导读

Feature测试环境(Feature Pool)是指开发同学在开发时候使用的环境。开发同学在自己的dev分支上开发某一个新的功能，在本地开发完成以后需要在Feature测试环境进行测试。通常一个开发集群包括虚拟机（集群）、负载均衡以及DNS。现有的开发集群系统名叫Monterey，这套系统基于Openstack，缓存了大概3000台的虚拟机，利用Openstack Neutron LbaaS实现7层软件负载均衡，所有Feature测试环境共享同一个软件负载均衡网关。所以这套系统创建一个Feature测试环境的主要步骤是：从虚拟机缓存池中取出一台虚机，把虚机配到软件负载均衡上面，然后再创建一个新的域名CNAME到网关VIP的A记录，并且通过Neutron LbaaS配置一条匹配域名的L7规则到后端的虚机。这里我们给这个共享网关配置了一个wildcard证书，因此能够支持所有Feature测试环境不同CNAME的HTTPs访问。在完成这些工作后, 一个Feature测试环境就创建完整了。

这套系统本身已经很稳定运行多年，平时也不需要很多维护工作，但依旧存在一些问题：

1. 虚拟机的缓存池效率相对于容器，资源利用率不高。尽管在这套实现中也用到了容器，但其实它只会在每个虚拟机里面起一个docker并且只运行一个应用。
2. eBay的基础设施已经开始全面往k8s迁移，其他环境包括预发布环境以及生成环境已经有很大的比例迁移到k8s, 反而Feature测试环境还是基于Openstack，这也意味着很多k8s的功能我们还没有通过基于k8s的Feature测试环境验证，就直接上测试和生产环境。

所以这也就促使我们要尽快开发出一套基于Tess(eBay基于k8s的云平台)的Feature测试环境以取代原有的系统，并且稳定性和可用性不能比原有的系统差。

1. 基于Istio 的方案

由于k8s本身对于计算资源的管理已经颇为成熟，也就是deploy/replicaset/pod，用Pod取代VM可以说是无缝替代，难点在于负载均衡。这里我们用到了Istio，之所以选择它主要是因为其通过定义一整套资源实现了强大的L4/L7流量管理，和Neutron LbaaS一样也支持网关模式。并且, 相对于k8s Ingress，Istio为Envoy提供的路由功能更加强大，而在Kubernetes Ingress API中进行高级路由的唯一方法是为不同的入口控制器添加注解。而且长远来看，Istio本身支持service-mesh，这也是以后我们期望发展的方向。

1.1 Istio 网关模式

Istio网关模式的架构和基于Neutron LbaaS的网络架构很类似。在这种架构中，有一个或者多个用于终结外部TLS连接，将流量引入网关在负载均衡器上配置的接入点（VIP），并且在负载均衡器上通过给不同的DNS配置相应的L7 rule, 最终将外部请求通过负载均衡器导入后端真实服务器，如图1-1所示：

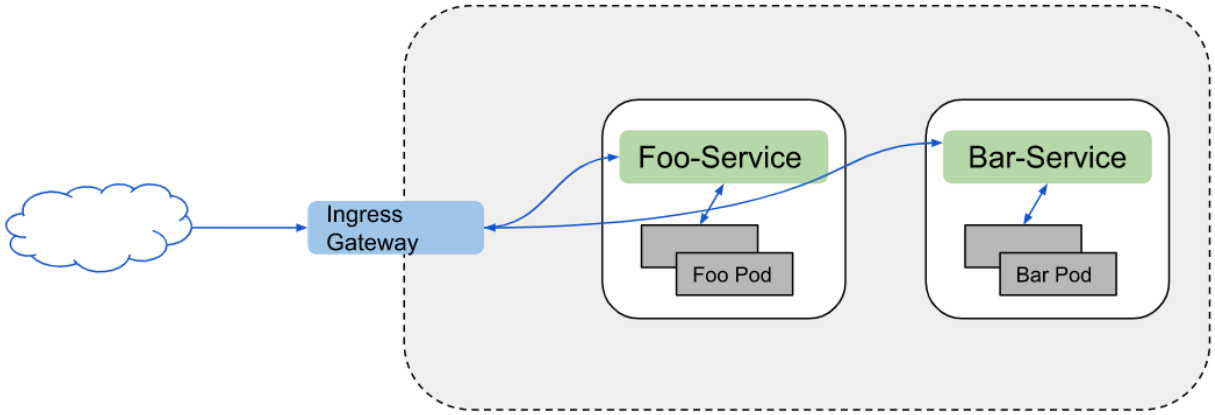

图1-1 基于Istio网关模式的架构图

Istio为了实现L4到L7的流量管理，引入了一下这些配置资源来控制进入网格，网格内部和离开网格的流量路由。

1.Gateway
2.VirtualService
3.DestionRule
4.ServiceEntry

目前我们用到的主要是Gateway和VirtualService，如图1-2所示：

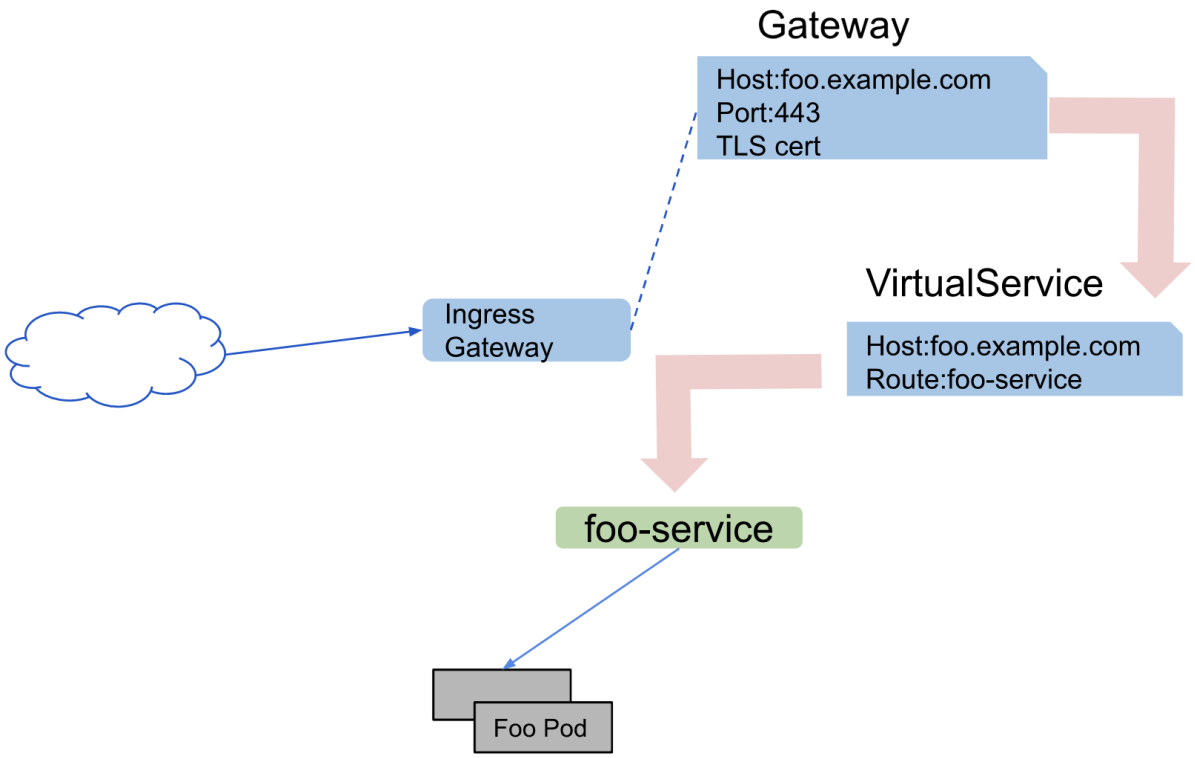

图1-2 Istio配置资源的控制流程

Gateway

Gateway用来给HTTP/TCP流量配置负载均衡器，在同一个网格中对Gateway数量没有限制，所以可以给每一个网格中的应用分别配置Gateway。实际上，Gateway配置会绑定到一组Istio-IngressGateway Pod。Gateway将L4-L6配置与L7进行了分离，它只用于配置对外公开的端口，TLS配置等等，然后将VirtualService绑定到Gateway实现通过Istio规则控制进入Gateway的流量。

VirtualService

VirtualService用于在服务网格内将请求通过一定的规则路由到后端服务，它基于Istio的服务发现能力。每个虚拟服务包含一组路由规则，比如HTTP request的host匹配，path匹配，header匹配等等，Istio 根据规则出现的顺序依次评估它们，规则出现的顺序即是它的优先级，如果有规则匹配，Istio会将请求转到虚拟服务中的Destination字段指定的实际目标地址。

比如，以下是一个简单的Gateway配置，上面配置了网关的端口、协议以及证书，这个配置允许访问 host myapp.example.com 的 https 外部流量进入网格中：

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: myapp-gateway
spec:
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - myapp.example.com
      tls:
        mode: SIMPLE
        serverCertificate: /tmp/tls.crt
        privateKey: /tmp/tls.key
```

在配置完Gateway之后，需要为进入到这个Gateway的流量配置相应的L7路由，为同一个host定义一个VirtualService，并且将该VirtualService绑定到Gateway上：

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: myapp
spec:
  hosts:
    - myapp.example.com
  gateways:
    - myapp-gateway # <---- bind to gateway
      http:
    - match:
        - uri:
            prefix: /hello
      route:
```

Istio服务发现

Istio通过Pilot组件实现了服务网格中的流量管理以及控制面和数据面之间的配置下发。Pilot 是控制面板的核心，它为Envoy提供了 xDS 服务，本质上就是将 Pod, Service, VirtualService和Gateway等信息翻译成 Envoy 需要的相关 xDS API，并将这些配置信息动态推送到Envoy，实现服务发现。

xDS包括：

CDS(cluster discovery service)

LDS(listener discovery service)

RDS(route discovery service)

EDS(endpoint discovery service)

SDS(secret discovery service)

HDS(health discovery service)

1.2 TFAP

Istio 定义了流量管理的配置资源，但是这些资源太松散，因为实现流量管理还需要定义Service。基于此，我们引入了TessFederatedAccessPoint(简称TFAP)用来封装整个L4以及L7的资源，它就像一个盒子一样，能够装入Virtualservice，Gateway和Service。他只需要定义一个TFAP就能把一个Feature测试集群所有流量相关的资源定义并且通过相应的控制器创建出来。

同时这个资源支持k8s联邦集群的功能，由于TFAP包含k8s集群的Availability Zone以及Cluster的信息，所以能够支持在不同的k8s集群中分别创建出这些资源来。

例如，下面就是一个在创建Feature测试集群用到的TFAP的例子：

```yaml
apiVersion: apps.tess.io/v1alpha2
kind: TessFederatedAccessPoint
metadata:
  name: my-app
  namespace: my-app-ns
spec:
  distribution:
    - children:
        - id: lvs02
          type: availabilityzone
      scope:
        id: ebay
        type: global
    - children:
        - id: "100"
          type: cluster
      gateway:
        apiVersion: networking.istio.io/v1alpha3
        kind: Gateway
        metadata:
          name: my-app-gateway
        spec:
          selector:
            istio: ingressgateway
          servers:
            - hosts:
                - my-app.example.com
              port:
                name: https-my-app
                number: 443
                protocol: HTTPS
              tls:
                mode: SIMPLE
                privateKey: /etc/istio/ingressgateway-certs/tls.key
                serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
      scope:
        id: az02
        type: availabilityzone
```

```yaml
supervisor: global
  virtualService:
    apiVersion: networking.istio.io/v1alpha3
    kind: VirtualService
    metadata:
      name: my-app-vs
    spec:
      gateways:
        - my-app-gateway
      hosts:
        - my-app.example.com
        - my-app-alias.example.com
      http:
        - match:
            - port: 443
              uri:
                prefix: /
          route:
            - destination:
                host: myapp-backend-svc.my-app-ns.svc.cluster.local
                port:
                  number: 8082
  - scope:
      id: "100"
      type: cluster
    service:
      apiVersion: v1
      kind: Service
      name: myapp-backend-svc
      spec:
        ports:
          - name: tcp-1234
            port: 1234
            protocol: TCP
        selector:
          run: myapp
```

TFAP定义了Feature测试环境的L4和L7配置资源，它包含了Gateway，VirtualService以及Backend Service的信息。又由于TFAP里面有集群和可用区分布的信息，所以可以指定每个资源的部署位置。从上面的例子可以看到，myapp-backend-svc其实是一个ClusterIP的Service，这个Service中所包含的Endpoint信息，也就是后端真实服务器Pod的IP和端口的信息，最终会通过Istio的服务发现机制将相应的L7规则配置推送到IngressGateway，进而将流量转发到与规则匹配的后端Pod。

1.3 TLB

TFAP中除了myapp-backend-svc的ClusterIP service，其实还包含一个Gateway的基于负载均衡的Service，它主要负责将外部流量导入到Istio网关。 这里我们又定义了一个新的资源TessLoadBalancer，里面包括了一些与负载均衡相关的配置资源：

TLB：是对硬件负载均衡的LB Pool的抽象，在SLB上是VIP到member的映射关系。
TLBVIP：对应于负载均衡的VirtualIP。
TLBMember：对应于后端真实服务器。

相对于k8s的Service， 这一套负载均衡的配置资源包含更丰富的信息，能够描述VIP以及Member的信息, 也是对社区Service的一种扩展。而Gateway的Service， 其实就是一个基于TLB实现LoadBalancer类型的Service，通过指定Service的provider annotation为TLB，TLB的控制器就会配置出一个包含VIP的Service。

另外对TLB service我们有两种实现，一种是基于Netscaler的硬件负载均衡的实现，这种负载均衡广泛地应用于eBay的生产环境，但是在Feature测试环境并没有这样的硬件设备，所以我们需要另一种类似于Neutron LbaaS的软件实现作为四层的负载均衡，这里用的就是IPVS。IPVS（IP Virtual Server）是构建于Netfilter之上, 作为Linux内核的一部分提供传输层负载均衡的模块，这里我们实现了一个TLB IPVS控制器，它能够对IPVS进行编程并实现一个LoadBalancer类型的k8s Service。

将IPVS模块和相关的TLB控制器部署在一个 Pod上，这个TLB Pod就相当于一个软件负载均衡器。然后给它配置一个VIP的subnet，由于eBay的开发环境网络是基于OVN实现的overlay网络，不支持使用BGP，因此有一个问题就是如何让将TLB创建的VIP宣告出去？当时我们采用的方案是在TLB pod所在的node上设置arp_proxy，它会响应这个VIP的ARP请求。这种方案扩展性不是很好，但是考虑到Istio只需要一个共享VIP，而且简单容易实现，所以就采用了。这里的TLB Pod既是控制节点，同时也是数据节点。TLB Pod通过IPVS虚拟出一个Virtual IP地址，然后通过TLBMember的信息找到后端真实服务器并配置相应的IPVS规则，最终把目的地址为VIP的用户请求转发到后面的真实服务器。

IPVS本身支持下面三种工作模式:
1. NAT（Network Address Translation）工作模式。在这种模式下，进出流量都需要经过调度器, 调度器会选择一个目的服务器，将进入流量的目标IP改写为负载均衡到的目标服务器，同时源IP地址也会改为调度器IP地址。
2. DR工作模式, 即Direct Routing模式, 这种模式中，调度器直接重写进入包的mac地址, 将其改为选定的目标服务器的mac地址，这样就可以到达服务器，所以在这种模式下，IPVS服务器需要和真实服务器在同一子网。
3. TUN工作模式, 即IP Tunneling模式。这种模式中，调度器将进入的包通过IP封装技术（IP encapsulation）重新包成一个IP包, 然后发送给选定的目的服务器，目的服务器处理后，通过DSR(Direct Server Return)直接将响应发送给客户。

在TLB IPVS实现中，我们用到的是TUN模式，主要基于以下两个考虑：

1. 为了避免NAT模式，进出流量都要进过TLB Pod，导致其成为瓶颈。
2. VIP的subnet和后端真实服务器的IP地址不在一个subnet。

TUN模式下IP地址的封装过程如图1-3所示：

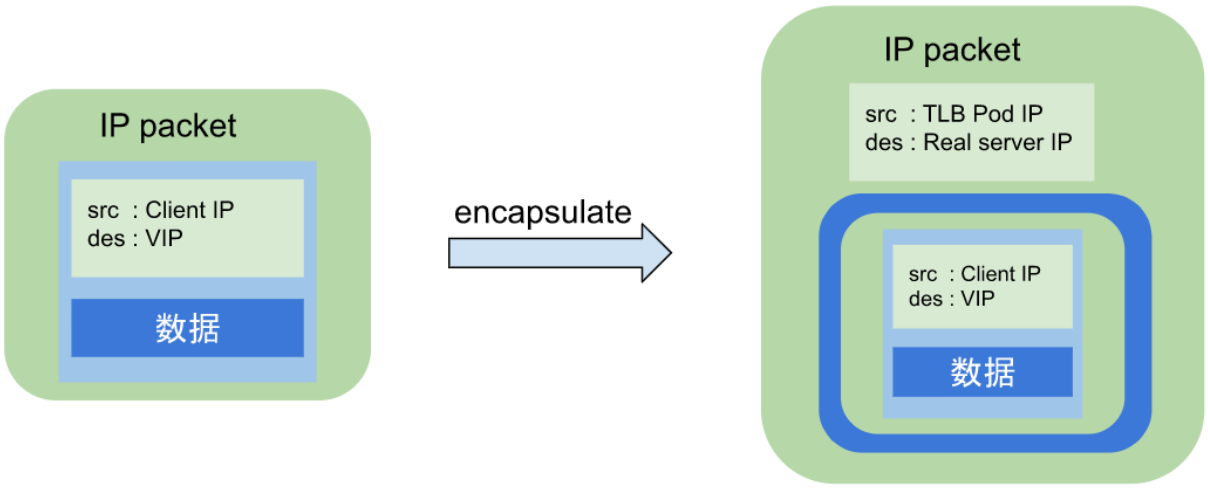

图1-3 IPIP tunnel数据包封装图

TLB的IPVS实现为Istio Gateway提供了L4的负载均衡，而Isito本身通过GatewayPod实现了L7流量的管理，Istio Gateway pod本身是一个Envoy集群，所以到这里，一个完整的具备L4/L7流量管理的负载均衡器实现了，其架构图如图1-4所示：

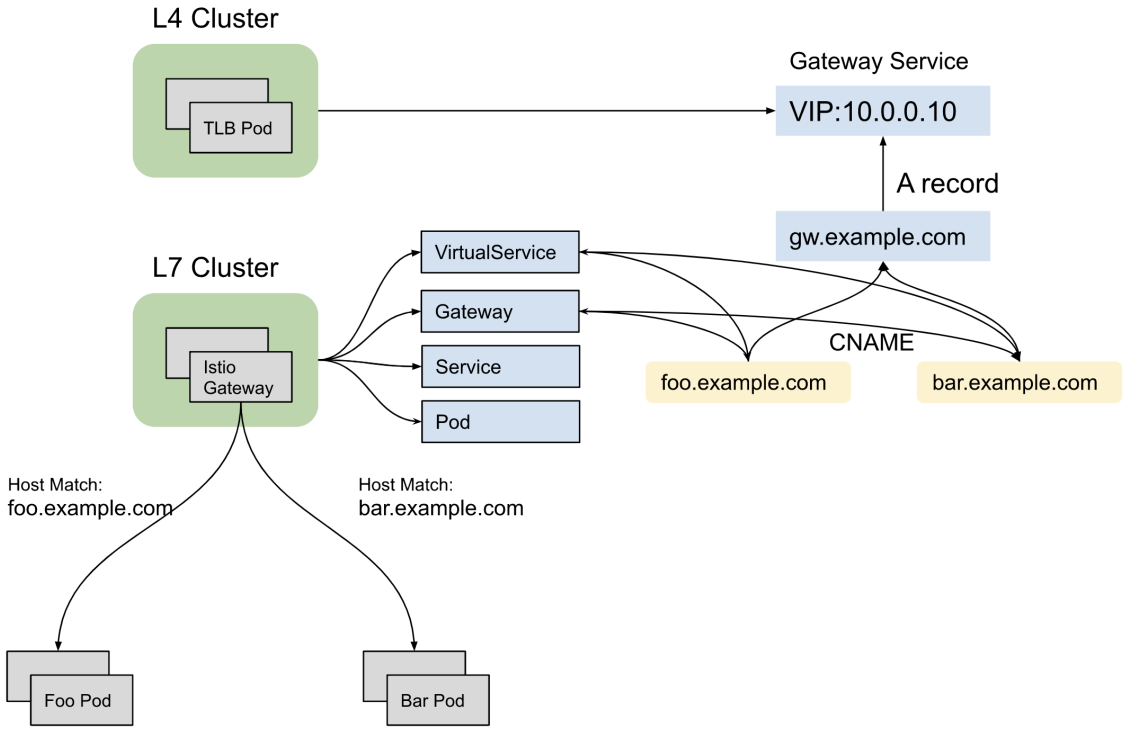

图1-4 L4/L7流量管理的负载均衡器架构

1.4 NameService

解决了软件负载均衡的L4/L7的配置资源管理问题之后，由于每个Feature测试环境可能会创建多个CNAME，所以我们需要定义一个新的资源来管理DNS，于是引入了NameService。这里是一个Feature测试环境的NameService的spec示例：

```yaml
apiVersion: v1alpha2
kind: NameService
metadata:
  name: my-app
  namespace: my-app-ns
spec:
  alias:
    - my-app1.example.com
    - my-app2.example.com
  name: gateway.svc.example.com
```

其中的name是Istio gateway 的A record，从这里也可以看出，所有Feature测试环境的CNAME都指向同一个A record 并且通过Virtualservice的路由规则最终将请求导入到后端的真实服务器。

1.5 TFD

TFD是Tess Federated Deployment，它的主要作用是将社区的deployment联邦化以满足eBay的业务需求。eBay目前已经有上百个k8s集群，通过定义TFD，我们可以将一个deploy同时在若干个k8s集群中创建出来。

1.6 Istio Feature测试环境从创建到宕机

在准备好了NameService， TFAP以及TFD之后，一个基于Istio的Feature测试环境就创建出来了。这里其实还有一些CMS(configuration management service) controller 会监控这些资源，并且创建出相应的模型存储到配置管理系统，最终呈现给用户的是一个完整的Topology。

我们在Altus（eBay在PaaS上的一个应用生命周期管理系统）上逐步放开了用户自助创建Feature测试环境，并且我们控制了开放的应用的总量。整个系统在初期表现得还是比较稳定的，集群里面的Feature测试环境总量也是在缓慢增长，从几十个到几百个。等到将近上千个的时候Istio开始有些不太稳定，具体表现是Pilot消耗的内存一直在增长，越来越频繁地出现OOM killed，同时新创建的VirtualService/Gateway也需要更长的时间在Envoy上配置生

效。在这种情况下，通常重启Pilot和Ingress gateway pod能够让集群恢复，直到有一天压死骆驼的最后一根稻草出现了：Framework team一口气创建了400多个Feature测试集群，当时Feature测试集群总量1700个，此时整个Isito控制面和数据面彻底不工作了，不仅影响了新建的pool, 现存的测试集群也无法通过DNS正常访问，因为整个系统的L7都已经不工作了，Pilot陷入重启无法自拔。这时候遇到了各种Pilot xDS相关的问题：

1. Ingressgateway缺失listener信息, 原因是Pilot推送CDS太慢, 推送无法正常结束, 而LDS只有在CDS推送完成后才会开始。
2. Pilot频繁内存溢出，主要原因是k8s集群上除了Feature测试环境创建的service和pod, 还有很多其它应用也在创建，也就是说这是一个被不同业务共享的集群，并且上面还存在很多crashloop的pod，导致的结果就是Envoy配置信息在不断更新，最终生产新的配置信息的速度大于推送到Envoy的速度, 进而导致内存溢出。
3. 当时我们用的Istio版本是1.3，Pilot discovery service本身的实现也有性能问题，在计算RDS的时候有一个很重的for循环，其中每次循环都会做一次耗时将近1秒的gateway merge, 3000个gw就花了3000秒, 这也导致了Pilot重启之后长时间无法推送配置。最新的Istio版本已经重构成Istiod了，但是类似的问题依旧存在。
4. Pilot出现connection reset by peer，通过TCP dump发现Pilot意外地收到了来自Ingress gateway的TCP reset, 抓包分析后发现有大量的乱序TCP包以及重传，怀疑有可能是大量的配置信息推送导致了OVN网络出现性能问题。

这些问题归根结底一句话就是Pilot的discovery service无法正常工作了，这也就导致Envoy的配置信息无法正常推送。虽然经过一段时间的分析我们找到了原因，但是短期内无法解决Pilot的性能问题。这里我们也列举了一些可以优化和解决这个问题的措施：

1. Istio gateway分区(sharding), 通过创建多个gateway， 每个gateway承载一定数量的Feature测试集群，从原来的一个gateway横向扩展成多个，这样也就减轻了Pilot的负担。
2. 支持namespace过滤。由于一个k8s集群有很多用户在同时使用，其它用户所创建的pod/service并不需要生成配置信息并通过xDS推送到Envoy，Pilot只需要同步Feature测试环境的资源到Envoy。
3. Pilot增量推送xDS。Pilot现有的实现是即使集群里面只有一个endpoint发生改变，它也会全量推送整个集群的endpoint信息。

不幸的是, 这些措施短期内无法实现，而Feature测试环境还需要继续推进，所以需要寻找其它可行的方案。

2. 基于Headless service的方案

经历了一次全体Feature测试环境宕机之后，摆在我们面前的问题是：

1. Istio共享Ingress gateway方案已经不可用，无法提供L4/L7流量管理功能。
2. 既然不能共享ingress gateway, 可以考虑给每个service单独创建一个基于IPVS的VIP, 然后通过Envoy sidecar实现tls终结。但是这个方案难以实施，因为当时TLB也还没有解决性能的问题，包括TLB分组， LnP测试以及和OVN集成等。

所以当时给我们唯一的选项就是，既不用共享网关，也不给每个service单独创建VIP，直接利用headless service，也就是将service spec中的clusterIP: None，而我们自己实现的DNS控制器会给headless service创建A记录，指向Pod的IP地址。
但是还有一个问题，怎么支持TLS终结呢？其实本质上还是和Istio一样，继续用Envoy。区别在于Istio的Envoy是一个集中式的集群，每个Istio网关后面都有一个Envoy集群。这里我们便是将一个集中式的Envoy集群拆解成给每个backend pod起一个sidecar容器，通过configmap保存Envoy的静态配置，再通过secret保存cert和key，最后通过volume mount的方式挂在到Envoy容器，也就实现了TLS终结的功能。这里由于eBay的应用通常是监听非80和443的内部端口，为了能通过https直接访问pod的443端口，Envoy设置成监听80和443端口并且转发到后端应用监听的内部端口，其架构图如图2-1所示:

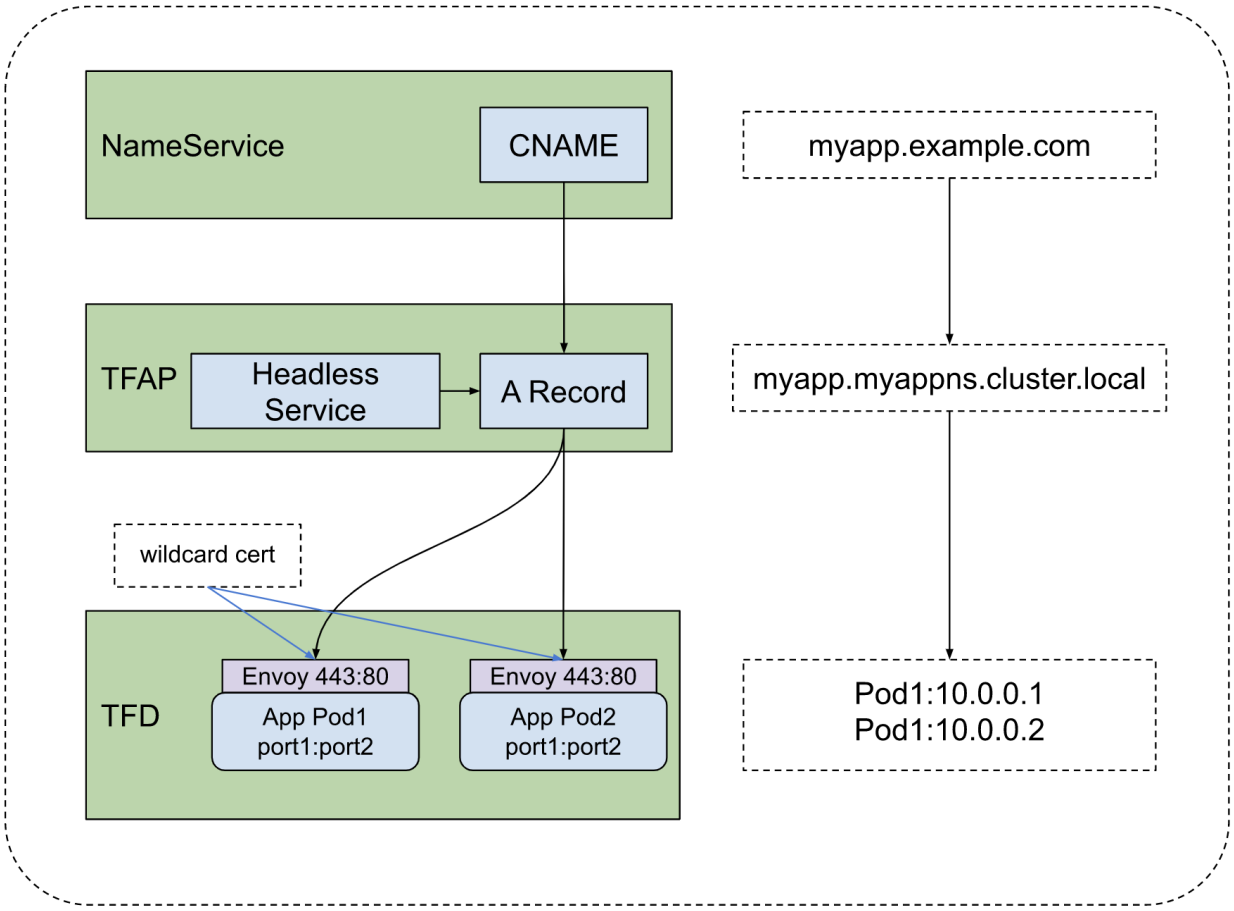

图2-1 Headless service架构图

在完成上面的配置之后，问题看起来好像都已经解决了，我们可以通过这套方案创建Feature测试环境了。但是等我们创建出一个基于headless service的Feature测试环境后去访问这个DNS，我们发现浏览器上返回的是ERR_NAME_NOT_RESOLVED，这和Loadbalancer类型的service的行为并不一致。之前的行为是会返回ERR_CONNECTION_RESET，因为尽管这个时候pod上的应用还没有部署，但是DNS是能正常解析，而这里就是因为Pod没有部署代码，endpoint里面没有ready的地址，导致A记录没有被创建。为了保持行为一致，我们在这里又增加了一个fallback的VIP，也就是说如果

headless service没有ready的pod IP，我们会通过NameService给它创建一个CNAME指向一个没有开放80和443端口的VIP的A记录，而一旦后端的pod IP变成ready了，NameService会将这个pool的CNAME指向headless service的A记录。

到这里已经大功告成，接下来需要做的是跑一些完整的测试用例，通过Jenkins不断地创建一个测试集群，部署代码，再通过http和https访问DNS看是否能够拿到正确的响应内容。随着测试的增多，逐渐发现了DNS会有延时的问题，也就是我们通过日志可以看到pool的CNAME已经被成功更新到指向headless service的A记录，但是依旧无法正常访问DNS。需要等待几秒最多到5分钟后才能正常访问，后来和基础设施部门的同事确认后，这里主要有三个原因造成了延迟。

1. CNAME的TTL时间是30秒，也就是当我们更新了CNAME之后，最多可能要等30秒钟才会生效。
2. 公司域名服务器给 Feature测试环境 DNS的域设置的无效缓存(negative cache)TTL是60秒钟，这个我们可以通过Dig命令 (Domain Information Groper)解析一个不存在DNS来模拟：

```
dig xxx.example.com

;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 1234
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1

;; AUTHORITY SECTION:
example.com.    60  IN  SOA example1.com. hostmaster.example.com. 4953537 10800 300 1209600 60
```

   也就是当我们触发了无效缓存，接下来至少要等60秒之后才会重新去解析这个DNS。后来为了这个问题专门找过infra ops的同事，之前给example.com和example.io这两个域配置的negative cache TTL分别是60秒以及300秒，后来都调整减小到30秒，在一定程度上也缓解了这个问题。
3. 除了无效缓存，另外还有一个很重要的原因就是DNS的传播延时。所有的DNS更新都是在主服务器上发生，需要多达数分钟才能同步到所有的域名服务器。这个延时是由我们的DNS架构决定的，而且也无法在短时间内把这个时间缩短。产生延时的原因如下图所示，当时还请infra ops同事给大家做了一次DNS架构的技术分享，图2-2也是infra ops同事提供的。

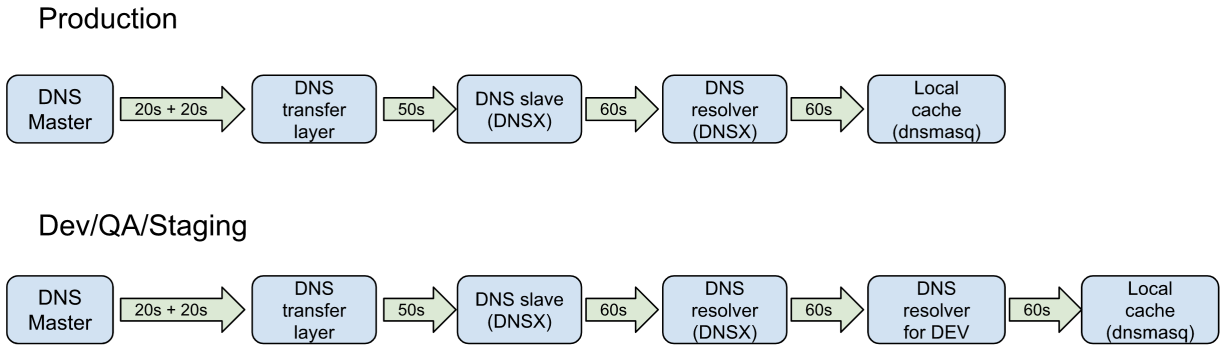

图2-2 DNS传播延迟

下面是当时测试的时候DNS延时的数据，总共创建了600多个Feature测试环境，这个延时是指从Pod部署好代码变成ready到DNS能够正常访问的延时，具体如表2-1所示：

表2-1 Headless service数据面延时

Percentile of Seconds	Code deploy
100%	102s
95%	63s
75%	32s
50%	11s
45%	2s

看到这些数据，headless service的方案也无法推进了，因为不可能接受应用启动以后要再等待一分钟才能访问DNS。

3. 基于TLB和Envoy sidecar的方案

到这一步，我们唯一的选项就只有用TLB service了，也就是会给每个Feature测试环境分配一个VIP, 创建一个基于IPVS的service，同时给VIP创建A记录和CNAME，这样就能解决DNS频繁更新造成的延时，这个方案主要需要解决下面几个TLB的问题：

1. TLB和OVN集成, 原有的arp_proxy方案无法支撑Feature测试环境的规模。
2. TLB分组（sharding)。由于有了Istio的教训，我们需要在一个集群里面创建多个TLB Group, 每个TLB group相当于一个软件负载均衡器，具有大概1000个VIP的容量，每个Group的控制面和数据面是隔离的。
3. TLB LnP测试。我们预计可能要创建4000个TLB service，为了验证TLB service的稳定性和可扩展性，在LnP测试中我们将创建8000个service。

这里我们引入了一个集中式的OVN路由控制器，TLB工作在active/active模式，通过ECMP选择TLB Pod，并且实现了TLB Pod健康检查机制。这个控制器会给active的TLB pod在OVN网关上配置一条静态路由，将所有目的从这个VIP所在的subnet的请求转到这个TLB Pod 的IP，具体架构如图3-1所示：

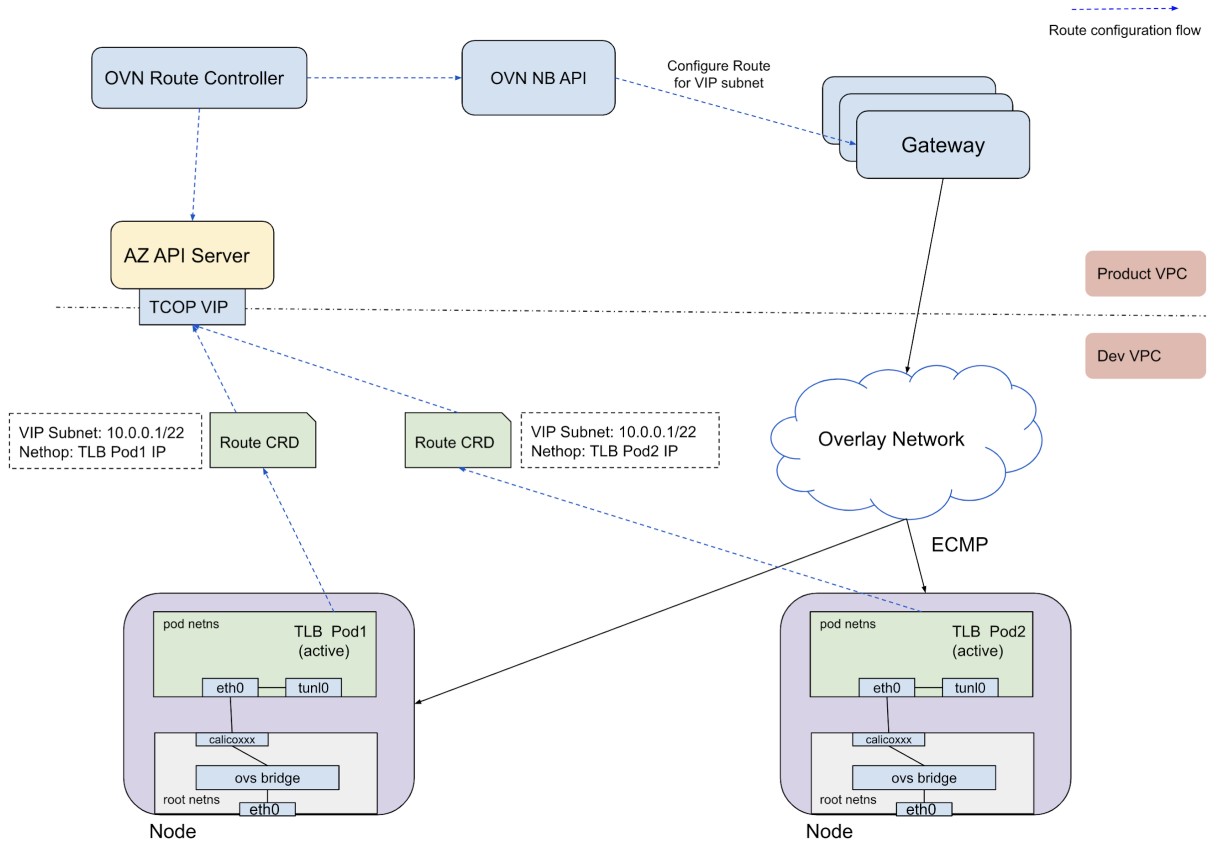

图3-1 OVN路由控制器架构图

为了实现L4的分组，我们定义了TLBGroup的资源，每一个TLBGroup会配有一个VIP的子网以及相应的OVN路由。目前一个TLB group有一千个VIP，通常硬件负载均衡的容量也是在1000左右，所以可以把一个TLBGroup当成一个软件负载均衡器。

最后是一组TLB service的LnP数据，这里面统计的时间是从一个TLB service创建到TLB控制器给service创建VIP并且更新状态的时间，具体如表2-2所示：

表2-2 TLB service LnP测试结果

SVC/Lat percent	50th	75th	90th	99th	100th	
100	8.88	8.89	8.90	8.92	8.92	1 TLBGroup
200	12.91	19.00	19.11	19.31	19.31	
300	17.46	17.56	22.29	22.32	22.54	
400	19.45	19.77	19.89	19.93	20.02	
500	25.3662451	27.27600085	29.81157702	30.80866498	34.76745913	
1000	32.30378577	38.47817151	45.15160203	51.42680498	51.89158787	
2000	33.70930234	46.26783893	47.56128547	93.78396767	93.84866748	
3000	54.87070244	147.3983379	365.7067854	423.5828028	474.9209606	3 TLBGroups
5000	391.6924965	481.1934715	540.4453897	605.0673841	605.0673842	

基于以上工作，TLB + Envoy sidecar的方案就基本定下来了，图3-2是这套方案的数据面流程图：

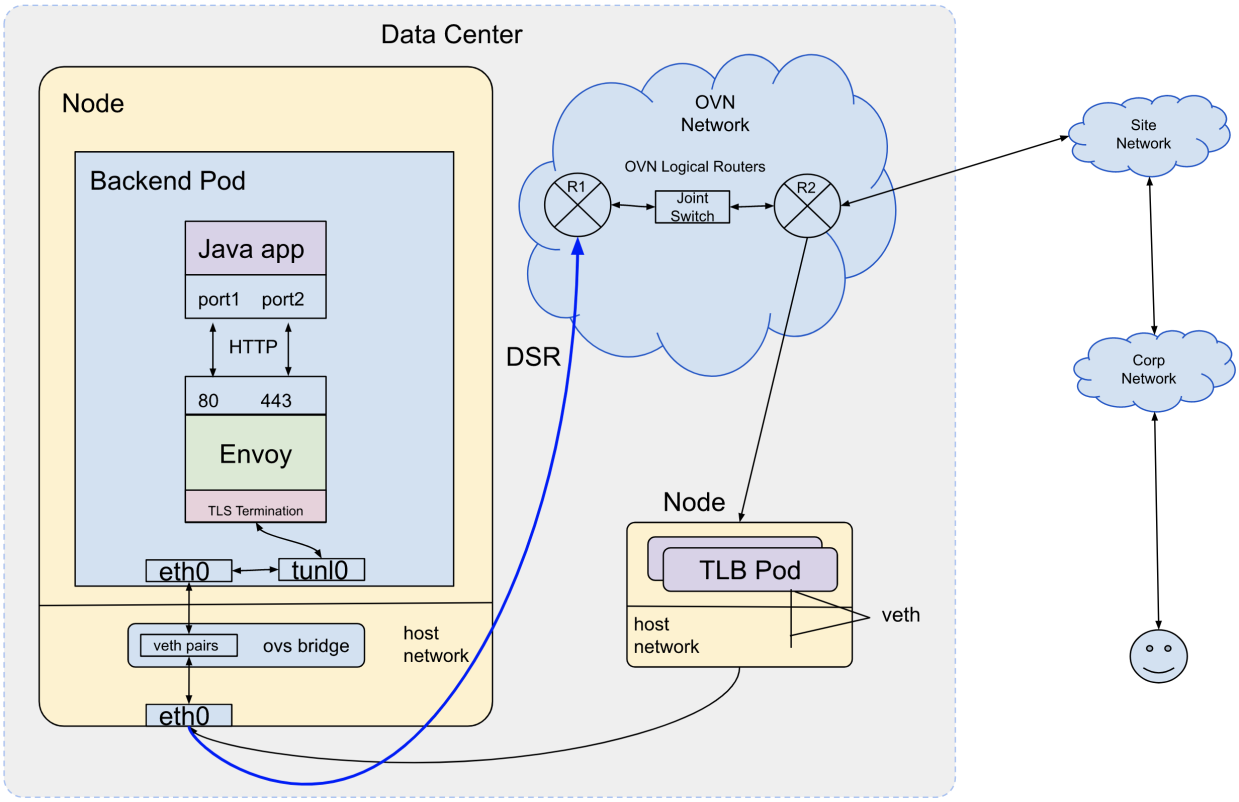

图3-2 TLB + Envoy sidecar 数据面流程图

后期我们为这套方案做了大量的测试工作，包括运行E2E CI，会同时对Feature测试环境的控制面板和数据面做验证，在两个月的时间里，我们创建和删除了14k个Feature测试集群。同时还模拟了Istio被压垮的场景：我们以每五分钟创建20个pool的并发量，创建1000个Feature测试集群，耗时2天半，共创建2500个Feature测试环境, 同时观察各种控制器的CPU/Memory以及性能指标，验证发现都很稳定，这也就证明了这套方案的扩展性已经没有什么问题了。

Feature测试环境的第一个用户是framework team的site wide upgrade, 需要给eBay将近1800个应用创建Feature测试环境并且部署代码，用来做新的framework升级验证，到这里我们对这个方案有比较充足的信心了。虽然后期还是遇到一些比如数据模型的不一致甚至还遇到了kernel的问题导致大量D state的进程，但是通过和其他Tess SIG合作，最终都解决了这些问题。最多的时候我们在集群上创建了3000多个Feature测试环境，并且通过观察各种性能指标没有发现控制面和数据面的问题，这套方案基本就稳定下来了。

3.1 Envoy sidecar

前面已经对sidecar功能有过介绍，它主要的两个功能是端口转发以及TLS终结，但是在实际的应用中我们还遇到下面一些问题:

1. 有开发同学需要在Feature测试环境上下载一个很大的文件，可能达上百兆。他们遇到的问题是在之前的Monterey Feature测试环境上每次下载到几十兆连接就断开了，后来在Tess的Feature测试环境上也遇到了同样的问题。通过分析，发现是因为Envoy静态配置里面的Route timeout默认值是15秒，导致Envoy连接backendPod的时候还没有下载完连接就中断了，后来将这个默认值改成了45秒就没有出现过上述问题。
2. eBay的应用需要将Client IP透传给后端真实服务器。在HLB的时候，我们会在VIP上创建默认的L7规则，将Client IP插入到请求的header里面，为了实现同样的功能，在Envoy sidecar里面增加了一条如下的默认规则将client真实IP透传下去：

```
{"request_headers_to_add": [{ "header": { "key": "REAL-CLIENT-IP",
 "value": "%DOWNSTREAM_REMOTE_ADDRESS_WITHOUT_PORT%"}}]
```

3. 由于目前我们sidecar用的是wildcar cert，*.example.com。这个cert无法匹配比如a.b.example.com, 但是有用户提出了这样需求，解决的办法其实也很简单，让用户去申请一个包含这个SAN list的证书，通过更改secret以及重建pod的方式解决了。也就是说基于Envoy sidecar，我们也能够提供有限的L7支持。
4. 在增加了Envoy sidecar以后, backend pod变成了两个container，我们在用户的container里面是有readiness probe的, 为了避免用户应用container没有ready但是Envoy会ready的问题，我们也给Envoy sidecar增加了readiness probe, 刚开始的时候是用的exec probe去检查health，后来遇到一个问题是在node上面造成了很多zombie processes, 这是kubernetes exec probe的一个缺陷。并且，因为Envoy sidecar是直接通过pilot-agent启动的，最后的解决办法是用bash来启动pilot-agent，因为bash作为init进程能够回收僵尸进程。

```
root@myapp-m4qpl-tess:/# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 May15 ?        00:00:11 /usr/local/bin/pilot-agent proxy --customConfigFile /etc/istio/proxy/Envoy_static.json
```

5. 最后一点，这个sidecar其实就是Istio的 pilot-agent，通过该方式其实我们已经搭了一个service-mesh的壳，同时也实现了end to end TLS，只是sidecar不是通过Istio注入，并且没有被Istio pilot接管，但是也为我们将来用service-mesh打下了基础。

4. Feature测试环境实战

这个项目是在9月28号全面上线的，当时做了全局的切换，所有新Feature测试环境的创建都在Tess。整个十月份也是domain team新功能开发上线最繁忙的时候，因为十一月份就是购物季了，在购物季的时候会对应用新功能上线做出限制，以免影响其间的线上流量。另外在这个月里，平台部门也在对eBay全站的应用升级做测试，同样需要创建大量的Feature测试环境。最后的结果就是在一个月的时间里，总共创建了大概7000个集群，并且成功率很高，原本以为上线的第一个月里需要babysit一下，但实际情况却是很少有需要帮用户做一些支持，具体十月份的数据如表4-1所示：

表4-1 Feature Pool 10月份实战指标

Job Type	Reliability	50th Percentile of seconds	90th Percentile of seconds	95th Percentile of seconds
Create	99.58%(30 failure out of **7111** jobs)	19	52	98
Delete	99.44%(24 failure out of **5912** jobs)	17	28	36

我们为Feature测试环境准备了大概5500个pod以及5000个VIP的资源。在生产环境下我们的硬件负载均衡大概也就是配置1000个VIP，虽然Staging环境由于只有一对硬件负载均衡器，上面疯狂地配置了大概4000多个VIP，但这样导致的结果就是硬件负载均衡控制面太卡，配置一个新的VIP可能要几分钟甚至数十分钟，而SLB的配置都是在秒级。

另外我们的部门总监Xu Jian曾经举过一个例子，他说某OTA公司的负责人几年前曾经给他做过一个演示，也是同样的创建一个测试集群，当时只花了17秒钟。虽然他们可能采用的是不同的技术架构，但是能够做到17秒也是很不容易的，因为技术的演进也应该朝着更快，更可靠，更高效的方向去。之前我们给Feature测试集群定的SLO是，创建一个测试集群第95个百分数是三分钟，这个指标已经完成了，但是离17秒还有差距。就目前的数据看我们只完成了不到一半，但是这里面还是有改进空间的。比如之前我们发现创建一个pod需要花将近30秒，基于这个时间我们是不可能做到17秒创建一个Feature测试环境的。后来我们就分析pod创建的每一个event，找到里面的一个瓶颈是等IP分配花了15秒，找到问题后就做了优化上线，现在一个pod创建大概是15秒。在k8s的环境里，之前大而全的集中式控制器被拆散成了很多的微服务，Feature测试环境其实是对所有的这些微服务的一个整合，这也就要求每

个组件都要高效高可靠，任何的小问题都会被放大，也只有这样才能保证Feature测试环境的成功率是2个9甚至是3个9。

在完成全部的迁移之后，我尝试做了一个面向所有用户的问卷调查，之前听说这种调查一般只有angry user才会来填，实际最后只收到个位数用户的反馈，这也从侧面说明对用户体验来说没有太多变化，可能很多用户都没有注意到这种切换。

这里再附上一张图，这张图就是这一年来基于VM和基于k8s的Feature测试环境的此消彼长的趋势图，具体如图4-1所示：

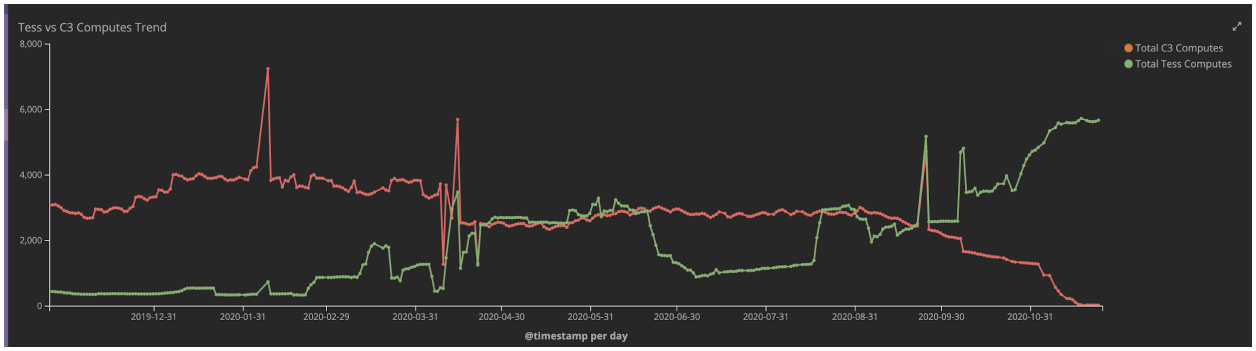

图4-1 Feature Pool VM到k8s迁移

如果用打仗来形容这张图，我觉得主要分为三个阶段：

1. 2019/12-2020/03/31：这个阶段是基于k8s方案的撤退阶段，因为当时我们还没有找到一个能够支撑我们需求的方案，为了不影响开发人员创建测试集群，不得已我们回滚到了基于VM的方案。
2. 2020/04/01-2020/09/30：这个阶段是相持阶段，我们需要做很多新的功能开发、性能验证以及持续打磨，增强我们对这个方案的信心。
3. 2020/10/01-2020/11/26：这个阶段是反攻阶段。我记得我们是9月28号全面上线的，从28号开始就没有基于VM的新Feature测试环境创建，当时存量的VM Feature测试环境有3000多个，经过一个月有大概一半被自动回收了。然后是在10月28号开始迁移存量的1400多个基于VM的Feature测试环境，用了大概三个星期的时间把所有存量的Feature测试环境都迁移到k8s，然后在12月初将原来基于VM的系统关闭。

5. 总结

整个项目历时一年半，尝试了三个方案，从我们的角度来说，是要对End to End负责。这里我对E2E的理解是：从方案POC，到Feature 开发，到集成测试，到LnP/E2E测试，到产品全面上线，再到完成系统迁移，最后到关闭旧系统，整个一条完整的链路负责到底。同时还需要有工匠精神，对项目中遇到的各种瓶颈和技术难点要刨根究底，追求极致。在整个项目中其实会遇到各种跨SIG的问题，App SIG，Network SIG， Core SIG， HRT SIG中间缺什么补什么，不管是bug修复还是新功能开发。另外还需要兼容现有的发布系统以及cloud的前端，最终建立起一套spec template驱动的provisioning系统，虽然现在只有Feature测试环境，但

是这套方案很快会继续演进到LnP和Staging环境，甚至在不久的将来也会应用到生产环境。

经过这一波三折的实践，也收获了很多的经验教训，我觉得主要有以下几点：

1. Feature测试环境上Tess的初衷是为了能够让公司的开发环境和测试以及生产环境保持一致, 而不是单纯的为了追求新的技术，所以在考虑新的替代方案之前，必须保证新的技术方案的用户体验、稳定性以及可用性要比前一代的方案有所提高而不是变得更差。
2. 开源的并不是免费的。这里并不是说开源软件不够好，而是因为开源软件的实现和公司的需求总会有一些差异，特别是对于大规模应用中的性能问题，其实还有很多优化的空间。对我们来说，具体的教训就是一个集中式的Istio集群是无法承载2000个Feature测试集群的，特别是当k8s集群上还有其它应用创建的pod，Service的时候，Istio的Pilot以及Envoy集群都会遇到各种性能问题。
3. 对于大规模PaaS的应用，L4和L7的数据面以及控制面的分区（sharding)是必不可少的, 这样既能够有效隔离和控制故障点，避免了一个控制集群宕机会影响所有应用，另一方面正因为有分区的功能，我们能够很容易地对L4/L7的控制面和数据面做横向扩展，比如一个k8s集群上面的TLB VIP资源不够了，我们只需要上线一个新的subnet同时创建一组新的TLB deployments就行了。
4. 测试以及持续更多的测试。现在我们平台部门已经没有测试岗位，开发人员要对自己写的代码负责，对于整个Feature Pool来说，需要保证整个E2E的控制面板和数据面都能正常工作, 于是通过设置Jekins CI job运行了大量的E2E测试，在过去一年里我们运行了大概十万次测试，这些测试其实也是我们信心的来源。后来我们的总监Karthik引用了一句李小龙的话来评价这些测试，我也放在这里和大家共勉：
   “I fear not the man who has practiced 10,000 kicks once, but I fear the man who has practiced one kick 10,000 times.” – Bruce Lee.

最后, 感谢阅读本文，在下一篇文章中，我将和大家深入具体地分享一下软件负载均衡的实现，敬请关注。