

- 1 办公地点租赁
- 2 呼叫中心系统
- 3 陆家嘴写字楼
- 4 写字楼租赁
- 5 分布式光伏发电
- 6 甲醛治理加盟
- 7 微商分销系统
- 8 程序员外包公司
- 9 工位出租
- 10 访客系统
- 11 增强视力的方法
- 12 modbus网关
- 13 想租办公室

人月神话的博客

博客认证

深圳市远行科技有限公司 公司副总经理

https://blog.sina.com.cn/emmi

[订阅]

[手机订阅]

首页

博文目录

关于我

个人资料



人月神话

微博

加好友

发纸条

写留言

加关注



博客等级: 21

博客积分: 6456

博客访问: 4,443,195

关注人气: 5,979

获赠金笔: 702

赠出金笔: 1

荣誉徽章:



上海高级养老院



分布式光伏发电



modbus网关



甲醛治理

正文

字体大小: 大 中 小

从API网关的去中心化到ServiceMesh分布式服务治理(200826)

(2020-08-26 08:06:20)

转载 ▼

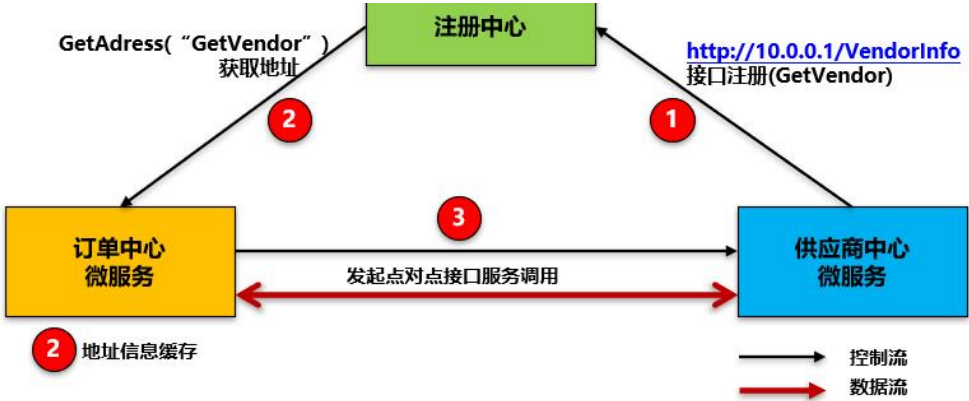
标签: 微服务 servicemesh

分类: 微服务架构



在前面文章我们已经介绍到，API网关本身仍然是中心化的，那么对API网关是否可能进一步去中心化处理，当我们将对该点进行思考的时候，发现去中心化后的API网关核心能力，更多的会迁移到ServiceMesh服务网格里面去实现。

API网关概述



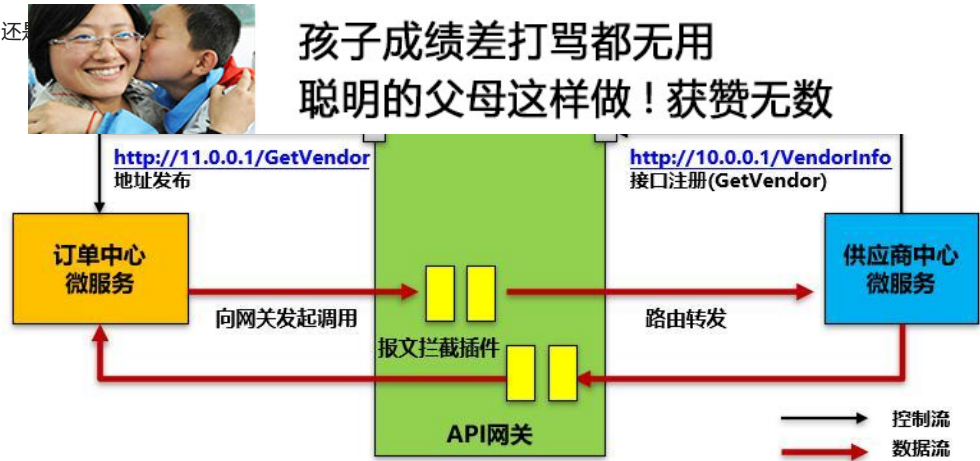
在该图上可以看到，服务注册中心核心能力为服务注册和发现，最多再承担一个负载均衡能力，而实际的接口服务调用仍然是点对点调用，即我们常说的整个消息的输入和输出数据流没有通过服务注册中心。

由于消息报文没有过注册中心，那么我们前面谈到的需要通过拦截实现的安全，流控，日志审计等能力自然无法实现。

在这种架构下为何可以称为去中心化的架构，主要体现在两点。

- 注册中心本身不走数据流，其性能负荷很小，能够承担高并发访问
- 地址信息往往可以在消费端缓存，因此注册中心短暂宕机并不影响接口访问

如果要对API接口实现更加完全的管控能力，即我们需要将API接口服务注册到API网关进行管理，通过API网关来进行日志，安全，限流熔断的统一管控治理。



从这个图可以看到最大的一个变化就是所有的数据流全部都会经过API网关的管道，因此我们才能够实现服务访问代理，各种安全，日志，流控的插件能力。

即在API网关中心化的架构下，最核心的即是上图里面的黄色小方块。这些黄色小方块可以理解为一个一个的独立拦截插件，通过这些插件来实现进一步的管控治理能力。

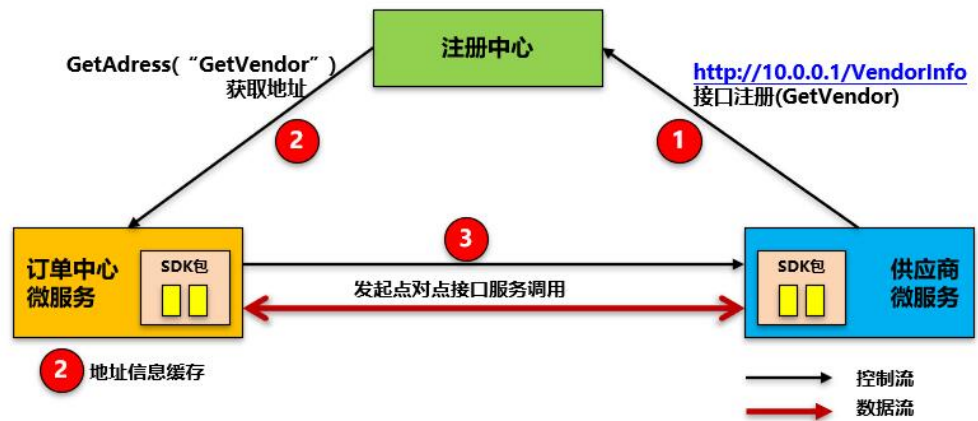
### 对API网关的去中心化处理

通过前面分析可以看到，服务注册中心本身是去中心化的，但是API网关本身是中心化，而API网关本身的中心化最重要的又是对API接口报文的消息拦截，来实现安全，日志，限流等各种接口管控要求。

那么如何既能够实现去中心化，又能够满足对API接口的治理管控要求？

自然，我们最重要想到的就是将服务注册中心和API网关两者的能力结合，既我们可以将上图API网关中各种拦截插件的能力下沉到微服务模块中去实现，在各个微服务模块中配置一个SDK包来完成各种管控能力。

讲到这里，大家可能会看到，这个实际和ServiceMesh里面谈到的SideCar完全是一个道理，既管控能力下沉到各个微服务模块中去，实现彻底的去中心化。



基于上图可以看到，我们将API网关的核心能力打包为一个独立的SDK包，类似SideCar的方式下沉到各个微服务模块中去。在前面的文章里面我们曾经谈到过，去中心化的一个核心思路就是在微服务架构实施过程中，结合容器化自动化部署过去，在微服务模块部署的时候，自动在容器里面同时部署相应的SDK代理包。

这个SDK代理包将传统的网关能力下沉到微服务模块本地化容器中去实现和执行。在这个构图中我们仍然保留服务注册中心来实现服务注册发现，同时本地的SDK服务代理包具备相应的缓存能力，即在服务注册中心宕机的时候仍然可以读取本地缓存信息进行服务地址获取和访问。

本地的SDK包是一个本地化的JAR包，我们可以在JAR包里面面对Http Rest接口的访问和消费进一步封装，将其封装为本地API方法，同时在本地API方法中进行类似安全，日志等方面内容的拦截处理。

在这种实现方法下，我们可以看下整个服务调用过程。



## 孩子成绩差打骂都无用 聪明的父母这样做！获赞无数

在整个过程中还可以在SDK包中添加安全，日志，流控等各种拦截器插件实现进一步的控制和处理。基于这个基础流程，我们来看实际的服务网关的关键能力实现。

### 服务代理能力

订单中心微服务模块只能看到本地SDK包的查询供应商API方法，而看不到具体的远程调用地址，而远程调用地址是SDK包通过动态查询服务注册库获取到后，再发起的调用。也就是说在去中心化架构下，传统服务网关的服务代理层前置到了本地微服务模块容器中的SDK代理包。

### 负载均衡和路由转发能力

SDK包中本身也不存储具体的远程服务访问地址，而是告诉服务中心我们需要访问哪个服务，服务中心根据服务编码或名称返回具体服务的访问地址，如果一个服务在服务中心注册有多个可访问地址，那么服务中心就可以实现最简单的负载均衡能力。即实际的负载均衡在服务注册中心实现。

第二种方法是SDK包调用服务注册中心后，服务注册中心把所有可访问的服务地址全部返回给SDK包，由SDK包在进行负载均衡算法选择和路由。

可以看到第一种方法往往更好，因为一个接口服务往往有多个微服务模块都在消费和调用，第二种方法是无法真正做到完全的负载均衡的。但是第一种方法仍然存在问题就是会增加服务注册中心的计算负荷。

### 日志和监控能力

在这种架构下，本地的SDK包是完全可以拦截到具体的服务输入和服务输出信息的，而且在这种去中心化的架构下还可以做到对于A模块和B模块各自本地的SDK包分别拦截到服务的输入和输出，以方便后续的日志审计。在SDK包拦截到服务日志后，在这里需要通过另外一个JMS API接口，将日志信息通过异步消息的模式写入到JMS消息中间件中，然后再通过消息中间件对日志进行相关的持久化处理。

如果写JMS消息中间件报错，我们可以对异常信息直接写入到本地的磁盘文件中，在JMS消息中间件恢复后再重新写回到消息中间件中。

### 限流和流量控制

首先还是需要在服务注册中心设置相应的限流策略，然后将限流策略分发到本地的SDK包中，本地的SDK包对服务调用次数，时间等信息进行计算统计，并将计算统计信息仅缓存，同时对缓存的数据进行实时计算，当满足流





做鼻子的后遗症



电动滑台



甲醛治理



modb



modbus网关

- 相关博文
- 科技主线再次来临！  
福哥看盘

每周运势提前报（8月31—9月6日）  
郑博士

唐山有一座可以吃的博物馆，游客  
落榜进士

孩子这个小动作暗含大隐患，很多  
洪兰谈育儿脑心理

了解树叶为什么会变红的原因  
艺圃叟

京城：国际光影艺术季万物共生一  
dysz大燕子

只谈理想不谈钱和只谈钱不谈理想  
明哥聊求职

人口负增长，正式进入倒计时了？  
杨国英观察

白色宽松的七分裤，搭配拼接色尖  
荒古棋盘

秦岭一白说历史人物：历史上真正  
秦岭一白的土蜂蜜
- 更多>>

- 推荐博文
- 

火山徒步奇遇记



跟着圣经去旅行



漂移赛车的发源地



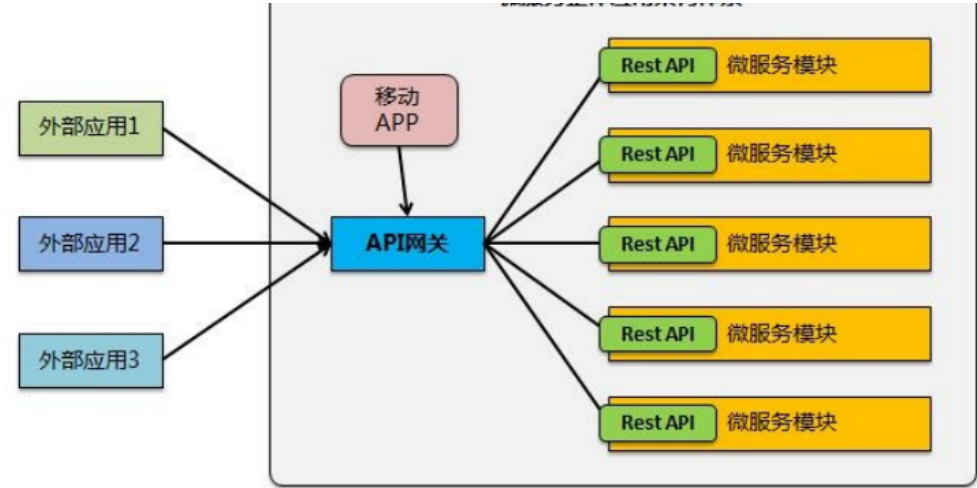
最美白色悬崖小镇



孤寂的欧洲最佳旅行目的地



等你邂逅的“老巷子”
- 查看更多>>



在微服务架构体系里面，我们一般会使用到微服务网关或叫API网关。

大家都比较清楚，在微服务架构体系下本身是去中心化的架构，通过服务注册中心来实现服务注册发现和消费调用，那么为何又需要使用API网关？

在传统的ESB总线进行服务集成的时候我们就经常谈到一个概念就是位置透明，即需要屏蔽底的业务模块提供API接口服务地址信息，并实现多个微服务API接口的统一出口。即类似设计模式里面经常谈到的门面模式。

**如何给API网关一个定义？**

简单来说API网关就是将所有的微服务提供的API接口服务能力全部汇聚进来，统一接入进行管理，也正是通过统一拦截，就可以通过网关实现对API接口的安全、日志、限流熔断等共性需求。如果再简单说下，通过网关实现了



### 孩子成绩差打骂都无用 聪明的父母这样做！获赞无数

从这里，我们就可以看到API网关本身仍然是中心化的架构，由于需要实现统一的日志，安全和限流熔断处理，那么所有的请求，消息输入输出全部需要进入API网格的总线管道。

那么对于微服务架构，为何我们经常会谈到是一种去中心化架构。

在前面我们也谈到过，微服务架构体系里面，API接口交互实际提供两种方式，一种是走服务注册中心的注册发现，一种是走API网格。而对于服务注册中心方式本身是一种完全去中心化的架构模式，即使注册中心宕机，往往也不会直接影响到API接口交互和服务运行。

对于API网关的详细描述可以参考前面的一篇文章：

一文详细讲解API网关核心功能和API管理扩展

## 服务注册中心和API网关实现对比

在这里，我们还是举例来说明两种方式的区别点。

比如我们当前有两个微服务模块，一个是订单中心，一个是供应商中心，订单中心在订单制作的过程中需要访问供应商中心提供的API接口服务获取供应商信息。

在这种场景下，如果是采用注册中心，整个交互过程如下。

由于在订单中心，供应商中心两个微服务模块我们都部署了服务代理SDK包，因此也很容易实现类似服务总线的输入端限流和提供端限流的双向限流能力。

服务访问安全控制

如果一个服务没有进行授权，首先可以做到SDK包在获取服务地址信息的时候就返回安全校验不通过，实现最基本的服务访问鉴权。

那我们再来看额外的服务访问控制场景，基础的做到的就是基于IP地址的服务访问控制，只有授权的IP地址才能够访问服务。在这里也是同样的道理，需要将IP访问控制策略信息下发到服务代理SDK包中。

同时去中心化的架构下我们也很容易实现类似动态Token的访问安全控制，比如在A模块，我们完全可以在A模块发起调用的时候通过传递业务系统ID，日期，再传递一个计算出来的MD5码值到服务提供端。提供端根据通用的算法也计算MD5码值，只有当两个值匹配的时候才认为校验通过。


在去中心化的架构下，可以看到仍然保留了中心化的服务注册中心，同时对于一些管控需要的基础配置元数据也需要在服务注册中心完成。对于各种配置元数据，我们需要实时下发到各个微服务模块。因此最好的做法技术在于各个微服务模块都实现一个JMS消息订阅接口，通过消息发布订阅的模式实时下发消息。

将去中心化的注册中心提升为控制中心

结合上个小节的内容，在去中心化的架构下，原来API网关中的各种拦截插件能力全部下沉到了各个微服务模块的SDK包中。但是SDK包中的日志，安全，限流等各种插件拦截能力，仍然还是需要有一个集中化的控制中心实现集中管控。

即我们常说的虽然数据流实现点对点，但是控制流仍然需要集中化。

其次，如果我们在去中心化的架构下仍然需要实现对Log日志的Trace监控能力，那么就需要对日志进行集中化的采集和存储，这个看起来好像破坏了前面谈到的去中心化思路，但是里面最关键的一点就是日志信息的采集本身通过消息中间件进行。



孩子成绩差打骂都无用  
聪明的父母这样做！获赞无数

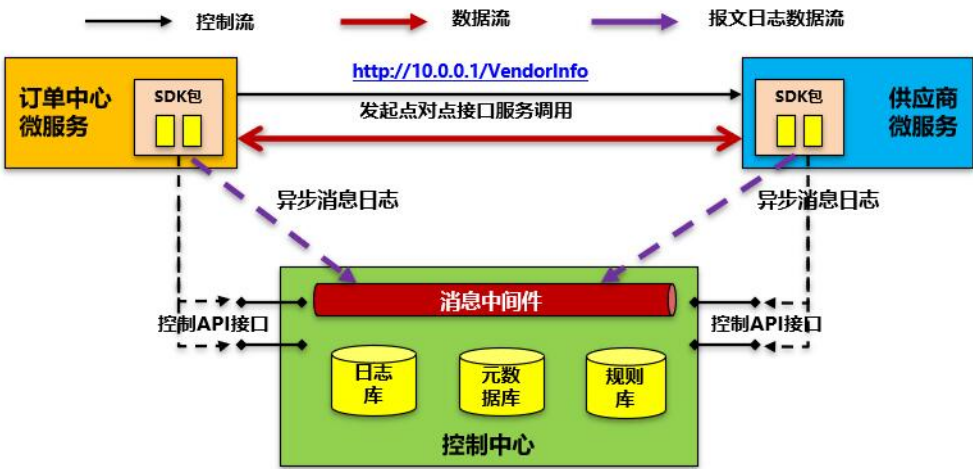
而到

限制熔断能力

而这些能力也就是我们常说的API网关核心的能力。在这里我们看到一个关键点，即下沉到微服务模块里面的SDK包更多的是进行执行和控制，但是相关规则策略的提供仍然是需要有一个统一的控制中心来进行提供。

控制中心提供这些能力，返回相应的控制策略给SDK包，SDK包执行控制。而控制中心提供这些能力本身即需要提供相应的元数据配置库，规则策略库。而对于日志审计的功能，还需要提供相应的分布式日志存储库。

因此我们重新构图如下：



到了这里我们基本形成一个去中心化的API网关雏形。

注：对于常规的微服务架构，比如SpringCloud整个架构体系，可以看到由于存在zuul等中心化的网关导致整个微服务架构仍然不是完全去中心化的。而到了SericeMesh服务化网关阶段，

看到上图已经和Istio整个架构思路很类似。

为了真正实现完全的去中心化，我们再来看下几个关键点。

消息日志的存储和查询

消息日志在微服务模块获取后通过消息中间件异步写入到持久化存储。如果消息中间件本身不可用，那么消息日志可以先写入到本地存储文件，待恢复后再重试写入。

限流和熔断

在前面我们谈到过限流熔断实现思路和滑动时间窗口，在去中心化架构下，可以是1到2小时的最终统计数据全部缓存在本地用于限流熔断控制规则计算，那么这种情况下即使控制中心出现问题也不会影响到限流熔断规则的执行。

安全策略控制

安全策略控制在通过API接口获取后，可以设置具体的失效时间，在失效前仍然是可以通过缓存获取，或者安全策略本身也可以在微服务模块本地存储文件进行临时存储。

完全去中心化的ServiceMesh



根据Linkerd CEO William Morgan定义，Service Mesh是用于处理服务间通信的基础设施层，用于在云原生应用复杂的服务拓扑中实现可靠的请求传递。

在实践中，Service Mesh通常是一组与应用一起部署，但对应用透明的轻量级网络代理。Service Mesh与传统基础设施的不同之处在于，它形成了一个分布式的互连代理网络，以sidecar形式部署在服务两侧，服务对于代理无感知，且服务间所有通信都由代理进行路由。

为什么需要Service Mesh

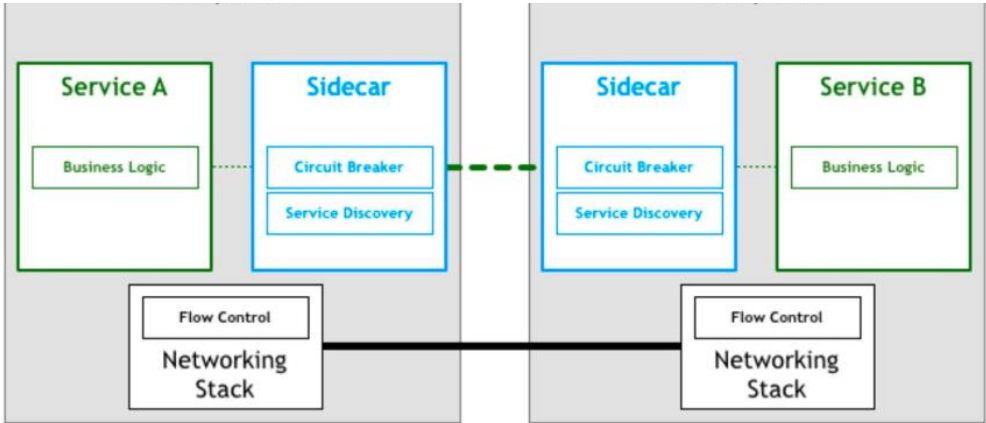
首先我们看下服务和进程间通信，除了完成基本的消息交互外，还需要关注服务注册发现，代理，路由，流量控制，安全，日志等共性的技术问题。而这些共性技术问题，在前面我们谈微服务架构的时候谈到过，这些不是服务注册中心能够全部解决的，更多的是依赖于微服务网关或API网关来解决这些共性问题。

通过API网关解决这些共性问题是当前的一种主流做法，而这种做法唯一的问题点在哪里呢？即API网关本身又变成了我们微服务架构中的一个中心点，即整个架构体系不再是完全去中心化的架构体系，API网关变成了一个中心点，那么就存在单点故障可能，存在性能瓶颈可能等。

而Service Mesh服务网格可以理解为API网关能力本身下沉到我们实际的服务两侧进行部署，即在部署微服务模块的机器上同时部署ServiceMesh服务网格能力，以达到完全去中心化的目的。

在服务发现模式我们曾经谈到过，一种是类似API网关来实现完全的集中化管理，还有就是客户端嵌入代理和主机独立进程代理模式。而Service Mesh网格是主机独立进程代理模式，即虽然和微服务模块一样部署在服务两侧，但是并不和微服务模块一起打包进行部署，是独立进程部署。即在一个Server上，可以多个微服务共享一个Service Mesh网格代理端。

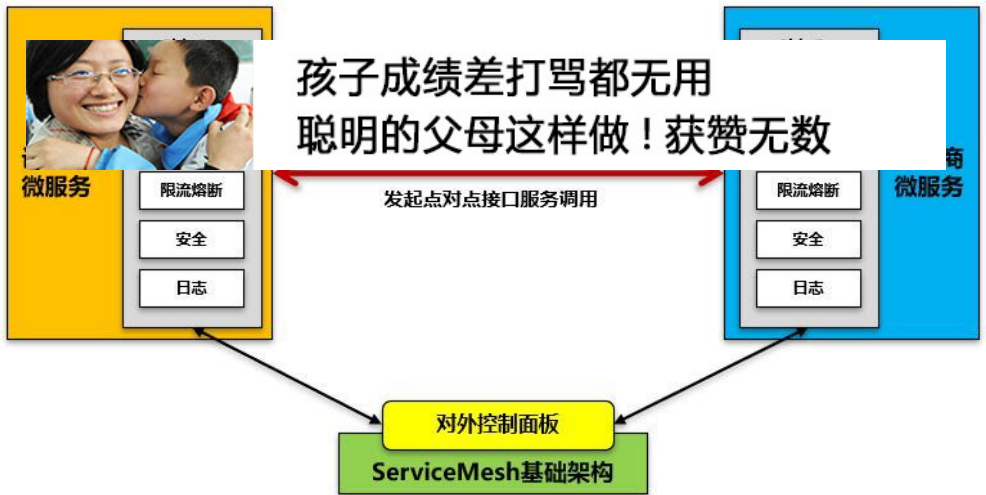




Service Mesh由data plane构成，其中所有服务通过sidecar代理进行服务通信。（所有代理相互连接形成一个Mesh，Service Mesh由此得名）网格同时包含一个control plane——可以将所有独立的sidecar代理连接到一个分布式网络中，并设置网格还包括一个控制平面——它将所有独立的sidecar代理连接到一个分布式网络中，并设置由data plane指定的策略。

对于ServiceMesh可以理解为我们部署一个或多个微服务模块的独立Server上，都会同时部署一个Sidecar代理，可以一个业务模块或多个业务模块共享一个代理。代理除了负责服务发现和负载均衡，还负责动态路由、容错限流、监控度量和安全日志等功能，这些功能在具体业务无关的。

可以看到ServiceMesh思路完全和我们前面谈到的API网关的去中心化思路是一致的，即将各种拦截插件能力下沉到微服务模块的SideCar里面。具体如下：



如果微服务模块全部是通过Kubernetes部署到Docker容器里面，那么我们可以看到完全可以在k8s进行镜像制作和容器部署的时候将SideCar的内容附加到具体的部署包里面实现集成。

简单来说，就是：

我们在开发微服务模块的时候完全不需要考虑太多的分布式API接口集成交互，但是和Kubernetes和Service Mesh集成后就具备了分布式接口调用和集成的能力。同时也具备了对API接口的安全，日志，限流熔断的管理能力。

因此也常说，Service Mesh是Kubernetes支撑微服务能力拼图的最后一块。

在为什么 kubernetes 天然适合微服务中我们提到，Kubernetes是一个奇葩所在，他的组件复杂，概念复杂，在没有实施微服务之前，你可能会觉得为什么Kubernetes要设计的这么复杂，但是一旦你要实施微服务，你会发现Kubernetes中的所有概念，都是有用的。在我们微服务设计的是个要点中，我们会发现Kubernetes都能够有相应的组件和概念，提供相应的支持。

设计要点一：API 网关	Ingress
设计要点二：无状态化，区分有状态的和无状态的应用。	无状态对应Deployment，有状态对应 StatefulSet
设计要点三：数据库的横向扩展。	headless service指向PaaS服务，或者StatefulSet部署
设计要点四：缓存	headless service指向PaaS服务，或者StatefulSet部署
设计要点五：服务拆分和服务发现	Service
设计要点六：服务编排与弹性伸缩	Deployment的Replicas
设计要点七：统一配置中心	ConfigMap
设计要点八：统一的日志中心	DaemonSet部署日志Agent
设计要点九：熔断，限流，降级	Service Mesh
设计要点十：全方位的监控	Cadvisor, DaemonSet部署监控Agent


其中最后的一块拼图就是服务发现，与熔断限流降级。

众所周知，Kubernetes的服务发现是通过Service来实现的，服务之间的转发是通过kube-proxy下发iptables规则来实现的，这个只能实现最基本的服务发现和转发能力，不能满足高并发应用下的高级服务特性，比较SpringCloud和Dubbo有一定的差距，于是Service Mesh诞生了，他期望将熔断，限流，降级等特性，从应用层，下沉到基础设施层去实现，从而使得Kubernetes和容器全面接管微服务。

在实施微服务架构中，如果需要对外暴露能力和交互，我们谈到了必须要使用API网关来实现能力的暴露以实现服务代理和服务透明。但是如果一个微服务架构不需要和外部打交道，同时仍然需要实现安全，日志和服务限流能力，那么我们就完全可以考虑采用ServiceMesh服务网格能力来实现，同时和Kubernetes和容器化技术实现集成。

ServiceMesh的开源实现Istio

Istio 是一个由谷歌、IBM 与 Lyft 共同开发的开源项目，旨在提供一种统一化的微服务连接、安全保障、管理与监控方式。Istio 项目能够为微服务架构提供流量管理机制，同时亦为其它增值功能（包括安全性、监控、路由、负载均衡等）提供基础支持。



## 孩子成绩差打骂都无用 聪明的父母这样做！获赞无数

「视  
规性

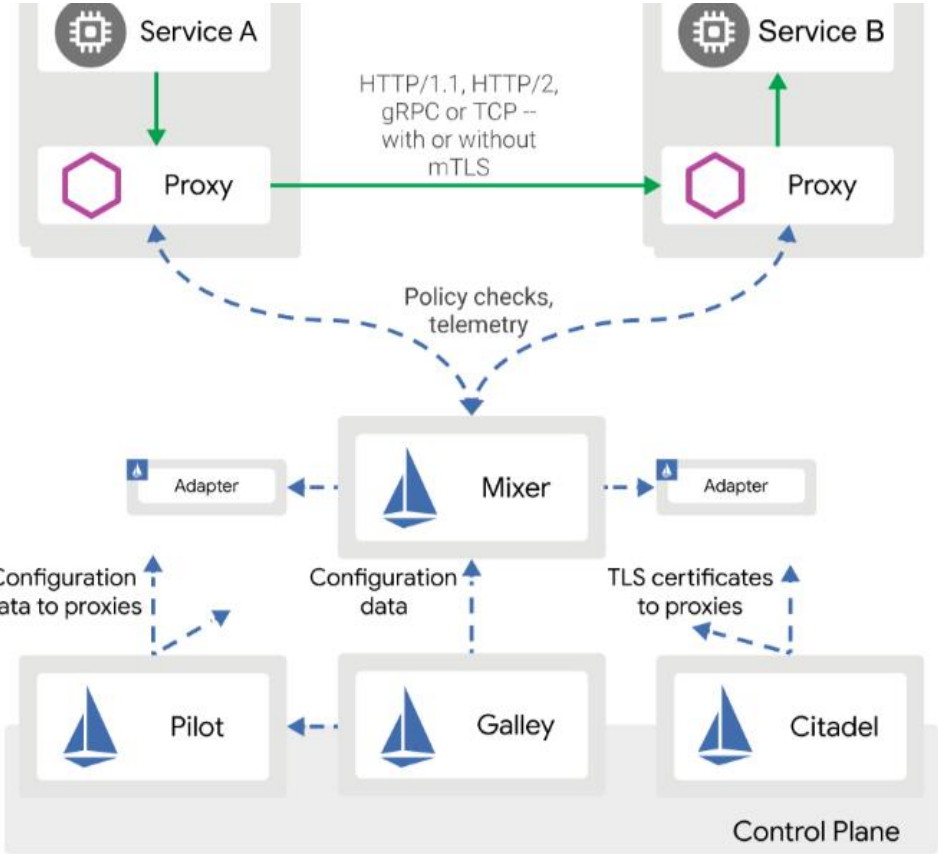
官方对 Istio 的介绍浓缩成了一句话：

An open platform to connect, secure, control and observe services.

翻译过来，就是“连接、安全加固、控制和观察服务的开放平台”。开放平台就是指它本身是开源的，服务对应的是微服务，也可以粗略地理解为单个应用。中间的四个动词就是 Istio 的主要功能，官方也各有一句话的说明。这里再阐释一下：

- 连接 (Connect)：**智能控制服务之间的调用流量，能够实现灰度发布等功能
- 安全加固 (Secure)：**自动为服务之间的调用提供认证、授权和加密。
- 控制 (Control)：**应用用户定义的 policy，保证资源在消费者中公平分配。
- 观察 (Observe)：**查看服务运行期间的各种数据，比如日志、监控和 tracing





## 孩子成绩差打骂都无用 聪明的父母这样做！获赞无数

控制中心做了进一步的细分，分成了 Pilot、Mixer 和 Citadel，它们的各自功能如下：

**Pilot：**为 Envoy 提供了服务发现，流量管理和智能路由（AB 测试、金丝雀发布等），以及错误处理（超时、重试、熔断）功能。用户通过 Pilot 的 API 管理网络相关的资源对象，Pilot 会根据用户的配置和服务的信息把网络流量管理变成 Envoy 能识别的格式分发到各个 Sidecar 代理中。

**Mixer：**为整个集群执行访问控制（哪些用户可以访问哪些服务）和 Policy 管理（Rate Limit，Quota 等），并且收集代理观察到的服务之间的流量统计数据。

**Citadel：**为服务之间提供认证和证书管理，可以让服务自动升级成 TLS 协议。

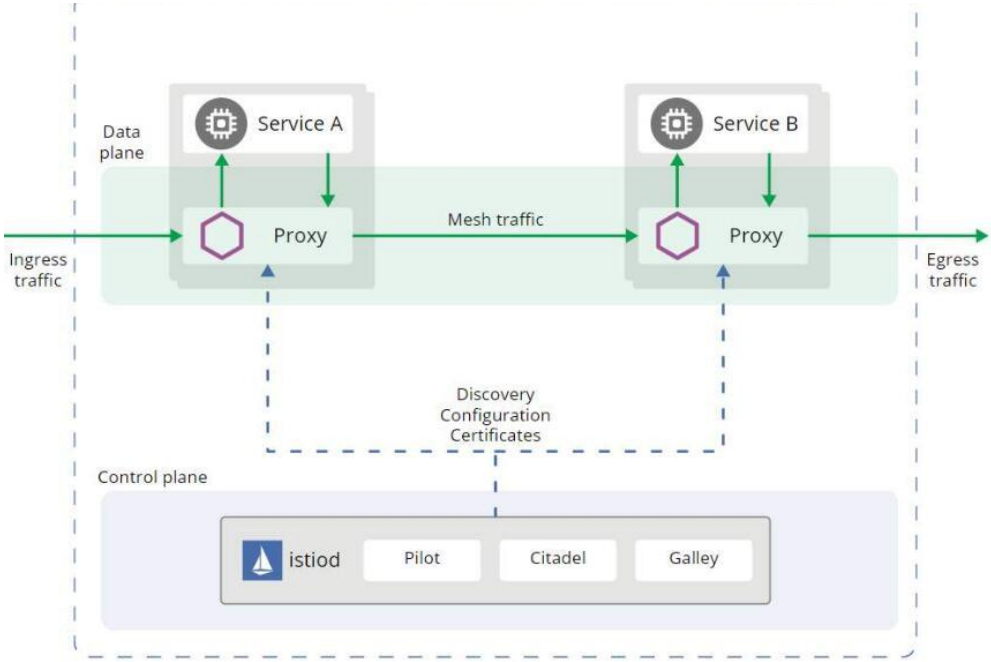
代理会和控制中心通信，一方面可以获取需要的服务之间的信息，另一方面也可以汇报服务调用的 Metrics 数据。

### 为什么使用 Istio？

通过负载均衡、服务间的身份验证、监控等方法，Istio 可以轻松地创建一个已经部署了服务的网络，而服务的代码只需很少更改甚至无需更改。通过在整个环境中部署一个特殊的 sidecar 代理为服务添加 Istio 的支持，而代理会拦截微服务之间的所有网络通信，然后使用其控制平面的功能来配置和管理 Istio，这包括：

- 为 HTTP、gRPC、WebSocket 和 TCP 流量自动负载均衡。
- 通过丰富的路由规则、重试、故障转移和故障注入对流量行为进行细粒度控制。
- 可插拔的策略层和配置 API，支持访问控制、速率限制和配额。
- 集群内（包括集群的入口和出口）所有流量的自动化度量、日志记录和追踪。
- 在具有强大的基于身份验证和授权的集群中实现安全的服务间通信。

Istio 为可扩展性而设计，可以满足不同的部署需求。



Istio 独立于平台，被设计为可以在各种环境中运行，包括跨云、内部环境、Kubernetes、Mesos 等等。您可以在 Kubernetes 或是装有 Consul 的 Nomad 环境上部署 Istio。

可以看到，Istio是一个与Kubernetes紧密结合的适用于云原生场景的Service Mesh形态的用于服务治理的开放平台。在我们实施微服务+DevOps+容器云的过程中，为了实现完全的去中心化，并增加对微服务模块，API接口服务的治理解耦能力，和Service Mesh生态进行结合，是必然的发展路径。



## 孩子成绩差打骂都无用 聪明的父母这样做！获赞无数

0

喜欢

0

赠金笔

分享：  
阅读(128) | 收藏(0) | 转载(0) | 喜欢▼ | 打印 | 举报/Report

前一篇：对IT项目售前解决方案的一些思考(200825)  
后一篇：从钉钉云服务到SaaS云生态(200827)



孩子成绩差打骂都无用  
聪明的父母这样做！获赞无数