InfoQ

腾讯云 Service Mesh 生产实践及架构演进

作者：单家骏

阅读数：1250 | 2018 年 12 月 23 日 | 话题：架构 最佳实践 Service Mesh 腾讯云

背景介绍

Service Mesh（服务网格）是一个基础设施层，让服务之间的通信更安全、快速和可靠，是云原生技术栈的关键组建之一。2018 年是 Service Mesh 高歌猛进的一年，Service Mesh 数据面板百花齐放，遍地开花，业界几乎所有大厂都在推出自己的 Service Mesh 产品。2018 年 Service Mesh 大事件如下：

- 2018 年 7 月 31 日，Istio 1.0 版本发布，标志着 Istio 可用于生产环境。
- 2018 年 9 月 19 日，Conduit，这个史上唯一一个主打 rust 语言的 Mesh，宣布合入到 Linkerd，后续作为 linkerd2.x 版本继续演进。
- 2018 年 11 月 28 日，Istio 的官配 sidecar，高性能边缘代理 Envoy，成为了继 k8s 以及 prometheus 之后，第三个从 CNCF 毕业的项目。
- 2018 年 12 月 5 日，AWS 推出服务网状网络 App Mesh 公开预览版，供用户轻松的监视与控制 AWS 上，构成应用程序微服务之间的通信。

早在 2017 年腾讯云中间件团队就选定 Istio 为技术路线，开始 Service Mesh 的相关研发工作，作为腾讯云 TSF（微服务平台）的无侵入式服务框架的核心实现，并在 18 年初在腾讯广告平台投入，打磨稳定后陆续开始对外输出，目前在银行、电商、零售、汽车等行业都有落地案例。

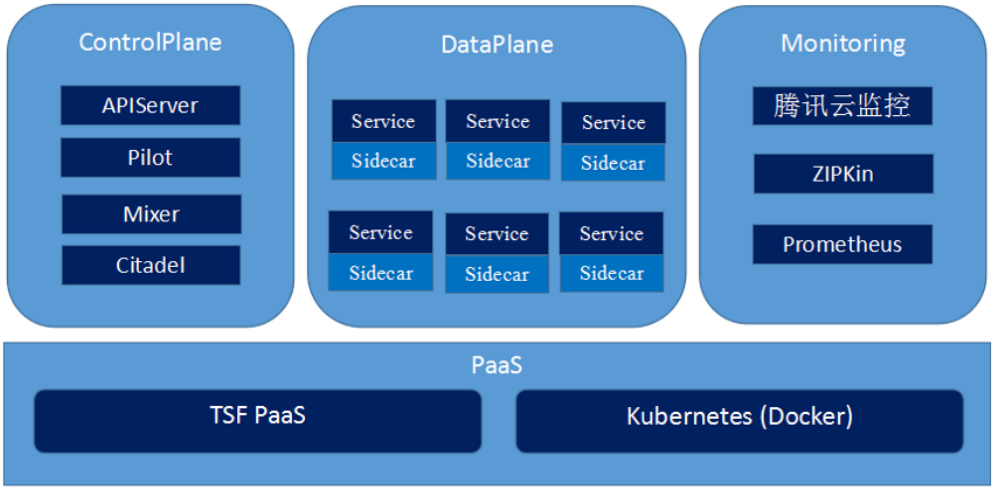
落地过程并非一帆风顺，本文将对腾讯云 Service Mesh 在生产实践过程中遇到的典型问题以及解决方案进行总结分享，同时对腾讯云 Service Mesh 后续重点探索的技术方案进行简要阐述。

腾讯云 Service Mesh 核心技术实现

在实现上，基于业界达到商用标准的开源软件 Istio、envoy 进行构建。整体架构上，从功能逻辑上分为数据面和控制面：

控制面主要提供配置及控制指令支撑 sidecar 的正常运行，以及对服务运行过程中的相关数据进行采集。

数据面主要是提供通信代理（sidecar）来进行透明的服务调用，支撑正常的业务流程。



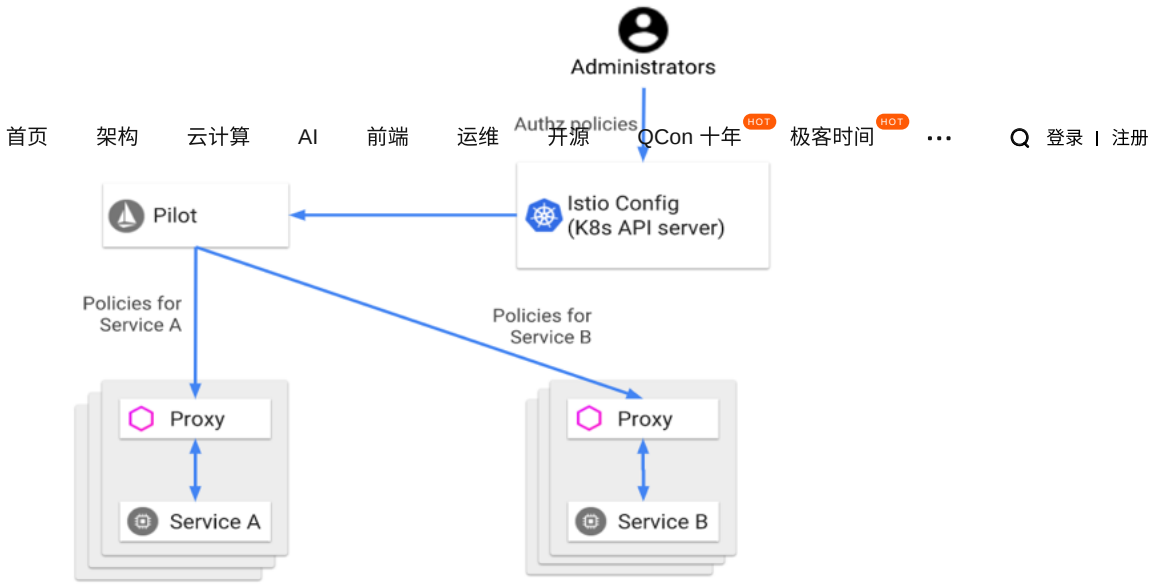
腾讯云Service Mesh

接下来，让我们对腾讯云 Service Mesh 各关键优化点做详细的描述。

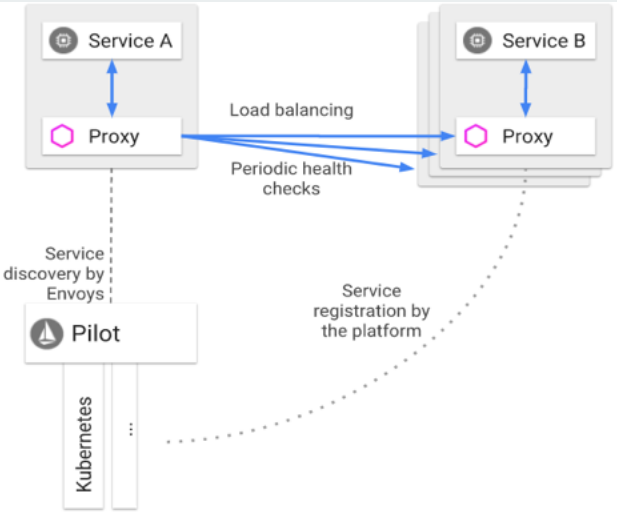
解耦 k8s，拥抱其他计算平台

众所周知，Istio 强依赖于 Kubernetes ，大部分功能都依托于 Kubernetes 平台进行构建。下面列举几个核心的功能：

1. **服务配置管理**：Istio 配置通过 Kubernetes Crd (custom resources definition) 以及 configmap 进行存取。

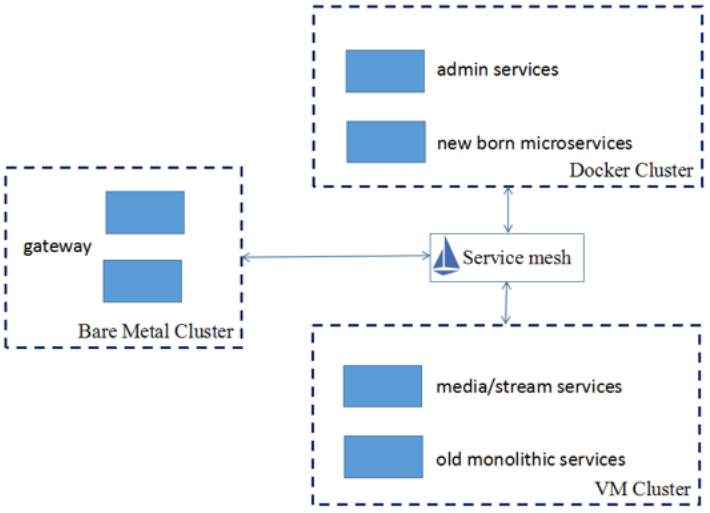


2. **服务发现及健康检查**：Istio 全功能的服务注册发现能力是基于 Kubernetes 的 PodServices 能力以及 Endpoints 机制实现的，节点健康检查能力基于 ReadinessProbe 机制实现 （当前社区上面也有基于



Discovery and Load Balancing

但实际落地过程中，TSF 的用户并非全部是 Kubernetes 用户，例如公司内部的一个业务因历史遗留问题，不能完全容器化部署，同时存在 VM 和容器环境，架构如下：

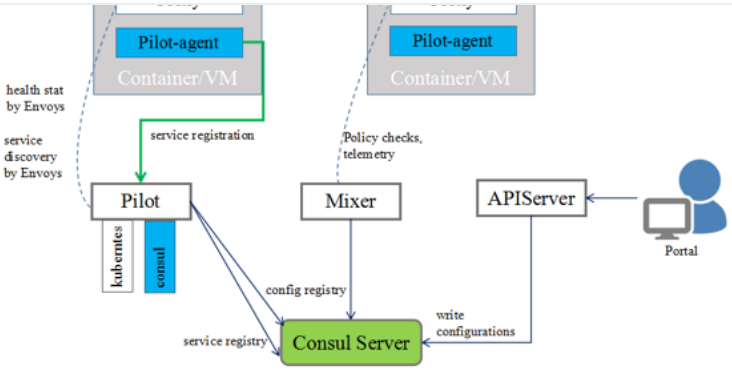


从业务架构图可以看出，业务要求 TSF 可以支持其部署在自研 PAAS 以及 Kubernetes 的容器、虚拟机以及裸金属的服务都可以通过 Service Mesh 进行相互访问。

因此，为了实现多平台的部署，必须与 Kubernetes 进行解耦。经过分析发现，脱离 Kubernetes 后，Istio 存在以下三个问题：

1. Pilot/Mixer 的远程动态配置能力不可用（只能用本地配置）；
2. Pilot 无法获取服务节点健康信息；
3. 无法通过 Istioctl（Istio 小工具）进行服务注册 / 反注册以及写配置能力。

针对这 3 个问题，TSF 团队对 Istio 的能力进行了扩展和增强，增强后的架构如下：



下表更详细的描述了存在的问题、解决方案以及所得到的目的，同时 TSF 团队实现了 Istio 对 Consul 的完整适配。

问题	增强点	目的
Pilot/Mixer 的远程动态配置能力不可用	组网中增加 Consul Server 集群作为服务注册及配置中心 Pilot/Mixer 扩展 ConfigController 接口，增加基于 Consul 的配置管理能力	彻底使用 Consul 作为配置管理中心，脱离 Kubernetes 的 crd 能力
Pilot 无法获取服务节点健康信息	Pilot 实现 HDS 接口，envoy 通过 HDS 接口上报健康信息给 Pilot，Pilot 将健康信息上报给服务注册中心	使用 envoy 原生的 HDS 健康上报能力进行节点健康上报，脱离 Kubernetes 的 ReadinessProbe 能力
无法通过 Istioctl 进行服务注册/反注册以及写配置能力	增强 Pilot-agent 能力，支持启动时根据描述文件自动注册服务 新增 APIServer 组件，对接 Consul，负责配置的 CURD	服务通过描述文件自动注册，方便用户进行一站式的微服务开发 通过 APIServer 屏蔽底层配置中心，对外提供简单的 REST 接口，方便管理 Protal 和二次开发 SDK 的对接

经过改造后，Service Mesh 成功与 Kubernetes 平台解耦，组网变得更加简洁，通过 REST API 可以对数据面进行全方位的控制，可从容适配任何的底层部署环境，对于私有云客户可以提供更好的体验。

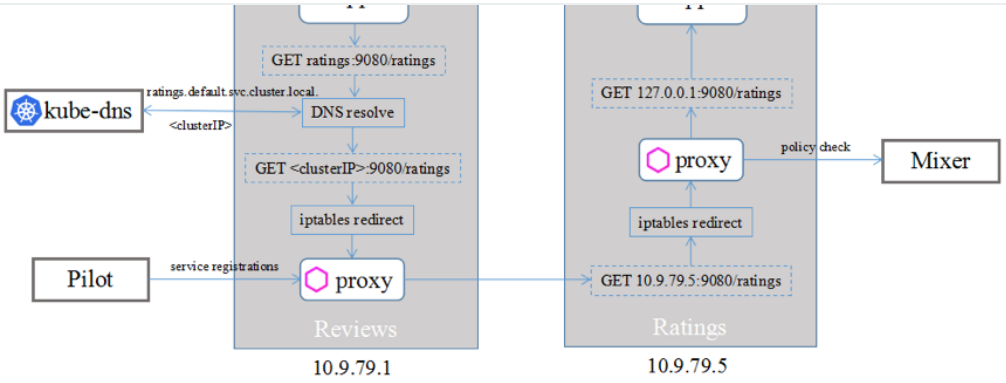
服务寻址模式的演进

解决了跨平台部署问题后，第二个面临的问题就是服务的寻址互通问题。

Istio 下的应用使用 FQDN（fully qualified domain name）进行相互调用，基于 FQDN 的寻址依赖 DNS 服务器，Istio 官方对 DNS 服务器的说明如下：

> Envoy determines its actual choice of service version dynamically based on the routing rules that you specify by using Pilot. This model enables the application code to decouple itself from the evolution of its dependent services, while providing other benefits as well (see Mixer). Routing rules allow Envoy to select a version based on conditions such as headers, tags associated with source/destination, and/or by weights assigned to each version.
>
> Istio also provides load balancing for traffic to multiple instances of the same service version. See Discovery and Load Balancing for more.
>
> Istio does not provide a DNS. Applications can try to resolve the FQDN using the DNS service present in the underlying platform (kube-dns, mesos-dns, etc.).

Istio 的官方 demo中，Reviews 与 Ratings 之间的完整的服务调用会经过以下过程：



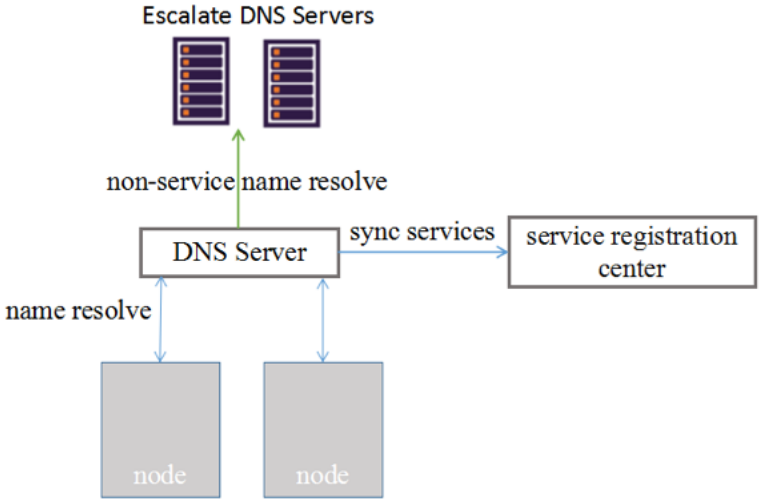
从图上可以看出，Reviews 和 Ratings 的互通，kube-dns 主要实现 2 个功能：

1. 应用程序的 DNS 请求被 kube-dns 接管；
2. kube-dns 可以将服务名解析成可被 iptables 接管的虚拟 IP（clusterIP）。

在私有云的实际交付中，客户的生产环境不一定包含 Kubernetes 或者 kube-dns，我们需要另外寻找一种机制来实现上面的两个功能。

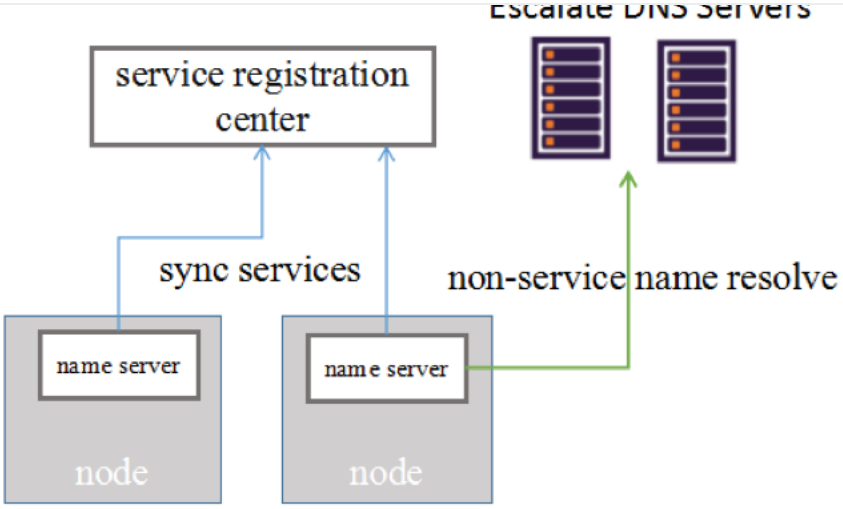
在 DNS 选型中，有集中式和分布式两种方案，分别如下：

- 集中式 DNS：代表有 ConsulDNS, CoreDNS 等，通过内置机制或者插件的方式，实现与服务注册中心进行数据同步。其架构组网如下：



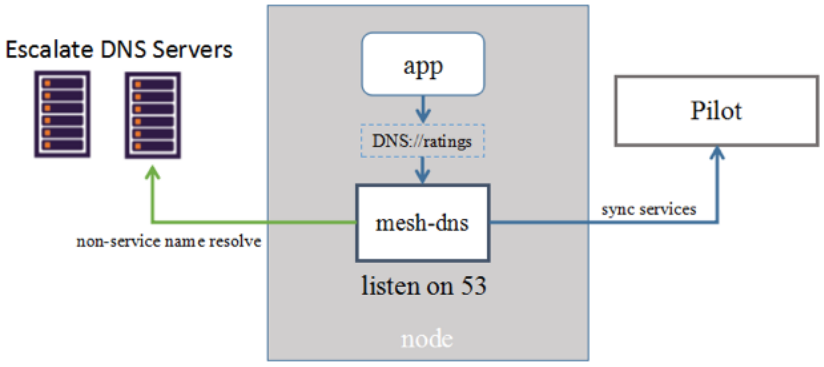
kube-dns 也属于集中式 DNS 的一种，集中式 DNS 存在以下问题：组网中额外增加一套 DNS 集群，并且一旦 DNS Server 集群不可服务，所有数据面节点在 DNS 缓存失效后都无法工作，因此需要为 DNS Server 考虑高可用甚至容灾等一系列后续需求，会导致后期运维成本增加。

- 分布式 DNS：就是将服务 DNS 的能力下沉到数据平面中，其架构组网如下：



分布式 DNS 运行在数据面节点上，DNS 无单点故障，无需考虑集群容灾等要素，只需要有机制可以在其 down 掉后重新拉起即可。但是，由于其与业务进程运行在同一节点，因此其资源占用率必须控制得足够低，才不会对业务进程产生影响。

综合考虑，最终选用了分布式 DNS 的方案，最开始团队采用独立进程作为 DNS Server 的方案，如下图：

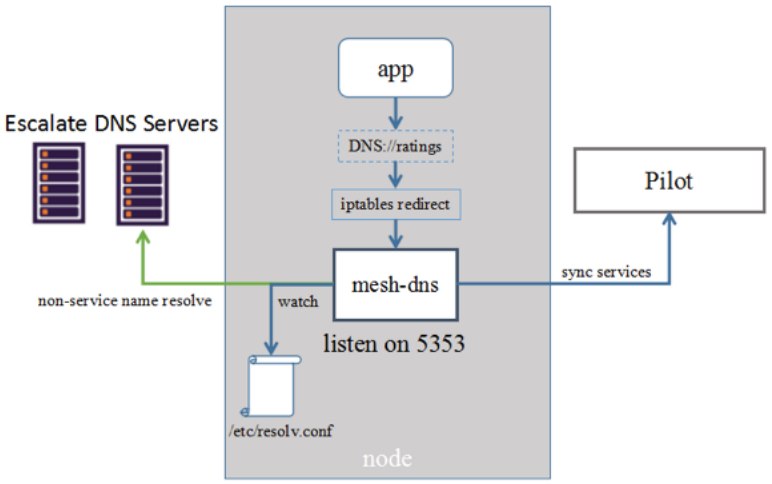


该方案新增监听在 127.0.0.1:53 上的 mesh-dns 进程，该进程实时从 Pilot 同步服务列表。Mesh-dns 在节点启动时将 127.0.0.1 写入到 /etc/resolv.conf 首行中，同时接管 /etc/resolv.conf 的其他 nameserver。这样，当 app 发起 DNS 查询时，DNS 请求首先会到达 mesh-dns，遇到匹配服务名的查询则直接返回，而当遇到不是针对服务名的 DNS 查询时，就把 DNS 请求转发给其他 nameserver 进行处理。

该方案看起来简单可行，但是经测试验证后发现存在以下问题：

1. resolv.conf 修改时间差问题：该方案需要对 /etc/resolv.conf 进行修改，在 linux 环境，域名解析机制是通过 glibc 提供的。而 glibc 2.26 之前的版本有个 BUG，导致假如在进程启动后，对 resolv.conf 就行修改，则该修改无法被该进程感知，直到进程重启。而由于在容器部署的场景中，mesh-dns 和应用分别部署在同一个 POD 的不同容器中，容器的启动是相互独立的，所以无法保证对 resolv.conf 的修改一定在应用启动前。即使改成通过 InitContainer 进行修改，当容器异常重启后，resolv.conf 也同样会被还原导致服务不可用。
2. 端口监听冲突问题：由于 mesh-dns 必须监听 53 端口，假如客户节点环境已经安装了 dnsmasq 等同样需要占用 53 的进程，则可能会出现端口冲突导致启动失败。
3. nameserver 选择策略问题：假如存在多个 nameserver，部分操作系统，默认会使用 rotate（随机选取一个作为首选查询的 nameserver）作为 nameserver 的选择策略。此时会出现一定概率下会选不到

针对上述问题，对方案进行了进一步的优化，优化后的方案如下图：

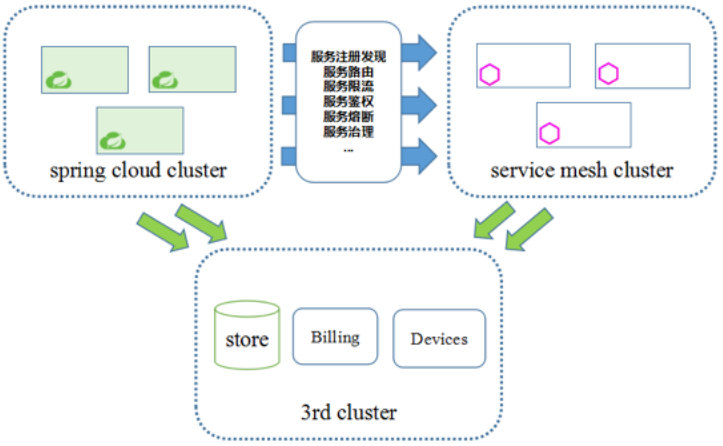


mesh-dns 不再监听 53 端口，而是监听在 5353 端口（可配置），启动时无需修改 resolv.conf。通过增加 iptables 规则，将所有发往 nameserver 的流量导入到 mesh-dns，从而解决了上文中的“端口监听冲突”以及“nameserver 选择策略”的问题。

mesh-dns 通过 inotify 监听 /etc/resolv.conf，可以随时获取环境中 dns 配置的更改，从而解决了上文中的“resolv.conf 修改时间差”的问题。

与非 Service Mesh 服务的互通

现实总是复杂的，前面解决 mesh 服务之间相互访问的问题，如何解决用户 Service Mesh 应用和其他非 Mesh 应用的相互访问呢？ 用户内部有不同技术栈，一部分服务基于 service mesh 进行实现服务，另外一部分服务基于 spring cloud 框架进行实现。同时，客户的微服务组网中，存在大量第三方服务如支付网关、分布式存储、设备等，微服务需要与这些第三方服务也存在交互。用户期望支持的架构如下图所示：



这个架构中，最大的挑战在于涉及了两个不同的微服务框架之间的互通。但是，这两个微服务框架从架构模式、概念模型、功能逻辑上，都存在较大的差异。唯一相通的点，就是他们都是微服务框架，可以将应用的能力通过服务的形式提供出来，给消费者调用，消费者实际上并不感知服务的具体实现。

基于这个共通点，为了使得不同框架开发的服务能够正常工作，TSF 团队做了大量的开发工作，将两个微服务框架，从部署模式、服务及功能模型上进行了拉通，主要包括如下几点：

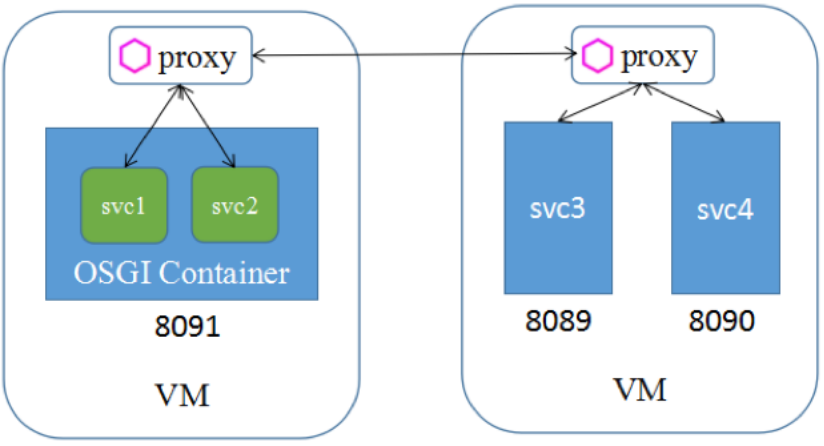
1. 服务模型的互通：基于统一的服务元数据模型，针对 pilot registry 及 spring cloud registry 的服务注册发现机制进行拉通。

3. 服务路由能力互通：基于标准权重算法以及标签模型，针对 pilot virtual-service 以及 spring cloud ribbon 能力进行拉通。
4. 服务限流能力互通：基于标准令牌桶架构和模型，以及条件匹配规则，对 mixer 及 spring cloud ratelimiter 能力进行拉通。

代理单节点多服务

用户的需求是多种多样的，在交付过程中存在如下多服务场景：

1. 客户机器资源不足，且没有做容器化，因此需要把多个服务部署到一个节点上。
2. 客户的传统应用使用 OSGI（一种 Java 模块化技术）实现，一个进程中包含多个服务，监听在同一个端口。



为了支持多服务场景，简化用户的使用流程，TSF 提供了服务描述文件，可支持多服务场景，服务配置文件与 Kubernetes 标准格式一致：

```
apiVersion: v1
kind: Application
metadata:
  name: service1
  namespace: nsTester
spec:
  services:
    - name: user
      healthCheck:
        path: /health
      ports:
        - targetPort: 8089
          protocol: http
    - name: venue.service
      healthCheck:
        path: /health
      ports:
        - targetPort: 8090
          protocol: http
```

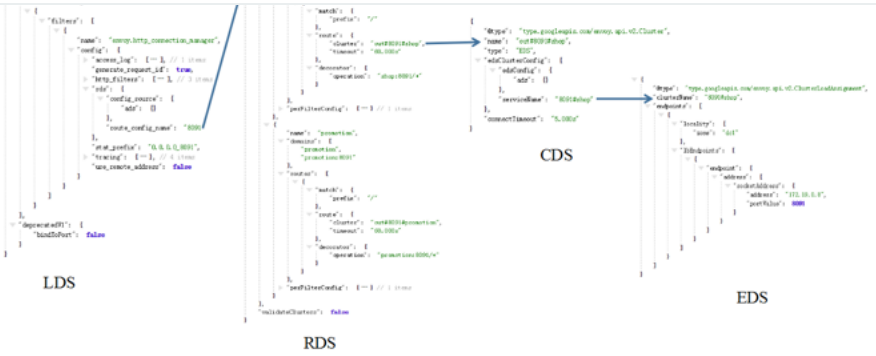
单节点多服务多端口

```
apiVersion: v1
kind: Application
metadata:
  name: service2
  namespace: nsTester
spec:
  services:
    - name: shop
      healthCheck:
        path: /health
      ports:
        - targetPort: 8091
          protocol: http
    - name: promotion
      healthCheck:
        path: /health
      ports:
        - targetPort: 8091
          protocol: http
```

单节点多服务单端口

pilot-agent 会根据服务配置，按照--的格式将配置中 services 注册成多个独立的服务实例。

在 OutBound 服务路由时，可以通过 LDS->RDS->CDS->EDS 的方式进行路由，和独立部署的服务没有区别：



然而，在 InBound 服务路由过程中，通过开源 Istio 生成的 listener 会遇到一些坑。

对于多服务监听同一端口的场景，开源 Istio 在生成 inbound 的时候，会将同 IP+Port 的其中一个服务给 reject 掉。

```
listenerMapKey := fmt.Sprintf( format: "%s:%d", endpoint.Address, endpoint.Port)
if old, exists := listenerMap[listenerMapKey]; exists {
    push.Add(model.ProxyStatusConflictInboundListener, node.ID, node,
        fmt.Sprintf( format: "Rejected %s, used %s for %s", instance.Service.Hostname, old.Service.Hostname, listenerMapKey))
    // Skip building listener for the same ip port
    continue
}
```

因此，生成的 LDS 中，只有其中一个服务的相关路由信息：

```
address": {
  "socketAddress": {
    "address": "172.19.0.8",
    "portValue": 8091
  }
},
"filterChains": [
  {
    "filters": [
      {
        "name": "envoy.http_connection_manager",
        "config": {
          "access_log": [ ... ], // 1 items
          "generate_request_id": true,
          "http_filters": [ ... ], // 3 items
          "route_config": {
            "name": "in#8091",
            "validate_clusters": false,
            "virtual_hosts": [
              {
                "domains": [
                  "*"
                ],
                "name": "promotion",
                "routes": [
                  {
                    "decorator": { ... }, // 1 items
                    "match": {
                      "prefix": "/"
                    },
                    "per_filter_config": {
                      "mixer": {
                        "http_api_spec": [ ... ], // 1 items
                        "mixer_attributes": { ... }, // 1 items
                        "network_fail_policy": {},
                        "quota_spec": [ ... ] // 1 items
                      }
                    },
                    "route": {
                      "cluster": "in#8091#promotion",
                      "timeout": "60.000s"
                    }
                  }
                ]
              }
            ]
          },
          "stat_prefix": "172.19.0.8_8091",
```


这样一来，普通消息投递，不会有什么问题（目标端点信息是一致的），但是假如需要与 mixer 结合，做 api 鉴权或者限流等操作，则会出现两个服务的 mixer_attribute 互相混淆的情况，导致功能不可用。

为了解决这个问题，团队分析了envoy 的 filter_chain_match 能力，对 pilot 进行改造，扩展了 listener 能力，通过 server_name 来分流数据包到不同的 filter 中。

最终生成的 LDS 如下：

```
  "name": "172.19.0.8_8091",
  "address": {
    "socketAddress": {
      "address": "172.19.0.8",
      "portValue": 8091
    }
  },
  "filterChains": [
    {
      "filterChainMatch": {
        "serverNames": [
          "promotion",
          "promotion:8091"
        ]
      },
      "filters": [...] // 1 items
    },
    {
      "filterChainMatch": {
        "serverNames": [
          "shop",
          "shop:8091"
        ]
      },
      "filters": [...] // 1 items
    }
  ],
  "deprecatedV1": {
    "bindToPort": false
  }
}
```


route to cluster: in#8091#promotion

route to cluster: in#8091#shop

经过这样的改造，同一端口上，不同的服务的 filter 配置不再冲突，两个服务的 mixer_attribute 也能相互隔离，顺利支持同端口多服务的场景。

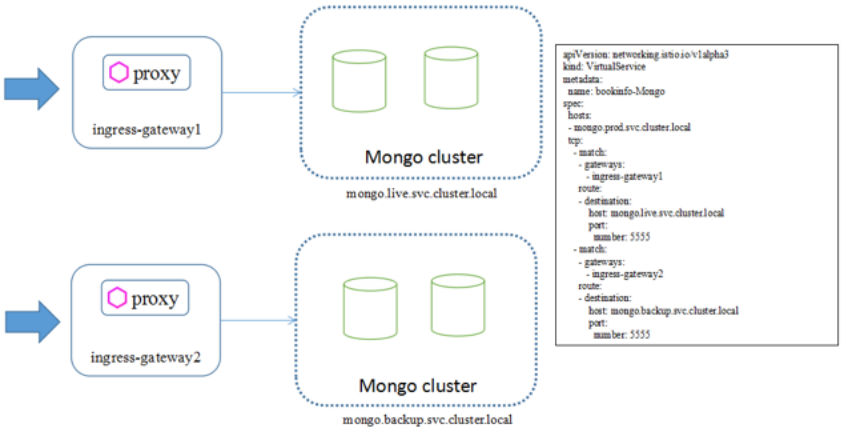
二进制协议的支持

在当前业界的开源 Service Mesh 产品中，主打的协议都是标准协议（HTTP1/2, GRPC），标准协议都有一个特点，那就是协议头中包含了目的端相关的所有信息，Service Mesh 会根据这些信息进行路由。如下表所示：

协议	头信息	用途
HTTP 1.1	Host	RDS 中寻找 VirtualHost，做服务路由
	Method, Path, 其他 headers	RDS 进行 route match； Mixer 中进行 rule match
HTTP 2.0/GRPC	:authority	RDS 中寻找 VirtualHost，做服务路由
	:schema, :path, 其他 headers	RDS 进行 route match； Mixer 中进行 rule match

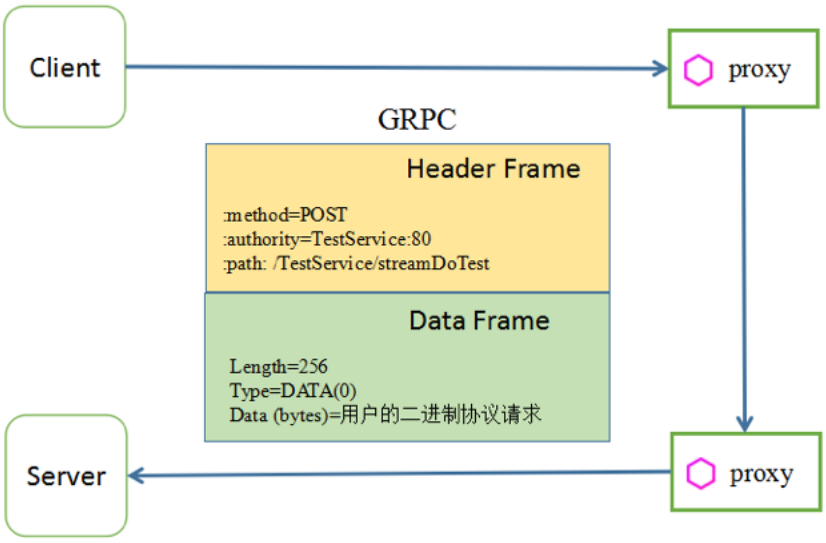
对于其他二进制协议，则分为 2 大类：

- 第一种是协议中带有目标端信息的二进制协议，如 thrift，dubbo 等；
- 第二种是协议中不带有目标端信息的二进制协议，这种就比较多了，一般常见于私有云中的各种私有通信协议。



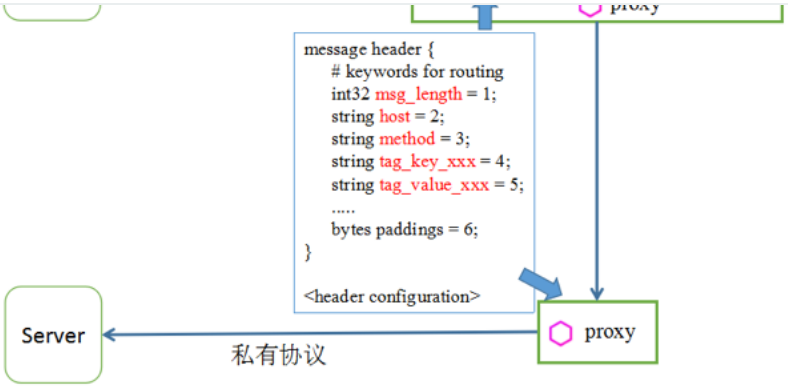
单纯的四层转发，无法满足复杂的微服务路由的需求。当前 TSF 交付的客户中，多个客户都提出了需要支持私有协议路由的需求，因此，针对这个问题，TSF 团队提供了两种解决方案。

1. 用户将私有协议转换成 GRPC 协议，接入到 Service Mesh：



由于 GRPC 的 Data Frame 本身传输的就可以是 TCP 协议，因此用户可以直接把自己的二进制协议通过 GRPC 的 bytes 类型编码，然后通过 Data Frame 传输过来.
该方案适用于本身有一定的技术积累，愿意去做 GRPC 改造的用户。

2. 根据用户定义的协议头描述文件，进行私有协议七层路由中间件团队对 envoy 的 filter 进行了扩展，用户提供一个 protobuf 格式的描述文件，指定协议头的字段顺序，proxy 根据描述文件的定义，进行消息头的接收及解析，然后根据解析后的消息头内容，进行七层路由和转发。



该方案适用于自身带有目标端信息的二进制协议，可以让私有协议的用户无需任何的改造，即可接入 Service Mesh。

总结

腾讯云 Service Mesh 当前通过 TSF 平台在持续交付中，上文主要针对落地过程中遇到的典型功能性需求及技术方案演进进行了总结介绍，除此之外，中间件团队在 Service Mesh 性能方面也有很多优化和探索，主要包括减少 envoy 和 mixer 之间的网络交互、优化数据包在 envoy 节点内部从内核态到用户态的拷贝次数、envoy 到 envoy 之间数据的转发性能等，后续将针对性能优化进行专项分享。

作者简介：

单家骏，来自腾讯公司。腾讯云高级工程师，负责腾讯云中间件 paas 以及 servicemesh 的研发与架构，关注云原生与中间件技术。热爱开源、崇尚技术，希望能够使用技术使软件的应用变得简单、高效和美好。

架构 最佳实践 Service Mesh 腾讯云

最新评论