实现全托管，腾讯云服务网格的架构演进

原创

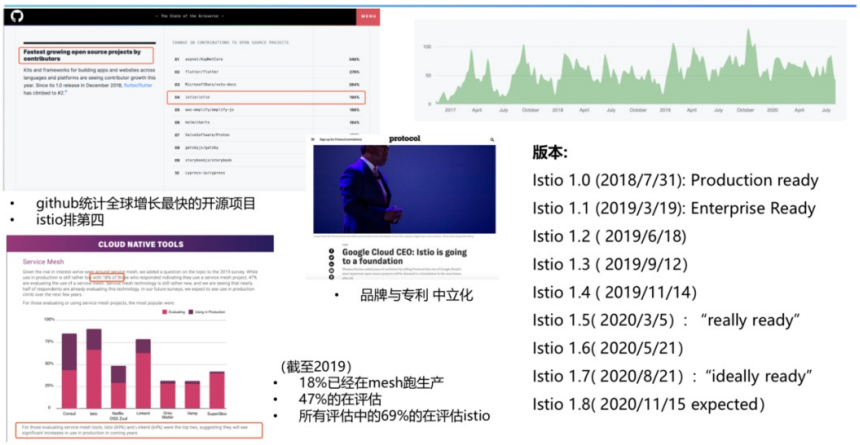
QcloudCommunity 于 2020-09-29 19:02:00 发布 1005 收藏 1
文章标签： 大数据 分布式 编程语言 人工智能 java

导语 | 腾讯云服务网格（TCM）作为一个兼容 isito 的服务网格平台，已经在腾讯内外部有诸多落地案例。本文是对腾讯云高级工程师钟华、苗艳强在云+社区沙龙online的分享整理，深度解析服务网格架构演进和发展趋势，希望与大家一同交流。

点击视频查看完整直播回放

一、 istio 现状和发展趋势

1. istio发展现状



istio现在是目前最流行的服务网格实现，它的流行主要体现在两个方面。

一是社区非常的活跃，过去一年，Istio 在 GitHub 增长最快的开源项目排行榜上名列第四。

另一方面 istio 在业界有了越来越多的生产落地。在一项云原生调研报告中，已经有18% 的用户在生产环境中使用mesh 技术，而另外47% 的用户正在进行 mesh 落地评估，而在这部分评估和测试的用户中，有接近7成的用户是在评估 istio。

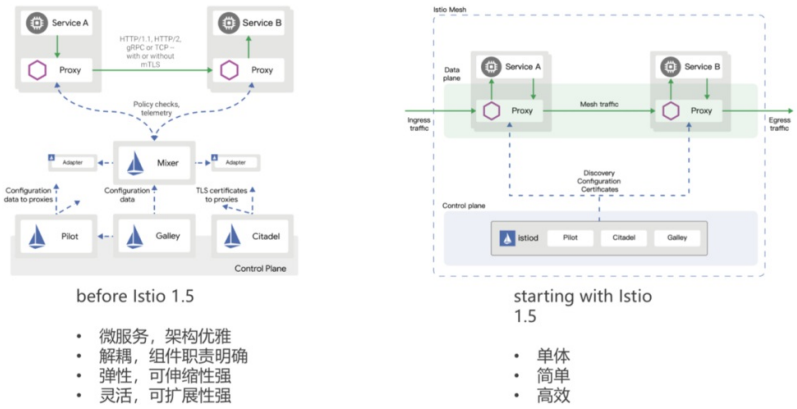
istio 今年的一个大事件是将商标转让给了OUC（Open Usage Commons ），官方宣称的目的是给 istio 提供更加中立和独立的监督，当然这个做法不太符合社区的主流期望，也就是大家一直在讨论 istio 什么时候加入 cncf，这里我们就不展开讨论。

istio 是在 17 年 5月份发布的第一个版本，到目前为止已经演进了三年多，目前有固定的发布节奏，每个季度会发布一个子版本。

istio 是在 18年7月发布 1.0 版本，宣布 Production ready，但在这之后 istio 多次的重新定义生产落地，这从一个侧面也反映了 istio 还处在一个高速演进的过程中，大的架构变迁仍然会时有发生。

所以说如果大家尝试在生产环境中使用开源 istio，开发和运维人员需要对 istio 组件有深刻的理解，以及持续的技术投入。

2. istio 架构演进



istio 最大的一次架构演进发生在今年三月份，将微服务控制面调整为单体式的 istiod。我们简单对比下当时的架构变化，并探讨下演进的背景和原因：

istio 在1.5 之前，整个控制面是一个复杂的微服务架构。其中：

> Galley 是整个 mesh 的配置管理中心，包括配置验证功能，可以对接不同的服务注册中心，服务配置在这里统一接入。他有两个主要的消费者是 pilot 和 Mixer。
>
> Pilot 是整个控制面的核心，它负责聚合服务发现数据，抽象成 istio 的数据模型，以 xDS 的形式下发给数据面。
>
> Mixer 包括 Telemetry 和 Policy 两个组件，Telemetry 负责遥测数据采集，并以 Adapter 方式对接到各种监控后端，Policy 负责在服务相互调用过程中对请求进行策略检查。
>
> Citadel 负责安全和证书管理。

但实际的实现还要更复杂些，Mixer 可以利用复杂的扩展模型对接第三方系统。控制面还包括 Prometheus Grafana Kiali 等遥测组件。还有独立负责数据面 Sidecar 模板注入的 sidecar-injector 组件等。

再来看单体化的 istio 控制面，控制面只有一个单体组件 istiod，原来的微服务组件全面纳入其中，同时这个架构直接把原来的性能消耗大户 Mixer 去掉了，后面引入了 Telemetry v2 作为 istio 新的扩展方式。

istio 单体化的发展，似乎和目前大热的微服务趋势背道而驰，我们可以稍微分析下背后的原因：

> istio 诞生在微服务架构大热的背景下，从一开始， 控制面就规划了不同职责的组件，所以 istio 最开始就选择微服务架构是很自然的结果。
>
> 在微服务里的理论背景中，不同的组件有不同的资源和弹性需求，比如上图中，Pilot 需要处理服务发现数据，需要对接数据面 Sidecar，Mixer 的消耗和流量规模正相关，这 2 个组件会占用控制面 80% 以上的资源消耗，把它们独立出来是有意义的， 他俩可以各自进行弹性伸缩。比如 Mixer 通常需要跑多份，其他组件可能只有一份。

另外微服务的控制面还做了一些假设，设计者认为控制面不同的组件通常会由不同的团队来维护，这也是典型的微服务架构理论，比如 Citadel 由安全或运维团队来负责，Pilot 和业务开发关系更紧密等等。另外 istio 最初还非常理想化的提出控制面的各个组件都可以独立部署，按需选择安装组件，在实际应用场景里却并非如此。

微服务架构看起来非常优雅，兼顾可伸缩和可扩展性，但是呢在实际使用过程中的，用户却发现该架构极度复杂。这是单体化演进的根本原因。

istio 单体化演进的另一个原因是和 Mixer 组件演进相关，Mixer 一直饱受诟病，check report 模型精巧但性能低下，随着 Envoy 支持 wasm 作为新的扩展方式，社区决定把 Mixer 从控制面中移除。

当 Mixer 移除后我们再来看控制面：Galley 的消费者只剩下 Pilot，没有独立存在的必要，所以合并到 istiod；伸缩性方面， 2个性能消耗大户去掉了一个，只剩 Pilot 比较占用资源，分组件独立伸缩的需求也没有那么重要了。

单体 istiod 受到社区的广泛欢迎，特别是我们这种 istio 相关的研发和维护人员：首先我们使用 istio 的难度降低了，安装和排错都会更简单。

另外 istio 的性能也有很大的提升，一个原因是 Mixer 的去除，另一个原因是 之前复杂的组件合并到一个进程中后，原来组件之间的 RPC 通信，变成了单体中的方法调用，原来组件之间需要传递的数据，变成了单体中的共享内存，所以说不管是响应速度，还是资源消耗，都有较大的优化。

3. 原生 istio 在企业级应用中的挑战

mesh 技术在腾讯内部已经有多年的应用实践，从最早的单纯 sidecar 模式，到最近几年以 Envoy 和 istio 为主的云原生模式，都有大量的生产实践。

但不管是内部用户还是公有云用户，大家的一致反馈是，使用 istio 成本高，落地难，这里有一些共性问题：

版本迭代快，维护周期短；

内部组件复杂；

外部依赖众多；

性能瓶颈；

多集群，混合云，托管诉求。

istio 对 每个子版本的官方支持时间比较短，根据 istio 的 Support Policy，istio 发布了新的 LTS 版本后，上一个版本只会在支持三个月。按照现在 istio 三个月一个 LTS 的节奏，大概就是每个版本官方只支持半年。

举个例子，备受关注的 istio 1.5 在今年 3 月 5 号发布， 而在今年 8 月 21 号 就宣布不再维护，整个周期不到半年。

另外 istio 每个版本只支持 k8s 最新的三个版本。比如最新的 istio 1.7 只支持 k8s 1.16， 1.17 和 1.18。这要求用户频繁地对基础架构进行升级。

即便现在把控制面合并为一个单体后，istiod内部实现也比较复杂，包括高度的模型抽象。

另外 istio 很多功能依赖其他系统，比如遥测，就用到了 Prometheus， Grafana， Kiali，Jaeger 等等。

所以使用者需要理解和运维的，不仅仅是istio本身，比如说 Prometheus 单点问题如何解决，比如说 Jaeger 选用什么存储，这些组件高可用怎么处理等等，这整套对团队的技术投入要求很高。

另外服务网格的性能开销也是用户比较顾虑的点，目前对于最新的 istio 1.7，单次 RPC 增加耗时开销大概是 3~5 毫秒，这对一些耗时敏感的系统是无法接受的。

支持协议方面，目前istio 支持的协议有限，应用层主要是支持 HTTP，gRPC，但现实企业中有大量的其他rpc 协议治理的需求：比如dubbo、thrift以及其他私有 RPC。这也会限制很多用户直接接入。

另外 istio 对多集群和多平台的支持也还不够完善，用户有很强的托管 istio诉求。

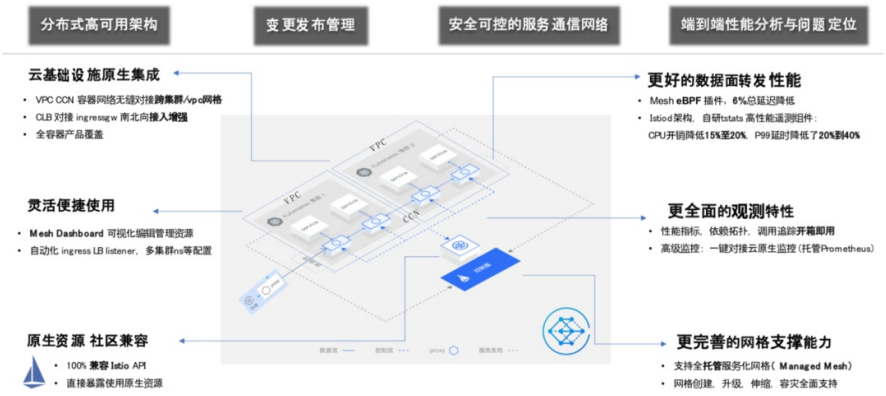
二、Tencent Cloud Mesh 架构演进

1. TCM概览

腾讯云服务网格 Tencent Cloud Mesh （以下简称TCM）的定位是云原生服务级别的网络管控基础设施。

为什么说是基础设施呢？一个原因是我们对 mesh 的理解，我们希望接入的业务团队不需要过多地关心 mesh 的底层实现，只需要专注将业务本身。

另一方面，我们认为 mesh 提供的能力，不仅仅局限在常规的服务治理领域，其他需要用到流控能力的场景，比如发布系统，比如 serverless，都可以使用 TCM 暴露的能力，下文也会介绍一些 TCM 的典型落地案例。



如上图所示，是 TCM 的一个能力概览：

TCM 和腾讯云上其他的网络和云原生设施进行了深度集成，在 TCM 中，我们使用腾讯云CLB 对接 istio ingress gateway，用腾讯云云联网实现 跨地域的多集群服务网格的互通，同时我们支持多种服务发现模式的接入，包括腾讯容器平台 TKE，EKS和 VM 等。

TCM包括一套 UI 化的 mesh dashboard，并且提供了 mesh 相关资源的自动化关联和联动修改, 降低了用户的理解和使用成本。

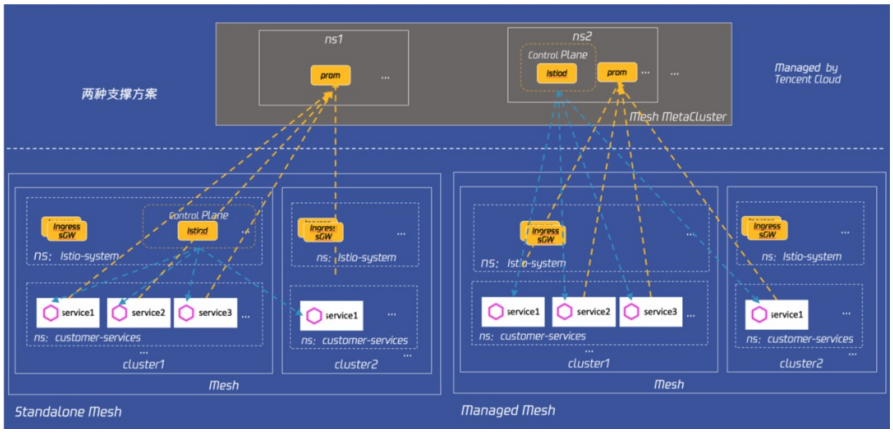
TCM 底层是直接暴露 istio api， 完全兼容开源社区。

另外我们对 istio 数据面的性能短板做了持续的优化，这主要包括两方面，一个是流量拦截时的内核态，另一个是 Envoy 流量处理时的用户态优化。

在内核态，我们开发了 mesh ebpf 插件来短路 iptables 带来的开销。在用户态，我们通过优化和定制 Envoy 的遥测组件，显著的降低了 cpu 开销和请求延时。

最后我们还提供了全托管的遥测系统，以及全托管的控制面，优化了 istio 多集群场景的中心化管理。整体可以帮助用户能低门槛的接入，高性能地落地。

2. 网站支持能力-独立与托管



TCM 提供了2种 网格模式，独立部署版和全托管版本，独立部署版本会将所有 mesh 相关组件安装到用户集群，包括 istio 组件和 TCM 相关自研组件。而全托管版本中，在用户集群不会有任何 控制面相关组件，所有组件都由平台托管。

TCM 最初只有独立部署版本，为了降低用户使用mesh 的技术门槛，我们逐渐演进出全托管的版本。不过这两种产品形态是共存的，不是替代，用户应该根据不同的需求场景进行选用：

如果对 istio 有很多个性化的需求场景，比如说公司内部的私有协议，比如说要进行深度的调参，同时对istio也有足够的理解和技术投入，可以考虑独立部署版本。

如果业务方对 istio 缺乏运维人力，或者希望更高效快速的接入 istio，又或者希望由平台来提供高可用的控制面，那么应该选择 TCM 托管版本。

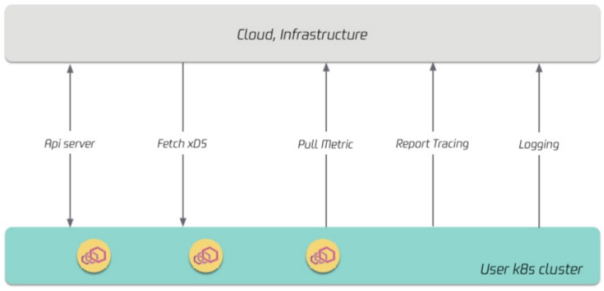
不过这两种版本对外提供的能力都是一致的，业务使用上没有太大差异。

3. TCM托管演进-数据面视角

下面介绍托管控制面架构的演进，让我们从数据面的视角下，看看数据面对控制面有哪些依赖：

数据面运行的是用户业务，服务网格需要控制面，但是用户业务并不直接依赖控制面。用户应用只是需要网格的功能（流控，安全，可观测性）。我们希望让用户把关注点集中到数据面的业务中，控制面的管理尽量由平台来负责。

将 mesh 数据面需要的数据流和控制流抽象一下，可以得到这张图：



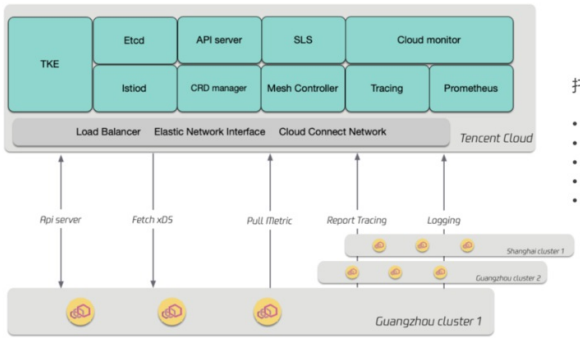
数据面视角

- 服务发现
- 流量管控和服务治理
- 遥测三要素

首先集群需要 k8s api server，数据面需要上报服务发现信息，业务可能需要读写 k8s 资源和 istio 的 crd。接下来是 xDS，这是服务网格的核心数据，包括数据面所需要的所有流控规则。然后是服务网格的可观测性，这包括了遥测的三要素：metric，tracing 和 logging。

所以用户数据面需要的是这几类服务或者说 interface，不管是微服务的 istio 还是单体的istiod 架构都是围绕以上的数据流和控制流。而云平台托管的好处之一就是，平台提供用户所需的能力，屏蔽掉后面复杂的实现，进而降低维护成本和难度。

4. TCM 托管架构- 组件概览



托管架构

- 托管 Kubernetes master
- 托管 istio 控制面
- 托管 istio crd 资源
- 托管 遥测系统
- 跨地域多集群网格

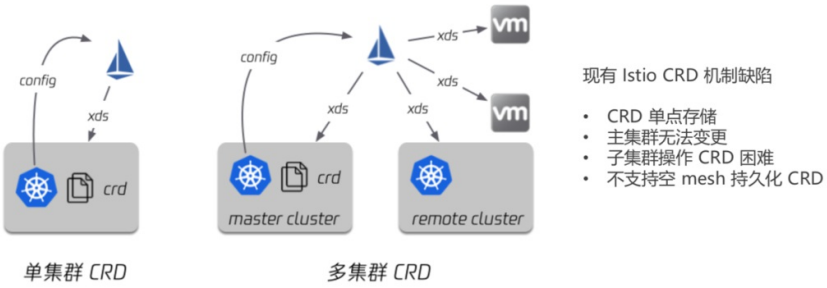
刚才的图是接口模型抽象，这张图是核心的托管组件。接口是简洁的，统一的，但是实现往往是复杂的，具体的。

首选我们提供了全托管的 Kubernetes，TKE平台，然后托管了 istio 的核心组件istiod。

我们尽量保证和 istio 社区的兼容，不过在一些开源版本还不是很好用的地方，不太符合用户需求的地方，我们也做了很多增强。

比如我们提供了中心化的 crd 存储管理，解决用户在使用多集群和混合云 mesh 中的流控配置管理的问题；另外我们提供了全托管的遥测系统，我们还支持跨地域的多集群构建成一个 mesh，mesh 中的集群可以随着业务需求进行增删，不同集群之间的服务可以自动就近访问和失效转移。

5. TCM 托管架构- CRD 托管



接下来为大家介绍 CRD 管理方式的演进。

这里说的 CRD 是指的 istio 的流控规则，包括 virtualservice，destinationrule，gateway等等，在开源的 istio实现中，这些资源的存储在用户 k8s 资源中的。

基于 k8s 的云原生扩展，大部分都是这样，把扩展的模型用 CRD 存储到用户 k8s 中，这样非常方便，扩展不用考虑存储问题，可以直接使用 kubernetes controller 模式实现。

但是这种模式在 mesh 场景下存在一些问题，特别是在多集群和混合云和混合云场景下：

CRD 无法分布存储，只能存储到一个固定集群上， istio 默认只会从一个k8s 读取 CRD

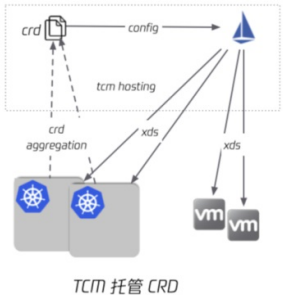
主集群无法变更，主集群是整个 mesh 的单点瓶颈

子集群操作 CRD 非常不方便

特殊场景，比如一个不包括任何集群的空网格，无法持久化流控 CRD

问题原因在于， mesh 中的流控配置带有全局性，是对整个网格生效，而在整个 mesh 中，所有 cluster 应该是对等的，出现 master 和 remote 集群的区分，仅仅是网格需要一个集群来运行 istiod，以及存储 istio crd。

如果我们将 istiod 和相关组件，以及 CRD 从用户的业务集群中剥离，那以上问题就都解决了。



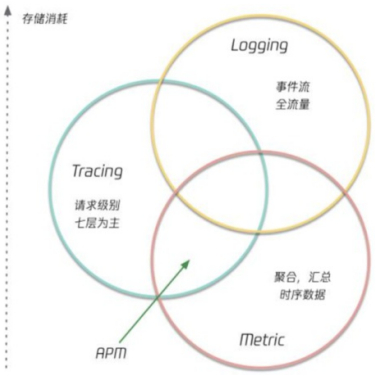

```
~ % kubectl -n istio-system get pod
NAME                                    READY   STATUS
istio-ingressgateway-1-7c57b77f48-ljpc7   1/1     Running
~ %
~ % kubectl get crd
NAME                                    CREATED AT
tkeserviceconfigs.cloud.tencent.com     2020-09-09T13:11:23Z
~ %
~ % kubectl -nbase get virtualservice
NAME    GATEWAYS          HOSTS   AGE
mall    [mall-gateway]    [*]     50m
```

在 TCM 中，我们将 CRD 和 istiod 都做了托管，这样网格中的集群和虚拟机可以按需增加或删除，平台负责提供高可用保证，网格中的集群都是 remote， 不会存在某一用户集群的故障导致网格不可用。

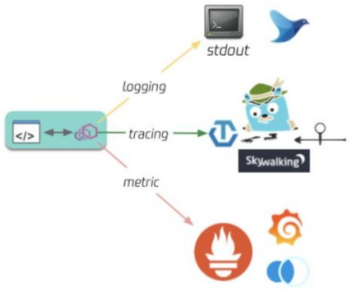
仅实现 CRD 托管还不够，因为这样会破坏用户的使用行为，因为用户习惯从自己的 k8s 集群中读写 istio 资源，比如使用 kubectl，或者 k8s in cluster 模式操纵 crd 资源。

所以我们还做了一个 crd aggregation 的实现，也就是用户可以在任一集群上读写 istio crd，而且数据始终只有 TCM 托管的一份，也不会出现数据不一致的情况。整个过程对用户透明。

6. 服务网格中的遥测



遥测三要素



Istio 中的遥测

遥测是服务网格领域重要的一环，mesh 的流行主要有两个背景，一个是容器化，一个是微服务化，容器的隔离性让服务进程更加的黑盒化，而微服务让请求链路会变长更复杂。

所以在mesh 化的业务中，遥测是必不可少的一环。可以说，如果没有一个易用且完善的遥测体系，mesh 落地是不可能成功的。

遥测数据源主要有三大类：Metric，Tracing 和 Logging，不管是传统服务监控还是云原生的网格遥测都会涉及这三类数据源，也就是我们通常说的遥测三要素：

Metric：指标数据，可聚合，如吞吐率，错误率，延迟，分位值，top值等等，通常需要使用统计学的原理来做一些设计。

实现：时序数据，prometheus 采集。

应用：服务拓扑，指标分析，监控报表，告警。

Tracing：全链路跟踪数据，请求级别。

实现：jaeger，zipkin 等数据格式，目标是七层流量，受采样率控制。

标准：OpenTelemetry（OpenTracing，OpenCensus）。

应用：服务拓扑，请求分析，链路回溯，告警源。

Logging：事件数据。

实现：Envoy access log，Envoy 自定义访问日志。

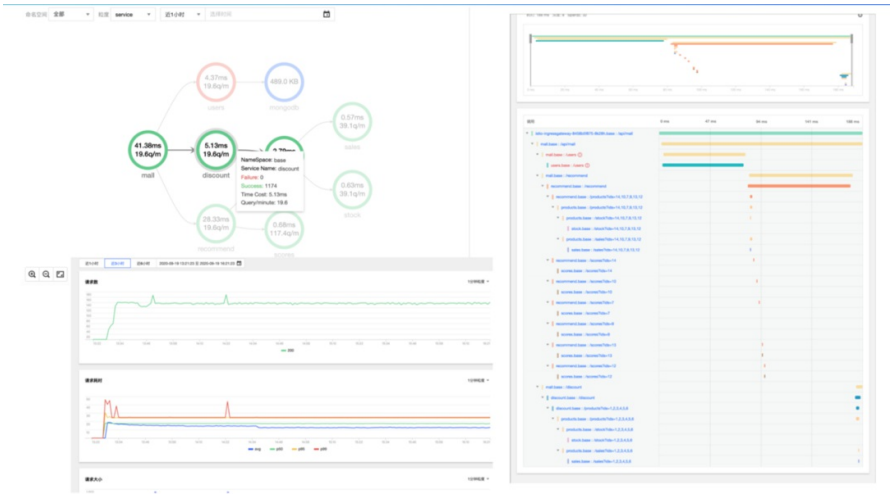
应用：链路端点分析，异常分析，指标分析，告警源。

一个生产可用的遥测系统环节非常多，包括：采集，存储，索引，聚合，计算，展示，告警等等。istio 提供的遥测开箱即用，但还是原型级别的，如果用户想独立运维一套兼容 istio 的遥测体系，不是不可以，但是成本非常高，对团队的技术和运维能力要求非常高。

同时遥测系统本身对弹性伸缩的要求很高，大部分遥测数据量的规模和业务的流量正相关。如果遥测系统不够健壮，不但无法提升业务系统的可观测性，还有可能成为整个系统的瓶颈。

而云计算本身是一个强调弹性、提供可用性保证的基础设施。将 mesh 中的遥测接入云平台，用户可以在成本和质量上都得到收益。

7. TCM 遥测概览

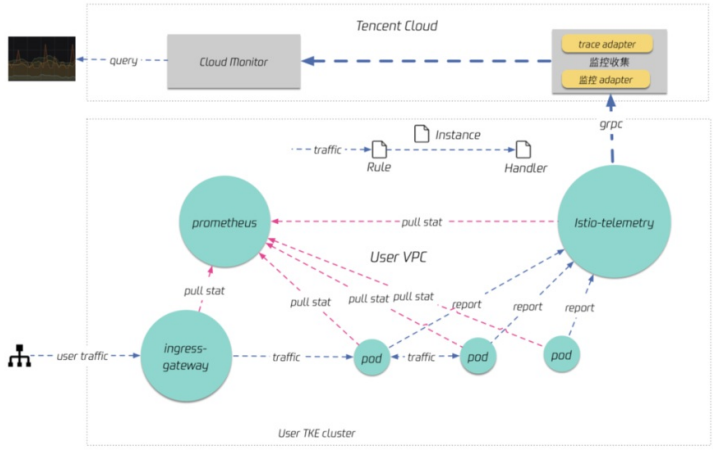


TCM 提供的遥测系统，包括服务拓扑，全链路跟踪，以及核心指标分析等等。

通过服务拓扑，我们能知道一个服务依赖的上下游，快速感知系统整体健康度。全链路跟踪可以对某一请求进行深入分析，用户可以识别出整个链路中的慢请求，系统瓶颈在哪里，有没有不合理的调用，多次调用是否可以合并为单次批量调用。

核心指标中包括 top 值分析，关于为什么需要有top值的分析，当一个系统的体量很大的时候，qps上万或者超百万这种， 系统出现非致命的异常时，平均耗时很难体现， 被中和了。而 top95 或者 top99 指标却会非常敏感，能更准确地反映系统异常。

8. TCM 遥测体系演进

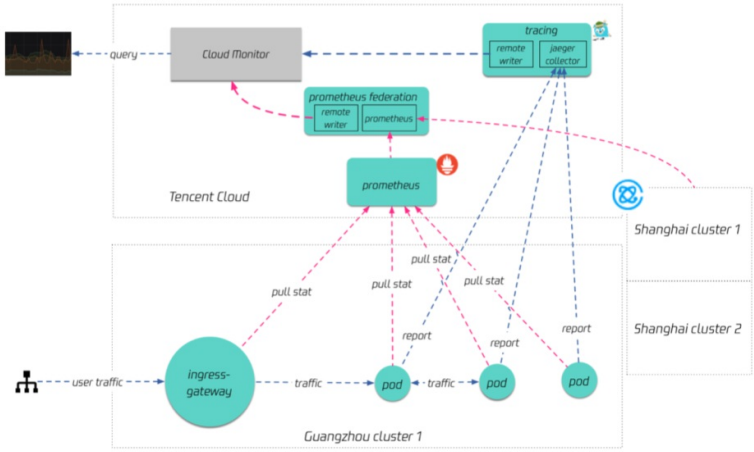


在 istio1.0 release 后不久，TCM 就开始在腾讯内部提供服务，当时社区的遥测还是基于 Mixer v1 模式，Mixer v1 架构非常灵活，可以方便的对接第三方遥测系统。

TCM 遥测 1.0 也是基于 Mixer 进行扩展，我们利用开发 mesh monitor 组件将用户遥测数据采集并上报到腾讯云监控。

但是 Mixer v1 的短板就是性能，不管是 Check 还是 Report， 这种架构都在流程中增加了一跳，而且 Mixer 的组件 Telemetry 和 Policy 都是运行在用户集群中，非常消耗资源。

9. TCM 遥测 2.0



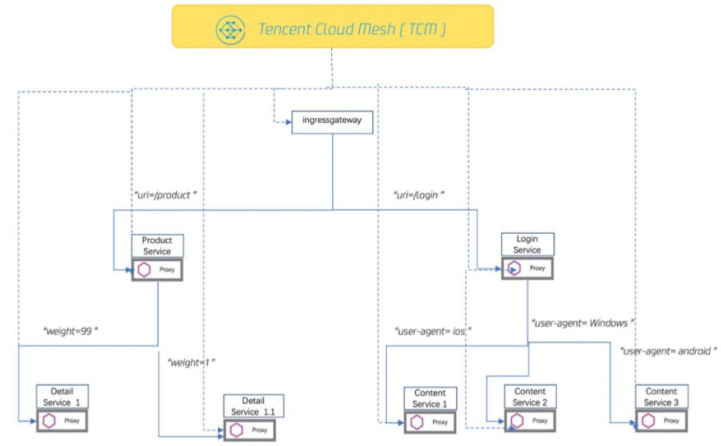
随着istio 逐渐将 Mixer v1 过渡到 Mixer v2，我们在后续版本中也将遥测架构进行了升级，新版本中 Mixer 已经移除，我们在 TCM 云平台中直接托管了 metric 和tracing 相关的组件，托管集群和用户集群利用弹性网卡打通，云平台组件直接可以采集到服务网格中的遥测数据。

这样遥测组件不会直接占用用户集群资源，监控体系的高可用由平台来提供，后续的遥测数据分析，指标计算，存储和查询等，都可以直接利用腾讯云已有的能力。

另外，我们也注意到有越来越多的多集群服务网格遥测的需求，用户的业务分部在不同的集群中，有的集群可能还是跨地域的，用户希望从单一的视图去观测多集群整体的服务状况，因此我们在遥测 2.0 中引入了 federation 架构：托管端通过腾讯云云联网打通，各集群数据在云端聚合后统一展示给用户。

三、Tencent Cloud Mesh 典型落地场景

1. 发布变更/流量管理



下文将为大家分享一些 TCM 经典落地场景，首先来看第一个场景：发布变更/流量管理，这里实际包含了两个场景 ，即应用的灰度发布变更，和请求流量分发管理。

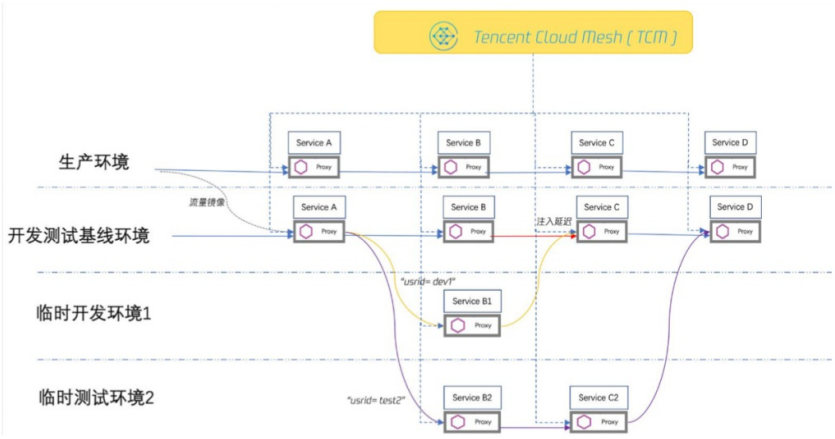
上图左边的部分，给出的是灰度发布的简单模型，当我们发布新的变更的时候，往往希望通过灰度发布的形式发布，将对用户的影响降到最低。

因此我们可以通过 TCM 的能力来完成。当发布新的版本时，可以通过 TCM 下发配置，通过设置权重，将少部分的流量导入到新的版本中。如果新版本的有问题，也只会影响少部分用户，我们可以马上切回到老版本，如果新版本功能 OK 后，再通过权重变更，逐渐的将流量全部转到新的版本，从而实现恢复发布。

右面的部分，是流量管理比较经典的一个例子，比如在页面访问时，不同的设备或系统（比如手机，平板，电脑），对于相同的内容，数据呈现的格式，可能有所差异，比如例子中给出的登录界面。

对于这种场景，通常情况下，浏览器在访问服务的时候，请求头中都会带有请求设备的平台信息，（比如图中给出的操作系统的类型），因此，我们在服务部署的时，分别针对不同的平台部署不同的后端服务，通过 TCM 的流量管理能力，结合请求的头信息，将请求发送到对应的服务后端，从而实现平台差异化的服务响应。

2. 多分支测试环境



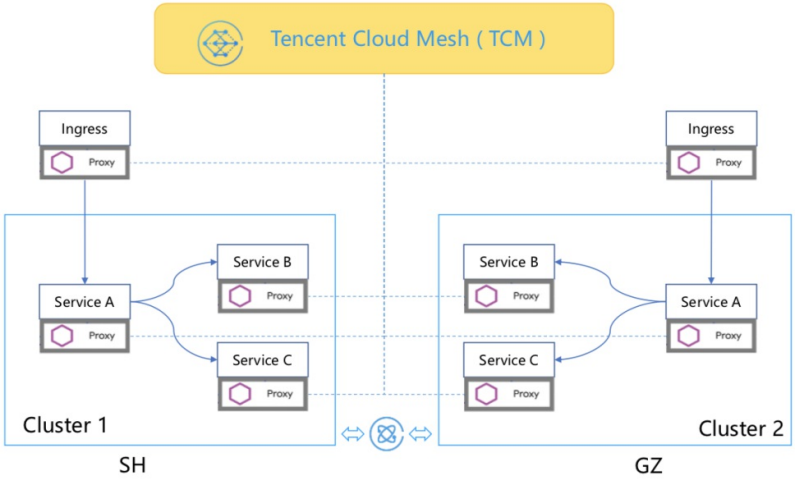
多分支测试环境，是在产品开发中很有用的一个场景。做一个比较大的系统，一般情况下都会有一套比较完整的测试环境，然后把生产中的请求 镜像到测试系统，来测试系统的新功能特性。

然而，对于一个微服务化的系统，一般都会由很多个模块组成，而每个模块可能由一个独立的团队来负责。

当团队要对功能模块做系统测试的时候，如果大家都用同一套测试系统，很可能会相互影响，从而得不到预期的测试效果，而如果每个团队都部署一套独立的测试系统，则不仅会产生大量的重复工作，造成人力的浪费，又会造成资源的浪费。Servicemesh很好的解决了这个问题。

如图所示，各个团队可以对镜像后的流量染色，即打上团队自己特定的标签，通过在 TCM 中配置相应的转发规则，即可将流量转发到需要测试的模块。而其他路径，仍然复用整体的测试环境，从而实现了测试环境的复用的同时，避免了不同测试分支的互相干扰。

3. 分布式可用架构

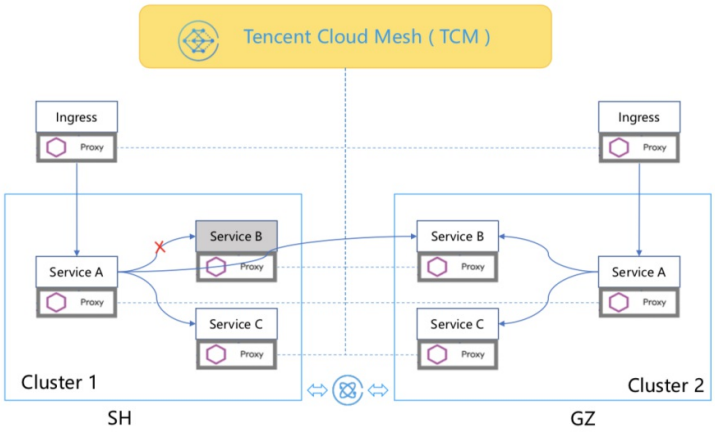


对于分布式可用架构的部署场景，比如游戏，或者电商应用，为了提高系统的高可用性，往往需要将服务分地域部署。而为了让用户得到更好的使用体验，则又希望就近接入，从而减少不必要的网络延时。

对于这种场景，Servicemesh 可以很好的解决这个问题。比如 TCM 就提供了地域感知和失效转移的能力。

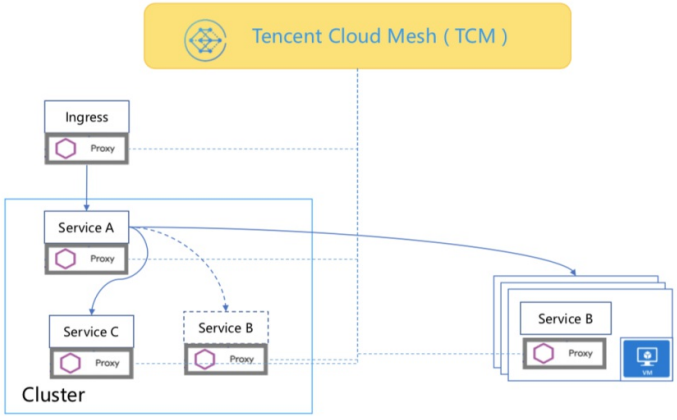
当应用以多集群跨地域方式部署时，用户可以通过配置规则，设置后端服务的地域亲和性，以及失效转移优先级，来控制流量的转发策略。

正常访问情况下，TCM 通过地域感知负载均衡能力，会将流量优先转发到本地域的服务副本中，因此可以提高用户的使用体验，同时降低了跨地域通信的网络成本。



如果本地的应用出现故障，导致不可用的时候，TCM 会通过预先配置好的失效转移优先级策略，将流量转发到其他可用地域的同名服务中，从而实现跨地域的高可用。

4. 业务跨平台混部、平滑迁移



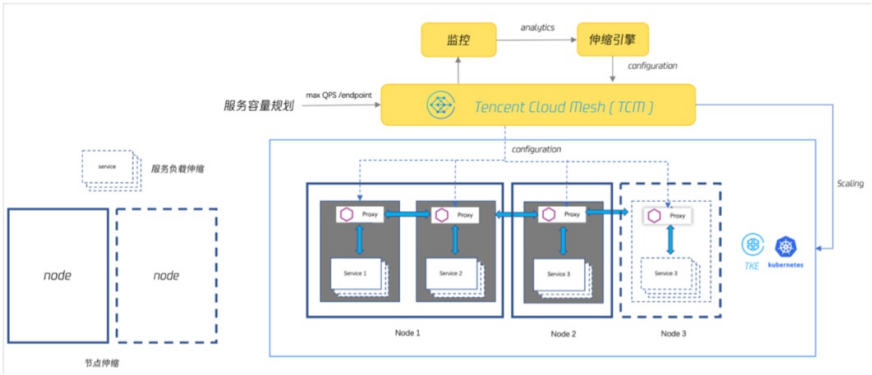
这里也是比较经典的一个场景，例如我们有的客户在进行微服务改造的时候，很难一次性将所有的应用全部以容器的形式部署，总会由于种种原因（历史包袱，或者业务本身的原因），不方便以容器方式部署应用。

这个时候，我们可以借助 TCM 的能力，将传统的比如以虚机方式部署的应用，加入到网格中，作为 TCM 统一管理的 serviceworkload。这时我们就可以通过配置相应的规则，将容器服务和虚机服务无缝对接，实现互访，从而实现微服务应用和传统应用的混合部署。

当然，混部的形式，可能只是我们临时的过渡方案，最终目的是实现全量微服务部署，以降低我们的维护成本。

随着业务的改造，那么当我们应用满足微服务部署的时候，可以通过控制流量的转发权重，平滑的将应用迁移到以微服务部署的工作负载上来，从而实现跨平台平滑迁移。

5. 成本与性能优化



接下来介绍通过mesh实现业务性能和成本优化的场景。通常情况下，微服务的基础部署平台，比如 TKE，都会提供弹性伸缩功能，会根据各项指标，实现服务实例的动态增删。

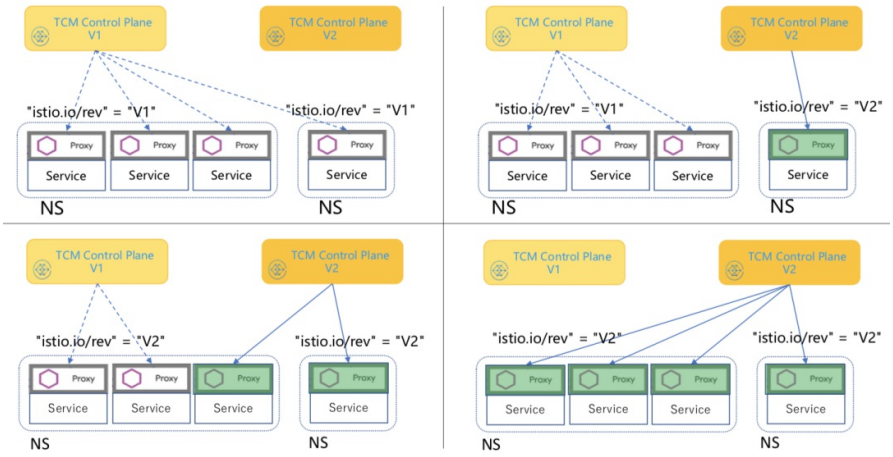
然而部署平台收集的指标，一般情况下，都只资源相关的指标，比如 CPU，MEM，带宽等信息。但是有些业务应用，其向外提供服务的性能瓶颈，不一定都是资源相关的，也可能是其内部的实现逻辑造成的性能瓶颈（比如服务对业务的并发处理能力）。

因此，通过业务相关的指标，来控制服务负载的动态伸缩，也是很多客户的诉求。实际上借助 Servicemesh 的能力，能够很好的 cover 这种场景。

通过 Servicemesh，我们可以采集到的很多业务相关的性能指标，比如 qps，请求延时等。我们可以实现一个基于业务性能数据 实现一个弹性伸缩引擎，通过 mesh 的监控系统，去收集业务的各项业务指标，来作为伸缩引擎的输入，然后去对接用户集群的 api-server，通过业务指标来控制服务负载的副本数，从而实现基于业务指标驱动的弹性伸缩策略。

业务维度弹缩策略的引入，填补了基础部署平台弹缩机制的不足，从而为用户提供优质服务的同时，又可以最大程度的降低资源成本。

6. 控制面灰度升级



前文我们提到，借助 TCM 的能力，我们的应用可以实现业务无感知的平滑升级。其实在我们使用 Servicemesh 的时候，Servicemesh 的控制面也有类似的需求。

我们知道当前 istio 版本迭代比较快，而且中间会修复用户在使用过程中遇到的 bug，要修复这些 bug，只有通过升级控制来解决。

针对控制面的升级，TCM 提供了灰度升级升级控制面的机制。在升级过程中，在保留老版本的前提下，安装新的控制面，新旧两个控制面同时存在且相互独立。

由于它们连接相同的配置中心和服务发现中心，因此两个控制面都拥有相同的全量的配置和服务数据。我们可以通过 Namespace 中设置标签的方式，来控制新创建服务的数据面由新的控制面来注入，并对接新的控制面。

通过滚动升级的方式，将所有的业务过渡到由新的控制面来接管。当所有的数据面都对接到新的控制面后，我们就可以把老的控制面下线，从而实现控制面的平滑升级。通过这种机制，我们可以很大程度上降低控制面升级对业务的影响。

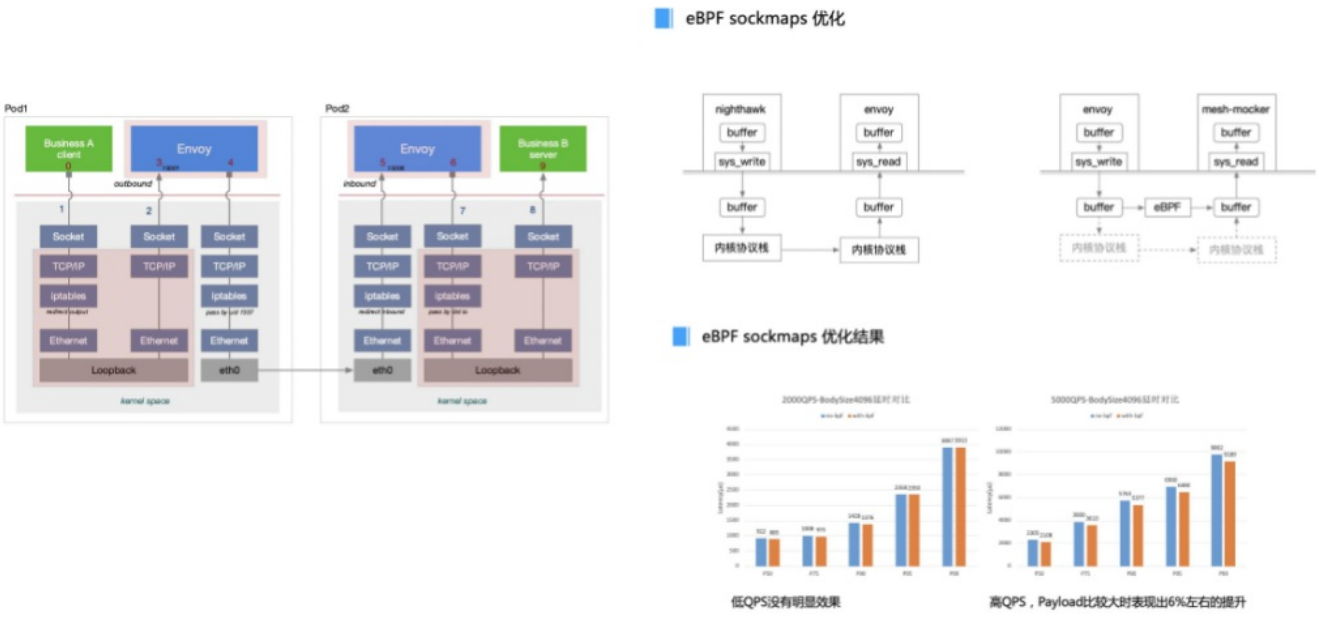
7. 低延时网格代理

在有些系统中，用户对应用之间的通信延时有比较苛刻的要求。在新一代 S ervicemesh 中，由于 sidecar 的引入，使两个微服务之间的通信路径变长，导致服务延时受到了一些影响.

同时，由于 sidecar 中，需要对流量进行各种治理操作，比如链路跟踪，遥测等功能。因此通信性能要求较高的场景下，会对 sidecar模 式的性能有一些担忧。针对低延时的场景，我们内部对 Envoy 做了深入研究，并通过两种方式，对其性能做了优化。

首先，通过缩短通信路径，来降低通信延时。如下图所示，我们使用 ebpf 技术对 sidecar 与服务之间的数据进行拦截，使数据在 socket 层进行转发，而不需要再往下经过更底层的 TCP/IP 协议栈的处理，从而减少它在数据链路上的通信链路长度。

从柱状图上我们可以看到，经过优化，还是有比较明显的优化效果。



其次，通过优化 Envoy 内部实现逻辑来达到提高性能的目的，由于 istio 1.5 版本前后，Envoy 实现机制有较大的调整，因此，我们分别以不同的方式，来优化两个版本的性能。

通过分析，我们发现，遥测数据的处理是导致性能问题的根本原因。

在 isito1.5 之前的版本中，由于 Envoy 需要提取所有消息的属性，然后批量压缩上报到 istio 的 Mixer 组件中，这些属性的提取和压缩是一个高 CPU 消耗的操作，而在处理逻辑中，Envoy 又是以同步处理的方式，将这个处理流程插入到整个流量处理的过程中，所以对通信的延时产生比较大的影响。

因此，我们对 Envoy 的内部的处理逻辑作了一些改进，增加了执行非关键任务的异步任务线程。然后将遥测的逻辑拆分出来放到异步线程中去执行，从而解决了主处理流程被阻塞的问题，进而很大程度上降低了Envoy转发消息的延时。

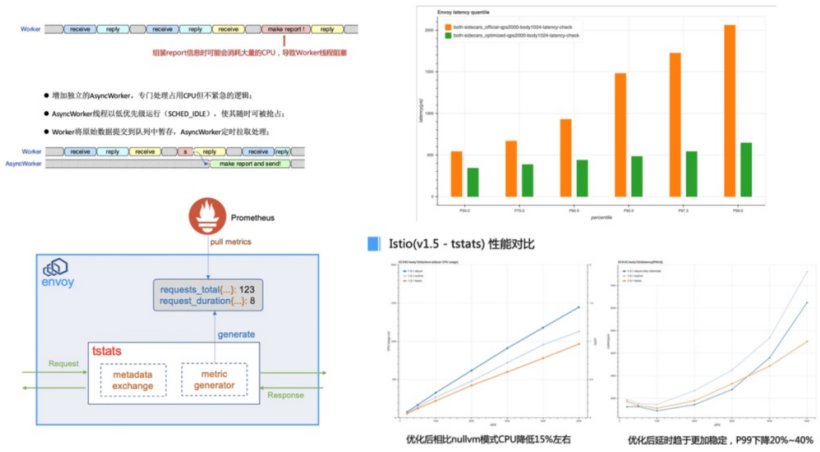
柱状图中，是我们在 2000 qps 下的一个测试结果，对比了官方 Envoy 和优化后的结果，可以看出 Envoy 整体的延时降低了 40% 以上，其实是一个很可观的数据。

基于这个优化，使仍然在使用低于 isito1.5 版本的 mesh 产品的用户，其数据面延时性能有了一个很大的提升。

在 istio1.5 版本开始，社区虽然对遥测功能做了较大的优化，Envoy 中移除了老版本中 Mixer 上报遥测数据的部分，取而代之的是通过遥测系统主动拉的方式上报遥测数据。除去了 Envoy 主动上报的逻辑，一定程度上，优化了 Envoy 的性能。

但遥测数据的收集，isito 是借助了 Envoy 提供的 Wasm 扩展机制实现的，由于 Wasm 扩展机制跟 Envoy 主体是通过内存拷贝的方式进行数据传递，Envoy 内部会进行大量的属性数据的拷贝，数据拷贝是一个 CPU 密集型的操作，即会占用了大量的 CPU 时间，体现在通信上，实际上就会增加延时。

因此，我们基于 Envoy 原生的方式开发了一个新的遥测插件 tstats，tstats 是通过有引用的方式来传递数据，所以避免了大量的内存拷贝，从而有效的降低了通信延时。



由上图中可以看出优化前后的数据对比。在不同的 qps 下，即使是 p99 延时，在官方的基础上也下降了 20% 到 40%，同时 CPU 也有明显的下降。通过延时和资源占用的优化，实际上也给用户带来了直接的经济效益。

Q&A

Q：istio 可以实现哪些场景，能分享下吗？现在腾讯云有哪些业务在使用 istio？

A： 比如腾讯的欢乐工作室，欢乐斗地主、欢乐麻将，游戏是对延时非常敏感的业务。还有腾讯文档，环境治理，多人协作等等。

Q：对于服务之间通信，服务网格如何保证数据安全呢？

A： 安全是服务网格提供的重要能力，在服务网格中，安全主要由两个方面：第一是认证，通常是指 mtls，服务端和客户端通信时需要互相校验身份。第二是鉴权，也就是服务端除了要知道访问者是谁外，还要确定访问者有什么操作权限，在 istio 中可以用 AuthorizationPolicy 来定义。

Q：Service mesh 内部的网络用的什么技术？

A： 如果指的是 istio 话，istio 是谷歌主导的，控制面交互协议主要是基于 gRPC，数据面的化是和用户业务相关，理论上用什么协议都可以。不过目前 istio 在数据面主要支持的协议还只是 http，gRPC。

Q：1.7版本只支持 K8S 1.16 以上，那历史使用 istio 的服务升级成本就一定很高吗？

A： 官方不会对 1.16 之前的版本进行测试，也不做可用保证，但是不一定是跑不起来的。我们之前也有支持老版本 kubernetes 的需求，整体看还是有些麻烦的，有一定的改动成本。

Q：链路追踪，istio 没法做到无埋点吧，也就是说业务服务并不能知道如何区分请求，这个如何处理呢？

A： istio 链路跟踪的确并不是完全透明的，如果用传统的 SDK 模式实现链路跟踪的确是完全透明的，因为在 SDK 里可以串联起来进程里的上下游 RPC 关系，但是在 mesh 中，应用和 sidecar 虽然是部署到一个托管里面，但是 sidecar 并不知道不同请求之间的关系，所以在 mesh 中 上下游关系需要用户在业务代码中维护，具体地就是需要业务传递少量的 header 信息，包括 trace id，parent id 等。

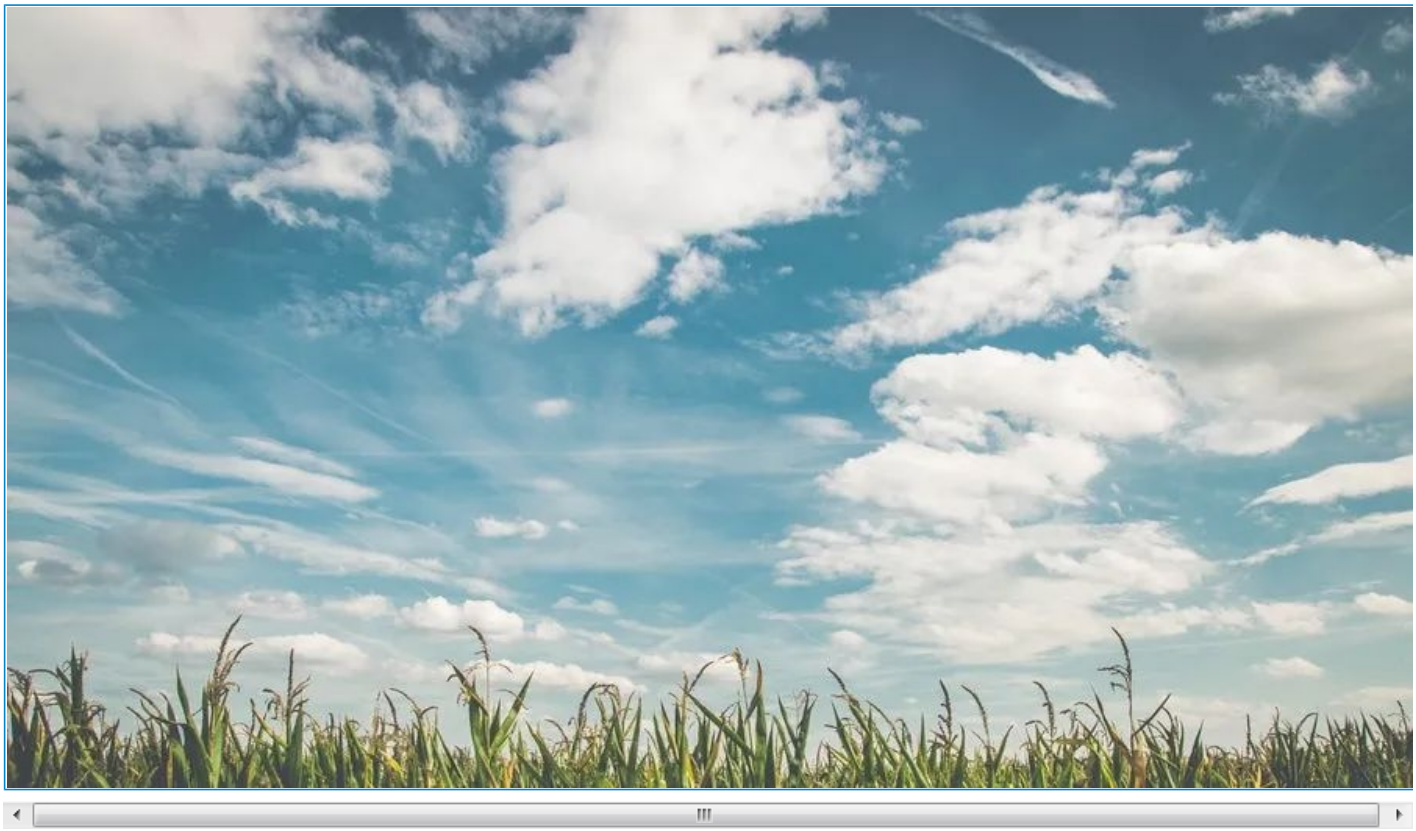
Q：这套托管方案会合并到社区吗？

A： 社区的发展方向和云上的托管诉求还是不太一样，不能直接使用，不过我们会将其中的一些通用的解决方案贡献到社区。

Q：现在在用的非托管 TCM，后面切换到托管的 TCM，是平滑的吗？

A： 未来是可以实现的。

文章推荐



降本提效，贝壳搜索推荐架构统一之路

— 关注云加社区 —

看腾讯技术，学云计算知识