8.1.1 Istio 治理三层微服务

在上一节中，我们只是将应用迁移到 Istio 中，下面我们将着重介绍 Istio 如何实现对迁移成功的三层微服务进行治理。

1. 治理场景重要说明

针对场景展现有两点需要说明。

1. 在场景的展现中，我们有的步骤修改了三层微微服务的源码（如在超时设置），在功能展示结束后，如果不需要这个功能，需要将源码改回，并重新编译。由于篇幅有限，场景展现中不再赘述。
2. 每个功能展现场景初始是没有 Virtual Service 和 Destination Rule 的。如果读者要进行体验，需要在每个功能配置之前，手工或者使用脚本清除上一个场景的配置。

   脚本链接：https://github.com/ocp-msa-devops/istio-tutorial/blob/master/scripts/clean.sh。

   执行方式如下，也就是清除 tutorial 项目中的 Destination Rule 和 Virtual Service。

   ```
   # sh clean.sh tutorial
   ```

   由于每个场景清除配置方式相同，因此下面每个配置章节中将不再赘述。

2. 三层微服务配置路由管理

在微服务中，很重要的一部分就是路由管理，主要指通过策略配置实现对微服务之间访问的管理，主要的场景有：

- 灰度/蓝绿发布：在发布新版本应用时，通过路由管理实现现发布一部分用于测试，没有问题再逐步迁移流量。
- 外部访问控制：管理外部访问 Istio 微服务的路由，并提供一些安全、权限上的控制。

- 访问外部服务控制：控制所有出站流量。
- 服务推广：通过逐步提升应用的稳定性和功能，替换线上应用。

（1） 三层微服务的对外访问和访问安全

在前面章节，我们已经部署了三层微服务，并且为对外的 customer 服务创建了路由，我们可以直接通过 curl 对 customer 服务发起多次调用：

curl customer-tutorial.apps.example.com

customer => preference => recommendation v1 from 'f967df69-m9k8f': 1

curl customer-tutorial.apps.example.com

customer => preference => recommendation v2 from '58fcd486f6-j5qfj ': 2

由于 recommendation 服务有两个版本 v1 和 v2，可以看到对 recommendation 服务的访问是默认的 round-robin 负载策略。

这个时候，如果我们通过 Kiali 观测服务调用，会出现显示的不准确的现象，如下图 8-51 所示：

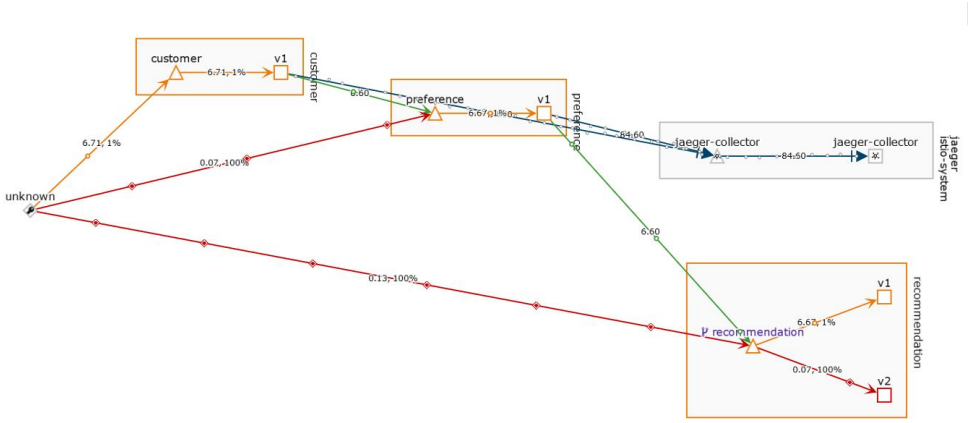

图 8-51 Kiali 展示

可以发现，客户端并没有直接去访问 preference 和 customer 服务，但是图中却出现红色的调用线。多余的红线出现的根源就是：prometheus 会访问各个个微服务。我们需要在 istio 的 rules.config.istio.io 中屏蔽显示。查看项目中的 rules.config.istio.io，如下图 8-52 所示：

```
[root@master ~]# oc get rules.config.istio.io
NAME                          AGE
denyreviewsv3                 79d
kubeattrgenrulerule           92d
promhttp                      92d
promtcp                       92d
promtcpconnectionclosed       92d
promtcpconnectionopen         92d
tcpkubeattrgenrulerule        92d
```

图 8-52 查看项目中的 rules.config.istio.io

因为显示与 http 相关，我们修改 promhttp。在配置的尾部，增加 && (match((request.useragent | "-"), "Prometheus*") == false)内容即可。从而实现通过规则来过滤掉此流量 Prometheus 收集 request.useragent 的流量，如下图 8-53 所示：

[root@master ~]# oc edit rules.config.istio.io promhttp -n istio-system

```
spec:
  actions:
  - handler: prometheus
    instances:
    - requestcount.metric
    - requestduration.metric
    - requestsize.metric
    - responsesize.metric
  match: (context.protocol == "http" || context.protocol == "grpc") && (match((request.useragent
    | "-"), "kube-probe*") == false) &&   (match((request.useragent | "-"), "Prometheus*")  == false)
```

图 8-53 修改项目中的 rules.config.istio.io

配置完毕以后，再次收集监控数据，没有红线出现了，如下图 8-54 所示：

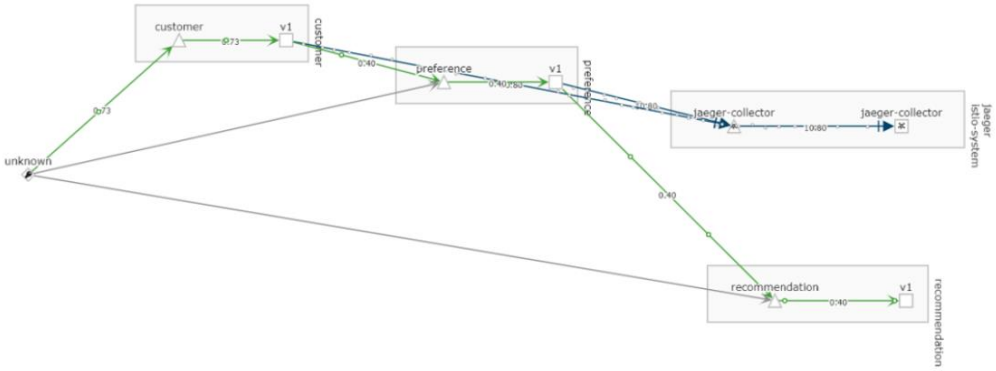

图 8-54 Kiali 正常显示

下面我们先配置 Gateway 和 Virtual Service，内容如下：

```
apiVersion: networking.istio.io/v1alpha3

kind: Gateway

metadata:

  name: customer-gateway

spec:

  selector:
```

```
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: customer
  namespace: tutorial
spec:
  hosts:
  - "*"
  gateways:
  - customer-gateway
  http:
  - match:
    - uri:
        exact: /
    route:
    - destination:
        host: customer
        port:
          number: 8080
```

在上述配置中，包含以下信息：

- Gateway 开放的端口是 80，设定的 hosts 是*。
- Virtual Service 中配置的 hosts 是*。
- Virtual Service 配置的 URI 是/。
- Virtual Service 配置的目标服务是 customer，端口是 8080。

应用配置

oc create -f gateway-customer.yml

现在 customer 服务的 Gateway 已经配置成功了，接下来我们启用 mTLS。首先创建 policy 以在项目 tutorial 中启用 mTLS，policy 文件内容如下：

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "default"
spec:
  peers:
  - mtls: {}
```

接着我们需要声明 tutorial 项目中的微服务之间通信需要使用 mTLS，Destination Rules 内容如下：

```
apiVersion: "networking.istio.io/v1alpha3"
kind: "DestinationRule"
metadata:
  name: "default"
spec:
  host: "*.tutorial.svc.cluster.local"
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
```

可以看到在 hosts 部分直接定义为*.tutorial.svc.cluster.local，表示 tutorial 项目下的所有服务。

应用配置：

oc create -f istiofiles/authentication-enable-tls.yml

policy.authentication.istio.io/default created

oc create -f istiofiles/destination-rule-tls.yml

destinationrule.networking.istio.io/default created

需要注意的是，当 Istio 启动了 mTLS 以后，Destination Rule 也需要调整为加密的模式，否则访问的是会出现报错。

例如，如果在启动 mTLS 之前我们使用的 Destination Rules 的内容是：

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
spec:
  host: recommendation
  subsets:
  - labels:
      version: virtualized
    name: version-virtualized
  - labels:
      version: v1
name: version-v1
```

那么在启动 mTLS 之后，就需要将配置文件修改为：

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
spec:
  host: recommendation
  trafficPolicy:
    tls:
```

```
      mode: ISTIO_MUTUAL
  subsets:
  - labels:
      version: virtualized
    name: version-virtualized
  - labels:
      version: v1
    name: version-v1
```

然后执行 oc replace -f destination-rule-recommendation-v1.yml。这样客户端的访问才是正常的。

到这里我们就配置好了应用的对外访问，并开启了服务间调用的双向 TLS 认证。接下来我们就可以经过 ingressgateway 访问我们的三层架构的微服务了。

在创建 customer 服务的 Gateway 和 Virtual Service 的时候，我们定义的 hosts 都是*，所以支持在 9.2.1 中介绍的三种方式访问，我们就采用第一种方式进行访问，通过浏览器可以访问路由，得到正确的返回，如下图 8-55 所示：

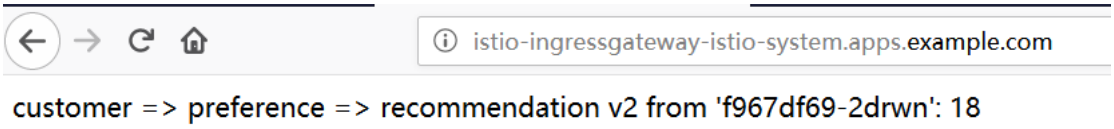

图 8-55 浏览器访问应用

此时，我们再观测 Kiali，收集到的信息是正常的。我们也可以看到流量是加密的，如下图 8-56 所示：

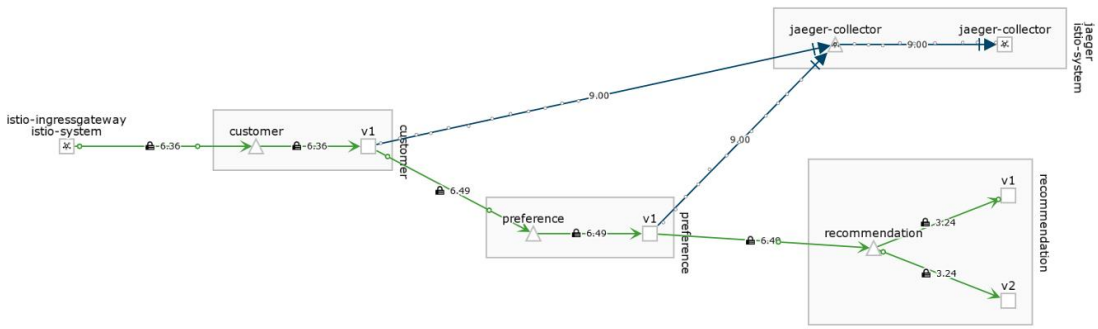

图 8-56 Kiali 展示

（2） 三层微服务基于目标端的灰度/蓝绿发布

在 OpenShift 中，我们可以通过将一个应用的 Route（FQDN）与两个或者多个 Service 相关联，从而实现 A/B 测试、蓝绿发布等。在 Istio 中，实现方法更为灵活，通过配置 Virtual Service 就可以实现。

接下来，我们对三层微服务实现灰度/蓝绿发布。我们再次对三层微服务发起多次访问请求，我们可以看到对 recommendation 依然是 round-robin 方式调用的。

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ;done

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 4309

customer => preference => recommendation v2 from 'f967df69-2drwn': 346

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 4310

customer => preference => recommendation v2 from 'f967df69-2drwn': 347

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 4311

接下来，通过配置 Virtual Service 和 Destination Rules，让所有用户请求都转到 recommendation v1，实现蓝绿发布。

为 recommendation 服务配置 Virtual Service，内容如下：

cat virtual-service-recommendation-v1.yml

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
```

```
      weight: 100
```

可以看到在 Virtual Service 中定义对 recommendation 微服务的访问全部路由到 v1 版本。

应用配置:

oc create -f virtual-service-recommendation-v1.yml

virtualservice.networking.istio.io/recommendation created

配置 Destination Rules，内容如下：

cat destination-rule-recommendation-v1-v2.yml

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
spec:
  host: recommendation
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
  subsets:
  - labels:
      version: v1
    name: version-v1
  - labels:
      version: v2
    name: version-v2
```

oc create -f destination-rule-recommendation-v1.yml

destinationrule.networking.istio.io/recommendation created

对三层微服务发起 curl 请求，我们看到所有请求都访问 recommendation v1，蓝绿发布成功。

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep

1;done

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 12433

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 12434

在 Kiali 中查看调用流量图，如下图 8-57 所示：

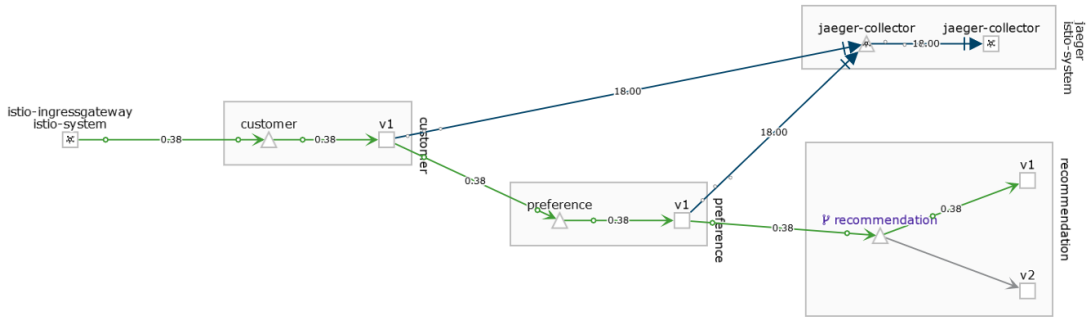

图 8-57 Kiali 展示

接下来，我们对 recommendation 进行分流：90%的请求到 v1，10%的请求到 v2，模拟灰度发布。

调整 recommendation 的 Virtual Service，内容如下：

cat virtual-service-recommendation-v1_and_v2.yml

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
      weight: 90
```

```
    - destination:
        host: recommendation
        subset: version-v2
      weight: 10
```

用新的配置替换原有 Virtual Services 配置：

oc replace -f virtual-service-recommendation-v1_and_v2.yml

virtualservice.networking.istio.io/recommendation replaced

使用 curl 发起多次调用，并通过 Kiali 进行观测。可以在看到对 recommendation 的请求，v1 版本的访问量为每秒 3.82，v2 版本为 0.47，v1 版本的请求量为 v2 版本接近 9 倍，这与我们的预期是一致的，如下图 8-58 所示：

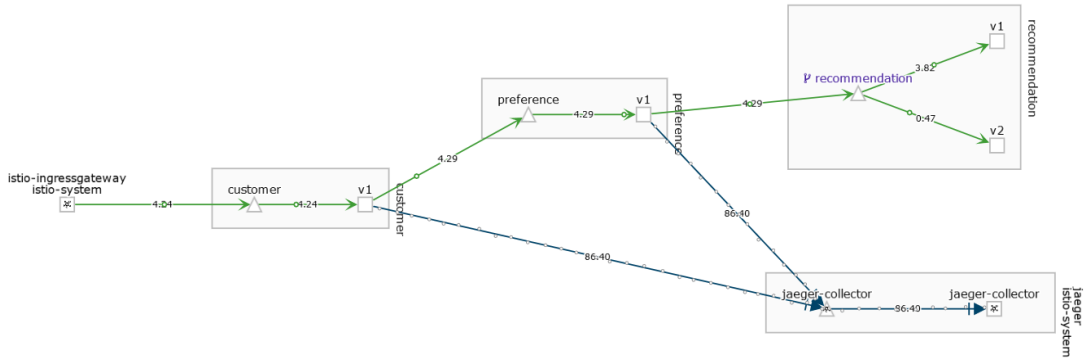

图 8-58 Kiali 展示

（3） 三层微服务基于源端 User-Agent 的蓝绿发布

无论是前面章节介绍的 Istio 基于 Virtual Services 实现的灰度，还是通过 OpenShift Router 实现的灰度，都是基于目标端的版本选择。但是在 Istio 中，我们还可以配置基于源端 User-Agent 的智能路由。User-Agent header 包含了一个特征字符串，用来让网络协议的对端来识别发起请求的用户代理软件的应用类型、操作系统、软件开发商以及版本号。

在本小节中，我们基于源端的浏览器类型，设置智能路由，实现基于源端的蓝绿发布。Virtual Services 配置内容如下：

cat virtual-service-safari-recommendation-v2.yml

apiVersion: networking.istio.io/v1alpha3

```
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - match:
    - headers:
        baggage-user-agent:
          regex: .*Safari.*
    route:
    - destination:
        host: recommendation
        subset: version-v2
  - route:
    - destination:
        host: recommendation
        subset: version-v1
```

可以看到配置中定义了当发起请求的客户端为 Safari 浏览器时，调用 recommendation v2 版本；如果客户端不是 Safari 浏览器，则调用 recommendation v1 版本。

应用 Virtual Service 配置

oc apply -f virtual-service-safari-recommendation-v2.yml

virtualservice.networking.istio.io/recommendation replaced

使用 curl 对微服务发起请求，通过-A 参数指定 User-Agent 特征字符串：Safari，模拟使用 Safari 浏览器访问。

while true;do curl -A Safari http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep .1 ;done

customer => preference => recommendation v2 from 'f967df69-2drwn': 2836

customer => preference => recommendation v2 from 'f967df69-2drwn': 2837

customer => preference => recommendation v2 from 'f967df69-2drwn': 2838

我们看到，访问的 recommendation 的版本为 v2。访问结果的 Kiali 展示如下图 8-58 所示：

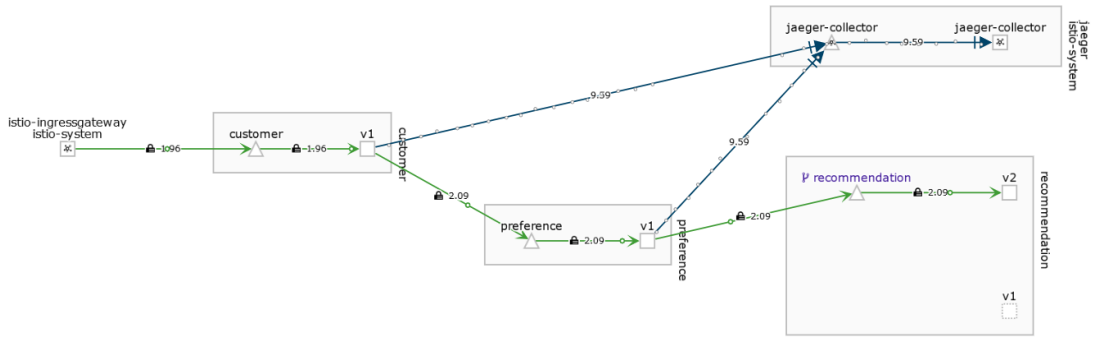

图 8-58 Kiali 展示

使用 curl 对微服务发起请求，通过-A 参数指定 User-Agent 字符串：Firefox，模拟使用火狐浏览器访问。

#while true;do curl -A Firefox http://istio-ingressgateway-istio-system.apps.example.com/;done

root@master ~]# while true;do curl -A Firefox http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep .1 ;done

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 14986

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 14987

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 14988

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 14989

我们看到，访问的 recommendation 的版本为 v1。基于客户端浏览器类型来实现应用蓝绿发布成功，如下图 8-59 所示：

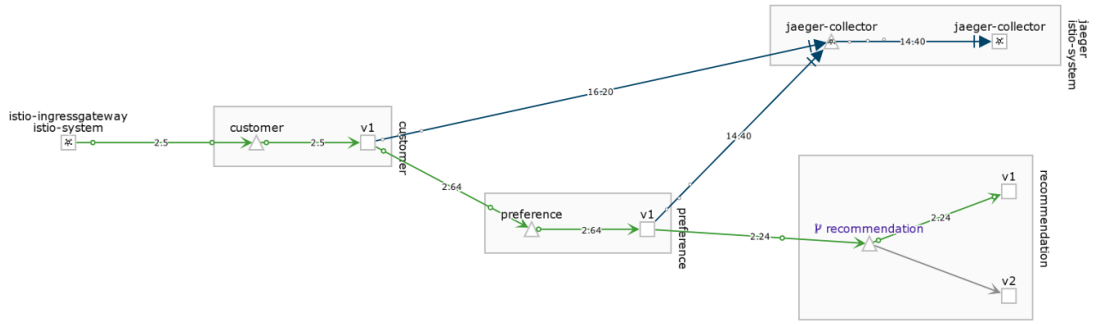

图 8-59 Kiali 展示

（4） 三层微服务的灰度上线

在前面 Bookinfo 微服务中，我们展现了通过流量镜像实现灰度上线。接下来，我们通过三层微服务展示流量镜像。

在三层微服务中，我们先将路由规则将 100％的流量发送到 recommendation v1 版本，指定流量要镜像到 recommendation v2 版本，模拟 recommendation v2 版本灰度上线。

配置 Virtual Servcie 内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
        subset: version-v1
    mirror:
      host: recommendation
```

```
        subset: version-v2
```

可看到配置比较简单，仅需要通过 mirror 字段配置流量镜像的目标服务。

通过 curl 发起对三层微服务的请求。在应用配置前，请求访问的返回是：

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep .01 ;done

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22170

customer => preference => recommendation v2 from 'f967df69-6mnqp': 631

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22171

customer => preference => recommendation v2 from 'f967df69-6mnqp': 632

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22172

应用流量镜像的配置

oc replace -f virtual-service-recommendation-v1-mirror-v2.yml -n tutorial

virtualservice.networking.istio.io/recommendation replaced

再度发起请求，可看到返回值只有 recommendation v1 版本。

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep .01 ;done

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22174

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22175

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22176

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22177

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22178

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22179

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22180

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 22181

我们查看 recommendation-v1 pod 的日志，有被访问的返回信息：

oc logs -f $(oc get pods|grep recommendation-v1|awk '{ print $1 }') -c recommendation

INFO: recommendation request from 58fcd486f6-j5qfj: 995

May 01, 2019 10:29:07 AM com.redhat.developer.demos.recommendation.RecommendationVerticle

INFO: recommendation request from 58fcd486f6-j5qfj: 996

May 01, 2019 10:29:08 AM

com.redhat.developer.demos.recommendation.RecommendationVerticle

INFO: recommendation request from 58fcd486f6-j5qfj: 997

Kiali 流量展示如下图 8-60 所示：

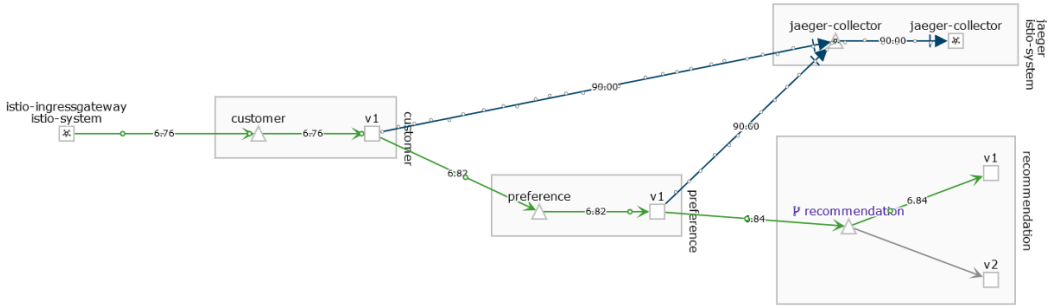

图 9-60 Kiali 展示

我们查看 recommendation-v2 pod 的日志，有被访问的返回信息，并且被访问的 Pod 的 image id 为：f967df69-m9k8f

oc logs -f $(oc get pods|grep recommendation-v2|awk '{ print $1 }') -c recommendation

INFO: recommendation request from f967df69-m9k8f: 230

May 01, 2019 9:22:42 AM

com.redhat.developer.demos.recommendation.RecommendationVerticle

INFO: recommendation request from f967df69-m9k8f: 231

May 01, 2019 9:22:43 AM

com.redhat.developer.demos.recommendation.RecommendationVerticle

INFO: recommendation request from f967df69-m9k8f: 232

May 01, 2019 9:22:44 AM

com.redhat.developer.demos.recommendation.RecommendationVerticle

INFO: recommendation request from f967df69-m9k8f: 233

也就是说，配置流量镜像后，在 Kiali 的流量展示中不会显示对 recommendation v2 版本的访问，但是 recommendation v1 的流量确实被镜像到了 recommendation v2 版本。

（5） 三层微服务访问外部服务

在前面 Istio 路由基本概念中，我们介绍了 Istio 中的微服务访问 Istio 之外的服务需要通过 ServcieEntry 注册外部服务到 Istio 中。但是在实际微服务中，通常会调用一些外部服务的接口，下面我们就用三层微服务演示如何实现访问外部服务。

在前文，我们部署了 recommendation 的 v1 和 v2 版本。接下来，我们部署 recommendationv3 版本。修改 RecommendationVerticle.java 源码，一共修改两处：

第一处更改是将响应字符串更新为：

```
  private static final String RESPONSE_STRING_FORMAT = "recommendation v3 from
'%s': %d\n";
```

第二处是反注释 getNow 的方法，将 getRecommendations 方法注释。

```
        //router.get("/").handler(this::getRecommendations);
        router.get("/").handler(this::getNow);
```

调整以后，当访问 Recommendations 微服务的时候，将不再调用 getRecommendations 方法，返回 RESPONSE_STRING_FORMAT, HOSTNAME, count。而是调用 getNow，访问外部的 webapi。代码内容如下：

```
  private void getNow(RoutingContext ctx) {
        count++;
        final WebClient client = WebClient.create(vertx);
        client.get(80, HTTP_NOW, "/api/json/cet/now")
        .timeout(5000)
        .as(BodyCodec.jsonObject())
           .send(ar -> {
              if (ar.succeeded()) {
                 HttpResponse<JsonObject> response = ar.result();
                 JsonObject body = response.body();
                 String now = body.getString("currentDateTime");
```

```
                    ctx.response().end(now + " " +
String.format(RESPONSE_STRING_FORMAT, HOSTNAME, count));
```

代码逻辑为：如果 Recommendations v3 调用外部 web api 成功，那么对 Recommendations v3 的调用返回显示格式将是 RESPONSE_STRING_FORMAT, HOSTNAME, count，其中

- RESPONSE_STRING_FORMAT 是 currentDateTime，也就是当前时间。
- HOSTNAME 是主机名，即 Recommendations v3。
- count 是被调用的次数。

接下来，重新编译源码、生成 Docker Image、部署到 OpenShift 中：

cd /root/istio-tutorial/recommendation/java/vertx/

mvn clean package

docker build -t example/recommendation:v3 .

oc apply -f <(~/istio-1.1.2/bin/istioctl kube-inject -f ../../kubernetes/Deployment-v3.yml) -n tutorial

deployment.extensions/recommendation-v3 created

过一会，recommendation v3 容器部署成功。

oc get pods

NAME READY STATUS RESTARTS AGE

customer-775cf66774-qsvv9 2/2 Running 5 16h

preference-v1-667895c986-ljqpg 2/2 Running 3 15h

recommendation-v1-58fcd486f6-m42lh 2/2 Running 0 4h

recommendation-v2-f967df69-6mnqp 2/2 Running 0 3h

recommendation-v3-97ff85fdb-lj9l7 2/2 Running 0 1m

下面我们应用新的配置，将所有流量重定向到 reccomendation：v3。此前应用的 Destination Rule 只包含 recommendation v1 和 recommendation v2 的描述，没有到 v3 的描述。我们需要创建包含 v3 的 Destination Rule 和 Virtual Servcie。

包含 recommendation v3 的 Destination Rule 内容如下：

```
   apiVersion: networking.istio.io/v1alpha3
```

```
kind: DestinationRule
metadata:
  name: recommendation
spec:
  host: recommendation
  trafficPolicy:
    tls:
      mode: ISTIO_MUTUAL
  subsets:
  - labels:
      version: v1
    name: version-v1
  - labels:
      version: v2
    name: version-v2
  - labels:
      version: v3
    name: version-v3
```

应用上述 Destination Rule：

oc replace -f istiofiles/destination-rule-recommendation-v1-v2-v3.yml -n tutorial

destinationrule.networking.istio.io/recommendation created

定义 recommendation v3 的 Vritual Service，内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
```

```
    http:
    - route:
      - destination:
          host: recommendation
          subset: version-v3
        weight: 100
```

应用 Virtual Service 配置：

oc create -f istiofiles/virtual-service-recommendation-v3.yml

virtualservice.networking.istio.io/recommendation created

由于此时并没有配置 ServiceEntry 对象，因此访问返回 503 报错。

curl http://istio-ingressgateway-istio-system.apps.example.com/

<html><body><h1>504 Gateway Time-out</h1>

The server didn't respond in time.

</body></html>

下面我们创建 ServiceEntry，将 worldclockapi.com 注册到 Istio 中，ServcieEntry 内容如下：

```
  apiVersion: networking.istio.io/v1alpha3
  kind: ServiceEntry
  metadata:
    name: worldclockapi-egress-rule
  spec:
    hosts:
    - worldclockapi.com
    ports:
    - name: http-80
      number: 80
      protocol: http
```

应用配置：

oc create -f istiofiles/service-entry-egress-worldclockapi.yml

serviceentry.networking.istio.io/worldclockapi-egress-rule created

此时我们可以看到 ServiceEntry 配置已经生效。

oc get serviceentry

NAME HOSTS LOCATION RESOLUTION AGE

worldclockapi-egress-rule [worldclockapi.com] 1m

我们再次发起访问请求

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => 2019-05-07T14:38+02:00 recommendation v3 from '97ff85fdb-lj9l7': 38

可以看到，此次不再报错，说明 recommendation v3 调用外部 web api 成功。而且返回结果与源码定义的格式一致。

（6） 三层微服务的服务推广

服务推广（Service Promotion）也是微服务版本控制中的一个重要功能。在测试和运维中，通常旧版本的稳定性要高于新版本，但新版的功能要强于旧版本。有的最终客户喜欢稳定，有的最终客户喜欢尝鲜。

在 Istio 中，我们可以允许用户选择是否要尝试最新部署的应用程序版本（实验级别）或稳定版本的应用程序（稳定级别）

当前 Istio 中包含 recommendation 的三个版本：v1、v2 和 v3。

我们对微服务发起请求，对 recommendation 的版本访问是 round-robin 的。

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v3 from '97ff85fdb-wrwvk': 1

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 2

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v2 from 'f967df69-6mnqp': 1

在默认情况下，最终客户无法选择访问哪个版本。

接下来，我们分别为 recommendation 的三个版本定义三个级别：实验、测试和生产。

然后客户可以根据自己的需求访问不同的版本。

定义 Virtual Servcie 配置，内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - match:
    - headers:
        baggage-user-preference:
          prefix: "123"
    route:
    - destination:
        host: recommendation
        subset: version-v3
  - match:
    - headers:
        baggage-user-preference:
          prefix: "12"
    route:
    - destination:
        host: recommendation
        subset: version-v2
  - route:
    - destination:
        host: recommendation
```

```
      subset: version-v1
```

在上述配置中，我们为不同版本的 recommendation 设置不同的标头：

- Recommendation v3：实验版本，标头为 123
- Recommendation v2：测试版本，标头为 12
- Recommendation v1：生产版本，无标头

应用配置：

oc create -f istiofiles/virtual-service-promotion-v1-v2-v3.yml

virtualservice.networking.istio.io/recommendation created

配置成功以后，我们从客户端发起对微服务的请求：可以看到，通过在访问时设置不同的标头，可以选择访问不同版本的 recommendation 服务。

访问实验版：

curl -H "user-preference: 123" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v3 from '97ff85fdb-wrwvk': 2

访问测试版：

curl -H "user-preference: 12" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v2 from 'f967df69-6mnqp': 2

访问生产版：

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 3

接下来，我们进行服务推广。也就是说 recommendation v3 版本经过了一段时间，他的稳定性大幅提升，可以由实验级别提升到测试级别。也就是说测试级别和实验级别都是 v3，而 v2 被提升打到生产级别。v1 由于功能太少退役。

我们通过修改 Vritual Service 实现，将 recommendation v3 的 prefix 设置为 12，和 recommendation v2 相同，内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
```

```
  hosts:
  - recommendation
  http:
  - match:
    - headers:
        baggage-user-preference:
          prefix: "12"
    route:
    - destination:
        host: recommendation
        subset: version-v3
  - route:
    - destination:
        host: recommendation
        subset: version-v2
```

应用配置：

oc replace -f istiofiles/virtual-service-promoted-v3.yml -n tutorial

virtualservice.networking.istio.io/recommendation replaced

访问实验版：

curl -H "user-preference: 12" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v3 from '97ff85fdb-wrwvk': 3

访问测试版：

curl -H "user-preference: 12" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v3 from '97ff85fdb-wrwvk': 4

访问生产版：

curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v2 from 'f967df69-6mnqp': 3

我们看到，访问结果符合我们的预期。通过 istio 的路由管理，我们实现了服务推广。

在配置三层微服务的路由管理过程中，可以看到 Istio 的路由管理是多么的强大，而它的能力远不止于此，我们只是实现了很少的一部分。

3. 三层微服务配置限流熔断

（1） 限流的实现

接下来，我们配置三层微服务的限流，有两个配置文件：rate_limit_rule.yml 和 recommendation_rate_limit_handler.yml。

在 rate_limit_rule.yml 配置文件中，定义了如下配置：quota rule（mixer 端）、Quota 实例（mixer 端）、QuotaSpec（客户端）、QuotaSpecBinding（客户端）。内容如下：

```
apiVersion: "config.istio.io/v1alpha2"
kind: quota
metadata:
  name: requestcount
spec:
  dimensions:
    source: source.labels["app"] | source.service | "unknown"
    sourceVersion: source.labels["version"] | "unknown"
    destination: destination.labels["app"] | destination.service |
"unknown"
    destinationVersion: destination.labels["version"] | "unknown"
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: quota
  namespace: istio-system
spec:
  actions:
```

```
  - handler: handler.memquota
    instances:
    - requestcount.quota
---
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpec
metadata:
  creationTimestamp: null
  name: request-count
  namespace: istio-system
spec:
  rules:
  - quotas:
    - charge: 1
      quota: RequestCount
---
apiVersion: config.istio.io/v1alpha2
kind: QuotaSpecBinding
metadata:
  creationTimestamp: null
  name: request-count
  namespace: istio-system
spec:
  quotaSpecs:
  - name: request-count
    namespace: istio-system
  services:
  - name: customer
    namespace: tutorial
  - name: preference
```

```
    namespace: tutorial
  - name: recommendation
    namespace: tutorial
```

可以看到在上述配置中定义了如下内容：

- Quota 中定义了名为 requestcount 的配额实例，实例中定义了 source、sourceversion、destination、destinationversion。
- QuotaSpec 中定义了配额实例的名称为 requestcount，每次请求消费的 quota 实例数量为 1 个。
- QuotaSpecBinding 定义了：将 QuotaSpec 与 tutorial 项目中的三个微服务：customer、preference、recommendation 进行绑定。
- Rule：在 rule 中执行了配额实例使用的限流 handler 为 memquota。

在 recommendation_rate_limit_handler.yml 配置文件中，定义了如下配置：memquota（mixer 端）。内容如下：

```
apiVersion: "config.istio.io/v1alpha2"
kind: memquota
metadata:
  name: handler
spec:
  quotas:
  - name: requestcount.quota.istio-system
    # default rate limit is 5000qps
    maxAmount: 5000
    validDuration: 1s
    # The first matching override is applied.
    # A requestcount instance is checked against override dimensions.
    overrides:
    - dimensions:
        destination: recommendation
        destinationVersion: v2
```

```
        source: preference
      maxAmount: 1
      validDuration: 1s
```

在配置文件中定义了 memquota handler：从 preference 到 recommendation v2 的请求，最多每秒一次调用。

应用所有的配置：

oc create -f recommendation_rate_limit_handler.yml

oc create -f rate_limit_rule.yml

对三层微服务发起压力测试，观测结果：

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .1; done

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 21760

customer => 503 upstream connect error or disconnect/reset before headers

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 21761

customer => 503 preference => 429 RESOURCE_EXHAUSTED:Quota is exhausted for: RequestCount

可以看到，出现了 429 RESOURCE_EXHAUSTED:Quota 的报错，说明限流起到了效果。

（2） 熔断的实现

接下来，我们为三层微服务配置熔断。还原到初始情况下，对 recommendation 的访问是 v1 和 v2 版本的轮询方式。

我们对 v2 版本设置熔断，Destination Rule 内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: recommendation
spec:
  host: recommendation
```

```
    trafficPolicy:
      tls:
        mode: ISTIO_MUTUAL
    subsets:
      - name: version-v1
        labels:
          version: v1
      - name: version-v2
        labels:
          version: v2
        trafficPolicy:
          connectionPool:
            http:
              http1MaxPendingRequests: 1
              maxRequestsPerConnection: 1
            tcp:
              maxConnections: 1
          outlierDetection:
            baseEjectionTime: 120.000s
            consecutiveErrors: 1
            interval: 1.000s
            maxEjectionPercent: 100
```

可以看到在上述配置中，设置了对 recommendation v2 的 pending 请求最大为 1；每个连接的最大请求数量为 1，最大连接数量为 1。

应用配置：

oc replace -f istiofiles/destination-rule-recommendation_cb_policy_version_v2.yml -n tutorial

destinationrule.networking.istio.io/recommendation replaced

接下来用 siege 发起对三层微服务的请求：

```
# siege -r 1 -c 20 -v http://istio-ingressgateway-istio-system.apps.example.com/
** SIEGE 4.0.2
** Preparing 20 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.04 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.06 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.08 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.12 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.13 secs:      73 bytes ==> GET  /
HTTP/1.1 503   0.18 secs:      92 bytes ==> GET  /
HTTP/1.1 200   0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.20 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.22 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.23 secs:      73 bytes ==> GET  /
HTTP/1.1 503   0.29 secs:      92 bytes ==> GET  /
HTTP/1.1 200   0.31 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.33 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.35 secs:      73 bytes ==> GET  /
HTTP/1.1 200   0.35 secs:      73 bytes ==> GET  /
HTTP/1.1 200   3.05 secs:      70 bytes ==> GET  /
HTTP/1.1 200   6.04 secs:      70 bytes ==> GET  /

Transactions:                 18 hits
Availability:                 90.00 %
Elapsed time:                 6.40 secs
Data transferred:             0.00 MB
Response time:                0.69 secs
```

Transaction rate: 2.81 trans/sec

Throughput: 0.00 MB/sec

Concurrency: 1.95

Successful transactions: 18

Failed transactions: 2

Longest transaction: 6.04

Shortest transaction: 0.04

在返回的结果中，可以看到显示了 503 错误。只要 Istio 检测到 recommendation v2 Pod 有多个待处理的请求，就会打开断路器。

4. 三层微服务配置超时和重试

Istio 的超时和重试是为了更好的处理错误，例如网络故障、应用故障。以超时为例，如果不设置超时，在出现网络故障时，可能导致慢请求堆积占用连接和资源，导致请求响应变慢，甚至导致应用崩溃。作为微服务治理框架 Istio 原生可以支持设置超时和重试，下面我们就分别为三层架构微服务配置超时和重试。

（1） 超时的实现

在微服务的高可用实现中，超时也很重要。超时是指请求在放弃和失败之前等待的时间。在 Istio 中默认服务之间调用的超时时间为 15 秒，我们可以通过 Virtual Service 灵活的为每个服务设置超时时间。需要注意的是，如果想要 Istio 设置的超时时间生效，则必须保证小于应用中设置的超时时间。

下面我们为 recommendation 应用添加 3s 的延迟等待，以模拟应用在 3s 内处理完数据。

修改 RecommendationResource.java 的源代码，添加了两行代码：

```
    @Override
    public void start() throws Exception {
        Router router = Router.router(vertx);
        router.get("/").handler(this::timeout);  //添加内容
```

```
        router.get("/").handler(this::logging);
        router.get("/").handler(this::getRecommendations);
        router.get("/").handler(this::getNow);  //添加内容
        router.get("/misbehave").handler(this::misbehave);
        router.get("/behave").handler(this::behave);

    private void timeout(RoutingContext ctx) {
        ctx.vertx().setTimer(3000, handler -> ctx.next());
```

源码修改以后，重新编译、生成 recommendation v2 的应用包，然后生成 Docker Image，并重新部署到 OpenShift 中，替代原有的 recommendation v2 版本。

cd /root/istio-tutorial/recommendation/java/vertx/

mvn clean package

docker build -t example/recommendation:v2 .

oc delete pod -l app=recommendation,version=v2 -n tutorial

Recommendation v2 重新部署成功：

oc get pods

NAME READY STATUS RESTARTS AGE

customer-775cf66774-qsvv9 2/2 Running 5 11h

preference-v1-667895c986-ljqpg 2/2 Running 3 11h

recommendation-v1-58fcd486f6-cnrj7 2/2 Running 0 4h

recommendation-v2-f967df69-dg7qd 2/2 Running 0 2m

我们对微服务发起访问请求，并且记录命令执行的时间。

while true; do time curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .5; done

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 23

real 0m0.055s

user 0m0.004s

sys 0m0.007s

customer => preference => recommendation v2 from 'f967df69-dg7qd': 254

real 0m3.030s

user 0m0.003s

sys 0m0.007s

从结果我们可以看到，对 recommendation v2 的请求需要等待 3 秒，说明在源码中的配置生效。

接下来，我们设置对 recommendation 服务访问的超时。Virtual Service 的配置内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - route:
    - destination:
        host: recommendation
    timeout: 1.000s
```

在上面的配置文件中，设置对 recommendation 服务的访问超时时间为 1 秒，也就是说，如果被调用的服务在 1 秒内没有响应，就认为这个服务出现故障，不再进行调用。由于我们之前在 recommendation v2 的源码中设置了 3 秒的延迟，因此当该配置生效后，请求不会返回对 recommendation v2 的调用。

应用配置

oc apply -f istiofiles/virtual-service-recommendation-timeout.yml

virtualservice.networking.istio.io/recommendation created

我们通过 curl 发起请求，并记录执行时间。

while true; do time curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .5; done

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 56

real 0m1.065s

user 0m0.005s

sys 0m0.013s

customer => preference => recommendation v1 from '58fcd486f6-m42lh': 57

real 0m1.041s

user 0m0.009s

sys 0m0.010s

可以看到，对 recommendation v1 的访问需要等待 1 秒，不会展示对 recommendation v2 的访问结果。

（2） 重试的实现

微服务中的重试，指的是当某一服务出现故障时，其他微服务访问这个微服务发现无法访问时，并不马上返回访问错误，而是对其进行重试。下面我们为 recommendation 服务配置重试，Virtual Servcie 内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - route:
```

```
    - destination:
        host: recommendation
      retries:
        attempts: 3
        perTryTimeout: 2s
```

可以看到在上述配置中，设置当 recommendation 服务访问出现问题时，重试 3 次，每次重试的间隔是 2 秒。

应用配置：

oc create -f virtual-service-recommendation-v2_retry.yml

virtualservice.networking.istio.io/recommendation created

接下来，我们在代码层面触发 recommendation v2 的错误，模拟真实的服务故障。

oc exec -it -n tutorial $(oc get pods -n tutorial|grep recommendation-v2|awk '{ print $1 }'|head -1) -c recommendation /bin/bash

[jboss@recommendation-v2-f967df69-m8n4j ~]$ curl localhost:8080/misbehave

Following requests to '/' will return a 503

上面触发的是源码中的一个特殊 endpoint，它将使我们的应用程序仅返回`503`。

通过 curl 对服务发起请求：

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .1; done

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1504

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1505

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1506

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1507

如果打开 Kiali，会注意到 v2 版本也接收到了请求，但该失败请求永远不会返回给用户，因为微服务会尝试重新连接 recommendation v2，客户端响应是 recommendation v1 的返回。

通过 Kiali 检测，recommendation v2 有很多报错，如下图 8-60 所示：

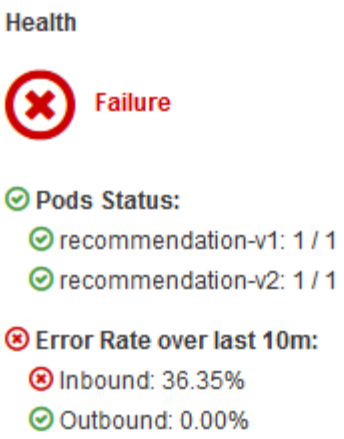

图 8-60 Kiali 告警展示

现在，让 recommendation v2 恢复正常。

oc exec -it -n tutorial $(oc get pods -n tutorial|grep recommendation-v2|awk '{ print $1 }'|head -1) -c recommendation /bin/bash

[jboss@recommendation-v2-f967df69-m8n4j ~]$ curl localhost:8080/behave

Following requests to '/' will return a 200

再度发起访问：

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .1; done

customer => preference => recommendation v2 from 'f967df69-m8n4j': 1

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 2556

可以看到对 recommendation v2 版本的访问已经恢复，重试机制有效。

上面我们配置了 Istio 的超时和重试，可以看到通过 Virtual Service 配置的是每个服务或版本超时、重试的全局默认值。然而，服务的消费者也可以通过特殊的 HTTP header 提供请求级别的值覆盖默认的超时和重试设置。在 Envoy 代理的实现中，对应的 Header 分别是 x-envoy-upstream-rq-timeout-ms 和 x-envoy-max-retries。

5. 三层微服务配置错误注入

在微服务的测试中，有时候需要进行混沌测试，也就是模拟各种微服务的故障。在混沌测试方面，Istio 可以实现错误注入，目前支持的错误有延迟和退出，下面我们分别说明。

（1） 退出错误的实现

我们为 recommendation 服务配置错误注入，模拟访问一个微服务时，返回 503 错误。

默认情况下，对 recommendation 的 v1 和 v2 的访问是正常的。

oc get pods -l app=recommendation -n tutorial

NAME READY STATUS RESTARTS AGE

recommendation-v1-58fcd486f6-cnrj7 2/2 Running 0 1d

recommendation-v2-f967df69-28b6h 2/2 Running 0 3h

配置 Virtual Service，实现访问 recommendation 时，出现 50%的 503 错误。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
spec:
  hosts:
  - recommendation
  http:
  - fault:
      abort:
        httpStatus: 503
        percent: 50
    route:
    - destination:
        host: recommendation
        subset: app-recommendation
```

在上述配置中，为 recommendation 所有版本注入了退出的错误，并定义 HTTP 状态码和错误注入比率。

应用配置：

oc create -f istiofiles/virtual-service-recommendation-503.yml -n tutorial

通过 curl 发起访问，观察返回结果：

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ; sleep .1; done

customer => preference => recommendation v2 from 'f967df69-m8n4j': 25839

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1487

customer => preference => recommendation v2 from 'f967df69-m8n4j': 25840

customer => 503 preference => 503 fault filter abort

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1488

customer => 503 preference => 503 fault filter abort

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1489

通过上面的结果，我们看到注入错误成功实现。

（2） 延迟错误的实现

在本小节中，我们配置混沌测试中的延迟。延迟通常用来模拟微服务调用中的网络延迟。当然，我们可以在微服务的源码中配置延迟来模拟（如 9.4.9 中），但在它的便捷性较低，我们更倾向于使用 Istio 的自带功能实现。

默认情况下，对 recommendation 服务的访问是没有延迟的。我们通过配置 Virtual Service ，实现在 recommendation 服务的访问中加入 50%请求发生延迟，延迟时长为 7s。Virtual Service 配置内容如下：

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: recommendation
  namespace: tutorial
spec:
  hosts:
  - recommendation
  http:
```

```
    - fault:
        delay:
          fixedDelay: 7.000s
          percent: 50
      route:
      - destination:
          host: recommendation
          subset: app-recommendation
```

应用配置：

oc create -f istiofiles/virtual-service-recommendation-delay.yml -n tutorial

virtualservice.networking.istio.io/recommendation created

然后我们通过 curl 发起对微服务的访问，并记录命令的返回时间。

while true; do time curl http://istio-ingressgateway-istio-system.apps.example.com/; sleep .5;done

customer => preference => recommendation v2 from 'f967df69-m8n4j': 25835

real 0m7.051s

user 0m0.006s

sys 0m0.007s

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1484

real 0m0.036s

user 0m0.005s

sys 0m0.007s

customer => preference => recommendation v2 from 'f967df69-m8n4j': 25836

real 0m0.030s

user 0m0.006s

sys 0m0.005s

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1485

real 0m7.036s

user 0m0.007s

sys 0m0.006s

customer => preference => recommendation v2 from 'f967df69-m8n4j': 25837

real 0m7.038s

user 0m0.006s

sys 0m0.009s

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 1486

real 0m0.043s

user 0m0.004s

sys 0m0.011s

从返回结果我们可以看出，对 recommendation 的请求，有 50%出现了 7 秒的延迟。并且延时是不区分版本的。

6. 三层微服务配置黑白名单

（1） 为三层微服务配置黑名单

默认情况下，三层微服务的访问是正常的。我们设置黑名单，让 customer 无法访问 preference。黑名单配置内容如下：

```
apiVersion: "config.istio.io/v1alpha2"
kind: denier
metadata:
  name: denycustomerhandler
spec:
```

```
    status:
      code: 7
      message: Not allowed
  ---
  apiVersion: "config.istio.io/v1alpha2"
  kind: checknothing
  metadata:
    name: denycustomerrequests
  spec:
  ---
  apiVersion: "config.istio.io/v1alpha2"
  kind: rule
  metadata:
    name: denycustomer
  spec:
    match: destination.labels["app"] == "preference" &&
source.labels["app"]=="customer"
    actions:
    - handler: denycustomerhandler.denier
      instances: [ denycustomerrequests.checknothing ]
```

应用配置并发起请求：

oc apply -f acl-blacklist.yml

while true; do curl http://istio-ingressgateway-istio-system.apps.example.com/ ;sleep .001; done

customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.tutorial:Not allowed

customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.tutorial:Not allowed

customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.tutorial:Not allowed

customer => 403 PERMISSION_DENIED:denycustomerhandler.denier.tutorial:Not allowed

返回结果是 403 报错。符合预期，Kiali 展示如下图 8-61 所示：

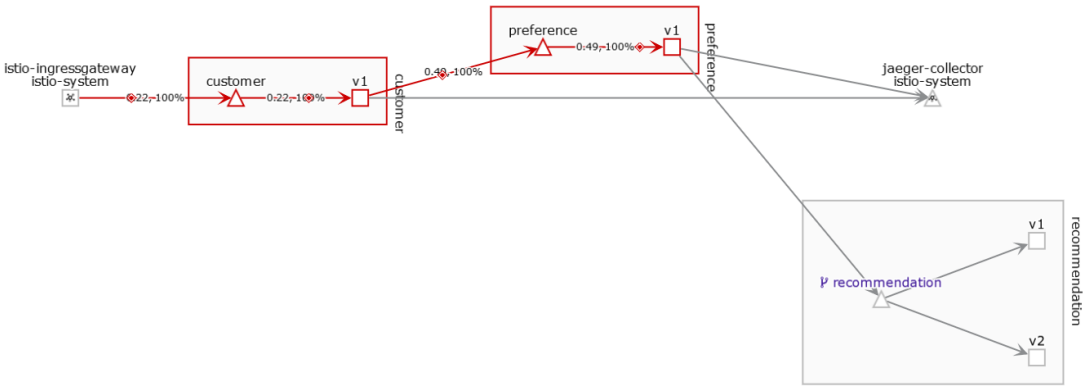

图 8-61 Kiali 展示

（2） 为三层应用配置白名单

在本小节中，我们为 preference 创建一个白名单。这个白名单只允许 preference 访问 recommendation。配置完以后 customer 到 preference 的访问将会返回报错。配置内容如下：

```
apiVersion: "config.istio.io/v1alpha2"
kind: listchecker
metadata:
  name: preferencewhitelist
spec:
  overrides: ["recommendation"]
  blacklist: false
---
apiVersion: "config.istio.io/v1alpha2"
kind: listentry
metadata:
  name: preferencesource
spec:
  value: source.labels["app"]
```

```
---
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: checkfromcustomer
spec:
  match: destination.labels["app"] == "preference"
  actions:
  - handler: preferencewhitelist.listchecker
    instances:
    - preferencesource.listentry
```

应用配置并发起请求：

oc create -f acl-whitelist.yml

#curl http://istio-ingressgateway-istio-system.apps.example.com/

customer => 403 PERMISSION_DENIED:preferencewhitelist.listchecker.tutorial:customer is not whitelisted

customer => 403 PERMISSION_DENIED:preferencewhitelist.listchecker.tutorial:customer is not whitelisted

从结果可以看到返回报错，Kiali 展示如下图 8-62 所示：

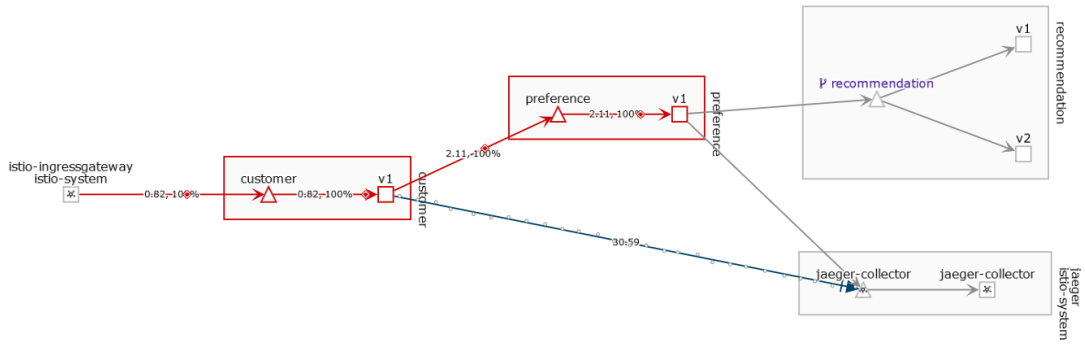

图 8-62 Kiali 展示

7. 三层微服务配置验证与授权

在微服务中，安全中很重要的一部分是认证和授权。在 Istio 中提供了两种类型的验证：

- 传输身份验证：也就是服务到服务的身份验证，Istio 通过双向 TLS（mtls）实现。
- 来源身份认证：也称为最终用户验证，用于最终用户是否有效，常用的有 JWT 认证和 Auth0 等。

在 Istio 中的授权功能也称为基于角色的访问控制（RBAC），支持 Namespace 级别、服务级别和方法级别的访问控制。具有如下特点：

- 简单易用：基于角色的访问控制，可以灵活的定义角色配置权限。
- 覆盖面广：同时支持微服务之间调用的授权和最终用户对服务调用的授权。
- 高性能：授权策略是在每个代理 Envoy 本地执行，效率高。

下面我们就为三层架构微服务配置 JWT 认证和基于角色的访问控制。

（1） 配置客户端 JWT 认证

无论是微服务，还是基于 OpenShift 的普通容器化服务，最外层的应用始终是要暴露给外部使用。如果是暴露给企业内部或者内部员工使用，那么通过证书方式访问就可以。

如果应用暴露给互联网，为了保证数据安全可靠地在用户与服务端之间传输，实现服务端的认证就显得极为必要。目前业内比较标准的做法是使用 JSON Web Token（以下简称 JWT）。

JWT 是一套开放的标准（RFC 7519），它定义了一套简洁（compact）且 URL 安全（URL-safe）的方案，以安全地在客户端和服务器之间传输 JSON 格式的信息。

在基于 OpenShift 的 Istio 中，我们启动 JWT 需要考虑设置的点。以三层微服务为例，如果我们不使用 ingressgateway，我们需要将 JWT 认证的目标端（下面的配置文件 target）放在 customer 微服务处。这种模式在开发测试环境是可以的。

在生产环境，三层微服务使用 ingressgateway，并且入口通过 OpenShift Router 上的路由。那么，我们的 JWT 目标端（下面的配置文件 target）需要设置在 ingressgateway 的 Service 上。

我们的验证采取第二种方式。认证配置文件内容如下：

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "customerjwt"
spec:
  targets:
    - name: istio-ingressgateway
  origins:
    - jwt:
        issuer: "testing@secure.istio.io"
        jwksUri:
"https://gist.githubusercontent.com/lordofthejars/7dad589384612d7a6e183
98ac0f10065/raw/ea0f8e7b729fb1df25d4dc60bf17dee409aad204/jwks.json"
  principalBinding: USE_ORIGIN
```

配置中定义 JWT 的目标端为 istio-ingressgateway，JWT 的 issuer 和 jwksUri 我们使用 Istio 提供的测试链接，否则需要自行搭建认证服务器。

应用配置，注意指定项目为 istio-system，因为 istio-ingressgateway 是该项目中的 Service。

oc create -f istiofiles/enduser-authentication-jwt-tutorial.yml -n istio-system

对应用发起请求：

curl http://istio-ingressgateway-istio-system.apps.example.com/
Origin authentication failed.

可以看到会有 Origin authentication failed 报错，也就是说，没有 token 的访问被拒绝。

接下来，我们从 jwksUri 获取 token：

token=$(curl
https://gist.githubusercontent.com/lordofthejars/a02485d70c99eba70980e0a92b2c97ed/raw/f16b9
38464b01a2e721567217f672f11dc4ef565/token.simple.jwt -s)

使用 token 发起请求，访问正常。

curl -H "Authorization: Bearer $token" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 775

至此，我们验证了将 JWT 目标端设置在 ingressgateway 是成功的。这也是在生产中我们推荐的方式。

（2） 配置基于角色的访问控制

lstio RBAC (Role Based Access Control)的实现是通过 ServiceRole 和 ServiceRoleBinding 两个对象的定义。

ServiceRole 的定义包含了一系列规则 。每个规则有如下标准字段：

- services: services 列表。
- methods: HTTP 方法。对于 gRPC，此字段将被忽略，因为该值始终为“POST”。
- paths: HTTP 路径或 gRPC 方法。请注意，gRPC 方法应以“/packageName.serviceName/methodName”的形式呈现，并且区分大小写。

下面是 ServiceRole 是对 products 服务的设置：

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: products-viewer
  namespace: default
spec:
  rules:
  - services: ["products.svc.cluster.local"]
    methods: ["GET", "HEAD"]
    constraints:
    - key: "destination.labels[version]"
      values: ["v1", "v2"]
```

可以看到 ServiceRole 定义了对版本“v1”和“v2”的“products.svc.cluster.local”服务具有“read”（“GET”和“HEAD”）访问权限。未指定“path”，因此它适用于服务中的任

何路径。

接下来，我们看 ServiceRoleBinding。它的规范包括两部分：

- roleRef 字段，它引用同一 Project 的 ServiceRole 对象。
- 分配给 roles 的 subjects 列表。

subjects 定义了一个身份，包含用户或由一组属性标识，如下表 8-9 所示

表 8-9 Subject 字段描述

字段	类型	描述
user	string	可选项。代表一个 subject 的用户 name/ID。
properties	map<string,string>	可选项。一组标识 subject 的属性。前面的 `ServiceRoleBinding` 例子展示了一个 `source.namespace` 属性的例子。

如下是一个名为 test-binding-products 的 ServiceRoleBinding 对象，将两个 subjects 绑定到了名为 product-viewer 的 ServiceRole 上。

- alice@yahoo.com 用户
- abc namespace 下的所有 Service。

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: test-binding-products
  namespace: default
spec:
  subjects:
  - user: alice@yahoo.com
  - properties:
      source.namespace: "abc"
  roleRef:
    kind: ServiceRole
   name: "products-viewer"
```

也就是说，将名为 products-viewer 的 ServiceRole（有 read 权限访问

products.svc.cluster.local）赋予给从 abc 项目的 alice@yahoo.com 用户。在实际的应用中，我们也可以不做 ServiceRoleBinding ，也就是，客户端访问的时候，需要手工指定 role

接下来，我们为三层微服务配置基于角色的访问控制。首先，使用 ClusterRbacConfig 对象在集群范围内启用 Istio Authorization。配置内容如下：

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ClusterRbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["tutorial"]
```

应用配置文件：

oc create -f istiofiles/authorization-enable-rbac.yml -n istio-system

clusterrbacconfig.rbac.istio.io/default created

我们已经在 Istio 中启用了 RBAC。在没进行任何授权的情况下，对三层微服务发起请求，出现报错：

curl http://istio-ingressgateway-istio-system.apps.example.com/

RBAC: access denied

接下来，我们使用 RBAC+JWT 的方式，对微服务进行授权。在上一小节中，我们已经针对 ingressgateway 作为目标端进行了配置。因此重复步骤不再赘述。

我们定义 Servicerole，它配置了对 istio-ingressgateway 与 get 权限，内容如下：

```
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRole
metadata:
  name: istio-ingressgateway
spec:
  rules:
    - services: ["*"]
```

```
      methods: ["GET"]
---
apiVersion: rbac.istio.io/v1alpha1
kind: ServiceRoleBinding
metadata:
  name: bind-istio-ingressgateway
spec:
  subjects:
    - user: "*"
      properties:
        request.auth.claims[role]: "istio-ingressgateway"
  roleRef:
    kind: ServiceRole
  name: istio-ingressgateway
```

应用配置：

oc create -f istiofiles/namespace-rbac-policy-jwt.yml -n istio-system

servicerole.rbac.istio.io/customer created

servicerolebinding.rbac.istio.io/bind-customer created

接下来，我们用包含 customer 的 role 声明的 token 进行新的调用：

首先获取 token：

token=$(curl

https://gist.githubusercontent.com/lordofthejars/f590c80b8d83ea1244febb2c73954739/raw/21ec0ba0184726444d99018761cf0cd0ece35971/token.role.jwt -s)

进行调用，返回成功。

curl -H "Authorization: Bearer $token" http://istio-ingressgateway-istio-system.apps.example.com/

customer => preference => recommendation v1 from '58fcd486f6-cnrj7': 777

至此，基于 RBAC 和 JWT 的认证方式就配置完成了。