Service Mesh

Service Mesh（服务网格），是服务间通信的基础设施层，其负责云原生应用中复杂服务拓扑中请求的可靠传递。Service Mesh与应用程序部署在一起，作为一组轻量级网络代理，对应用程序透明。在云原生模型中，单个应用程序可能由数百个服务组成，每个服务有数千个实例，而这些实例中每一个都可能处于不断变化的状态（像Kuberetes编排器一样的动态调度）。服务间通信不仅异常复杂，而且也是运行时行为的基础，因此服务间通信对于保证端到端的性能和可靠性是非常重要的。

Service Mesh是位于TCP/IP之上的网络模型抽象层，其底层的L3/L4网络存在并且能够从点到点传送字节，而且网络和环境的其他方面一样不可靠，因此Service Mesh必须具备处理网络故障的能力。Service Mesh有点类似TCP/IP，TCP对网络终点间传输字节的机制进行了抽象，而Service Mesh则是对服务节点间请求的路由机制进行了抽象，不关心实际的有效载荷或者编码方式，而是处理应用程序之间消息传送的过程中出现的任何故障。但是与TCP不同的是，Service Mesh为应用运行时提供统一的、应用层面的可见性和可控性，将服务的交互从隐形基础设施变成生态系统的first-class角色，对服务交互进行监控、管理和控制。

Service Mesh并非新实现的功能，Web应用程序一直需要自己管理复杂的服务间通信，其一般使用三层模型：应用逻辑层、Web服务逻辑层和存储逻辑层，层与层之间交互随着复杂，但范围有限，毕竟一个请求最多只需要两个跳转，不存在网格，但仍然存在跳转通信逻辑。随着规模的增长，这种架构就力不从心，像Google、Netflix、Twitter公司面临着大规模流量的需求，他们实现了云原生应用的前身：应用层被拆分为多个服务（微服务），层级成了一种拓扑结构，这样的系统需要一个通用的通信层，以一个富客户端包的形式存在，如Twitter的Finagle、Netflix的Hystrix和Google的Stubby。这三个组件允许具有自然机制的应用程序在负载下进行缩放，并处理云环境的永久存在的局部故障。

但是，现代云原生应用程序中有数百个服务或数千个服务，以及一个不时重新调度实例的编排层，现代云原生应用程序中云原生模式将许多小型服务的微服务方法与另外两个因素相结合：容器（例如，提供资源隔离和依赖管理的Docker），以及将底层硬件抽象为同质池的编排层（例如Kubernetes）。单个请求通过服务拓扑所遵循的路径非常复杂，并且由于容器使得每个服务都可以很容易写入不同语言。这种复杂性和重要性结合激发了对应用程序代码分离的服务到服务通信的专用层的需求，并且能够捕获底层环境的高度动态性质，这层称为Service Mesh，是云原生技术栈中非常关键的组件。下面是Service Mesh列表：

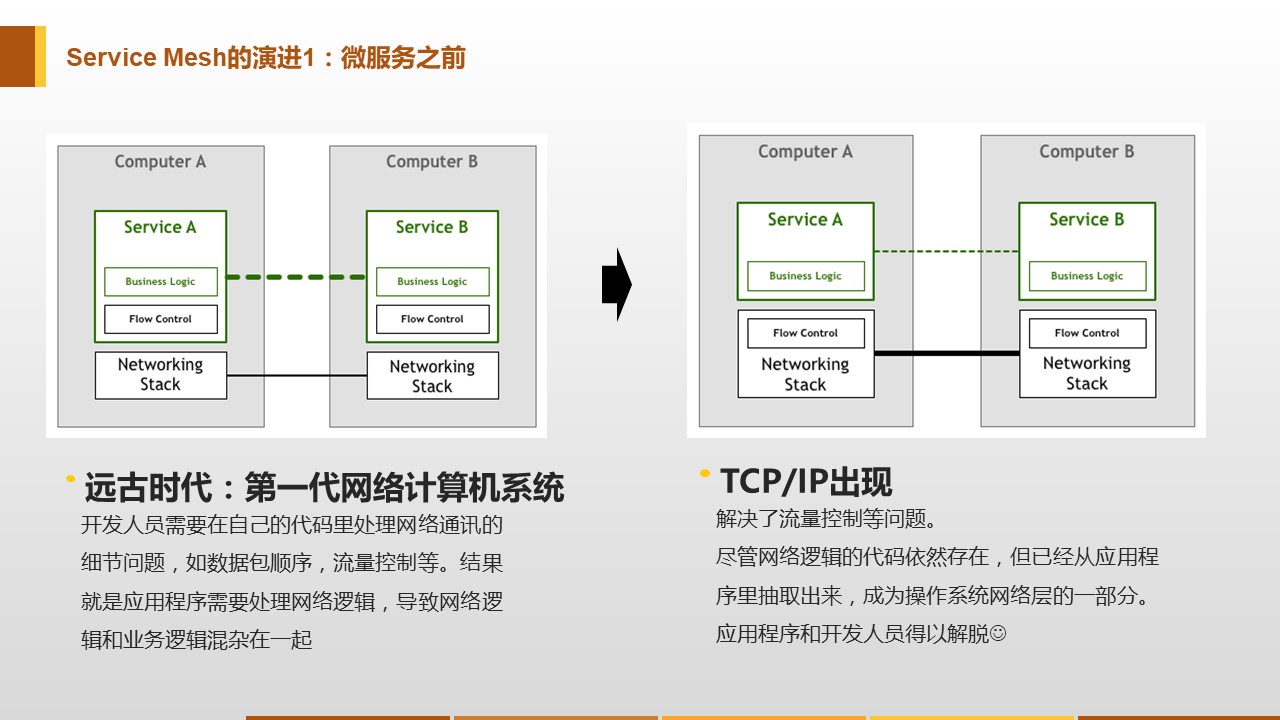
* 2016年9月13日，Lyft公司的Envoy在Github开源
* 2016年下半年，Bouyant公司的”Service Mesh for Kubernetes” Linkerd陆续发布0.8和0.9，开始支持HTTP/2和gRPC，年底加入CNCF
* 2017年，Google和IBM，联合Lyft启动Istio项目
* 2017年12月，Bouyant公司推出Conduit，整体架构与Istio一致。
* 2018年，MicroSoft开源Service Fabric

http://www.infoq.com/cn/articles/2017-service-mesh

https://my.oschina.net/shurenyun/blog/1553747&quot;

# Service Mesh的演进和功能

在第一代网络计算机系统，最早的时候开发人员需要在自己的代码中处理网络通讯的细节问题，比如数据包顺序、流量控制等，导致网络逻辑和业务逻辑混杂在一起，出现TCP/IP技术后解决了流量控制问题：

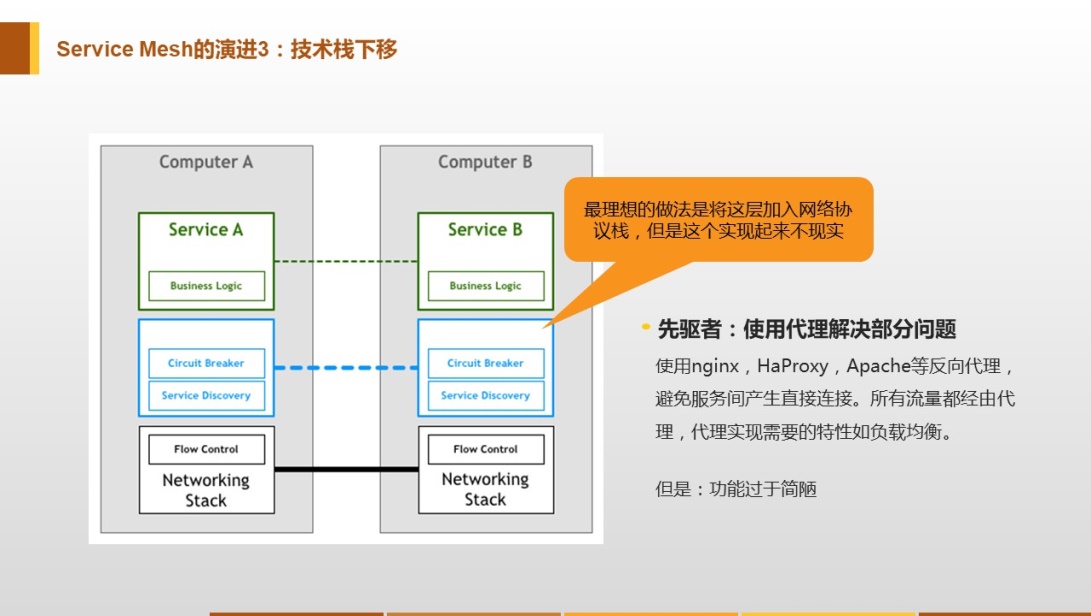


微服务出现后，比如常见的服务注册、服务发现，在得到服务实例之后做负载均衡，为了保护服务器的熔断/重试等，这些功能需要写入到应用程序中，为了简化开发，开始使用类库，比如典型的Netflix OSS套件：

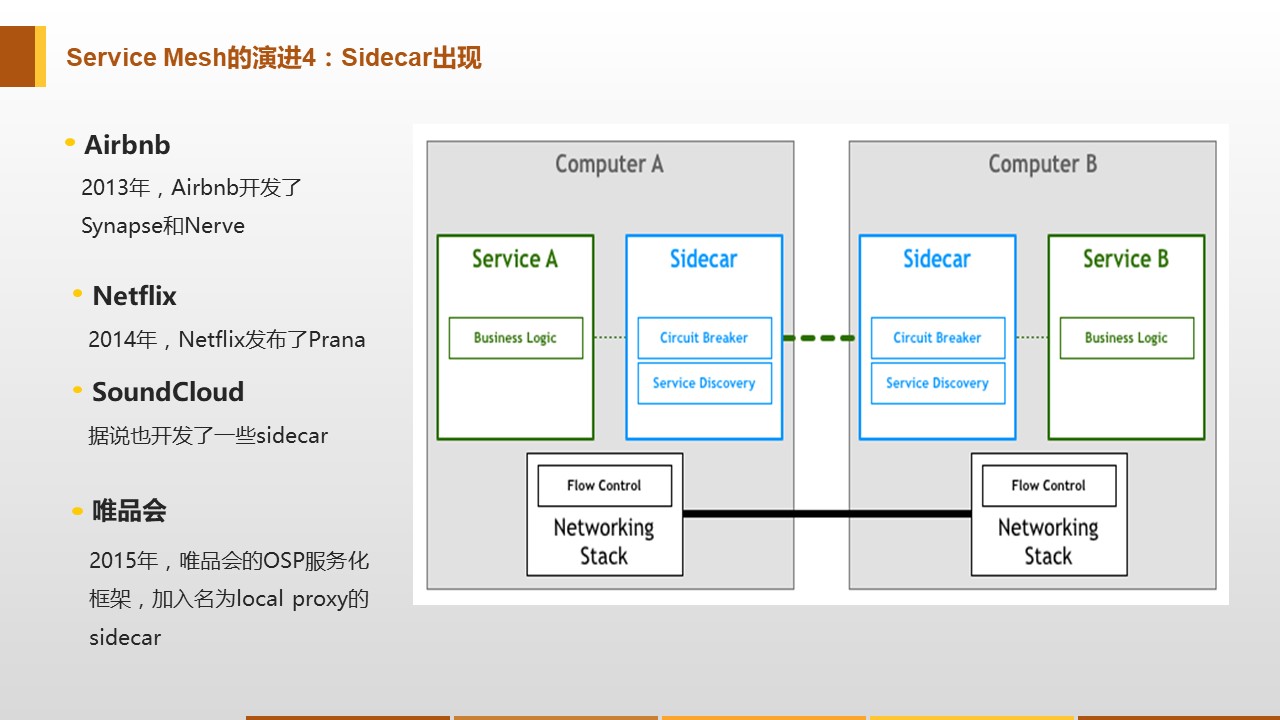


但是实现过程难度较大，业务开发需要对业务详细了解，并且根据需要对微服务拆分。

既然将网络访问的技术栈下移到TCP，也可以将微服务的技术栈下移，增加一个微服务层来完成服务通信，出现了一种代理方案，常见的nginx、haproxy和apache等代理，这些代理和微服务关系不大，在服务器端和客户端之间插入代理，避免两者直接通讯，，如下图：



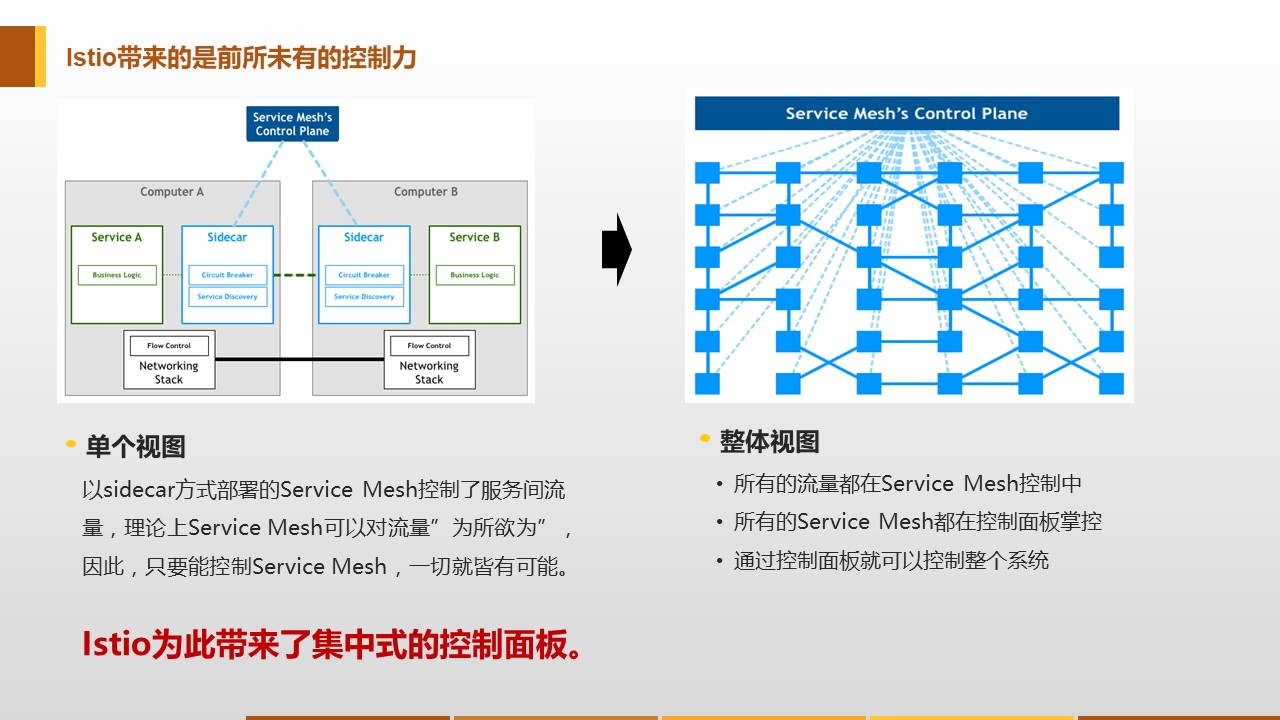
但是这些代理方案功能太少，因此针对微服务引入了SideBar，基本上原来微服务框架在客户端实现的功能都会对应实现，如下图所示：



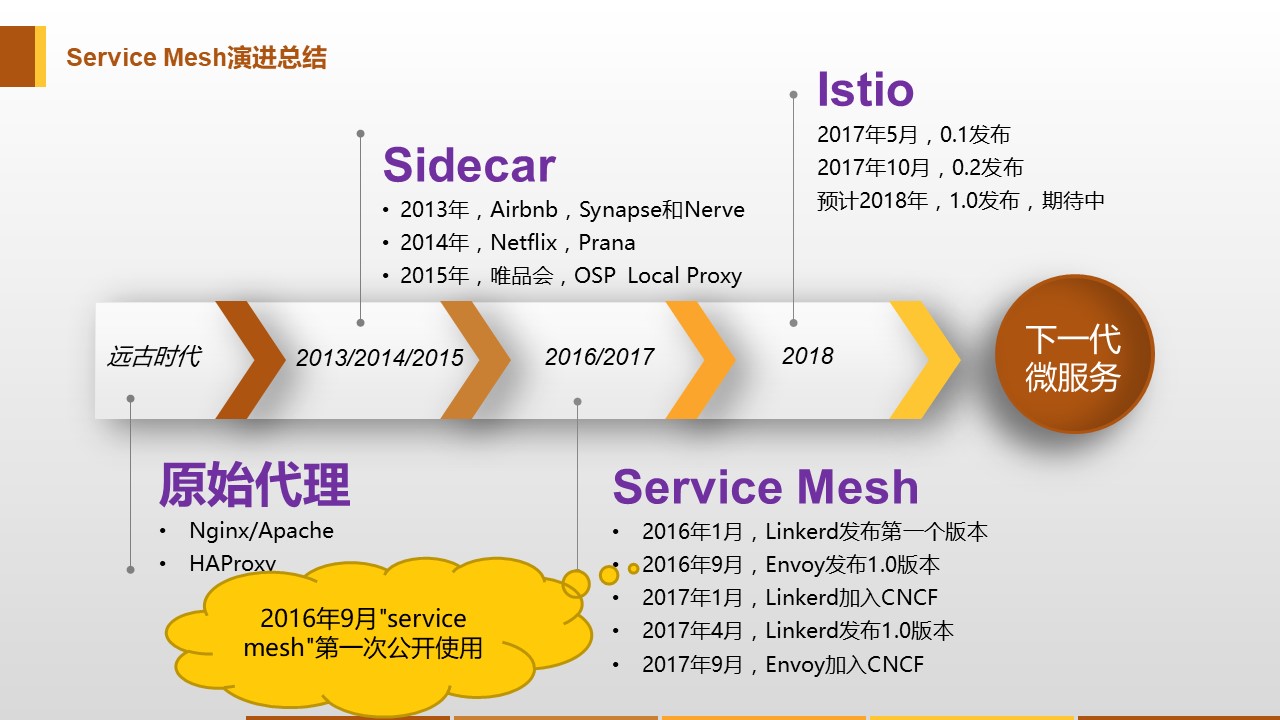
但是这些Sidecar有局限性，都是为特定基础设施而设计，在原有体现搭出来的，无法拆出来给其他系统使用，如Airbnb一定要用到Zookeeper，Netflix用Eureka等。因此，出现了Service Mesh，如下图：



其中来自Google，IBM和Lyft实现的Istio是典型的Service Mesh实现，如下图所示：



Service Mesh从原始的代理，到限制很多的Sidecar，再到通用性的Service mesh，然后到加强管理的Istio，在未来成为下一代的微服务，演进总结如下：

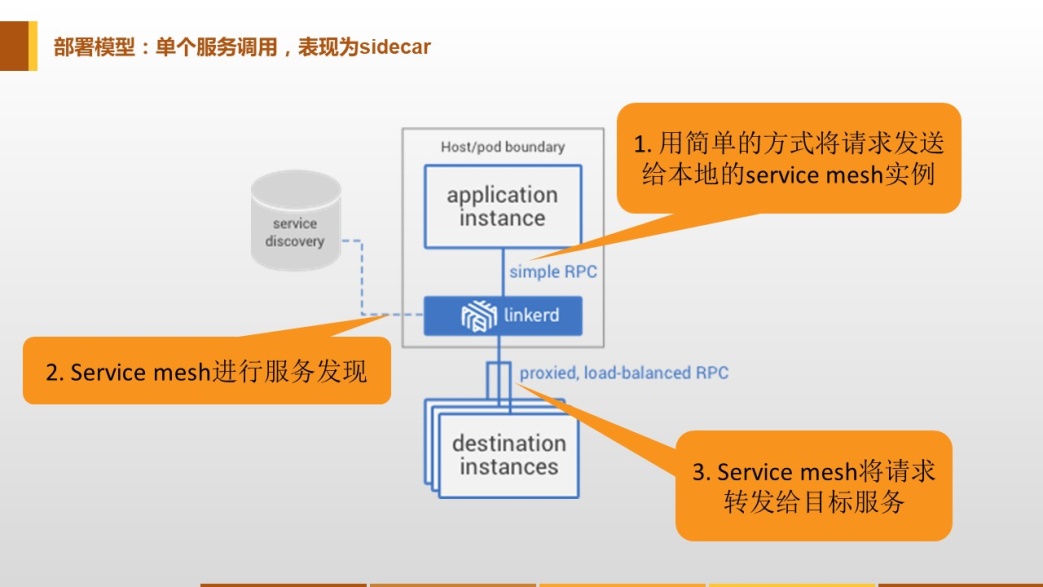


# Service Mesh

Service Mesh是一个基础设施层，功能在于处理服务间通信，职责是负责实现请求的可靠传递，在实践中，服务网格通常实现为轻量级网络代理，通常与应用程序部署在一起，但是对应用程序透明。

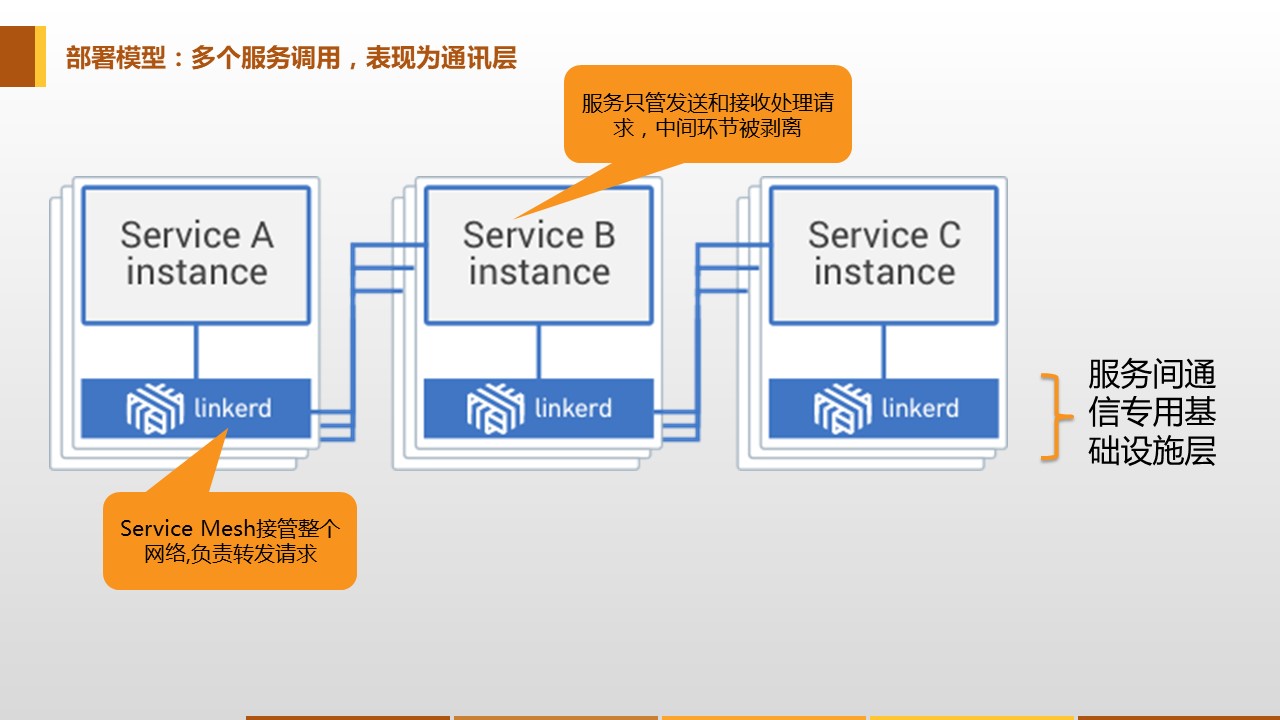
## 2.1 单服务部署模型

对于单个服务调用的部署模型，如下图：



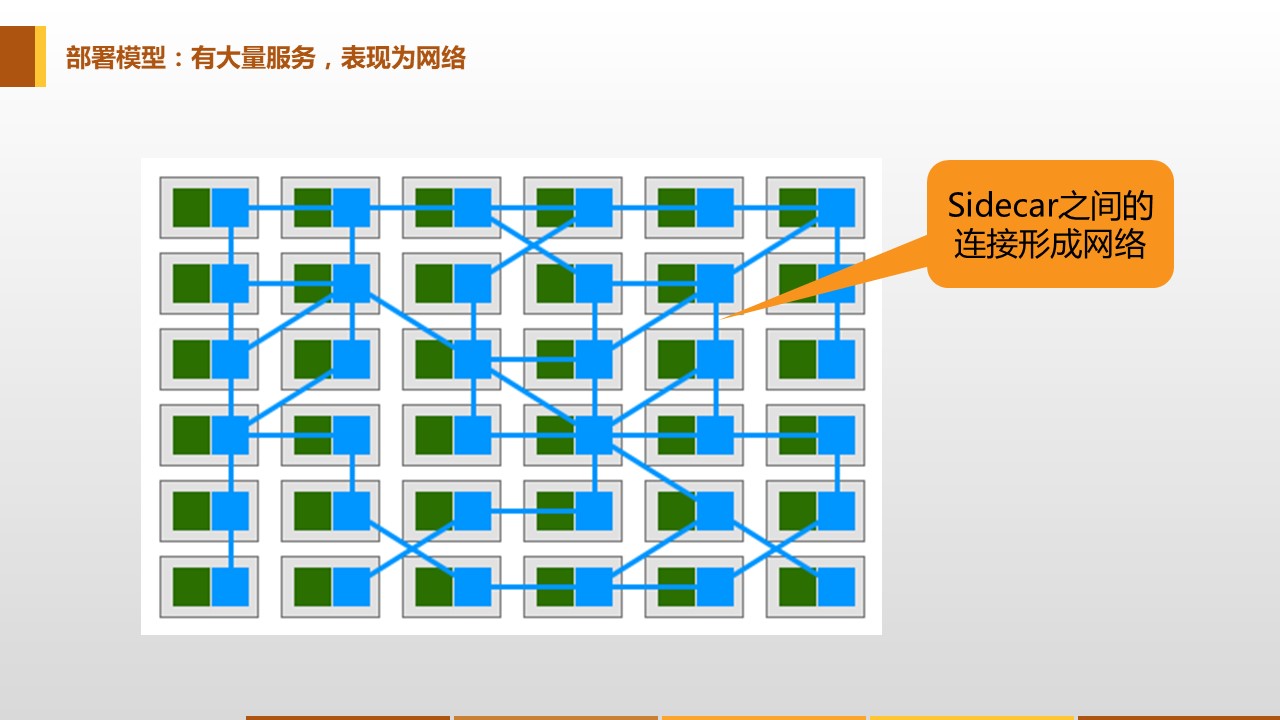
对于单个服务调用的部署模型，对于一个简单的请求，作为请求发起者的客户端应用实例，会首先用简单方式将请求发送本地的Service Mesh实例。这两个独立进程，他们之间是远程调用，Service Mesh会完成完成的服务间调用流程，如服务发现负责均衡，最后将请求发送给目标服务。

## 多服务调用



Service Mesh在所有的服务下面，这一层称之为服务间通讯专用基础设施层，其接管整个网络，把所有请求在服务器之间转发。在这种情况下，服务不再负责传递请求的具体逻辑，只负责完成业务处理。服务间通讯的环节就从应用里面剥离出来，呈现出一个抽象层。

## 网络部署模型



左边方格为应用，右边蓝色方框为Service Mesh，蓝色之间的线条表示服务之间的调用关系。Sidecar之间的连接就会形成一个网络。首先服务网格是在应用之外抽象出来的一个基础设施层，其次功能是实现请求的可靠传递，在部署上为轻量级的网络代理，对应用程序透明。

http://www.uml.org.cn/wfw/201710261.asp

https://buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/

# Envoy

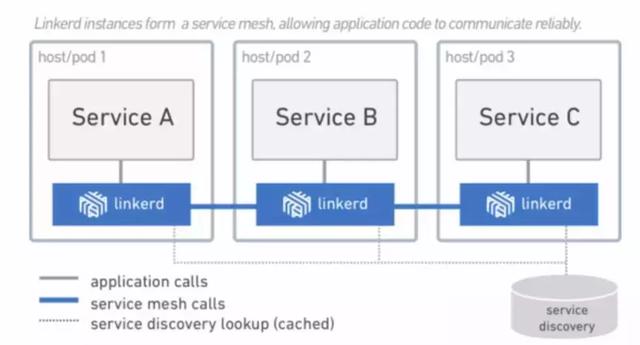
http://blog.csdn.net/zvayivqt0ufji/article/details/78351355

# Linkerd

在云本地应用程序中可靠地传送请求可能非常复杂，如Linkerd这样的Service Mesh通过一系列强大的技术来管理这种复杂性：电路中断、延迟感知的负载均衡、最终一致性服务发现、重试和超时等，这些功能必须全部配合使用，这些功能与其运行的复杂环境之间的交互可能非常微妙，例如通过Linkerd向服务请求时，过程如下：

1. Linkerd根据动态路由规则确定请求时发给哪个服务，比如是发给生产环境里的服务还是发给Staging环境里的服务，发给本地数据中心的服务还是云端服务，是发给最新版本的服务还是发给旧版本的服务，这些路由规则可以动态配置，可以应用在全局的流量上，也可以应用在部分流量上。
2. 在确定了请求的目标服务后，Linkerd从服务发现断点获取相应的服务实例，如果服务实例的信息出现偏差，Linkerd需要决定哪个信息来源更值得信任
3. Linkerd基于某些因素（比如最近处理请求的延迟情况）选择更有可能快速发挥响应的实例
4. Linkerd向选中的实例发送请求，并把延迟情况和响应类型记录下来
5. 如果选中的实例发生宕机、没有响应或无法处理请求，Linkerd就把请求发给另一个实例（前提是请求必须是幂等的）
6. 如果一个实例持续返回错误，Linkerd就会将其从负载均衡池移除，并在稍后定时重试（例如这个实例有可能只是临时发生故障）
7. 如果请求超时，Linkerd会主动发起请求，不会进行额外的重试
8. Linkerd以度量和分布式跟踪的形式捕捉上述行为的每一个方面，并将其发送到集中的度量系统。

除此之外，Linkerd还能启动和终止TLS、执行协议升级、动态调整流量，在数据中心之间进行故障切换



Linkerd的这些特性可以保证局部的弹性和应用层面的弹性，大规模分布式系统有一个共性：局部故障累积到一定程度就会造成系统层面的灾难，Service Mesh的作用就是在底层系统的负载达到上限之前分散流量和快速失效来防止这些故障破坏到整个系统。

Linkerd为云原生应用提供弹性的Service Mesh，透明高性能网络代理，提供服务发现机制、动态路由、错误处理机制及应用运行时可视化，其特性：

* 快速、轻量级、高性能，每秒以最小的时延及负载处理万级请求且易于扩展
* 支持任意开发语言及任意环境
* 提供基于感知时延的负载均衡，通过实时性能数据分发请求，Linkerd工作与RPC层，可根据实时观测到的RPC延迟，要处理请求队列大小决定如何分发请求，提供多种负载均衡算法：P2C：Least Loaded，Peak EWMA等
* 运行时流量路由，通过特定HTTP头进行Per-Request级别路由，动态修改dtab规则实现流量迁移、跨数据中心failover等
* 熔断机制，Fail Fast及Failure Accural
* 插入式服务发现，支持各种服务发现机制：基于文件，Zookeeper，Consul及K8s
* 支持多种协议：HTTP/1.1,HTTP/2，gRPC、Thrift及Mux
* 经过产线测试及验证

https://github.com/linkerd/linkerd-tcp

https://github.com/linkerd/linkerd

https://github.com/doczhcn/linkerd/blob/master/doc/overview/what-is-linkerd.md

https://zhuanlan.zhihu.com/p/32635626

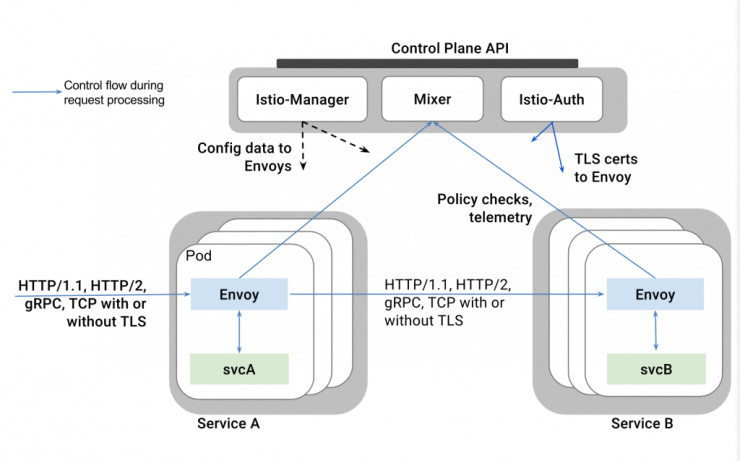
http://blog.csdn.net/hty46565/article/details/78188434?locationNum=5&fps=1

http://blog.csdn.net/wangqingjiewa/article/details/78677912

# Istio

Istio平台的目的是为了帮助开发者和DevOps专业人员管理和保护其基于微服务的应用程序。Istio不需要对其实际应用程序进行修改，也不依赖像gRRPC这样常见的RPC库，可以直接使用代理来拦截微服务之间的所有通信。

其核心在于，基于Lyft的Envoy代理和其内置的服务发现Service Discovery和负载均衡工具，能为特定的微服务添加安全性、可见性和合规性等功能，以阶级传统应用程序分解为一组微服务时而出现的各种问题：



Istio在监控微服务的同时还能向用户提供该项数据及其网络行为的相关数据，以便他们可以快速检测出异常情况并作出回应。该服务还有助于提高开发者的弹性及创造力，从而简化特定环境下的开发过程。

https://www.leiphone.com/news/201705/RwRlyAs7Mi8pqhSb.html

https://github.com/doczhcn/istio

https://my.oschina.net/shurenyun/blog/1553747&quot;

# conduit

# 参考文献：

开源github:

https://github.com/Microsoft/service-fabric

https://github.com/runconduit/conduit

https://github.com/nginmesh/nginmesh

https://github.com/ServicemeshCN/awesome-servicemesh

http://www.infoq.com/cn/articles/2017-service-mesh