

## 9.1.1 Istio 调优参考

在不进行任何调优的情况下，Istio 在大量请求时可能发生性能问题，比如 istio-pilot 自动横向扩展为多个实例，每个实例占用大量内存，但是 Istio 和 Envoy 的连接数却很低，也就是访问请求量提升不上去。解决此类问题有几个方面调整参考：

- 启用 Istio Namespace Isolation。。
- 为每个项目单独创建 Ingressgateway。
- Istio 参数调整。

接下来，我们针对这三方面展开讨论。

### 1. 启用 Istio Namespace Isolation

Namespace Isolation 是 Istio1.1 中的功能。默认情况下，在 Istio 管理的不同 Namespace 中，Sidecar 之间是可以相互通讯的。Namespace Isolation 可以让一个 Namespace 里的 Sidecar 只能与内部 Sidecar 以及 Istio-System 全局 Namespace 通讯。这样做的好处是减少微服务之间不必要的交互，提升性能。接下来，我们介绍 Namespace Isolation 的具体实现。

在我们的环境中，和 Istio 相关的 Namespace 有三个：myproject、istio-system、tutorial。其中 myproject 和 tutorial 分别运行两套微服务。istio-system 运行 Istio 的相关容器，如图 9-24 所示，它们是 Istio 的各个组件。

```
[root@master ~]# oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
3scale-istio-adapter-5b4f4fcd77-vvxj9	1/1	Running	11	52d
elasticsearch-0	0/1	Running	54	52d
grafana-6c5dfdf5bd-ns8b2	1/1	Running	14	52d
ior-679b475484-x5bdl	1/1	Running	22	52d
istio-citadel-66cf447cbd-9tvd8	1/1	Running	13	52d
istio-egressgateway-69b65dddf5-tv7vd	1/1	Running	4	16d
istio-egressgateway-69b65dddf5-z4lt6	1/1	Running	0	2m
istio-galley-5dbd58568d-kx9n8	1/1	Running	6	10d
istio-ingressgateway-b688c9d9b-rbqww	1/1	Running	3	16d
istio-pilot-79668d4bf6-2zxhw	2/2	Running	22	52d
istio-policy-5f45fcf95f-69zhm	2/2	Running	163	52d
istio-sidecar-injector-7c44bcbbcd-xd8jx	1/1	Running	4	10d
istio-telemetry-7fcd854d6b-jz6dq	2/2	Running	151	51d
jaeger-agent-fqqlq	1/1	Running	11	65d
jaeger-collector-576b66f88c-vqcjw	1/1	Running	43	51d
jaeger-query-7549b87c55-t8dlb	1/1	Running	35	51d
kiali-7475849854-lw2h2	1/1	Running	11	52d
prometheus-5dfcf8dcf9-7z7wh	1/1	Running	22	52d

图 9-42 istio-system 项目下的 Pod

myproject 项目下运行的是 bookinfo 的微服务。

```
# oc get pods -n myproject
```

NAME	READY	STATUS	RESTARTS	AGE
productpage-v1-68f9bc6f97-lzrlk	2/2	Running	18	52d
ratings-v1-78cbc4df5-tfvtq	2/2	Running	16	52d
reviews-v1-778cf955bb-6l4ss	2/2	Running	20	52d
reviews-v2-d4c99fdc8-pwnrz	2/2	Running	22	52d
reviews-v3-78cbff4cfd-rm9mm	2/2	Running	18	52d

tutorial 项目下运行的是我们迁移到 Istio 的三层微服务。

```
# oc get pods -n tutorial
```

NAME	READY	STATUS	RESTARTS	AGE
customer-775cf66774-6zdvt	2/2	Running	76	27d
preference-v1-667895c986-g7lng	2/2	Running	45	27d
recommendation-v1-58fcd486f6-zfljg	2/2	Running	19	27d

默认情况下，tutorial 和 myproject 之间的 Sidecar 是可以互相通讯的，我们进行简单的测试。

获取 tutorial 项目中微服务的 Service IP 以便后面测试。

```
# oc get svc -n tutorial
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
customer	ClusterIP	172.30.27.184	<none>	8080/TCP	51d
preference	ClusterIP	172.30.98.89	<none>	8080/TCP	51d
recommendation	ClusterIP	172.30.113.197	<none>	8080/TCP	51d

在 myproject 里 productpage Pod 的 Sidecar 中，对 tutorial namespace 微服务

Preference 的 Service IP 发 curl 请求可以成功，如图 9-43 所示：

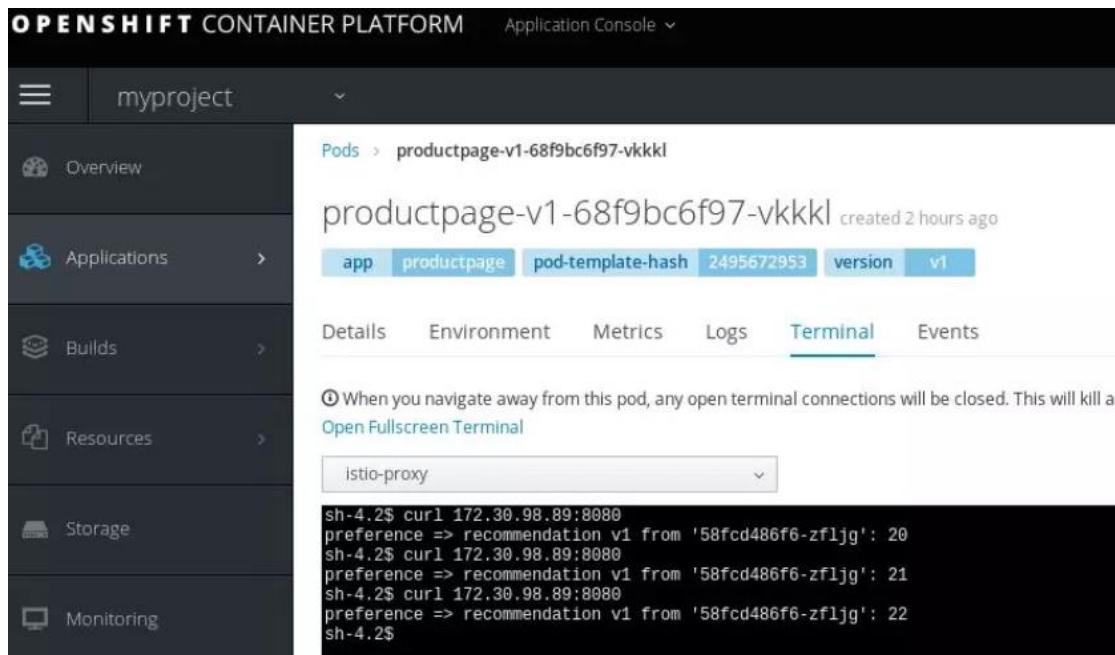


图 9-43 不同项目 Sidecar 可以访问

接下来，我们启用两个 Sidecar 命名空间隔离配置。

在下面的配置中，通过配置 Sidecar 对象的 egress 字段，设定 Sidecar 的作用域：只能与本 Namespace（namespace: myproject）内的 Sidecar 以及与 istio-system 中 istio-telemetry 和 istio-policy 的 Sidecar 通信。在 myproject 中应用的 Sidecar 配置内容如下：

```
# cat sidecar-myproject.yaml
```

```
apiVersion: networking.istio.io/v1alpha3
kind: Sidecar
metadata:
  name: default-sidecar-scope
  namespace: myproject
spec:
  egress:
    - hosts:
      - ".*"
      - "istio-system/istio-telemetry.istio-system.svc.cluster.local"
      - "istio-system/istio-policy.istio-system.svc.cluster.local"
```

---

Sidecar 的隔离是单向的，同样需要在 tutorial 项目配置 Sidecar 对象，内容如下：

# cat sidecar-tutorial.yaml

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

kind: Sidecar

metadata:
  name: default
  namespace: tutorial

spec:
  egress:
    - hosts:
      - ".*/*"
      - "istio-system/istio-telemetry.istio-system.svc.cluster.local"
      - "istio-system/istio-policy.istio-system.svc.cluster.local"
  ---
```

应用两个配置文件如下：

# oc apply -f sidecar-myproject.yaml

sidecar.networking.istio.io/default-sidecar-scope created

# oc apply -f sidecar-tutorial.yaml

sidecar.networking.istio.io/default created

再次在 myproject 里 productpage Pod 的 Sidecar 中，对 tutorial namespace 微服务

Preference 的 Service IP 发 curl 请求失败，出现 404 报错。Sidecar 隔离成功，如图 9-44 所示：

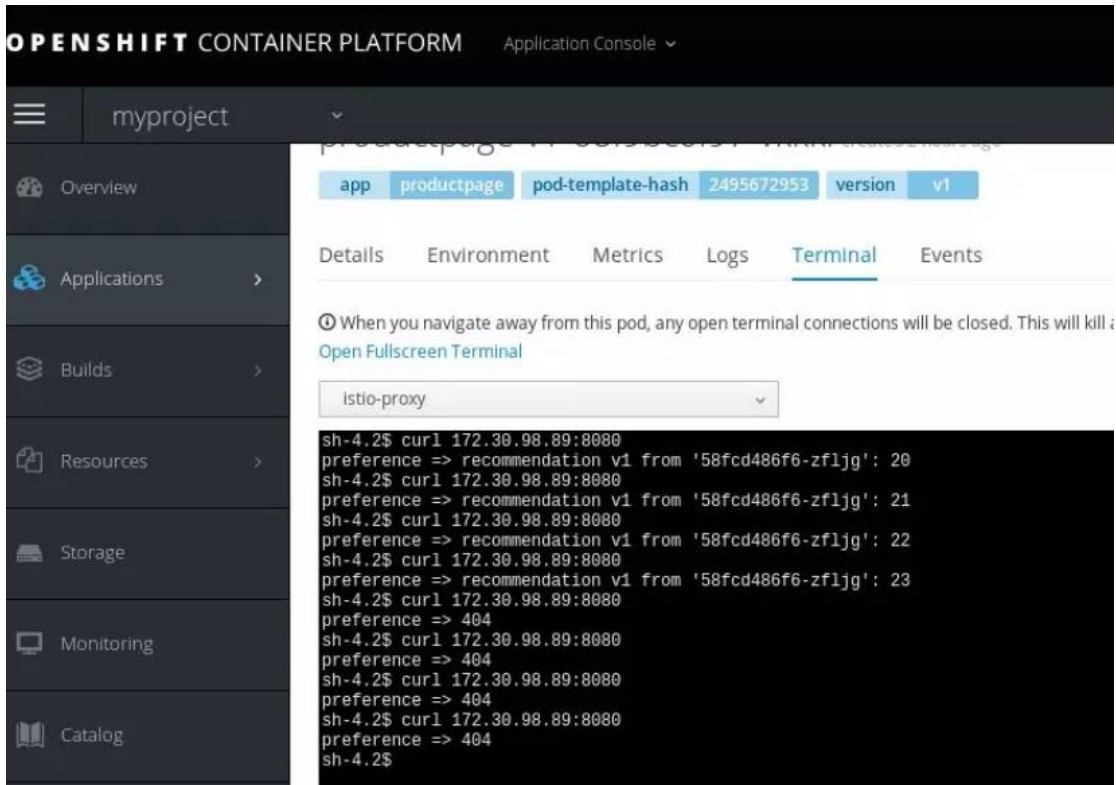


图 9-44 不同项目 Sidecar 不可访问

接下来，我们验证两个微服务项目中的 Sidecar 与 istio-system Sidecar 之间的通讯是正常的。

首先获取 istio-telemetry 的 Service IP，以便后面测试使用。

```
# oc get svc |grep -i istio-telemetry
```

istio-

telemetry	ClusterIP	172.30.233.126	<none>	9091/TCP,15004/TCP,15014/TCP,42422/TCP
-----------	-----------	----------------	--------	--

65d

对 istio-telemetry 的 Service IP 发起 curl 请求，可以正常访问，如图 9-45 所示：

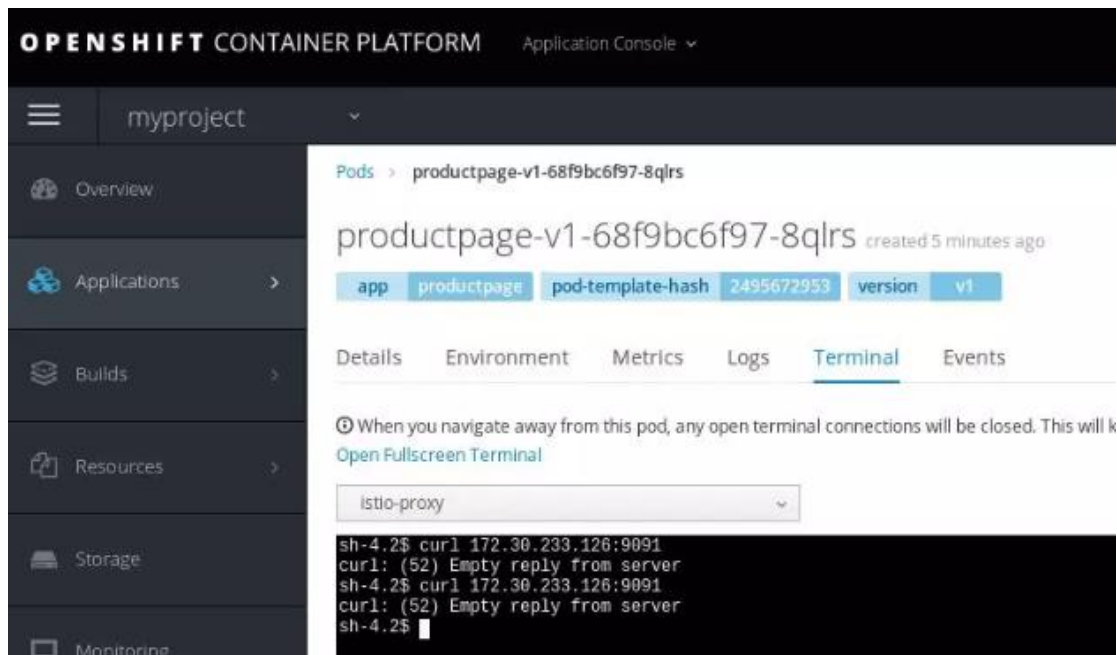


图 9-45 微服务的 Sidecar 可以访问 istio

至此，微服务之间 Sidecar 的隔离就配置完成了。

## 2. 为每个项目单独创建 Ingressgateway

默认情况下，一个 Istio 在 istio-system 命名空间中配置一个全局 Ingressgateway，作为微服务整体流量的入口。在 OpenShift 上，在全局 Ingressgateway 上创建 Gateway 后，OpenShift 会自动创建 Gateway 对应的路由。这样入口流量会先经过 OpenShift Router 再经过 Istio Ingressgateway 最后达到应用微服务。

当入口流量过大时，我们可以在每个 Namespace 中独立配置一组 Ingressgateway。这样做的好处是提高了 Istio 系统的吞吐量，避免全局 Ingressgateway 成为瓶颈。当然，此时如果入口流量过大，OpenShift Router 同样可能存在性能的瓶颈。这时候可以增加 Router Pod 的副本数，或者为单独的 Namespace 创建独立的一组 Router（即 Namespace 中既有独立的 Ingressgateway，又有独立的 Router），以此来提升 Router 的处理能力。接下来，我们展示配置独立 Ingressgateway 的步骤。

默认情况下，myproject 使用的是 istio-system 中的全局 Ingressgateway，接下来，我们在 myproject 中创建一个独立的 Ingressgateway。

我们使用两个配置文件完成这个操作。customgateway.yaml 用于在指定 myproject 中部

署 Ingressgateway。gateway.yaml 用于配置 Gateway 对象。由于篇幅有限，我们将两个配置文件放到 Github 上，地址如下：

<https://github.com/ocp-msa-devops/istio-tutorial/blob/master/customgateway.yaml>

<https://github.com/ocp-msa-devops/istio-tutorial/blob/master/gateway.yaml>

为了操作便捷，customgateway.yaml 配置文件中指定的 proxy 的 docker image 地址为：docker.io/istio/proxyv2:1.0.3。红帽企业级用户建议使用红帽提供的 proxyv2 镜像，然后将该镜像 tag 后推送到本地镜像仓库地址，在 customgateway.yaml 配置文件 image 字段变更为本地仓库的地址。

应用独立 Ingressgateway 配置：

```
# oc create -f customgateway.yaml

serviceaccount/customgateway-service-account created

clusterrole.rbac.authorization.k8s.io/customgateway-myproject created

clusterrolebinding.rbac.authorization.k8s.io/customgateway-myproject created

service/customgateway created

deployment.extensions/customgateway created

horizontalpodautoscaler.autoscaling/customgateway created

应用 Gateway 资源对象

# oc create -f gateway.yaml

gateway.networking.istio.io/bookinfo-gateway created

virtualservice.networking.istio.io/bookinfo created

创建成功之后，会启动 Ingressgateway 的 Pod 和 Gateway 对象。

# oc get pods
```

NAME	READY	STATUS	RESTARTS	AGE
customgateway-cddc84cc8-t2d54	1/1	Running	2	19h
details-v1-7476c8db95-2b92t	2/2	Running	4	20h
productpage-v1-68f9bc6f97-jzx9g	2/2	Running	4	20h
ratings-v1-78cbc4df5-nqc7m	2/2	Running	4	20h
reviews-v1-778cf955bb-c76dt	2/2	Running	4	20h
reviews-v2-d4c99fdc8-nmk5m	2/2	Running	4	20h

```
reviews-v3-78cbff4cfd-9lgl2      2/2      Running    4          20h
```

查看创建好的 gateway 和 virtualservice。

```
# oc get gateway
```

```
NAME                                AGE
```

```
bookinfo-gateway    7s
```

```
# oc get virtualservice
```

```
NAME          GATEWAYS          HOSTS      AGE
```

```
bookinfo    [bookinfo-gateway]  [*]        16s
```

可以看到在 bookinfo 的 VirtualService 中指定的 Gateway 是创建的独立的

Ingressgateway bookinfo-gateway。

接下来，手工为创建的独立 Ingressgateway 在 Router 上创建路由，以便外部请求可以通过 Router 访问。

查看自定义 ingressgateway 的 Service 名称：

```
# oc get svc | grep -i custom
```

```
customgateway    LoadBalancer    172.30.97.172    172.29.123.118,172.29.123.118
```

```
80:30585/TCP,443:32123/TCP    1h
```

为独立的 ingressgateway 在 Router 创建路由，如图 9-46 所示：

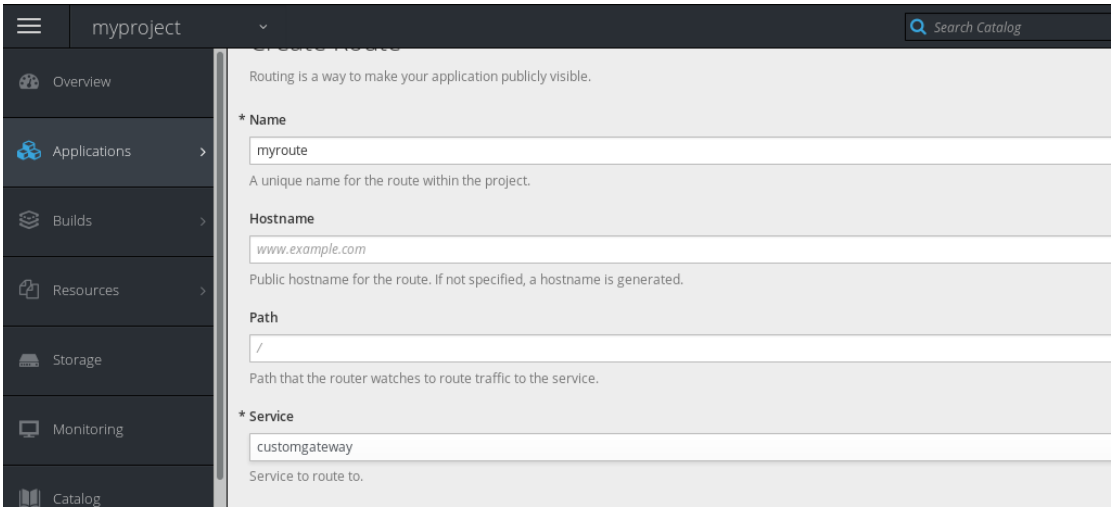


图 9-46 创建路由

其中 Name 指定为 myrouter，选择 Service 为 customgateway。配置访问方式（启用 TLS）如图 9-47 所示：



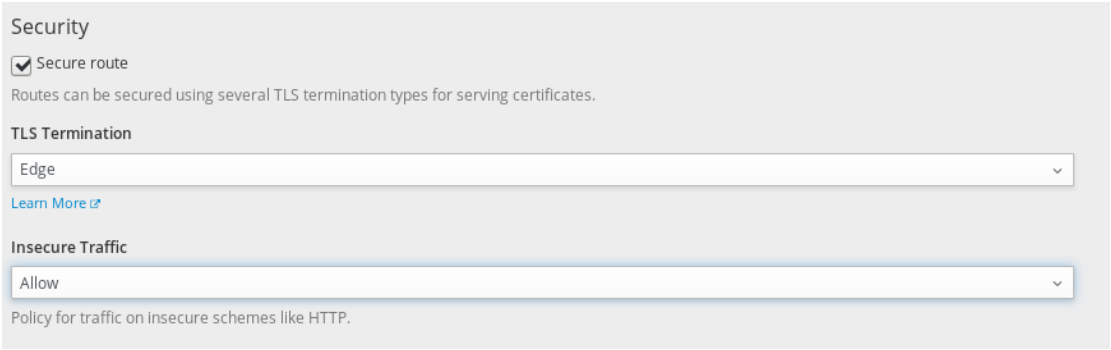


图 9-47 开启 Route 的 SSL

路由创建成功:

```
# oc get route
```

NAME	HOST/PORT	PATH	SERVICES
myroute	myroute-myproject.apps.example.com	customgateway	http
edge/Allow	None		

接下来，通过浏览器访问应用的路由，可以正常访问，如图 9-48 所示:

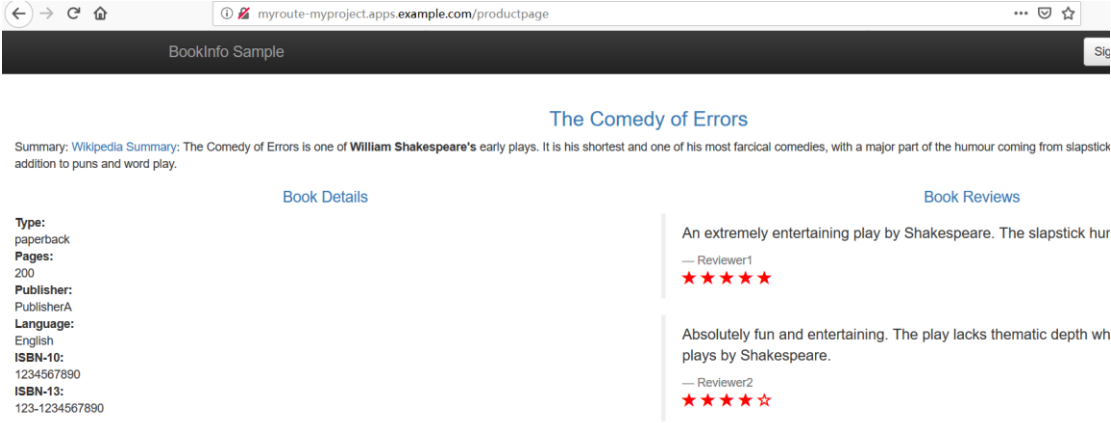


图 9-48 访问 bookinfo

进行多次访问后，使用 Kiali 进行观测，可以看到流量的入口是 customgateway，如图 9-49 所示:

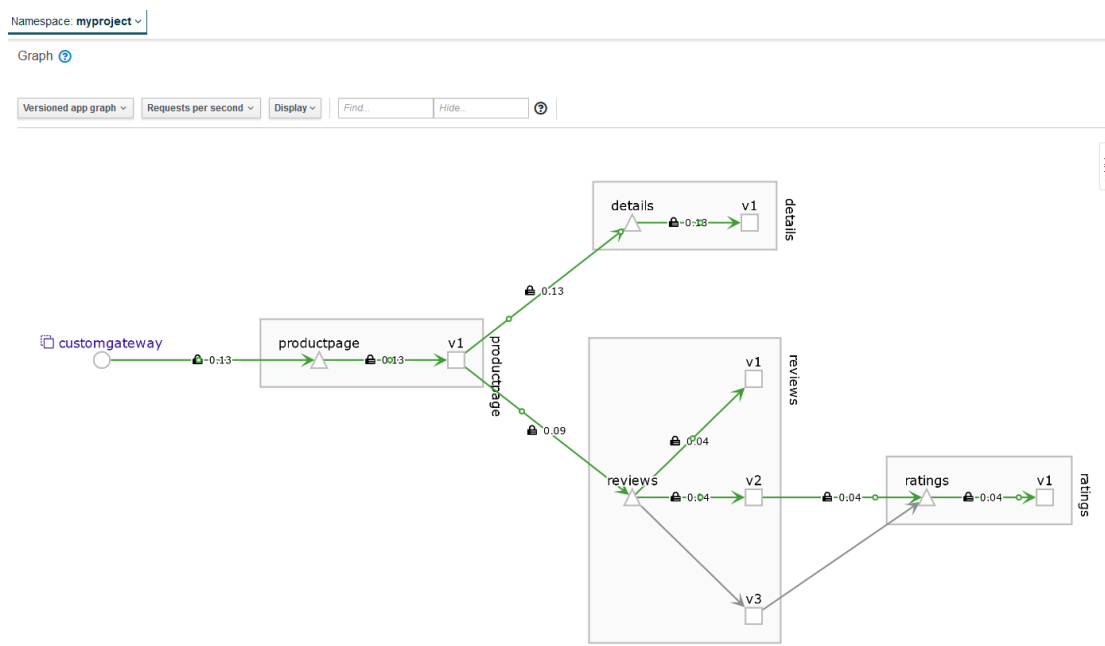


图 9-49 Kiali 展现

至此，基于 Namespace 创建独立的 Ingressgateway 就配置成功了。

### 3. Istio 参数调整

关于 Istio 安装方式，目前社区采用 Helm 的方法。红帽 Istio 采用 Operator 的方式。目前 Operator 中的有 Istio 的安装参数未开放，后续可能会开放，为了使读者能够有个清晰的理解，本小节也将进行介绍。

大规模运行 Istio 时，和性能相关的参数有多个，我们列出几个重要性最高的。其他参数请参照 Istio 官方文档。<https://istio.io/docs/reference/commands/pilot-discovery/>

#### (1) keepaliveMaxServerConnectionAge

本参数为这是 Helm 部署参数。它控制的是这是参数控制 Sidecar 连接 Polit 的最长时间。在 Istio 1.1 中，默认时间是 30 分钟。

我们知道，在 Istio 中 Sidecar 是需要和 Polit 进行通讯。当访问请求量较大时，如果 Polit 启动 HPA，Polit 的实例数量通常不止一个。这就带来一个问题：如果 Istio 的访问量突然增大，已经与现有 Polit 的实例建立连接的 Sidecar，何时连接到新的 Polit 实例？这与 keepaliveMaxServerConnectionAge 参数设置有关。

`keepaliveMaxServerConnectionAge` 参数如果设置的太小，可能会造成访问流量的丢失（Sidecar 与相同或不同的 Polit 实例频繁建立链接），如果设置的太大，在压力激增的情况下，Sidecar 无法与旧的 Polit 断开、不能连接新的 Polit，将会造成 Polit 实例负载不均衡。

目前关于这个参数的设置，社区里也做过一些测试，有的开发者设置为 6 分钟。具体的设置数值，需要根据自身 Istio 的情况进行测试后获得。

目前在 OpenShift 上安装 Istio，这个参数的配置未开放。后续基于 Istio 的 OpenShift 有开放的可能。

## (2) `global.proxy.concurrency`

本参数为 Helm 部署参数。它控制 Proxy 工作的线程数量。默认数值是 2。它会影响 Sidecar 的资源利用率和延迟。如果将 `global.proxy.concurrency` 设置为 0，每个 Proxy 工作线程占用一个 CPU Core（如果开启 CPU 硬件超线程，则占用 CPU 硬件超线程）。默认数值是 2 的情况下，两个 Proxy 工作的线程共享一个 CPU Core（如果开启 CPU 硬件超线程，则占用 CPU 硬件超线程）。

针对 Sidecar 的资源分配，我们给它分配的 CPU 资源时间片越多，它的性能就会越好、应用延迟会越低，但资源消耗会越大。具体的参数设置，取决于自身环境实际测试结果。

目前在 OpenShift 上安装 Istio，这个参数的配置未开放。后续基于 Istio 的 OpenShift 有开放的可能。

## (3) `global.enableTracing`

本参数为 Polit 配置参数。可以通过 Polit ConfigMap 进行配置，重启 Polit 后生效，参数默认为 `disable`。Tracing 对的是 Istio 中的 Jaeger。启动 Tracing 会造成大量的资源开销并影响吞吐量。生产环境中，我们建议将此参数设置为 `disable`。

基于 OpenShift 的 Istio 可以调整此参数，可以在安装配置文件 `istio-installation.yaml` 中进行设置（下面配置为开启 tracing）：

```
tracing:
  enabled: true
```

## (4) Telemetry 和 Gateways HPA 阈值

本参数配置参数。他们会影响 Istio 的性能。在规模较小的 Istio 环境中，Gateway 和 Telemetry 的 HPA 可以关闭。当 Istio 规模较大时，需要开启此参与，以应对突发的大流量访问。

在基于 OpenShift 的 Istio 中，HPA 是否打开以及 Pod 的伸缩范围配置参数，在 Istio 的安装文件可以配置（istio-installation.yaml）。

### Istio gateway 配置部分

```
gateways:

  istio-egressgateway:

    autoscaleEnabled: false

    autoscaleMin: 1

    autoscaleMax: 5

  istio-ingressgateway:

    autoscaleEnabled: false

    autoscaleMin: 1

    autoscaleMax: 5

    ior_enabled: false
```

autoscaleEnabled 设置是否启动用自动扩展，autoscaleMin 设置扩容的最少 Pod 数量，autoscaleMax 设置扩容的最多 Pod 数量。

### Istio Mixer 配置部分：

```
mixer:

  enabled: true

  policy:

    autoscaleEnabled: false

  telemetry:

    autoscaleEnabled: false

  resources:
```

```
requests:

  cpu: 100m

  memory: 1G

limits:

  cpu: 500m

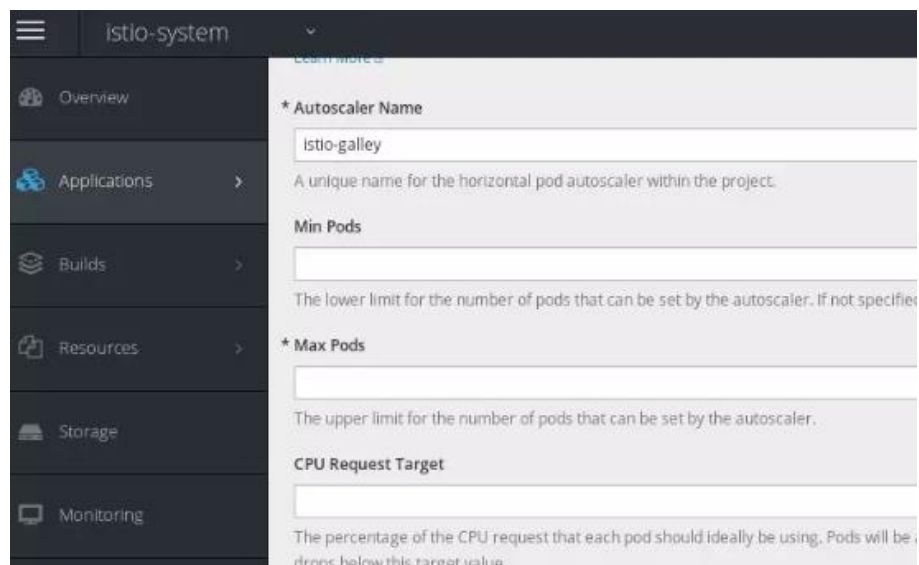
  memory: 4G
```

enabled 设置是否使用启用 Mixer Policy。autoscaleEnabled 设置是否启动用自动扩展，autoscaleMin 设置缩容的最少 Pod 数量，autoscaleMax 设置扩容的最多 Pod 数量。

Resources 代表 Telemetry Pod request 的 CPU 和内存，limit 代表 Telemetry Pod 最多使用的 cpu 和内存。

Resources 100m 代表 0.1 个 CPU Core 的运算能力。Limit 为 500m，表示 Pod 最多可以获得 0.5 CPU Core 的运算能力。

OpenShift 的 HPA 可以基于 CPU 和内存的利用率。CPU 利用率的计算方法是用 Pod 运行实际获取到 CPU 资源，除以 CPU Request（Resources 的设置）得到最近一分钟内一个平均值。在配置 HPA 的参数时，需要设置 CPU Request Target，如图 9-50 所示。当 CPU 利用率拆过 CPU Request Target 时，将会触发 HPA。



The screenshot shows the OpenShift console interface for configuring a Horizontal Pod Autoscaler (HPA) in the `istio-system` namespace. The left sidebar contains navigation links: Overview, Applications, Builds, Resources, Storage, and Monitoring. The main panel displays the configuration form for the HPA, with the following fields and descriptions:

- \* Autoscaler Name:** A text input field containing the value `istio-galley`. Below the field is the description: "A unique name for the horizontal pod autoscaler within the project."
- Min Pods:** A text input field. Below the field is the description: "The lower limit for the number of pods that can be set by the autoscaler. If not specified, the autoscaler will default to 1 pod."
- \* Max Pods:** A text input field. Below the field is the description: "The upper limit for the number of pods that can be set by the autoscaler."
- CPU Request Target:** A text input field. Below the field is the description: "The percentage of the CPU request that each pod should ideally be using. Pods will be scaled up or down based on this target value."

图 9-50 HPA CPU 参数配置

如果我们将 Resources 设置的太高，那么 Telemetry 和 Gateways 扩容几乎应用不会发生，这就失去了 HPA 的意义，也浪费资源。如果设置的太低，频繁发生扩缩容，会降低

Istio 的稳定性。Telemetry 的 ResourcesCPU 和内存可以适当调小。

## (5) 设置 Metrics 的 Prometheus Adapter

在基于 OpenShift 的 Istio 中，Metrics 信息通常有两种展现方式：

- 输出到 Prometheus（默认方式），然后通过 Grafana 进行展现。
- Stdio 输出日志和指标数据。

Metrics 输出到 Prometheus 的好处是方便统一查询和通过 Grafana 展现。缺点是会增加 Prometheus 的资源消耗。Stdio 输出日志和指标数据不会增加 Istio 的负载，缺点是无法统一展现。我们可以根据日志和指标数据的特点，将不同类型的日志和指标数据输出到不同的位置。在基于 OpenShift 的 Istio 中，我们推荐将日志输出到 Prometheus 中，因此对本机输出不做介绍。

在日志重定向到 Prometheus 的情况下，我们可以设置哪些日志传输到 Prometheus。原始的定义在 Helm 安装 Istio 的配置文件中，

<https://raw.githubusercontent.com/istio/istio/master/install/kubernetes/helm/istio/charts/mixer/templates/config.yaml>

在 config.yaml 文件中定了 promhttp rule，它将 requestcount.metric、requestduration.metric、requestsize.metric、responsesize.metric 发送到 handler.prometheus，实现将 http metric 发送到 prometheus。内容如下：

```
apiVersion: "config.istio.io/v1alpha2"

kind: rule

metadata:

  name: promhttp

  namespace: {{ .Release.Namespace }}

  labels:

    app: {{ template "mixer.name" . }}

    chart: {{ template "mixer.chart" . }}

    heritage: {{ .Release.Service }}

    release: {{ .Release.Name }}

spec:
```

```

    match: (context.protocol == "http" || context.protocol == "grpc")
    && (match((request.useragent | "-"), "kube-probe*") == false) &&
    (match((request.useragent | "-"), "Prometheus*") == false)

    actions:

    - handler: prometheus

    instances:

    - requestcount

    - requestduration

    - requestsize

    - responsesize

---

```

我们 Istio 中可以通过创建 **Metric** 模板来定义指标。针对输出到 **Prometheus** 的情况，我们可以设置想要收集的日志和指标数据，避免收集所有数据，这样可以减少 **Prometheus** 的资源开销。

我们展示如何自定义一个 **Metrics**，然后将监控指标重定向到 **Prometheus** 的 UI，配置文件内容如下。

**# cat metrics-crd.yaml**

```

# Configuration for metric instances

apiVersion: "config.istio.io/v1alpha2"

kind: instance

metadata:

  name: doublerequestcount

  namespace: istio-system

spec:

  compiledTemplate: metric

  params:

    value: "2" # count each request twice

    dimensions:

```

```

        reporter: conditional((context.reporter.kind | "inbound") ==
"outbound", "client", "server")

        source: source.workload.name | "unknown"

        destination: destination.workload.name | "unknown"

        message: '"twice the fun!'"

        monitored_resource_type: '"UNSPECIFIED"'
---
# Configuration for a Prometheus handler
apiVersion: "config.istio.io/v1alpha2"

kind: prometheus

metadata:

  name: doublehandler

  namespace: istio-system

spec:

  metrics:

    - name: double_request_count # Prometheus metric name

      instance_name: doublerequestcount.instance.istio-system # Mixer
instance name (fully-qualified)

      kind: COUNTER

      label_names:

        - reporter

        - source

        - destination

        - message
---
# Rule to send metric instances to a Prometheus handler
apiVersion: "config.istio.io/v1alpha2"

kind: rule

metadata:

```



```
name: doubleprom

namespace: istio-system

spec:

  actions:

    - handler: doublehandler.prometheus

      instances:

        - doublerequestcount
```

上面的配置文件让 Mixer 把指标数值发送给 Prometheus。其中包含部分内容：  
instance 配置、handler 配置以及 rule 配置。

kind: metric 为指标值（或者 instance）定义了结构，命名为 doublerequestcount。  
Instance 配置告诉 Mixer 如何为所有请求生成指标。指标来自于 Envoy 汇报的属性（然后由 Mixer 生成）。

doublerequestcount.metric 配置让 Mixer 给每个 instance 赋值为 2。因为 Istio 为每个请求都会生成 instance，这就意味着这个指标的记录的值得于收到请求数量的两倍。并且输出为"twice the fun!"，以便我们识别。

每个 doublerequestcount.metric 都有一系列的 dimension。dimension 提供了一种为不同查询和需求对指标数据进行分割、聚合以及分析的方式。例如在对应用进行排错的过程中，可能只需要目标为某个服务的请求进行报告。这种配置让 Mixer 根据属性值和常量为 dimension 生成数值。例如 source 这个 dimension，他首先尝试从 source.service 属性中取值，如果取值失败，则会使用缺省值 "unknown"。而 message 这个 dimension，所有的 instance 都会得到一个常量值："twice the fun!"。

kind: prometheus 这一段定义了一个叫做 doublehandler 的 handler。spec 中配置了 Prometheus 适配器收到指标之后，如何将指标 instance 转换为 Prometheus 能够处理的指标数据格式的方式。配置中生成了一个新的 Prometheus 指标，取名为 double\_request\_count。Prometheus 适配器会给指标名称加上 istio\_ 前缀，因此这个指标在 Prometheus 中会显示为 istio\_double\_request\_count。指标带有三个标签，和 doublerequestcount.metric 的 dimension 配置相匹配。

我们应用配置文件：

```
# oc apply -f metrics-crd.yaml
```

```
instance.config.istio.io/doublerequestcount created
```

```
prometheus.config.istio.io/doublehandler created
```

```
rule.config.istio.io/doubleprom created
```

查看对应的 instance 已经生成：

```
# oc get instance
```

NAME	AGE
------	-----

doublerequestcount	2m
--------------------	----

查看对应的 rule 已经生成：

```
# oc get rules |grep -i do
```

doubleprom	1m
------------	----

查看 handler, Prometheus handler 在 67 天已经被创建，新创建的 doublehandler 将会在 Prometheus adapter 看到。

```
# oc get handler
```

NAME	AGE
------	-----

prometheus	67d
------------	-----

为 Prometheus 设置端口转发，以便 Prometheus UI 可以访问到我们定义的指标。

```
# oc -n istio-system port-forward $(oc -n istio-system get pod -l app=prometheus -o  
jsonpath='{.items[0].metadata.name}') 9090:9090 &
```

```
[1] 101959
```

```
# Forwarding from 127.0.0.1:9090 -> 9090
```

```
Forwarding from [::1]:9090 -> 9090
```

接下来，对 bookinfo 应用发起访问请求，以便 Prometheus 可以检测到请求。

然后登陆 Prometheus，在 UI 上可以找到我们定义的指标：

istio\_double\_request\_count，如图 9-51 所示：

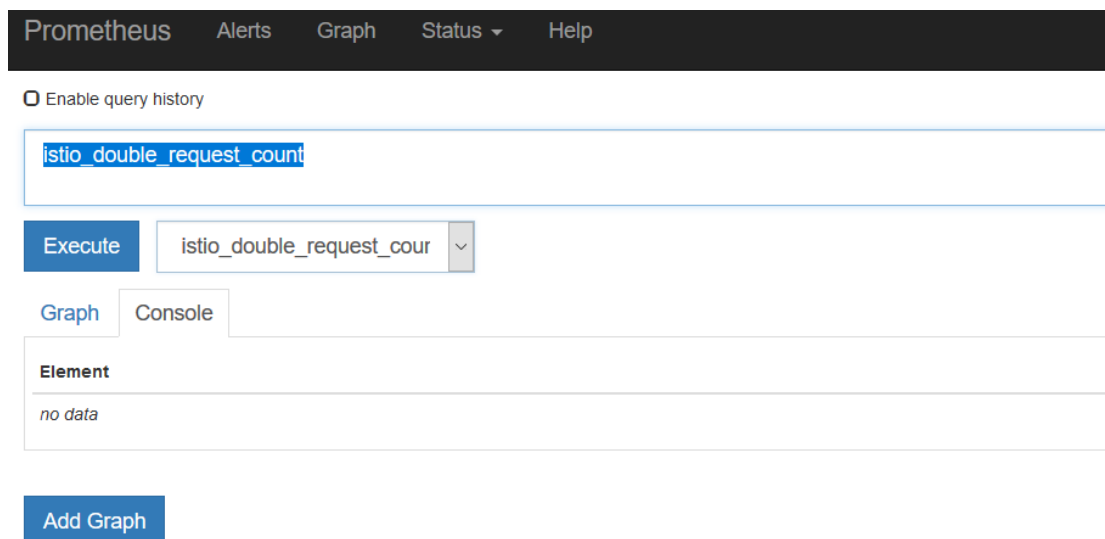


图 9-51: Prometheus UI 查看指标

点击 **execute**，获得监测数据，如下图所示，可以看到我们设置的 **message** 信息：  
**message="twice the fun!"**，如图 9-52 所示：

Element
istio_double_request_count{destination="customer",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="unknown"}
istio_double_request_count{destination="details-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="client",source="productpage-v1"}
istio_double_request_count{destination="details-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="productpage-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="customer"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="details-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="preference-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="productpage-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="ratings-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="recommendation-v1"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="reviews-v3"}
istio_double_request_count{destination="istio-telemetry",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="unknown"}
istio_double_request_count{destination="preference-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="unknown"}
istio_double_request_count{destination="productpage-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="client",source="customgateway"}
istio_double_request_count{destination="productpage-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="customgateway"}
istio_double_request_count{destination="ratings-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="client",source="reviews-v3"}
istio_double_request_count{destination="ratings-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="reviews-v3"}
istio_double_request_count{destination="recommendation-v1",instance="10.129.0.208:42422",job="istio-mesh",message="twice the fun!",reporter="server",source="unknown"}

图 9-52: Prometheus UI 查看日志

登陆 **Grafana**，在 **Prometheus adapter** 中可以看到我们定义的 **doublehandler** 的 **handler**，  
如图 9-53 所示：

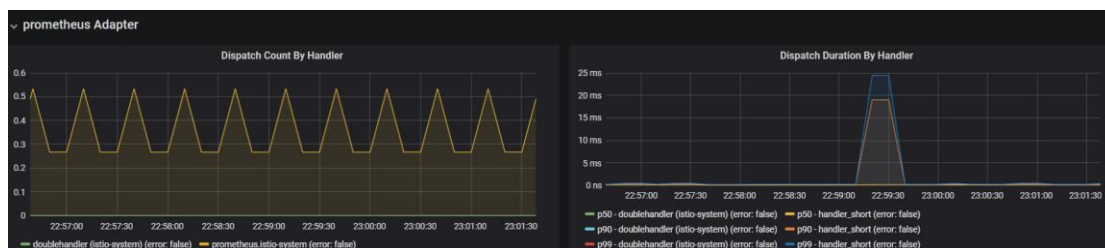


图 9-53: Grafana 查看统计报表

