

-

整理自互联网，仅供学习交流。



2018-10-30

[公司名称]

[公司地址]

目录

**未找到目录项。**

123

# 黑马18

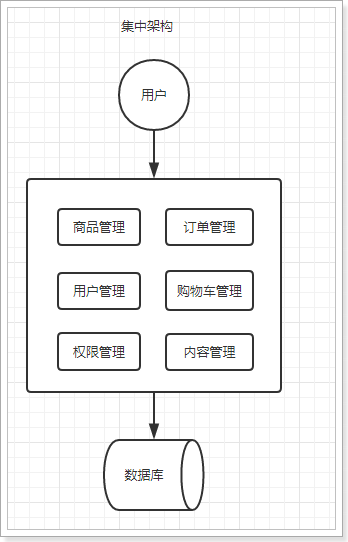
## 系统架构演变

随着互联网的发展，网站应用的规模不断扩大。需求的激增，带来的是技术上的压力。系统架构也因此也不断的演进、升级、迭代。从单一应用，到垂直拆分，到分布式服务，到SOA，以及现在火热的微服务架构，还有在Google带领下来势汹涌的Service Mesh。我们到底是该乘坐微服务的船只驶向远方，还是偏安一隅得过且过？

其实生活不止眼前的苟且，还有诗和远方。所以我们今天就回顾历史，看一看系统架构演变的历程；把握现在，学习现在最火的技术架构；展望未来，争取成为一名优秀的Java工程师。

### 1.1. 集中式架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是影响项目开发的关键。



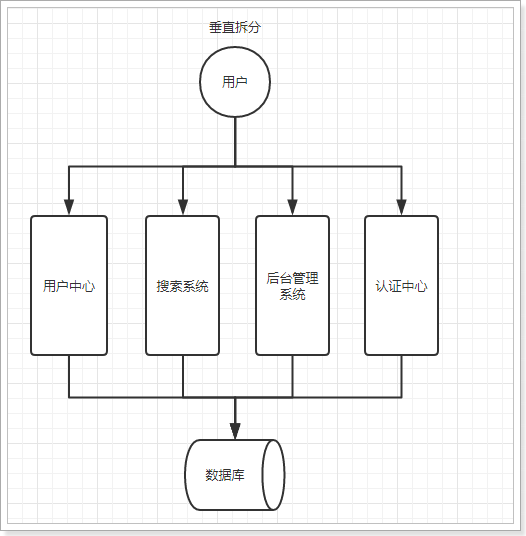
1525529091749

存在的问题：

* 代码耦合，开发维护困难
* 无法针对不同模块进行针对性优化
* 无法水平扩展
* 单点容错率低，并发能力差

### 1.2.垂直拆分

当访问量逐渐增大，单一应用无法满足需求，此时为了应对更高的并发和业务需求，我们根据业务功能对系统进行拆分：



1525529671801

优点：

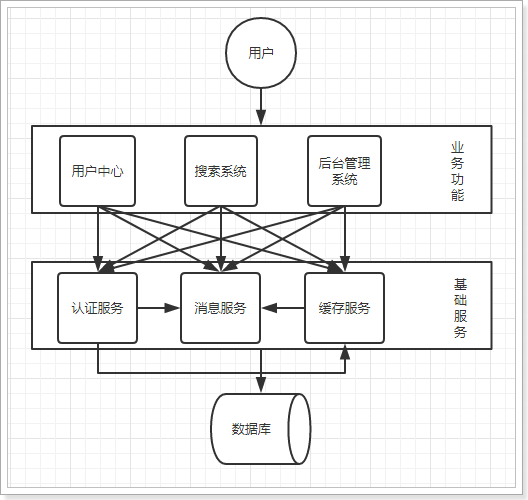
* 系统拆分实现了流量分担，解决了并发问题
* 可以针对不同模块进行优化
* 方便水平扩展，负载均衡，容错率提高

缺点：

* 系统间相互独立，会有很多重复开发工作，影响开发效率

### 1.3.分布式服务

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式调用是关键。



1525530657919

优点：

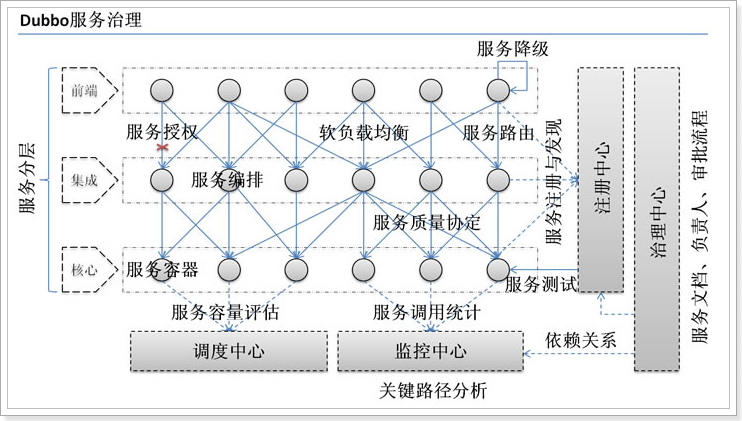
* 将基础服务进行了抽取，系统间相互调用，提高了代码复用和开发效率

缺点：

* 系统间耦合度变高，调用关系错综复杂，难以维护

### 1.4.服务治理（SOA）

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键



1525530804753

以前出现了什么问题？

* 服务越来越多，需要管理每个服务的地址
* 调用关系错综复杂，难以理清依赖关系
* 服务过多，服务状态难以管理，无法根据服务情况动态管理

服务治理要做什么？

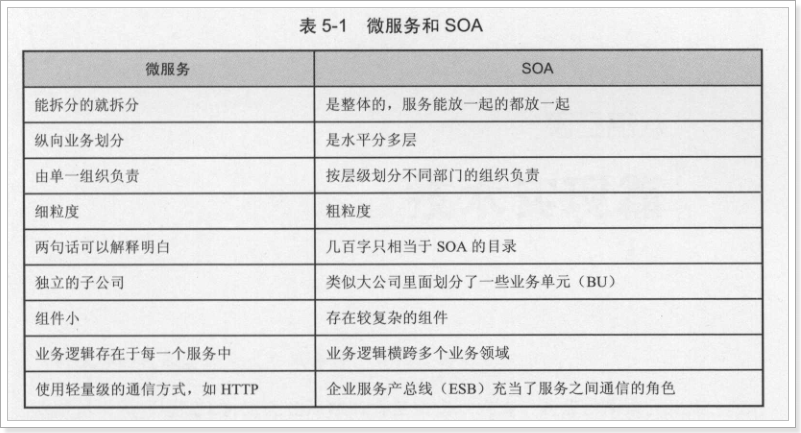
* 服务注册中心，实现服务自动注册和发现，无需人为记录服务地址
* 服务自动订阅，服务列表自动推送，服务调用透明化，无需关心依赖关系
* 动态监控服务状态监控报告，人为控制服务状态

缺点：

* 服务间会有依赖关系，一旦某个环节出错会影响较大
* 服务关系复杂，运维、测试部署困难，不符合DevOps思想

### 1.5.微服务

前面说的SOA，英文翻译过来是面向服务。微服务，似乎也是服务，都是对系统进行拆分。因此两者非常容易混淆，但其实缺有一些差别：

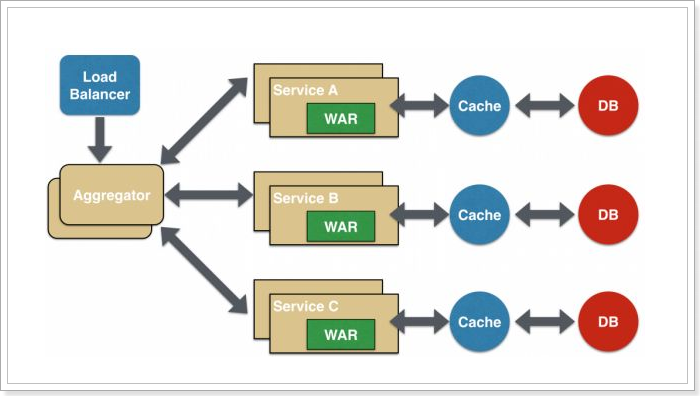


1525532344817

微服务的特点：

* 单一职责：微服务中每一个服务都对应唯一的业务能力，做到单一职责
* 微：微服务的服务拆分粒度很小，例如一个用户管理就可以作为一个服务。每个服务虽小，但“五脏俱全”。
* 面向服务：面向服务是说每个服务都要对外暴露服务接口API。并不关心服务的技术实现，做到与平台和语言无关，也不限定用什么技术实现，只要提供Rest的接口即可。
* 自治：自治是说服务间互相独立，互不干扰
* 团队独立：每个服务都是一个独立的开发团队，人数不能过多。
* 技术独立：因为是面向服务，提供Rest接口，使用什么技术没有别人干涉
* 前后端分离：采用前后端分离开发，提供统一Rest接口，后端不用再为PC、移动段开发不同接口
* 数据库分离：每个服务都使用自己的数据源
* 部署独立，服务间虽然有调用，但要做到服务重启不影响其它服务。有利于持续集成和持续交付。每个服务都是独立的组件，可复用，可替换，降低耦合，易维护

微服务结构图：



1526860071166

## 远程调用方式

无论是微服务还是SOA，都面临着服务间的远程调用。那么服务间的远程调用方式有哪些呢？

常见的远程调用方式有以下几种：

* RPC：Remote Produce Call远程过程调用，类似的还有RMI。自定义数据格式，基于原生TCP通信，速度快，效率高。早期的webservice，现在热门的dubbo，都是RPC的典型
* Http：http其实是一种网络传输协议，基于TCP，规定了数据传输的格式。现在客户端浏览器与服务端通信基本都是采用Http协议。也可以用来进行远程服务调用。缺点是消息封装臃肿。

现在热门的Rest风格，就可以通过http协议来实现。

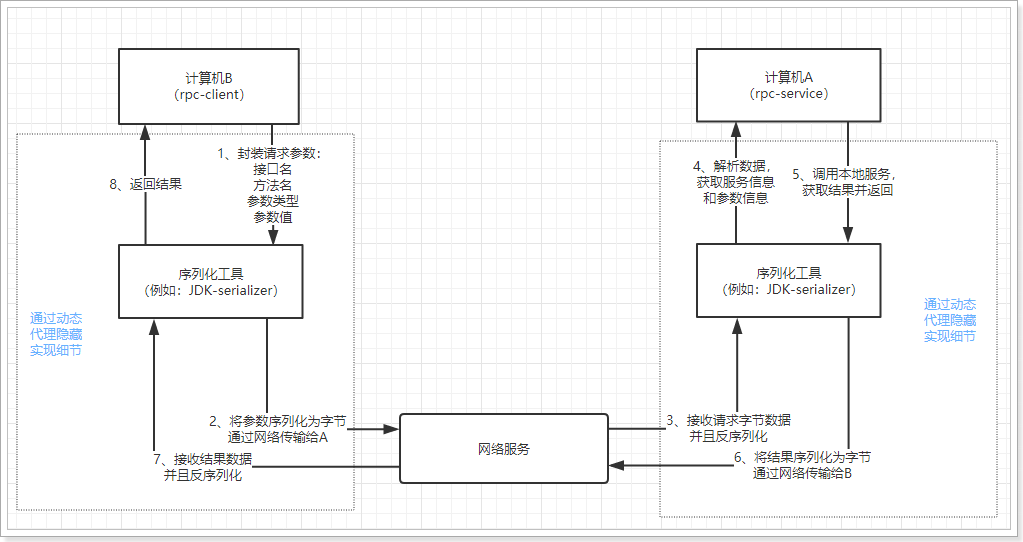
### 2.1.认识RPC

RPC，即 Remote Procedure Call（远程过程调用），是一个计算机通信协议。 该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程。说得通俗一点就是：A计算机提供一个服务，B计算机可以像调用本地服务那样调用A计算机的服务。

通过上面的概念，我们可以知道，实现RPC主要是做到两点：

* 实现远程调用其他计算机的服务
* 要实现远程调用，肯定是通过网络传输数据。A程序提供服务，B程序通过网络将请求参数传递给A，A本地执行后得到结果，再将结果返回给B程序。这里需要关注的有两点：
  + 1）采用何种网络通讯协议？
  + 现在比较流行的RPC框架，都会采用TCP作为底层传输协议
  + 2）数据传输的格式怎样？
  + 两个程序进行通讯，必须约定好数据传输格式。就好比两个人聊天，要用同一种语言，否则无法沟通。所以，我们必须定义好请求和响应的格式。另外，数据在网路中传输需要进行序列化，所以还需要约定统一的序列化的方式。
* 像调用本地服务一样调用远程服务
* 如果仅仅是远程调用，还不算是RPC，因为RPC强调的是过程调用，调用的过程对用户而言是应该是透明的，用户不应该关心调用的细节，可以像调用本地服务一样调用远程服务。所以RPC一定要对调用的过程进行封装

RPC调用流程图：



1525568965976

想要了解详细的RPC实现，给大家推荐一篇文章：[自己动手实现RPC](https://legacy.gitbook.com/book/huge0612/tour-of-rpc/details)

### 2.2.认识Http

Http协议：超文本传输协议，是一种应用层协议。规定了网络传输的请求格式、响应格式、资源定位和操作的方式等。但是底层采用什么网络传输协议，并没有规定，不过现在都是采用TCP协议作为底层传输协议。说到这里，大家可能觉得，Http与RPC的远程调用非常像，都是按照某种规定好的数据格式进行网络通信，有请求，有响应。没错，在这点来看，两者非常相似，但是还是有一些细微差别。

* RPC并没有规定数据传输格式，这个格式可以任意指定，不同的RPC协议，数据格式不一定相同。
* Http中还定义了资源定位的路径，RPC中并不需要
* 最重要的一点：RPC需要满足像调用本地服务一样调用远程服务，也就是对调用过程在API层面进行封装。Http协议没有这样的要求，因此请求、响应等细节需要我们自己去实现。
* 优点：RPC方式更加透明，对用户更方便。Http方式更灵活，没有规定API和语言，跨语言、跨平台
* 缺点：RPC方式需要在API层面进行封装，限制了开发的语言环境。

例如我们通过浏览器访问网站，就是通过Http协议。只不过浏览器把请求封装，发起请求以及接收响应，解析响应的事情都帮我们做了。如果是不通过浏览器，那么这些事情都需要自己去完成。



1525569352313

### 2.3.如何选择？

既然两种方式都可以实现远程调用，我们该如何选择呢？

* 速度来看，RPC要比http更快，虽然底层都是TCP，但是http协议的信息往往比较臃肿，不过可以采用gzip压缩。
* 难度来看，RPC实现较为复杂，http相对比较简单
* 灵活性来看，http更胜一筹，因为它不关心实现细节，跨平台、跨语言。

因此，两者都有不同的使用场景：

* 如果对效率要求更高，并且开发过程使用统一的技术栈，那么用RPC还是不错的。
* 如果需要更加灵活，跨语言、跨平台，显然http更合适

那么我们该怎么选择呢？

微服务，更加强调的是独立、自治、灵活。而RPC方式的限制较多，因此微服务框架中，一般都会采用基于Http的Rest风格服务。

## Http客户端工具

既然微服务选择了Http，那么我们就需要考虑自己来实现对请求和响应的处理。不过开源世界已经有很多的http客户端工具，能够帮助我们做这些事情，例如：

* HttpClient
* OKHttp
* URLConnection

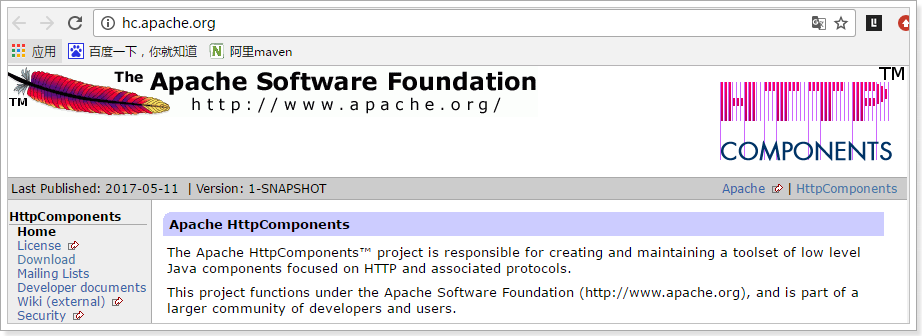
接下来，我们就一起了解一款比较流行的客户端工具：HttpClient

### 3.1.HttpClient

#### 3.1.1.介绍

HttpClient是Apache公司的产品，是Http Components下的一个组件。

[官网地址：http://hc.apache.org/index.html](http://hc.apache.org/index.html)



特点：

* 基于标准、纯净的Java语言。实现了Http1.0和Http1.1
* 以可扩展的面向对象的结构实现了Http全部的方法（GET, POST, PUT, DELETE, HEAD, OPTIONS, and TRACE）
* 支持HTTPS协议。
* 通过Http代理建立透明的连接。
* 自动处理Set-Cookie中的Cookie。

#### 3.1.2.使用

我们导入课前资料提供的demo工程：《http-demo》

发起get请求：

@Test  
 public void testGet() throws IOException {  
 HttpGet request = new HttpGet("http://www.baidu.com");  
 String response = this.httpClient.execute(request, new BasicResponseHandler());  
 System.out.println(response);  
 }

发起Post请求：

@Test  
public void testPost() throws IOException {  
 HttpPost request = new HttpPost("http://www.oschina.net/");  
 request.setHeader("User-Agent",  
 "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36");  
 String response = this.httpClient.execute(request, new BasicResponseHandler());  
 System.out.println(response);  
}

尝试访问昨天编写的接口：http://localhost/hello

这个接口返回一个User对象

@Test  
public void testGetPojo() throws IOException {  
 HttpGet request = new HttpGet("http://localhost/hello");  
 String response = this.httpClient.execute(request, new BasicResponseHandler());  
 System.out.println(response);  
}

我们实际得到的是一个json字符串：

{  
 "id": 8,  
 "userName": "liuyan",  
 "password": "123456",  
 "name": "柳岩",  
 "age": 21,  
 "sex": 2,  
 "birthday": "1995-08-07T16:00:00.000+0000",  
 "created": "2014-09-20T03:41:15.000+0000",  
 "updated": "2014-09-20T03:41:15.000+0000",  
 "note": "柳岩同学在传智播客学表演"  
}

如果想要得到对象，我们还需要手动进行Json反序列化，这一点比较麻烦。

#### 3.1.3.Json转换工具

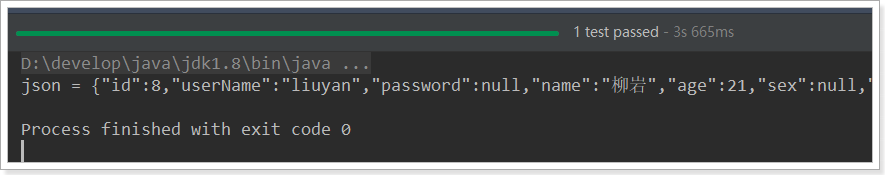
HttpClient请求数据后是json字符串，需要我们自己把Json字符串反序列化为对象，我们会使用JacksonJson工具来实现。

JacksonJson是SpringMVC内置的json处理工具，其中有一个ObjectMapper类，可以方便的实现对json的处理：

##### 对象转json

// json处理工具  
 private ObjectMapper mapper = new ObjectMapper();  
 @Test  
 public void testJson() throws JsonProcessingException {  
 User user = new User();  
 user.setId(8L);  
 user.setAge(21);  
 user.setName("柳岩");  
 user.setUserName("liuyan");  
 // 序列化  
 String json = mapper.writeValueAsString(user);  
 System.out.println("json = " + json);  
 }

结果：

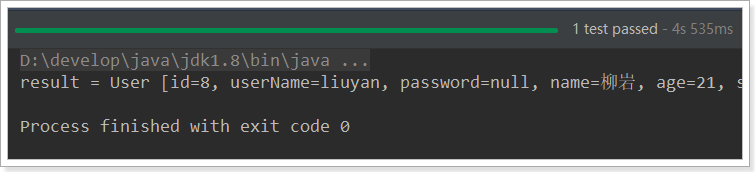


1526877496885

##### json转普通对象

// json处理工具  
private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
 User user = new User();  
 user.setId(8L);  
 user.setAge(21);  
 user.setName("柳岩");  
 user.setUserName("liuyan");  
 // 序列化  
 String json = mapper.writeValueAsString(user);  
  
 // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
 User result = mapper.readValue(json, User.class);  
 System.out.println("result = " + result);  
}

结果：



1526877647406

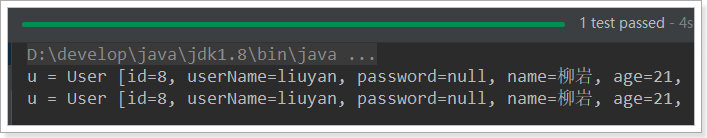
##### json转集合

json转集合比较麻烦，因为你无法同时把集合的class和元素的class同时传递到一个参数。

因此Jackson做了一个类型工厂，用来解决这个问题：

// json处理工具  
private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
 User user = new User();  
 user.setId(8L);  
 user.setAge(21);  
 user.setName("柳岩");  
 user.setUserName("liuyan");  
  
 // 序列化,得到对象集合的json字符串  
 String json = mapper.writeValueAsString(Arrays.asList(user, user));  
  
 // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
 List<User> users = mapper.readValue(json, mapper.getTypeFactory().constructCollectionType(List.class, User.class));  
 for (User u : users) {  
 System.out.println("u = " + u);  
 }  
}

结果：



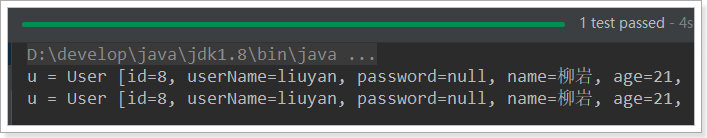
1526877995530

##### json转任意复杂类型

当对象泛型关系复杂时，类型工厂也不好使了。这个时候Jackson提供了TypeReference来接收类型泛型，然后底层通过反射来获取泛型上的具体类型。实现数据转换。

// json处理工具  
private ObjectMapper mapper = new ObjectMapper();  
@Test  
public void testJson() throws IOException {  
 User user = new User();  
 user.setId(8L);  
 user.setAge(21);  
 user.setName("柳岩");  
 user.setUserName("liuyan");  
  
 // 序列化,得到对象集合的json字符串  
 String json = mapper.writeValueAsString(Arrays.asList(user, user));  
  
 // 反序列化，接收两个参数：json数据，反序列化的目标类字节码  
 List<User> users = mapper.readValue(json, new TypeReference<List<User>>(){});  
 for (User u : users) {  
 System.out.println("u = " + u);  
 }  
}

结果：



1526877988488

### 3.3.Spring的RestTemplate

Spring提供了一个RestTemplate模板工具类，对基于Http的客户端进行了封装，并且实现了对象与json的序列化和反序列化，非常方便。RestTemplate并没有限定Http的客户端类型，而是进行了抽象，目前常用的3种都有支持：

* HttpClient
* OkHttp
* JDK原生的URLConnection（默认的）

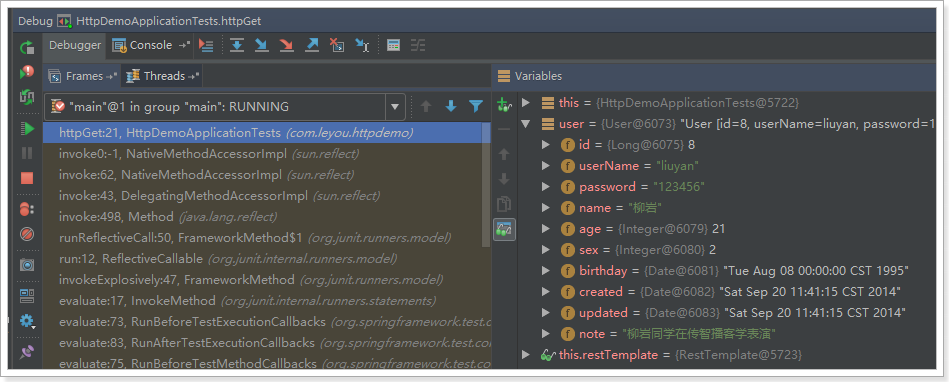
首先在项目中注册一个RestTemplate对象，可以在启动类位置注册：

@SpringBootApplication  
public class HttpDemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(HttpDemoApplication.class, args);  
 }  
  
 @Bean  
 public RestTemplate restTemplate() {  
 // 默认的RestTemplate，底层是走JDK的URLConnection方式。  
 return new RestTemplate();  
 }  
}

在测试类中直接@Autowired注入：

@RunWith(SpringRunner.class)  
@SpringBootTest(classes = HttpDemoApplication.class)  
public class HttpDemoApplicationTests {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @Test  
 public void httpGet() {  
 User user = this.restTemplate.getForObject("http://localhost/hello", User.class);  
 System.out.println(user);  
 }  
}

* 通过RestTemplate的getForObject()方法，传递url地址及实体类的字节码，RestTemplate会自动发起请求，接收响应，并且帮我们对响应结果进行反序列化。



1525573702492

学习完了Http客户端工具，接下来就可以正式学习微服务了。

## 初始SpringCloud

微服务是一种架构方式，最终肯定需要技术架构去实施。

微服务的实现方式很多，但是最火的莫过于Spring Cloud了。为什么？

* 后台硬：作为Spring家族的一员，有整个Spring全家桶靠山，背景十分强大。
* 技术强：Spring作为Java领域的前辈，可以说是功力深厚。有强力的技术团队支撑，一般人还真比不了
* 群众基础好：可以说大多数程序员的成长都伴随着Spring框架，试问：现在有几家公司开发不用Spring？SpringCloud与Spring的各个框架无缝整合，对大家来说一切都是熟悉的配方，熟悉的味道。
* 使用方便：相信大家都体会到了SpringBoot给我们开发带来的便利，而SpringCloud完全支持SpringBoot的开发，用很少的配置就能完成微服务框架的搭建

### 4.1.简介

SpringCloud是Spring旗下的项目之一，[官网地址：http://projects.spring.io/spring-cloud/](http://projects.spring.io/spring-cloud/)

Spring最擅长的就是集成，把世界上最好的框架拿过来，集成到自己的项目中。

SpringCloud也是一样，它将现在非常流行的一些技术整合到一起，实现了诸如：配置管理，服务发现，智能路由，负载均衡，熔断器，控制总线，集群状态等等功能。其主要涉及的组件包括：

netflix

* Eureka：注册中心
* Zuul：服务网关
* Ribbon：负载均衡
* Feign：服务调用
* Hystix：熔断器

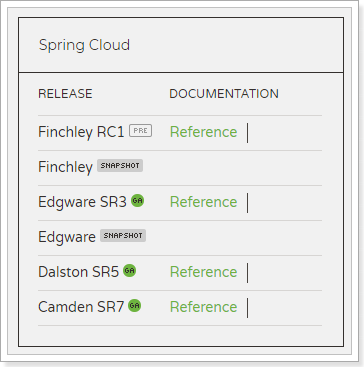
以上只是其中一部分，架构图：



1525575656796

### 4.2.版本

SpringCloud的版本命名比较特殊，因为它不是一个组件，而是许多组件的集合，它的命名是以A到Z的为首字母的一些单词组成：



我们在项目中，会是以Finchley的版本。

其中包含的组件，也都有各自的版本，如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Edgware.SR3 | Finchley.RC1 | Finchley.BUILD-SNAPSHOT |
| spring-cloud-aws | 1.2.2.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-bus | 1.3.2.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-cli | 1.4.1.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-commons | 1.3.3.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-contract | 1.2.4.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-config | 1.4.3.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-netflix | 1.4.4.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-security | 1.2.2.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-cloudfoundry | 1.1.1.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-consul | 1.3.3.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-sleuth | 1.3.3.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-stream | Ditmars.SR3 | Elmhurst.RELEASE | Elmhurst.BUILD-SNAPSHOT |
| spring-cloud-zookeeper | 1.2.1.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-boot | 1.5.10.RELEASE | 2.0.1.RELEASE | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-task | 1.2.2.RELEASE | 2.0.0.RC1 | 2.0.0.RELEASE |
| spring-cloud-vault | 1.1.0.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-gateway | 1.0.1.RELEASE | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |
| spring-cloud-openfeign |  | 2.0.0.RC1 | 2.0.0.BUILD-SNAPSHOT |

#### spring boot版本和spring cloud版本的匹配关系：

| **Spring Cloud** | **Spring Boot** |
| --- | --- |
| Finchley | 兼容Spring Boot 2.0.x，不兼容Spring Boot 1.5.x |
| Dalston和Edgware | 兼容Spring Boot 1.5.x，不兼容Spring Boot 2.0.x |
| Camden | 兼容Spring Boot 1.4.x，也兼容Spring Boot 1.5.x |
| Brixton | 兼容Spring Boot 1.3.x，也兼容Spring Boot 1.4.x |
| Angel | 兼容Spring Boot 1.2.x |

接下来，我们就一一学习SpringCloud中的重要组件。

## 微服务场景模拟

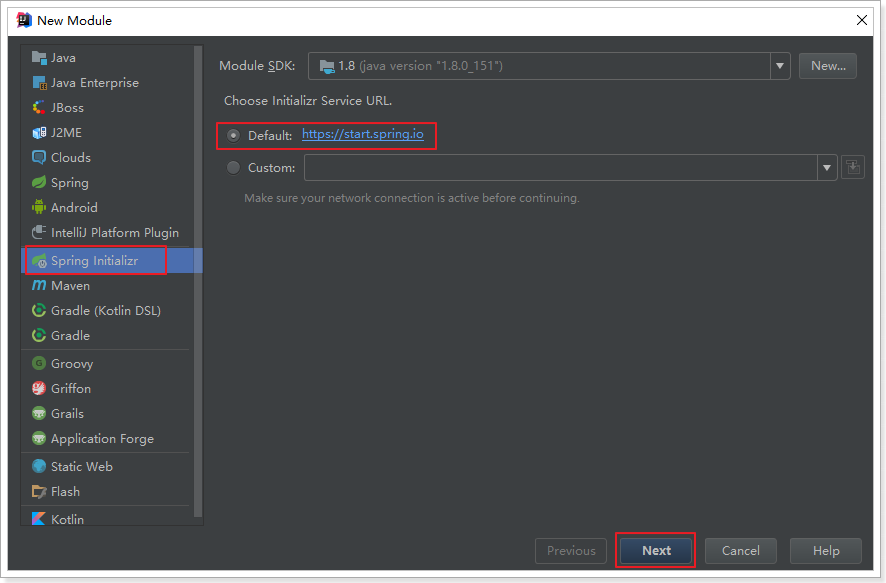
首先，我们需要模拟一个服务调用的场景。方便后面学习微服务架构

### 5.1.服务提供者

我们新建一个项目，对外提供查询用户的服务。

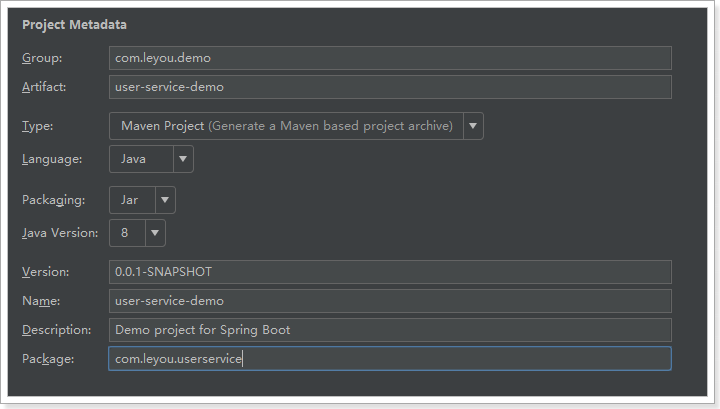
#### 5.1.1.Spring脚手架创建工程

借助于Spring提供的快速搭建工具：



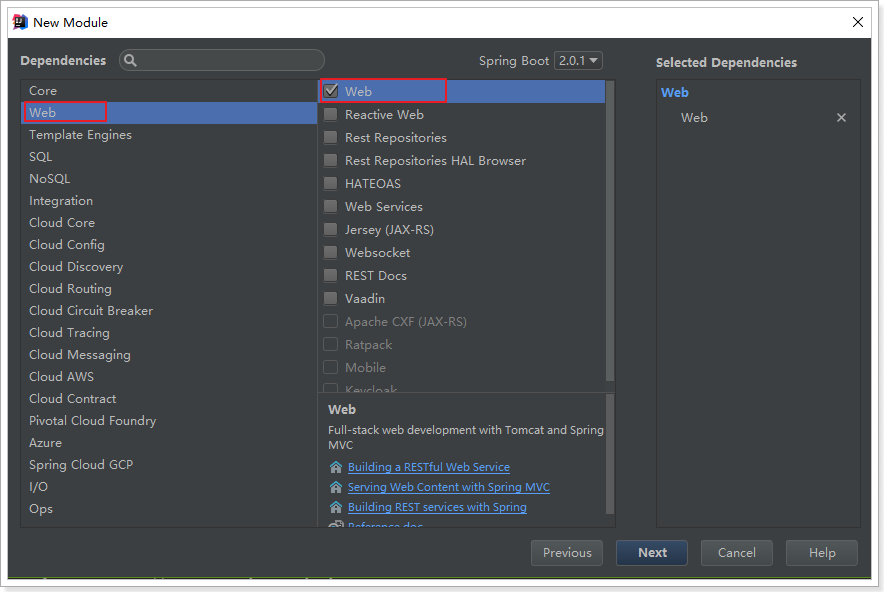
1525576816916

填写项目信息：



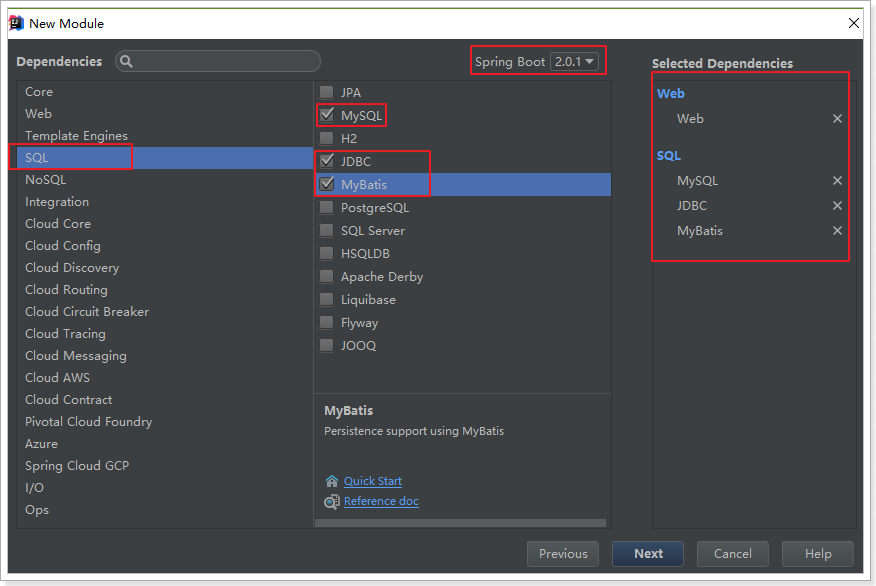
1525576909381

添加web依赖：



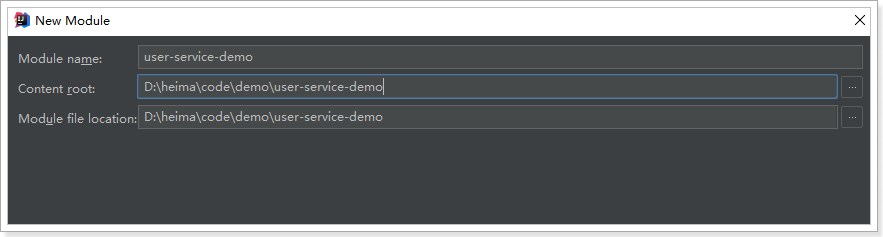
1525576950842

添加mybatis依赖：



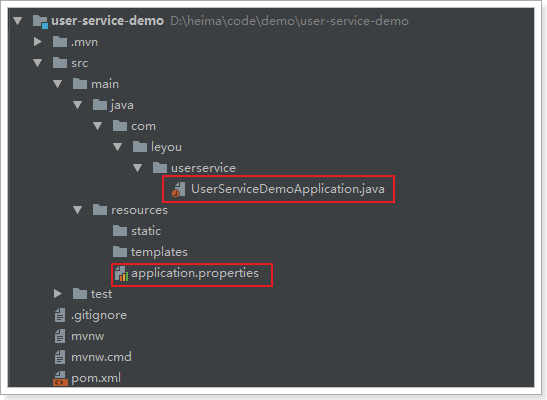
1525576999052

填写项目位置：



1525577029150

生成的项目结构：



1525577106711

依赖也已经全部自动引入：

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.leyou.demo</groupId>  
 <artifactId>user-service-demo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <name>user-service-demo</name>  
 <description>Demo project for Spring Boot</description>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.1.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 <version>1.3.2</version>  
 </dependency>  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

当然，因为要使用通用mapper，所以我们需要手动加一条依赖：

<dependency>  
 <groupId>tk.mybatis</groupId>  
 <artifactId>mapper-spring-boot-starter</artifactId>  
 <version>2.0.2</version>  
</dependency>

非常快捷啊！

#### 5.1.2.编写代码

添加一个对外查询的接口：

@RestController  
@RequestMapping("user")  
public class UserController {  
  
 @Autowired  
 private UserService userService;  
  
 @GetMapping("/{id}")  
 public User queryById(@PathVariable("id") Long id) {  
 return this.userService.queryById(id);  
 }  
}

Service：

@Service  
public class UserService {  
  
 @Autowired  
 private UserMapper userMapper;  
  
 public User queryById(Long id) {  
 return this.userMapper.selectByPrimaryKey(id);  
 }  
}

mapper:

@Mapper  
public interface UserMapper extends tk.mybatis.mapper.common.Mapper<User>{  
}

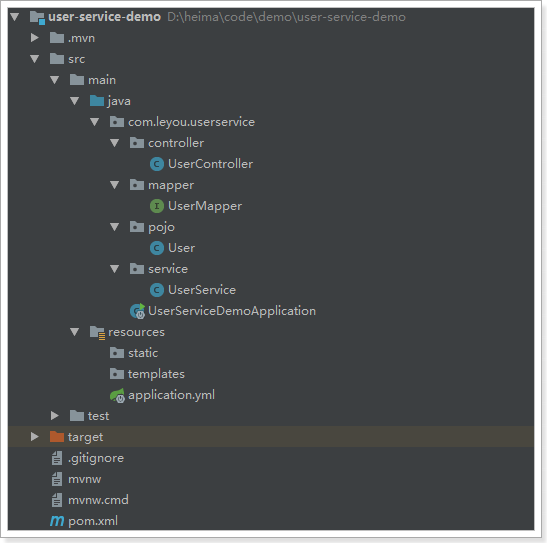
实体类：

@Table(name = "tb\_user")  
public class User implements Serializable {  
  
 private static final long serialVersionUID = 1L;  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 // 用户名  
 private String userName;  
  
 // 密码  
 private String password;  
  
 // 姓名  
 private String name;  
  
 // 年龄  
 private Integer age;  
  
 // 性别，1男性，2女性  
 private Integer sex;  
  
 // 出生日期  
 private Date birthday;  
  
 // 创建时间  
 private Date created;  
  
 // 更新时间  
 private Date updated;  
  
 // 备注  
 private String note;  
  
 // 。。。省略getters和setters  
}

属性文件,这里我们采用了yaml语法，而不是properties：

server:  
 port: 8081  
spring:  
 datasource:  
 url: jdbc:mysql://localhost:3306/mydb01  
 username: root  
 password: 123  
 hikari:  
 maximum-pool-size: 20  
 minimum-idle: 10  
mybatis:  
 type-aliases-package: com.leyou.userservice.pojo

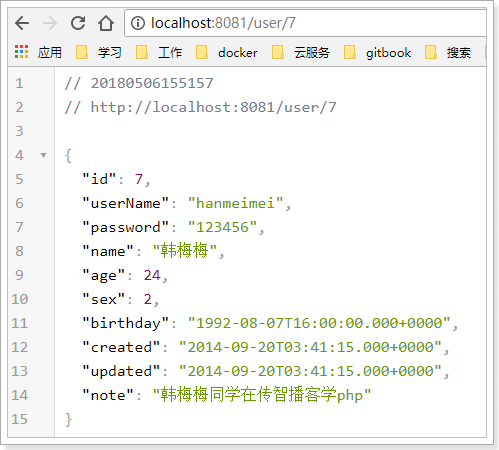
项目结构：



1525577911331

#### 5.1.3.启动并测试：

启动项目，访问接口：http://localhost:8081/user/7



1525593139364

### 5.2.服务调用者

#### 5.2.1.创建工程

与上面类似，这里不再赘述，需要注意的是，我们调用user-service的功能，因此不需要mybatis相关依赖了。

pom：

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.leyou.demo</groupId>  
 <artifactId>user-consumer-demo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <name>user-consumer-demo</name>  
 <description>Demo project for Spring Boot</description>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.1.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!-- 添加OkHttp支持 -->  
 <dependency>  
 <groupId>com.squareup.okhttp3</groupId>  
 <artifactId>okhttp</artifactId>  
 <version>3.9.0</version>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

#### 5.2.2.编写代码

首先在启动类中注册RestTemplate：

@SpringBootApplication  
public class UserConsumerDemoApplication {  
  
 @Bean  
 public RestTemplate restTemplate() {  
 // 这次我们使用了OkHttp客户端,只需要注入工厂即可  
 return new RestTemplate(new OkHttp3ClientHttpRequestFactory());  
 }  
  
 public static void main(String[] args) {  
 SpringApplication.run(UserConsumerDemoApplication.class, args);  
 }  
}

然后编写UserDao，注意，这里不是调用mapper查数据库，而是通过RestTemplate远程查询user-service-demo中的接口：

@Component  
public class UserDao {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 public User queryUserById(Long id){  
 String url = "http://localhost:8081/user/" + id;  
 return this.restTemplate.getForObject(url, User.class);  
 }  
}

然后编写user-service，循环查询UserDAO信息：

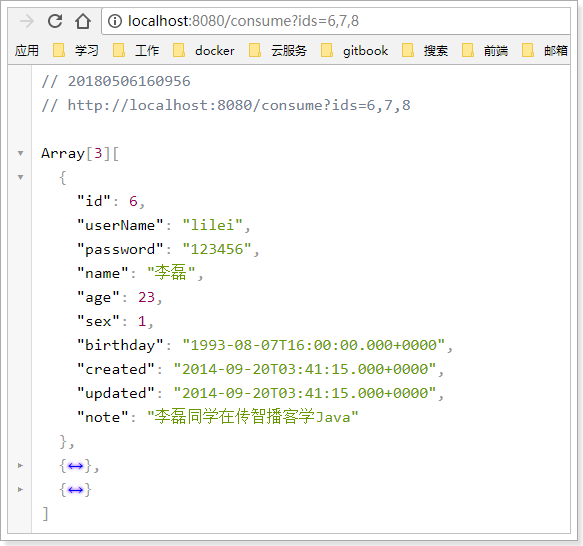
@Service  
public class UserService {  
  
 @Autowired  
 private UserDao userDao;  
  
 public List<User> querUserByIds(List<Long> ids){  
 List<User> users = new ArrayList<>();  
 for (Long id : ids) {  
 User user = this.userDao.queryUserById(id);  
 users.add(user);  
 }  
 return users;  
 }  
}

编写controller：

@RestController  
@RequestMapping("consume")  
public class ConsumerController {  
  
 @Autowired  
 private UserService userService;  
  
 @GetMapping  
 public List<User> consume(@RequestParam("ids") List<Long> ids) {  
 return this.userService.queryUserByIds(ids);  
 }  
}

#### 5.2.3.启动测试：

因为我们没有配置端口，那么默认就是8080，我们访问：http://localhost:8080/consume?ids=6,7,8



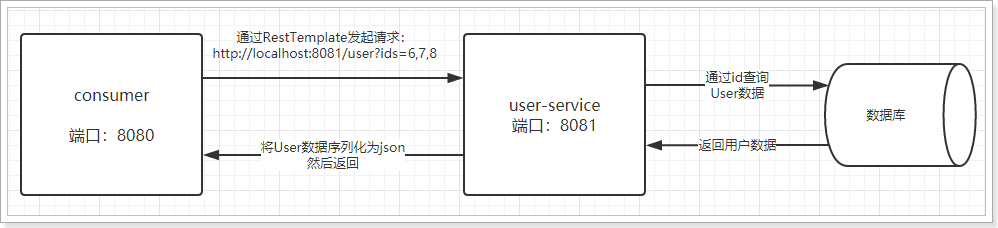
一个简单的远程服务调用案例就实现了。

### 有没有问题？

简单回顾一下，刚才我们写了什么：

* use-service-demo：一个提供根据id查询用户的微服务
* consumer-demo：一个服务调用者，通过RestTemplate远程调用user-service-demo

流程如下：



存在什么问题？

* 在consumer中，我们把url地址硬编码到了代码中，不方便后期维护
* consumer需要记忆user-service的地址，如果出现变更，可能得不到通知，地址将失效
* consumer不清楚user-service的状态，服务宕机也不知道
* user-service只有1台服务，不具备高可用性
* 即便user-service形成集群，consumer还需自己实现负载均衡

其实上面说的问题，概括一下就是分布式服务必然要面临的问题：

* 服务管理
* 如何自动注册和发现
* 如何实现状态监管
* 如何实现动态路由
* 服务如何实现负载均衡
* 服务如何解决容灾问题
* 服务如何实现统一配置

以上的问题，我们都将在SpringCloud中得到答案。

## Eureka注册中心

### 6.1.认识Eureka

首先我们来解决第一问题，服务的管理。

问题分析

在刚才的案例中，user-service对外提供服务，需要对外暴露自己的地址。而consumer（调用者）需要记录服务提供者的地址。将来地址出现变更，还需要及时更新。这在服务较少的时候并不觉得有什么，但是在现在日益复杂的互联网环境，一个项目肯定会拆分出十几，甚至数十个微服务。此时如果还人为管理地址，不仅开发困难，将来测试、发布上线都会非常麻烦，这与DevOps的思想是背道而驰的。

网约车

这就好比是 网约车出现以前，人们出门叫车只能叫出租车。一些私家车想做出租却没有资格，被称为黑车。而很多人想要约车，但是无奈出租车太少，不方便。私家车很多却不敢拦，而且满大街的车，谁知道哪个才是愿意载人的。一个想要，一个愿意给，就是缺少引子，缺乏管理啊。

此时滴滴这样的网约车平台出现了，所有想载客的私家车全部到滴滴注册，记录你的车型（服务类型），身份信息（联系方式）。这样提供服务的私家车，在滴滴那里都能找到，一目了然。

此时要叫车的人，只需要打开APP，输入你的目的地，选择车型（服务类型），滴滴自动安排一个符合需求的车到你面前，为你服务，完美！

Eureka做什么？

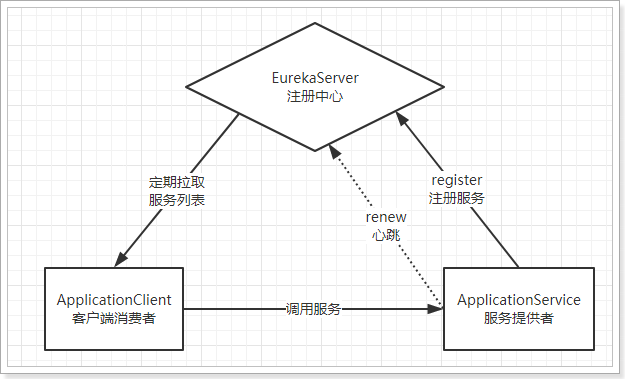
Eureka就好比是滴滴，负责管理、记录服务提供者的信息。服务调用者无需自己寻找服务，而是把自己的需求告诉Eureka，然后Eureka会把符合你需求的服务告诉你。

同时，服务提供方与Eureka之间通过“心跳”机制进行监控，当某个服务提供方出现问题，Eureka自然会把它从服务列表中剔除。

这就实现了服务的自动注册、发现、状态监控。

### 6.2.原理图

基本架构：



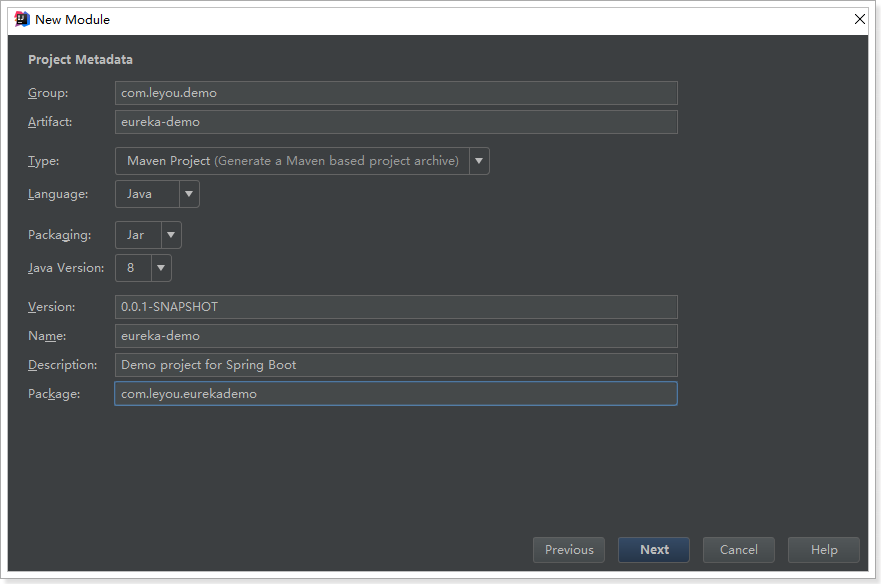
* Eureka：就是服务注册中心（可以是一个集群），对外暴露自己的地址
* 提供者：启动后向Eureka注册自己信息（地址，提供什么服务）
* 消费者：向Eureka订阅服务，Eureka会将对应服务的所有提供者地址列表发送给消费者，并且定期更新
* 心跳(续约)：提供者定期通过http方式向Eureka刷新自己的状态

### 6.3.入门案例

#### 6.3.1.编写EurekaServer

接下来我们创建一个项目，启动一个EurekaServer：

依然使用spring提供的快速搭建工具：



选择依赖：



完整的Pom文件：

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.leyou.demo</groupId>  
 <artifactId>eureka-demo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <packaging>jar</packaging>  
  
 <name>eureka-demo</name>  
 <description>Demo project for Spring Boot</description>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.0.1.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
  
 <properties>  
 <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  
 <java.version>1.8</java.version>  
 <!-- SpringCloud版本，是最新的F系列 -->  
 <spring-cloud.version>Finchley.RC1</spring-cloud.version>  
 </properties>  
  
 <dependencies>  
 <!-- Eureka服务端 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>  
 </dependencies>  
  
 <dependencyManagement>  
 <dependencies>  
 <!-- SpringCloud依赖，一定要放到dependencyManagement中，起到管理版本的作用即可 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
 <repositories>  
 <repository>  
 <id>spring-milestones</id>  
 <name>Spring Milestones</name>  
 <url>https://repo.spring.io/milestone</url>  
 <snapshots>  
 <enabled>false</enabled>  
 </snapshots>  
 </repository>  
 </repositories>  
</project>

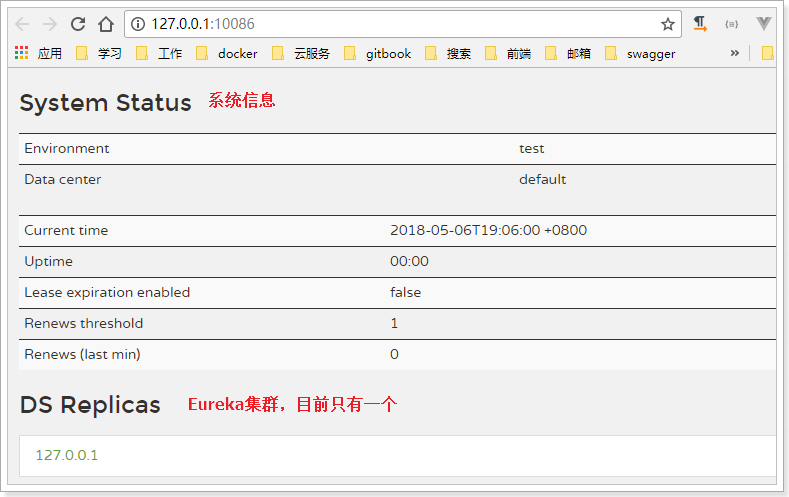
编写启动类：

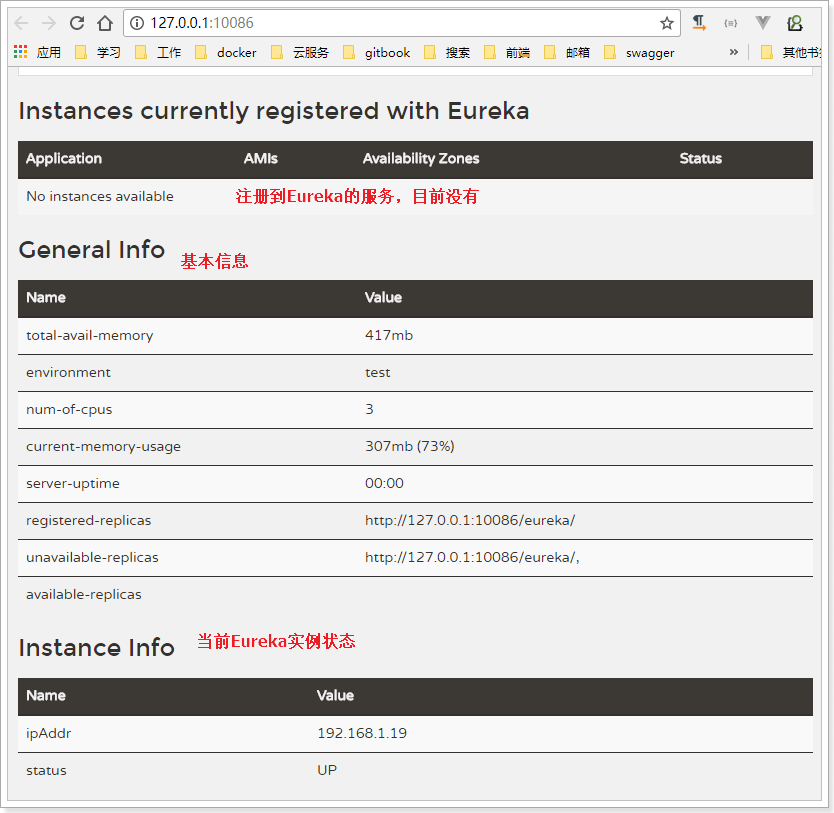
@SpringBootApplication  
@EnableEurekaServer // 声明这个应用是一个EurekaServer  
public class EurekaDemoApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.run(EurekaDemoApplication.class, args);  
 }  
}

编写配置：

server:  
 port: 10086 # 端口  
spring:  
 application:  
 name: eureka-server # 应用名称，会在Eureka中显示  
eureka:  
 client:  
 register-with-eureka: false # 是否注册自己的信息到EurekaServer，默认是true  
 fetch-registry: false # 是否拉取其它服务的信息，默认是true  
 service-url: # EurekaServer的地址，现在是自己的地址，如果是集群，需要加上其它Server的地址。  
 defaultZone: http://127.0.0.1:${server.port}/eureka

启动服务，并访问：http://127.0.0.1:10086/eureka





#### 将user-service注册到Eureka

注册服务，就是在服务上添加Eureka的客户端依赖，客户端代码会自动把服务注册到EurekaServer中。

我们在user-service-demo中添加Eureka客户端依赖：

先添加SpringCloud依赖：

<!-- SpringCloud的依赖 -->  
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Finchley.RC1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>  
<!-- Spring的仓库地址 -->  
<repositories>  
 <repository>  
 <id>spring-milestones</id>  
 <name>Spring Milestones</name>  
 <url>https://repo.spring.io/milestone</url>  
 <snapshots>  
 <enabled>false</enabled>  
 </snapshots>  
 </repository>  
</repositories>

然后是Eureka客户端：

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

在启动类上开启Eureka客户端功能

通过添加@EnableDiscoveryClient来开启Eureka客户端功能

@SpringBootApplication  
@EnableDiscoveryClient // 开启EurekaClient功能  
public class UserServiceDemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(UserServiceDemoApplication.class, args);  
 }  
}

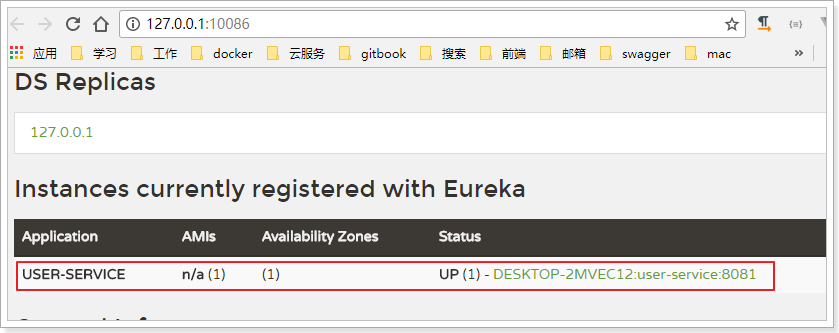
编写配置

server:  
 port: 8081  
spring:  
 datasource:  
 url: jdbc:mysql://localhost:3306/mydb01  
 username: root  
 password: 123  
 hikari:  
 maximum-pool-size: 20  
 minimum-idle: 10  
 application:  
 name: user-service # 应用名称  
mybatis:  
 type-aliases-package: com.leyou.userservice.pojo  
eureka:  
 client:  
 service-url: # EurekaServer地址  
 defaultZone: http://127.0.0.1:10086/eureka  
 instance:  
 prefer-ip-address: true # 当调用getHostname获取实例的hostname时，返回ip而不是host名称  
 ip-address: 127.0.0.1 # 指定自己的ip信息，不指定的话会自己寻找

注意：

* 这里我们添加了spring.application.name属性来指定应用名称，将来会作为应用的id使用。
* 不用指定register-with-eureka和fetch-registry，因为默认是true

重启项目，访问[Eureka监控页面](http://127.0.0.1:10086/eureka)查看



1525609225152

我们发现user-service服务已经注册成功了

#### 6.3.3.消费者从Eureka获取服务

接下来我们修改consumer-demo，尝试从EurekaServer获取服务。

方法与消费者类似，只需要在项目中添加EurekaClient依赖，就可以通过服务名称来获取信息了！

1）添加依赖：

先添加SpringCloud依赖：

<!-- SpringCloud的依赖 -->  
<dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>Finchley.RC1</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 </dependencies>  
</dependencyManagement>  
<!-- Spring的仓库地址 -->  
<repositories>  
 <repository>  
 <id>spring-milestones</id>  
 <name>Spring Milestones</name>  
 <url>https://repo.spring.io/milestone</url>  
 <snapshots>  
 <enabled>false</enabled>  
 </snapshots>  
 </repository>  
</repositories>

然后是Eureka客户端：

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

2）在启动类开启Eureka客户端

@SpringBootApplication  
@EnableDiscoveryClient // 开启Eureka客户端  
public class UserConsumerDemoApplication {  
 @Bean  
 public RestTemplate restTemplate() {  
 return new RestTemplate(new OkHttp3ClientHttpRequestFactory());  
 }  
 public static void main(String[] args) {  
 SpringApplication.run(UserConsumerDemoApplication.class, args);  
 }  
}

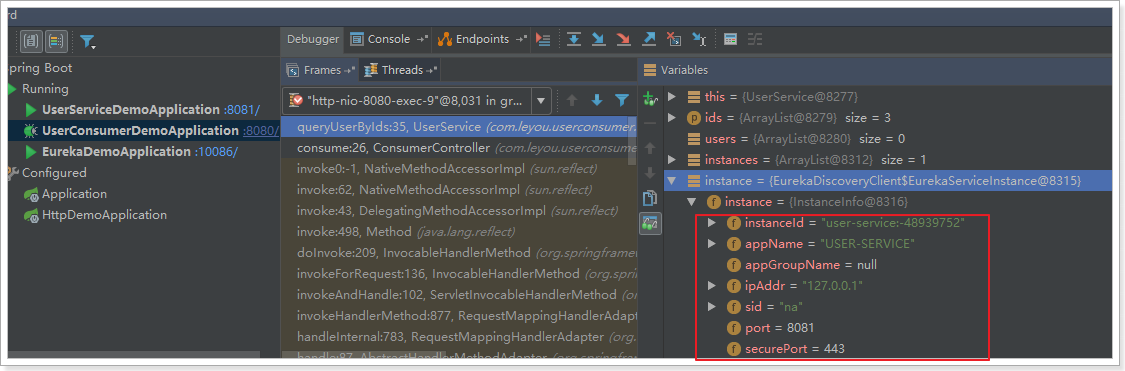
3）修改配置：

server:  
 port: 8080  
spring:  
 application:  
 name: consumer # 应用名称  
eureka:  
 client:  
 service-url: # EurekaServer地址  
 defaultZone: http://127.0.0.1:10086/eureka  
 instance:  
 prefer-ip-address: true # 当其它服务获取地址时提供ip而不是hostname  
 ip-address: 127.0.0.1 # 指定自己的ip信息，不指定的话会自己寻找

4）修改代码，用DiscoveryClient类的方法，根据服务名称，获取服务实例：

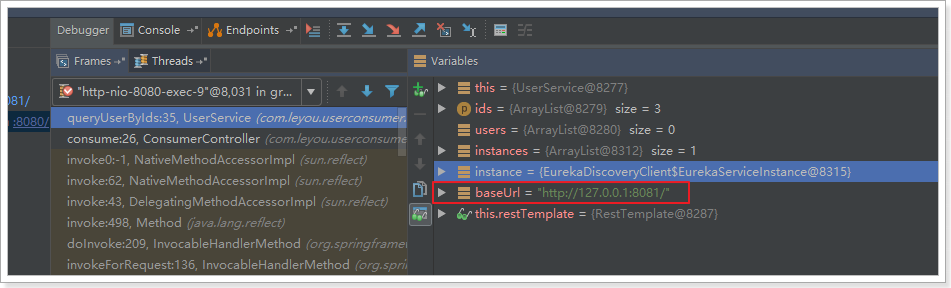
@Service  
public class UserService {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @Autowired  
 private DiscoveryClient discoveryClient;// Eureka客户端，可以获取到服务实例信息  
  
 public List<User> queryUserByIds(List<Long> ids) {  
 List<User> users = new ArrayList<>();  
 // String baseUrl = "http://localhost:8081/user/";  
 // 根据服务名称，获取服务实例  
 List<ServiceInstance> instances = discoveryClient.getInstances("user-service");  
 // 因为只有一个UserService,因此我们直接get(0)获取  
 ServiceInstance instance = instances.get(0);  
 // 获取ip和端口信息  
 String baseUrl = "http://"+instance.getHost() + ":" + instance.getPort()+"/user/";  
 ids.forEach(id -> {  
 // 我们测试多次查询，  
 users.add(this.restTemplate.getForObject(baseUrl + id, User.class));  
 // 每次间隔500毫秒  
 try {  
 Thread.sleep(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 return users;  
 }  
}

5）Debug跟踪运行：



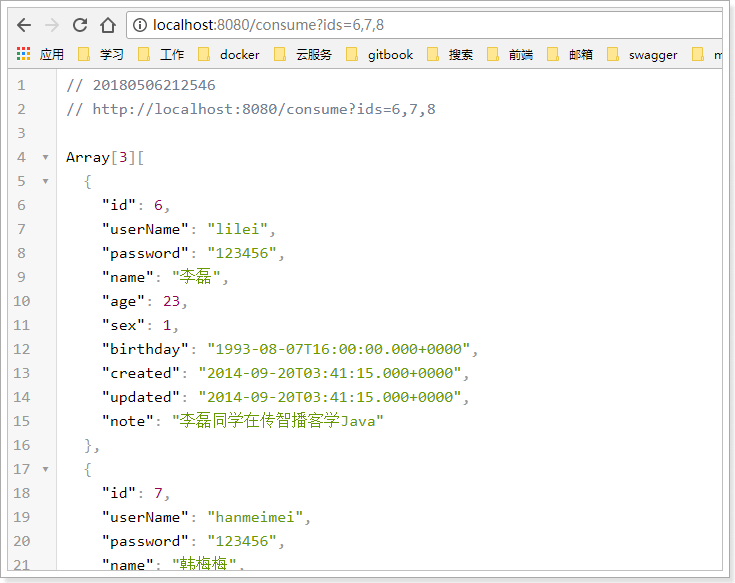
1525613025086

生成的URL：



1525613051210

访问结果：



1525613160920

### 6.4.Eureka详解

接下来我们详细讲解Eureka的原理及配置。

#### 6.4.1.基础架构

Eureka架构中的三个核心角色：

* 服务注册中心

Eureka的服务端应用，提供服务注册和发现功能，就是刚刚我们建立的eureka-demo

* 服务提供者

提供服务的应用，可以是SpringBoot应用，也可以是其它任意技术实现，只要对外提供的是Rest风格服务即可。本例中就是我们实现的user-service-demo

* 服务消费者

消费应用从注册中心获取服务列表，从而得知每个服务方的信息，知道去哪里调用服务方。本例中就是我们实现的consumer-demo

#### 6.4.2.高可用的Eureka Server

Eureka Server即服务的注册中心，在刚才的案例中，我们只有一个EurekaServer，事实上EurekaServer也可以是一个集群，形成高可用的Eureka中心。

服务同步

多个Eureka Server之间也会互相注册为服务，当服务提供者注册到Eureka Server集群中的某个节点时，该节点会把服务的信息同步给集群中的每个节点，从而实现**数据同步**。因此，无论客户端访问到Eureka Server集群中的任意一个节点，都可以获取到完整的服务列表信息。

动手搭建高可用的EurekaServer

我们假设要搭建两条EurekaServer的集群，端口分别为：10086和10087

1）我们修改原来的EurekaServer配置：

server:  
 port: 10086 # 端口  
spring:  
 application:  
 name: eureka-server # 应用名称，会在Eureka中显示  
eureka:  
 client:  
 service-url: # 配置其他Eureka服务的地址，而不是自己，比如10087  
 defaultZone: http://127.0.0.1:10087/eureka

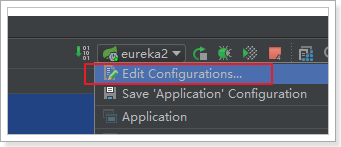
所谓的高可用注册中心，其实就是把EurekaServer自己也作为一个服务进行注册，这样多个EurekaServer之间就能互相发现对方，从而形成集群。因此我们做了以下修改：

* 删除了register-with-eureka=false和fetch-registry=false两个配置。因为默认值是true，这样就会吧自己注册到注册中心了。
* 把service-url的值改成了另外一台EurekaServer的地址，而不是自己

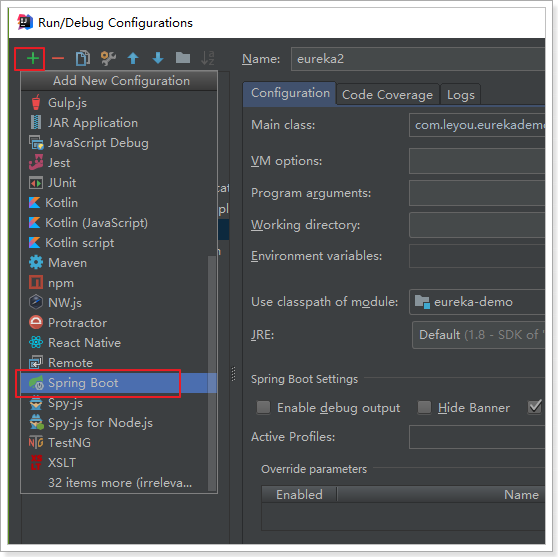
2）另外一台配置恰好相反：

server:  
 port: 10087 # 端口  
spring:  
 application:  
 name: eureka-server # 应用名称，会在Eureka中显示  
eureka:  
 client:  
 service-url: # 配置其他Eureka服务的地址，而不是自己，比如10087  
 defaultZone: http://127.0.0.1:10086/eureka

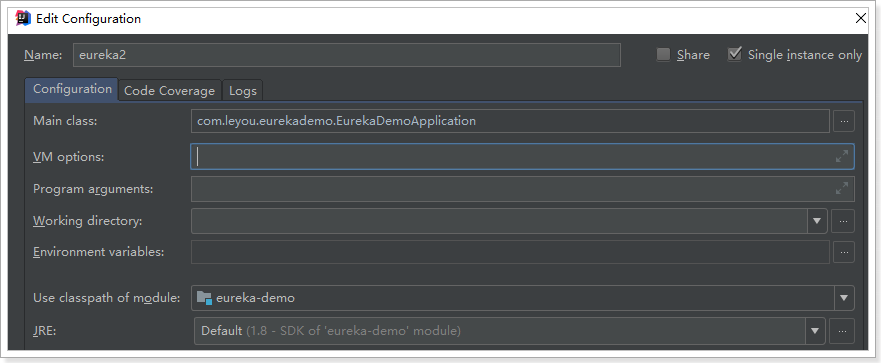
注意：idea中一个应用不能启动两次，我们需要重新配置一个启动器：



1525615070033



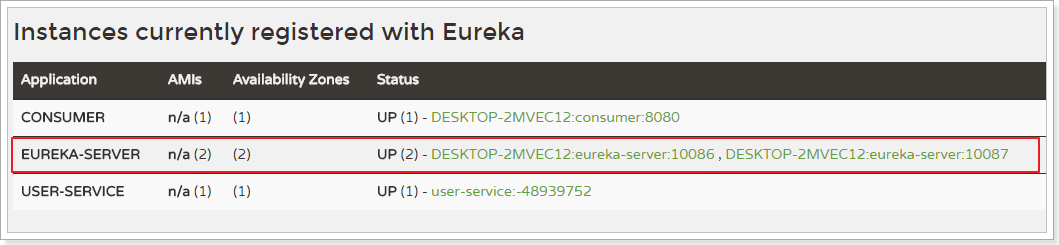
1525615095693



1525615026937

然后启动即可。

3）启动测试：



1525615165157

4）客户端注册服务到集群

因为EurekaServer不止一个，因此注册服务的时候，service-url参数需要变化：

eureka:  
 client:  
 service-url: # EurekaServer地址,多个地址以','隔开  
 defaultZone: http://127.0.0.1:10086/eureka,http://127.0.0.1:10087/eureka

#### 6.4.3.服务提供者

服务提供者要向EurekaServer注册服务，并且完成服务续约等工作。

服务注册

服务提供者在启动时，会检测配置属性中的：eureka.client.register-with-erueka=true参数是否正确，事实上默认就是true。如果值确实为true，则会向EurekaServer发起一个Rest请求，并携带自己的元数据信息，Eureka Server会把这些信息保存到一个双层Map结构中。第一层Map的Key就是服务名称，第二层Map的key是服务的实例id。

服务续约

在注册服务完成以后，服务提供者会维持一个心跳（定时向EurekaServer发起Rest请求），告诉EurekaServer：“我还活着”。这个我们称为服务的续约（renew）；

有两个重要参数可以修改服务续约的行为：

eureka:  
 instance:  
 lease-expiration-duration-in-seconds: 90  
 lease-renewal-interval-in-seconds: 30

* lease-renewal-interval-in-seconds：服务续约(renew)的间隔，默认为30秒
* lease-expiration-duration-in-seconds：服务失效时间，默认值90秒

也就是说，默认情况下每个30秒服务会向注册中心发送一次心跳，证明自己还活着。如果超过90秒没有发送心跳，EurekaServer就会认为该服务宕机，会从服务列表中移除，这两个值在生产环境不要修改，默认即可。

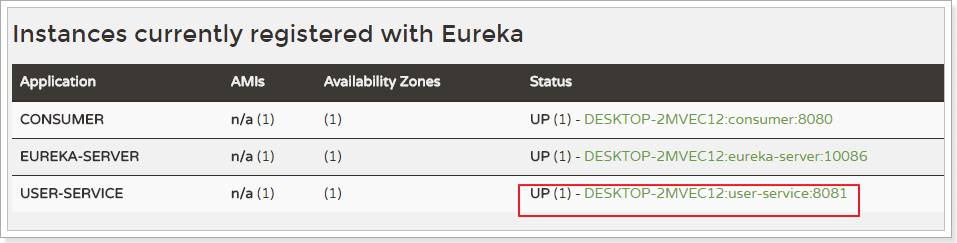
但是在开发时，这个值有点太长了，经常我们关掉一个服务，会发现Eureka依然认为服务在活着。所以我们在开发阶段可以适当调小。

eureka:  
 instance:  
 lease-expiration-duration-in-seconds: 10 # 10秒即过期  
 lease-renewal-interval-in-seconds: 5 # 5秒一次心跳

实例id

先来看一下服务状态信息：

在Eureka监控页面，查看服务注册信息：



1525617060656

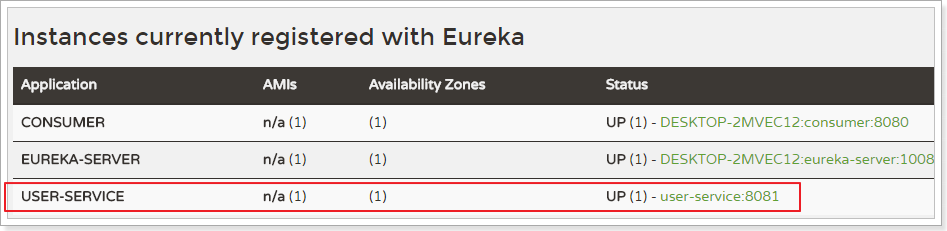
在status一列中，显示以下信息：

* UP(1)：代表现在是启动了1个示例，没有集群
* DESKTOP-2MVEC12:user-service:8081：是示例的名称（instance-id），
* 默认格式是：${hostname} + ${spring.application.name} + ${server.port}
* instance-id是区分同一服务的不同实例的唯一标准，因此不能重复。

我们可以通过instance-id属性来修改它的构成：

eureka:  
 instance:  
 instance-id: ${spring.application.name}:${server.port}

重启服务再试试看：



#### 服务消费者

获取服务列表

当服务消费者启动是，会检测eureka.client.fetch-registry=true参数的值，如果为true，则会从Eureka Server服务的列表只读备份，然后缓存在本地。并且每隔30秒会重新获取并更新数据。我们可以通过下面的参数来修改：

eureka:  
 client:  
 registry-fetch-interval-seconds: 5

生产环境中，我们不需要修改这个值。

但是为了开发环境下，能够快速得到服务的最新状态，我们可以将其设置小一点。

#### 失效剔除和自我保护

失效剔除

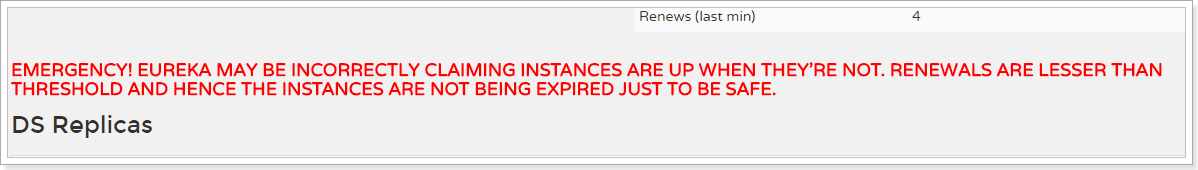
有些时候，我们的服务提供方并不一定会正常下线，可能因为内存溢出、网络故障等原因导致服务无法正常工作。Eureka Server需要将这样的服务剔除出服务列表。因此它会开启一个定时任务，每隔60秒对所有失效的服务（超过90秒未响应）进行剔除。

可以通过eureka.server.eviction-interval-timer-in-ms参数对其进行修改，单位是毫秒，生成环境不要修改。

这个会对我们开发带来极大的不变，你对服务重启，隔了60秒Eureka才反应过来。开发阶段可以适当调整，比如10S

自我保护

我们关停一个服务，就会在Eureka面板看到一条警告：



这是触发了Eureka的自我保护机制。当一个服务未按时进行心跳续约时，Eureka会统计最近15分钟心跳失败的服务实例的比例是否超过了85%。在生产环境下，因为网络延迟等原因，心跳失败实例的比例很有可能超标，但是此时就把服务剔除列表并不妥当，因为服务可能没有宕机。Eureka就会把当前实例的注册信息保护起来，不予剔除。生产环境下这很有效，保证了大多数服务依然可用。

但是这给我们的开发带来了麻烦， 因此开发阶段我们都会关闭自我保护模式：

eureka:  
 server:  
 enable-self-preservation: false # 关闭自我保护模式（缺省为打开）  
 eviction-interval-timer-in-ms: 1000 # 扫描失效服务的间隔时间（缺省为60\*1000ms）

## 负载均衡Robbin

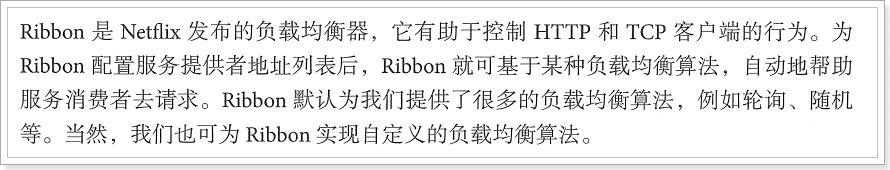
在刚才的案例中，我们启动了一个user-service，然后通过DiscoveryClient来获取服务实例信息，然后获取ip和端口来访问。

但是实际环境中，我们往往会开启很多个user-service的集群。此时我们获取的服务列表中就会有多个，到底该访问哪一个呢？

一般这种情况下我们就需要编写负载均衡算法，在多个实例列表中进行选择。

不过Eureka中已经帮我们集成了负载均衡组件：Ribbon，简单修改代码即可使用。

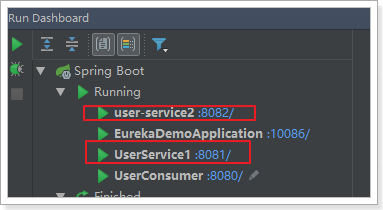
什么是Ribbon：



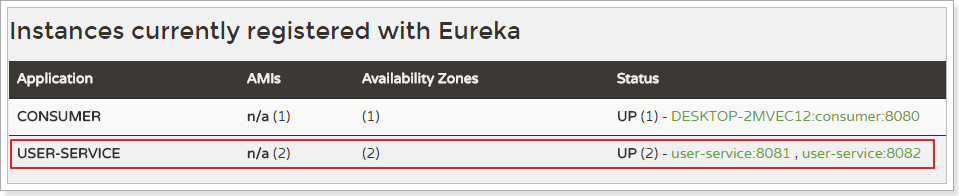
接下来，我们就来使用Ribbon实现负载均衡。

### 启动两个服务实例

首先我们启动两个user-service实例，一个8081，一个8082。



Eureka监控面板：



### 开启负载均衡

因为Eureka中已经集成了Ribbon，所以我们无需引入新的依赖。直接修改代码：

在RestTemplate的配置方法上添加@LoadBalanced注解：

@Bean  
@LoadBalanced  
public RestTemplate restTemplate() {  
 return new RestTemplate(new OkHttp3ClientHttpRequestFactory());  
}

修改调用方式，不再手动获取ip和端口，而是直接通过服务名称调用：

@Service  
public class UserService {  
  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @Autowired  
 private DiscoveryClient discoveryClient;  
  
 public List<User> queryUserByIds(List<Long> ids) {  
 List<User> users = new ArrayList<>();  
 // 地址直接写服务名称即可  
 String baseUrl = "http://user-service/user/";  
 ids.forEach(id -> {  
 // 我们测试多次查询，  
 users.add(this.restTemplate.getForObject(baseUrl + id, User.class));  
 // 每次间隔500毫秒  
 try {  
 Thread.sleep(500);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 return users;  
 }  
}

访问页面，查看结果：



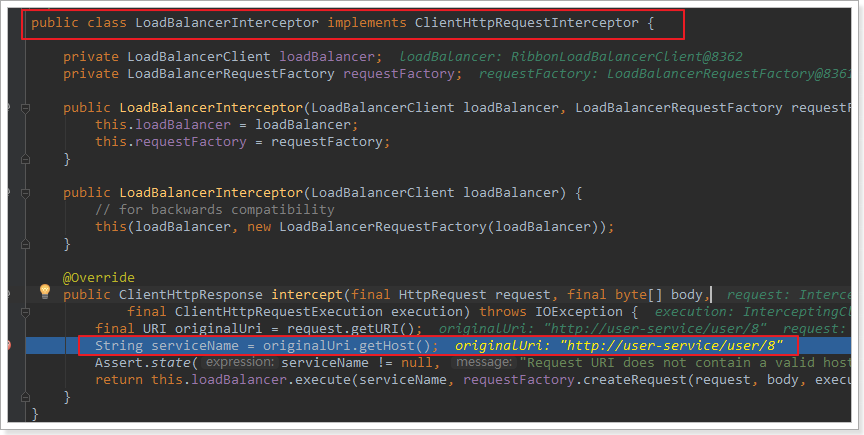
完美！

### 源码跟踪

为什么我们只输入了service名称就可以访问了呢？之前还要获取ip和端口。

显然有人帮我们根据service名称，获取到了服务实例的ip和端口。它就是LoadBalancerInterceptor

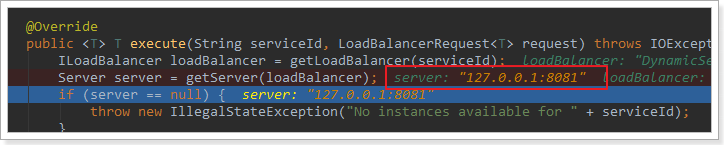
我们进行源码跟踪：



继续跟入execute方法：发现获取了8082端口的服务



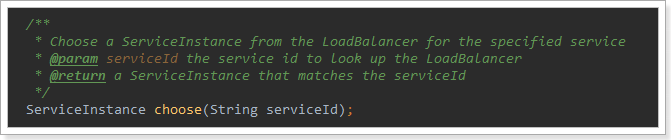
再跟下一次，发现获取的是8081：



### 负载均衡策略

Ribbon默认的负载均衡策略是简单的轮询，我们可以测试一下：

编写测试类，在刚才的源码中我们看到拦截中是使用RibbonLoadBalanceClient来进行负载均衡的，其中有一个choose方法，是这样介绍的：

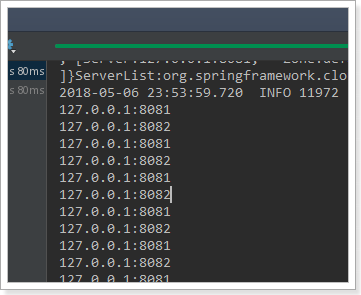


现在这个就是负载均衡获取实例的方法。

我们对注入这个类的对象，然后对其测试：

@RunWith(SpringRunner.class)  
@SpringBootTest(classes = UserConsumerDemoApplication.class)  
public class LoadBalanceTest {  
  
 @Autowired  
 RibbonLoadBalancerClient client;  
  
 @Test  
 public void test(){  
 for (int i = 0; i < 100; i++) {  
 ServiceInstance instance = this.client.choose("user-service");  
 System.out.println(instance.getHost() + ":" + instance.getPort());  
 }  
 }  
}

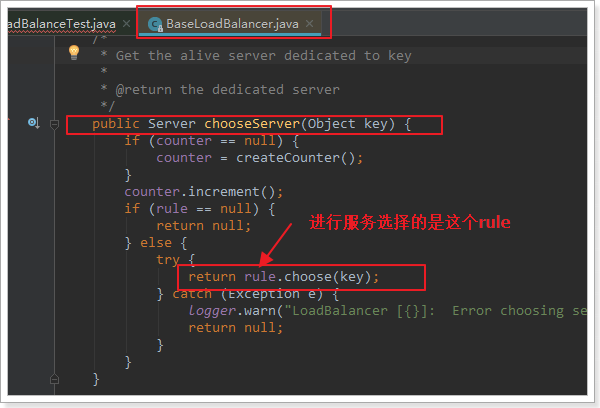
结果：



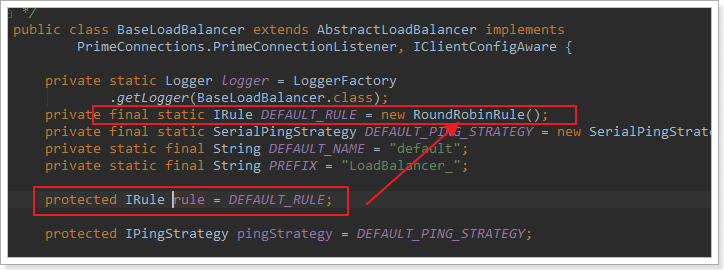
符合了我们的预期推测，确实是轮询方式。

我们是否可以修改负载均衡的策略呢？

继续跟踪源码，发现这么一段代码：

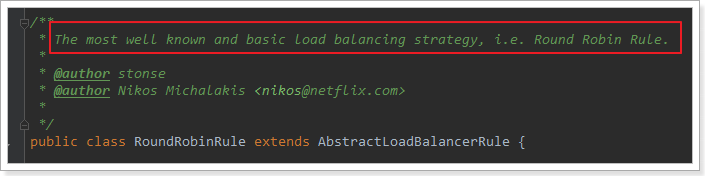


我们看看这个rule是谁：



1525622699666

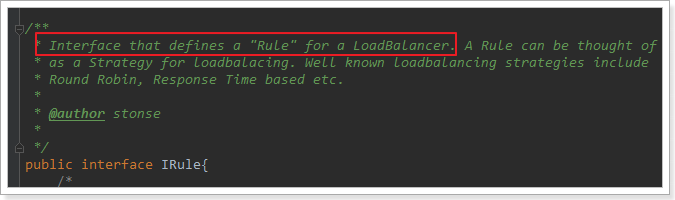
这里的rule默认值是一个RoundRobinRule，看类的介绍：



1525622754316

这不就是轮询的意思嘛。

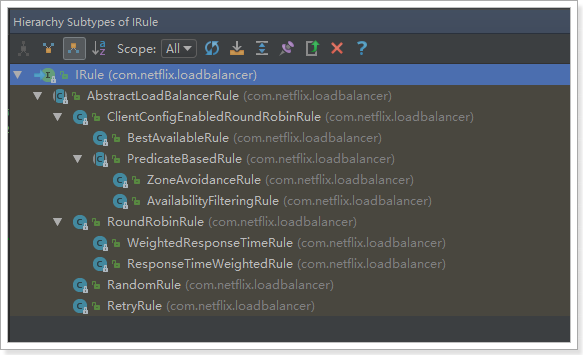
我们注意到，这个类其实是实现了接口IRule的，查看一下：



1525622817451

定义负载均衡的规则接口。

它有以下实现：



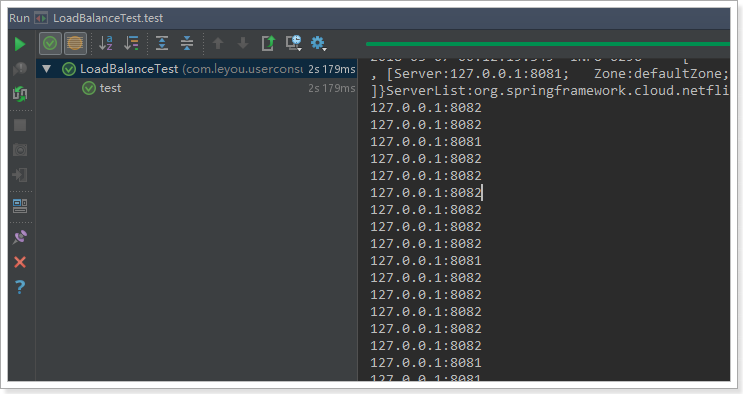
1525622876842

SpringBoot也帮我们提供了修改负载均衡规则的配置入口：

user-service:  
 ribbon:  
 NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

格式是：{服务名称}.ribbon.NFLoadBalancerRuleClassName，值就是IRule的实现类。

再次测试，发现结果变成了随机：



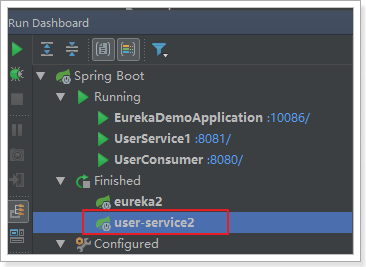
1525623193949

### 重试机制

Eureka的服务治理强调了CAP原则中的AP，即可用性和可靠性。它与Zookeeper这一类强调CP（一致性，可靠性）的服务治理框架最大的区别在于：Eureka为了实现更高的服务可用性，牺牲了一定的一致性，极端情况下它宁愿接收故障实例也不愿丢掉健康实例，正如我们上面所说的自我保护机制。

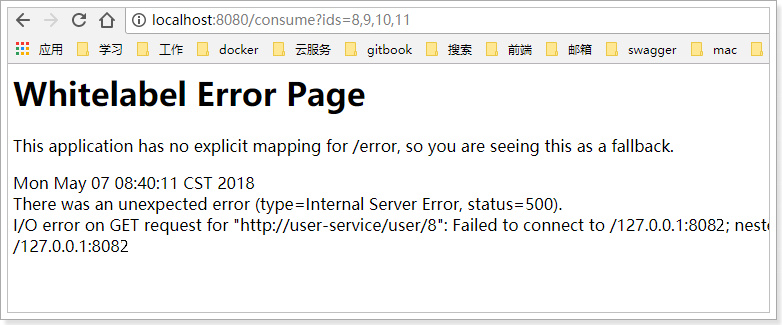
但是，此时如果我们调用了这些不正常的服务，调用就会失败，从而导致其它服务不能正常工作！这显然不是我们愿意看到的。

我们现在关闭一个user-service实例：



1525653565855

因为服务剔除的延迟，consumer并不会立即得到最新的服务列表，此时再次访问你会得到错误提示：



1525653715488

但是此时，8081服务其实是正常的。

因此Spring Cloud 整合了Spring Retry 来增强RestTemplate的重试能力，当一次服务调用失败后，不会立即抛出一次，而是再次重试另一个服务。

只需要简单配置即可实现Ribbon的重试：

spring:  
 cloud:  
 loadbalancer:  
 retry:  
 enabled: true # 开启Spring Cloud的重试功能  
user-service:  
 ribbon:  
 ConnectTimeout: 250 # Ribbon的连接超时时间  
 ReadTimeout: 1000 # Ribbon的数据读取超时时间  
 OkToRetryOnAllOperations: true # 是否对所有操作都进行重试  
 MaxAutoRetriesNextServer: 1 # 切换实例的重试次数  
 MaxAutoRetries: 1 # 对当前实例的重试次数

根据如上配置，当访问到某个服务超时后，它会再次尝试访问下一个服务实例，如果不行就再换一个实例，如果不行，则返回失败。切换次数取决于MaxAutoRetriesNextServer参数的值

引入spring-retry依赖

<dependency>  
 <groupId>org.springframework.retry</groupId>  
 <artifactId>spring-retry</artifactId>  
</dependency>

我们重启user-consumer-demo，测试，发现即使user-service2宕机，也能通过另一台服务实例获取到结果！

### Ribbon和Feign区别

Ribbon和Feign都是用于调用其他服务的，不过方式不同。

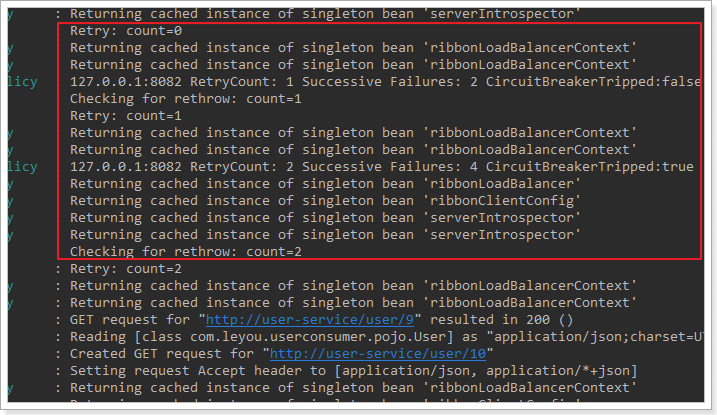
1.启动类使用的注解不同，Ribbon用的是@RibbonClient，Feign用的是@EnableFeignClients。

2.服务的指定位置不同，Ribbon是在@RibbonClient注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient声明。

3.调用方式不同，Ribbon需要自己构建http请求，[模拟](https://www.baidu.com/s?wd=%E6%A8%A1%E6%8B%9F&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)http请求然后使用RestTemplate发送给其他服务，步骤相当繁琐。

  Feign则是在Ribbon的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，

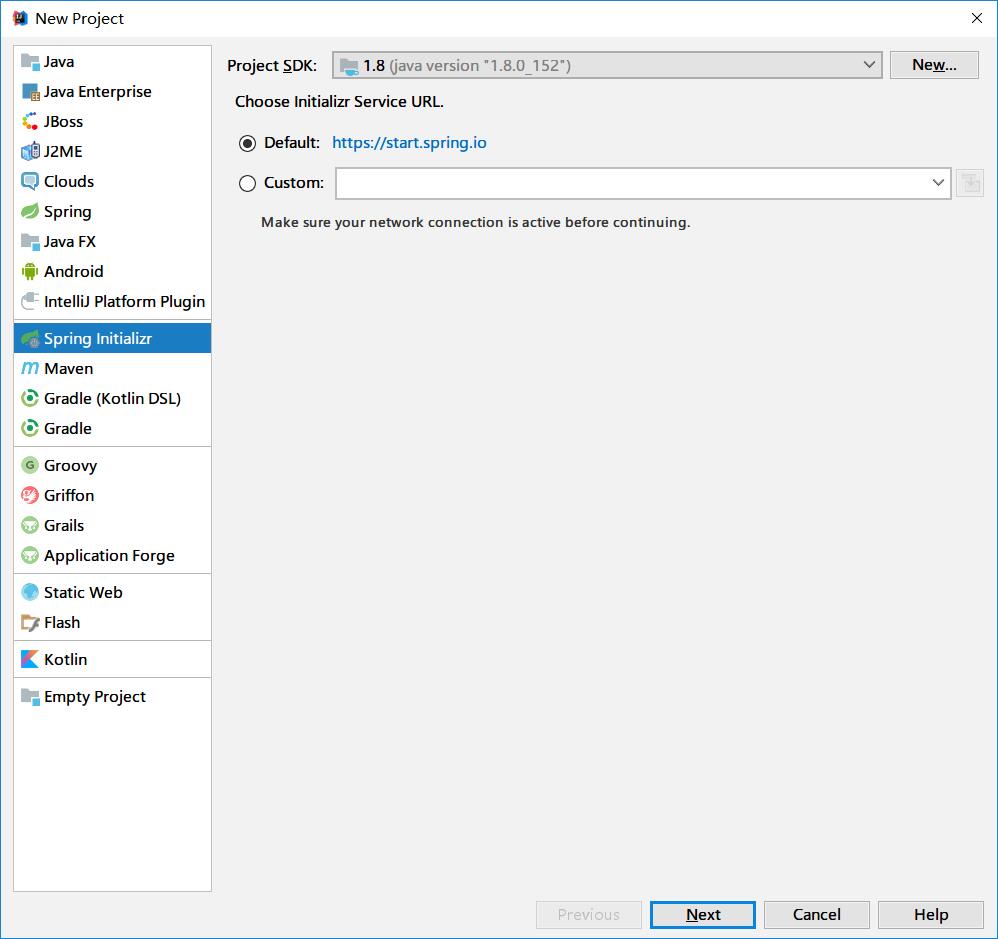
  不需要自己构建http请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。



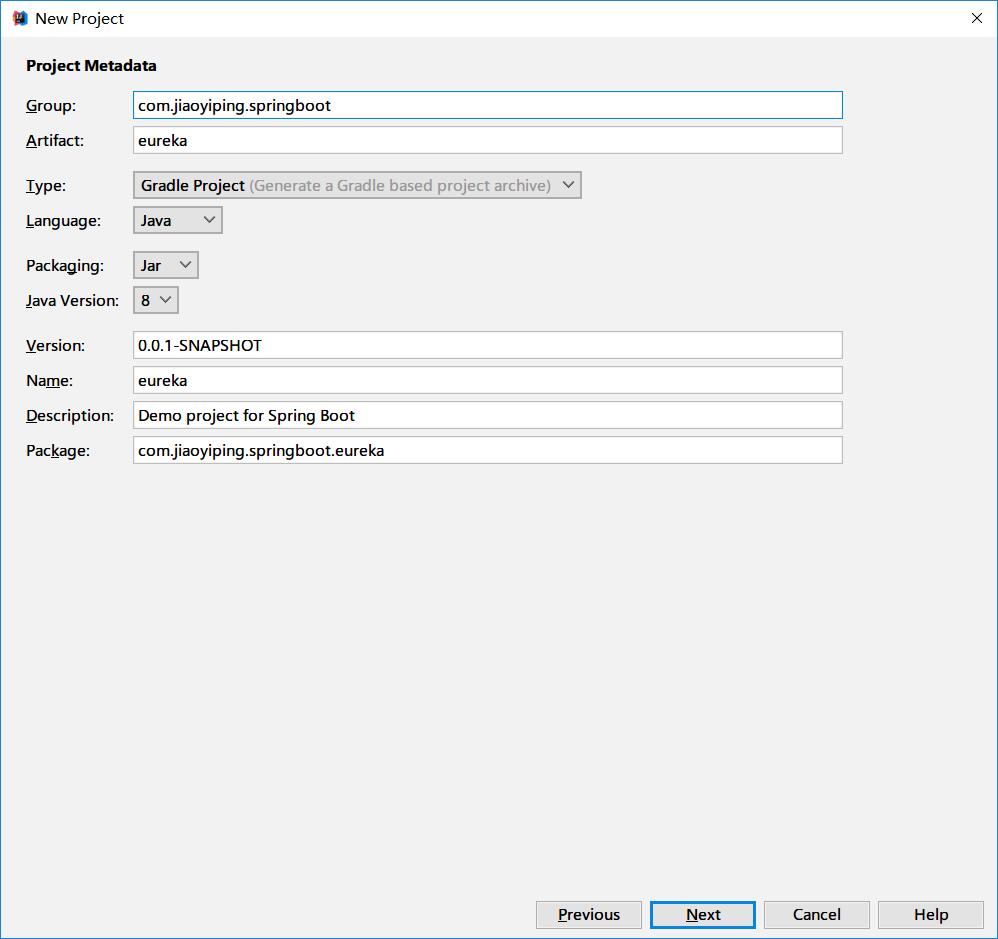
# **基于spring-cloud的微服务**

## [服务注册中心eureka](https://www.cnblogs.com/jiaoyiping/p/8689192.html)

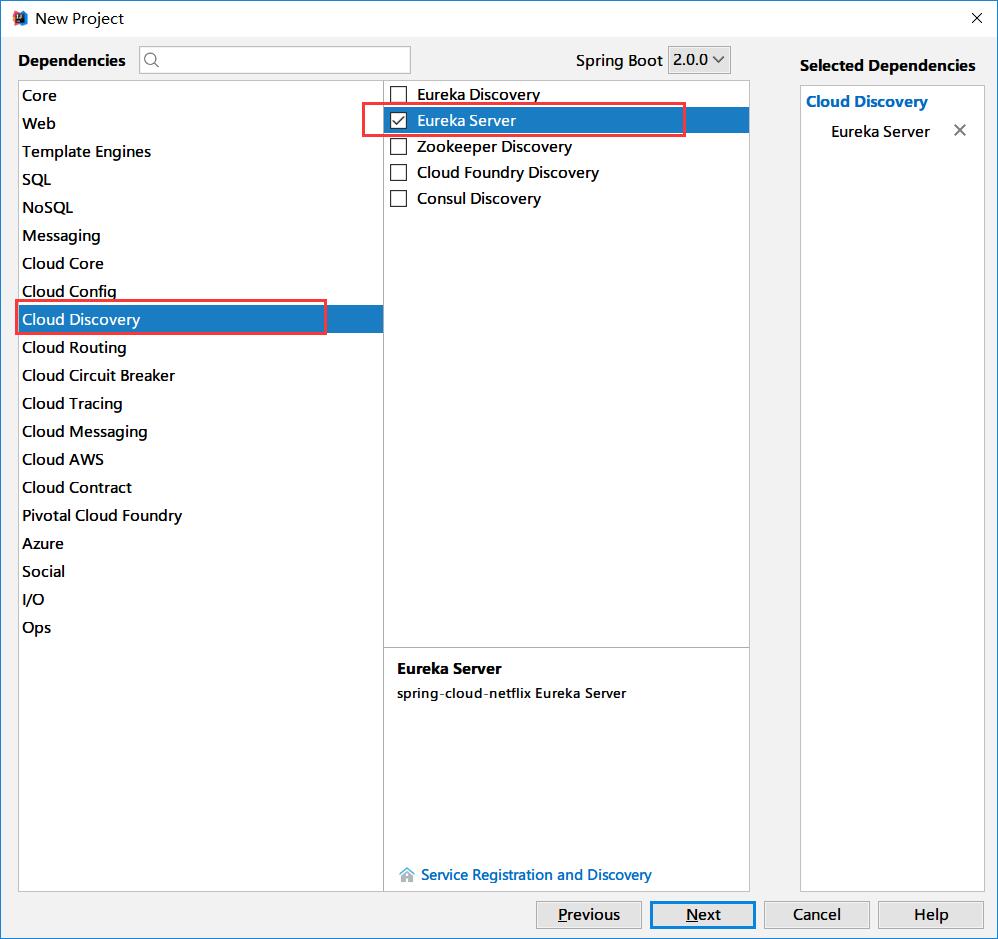
eureka是Netflix提供的服务注册中心组建,springcloud将其做了封装,作为自己的微服务架构中的一个注册中心组建  
下面的例子在IDEA中启动一个eureka的实例,然后提供一个provider,注册到这个注册中心去  
在IDEA中使用spring Initializer来新建一个Eureka项目



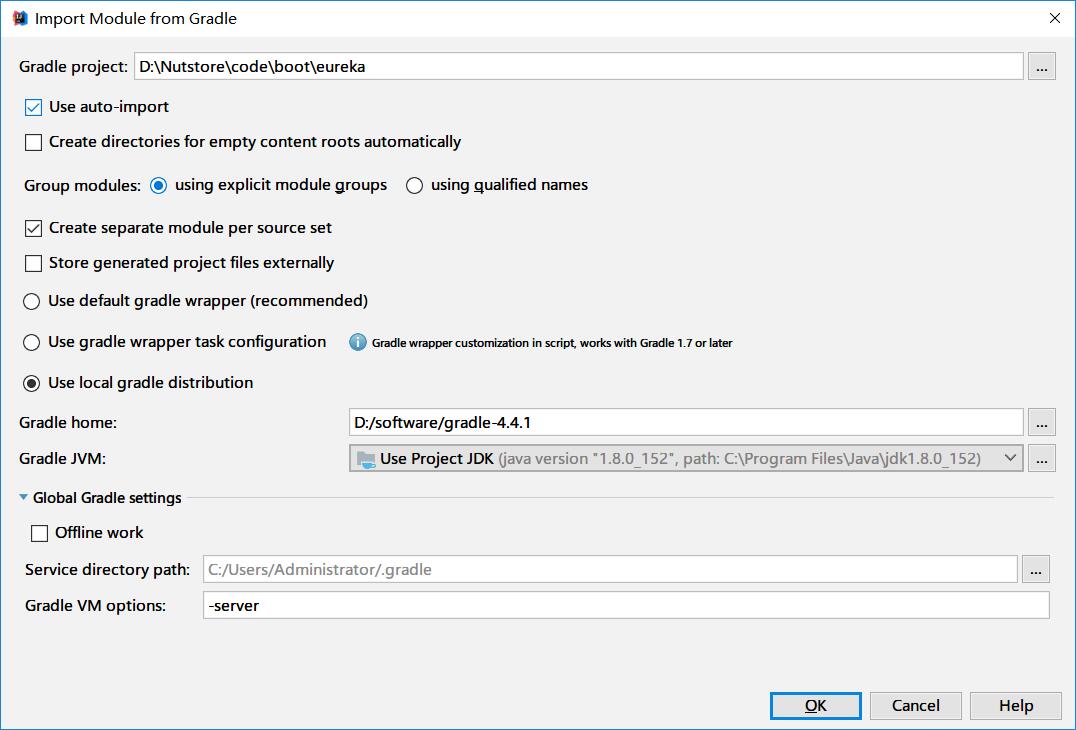
填写相关的信息



选择eureka server



选择项目地址和gradle等信息,然后创建项目



修改启动类EurekaApplication 增加@EnableEurekaServer注解,整个启动类的代码如下:

package com.jiaoyiping.springboot.eureka;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

public class EurekaApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaApplication.class, args);

}

}

修改spring-boot应用的配置文件,这里了使用的是yml文件的格式,整个配置文件内容如下:

spring:

application:

name: eureka-server

info:

app:

name: ${spring.application.name}

version: v1.0.0

server:

port: 8081

tomcat:

uri-encoding: UTF-8

servlet:

context-path: /

logging:

config: classpath:logback.xml

eureka:

instance:

hostname: 127.0.0.1

prefer-ip-address: true

client:

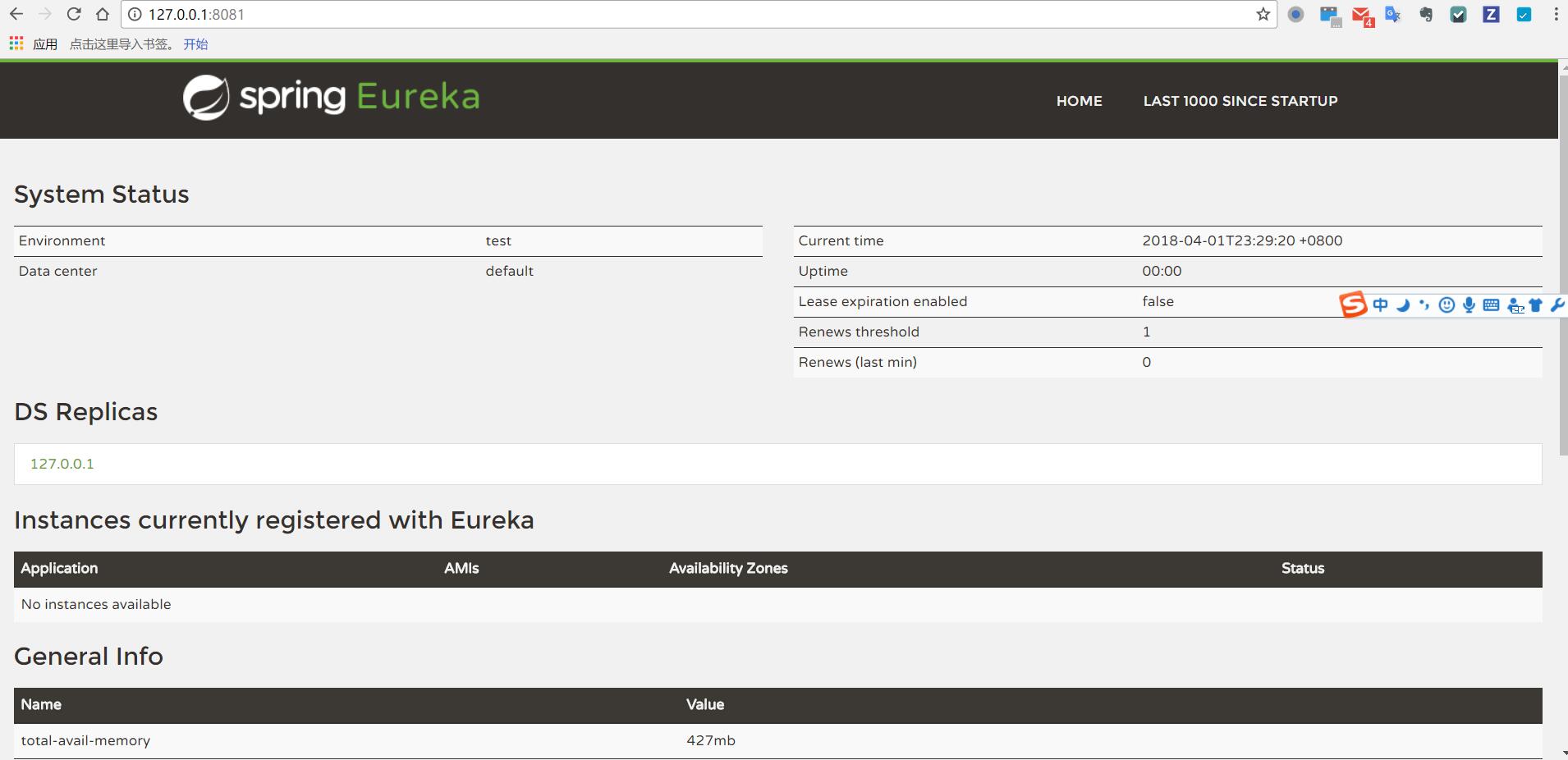
registerWithEureka: false

fetchRegistry: false

service-url:

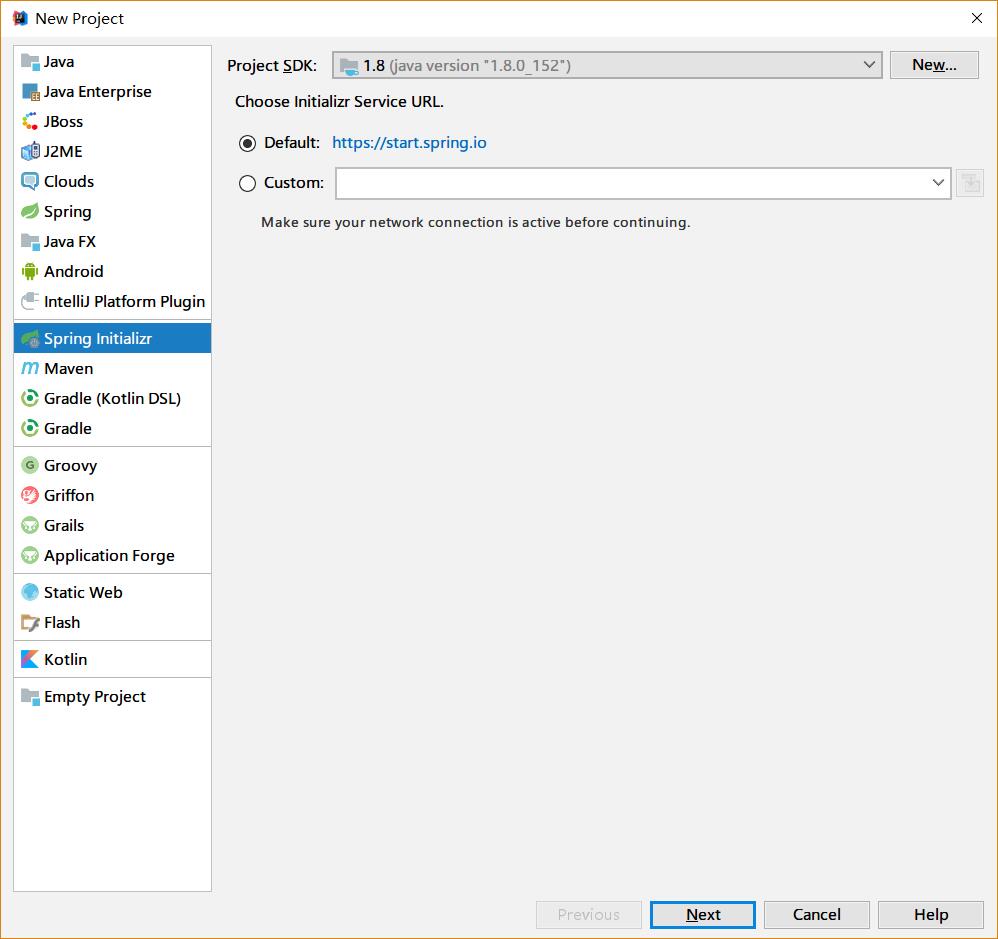
defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/

启动EurekaApplication 然后访问 http://localhost:8081 看到如下的页面,算是启动成功:

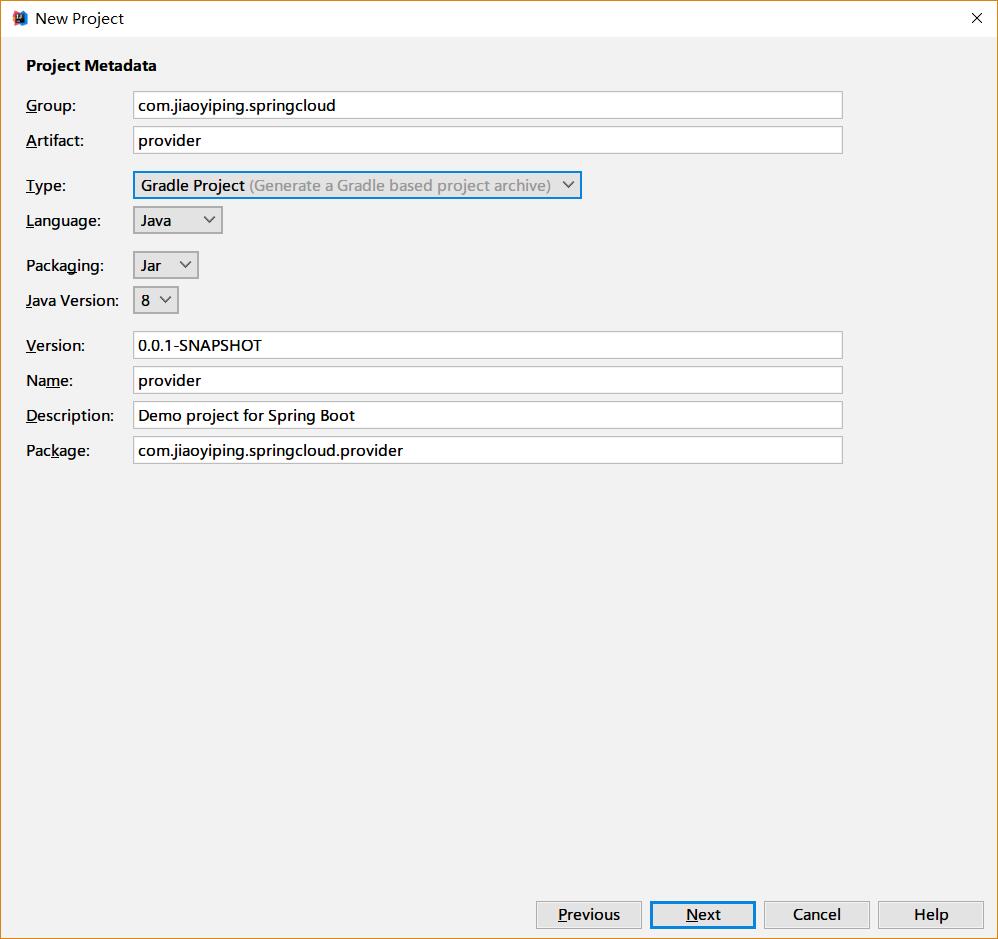


## [eureka服务提供方的注册和消费方的消费](https://www.cnblogs.com/jiaoyiping/p/8701110.html)

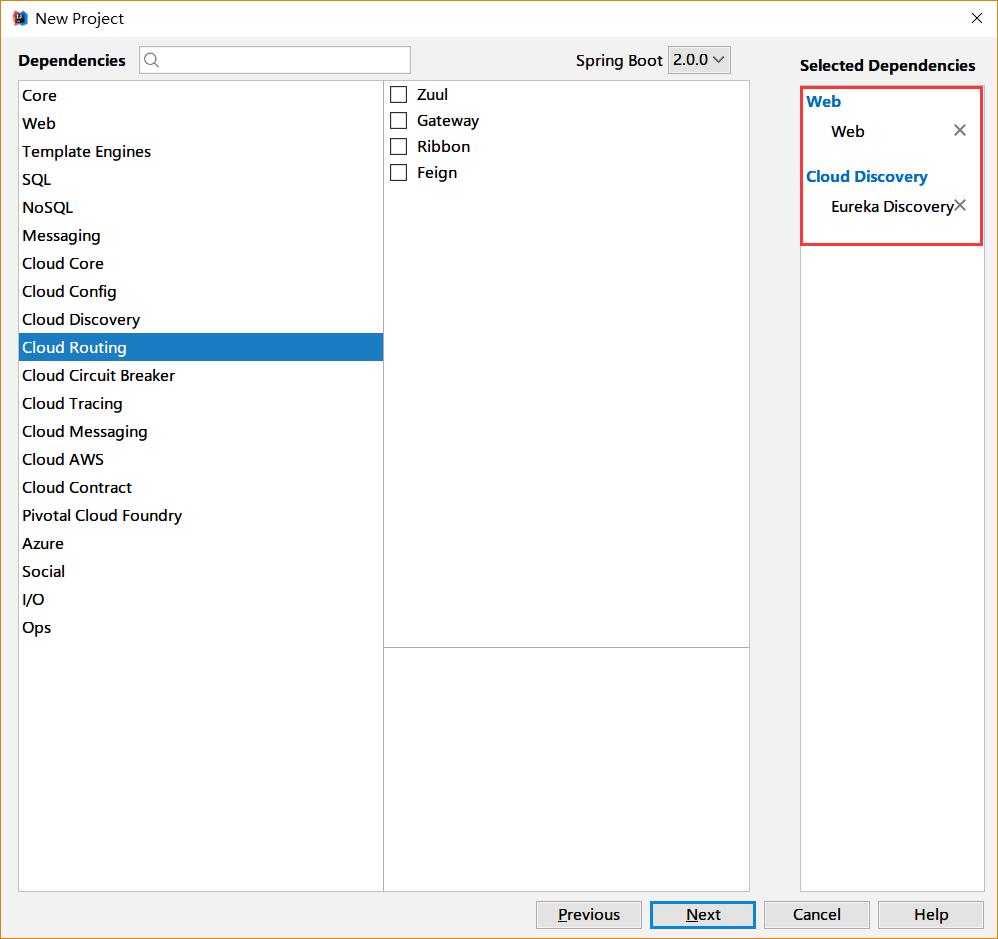
启动Eureka注册中心之后,服务提供方就可以注册到Eureka上去(作为一个Eureka的客户端)  
我们使用IDEA提供的spring initializer来新建一个springcloud项目



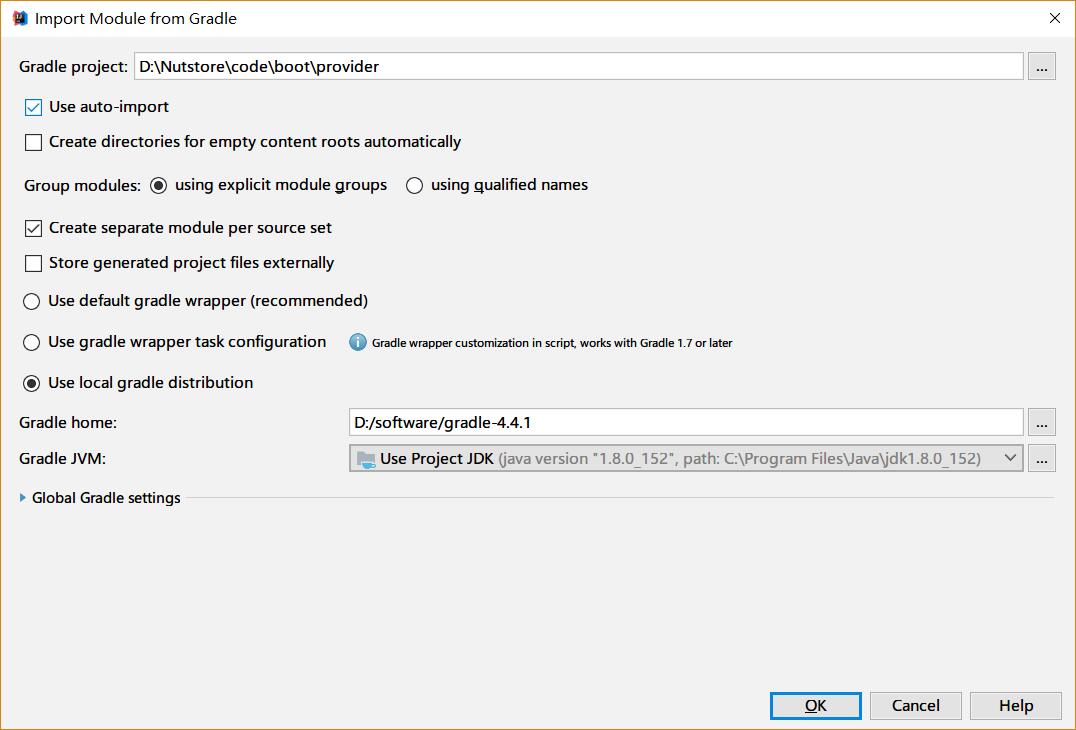
填写相关的包名等信息:



选择web和Erureka Discorvery



填写Gradle等相关信息完成创建



我们需要在启动类里边增加注解 @EnableDiscoveryClient 来标识这是一个Eureka客户端  
启动类的代码如下:

package com.jiaoyiping.springcloud.provider;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

public class ProviderApplication {

public static void main(String[] args) {

SpringApplication.run(ProviderApplication.class, args);

}

}

创建一个我们用来真正提供服务的Controller(在本例中,我们假设从路径中取到两个int值,将他们的和用字符形式返回,代码如下):

package com.jiaoyiping.springcloud.provider;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ProductionController {

//简单计算两个数字的和,转换为字符串返回

@RequestMapping(value = "/add/{a}/{b}")

public String add(@PathVariable("a") int a, @PathVariable("b") int b) {

return (a + b) + "";

}

}

修改应用的配置文件,指定服务的名称,网卡信息和eureka等相关信息:配置文件如下:

spring:

application:

name: provider

cloud:

inetutils:

preferred-networks: 192.168.1.

server:

port: 8082

eureka:

client:

service-url:

defaultZone: http://127.0.0.1:8081/eureka

instance:

prefer-ip-address: true

instance-id: 192.168.1.5:${server.port}

运行启动类,可以在eureka的页面上看到一个名为provider的服务已经注册上去了:



### 服务的消费方:

服务的消费方也是和服务提供方一样,也会到Eureka中去注册,按照上边的步骤,我们新建一个consumer项目,比前边的provider多加上Feign的依赖



在启动类上边,多加两个注解:

@EnableFeignClients

@EnableDiscoveryClient

启动类的代码如下:

package com.jiaoyiping.springcloud.consumer;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.cloud.openfeign.EnableFeignClients;

@SpringBootApplication

@EnableFeignClients

@EnableDiscoveryClient

public class ConsumerApplication {

public static void main(String[] args) {

SpringApplication.run(ConsumerApplication.class, args);

}

}

新建一个ProductControllerRemote接口,作为FeignClient,里边定义的方法和服务提供方的方法一致;

package com.jiaoyiping.springcloud.consumer;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

@FeignClient(name = "provider")

public interface ProductControllerRemote {

@RequestMapping(value = "/add/{a}/{b}")

String add(@PathVariable("a") int a, @PathVariable("b") int b);

}

@FeignClient注解的name,指的是注册到eureka上的应用的名称(在服务提供方的spring的配置文件里配置) 接口的方法映射到服务提供方的相应的方法上,只需要定义,不需要实现,Feign会自动调用服务提供方的相应的方法

使用@FeignClient注解的这个类,可以作为一个spring的bean,注入到需要的地方去,直接调用相应的方法即可

作为例子,我们在消费方新定义一个Controller,来调用ProductControllerRemote:

package com.jiaoyiping.springcloud.consumer;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

public class ConsumerController {

@Autowired

private ProductControllerRemote productControllerRemote;

@RequestMapping("/getresult/{a}/{b}")

public String getResult(@PathVariable("a") int a, @PathVariable("b") int b) {

return productControllerRemote.add(a, b);

}

}

然后是消费方的配置文件,需要配置Eureka的地址和服务的端口等信息:

spring:

application:

name: cunsumer

cloud:

inetutils:

preferred-networks: 192.168.1.

server:

port: 8083

eureka:

client:

service-url:

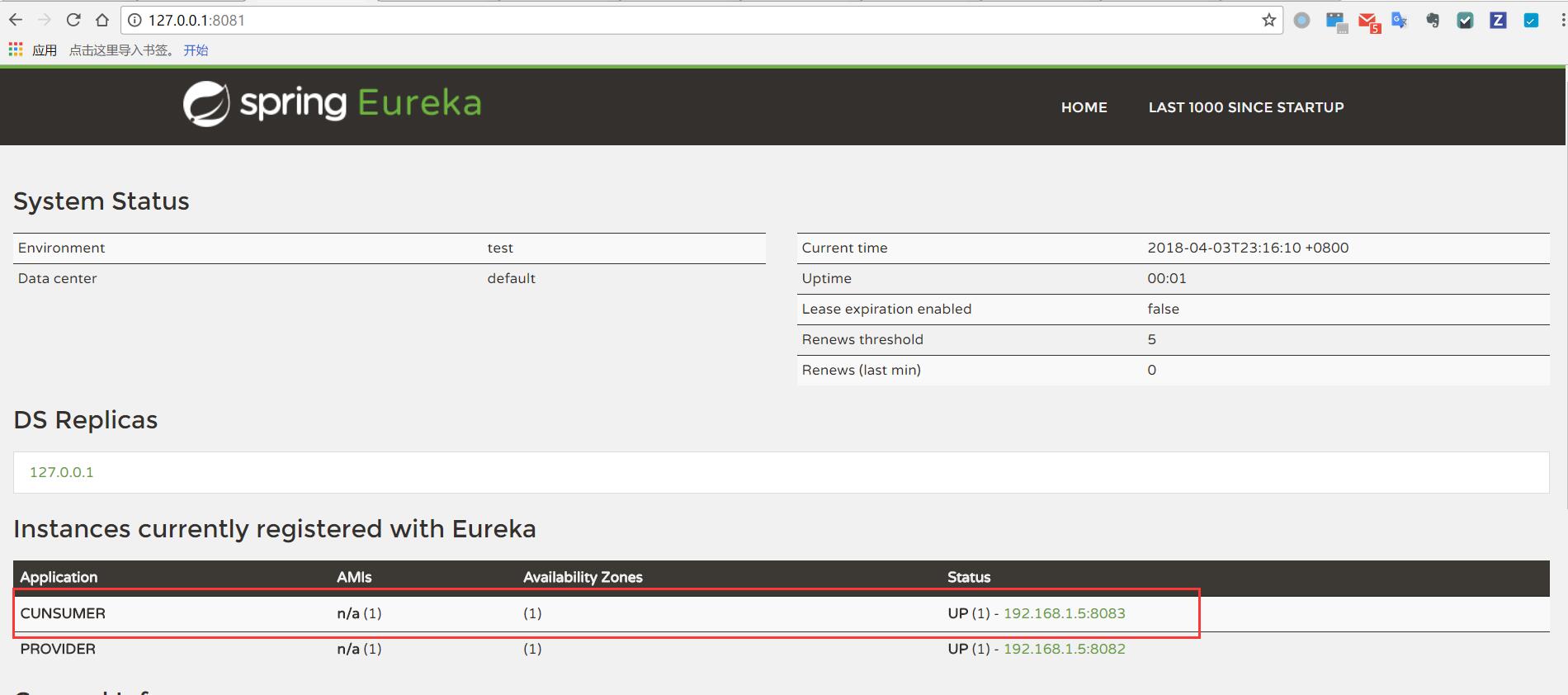
defaultZone: http://127.0.0.1:8081/eureka/

instance:

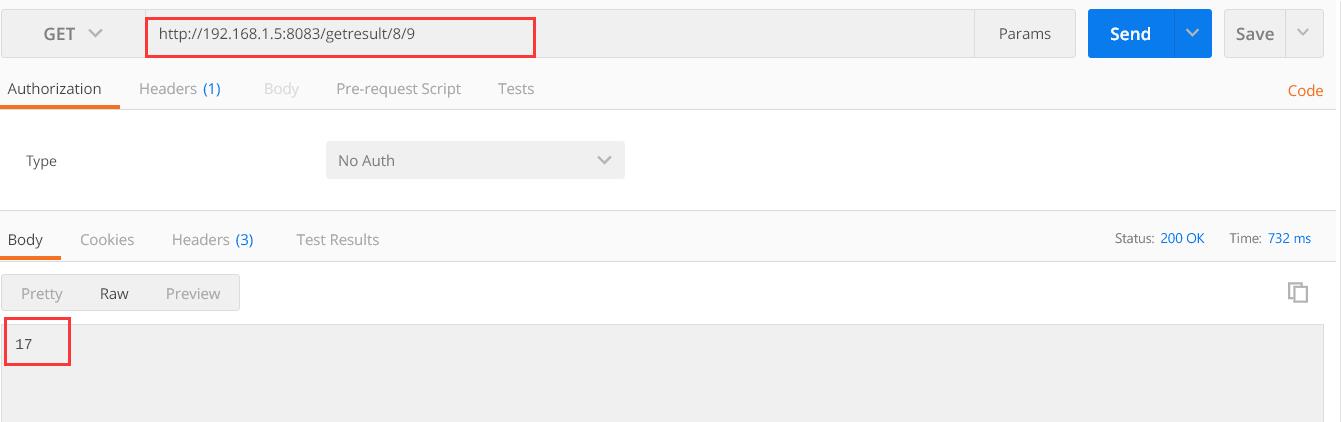
prefer-ip-address: true

instance-id: 192.168.1.5:${server.port}

启动消费方的启动类,可以看到,消费方也被注册到Eureka中去了:



使用postman调用消费方对外提供的接口,我们可以看到,消费方调用了服务提供方提供的服务,正确的返回了结果:



## [eureka的客户端如何使用IP地址来进行注册](https://www.cnblogs.com/jiaoyiping/p/8711861.html)

例子中和我写的代码里,使用的spring-boot的版本是2.0

Eureka的客户端默认是使用hostname来进行注册的,有的时候,hostname是不可靠的,需要使用IP地址来进行注册

name如何正确的使用IP地址来进行注册呢?根据我自己的实践,下边的方法是可行的(基于spring-boot2.0版本)

首先是Eureka的服务器端配置:

eureka.instance.prefer-ip-address=true

然后是eureka的客户端的配置:

eureka.instance.prefer-ip-address=true

eureka.instance.instance-id: ${客户端注册时显示的IP地址}:${server.port}

spring.cloud.inetutils.preferred-networks=想要使用的网卡分配的IP地址的通配符(例如191.168.1.)

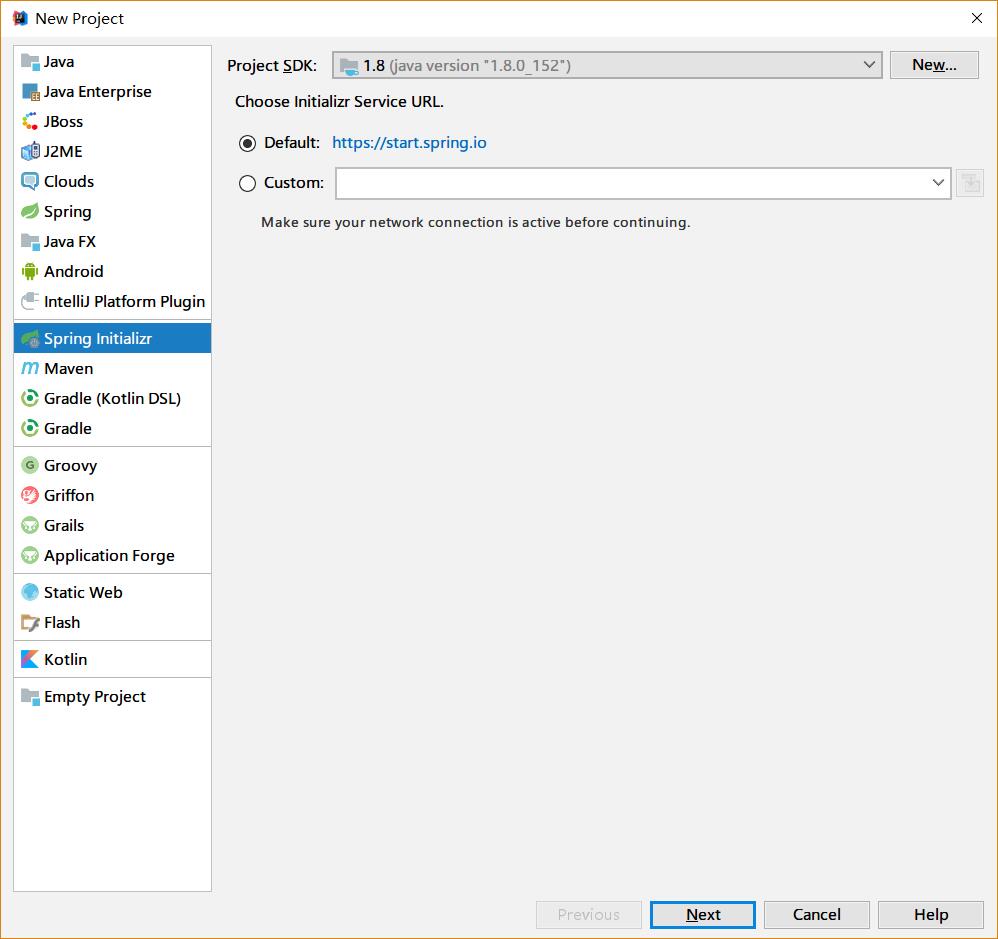
之所以要配置网卡的信息,是因为,如果不配置的话,springcloud会找到第一个不是本地的网卡 来使用这个网卡分配的ip地址来进行注册,但是一般情况下 spring.cloud.inetutils.preferred-networks这个属性,不建议配置在配置文件里, 可以作为环境变量或者是应用的启动信息传入即可,因为每个服务器上的网卡信息都是不一样的,写死了之后,不方便迁移和部署多套环境

服务器端和客户端分别做完上述的配置之后,客户端就可以使用IP地址来到服务器端来进行注册了

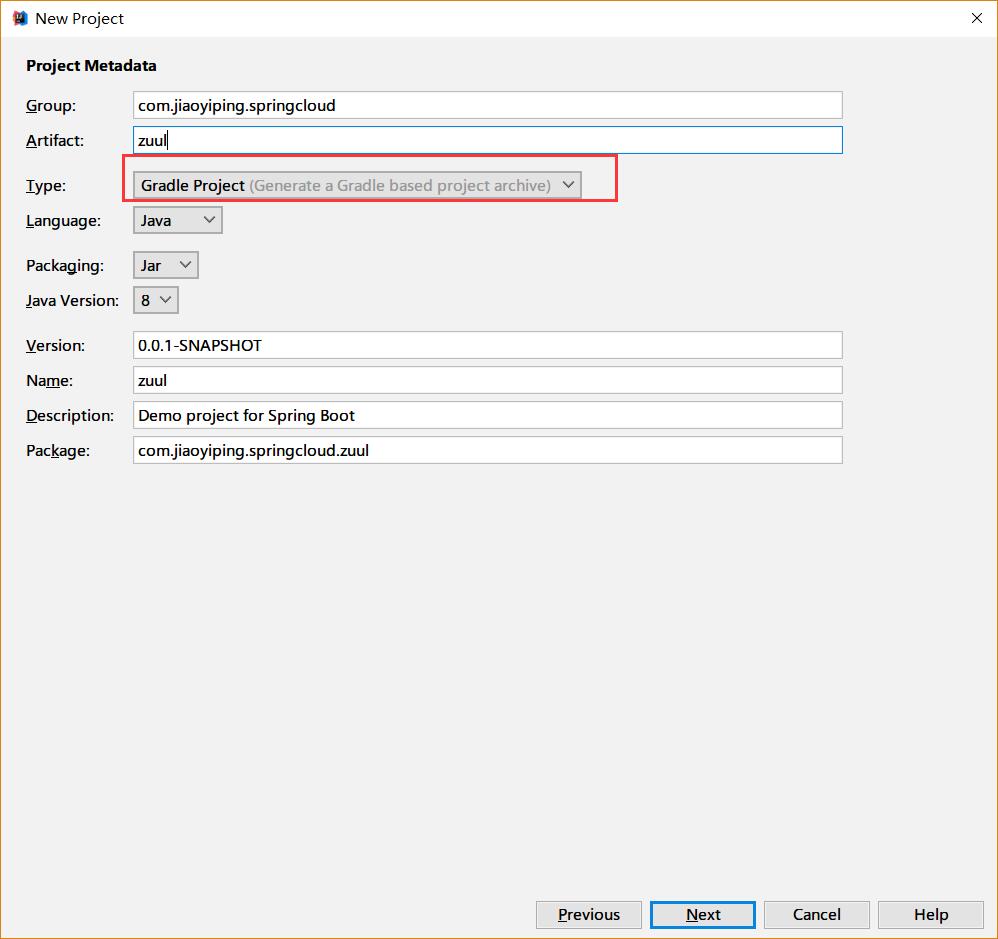
## [基于spring-cloud的微服务(4)API网关zuul](https://www.cnblogs.com/jiaoyiping/p/8727054.html)

API网关是微服务架构中的很重要的一个部分,内部有多个不同的服务提供给外部来使用,API网关可以对外做统一的入口,也可以在网关上做协议转换,权限控制和请求统计和限流等其他的工作  
spring-cloud封装了Netflix提供的开源的API网关实现zuul,我们可以很方便地启动一个zuul网关的实例,并支持向eureka进行注册,并对在eureka上已经注册的服务进行代理

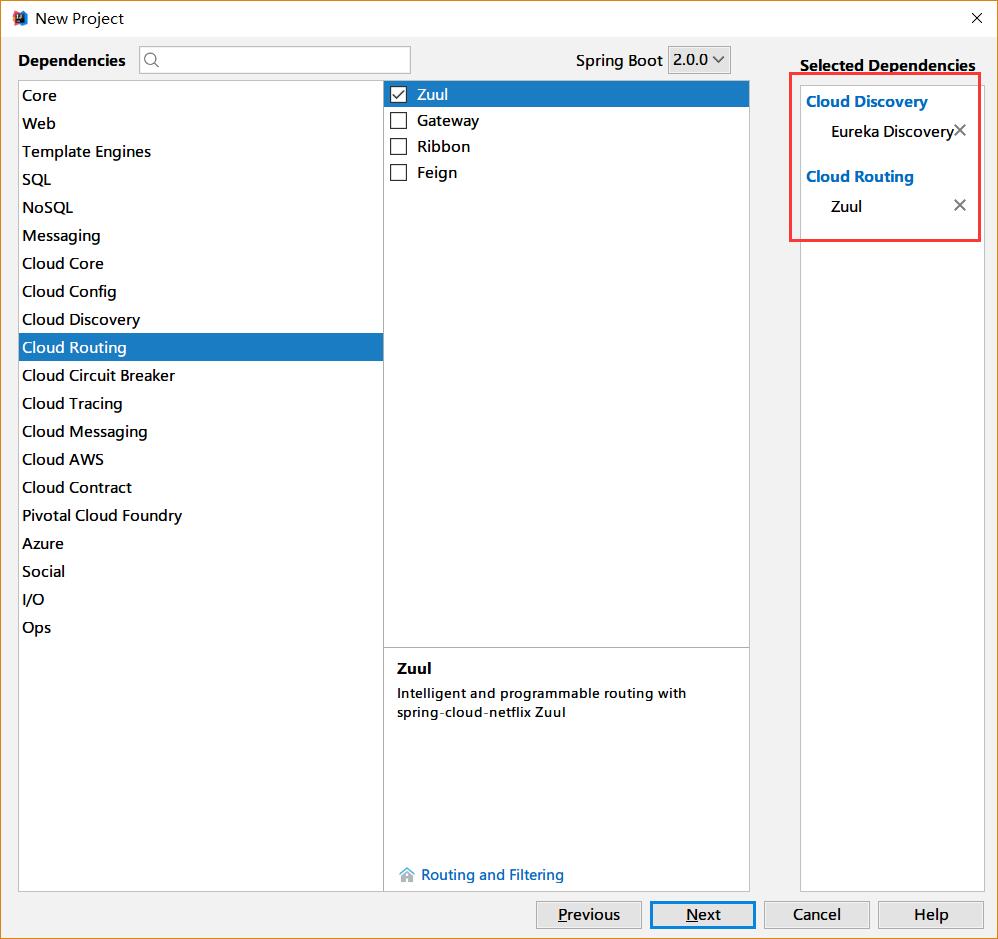
使用IDEA的spring initializer来创建一个zuul项目



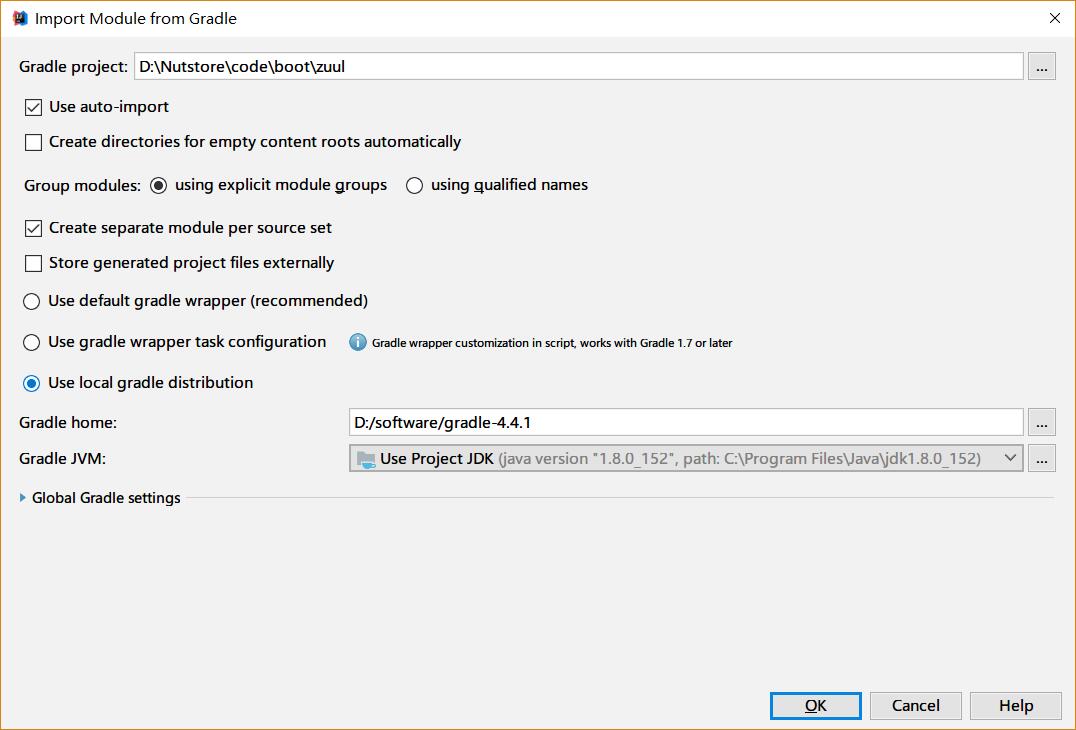
填写相关的group类型等信息,选择使用gradle来进行构建



选择zuul和eureka client



选择项目位置和gradle安装的位置:



在启动类上添加 @EnableZuulProxy 注解,这个注解是@EnableZuulServer的加强版,不仅标识了这是一个zuul Server 而且启用了eureka注册中心和负载均衡ribbon

启动类的代码:

package com.jiaoyiping.springcloud.zuul;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication

@EnableZuulProxy

public class ZuulApplication {

public static void main(String[] args) {

SpringApplication.run(ZuulApplication.class, args);

}

}

zuul最本质的功能是做反向代理,路由转发和对请求的拦截和处理,路由转发的配置在配置文件中,zuul可以做一下几种形式的转发:

将请求重定向到一个外部的URL上,  
如果我们要配置,符合/baidu前缀的请求都重定向到baidu.com去,可以做如下的配置:

zuul:

routes:

baidu:

path: /baidu/\*\*

url: http://www.baidu.com

将请求转发到内部提供的请求路径上去,使用forforward:  
比如,如果我们的网关自己提供了一个以/session开头的服务,我们想让对网关的请求中以/session开头的请求都让网关自己提供的这个服务来处理,则可以进行如下的配置

zuul:

routes:

session:

path: /session/\*\*

url: forward:/session

对已经注册到eureka中的服务进行代理和请求转发,此时只需要提供eureka中服务的serviceid即可

zuul:

routes:

provider:

path: /provider/\*\*

serviceId: PROVIDER

#(去请求目标服务的时候)是否丢掉前缀,根据需求来配置

stripPrefix: true

完整的配置如下:

spring:

application:

name: zuul

cloud:

inetutils:

preferred-networks: 192.168.1.

server:

port: 8084

tomcat:

uri-encoding: UTF-8

servlet:

context-path: /

logging:

config: classpath:logback.xml

zuul:

routes:

baidu:

path: /baidu/\*\*

url: http://www.baidu.com

provider:

path: /provider/\*\*

serviceId: PROVIDER

#(去请求目标服务的时候)是否丢掉前缀,根据需求来配置

stripPrefix: true

session:

path: /session/\*\*

url: forward:/session

eureka:

instance:

hostname: 192.168.1.5

prefer-ip-address: true

instance-id: 192.168.1.5:${server.port}

client:

healthcheck:

enabled: true

registerWithEureka: true

fetchRegistry: true

service-url:

defaultZone: http://127.0.0.1:8081/eureka/

hystrix:

command:

default:

execution:

isolation:

thread:

timeoutInMilliseconds: 25000

ribbon:

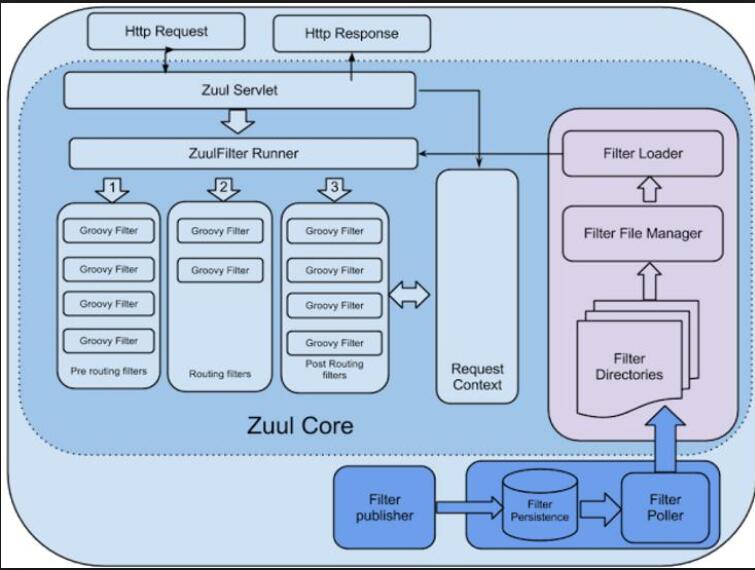
MaxAutoRetries: 2

MaxAutoRetriesNextServer: 3

restclient:

enabled: true

zuul中的Filter的配置,zuul中提供了三种类型的Filter,preFilter,routeFilter和postFilter,分别对应请求中的不同的阶段,针对同一个请求,有一个RequestContext对象,在三个阶段的Filter中进行共享



假设我们要开发一个统计请求时间的功能,需要在preFilter里边记录开始时间,并将整个开始时间放在RequestContext中,在postFilter里边拿到开始时间,用当前的时间减去开始时间,就是请求执行的时间

定义一个preFilter:

package com.jiaoyiping.springcloud.zuul.filter;

import com.netflix.zuul.ZuulFilter;

import com.netflix.zuul.context.RequestContext;

import com.netflix.zuul.exception.ZuulException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

/\*\*

\* Created with Intellij IDEA

\*

\* @author: jiaoyiping

\* Mail: jiaoyiping@gmail.com

\* Date: 2018/04/05

\* Time: 16:24

\* To change this template use File | Settings | Editor | File and Code Templates

\*/

public class TimeCostPreFilter extends ZuulFilter {

public static final String START\_TIME\_KEY = "start\_time";

private Logger logger = LoggerFactory.getLogger(TimeCostPreFilter.class);

@Override

public String filterType() {

return FilterConstants.PRE\_TYPE;

}

@Override

public int filterOrder() {

return 0;

}

/\*\*

\* 判断是否要拦截的逻辑

\*

\* @return

\*/

@Override

public boolean shouldFilter() {

return true;

}

@Override

public Object run() throws ZuulException {

long startTime = System.currentTimeMillis();

RequestContext.getCurrentContext().set(START\_TIME\_KEY, startTime);

return null;

}

}

定义以postFilter:

package com.jiaoyiping.springcloud.zuul.filter;

import com.netflix.zuul.ZuulFilter;

import com.netflix.zuul.context.RequestContext;

import com.netflix.zuul.exception.ZuulException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.netflix.zuul.filters.support.FilterConstants;

/\*\*

\* Created with Intellij IDEA

\*

\* @author: jiaoyiping

\* Mail: jiaoyiping@gmail.com

\* Date: 2018/04/05

\* Time: 16:37

\* To change this template use File | Settings | Editor | File and Code Templates

\*/

public class TimeCostPostFilter extends ZuulFilter {

private static final String START\_TIME\_KE = "start\_time";

private Logger logger = LoggerFactory.getLogger(TimeCostPostFilter.class);

@Override

public String filterType() {

return FilterConstants.POST\_TYPE;

}

@Override

public int filterOrder() {

return 0;

}

@Override

public boolean shouldFilter() {

return true;

}

@Override

public Object run() throws ZuulException {

long startTime = (long) RequestContext.getCurrentContext().get(START\_TIME\_KE);

logger.info("请求完成,耗时{}秒", (System.currentTimeMillis() - startTime) / 1000);

return null;

}

}

在一个配置类中将这两个Filter注入:

package com.jiaoyiping.springcloud.zuul.config;

import com.jiaoyiping.springcloud.zuul.filter.PDSFilter;

import com.jiaoyiping.springcloud.zuul.filter.TimeCostPostFilter;

import com.jiaoyiping.springcloud.zuul.filter.TimeCostPreFilter;

import com.netflix.zuul.ZuulFilter;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

/\*\*

\* Created with Intellij IDEA

\*

\* @author: jiaoyiping

\* Mail: jiaoyiping@gmail.com

\* Date: 2018/04/05

\* Time: 17:32

\* To change this template use File | Settings | Editor | File and Code Templates

\*/

@Configuration

public class FilterConfig {

@Bean

public ZuulFilter timeCostPreFilter() {

return new TimeCostPreFilter();

}

@Bean

public ZuulFilter timeCostPostFilter() {

return new TimeCostPostFilter();

}

@Bean

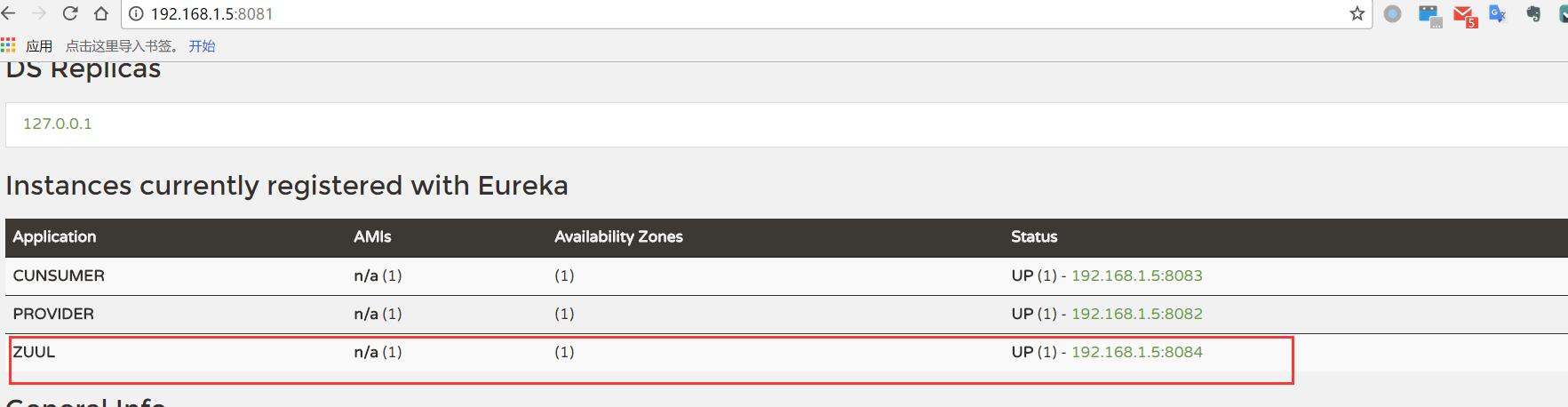
public ZuulFilter pdsFilter() {

return new PDSFilter();

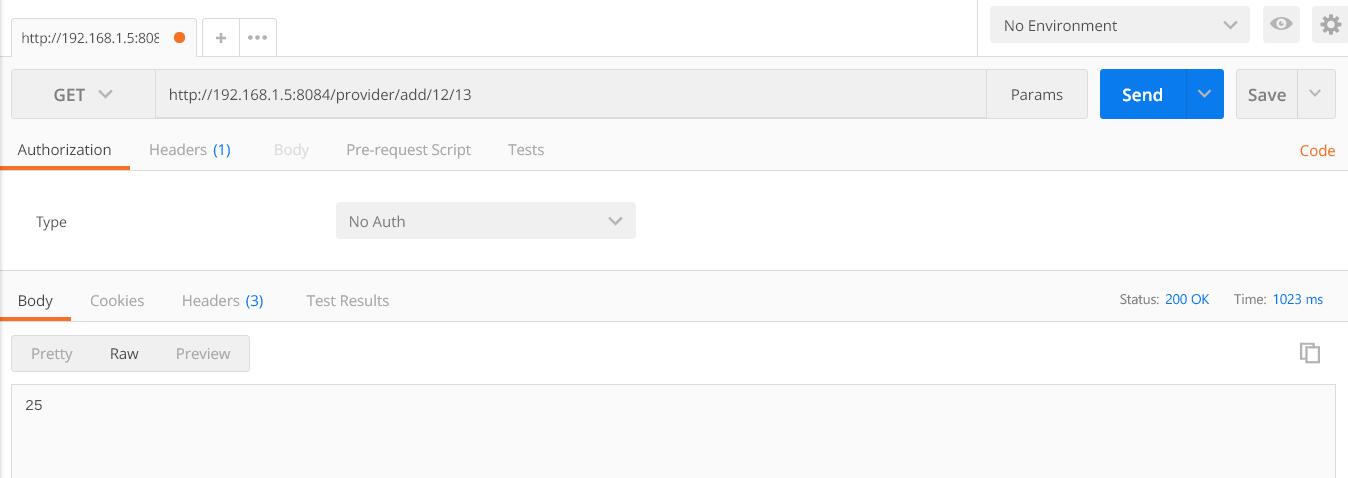
}

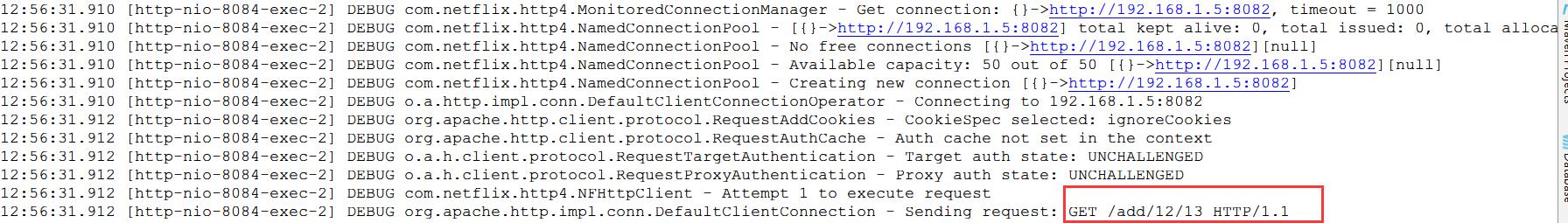
}

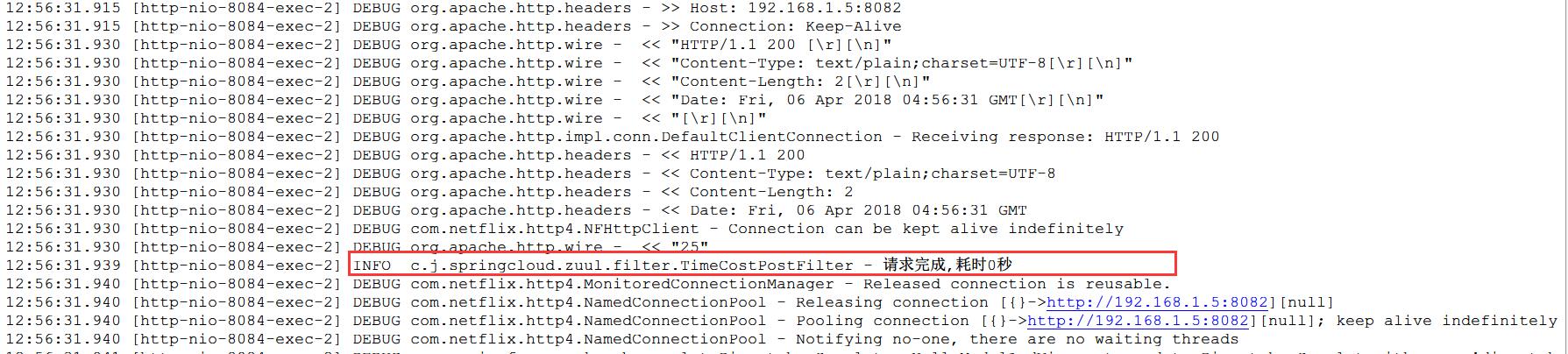
启动项目,可以发现,zuul网关已经注册到了eureka上:



请求provide对应的地址,发现,zuul可以成功地调用eureka上对应的服务,并将结果正确返回:







# 每特

## 微服务架构

## 什么是分布式

不同模块部署在不同服务器上

作用：分布式解决网站高并发带来问题

## 什么是集群

多台服务器部署相同应用构成一个集群

作用：通过负载均衡设备共同对外提供服务

## 什么是RPC

RPC 的全称是 Remote Procedure Call 是一种进程间通信方式。  
它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而**不用程序员显式编码这个远程调用的细节**。即无论是调用本地接口/服务的还是远程的接口/服务，本质上编写的调用代码基本相同。  
比如两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数或者方法，由于不在一个内存空间，不能直接调用，这时候需要通过就可以应用RPC框架的实现来解决

#### restful、soap、rpc

（1）RESTful是一种架构设计风格，提供了设计原则和约束条件，而不是架构。而满足这些约束条件和原则的应用程序或设计就是 RESTful架构或服务。  
（2）SOAP，简单对象访问协议是一种数据交换协议规范，  
是一种轻量的、简单的、基于XML的协议的规范。SOAP协议和HTTP协议一样，都是底层的通信协议，只是请求包的格式不同而已，SOAP包是XML格式的。  
SOAP的消息是基于xml并封装成了符合http协议，因此，它符合任何路由器、 防火墙或代理服务器的要求。  
soap可以使用任何语言来完成，只要发送正确的soap请求即可，基于soap的服务可以在任何平台无需修改即可正常使用。  
（3）RPC就是从一台机器（客户端）上通过参数传递的方式调用另一台机器（服务器）上的一个函数或方法（可以统称为服务）并得到返回的结果。  
RPC 会隐藏底层的通讯细节（不需要直接处理Socket通讯或Http通讯）  
RPC 是一个请求响应模型。客户端发起请求，服务器返回响应（类似于Http的工作方式）  
RPC 在使用形式上像调用本地函数（或方法）一样去调用远程的函数（或方法）。

#### rpc远程调用框架

几种比较典型的RPC的实现和调用框架。   
（1）RMI实现，利用java.rmi包实现，基于Java远程方法协议(Java Remote Method Protocol)   
和java的原生序列化。   
（2）Hessian，是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。 基于HTTP协议，采用二进制编解码。   
（3）thrift是一种可伸缩的跨语言服务的软件框架。thrift允许你定义一个描述文件，描述数据类型和服务接口。依据该文件，编译器方便地生成RPC客户端和服务器通信代码。

（4）SpringCloud 为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等。

## 什么是SOA

业务系统分解为多个组件，让每个组件都独立提供离散，自治，可复用的服务能力

通过服务的组合和编排来实现上层的业务流程

作用：简化维护,降低整体风险,伸缩灵活

## 什么是微服务

架构设计概念,各服务间隔离（分布式也是隔离）,自治（分布式依赖整体组合）其它特性(单一职责,边界,异步通信,独立部署)是分布式概念的跟严格执行

SOA到微服务架构的演进过程

作用：各服务可独立应用，组合服务也可系统应用(巨石应用[monolith]的简化实现策略-平台思想)

## 使用RPC http技术实现会员与订单系统通讯

## 微服务架构

## SpringCloud

SpringCloud 为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等。它运行环境简单，可以在开发人员的电脑上跑。另外说明spring cloud是基于Springboot的，所以需要开发中对Springboot有一定的了解，如果不了解的话可以看蚂蚁课堂SpringBoot课程。

## 服务提供者与消费关系

服务提供者:提供服务被人调用

消费者:调用被人服务

## 服务的注册与发现(Eureka)

在这里，我们需要用的的组件上Spring Cloud Netflix的Eureka ,eureka是一个服务注册和发现模块。

### 服务注册

#### 创建eurekaserver 项目

#### 引入maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.2.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <!--eureka server -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <!-- spring boot test -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.RC1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

#### 配置application.yml

|  |
| --- |
| server:  port: 8761  eureka:  instance:  hostname: localhost  client:  registerWithEureka: **false**  fetchRegistry: **false**  serviceUrl:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ |

#### 启动EurekaServer

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  public class EurekaServerApplication {  public static void main(String[] args) {  SpringApplication.*run*(EurekaServerApplication.class, args);  }  } |

1. eureka.client.registerWithEureka=**true**  #是否将自身注册
2. eureka.client.fetchRegistry=**false**    #如果为**true**，启动时报警.

#### 4.5打开eureka server 界面的

|  |
| --- |
| [http://localhost:8761](http://localhost:8761/) ,界面如下：  Paste_Image.png  No application available 没有服务被发现 ……^\_^  因为没有注册服务当然不可能有服务被发现了。 |

### 服务提供者

创建一个服务提供者 (eureka client),当client向server注册时，它会提供一些元数据，例如主机和端口，URL，主页等。Eureka server 从每个client实例接收心跳消息。 如果心跳超时，则通常将该实例从注册server中删除。

#### 创建项目eurekaclient

#### 引入maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.2.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.RC1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

#### application.yml配置

|  |
| --- |
| eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  server:  port: 8762  spring:  application:  name: service-hi |

#### 发布服务

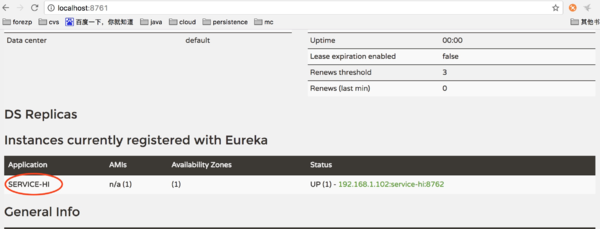
通过注解@EnableEurekaClient 表明自己是一个eurekaclient.

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @RestController  public class ServiceHiApplication {  public static void main(String[] args) {  SpringApplication.*run*(ServiceHiApplication.class, args);  }  @Value("${server.port}")  String port;  @RequestMapping("/hi")  public String home(@RequestParam String name) {  return "hi " + name + ",i am from port:" + port;  }  } |

#### 演示效果

需要指明spring.application.name,这个很重要，这在以后的服务与服务之间相互调用一般都是根据这个name 。

启动工程，打开[http://localhost:8761](http://localhost:8761/) ，即eureka server 的网址：



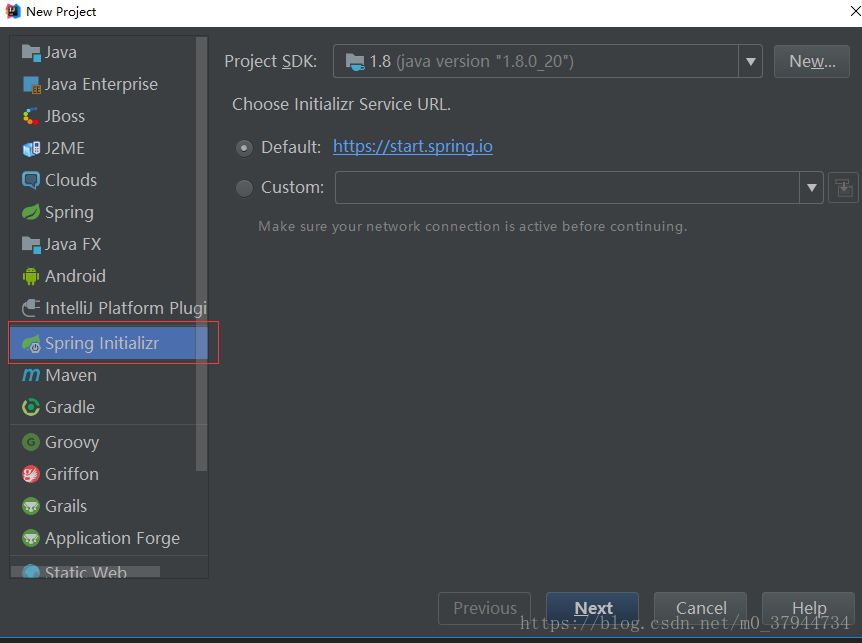
你会发现一个服务已经注册在服务中了，服务名为SERVICE-HI ,端口为7862

这时打开 <http://localhost:8762/hi?name=forezp> ，你会在浏览器上看到 :

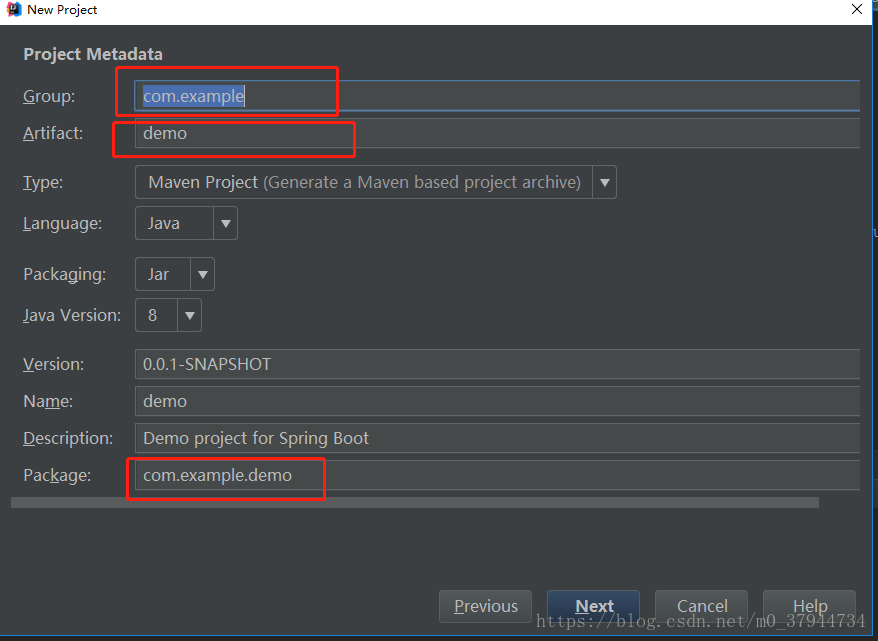
hi forezp,i am from port:8762

## IDEA创建eureka

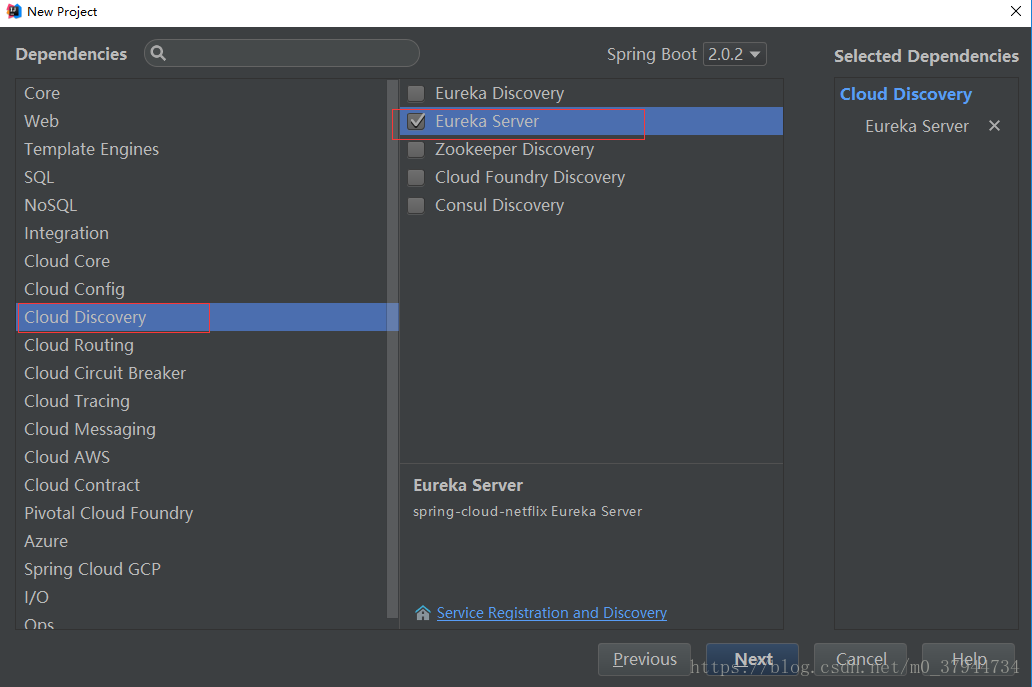
一、首先new 一个project



二、点击Next之后填写Group、Artifact、和创建主类的包路劲package

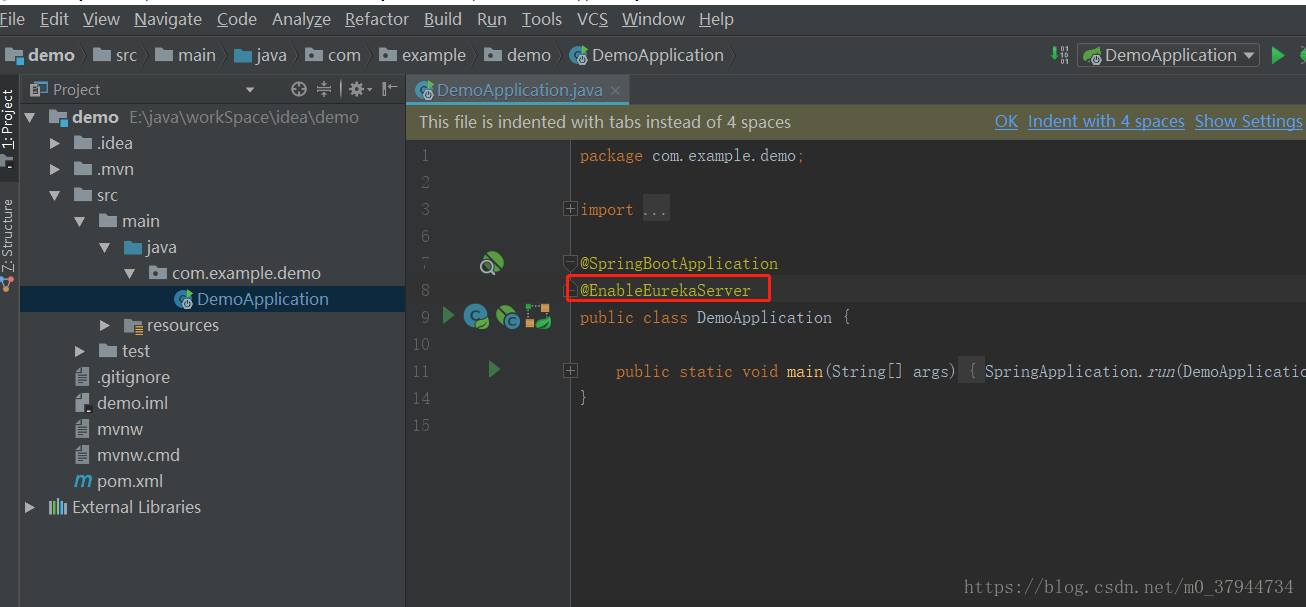


三、下一步之后选择



四、下一步后点击[Finish](https://www.baidu.com/s?wd=Finish&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)

五、在主类添加@EnableEurekaServer注解



六、在配置文件添加如下内容

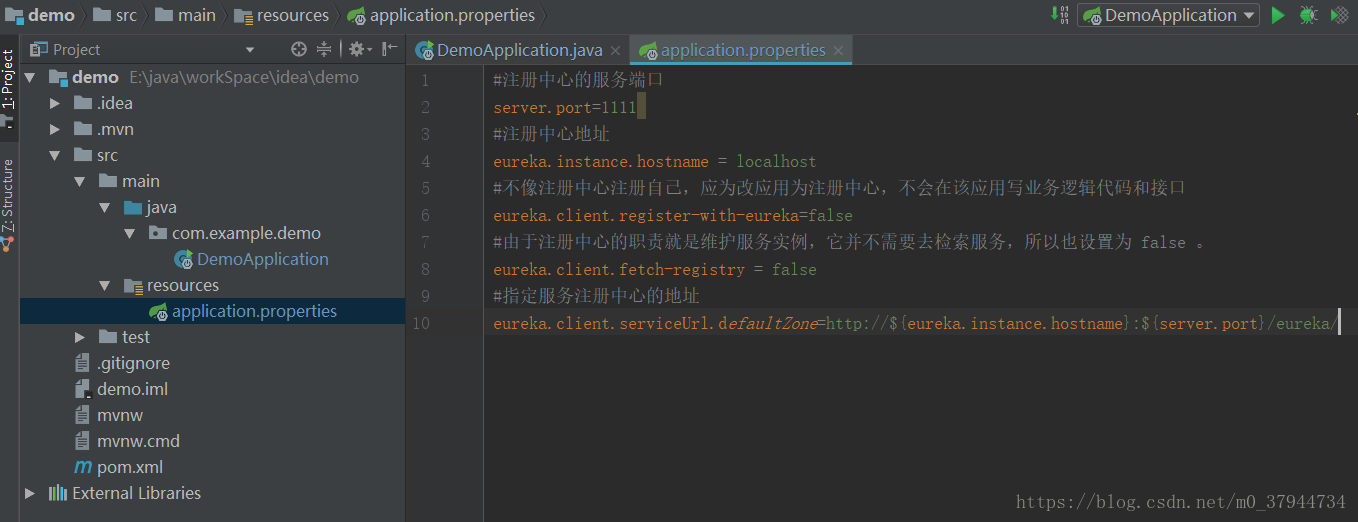
server.port=1111

eureka.instance.hostname=localhost

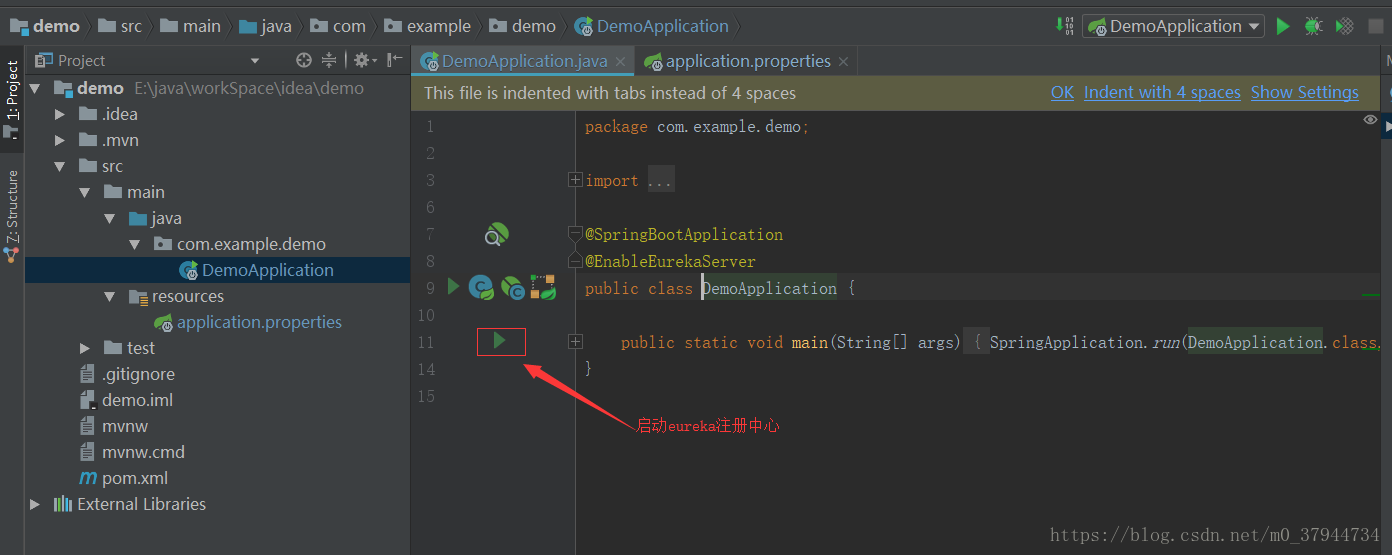
eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

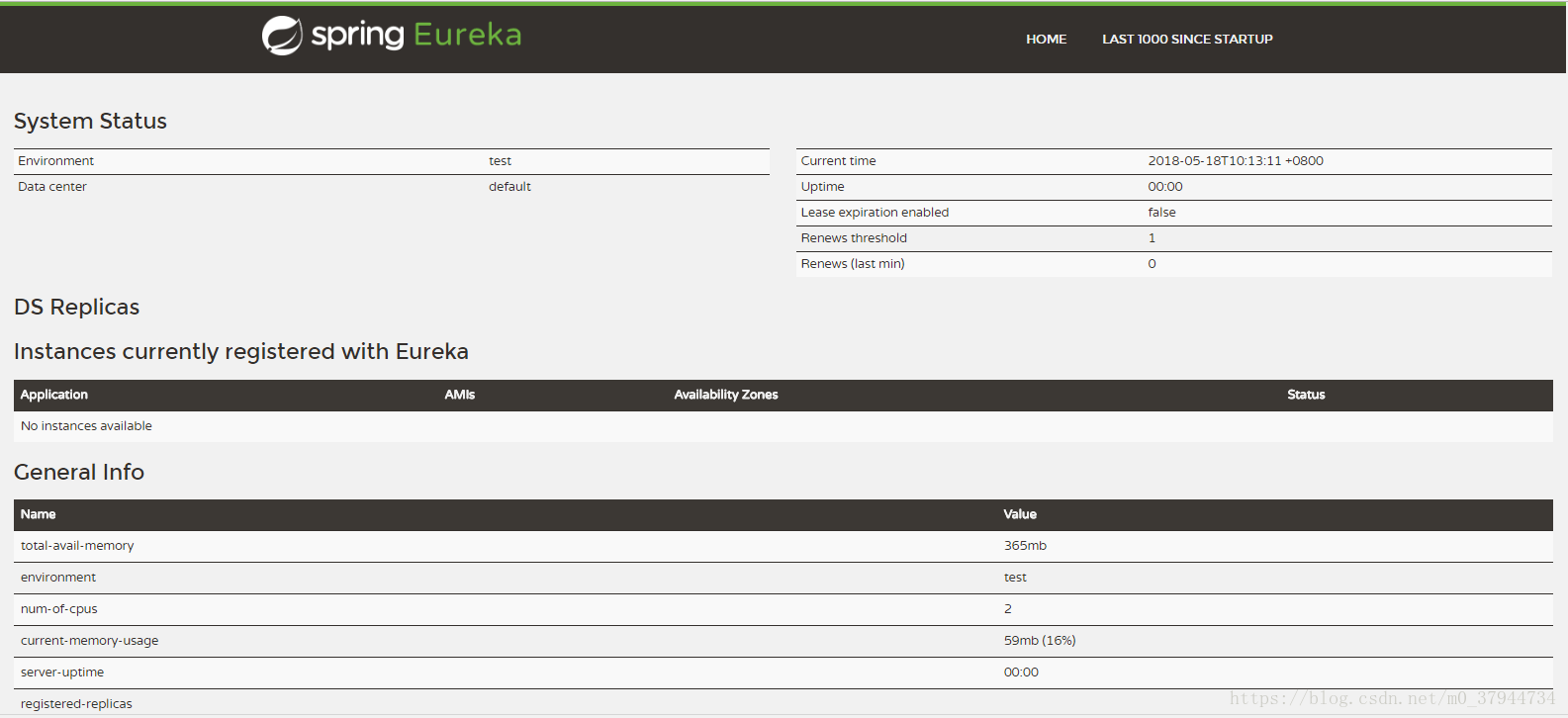
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/



七、启动



八、打开  http://localhost:1111/  出现，安装完成。一个简单的注册中心就搭建完成了。



## 服务消费者(rest+ribbon)

在微服务架构中，业务都会被拆分成一个独立的服务，服务与服务的通讯是基于http restful的。Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign。

### 什么是ribbon

ribbon是一个负载均衡客户端，可以很好的控制htt和tcp的一些行为。Feign默认集成了ribbon。

## 准备工作

这一篇文章基于上一篇文章的工程，启动eureka-server 工程；启动service-hi工程，它的端口为8762；将service-hi的配置文件的端口改为8763,并启动，这时你会发现：service-hi在eureka-server注册了2个实例，这就相当于一个小的集群。访问localhost:8761如图所示：



## 建立一个消费者

#### 创建一个工程为service-ribbon

重新新建一个spring-boot工程，取名为：service-ribbon;   
在它的pom.xml文件分别引入起步依赖spring-cloud-starter-eureka、spring-cloud-starter-ribbon、spring-boot-starter-web，代码如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.2.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.RC1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

#### application.yml配置

在工程的配置文件指定服务的注册中心地址为<http://localhost:8761/eureka/>，程序名称为 service-ribbon，程序端口为8764。配置文件application.yml如下：

|  |
| --- |
| eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  server:  port: 8764  spring:  application:  name: service-ribbon |

#### 启动类@EnableDiscoveryClient

在工程的启动类中,通过@EnableDiscoveryClient向服务中心注册；并且向程序的ioc注入一个bean: restTemplate;并通过@LoadBalanced注解表明这个restRemplate开启负载均衡的功能。

|  |
| --- |
| @EnableAutoConfiguration  @ComponentScan(basePackages={"com.itmayiedu.controller","com.itmayiedu.service","com.itmayiedu.app"})  @EnableDiscoveryClient  **public** **class** ServiceRibbonApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ServiceRibbonApplication.**class**, args);  }  @Bean  @LoadBalanced  RestTemplate restTemplate() {  **return** **new** RestTemplate();  }  } |

#### 编写一个service

|  |
| --- |
| @Service  **public** **class** HelloService {  @Autowired  RestTemplate restTemplate;  **public** String hiService(String name) {  **return** restTemplate.getForObject("http://SERVICE-HI/hi?name="+name,String.**class**);  }  } |

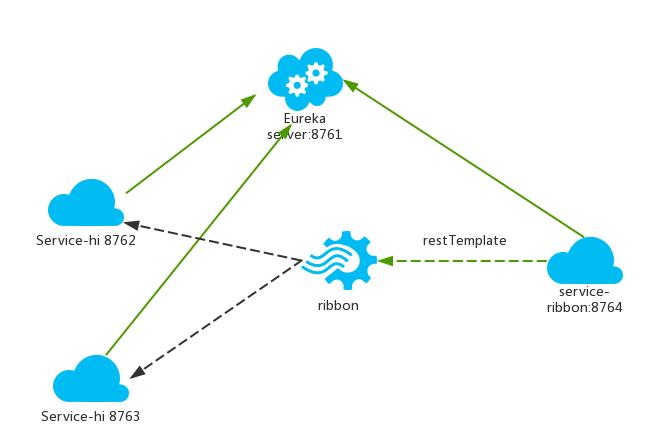
#### 编写一个控制器层

|  |
| --- |
| @RestController  **public** **class** HelloControler {  @Autowired  HelloService helloService;  @RequestMapping(value = "/hi")  **public** String hi(@RequestParam String name){  **return** helloService.hiService(name);  }  } |

#### 5.1.3.6 演示效果

|  |
| --- |
| 在浏览器上多次访问<http://localhost:8764/hi?name=forezp>，浏览器交替显示：  hi forezp,i am from port:8762  hi forezp,i am from port:8763  这说明当我们通过调用restTemplate.getForObject(“[http://SERVICE-HI/hi?name=](http://service-hi/hi?name=)“+name,String.class)方法时，已经做了负载均衡，访问了不同的端口的服务实例。 |

#### 5.1.3.7 此时架构



* 一个服务注册中心，eureka server,端口为8761
* service-hi工程跑了两个实例，端口分别为8762,8763，分别向服务注册中心注册
* sercvice-ribbon端口为8764,向服务注册中心注册
* 当sercvice-ribbon通过restTemplate调用service-hi的hi接口时，因为用ribbon进行了负载均衡，会轮流的调用service-hi：8762和8763 两个端口的hi接口；
* @LoadBalanced注解表明这个restRemplate开启负载均衡的功能。

## [服务消费者（Feign）](http://blog.csdn.net/forezp/article/details/69808079)

轻量级rest客户端，和resttemple一样，可以把远程调用过程简化，不用像定义url那么麻烦，像定义本地方法一样，进行远程调用，

首先定义一个接口，在接口方法上用springmvc 注解url和http类型，并且在接口类上标示feignclient注解，指定服务名，最后在springboot启动类上开启enableFeignclients注解，一旦开启，会扫描所有标记feignclient的接口，生成client对象，注入到spring容器中，就可以在其他类中注入接口的代理对象，从而使用这个接口，发起远程调用。

### feign注意点

SpringCloud对Feign进行了增强兼容了SpringMVC的注解 ，我们在使用SpringMVC的注解时需要注意：

1、feignClient接口 有参数在参数必须加@PathVariable("XXX")和@RequestParam("XXX")

2、feignClient返回值为复杂对象时其类型必须有无参构造函数。

### 什么是Feign

Feign是一个声明式的伪Http客户端，它使得写Http客户端变得更简单。使用Feign，只需要创建一个接口并注解。它具有可插拔的注解特性，可使用Feign 注解和JAX-RS注解。Feign支持可插拔的编码器和解码器。Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

简而言之：

Feign 采用的是基于接口的注解

Feign 整合了ribbon

### 6.2 准备工作

继续用上一节的工程， 启动eureka-server，端口为8761; 启动service-hi 两次，端口分别为8762 、8773.

#### 6.2.1 准备工创建一个feign的服务

新建一个spring-boot工程，取名为serice-feign，在它的pom文件引入Feign的起步依赖spring-cloud-starter-feign、Eureka的起步依赖spring-cloud-starter-eureka、Web的起步依赖spring-boot-starter-web，代码如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.2.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.RC1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

#### 6.2.2 application.yml配置

在工程的配置文件application.yml文件，指定程序名为service-feign，端口号为8765，服务注册地址为<http://localhost:8761/eureka/> ，代码如下：

|  |
| --- |
| eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  server:  port: 8765  spring:  application:  name: service-feign |

#### 6.2.3 定义一个feign接口

|  |
| --- |
| **@FeignClient(value = "service-hi")**  **public interface SchedualServiceHi {**  **@RequestMapping(value = "/hi", method = RequestMethod.*GET*)**  **String sayHiFromClientOne(@RequestParam(value = "name") String name);**  **}** |

#### 6.2.4一个”/hi”的API接口

|  |
| --- |
| @RestController  **public** **class** HiController {  @Autowired  SchedualServiceHi schedualServiceHi;  @RequestMapping(value = "/hi",method = RequestMethod.***GET***)  **public** String sayHi(@RequestParam String name){  **return** schedualServiceHi.sayHiFromClientOne(name);  }  } |

#### 6.2.5启动方式

|  |
| --- |
| **@SpringBootApplication**  **@EnableDiscoveryClient**  **@EnableFeignClients**  **public class SericeFeign {**  **public static void main(String[] args) {**  **SpringApplication.*run*(SericeFeign.class, args);**  **}**  **}** |

#### 6.2.6 演示效果

|  |
| --- |
| 启动程序，多次访问<http://localhost:8765/hi?name=forezp>,浏览器交替显示：  hi forezp,i am from port:8762  hi forezp,i am from port:8763 |

## Hystrix断路器

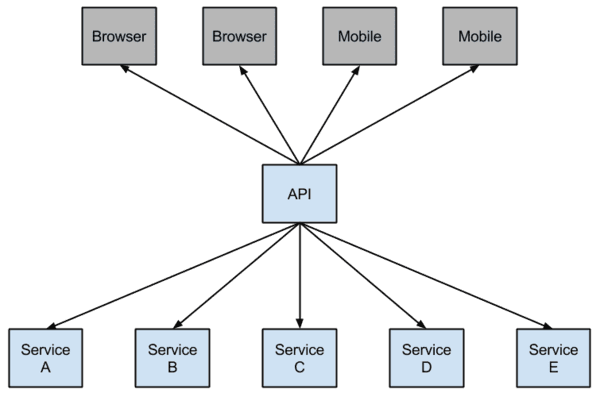
解决调用失败。

在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用（RPC），在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

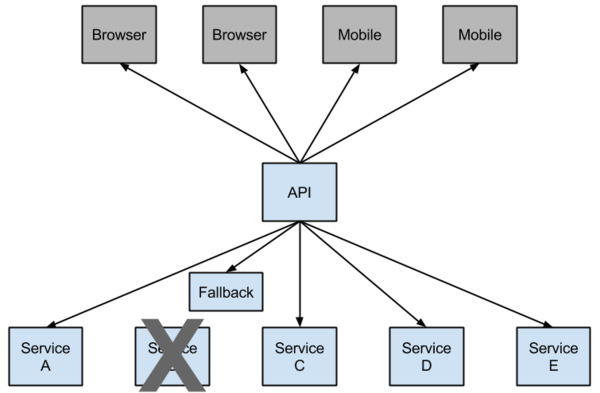
为了解决这个问题，业界提出了断路器模型。

### 什么是Hystrix

Netflix开源了Hystrix组件，实现了断路器模式，SpringCloud对这一组件进行了整合。 在微服务架构中，一个请求需要调用多个服务是非常常见的，如下图：



较底层的服务如果出现故障，会导致连锁故障。当对特定的服务的调用的不可用达到一个阀值（Hystric 是5秒20次） 断路器将会被打开。



断路打开后，可用避免连锁故障，fallback方法可以直接返回一个固定值。

### 准备工作

这篇文章基于上一篇文章的工程，首先启动上一篇文章的工程，启动eureka-server 工程；启动service-hi工程，它的端口为8762。

#### 在ribbon使用断路器

改造serice-ribbon 工程的代码，首先在pox.xml文件中加入spring-cloud-starter-hystrix的起步依赖：

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

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

#### 添加启动类断路器注解

|  |
| --- |
| package com.ysy.ribbon;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.client.discovery.EnableDiscoveryClient;  import org.springframework.cloud.client.loadbalancer.LoadBalanced;  import org.springframework.cloud.netflix.hystrix.EnableHystrix;  import org.springframework.context.annotation.Bean;  import org.springframework.web.client.RestTemplate;  @SpringBootApplication  @EnableDiscoveryClient  @EnableHystrix  public class RibbonApplication {  public static void main(String[] args) {  SpringApplication.run(RibbonApplication.class, args);  }  @Bean  @LoadBalanced  RestTemplate restTemplate() {  return new RestTemplate();  }  } |

#### 改造service

改造HelloService类，在hiService方法上加上@HystrixCommand注解。该注解对该方法创建了熔断器的功能，并指定了fallbackMethod熔断方法，熔断方法直接返回了一个字符串，字符串为”hi,”+name+”,sorry,error!”，代码如下：

|  |
| --- |
| @Service  **public** **class** HelloService {  @Autowired  RestTemplate restTemplate;  @HystrixCommand(fallbackMethod = "hiError")  **public** String hiService(String name) {  **return** restTemplate.getForObject("http://SERVICE-HI/hi?name=" + name, String.**class**);  }  **public** String hiError(String name) {  **return** "hi," + name + ",sorry,error!";  }  } |

#### 演示效果

此时关闭 service-hi 工程，当我们再访问<http://localhost:8764/hi?name=forezp>，浏览器会显示：

hi ,forezp,orry,error!

这就说明当 service-hi 工程不可用的时候，service-ribbon调用 service-hi的API接口时，会执行快速失败，直接返回一组字符串，而不是等待响应超时，这很好的控制了容器的线程阻塞。

### Feign中使用断路器

Feign是自带断路器的，在D版本的Spring Cloud中，它没有默认打开。需要在配置文件中配置打开它，在配置文件加以下代码：

feign.hystrix.enabled=true

基于service-feign工程进行改造，只需要在FeignClient的SchedualServiceHi接口的注解中加上fallback的指定类就行了：

|  |
| --- |
| @FeignClient(value = "service-hi",fallback=SchedualServiceHiHystric.class)  public interface SchedualServiceHi {  @RequestMapping(value = "/hi", method = RequestMethod.GET)  String sayHiFromClientOne(@RequestParam(value = "name") String name);  } |

SchedualServiceHiHystric需要实现SchedualServiceHi 接口，并注入到Ioc容器中，代码如下：

|  |
| --- |
| @Component  public class SchedualServiceHiHystric implements SchedualServiceHi {  public String sayHiFromClientOne(String name) {  return "sorry " + name;  }  } |

开启hystrix

feign:

hystrix:

enabled: true

### Hystrix Dashboard (断路器：Hystrix 仪表盘)

基于service-ribbon 改造，Feign的改造和这一样。

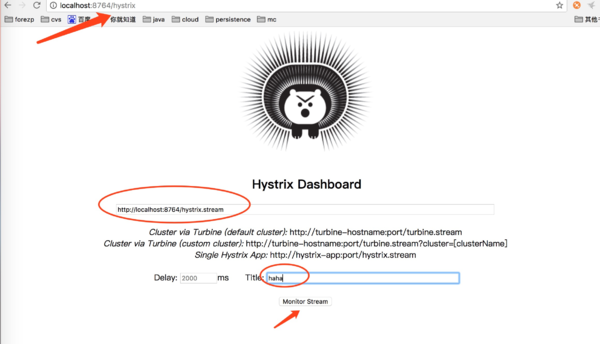
首选在pom.xml引入spring-cloud-starter-hystrix-dashboard的起步依赖：

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>  </dependency> |

在主程序启动类中加入@EnableHystrixDashboard注解，开启hystrixDashboard：

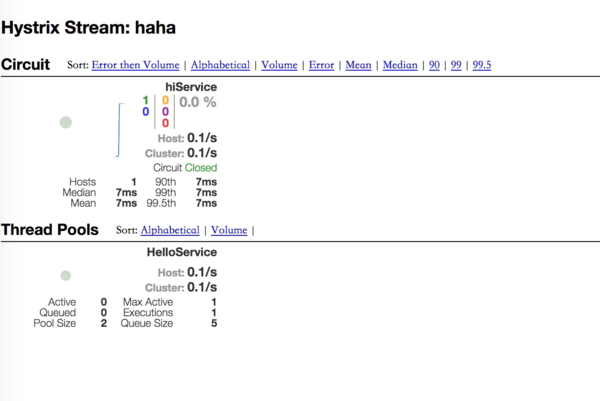
|  |
| --- |
| @EnableAutoConfiguration  @ComponentScan(basePackages = { "com.itmayiedu.controller", "com.itmayiedu.service", "com.itmayiedu.app" })  @EnableHystrixDashboard  @EnableDiscoveryClient  @EnableHystrix  **public** **class** ServiceRibbonApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ServiceRibbonApplication.**class**, args);  }  @Bean  @LoadBalanced  RestTemplate restTemplate() {  **return** **new** RestTemplate();  }  } |

打开浏览器：访问<http://localhost:8764/hystrix>,界面如下：



点击monitor stream，进入下一个界面，访问：<http://localhost:8764/hi?name=forezp>

此时会出现监控界面：



## 使用Zuul构建API Gateway

### 8.1什么是API Gateway？

在Spring Cloud微服务系统中，一种常见的负载均衡方式是，客户端的请求首先经过负载均衡（zuul、Ngnix），再到达服务网关（zuul集群），然后再到具体的服务。

### 8.2 什么是Zuul？

Routing in an integral part of a microservice architecture. For example, / may be mapped to your web application, /api/users is mapped to the user service and /api/shop is mapped to the shop service. Zuul is a JVM based router and server side load balancer by Netflix.

路由在微服务架构的一个组成部分。 例如，/可以映射到您的Web应用程序，/ api / users映射到用户服务，并且/ api / shop映射到商店服务。 Zuul是Netflix的基于JVM的路由器和服务器端负载均衡器。

其功能包括 

验证

见解

压力测试

金丝雀测试

动态路由

服务迁移

减载

安全

静态响应处理

主动/主动流量管理

Zuul的规则引擎允许规则和过滤器基本上用任何JVM语言编写，内置支持Java和Groovy。

### 8.3 创建service-zuul工程

其pom.xml文件如下：

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.2.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Dalston.RC1</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

### 8.4 applicaton类

在其入口applicaton类加上注解@EnableZuulProxy，开启zuul的功能：

|  |
| --- |
| **@EnableZuulProxy**  **@EnableEurekaClient**  **@SpringBootApplication**  **public class ServiceZuulApplication {**  **public static void main(String[] args) {**  **SpringApplication.*run*(ServiceZuulApplication.class, args);**  **}**  **}** |

### 8.5 application.yml配置

|  |
| --- |
| eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  server:  port: 8769  spring:  application:  name: service-zuul  zuul:  routes:  api-a:  path: /api-a/\*\*  service-id: service-ribbon  api-b:  path: /api-b/\*\*  service-id: service-feign |

首先指定服务注册中心的地址为<http://localhost:8761/eureka/>，服务的端口为8769，服务名为service-zuul；以/api-a/ 开头的请求都转发给service-ribbon服务；以/api-b/开头的请求都转发给service-feign服务；

依次运行这五个工程;打开浏览器访问：<http://localhost:8769/api-a/hi?name=forezp> ;浏览器显示：

hi forezp,i am from port:8762

### 8.6服务过滤

zuul不仅只是路由，并且还能过滤，做一些安全验证。继续改造工程；

|  |
| --- |
| @Component  public class MyFilter extends ZuulFilter{  private static Logger log = LoggerFactory.getLogger(MyFilter.class);  @Override  public String filterType() {  return "pre";  }  @Override  public int filterOrder() {  return 0;  }  @Override  public boolean shouldFilter() {  return true;  }  @Override  public Object run() {  RequestContext ctx = RequestContext.getCurrentContext();  HttpServletRequest request = ctx.getRequest();  log.info(String.format("%s >>> %s", request.getMethod(), request.getRequestURL().toString()));  Object accessToken = request.getParameter("token");  if(accessToken == null) {  log.warn("token is empty");  ctx.setSendZuulResponse(false);  ctx.setResponseStatusCode(401);  try {  ctx.getResponse().getWriter().write("token is empty");  }catch (Exception e){}  return null;  }  log.info("ok");  return null;  }  } |

* filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：   
  + pre：路由之前
  + routing：路由之时
  + post： 路由之后
  + error：发送错误调用
  + filterOrder：过滤的顺序
  + shouldFilter：这里可以写逻辑判断，是否要过滤，本文true,永远过滤。
  + run：过滤器的具体逻辑。可用很复杂，包括查sql，nosql去判断该请求到底有没有权限访问。

这时访问：<http://localhost:8769/api-a/hi?name=forezp> ；网页显示：

token is empty

## 分布式配置中心

[分布式配置中心(Spring Cloud Config)](http://blog.csdn.net/forezp/article/details/70037291)，在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件spring cloud config ，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。在spring cloud config 组件中，分两个角色，一是config server，二是config client。

### 9.1 构建Config Server

创建一个spring-boot项目，取名为config-server,其pom.xml:

|  |
| --- |
| **<parent>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-parent</artifactId>**  **<version>1.5.2.RELEASE</version>**  **<relativePath /> <!-- lookup parent from repository -->**  **</parent>**  **<properties>**  **<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**  **<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>**  **<java.version>1.8</java.version>**  **</properties>**  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-config-server</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-test</artifactId>**  **<scope>test</scope>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-eureka</artifactId>**  **</dependency>**  **</dependencies>**  **<dependencyManagement>**  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-dependencies</artifactId>**  **<version>Camden.SR6</version>**  **<type>pom</type>**  **<scope>import</scope>**  **</dependency>**  **</dependencies>**  **</dependencyManagement>**  **<build>**  **<plugins>**  **<plugin>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-maven-plugin</artifactId>**  **</plugin>**  **</plugins>**  **</build>**  **<repositories>**  **<repository>**  **<id>spring-milestones</id>**  **<name>Spring Milestones</name>**  **<url>https://repo.spring.io/milestone</url>**  **<snapshots>**  **<enabled>false</enabled>**  **</snapshots>**  **</repository>**  **</repositories>** |

#### 9.1.1 Application类加上@EnableConfigServer

|  |
| --- |
| **@SpringBootApplication**  **@EnableConfigServer**  **public class ConfigServerApplication {**  **public static void main(String[] args) {**  **SpringApplication.*run*(ConfigServerApplication.class, args);**  **}**  **}** |

#### 9.1.2 application.properties文件配置以下

|  |
| --- |
| **spring.application.name=config-server**  **server.port=8888**  **spring.cloud.config.server.git.uri=https://gitee.com/itmayi/itmayiedu2.git**  **spring.cloud.config.server.git.searchPaths=/itmayi/itmayiedu2.git**  **spring.cloud.config.label=master**  **spring.cloud.config.server.git.username=**  **spring.cloud.config.server.git.password=** |

* spring.cloud.config.server.git.uri：配置git仓库地址
* spring.cloud.config.server.git.searchPaths：配置仓库路径
* spring.cloud.config.label：配置仓库的分支
* spring.cloud.config.server.git.username：访问git仓库的用户名
* spring.cloud.config.server.git.password：访问git仓库的用户密码

如果Git仓库为公开仓库，可以不填写用户名和密码，如果是私有仓库需要填写，本例子是公开仓库，放心使用。

启动程序：访问<http://localhost:8888/foo/dev>

{"name":"foo","profiles":["dev"],"label":"master",

"version":"792ffc77c03f4b138d28e89b576900ac5e01a44b","state":null,"propertySources":[]}

### 9.2 构建config-client

重新创建一个springboot项目，取名为config-client,其pom文件：

|  |
| --- |
| **<parent>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-parent</artifactId>**  **<version>1.5.2.RELEASE</version>**  **<relativePath /> <!-- lookup parent from repository -->**  **</parent>**  **<properties>**  **<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>**  **<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>**  **<java.version>1.8</java.version>**  **</properties>**  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-config</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-web</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-starter-test</artifactId>**  **<scope>test</scope>**  **</dependency>**  **</dependencies>**  **<dependencyManagement>**  **<dependencies>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-dependencies</artifactId>**  **<version>Dalston.RC1</version>**  **<type>pom</type>**  **<scope>import</scope>**  **</dependency>**  **</dependencies>**  **</dependencyManagement>**  **<build>**  **<plugins>**  **<plugin>**  **<groupId>org.springframework.boot</groupId>**  **<artifactId>spring-boot-maven-plugin</artifactId>**  **</plugin>**  **</plugins>**  **</build>**  **<repositories>**  **<repository>**  **<id>spring-milestones</id>**  **<name>Spring Milestones</name>**  **<url>https://repo.spring.io/milestone</url>**  **<snapshots>**  **<enabled>false</enabled>**  **</snapshots>**  **</repository>**  **</repositories>** |

#### 9.2.1其配置文件**bootstrap.properties**

|  |
| --- |
| **spring.application.name=config-client**  **spring.cloud.config.label=master**  **spring.cloud.config.profile=dev**  **spring.cloud.config.uri= http://localhost:8888/**  **server.port=8881** |

* spring.cloud.config.label 指明远程仓库的分支
* spring.cloud.config.profile
  + dev开发环境配置文件
  + test测试环境
  + pro正式环境
* spring.cloud.config.uri= <http://localhost:8888/> 指明配置服务中心的网址

#### 9.2.2写一个API接口“／hi”

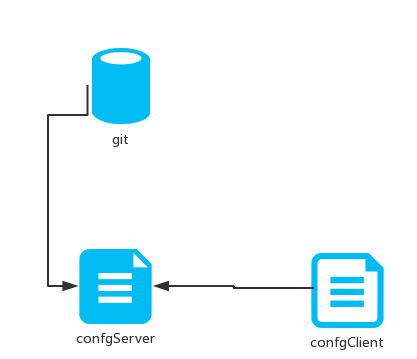
|  |
| --- |
| @SpringBootApplication  @RestController  public class ConfigClientApplication {  public static void main(String[] args) {  SpringApplication.*run*(ConfigClientApplication.class, args);  }  @Value("${foo}")  String foo;  @RequestMapping(value = "/hi")  public String hi() {  return foo;  }  } |

#### 9.2.3网址访问

：<http://localhost:8881/hi>，网页显示：

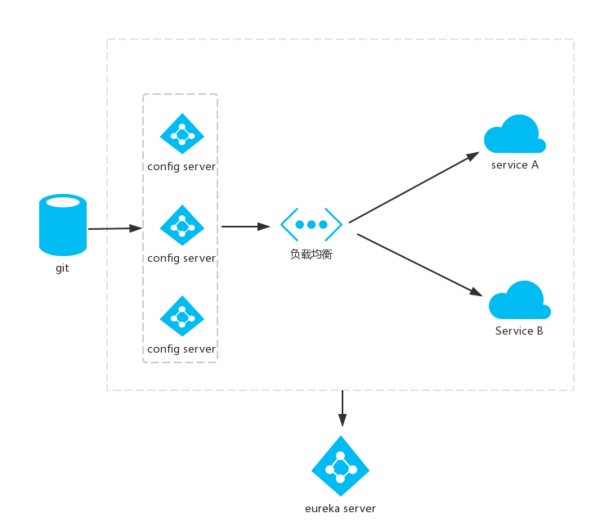
foo version 3

这就说明，config-client从config-server获取了foo的属性，而config-server是从git仓库读取的,如图：



### 9.3 [高可用的分布式配置中心](http://blog.csdn.net/forezp/article/details/70037513)

[高可用的分布式配置中心(Spring Cloud Config)](http://blog.csdn.net/forezp/article/details/70037513) 配置中心如何从远程git读取配置文件，当服务实例很多时，都从配置中心读取文件，这时可以考虑将配置中心做成一个微服务，将其集群化，从而达到高可用，架构图如下：



#### 9.3.1 准备工作

继续使用上一篇文章的工程，创建一个eureka-server工程，用作服务注册中心。

#### 9.3.2 改造config-server

在其pom.xml文件加上EurekaClient的起步依赖spring-cloud-starter-eureka，代码如下: