SOFAStack

服务网格

技术白皮书

产品版本：AntStack Plus 1.13.1

文档版本：20230707

蚂蚁集团
ANT GROUP

法律声明

蚂蚁集团版权所有©2022，并保留一切权利。

商标声明

免责声明

通用约定

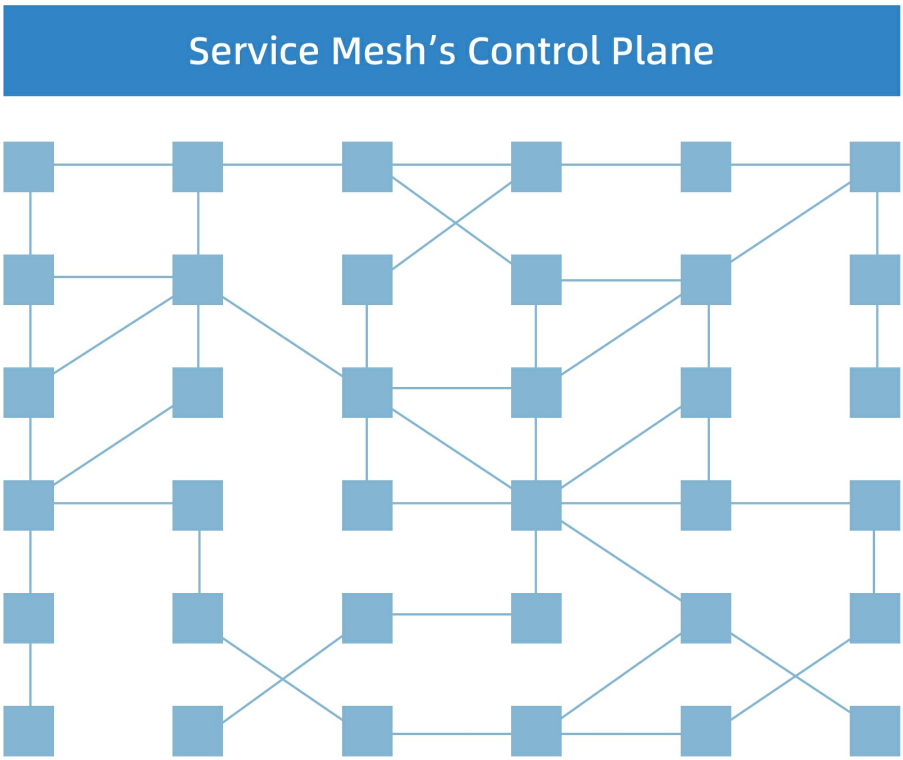
格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**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

1.什么是服务网格

1.1. 产品简介

服务网格 Service Mesh 是一个基础设施层，用于处理服务间通讯。现代云原生应用有着复杂的服务拓扑，服务网格负责在这些拓扑中实现请求的可靠传递。在实践中，服务网格通常实现为一组轻量级网络代理，它们与应用程序部署在一起，而对应用程序透明。服务网格就像是应用程序或者说微服务间的 TCP/IP，负责服务之间的网络调用、限流、熔断和监控。



1.2. 产品背景

Service Mesh 概念一经提出，就受到了业界高度的关注，究其原因是因为相比于传统微服务体系，Service Mesh 解决了如下几个业务痛点：

服务治理与业务逻辑耦合严重

在 Service Mesh 之前，微服务体系的使用方式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。在运行时，SDK 和业务应用的代码其实是混合在一个进程中运行的，耦合度非常高，这就带来了如下问题：

- 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。

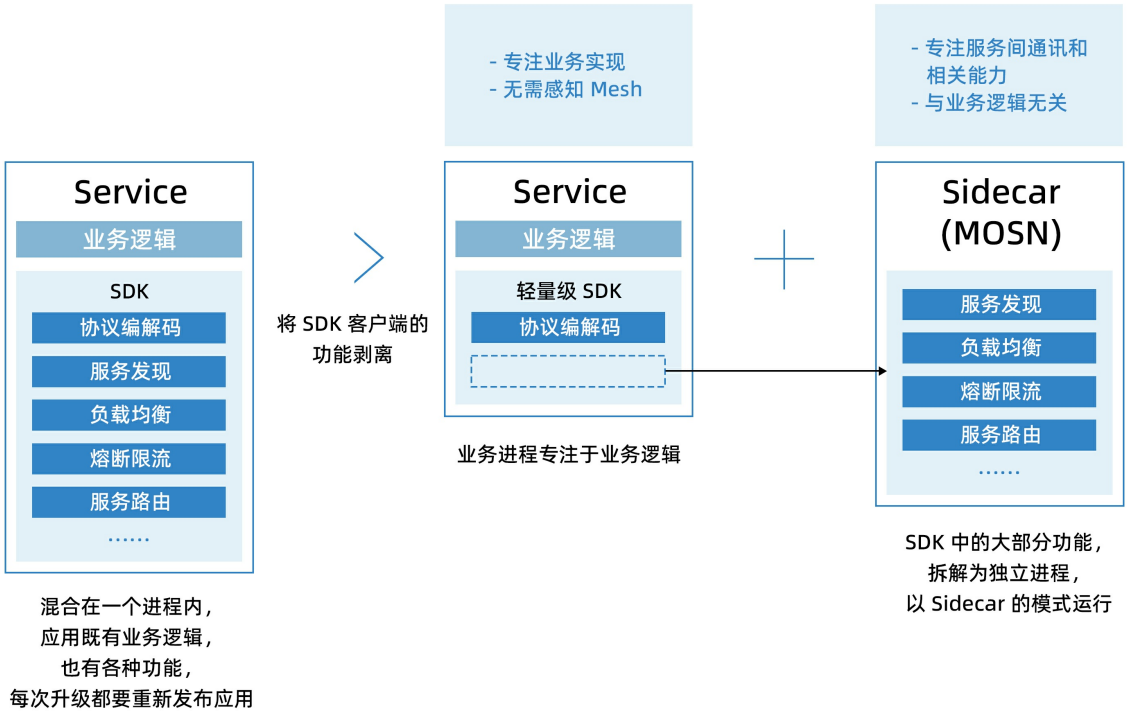
- 版本碎片化严重

  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，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就需要在代码中兼容各种各样的老版本逻辑，无法实现快速迭代。

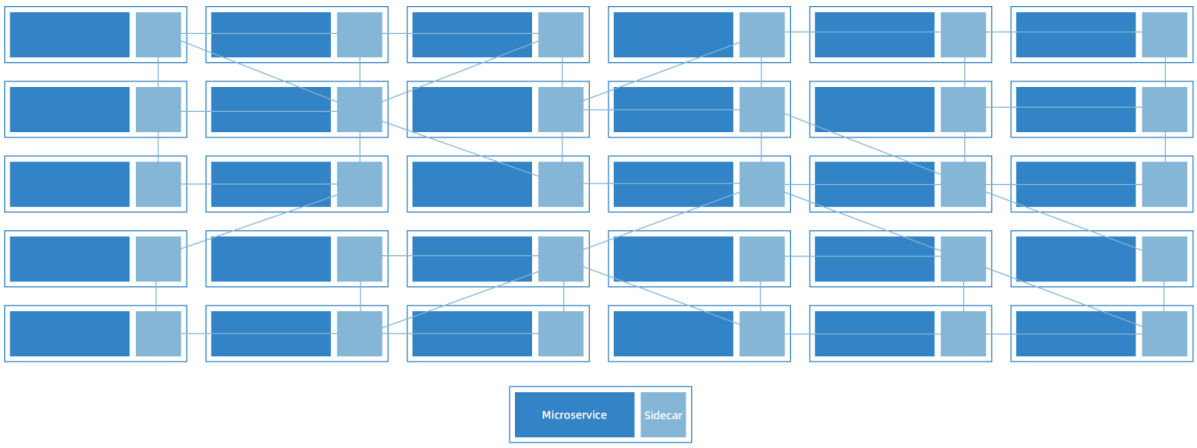
有了 Service Mesh 之后，就可以把 SDK 中的大部分能力从应用中剥离出来，拆解为独立进程，以 Sidecar 的模式部署。通过将服务治理能力下沉到基础设施，可以让业务更加专注于业务逻辑，中间件团队则更加专注于各种通用能力，真正实现独立演进、透明升级、提升整体效率。



异构系统难以统一治理

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。

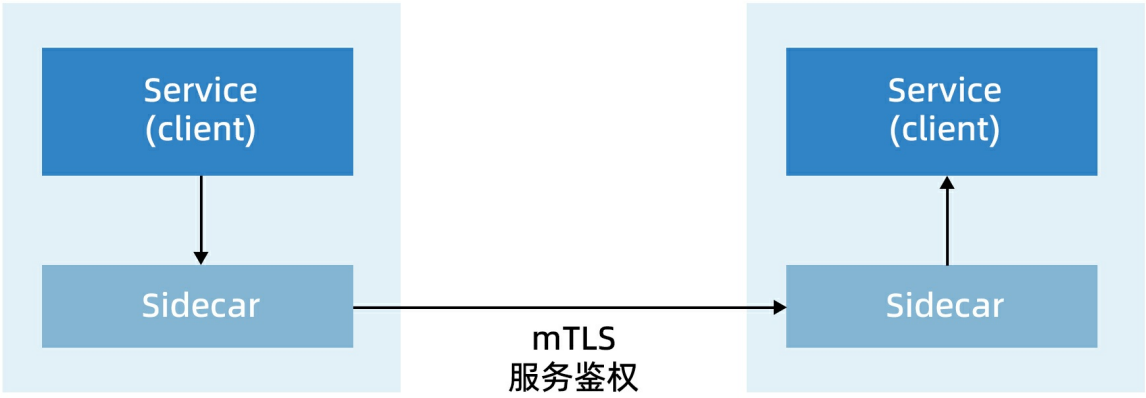
有了 Service Mesh 之后，通过将主体的服务治理能力下沉到基础设施，多语言的支持就会轻松很多。您只需要提供一个非常轻量的 SDK，甚至很多情况都不需要一个单独的 SDK，就可以方便地实现多语言、多协议的统一流量管控、监控等治理需求。



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 Service Mesh，我们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。

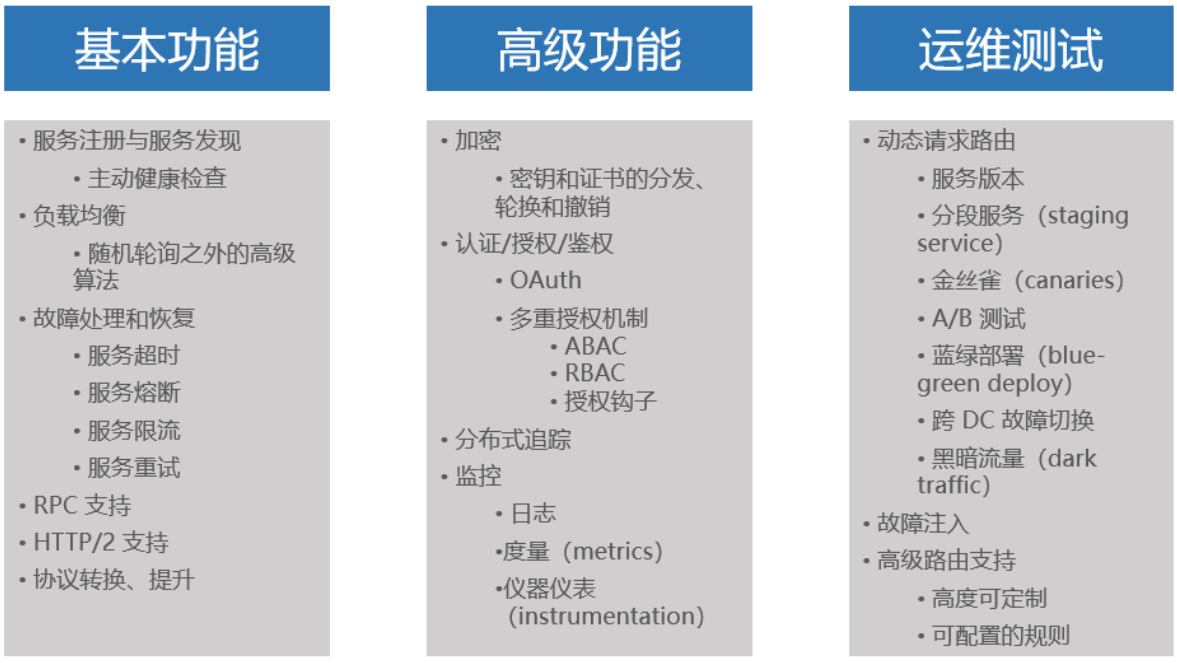


微服务内容多、门槛高

学习微服务的相关能力成本较高，大部分人将微服务的相关能力从 0 到熟练掌握可能需要 3~6 个月。例如 Spring Cloud 的常见子项目 Spring Cloud netflix 下还有 netflix OSS 套件的很多内容，要真正了解 Spring Cloud，则需要把包括 Spring Cloud 子项目在内的所有内容都了解清楚。

服务治理能力缺乏

服务治理的常见功能如下：



以 Spring Cloud 为例，如果要将以上功能一一对应实现，仅仅依靠 Spring Cloud 直接提供的功能远远不够，很多功能都需要用户在 Spring Cloud 的基础上自行开发解决，投入成本巨大。

跨语言

微服务在刚面世时，承诺了一个很重要的特性：不同的微服务可以采用自己最擅长、最适合的编程语言来编写。但这个承诺只实现了一半，因为应用通信通常是基于一个类库或者框架实现，但当您使用具体编程语言开始编码时会发现一个问题：我们到底要为多少种语言提供类库和框架？

要解决这个问题，通常只有以下两种方式：

- 统一编程语言，全公司就用一种编程语言。
- 有多少种编程语言就写多少个类库。

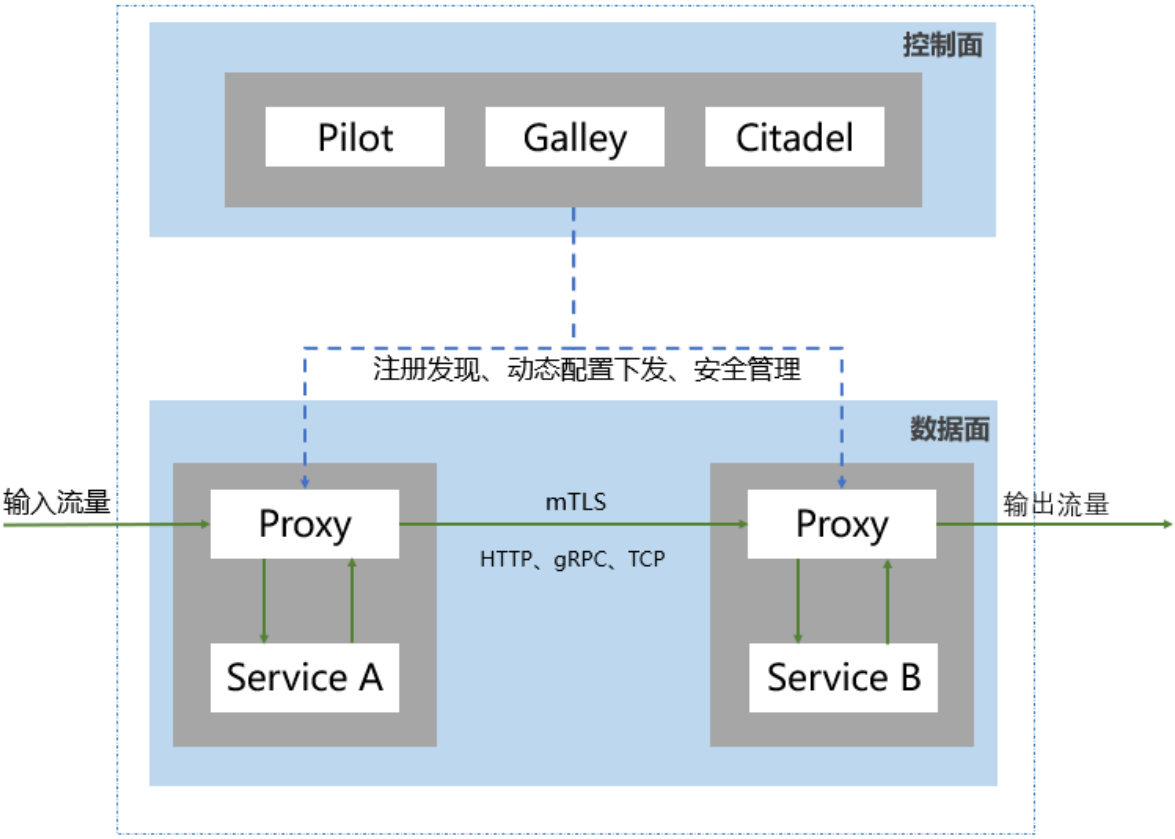
但无论使用哪种方式，缺点都非常明显。

1.3. 发展现状

因为 Service Mesh 具有非常多的优势，所以这两年社区中对 Service Mesh 的关注度越来越高，也涌现出了很多优秀的 Service Mesh 产品，Istio 就是其中一款非常典型的标杆产品。

框架介绍

Istio 来自希腊语，英文是 Sail，翻译为中文是“启航”。Istio 首先是一个服务网络，但是 Istio 又不仅仅是服务网格。在 Linkerd、 Envoy 这样的典型服务网格之上，Istio 提供了一个完整的解决方案，为整个服务网格提供行为洞察和操作控制，以满足微服务应用程序的多样化需求。Istio 架构图如下：



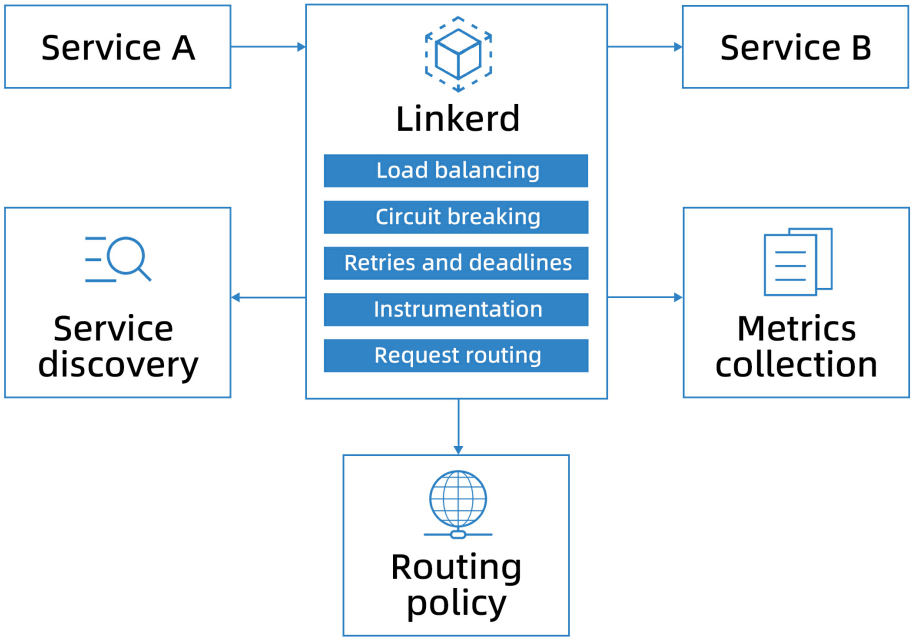
Istio 中的数据面分类

- Envoy（默认）

  基于网络应该对应用程序透明，并且一旦出现网络问题，应该能够快速检测出问题的来源的原则编写。更多信息请参见 Envoy。

- Linkerd（可选）

  linkerd 是一个透明的代理，为现代软件应用程序增加了服务发现、路由、故障处理和可视性。更多信息请参见 Linkerd。



Istio 中的数据面组成

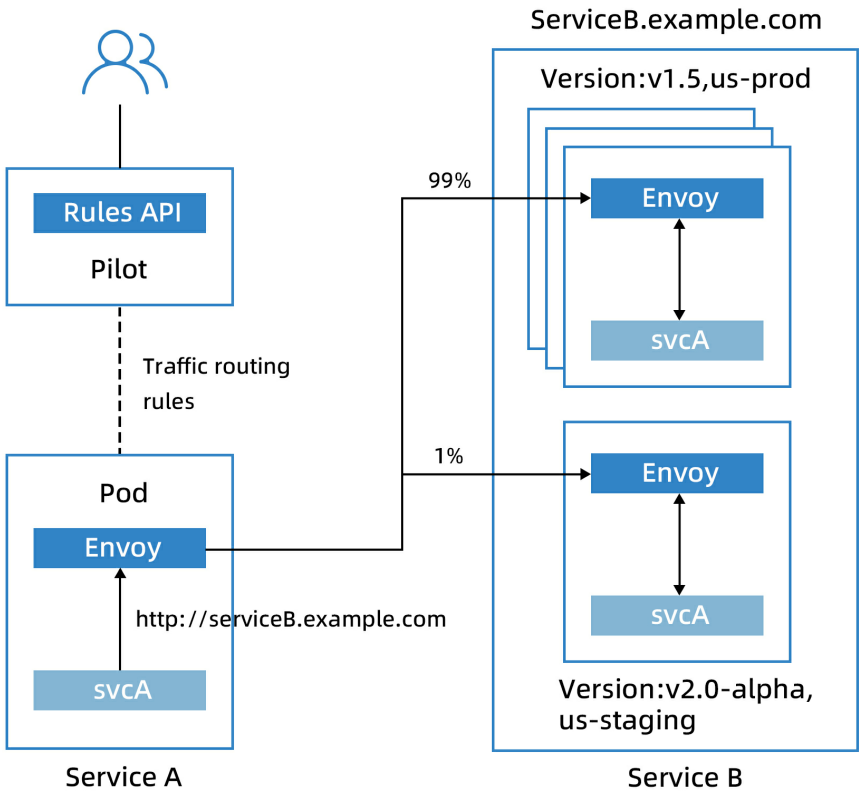
Istio 数据面主要包括三部分：Pilot、Citadel 和 Galley。

Pilot

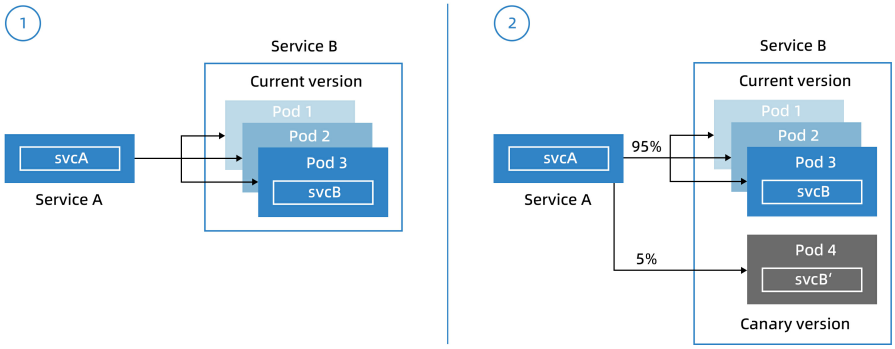
Istio 最核心的功能是流量管理，前面我们看到的数据面板，由 Envoy 组成的服务网格，将整个服务间通讯和入口/出口请求都承载于其上。

使用的流量管理模型，本质上将流量和基础设施扩展解耦，让运维人员通过 Pilot 指定它们希望流量遵循什么规则，而不是哪些特定的 pod 应该接收流量。

首先要介绍一下流量管控的这个整体思路：Pilot 下发规则，数据面实现层来做路由。



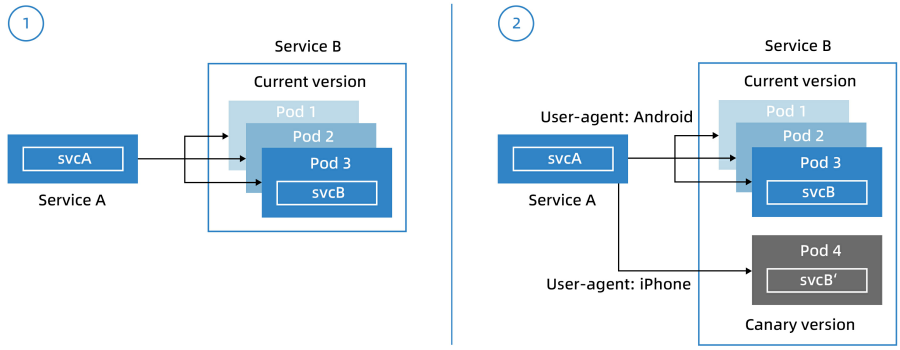
举个例子，假定我们原有服务B，部署在 Pod1/2/3 上，现在我们部署一个新版本在 Pod4 在，希望实现切 5% 的流量到新版本。



Traffic splitting decoupled from infrastructure scaling-proportion of traffic routed to a version is independent of number of instances supporting the version

如果以基础设施为基础实现上述 5% 的流量切分，则需要通过某些手段将流量切 5% 到 Pod4 这个特定的部署单位，实施时就必须和 ServiceB 的具体部署还有 ServiceA 访问 ServiceB 的特定方式紧密联系在一起。比如如果两个服务之间是用 Nginx 做反向代理，则需要增加 Pod4 的 IP 作为 Upstream，并调整 Pod1/2/3/4 的权重以实现流量切分。

如果使用 Istio 的流量管理功能，由于 Envoy 组成的服务网络完全在 Istio 的控制之下,因此要实现上述的流量拆分非常简单. 假定原版本为 1.0，新版本为 2.0，只要通过 Polit 给 Envoy 发送一个规则：2.0 版本 5% 流量，剩下的给 1.0。 这种情况下，我们无需关注 2.0 版本的部署，也无需改动任何技术设置，更不需要在业务代码中为此提供任何配置支持和代码修改。一切由 Pilot 和智能 Envoy 代理搞定。



Content-based traffic steering-The content of a request can be used to determine the destination of a request

Galley

Galley 负责将其余的 Istio 组件与从底层平台获取用户配置的细节隔离开来。它包含用于收集配置的 Kubernetes CRD 侦听器，用于分发配置的 MCP 协议服务器实现，以及用于 Kubernetes API Server 进行预摄取（pre-ingest）验证的验证 Web 挂钩。

Citadel

Citadel 提供强大的服务到服务和终端用户认证，使用交互 TLS，内置身份和凭据管理。它可用于升级服务网格中的未加密流量，并为运维人员提供基于服务身份而不是网络控制实施策略的能力。

1.4. 面临的问题及关键挑战

以 Istio 为例的 Service Mesh 方案目前主要存在以下问题：

不支持非 K8s 体系的应用

Istio 以其前瞻的设计结合云原生的概念，一出现就让人眼前一亮，心之向往。不过在现实中，我们发现目前大部分的公司还没有走向云原生，或者还刚刚在开始探索，所以大量的应用其实还跑在非 K8s 的体系中，比如跑在虚拟机上或者是基于独立的服务注册中心构建微服务体系。 虽然确实有少量新应用已经在基于云原生来构建了，但现实是那些大量的老应用是公司业务的顶梁柱，承载着更大的业务价值，如果不能把它们纳入 Service Mesh 统一管控，那整体 Service Mesh 的业务价值是很有限的。

服务注册发现

传统微服务在 Istio 下的服务注册发现问题：

- SOFA、Dubbo、Spring Cloud 是单进程多服务模型，K8s 是单进程单服务模型（业内共性问题）。
- Istio 的服务注册发现依赖 K8s，导致按照接口调用无法完成寻址。
- Pilot 通过 EDS 下发实现服务发现，在大量服务实例的情况下，EDS 全量下发会严重影响 Pilot 和 Sidecar 的性能和稳定性。

配置下发

XdsAPI 全量下发模式，Sidecar 会接受到自己不需要的配置。同时，一个小小的配置修改，也会触发全量下发。

配置巡检

生产落地场景下，需要保证配置下发的时效性和配置准确性（istio 通过配置中心下发无法保证最终一致性）。

控制面

Istiod 作为新的架构，存在架构演进不稳定、横向扩展存疑等问题。在演进过程中，Istiod 架构已经已经经历三次大规模架构变更。

数据面

- 只支持 Spring Cloud，不支持中国国内常见的 Dubbo。
- 流量劫持方式局限。
- 居于 C++ 语言编写， 扩展难度大。
- 在 Istio 1.17.1 的默认配置中（即带有遥测 v2 的 Istio）， 两个代理在基线数据平面延迟的 90 和 99 分位延迟上增加约 1.7 和 2.7 毫秒，RT 损耗时间略长。详情请参见 Istio 性能和可扩展性。

2.产品架构

2.1. 系统架构

蚂蚁集团服务网格产品全称是 SOFAStack 服务网格，可以基于 Sidecar 的 Service Mesh 微服务实现下列目标：

- SDK 轻量化

  将通用的服务路由、限流、熔断、安全通信、鉴权等能力下沉到 Sidecar，解耦 SDK 和业务应用，减少 SDK 升级成本和对业务应用的打扰。

- 统一服务治理

  完成异构技术栈的统一服务治理、统一监控，降低运维成本，提升应急发现和处置能力。

- 安全通信

  构建低成本、高效率、零信任、个性化的安全通信能力，提升整体安全通信水位。

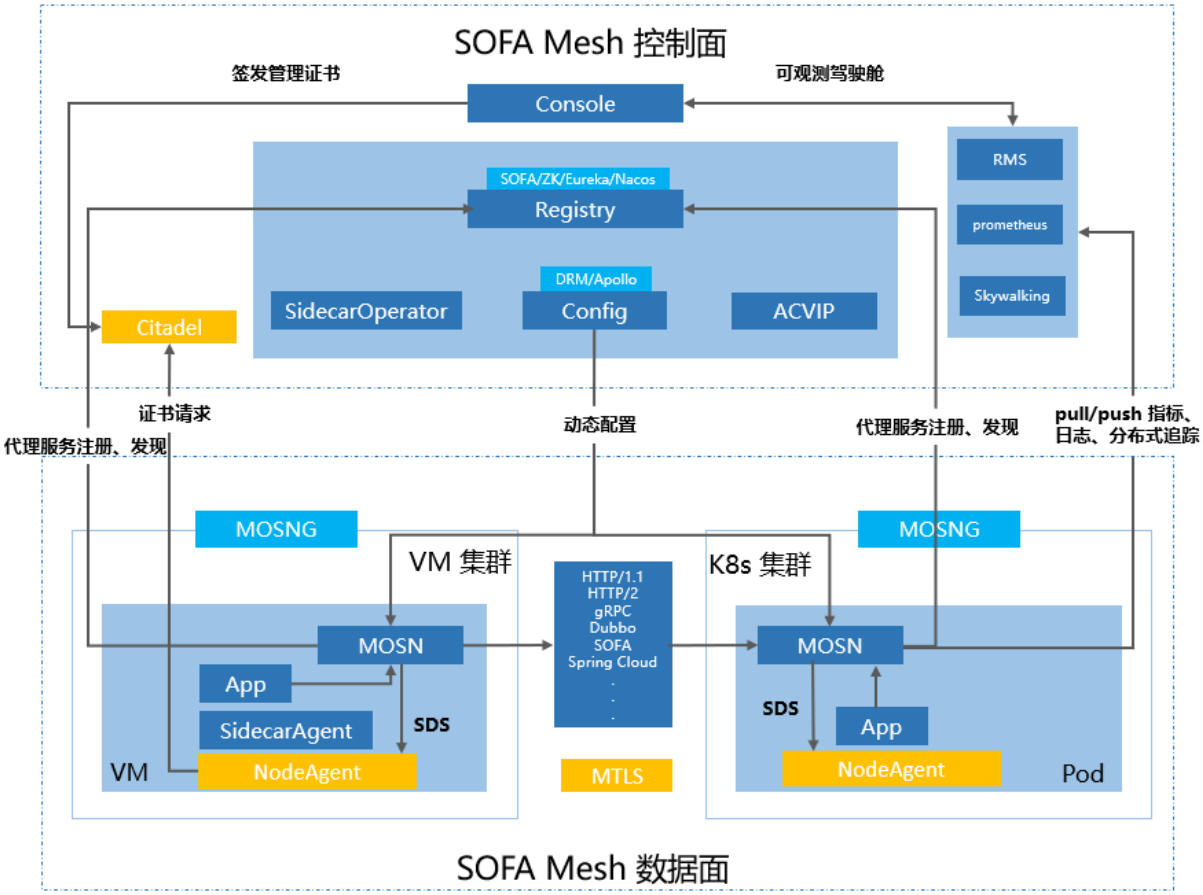
- 异构互联互通

  不同架构之间的应用可以实现互相通信，例如 Dubbo 和 Spring Cloud 服务通信。

在控制面上，我们灵活集成了 SOFARegistry、ZooKeeper、Nacos 注册中心，可观测性上支持对接开源 Skywalking、Zipkin 、Promthueus 等组件。

在数据面上，我们使用了自研的 SOFAMOSN，不仅支持 SOFA 应用，同时也支持 Dubbo 和 Spring Cloud 应用和自建协议系统接入，配置中心可以支持 DRM 和 Apollo的对接。

在部署模式上，我们不仅支持容器 K8s，同时借助 SidecarAgent 支持虚拟机场景。

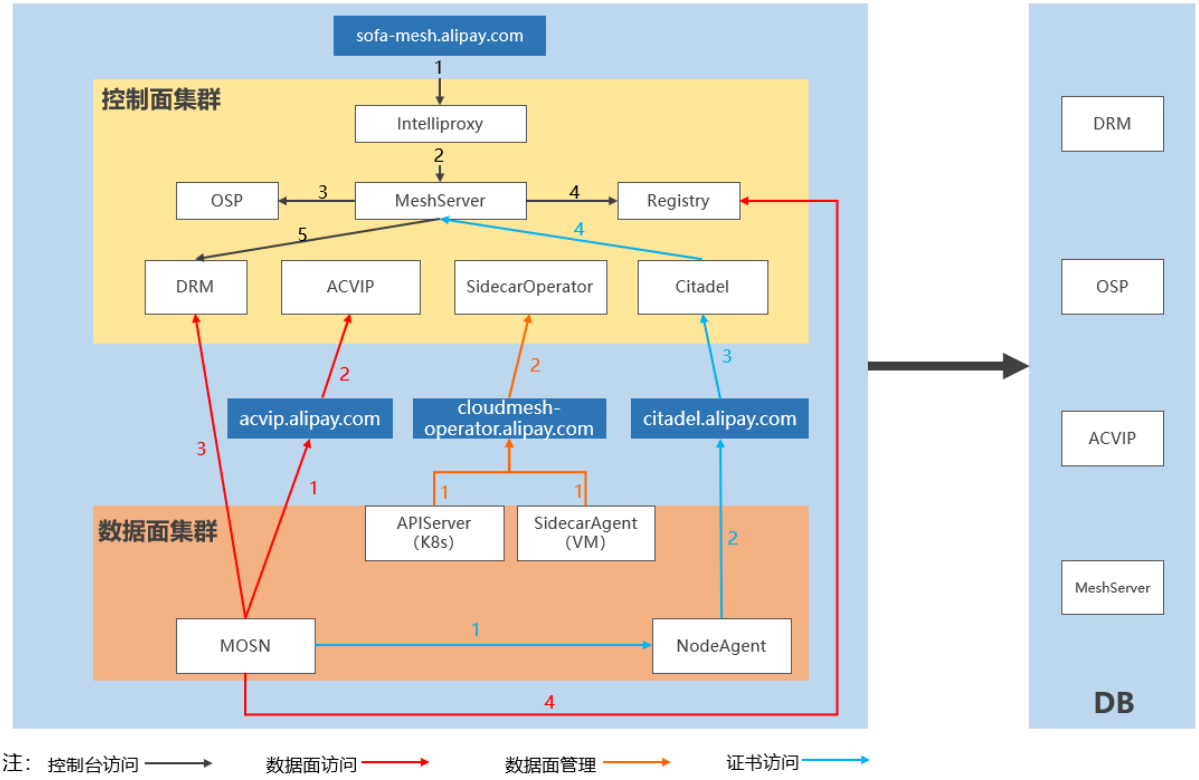


2.2. 技术架构

2.2.1. 单机房架构

架构图

单机房架构图如下：



组件说明

Mesh 组件

组件	组件功能
Intelliproxy	智能服务网关，用来转发路由请求、统一权限和 Cookie 等切面管理。提供统一访问入口。
OSP	运维支持系统，主要提供和管理中台的元数据服务，是被其他应用（例如 tenant、workspace 管理中心、用户等）所依赖的基础服务系统。
ACVIP	虚拟域名寻址服务，提供中间件服务端寻址，且可以进行不同维度的服务端灰度。
MeshServer	服务治理中心管理控制台。
DRM	动态配置中心。
Citadel	安全组件，提供 CA 和 MCP 服务，转发 CSR 请求至 MeshServer（MeshCA）。
Registry	SOFA 服务注册中心。

SIdecarOperator	Sidecar 注入服务，控制容器和虚拟机场景下的 Sidecar 注入。
SidecarAgent	负责接管虚拟机 Sidecar 的生命周期。
NodeAgent	安全组件，作为 SDS Server 发起 CSR 请求。
MOSN	数据面组件，提供流量管理能力，执行控制面下发指令。

基础环境配件

环境需要提供如下域名解析和 DB 集群：

域名	代理组件	用途
sofa-mesh.alipay.com	Intellipoxy	访问控制台，如 Mesh 控制台、ACVIP、OSP控制台等。
acvip.alipay.com	ACVIP	数据面通过 ACVIP 获取控制面真实地址。
citadel.alipay.com	Citadel	安全组件，转发 Sidecar 的 CSR 请求。
cloudmesh-operator.alipay.com	sidecar-operator	Sidecar 注入和升级 Sidecar 操作。
meshserver.alipay.com	Mesh-Server	访问 CA 中心、WebShell 界面操作。

DB	依赖组件	说明
MySQL	MeshServer、ACVIP、OSP、DRM	具备同城双活能力，一主多从，读写分离。 负责维护服务治理、租户管理、虚拟域名的数据管理。

调用关系

控制台

1. 用户通过域名访问到 Intelliproxy 智能网关组件。
2. 智能网关通过数据处理把请求转发到 MeshServer。
3. MeshServer 通过 OSP 判断请求操作的合法性。
4. MeshServer 会从注册中心拉取全量服务数据，用于服务粒度的管理。
5. 开启服务治理等功能时，MeshServer 会将策略下发至配置中心，由配置中心推送至 Sidecar。

数据面

1. Sidecar 进程启动时，会访问 `acvip.alipay.com` 。
2. ACVIP 存储注册配置中心、注册中心地址，并将地址返回给 Sidecar。
3. Sidecar 启动后会主动和配置中心创建并保持长连接，监听 DRM 下发动态配置或服务治理策略。
4. 当应用发起注册、订阅时，Sidecar 将拦截注册、订阅流量，由 Sidecar 代理向注册中心发起注册、订阅请求。

证书管理

1. Sidecar 向 NodeAgent 发起 SDS 请求。
2. NodeAgent 会根据 ConnectID 从 Citadel MCP Server 下发的 Pod 信息中获取应用信息，然后向 Citadel 发起签发证书（CSR）请求（请求上下文中将携带应用信息），获取应用级证书。
3. Citadel 收到请求后会将请求转发给 MeshServer。
4. MeshServer 收到请求后，将根据应用名称查询应用级证书并返回；若不存在，则根据应用名称新签证书。

数据面管理

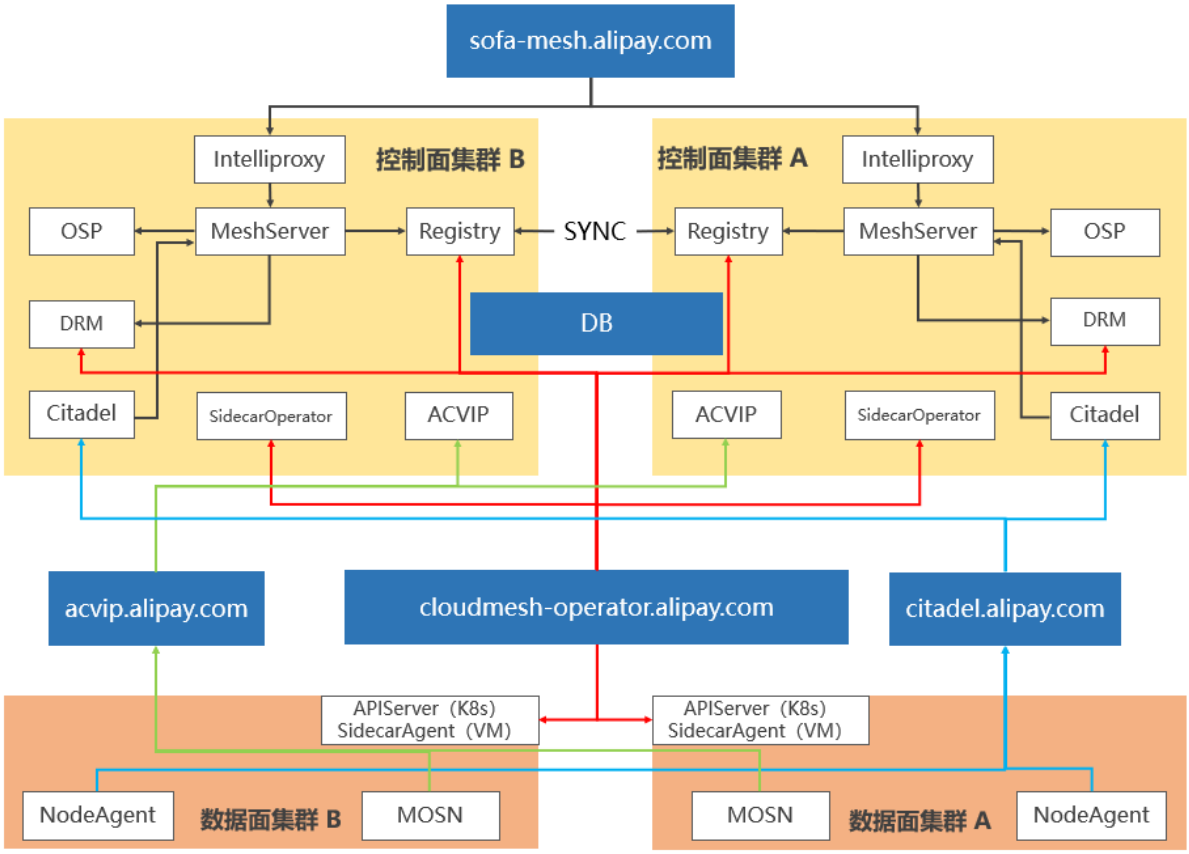
1. 应用注入 Sidecar 时，K8s 中的 APIServer 或者虚拟机的 SidecarAgent 会访问 `cloudmesh-operator.alipay.com` 。
2. Operator 收到请求后，会根据使用场景进行以下操作：
   - 容器场景：
     a. APIServer 将根据 Webhook 配置的 Operator 地址请求 Operator。
     b. Operator 将 Sidecar 对应的 Sidecar 模板信息渲染到 Pod 信息中。
     c. APIServer 收到 Operator 返回的内容后，将启动带有 Sidecar 的 Pod。
   - 虚机场景：
     a. 虚机被注入时，MeshServe 执行命令进入容器，安装 SidecarAgent。
     b. SidecarAgent 启动后，SidecarAgent 将发起向 Operator 发起请求，获取 Sidecar 模板信息， 并按照模板信息启动 Sidecar。

        Sidecar 启动后，SidecarAgent 将负责 Sidecar 的生命周期管理，如重启、下线、保活等。

2.2.2. 同城双活

部署拓扑

Mesh 同城双活部署方案如下图所示：



除了注册中心，其他服务状态都是通过 DB 存储，服务自身无状态。注册中心支持多机房同步方案，保证了自身的高可用。

组件说明

Mesh 组件

组件	组件功能
Intelliproxy	智能服务网关，用来转发路由请求、统一权限和 Cookie 等切面管理。提供统一访问入口。
OSP	运维支持系统，主要提供和管理中台的元数据服务，是被其他应用（例如 tenant、workspace 管理中心、用户等）所依赖的基础服务系统。
ACVIP	虚拟域名寻址服务，提供中间件服务端寻址，且可以进行不同维度的服务端灰度。
MeshServer	服务治理中心管理控制台。
DRM	动态配置中心。

Citadel	安全组件，提供 CA 和 MCP 服务，转发 CSR 请求至 MeshServer（MeshCA）。
Registry	SOFA 服务注册中心。
SidecarOperator	Sidecar 注入服务，控制容器和虚拟机下的 Sidecar 注入。
SidecarAgent	负责接管虚拟机 Sidecar 的生命周期。
NodeAgent	安全组件，作为 SDS Server 发起 CSR 请求。
MOSN	数据面组件，提供流量管理能力，执行控制面下发指令。

基础环境配件

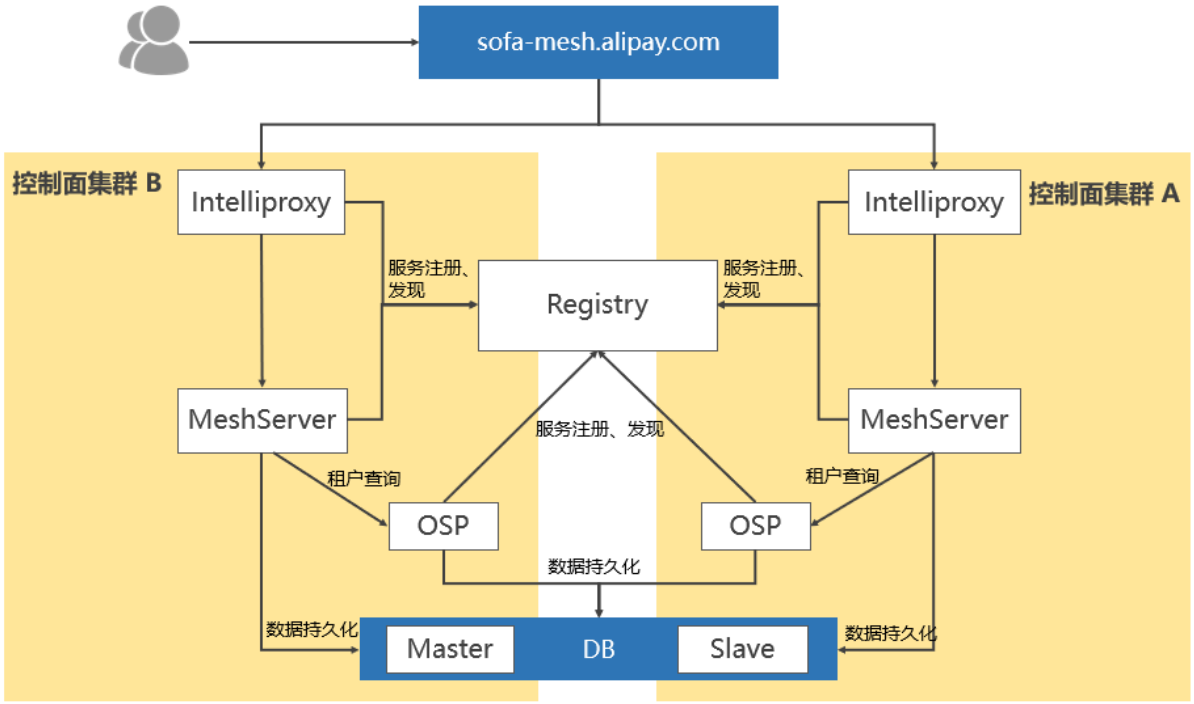
环境需要提供如下域名解析和 DB 集群：

域名	代理组件	用途
sofa-mesh.alipay.com	Intellipoxy	访问控制台，如 Mesh 控制台、ACVIP、OSP控制台等。
acvip.alipay.com	ACVIP	数据面通过 ACVIP 获取控制面真实地址。
citadel.alipay.com	Citadel	安全组件，转发 Sidecar 的 CSR 请求。
cloudmesh-operator.alipay.com	SidecarOperator	处理数据面注入和升级 Sidecar 操作。
meshserver.alipay.com	MeshServer	访问 CA 中心、WebShell 界面操作。

DB	依赖组件	说明
MySQL	MeshServer、ACVIP、OSP、DRM	具备同城双活能力，一主多从，读写分离。 负责维护服务治理、租户管理、虚拟域名的数据管理。

双活策略

控制台访问

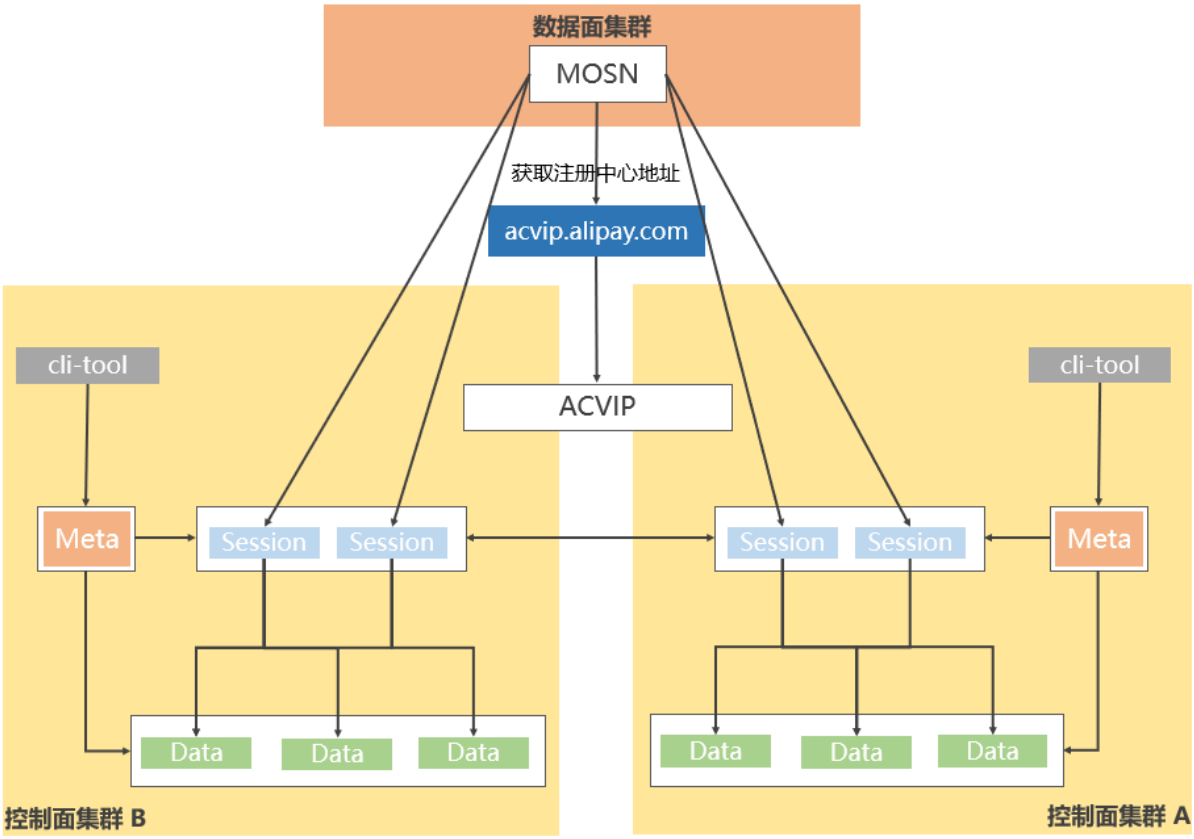


组件说明如下：

组件名称	方案	影响范围
Intelliproxy	无状态，双机房部署，LB 负载	Mesh 控制台访问
OSP	多机房共库，双机房部署	租户授权、控制台访问、控制面组件访问
MeshServer	多机房共库，双机房部署	Mesh控制台、Mesh API
Registry	多机房同步，双机房部署	服务注册、服务订阅

注册中心

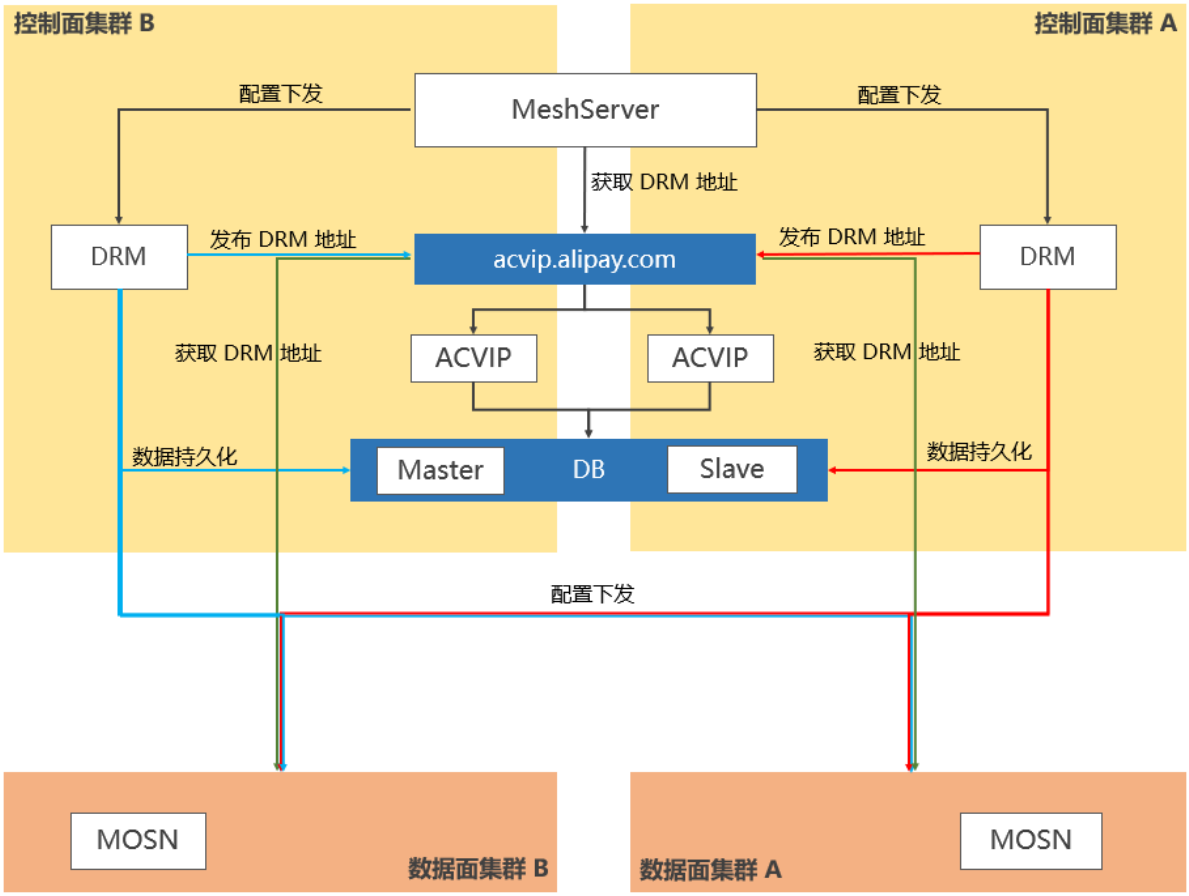
目前 Mesh 具备集成三方注册中心能力，以 SOFA Registry 为例，架构如图所示：



组件说明如下：

组件名称	方案	影响范围
Data	集群部署了多个副本	订阅关系丢失
Session	集群部署了多个副本	客户端访问报错
Meta	集群部署了多个副本	当前集群无法扩容
ACVIP	多机房共库，双机房部署，LB 负载	客户端无法获取的控制面地址

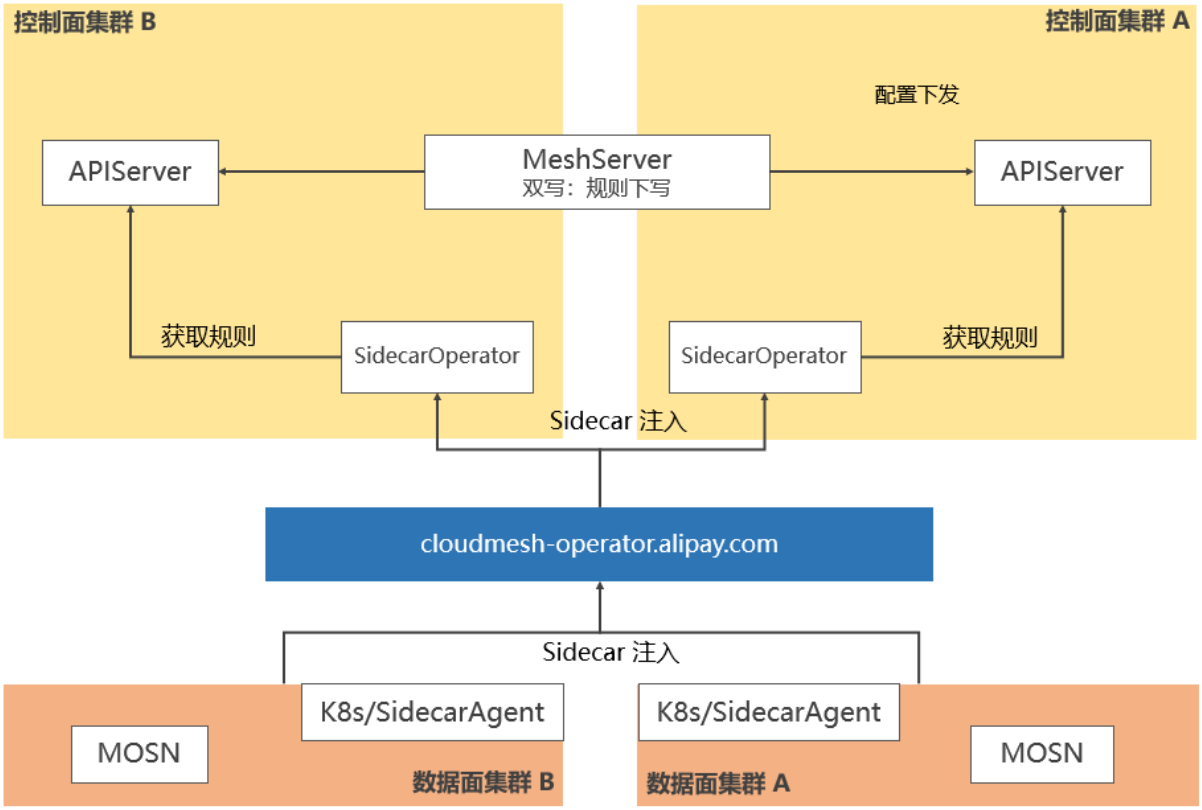
配置中心



组件说明如下：

组件名称	方案	影响范围
MeshServer	多机房共库，双机房部署，LB 负载	Mesh 控制台、Mesh API
DRM	多机房共库，双机房部署、ACVIP 客户端提供就近路由逻辑	动态配置无法下发（服务治理、 声明式服务发现等依赖动态配置的相关功能将不可用）
ACVIP	多机房共库，双机房部署，LB 负载	客户端无法获取的控制面地址

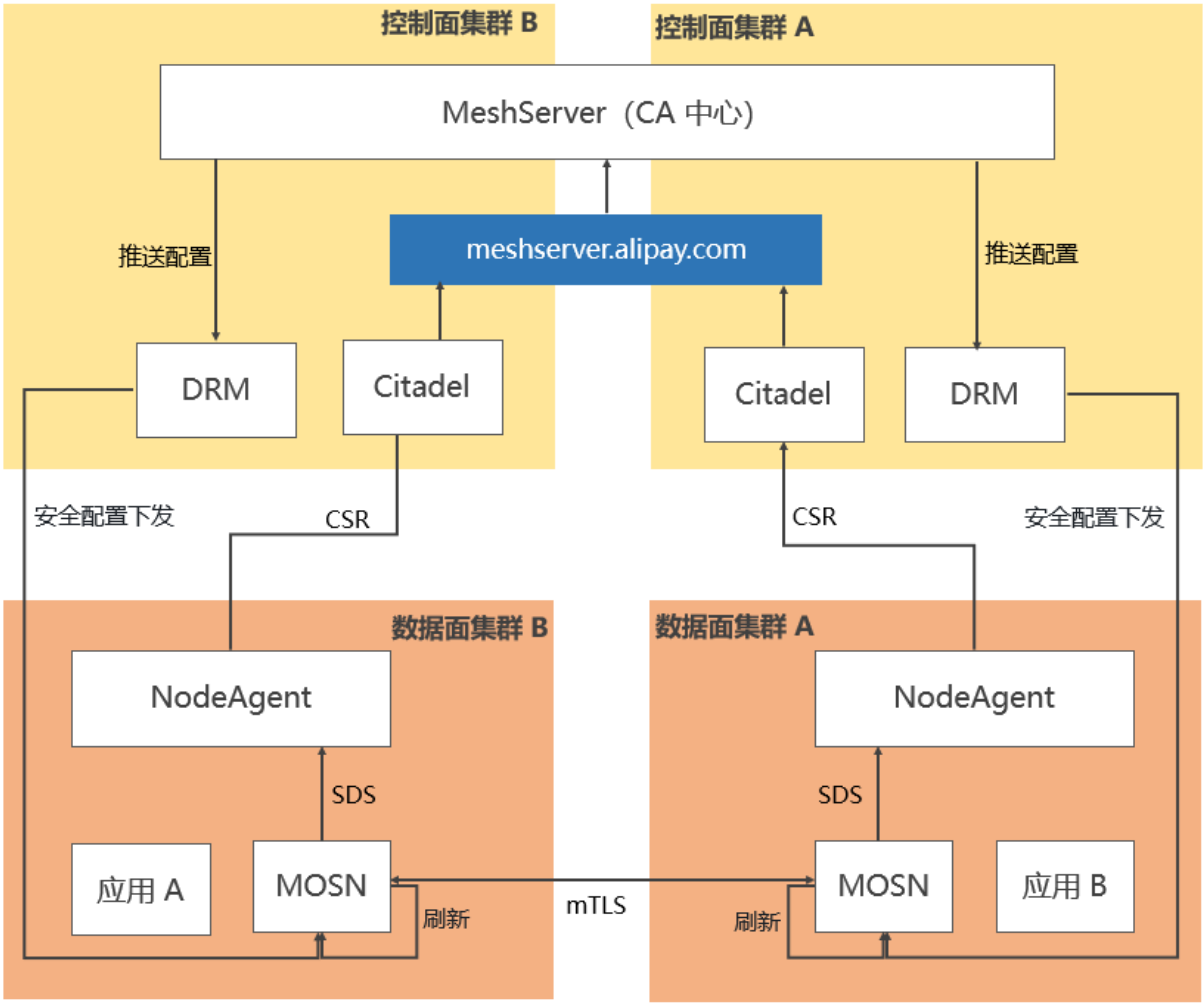
Sidecar 管理



组件说明如下：

组件名称	方案	影响范围
MeshServer	多机房共库， 双机房部署，LB 负载	Mesh 控制台、Mesh API
SidecarOperator	无状态，双机房部署，LB 负载	Sidecar 注入失败
APIServer	K8s 标准组件，单机房部署	整个控制面不可用

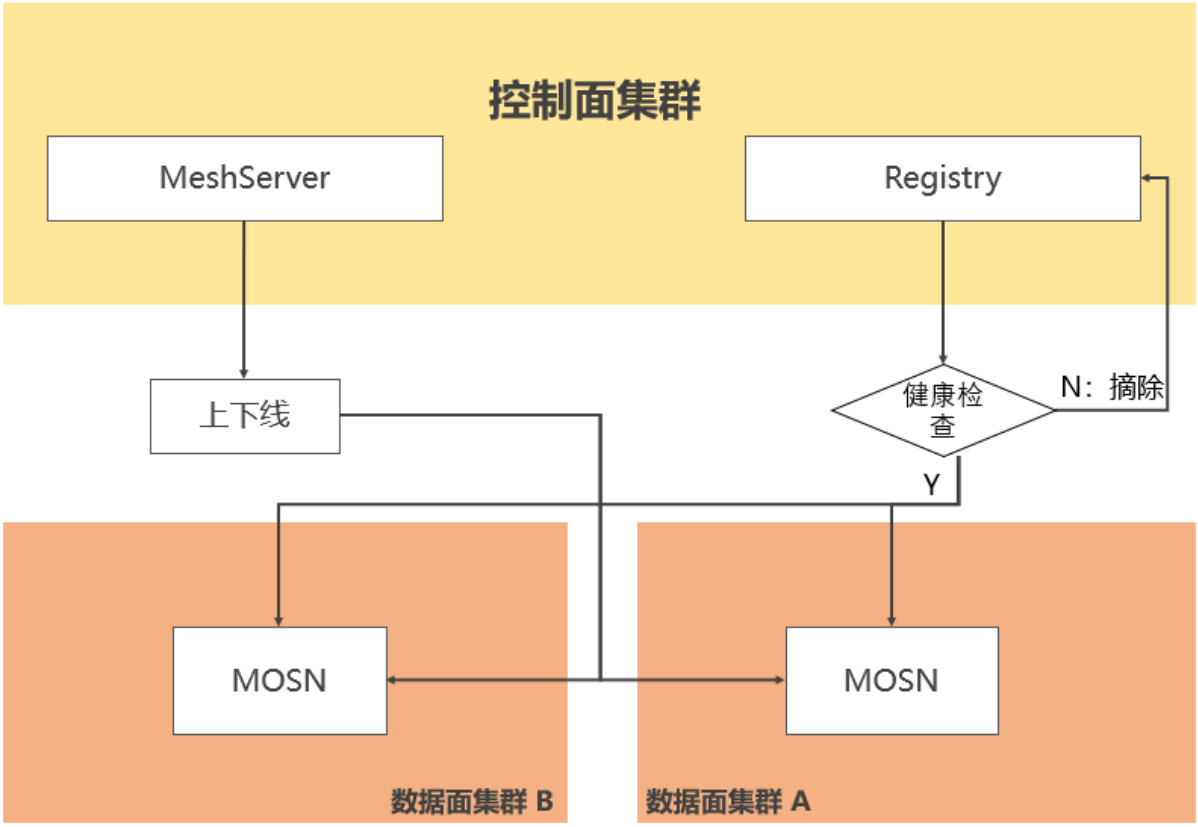
安全管理



组件说明如下：

组件名称	方案	影响范围
MeshServer	多机房共库，双机房部署	Mesh 控制台、Mesh API、 证书管理（签发、轮转、吊销等）
CitadelAgent	Citadel Agent 无状态	新签证书请求失败
Citadel	Citadel 无状态，LB负载均衡	新签证书请求失败
DRM	多机房共库，双机房部署、ACVPI 负载均衡	动态配置无法下发（服务治理、 声明式服务发现等依赖动态配置的相关功能将不可用）

数据面



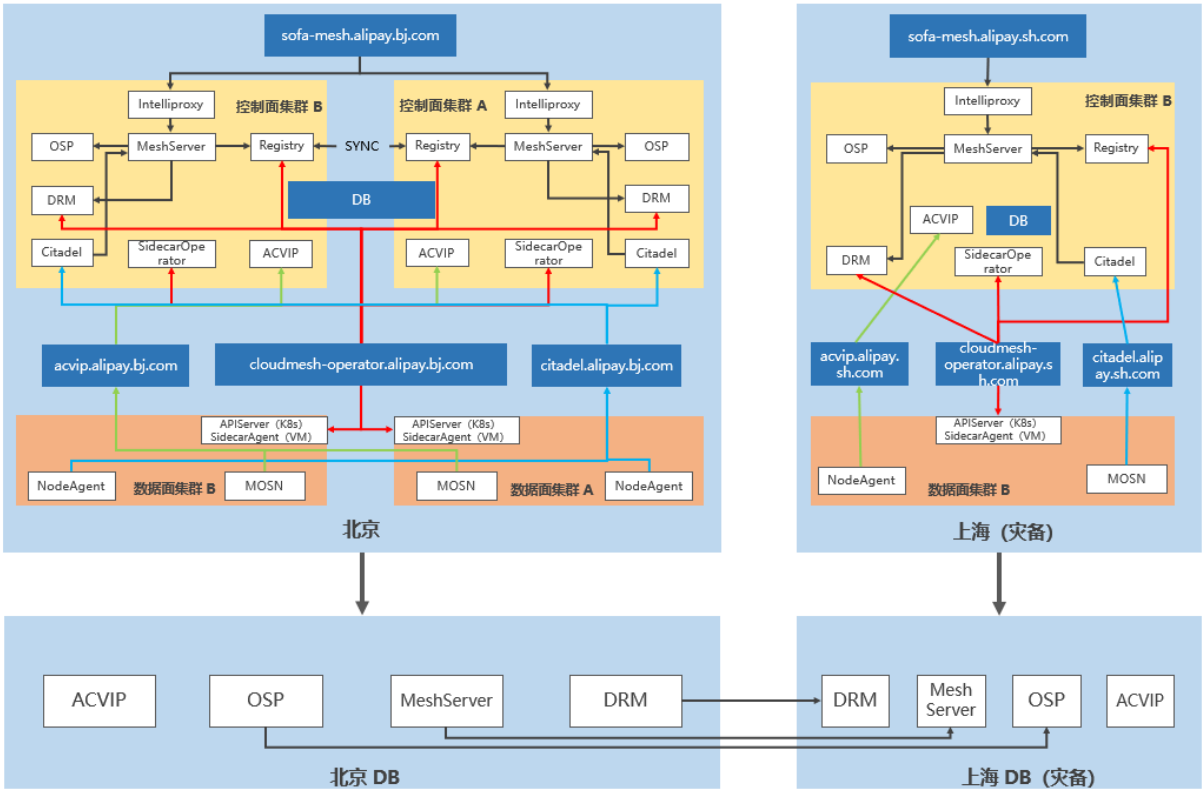
- MOSN 高可用机制
  - 进程保活：容器内置 supervisor 监听 MOSN 进程，进程异常退出时，将自动恢复 MOSN进程。
  - 注册中心监测： MOSN 宕机时，注册中心将秒级自动摘除节点，保障请求不会请求到异常节点。
- MOSN 不可恢复异常

  通过控制台或 Mesh API 对异常节点 MOSN 执行下线操作，即可恢复至未接入状态。

2.2.3. 两地三中心

部署拓扑

两地三中心部署架构基于同城双活部署架构实现，以在北京和上海部署为例，架构图如下：



域名规划

域名	代理组件	用途
sofa-mesh.alipay.bj.com	Intellipoxy	访问控制台，如 Mesh 控制台、ACVIP、OSP 控制台等。
sofa-mesh.alipay.sh.com		
acvip.alipay.bj.com	ACVIP	数据面通过 ACVIP 获取控制面真实地址。
acvip.alipay.sh.com		
citadel.alipay.bj.com	Citadel	安全组件，转发 Sidecar 的 CSR 请求。
citadel.alipay.sh.com		
cloudmesh-operator.alipay.bjcom	SidecarOperator	处理数据面注入和升级 Sidecar 操作。
cloudmesh-operator.alipay.sh.com		

meshserver.alipay.bj.com	MeshServer	访问 CA 中心、WebShell 界面操作。
meshserver.alipay.shcom		

DB 规划

DB	依赖组件	说明
MySQL 一主多从，异地备份 北京主机房、上海为备份机房	MeshServer	负责维护服务治理
	OSP	租户管理
	DRM	保管动态配置
MySQL 一主多从	ACVIP	虚拟域名的数据管理

灾备说明

北京发生集体断电，需要切换到备份机房。

DNS 切换

域名	切换状态	代理组件
sofa-mesh.alipay.bj.com	sofa-mesh.alipay.sh.com	Intellipoxy
acvip.alipay.bj.com	acvip.alipay.sh.com	ACVIP
citadel.alipay.bj.com	citadel.alipay.sh.com	Citadel
cloudmesh-operator.alipay.bjcom	operator.alipay.sh.com	SidecarOperator
meshserver.alipay.bj.com	meshserver.alipay.shcom	MeshServer

DB 切换

DB	切换状态	依赖组件

北京主机房、上海为备份机房	上海主机房、北京为备份机房	MeshServer
		OSP
		DRM

3.性能指标

概述

Service Mesh 的压测主要围绕以下三个维度进行：

- 损耗：测试固定规格下的资源损耗情况。

  MOSN 分别在 1000、2000、4000 QPS ，双向1 KB 报文压力下，RT 损耗在 0.3~0.6 ms 之间，CPU 在 0.5c~0.6c 左右，内存在 170 MiB 以内。

- 阈值：测试固定规格下的极限表现。

  MOSN 在 1c1g 规格下，根据协议不同，TPS 峰值在 2000~2500 之间；2c2g 规格下，TPS 峰值在 3000~3500 之间。

- 最小规格：测试最小资源消耗。

  MOSN 在 0.1c0.1g 规格下可以稳定运行，保证正常的服务通信。

压测结论

损耗

- RT损耗：整体延迟损耗在 0.6 ms 内。
- CPU 损耗：
  - 1000 TPS：CPU 损耗 0.6c 左右，MOSN Sidecar 规格在 1c1g 即可支撑。
  - 2000 TPS：CPU 损耗 1c 以上，MOSN Sidecar 规格需大于 1c1g。
  - 4000 TPS：CPU 损耗 2c 以上，MOSN Sidecar 规格需大于 2c2g。
- 内存损耗：不同 TPS 损耗差别不大，最大损耗 170 MiB 以内。

协议	TPS	并发线程数	延迟损耗（ms）	实际 CPU 损耗（c）	内存损耗（MiB）
Spring Cloud	1000	10	0.38	59.49%	162.24
	2000	20	0.47	122.42%	164.72
	4000	25	0.57	240.46%	166.94
Dubbo	1000	10	0.53	62%	156.2
	2000	20	0.59	126.45%	158.8
	4000	25	0.51	216.19%	160.36

SOFA	1000	10	0.38	52.92%	127.69
	2000	20	0.44	104%	103.4
	4000	25	0.48	206.96%	130.6
HTTP1	1000	10	0.45	55.86%	130.99
	2000	20	0.36	106%	133.7
	4000	25	0.58	188.03%	135.69

阈值

- MOSN Sidecar 规格为 1c1g，在 10 并发线程、双向 1 KB 报文压力下，MOSN 运行稳定，请求成功率 100%，CPU 使用率接近 100%，TPS 最高峰值 2000 左右，RT 损耗单边在 1ms 左右，详细如下表所示：

协议	MOSN 规格（limit 值）	并发数	TPS		RT		差值		
			no-Sidecar	Sidecar	no-Sidecar	Sidecar	TPS	RT	RT/2
Spring Cloud	1c1g	10	2461	1589	4.03	6.26	-872	2.23	1.115
Dubbo	1c1g	10	2637	1679	3.77	5.92	-958	2.15	1.075
SOFA	1c1g	10	2472	1754	4.02	5.67	-718	1.65	0.825
HTTP1	1c1g	10	1976	1521	5.4	6.54	-455	1.14	0.57

- MOSN Sidecar 规格为 2c2g，在 20 并发线程、双向 1 KB 报文压力下，MOSN 运行稳定，请求成功率 100%，CPU 使用率接近 200%，TPS 最高峰值 3000 左右，RT 损耗单边在 1ms 以内，详细如下表所示：

协议	MOSN 规格		TPS	RT	差值

	（limit 值）	并发数	no-Sidecar	Sidecar	no-Sidecar	Sidecar	TPS	RT	RT/2
Spring Cloud	2c2g	20	4820	3326	4.15	5.98	-1494	1.83	0.915
Dubbo	2c2g	20	5198	3560	3.82	5.59	-1638	1.77	0.885
SOFA	2c2g	20	4759	3575	4.18	5.56	-1184	1.38	0.69
HTTP1	2c2g	20	3938	3180	5.05	6.25	-758	1.2	0.6

最小运行规格

MOSN 最小运行规格为 0.1c0.1g，在此规格下应用可注入成功且能正常调用。

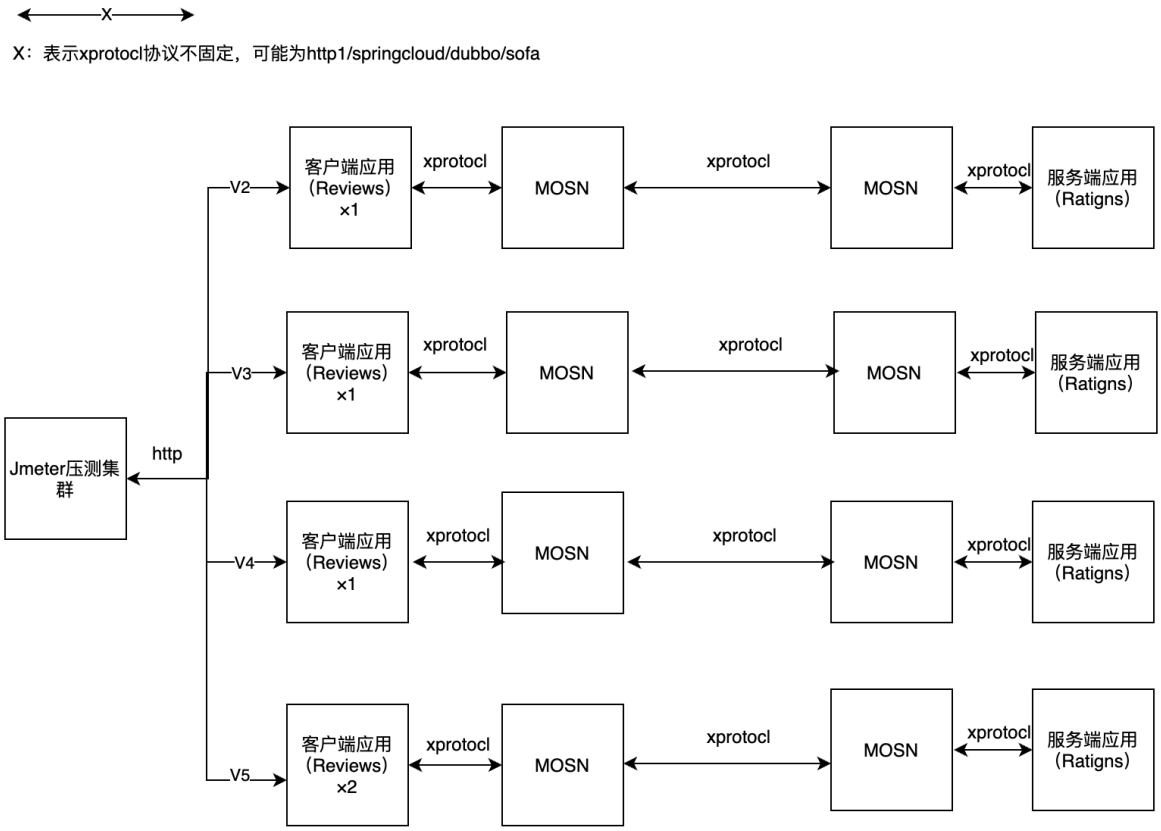
压测过程

压测方案

分析数据面性能表现有很多影响因素，如：QPS（压力）、请求和响应报文大小、协议、CPU、客户端连接数、代理工作线程、代理过滤器数量（开启的功能数量）、网络延迟等。压测过程中，将控制这些变量进行压测，并进行数据统计。

- 压测的参数设置如下：
  - 压力机：jmeter（8c16g）
  - 压力机参数：
    - 吞吐量：1000、2000、4000 TPS
    - 线程数：10、20、25
    - 报文：1 KB（双向）
  - 压测时长：5分钟
  - 业务规格：4c8g，jvm：-Xms5734m -Xmx5734m
  - MOSN规格： 1c1g、2c2g、3c3g
  - 压测协议： Spring Cloud、Dubbo、SOFA、SpringBoot（HTTP1）
  - 压测地址：
    - Spring Cloud: ${reviewsv2 Service ip}:9080/send_1k
    - Dubbo: ${reviewsv3 Service ip}:9080/send_1k
    - SOFA: ${reviewsv4 Service ip}:9080/send_1k
    - HTTP1: ${reviews http1 Service ip}:9080/send_1k

- 压测链路：Mesh POC Demo（Reviews<->Ratigns 链路），通过 Service 负载均衡访问。其中 HTTP1 链路 Reviews 配置 2 个实例（原因：短连接导致施压不上，未保存），Ratigns服务端保持 1 个实例，其他协议链路 1：1。
- 施压目标应用：服务端应用 Ratigns。
- 压测结果统计：MOSN CPU 和内存损耗通过 Prometheus 统计，延迟损耗通过 jmeter 聚合报告统计。



压测结果

阈值探测

报文 1 KB，请求成功率 100% 的条件下，阈值探测的结果如下：

协议	MOSN 规格（limit 值）	并发数	TPS		RT		差值		
			no-Sidecar	Sidecar	no-Sidecar	Sidecar	TPS	RT	RT/2
Spring Cloud	1c1g	10	2461	1589	4.03	6.26	-872	2.23	1.115
	2c2g	20	4820	3326	4.15	5.98	-1494	1.83	0.915
	1c1g	10	2637	1679	3.77	5.92	-958	2.15	1.075

Dubbo	2c2g	20	5198	3560	3.82	5.59	-1638	1.77	0.885
SOFA	1c1g	10	2472	1754	4.02	5.67	-718	1.65	0.825
	2c2g	20	4759	3575	4.18	5.56	-1184	1.38	0.69
HTTP1	1c1g	10	1976	1521	5.4	6.54	-455	1.14	0.57
	2c2g	20	3938	3180	5.05	6.25	-758	1.2	0.6

损耗测试

报文 1 KB，请求成功率 100% 的条件下，损耗测试的结果如下：

协议	MOSN 规格（limit 值）	TPS	线程数	延迟损耗（ms）	实际 CPU 损耗（c）	内存损耗（MiB）
Spring Cloud	3c3g	1000	10	0.38	59.49%	162.24
		2000	20	0.47	122.42%	164.72
		4000	25	0.57	240.46%	166.94
Dubbo	3c3g	1000	10	0.53	62%	156.2
		2000	20	0.59	126.45%	158.8
		4000	25	0.51	216.19%	160.36
SOFA	3c3g	1000	10	0.38	52.92%	127.69
		2000	20	0.44	104%	103.4
		4000	25	0.48	206.96%	130.6
		1000	10	0.45	55.86%	130.99

HTTP1	3c3g	2000	20	0.36	106%	133.7
		4000	25	0.58	188.03%	135.69

jemter 执行汇总结果如下：

- 未注入 MOSN 基础数据

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**16885423**	**0**	**0.00%**	**4.46**	**3**	**412**	**5.00**	**5.00**	**6.00**	**8.00**	**6118.17**	**721.22**	**776.72**
dubbo-100(preheat)	11985	0	0.00%	4.27	3	224	4.00	5.00	6.00	8.00	99.89	11.78	12.68
dubbo-1000	597559	0	0.00%	3.84	3	111	4.00	4.00	4.00	7.00	995.93	117.40	126.44
dubbo-2000	1192427	0	0.00%	4.05	3	205	4.00	4.00	5.00	7.00	1987.38	234.27	252.30
dubbo-4000	2445322	0	0.00%	4.07	3	288	4.00	4.00	5.00	8.00	4075.50	480.42	517.40
http1-100(preheat)	12000	0	0.00%	5.32	4	98	5.00	6.00	6.00	8.00	100.00	11.79	12.79
http1-1000	596467	0	0.00%	5.15	3	246	5.00	5.00	6.00	8.00	994.11	117.19	127.18
http1-2000	1189464	0	0.00%	5.34	3	205	5.00	7.00	8.00	8.00	1982.44	233.69	253.61
http1-4000	2425911	0	0.00%	5.15	3	290	5.00	5.00	6.00	8.00	4043.16	476.61	517.24
sofa-100(preheat)	11980	0	0.00%	4.64	3	289	4.00	5.00	6.00	8.00	99.84	11.77	12.67
sofa-1000	595796	0	0.00%	4.11	3	244	4.00	4.00	5.00	7.00	992.99	117.06	126.06
sofa-2000	1185378	0	0.00%	4.33	3	205	4.00	5.00	5.00	12.00	1975.62	232.89	250.81
sofa-4000	2402347	0	0.00%	4.41	3	286	4.00	4.00	5.00	16.00	4003.90	471.98	508.31
springcloud-100(preheat)	11950	0	0.00%	5.02	3	412	4.00	6.00	8.00	9.00	99.73	11.76	12.56
springcloud-1000	596975	0	0.00%	4.08	3	190	4.00	4.00	5.00	7.00	994.96	117.29	125.34
springcloud-2000	1187935	0	0.00%	4.23	3	206	4.00	5.00	5.00	8.00	1979.88	233.39	249.42
springcloud-4000	2421927	0	0.00%	4.30	3	287	4.00	5.00	5.00	12.00	4036.52	475.83	508.51

- 注入 MOSN，规格为 3c3g

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**8308961**	**0**	**0.00%**	**5.46**	**4**	**4737**	**6.00**	**7.00**	**8.00**	**14.00**	**2715.45**	**321.41**	**344.73**
dubbo-100(preheat)	12004	0	0.00%	4.61	4	37	5.00	5.00	5.00	8.00	100.04	11.79	12.70
dubbo-1000	298704	0	0.00%	4.91	4	216	5.00	5.00	6.00	8.00	995.69	117.37	126.41
dubbo-2000	595169	0	0.00%	5.24	4	156	5.00	6.00	7.00	9.00	1983.87	233.86	251.86
dubbo-4000	1195393	0	0.00%	5.09	4	137	5.00	6.00	8.00	12.00	3984.58	469.71	505.85
http1-100(preheat)	11994	0	0.00%	7.08	5	297	7.00	8.00	9.00	11.00	99.95	11.98	12.79
http1-1000	297573	0	0.00%	6.05	4	159	6.00	6.00	7.00	10.00	991.89	118.86	126.89
http1-2000	595064	0	0.00%	6.06	4	139	6.00	7.00	7.00	10.00	1983.55	237.70	253.75
http1-4000	1153195	0	0.00%	6.31	4	4737	6.00	7.00	8.00	14.00	3843.94	460.64	491.75
sofa-100(preheat)	12005	0	0.00%	4.74	4	40	5.00	5.00	5.00	8.00	100.03	11.79	12.70
sofa-1000	297937	0	0.00%	4.88	4	192	5.00	5.00	5.00	8.00	993.11	117.07	126.08
sofa-2000	592102	0	0.00%	5.21	4	135	5.00	6.00	6.00	9.00	1973.67	232.66	250.56
sofa-4000	1170380	0	0.00%	5.38	4	231	5.00	6.00	8.00	21.00	3901.23	459.88	495.27
springcloud-100(preheat)	11985	0	0.00%	5.04	4	59	5.00	5.00	6.00	8.00	100.03	11.79	12.60
springcloud-1000	298928	0	0.00%	4.84	4	79	5.00	5.00	5.00	8.00	996.42	117.46	125.53
springcloud-2000	594044	0	0.00%	5.17	4	215	5.00	6.00	6.00	11.00	1980.71	233.49	249.52
springcloud-4000	1172484	0	0.00%	5.45	4	161	5.00	6.00	8.00	17.00	3908.21	460.71	492.34

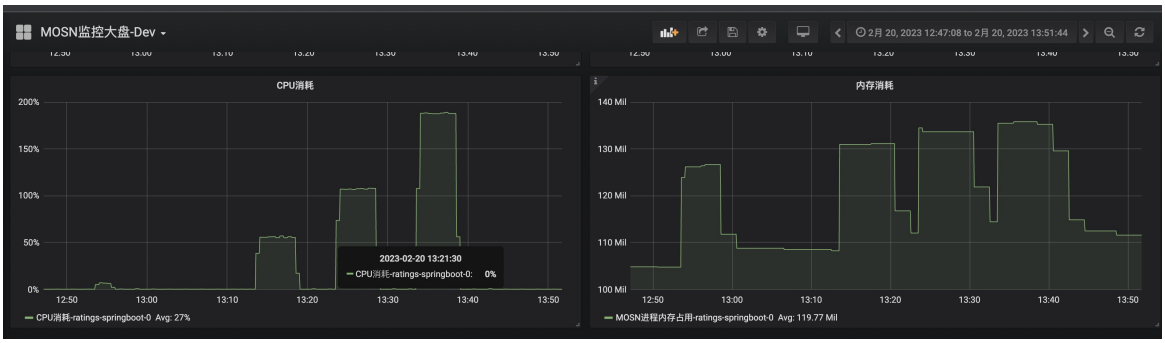
注入 MOSN 后，MOSN 进程消耗的 CPU 和内存情况如下：

- Spring Cloud 协议

- Dubbo 协议

- SOFA 协议

- HTTP1 协议



最小资源消耗测试

在 MOSN 最小运行规格（0.1c0.1g）下，应用可注入成功且能正常调用。测试结果如下：

- Spring Cloud

```
[root@ts-sofaperformance-loadrunner-0 /root]
# curl 172.16.95.30:9080/send_1k -v
* About to connect() to 172.16.95.30 port 9080 (#0)
*   Trying 172.16.95.30...
* Connected to 172.16.95.30 (172.16.95.30) port 9080 (#0)
> GET /send_1k HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.16.95.30:9080
> Accept: */*
>
< HTTP/1.1 200
< Content-Length: 0
< Date: Fri, 10 Feb 2023 06:29:07 GMT
<
* Connection #0 to host 172.16.95.30 left intact
```

- Dubbo

```
[root@ts-sofaperformance-loadrunner-0 /root]
# curl 172.16.57.109:9080/send_1k -v
* About to connect() to 172.16.57.109 port 9080 (#0)
*   Trying 172.16.57.109...
* Connected to 172.16.57.109 (172.16.57.109) port 9080 (#0)
> GET /send_1k HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.16.57.109:9080
> Accept: */*
>
< HTTP/1.1 200
< Content-Length: 0
< Date: Fri, 10 Feb 2023 06:29:13 GMT
<
* Connection #0 to host 172.16.57.109 left intact

[root@ts-sofaperformance-loadrunner-0 /root]
```

- SOFA

```
[root@ts-sofaperformance-loadrunner-0 /root]
# curl 172.16.107.23:9080/send_1k -v
* About to connect() to 172.16.107.23 port 9080 (#0)
*   Trying 172.16.107.23...
* Connected to 172.16.107.23 (172.16.107.23) port 9080 (#0)
> GET /send_1k HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.16.107.23:9080
> Accept: */*
>
< HTTP/1.1 200
< Content-Length: 0
< Date: Fri, 10 Feb 2023 06:29:26 GMT
<
* Connection #0 to host 172.16.107.23 left intact

[root@ts-sofaperformance-loadrunner-0 /root]
```

- HTTP1

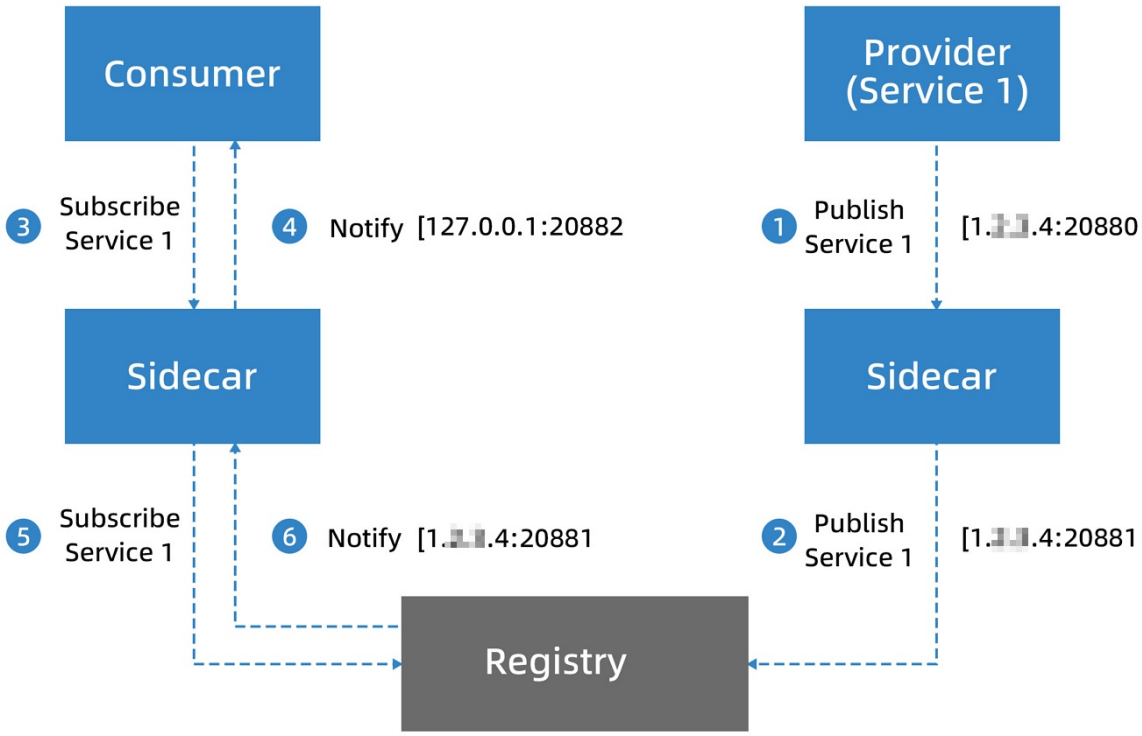
```
[root@ts-sofaperformance-loadrunner-0 /root]
# curl 172.16.241.138:9080/send_1k -v
* About to connect() to 172.16.241.138 port 9080 (#0)
*   Trying 172.16.241.138...
* Connected to 172.16.241.138 (172.16.241.138) port 9080 (#0)
> GET /send_1k HTTP/1.1
> User-Agent: curl/7.29.0
> Host: 172.16.241.138:9080
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 10 Feb 2023 06:33:06 GMT
< Content-Length: 0
<
* Connection #0 to host 172.16.241.138 left intact
```

4.功能原理

4.1. 流量劫持

Service Mesh 中另一个重要的话题就是如何实现流量劫持：使得业务应用的 Inbound 和 Outbound 服务请求都能够经过 Sidecar 处理。

Service Mesh 的流量劫持方案如下：



1. 服务端向自己的 Sidecar 发起服务注册请求，告知 Sidecar 需要注册的服务以及 IP 和端口。

   示例中，服务端运行在 1.**.**.4 这台机器上，监听 20880 端口，需要注册的服务以及 IP 和端口为：1.**.**.4:20880。

2. 服务端的 Sidecar 向 SOFA 服务注册中心发起服务注册请求，告知需要注册的服务以及 IP 和端口。

   这里注册上去的并不是业务应用的端口（20880），而是 Sidecar 自己监听的一个端口（例如：20881）。

3. 调用端向自己的 Sidecar 发起服务订阅请求，告知需要订阅的服务信息。

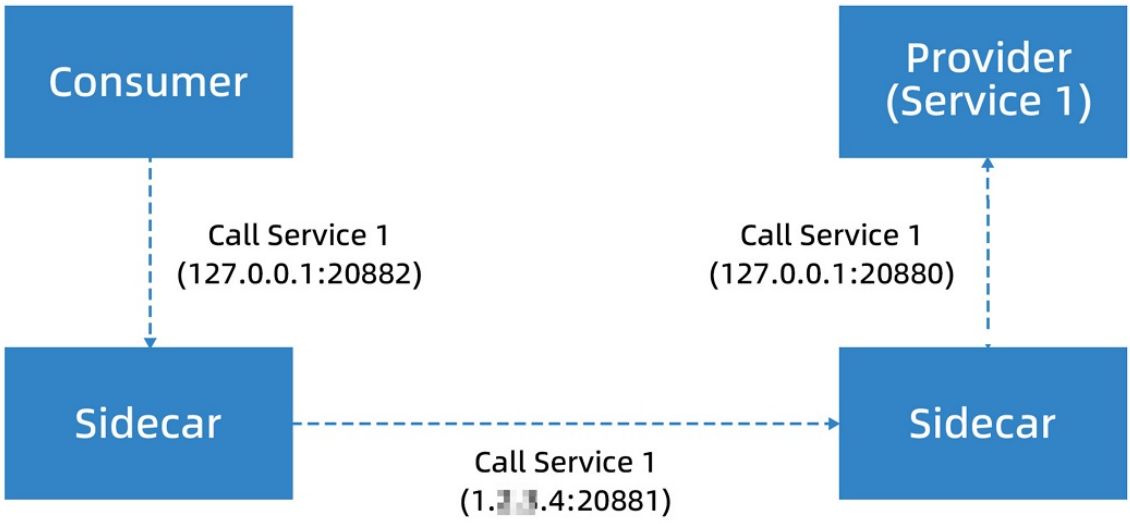
4. 调用端的 Sidecar 向调用端推送服务地址。

   这里推送的 IP 是本机，端口是调用端的 Sidecar 监听的端口（例如：20882）。

5. 调用端的 Sidecar 向 SOFA 服务注册中心发起服务订阅请求，告知需要订阅的服务信息。

6. SOFA 服务注册中心向调用端的 Sidecar 推送服务地址（1.**.**.4:20881）。

经过上述的服务发现过程后，调用过程变为如下流程：



1. 调用端向获取的服务端发起服务调用。

   示例中，获取的服务端地址是 127.0.0.1:20882，所以就会向这个地址发起服务调用。

2. 调用端的 Sidecar 接收到请求后，通过解析报文获取要调用的服务信息，然后获取之前从服务注册中心返回的地址（1.**.**.4:20881）后，发起真实调用。

3. 服务端的 Sidecar 接收到请求后，经过一系列处理，最终会把请求发送给服务端（127.0.0.1:20880）。

> **说明**
>
> 相较于 Service Mesh 的流量劫持方案，社区的 iptables 方案在规则配置较多时，性能下滑严重，且管控性和可观测性较差，出现问题难以排查。

4.2. 扩展机制

蚂蚁集团针对 Serice Mesh 扩展能力拥有完整的从内核态、开发态、运维态的产品化输出，无相关经验的用户近过整套学习体系，可以在一周技术培训的 40 小时内掌握开发到使用，独立完成简单需求。

- 内核态

  提供版本化的插件自动装填能力，每个版本会有标准可运行的二进制、镜像和母盘产出，其中母盘用于插件的编译，实现对 Go plugin 相关问题规避。

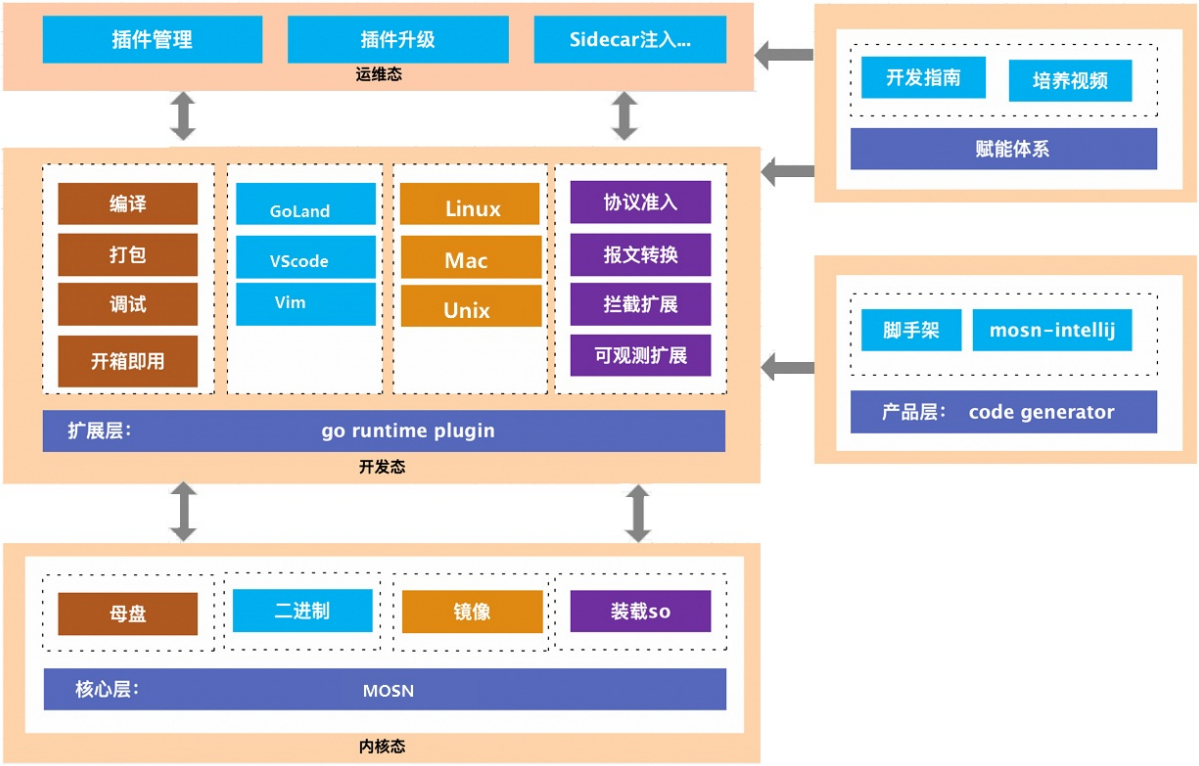
- 开发态
  - 蚂蚁开源扩展能力项目提供了完整插件研发机制，支持命令行交互式的编译、打包、调试的操作，简单易用。
  - 支持在 Linux 、MAC、UNIX 操纵系统的运行开发，用户可以选择 GoLand 和 VSCode 作为开发软件，当然针对技术能力较强的用户，也可以使用 Vim 进行开发。
  - 用户可以在当前项目开发协议、协议转换、拦截器、可观测性等插件。

- 运维态

  Serice Mesh 控制面提供包含插件自动注入数据面、插件版本管理、插件的批量升级能力。用户可以根据操作指南进行界面化的操作。

为了进一步降低用户学习成本，蚂蚁集团开放提供了完善的赋能体系和脚手架：

- 赋能体系会在业务中进行宣讲，通过一个月赋能培养，业务部门可以做到独立开发和实践。
- 脚手架的产品名字为 mosn-intellij，它可以自动化生成插件框架，减少在开发中编写误写的问题。它打通了开发态和运维态关联壁垒，可以通过 GoLand 实现在开发态到运维态的自动部署和升级的能力。



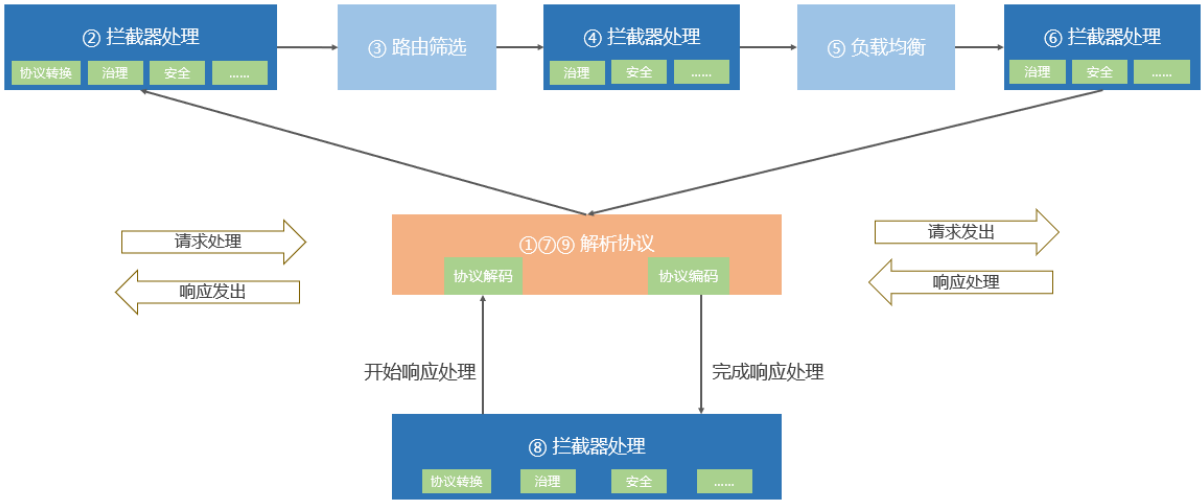
4.3. 插件介绍

Service Mesh 借助 Go plugin 机制实现数据面扩展能力。Go plugin 机制具体指将 Go 包编译为共享库（.so）的形式单独发布，主程序可以在运行时动态加载这些编译为动态共享库文件的go plugin，从中提取导出（exported）变量或函数的符号并在主程序的包中使用。

根据使用场景，插件可以拆分为流处理插件和可观测性插件两种：

流处理插件

流处理插件提供对数据流量实时处理和管控的能力，如下图展示了从 MOSN 到收到报文到发出报文整个处理过程：



1. MOSN 收到请求报文，经过协议解析获取数据报文的概要信息（head、body、tailer）。
2. 数据流先经过拦截器进行数据处理。

   这个部分通常会包含：协议报文之间的转换、header 数据的加工、服务限流等流量处理。
3. 数据流在经过拦截器第一次处理后进行路由筛选。

   这个部分通常是把某个服务下 Endpoints 拆成多组，并选择对应的一组 Endpoints。其中单元化 、灰度、就近路由等功能就是基于步骤实现。
4. 经过路由筛选之后，数据流需要再次经过拦截器处理。

   这个部分会根据步骤 3 中的路由信息进行流量治理，常见功能为流量录制、流量镜像等。
5. 数据流量在经过路由筛选后进入负载均衡，即需要根据一定的规则从一组 Endpoints 中挑选其中一个 Endpoints 实例。

   常见算法有：轮询、权重、随机 。
6. 在经过负载均衡之后，数据流量拥有完整的 RPC 请求信息，再经过一次拦截器处理，作为对当前选中的 Endpoint 进行逻辑检查。

   常见实现功能有：故障隔离、无损上下线等。
7. 流量经过 2、3、4、5、6 步骤处理后，进行协议编码并发送到某一台服务端，然后等待响应。
8. MOSN 收到响应后即经过协议解析，传递给拦截器进行数据流量处理。

   通常在这个步骤会实现协议转换、服务治理、安全等相关事项。
9. 经过拦截器处理的流量，通过协议编码转发给应用端。

以上步骤中提到协议解析、拦截器处理、路由筛选等功能将上述操作抽象为插件能力，整理成如下表格：

插件名称	功能介绍	使用场景
协议	对协议数据解码和编码过程	• 基于 TCP 层私有协议封装：例如 Dubbo。 • 基于 HTTP 协议定制化封装：例如 Spring Cloud。
协议转换	协议报文结构映射转换	实现 Dubbo 和 Spring Cloud 之间互转。

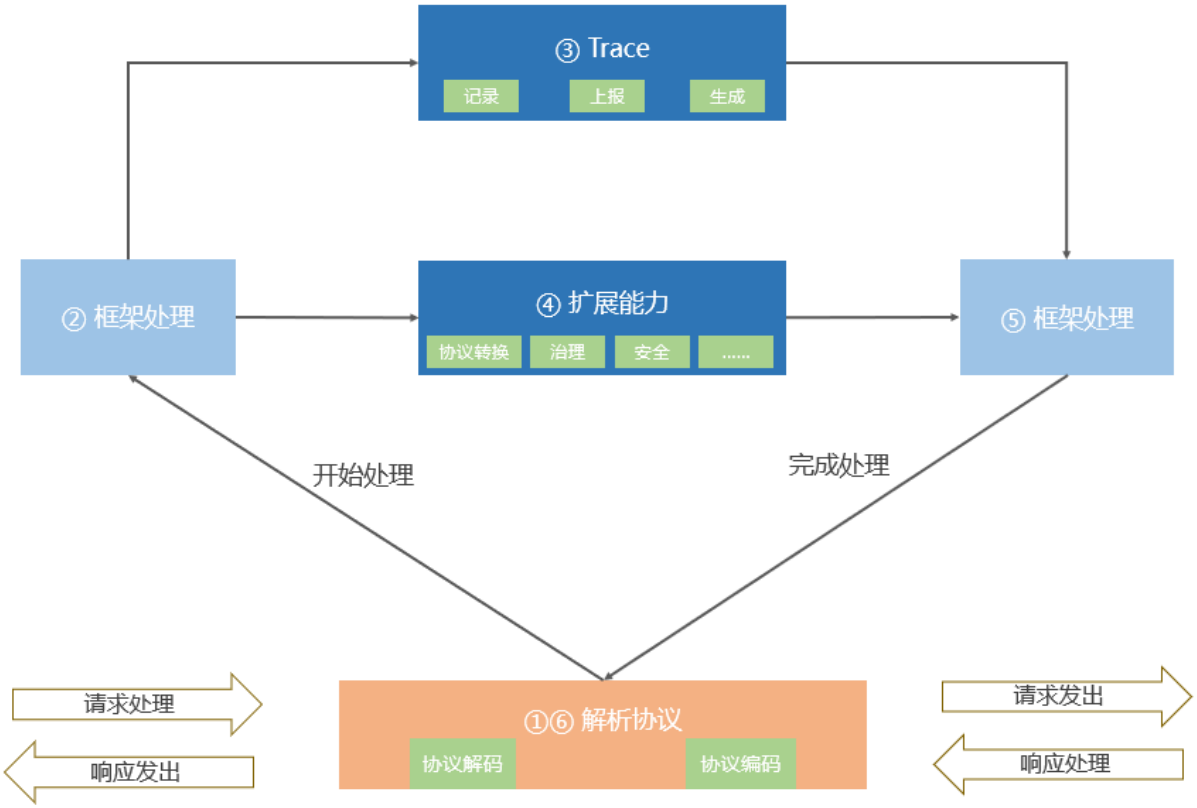
拦截器	对于数据流量加工和管理	• 服务治理：限流、熔断、故障隔离、故障注入等。 • 安全管理：防篡改、访问鉴权等。
路由插件	从服务列表中筛选出一组可用实例。	灰度路由、权重路由、就近路由等。

可观测插件

可观测能力是网格落地的必经过程，目前插件支持 Trace、日志、metrics 三部分扩展开发，可以根据行内自定义规范进行适配开发，灵活便捷。

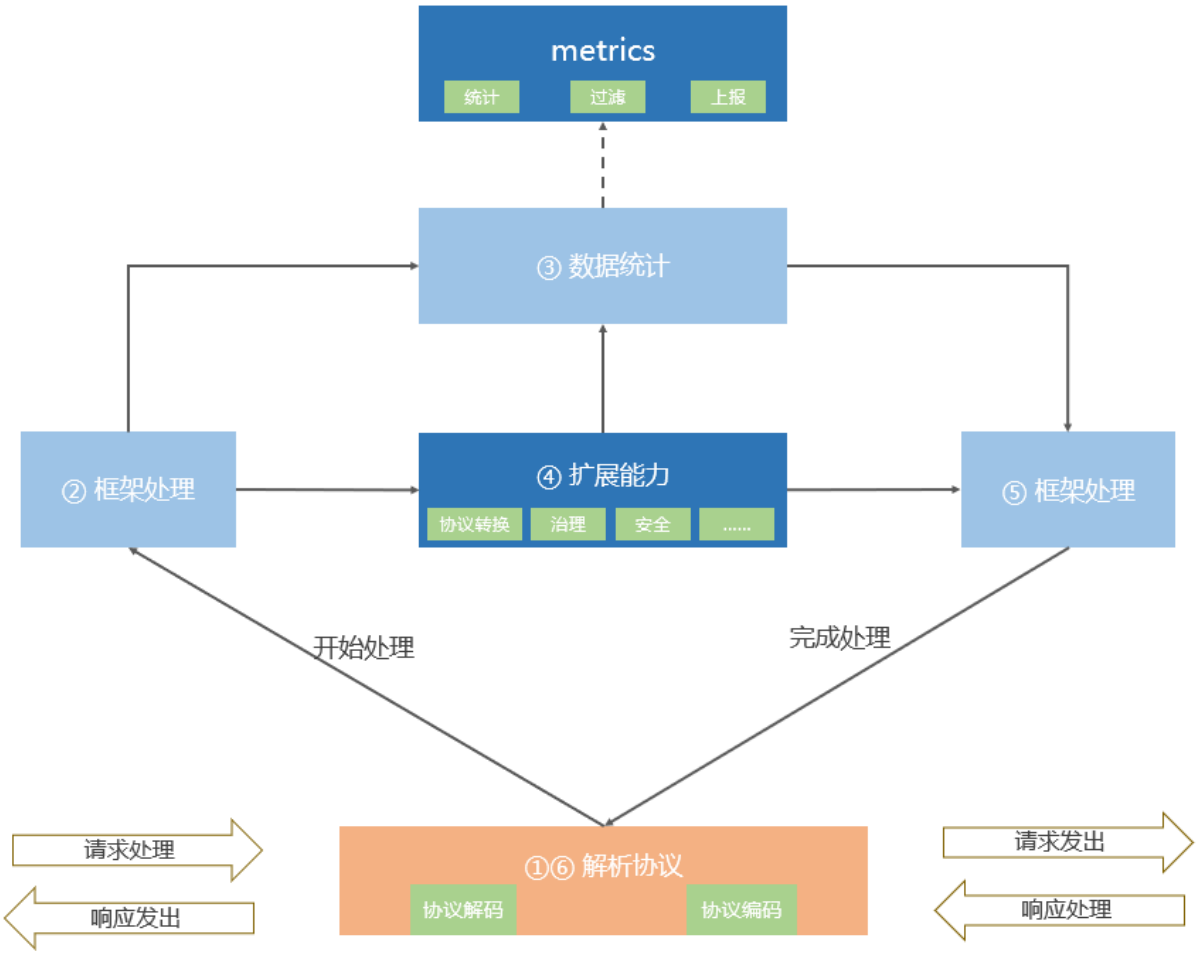
Trace 扩展

在协议解析时，框架通过 Trace 插件生成链路 span（span 中保存在请求上下文中，记录每一步步骤操作的概要信息）。在请求发出发出的前一刻，框架会通知 Trace 插件结束当前的 span 生命周期。Trace 插件会异步处理进行数据上报。常见的 Trace 插件有 skywalking、zipkin 等。



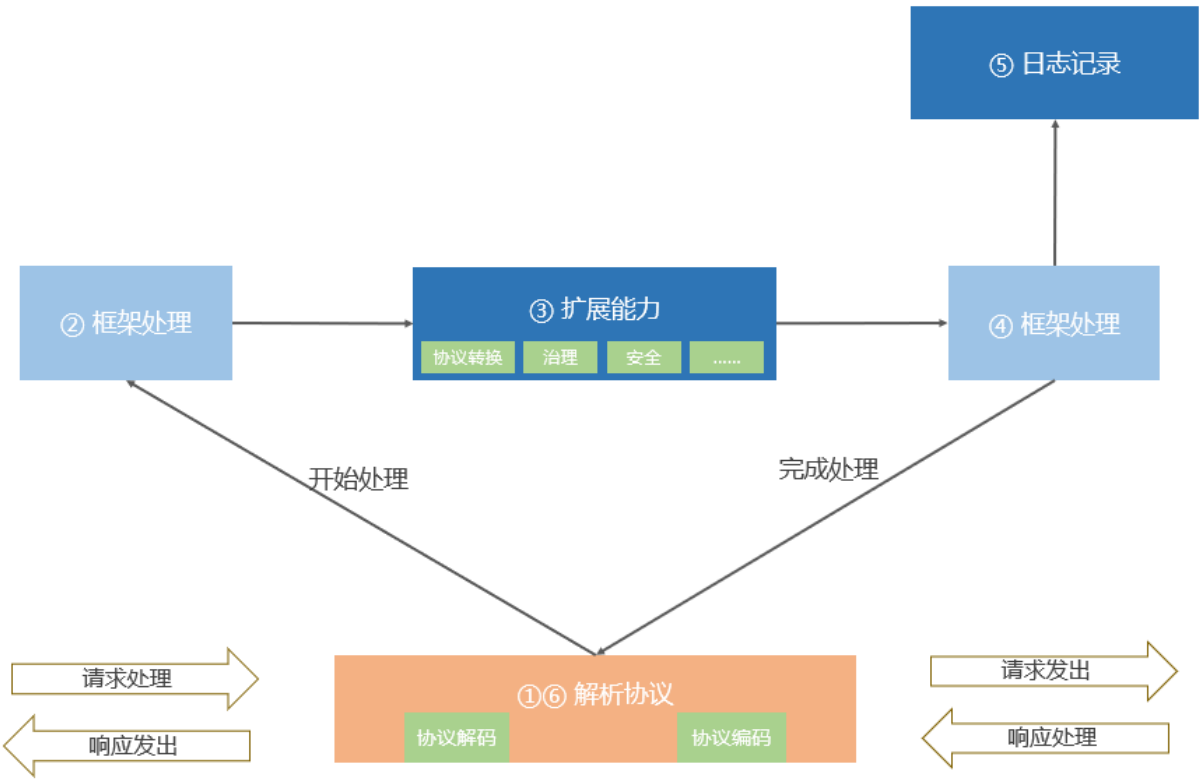
metrics 扩展

metrics 是统计数据运行时的状态信息，在 MOSN 内部会对数据运行状态进行统计，用户可以使用插件对统计数据进行加工上报自定义的运维监控平台。目前这部分已经实现对标准 Promethues 的对接。



日志扩展

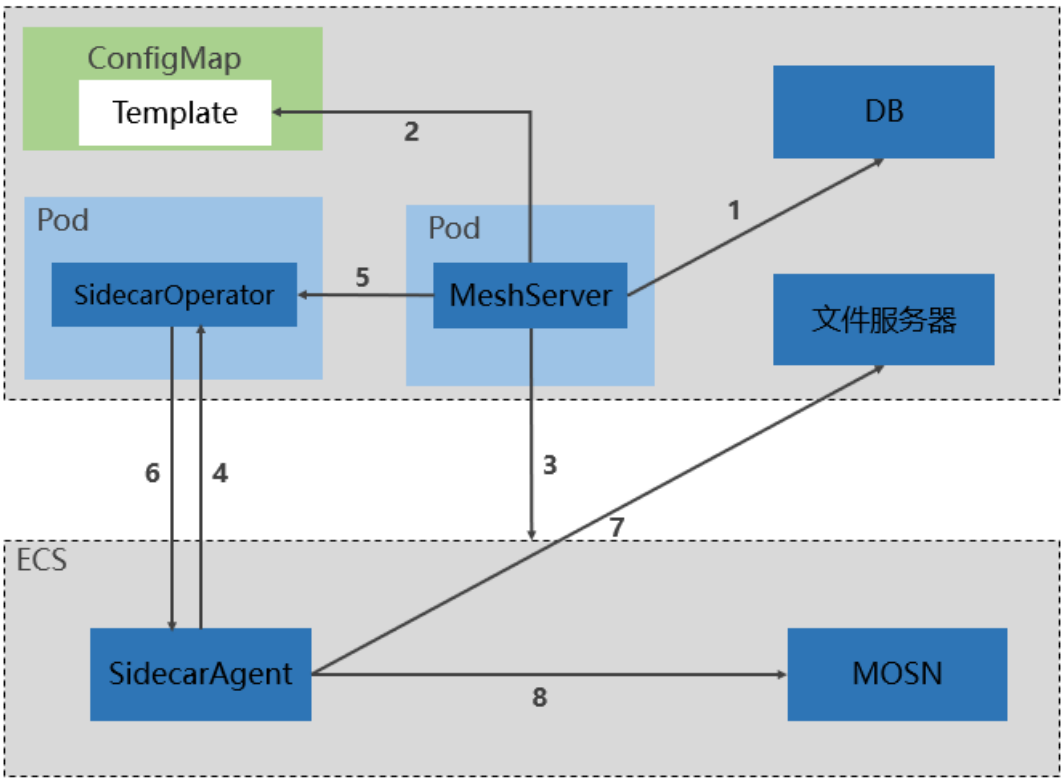
日志插件用于系统或应用运行过程中所产生的记录，是故障排查的依据。这部分能力是在请求结束之后统计记录。



4.4. 虚拟机支持

在云原生架构下，Sidecar 借助于 K8s 的 Webhook/Operator 机制可以方便地实现注入、升级等运维操作。然而大量系统还没有运行在 K8s 上，所以我们通过 Agent 的模式来管理 Sidecar 进程，从而可以使 Service Mesh 能够帮助老架构下的应用完成服务化改造，并支持新架构和老架构下服务的统一管理。

虚拟机场景的方案如下：



1. 登录 MeshServer 控制台开通集群，录入虚拟机的元信息（包含虚拟机的 IP、用户名和密码、机房和地域信息）。
2. 在控制台配置 Sidecar 模板，建立注入规则。

   创建规则后，会自动生成 ConfigMap 模板。
3. MeshServer 登录 ECS 虚拟机部署 SidecarAgent 组件。
4. SidecarAgent 上报元数据给 SidecarOperator（包括 Tenant、Workspace、Instance、指定的标签等），SidecarOperator 将元数据存储为一个单独的 ConfigMap。
5. MeshServer 向 SidecarOperator 发起在 ECS 虚拟机上装载并启动 Sidecar 的请求。
6. SidecarOperator 根据 MeshServer 请求中的模板名称，在缓存中获取 Sidecar 启动模板，然后将模板信息发送给 SidecarAgent 进行 Sidecar 部署。
7. SidecarAgent 根据模板信息向文件服务请求下载 MOSN 的部署安装包。

   文件服务器是由业务方提供的 FTP 服务器。
8. SidecarAgent 根据模板信息渲染 Sidecar 启动配置，然后启动 Sidecar。

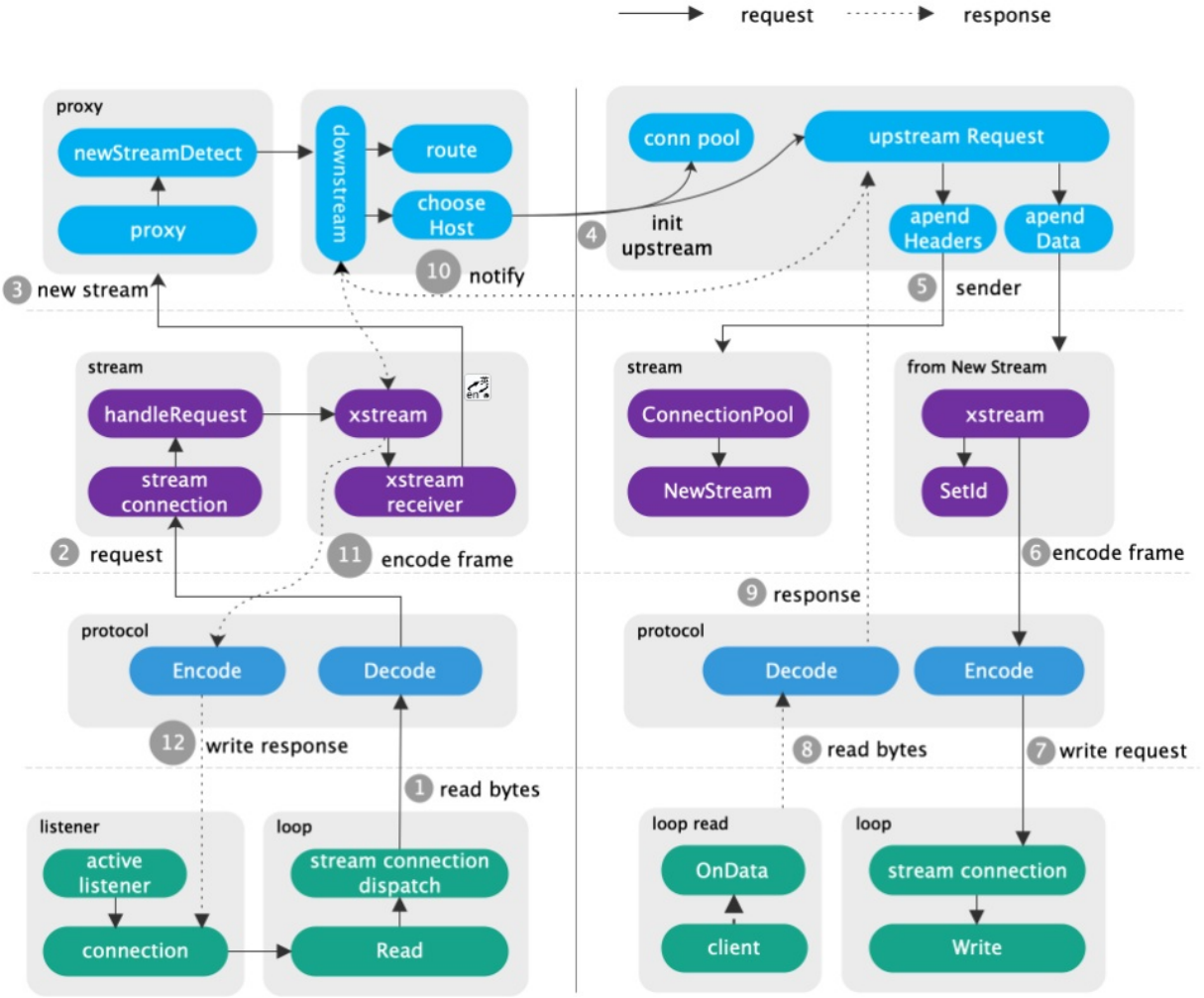
4.5. Mesh 核心转发流程

MOSN 作为数据面，整体分为 4 层：

- Network/IO 主要负责网络读写。
- Protocol 主要负责协议编解码。
- stream 主要负责 request、response 等模型转发和协调。
- proxy 主要负责路由等功能。

原始的 MOSN 简介，请参见 MOSN 架构简介。

为了便于理解，本文将 MOSN 的流程简化为 12 个步骤，如下图所示：



1. MOSN 在启动期间，会暴露本地 egress 端口接收客户端 App 的 RPC 请求。MOSN 会开启 2 个协程，分别死循环去对 TCP 进行读和写处理。MOSN 会通过读协程获取到请求字节流，进入MOSN的协议层处理。
2. MOSN 会通过 streamconn 实现类中循环解码，直到解析到完整的请求报文。一旦解析到请求报文，会创建和请求 frame 关联的 xstream。

   这里的 xtream 用来保持和客户端 App 的 TCP 关联，后续用来用于响应 response。
3. MOSN 需要将请求转发到服务集群的某一台机器，会到达 proxy 层创建 downstream。

   此步骤实现目的如下：

   - 执行 filter 请求/响应链。
   - 执行路由匹配。
   - 执行负载均衡。
4. 在选择服务集群的某一台后，MOSN 会首先初始化选中 IP 对应 host 的连接池。

   此时 MOSN 的角色变成了中间人的角色。一方面需要承担客户端 App 的服务端，另一方面需要承担远程服务方的客户端。

   upstreamrequest 对象起到关键作用：

- 保持着对客户端 App 的 TCP 引用。
- 保持着对转发服务端 TCP 引用，转发客户端 App 请求以及响应服务端 response 时的通知。

5. 在 upstream 的 appendHeaders + appendData 阶段，会用第 4 步骤中选择的 host 创建 sender xstream。

   xstream 是客户端的流对象，主要有 2 个目的：

   - 充当 Client 角色，初始化客户端请求信息，将待转发的请求创建对应的 stream 绑定关系。
   - 在真正转发前，需要替换请求 ID 信息，用来解决连接 IO 复用导致请求互相覆盖的问题。

   > **说明**
   >
   > 出现请求相互覆盖问题的原因：客户端 App 有多个 TCP 连接，MOSN 转发到服务端只有 1 条 TCP 连接。如果客户端 2 个 TCP 连接同时有个 ID=1 的 request，MOSN 会通过同一条 TCP 转发给服务端，因此响应回来时，MOSN 没办法区分 ID=1 的 response 属于哪个客户端 App 的 TCP 连接。
   >
   > 这里的解决办法是 MOSN 转发到服务端的 1 条 TCP 连接，会重新修改请求的 ID 成全局，就不会混淆请求和响应关联关系了。

6. 因为 MOSN 在转发之前修改了请求 ID，因此会重新 encode 请求。

   一般优化手段不会 encode 完整报文，只会修改协议头的个别几个字节。

7. 一旦客户端 xstream 准备转发就绪（endOfStream），就会通过第 4 步骤中选择下游 host 直接发送。

   此时请求的携程会被阻塞。

8. MOSN 转发给服务端 host 时，会新建 TCP 连接。此时，每个 TCP 连接也会有 2 个携程去处理读写。

   MOSN 客户端 xstream 的会通过读 IO，收到响应字节流，并且交付上层 protocol 去解码。

9. 一旦完整解码成 response 对象，会通知 upstream request 对象。

0. upstream request 持有客户端请求的 downstream，唤醒 downstream 阻塞的协程。

1. 对应步骤 2 中 MOSN 作为服务方 xstream 被唤醒，会将收到的响应 response，重新替换回正确的 reqest id，并能且去调用协议层重新 encode 成字节流。

2. xstream 中持有客户端 App 请求时的 TCP 连接，直接将响应写会客户端，并且销毁 MOSN 中所有请求相关的资源。

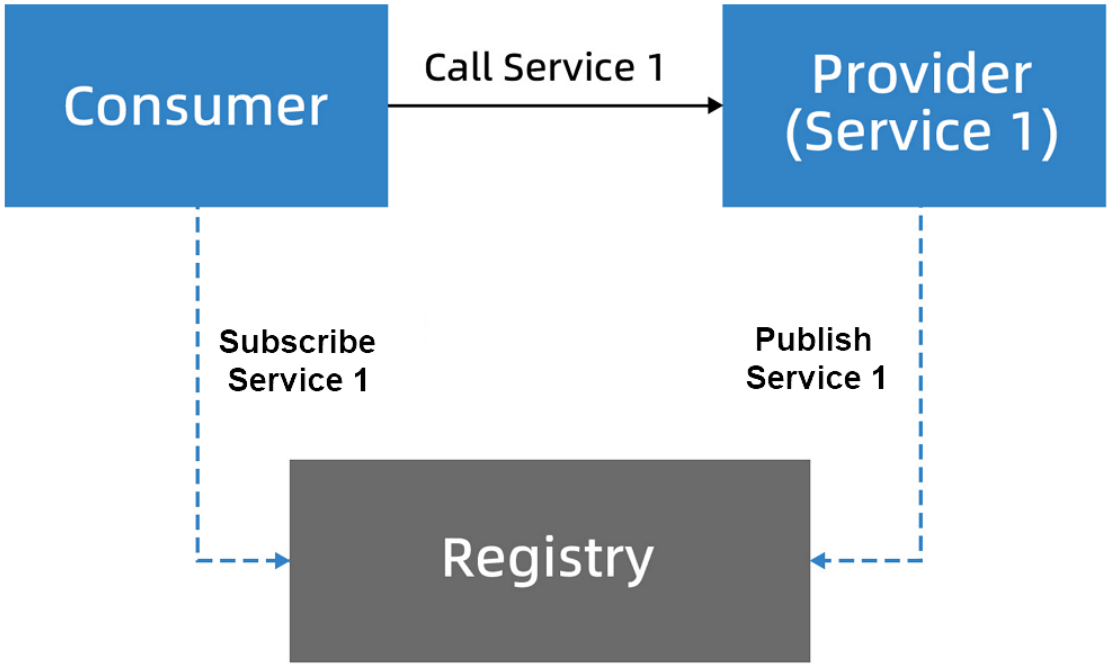
4.6. 平滑迁移

平滑迁移可能是整个方案中最为重要的一个环节了，前面也提到，在目前任何一家公司都存在着大量的 Brownfield 应用，它们有些可能承载着公司最有价值的业务，稍有闪失就会给公司带来损失，有些可能是非常核心的应用，稍有抖动就会造成故障。所以对于 Service Mesh 这样一个大的架构改造，平滑迁移是一个必选项，同时还需要支持可灰度和可回滚。

ServiceMesh 平滑迁移方案流程如下：

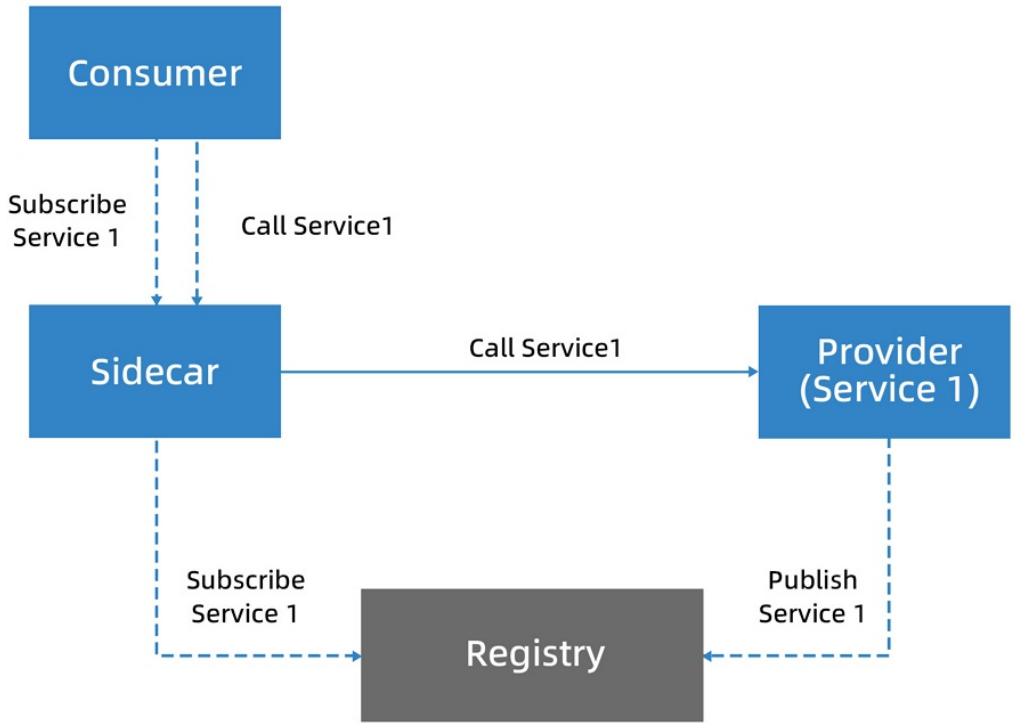
1. 初始状态。

   以一个服务为例，初始有一个服务提供者，有一个服务调用者。



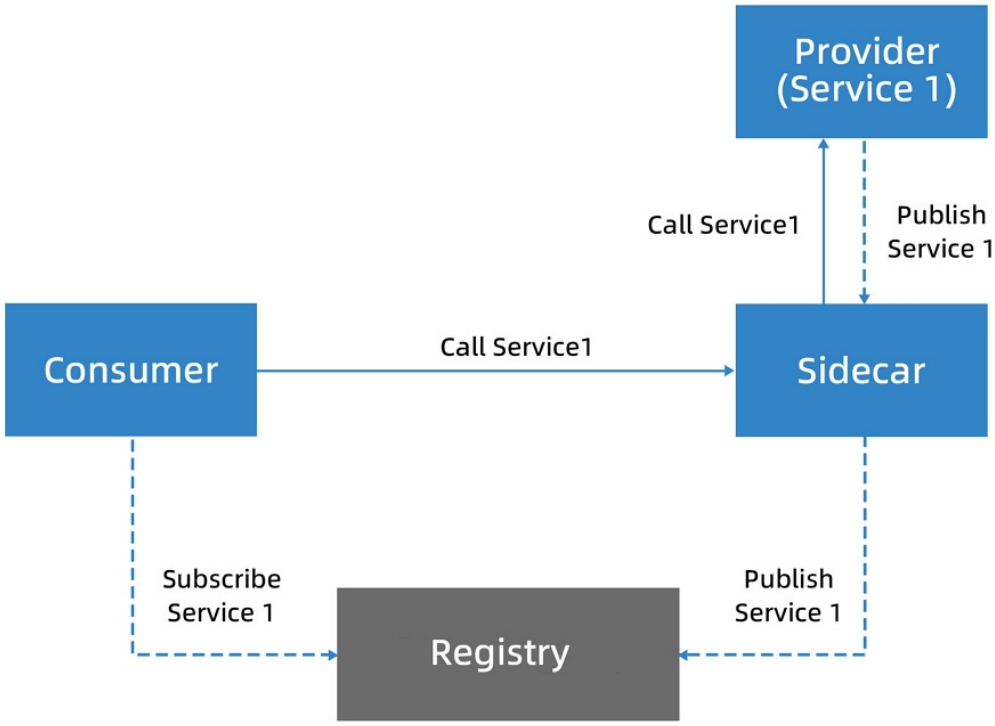
2. 透明迁移调用方。

   在我们的方案中，对于先迁移调用方还是先迁移服务方没有任何要求，这里假设调用方希望先迁移到 Service Mesh 上，那么只要在调用方开启 Sidecar 的注入即可，服务方完全不感知调用方是否迁移了。所以调用方可以采用灰度的方式一台一台开启 Sidecar，如果有问题直接回滚即可。

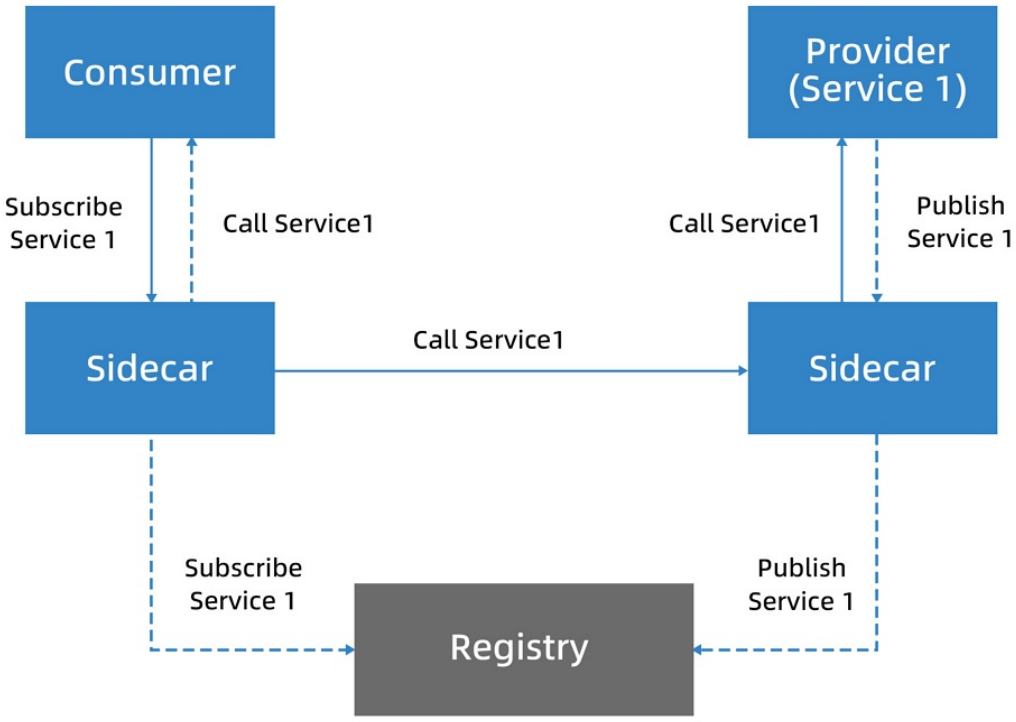


3. 透明迁移服务方。

假设服务方希望先迁移到 Service Mesh 上，那么只要在服务方开启 Sidecar 的注入即可，调用方完全不感知服务方是否迁移了。所以服务方可以采用灰度的方式一台一台开启 Sidecar，如果有问题直接回滚即可。



4. 最终状态。



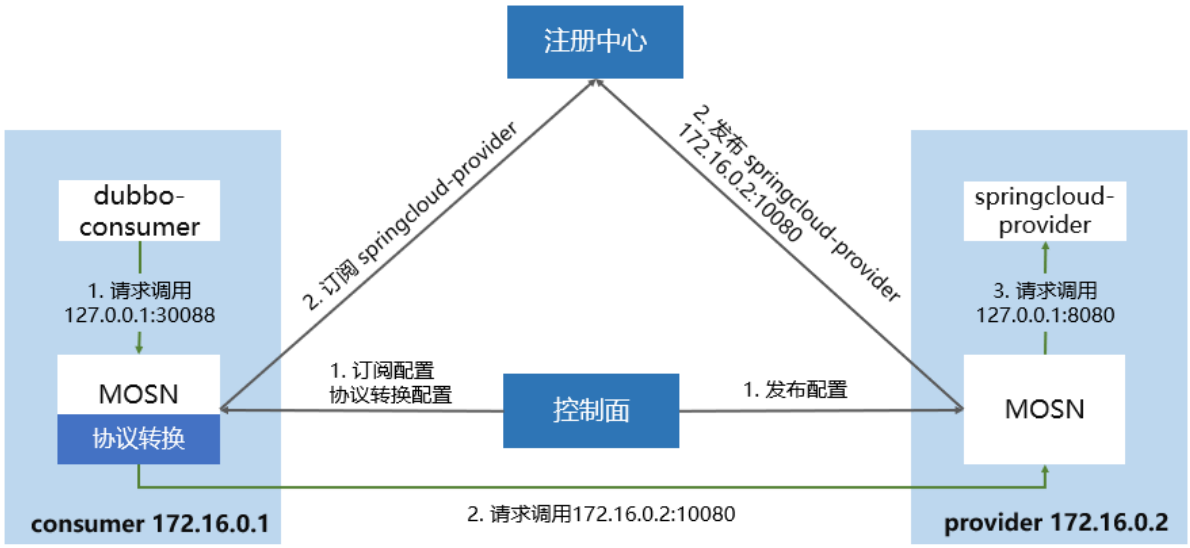
4.7. 异构系统集成

由于现代软件系统的复杂性和多样性，软件异构集成变得越来越重要。大多数实现都是由各自业务中心提供自有的一套集成规则，实现各不相同，无法做到统一。在云原生时代，Serivce Mesh 通过抽象中间层，在 Sidecar 中实现转换，架构部门可以做到整体统一设计和实现。

下面通过示例说明如何通过 Sidecar（MOSN）完成异构的集成。现在有 Spring Cloud 和 Dubbo 两个应用分别为 dubbo-consumer 和 springcloud-provider，不同场景的流程如下：

双 Mesh 场景

下图展示了 dubbo-consumer 访问 springcloud-provider 应用双 Mesh 使用场景。在启用 Sidecar 后，应用不需要任何改动，协议转换可以通过客户端的 Sidecar 实现，整体过程交互如下：



发布应用

1. 控制台配置应用 springcloud-provider 的发布配置信息。
2. MOSN 启动后自动拉取控制台配置，provider MOSN 向注册中心发布注册。

订阅应用

1. 控制台配置应用 dubbo-consumer 的订阅配置信息。
2. MOSN 启动后自动拉取控制台配置，consumer MOSN 向注册中心发布订阅。

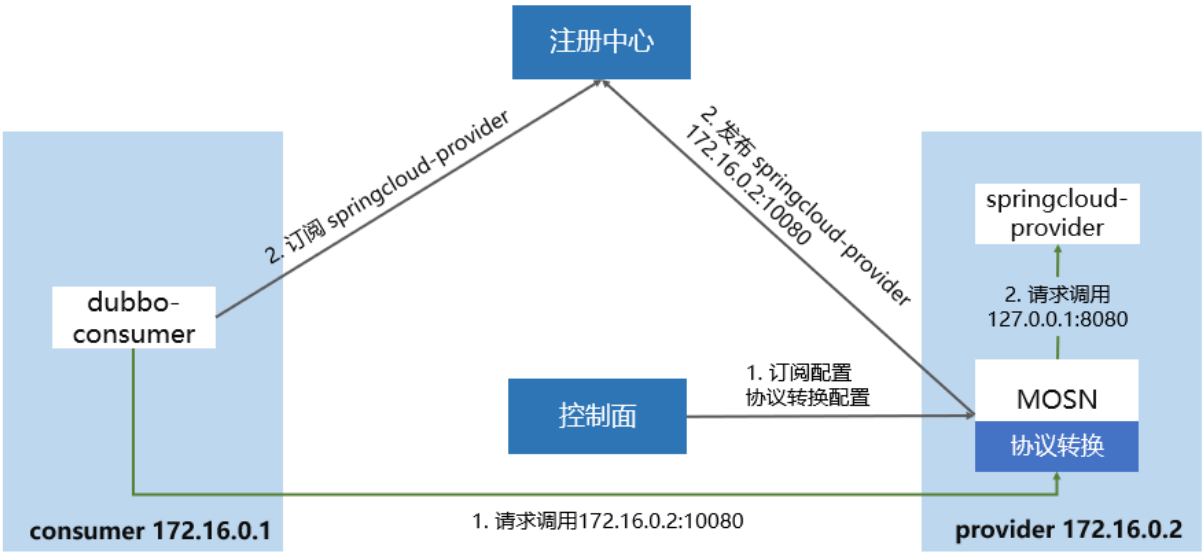
应用通信

1. dubbo-consumer 主动发送或者通过透明劫持方式把请求转到 MOSN 端口 127.0.0.1:30088。
2. dubbo-consumer MOSN 实现 Dubbo 协议转换 Spring Cloud 协议（具体转换规则可以业务协商，通过协议转换插件实现业务自定义方式），然后发送 Spring Cloud 协议请求到 provider MOSN。
3. MOSN 收到请求后，经过服务治理能力后转发到 springcloud-provider 应用。

单 Mesh 场景

服务端

下图展示了 dubbo-consumer 访问 springcloud-provider 应用的单 Mesh 使用场景。在启用 Sidecar 后，应用不需要任何改动，协议转换可以通过服务端的 Sidecar 实现，整体过程交互如下：



发布

1. 控制台配置应用 springcloud-provider 发布配置信息。
2. MOSN 启动后自动拉取控制台配置，provider MOSN 向注册中心发布注册。

订阅

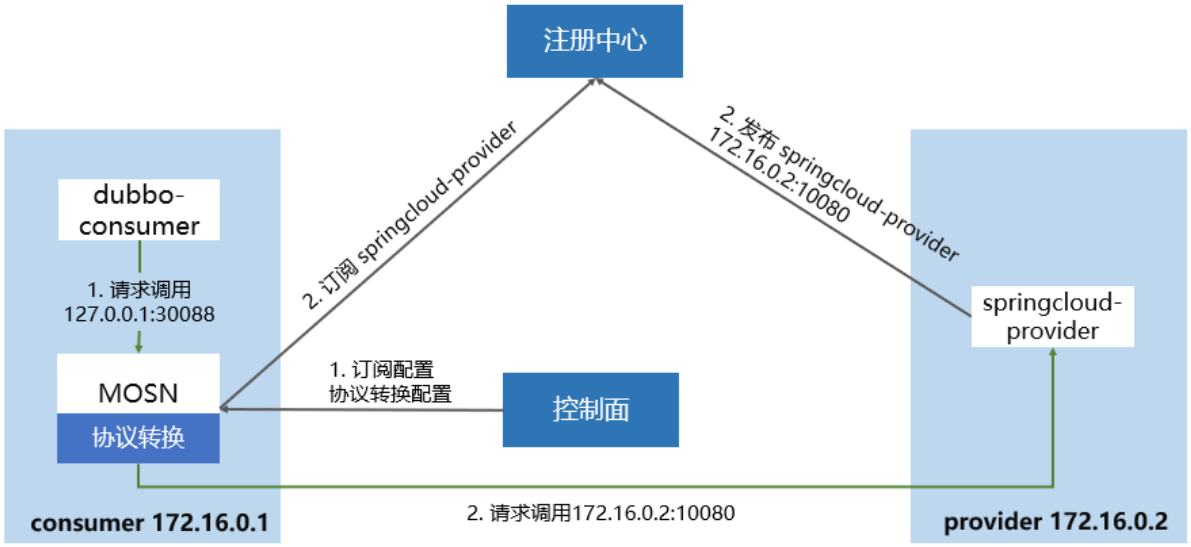
应用 dubbo-consumer 直接向注册中心进行订阅，获取服务信息。

通信

1. dubbo-consumer 把请求转发到 provider 中的 MOSN 端口 172.16.0.2:10088。
2. dubbo-consumer MOSN 实现 Dubbo 协议转换 Spring Cloud 协议（具体转换规则可以业务协商，通过协议转换插件实现业务自定义方式），然后发送 Spring Cloud 协议请求到 provider。

客户端

下图展示了 dubbo-consumer 访问 springcloud-provider 应用单 Mesh 使用场景，在启用 sidecar 后，应用不需要任何改动，协议转换可以通过服务端的 sidecar 实现，整体过程交互如下：



发布

springcloud-provider 直接向注册中心发布注册。

订阅

1. 控制台配置应用 dubbo-consumer 的订阅配置信息。
2. MOSN 启动后自动拉取控制台配置 ，consumer MOSN 向注册中心发布订阅。

通信

1. dubbo-consumer 主动发送或者通过透明劫持方式把请求转到 MOSN 端口 127.0.0.1:30088。
2. dubbo-consumer MOSN 实现 Dubbo 协议转换 Spring Cloud 协议（具体转换规则可以业务协商，通过协议转换插件实现业务自定义方式），然后发送 Spring Cloud 协议请求到 provider。

4.8. 多注册中心集成与迁移

在云原生时代，软件演进速度日新月异。其中，云注册中心是云原生架构中较为重要的基础设施服务之一，用于管理服务的注册和发现，从而保证微服务之间的正常通讯和流量均衡。

目前，有一些比较流行的云原生注册中心实现，例如：

- SOFARegistry：阿里巴巴开源的服务注册中心，是一种基于云原生架构的服务注册和发现工具。
- Consul：HashiCorp 公司开发的可以用来支持分布式系统或服务的发现与配置软件。
- ZooKeeper：Apache 发布的分布式存储系统。
- Eureka：Netflix 开源的一个注册中心，主要面向 Java 开发的云服务。

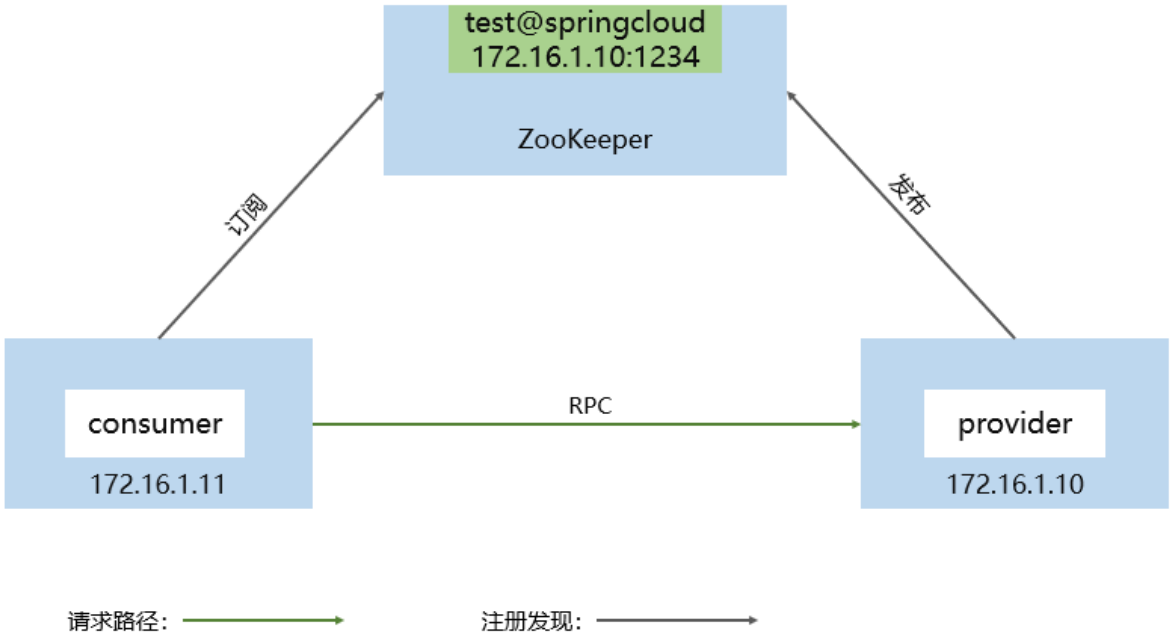
以上这四种云原生注册中心或存储后端都有广泛的应用场景，由于历史原因，大多数公司可能同时包含多种注册中心，注册中心之间如何平滑的迁移统是各个公司面临的一大难题。

Service Mesh 作为网络代理，不仅可以管理 RPC 请求，还可以助力注册中心的切换。假设当前一家公司同时有 ZooKeeper 和 SOFARegistry 两种注册中心。由于历史原因，其使用 ZooKeeper 的服务，但没有精力投入微服务改造，Service Mesh 可以代替他完成从 ZooKeeper 到 SOFARegistry 的改造。

整个过程可以可以分为现状、集成、终态三阶段：

现状

当前应用接入 ZooKeeper 完成微服务之间的注册发现，如下图所示：



发布

consumer 向 ZooKeeper 的 2181 端口发起服务注册请求，注册信息包含服务名称（test@springcloud）和地址（172.16.1.10:1234）。

订阅

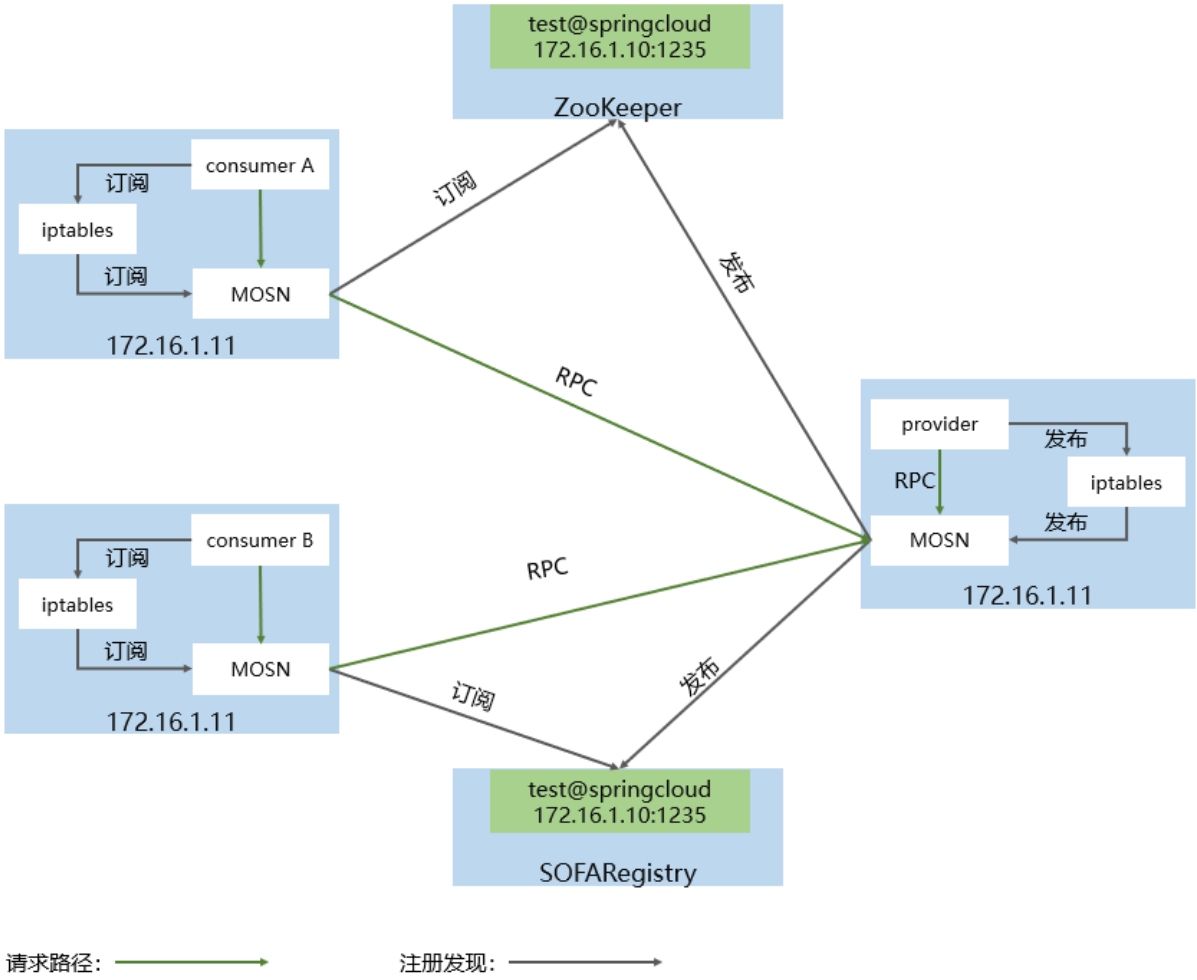
provider 向 ZooKeeper 的 2181 端口发起服务订阅请求，订阅信息包含服务名称（test@springcloud）和地址（172.16.1.10:1234）。

请求链路

consumer 向 172.16.1.10:1234 地址的 provider 发起请求。

集成

集成后，provider 保持透过透明劫持接入 MOSN，comusmer 变更为 comsumer A 和 comsumer B 两组服务。其中，comsumer A 依旧通过透明劫持方式接入 MOSN，服务发现依赖 ZooKeeper；comsumer B 服务发现的注册中心从 ZooKeeper 切换为 SOFARegistry。具体如下图所示：



发布

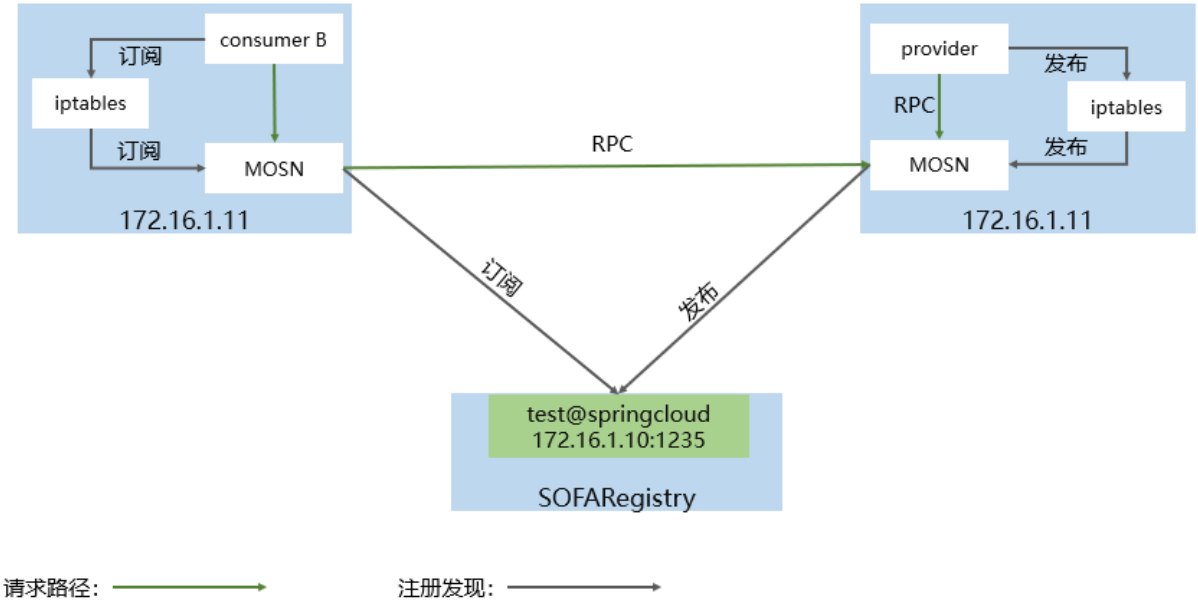
1. provider 向 ZooKeeper 的 2181 端口发起服务注册请求。
2. iptables 规则拦截请求，并将请求转发到 MOSN 的 12181 端口。
3. MOSN 将请求中注册的端口修改成 MOSN 对应协议的监听端口，然后向注册中心 ZooKeeper 发起注册请求。MOSN 会同步向 SOFARegistry 进行注册。

订阅

1. consumer 向 ZooKeeper 的 2181 端口发起服务订阅请求。
2. iptables 规则拦截请求，并将请求转发到 MOSN 的 12181 端口。
3. MOSN 向注册中心 SOFARegistry 或者 ZooKeeper 发起订阅请求获取服务列表，同时缓存服务列表，并将修改的服务列表返回给 consumer。

终态

在 comsumer A 服务切换为 comsumer B 服务后，ZooKeeper 服务可以进行下线，整体链路服务的注册发现切换为 SOFARegistry，如下图所示：



发布

1. provider 向 ZooKeeper 的 2181 端口发起服务注册请求。
2. iptables 规则拦截请求，并将请求转发到 MOSN 的 12181 端口。
3. MOSN 将请求中注册的端口修改为 MOSN 对应协议的监听端口，然后向注册中心发起注册请求。

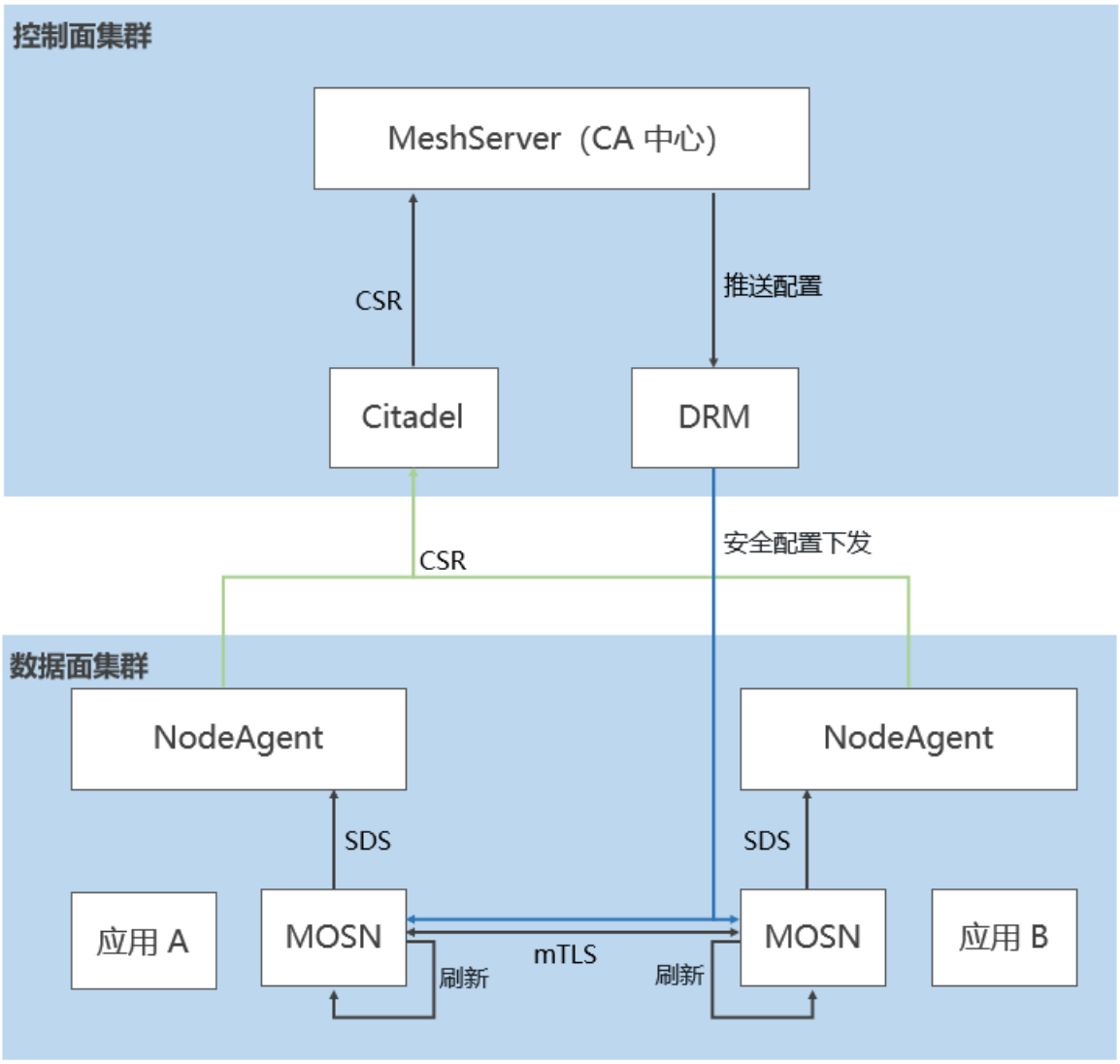
订阅

1. consumer 向 ZooKeeper 的 2181 端口发起服务订阅请求。
2. iptables 规则拦截请求，并将请求转发到 MOSN 的 12181 端口。
3. MOSN 向注册中心 SOFARegistry 发起订阅请求获取服务列表，同时缓存服务列表，并将修改的服务列表返回给 consumer。

4.9. 安全能力

目前 Service Mesh 针对生产部署提供对接三方 CA 的能力，在日常测试 POC 场景控制台提供 CA 中心能力。

SOFAMesh 安全架构



架构描述

组件介绍

组件名称	作用	备注（在 SOFAMesh 安全体系下）
MOSN	Mesh Sidecar	加密通信、防篡改通信等。
CitadelAgent	安全组件	作为 Sidecar 的 SDS Server，向 Citadel 发起 CSR 请求。
Citadel	Istio CA中心	仅负责下发应用信息，用于签发应用级证书。
DRM	配置中心	在 SOFAMesh 安全体系中，负责通知 MOSN 证书状态，触发证书轮转、吊销等动作。

MeshServer（历史名称为：dsrconsole）	CA中心	负责证书的管理，如签发、轮转、吊销、查询等。

支持功能

- 安全加密通信

  证书在全链路传递过程中，采用 TLS 方式传输，防止传输窃取。

- 防篡改通信
  - 证书目前是明文保存到数据库，未来会通过加密保存。
  - 在证书在 Citadel、CitadelAgent 组件透传，MOSN 仅在内存操作，整个过程不落盘。
- 安全功能所需的证书管理功能（证书签发、证书轮转、证书吊销）
  - 证书签发

    证书签发流程如下：

    a. 用户开启安全加密。

    b. MOSN 发起 SDS 请求，与本地 Citadel Agent（SDS Server）进行通信。

    c. Citadel Agent 收到请求后，向 Citadel 发起 CSR 请求，并向报文中添加应用的相关信息。

    d. Citadel 收到请求后，将请求转发至 Mesh Server（CA中心）。

    e. MeshServer 根据 CSR 报文中的信息，签发应用级证书及私钥。
  - 证书轮转

    证书轮转流程如下：

    a. Mesh Server 每小时轮询检查证书状态。

    b. 证书到期前 48 小时，将打印过期警告日志。

    c. 证书到期前 24 小时，将触发证书自动轮转，签发新证书，并向 MOSN 推送更新信息。

    d. MOSN 收到消息后，将重新拉取证书，并更新至内存。
  - 证书吊销

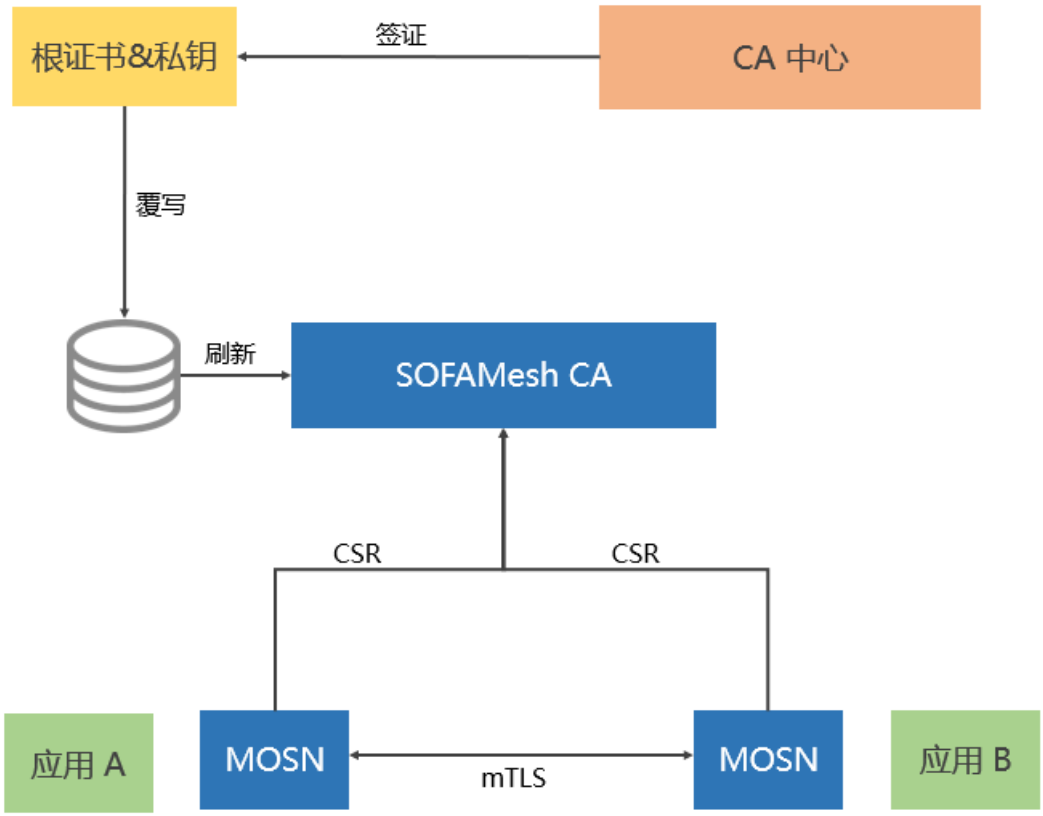
    证书吊销流程如下：

    a. 用户触发吊销操作。

    b. Mesh Server 更新数据库中的证书状态，并向 MOSN 推送消息。

    c. MOSN 收到消息后，删除内存中的证书，并重新申请签发证书。

CA 中心托管



未启用安全场景

未启用安全场景特指还没有应用使用安全功能，当前 CA 中心未颁发应用证书。若已签发证书，需要兼容旧证书。对接步骤如下：

1. 开启安全功能前，由客户的安全体系签发 CA 证书（commonName="dsr-ca"）、私钥，并写入数据库。
2. 写入数据后，推送动态配置或重启 MeshServer，刷新证书体系。
3. 当应用申请证书时，将使用新私钥进行签发，并使用新证书进行验证。

已启用安全场景

若存在应用使用了安全功能（即存在存量已签发的应用证书，相关应用已开启相关功能，且处于运行中），此时直接更新私钥、根证书，将导致新 CA 证书（根证书）无法验证应用证书。

对于此场景，若要兼容场景，且应用不发生变更，则签发的私钥需要使用旧的私钥进行签发。若能接受变更已接入应用的方式（重新开启安全功能，不适用大量应用已接入的场景），则也可以参考未启用安全场景的操作方式更换根证书和私钥，同时清空其他应用证书，而后应用重启安全功能，重新签发证书。

4.10. 可观测性实践

可观测性三大支柱围绕指标、日志、链路追踪展开，企业会打造一个与具体监控产品无关而又服务于各类监控产品的数据平台。当下产品已经集成了 OpenTelemetry、SkyWalking、Zipkin 的链路追踪和监控，已经实现对 Prometheus 的支持，还可以通过扩展能力数据面支持自定义对接三方平台，具备灵活的接入能力。

指标

通过 Service Mesh 不仅可以统计网络链接通信性能，还可以实时反馈服务节点健康状态。包括如下指标：

- 请求成功率：衡量请求在 Service Mesh 中是否成功完成。

- 延迟：衡量请求在 Service Mesh 中完成所需的时间。
- 流量：衡量 Service Mesh 中的流量。
- 错误率：衡量请求在 Service Mesh 中遇到的错误率。
- 熔断：衡量在 Service Mesh 中出现的熔断事件。
- 重试：衡量在 Service Mesh 中出现的重试事件。
- 健康检查：衡量 Service Mesh 中的健康检查。

这些指标可以帮助开发人员和运维人员了解 Service Mesh 中服务的运行情况，以及服务之间的流量和性能问题。这些指标还可以用于优化 Service Mesh 的配置和调整，以提高其性能和可靠性。

当前 Service Mesh 已经提供对接 Prometheus 的拉取和推送两种能力，同时用户可以通过扩展插件的方案实现数据指标的自定义格式上报。如下表格展示了对接 Prometheus 重点指标：

指标名称	指标 Key	聚合维度
下游链接数	downstream_connection_active	端口维度
下游请求 QPS	downstream_request_active	端口维度
下游请求平均耗时	downstream_request_time_total / downstream_request_total	端口维度
请求总数	downstream_request_total	端口维度
请求耗时	downstream_request_time_total	端口维度
MOSN 处理耗时	downstream_process_time_total	端口维度
请求失败统计	downstream_request_failed	端口维度
上游链接数	upstream_connection_active	集群维度
上游请求 QPS	upstream_request_active	集群维度
上游请求失败次数	upstream_request_failure_eject	集群维度
上游请求平均耗时	upstream_request_duration_time_total / upstream_request_total	集群维度

日志收集

Service Mesh 可以检测到网格内的服务通信的流转情况并生成详细的遥测数据日志。用户可以通过插件或者配置的方式实现日志的记录。通常情况下日志包含字端如下：

字段名称	含义
appname	应用的名称
target.app	目的端 App 名称
hijack	透明劫持
source.app	调用端 App 名称
rpc.method	目的端方法
rpc.service	服务标识
upstream.address	目的端地址
downstream.address	调用端地址
downstream.protocol	调用端协议
host.name	目的端 host 名称
upstream.protocol	目的端协议
request.size	请求大小
respone.size	响应大小
duration	整体耗时
mosn.process.duration	MOSN 处理耗时
mosn.process.request.duration	MOSN 请求处理耗时

mosn.process.respone.duration	MOSN 响应处理耗时
mosn.process.fail	MOSN 处理失败标志
http.url	请求 URL HTTP 特有
http.method	请求方法 HTTP 特有

链路追踪

Service Mesh 链路追踪是指通过在服务网格中实现的一些技术手段，对服务调用链路进行追踪和监控。在服务网格中，每个服务节点都会被注入一个 Sidecar 代理，这个代理负责收集和发送服务节点的监控信息，同时还可以进行服务发现、负载均衡等功能。在这个代理的帮助下，服务网格可以收集和处理服务之间的各种交互信息，来实现链路追踪和监控。

服务网格中的链路追踪通常包括以下几个步骤：

1. 收集调用信息。

   代理会记录每次服务调用的相关信息，例如调用方和被调用方的信息、请求参数和响应结果等。

2. 构建调用链路。

   通过将不同服务节点之间的调用信息进行关联，可以构建服务调用链路。

3. 可视化调用链路。

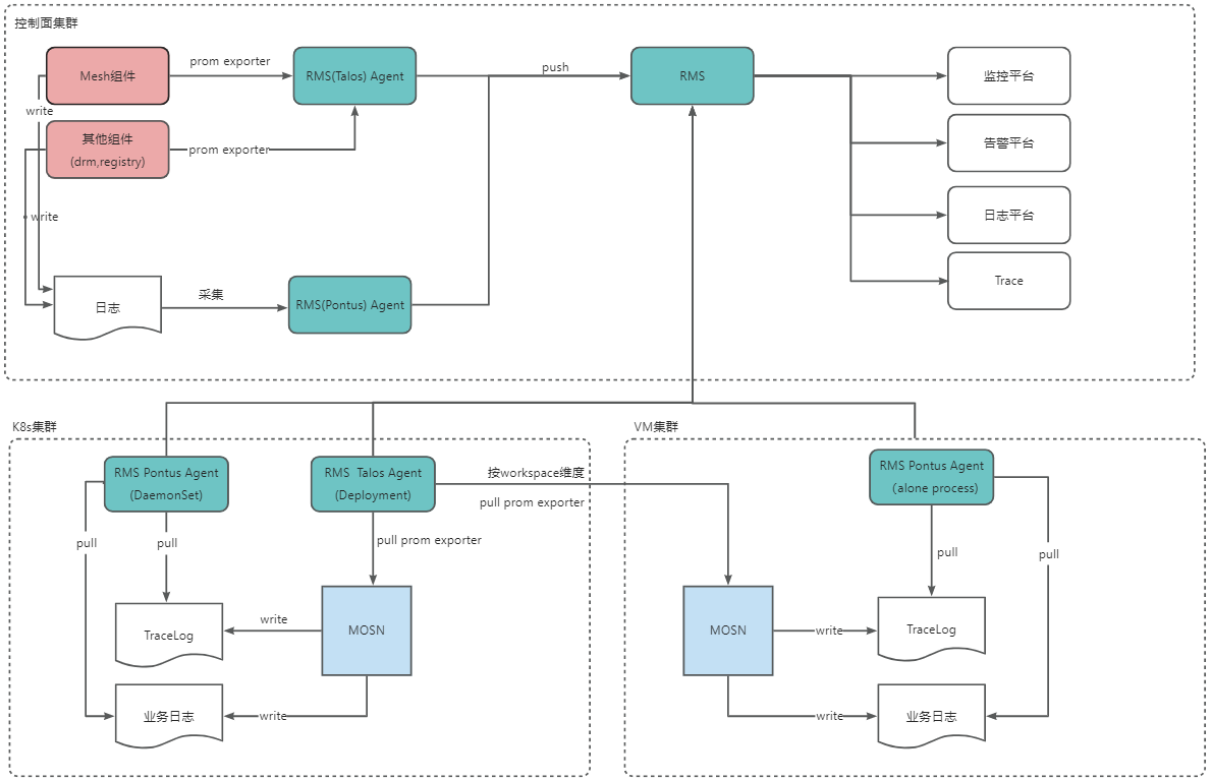
   将构建的调用链路进行可视化呈现，可以帮助开发人员快速定位服务调用的问题，例如延迟、错误等。

Mesh 自身支持多种 SOFA trace、SKyWalking、Zipkin 的链路追踪工具，同样也提供定制化版本的链路追踪开发。

三方平台接入

RMS（Real-time Monitoring Service，简称 RMS） 是蚂蚁集团的一款具有可视化监测能力的金融级监控产品实时监控服务。基于日志、指标、链路等海量数据进行多维聚合，向用户提供业务监控、应用监控、云原生监控、基础资源监控、日志查询分析、分布式链路等多角度的可视化监测功能，有丰富的可视化大盘，并提供了告警订阅功能。该服务可以帮助运维、研发、SRE （Site Reliability Engineer）等快速地发现问题、定位问题、分析问题、解决问题，为线上系统可用率提供有效保障。

如下展示的是 Service Mesh 对接 RMS 系统的概要：



- Talos Agent 指标收集器：负责收集 prometheus 指标数据，转换为 RMS 数据指标进行上报。
- DaemonSet Agent（又称 Pontus-Agent）日志采集代理：负责所有的数据的最终采集，包括原始日志拉取和指标型数据采集。