ALIBABA CLOUD

阿里云

服务网格

可观测性

文档版本：20220114

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
Courier字体	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.在ASM中实现分布式跟踪

服务网格ASM集成了阿里云链路追踪服务Tracing Analysis，为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等能力，可以帮助开发者快速分析和诊断分布式应用架构下的性能瓶颈，提升开发诊断效率。本文介绍如何在ASM中使用链路追踪服务。

前提条件

- 已开通链路追踪服务。
- 创建ASM实例时，启用了链路追踪，详情参见创建ASM实例。

背景信息

分布式跟踪是一种用于对应用程序进行概要分析和监视的方法，尤其是针对使用微服务架构构建的应用程序。虽然Istio代理能够自动发送Span信息，但是应用程序仍然需要传播适当的HTTP标头，以便在代理发送Span时，可以将Span正确地关联到单个跟踪中。为此，应用程序需要收集以下标头并将其从传入请求传播到任何传出请求：

- `x-request-id`
- `x-b3-traceid`
- `x-b3-spanid`
- `x-b3-parentspanid`
- `x-b3-sampled`
- `x-b3-flags`
- `x-ot-span-context`

部署示例

在按照部署应用到ASM实例进行应用部署之后，查看示例中以Python语言实现的productpage服务，则会发现该应用程序使用了OpenTracing库从HTTP请求中提取了所需的标头。

```
def getForwardHeaders(request):
    headers = {}
    # x-b3-*** headers can be populated using the opentracing span
    span = get_current_span()
    carrier = {}
    tracer.inject(
        span_context=span.context,
        format=Format.HTTP_HEADERS,
        carrier=carrier)
    headers.update(carrier)
    # ...
    incoming_headers = ['x-request-id']
    # ...
    for ihdr in incoming_headers:
        val = request.headers.get(ihdr)
        if val is not None:
            headers[ihdr] = val
    return headers
```

查看以Java语言实现的reviews服务是也可以看到相应的HTTP标头。

```
@GET
@Path("/reviews/{productId}")
public Response bookReviewsById(@PathParam("productId") int productId,
                            @HeaderParam("end-user") String user,
                            @HeaderParam("x-request-id") String xreq,
                            @HeaderParam("x-b3-traceid") String xtraceid,
                            @HeaderParam("x-b3-spanid") String xspanid,
                            @HeaderParam("x-b3-parentspanid") String xparentspanid,
                            @HeaderParam("x-b3-sampled") String xsampled,
                            @HeaderParam("x-b3-flags") String xflags,
                            @HeaderParam("x-ot-span-context") String xotspan) {
  if (ratings_enabled) {
    JsonObject ratingsResponse = getRatings(Integer.toString(productId), user, xreq, xtrace
id, xspanid, xparentspanid, xsampled, xflags, xotspan);
```

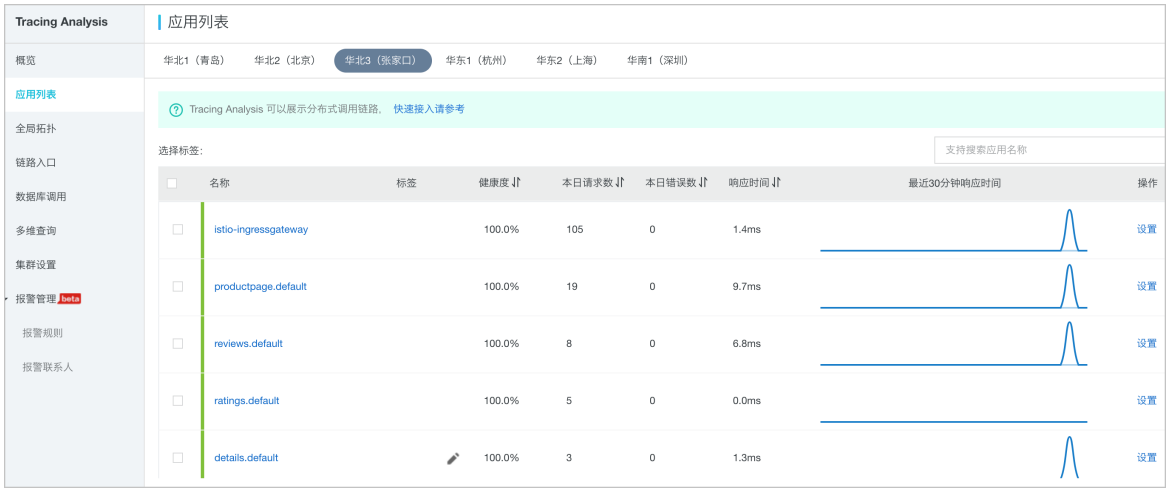
访问示例

在浏览器地址栏输入*http://{入口网关服务的IP地址}/productpage*，可以看到Bookinfo应用的页面。

查看应用列表

应用列表页面展示了所有被监控应用的健康度得分、本日请求数、本日错误数等关键指标。您还可以为应用设置自定义标签，从而通过标签进行筛选。

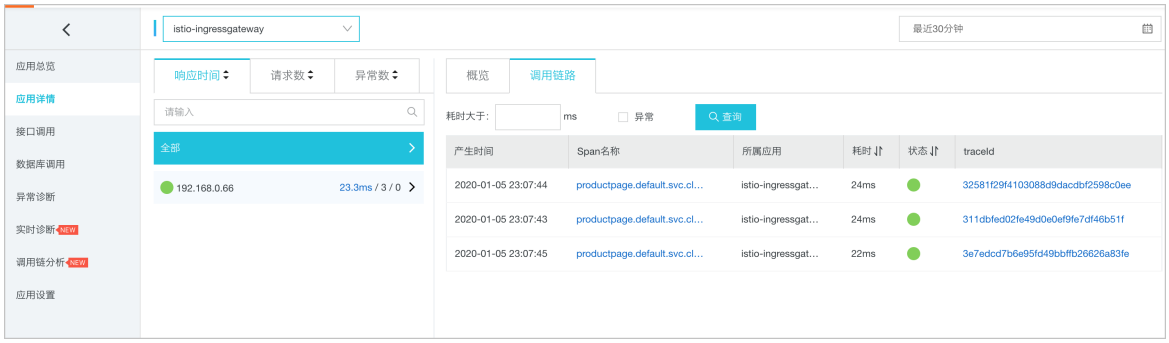
1. 登录链路追踪 Tracing Analysis 控制台
2. 在左侧导航栏中选择**应用列表**，并在应用列表页面顶部选择目标地域。



查看应用详情

应用详情页面可展示应用在所部属的每一台机器上的关键性能指标、调用拓扑图和调用链路。

1. 登录链路追踪 Tracing Analysis 控制台
2. 在左侧导航栏中选择**应用列表**，并在应用列表页面顶部选择目标地域，然后单击对应的应用名称。
3. 在左侧导航栏中选择**应用详情**，在左侧的机器列表中单击**全部**或一台以IP地址标识的机器。
   在**概览**页面上查看调用拓扑图和关键性能指标。在**调用链路**页面上查看该应用在所选机器上的调用链路列表，按耗时降序排列，最多可列出100个调用链路。

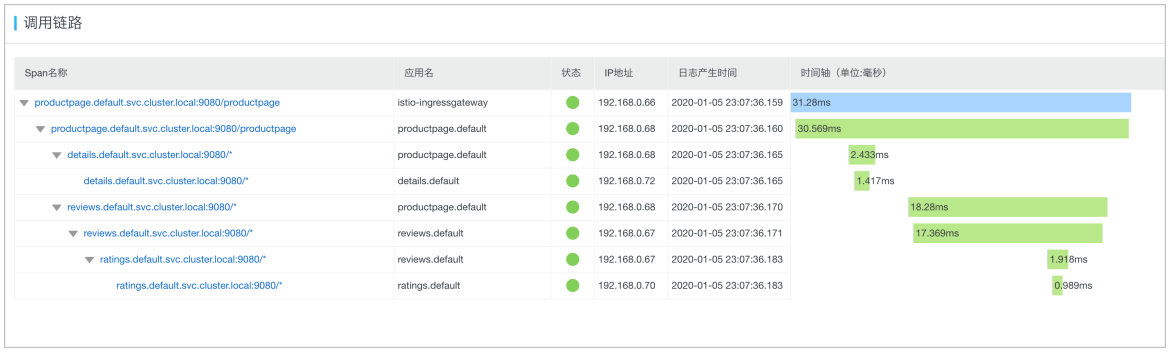


查看调用链瀑布图

调用链路的瀑布图展示了调用链路的日志产生时间、状态、IP址/机器名称、服务名、时间轴等信息。

1. 在应用详情页面，选择**调用链路**页签，单击指定链路的Trace ID。
2. 在新弹出的**调用链路**页面上查看该调用链路的瀑布图。

> **说明**
>
> - **IP地址**字段显示的是IP地址还是机器名称，取决于**应用设置**页面上的显示配置。详情请参见管理应用和标签。
> - 将鼠标悬浮于服务名上，还可以查看该服务的时长、开始时间、Tag和日志事件等信息。详情参见查看应用详情。



2.向自建系统导出ASM链路追踪数据

ASM不仅支持向阿里云链路追踪服务导出链路追踪数据，同样也支持向您自建的兼容Zipkin协议的系统导出追踪数据，本文介绍如何对ASM进行配置，将追踪数据导出至您的自建系统。

前提条件

- 该自建系统支持标准Zipkin协议，并通过标准Zipkin端口9411监听。若您使用Jaeger，则需要部署Zipkin Collector。
- 该自建系统部署于数据面集群内。
- 已创建一个ACK实例并添加到ASM实例中，请参见添加集群到ASM实例。
- ASM实例部署了入口网关，请参见添加入口网关服务。

步骤一：为集群启用链路追踪

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 单击页面右上角的**功能设置**。
5. 在**功能设置更新**的面板中，选中**启用链路追踪**，并选择**自行搭建Zipkin**。
6. 单击**确定**。

步骤二：在数据面集群部署Zipkin

1. 将下面的YAML保存为文件zipkin-server.yaml。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zipkin-server
  namespace: istio-system
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zipkin-server
      component: zipkin
  template:
   metadata:
    labels:
      app: zipkin-server
      component: zipkin
   spec:
    containers:
     - name: zipkin-server
       image: openzipkin/zipkin
       imagePullPolicy: IfNotPresent
       readinessProbe:
            httpGet:
              path: /health
              port: 9411
            initialDelaySeconds: 5
            periodSeconds: 5
```

说明 如果您需要使用自行准备的追踪系统YAML文件部署，请确保Deployment处在istio-system namespace下。

2. 执行以下命令，将该配置应用到数据面集群。

```
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f zipkin-server.yaml
```

注意 本文中的${DATA_PLANE_KUBECONFIG}应当替换为数据面集群的Kubeconfig文件路径，${ASM_KUBECONFIG}应当替换为网格实例的Kubeconfig文件路径。

3. 部署完毕后确认ZipkinServer Pod正常启动。

步骤三：创建Service暴露ZipkinServer

您需要在istio-system namespace下创建名为zipkin的服务，来接收ASM的链路追踪信息或访问该服务。若需要将Zipkin暴露于公网，请使用zipkin-svc-expose-public.yaml；若不希望暴露于公网，请使用zipkin-svc.yaml。为了便于查看追踪数据，本示例使用zipkin-svc-expose-public.yaml将Zipkin Server暴露于公网端口。

注意 创建的服务名称必须为zipkin。

1. 将下面的内容保存为YAML文件。
   - 若需要将Zipkin暴露于公网，请使用zipkin-svc-expose-public.yaml，文件内容如下：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: tracing
    component: zipkin
  name: zipkin
  namespace: istio-system
spec:
  externalTrafficPolicy: Cluster
  ports:
  - name: zipkin
    port: 9411
    protocol: TCP
    targetPort: 9411
  selector:
    app: zipkin-server
    component: zipkin
  type: LoadBalancer
```

- 若不希望暴露于公网，请使用zipkin-svc.yaml，文件内容如下：

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: tracing
    component: zipkin
  name: zipkin
  namespace: istio-system
spec:
  externalTrafficPolicy: Cluster
  ports:
  - name: zipkin
    port: 9411
    protocol: TCP
    targetPort: 9411
  selector:
    app: zipkin-server
    component: zipkin
  type: ClusterIP
```

> **说明** 如果您需要使用自行准备的YAML文件部署Service，请确保Service处在istio-system namespace下。

2. 执行以下命令将Zipkin Service应用到数据面集群。

```
#部署内网zipkin。
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f zipkin-svc.yaml
#部署公网可以访问的zipkin。
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f zipkin-svc-expose-public.yaml
```

步骤四：部署测试应用BookInfo

1. 通过Kubectl执行以下命令，将Bookinfo应用部署到数据面集群中。

```
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f bookinfo.yaml
```

bookinfo.yaml文件内容如下：

```
apiVersion: v1
kind: Service
metadata:
  name: details
  labels:
    app: details
    service: details
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: details
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-details
  labels:
    account: details
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: details-v1
  labels:
    app: details
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: details
      version: v1
  template:
    metadata:
      labels:
        app: details
        version: v1
    spec:
      serviceAccountName: bookinfo-details
      containers:
      - name: details
        image: docker.io/istio/examples-bookinfo-details-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
---
##################################################################################################
```

```
##################################################################################################
############
# Ratings service
##################################################################################################
############
apiVersion: v1
kind: Service
metadata:
  name: ratings
  labels:
    app: ratings
    service: ratings
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: ratings
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-ratings
  labels:
    account: ratings
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ratings-v1
  labels:
    app: ratings
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ratings
      version: v1
  template:
    metadata:
      labels:
        app: ratings
        version: v1
    spec:
      serviceAccountName: bookinfo-ratings
      containers:
      - name: ratings
        image: docker.io/istio/examples-bookinfo-ratings-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
---
##################################################################################################
############
```

```
# Reviews service
##################################################################################################
###########
apiVersion: v1
kind: Service
metadata:
  name: reviews
  labels:
    app: reviews
    service: reviews
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: reviews
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-reviews
  labels:
    account: reviews
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v1
  labels:
    app: reviews
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v1
  template:
    metadata:
      labels:
        app: reviews
        version: v1
    spec:
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v1:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
```

```
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v2
  labels:
    app: reviews
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v2
  template:
    metadata:
      labels:
        app: reviews
        version: v2
    spec:
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v2:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: reviews-v3
```

```
  labels:
    app: reviews
    version: v3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: reviews
      version: v3
  template:
    metadata:
      labels:
        app: reviews
        version: v3
    spec:
      serviceAccountName: bookinfo-reviews
      containers:
      - name: reviews
        image: docker.io/istio/examples-bookinfo-reviews-v3:1.16.2
        imagePullPolicy: IfNotPresent
        env:
        - name: LOG_DIR
          value: "/tmp/logs"
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
        - name: wlp-output
          mountPath: /opt/ibm/wlp/output
      volumes:
      - name: wlp-output
        emptyDir: {}
      - name: tmp
        emptyDir: {}
---
##################################################################################################
############
# Productpage services
##################################################################################################
############
apiVersion: v1
kind: Service
metadata:
  name: productpage
  labels:
    app: productpage
    service: productpage
spec:
  ports:
  - port: 9080
    name: http
  selector:
    app: productpage
```

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: bookinfo-productpage
  labels:
    account: productpage
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: productpage-v1
  labels:
    app: productpage
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: productpage
      version: v1
  template:
    metadata:
      labels:
        app: productpage
        version: v1
    spec:
      serviceAccountName: bookinfo-productpage
      containers:
      - name: productpage
        image: docker.io/istio/examples-bookinfo-productpage-v1:1.16.2
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        volumeMounts:
        - name: tmp
          mountPath: /tmp
      volumes:
      - name: tmp
        emptyDir: {}
---
```

2. 通过Kubectl执行以下命令，部署Bookinfo应用的VirtualServices。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f virtual-service-all-v1.yaml
```

virtual-service-all-v1.yaml文件内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: productpage
spec:
  hosts:
  - productpage
  http:
  - route:
    - destination:
        host: productpage
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - route:
    - destination:
        host: ratings
        subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: details
spec:
  hosts:
  - details
  http:
  - route:
    - destination:
        host: details
        subset: v1
---
```

3. 通过Kubectl执行以下命令，部署Bookinfo应用的DestinationRules。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f destination-rule-all.yaml
```

destination-rule-all.yaml文件内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: productpage
spec:
  host: productpage
  subsets:
  - name: v1
    labels:
      version: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: reviews
spec:
  host: reviews
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: ratings
spec:
  host: ratings
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v2-mysql
    labels:
      version: v2-mysql
  - name: v2-mysql-vm
    labels:
      version: v2-mysql-vm
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
```

```
metadata:
  name: details
spec:
  host: details
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
---
```

4. 通过Kubectl执行以下命令，部署Bookinfo应用的Gateway。

```
kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f bookinfo-gateway.yaml
```

bookinfo-gateway.yaml文件内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  selector:
    istio: ingressgateway # use istio default controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
  - "*"
  gateways:
  - bookinfo-gateway
  http:
  - match:
    - uri:
        exact: /productpage
    - uri:
        prefix: /static
    - uri:
        exact: /login
    - uri:
        exact: /logout
    - uri:
        prefix: /api/v1/products
    route:
    - destination:
        host: productpage
        port:
          number: 9080
```

步骤五：产生追踪数据

1. 执行以下命令，获得入口网关地址。

```
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} get svc -n istio-system|grep ingressgateway|awk -F ' ' '{print $4}'
```

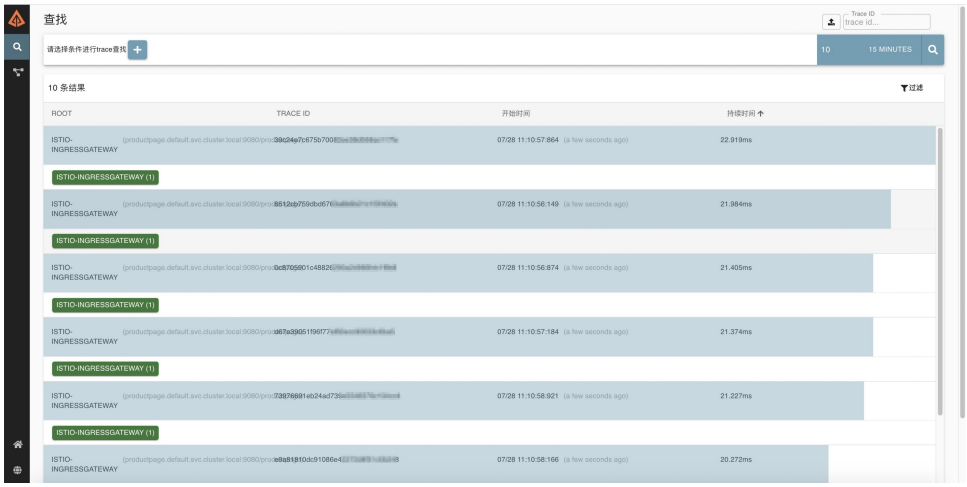
2. 使用地址 `入口网关地址/productpage` 访问Bookinfo应用。

步骤六：查看链路追踪数据

1. 执行下列命令获取Zipkin Service地址。

```
kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} get svc -n istio-system|grep zipkin|awk -F ' ' '{print $4}'
```

2. 使用 `Zipkin Service地址:9411` ，访问Zipkin控制台，查看追踪数据。



3.使用日志服务采集数据平面入口网关日志

容器服务ACK（Alibaba Cloud Container Service for Kubernetes）集成了日志服务功能，您可在创建集群时启用日志服务，采集服务网格数据平面集群入口网关的访问日志。本文主要介绍如何开启日志采集、配置日志服务以及查看采集的日志。

前提条件

- 已创建至少一个ASM实例，请参见创建ASM实例。
- 已添加至少一个ACK集群至ASM实例，请参见添加集群到ASM实例。
- 已部署至少一个入口网关到ACK集群，请参见添加入口网关服务。

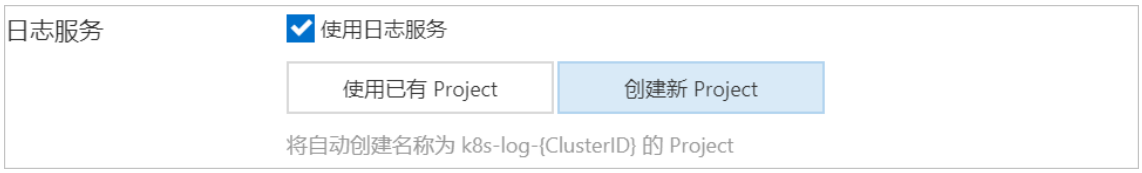
步骤一：为Kubernetes集群安装日志服务组件

如果您尚未创建Kubernetes集群，请执行以下步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面，单击右上角**创建Kubernetes集群**，具体步骤，请参见快速创建Kubernetes托管版集群。
4. 在**组件配置**步骤中，选择**日志服务**配置项，表示在新建的Kubernetes集群中安装日志插件。
   - 选择**使用已有Project**，选择一个现有的Project来管理采集的日志。

   - 选择**创建新Project**，则自动创建一个新的Project来管理采集的日志，Project会自动命名为k8s-log-{ClusterID}，ClusterID表示您新建的Kubernetes集群的唯一标识。

5. 完成配置后，单击**创建集群**，完成集群创建。

如果您已创建了Kubernetes集群，但未安装日志组件，请执行以下步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**运维管理 > 组件管理**。
5. 在**日志与监控**列表中找到**logtail-ds**，单击**安装**。
6. 在**提示**对话框单击**确定**。

如果您已经创建了Kubernetes集群，并且已为集群安装了日志组件，但版本低于v0.16.24.0-1fa7551-aliyun，请执行以下步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**运维管理 > 组件管理**。
5. 在**日志与监控**列表中找到**logtail-ds**，单击右侧的**升级**。
6. 在**提示**对话框单击**确定**。

步骤二：配置日志服务

使用logtail组件采集入口网关日志，需要创建采集配置。

如果您的ASM为v1.7.5.26-gd318a562-aliyun及以上版本，需要按照以下步骤操作

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面单击右上角的**功能设置**。
5. 在**功能设置更新**面板中选中**启用访问日志查询**，然后单击**确定**。

如果您的ASM版本低于v1.7.5.26-gd318a562-aliyun，需要按照以下步骤操作

1. 创建YAML文件，文件模板如下。

> **说明** 您需要为每个数据平面集群准备配置文件。

```
apiVersion: log.alibabacloud.com/v1alpha1
kind: AliyunLogConfig
metadata:
  # your config name, must be unique in you k8s cluster
  name: mesh-ingress-log-config
  namespace: kube-system
spec:
  project: k8s-log-${K8SClusterId}
  # logstore name to upload log
  logstore: mesh-ingress-log
  # product code, you should not change it
  productCode: k8s-istio-ingress
  # logtail config detail
  logtailConfig:
    inputType: plugin
    configName: mesh-ingress-log-config
    inputDetail:
      plugin:
        inputs:
        - detail:
            IncludeLabel:
              io.kubernetes.pod.name: ^istio-ingressgateway-.*$
            Stderr: false
            Stdout: true
          type: service_docker_stdout
        processors:
        - detail:
```

```
        Anchors:
        - FieldName: log
          FieldType: json
        KeepSource: true
        NoKeyError: true
        NoMatchError: true
        SourceKey: content
      type: processor_anchor
    - type: processor_rename
      detail:
        DestKeys:
        - host
        - request_length
        - body_bytes_sent
        - request_time
        - method
        - url
        - version
        - req_id
        - status
        - proxy_upstream_name
        - upstream_addr
        - upstream_response_time
        - http_user_agent
        - x_forward_for
        SourceKeys:
        - log_authority
        - log_bytes_received
        - log_bytes_sent
        - log_duration
        - log_method
        - log_path
        - log_protocol
        - log_request_id
        - log_response_code
        - log_upstream_cluster
        - log_upstream_host
        - log_upstream_service_time
        - log_user_agent
        - log_x_forwarded_for
```

2. 替换模板中的*${K8SClusterID}*为您的Kubernetes集群ClusterID。
3. （可选）若您存在未经过服务网格控制台创建的入口网关，且您希望采集该入口网关的日志，则您需要按照以下格式修改模板中*io.kubernetes.pod.name*的参数值。

```
^(^istio-ingressgateway-.*$)|(^[Name-of-your-customized-ingressgateway]-.*$)$
```

例如Ingressgateway deployment名称为my-ingressgateway，则您需要按照以下格式修改模板中*io.kubernetes.pod.name*的参数值。

```
^(^istio-ingressgateway-.*$)|(^my-ingressgateway-.*$)$
```

4. 连接到Kubernetes集群，详细描述请参见通过kubectl工具连接集群或通过SSH连接ACK专有版集群的Master节点。

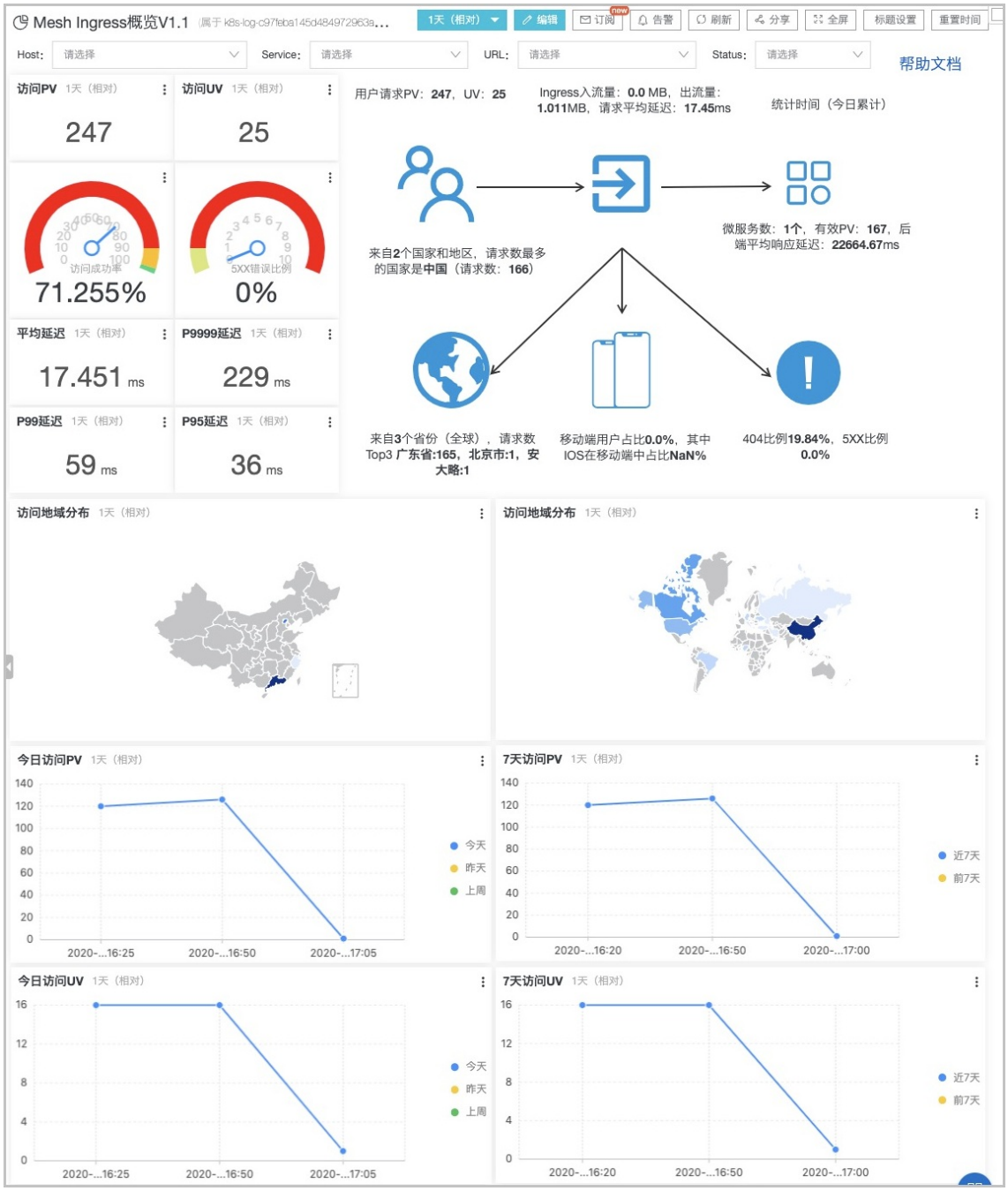
5. 在Kubernetes集群应用已完成的YAML文件。

```
kubectl apply -f [yaml文件路径]
```

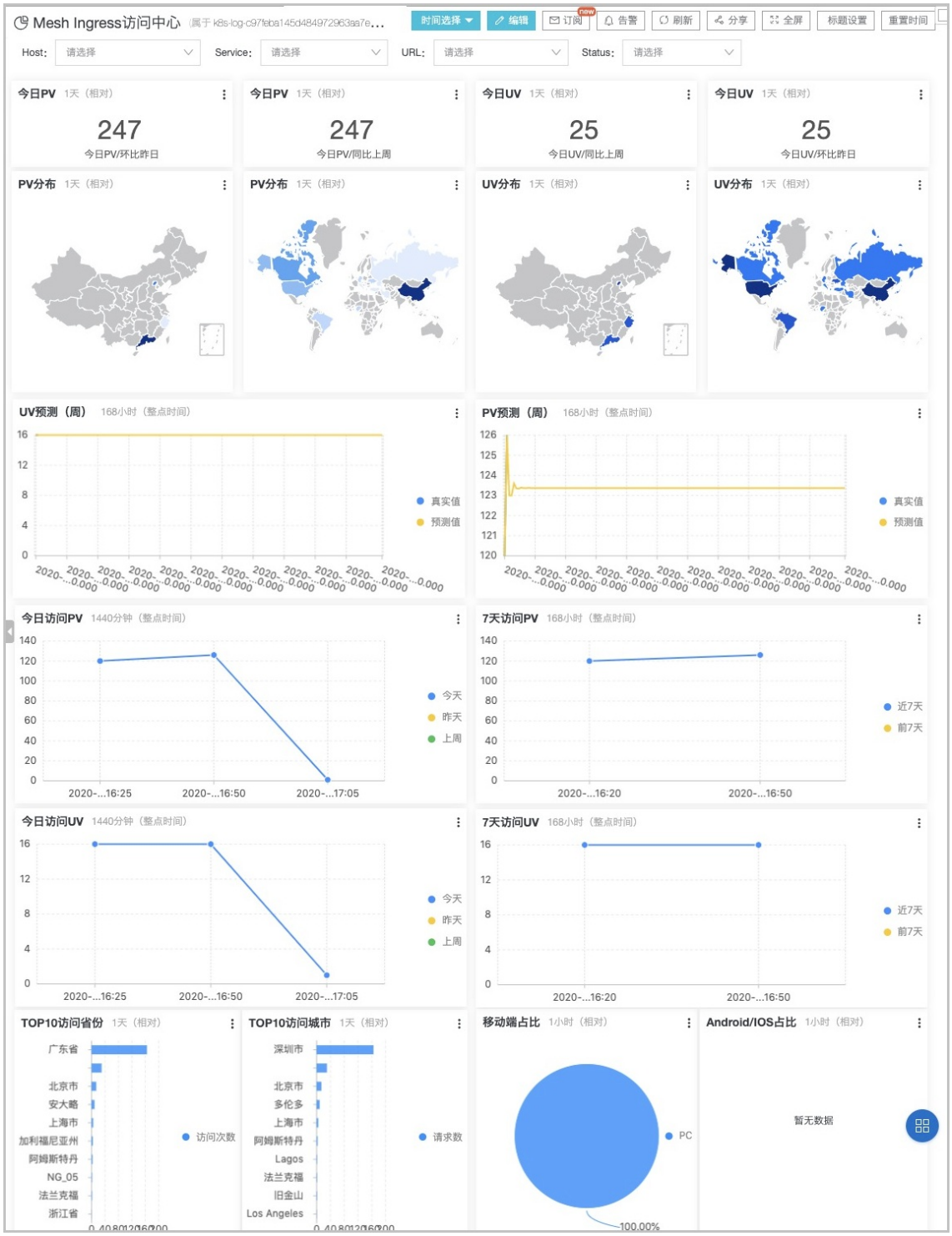
步骤三：查看日志

完成配置后，将采集数据平面入口网关日志并存储到日志服务指定的LogProject和LogStore，您可以通过以下步骤查看日志。

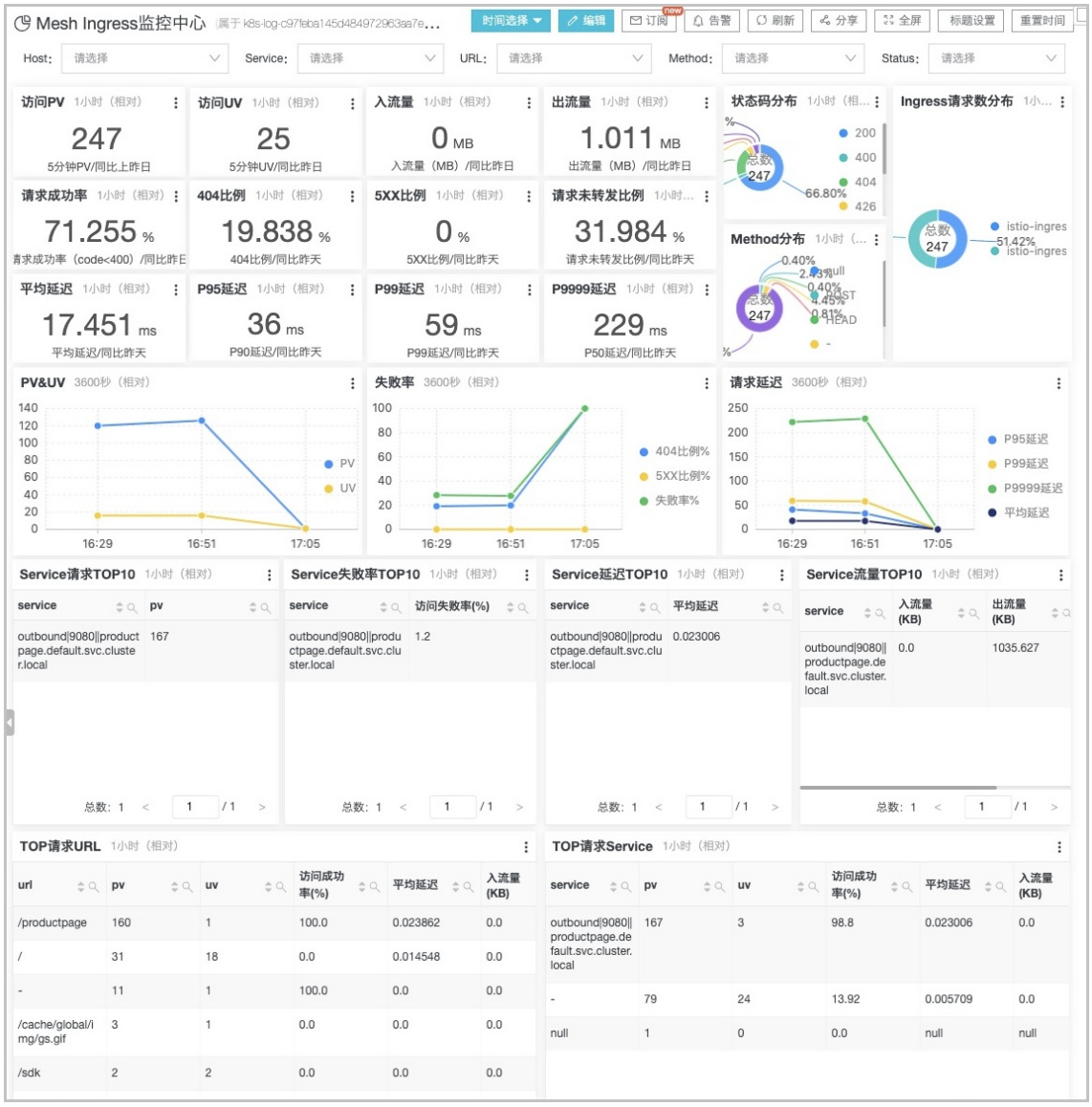
1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**数据平面（服务发现） > Kubernetes集群**。
5. 在**Kubernetes集群**页面单击目标集群右侧**可观测性**列下的**查看报表**，可选择查看**入口网关概览**、**入口网关访问中心**、**入口网关监控中心**。
   - 选择**入口网关概览**，进入入口网关概览界面。该界面展示了网格入口的统计信息概览，包括访问地理、PV/UV、延迟、成功率等。



- 选择**入口网关访问中心**，进入入口网关访问中心界面。该界面详细展示了PV/UV、地理位置、访问来源设备等统计信息，对判断用户分布、行为有参考意义。



- 选择**入口网关监控中心**，进入入口网关监控中心界面。该界面展示了成功率、请求状态码、延迟等信息，对判断当前服务状态有重要参考意义。


Mesh Ingress监控中心
时间选择
编辑
订阅
告警
刷新
分享
全屏
标题设置
重置时间
Host:
Service:
URL:
Method:
Status:
访问PV 1小时（相对）
247
5分钟PV/同比上昨日
访问UV 1小时（相对）
25
5分钟UV/同比昨日
入流量 1小时（相对）
0 MB
入流量（MB）/同比昨日
出流量 1小时（相对）
1.011 MB
出流量（MB）/同比昨日
状态码分布 1小时（相...
200
400
404
426
66.80%
Ingress请求数分布 1小...
istio-ingres
51.42%
请求成功率 1小时（相对）
71.255 %
404比例 1小时（相对）
19.838 %
404比例/同比昨天
5XX比例 1小时（相对）
0 %
5XX比例/同比昨天
请求未转发比例 1小时...
31.984 %
请求未转发比例/同比昨天
Method分布 1小时（...
0.40%
GET
HEAD
平均延迟 1小时（相对）
17.451 ms
平均延迟/同比昨天
P95延迟 1小时（相对）
36 ms
P90延迟/同比昨天
P99延迟 1小时（相对）
59 ms
P99延迟/同比昨天
P9999延迟 1小时（相对）
229 ms
P50延迟/同比昨天
PV&UV 3600秒（相对）
PV
UV
16:29
16:51
17:05
失败率 3600秒（相对）
404比例%
5XX比例%
失败率%
请求延迟 3600秒（相对）
P95延迟
P99延迟
P9999延迟
平均延迟
Service请求TOP10 1小时（相对）
service
pv
outbound|9080||productpage.default.svc.cluster.local
167
Service失败率TOP10 1小时（相对）
访问失败率(%)
1.2
Service延迟TOP10 1小时（相对）
平均延迟
0.023006
Service流量TOP10 1小时（相对）
入流量(KB)
出流量(KB)
0.0
1035.627
TOP请求URL 1小时（相对）
url
uv
访问成功率(%)
入流量(KB)
/productpage
160
1
100.0
0.023862
0.0
/
31
18
0.014548
-
11
100.0
/cache/global/img/gs.gif
3
/sdk
2
TOP请求Service 1小时（相对）
79
24
13.92
0.005709
null
1
0

4.自定义数据面访问日志

部署在数据平面（即加入网格的Kubernetes集群）的Envoy Proxy可以输出所有访问日志，ASM支持自定义Envoy Proxy输出的访问日志内容。本文介绍如何自定义Envoy Proxy输出的访问日志内容。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署入口网关服务。具体操作，请参见添加入口网关服务。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。

步骤一：启用访问日志

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格信息页面单击**功能设置**。
5. 在**功能设置更新**面板选中**启用访问日志**，然后单击**确定**。

   启用访问日志（默认是开启状态），istio-proxy容器默认输出包含以下字段的日志，如果关闭访问日志，istio-proxy容器将不会产生JSON格式的访问日志。

```
"authority_for":"%REQ(:AUTHORITY)%",
"bytes_received":"%BYTES_RECEIVED%",
"bytes_sent":"%BYTES_SENT%",
"downstream_local_address":"%DOWNSTREAM_LOCAL_ADDRESS%",
"downstream_remote_address":"%DOWNSTREAM_REMOTE_ADDRESS%",
"duration":"%DURATION%",
"istio_policy_status":"%DYNAMIC_METADATA(istio.mixer:status)%",
"method":"%REQ(:METHOD)%",
"path":"%REQ(X-ENVOY-ORIGINAL-PATH?:PATH)%",
"protocol":"%PROTOCOL%",
"request_id":"%REQ(X-REQUEST-ID)%",
"requested_server_name":"%REQUESTED_SERVER_NAME%",
"response_code":"%RESPONSE_CODE%",
"response_flags":"%RESPONSE_FLAGS%",
"route_name":"%ROUTE_NAME%",
"start_time":"%START_TIME%",
"trace_id":"%REQ(X-B3-TRACEID)%",
"upstream_cluster":"%UPSTREAM_CLUSTER%",
"upstream_host":"%UPSTREAM_HOST%",
"upstream_local_address":"%UPSTREAM_LOCAL_ADDRESS%",
"upstream_service_time":"%RESP(X-ENVOY-UPSTREAM-SERVICE-TIME)%",
"upstream_transport_failure_reason":"%UPSTREAM_TRANSPORT_FAILURE_REASON%",
"user_agent":"%REQ(USER-AGENT)%",
"x_forwarded_for":"%REQ(X-FORWARDED-FOR)%"
```

步骤二：自定义数据面访问日志内容

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格信息页面单击**启用访问日志**右侧的**自定义访问日志格式**。
5. 在**自定义访问日志格式**对话框中设置**变量名称**为my_custom_key，**变量值**为%REQ(end-user)%，然后单击**确认**。

   本文以获取Bookinfo示例中HTTP请求的Header字段end-user为例

步骤三：查看访问日志

启用访问日志后，发起请求的Sidecar容器会按照自定义的访问日志格式输出访问日志。

1. 在浏览器地址栏输入*入口网关地址：productpage*，请求Productpage应用。
2. 登录容器服务管理控制台。
3. 在控制台左侧导航栏中，单击**集群**。
4. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
5. 在集群管理页左侧导航栏中，选择**工作负载 > 无状态**。
6. 在**无状态**页面顶部设置**命名空间**为default，然后单击productpage-v1应用右侧**操作**列下的**详情**。
7. 在应用详情页面单击**日志**页签，设置**Container**为istio-proxy。

   在日志输出框中可以看到如下日志。

{"method":"GET","x_forwarded_for":null,"upstream_host":"172.19.16.90:9080","protocol":"HTTP/1.1","my_custom_key":"jason","authority_for":"addedvalues:9080","response_code":200,"start_time":"2021-10-21T11:40:12.055Z","request_id":"5222b7fb-05a6-4fae-8e13-d44525ae3ca0","bytes_sent":883,"downstream_remote_address":"172.19.16.11:33752","upstream_transport_failure_reason":null,"downstream_local_address":"192.168.237.140:9080","requested_server_name":null,"response_flags":"-","duration":4,"user_agent":"python-requests/2.18.4","route_name":"default","trace_id":null,"istio_policy_status":null,"path":"/addedvalues/0","upstream_cluster":"outbound|9080||addedvalues.default.svc.cluster.local","bytes_received":0,"upstream_service_time":"3","authority":"addedvalues:9080","upstream_local_address":"172.19.16.11:52430"}

   可以看到日志中包含名为jason的end-user，说明自定义日志内容成功。

相关操作

您还可以使用日志服务采集数据平面的AccessLog。具体操作，请参见使用日志服务采集数据平面入口网关日志。

5.使用日志服务采集数据平面的AccessLog

容器服务ACK（Alibaba Cloud Container Service for Kubernetes）集成了日志服务功能，可对服务网格数据平面集群的AccessLog进行采集。本文介绍如何开启日志采集、配置日志服务以及查看采集的日志。

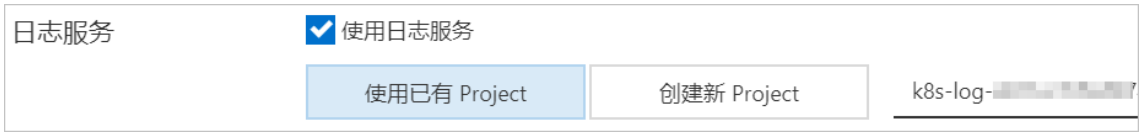
背景信息

部署在数据平面（即加入网格的Kubernetes集群）的Envoy Proxy可以输出所有访问日志，这些日志被称为Envoy Access Log，请参见Envoy Access Log。您可以通过**kubectl logs**指令查看这些日志。借助阿里云日志服务，不仅可以更便捷地查看日志，还可以对这些日志进行收集、检索或建立Dashboard。

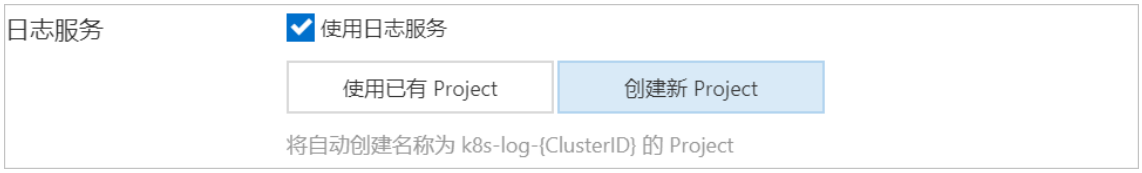
步骤一：为Kubernetes集群安装日志服务组件

如果您尚未创建Kubernetes集群，请执行以下步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面，单击右上角**创建Kubernetes集群**，具体步骤，请参见快速创建Kubernetes托管版集群。
4. 在**组件配置**步骤中，选择**日志服务**配置项，表示在新建的Kubernetes集群中安装日志插件。
   - 选择**使用已有Project**，选择一个现有的Project来管理采集的日志。

   - 选择**创建新Project**，则自动创建一个新的Project来管理采集的日志，Project会自动命名为k8s-log-{ClusterID}，ClusterID表示您新建的Kubernetes集群的唯一标识。

5. 完成配置后，单击**创建集群**，完成集群创建。

如果您已创建了Kubernetes集群，但未安装日志组件，请执行以下步骤：

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**运维管理 > 组件管理**。
5. 在**日志与监控**列表中找到**logtail-ds**，单击**安装**。
6. 在**提示**对话框单击**确定**。

如果您已经创建了Kubernetes集群，并且已为集群安装了日志组件，但版本低于v0.16.24.0-1fa7551-aliyun，请执行以下步骤：

1. 登录容器服务管理控制台。

2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，单击**运维管理 > 组件管理**。
5. 在**日志与监控**列表中找到**logtail-ds**，单击右侧的**升级**。
6. 在**提示**对话框单击**确定**。

步骤二：配置日志服务

使用Logtail组件采集Envoy Access Log，需要创建采集配置。

如果您的ASM为v1.7.5.26-gd318a562-aliyun及以上版本，需要按照以下步骤操作：

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面单击右上角的**功能设置**。
5. 在**功能设置更新**面板中选中**启用访问日志查询**，然后单击**确定**。

如果您的ASM版本低于v1.7.5.26-gd318a562-aliyun，需要按照以下步骤操作：

1. 准备YAML文件，文件模版如下。

```
apiVersion: log.alibabacloud.com/v1alpha1
kind: AliyunLogConfig
metadata:
  # your config name, must be unique in you k8s cluster
  name: mesh-access-log-config
  namespace: kube-system
spec:
  # must use same project with k8s cluster
  project: k8s-log-${K8SClusterID}
  # logstore name to upload log
  logstore: mesh-access-log
  # product code always been mesh-access-log
  productCode: mesh-access-log
  # logtail config detail
  logtailConfig:
    # docker stdout's input type is 'plugin'
    inputType: plugin
    # logtail config name, should be same with [metadata.name]
    configName: mesh-access-log-config
    inputDetail:
      plugin:
        inputs:
          - type: service_docker_stdout
            detail:
              # collect stdout and stderr
              Stdout: true
              Stderr: true
              IncludeEnv:
                ISTIO_META_POD_NAME: ""
              IncludeLabel:
                io.kubernetes.container.name: "istio-proxy"
        processors:
          - type: processor_json
            detail:
              # 从docker采集的数据默认key为"content"
              SourceKey: content
              ExpandConnector: ""
              KeepSource: true
              NoKeyError: true
```

说明

- 替换模板中的*${K8SClusterID}*为您的K8s集群ClusterID。
- 每个K8s集群都需要单独准备配置文件。
- YAML文件中的project字段请务必按照k8s-log-${K8SClusterID}规则填写，logstore字段务必保持原值，否则会影响服务网格控制面板的显示。

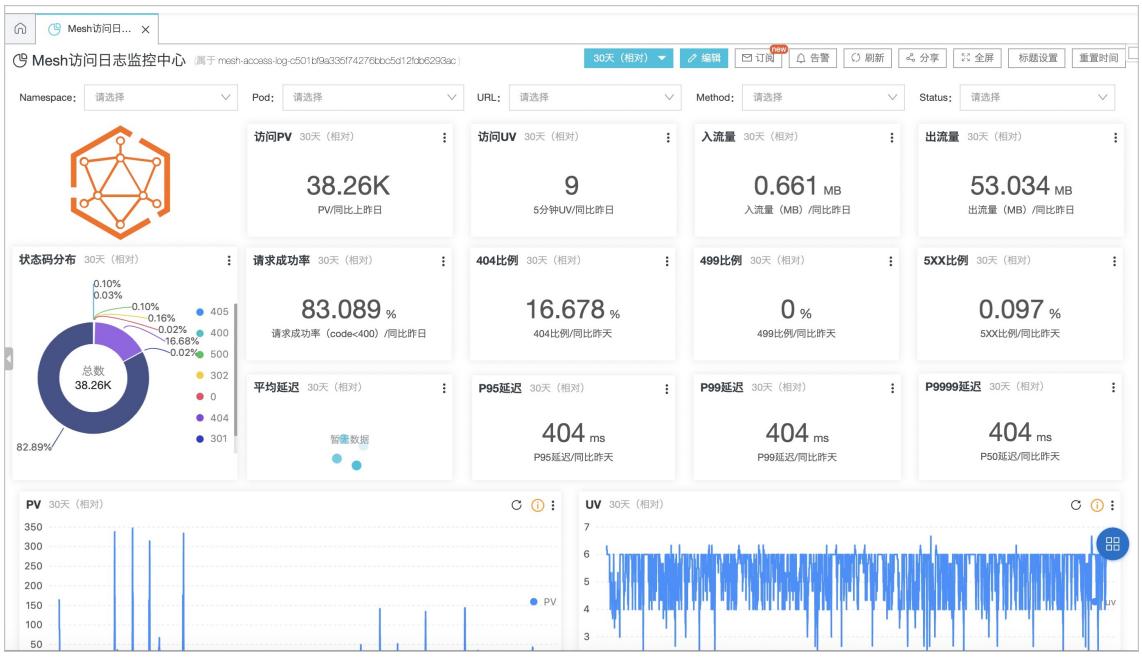
2. 在k8s集群应用在已编辑完成的YAML文件。

```
kubectl apply -f [YAML文件路径]
```

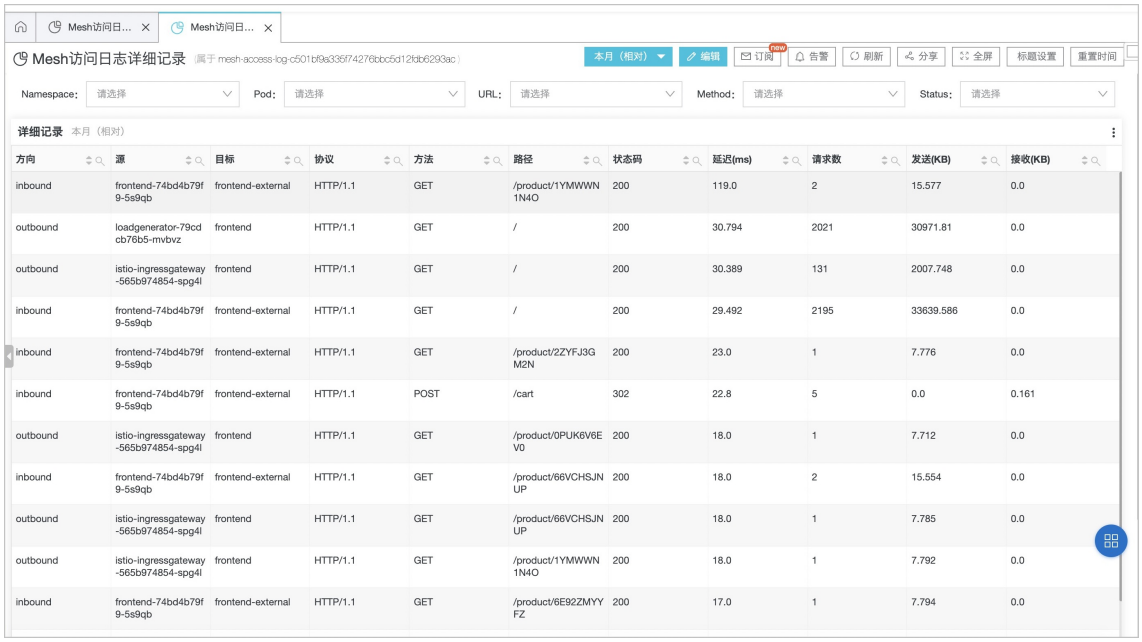
步骤三：查看日志

完成配置后，Envoy Access Log将被采集并存储到日志服务您指定的LogProject和LogStore中，您可以通过以下步骤来查看日志。

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到目标实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**数据平面（服务发现） > Kubernetes集群**。
5. 在**Kubernetes集群**页面，单击目标集群右侧**可观测性**列下的**查看报表**，可选择查看**访问日志详细记录**或**访问日志监控中心**。
   - **访问日志监控中心**则展示了多维度的日志统计数据。



   - **访问日志详细记录**以更为易读的方式展示原始访问日志。



6.集成ARMS Prometheus实现网格监控

服务网格ASM集成了ARMS Prometheus功能，可以实现对服务网格的监控。

前提条件

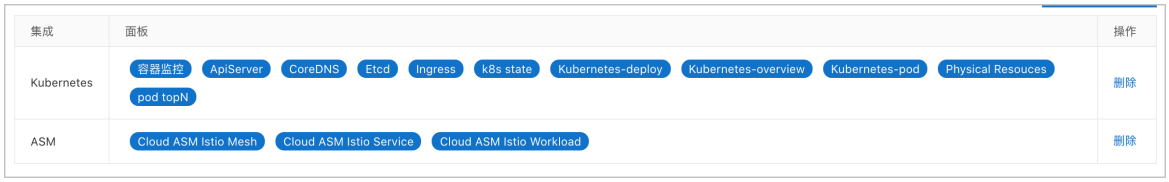
已在对应的ACK集群中安装Prometheus监控插件，详情请参见安装Prometheus监控插件。

背景信息

在开通ARMS之后，您可以在ARMS中为ACK集群一键安装Prometheus监控插件，此后即可通过ARMS预定义的仪表板监控Kubernetes集群的众多性能指标。

步骤一：在ARMS Prometheus中接入ASM

1. 登录ARMS控制台。
2. 在左侧导航栏选择**Prometheus监控 > Prometheus实例列表**。
3. 在页面左上角选择目标地域，然后单击Prometheus实例名称。
4. 在左侧导航栏中单击**Integration接入**。
5. 在Integration接入页面，单击**添加Integration**。
6. 在**Integration列表**对话框，单击**ASM**图标。
   在**ASM集成**页面可以看到用于服务发现的YAML内容。
7. 单击**确定**。
   可以看到Prometheus监控已集成了ASM。



步骤二：在ASM控制台上查看监控报表

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**数据平面（服务发现） > Kubernetes集群**。
5. 在**Kubernetes集群**页面，单击目标集群右侧**可观测性**列下的**查看报表**。
   您可以选择查看**网格服务统计**或**工作负载统计**。

步骤三：在ARMS控制台上查看Prometheus访问地址

1. 登录ARMS控制台。
2. 在页面左上角选择目标地域，然后单击Prometheus实例名称。
3. 在左侧导航栏中单击**设置**。
4. 在设置页面单击**设置**页签。
   在**设置**页签下**HTTP API地址（Grafana读取地址）**区域获取API接口地址，根据实际需求选择公网或内网地址，地址格式如下：

- 公网地址

```
http://{region-id}.arms.aliyuncs.com:9090/api/v1/prometheus/xxxxx/{ali-uid}/{cluster-id}/{region-id}
```

- 内网地址

```
http://{region-id}-intranet.arms.aliyuncs.com:9090/api/v1/prometheus/xxxxx/{ali-uid}/{cluster-id}/{region-id}
```

该地址是ARMS提供的Prometheus的访问地址，您可以在Grafana中添加该地址，然后您就可以在Grafana中查看监控图表。具体操作，请参见将阿里云Prometheus监控数据接入本地Grafana。

7.集成自建Prometheus实现网格监控

Prometheus是一款面向云原生应用程序的开源监控工具，本文介绍如何在ASM集成自建Prometheus实现网格监控。

前提条件

- 已创建ACK集群，详情请参见创建Kubernetes托管版集群。
- 已创建ASM实例，详情请参见创建ASM实例。
- 已在ACK集群中创建Prometheus实例和Grafana示例，详情请参见开源Prometheus监控。

步骤一：配置网格数据指标

将以下内容更新到Prometheus配置中，然后重启Prometheus实例，确保配置生效。

```
scrape_configs:
# Mixer scrapping. Defaults to Prometheus and mixer on same namespace.
- job_name: 'istio-mesh'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istio-telemetry;prometheus
# Scrape config for envoy stats
- job_name: 'envoy-stats'
  metrics_path: /stats/prometheus
  kubernetes_sd_configs:
  - role: pod
  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_container_port_name]
    action: keep
    regex: '.*-envoy-prom'
  - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_port]
    action: replace
    regex: ([^:]+)(?::\d+)?;(\d+)
    replacement: $1:15090
    target_label: __address__
  - action: labeldrop
    regex: __meta_kubernetes_pod_label_(.+)
  - source_labels: [__meta_kubernetes_namespace]
    action: replace
    target_label: namespace
  - source_labels: [__meta_kubernetes_pod_name]
    action: replace
    target_label: pod_name
- job_name: 'istio-policy'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
```

```
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istio-policy;http-policy-monitoring
- job_name: 'istio-telemetry'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istio-telemetry;http-monitoring
- job_name: 'pilot'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istiod;http-monitoring
  - source_labels: [__meta_kubernetes_service_label_app]
    target_label: app
- job_name: 'sidecar-injector'
  kubernetes_sd_configs:
  - role: endpoints
    namespaces:
      names:
      - istio-system
  relabel_configs:
  - source_labels: [__meta_kubernetes_service_name, __meta_kubernetes_endpoint_port_name]
    action: keep
    regex: istio-sidecar-injector;http-monitoring
```

步骤二：创建EnvoyFilter

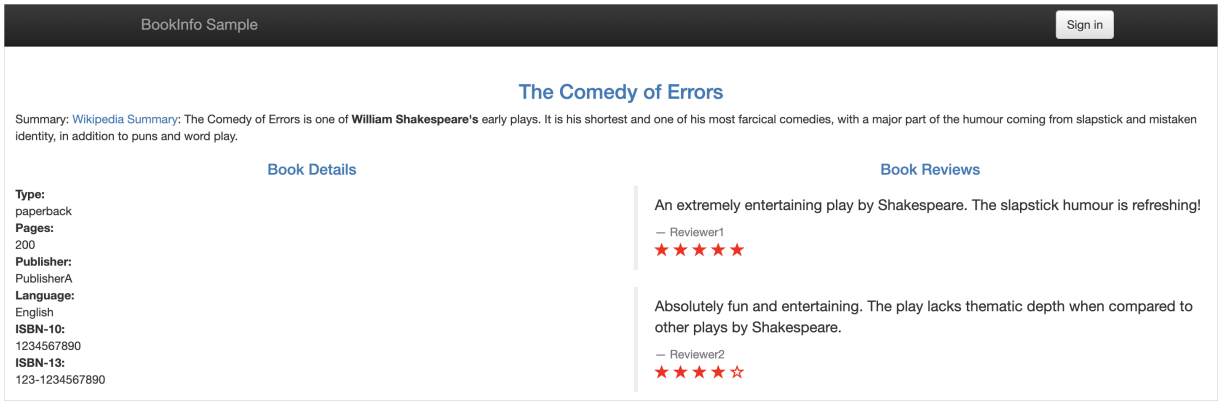
1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面单击右上角的**功能设置**。

> **说明** 请确保ASM实例的Istio为v1.6.8.4及以上版本。

5. 在**功能设置更新**对话框中选中**开启采集Prometheus监控指标**，然后单击**确定**。
   ASM将自动生成采集Prometheus监控指标相关的EnvoyFilter配置。

步骤三：产生监控数据

请求数据平面的服务，这里以Bookinfo为例，详细介绍请参见入门指引。进入Productpage页面，多次刷新页面，以产生监控数据。



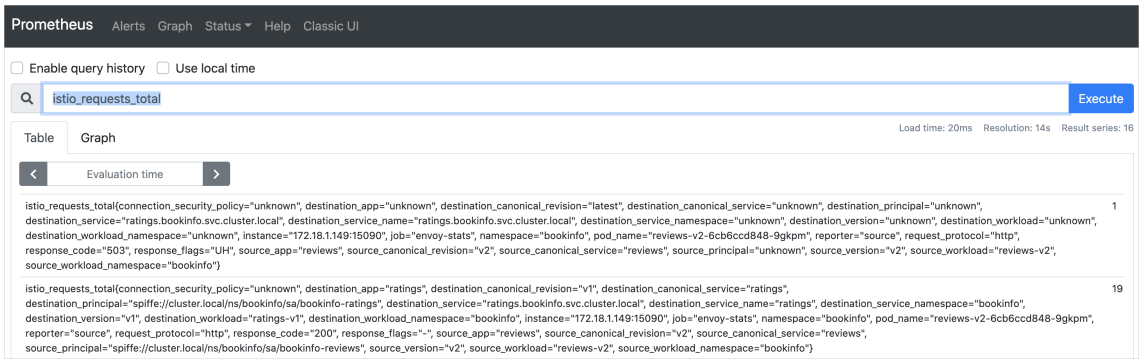
结果验证

- 验证Envoy是否采集监控指标数据。

  执行以下命令，返回监控指标数据，表示Envoy采集监控指标数据。若没有数据，则Envoy没有在采集监控指标数据。

  ```
  details=$(kubectl get pod -l app=details -o jsonpath={.items..metadata.name})
  kubectl exec $details -c istio-proxy -- curl -s localhost:15090/stats/prometheus |grep is
  tio
  ```

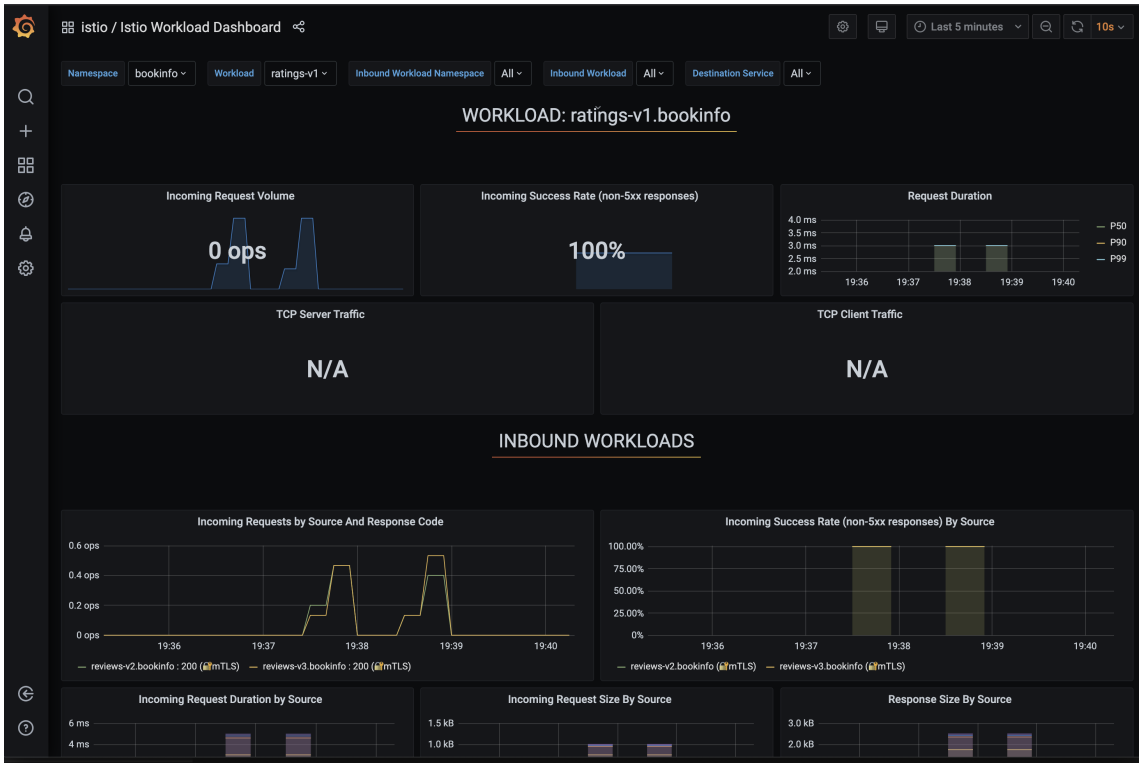
- 查看Prometheus网格监控指标数据。
  i. 登录容器服务管理控制台。
  ii. 在控制台左侧导航栏中，单击**集群**。
  iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
  iv. 在集群管理页左侧导航栏中，选择**网络 > 服务**。
  v. 在服务页面找到Prometheus，单击Prometheus对应的外部端点。
  vi. 在Prometheus页面输入istio_requests_total，单击**Execute**，显示以下页面所示结果。



- 查看Grafana网格监控指标数据。
  i. 登录容器服务管理控制台。
  ii. 在控制台左侧导航栏中，单击**集群**。
  iii. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
  iv. 在集群管理页左侧导航栏中，选择**网络 > 服务**。

v. 在服务页面搜索找到Grafana，单击Grafana对应的外部端点。

vi. 在Grafana页面选择Istio Workload Dashboard，显示以下页面所示结果。



8.集成自建Skywalking实现网格可观测性

Skywalking是一款面向云原生应用程序的开源APM工具，具有分布式追踪、性能指标分析、应用和服务依赖分析等功能。本文介绍ASM如何集成Skywalking查看应用的监控指标，实现网格可观测性。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已创建ACK集群。具体操作，请参见创建Kubernetes托管版集群。
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已部署应用到ASM实例的集群中。具体操作，请参见部署应用到ASM实例。本文以Bookinfo应用为例。
- 已在ACK集群中创建Skywalking。具体操作，请参见使用 SkyWalking 和 Envoy 访问日志服务对服务网格进行观察。

步骤一：创建Skywalking的服务

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。
4. 在集群管理页左侧导航栏中，选择**网络 > 服务**。
5. 在**服务**页面单击**创建**。
6. 在**创建服务**对话框中配置参数，然后单击**创建**。

   以下为重点参数描述，其他参数请参见管理服务。

   - 类型：选择服务访问的方式，本例选择**负载均衡**和**公网访问**。
   - 关联：选择服务要绑定的后端应用，本例为Skywalking。
   - 端口映射：添加服务端口和容器端口，容器端口需要与后端Pod暴露的容器端口一致，本例为11800。

   在**服务**页面单击目标服务右侧**外部端点**列下的地址，即可跳转到Skywalking页面。

步骤二：启用自建Skywalking

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面单击右上角的**功能设置**。
5. 在功能设置更新面板选中**启用自建Skywalking**，输入Skywalking服务的域名和端口，然后单击**确定**。

步骤三：请求Bookinfo应用

1. 下载siege压测工具，并配置环境变量。

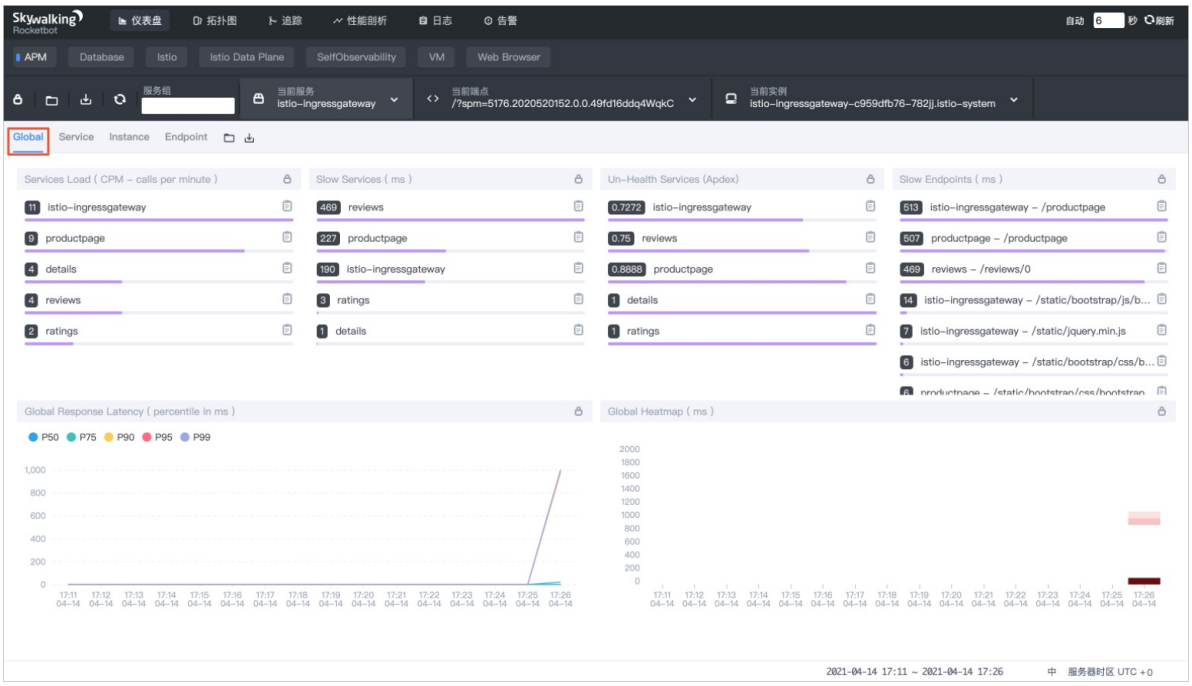
   关于siege的详细信息，请参见siege。
2. 在本地终端执行以下命令，持续请求bookinfo应用，以便产生流量数据。

```
GATEWAY_URL=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.
loadBalancer.ingress[0].ip}')
siege -c 3 "http://${GATEWAY_URL}/productpage"
```

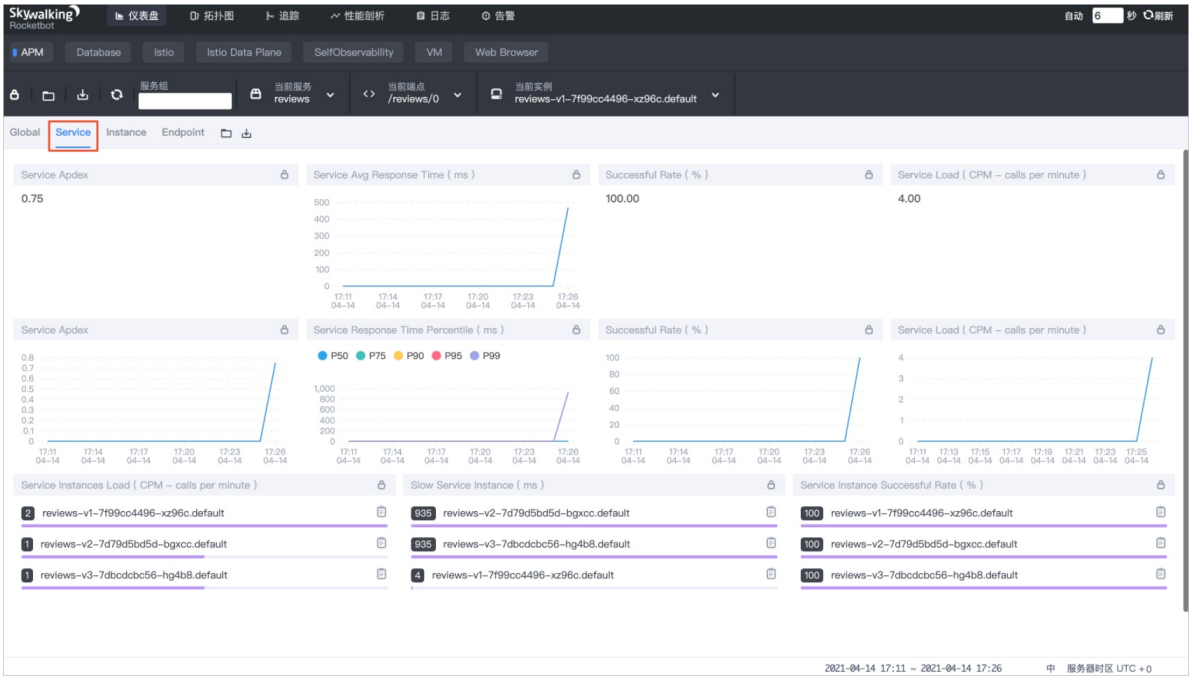
步骤四：查看应用的监控指标

- 在Skywalking页面顶部单击**拓扑图**，查看服务之间的调用链路情况。

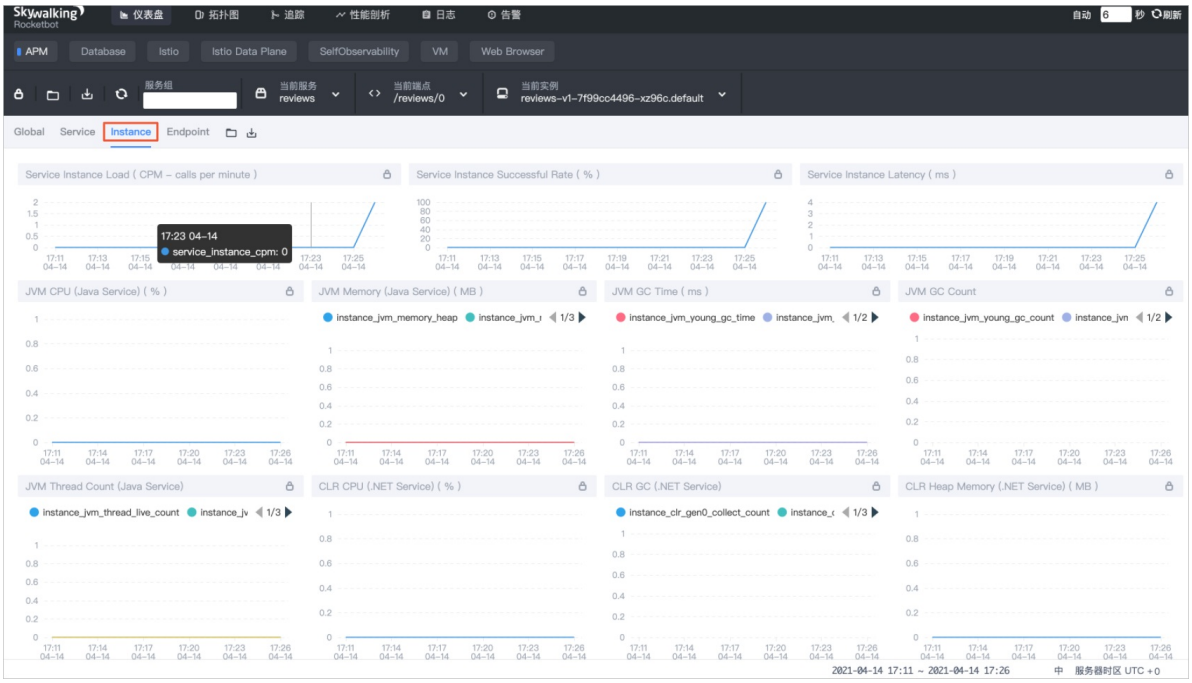
- 在Skywalking页面顶部单击**仪表盘**，然后单击**Global**，查看服务的负载、延迟、健康等状态。



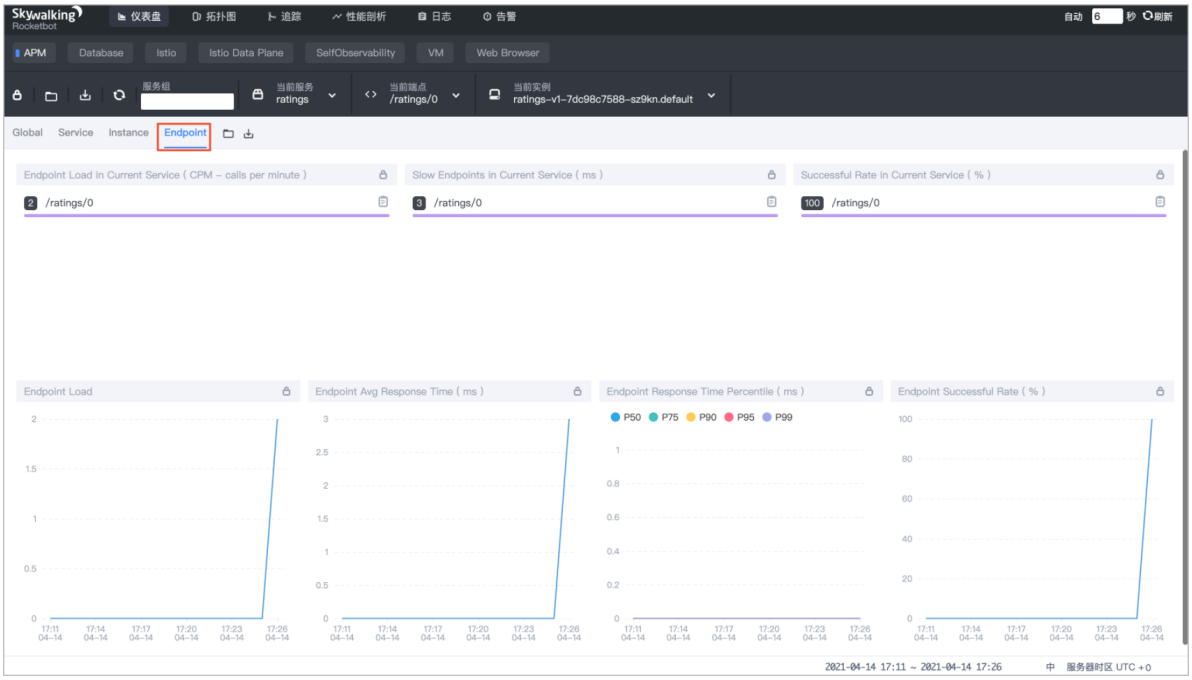
- 在Skywalking页面顶部单击**仪表盘**，然后单击**Service**，查看服务的平均延迟、负载、成功率等数据。



- 在Skywalking页面顶部单击**仪表盘**，然后单击**Instance**，查看实例的各项监控指标。



- 在Skywalking页面顶部单击**仪表盘**，然后单击**Endpoint**，查看端点的各项监控指标。


Skywalking
Rocketbot
仪表盘
拓扑图
追踪
性能剖析
日志
告警
自动
6
秒
刷新
APM
Database
Istio
Istio Data Plane
SelfObservability
VM
Web Browser
服务组
当前服务
ratings
当前端点
/ratings/0
当前实例
ratings-v1-7dc98c7588-sz9kn.default
Global
Service
Instance
Endpoint
Endpoint Load in Current Service (CPM – calls per minute)
2 /ratings/0
Slow Endpoints in Current Service (ms)
3 /ratings/0
Successful Rate in Current Service (%)
100 /ratings/0
Endpoint Load
Endpoint Avg Response Time (ms)
Endpoint Response Time Percentile (ms)
P50 P75 P90 P95 P99
Endpoint Successful Rate (%)
2021-04-14 17:11 ~ 2021-04-14 17:26
中 服务器时区 UTC +0

9.使用链路追踪实现网格内外应用的一体化追踪

ASM可以便捷地为部署在网格内的应用提供链路追踪能力，但要实现网格内和网格外应用的一体化追踪，需要使用链路追踪系统以形成调用链。本文将演示如何通过阿里云链路追踪让网格外部应用调用网格内部应用并形成一个完整的链路追踪信息。

前提条件

- 已创建一个ASM实例，并已将ACK集群添加到ASM实例中。具体操作，请参见创建ASM实例和添加集群到ASM实例。
- ASM实例部署了入口网关，便于访问网格内应用。详情请参见添加入口网关服务。
- 准备一个用于运行网格外示例应用的环境，并安装Python 2.7。
- 启用了链路追踪并选择了阿里云链路追踪。详情请参见编辑ASM实例。

背景信息

- 本示例将部署一个用Python编写的应用ExternalProxy。该应用会启用一个HTTP Server，当ExternalProxy的根路径被访问时，会调用网格内应用BookInfo的Productpage应用。
- 本示例将使用XTrace为追踪系统进行演示。若您使用自行部署的Zipkin兼容的追踪系统，可以将该系统上报地址暴露于网格外应用可以访问的地址，可以跳过步骤二：获取链路追踪Tracing Analysis的接入点地址，直接参照步骤三：部署网格外应用ExternalProxy使用该地址作为上报地址即可。

步骤一：部署网格内应用BookInfo

1. 通过kubectl执行以下命令，将Bookinfo应用部署到ASM实例的集群中。

   BookInfo应用的YAML文件，请从GitHub下载。

   ```
   kubectl --kubeconfig=${DATA_PLANE_KUBECONFIG} apply -f bookinfo.yaml
   ```

2. 部署Bookinfo应用的VirtualService。

   VirtualService的YAML文件，请从GitHub下载。

   ```
   kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f virtual-service-all-v1.yaml
   ```

3. 部署Bookinfo应用的DestinationRule。

   DestinationRule的YAML文件，请从Github下载。

   ```
   kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f destination-rule-all.yaml
   ```

4. 部署Bookinfo应用的Gateway。

   Gateway的YAML文件，请从GitHub下载。

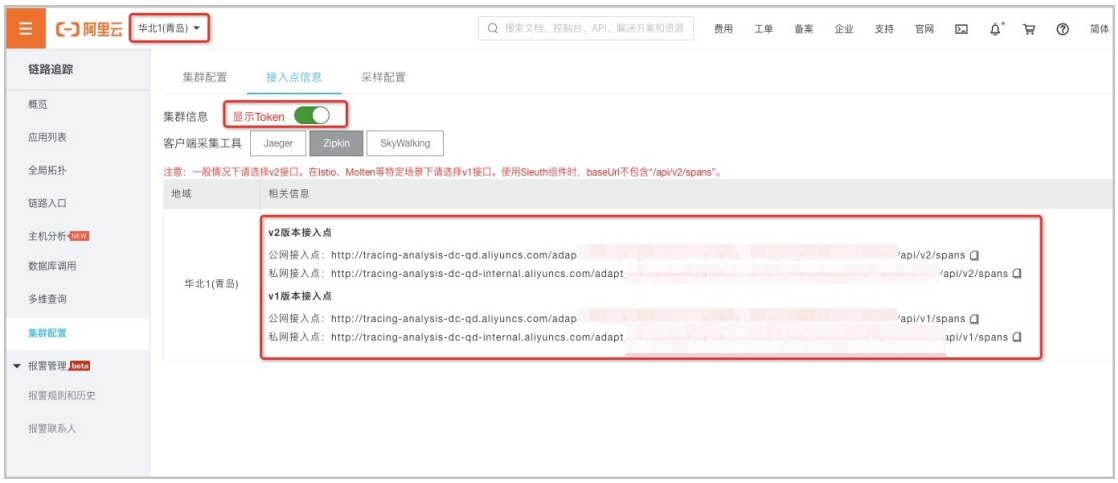
   ```
   kubectl --kubeconfig=${ASM_KUBECONFIG} apply -f bookinfo-gateway.yaml
   ```

步骤二：获取链路追踪Tracing Analysis的接入点地址

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏单击**概览**。
3. 在右侧区域单击**接入流程**页签，然后单击**查看接入点信息**。

4. 查看接入点地址。
   i. 选择地域。应此处您需要选择BookInfo应用部署的ACK集群所在地域来上报数据，否则追踪信息无法关联。
   ii. 单击下方**显示Token**，并根据您的ExternalProxy部署位置决定选择公网或私网接入点地址。

   因为ExternalProxy通过Zipkin V1 API进行数据上报，所以这里我们需用选用V1版本的接入点地址。



步骤三：部署网格外应用ExternalProxy

1. 将下面的代码保存至网格外应用运行环境，命名为*ExternalProxy.py*。

```
import requests
from flask import Flask
from py_zipkin.zipkin import zipkin_span,create_http_headers_for_new_span
from py_zipkin.util import generate_random_64bit_string
from py_zipkin.util import ZipkinAttrs
import time
app = Flask(__name__)
def do_stuff(trace_id, span_id, parent_span_id):
    time.sleep(2)
    headers = create_http_headers_for_new_span()
    headers["X-B3-TraceId"] = trace_id
    headers["X-B3-SpanId"] = span_id
    headers["X-B3-ParentSpanId"] = parent_span_id
    print "SEND TO INGRESS HEADERS : {0}".format(headers)
    r = requests.get('http://${INGRESS_GATE_WAY_IP}/productpage', headers=headers)
    return 'OK'
def http_transport(encoded_span):
    # encoding prefix explained in https://github.com/Yelp/py_zipkin#transport
    body=encoded_span
    zipkin_url="${XTRACE_ZIPKIN_V1_ENDPOINT}"
    headers = {"Content-Type": "application/x-thrift"}
    # You'd probably want to wrap this in a try/except in case POSTing fails
    r=requests.post(zipkin_url, data=body, headers=headers)
    print(body)
@app.route('/')
def index():
    with zipkin_span(
        service_name='external-proxy',
        span_name='external-proxy/inbound',
        transport_handler=http_transport,
        port=5000,
        sample_rate=100,
    ) as inbound_span:
    do_stuff(inbound_span.zipkin_attrs.trace_id, inbound_span.zipkin_attrs.span_id, inb
ound_span.zipkin_attrs.parent_span_id)
    return 'OK', 200
if __name__=='__main__':
    app.run(host="0.0.0.0",port=5000,debug=True)
```

2. 替换代码中的下述内容。
   - 将*{XTRACE_ZIPKIN_V1_ENDPOINT}*替换为上一步获取到的接入点地址。如果您希望上报于自行部署的追踪系统，这里请替换为其上报地址。
   - 将*{INGRESS_GATE_WAY_IP}*替换为ACK集群Ingress Gateway的地址。
3. 执行下面的命令，启动ExternalProxy。

```
python ExternalProxy.py
```

```
* Serving Flask app "main" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 317-792-686
```

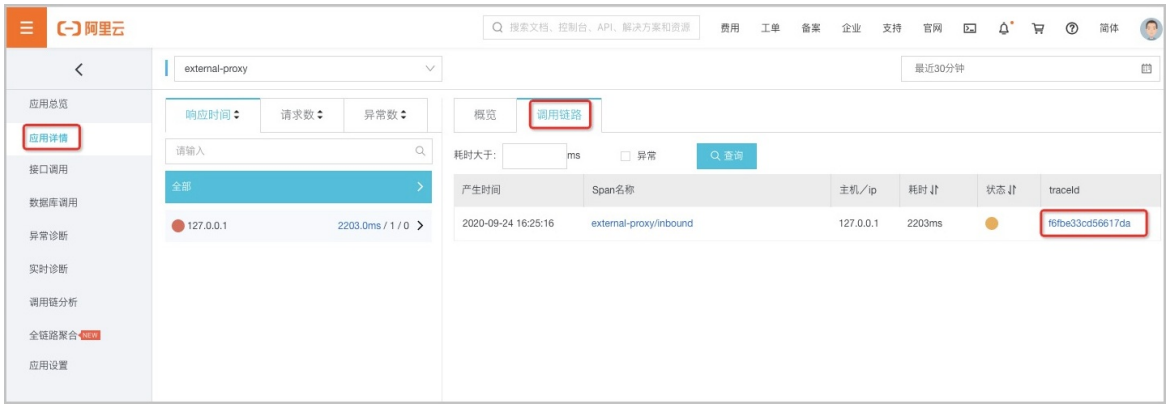
4. 调用ExternalProxy。

```
curl localhost:5000
```

```
OK
```

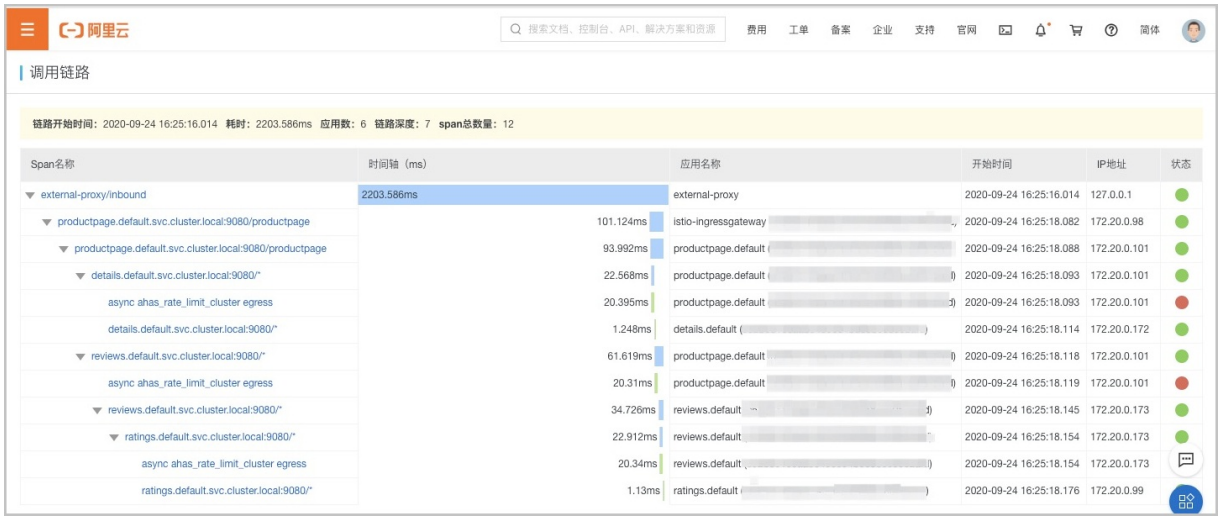
步骤四：查看链路追踪信息

1. 登录链路追踪Tracing Analysis控制台。
2. 在左侧导航栏单击**应用列表**，然后在上方选择地域，单击下方列表中的ExternalProxy。
3. 在应用页面的左侧导航栏单击**应用详情**，然后单击**调用链路**页签，最后单击相应的traceid查看链路追踪信息。



执行结果

可以看到，网格外部应用ExternalProxy与内部应用BookInfo之间已形成完整的调用链路。


阿里云
搜索文档、控制台、API、解决方案和资源
费用
工单
备案
企业
支持
官网
简体
调用链路
链路开始时间：2020-09-24 16:25:16.014 耗时：2203.586ms 应用数：6 链路深度：7 span总数量：12
Span名称
时间轴（ms）
应用名称
开始时间
IP地址
状态
external-proxy/inbound 2203.586ms external-proxy 2020-09-24 16:25:16.014 127.0.0.1
productpage.default.svc.cluster.local:9080/productpage 101.124ms istio-ingressgateway 2020-09-24 16:25:18.082 172.20.0.98
productpage.default.svc.cluster.local:9080/productpage 93.992ms productpage.default 2020-09-24 16:25:18.088 172.20.0.101
details.default.svc.cluster.local:9080/* 22.568ms productpage.default 2020-09-24 16:25:18.093 172.20.0.101
async ahas_rate_limit_cluster egress 20.395ms productpage.default 2020-09-24 16:25:18.093 172.20.0.101
details.default.svc.cluster.local:9080/* 1.248ms details.default 2020-09-24 16:25:18.114 172.20.0.172
reviews.default.svc.cluster.local:9080/* 61.619ms productpage.default 2020-09-24 16:25:18.118 172.20.0.101
async ahas_rate_limit_cluster egress 20.31ms productpage.default 2020-09-24 16:25:18.119 172.20.0.101
reviews.default.svc.cluster.local:9080/* 34.726ms reviews.default 2020-09-24 16:25:18.145 172.20.0.173
ratings.default.svc.cluster.local:9080/* 22.912ms reviews.default 2020-09-24 16:25:18.154 172.20.0.173
async ahas_rate_limit_cluster egress 20.34ms reviews.default 2020-09-24 16:25:18.154 172.20.0.173
ratings.default.svc.cluster.local:9080/* 1.13ms ratings.default 2020-09-24 16:25:18.176 172.20.0.99

10.使用ASM指标实现工作负载的自动弹性伸缩

服务网格ASM为ACK集群内的服务通信提供了一种非侵入式的生成遥测数据的能力。这种遥测功能提供了服务行为的可观测性，可以帮助运维人员对应用程序进行故障排除、维护和优化，而不会带来任何额外负担。根据监控的四个黄金指标维度（延迟、流量、错误和饱和度），服务网格ASM为管理的服务生成一系列指标。本文介绍如何使用ASM指标实现工作负载的自动弹性伸缩。

前提条件

- 已创建ACK集群。更多信息，请参见创建Kubernetes托管版集群。
- 已创建ASM实例。更多信息，请参见创建ASM实例。
- 已在ACK集群中创建Prometheus实例和Grafana示例。更多信息，请参见开源Prometheus监控。
- 已集成Prometheus实现网格监控。更多信息，请参见集成自建Prometheus实现网格监控。

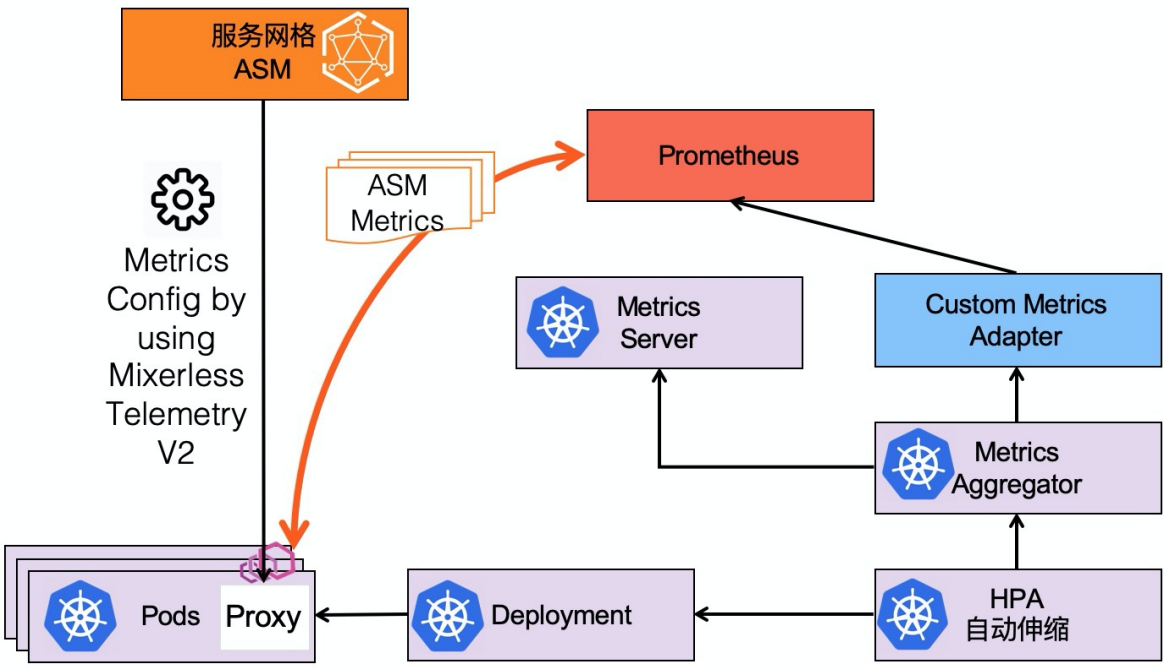
背景信息

服务网格ASM为管理的服务生成一系列指标。更多信息，请参见Istio标准指标。

自动伸缩是一种根据资源使用情况进行自动扩缩工作负载的方法。Kubernetes中的自动伸缩具有以下两个维度：

- 集群自动伸缩器CA（Cluster Autoscaler）：用于处理节点伸缩操作，可以增加或减少节点。
- 水平自动伸缩器HPA（Horizontal Pod Autoscaler）：用于自动伸缩应用部署中的Pod，可以调节Pod的数量。

Kubernetes提供的聚合层允许第三方应用程序将自身注册为API Addon组件来扩展Kubernetes API。这样的Addon组件可以实现Custom Metrics API，并允许HPA访问任意指标。HPA会定期通过Resource Metrics API查询核心指标（例如CPU或内存）以及通过Custom Metrics API获取特定于应用程序的指标，包括ASM提供的可观测性指标。



步骤一：开启采集Prometheus监控指标

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面单击右上角的**功能设置**。

   > **说明** 请确保ASM实例的Istio为1.6.8.4及以上版本。

5. 在**功能设置更新**面板中选中**开启采集Prometheus监控指标**，然后单击**确定**。

   ASM将自动生成采集Prometheus监控指标相关的EnvoyFilter配置。

步骤二：部署自定义指标API Adapter

1. 下载Adapter安装包，关于Adapter安装包请参见kube-metrics-adapter。然后在ACK集群中安装部署自定义指标API Adapter。

   ```
   ## 如果使用Helm v3。
   helm -n kube-system install asm-custom-metrics ./kube-metrics-adapter  --set prometheus
   .url=http://prometheus.istio-system.svc:9090
   ```

2. 安装完成后，通过以下方式确认kube-metrics-adapter已启用。
   - 确认autoscaling/v2beta已存在。

     ```
     kubectl api-versions |grep "autoscaling/v2beta"
     ```

     预期输出：

     ```
     autoscaling/v2beta
     ```

   - 确认kube-metrics-adapter Pod状态。

     ```
     kubectl get po -n kube-system |grep metrics-adapter
     ```

     预期输出：

     ```
     asm-custom-metrics-kube-metrics-adapter-85c6d5d865-2cm57          1/1     Running   0
     19s
     ```

   - 列出Prometheus适配器提供的自定义外部指标。

     ```
     kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
     ```

     预期输出：

     ```
     {
       "kind": "APIResourceList",
       "apiVersion": "v1",
       "groupVersion": "external.metrics.k8s.io/v1beta1",
       "resources": []
     }
     ```

步骤三：部署示例应用

1. 创建test命名空间。具体操作，请参见管理命名空间。
2. 启用Sidecar自动注入。具体操作，请参见安装Sidecar代理。
3. 部署示例应用。
   i. 创建*podinfo.yaml*文件。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: podinfo
  namespace: test
  labels:
    app: podinfo
spec:
  minReadySeconds: 5
  strategy:
    rollingUpdate:
      maxUnavailable: 0
    type: RollingUpdate
  selector:
    matchLabels:
      app: podinfo
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
      labels:
        app: podinfo
    spec:
      containers:
      - name: podinfod
        image: stefanprodan/podinfo:latest
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9898
          name: http
          protocol: TCP
        command:
        - ./podinfo
        - --port=9898
        - --level=info
        livenessProbe:
          exec:
            command:
            - podcli
            - check
            - http
            - localhost:9898/healthz
          initialDelaySeconds: 5
          timeoutSeconds: 5
        readinessProbe:
          exec:
            command:
            - podcli
```

```
            - check
            - http
            - localhost:9898/readyz
          initialDelaySeconds: 5
          timeoutSeconds: 5
        resources:
          limits:
            cpu: 2000m
            memory: 512Mi
          requests:
            cpu: 100m
            memory: 64Mi
---
apiVersion: v1
kind: Service
metadata:
  name: podinfo
  namespace: test
  labels:
    app: podinfo
spec:
  type: ClusterIP
  ports:
    - name: http
      port: 9898
      targetPort: 9898
      protocol: TCP
  selector:
    app: podinfo
```

ii. 部署podinfo。

```
kubectl apply -n test -f podinfo.yaml
```

4. 为了触发自动弹性伸缩，需要在命名空间test中部署负载测试服务，用于触发请求。

i. 创建*loadtester.yaml*文件。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loadtester
  namespace: test
  labels:
    app: loadtester
spec:
  selector:
    matchLabels:
      app: loadtester
  template:
    metadata:
      labels:
        app: loadtester
      annotations:
        prometheus.io/scrape: "true"
    spec:
```

```
      containers:
        - name: loadtester
          image: weaveworks/flagger-loadtester:0.18.0
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
          command:
            - ./loadtester
            - -port=8080
            - -log-level=info
            - -timeout=1h
          livenessProbe:
            exec:
              command:
                - wget
                - --quiet
                - --tries=1
                - --timeout=4
                - --spider
                - http://localhost:8080/healthz
            timeoutSeconds: 5
          readinessProbe:
            exec:
              command:
                - wget
                - --quiet
                - --tries=1
                - --timeout=4
                - --spider
                - http://localhost:8080/healthz
            timeoutSeconds: 5
          resources:
            limits:
              memory: "512Mi"
              cpu: "1000m"
            requests:
              memory: "32Mi"
              cpu: "10m"
          securityContext:
            readOnlyRootFilesystem: true
            runAsUser: 10001
---
apiVersion: v1
kind: Service
metadata:
  name: loadtester
  namespace: test
  labels:
    app: loadtester
spec:
  type: ClusterIP
  selector:
    app: loadtester
```

```
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: http
```

ii. 部署负载测试服务。

```
kubectl apply -n test -f loadtester.yaml
```

5. 验证部署示例应用和负载测试服务是否成功。

i. 确认Pod状态。

```
kubectl get pod -n test
```

预期输出：

```
NAME                          READY   STATUS    RESTARTS   AGE
loadtester-64df4846b9-nxhvv   2/2     Running   0          2m8s
podinfo-6d845cc8fc-26xbq      2/2     Running   0          11m
```

ii. 进入负载测试器容器，并使用hey命令生成负载。

```
export loadtester=$(kubectl -n test get pod -l "app=loadtester" -o jsonpath='{.item
s[0].metadata.name}')
kubectl -n test exec -it ${loadtester} -c loadtester -- hey -z 5s -c 10 -q 2 http:/
/podinfo.test:9898
```

返回结果，生成负载成功，说明示例应用和负载测试服务部署成功。

步骤四：使用ASM指标配置HPA

定义一个HPA，该HPA将根据每秒接收的请求数来扩缩Podinfo的工作负载数量。当平均请求流量负载超过10 req/sec时，将指示HPA扩大部署。

1. 创建*hpa.yaml*。

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/processed-requests-per-second: |
      sum(
          rate(
              istio_requests_total{
                destination_workload="podinfo",
                destination_workload_namespace="test",
                reporter="destination"
              }[1m]
          )
      )
spec:
  maxReplicas: 10
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
    - type: External
      external:
        metric:
          name: prometheus-query
          selector:
            matchLabels:
              query-name: processed-requests-per-second
        target:
          type: AverageValue
          averageValue: "10"
```

2. 部署HPA。

```
kubectl apply -f hpa.yaml
```

3. 验证HPA是否部署成功。

   列出Prometheus适配器提供的自定义外部指标。

```
kubectl get --raw "/apis/external.metrics.k8s.io/v1beta1" | jq .
```

   预期输出：

```
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "external.metrics.k8s.io/v1beta1",
  "resources": [
    {
      "name": "prometheus-query",
      "singularName": "",
      "namespaced": true,
      "kind": "ExternalMetricValueList",
      "verbs": [
        "get"
      ]
    }
  ]
}
```

返回结果中包含自定义的ASM指标的资源列表，说明HPA部署成功。

验证自动弹性伸缩

1. 进入测试器容器，并使用hey命令生成工作负载请求。

```
kubectl -n test exec -it ${loadtester} -c loadtester -- sh
~ $ hey -z 5m -c 10 -q 5 http://podinfo.test:9898
```

2. 查看自动伸缩状况。

> **说明** 默认情况下，指标每30秒执行一次同步，并且只有在最近3分钟~5分钟内容器没有重新缩放时，才可以进行放大或缩小。这样，HPA可以防止冲突决策的快速执行，并为集群自动扩展程序预留时间。

```
watch kubectl -n test get hpa/podinfo
```

预期输出：

```
NAME      REFERENCE             TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
podinfo   Deployment/podinfo    8308m/10 (avg)   1         10        6          124m
```

一分钟后，HPA将开始扩大工作负载，直到请求/秒降至目标值以下。负载测试完成后，每秒的请求数将降为零，并且HPA将开始缩减工作负载Pod数量，几分钟后上述命令返回结果中的REPLICAS副本数将恢复为一个。

11.通过ASM实现gRPC链路追踪

链路追踪Tracing Analysis为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等工具。本文介绍如何通过Headers在ASM实现gRPC链路追踪。

示例工程

gRPC的示例工程请参见hello-servicemesh-grpc，本文档中提到的目录都为hello-servicemesh-grpc下的目录。

GRPC协议Headers编程实践

服务端获取Headers

- 基本方法
  - 使用Java语言通过服务端获取Headers实现基本方法。

    实现拦截器 `ServerInterceptor` 接口的 `interceptCall(ServerCall<ReqT, RespT> call,final Metadata m,ServerCallHandler<ReqT, RespT> h)` 方法，通过 `String v = m.get(k)` 获取header信息， `get` 方法入参类型为 `Metadata.Key<String>` 。
  - 使用Go语言通过服务端获取Headers实现基本方法。

    `metadata.FromIncomingContext(ctx)(md MD, ok bool)` ，MD是一个 `map[string][]string` 。
  - 使用NodeJS语言通过服务端获取Headers实现基本方法。

    `call.metadata.getMap()` ，返回值类型是 `[key: string]: MetadataValue` ， `MetadataValue` 类型定义为 `string/Buffer` 。
  - 使用Python语言通过服务端获取Headers实现基本方法。

    `context.invocation_metadata()` ，返回值类型为2-tuple数组，2-tuple的形式为 `('k','v')` ，使用 `m.key, m.value` 获取键值对。
- Unary RPC
  - 使用Java语言通过服务端获取Headers实现Unary RPC。

    对Headers无感知。
  - 使用Go语言通过服务端获取Headers实现Unary RPC。

    在方法中直接调用 `metadata.FromIncomingContext(ctx)` ，上下文参数ctx来自Talk的入参。
  - 使用NodeJS语言通过服务端获取Headers实现Unary RPC。

    在方法内直接调用 `call.metadata.getMap()` 。
  - 使用Python语言通过服务端获取Headers实现Unary RPC。

    在方法内直接调用 `context.invocation_metadata()` 。
- Server streaming RPC
  - 使用Java语言通过服务端获取Headers实现Server streaming RPC。

    对Headers无感知。

- 使用Go语言通过服务端获取Headers实现Server streaming RPC。

  在方法中直接调用 `metadata.FromIncomingContext(ctx)` ，上下文参数 `ctx` 从TalkOneAnswerMore的入参 `stream` 中获取 `stream.Context()` 。

- 使用NodeJS语言通过服务端获取Headers实现Server streaming RPC。

  在方法内直接调用 `call.metadata.getMap()` 。

- 使用Python语言通过服务端获取Headers实现Server streaming RPC。

  在方法内直接调用 `context.invocation_metadata()` 。

- Client streaming RPC
  - 使用Java语言通过服务端获取Headers实现Client streaming RPC。

    对Headers无感知。

  - 使用Go语言通过服务端获取Headers实现Client streaming RPC。

    在方法中直接调用 `metadata.FromIncomingContext(ctx)` ，上下文参数 `ctx` 从TalkMoreAnswerOne的入参 `stream` 中获取 `stream.Context()` 。

  - 使用NodeJS语言通过服务端获取Headers实现Client streaming RPC。

    在方法内直接调用 `call.metadata.getMap()` 。

  - 使用Python语言通过服务端获取Headers实现Client streaming RPC。

    在方法内直接调用 `context.invocation_metadata()` 。

- Bidirectional streaming RPC
  - 使用Java语言通过服务端获取Headers实现Bidirectional streaming RPC。

    对Headers无感知。

  - 使用Go语言通过服务端获取Headers实现Bidirectional streaming RPC。

    在方法中直接调用 `metadata.FromIncomingContext(ctx)` ，上下文参数 `ctx` 从TalkBidirectional的入参 `stream` 中获取 `stream.Context()` 。

  - 使用NodeJS语言通过服务端获取Headers实现Bidirectional streaming RPC。

    在方法内直接调用 `call.metadata.getMap()` 。

  - 使用Python语言通过服务端获取Headers实现Bidirectional streaming RPC。

    在方法内直接调用 `context.invocation_metadata()` 。

客户端发送Headers

- 基本方法
  - 使用Java语言通过客户端发送Headers实现基本方法。

    实现拦截器 `ClientInterceptor` 接口的 `interceptCall(MethodDescriptor<ReqT, RespT> m , CallOptions o, Channel c)` 方法，实现返回值类型 `ClientCall<ReqT , RespT>`的`start((Listener<RespT> l, Metadata h))` 方法，通过 `h.put(k, v)` 填充header信息， `put` 方法入参 `k` 的类型为 `Metadata.Key<String>` ， `v` 的类型为 `String` 。

  - 使用Go语言通过客户端发送Headers实现基本方法。

    `metadata.AppendToOutgoingContext(ctx,kv ...) context.Context`

- 使用NodeJS语言通过客户端发送Headers实现基本方法。

  ```
  metadata=call.metadata.getMap()metadata.add(key, headers[key])
  ```

- 使用Python语言通过客户端发送Headers实现基本方法。

  `metadata_dict = {}` 变量填充 `metadata_dict[c.key] = c.value`，最终转为 `list tuple` 类型 `list(metadata_dict.items())`。

- Unary RPC
  - 使用Java语言通过客户端发送Headers实现Unary RPC。

    对Headers无感知。
  - 使用Go语言通过客户端发送Headers实现Unary RPC。

    在方法中直接调用 `metadata.AppendToOutgoingContext(ctx,kv)`。
  - 使用NodeJS语言通过客户端发送Headers实现Unary RPC。

    在方法内直接使用基本方法。
  - 使用Python语言通过客户端发送Headers实现Unary RPC。

    在方法内直接使用基本方法。
- Server streaming RPC
  - 使用Java语言通过客户端发送Headers实现Server streaming RPC。

    对Headers无感知。
  - 使用Go语言通过客户端发送Headers实现Server streaming RPC。

    在方法中直接调用 `metadata.AppendToOutgoingContext(ctx,kv)`。
  - 使用NodeJS语言通过客户端发送Headers实现Server streaming RPC。

    在方法内直接使用基本方法。
  - 使用Python语言通过客户端发送Headers实现Server streaming RPC。

    在方法内直接使用基本方法。
- Client streaming RPC
  - 使用Java语言通过客户端发送Headers实现Client streaming RPC。

    对Headers无感知。
  - 使用Go语言通过客户端发送Headers实现Client streaming RPC。

    在方法中直接调用 `metadata.AppendToOutgoingContext(ctx,kv)`。
  - 使用NodeJS语言通过客户端发送Headers实现Client streaming RPC。

    在方法内直接使用基本方法。
  - 使用Python语言通过客户端发送Headers实现Client streaming RPC。

    在方法内直接使用基本方法。
- Bidirectional streaming RPC
  - 使用Java语言通过客户端发送Headers实现Bidirectional streaming RPC。

    对Headers无感知。

- 使用Go语言通过客户端发送Headers实现Bidirectional streaming RPC。

  在方法中直接调用 `metadata.AppendToOutgoingContext(ctx,kv)` 。
- 使用NodeJS语言通过客户端发送Headers实现Bidirectional streaming RPC。

  在方法内直接使用基本方法。
- 使用Python语言通过客户端发送Headers实现Bidirectional streaming RPC。

  在方法内直接使用基本方法。

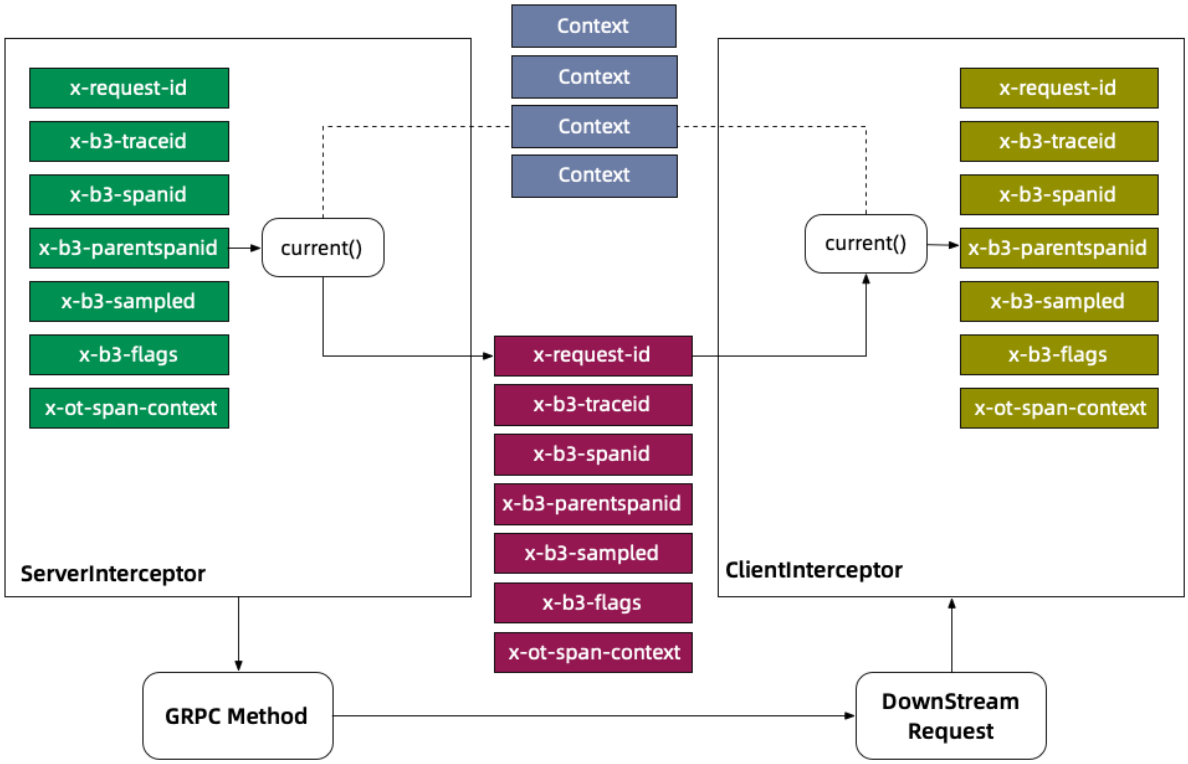
Propaganda Headers

由于链路追踪需要将上游传递过来的链路元数据透传给下游，以形成同一条请求链路的完整信息，需要将服务端获取的Headers信息中，和链路追踪相关的Headers透传给向下游发起请求的客户端。

除了Java语言的实现，其他语言的通信模型方法都对Headers有感知，因此可以将服务端读取Headers-传递Headers-客户端发送Headers这三个动作有顺序地在4种通信模型方法内部实现。

Java语言读取和写入Headers是通过两个拦截器分别实现的，因此Propaganda Headers无法在一个顺序的流程里实现，且考虑到并发因素，以及只有读取拦截器知道链路追踪的唯一ID，所以无法通过最直觉的缓存方式搭建两个拦截器的桥梁。

Java语言的实现提供了一种Metadata-Context Propagation的机制。



在服务器拦截器读取阶段，通过 `ctx.withValue(key, metadata)` 将 `Metadata/Header` 存入Context，其中Key是 `Context.Key<String>` 类型。然后在客户端拦截器中，通过 `key.get()` 将 `Metadata`从 `Context` 读出，get方法默认使用 `Context.current()` 上下文，这就保证了一次请求的Headers读取和写入使用的是同一个上下文。

有了Propaganda Headers的实现，基于GRPC的链路追踪就有了机制上的保证。

部署和验证网格拓扑

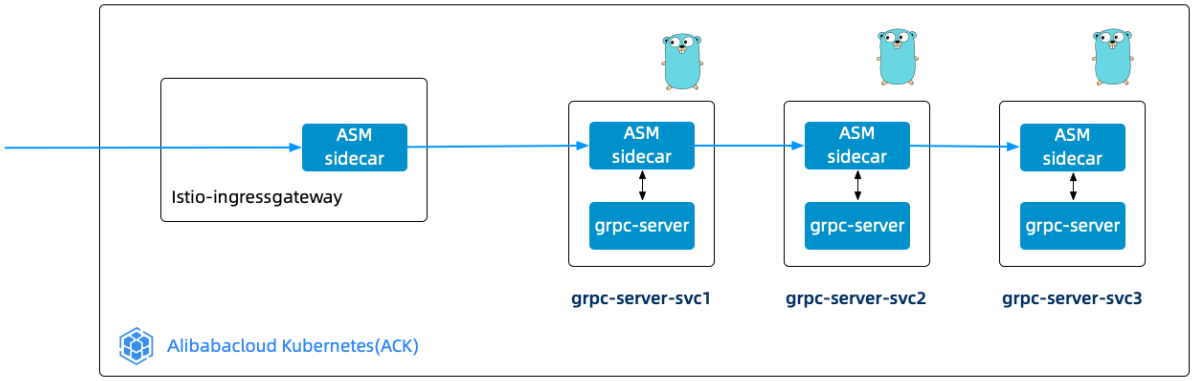
实现gRPC链路追踪之前，您需要部署和验证网格拓扑，确保网格拓扑是可以通信的。

进入示例工程的tracing目录，该目录下包含4种编程语言的部署脚本。以下以Go版本为例，部署和验证网格拓扑。

```
cd go
# 部署
sh apply.sh
# 验证
sh test.sh
```

如果没有出现异常信息，则说明网格拓扑是可以通信的。

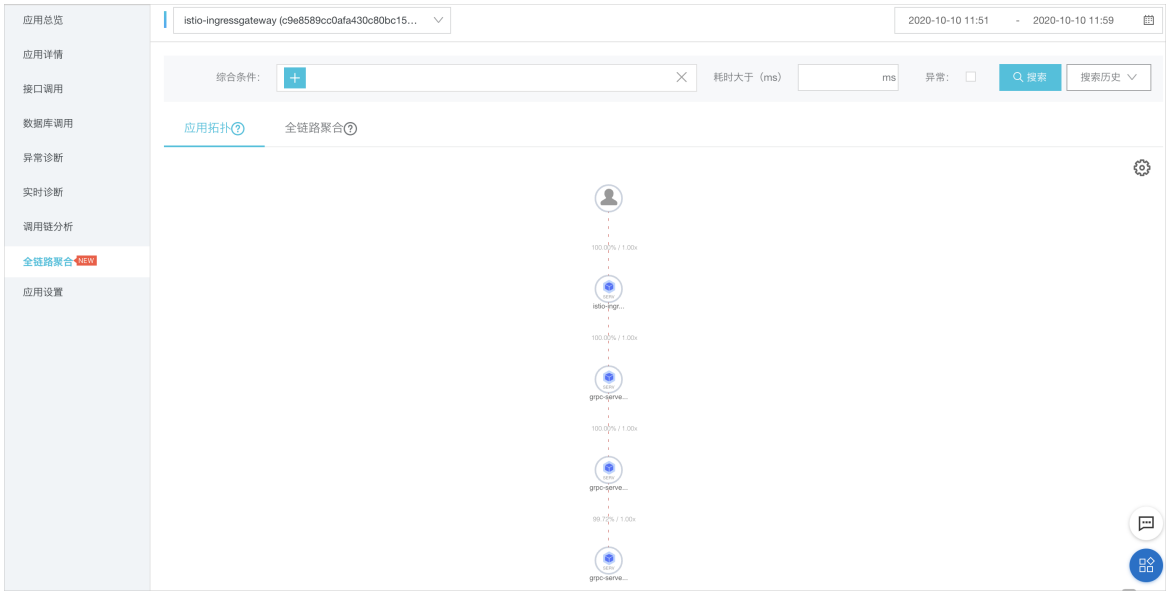
部署后的服务网格拓扑如下图所示。



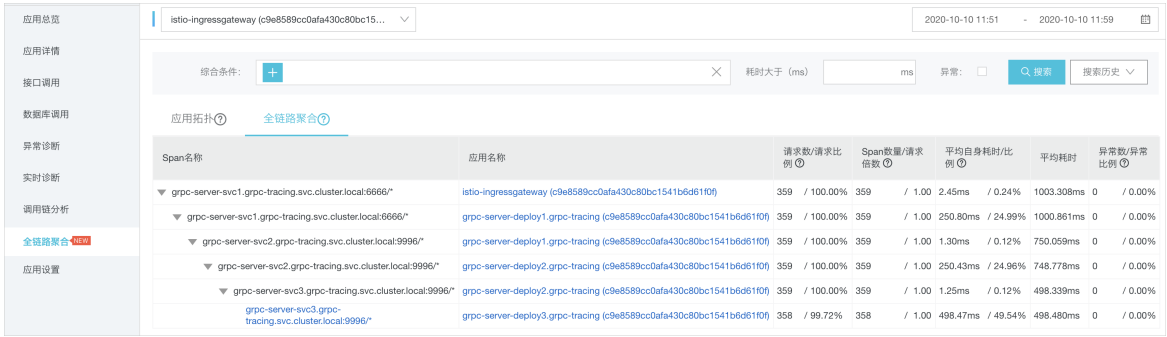
链路追踪

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面右上角单击**功能设置**。
5. 在**功能设置更新**对话框中选中**启用链路追踪**，设置**采用方式**为**阿里云XTrace**，然后单击**确定**。
6. 在左侧导航栏中单击**链路追踪**，跳转到链路追踪控制台。
7. 在链路追踪控制台左侧导航栏中单击**链路入口**。
8. 在**链路入口**页面单击目标应用的**应用拓扑**。

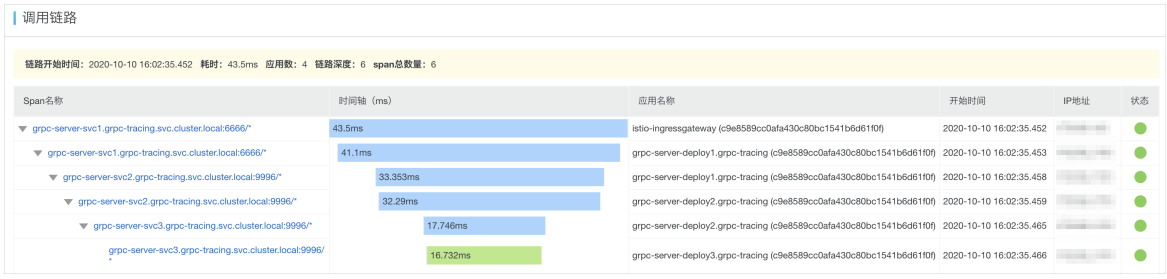
   可以看到完整的链路，包括本地请求端-Ingressgateway-grpc-server-svc1-grpc-server-svc2-grpc-server-svc3。



9. 在**全链路聚合**页面单击**全链路聚合**页签，查看全链路聚合。



10. 在**全链路聚合**页签下单击**Span名称**下的链路，查看调用链路。



12.通过ASM控制台开启Kiali的可观测性

Kiali for ASM是一个服务网格可观测性工具，提供了查看相关服务与配置的可视化界面。ASM从1.7.5.25版本开始支持内置Kiali for ASM。本文介绍如何通过ASM控制台开启Kiali for ASM的可观测性。

前提条件

- 已创建容器服务托管版集群。具体操作，请参见创建Kubernetes托管版集群。
- 已创建ASM实例。具体操作，请参见创建ASM实例。
- 已为集群添加入口网关。具体操作，请参见添加入口网关服务。
- 已自建Prometheus或集成ARMS Prometheus。具体操作，请参见集成自建Prometheus实现网格监控或集成ARMS Prometheus实现网格监控。

启用Kiali for ASM

在创建ASM实例时启用Kiali for ASM

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面单击**创建新网格**。
4. 在创建新网格面板选中**开启采集Prometheus监控指标**，然后选中**启用Kiali提升网格可观测**，其他参数配置，请参见创建ASM实例，最后单击**确定**。

> **说明**
> - 选中**开启采集Prometheus监控指标**，只是开启采集服务网格实例指标功能，并不会自动创建ARMS实例或者自建的Prometheus实例。
> - 去掉选中**开启采集Prometheus监控指标**和**启用Kiali**，就可以关闭Kiali for ASM。此时再重新开启Kiali for ASM，将会是全新的Kiali for ASM。

在编辑ASM实例时启用Kiali for ASM

1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格管理详情页面右上角单击**功能设置**。
5. 在**功能设置更新**对话框中选中**开启采集Prometheus监控指标**，然后选中**启用Kiali**，最后单击**确定**。

> **说明** 去掉选中**开启采集Prometheus监控指标**和**启用Kiali**，就可以关闭Kiali for ASM。此时再重新开启Kiali for ASM，将会是全新的Kiali for ASM。

访问Kiali for ASM

使用Ingress Gateway访问Kiali for ASM

1. 添加入口网关服务。具体操作，请参见添加入口网关服务或自定义入口网关服务。
2. 为入口网关添加以下端口配置，以支持Kiali访问。具体操作，请参见修改入口网关服务。

```
- name: http-kiali
  port: 20001
  protocol: TCP
  targetPort: 20001
```

```
spec:
  clusterIds:
    - c729bdf9ef09b4a259e693f76a67e194e
  cpu: {}
  externalTrafficPolicy: Local
  maxReplicas: 5
  minReplicas: 2
  ports:
    - name: status-port
      port: 15020
      targetPort: 15020
    - name: http2
      port: 80
      targetPort: 80
    - name: https
      port: 443
      targetPort: 443
    - name: tls
      port: 15443
      targetPort: 15443
    - name: http-kiali
      port: 20001
      protocol: TCP
      targetPort: 20001
```

3. 创建网关规则，以下为网关规则配置。具体操作，请参见管理网关规则。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: kiali-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: http
        number: 20001
        protocol: HTTP
```

4. 创建虚拟服务，以下为虚拟服务。具体操作，请参见管理虚拟服务。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: kiali-vs
  namespace: istio-system
spec:
  gateways:
    - kiali-gateway
  hosts:
    - '*'
  http:
    - route:
        - destination:
            host: kiali
            port:
              number: 20001
```

在网格管理详情页面**基本信息**区域**启用Kiali**右侧显示**通过入口网关访问**链接，单击该链接，跳转到Kiali的登录页面。

使用Service访问Kiali for ASM

1. 步骤二：选择集群凭证类型。
2. 部署Service。
   i. 创建*service.yaml*。

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: kiali-svc
     namespace: istio-system
     labels:
       app: kiali-svc
   spec:
     ports:
       - name: http
         port: 20001
         targetPort: 20001
         protocol: TCP
     selector:
       istio: kiali
     type: LoadBalancer
   ```

   ii. 执行以下命令，部署Service。

   ```
   kubectl apply -f service.yaml
   ```

登录Kiali

通过ACK控制台获取Token，然后登录Kiali。

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，单击**集群**。
3. 在**集群列表**页面中，单击目标集群名称或者目标集群右侧**操作**列下的**详情**。

4. 在集群管理页左侧导航栏中，选择**配置管理 > 保密字典**。
5. 在**保密字典**页面设置命名空间为**istio-system**，单击**kiali-service-account-token-******，单击**token**行的图标，获取Token。
6. 在Kiali控制台登录页面输入Token，单击**log in**，登录Kiali控制台。

通过命令行获取Token，然后登录Kiali。

1. 执行以下命令，获取Token。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
k get secrets -o jsonpath="{.items[?(@.metadata.annotations['kubernetes\.io/service-acc
ount\.name']=='kiali-service-account')].data.token}" -n istio-system | base64 --decode
```

2. 在Kiali控制台登录页面输入Token，单击**log in**，登录Kiali控制台。

13.在服务网格ASM中自定义Metrics

在服务网格ASM开启采集Prometheus监控指标后，Envoy会输出默认的监控指标。ASM支持使用自定义Metric的功能针对性输出监控指标，支持自定义网格级别、命名空间级别、工作负载级别的监控指标。本文介绍如何在服务网格ASM中自定义Metrics。

背景信息

Istio会默认生成监控指标，每个监控指标包含标签。以下介绍Istio会生成的指标：

- 对于HTTP、HTTP/2和GRPC流量，Istio会生成以下指标：

```
Request Count (istio_requests_total): This is a COUNTER incremented for every request handled by an Istio proxy.
Request Duration (istio_request_duration_milliseconds): This is a DISTRIBUTION which measures the duration of requests.
Request Size (istio_request_bytes): This is a DISTRIBUTION which measures HTTP request body sizes.
Response Size (istio_response_bytes): This is a DISTRIBUTION which measures HTTP response body sizes.
gRPC Request Message Count (istio_request_messages_total): This is a COUNTER incremented for every gRPC message sent from a client.
gRPC Response Message Count (istio_response_messages_total): This is a COUNTER incremented for every gRPC message sent from a server.
```

- 对于TCP流量，Istio生成以下指标：

```
Tcp Bytes Sent (istio_tcp_sent_bytes_total): This is a COUNTER which measures the size of total bytes sent during response in case of a TCP connection.
Tcp Bytes Received (istio_tcp_received_bytes_total): This is a COUNTER which measures the size of total bytes received during request in case of a TCP connection.
Tcp Connections Opened (istio_tcp_connections_opened_total): This is a COUNTER incremented for every opened connection.
Tcp Connections Closed (istio_tcp_connections_closed_total): This is a COUNTER incremented for every closed connection.
```

每个指标都会包含标签，以下为每个Istio指标默认包含的标签：

```
DefaultStatTags=["reporter", "source_namespace","source_workload","source_workload_namespace","source_principal","source_app","source_version","source_app","source_version","source_cluster",destination_namespace","destination_principal","destination_app","destination_version","destination_service","destination_service_name","destination_service_namespace","destination_port","destination_cluster","request_protocol","request_operation","request_host","response_flags","grpc_response_status","connection_security_policy","source_canonical_service","destination_canonical_service","source_canonical_revision","destination_canonical_revision"]
```

标签含义

标签	含义

标签	含义
reporter	请求的上报者，不同的上报者标签值不同： • 如果上报者来自服务器Istio代理，则标签值为 `destination` 。 • 如果上报者来自客户端Istio代理或网关，则标签值为 `source` 。
source_workload	控制源的源工作负载的名称。如果没有采集到标签值，则标签值为 `unknown` 。
source_workload_namespace	源工作负载的名称空间。如果没有采集到标签值，则标签值为 `unknown` 。
source_principal	流量源的对等主体。只有创建了对等身份验证时，才可以采集到该标签的值。
source_app	根据源工作负载的应用程序标签标识源应用程序。如果没有采集到标签值，则标签值为 `unknown` 。
source_version	源工作负载的版本。如果没有采集到标签值，则标签值为 `unknown` 。
source_cluster	源工作负载的集群。
destination_namespace	目标工作负载的名称空间。如果没有采集到标签值，则标签值为 `unknown` 。
destination_workload	目标工作负载的名称。如果没有采集到标签值，则标签值为 `unknown` 。
destination_workload_namespace	目标工作负载的名称空间。如果没有采集到标签值，则标签值为 `unknown` 。
destination_principal	通信目的地的对等主体。只有创建了对等身份验证时，才可以采集到该标签的值。
destination_app	根据目标工作负载的应用程序标签标识目标应用程序。如果没有采集到标签值，则标签值为 `unknown` 。
destination_version	目标工作负载的版本。如果没有采集到标签值，则标签值为 `unknown` 。
destination_service	负责传入请求的目标服务主机。
destination_service_name	目标服务名称。
destination_service_namespace	目标服务的名称空间。
destination_port	目标端口。
destination_cluster	目标工作负载的集群。
request_protocol	请求的协议。
request_operation	请求的操作。
request_host	请求的主机。

标签	含义
response_flags	有关来自代理的响应或连接的其它详细信息。
grpc_response_status	GRPC返回状态。
connection_security_policy	请求的服务身份验证策略。

按照网格级别自定义监控指标

按照网格级别自定义监控指标将会对网格全局进行生效。

1. 开启Prometheus监控。

   > **说明** 如果您已开启Prometheus监控，无需执行此步骤。

   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
   iv. 在网格管理页面选择**网格实例 > 基本信息**。
   v. 在**基本信息**页面单击右上角的**功能设置**。
   vi. 在**功能设置更新**面板选中**开启采集Prometheus监控指标**，选中**启用ARMS或者启用已有Prometheus**，然后单击**确定**。
2. 在网格管理页面选择**可观测性管理 > 自定义Metrics**，在右侧页面单击**创建**。
3. 在**创建**页面设置**命名空间**为istio-system。
4. 单击**inboundSidecar**右侧的图标，单击**新增指标**。
5. 设置**指标名称**为空，单击新增**指标维度**，输入**标签**和**表达式**，以及您想删除的标签。

   设置指标名称为空，表示所有指标都将采集指定的标签。
6. 参考步骤4和步骤5设置outboundSidecar和gateway，然后单击**创建**。

   如果您设置了**inboundSidecar**，没有设置**outboundSidecar**和**gateway**，表示仅入口流量采用自定义标签，出口流量和网关仍然使用默认的方式生成指标。

按照命名空间级别自定义监控指标

仅指定的命名空间会按照设置采集监控指标，其他命名空间仍然按照默认设置进行采集。

1. 关闭Prometheus监控。

   如果您开启了Prometheus监控，然后再自定义监控指标，将会产生两条重复的指标。

   > **说明** 如果您未开启Prometheus监控，无需执行此步骤。

   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
   iv. 在网格管理页面选择**网格实例 > 基本信息**。
   v. 在**基本信息**页面单击右上角的**功能设置**。

vi. 在**功能设置更新**面板取消选中**开启采集Prometheus监控指标**，然后单击**确定**。

2. 在网格管理页面选择**可观测性管理 > 自定义Metrics**，在右侧页面单击**创建**。
3. 在**创建**页面选择**命名空间**，设置**名称**。
4. 单击**inboundSidecar**右侧的图标，单击**新增指标**。
5. 设置**指标名称**，单击新增**指标维度**，输入**标签**和**表达式**，以及您想删除的标签。
6. 参考步骤4和步骤5设置outboundSidecar，然后单击**创建**。

   如果您设置了**inboundSidecar**，没有设置**outboundSidecar**，表示仅入口流量采用自定义标签，出口流量仍然使用默认的方式生成指标。

按照工作负载级别自定义监控指标

仅指定的工作负载会按照设置采集监控指标，其他工作负载仍然按照默认设置进行采集。

1. 关闭Prometheus监控。

   如果您开启了Prometheus监控，然后再自定义监控指标，将会产生两条重复的指标。

   > **说明** 如果您未开启Prometheus监控，无需执行此步骤。

   i. 登录ASM控制台。
   ii. 在左侧导航栏，选择**服务网格 > 网格管理**。
   iii. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
   iv. 在网格管理页面选择**网格实例 > 基本信息**。
   v. 在**基本信息**页面单击右上角的**功能设置**。
   vi. 在**功能设置更新**面板取消选中**开启采集Prometheus监控指标**，然后单击**确定**。
2. 在网格管理页面选择**可观测性管理 > 自定义Metrics**，在右侧页面单击**创建**。
3. 在**创建**页面选择**命名空间**，设置**名称**，单击新增**匹配标签**，设置**名称**为app，**值**为应用的名称。

   ASM将会根据设置的值匹配应用，表示自定义指标仅对该应用生效。
4. 单击**inboundSidecar**右侧的图标，单击**新增指标**。
5. 设置**指标名称**，单击新增**指标维度**，输入**标签**和**表达式**，以及您想删除的标签。
6. 参考步骤4和步骤5设置outboundSidecar，然后单击**创建**。

   如果您设置了**inboundSidecar**，没有设置**outboundSidecar**，表示仅入口流量采用自定义标签，出口流量仍然使用默认的方式生成指标。