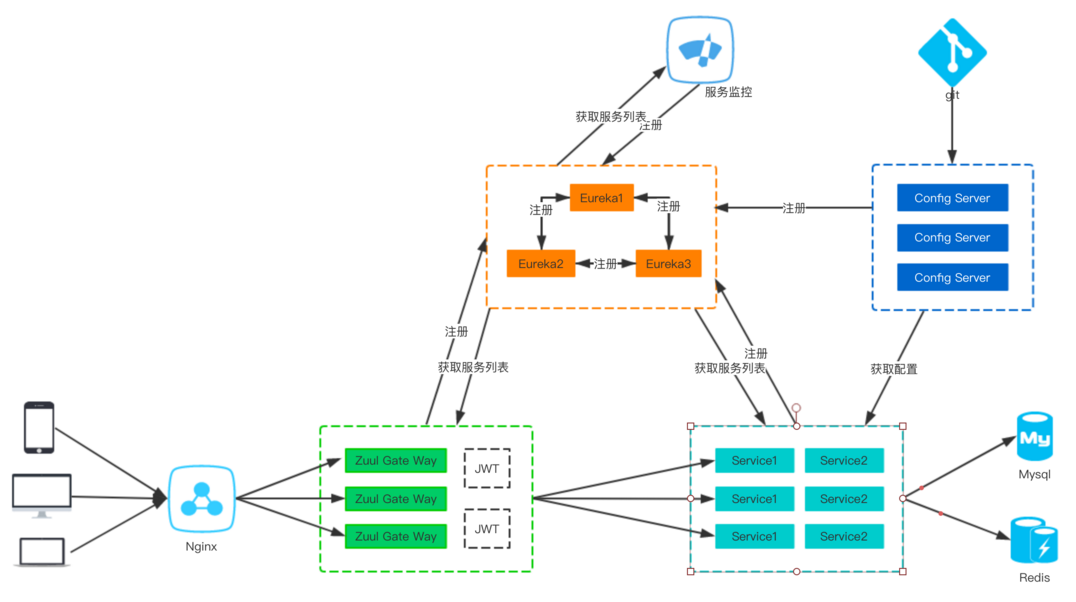
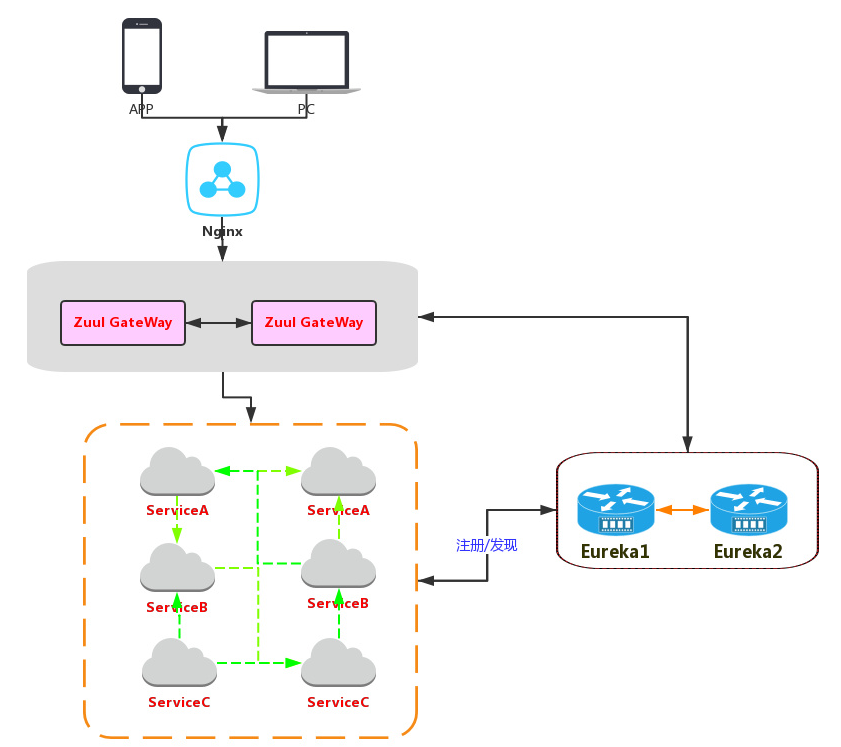
**SpringCloud**





**目录**

1、Eureka

1.1 eureka单节点搭建

1.2 eureka多节点搭建

2、Ribbon

3、Hystrix

4、Feign

5、Zuul

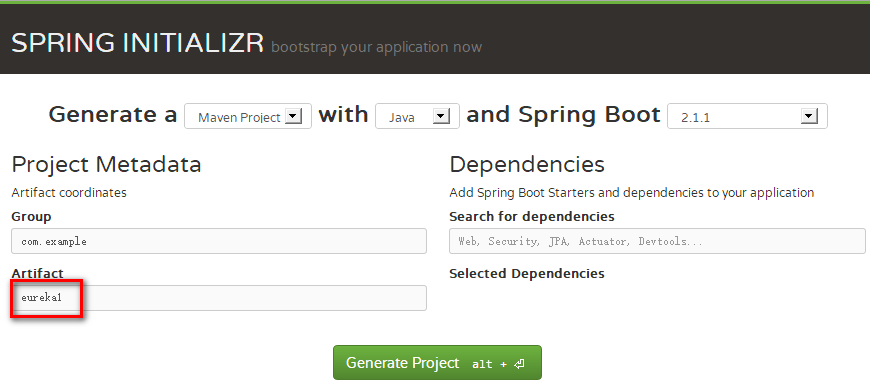
6、参数配置

1. **Eureka**

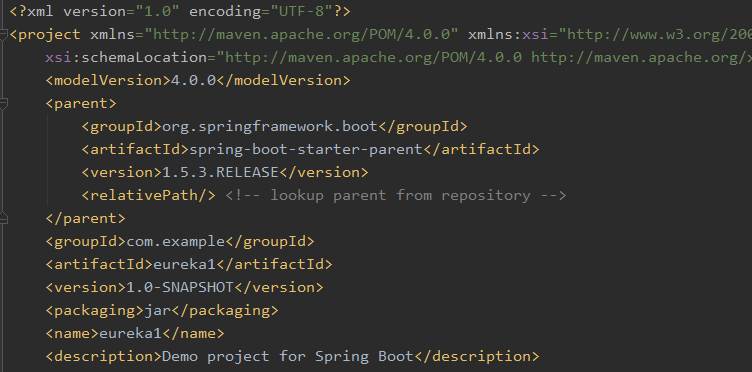
服务治理模块，用来实现各个微服务实例的自动化注册与发现。监控和维护可用服务列表，对外提供可用服务列表清单。

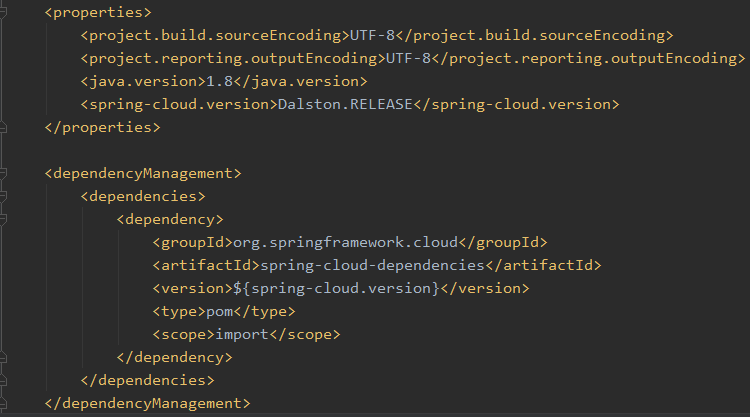
**1.1 eureka单节点搭建**

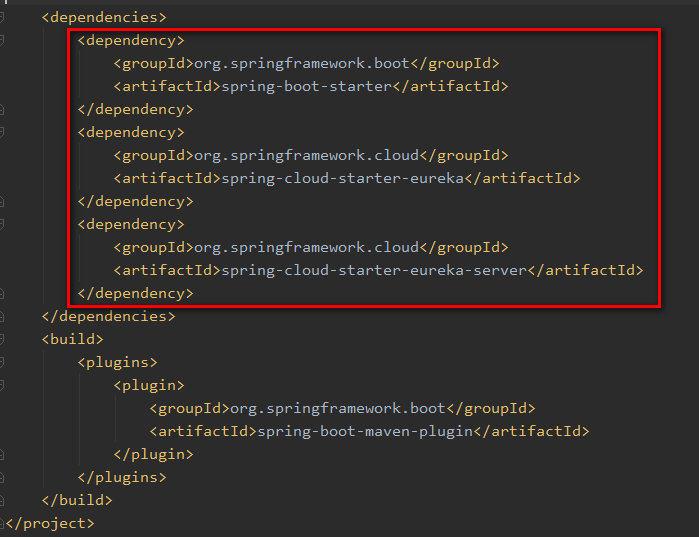
访问 <https://start.spring.io/>下载模板



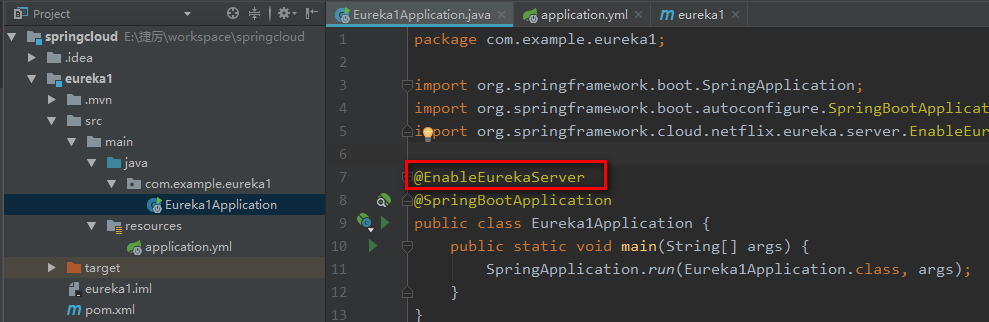
pom文件配置



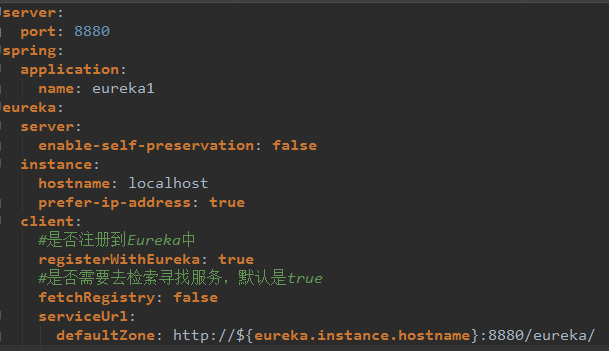




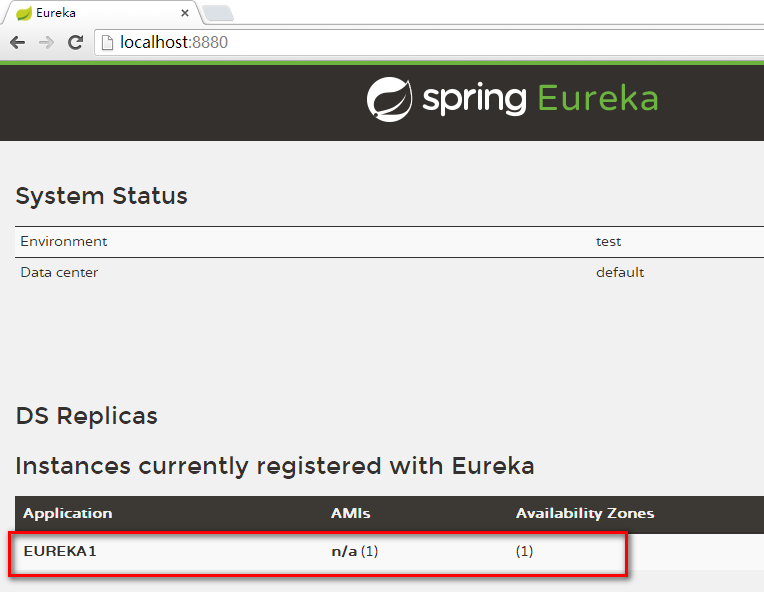
配置启动类



配置application.yml



启动项目，访问localhost:8880



配置参数说明：

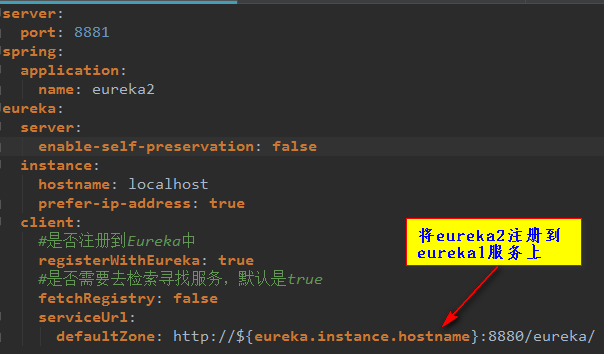
**eureka.client.registerWithEureka**：是否向注册中心注册自己。

**eureka.client.fetch-registry**：是否需要去检索查找服务，由于注册中心的职责就是维护服务列表，所以这里并不需要去检索服务，配置false。

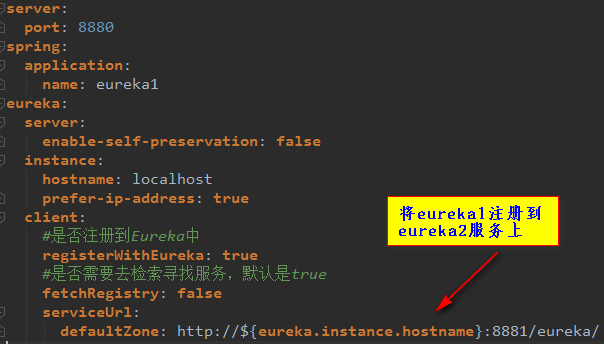
**1.2 eureka多节点搭建**

单节点注册中心风险较大，一旦发生故障，导致整个服务瘫痪，为增强系统的可用性，这里部署多节点，节点之间相互注册，这样每个节点都维护一份服务列表清单，一个节点发生故障不会影响服务的正常运行。

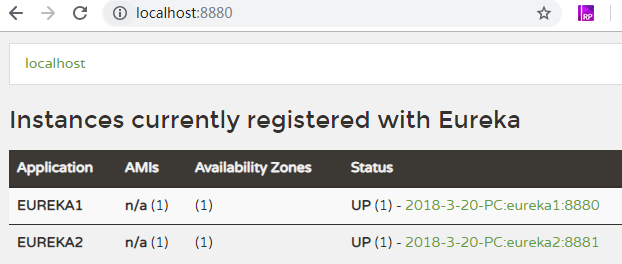
新建项目eureka2，把eureka1代码复制过来，更改application.yml配置，如下图：

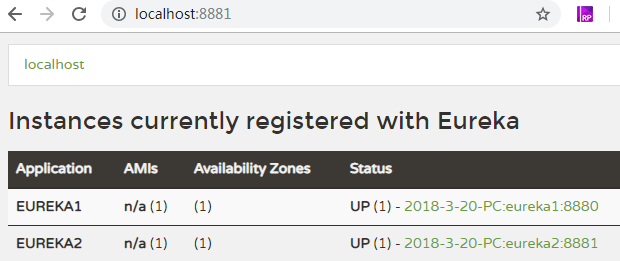


更改eureka1的application.yml配置，如下图：



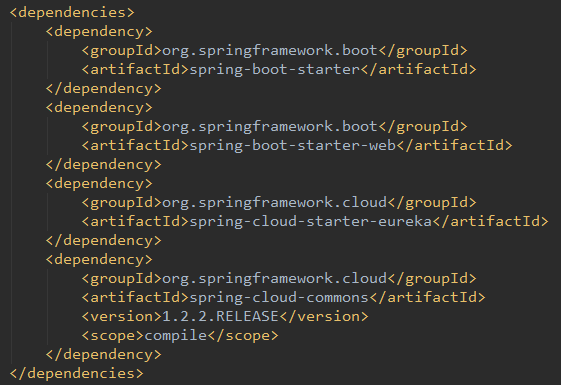
访问eureka地址，可以看到两个eureka服务正常，并且相互注册成功。





新建product服务（服务提供者）测试注册到eureka中，product服务配置如下：

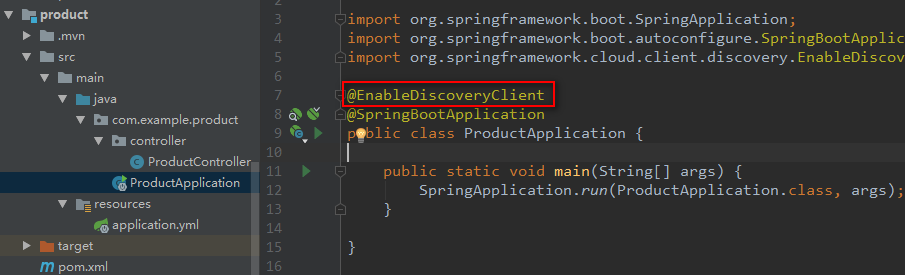
Pom配置：



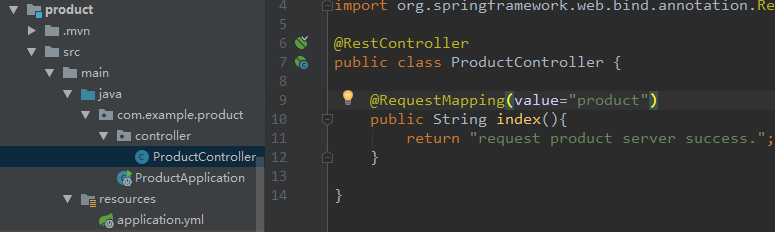
application.yml配置：

**server**:  
 **port**: 8890  
**spring**:  
 **application**:  
 **name**: product  
**eureka**:  
 **instance**:  
 **hostname**: localhost  
 **prefer-ip-address**: **true  
 client**:  
 *#是否注册到Eureka中* **registerWithEureka**: **true** *#是否需要去检索寻找服务，默认是true* **fetchRegistry**: **true  
 serviceUrl**:  
 **defaultZone**: http://${**eureka.instance.hostname**}:8880/eureka/,http://${**eureka.instance.hostname**}:8881/eureka/

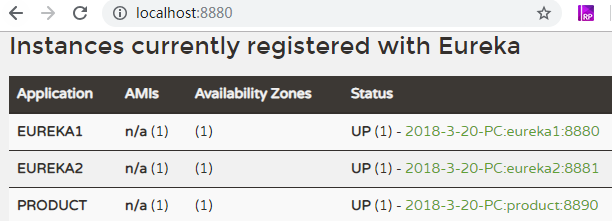
启动类配置：

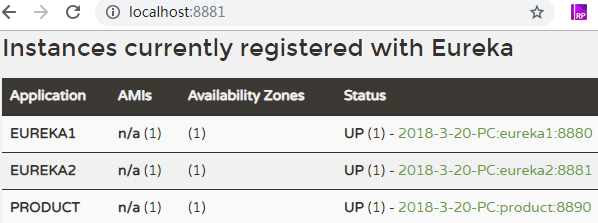


编写一个测试接口：

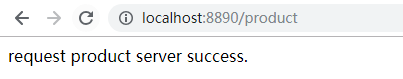


访问eureka服务端口，可以看到两个注册中心数据是同步的，上面product服务的application.yml配置文件向8880和8881两个注册中心服务注册了，不过如果修改成只向8880注册，两个注册中心的数据也是一样的，他们之间的数据是相互同步的，不过要注意他们之间的数据同步是有时间差的。





访问：<http://localhost:8890/product>，可以看到服务一切正常。



如果关闭一台注册中心，请求也是成功的。

至此我们已经搭建了一套高可用注册中心。

1. **Ribbon**

Ribbon起到负载均衡的作用。

下面验证负载均衡的现象，启动服务：

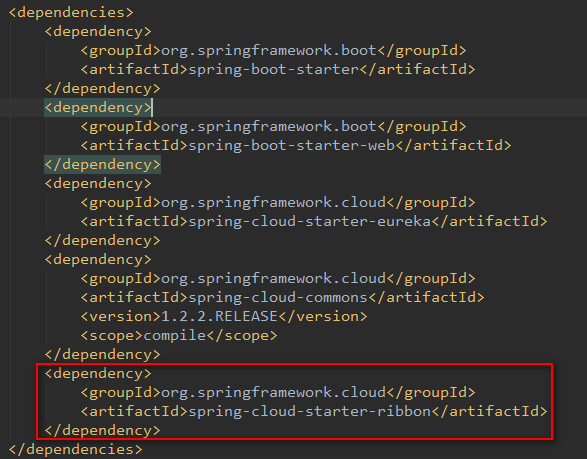
Eureka：启动一个或者两个节点。

Product服务（服务提供者）：启动两个节点。

Consumer服务（服务消费者）：启动一个节点。

新建consumer服务（服务消费者），去消费product服务中的 ”product” 接口，consumer服务配置如下：

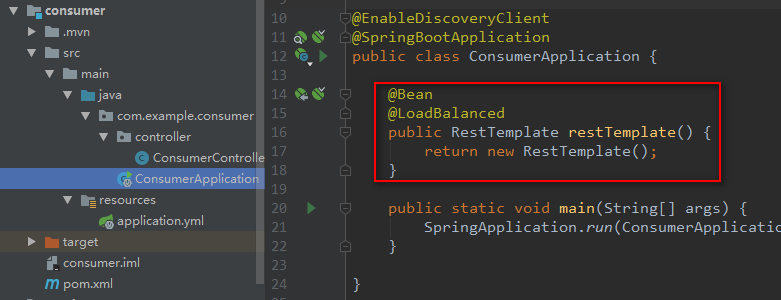
Pom配置：



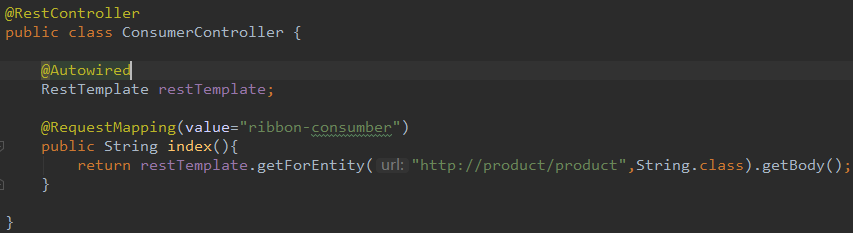
Application.yml配置：

**server**:  
 **port**: 8895  
**spring**:  
 **application**:  
 **name**: consumer  
**eureka**:  
 **instance**:  
 **hostname**: localhost  
 **prefer-ip-address**: **true  
 client**:  
 *#是否注册到Eureka中* **registerWithEureka**: **true** *#是否需要去检索寻找服务，默认是true* **fetchRegistry**: **true  
 serviceUrl**:  
 **defaultZone**: http://${**eureka.instance.hostname**}:8880/eureka/,http://${**eureka.instance.hostname**}:8881/eureka/

启动类配置：

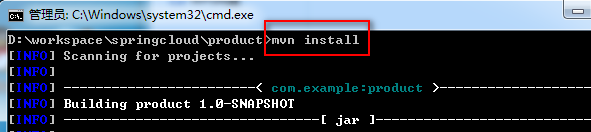


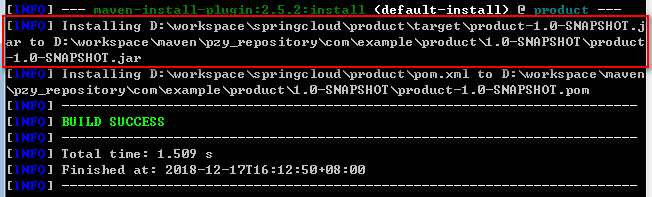
新建ribbon-consumber接口：



消费者服务（consumer）配置完成，下面手动启动两个product服务：

进入product服务目录中，执行 mvn install 命令：



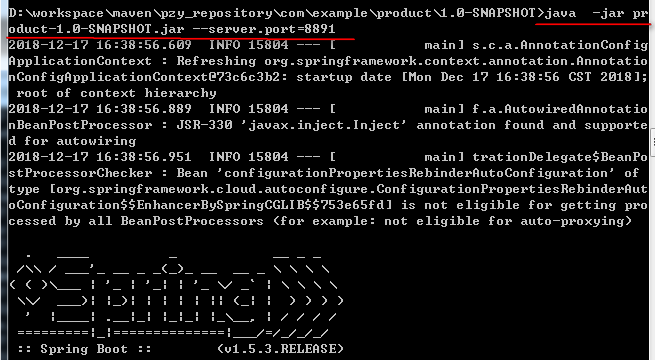


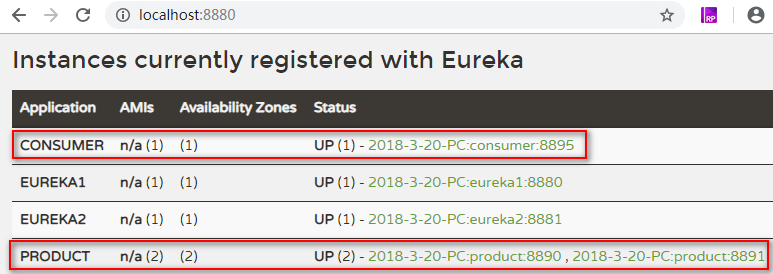
Jar包打在了红色框的位置，进入目录：

执行命令：

java -jar \*.jar --server.port=8890;

java -jar \*.jar --server.port=8891;

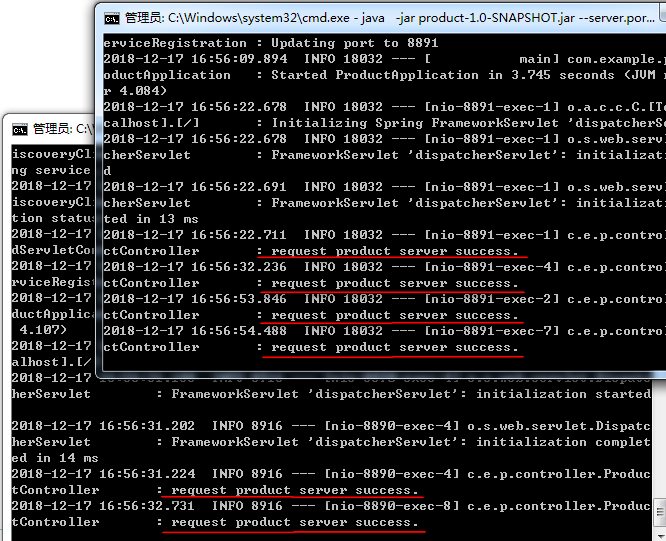




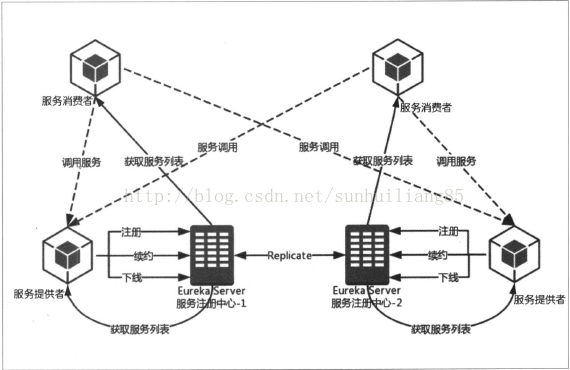
此时服务一切正常，访问：<http://localhost:8895/ribbon-consumber> 接口。

此次测试调了6次，可以看到请求已经负载到两个服务提供者节点上。

这里把服务提供者里增加了一个”request product server success.”日志。



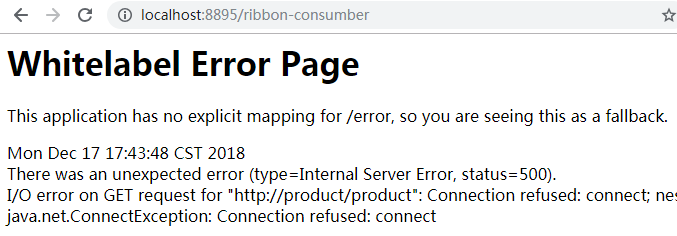
此时服务搭建完成如下图：



1. **Hystrix**

Hystrix是容错保护服务，在微服务架构中很多请求可能因为网络问题或者依赖服务自身问题出现调用故障或延迟，这样也会导致调用方也出现延迟，当请求不断增加时，响应任务积压过多，导致调用方服务的瘫痪，服务之间依赖较多时，最终会导致服务间的雪崩效应。hystrix的引入是将响应超时、服务故障等问题采用服务降级、服务熔断、请求缓存等一系列解决方案进行解决。

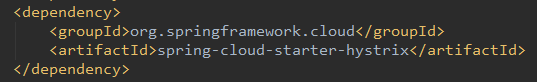
在引入hystrix之前，当访问的服务挂掉时，返回结果如下，体验很不好。



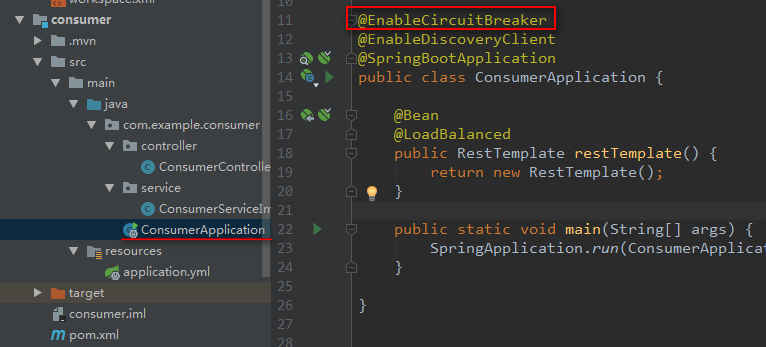
**下面演示当服务提供者出现故障、超时时，服务消费者采用降级处理。**

在上面的服务消费者代码的基础上增加配置：

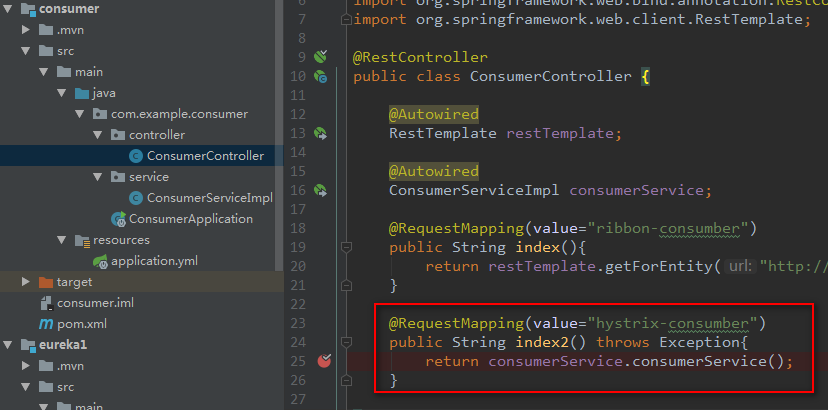
Pom文件增加：



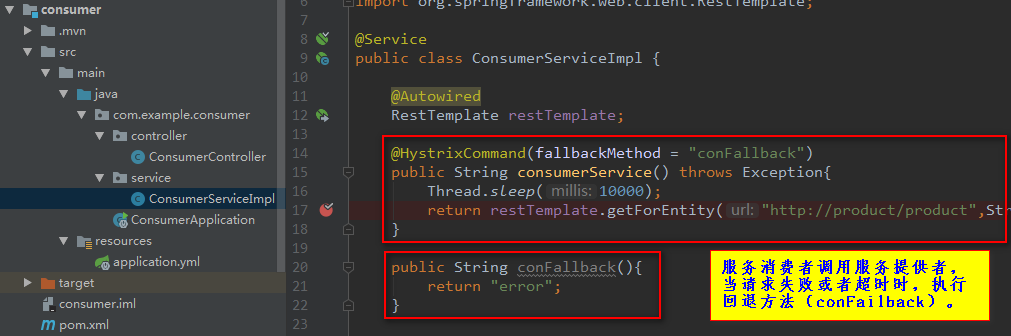
启动类增加注解：



增加接口：

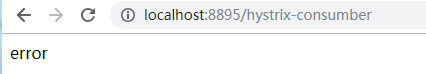


接口中调用的方法如下：



测试：

当服务提供者挂掉或者响应超时时会返回如下：



至此，服务降级解决方案已完成，请求缓存等其它一系列解决方案后期再演示。

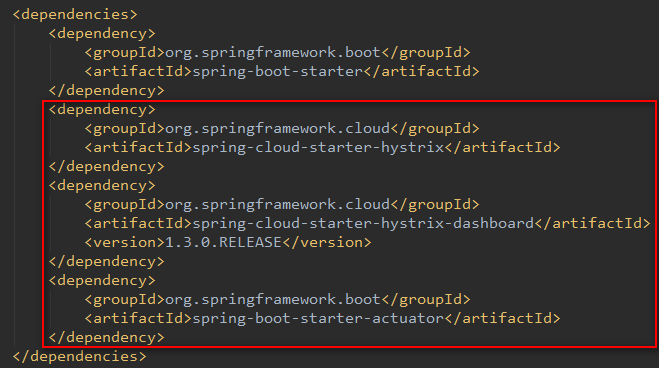
**Hystrix仪表盘**

实时监控请求状况。

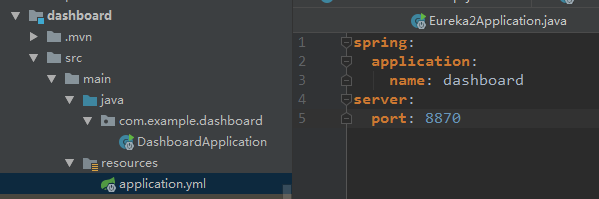
**单点监控dashboard**

新建项目dashboard。

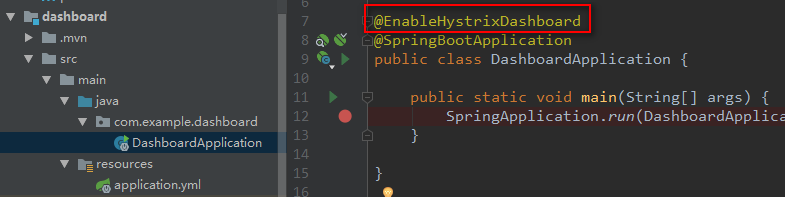
配置pom:



配置application.yml



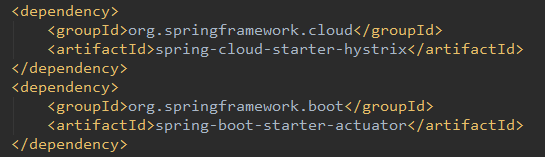
配置启动类：



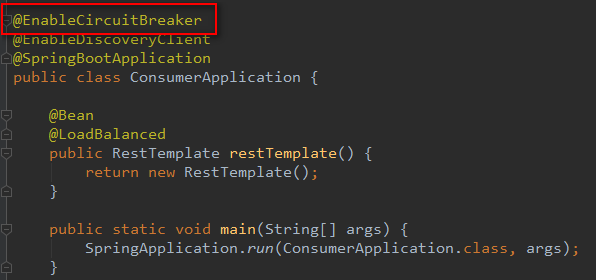
Dashboard项目已经配置完成，另外需要在被监控的项目中引入下面配置：

这里测试监控消费者服务项目（consumer）。

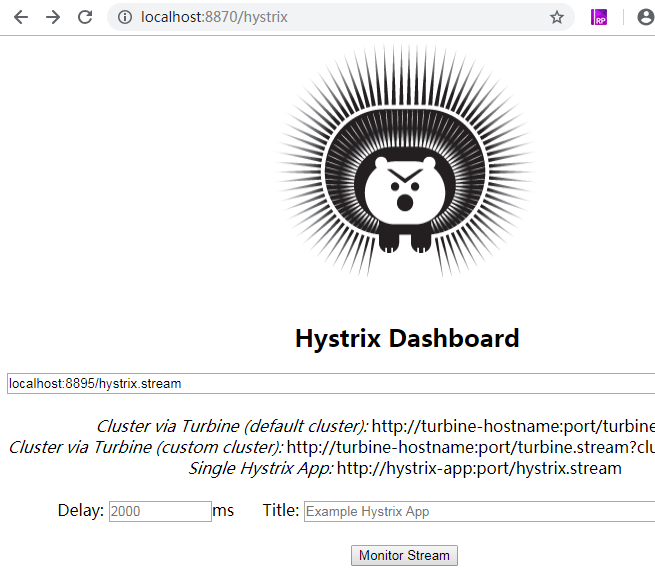
Pom增加：



确保监控的服务的启动类有下面注解，并且开启了断路器功能。

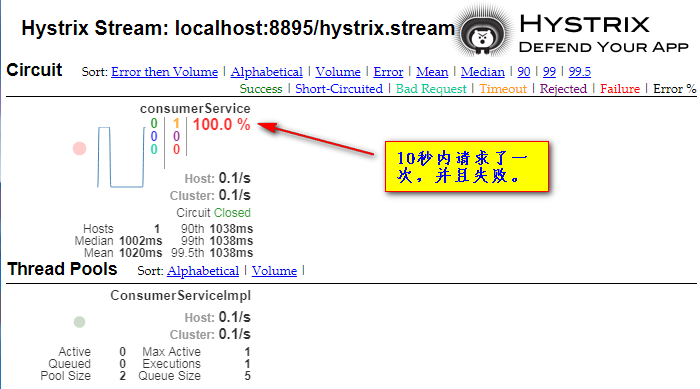


此时，访问 <http://localhost:8870/hystrix>

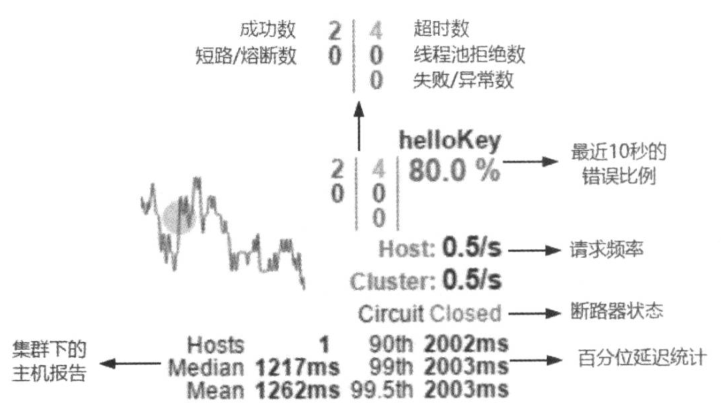


Input框中输入需要监控的服务地址和端口：localhost:8895/hystrix.stream

下面就是消费者项目的实时请求状况，图中可以看出10秒内请求一次并且失败。



参数详解如下：



**集群监控Turbine**

后期再演示

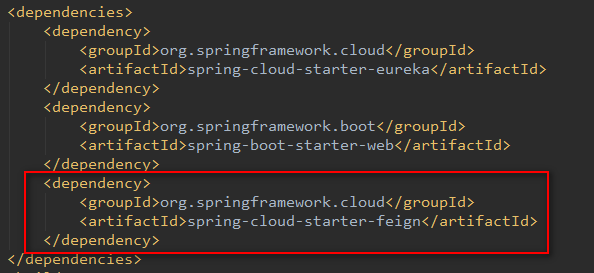
1. **Feign**

服务间的接口调用组件。

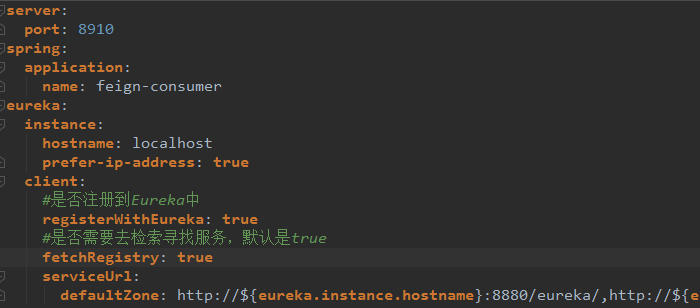
新建一个feign-consumer项目，演示在此项目中调用上面product（服务提供者）中的product接口。

新建feign-consumer项目：

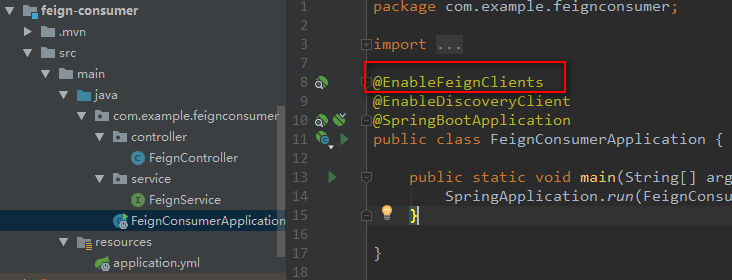
配置pom：



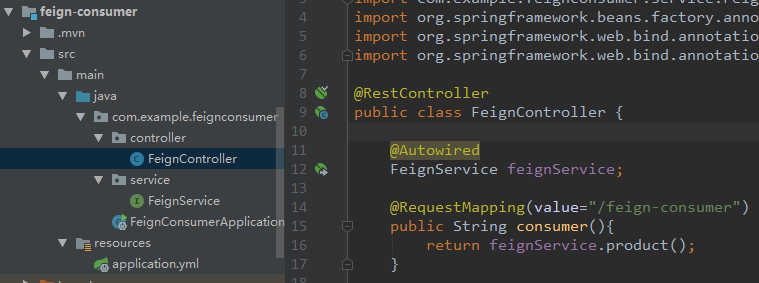
配置application.yml



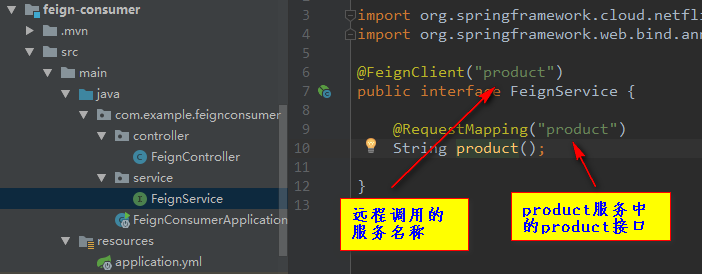
配置启动类：



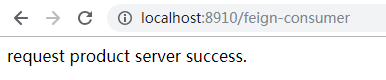
新建接口：



上面接口远程调用product服务中的product方法：



测试：

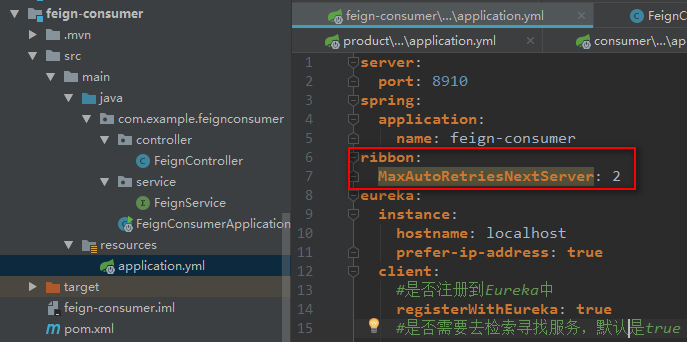


**Ribbon配置（Feign）**

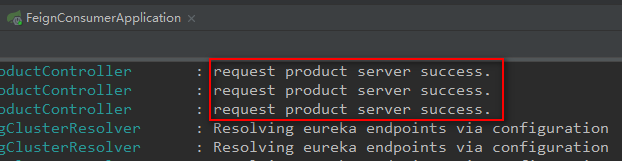
服务间的远程调用参数配置下面会统一介绍。

**重试机制（Feign）**

配置feign-consumer的重试次数



上面设置重试2次，现在还是调用上面的http://localhost:8910/feign-consumer接口，将product服务的product接口打上断点，feign-consumer项目请求超时时启动重试机制，再次调用product接口，最终可以看到product服务被调了3次。



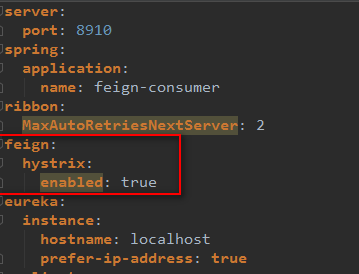
这里要注意Hystrix的超时时间要大于Ribbon的超时时间，否则Hystrix超时直接熔断，重试机制就没有意义，参数配置后面统一介绍。

**服务降级配置（Feign）**

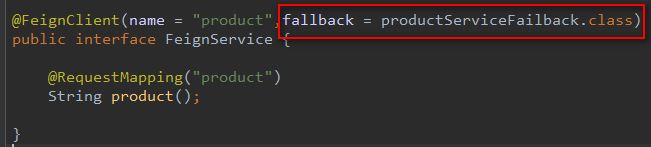
与前面介绍的Hystrix服务降级原理一样，只是这里写法不同而已，这里做回退方法的统一配置。

下面在feign-consumer项目的基础上演示服务降级。

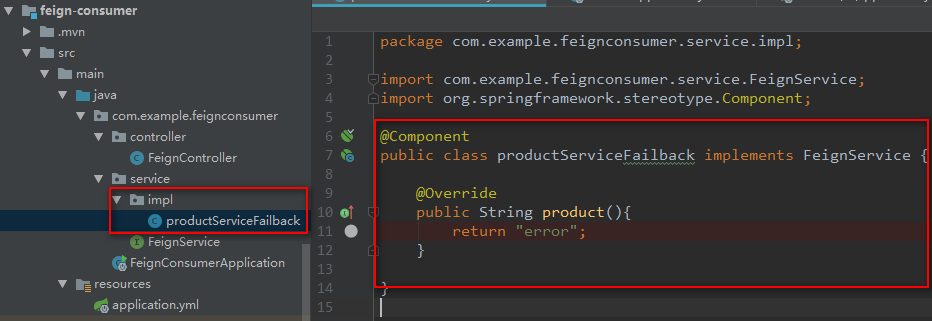
Application.yml配置文件增加下面配置：



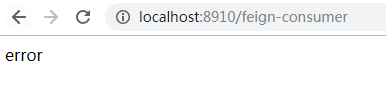
在原远程调用的接口类上增加回退类配置：



下面是新增的回退类，回退类实现远程调用接口类：

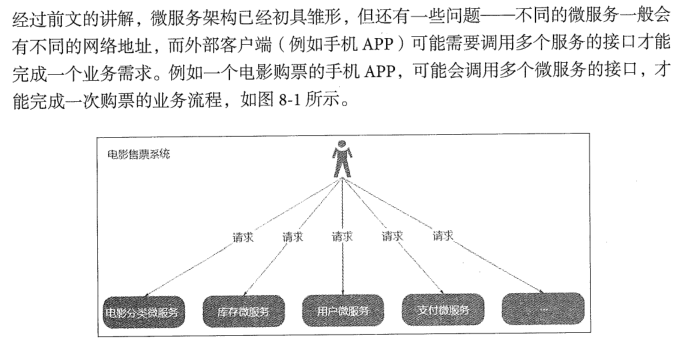


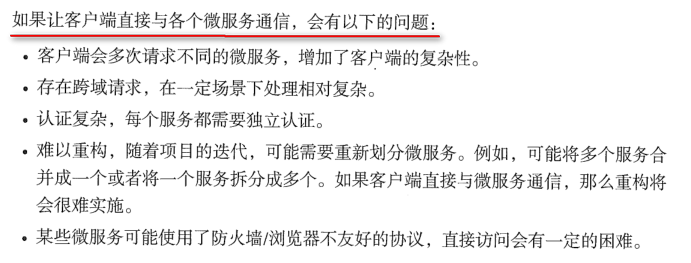
当请求超时和product服务挂掉时返回结果如下，执行了回退方法，测试成功。

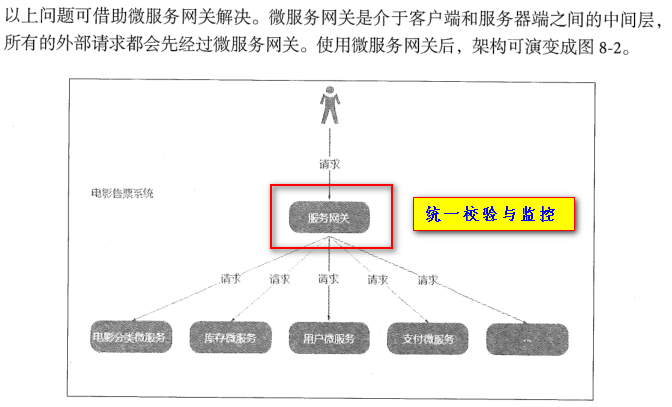


至此，Feign的基本用法已经介绍完成，feign的引入使得服务间的调用比较方便，但是这里也要注意，由于接口在构建期间就建立起了依赖，接口的变动会对项目的构建造成影响，所以接口变动时要注意修改调用方的代码。

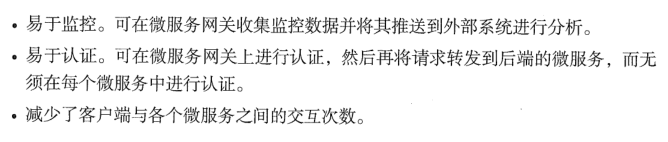
1. **Zuul**







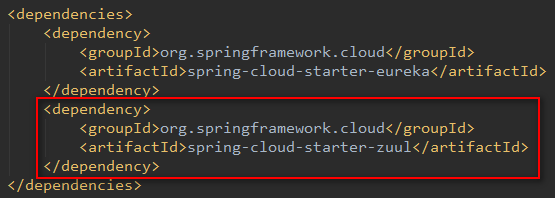
优点:



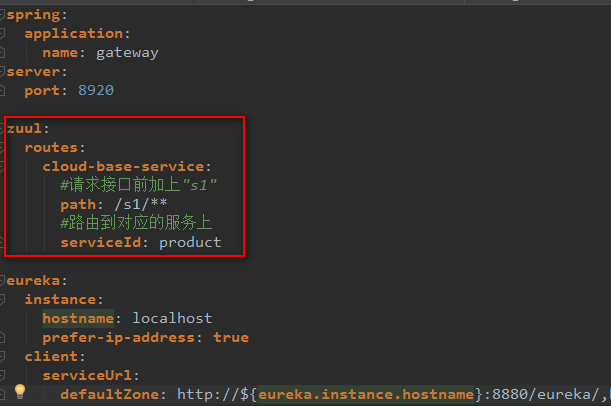
下面演示通过网关来访问其他服务：

创建一个新的项目gateway：

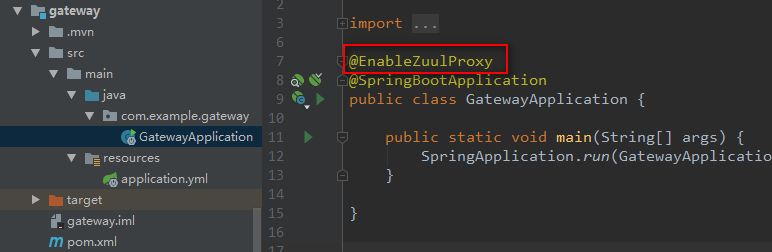
配置pom：



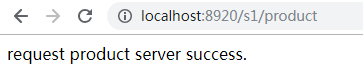
配置application.yml:



配置启动类：

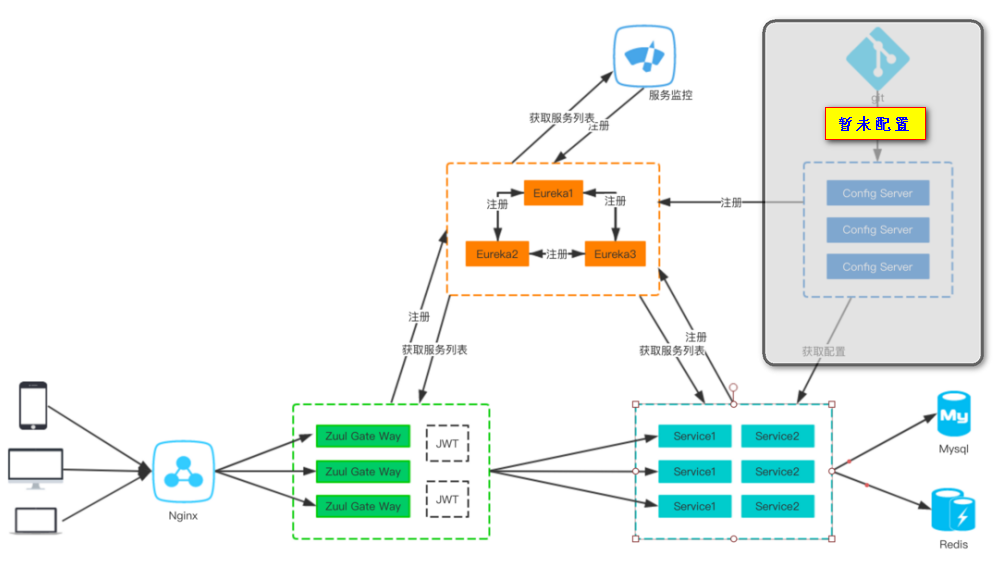


访问网关地址，测试成功：



其它过滤和路由配置暂不演示。

至此，已完成服务搭建如下：



上面演示代码地址：

**<https://github.com/403977059/SpringCloud>**

1. **参数配置**
2. **Eureka**

**eureka.server.enable-self-preservation**：是否开启自我保护模式。

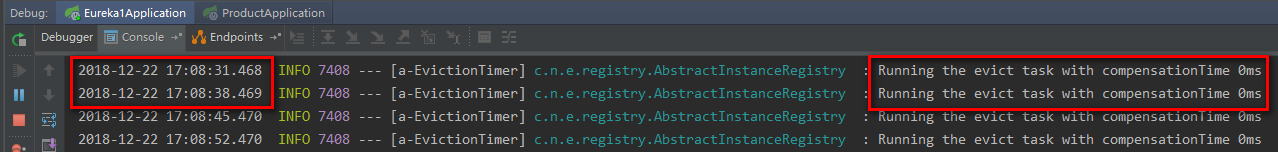
默认情况下，如果Eureka Sever在一定时间内没有接收到某个微服务实例的心跳信息，Eureka Server 会直接注销该实例。但是当网络分区发生故障时，微服务与Eureka Server之间无法通信，直接注销微服务实例就会非常危险。服务本身而言是健康的，不应该直接注销微服务。

Eureka Server 通过 自我保护模式 来解决这个问题。当Eureka Server 节点在短时间内丢失多个客户端时，那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server 就会保护服务注册表的信息，不会再删除注册表信息（不会注销实例信息）。当网络故障恢复后，该Eureka Server 节点会自动退出自我保护模式。

**eureka.server.registerWithEureka：**是否将自身实例信息注册到eureka服务端，默认true。

**eureka.server.fetchRegistry：**是否从Eureka服务端获取注册信息，默认true。

**eureka.server.eviction-interval-timer-in-ms：**指定 Eviction Task 定时任务的调度频率，用于剔除过期的实例，有效防止的问题是：应用实例异常挂掉，没能在挂掉之前告知Eureka server要下线掉该服务实例信息。



上图可以看出，eureka服务设置的是7秒清除一次。

1. **服务提供者**

**eureka.instance.lease-renewal-interval-in-seconds:** 向注册中心发送心跳频率。默认30s。

**eureka.instance.lease-expiration-duration-in-seconds：**告诉eureka多久没有给它发送心跳就自己的注册信息从eureka服务列表中清除。

下面演示eviction-interval-timer-in-ms、lease-renewal-interval-in-seconds、lease-expiration-duration-in-seconds这3个参数的配置，如果服务挂掉，eureka如何清除该服务现象：

首先将eureka1服务中加入服务监听类，可以在控制台中看到服务的实时状态。

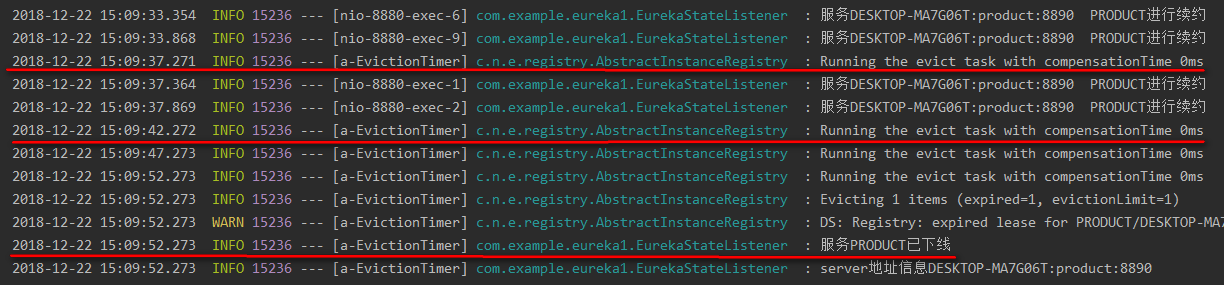
下面的测试前提都是**eureka.server.enable-self-preservation**参数为false，关闭保护模式。

测试一：

配置eureka服务： eviction-interval-timer-in-ms=5000

配置product服务：lease-renewal-interval-in-seconds=4

lease-expiration-duration-in-seconds=7



时间轴：33 37 37 41 42 45 47 49 52 53 （秒）

蓝色是product服务心跳频率；

灰色是product服务挂掉后没有发送的心跳；

红色是eureka服务剔除无效服务列表频率；

从时间轴上可以看出，41 + 7（8） = 48（49） ，47 < 49 < 52，所以product服务会在52秒的时候被清除掉。

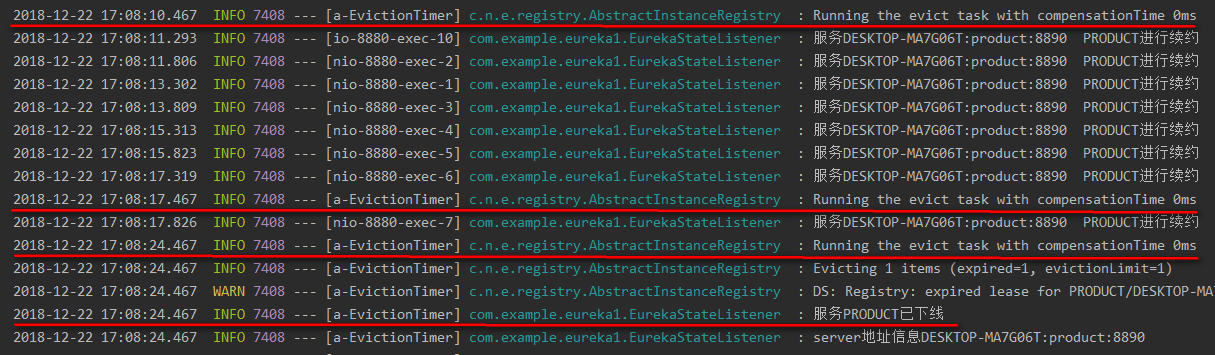
注：上面41+7（8）中的7是lease-expiration-duration-in-seconds，8是 4\*2 > 7，两次心跳才大于7，所以至少两次没有心跳才能剔除该服务。

测试二：

配置eureka服务： eviction-interval-timer-in-ms=7000

配置product服务：lease-renewal-interval-in-seconds=2

lease-expiration-duration-in-seconds=3



时间轴：15 17 17 19 21 23 24 25 （秒）

蓝色是product服务心跳频率；

灰色是product服务挂掉后没有发送的心跳；

红色是eureka服务剔除无效服务列表频率；

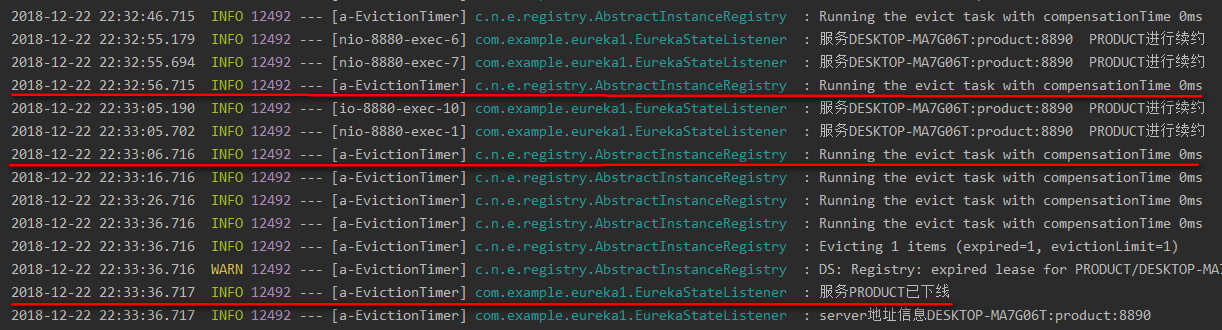
从时间轴上可以看出，19 + 3（4） = 22（23） ，17 < 23 < 24，所以product服务会在24秒的时候被清除掉。

测试三：

配置eureka服务： eviction-interval-timer-in-ms=10000

配置product服务：lease-renewal-interval-in-seconds=10

lease-expiration-duration-in-seconds=12



时间轴：55 56 05 06 15 16 25 26 35 36（秒）

蓝色是product服务心跳频率；

灰色是product服务挂掉后没有发送的心跳；

红色是eureka服务剔除无效服务列表频率；

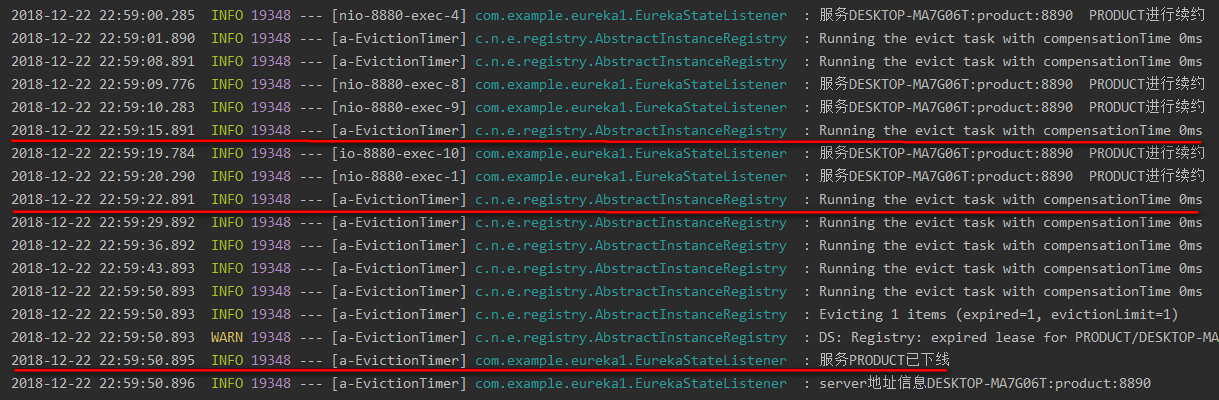
从时间轴上可以看出，15 + 12（20） = 27（35） ，26 < 35 < 36，所以product服务会在36秒的时候被清除掉。

测试四：

配置eureka服务： eviction-interval-timer-in-ms=7000

配置product服务：lease-renewal-interval-in-seconds=10

lease-expiration-duration-in-seconds=12



时间轴：10.2 15.8 20.2 22.8 29.8 30.2 36.8 40.2 43.8 50.2 50.8 0.2（秒）

蓝色是product服务心跳频率；

灰色是product服务挂掉后没有发送的心跳；

红色是eureka服务剔除无效服务列表频率；

从时间轴上可以看出，30.2 + 12（20） = 42.2（50.2） ，43.8 < 50.2 < 50.8，所以product服务会在50.8秒的时候被清除掉。

**结论：**

**当服务挂掉时，从第一次没有心跳开始计算（30.2），当心跳值（50.2-30.2 > 12），则认为该服务已经挂掉，等待下一次eureka剔除时间（50.8）到时，将该服务剔除。**

该结论仅是多次测试结果中得出，没有查看源码，后期会再查看源码，如有出入会及时更正。

1. **Ribbon**

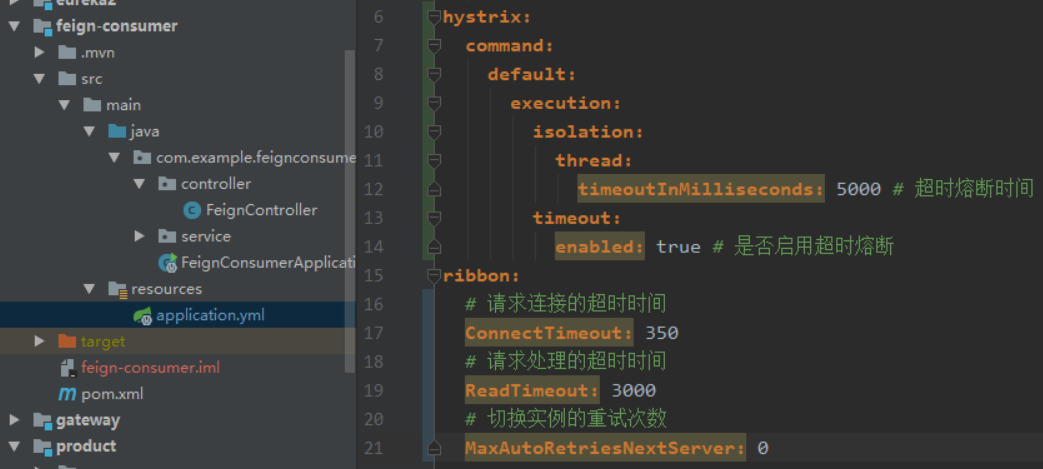
**ribbon.ConnectTimeout:**  ribbon创建连接的超时时间

**ribbon.ReadTimeout：**请求处理的超时时间

**ribbon.MaxAutoRetriesNextServer：**切换实例的重试次数（上文有演示，暂时没有演示多节点）

**ribbon.MaxAutoRetries：**对当前实例的重试次数

**ribbon.OkToRetryOnAllOperations：**是否对所有操作请求都进行重试



**注意**：

在测试**ribbon.ReadTimeout**参数时，设置超时时间5000，测试代码执行只需2.1s，但是还是执行了超时降级服务代码。原因是feign-consumer项目没有配置熔断的相关参数，但是也会有一些默认参数，默认熔断超时是开启的，而且默认超时时间是1s，所以**ribbon.ReadTimeout**参数配置没有生效。将熔断超时配置成5000，这时请求成功。

一般熔断超时时间要大于ribbon超时时间，不然超时一旦达到熔断的超时时间，直接熔断服务，配置的ribbon重试参数无法生效。

1. **Hystrix**

**hystrix.command.default.execution.timeout.enabled :** 是否启用执行超时，默认true

**hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds：**熔断超时时间

其它参数详情：<https://www.cnblogs.com/520playboy/p/8074347.html>

1. **Feign**

**feign.hystrix.enabled：**是否开启Hystrix功能

**feign.compression.request.enabled：**请求是否GZIP压缩（减少通讯性能消耗）

**feign.compression.response.enabled:** 响应是否GZIP压缩（减少通讯性能消耗）

**feign.compression.request.mime-types=text/xml,application/xml,application/json：**指定压缩的请求数据类型

**feign.compression.request.min-request-size=2048**：配置压缩数据大小的下限，只有超过这个大小才会进行压缩

以上演示代码地址：

**https://github.com/403977059/SpringCloud2**