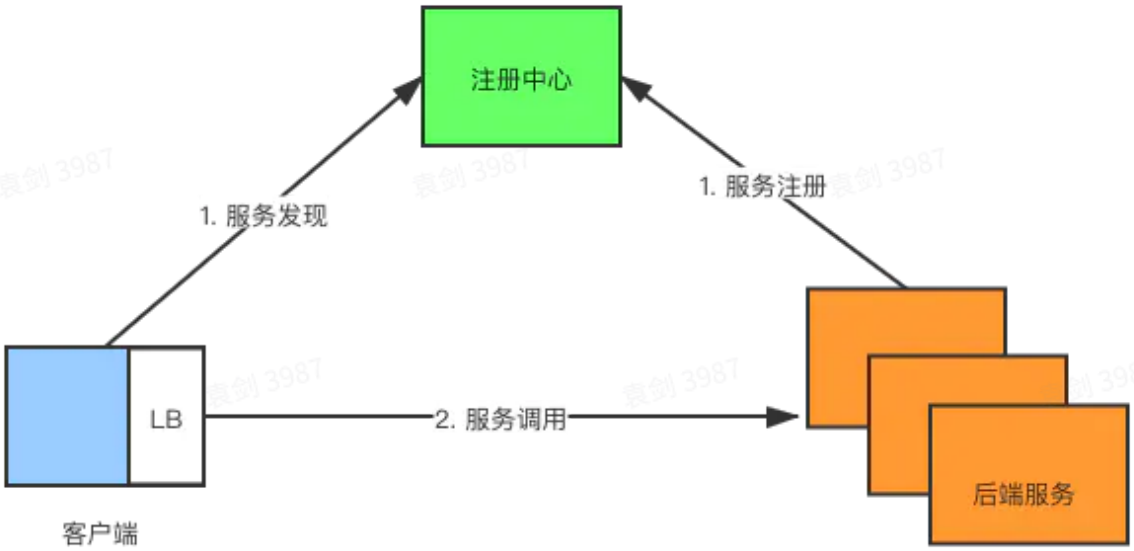
新人串讲 - ServiceMesh理解和介绍

传统流量治理模式

富客户端模式

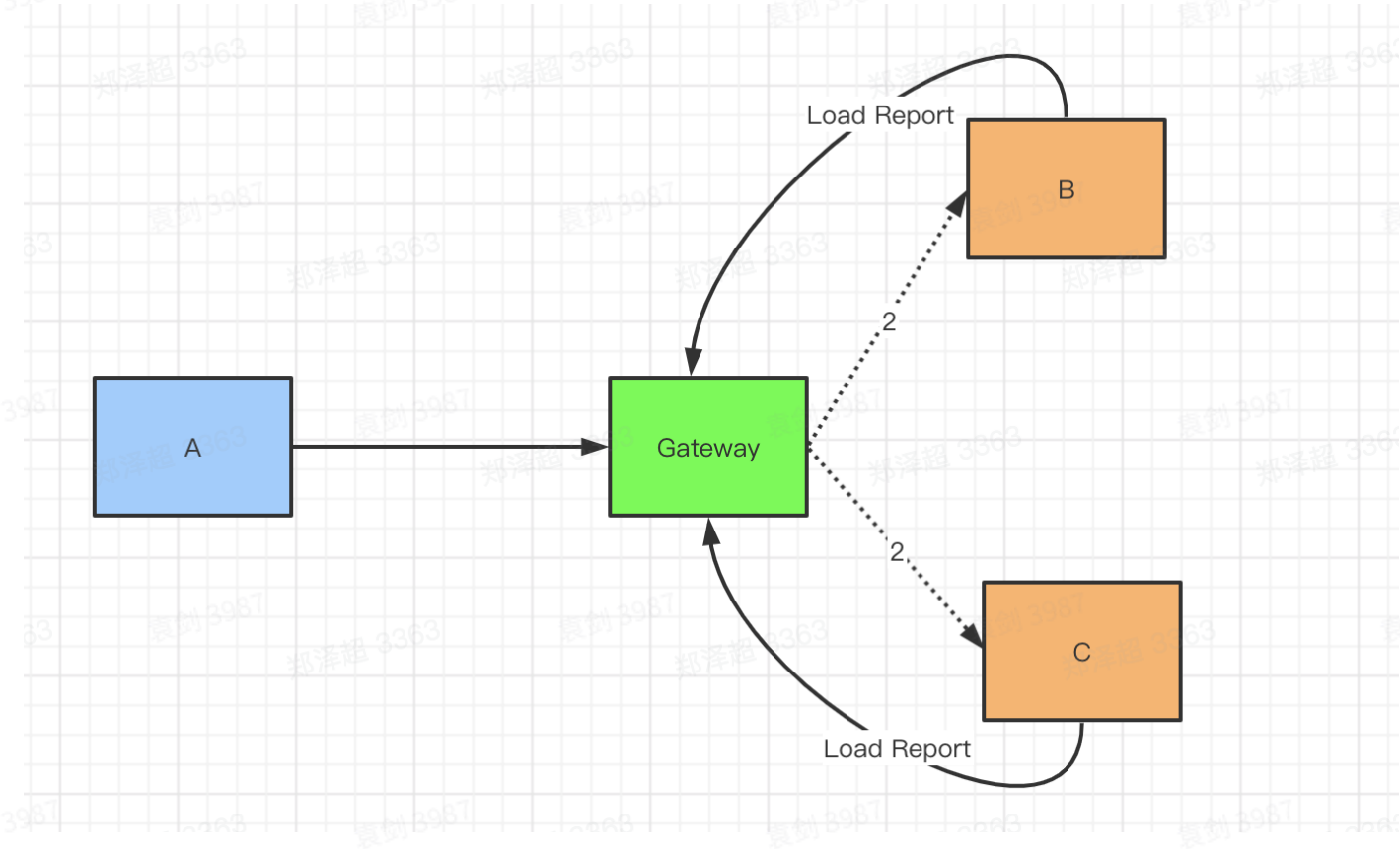


通过客户端内置对应的服务治理模块实现，该方式实现简单，成本低，不需要任何外部的负载均衡设备和进程。大家熟知的一些微服务框架，例如SpringCloud、Dubbo等都是使用的这种流量治理模式。客户端通常集成负载均衡、服务发现、链路追踪、健康检查、指标上报等基础功能。适合客户端和服务端流量比较大的场景，减少端到端的额外跳数。

缺点：

1. 语言绑定的。当团队内使用多语言开发时，以及与业务代码耦合度高造成维护升级困难，业务接入成本高。负载均衡算法实现一般简单，无法没有根据丰富服务端的负载信息(CPU load、mem used)等进行动态调整的负载均衡能力。

服务网关模式



一般也称作服务端负载均衡，但并不是指其部署在服务端进程内。而是该后端服务地址一般对客户端透明。网关通过接受客户端请求，并根据配置的负载均衡方式进行负载均衡转发到某一个后端服务节点上，具体的工作原理如下：

1. 客户端向具体网关发出请求，上图步骤1。
2. 网关通过负载均衡算法，选择其中的一台后端服务节点进行发送请求，如上步骤2。并接受响应。随后再转发给客户端。
3. 网关定期拉或者由服务端推的方式进行获取当前后端节点的负载状态，例如服务健康状态、连接数、CPU负载等，为gateway智能的负载均衡算法提供数据支持。
4. 网关除了代理流量分发之外一般有身份认证鉴权、限流等服务治理能力。

该方式的特点：

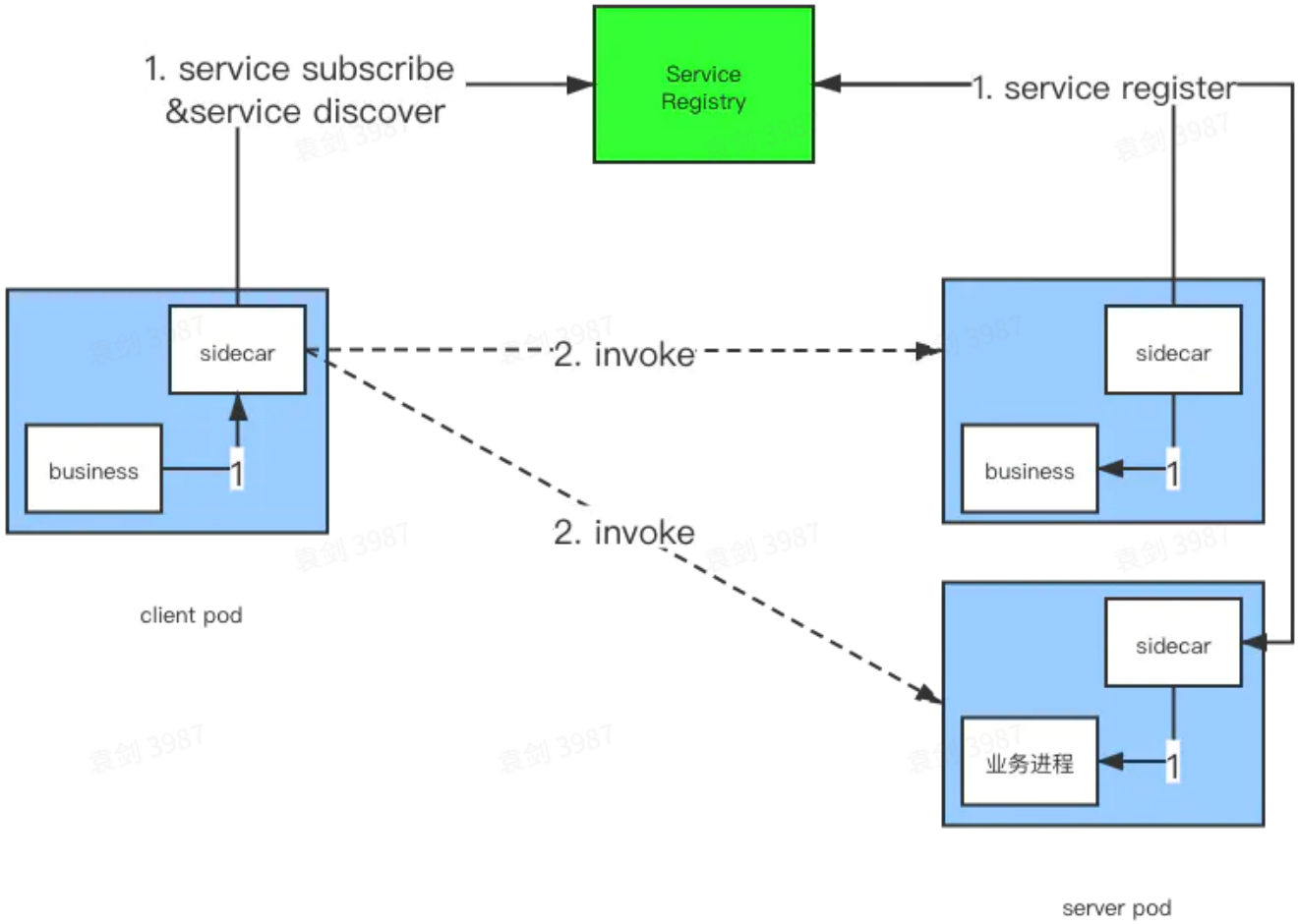
1. gateway的方式一般可以实现更智能的负载均衡算法，例如根据后端服务的系统负载进行优先选择。
2. 降低开发成本和业务代码耦合，客户端完全不需要处理服务发现、负载均衡等逻辑，一些复杂的认证健全、限流机制。

缺点：

a. 增加了额外的一跳，即需要经过独立的Load Balancer代理，增加了网络延迟。
b. 需要保证gateway的可用性，服务网关的可用性会直接影响服务的质量。

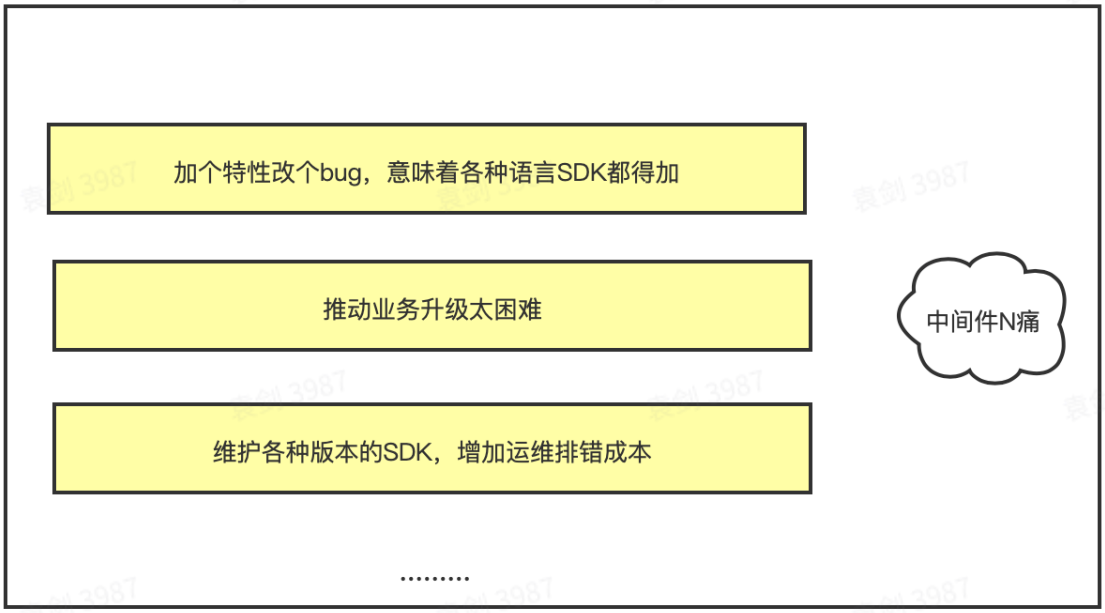
c. 在大流量情况下，需要支持其性能要求。

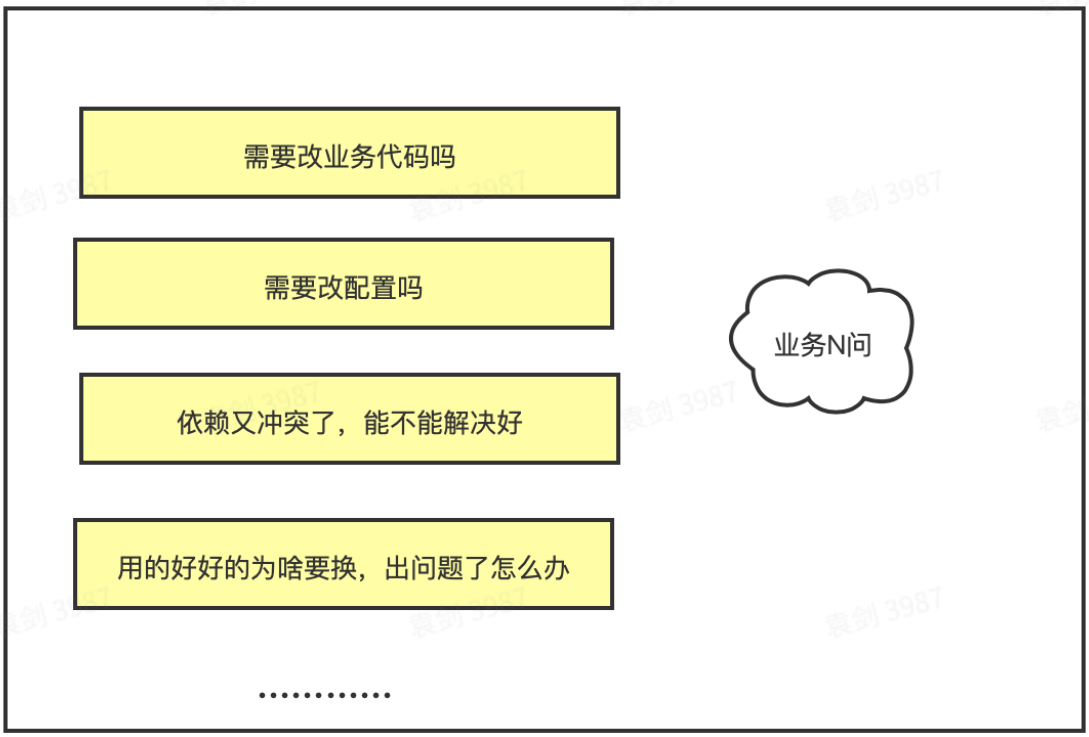
sidecar方式



该方式很类似于客户端负载均衡的架构，可以认为该方式是由上面两种架构衍生而来的。其为每个业务进程绑定一个sidecar进程，出入该业务进程的流量都需要经过该sidecar，并且可以将负载均衡以及负载均衡的依赖服务如服务注册发现、服务跟踪等一系列与业务逻辑无关的代码抽象分离到sidecar这个程序中。

传统微服务架构的痛点





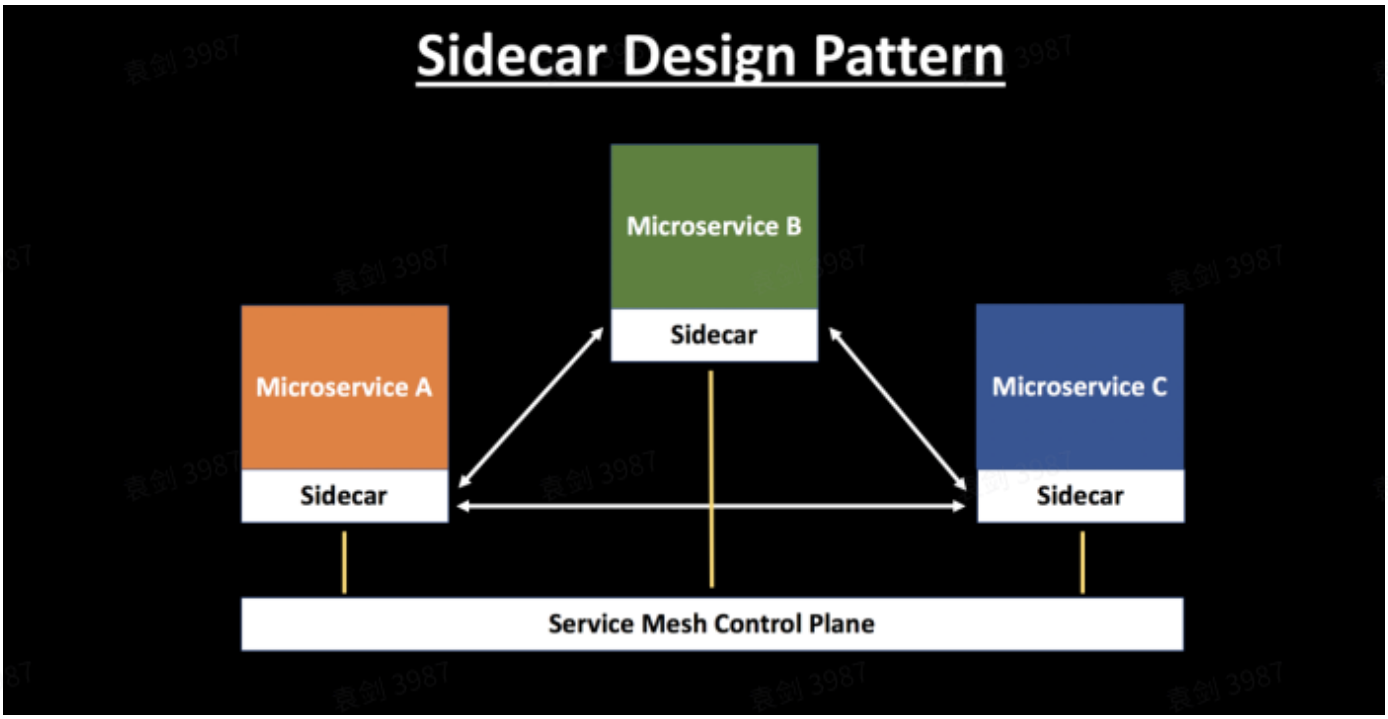
总结：

1. 上层开发需要关注更多服务治理层的技术问题，无法聚焦在业务本身。
2. 升级成本高，中间件开发推动升级难，大规模服务治理困难。线上不同服务引用的 SDK 版本不统一、能力参差不齐，造成很难统一治理。
3. 多语言支持困难，任何一个小改动都需要同步到多语言SDK。

什么是ServiceMesh

ServiceMesh，译为服务网格，是一个基础设施层，其功能在于处理服务间通信，职责是负责实现流量的可靠传递、服务治理以及整合异构微服务体系。在实践中，服务网格通常实现方式为通过轻量级网络代理与应用程序部署在一起，其对应用程序透明，网络代理在业务之上沉淀许多关键功能，包括服务发现，负载平衡，加密，可观察性，可追溯性，身份验证和授权，以及对断路器模式的支持。其能帮助我们更规模化、标准化更地实施、运维整个微服务体系。

边车模式与上面的网关有类似之处，但是粒度更细，很多人也称为分布式的网关。



其为每个服务都配备一个“边车”，这个“边车“可以理解为一个 proxy ，负责对应服务的所有进出流量，在这个 proxy中，像服务注册、服务发现、监控、流量控制、日志记录、服务限流和服务熔断、身份验证授权等功能完全可以做成标准化的组件和模块，不需要在单独实现其功能来消耗业务开发的精力和时间来开发和调试这些功能，从而把系统控制层和业务层分割开，让开发人员更专注于业务开发，无需关注底层实现。

这种架构其实很早就已经有人提出，但真正大规模践行是在最近几年。其根本原因是分布式架构、微服务、服务编排、容器化等新型技术被互联网公司逐渐接受，服务编排和治理运维的便捷性和效率大幅度提升。

ServiceMesh主要组成为数据面和控制面，数据面是边车，控制面是中央大脑，为所有边车分发数据和规则，使得边车真正地Run起来，下面会具体介绍。

数据面

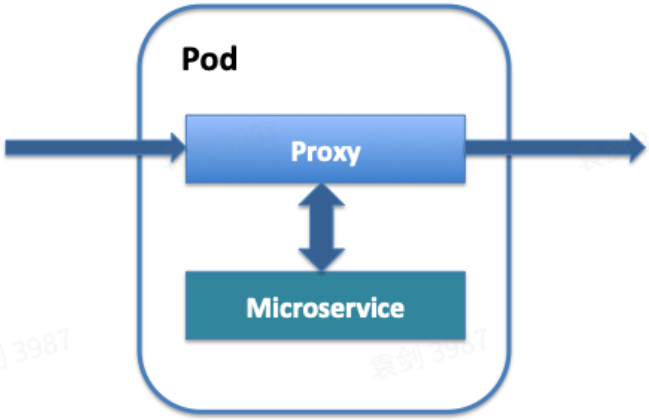
下文proxy、sidecar都可以理解为数据面。

数据面是流量的进出口，通常指的是我们上面说的sidecar部分，其一般独立进程部署，提供原先SDK中集成的核心能力：

1. 服务发现：有哪些可用的上游/后端服务实例
2. 健康检查（oob ib）：主动发现，例如通过一些轻请求主动请求后端示例的/health接口，或者是通过响应码来做一些检查。https://docs.google.com/document/d/1NSnK3346BkBo1JUU3I9I5NYYnaJZQPt8_Z_XCBCI3uA/edit#
3. 路由决策：通过路径匹配，Cookie匹配等一些请求信息进行决定该请求应该发往哪个集群。
4. 负载均衡：在路由过程中选择了上游服务集群后，应将请求发送到哪个上游服务实例？超时策略？用什么断路设置？如果请求失败，应该重试吗？
5. 身份认证鉴权：对请求方的身份识别和调用鉴权。
6. 可观测性：对每个请求生成详细的统计信息，日志记录和分布式调用链数据，并且定时上报。
7. 故障注入：向系统中刻意引入故障，并观察系统中存在故障时的行为。
8. 流量镜像：流量镜像，也称为影子流量。流量镜像转发以达到在不影响线上服务的情况下对流量或请求内容做具体分析的目的，它的设计思想是只做转发而不接收响应（fire and forget）。并会通过某些手段区分影子流量和原始流量的区别。例如在原始流量请求标头中的 `host` 属性值拼接上 `“-shadow”` 字样作为镜像流量的 `host` 请求标头。
9. 可自由扩展（WebAssembly）
10. ……

部署形式：

在K8S环境下，因为K8S抽象了Pod这个最小的调度单位。较为常见的部署方式是把数据面容器和业务容器部署在同一个pod，行成一个贴合的线程组，同一个Pod内共享网络、数据卷等。



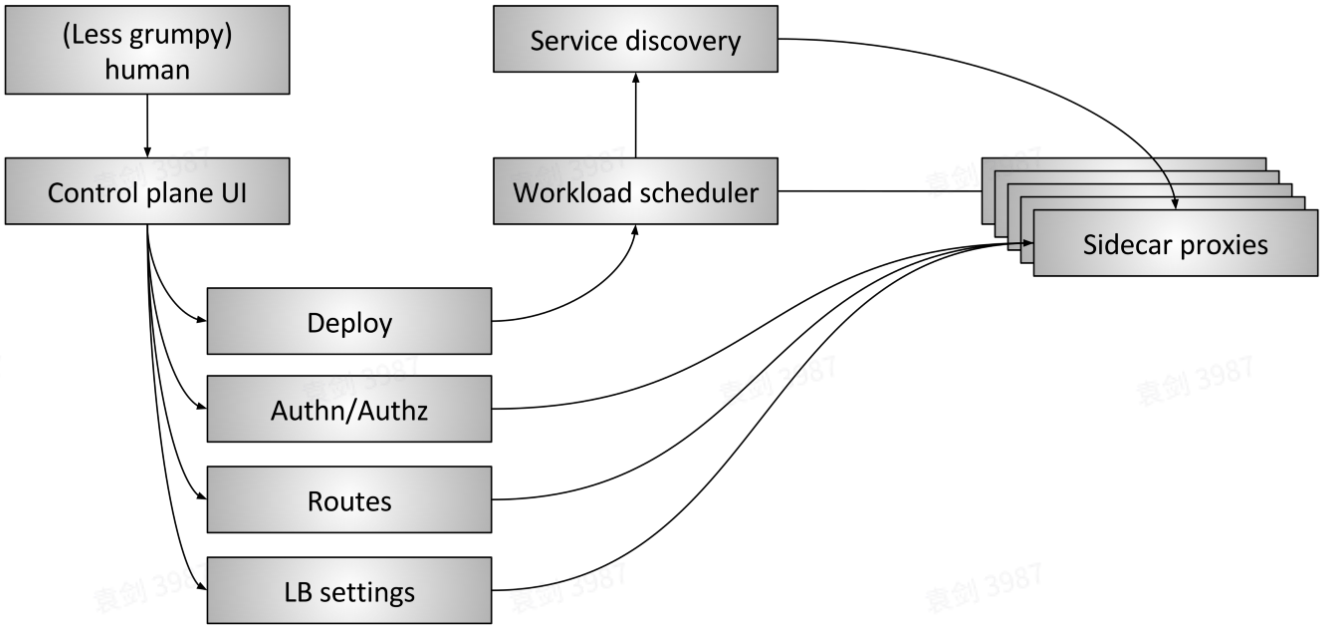
共享网络意味着Pod内的容器可以使用localhost互相通信，甚至可以通过IPC（InterProcess Communication）机制，如Unix Domain Socket进行大幅度降低网络通信的性能损耗。

控制面

数据面相对比较容易理解，数据面本身并不知道当前实例的路由策略、鉴权规则、上游集群以及实例列表，这些信息都需要在控制面运行时下发。

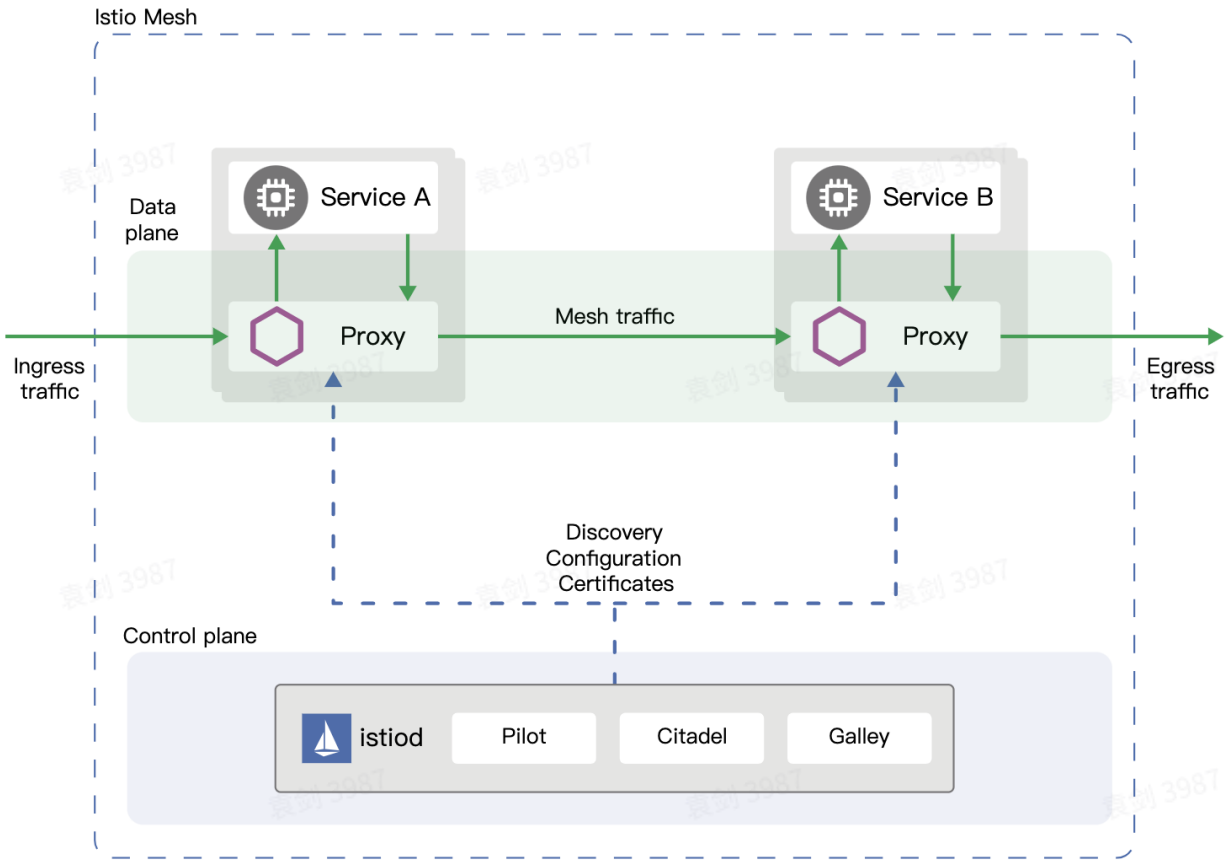
控制面的核心可以理解为一个巨大的配置规则服务器，其负责收集并聚合服务部署信息、元数据信息以、服务治理规则、安全证书等信息。并将这些配置按照规则推送给envoy，从而控制代理的行为。数据面和控制面之间的通信是通过API定义的，这被称为Data Plane API。

除此之外，控制面还被希望提供一些高级的功能减少人工操作的成本:



1. 例如可视化的的UI平台，提供服务治理规则等基础配置功能。
2. 收集更多的元数据信息推送给数据面，以便数据面作出更多对服务有效的决策，例如路由规则、集群节点、服务治理信息的信息。
3. 数据面的自动部署、生命周期管理，流量的自动拦截。
4. 兼容不同平台的服务发现机制、聚合服务节点信息并向envoy提供服务注册信息。

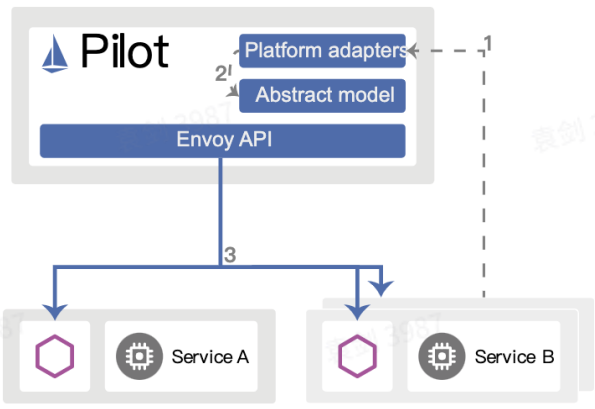
目前业界比较应用广泛的开源控制面实现是istio。其提供了对于服务网格的解决方案。使得接入服务网格对业务完全透明，业务无需修改代码。其底层主要依赖了iptables流量拦截以及通过集成各种平台adapter做到sidecar的自动注入以及对各种规则的监听、收集（例如在k8s平台上，借助k8s的webhook）。



Istio Architecture

其中主要部分：

1. Pilot



Service discovery

Pilot是Istio的核心部分。其为数据面提供服务发现、用于智能路由的流量管理功能（例如，A/B 测试、金丝雀发布等）以及弹性功能（超时、重试、熔断器等）。

Pilot 将控制流量行为的高级路由规则转换为特定于环境的配置，并在运行时将它们传播到对于的sidecar。Pilot将特定于平台的服务发现机制抽象出来，做到与部署平台无关，并将它们合成为任何符

合 Envoy Data Plane API 的 sidecar 都可以使用的标准格式。

2. Citadel

通过内置的身份和证书管理，可以支持强大的服务到服务以及最终用户的身份验证。包括TLS身份认证、基于HTTP、TCP的流量授权。

3. Galley

Galley 是 Istio的配置验证、提取、处理和分发组件。它负责将其余的 Istio 组件与从底层平台（例如 Kubernetes）获取服务配置的细节隔离开来，从而确保其他组件不必感知平台的差异。

xDS协议

数据面通过一套标准的流程和控制面进行交互，并获取所需资源的协议。这些资源不只是服务发现的数据、也可以是一些路由信息、熔断降级信息等服务治理相关的能力。这一套标准的流程被抽象为了xDS协议，该协议早期版本通过RESTful进行通讯传输，因为性能考虑和RESTful较为简单的request-response语义，目前更多地是采用gRPC双向流+ProtocolBuffers进行通讯传输。

xDS协议：

1. RDS：路由发现服务
2. LDS：Listener发现服务
3. CDS：集群发现服务
4. EDS：节点发现服务
5. SDS：安全发现服务
6. ADS：聚合发现服务
7. ...

MCP协议

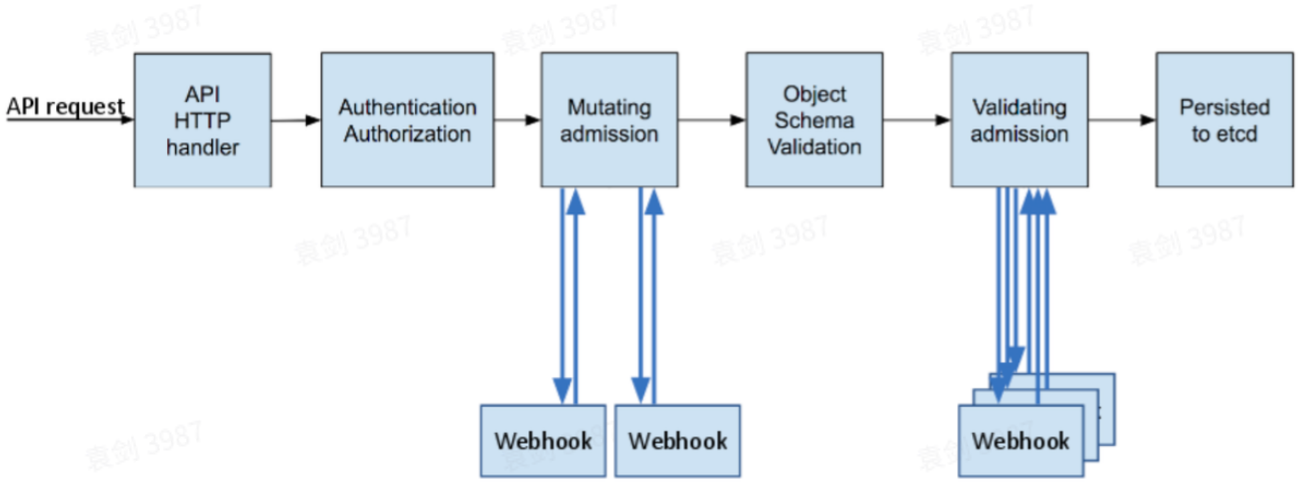
MCP协议设计灵感也来自于xDS，其在控制层通过定义一套标准的通讯协议将核心逻辑与配置获取解偶，也提供了另一种配置获取的方式。外部系统可以通过对接MCP的方式，将配置信息转化为Pilot认识的格式，交给Pilot分发，该方式能标准化地扩展服务网格体系，使得原有架构和服务网格架构中的服务信息双方可见，从而进行无缝互通。

istio最大的特点之一就是透明，对上层无感知，用户无需改动任何代码既可以享受ServiceMesh带来的便携性。

主要包括如下两点：

Sidecar透明注入

istio使用K8S的 `MutatingWebhook` 机制在 pod 创建的时候将 sidecar 的配置信息添加到每个 pod 的模版里然后持久化到etcd，然后被Scheduler进行调度执行。用户可以通过 webhooks `namespaceSelector` 机制来限定需要启动自动注入的范围，也可以通过注解的方式针对每个 pod 来单独启用和禁用自动注入功能。



具体代码可以看 `pkg/kube/inject/webhook.go:766`

```
Init Containers:
  istio-init: // 初始化容器
    Container ID:
docker://d7196ccd78b15d1fcb1c7d1c1636844197a2203a25deb778cd92e69e43e4e0d9
    Image:         docker.io/istio/proxyv2:1.7.0
    Image ID:      docker-
pullable://istio/proxyv2@sha256:c1f1b45a4162509f86aa82d0148aef55824454e7204f27f23d
ddc9d7f4ae7cd1

Containers:
  busybox:
    Container ID:
docker://401fe52f4788fad6e5022f042c39f8097e6b2c8cfe09c178d65154e199559418
    Image:         busybox
    Image ID:      docker-
pullable://busybox@sha256:c3dbcbbf6261c620d133312aee9e858b45e1b686efbcead7b34d9aae
58a37378
```

```
12
13   istio-proxy: // proxy容器
14     Container ID:
   docker://80a8f26fa4911cb1b62704ef2e0d904bf6daa35b9073ee7113de143cf404eb8b
15     Image:         docker.io/istio/proxyv2:1.7.0
```

透明流量劫持

istio默认的透明流量劫持是通过iptables进行拦截进出口流量实现的。

`iptables` 是 Linux 内核中的防火墙软件 netfilter 的管理工具，位于用户空间，同时也是 netfilter 的一部分。Netfilter 位于内核空间，不仅有网络地址转换的功能，也具备数据包内容修改、以及数据包过滤等防火墙功能。

istio通过在部署的时候除了对业务Pod进行自动注入sidecar容器之外，还会在Pod中注入一个istio-init容器。如下所示：

```
1  kd pod reviews-v1-7f99cc4496-5pl7b -n istio-demo
2 Name:         reviews-v1-7f99cc4496-5pl7b
3 Namespace:    istio-demo
4 Priority:     0
5 ...
6 Init Containers:
7   istio-init:
8     Container ID:
   docker://0aa880abb8569fa7ed1fb8e4a1a682ea7e2b12ad6133c392c0bd57c403c9fadb
9     Image:         docker.io/istio/proxyv2:1.7.0
10    Image ID:      docker-
   pullable://istio/proxyv2@sha256:c1f1b45a4162509f86aa82d0148aef55824454e7204f27f23d
   ddc9d7f4ae7cd1
11    Port:          <none>
12    Host Port:     <none>
13    Args:
14      istio-iptables
15      -p
16      15001
17      -z
18      15006
19      -u
20      1337
21      -m
22      REDIRECT
23      -i
24      *
```

```
      -x

      -b
      *
      -d
      15090,15021,15020
    State:          Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Fri, 11 Sep 2020 14:51:59 +0800
      Finished:     Fri, 11 Sep 2020 14:51:59 +0800
    Ready:          True
    Restart Count:  2
    Limits:
      cpu:     2
      memory:  1Gi
    Requests:
      cpu:     10m
      memory:  10Mi
    Environment:
      DNS_AGENT:
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from bookinfo-reviews-token-w2vgx (ro)
Containers:
```

借助k8s的initContainer的特性保证在业务容器和sidecar容器启动之后能正常进行流量的拦截。

istio-init的主要作用就是设置iptables，如下所示：

```
# iptables -t nat -L -v
Chain PREROUTING (policy ACCEPT 1675 packets, 101K bytes)
 pkts bytes target     prot opt in     out     source               destination
 1675  101K ISTIO_INBOUND  tcp  --  any    any     anywhere             anywhere

Chain INPUT (policy ACCEPT 1675 packets, 101K bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 810 packets, 76106 bytes)
 pkts bytes target     prot opt in     out     source               destination
    9   540 ISTIO_OUTPUT  tcp  --  any    any     anywhere             anywhere

Chain POSTROUTING (policy ACCEPT 810 packets, 76106 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain ISTIO_INBOUND (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     tcp  --  any    any     anywhere             anywhere             tcp dpt:15008
    0     0 RETURN     tcp  --  any    any     anywhere             anywhere             tcp dpt:ssh
    0     0 RETURN     tcp  --  any    any     anywhere             anywhere             tcp dpt:15090
 1675  101K RETURN     tcp  --  any    any     anywhere             anywhere             tcp dpt:15021
    0     0 RETURN     tcp  --  any    any     anywhere             anywhere             tcp dpt:15020
    0     0 ISTIO_IN_REDIRECT  tcp  --  any    any     anywhere             anywhere

Chain ISTIO_IN_REDIRECT (3 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 REDIRECT   tcp  --  any    any     anywhere             anywhere             redir ports 15006

Chain ISTIO_OUTPUT (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  any    lo      127.0.0.6            anywhere
    0     0 ISTIO_IN_REDIRECT  all  --  any    lo      anywhere            !localhost           owner UID match 1337
    0     0 RETURN     all  --  any    lo      anywhere             anywhere             ! owner UID match 1337
    9   540 RETURN     all  --  any    any     anywhere             anywhere             owner UID match 1337
    0     0 ISTIO_IN_REDIRECT  all  --  any    lo      anywhere            !localhost           owner GID match 1337
    0     0 RETURN     all  --  any    lo      anywhere             anywhere             ! owner GID match 1337
    0     0 RETURN     all  --  any    any     anywhere             anywhere             owner GID match 1337
    0     0 RETURN     all  --  any    any     anywhere             localhost
    0     0 ISTIO_REDIRECT  all  --  any    any     anywhere             anywhere

Chain ISTIO_REDIRECT (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 REDIRECT   tcp  --  any    any     anywhere             anywhere             redir ports 15001
```

1. PREROUTING 链：用于目标地址转换（DNAT），将所有入站 TCP 流量跳转到 ISTIO_INBOUND 链上。

2. OUTPUT 链：将所有出站数据包跳转到 ISTIO_OUTPUT 链上。

3. ISTIO_INBOUND 链：将所有入站流量重定向到 ISTIO_IN_REDIRECT 链上，排除了ssh、15008、15090、15021、15020端口的流量。

4. ISTIO_OUTPUT 链：目的地非localhost并且来自于1337用户空间的流量都转发到ISTIO_IN_REDIRECT链

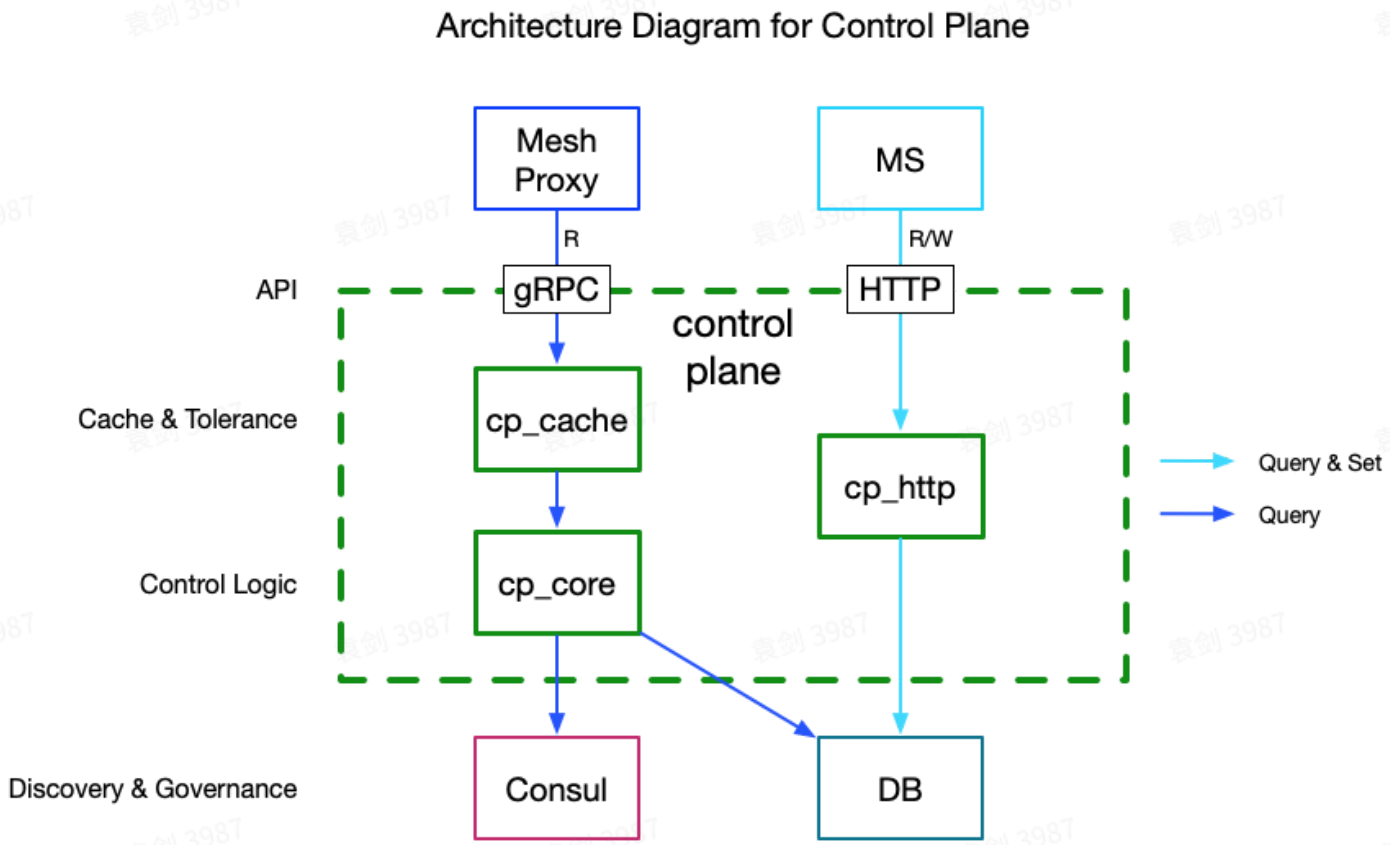
5. ISTIO_IN_REDIRECT：所有流量转发到15006端口。

6. ISTIO_REDIRECT 链：将所有流量重定向到 Sidecar（即本地） 的 15001 端口。

最终的效果是所有入站（inbound）流量重定向到 15006 端口（sidecar），再拦截应用容器的出站（outbound）流量经过 sidecar 处理（通过 15001 端口监听）后再出站。

头条Mesh组件架构

控制面



cp_core: 服务发现、聚合服务治理逻辑、推送给数据面的核心逻辑

cp_cache: 缓存模块，提高查询性能

cp_http: MS平台后台

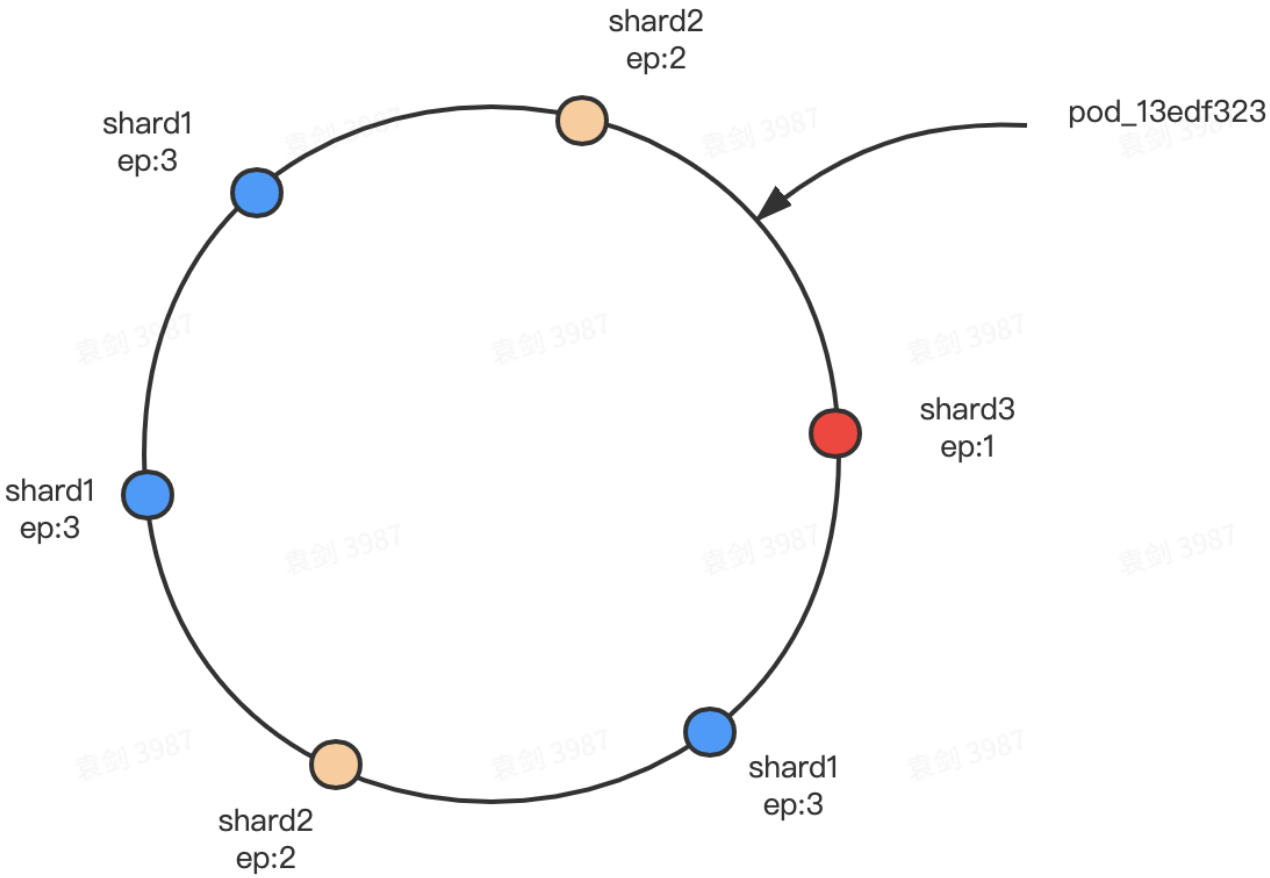
consul: 服务注册中心

DB: 存储一些服务治理规则，审计、操作信息等

内部在控制面上做的优化：

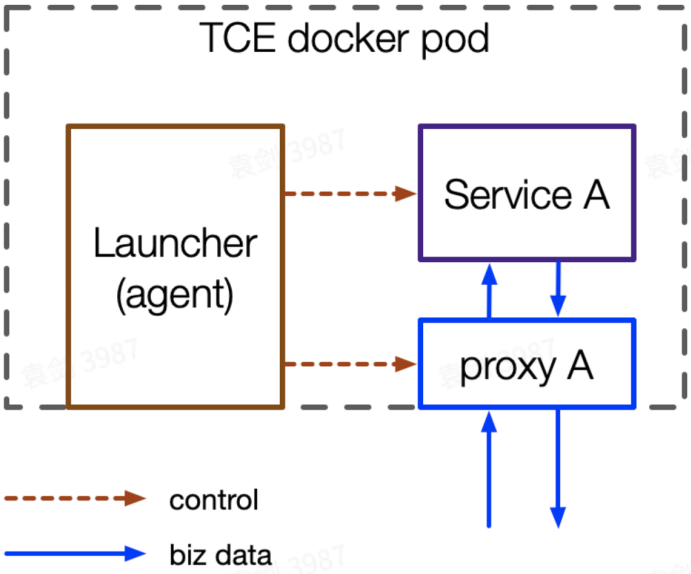
1. 缓存，并且无变化时告知proxy。类似HTTP304
2. 对于上下游实例数比较大的，不全量推送依赖服务的所有节点信息，该方式不仅有利于减少网络带宽和减少CPU计算，也能发挥长链接的优势。
3. 下游服务实例发生变动时，尽可能减少影响范围。

主要实现思路是通过将下游服务节点映射到哈希环的分片上，并且根据上游示例名的哈希取到对应的分片。只将该分片上的下游服务节点分配给映射在该分片区域的上游请求节点，通过该方式，即便下游服务扩容或者缩容也只会针对部分上游实例的进行重新推送信息。并且只会推送某一个分片的下游服务节点信息，另一方面也减少上游实例维护的长链接，发挥长链接RPC的优势。



数据面

头条数据面在TCE的部署方式如下图所示：



Service Mesh TCE部署方案

头条目前并没有采用“一个Pod多个容器”部署，而是才用富容器（一个容器多个进程）的部署方式。Latcher(agent)是父进程，负责并管理业务进程和proxy进程的生命周期（加载、启动、平滑关机、平滑迁移等）。 Mesh proxy 热升级 连接迁移方案

头条数据面流量

头条内部的数据面流转并没有基于iptables的透明劫持方案，而是采用了直接转发的方式。主要原因是

- iptables 在规则配置较多时，性能下滑严重。
- 管控性、可观测性不好，出了问题比较难排查。

直接转发的方式相对比较容易理解，缺点就是对代码有一点侵入性的直接转发方式。需要在服务框架层做相对应的改造。

以kitex为例，我们解释下。

当我们在MS平台上部署服务时，如果我们在服务的高级配置上打开了Service Mesh开关

TCE的调度系统会在创建对应的容器时，注入Mesh相关的环境变量，我们通过`cat /proc/${pid}/envion`查看。以下图服务为例：

```
dp-7d7b4cfe9f-5897f9677c-2fck6(webcast.scheduler.base@default:canary):base# cat /proc/144031/environ | tr '\0' '\n' | grep MESH
SERVICE_MESH_HTTP_EGRESS_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/http.egress.sock
MESH_CP_CLUSTERS=10.129.16.227:6022,10.129.21.212:6022,10.129.22.161:6022,10.129.43.146:6022
SERVICE_MESH_INGRESS_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/rpc.ingress.sock
SERVICE_MESH_HEALTHZ_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/healthz.sock
SERVICE_MESH_ADMIN_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/admin.sock
MESH_ORIGIN_COMMAND=/opt/tiger/toutiao/load/../load/load.sh /opt/tiger/webcast/scheduler/base
WITH_RPC_MESH_INGRESS=1
SERVICE_MESH_HTTP_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/http.sock
MESH_INGRESS_PORT=8888
SERVICE_MESH_EGRESS_ADDR=/opt/tiger/toutiao/var/service/webcast.scheduler.base.mesh/rpc.egress.sock
SERVICE_MESH_MODE=True
WITH_RPC_MESH_EGRESS=1
```

其中比较关键的是SERVICE_MESH_INGRESS_ADDR，表示数据面接受入口流量时转发的流量，SERVICE_MESH_EGRESS_ADDR表示出口流量转发到地址即数据面监听的地址、MESH_CP_CLUSTERS表示控制面集群地址，并且这里都是本地的转发都用Unix Domain Socket代替TCP减少了网络传输的性能损耗。

出口流量：

通过判断是否打开了Mesh

```
else if mesh.EgressEnabled() {  // mesh开关打开了，
   remote = config.DefaultClientMeshRemoteConfiger()
   opts = append(opts, client.WithMiddlewareBuilder(emptyMWBToAddCache4RemoteConfiger(remote)))
   opts = append(opts, client.WithMiddleware(config.NewRPCConfigUpdateMW(remote)))
   // meshMoreTimeout try to wait timeout resp from mesh but not happen in client,
   // or it will loss remote info in mesh case
   meshMoreTimeout := 2 * time.Millisecond
   opts = append(opts, client.WithMiddlewareBuilder(rpctimeout.MiddlewareBuilder(meshMoreTimeout)))
   opts = append(opts, internal_client.WithTTHeader())
   opts = append(opts, internal_client.WithProxy(mesh.NewEgressProxy())) // 注入proxy
}
```

初始化中间件

```
func (kc *kClient) initMiddlewares(ctx context.Context) {
   kc.mws = richMWsWithBuilder(ctx, kc.opt.MWBs, kc)
   // add new middlewares
   kc.mws = append(kc.mws, acl.NewACLMiddleware(kc.opt.ACLRules))
   if kc.opt.Proxy == nil {
      kc.mws = append(kc.mws, newResolveMW(kc.opt.Resolver, kc.opt.Balancer, kc.opt.Bus)) // 服务发现、负载均衡MW
      kc.mws = richMWsWithBuilder(ctx, kc.opt.IMWBs, kc)
   } else {
      kc.mws = append(kc.mws, newProxyMW(kc.opt.Proxy)) // proxyMW
```

```
    }
    kc.mws = append(kc.mws, newIOErrorHandleMW(kc.opt.ErrHandle))

```

ProxyMW中间件做的事就是解析数据面proxy的地址，并且注入到当前请求的上下文中，从而使请求发往该proxy。

```
func newProxyMW(prx proxy.ForwardProxy) endpoint.Middleware {
   return func(next endpoint.Endpoint) endpoint.Endpoint {
      return func(ctx context.Context, request, response interface{}) error {
         // 根据环境变量获取对应的egress出口地址（即envoy监听的地址），将其作为本次调用的instance。从而做到转发到envoy
         err := prx.ResolveProxyInstance(ctx)
         if err != nil {
            return err
         }
         err = next(ctx, request, response)
         return err
      }
   }
}
```

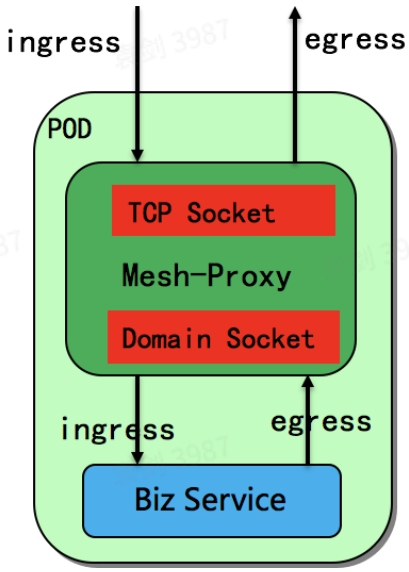
入口流量通过替换业务容器server监听的地址。来接受envoy转发的入口流量。

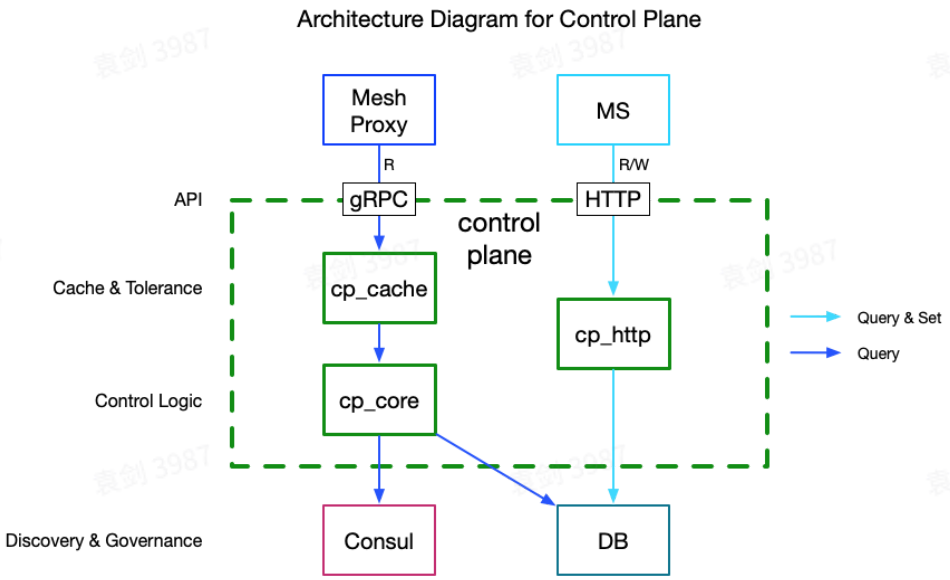
```
func (s *server) Run(opts ...RunOption) (err error) {
   if s.svcInfo == nil {
      return errors.New("Run: no service. Use RegisterService to set one")
   }
   if err = s.check(); err != nil {
      return err
   }
   svrCfg := s.opt.RemoteOpt
   addr := svrCfg.Address // should not be nil
   if s.opt.Proxy != nil { // proxy不为空，开启了mesh，则将原始注册地址改为unix domain socket地址来代替tcp，降低与envoy之间的网络传输性能损耗。
      svrCfg.Address, err = s.opt.Proxy.Replace(addr)
      if err != nil {
         return
      }
   }

   s.richRemoteOption()
   transHdlr, err := s.newSvrTransHandler()
```

```
19      if err != nil {
20          return err
21      }
22      s.Lock()
23      s.svr, err = remotesvr.NewServer(s.opt.RemoteOpt, s.eps, transHdlr)
24      s.Unlock()
25      if err != nil {
26          return err
27      }
28      err = s.svr.Start()
29      if err != nil {
30          s.opt.Logger.Errorf("Kitex Run: %s", err)
31          return err
32      }
33      return s.Stop()
34  }
```

头条内部的Mesh-Proxy流量流转示意图，本地流量通过Unix Domain Socket，外部流量使用TCP Socket。





以出口流量egress为例

1. 服务启动，proxy进程一起启动，此时proxy中没有任何规则，也不含任何服务发现信息

2. 服务准备发送第一个请求，它向 proxy 发送第一个数据包，声明：本服务是A，要访问服务B，访问request如下。
3. proxy 在本地查找 A->B 是否有缓存，如果有，直接使用，如果没有，则proxy 向 cp_cache 询问 A->B 的各种规则
4. cp_cache 查找一下 A->B 是否有缓存，如果有，直接返回给 proxy， 如果没有，则向 cp_core 询问 A->B 的规则
5. cp_core 接受 cp_cache 的请求，使用 consul 和 DB 组装一系列的规则，返回给 cp_cache。
6. proxy拿到规则，执行规则，通过指定的负载均衡算法，然后选中其中一个节点，建立链接）发送出去。
7. proxy拿到响应然后发送将响应返回给业务服务进程。

其他，随着微服务架构的Mesh化持续推进，其他一些基础组件都在准备搭上Mesh这辆“特快列车”，享受Mesh带来的红利。

1. Redis Mesh
2. Mysql Mesh
3. MQ Mesh
4. ...

ServiceMesh缺点：

1. 多了一个故障点，控制面和数据面。
2. 代理请求转发的方式一定程度会降低系统通信性能。
3. 像业务屏蔽了细节的同时，也需要更专业的排错运维能力。