ALIBABA CLOUD

阿里云

服务网格

最佳实践

文档版本：20220712

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.工作负载

1.1. 通过ASM入口网关实现HTTP请求网格内gRPC服务

ASM入口网关支持协议转码的能力，使用户及其客户端可以使用HTTP/JSON访问服务网格内的gRPC服务。本文介绍如何通过ASM入口网关实现HTTP请求网格内gRPC服务。

前提条件

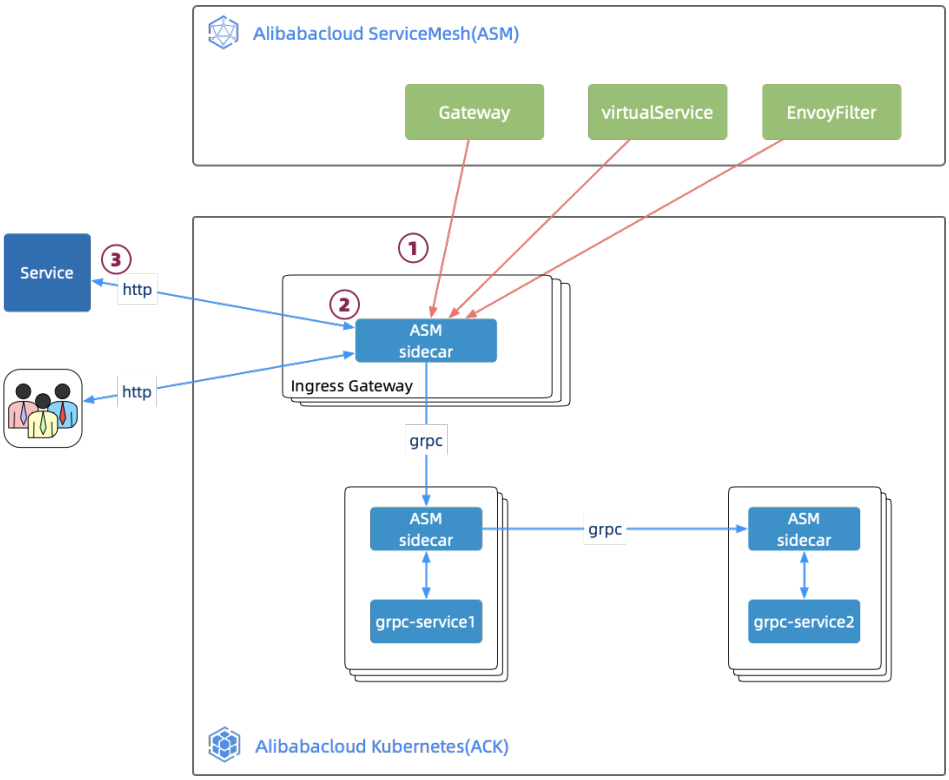
- 已安装自动化生成EnvoyFilter的工具grpc-transcoder。更多信息，请参见grpc-transcoder。
- 已安装Protocol工具。更多信息，请参见Protocol Buffers v3.14.0。

背景信息

Envoy作为服务网格ASM数据平面的Proxy组件，内置了多种HTTP扩展过滤器，包括HTTP到gRPC的转码器。为了启用这个过滤器，Envoy定义了相应的过滤器协议，关于过滤器协议的详细信息，请参见过滤器协议。为此，ASM的控制平面需要定义一个EnvoyFilter来声明某个阶段启用这个过滤器，然后下发这个EnvoyFilter在指定环节启用转码器。

转码流程

ASM入口网关可将HTTP/JSON转码为gRPC。下图为完整的HTTP请、gRPC转码以及gRPC请求流程。

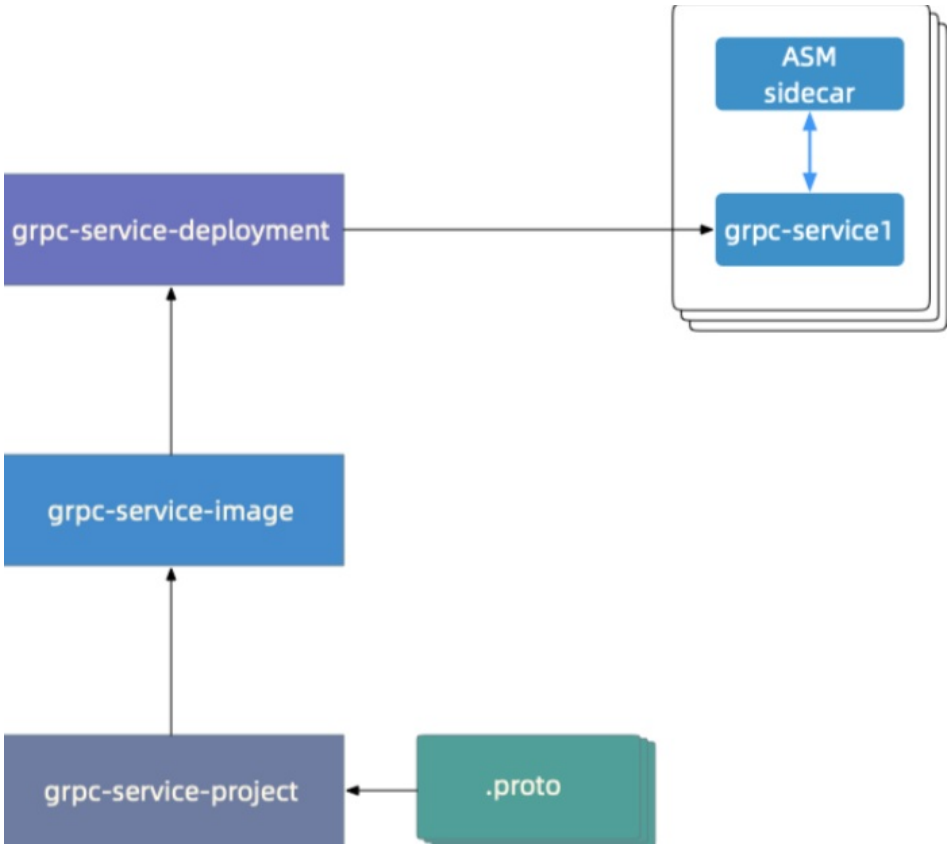


序号	说明
①	ASM控制平面下发用于gRPC转码的EnvoyFilter，用于路由到gRPC服务端口的规则配置Gateway和VirtualService到ASM入口网关，入口网关接收后即时加载生效。

序号	说明
②	入口网关收到用户或其客户端HTTP协议的请求后，将进行路由规则匹配和协议转换，然后以gRPC协议请求服务网格内的gRPC服务。
③	入口网关收到后端服务的gRPC响应，再将其转换为HTTP的响应返回给请求方。

步骤一：补充转码声明

gRPC服务是从protobuf格式的gRPC服务协议文件*.proto*的定义开始的。grpc-service-project封装gRPC接口，后续步骤包括构建镜像，编写Deployment，最终通过ASM将gRPC服务以Pod的形式部署到ACK集群。



为了支持HTTP转码gRPC，您需要在*.proto*文件的实践方法中补充以下支持转码的声明。

```
option(google.api.http) = {
  get: "/v1/talk/{data}/{meta}"
};
```

以hello-servicemesh-rpc示例中的*.proto*为例，以下为补充了转码声明后的*.proto*。关于hello-servicemesh-grpc的详细信息，请参见hello-servicemesh-grpc。

```
import "google/api/annotations.proto";
service LandingService {
  //Unary RPC
  rpc talk (TalkRequest) returns (TalkResponse) {
    option(google.api.http) = {
      get: "/v1/talk/{data}/{meta}"
    };
  }
...
}
message TalkRequest {
  string data = 1;
  string meta = 2;
}
```

步骤二：生成Proto Descriptors文件

在Protocol工具上执行以下命令，使用Protoc命令从*landing.proto*生成*landing.proto-descriptor*文件。

```
# https://github.com/AliyunContainerService/hello-servicemesh-grpc
proto_path={path/to/hello-servicemesh-grpc}/grpc/proto
#  https://github.com/googleapis/googleapis/tree/master/
proto_dep_path={path/to/googleapis}
protoc \
    --proto_path=${proto_path} \
    --proto_path=${proto_dep_path} \
    --include_imports \
    --include_source_info \
    --descriptor_set_out=landing.proto-descriptor \
    "${proto_path}"/landing.proto
```

步骤三：创建EnvoyFilter的YAML文件

在本地的运行窗口输入以下调用接口的请求，将自动使用grpc-transcoder工具生成EnvoyFlter的YAML文件。

```
grpc-transcoder \
--version 1.7 \
--service_port 9996 \
--service_name grpc-server-svc \
--proto_pkg org.feuyeux.grpc \
--proto_svc LandingService \
--descriptor landing.proto-descriptor
```

- `version`：ASM实例的版本。
- `service_port`：gRPC服务端口。
- `service_name`：gRPC服务名称。
- `proto_pkg`：gRPC服务*.proto*包名的定义。
- `proto_svc`：gRPC服务*.proto*中服务名的定义。
- `descriptor`：*Proto Descriptors*文件本地路径。

使用上述请求自动生成以下EnvoyFilter，将以下内容复制到*grpc-transcoder-envoyfilter.yaml*。

```
#Generated by ASM(http://servicemesh.console.aliyun.com)
#GRPC Transcoder EnvoyFilter[1.7]
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  name: grpc-transcoder-grpc-server-svc
spec:
  workloadSelector:
    labels:
      app: istio-ingressgateway
  configPatches:
    - applyTo: HTTP_FILTER
      match:
        context: GATEWAY
        listener:
          portNumber: 9996
          filterChain:
            filter:
              name: "envoy.filters.network.http_connection_manager"
              subFilter:
                name: "envoy.filters.http.router"
        proxy:
          proxyVersion: ^1\.7.*
      patch:
        operation: INSERT_BEFORE
        value:
          name: envoy.grpc_json_transcoder
          typed_config:
            '@type': type.googleapis.com/envoy.extensions.filters.http.grpc_json_transcoder.v
3.GrpcJsonTranscoder
            proto_descriptor_bin: Ctl4ChVnb29nbGUvYXBpL2h0dHAucHJ...
            services:
            - org.feuyeux.grpc.LandingService
            print_options:
              add_whitespace: true
              always_print_primitive_fields: true
              always_print_enums_as_ints: false
              preserve_proto_field_names: false
```

步骤四：在ASM控制台创建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. 在网格详情页面左侧导航栏选择**插件中心 > Envoy过滤器**，然后在右侧页面单击**使用YAML创建**。
5. 选择**命名空间**，将步骤三：创建EnvoyFilter的YAML文件的*grpc-transcoder-envoyfilter.yaml*内容复制到文本框，然后单击**创建**。

步骤五：验证Envoy配置

依次执行以下命令，验证Envoy动态配置中是否包含解码器GrpcJsonTranscoder。

```
#获取入口网关POD名称
ingressgateway_pod=$(kubectl get pod -l app="istio-ingressgateway" -n istio-system -o jsonpat
h='{.items[0].metadata.name}')
#时间戳
timestamp=$(date "+%Y%m%d-%H%M%S")
#获取envoy动态配置并保存到dynamic_listeners-"$timestamp".json
kubectl -n istio-system exec $ingressgateway_pod \
  -c istio-proxy \
  -- curl -s "http://localhost:15000/config_dump?dynamic_listeners" >dynamic_listeners-"$time
stamp".json
#确认配置中存在GrpcJsonTranscoder
grep -B3 -A7 GrpcJsonTranscoder dynamic_listeners-"$timestamp".json
```

输出配置中应包含以下内容，说明Envoy动态配置中包含解码器GrpcJsonTranscoder。

```
{
  "name": "envoy.grpc_json_transcoder",
  "typed_config": {
    "@type": "type.googleapis.com/envoy.extensions.filters.http.grpc_json_transcoder.v3.GrpcJ
sonTranscoder",
    "services": [
      "org.feuyeux.grpc.LandingService"
    ],
    "print_options": {
      "add_whitespace": true,
      "always_print_primitive_fields": true
    },
    ...
```

步骤六：验证HTTP请求网格内gRPC服务

*.proto*文件定义了gRPC服务的请求接口和响应声明，根据请求接口调用gRPC服务，将返回定义的响应声明。以下为*.proto*文件定义的请求接口和响应声明：

- *.proto*文件定义的请求接口

```
rpc talk (TalkRequest) returns (TalkResponse) {
  option(google.api.http) = {
    get: "/v1/talk/{data}/{meta}"
  };
}
```

- *.proto*文件定义的响应声明

```
message TalkResponse {
  int32 status = 1;
  repeated TalkResult results = 2;
}
message TalkResult {
  //timestamp
  int64 id = 1;
  //enum
  ResultType type = 2;
  // id:result uuid
  // idx:language index
  // data: hello
  // meta: serverside language
  map<string, string> kv = 3;
}
enum ResultType {
  OK = 0;
  FAIL = 1;
}
```

执行以下命令，通过HTTP请求入口网关调用gRPC服务。

```
#获取入口网关IP
INGRESS_IP=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBal
ancer.ingress[0].ip}')
#http请求入口网关的9996端口,/v1/talk/{data}/{meta}路径
curl http://$INGRESS_IP:9996/v1/talk/0/java
```

预期输出：

```
{
 "status": 200,
 "results": [
  {
   "id": "699882576081691",
   "type": "OK",
   "kv": {
    "data": "Hello",
    "meta": "JAVA",
    "id": "8c175d5c-d8a3-4197-a7f8-6e3e0ab1fe59",
    "idx": "0"
   }
  }
 ]
}
```

通过HTTP请求入口网关，调用gRPC服务，返回结果与预期定义相同。说明调用gRPC服务成功，转码成功。

1.2. 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。

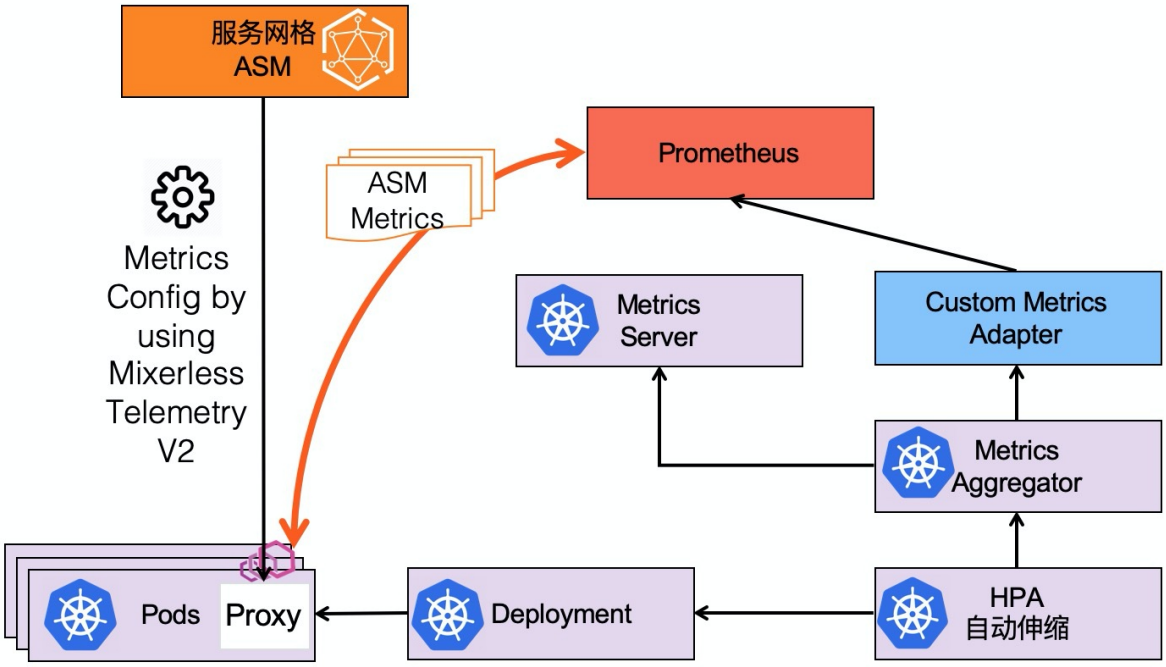
背景信息

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. 在网格详情页面左侧导航栏选择**网格实例 > 基本信息**，然后在右侧页面单击**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**，选中**启用已有Prometheus**，输入Prometheus地址，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.u
rl=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
```

```
            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.yam*l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='{.items[
0].metadata.name}')
kubectl -n test exec -it ${loadtester} -c loadtester -- hey -z 5s -c 10 -q 2 http://p
odinfo.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。

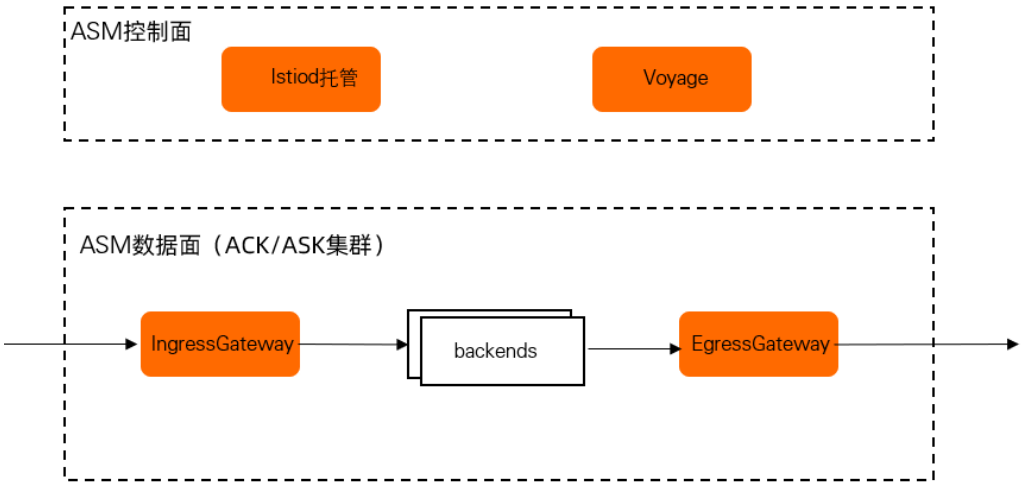
2.ASM网关

2.1. 配置高性能和高可用的ASM网关

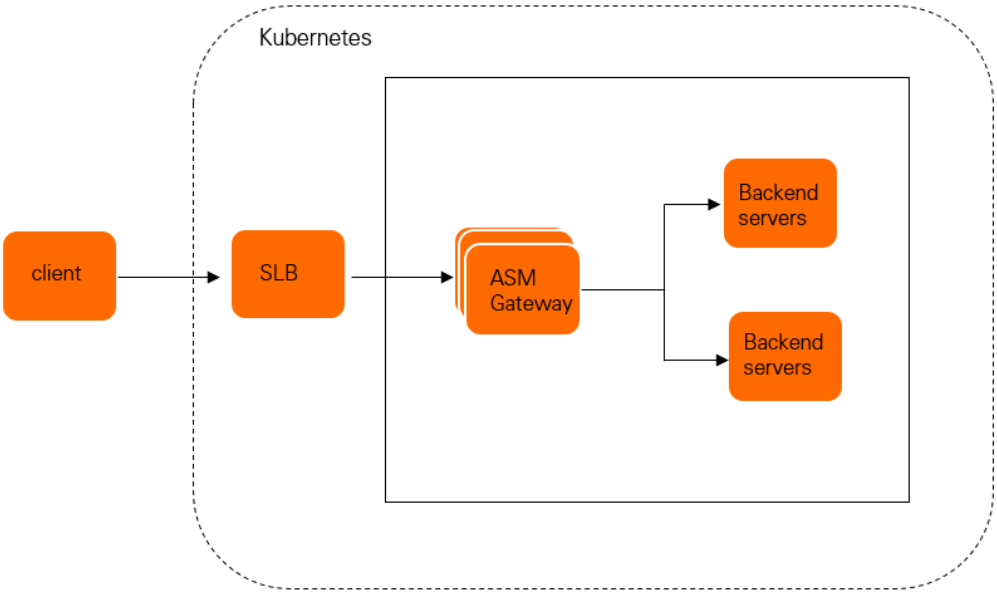
配置高性能和高可用的ASM网关，可以保障业务的连续性，提高用户体验。本文介绍如何配置ASM网关实现业务请求的高性能和高可用。

背景信息

ASM网关是Istio的一个重要组件，主要用于管理服务的入口和出口流量。



当创建一个ASM网关时，同时会在Kubernetes集群的istio-system命名空间下创建一个对应的istio-ingressgateway Deployment，这个Deployment会关联一个SLB，相关实例作为SLB的后端实例。



如上图所示，可以看到业务请求流量会经过的路径。从高性能、高可用角度来说，请求经过的每个路径都会影响实际请求的响应时间以及可用性。接下来将从SLB、ASM网关两个关键节点介绍如何实现业务请求的高性能和高可用。

高性能

- 业务多地域部署，客户端就近访问
  - ASM支持管理多个集群，多地域场景支持地域就近访问，以及区域感知的负载均衡。具体操作，请参见通过ASM实现跨地域容灾和流量负载均衡。
  - 结合DNS智能解析，可以将域名解析到客户端最近的IP（也就是SLB）。
- 使用SLB访问ASM网关实例
  - 创建Kubernetes集群时，如果您使用的是Terway CNI，SLB将会直接对接Gateway Pod网络。如果您使用的是Flannel CNI，则会通过Node节点的NodePort多一层网络转发。如果您想提高性能，推荐使用Terway。关于Terway和Flannel对比，请参见Terway与Flannel对比。
  - 当网关流量较大，单个SLB无法满足性能需求时，您可以配置多个SLB关联同一个网关，从而可以使用多个SLB访问ASM网关。具体操作，请参见使用多个SLB访问ASM网关。
- TLS加速
  商业版的ASM网关支持基于Intel的MultiBuffer实现HTTPS请求加速，实测QPS提升80%的性能。具体操作，请参见启用Multi-Buffer实现TLS加速。

高可用

- 业务多地域部署，多地域多活，实现跨地域容灾
  关于跨地域容灾的具体操作，请参见通过ASM实现跨地域容灾和流量负载均衡。
- SLB高可用
  ASM支持配置多个SLB实例关联同一网关，当其中一个SLB故障时，您可以使用另一个SLB。具体操作，请参见使用多个SLB访问ASM网关。
- ASM网关节点级别的高可用
  您可以将ASM网关的Pod分布到不同的Node节点或者可用区，增强网关高可用性。具体操作，请参见增强ASM网关高可用性。
- 业务服务部署优雅上下线
  业务服务对应的Deployment可以配置PreStop脚本来实现Pod的优雅终止，避免请求失败或丢失。具体操作，请参见方案二：配置Sidecar代理生命周期。
- ASM网关的优雅上下线
  使用ASM网关的优雅上下线功能后，当对网关进行扩缩容时，现有连接在一定时间内仍能正常传输，流量将不会有损失。具体操作，请参见使用SLB优雅下线功能避免流量损失。

3.流量管理

3.1. 基于ASM完成蓝绿和灰度发布

本文介绍如何通过ASM定义的虚拟服务和目标规则配合完成蓝绿和灰度发布。

前提条件

- 创建至少一个ASM实例，详情请参见创建ASM实例。
- 添加至少一个ACK集群到ASM实例中，详情请参见添加集群到ASM实例。
- 部署Bookinfo应用示例到ASM实例下管理的ACK集群中，详情请参见部署应用到ASM实例。
- 在ASM实例的ACK集群中部署入口网关，详情请参见添加入口网关服务。

创建目标规则

针对上述部署的Bookinfo示例，创建所需的目标规则，详情请参见管理目标规则。目标规则的配置信息如下。

```
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
```

创建虚拟服务

针对上述部署的Bookinfo示例，创建所需的虚拟服务，详情请参见管理虚拟服务。虚拟服务的配置信息如下。

```
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
```

蓝绿发布v2版本

创建目标规则和虚拟服务后，reviews的v2版本已经在运行，但还没有流量切换到v2版本，因此需要用蓝绿部署的方式，让v2版本上线。

针对上述部署的Bookinfo示例，创建蓝绿发布v2版本的虚拟服务，详情请参见管理虚拟服务。虚拟服务的配置信息如下。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
```

部署完成后刷新页面，您可以看到book reviews已经有了评分，而且评分的星星为黑色。

按权重灰度发布v3版本

您可以让v2、v3版本同时在线，且两个版本各处理50%的流量。

针对上述部署的Bookinfo示例，创建灰度发布v3版本的虚拟服务，详情请参见管理虚拟服务。虚拟服务的配置信息如下。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
      weight: 50
    - destination:
        host: reviews
        subset: v3
      weight: 50
```

部署完成后刷新页面，可以看到book reviews一栏v2，v3版本随机出现，v3版本的评分的星星为红色。

按请求内容灰度发布v3版本

只按照流量来简单的灰度还是不能满足很多场景，您还可以按照用户来灰度，不同的用户访问不同的页面。

针对上述部署的Bookinfo示例，创建灰度发布的虚拟服务，详情请参见管理虚拟服务。虚拟服务的配置信息如下。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v3
  - route:
    - destination:
        host: reviews
        subset: v2
```

部署完成后刷新页面，评分区的星星始终为黑色。此时，您可以单击右上角Sign in，以用户名jason登录（不需要输入密码），您可以看到评分区的星星变成红色。

> **说明** 登录并访问后端服务时，会带上 `end-user=XXX` 的http header，使用jason登录后，匹配到了YAML的规则，使得流量被引导到了v3版本的reviews。

3.2. 基于ASM实现无侵入全链路AB测试

服务网格ASM可以为运行其上的微服务提供无侵入式的流量治理能力，但是仅通过服务网格ASM无法实现无侵入全链路AB测试。引入WebAssembly（WASM）后，无需修改微服务代码，即可实现无侵入全链路AB测试。本文介绍如何基于ASM实现无侵入全链路AB测试。

前提条件

- 已创建ASM实例。具体操作，请参见创建ASM实例。

  > **说明** 请确保您的ASM实例为1.8及以上版本。

- 已添加集群到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。具体操作，请参见添加入口网关服务。
- 已创建容器镜像仓库，并获取镜像仓库地址和登录信息。具体操作，请参见使用企业版实例推送和拉取镜像。

背景信息

WebAssembly（WASM）是一种有效的可移植二进制指令格式，可以用于扩展ASM数据平面的功能。关于无侵入全链路AB测试和WASM开发的更多信息，请参见基于WASM的无侵入式全链路A/B Test实践。

> **说明** 本文的镜像仓库仅供参考，请根据镜像脚本自行构建和推送镜像至自建仓库。关于镜像脚本的具体信息，请参见hello-servicemesh-grpc。

步骤一：启用使用WASM Filter的功能

1. 使用以下内容，创建名为*runtime-config.json*的文件。

```
{
  "type": "envoy_proxy",
  "abiVersions": [
    "v0-541b2c1155fffb15ccde92b8324f3e38f7339ba6",
    "v0-097b7f2e4cc1fb490cc1943d0d633655ac3c522f",
    "v0-4689a30309abf31aee9ae36e73d34b1bb182685f",
    "v0.2.1"
  ],
  "config": {
    "rootIds": [
      "propaganda_filter_root"
    ]
  }
}
```

2. 执行以下命令，上传WASM Filter到ACR仓库。

```
oras push ${WASM_REGISTRY}/propagate_header:0.0.1 \
  --manifest-config \
  --runtime-config.json:application/vnd.module.wasm.config.v1+json \
  ${WASM_IMAGE}:application/vnd.module.wasm.content.layer.v1+wasm
```

   - `WASM_REGISTRY`：容器镜像仓库地址。
   - `WASM_IMAGE`：当前路径下的WASM文件名。
   - `runtime-config.json`：当前路径下的运行时配置文件。

3. 启用使用WASM Filter的功能。

   i. 执行以下命令，确认Alinyun CLI的版本。

   Aliyun CLI必须为3.0.73及以上版本。

   ```
   aliyun version
   ```

   ii. 执行以下命令，启用使用WASM Filter的功能。

   ```
   aliyun servicemesh UpdateMeshFeature --ServiceMeshId=xxxxxx --WebAssemblyFilterEnable
   d=true
   ```

4. 执行以下命令，验证服务网格实例的开启状态。

```
aliyun servicemesh DescribeServiceMeshDetail \
  --ServiceMeshId $MESH_ID |
  jq '.ServiceMesh.Spec.MeshConfig.WebAssemblyFilterDeployment'
```

   预期输出：

```
{
  "Enabled": true
}
```

5. 执行以下命令，验证asmwasm-cache的状态。

   WASM Filter功能开启后，会在ACK集群的每个节点上创建一个名为asmwasm-cache的DaemonSet。

```
kubectl get daemonset -n istio-system
```

预期输出：

```
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR
AGE
asmwasm-cache   4         4         4       4            4           kubernetes.io/os=lin
ux   34
```

步骤二：部署实验资源

1. 使用以下内容，在*kube*目录下创建名为*hello.yaml*的文件。

   该YAML文件包含v1和v2版本的Hello1、Hello2和Hello3应用。

   > **说明** 您也可以在GitHub上获取Hello应用的YAML文件。详细信息，请参见kube。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello1-deploy-v1
  labels:
    app: hello1-deploy-v1
    service: hello1-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello1-deploy-v1
      service: hello1-deploy
      version: v1
  template:
    metadata:
      labels:
        app: hello1-deploy-v1
        service: hello1-deploy
        version: v1
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v1-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello2-svc"
          ports:
            - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello1-deploy-v2
  labels:
    app: hello1-deploy-v2
```

```
    service: hello1-deploy
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello1-deploy-v2
      service: hello1-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello1-deploy-v2
        service: hello1-deploy
        version: v2
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v2-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello2-svc"
          ports:
            - containerPort: 8001apiVersion: v1
---
kind: Service
metadata:
  name: hello1-svc
  labels:
    app: hello1-svc
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello1-deployapiVersion: apps/v1
kind: Deployment
metadata:
  name: hello2-deploy-v1
  labels:
    app: hello2-deploy-v1
    service: hello2-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello2-deploy-v1
      service: hello2-deploy
      version: v1
  template:
    metadata:
      labels:
        app: hello2-deploy-v1
        service: hello2-deploy
```

```
        service: hello2-deploy
        version: v1
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v1-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello3-svc"
          ports:
            - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello2-deploy-v2
  labels:
    app: hello2-deploy-v2
    service: hello2-deploy
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello2-deploy-v2
      service: hello2-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello2-deploy-v2
        service: hello2-deploy
        version: v2
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v2-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
          env:
            - name: HTTP_HELLO_BACKEND
              value: "hello3-svc"
          ports:
            - containerPort: 8001apiVersion: v1
---
kind: Service
metadata:
  name: hello2-svc
  labels:
    app: hello2-svc
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello2-deployapiVersion: apps/v1
```

```
---
kind: Deployment
metadata:
  name: hello3-deploy-v1
  labels:
    app: hello3-deploy-v1
    service: hello3-deploy
    version: v1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello3-deploy-v1
      service: hello3-deploy
      version: v1
  template:
    metadata:
      labels:
        app: hello3-deploy-v1
        service: hello3-deploy
        version: v1
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v1-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v1:1.0.0
          ports:
            - containerPort: 8001
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello3-deploy-v2
  labels:
    app: hello3-deploy-v2
    service: hello3-deploy
    version: v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello3-deploy-v2
      service: hello3-deploy
      version: v2
  template:
    metadata:
      labels:
        app: hello3-deploy-v2
        service: hello3-deploy
        version: v2
    spec:
      serviceAccountName: http-hello-sa
      containers:
        - name: hello-v2-deploy
          image: registry.cn-beijing.aliyuncs.com/asm_repo/http_springboot_v2:1.0.0
```

```
          ports:
            - containerPort: 8001apiVersion: v1
---
kind: Service
metadata:
  name: hello3-svc
  labels:
    app: hello3-svc
spec:
  ports:
    - port: 8001
      name: http
  selector:
    service: hello3-deployapiVersion: v1
---
kind: ServiceAccount
metadata:
  name: http-hello-sa
  labels:
    account: http-hello-deploy
```

2. 使用以下内容，在*mesh*目录下创建名为*mesh.yaml*的文件。

> **说明** 您也可以在GitHub上获取Ingress Gateway、DestinationRule和VirtualService的YAML文件。详细信息，请参见mesh。

YAML文件中包含DestinationRule、VirtualService和Ingress Gateway。
在DestinationRule设置以下子集：

- 设置v1版本的Hello1应用为hello1v1，v2版本的Hello1应用为hello1v2。
- 设置v1版本的Hello2应用为hello2v1，v2版本的Hello2应用为hello2v2。
- 设置v1版本的Hello3应用为hello3v1，v2版本的Hello3应用为hello3v2。

在VirtualService中设置了以下路由规则：

- 仅包含route-v且版本为v2的请求才能路由到hello1v2，否则路由到hello1v1。
- 仅包含route-v且版本为hello2v2的请求才能路由到hello2v2，否则路由到hello2v1。
- 仅包含route-v且版本为hello3v2的请求才能路由到hello3v2，否则路由到hello3v1。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello1-dr
spec:
  host: hello1-svc
  subsets:
    - name: hello1v1
      labels:
        version: v1
    - name: hello1v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
```

```
metadata:
  name: hello-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 8001
        name: http
        protocol: HTTP
      hosts:
        - "*"
---
# https://istio.io/latest/docs/reference/config/networking/virtual-service/
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello1-vs
spec:
  hosts:
    - "*"
  gateways:
    - hello-gateway
  #  - mesh
  http:
    - name: hello1-v1-route
      match:
        - headers:
            route-v:
              exact: v2
      route:
        - destination:
            host: hello1-svc
            subset: hello1v2
    - route:
        - destination:
            host: hello1-svc
            subset: hello1v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello2-dr
spec:
  host: hello2-svc
  subsets:
    - name: hello2v1
      labels:
        version: v1
    - name: hello2v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  name: hello2-vs
spec:
  hosts:
    - hello2-svc
  http:
  - name: hello2-v2-route
    match:
    - headers:
        route-v:
          exact: hello2v2
    route:
    - destination:
        host: hello2-svc
        subset: hello2v2
  - route:
    - destination:
        host: hello2-svc
        subset: hello2v1
---
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: hello3-dr
spec:
  host: hello3-svc
  subsets:
    - name: hello3v1
      labels:
        version: v1
    - name: hello3v1
      labels:
        version: v2
    - name: hello3v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: hello3-vs
spec:
  hosts:
  - hello3-svc
  http:
  - match:
    - headers:
        route-v:
          exact: hello3v2
    route:
    - destination:
        host: hello3-svc
        subset: hello3v2
  - route:
    - destination:
        host: hello3-svc
```

```
        subset: hello3v1
```

3. 执行以下命令，部署Hello应用、Ingress Gateway、VirtualService和DestinationRule。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
alias m="kubectl --kubeconfig $MESH_CONFIG"
k -n "$NS" apply -f kube/kube.yaml
m -n "$NS" apply -f mesh/mesh.yaml
```

步骤三：部署自定义资源ASMFilterDeployment

1. 在ACK集群中创建用来访问镜像仓库的Secret。

   关于Secret的详细信息，请参见Secret。

   i. 使用以下内容，创建名为*myconfig.json*的文件。

```
{
    "auths":{
        "**********.cn-hangzhou.cr.aliyuncs.com":{
            "username":"*****username*****",
            "password":"*****password*****"
        }
    }
}
```

   - `**********.cn-hangzhou.cr.aliyuncs.com`：镜像仓库地址。
   - `username`：镜像仓库用户名。
   - `password`：镜像仓库密码。

   ii. 执行以下命令，创建Sercet。

> 说明 Secret名字必需为asmwasm-cache，命名空间为istio-system。

```
kubectl create secret generic asmwasm-cache -n istio-system --from-file=.dockerconfig
json=myconfig.json --type=kubernetes.io/dockerconfigjson
```

2. 部署ASMFilterDeployment。

i. 使用以下内容，创建名为*hello1-afd.yaml*的文件。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMFilterDeployment
metadata:
  name: hello1-propagate-header
spec:
  workload:
    kind: Deployment
    labels:
      app: hello1-deploy-v2
      version: v2
  filter:
    patchContext: 'SIDECAR_OUTBOUND'
    parameters: '{"head_tag_name": "route-v", "head_tag_value": "hello2v2"}'
    image: 'wasm-repo-registry.cn-beijing.cr.aliyuncs.com/asm_wasm/propagate_header:0
.0.1'
    rootID: 'propaganda_filter_root'
    id: 'hello1-propagate-header'
```

- `workload`下的参数解释：
  a. `kind`：目标工作负载的类型。
  b. `labels`：筛选的条件。
- `filter`下的参数解释：
  a. `patchContext`：生效的上下文阶段。
  b. `parameters`：运行WASM Filter所需的配置参数。
  c. `image`：WASM Fitler对应的镜像仓库地址。
  d. `rootID`：WASM Filter扩展插件对应的RootID。
  e. `id`：该WASM Filter的唯一ID。

ii. 使用以下内容，创建名为*hello2-afd.yaml*的文件。

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: ASMFilterDeployment
metadata:
  name: hello2-propagate-header
spec:
  workload:
    kind: Deployment
    labels:
      app: hello2-deploy-v2
      version: v2
  filter:
    patchContext: 'SIDECAR_OUTBOUND'
    parameters: '{"head_tag_name": "route-v", "head_tag_value": "hello3v2"}'
    image: 'wasm-repo-registry.cn-beijing.cr.aliyuncs.com/asm_wasm/propagate_header:0
.0.1'
    rootID: 'propaganda_filter_root'
    id: 'hello2-propagate-header'
```

- `workload`下的参数解释：
  a. `kind`：目标工作负载的类型。
  b. `labels`：筛选的条件。
- `filter`下的参数解释：
  a. `patchContext`：生效的上下文阶段。
  b. `parameters`：运行WASM Filter所需的配置参数。
  c. `image`：WASM Fitler对应的镜像仓库地址。
  d. `rootID`：WASM Filter扩展插件对应的RootID。
  e. `id`：该WASM Filter的唯一ID。

iii. 执行以下命令，部署ASMFilterDeployment。

```
alias m="kubectl --kubeconfig $MESH_CONFIG"
m apply -f hello1-afd.yaml -n "$NS"
m apply -f hello2-afd.yaml -n "$NS"
```

3. 执行以下命令，验证ASMFilterDeployment部署情况。

ASMFilterDeployment部署后，会自动生成EnvoyFilter。

```
alias m="kubectl --kubeconfig $MESH_CONFIG"
m get envoyfilter -n "$NS"
m get ASMFilterDeployment -n "$NS"
```

预期输出：

```
NAME                      AGE
hello1-propagate-header   1s
hello2-propagate-header   0s
NAME                      STATUS      REASON   AGE
hello1-propagate-header   Available            1s
hello2-propagate-header   Available            1s
```

验证AB测试

执行以下命令，验证AB测试。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
ingressGatewayIp=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.l
oadBalancer.ingress[0].ip}')
for j in {1..3}; do
  curl -H "route-v:v2" "http://$ingressGatewayIp:8001/hello/eric"
  echo
done
```

预期输出：

```
Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.1
7.68.208
Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.1
7.68.208
Bonjour eric@hello1:172.17.68.239<Bonjour eric@hello2:172.17.68.209<Bonjour eric@hello3:172.1
7.68.208
```

可以看到，当接收到route-v且版本为v2的请求时，该请求路由到hello1v2、hello2v2和hello3v2。

相关信息

如果您没有得到预期结果，您可以使用以下脚本检查服务的负载日志：

- 检查Envoy日志

```
alias k="kubectl --kubeconfig $USER_CONFIG"
hello1_v2_pod=$(k get pod -l app=hello1-deploy-v2 -n "$NS" -o jsonpath={.items..metadata.na
me})
# 修改envoy日志级别为info
k -n "$NS" exec "$hello1_v2_pod" -c istio-proxy -- curl -XPOST -s "http://localhost:15000/l
ogging?level=info"
# 打印envoy日志
k -n "$NS" logs -f deployment/hello1-deploy-v2 -c istio-proxy
```

- 检查Hello服务日志

```
lias k="kubectl --kubeconfig $USER_CONFIG"
k -n "$NS" logs -f deployment/hello2-deploy-v1 -c hello-v1-deploy
```

3.3. 使用服务网格ASM结合KubeVela实现渐进式灰度发布

KubeVela是一个开箱即用的、现代化的应用交付与管理平台。使用服务网格ASM结合KubeVela可以实现应用的渐进式灰度发布，达到平缓升级应用的目的。本文介绍如何使用服务网格ASM结合KubeVela实现渐进式灰度发布。

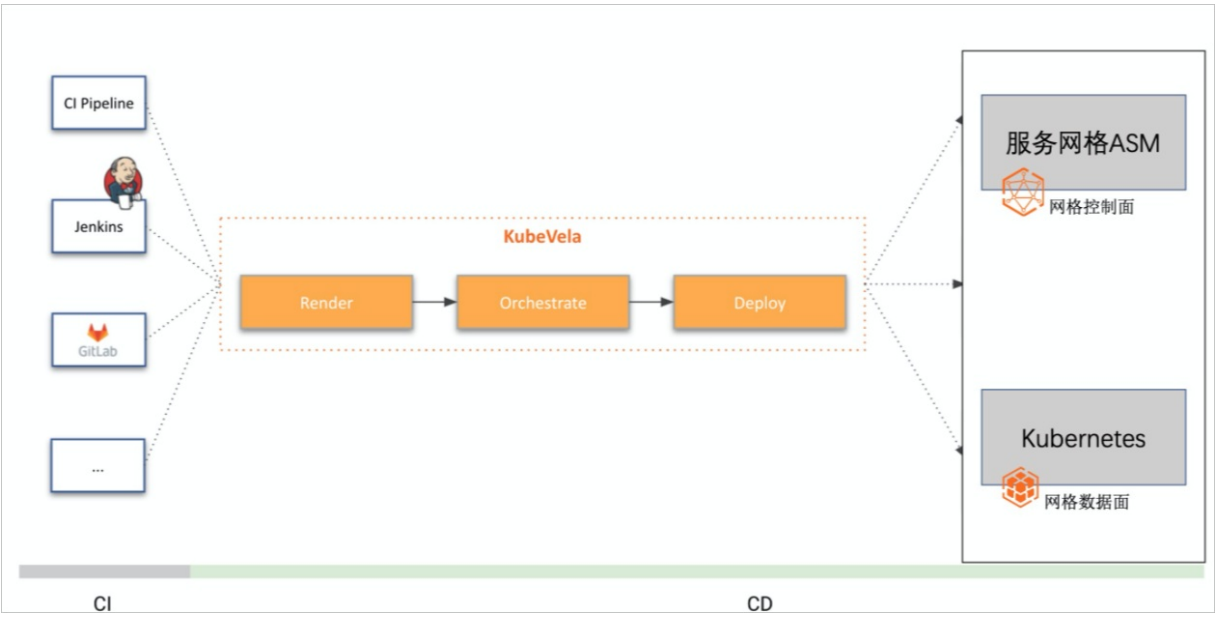
前提条件

- 创建ASM实例，且ASM实例为v1.9.7.93-g7910a454-aliyun以上版本。具体操作，请参见创建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。
- 使用kubectl连接集群。具体操作，请参见通过kubectl工具连接集群。
- 已安装KubeVela CLI。具体操作，请参见安装KubeVela CLI。
- 已开启数据面集群Kubernetes API访问Istio资源。具体操作，请参见使用集群Kubernetes API访问Istio资源（旧版本）。

背景信息

KubeVela是一个开箱即用的、现代化的应用交付与管理平台，能够简化面向混合环境的应用交付过程；同时又足够灵活可以随时满足业务不断高速变化所带来的迭代压力。KubeVela后的应用交付模型Open Application Model（OAM）是一个从设计与实现上都高度可扩展的模型，具有完全以应用为中心、可编程式交付工作流、与基础设施无关的特点。更多信息，请参见基于Istio的渐进式发布。



操作前说明

本文涉及的所有文件都在asm_kubevela文件夹中，在进行操作前，您需要下载并解压asm_kubevela至本地。

asm_kubevela文件夹包含*application.yaml*、*application_rollback.yaml*、*application_rollout-v2.yaml*、*canary-rollout-wf-def.yaml*、*rollback-wf-def.yaml*和*traffic-trait-def.yaml*文件。其中：

- 步骤三中将使用到*canary-rollout-wf-def.yaml*、*rollback-wf-def.yaml*和*traffic-trait-def.yaml*文件。
- 步骤四将使用到*application.yaml*文件。
- 步骤五将使用到*application_rollout-v2.yaml*文件。
- 步骤六将使用到*application_rollback.yaml*文件。

步骤一：安装KubeVela

1. 登录容器服务管理控制台。
2. 在控制台左侧导航栏中，选择**市场 > 应用市场**。
3. 在**应用目录**页面搜索ack-kubevela，然后单击ack-kubevela。
4. 在ack-kubevela详情页面**创建**面板选择集群，然后单击**创建**。

步骤二：开启自动注入

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**网格实例 > 全局命名空间**。
5. 在**命名空间**页面单击default右侧**自动注入**列下的**启用Sidecar自动注入**。
6. 在**确认**对话框单击**确定**。

步骤三：部署KubeVela配置文件

部署KubeVela配置文件，用于实现KubeVela和服务网格ASM流量治理规则的集成。

> **说明** 此操作前，请务必开启数据面集群Kubernetes API访问Istio资源，否则会报错。具体操作，请参见使用集群Kubernetes API访问Istio资源（旧版本）。

进入asm_kubevela文件夹中，执行以下命令，逐个部署KubeVela配置文件。

```
kubectl apply -f rollback-wf-def.yaml
```

```
kubectl apply -f canary-rollout-wf-def.yaml
```

```
kubectl apply -f traffic-trait-def.yaml
```

步骤四：部署应用和网关

1. 进入asm_kubevela文件夹中，执行以下命令，部署Bookinfo应用。

   ```
   kubectl apply -f application.yaml
   ```

   *application.yaml*文件中为review服务配置 `traits` 字段下type为*canary-traffic*，表示配置了渐进式流量发布的运维特征。
2. 在ASM控制台中部署网关规则和虚拟服务。
   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. 部署网关规则。

a. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，在右侧页面单击**使用YAML创建**。

b. 在**创建**页面设置命名空间为default，复制以下内容到文本框中，然后单击**创建**。

```
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:
    - "*"
```

v. 部署虚拟服务。

a. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，在右侧页面单击**使用YAML创建**。

b. 在**创建**页面设置命名空间为default，复制以下内容到文本框中，然后单击**确定**。

```
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
```

vi. 访问Bookinfo应用。

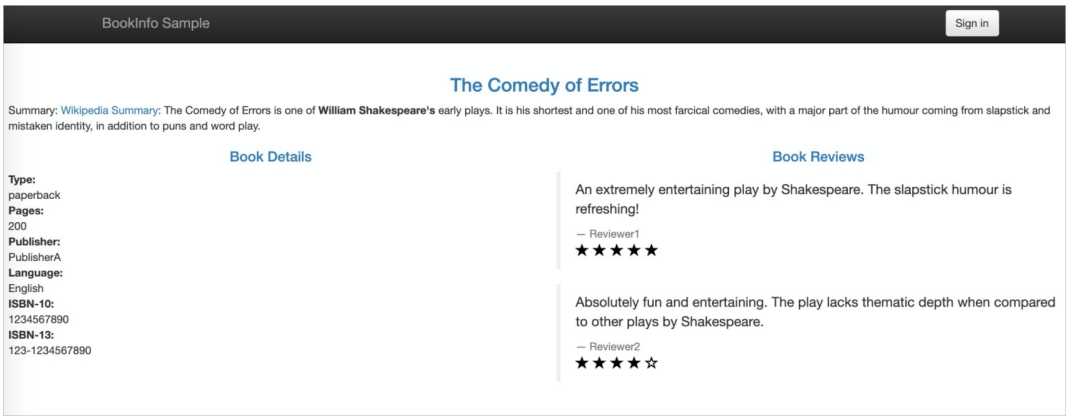
a. 登录容器服务管理控制台。

b. 在控制台左侧导航栏中，单击**集群**。

c. 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**。

d. 在集群管理页面选择**网络 > 服务**。

e. 在服务页面顶部设置命名空间为istio-system，查看istio-ingressgateway右侧80端口的外部端点，然后在浏览器地址栏中输入*80端口的入口网关地址/productpage*，访问Bookinfo应用。多次刷新页面，可以看到页面上显示黑色星星。



步骤五：渐进式发布应用

1. 进入asm_kubevela文件夹中，执行以下命令，升级reviews应用，并调整流量。

```
kubectl apply -f application_rollout-v2.yaml
```

- 在application_rollout-v2.yaml文件中更新了reviews镜像，从review v2升级到了review v3版本。并且设置升级的目标实例个数为2个，分两个批次升级，每批升级1个实例。

```
...
    - name: reviews
      type: webservice
      properties:
        image: docker.io/istio/examples-bookinfo-reviews-v3:1.16.2
        port: 9080
        volumes:
          - name: wlp-output
            type: emptyDir
            mountPath: /opt/ibm/wlp/output
          - name: tmp
            type: emptyDir
            mountPath: /tmp
      traits:
        - type: expose
          properties:
            port:
              - 9080
        - type: rollout
          properties:
            targetSize: 2
            rolloutBatches:
              - replicas: 1
              - replicas: 1
        - type: canary-traffic
          properties:
            port: 9080
...
```

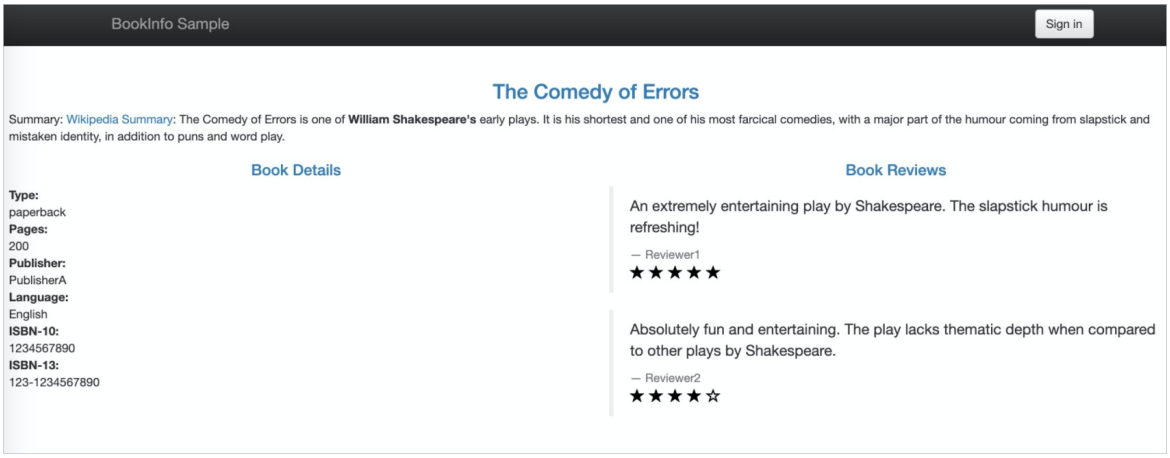
- targetSize：升级实例的批次。
- rolloutBatches：每批实例升级的个数。

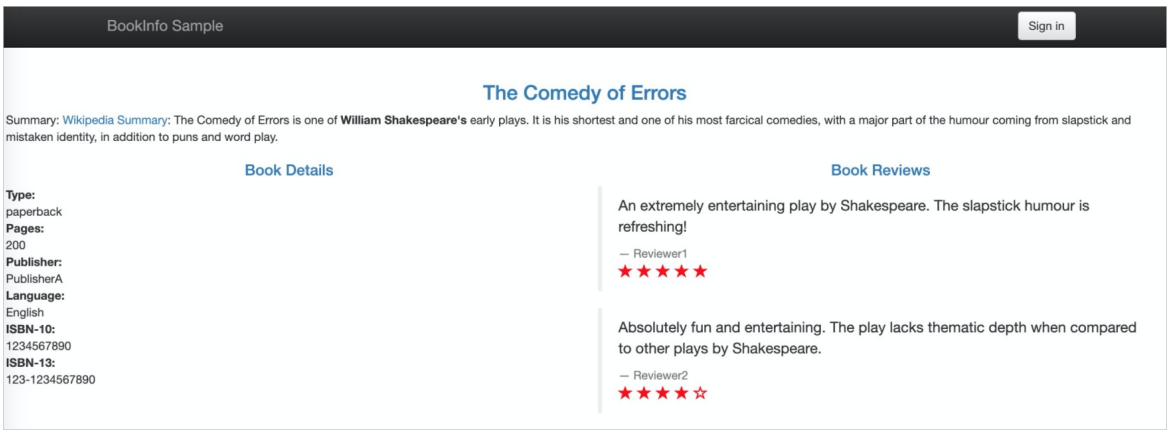
- 在application_rollout-v2.yaml文件中设置了以下三个执行工作流：
  a. 设置batchPartition等于0，表示只升级第一批次实例，即将reviews服务的2个Pod中，只升级其中的1个Pod。然后设置traffic.weightedTargets参数，将10%流量导向新升级的reviews服务，90%的流量仍然流向旧版本的服务。
  b. 设置type为*suspend*，完成第一步工作流后，将暂停工作流。
  c. 设置batchPartition等于1，表示升级第二批次实例，即将reviews服务的2个Pod都升级到v3版本镜像。然后设置traffic.weightedTargets参数，将100%流量导向新升级的reviews服务。

```
...
  workflow:
    steps:
      - name: rollout-1st-batch
        type: canary-rollout
        properties:
          # just upgrade first batch of component
          batchPartition: 0
          traffic:
            weightedTargets:
              - revision: reviews-v1
                weight: 90 # 90% shift to new version
              - revision: reviews-v2
                weight: 10 # 10% shift to new version
      # give user time to verify part of traffic shifting to newRevision
      - name: manual-approval
        type: suspend
      - name: rollout-rest
        type: canary-rollout
        properties:
          # upgrade all batches of component
          batchPartition: 1
          traffic:
            weightedTargets:
              - revision: reviews-v2
                weight: 100 # 100% shift to new version
...
```

2. 在浏览器地址栏中输入*80端口的入口网关地址/productpage*，访问Bookinfo应用。

多次刷新页面，10%概率可以看到红色星星，90%概率看到黑色星星。



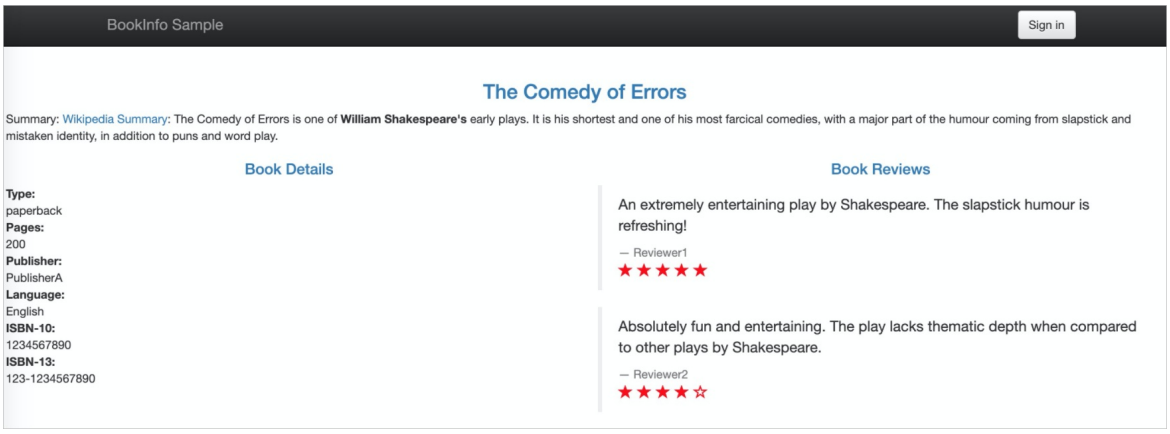


3. 执行以下命令，继续执行工作流，使reviews服务全部升级到v3版本。

```
vela workflow resume book-info
```

4. 在浏览器地址栏中输入*80端口的入口网关地址/productpage*，访问Bookinfo应用。

   多次刷新页面，页面上只显示红色星星。说明reviews服务全部升级到v3版本。



（可选）步骤六：回滚应用

如果发现发布的新版本应用不符合预期，您可以终止发布工作流量，并将应用回滚到之前的版本。

1. 执行以下命令，回滚应用。

```
kubectl apply -f  rollback.yaml
```

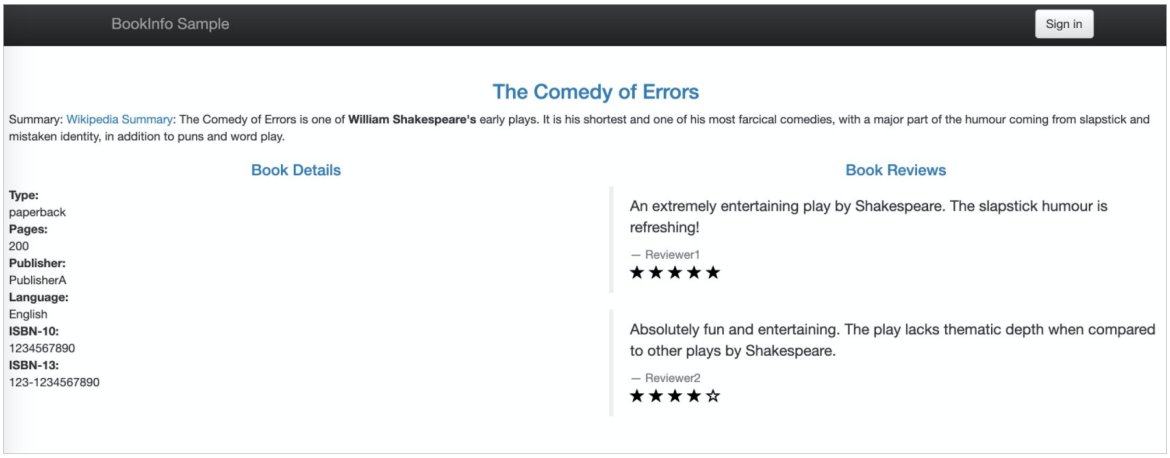
在rollback.yaml文件中设置type为*canary-rollback*，会自动进行以下操作：

- 更新Rollout对象的 `targetRevisionName` 成旧的版本，自动回滚所有已发布的新版本的实例回到旧版本，并且保持还没升级的旧版本实例。
- 更新VirtualService对象的 `route` 字段，将所有流量导向旧的版本。
- 更新DestinationRule对象的 `subset` 字段，只容纳旧的版本。

```
...
workflow:
  steps:
    - name: rollback
      type: canary-rollback
```

2. 在浏览器地址栏中输入*80端口的入口网关地址/productpage*，访问Bookinfo应用。

   多次刷新页面，页面上只显示黑色星星。说明reviews服务回滚到v2版本。



3.4. 基于ASM商业版实现全链路灰度发布

当您需要在多个服务间实现全链路的灰度发布时，您可以通过配置TrafficLabel来识别流量特征，将网关入口流量分为正常流量和灰度流量，灰度流量特征会在请求调用链经过的各个服务间进行传递，从而实现全链路灰度发布。本文通过一个Demo示例来介绍如何通过TrafficLabel能力来实现微服务的全链路灰度发布。

前提条件

- 已创建ASM实例，选择商业版（专业版），版本v1.10.5.40+。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。
- ASM实例启用链路追踪，在创建ASM实例时，选中**启用链路追踪**。具体操作，请参见创建ASM实例。
- 创建ASM网关。具体操作，请参见添加入口网关服务。
- 在ACK集群下开启应用监控。具体操作，请参见应用性能监控。

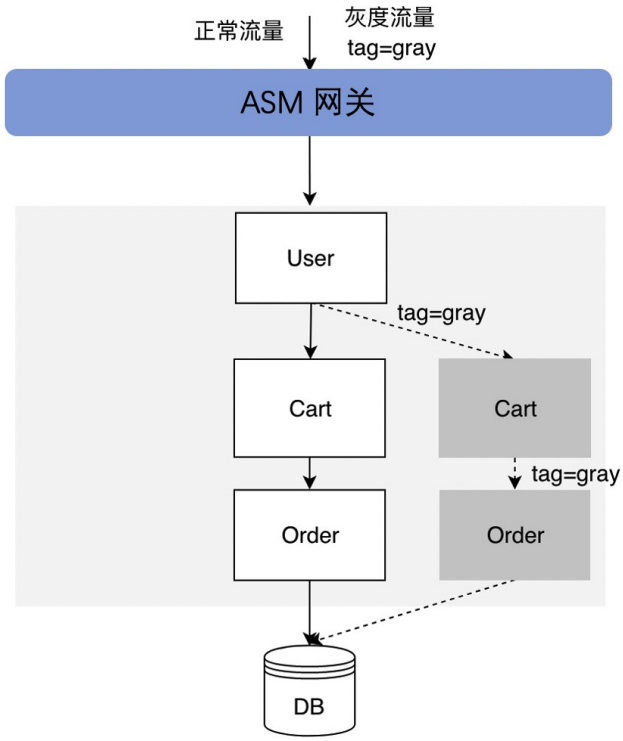
> **说明** 本示例Demo应用通过Java Agent方式对接ARMS，若非Java的其他多语言应用需要对接ARMS，请参见应用监控接入概述。

- 通过kubectl连接ACK集群。具体操作，请参见通过kubectl工具连接集群。
- 通过kubectl连接ASM集群。具体操作，请参见通过kubectl连接ASM实例。

背景信息

灰度发布有多种实现方式，例如使用服务网格ASM结合KubeVela实现渐进式灰度发布和基于ASM完成蓝绿和灰度发布。以上两种灰度，偏重于单个服务的发布；背后的技术都是利用Istio原生提供的VirtualService标签路由和权重分流来实现的。

某些场景下，仅限于两个服务间的灰度不能满足需求，例如以下场景：Cart和Order同时发布了灰度版本。



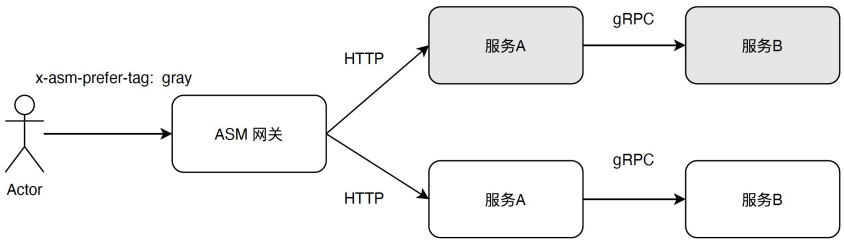
这种情况下对业务进行功能灰度验证：因为入口流量分为正常流量和灰度流量，其中User服务需要对请求流量做流量特征识别，若是灰度流量则需要请求灰度的Cart。不再是简单的按流量比例灰度分发到后端不同的版本，而且灰度流量特征会在请求调用链经过的各个服务间进行传递。

ASM全链路灰度功能基于流量打标和标签路由功能，更多信息，请参见流量打标和标签路由。

Demo介绍

Demo示例编排及相关配置，您可以通过此文件下载。

Demo的架构图如下：



部署编排文件*demo.yaml*如下：

```
apiVersion: v1
kind: Service
metadata:
  name: spring-boot-istio-client
```

```
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 19090
  selector:
    app: spring-boot-istio-client
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-client
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-client
      version: base
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-client
      labels:
        app: spring-boot-istio-client
        version: base
    spec:
      containers:
        - name: spring-boot-istio-client
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-clien
t:Abase
          imagePullPolicy: Always
          tty: true
          ports:
            - name: http
              protocol: TCP
              containerPort: 19090
---
apiVersion: v1
kind: Service
metadata:
  name: spring-boot-istio-server
spec:
  type: ClusterIP
  ports:
    - name: http
      port: 18080
      targetPort: 18080
    - name: grpc
      port: 18888
      targetPort: 18888
  selector:
    app: spring-boot-istio-server
---
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-server
      version: base
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-server
      labels:
        app: spring-boot-istio-server
        version: base
    spec:
      containers:
        - name: spring-boot-istio-server
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-serve
r:Bbase
          imagePullPolicy: Always
          tty: true
          ports:
            - name: http
              protocol: TCP
              containerPort: 18080
            - name: grpc
              protocol: TCP
              containerPort: 18888
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-client-gray
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-client
      version: gray
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-client
      labels:
        app: spring-boot-istio-client
        version: gray
    spec:
      containers:
        - name: spring-boot-istio-client
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-clien
```

```
t:Agray
          imagePullPolicy: Always
          tty: true
          ports:
            - name: http
              protocol: TCP
              containerPort: 19090
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-istio-server-gray
spec:
  replicas: 2
  selector:
    matchLabels:
      app: spring-boot-istio-server
      version: gray
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-server
      labels:
        app: spring-boot-istio-server
        version: gray
    spec:
      containers:
      - name: spring-boot-istio-server
        image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/spring-boot-istio-server:
Bgray
        imagePullPolicy: Always
        tty: true
        ports:
         - name: http
           protocol: TCP
           containerPort: 18080
         - name: grpc
           protocol: TCP
           containerPort: 18888
```

本示例服务都为SpringBoot的Java应用，且开启了ARMS应用监控。具体操作，请参见应用性能监控 。
对应的Deployment的 `template.metadata` 下类似如下配置：

```
  template:
    metadata:
      annotations:
        armsPilotAutoEnable: 'on'
        armsPilotCreateAppName: spring-boot-istio-server
```

步骤一：在ACK集群下部署Demo微服务

执行以下命令，部署Demo。

```
kubectl apply -f demo.yaml
```

步骤二：配置简单路由

1. 使用以下内容，创建名为*istio-config.yaml*的文件。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: simple-springboot-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - "*"
      port:
        name: http
        number: 80
        protocol: HTTP
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-client-vs
spec:
  gateways:
  - simple-springboot-gateway
  hosts:
    - "*"
  http:
  - match:
    - uri:
       prefix: "/hello"
    route:
      - destination:
          host: spring-boot-istio-client
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-server-vs
spec:
  hosts:
    - spring-boot-istio-server
  http:
    - route:
        - destination:
            host: spring-boot-istio-server
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: springboot-istio-client-dr
spec:
```

```
  host: spring-boot-istio-client
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - labels:
        version: base
      name: version-base
    - labels:
        version: gray
      name: version-gray
---
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: springboot-istio-server-dr
spec:
  host: spring-boot-istio-server
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - labels:
        version: base
      name: version-base
    - labels:
        version: gray
      name: version-gray
```

2. 执行以下命令，配置路由。

```
kubectl --kubeconfig <ASM实例的kubeconfig文件> apply -f istio-config.yaml
```

3. 验证服务访问是否可以连通。

   i. 通过ASM控制台获取网关的公网IP，执行以下命令。

   ```
   export ASM_GATEWAY_IP=xxx
   ```

   ii. 执行以下命令，验证服务访问是否可以连通。

   ```
   while true; do curl -H'x-asm-prefer-tag: gray'  http://${ASM_GATEWAY_IP}/hello ; echo
   ;sleep 1;done
   ```

   预期输出：

   ```
   --> HTTP A-Base --> gRPC B-Gray.
   --> HTTP A-Gray --> gRPC B-Base.
   --> HTTP A-Base --> gRPC B-Gray.
   --> HTTP A-Gray --> gRPC B-Base.
   --> HTTP A-Base --> gRPC B-Base.
   ```

   gateway->A->B都是负载均衡到Base和灰度版本实例，此时通过curl命令指定的 `x-asm-prefer-tag` 没有效果，需要配置TrafficLabel以及对应的标签路由后才能生效，*istio-config.yaml*下默认配置是简单的路由，VirtualService下路由配置未指定subset分组。

步骤三：配置TrafficLabel

1. 使用以下内容，创建文件*traffic_label_default.yaml*。

```
---
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example1
  namespace: default
spec:
  rules:
  - labels:
      - name: userdefinelabel1
        valueFrom:
        - $getContext(x-b3-traceid)
        - $localLabel
    attachTo:
    - opentracing
    # 表示生效的协议，空为都不生效，*为都生效
    protocols: "*"
  hosts: # 表示生效的服务
  - "*"
---
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: ingressgateway
  namespace: istio-system
spec:
  hosts:
  - '*'
  rules:
  - attachTo:
    - opentracing
    labels:
    - name: userdefinelabel1
      valueFrom:
      - $getContext(x-b3-traceid)
      - $localLabel
    protocols: '*'
  workloadSelector:
    labels:
      app: istio-ingressgateway
```

2. 执行以下命令，使用ASM实例的kubeconfig进行部署。

```
kubectl --kubeconfig <ASM实例的kubeconfig文件> apply -f traffic_label_default.yaml
```

该TrafficLabel定义针对default下所有服务生效，也就是*demo.yaml*下部署的A和B服务。

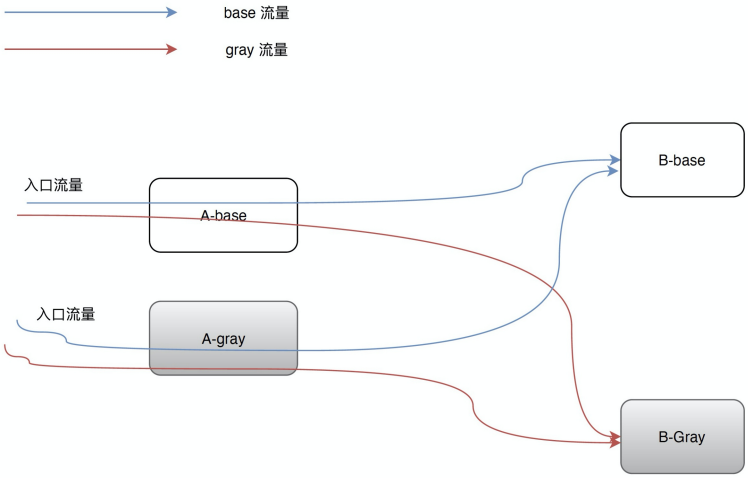
> **说明** 因为本示例Demo对接了ARMS采用zipkin trace类型的 `traceId` ，因此 `getContext` 参数为 `x-b3-traceid` 。

步骤四：验证TrafficLabel路由

1. 验证A->B是否符合预期，其中包括流到A的灰度流量打到灰度版本，以及Base流量打到Base版本

1. 验证A->B是否符合预期，其中包括流到A的灰度流量打到灰度版本，以及base流量打到base版本。

   配置B服务的TrafficLabel路由*b-vs-tf.yaml*，在A服务侧生效，对应如下模型：



   i. 使用以下内容，创建文件*b-vs-tf.yaml*。

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-server-vs
spec:
  hosts:
    - spring-boot-istio-server
  http:
    - route:
        - destination:
            host: spring-boot-istio-server
            subset: $userdefinelabel1
```

   ii. 执行以下命令，在A服务侧生效。

```
kubectl -f <ASM实例的kubeconfig文件> apply -f b-vs-tf.yaml
```

iii. 执行以下命令，验证流到A的灰度流量打到灰度版本。

```
while true; do curl -H'x-asm-prefer-tag: version-gray'  http://${ASM_GATEWAY_IP}/hell
o ; echo;sleep 1;done
```

预期输出：

```
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
```

iv. 执行以下命令，验证流到A的Base流量打到Base版本。

```
while true; do curl -H'x-asm-prefer-tag: version-base'  http://${ASM_GATEWAY_IP}/hell
o ; echo;sleep 1;done
```

预期输出：

```
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Gray --> gRPC B-Base.
```

> **说明** 入口流量访问A服务并未流转到指定版本，需要再配置A服务的TrafficLabel路由。

2. 验证ASM网关->A是否符合预期，其中包括入口请求灰度流量打到A的灰度版本，Base流量打到A的Base版本，并传递到B服务。

   配置A服务的TrafficLabel路由*a-vs-tf.yaml*，在ASM网关侧生效。

> **说明** ASM网关也支持TrafficLabel路由。

i. 使用以下示例，创建文件*a-vs-tf.yaml*。

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-client-vs
spec:
  gateways:
  - simple-springboot-gateway
  hosts:
    - "*"
  http:
  - match:
    - uri:
       prefix: "/hello"
    route:
      - destination:
          host: spring-boot-istio-client
          subset: $userdefinelabel1
```

ii. 执行以下命令，在ASM网关侧生效。

```
kubectl -f <ASM实例的kubeconfig文件> apply -f a-vs-tf.yaml
```

iii. 执行以下命令，验证入口请求灰度流量打到A的灰度版本。

```
 while true; do curl -H'x-asm-prefer-tag: version-gray'  http://${ASM_GATEWAY_IP}/hel
lo ; echo;sleep 1;done
```

预期输出：

```
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
```

iv. 执行以下命令，验证Base流量打到A的Base版本，并传递到B服务。

```
while true; do curl -H'x-asm-prefer-tag: version-base'  http://${ASM_GATEWAY_IP}/hell
o ; echo;sleep 1;done
```

预期输出：

```
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
--> HTTP A-Base --> gRPC B-Base.
```

3. 验证TrafficLabel路由对应的权重分流是否符合预期。

i. 使用以下示例，创建名为*a-vs-tf-10-90.yam*l文件。

```
---
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: springboot-istio-client-vs
spec:
  gateways:
  - simple-springboot-gateway
  hosts:
    - "*"
  http:
  - match:
    - uri:
       prefix: "/hello"
    route:
      - destination:
          host: spring-boot-istio-client
          subset: $userdefinelabel1
        weight: 10
      - destination:
          host: spring-boot-istio-client
          subset: version-base
        weight: 90
```

> **说明** 无论Gray或Base什么流量，只有10%打到对应的分组，其余打到version-base。

ii. 执行以下命令，在网关侧生效。

```
kubectl --kubeconfig <ASM实例的kubeconfig文件> apply -f a-vs-tf-10-90.yaml
```

iii. 执行以下命令，验证灰度流量。

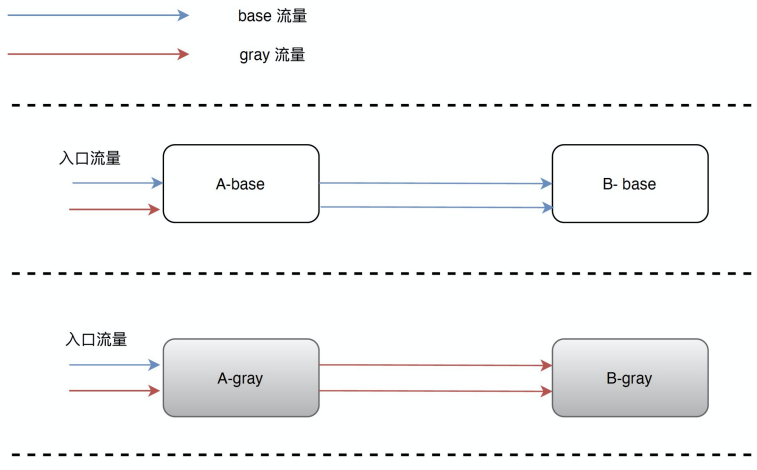
```
 while true; do curl -H'x-asm-prefer-tag: version-gray'  http://${ASM_GATEWAY_IP}/hel
lo ; echo;sleep 1;done
```

预期输出：

```
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
--> HTTP A-Gray --> gRPC B-Gray.
--> HTTP A-Base --> gRPC B-Gray.
```

泳道模式，流量特征不传递

某些场景下，您可能希望开启类似泳道模式，也就是不传递上下文流量特征（即流量颜色标记，红色表示Gray流量，蓝色表示Base流量），出口流量标记采用本地Label，对应以下模型：



此场景中，您只需修改TrafficLabel定义，去除 `$getContext(x-b3-traceid)` 关闭流量标签的传递，将颜色标记从 `$localLabel` 获取即可。

*cat traffic_label_default_swimlane.yaml*示例如下：

```
apiVersion: istio.alibabacloud.com/v1beta1
kind: TrafficLabel
metadata:
  name: example1
  namespace: default
spec:
  rules:
  - labels:
      - name: userdefinelabel1
        valueFrom:
        - $localLabel
    attachTo:
    - opentracing
    # 表示生效的协议，空为都不生效，*为都生效。
    protocols: "*"
  hosts: # 表示生效的服务。
  - "*"
```

localLabel是从服务Deploymen下Pod的label中获取流量标识，因此demo例子下的Deployment需要添加Pod标签 `ASM_TRAFFIC_TAG: version-base` ，例如：

```
 template:
    metadata:
      annotations:
        armsPilotAutoEnable: "on"
        armsPilotCreateAppName: spring-boot-istio-client
      creationTimestamp: null
      labels:
        ASM_TRAFFIC_TAG: version-base
        app: spring-boot-istio-client
        version: base
```

如果是gray版本对应的Deployment ，则需要添加 `ASM_TRAFFIC_TAG: version-gray` 标签。

ASM网关侧的流量灰度

如上文中的*a-vs-tf.yaml*，配置入口流量 `<gatewayIP>/hello` 接口的灰度，对入口请求有一定的要求：要求流量标识通过 `x-asm-prefer-tag` 指定流量Tag，如上测试是 `curl -H 'x-asm-prefer-tag: xxx'` 手动指定的。

实际业务场景下，客户端App或者真实用户通过浏览器访问未必会设置该Header，这种情形下，您可以通过ASM网关的自定义Header功能以及Lua插件的能力，实现将业务场景的灰度映射到 `x-asm-prefer-tag` Header, 进行标准化处理。

例如您可以通过 `EnvoyFilter` 设置“使用iPhone13的用户”为灰度流量，示例如下。

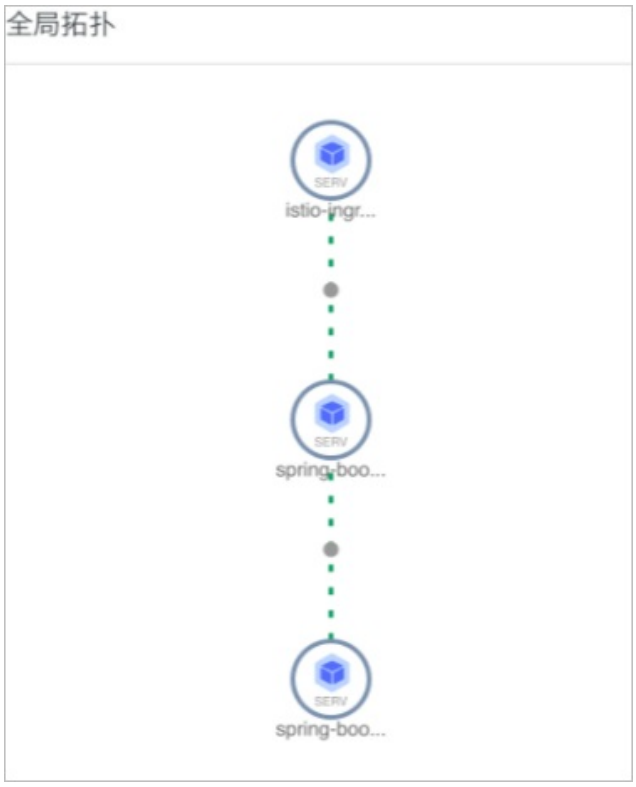
```
apiVersion: networking.istio.io/v1alpha3
kind: EnvoyFilter
metadata:
  labels:
    provider: "asm"
    asm-system: "true"
  name: gateway-lua-filter-add-x-asm-prefer-tag-header
  namespace: istio-system
spec:
  workloadSelector:
    labels:
      istio: ingressgateway
  configPatches:
  - applyTo: HTTP_FILTER
    match:
      proxy:
        proxyVersion: "^1.*"
      context: GATEWAY
      listener:
        filterChain:
          filter:
            name: "envoy.filters.network.http_connection_manager"
            subFilter:
              name: "envoy.filters.http.router"
    patch:
      operation: INSERT_BEFORE
      value:
       name: envoy.lua
       typed_config:
         "@type": "type.googleapis.com/envoy.extensions.filters.http.lua.v3.Lua"
         inlineCode: |
              function envoy_on_request(request_handle)
                local user_agent = request_handle:headers():get("user-agent")
                request_handle:logInfo("user_agent:"..user_agent)
                if string.match(user_agent,"^.*iPhone13.*") then
                    request_handle:headers():add("x-asm-prefer-tag","version-gray")
                else
                   request_handle:headers():add("x-asm-prefer-tag","version-base")
                end
              end
              function envoy_on_response(response_handle)
              end
```

FAQ

为什么全链路灰度功能没有生效？

全链路灰度功能生效的前提是应用的Trace能力生效，本文的SpringCloud服务采用ARMS无侵入方式接入Trace，若测试结果不符合预期，请确认是否正确开启了应用性能监控，您可以通过以下方式检查是否开启应用性能监控：

登录链路追踪Tracing Analysis控制台，在控制台左侧导航栏单击**全局拓扑**。在**全局拓扑**页面可以看到调用链ingressgateway -> springcloud-istio-client -> springcloud-istio-server，说明开启应用性能监控成功。



若您是在部署demo服务后开启的应用性能监控，您需要在开启应用性能监控后，重新部署demo服务。关于开启应用监控的具体操作，请参见应用性能监控。

相关文档

- 应用性能监控
- 流量打标和标签路由

4.安全

4.1. 在ASM中实现跨域访问

当一个客户端去访问另一个不同域名或者同域名不同端口的服务时，就会发出跨域请求。如果此时该服务不允许其进行跨域资源访问，那么就会因为跨域问题而导致访问失败。跨源域资源共享CORS（Cross-Origin Resource Sharing）允许Web应用服务器进行跨域访问控制。本文介绍如何在ASM的Virtualservice中配置corsPolicy，实现跨域访问。

跨源域资源共享CORS介绍

出于安全性考虑，浏览器会限制页面脚本内发起的跨源HTTP请求，例如在使用XMLHttpRequest和Fetch API遵循同源策略，使用这些API的Web应用程序只能从加载应用程序的同一个域请求HTTP资源，除非响应报文包含了正确CORS响应头。

跨域资源共享CORS是一种基于HTTP头的机制，该机制允许服务器标示除了它自己以外的其它域，协议和端口，使得浏览器可以访问加载这些资源。

跨域资源共享CORS的验证机制分两种模式：简单请求和预先请求。

- 简单请求模式：
  浏览器直接发送跨域请求，并在请求头中携带Origin的头，表明这是一个跨域的请求。服务器端接到请求后，会根据自己的跨域规则，通过Access-Control-Allow-Origin和Access-Control-Allow-Methods响应头来返回验证结果。
- 预先请求模式：
  浏览器会先发送Preflighted requests（预先验证请求），Preflighted requests是一个OPTION请求，用于询问要被跨域访问的服务器，是否允许当前域名下的页面发送跨域的请求。在得到服务器的跨域授权后才能发送真正的HTTP请求。
  OPTIONS请求头部中会包含以下头部：Origin、Access-Control-Request-Method、Access-Control-Request-Headers。服务器收到OPTIONS请求后，设置Access-Control-Allow-Origin、Access-Control-Allow-Method、Access-Control-Allow-Headers、Access-Control-Max-Age头部与浏览器沟通来判断是否允许这个请求。如果Preflighted requests验证通过，浏览器才会发送真正的跨域请求。

当请求同时满足下面三个条件时，CORS验证机制会使用简单请求模式进行处理，否则CORS验证机制会使用预先请求模式进行处理。

- 请求方法是下列之一：
  GET、HEAD、POST
- 请求头中的Content-Type请求头的值是下列之一：
  ext/plain、application/x-www-form-urlencoded、multipart/form-data
- Fetch规范定义了CORS安全头的集合，安全头的集合是下列之一：
  Accept、Accept-Language、Content-Language、Content-Type（需要注意额外的限制）

在VirtualService中配置corsPolicy

整个CORS通信过程都是浏览器自动完成，您需要在对应服务的VirtualService中添加 `corsPolicy` 字段，使服务允许跨域请求，从而实现跨域通信。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings-route
spec:
  hosts:
  - ratings.prod.svc.cluster.local
  http:
  - route:
    - destination:
        host: ratings.prod.svc.cluster.local
        subset: v1
    corsPolicy:
      allowOrigins:
      - exact: https://example.com
      - regex: *     #支持regex
      allowMethods:
      - POST
      - GET
      allowCredentials: false
      allowHeaders:
      - X-Foo-Bar
      maxAge: "24h"
```

参数	说明
allowOrigins	允许请求服务的来源，允许带哪些Origin地址的请求，支持regex匹配。对于不需要携带身份凭证的请求，服务器可以指定该字段的值为通配符，表示允许来自所有域的请求。
allowMethods	允许请求服务的方法，实际请求所允许使用的HTTP方法。
allowHeaders	允许请求服务的标头，用于预检请求的响应。其指明了实际请求中允许携带的首部字段。
exposeHeaders	公开请求服务的标头，让服务器把允许浏览器访问的头放入白名单。
maxAge	最大浏览器缓存时间，指定了浏览器能够缓存preflight请求结果的时间。
allowCredentials	允许请求服务的凭证，符合要求的凭证才能请求服务。

跨域访问最佳实践

操作前准备：

- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。
- 已创建default和foo命名空间，并为命名空间开启自动注入。具体操作，请参见多种方式灵活开启自动注入。

步骤一：部署应用

1. 部署后端应用。
    i. 通过kubectl工具连接集群。

ii. 使用以下内容，创建*details.yaml*文件。

```
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-v1:1.16.4
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 9080
        securityContext:
          runAsUser: 1000
```

iii. 执行以下命令，在default命名空间部署details应用。

```
kubectl apply -f details.yaml -n default
```

2. 部署前端应用。

   i. 使用以下内容，创建*istio-cors-demo.yaml*文件。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: istio-cors-demo
---
apiVersion: v1
kind: Service
metadata:
  name: istio-cors-demo
  labels:
    app: istio-cors-demo
    service: istio-cors-demo
spec:
  ports:
    - name: http
      port: 8000
      targetPort: 80
  selector:
    app: istio-cors-demo
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: istio-cors-demo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: istio-cors-demo
      version: v1
  template:
    metadata:
      labels:
        app: istio-cors-demo
        version: v1
    spec:
      serviceAccountName: istio-cors-demo
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/build-test/istio-cors-demo:v1.0-g8
e215f6-aliyun
          imagePullPolicy: IfNotPresent
          name: istio-cors-demo
          ports:
            - containerPort: 80
```

   ii. 执行以下命令，在foo命名空间部署istio-cors-demo应用。

```
kubectl apply -f istio-cors-demo.yaml -n foo
```

步骤二：创建ASM网关

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单击**ASM网关**，然后在右侧页面单击**创建**。
5. 在**创建**页面设置**名称**为*ingressgateway*，选择**部署集群**，设置**负载均衡类型**为**公网访问**，在**新建负载均衡**下选择负载均衡规格，其他参数采用默认设置，单击**创建**。
6. 重复步骤4和步骤5，创建名为ingressgateway2的网关。

步骤三：创建路由规则

1. 创建后端应用的路由规则。
   i. 创建网关规则。

      创建网关规则，绑定details应用和ingressgateway网关。

      a. 登录ASM控制台。
      b. 在左侧导航栏，选择**服务网格 > 网格管理**。
      c. 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**。
      d. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。
      e. 在**创建**页面设置**命名空间**为default，选择任意模板，将**YAML**文本框中的内容替换为以下内容，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: bookinfo-gateway
  namespace: default
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: http
        number: 80
        protocol: HTTP
```

ii. 创建虚拟服务。

a. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

b. 在**创建**页面设置**命名空间**为default，选择任意模板，将**YAML**文本框中的内容替换为以下内容，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: bookinfo
  namespace: default
spec:
  gateways:
    - bookinfo-gateway
  hosts:
    - '*'
  http:
    - match:
        - uri:
            prefix: /details
      route:
        - destination:
            host: details
            port:
              number: 9080
```

iii. 访问后端应用。

a. 获取ingressgateway网关的IP地址，具体操作，请参见添加入口网关服务。

b. 在浏览器地址栏中输入*http://<ingressgateway网关的IP>/details/2* 。

返回以上页面，说明访问后端应用details成功。

2. 创建前端应用的路由规则。

i. 创建网关规则。

创建网关规则，绑定istio-cors-demo应用和ingressgateway2网关。

a. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。

b. 在**创建**页面设置**命名空间**为foo，选择任意模板，将**YAML**文本框中的内容替换为以下内容，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: istio-cors-demo-gateway
  namespace: foo
spec:
  selector:
    istio: ingressgateway2
  servers:
    - hosts:
        - '*'
      port:
        name: http
        number: 80
        protocol: HTTP
```

ii. 创建虚拟服务。

a. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

b. 在**创建**页面设置**命名空间**为foo，选择任意模板，将**YAML**文本框中的内容替换为以下内容，单击**创建**。

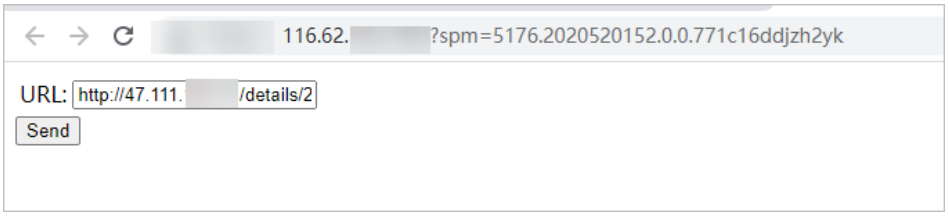
```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: istio-cors-demo
  namespace: foo
spec:
  gateways:
    - istio-cors-demo-gateway
  hosts:
    - '*'
  http:
    - route:
        - destination:
            host: istio-cors-demo
            port:
              number: 8000
```

3. 使用前端应用访问后端应用。

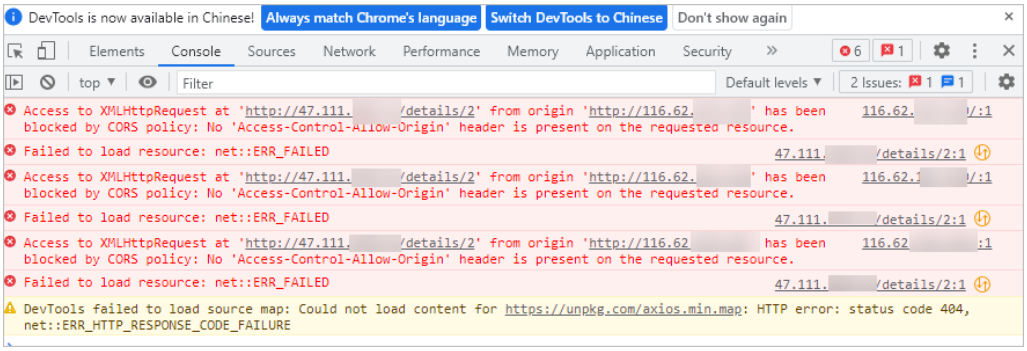
i. 获取ingressgateway2网关的IP地址，具体操作，请参见添加入口网关服务。

ii. 在谷歌浏览器地址栏中输入*http://<ingressgateway2网关的IP地址>*。

iii. 在URL文本框中输入*http://<ingressgateway网关的IP地址>/details/2*，单击Send。



iv. 在谷歌浏览器右上角单击图标，选择**更多工具 > 开发者工具**。



由于前端istio-cors-demo应用访问后端details应用需要跨域，访问会失败，因此出现以上报错。

步骤四：配置跨域访问

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**流量管理 > 虚拟服务**。
5. 在**虚拟服务**页面单击bookinfo右侧**操作**列下的**查看YAML**。
6. 在**编辑**对话框http参数下添加以下跨域配置，然后单击**确定**。

```
- corsPolicy:
    allowCredentials: false
    allowMethods:
      - POST
      - GET
    allowOrigins:
      - prefix: 'http://<ingressgateway2网关的IP>'
    maxAge: 24h
```

```
15    http:
16      - match:
17          - uri:
18              prefix: /details
19        route:
20          - destination:
21              host: details
22              port:
23                number: 9080
24        corsPolicy:
25          allowCredentials: false
26          allowMethods:
27            - POST
28            - GET
29          allowOrigins:
30            - prefix: 'http://116.62.
31          maxAge: 24h
```

步骤五：验证跨域访问是否成功

1. 在谷歌浏览器地址栏中输入*http://<ingressgateway2网关的IP地址>*。
2. 在URL文本框中输入*http://<ingressgateway网关的IP地址>/details/2*，单击**send**。

返回以上页面，说明前端istio-cors-demo应用访问后端details应用成功，跨域访问成功。

4.2. 启用Multi-Buffer实现TLS加速

ASM商业版（专业版）结合Intel的Multi-Buffer加解密技术，可以加速Envoy中TLS的处理过程。本文介绍如何启用Multi-Buffer实现TLS加速。

前提条件

- 已创建ASM商业版（专业版）实例，且实例为1.10及以上版本。具体操作，请参见创建ASM实例。
- 已创建ACK，且集群节点的实例规格族需要支持Multi-Buffer CPU机型Intel Ice Lake。具体操作，请参见创建Kubernetes托管版集群。
  以下实例规格族支持Multi-Buffer CPU机型Intel Ice Lake：

> **说明** 关于实例规格的详细介绍，请参见实例规格族。

规格族系列	实例规格族
	存储增强通用型实例规格族g7se

规格族系列	实例规格族
g7系列	通用型实例规格族g7
	安全增强通用型实例规格族g7t
c7系列	计算型实例规格族c7
	RDMA增强型实例规格族c7re
	存储增强计算型实例规格族c7se
	安全增强计算型实例规格族c7t
r7系列	内存型实例规格族r7p
	存储增强内存型实例规格族r7se
	内存型实例规格族r7
	安全增强内存型实例规格族r7t
其他	内存增强型实例规格族re7p
	GPU虚拟化型实例规格族vgn7i-vws
	GPU计算型实例规格族gn7i
	GPU计算型弹性裸金属服务器实例规格族ebmgn7i
	计算型超级计算集群实例规格族sccc7
	通用型超级计算集群实例规格族sccg7

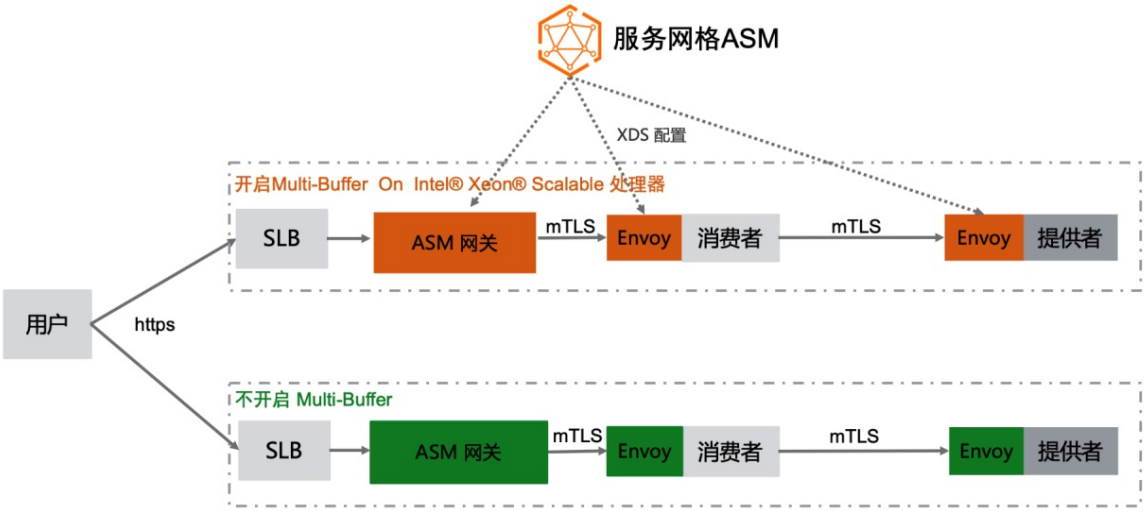
- 添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

背景信息

随着网络安全技术的发展，TLS已经成为网络通信的基石。一个TLS会话的处理过程总体上可分为握手阶段和数据传输阶段。握手阶段最重要的任务是使用非对称加密技术协商出一个会话密钥，然后在数据传输阶段，使用该会话密钥对数据执行对称加密操作，再进行数据传输。

在微服务场景下，Envoy无论是作为Ingress Gateway还是作为微服务的代理，都需要处理大量的TLS请求，尤其在握手阶段执行非对称加解密的操作时，需要消耗大量的CPU资源，在大规模微服务场景下这可能会成为一个瓶颈。ASM结合Intel的Multi-Buffer加解密技术，可以加速Envoy中TLS的处理过程。

Multi-Buffer加解密技术使用Intel CPU AVX-512指令同时处理多个独立的缓冲区，即可以在一个执行周期内同时执行多个加解密的操作，成倍的提升加解密的执行效率。Multi-Buffer技术不需要额外的硬件，只需要CPU包含特定的指令集。目前阿里云在Ice Lake处理器中已经包含了最新的AVX-512指令集。



操作步骤

您可以通过以下两种方式来启用Multi-Buffer功能：

- 如果您没有创建ASM实例，您可以在创建ASM实例时选中**启用基于MultiBuffer的TLS加解密性能优化**。具体操作，请参见创建ASM实例。
- 如果您已创建ASM实例，您可以在ASM实例的**网格信息**页面启用基于MultiBuffer的TLS加解密性能优化功能。本文以已创建ASM实例场景为例。

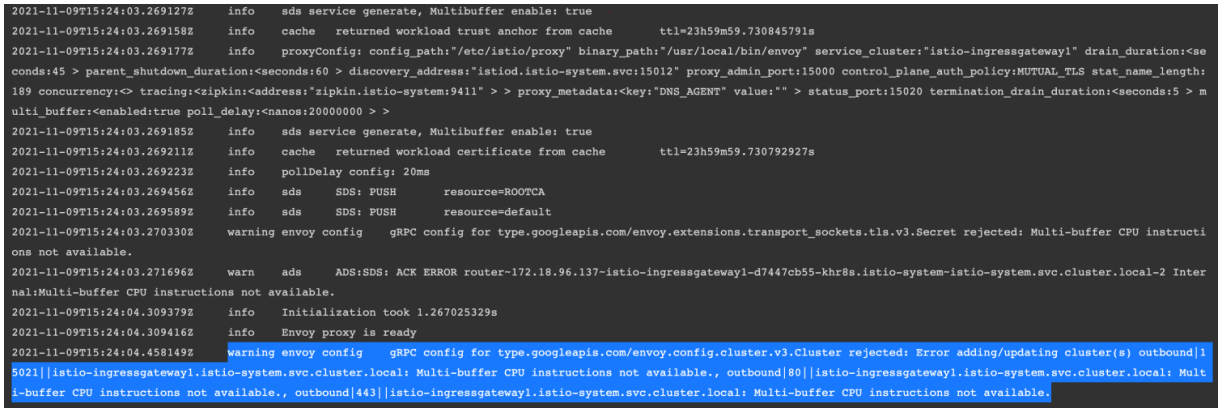
1. 登录ASM控制台。
2. 在左侧导航栏，选择**服务网格 > 网格管理**。
3. 在**网格管理**页面，找到待配置的实例，单击实例的名称或在**操作**列中单击**管理**。
4. 在网格详情页面左侧导航栏选择**网格实例 > 基本信息**，然后在右侧页面单击**功能设置**。
5. 在**功能设置更新**面板选中**启用基于MultiBuffer的TLS加解密性能优化**，然后单击**确定**。

   如果您使用通用型实例规格族g7作为Kubernertes节点，启用Multi-Buffer功能后，每秒查询率（QPS）将提升75%的性能。如果您使用的是弹性裸金属节点，提升的性能将更高。

FAQ

如果在控制面启用了MultiBuffer功能，但数据面Kubernetes集群下的节点不是Intel Ice Lake的机型会怎么样？

Envoy会输出告警日志，且MultiBuffer功能将不会生效。



```
2021-11-09T15:24:03.269127Z     info    sds service generate, Multibuffer enable: true
2021-11-09T15:24:03.269158Z     info    cache   returned workload trust anchor from cache       ttl=23h59m59.730845791s
2021-11-09T15:24:03.269177Z     info    proxyConfig: config_path:"/etc/istio/proxy" binary_path:"/usr/local/bin/envoy" service_cluster:"istio-ingressgateway1" drain_duration:<se
conds:45 > parent_shutdown_duration:<seconds:60 > discovery_address:"istiod.istio-system.svc:15012" proxy_admin_port:15000 control_plane_auth_policy:MUTUAL_TLS stat_name_length:
189 concurrency:<> tracing:<zipkin:<address:"zipkin.istio-system:9411" > > proxy_metadata:<key:"DNS_AGENT" value:"" > status_port:15020 termination_drain_duration:<seconds:5 > m
ulti_buffer:<enabled:true poll_delay:<nanos:20000000 > >
2021-11-09T15:24:03.269185Z     info    sds service generate, Multibuffer enable: true
2021-11-09T15:24:03.269211Z     info    cache   returned workload certificate from cache       ttl=23h59m59.730792927s
2021-11-09T15:24:03.269223Z     info    pollDelay config: 20ms
2021-11-09T15:24:03.269456Z     info    sds     SDS: PUSH       resource=ROOTCA
2021-11-09T15:24:03.269589Z     info    sds     SDS: PUSH       resource=default
2021-11-09T15:24:03.270330Z     warning envoy config    gRPC config for type.googleapis.com/envoy.extensions.transport_sockets.tls.v3.Secret rejected: Multi-buffer CPU instructi
ons not available.
2021-11-09T15:24:03.271696Z     warn    ads     ADS:SDS: ACK ERROR router~172.18.96.137~istio-ingressgateway1-d7447cb55-khr8s.istio-system~istio-system.svc.cluster.local~2 Inter
nal:Multi-buffer CPU instructions not available.
2021-11-09T15:24:04.309379Z     info    Initialization took 1.267025329s
2021-11-09T15:24:04.309416Z     info    Envoy proxy is ready
2021-11-09T15:24:04.458149Z     warning envoy config    gRPC config for type.googleapis.com/envoy.config.cluster.v3.Cluster rejected: Error adding/updating cluster(s) outbound|1
5021||istio-ingressgateway1.istio-system.svc.cluster.local: Multi-buffer CPU instructions not available., outbound|80||istio-ingressgateway1.istio-system.svc.cluster.local: Mult
i-buffer CPU instructions not available., outbound|443||istio-ingressgateway1.istio-system.svc.cluster.local: Multi-buffer CPU instructions not available.
```

ASM Pro 1.10及以上版本提供了开启TLS加速时的自适应判断能力，若业务或者网关Pod被调度到的Node节点为非Intel Ice Lake机型，则不会下发对应的加速配置，TLS加速不会生效。

如果Kubernetes集群没有支持Multi-Buffer功能类型的节点，那该集群如何才能使用MultiBuffer功能？

1. 在该Kubernetes集群添加新的节点，且节点的实例规格需要支持Multi-Buffer CPU机型Intel Ice Lake。具体操作，请参见添加已有节点。
2. 在新添加的节点上设置 `multibuffer-support: true` 标签。具体操作，请参见管理节点标签。
3. 在ASM网关的YAML配置中添加以下内容。具体操作，请参见修改入口网关服务。
   通过增加节点亲和性，使Gateway实例调度到新添加的支持Multi-Buffer功能的节点上。

   ```
   spec:
     affinity:
       nodeAffinity:
         requiredDuringSchedulingIgnoredDuringExecution:
           nodeSelectorTerms:
             - matchExpressions:
                 - key: multibuffer-support
                   operator: In
                   values:
                     - true
   ```

4. 在ASM商业版（专业版）启用MultiBuffer功能。具体操作，见上文。
   启用MultiBuffer功能后，该集群新添加的节点即可使用MultiBuffer功能，加速TLS处理过程。

5.服务网格gRPC协议

5.1. gRPC协议实践的设计原理

服务网格支持gRPC协议服务开发、容器化和网格化。本文介绍gRPC协议在ASM实践的设计原理。

通信模型

设计宗旨

- 覆盖gRPC的4种通信模型。
- 方法名和参数名不引入任何业务因素，避免额外思考，专注技术本身。

4种通信模型和实践方法的对应关系

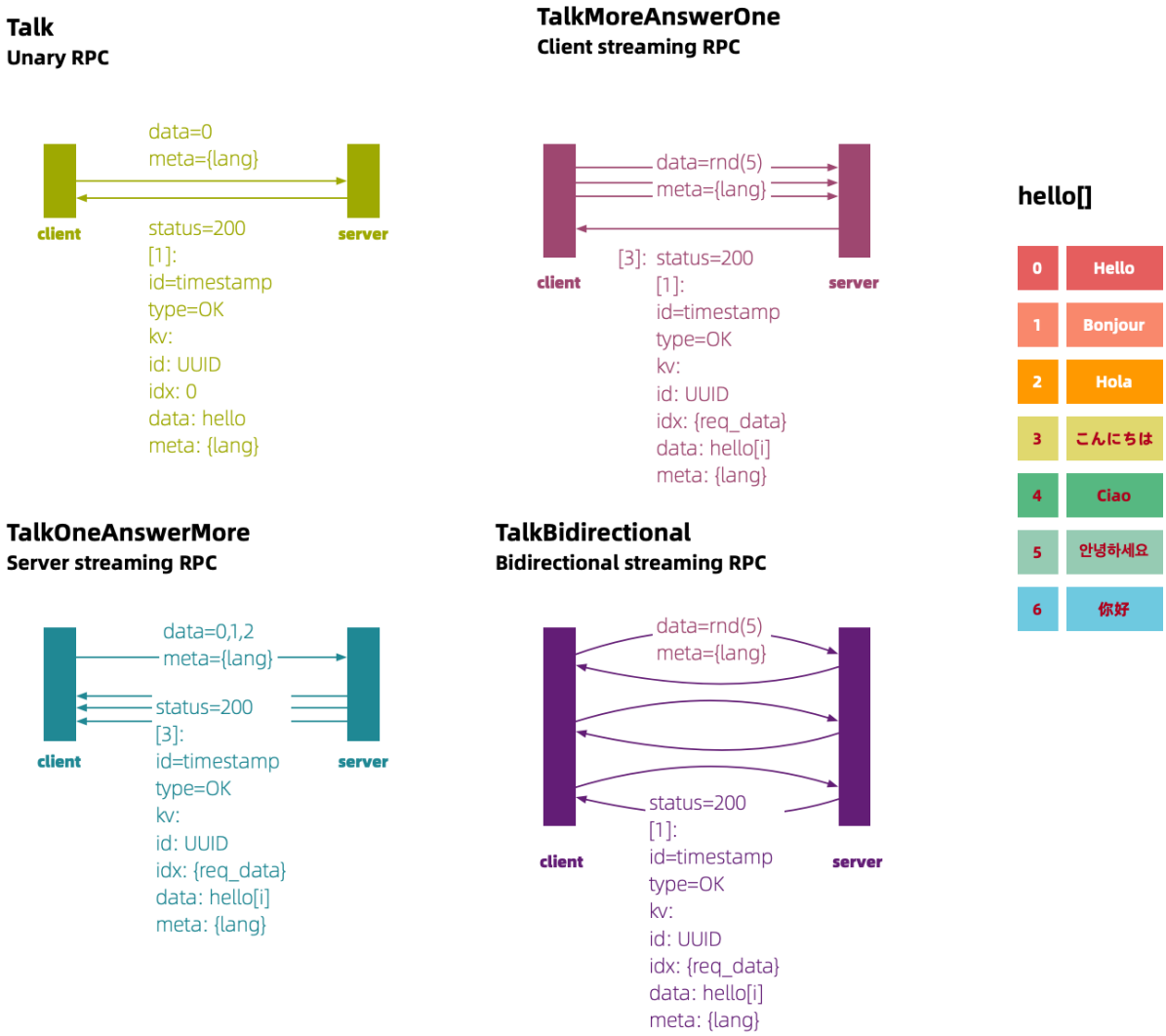
通信模型	实践方法
Unary RPC	talk
Server streaming RPC	talkOneAnswerMore
Client streaming RPC	talkMoreAnswerOne
Bidirectional streaming RPC	talkBidirectional

Protobuf定义

```
service LandingService {
  //Unary RPC
  rpc talk (TalkRequest) returns (TalkResponse) {
  }
  //Server streaming RPC
  rpc talkOneAnswerMore (TalkRequest) returns (stream TalkResponse) {
  }
  //Client streaming RPC with random & sleep
  rpc talkMoreAnswerOne (stream TalkRequest) returns (TalkResponse) {
  }
  //Bidirectional streaming RPC
  rpc talkBidirectional (stream TalkRequest) returns (stream TalkResponse) {
  }
```

方法设计

- 简单的主线逻辑：服务端将请求参数中的data字段的值作为hello数组的下标，并将相应的值返回给客户端。
- 简化请求和响应形式，避免使用多个类型来区分单复数。
  - 请求统一使用字符串，复数形式使用逗号分开。
  - 响应统一使用数组，单数时数组只包含一条结果。
- 客户端和服务端都传递编程语言信息，以 `lang` 值显式展示流量管理的配置效果。



协议设计

设计宗旨

- 请求参数足够简单，以方便调试，但要包含足够的信息。
- 响应参数的数据类型要尽可能覆盖全面，以实现演示的目的。

请求协议

只使用字符串类型，包含请求Hello数组的下标值和编程语言信息。

```
message TalkRequest {
  //language index
  string data = 1;
  //clientside language
  string meta = 2;
}
```

响应协议

- 响应只包含两个字段，整数类型的状态码和TalkResult类型的数组。
- `TalkResult` 类型内部，分别定义了长整型、枚举类型、键值类型（k/v的泛型为字符串）。

```
message TalkResponse {
  int32 status = 1;
  repeated TalkResult results = 2;
}
message TalkResult {
  //timestamp
  int64 id = 1;
  //enum
  ResultType type = 2;
  // result uuid
  // language index
  // data hello
  // meta serverside language (It's not good here,
  //      but ok since I feel like to keep the response)
  map<string, string> kv = 3;
}
enum ResultType {
  OK = 0;
  FAIL = 1;
}
```

功能函数

功能函数	说明
环境变量	需要为gRPC的客户端提供一个变量GRC_SERVER，在本地开发调试时其值为Localhost，在Pod启动时动态定义为gRPC Service的值，以便客户端调用。
随机数	在Client streaming和Bidirectional streaming两种通信方式下，客户端需要随机产生一个整型数值，取值要求在Hello数组下标的范围内。
时间戳	TalkResult.id是int64类型的唯一标识，通过时间戳来实现。
UUID	TalkResult.kv[id]是字符串类型的唯一标识，通过UUID来实现。
Sleep	在Streaming方式下，为了更好地观察，通过Sleep方式设置两次请求的间隔。

5.2. gRPC协议的通信模型实现

本文介绍如何使用Java、Go、NodeJS、Python编程语言实现gRPC的Unary RPC、Server streaming RPC、Client streaming RPC、Client streaming RPC通信模型。

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。

步骤一：转换代码

1. 执行以下命令，安装gRPC和Protobuf，以下以macOS为例。

   ```
   brew install grpc protobuf
   ```

2. 将Protobuf定义转换为使用的语言的代码。以下为不同语言的转换方式。

> **说明** 在示例工程中，每种语言的代码目录中都有一个proto目录，其中的*landing.proto*文件是示例工程根目录下*proto/landing.proto*文件的软连接，这样有利于统一更新Protobuf的定义。

- Java的构建工具Maven提供了自动转换插件protobuf-maven-plugin，执行 `mvn package` 会自动使用protoc-gen-grpc-java创建gRPC的模板代码。详情请参见*hello-grpc-java/pom.xml*。
- Go需要执行 `go get github.com/golang/protobuf/protoc-gen-go` 安装protoc-gen-go，然后使用protoc命令生成gRPC代码。详情请参见*hello-grpc-go/proto2go.sh*。
- NodeJS需要执行 `npm install -g grpc-tools` 安装grpc_tools_node_protoc，然后使用protoc命令生成gRPC代码。详情请参见见*hello-grpc-nodejs/proto2js.sh*。
- Python需要执行 `pip install grpcio-tools` 安装grpcio-tools，然后使用protoc命令生成gRPC代码。详情请参见*hello-grpc-python/proto2py.sh*。

步骤二：设置通信模型

1. 设置Hello数组。

   Java | Go | NodeJS | Python

   ```
   private final List<String> HELLO_LIST = Arrays.asList("Hello", "Bonjour", "Hola", "こんにちは", "Ciao", "안녕하세요");
   kv.put("data", HELLO_LIST.get(index));
   ```

2. 设置通信模型。

   - 设置Unary RPC通信模型。

     Java | Go | NodeJS | Python

     ```
     // 使用blockingStub与服务端通信
     public TalkResponse talk(TalkRequest talkRequest) {
         return blockingStub.talk(talkRequest);
     }
     //服务端处理请求后触发StreamObserver实例的两个事件onNext和onCompleted
     public void talk(TalkRequest request, StreamObserver<TalkResponse> responseObserver) {
         ...
         responseObserver.onNext(response);
         responseObserver.onCompleted();
     }
     ```

   - 设置Server streaming RPC通信模型。

     Java | Go | NodeJS | Python

   - 设置Client streaming RPC通信模型。

   - 设置Bidirectional streaming RPC通信模型。

```
public List<TalkResponse> talkOneAnswerMore(TalkRequest request) {
    Iterator<TalkResponse> talkResponses = blockingStub.talkOneAnswerMore(request);
    talkResponses.forEachRemaining(talkResponseList::add);
    return talkResponseList;
}

public void talkOneAnswerMore(TalkRequest request, StreamObserver<TalkResponse>
responseObserver) {
    String[] datas = request.getData().split(",");
    for (String data : datas) {...}
    talkResponses.forEach(responseObserver::onNext);
    responseObserver.onCompleted();
}
```

Java Go NodeJS Python

步骤三：开发功能函数

1. 设置环境变量。

```
public void talkMoreAnswerOne(List<TalkRequest> requests) throws InterruptedException {
    final CountDownLatch finishLatch = new CountDownLatch(1);
    StreamObserver<TalkResponse> responseObserver = new StreamObserver<TalkResponse> ()
{
        @Override
        public void onNext(TalkResponse talkResponse) {
            log.info("Response=\n{}", talkResponse);
        }
        @Override
        public void onCompleted() {
            finishLatch.countDown();
        }
    };
    final StreamObserver<TalkRequest> requestObserver =
asyncStub.talkMoreAnswerOne(responseObserver);
    try {
        requests.forEach(request -> {
            if (finishLatch.getCount() > 0) {
                requestObserver.onNext(request);
        });
    requestObserver.onCompleted();
}

public StreamObserver<TalkRequest> talkMoreAnswerOne(StreamObserver<TalkResponse>
responseObserver) {
    return new StreamObserver<TalkRequest>() {
        @Override
        public void onNext(TalkRequest request) {
            talkRequests.add(request);
        }
        @Override
        public void onCompleted() {
            responseObserver.onNext(buildResponse(talkRequests));
            responseObserver.onCompleted();
        }
    };
}
```

Java Go NodeJS Python

```
public void talkBidirectional(List<TalkRequest> requests) throws InterruptedException {
    final CountDownLatch finishLatch = new CountDownLatch(1);
    StreamObserver<TalkResponse> responseObserver = new StreamObserver<TalkResponse>()
{
        @Override
        public void onNext(TalkResponse talkResponse) {
            log.info("Response=\n{}", talkResponse);
        }
        @Override
        public void onCompleted() {
            finishLatch.countDown();
        }
    };
    final StreamObserver<TalkRequest> requestObserver =
asyncStub.talkBidirectional(responseObserver);
    try {
        requests.forEach(request -> {
            if (finishLatch.getCount() > 0) {
                requestObserver.onNext(request);
    ...
    requestObserver.onCompleted();
}

public StreamObserver<TalkRequest> talkBidirectional(StreamObserver<TalkResponse>
responseObserver) {
    return new StreamObserver<TalkRequest>() {
        @Override
        public void onNext(TalkRequest request) {
            responseObserver.onNext(TalkResponse.newBuilder()
                    .setStatus(200)
                    .addResults(buildResult(request.getData())).build());
        }
        @Override
        public void onCompleted() {
            responseObserver.onCompleted();
        }
    };
}
```

2. 设置随机数。

Java Go NodeJS Python

```
private static String getGrcServer() {
    String server = System.getenv("GRPC_SERVER");
    if (server == null) {
        return "localhost";
    }
    return server;
}
```

3. 设置时间戳。

Java Go NodeJS Python

```
public static String getRandomId() {
    return String.valueOf(random.nextInt(5));
}
```

Java Go NodeJS Python 设置UUID。 4.

```
TalkResult.newBuilder().setId(System.nanoTime())
```

Java Go NodeJS Python 设置Sleep。 5.

```
kv.put("id", UUID.randomUUID().toString());
```

Java Go NodeJS Python

```
TimeUnit.SECONDS.sleep(1);
```

结果验证

功能验证

在一个终端启动gRPC服务端，在另一个终端启动gRPC客户端。启动服务端和客户端后，客户端将分别对4个通信接口进行请求。

- 使用Java语言时启动gRPC服务和客户端。

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.server.ProtoServer"
```

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.client.ProtoClient"
```

- 使用Go语言时启动gRPC服务和客户端。

```
go run server.go
```

```
go run client/proto_client.go
```

- 使用NodeJS语言时启动gRPC服务和客户端。

```
node proto_server.js
```

```
node proto_client.js
```

- 使用Python语言时启动gRPC服务和客户端。

```
python server/protoServer.py
```

```
python client/protoClient.py
```

如果没有出现通信错误，则表示启动gRPC服务和客户端成功。

交叉通信

交叉通信用以确保不同编程语言实现的gRPC通信行为一致。从而保证路由到不同编程语言版本，结果是一致的。

1. 启动任意一种编程语言的gRPC服务端，以下以Java为例。

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.server.ProtoServer"
```

2. 使用4种编程语言客户端进行验证。

```
mvn exec:java -Dexec.mainClass="org.feuyeux.grpc.client.ProtoClient"
```

```
go run client/proto_client.go
```

```
node proto_client.js
```

```
python client/protoClient.py
```

如果没有出现通信错误，则表示交叉通信成功。

构建工程和镜像

完成gRPC服务端和客户端功能验证后，您还可以构建gRPC服务端和客户端的镜像。

构建工程

使用4种语言构建服务端和客户端的工程。

- Java
  分别构建服务端和客户端的JAR，将其拷贝到Docker目录备用。

```
mvn clean install -DskipTests -f server_pom
cp target/hello-grpc-java.jar ../docker/
mvn clean install -DskipTests -f client_pom
cp target/hello-grpc-java.jar ../docker/
```

- Go
  Go编译的二进制是平台相关的，最终要部署到Linux上，因此构建命令如下。然后将二进制拷贝到Docker目录备用。

```
env GOOS=linux GOARCH=amd64 go build -o proto_server server.go
mv proto_server ../docker/
env GOOS=linux GOARCH=amd64 go build -o proto_client client/proto_client.go
mv proto_client ../docker/
```

- NodeJS
  NodeJS需要在Docker镜像中进行构建，才能支持运行时所需的各种C++依赖。因此这一步主要是拷贝备用。

```
cp ../hello-grpc-nodejs/proto_server.js node
cp ../hello-grpc-nodejs/package.json node
cp -R ../hello-grpc-nodejs/common node
cp -R ../proto node
cp ../hello-grpc-nodejs/*_client.js node
```

- Python
  Python无需编译，拷贝备用即可。

```
cp -R ../hello-grpc-python/server py
cp ../hello-grpc-python/start_server.sh py
cp -R ../proto py
cp ../hello-grpc-python/proto2py.sh py
cp -R ../hello-grpc-python/client py
cp ../hello-grpc-python/start_client.sh py
```

构建GRPC服务端和客户端镜像

构建完毕后，Docker路径下存储了Dockerfile所需的全部文件，Dockerfile中重点信息说明如下。

- 基础镜像尽量选择alpine，因为尺寸最小。示例中Python的基础镜像选择的是2.7版本的python:2，您可以根据实际情况修改Python的基础镜像版本。
- NodeJS需要安装C++及编译器Make，Npm包需要安装grpc-tools。

这里以NodeJS 服务端的镜像作为示例，说明构建镜像的过程。

1. 创建*grpc-server-node.dockerfile*文件。

```
FROM node:14.11-alpine
RUN apk add --update \
      python \
      make \
      g++ \
  && rm -rf /var/cache/apk/*
RUN npm config set registry http://registry.npmmirror.com && npm install -g node-pre-gyp
grpc-tools --unsafe-perm
COPY node/package.json .
RUN npm install --unsafe-perm
COPY node .
ENTRYPOINT ["node","proto_server.js"]
```

2. 构建镜像。

```
docker build -f grpc-server-node.dockerfile -t registry.cn-beijing.aliyuncs.com/asm_repo/
grpc_server_node:1.0.0 .
```

   最终会构建出8个镜像。

3. 使用Push命令将镜像分发到容器镜像ACR中。

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_java:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_java:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_go:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_go:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_node:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_node:1.0.0
```

```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_server_python:1.0.0
```

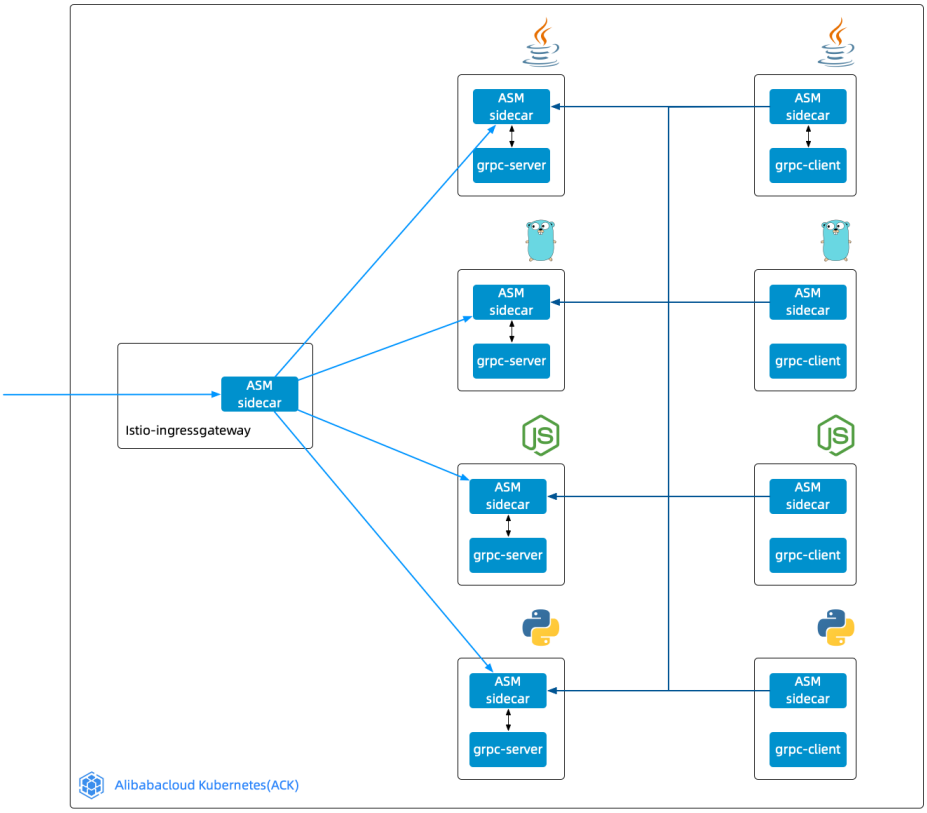
```
docker push registry.cn-beijing.aliyuncs.com/asm_repo/grpc_client_python:1.0.0
```

5.3. 实现gRPC服务端Service的负载均衡

gRPC服务端通过调用变量 `GRPC_SERVER` 定义的服务grpc-server-svc.grpc-best.svc.cluster.local，将请求均匀地路由到4个编程语言版本的服务端上，本文介绍如何在ACK上部署gRPC服务端的Service，并对Service的负载均衡进行验证。

背景信息

4个编程语言版本的客户端通过调用变量`GRPC_SERVER`定义的服务grpc-server-svc.grpc-best.svc.cluster.local，当接收到内部请求时，可以均匀地路由到4个编程语言版本的服务端上。与此同时，通过配置Istio中的Ingress Gateway，可以将外部请求按负载均衡策略路由到4个版本的gRPC服务端上。



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。

> **说明** 本文的镜像仓库仅供参考，请根据镜像脚本自行构建和推送镜像至自建仓库。关于镜像脚本的具体信息，请参见hello-servicemesh-grpc。

步骤一：创建gRPC服务端的Service

本系列的示例只有一个命名为*grpc-server-svc*的gRPC类型的Service。

> **说明** `spec.ports.name`的值需要以grpc开头。

1. 创建名为*grpc-server-svc*的YAML文件。

```
apiVersion: v1
kind: Service
metadata:
  namespace: grpc-best
  name: grpc-server-svc
  labels:
    app: grpc-server-svc
spec:
  ports:
    - port: 9996
      name: grpc-port
  selector:
    app: grpc-server-deploy
```

2. 执行以下命令，创建Service。

```
kubectl apply -f grpc-server-svc.yaml
```

步骤二：创建gRPC服务端的Deployment

完整的Deployment请参见*kube/deployment*，以下以NodeJS语言的gRPC服务端的Deployment文件*grpc-server-node.yaml*为例，创建gRPC服务端的Deployment。

> **说明** 服务端的4个Deployment都需要定义 `app` 标签的值为 `grpc-server-deploy` ，以匹配gRPC服务端Service的Selector。同时每种语言的version标签需要保证各不相同。

1. 创建名为*grpc-server-node*的YAML文件。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: grpc-best
  name: grpc-server-node
  labels:
    app: grpc-server-deploy
    version: v3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-server-deploy
      version: v3
  template:
    metadata:
      labels:
        app: grpc-server-deploy
        version: v3
    spec:
      containers:
        - name: grpc-server-deploy
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/asm-grpc-server-nod
e:1.0.0
          imagePullPolicy: Always
          ports:
            - containerPort: 9996
              name: grpc-port
```

2. 执行以下命令，创建Depolyment。

```
kubectl apply -f grpc-server-node.yaml
```

步骤三：创建gRPC客户端的Deployment

客户端和服务端有以下两处不同。

- 服务端启动后会持续运行，而客户端完成请求后就会结束进程，因此，需要实现一种死循环的方式保持客户端容器不退出。
- 客户端需要定义变量GRPC_SERVER的值，在客户端容器启动时传递给gRPC客户端。

以下以Go语言的gRPC客户端的Deployment文件*grpc-client-go.yaml*为例，创建gRPC客户端的Deployment。

1. 创建名*grpc-client-go*的YAML文件。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: grpc-best
  name: grpc-client-go
  labels:
    app: grpc-client-go
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grpc-client-go
  template:
    metadata:
      labels:
        app: grpc-client-go
    spec:
      containers:
        - name: grpc-client-go
          image: registry.cn-hangzhou.aliyuncs.com/aliacs-app-catalog/asm-grpc-client-go:
1.0.0
          command: ["/bin/sleep", "3650d"]
          env:
            - name: GRPC_SERVER
              value: "grpc-server-svc.grpc-best.svc.cluster.local"
          imagePullPolicy: Always
```

2. 执行以下命令，创建Depolyment。

```
kubectl apply -f grpc-client-go.yaml
```

其中，`command: ["/bin/sleep", "3650d"]` 是定义Go语言的gRPC客户端启动后执行的命令，通过Sleep的方式保持客户端容器运行。`env` 中定义了GRPC_SERVER变量，其值为 `grpc-server-svc.grpc-best.svc.cluster.local` 。

步骤四：部署服务和应用

1. 在ACK实例中创建名为grpc-best的命名空间。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
k create ns grpc-best
```

2. 为命名空间启用自动注入Sidecar。

```
k label ns grpc-best istio-injection=enabled
```

3. 部署Service，及8个Deployment。

```
kubectl apply -f grpc-svc.yaml
kubectl apply -f deployment/grpc-server-java.yaml
kubectl apply -f deployment/grpc-server-python.yaml
kubectl apply -f deployment/grpc-server-go.yaml
kubectl apply -f deployment/grpc-server-node.yaml
kubectl apply -f deployment/grpc-client-java.yaml
kubectl apply -f deployment/grpc-client-python.yaml
kubectl apply -f deployment/grpc-client-go.yaml
kubectl apply -f deployment/grpc-client-node.yaml
```

结果验证

从Pod侧验证gRPC服务的负载均衡

从客户端容器请求gRPC服务端的Service，验证gRPC服务端Service的负载均衡。

1. 获取4个客户端容器的名称。

```
client_java_pod=$(k get pod -l app=grpc-client-java -n grpc-best -o jsonpath={.items..met
adata.name})
```

```
client_go_pod=$(k get pod -l app=grpc-client-go -n grpc-best -o jsonpath={.items..metadat
a.name})
```

```
client_node_pod=$(k get pod -l app=grpc-client-node -n grpc-best -o jsonpath={.items..met
adata.name})
```

```
client_python_pod=$(k get pod -l app=grpc-client-python -n grpc-best -o jsonpath={.items.
.metadata.name})
```

2. 在客户端容器中，对4个gRPC服务端的Service发起请求。

```
k exec "$client_java_pod" -c grpc-client-java -n grpc-best -- java -jar /grpc-client.jar
```

```
k exec "$client_go_pod" -c grpc-client-go -n grpc-best -- ./grpc-client
```

```
k exec "$client_node_pod" -c grpc-client-node -n grpc-best -- node proto_client.js
```

```
k exec "$client_python_pod" -c grpc-client-python -n grpc-best -- sh /grpc-client/start_c
lient.sh
```

3. 以NodeJS客户端为例，通过一个循环，验证gRPC服务端Service的负载均衡。

```
for ((i = 1; i <= 100; i++)); do
kubectl exec "$client_node_pod" -c grpc-client-node -n grpc-best -- node kube_client.js >
kube_result
done
sort kube_result | grep -v "^[[:space:]]*$" | uniq -c | sort -nrk1
```

预期输出：

```
  26 Talk:PYTHON
  25 Talk:NODEJS
  25 Talk:GOLANG
  24 Talk:JAVA
```

结果显示4个版本的gRPC服务端Service收到相近的请求数。说明ASM收到外部请求时，可以将外部请求按

负载均衡策略路由到4个版本的gRPC服务端Service上。

从本地验证gRPC服务的负载均衡

从本地请求Istio中的Ingress Gateway，验证gRPC服务端Service的负载均衡。

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单击**ASM网关**，然后在右侧页面单击**使用YAML创建**。
5. 选择命名空间，将以下内容拷贝到文本框中，然后单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  namespace: grpc-best
  name: grpc-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 9996
        name: grpc
        protocol: GRPC
      hosts:
        - "*"
```

6. 获取Ingress Gateway的IP。

```
INGRESS_IP=$(k -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loa
dBalancer.ingress[0].ip}')
```

7. 验证gRPC服务端Service的负载均衡。

```
docker run -d --name grpc_client_node -e GRPC_SERVER="${INGRESS_IP}" registry.cn-hangzhou
.aliyuncs.com/aliacs-app-catalog/asm-grpc-client-node:1.0.0 /bin/sleep 3650d
client_node_container=$(docker ps -q)
docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node kube_client.
js
for ((i = 1; i <= 100; i++)); do
docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node kube_client.
js >> kube_result
done
sort kube_result | grep -v "^[[:space:]]*$" | uniq -c | sort -nrk1
```

预期输出：

```
  26 Talk:PYTHON
  25 Talk:NODEJS
  25 Talk:GOLANG
  24 Talk:JAVA
```

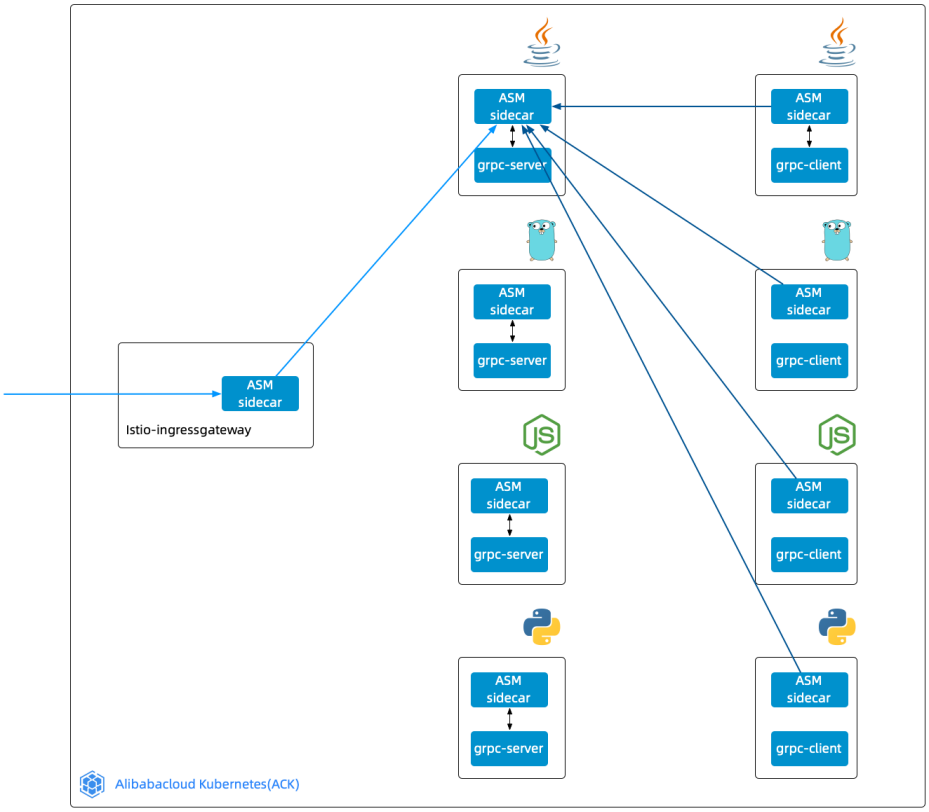
结果显示4个版本的gRPC服务端Service收到相近的请求数。说明ASM收到外部请求时，可以将外部请求按负载均衡策略路由到4个版本的gRPC服务端Service上。

5.4. 管理gRPC协议示例流量

服务网格ASM支持对gRPC协议服务端进行流量管理，本文介绍如何按照gRPC协议版本和gRPC API进行流量管理。

按gRPC协议版本进行流量管理

gRPC协议服务端Service包括Java、Go、NodeJS和Python版本，以下以100%流量路由到Java版本服务端Service作为示例。



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**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。
5. 选择命名空间，在文本框中输入以下信息，然后单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  namespace: grpc-best
  name: grpc-server-dr
spec:
  host: grpc-server-svc
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
    - name: v3
      labels:
        version: v3
    - name: v4
      labels:
        version: v4
```

6. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
7. 选择命名空间，在文本框中输入以下信息，然后单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
    - "*"
  gateways:
    - grpc-gateway
  http:
    - match:
        - port: 9996
      route:
        - destination:
            host: grpc-server-svc
            subset: v1
          weight: 100
```

执行如下命令，验证是否100%流量路由到Java版本服务端Service。

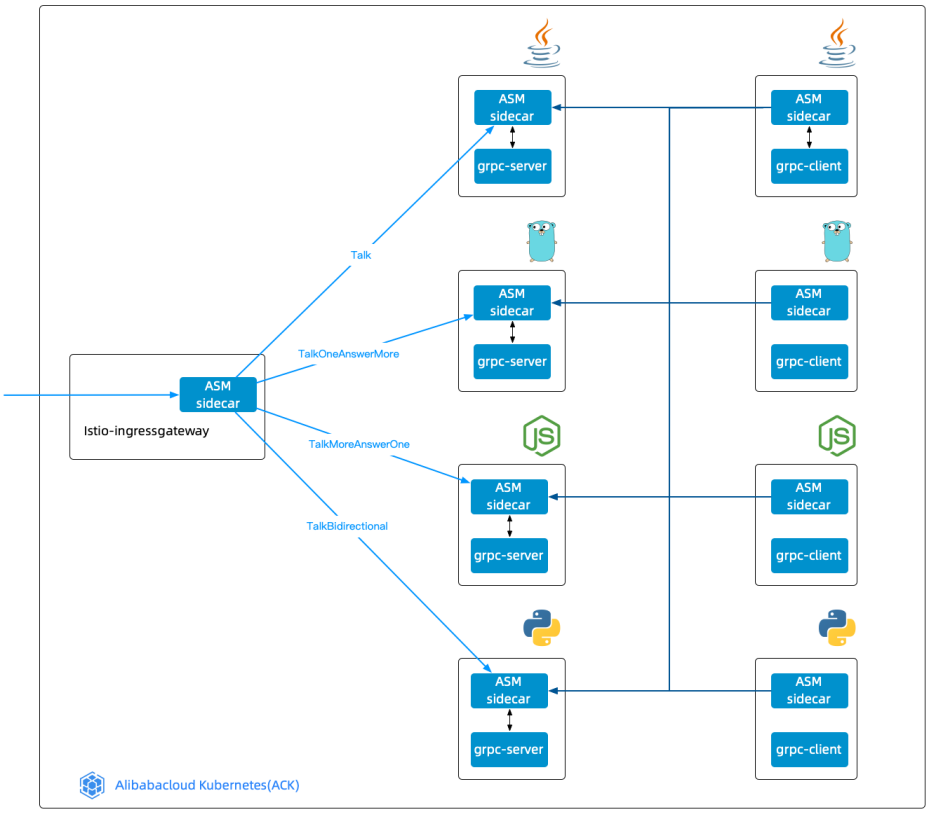
```
for i in {1..100}; do
  docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node mesh_clien
t.js >> mesh_result
done
sort mesh_result | grep -v "^[[:space:]]*$"| uniq -c | sort -nrk1
```

预期输出：

```
100 TalkOneAnswerMore:JAVA
100 TalkMoreAnswerOne:JAVA
100 TalkBidirectional:JAVA
100 Talk:JAVA
```

按gRPC API进行流量管理

按gRPC API进行流量管理是更细粒度的流量管理方式。通过实现通信模型最终构建gRPC API，详细介绍请参见gRPC协议的通信模型实现。现在有4个协议gRPC的API、4个gRPC协议版本的Service。这里演示一种极端的情况，每一种API路由到指定的一个版本的服务上。



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**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 选择命名空间，在文本框中输入以下信息，然后单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
    - "*"
  gateways:
    - grpc-gateway
  http:
    - match:
        - port: 9996
        - uri:
            exact: /org.feuyeux.grpc.LandingService/talk
      route:
        - destination:
            host: grpc-server-svc
            subset: v1
          weight: 100
    - match:
        - port: 9996
        - uri:
            exact: /org.feuyeux.grpc.LandingService/talkOneAnswerMore
      route:
        - destination:
            host: grpc-server-svc
            subset: v2
          weight: 100
    - match:
        - port: 9996
        - uri:
            exact: /org.feuyeux.grpc.LandingService/talkMoreAnswerOne
      route:
        - destination:
            host: grpc-server-svc
            subset: v3
          weight: 100
    - match:
        - port: 9996
        - uri:
            exact: /org.feuyeux.grpc.LandingService/talkBidirectional
      route:
        - destination:
            host: grpc-server-svc
            subset: v4
          weight: 100
```

执行如下命令，验证是否每一种API路由到指定的一个版本的服务。

```
for i in {1..100}; do
  docker exec -e GRPC_SERVER="${INGRESS_IP}" -it "$client_node_container" node mesh_clien
t.js >> mesh_result
done
sort mesh_result | grep -v "^[[:space:]]*$"| uniq -c | sort -nrk1
```

预期输出：

```
 100 TalkOneAnswerMore:GOLANG
 100 TalkMoreAnswerOne:NODEJS
 100 TalkBidirectional:PYTHON
 100 Talk:JAVA
```

5.5. 转移gRPC协议示例的应用流量

在ASM中定义Headers键值的匹配条件，可以根据请求动态地进行流量转移。本文介绍如何通过Headers在ASM实现应用流量转移。

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。

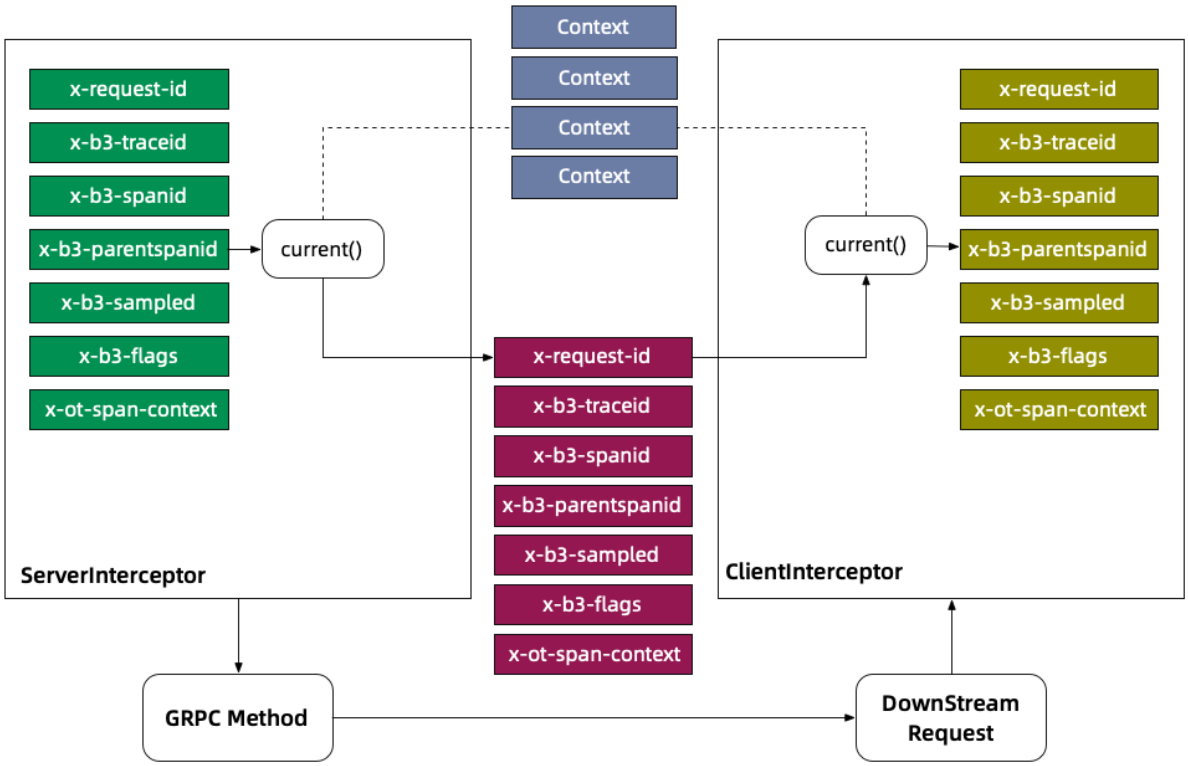
Propaganda Headers

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

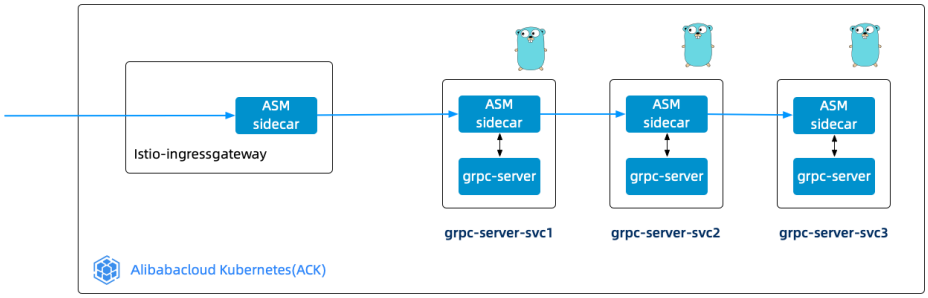
实现流量转移之前，您需要部署和验证网格拓扑，确保网格拓扑是可以通信的。

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。



流量转移

在VirtualService中通过定义Headers键值的匹配条件，可以实现根据请求动态地进行流量转移。如果再结合按API和按版本进行流量管理的实践，就可以完成应用级的精细化流量管理。流量管理的详细介绍请参见管理gRPC协议示例流量。以下VirtualService定义了Headers中 `server-version=go` 的请求100%流量路由到Go版本服务。

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**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
5. 选择命名空间，在文本框中输入以下信息，然后单击**创建**。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  namespace: grpc-best
  name: grpc-server-vs
spec:
  hosts:
    - "*"
  gateways:
    - grpc-gateway
  http:
    - match:
      - headers:
          server-version:
            exact: go
      route:
        - destination:
            host: grpc-server-svc
            subset: v2
          weight: 100
```

6.服务网格Flagger

6.1. 基于Mixerless Telemetry实现服务网格的可观测性

服务网格ASM的Mixerless Telemetry技术，为业务容器提供了无侵入式的遥测数据。遥测数据作为监控指标被ARMS Prometheus或Prometheus采集，实现服务网格可观测性。本文以Prometheus为例，介绍如何基于ASM采集应用监控指标实现服务网格的可观测性。

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。

步骤一：安装Prometheus

1. 下载Istio安装包，并解压。关于Istio安装包下载地址，请参见Download Istio。
2. 通过kubectl连接集群。具体操作，请参见通过kubectl工具连接集群。
3. 执行以下命令，安装Prometheus。

```
kubectl --kubeconfig <kubeconfig位置> apply -f <解压后Istio安装包位置>/samples/addons/prometheus.yaml
```

步骤二：创建服务条目

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**网格实例 > 基本信息**，然后在右侧页面单击**功能设置**。
5. 在**功能设置更新**面板选中**开启采集Prometheus监控指标**，选择**启用已有Prometheus**，在文本框中输入Prometheus服务地址，本文使用默认服务地址http://prometheus:9090，然后单击**确定**。

> **说明** 本文已自建Prometheus为例，如果您使用的是ARMS Prometheus，请参见集成ARMS Prometheus实现网格监控。

在网格详情页面左侧导航栏选择**流量管理 > 服务条目**，可以看到生成的相关EnvoyFilter。

步骤三：配置Prometheus

1. 配置Istio的监控指标。
   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**配置管理 > 配置项**。
   v. 在**配置项**页面顶部设置命名空间为Istio-sysytem，单击prometheus**操作**列下的**编辑**。
   vi. 在**编辑**面板**值**文本框中添加配置信息，然后单击**确定**。

2. 删除Prometheus Pod，使Prometheus配置生效。
   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**工作负载 > 容器组**。
   v. 在**容器组**页面单击Prometheus**操作**列下的**删除**。
   vi. 在**删除容器组**对话框单击**确定**。
3. 执行以下命令，查看Prometheus配置中的 `job_name` 。

```
kubectl --kubeconfig <kubeconfig位置> get cm prometheus -n istio-system -o jsonpath={.data
.prometheus\\.yml} | grep job_name
```

预期输出：

```
- job_name: 'istio-mesh'
- job_name: 'envoy-stats'
- job_name: 'istio-policy'
- job_name: 'istio-telemetry'
- job_name: 'pilot'
- job_name: 'sidecar-injector'
- job_name: prometheus
  job_name: kubernetes-apiservers
  job_name: kubernetes-nodes
  job_name: kubernetes-nodes-cadvisor
- job_name: kubernetes-service-endpoints
- job_name: kubernetes-service-endpoints-slow
  job_name: prometheus-pushgateway
- job_name: kubernetes-services
- job_name: kubernetes-pods
- job_name: kubernetes-pods-slow
```

步骤四：生成监控指标数据

1. 执行以下命令，在ACK集群中部署podinfo示例应用。
   i. 下载podinfo示例应用的YAML文件。关于podinfo示例应用的下载地址，请参见podinfo。
   ii. 执行以下命令，部署podinfo示例应用。

```
kubectl --kubeconfig <kubeconfig位置> apply -f <podinfo位置>/kustomize/deployment.yaml
-n test
kubectl --kubeconfig <kubeconfig位置> apply -f <podinfo位置>/kustomize/service.yaml -n
test
```

2. 执行以下命令，请求podinfo应用，以产生监控指标数据。

```
podinfo_pod=$(k get po -n test -l app=podinfo -o jsonpath={.items..metadata.name})
for i in {1..10}; do
   kubectl --kubeconfig "$USER_CONFIG" exec $podinfo_pod -c podinfod -n test -- curl -s p
odinfo:9898/version
  echo
done
```

3. 在Envoy容器内确认监控指标已生成。

i. 执行以下命令，请求Envoy确认是否包含istio_requests_total监控指标。

```
kubectl --kubeconfig <kubeconfig位置> exec $podinfo_pod -n test -c istio-proxy -- curl
-s localhost:15090/stats/prometheus | grep istio_requests_total
```

预期输出：

```
:::: istio_requests_total ::::
# TYPE istio_requests_total counter
istio_requests_total{response_code="200",reporter="destination",source_workload="podi
nfo",source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/tes
t/sa/default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4
e3104a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespa
ce="test",destination_principal="spiffe://cluster.local/ns/test/sa/default",destinati
on_app="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.
cluster.local",destination_service_name="podinfo",destination_service_namespace="test
",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",res
ponse_flags="-",grpc_response_status="",connection_security_policy="mutual_tls",sourc
e_canonical_service="podinfo",destination_canonical_service="podinfo",source_canonica
l_revision="latest",destination_canonical_revision="latest"} 10
istio_requests_total{response_code="200",reporter="source",source_workload="podinfo",
source_workload_namespace="test",source_principal="spiffe://cluster.local/ns/test/sa/
default",source_app="podinfo",source_version="unknown",source_cluster="c199d81d4e3104
a5d90254b2a210914c8",destination_workload="podinfo",destination_workload_namespace="t
est",destination_principal="spiffe://cluster.local/ns/test/sa/default",destination_ap
p="podinfo",destination_version="unknown",destination_service="podinfo.test.svc.clust
er.local",destination_service_name="podinfo",destination_service_namespace="test",des
tination_cluster="c199d81d4e3104a5d90254b2a210914c8",request_protocol="http",response
_flags="-",grpc_response_status="",connection_security_policy="unknown",source_canoni
cal_service="podinfo",destination_canonical_service="podinfo",source_canonical_revisi
on="latest",destination_canonical_revision="latest"} 10
```

ii. 执行以下命令，请求Envoy确认是否包含istio_request_duration监控指标。

```
kubectl --kubeconfig <kubeconfig位置> exec $podinfo_pod -n test -c istio-proxy -- curl
-s localhost:15090/stats/prometheus | grep istio_request_duration
```

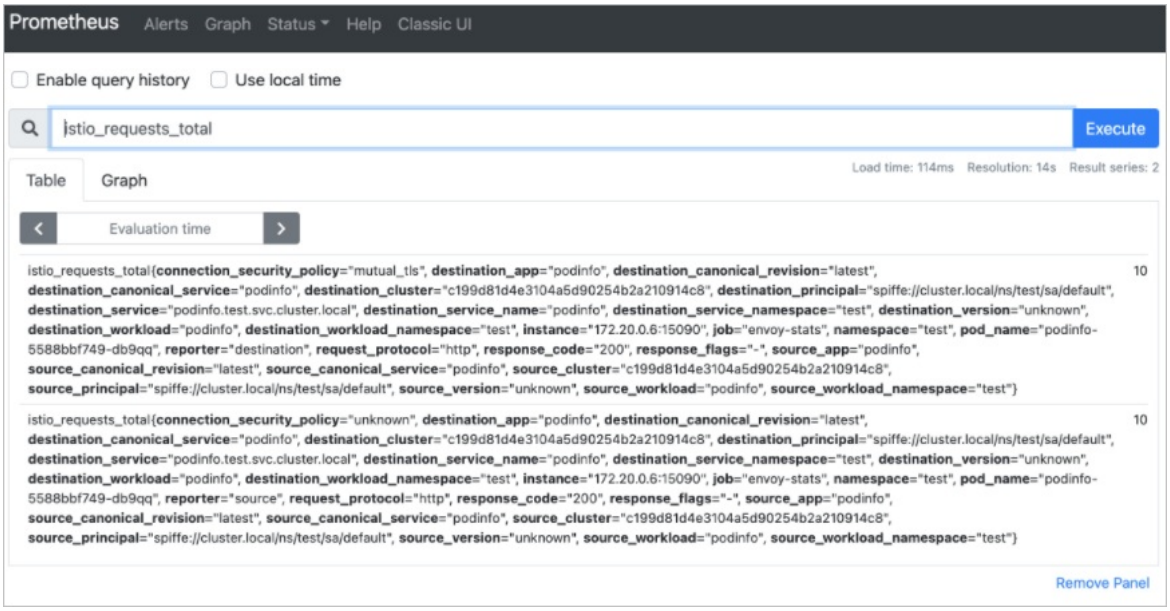
预期输出：

```
:::: istio_request_duration ::::
# TYPE istio_request_duration_milliseconds histogram
istio_request_duration_milliseconds_bucket{response_code="200",reporter="destination"
,source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe:
//cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",sou
rce_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destin
ation_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test
/sa/default",destination_app="podinfo",destination_version="unknown",destination_serv
ice="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_s
ervice_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",reque
st_protocol="http",response_flags="-",grpc_response_status="",connection_security_pol
icy="mutual_tls",source_canonical_service="podinfo",destination_canonical_service="po
dinfo",source_canonical_revision="latest",destination_canonical_revision="latest",le=
"0.5"} 10
istio_request_duration_milliseconds_bucket{response_code="200",reporter="destination"
,source_workload="podinfo",source_workload_namespace="test",source_principal="spiffe:
//cluster.local/ns/test/sa/default",source_app="podinfo",source_version="unknown",sou
rce_cluster="c199d81d4e3104a5d90254b2a210914c8",destination_workload="podinfo",destin
ation_workload_namespace="test",destination_principal="spiffe://cluster.local/ns/test
/sa/default",destination_app="podinfo",destination_version="unknown",destination_serv
ice="podinfo.test.svc.cluster.local",destination_service_name="podinfo",destination_s
ervice_namespace="test",destination_cluster="c199d81d4e3104a5d90254b2a210914c8",reque
st_protocol="http",response_flags="-",grpc_response_status="",connection_security_pol
icy="mutual_tls",source_canonical_service="podinfo",destination_canonical_service="po
dinfo",source_canonical_revision="latest",destination_canonical_revision="latest",le=
"1"} 10
...
```

结果验证

1. 使用负载均衡的方式对外暴露Prometheus服务。具体操作，请参见管理服务。
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**网络 > 服务**。
6. 在**服务**页面单击Prometheus**外部端点**列下的IP地址。
7. 在Prometheus中输入istio_requests_total，单击**Execute**。

   如下图所示，说明Prometheus采集应用监控指标成功。



6.2. 基于Mixerless Telemetry实现应用扩缩容

服务网格ASM的Mixerless Telemetry技术，为业务容器提供了无侵入式的遥测数据。您可以使用Prometheus采集应用的请求数量、请求延迟、P99分布等指标。HPA可以根据采集到的请求数量、请求延迟、P99分布等指标进行自动扩缩容。本文介绍如何基于Mixerless Telemetry实现应用扩缩容。

前提条件

使用Prometheus采集应用监控指标。具体操作，请参见基于Mixerless Telemetry实现服务网格的可观测性。

步骤一：部署metrics-adapter和flagger loadtester

1. 通过kubectl连接集群。具体操作，请参见通过kubectl工具连接集群。
2. 执行以下命令，部署metrics-adapter。

   > 说明 关于metrics-adapter完整脚本，请参见demo_hpa.sh。

   ```
   helm --kubeconfig <kubeconfig位置> -n kube-system install asm-custom-metrics \  $KUBE_METRICS_ADAPTER_SRC/deploy/charts/kube-metrics-adapter \
     --set prometheus.url=http://prometheus.istio-system.svc:9090
   ```

3. 验证metrics-adapter部署情况。
   i. 执行以下命令，查看metrics-adapter的Pod部署情况。

   ```
   kubectl --kubeconfig <kubeconfig位置> get po -n kube-system | grep metrics-adapter
   ```

   预期输出：

   ```
   asm-custom-metrics-kube-metrics-adapter-6fb4949988-ht8pv   1/1     Running     0          30s
   ```

ii. 执行以下命令，查看autoscaling/v2beta的CRD部署情况。

```
kubectl --kubeconfig <kubeconfig位置> api-versions | grep "autoscaling/v2beta"
```

预期输出：

```
autoscaling/v2beta1
autoscaling/v2beta2
```

iii. 执行以下命令，查看metrics-adapter部署情况。

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

预期输出：

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

4. 部署flagger loadtester。

i. 下载flagger loadtester的YAML文件。关于flagger loadtester的YAML文件下载地址，请参见flagger。

ii. 执行以下命令，部署flagger loadtester。

```
kubectl --kubeconfig <kubeconfig位置> apply -f <flagger的位置>/kustomize/tester/deploy
ment.yaml -n test
kubectl --kubeconfig <kubeconfig位置> apply -f <flagger的位置>/kustomize/tester/servic
e.yaml -n test
```

步骤二：创建HPA

1. 创建根据应用请求数量（istio_requests_total）扩缩容的HPA。

i. 使用以下内容，创建一个名为*requests_total_hpa*的YAML文件。

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-total
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/processed-requests-per-second:
|
      sum(rate(istio_requests_total{destination_workload_namespace="test",reporter="d
estination"}[1m]))
spec:
  maxReplicas: 5
  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"
```

- annotations：设置应用的Pod会受到应用请求数量（istio_requests_total）影响进行扩缩容。
- target：当应用请求数量的平均值大于等于10时，应用的Pod会自动扩容。

ii. 执行以下命令，部署HPA。

```
kubectl --kubeconfig <kubeconfig位置> apply -f resources_hpa/requests_total_hpa.yaml
```

iii. 验证HPA部署情况。

```
kubectl --kubeconfig <kubeconfig位置> 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"
      ]
    }
  ]
}
```

2. 使用以下内容，部署根据平均延迟（istio_request_duration_milliseconds_sum）扩缩容的HPA。

   部署根据平均延迟扩缩容的HPA的具体步骤，请参见上文。

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-latency-avg
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/latency-average: |
      sum(rate(istio_request_duration_milliseconds_sum{destination_workload_namespace="te
st",reporter="destination"}[1m]))
      /sum(rate(istio_request_duration_milliseconds_count{destination_workload_namespace=
"test",reporter="destination"}[1m]))
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
    - type: External
      external:
        metric:
          name: prometheus-query
          selector:
            matchLabels:
              query-name: latency-average
        target:
          type: AverageValue
          averageValue: "0.005"
```

- annotations：设置HPA会受到平均延迟（istio_request_duration_milliseconds_sum）影响进行扩缩容。
- target：当平均延迟的平均值大于等于0.005s时，应用的Pod会进行自动扩容。

3. 使用以下内容，部署根据P95分布（istio_request_duration_milliseconds_bucket）扩缩容的HPA。

   部署根据P95分布扩缩容的HPA的具体步骤，请参见上文。

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: podinfo-p95
  namespace: test
  annotations:
    metric-config.external.prometheus-query.prometheus/p95-latency: |
      histogram_quantile(0.95,sum(irate(istio_request_duration_milliseconds_bucket{destin
ation_workload_namespace="test",destination_canonical_service="podinfo"}[5m]))by (le))
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  metrics:
    - type: External
      external:
        metric:
          name: prometheus-query
          selector:
            matchLabels:
              query-name: p95-latency
        target:
          type: AverageValue
          averageValue: "4"
```

- annotations：设置应用的Pod会受到P95分布（istio_request_duration_milliseconds_bucket）影响进行扩缩容。
- target：当P95分布的平均值大于等于4时，应用的Pod会自动扩容。

验证应用扩缩容

以下以应用请求数量为例，验证当应用请求数量大于等于10时，应用的Pod是否会自动进行扩容。

1. 执行以下命令，运行了一个持续5分钟、QPS为10、并发数为2的请求。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
loadtester=$(k -n test get pod -l "app=flagger-loadtester" -o jsonpath='{.items..metadata
.name}')
k -n test exec -it ${loadtester} -c loadtester -- hey -z 5m -c 2 -q 10 http://podinfo:989
8
```

   - `-z`：请求持续时间。
   - `-c`：请求数量。
   - `-q`：请求并发数。

2. 执行以下命令，查看Pod扩容情况。

```
watch kubectl --kubeconfig $USER_CONFIG -n test get hpa/podinfo-total
```

   预期输出：

```
Every 2.0s: kubectl --kubeconfig /Users/han/shop_config/ack_zjk -n test get hpa/podinfo
East6C16G: Tue Jan 26 18:01:30 2021
NAME      REFERENCE            TARGETS           MINPODS   MAXPODS   REPLICAS   AGE
podinfo   Deployment/podinfo   10056m/10 (avg)   1         5         2          4m45s
```

可以看到 `REPLICAS` 为2，说明当前应用的Pod数量为2。

3. 执行以下命令，运行了一个持续5分钟、QPS为15、并发数为2的请求。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
loadtester=$(k -n test get pod -l "app=flagger-loadtester" -o jsonpath='{.items..metadata
.name}')
k -n test exec -it ${loadtester} -c loadtester -- hey -z 5m -c 2 -q 15 http://podinfo:989
8
```

4. 执行以下命令，查看Pod扩容情况。

```
watch kubectl --kubeconfig $USER_CONFIG -n test get hpa/podinfo-total
```

预期输出：

```
Every 2.0s: kubectl --kubeconfig /Users/han/shop_config/ack_zjk -n test get hpa/podinfo
East6C16G: Tue Jan 26 18:01:30 2021
NAME      REFERENCE            TARGETS           MINPODS   MAXPODS   REPLICAS   AGE
podinfo   Deployment/podinfo   10056m/10 (avg)   1         5         3          4m45s
```

可以看到 `REPLICAS` 为3，当前应用的Pod数量为3。说明随着请求数增加，应用的Pod自动进行了扩容。减少对应用的请求，可以看到 `REPLICAS` 逐渐变为1。随着请求数减少，应用的Pod自动进行了缩容。

6.3. 基于Mixerless Telemetry实现渐进式灰度发布

服务网格ASM的Mixerless Telemetry技术，为业务容器提供了无侵入式的遥测数据。遥测数据作为监控指标被Prometheus采集。Flagger是一个应用发布流程全自动的工具，Flagger可以监控Prometheus中访问指标控制灰度发布的流量。本文介绍如何基于Mixerless Telemetry实现渐进式灰度发布。

前提条件

使用Prometheus采集应用监控指标。具体操作，请参见基于Mixerless Telemetry实现服务网格的可观测性。

渐进式灰度发布流程

1. 接入Prometheus，使Prometheus采集应用监控指标。
2. 部署Flagger和Gateway。
3. 部署flagger-loadtester，用于探测灰度发布阶段应用的Pod实例。
4. 部署3.1.0版本的podinfo应用，作为示例应用。
5. 部署HPA，设置当podinfo应用的CPU使用率达到99%时，容器会进行扩容。
6. 部署Canary，设置当P99分布的数值持续30s达到500时，逐步按照10%的比例增加导向podinfo应用的流量。
7. Flagger会复制podinfo应用，生成一个名为podinfo-primary的应用。podinfo将作为灰度版本的Deployment，podinfo-primary将作为生产版本的Deployment。
8. 升级podinfo，将灰度版本的podinfo应用升级为3.1.1版本。

9. Flagger监控Prometheus中访问指标控制灰度发布的流量。当P99分布的数值持续30s达到500时，Flagger逐步按照10%的比例增加导向3.1.1版本的podinfo应用的流量。同时HPA会根据灰度情况，逐步扩容podinfo的Pod，缩容podinfo-primary的Pod。

操作步骤

1. 通过kubectl工具连接集群。
2. 执行以下命令，部署Flagger。

```
alias k="kubectl --kubeconfig $USER_CONFIG"
alias h="helm --kubeconfig $USER_CONFIG"
cp $MESH_CONFIG kubeconfig
k -n istio-system create secret generic istio-kubeconfig --from-file kubeconfig
k -n istio-system label secret istio-kubeconfig istio/multiCluster=true
h repo add flagger https://flagger.app
h repo update
k apply -f $FLAAGER_SRC/artifacts/flagger/crd.yaml
h upgrade -i flagger flagger/flagger --namespace=istio-system \
    --set crd.create=false \
    --set meshProvider=istio \
    --set metricsServer=http://prometheus:9090 \
    --set istio.kubeconfig.secretName=istio-kubeconfig \
    --set istio.kubeconfig.key=kubeconfig
```

3. 通过kubectl连接ASM实例。
4. 部署网关规则。
   i. 使用以下内容，创建名为*public-gateway*的YAML文件。

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: public-gateway
  namespace: istio-system
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
```

   ii. 执行以下命令，部署网关规则。

```
kubectl --kubeconfig <ASM kubeconfig的位置> apply -f resources_canary/public-gateway.yaml
```

5. 执行以下命令，在ACK集群部署flagger-loadtester。

```
kubectl --kubeconfig <ACK kubeconfig的位置> apply -k "https://github.com/fluxcd/flagger//kustomize/tester?ref=main"
```

6. 执行以下命令，在ACK集群部署podInfo和HPA。

```
kubectl --kubeconfig <ACK kubeconfig的位置> apply -k "https://github.com/fluxcd/flagger//kustomize/podinfo?ref=main"
```

7. 在ACK集群部署Canary。

> **说明** 关于Canary的详细介绍，请参见How it works。

i. 使用以下内容，创建名为*podinfo-canary*的YAML文件。

```yaml
apiVersion: flagger.app/v1beta1
kind: Canary
metadata:
  name: podinfo
  namespace: test
spec:
  # deployment reference
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  # the maximum time in seconds for the canary deployment
  # to make progress before it is rollback (default 600s)
  progressDeadlineSeconds: 60
  # HPA reference (optional)
  autoscalerRef:
    apiVersion: autoscaling/v2beta2
    kind: HorizontalPodAutoscaler
    name: podinfo
  service:
    # service port number
    port: 9898
    # container port number or name (optional)
    targetPort: 9898
    # Istio gateways (optional)
    gateways:
    - public-gateway.istio-system.svc.cluster.local
    # Istio virtual service host names (optional)
    hosts:
    - '*'
    # Istio traffic policy (optional)
    trafficPolicy:
      tls:
        # use ISTIO_MUTUAL when mTLS is enabled
        mode: DISABLE
    # Istio retry policy (optional)
    retries:
      attempts: 3
      perTryTimeout: 1s
      retryOn: "gateway-error,connect-failure,refused-stream"
  analysis:
    # schedule interval (default 60s)
    interval: 1m
    # max number of failed metric checks before rollback
    threshold: 5
    # max traffic percentage routed to canary
```

```
    # percentage (0-100)
    maxWeight: 50
    # canary increment step
    # percentage (0-100)
    stepWeight: 10
    metrics:
    - name: request-success-rate
      # minimum req success rate (non 5xx responses)
      # percentage (0-100)
      thresholdRange:
        min: 99
      interval: 1m
    - name: request-duration
      # maximum req duration P99
      # milliseconds
      thresholdRange:
        max: 500
      interval: 30s
    # testing (optional)
    webhooks:
      - name: acceptance-test
        type: pre-rollout
        url: http://flagger-loadtester.test/
        timeout: 30s
        metadata:
          type: bash
          cmd: "curl -sd 'test' http://podinfo-canary:9898/token | grep token"
      - name: load-test
        url: http://flagger-loadtester.test/
        timeout: 5s
        metadata:
          cmd: "hey -z 1m -q 10 -c 2 http://podinfo-canary.test:9898/"apiVersion: fla
gger.app/v1beta1
kind: Canary
metadata:
  name: podinfo
  namespace: test
spec:
  # deployment reference
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: podinfo
  # the maximum time in seconds for the canary deployment
  # to make progress before it is rollback (default 600s)
  progressDeadlineSeconds: 60
  # HPA reference (optional)
  autoscalerRef:
    apiVersion: autoscaling/v2beta2
    kind: HorizontalPodAutoscaler
    name: podinfo
  service:
    # service port number
    port: 9898
    # container port number or name (optional)
```

```
    targetPort: 9898
    # Istio gateways (optional)
    gateways:
    - public-gateway.istio-system.svc.cluster.local
    # Istio virtual service host names (optional)
    hosts:
    - '*'
    # Istio traffic policy (optional)
    trafficPolicy:
      tls:
        # use ISTIO_MUTUAL when mTLS is enabled
        mode: DISABLE
    # Istio retry policy (optional)
    retries:
      attempts: 3
      perTryTimeout: 1s
      retryOn: "gateway-error,connect-failure,refused-stream"
  analysis:
    # schedule interval (default 60s)
    interval: 1m
    # max number of failed metric checks before rollback
    threshold: 5
    # max traffic percentage routed to canary
    # percentage (0-100)
    maxWeight: 50
    # canary increment step
    # percentage (0-100)
    stepWeight: 10
    metrics:
    - name: request-success-rate
      # minimum req success rate (non 5xx responses)
      # percentage (0-100)
      thresholdRange:
        min: 99
      interval: 1m
    - name: request-duration
      # maximum req duration P99
      # milliseconds
      thresholdRange:
        max: 500
      interval: 30s
    # testing (optional)
    webhooks:
      - name: acceptance-test
        type: pre-rollout
        url: http://flagger-loadtester.test/
        timeout: 30s
        metadata:
          type: bash
          cmd: "curl -sd 'test' http://podinfo-canary:9898/token | grep token"
      - name: load-test
        url: http://flagger-loadtester.test/
        timeout: 5s
        metadata:
          cmd: "hey -z 1m -q 10 -c 2 http://podinfo-canary.test:9898/"
```

- `stepWeight`：设置逐步切流的百分比，本文设置为10。
- `max`：设置P99分布的值。
- `interval`：设置P99分布持续时间。

ii. 执行以下命令，部署Canary。

```
kubectl --kubeconfig <ACK kubeconfig的位置> apply -f resources_canary/podinfo-canary.yaml
```

8. 执行以下命令，将podinfo从3.1.0升级到3.1.1版本。

```
kubectl --kubeconfig <ACK kubeconfig的位置> -n test set image deployment/podinfo podinfod=stefanprodan/podinfo:3.1.1
```

验证渐进式发布

执行以下命令，查看渐进式切流过程。

```
while true; do kubectl --kubeconfig <ACK kubeconfig的位置> -n test describe canary/podinfo; sleep 10s;done
```

预期输出：

```
Events:
  Type     Reason  Age                From     Message
  ----     ------  ----               ----     -------
  Warning  Synced  39m                flagger  podinfo-primary.test not ready: waiting for rollout to finish: observed deployment generation less then desired generation
  Normal   Synced  38m (x2 over 39m)  flagger  all the metrics providers are available!
  Normal   Synced  38m                flagger  Initialization done! podinfo.test
  Normal   Synced  37m                flagger  New revision detected! Scaling up podinfo.test
  Normal   Synced  36m                flagger  Starting canary analysis for podinfo.test
  Normal   Synced  36m                flagger  Pre-rollout check acceptance-test passed
  Normal   Synced  36m                flagger  Advance podinfo.test canary weight 10
  Normal   Synced  35m                flagger  Advance podinfo.test canary weight 20
  Normal   Synced  34m                flagger  Advance podinfo.test canary weight 30
  Normal   Synced  33m                flagger  Advance podinfo.test canary weight 40
  Normal   Synced  29m (x4 over 32m)  flagger  (combined from similar events): Promotion completed! Scaling down podinfo.test
```

可以看到，流向3.1.1版本的podinfo应用的流量逐渐从10%切到40%。

7.命名空间授权控制

7.1. 对命名空间下服务访问通信进行授权控制

Kubernetes集群中命名空间之间的服务访问默认没有限制，例如开发环境命名空间中的服务可以调用生产环境中服务。您可以使用ASM零安全体系，动态配置策略，实现禁止A命名空间下的全部服务访问B命名空间下的服务，从而降低风险。本文以demo-frontend和demo-server命名空间为例，介绍如何使用授权策略实现对跨命名空间下服务访问的控制。

前提条件

添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

步骤一：注入Sidecar代理

为命名空间注入Sidecar代理，便于对该命名空间下的服务进行授权管理。

1. 新建demo-frontend和demo-server命名空间。
   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. 在网格详情页面左侧导航栏选择**网格实例 > 全局命名空间**，然后在右侧页面单击**新建**。
   v. 在**新建命名空间**面板设置名称为demo-frontend，然后单击**确定**。
   vi. 重复执行以上步骤，创建demo-server命名空间。
2. 为demo-frontend和demo-server命名空间注入Sidecar代理。
   i. 在**全局命名空间**页面单击demo-frontend命名空间右侧**自动注入**列下的**启用Sidecar自动注入**。
   ii. 在**确认**对话框单击**确定**。
   iii. 重复执行以上步骤，为demo-server命名空间注入Sidecar代理。

步骤二：创建测试服务

在demo-frontend命名空间下创建发起请求的sleep服务，在demo-server命名空间下创建被访问的httpbin服务。

1. 通过kubectl工具连接集群。
2. 在demo-frontend命名空间下创建sleep服务。
   i. 使用以下内容，创建*sleep.yaml*。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sleep
---
apiVersion: v1
kind: Service
metadata:
  name: sleep
  labels:
    app: sleep
    service: sleep
spec:
  ports:
  - port: 80
    name: http
  selector:
    app: sleep
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sleep
  template:
    metadata:
      labels:
        app: sleep
    spec:
      terminationGracePeriodSeconds: 0
      serviceAccountName: sleep
      containers:
      - name: sleep
        image: curlimages/curl
        command: ["/bin/sleep", "3650d"]
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - mountPath: /etc/sleep/tls
          name: secret-volume
      volumes:
      - name: secret-volume
        secret:
          secretName: sleep-secret
          optional: true
---
```

ii. 执行以下命令，创建sleep服务。

```
kubectl apply -f sleep.yaml -n demo-frontend
```

3. 在demo-server命名空间下创建httpbin服务。

i. 使用以下内容，创建*httpbin.yaml*。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: httpbin
---
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
    service: httpbin
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: httpbin
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpbin
spec:
  replicas: 1
  selector:
    matchLabels:
      app: httpbin
      version: v1
  template:
    metadata:
      labels:
        app: httpbin
        version: v1
    spec:
      serviceAccountName: httpbin
      containers:
      - image: docker.io/kennethreitz/httpbin
        imagePullPolicy: IfNotPresent
        name: httpbin
        ports:
        - containerPort: 80
```

ii. 执行以下命令，创建httpbin服务。

```
kubectl apply -f httpbin.yaml -n demo-server
```

4. 验证测试服务是否成功注入Sidecar。

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**工作负载 > 容器组**。

v. 在**容器组**页面单击sleep服务的容器组名称。

在**容器**页签下可以看到istio-proxy，说明sleep服务注入Sidecar成功。

vi. 重复执行以上步骤，可以看到httpbin服务注入Sidecar成功。

步骤三：创建对等身份认证

创建对等身份认证，便于授权策略使用TLS对服务进行授权。

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**零信任安全 > 对等身份认证**。
5. 在**对等身份认证**页面单击**创建双向mTLS模式**。
6. 设置**命名空间**为demo-frontend，输入名称，设置**mTLS模式**为STRICT模式，然后单击**创建**。
7. 重复执行以上步骤，为demo-server命名空间启用双向对等身份认证。

步骤四：验证使用授权策略实现对跨命名空间服务访问的控制

通过修改授权策略的动作，您可以禁止或允许demo-frontend命名空间下的服务访问demo-server下的服务，实现对跨命名空间服务访问的控制。

1. 创建授权策略，禁止来自demo-frontend命名空间的访问请求。

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. 在网格详情页面左侧导航栏选择**零信任安全 > 授权策略**，然后在右侧页面单击**创建**。

v. 设置授权策略参数，然后单击**创建**。

参数	说明
命名空间	设置**命名空间**为demo-server。
名称	输入授权策略的名称。
策略	设置**策略**为**RULES**。
动作	设置**动作**为**DENY**。
请求来源	打开**请求来源**开关，单击**增加请求来源到列表中**，单击**增加请求来源**，设置**请求来源域**为**namespaces**，**值**为demo-frontend。

2. 访问httpbin服务。

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**工作负载 > 容器组**。

v. 在**容器组**页面单击sleep容器右侧**操作**列下的**终端**，单击**容器：sleep**。

vi. 在容器组终端中执行以下命令，访问httpbin服务。

```
curl -I httpbin.demo-server.svc.cluster.local:8000
```

预期输出：

```
HTTP/1.1 403 Foribidden
```

返回以上结果，说明访问httpbin服务失败，demo-frontend命名空间下的服务访问demo-server下的服务失败。

3. 修改授权策略动作为ALLOW，允许来自demo-frontend命名空间的访问请求。
   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. 在网格详情页面左侧导航栏选择**零信任安全 > 授权策略**。
   v. 在**授权策略**页面单击目标策略右侧**操作**列下的**YAML**。
   vi. 在**编辑**面板修改action参数值为*ALLOW*，然后单击**确定**。
4. 在sleep容器组终端中执行以下命令，访问httpbin服务。

```
curl -I httpbin.demo-server.svc.cluster.local:8000
```

预期输出：

```
HTTP/1.1 200 OK
```

返回以上结果，说明访问httpbin服务成功，demo-frontend命名空间下的服务可以正常访问demo-server下的服务。
可以看到，创建动作为Deny的授权策略后，demo-frontend命名空间下的服务访问demo-server下的服务失败，然后修改授权策略动作为ALLOW，demo-frontend命名空间下的服务访问demo-server下的服务成功，说明通过授权策略控制跨命名空间服务访问成功。

7.2. 对命名空间下服务访问外部网站进行授权控制

Kubernetes集群中的NetworkPolicy能够控制某些外部网站允许或禁止被某些命名空间下的服务访问，但是NetworkPolicy的方式对网络隔离比较粗粒度，对应用安全、业务安全的保护不够到位。您可以使用ASM零安全体系，动态配置授权策略，实现对命名空间下服务访问外部网站进行授权控制，从而降低风险。本文以demo-frontend命名空间和aliyun.com网站为例，介绍如何限制demo-frontend命名空间下的全部服务去访问外部网站aliyun.com 。

前提条件

添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

步骤一：注入Sidecar代理

为命名空间注入Sidecar代理，便于对该命名空间下的服务进行授权管理。

1. 新建demo-frontend命名空间。
   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. 在网格详情页面左侧导航栏选择**网格实例 > 全局命名空间**，然后在右侧页面单击**新建**。

v. 在**新建命名空间**面板设置名称为demo-frontend，然后单击**确定**。

2. 为demo-frontend命名空间注入Sidecar代理。

i. 在**全局命名空间**页面单击demo-frontend命名空间右侧**自动注入**列下的**启用Sidecar自动注入**。

ii. 在**确认**对话框单击**确定**。

步骤二：创建测试服务

在demo-frontend命名空间下创建sleep服务。

1. 通过kubectl工具连接集群。
2. 在demo-frontend命名空间下创建sleep服务。

i. 使用以下内容，创建*sleep.yaml*。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sleep
---
apiVersion: v1
kind: Service
metadata:
  name: sleep
  labels:
    app: sleep
    service: sleep
spec:
  ports:
  - port: 80
    name: http
  selector:
    app: sleep
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sleep
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sleep
  template:
    metadata:
      labels:
        app: sleep
    spec:
      terminationGracePeriodSeconds: 0
      serviceAccountName: sleep
      containers:
      - name: sleep
        image: curlimages/curl
        command: ["/bin/sleep", "3650d"]
        imagePullPolicy: IfNotPresent
        volumeMounts:
        - mountPath: /etc/sleep/tls
          name: secret-volume
      volumes:
      - name: secret-volume
        secret:
          secretName: sleep-secret
          optional: true
---
```

ii. 执行以下命令，创建sleep服务。

```
kubectl apply -f sleep.yaml -n demo-frontend
```

3. 验证测试服务是否注入Sidecar成功。

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**工作负载 > 容器组**。

v. 在**容器组**页面单击sleep服务的容器组名称。

在**容器**页签下可以看到istio-proxy，说明sleep服务注入Sidecar成功。

步骤三：创建出口网关

服务网格内的服务访问网格外的网站时，可以通过出口网关管控流量。配置出口网关的授权策略后，还可以设置条件来控制是否允许访问外部网站。

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单击**ASM网关**，然后在右侧页面单击**创建**。
5. 输入出口网关的名称，选择**部署集群**，设置**网关类型**为**南北向-出口**，然后单击**创建**。本文设置出口网关的名称为egressgateway。

步骤四：创建对等身份认证

创建对等身份认证，便于授权策略使用TLS对服务进行授权。

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**零信任安全 > 对等身份认证**。
5. 在**对等身份认证**页面单击**创建双向mTLS模式**。
6. 设置**命名空间**为demo-frontend，输入名称，设置**mTLS模式**为STRICT模式，然后单击**创建**。

步骤五：设置外部服务的访问策略

默认对外部服务的访问策略为允许访问全部外部服务。为了实现对特定的外部网站进行访问控制，您需要设置外部服务访问策略为**REGISTRY_ONLY**，未注册为ServiceEntry的外部服务将无法被服务网格中的服务访问。

1. 设置外部服务的访问策略。

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. 在网格详情页面左侧导航栏选择**Sidecar管理（数据面） > Sidecar代理配置**。

v. 在**全局**页签下单击**外部服务访问策略**，配置**对外部服务的访问策略**OutboundTrafficPolicy为**REGISTEY_ONLY**，然后单击**更新设置**。

2. 将外部服务注册到ServiceEntry中。

i. 在网格详情页面左侧导航栏选择**集群与工作负载条目 > 服务条目**，然后在右侧页面单击**使用YAML创建**。

ii. 设置**命名空间**为**istio-system**，将以下内容复制到文本框中，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: aliyuncom-ext
  namespace: istio-system
spec:
  hosts:
    - www.aliyun.com
  location: MESH_EXTERNAL
  ports:
    - name: http
      number: 80
      protocol: HTTP
    - name: tls
      number: 443
      protocol: TLS
  resolution: DNS
```

步骤六：创建流量策略

创建网关规则、目标规则和虚拟服务，使demo-frontend命名空间下的流量路由到出口网关，再由出口网关路由到外部网站。

1. 创建网关规则。
   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. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。
   v. 设置**命名空间**为istio-system，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: istio-egressgateway
  namespace: istio-system
spec:
  selector:
    istio: egressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTPS
      tls:
        mode: ISTIO_MUTUAL
      hosts:
        - '*'
```

   mode：设置为*ISTIO_MUTUAL*，表示启用双向TLS服务认证，即网格内服务访问外部网站需要TLS服务认证。

2. 创建目标规则。

i. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。

ii. 设置**命名空间**为demo-frontend，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: target-egress-gateway
  namespace: demo-frontend
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: target-egress-gateway-mTLS
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN
        tls:
          mode: ISTIO_MUTUAL
```

mode：设置为*ISTIO_MUTUAL*，表示启用双向TLS服务认证，即外部网站访问网格内服务需要TLS服务认证。

3. 创建虚拟服务。

i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。

ii. 设置**命名空间**为demo-frontend，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: example-com-through-egress-gateway
  namespace: demo-frontend
spec:
  exportTo:
    - istio-system
    - demo-frontend
  gateways:
    - mesh
    - istio-system/istio-egressgateway
  hosts:
    - www.aliyun.com
  http:
    - match:
        - gateways:
            - mesh
          port: 80
      route:
        - destination:
            host: istio-egressgateway.istio-system.svc.cluster.local
            port:
              number: 80
            subset: target-egress-gateway-mTLS
          weight: 100
    - match:
        - gateways:
            - istio-system/istio-egressgateway
          port: 80
      route:
        - destination:
            host: www.aliyun.com
            port:
              number: 80
          weight: 100
```

http设置了2条匹配规则，第一条设置gateways为*mesh*，表示作用范围为demo-frontend命名空间下的Sidecar代理，将demo-frontend命名空间下的流量路由到出口网关。第二条设置为gateways为*istio-system/istio-egressgateway*，将出口网关的流量路由到注册的外部服务。

步骤七：创建授权策略

在demo-frontend命名空间下创建授权策略，作用在出口网关egressgateway上，拒绝来自demo-frontend命名空间的访问。

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**零信任安全 > 授权策略**，然后在右侧页面单击**创建**。
5. 设置授权策略参数，然后单击**创建**。

参数	说明
命名空间	设置**命名空间**为demo-frontend。
名称	输入授权策略的名称。
策略	设置**策略**为**RULES**。
动作	设置**动作**为**DENY**。
工作负载标签选择	打开**工作负载标签选择**开关，单击**新增匹配标签**，设置**名称**为**istio**，**值**为egressgateway。
请求来源	打开**请求来源**开关，单击**增加请求来源到列表中**，单击**增加请求来源**，设置**请求来源域**为**namespaces**，**值**为demo-frontend。

步骤八：验证限制demo-frontend命名空间中的服务访问外部网站是否成功

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**工作负载 > 容器组**。
5. 在**容器组**页面单击sleep容器右侧**操作**列下的**终端**，单击**容器：sleep**。
6. 执行以下命令，访问外部网站aliyun.com。

   ```
   curl -I http://www.aliyun.com
   ```

   预期输出：

   ```
   HTTP/1.1 403 Foribidden
   ......
   RBAC: ccess denied
   ```

   提示 `RBAC access denied 403` 错误，说明demo-frontend命名空间下的服务访问外部网站aliyun.com失败。限制demo-frontend命名空间中的服务访问外部网站成功。

7.3. 对命名空间下服务访问数据库进行授权控制

为了保障数据库的安全性，需要对访问数据库的服务进行限制，例如只有某些生产环境的命名空间才能被允许访问生产数据库，开发环境不能访问生产数据库。您可以使用ASM零安全体系，动态配置授权策略，实现对命名空间下服务访问外部数据库进行授权控制，从而降低风险。本文以demo-server命名空间为例，介绍如何使用授权策略实现对外部特定的RDS数据库访问的控制。

前提条件

添加集群到ASM实例。具体操作，请参见添加集群到ASM实例。

步骤一：注入Sidecar代理

为命名空间注入Sidecar代理，便于对该命名空间下的服务进行授权管理。

1. 新建demo-server命名空间。

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. 在网格详情页面左侧导航栏选择**网格实例 > 全局命名空间**，然后在右侧页面单击**新建**。
v. 在**新建命名空间**面板设置名称为demo-server，然后单击**确定**。

2. 为demo-server命名空间注入Sidecar代理。
   i. 在**全局命名空间**页面单击demo-server命名空间右侧**自动注入**列下的**启用Sidecar自动注入**。
   ii. 在**确认**对话框单击**确定**。

步骤二：创建数据库客户端

在demo-server命名空间下创建发起数据库连接请求的客户端。

1. 在本地命令行工具中对数据库连接密码进行Base64编码。

```
echo  <数据库连接密码> | base64
```

2. 通过kubectl工具连接集群。
3. 在demo-server命名空间下创建MySQL客户端。
   i. 使用以下内容，创建*k8s-mysql.yaml*。

```
apiVersion: v1
data:
  password: {yourPasswordBase64}  #数据库连接密码的Base64编码。
kind: Secret
metadata:
  name: mysql-pass
type: Opaque
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    name: lbl-k8s-mysql
  name: k8s-mysql
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      name: lbl-k8s-mysql
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        name: lbl-k8s-mysql
    spec:
      containers:
```

```
      - env:
          - name: MYSQL_ROOT_PASSWORD
            valueFrom:
              secretKeyRef:
                key: password
                name: mysql-pass
        image: 'mysql:latest'
        imagePullPolicy: Always
        name: mysql
        ports:
          - containerPort: 3306
            name: mysql
            protocol: TCP
        resources:
          limits:
            cpu: 500m
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        volumeMounts:
          - mountPath: /var/lib/mysql
            name: k8s-mysql-storage
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
    volumes:
      - emptyDir: {}
        name: k8s-mysql-storage
```

ii. 执行以下命令，创建MySQL客户端。

```
kubectl apply -f k8s-mysql.yaml -n demo-server
```

4. 验证MySQL客户端是否注入Sidecar成功。
   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**工作负载 > 容器组**。
   v. 在**容器组**页面单击MySQL客户端的容器组名称。

   在**容器**页签下可以看到istio-proxy，说明创建MySQL客户端注入Sidecar成功。

步骤三：创建出口网关

服务网格内的服务访问网格外的网站时，需要通过出口网关管控流量。配置出口网关的授权策略后，还可以设置条件来控制是否允许访问数据库。

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单击**ASM网关**，然后在右侧页面单击**创建**。
5. 输入出口网关的名称，选择**部署集群**，设置**网关类型**为**南北向-出口**，单击**端口映射**右侧的**添加端口**，设置**协议**为**TCP**，**服务端口**为13306，然后单击**创建**。本文设置出口网关的名称为egressgateway。

步骤四：创建对等身份认证

创建对等身份认证，便于授权策略使用TLS对服务进行授权。

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**零信任安全 > 对等身份认证**。
5. 在**对等身份认证**页面单击**创建双向mTLS模式**。
6. 设置**命名空间**为demo-server，输入名称，设置**mTLS模式**为STRICT模式，然后单击**创建**。

步骤五：设置外部服务的访问策略

默认对外部服务的访问策略为允许访问全部外部服务。为了实现对特定的外部网站进行访问控制，您需要设置外部服务访问策略为**REGISTRY_ONLY**，未注册为ServiceEntry的外部服务将无法被服务网格中的服务访问。

1. 设置外部服务的访问策略。
   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. 在网格详情页面左侧导航栏选择**Sidecar管理（数据面） > Sidecar代理配置**。
   v. 在**全局**页签下单击**外部服务访问策略**，配置**对外部服务的访问策略OutboundTrafficPolicy**为**REGISTEY_ONLY**，然后单击**更新设置**。
2. 将外部服务注册到ServiceEntry中。
   i. 在网格详情页面左侧导航栏选择**集群与工作负载条目 > 服务条目**，然后在右侧页面单击**使用YAML创建**。
   ii. 设置**命名空间**为**istio-system**，将以下内容复制到文本框中，单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: demo-server-rds
  namespace: demo-server
spec:
  endpoints:
    - address: rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com   #数据库地址
      ports:
        tcp: 3306
  hosts:
    - rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
  location: MESH_EXTERNAL
  ports:
    - name: tcp
      number: 3306  #数据库端口
      protocol: TCP  #数据库协议
  resolution: DNS
```

步骤六：创建流量策略

创建网关规则、目标规则和虚拟服务，使demo-server命名空间下的流量路由到出口网关的13306端口，再由出口网关路由到数据库的3306端口。

1. 创建网关规则。
   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. 在网格详情页面左侧导航栏选择**流量管理 > 网关规则**，然后在右侧页面单击**使用YAML创建**。
   v. 设置**命名空间**为istio-system，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: istio-egressgateway
  namespace: istio-system
spec:
  selector:
    istio: egressgateway
  servers:
    - hosts:
        - '*'
      port:
        name: http-0
        number: 13306
        protocol: TLS
      tls:
        mode: ISTIO_MUTUAL
```

   mode：设置为*ISTIO_MUTUAL*，表示启用双向TLS服务认证，网格内服务访问外部网站需要TLS服务认证。

2. 创建目标规则。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 目标规则**，然后在右侧页面单击**使用YAML创建**。
   ii. 设置**命名空间**为demo-server，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: demo-server-egress-gateway
  namespace: demo-server
spec:
  host: istio-egressgateway.istio-system.svc.cluster.local
  subsets:
    - name: mysql-gateway-mTLS
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN
        portLevelSettings:
          - port:
              number: 13306 #网关映射端口
            tls:
              mode: ISTIO_MUTUAL
              sni: rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com  #数据库host地址
```

   mode：设置为*ISTIO_MUTUAL*，表示启用双向TLS服务认证，外部网站访问网格内服务需要TLS服务认证。

3. 创建虚拟服务。
   i. 在网格详情页面左侧导航栏选择**流量管理 > 虚拟服务**，然后在右侧页面单击**使用YAML创建**。
   ii. 设置**命名空间**为demo-server，将以下内容复制到文本框，然后单击**创建**。

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: demo-server-through-egress-gateway
  namespace: demo-server
spec:
  exportTo:
    - istio-system
    - demo-server
  gateways:
    - mesh
    - istio-system/istio-egressgateway
  hosts:
    - rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
  tcp:
    - match:
        - gateways:
            - mesh
          port: 3306
      route:
        - destination:
            host: istio-egressgateway.istio-system.svc.cluster.local
            port:
              number: 13306
            subset: mysql-gateway-mTLS
          weight: 100
    - match:
        - gateways:
            - istio-system/istio-egressgateway
          port: 13306
      route:
        - destination:
            host: rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
            port:
              number: 3306
          weight: 100
```

http设置了2条匹配规则，第一条设置gateways为*mesh*，表示作用范围为demo-server命名空间下的Sidecar代理，将demo-server命名空间下的流量路由到出口网关的13306端口。第二条设置为gateways为*istio-system/istio-egressgateway*，将出口网关的流量路由到注册的数据库3306端口。

步骤七：验证使用授权策略实现对外部数据库访问的控制

通过修改授权策略的动作，您可以禁止或允许demo-server命名空间下的服务访问外部数据库，实现对外部数据库访问的控制。

1. 创建授权策略，禁止来自demo-server命名空间的访问请求。
   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. 在网格详情页面左侧导航栏选择**零信任安全 > 授权策略**，然后在右侧页面单击**创建**。

v. 设置授权策略参数，然后单击**创建**。

参数	说明
命名空间	设置**命名空间**为demo-server。
名称	输入授权策略的名称。
策略	设置**策略**为**RULES**。
动作	设置**动作**为**DENY**。
工作负载标签选择	打开**工作负载标签选择**开关，单击**新增匹配标签**，设置**名称**为**istio**，**值**为egressgateway。
请求来源	打开**请求来源**开关，单击**增加请求来源到列表中**，单击**增加请求来源**，设置**请求来源域**为**namespaces**，**值**为demo-server。

2. 访问外部数据库。

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**工作负载 > 容器组**。

v. 在**容器组**页面单击k8s-mysql容器右侧**操作**列下的**终端**，单击**容器：mysql**。

vi. 在容器组终端中执行以下命令，访问外部数据库。

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --host rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
```

返回结果中提示 `ERROR 2013` 错误，说明访问数据库失败。

3. 修改授权策略动作为ALLOW，允许来自demo-server命名空间的访问请求。

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. 在网格详情页面左侧导航栏选择**零信任安全 > 授权策略**。

v. 在**授权策略**页面单击目标策略右侧**操作**列下的**YAML**。

vi. 在**编辑**面板修改action参数值为*ALLOW*，然后单击**确定**。

4. 在容器组终端中执行以下命令，访问外部数据库。

```
mysql --user=root --password=$MYSQL_ROOT_PASSWORD --host rm-xxxxxxx.mysql.xxxx.rds.aliyuncs.com
```

返回结果中显示 `Welcome to the MySQL monitor` ，说明访问数据库成功。
根据以上操作结果，可以看到使用授权策略控制对外部数据库的访问成功。