SOFAStack

服务网格

技术白皮书

产品版本：AntStack Plus 1.11.0

文档版本：20230314

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}`

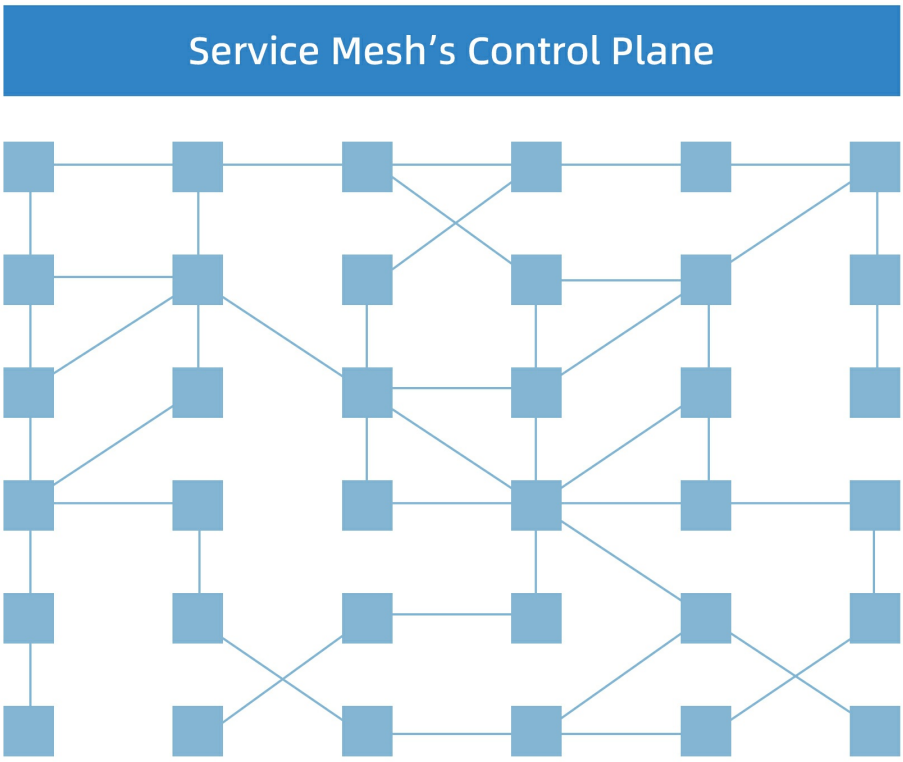
目录

1.什么是服务网格

1.1. 产品简介

ServiceMesh 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。

可以将它比作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

ServiceMesh 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，微服务体系的玩法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，这就带来了一系列的问题：

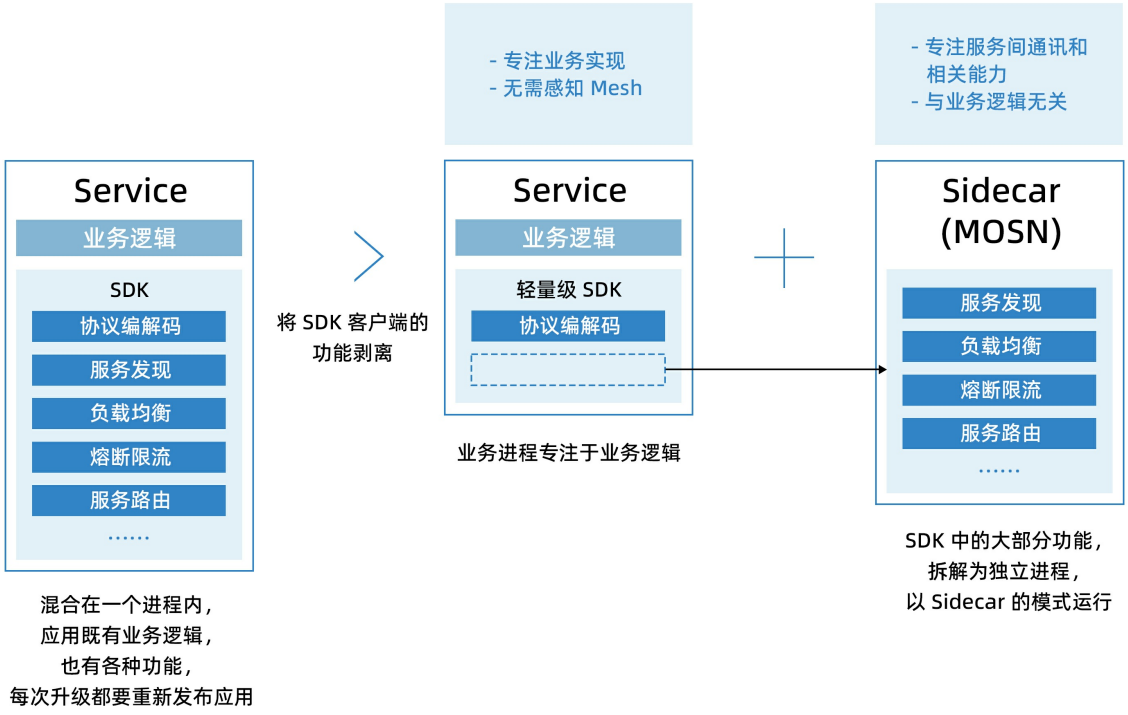
- 升级成本高
  - 每次升级都需要业务应用修改 SDK 版本号，重新发布。
  - 在业务飞速往前跑的时候，是不太愿意停下来做这些和自身业务目标不太相关的事情的。
- 版本碎片化严重

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就需要在代码中兼容各种各样的老版本逻辑，戴着『枷锁』前行，无法实现快速迭代。

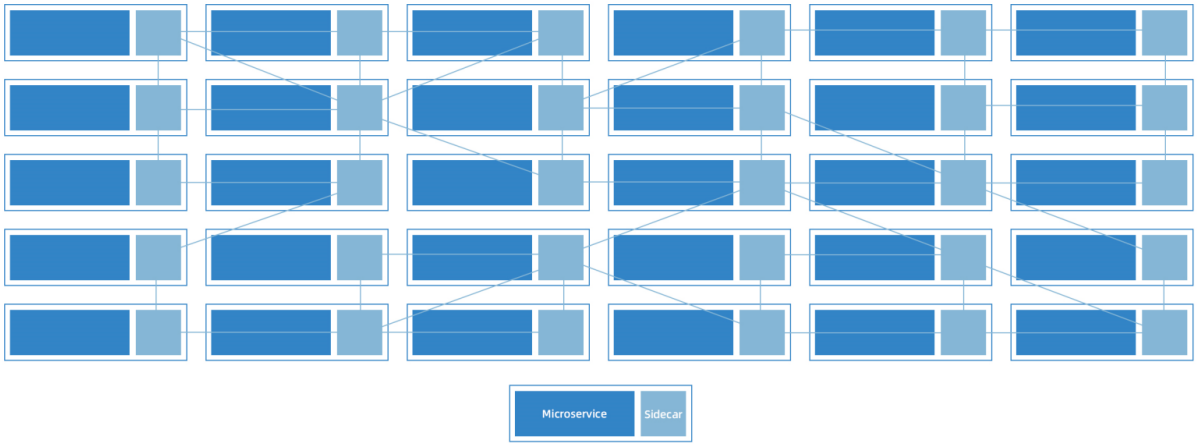
有了 Service Mesh 之后，我们就可以把 SDK 中的大部分能力从应用中剥离出来，拆解为独立进程，以 Sidecar 的模式部署。通过将服务治理能力下沉到基础设施，可以让业务更加专注于业务逻辑，中间件团队则更加专注于各种通用能力，真正实现独立演进，透明升级，提升整体效率。



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。

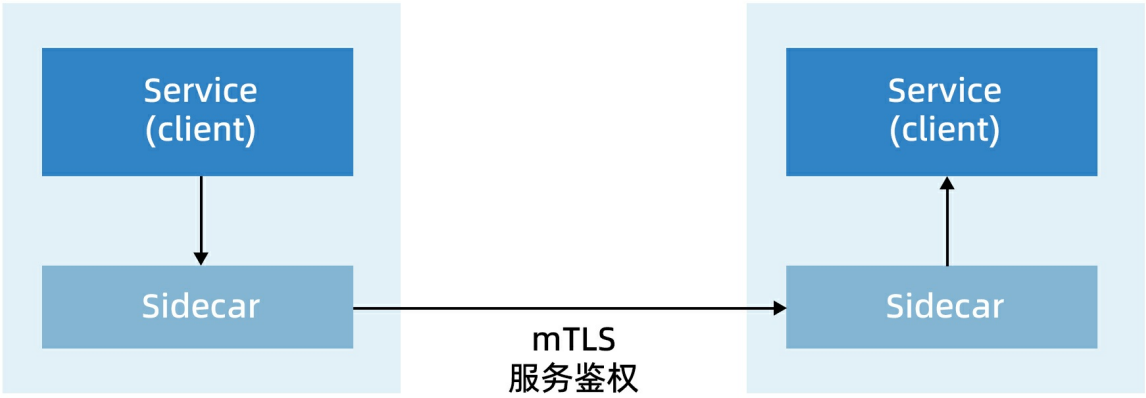
有了之后，通过将主体的服务治理能力下沉到基础设施，多语言的支持就轻松很多了，只需要提供一个非常轻量的 SDK、甚至很多情况都不需要一个单独的 SDK，就可以方便地实现多语言、多协议的统一流量管控、监控等治理需求。



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 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。



1.3. 发展现状

正因为带来了上述种种的好处，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 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 来自希腊语，英文是 Sail，翻译为中文是“启航”。

Istio 中的数据面分类

- Envoy（默认）

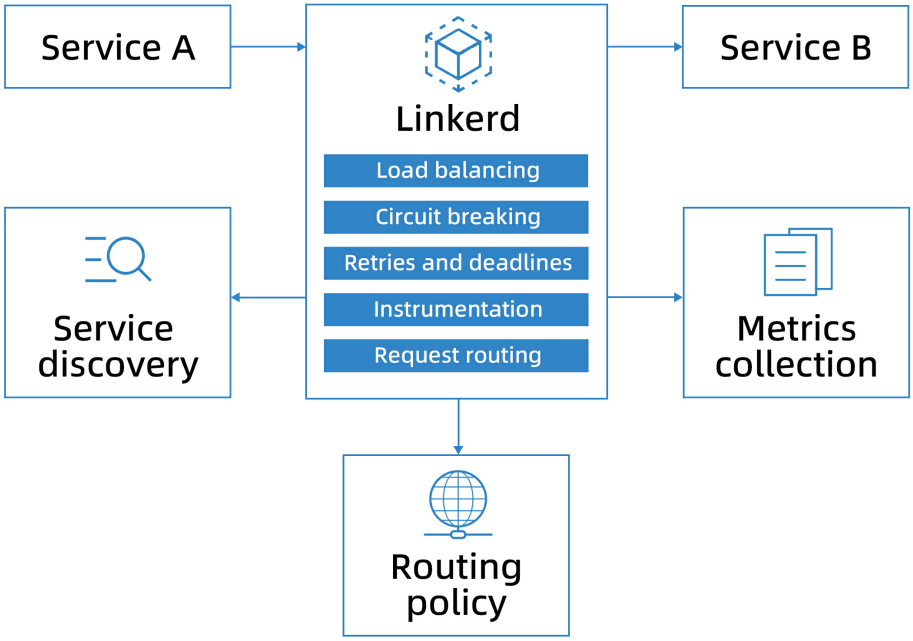
  官方地址：https://www.envoyproxy.io

  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。

- Linkerd（可选）

  官方地址：https://linkerd.io/

  linker·d 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。



Istio 中的数据面组成

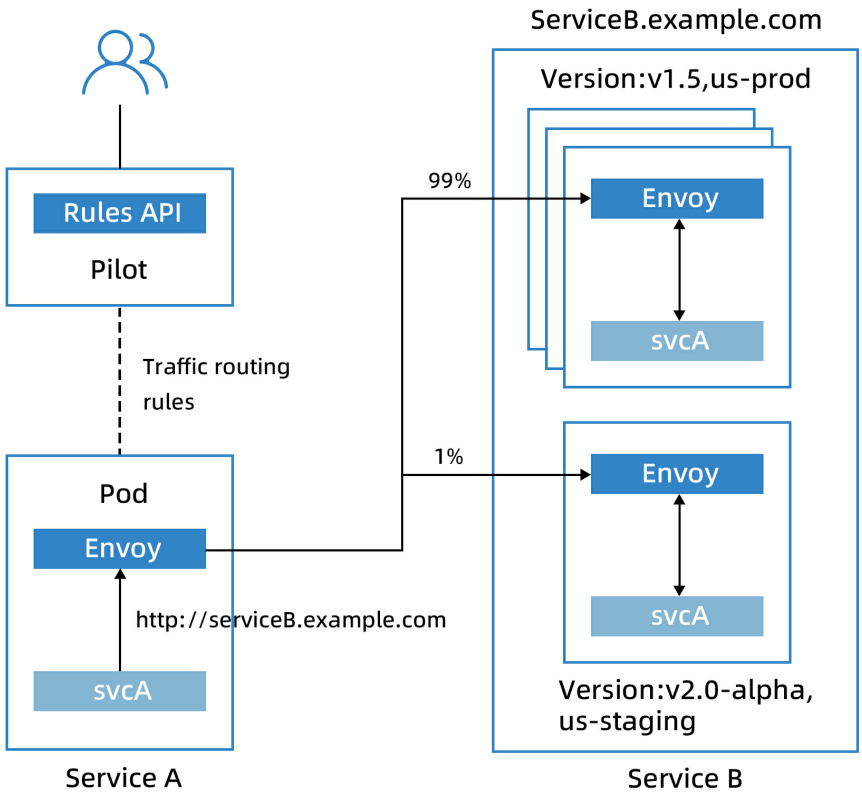
Istio 数据面主要包括三部分：Pilot、Citadel 和 Galley。

Pilot

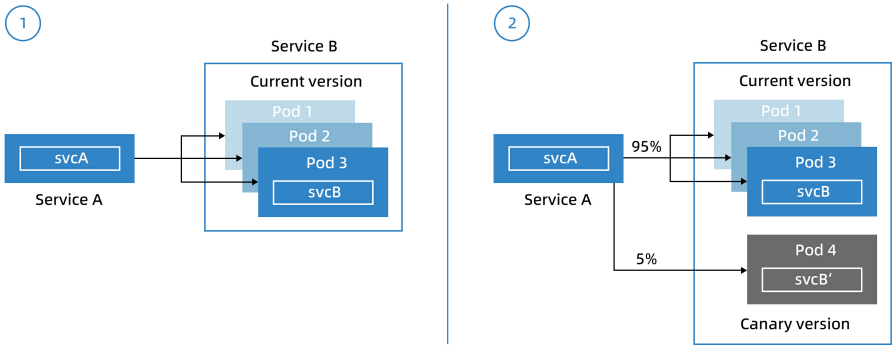
Istio 最核心的功能是流量管理，前面我们看到的数据面板，由 Envoy 组成的服务网格，将整个服务间通讯和入口/出口请求都承载于其上。

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。



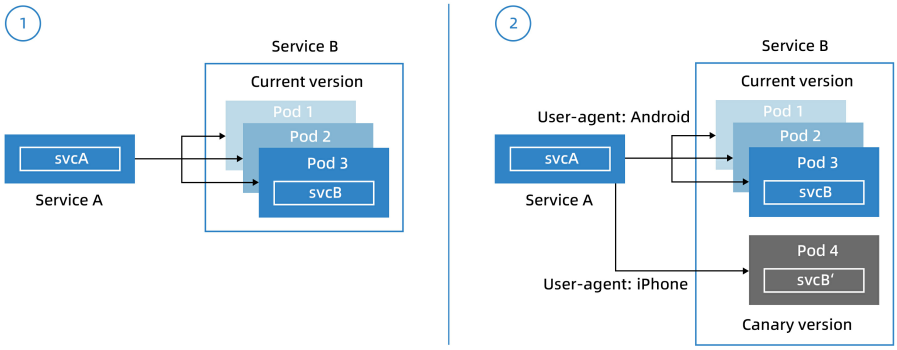
举个例子，假定我们原有服务B，部署在 Pod1/2/3 上，现在我们部署一个新版本在 Pod4 在，希望实现切 5% 的流量到新版本。



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方案目前主要存在两个问题。

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。

离生产级还有一定距离

目前 Pilot 组件存在如下问题：

- 无法支撑海量数据。
- 每次变化都会触发全量推送，性能较差。

2.产品架构

系统架构

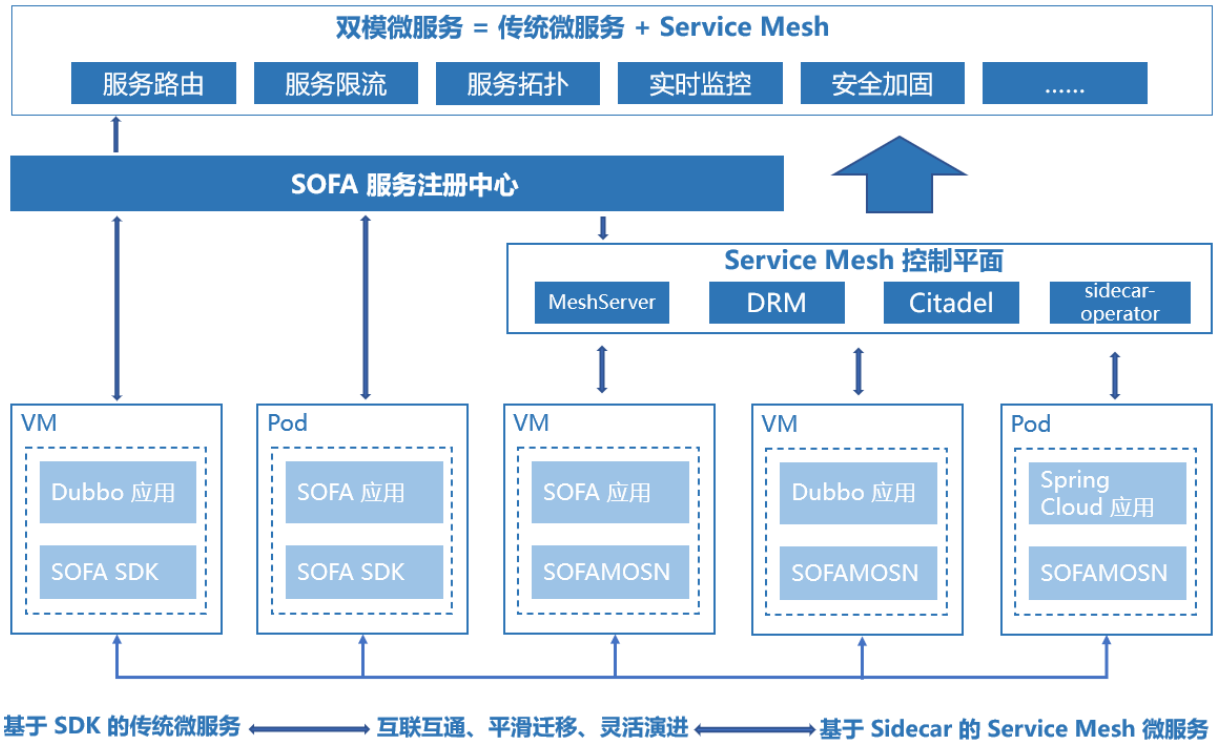
我们的服务网格产品名是 SOFAStack 双模微服务平台，这里的双模微服务是指传统微服务和 Service Mesh 结合，即基于 SDK 的传统微服务可以和基于 Sidecar 的 Service Mesh 微服务实现下列目标：

- 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，我们引入了 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，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。

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。



3.性能指标

3.1. 单核性能（Sidecar 场景）

测试环境

机器信息

机器	OS	CPU
11.**.**.224	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel（R） Xeon（R） CPU E5-2640 v3 @ 2.60GHz
11.**.**.110	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel（R） Xeon（R） CPU E5-2430 0 @ 2.20GHz
bolt client	client 为压力平台，有 5 台压力机，共计与client SOFAMosn 之间会建立 500 条链接	-
http1 client（10.**.**.5）	ApacheBench/2.3	-n 2000000 -c 500 -k
http2 client（10.**.**.5）	nghttp.h2load	-n1000000 -c5 -m100 -t4

部署结构

压测模式	部署结构
串联client --> SOFAMosn（11.**.**.224） --> SOFAMosn（11.**.**.110） --> server（11.**.**.110）	串联client --> SOFAMosn（11.**.**.224） --> SOFAMosn（11.**.**.110） --> server（11.**.**.110）

网络延时

节点	PING
client --> SOFAMosn（11.**.**.224）	1.356ms
SOFAMosn（11.**.**.224） --> SOFAMosn（11.**.**.110）	0.097 ms

请求模式

请求内容
1K req/resp

7 层代理

场景	QPS	RT(ms)	MEM(K)	CPU(%)
Bolt	16000	15.8	77184	98
Http/1.1	4610	67	47336	90
Http/2	5219	81	31244	74

3.2. 多核性能（Gateway 场景）

测试环境

机器信息

机器	OS	CPU
11.**.**.224	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel（R） Xeon（R） CPU E5-2640 v3 @ 2.60GHz
11.**.**.110	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel（R） Xeon（R） CPU E5-2430 0 @ 2.20GHz
bolt client	client为压力平台，有5台压力机，共计与client SOFAMosn之间会建立500条链接	
http1 client（10.**.**.5）	ApacheBench/2.3	-n 2000000 -c 500 -k

http2 client（10.**.**.5）	nghttp.h2load	-n1000000 -c5 -m100 -t4

部署结构

压测模式	部署结构
直连	client --> SOFAMosn（11.**.**.224） --> server（11.**.**.110）

网络延时

节点	PING
client --> SOFAMosn（11.**.**.224）	1.356ms
SOFAMosn（11.**.**.224） --> SOFAMosn（11.**.**.110）	0.097 ms

请求模式

请求内容
1K req/resp

7 层代理

场景	QPS	RT(ms)	MEM(K)	CPU(%)
Bolt	45000	23.4	544732	380
Http/1.1	21584	23	42768	380
Http/2	8180	51.7	173180	300

3.3. 长连接网关

测试环境

机器信息

机器	OS	CPU
11.**.**.224	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
11.**.**.110	3.10.0-327.ali2010.rc7.alios7.x86_64	Intel(R) Xeon(R) CPU E5-2430 0 @ 2.20GHz

部署结构

压测模式	部署结构
直连	client --> SOFAMosn（11.**.**.224） --> server（11.**.**.110）

网络延时

节点	PING
client --> SOFAMosn（11.**.**.224）	1.356ms
SOFAMosn（11.**.**.224） --> SOFAMosn（11.**.**.110）	0.097 ms

请求模式

链接数	请求内容
2 台压力机，每台 5 万连接 + 500 QPS，共计10 万连接 + 1000 QPS	1K req/resp

长连接网关

场景	QPS	MEM(g)	CPU(%)	goroutine
RWLoop + goroutine	1000	3.3	60	200028

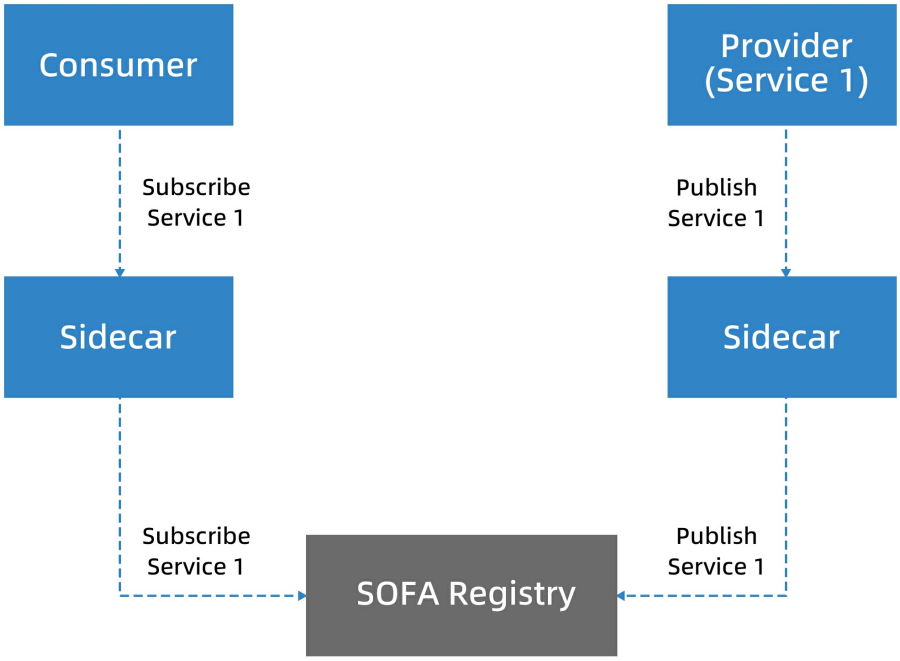
Raw epoll	1000	2.5	18	28

4.功能原理

4.1. 大规模场景下的服务发现

要在蚂蚁集团落地，首先一个需要考虑的就是如何支撑双十一这样的大规模场景。前面已经提到，目前 Pilot 本身在集群容量上比较有限，无法支撑海量数据，同时每次变化都会触发全量推送，无法应对大规模场景下的服务发现。

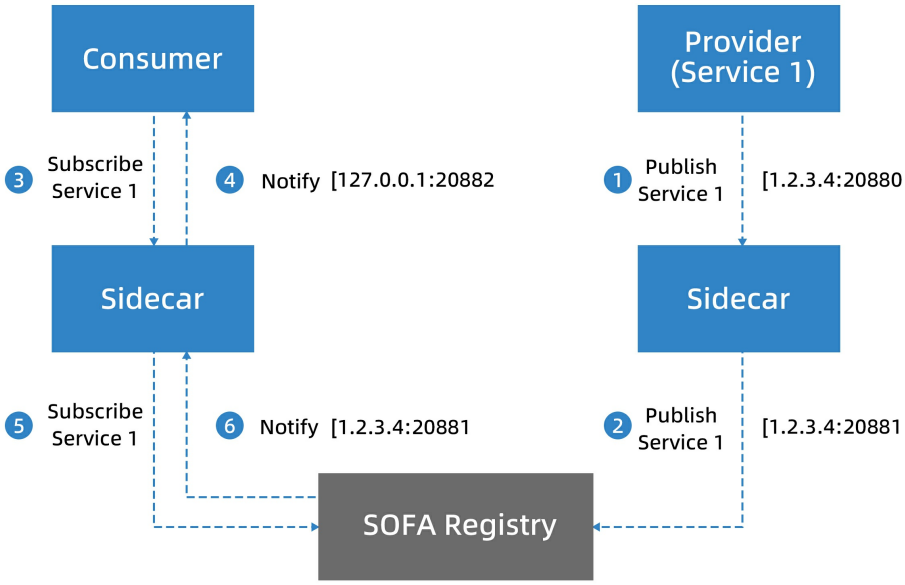
所以，我们的方案是保留独立的 SOFA 服务注册中心来支持千万级的服务实例信息和秒级推送，业务应用通过直连来实现服务注册和发现。



4.2. 流量劫持

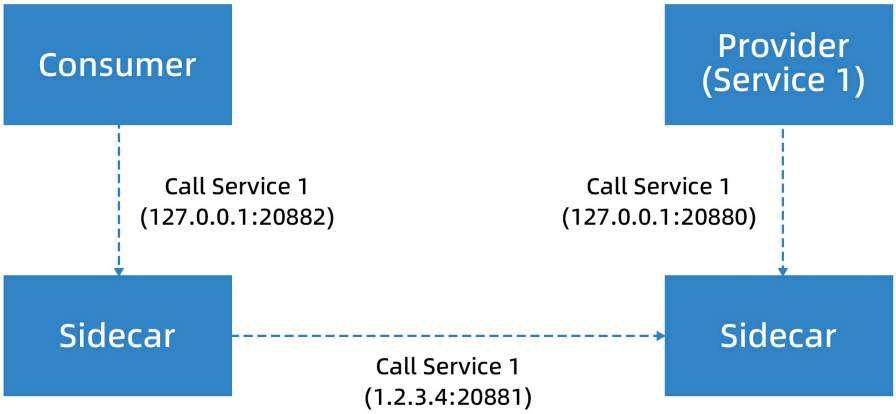
ServiceMesh 中另一个重要的话题就是如何实现流量劫持：使得业务应用的 Inbound 和 Outbound 服务请求都能够经过 Sidecar 处理。

区别于社区的 iptables 等流量劫持方案，我们的方案就显得比较简单直白了，以下图为例：



1. 假设服务端运行在 1.2.3.4 这台机器上，监听 20880 端口，首先服务端会向自己的 Sidecar 发起服务注册请求，告知 Sidecar 需要注册的服务以及 IP + 端口（1.2.3.4:20880）。
2. 服务端的 Sidecar 会向 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 + 端口，不过这里需要注意的是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，这里需要注意的是推送的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 会向 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（1.2.3.4:20881）。

经过上述的服务发现过程，流量劫持就显得非常自然了：



1. 调用端拿到的服务端地址是 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，通过解析请求头，可以得知具体要调用的服务信息，然后获取之前从服务注册中心返回的地址后就可以发起真实的调用（1.2.3.4:20881）。
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）。

可能会有人问，为何不采用 iptables 的方案呢？主要的原因是一方面 iptables 在规则配置较多时，性能下滑严重，另一个更为重要的方面是它的管控性和可观测性不好，出了问题比较难排查。

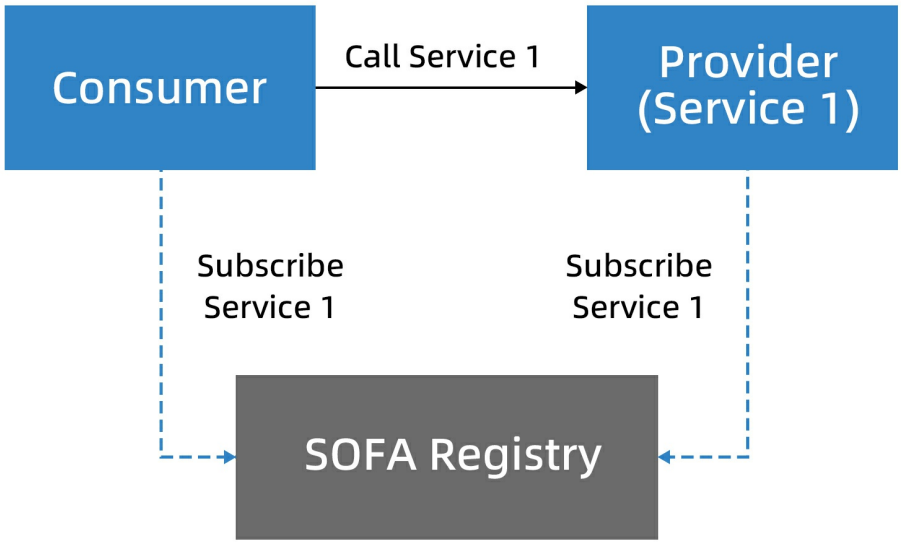
4.3. 平滑迁移

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。

得益于独立的服务注册中心，我们的平滑迁移方案也非常简单直白：

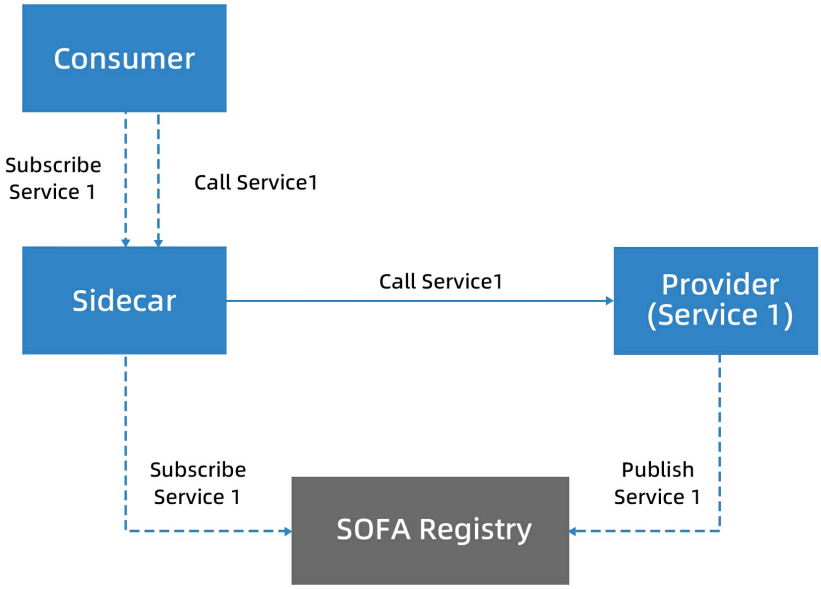
1. 初始状态。

   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。



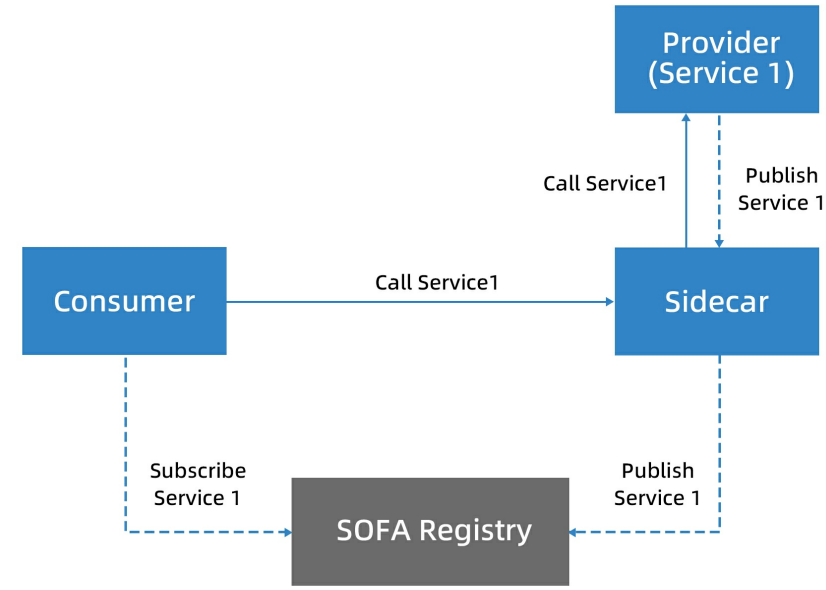
2. 透明迁移调用方。

   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。

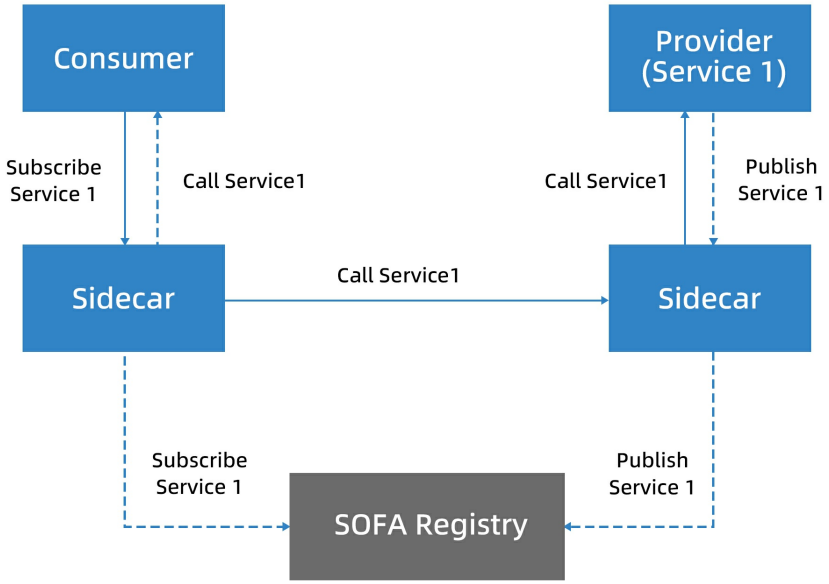


3. 透明迁移服务方。

   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。

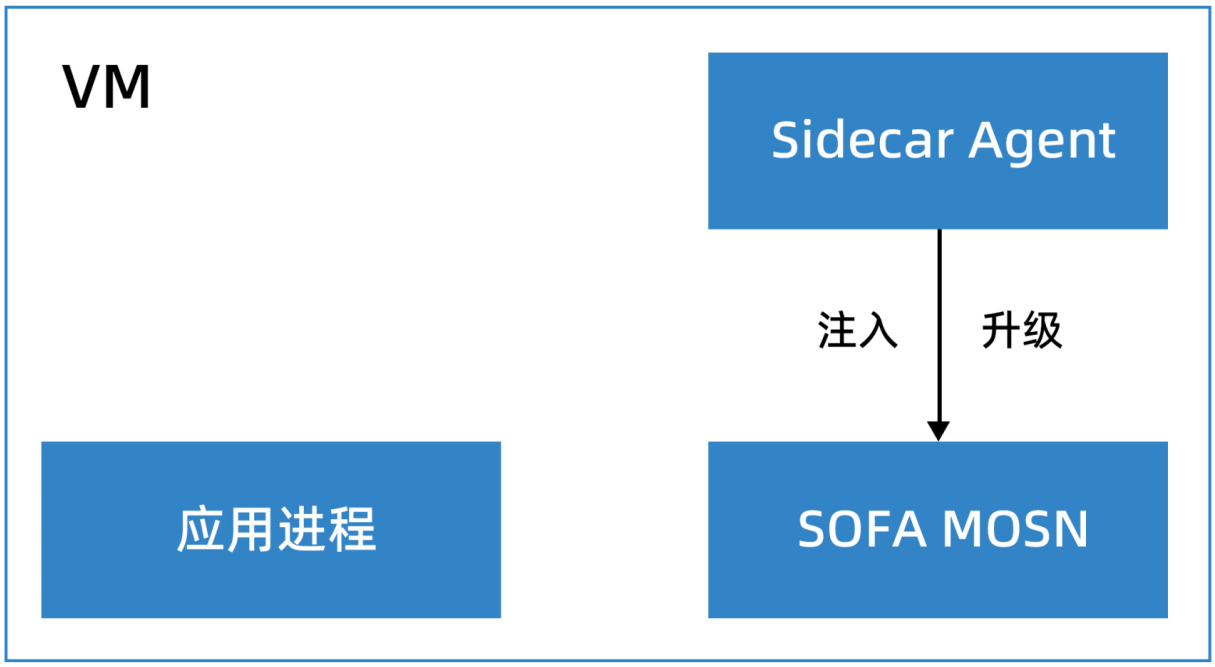


4.4. 多协议支持

考虑到目前大部分用户的使用场景，除了 SOFA 应用，我们同时也支持 Dubbo 和 Spring Cloud 应用接入 SOFAStack 双模微服务平台，提供统一的服务治理。多协议支持采用通用的 x-protocol，未来也可以方便地支持更多协议。

4.5. 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跑在 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。



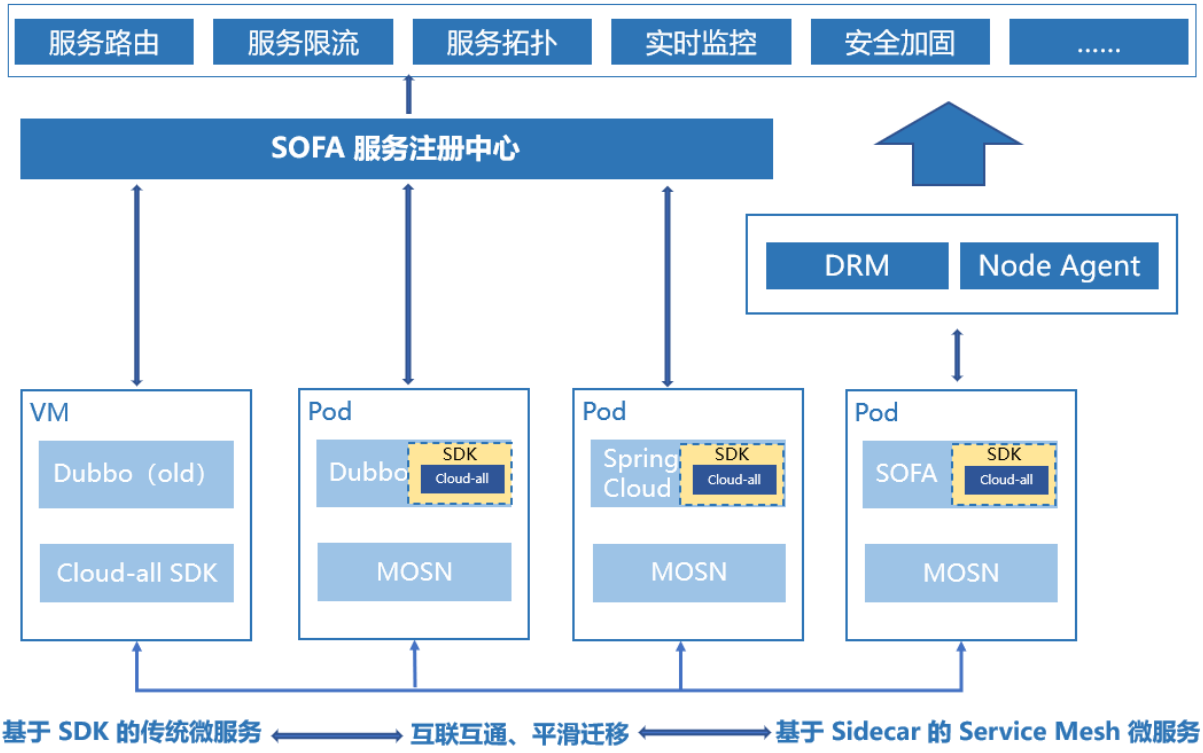
4.6. 从配置了解 Mesh 工作原理

MOSN 形态现状

目前 MOSN 属于数据面的产品，以 Sidecar 的模式和应用部署在同一个 Pod 或者在虚拟机中，属于独立进程。MOSN 最早支持基于轻量 SDK + Mesh 的方式接管网格流量。目前主要支持 3 种流量劫持方式：

- 轻量级 SDK 端口欺骗的方式接管流量。
- 基于修改配置 IP+Port -> 127.0.0.1 的方式接管流量。
- 基于透明劫持 iptables 接管流量。

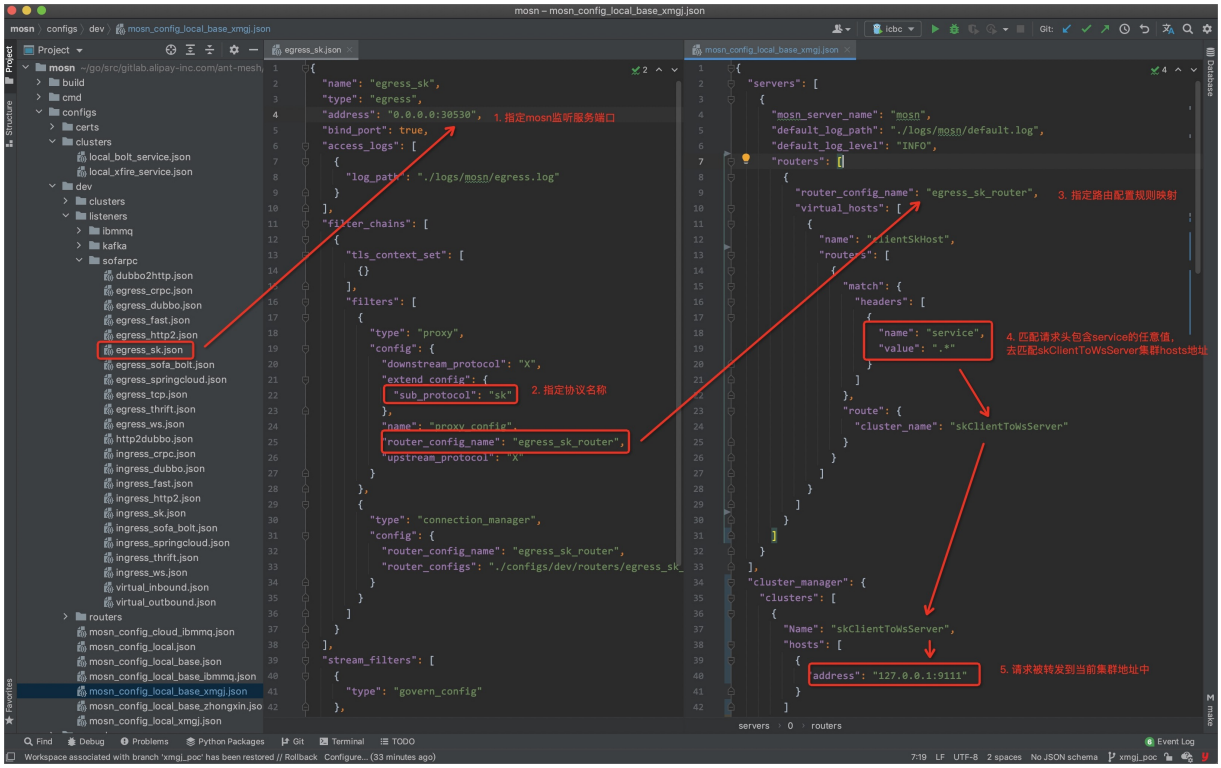
目前自研 MOSN 已经支持国内主流框架，比如蚂蚁的 SOFA RPC、社区 Dubbo 和 Spring Cloud 技术栈 。服务发现继续使用经典的注册中心机制，内部注册中心产品是 SOFARegistry。



数据面 MOSN 产品目前支持虚拟机和容器场景部署，应用接入 MOSN。作为服务提供方，MOSN 会将自身端口号写入注册中心。作为消费方，MOSN 会从注册中心订阅服务，也就是拿到了服务方 MOSN 的 IP+Port，然后完成服务通信。

MOSN 配置关联关系

为了帮助您梳理清楚 MOSN 的一串配置关系，这里从链路关系逐一讲解。以静态配置为例，目前配置主要分为几大类，如 listeners、routers 和 clusters 等。配置截图如下：



- listeners 主要是 TCP、UDP 等端口监听配置，包括制定端口和协议绑定关系等。
  - RPC 类别，主要支持 request-response 等通信模型，比如 Dubbo、SOFA、Spring Cloud 等私有协议。
  - MQ 场景，处理消息框架通信场景。
- routers 主要保存和协议相关的路由配置映射，匹配 Header 中 Service 信息到具体的 Cluster。
- clusters 主要保存 Cluster 中真实的服务地址列表。针对 RPC 场景，一般 Cluster 的 name 是接口，根据接口对应地址列表的映射关系。

上图中，箭头的数字代表流程进度：

1. 客户端 MOSN 监听 30530 端口，等待客户端应用 App 调用。
2. 在 MOSN 配置中，我们指定 30530 端口给私有协议 SK 使用，因此协议叫做 SK。
3. MOSN 静态配置中指定路由配置映射关系 name 为 `egress_sk_router`，一般规则为 `egress_协议名_router`，如果是服务提供方，一般规则为 `ingress_协议名_router`。可以理解为这个配置映射关系是数组，里面会存储很多 Header 中包含 Service 的不同值（一般值是接口名），并且根据不同值映射到具体的 Cluster。值 `.*` 是特殊值，代表只要 Header 中有 Service 这个 Key，则直接转发到目标 Cluster 对应的名称。
4. 请求 Header 包含 Service 任意值，转发到 skClientToWsServer 这个集群。
5. 集群 skClientToWsServer 中包含的地址列表都会被 MOSN 选择调用。

如果理解静态配置和寻址关联关系，MOSN 在运行时，也会动态构造这些 Cluster 信息，因为路由都是在客户端处理的。当服务提供方上线时，MOSN 客户端会收到地址列表推送，MOSN 会去根据推送的 dataId（一般是接口名）去找到 Cluster，并且把地址列表动态更新。

值得一提的是，listeners 中 SOFARPC 目录包含较多私有协议扩展，MOSN 启动的时候默认都会加载并且开启对应端口监听，输出到专有云一般无用的协议，我们会通过 Dockerfile 文件删除无用的协议。

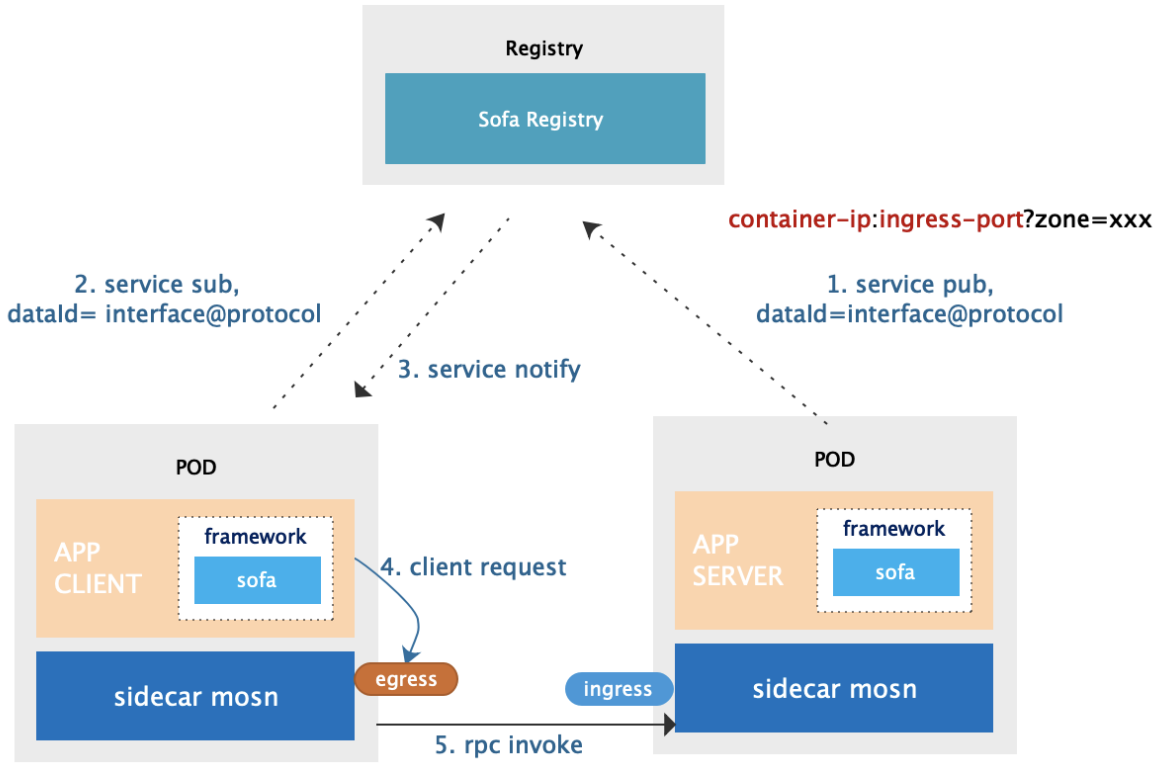
MOSN 调用关系

为了通俗易懂的理解 MOSN 调用链路，需要彻底弄懂 ingress 和 egress 的概念和区分。

站在 Sidecar 视角去理解 egress 和 ingress，比如 MOSN 接收流量后转发给远端还是本地：

- egress：出口流量，MOSN 接收流量需要转发给远端的服务提供方。
- ingress：入口流量，MOSN 接收流量无脑转发给本地的服务提供方

MOSN 调用关系图示如下：



以上入为例，MOSN 的调用链路关系如下：

1. 服务提供方 MOSN 会将自身 ingress 的协议端口写入到注册中心。
2. 调用方 MOSN 会从注册中心订阅地址列表。

   第一次订阅也会返回全量地址列表，端口号是服务方 ingress 绑定的端口号。
3. 注册中心会实时推送地址列表变更到客户端（每次都是全量地址列表）。
4. 客户端 App 发起业务 RPC 请求，请求会被 SDK 转发到本地客户端 MOSN 的 egress 端口号上。
5. 客户端 MOSN 将 RPC 请求通过网络转发，将流量通过负载均衡转发到某一台服务方 MOSN 的 ingress 端口处理。
6. 最终服务到了服务端 ingress listener，会直接转发给本地 App Server 应用。响应返回时，会根据原来的 TCP 链路反向转发。

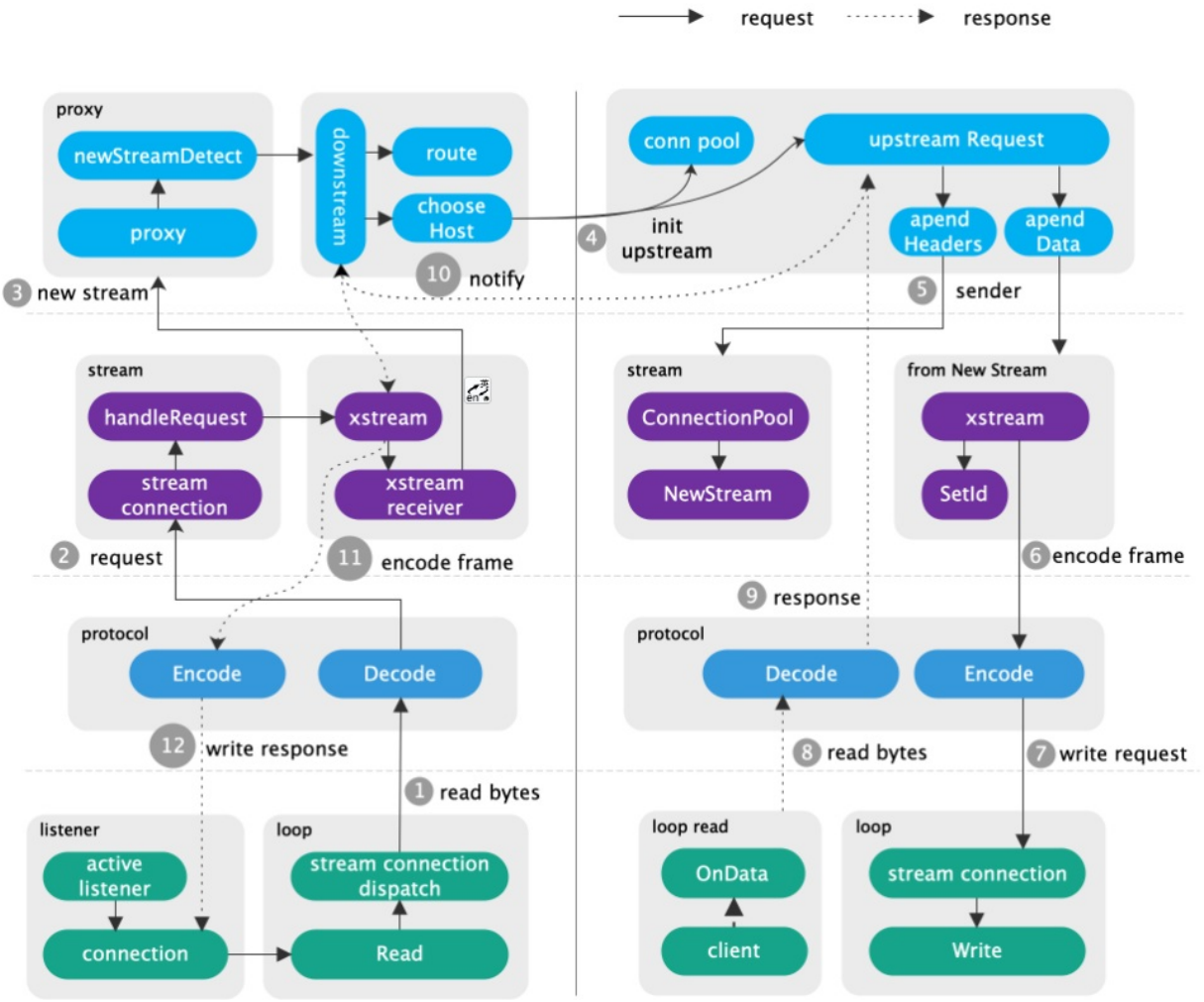
4.7. Mesh 核心转发流程

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。

   这个 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 byte 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。

5.技术解析

5.1. 服务网格落地

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

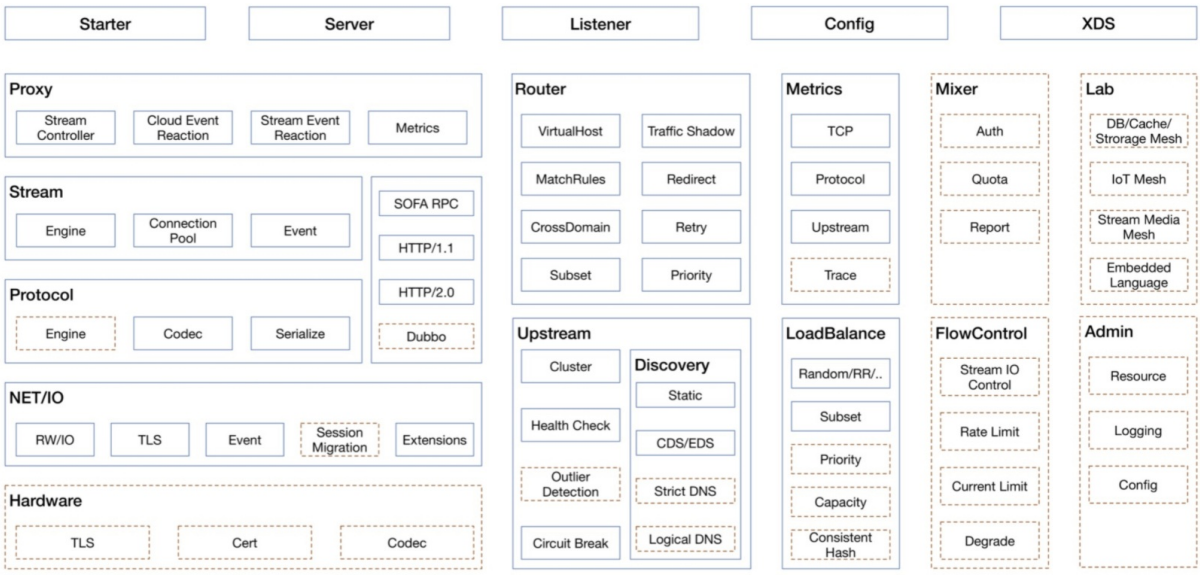
基础能力建设

SOFAMosn 能力大图

SOFAMosn 主要包括了下述能力：

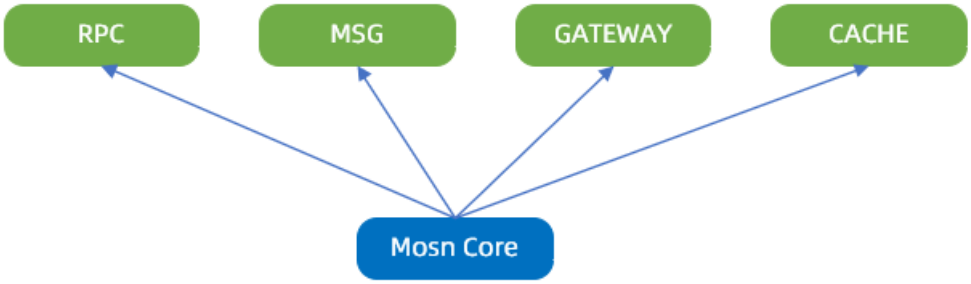
- 网络代理具备的基础能力。
- XDS（Extended Discovery Service） 等云原生能力。

SOFAMosn 主要模块图



业务支持

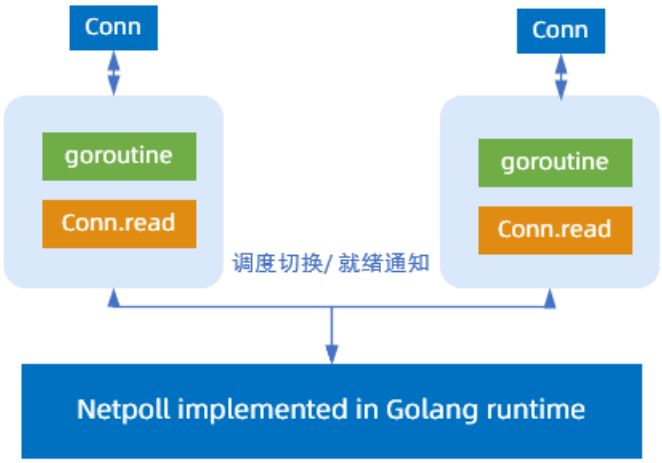
SOFAMosn 作为底层的高性能安全网络代理，支撑的业务场景包括：RPC、MSG、GATEWAY 等。



IO 模型

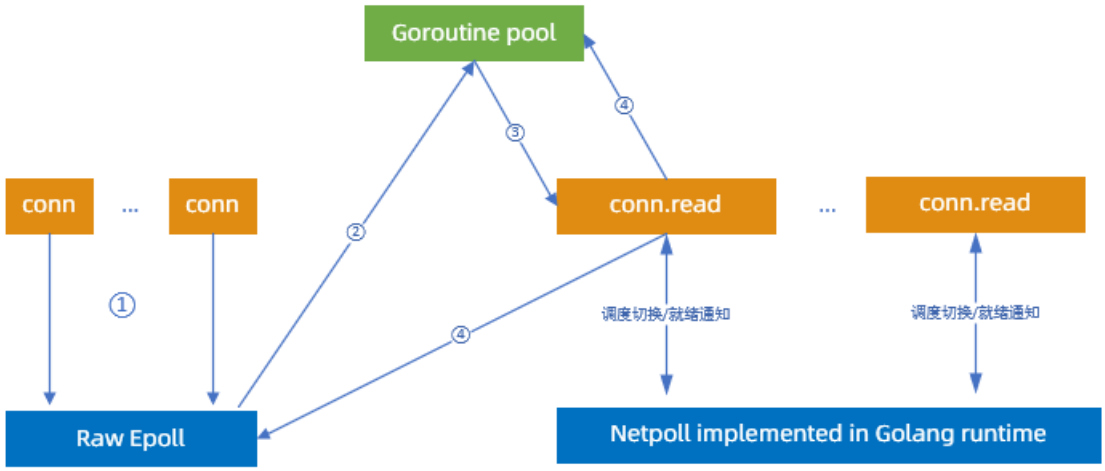
SOFAMosn 支持两种 IO 模型：

- **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：

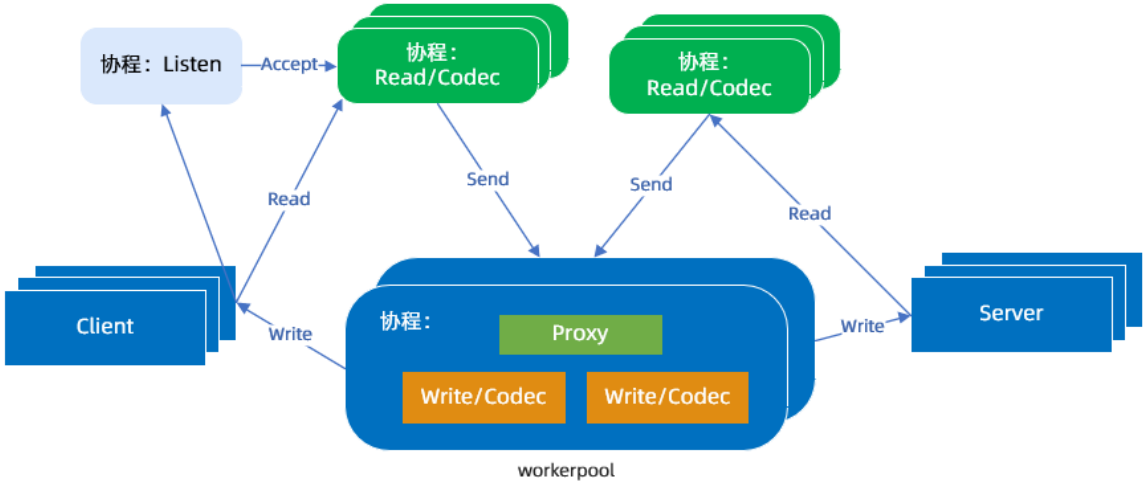
   向 Epoll 注册 oneshot 可读事件监听。此时不允许有协程调用 `conn.read`，以免与 `runtime Netpoll` 冲突。

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 重新添加到 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

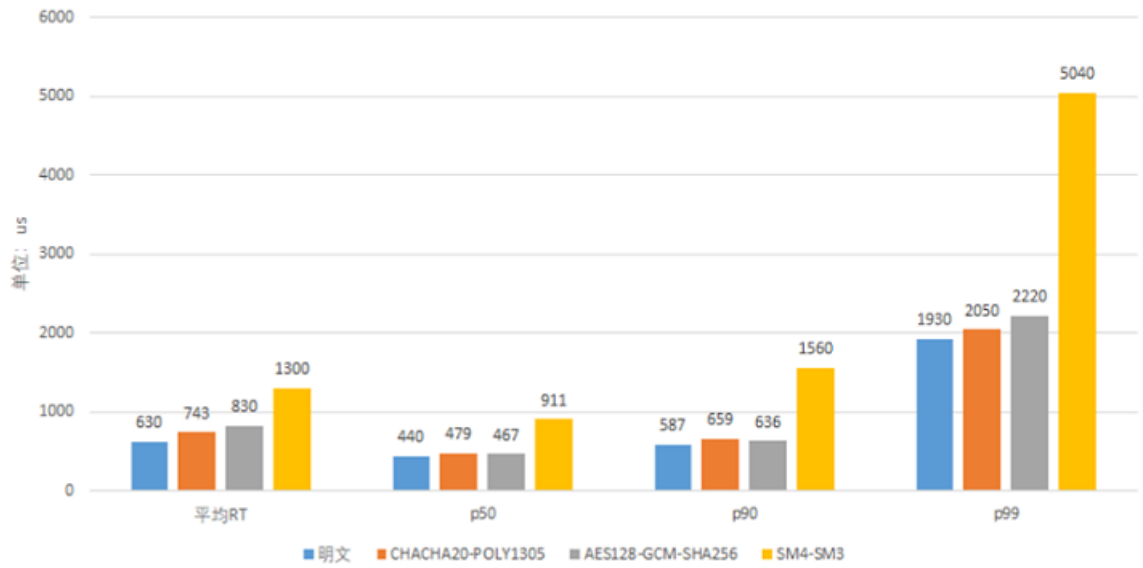
作为金融科技公司，资金安全是最重要的一环，链路加密又是其中最基础的能力。在 TLS 安全链路上，蚂蚁团队进行了大量的调研测试。测试结果显示：

- 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。

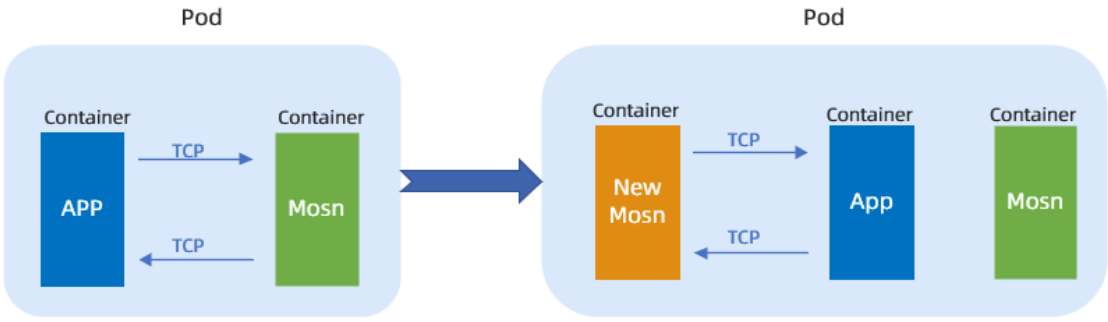


平滑升级能力

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。

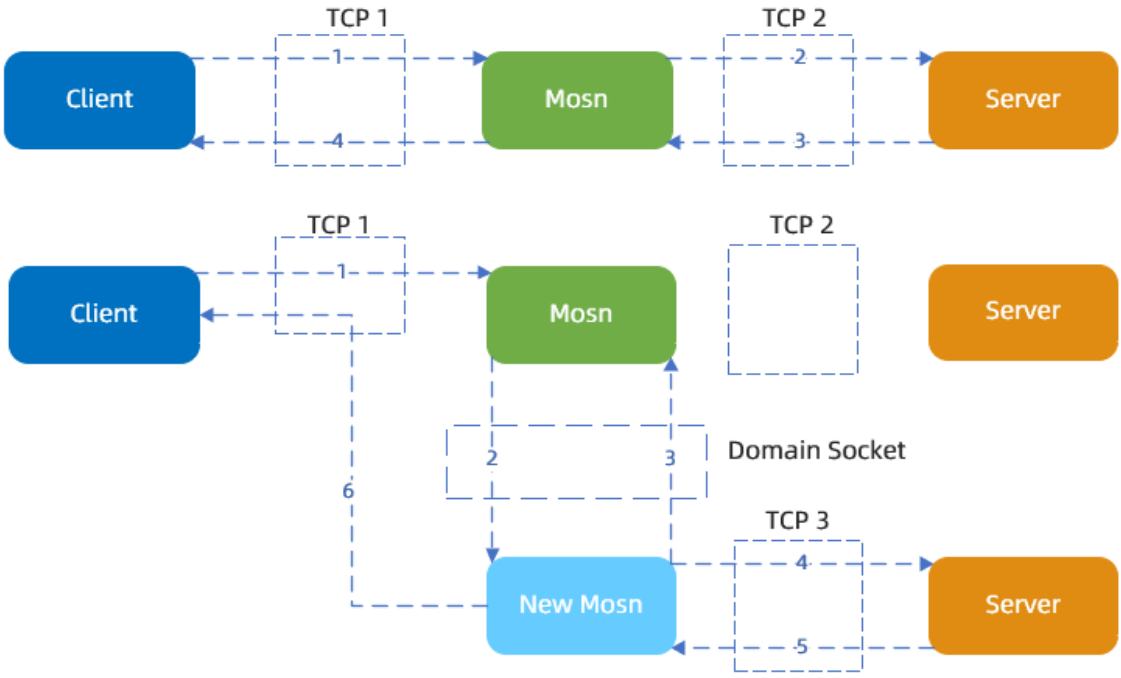


i. 先注入一个新的 SOFAMosn。

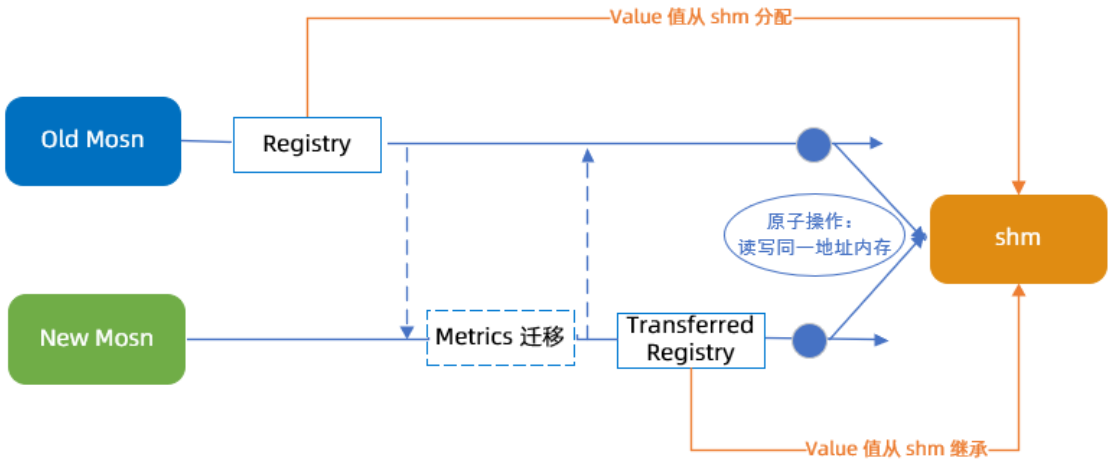
ii. 通过共享卷的 UnixSocket 去检查是否存在老的 SOFAMosn。

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。



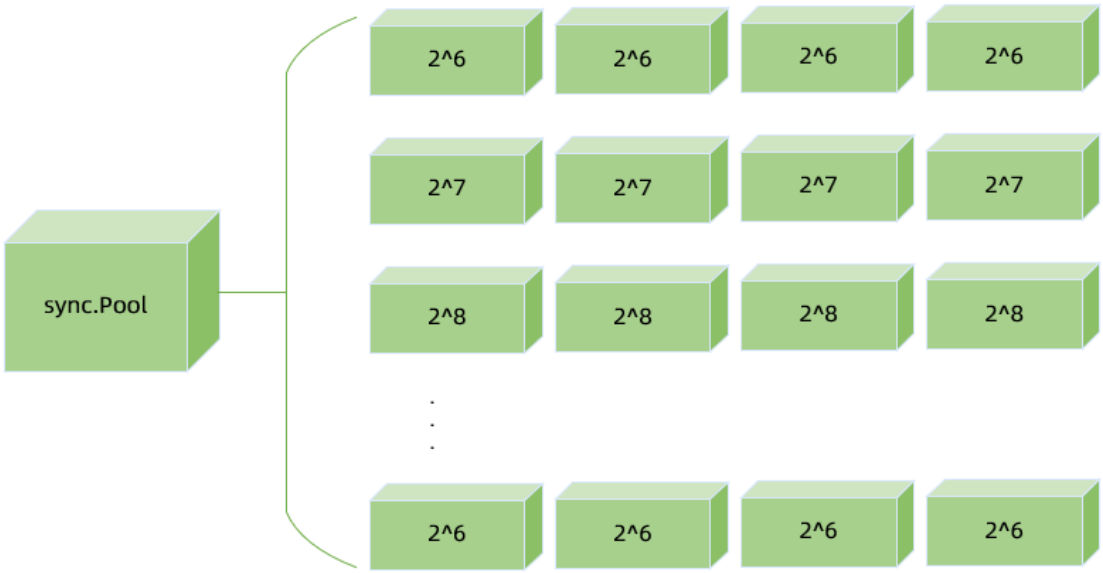
- **SOFAMosn 的 Metric 迁移**：蚂蚁团队使用了共享内存来共享新老进程的 Metric 数据，保证在迁移的过程中 Metric 数据也是正确的。



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%。

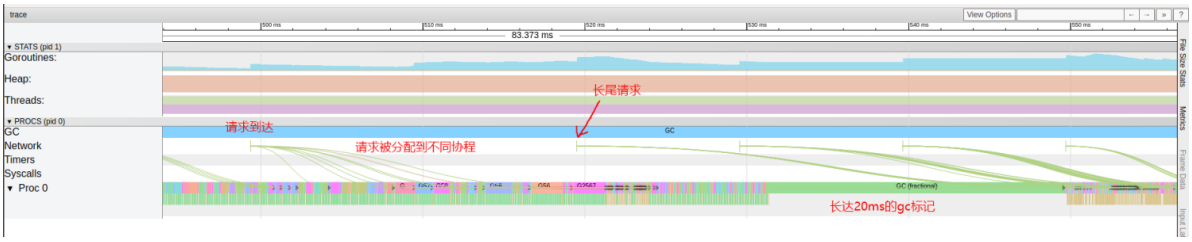
SOFAMosn 所做的大量核心优化和下沉的 Route Cache 等业务逻辑优化，更带来了架构的红利。

Go 版本选择

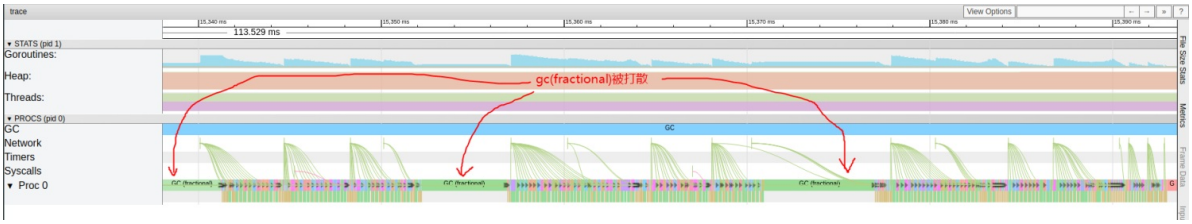
版本的升级都需要做一系列测试，新版本并不都最适合目标场景。该项目最开始使用的版本为 Go 1.9.2，在经过一年迭代之后，蚂蚁团队开始调研当时的最新版 Go 1.12.6，测试验证了新版很多好的优化，也修改了内存回收的默认策略，以便更好地满足项目需求。

- 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  +               }
```

5.2. RPC

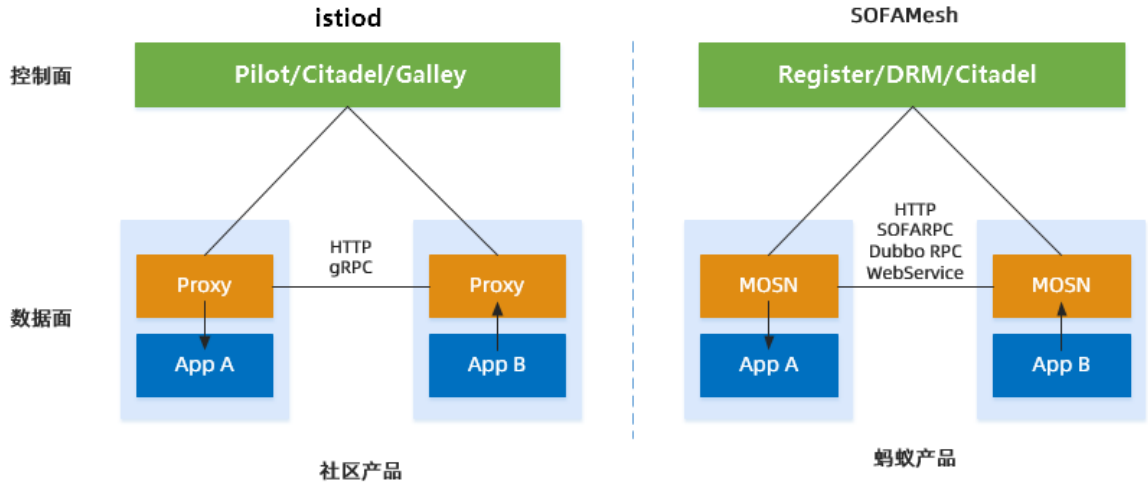
服务网格（Service Mesh）是蚂蚁集团下一代架构的核心，在蚂蚁集团当前的体量下，将现有的 SOA 体系快速演进至 Service Mesh 架构，犹如给奔跑的火车换轮子。本文以 RPC 层面的设计和改造方案为中心，分享蚂蚁集团在双十一大促面临大流量挑战时，核心应用如何将现有的微服务体系平滑过渡到 Service Mesh 架构下，并降低大促成本。

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：

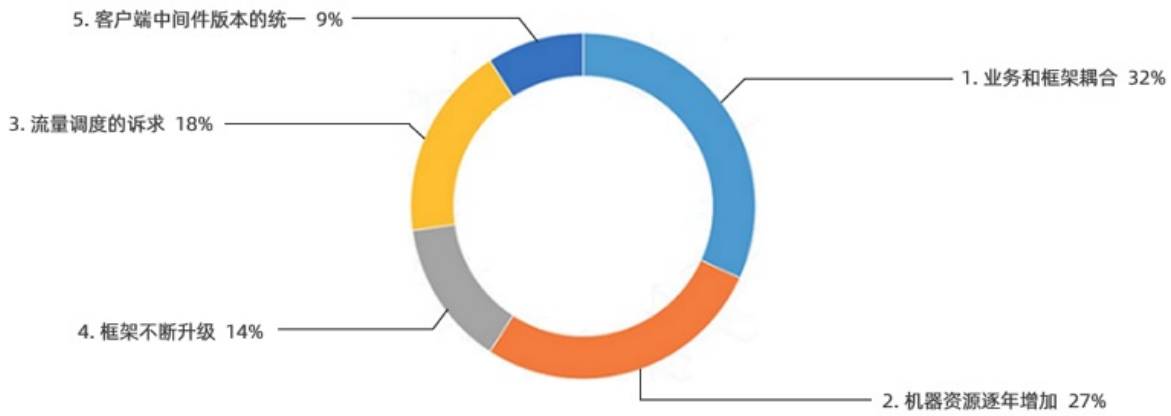
- 基础架构和业务研发耦合问题
- 业务透明的稳定性与高可用性等问题

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。

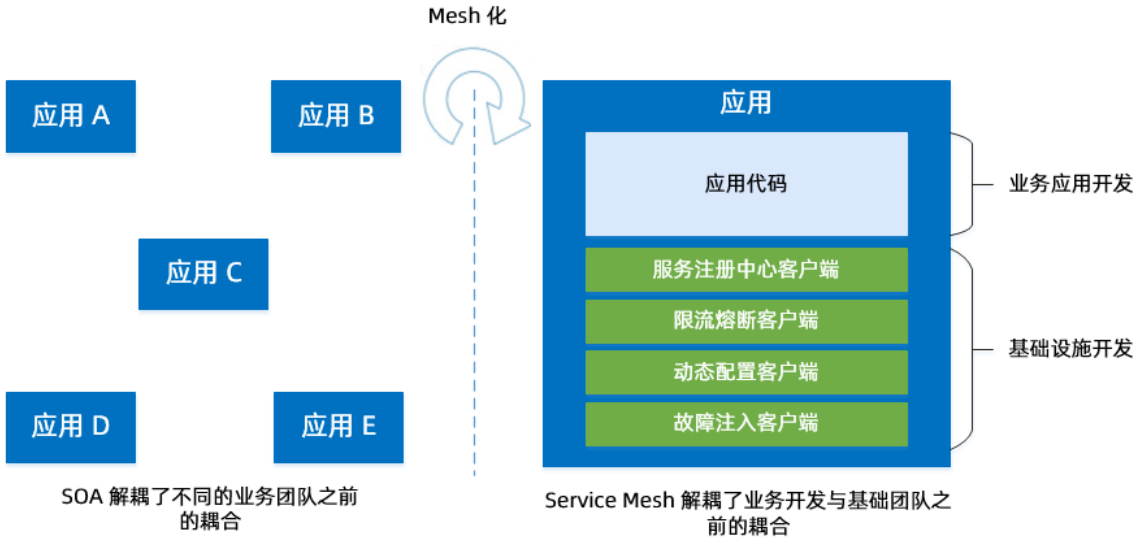
在 SOA 的架构下，负责业务的团队和负责基础设施的团队，合作现状如下：

- **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

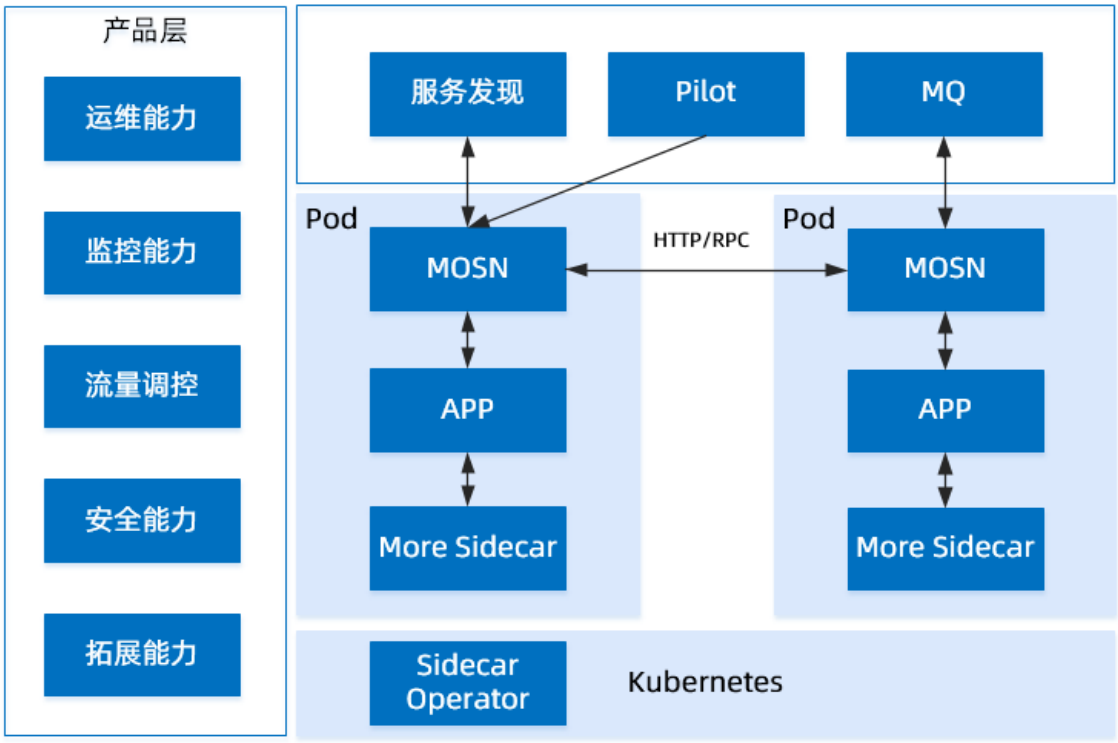


Service Mesh 解决方案

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：

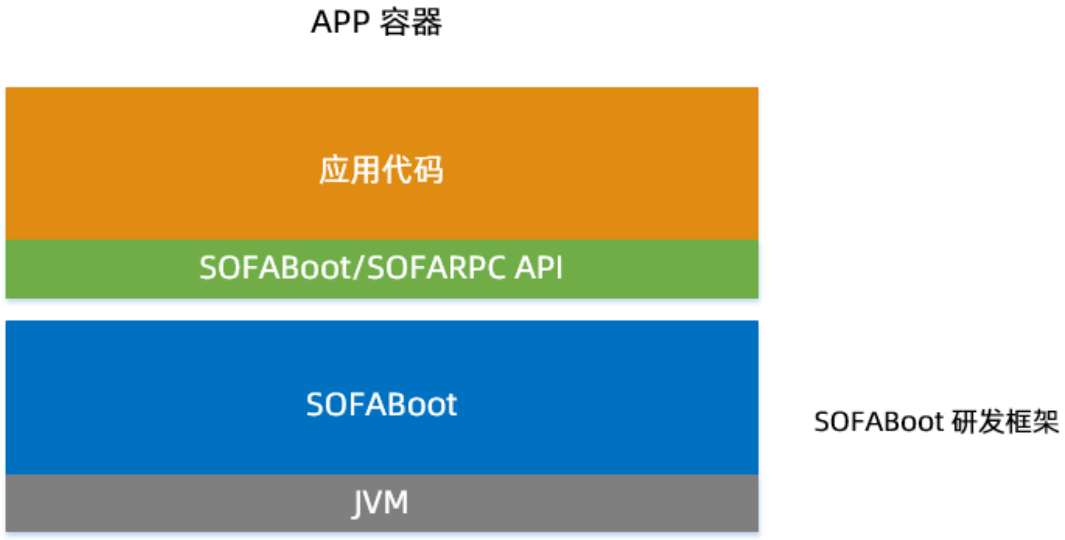
- 运维
- 监控
- 安全

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

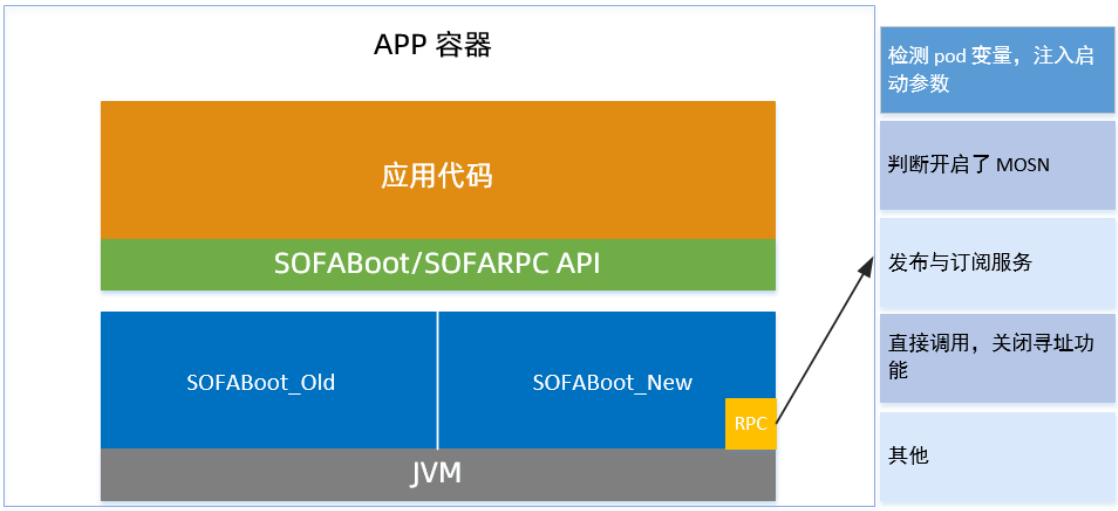
框架升级方案

- **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。

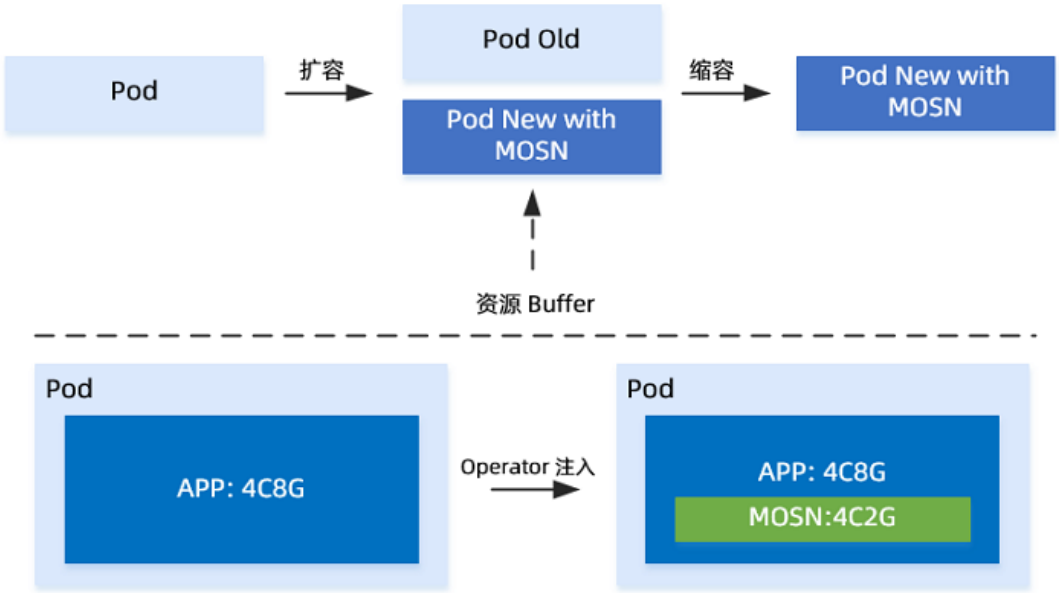


- **优缺点**：
  - 优点：通过批量的运维操作，直接修改了线上的 SOFABoot 版本，使得现有的应用具备了 MOSN 的能力。
  - 缺点：
    - 该升级方案运行时需要关闭流量等辅助操作。
    - 不具备平滑升级的能力。
    - 直接和业务代码强相关，不适合长期操作。
- **不采用社区流量劫持方案的考虑**：
  - iptables 在规则配置较多时，性能下滑严重。
  - 它的管控性和可观测性不好，出了问题比较难排查。
  - Service Mesh 从初期就把蚂蚁集团线上系统全量切换 Mesh 作为目标，对性能和运维的要求非常高，不能接受业务有损或者资源消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

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。

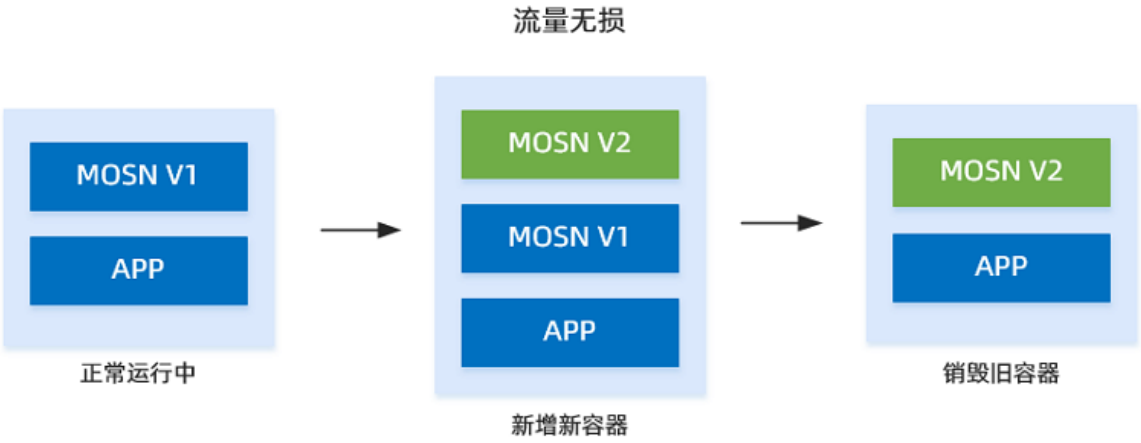
简版升级方案

- **方案思路**：销毁容器，用新容器重建，示例如下。

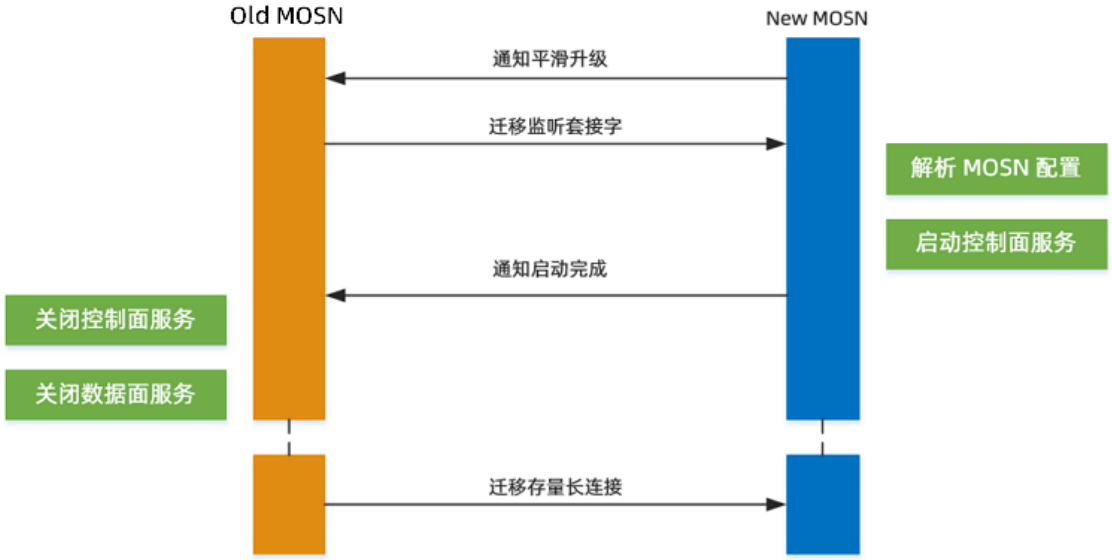
- **方案缺陷**：
  - 在容器数量很多时，运维成本不可承受。
  - 销毁容器后，如果重建速度不够快，就可能会影响业务的容量，造成业务故障。

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

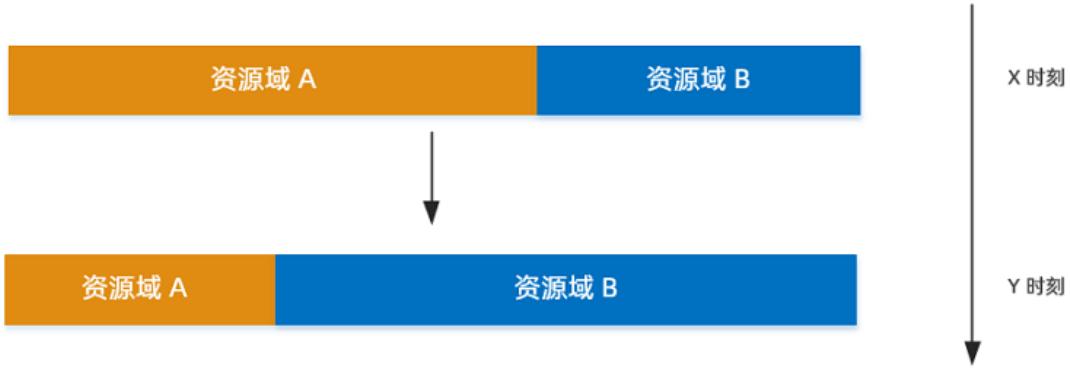
技术的变革通常并不是技术本身的诉求，而是业务的诉求，是场景的诉求。很少有人会为了升级而升级，为了革新而革新。通常，技术受业务驱动，也反过来驱动业务。

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。

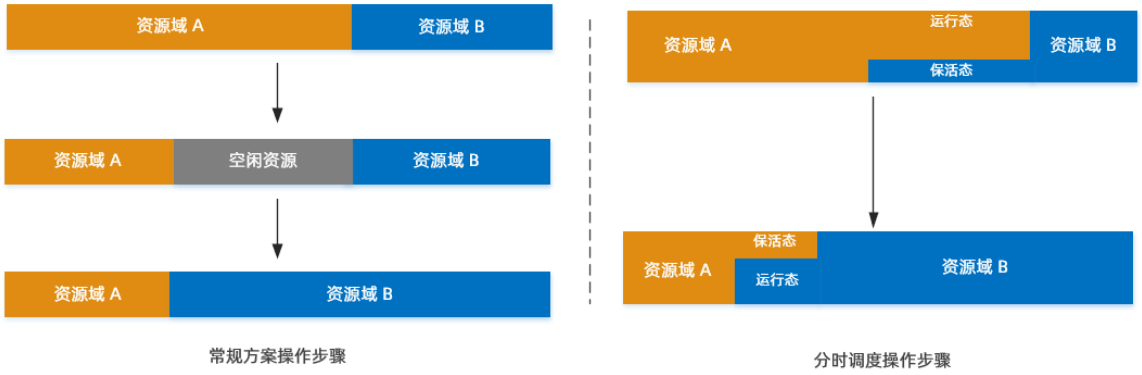
这种场景，对于业务来说就意味着代码已经最优化而流量却几乎无法支撑，解决办法就是扩容增加机器，而更多的机器意味着更多的成本付出。对于这样的情况，Service Mesh 是一个很好的解决办法。

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。

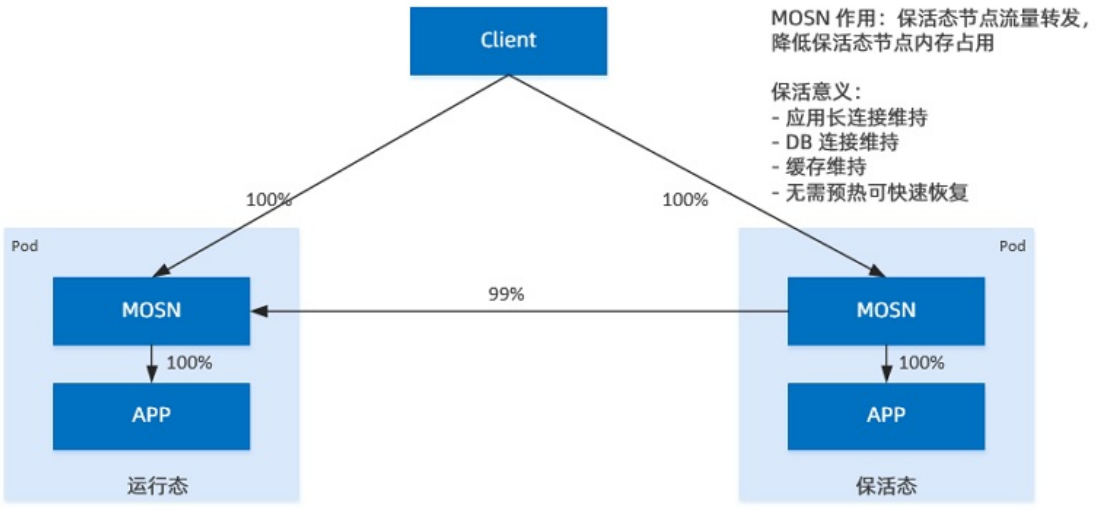


- **借调方案**：上述资源需求需要通过借调机器来完成，借调方案有下述 2 种。



- **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



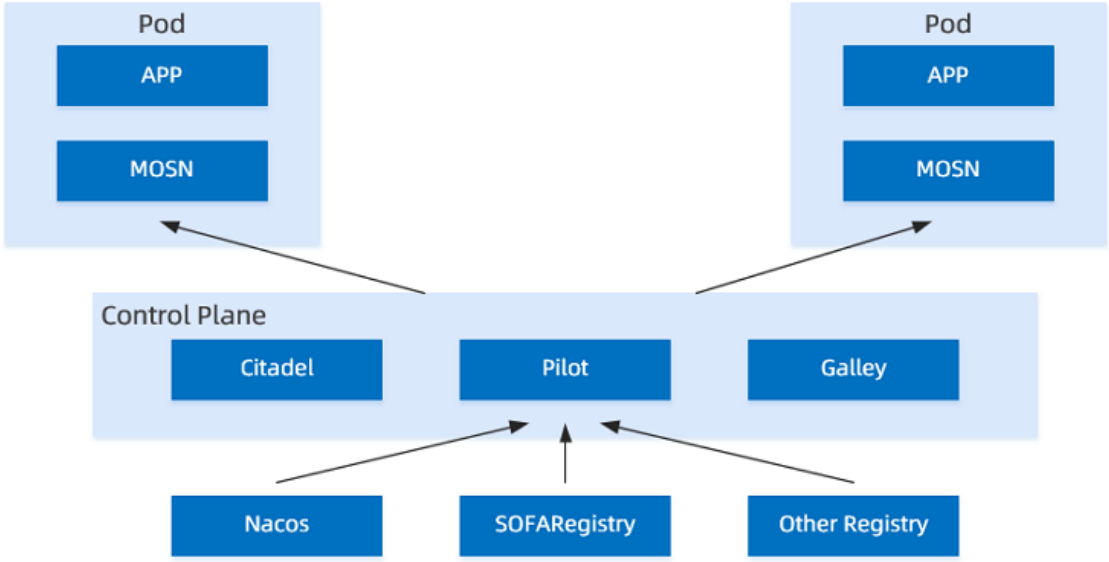
Service Mesh 未来之展望

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

5.3. 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。

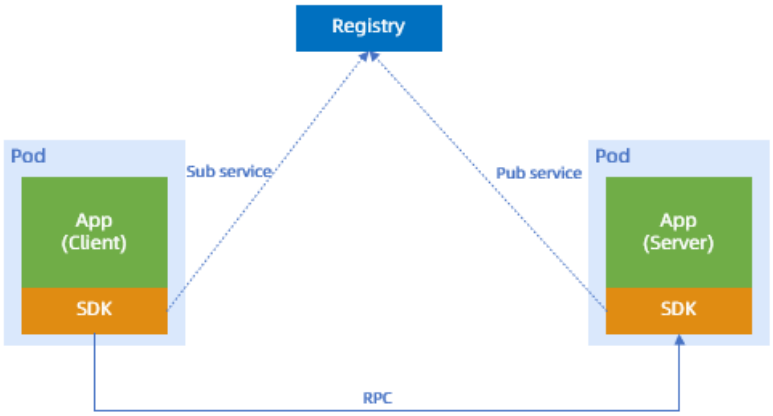
MOSN 简介

经典微服务架构和 Service Mesh 架构的差别主要包括：

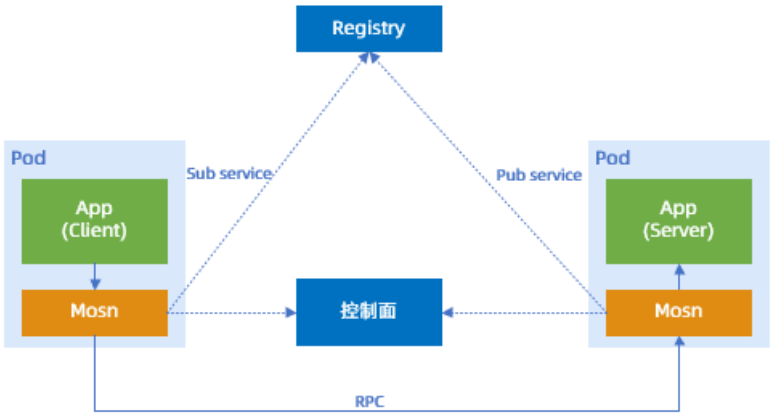
- 经典微服务体系：基于 SDK 实现服务的注册和发现。

- 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



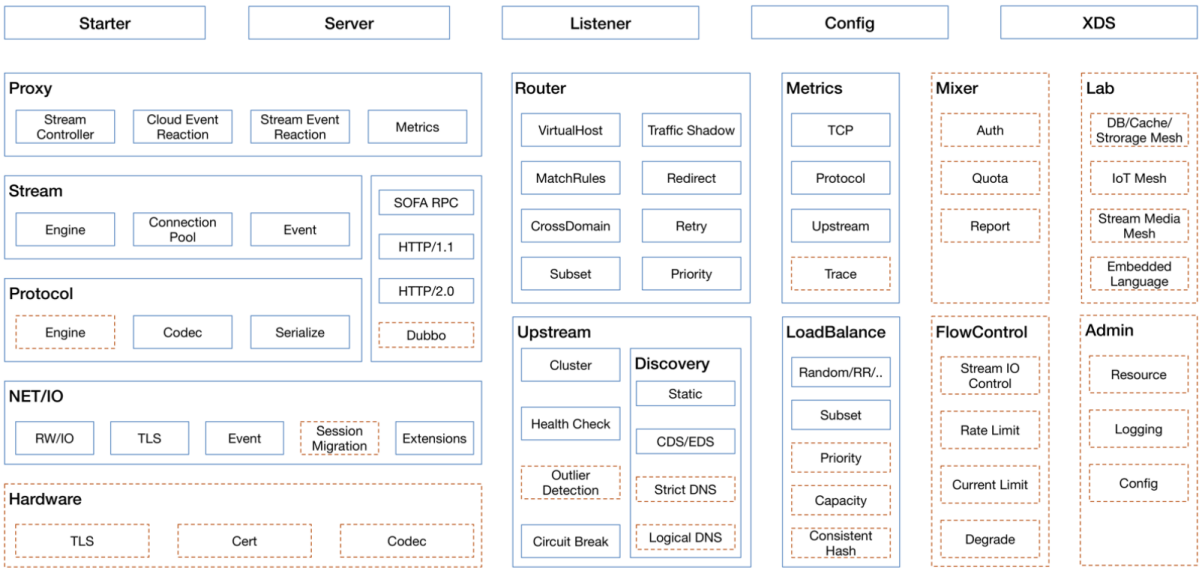
Service Mesh 微服务架构



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：

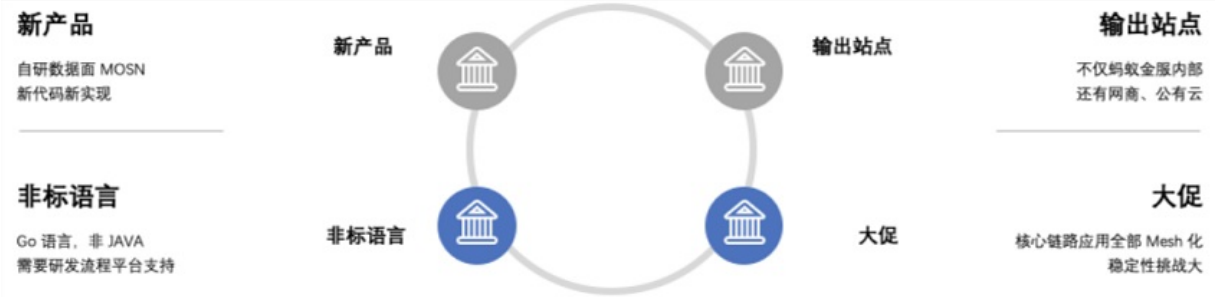


落地面临的问题

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？

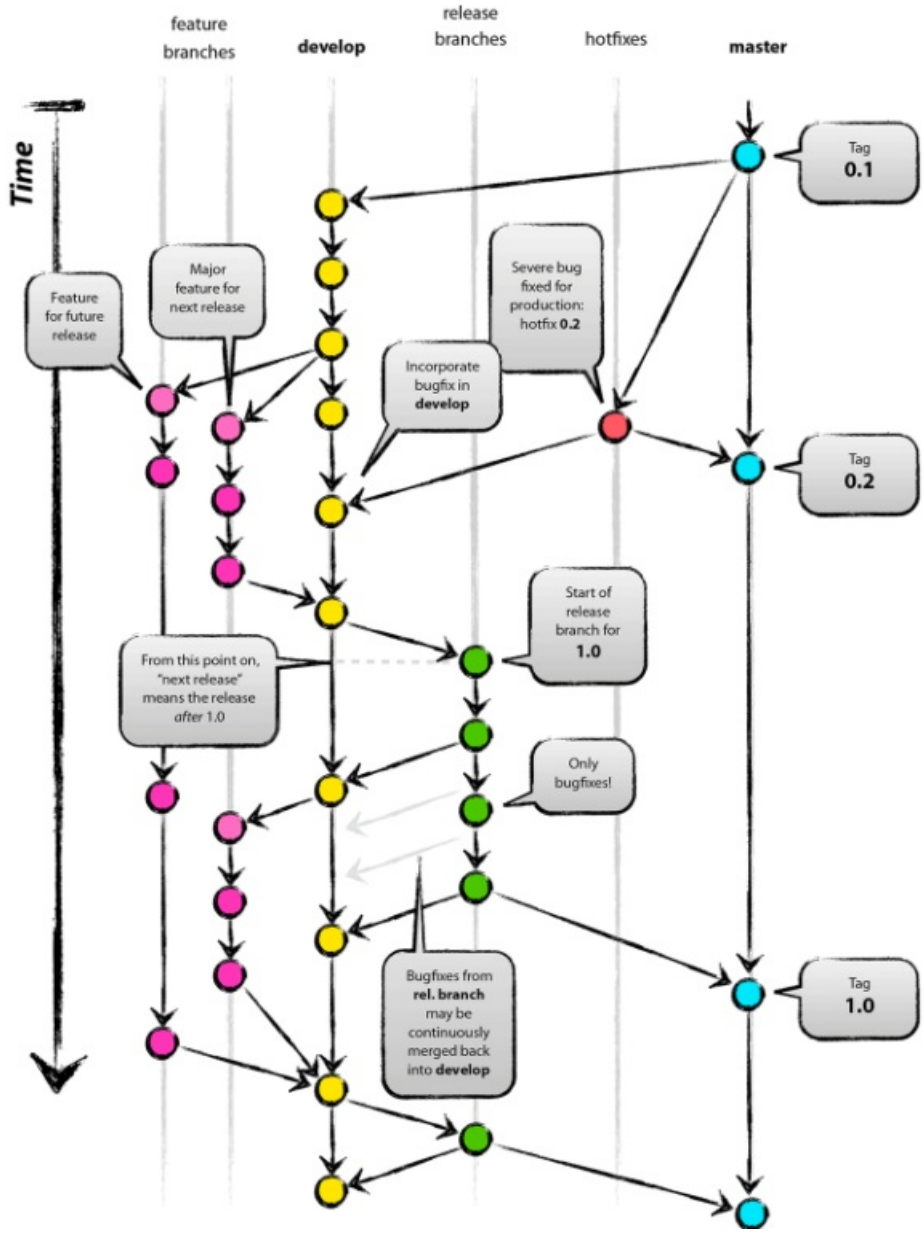
如何应对质保挑战？

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：

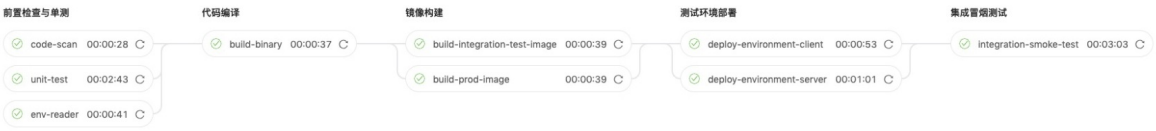
- **git-flow 分支管控**

  代码管理需要一个清晰的流程和规范，蚂蚁团队引入了 Vincent Driessen 提出的代码管理解决方案。详情请参见 A Successful Git Branching Model。



- **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

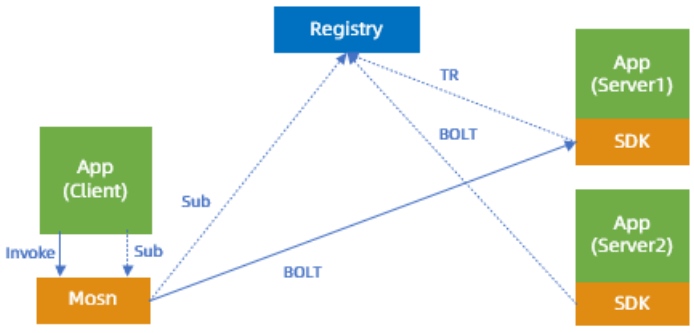
MOSN 兼容性验证图

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。

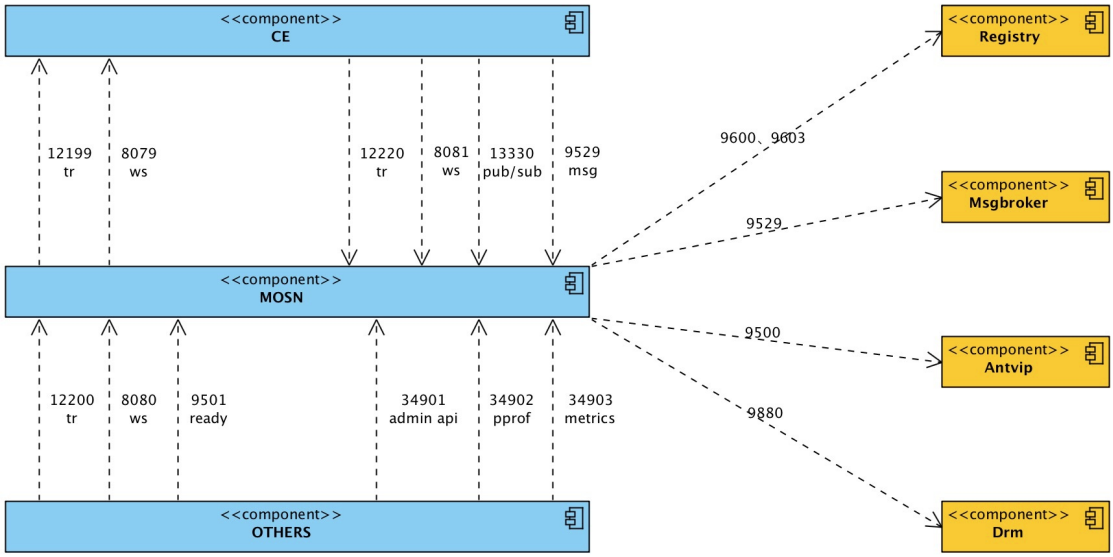
修复方案：最终的修复方案是，MOSN 收到配置中心的地址列表后，会做一层过滤，只要服务端列表中包含 TR 协议的，则全部降级到 TR 协议。

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

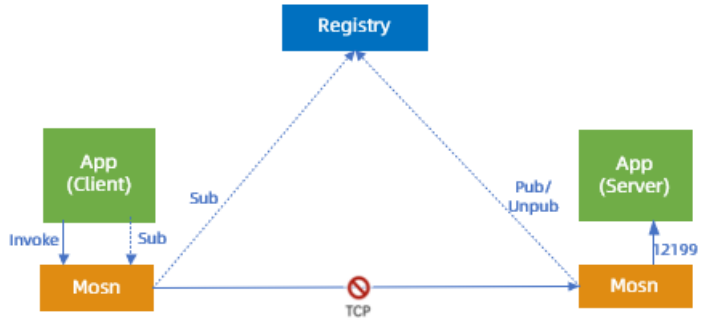
从 MOSN 的视角来看，其外部依赖如下：



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。

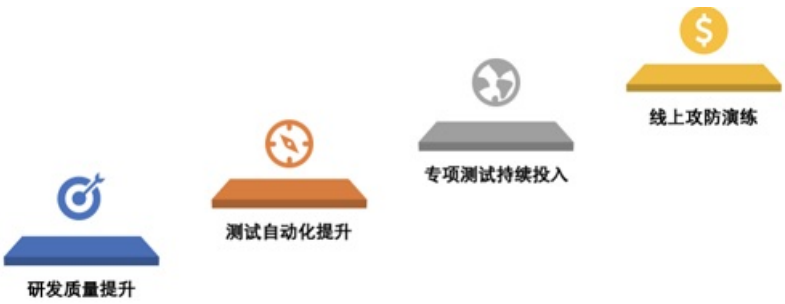
监控及巡检

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

5.4. Mesh 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：

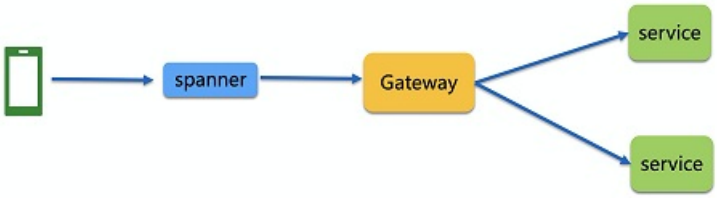
- 网关的演变历史：解释网关为什么要 Mesh 化。
- 网关 Mesh 化：阐述如何进行 Mesh 化改造。
- 双十一落地：介绍在此过程中实现三板斧能力。

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：



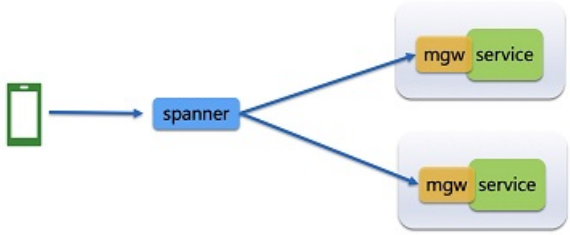
流程说明：

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。

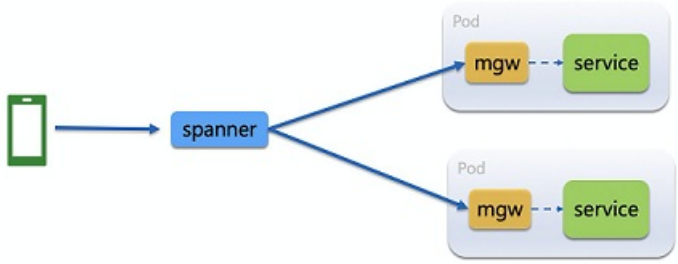
但凡事具有两面性，随着在 TOP 30 的网关应用中落地铺开，去中心化网关的缺点也逐步显现：

- **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：

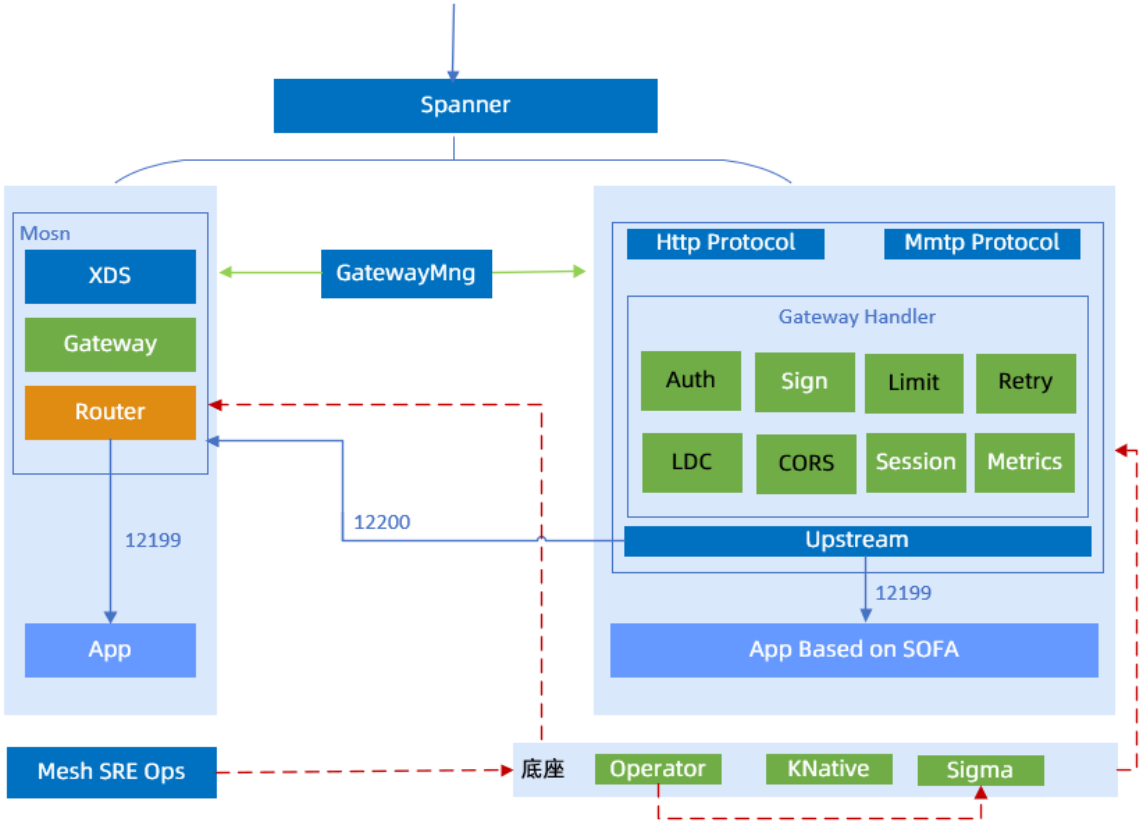
- Mesh 化之后的请求处理流程不是同进程调用，比去中心化网关多了一跳，是否有性能损耗？
- 如此大的改革之后，在上线过程中如何保障稳定性？

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，请参见 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：

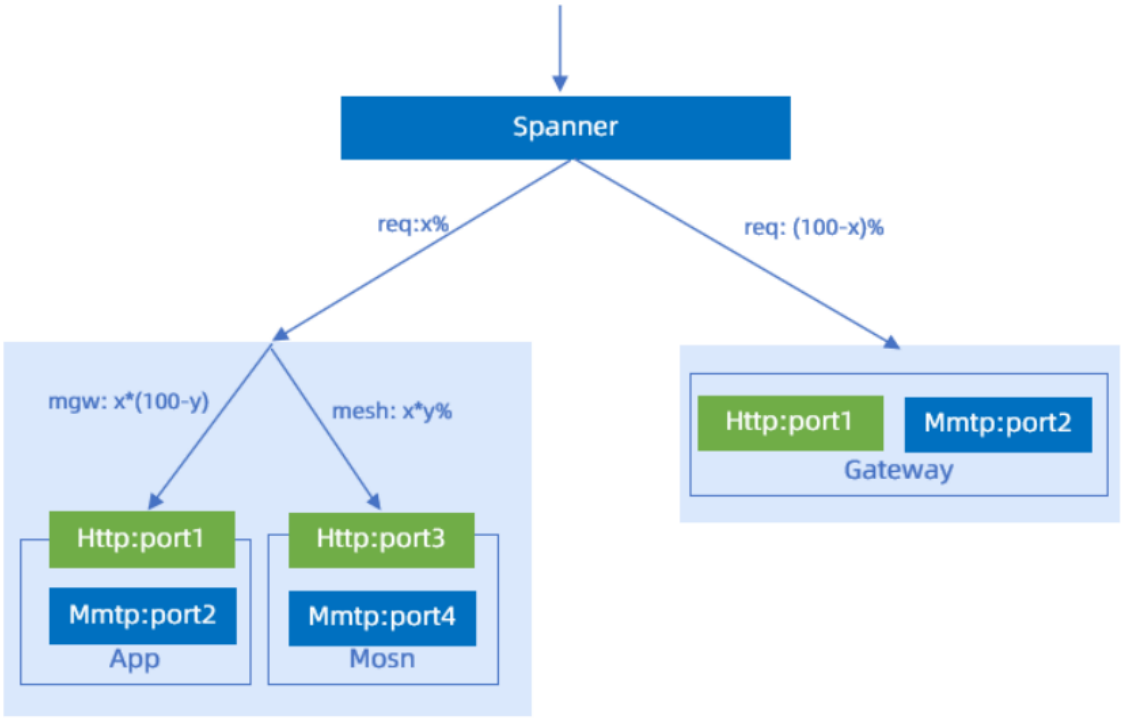
- 大促支付链路淘系支付 API 100% 引流。
- 大促会员链路主战场应用 100% 引流。
- 网关 Top API 5% 引流。

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。

5.5. 消息 Mesh

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。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，也变相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

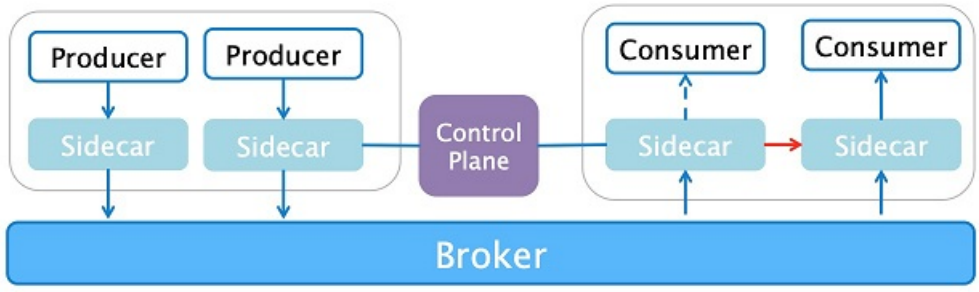
在蚂蚁集团内部，Msgbroker 基于推送模型提供高可靠、高实时、事务消息、Header 订阅等特性，帮助核心链路进行异步解耦，提升业务的可扩展能力，并先后伴随蚂蚁集团众多核心系统一起经历了分布式改造、单元化改造与弹性改造，目前主要承载蚂蚁内部交易、账务、会员、消费记录等核心在线业务的异步消息流量。

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：



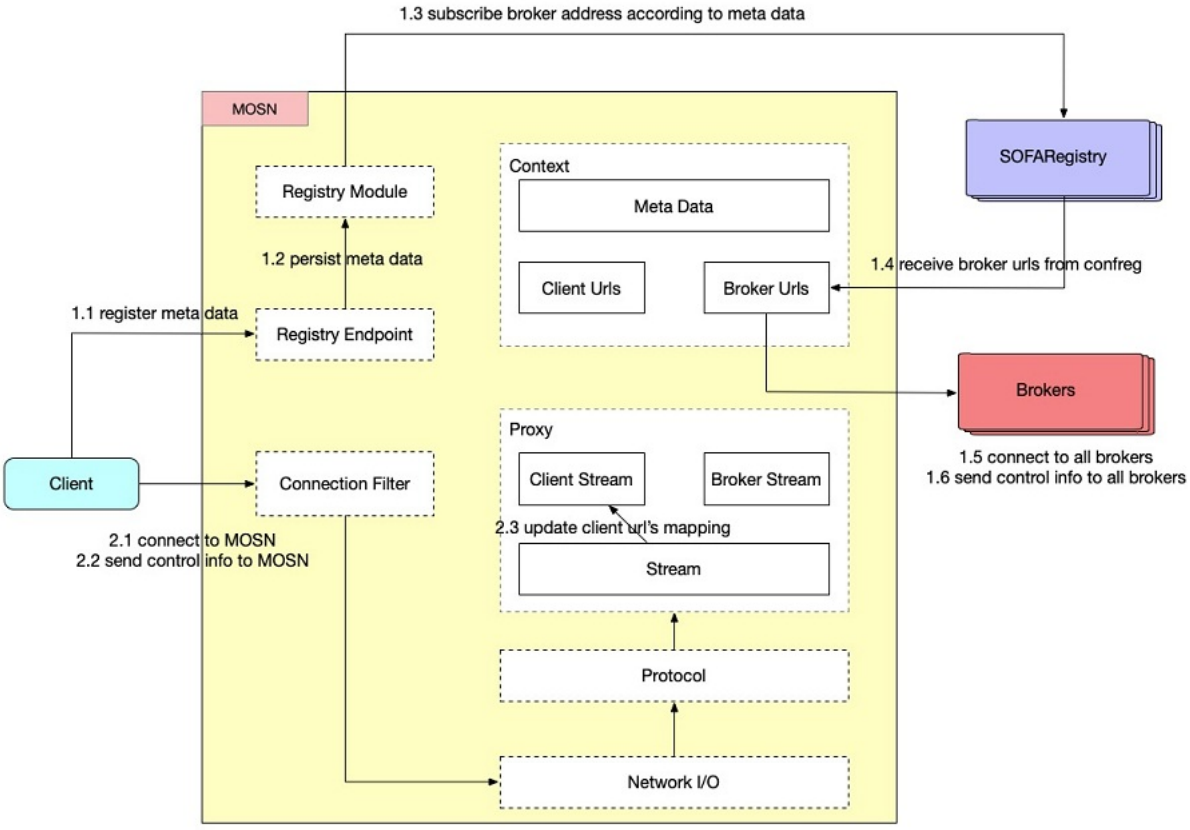
与原有的消息架构相比，主要的变化有：

- 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

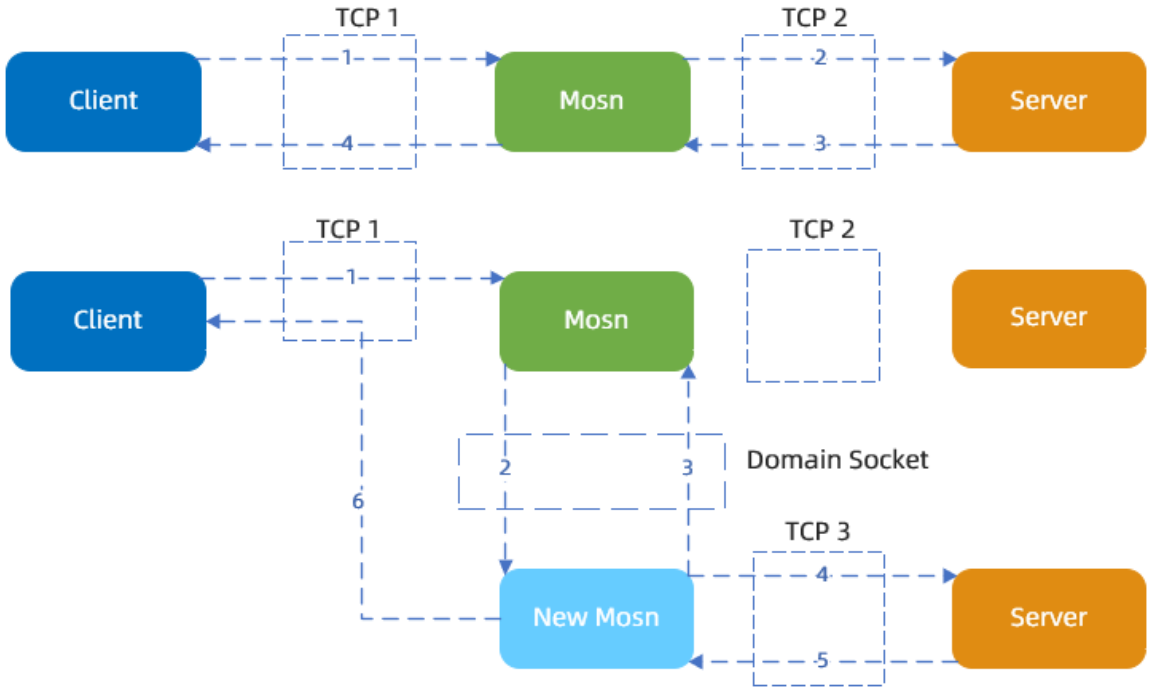
Service Mesh 将原有的与业务无关的逻辑下沉到 Sidecar，其占用的资源实际是原来业务容器会使用的资源，基于这一假设，在零新增成本的情况下，Service Mesh 平稳支撑了数十万规模级别的 Sidecar 容器分配。

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。

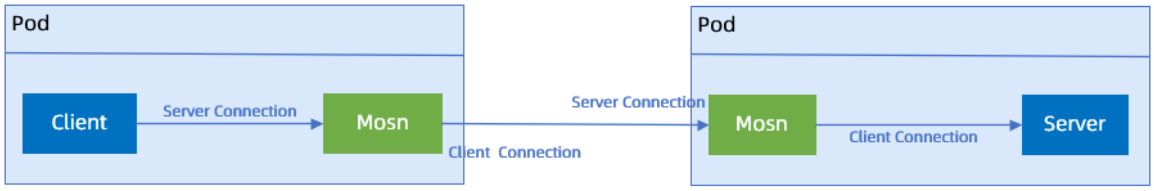
- 平滑升级的隐含前提条件：单条连接上的请求必须是单向的。
- 平滑升级的过程为：新的 MOSN 首先会被注入，并通过共享卷的 UnixSocket 检查是否存在老的 MOSN。若存在，则利用内核 Socket 的迁移实现老 MOSN 的连接全部迁移给新 MOSN，最终让老 MOSN 优雅退出，从而实现 MOSN 在整个升级和发布过程中对业务无任何打扰。

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。

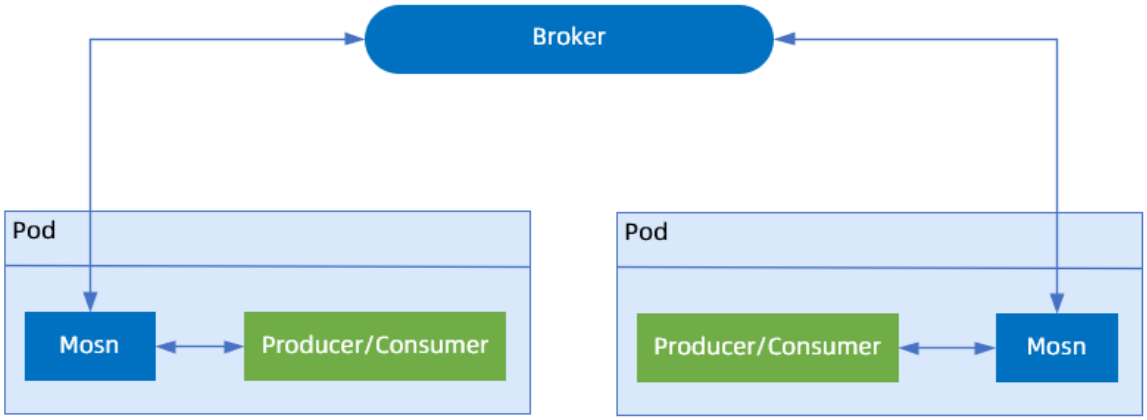
平滑升级方案与 RPC



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：

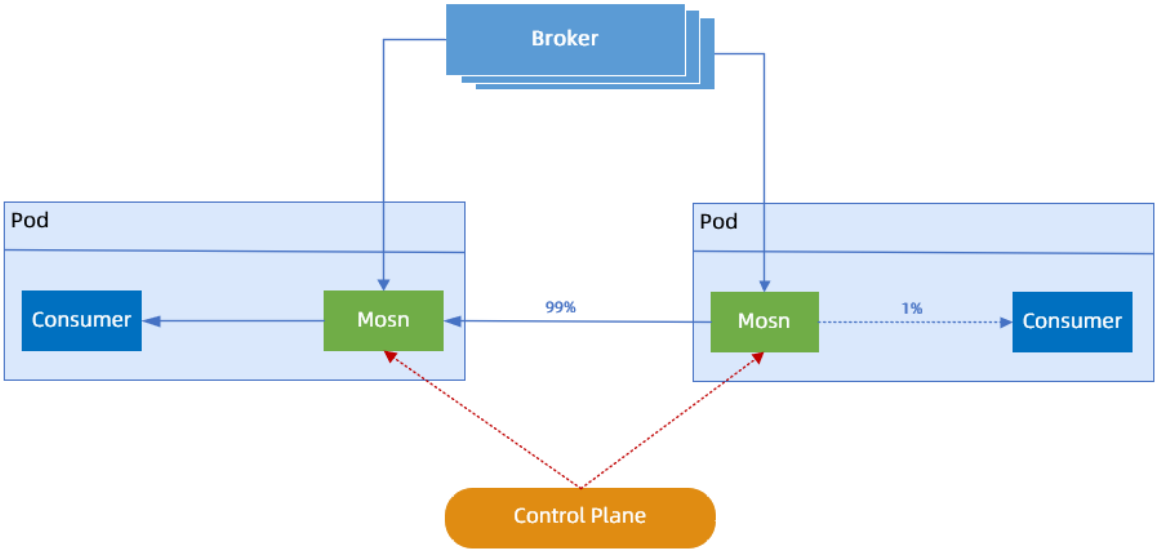
- 在连接迁移的过程中，如果消息客户端已处理完经过 MOSN 转发的服务端投递消息请求，但是还未回复响应，此时若把连接迁移到新的 MOSN，则新的 MOSN 将收到上述响应，但由于新 MOSN 缺失上下文，无法将该响应返回给正确的消息服务端。
- 在连接迁移完成，但老 MOSN 还未优雅退出期间，由于两个 MOSN 与消息服务端都存在连接，两者都会收到服务端发送的投递消息请求，但因两个 MOSN 与服务端连接的状态各自独立，可能会使客户端收到的请求 ID 相冲突。

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。

5.6. 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。

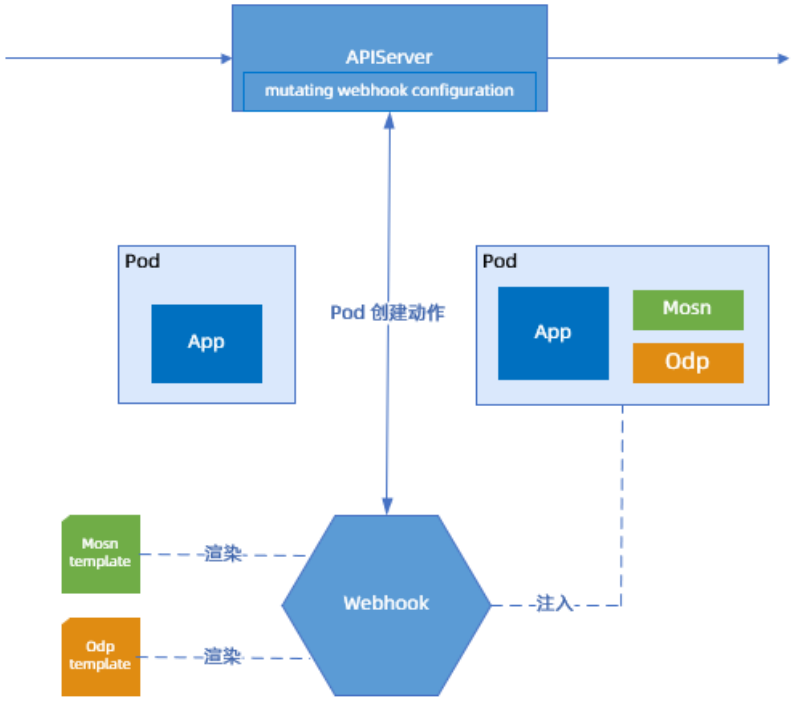
本文将从 MOSN（Sidecar Proxy）的运维和风险管控方面，分享 Operator 的实践经验。

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。

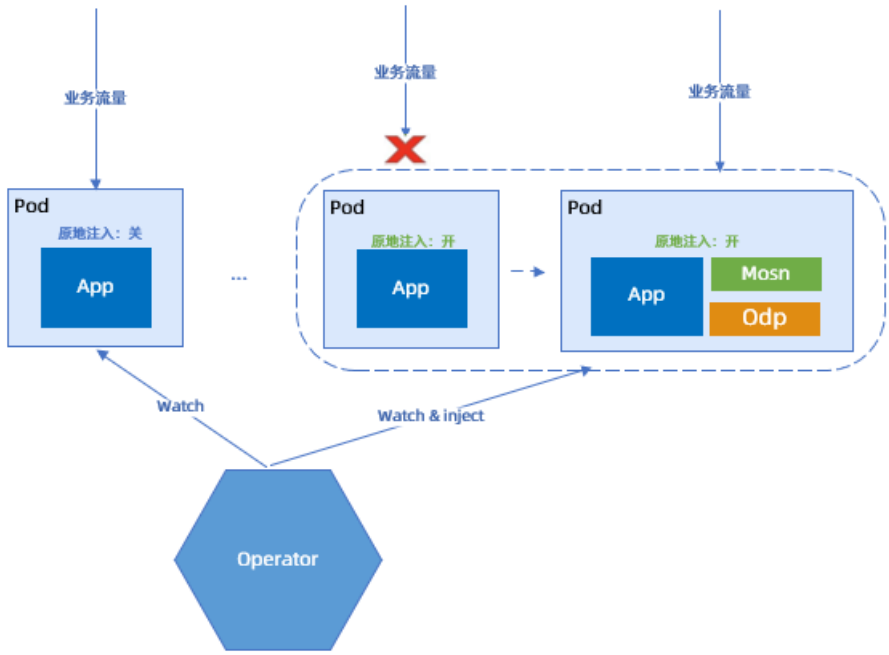
原地注入

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。

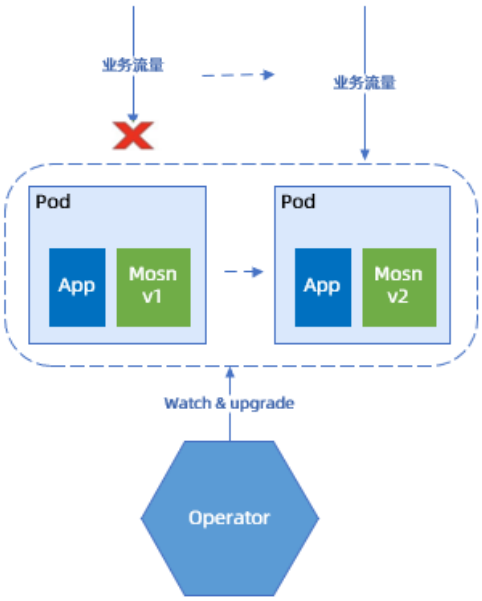
替换升级

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。



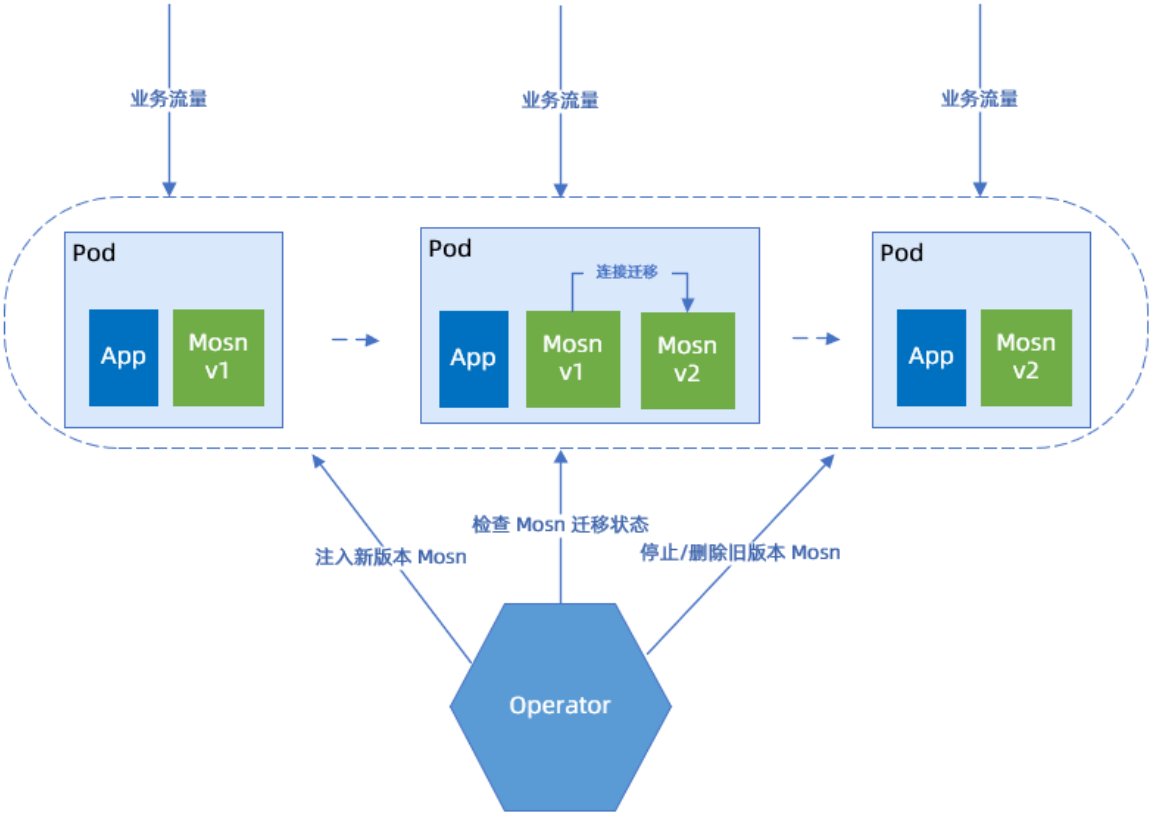
非平滑升级步骤：

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”。

5.7. 服务运维

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：

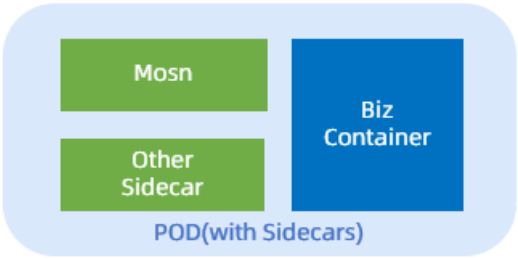
- 可以将 Service Mesh 本身的运维与业务容器解耦开来，实现中间件运维能力的下沉。
- 在业务镜像内，仅仅保留长期稳定的 Service Mesh 相关 JVM 参数，从而仅通过少量环境变量完成与 Service Mesh 的联结。
- 考虑到面向容器的运维模式的演进，接入 Service Mesh 还同时要求业务完成镜像化，为进一步的云原生演进打下基础。

	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

	优势	劣势
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。

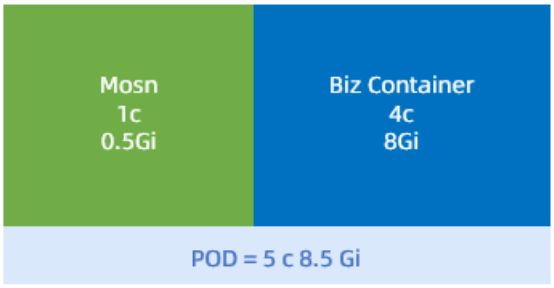
对资源模型的挑战

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：

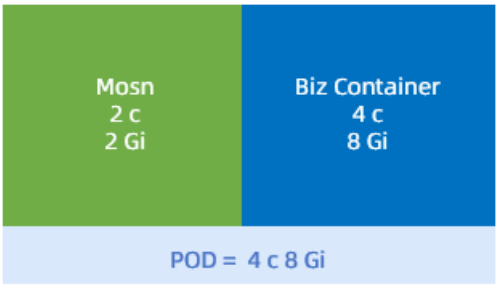
- 业务可见内存不一致。
- 业务监控偏差。
- 业务进程 OOM 风险。

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。

6.附录：基础术语

术语	说明
服务网格（ServiceMesh）	ServiceMesh 是一个基础设施层，用于处理服务间通信。通常是由一系列轻量级的网络代理组成的，它们与应用程序部署在一起，但应用程序不需要知道它们的存在。提供了一种透明的、与编程语言无关的方式，使网络配置、安全配置以及服务观察等操作能够灵活而简便地实现自动化。
Istio	Istio 是一个 Service Mesh 开源项目，是完整的非侵入式的微服务治理解决方案。Istio 提供一种简单的方式来为已部署的服务建立网络，该网络具有负载均衡、服务间认证、监控等功能，而不需要对服务的代码做任何改动。
边车（Sidecar）	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（fault injection）、熔断器以及分阶段发布等功能。
SOFAMOSN	SOFAMosn 全名 Modular ObservableSmart Network，可作为 SOFAMesh 中的数据平面 Sidecar。使用 Go 语言编写，兼容 Envoy 的 API。
Host	能够进行网络通信的实体（在手机或服务器等上的应用程序）。在 Envoy 中主机是指逻辑网络应用程序。只要每台主机都可以独立寻址，一块物理硬件上就运行多个主机。
下游（Downstream）	Downstream 主机连接到 Envoy，发送请求并或获得响应。
上游（Upstream）	Upstream 主机获取来自 Envoy 的链接请求和响应。
集群（Cluster）	Cluster 是 Envoy 连接到的一组逻辑上相似的上游主机。Envoy 通过服务发现发现集群中的成员。Envoy 可以通过主动运行状况检查来确定集群成员的健康状况。Envoy 如何将请求路由到集群成员由负载均衡策略确定。
运行时配置	与 Envoy 一起部署的带外实时配置系统。可以在无需重启 Envoy 或 更改 Envoy 主配置的情况下，通过更改设置来影响操作。

术语	说明
监听器（Listener）	Listener 是可以由下游客户端连接的命名网络位置（例如端口、unix 域套接字等）。Envoy 公开一个或多个下游主机连接的侦听器，一般是每台主机运行一个 Envoy，使用单进程运行，但是每个进程中可以启动任意数量的 Listener。目前只监听 TCP，每个监听器都独立配置一定数量的（L3/L4）网络过滤器。 Listenter 也可以通过 Listener Discovery Service（LDS）动态获取。
监听器过滤器（Listener filter）	Listener 使用 Listener filter来操作链接的元数据。它的作用是在不更改 Envoy 的核心功能的情况下添加更多的集成功能。 Listener filter 的 API 相对简单，因为这些过滤器最终是在新接受的套接字上运行。在链中可以互相衔接以支持更复杂的场景，例如调用速率限制。 Envoy 已经包含了多个监听器过滤器。
Http Route Table	HTTP 的路由规则，例如请求的域名、Path 符合什么规则、转发给哪个 Cluster。
健康检查（Health checking）	健康检查会与 SDS 服务发现配合使用。但是，即使使用其他服务发现方式，也有相应需要进行主动健康检查的情况。