 版本各不统一、能力参差不齐，造成很难统一治理。
- 中间件演进困难：由于版本碎片化严重，导致中间件向前演进过程中需要兼容各种老版本的逻辑，无法实现快速迭代。

使用 Service Mesh 后，您可以将 SDK 中的大部分能力从应用中剥离拆解为独立进程，以 Sidecar 的模式部署。通过将服务治理能力下沉到基础设施，可以让业务更加专注于业务逻辑，中间件团队则更加专注于各种通用能力，真正实现独立演进、透明升级，提升整体效率。

多语言、多协议支持

随着新技术的发展和人员更替，在同一家公司中往往会出现使用各种不同语言、不同框架的应用和服务，为了能够统一管控这些服务，以往的做法是为每种语言、每种框架都重新开发一套完整的 SDK，维护成本非常高，而且对中间件团队的人员结构也带来了很大的挑战。
有了服务网格之后，通过将主体的服务治理能力下沉到基础设施，多语言的支持就轻松很多了，只需要提供一个非常轻量的 SDK，甚至很多情况都不需要一个单独的 SDK，就可以方便地实现多语言、多协议的统一流量管控、监控等治理需求。

云原生架构转型助力

从单体应用到微服务架构改造以及全面容器化的云原生架构基础往往带来很高的改造成本。SOFAStack 服务网格可以满足未容器化的虚拟机部署方案，也可以兼容过渡阶段的部分容器化和虚拟化混合部署的场景，加速企业云原生架构转型。

金融场景网络安全

当前很多公司的微服务体系建设都建立在内网可信的假设之上，然而这个原则在当前大规模上云的背景下可能显得有点不合时宜，尤其是涉及到一些金融场景的时候。
通过服务网格可以更方便地实现应用的身份标识和访问控制，辅之以数据加密，就能实现全链路可信，从而使得服务可以运行于零信任网络中，提升整体安全水位。

4.2.1.6. 使用限制

服务网格在使用时有一些限制，比如 JDK 版本、微服务类型等。
详细的限制信息如下：

限制项	限制范围	限制说明
微服务开发框架	Dubbo、SpringCloud、SOFA	服务注册中心 SDK 仅支持这三种框架的兼容。
SOFA SDK 语言限制	Java	SOFA SDK 仅支持 Java 语言开发。
JDK 版本	JDK 1.8 及以上	仅支持 JDK 1.8 及以上的版本。
虚拟机操作系统版本（虚拟机部署）	虚拟机操作系统版本（虚拟机部署）	虚拟机操作系统版本（虚拟机部署）。

4.2.2. 快速入门

4.2.2.1. 准备工作

服务网格将 SDK 拆解为单独进程，使得业务进程专注于业务逻辑；将服务通讯、治理部分以 Sidecar 形式组成网格，实现异构网络的统一治理。体验服务网格需进行服务发布，在进行服务发布之前，需要完成服务发布所需的准备工作。
服务网格支持 SOFA、Dubbo、SpringCloud 类型的微服务。下面以 SOFABoot 工程为例指导您快速入门服务网格。鉴于不同发布方式对应用的构建、发布和管控流程不同，下文对经典虚机发布和容器服务发布要做的准备工作分别进行说明。

视频内容时间段说明如下：

操作步骤	时间段
1、安装基础环境	开始~7:47
2、创建工程	7:48~14:26
3、引入 SOFA 中间件	14:27~18:24
4、打发布包	18:25~19:24
5、镜像制作	19:25~26:19
6、镜像上传	26:19~结束

经典虚机发布

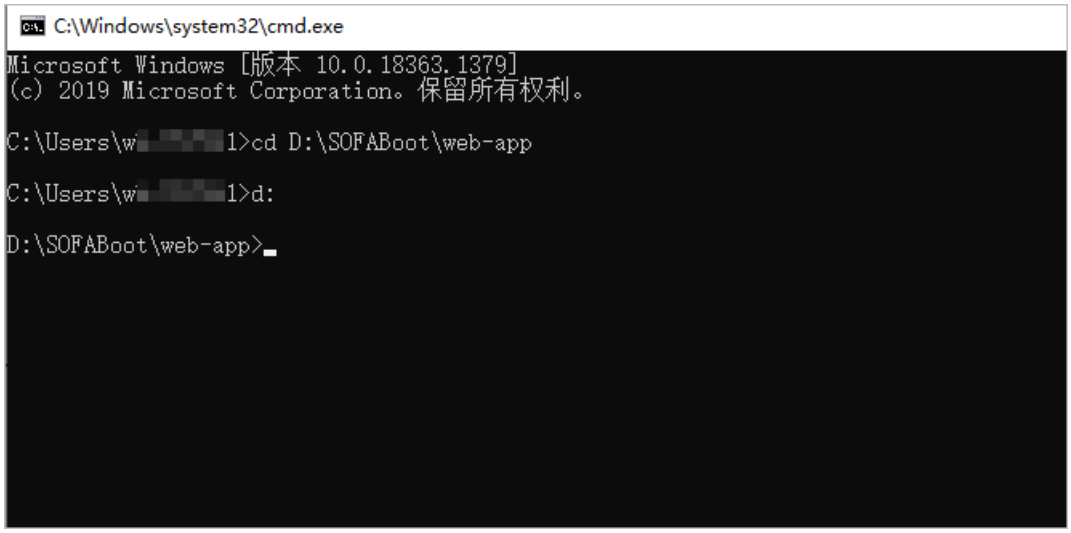
经典虚机发布的对象是 JAR 包，在经典虚机发布之前，需要准备好 JAR 包。

- 直接下载示例 JAR 包。
  - 下载 Web 工程 JAR 包。
  - 下载 Core 工程 JAR 包。
- 使用本地工程打 JAR 包。
  i. 环境准备，详情请参见 搭建环境。
  ii. 工程准备。
  可以通过以下两种方式获取可运行的 Demo 工程。

- 创建工程
  通过 Maven 命令创建 Web 工程和 Core 工程，详细创建步骤请参见 创建工程。
- 下载示例工程
  Web 示例工程下载地址，请参见 编译运行 Web 工程 > 下载示例 Demo。
  Core 示例工程下载地址，请参见 编译运行 Core 工程 > 下载示例 Demo。

iii. 引入 SOFA 中间件。
详细信息请参见 引入 SOFA 中间件。

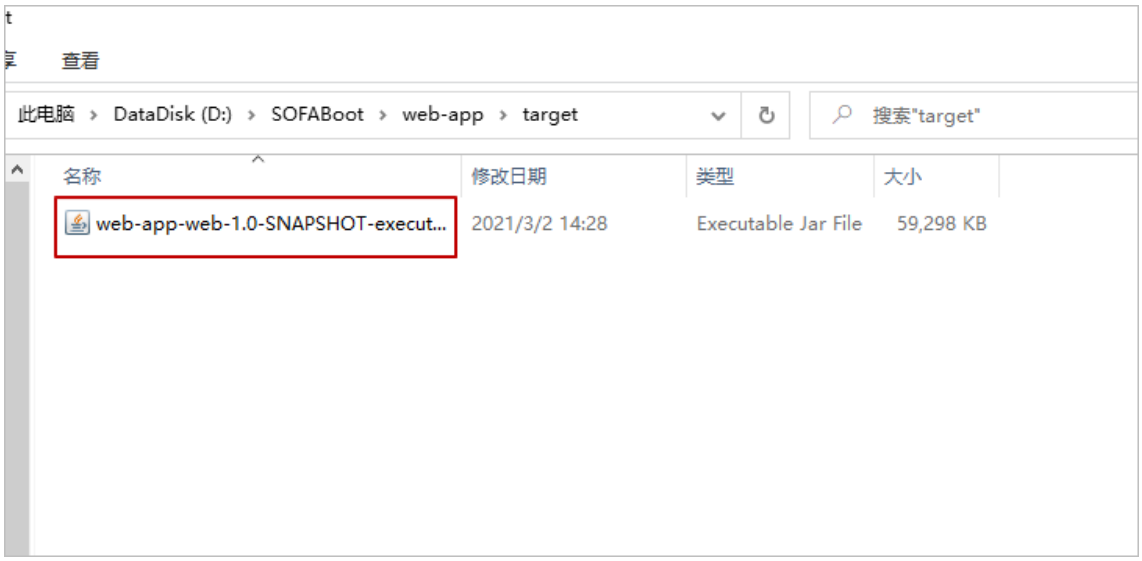
iv. 打开 DOS 命令行，进入工程根目录，如：`D:\SOFABoot\web-app` 。



v. 执行如下命令打包。

```
mvn clean package
```

vi. 打开根目录下的 `target` 文件夹，查看打包好的工程 JAR 包。



容器服务发布

容器服务发布的对象是镜像。在容器服务发布之前，需要准备好镜像。提供以下两种方式准备镜像。

- 示例镜像

蚂蚁金融科技提供一系列的镜像示例，您可以根据应用服务所在地域进行选择。具体镜像请参见 **示例镜像列表**。

- 本地镜像
  您也可以使用本地构建的自定义镜像，并将镜像上传到阿里云 ACR 镜像仓库。下面介绍如何制作镜像并上传至 ACR 镜像仓库。

  i. 准备工程
  详情请参见 **创建工程**。

  ii. 制作镜像

  a. 安装 Docker。

  > 注意
  > 安装的 Docker 版本必须为 1.6.0 及以上，安装步骤请参见 **Docker 官网安装步骤**。

  b. 制作镜像。
  详情请参见 **制作 SOFABoot 应用的 Docker 镜像**。

  iii. 上传镜像
  安全起见，金区不能通过外网访问，所以往金区和非金区上传镜像的方式不同，下面对金区和非金区上传分别进行说明。金区和非金区的详细信息请参见 **金融地域（金区）**。

  - 往金区上传镜像。
    详情请参见 **往金区上传镜像**。
  - 往非金区上传镜像。
    详情请参见 **镜像仓库**。

  iv. 获取镜像
  获取镜像地址的步骤如下：

  a. 选择 **容器应用服务** > **镜像中心** > **镜像仓库**，进入镜像仓库列表页。

  b. 定位到准备工作里创建的镜像仓库，单击镜像仓库名称。

  c. 在镜像仓库详情页，单击镜像版本号旁的复制图标获取镜像地址。

镜像地址由仓库地址和版本构成，示例如下：

```
registry-vpc.cn-hangzhou-finance.aliyuncs.com/sofaboot-public/hz_fin_sofaboot:1.0.0
```

4.2.2.2. 服务发布

应用服务发布到 SOFAStack 平台，相应的应用就可以在云端运行，进行后续的服务管控和服务治理操作。本文介绍如何进行服务发布。
根据服务发布方式分为 **经典服务发布** 和 **容器服务发布**。

经典服务发布

将本地工程打成 JAR 包，采用虚拟机构建模式发布。

1. 创建应用。
   详细操作步骤请参见 **管理应用**。

   > **注意**
   > 应用要开启服务网格功能，需注意以下两点：
   > - 应用名称要和项目的应用名称一致。
   >   项目的应用名称可以在 `app/web/src/main/resources/config/application.properties` 文件里查看，示例如下：
   >
   >   ```
   >   #required
   >   spring.application.name=web-app
   >   ```
   > - 应用的技术栈需选择 SOFA Boot，了解什么是技术栈请参见 **技术栈使用指南**。

2. 创建部署单元。
   详细操作步骤请参见 **部署单元**。
3. 上传发布包。
   将在 **准备工作** 生成的 JAR 包，上传至 SOFAStack 平台。详细操作步骤请参见 **发布包管理**。
4. 创建应用服务。
   详细创建步骤请参见 **应用服务实例**。

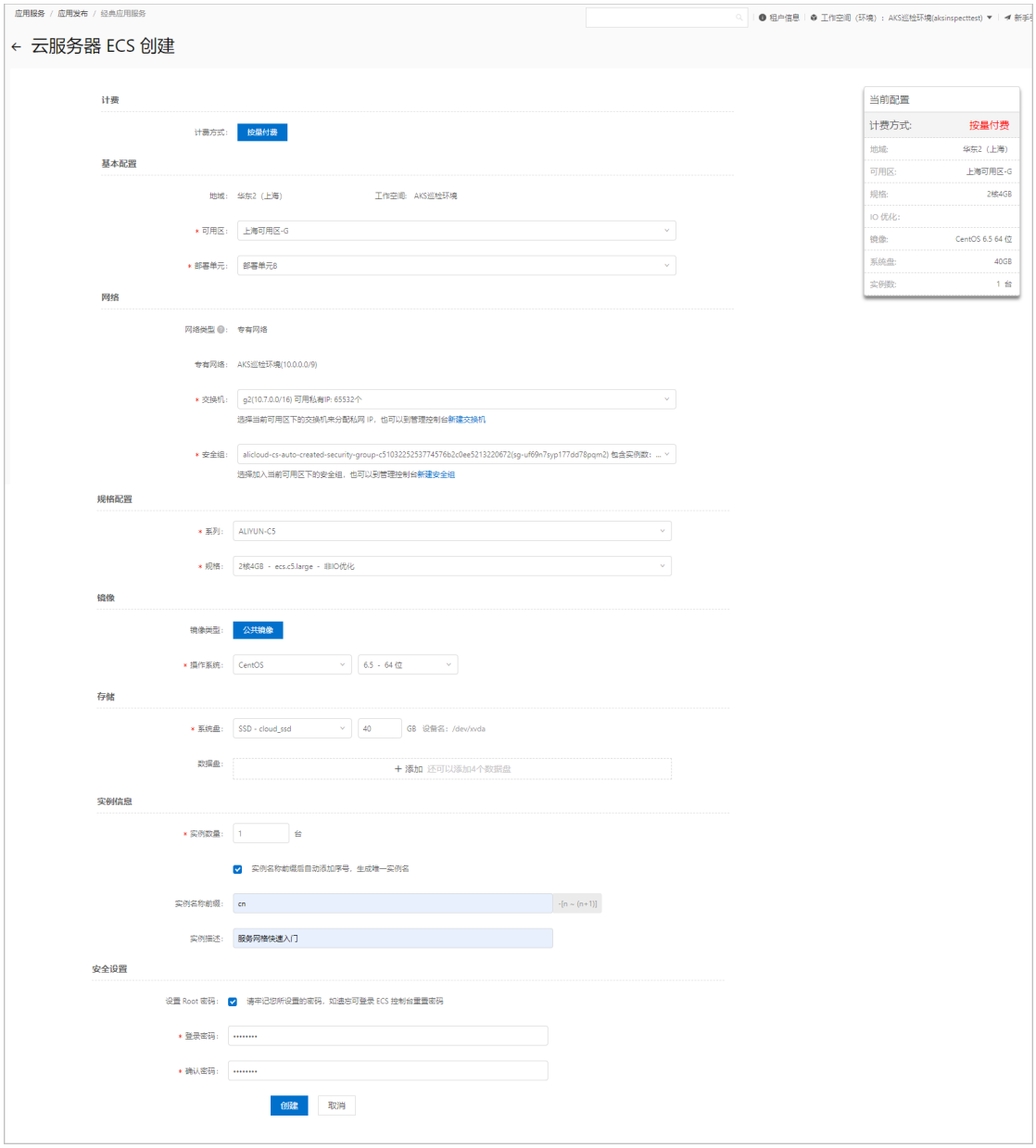
   > **注意**
   > 应用服务要开启服务网格功能，技术栈版本要选开通 Mesh 功能的版本，技术栈版本说明请参见 **技术栈说明**。

   创建完应用服务之后，需添加云服务器 ECS，下面介绍如何添加云服务器 ECS。

   i. 在 **应用服务列表** 页，定位到新创建的应用服务，单击服务实例名，进入应用服务实例详情页。

   ii. 单击 **云服务器 ECS** 页签下的 **添加** 按钮。

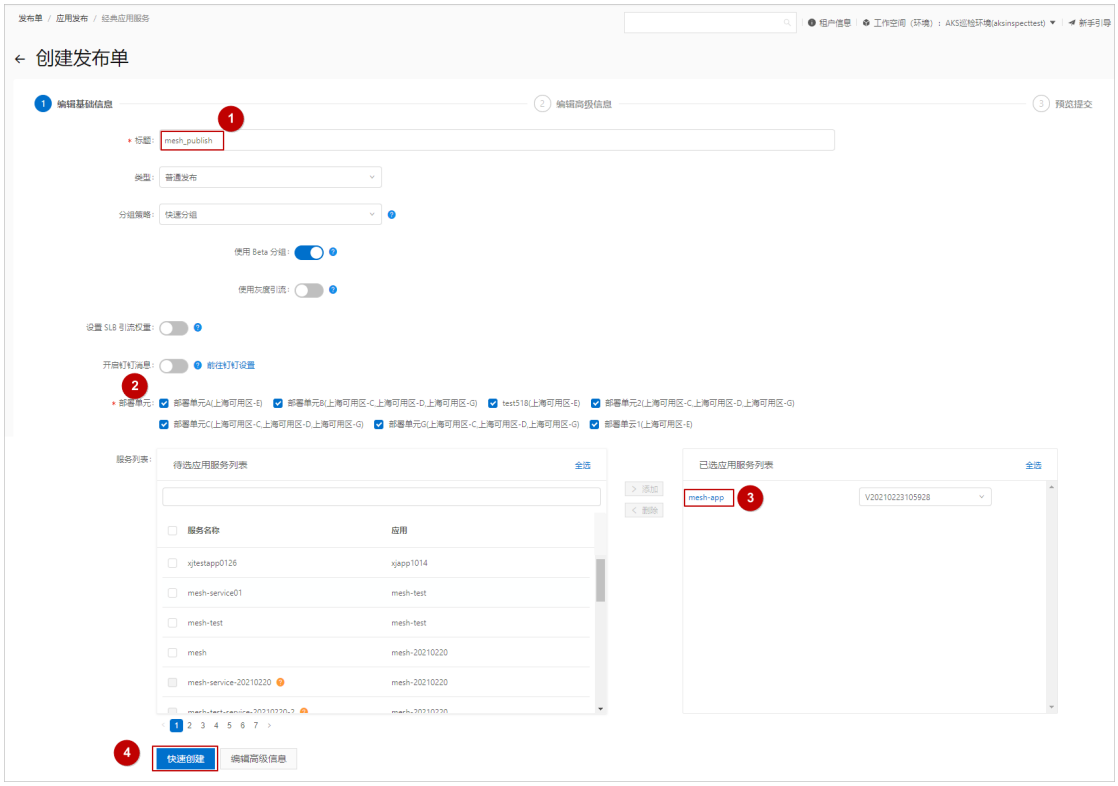
iii. 输入ECS 信息。
详细操作步骤请参见 云服务器 ECS。



iv. 单击 **创建**。

5. 创建发布单。
   i. 进入 发布单 页面，单击 **创建** 按钮。
   ii. 进行发布单配置，然后单击 **快速创建** 按钮。

发布单配置示例，如下图所示：



如果要在创建的时候修改中间件四元组信息，可以单击 **编辑高级信息** 按钮，进行下面的操作进行修改；

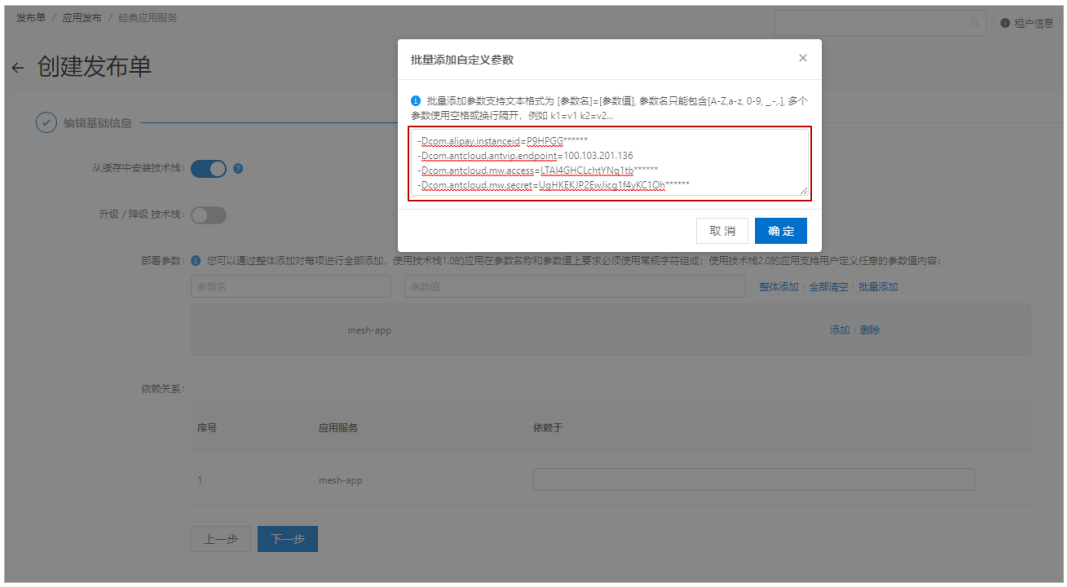
如果不需要修改，直接进行下一步的操作。

a. 单击 **批量添加** 按钮。

b. 添加自定义参数。
示例值如下：

```
-Dcom.alipay.instanceid=P9HPGG****** #当前登录账号的实例 ID
-Dcom.antcloud.antvip.endpoint=cn-hangzhou-middleware-acvip-prod.cloud.alipaycs.net #当前地域的 AntVIP 地址值
-Dcom.antcloud.mw.access=LTAI4GHCLchtYNq1tb****** #当前登录账号的 AK 信息
-Dcom.antcloud.mw.secret=UgHKEKJP2EwJicg1f4yKC1Oh****** #当前登录账号的 SK信息
```

> **说明**
> 以上参数值是中间件的全局配置项，可在 **脚手架控制台** 获取，详情请参见 **引入 SOFA 中间件 > 中间件全局配置**。



c. 单击 **确定**。

d. 单击 **下一步**。

e. 单击 **创建**，在原始发布单基础上创建一个新的发布单。

iii. 在发布单详情页，单击右上方的 **整体发布** 按钮发布单。

6. 查看已发布的服务。
发布单发布成功后，即可在 **微服务平台 > 服务网格 > 服务管控** 页面查看已发布的服务。

容器服务发布

将本地工程制作成镜像文件，采用容器构建模式发布。

1. 创建集群。
详细操作步骤请参见 **创建集群**。

> **注意**
> 需给集群开启 **服务网格** 功能。

2. 创建 SOFA Boot 应用。
详细操作步骤请参见 **管理应用**。
3. 创建应用服务。

详细操作步骤请参见 创建应用服务。创建应用服务时，需填写 AKS 可以访问到的镜像地址，并且为应用服务注入中间件四元组信息。

> 注意
> - 需 注入 SOFAMesh。
> - 镜像地址需和服务在同一个地域。

在创建应用服务时，可以修改中间件四元组信息。修改步骤如下：

i. 在 应用服务 列表页，单击要修改的应用服务的名称。

ii. 单击右上方的 **编辑配置信息**。

iii. 单击 **下一步**。

iv. 单击 **高级配置** 下的 **环境变量配置** 对应的展开图标 > 。

配置示例如下：

变量名	示例值
com.alipay.instanceid	P9HPGG******
com.antcloud.antvip.endpoint	cn-hangzhou-middleware-acvip-prod.cloud.alipaycs.net
com.antcloud.mw.access	LTAI4GHCLchtYNq1tb******
com.antcloud.mw.secret	UgHKEKJP2EwJicg1f4yKC1Oh******

4. 创建发布单。
   详细操作请参见 发布单。
5. 查看已发布的服务。
   发布单发布成功后，即可在 微服务平台 > **服务网格** > **服务管控** 页面查看已发布的服务。

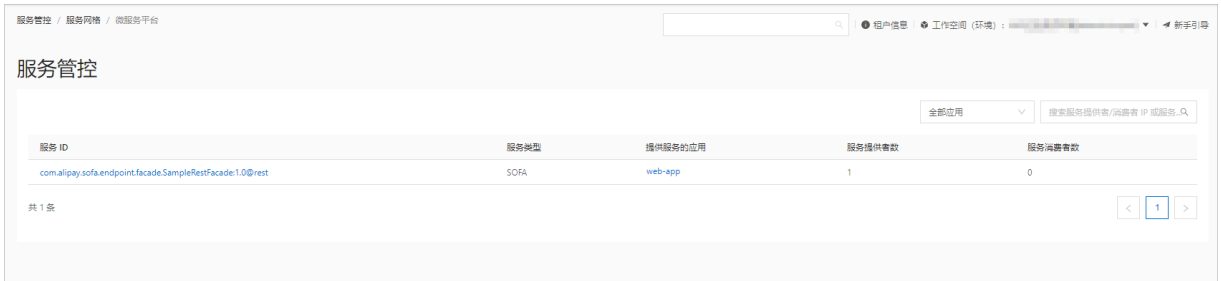
4.2.2.3. 服务管控和治理

将 SOFA、Dubbo、SpringCloud 类型的微服务发布成功后，您可以在服务网格控制台上进行服务管控和服务治理的操作。

服务管控

服务管控提供服务查询、服务消费者和提供者信息查询功能。更多信息请参见 服务管控。

在 SOFAStack 控制台，单击**中间件**下的 微服务平台 > **服务网格** > **服务管控** 进入服务管控页面。



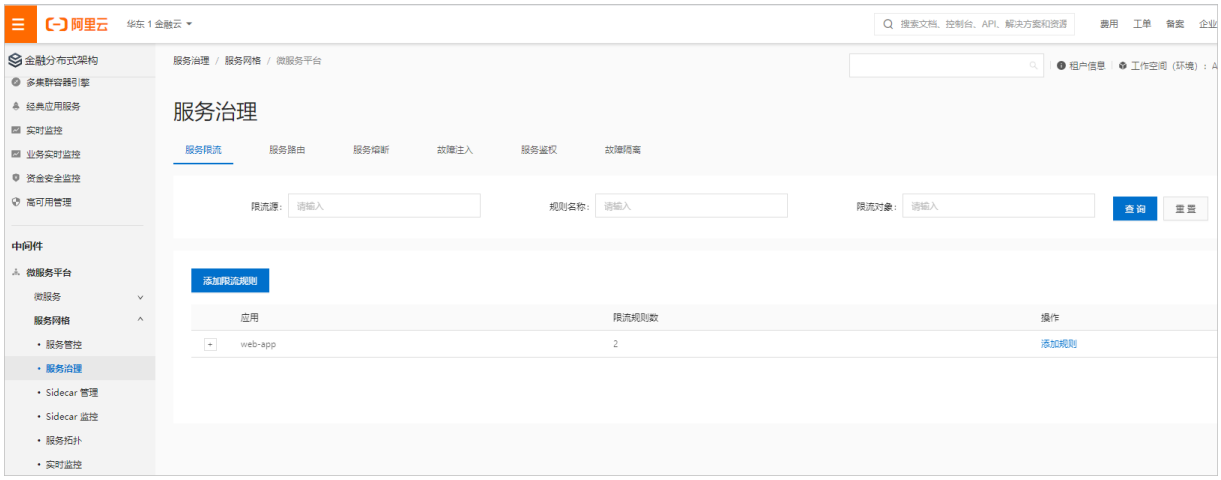
服务列表页可以查看服务 ID、服务类型、提供服务的应用、服务提供者数、服务消费者数。

单击服务 ID 进入服务详情页，服务详情页可以查看服务提供者信息和服务消费者信息。

- 服务提供者信息：IP、启用服务网格、集群类型、协议、端口、应用版本、权重、状态。
- 服务消费者信息：IP，应用。

服务治理

服务治理包含服务限流、服务路由、服务鉴权、服务熔断、故障注入和故障隔离相关服务治理。
在 SOFAStack 控制台，单击 **中间件** 下的 **微服务平台** > **服务网格** > **服务治理** 进入服务治理页面。



- 服务限流
  在高并发场景下，为保障系统稳定，提供 QPS 、最大并发数两种方式的限流。更多信息请参见 **服务限流**。
- 服务路由
  通过设置路由规则对服务提供方和服务消费方进行流量分配调整，常用于线下测试联调、灰度发布、蓝绿发布场景。更多信息请参见 **服务路由**。
- 服务鉴权
  通过设置黑名单规则、白名单规则的方式，对服务调用方进行鉴权控制。更多信息请参见 **服务鉴权**。
- 服务熔断
  当发现服务故障或异常时，对故障服务进行熔断操作，防止雪崩，使故障处于可控范围。更多信息请参见 **服务熔断**。
- 故障注入
  为检测系统对异常的应对能力，人为注入故障观察系统情况，提高和优化系统抵御风险的能力，目前提供了中断异常和超时异常两种类型。更多信息请参见 **故障注入**。
- 故障隔离
  是对故障进行实例级别的隔离，使故障的影响范围更小。更多信息请参见 **故障隔离**。

4.2.3. SDK 版本说明

本文提供了服务网格 sofa-registry-cloud-all SDK 的版本信息，包含其发布时间、更新点等，以便您按需获取适用的 SDK。

应用服务接入 SOFA 注册中心，只需在 endpoint 下的 `pom.xml` 文件中引入以下依赖：

```
<!-- 需将 x.x.x 替换为 SDK 具体的版本号 -->
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>sofa-registry-cloud-all</artifactId>
  <version>x.x.x</version>
</dependency>
```

1.2.8 (2020-03-25)

新增

- 支持取消代理接口不走 Service Mesh。
- 支持 Dubbo 框架非 Dubbo 协议注册与发现。
- 支持服务鉴权，透传调用方应用名。

修复

服务多次暴露场景下，错误使用 path 构建 dataId 导致寻址错误和 RPC 调用失败问题。

1.2.7 (2020-03-16)

> **说明**：不推荐使用此版本。

新增

支持 Dubbo 泛化调用。

修复

- 同一个接口多次暴露或者消费构建 dataId 错误，导致获取不到服务提供者问题。
- 通过 API 创建注册中心时，触发多次向 MOSN 注册 appInfo，导致部分服务或者订阅数据丢失问题。

1.2.6 (2020-03-05)

> **说明**：不推荐使用此版本。

修复

- 1.2.5 版本中 Gateway 调用依赖时的 `global filter class not found` 错误。
- 商业版 MOSN 版本小于等于 1.2.4 中，服务推送 path 可能不正确的问题。支持 Dubbo 分组版本调用。

1.2.5 (2020-03-03)

> **说明**：不推荐使用此版本。

新增

支持 Spring Cloud Gateway 服务调用接入 Mesh。

更新

- 升级 fastjson 版本至 1.2.66（修复安全漏洞版本）。
- 提升测试覆盖率至 75.94%。
- 要求的 JDK 版本为 JDK6 及以上。

修复

SDK 输出 provider 信息错误，修正 log 输出级别。

1.2.4 (2020-02-19)

新增
支持 Dubbo 框架无缝接入 Servce Mesh。
更新
兼容 Dubbo 2.6.x 和 Dubbo 2.7.x。
修复
Mesh Registry 重复 Bean 定义导致 Spring Cloud 应用无法启动的问题。

1.2.3 (2020-01-02)

更新

- 优化 Spring Cloud 服务注册与订阅，减少与 MOSN 进行 HTTP 请求交互。
- 优化第一次与 MOSN 订阅无数据时，触发重试订阅数据。
- 优化 SDK 与 Eureka 兼容性，引入 SDK 无需排除 Eureka。
- 更改 LOG JAR 包依赖为 TEST，防止和业务日志框架包冲突。

4.2.4. 连接 SOFA 服务注册中心

在使用服务网格之前，您需要将本地工程接入 SOFA 服务注册中心，完成服务注册。本文介绍如何把微服务接入 SOFA 服务注册中心。
下面根据微服务类型，详细介绍以下三种微服务如何接入 SOFA 服务注册中心（即 SOFARegistry），完成服务注册。

SOFABoot 服务

1. 升级 SOFABoot 依赖版本至最新版本。
   版本信息，请参见 SOFABoot 版本说明。

```
<parent>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>sofaboot-enterprise-dependencies</artifactId>
  <version>3.x.x</version>
</parent>
```

2. 通过以下任一方式，添加应用启动参数：
   - 在配置文件中增加配置：
     在 `app/web/src/main/resources/config/application.properties` 文件中新增如下配置：

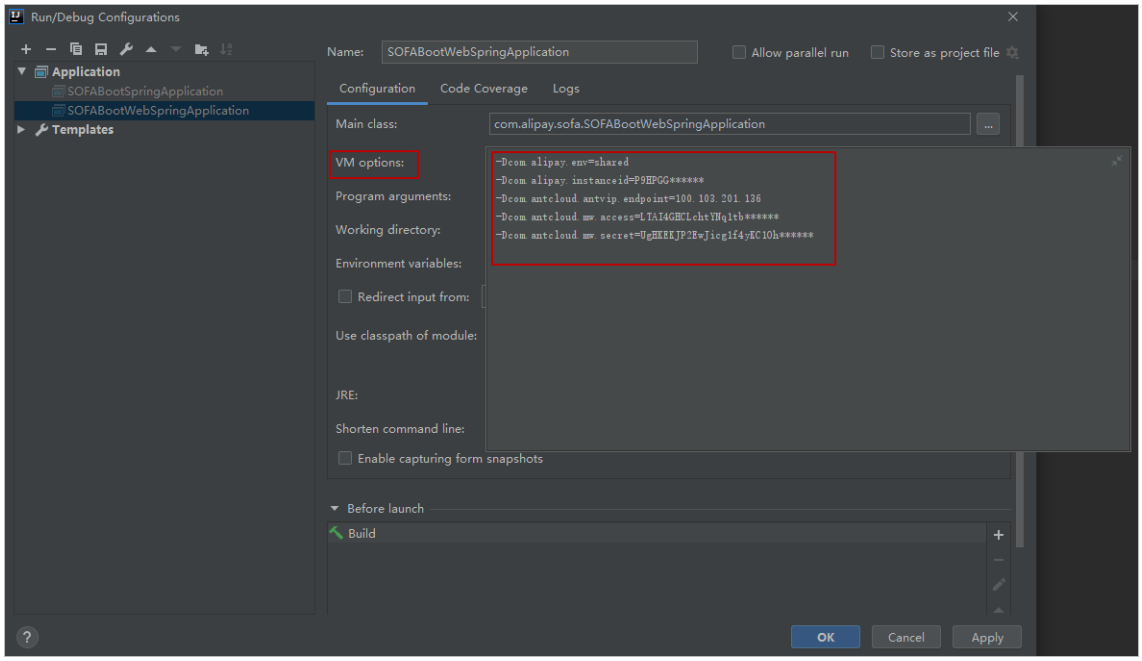
```
com.alipay.env=shared
com.alipay.instanceid=          //填写实例 ID。
com.antcloud.antvip.endpoint=   //填写环境的 AntVIP 地址。
com.antcloud.mw.access=         //填写账号的 AccessKey ID。
com.antcloud.mw.secret=         //填写账号的 AccessKey Secret。
```

     > **说明**
     > 以上参数值可在 脚手架控制台 获取，详情请参见 配置项说明。

   - 指定 JVM 启动参数：

     > **说明**
     > 这种方式优先级高于代码。

     在 Intellij IDEA 中的配置示例如下截图：




```
-Dcom.alipay.env=shared
-Dcom.alipay.instanceid=          //填写实例 ID。
-Dcom.antcloud.antvip.endpoint=   //填写环境的 AntVIP 地址。
-Dcom.antcloud.mw.access=         //填写账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=         //填写账号的 AccessKey Secret。
```

> **说明**
> 以上参数值可在 脚手架控制台 获取，详情请参见 配置项说明。

Dubbo 服务

1. 引入 `sofa-registry-cloud-all` SDK 依赖。
   版本信息，请参见 SDK 版本说明。

```
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>sofa-registry-cloud-all</artifactId>
  <!-- 替换 x.x.x 为该 SDK 最新版本号 -->
  <version>x.x.x</version>
</dependency>
```

2. 引入 Tracer 依赖。
   Tracer 针对 Dubbo 框架提供 3 种方式接入，分别针对 Spring Boot 1.x、Spring Boot 2.x 和非 Spring Boot 类型的 Dubbo 应用。请根据您的实际业务需求，选择相应的 Tracer 依赖接入。
   - **Spring Boot 1.x**

```
<!-- for Spring Boot 1.X -->
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
  <version>2.3.7.JST.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.alipay.sofa.common</groupId>
      <artifactId>sofa-common-tools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>tracer-enterprise-dst-plugin</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.alipay.sofa.common</groupId>
  <artifactId>sofa-common-tools</artifactId>
  <version>1.0.17</version>
</dependency>
```

- Spring Boot 2.x

```
<!-- for Spring Boot 2.X -->
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
  <version>3.2.3.JST.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.alipay.sofa.common</groupId>
      <artifactId>sofa-common-tools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>tracer-enterprise-dst-plugin</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.alipay.sofa.common</groupId>
  <artifactId>sofa-common-tools</artifactId>
  <version>1.0.17</version>
</dependency>
```

- 非 Spring Boot

  Tracer 依赖信息和与上文 Spring boot 2.x 相同。引入后，还需要在代码 Main 启动入口第一行加入以下开关：

```
SofaTracerConfiguration.setProperty(SofaTracerConfiguration.JSON_FORMAT_OUTPUT,"false");
```

3. 配置 Dubbo 的注册中心，使用 dsr：`<dubbo:registry address="dsr://dsr"/>` 。

4. 通过以下任一方式，添加应用启动参数：
   - 在 dubbo.properties 中配置如下：

```
com.alipay.instanceid=            //填写实例 ID。
com.antcloud.antvip.endpoint=   //填写环境的 AntVIP 地址。
com.antcloud.mw.access=         //填写账号的 AccessKey ID。
com.antcloud.mw.secret=         //填写账号的 AccessKey Secret。
```

   - 指定 JVM 启动参数：

```
-Dcom.alipay.env=shared
-Dcom.alipay.instanceid=          //填写实例 ID。
-Dcom.antcloud.antvip.endpoint=  //填写环境的 AntVIP 地址。
-Dcom.antcloud.mw.access=        //填写账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=        //填写账号的 AccessKey Secret。
```

   - 指定系统环境变量：

```
SOFA_INSTANCE_ID=          // 填写实例 ID。
SOFA_ANTVIP_ENDPOINT=    //填写环境的 AntVIP 地址。
SOFA_ACCESS_KEY=          //填写账号的 AccessKey ID。
SOFA_SECRET_KEY=          //填写账号的 AccessKey Secret。
```

> **说明**
> 以上参数值可在 脚手架控制台 获取，详情请参见 配置项说明。

Spring Cloud 服务

1. 引入以下 sofa-registry-cloud-all SDK 依赖。
   版本信息，请参见 SDK 版本说明。

```
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>sofa-registry-cloud-all</artifactId>
  <!-- 替换 x.x.x 为该 SDK 最新版本号 -->
  <version>x.x.x</version>
</dependency>
```

2. 根据您的 Spring Cloud 版本信息，引入对应的 Tracer 依赖。
   - **Camden、Dalston 和 Edgware 版本**（对应 Spring Boot 1.x 版本）

```
<!-- 支持服务调用tracer日志记录能力 -->
<!-- for Spring Boot 1.X -->
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
  <version>2.3.7.JST.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.alipay.sofa.common</groupId>
      <artifactId>sofa-common-tools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>tracer-enterprise-dst-plugin</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.alipay.sofa.common</groupId>
  <artifactId>sofa-common-tools</artifactId>
  <version>1.0.17</version>
</dependency>
```

- **Finchley、Greenwich 版本**（对应 Spring Boot 2.x 版本）

```
<!-- 接入tracer -->
<!-- for Spring Boot 2.X -->
<dependency>
  <groupId>com.alipay.sofa</groupId>
  <artifactId>tracer-enterprise-sofa-boot-starter</artifactId>
  <version>3.2.3.JST.1</version>
  <exclusions>
    <exclusion>
      <groupId>com.alipay.sofa.common</groupId>
      <artifactId>sofa-common-tools</artifactId>
    </exclusion>
    <exclusion>
      <groupId>com.alipay.sofa</groupId>
      <artifactId>tracer-enterprise-dst-plugin</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>com.alipay.sofa.common</groupId>
  <artifactId>sofa-common-tools</artifactId>
  <version>1.0.17</version>
</dependency>
```

3. 通过以下任一方式，添加应用启动参数。

> **注意**
> 如您即将使用的应用服务发布平台不是 SOFAStack 提供的容器应用服务或经典应用服务，则无需添加以下参数配置。

- 在应用 YAML 文件中指定以下参数：

```
sofa:
registry:
 discovery:
    instanceId:     //填写实例 ID。
    antcloudVip:    //填写环境的 AntVIP 地址。
    accessKey:      //填写账号的 AccessKey ID。
    secretKey:      //填写账号的 AccessKey Secret。
```

- 指定 JVM 启动参数：

```
-Dcom.alipay.instanceid=            //填写实例 ID。
-Dcom.antcloud.antvip.endpoint=   //填写环境的 AntVIP 地址。
-Dcom.antcloud.mw.access=         //填写账号的 AccessKey ID。
-Dcom.antcloud.mw.secret=         //填写账号的 AccessKey Secret。
```

- 指定系统环境变量：

```
SOFA_INSTANCE_ID=          //填写实例 ID。
SOFA_ANTVIP_ENDPOINT=    //填写环境的 AntVIP 地址。
SOFA_SECRET_KEY=           //填写账号的 AccessKey ID。
SOFA_ACCESS_KEY=           //填写账号的 AccessKey Secret。
```

> **说明**
> 以上参数值可在 脚手架控制台 获取，详情请参见 配置项说明。

本地应用改造完成后，您即可将该应用发布部署至 **容器应用服务** 或 **经典应用服务**（仅专有云）。部署步骤详情请参见 服务发布。

4.2.5. 控制台用户指南

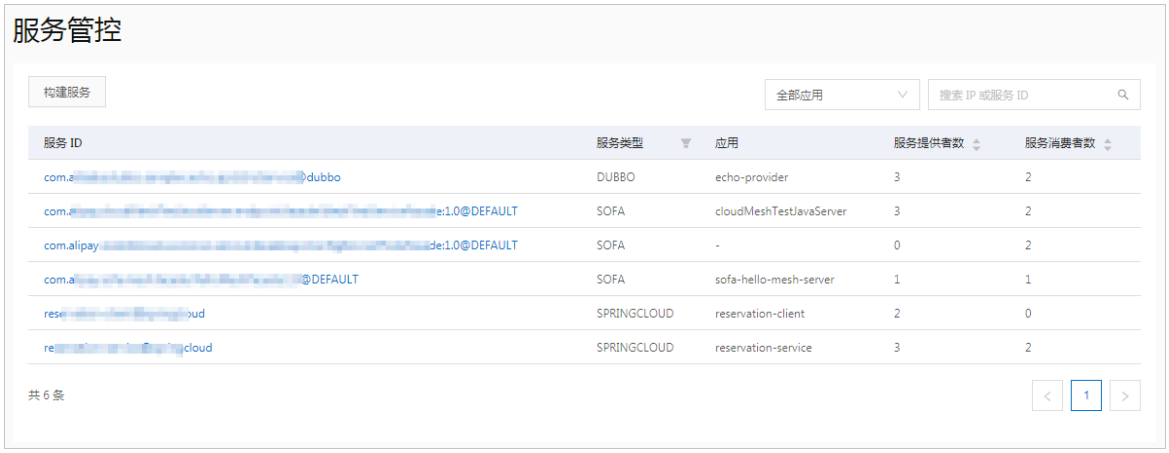
4.2.5.1. 服务管控

您可以在控制台查看当前工作空间下所有在容器服务发布并启用了服务网格的应用服务，包括查看服务详细信息、服务提供者信息、服务消费者信息等。

操作步骤

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 服务管控。**
4. 在左侧菜单栏选择 **服务网格 > 服务管控。**

5. 在服务管控页面，查看当前工作空间下所有在容器服务发布并启用了服务网格的应用服务。



您可以查看服务 ID、服务类型、应用、服务提供者数、服务消费者数信息。

6. 单击目标服务 ID，查看目标服务的详细信息。
您可以查看服务的基本信息、服务提供者信息、服务消费者信息。

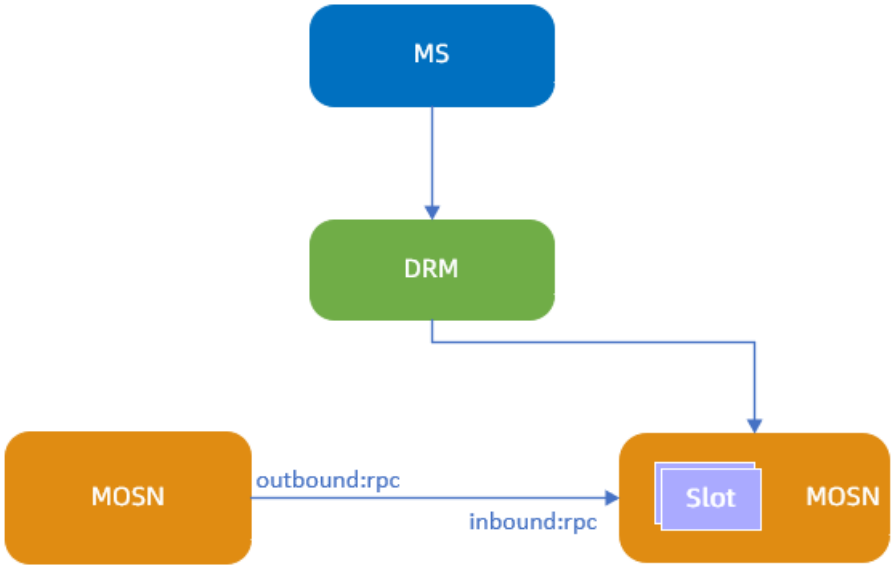
- 基本信息：查看服务的服务 ID、服务类型、应用。
- 服务提供者：查看服务提供者的 IP、是否启用服务网格、集群类型、协议、端口、应用版本、权重和状态信息。您可以在此处启用、禁用服务或修改服务权重。
- 服务消费者：查看服务消费者的 IP 与应用名。

4.2.5.2. 服务限流

在高并发场景下，为保证在现有资源条件下服务正常运行，使用服务限流让请求和并发在应用可接受的范围内，达到高可用的目的。服务网格的限流通过 MOSN 实现，支持统一的限流模型，对用户透明，并且在产品层增加了 trafficCondition 的流量匹配规则。

限流原理

服务限流生效流程如下：



1. 服务限流通过微服务管控台 MS 下发限流配置到分布式配置中心 DRM。
2. DRM 将限流的动态配置下发到 MOSN。
   限流规则在客户端或服务端生效，以下流程以客户端限流为例。
3. 客户端应用进行访问时，MOSN 会监控应用流量，如果触发限流规则，应用流量即被限制。

注意事项

- 您可以添加多条限流规则，若存在服务名和匹配条件相同，但限速不同的限流规则时，生效限速更小的限流规则。
- 请根据您的实际环境合理配置限速规则，不合理的限速规则可能导致应用访问异常。

添加限流规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 服务治理**，然后单击 **服务限流** 页签。
4. 在左侧菜单栏选择 **服务网格 > 服务治理**，然后单击 **服务限流** 页签。
5. 单击 **添加限流规则**，然后配置以下参数：

参数	说明
规则名称	配置服务限流的规则名称。
应用	配置应用名称。星号（*）表示所有应用。 需填写客户端的应用

参数	说明
限流方向	选择流量限制的方向，可选值为： ○ **服务端限流**：规则在服务端生效。 选择此项时，**应用** 需填写服务端的应用，**服务** 和 **方法** 填写服务端的服务和方法。 ○ **客户端限流**：规则在客户端生效。 选择此项时，**应用** 需填写客户端的应用；**服务** 填写客户端需要调用的服务； **方法** 填写客户端需要调用服务端服务的方法。 目前仅支持客户端限流，规则在客户端生效。**服务** 填写客户端需要调用的服务； **方法** 填写客户端需要调用服务端服务的方法。
服务	根据限流方向设置限流的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。 配置为客户端的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
方法	配置限流的方法。星号（*）表示所有方法。 需填写客户端的方法。
限流类型	选择限流的类型，可选值为： ○ **QPS**：单位时间内允许的最大请求数。 ○ **最大并发数**：同一时间内允许的最大访问线程数。
运行模式	设置限流规则的运行方式，可选值为： ○ 拦截模式：限流规则生效。 ○ 观察者模式：限流规则不生效，仅在 MOSN 中打印限流日志。
限流算法	使用令牌桶（Token Bucket）算法进行限流。仅在 **限流类型** 为 **QPS** 时设置。 系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。详情请参见 令牌桶算法。
令牌桶系数	设置令牌桶的系数，以决定令牌桶的最大值。默认为 1。 令牌数最大值 =（限流阈值 ÷ 单位时间）× 令牌桶系数
单位时间 ms	设置限速规则的单位时间，单位为 ms。仅在 **限流类型** 为 **QPS** 时设置。
限流阈值	根据选择的限流类型设置限流阈值： ○ **限流类型** 为 **QPS** 时：表示单位时间内允许通过最大请求数的值，超过该数值的请求会被限流。 ○ **限流类型** 为 **最大并发数** 时：表示同一时间内允许连接的最大访问线程数，超过该阈值的线程会被限流。

参数	说明
流量精确匹配	设置流量的匹配条件，满足匹配条件的流量才会使用限流规则。置空此项时表示匹配所有流量。 您可以配置多条匹配条件，多个条件是与的关系，按顺序进行匹配。参数配置如下： ○ **字段**：可选择系统字段和请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ 请求头：请求头是指协议的请求头，比如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。

6. 单击 **提交**，然后单击 **确定**。
7. 在限流规则列表中，将刚刚创建的限流规则的状态改为 **开**。

编辑限流规则

您可以随时编辑已创建的限流规则，编辑规则会在提交后实时生效。

1. 在 **服务限流** 页签，单击目标限流规则右侧的 **编辑**。
2. 按需求编辑限流规则后，单击 **提交**。

删除限流规则

您可以删除已创建的限流规则，删除规则的操作会实时生效，请谨慎操作。

1. 在 **服务限流** 页签，单击目标限流规则右侧的 **编辑**。
2. 单击 **确定**。

4.2.5.3. 服务路由

当服务消费者面临多个服务提供者时，需要通过路由规则来确定具体的服务提供者。服务路由功能提供了灵活的路由功能，允许您定义多条服务路由规则，可以帮助您解决多个场景下的难题。

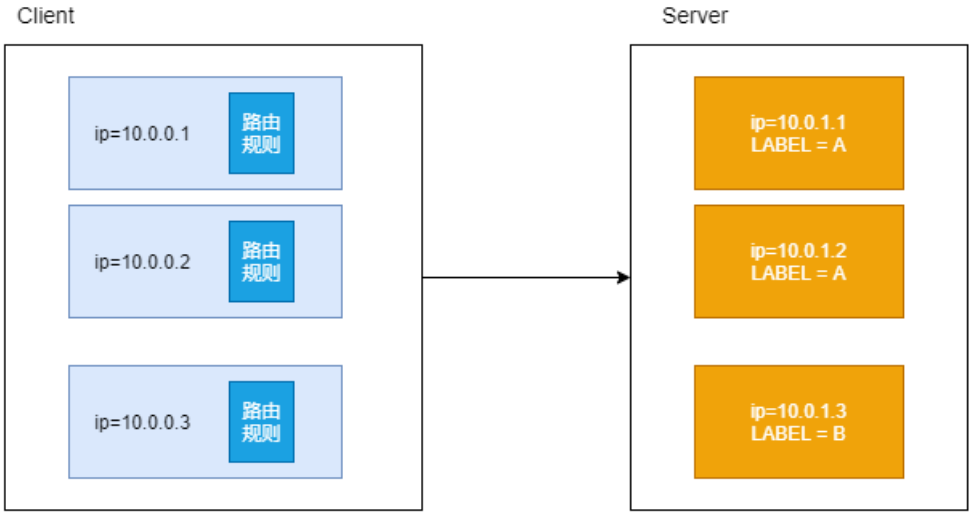
功能简介

服务路由多用于灰度引流、蓝绿发布场景，将因版本升级造成的如下问题影响降到最低。

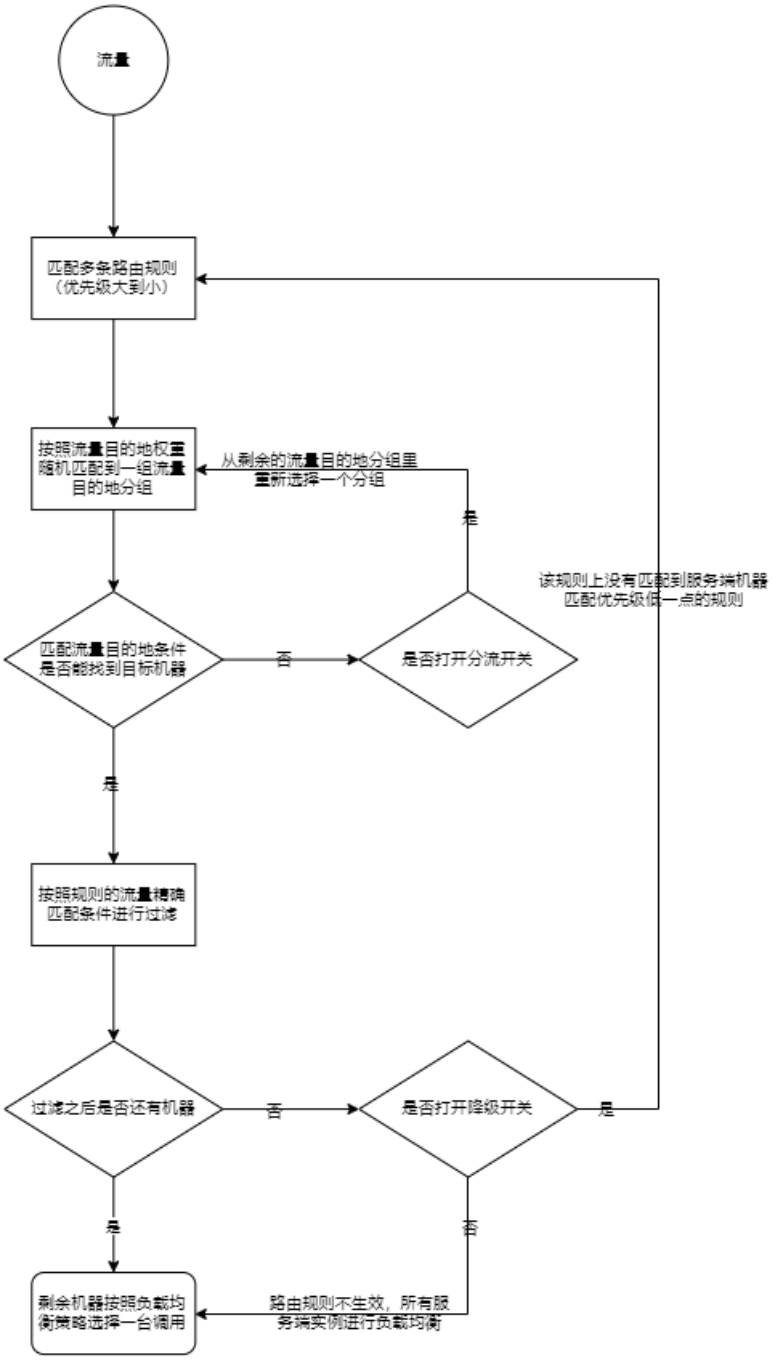
- 线下测试联调问题。
- 蓝绿发布问题。
- 灰度引流问题。

> **说明**
> 当前版本仅支持 SOFA 和 Dubbo 服务的路由，暂不支持 Spring Cloud 服务的路由。

配置的路由规则实际在服务消费方（即客户端）生效，您可以配置服务路由过滤服务提供方。目前主要有 RPC-Client（经典微服务生效）和 MOSN（服务网格生效）两种执行实体。
路由模型服务端在部署时，可以在机器上加固定格式的标签，比如 LABEL=A。



路由规则生效流程流程如下图所示：



添加路由规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件** > **微服务平台** > **服务网格** > **服务治理**，然后单击 **服务路由** 页签。
4. 在左侧菜单栏选择 **服务网格** > **服务治理**，然后单击 **服务路由** 页签。
5. 单击 **添加服务路由**，然后配置以下参数：

参数	说明
基本信息	
规则组名称	配置路由规则组名称。
应用	配置服务端的应用名称。
服务	配置应用包含的服务。 单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
规则属性	
规则名称	配置路由规则名称。
优先级	配置路由的优先级，数字越大优先级越高。 同时存在多条路由时，按照路由优先级大小进行匹配。
规则开关	配置是否开启该条路由规则。默认开启。
分流开关	当流量目的地分组没有服务实例时，是否转发到其他分组。 ○ 开启后，流量在高权重分组中未匹配到服务实例，会继续去匹配其他分组。 ○ 关闭后，流量不会匹配其他分组。
降级开关	当整条规则都匹配不到服务实例时，是否继续匹配其他规则。 ○ 开启后，流量未在当前规则中匹配到服务实例时，会继续去匹配优先级更低的路由规则。 ○ 关闭后，流量不会匹配其他路由规则。
流量精确匹配（可选）	通过流量匹配条件来过滤需要执行路由规则的请求，满足这些条件的流量才会使用这条规则，不满足这条规则的就走降级开关逻辑，往低优先级的路由规则去匹配。不填表示匹配所有流量。 多个条件按照顺序执行，直到被某一个规则被拦截或全部通过。规则主要包括下述内容： ○ **字段**：包括系统字段、请求头、链路。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ 请求头：请求头是指协议的请求头，比如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ■ 链路：即 Trace 信息，通过代码里配置文件获取。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。
流量目的地	

参数	说明
权重	设置当前流量目的地分组的权重，多个流量目的地的权重之和为 100%。 您可以单击最下方的 **添加** 按钮，配置多个流量目的地分组。
服务实例分组条件	配置服务实例分组条件，多个条件之间是与的关系。 ○ 字段名：可从下拉列表选择，或者自定义输入。 您还可以配置自定义 label，配置示例见下表的 label 配置示例。 ○ 逻辑：等于。 ○ 字段值：配置字段名对应的值。 字段名是您自定义的 label 时，此处填写 label 对应的 value。 您可以单击下方的 **添加** 按钮，配置多个分组条件。

label 配置示例：

6. 单击 **提交**。
7. 在服务路由列表中，将刚创建的熔断规则的状态改为 **开启**。

> **注意**
> 您可以添加多条路由规则，相同服务名称的路由规则会被合并。

编辑路由规则

您可以随时编辑已创建的路由规则，规则提交后实时生效。

1. 在 **服务路由** 页签，单击目标路由规则右侧的 **编辑**。
2. 按需求编辑路由规则后，单击 **提交**。

删除路由规则

您可以删除已创建的路由规则，删除规则会实时生效，请谨慎操作。

1. 在 **服务路由** 页签，单击目标路由规则右侧的 **删除**。
2. 单击 **确定**。

4.2.5.4. 服务熔断

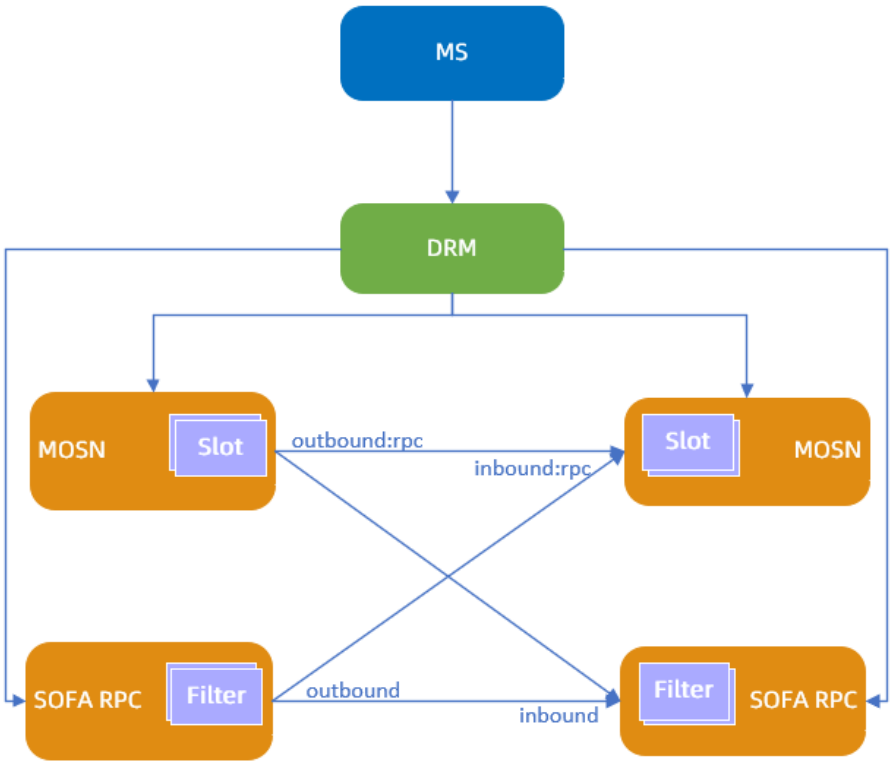
当为服务中的服务端接口不稳定，出现频繁超时或错误时，可能会引起服务调用雪崩。您可以对应用开启服务熔断功能，使有故障的服务端及时返回错误，并释放系统资源，提高用户体验和系统性能。

功能简介

您可以通过下述操作让故障处于可控范围：

- 通过监控或者服务拓扑查看到某个服务延时较大、错误率较多后，进行服务治理。
- 选择异常服务后，进行服务熔断自动设置，可自定义熔断条件，例如：在某个时间段内（例如 10s）请求数达到某个值，且错误率或者延时达到某个值。

满足条件，则可以触发熔断，还可以持续测试该服务，进行自动熔断恢复。服务熔断的规则发布流程如下：



1. 通过微服务管控台 MS 下发熔断的动态配置到分布式配置中心 DRM。
2. DRM 将熔断的动态配置下发到 MOSN。
3. MOSN 根据平均 RT 和错误率触发熔断。
   熔断可在客户端或服务端生效。

创建熔断规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 服务治理**，然后单击 **服务熔断** 页签。
4. 在左侧菜单栏选择 **服务网格 > 服务治理**，然后单击 **服务熔断** 页签。
5. 单击 **添加熔断组规则**，然后配置以下参数：

参数	说明
规则名称	配置服务熔断规则的名称。 支持中文、英文、数字、下划线。
应用	配置检测的目标应用。 您可以手动填写或在下拉列表中选择。

参数	说明
熔断方向	配置服务熔断的方向，取值如下： ○ **客户端熔断**：表示熔断规则在客户端生效。在客户端侧统计调用服务端的请求指标，应用填写客户端的应用，服务和方法填写引用的服务端的服务和方法。 ○ **服务端熔断**：表示熔断规则在服务端生效，每个服务端单独统计请求指标，应用填写服务端的应用，服务和方法填写服务端的服务和方法。
熔断服务	配置熔断的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
方法	配置需要熔断的服务的方法名。
熔断类型	配置熔断的类型，取值如下： ○ **错误比率**：熔断时间窗口内的请求错误率大于错误率阈值时，触发熔断。 ○ **平均 RT**：熔断时间窗口内的请求数阈值大于平均 RT 阈值时，触发熔断。
运行模式	配置熔断的运行模式，取值如下： ○ **拦截模式**：满足条件的请求会触发熔断规则。 ○ **观察者模式**：熔断规则不会被触发，只会在 MOSN 里打印日志。
熔断配置	配置触发熔断的各项参数： ○ **熔断指标窗口**：和请求数阈值搭配使用，表示熔断指标窗口内至少有请求数阈值的请求就会纳入熔断指标统计。单位为秒。 ○ **请求数阈值**：和熔断指标窗口搭配使用。单位为个。 ○ **错误率阈值**：请求数的平均错误率达到此阈值时触发熔断。熔断类型配置为 **错误比率** 时配置。 ○ **平均 RT 阈值**：请求响应时间超过此阈值时触发熔断。单位为毫秒。熔断类型配置为 **平均 RT** 时配置。 ○ **熔断时间段**：触发熔断后的熔断时长，熔断时长之内的请求会直接返回错误。当熔断超过该时间后会再次尝试调用服务端，如果还是失败则继续触发熔断。 ○ **RPC 超时时间**：经典微服务场景会使用。单位为毫秒。熔断类型配置为 **错误比率** 时配置。

参数	说明
流量精确匹配（可选）	配置之后，只有满足匹配条件的流量才会使用这份熔断规则。置空表示匹配所有流量。 您可以配置多条匹配条件，多个条件是与的关系，按顺序进行匹配。参数配置如下： ○ **字段**：可选择系统字段和请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ 请求头：请求头是指协议的请求头，比如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。

6. 单击 **提交**，然后单击 **确定**。
7. 在熔断规则列表中，将刚刚创建的熔断规则的状态改为 **开**。

验证熔断规则

要验证服务熔断，需要在服务端侧设置接口的抛错或者超时。您可以在服务端 MOSN 容器中执行以下命令：

- 错误比率

```
kubectl exec -it dubbo-echo-server-v1-6c558677d9-fzdmz -c mosn-sidecar-container bash

curl localhost:8080/status/set?throwException=true&sleep=0
## 或者 curl localhost:8080/status/set?throwException=false&sleep=2000
```

```
[root@d                                    5f5-sthcb /tmp/pip-18.0]
#curl localhost:8082/hi?count=10
总请求数 : 10
总响应耗时 : 21 毫秒
总平均耗时 : 2.1 毫秒
单个请求平均耗时 : 1.0 毫秒
QPS : 476
Thread-134
第 1次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: mosn limit excepiton
第 2次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: Hi  dubbo! 我是[17          26]
第 3次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 4次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: Hi  dubbo! 我是[17          26]
第 5次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 6次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: Hi  dubbo! 我是[17          26]
第 7次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 8次调用, sleep 0 毫秒, 耗时 1 毫秒, 结果: Hi  dubbo! 我是[17          26]
第 9次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 10次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: Hi  dubbo! 我是[17          26]
```

- 平均 RT 熔断

```
kubectl exec -it dubbo-echo-server-v1-6c558677d9-fzdmz -c mosn-sidecar-container bash
curl localhost:8080/status/set?throwException=false&sleep=500
```

进入客户端容器，多次调用客户端接口：

```
kubectl exec -it dubbo-echo-client-547c6f65f5-sthcb -c mosn-sidecar-container bash

## 访问 8082 端口的 hi 接口
curl localhost:8082/hi?count=10
```

```
[root@d                                    5f5-sthcb /tmp/pip-18.0]
#curl localhost:8082/hi?count=10
总请求数 : 10
总响应耗时 : 2034 毫秒
总平均耗时 : 203.4 毫秒
单个请求平均耗时 : 202.3 毫秒
QPS : 4
Thread-92
第 1次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 2次调用, sleep 0 毫秒, 耗时 1007 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 3次调用, sleep 0 毫秒, 耗时 1005 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 4次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: mosn limit excepiton
第 5次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 6次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 7次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 8次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
第 9次调用, sleep 0 毫秒, 耗时 3 毫秒, 结果: Hi  dubbo! 我是[17                  26]
第 10次调用, sleep 0 毫秒, 耗时 2 毫秒, 结果: mosn limit excepiton
```

编辑熔断规则

您可以随时编辑已创建的熔断规则，规则提交后会实时生效。

1. 在 **服务熔断** 页签，单击目标熔断规则右侧的 **编辑**。
2. 按需求编辑熔断规则后，单击 **提交**。

删除熔断规则

您可以删除已创建的熔断规则，删除操作会实时生效，请谨慎操作。

1. 在 **服务熔断** 页签，单击目标熔断规则右侧的 **删除**。
2. 单击 **确定**。

4.2.5.5. 故障注入

您可以通过故障注入功能向测试应用注入故障，检测应用面对异常时的处理情况。您可以根据检测的情况调整您的应用，以减少应用在正式使用时出现的异常问题。多用于测试环境。

配置故障注入规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件** > **微服务平台** > **服务网格** > **服务治理**，然后单击 **故障注入** 页签。
4. 在左侧菜单栏选择 **服务网格** > **服务治理**，然后单击 **故障注入** 页签。
5. 单击 **添加注入组规则**，然后配置以下参数：

参数	说明
规则名称	设置故障注入规则的名称。 最多不超过 255 个字符。

参数	说明
应用	选择或填写目标应用的名称。星号（*）表示所有应用。
注入方向	设置故障注入的方向，可选值为： ○ **服务端注入**：在应用的服务端注入故障。 ○ **客户端注入**：在应用的客户端注入故障。
服务	配置注入故障的服务。单击 **切换输入模式** 可在手动填写与下拉选择之间切换。
方法	配置故障注入的方法。星号（*）表示所有方法。
运行模式	配置故障注入规则的运行模式，取值如下： ○ **拦截模式**：满足条件的故障注入请求会被注入。 ○ **观察者模式**：满足条件的故障注入请求不会被注入，只会在 MOSN 里打印日志。
故障类型	故障注入支持注入错误或者超时等事件，方便服务的异常测试，用于模拟服务异常的情况。取值如下： ○ **中断异常**：注入运行时异常，中断请求并返回既定的错误状态码。 ○ **超时异常**：满足条件的请求，会增加响应时间，请求正常调用。
异常比例	设置注入异常流量的比例。例如设置为 80，则只注入 80% 的异常流量。
流量精确匹配（可选）	设置流量的匹配条件，满足匹配条件的流量才会使用故障注入规则。置空此项时表示匹配所有流量。 您可以配置多条匹配条件，多个条件是与的关系，按顺序进行匹配。参数配置如下： ○ **字段**：可选择系统字段和请求头。 ○ **字段名**：根据字段类型有不同的值。 ■ **系统字段**：包括流量类型、调用方应用名、调用方 IP、服务方应用名。 ■ **请求头**：请求头是指协议的请求头。例如 Dubbo 协议取的是 attachment，HTTP 协议取的是 Request Header。用户可以在应用系统中自定义请求头参数和值。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：字段名对应的值。

6. 单击 **提交**，然后单击 **确定**。
7. 在故障注入规则列表中，将刚刚创建的故障注入规则的状态改为 **开启**。

> **说明**
> 当多条故障注入规则针对同一个服务时，只会生效第一条。

编辑故障注入规则

您可以随时编辑已创建的故障注入规则，规则提交后实时生效。

1. 在 **故障注入** 页签，单击目标故障注入规则右侧的 **编辑**。
2. 按需求编辑故障注入规则后，单击 **提交**。

删除故障注入规则

您可以删除已创建的故障注入规则，删除操作实时生效，请谨慎操作。

1. 在 **故障注入** 页签，单击目标故障注入规则右侧的 **删除**。
2. 单击 **确定**。

4.2.5.6. 服务鉴权

服务提供者提供服务后，您可以通过服务鉴权功能对服务调用方进行鉴权。

注意事项

- 服务鉴权功能支持 SOFARPC、Dubbo、Spring Cloud 三种框架。其中，Dubbo 和 Spring Cloud 需要确保引入的 `sofa-registry-cloud-all` 依赖版本为 1.2.8 及以上版本，详情请参见 SDK 版本说明。
- 在使用容器应用服务发布应用时，应用名称必须与本地应用注册代码配置的 `spring.application.name` 一致。
- 请确保 SOFABoot 版本在 3.3.3 及以上。有关 SOFABoot 的版本信息，请参见 SOFABoot 版本说明。

添加鉴权规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 服务治理**，然后单击 **服务鉴权** 页签。
4. 在左侧菜单栏选择 **服务网格 > 服务治理**，然后单击 **服务鉴权** 页签。
5. 单击 **添加鉴权规则**，然后配置以下参数：

参数	说明
鉴权粒度	目前仅支持 **服务级**，可以对应用下的一个服务添加鉴权规则。 配置鉴权的粒度，可选值为： ○ **应用级**：对某个应用添加鉴权规则。 ○ **服务级**：对应用下的一个或多个服务添加鉴权规则。
规则名称	配置鉴权规则的名称。 仅支持中文、英文、数字、下划线，最多支持 255 个字符。
类型	配置鉴权类型，可选值为： ○ 白名单：选择此项时，匹配条件的请求会被放通。 ○ 黑名单：选择此项时，匹配条件的请求会被拒绝。
应用	选择或填写待鉴权的应用。星号（*）表示所有应用。 选择待鉴权的应用。
服务	选择或填写应用下的一个或多个服务。仅在 **鉴权粒度** 选择 **服务级** 时配置。

参数	说明
运行模式	配置服务鉴权规则的运行方式，取值如下： ○ **拦截模式**：鉴权规则生效，则拒绝请求。 ○ **观察者模式**：鉴权规则生效时，不拒绝请求，只打印日志。
匹配条件	配置鉴权规则的匹配条件，符合条件的流量会被鉴权。 可配置多条匹配规则，各匹配规则之间是与的关系。参数配置如下： ○ **字段**：可选择系统字段和自定义字段。 ○ **字段名**：根据字段类型有不同的值。 ■ 系统字段：可选择调用方应用名、调用方 IP、服务方应用名、服务方方法名。 ■ 自定义字段：根据实际需求自行设置字段名。 ○ **选择逻辑**：包括等于、不等于、属于、不属于、正则。 ○ **字段值**：填入所选字段的值。

6. 单击 **提交**，然后单击 **确定**。
7. 在鉴权规则列表中，将刚刚创建的鉴权规则的状态改为 **开**。
8. 根据设置的类型打开白名单或黑名单的开关。
   规则类型为白名单时，打开白名单开关；反之，打开黑名单开关。否则，鉴权规则不生效。

编辑鉴权规则

您可以随时编辑已创建的鉴权规则，规则提交后实时生效。

1. 在 **服务鉴权** 页签，单击目标鉴权规则右侧的 **编辑**。
2. 按需求编辑鉴权规则后，单击 **提交**。

删除鉴权规则

您可以删除已创建的鉴权规则，删除操作实时生效，请谨慎操作。

1. 在 **服务鉴权** 页签，将目标鉴权规则的状态改为 **关**。
2. 单击目标鉴权规则右侧的 **删除**。然后单击 **确定**。

导出鉴权规则

您可以将已配置的鉴权规则导出，以备份您的鉴权规则。

1. 在 **服务鉴权** 页签，选中需要导出的鉴权规则。
2. 单击 **批量导出**，然后单击 **确定**。

您也可以选择 **更多** > **全部导出**，一次性导出所有的鉴权规则。导出的鉴权规则会存放在浏览器默认的下载文件夹中。

导入鉴权规则

您可以导入备份的鉴权规则，以快速创建鉴权规则。

1. 在 **服务鉴权** 页签，选择 **更多** > **全部导入**。
2. 单击 **选择文件**，然后选中目标文件后，单击 **确定**。

4.2.5.7. 故障隔离

某个服务故障或者异常时，如果该服务触发熔断会造成整个服务的不可用。而故障隔离能够定位到异常的服务实例，实现实例级别精细化的隔离和摘流，使故障影响的范围更小、更可控。

配置故障隔离规则

1. 登录 SOFAStack 控制台。
2. 登录微服务控制台。
3. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 服务治理**，然后单击 **服务限流** 页签。
4. 在左侧菜单栏选择 **服务网格 > 服务治理**，然后单击 **服务限流** 页签。
5. 单击 **添加限流规则**，然后配置以下参数：

参数	说明
规则名称	配置故障隔离规则的名称。 最多支持 255 个字符。
应用	填写或选择客户端应用。
隔离方向	设置故障隔离的方向，可选值为： ○ 服务端隔离：从服务端侧发现有故障的服务器，并将之暂时隔离。 ○ 客户端隔离：从客户端侧发现有故障的服务器，并将之暂时隔离。
运行模式	配置故障隔离规则的运行方式，取值如下： ○ **拦截模式**：故障隔离规则生效，有故障的服务器会被暂时隔离。 ○ **监控模式**：有故障的服务器不会被隔离，只打印日志。
时间窗口大小	设置故障检测的时间，与 **时间窗口内最少调用次数** 配合，只有在指定时间段内进行指定次数的请求才会被采集。 取值范围：(1,60] 单位：秒
时间窗口内最少调用次数	设置指定时间内的最少调用请求次数。 取值 ≥ 0。
异常比例阈值（%）	设置异常请求的比例，配合 **异常比例倍数** 参数来确定是否隔离目标服务器。 当异常请求的比例 > **异常比例阈值** × **异常比例倍数** 时，系统认为目标服务器异常，将隔离目标服务器。 取值范围：(0,100]
异常比例倍数	设置异常比例的倍数，配合 **异常比例阈值** 参数来确定是否隔离目标服务器。 取值 ≥ 0。
最大隔离数量	设置服务器的隔离数量，即多台服务器故障时最多隔离几台。 取值 ≥ 0。

参数	说明
流量精确匹配（可选）	符合流量精确匹配的流量才进行隔离。 可配置多条匹配规则，各匹配规则之间是与的关系。参数配置如下： ○ 字段：选择 **系统字段**。 ○ 字段名：选择 **服务方应用名**。 ○ 逻辑：包括 **等于**、**不等于**、**属于**、**不属于**、**正则**。 ○ 字段值：填入所选字段的值。

6. 单击 **提交**，然后单击 **确定**。
7. 在故障隔离规则列表中，将刚刚创建的故障隔离规则的状态改为 **开**。

配置示例

某应用有 A、B、C 三台服务器，三台服务的状态如下：

- A：正常
- B：异常，异常率 20%
- C：异常，异常率 40%

已配置的故障隔离规则如下：

- 时间窗口大小：10
- 时间窗口内最少调用次数：20
- 异常比例阈值：20
- 异常比例倍数：1
- 最大隔离数量：1

如果 10 秒内有 20 个请求，且异常率 ≥ 20%（异常比例倍数 × 异常比例阈值），会触发故障隔离规则。因为 B 的故障率更高，系统会优先将 B 隔离。此时，若将最大隔离数量调整为 2，则 A、B 均会被剔除。

编辑故障隔离规则

您可以随时编辑已创建的故障隔离规则，规则提交后实时生效。

1. 在 **故障隔离** 页签，单击目标故障隔离规则右侧的 **编辑**。
2. 按需求编辑故障隔离规则后，单击 **提交**。

删除故障隔离规则

您可以删除已创建的故障隔离规则，删除操作会实时生效，请谨慎操作。

1. 在 **故障隔离** 页签，单击目标故障隔离规则右侧的 **删除**。
2. 单击 **确定**。

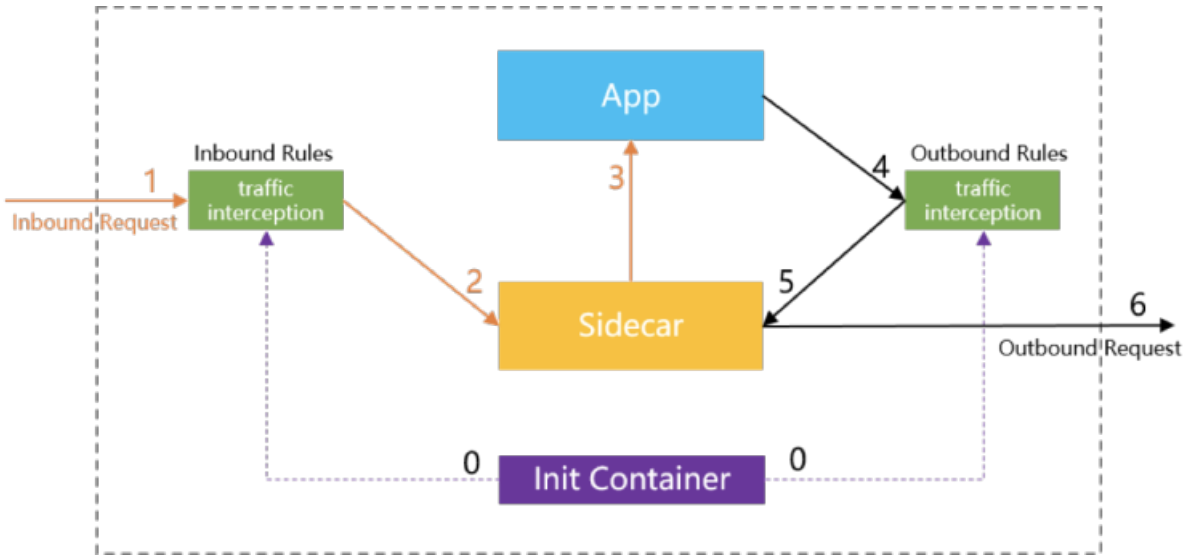
4.2.5.8. 透明劫持

通过透明劫持功能，应用无需改造即可获得服务治理和观测能力。应用开启透明劫持功能后，出入应用的业务流量将会被 Sidecar Proxy 自动拦截，继而按照您在控制台配置的规则进行观测与治理。

> **注意**
> 透明劫持当前仅适用于 HTTP 协议的服务端应用，后续版本将会同时适用于客户端和服务端，也将支持更多协议。

透明劫持流程

透明劫持流程图如下：



透明劫持流程说明如下：

1. Init 流程（0）：下方紫色虚线表示 Init 流程。
   当 Pod 启动时，通过 Init Container 特权容器开启流量劫持，并设置流量劫持规则（分为 Inbound 规则和 Outbound 规则）。Init Container 在设置结束后立即退出。
2. Inbound 流程（1、2、3）：左边橙色实线表示 Inbound 流程。
   i. 请求发给 APP 前被 traffic intercetion 劫持。
   ii. traffic intercetion 根据 Inbound 规则，将请求转发到 Sidecar。
   iii. Sidecar 将请求转发给 APP。
3. Outbound 流程（4、5、6）：右边黑色实线表示 Outbound 请求。
   i. App 发出的 Outbound 请求被 traffic interception 劫持。
   ii. traffic interception 根据 Outbound 规则将请求转发给 Sidecar。
   iii. Sidecar 处理之后将请求回复给请求者。

创建透明劫持规则

1. 添加应用。
   i. 登录 SOFAStack 控制台。
   ii. 在左侧菜单栏选择 **中间件 > 微服务平台 > 服务网格 > 透明劫持**。

iii. 单击 **添加应用**，然后配置以下参数：

参数	说明
应用名称	填写需进行透明劫持的目标应用名称。 仅支持英文、数字和短划线（-）。
添加端口	单击 **添加**，然后输入应用的端口号。目前仅支持 HTTP 协议。 您可以添加多个端口号。
添加节点	单击 **添加**，然后输入目标节点的 IP 地址，集群类型仅支持虚拟机。 您可以添加多个节点。

iv. 单击 **提交**。

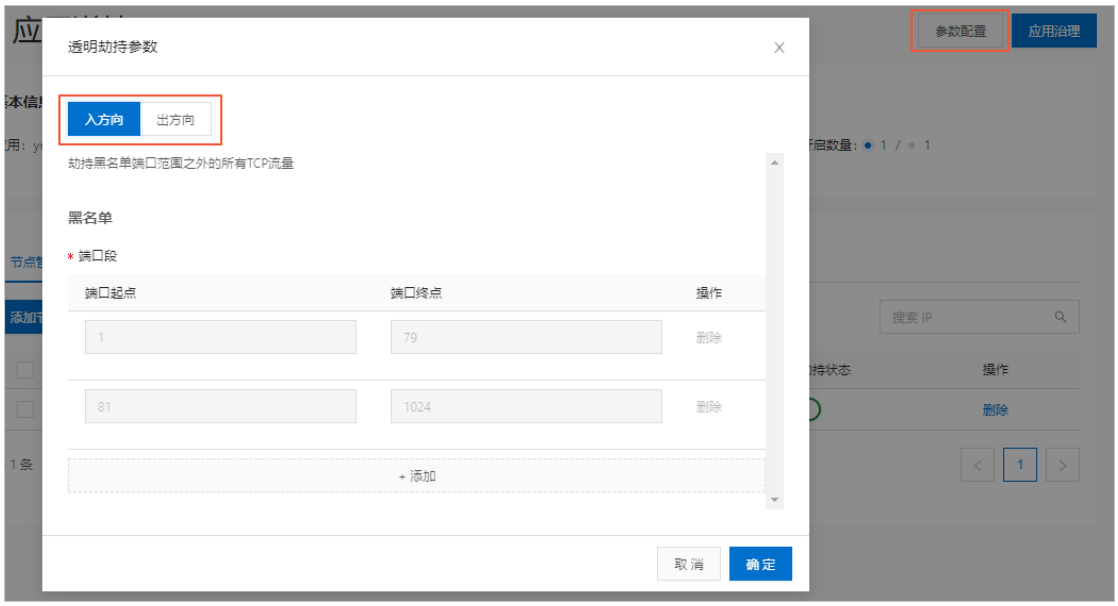
2. 开启透明劫持，并配置透明劫持参数。

i. 在透明劫持列表，单击目标应用名称。

ii. 在 **透明管理** 页签，打开目标节点的 **透明劫持状态** 开关。
若有多个节点，可选中目标节点，然后单击 **开启透明劫持**。

iii. 单击 确定。

iv. 返回透明劫持列表，单击目标应用右侧的 **参数配置**，然后配置透明劫持参数。



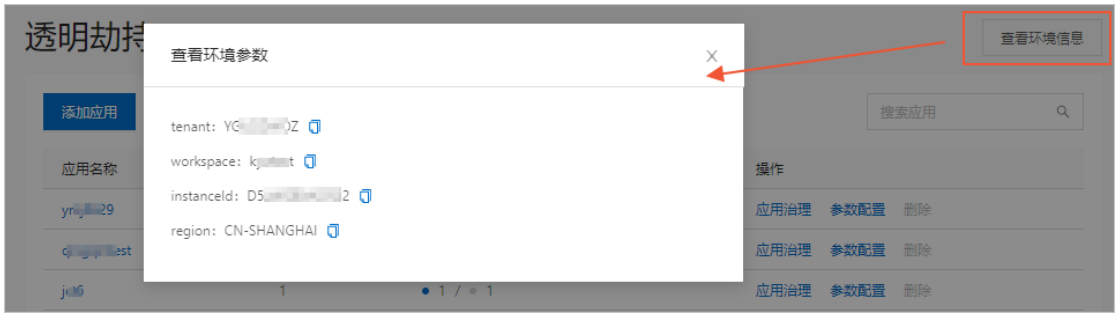
- **入方向**：在流量进入时，劫持黑名单中端口范围之外的所有TCP流量。
单击 **添加**，然后添加您无需劫持的端口范围。取值范围为[1025,65535]，默认劫持 80 和大于 1024 的端口。
- **出方向**：在流量流出时，劫持黑名单网段范围和端口范围之外的TCP流量。
在 **网段** 或 **端口段** 添加无需劫持的端口和网段范围。端口的取值范围与入方向一致。

v. 单击 **确定**。

3. 运行透明劫持脚本。
透明劫持开启并配置完成后，您需要下载并安装透明劫持脚本。

> 注意
>
> - 在控制台页面中开启或关闭透明劫持功能后，必须运行透明劫持脚本，以安装或卸载 Sidecar。
> - Sidecar 安装完成后，若修改了劫持参数配置，您必须重新运行脚本，进行 Sidecar 的卸载或安装。

i. 在 **透明劫持** 页面，单击右上角的 **查看环境信息**。



ii. 在目标应用节点上下载透明劫持脚本。
目前支持在上海非金融云地域下载，链接如下：

- 公网环境：http://middleware-library.oss-cn-shanghai.aliyuncs.com/sofamesh.sh
- VPC 环境：http://middleware-library.vpc100-oss-cn-shanghai.aliyuncs.com/sofamesh.sh

iii. 在目标节点上执行以下命令修改脚本权限。

```
chmod +x sofamesh.sh
```

iv. 执行以下透明劫持脚本（bash 脚本），安装并启动 Sidecar proxy。

```
sofamesh.sh install \
--tenant <租户名> \                    //选项也可缩写为-t，可在透明劫持页面上单击右上角“查看环境信息”获取。
--workspace <工作空间名称> \           //选项也可缩写为 -w，可在透明劫持页面上单击右上角“查看环境信息”获取。
--instanceid <中间件实例 ID> \         //选项也可缩写为 -i，可在透明劫持页面上单击右上角“查看环境信息”获取。
--appname <应用名称>\                  //选项也可缩写为 -n，必须与您添加应用时的应用名称保持一致。
--accesskey <阿里云 AccessKey Id>\     //选项也可缩写为 -a。
--secretkey <阿里云 AccessKey SecretKey>\  //选项也可缩写为 -s。
--artifacts <安装包链接> \             //选项也可缩写为 -f，脚本依赖的安装包下载链接，下载地址参见下方参数说明。
--verbose                              //选项也可缩写为-v，以打印更详细的执行安装过程。
```

> 说明
>
> - AccessKey 获取方法，请参见 创建 AccessKey。
> - artifacts 参数的下载地址请参见 artifacts。

脚本执行完成后，节点完成初始化，请求将被 MOSN劫持转发。

4. 治理应用。

i. 在透明劫持列表，单击目标应用名称。

ii. 在 **应用详情** 页面右上角，单击 **应用治理**，然后配置最大请求数。
最大请求数不能超过 2147483647，设置为 0 则表示无限制。

iii. 单击 **提交**。
配置完成后，单击 **应用治理记录** 页签即可查看应用治理记录，包括了修改人、修改时间、针对的端口号以及修改值变化信息，如下图所示。

删除透明劫持规则

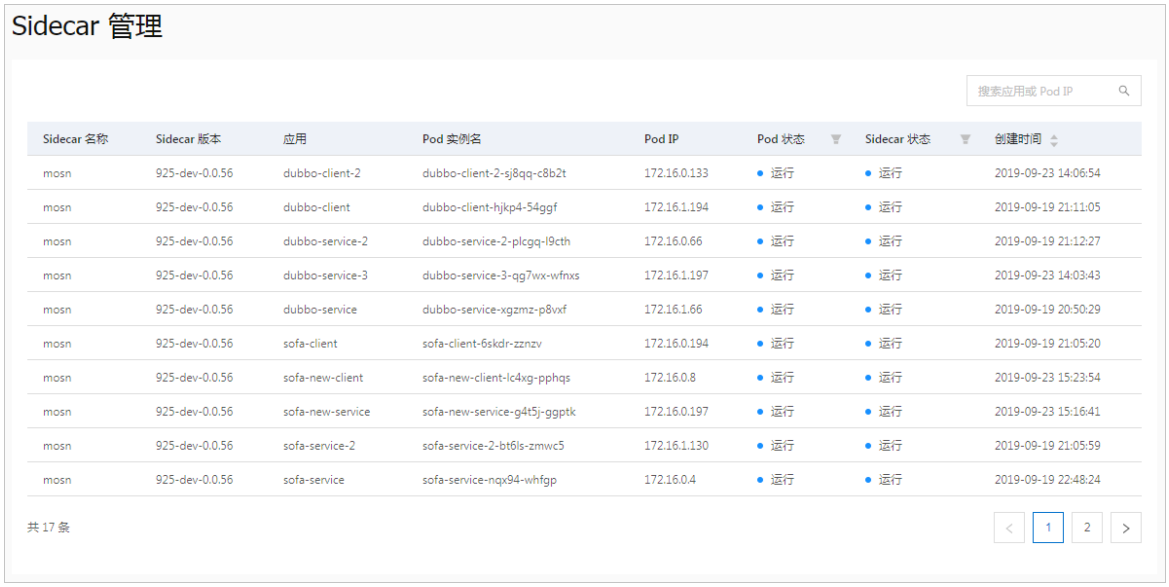
对于不再需要的透明劫持规则，您可以随时删除。

1. 在透明劫持列表，单击目标应用名称。
2. 在 **节点管理** 页签，删除所有节点。
3. 回到透明劫持列表，单击目标应用右侧的 **删除**，然后单击 **确定**。

4.2.5.9. Sidecar 管理

在 Sidecar 管理页面，您可以查看到当前工作空间下所有的 Sidecar 实例列表，以及每个 Sidecar 实例的信息。

1. 登录微服务控制台。
2. 登录 SOFAStack 控制台。
3. 在左侧菜单栏选择 **中间件** > **微服务平台** > **服务网格** > **Sidecar 管理**，查看 Sidecar 实例信息。



参数说明如下：

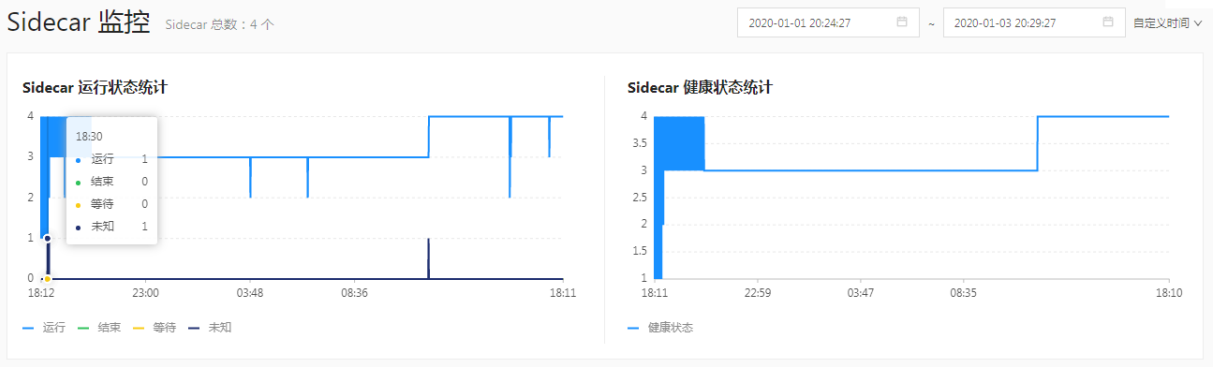
- **Sidecar 名称**：Sidecar 实例的名称。
- **Sidecar 版本**：注入的 Sidecar 版本。
- **应用**：应用名称。
- **Pod 实例名**：Pod 实例的名称。
- **Pod IP**：业务 Pod 的 IP 地址信息。
- **Pod 状态**：业务 Pod 的状态信息，包括堵塞、运行、成功、失败、未知。
- **Sidecar 状态**：Sidecar 实例的状态，包括运行、结束、等待、未知。
- **创建时间**：Sidecar 实例创建时间。

4.2.5.10. Sidecar 监控

您可以在 Sidecar 监控页面查看当前环境中所有 Sidecar 状态统计数据和 Sidecar 版本的各项监控指标数据。

Sidecar 状态统计

在 **Sidecar 监控** 页面，您可以在页面的右上角选择或自定义时间范围，默认范围是最近 5 分钟。统计时间范围设置完成后，可查看该时段内 Sidecar 运行状态统计与 Sidecar 健康状态统计。



- **Sidecar 运行状态统计**：展示了在统计时段内不同状态下的 Sidecar 数量的变化趋势。
  - 运行：表示 Sidecar 正在正常运行中。
  - 结束：表示 Sidecar 已成功执行或因某些原因执行失败，进入停止运行状态。
  - 等待：Sidecar 的默认状态。如果 Sidecar 不处于运行、结束或等待状态，则处于等待状态。
  - 未知：由于某些未知错误引起的状态，例如 sidecar 注入失败等。
- **Sidecar 健康状态统计**：展示了在统计时段内处于健康状态下的 Sidecar 数量的变化趋势。

Sidecar 列表

在 **Sidecar 监控** 页面下方，您可以查看到实时更新的 Sidecar 版本列表。

4.2.5.11. 服务拓扑

实际业务中，应用之间的关联与依赖非常复杂，需要通过全局视角检查具体的局部异常。您可以在服务拓扑页面查看应用在指定时间内的调用及其性能状况。
通过服务拓扑发现的问题，可以通过 服务路由 和 服务限流 进行服务治理。

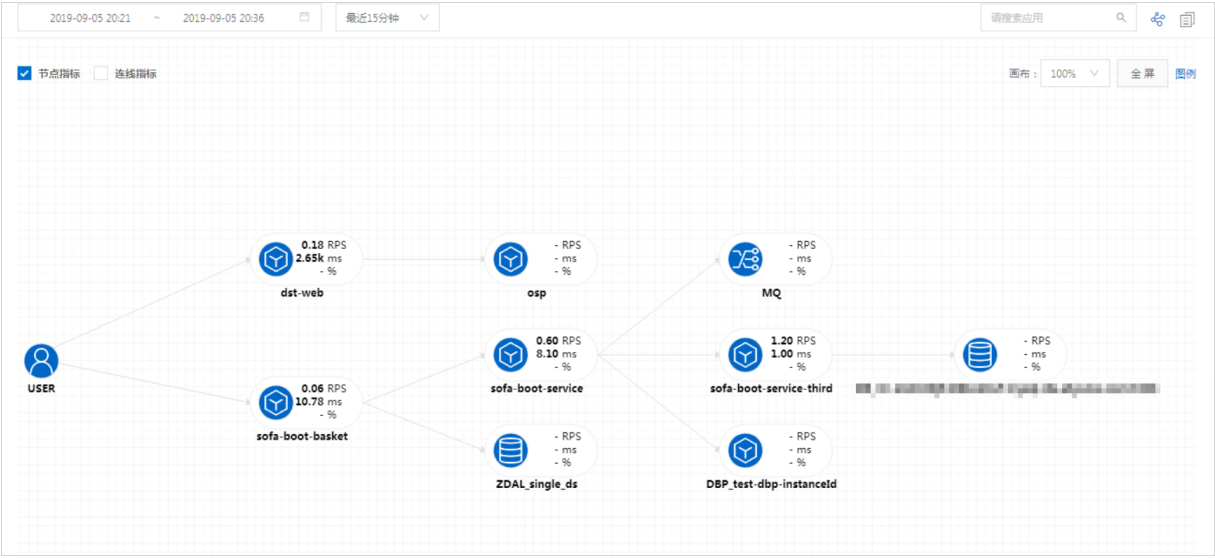
操作步骤

1. 在微服务平台，选择 **服务网格 > 服务拓扑**，打开服务拓扑图。
2. 在拓扑图的左上方，设置起止时间。默认范围为最近 15 分钟，最长时间间隔为 7 天。
3. 选择您想要查看的指标（**节点指标** 或 **连线指标**），拓扑图会相应显示或隐藏相关的指标信息。

服务拓扑图

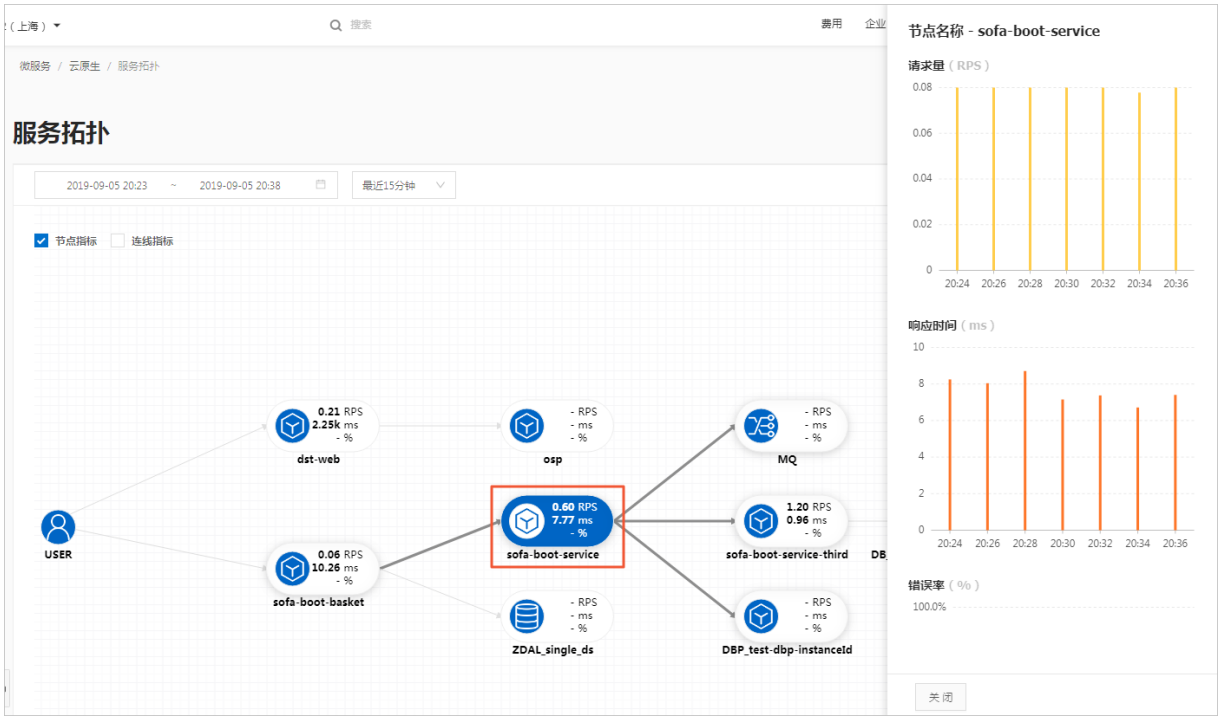
在服务拓扑图中，您可以获取以下信息：

- 应用服务的名称及版本号
- 应用服务间的调用关系
- 服务的请求量（RPS）
- 服务的响应时间（ms）
- 服务的错误率（%）



在服务拓扑图中，单击一个节点图标，即可查看该节点的详细信息。

- 节点上下游相关的拓扑连线。
- 应用节点信息：包括 **请求量**、**响应时间** 及 **错误率**。
- 数据库节点信息：包括 **实例名称**、**IP**、**机房**、**QPS**、**错误率** 及 **响应时间**。
- 缓存节点信息：包括 **实例名称**、**IP**、**机房**、**QPS**、**命中率** 及 **响应时间**。

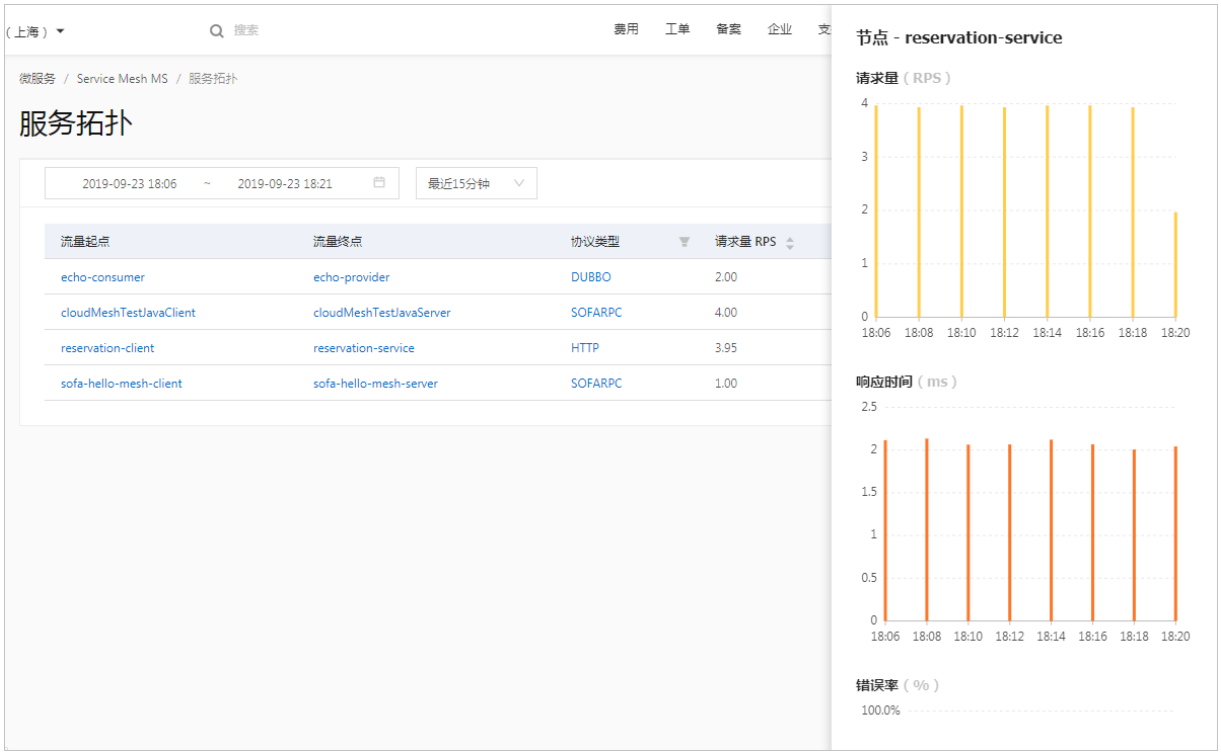


服务列表

您可以单击拓扑图右上角的列表图标切换至列表视图，查看指定统计时段内的服务信息，包括流量起点、流量终点、协议类型、请求量 RPS、响应时间以及错误率。

您还可以对应用进行如下操作：

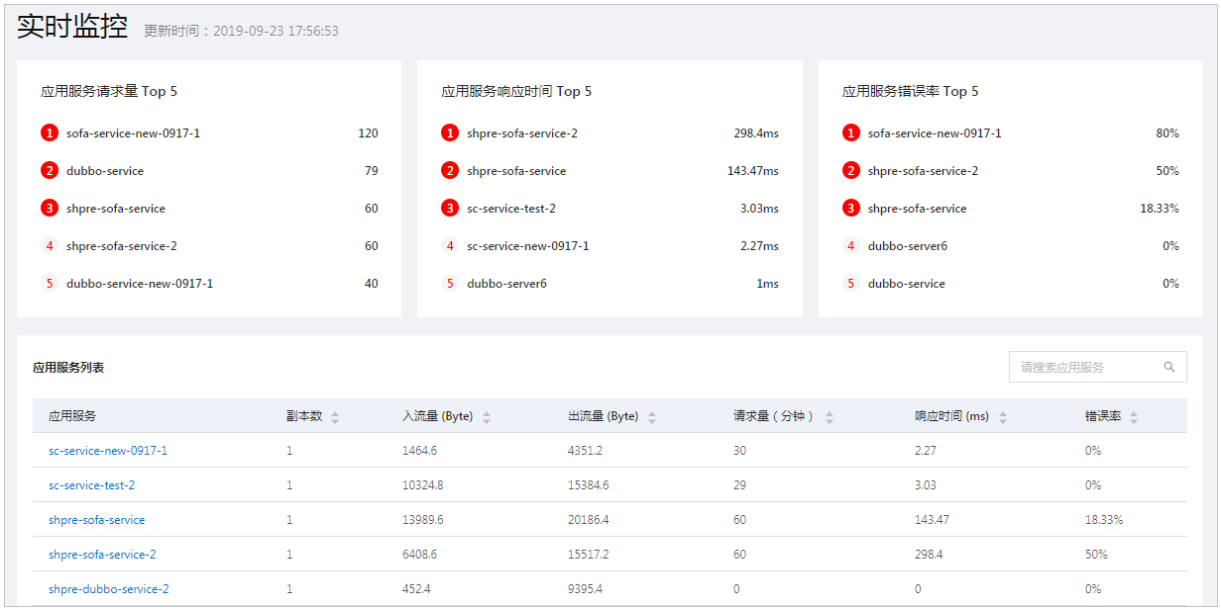
- 单击任意流量起点应用或终点应用名称，即可查看该节点详情。
- 单击协议类型，即可查看流量概览信息。



4.2.5.12. 实时监控

您可以在实时监控页面查看应用服务各项指标的总体统计数据，包括应用服务响应时间、错误率及请求量。

实时监控总览



应用服务 Top 5排行榜

- **应用服务请求量 Top 5**：即请求量最多的 5 个应用，显示在默认统计时间窗口内请求量最多的 5 个应用及其具体请求量数据。
- **应用服务响应时间 Top 5**：即响应时间最长的 5 个应用，显示在默认统计时间窗口内响应时间最长的 5 个应用及其具体响应时长。

- **应用服务错误率 Top 5**：即请求错误率最高的 5 个应用，显示在默认统计时间窗口内请求错误率最高的 5 个应用及其具体响应时长。

应用服务列表

列出当前环境下运行的所有应用及其监控概览信息。

- 应用服务：应用服务名称
- 副本数：副本数量
- 入流量：应用服务的入流量（Byte）
- 出流量：应用服务的出流量（Byte）
- 请求量：应用服务的请求量（分钟）
- 响应时间：应用服务的响应时间（ms）
- 错误率：应用服务的错误率

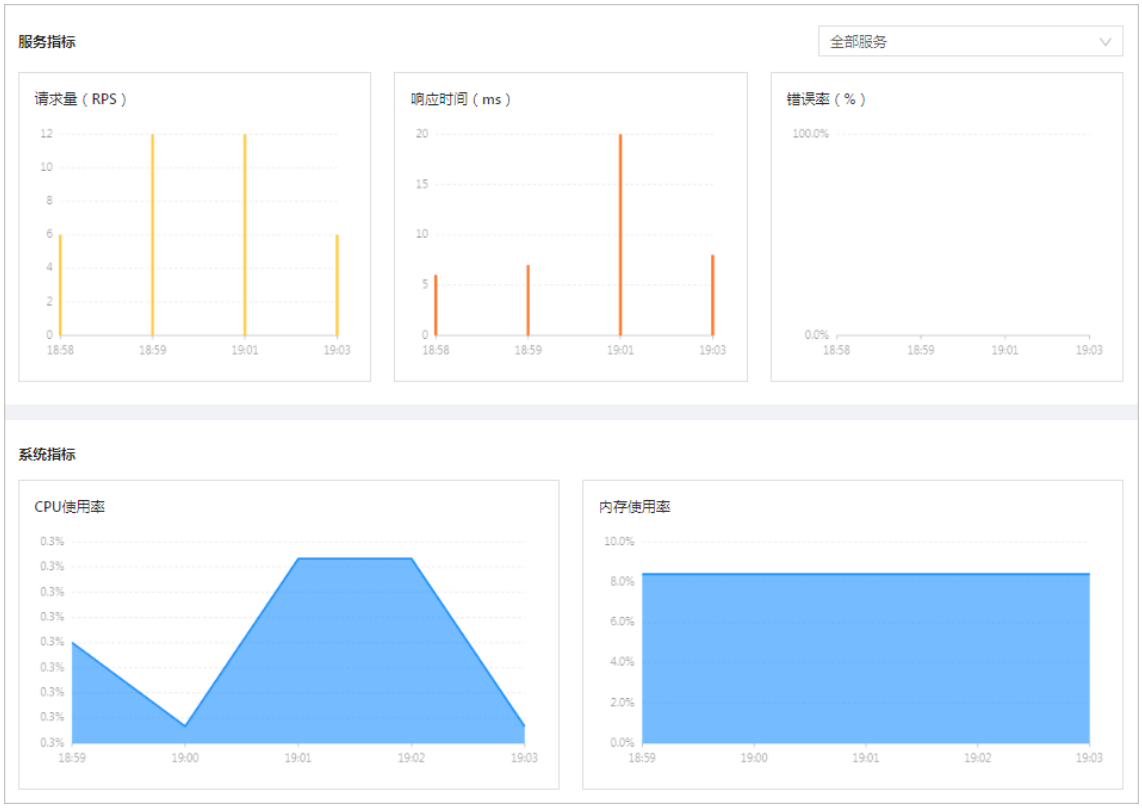
应用服务实时监控详情

在 **实时监控** 页面，单击目标应用服务名称，进入服务的监控详情页，查看该应用服务的各项监控指标数据，及时发现 Pod 异常状况或服务流量过载等，及时处理问题。

1. 在 **实时监控** 页面，单击目标应用服务名称。
2. 在页面顶部的筛选区域，您可以指定以下信息：
   - **应用版本**：选择应用版本。
   - **Pod**：选择 Pod。
   - **时间段**：选择统计窗口的起止时间，默认为最近 5 分钟。

3. 单击**搜索**，页面显示当前指定条件内应用的监控信息，包括以下两部分。
   - **服务指标**：默认显示该应用所有服务的请求量、响应时间以及错误率趋势。您也可以选择查看指定服务的性能状况。

- **系统指标**：显示了统计时间段内应用的 CPU 使用率、内存使用率、入流量以及出流量的趋势走向。



如发现某个应用服务异常，可通过添加限流规则或路由规则进行服务治理。详见 服务路由 及 服务限流。

4.2.6. 日志说明

4.2.6.1. 管理 MOSN 日志级别

查询 MOSN 日志级别

执行以下命令查询 MOSN 日志级别：

```
curl 127.0.0.1:34901/api/v1/get_loglevel
```

获取示例如下：

```
{
  "":"INFO",
  "/home/admin/logs/mosn/antvip.access.log":"INFO",
  "/home/admin/logs/mosn/antvip.error.log":"INFO",
  "/home/admin/logs/mosn/fuse.log":"INFO",
  "/home/admin/logs/mosn/mirror.log":"INFO",
  "/home/admin/logs/mosn/msg_meta.log":"INFO",
  "/home/admin/logs/mosn/registry.access.log":"INFO",
  "/home/admin/logs/mosn/registry.error.log":"INFO",
  "/home/admin/logs/mosn/routerecord.log":"INFO",
  "/home/admin/logs/mosn/routerule.log":"INFO",
  "/home/admin/logs/mosn/zoneclient.access.log":"INFO",
  "/home/admin/logs/mosn/zoneclient.error.log":"INFO"
}
```

修改日志级别

可用的日志级别如下：

- FATAL：输出会导致应用程序退出的严重错误事件日志。
- ERROR：输出不影响系统继续运行的错误和异常信息日志。
- WARN：删除会出现潜在错误的信息。部分信息可能不是错误信息，但需要给程序员的一些提示。
- INFO：输出一些您感兴趣的或者重要的信息，这个可以用于生产环境中输出程序运行的一些重要信息，这些信息在粗粒度级别上突出应用程序的运行过程。不能滥用，避免打印过多的日志。
- DEBUG：主要用于开发过程中打印一些运行信息。一般在调试应用程序时使用。
- TRACE：输出跟踪日志。日志消息的粒度太细，一般不会使用。

执行以下命令修改日志级别：

```
curl -d '{"log_level":"INFO","log_path":"./logs/mosn/antvip.access.log}' 127.0.0.1:34901/api/v1/update_loglevel
```

若修改成功，会返回以下信息：

```
update logger success
```

4.2.6.2. MOSN 内部日志

MOSN 内部日志主要打印在 `/home/admin/logs/mosn` 目录下。

日志名称	说明
default.log	主要用于打印 MOSN 内部的常规日志。 **说明** MOSN 的日志级别默认为 ERROR，仅输出异常日志。当您定位问题时可动态开启为 INFO 级别。
out.log	打印 MOSN 启动的重定向输出日志，如 MOSN 启动异常、GC 或者出现 panic 等。

日志名称	说明
antvip.access.log	打印 acvip 访问日志。主要包括： • acvip 初始化日志 2020-12-20 16:23:15,999 [INFO] [antvip] initialized antvip client with config: &{metrics:0x35a67c0 appName:crpc-client datacenter:dc1 zone:GZ00I instanceID: endpoint: accessKey: accessSecret: trFrom:go-client version:1.0.4 domainChecksumMode:1 syncInterval:1000000000 syncTimeout:90000000000 httpLocator:{https:false address:antvip-pool port:9500 endpoint:/antvip/serversByZone.do timeout:5000000000 interval:30000000000 accesslog:false} registryLocator: {timeout:5000000000} alipaySyncer:{https:false port:9500 endpoint:/antvipDomains accesslog:false zi:<nil>} cloudSyncer: {https:false address:11.239.139.142 port:9003 endpoint:/antcloud/antvip/instances/get accesslog:true}} • 从 acvip 服务端拉取地址日志 2020-12-20 16:25:47,118 [INFO] [antvip] AlipaySyncer POST(polling=true) http://11.239.139.142:9003/antcloud/antvip/instances/get 200 000001-GZ00I-GW_CLOUD&000001-DDCS_CLOUD&[1] <nil> 17.126614ms timeout=1m30s
	打印 DRM 访问日志。主要包括： • DRM 初始化日志 2020-12-20 16:23:17,50 [INFO] [drm] initialized drm client with config: &{instanceID:000001 zone:GZ00I dataCenter:dc1 appName:crpc-client profile: advertiseIFace: accessKey: secretKey: hear tbeatInterval:30000000000 registryLocator: {timeout:3000000000} boltTransport:{timeout:30000000000 maxPendingCommands:2048 maxHeartbeatAttempts:3} antvipLocator:{domain:000001-DDCS_CLOUD tim eout:3000000000}} • 从 acvip 获取服务地址日志 2020-12-20 16:23:16,44 [INFO] [drm] get servers from antvip: [{11.165.199.37 1 80 true DEV-SHANGHAI-ET15-ZONEA dc2 map[]} {11.165.199.49 1 80 true DEV-SHANGHAI-ET15-ZONEA dc1 map[]} {11.1 65.198.1 1 80 false DEV-SHANGHAI-ET15-ZONEA dc1 map[]} {11.167.21.135 1 80 false DEV-SHANGHAI-ET15-ZONEA map[]}] 2020-12-20 16:23:16,50 [INFO] [drm] redial choose 11.165.199.49:9880 • 订阅 dataId 日志

日志名称	说明
drm.access.log	2020-12-20 16:23:17,51 [INFO] [drm] send class=com.alipay.drm.client.api.model.pb.SubscriberRegReqPb req=baseInfo:<zone:"GZ00I" dataId:"Alipay.crpc-client:name=com.alipay.sofa.middleware.mesh.drm.MsgResource.msgSep,version=3.0@DRM" uuid:"0ce5252f-8c89-4d8c-8c63-9ffaa0dcab98" instanceId:"000001" > res=nil err=nil elapsed=18.399µs • 收到 DRM 服务端推送通知，进行更新 2020-12-20 16:24:43,554 [INFO] [drm] drm push dataID=sofa.config:name=com.alipay.sofa.middleware.servicegov.apps.crpc-client.downgradeRule,version=3.0@DRM command="CMD:zone=DEFAULT_ZONE,version=125:R@DRM" 2020-12-20 16:24:43,569 [INFO] [drm] fetch GET http://11.165.199.37/queryDrmData.htm?dataId=sofa.config:name=com.alipay.sofa.middleware.servicegov.apps.crpc-client.downgradeRule,version=3.0@DRM&zone=DEFAULT_ZONE&version=125&profile=&instanceId=000001 200 <{"appName":"crpc-client","type":"SOFA","enabled":true,"downgrade...> [125] 78 <nil> 14.886412ms 2020-12-20 16:24:43,569 [INFO] [drm] update local value dataID=sofa.config:name=com.alipay.sofa.middleware.servicegov.apps.crpc-client.downgradeRule,version=3.0@DRM value={"appName":"crpc-client","type":"SOFA","enabled":true,"downgradeRules":[]} version=125

日志名称	说明
endpoint.access.log	打印 MOSN 启动时，模块初始化日志。主要包括： • Annotation 初始化日志 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] annoApp info from annotation: 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] {"antShareCloud":true,"instanceId":"000001","AntvipServerAddress":"11.239.139.142","dataCenter":"dc1","zone":"GZ00I","appName":"crpc-client"} 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] pub info from annotation: 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] [] 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] sub info from annotation: 2020-12-20 16:23:15,999 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] [{"serviceName":"org.example.services.userInterface.IHelloService:1.0.0:default@crpc","protocol":"crpc"}{"serviceName":"org.example.services.echoService.XyEchoService:1.0.0:default@crpc","protocol":"crpc"}] 2020-12-20 16:23:28,230 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] pubsub annotations done 2020-12-20 16:23:28,230 [INFO] [endpoint] [registryengine] [pubSubFromAnnotations] done • 后端服务健康检查日志 2020-12-20 16:37:00,32 [INFO] [endpoint] [health check] [confreg callback] host=127.0.0.1:9999 changed=false isHealthy=true 2020-12-20 16:37:00,63 [INFO] [endpoint] [health check] [conf callback once] host=127.0.0.1:9999 cluster=local_crpc_service_org.example.services.echoService.XyEchoService:1.0.0:default@crpc changed=false isHealthy=true

日志名称	说明
gateway.access.log	打印 APIGateway 初始化日志和从 acvip 获取的网关服务地址信息。 2020-12-20 16:42:22,199 [INFO] [gateway] Received gateway address changed event. domain = &{Name:000001-GZ00S-GW_CLOUD ProtectThreshold:50 HealthCheckType:TCP HealthCheckDefaultPort:12200 HealthCheckTimeout:2000 HealthCheckInterval:5000 HealthCheckRaise:1 HealthCheckFall:3 HealthCheckEnable:false HealthCheckPayload:map[] Version:4 RealNodes:[{Ip:100.88.177.40 Weight:1 HealthCheckPort:30777 Zone:ACVIP-TEST Available:true RoundTripTime:5 Reason:Success LastHealthCheckTime:1607689660570 DataCenter: Falling:false Labels:map[]}] Labels:map[] IsDeleted:false ZoneInfoList:<nil> weightedAvailableRealServers:[] availableRealServers:[] allRealServer:[] weightedAllRealServers:[] idcWeightedAvailableRsMap:map[] cityWeightedAvailableRsMap:map[] idcWeightedCntMap:map[]} 2020-12-20 16:42:22,199 [INFO] [gateway] Update gateway address success. new servers = [{Address:100.88.177.40:30777 Host:100.88.177.40}]
skywalking.access.log	打印 Skywalking 初始化日志和从 acvip 获取 Skywalking 地址信息。 2020-12-20 16:42:22,217 [INFO] [skywalking] Received sw address changed event. domain = &{Name:000001-SkyWalking_CLOUD ProtectThreshold:50 HealthCheckType:TCP HealthCheckDefaultPort:12200 HealthCheckTimeout:2000 HealthCheckInterval:5000 HealthCheckRaise:1 HealthCheckFall:3 HealthCheckEnable:false HealthCheckPayload:map[] Version:6 RealNodes:[{Ip:11.166.176.49 Weight:1 HealthCheckPort:11800 Zone:CZ00A Available:true RoundTripTime:6 Reason:Success LastHealthCheckTime:1608112125577 DataCenter: Falling:false Labels:map[]}] Labels:map[] IsDeleted:false ZoneInfoList:<nil> weightedAvailableRealServers:[] availableRealServers:[] allRealServer:[] weightedAllRealServers:[] idcWeightedAvailableRsMap:map[] cityWeightedAvailableRsMap:map[] idcWeightedCntMap:map[]} 2020-12-20 16:42:22,217 [INFO] [skywalking] Update sw address success. new servers = [11.166.176.49:11800

日志名称	说明
flow-control.log	打印限流 metric 日志。 2020-12-20 16:24:46,909 [INFO] |org.example.services.userInterface.IHelloService:1.0.0:default@crpc.sayHi:o:r#o|3|0|3|0|2|0|0|2|0|0|0 日志格式说明： datetime | 资源点名 | pass | block | complete | exception | rt | 预占用值（忽略） | 瞬时并发量 | 资源点类型 | 监控模式 block | 秒级最大并发量 | CE 汇报的线程占用量
fault_tolerance.log	打印故障隔离日志。
fault_inject.log	打印故障注入日志。
downgrade.log	打印降级日志。 2020-12-20 16:23:17,955 [INFO] [downgrade] receive drm resource push sofa.config:name=com.alipay.sofa.middleware.servicegov.apps.crpc-client.downgradeRule,version=3.0@DRM={"appName":"crpc-client","type":"SOFA","enabled":true,"downgradeRules":[{"enabled":true,"conditions":[{"type":"SYSTEM","field":"destination.service.name","operation":"EQUAL","value":["org.example.services.userInterface.IHelloService:1.0.0:default@crpc"]},{"type":"SYSTEM","field":"destination.method.name","operation":"EQUAL","value":["sayHi"]}],"actions":[{"type":"REJECT","enabled":true,"extension":[{}]}],"logConfig":{"level":"WARN","enabled":true}}]} 2020-12-20 16:23:17,956 [INFO] local downgrade rules: &{AppName:crpc-client Enabled:true DowngradeRules:[0xc00220ad20]} 2020-12-20 16:24:43,569 [INFO] [downgrade] receive drm resource push sofa.config:name=com.alipay.sofa.middleware.servicegov.apps.crpc-client.downgradeRule,version=3.0@DRM={"appName":"crpc-client","type":"SOFA","enabled":true,"downgradeRules":[]} 2020-12-20 16:24:43,569 [INFO] local downgrade rules: &{AppName:crpc-client Enabled:true DowngradeRules:[]}

4.2.6.3. Pilot 日志

本文介绍 Pilot 日志的一些相关配置说明和查询，包括启动日志参数、修改日志级别及日志查询。

Pilot 启动日志参数

```
- --log_target=stdout //日志输出目标
- --log_rotate=/home/admin/logs/discovery.log //日志文件路径
- --log_rotate_max_size=512  //日志滚动大小，单位是 KB
- --log_rotate_max_backups=24 //日志滚动时间，时间单位是 s
- --log_as_json=false  //是否以 JSON 格式输出，true - 是，false - 否
```

修改 Pilot 日志级别

```
~ curl 127.0.0.1/logging
  Usage: curl 'http://127.0.0.1:8080/logging?<scope>=<logLevel>&<scope>=<logLevel>'
  default : info
  model : info
  proxy connection : info
  ads : info
~ curl "127.0.0.1:8080/logging?all=debug"
  set scope ads log level to debug
  set scope default log level to debug
  set scope model log level to debug
  set scope proxy connection log level to debug
```

查询 Pilot 日志

```
//进入 pilot-pod
kubectl -n cloudmesh exec -it pilot-name -c discovery /bin/bash
//查看日志信息
tail -10000f /home/admin/logs/discovery.log
```

4.2.6.4. Pilot 详细日志说明

本文对 Pilot 日志进行详细说明。 对连接日志和 MOSN xds 下发日志两种类型日志的详细说明。

连接日志

Pilot 连接日志

```
2020-12-07 14:02:05 debug ads node.id id:"sidecar~192.168.20.181~acctrans-eu95-1-rz30a-test-alipay-net-mx0ogdef-gtjsqa.acctrans~acctrans.rz30a.alipay.net" cluster:"gtjsqa" metadata:<fields:<key:"ISTIO_PROXY_VERSION" value:<string_value:"1.8.0" > > > , typeUrl type.googleapis.com/envoy.api.v2.Cluster
2020-12-07 14:02:05 debug ads ADS:CDS: ACK 192.168.20.181:53534 node:<id:"sidecar~192.168.20.181~acctrans-eu95-1-rz30a-test-alipay-net-mx0ogdef-gtjsqa.acctrans~acctrans.rz30a.alipay.net" cluster:"gtjsqa" metadata:<fields:<key:"ISTIO_PROXY_VERSION" value:<string_value:"1.8.0" > > > > type_url:"type.googleapis.com/envoy.api.v2.Cluster"
2020-12-07 14:02:06 debug Handling event update for pod 21f0e2e9-ab26-4890-921d-f661229888b6 in namespace ant-migrate-container -> 21f0e2e9-ab26-4890-921d-f661229888b6.ant-migrate-container.Kubernetes
2020-12-07 14:02:06 debug ads ADS count add 1, current  1
```

MOSN 连接日志

```
2020-12-07 14:36:56,522 [INFO] xds client start
2020-12-07 14:36:56,525 [INFO] mosn estab grpc connection to pilot at 192.168.214.205:15050
```

MOSN xds 下发日志

CDS Pilot 日志

```
//CDS REQ 接收到 MOSN CDS 请求
2020-12-07 14:02:06 info ads ADS:CDS: REQ 192.168.129.1:51467 sidecar~192.168.54.18~crpc-client-79597c896d-k2svb.et15-dev~et15-dev.||multitenancy.tenant=ALIPAYCN~multitenancy.workspace=middleware~multitenancy.cluster=cb7aaa7a31fd0490299c530e83a868231||000001~BQwUKxGNmbZuaIBuHv~8UslmO/FiaQOqp7c6W6Y8w20ggFMuHYG2aTyKDfEc+w=~HmacSHA256-1 86.463005ms raw: node:<id:"sidecar~192.168.54.18~crpc-client-79597c896d-k2svb.et15-dev~et15-dev.||multitenancy.tenant=ALIPAYCN~multitenancy.workspace=middleware~multitenancy.cluster=cb7aaa7a31fd0490299c530e83a868231||000001~BQwUKxGNmbZuaIBuHv~8UslmO/FiaQOqp7c6W6Y8w20ggFMuHYG2aTyKDfEc+w=~HmacSHA256" metadata:<fields:<key:"ISTIO_PROXY_VERSION" value:<string_value:"0.0.1" > > fields:<key:"LABELS" value:<string_value:"{\"VMMODE\":\"true\",\"ac.vm.clusterName\":\"data-plane\",\"app.kubernetes.io/name\":\"reservation-service\",\"cafe.sofastack.io/tenant\":\"ALIPAYCN\",\"cafe.sofastack.io/workspace\":\"middleware\"}" > > fields:<key:"USING_ANY" value:<string_value:"1" > > > > type_url:"type.googleapis.com/envoy.api.v2.Cluster"
2020-12-07 14:02:06 debug build cluster configs
// CDS RESPON 向 MOSN 下发 Cluster 信息，数量为 2 个
2020-12-07 14:02:06 debug ads CDS: PUSH 2020-12-07T14:00:04+08:00/1006 for node:crpc-client-79597c896d-k2svb.et15-dev.Kubernetes() ConID:sidecar~192.168.54.18~crpc-client-79597c896d-k2svb.et15-dev~et15-dev.||multitenancy.tenant=ALIPAYCN~multitenancy.workspace=middleware~multitenancy.cluster=cb7aaa7a31fd0490299c530e83a868231||000001~BQwUKxGNmbZuaIBuHv~8UslmO/FiaQOqp7c6W6Y8w20ggFMuHYG2aTyKDfEc+w=~HmacSHA256-1 addr:"192.168.129.1:51467", Clusters: 2
```

CDS MOSN 日志

```
// MOSN 接收到的 CDS日志
2020-12-07 14:36:56,540 [INFO] [xds] parse cluster. cluster = name:"outbound|12220||com.alipay.antscheduler.facade.IActivityInstanceFacade:1.0@DEFAULT" type:EDS connect_timeout:<seconds:1 > lb_subset_config:<fallback_policy:ANY_ENDPOINT subset_selectors:<keys:"app.kubernetes.io/version" > >
2020-12-07 14:36:56,540 [INFO] [xds] parse cluster. cluster = name:"outbound|12220||egress-apps-rest-client" type:EDS connect_timeout:<seconds:1 > lb_subset_config:<fallback_policy:ANY_ENDPOINT subset_selectors:<keys:"com.alipay.sofa.label.k1" > >
2020-12-07 14:36:56,540 [INFO] [xds] parse cluster. cluster = name:"BlackHoleCluster" type:STATIC connect_timeout:<seconds:1 >
2020-12-07 14:36:56,540 [INFO] [xds] parse cluster. cluster = name:"PassthroughCluster" type:ORIGINAL_DST connect_timeout:<seconds:1 > lb_policy:ORIGINAL_DST_LB
```

LDS Pilot 日志

```
// Pilot 接收到 MOSN 的 LDS 请求
2020-12-07 14:41:30 info ads ADS:LDS: REQ sidecar~192.168.215.88~crpc-client-2-6bdd769cf8-wz9gp.et15-dev~et15-dev.||multitenancy.tenant=ALIPAYCN~multitenancy.workspace=middleware~multitenancy.cluster=cb7aaa7a31fd0490299c530e83a868231||000001~~/ZqENC5hN7e/P6ZVJKyoGOm9pgJJKCpZE5tAOBQO0GQ=~HmacSHA256-1894 192.168.215.88:33808
```

LDS MOSN 日志

```
//MOSN 接收到 Pilot 下发的 LDS
2020-12-07 14:36:56,555 [INFO] get 16 listeners from LDS
```

RDS Pilot 日志

```
// Pilot 接收到 MOSN RDS REQ 请求
info ads ADS:RDS: REQ 192.168.129.1:50078 sidecar~192.168.54.18~crpc-client-79597c896d-k2svb.et15-dev~et15-dev.||multitenancy.tenant=ALIPAYCN~multitenancy.workspace=middleware~multitenancy.cluster=cb7aaa7a31fd0490299c530e83a868231||000001~BQwUKxGNmbZuaIBuHv~8UslmO/FiaQOqp7c6W6Y8w20ggFMuHYG2aTyKDfEc+w=~HmacSHA256-1884  routes: 4

//  Pilot Push MOSN RDS
debug ads ADS: RDS: PUSH for node: crpc-client-79597c896d-k2svb.et15-dev.data-plane(ALIPAYCN-middleware) addr:192.168.129.1:50229 routes:4
```

4.2.6.5. MOSN Trace 日志

服务网格使用 SOFATracer、Zipkin 实现分布式链路跟踪，将调用链路中的各种网络调用情况以日志的方式记录下来，以达到透视化网络调用的目的，这些链路数据可用于故障的快速发现和服务治理等。本文介绍 SOFATracer 和 Zipkin 的日志格式和示例。

SOFATracer

服务网格支持 SOFA、Dubbo、SpringCloud 三种微服务。SOFATracer 将微服务日志记录在 `/home/admin/logs/tracelog/mosn` 路径下，下面根据微服务类型介绍 SOFATracer 日志及示例。

SOFA

服务调用方

日志文件名： `rpc-client-digest.log` 。

日志示例如下：

```
2021-02-05 13:46:44.264, //日志打印时间
sofa-echo-server, //当前应用名
1e49a2be161250400426012467 5912,//TraceId
0,//RpcId
com.alipay.sofa.ms.service.SofaEchoService:1.0,//服务名
echo,//方法名
bolt,//协议
SYNC,//调用方式
30.73.162.190:12200,//目标地址
sofa-echo-server,//目标系统名
GZ00B,//目标 Zone
,//目标 IDC
,//目标 City
,//uid
00,//结果码
388B,//请求大小
105B,//响应大小
2ms,//调用耗时
0ms,//建立连接耗时
0ms,//请求序列化耗时
0ms,超时参考耗时
,//当前线程名
,//路由记录
,//弹性数据位
,//是否需要弹性
,//转发的服务名称
127.0.0.1,//Client IP
61181,//Client Port
,//当前 Zone
F,//是否物理机器
,//systemMap
mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default
&mosn_log=true&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true
&mosn_data_id=com.alipay.sofa.ms.service.SofaEchoService:1.0@DEFAULT
&mosn_data_ver=&,//系统穿透数据
606.52μs//MOSN 处理时间
```

日志格式说明如下：

日志内容	说明
2021-02-05 13:46:44.264	日志打印时间，格式为 `yyyy-MM-dd HH:mm:ss.SSS` 。
sofa-echo-server	当前应用名。
1e49a2be161250400426012467 5912	TraceId，分布式链路追踪的唯一标识。
0	RpcId。

日志内容	说明
com.alipay.sofa.ms.service.SofaEchoService:1.0	服务名。
echo	方法名。
bolt	协议。有 bolt 和 rest 两种。
SYNC	调用方式。 • SYNC：同步。 • ASYNC：异步。
30.73.162.190:12200	目标地址。
sofa-echo-server	目标应用名。
GZ00B	目标 Zone。
-	目标 IDC。
-	目标 City，示例值：beijing。
-	uid。
00	结果码。 • 00：成功 • 01：业务异常 • 02：RPC 逻辑错误 • 03：超时失败 • 04：路由失败
388B	请求大小，单位字节（B）。
105B	响应大小，单位字节（B）。
2ms	调用耗时，单位毫秒（ms）。
0ms	建立连接耗时，单位毫秒（ms）。
0ms	请求序列化耗时，单位毫秒（ms）。
0ms	超时参考耗时，单位毫秒（ms）。
-	当前线程名。
-	路由记录。
-	弹性数据位。
-	是否需要弹性。

日志内容	说明
-	转发的服务名称，示例值：`com.alipay.sofa.ms.service.SofaEchoService:1.0@DEFAULT`。
127.0.0.1	Client IP。
61181	Client Port。
-	当前 Zone，示例值：Gzone。
F	是否物理机。
-	systemMap。
mosn_cluster=cb7aaa7a31fd0490299c530e83a868231 &mosn_namespace=default &mosn_log=true&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231 &mosn_namespace=default&mosn_log=true &mosn_data_id=com.alipay.sofa.ms.service.SofaEchoService:1.0@DEFAULT &mosn_data_ver=&	系统穿透数据。
606.52μs	MOSN 处理时间，单位微秒（μs）。

服务提供方

日志文件名：`rpc-server-digest.log`。

日志示例如下：

```
2021-02-05 13:47:13.311,//日志打印时间
sofa-echo-server,//当前应用名
0ba68508161250403323534141O,//TraceId
0,//RpcId
com.alipay.sofa.ms.service.SofaEchoService:1.0,//服务名
echo,//方法名
bolt,//协议
,//调用方式
11.166.133.8:41711,//调用者 URL
,//调用者应用名
,//调用者 Zone
,//调用者 IDC
2ms,//处理请求耗时（ms）
0ms,//服务端响应序列化耗时（ms）
,//当前线程名
00,// 结果码
,//beElasticServiceName
,//beElastic
0,//RPC 线程池等待时间
,//系统穿透数据
mosn_tls_state=off&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default
&mosn_log=true&mosn_tls_state=off
&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true
&mosn_data_id=com.alipay.sofa.ms.service.SofaEchoService:1.0@DEFAULT
&mosn_data_ver=&,//穿透数据（kv格式）
681.534μs,//MOSN 处理时间
373B,//请求大小
105B//响应大小
```

日志格式说明如下：

日志内容	说明
2021-02-05 13:47:13.311	日志打印时间，格式为 `yyyy-MM-dd HH:mm:ss.SSS` 。
sofa-echo-server	当前应用名。
0ba68508161250403323534141O	TraceId，分布式链路追踪的唯一标识。
0	RpcId。
com.alipay.sofa.ms.service.SofaEchoService:1.0	服务名。
echo	方法名。
bolt	协议，有 bolt 和 rest 两种。

日志内容	说明
-	调用方式。 • SYNC：同步。 • ASYNC：异步。
11.166.133.8:41711	调用者 URL。
-	调用者应用名。
-	调用者 Zone。
-	调用者 IDC。
2ms	处理请求耗时，单位毫秒（ms）。
0ms	服务端响应序列化耗时，单位毫秒（ms）。
-	当前线程名。
00	结果码。 • 00：成功 • 01：业务异常 • 02：RPC逻辑错误
-	表明这次调用是转发调用，转发的服务名称和方法名称，示例值： `com.test.service.testservice.TestService:1.0:biztest---doProcess` ）。
-	表示没有被转发的处理。
0	RPC 线程池等待时间。
-	系统穿透数据（kv 格式，用于传送系统灾备信息等）。
mosn_tls_state=off&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231 &mosn_namespace=default&mosn_log=true&mosn_tls_state=off &mosn_cluster=cb7aaa7a31fd0490299c530e83a868231 &mosn_namespace=default&mosn_log=true &mosn_data_id=com.alipay.sofa.ms.service.SofaEchoService:1.0@DEFAULT &mosn_data_ver=&	穿透数据（kv格式）。
681.534μs	MOSN 处理时间，单位微妙（μs）。
373B	请求大小，单位字节（B）。
105B	响应大小，单位字节（B）。

Dubbo

服务调用方

日志文件名：`rpc-client-digest.log` 。

日志示例如下：

```
2021-02-05 11:57:28.284,//日志打印时间
reservation-service,//当前应用名
1e49a2be161249744828114516886,//TraceId
0,//RpcId
com.alipay.sofa.ms.service.EchoService,//服务名
echo,//方法名
Dubbo,//协议
SYNC,//调用方式
30.73.162.190:30800,//目标地址
reservation-service,//目标系统名
GZ00B,//目标 Zone
,//目标 IDC
,//目标 City
,//uid
00,//结果码（00=成功、01=业务异常、02=RPC逻辑错误、03=超时失败、04=路由失败）
242B,//请求大小
118B,//响应大小
2ms,//调用耗时
0ms,//建立连接耗时
0ms,//请求序列化耗时
0ms,//超时参考耗时
,//当前线程名
,//路由记录
,//弹性数据位
,//是否需要弹性
,//转发的服务名称
30.73.162.190,//Client IP
57510,//Client Port
,//当前 Zone
F,//是否物理机器
,//systemMap
mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true
&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true&
mosn_data_id=com.alipay.sofa.ms.service.EchoService:aaa:bbb@dubbo
&mosn_data_ver=&,//系统穿透数据
325.799µs//MOSN 处理时间
```

Dubbo 日志格式和 SOFA 日志格式完全一致。

服务提供方

日志文件名：`rpc-server-digest.log` 。

日志示例如下：

```
2021-02-05 11:57:53.431,//日志打印时间
reservation-service,//当前应用名
1e49a2be161249747343014806886,//TraceId
0,//RpcId
com.alipay.sofa.ms.service.EchoService,//服务名
echo,//方法名
Dubbo,//协议
,//调用方式
30.73.162.190:57523,//调用者 URL
,//调用者应用名
,//调用者 Zone
,//调用者 IDC
1ms,//请求处理耗时（ms）
0ms,//服务端响应序列化耗时（ms）
,//当前线程名
00,//结果码（00=成功、01=业务异常、02=RPC逻辑错误）
,//表明这次调用是转发调用，转发的服务名称和方法名称，如：
com.test.service.testservice.TestService:1.0:biztest---doProcess

,//表示没有被转发的处理
0,//rpc线程池等待时间
,//系统穿透数据（kv 格式，用于传送系统灾备信息等）
mosn_tls_state=off&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true&mosn_tls_state=off
&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true
&mosn_data_id=com.alipay.sofa.ms.service.EchoService:aaa:bbb@dubbo
&mosn_data_ver=&,//穿透数据（kv格式）
285.268μs,//MOSN 处理时间
242B,//请求大小
118B//响应大小
```

Dubbo 日志格式和 SOFA 日志格式完全一致。

SpringCloud

服务调用方

日志文件名：`springcloud-client-digest.log`

日志示例如下：

```
2021-02-05 11:55:33.61,//日志打印时间
reservation-service,//当前应用名
1e49a2be161249733360713416886,//TraceId
0,//RpcId
127.0.0.1:10088,//host
/echo/name,//uri
HTTP,//协议
SYNC,//调用方式
30.73.162.190:10080,//目标地址
reservation-service,//目标系统名
GZ00B,//目标 Zone
,//目标 IDC
,//目标 City
,//uid
00,//结果码（00=成功、01=业务异常、02=RPC逻辑错误、03=超时失败、04=路由失败）
18B,//请求大小
20B,//响应大小
2ms,//调用耗时
0ms,//建立连接耗时
0ms,//请求序列化耗时
0ms,//超时参考耗时
,//当前线程名
,//路由记录
,//弹性数据位
,//是否需要弹性
,//转发的服务名称
127.0.0.1,//Client IP
57817,//Client Port
,//当前 Zone
F,//是否物理机器
,//systemMap
mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true&,//系统穿透数据
426.311µs//MOSN 处理时间
```

日志内容说明如下：

日志内容	说明
2021-02-05 11:55:33.61	日志打印时间，格式为 `yyyy-MM-dd HH:mm:ss.SSS` 。
reservation-service	当前应用名。
1e49a2be161249733360713416886	TraceId，分布式链路追踪的唯一标识。
0	RpcId。
127.0.0.1:10088	host。
/echo/name	uri。

日志内容	说明
HTTP	协议，有 bolt 和 rest 两种。
SYNC	调用方式。 • SYNC：同步。 • ASYNC：异步。
30.73.162.190:10080	目标地址。
reservation-service	目标系统名。
GZ00B	目标 Zone。
-	目标 IDC。
-	目标 City。
-	uid。
00	结果码。 • 00：成功 • 01：业务异常 • 02：RPC逻辑错误 • 03：超时失败 • 04：路由失败
18B	请求大小，单位字节（B）。
20B	相应大小，单位字节（B）。
2ms	调用耗时，单位毫秒（ms）
0ms	建立连接耗时，单位毫秒（ms）。
0ms	请求序列化耗时，单位毫秒（ms）。
0ms	超时参考耗时，单位毫秒（ms）。
-	当前线程名。
-	路由记录。
-	弹性数据位。
-	是否需要弹性。
-	转发的服务名称。
127.0.0.1	Client IP。

日志内容	说明
57817	Client Port。
-	当前 Zone。
F	是否物理机。
-	systemMap。
mosn_cluster=cb7aaa7a31fd0490299c530e83a868231 &mosn_namespace=default&mosn_log=true&	系统穿透数据。
426.311µs	MOSN 处理时间，单位微妙（µs）。

服务提供方

日志文件名：`springcloud-server-digest.log`

日志示例如下：

```
2021-02-05 11:56:30.962,//日志打印时间
reservation-service,//当前应用名
1e49a2be161249739096013986888 86,//TraceId
0,//RpcId
127.0.0.1:10088,//host
/echo/name/aaa,//uri
HTTP,//协议
,//调用方式
30.73.162.190:57818,//调用者 URL
reservation-service,//调用者应用名
,//调用者 Zone
,//调用者 IDC
1ms,//处理请求耗时（ms）
0ms,//服务端响应序列化耗时（ms）
,//当前线程名
00,//结果码（00=成功、01=业务异常、02=RPC逻辑错误）
,//表明这次调用是转发调用，转发的服务名称和方法名称，如:
com.test.service.testservice.TestService:1.0:biztest---doProcess

,//表示没有被转发的处理
0,//RPC 线程池等待时间
,//系统穿透数据（kv 格式，用于传送系统灾备信息等）
mosn_tls_state=off&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231
&mosn_namespace=default&mosn_log=true&,//穿透数据（kv格式）
262.787µs,//MOSN 处理时间
0B,//请求大小
16B//响应大小
```

日志内容说明如下：

日志内容	说明

日志内容	说明
2021-02-05 11:56:30.962	日志打印时间，格式为 `yyyy-MM-dd HH:mm:ss.SSS` 。
reservation-service	当前应用名。
1e49a2be161249739096013986886	TraceId，分布式链路追踪的唯一标识。
127.0.0.1:10088	host。
/echo/name/aaa	uri。
HTTP	协议，有 bolt 和 rest 两种。
-	调用方式。 • SYNC：同步。 • ASYNC：异步。
30.73.162.190:57818	调用者 URL。
reservation-service	调用者应用名。
-	调用者 Zone。
-	调用者 IDC。
1ms	处理请求耗时，单位毫秒（ms）。
0ms	服务端响应序列化耗时，单位毫秒（ms）。
-	当前线程名。
00	结果码。 • 00：成功 • 01：业务异常 • 02：RPC 逻辑错误
-	表明这次调用是转发调用，转发的服务名称和方法名称，示例值：`com.test.service.testservice.TestService:1.0:biztest---doProcess` ）
-	表示没有被转发的处理。
0	RPC 线程池等待时间。
-	系统穿透数据（kv 格式，用于传送系统灾备信息等）。

日志内容	说明
mosn_tls_state=off&mosn_cluster=cb7aaa7a31fd0490299c530e83a868231&mosn_namespace=default&mosn_log=true&	穿透数据（kv 格式）。
262.787µs	MOSN 处理时间，单位微妙（µs）。
0B	请求大小，单位字节（B）。
16B	响应大小，单位字节（B）。

4.2.7. 最佳实践

4.2.7.1. 服务网格落地之核心篇

2019 年双十一是蚂蚁集团架构云化的关键时间节点，Service Mesh 是应用云化非常重要的一环。业务与基础设施层的解耦势在必行，Mesh 化为这层解耦带来了实际可落地的解决方案。本文主要介绍蚂蚁集团 Service Mesh 落地实践的核心部分。
本文主要内容分为下述几个方面：

- 基础能力建设
- 前期准备

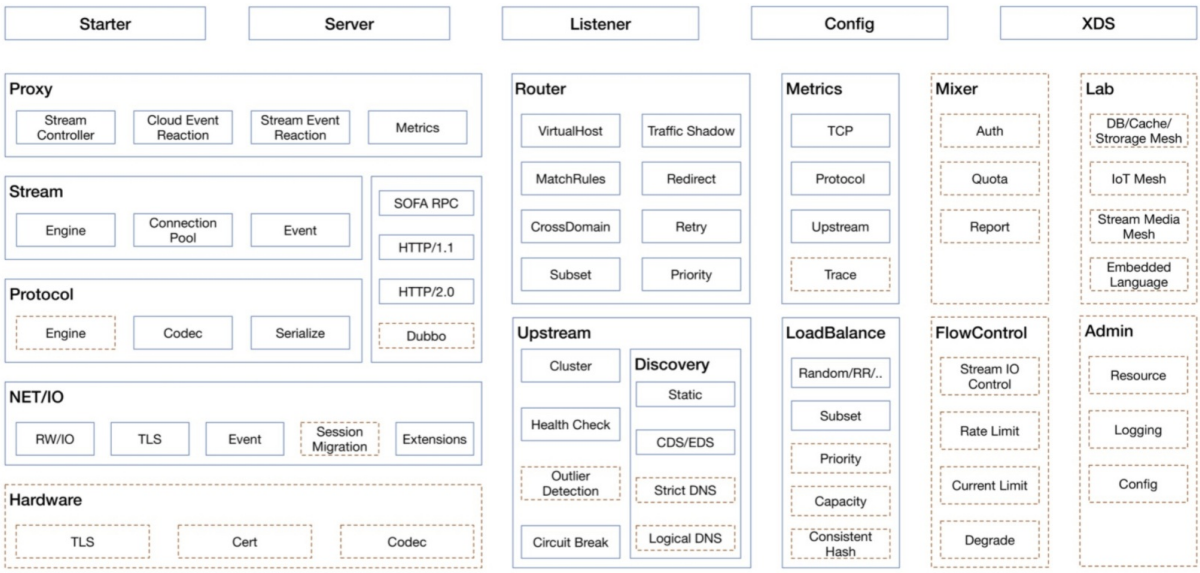
基础能力建设

SOFAMosn 能力大图

SOFAMosn 主要包括了下述能力：

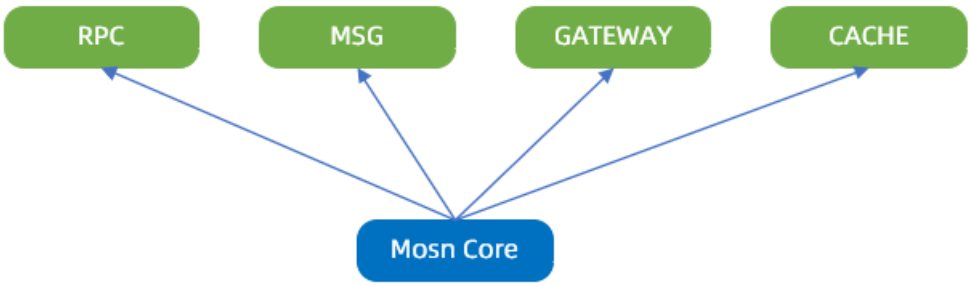
- 网络代理具备的基础能力。
- XDS（Extended Discovery Service） 等云原生能力。

SOFAMosn 主要模块图



业务支持

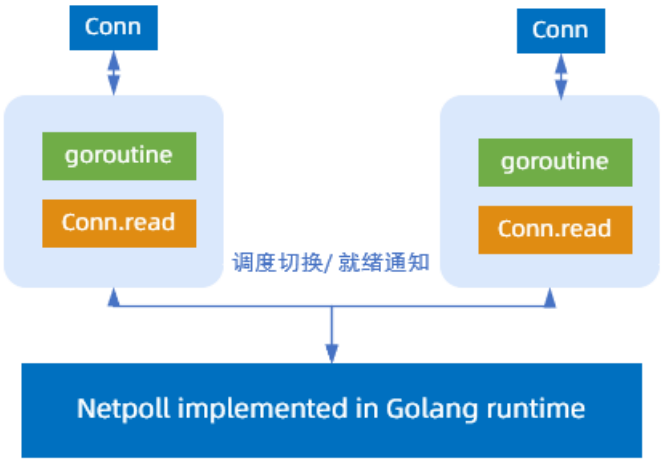
SOFAMosn 作为底层的高性能安全网络代理，支撑的业务场景包括：RPC、MSG、GATEWAY 等。



IO 模型

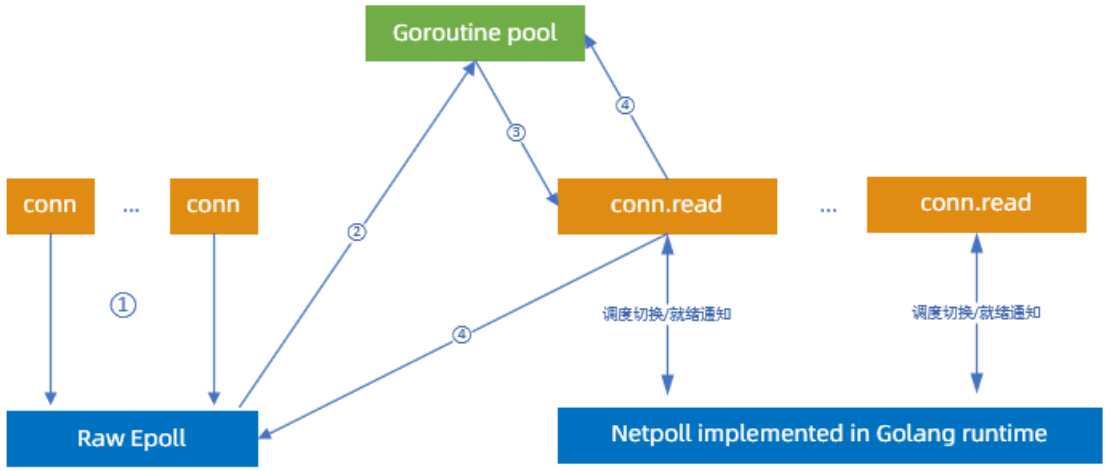
SOFAMosn 支持两种 IO 模型：

- **Golang 经典模型**：在蚂蚁集团内部的落地场景，连接数不是瓶颈，都在几千或者上万的量级，蚂蚁集团选择了 Golang 经典模型 goroutine-per-connection。



  模型缺陷：协程数量与连接数量成正比，大链接场景下，协程数量过多，存在以下开销：
  - Stack 内存开销
  - Read buffer 开销
  - Runtime 调度开销

- **RawEpoll 模型**：也就是 Reactor 模式，即 I/O 多路复用（I/O multiplexing） + 非阻塞 I/O（non-blocking I/O）模式。对于接入层和网关有大量长连接的场景，更加适合于 RawEpoll 模型。



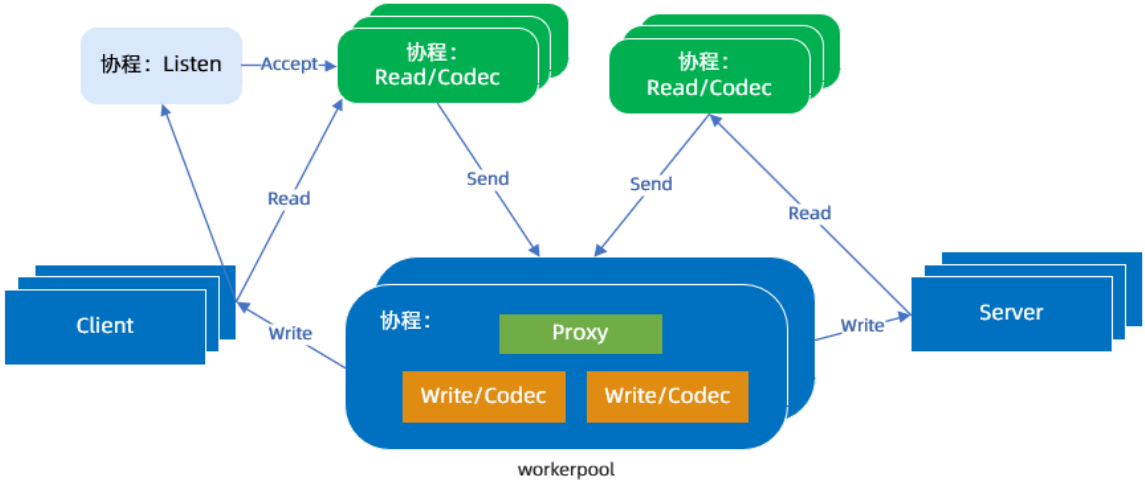
步骤说明：

1. 建立连接：

   向 Epoll 注册 oneshot 可读事件监听。此时不允许有协程调用 `conn.read`，以免与 `runtime Netpoll` 冲突。

2. 可读事件到达，从 goroutine pool 挑选一个协程进行读事件处理。由于使用的是 oneshot 模式，该 fd 后续可读事件不会再触发。
3. 请求处理过程中，协程调度与经典 Netpoll 模式一致。
4. 请求处理完成，将协程归还给协程池，同时将 fd 重现添加到 RawEpoll 中。

协程模型



> **说明**
>
> - 一个 TCP 连接对应一个 Read 协程，执行收包和协议解析。
> - 一个请求对应一个 Worker 协程，执行业务处理、Proxy 和 Write 逻辑。
> - 在常规模型中，一个 TCP 连接有 Read/Write 两个协程，蚂蚁团队取消了单独的 Write 协程，让 workerpool 工作协程代替它，减少了调度延迟和内存占用。

能力扩展

能力扩展主要包括下述几个方面：

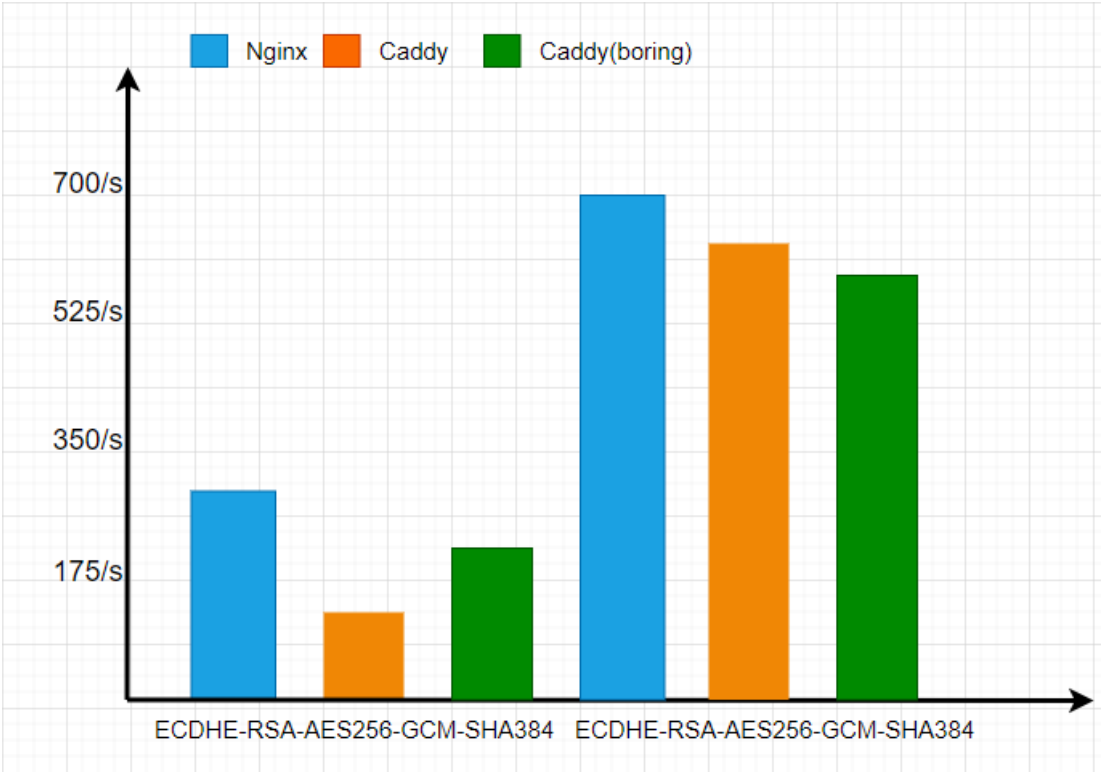
- **协议扩展**：SOFAMosn 通过使用统一的编、解码引擎，以及编、解码器核心接口，提供协议的 plugin 机制。支持下述协议：
  - SOFARPC
  - HTTP1.x/HTTP2.0
  - Dubbo
- **NetworkFilter 扩展**：SOFAMson 通过提供 Network Filter 注册机制，以及统一的 packet read/write filter 接口，实现了 Network filter 扩展机制，当前支持下述功能。
  - TCP proxy
  - Fault injection
- **StreamFilter 扩展**：SOFAMosn 通过提供 stream filter 注册机制，以及统一的 stream send/receive filter 接口，实现了 Stream filter 扩展机制，支持下述功能。
  - 流量镜像
  - RBAC 鉴权

TLS 安全链路

作为金融科技公司，资金安全是最重要的一环，链路加密又是其中最基础的能力。在 TLS 安全链路上，蚂蚁团队进行了大量的调研测试。测试结果显示：

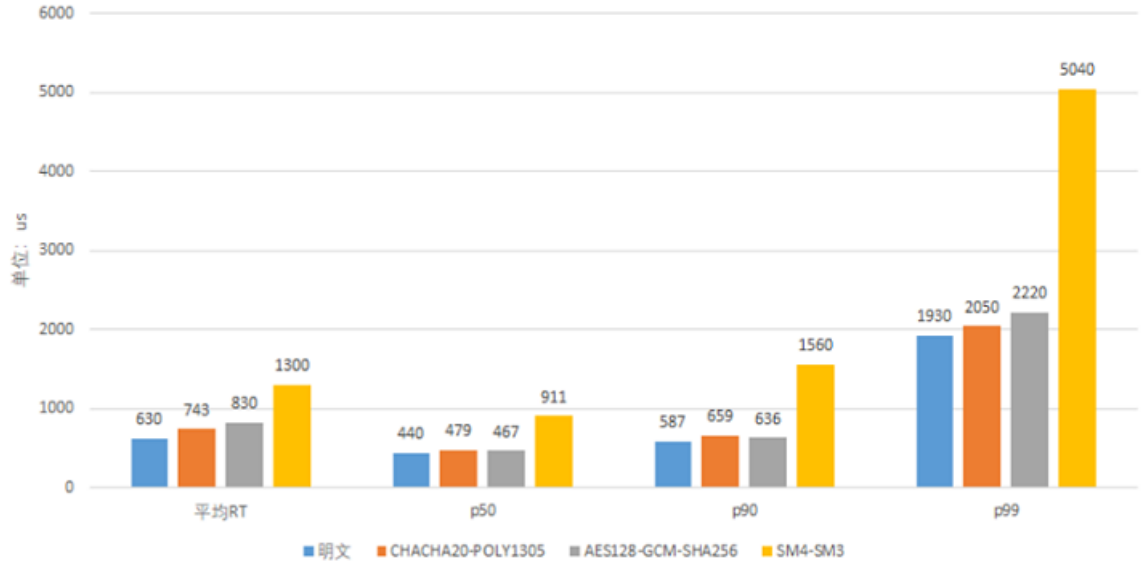
- 原生 Go 的 TLS 经过了大量的汇编优化，在性能上是 Nginx（OpenSSL）的 80%。
- Boring 版本的 Go，使用 CGO 调用 BoringSSL，因为 CGO 的性能问题， 该版本并不占优势。

所以，蚂蚁团队最后选择了原生 Go 的 TLS，相信 Go Runtime 团队后续会有更多的优化，蚂蚁团队也会有一些优化计划。



> **说明**
>
> - Go 在 RSA 上没有太多优化，Go-boring（CGO）的能力是 Go 的 1 倍。
> - p256 在 Go 上有汇编优化，ECDSA 优于 Go-boring。
> - 在 AES-GCM 对称加密上，Go 的能力是 Go-boring 的 20 倍。
> - 在 SHA、MD 等 HASH 算法上，也有对应的汇编优化。

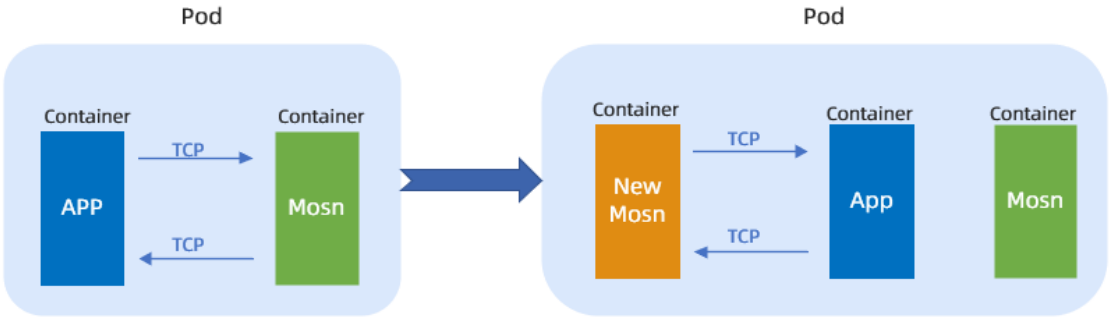
为了满足金融场景的安全合规，蚂蚁团队同时也对国产密码进行了开发支持，这个是 Go Runtime 所没有的。相比国际标准 AES-GCM，目前的性能有大概 50% 的差距，蚂蚁团队已经有了后续的一些优化计划，敬请期待。



平滑升级能力

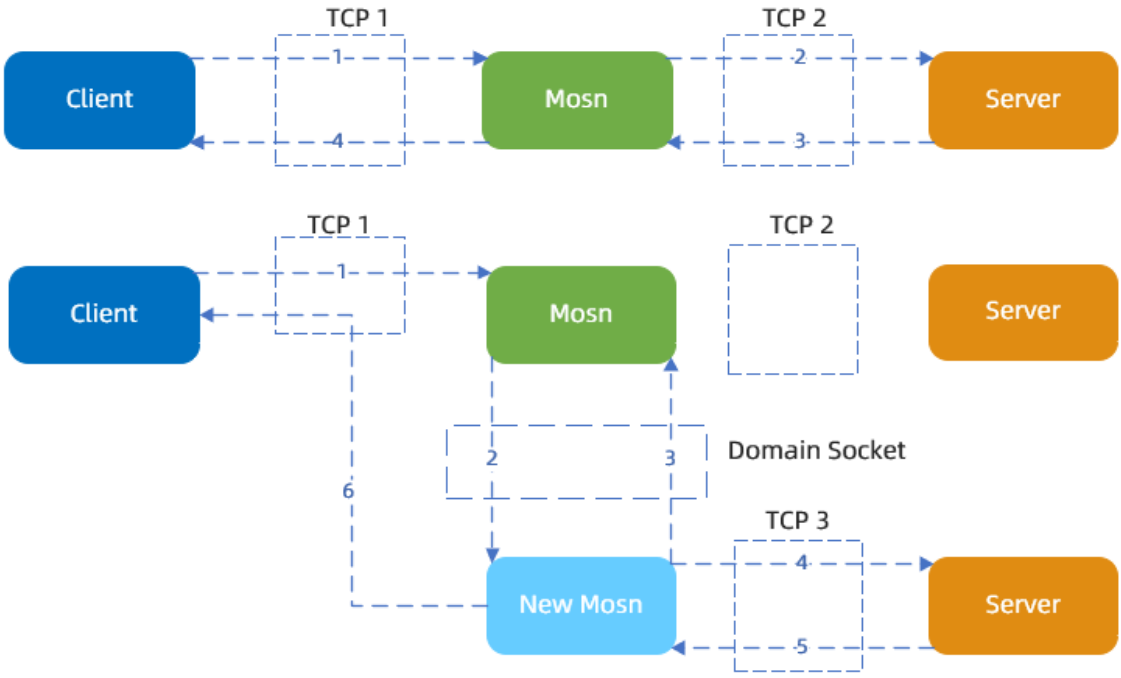
为了让 SOFAMosn 的发布对应用无感知，蚂蚁团队开发了平滑升级方案，该方案类似 Nginx 的二进制热升级能力，最大的区别是 SOFAMosn 老进程的连接不会断，会迁移给新的进程，包括底层的 socket FD 和上层的应用数据。这样可以保证整个二进制发布过程中，业务不受损，对业务无感知。除了支持 SOFARPC、Dubbo、消息等协议，还支持 TLS 加密链路的迁移。
平滑升级能力主要包括下述几个方面的内容：

- **容器升级**：主要流程包括下述几个方面。

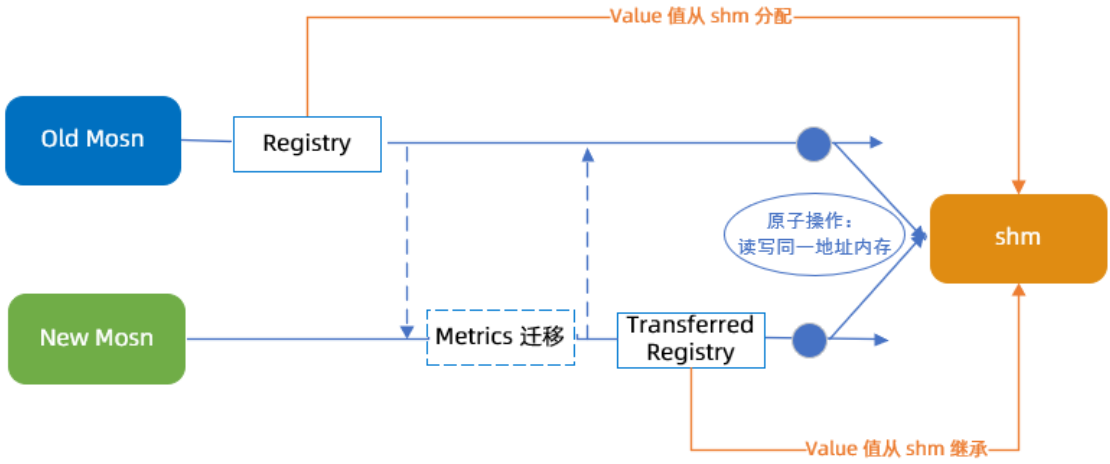


i. 先注入一个新的 SOFAMosn。
ii. 通过共享卷的 UnixSocket 去检查是否存在老的 SOFAMosn。
iii. 如果存在老的 SOFAMosn，就和老的 SOFAMosn 进行连接迁移，然后老的 SOFAMosn 退出。

- **SOFAMosn 的连接迁移**：连接迁移的核心是内核 Socket 的迁移和应用数据的迁移。连接不断，且对用户无感知。



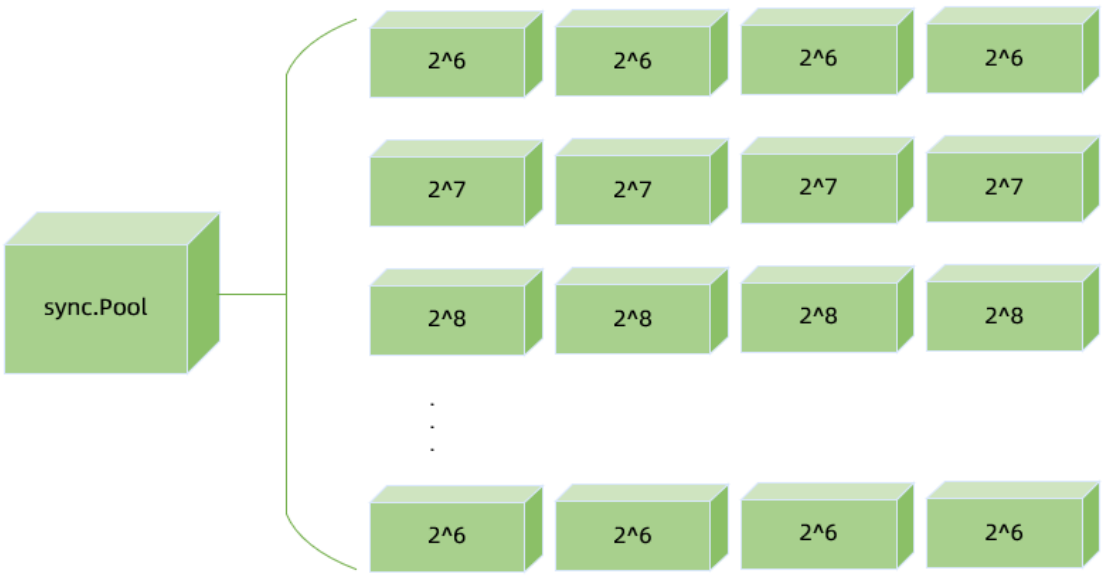
- **SOFAMosn 的 Metric 迁移**：蚂蚁团队使用了共享内存来共享新老进程的 Metric 数据，保证在迁移的过程中 Metric 数据也是正确的。



内存复用机制

内存复用机制主要特征如下：

- 基于 sync.Pool。
- Slice 复用使用 Slab 细粒度，提高复用率。
- 常用结构体复用。



当前现状：

- 线上复用率可以达到 90% 以上。
- sync.Pool 还存在一些问题，随着 Runtime 对 sync.Pool 的持续优化，比如 Go 1.13 使用 lock-free 结构减少锁竞争和增加了 victim cache 机制，它在未来会越来越完善。

XDS（UDPA）

支持云原生统一数据面 API，全动态配置更新。其中，XDS 指 Extended Discovery Service。UDPA 指 Universal Data Plane API。

```
{
  "servers":[
    {
      "default_log_path":"stdout",
       "default_log_level": "DEBUG",
      "listeners":[
        {
       "name":"serverListener",
       "address": "127.0.0.1:2046",
       "bind_port": true,
       "log_path": "stdout",
       "filter_chains": [{
         "tls_context":{},
         "filters": [
           {
             "type": "proxy",
         "config": {
           "downstream_protocol": "Auto",
           "upstream_protocol": "Http1",
           "router_config_name":"server_router"

         }
           },
           {
           "type":"connection_manager",
           "config":{
             "router_config_name":"server_router"
```

前期准备

性能压测和优化

在上线前的准备过程中，蚂蚁团队在灰度环境中针对核心应用 cashiercloudtb 进行了大量的压测和优化，为后面的落地打下了坚实的基础。

从线下环境到灰度环境，蚂蚁团队遇到了很多线下没有的大规模场景，比如：

- 单实例数万后端节点，数千路由规则：不仅占用内存，对路由匹配效率也有很大影响。
- 海量高频的服务发布注册：对性能和稳定性有很大挑战。

整个压测优化过程历时五个月，从最初的 CPU 整体增加 20%，RT 每跳增加 0.8 ms, 到最后 CPU 整体增加 6%，RT 每跳增加 0.25 ms，内存占用峰值优化为之前的 1/10。

	整体增加CPU	每跳RT	内存占用峰值
优化前	20%	0.8 ms	2365 M
优化后	6%	0.25 ms	253 M

部分优化措施：

在 6.18 大促时，蚂蚁团队上线了部分核心链路应用，CPU 损耗最多增加 1.7%，有些应用从 Java 迁移到 Go，CPU 损耗还降低了 8% 左右。延迟方面平均每跳增加 0.17 ms，两个合并部署系统全链路增加 5~6 ms，有 7% 左右的损耗。

在单机房上线 SOFAMosn 时，SOFAMosn 在全链路压测下的整体性能表现更好。比如：交易付款时，带 SOFAMosn 比不带 SOFAMosn 的响应时间（RT）降低了 7.5%。

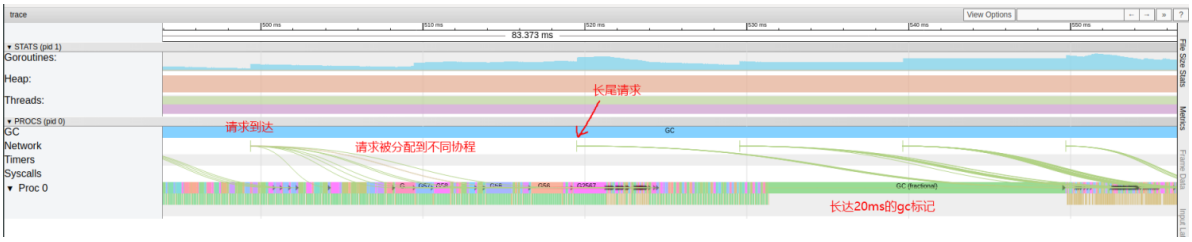
SOFAMosn 所做的大量核心优化和下沉的 Route Cache 等业务逻辑优化，更带来了架构的红利。

Go 版本选择

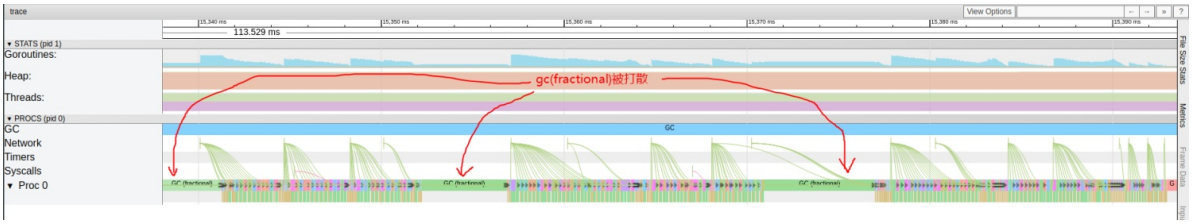
版本的升级都需要做一系列测试，新版本并不都最适合目标场景。该项目最开始使用的版本为 Go 1.9.2，在经过一年迭代之后，蚂蚁团队开始调研当时的最新版 Go 1.12.6，测试验证了新版很多好的优化，也修改了内存回收的默认策略，以便更好地满足项目需求。

- GC 优化，减少长尾请求：新版的自我抢占（self-preempt）机制，将耗时较长的 GC 标记过程打散，来换取更为平滑的GC 表现，减少对业务的延迟影响。

- Go 1.9.2



- Go 1.12.6



- 内存回收策略：Go 1.12 修改了内存回收策略，从默认的 MADV_DONTNEED 修改为了 MADV_FREE。虽然这是一个性能优化，但是在实际使用中，测试显示性能并没有大的提升，却占用了更多的内存，对监控和问题判断有很大的干扰。蚂蚁团队通过 GODEBUG=madvdontneed=1 恢复为之前的策略。在 issue 里也有相关讨论，后续版本可能也会改动这个值。
- runtime: use MADV_FREE on Linux if available

> **runtime: use MADV_FREE on Linux if available**
>
> On Linux, sysUnused currently uses madvise(MADV_DONTNEED) to signal the kernel that a range of allocated memory contains unneeded data. After a successful call, the range (but not the data it contained before the call to madvise) is still available but the first access to that range will unconditionally incur a page fault (needed to 0-fill the range).
>
> A faster alternative is MADV_FREE, available since Linux 4.5. The mechanism is very similar, but the page fault will only be incurred if the kernel, between the call to madvise and the first access, decides to reuse that memory for something else.
>
> In sysUnused, test whether MADV_FREE is supported and fall back to MADV_DONTNEED in case it isn't. This requires making the return value of the madvise syscall available to the caller, so change runtime.madvise to return it.

- 使用 Go 1.12 默认的 MADV_FREE 策略时，HeapInuse = 43 M， 但是 HeapIdle = 600 M，一直不能释放。

```
# runtime.MemStats
# Alloc = 27187704
# TotalAlloc = 1577093705032
# Sys = 709878008
# Lookups = 0
# Mallocs = 24830817578
# Frees = 24830664004
# HeapAlloc = 27187704
# HeapSys = 654082048
# HeapIdle = 610672640
# HeapInuse = 43409408
# HeapReleased = 583999488
# HeapObjects = 153574
# Stack = 17006592 / 17006592
# MSpan = 1074816 / 8437760
# MCache = 3472 / 16384
# BuckHashSys = 2324010
# GCSys = 25632768
# OtherSys = 2378446
# NextGC = 47365264
# LastGC = 1567823909422965541
# PauseNs = [33239 46658 29090 41939 25150 32649 32959 33759 3
```

Go Runtime Bug 修复

在前期灰度验证时，SOFAMosn 线上出现了较严重的内存泄露，一天泄露了 1 G 内存，最终排查显示，是 Go Runtime 的 Writev 实现存在缺陷，导致 Slice 的内存地址被底层引用，GC 不能释放。
蚂蚁团队给 Go 官方提交了 Bugfix，已合入 Go 1.13 最新版，参见 internal/poll: avoid memory leak in Writev。

```
54                  if fd.iovecs == nil {
55                          fd.iovecs = new([]syscall.Iovec)
56                  }
57                  *fd.iovecs = iovecs // cache
58
59                  var wrote uintptr
60                  wrote, err = writev(fd.Sysfd, iovecs)
61                  if wrote == ^uintptr(0) {
62                          wrote = 0
63                  }
64                  TestHookDidWritev(int(wrote))
65                  n += int64(wrote)
66                  consume(v, int64(wrote))
67  +               for i := range iovecs {
68  +                       iovecs[i] = syscall.Iovec{}
69  +               }
```

4.2.7.2. 服务网格最佳实践之 RPC

服务网格（Service Mesh）是蚂蚁集团下一代架构的核心，在蚂蚁集团当前的体量下，将现有的 SOA 体系快速演进至 Service Mesh 架构，犹如给奔跑的火车换轮子。本文以 RPC 层面的设计和改造方案为中心，分享蚂蚁集团在双十一大促面临大流量挑战时，核心应用如何将现有的微服务体系平滑过渡到 Service Mesh 架构下，并降低大促成本。

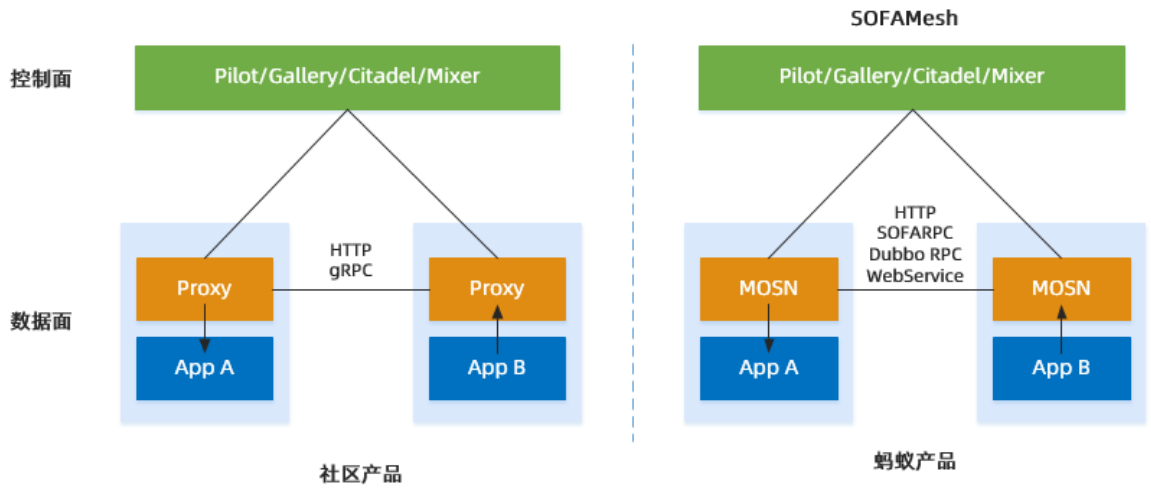
Service Mesh 简介

与社区 Service Mesh 相比，蚂蚁 Service Mesh 的结构也分为下述两个部分：

- **控制面**：名为 SOFAMesh。未来会以更加开放的姿态参与到 Istio 中。

- **数据面**：名为 MOSN，支持 HTTP、SOFARPC、Dubbo、WebService。下文主要讲述的即数据面的落地。

社区 Service Mesh 架构和蚂蚁集团 Service Mesh 架构对比图



为什么要 Service Mesh?

Service Mesh 解决了在 SOA（Service-Oriented Architecture） 下面存在的亟待解决的如下问题：

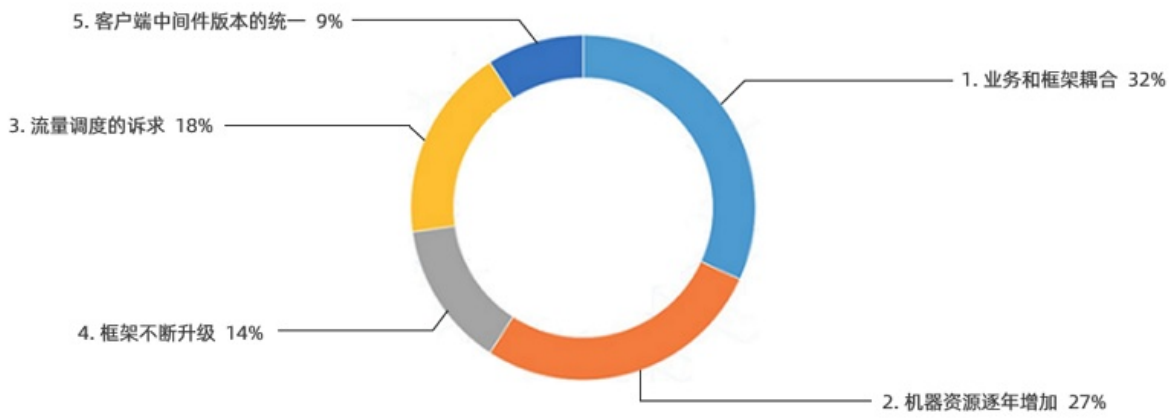
- 基础架构和业务研发耦合问题
- 业务透明的稳定性与高可用性等问题

使用 Service Mesh 前状态

在没有 Service Mesh 之前，整个 SOFAStack 技术演进的过程中，主要存在下述的问题：

- 框架和业务过于耦合。
- 一些 RPC 层面的需求，比如：流量调度、流量镜像、灰度引流等，需要投入更多开发资源进行支持。
- 需要业务方来升级对应的中间件版本。

主要问题示例如下：



- **框架和业务耦合**：升级成本高，很多需求由于在客户端无法推动，需要在服务端提供相应的功能，方案不够优雅。
- **机器资源逐年增加**：如果不增加机器，很难应对双十一的巨大流量。
- **流量调度诉求**：流量调拨、灰度引流、蓝绿发布、AB Test 等新的诉求不容易满足。

- **框架升级诉求**：在基础框架准备完成后，对于新功能，如果用户的 API 层不升级，无法确定是否能兼容旧版本。
- **版本不统一**：线上客户端框架版本不统一。

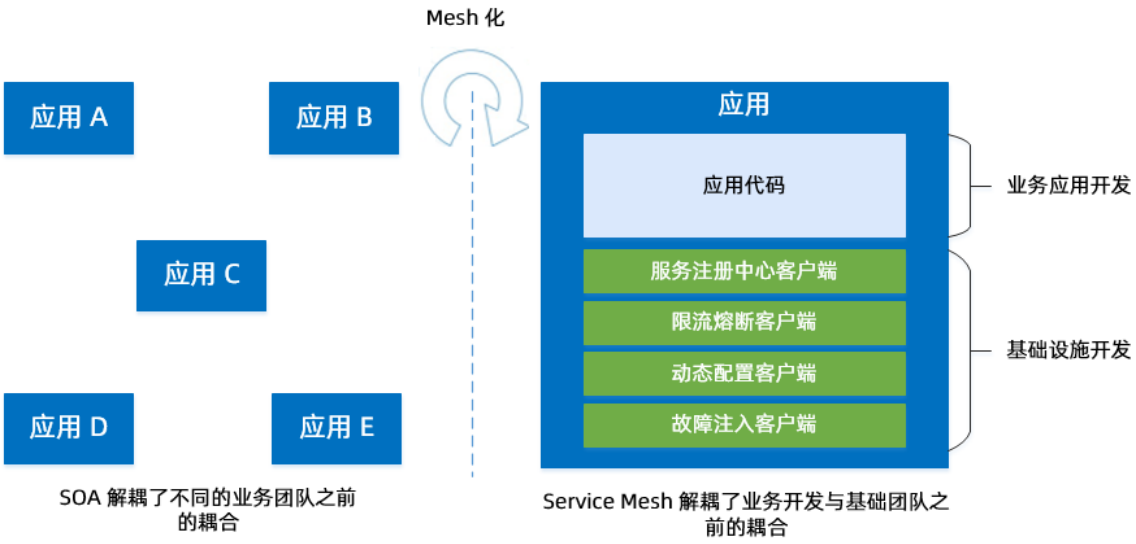
在 SOA 的架构下，负责业务的团队和负责基础设施的团队，合作现状如下：

- **业务团队之间可以实现解耦**：负责业务的团队都可以独立地去负责一个或者多个业务，这些业务的升级维护也不需要其他团队的介入。SOA 做到的是团队之间可以按照接口的契约来解耦。
- **基础设施团队和业务团队耦合严重**：长期以来，基础设施团队要推动很多业务时，都需要服务团队进行紧密的配合，例如帮助升级 JAR 包等。这种耦合会带来各种问题，例如线上客户端版本的不一致、升级成本高等。

使用 Service Mesh 后状态

Service Mesh 能够将基础设施下沉，让基础设施团队和业务团队解耦，从而允许各自更加快步地往前跑。

解耦前后对比图

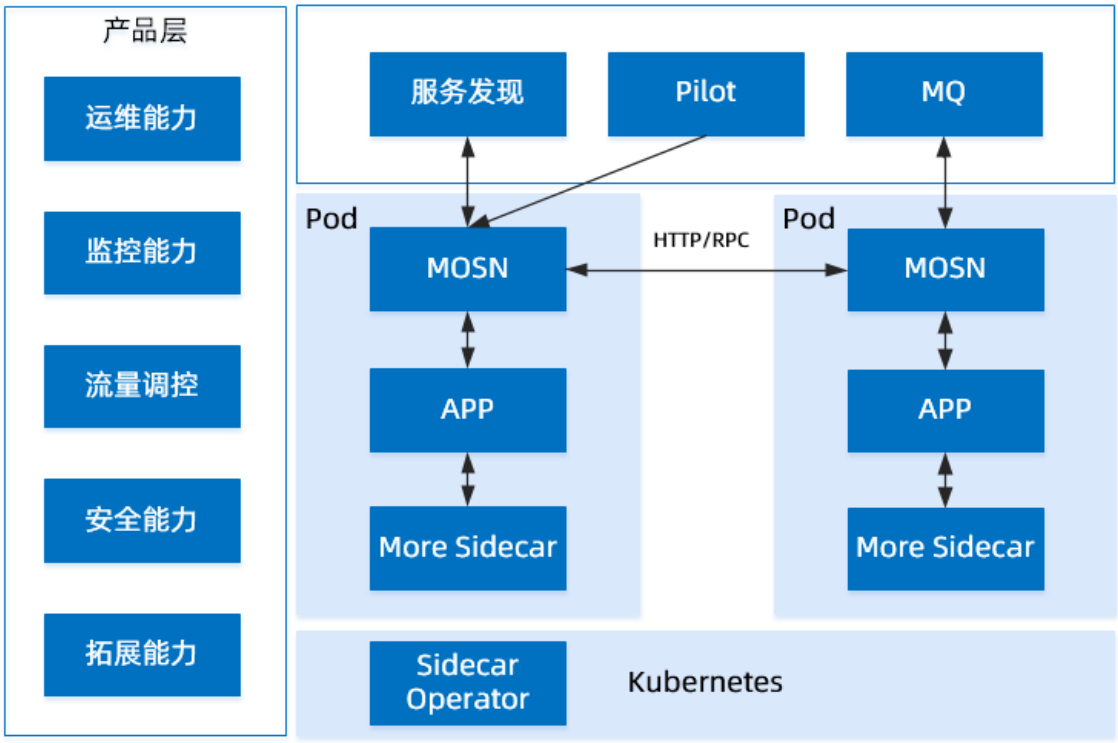


Service Mesh 解决方案

选型思考

- **问题**：要实现 Service Mesh 的预期效果，首先要面临技术选型。技术选型主要面临下述几个问题。
  - **开源/自研**：由于存在自有协议和历史遗留问题，不适合全部迁移到 Envoy。
  - **SDK/透明劫持**：透明劫持的运维和可监控性不好、性能不高、风险不太可控。
- **最终选型方案**：自研数据面 + 轻量 SDK，也就是 MOSN（Modular Open Smart Network-Proxy）。

总体目标架构



MOSN 目前支持下述组件：

- Pilot
- 蚂蚁的服务发现组件 SOFARegistry
- 蚂蚁的消息组件 SOFAMQ
- 数据库组件

MOSN 在产品层已向开发者提供下述能力：

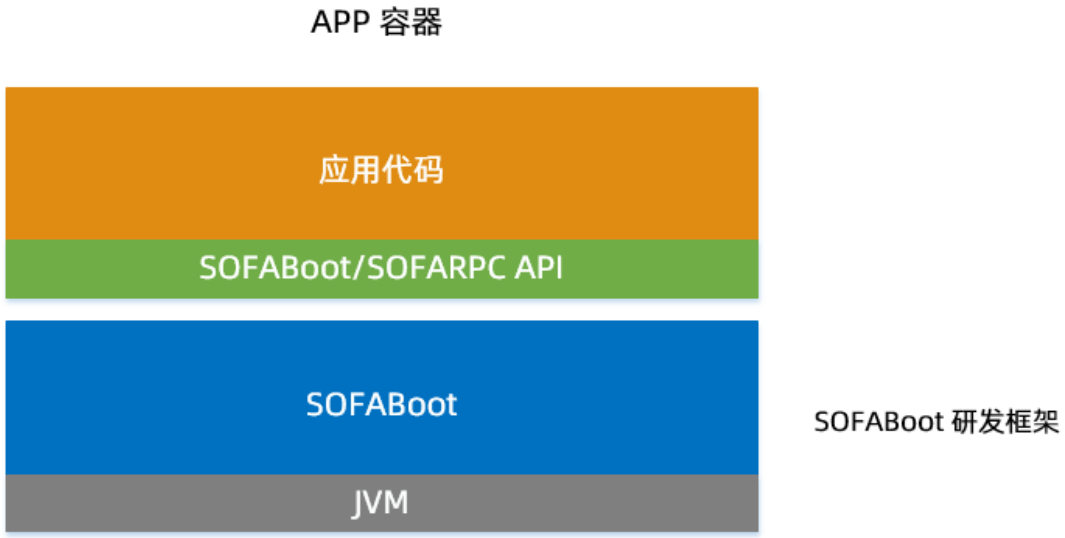
- 运维
- 监控
- 安全

要实现上述状态，我们要解决业务的三大诉求：

- 框架升级方案
- 容器替换方案
- MOSN 升级方案

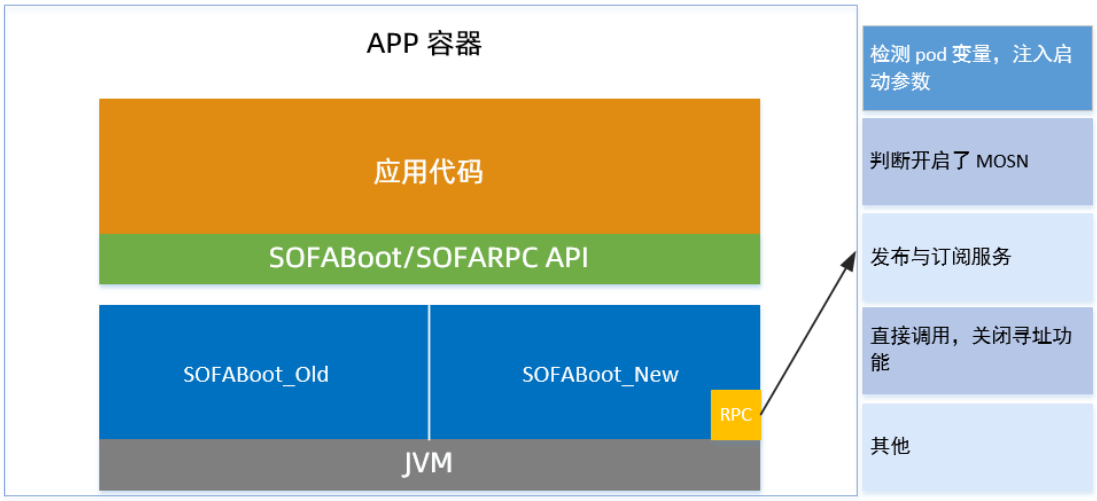
框架升级方案

- **升级前，线上现状**：
  - 应用代码和框架有一定程度的解耦。
  - 用户面向的是一个 API。
  - 代码需要打包后，在 SOFABoot 中运行。



- **升级方案**：主要步骤示例如下。
  i. 评估后，在风险可控的情况下，直接升级底层的 SOFABoot。
  ii. 让 RPC 去检测一些信息，来确定当前 Pod 是否需要开启 MOSN 的能力。

iii. 通过检测 PaaS 传递的容器标识，确定 MOSN 开启状态，将发布和订阅传给 MOSN，直接完成调用，不再寻址。

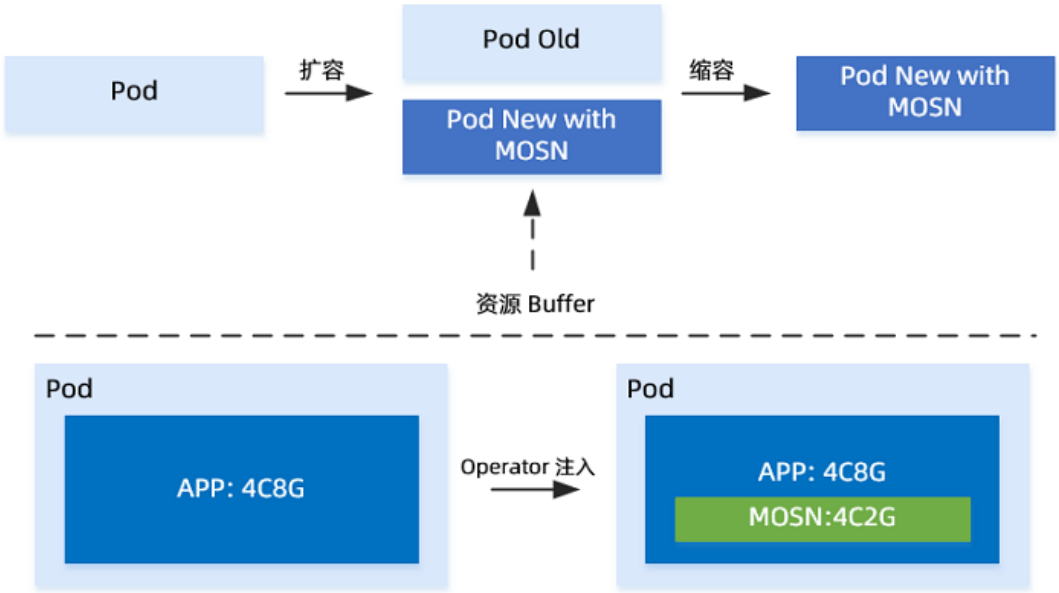


- **优缺点**：
  - 优点：通过批量的运维操作，直接修改了线上的 SOFABoot 版本，使得现有的应用具备了 MOSN 的能力。
  - 缺点：
    - 该升级方案运行时需要关闭流量等辅助操作。
    - 不具备平滑升级的能力。
    - 直接和业务代码强相关，不适合长期操作。
- **不采用社区流量劫持方案的考虑**：
  - iptables 在规则配置较多时，性能下滑严重。
  - 它的管控性和可观测性不好，出了问题比较难排查。
  - Service Mesh 从初期就把蚂蚁集团线上系统全量切换 Mesh 作为目标，对性能和运维的要求非常高，不能接受业务有损或者资源消耗率大幅度上升。

容器替换方案

框架升级方案，只是解决了可以做，而并不能做得好，更没有做得快，面对线上数十万带着流量的业务容器， 如何实现这些容器的快速稳定接入？在流量很大情况下，传统的替换接入需消耗大量接入成本，于是，蚂蚁团队选择了原地接入。

传统接入和原地接入对比图



传统接入和原地接入对比：

- **传统接入**：
  - 升级操作需要有一定的资源 Buffer，然后批量的进行操作，通过替换 Buffer 的不断移动，来完成升级。
  - 要求 PaaS 层有非常多的 Buffer，需要更多的钱来买资源。
  - CPU 利用率低。
- **原地接入**：
  - 通过 Paas 层，Operator 操作直接在现有容器中注入，并原地重启，在容器级别完成升级。升级完成后，该 Pod 就具备了 MOSN 的能力。
  - 提高了 CPU 利用率。
  - 是一个类似超卖的方案，看上去分配了 CPU 和内存，实际上，基本没增加。

MOSN 升级方案

容器替换方案完成后，我们要面临第三个问题：由于是大规模的容器，所以 MOSN 在开发过程中，势必会存在一些问题，MOSN 出现问题，如何升级？线上几十万容器升级一个组件的难度是很大的，因此，在版本初期就需考虑到 MOSN 的升级方案。

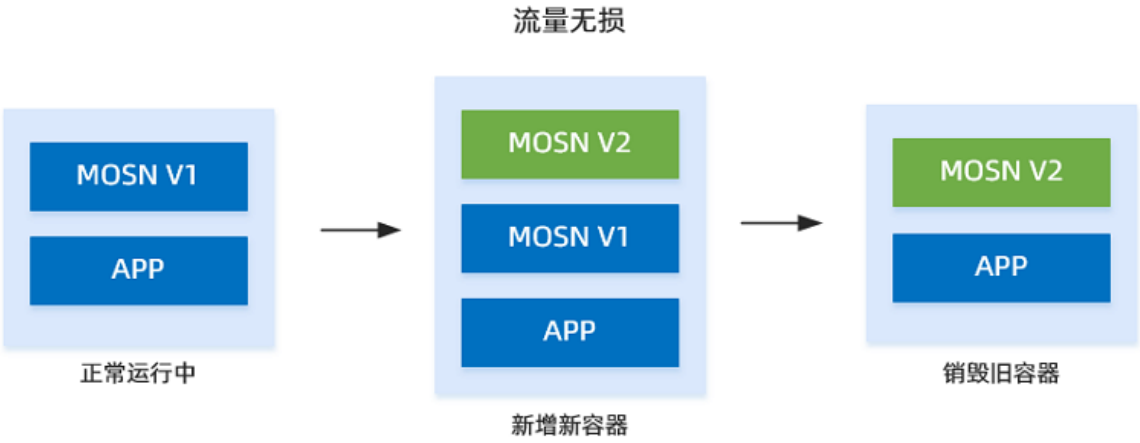
简版升级方案

- **方案思路**：销毁容器，用新容器重建，示例如下。

- **方案缺陷**：
  - 在容器数量很多时，运维成本不可承受。
  - 销毁容器后，如果重建速度不够快，就可能会影响业务的容量，造成业务故障。

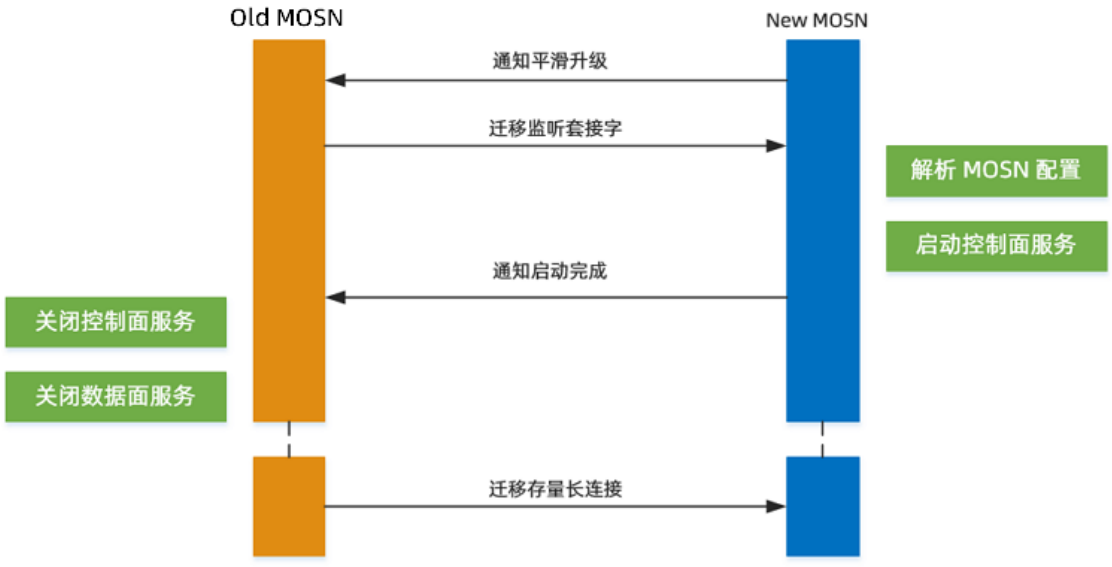
无损升级方案

为了解决简版升级方案的不足，在 MOSN 层面，和 PaaS 一起，开发了无损流量升级方案，示例如下：



- **方案思路**：
  - MOSN 会感知自己的状态。
  - 新的 MOSN 启动时，会通过共享卷的 Domain Socket 来检测同 Pod 是否已有老的 MOSN 在运行，如果有，则通知原有 MOSN 进行平滑升级操作，流程如下：
    a. New MOSN 通知 Old MOSN，进入平滑升级流程。
    b. Old MOSN 把服务的 Listen Fd 传递给 New MOSN，New MOSN 接收到 Fd 之后启动， 此时 Old 和 New MOSN 都正常提供服务。
    c. New MOSN 通知 Old MOSN 关闭 Listen Fd，然后开始迁移存量的长连接。
    d. Old MOSN 迁移完成，销毁容器；

- **方案示例如下**：

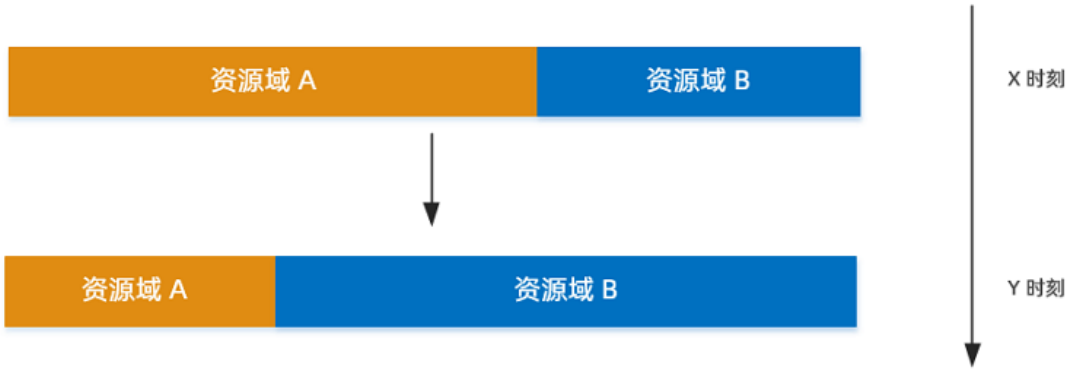


- **方案优势**：线上做任意的 MOSN 版本升级，都不影响老业务。

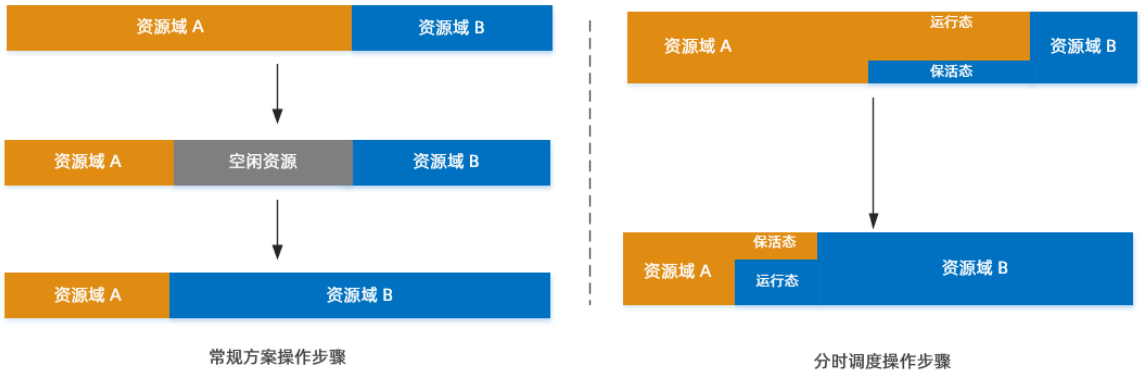
MOSN 应用之分时调度

技术的变革通常并不是技术本身的诉求，而是业务的诉求，是场景的诉求。很少有人会为了升级而升级，为了革新而革新。通常，技术受业务驱动，也反过来驱动业务。
在阿里经济体下，淘宝直播、实时红包、蚂蚁森林等各种活动的不断扩张，给技术带了复杂的场景考验。这种场景，对于业务来说就意味着代码已经最优化而流量却几乎无法支撑，解决办法就是扩容增加机器，而更多的机器意味着更多的成本付出。对于这样的情况，Service Mesh 是一个很好的解决办法。
通过和 JVM 及系统部的配合，利用进阶的分时调度实现灵活的资源调度，可以实现更好的效果且不必加机器资源，说明如下：

- **资源需求**：假设有如下两个大的资源池的资源需求。
  - 在 X 点的时候，资源域 A 需要更多的资源。
  - 在 Y 点的时候，资源域 B 需要更多的资源。
  - 资源总量不得增加。



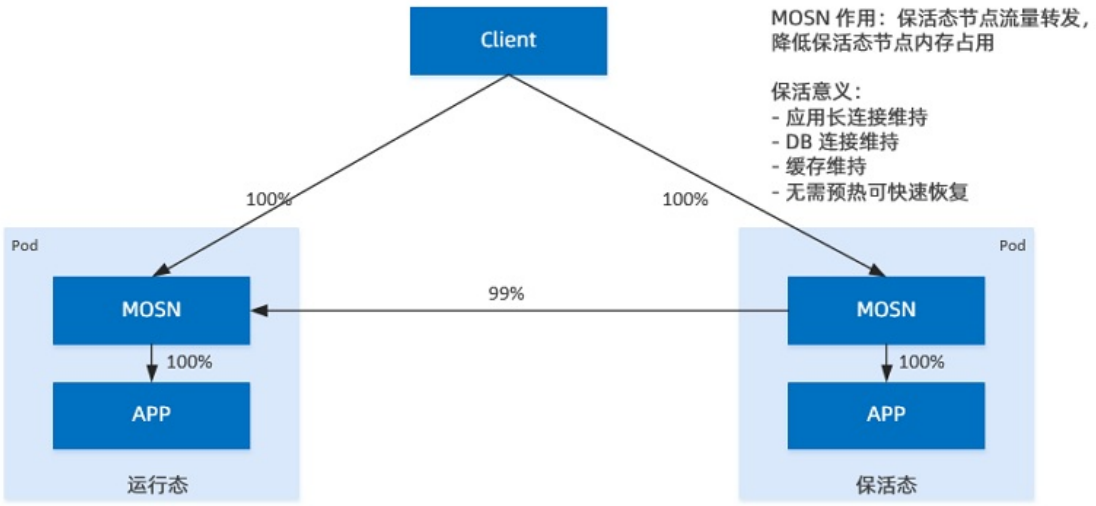
- **借调方案**：上述资源需求需要通过借调机器来完成，借调方案有下述 2 种。



  - **常规方案**：
    - 先释放资源，然后销毁进程，接着重建资源，最后启动资源域 B 的资源。
    - 该过程对于大量的机器是重型操作，而且变更就是风险，关键时候做这种变更，很有可能带来衍生影响。

- **分时调度方案**：MOSN 的解决思路如下。
  - 有一部分资源一直通过超卖运行着两种应用。
  - X 时刻：资源域 A 处于运行态，资源域 B 处于保活态。资源域 A 通过 MOSN 将流量全部转走，应用的 CPU 和内存就被限制到非常低的情况，大概保留 1% 的能力。这样操作，机器依然可以预热，进程也不停。
  - Y 时刻：资源域 B 处于运行态，资源域 A 处于保活态。在需要比较大的资源调度时，通过推送开关，可以打开资源限制，包活状态取消等。资源域 B 瞬间可以满血复活。而资源域 A 此时进入 X 时刻状态，CPU 和内存被限制。
  - MOSN 以一个极低的资源占用完成流量保活的能力，使得资源的快速借调成为可能。

MOSN 分时调度细节图



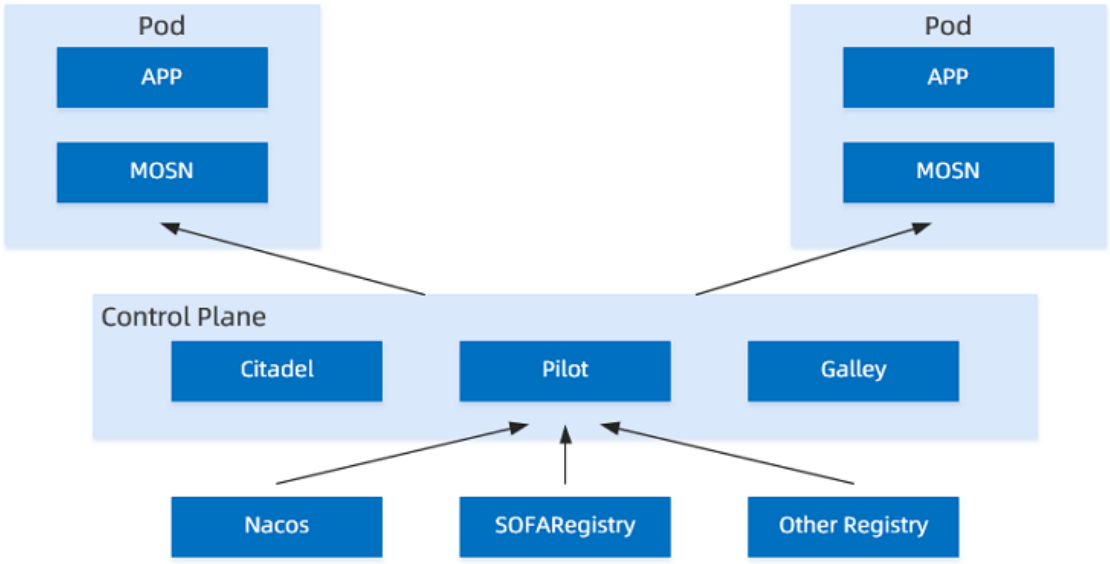
Service Mesh 未来之展望

Service Mesh 在蚂蚁集团经过两年的沉淀，最终经受住了双十一的检验。在双十一活动中，蚂蚁集团覆盖了数百个双十一交易核心链路，MOSN 注入的容器数量达到了数十万，双十一当天处理的 QPS 达到了几千万，平均处理 RT < 0.2 ms，MOSN 本身在大促中间完成了数十次的在线升级，基本上达到了设计的预期效果，初步完成了基础设施和业务第一步的分离，见证了 Mesh 化之后基础设施的迭代速度。

不论何种架构，软件工程没有银弹。架构设计与方案落地总是一种平衡与取舍，目前还有一些 Gap 需要继续努力去攻克，但是，蚂蚁人相信云原生是远方和未来。

经过两年的探索和实践，蚂蚁集团积累了丰富的经验。Service Mesh 可能会是云原生下最接近 “银弹” 的那一颗子弹，未来 Service Mesh 会成为云原生下微服务的标准解决方案，接下来蚂蚁集团将和阿里集团一起深度参与到 Istio 社区中去，和社区一起把 Istio 打造成 Service Mesh 的事实标准。

期待的 Service Mesh 架构



参考资料

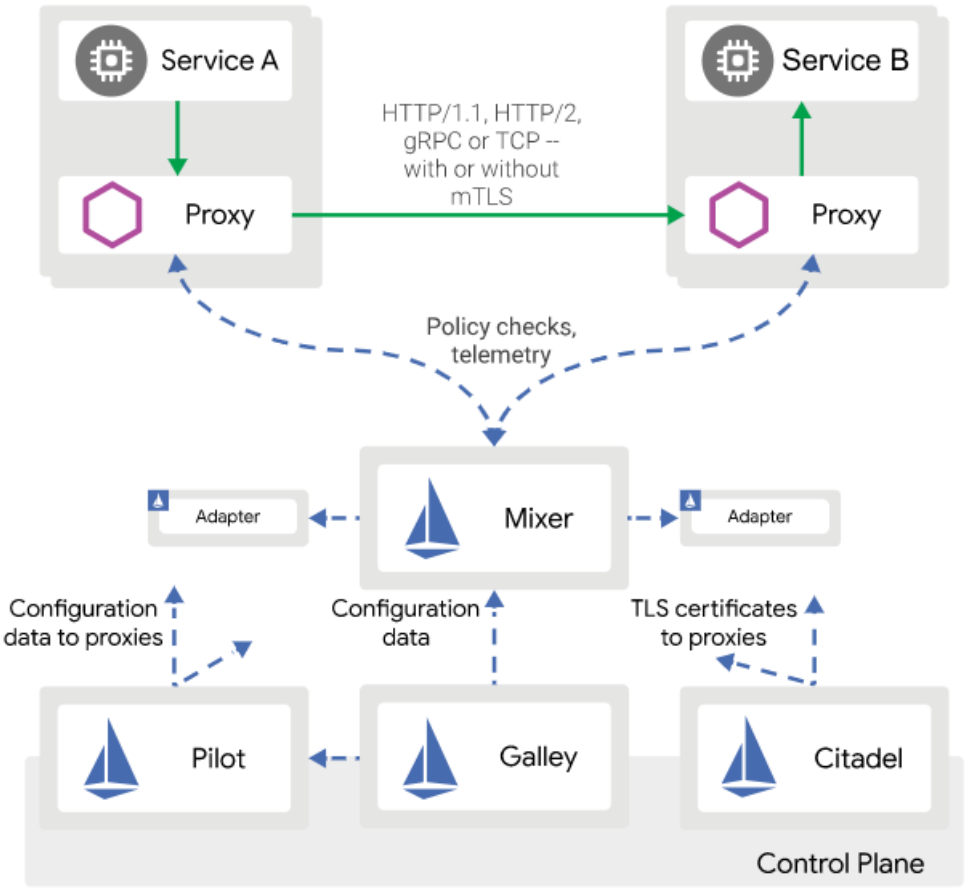
- SOFAStack 官网
- Envoy 仓库
- SOFAStack Github 仓库
- MOSN 仓库
- SOFARegistry 仓库
- SOFABoot 仓库

4.2.7.3. 服务网格最佳实践之控制面

Service Mesh 是蚂蚁集团下一代架构的核心，本文主要分享在蚂蚁集团当前的体量下，控制面平稳支撑大规模 Sidecar 的落地实践。主体部分将聚焦控制面核心组件 Pilot 和 Citadel，分享蚂蚁金服双十一控制面如何管理并服务好全站 Sidecar。

Pilot 落地实践

在开始落地实践部分之前，先引入 Istio 的架构图：



- 出于性能等方面的综合考虑，在落地过程中，蚂蚁团队将控制面的组件精简为 Pilot 和 Citadel 两个组件，不使用因性能问题争议不断的 Mixer，不引入 Galley 来避免多一跳的开销。
- 在架构图中，控制面组件 Pilot 是与 Sidecar 交互最重要的组件，负责配置转化和下发，直面 Sidecar 规模化带来的挑战。这也是双十一大促中，控制面最大的挑战。

在生产实践中，规模化问题是一个组件走向生产可用级的必经之路。

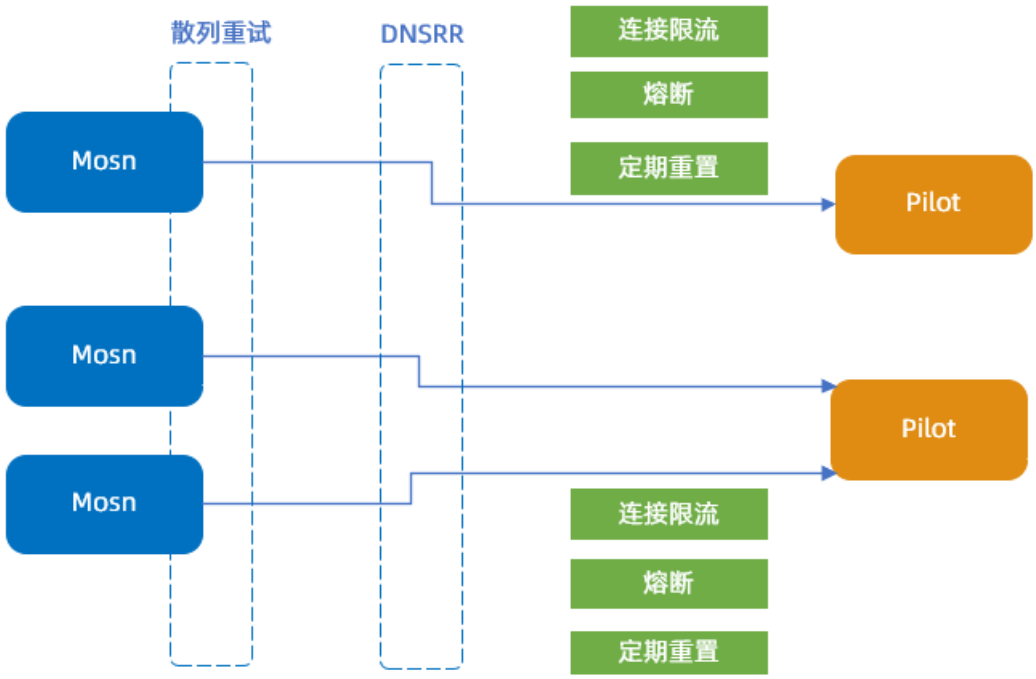
稳定性增强

从功能实现上来看，Pilot 提供的服务能力可以总结为：Pilot 是一个 Controller + gRPC Server 的服务，通过 List/Watch 各类 K8S 资源进行整合计算，生成 XDS 协议的下发内容，并提供 gRPC 接口服务。
本文将把关注点放在 gRPC 接口服务这个环节：如何保证接口服务支撑大规模 Sidecar 实例，是规模化的一道难题。

负载均衡

要具备规模化能力，横向扩展能力是基础。蚂蚁团队采用常用的 DNSRR 方案来访问 Pilot，Sidecar 随机访问 Pilot 实例。由于是长连接访问，所以在扩容时，原有的连接没有重连，会造成负载不均。为解决这个问题，蚂蚁团队配合 Sidecar 散列重连逻辑，给 Pilot 增加了下述功能，以避免产生连接风暴。

- 连接限流
- 熔断
- 定期重置



- **连接限流**：为了降低大量 MOSN 同时连接同一个 Pilot 实例的风险，在 gRPC 首次连接时，Pilot 增加基于令牌桶方案的流控能力，控制新连接的处理响应，并将等待超时的连接主动断连，等待 Sidecar 下一次重连。
- **熔断**：基于使用场景的压测数据，限制单实例 Pilot 同时可服务的 Sidecar 数量上限，超过熔断值的新连接会被Pilot 主动拒绝。
- **定期重置**：为了实现负载均衡，对于已经存在的旧连接，蚂蚁团队选择让 Pilot 主动断开连接，不过断开连接的周期需要考虑错开大促峰值、退避扩缩容窗口等问题，需要按具体的业务场景来确定。
- **Sidecar 散列重连**：该功能涉及 Client 端的配合，让 Sidecar 重连 Pilot 时，采用退避式重试逻辑，避免对 DNS 和 Pilot 造成负载压力。

性能优化

规模化的另一道难题是如何保证服务的性能。在 Pilot 的场景，最受关注的是配置下发的时效性。性能优化离不开细节，其中部分优化是通用的，也有部分优化是面向业务场景定制的，接下来会介绍一下蚂蚁团队优化的一些细节点。

- **首次请求优化**：社区方案里 Pilot 是通过 Pod.Status 来获取 Pod 的 IP 信息，在小集群的测试中，这个时间基本秒级内可以完成。大集群生产环境显示 Status 的更新事件时间较慢，甚至出现超过 10s 以上的情况，而且延迟时间不稳定，会增加 Pilot 首次下发的时延。通过与基础设施 K8s 打通，由 PaaS 侧将 Pod 分配到的 IP 直接标记到Pod.Annotation 上，从而实现在第一次获取 Pod 事件时，就可以获取到 IP，将该环节的时延减少到 0。
- **按需获取 & Custom Resource 缓存**：这是一个面向 DBMesh 业务场景的定制性优化，是基于按需获取的逻辑来实现的。其目的在于解决 DBMesh CR 数量过多、过大导致的性能问题。同时避免 Pilot 由于 List/Watch CR 资源导致 OOM 问题。Pilot 采用按需缓存和过期失效的策略来优化内存占用。
- **局部推送**：社区方案中当 Pilot List/Watch 的资源发生变更时，会触发全部 Sidecar 的配置推送，这种方案在生产环境大规模集群下，性能开销是巨大的。例如：如果单个集群有 10 W 以上的 Pod 数量，任何一个 Pod 的变更事件都会触发全部 Sidecar 的下发，这样的性能开销是不可接受的。
- **其他优化**：强管控能力是大促基本配备，蚂蚁团队给 Pilot Admin API 补充了一些额外能力：支持动态变更推送频率、推送限流、日志级别等功能。

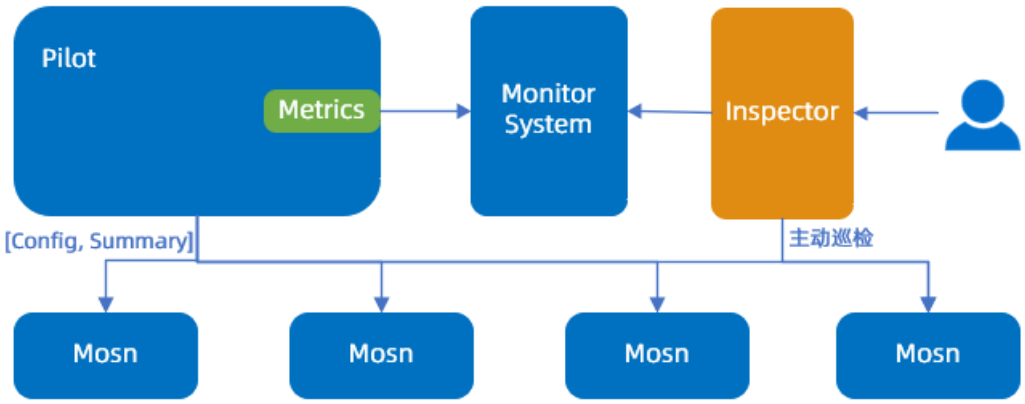
优化的思路也比较简单，如果能够控制下发范围，那就可以将配置下发限制在需要感知变更的 Sidecar 里。为此，蚂蚁团队定义了 ScopeConfig CRD，用于描述各类资源信息与哪些 Pod 相关，这样 Pilot 就可以预先计算出配置变更的影响范围，然后只针对受影响的 Sidecar 推送配置。

监控能力

安全生产的基本要求是具备快速定位和及时止血能力。对于 Pilot 来说，需要关注的核心功能是配置下发能力，该能力有下述几个核心监控指标：

- **下发时效性**：针对下发的时效性，蚂蚁团队在社区的基础上补充完善了部分下发性能指标，如：下发的配置大小分布、下发时延等。
- **配置准确性**：配置准确性的验证是相对比较复杂的，因为配置的准确性需要依赖 Sidecar 和 Pilot 的配置双方进行检验。因此，蚂蚁团队在控制面里引入了 Inspector 组件，用来定位配置巡检，版本扫描等运维相关功能模块。

配置巡检的流程，示例如下：



流程说明：

1. Pilot 下发配置时，将配置的摘要信息与配置内容同步下发。
2. MOSN 接收配置时，缓存新配置的摘要信息，并通过 Admin API 暴露查询接口。
3. Inspector 基于控制面的 CR 和 Pod 等信息，计算出对应 MOSN 的配置摘要信息，然后请求 MOSN 接口，对比配置摘要信息是否一致。

由于 Sidecar 的数量较大，Inspector 在巡检时，支持基于不同的巡检策略执行巡检。大体可以分为以下几类：

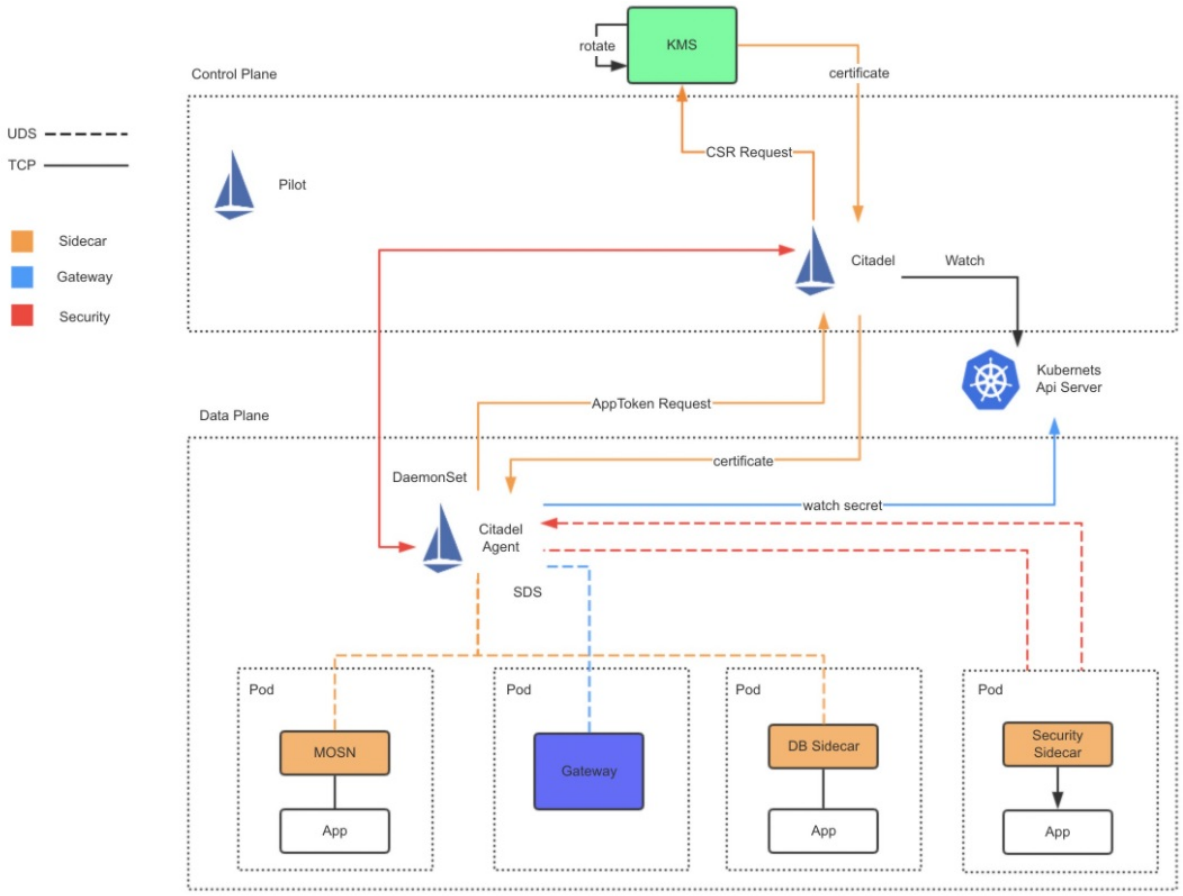
- 周期性自动巡检，一般使用抽样巡检。
- SRE 主动触发检查机制。

Citadel 安全方案

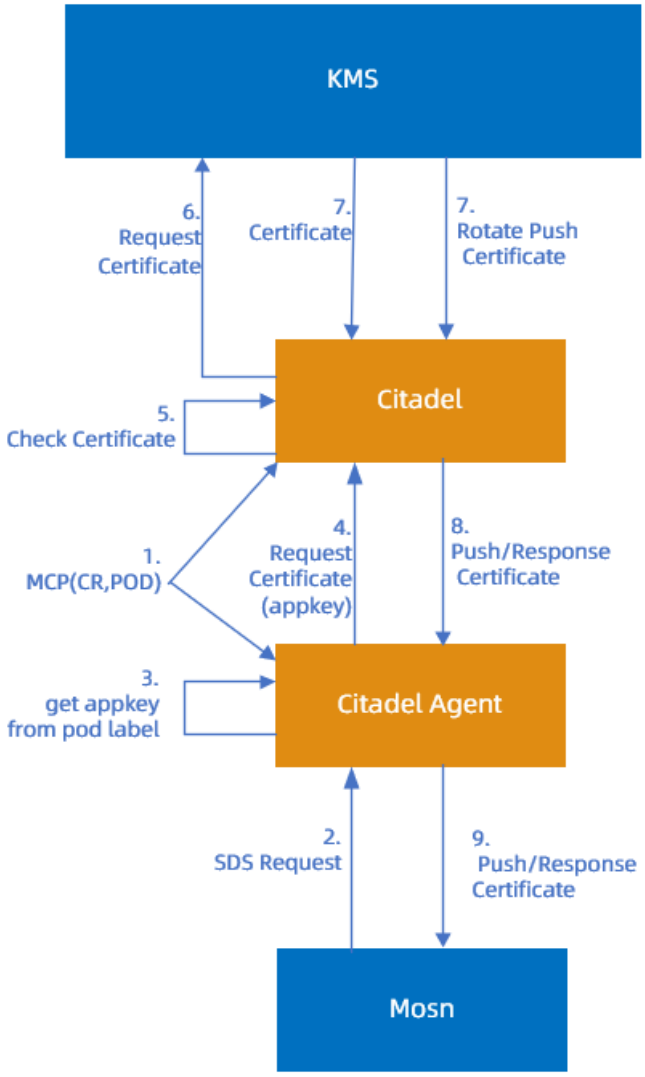
证书方案

Sidecar 基于社区 SDS 方案 （Secret Discovery Service），支持证书动态发现和热更新能力。蚂蚁集团是一家金融科技公司，对安全有更高的要求，不使用 Citadel 的证书自签发能力，而是通过对接内部 KMS 系统获取证书，同时提供证书缓存和证书推送更新能力。

证书方案架构图



Sidecar 获取证书的流程，示例如下：



流程说明：

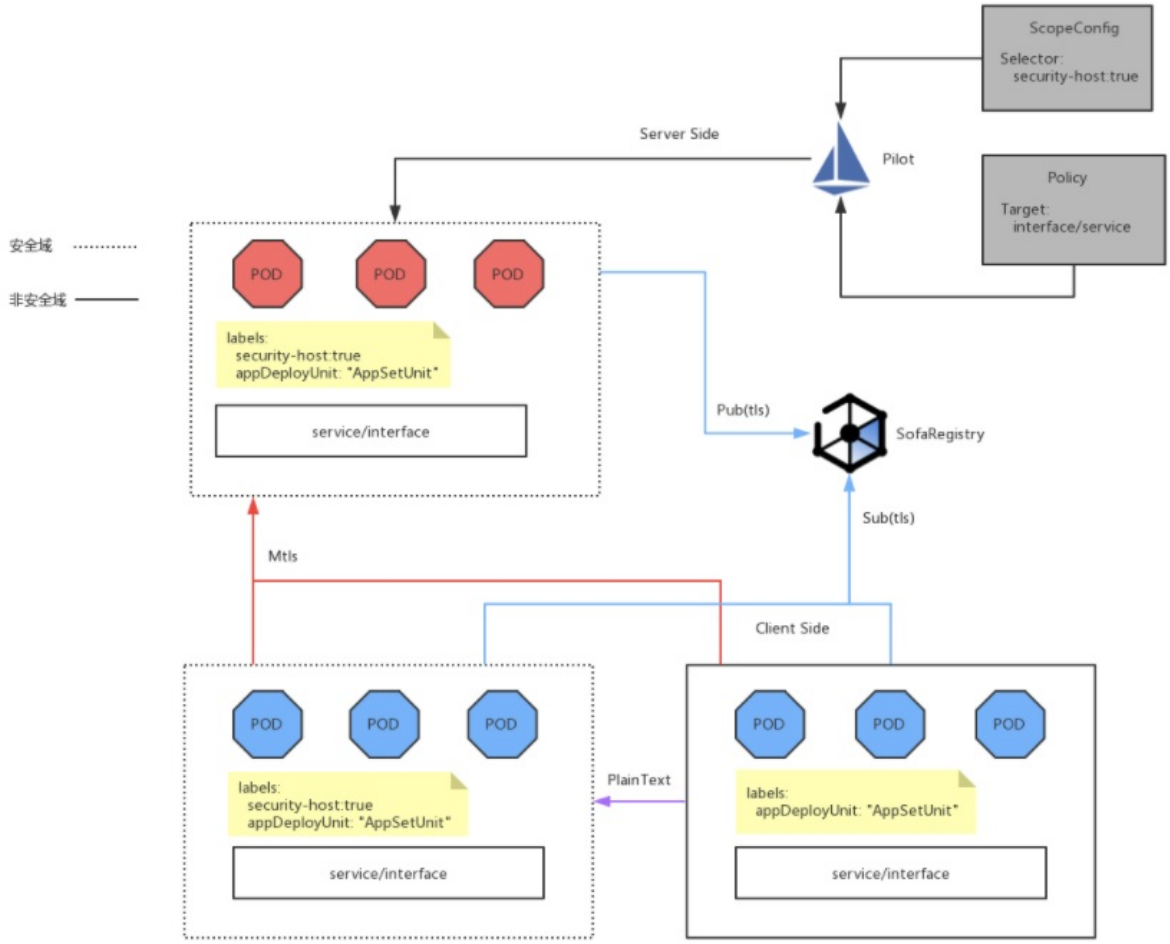
1. Citadel 与 Citadel Agent (nodeagent) 组件通过 MCP 协议(Mesh Configuration Protocol) 同步 Pod 和 CR 信息，避免 Citadel Agent 直接请求 API Server 所导致的 API Server 负载过高问题。
2. Citadel Agent 会进行防篡改校验，并提取 appkey。
3. Citadel Agent 携带 appkey 请求 Citadel 签发证书。
4. Citadel 检查证书是否已缓存，如无证书，则向 KMS 申请签发证书。
5. KMS 会将签发的证书响应返回 Citadel，另外 KMS 也支持证书过期轮换通知。
6. Citadel 收到证书后，会通过步骤 7、8、9 将证书层层传递，最终到达 MOSN。

国密通信

国密通信是基于 TLS 通信实现的，采用更复杂的加密套件来实现安全通信。该功能核心设计是由 Policy 和 Certificate 两部分组成：

- Pilot 负责 Policy 的下发。
- Citadel 负责 Certificate 下发 （基于 SDS 证书方案）。

在落地过程中，仅依靠社区的 PERMISSIVE TLS MODE 还不能满足蚂蚁集团可灰度、可监控、可应急的三板斧要求。所以，在社区方案的基础上，引入了 Pod 粒度的 Sidecar 范围选择能力（也是基于 ScopeConfig），方案示例如下：



流程如下：

1. Pilot List/Watch ScopeConfig CRD 和 Policy CRD，基于 Pod Label 选择 Pod 粒度范围实例。
2. Provider 端 MOSN 收到 Pilot 下发的国密配置后，通过 SDS 方案获取证书，成功获取证书后，会将服务状态推送至 SOFARegistry。
3. SOFARegistry 通知 Consumer 端：MOSN 特定 Provider 端已开启国密通信状态，要重新发起建连请求。

MCP 优化

Citadel Agent 通过 Citadel 同步 POD 及 CRD 等信息时，虽然避免了 Node 粒度部署的 Citadel Agent 对 API Server 的压力，但是，使用 MCP 协议同步数据时，蚂蚁团队遇到了下述挑战：

- 大集群部署时，POD 数量在 10 W 以上时，全量通信时，每次需同步的信息在 100 M 以上，性能开销巨大，网络带宽开销也不可忽视。
- Pod 和 CR 信息变更频繁，高频的全量推送直接制约了可拓展性，同时效率极低。

为了解决以上问题，就需要对 MCP 实现进行改造。改造的目标很明确：减少同步信息量，降低推送频率。为此，蚂蚁团队强化了社区 MCP 的实现，补充了下述功能：

- 为 MCP 协议支持增量信息同步模式，性能大幅优于社区原生方案全量 MCP 同步方式。

- Citadel Agent 是 Node 粒度组件，基于最小信息可见集的想法，Citadel 在同步信息给 Citadel Agent 时，通过 Host IP，Pod 及 CR 上的 Label 筛选出最小集，仅推送每个 Citadel Agent 自身服务范围的信息。
- 基于 Pod 和 CR 的变更事件，可以预先知道需要推送给哪些 Citadel Agent 实例，只对感知变更的 Citadel Agent 触发推送事件，即支持局部推送能力。

未来思考

本次大促，控制面的重心在于解决规模化问题，后续控制面将会在下述领域深入探索：

- 服务发现
- 精细化路由
- Policy As Code

蚂蚁团队将与社区深度合作，控制面将支持通过 MCP 对接多种注册中心，例如 SOFARegistry（已开源），Nacos等，并实现下述功能：

- 服务发现信息同步
- 大规模服务注册发现
- 增量推送大量 endpoint

同时控制面还能通过增强配置下发能力，为应用启动提速，将在 Serverless 极速启动场景获取技术红利。控制面还将结合 Policy As Code，具备极简建站，默认安全等能力，未来极具想象空间。

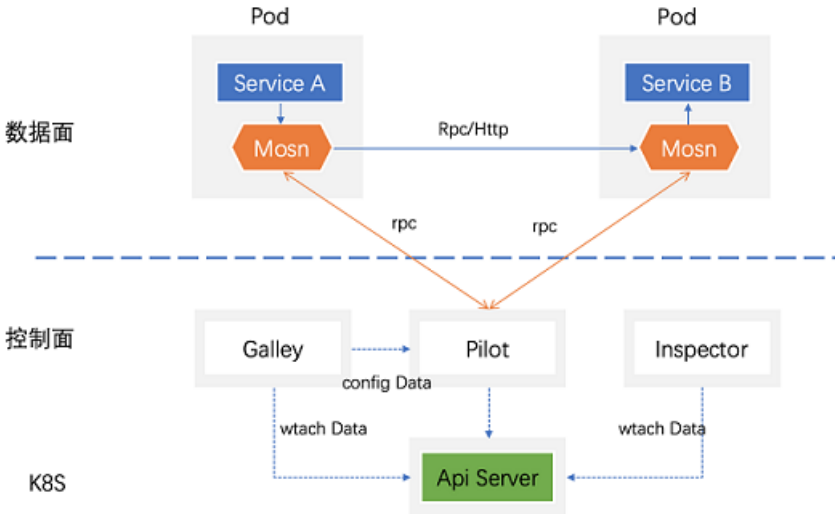
4.2.7.4. 服务网格最佳实践之控制面质量

最近几年，云原生概念越来越火，蚂蚁集团历来热衷于技术创新，积极在云原生领域实践 Service Mesh 理念，结合现有技术架构，将一些通用能力（通信/数据/安全等）抽离出来，沉淀出了 MOSN。同时，依托于 Istio 的能力，扩展出了 Service Mesh 控制面，为 MOSN 提供上层的管控能力。本文主要介绍 Service Mesh 控制面在蚂蚁集团落地过程中，我们如何提供可靠的质量保障。
主要内容分为下面几个方面：

- Service Mesh 组成
- 控制面与经典微服务的差异
- Service Mesh 控制面落地
- 挑战及应对
- 未来规划

Service Mesh 组成

Service Mesh 组成图

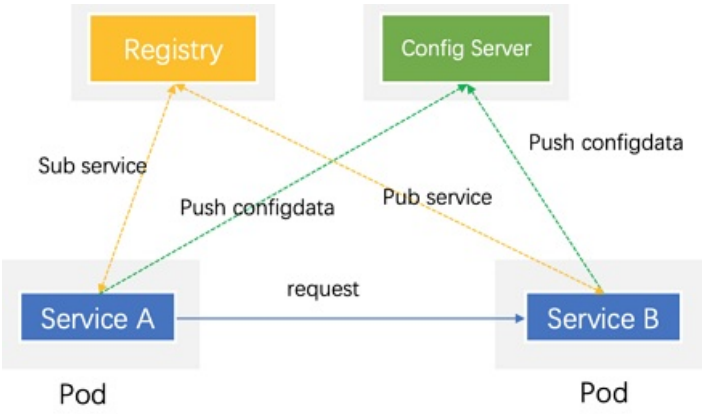


Service Mesh 组成说明：

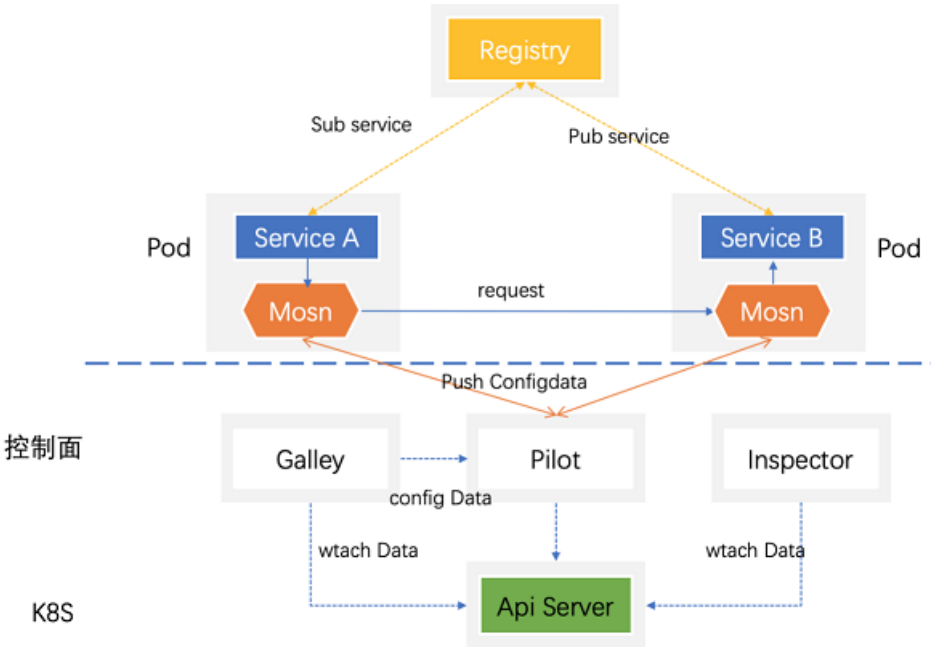
- 数据面：称为 MOSN，是处理应用数据请求的一个独立代理模块，脱离于应用，为应用提供请求代理及一些复杂通信逻辑处理。
- 控制面：称为 SOFAMesh，管理应用配置及业务规则等（例如业务开关、服务路由规则），通过下发配置，“指挥”数据面去执行，满足不同阶段不同实现的业务支持。

控制面与经典微服务的差异

- 经典微服务形态



- Service Mesh 融入形态



> 说明
> 2 者区别主要为下述几个方面：
>
> - Service Mesh 在数据面 Pod 中额外增加了 Mosn。
> - Service Mesh 取消了 Config Server，由 Service Mesh 控制面处理 Mosn 推送的配置数据。而经典微服务通过 Config Server 来推送配置数据。

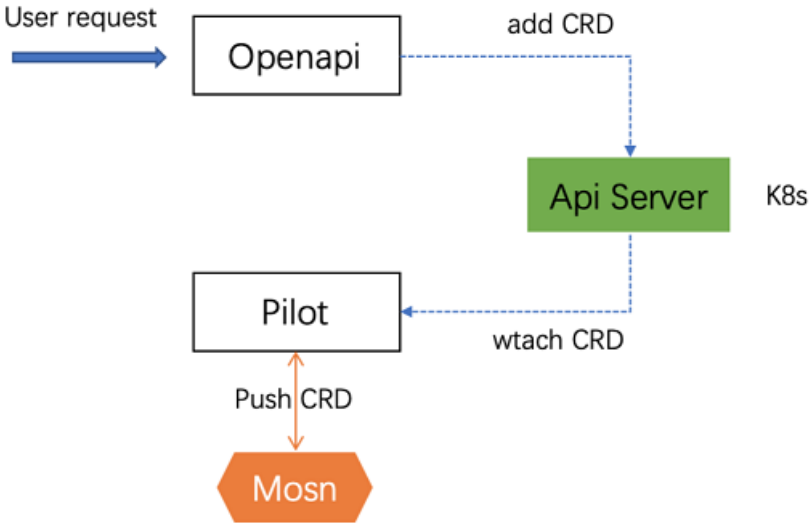
Service Mesh 控制面落地

Service Mesh 控制面，在 Kubenates 基础上对上层提供 CRD/RBAC 等操作能力；并在开源 Istio 的 Enovy 基础上，通过 xDS 协议扩展业务信息，支持将 “服务路由规则/配置开关” 等下发到 MOSN，让 MOSN 去执行，以满足不同业务需求。在双十一活动中，控制面落地了下述 2 个能力：

- CRD 配置下发：CRD 指 Custom Resource Definitions，是 K8S 原生支持的自定义资源语义。
- TLS 加密通信：TLS 指 Transport Layer Security。

CRD 配置下发

蚂蚁集团扩展出了 ScopeConfig 来限定 CR（Custom Resource） 生效的范围，支持机房/集群/应用/IP 等级别设置，在 ScopeConfig 基础上增加了一些 PilocyConfig CRD（策略/规则/开关等），这样可以实现线上灰度可控/降级回滚能力及一些 A/B TEST。

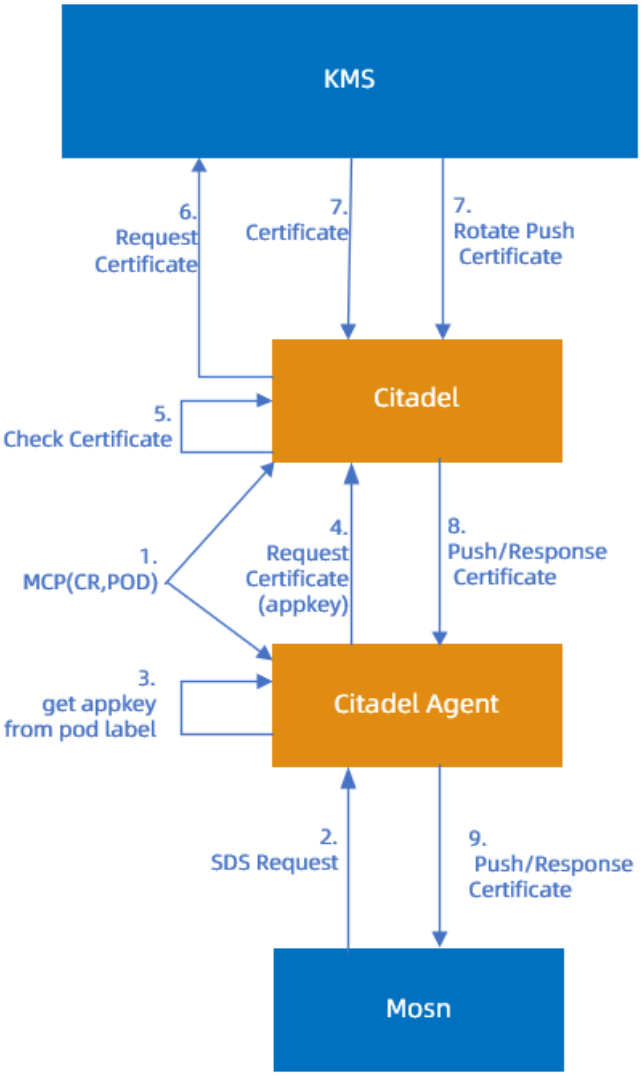


TLS 加密通信

TLS 加密通信实现过程：

1. 在开启 TLS 开关后，MOSN 通过 UDS（Unified Diagnostic Services） 向控制面的 Citadel Agent 获取 TLS 证书信息。
2. TLS 证书由蚂蚁集团自身安全证书服务 KMS 授权给控制面 Citadel 服务，Citadel 会做一些证书检测等处理。
3. 将校验通过的证书同步给 Citadel Agent，依次循环实现证书流转更新。

TLS 加密通信过程示例



挑战及应对

测试架构改进

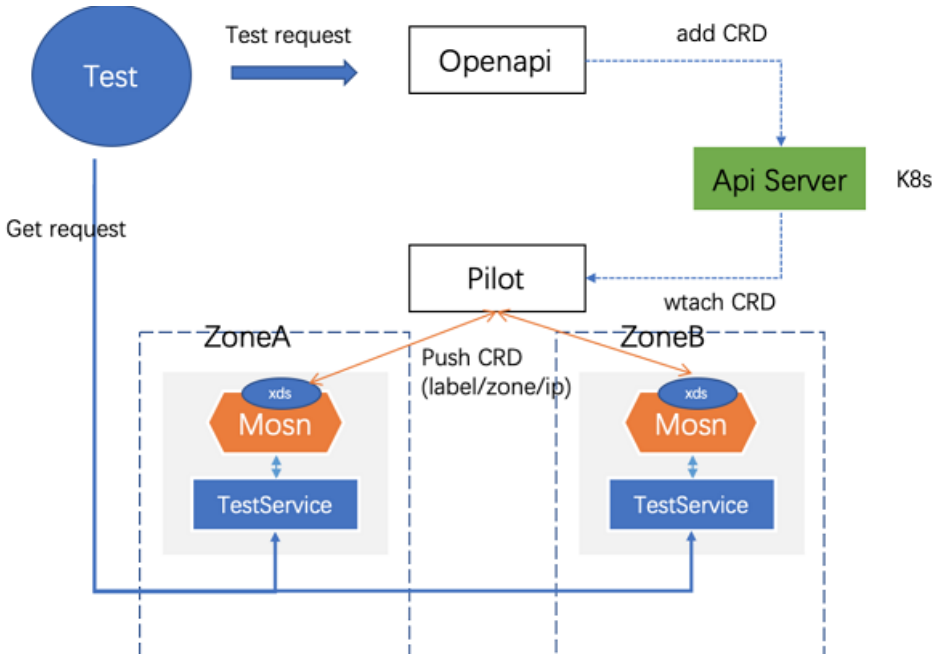
标准模式下， 在一个 Pod 中，Docker 镜像化服务一般只有一个业务应用容器在运行，这种情况下测试框架只要关心业务应用即可。

Service Mesh 是一种非标准模式。MOSN 作为 Sidecar，与业务容器共存于一个 Pod 中，资源与业务应用容器共享。每个业务逻辑都需要通过 MOSN 去处理，因而不能只关心业务，需要将测试验证扩展到 MOSN。

MOSN 集成了控制面 xds Client，与控制面 Pilot 建立通信，用于接收 Pilot 下发的配置信息。蚂蚁集团技术架构有 “三地五中心/同城双活” 等容灾能力，因而产生了 LDC、一个集群多个 Zone 等情况。

控制面 Pilot 下发，配置粒度可以为集群 + Zone + 应用 + IP，要验证这种多 Zone 下发规则的准确性，需要创建多个 xds Client（或者 MOSN）。

Sidecar 不能直接访问，需要通过测试应用暴露出接口，给上层测试。



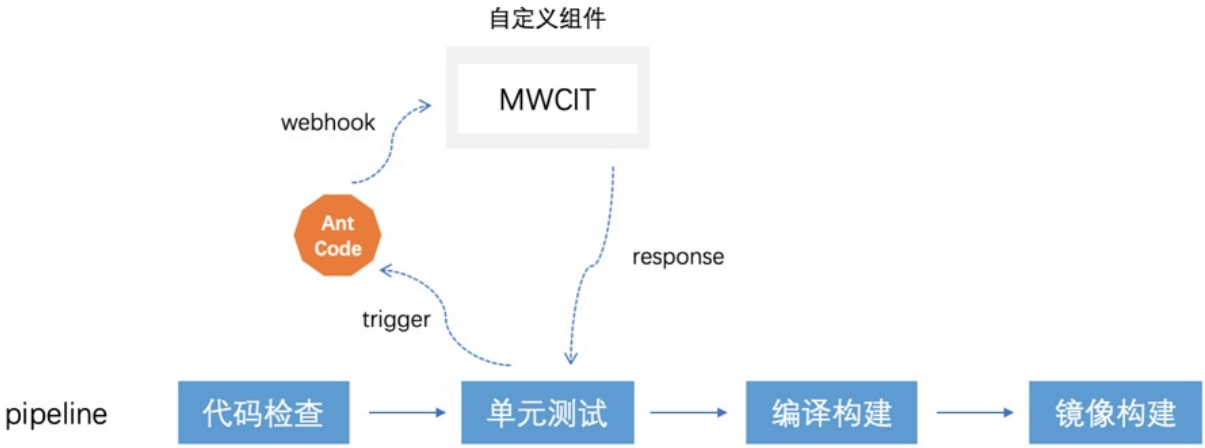
代码质量管控

新工程新代码，在快速迭代同时，还需要保证代码质量。控制面几乎全部由 Go 语言开发，不像标准开发语言 Java 一样有非常丰富的组件支持，因而在度量时没有丰富的质量指标。

借助蚂蚁集团效能团队的 Ant Code 服务，蚂蚁团队配置了适合 ServiceMesh 控制面的流水线，目前可以支持安全扫描/规范扫描/代码覆盖率等，也支持 Code Review，只有研发/质量 Review 通过，才能合并代码，而且在合并时会自动触发执行 UT（Unit Test），避免异常代码合入。

镜像统一化打包，Review 通过和 UT 通过之后，会自动构建测试镜像包，提升测试包质量，示例如下。

在 Ant Code 基础上，蚂蚁团队开发了自定义组件，动态管控代码覆盖率，不断提升代码研发质量，加强研发自测意识。



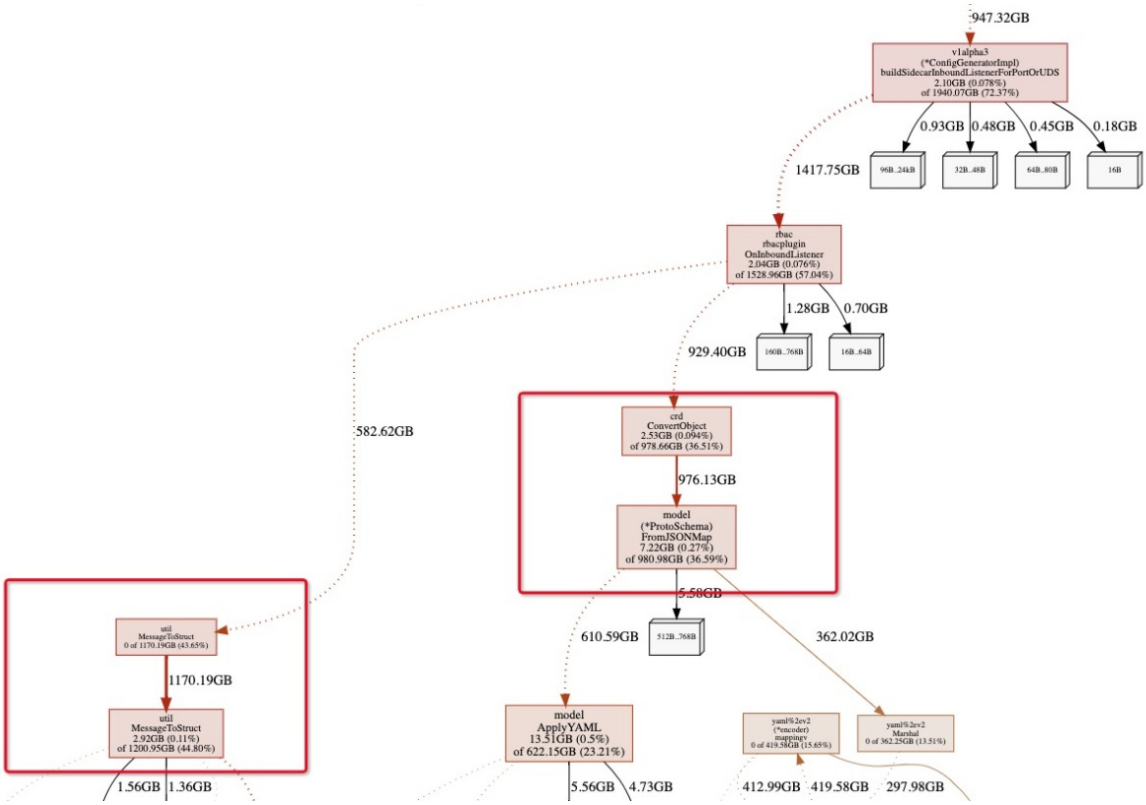
稳定性要求

CRD 下发能力是控制面核心，TLS 加密通信也是基于 CRD 下发开关触发，而下发的关键性能点在于以下几个因素：

- Pilot 支持的 Client 并发数。
- 下发到 Client 的耗时：因为对配置下发实时性要求比较高。

在压测过程中，由于没有足够资源用来创建很多 XDS Client，因而开发了 Mock Client（简化版 XDS Client），只保留核心通信模块，单 Pod 可以支持万级的 Client 模拟。在持续压测一段时间后，蚂蚁团队发现内存频繁在 GC，导致耗时很高。

PProf 内存分析图



PProf内存分析图说明：

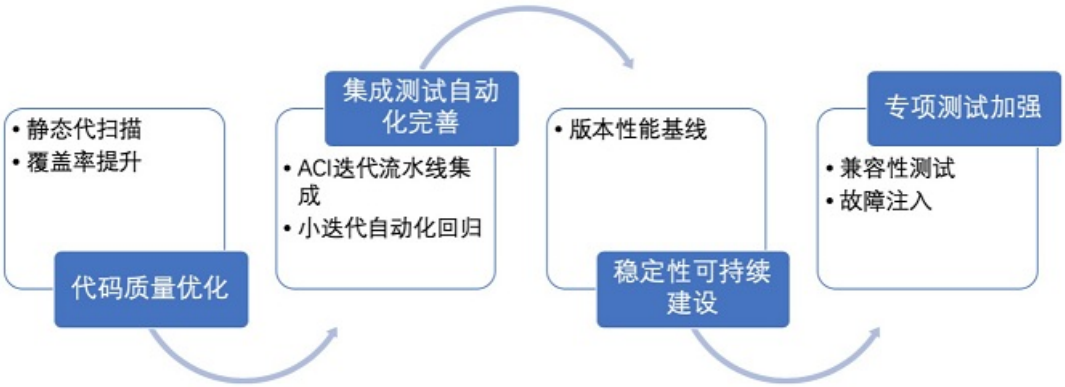
- MessageToStruct 和 FromJsonMap 最占内存，都是在对数据进行转换。
- MessageToStruct 之前有过同类优化，因此可以很快解决。
- FromJsonMap：是 CRD 数据转换的核心，需要将数据转成 K8S 能识别的 YAML 信息。蚂蚁团队对此进行改造，将一部分内存进行重用，并优化转换函数，耗时下降好几倍，降到毫秒级别。

运维管控

控制面服务基于 K8S 开发，以 Deployment 和 Daemonset 形式发布上线，每次发布都会影响到整个集群，需要一定风险把控手段。

依靠 Ant code 的代码 Review 机制，蚂蚁团队建立了研发、质量、SRE 三方评审机制，管控变更范围。评审完成后自动部署到对应集群，通过监控秒级巡检，发现问题可以及时告警，共同保障了线上生产质量。

未来规划



4.2.7.5. 服务网格最佳实践之数据面质量

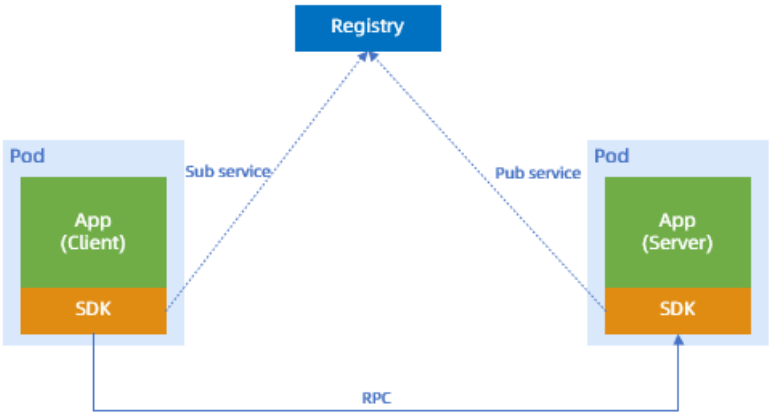
服务网格（Service Mesh） 包含了控制面和数据面，其中，数据面为自研的 MOSN。本文介绍 Service Mesh 数据面 MOSN 在蚂蚁集团的落地过程中，如何进行质量保障，以及如何确保从现有的微服务体系平滑演进至 Service Mesh 架构。

MOSN 简介

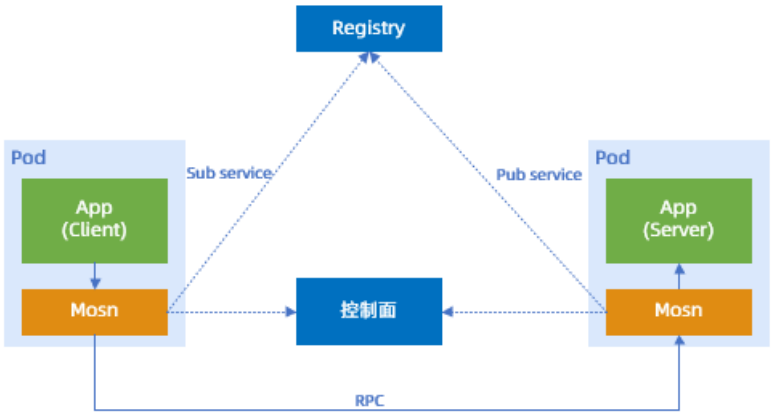
经典微服务架构和 Service Mesh 架构的差别主要包括：

- 经典微服务体系：基于 SDK 实现服务的注册和发现。
- Service Mesh 体系：基于数据面 MOSN 实现服务的注册和发现，且服务调用也通过 MOSN 来完成。

经典微服务架构



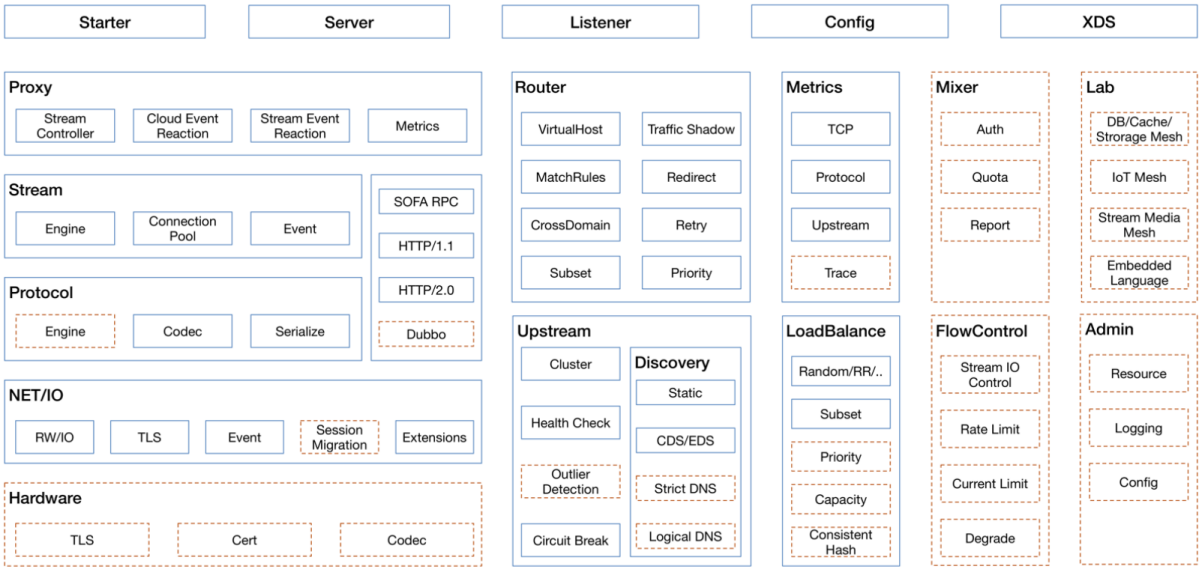
Service Mesh 微服务架构



在 ServiceMesh 体系下，原先的 SDK 逻辑下沉至数据面 MOSN 中，会带来如下优点：

- 基础能力下沉后，基础能力的升级不再依赖应用的改造和发布，降低应用打扰率。
- 基础能力升级后，可快速迭代并铺开，对应用无感。

蚂蚁集团自研数据面 MOSN 具备的能力：

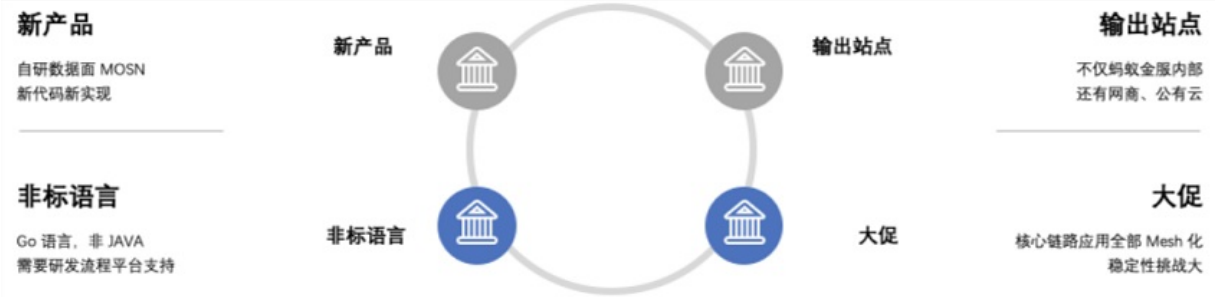


落地面临的问题

MOSN 具备丰富的能力，在落地过程中，也面临下述几个问题：

- 在质量保障上有哪些挑战？
- 如何应对质保挑战？

在质量保障上有哪些挑战？



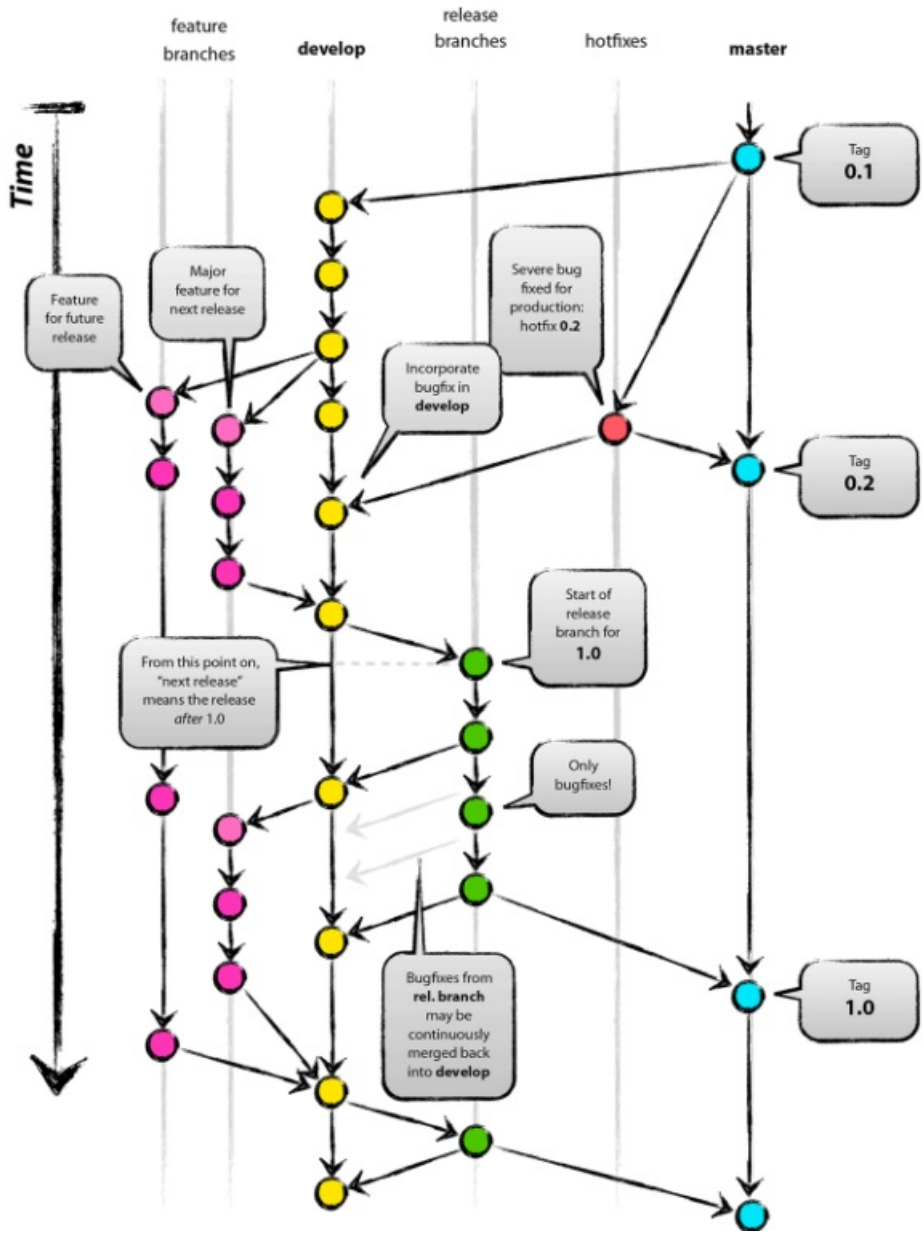
- **新产品**：蚂蚁团队并没有采用社区方案 Envoy 或 Linkerd，而是选择自研数据面 MOSN。从底层的网络模型，到上层的业务模块，全部需要研发重新编码来实现，并作为基础设施，去替代线上运行稳定的 SDK。这种线上变更的风险很高。
- **非标语言**：这套自研的数据面 MOSN，采用的是 Golang 语言。蚂蚁集团内部的技术栈主要是 Java，相应的研发流程也是围绕着 Java 来构建。这套新引入的技术栈，需要新的流程平台来支撑。
- **大促**：从蚂蚁集团内部着手基础设施升级开始到双十一来临，时间短，任务重。完成核心链路应用的全部 Mesh 化，涉及应用数量达 100 多个，涉及 POD 达数十万多个，还需要经受住双十一大促考验。
- **线上稳定性**：在升级过程中，要求对业务无损，大规模升级完成后，业务能正常运行。
- **输出站点**：包括蚂蚁集团、网商银行和公有云。但是，3 个站点所依赖的基础设施和所要求的能力，存在差异，这些不同能力要求会造成代码分支碎片化。因此需要考虑下述 2 个问题。
  - 如何同时保障多站点输出的质量？
  - 如何管理好这些碎片化分支？

如何应对质保挑战？

规范研发流程

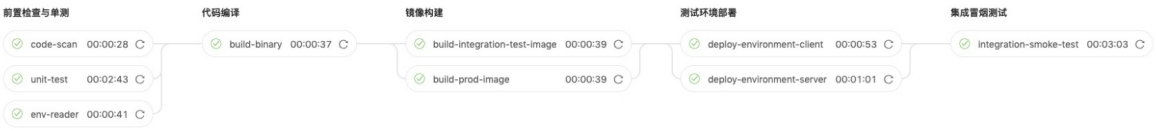
在推进 Mesh 化落地时，蚂蚁集团内部的研发效能部开始推出 ANT CODE 研发平台，其支持 Golang 语言，提供自定义流水线编排能力和部分原生插件。基于该平台，蚂蚁团队对研发过程做了如下规范：

- **git-flow 分支管控**


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

- **代码审查（CR）**：从现有的质量保障手段来看，Code Review（CR） 是低成本发现问题的有效方式，基于 ANT CODE 平台，蚂蚁团队定义了下述 CR 规范。

- **交付流水线**：以 MOSN 持续交付流水线为例，图示如下。



- **流水线卡点**

  - 上述任一步骤执行失败，整个流水线将会失败，代码将无法合并。
  - 卡点规则在工程根目录的 yaml 配置文件中。通过自定义插件统一收口这些卡点规则，精简了工程根目录的 yaml 配置文件，也使得规则的变更只需更新插件的内存配置即可。

集成测试

为了验证 MOSN，蚂蚁团队搭建了一套完整的测试环境。这里以 MOSN 中的 RPC 功能为例，阐述这套环境的构成要素及环境部署架构。

集成测试环境构成要素

集成测试环境优缺点：

- 优点：
  - MOSN 中的路由能力，完全兼容原先 SDK 中的能力，且在其基础上不断优化，如通过路由缓存提升性能等。
  - 依托于这套环境做集成测试，可完成自动化脚本编写，并在迭代中持续集成。
- 缺点：这套测试环境，并不能发现所有的问题，有一些问题会遗留到线上，并给业务带来干扰。

下文以 RPC 路由为例，阐述对线下集成测试的一些思考。

业务在做跨 IDC 路由时，主要通过 ANTVIP 实现，这就需要业务在自己的代码中设置 VIP 地址，格式如下：

```
<sofa:reference interface="com.alipay.APPNAME.facade.SampleService" id="sampleRpcService">
<sofa:binding.tr>
<sofa:vip url="APPNAME-pool.zone.alipay.net:12200"/>
</sofa:binding.tr>
</sofa:reference>
```

但运行中，有部分应用，因为历史原因，配置了不合法的 URL，如：

```
<sofa:reference interface="com.alipay.APPNAME.facade.SampleService" id="sampleRpcService">
<sofa:binding.tr>
<sofa:vip url="http://APPNAME-pool.zone.alipay.net:12200?_TIMEOUT=3000"/>
</sofa:binding.tr>
</sofa:reference>
```

> **说明**
> 上述 VIP URL 指定了 12200 端口，却又同时指定了 http，这种配置是不合法的。因为历史原因，这种场景在原先的 CE（Cloud Engine）中做了兼容，而在 MOSN 中未继续这种兼容。

这类历史遗留问题，在多数产品里都会遇到，可以归为测试场景分析遗漏。一般对于这种场景，可以借助于线上流量回放的能力，将线上的真实流量复制到线下，作为测试场景的补充。但现有的流量回放能力，并不能直接用于 MOSN，原因在于 RPC 的路由寻址与部署结构有关，线上的流量并不能够在线下直接运行，因此需要一套新的流量回放解决方案。目前，这部分能力还在建设中。

专项测试

除了上述功能测试之外，蚂蚁团队还引入了如下专项测试：

- 兼容性测试
- 性能测试
- 故障注入测试

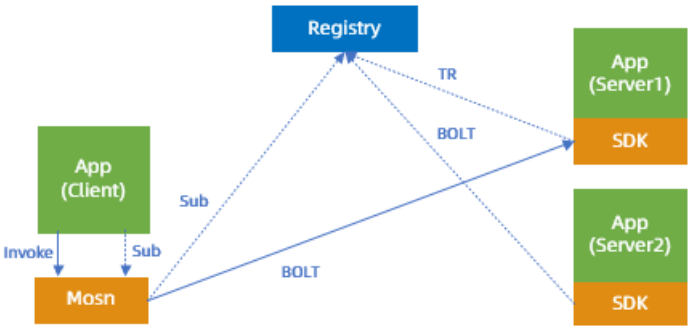
兼容性测试

MOSN 兼容性验证图

发现的问题：通过兼容性测试，发现问题主要集中在 **接入/未接入MOSN** 这个场景中。
例如，在线下验证过程中，接入了 MOSN 的客户端调用未接入 MOSN 的服务端会偶发失败，服务端会有如下协议解析报错：

```
[SocketAcceptorIoProcessor-1.1]Error com.taobao.remoting -Decode meetexception
java.io.IOException:Invalid protocol header:1
```

经分析，原因在于老版本 RPC 支持 TR 协议，后续的新版支持 BOLT 协议，应用升级过程中，存在同时提供 TR 协议和 BOLT 协议服务的情况，即相同接口提供不同协议的服务，示例如下：



配图说明：

- 应用向 MOSN 发布服务订阅的请求，MOSN 向配置中心订阅，配置中心返回给 MOSN 两个地址，分别支持 TR 和 BOLT，MOSN 从两个地址中选出一个返回给应用 APP。
- MOSN 返回给 APP 的地址是直接取配置中心返回的第一条数据，有下述 2 种可能：
  - 地址是支持 BOLT 协议的服务端：后续应用发起服调用时，会直接以 BOLT 协议请求 MOSN，MOSN 选址时，会轮询两个服务提供方，如果调用到 Server1，就出现了上述协议不支持的报错。
  - 地址是支持 TR 协议的服务端：因 BOLT 对 TR 做了兼容，因而不会出现上述问题。

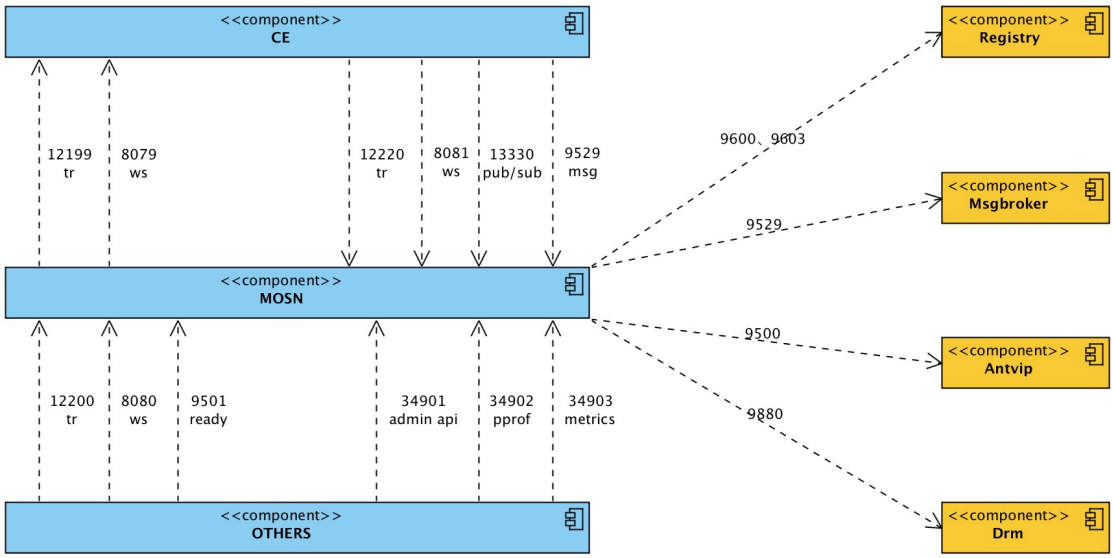
修复方案：最终的修复方案是，MOSN 收到配置中心的地址列表后，会做一层过滤，只要服务端列表中包含 TR 协议的，则全部降级到 TR 协议。

性能测试

为了验证业务接入 MOSN 后的性能影响，蚂蚁团队将业务的性能压测环境应用接入 MOSN，并使用业务的压测流量做验证。下述为业务接入 MOSN 后的性能对比图：

故障注入测试

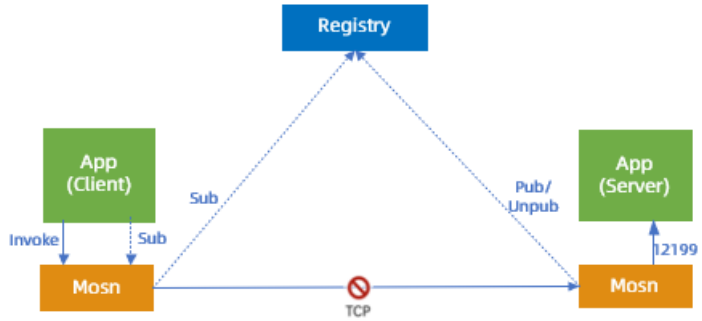
从 MOSN 的视角来看，其外部依赖如下：



除了验证 MOSN 自身的功能外，蚂蚁团队还通过故障注入的方式，对 MOSN 的外部依赖做了专项测试。通过这种方式发现了一些上述功能测试未覆盖的场景。

下文以应用和 MOSN 之间的 12199 端口为例，进行说明。

MOSN 与 APP 心跳断连处理示意图



配图说明：

- 应用 APP 接入 MOSN 后，原先应用对外提供的 12200 端口改由 MOSN 去监听，应用的端口修改为 12199，MOSN 会向应用的 12199 端口发送心跳，检测应用是否存活。
- 如果应用运行过程中出现问题，MOSN 可以通过心跳的方式及时感知到。
- 如果 MOSN 感知到心跳异常后，会向配置中心取消服务注册，同时关闭对外提供的 12200 端口服务。这样做的目的是防止服务端出现问题后，仍收到客户端的服务调用，导致请求失败。

为了验证该场景，蚂蚁团队在线下测试环境中，通过 iptables 命令 drop 掉 APP 返回给 MOSN 的响应数据，人为制造应用 APP 异常的场景。通过这种方式，蚂蚁团队也发现了一些其它不符合预期的 bug，并修复完成。

快速迭代

从项目立项到双十一，留给整个项目组的时间并不充裕，为了确保双十一能够平稳过度，蚂蚁团队采取的策略是：在质量可控范围内，通过快速迭代，让 MOSN 在线上能够获得充分的时间做验证。这离不开上述基础测试能力的建设，同时也依赖蚂蚁集团线上变更三板斧原则。

线上压测及演练

MOSN 全部上线之后，跟随着业务一起，由业务的 SRE（Site Reliability Engineer） 触发多轮的线上压测，以此发现更多的问题。线上演练，主要是针对 MOSN 自身的应急预案的演练，如线上 MOSN 日志降级等。

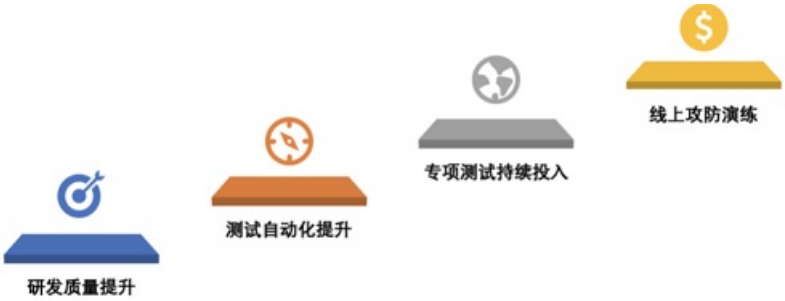
监控及巡检

在 Mesh 化之前，线上的监控主要是整个 POD 级别的监控；Mesh 化之后，在监控团队的配合下，线上监控支持了 MOSN 的独立监控，以及 POD 中 Sidecar 与 APP 容器的监控。

巡检是对监控的补充，部分信息无法通过监控直接获取，如 MOSN 版本的一致性。通过巡检可以发现，哪些 POD 的 MOSN 版本还没有升级到最新，是否存在有问题的版本还在线上运行的情况等。

未来如何完善产品？

通过双十一，Service Mesh 更多是在性能和线上稳定性方面给出了证明，但技术风险底盘依然不够稳固。接下来，除了建设新功能外，蚂蚁团队还会在技术风险领域做更多的建设。在质量这块，主要包括：



在这个过程中，蚂蚁团队期望能够引入一些新的测试技术，能够有一些新的质量创新，以便把 Service Mesh 做的越来越好。

参考资料

- MOSN Github 仓库

- 从网络接入层到 Service Mesh，蚂蚁集团网络代理的演进之路
- 诗和远方：蚂蚁集团 Service Mesh 深度实践 | QCon 实录

4.2.7.6. 服务网格最佳实践之网关

本文结合无线网关的发展历程，解读进行 Service Mesh 改造的缘由和价值，同时介绍在双十一落地过程中如何保障业务流量平滑迁移至新架构下的 Mesh 网关。
具体内容将从下述几个方面展开：

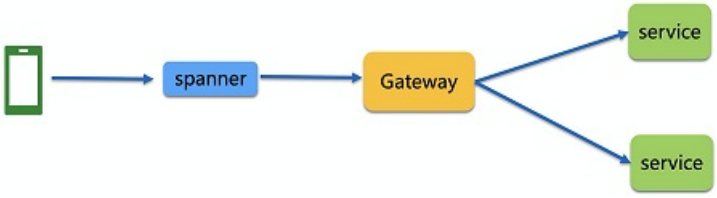
- 网关的演变历史：解释网关为什么要 Mesh 化。
- 网关 Mesh 化：阐述如何进行 Mesh 化改造。
- 双十一落地：介绍在此过程中实现三板斧能力。

网关的演变历史

当前，蚂蚁集团的无线网关接入了数百个业务系统，提供数万个 API 服务，是蚂蚁集团客户端绝对的流量入口，支持的业务横跨支付宝、网商、财富、微贷、芝麻和国际 A+ 等多种场景。面对多种业务形态、复杂网络结构，无线网关的架构也在不断演进。

中心化网关

在 All In 无线的年代，随着通用能力的沉淀，形成了中心化网关 Gateway，示例如下：



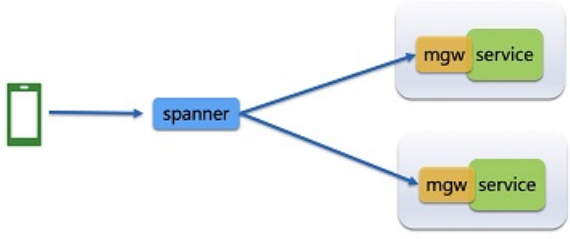
流程说明：

1. 客户端连接到网关接入层集群 Spanner。
2. Spanner 会把客户端请求转发到无线网关集群 Gateway。
3. Gateway 提供通用能力如鉴权、限流等处理请求，并根据服务标识将请求路由到正确的后端服务；服务处理完请求，响应原路返回。

2016 年新春红包爆发，蚂蚁森林等新兴业务发展壮大，网关集群机器数不断增长。这加剧了运维层面的复杂性，IT成本也面临不能承受之重。同时，一些核心链路的业务如无线收银台、扫一扫等，迫切需要与其他业务隔离，避免不可预知的突发流量影响到这些高保业务的可用性。因此，2016 年下半年开始建设和推广去中心化网关。

去中心化网关

去中心化网关示例



去中心化网关将原先集中式网关的能力进行了拆分：

- 转发逻辑：将 Gateway 中根据服务标识转发的能力迁移到 Spanner 上。
- 网关逻辑：将网关通用能力如鉴权、限流、LDC 等功能，抽成一个 mgw JAR，集成到业务系统中，与后端服务 JVM 进程一起运行。

此时，客户端请求的处理流程如下：

- 客户端请求到 Spanner 后，Spanner 会根据服务标识转发请求到后端服务的 mgw 中。
- mgw 进行通用网关能力处理，90% 的请求随即进行 JVM 内部调用。

去中心化网关与集中式网关相比，具有如下优点：

- **提升性能**：
  - 少一层网关链路，网关 JAR 调用业务是 JVM 内部调用。
  - 大促期间，无需关心网关的容量，有多少业务就有多少网关。
- **提高稳定性**：
  - 集中式网关形态下，网关出现问题，所有业务都会受到影响。
  - 去中心化后，网关的问题，不会影响去中心化的应用。

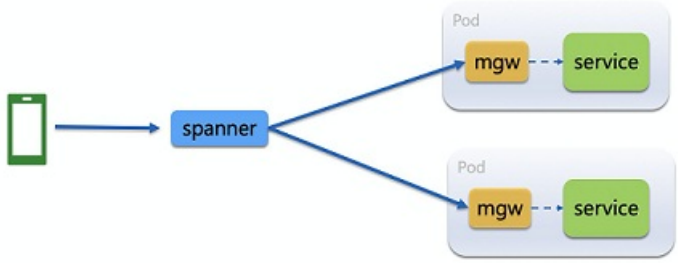
但凡事具有两面性，随着在 TOP 30 的网关应用中落地铺开，去中心化网关的缺点也逐步显现：

- **研发效能低**：
  - 接入难：需要引入 15 + 的 pom 依赖、20 + 的配置，深度侵入业务配置。
  - 版本收敛难：当前 mgw.jar 已迭代了 40+ 版本，但是，还有业务使用的是初版，难以收敛。
  - 新功能推广难：新能力上线要推动业务升级和发布，往往需要一个月甚至更久时间。
- **干扰业务稳定性**：
  - 依赖冲突，干扰业务稳定性，这种情况发生了不止一次。
  - 网关功能无法灰度、独立监测，资源占用无法评估和隔离。
- **不支持异构接入**：非 Java 应用，无法使用去中心化网关。

Mesh 网关

去中心化网关存在的诸多问题，多数是由于网关功能与业务进程捆绑在一起造成的。这引发了蚂蚁团队的思考：如果网关功能从业务进程中抽离出来，这些问题是否就可以迎刃而解了？这种想法，与 Service Mesh 的 Sidecar 模式不谋而合。因此在去中心化网关发展了三年后，我们再出发对蚂蚁集团无线网关进行了 Mesh 化改造，以期解决相关痛点。

Mesh 网关示例



Mesh 网关与后端服务同一个 Pod 部署，即 Mesh 网关与业务系统同服务器、不同进程，具备网关的全量能力。客户端请求的处理流程如下：

- 客户端请求到 Spanner 后，Spanner 会根据服务标识转发请求到后端服务同一 Pod 中的 Mesh 网关。
- Mesh 网关执行通用逻辑后，调用同机不同进程的业务服务，完成业务处理。

对比去中心化网关的问题，我们来分析下 Mesh 网关所带来的优势：

- 研发效能：
  - 接入方便：业务无需引入繁杂的依赖和配置，即可接入 Mesh 网关。

- 版本容易收敛、新功能推广快：新版本在灰度验证通过后，即可全网推广升级，无需维护和排查多版本带来的各种问题。

- 业务稳定性：
  - 无损升级：业务系统无需感知网关的升级变更，且网关的迭代升级可以利用 MOSN 现有的特性进行细粒度的灰度和验证，做到无损升级。
  - 独立监测：由于和业务系统在不同进程，可以实时遥测 Mesh 网关进程的表现，并据此评估和优化，增强后端服务稳定性。
- 异构系统接入：Mesh 网关相当于一个代理，对前端屏蔽后端的差异，支持 SOFARPC、Dubbo 和 gRPC 等主流 RPC 协议，并支持非 SOFA 体系的异构系统接入。

至此，无线网关的演变历史已经说明完毕。
细心的读者可能有下述疑问：

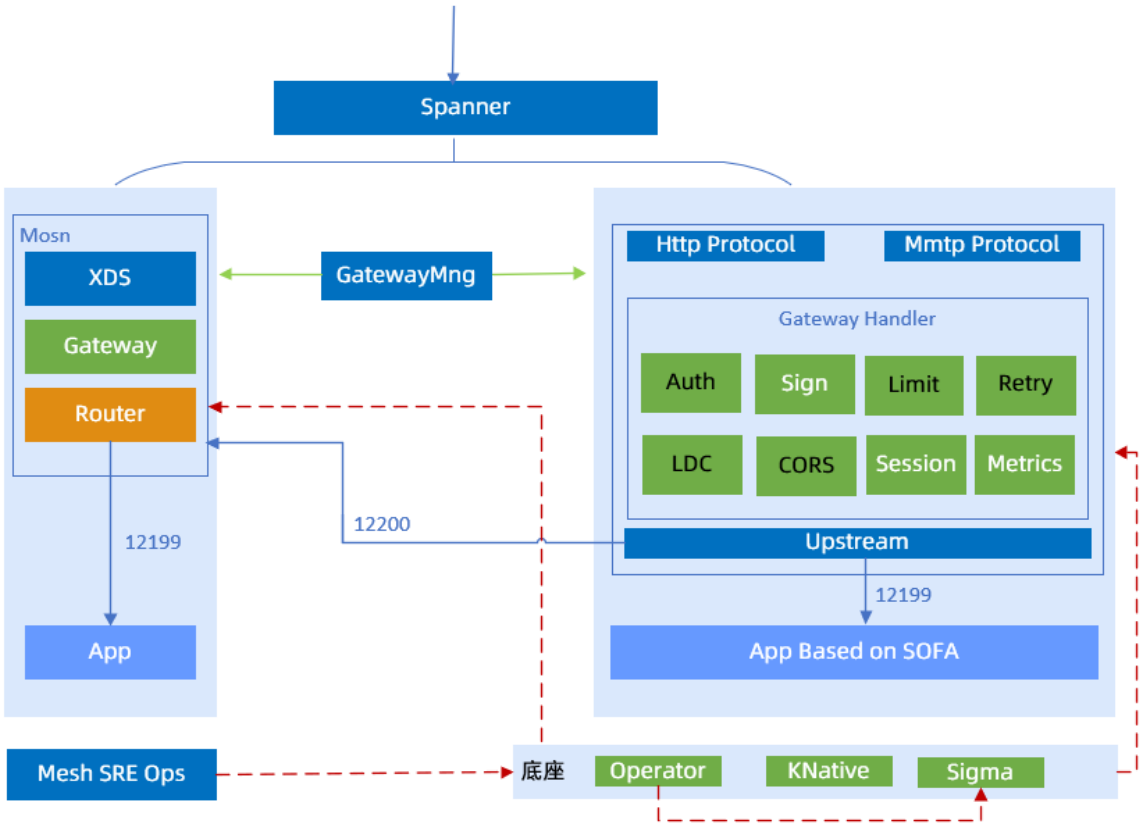
- Mesh 化之后的请求处理流程不是同进程调用，比去中心化网关多了一跳，是否有性能损耗？
- 如此大的改革之后，在上线过程中如何保障稳定性？

下文将尝试回答上述疑问。

网关 Mesh 化

架构

依照 Service Mesh 的分层模型，Mesh 网关分为数据面和控制面，示例如下：



网关Mesh化分层模型图说明：

- 蓝色箭头线是客户端请求的处理流程，Mesh 网关数据面在蚂蚁集团内部 MOSN 的 Sidecar 中。
- 绿色箭头线是网关配置下发过程，Mesh 网关控制面当前还是由网关管控平台来承载。
- 红色箭头线是 MOSN Sidecar 的运维体系。

数据面

数据面，采用 Go 语言实现了原先网关的全量能力，融合进 MOSN 的模型中，复用了其他组件已有的能力。同时网络接入层 Spanner 也实现了请求分发决策能力。数据面包含下述几个功能：

- **数据转换**：将客户端的请求数据转换成后端请求数据，将后端响应数据转换成客户端响应。利用 MOSN 协议层扩展能力，实现了对网关自有协议 Mmtp 的解析能力。
- **通用功能**：具备授权、安全、限流、LDC 和重试等网关通用能力。利用 MOSN Stream Filter 注册机制以及统一的 Stream Send/Receive Filter 接口扩展而来。
- **请求路由**：客户端请求按照特定规则路由到正确的后端系统。在网关层面实现 LDC 逻辑后，复用 MOSN RPC 组件的路由匹配能力。其中大部分请求都会路由到当前 Zone，从而直接请求到当前 Pod 的业务进程端口。
- **后端调用**：支持调用多种类型的后端服务，支持 SOFARPC、Chair 等后端，后期计划支持更多的 RPC 框架和异构系统。

控制面

为网关用户提供新增、配置 API 等服务的管控系统，可将网关配置数据下发至 Mesh 网关的 Sidecar 实例。

多一跳对业务性能是否有影响？

MOSN 层性能损耗分析图

更多性能分析详情，请参考 蚂蚁金服 Service Mesh 深度实践。

分析结论是：相较于复杂的业务逻辑，Mesh 的性能损耗在可接受的范围内，同时带来了快速获得收益的能力。Mesh 网关在此次接入过程中也做了 Session 精简化：

- 内容精简：从 7k 字节降低到 650 字节。
- 无解压缩：节省 GZip 的性能损耗。
- 无线 PC 隔离：解决 Session 污染问题。

在带 Session 校验场景下，相较于去中心化网关，基准性能压测得出的结论是：QPS 提升近 1 倍，RT 下降约 15%。

双十一落地

2019 年双十一，Mesh 网关的落地情况如下：

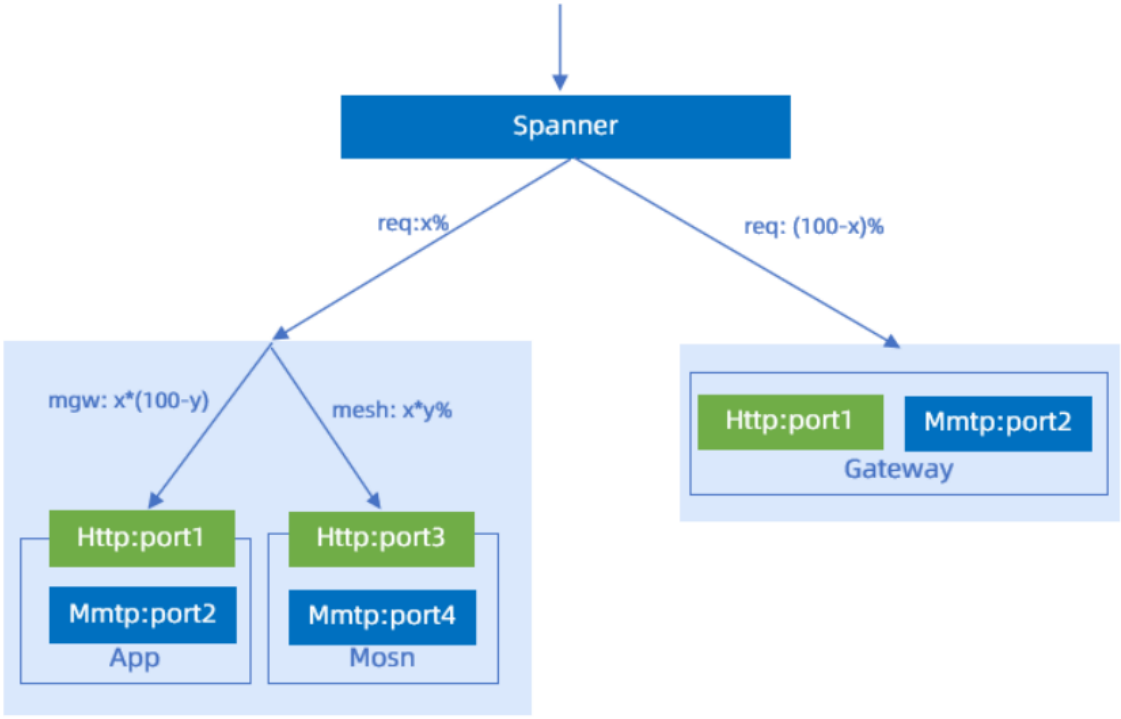
- 大促支付链路淘系支付 API 100% 引流。
- 大促会员链路主战场应用 100% 引流。
- 网关 Top API 5% 引流。

从上述引流情况可以看出，Mesh 网关支持多维度百分比引流。当然，新的架构体系在大促时落地，充满了各种风险，图示如下：

为了做好风险管控，我们严格按照三板斧（可监控、可灰度、可回滚）的要求建设相关能力。当前 Mesh 网关的路由具备 API 百分比、服务器、Zone 以及应用级别的开关能力，支持业务灰度和应急止血。

开关	生效时机	作用
Mesh 百分比	立即	控制某个 API 的 Mesh 路由是否开启，支持百分比。
Label	自动感知	控制某台服务器的 Mesh 路由是否开启。
Zone	Spanner Reload	控制某个 Zone 的 Mesh 路由是否开启。
MeshOnly	Spanner Reload	控制某个应用的 Mesh 路由是否开启。

这里着重分享下 Mesh 网关 API 百分比分流策略。通过和网络接入层 Spanner 配合，蚂蚁团队开发了 Mesh 分流功能，实现了秒级生效的单个 API 百分比切流 Mesh 网关能力。某 API 配置了 `去中心化 x%，Mesh 化 y%`，其切流规则生效时的流量，示意如下：



- 去中心化网关服务：由 Port 1（Http）或 Port 2（Mmtp）端口提供服务。
- Mesh 网关服务：由 Port 3（Http）或 Port 4（Mmtp）端口提供服务。

其中百分比信息由业务方在 API 管控页面配置，随着 API 响应头带回 Spanner Worker，由 Worker 自主学习后，按照对应的百分比做分流决策。同时实现了路由失败回退机制，优先级 `Service:去中心化端口 > Service:Mesh 端口 > Gateway` ，由集中式网关进行兜底保证业务不失败。

4.2.7.7. 服务网格最佳实践之消息

作为蚂蚁集团向下一代云原生架构演进的核心基础设施，Service Mesh 在 2019 年得到了大规模的应用与落地。截止目前，蚂蚁集团的 Service Mesh 数据平面 MOSN 已接入应用数百个，接入容器数量达数十万，一举成为全世界最大的 Service Mesh 集群。同时，在双十一大促中，Service Mesh 的表现也十分亮眼，RPC 峰值 QPS 达到了几千万，消息峰值 TPS 达到了几百万，且引入 Service Mesh 后的平均 RT 增长幅度控制在 0.2 ms 以内。
本文将从以下几个方面对消息 Mesh 进行解读：

- 消息 Mesh 介绍：解答消息 Mesh 在整个 Service Mesh 中的地位是什么，它又能为业务带来哪些价值。
- 消息 Mesh 的价值：介绍消息 Mesh 所能带来的价值和收益。
- 消息 Mesh 化改造：结合蚂蚁集团消息中间件团队关于 Mesh 化的实践与思考，阐述如何在消息领域进行 Mesh 化改造。
- 消息 Mesh 的挑战：消息 Mesh 在蚂蚁集团内部大规模落地过程中遇到的问题与挑战，以及对应的解决方案。
- 消息 Mesh 流量调度：消息流量精细化调度上的思考和在 Mesh 上的实现与落地。

消息 Mesh 简介

Service Mesh 作为云原生场景下微服务架构的基础设施（轻量级的网络代理），正受到越来越多的关注。Service Mesh 不仅负责在微服务架构的复杂拓扑中可靠地传递请求，也将限流、熔断、监控、链路追踪、服务发现、负载均衡、异常处理等与业务逻辑无关的流量控制或服务治理行为下沉，让应用程序能更好地关注自身业务逻辑。

微服务架构中的通信模式实际上是多种多样的，包含：

- 同步的请求调用。
- 异步的消息或事件驱动。

流行的 Service Mesh 实现（Istio、Linkerd、Consul Connect 等），仍局限在对微服务中同步请求调用的关注，却无法管理和追踪异步消息流量。
消息 Mesh 是对这一块的重要补充，通过将消息 Mesh 有机地融合到 Service Mesh 中，可以帮助 Service Mesh 实现对所有微服务流量的管控和追踪，从而进一步完善其架构目标。

消息 Mesh 的价值

在传统的消息中间件领域，我们更关注的核心指标为：

- 服务端的性能
- 服务可用性
- 数据可靠性等

有些能力与业务应用密切相关却表现不佳，主要包括：

- 消息的流量控制：限流、熔断、灰度、着色、分组等。
- 消息的服务治理：消息量级与消息应用拓扑等。
- 消息链路的追踪：消息轨迹

造成这个局面的原因包括：

- 传统模式下上述能力的提供和优化都涉及客户端的改造与大规模升级，整个过程常常比较漫长，难以快速根据有效反馈不断优化。
- 缺乏一个统一的架构指导思想，混乱无序地向客户端叠加相关功能只会让消息客户端变得越来越臃肿和难以维护，也变向增加了业务系统的接入、调试和排查问题的成本。

消息 Mesh 作为 Service Mesh 的补充，能显著带来如下价值和收益：

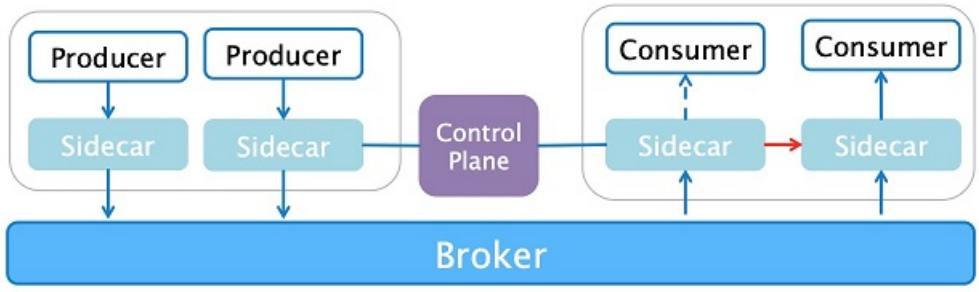
- **快速升级**：通过将与业务逻辑无关的一些核心能力下沉到 Sidecar 中，使这些能力的单独快速迭代与升级成为可能。
- **流量控制**：可以向 Sidecar 中集成各种流量控制策略，例如可根据消息类型、消息数量、消息大小等多种参数来控制消息的发送和消费速率。
- **流量调度**：通过打通 Sidecar 节点之间的通信链路，可以利用 Sidecar 的流量转发来实现任意精度的消息流量调度，帮助基于事件驱动的微服务应用进行多版本流量管理、流量着色、分组路由、细粒度的流量灰度与A/B策略等。
- **消息验证**：消息验证在基于事件驱动的微服务架构中正变得越来越重要，通过将这部分能力下沉到 Sidecar，不仅可以让业务系统无缝集成消息验证能力，也可以让 Sidecar 通过 Schema 理解消息内容，并进一步具备恶意内容识别等安全管控能力。
- **可观测性**：由于所有的消息流量都必须通过 Sidecar，因此可以为 Sidecar 上的消息流量按需增加 Trace 日志、Metrics 采集、消息轨迹数据采集等能力，并借此进一步丰富消息服务的治理能力。

消息 Mesh 化改造

在蚂蚁集团内部，Msgbroker 基于推送模型提供高可靠、高实时、事务消息、Header 订阅等特性，帮助核心链路进行异步解耦，提升业务的可扩展能力，并先后伴随蚂蚁集团众多核心系统一起经历了分布式改造、单元化改造与弹性改造，目前主要承载蚂蚁内部交易、账务、会员、消费记录等核心在线业务的异步消息流量。
由于 Service Mesh 的推进目标也是优先覆盖交易支付等核心链路，为在线业务赋能，因此我们优先选择对 Msgbroker 系统进行 Mesh 化改造。
下文将以 Msgbroker 为例，重点剖析 Mesh 化后，其在整体架构和核心交互流程上的变化，为消息领域的 Mesh 化改造提供参考。

整体架构

消息 Mesh 化后的整体架构，如下图所示：

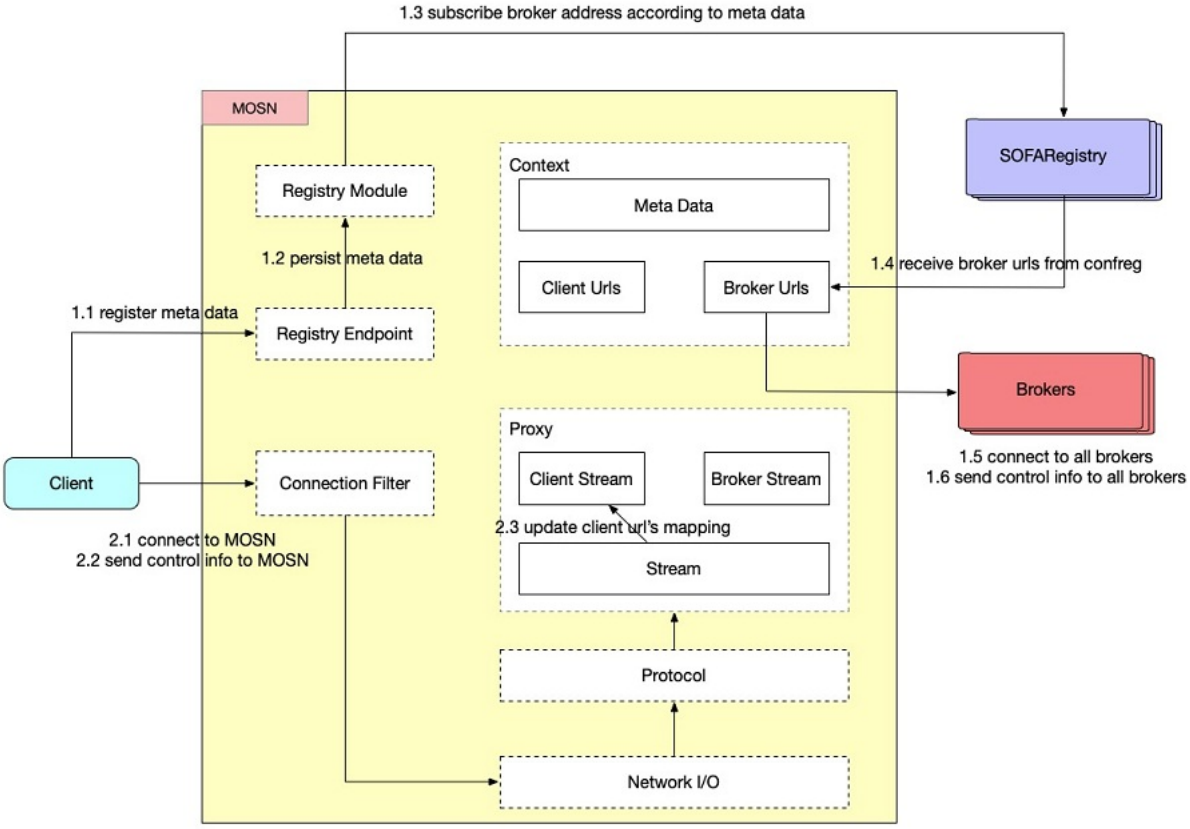


与原有的消息架构相比，主要的变化有：

- 客户端不再与服务端直连，而是通过 Sidecar 进行请求的中转。对客户端而言，Sidecar 实际上是它唯一能感知到的消息服务端；对服务端而言，Sidecar 则扮演着客户端的角色。
- 所有 Sidecar 都会与控制平面交互，接收服务端地址列表、流量管控和调度配置、运行时动态配置等的下发，从而使数据平面具备限流、熔断、异常重试、服务发现、负载均衡、精细化流量调度等能力。

核心交互流程

当 Sidecar 代理了消息客户端的所有请求后，一旦 Sidecar 完成消息服务的发现与服务端/客户端路由数据的缓存，无论是客户端的发消息请求还是服务端的推消息请求，都能由 Sidecar 进行正确的代理转发，而这一切的关键，则是 Sidecar 与消息客户端协同完成一系列的初始化操作。
消息 Mesh 化后具体的初始化流程，如下图所示：



与原有的初始化流程相对比，主要有如下不同：

- 消息服务端的地址处理：
  - Mesh 化前：消息客户端直接向 SOFARegistry 订阅消息服务端的地址。

- Mesh 化后：
  a. 消息客户端将所有消息元数据（包含业务应用声明的消息 Topic、发送/订阅组 GroupId 等关键信息）通过 HTTP 请求上报给 MOSN。
  b. 由 MOSN 进行元数据的持久化（用于 MOSN 异常 Crash 后的恢复）以及消息服务端地址的订阅和处理。

- 客户端收到 MOSN 注册请求响应的处理：
  客户端会主动与 MOSN 建立连接，并将与该连接相关的 Group 集合信息通过控制指令发送给 MOSN。由于客户端与 MOSN 可能存在多个连接，且不同连接上的 Group 集合可以不同，而 MOSN 与同一个消息服务端只维持一个连接，因此控制指令无法向消息数据一样直接进行转发，而是需要汇总计算所有 GroupId 集合后，再统一广播给消息服务端集群。
  由于上述计算逻辑十分复杂，需要包含过滤和聚合，且存在动态和并发行为，一旦因计算错误则会严重影响到消息投递的正确性。因此，当前 MOSN 绕过了该指令的代理，只利用客户端的控制指令进行相关数据的校验，以及更新客户端连接的映射信息（用于 MOSN 的客户端连接选择），选择依赖消息客户端的改造，引入上述 HTTP 注册请求，来构造全量控制指令。

消息 Mesh 的挑战

消息中间件最关键的挑战在于如何在洪峰流量下依然保障消息服务的高可靠与高实时。而在消息 Mesh 化的大规模实施过程中，还需要考虑数十万的容器节点对数据面整体稳定性和控制面性能带来的巨大挑战。这些都依赖于完善的 Sidecar 运维体系，包括 Sidecar 的资源分配策略，以及 Sidecar 独立变更升级的策略。

资源管理

当 Service Mesh 的实体 MOSN 作为 Sidecar 与业务容器部署在一起时，就不再像消息服务端一样，只需要关心自身的资源消耗，而是必须精细化其 CPU、内存等资源的分配，才能达到与应用最优的协同合作方式。Sidecar 的精细化资源管理经历了下述多个阶段：

- 独立分配
- 通过超卖与业务容器共享
- 细粒度的 CPU 资源分配策略
- 内存 OOM 策略调整等

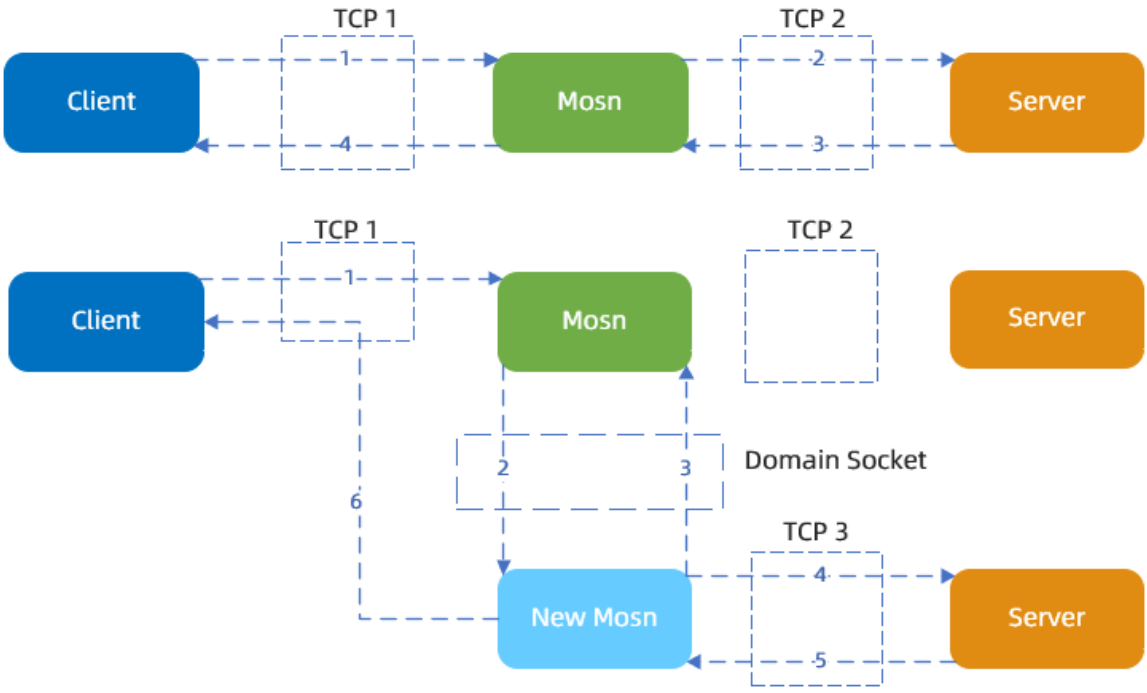
Service Mesh 将原有的与业务无关的逻辑下沉到 Sidecar，其占用的资源实际是原来业务容器会使用的资源，基于这一假设，在零新增成本的情况下，Service Mesh 平稳支撑了数十万规模级别的 Sidecar 容器分配。

平滑升级

为了达到 Sidecar 这一类基础设施的变更升级对业务完全无感知的目的，就需要使 MOSN 具备平滑升级的能力。

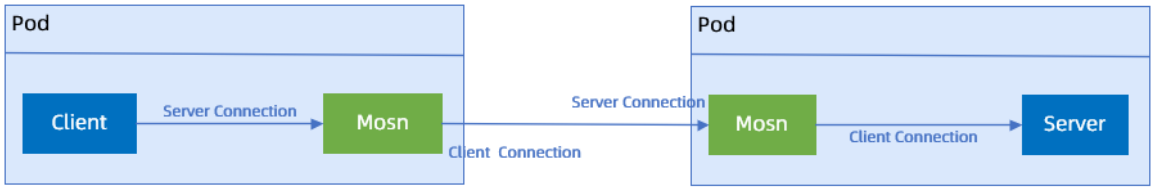
- 平滑升级的隐含前提条件：单条连接上的请求必须是单向的。
- 平滑升级的过程为：新的 MOSN 首先会被注入，并通过共享卷的 UnixSocket 检查是否存在老的 MOSN。若存在，则利用内核 Socket 的迁移实现老 MOSN 的连接全部迁移给新 MOSN，最终让老 MOSN 优雅退出，从而实现 MOSN 在整个升级和发布过程中对业务无任何打扰。

平滑升级过程示意图



关于 MOSN 本身平滑升级更多的内容，请参考 服务网格最佳实践之核心篇。

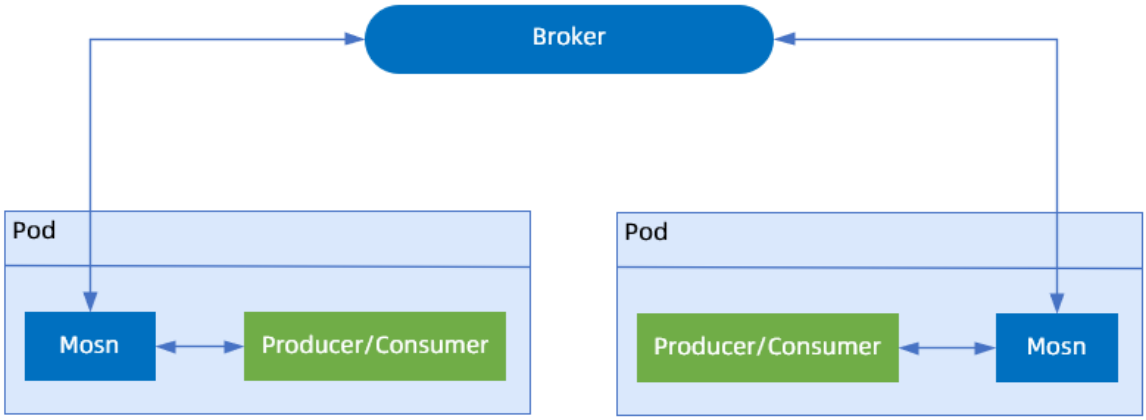
平滑升级方案与 RPC



RPC 可以直接使用上述平滑升级方案，理由如下：

- 其单条连接的角色是固定的，只能是服务端连接或客户端连接。
- 对一次请求的代理过程也是固定的，总是从服务端连接上收到一个请求，再从客户端连接将请求转发出去。

平滑升级方案与 MsgBroker



在消息场景特别是 Msgbroker 场景下，MOSN 上的连接请求实际上是双向的：

- 客户端会使用该连接进行消息的发送。
- 服务端也会利用该连接将消息主动推送给 MOSN。

这会给连接迁移带来新的问题和挑战：

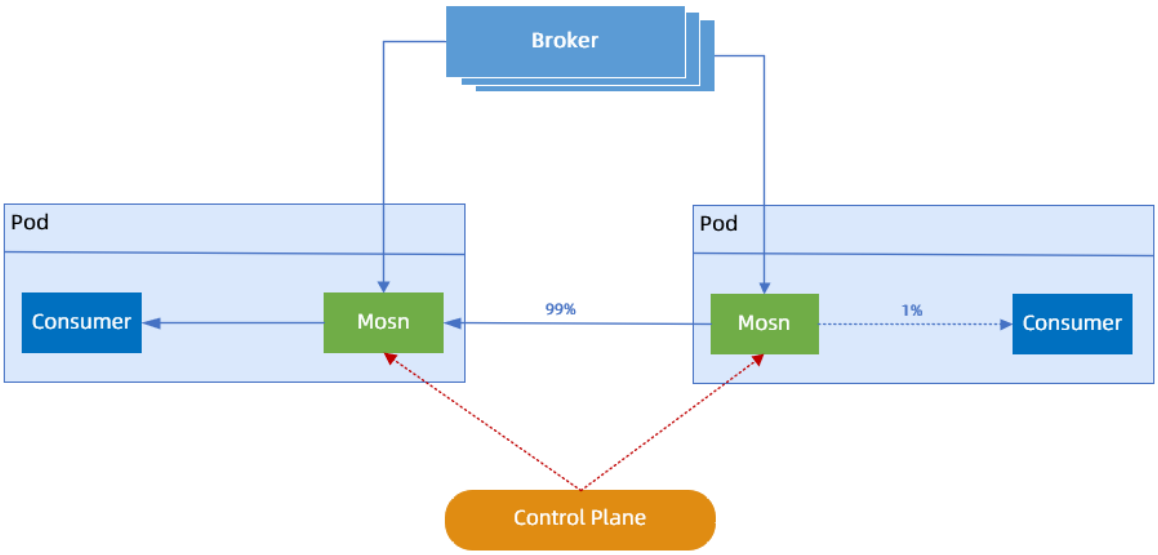
- 在连接迁移的过程中，如果消息客户端已处理完经过 MOSN 转发的服务端投递消息请求，但是还未回复响应，此时若把连接迁移到新的 MOSN，则新的 MOSN 将收到上述响应，但由于新 MOSN 缺失上下文，无法将该响应返回给正确的消息服务端。
- 在连接迁移完成，但老 MOSN 还未优雅退出期间，由于两个 MOSN 与消息服务端都存在连接，两者都会收到服务端发送的投递消息请求，但因两个 MOSN 与服务端连接的状态各自独立，可能会使客户端收到的请求 ID 相冲突。

解决上述问题的思路其实很简单，即在平滑升级的过程中，禁止服务端向老 MOSN 发送投递消息请求，以保证即使在消息场景整个平滑升级过程中，所有连接仍然是单工通信的。对平滑升级流程的具体改动，说明如下：

- 老 MOSN 平滑升级指令后，会立即向所有的消息服务端发送禁止再接收消息的控制指令。
- 新 MOSN 感知老 MOSN 完成前置操作，开始进行原有的平滑升级流程，进行初始化和存量连接迁移。
- 新 MOSN 完成存量连接迁移后，向所有的消息服务端发送接收消息的控制指令，开始正常的消息订阅。

消息 Mesh 流量调度

消息 Mesh 的流量调度，示例如下：



流量调度流程说明如下：

1. 控制平面会将与流量调度相关的规则下发至 MOSN，规则主要包含该应用下所有容器节点的 IP 地址与流量权重，这是能够进行精细化流量调度的前提。
2. 当 MOSN 收到消息投递请求时，会判断请求的来源：
   - 若来自于其他 MOSN 节点，则会直接将该请求转发给客户端，避免消息投递请求的循环转发。
   - 若来自于消息服务端，则 MOSN 会根据自身的流量权重来决定下一步的路由：
     - 若自身的流量权重是 100%，会同样将该请求转发给客户端。
     - 若自身权重小于 100%，则会按照配置的权重，将剩余请求均匀转发给其他流量权重为 100% 的 MOSN 节点。

3. 与 RPC 的点对点通信方式不同，无论是消息发送端还是订阅端，都只与消息服务端通信。这意味着：
    - 即使进行了消息 Mesh 化改造后，MOSN 也只与消息服务端通信。
    - 同一个应用的 MOSN 节点之间是不存在消息连接的。
    - 为了实现 MOSN 之间的消息流量转发，需要内置实现一个与业务应用进程同生命周期的消息转发服务，由同应用内的所有其他 MOSN 节点订阅，并在需要转发时调用。

消息 Mesh 经过蚂蚁消息中间件团队大半年的打磨和沉淀，已经迈出了坚实的一大步：在开源社区迟迟未在消息 Mesh 上取得实质性进展时，蚂蚁团队已经为蚂蚁内部主流消息中间件打通了数据平面。
同时，依赖消息的精细化流量调度，预期可以发掘出更大的业务价值，包括：

- 基于事件驱动的 Serverless 化应用多版本流量管理
- 流量着色
- 分组路由
- 细粒度的流量灰度与 A/B 策略

未来，蚂蚁将会持续加大对消息 Mesh 的投入，为消息 Mesh 支持更多的消息协议，赋予更多开箱即用的的消息流量管控和治理能力，并进一步结合 Knative 探索消息精细化流量调度在 Serverless 下的应用场景。

4.2.7.8. 服务网格最佳实践之 Operator

Service Mesh 是蚂蚁集团下一代技术架构的核心，也是蚂蚁集团内部双十一应用云化的重要一环，本文主要分享在蚂蚁集团当前的体量下，如何支撑应用从现有微服务体系大规模演进到 Service Mesh 架构，并平稳落地。

为什么需要 Service Mesh ?

使用 Service Mesh 之前

SOFAStack 作为蚂蚁集团微服务体系下服务治理的核心技术栈，通过提供若干中间件来实现服务发现和流量管控等能力，例如：

- Cloud Engine 应用容器
- SOFABoot 编程框架（已开源）
- SOFARPC（已开源）

经过若干年的严苛金融场景的锤炼，SOFAStack 已经具备极高的可靠性和可扩展性。通过开源共建，也已形成了良好的社区生态，能够与其他开源组件相互替换和集成。在研发迭代上，中间件类库已经与业务解耦。但是，由于运行时两者在同一个进程内，这意味着在基础库升级时，需要推动业务方升级对应的中间件版本。

使用 Service Mesh 之后

蚂蚁团队引入的 Service Mesh，将原先通过类库形式提供的服务治理能力进行提炼和优化后，下沉到与业务进程协同，但独立运行的 Sidecar Proxy 进程中，大量的 Sidecar Proxy 构成了一张规模庞大的服务网络，为业务提供一致的、高质量的用户体验。同时，也实现了服务治理能力在业务无感的条件下独立进行版本迭代的目标。

应用 Service Mesh 的挑战

Service Mesh 提供的能力很美好，但现实的挑战同样很多，例如：

- 数据面技术选型和私有协议支持。
- 控制面与蚂蚁集团内部现有系统对接。
- 配套监控运维体系建设。
- 在调用链路增加两跳的情况下，如何优化请求延迟和资源使用率等等。

什么是 Operator?

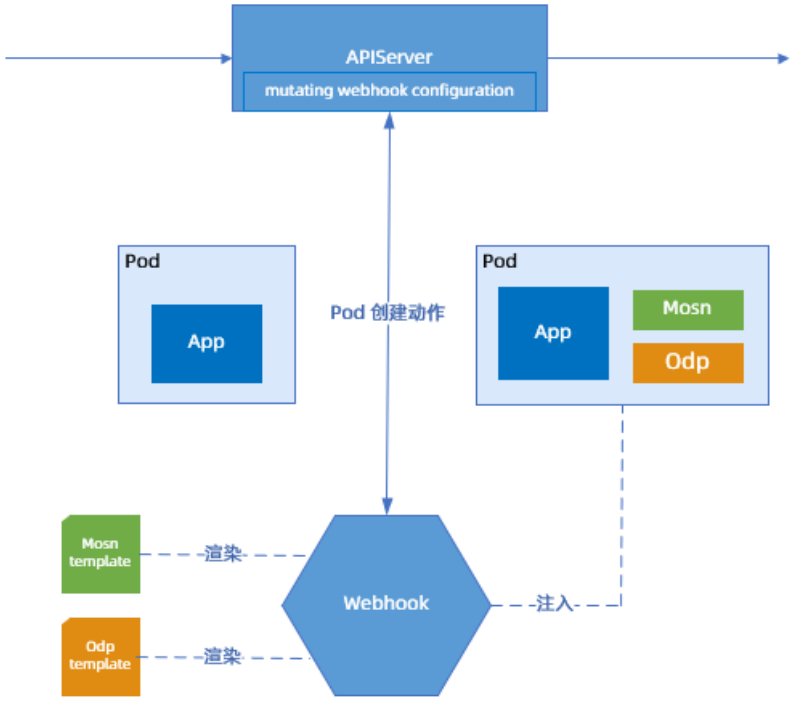
如果说 Kubernetes 是 “操作系统” 的话，Operator 是 Kubernetes 的第一层应用，它部署在 Kubernetes 里，使用 Kubernetes “扩展资源” 接口的方式向更上层用户提供服务。
本文将从 MOSN（Sidecar Proxy）的运维和风险管控方面，分享 Operator 的实践经验。

Sidecar 运维

注入

创建注入

已经完成容器化改造且运行在 Kubernetes 中的应用，接入到 Service Mesh 体系中的方式中，最简单的方式为：在应用发布阶段，通过 Mutating Webhook 拦截 Pod 创建请求，在原始 Pod Spec 的基础上，为 Pod 注入一个新的 MOSN 容器。该方案也是以 Istio 为代表的 Service Mesh 社区方案所采用的，示例如下：



初始配置

在资源分配上，起初依据经验值，在应用 8 GB 内存的场景下，为 Sidecar 分配了 512 MB 内存，即

- App: req=8G, limit=8G
- Sidecar: req=512M, limit=512M

但是，这种分配方案带来了一些问题：

- 部分流量比较高的应用，其 MOSN 容器出现了严重的内存不足甚至 OOM。
- 注入进去的 Sidecar 容器额外向调度器申请了一部分内存资源，这部分资源脱离了业务的 Quota 管控。

应对策略

为了消除内存 OOM 风险和避免业务资源容量规划上的偏差，蚂蚁团队制定了新的 “共享内存” 策略。该策略主要内容：

- Sidecar 的内存 request 被置为 0，不再向调度器额外申请资源。
- Limit 被设置为应用的 1/4，保障 Sidecar 在正常运行的情况下，有充足的内存可用。

为了确实达到 “共享” 的效果，蚂蚁集团针对 Kubelet 做了调整，使之在设置 Sidecar 容器 Cgroups Limit 为应用 1/4 的同时，保证整个 Pod 的 Limit 没有额外增加。

新风险及解决方案

在上述应对策略下，会出现新的风险，蚂蚁也提出了对应的解决方案，说明如下：

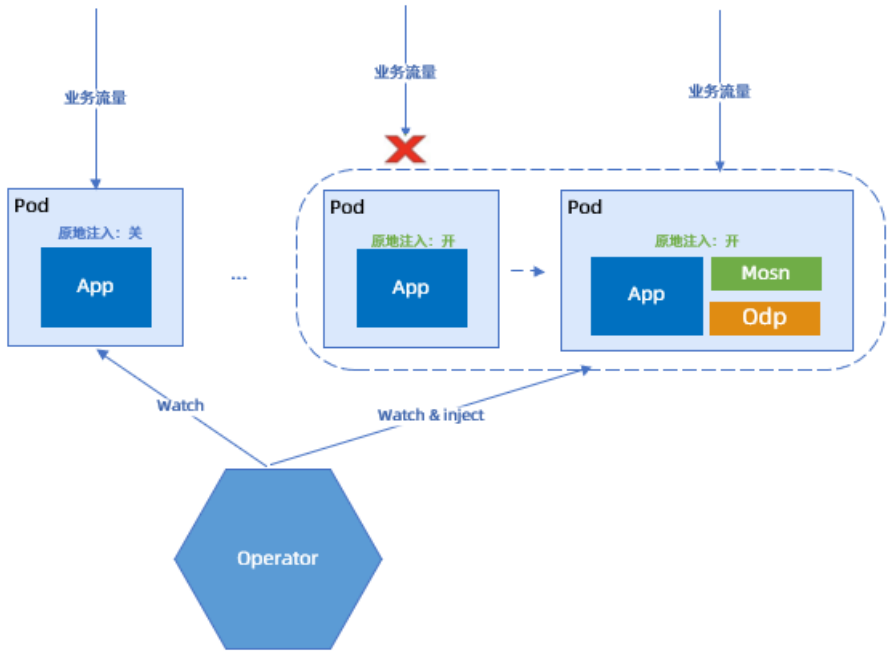
- **风险**：Sidecar 与应用“共享”分配到的内存资源，导致在异常情况（比如内存泄露）下，Sidecar 跟应用抢内存资源。
  解决方案：通过扩展 Pod Spec（即相应的 apiserver、Kubelet 链路），为 Sidecar 容器额外设置了 `Linux oom_score_adj` 这个属性，以保障在内存耗尽的情况下，Sidecar 容器会被 OOM Killer 更优先选中，从而让 Sidecar 比应用能够更快速重启，从而更快恢复到正常服务。
- **风险**：在 CPU 资源的分配上，可能出现 MOSN 抢占不到 CPU 资源从而导致请求延迟大幅抖动。
  解决方案：确保在注入 Sidecar 时，根据 Pod 内的容器数量，为每个 Sidecar 容器计算出相应的 cpushare 权重，通过工具扫描并修复全站所有未正确设置的 Pod。

原地注入

原地注入的背景为下述几个方面：

- **接入方式**：相对比较简单的接入方式是在创建 Pod 的时候注入 Sidecar。此时应用只需先扩容，再缩容，就可以逐步用带有 Sidecar 的 Pod，替换掉旧的没有 Sidecar 的 Pod。
- **存在的问题**： 在大量应用、大规模接入的时候，需要集群有较大的资源 Buffer 来供应用实例进行滚动替换，否则替换过程将变得十分艰难且漫长。
- **蚂蚁目标**：双十一大促不加机器，提高机器使用率。

为了解决存在的问题并实现预期目标，蚂蚁团队提出了“原地注入”的概念，即在 Pod 不销毁，不重建的情况下，原地把 Sidecar 注入进去。
原地注入的步骤如下图所示：



1. 在 PaaS 提交工单，选择一批需要原地注入的 Pod。
2. PaaS 调用中间件接口，关闭业务流量并停止应用容器。
3. PaaS 以 Annotation 的形式打开 Pod 上的原地注入开关。
4. Operator 观察到 Pod 原地注入开关打开，渲染 Sidecar 模版，注入到 Pod 中，并调整 CPU、Memory 等参数。
5. Operator 将 Pod 内的容器期望状态置为运行。
6. Kubelet 将 Pod 内的容器重新拉起。

7. PaaS 调用中间件接口，打开业务流量。

升级

蚂蚁团队将 RPC 等能力从基础库下沉到 Sidecar 之后，基础库升级与业务绑定的问题虽然消除了，但是这部分能力的迭代需求依然存在，只是从升级基础库变成了如何升级 Sidecar。

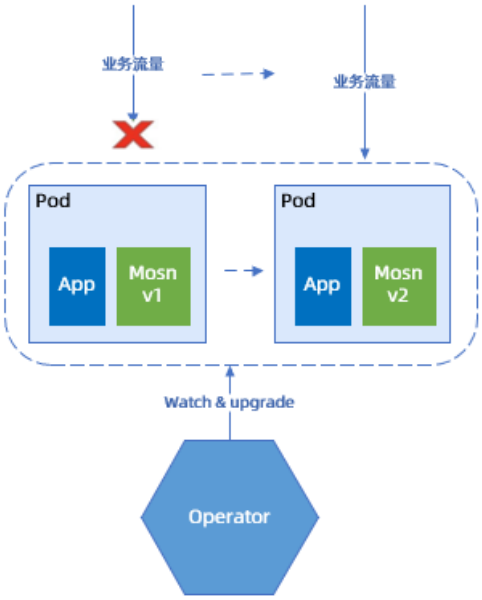
替换升级

最简单的升级就是替换，即销毁 Pod 并重新创建，这样新建出来的 Pod 所注入的 Sidecar 自然就是新版本了。

但通过替换的升级方式，与创建注入存在相似的问题，即需要大量的资源 Buffer，并且，这种升级方式对业务的影响最大，也最慢。

非平滑升级

为了避免销毁重建 Pod，蚂蚁团队通过 Operator 实现了“非平滑升级”能力，示例如下。



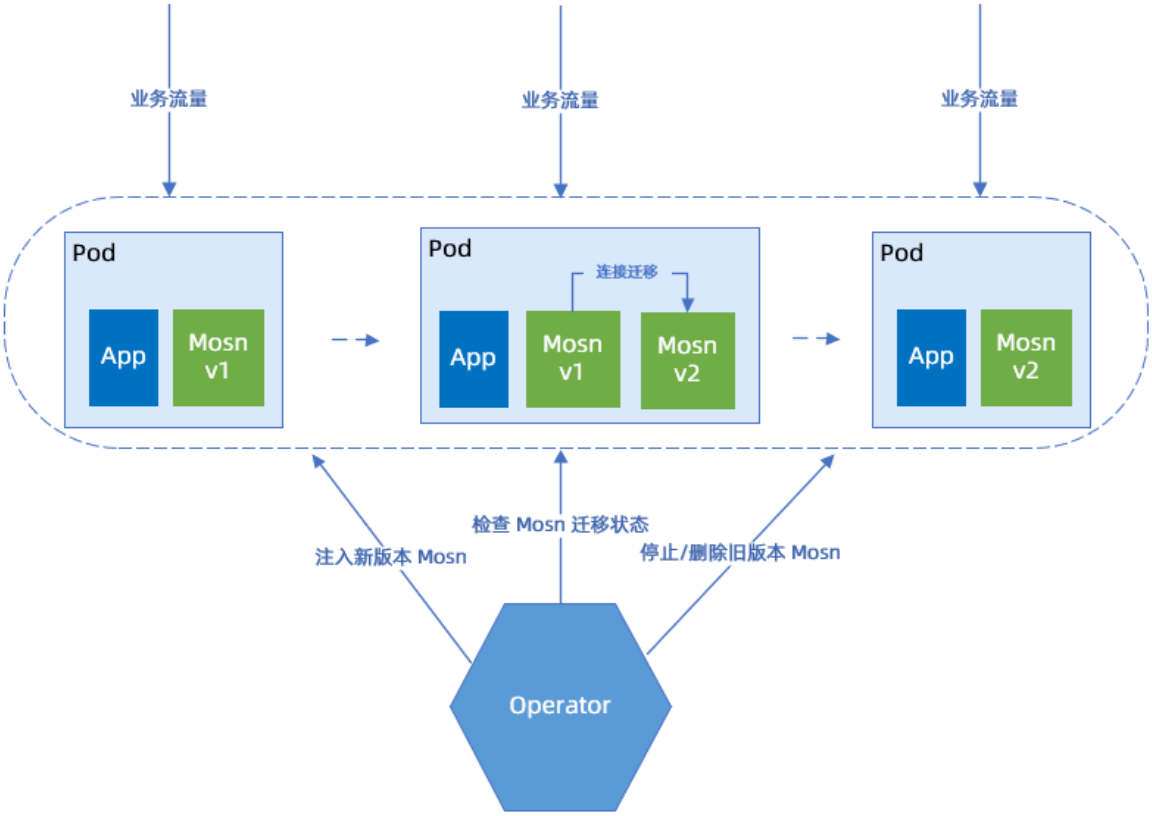
非平滑升级步骤：

1. PaaS 关流量，停容器。
2. Operator 替换 MOSN 容器为新版本，重新拉起容器。
3. PaaS 重新打开流量。

原地升级 Pod 打破了 Kubernetes Immutable Infrastructure 的设计，为了能够实现预期目标，蚂蚁团队修改了 apiserver validation 和 admission 相关的逻辑，以允许修改运行中的 Pod Spec，也修改了 Kubelet 的执行逻辑以实现容器的增、删、启、停操作。

平滑升级

为了进一步降低 Sidecar 升级对应用带来的影响，蚂蚁团队针对 MOSN Sidecar 开发了“平滑升级”能力，以实现在 Pod 不重建、流量不关停，应用无感知的条件下对 MOSN 进行版本升级。



- **平滑升级原理**：Operator 通过注入新 MOSN，等待 MOSN 自身连接和 Metrics 数据迁移的完成，再停止并移除旧 MOSN，来达到应用无感、流量无损的效果。
- **努力方向**：
  - 提高成功率。
  - 改进 Operator 的状态机来提升性能。

回滚

为了确保大促活动万无一失，蚂蚁团队还提供了 Sidecar 回滚的保底方案，以备在识别到 Service Mesh 出现严重问题的情况下，迅速将应用回滚到未接入 Sidecar 的状态，通过应用原先的能力继续提供业务服务。

风险管控

主要从下述几个角度来分析：

- **技术风险**：关于 Sidecar 的所有运维操作，都要具备三板斧能力。在灰度能力上，Operator 为升级等运维动作增加了显式的开关，确保每个执行动作符合用户和 SRE（Site Reliability Engineer，简称 SRE）的期望，避免不受控制地或不被察觉地自动执行变更操作。
- **监控**：在基本的操作成功率统计、操作耗时统计、资源消耗等指标之外，仍需以快速发现问题、快速止血为目标，继续完善精细化监控。

Operator 目前对外提供的几个运维能力，细节上都比较复杂，一旦出错，影响面又很大，因此单元测试覆盖率和集成测试场景覆盖率，也会是后续 Service Mesh 稳定性建设的一个重要的点去努力完善。

对未来的思考

演进到 Service Mesh 架构后，保障 Sidecar 自身能够快速、稳定地迭代十分重要。未来蚂蚁会向下述几个方向进行发力：

- 继续增强 Operator 的能力。
- 可能通过以下几个优化手段，来做到更好的风险控制：

- 对 Sidecar 模板做版本控制，由 Service Mesh 控制面，而非用户来决定某个集群下某个应用的某个 Pod 应该使用哪个版本的 Sidecar。这样既可以统一管控全站的 Sidecar 运行版本，又可以将 Sidecar 二进制和其 Container 模板相绑定，避免出现意外的、不兼容的升级。
- 提供更加丰富的模板函数，在保持灵活性的同时，简化 Sidecar 模板的编写复杂度，降低出错率。
- 设计更完善的灰度机制，在 Operator 出现异常后，快速熔断，避免故障范围扩大。
- 持续思考：整个 Sidecar 的运维方式能否更加“云原生”。

4.2.7.9. 服务网格最佳实践之运维

Service Mesh 在 2019 年得到了大规模的应用与落地，截止目前，蚂蚁集团的 Service Mesh 数据平面 MOSN 已接入应用数百个，接入容器数量达数十万，是目前已知的全世界最大的 Service Mesh 集群。同时，在刚刚结束的双十一大促中，Service Mesh 的表现也十分亮眼，RPC 峰值 QPS 达到了几千万，消息峰值 TPS 达到了几百万，且引入 Service Mesh 后的平均 RT 增长幅度控制在 0.2 ms 以内。
本文将主要分享大规模服务网格，在蚂蚁集团当前体量下，落地到支撑蚂蚁金服双十一大促过程中，运维所面临的挑战与演进。

云原生化的选择与问题

传统的 Service Mesh：

- 在软件形态上：将中间件的能力从框架中剥离成独立软件。
- 在具体部署上：保守的做法是以独立进程的方式与业务进程共同存在于业务容器内。

蚂蚁集团从开始就选择了拥抱云原生。

Sidecar 模式

业务容器内独立进程的优缺点为：

- 优点：与传统的部署模式兼容，易于快速上线。
- 缺点：强侵入业务容器，对于镜像化的容器更难于管理。

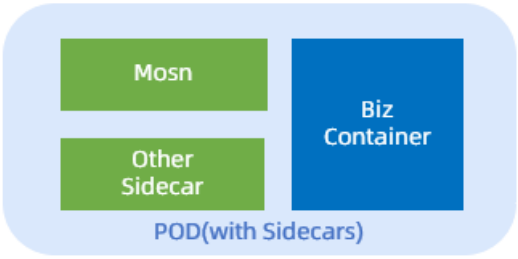
而云原生化，可以解决独立进程的缺点，并带来一些优点：

- 可以将 Service Mesh 本身的运维与业务容器解耦开来，实现中间件运维能力的下沉。
- 在业务镜像内，仅仅保留长期稳定的 Service Mesh 相关 JVM 参数，从而仅通过少量环境变量完成与 Service Mesh 的联结。
- 考虑到面向容器的运维模式的演进，接入 Service Mesh 还同时要求业务完成镜像化，为进一步的云原生演进打下基础。

	优势	劣势
独立进程	兼容传统的部署模式改造成本低快速上线	侵入业务容器镜像化难于运维
Sidecar	面向终态运维解耦	依赖 K8s 基础设施运维环境改造成本高应用需要镜像化改造

在接入 Service Mesh 之后，一个典型的 Pod 结构可能包含多个 Sidecar：

- MOSN：RPC Mesh、MSG Mesh 等。
- 其它 Sidecar。



这些 Sidecar 容器，与业务容器共享相同的网络 Namespace，使得业务进程可以从本地端口访问 Service Mesh 提供的服务，保证了与保守做法一致的体验。

基础设施云原生支撑

蚂蚁团队也在基础设施层面同步推进了面向云原生的改造，以支撑 Service Mesh 的落地。

业务全面镜像化

首先是在蚂蚁集团内部推进了全面的镜像化，完成了内部核心应用的全量容器的镜像化改造。改造点包括：

- 基础镜像层面增加对于 Service Mesh 的环境变量支撑。
- 应用 Dockerfile 对于 Service Mesh 的适配。
- 由于历史原因，前后端分离管理的静态文件还有一些存量，蚂蚁团队推进解决了存量文件的镜像化改造。
- 对于大量使用前端区块分发的应用，进行了推改拉的改造。
- 大批量的 VM 模式的容器升级与替换。

容器 Pod 化

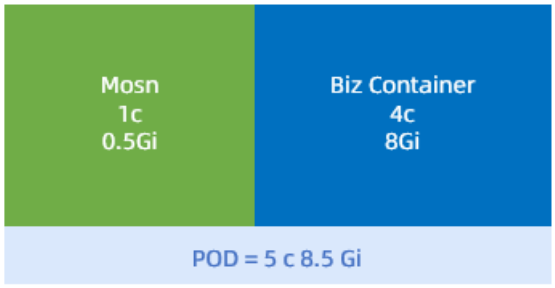
除了业务镜像层面的改造，Sidecar 模式还需要业务容器全部跑在 Pod 上，来适应多容器共享网络。由于直接升级的开发和试错成本很高，接入 Service Mesh 的数百个应用有数万个非 K8S 容器，蚂蚁团队最终选择通过大规模扩缩容的方式，将其全部更换成了 K8S Pods。
经过这两轮改造，蚂蚁团队在基础设施层面同步完成了面向云原生的改造。

对资源模型的挑战

Sidecar 模式带来的一个重要的问题是：如何分配资源。

理想比例的假设

最初的资源设计基于内存无法超卖的现实。蚂蚁做了一个假设：MOSN 的基本资源占用与业务选择的规格同比例。即 CPU 和 Memory 申请的额外资源与业务容器成相应比例，这一比例最后设定在了 CPU 1/4，Memory 1/16。
此时一个典型 Pod 的资源分配如下图示：



理想比例的缺陷

理想比例的假设带来了两个问题：

- 蚂蚁集团已经实现了业务资源的 Quota 管控，但 Sidecar 并不在业务容器内，Service Mesh 容器成为了一个资源泄漏点。

- 由于业务多样性，部分高流量应用的 Service Mesh 容器出现了严重的内存不足和 OOM 情况。

为了快速支撑 Service Mesh 在非云环境的铺开，上线了原地接入 Service Mesh。而原地接入 Service Mesh 的资源无法额外分配，在内存不能超卖的情况下，采取了二次分割的分配方式。此时的 Pod 内存资源分配方式为：

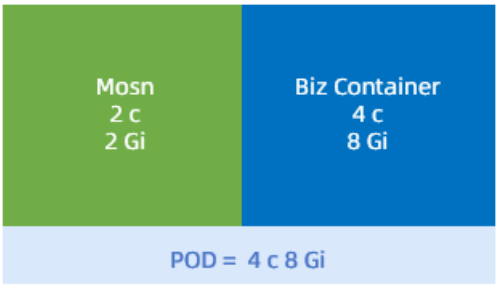
- 1/16 内存给 Sidecar。
- 15/16 内存给业务容器。

还有一些新的问题：

- 业务可见内存不一致。
- 业务监控偏差。
- 业务进程 OOM 风险。

解决方案

为了解决上述问题，蚂蚁团队追加了一个假设：在接入 Service Mesh 之前，业务已使用的资源，才是 Service Mesh 容器占用的资源。接入 Service Mesh 的过程，同时也是一次资源置换。
基于这个假设，推进了调度层面支持 Pod 内的资源超卖。Service Mesh 容器的 CPU、MEM 都从 Pod 中超卖出来，业务容器内仍然可以看到全部的资源。
新的资源分配方案，示例如下：



新的分配方案考虑了下述因素：

- 内存超卖
- 引入了 Pod OOM 的风险

因此，对于 Sidecar 容器还调整了 OOM Score，保证在内存不足时，通过 Service Mesh 进程比 Java 业务进程启动更快，从而更降低影响。
同时，新的分配方案还解决了以上两个问题，并且平稳支持了大促前的多轮压测。

重建

新的分配方案上线时，Service Mesh 已经在弹性建站时同步上线。同时，蚂蚁团队还发现：在一些场景下，Service Mesh 容器无法抢占到 CPU 资源，导致业务 RT 出现了大幅抖动。原因是：在 CPU Share 模式下，Pod 内默认并没有将等额的CPU Quota 分配给 Sidecar。
于是又产生了两个新问题：

- 存量的已分配 Sidecar 仍有 OOM 风险。
- Sidecar 无法抢占到 CPU。

蚂蚁已经无法承受更换全部 Pod 的代价。最终在调度的支持下，通过手动对 Pod Annotation 进行重新计算及修改，在 Pod 内进行了全部资源的重分配，来修复这两个风险。最终修复的容器总数约为 25 w 个。

大规模场景下运维设施的演进

Service Mesh 的变更包括了接入与升级，所有变更底层都是由 Operator 组件来接受上层写入到 Pod annotation 上的标识，然后通过对相应 Pod Spec 进行修改来完成，这是典型的云原生的方式。由于蚂蚁集团的资源现状与运维需要，又发展出了原地接入与平滑升级。

接入

有 2 种接入方式：

- 创建接入：最初 Service Mesh 接入只提供了创建时注入 Sidecar。
- 原地接入：后来引入了原地接入，主要是为了支撑大规模的快速接入与回滚。

2 种接入方式的优缺点如下：

- 创建接入：
  - 资源替换过程需要大量 Buffer。
  - 回滚困难。
- 原地接入：
  - 不需要重新分配资源。
  - 可原地回滚。

原地接入/回滚需要对 Pod Spec 进行精细化的修改，实践中发现了很多问题，当前能力只做了小范围的测试。

升级

Service Mesh 是深度参与业务流量的，因此最初的 Sidecar 的升级方式也需要业务伴随重启。这个过程看似简单，蚂蚁却遇到了 2 个严重问题：

- Pod 内的容器启动顺序随机，导致业务无法启动。这个问题最终通过调度层修改启动逻辑来解决：Pod 内需要优先等待所有 Sidecar 启动完成。但是，这导致了下述第二所述的新问题。
- Sidecar 启动慢了，上层超时。此问题仍在解决中。

Sidecar 中，MOSN 提供了更为灵活的平滑升级机制：由 Operator 控制启动第二个 MOSN Sidecar，完成连接迁移，再退出旧的 Sidecar。小规模测试显示，整个过程业务可以做到流量不中断，几近无感。目前平滑升级同样涉及到 Pod Spec 的大量操作，考虑到大促前的稳定性，目前此方式未做大规模应用。

规模化的问题

在逐渐达到大促状态的过程中，接入 Service Mesh 的容器数量开始大爆炸式增加。容器数量从千级别迅速膨胀到10 w +，最终达到全站数十万容器规模，并在膨胀后还经历了数次版本变更。
快速奔跑的同时，缺少相应的平台能力也给大规模的 Sidecar 运维带来了极大挑战：

- 版本管理混乱：
  - Sidecar 的版本与应用 / Zone 的映射关系维护在内部元数据平台的配置中。大量应用接入后，全局版本、实验版本、特殊 Bugfix 版本等混杂在多个配置项中，统一基线被打破，难于维护。
- 元数据不一致：
  - 元数据平台维护了 Pod 粒度的 Sidecar 版本信息，但是由于 Operator 是面向终态的，会出现元数据与底层实际不一致的情况，当前仍依赖巡检发现。
- 缺少完善的 Sidecar ops 支撑平台：
  - 缺少多维度的全局视图。
  - 缺少固化的灰度发布流程。
  - 依赖于人工经验配置管理变更。
- 监控噪声巨大。

目前，Service Mesh 与 PaaS 的开发团队都正在在建设相应的能力，这些问题正得到逐步的缓解。

周边技术风险能力的建设

监控能力

蚂蚁的监控平台为 Service Mesh 提供了基础的监控能力和大盘，以及应用维度的 Sidecar 监控情况，包括：

- 系统监控：
  - CPU
  - MEM
  - LOAD
- 业务监控：
  - RT
  - RPC 流量
  - MSG 流量
  - Error 日志监控

Service Mesh 进程还提供了相应的 Metrics 接口，提供服务粒度的数据采集与计算。

巡检

在 Service Mesh 上线后，巡检也陆续被加入：

- 日志 Volume 检查
- 版本一致性检查
- 分时调度状态一致性检查

预案与应急

Service Mesh 自身具备按需关闭部分功能的能力，当前通过配置中心实现下述功能：

- 日志分级降级
- Tracelog 日志分级降级
- 控制面（Pilot）依赖降级
- 软负载均衡长轮询降级

对于 Service Mesh 依赖的服务，为了防止潜在的抖动风险，也增加了相应的预案：

- 软负载均衡列表停止变更。
- 服务注册中心高峰期关闭推送。

Service Mesh 是非常基础的组件，目前的应急手段主要是下述重启方式：

- Sidecar 单独重启
- Pod 重启

变更风险防控

除了传统的变更三板斧之外，蚂蚁还引入了无人值守变更，对 Service Mesh 变更做了自动检测、自动分析与变更熔断。
无人值守变更防控主要关注变更后对系统和业务和影响，串联了多层检测，主要包括：

- 系统指标：机器内存、磁盘、CPU。
- 业务指标：业务和 Service Mesh 的 RT、QPS 等。
- 业务关联链路：业务上下游的的异常情况。
- 全局的业务指标。

经过这一系列防控设施，可以在单一批次变更内，发现和阻断全站性的 Service Mesh 变更风险，避免了风险放大。

未来展望

Service Mesh 在快速落地的过程中，遇到并解决了一系列的问题，但同时也要看到还有更多的问题亟待解决。做为下一代云原生化中间件的核心组件之一，Service Mesh 的技术风险能力还需要持续的建议与完善。

未来需要在下述领域持续建设：

- 大规模高效接入与回滚能力支撑。
- 更灵活的变更能力，包括业务无感的平滑/非平滑变更能力。
- 更精准的变更防控能力。
- 更高效，低噪声的监控。
- 更完善的控制面支持。
- 应用维度的参数定制能力。