istio sidecar

从pushContext说起

pushContext是pilot源码中的重要概念，推送上下文，意思很明显，就是在推送给node proxy前准备的资源，其中包括所有的*service*，这个service不止包含k8s中的service，同时包括serviceEntry导入的外部service，然后包装成istio内部的model.Service，另外就是pilot的crd资源，*virtualServcie*，*destinationRule*等，再有就是我们今天要讨论的*sidecar*资源。这里需要注意的是，这个pushContext包含了纳入istio管理的所有资源，包括multi-cluster多集群下的所有服务及集群外部的服务。这些都是用来后续生成envoy listener,cluster用到的，而我们这里要说的sidecar，则是对应用sidecar的node proxy做资源过滤，以避免全量下发影响性能。

pushContext的sidecarIndex初始化

```
func (ps *PushContext) initSidecarScopes(env *Environment) {}
```

1. 从k8s crd中获取所有的命名空间下的sidecar
2. 按有无workloadSelector分类，有则为sidecar应用范围为单个服务
3. 聚合到一起，先添加有workloadSelector的
4. 按namespace划分，添加到pushcontext的sidecarIndex中
5. 如果在root namespace(在meshConfig中配置)下配置了sidecar，并且没有workloadselector，则为全局默认的sidecar，会应用到所有ns下的服务中，如果其他ns配置了sidecar，或者服务本身配置了sidecar，则会覆盖全局的sidecar。
   pushContext为一个资源的全局视角，其中保存了所有的资源，除了这里展示的sidecarIndex资源，还有k8s service和serviceEntry转化成的[]istio model.Service资源serviceIndex，virtualservice，destinationrule，gateway，envoy filter，wasm插件等。至于pushContext的初始化，与service controller和config controller相关，此处先不再赘述。
   有了这个上下文的全局资源，就可以针对一个服务(node proxy)，将这些资源转化为envoy的listener，cluster等。
   当然针对一个特定的node proxy生成，就需要根据其namespace，workload label，端口，node流量截取方式(redirect，tproxy)等特性(node proxy在pilot agent初始化时从本地获取在向pilot发起请求时带过来的)来过滤掉非本node proxy的资源，然后再处理生成listener，cluster。
   而这个过滤的结果则存储在nodeProxy的sidecarScope中。除了sidecarScope对象，还有一个ServiceInstance对象，即与node proxy关联的服务实例，如服务部署在k8s中，则为一个pod中的一个或多个服务实例描述，主要在inbound listener时从实例中获取端口生成listener用到，这里也先抛开不谈，先看sidecarScope生成流程。

nodeProxy中sidecarscope获取

从pushContext的sidecarIndex中找出node proxy的sidecar

1. 先通过node proxy的namespace，在sidecarIndex中获取此ns下所有的sidecar资源，如果存在，sidecar类别有：
   - 有具体一个服务的sidecar在此ns下
   - 此ns下配置了sidecar(如果存在，则肯定为最后一项)
2. 如果都未匹配中，则看看全局的root ns下有没配置sidecar
3. 如果root ns下也没有，则就是我们都知道的全局下发了，即把所有的服务都添加到默认生成的sidecarscope中。

```
func (ps *PushContext) getSidecarScope(proxy *Proxy, workloadLabels labels.Instance)
*SidecarScope {}
```

这里有几点需要注意下：

- sidecarScope中存了哪些东西?
  当我们手动配置了一个sidecar后，则会生成node proxy的一个sidecarScope，后续会用一个例子让大家比较直观的理解sidecarScope中的一些关键属性，当然sidecarScope中会根据配置的一些属性从k8s等地方获取更详细的一些东西。这里简单概述下：

```go
type SidecarScope struct {
    Name string
    // This is the namespace where the sidecar takes effect,
    // maybe different from the ns where sidecar resides if sidecar is in root ns.
    Namespace string
    // The crd itself. Can be nil if we are constructing the default
    // sidecar scope
    Sidecar *networking.Sidecar

    // Version this sidecar was computed for
    Version string

    // Set of egress listeners, and their associated services.  A sidecar
    // scope should have either ingress/egress listeners or both.  For
    // every proxy workload that maps to a sidecar API object (or the
    // default object), we will go through every egress listener in the
    // object and process the Envoy listener or RDS based on the imported
    // services/virtual services in that listener.
    EgressListeners []*IstioEgressListenerWrapper

    // Union of services imported across all egress listeners for use by CDS code.
    services           []*Service
    servicesByHostname map[host.Name]*Service

    // Destination rules imported across all egress listeners. This
    // contains the computed set based on public/private destination rules
    // as well as the inherited ones, in addition to the wildcard matches
    // such as *.com applying to foo.bar.com. Each hostname in this map
    // corresponds to a service in the services array above. When computing
    // CDS, we simply have to find the matching service and return the
    // destination rule.
    destinationRules map[host.Name][]*ConsolidatedDestRule

    // OutboundTrafficPolicy defines the outbound traffic policy for this sidecar.
    // If OutboundTrafficPolicy is ALLOW_ANY traffic to unknown destinations will
    // be forwarded.
    OutboundTrafficPolicy *networking.OutboundTrafficPolicy

    // Set of known configs this sidecar depends on.
    // This field will be used to determine the config/resource scope
    // which means which config changes will affect the proxies within this scope.
    configDependencies map[ConfigHash]struct{}

    // The namespace to treat as the administrative root namespace for
    // Istio configuration.
    //
    // Changes to Sidecar resources in this namespace will trigger a push.
    RootNamespace string
}
```

Sidecar: 如果手动创建过node proxy相关的sidecar或者node proxy所在ns或者root ns下有sidecar，则映射到这里，但是如果是默认创建的sidecarScope，这里则为nil。sidecar中各个字段含义及配置，后续举例说明。

EgressListeners: egress listener是基于sidecar.egress做了再加工，顾名思义，主要用来生成出口outbound listener的，通过手工配置的sidecar.egress规则，会覆盖掉默认的outbound listener生成方式(即当没有找到node proxy相关的sidecar时，把所有导入到node proxy所在ns中服务都生成规则，即所谓的全量下发，见*DefaultSidecarScopeForNamespace*函数，virtualService和services的生成)，这里简单概述下node proxy对应sidecar存在时如何生成的services和virtualServices，这两个用于后续生成listener和cluster
service生成流程如下(见*selectServices*方法)：

- 获取导出到此ns的所有服务，遍历服务列表，和engressListener中的host做对比
- 先对比host[service.Attributes.Namespace]下的服务列表
- 没有找到再看hosts[*]，即egressListener的host中配置了*/hostname，表示本node proxy可以访问所有ns下有hostname的服务
- 假如存在服务A和服务B在不同ns下有相同的hostname，如服务A default/aa.default.cluster.local和服务B a/aa.default.cluster.local，则只留与当前node proxy相同ns的service，如果服务A和服务B都与其不相同，随机留一个
- 随后这些过滤出来的服务就是egressListener中host列表匹配中的服务，这些服务会用来生成outbound listener，cluster，通过这种过滤极大减少了生成的istio sidecar中的规则并且不会频繁的做规则下发，生成这块后续再说。
  istio-system [*, "mysql.aa.com"] *[*, "aa.default.svc.cluster.local"] default [aa.com] (serviceC)
  virtualServices生成流程如下(见*servicesExportedToNamespace*)：
  和service差不多，也是获取导入到node proxy ns下的所有vritualServices，遍历virtualServices.spec.hosts，和host匹配过滤，获取到与egressListeners相应的virtualServices。
  那么这里的service和virtualService是什么关系呢?
  以一个实际的sidecar例子来看，我们通过workloadSelector指定了sidecar规则要应用到的node proxy为ratings服务，default这个ns下。而我们的egress配置了如下规则，其中./*中的.表示当前ns，即等价于default/*，即会将default，prod-us1和istio-system下的所有服务和aa ns下域名为mylocalmysql.com的服务(如用serviceEntry加入的mysql服务)和aa ns下的test服务，汇总后用于之后的listener,cluster和route生成。

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
name: ratings
namespace: default
spec:
workloadSelector:
  labels:
    app: ratings
    version: v1
ingress:
- port:
    number: 9098
    protocol: HTTP
    name: somename
  defaultEndpoint: unix:///var/run/someuds.sock
egress:
- port:
    number: 9080
    protocol: HTTP
    name: egresshttp
  hosts:
  - "prod-us1/*"
- hosts:
  - "istio-system/*"
  - "./*"
  - "aa/mylocalmysql.com"
  - "aa/test.aa.svc.cluster.local"
```

而virtualService的作用呢？假如default ns下存在reviews服务有如下virtualService配置，这样在ratings的node proxy视角中，有reviews services和reviews virtual services两个，那后面生成lds, cds, rds这些用哪个呢？实际上，virtualService更像是对service的补充，其中的http描述的为路由信息，主要用于rds中生成路由，对于没有virtualService的service，则会生成默认的路由，因此在rds生成时，会先遍历此node proxy下的virtualService，然后找到其对应的service，因为service中有port等重要信息，然后再从node proxy下的service中剔除掉这些有virtualService的，两者的不同就在于virtualService中的路由是手动配置的，service中的路由是默认生成的，然后合并再去生成rds。

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
name: reviews-route
spec:
hosts:
- reviews.prod.svc.cluster.local
http:
- name: "reviews-v2-routes"
  match:
  - uri:
      prefix: "/wpcatalog"
  - uri:
      prefix: "/consumercatalog"
  rewrite:
    uri: "/newcatalog"
  route:
  - destination:
      host: reviews.prod.svc.cluster.local
      subset: v2
- name: "reviews-v1-route"
  route:
  - destination:
      host: reviews.prod.svc.cluster.local
      subset: v1
```

这里有个细节则是有没有可能virtualService找不到对应的service？如果是k8s中的服务，是必然可以找到的，但是如果virtualHost中的hosts为www.test.com这样的域名，并且没有用serviceEntry添加外部service，这时候也就没有对应的service了，但不影响rds的生成。

另外一点则是在路由配置中的destination，是一个host+subset组成，这个host也是需要从node proxy过滤得到的service列表中找到对应的service，然后以service的信息生成envoy rds中路由的cluster，简单说明下，listener中的rds是通过ads动态获取的，通过listener到rds，通过rds中的路由再到cluster，再到endpoint。有些listener，如inbound listener中的路由配置并不是动态获取的，而是配置死的，这个单独写文章探究下inbound和outbound listener的生成。

粗略看下rds中生成逻辑

```
func BuildSidecarOutboundVirtualHosts(node *model.Proxy, push
*model.PushContext,
 routeName string,
 listenerPort int,
 efKeys []string,
 xdsCache model.XdsCache,
) ([]*route.VirtualHost, *discovery.Resource, *istio_route.Cache) {
 virtualHostWrappers := istio_route.BuildSidecarVirtualHostWrapper(routeCache,
node, push, servicesByName, virtualServices, listenerPort)
}

//buildSidecarVirtualHostWrapper会分别对virtualHosts和service(移出掉virtualhost对应
的service)生成virtualHostWrapper，再合并。
//virtualHosts是在buildSidecarVirtualHostsForVirtualService中进行的，对其spec.hosts
逐个查找对应的service记录到Services字段中,没找到的则记录到VirtualServiceHosts字段中。
//services生成wrapper是在buildSidecarVirtualHostForService中进行的。
//在virtualHostWrapper中很重要的一个字段为route，用来生成路由规则的，其来源于
virtualService中的route.destination，通过BuildHTTPRoutesForVirtualService方法去生
成。其中的action对应envoy route配置中的virtualhosts.routes，其中routes.route则为每一项
规则，routes.route.cluster则对应action.ClusterSpecifier
//clusterSpecifier的生成则是通过destination中的host，找到对应的service，在通过
service.port和destination中的subset生成clusterWeight(n :=
GetDestinationCluster(dst.Destination, serviceRegistry[hostname],
listenerPort)),
//再之后就是生成rds的virtualHosts。
VirtualHostWrapper{
 Port:     port.Port,
 Services: []*model.Service{svc},
 Routes:   []*route.Route{httpRoute},
}

out = append(out, VirtualHostWrapper{
 Port:                port,
 Services:            services,            //在service中找到了virtual service的
host，每个virtualservice可能有多个services，按正常应该都能够找到的，egress中的services存
储了所有node proxy可以到达的service列表，但是如果virtual host是一个域名，此时就找不到了。
 VirtualServiceHosts: hosts,         //在service中没找到的virtual services的host
 Routes:              routes,
})
```

Services：这个servcies是用来生成cds用的，上面我们已经看到egressListener是个数组，每个egressListener都是一个出口过滤规则，其中包含了一组services和vitualServices，而这的servcies则是要把所有的servcies汇聚到一起用于后续生成cluster，有一点注意的是，virtualServices中route.destination.host也是一个service，并且和virtualServices的hosts并不一致的，有可能会配置成serviceEntry导入的外部服务，k8s中其他服务或者一个domain，也是要添加到services中来的。

sidecar的egress listener

sidecar的ingress listener

ingress listener会覆盖掉默认通过service创建的inbound listener，默认是替换，即不会再生成k8s service的inbound listener然后进行merge，而是用ingress的listener全部替换掉。
captureMode为DEFAULT, IPTABLES, NONE
当为DEFAULT时，这时候ingress listener会生成到virtualinbound的inbound listener中，监听的端口是15006，并将请求转发到以端口生成的cluster inbound|9098||。
当为NONE时，表示没有traffic capture，这时候会单独创建出一个inbound listener，监听配置的端口9098，并将请求转发到以端口生成的cluster inbound|9098||。

YAML

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: ratings
  namespace: default
spec:
  workloadSelector:
    labels:
      app: ratings
      version: v1
  ingress:
  - port:
      number: 9098
      protocol: HTTP
      name: somename
    defaultEndpoint: unix:///var/run/someuds.sock
    captureMode: NONE
```

这里不再对比没有sidecar ingress配置，sidecar ingress的capture mode为default和none时的listener是什么样了，我们看一下生成的cluster是怎样的。
这里可以看到是通过defaultEndpoint生成的，这里识别出了value为一个socket管道，所以lbEndpoints设置为了pipe，也可以配置k8s fqdn类型地址，即配置service name。

```
//./istioctl pc cluster ratings-v1-85cc46b6d4-4kngk -ojson
{
    "name": "inbound|9098||",
    "type": "STATIC",
    "connectTimeout": "10s",
    "loadAssignment": {
        "clusterName": "inbound|9098||",
        "endpoints": [
            {
                "lbEndpoints": [
                    {
                        "endpoint": {
                            "address": {
                                "pipe": {
                                    "path": "/var/run/someuds.sock"
                                }
                            }
                        }
                    }
                ]
            }
        ]
    },
    "circuitBreakers": {
        "thresholds": [
            {
                "maxConnections": 4294967295,
                "maxPendingRequests": 4294967295,
                "maxRequests": 4294967295,
                "maxRetries": 4294967295,
                "trackRemaining": true
            }
        ]
    },
    "upstreamBindConfig": {
        "sourceAddress": {
            "address": "127.0.0.6",
            "portValue": 0
        }
    },
    "commonLbConfig": {},
    "metadata": {
        "filterMetadata": {
            "istio": {
                "services": []
            }
        }
    }
}
```

这里说一下sidecar ingress的作用，上面例子已经指出了一种情况，endpoint设置为了socket管道，我们会有这样的通信需求，pod中服务监听的是socket管道，那外部服务请求时，就可以先拦截到15006端口，再转到socket pipe中。

再有临时有需要将流量重定向的需求，如需将访问服务A的流量转到服务B，可以设置sidecar的ingress defaultEndpoint做到。主要做的就是拦截再转发。
而captrueMode为none或者captrueMode不为none，但node proxy的TrafficInterceptionMode为none(这时候bindToPort为true，顾名思义，就是创建的inbound listener绑定到一个新的端口上)，会新增一个新的listener监听配置的端口，如9081(需要注意pod主服务的端口假如为9080，不能冲突了)，然后配置defaultEndpoint为本地的9080，这时候外部请求需要请求9081端口，不太能想到什么场景下需要用到。
node proxy的TrafficInterceptionMode表示node如何截获流量并转发给envoy，这个可以在meshconfig中(在名为istio的configmap中调整，应该也可以针对单独一个node进行调整)，这个属于静态配置，但是有时候需要动态对流量的captrue行为做一些”调整“，如sidecar配置中这种情况，所以在sidecar中做了一个针对node proxy级别的流量调整参数captrueMode。
具体TPROXY,REDIRECT展开来讲篇幅太长，与iptables规则牵扯太多，这里先跳过了，默认istio采用的是REDIRECT。

```
// TrafficInterceptionMode indicates how traffic to/from the workload is captured and
// sent to Envoy. This should not be confused with the CaptureMode in the API that
indicates
// how the user wants traffic to be intercepted for the listener.
TrafficInterceptionMode is
// always derived from the Proxy metadata
type TrafficInterceptionMode string

const (
    // InterceptionNone indicates that the workload is not using IPtables for traffic
interception
    InterceptionNone TrafficInterceptionMode = "NONE"

    // InterceptionTproxy implies traffic intercepted by IPtables with TPROXY mode
    InterceptionTproxy TrafficInterceptionMode = "TPROXY"

    // InterceptionRedirect implies traffic intercepted by IPtables with REDIRECT mode
    // This is our default mode
    InterceptionRedirect TrafficInterceptionMode = "REDIRECT"
)
```

```
// getBindToPort determines whether we should bind to port based on the chain-specific
config and the proxy
func getBindToPort(mode networking.CaptureMode, node *model.Proxy) bool {
   if mode == networking.CaptureMode_DEFAULT {
      // Chain doesn't specify explicit config, so use the proxy defaults
      return node.GetInterceptionMode() == model.InterceptionNone
   }
   // Explicitly configured in the config, ignore proxy defaults
   return mode == networking.CaptureMode_NONE
}
```