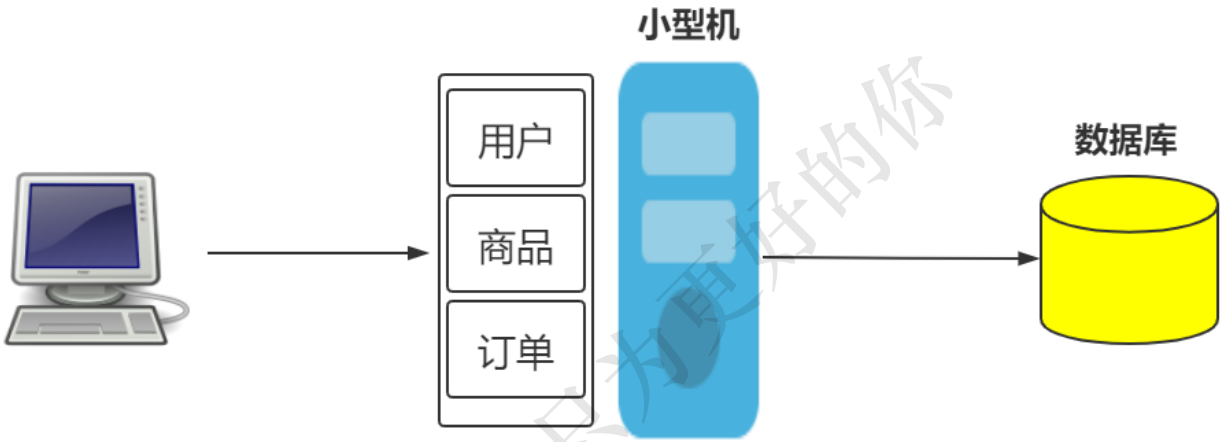
01 分布式架构发展史

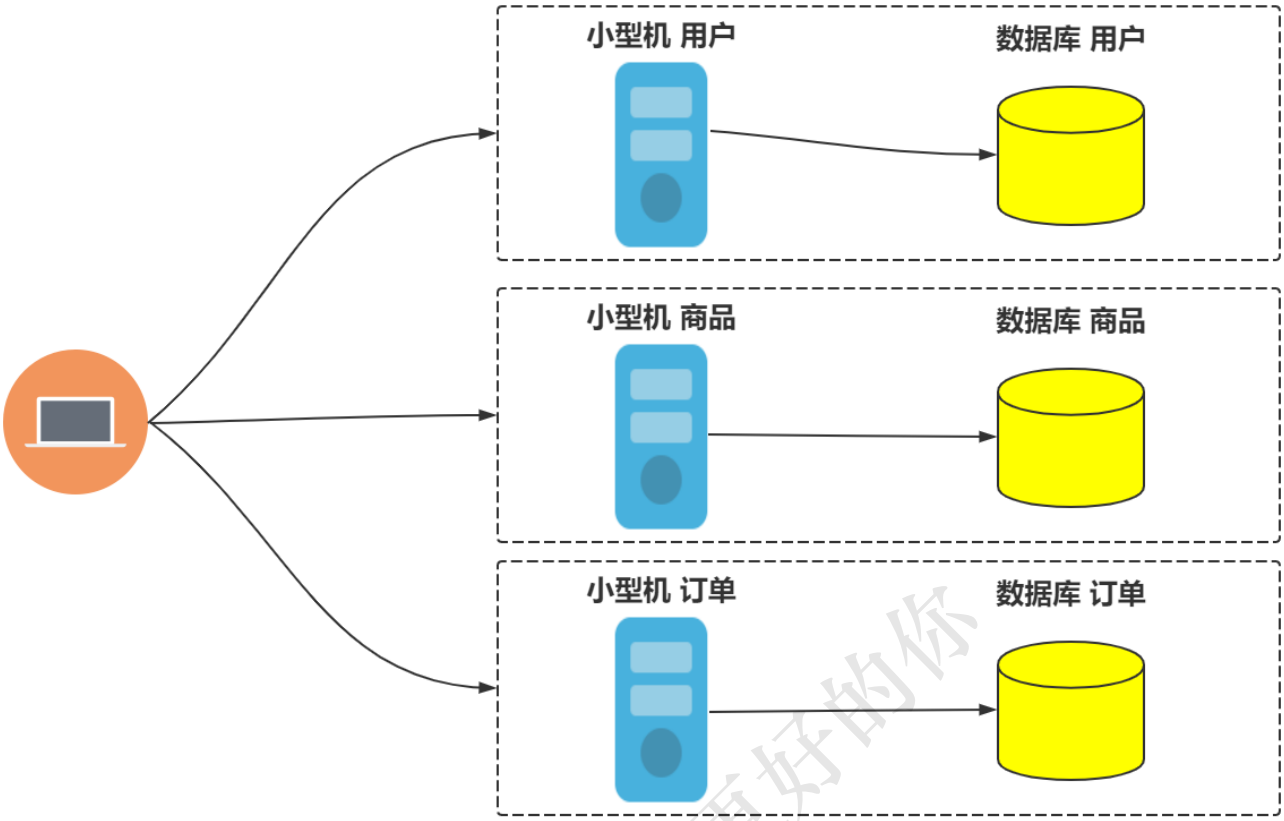
> What?不是聊Service Mesh吗？怎么扯到分布式架构发展史了？别急，一切得从故事刚刚开始的地方聊起。

1.1 单机小型机时代

- 1969年，阿帕网诞生，最初是为了军事目的，后来衍变成Internet
- 2000年左右，互联网在中国开始盛行起来，但是那时候网民数较少，所以多数企业服务单一，采用集中式部署的方式就能满足需求



- 一旦小型机或者数据库出现问题，会导致整个系统的故障，而且功能修改发布也不方便，所以不妨把大系统拆分成多个子系统，比如“用户”，“商品”，“论坛”等，也就是”垂直拆分“，并且每个子系统都有各自的硬件

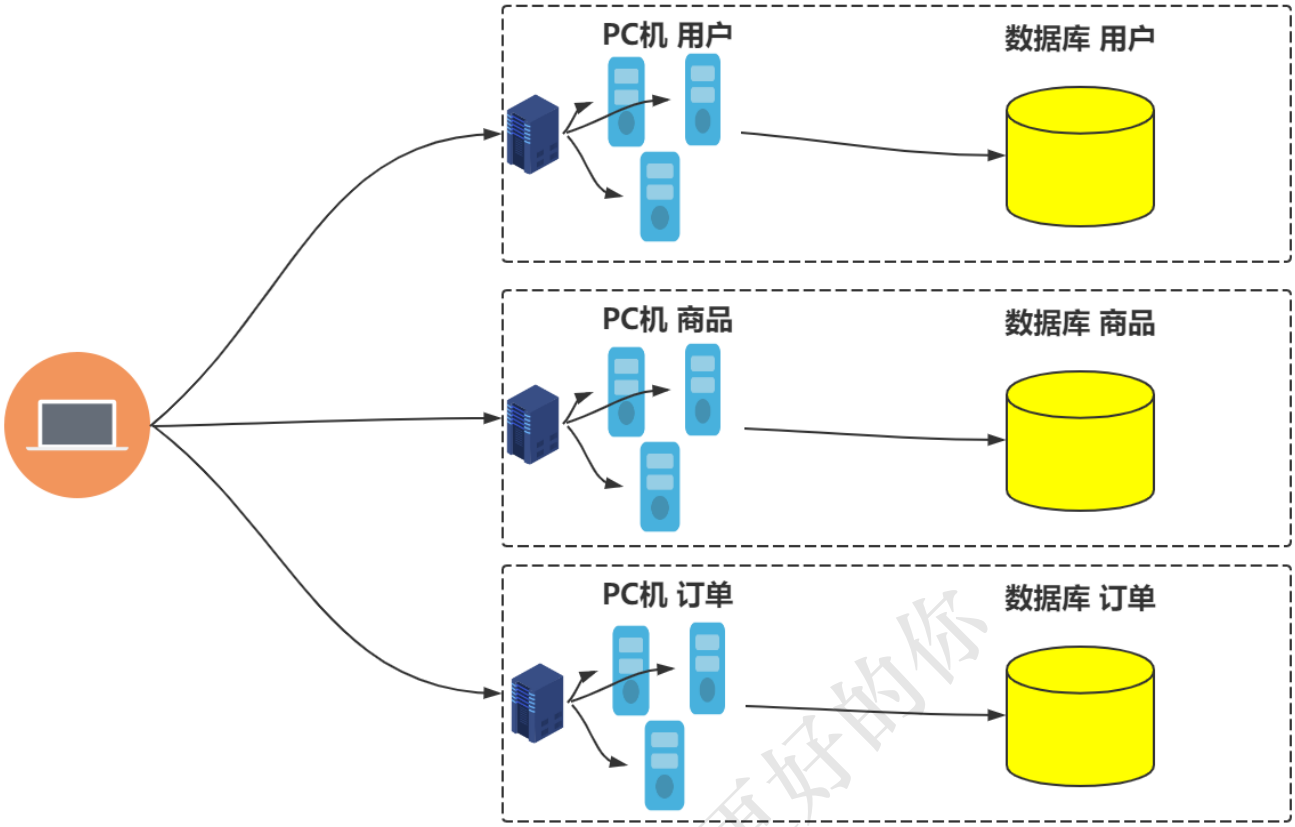


1.2 集群化负载均衡时代

- 用户量越来越大，就意味着需要更多的小型机，价格昂贵，操作维护成本高，不妨把小型机换成普通的PC机，采用多台PC机部署同一个应用的方案，但是此时就需要对这些应用做负载均衡，因为客户端不知道请求哪一个。

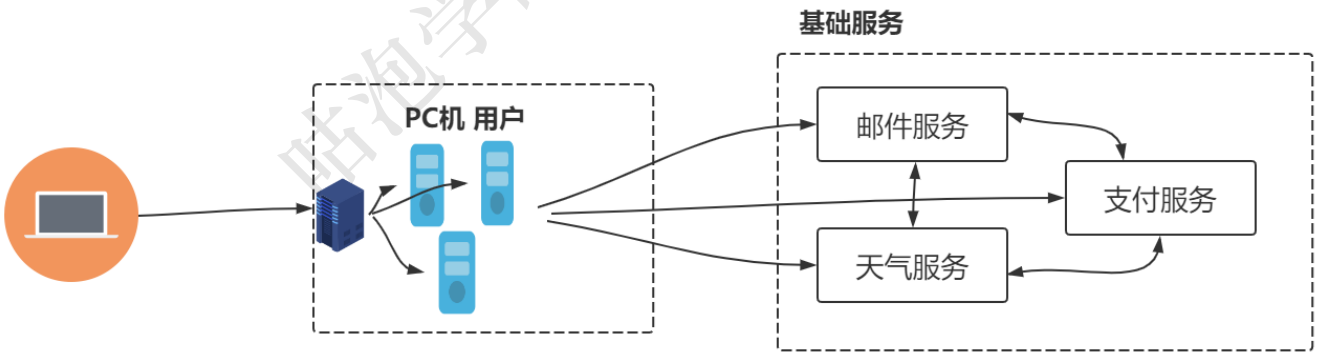
> 2013年5月17日，阿里集团最后一台IBM小型机在支付宝下线。
>
> 这是自2009年“去IOE”战略透露以来，“去IOE”非常重要的一个节点。“去IOE”指的是摆脱掉IT部署中原有的IBM小型机、Oracle数据库以及EMC存储的过度依赖。告别最后一台小机，意味着整个阿里集团尽管还有一些Oracle数据库和EMC存储，但是IBM小型机已全部消失。

- 负载均衡可以分为硬件和软件，硬件比如F5，软件比如LVS(Linux Virtual Server)。负载均衡的思路是对外暴露一个统一的接口，然后根据用户的请求进行对应规则的转发。同时负载均衡还可以做健康检查、限流等
- 有了负载均衡，后端的应用就可以根据流量的大小进行动态的扩缩容了，也就是“水平扩展”



1.3 服务化时代

- 此时发现在用户，商品和订单等中有重复的功能，比如登录、注册、邮件等。一旦项目大了，集群部署多了，这些重复的功能无疑是浪费资源，不妨把重复的功能抽取出来，起个名字叫“服务Service”
- 其实对于“服务”现在已经比较广泛了，比如“基础设施即服务IaaS”、“平台即服务PaaS”、“软件即服务SaaS”等



- 这时候急需解决的就是服务之间的调用问题，RPC(Remote Procedure Call)，同时这种调用关系得维护起来，比如某个服务在哪里，是不是得知道？所以不仅仅要解决服务调用的问题，还要解决服务治理的问题，比如像Dubbo，默认采用Zookeeper作为注册中心，Spring Cloud使用Eureka作为注册中心
- 当然，要关注的还有很多，比如限流、降级、负载均衡、容错等

1.4 分布式微服务时代

- 在分布式架构下，服务可能拆分的没那么细，可以进一步的拆分
- Microservices

> https://martinfowler.com/articles/microservices.html

- Java中微服务主流解决方案Spring Cloud，Spring Cloud可以通过引入几个注解，写少量代码完成微服务架构中的开发，相比Dubbo而言方便很多。并且使用的是HTTP协议，所以对多语言也可以很好地支持

> https://spring.io/projects/spring-cloud
>
> spring-cloud-bus
>
> spring-cloud-consul
>
> spring-cloud-config
>
> spring-cloud-netflix:eureka、hystrix、feign、zuul、ribbon等
>
> .....

1.5 服务网格时代

> **然后微服务时代有了Spring Cloud就完美了吗？不妨想一想会有哪些问题**
>
> （1）最初是为了业务而写代码，比如登录功能、支付功能等，到后面会发现要解决网络通信的问题，虽然有Spring Cloud里面的组件帮我们解决了，但是细想一下它是怎么解决的？引入以来dependency，加注解，写配置，最后将这些非业务功能的代码打包一起部署，和业务代码在一起，称为“侵入式框架”；
>
> （2）微服务中的服务支持不同语言开发，维护不同语言和非业务代码的成本；
>
> （3）业务代码开发者应该把更多的精力投入到业务熟悉度上，而不应该是非业务上，Spring Cloud虽然能解决微服务领域的很多问题，但是学习成本还是较大的；
>
> （4）对于Spring Cloud而言，并不是微服务领域的所有问题都有对应的解决方案，也就是功能有限，比如Content Based Routing和Version Based Routing。当然可以将Spring Cloud和Service Mesh结合起来使用，也就是对SC的补充、扩展和加强等；
>
> （5）互联网公司产品的版本升级是非常频繁的，为了维护各个版本的兼容性、权限、流量等，因为Spring Cloud是“代码侵入式的框架”，这时候版本的升级就注定要让非业务代码一起，一旦出现问题，再加上多语言之间的调用，工程师会非常痛苦；
>
> （6）其实大家有没有发现，服务拆分的越细，只是感觉上轻量级解耦了，但是维护成本却越高了，那么怎么办呢？网络的问题当然还是要交给网络本身来解决

1.5.1 问题解决思路

- 本质上是要解决服务之间通信的问题，不应该将非业务的代码融合到业务代码中
- 也就是从客户端发出的请求，要能够顺利到达对应的服务，这中间的网络通信的过程要和业务代码尽量无关

> 通信的问题：服务发现、负载均衡、版本控制、蓝绿部署等

- 在很早之前的单体架构中，其实通信问题也是需要写在业务代码中的，那时候怎么解决的呢？

> 把网络通信，流量转发等问题放到了计算机网络模型中的TCP/UDP层，也就是非业务功能代码下沉

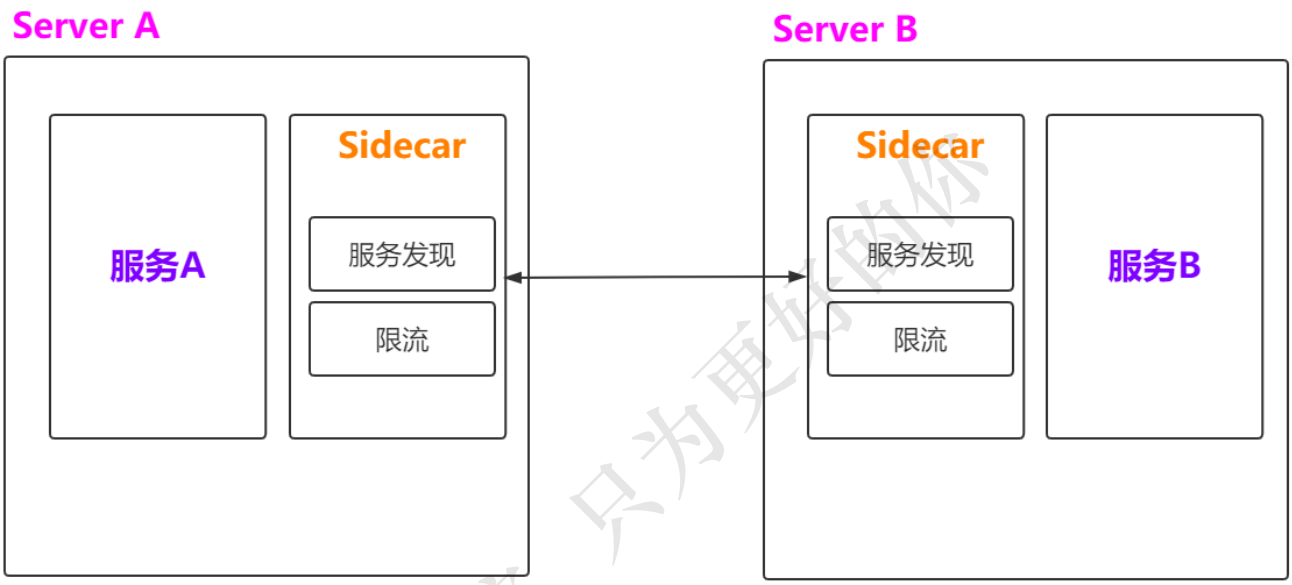
- 那就不妨把这些通信的问题都交给计算机网络模型组织去解决咯？别人肯定不会答应，怎么办？

既然不答应，那我们就自己想办法解决通信问题，搞一些产品岂不快哉？没错，代理嘛

- 不妨这样在帮每个服务配置一个代理，所有通信的问题都交给这个代理去做
- 大家肯定接触过，比如最初Nginx、HaProxy等，其实它们做反向代理把请求转发给其他服务器，也就为Service Mesh的诞生和完成提供了一个解决思路

1.5.2 一些公司对于代理的探索Sidecar

> 很多公司借鉴了Proxy模式，推出了Sidecar的产品，比如像14年Netflix的Prana、15年唯品会的local proxy
>
> 其实Sidecar模式和Proxy很类似，但是Sidecar功能更全面，也就是Sidecar功能和侵入式框架的功能对应

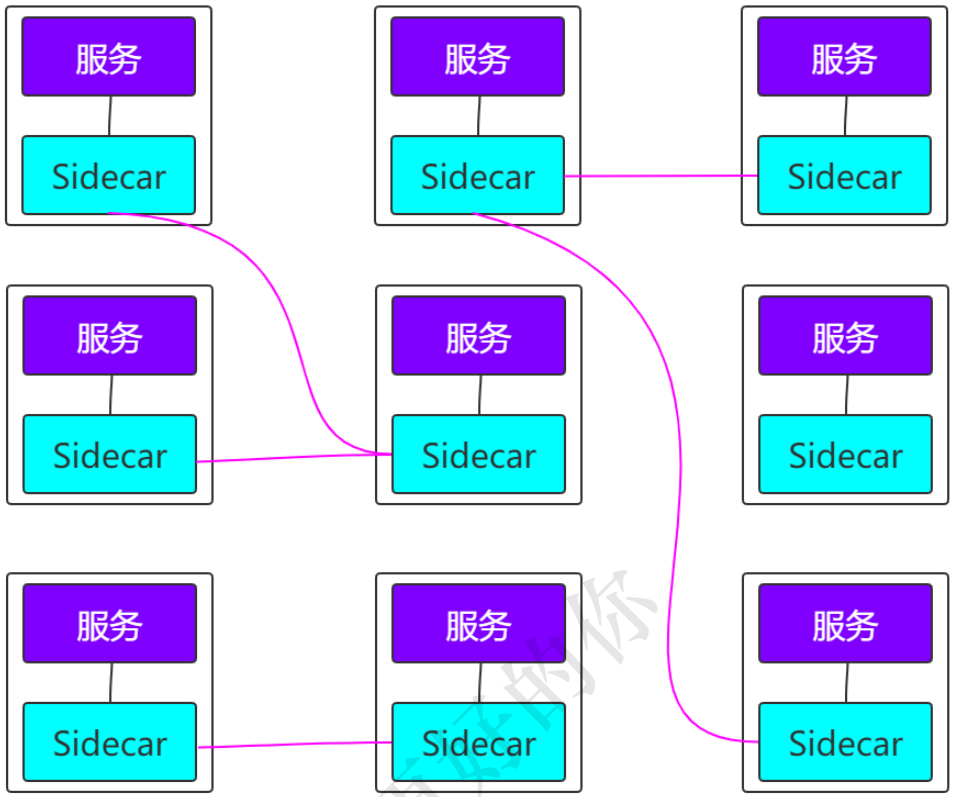


问题：这种Sidecar是为特定的基础设施而设计的，也就是跟公司原有的框架技术绑定在一起，不能成为通用性的产品，所以还需要继续探索。

1.5.3 Service Mesh之Linkerd

> 2016年1月，离开Twitter的基础设施工程师William Morgan和Oliver Gould，在github上发布了Linkerd 0.0.7版本，从而第一个Service Mesh项目由此诞生。并且Linkerd是基于Twitter的Finagle开源项目，实现了通用性。
>
> 后面又出现了Service Mesh的第二个项目Envoy，并且在17年都加入了CNCF项目。

Linkerd设计思想



小结：Linkerd解决了通用性问题，在Linkerd思想要求所有的流量都走Sidecar，而原来的Sidecar方式可以走可以直连，这样一来Linkerd就帮业务人员屏蔽了通信细节，也不需要侵入到业务代码中，让业务开发者更加专注于业务本身。

问题：但是Linkerd的设计思想在传统运维方式中太难部署和维护了，所以就后来没有得到广泛的关注，其实主要的问题是Linkerd只是实现了数据层面的问题，但没有对其进行很好的管理。

1.5.4 Service Mesh之Istio

> 由Google、IBM和Lyft共同发起的开源项目，17年5月发布0.1 release版本，17年10月发布0.2 release版本，18年7月发布1.0 release版本。
>
> 很明显Istio不仅拥有“数据平面（Data Plane）”，而且还拥有“控制平面（Control Plane），也就是拥有了数据接管与集中控制能力。
>
> 官网why use Istio?：https://istio.io/docs/concepts/what-is-istio/#why-use-istio
>
> ```
> Istio makes it easy to create a network of deployed services with load balancing,
> service-to-service authentication, monitoring, and more, with few or no code changes
> in service code. You add Istio support to services by deploying a special sidecar
> proxy throughout your environment that intercepts all network communication between
> microservices, then configure and manage Istio using its control plane functionality
> ```

02 Service Mesh

2.1 What's a service mesh[William]

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

```
A service mesh is a dedicated infrastructure layer for making service-to-service communication safe, fast, and reliable. If you’re building a cloud native application, you need a service mesh.
```

2.2 What is a service mesh?[Istio]

> istio官网：https://istio.io/docs/concepts/what-is-istio/#what-is-a-service-mesh

```
The term service mesh is used to describe the network of microservices that make up such applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary rollouts, rate limiting, access control, and end-to-end authentication.
```

2.3 Linkerd和Istio发展历程

- Mircroservices

```
Martin Fowler
14年提出
https://martinfowler.com/articles/microservices.html
```

- Linkerd

```
William Morgan[Buoyant]
Scala语言编写，运行在JVM中，Service Mesh名词的创造者
16年01月15号，0.0.7发布
17年01月23号，加入CNCF
17年04月25号，1.0版本发布
```

- Envoy

```
C++语言编程[Lyft]
16年09月13号，1.0发布
17年09月14号，加入CNCF
```

- Istio

```
Google、IBM、Lyft发布0.1版本
```

2.4 国内发展情况

- 蚂蚁进入的SOFA

```
全称Scalable Open Financial Architecture
前身是SOFA RPC
18年07月正式开源
```

- 腾讯的Tencent Service Mesh
- 华为的CSE Mesher

`小结`：基本上都借鉴了Sidecar、Envoy和Istio等设计思想

> **最终目的**：
>
> TCP/IP协议解决了计算机之间连接的问题，其实我们很少感知到它的存在。
>
> 而目前的服务治理也面临相似的问题，也就是要让计算机中的服务更好的连接起来，而且要做到业务代码尽可能无感知。

2.5 为何Service Mesh能够迅速走红?

随着微服务和容器化技术的发展，使得开发和运维的方式变得非常方便。

从而让服务能够方便地迁移到不同的云平台上。

2.6 回顾一下Service Mesh的发展历程

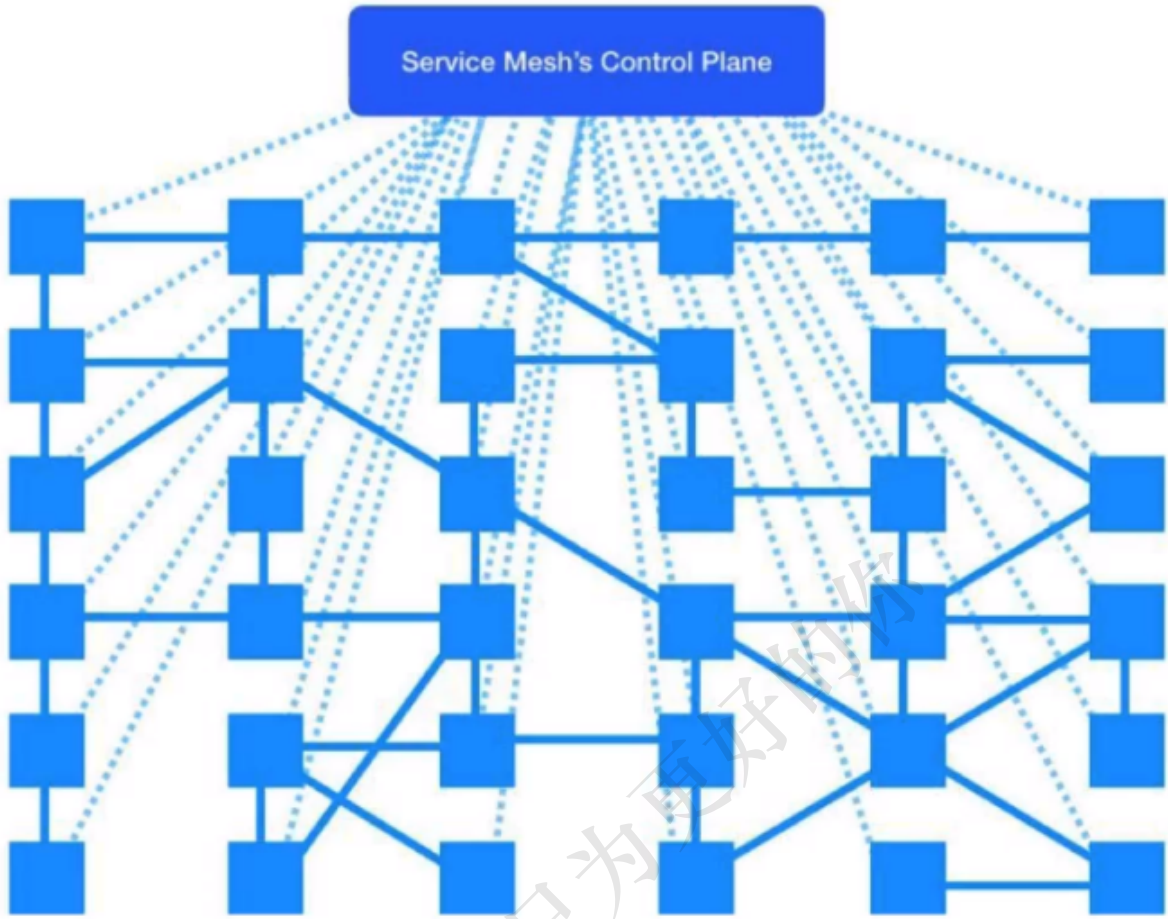
- 借鉴反向代理，对Proxy模式进行探索，让非业务的代码下沉
- 使用Sidecar弥补Proxy模式功能的不足，解决“侵入式框架”中非业务代码的问题
- Linkerd解决传统Sidecar模式中通用性的问题
- Istio增加了控制平面，解决整个系统中的流量完全控制的问题

03 Istio

> 基于Sidecar模式、数据平面和控制平台、主流Service Mesh解决方案
>
> `官网`：https://istio.io/
>
> `github`：https://github.com/istio

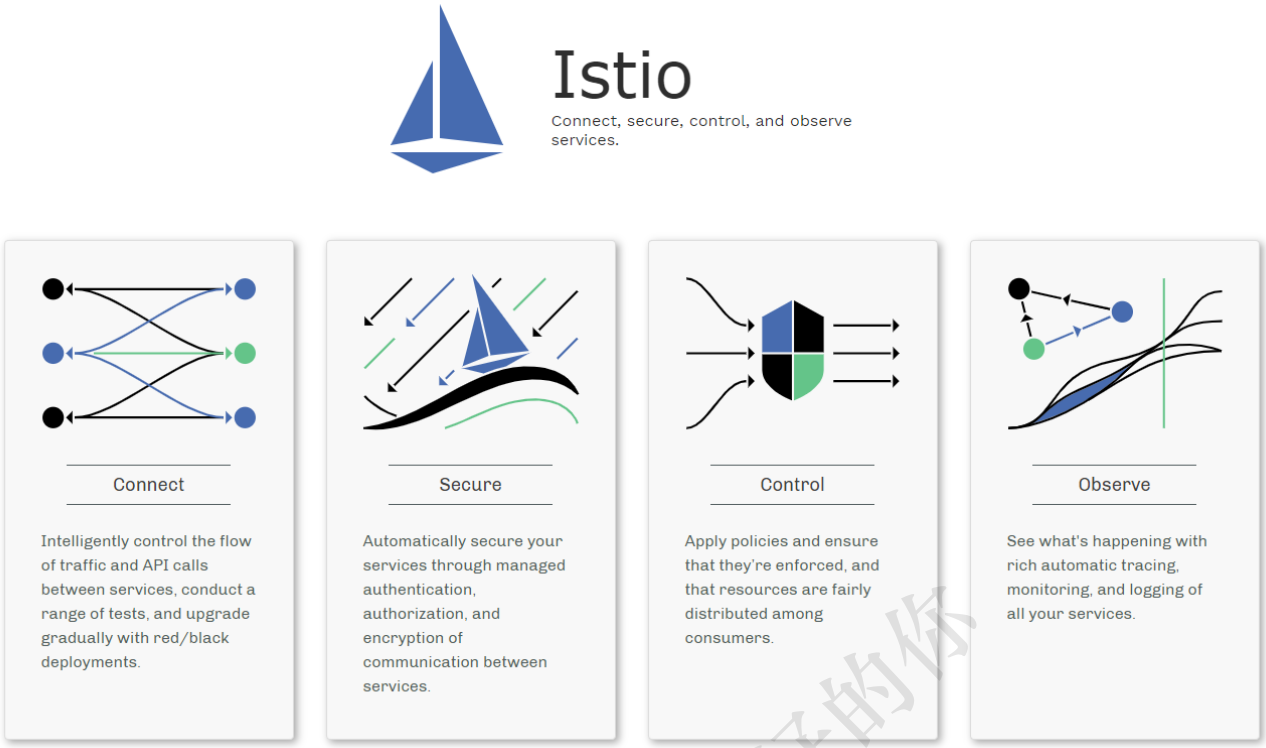
3.1 整体感受

> `数据平面和控制平面`
>
> Sidecar的方式解决了数据通信的问题，而在Istio中还加入了控制平面，所有的流量都能够有效地被控制，也就是通过控制平面可以控制整个系统。



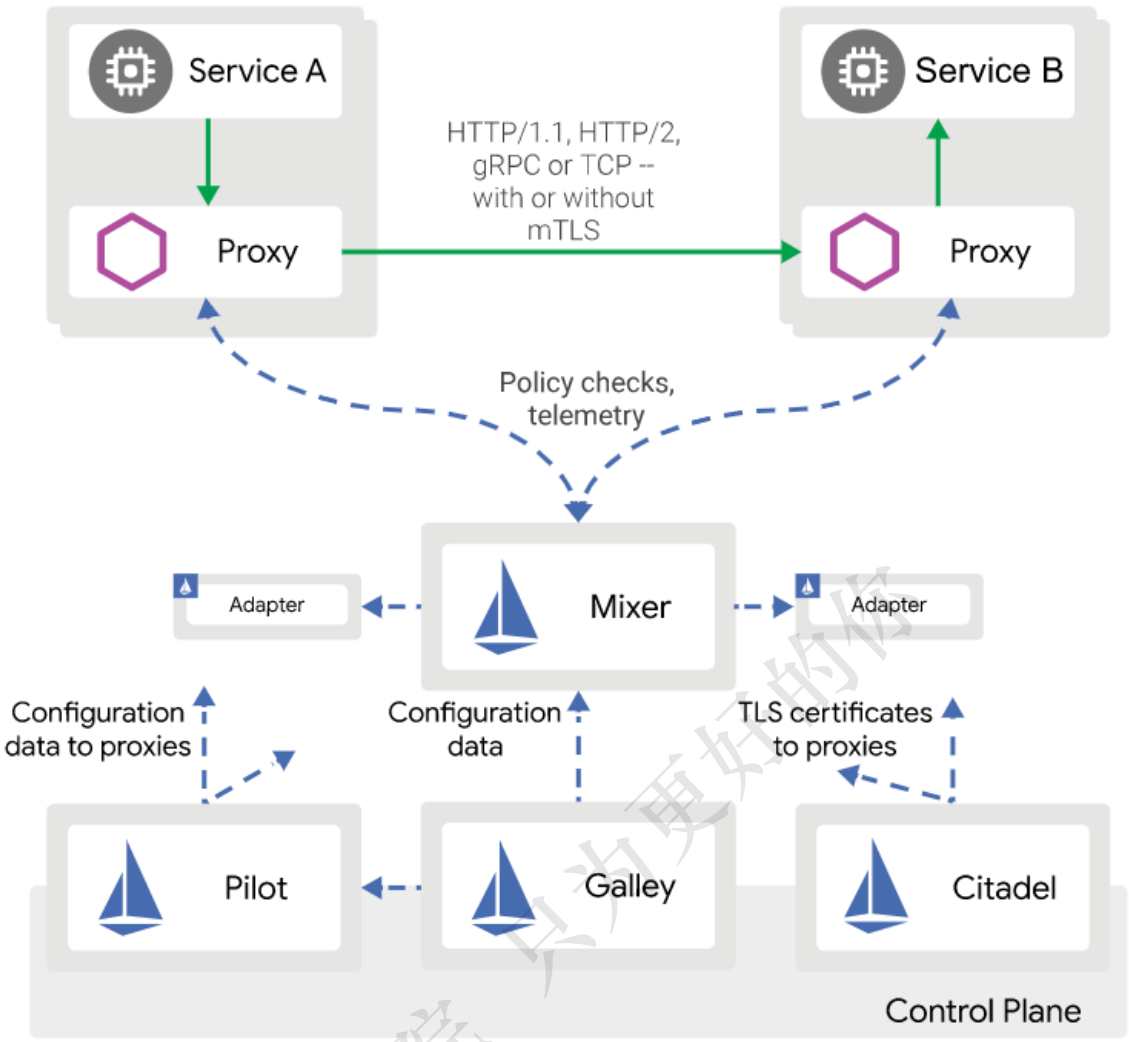
3.2 在Istio中到底能解决哪些问题

> (1) 针对HTTP、gRPC、WebSocket等协议的自动负载均衡
>
> (2) 故障的排查、应用的容错、众多路由
>
> (3) 流量控制、全链路安全访问控制与认证
>
> (4) 请求遥测、日志分析以及全链路跟踪
>
> (5) 应用的升级发布、频率限制和配合等



3.3 Architecture

> 有了感性的认知和功能了解之后，接下来就要考虑落地的问题了，也就是Istio的架构图该如何设计。
>
> 说白了就是根据数据平面和控制平面的理念，该怎么设计架构图，这个官方已经帮我们考虑好了。
>
> `Architecutre`：https://istio.io/docs/ops/deployment/architecture/
>
> An Istio service mesh is logically split into a **data plane** and a **control plane**.
>
> - The **data plane** is composed of a set of intelligent proxies (Envoy) deployed as sidecars. These proxies mediate and control all network communication between microservices along with Mixer, a general-purpose policy and telemetry hub.
> - The **control plane** manages and configures the proxies to route traffic. Additionally, the control plane configures Mixers to enforce policies and collect telemetry.



Istio Architecture

04 安装Istio

4.1 Getting Started

> 官网：https://istio.io/docs/setup/getting-started/
>
> - Set up your platform
> - Download the release
> - Install Istio

4.1.1 Set up your platform

> **这里我们使用之前安装好的Kubernetes1.14版本**

4.1.2 Download the release

> (1)Go to the Istio release page...

> 放到master节点

```
# 方式一：到github上下载
    https://github.com/istio/istio/releases

# 方式二：macOs or Linux System直接下载，这个地址下载国内下载会很慢，不推荐
    curl -L https://istio.io/downloadIstio | sh -

# 方式三：
    上课用的istio，在网盘/课件源码/istio-1.0.6-linux.tar.gz
```

> (2)解压 tar -zxvf istio-1.x.y.tar.gz，进入到istio文件目录下
>
> `The installation directory contains:`
>
> - Installation YAML files for Kubernetes in `install/kubernetes`
> - Sample applications in `samples/`
> - The `istioctl` client binary in the `bin/` directory. `istioctl` is used when manually injecting Envoy as a sidecar proxy.

> (3)将itsioctl添加到环境变量中

```
export PATH=$PWD/bin:$PATH
```

4.1.3 Install Istio

4.1.3.1 Kubernetes CRD

crd这块课程中没有演示安装，故意埋下的坑，小伙伴们一定记得先安装哦！！！

Kubernetes平台对于分布式服务部署的很多重要的模块都有系统性的支持，借助如下一些平台资源可以满足大多数分布式系统部署和管理的需求。

但是在不同应用业务环境下，对于平台可能有一些特殊的需求，这些需求可以抽象为Kubernetes的扩展资源，而Kubernetes的CRD(CustomResourceDefinition)为这样的需求提供了轻量级的机制，保证新的资源的快速注册和使用。在更老的版本中，TPR(ThirdPartyResource)是与CRD类似的概念，但是在1.9以上的版本中被弃用，而CRD则进入的beta状态。

> Istio就是使用CRD在Kubernetes上建构出一层Service Mesh的实现

> istio-1.0.6/install/kubernetes/helm/istio/templates/crds.yaml

kubectl apply -f crds.yaml

4.1.3.2 提前准备好镜像

> istio-1.0.6/install/kubernetes/istio-demo.yaml
>
> 上述istio-demo.yaml文件中有很多镜像速度较慢，大家记得下载我的，然后tag，最好保存到自己的镜像仓库，以便日后使用。

> (1)从阿里云镜像仓库下载
>
> **记得是所有节点都要下载，因为pod会调度在不同的节点上**

```
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxy_init:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/hyperkube:v1.7.6_coreos.0
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/galley:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxyv2:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/grafana:5.2.3
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/mixer:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/pilot:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/prometheus:v2.3.1
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/citadel:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/servicegraph:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/sidecar_injector:1.0.6
docker pull registry.cn-hangzhou.aliyuncs.com/istio-k8s/all-in-one:1.5
```

> (2)把阿里云镜像仓库的镜像打成原有的tag
>
> **所有节点**

```
docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxy_init:1.0.6
docker.io/istio/proxy_init:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/hyperkube:v1.7.6_coreos.0
quay.io/coreos/hyperkube:v1.7.6_coreos.0

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/galley:1.0.6
docker.io/istio/galley:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxyv2:1.0.6
docker.io/istio/proxyv2:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/grafana:5.2.3 grafana/grafana:5.2.3

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/mixer:1.0.6
docker.io/istio/mixer:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/pilot:1.0.6
docker.io/istio/pilot:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/prometheus:v2.3.1
docker.io/prom/prometheus:v2.3.1

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/citadel:1.0.6
docker.io/istio/citadel:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/servicegraph:1.0.6
docker.io/istio/servicegraph:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/sidecar_injector:1.0.6
docker.io/istio/sidecar_injector:1.0.6

docker tag registry.cn-hangzhou.aliyuncs.com/istio-k8s/all-in-one:1.5
docker.io/jaegertracing/all-in-one:1.5
```

> (3)删除阿里云镜像仓库下载的镜像
>
> **所有节点**

```
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxy_init:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/hyperkube:v1.7.6_coreos.0
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/galley:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/proxyv2:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/grafana:5.2.3
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/mixer:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/pilot:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/prometheus:v2.3.1
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/citadel:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/servicegraph:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/sidecar_injector:1.0.6
docker rmi registry.cn-hangzhou.aliyuncs.com/istio-k8s/all-in-one:1.5
```

4.1.3.3 安装istio核心组件

> (1)根据istio-1.0.6/install/kubernetes/istio-demo.yaml创建资源

```
kubectl apply -f istio-demo.yaml
```

> (2)查看核心组件资源

```
kubectl get pods -n istio-system
kubectl get svc -n istio-system   # 可以给某个service配置一个ingress规则，访问试试
```

这样一来，istio的核心组件就安装完成咯

4.2 初步感受istio

4.2.1 手动注入sidecar

> 大家都知道，K8s中是通过Pod来部署业务，流量的进出是直接跟Pod打交道的
>
> 那在Istio中如何体现sidecar的作用呢?
>
> 思路：在Pod中除了业务的container，再增加一个container为sidecar岂不是快哉?

> (1)准备一个资源
>
> 网盘/课件源码/first-istio.yaml

```
kubectl apply -f first-istio.yaml
kubectl get pods    -> # 注意该pod中容器的数量
kubectl get svc
curl 192.168.80.227:8080/dockerfile
curl 10.105.179.205/dockerfile
```

> (2)删除上述资源，重新创建，使用手动注入sidecar的方式

```
kubectl delete -f first-istio.yaml
istioctl kube-inject -f first-istio.yaml | kubectl apply -f -
```

> (3)查看资源

```
kubectl get pods   -> # 注意该pod中容器的数量
kubectl get svc
kubectl describe pod first-istio-cc5d65fc-rzdxx -> # Containers:first-istio & istio-proxy
```

> (4)删除资源

```
istioctl kube-inject -f first-istio.yaml | kubectl delete -f -
```

这样一来，就手动给pod中注入了sidecar的container，也就是envoy sidecar

`In fact`：**其实这块是先改变了yaml文件的内容，然后再创建的pod：kubectl get pod pod-name -o yaml**

4.2.2 自动注入sidecar

> 上述每次都进行手动创建肯定不爽咯，一定会有好事之者来做这件事情。
>
> 这块需要命名空间的支持，比如有一个命名空间为istio-demo，可以让该命名空间下创建的pod都自动注入sidecar。

> (1)创建命名空间

```
kubectl create namespace istio-demo
```

> (2)给命名空间加上label

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

> (3)创建资源

```
kubectl apply -f first-istio.yaml -n istio-demo
```

> (4)查看资源

```
kubectl get pods -n istio-demo
kubectl describe pod pod-name -n istio-demo
kubectl get svc -n istio-demo
```

> (5)删除资源

```
kubectl delete -f first-istio.yaml -n istio-demo
```

4.2.3 感受prometheus和grafana

> 其实istio已经默认帮我们安装好了grafana和prometheus，只是对应的Service是ClusterIP，我们按照之前K8s的方式配置一下Ingress访问规则即可，但是要提前有Ingress Controller的支持哦，这块我就不多阐述了，遗忘的小伙伴可以看下前面的Kubernetes课程。

> (1)访问prometheus
>
> 搜索istio-demo.yaml文件

```
kind: Service
name: http-prometheus   # 主要是为了定位到prometheus的Service
```

> (2)配置prometheus的Ingress规则
>
> 网盘/课件源码/prometheus-ingress.yaml

```
#ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress
  namespace: istio-system
spec:
  rules:
  - host: prometheus.istio.itcrazy2016.com
    http:
      paths:
      - path: /
        backend:
          serviceName: prometheus
          servicePort: 9090
```

> (3)访问grafana
>
> istio-demo.yaml文件

```
kind: Service
targetPort: 3000
```

> (4)配置grafana的Ingress规则
>
> 网盘/课件源码/grafana-ingress.yaml

```
#ingress
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress
  namespace: istio-system
spec:
  rules:
  - host: grafana.istio.itcrazy2016.com
    http:
```

```
      paths:
      - path: /grafana
        backend:
          serviceName: grafana
          servicePort: 3000
```

> (5)根据两个ingress创建资源
>
> 并且访问测试
>
> istio.itcrazy2016.com/prometheus
>
> istio.itcrazy2016.com/grafana

```
kubectl apply -f prometheus-ingress.yaml
kubectl apply -f grafana-ingress.yaml
kubectl get ingress -n istio-system
```