## Pilot

Pilot是Istio控制面流量管理的核心组件。

### 架构

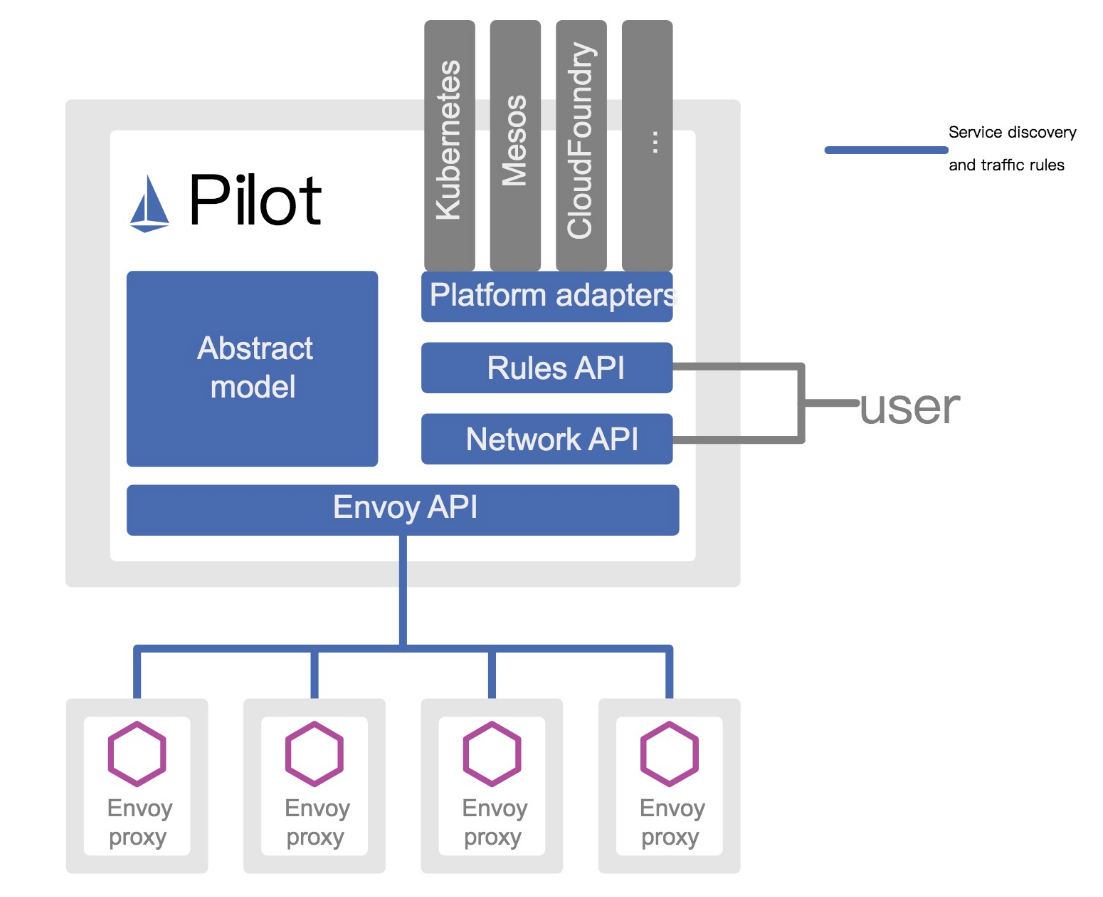


Figure :

*Pilot架构图*

* 平台适配器(Platform Adapter)：监听底层平台，并完成从平台特有的服务模型到Istio规范模型的转换。
  + 服务模型的转换：将K8s、Consul等平台的服务模型 —> Istio规范的服务模型
  + 实例模型转换：将K8s Endpoint资源 —> Istio规范的服务实例模型
  + Istio中的配置模型的转换：将非结构化的Custom Resource配置规则 —> VirtualService、Gateway、ServiceEntry、DestinationRule等API， K8s Ingress —> Istio Gateway
* 抽象聚合层，Abstract Model，聚合不同平台的服务、配置规则对外提供统一的接口，屏蔽底层的差异。
* Envoy API(也称为xDS API)，直接将Pilot流量治理能力暴露给客户端，将服务信息和流量规则下发到数据面的 [sidecar](https://www.servicemesher.com/istio-handbook/GLOSSARY.html#sidecar) 中。

Pilot和Envoy代理之间通过gRPC来维护一条gRPC长连接，所有配置的分发都基于此连接的一个Stream。

### 代码架构

主要分为两部分：pilot-agent和pilot-discovery

### Istio服务模型

*版本istio-1.3*

Service

type Service struct {  
 // Name of the service, e.g. "catalog.mystore.com"  
 Hostname host.Name `json:"hostname"`  
  
 // Address specifies the service IPv4 address of the load balancer  
 Address string `json:"address,omitempty"`  
  
 // Protect concurrent ClusterVIPs read/write  
 Mutex sync.RWMutex  
 // ClusterVIPs specifies the service address of the load balancer  
 // in each of the clusters where the service resides  
 ClusterVIPs map[string]string `json:"cluster-vips,omitempty"`  
 // Ports is the set of network ports where the service is listening for  
 // connections  
 Ports PortList `json:"ports,omitempty"`  
  
 // ServiceAccounts specifies the service accounts that run the service.  
 ServiceAccounts []string `json:"serviceAccounts,omitempty"`  
  
 // MeshExternal (if true) indicates that the service is external to the mesh.  
 // These services are defined using Istio's ServiceEntry spec.  
 MeshExternal bool  
  
 // 在进行路由前解析服务实例的地址  
 Resolution Resolution  
  
 // CreationTime records the time this service was created, if available.  
 CreationTime time.Time `json:"creationTime,omitempty"`  
  
 // Attributes contains additional attributes associated with the service  
 // used mostly by mixer and RBAC for policy enforcement purposes.  
 Attributes ServiceAttributes  
}

ServiceInstance

type ServiceInstance struct {  
 // 服务实例的网络地址、位置信息、负载均衡权重以及网络ID  
 Endpoint NetworkEndpoint `json:"endpoint,omitempty"`  
 // 关联的服务  
 Service \*Service `json:"service,omitempty"`  
 // 服务实例的标签，可以用于路由选择  
 Labels labels.Instance `json:"labels,omitempty"`  
 // 服务实例的身份信息  
 ServiceAccount string `json:"serviceaccount,omitempty"`  
}

NetworkEndpoint

type NetworkEndpoint struct {  
 // Family indicates what type of endpoint, such as TCP or Unix Domain Socket.  
 Family AddressFamily  
  
 // Address of the network endpoint. If Family is `AddressFamilyTCP`, it is  
 // typically an IPv4 address. If Family is `AddressFamilyUnix`, it is the  
 // path to the domain socket.  
 Address string // 对应于K8s当中Pod的IP地址  
  
 // Port number where this instance is listening for connections This  
 // need not be the same as the port where the service is accessed.  
 // e.g., catalog.mystore.com:8080 -> 172.16.0.1:55446  
 // Ignored for `AddressFamilyUnix`.  
 Port int // 服务进程监听的端口号  
  
 // Port declaration from the service declaration This is the port for  
 // the service associated with this instance (e.g.,  
 // catalog.mystore.com)  
 ServicePort \*Port // 服务端口号，等同于Service的虚拟IP，或负载均衡IP对应服务端口  
  
 // Defines a platform-specific workload instance identifier (optional).  
 UID string   
  
 // The network where this endpoint is present  
 Network string // 网络标识  
  
 // The locality where the endpoint is present. / separated string  
 Locality string // 用于基于位置的负载均衡策略  
  
 // The load balancing weight associated with this endpoint.  
 LbWeight uint32 // 负载均衡权重，主要用于服务网格外部服务(ServiceEntry定义)  
}

### xDS协议

xDS是一类发现服务的总称，包含：

* LDS：Listener发现服务
* RDS：Route发现服务
* CDS：Cluster发现服务
* EDS：Endpoint发现服务
* SDS：Secret发现服务

Envoy通过xDS API可以动态获取Listener（监听器）、Route（路由）、Cluster（集群）、Endpoint（集群成员）及Secret（证书）配置。

#### xDS API

完整的流程：请求、响应、ACK或者NACK

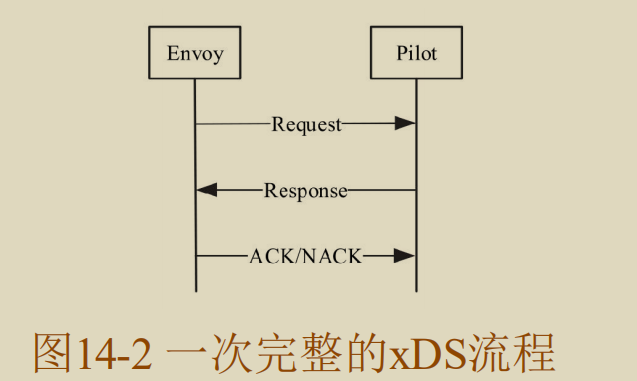


Figure :

#### ADS的演进

ADS Istio0.8版本之后的聚合发现服务。

基于gRPC协议的同一个流。

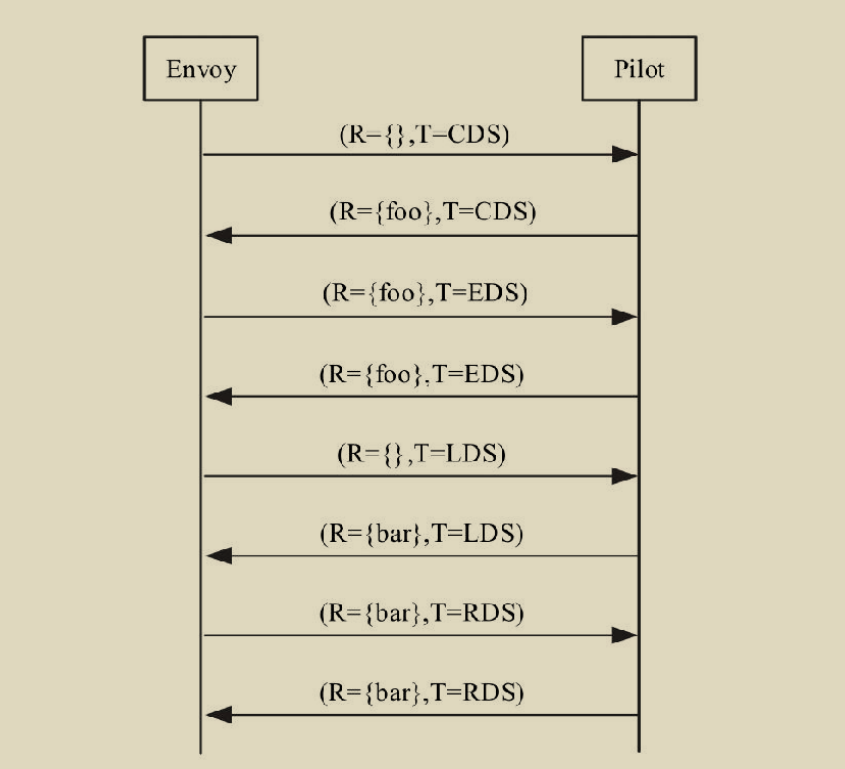


Figure :

*xDS基本的配置更新顺序*

Envoy在本质上采用最终一致性模型，不能保证不同资源的配置获取及加载时序：

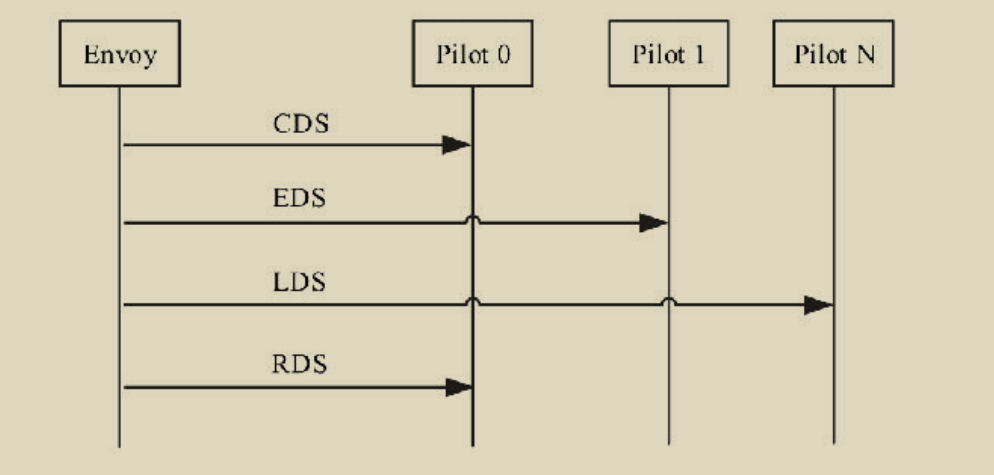


Figure :

*Envoy采用的最终一致性模型*

### Pilot的工作流程

Istio控制面的核心，主要职责是获取注册中心的配置规则或者服务，服务于所有的Envoy。

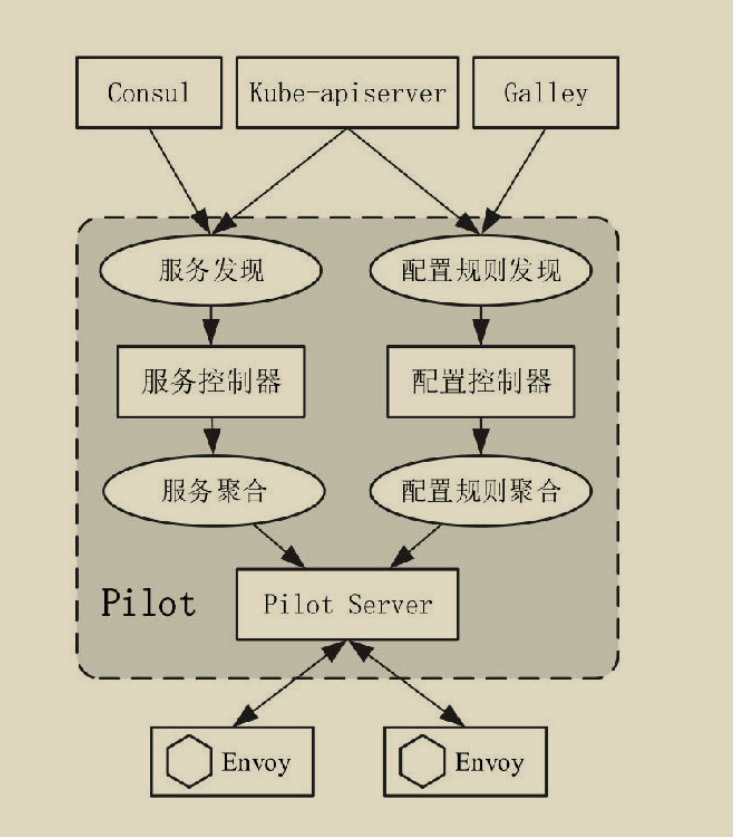


Figure :

*核心组件*

#### 启动和初始化

pilot-discovery进程启动：包含五个模块—Config Controller、Service Controller、xDS Server、HTTP Server、性能Monitor。

（1）xDS服务器用于处理Envoy代理的xDS请求，以及控制相关配

置的生成及下发；

（2）配置控制器主要用于监视底层注册中心及更新配置规则，并

通知 xDS 服务器异步更新xDS配置；

（3）服务控制器主要用于监视底层注册中心、更新服务及服务实

例，并通知 xDS 服务器异步更新xDS配置；

（4）HTTP服务器主要提供REST接口供管理员获取Debug信息；

（5）性能监视器主要提供性能分析的接口，可通过此接口获取进

程运行时内存、CPU占用等。

步骤：

1. 命令行参数解析
2. 初始化K8s客户端
3. 加载服务网格配置
4. 加载服务网格网络配置
5. 初始化Config Controller、Pilot核心模块
6. 初始化Service Controller
7. 初始化Pilot服务器
8. 初始化多集群服务发现
9. 启动所有Pilot Server及控制器

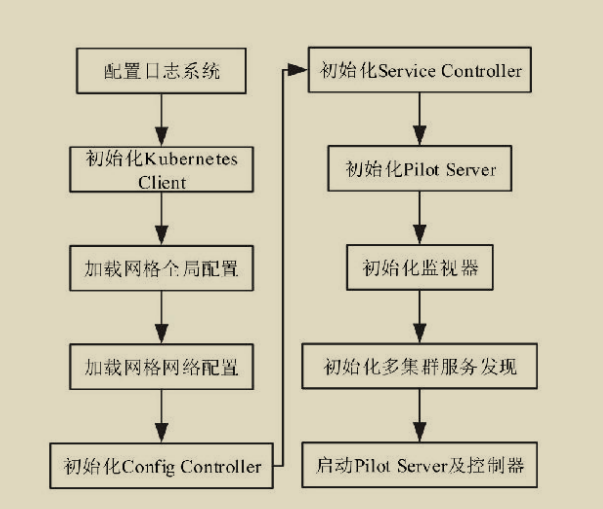


Figure :

#### Pilot Server初始化

是xDS的服务端，实现了各种发现服务功能。对外提供不同类型接口。

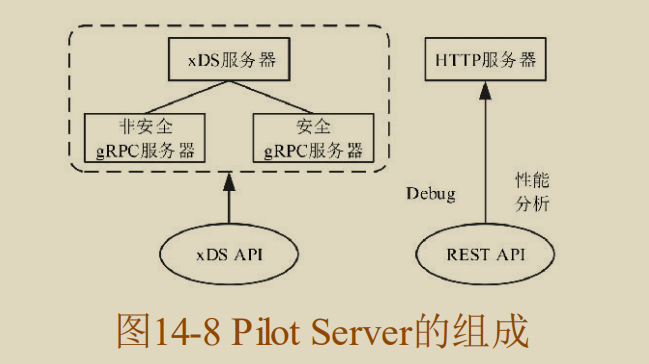


Figure :

流程：

1. 创建DiscoveryService，一个HTTP请求多路复用器，并注册分析处理器。
2. 创建EnvoyXdsServer，实现Envoy服务发现ADS接口，主要功能：维护和Envoy的连接、处理异步的事件通知、生成xDS配置等。
3. 向HTTP多路复用器注册Debug处理器
4. 初始化非安全gRPC服务器，注册xDS API处理函数
5. 初始化安全gRPC服务器，注册xDS API处理函数
6. 分别注册三种服务器的启动回调

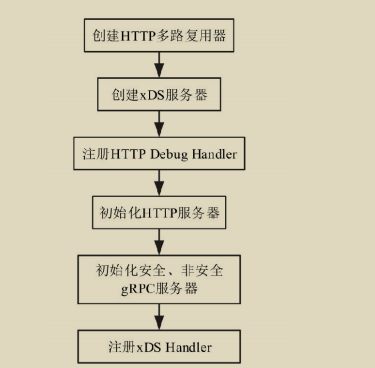


Figure :

#### 服务发现

指的是Istio服务、服务实例、服务端口、及服务身份信息的发现。

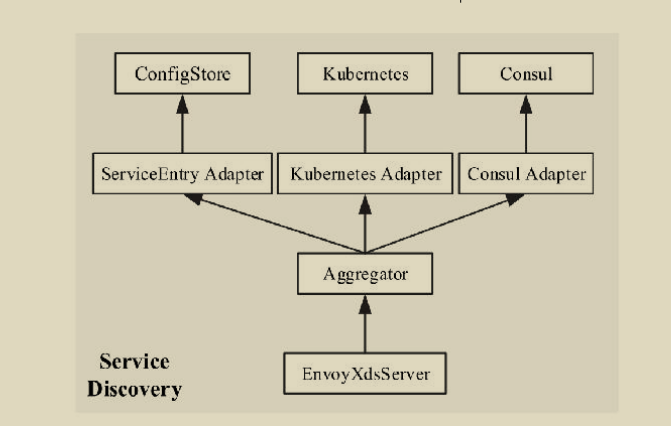


Figure :

*服务发现模型*

##### ServiceEntry

* 服务模型当中，除了网格之中的服务，还有网格外的服务，通过ServiceEntry来定义网格外的服务。  
  在Kubernetes环境下，ServiceEntry与其他配置规则一样是通过**CRD**定义的。

它分为两级服务发现：

1. Config Controller通过ConfigStore实现配置发现
2. Service Controller通过ServiceEntry适配器实现的服务发现。

##### 服务聚合

服务聚合器Aggregator是Pilot对所有Adapter的抽象封装

* 通过注册接口提供Adapter的注册
* 通过ServiceDiscovery接口实现的Service()查询所有服务信息。

⚡️注意：Aggregator只是汇聚Adapter的查询结果。

Istio中的所有Adapter都实现了ServiceDiscovery接口，以此来与Aggregator进行级联。

type ServiceDiscovery interface{  
 // 查询网格中所有的服务信息  
 Services() ([]\*Service, error)  
 // 根据hostname查询服务  
 GetService(hostname host.Name) \*Service  
   
 // 获取服务实例  
 InstanceByPort(svc \*Service, servicePort int, labels labels.Collection) []\*ServiceInstance  
 // 获取Sidecar代理相关的服务实例  
 GetProxyServiceInstances(\*Proxy) []\*ServiceInstance  
   
// --------------------------------------  
 GetProxyWorkloadLabels(\*Proxy) labels.Collection  
   
 GetIstioServiceAccounts(svc \*Service, ports []int) []string  
 NetworkGateways() []NetworkGateway  
 MCSServices() []MCSServiceInfo  
}

##### 异步通知机制

异步通知依赖于**回调函数**，系统发生更新事件时，系统会捕获到事件的同时调用回调函数。

通用的模型：

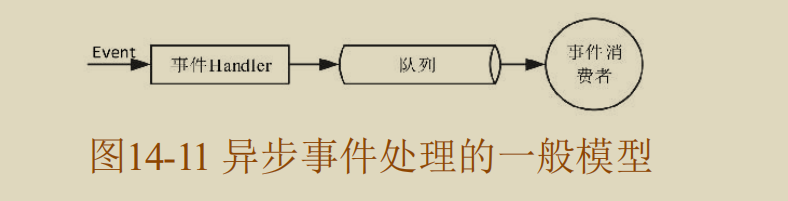


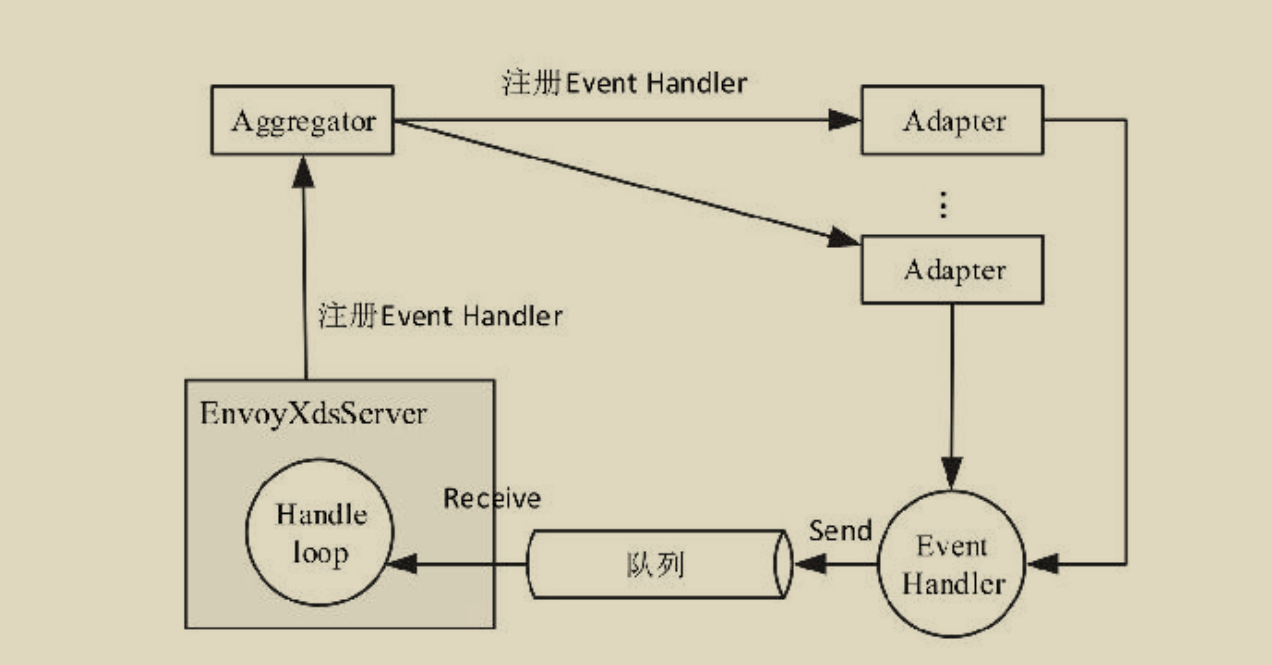
Figure :

Pilot也是基于此来实现的。

通过Controller接口，Aggregator给上层的EnvoyXdsServer提供**事件处理的注册方法**。

type Controller interface{  
// service目录更新事件提醒  
 AppendServiceHandler(f func(\*Service, Event))  
// 提醒对于workload的更改; 和直接对于Instance（Service和Instance的结合）的处理不同  
 AppendWorkloadHandler(f func(\*WorkloadInstance, Event))  
  
 Run(stop <-chan struct{})  
   
 HasSynced() bool  
}

实现原理：

  
 *异步通知机制*

分为上下部处理，上半部主要负责接收事件更改通知，下半部负责处理，分发配置。

EnvoyXdsServer初始化时通过Aggregator注册到每个Adapter当中，底层平台的Adpter基于底层注册中心提供的方式来监控资源的变化。

资源改变触发回调函数，即EnvoyXdsServer当中的clearCache，将事件更新通知转发到队列中。

##### Kubernetes Adapter

在K8s平台上，服务发现依赖于**Kubernetes Informer**机制：

* Service对象表示服务
* Endpoint表示服务实例
* 通过Pod、Node获取Envoy代理的标签以及可用域等信息。

Kubernetes Adapter通过SharedInformerFactory创建四种类型的SharedInformer，通过AddEventHandler接口注册ResourceEventHandler资源事件处理函数。

具体的流程：

1. Kubernetes Adapter以Controller形式运行，创建四种资源监听器： Service、Endpoint、Pod、Node。
2. 创建并注册ResourceEventHandler，它会将资源更新事件封装成任务发送到Controller的任务队列当中。
3. Controller的Run方法启动任务队列与4种类型的资源监听器。

这里的任务队列既存储消息通知，又存储任务处理Handler

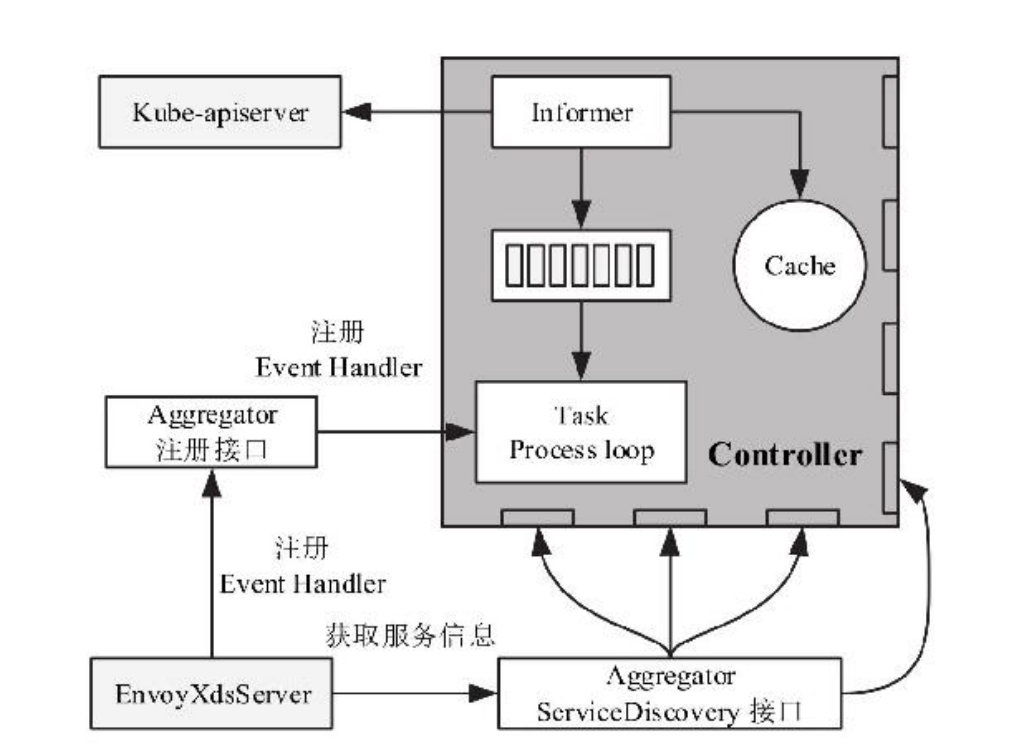


Figure :

#### 配置规则发现

Pilot配置规则指的是网络路由规则、Mixer配置规则和网络安全规则，包含VirtualService、DestinationRule、Gateway、ServiceEntry等。

支持对接三种注册中心：Kubernetes、MCP服务器和文件系统，分别实现了三种平台适配器，在平台适配器之上，进行了抽象接口的封装，即ConfigStoreCache接口。

ConfigStoreCache接口基于ConfigStore实现异步的事件通知机制：**主动同步本地状态和远端存储**，并提供接收更新事件通知及处理的能力，并且事件处理Handler必须在Controller运行之前注册。

##### Kubernetes适配器

基于Informer的监听能力。

工作原理：

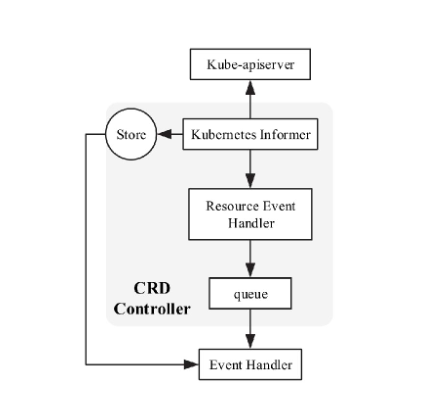


Figure :

##### 配置规则聚合

Aggregator封装ConfigStore接口，提供对缓存资源检索的抽象封装，掩盖平台的差异性，但并不能同时配置多种类型的配置注册中心。

IstioConfigStore比ConfigStore更好用，更方便。

ConfigAggregator为EnvoyXdsServer服务，提供xDS配置生成的数据源。

##### 异步通知

##### 配置分发

Envoy的基本配置包含：

* Listener
* Route
* Cluster
* Endpoint
* 路由/认证/授权等

**时机**:

1. 主动模式：主动将配置下发给Sidecar,由Config与服务更新事件触发; 由底层注册中心的资源更新触发
2. 被动模式：由Pilot接收Sidecar的连接请求DiscoveryRequest，然后做出响应(DiscoveryResponse) ；由Envoy发起主动订阅请求。

## 源码解析

pilot-discovery进程实现Pilot，启动流程：

istio.io/istio/pilot/cmd/pilot-discovery/main.go

// 读取网格的配置  
mesh, fail = cmd.ReadMeshConfig(flags.meshconfig) // 输入命令时进行指定flag来匹配配置文件  
// 失败则使用默认配置  
  
// 创建k8s的接口  
\_, client, kuberr := kube.CreateInterface(flags.kubeconfig)  
  
// 创建配置客户端  
configClient, err := crd.NewClient(flags.kubeconfig, model.ConfigDescriptor{  
 model.RouteRule,  
 model.EgressRule,  
 model.DestinationPolicy,  
 }, flags.controllerOptions.DomainSuffix)  
// 新建配置控制器和服务控制器  
configController := crd.NewController(configClient, flags.controllerOptions)  
serviceControllers := aggregate.NewController()  
// 进行命令行参数进行不同平台的注册  
registered := make(map[platform.ServiceRegistry]bool)  
for \_, r := range flags.registries{  
 serviceRegistry := platform.ServiceRegistry(r)  
 registered[serviceRegistry]=true  
   
 switch serviceRegistry{  
 case platform.KubernetesRegistry:  
 // 新建控制器  
 kubectl := kube.NewController(client, flags.controllerOptions)  
 //控制器进行注册  
 serviceControllers.AddRegistry()  
 }  
}  
// mixer的验证机制,可以忽略  
var mixerSAN[]string  
  
// 建立服务发现服务discovery service  
discovery, err := envoy.NewDiscoveryService(  
 serviceControllers,  
 configController,  
 environment,  
 flags.discoveryOptions)  
  
// 进行准入验证  
admissionController, err := admit.NewController(client, flags.admissionArgs)  
  
// 启动协程分别运行，主进程等待完成  
go admissionController.Run(stop)  
go serviceControllers.Run(stop)  
go configController.Run(stop)  
go discovery.Run()  
cmd.WaitSignal(stop) // stop在之前声明,make(chan struct{})，仅作为信号传递作用

### 关键代码

以Kubernetes注册中心为例，讲解Pilot如何监控底层注册中心。整体主要分为两个部分，服务发现和配置发现。

CRD控制器的定义：

type controller struct {  
 client \*Client // Kubernetes REST Client,创建Informer的时候使用  
 queue kube.Queue // 存储Config资源和Service呲牙u年的更新人物队列  
 kinds map[string]cacheHandler // 缓存所有Config资源和Handler  
}

#### ConfigController

用于监听Kube-apiserver中的配置资源，在内存中缓存监听到的所有配置资源，更新Config资源时调用注册的事件处理函数。

##### Controller定义

type ConfigStoreCache interface{  
 // 注册对应ConfigStore类型的事件处理函数  
 RegisterEventHandler(typ string, handler func(Config, Event))  
 // 运行控制器  
 Run(stop <-chan struct{})  
 // 配置缓存是否已经同步  
 HasSynced() bool  
}

ConfigStore为控制器核心的资源缓存接口，提供了Config资源的增删改查：

type ConfigStore interface{  
 //通过config store展示了配置的type schema  
 // type schema定义了配置类型和protobuf 编码schemaz  
 ConfigDescriptor() ConfigDescriptor  
   
 Get(typ, name, namespace string)(config \*Config, exists bool)   
   
 List(typ, namespace string) ([]Config,error)  
   
 Create(config Config)(revision string, err error)  
   
 Update(config Config)(newRevision string, err error)  
   
 Delete(typ, name, namespace string)error  
}

##### 初始化流程

ConfigController通过initConfigController来初始化，实际上是一个CRD控制器。

##### 工作机制

给每一个Config资源都注册一个Informer,并且使用EventHandler事件处理函数进行注册。

platform/kube/controller.go

informer.AddEventHandler(  
 cache.ResourceEventHandlerFuncs{  
 AddFunc: func(obj interface{}){  
 c.queue.Push(Task{handler: handler.Apply, obj: obj, event: model.EventAdd})  
 },  
 UpdataFunc: func(olg, cur interface{}){  
 if !reglect.DeepEqual(old, cur){  
 c.queue.Push(Task{handler: handler.Apply, obj, cur, event: model.EventUpdate})  
 }  
 },  
 DeleteFunc: func(obj interface{}){  
 c.queue.Push(Task{handler: handler, obj: obj, event: model.EventDelete})  
 },  
 }  
)

#### ServiceController

对外DiscoveryServer-即EnvoyXdsServer提供通用的服务模型查询接口：

adapter/serviceregistry/aggregate/controller.go

type Controller struct{  
 // 聚合底层注册中心的数据，进行监控  
 regitries []Registry  
}  
// 指明服务注册的接口集  
type Resgitry struct{  
 Name paltform.ServiceRegistry // string类型，表示名称  
 model.Controller // 接口  
 model.ServiceDiscovery  
 model.ServiceAccounts  
}

model/controller.go

// Controller定义了event controller loop  
type Controller interface{  
 // 提醒service catalog的变化  
 AppendServiceHandler(f func(\*Service, Event)) error  
 // 提醒实例的变化  
 AppendInstanceHandler(f func(\*ServiceInstance, Event)) error  
 // 运行Controller,直到接收到stop  
 Run(stop <-chan struct{})  
}

model/service.go

// 遍历Istio的Service 实例,提供统一的服务查询模型  
type ServiceRegistry struct{  
 // 系统中所有的服务  
 Services()([]\*Service, error)  
   
 GetService(hostname string)(\*Service, error)  
   
 Instances(hostname string, ports []string, labels LabelsCollection)  
   
 ...  
}

##### 工作机制

为四种资源分别注册Informer监听器，监听资源的变更。

* Service
* Endpoint
* Node
* Pod

## Istio和Kubernetes的版本匹配关系

* Istio is designed and built to be platform-independent. For our 0.1 release, however, Istio only supports environments running Kubernetes v1.5 or greater.
* 更新路径图：<https://istio.io/v0.1/docs/reference/release-roadmap.html>