

## 背景

---

目前我行已有多数系统转而使用微服务架构，即把一个单体应用拆分成若干个小型的服务，协同完成系统功能的一种架构模式，在系统架构层面进行解耦合，将一个复杂问题拆分成若干个简单问题，开发、维护、部署的难度就降低了很多，吞吐量和稳定性大大增加，且可以自主选择合适的技术框架，提高了项目开发的灵活性。然而，在转型过程中，由于银行业系统的多样性、复杂性，全部加入微服务行列是不现实的，新老系统共存是一种最为常见的现象。我行C与Java并存的系统并不在少数，而共存系统间的治理、运维等非常困难，很难做到统一维护、治理、监控等，在过度时期往往需要多个团队分而管之，维护难度很大。除此之外还会面临过于绑定技术栈，在面对这种异构系统时，需要花费大量精力来进行代码的改造，且在改造过程中会面临各种问题。

对于上述问题，现有的微服务架构无法规避，在业界技术的不断探索中：新一代微服务架构--服务网格应运而生。

## Istio同业使用情况

---

中国工商银行从2019年开始服务网格技术的预研工作，通过对服务网格技术深入研究和实践后，于2021年建设了服务网格平台。服务网格与现有微服务架构融合发展，助力工行应用架构向分布式、服务化转型，承载了未来开放平台核心银行系统。工行服务网格目前已完结多言语、异构技能、边缘场景的事务试点，基本论证服务网格在流量管控、体系扩展性的优势，具有下沉服务管理才能到根底设施层，高度解耦中间件与事务体系的可行性，是目前金融同业中最成熟的实践之一。

中国光大银行基于服务网格模式应用服务框架打造了企业分布式服务平台项目，目的是通过服务网格技术方向下的应用服务框架，避免现有集中式企业应用集成架构的中心交换服务瓶颈风险，提供服务化应用注册、发现及通讯的基础功能，形成企业级去中心化服务框架标准，支持企业为推进服务化转型所需的服务治理能力。项目采用服务网格技术领域事实标准的Istio开源框架作为项目分布式服务框架原型，开展集成开发。该项目于2018年启动，2019年投产，2020年正式进入推广阶段，截至目前已有86个应用服务运行在该项目支撑的分布式服务体系，覆盖范围正在快速增长。

中国民生银行基于多年SOA、分布式、微服务和云原生研发领域的经验积累和沉淀，已打造了技术栈统一、组件丰富、平台健壮、弹性扩展的云原生技术平台和能力体系，推动应用架构和技术架构持续转型和科技治理、赋能业务交付方面的能力不断提升，最近也正在进行Istio服务网格相关技术的研究和探索。

## 云原生技术

---

### 传统服务部署问题及演变

传统的tomcat部署：

```

<!-- A "Connector" represents an endpoint by which requests are received
and responses are returned. Documentation at :
Java HTTP Connector: /docs/config/http.html
Java AJP  Connector: /docs/config/ajp.html
APR (HTTP/AJP) Connector: /docs/apr.html
Define a non-SSL/TLS HTTP/1.1 Connector on port 8080
-->
<Connector port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
<!-- A "Connector" using the shared thread pool-->
<!--
<Connector executor="tomcatThreadPool"
           port="8080" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
-->
<!-- Define an SSL/TLS HTTP/1.1 Connector on port 8443
This connector uses the NIO implementation. The default
SSLImplementation will depend on the presence of the APR/native
library and the useOpenSSL attribute of the AprLifecycleListener.
Either JSSE or OpenSSL style configuration may be used regardless of
the SSLImplementation selected. JSSE style configuration is used below.
-->
<!--
<Connector port="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"
           maxThreads="150" SSLEnabled="true">
    <SSLHostConfig>
        <Certificate certificateKeystoreFile="conf/localhost-rsa.jks"
                    type="RSA" />
    </SSLHostConfig>
</Connector>

```

随着发展，springboot通过embedded tomcat方式改变了已有的打包方式，随着普及，目前大部分程序都通过jar方式启动，很少还有war包的形式了。

```

<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.37</version>
    <scope>compile</scope>
    <exclusions>
        <exclusion>
            <artifactId>tomcat-annotations-api</artifactId>
            <groupId>org.apache.tomcat</groupId>
        </exclusion>
    </exclusions>
</dependency>

```

```
server:
  tomcat

p tomcat
p server.tomcat.accept-count=100 (Maximum queue l... Integer
p server.tomcat.accesslog.buffered=true (Whether ... Boolean
p server.tomcat.accesslog.check-exists=false (Whe... Boolean
p server.tomcat.accesslog.condition-if (Whether lo... String
p server.tomcat.accesslog.condition-unless (Whethe... String
p server.tomcat.accesslog.directory=logs (Director... String
p server.tomcat.accesslog.enabled=false (Enable a... Boolean
p server.tomcat.accesslog.encoding (Character set ... String
p server.tomcat.accesslog.file-date-format=.yyyy-M... String
p server.tomcat.accesslog.ipv6-canonical=false (W... Boolean
p server.tomcat.accesslog.local (Locale used to f... String
Ctrl+向下箭头 and Ctrl+向上箭头 will move caret down and up in the editor Next Tip
```

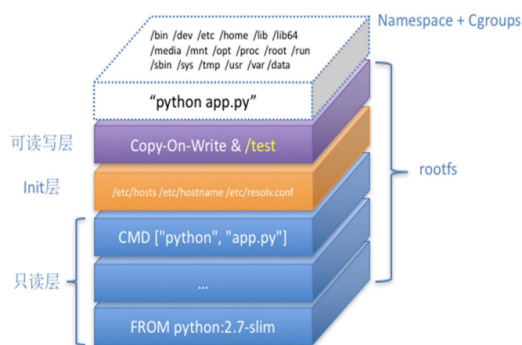
## 为什么使用Docker

开销更小：和传统的虚拟化技术不同，传统虚拟化增加了系统调节链的环节有性能损耗，而容器虚拟化共性内核，几乎没有性能损耗

提高可移植性：Docker可以确保你的应用程序与资源是分隔开的

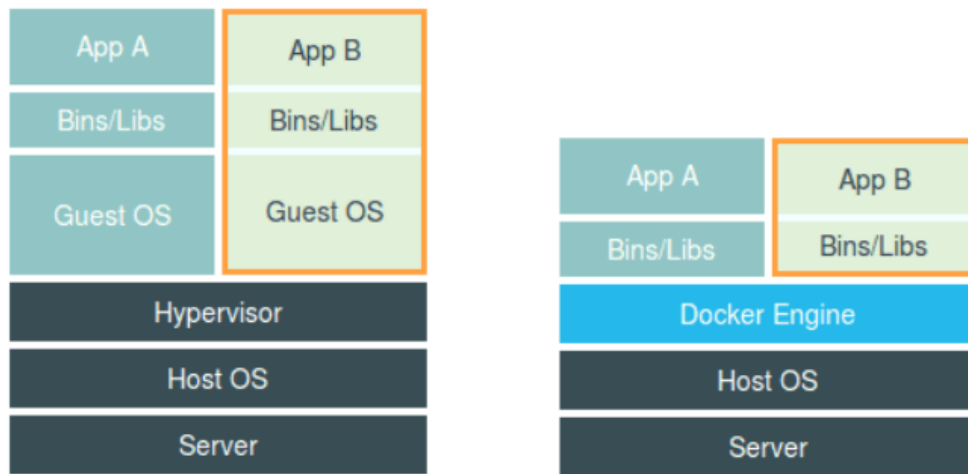
操作更加一致：“软件可以带环境安装”？也就是说安装的时候，把原始环境一模一样地复制过来，docker就是规定了环境的一致，保证迁移的时候打包、运行不走样

```
1 # 使用官方提供的Python开发镜像作为基础镜像
2 FROM python:2.7-slim
3
4 # 将工作目录切换为/app
5 WORKDIR /app
6
7 # 将当前目录下的所有内容复制到/app下
8 ADD . /app
9
10 # 使用pip命令安装这个应用所需要的依赖
11 RUN pip install --trusted-host pypi.python.org -r requirements.txt
12
13 # 允许外界访问容器的80端口
14 EXPOSE 80
15
16 # 设置环境变量
17 ENV NAME World
18
19 # 设置容器进程为: python app.py, 即: 这个Python应用的启动命令
20 CMD ["python", "app.py"]
```



## Docker浅析

### 虚拟机与docker对比图



通过图片可以明显看到两边的差异在Hypervisor和Docker Engine。

**Hypervisor**通过硬件虚拟化功能，模拟出了运行一个操作系统需要的各种硬件，比如 CPU、内存、I/O 设备等等。然后，它在这些虚拟的硬件上安装了一个新的操作系统，即 Guest OS。

**Docker Engine**在Linux系统中可以理解为一个进程，没错每一个容器都是一个进程。docker通过 **Namespace 配置，配置Cgroups 参数，Change Root调整根目录来实现一个容器。**

Namespace：可以简单理解为访问权限控制。举个例子如果Linux系统为我们的软件研发中心，那么诺德中心、总部基地等为一个个进程，我们为每一个进程启用Namespace可以理解为止办公场地之间人员互相访问，这样诺德场地的人就只能调用诺德的资源，由此实现了资源的隔离。

namespace 种类	namespace作用
UTS namespace	UTS namespace 对主机名和域名进行隔离
IPC namespace	IPC namespace 针对 System V 和 POSIX 消息队列，这些 IPC 机制会使用标识符来区别不同的消息队列
PID namespace	隔离进程号，不同namespace 的进程可以使用相同的进程号
Mount namespace	隔离文件挂载点，每个进程能看到的文件系统都记录在 /proc/\$\$/mounts里
Network namespace	隔离网络资源。每个 namespace 都有自己的网络设备、IP、路由表、/proc/net 目录、端口号
User namespace	隔离用户和用户组。可以让宿主机上的一个普通用户在 namespace 里成为 0 号用户，也就是 root 用户，影响范围也在容器内

Cgroups：通过Namespace我们知道容器之间是相互隔离的互相没有影响，但是容器就是Linux的一个进程，如果某一个进程疯狂吞噬操作系统的资源，可能会造其他容器无法运行。Cgroups是 Control Group。它最主要的作用，就是限制一个进程组能够使用的资源上限，包括 CPU、内存、磁盘、网络带宽等等。

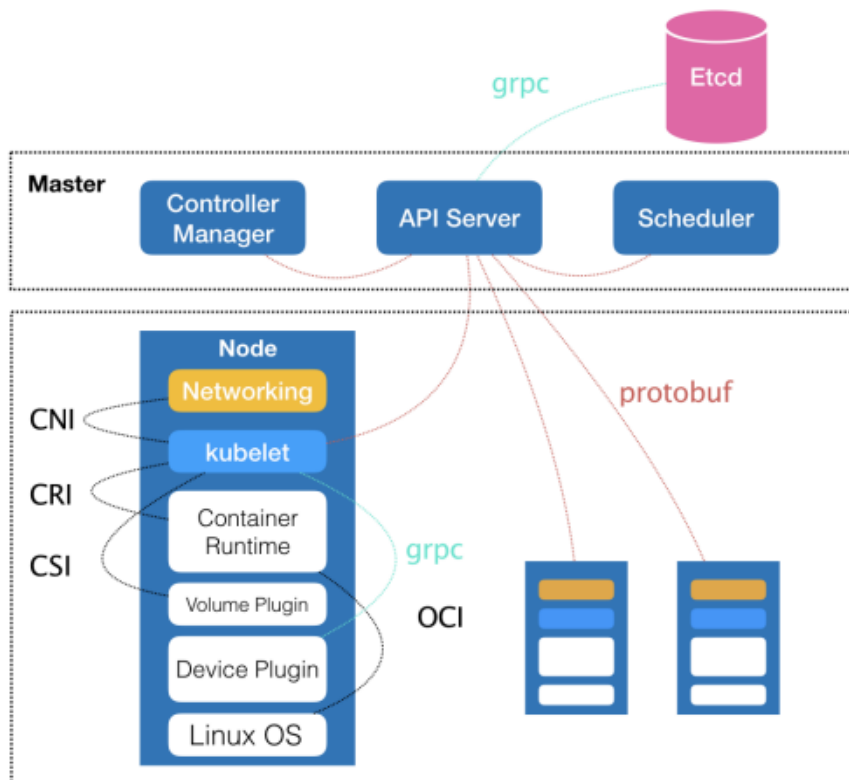
Cgroups子系统	具体作用
Devices	控制Cgroup的进程对哪些设备有访问权
cpuset子系统	cpuset可以为一组进程分配指定的CPU和内存节点
memory子系统	限制Cgroup所能使用的内存上限
cpu子系统	cpu子系统用于限制进程的CPU利用率
cpuacct子系统	统计各个Cgroup的CPU使用情况
blkio	限制Cgroup对阻塞IO的使用

rootfs: 作为一个沙箱环境，资源应该也是隔离的，每当创建一个新容器时，希望容器进程看到的文件系统就是一个独立的隔离环境，而不是继承自宿主机的文件系统。通过mount namespace（特殊的namespace，用于挂载）和容器镜像（rootfs）两种技术实现，mount namespace将根目录挂载到指定目录，rootfs用于恢复镜像内容。

## 为什么使用Kubernetes

用过docker的人都知道docker启动命令相当粗暴，比如说我们要启动一个容器就是docker run ...，从字面意思非常好理解。但是我们思考一下docker除了带给我们沙箱的环境和镜像的便利，其他的鞭长莫及。现在微服务动辄几百台节点，可能工作量并没有变化。2014年Kubernetes（下面简称k8s）发布第一个版本，集容器编排、调度、管理与一体。如果docker的交互方式是面向过程，k8s就是面向对象。那么k8s如何实现的呢？

## Kubernetes浅析



k8s之所以可以管理docker容器，重点在于图中的CRI，它屏蔽了具体的容器实现，提供了一层抽象来管理容器。在用户输入一条指令后，API Server接收指令解析后存储到etcd，Scheduler监听到了变更后选取容器需要去往的node节点等信息存储到etcd中，Controller Manager通过监听数据变化，调用API Server发送请求到kubelet，kubelet执行具体和容器、操作系统的交互。我们只需要告诉k8s我们需要的，k8s会协调资源帮助我们实现，这就是“声明式API”。

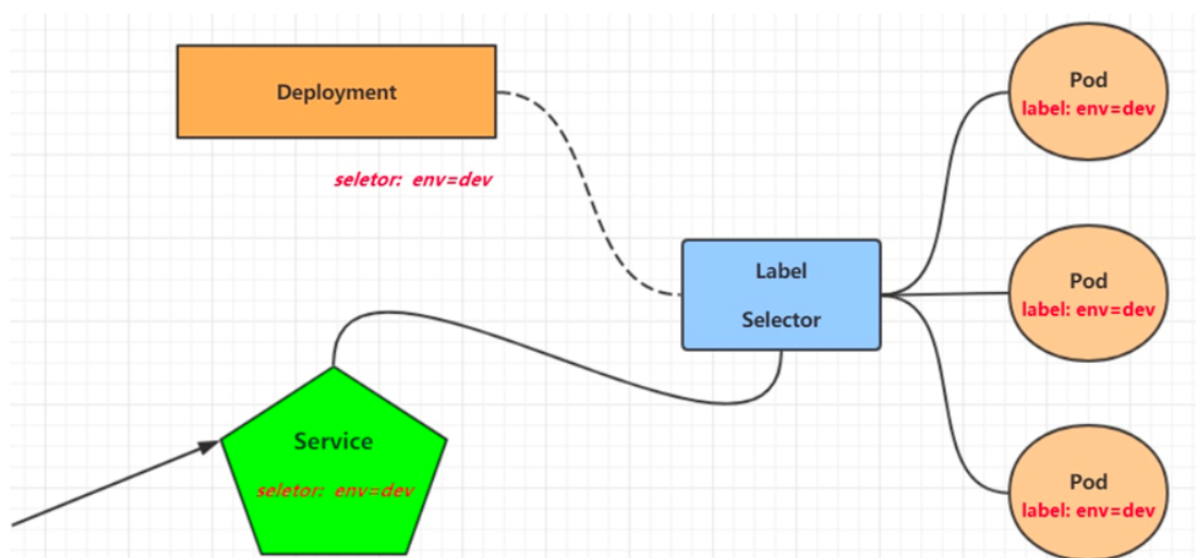
- 1、数据卷：Pod中容器之间共享数据，可以使用数据卷
- 2、应有程序健康检查：容器内服务可能进程堵塞无法处理请求，可以设置监控检查策略保证应用健壮性
- 3、复制应用程序实例：控制器维护着Pod副本数量，保证一个Pod或一组同类的Pod数量始终可用
- 4、弹性伸缩：根据设定的指标（CPU利用率）自动缩放Pod副本数
- 5、服务发现：使用环境变量或DNS服务插件保证容器中程序发现Pod入口访问地址
- 6、负载均衡：一组Pod副本分配一个私有的集群IP地址，负载均衡转发请求到后继容器。在集群内部其他Pod可通过这个ClusterIP访问应用
- 7、滚动更新：更新服务不中断，一次更新一个Pod，而不是同时删除整个服务
- 8、服务编排：通过文件描述部署服务，使的应用程序部署变得更高效
- 9、资源监控：Node节点组件集成cAdvisor资源收集工具，可通过Heapster汇总整个集群节点资源数据，然后存储到InfluxDB时序数据库，再由Grafana展示
- 10、提供认证和授权：支持角色访问控制（RBAC）认证授权等策略

虽然每个Pod都会分配一个单独的Pod IP，然而却存在如下两问题：

- Pod IP 会随着Pod的重建产生变化
- Pod IP 仅仅是集群内可见的虚拟IP，外部无法访问

这样对于访问这个服务带来了难度。因此，k8s设计了Service来解决这个问题

Service可以看作是一组同类Pod对外的访问接口。借助Service，应用可以方便地实现服务发现和负载均衡



## 为什么使用Istio

你一直在说部署层面和运维的优势，和我们开发有什么关系？现在我们说一说和业务开发有关系的侵入式微服务面临的挑战

- 1、异构困难：不同语言的复用和集成困难

- 2、流量管理复杂：需要引入大量第三方工具进行流量管理，实现细粒度的流量控制困难
- 3、非功能性需求耦合度高：日志记录和追踪等非功能性需求与业务代码耦合
- 4、缺乏弹性伸缩：对应用的弹性伸缩支持不足
- 5、安全性不足：对服务通信认证、授权和加密缺乏有力支持
- 6、开发测试的复杂性：分布式系统的编程难度更大、测试更复杂
- 7、运维的复杂性：需要成熟的运维团队来管理和组织众多的微服务

### 场景举例：

验证服务是否健康

```
curl localhost:30009/health
```

张三是我行VIP客户，李四是普通客户（灰度发布）

```
curl localhost:30009/abTest?username=zhangsan
```

升级后分拨小部分流量到新版本,多版本并行（流量分拨）

```
curl localhost:30009/routeControl
```

在正常的服务间调用，动态添加请求头（请求头添加）

```
curl localhost:30009/headerAdd
```

请求故障注入（配合测试）

```
curl localhost:30009/errhealth
```

请求熔断（配合测试）

```
curl localhost:30009/errTest
```

### 源代码：

```
@RestController
public class ServiceController {

    @RequestMapping("/health")
    public String health() { return "success"; }

    @RequestMapping("/abTest")
    public String abTest(String username) { return username + " : " + System.getProperty( key: "version", def: "default"); }

    @RequestMapping("/routeControl")
    public String routeControl() {
        return System.getProperty( key: "version", def: "default");
    }

    @RequestMapping("/headerAdd")
    public String headerAdd(HttpServletRequest request) { return request.getHeader( name: "headerAdd"); }

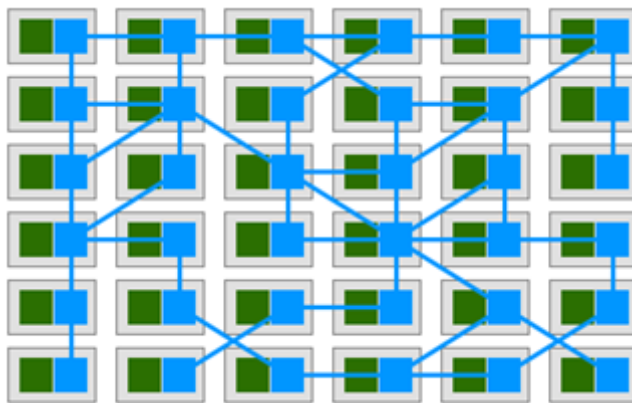
    @RequestMapping("/errTest")
    public String error() { throw new RuntimeException("errTest"); }
}
```



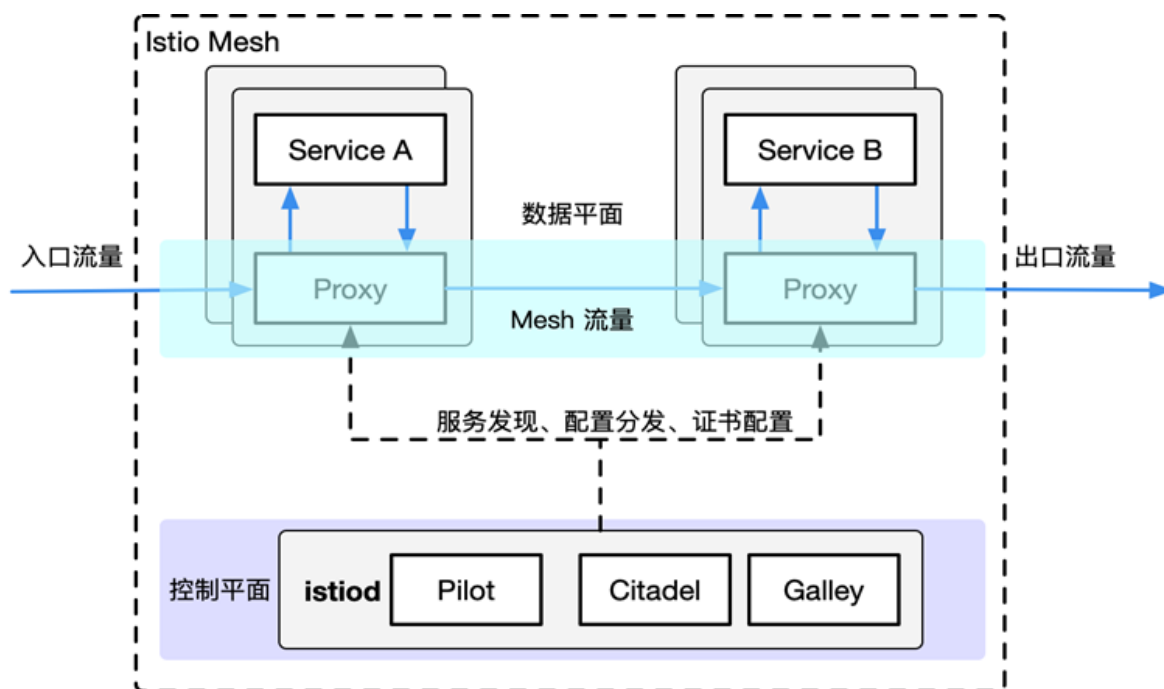
## Istio浅析

服务网格（Service Mesh）是一个专门处理服务通讯的基础设施层。它的职责是在由云原生应用组成服务的复杂拓扑结构下进行可靠的请求传送。其从总体架构上来讲比较简单：一组服务的用户代理，加上一组任务管理组件组成。

1. 用户代理在服务网格中被称为数据层或数据平面（data plane），直接处理入站和出站数据包，包括转发、路由、健康检查、负载均衡、认证、鉴权、产生监控数据等。
2. 管理组件被称为控制层或控制平面（control plane），负责与控制平面中的代理进行通信、下发策略和配置。



相较于传统微服务架构，服务网格不再需要服务网关、注册中心、负载均衡和流控等传统微服务架构所需的组件，以更轻量级的架构支持微服务间的网络通信和服务治理。其中Istio是服务网格的典型实现，目前需要依赖于k8s环境。以下是对Istio的架构及组件的简单介绍。



- Istio选择Envoy作为代理，其本质是一个为面向服务的架构而设计的代理和通信总线。
- Pilot是Istio实现流量管理的核心组件，它主要的作用是配置和管理 Envoy代理。
- Citadel是与安全相关的组件，主要负责密钥和证书的管理。
- Galley作为独立组件承担Istio的配置管理工作，负责配置的获取、处理和分发。

所以对于Istio的能力我们可以简单的用四个字概括（个人看法）：**流量治理**



# Istio为什么这么火

## 1.性能好，功能强大

上面介绍了Istio的数据平面组件是**Envoy**，控制平面（Istiod）主要由Pilot、Citadel、Galley组成（精简后），由于Istiod的功能很容易理解所以我们重点说一下Envoy，Envoy是cncf基金会第三个毕业的项目，前两个是k8s和Prometheus。Envoy出身名门性能也非常优秀。其优势：

**L3/L4 过滤器架构：** Envoy 的核心是一个 L3/L4 网络代理。可插入的过滤器链机制允许编写过滤器来执行不同的 TCP/UDP 代理任务并插入到主服务器中。已经编写了过滤器以支持各种任务，例如原始TCP代理、UDP 代理、HTTP 代理、TLS 客户端证书身份验证、Redis、 MongoDB、Postgres等。

**HTTP L7 过滤器架构：** HTTP 是现代应用程序架构的关键组件，Envoy支持额外的 HTTP L7 过滤器层。HTTP 过滤器可以插入 HTTP 连接管理子系统，执行不同的任务，如缓冲、速率限制、路由/转发、嗅探亚马逊的DynamoDB等。

**一流的 HTTP/2 支持：** 在 HTTP 模式下运行时，Envoy支持HTTP/1.1 和 HTTP/2。Envoy 可以作为透明的 HTTP/1.1 到 HTTP/2 双向代理运行。这意味着可以桥接 HTTP/1.1 和 HTTP/2 客户端和目标服务器的任意组合。推荐的服务到服务配置在所有 Envoy 之间使用 HTTP/2 来创建一个持久连接网格，请求和响应可以在该网格上多路复用。

**HTTP/3 支持（目前处于 alpha 阶段）：** 从 1.19.0 开始，Envoy 现在支持 HTTP/3 上游和下游，并在 HTTP/1.1、HTTP/2 和 HTTP/3 的任意组合之间进行双向转换。

**HTTP L7 路由：** 在 HTTP 模式下运行时，Envoy 支持 路由子系统，该子系统能够根据路径、权限、内容类型、运行时值等路由和重定向请求。当使用 Envoy 作为前端/边缘时，此功能最有用代理，但在构建服务网格服务时也会被利用。

**gRPC 支持：** gRPC是来自 Google 的 RPC 框架，它使用 HTTP/2 或更高版本作为底层多路复用传输。Envoy支持用作 gRPC 请求和响应的路由和负载平衡底层所需的所有 HTTP/2 功能。这两个系统非常互补。

**服务发现和动态配置：** Envoy 可选择使用一组分层的 动态配置 API进行集中管理。这些层为 Envoy 提供以下动态更新：后端集群中的主机、后端集群本身、HTTP 路由、侦听套接字和加密材料。对于更简单的部署，可以通过 DNS 解析（甚至 完全跳过）来完成后端主机发现，并将更多层替换为静态配置文件。

**健康检查：** 构建 Envoy 网络的推荐方法是将服务发现视为最终一致的过程。Envoy 包括一个健康检查子系统，它可以选择性地对上游服务集群执行主动健康检查。然后，Envoy 使用服务发现和健康检查信息的结合来确定健康的负载均衡目标。Envoy 还支持通过异常值检测子系统进行被动健康检查。

**高级负载均衡：** 分布式系统中不同组件之间的负载均衡是一个复杂的问题。因为 Envoy 是一个自包含的代理而不是一个库，所以它能够在 一个地方实现高级负载均衡技术，并让任何应用程序都可以访问它们。目前，Envoy 包括对自动重试、熔断、通过外部速率限制服务进行 全局速率限制、请求阴影和异常值检测的支持。

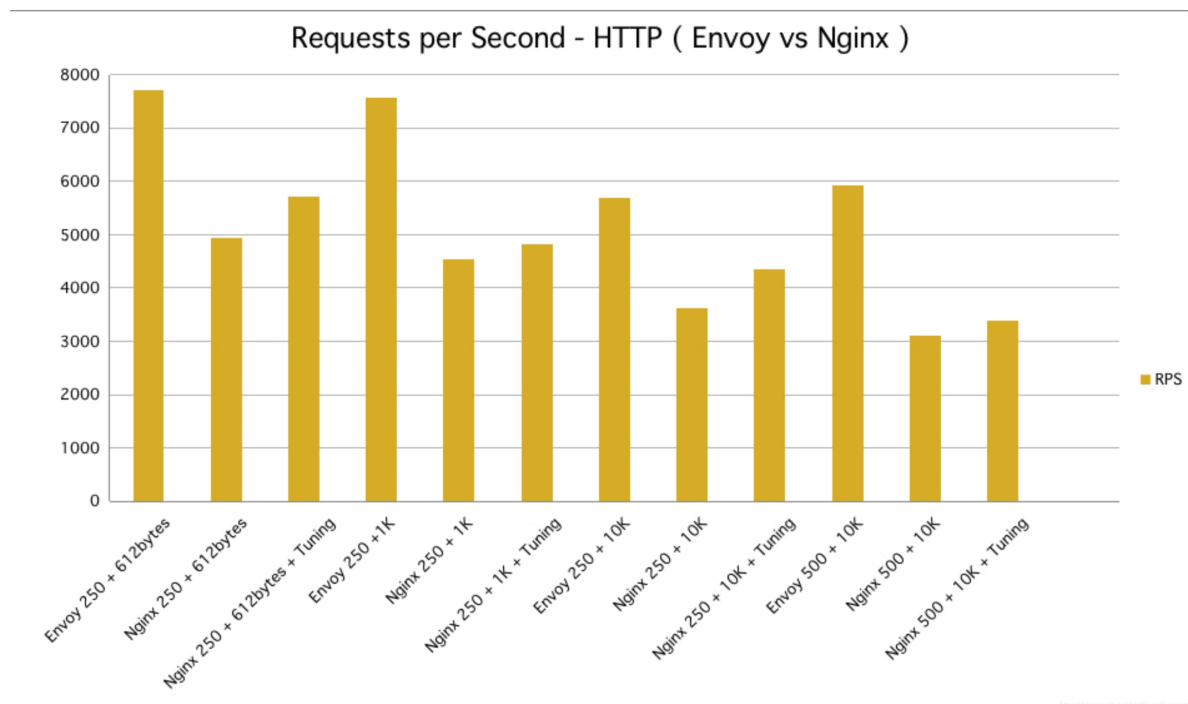
**前端/边缘代理支持：** 在边缘使用相同的软件有很大的好处（可观察性、管理、相同的服务发现和负载均衡算法等）。Envoy 具有一个功能集，使其非常适合作为大多数现代 Web 应用程序用例的边缘代理。这包括TLS终止、HTTP/1.1 HTTP/2 和 HTTP/3支持，以及 HTTP L7路由。

**一流的 可观察性：** 如上所述，Envoy 的主要目标是使网络透明。然而，问题出现在网络层面和应用层面。Envoy 包括对所有子系统的强大统计支持。statsd（和兼容的提供者）是当前支持的统计接收器，尽管插入不同的接收器并不困难。也可以通过管理端口查看统计信息。Envoy 还支持 通过第三方提供商进行分布式跟踪。

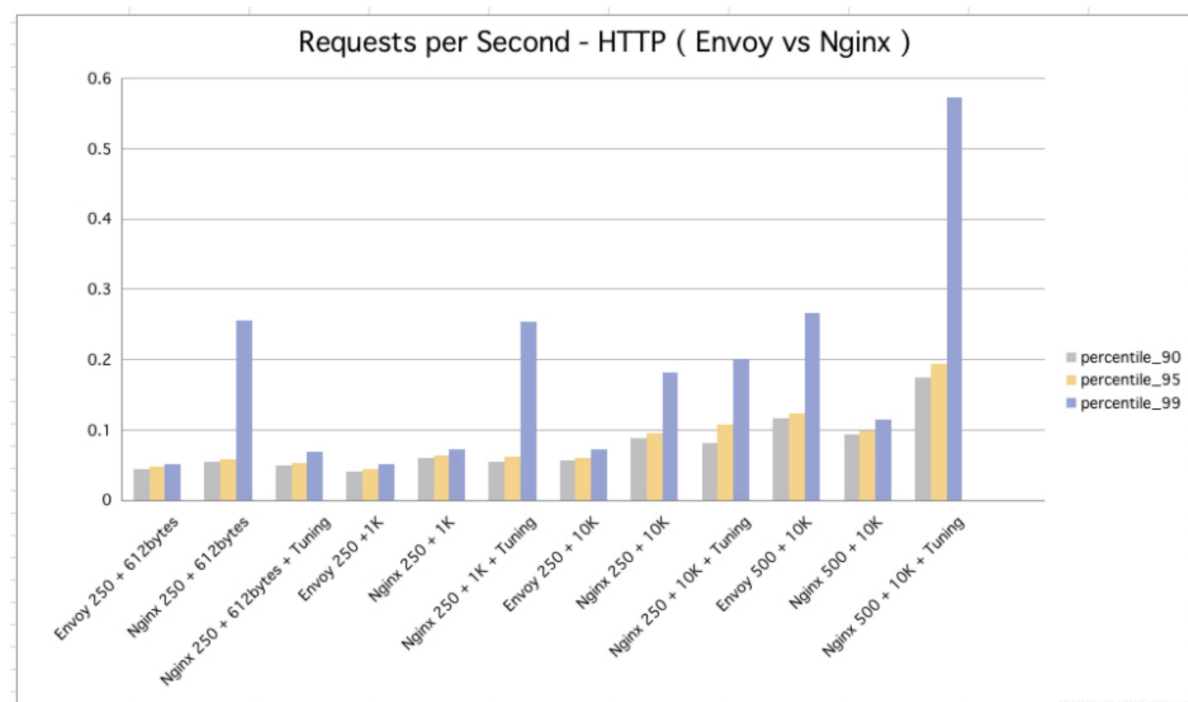
Istio可以无缝整合Jaeger、Kiali、Prometheus、Tracing、Zipkin等，这些组件提供了Istio的调用链、监控等功能，可以选择安装来完成完整的服务监控管理功能。

Envoy 与 Nginx 的基准性能对比数据：

## 吞吐量RPS对比:



## 响应时间RT 90值、95值、99值数据对比:



## 2.契合容器云环境

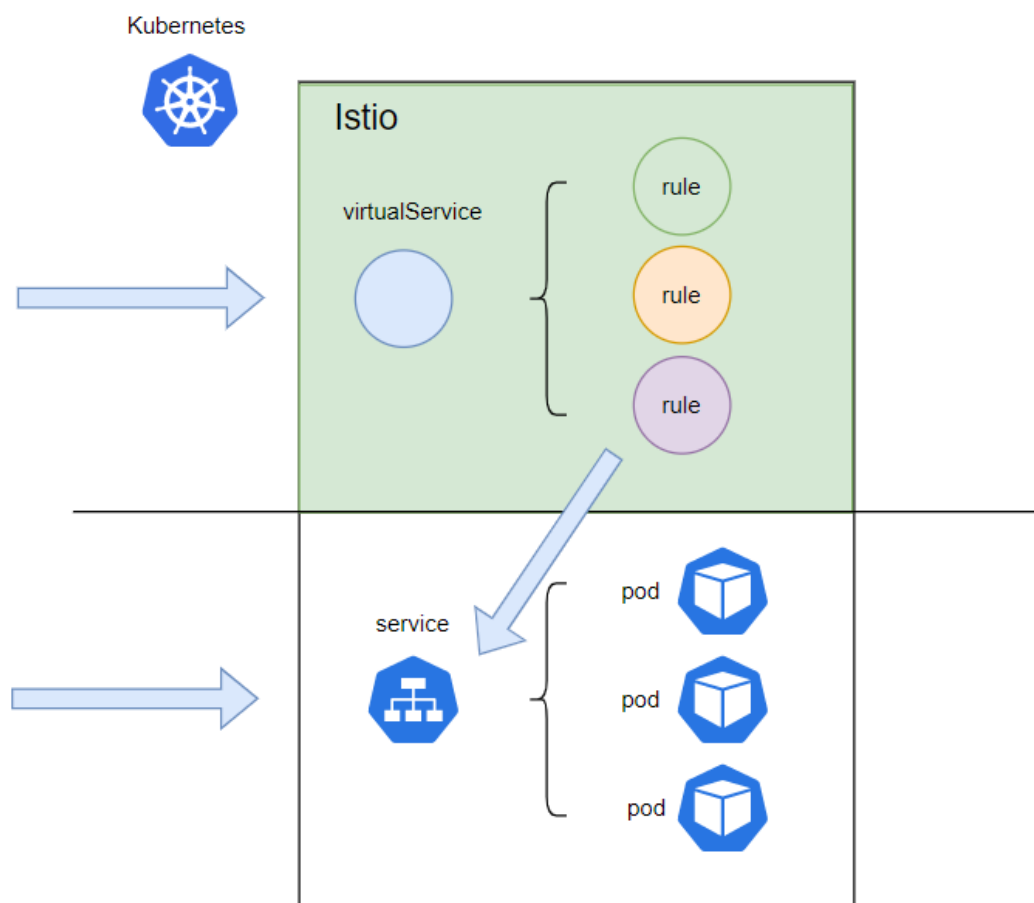
上面介绍了Istio的相关组件和功能，在性能方面固然强大，但是部署和使用的难易程度也是我们需要考量的另一个重要指标。

假设我们已经基于k8s来部署服务（没意见吧！）

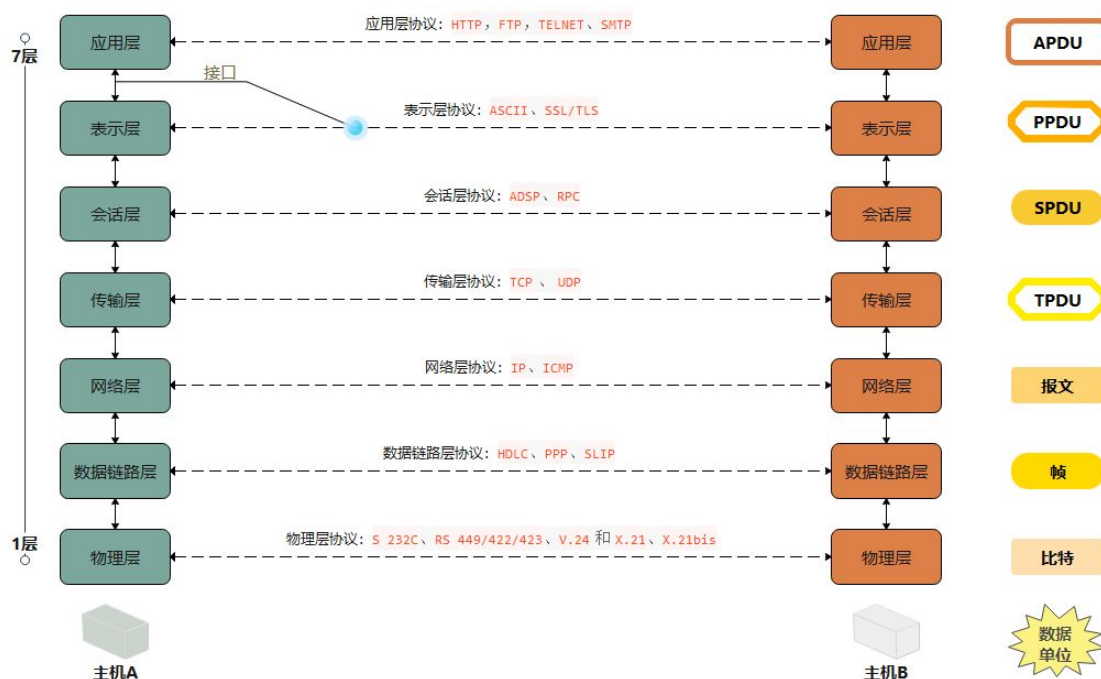
在k8s中我们想实现一些流量控制，我们可以继续使用hystrix和sentinel（java语言），但是我们需要为每一个服务嵌入这些组件，然后编写对应的逻辑处理。如果用sentinel的话虽然可以拥有远程下发配置的能力，但是我们需要额外部署控制台资源，还需要为这些交互打通网络关系。最主要的是我们需要熟悉这些组件才可以应用到系统中，那如果是多个语言组成的系统呢。。。

当然我们也可以自己编写组件，和应用服务部署在同一个pod里面，然后通过iptables来代理所有的流量，然后通过编写控制台动态下发配置。这个方式和Istio完全一致，并且我们不会比它做的更好。

通过Istio应用可以无感知嵌入数据平面控制组件（Envoy），同时还可以代理外部应用在网格内的流量管理，动态刷新这些配置，完美的复用了k8s的能力，并且可以无需额外编码仅通过配置文件便可以实现流量的管理。



## OSI参考模型

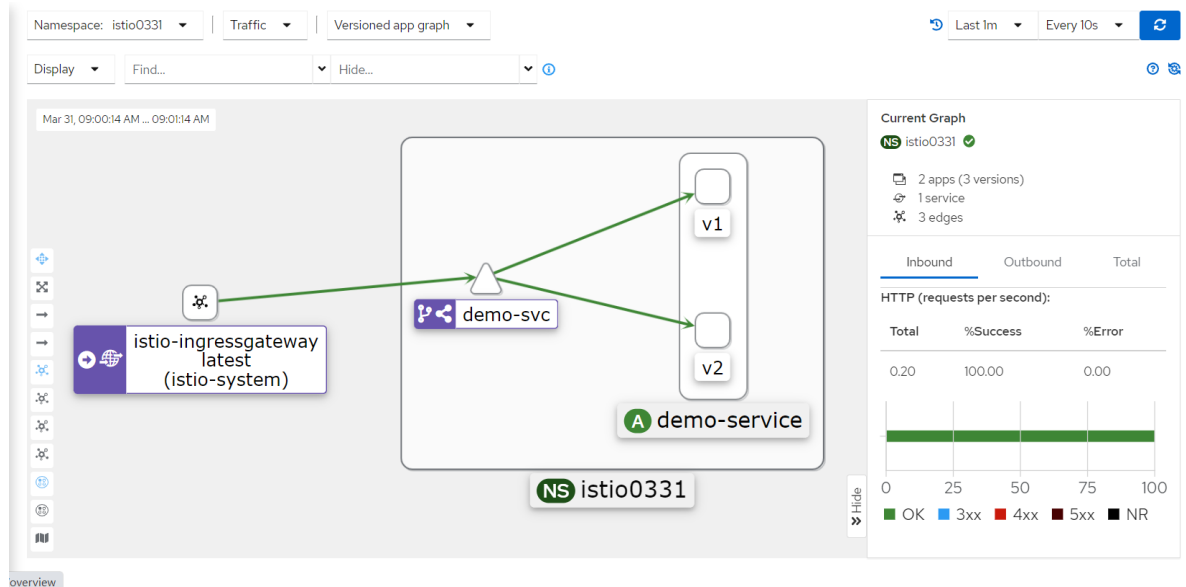


### 3.跨语言与解耦

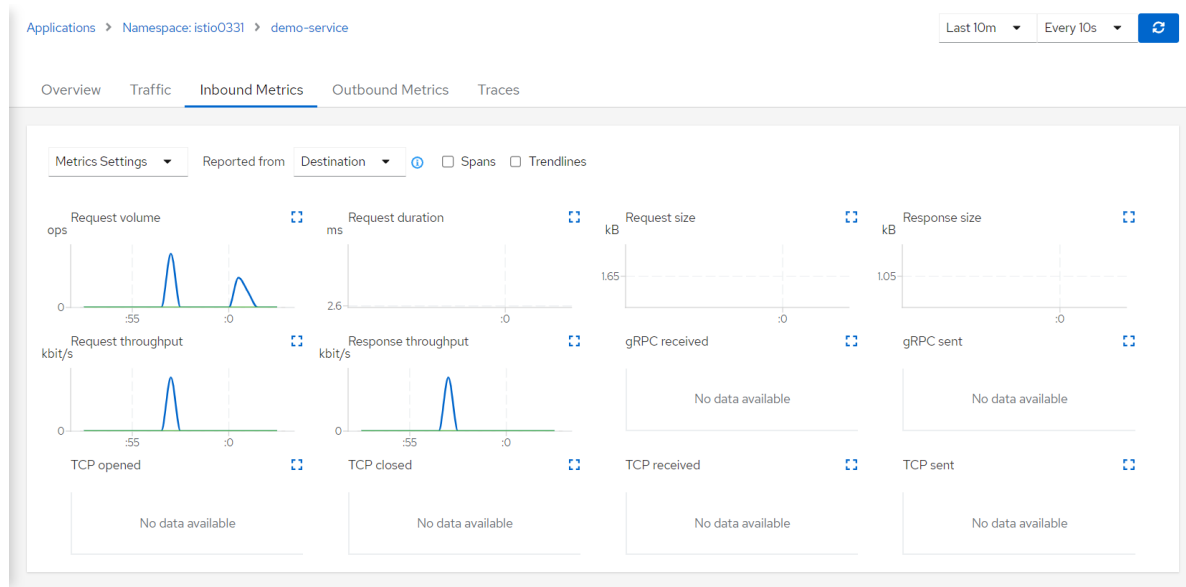
Istio不关心语言实现，因为Istio通过iptables拦截了所有流量转发给Envoy，Envoy过滤流量后再转发给业务模块。将服务之间、服务与集群外部的网络通讯和安全机制从微服务的业务逻辑中解耦，并作为一个与平台无关的、独立运行的程序，以减少开发和运维人员的工作量。

### 4.强大的监控能力

链路追踪能力



流量监控能力



<http://43.143.183.122:30287/kiali/>

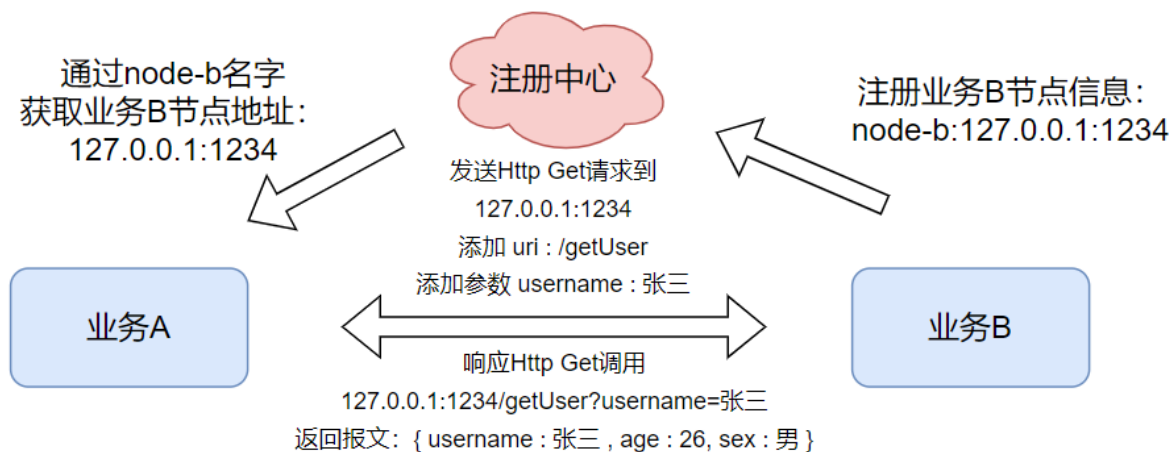
服务网格介绍结束，下面是实战部分，下面的配置文件较多，关联性较强，如有讲得不好的地方请多多包涵！

## 现有系统如何改造落地

## 主流方案场景

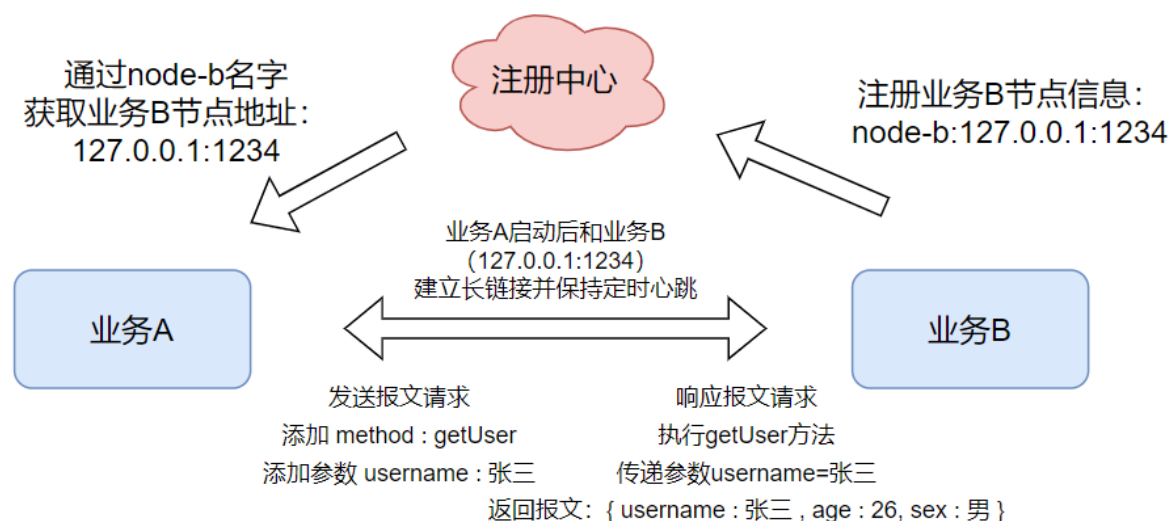
目前我行（业界）主流的交互方案有两种：

http远程调用体系（短连接）



典型体系: Feign

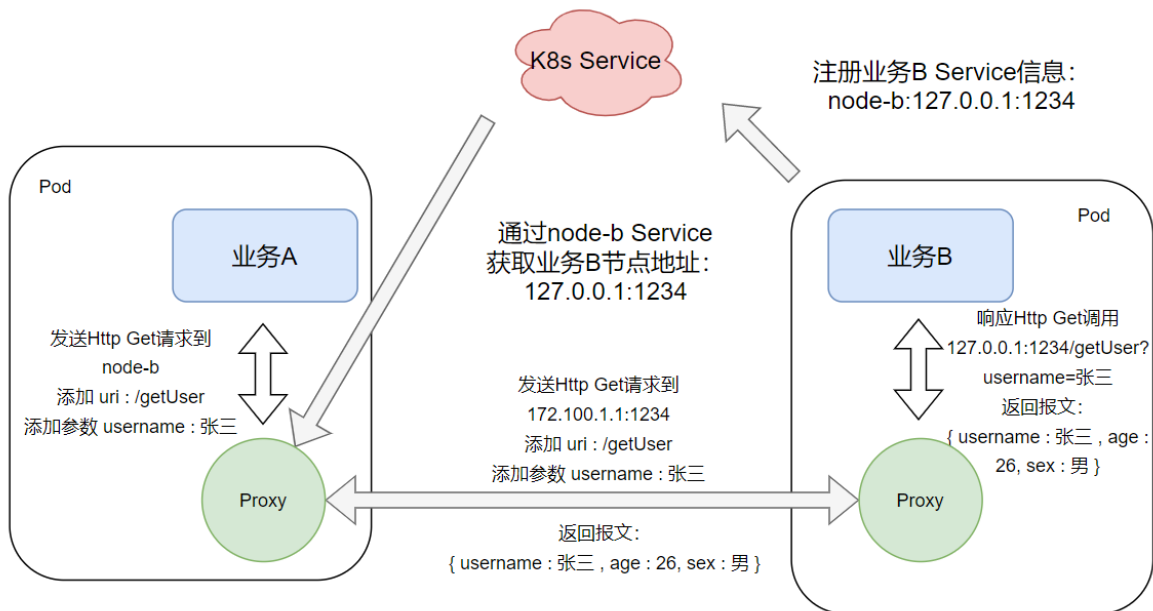
rpc远程过程调用体系（长连接）



典型体系: Dubbo

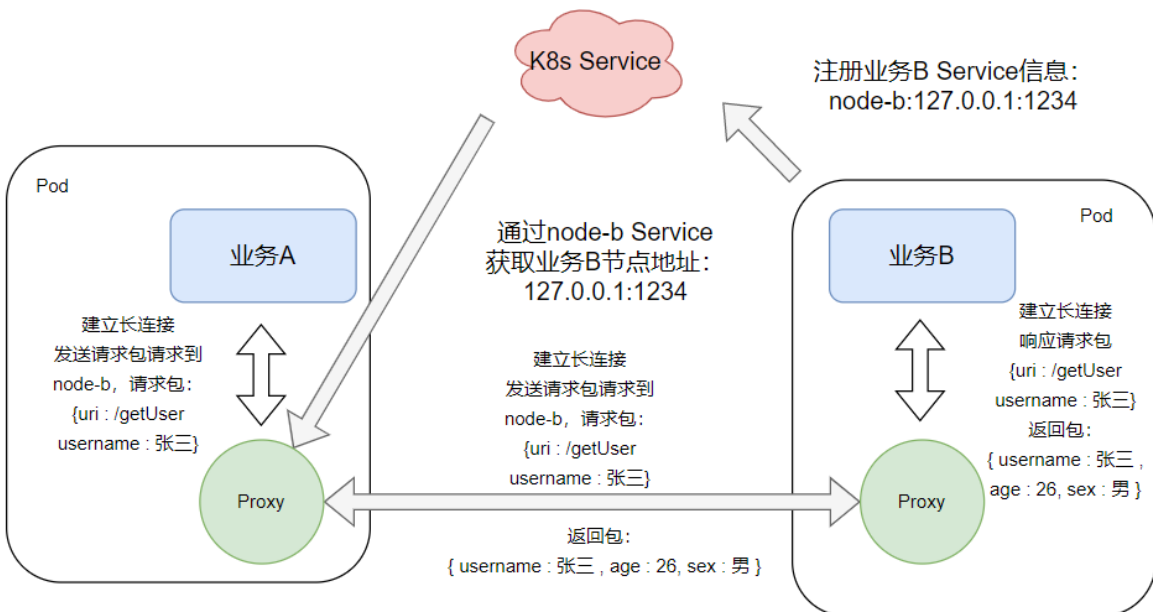
在网格内交互示意图

http远程调用体系（短连接）



rpc远程过程调用体系（长连接）

istio对于长连接的负载是连接层面的，并非请求层面的



## 服务改造

### 代码改造步骤示例:

#### http短连接调用

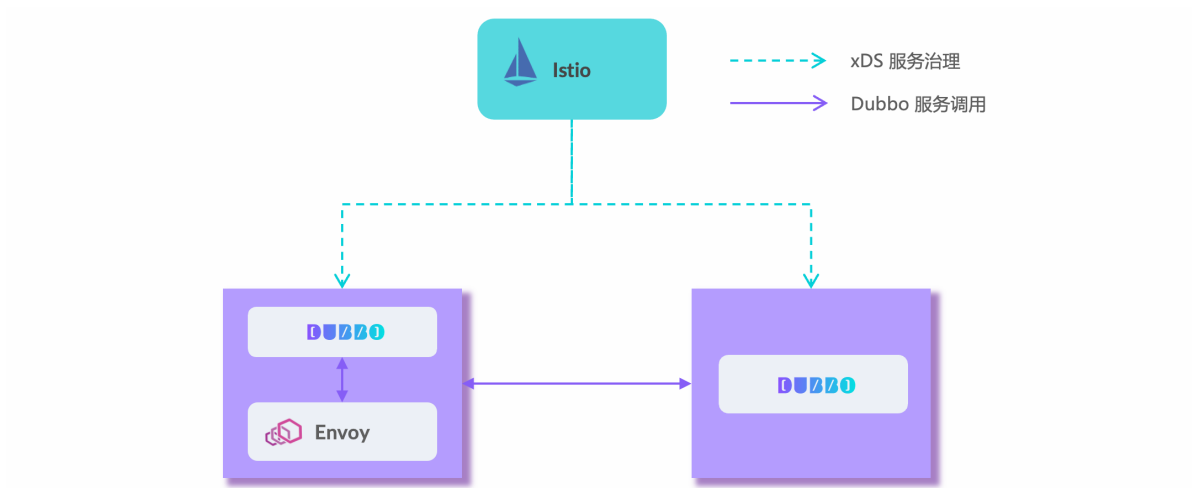
```
// 云环境调用方式
@FeignClient(name = "userService", url = "userService:9093") // 显式的指定域名和端口, 通过k8s service机制做负载均衡策略
@FeignClient(name = "userService", url = "${user-service.address}") // 通过配置获取域名和端口, 通过k8s service机制做负载均衡策略

// 微服务环境调用方式
@FeignClient(name = "userService") // 通过配置中心获取所有的节点, 通过ribbon做负载均衡策略
public interface UserService {

    @RequestMapping(value = "/getUser")
    String getUser(@RequestBody User user);
}
```

## rpc长连接调用

Dubbo的生态比较特殊，根据官网介绍：



或者自定义改造：

```
/**
 * 配合动态调整生产者地址改造
 * @param address
 */
public Consumer(@Value("${dubbo.service.address}") String address) {
    String[] split = address.split( regex: ";");
    for (String s : split) {
        String name = s.split( regex: "-")[0];
        String addr = s.split( regex: "-")[1];
        System.setProperty(name, addr);
    }
}

@Autowired // 正常使用
@Reference(url = "localhost:3009") // 显示的指定生产者地址，dubbo这个注解的url参数不支持动态读取spring配置文件
private UserService userService;

@RequestMapping("/dubbo-health")
public String health() {
    return "health";
}

@RequestMapping("/dubbo-send")
public String send() {
    return userService.getUser();
}

dubbo:
  service:
    address: com.demo.UserService-dubbo://localhost:3000;
spring:
  dubbo:
    appname: dubbo1
    registry: N/A
```



## 容器操作步骤示例：

以docker为底层运行容器为例，需要编写dockerfile，将服务制作成镜像，然后启动镜像发布服务。示例：

```
#添加jdk8镜像依赖
FROM openjdk:8
#指定启动文件
ADD jar/mvc.jar mvc.jar
#设置端口
EXPOSE 2000
```

将服务jar打包成镜像后

```
docker build -f MvcDockerfile -t demomvc:v1 .
```

通过docker run启动便实现了全服务容器化。

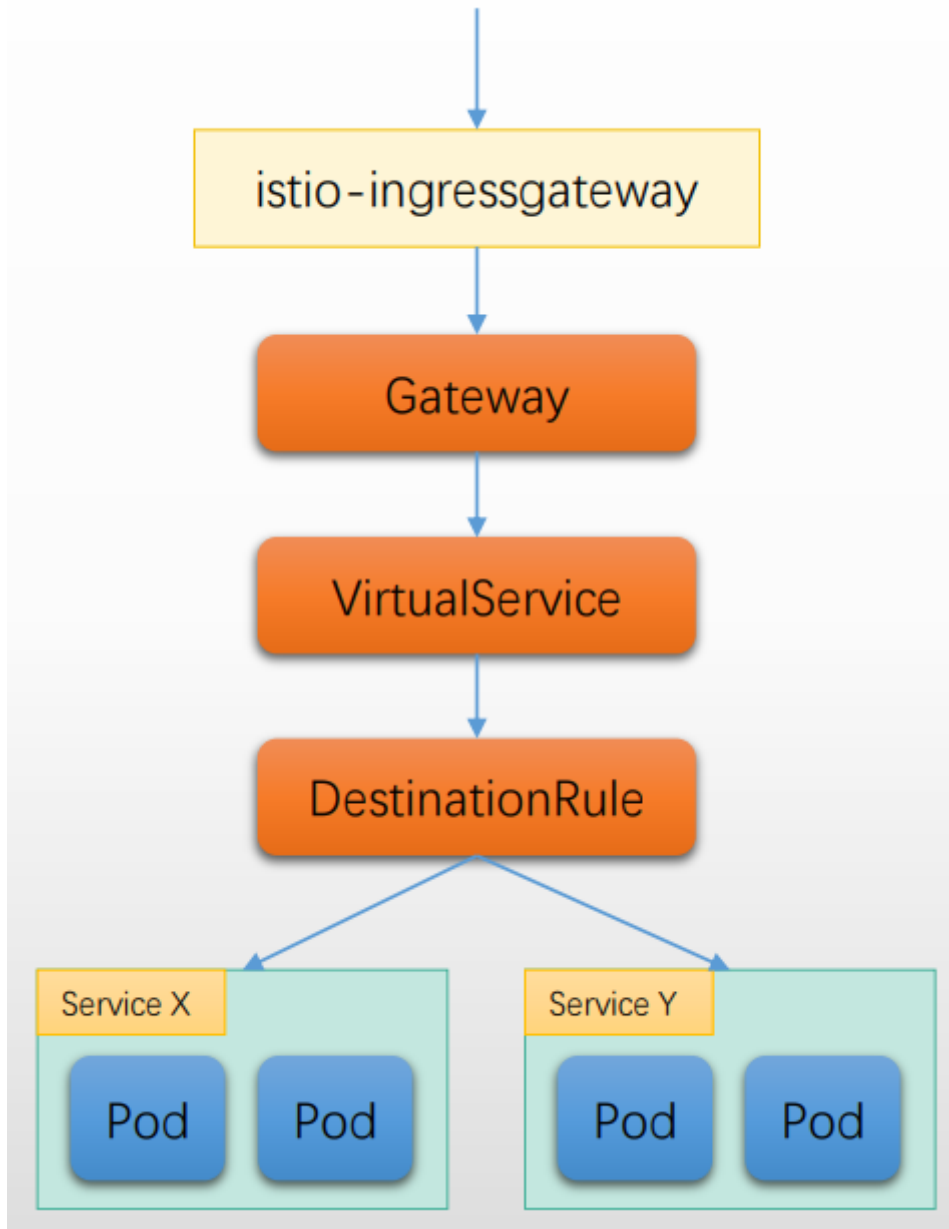
## 编排容器

k8s的配置文件示例如下，之后通过kubectl apply -f执行即可：

```
#service配置
apiVersion: v1
kind: Service
metadata:
  name: mvc-k8s-svc
  namespace: istio0322
spec:
  ports:
    #以下为端口映射设置
    #端口配置组名
    - name: http-mvc-k8s
      #service开放的端口
      port: 10001
      #容器内端口
      targetPort: 2000
    #对集群外暴露的端口
  selector:
    app: demo-mvc
  #service类型 默认clusterIp
---
#以下为pod配置
apiVersion: apps/v1
#pod类型
kind: Deployment
#元数据信息
metadata:
  #Deployment名字
  name: mvc-deployment
  #所在命名空间
  namespace: istio0322
spec:
  #启动服务分片个数
  replicas: 1
  selector:
```

```
#标签
matchLabels:
  app: demo-mvc
#模板
template:
  metadata:
    #匹配service的标签
    labels:
      app: demo-mvc
      version: v1
    #以下为镜像库设置
  spec:
    containers:
      - image: demomvc:v1
        name: demomvc
        imagePullPolicy: IfNotPresent
        #设置pod资源和容器端口号
        resources:
          limits:
            cpu: "0.1"
            memory: 128Mi
        ports:
          - containerPort: 2000
        command: ["java", "-jar", "mvc.jar"]
```

## 服务网格化



添加流控、灰度发布、故障注入等能力：

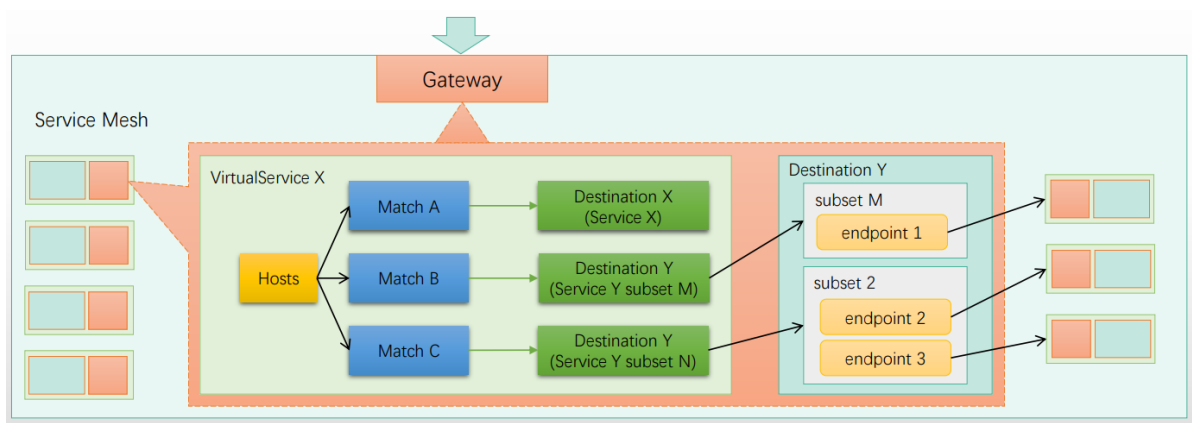
```
apiVersion: networking.istio.io/v1alpha3
#设置istio网关类型
kind: Gateway
metadata:
  name: mvc-istio-gateway
  namespace: istio0322
spec:
  selector:
    #关联istio的ingress，绑定上面的外部入口
    istio: ingressgateway
  servers:
    - port:
        #定义流量
        number: 10001
        name: http
        protocol: HTTP
        #不限制流量来源
        hosts:
          - "*"
---
```

```




apiVersion: networking.istio.io/v1alpha3
#istio的虚拟service类型定义
kind: VirtualService
metadata:
  name: mvc-istio-vs
  namespace: istio0322
spec:
  hosts:
  - "*"
  #绑定上述网关配置
  gateways:
  - istio0322/mvc-istio-gateway
  #匹配http协议的规则
  http:
  - match:
    #参数匹配
    - queryParams:
        color:
          exact: white
    - queryParams:
        color:
          exact: red
    - queryParams:
        color:
          exact: yellow
    - queryParams:
        color:
          exact: blue
    route:
    #转发的目的地
    - destination:
        host: mvc-k8s-svc
  - match:
    #uri匹配
    - uri:
        prefix: /headerTest
    route:
    - destination:
        host: mvc-k8s-svc
    #为请求添加请求头
    headers:
      request:
        set:
          test-header: "0322"

```

## 网格流量转发实例



设置三个版本的service服务，网格内不需要gateway

	ServiceDeploymentV1.yaml	2023/3/24 17:01	YAML 文件	1 KB
	ServiceDeploymentV2.yaml	2023/3/24 17:01	YAML 文件	1 KB
	ServiceDeploymentV3.yaml	2023/3/24 17:01	YAML 文件	1 KB

```
apiVersion: networking.istio.io/v1alpha3
```

#istio的虚拟service类型定义

```
kind: VirtualService
```

```
metadata:
```

```
  name: service-istio-vs
```

```
  namespace: istio0322
```

```
spec:
```

```
  hosts:
```

```
  - service-k8s-svc
```

#匹配http协议的规则

```
  http:
```

```
  - match:
```

#请求头匹配

```
    - headers:
```

```
      color:
```

```
        exact: red
```

```
    route:
```

#指定目的地，同时指向sub-v1版本

```
    - destination:
```

```
      host: service-k8s-svc
```

```
      subset: sub-v1
```

```
  - match:
```

```
    - headers:
```

```
      color:
```

```
        exact: yellow
```

```
    route:
```

#指定目的地，同时指向sub-v2版本

```
    - destination:
```

```
      host: service-k8s-svc
```

```
      subset: sub-v2
```

```
  - match:
```

```
    - headers:
```

```
      color:
```

```
        exact: blue
```

```
    route:
```

#指定目的地，同时指向sub-v3版本

```
    - destination:
```

```
      host: service-k8s-svc
```

```
      subset: sub-v3
```

```

#默认路由规则
- route:
  #指定目的地，同时指向sub-v1版本，且指定权重为33%
  - destination:
    host: service-k8s-svc
    subset: sub-v1
    weight: 33
  #指定目的地，同时指向sub-v2版本，且指定权重为33%
  - destination:
    host: service-k8s-svc
    subset: sub-v2
    weight: 33
  #指定目的地，同时指向sub-v3版本，且指定权重为34%
  - destination:
    host: service-k8s-svc
    subset: sub-v3
    weight: 34
---
apiVersion: networking.istio.io/v1alpha3
#指定目的规则类型
kind: DestinationRule
metadata:
  name: service-istio-rule
  namespace: istio0322
spec:
  host: service-k8s-svc
  subsets:
    #设置sub-v1对应的pod labels为v1
    - name: sub-v1
      labels:
        version: v1
    #设置sub-v2对应的pod labels为v2
    - name: sub-v2
      labels:
        version: v2
    #设置sub-v3对应的pod labels为v3
    - name: sub-v3
      labels:
        version: v3

```

## 结束语

综上所述，服务网格技术能够有效解决传统微服务架构存在维护成本过高、治理难度偏大、技术栈难以转型等痛点问题。与此同时，在诸多企业、机构的共同努力下，服务网格技术得以持续完善和优化，在银行业也有了较为充分的论证和应用。为稳步推进我行的信息系统转型任务，深度提升科技赋能实效，在当前信息化工作中，应当对服务网格技术的发展和应用提高重视，结合行业经验和我行现状，通过研讨、试验等方式，充分论证服务网格技术的可行性，深入挖掘应用潜力，为行内信息系统的技术转型探索出新的方向。

## 附录

项目实战代理仓库地址：<https://github.com/970263611/istio-demo/tree/main/%E5%AE%9E%E6%88%98>

dubbo项目仓库地址：<https://github.com/970263611/istio-demo/tree/main/dubbo>

操作步骤：

0.拥有k8s和istio环境

1.将两个服务打包制作成镜像为demomvc:v1, demoservice:v1

2.创建命名空间:

```
kubectl create namespace istio0322
```

3.自动注入边车指令

```
kubectl label namespace istio0322 istio-injection=enabled
```

4.调整istio监听网格外流量

```
kubectl get svc mvc-k8s-svc -n istio0322 -o yaml
```

添加需要的端口

```
- name: http-share  
  nodePort: 30001  
  port: 10001  
  protocol: TCP  
  targetPort: 2000
```

5.执行启动

```
kubectl apply -R -f *.yaml
```

6.请求测试(反复请求看效果)

```
curl localhost:30001?color=white
```

```
curl localhost:30001?color=red
```

```
curl localhost:30001?color=yellow
```

```
curl localhost:30001?color=blue
```