为什么要使用istio

由于Docker的出现，容器技术得到更广泛的认可和应用，容器的特性是什么：

轻量级、启动速度快、性能损失小、扩容缩容快、开发与生产环境统一等特性。

容器技术的飞速发展也大大加速了微服务的应用，现在大部分公司也在使用容器化，或者是容器化改造的路上来对微服务架构来重构，当微服务的服务数量越来越大时，微服务间的服务通信也越来越重要，我们所看到的一个应用，有可能背后需要协调成百上千个微服务来处理用户的请求。随着服务数和服务实例数的不断增长，服务可能上线下线，服务实例也可能出现上线下线和宕机的情况，服务之间的通信变得异常复杂，每个服务都需要自己处理复杂的服务间通信。

微服务架构流行以后，服务的数量在不断增长。在不使用服务网格的情况下，每个服务都需要自己管理复杂的服务间网络通信，开发人员不得不使用各种库和框架来更好地处理服务间的复杂网络通信问题，这导致代码中包含很多与业务逻辑完全不相关的代码，稍有不慎就有可能给业务带来额外的复杂度和bug。

当服务规模逐渐变大，复杂度增加，服务间的通信也变得越来越难理解和管理，这就要求服务治理包含很多功能，例如：服务发现、负载均衡、故障转移、服务度量指标收集和监控等。在使用服务网格时，我们甚至完全不需要改动现有的服务代码，服务开发完全可以使用不同的语言和技术栈，框架之类的也不会成为卡点。

❑目前微服务架构中的痛点：面对复杂的服务间通信问题，一般的解决方案是为服务开发统一的服务框架，所有服务依赖于服务框架开发，所有服务间通信、服务注册、服务路由等功能都由底层服务框架来实现，这样做固然可以在某种程度上解决服务间通信的问题，但是由于底层服务框架的限制，业务人员可能无法基于实际情况选择合适的技术栈；由于所有服务都依赖于底层的服务框架代码库，当框架代码需要更新时，业务开发人员可能并不能立即更新服务框架，导致服务框架整体升级困难。

服务治理的问题由来已久，微服务化盛行之后尤为突出。主流的方案有：基于Spring Cloud或者Dubbo等框架。但是这些方案的问题是：1.对代码有侵入，也就意味着，如果想换框架，得改很多东西。2.语言特异性(Java)，如果我们用的是 Go/Python，或者我们的微服务不全是 Java，就搞不定了。

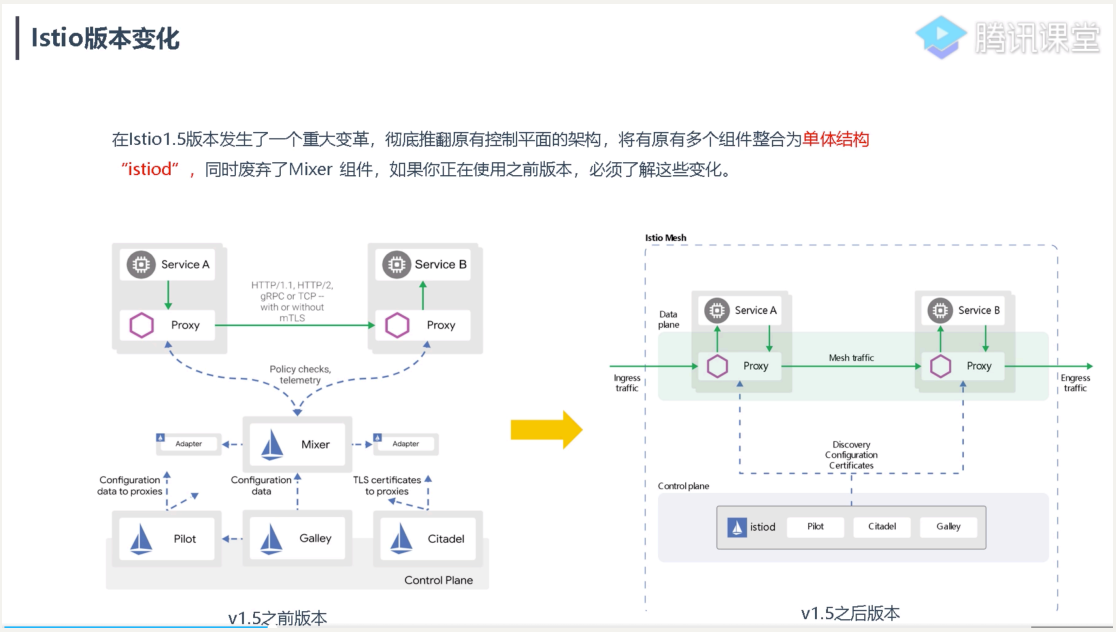
基于以上服务间通信出现的问题，我们能不能把服务间的复杂通信分层并下沉到基础设施层，让应用无感知呢？答案是肯定的。于是服务网格开始渐渐浮出水面，越来越多的人看到了服务网格的价值，尝试把服务网格应用于微服务实践中

在云原生昌盛的今天，容器和Kubernetes增强了应用的伸缩能力，开发者可以基于此快速地创造出依赖关系复杂的应用；而使用服务网格，开发者不用关心应用的服务管理，只需要专注于业务逻辑的开发，这将赋予开发者更多的创造性。

服务应用代码中将不再需要那些用于处理服务间复杂网络通信的底层代码，我们可以更好地控制请求的流量，对服务进行更好的路由，使服务间的通信更加安全可靠，让服务更具有弹性，还能让我们更好地观测服务，并可以提前给服务注入故障，以测试应用的健壮性和可用性。而拥有这些功能只需要我们的服务做出微小的改变，甚至不需要改变。以上提到的这些功能，在中小规模的公司中，使用服务网格技术，只需要少量的人力投入就能拥有以前大公司才具备的高级服务治理能力。

#下图为其他学院视频配图，不能上开课吧

Kubernetes与Istio结合



服务网格

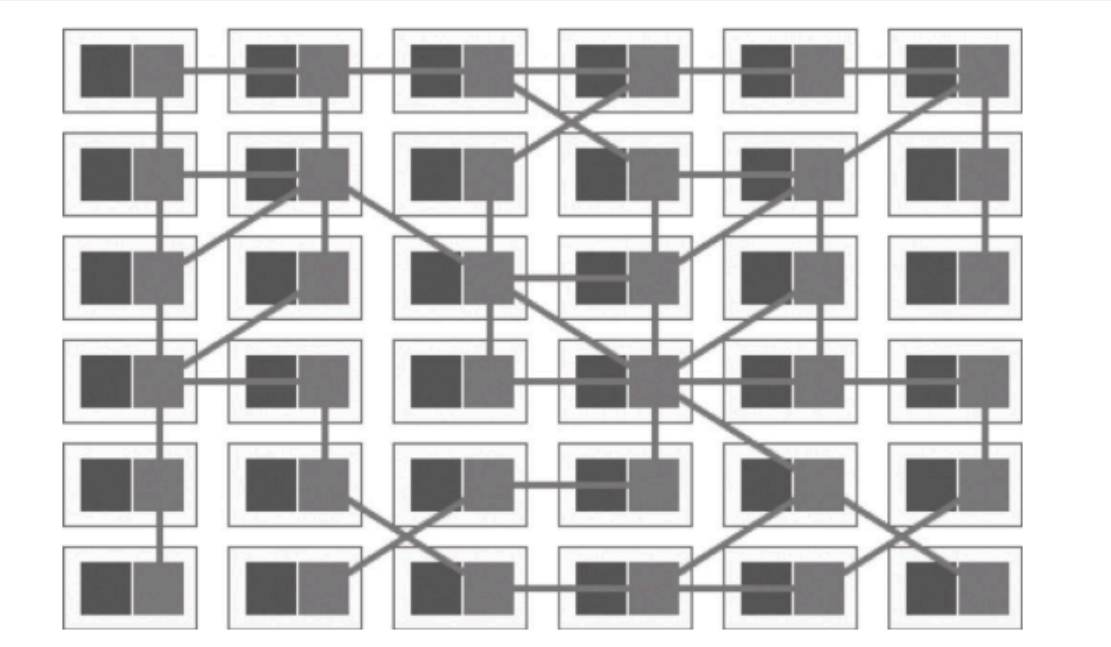
服务网格是一个专注于处理服务间通信的基础设施层，它负责在现代云原生应用组成的复杂服务拓扑中可靠地传递请求。在实践中，服务网格通常是一组随着应用代码部署的轻量级网络代理，而应用不用感知它的存在。服务网格的特点如下：

❑轻量级的网络代理。

❑应用无感知。

❑应用之间的流量由服务网格接管。（服务和服务之前要进行通信，必须要走sidcar来进行了，不能应用和应用之前进行通行了，被sidcar接管了）

❑把服务间调用可能出现的超时、重试、监控、追踪等工作下沉到服务网格层处理。



Istio出自希腊语，表示“航行”的意思，官方图标为一个白色的小帆船。使用Istio可以让服务间的通信更简单、更安全，控制服务更容易，观测服务更方便。

Istio是由Google、IBM、Lyft公司主导开发的影响力最大的开源服务网格实现，使用Go语言编码

Istio的主要功能特性Istio可以让你轻松部署一个服务网格，而不需要在服务代码中做任何改变。只需要在你的环境中部署一个特殊的代理用来拦截所有微服务间的网络通信，就可以通过控制平面配置和管理Istio。Istio的功能特性如下：

❑HTTP、g RPC、Web Socket、TCP流量的自动负载均衡。

❑细粒度的流量路由控制，包含丰富的路由控制、重试、故障转移和故障注入。

❑可插拔的访问控制策略层，支持ACL、请求速率限制和请求配额。

❑集群内度量指标，日志和调用链的自动收集，管理集群的入口、出口流量。

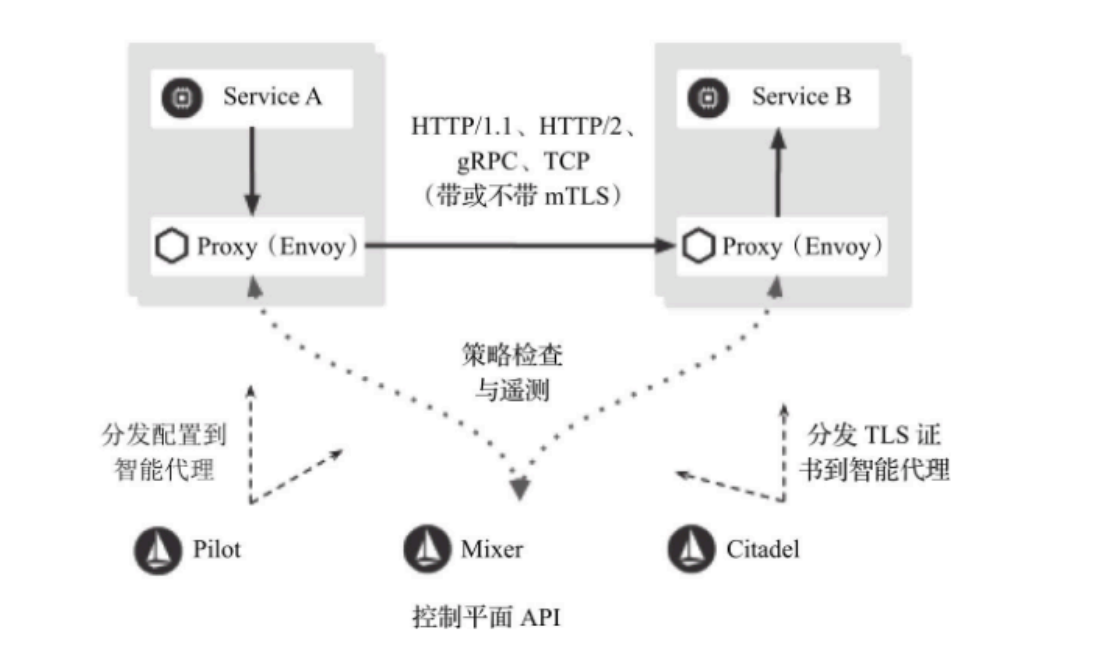
❑使用基于身份的认证和授权方式来管理服务间通信的安全。由于Istio提供了足够多的可扩展性，这也使得Istio能满足多样化的需求。基于Istio你完全可以搭建出一套适合自己公司基础设施层的服务网格。

Istio的架构设计

Istio的架构设计在逻辑上分为数据平面和控制平面：

❑数据平面由一系列称为“边车”（sidecar）的智能代理组成，这些代理通过Mixer来控制所有微服务间的网络通信，Mixer是一个通用的策略和遥测中心：Istio的数据平面主要负责流量转发、策略实施与遥测数据上报。

❑控制平面负责管理和配置代理来路由流量，另外，控制平面通过配置Mixer来实施策略与遥测数据收集：Istio的控制平面主要负责接收用户配置生成路由规则、分发路由规则到代理、分发策略与遥测数据收集。



数据平面

Istio在数据平面中使用一个Envoy代理的扩展版本。Envoy是使用C++语言开发的高性能代理，它能拦截服务网络中所有服务的入口和出口流量。Envoy 被用于 Sidecar 和对应的应用服务部署在同一个 Kubernetes 的 Pod 中,Istio利用了众多Envoy内置的功能特性，例如：

❑动态服务发现

❑负载均衡

❑TLS终止

❑HTTP/2和g RPC代理

❑熔断器

❑健康检查

❑基于百分比流量分隔的灰度发布

❑故障注入

❑丰富的度量指标: Envoy作为一个边车与对应的服务部署在同一个KubernetesPod中。这种部署方式使得Istio能提取丰富的流量行为信号作为属性。Istio又可以反过来使用这些数据在Mixer中进行策略决策，并发送这些数据到监控系统中，提供整个网络中的行为信息。

Sidecar部署方式，可以把Istio的功能添加到一个已经存在的部署中，并且不需要重新构建或者重新编写代码。

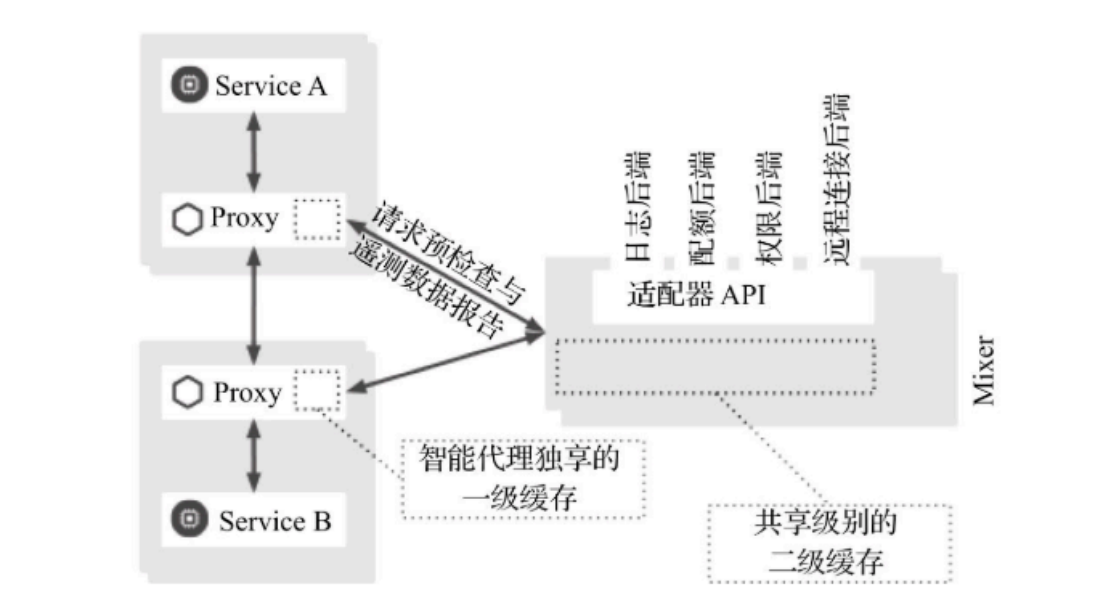
Sidecar部署在pod中的时候会增加一个proxy容器，这个proxy容器负责所有的微服务网络通信，实现转发和策略；

控制平面

控制平面中主要包括Mixer、Pilot、Citadel部件。

Mixer（1.5版本前）

1. Mixer负责在服务网络中实施访问控制和策略，并负责从Envoy代理和其他服务上收集遥测数据。代理提取请求级别的属性并发送到Mixer用于评估，评估请求是否能放行。
2. Mixer有一个灵活的插件模型。这个模型使得Istio可以与多种主机环境和后端基础设施对接。因此，Istio从这些细节中抽象了Envoy代理和Istio管理的服务



在每一个请求过程中，Envoy代理会在请求之前调用Mixer组件进行前置条件检查，在请求结束之后上报遥测数据给Mixer组件。为了提高性能，每个Envoy代理都会提前缓存大量前置条件检查规则，当需要进行前置条件检查时，直接在缓存中检查规则。如果本地缓存中没有需要的规则，再去调用Mixer组件获取规则。Mixer组件也有自己的缓存，以加速前置条件检查。需要上报的遥测数据也会被Envoy代理暂时缓存起来，等待时机上报Mixer组件，从而减少上报数据的调用次数。

- **Mixer 的设计还具有以下特点**：

1. 无状态：Mixer 本身是无状态的，它没有持久化存储的管理功能。
2. 高可用：Mixer 被设计成高度可用的组件，任何单独的 Mixer 实例实现 > 99.999% 的正常运行时间
3. 缓存和缓冲：Mixer 能够积累大量短暂的瞬间状态

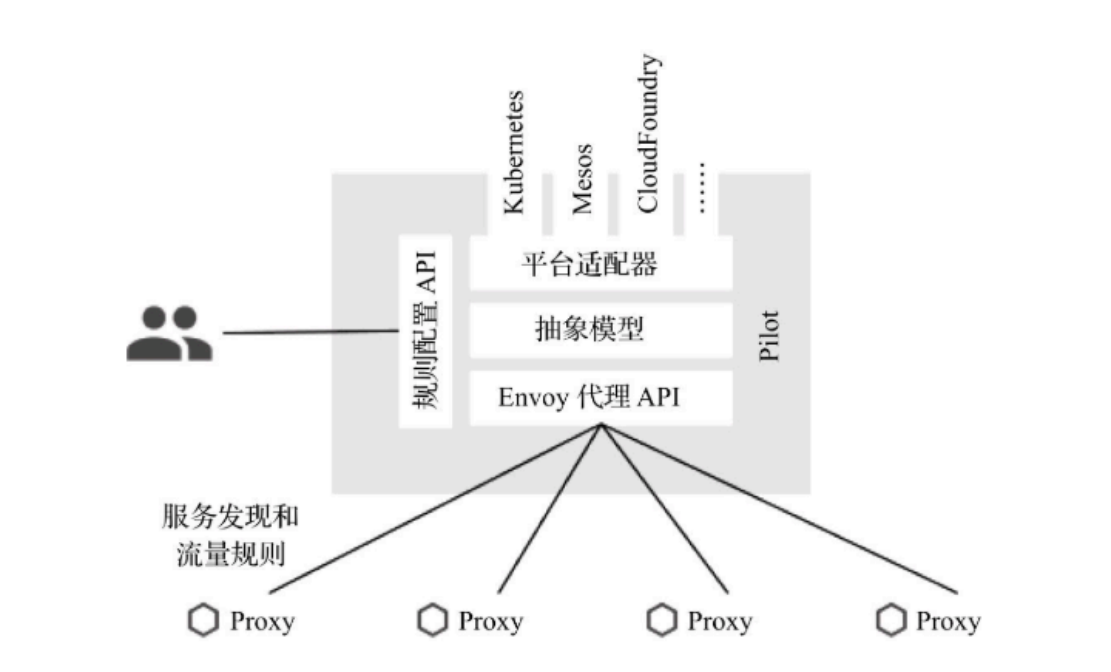
Pilot （主要是发现哪些服务注册到了istio集群中，也就是哪些pod注入了Sidecar）

为Envoy代理提供服务发现功能，并提供智能路由功能（例如：A/B测试、金丝雀发布等）和弹性功能（例如：超时、重试、熔断器等）。

Pilot将高级别的控制流量行为的路由策略转换为Envoy格式的配置形式，并在运行时分发给Envoy代理。Pilot抽象了平台相关的服务发现机制，并转换成Envoy数据平面支持的标准格式,Pilot 将这些“高级”的流量行为转换为详尽的 Sidecar (即 Envoy) 配置项，并在运行时将它们配置到 Sidecar 中。

Pilot 将服务发现机制转换为供数据面使用的 API ，即任何 Sidecar 都可以使用的标准格式。

这种松耦合设计使得Istio能运行在多平台环境，并保持一致的流量管理接口。



Pilot抽象不同平台的服务发现机制，只需要为不同的平台实现统一的抽象模型接口，即可实现对接不同平台的服务发现机制。用户通过规则配置API来提交配置规则，Pilot把用户配置的规则和服务发现收集到的服务转换成Envoy代理需要的配置格式，推送给每个Envoy代理。

Citadel

Citadel内置有身份和凭证管理，提供了强大的服务间和终端用户的认证。Citadel可以把不加密的通信升级为加密的通信。运维人员可以使用Citadel实施基于服务身份的策略而不用在网络层控制。现在，Istio还支持基于角色的访问控制，用于控制谁能够访问服务。

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。

Istio的功能特性

1. 强大的流量管理Istio提供了简单的规则配置和流量路由功能，用于控制服务间的流量流动和API调用。Istio简化了服务级别特性的配置，如熔断器、超时、重试，并且能更简单地配置一些复杂的功能，如A/B测试、金丝雀发布、基于百分比进行流量分隔的灰度发布等。Istio提供了开箱即用的故障恢复功能，你可以在问题出现之前找到它，使服务调用更可靠，网络更加健壮。
2. 安全可靠Istio提供了强大的安全功能，使得开发者不必再过分关注应用级别的安全问题。Istio提供了底层的安全通信，并且管理认证、授权和服务间通信加密。Istio对服务间通信默认加密以保证安全，在不同的协议和运行环境中也可使用统一的策略，只需要很少的修改或者完全不需要应用做出修改。虽然Istio是平台无关的，但是当你在Kubernetes上使用Istio时，结合Kubernetes的网络策略，就可以在网络层实现Pod级别安全隔离，使用Istio提供的安全功能实现服务级别的安全通信。
3. 便捷的观测能力Istio提供了强大的调用链跟踪、指标收集、日志收集和监控功能，使用者可以更深入地了解服务网格的部署情况和运行状态，可以真正了解服务性能如何影响上游和下游的功能，可以自定义仪表板对所有服务性能进行可视化管理，并了解该性能如何影响其他的应用程序。所有这些功能都可以帮助你更高效地观测服务，增强服务的SLO（服务等级目标），最重要的是，可以帮助你更高效地找到服务的问题并修复。
4. Istio的设计目标是平台无关，可以运行在多种环境，包括跨云环境、裸机环境、Kubernetes、Mesos等。

   虽然官方支持多种部署方式，但是目前有许多功能是基于Kubernetes的原生功能来实现的，Istio在Kubernetes上实现的功能也是最多的，比如自动注入代理。综合来看，在Kubernetes上尝试使用Istio是目前最好的选择。

5. 易于集成和定制Istio具备扩展和定制功能，并可以与当前已经存在的解决方案集成，包括访问控制（ACL）、日志、监控、配额、认证、审计等功能

最大的劣势：

复杂度高，技术门槛和学习成本高