10.1.1 Spring Cloud 在 OpenShift 上的实现

本节我们以实际的案例对如何在 OpenShift 上实现 Spring Cloud 微服务框架进行说明。

1. 案例场景说明

本案例是一个名为 CoolStore 的电商平台，底层通过 Spring Cloud 在实现，运行在 OpenShift 上。电商平台部署好之后，用户登录平台的 UI，可以购买如帽子、杯子、T-Shirt、眼镜等商品，就如同我们在京东、天猫的购物体验。CoolStore 首页如下图 8-4 所示：

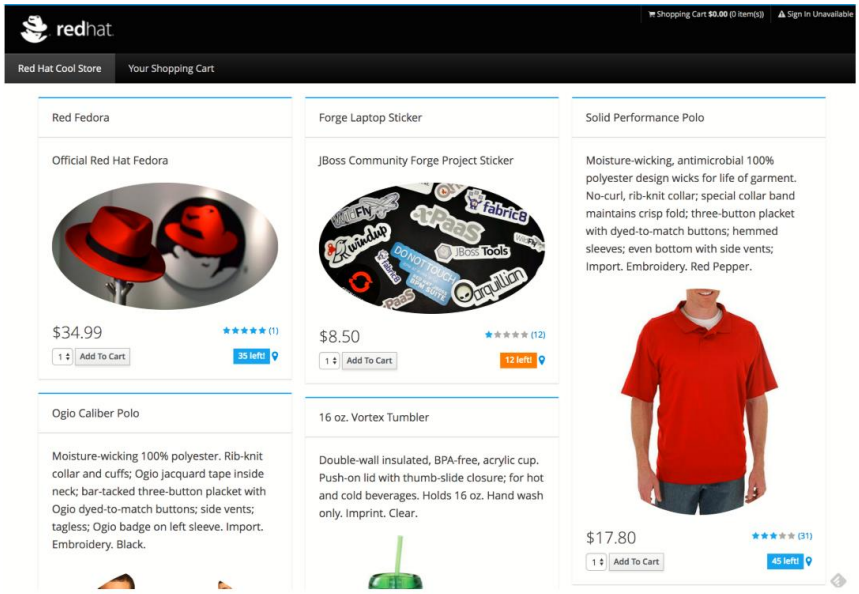

图 8-4 CoolStore 电商首页

CoolStore 电商平台使用的是 Spring Cloud 微服务架构，每一个功能模块都是一个微服务，如下图 8-5 所示：

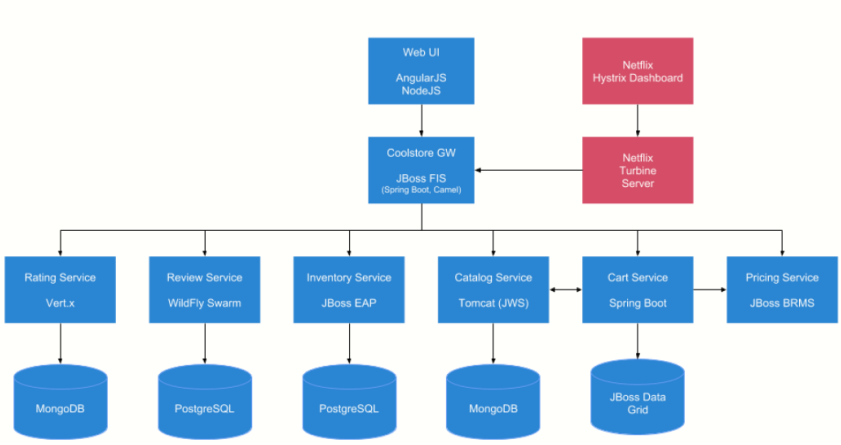

图 8-5 CoolStore 电商微服务架构

微服务的功能描述如下：

- Web UI：在 Node.js 容器中运行的基于 AngularJS 和 PatternFly 的前端。也就是客户访问电商平台的界面展示。
- Catalog Service：目录服务，基于 JBoss Web Server（企业级 Tomcat） 的 Java 应用程序，为零售产品提供产品信息和价格。
- Cart Service：购物车服务，基于的 OpenJDK 运行 Spring Boot 应用程序。
- Inventory Service：库存服务，在 JBoss EAP 7 和 PostgreSQL 上运行的 Java EE 应用程序，为零售产品提供库存和可用性数据。
- Pricing Service：定价服务，使用红帽 JBoss BRMS 产品实现定价业务规则。
- Review Service：审查服务，在 OpenJDK 上运行的 WildFly Swarm 服务，用于撰写和显示产品评论。
- Rating Service：评级服务，在 OpenJDK 上运行的 Vert.x 服务用于评级产品。
- Coolstore API 网关（Coolstore GW）：在 OpenJDK 上运行的 Spring Boot + Camel 应用程序作为后端服务的 API 网关。CoolStore 的 GW 引用了 Hystrix 和 Turbine 做微服务的容错管理。

我们可以看到在上图中，Rating Service、Review Service、Inventory Service、Catelog Service 这四个微服务，都有自己的数据库。这其实正是符合微服务的“share as little as possible”原则。也就是说微服务应该尽量设计成边界清晰不重叠、数据独享不共享，实现所谓的“高内聚、低耦合”。这样有助于实现微服务可独立部署。

2. 在 OpenShift 上部署 CoolStore 微服务

本案例应用源码存放于 Github，地址 https://github.com/ocp-msa-devops/openshift-demos-ansible。

首先将源代码进克隆到本地：

git clone https://github.com/ocp-msa-devops/openshift-demos-ansible.git

cd openshift-demos-ansible

由于我们使用 OpenShift 为 3.11 版本，因此源码使用 OCP-3.11 分支：

git checkout ocp-3.11

批注 [GG1]: 这个还能在 OCP4 上运行吗？

在 Master 节点使用 system:admin 登陆 OpenShift

oc login -u system:admin

新建项目并配置权限：

oc new-project demo-installer

oc adm policy add-cluster-role-to-user cluster-admin system:serviceaccount:demo-installer:default

接下来，用 Ansible 的方式自动化部署 coolstore：

```
# oc new-app -f helpers/coolstore-ansible-installer.yaml \
>       --param=DEMO_NAME=msa-full \
>       --param=PROJECT_ADMIN=developer \
>       --param=COOLSTORE_GITHUB_REF=ocp-3.11 \
>       --param=ANSIBLE_PLAYBOOKS_VERSION=ocp-3.11
--> Deploying template "demo-installer/coolstore-ansible-installer" for "helpers/coolstore-ansible-installer.yaml" to project demo-installer
        * With parameters:
            * Playbooks Git Repo=https://github.com/siamaksade/openshift-demos-ansible
            * Playbooks Git Ref=master
            * CoolStore Demo Name=msa-full
            * CoolStore GitHub Account=jbossdemocentral
```

* CoolStore GitHub Ref=ocp-3.11

* Maven Mirror URL=

* Project Suffix=demo

* Project Admin=developer

* Ephemeral=false

* Deploy Guides=true

* Ansible Extra Vars=

* Ansible Playbooks Version=ocp-3.11

--> Creating resources ...

job.batch "coolstore-ansible-installer" created

--> Success

Run 'oc status' to view your app.

上面命令会生成一个 Pod：coolstore-ansible-installer，Pod 中运行 Ansible 部署整套微服务。

可以通过命令进行监控:

oc logs -f jobs/coolstore-ansible-installer

在部署中，几个主要的微服务会由 S2I 的方式生成，我们以 coolstore-gw 为例，触发的 S2I 构建日志，如下图 8-6 所示：

```
[root@bastion openshift-demos-ansible]# oc logs -f coolstore-gw-12-build
Using registry.redhat.io/jboss-fuse-6/fis-java-openshift@sha256:058581c3d66b8050e683e762f
========================================================================
Starting S2I Java Build .....
S2I source build for Maven detected
Found pom.xml ...
Running 'mvn -Dmaven.repo.local=/tmp/artifacts/m2 package -DskipTests -Dfabric8.skip -e -
Apache Maven 3.3.9 (Red Hat 3.3.9-2.8)
Maven home: /opt/rh/rh-maven33/root/usr/share/maven
Java version: 1.8.0_212, vendor: Oracle Corporation
Java home: /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.212.b04-0.el7_6.x86_64/jre
Default locale: en_US, platform encoding: ANSI_X3.4-1968
OS name: "linux", version: "3.10.0-957.el7.x86_64", arch: "amd64", family: "unix"
[INFO] Error stacktraces are turned on.
[INFO] Scanning for projects...
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/felix/maven-bundle-plu
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/felix/maven-bundle-plu
[INFO] Downloading: https://repo.maven.apache.org/maven2/org/apache/felix/felix-parent/4/
[INFO] Downloaded: https://repo.maven.apache.org/maven2/org/apache/felix/felix-parent/4/f
```

图 8-6 coolstore-gw S2I 构建日志

coolstore-gw docker image 构建成功，镜像推送到 Docker Registry，如下图 8-7 所示：

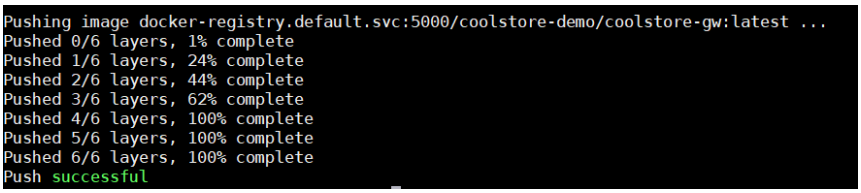

图 8-7 推送 coolstore-gw 到镜像仓库

在 coolstore 部署过程中，共进行了 8 个构建。

oc get pods

NAME	READY	STATUS	RESTARTS	AGE
cart-1-build	0/1	Completed	0	1h
catalog-1-build	0/1	Completed	0	1h
coolstore-gw-12-build	0/1	Completed	0	1h
inventory-1-build	0/1	Completed	0	1h
pricing-1-build	0/1	Completed	0	1h
rating-1-build	0/1	Completed	0	58m
review-1-build	0/1	Completed	0	58m
web-ui-1-build	0/1	Completed	0	58m

构建完成后，OpenShift 自动触发部署，最终创建如下 Pods。

oc get pods

cart-1-cs2m4	1/1	Running	0	11h
catalog-1-brn9c	1/1	Running	0	11h
catalog-mongodb-1-2dmbg	1/1	Running	0	11h
coolstore-gw-1-stxrv	1/1	Running	0	11h
datagrid-3-vw7v9	1/1	Running	0	10h
hystrix-dashboard-1-ftbmb	1/1	Running	0	11h
nexus-2-fm9nr	1/1	Running	0	11h
pricing-1-xg7hb	1/1	Running	0	11h
rating-1-jfg8l	1/1	Running	0	11h
rating-mongodb-2-mxlgh	1/1	Running	0	11h
review-3-zhxl9	1/1	Running	11	10h
review-postgresql-2-hkmk9	1/1	Running	0	10h

turbine-server-1-59zz8 1/1 Running 1 11h

web-ui-1-bqkgq 1/1 Running 0 11h

可以看到，上面的 Pod 中并没有原生 Spring Cloud 的服务注册发现、配置中心、微服务网关。

每一个微服务，在 OpenShift 中都创建了 Service。微服务之间的内部通讯是通过 Service 实现的。

oc get svc

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

cart ClusterIP 172.30.214.190 <none> 8080/TCP 3m

catalog ClusterIP 172.30.95.218 <none> 8080/TCP 3m

catalog-mongodb ClusterIP 172.30.224.162 <none> 27017/TCP 3m

coolstore-gw ClusterIP 172.30.50.86 <none> 8080/TCP 3m

datagrid-hotrod ClusterIP 172.30.96.239 <none> 11333/TCP 3m

hystrix-dashboard ClusterIP 172.30.182.40 <none> 8080/TCP 3m

inventory ClusterIP 172.30.172.95 <none> 8080/TCP 3m

inventory-postgresql ClusterIP 172.30.103.24 <none> 5432/TCP 3m

pricing ClusterIP 172.30.188.145 <none> 8080/TCP 3m

rating ClusterIP 172.30.69.167 <none> 8080/TCP 3m

rating-mongodb ClusterIP 172.30.242.109 <none> 27017/TCP 3m

review ClusterIP 172.30.223.184 <none> 8080/TCP 3m

review-postgresql ClusterIP 172.30.128.153 <none> 5432/TCP 3m

turbine-server ClusterIP 172.30.227.119 <none> 80/TCP 3m

web-ui ClusterIP 172.30.5.76 <none> 8080/TCP 3m

查看路由，我们看到除了 UI 以外，很多微服务也在 Router 上创建了路由。

oc get route

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

cart cart-coolstore-demo.apps.beijing-f671.openshiftworkshop.com cart <all> None

catalog catalog-coolstore-demo.apps.beijing-f671.openshiftworkshop.com catalog <all> None

coolstore-gw gw-coolstore-demo.apps.beijing-f671.openshiftworkshop.com coolstore-gw <all> None

hystrix-dashboard hystrix-dashboard-coolstore-demo.apps.beijing-f671.openshiftworkshop.com hystrix-dashboard 8080-tcp None

inventory inventory-coolstore-demo.apps.beijing-f671.openshiftworkshop.com inventory <all> None

pricing pricing-coolstore-demo.apps.beijing-f671.openshiftworkshop.com pricing <all> None

rating rating-coolstore-demo.apps.beijing-f671.openshiftworkshop.com rating <all> None

review review-coolstore-demo.apps.beijing-f671.openshiftworkshop.com review <all> None

turbine-server turbine-server-coolstore-demo.apps.beijing-

f671.openshiftworkshop.com turbine-server 80-tcp None

web-ui web-ui-coolstore-demo.apps.beijing-f671.openshiftworkshop.com web-ui <all> None

上面的路由中，除了 UI 是为了用户直接访问、Hystrix 是为管理员直接访问之外，其他微服务路由指向了其微服务的 Rest API，目的是方便我们学习和理解。真实生产环境中，不需要创建除了 UI 和 Hystrix 之外的路由。

下图我们以访问 Catalog 微服务的路由为例，最终访问是的它的 Rest API，如下图 8-8 所示：

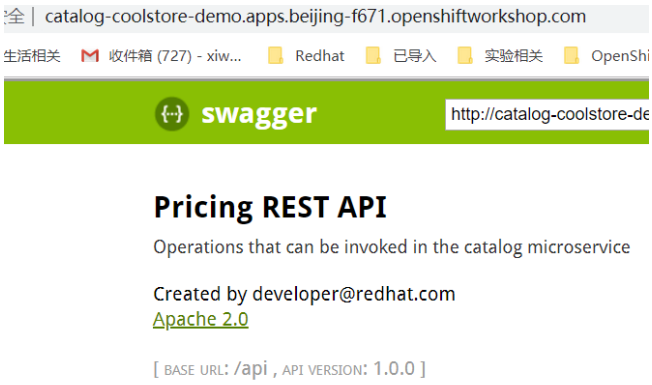

图 8-8 Catalog 服务 API

CoolStore 微服务部署成功后，我们通过浏览器访问 Web-UI 的路由，查看微服务的效果。

3. 微服务的效果展示

浏览器输入 Web-UI 的路由，登录 CoolStore 首页。我们可以看到，页面中有很多商品。每个商品都有对应的价格、评级、库存情况，如下图 8-9 所示：

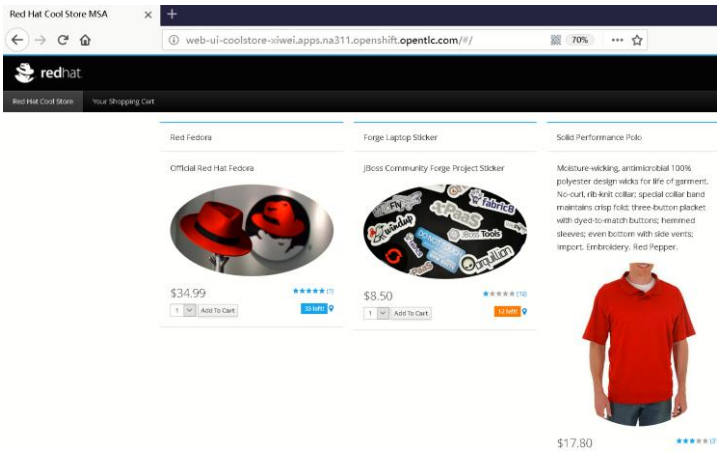

图 8-9 CoolStore 物品展示

我们以 Fedora 帽子为例，如下图 8-10 所示，它的价格是 34.99 美元，库存数量为 35。

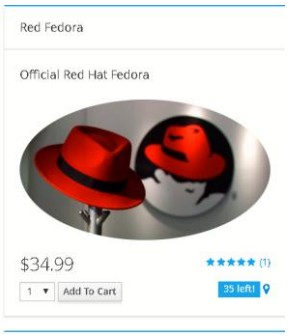

图 8-10 Fedora 帽子的价格和库存

我们点击商品的库存的位置，如图 8-11 所示，会调用 Google API，在地图上显示库存的位置。

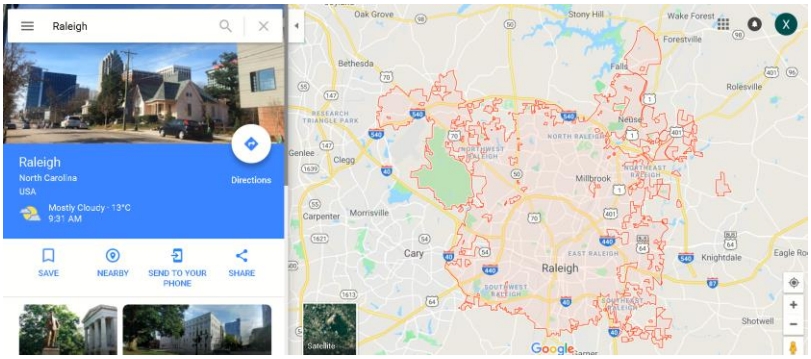

图 8-11 Fedora 帽子的库存位置

我们查看 Fedora 帽子的评论，如图 8-12 所示：

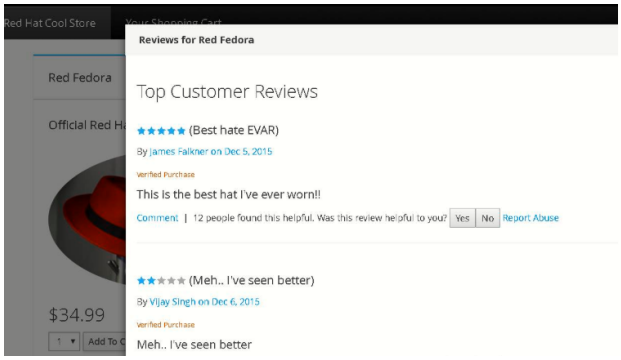

图 8-12 Fedora 帽子的购买评论

选择购买五个，加入到购物车，可以看到购物车中显示选中了五个帽子，如图 8-13 所示：

图 8-13 Fedora 帽子被加入购物车

我们将 Web-UI 的 Pod 数量增加到 2 个，然后再次通过 Web-UI 的路由访问，请求将会负载到两个 Pod 上。

4. CoolStore 微服务之间调用的实现

CoolStore 中每个微服务之间都是通过 API 方式进行调用的。在 CoolStore 中，API 网关（coolstore-gw）使用的是 Apache Camel 的 Java DSL 的模式。访问 coolstor-gw 可以看到有很多路由条目，如下图 8-16 所示：

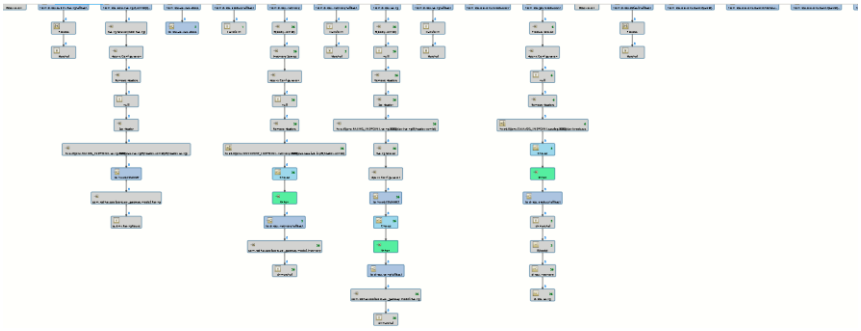

图 8-16 coolstore-gw Pod 的路由条目

这些路由定义了微服务之间的调用。对应的源码文件，如下图 8-17 所示：

图 8-17 路由的 Java 文件

我们访问 CoolStore 网关的 API。我们可以看到网关和 rating、catalog、cart、review 等微服务的调用关系。我们以 ProductGateway.java 为例，由于篇幅有限，只展示部分代码。

第一段定义了从 productFallback 的返回异常的路由。

```
        from("direct:productFallback")
                .id("ProductFallbackRoute")
                .transform()
                .constant(Collections.singletonList(new Product("0",
"Unavailable Product", "Unavailable Product", 0, null)));
                //.marshal().json(JsonLibrary.Jackson, List.class);
```

第二段定义了：两条路由

1. 定义了正常情况下，从 inventory 到 API 为 http4://{{env:INVENTORY_ENDPOINT:inventory:8080}}/api/availability/${header.itemId}")).end()的路由。路由中定义了对 hystrix 的引用。
2. 定义了异常情况下，即当请求中的产品 ID 号为空时，从 inventory 到 inventoryFallback，结果是返回：new Product("0", "Unavailable Product", "Unavailable Product", 0, null))

```
      from("direct:inventory")
            .id("inventoryRoute")
            .setHeader("itemId", simple("${body.itemId}"))
            .hystrix().id("Inventory Service")
               .hystrixConfiguration()
                  .executionTimeoutInMilliseconds(hystrixExecutionTime
out)
                  .groupKey(hystrixGroupKey)
                  .circuitBreakerEnabled(hystrixCircuitBreakerEnabled)
               .end()
               .setBody(simple("null"))
               .removeHeaders("CamelHttp*")
               .recipientList(simple("http4://{{env:INVENTORY_ENDPOINT
:inventory:8080}}/api/availability/${header.itemId}")).end()
            .onFallback()
               //.setHeader(Exchange.HTTP_RESPONSE_CODE,
constant(Response.Status.SERVICE_UNAVAILABLE.getStatusCode()))
               .to("direct:inventoryFallback")
            .end()
            .choice().when(body().isNull())
               .to("direct:inventoryFallback")
            .end()
```

```
            .setHeader("CamelJacksonUnmarshalType",
simple(Inventory.class.getName()))
            .unmarshal().json(JsonLibrary.Jackson, Inventory.class);
```

接下来，我们使用浏览器，访问 API 网关的路由地址，查看 API 之间的调用，如下图 8-18 所示：

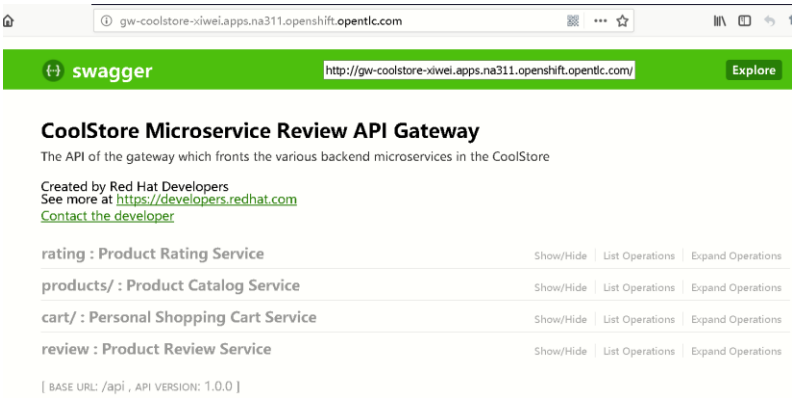

图 8-18 微服务之间 API 调用

我们可以看到，API 网关会调用其他四个微服务：rating、products、cart、review。在页面上点击查看 rating 微服务的 API，如下图 8-19 所示：

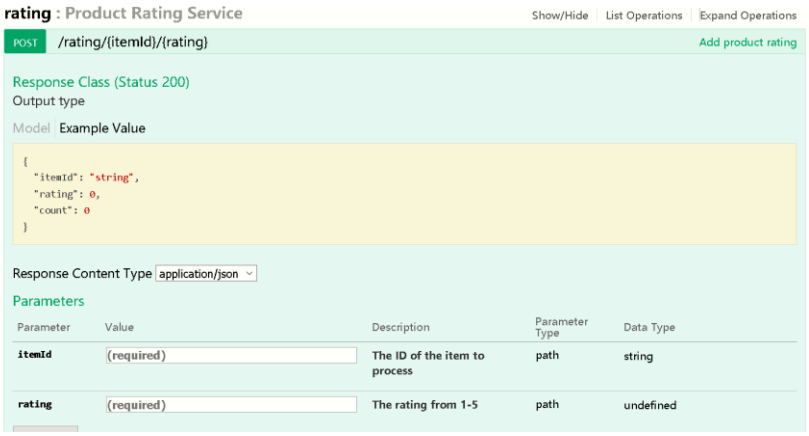

图 8-19 rating 微服务的 API

在页面上点击查看 catalog 微服务的 API，如下图 8-20 所示：

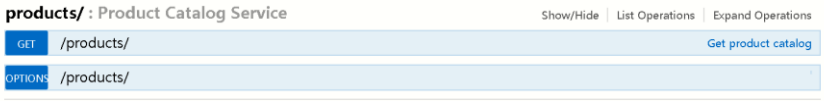

图 8-20 catalog 微服务的 API

我们调用 products API 的第一个端点，点击“Try it out！”，如下图 8-21 所示：

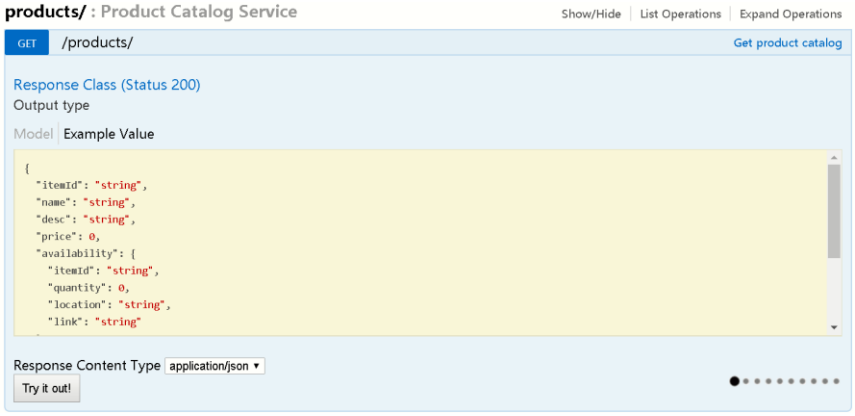

图 8-21 product 微服务的 API

Response Body 表示的是返回结果，如图 8-22 展示的 desc 为 Offcial Red Hat Fedora，就是电商首页展示的红帽子商品，返回值标明了帽子的价格、库存数量、库存地址等信息。

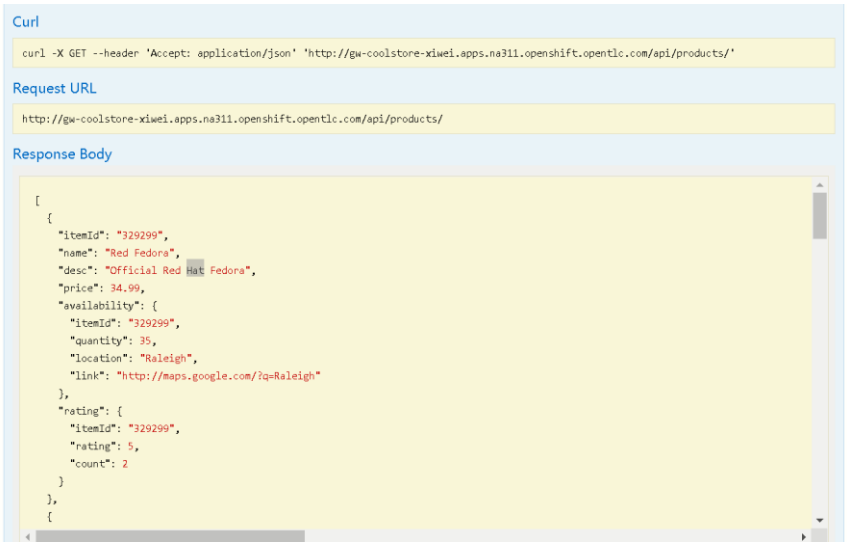

图 8-22 product 微服务的 API 调用结果

cart 微服务的 API 中的端点有很多，如图 8-23 所示：

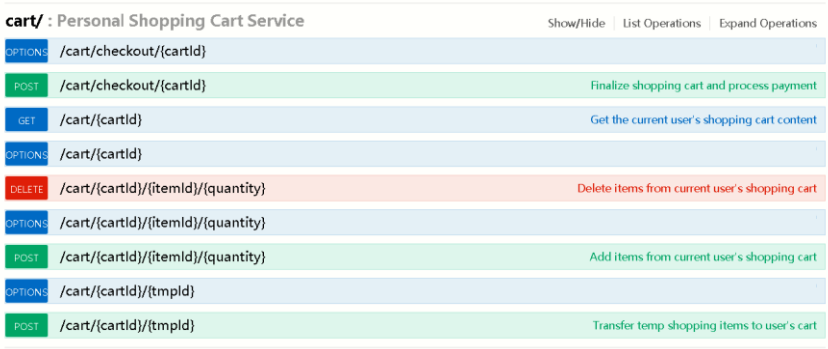

图 8-23 cart 微服务的 API 端点

查看 inventory 微服务的 API，如下图 8-24 所示：

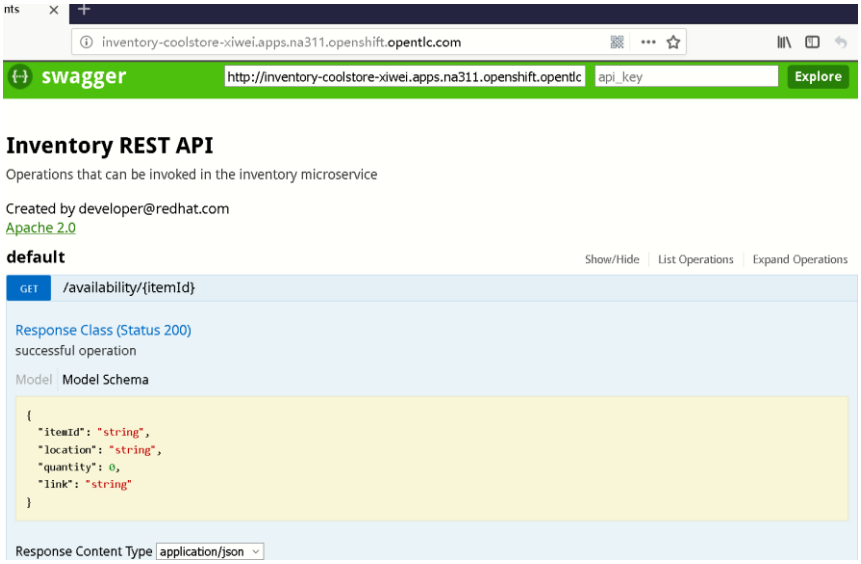

图 8-24 inventory 微服务的 API 端点

我们手工验证 API。在 itemId 中，输入红帽子的产品 id 号，然后点击 Try it out!，如下图 8-25 所示：

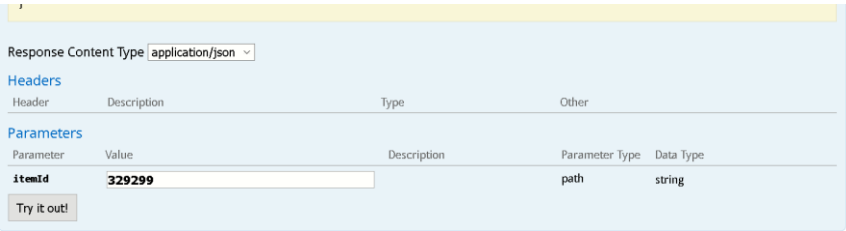

图 8-25. inventory 微服务的 API 调用

查看返回结果，可以获取到产品的库存数量（35 个），如下图 8-26 所示：

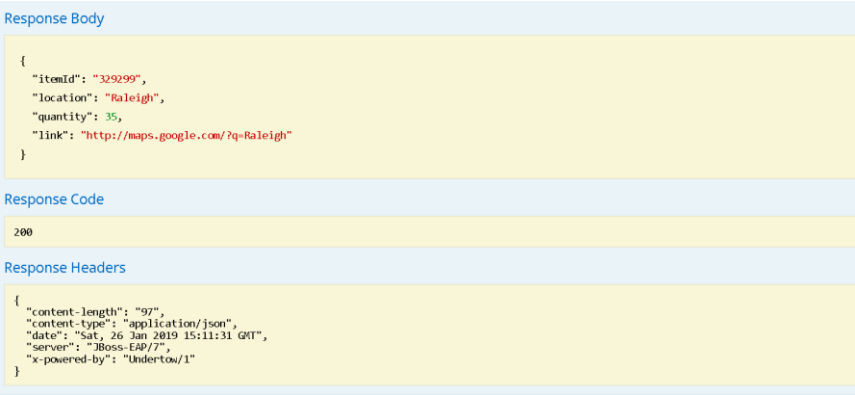

图 8-26 inventory 微服务的 API 调用

在 OpenShift 中使用 ConfigMap 实现配置中心，下面我们查看 CoolStore 的配置中心。

查看 OpenShift 中 coolstore 项目中的 ConfigMap：

oc get cm

NAME DATA AGE

rating-config 1 24m

review-config 1 24m

有两个 ConfigMap，分别是给 rating 和 review 两个微服务注入配置的。

查看 rating-config 的内容，它为 rating 微服务注入了访问 MongoDB 的用户名、密码、路径等。

oc describe cm rating-config

Name: rating-config

Namespace: coolstore-demo

Labels: app=rating

demo=coolstore-microservice

Annotations: <none>

Data

====

rating-config.yaml:

---

rating.http.port: 8080

connection_string: mongodb://rating-mongodb:27017

db_name: ratingdb

username: user4WK

password: ckB6gEsm

Events: <none>

从本小节我们可以看出，微服务之间的路由关系是通过 Camel 实现的，在书写微服务调用时，使用 OpenShift 的 Service，调用通过 Rest API 的方式。

接下来，我们介绍 CoolStore 的容错。

5. CoolStore 的容错

在微服务中，Hytrix 是针对微服务调用的源端生效，而非目标端生效。

Hystrix 和熔断相关的常用几个参数如下：

- circuitBreaker.enabled：设置断路器是否起作用。

默认值：true

默认属性：hystrix.command.default.circuitBreaker.enabled

实例属性：hystrix.command.HystrixCommandKey.circuitBreaker.enabled

实例默认的设置：HystrixCommandProperties.Setter().withCircuitBreakerEnabled(boolean value)

- circuitBreaker.requestVolumeThreshold：设置在一个滚动窗口中，打开断路器的最少请求数。比如：如果值是 20，在一个窗口内（比如 10 秒），收到 19 个请求，即使这 19 个请求都失败了，断路器也不会打开。

默认值：20

默认属性：hystrix.command.default.circuitBreaker.requestVolumeThreshold

实例属性：hystrix.command.HystrixCommandKey.circuitBreaker.requestVolumeThreshold

实例默认的设置：

HystrixCommandProperties.Setter().withCircuitBreakerRequestVolumeThreshold(int value)

- circuitBreaker.sleepWindowInMilliseconds：设置在回路被打开，拒绝请求到再次尝试请求并决定回路是否继续打开的时间。

默认值：5000（毫秒）

默认属性：hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds

实例属性：

hystrix.command.HystrixCommandKey.circuitBreaker.sleepWindowInMilliseconds

实例默认的设置：

HystrixCommandProperties.Setter().withCircuitBreakerSleepWindowInMilliseconds(int value)

- circuitBreaker.errorThresholdPercentage：设置打开回路并启动回退逻辑的错误比率。如果错误率>=该值，circuit 会被打开，并短路所有请求触发 fallback。

默认值：50

默认属性：hystrix.command.default.circuitBreaker.errorThresholdPercentage

实例属性：hystrix.command.HystrixCommandKey.circuitBreaker.errorThresholdPercentage

实例默认的设置：

HystrixCommandProperties.Setter().withCircuitBreakerErrorThresholdPercentage(int value)

微服务对 Hystrix 的使用，主要有如下两种方式：

1. SpringBoot 使用 annotation 的方式，如下面的这段代码：

```
   @SpringBootApplication
   @EnableCircuitBreaker
   public class ApiServiceApplication {


      public static void main(String[] args) {
         SpringApplication app = new
SpringApplication(ApiServiceApplication.class);
```

```
        app.run(args);
    }


}
```

2. 在 Apache Camel 提供 Java DSL 中使用 Hystrix EIP，如下面这段代码：

```
from("direct:start")
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(5000)
            .circuitBreakerSleepWindowInMilliseconds(10000)
        .end()
        .to("http://fooservice.com/slow")
    .onFallback()
        .transform().constant("Fallback message")
    .end()
    .to("mock:result");
```

CoolStore 微服务对 Hystrix 的使用，采用的是上述第二种方式。即是在 coolstore-gw，使用 Camel Java DSL 方式实现。

通过 IDE 工具，import CoolStore 的源码，可以看到 API_gateway 有多个 Java 的类，也就是微服务的类，查看 ReviewGateway.java，可以看到 Review 微服务启用了 Hystrix，如下图 8-27 所示：

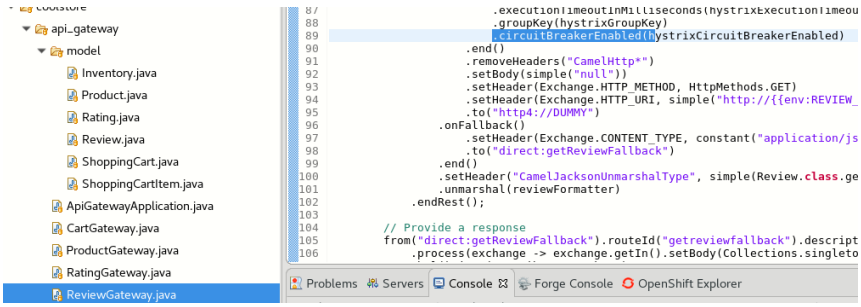

图 8-27 ReviewGateway.java 源代码启动 Hystrix

Hystrix 参数的设置，是通过另外一个配置文件传递进去的，如下图 8-28 所示：

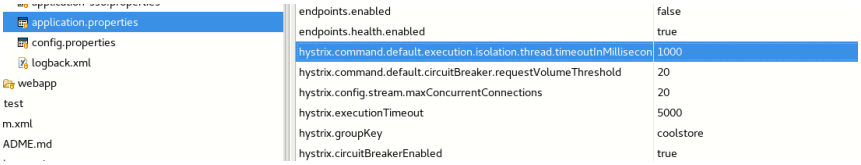

图 8-28 Hystrix 的参数设置

将 Hystrix 的参数通过配置文件传递的好处是显而易见的，否则的话，如果我们想调整参数，需要修改源码并进行重新编译（Java 系需要编译）。这无疑增加了开发人员的工作量。

通过浏览器访问 Hytrix Dashboard 的路由，可以看到每个微服务断路器的情况，都是关闭的。即所有的微服务模块都是正常工作，如下图 8-29 所示：

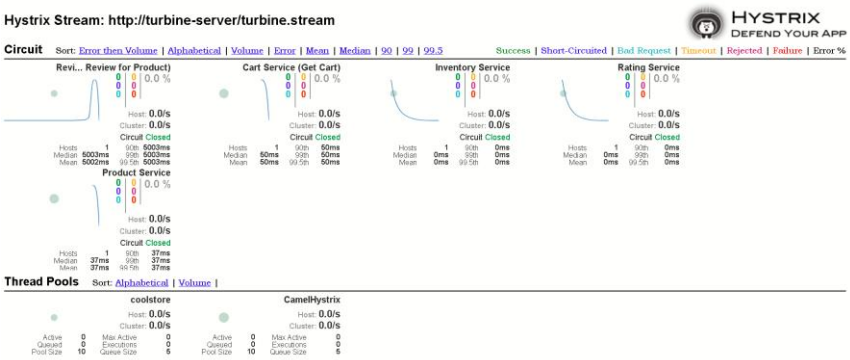

图 8-29 Hystrix 的界面展示

接下来，我们在 Review 微服务中，制造一些代码故障，然后再对 CoolStore 的 Web-UI 发起大量请求。Review 微服务的 Pod 出现了问题，如下图 8-30 所示：

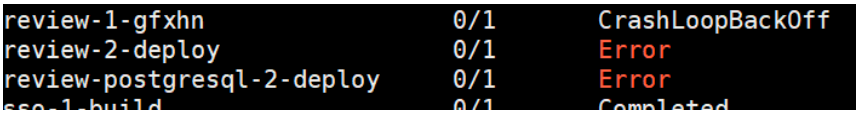

图 8-30 review 微服务状态展示

Hytrix 很快检测到了 Review 微服务的错误（错误率 100%），但由于没有到阈值，因此断路器并未打开，如下图 8-31 所示：

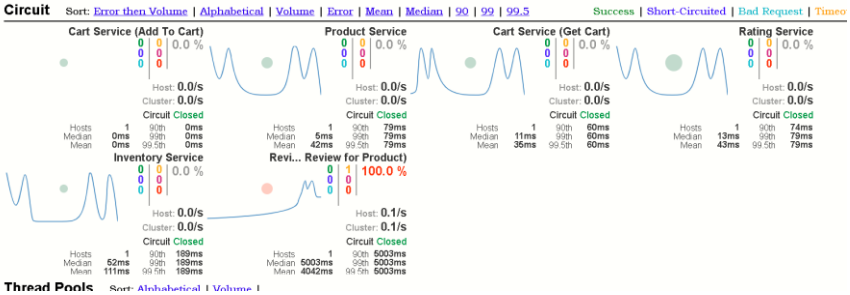

图 8-31 Hytrix 界面展示 1

随着访问量的持续，review 微服务响应时间的增加，review 的断路器被打开，如下图 8-32 和 8-33 所示（红字显示 Open）：

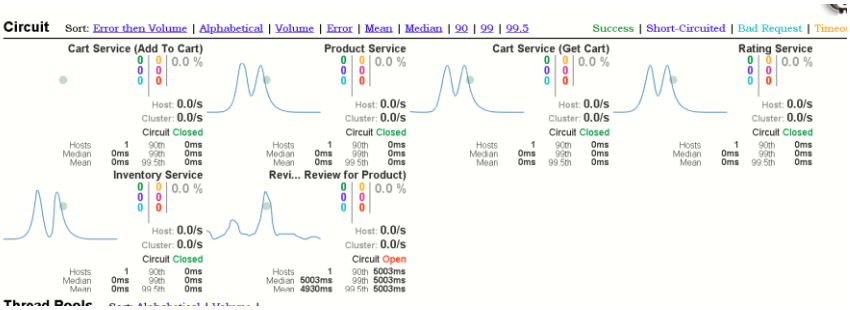

图 8-32 Hytrix 界面展示 2

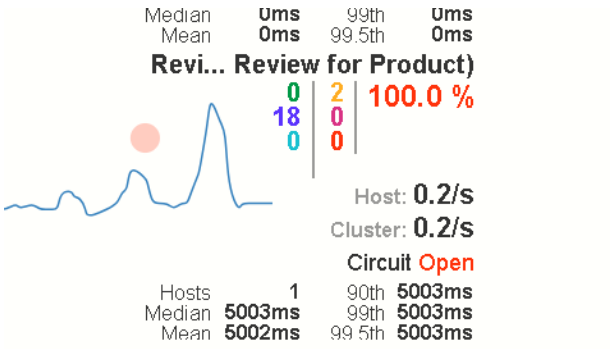

图 8-33 Hytrix 界面展示 3

此时，再访问网站时，除了 Review 无法查看，其余功能组件仍然正常工作，如下图 8-34 所示：

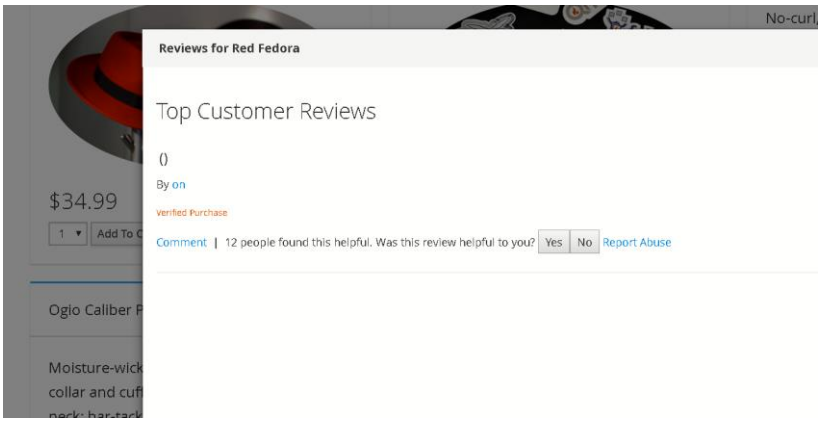

图 8-34 Review 微服务出现问题

6. Coolstore 的日志监控

OpenShift 中集成的 EFK 组件，我们通过命令行查看。

```
# oc get pods -n openshift-logging
NAME                                      READY     STATUS
RESTARTS   AGE
logging-es-data-master-h7z262ej-1-vgxs6   2/2       Running   0          3h
logging-fluentd-6lfps                     1/1       Running   0          3h
logging-fluentd-vn8dt                     1/1       Running   0          3h
logging-fluentd-x67fr                     1/1       Running   0          3h
logging-kibana-1-vn267                    2/2       Running   0          3h
```

OpenShift 也集成了性能监控的组件：

```
# oc get pods -n openshift-monitoring
NAME                                      READY     STATUS
RESTARTS   AGE
alertmanager-main-0                       3/3       Running   0
3h
alertmanager-main-1                       3/3       Running   0
```

3h

alertmanager-main-2 3/3 Running 0

3h

cluster-monitoring-operator-5d4f4c9c89-hhwpr 1/1 Running 0 3h

grafana-6b4ccb4b45-xd29w 2/2 Running 0

3h

kube-state-metrics-84f7c5cdc9-8wbk6 3/3 Running 0

3h

node-exporter-lfq7p 2/2 Running 0

3h

node-exporter-n6rh5 2/2 Running 0

3h

node-exporter-vnxzj 2/2 Running 0

3h

prometheus-k8s-0 4/4 Running 1

3h

prometheus-k8s-1 4/4 Running 1

3h

prometheus-operator-7bbc685dd9-dgfjk 1/1 Running 0

3h

我们通过 OpenShift 管理界面查看收集 coolstore 的 Event，如下图 8-35 所示：

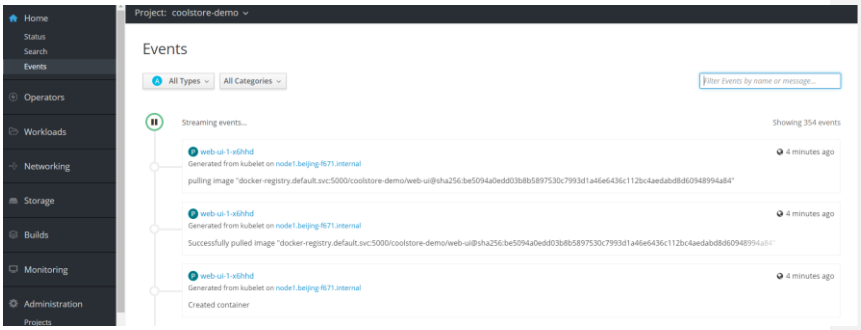

图 8-35 查看 event1

可以选择事件的类型级别，我们选择 DC，可以看到 8 分钟前，第一条信息是：Web-UI 的 Pod 数量从一个扩容到两个，如下图 8-36 所示：

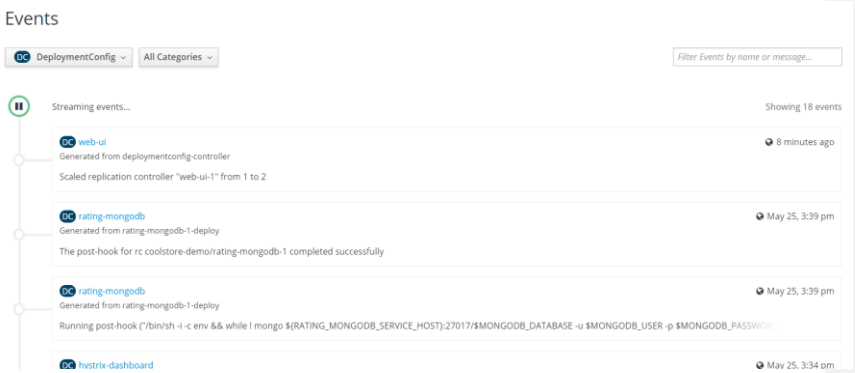

图 8-36 查看 event2

日志展现通过 Kibana 实现，可以根据关键词进行搜索，如下图 8-37 所示：

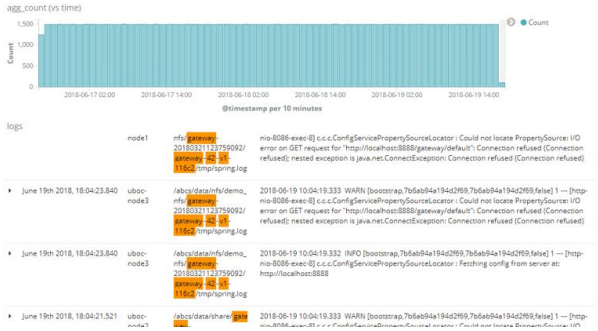

图 8-37 查看日志

在监控部分，红帽集成了 Prometheus 可以收集 CoolStore 的实时性能信息，并在 Grafana 上做统一展现。例如我们查看 CoolStore Pod 的资源利用率，如下图 8-38 所示：

图 8-38 CoolStore 资源利用率

也可以更为细致地查看某个 Pod 的具体性能信息，如下图 8-39 所示：

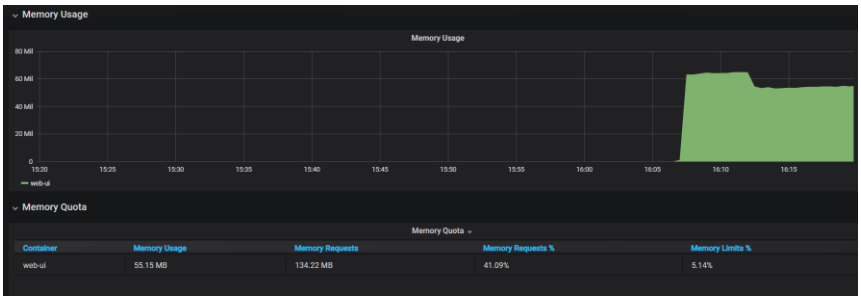

图 8-39 某个 Pod 的内存使用性能曲线

我们也可以登录 Prometheus，针对具体的探测点记性查看，如下图 8-40 所示：

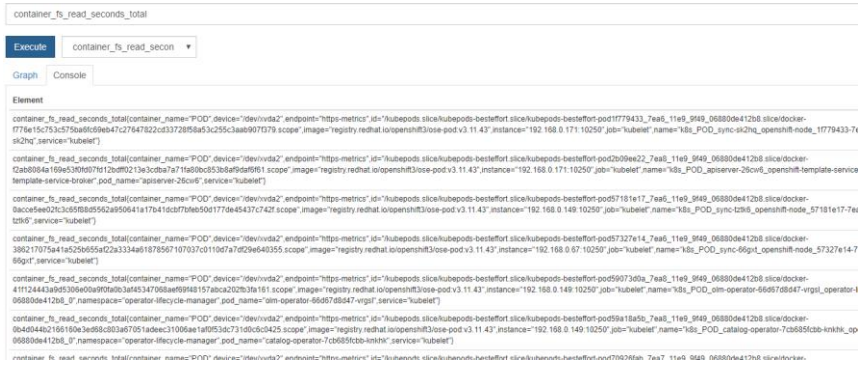

图 8-40 Prometheus 性能指标

可以生成趋势图展示，如下图 8-41 所示：

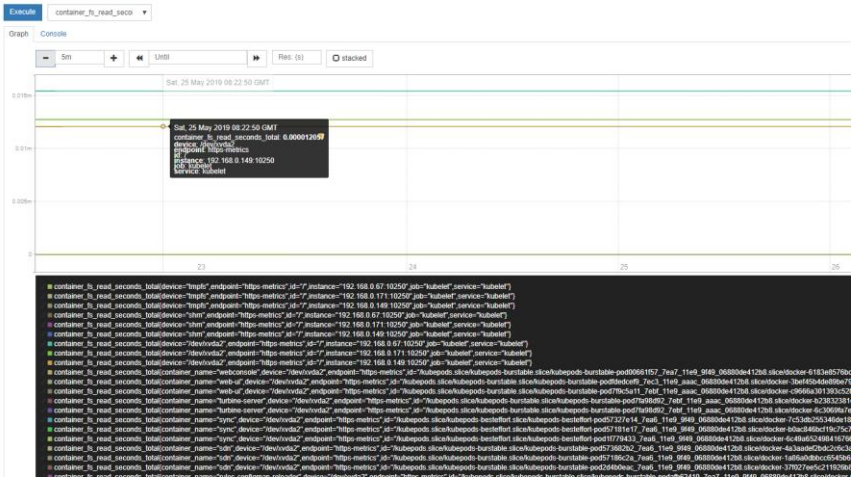

图 8-42 Prometheus 图形

我们可以看到，通过 OpenShift 的原生工具，就可以实现对 Spring Cloud 微服务的日志和监控，十分便捷。

7. CoolStore 展示总结

从 CoolStore 的展示中，我们验证了基于 OpenShift 实现 Spring Cloud 微服务方式：

- 注册发现由 OpenShift Etcd、Service 和内部 DNS 实现。
- 配置中心由 OpenShift ConfigMap 实现。
- 微服务网关由 Camel 实现。
- 入口流量由 OpenShift Router 实现。
- 微服务之间的项目隔离通过 OpenShift Project 实现。
- 日志监控由 OpenShift 集成工具实现。
- 熔断由 Hytrix 实现。

通过 CoolStore 这个案例，我们可以大致了解微服务的工作模式以及 Spring Cloud 的一些特性。从源码角度，CoolStore 的开发人员在书写代码的时候，需要考虑到微服务之间的调用关系。如果修改调用关系，也需要重新编译应用。也就是说，应用的开发人员不仅要关注应用本身，还需要关心微服务之间的路由和调用关系。

在本节中，我们介绍了微服务的概念以及几种微服务的架构。通过一个电商的案例，我们能够了解到 Spring Cloud 在 OpenShift 上落地的方式，也能够得出结论：基于

OpenShift 的 Spring Cloud 其功能性和可维护性都要高于原生 Spring Cloud。

接下来，我们将开始介绍新一代微服务架构 Istio。