# SpringCloud

## 简介

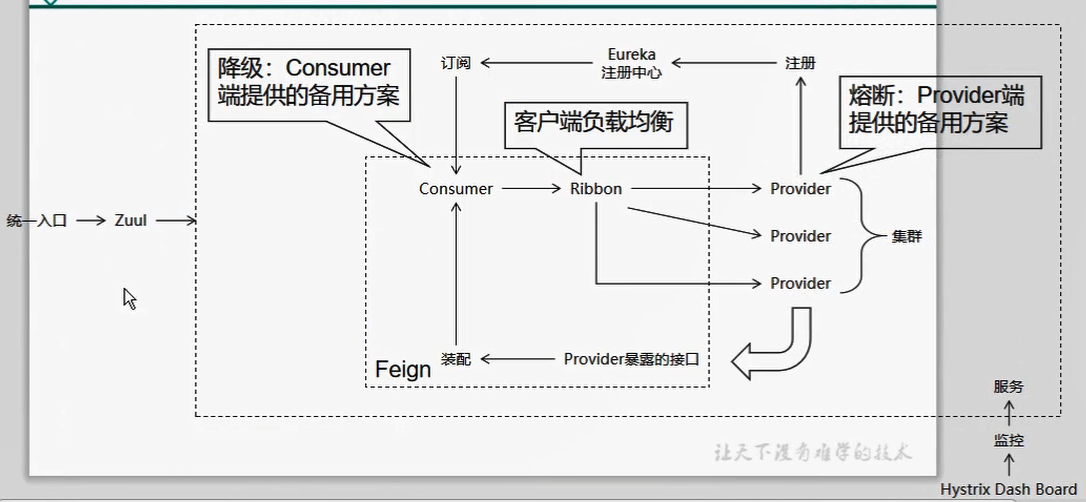
注册中心：Eureka 

负载均衡：Ribbon 

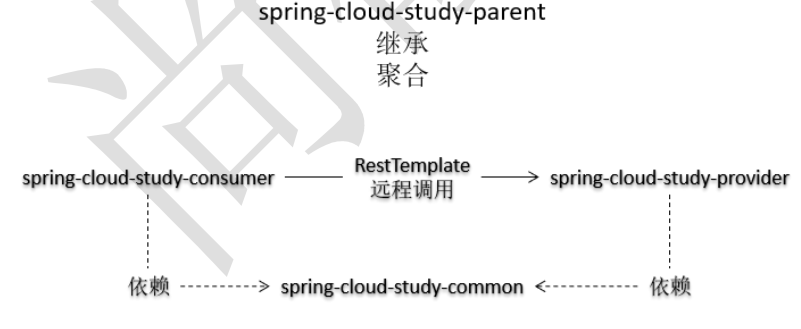
声明式调用远程方法：Feign 

熔断、降级、监控：Hystrix 

网关：Zuul

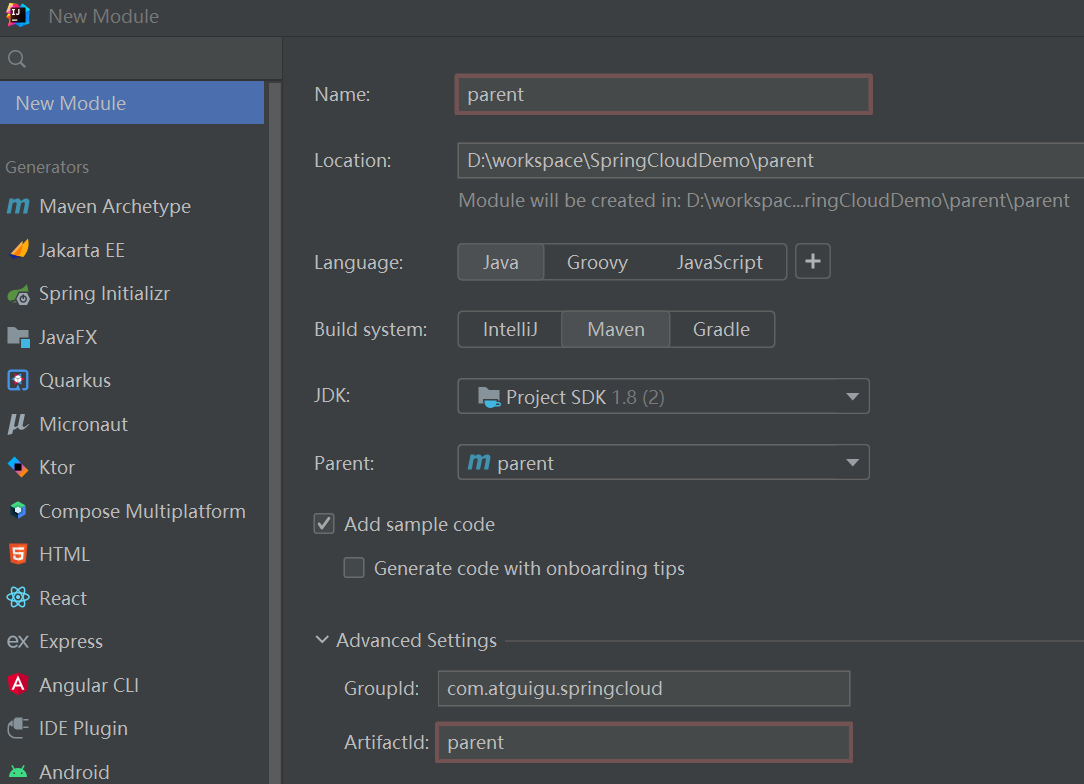


## 基础工程



### 父工程

父工程，用于管理依赖



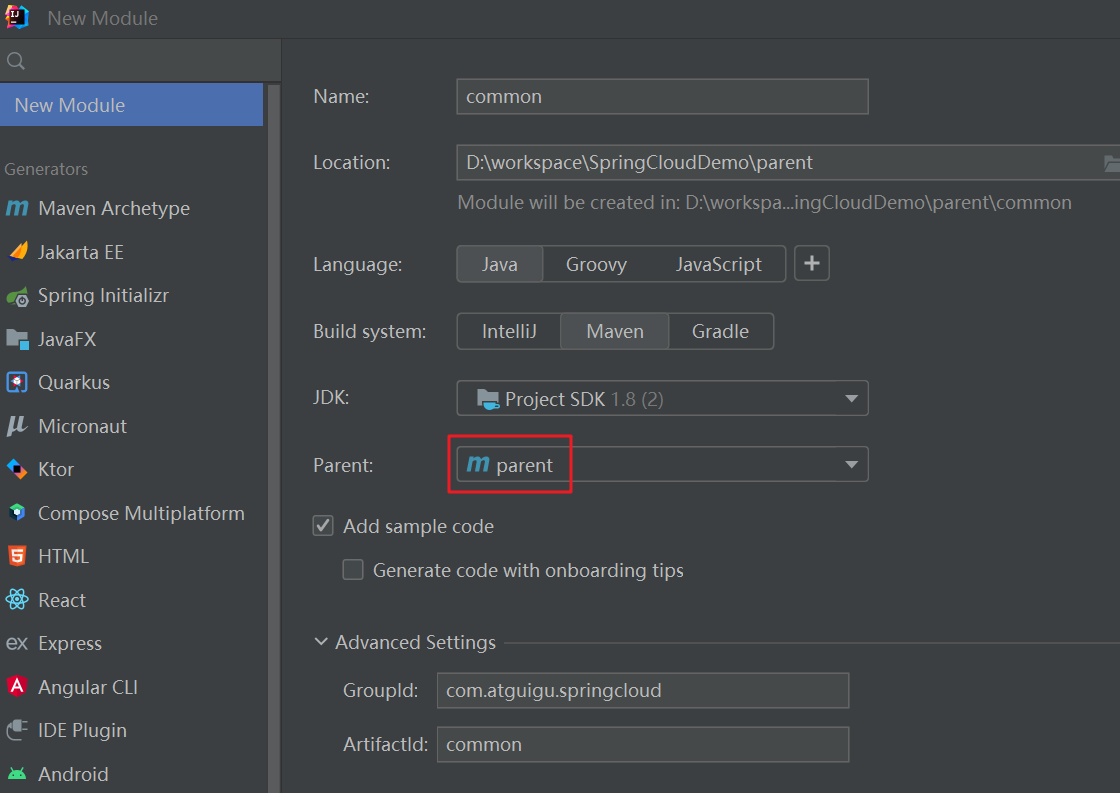
依赖导入：

<dependencyManagement>  
<dependencies>  
 <!-- 导入 SpringCloud 需要使用的依赖信息 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Greenwich.SR2</version>  
 <type>pom</type>  
 <!-- import 依赖范围表示将 spring-cloud-dependencies 包中的依赖信息导入--><scope>import</scope>  
 </dependency>  
 <!-- 导入 SpringBoot 需要使用的依赖信息 -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-dependencies</artifactId>  
 <version>2.1.6.RELEASE</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
</dependencies>  
</dependencyManagement>

### common工程

common工程依赖父工程

打jar包



只有一个实体类：

public class Employee {  
 private Integer empId;  
 private String empName;  
 private Double empSalary;

### provider工程

springboot工程，jar包，依赖父工程和common工程

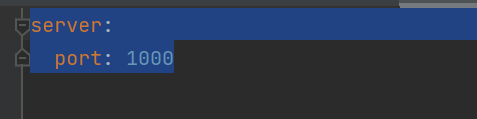
1.引入依赖：

<dependencies>  
 <dependency>  
 <!-- 指定groupId和artifactId即可，版本已在父工程中定义 -->  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.atguigu.springcloud</groupId>  
 <artifactId>common</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
</dependencies>

2.主启动类：

@SpringBootApplication  
public class ProviderMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ProviderMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

3.application.yml



4.控制器方法：

@RestController  
public class ProviderHandler {

@RequestMapping("/provider/getReomteEmploee")  
 public Employee getRemoteEmploee(){  
 return new Employee(10001,"shuyun",1000.05);  
 }  
}

### consumer工程

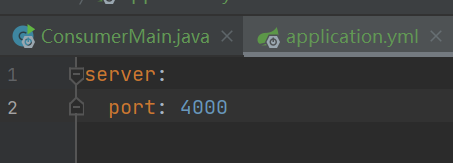
1.引入依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.atguigu.springcloud</groupId>  
 <artifactId>common</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
</dependencies>

2.主启动类

@SpringBootApplication  
public class ConsumerMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ConsumerMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

3.application.yml



4.创建配置类，提供RestTemplate，用于远程调用

@Configuration  
public class ConsumerConfig {  
  
 @Bean  
 public RestTemplate getRestTemplate(){  
 return new RestTemplate();  
 }  
}

5.控制器方法：

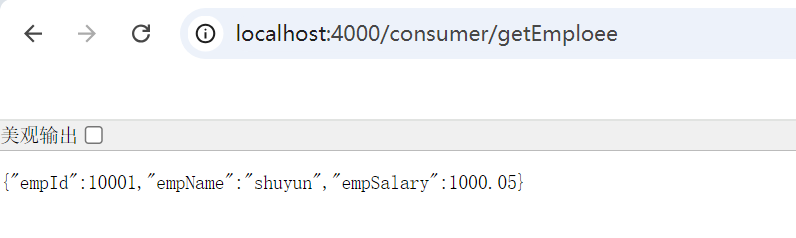
@RestController  
public class HumanResourceHandler {  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @RequestMapping("/consumer/getEmploee")  
 public Employee getEmploeeRemote(){  
  
 //远程调用方法的主机地址  
 String host = "http://localhost:1000";  
  
 //远程调用方法的具体url地址  
 String url = "/provider/getReomteEmploee";  
  
 //通过 restTemplate 调用远程方法  
 return restTemplate.getForObject(host+url, Employee.class);  
  
 }  
}

### 调用测试

1、启动provider工程

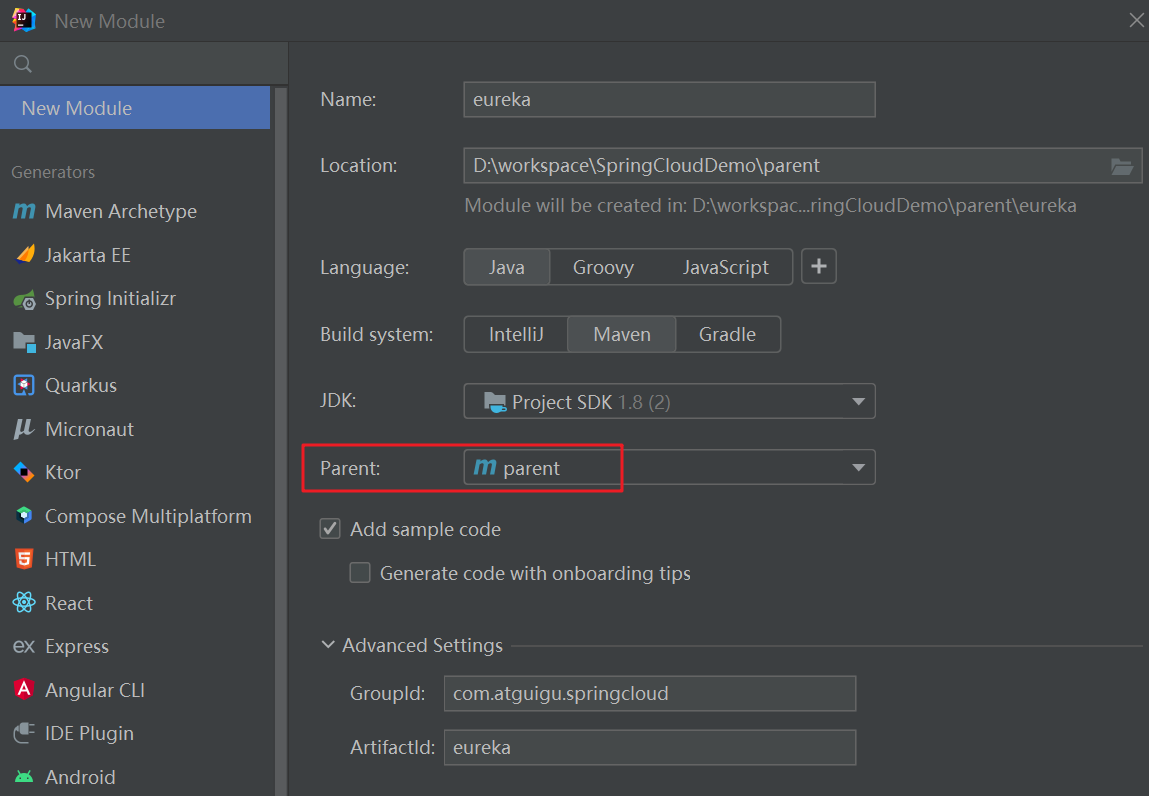
2、启动consumer工程

3、访问consumer工程的 getEmploeeRemote方法，返回的是provider工程的调用结果



## Eureka 注册中心

### 工程创建



### 依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>  
</dependencies>

### 启动类

@EnableEurekaServer 注解表示 启用 Eureka 服务器功能

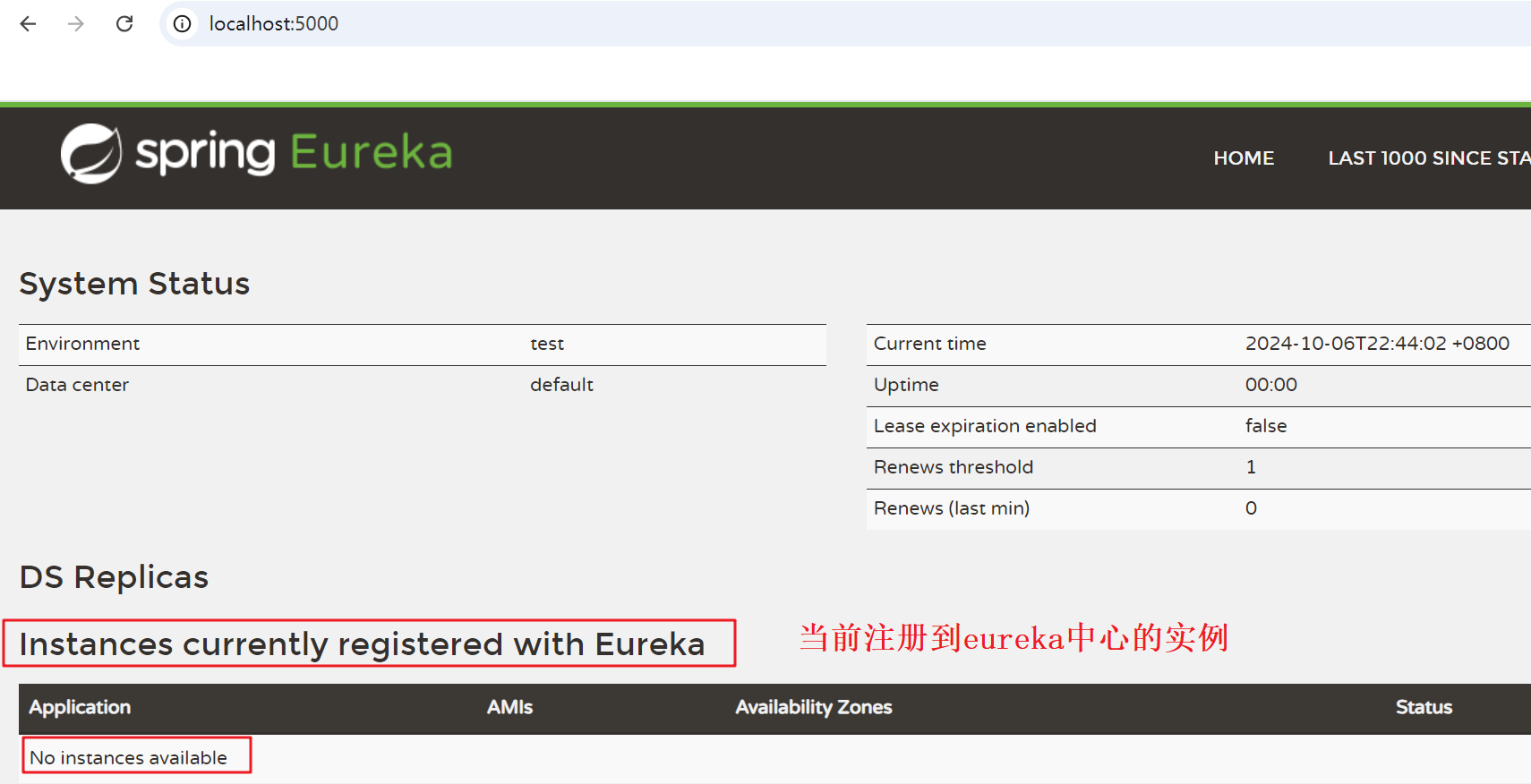
@SpringBootApplication  
@EnableEurekaServer  
public class EurekaMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(EurekaMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml配置

server:  
 port: 5000  
eureka:  
 instance:  
 hostname: localhost *#配置当前注册中心的服务主机地址* client:  
 register-with-eureka: false *# 自己就是注册中心，所以自己不注册自己* fetch-registry: false *# 自己就是注册中心，所以不需要“从注册中心取回信息”* service-url: *#客户端（consumer和provider）访问 Eureka 时使用的地址* defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

### 启动

启动服务器后，访问 localhost:5000，看到如下界面，说明eureka注册中心创建成功



### 将provider注册到eureka

provider工程做如下修改

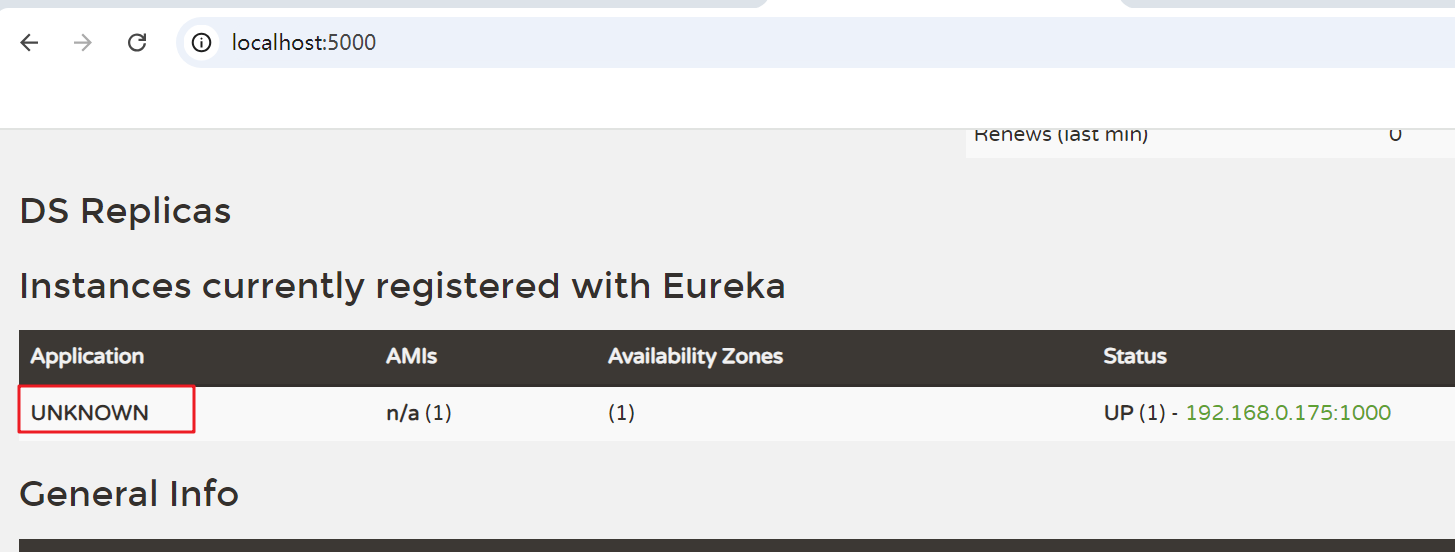
1.增加依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

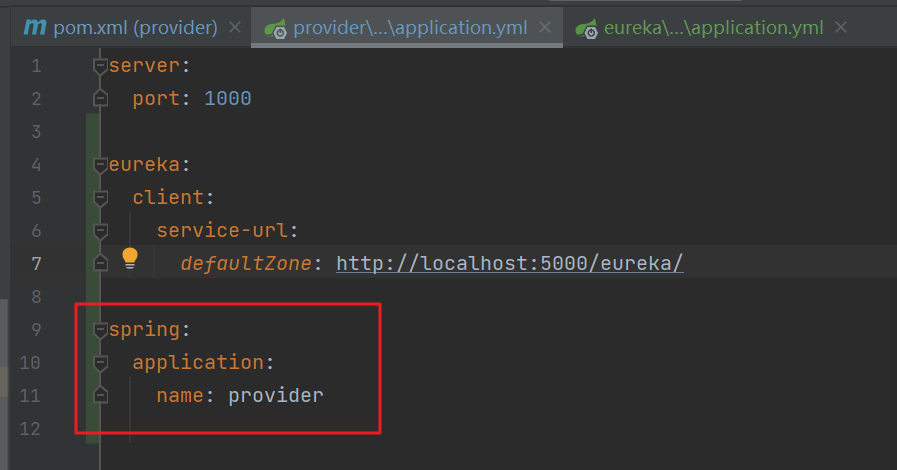
2.增加配置application.yml



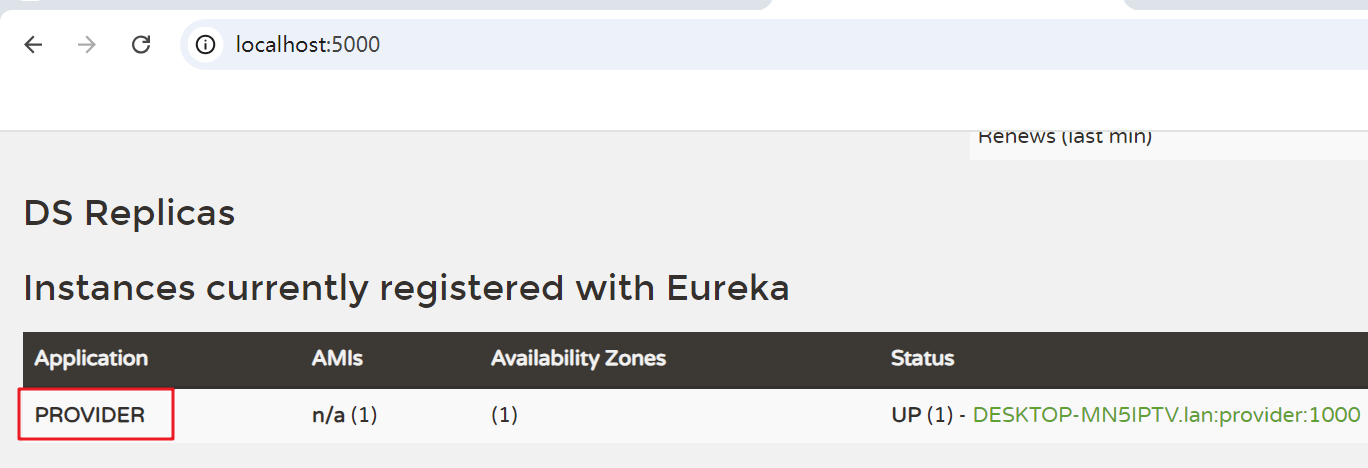
3.重新启动后，刷新eureka注册中心的地址，可以看到如下界面



显示unknown是因为provider 工程没有指定应用名称，指定应用名称配置方式如下：

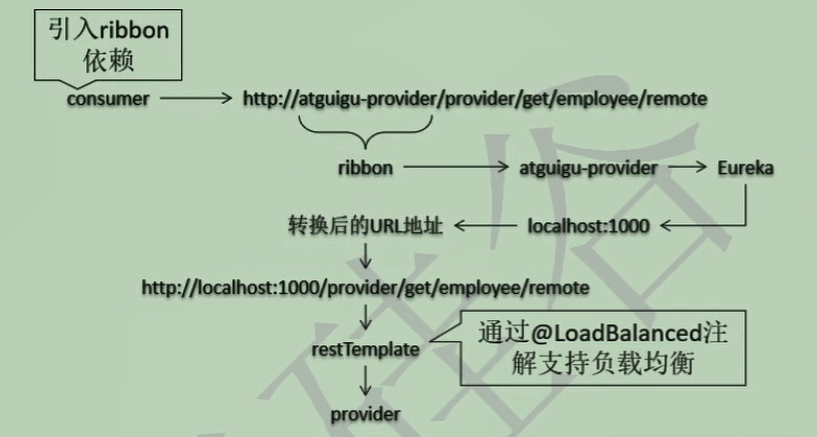


eureka注册中心显示如下：



以后在 SpringCloud 环境下开发，每一个微服务工程都要设置一个应用名称。

### consumer使用微服务名称调用provider



1.引入依赖

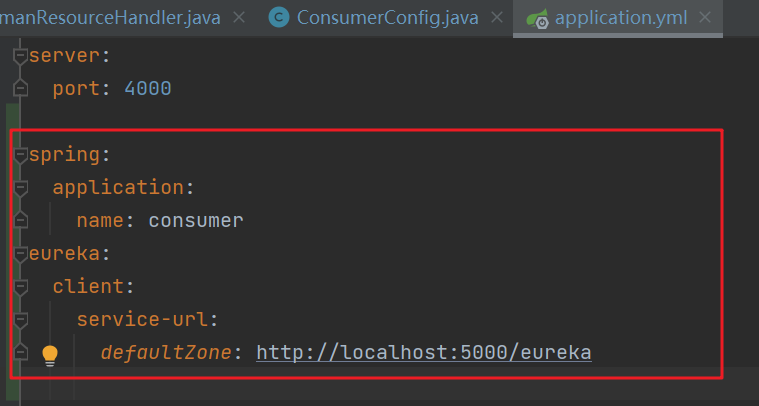
eureka-client是eureka客户端，用来注册到eureka

ribbon是将 微服务名转换成具体的ip和端口，负载均衡方式调用远程微服务

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

2.application.yml配置

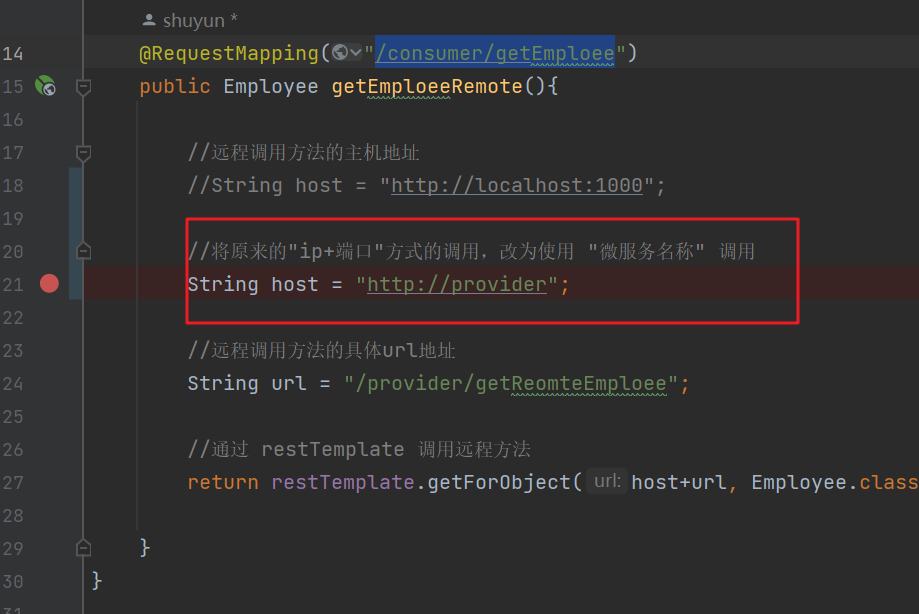
配置微服务名称和eureka服务器地址



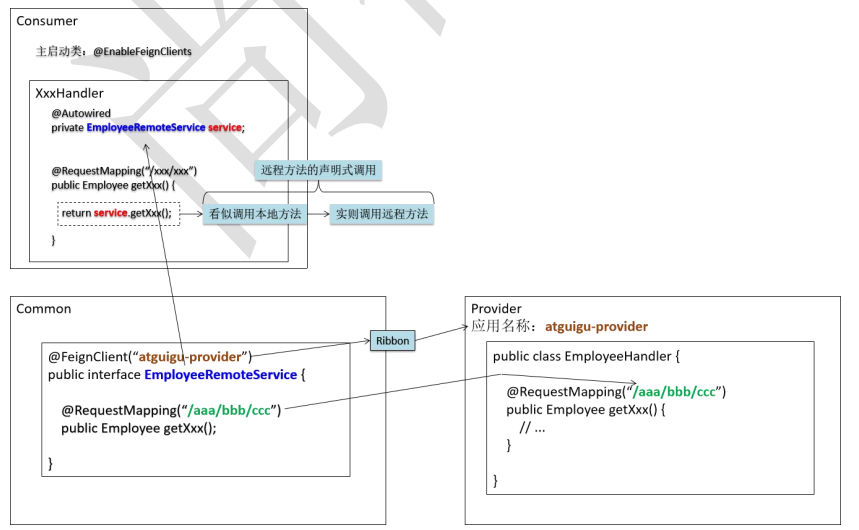
3.获取远程调用方法的Bean加上 @LoadBalance注解

@Configuration  
public class ConsumerConfig {  
  
 @Bean  
 @LoadBalanced  
 public RestTemplate getRestTemplate(){  
 return new RestTemplate();  
 }  
}

4.控制器方法远程调用



## Feign远程方法调用



1.common中声明接口和抽象方法

2.provider中声明与common中的接口相同的实现类，声明具体的方法（不是实现）

3.consumer中装配common中的接口，调用相应的方法。需要在启动类加上@EnableFeignClients注解

### common工程

1.引入依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
 </dependency>  
</dependencies>

2.创建远程调用方法的接口

注意，接口所在包要符合springboot扫描规则，要被扫描到IOC容器里面才能被consumer工程使用

注意：这里的方法要和 Provider中的具体实现的方法的声明要一致

@RequestMapping注解的url地址、@RequestParam、@RequestBody、@PathVariable两边一致

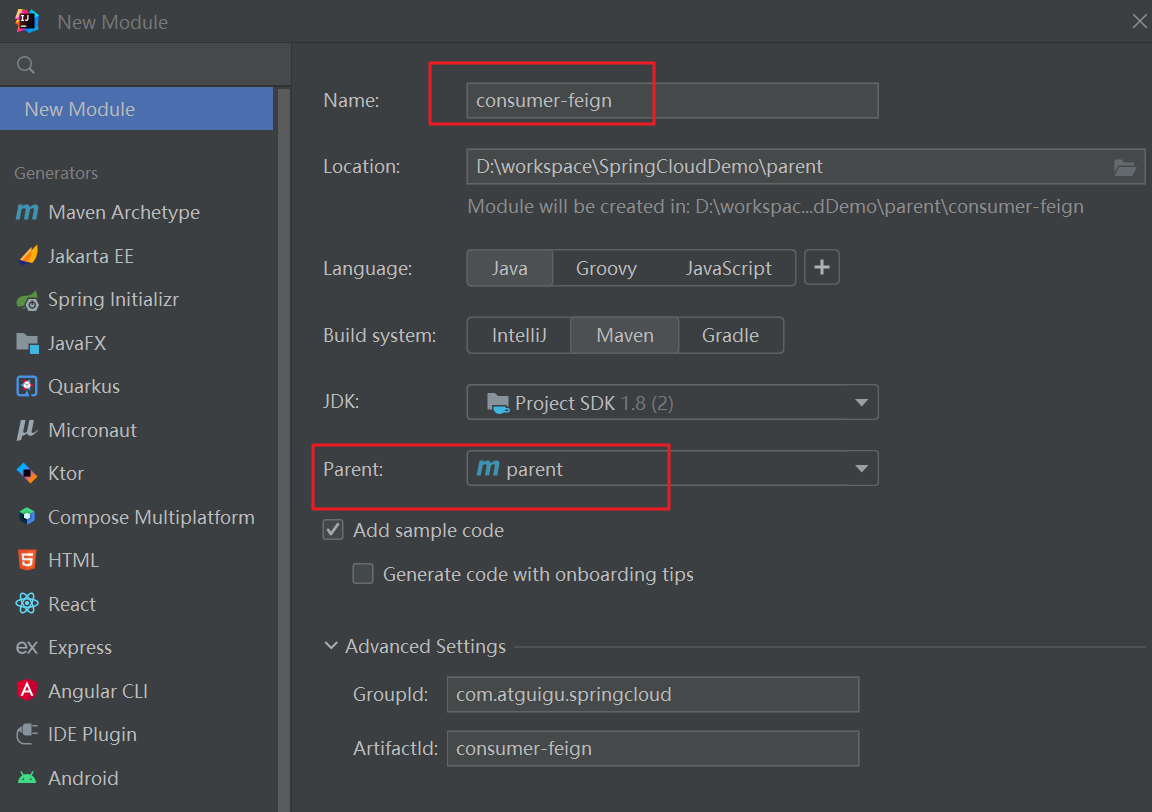
//@FeignClient注解表示和一个provider对应，value属性指定要调用的Provider的微服务名称  
@FeignClient(value = "provider")  
public interface EmployeeRemoteService {  
  
 //注意：这里的方法要和 Provider中的具体实现的方法的声明要一致：

//远程调用的接口方法  
 @RequestMapping("/provider/getReomteEmploee")  
 Employee getRemoteEmploee();  
}

### Feign-consumer工程

1.新建feign-consumer工程

不使用原来的consumer工程了，重新建一个consumer工程



2.引入依赖

ribbon注解在common中的feign已经引入

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.atguigu.springcloud</groupId>  
 <artifactId>common</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
</dependencies>

3.主启动类

加上@EnableFeignClients注解

@SpringBootApplication  
//启用Feign客户端功能  
@EnableFeignClients  
public class ConsumerFeignMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ConsumerFeignMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

4.application.yml配置

配置端口、微服务名称、eureka注册服务器地址

server:  
 port: 8000  
  
spring:  
 application:  
 name: feign-consumer  
  
eureka:  
 client:  
 service-url:  
 defaultZone: http://localhost:5000/eureka

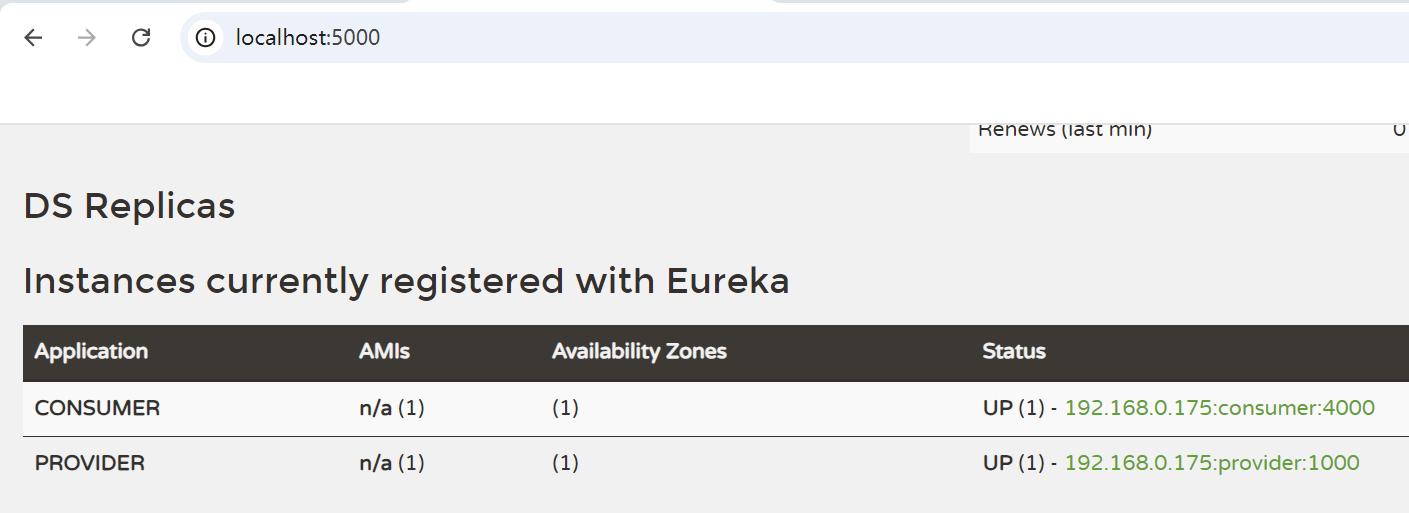
5.控制器方法

@RestController  
public class FeignConsumerHandler {  
  
 //装配远程调用微服务的接口，然后就可以像本地调用一样了  
 @Autowired  
 private EmployeeRemoteService employeeRemoteService;  
  
 @RequestMapping("/feign/getRemoteEmployee")  
 public String getRemoteEmployee(){  
 Employee remoteEmploee = employeeRemoteService.getRemoteEmploee();  
 return remoteEmploee.toString();  
 }  
}

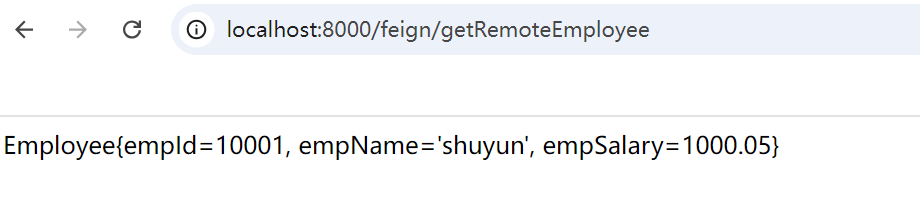
### 测试

先启动eureka，再启动provider和consumer

服务已经注册上



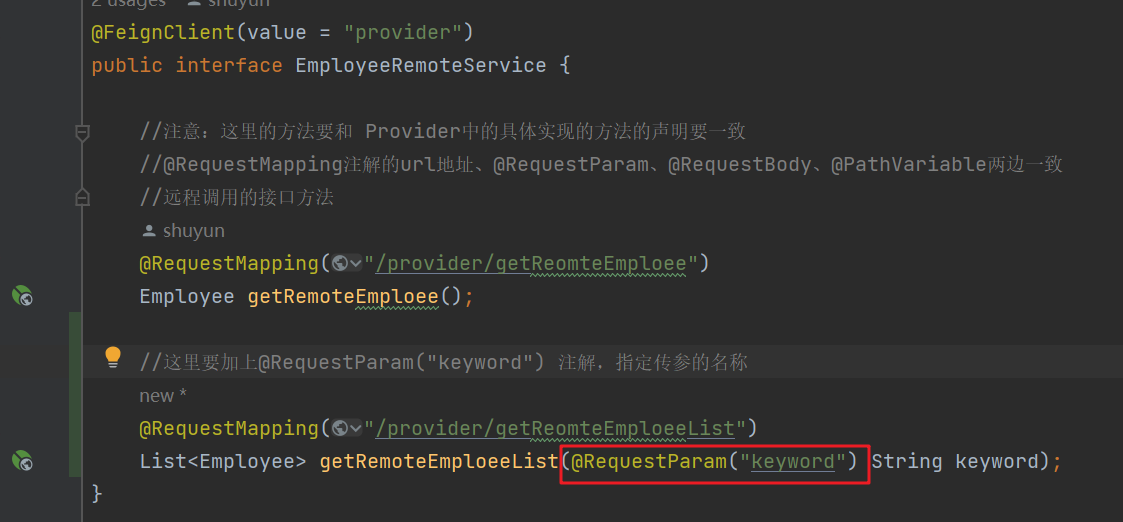
调用consumer的接口



### 简单传参

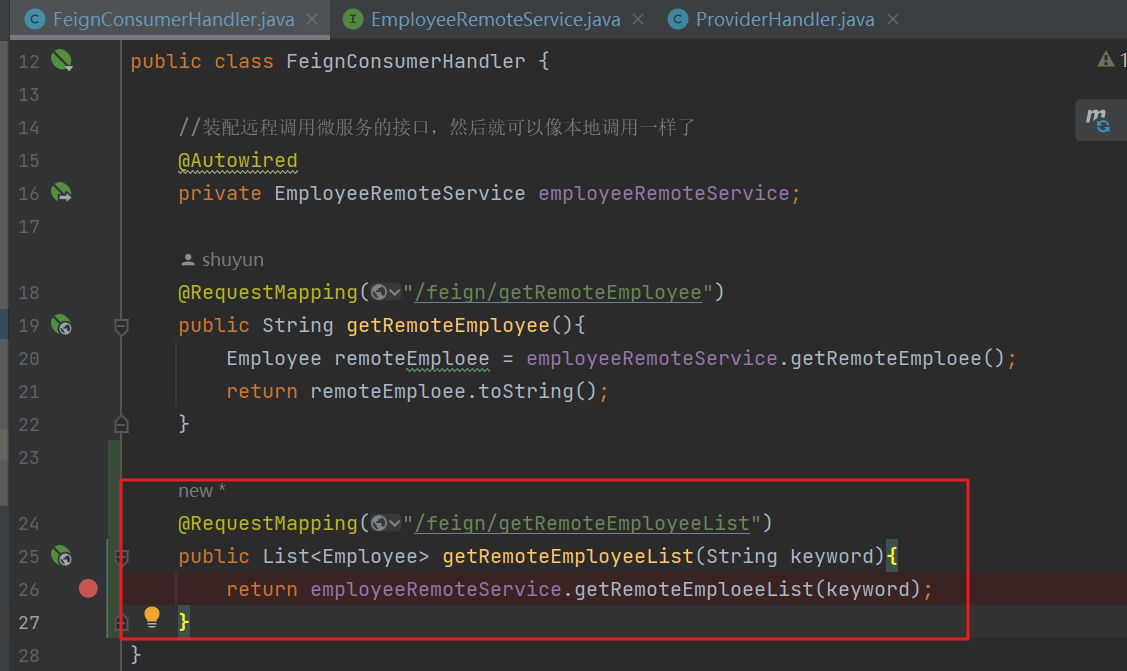
1.common

要加上@RequestParam注解，指定传参的名称



2.consumer

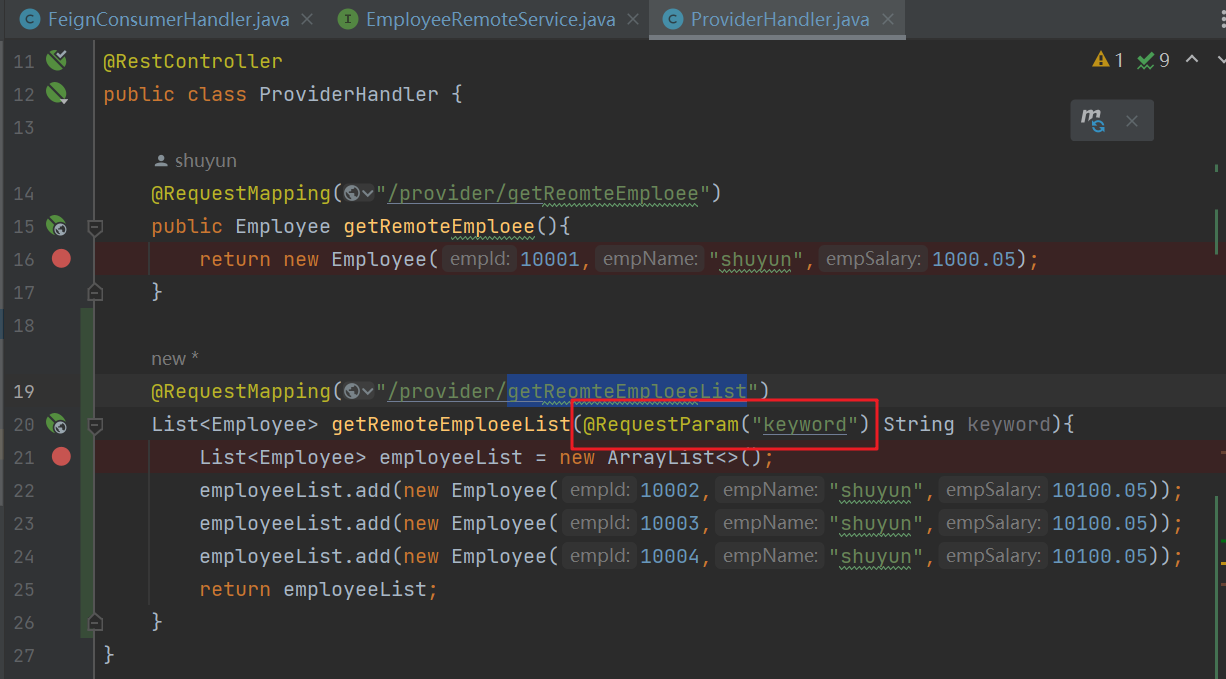
调用远程方法



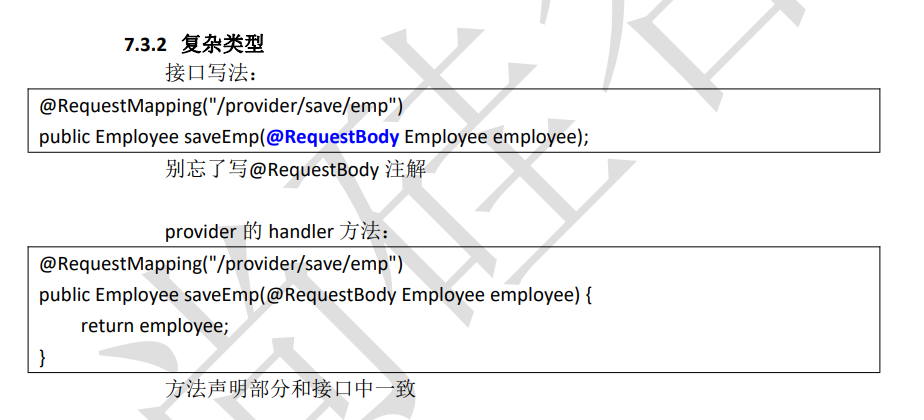
3.provider

方法声明要和common中的一致

@RequestParam注解也要加上

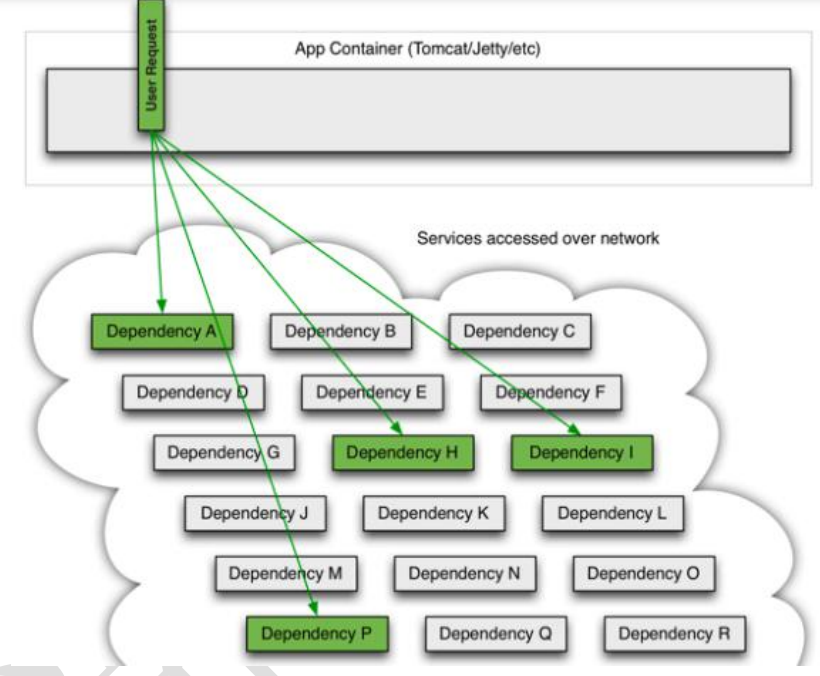


### 复杂传参



## Hystrix

### 分布式系统面临的问题



在微服务架构体系下，服务间的调用错综复杂，交织成一张大网。如果其中某个节点突然无法正常工作，则访问它的众多服务都会被卡住，进而有更多服务被卡住，系统中的线程、CPU、内存等资源有可能被迅速耗尽，最终整个服务体系崩溃。

我们管这样的现象叫服务雪崩。

### Hytrix 介绍

Hystrix 是一个用于处理分布式系统的延迟和容错的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix 能够保证在一个依赖出问题的情况下，不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性。

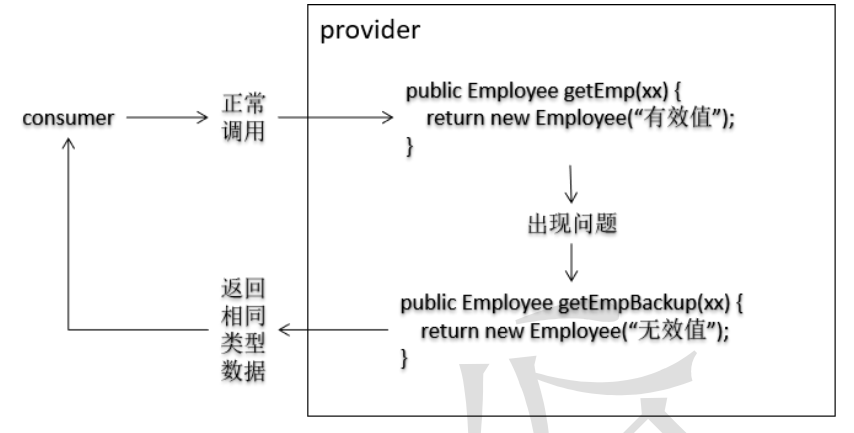
“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），向调用方返回一个符合预期的、可处理的备选响应（FallBack），而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

Hytrix 能够提供服务降级、服务熔断、服务限流、接近实时的监控等方面的功能。

### 服务熔断机制

熔断机制是应对雪崩效应的一种微服务链路保护机制。

当扇出链路的某个微服务不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速响应错误信息。当检测到该节点微服务调用响应正常后恢复调用链路。在 SpringCloud 框架里熔断机制通过 Hystrix 实现。Hystrix 会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是 5 秒内 20 次调用失败就会启动熔断机制。熔断机制的注解是@HystrixCommand。



### 熔断-provider

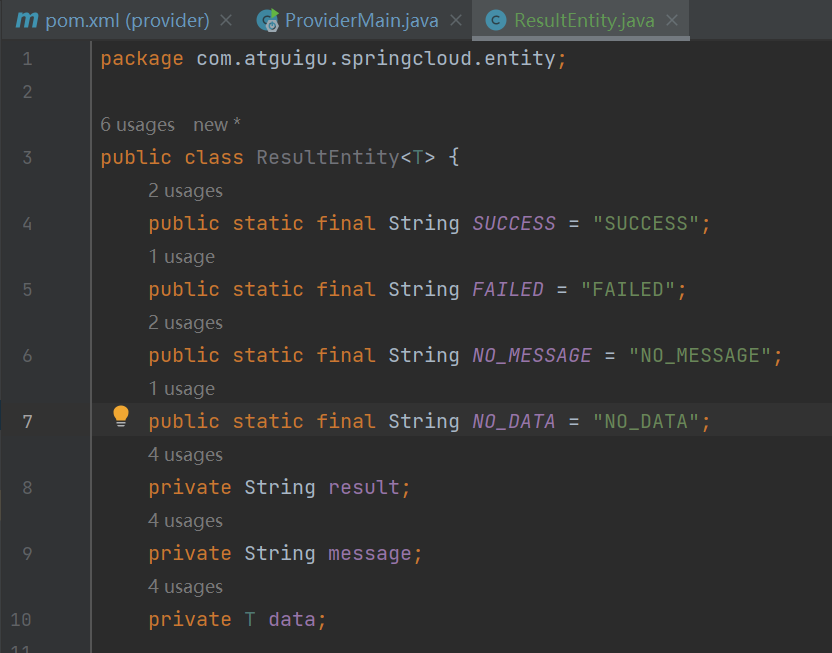
1.在provider工程中引入hystrix依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

2.主启动类加上注解 @EnableCircuitBreaker 启用断路器功能



3.创建ResultEntity，统一作为Ajax请求和远程方法调用返回的类型，放在common工程中



4.provider控制器方法

①在原来的方法上加上注解，指定方法调用出错后调用的熔断方法

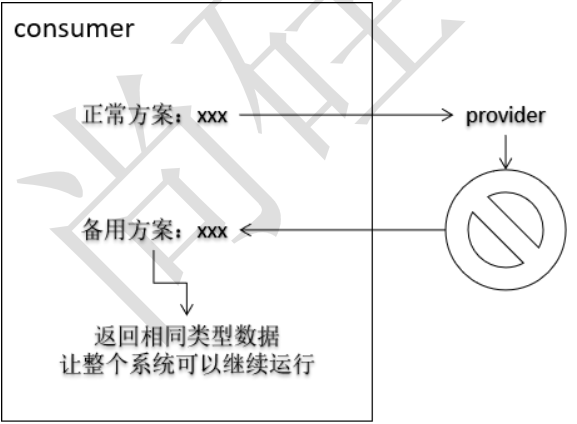
@HystrixCommand 注解通过 fallbackMethod 属性指定断路情况下要调用的备份方法

②新增熔断方法

@HystrixCommand(fallbackMethod = "getEmpBackUp")  
@RequestMapping("/provider/hystrix")  
public ResultEntity<Employee> getEmp(@RequestParam("key") String key){  
  
 if ("haha".equals(key)){  
 throw new RuntimeException();  
 }  
  
 return ResultEntity.*successWithData*(new Employee(10002,"shuyun",10100.05));  
}  
  
public ResultEntity<Employee> getEmpBackUp(@RequestParam("key") String key){  
 return ResultEntity.*failed*("触发熔断机制");  
}

### 降级

服务降级处理是在客户端(Consumer 端)实现完成的，与服务端(Provider 端)没有关系。当某个 Consumer 访问一个 Provider 却迟迟得不到响应时执行预先设定好的一个解决方案，而不是一直等待。



#### common工程依赖

common工程引入hystrix依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

#### common工程定义降级工厂类

请注意自动扫描包的规则

比如：feign-consumer 工程需要使用 MyFallBackFactory，那么 MyFallBackFactory 应该在feign-consumer 工程的主启动类所在包或它的子包下

简单来说：哪个工程用这个类，哪个工程必须想办法扫描到这个类

1.实现FallbackFactory接口，传入的是 @FeignClient 注解标记的接口类型

2.加上@Component注解，注入IOC容器中

3.实现 create()方法，该方法返回的是 @FeignClient 注解 标记的对象，当调用Provider的方法失败时，会调用该对象对应的方法

*/\*\*  
\** ***@author*** *shuyun  
\** ***@date*** *2024-10-08 10:40:57  
\*  
\* 实现Consumer服务降级功能  
\* 实现 FallbackFactory 接口时要传入 @FeignClient 注解标记的接口类型（远程调用的接口）  
\* 在 create() 方法中返回 @FeignClient 注解标记的接口类型的对象。当调用Provider失败后，会执行这个对象的对应的方法  
\*  
\*/*@Component  
public class MyFallBackFactory implements FallbackFactory<EmployeeRemoteService> {  
 @Override  
 public EmployeeRemoteService create(Throwable throwable) {  
 //匿名实现  
 return new EmployeeRemoteService(){  
  
 @Override  
 public Employee getRemoteEmploee() {  
 return null;  
 }  
  
 @Override  
 public List<Employee> getRemoteEmploeeList(String keyword) {  
 return null;  
 }  
  
 @Override  
 public ResultEntity<Employee> getEmp(String key) {  
 return ResultEntity.*failed*("服务降级："+throwable.getMessage());  
 }  
 };  
 }  
}

#### common工程远程调用注解加属性

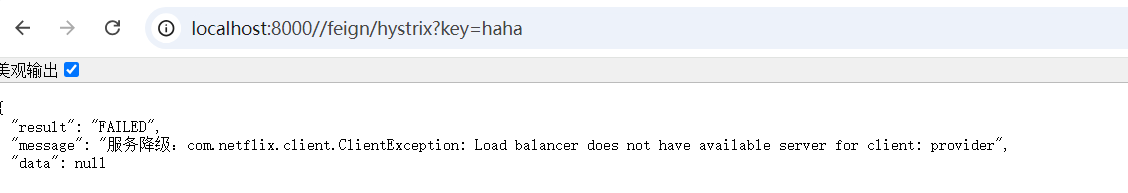
fallbackFactory属性指定 consumer 调用 provider 时如果失败所采取的备用方案



#### consumer工程配置



停了provider服务后



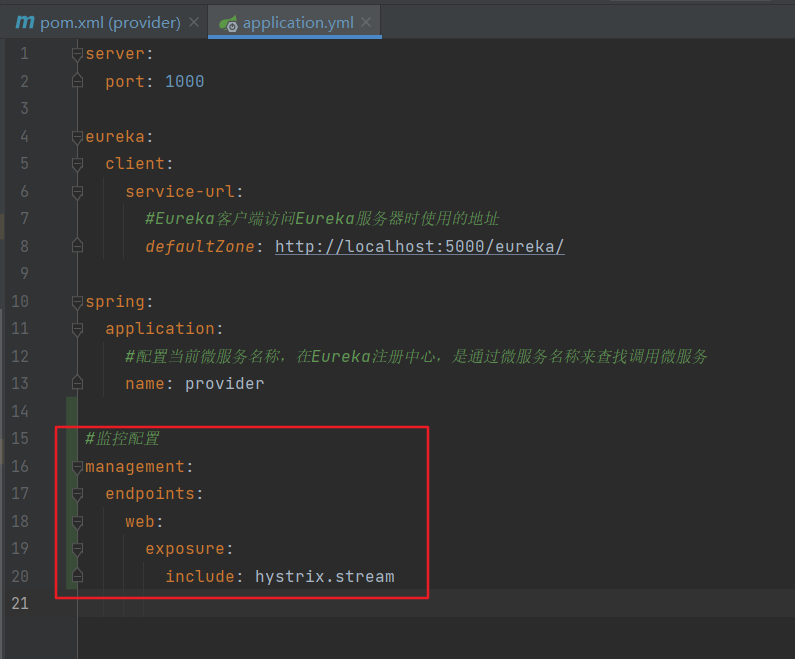
### 监控

#### provider工程

1.引入依赖

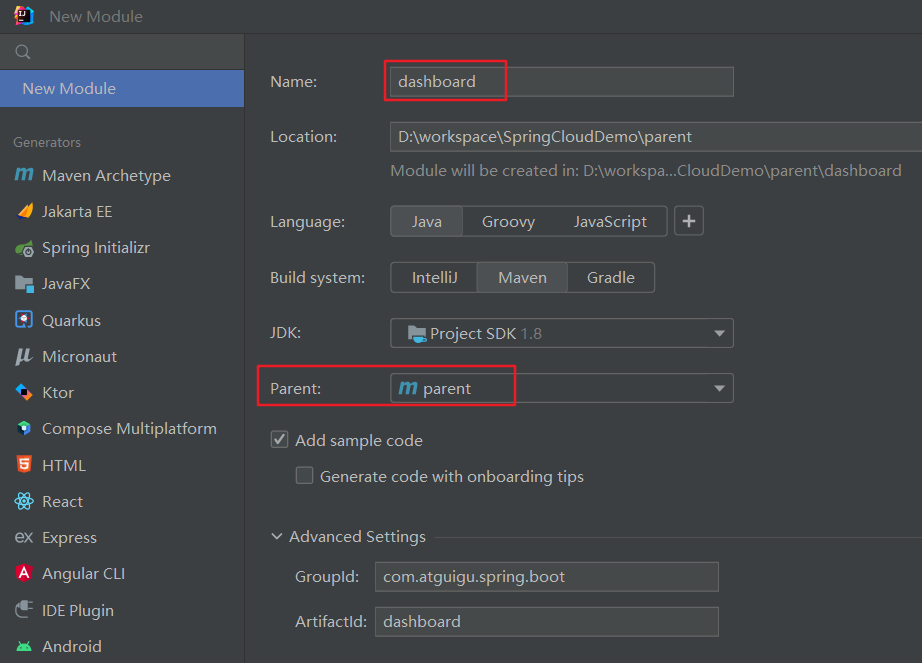
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

2.application.yml配置



#### 新建监控工程

1.新建工程



2.引入依赖

引入dashboard监控依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>  
 </dependency>  
</dependencies>

3.配置application.yml

配置端口和微服务名称

server:  
 port: 9000  
  
spring:  
 application:  
 name: dashboard

4.主启动类

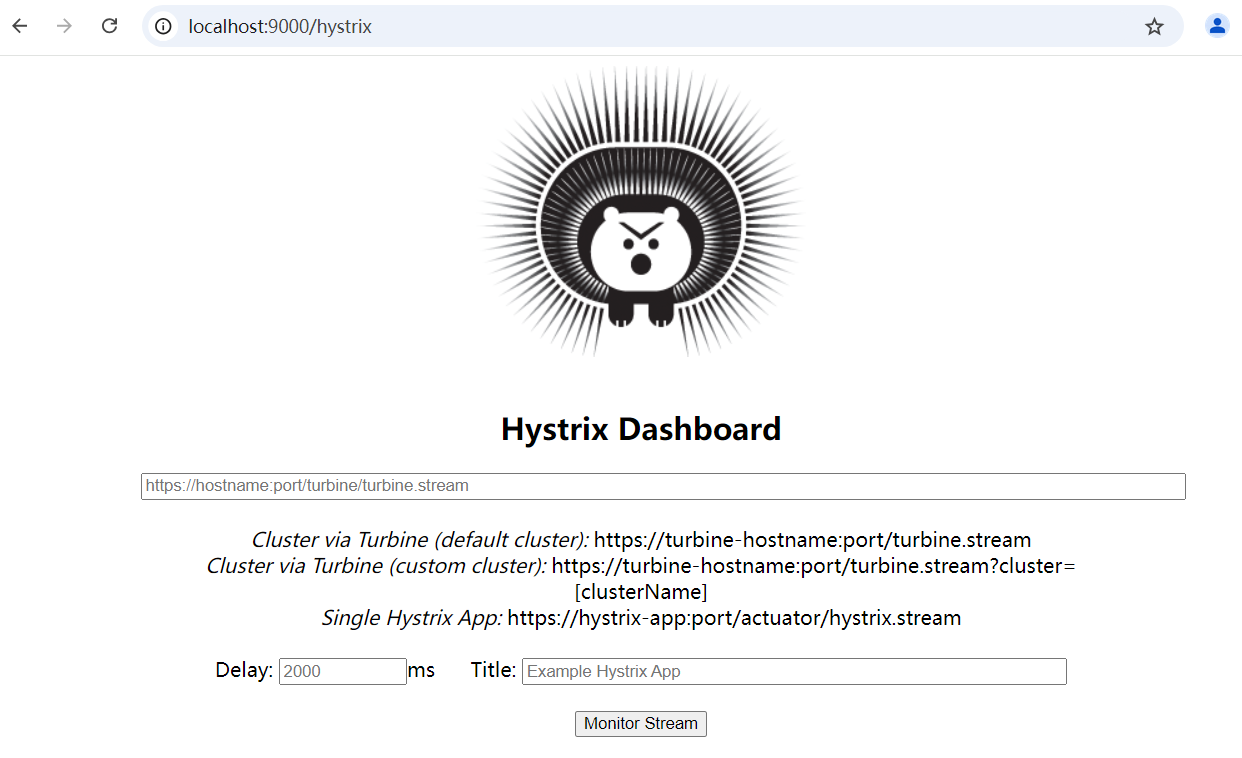
需要加上 @EnableHystrixDashboard 注解，开启Hystrix仪表盘功能

// 启用 Hystrix 仪表盘功能  
@EnableHystrixDashboard  
@SpringBootApplication  
public class DashboardMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(DashboardMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

#### 查看监控数据

1.访问监控首页，需要在ip和端口后面加上 hystrix

http://localhost:9000/hystrix



2.访问监控数据，provider工程

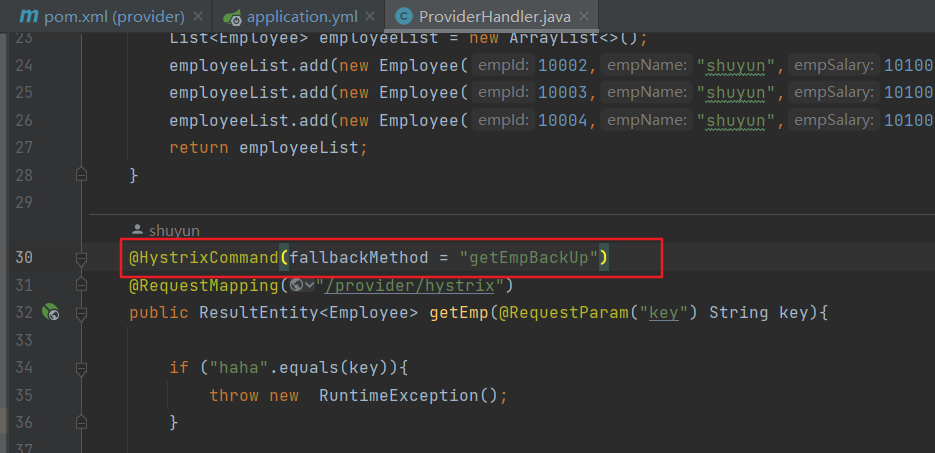
固定格式： http://localhost:1000/actuator/hystrix.stream

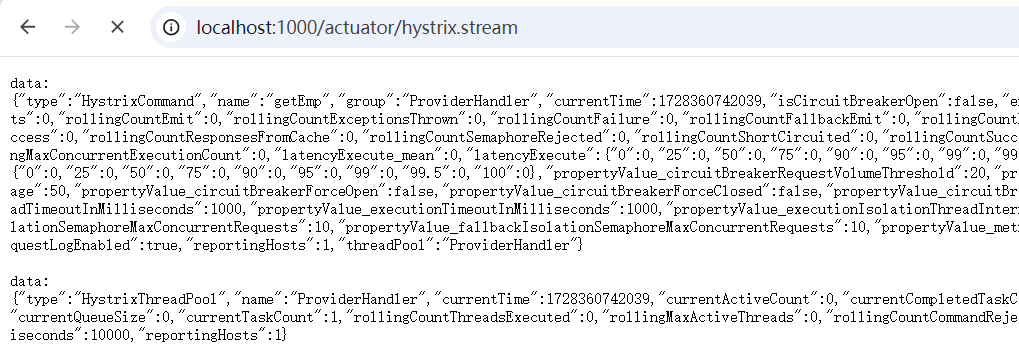
说明 1：http://localhost:1000 访问的是被监控的 provider 工程

说明 2：/actuator/hystrix.stream 是固定格式

说明 3：如果从 provider 启动开始它的方法没有被访问过，那么显示的数据只有“ping:”，要实际访问一个带熔断功能的方法才会有实际数据。

访问带有熔断功能的方法后出现的界面



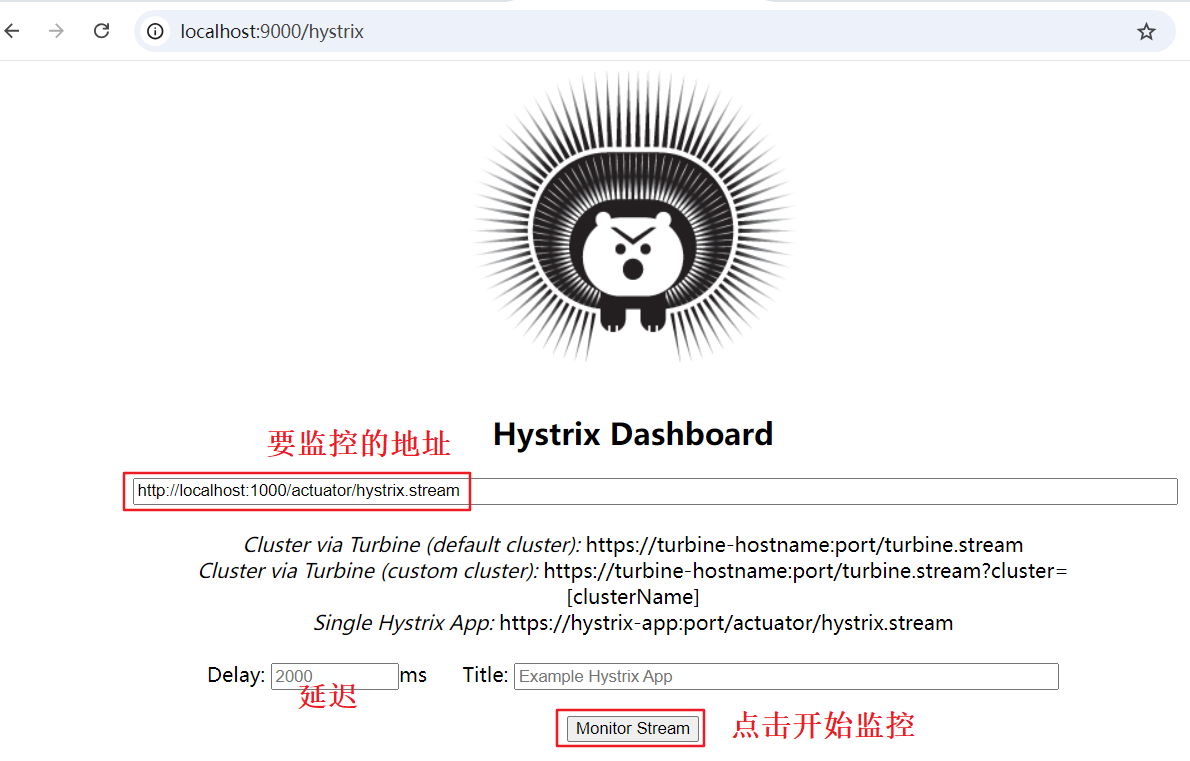


#### 通过仪表盘查看监控数据

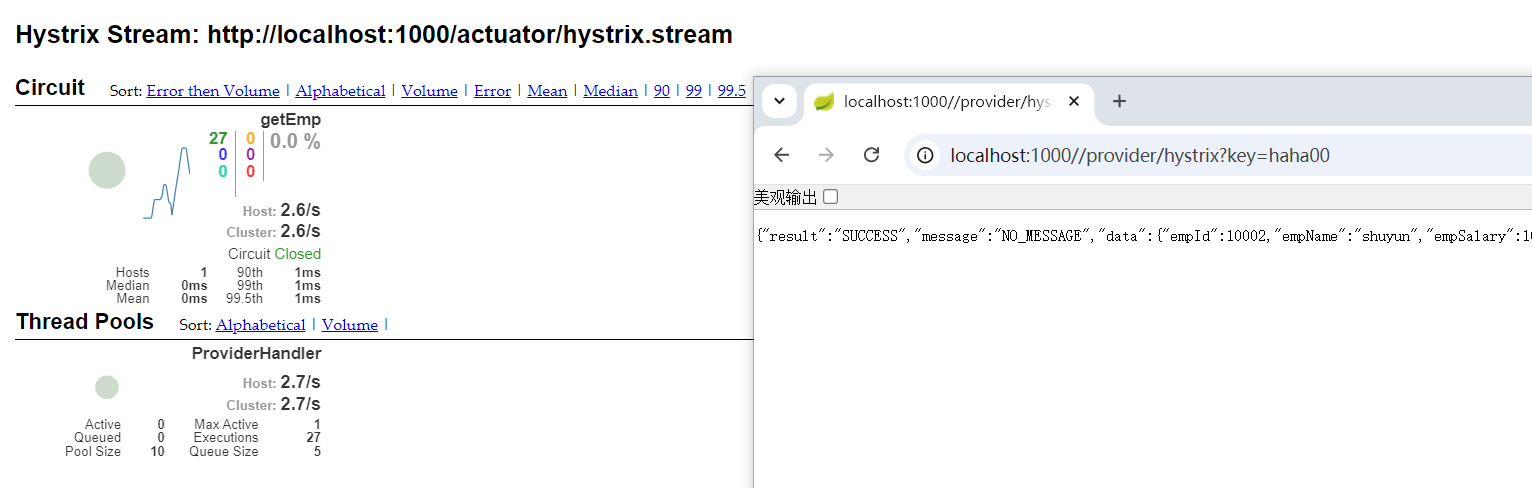
1.打开仪表盘首页

http://localhost:9000/hystrix

2.填写需要监控的地址



3.效果



## Zuul网关

### 简介

不同的微服务一般有不同的网络地址，而外部的客户端可能需要调用多个服务的接口才能完成一个业务需求。比如一个电影购票的手机 APP，可能会调用电影分类微服务，用户微服务，支付微服务等。如果客户端直接和微服务进行通信，会存在以下问题：

1.客户端会多次请求不同微服务，增加客户端的复杂性 

2.存在跨域请求，在一定场景下处理相对复杂 

3.认证复杂，每一个服务都需要独立认证 

4.难以重构，随着项目的迭代，可能需要重新划分微服务，如果客户端直接和微服务通信，那么重构会难以实施 

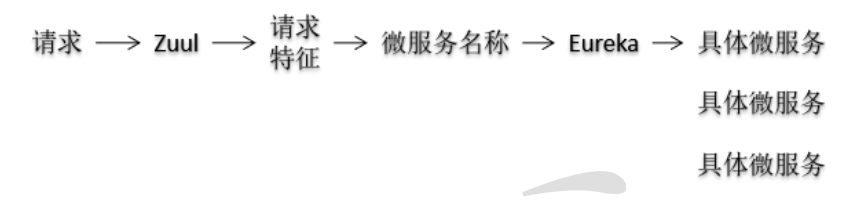
5.某些微服务可能使用了其他协议，直接访问有一定困难

Zuul 包含了对请求的路由和过滤两个最主要的功能：

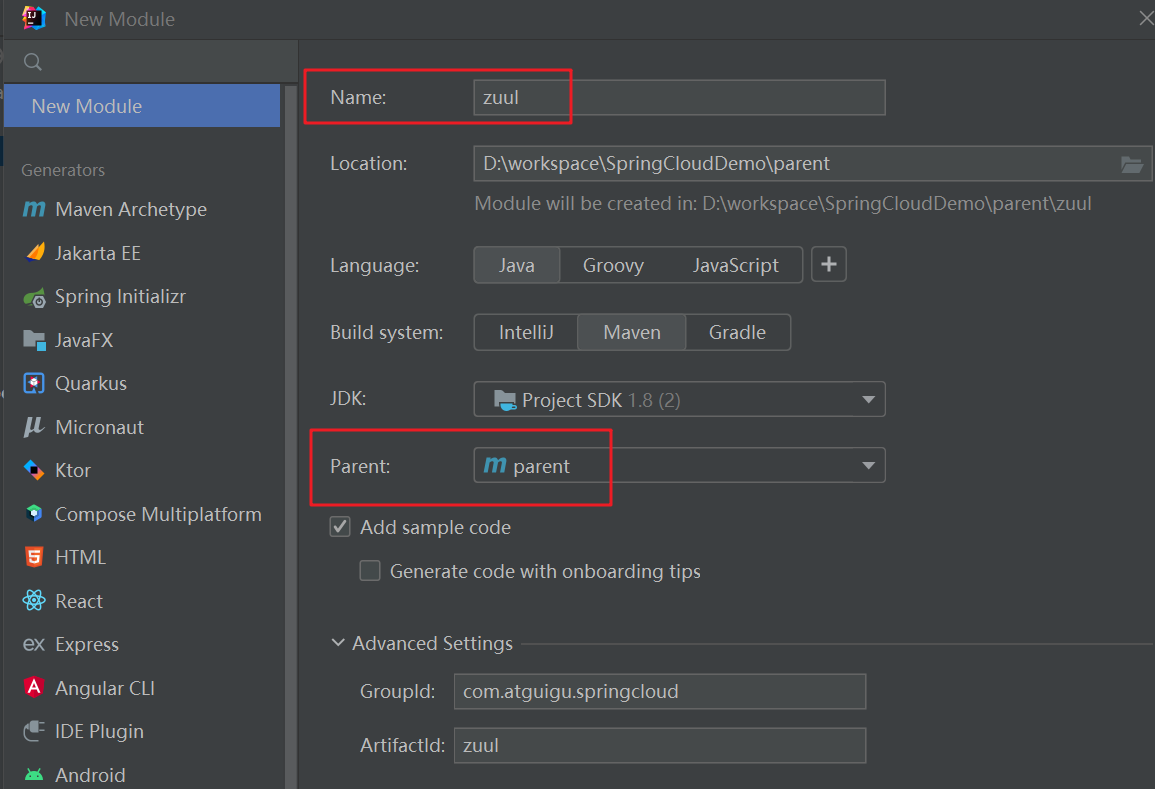
其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础。

Zuul 和 Eureka 进行整合，将 Zuul 自身注册为 Eureka 服务治理下的应用，同时从Eureka中获得其他微服务的信息，也即以后的访问微服务都是通过 Zuul 跳转后获得。

总体来说，Zuul 提供了代理、路由和过滤的功能。



### 创建Zuul工程



依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
 </dependency>  
</dependencies>

application.yml配置

server:  
 port: 3103  
  
*#微服务名称*spring:  
 application:  
 name: zuul  
  
eureka:  
 client:  
 service-url:  
 *#指定eureka注册中心的服务器地址* defaultZone: http://localhost:5000/eureka/

主启动类，加上@EnableZuulProxy注解，启用zuul代理功能

// 启用 Zuul 代理功能  
@EnableZuulProxy  
@SpringBootApplication  
public class ZuulMain {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ZuulMain.class);  
 System.*out*.println("Hello world!");  
 }  
}

### Zuul访问测试

#### 初步访问

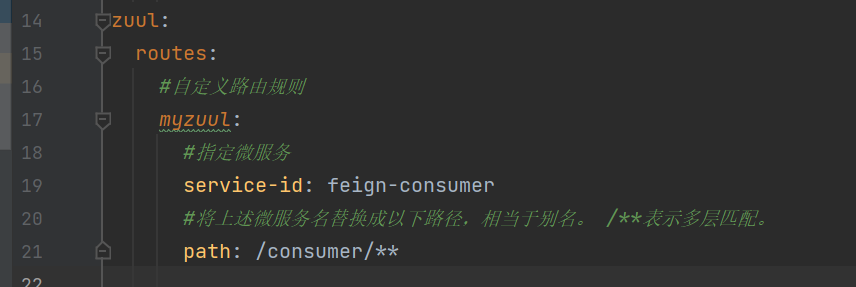


此时：通过 Zuul 可以访问，也可以不经过 Zuul 直接访问目标微服务。



#### 使用指定地址代替微服务名称

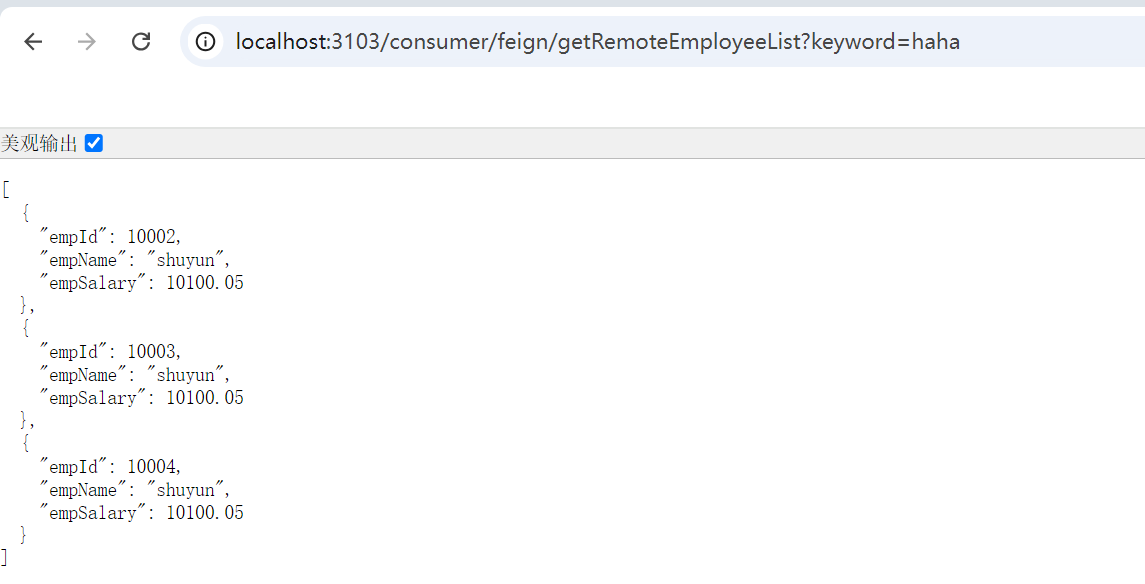
application.yml配置



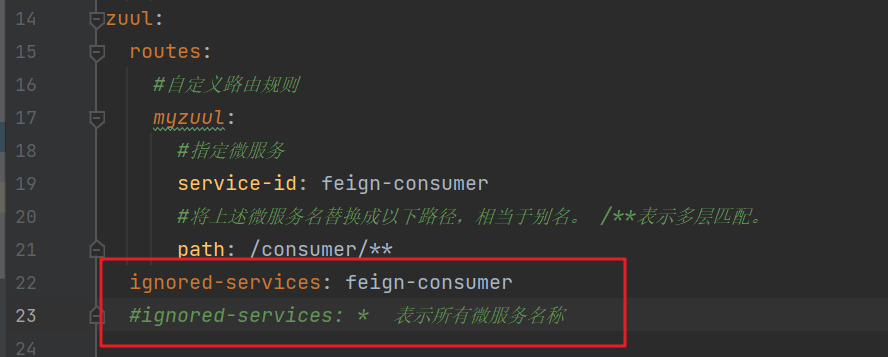
效果：使用微服务名称和新配置的地址都可以访问

http://localhost:3103/feign-consumer/feign/getRemoteEmployeeList?keyword=haha

http://localhost:3103/consumer/feign/getRemoteEmployeeList?keyword=haha



#### 不能通过微服务名称访问



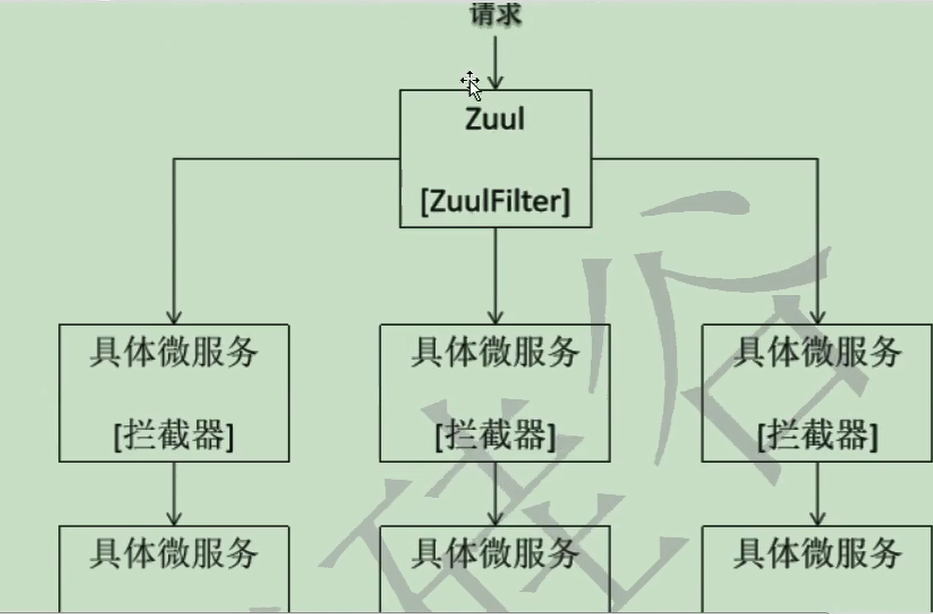
#### 忽略所有微服务名称



#### 给访问路径添加统一前缀



### ZuulFilter



shouldFilter() 方法，返回true表示执行run()方法逻辑，返回false表示放行

run()方法，表示具体的过滤逻辑，返回值为Null即可，会忽略这个返回值

filterType()方法，表示过滤器执行的时机，pre表示在要在目标微服务前面执行过滤操作

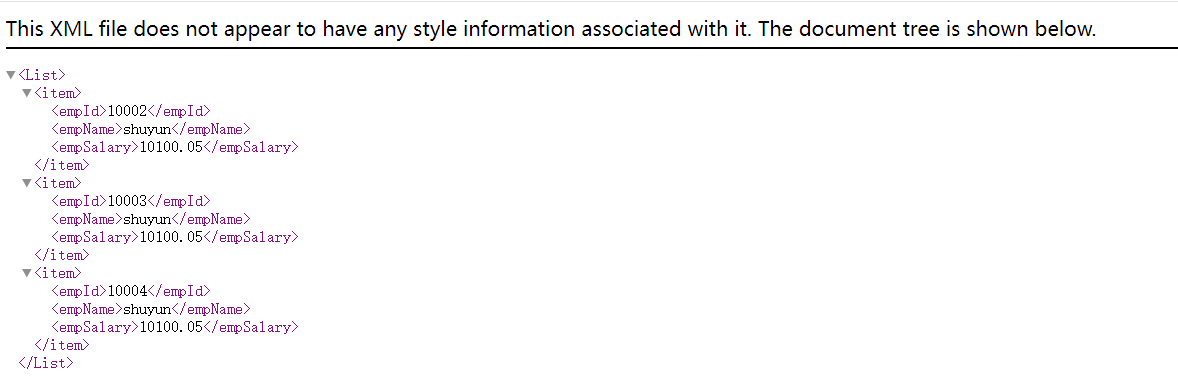
filterOrder() 方法，表示过滤器执行顺序，返回0即可

## 注意事项

provider和consumer工程，eureka的依赖是引入client

eureka工程，引入的是server

搞错了虽然也能注册上，但是会导致返回的报文是xml格式，不是json格式



# 会员系统

## 总体目标

## 整体架构

## 工程概述

# 环境搭建

## 约定

## eureka工程

## entity工程

## MySQL工程

## Redis工程

## 认证工程

## 网关

# 会员登录

# 发起项目

# 展示项目

# 支付案例

# 订单

# 支付

# 部署

# 总结

# 最后