**SpringCloud**

**Day1**

**学习目标**

1. 能够理解SpringCloud作用
2. 能够使用RestTemplate发送请求
3. 能够搭建Eureka[jʊ'rikə]注册中心

管理服务、监控服务、服务路由

1. 能够使用Ribbon ['rɪbən]负载均衡(消费方)
2. 能够使用Hystrix[hɪst'rɪks]熔断器

服务降级、防止程序雪崩

# 初识Spring Cloud

谈起微服务，它其实是种架构方式。其实现方式很多种：Spring Cloud，Dubbo，华为的Service Combo，Istio 。

那么这么多的微服务架构产品中，我们为什么要用Spring Cloud？因为它后台硬、技术强、群众基础好，使用方便；

## 目标

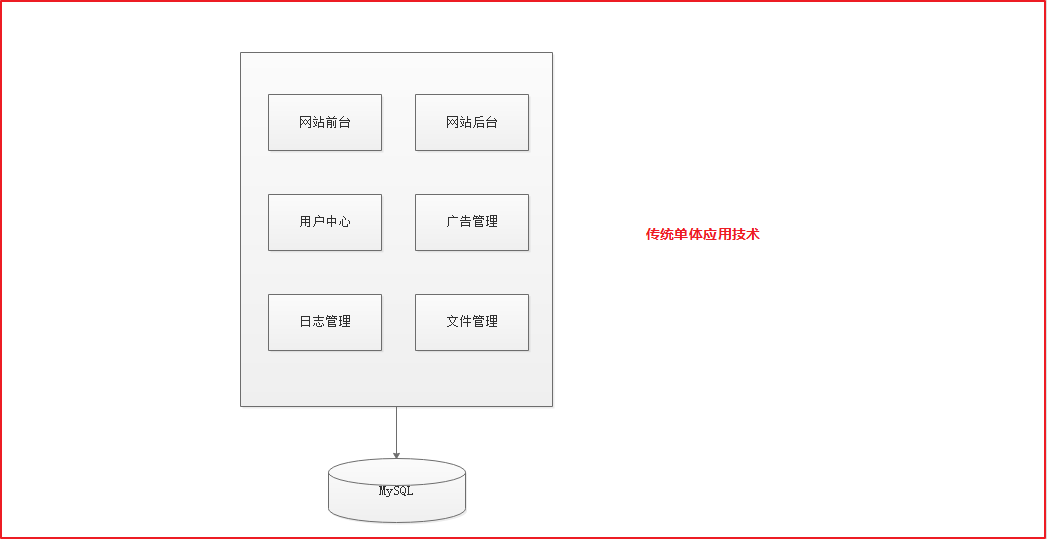
* 了解微服务架构
* 了解SpringCloud技术

## 讲解

### 技术架构演变

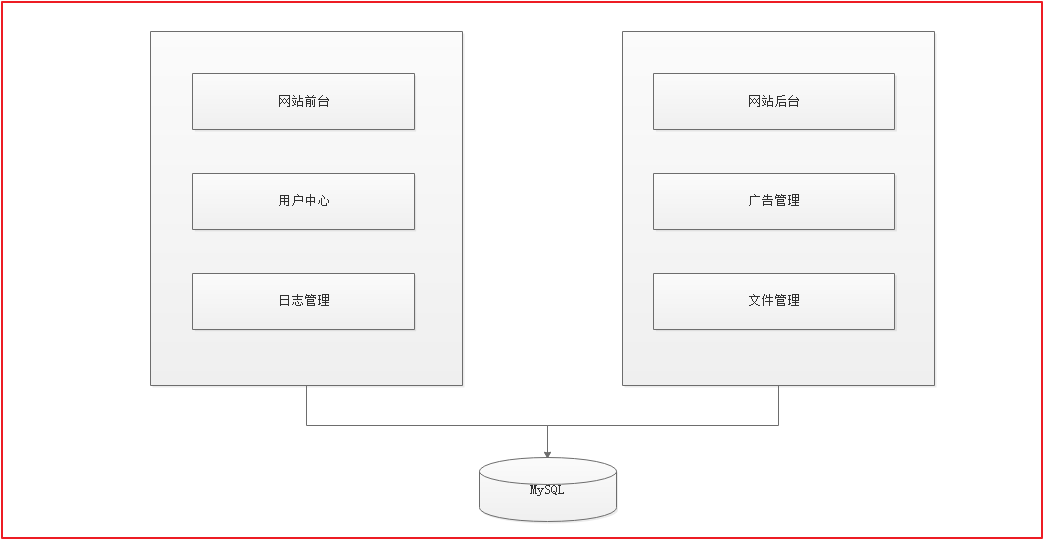
#### 单一应用架构(集中式)

当网站流量很小时，只需要一个应用，所有功能部署在一起，减少部署节点成本的框架称之为集中式框架。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。



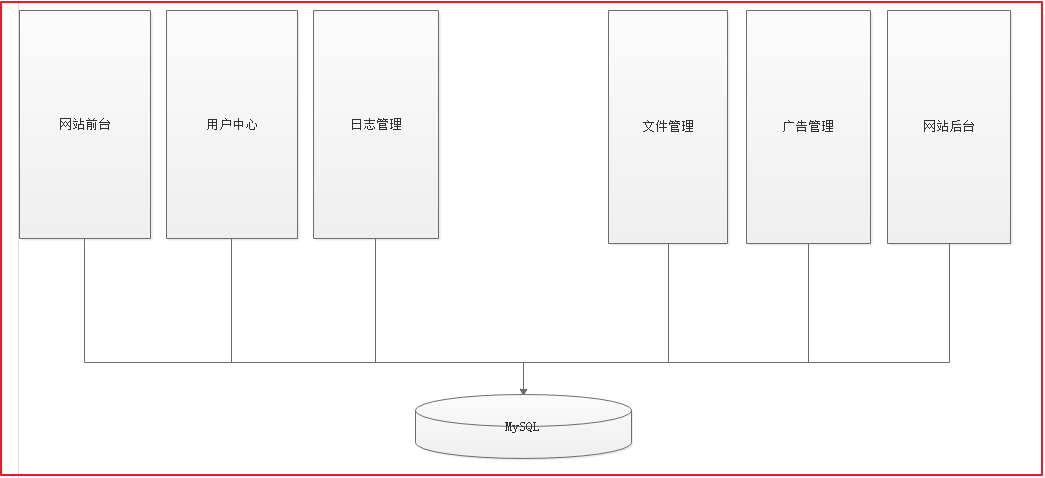
#### 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。



#### 分布式服务架构

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服务框架(RPC)是关键。



#### 面向服务(SOA)架构

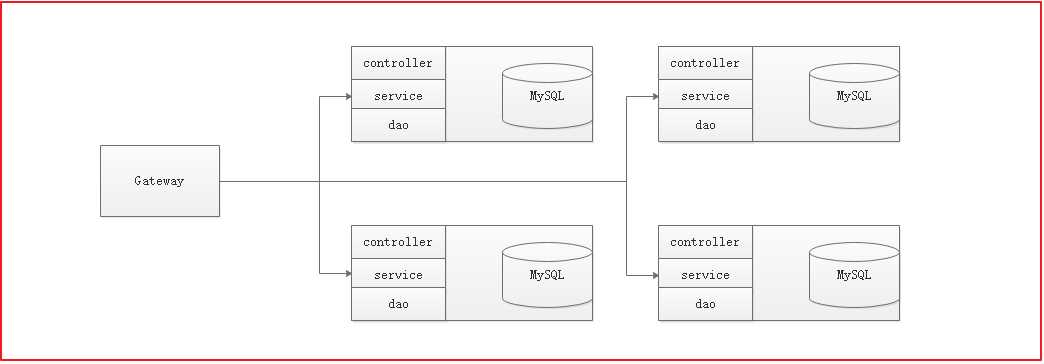
典型代表有两个：流动计算架构和微服务架构；

##### 流动计算架构(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。流动计算架构的最佳实践阿里的Dubbo。

##### 微服务架构

与流动计算架构很相似，除了具备流动计算架构优势外，微服务架构中的微服务可以独立部署，独立发展。且微服务的开发不会限制于任何技术栈。微服务架构的最佳实践是SpringCloud。



### SpringCloud简介

#### SpringCloud介绍

Spring Boot擅长的是集成，把世界上最好的框架集成到自己项目中

Spring Cloud本身也是基于SpringBoot开发而来，SpringCloud是一系列框架的有序集合,也是把非常流行的微服务的技术整合到一起。

Spring Cloud包含了：

注册中心：Eureka、consul、Zookeeper

负载均衡：Ribbon

熔断器：Hystrix

服务通信：Feign

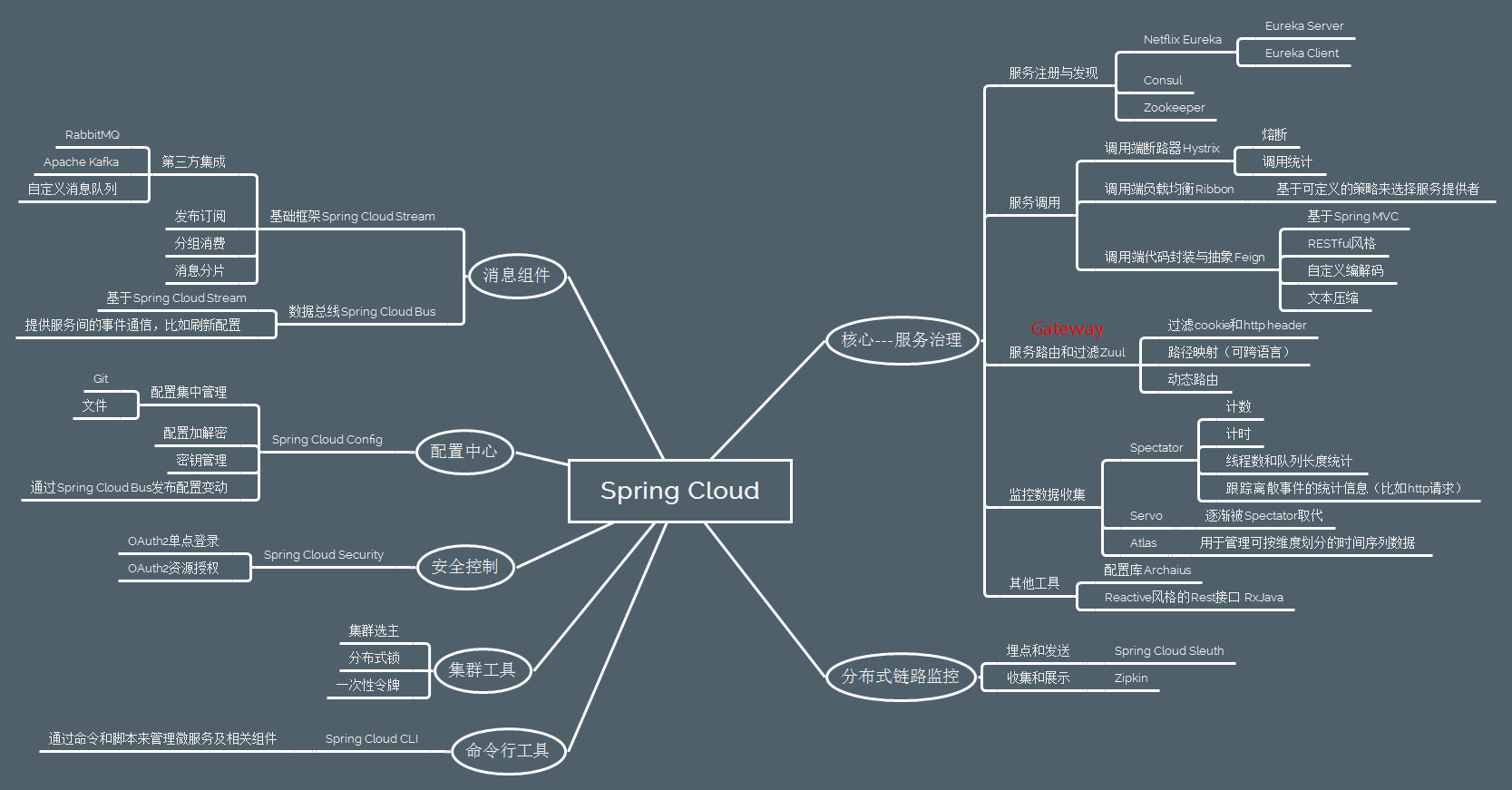
网关：Gateway

配置中心 ：config

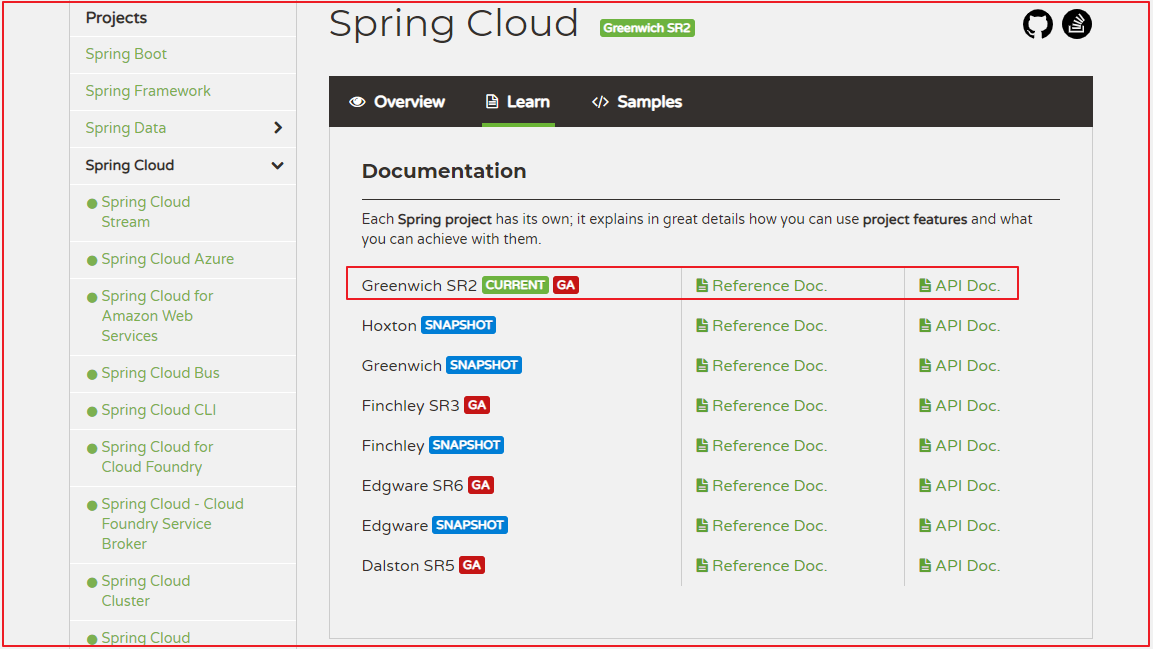
消息总线：Bus

集群状态等等….功能。

Spring Cloud协调分布式环境中各个微服务，为各类服务提供支持。



#### Spring Cloud的版本



版本说明：

SpringCloud是一系列框架组合，为了避免与框架版本产生混淆，采用新的版本命名方式，形式为大版本名+子版本名称

大版本名用伦敦地铁站名

子版本名称三种

SNAPSHOT：快照版本，尝鲜版，随时可能修改

M版本，MileStone，M1表示第一个里程碑版本，一般同时标注PRE，表示预览版

SR，Service Release，SR1表示第一个正式版本，同时标注GA(Generally Available)，稳定版

#### SpringCloud与SpringBoot版本匹配关系

|  |  |
| --- | --- |
| **SpringBoot** | **SpringCloud** |
| 1.2.x | Angel版本 |
| 1.3.x | Brixton版本 |
| 1.4.x | Camden版本 |
| 1.5.x | Dalston版本、Edgware |
| 2.0.x | Finchley版本 |
| 2.1.x | Greenwich GA版本 (2019年2月发布) |

鉴于SpringBoot与SpringCloud关系，SpringBoot建议采用2.1.x版本

## 小结

* 微服务架构：就是将相关的功能独立出来，单独创建一个项目，并且连数据库也独立出来，单独创建对应的数据库。
* Spring Cloud本身也是基于SpringBoot开发而来，SpringCloud是一系列框架的有序集合,也是把非常流行的微服务的技术整合到一起。

# 服务调用方式

## 目标

理解RPC和HTTP的区别

能使用RestTemplate发送请求

## RPC和HTTP

常见远程调用方式：

RPC:(Remote Produce Call)远程过程调用

1.基于Socket

2.自定义数据格式

3.速度快，效率高

4.典型应用代表：Dubbo，ElasticSearch集群间互相调用

HTTP：网络传输协议

1.基于TCP/IP

2.规定数据传输格式

3.缺点是消息封装比较臃肿、传输速度比较慢

4.优点是对服务提供和调用方式没有任何技术限定，自由灵活，更符合微服务理念

RPC和HTTP的区别：RPC是根据语言API来定义，而不是根据基于网络的应用来定义。

Http客户端工具

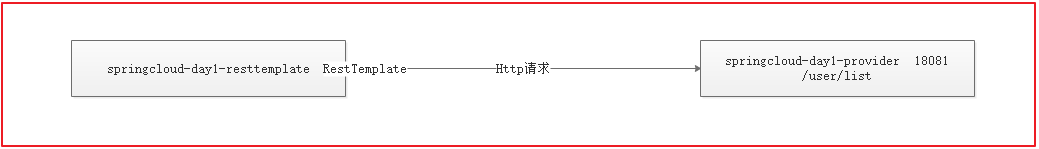
常见Http客户端工具：HttpClient、OKHttp、URLConnection。

## Spring的RestTemplate

### RestTemplate介绍

* RestTemplate是Rest的HTTP客户端模板工具类
* 对基于Http的客户端进行封装
* 实现对象与JSON的序列化与反序列化
* 不限定客户端类型，目前常用的3种客户端都支持：HttpClient、OKHttp、JDK原生URLConnection(默认方式)

### RestTemplate入门案例



我们可以使用RestTemplate实现上图中的请求，springcloud\_day1\_resttemplate通过发送请求，请求springcloud\_day1\_provider的/user/list方法。

#### 搭建springcloud\_day1\_provider

这里不演示详细过程了，大家直接使用IDEA搭建一个普通的SpringBoot工程即可，我们需要web启动器。

*<?***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 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>springcloud\_day1\_provider</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 <**name**>springcloud\_day1\_provider</**name**>  
 <**description**>Demo project for Spring Boot</**description**>  
  
 <**properties**>  
 <**java.version**>1.8</**java.version**>  
 </**properties**>  
  
 <**dependencies**>  
 <**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**>  
  
</**project**>

#### 创建com.itheima.domain.User

**public class** User **implements** Serializable {  
 **private** String **name**;  
 **private** String **address**;  
 **private** Integer **age**;  
 **public** User() {  
 }  
 **public** User(String name, String address, Integer age) {  
 **this**.**name** = name;  
 **this**.**address** = address;  
 **this**.**age** = age;  
 }  
 *//..get set toString 略*}

#### application.properties

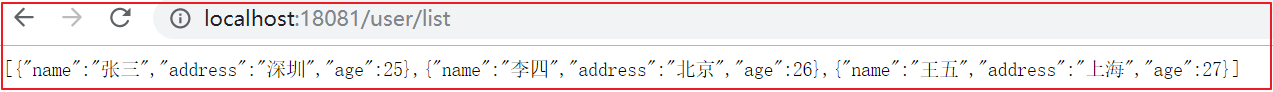
**server.port**=**18081**

#### 创建com.itheima.controller.UserController

@RestController  
@RequestMapping(value = **"user"**)  
**public class** UserController {  
  
 */\*\*\*  
 \* 获取用户列表  
 \** ***@return*** *\*/* @RequestMapping(value = **"list"**)  
 **public** List<User> list(){  
 List<User> users = **new** ArrayList<User>();  
 users.add(**new** User(**"张三"**, **"深圳"**, 25));  
 users.add(**new** User(**"李四"**, **"北京"**, 26));  
 users.add(**new** User(**"王五"**, **"上海"**, 27));  
 **return** users;  
 }  
}

#### 启动与测试

启动引导类，浏览器访问：<http://localhost:18081/user/list>



### 创建springcloud\_day1\_resttemplate

创建的详细过程也不讲解了，直接使用IDEA创建一个SpringBoot工程即可，我们也需要web启动器。

*<?***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 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 <**relativePath**/> *<!-- lookup parent from repository -->* </**parent**>  
 <**groupId**>com.itheima</**groupId**>  
 <**artifactId**>springcloud\_day1\_resttemplate</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 <**name**>springcloud\_day1\_resttemplate</**name**>  
 <**description**>Demo project for Spring Boot</**description**>  
  
 <**properties**>  
 <**java.version**>1.8</**java.version**>  
 </**properties**>  
  
 <**dependencies**>  
 <**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**>  
  
</**project**>

#### 在启动引导类中创建RestTemplate对象

@SpringBootApplication  
**public class** SpringcloudDay1ResttemplateApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(SpringcloudDay1ResttemplateApplication.**class**, args);  
 }

@Bean  
 **public** RestTemplate getRestTemplate(){  
 **return new** RestTemplate();  
 }  
}

#### 测试

在测试类SpringcloudDay1ResttemplateApplication中@Autowired注入RestTemplate

通过RestTemplate的getForObject()方法，传递url地址及实体类的字节码

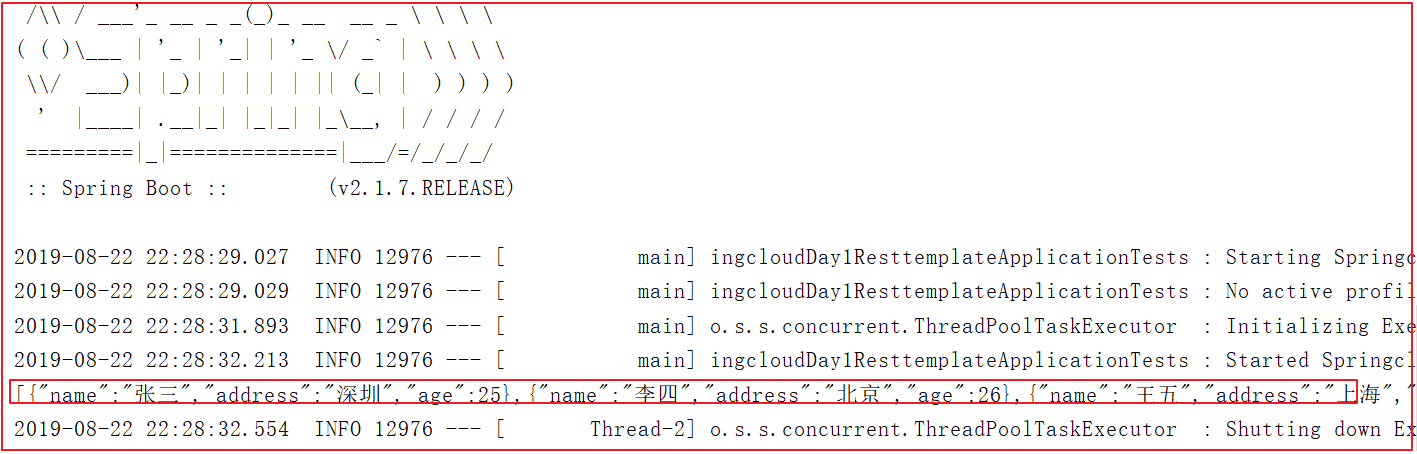
RestTemplate会自动发起请求，接收响应

并且帮我们对响应结果进行反序列化

代码如下：

@RunWith(SpringRunner.**class**)  
@SpringBootTest  
**public class** SpringcloudDay1ResttemplateApplicationTests {  
 @Autowired  
 **private** RestTemplate **restTemplate**;  
  
 @Test  
 **public void** testRestTemplateGet() {  
 String url = **"http://localhost:18081/user/list"**;  
 *//getForObject(请求地址，结果解析类型字节码)* String json = **restTemplate**.getForObject(url, String.**class**);  
 System.***out***.println(json);  
 }  
}

运行测试方法，效果如下：



## 小结

RPC和HTTP的区别：RPC是根据语言API来定义，而不是根据基于网络的应用来定义。

RestTemplate:

①RestTemplate是Rest的HTTP客户端模板工具类。

②对基于Http的客户端进行封装。

③实现对象与JSON的序列化与反序列化。

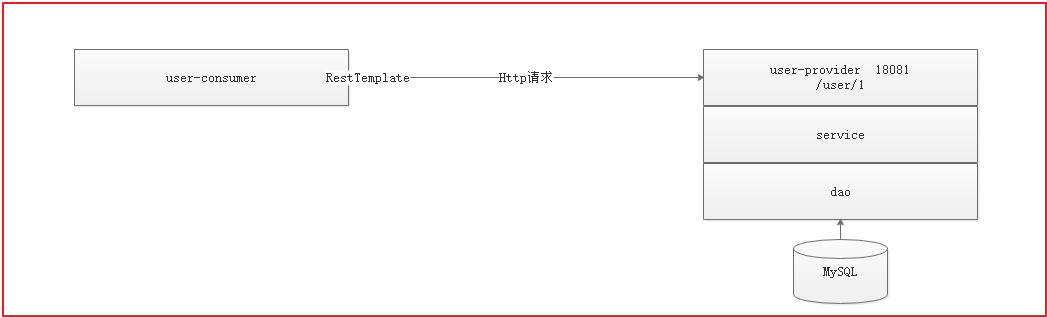
④不限定客户端类型

# 模拟微服务业务场景

模拟开发过程中的服务间关系。抽象出来，开发中的微服务之间的关系是生产者和消费者关系。

总目标：**模拟一个最简单的服务调用场景，场景中保护微服务提供者(Producer)和微服务调用者(Consumer)**，方便后面学习微服务架构。

**注意：实际开发中，每个微服务为一个独立的SpringBoot工程。**



## 目标

* 创建父工程
* 搭建服务提供者
* 搭建服务消费者
* 服务消费者使用RestTemplate调用服务提供者

### 创建父工程

新建一个普通Maven父工程 springcloud\_parent 依赖如下：

*<?***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.itheima</**groupId**>  
 <**artifactId**>springcloud\_parent</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
  
 *<!--父工程-->* <**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.1.7.RELEASE</**version**>  
 </**parent**>  
 *<!--SpringCloud包依赖管理-->* <**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>Greenwich.SR1</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
 </**dependencyManagement**>  
</**project**>

### 创建服务提供者(producer)工程

每个微服务工程都是独立的工程，连数据库都是独立的，所以我们一会要单独为该服务工程创建数据库。

工程创建步骤：

1.准备表结构

2.创建工程

3.引入依赖

4.创建Pojo，需要配置JPA的注解

5.创建Dao，需要继承JpaRepository<T,ID>

6.创建Service，并调用Dao

7.创建Controller，并调用Service

8.创建application.yml文件

9.创建启动类

10.测试

#### 建表

producer工程是一个独立的微服务，一般拥有独立的controller、service、dao、数据库，我们在springcloud数据库新建表结构信息，如下：

-- 使用springcloud数据库

**USE** springcloud;

-- ----------------------------

-- Table structure for tb\_user

-- ----------------------------

**CREATE** **TABLE** `tb\_user` (

`id` int(11) **NOT** NULL AUTO\_INCREMENT,

`username` varchar(100) **DEFAULT** NULL COMMENT '用户名',

`password` varchar(100) **DEFAULT** NULL COMMENT '密码',

`name` varchar(100) **DEFAULT** NULL COMMENT '姓名',

`age` int(11) **DEFAULT** NULL COMMENT '年龄',

`sex` int(11) **DEFAULT** NULL COMMENT '性别，1男，2女',

`birthday` date **DEFAULT** NULL COMMENT '出生日期',

`created` date **DEFAULT** NULL COMMENT '创建时间',

`updated` date **DEFAULT** NULL COMMENT '更新时间',

`note` varchar(1000) **DEFAULT** NULL COMMENT '备注',

**PRIMARY** **KEY** (`id`)

) **ENGINE**=**InnoDB** AUTO\_INCREMENT=2 **DEFAULT** **CHARSET**=utf8 COMMENT='用户信息表';

-- ----------------------------

-- Records of tb\_user

-- ----------------------------

**INSERT** **INTO** `tb\_user` **VALUES** ('1', 'zhangsan', '123456', '张三', '13', '1', '2006-08-01', '2019-05-16', '2019-05-16', '张三');

**INSERT** **INTO** `tb\_user` **VALUES** ('2', 'lisi', '123456', '李四', '13', '1', '2006-08-01', '2019-05-16', '2019-05-16', '李四');

#### 新建user\_provider工程

选中springcloud-parent工程->New Modul->Maven，依赖如下

*<?***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"**>  
 <**parent**>  
 <**artifactId**>springcloud\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>user\_provider</**artifactId**>  
  
 *<!--依赖包-->* <**dependencies**>  
 *<!--JPA包-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
 </**dependency**>  
 *<!--web起步包-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 *<!--MySQL驱动包-->* <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**scope**>runtime</**scope**>  
 </**dependency**>  
 *<!--测试包-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### 创建com.itheima.domain.User

@Entity  
@Table(name = **"tb\_user"**)  
**public class** User {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private** Integer **id**;*//主键id* **private** String **username**;*//用户名* **private** String **password**;*//密码* **private** String **name**;*//姓名* **private** Integer **age**;*//年龄* **private** Integer **sex**;*//性别 1男性，2女性* **private** Date **birthday**; *//出生日期* **private** Date **created**; *//创建时间* **private** Date **updated**; *//更新时间* **private** String **note**;*//备注  
 //..set get toString 略*}

#### 创建com.itheima.dao.UserDao

**public interface** UserDao **extends** JpaRepository<User,Integer> {  
}

#### 创建com.itheima.service.UserService接口

**public interface** UserService {  
 */\*\*\*  
 \* 根据ID查询用户信息  
 \** ***@param id*** *\** ***@return*** *\*/* **public** User findByUserId(Integer id);  
}

#### 创建com.itheima.service.impl.UserServiceImp

@Service  
**public class** UserServiceImpl **implements** UserService {  
  
 @Autowired  
 **private** UserDao **userDao**;  
   
 @Override  
 **public** User findByUserId(Integer id) {  
 **return userDao**.findById(id).get();  
 }  
}

#### 创建com.itheima.controller.UserController

@RestController  
@RequestMapping(value = **"user"**)  
**public class** UserController {  
  
 @Autowired  
 **private** UserService **userService**;  
  
 */\*\*\*  
 \* 根据ID查询用户信息  
 \** ***@param id*** *\** ***@return*** *\*/* @RequestMapping(value = **"find/{id}"**)  
 **public** User findById(@PathVariable(value = **"id"**) Integer id){  
 **return userService**.findByUserId(id);  
 }  
}

#### 创建application.yml配置

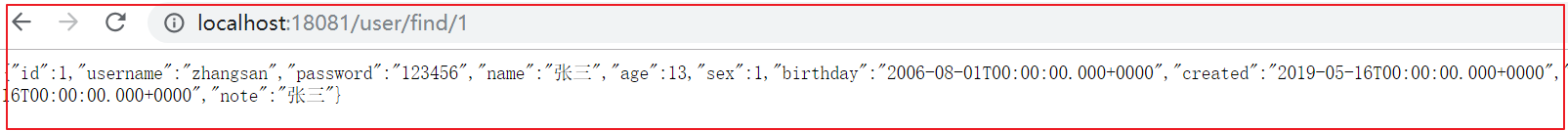
**server**:  
 **port**: 18081  
**spring**:  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **username**: root  
 **password**: root  
 **url**: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC

#### 启动与测试

创建com.itheima.UserProviderApplication启动引导类，并启动

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

访问：<http://localhost:18081/user/find/1>



### 创建服务消费者(consumer)工程

在该工程中使用RestTemplate来调用user\_consumer微服务。

实现步骤：

1.创建工程

2.引入依赖

3.创建Pojo

4.创建启动类，同时创建RestTemplate对象，并交给SpringIOC容器管理

5.创建application.yml文件，指定端口

6.编写Controller，在Controller中通过RestTemplate调用user-provider的服务

7.启动测试

#### 创建工程 user\_consumer

选中springcloud-parent工程->New Modul->Maven，依赖如下：

*<?***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"**>  
 <**parent**>  
 <**artifactId**>springcloud\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>user\_consumer</**artifactId**>  
  
 *<!--依赖包-->* <**dependencies**>  
 *<!--web起步依赖-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### 创建com.itheima.domain.User

可以复制user\_provider工程中的实体类User过来，注意要去掉JAP注解，代码略…..

#### 创建com.itheima.UserConsumerApplication

@SpringBootApplication  
**public class** UserConsumerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(UserConsumerApplication.**class**,args);  
 }  
  
 */\*\*\*  
 \* 将RestTemplate的实例放到Spring容器中  
 \** ***@return*** *\*/* @Bean  
 **public** RestTemplate restTemplate(){  
 **return new** RestTemplate();  
 }  
}

#### 创建application.yml

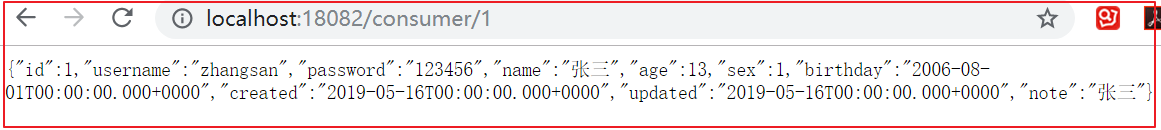
**server**:  
 **port**: 18082

#### 创建com.itheima.controller.UserController

@RestController  
@RequestMapping(value = **"consumer"**)  
**public class** UserController {  
  
 @Autowired  
 **private** RestTemplate **restTemplate**;  
  
 */\*\*\*\*  
 \* 在user-consumer服务中通过RestTemplate调用user-provider服务  
 \** ***@param id*** *\** ***@return*** *\*/* @GetMapping(value = **"/{id}"**)  
 **public** User queryById(@PathVariable(value = **"id"**)Integer id){  
 String url = **"http://localhost:18081/user/find/"**+id;  
 **return restTemplate**.getForObject(url,User.**class**);  
 }  
  
}

#### 启动与测试

启动引导类，访问：<http://localhost:18082/consumer/1>



### 思考

user-provider：对外提供用户查询接口

user-consumer：通过RestTemplate访问接口查询用户数据

存在的问题：

1. 在服务消费者中，我们把url地址硬编码到代码中，不方便后期维护
2. 在服务消费者中，不清楚服务提供者的状态(user-provider有可能没有宕机了)
3. 服务提供者只有一个服务，即便服务提供者形成集群，服务消费者还需要自己实现负载均衡
4. 服务提供者的如果出现故障，是否能够及时发现

其实上面说的问题，概括一下就是微服务架构必然要面临的问题

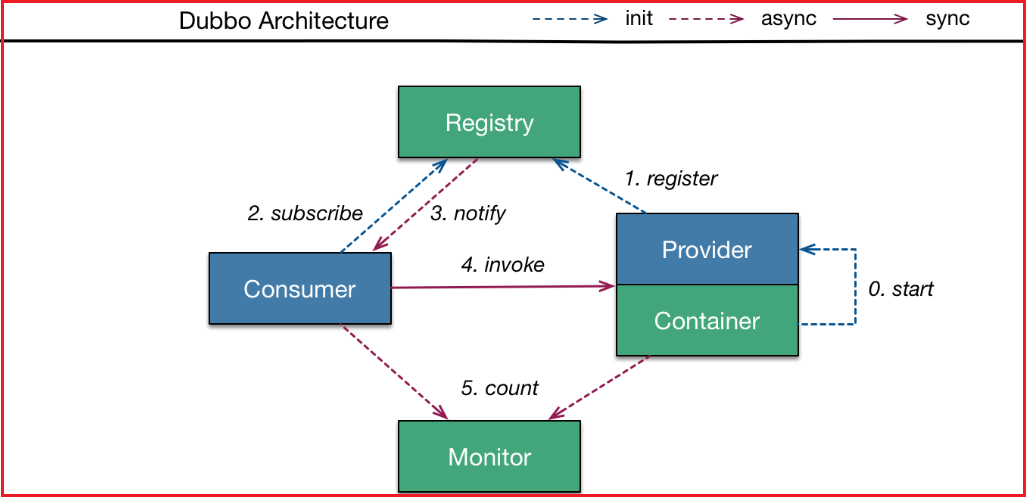
* 服务管理：自动注册与发现、状态监管
* 服务负载均衡
* 熔断器

## 小结

* 服务消费者使用RestTemplate调用服务提供者,使用RestTemplate调用的时候，需要先创建并注入到SpringIOC容器中。
* 在服务消费者中，我们把url地址硬编码到代码中，不方便后期维护。
* 在服务消费者中，不清楚服务提供者的状态(user-provider有可能没有宕机了)。
* 服务提供者只有一个服务，即便服务提供者形成集群，服务消费者还需要自己实现负载均衡
* 服务提供者的如果出现故障，不能及时发现。

# 注册中心 Spring Cloud Eureka

前面我们学过Dubbo，关于Dubbo的执行过程我们看如下图片：



执行过程：

1.Provider:服务提供者,异步将自身信息注册到Register（注册中心）

2.Consumer：服务消费者，异步去Register中拉取服务数据

3.Register异步推送服务数据给Consumer,如果有新的服务注册了，Consumer可以直接监控到新的服务

4.Consumer同步调用Provider

5.Consumer和Provider异步将调用频率信息发给Monitor监控

## 目标

* 理解Eureka的原理图
* 能实现Eureka服务的搭建
* 能实现服务提供者向Eureka注册服务
* 能实现服务消费者向Eureka注册服务
* 能实现消费者通过Eureka访问服务提供者
* 能掌握Eureka的详细配置

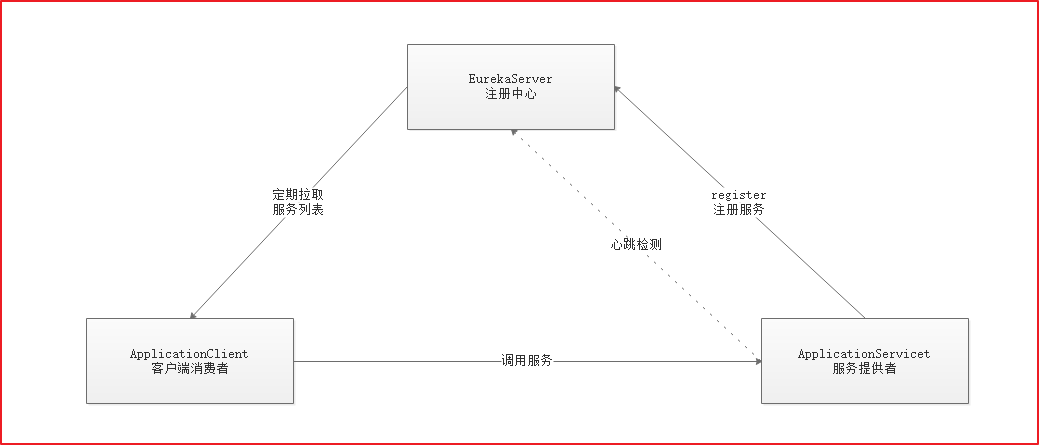
## Eureka 简介

Eureka解决了第一个问题：服务的管理，注册和发现、状态监管、动态路由。

Eureka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，Eureka自动匹配服务给调用者。

Eureka与服务之间通过心跳机制进行监控；

**基本架构图**



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

服务提供者：启动后向Eureka注册自己的信息(地址，提供什么服务)

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

## 入门案例

目标：搭建Eureka Server环境，创建一个eureka\_server工程。

步骤：分三步

1：搭建工程eureka\_server

2：服务提供者-注册服务，user\_provider工程

3：服务消费者-发现服务，user\_consumer工程

### 搭建eureka\_server工程

选中springcloud\_parent工程->New Modul->Maven，依赖如下

*<?***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"**>  
 <**parent**>  
 <**artifactId**>springcloud\_parent</**artifactId**>  
 <**groupId**>com.itheima</**groupId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 </**parent**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**artifactId**>eureka\_server</**artifactId**>  
  
 *<!--依赖包-->* <**dependencies**>  
 *<!--eureka-server依赖-->* <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-server</**artifactId**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

#### application.yml配置

**server**:  
 **port**: 7001 *#端口号***spring**:  
 **application**:  
 **name**: eureka-server *# 应用名称，会在Eureka中作为服务的id标识（serviceId）***eureka**:  
 **client**:  
 **register-with-eureka**: **false** *#是否将自己注册到Eureka中* **fetch-registry**: **false** *#是否从eureka中获取服务信息* **service-url**:  
 **defaultZone**: http://localhost:7001/eureka *# EurekaServer的地址*

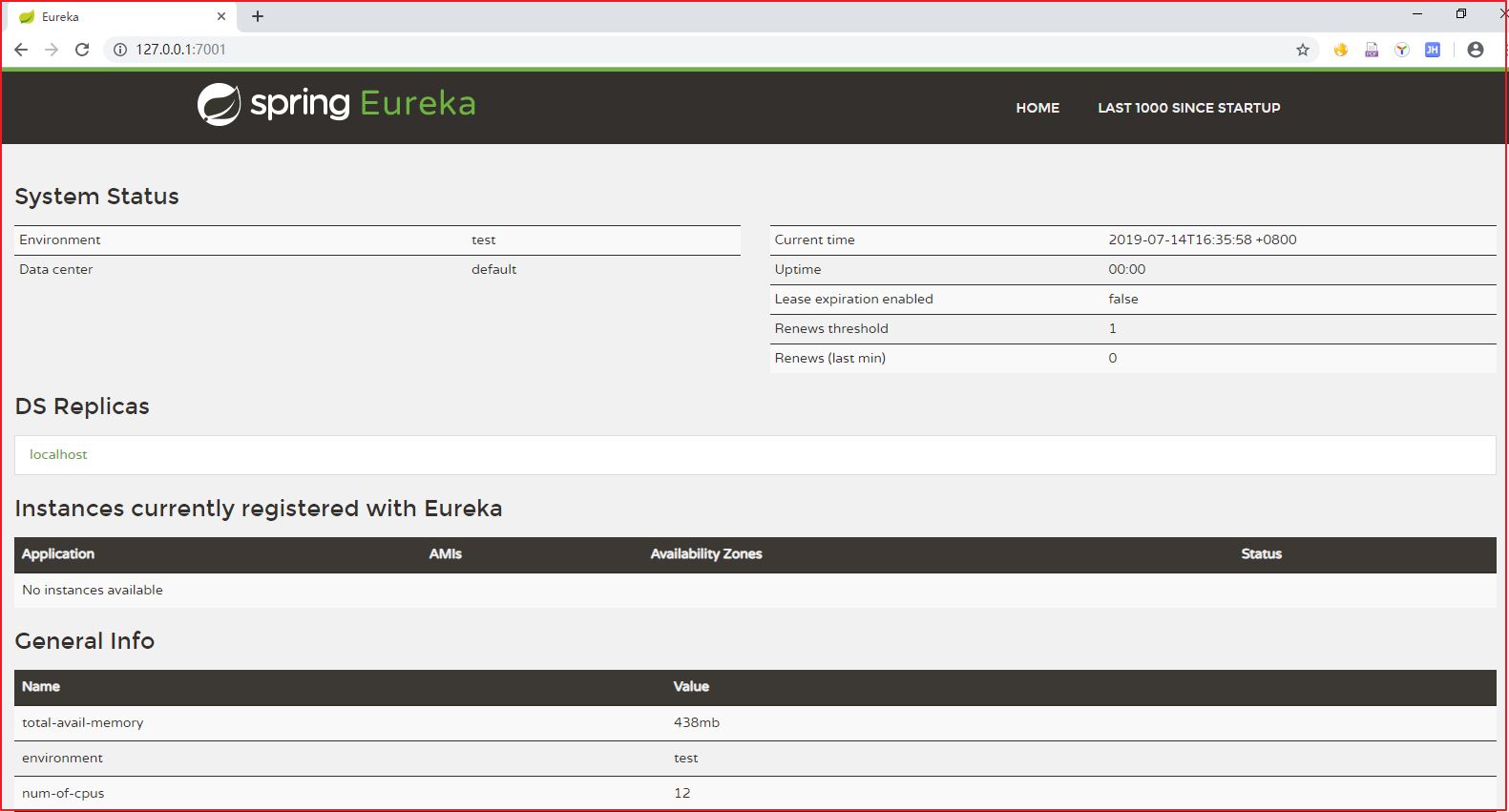
#### 创建启动类

创建com.itheima.EurekaServerApplication,在类上需要添加@SpringBootApplication、@EnableEurekaServer两个注解，用于开启Eureka服务,代码如下：

@SpringBootApplication  
@EnableEurekaServer *//开启Eureka服务***public class** EurekaServerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(EurekaServerApplication.**class**,args);  
 }  
}

#### 启动与测试

启动后，访问<http://127.0.0.1:7001>，效果如下：



### 服务提供者-注册服务

我们的user\_provider属于服务提供者，需要在user-provider工程中引入Eureka客户端依赖，然后在配置文件中指定Eureka服务地址,然后在启动类中开启Eureka服务发现功能。

步骤：

1.引入eureka客户端依赖包

2.在application.yml中配置Eureka服务地址

3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient

#### 引入依赖

在user\_provider的pom.xml中引入如下依赖

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

#### 配置Eureka服务地址

修改user\_provider的application.yml配置文件，添加Eureka服务地址，代码如下：

**server**:  
 **port**: 18081  
**spring**:  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **username**: root  
 **password**: root  
 **url**: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **application**:  
 **name**: user-provider *#服务的名字,不同的应用，名字不同，如果是集群，名字需要相同  
#指定eureka服务地址***eureka**:  
 **client**:  
 **service-url**:  
 *# EurekaServer的地址* **defaultZone**: http://localhost:7001/eureka

#### 开启Eureka客户端发现功能

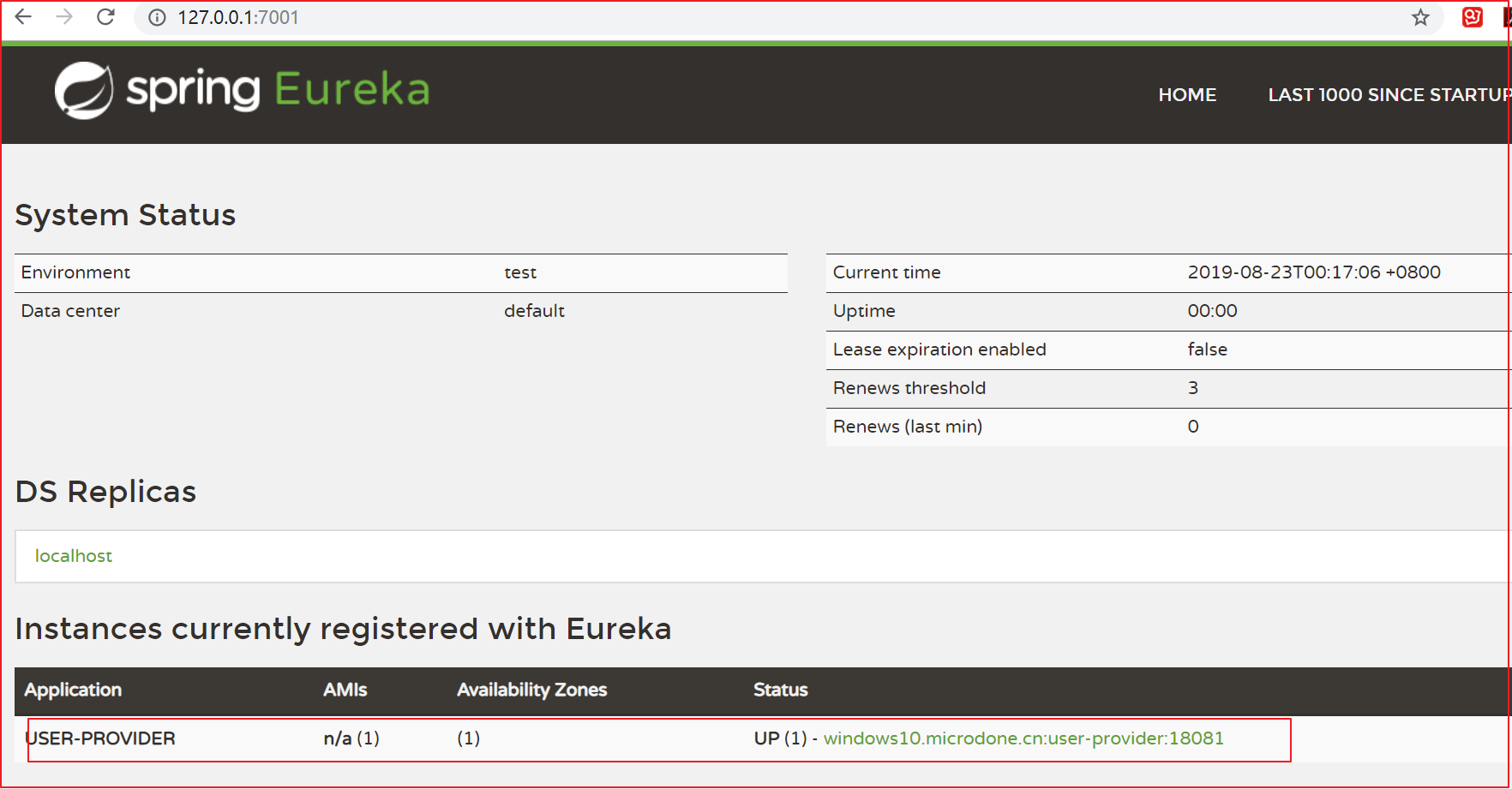
在user\_provider的启动类com.itheima.UserProviderApplication上添加@EnableDiscoveryClient注解或者@EnableEurekaClient，用于开启客户端发现功能。

@SpringBootApplication  
@EnableDiscoveryClient *//开启Eureka客户端发现功能(推荐使用)  
//@EnableEurekaClient //开启Eureka客户端发现功能，注册中心只能是Eureka***public class** UserProviderApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(UserProviderApplication.**class**,args);  
 }  
}

#### 启动测试

启动eureka-server，再启动user-provider。

访问Eureka地址<http://127.0.0.1:7001> ，效果如下：



### 服务消费者-注册服务中心

消费方添加Eureka服务注册和生产方配置流程一致。

步骤：

1.引入eureka客户端依赖包

2.在application.yml中配置Eureka服务地址

3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient

#### pom.xml引入依赖

修改user-consumer的pom.xml引入如下依赖

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

#### application.yml中配置eureka服务地址

修改user-consumer工程的application.yml配置，添加eureka服务地址，配置如下：

**server**:  
 **port**: 18082  
**spring**:  
 **application**:  
 **name**: user-consumer *#服务名字  
#指定eureka服务地址***eureka**:  
 **client**:  
 **service-url**:  
 *# EurekaServer的地址* **defaultZone**: http://localhost:7001/eureka

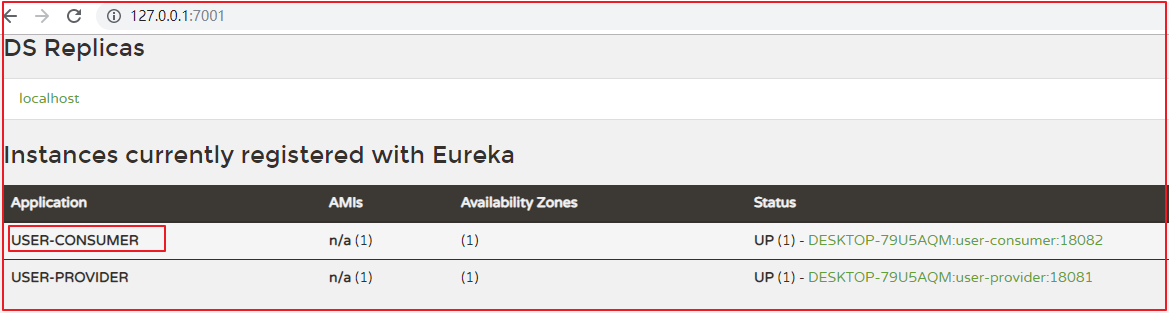
#### 在启动类上开启Eureka服务发现功能

修改user-consumer的com.itheima.UserConsumerApplication启动类，在类上添加@EnableDiscoveryClient注解，代码如下：

@SpringBootApplication  
@EnableDiscoveryClient  
**public class** UserConsumerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(UserConsumerApplication.**class**,args);  
 }  
  
 */\*\*\*  
 \* 将RestTemplate的实例放到Spring容器中  
 \** ***@return*** *\*/* @Bean  
 **public** RestTemplate restTemplate(){  
 **return new** RestTemplate();  
 }  
}

#### 测试

启动user-consumer，然后访问Eureka服务地址 <http://127.0.0.1:7001/> 效果如下：



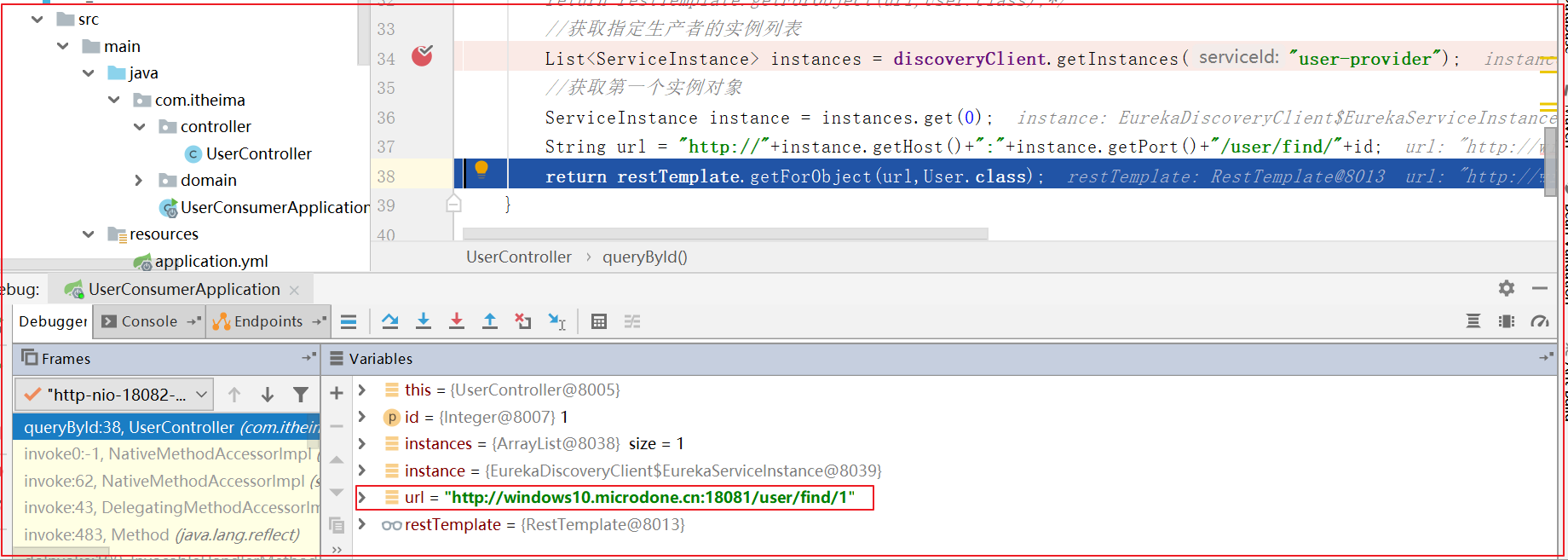
#### 消费者通过Eureka访问提供者

之前消费者user-consumer访问服务提供者user-provider是通过http://localhost:18081/user/find/1访问的，这里是具体的路径，没有从Eureka获取访问地址，我们可以让消费者从Eureka那里获取服务提供者的访问地址，然后访问服务提供者。

修改user-consumer的com.itheima.controller.UserController，代码如下：

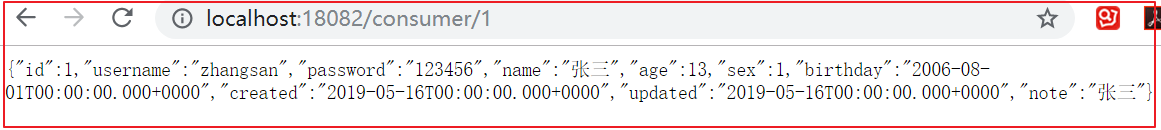
@RestController  
@RequestMapping(value = **"consumer"**)  
**public class** UserController {  
  
 @Autowired  
 **private** RestTemplate **restTemplate**;  
 @Autowired  
 **private** DiscoveryClient **discoveryClient**; *//此对象用于向注册中心获取服务列表  
  
 /\*\*\*\*  
 \* 在user-consumer服务中通过RestTemplate调用user-provider服务  
 \** ***@param id*** *\** ***@return*** *\*/* @GetMapping(value = **"/{id}"**)  
 **public** User queryById(@PathVariable(value = **"id"**)Integer id){  
 */\*String url = "http://localhost:18081/user/find/"+id;  
 return restTemplate.getForObject(url,User.class);\*/  
 //获取指定生产者的实例列表* List<ServiceInstance> instances = **discoveryClient**.getInstances(**"user-provider"**);  
 *//获取第一个实例对象* ServiceInstance instance = instances.get(0);  
 String url = **"http://"**+instance.getHost()+**":"**+instance.getPort()+**"/user/find/"**+id;  
 **return restTemplate**.getForObject(url,User.**class**);  
 }  
  
}

Debug跟踪运行，访问<http://localhost:18082/consumer/1> ，效果如下：



跟踪运行后，我们发现，这里的地址就是服务注册中的状态名字。

浏览器结果如下：

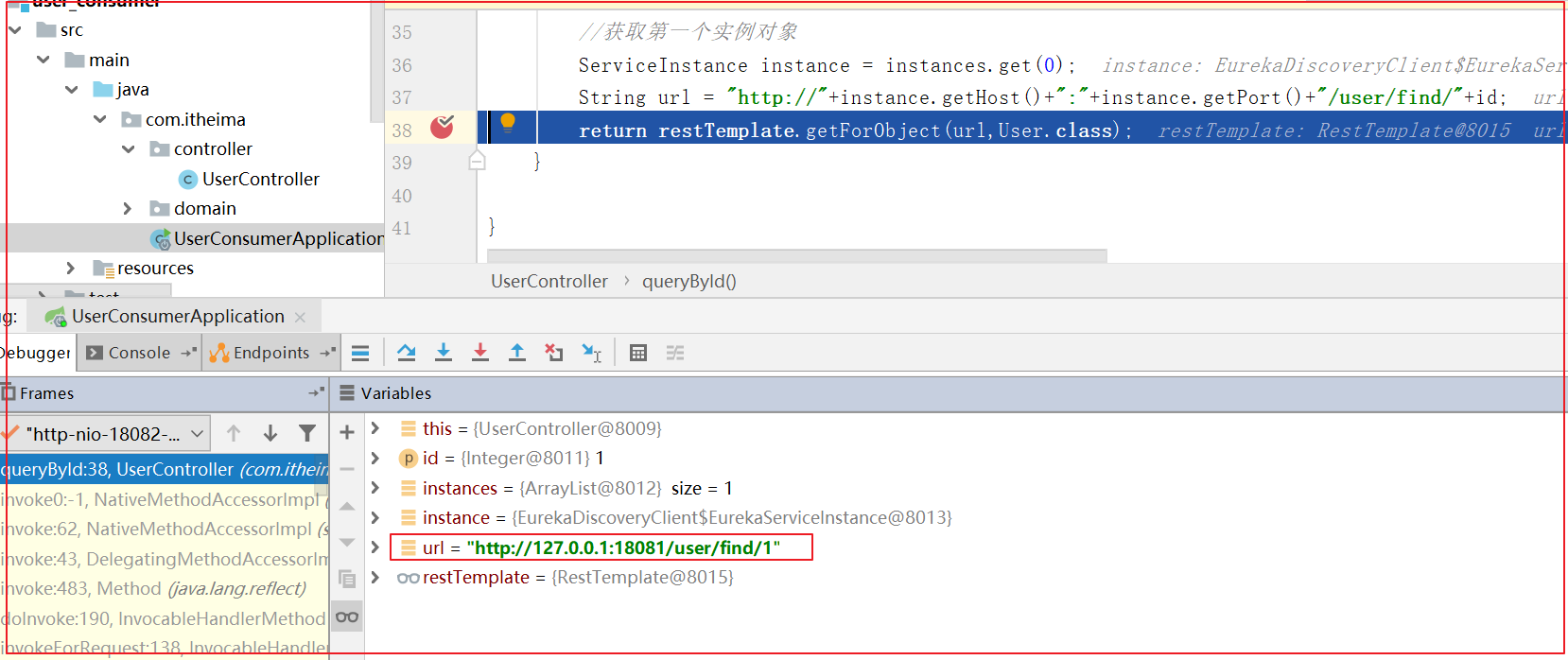


**使用IP访问配置**

上面的请求地址是服务状态名字，其实也是当前主机的名字，可以通过配置文件，将它换成IP，修改application.yml配置文件，代码如下：

**server**:  
 **port**: 18081  
**spring**:  
 **datasource**:  
 **driver-class-name**: com.mysql.cj.jdbc.Driver  
 **username**: root  
 **password**: root  
 **url**: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 **application**:  
 **name**: user-provider *#服务的名字,不同的应用，名字不同，如果是集群，名字需要相同  
#指定eureka服务地址***eureka**:  
 **client**:  
 **service-url**:  
 *# EurekaServer的地址* **defaultZone**: http://localhost:7001/eureka  
 **instance**:  
 *#指定IP地址* **ip-address**: 127.0.0.1  
 *#访问服务的时候，推荐使用IP* **prefer-ip-address**: **true**

重新启动user-provider与user-consumer，并再次测试，测试效果如下：



### Eureka配置详解（了解）

#### 基础架构

Eureka架构中的三个核心角色

1.服务注册中心：Eureka服务端应用，提供服务注册发现功能，eureka-server

2.服务提供者：提供服务的应用

要求统一对外提供Rest风格服务即可

本例子：user-provider

3.服务消费者：从注册中心获取服务列表，知道去哪调用服务方，user-consumer

#### Eureka客户端

服务提供者要向EurekaServer注册服务，并完成服务续约等工作

##### 服务注册

1. 当我们开启了客户端发现注解@DiscoveryClient。同时导入了eureka-client依赖坐标

2. 同时配置Eureka服务注册中心地址在配置文件中

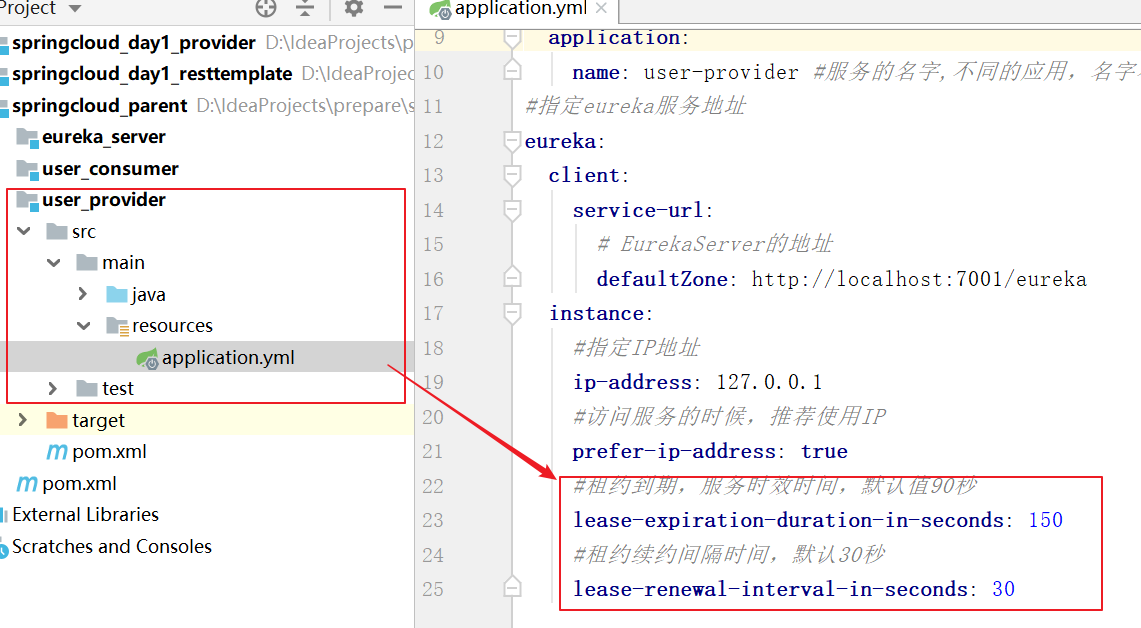
3. 服务在启动时，检测是否有@DiscoveryClient注解和配置信息

4. 如果有，则会向注册中心发起注册请求，携带服务元数据信息(IP、端口等)

5. Eureka注册中心会把服务的信息保存在Map中。

##### 服务续约

服务注册完成以后，服务提供者会维持一个心跳，保存服务处于存在状态。这个称之为服务续约(renew)。



上图配置如下：

*#租约到期，服务时效时间，默认值90秒***lease-expiration-duration-in-seconds**: 150  
*#租约续约间隔时间，默认30秒***lease-renewal-interval-in-seconds**: 30

参数说明：

1.两个参数可以修改服务续约行为

lease-renewal-interval-seconds:90，租约到期时效时间，默认90秒

lease-expiration-duration-in-seconds:30，租约续约间隔时间，默认30秒

2.服务超过90秒没有发生心跳，EurekaServer会将服务从列表移除[前提是EurekaServer关闭了自我保护]

##### 获取服务列表



上图配置如下：

*# 每隔30获取服务列表(只读备份)***registry-fetch-interval-seconds**: 30

说明：

服务消费者启动时，会检测是否获取服务注册信息配置

如果是，则会从 EurekaServer服务列表获取只读备份，缓存到本地

每隔30秒，会重新获取并更新数据

每隔30秒的时间可以通过配置registry-fetch-interval-seconds修改

#### 失效剔除和自我保护

##### 服务下线

当服务正常关闭操作时，会发送服务下线的REST请求给EurekaServer。

服务中心接受到请求后，将该服务置为下线状态

##### 失效剔除

服务中心每隔一段时间(默认60秒)将清单中没有续约的服务剔除。

通过eviction-interval-timer-in-ms配置可以对其进行修改，单位是毫秒

剔除时间配置



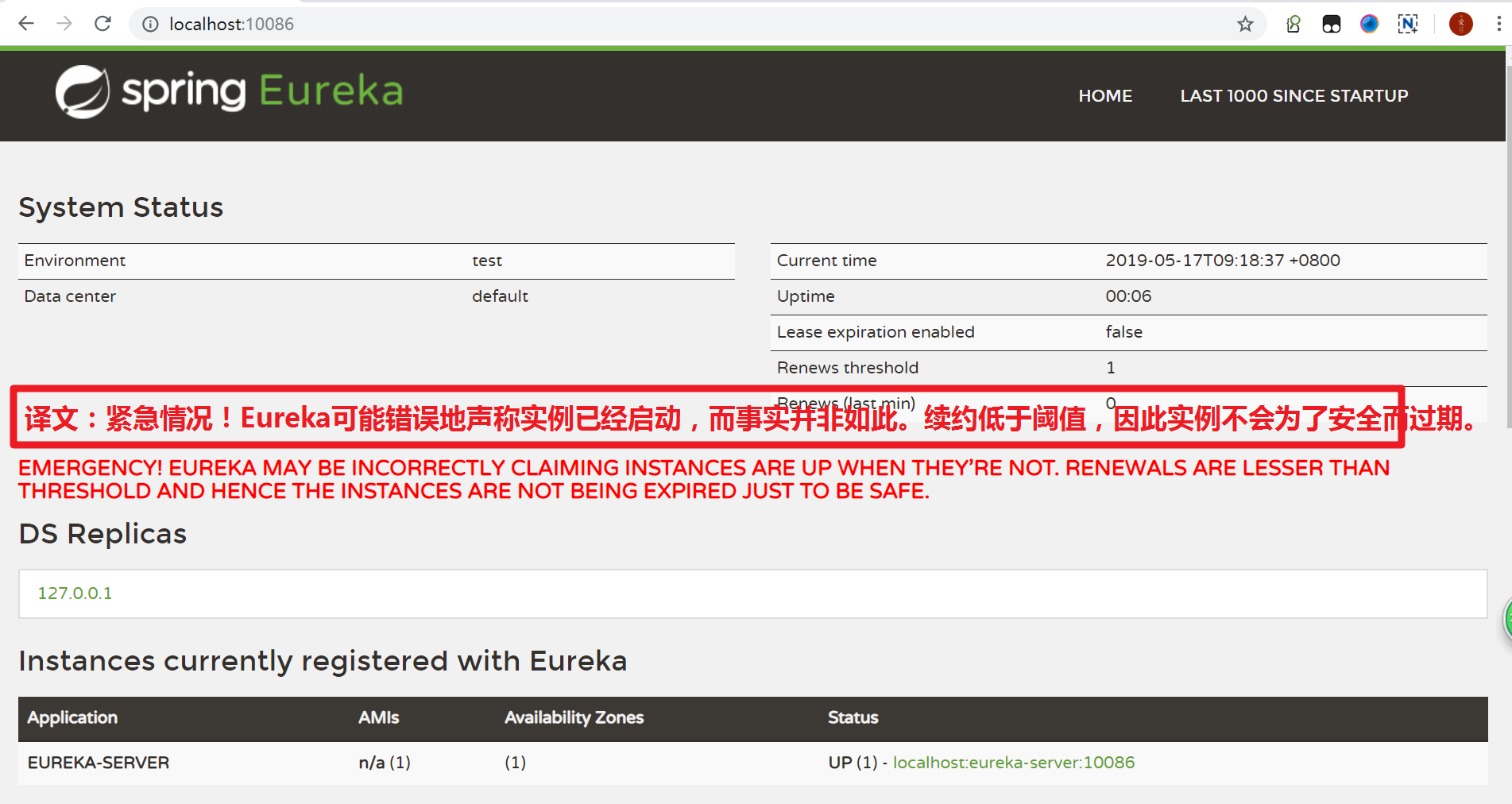
上图代码如下：

**server**:  
 *# 服务中心每隔一段时间(默认60秒)将清单中没有续约的服务剔除，单位是毫秒* **eviction-interval-timer-in-ms**: 5000

##### 自我保护

Eureka会统计服务实例最近15分钟心跳续约的比例是否低于85%，如果低于则会触发自我保护机制。

服务中心页面会显示如下提示信息。



含义：紧急情况！Eureka可能错误地声称实例已经启动，而事实并非如此。续约低于阈值，因此实例不会为了安全而过期。

1.自我保护模式下，不会剔除任何服务实例

2.自我保护模式保证了大多数服务依然可用

3.通过enable-self-preservation配置可用关停自我保护，默认值是打开

关闭自我保护



上图配置如下：

*# 关闭自我保护功能，默认是打开的***enable-self-preservation**: **false**

## 小结

* 理解Eureka的原理图

properties

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

服务提供者：启动后向Eureka注册自己的信息(地址，提供什么服务)

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

* 能实现Eureka服务的搭建:引入依赖包，配置配置文件，在启动类上加@EnableEurekaServer。
* 能实现服务提供者向Eureka注册服务

properties

1.引入eureka客户端依赖包

2.在application.yml中配置Eureka服务地址

3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient

* 能实现服务消费者向Eureka注册服务

properties

1.引入eureka客户端依赖包

2.在application.yml中配置Eureka服务地址

3.在启动类上添加@EnableDiscoveryClient或者@EnableEurekaClient

* 能实现消费者通过Eureka访问服务提供者

# 负载均衡 Spring Cloud Ribbon

Ribbon主要 解决集群服务中，多个服务高效率访问的问题。

## 目标

* 理解Ribbon的负载均衡应用场景
* 能实现Ribbon的轮询、随机算法配置
* 理解源码对负载均衡的切换

## Ribbon 简介

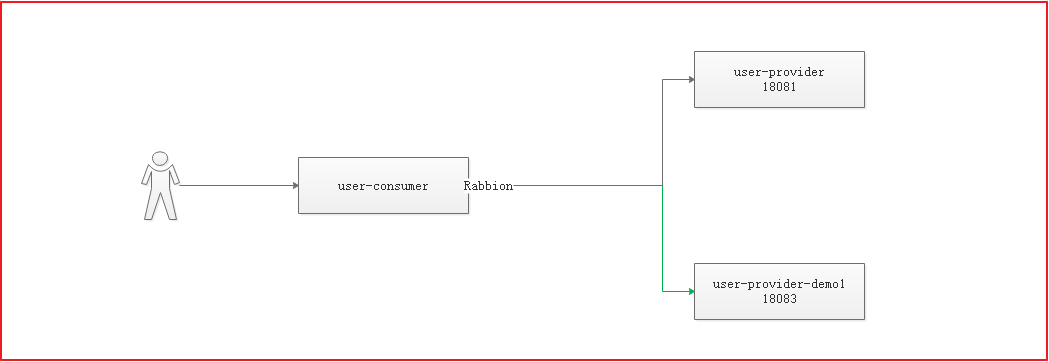
**什么是Ribbon？**

Ribbon是Netflix发布的负载均衡器，有助于控制HTTP客户端行为。为Ribbon配置服务提供者地址列表后，Ribbon就可基于负载均衡算法，自动帮助服务消费者请求。

Ribbon默认提供的负载均衡算法：轮询，随机,重试法,加权。当然，我们可用自己定义负载均衡算法

### 入门案例

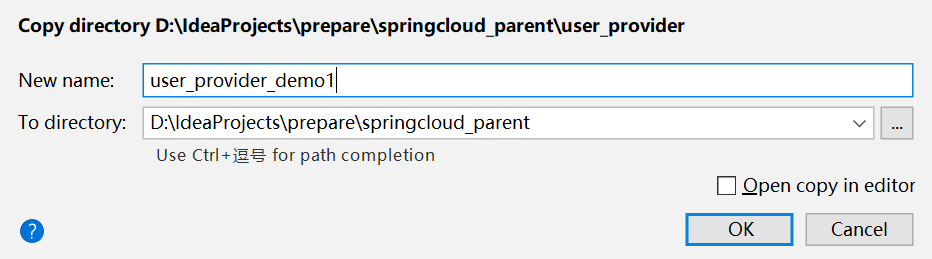
#### 多个服务集群



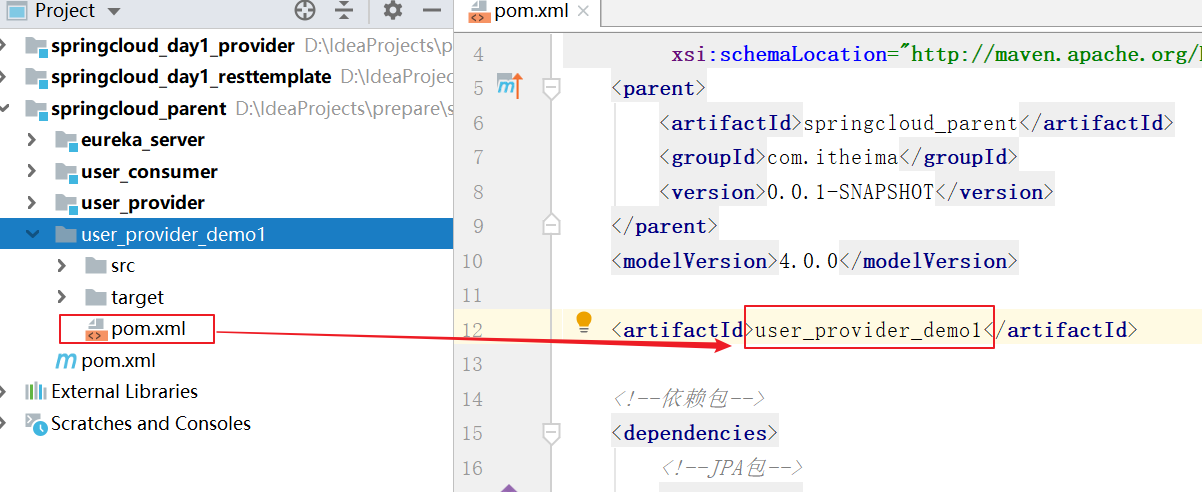
如果想要做负载均衡，我们的服务至少2个以上,为了演示负载均衡案例，我们可以复制2个工程，分别为user-provider和user-provider-demo1，可以按照如下步骤拷贝工程：

##### 选中user\_provider,按Ctrl+C，然后Ctrl+V

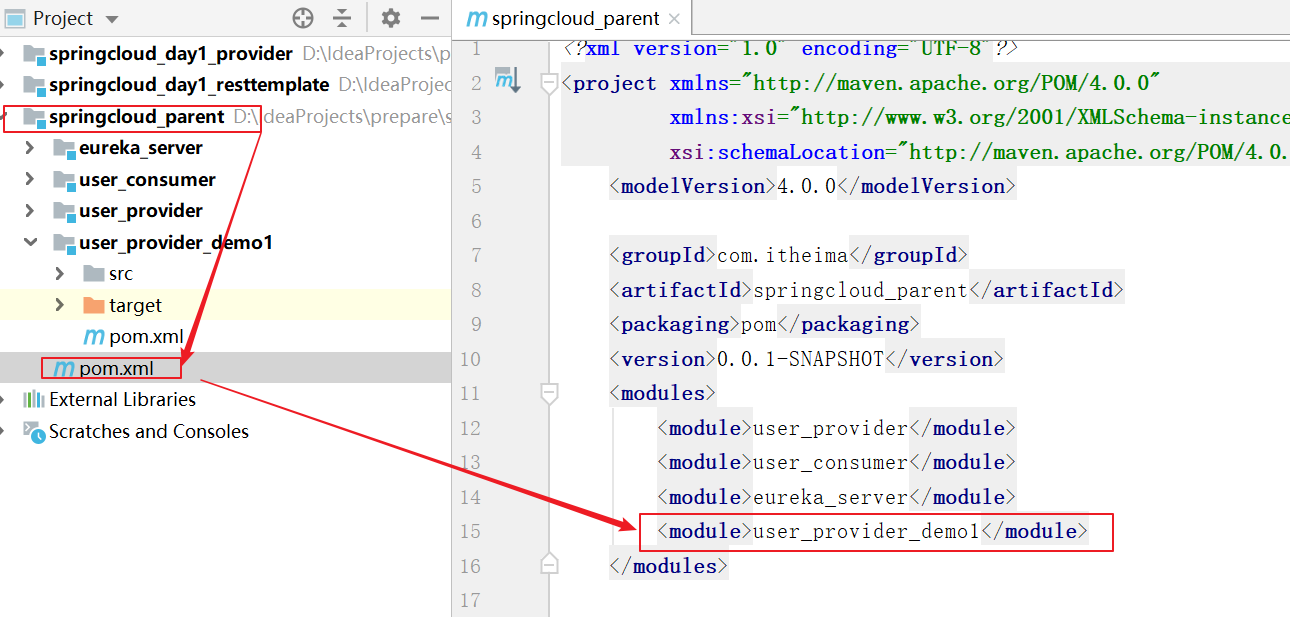
##### 名字改成user\_provider\_demo1,点击OK



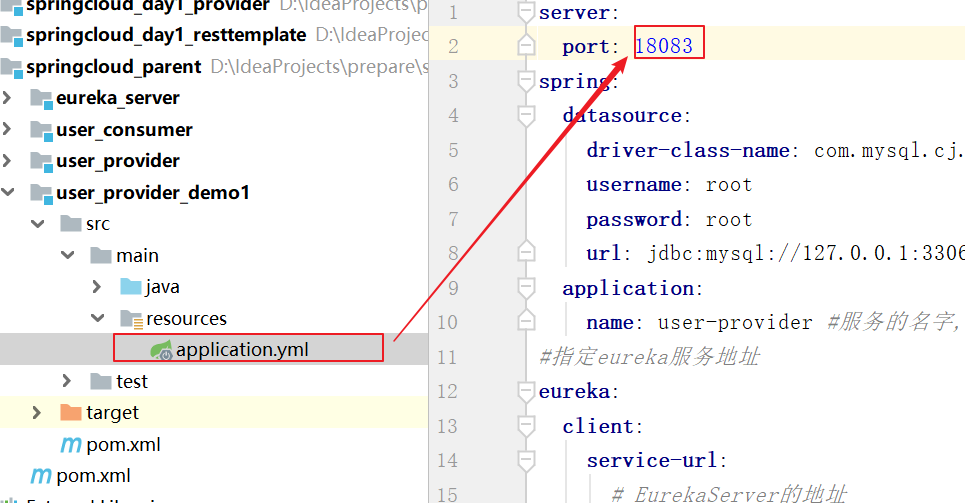
##### 将user-provider-demo1的artifactId换成user-provider-demo1



##### 在springcloud-parent的pom.xml中添加一个<module>user\_provider\_demo1</module>



##### 将user-provider-demo1的application.yml中的端口改成18083



为了方便测试，将2个工程对应的com.itheima.controller.UserController都修改一下：

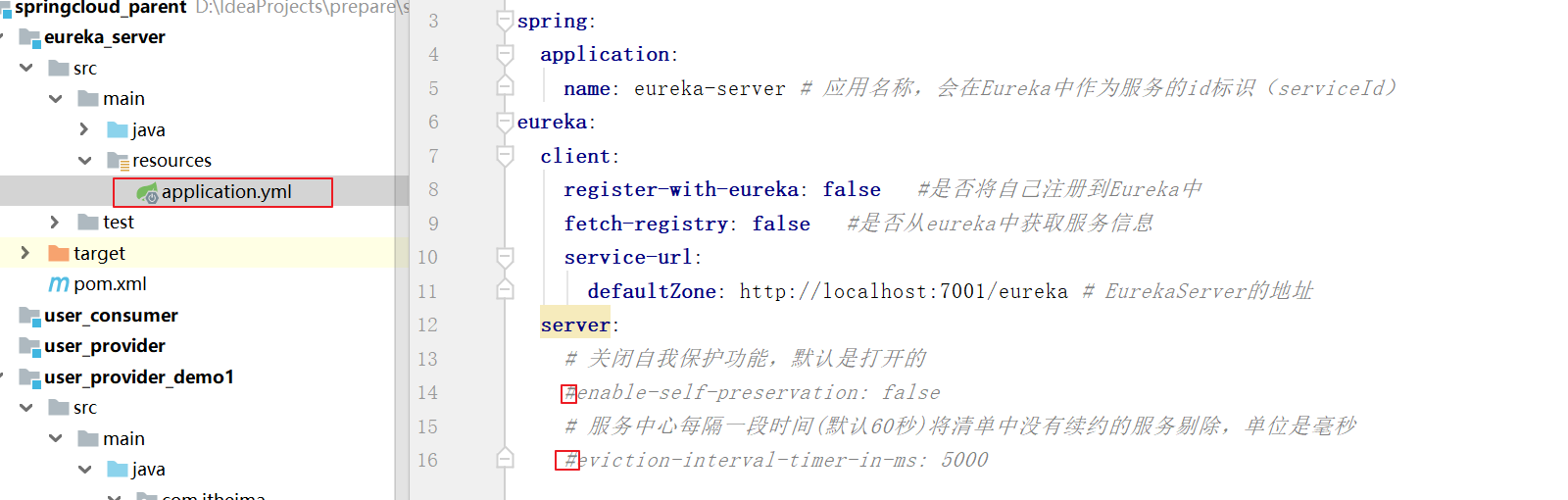
**user-provider**:

@RequestMapping(value = **"find/{id}"**)  
**public** User findById(@PathVariable(value = **"id"**) Integer id){  
 User user = **userService**.findByUserId(id);  
 user.setUsername(user.getUsername()+**" user-provider "**);  
 **return** user;  
}

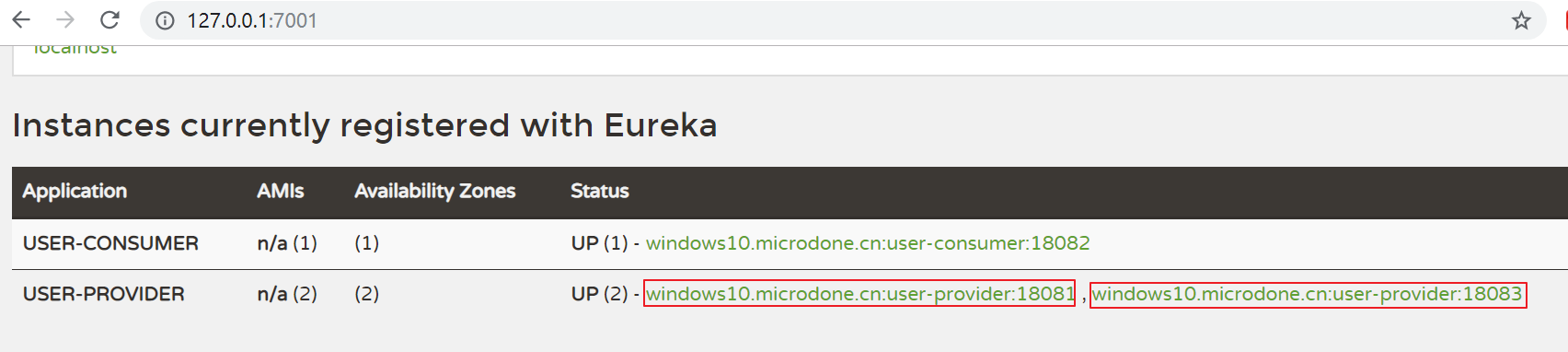
**user-provider-demo1**:

@RequestMapping(value = **"find/{id}"**)  
**public** User findById(@PathVariable(value = **"id"**) Integer id){  
 User user = **userService**.findByUserId(id);  
 user.setUsername(user.getUsername()+**" user-provider-demo1"**);  
 **return** user;  
}

##### 启动eureka-server和user-provider、user-provider-demo1、user-consumer，启动前先注释掉eureka-server中的自我保护和剔除服务配置。



访问eureka-server地址<http://127.0.0.1:7001/> 效果如下：



#### 开启负载均衡

##### 客户端开启负载均衡

Eureka已经集成Ribbon，所以无需引入依赖,要想使用Ribbon，直接在RestTemplate的配置方法上添加@LoadBalanced注解即可

修改user-consumer的com.itheima.UserConsumerApplication启动类，在restTemplate()方法上添加@LoadBalanced注解，代码如下：



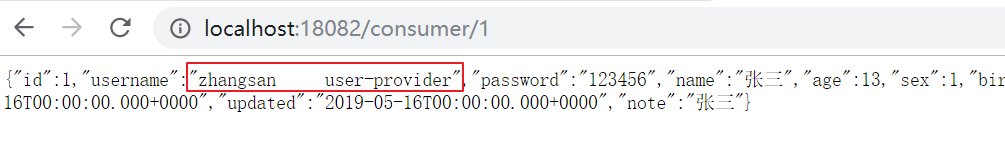
##### 采用服务名访问配置

修改user-consumer的com.itheima.controller.UserController的调用方式，不再手动获取ip和端口，而是直接通过服务名称调用，代码如下：

@GetMapping(value = **"/{id}"**)  
**public** User queryById(@PathVariable(value = **"id"**)Integer id){  
 */\*String url = "http://localhost:18081/user/find/"+id;  
 return restTemplate.getForObject(url,User.class);\*/  
 //获取指定生产者的实例列表  
  
 /\* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");  
 //获取第一个实例对象  
 ServiceInstance instance = instances.get(0);  
 String url = "http://"+instance.getHost()+":"+instance.getPort()+"/user/find/"+id;\*/* String url = **"http://user-provider/user/find/"**+id;  
 **return restTemplate**.getForObject(url,User.**class**);  
}

##### 测试

重启动并访问测试http://localhost:18082/consumer/1,可以发现，数据会在2个服务之间轮询切换。



#### 其他负载均衡策略配置

需要在消费者中配置修改轮询策略：Ribbon默认的负载均衡策略是轮询，通过如下

*# 修改服务地址轮询策略，默认是轮询，配置之后变随机***user-provider**:  
 **ribbon**:  
 *#轮询  
 #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule  
 #随机算法* **NFLoadBalancerRuleClassName**: com.netflix.loadbalancer.RandomRule  
 *#重试算法,该算法先按照轮询的策略获取服务,如果获取服务失败则在指定的时间内会进行重试，获取可用的服务  
 #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RetryRule  
 #加权法,会根据平均响应时间计算所有服务的权重，响应时间越快服务权重越大被选中的概率越大。刚启动时如果同统计信息不足，则使用轮询的策略，等统计信息足够会切换到自身规则。  
 #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.ZoneAvoidanceRule*

SpringBoot可以修改负载均衡规则，配置为ribbon.NFLoadBalancerRuleClassName

格式：{服务名称}.ribbon.NFLoadBalancerRuleClassName

### 负载均衡源码跟踪探究

为什么只输入了Service名称就可以访问了呢？不应该需要获取ip和端口吗？

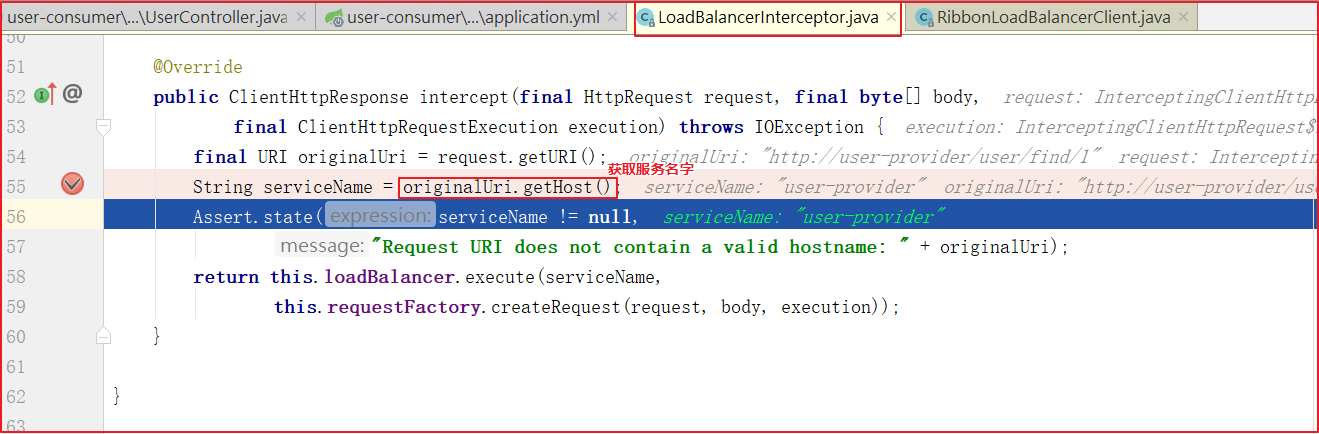
负载均衡器动态的从服务注册中心中获取服务提供者的访问地址(host、port)

显然是有某个组件根据Service名称，获取了服务实例ip和端口。就是LoadBalancerInterceptor

这个类会对RestTemplate的请求进行拦截，然后从Eureka根据服务id获取服务列表，随后利用负载均衡算法得到真正服务地址信息，替换服务id。

源码跟踪步骤：

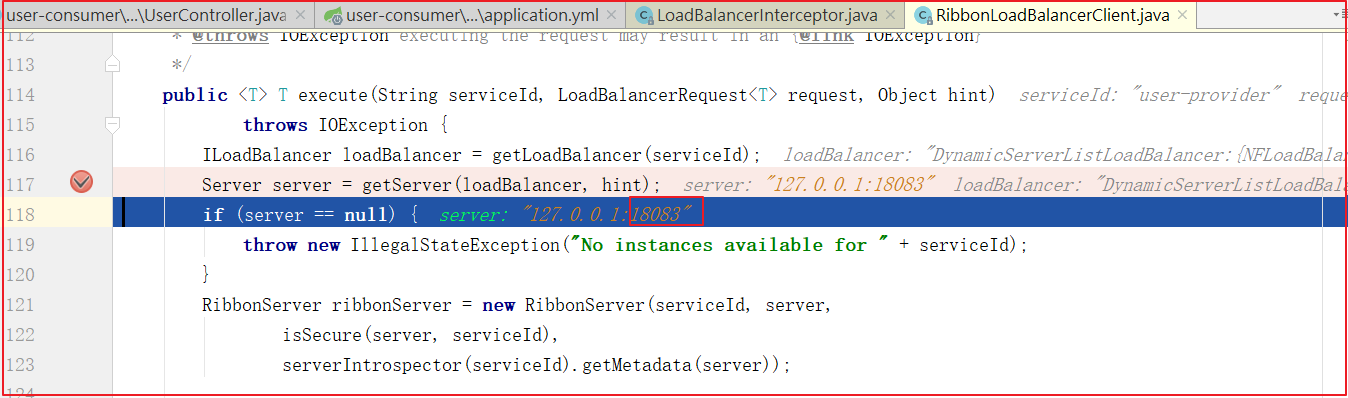
打开LoadBalancerInterceptor类，断点打入intercept方法中



继续跟入execute方法：发现获取了18081发端口的服务



再跟下一次，发现获取的是18081和18083之间切换



通过代码断点内容判断，果然是实现了负载均衡

## 小结

* Ribbon的负载均衡算法应用在客户端，只需要提供服务列表，就能帮助消费端自动访问服务端，并通过不同算法实现负载均衡。
* Ribbon的轮询、随机算法配置：在application.yml中配置 {服务名称}.ribbon.NFLoadBalancerRuleClassName
* 负载均衡的切换:在LoadBalancerInterceptor中获取服务的名字，通过调用RibbonLoadBalancerClient的execute方法，并获取ILoadBalancer负载均衡器，然后根据ILoadBalancer负载均衡器查询出要使用的节点，再获取节点的信息，并实现调用。

# 熔断器 Spring Cloud Hystrix

## 目标

* 理解Hystrix的作用
* 理解雪崩效应
* 知道熔断器的3个状态以及3个状态的切换过程
* 能理解什么是线程隔离，什么是服务降级
* 能实现一个局部方法熔断案例
* 能实现全局方法熔断案例

## Hystrix 简介



Hystrix，英文意思是豪猪，全身是刺，刺是一种保护机制。Hystrix也是Netflix公司的一款组件。

Hystrix的作用是什么？

Hystrix是Netflix开源的一个延迟和容错库，用于隔离访问远程服务、第三方库、防止出现级联失败也就是雪崩效应。

## 雪崩效应

什么是雪崩效应？

1.微服务中，一个请求可能需要多个微服务接口才能实现，会形成复杂的调用链路。

2.如果某服务出现异常，请求阻塞，用户得不到响应，容器中线程不会释放，于是越来越多用户请求堆积，越来越多线程阻塞。

3.单服务器支持线程和并发数有限，请求如果一直阻塞，会导致服务器资源耗尽，从而导致所有其他服务都不可用，从而形成雪崩效应；

Hystrix解决雪崩问题的手段，主要是服务降级\*\*(兜底)\*\*，线程隔离；

## 熔断原理分析



熔断器的原理很简单，如同电力过载保护器。

熔断器状态机有3个状态：

1.Closed：关闭状态，所有请求正常访问

2.Open：打开状态，所有请求都会被降级。

Hystrix会对请求情况计数，当一定时间失败请求百分比达到阈(yu：四声)值(极限值)，则触发熔断，断路器完全关闭

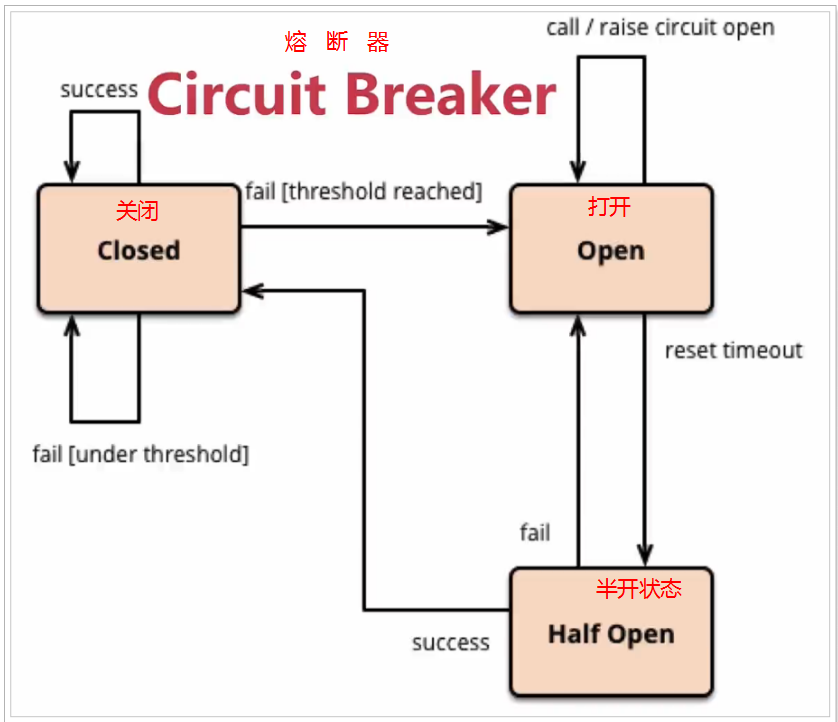
默认失败比例的阈值是50%，请求次数最低不少于20次

3.Half Open：半开状态

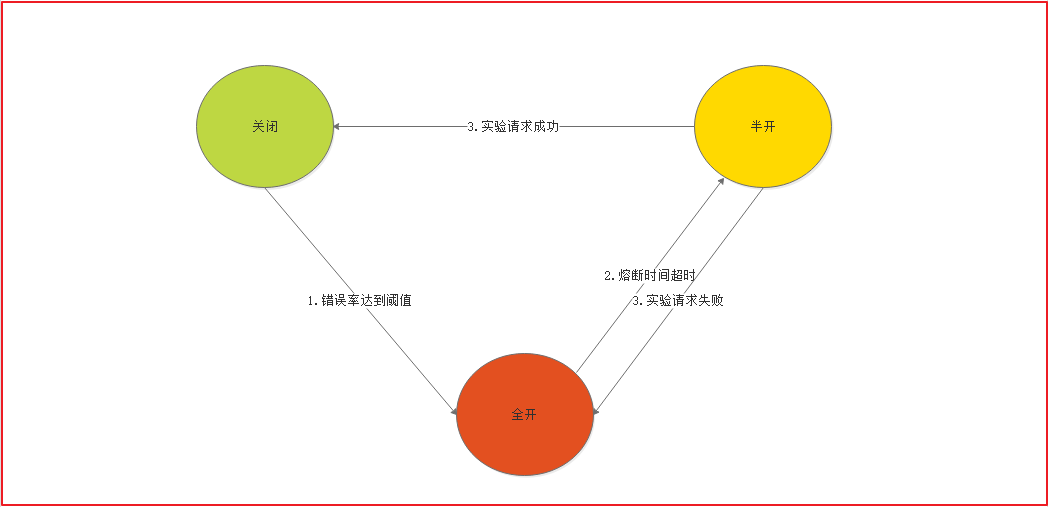
Open状态不是永久的，打开一会后会进入休眠时间(默认5秒)。休眠时间过后会进入半开状态。

半开状态：熔断器会判断下一次请求的返回状况，如果成功，熔断器切回closed状态。如果失败，熔断器切回open状态。

threshold reached 到达阈(yu：四声)值 under threshold 阈值以下



【Hystrix熔断状态机模型：配图】



熔断器的核心：线程隔离和服务降级。

1.线程隔离：是指Hystrix为每个依赖服务调用一个小的线程池，如果线程池用尽，调用立即被拒绝，默认不采用排队。

2.服务降级(兜底方法)：优先保证核心服务，而非核心服务不可用或弱可用。触发Hystrix服务降级的情况：线程池已满、请求超时。

线程隔离和服务降级之后，用户请求故障时，线程不会被阻塞，更不会无休止等待或者看到系统奔溃，至少可以看到执行结果(熔断机制)。

## 局部熔断案例

目标：服务提供者的服务出现了故障，服务消费者快速失败给用户友好提示。体验服务降级

### 引入熔断的依赖坐标

在user-consumer中加入依赖

*<!--熔断器-->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**>  
</**dependency**>

### 开启熔断的注解

修改user-consumer的com.itheima.UserConsumerApplication,在该类上添加@EnableCircuitBreaker,代码如下：

@SpringBootApplication  
@EnableDiscoveryClient *//开启eureka发现功能*@EnableCircuitBreaker *//开启熔断器***public class** UserConsumerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(UserConsumerApplication.**class**,args);  
 }  
  
 */\*\*\*  
 \* 将RestTemplate的实例放到Spring容器中  
 \** ***@return*** *\*/* @Bean  
 @LoadBalanced *//开启负载均衡* **public** RestTemplate restTemplate(){  
 **return new** RestTemplate();  
 }  
}

注意：这里也可以使用@SpringCloudApplication,写了@SpringCloudApplication后，其他注解需要全部去掉。

### 服务降级处理

在user-consumer的com.itheima.controller.UserController中添加降级处理方法，方法如下：

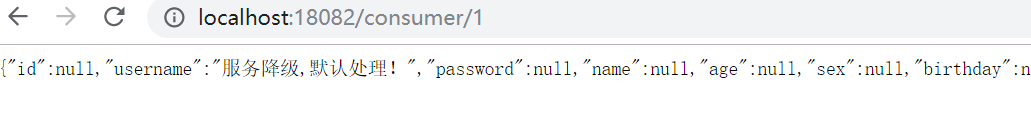
*/\*\*\*\*  
 \* 服务降级处理方法  
 \** ***@return*** *\*/***public** User failBack(Integer id){  
 User user = **new** User();  
 user.setUsername(**"服务降级,默认处理！"**);  
 **return** user;  
}

在有可能发生问题的方法上添加降级处理调用，例如在queryById方法上添加降级调用，代码如下：

@GetMapping(value = **"/{id}"**)  
@HystrixCommand(fallbackMethod = **"failBack"**) *//方法如果处理出问题，就调用降级处理方法***public** User queryById(@PathVariable(value = **"id"**)Integer id){  
 */\*String url = "http://localhost:18081/user/find/"+id;  
 return restTemplate.getForObject(url,User.class);\*/  
 //获取指定生产者的实例列表  
  
 /\* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");  
 //获取第一个实例对象  
 ServiceInstance instance = instances.get(0);  
 String url = "http://"+instance.getHost()+":"+instance.getPort()+"/user/find/"+id;\*/* String url = **"http://user-provider/user/find/"**+id;  
 **return restTemplate**.getForObject(url,User.**class**);  
}

### 测试

将服务全部停掉，启动eureka-server和user-consumer,然后请求<http://localhost:18082/consumer/1> 测试效果如下：



## 其他熔断策略配置(了解)

1. 熔断后休眠时间：sleepWindowInMilliseconds

2. 熔断触发最小请求次数：requestVolumeThreshold

3. 熔断触发错误比例阈值：errorThresholdPercentage

4. 熔断超时时间：timeoutInMilliseconds

配置如下：

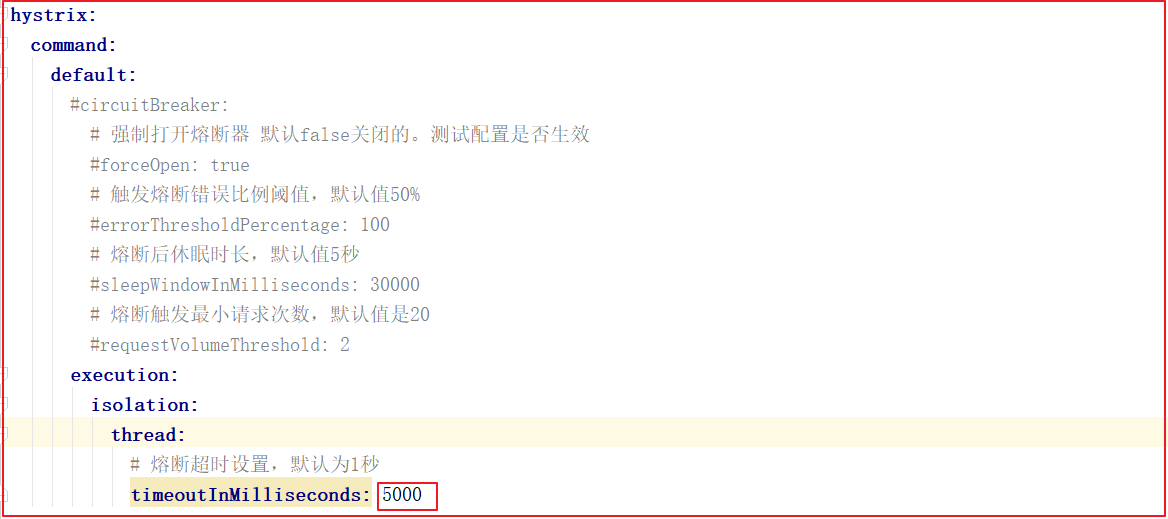
*# 配置熔断策略：***hystrix**:  
 **command**:  
 **default**:  
 **circuitBreaker**:  
 *# 强制打开熔断器 默认false关闭的。测试配置是否生效，打开后，所有请求都被拒绝* **forceOpen**: false

*# 熔断触发最小请求次数，默认值是20，允许同时请求的线程数* **requestVolumeThreshold**: 10  
 *# 触发熔断错误比例阈值，默认值50%，如果上面是10，经过些参数后就是5* **errorThresholdPercentage**: 50  
 *# 熔断后休眠时长，默认值5秒* **sleepWindowInMilliseconds**: 10000  
 **execution**:  
 **isolation**:  
 **thread**:  
 *# 熔断超时设置，默认为1秒* **timeoutInMilliseconds**: 2000

### 超时时间测试

a.修改user-provider的com.itheima.controller.UserController的findById方法，让它休眠3秒钟。

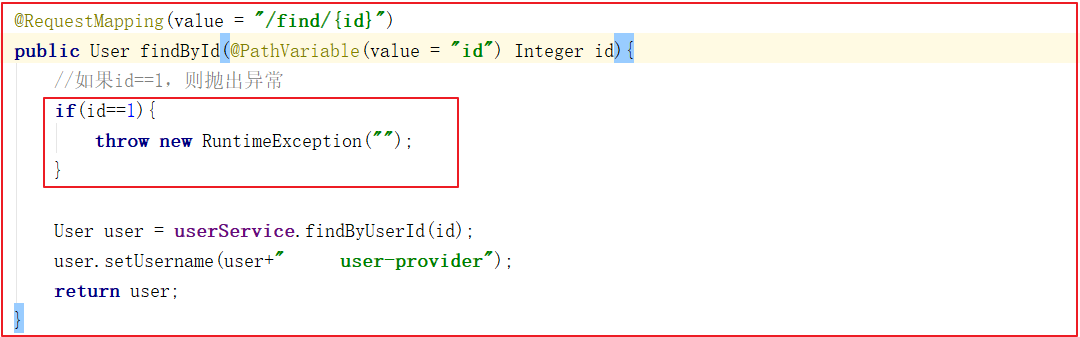
b.修改user-consumer的application.yml，设置超时时间5秒，此时不会熔断。



c.如果把超时时间改成2000，此时就会熔断。

### 熔断触发最小请求次数测试

a.修改user-provider的com.itheima.controller.UserController,在方法中制造异常，代码如下：



b.3次并发请求<http://localhost:18082/consumer/1>，会触发熔断

再次请求<http://localhost:18082/consumer/2>的时候，也会熔断，5秒钟会自动恢复。

并发请求建议使用jmeter工具。

## 扩展-服务降级的fallback方法：

两种编写方式：编写在类上，编写在方法上。在类的上边对类的所有方法都生效。在方法上，仅对当前方法有效。

### 方法上服务降级的fallback兜底方法

使用HystrixCommon注解，定义

@HystrixCommand(fallbackMethod="failBack")用来声明一个降级逻辑的fallback兜底方法

### 类上默认服务降级的fallback兜底方法

刚才把fallback写在了某个业务方法上，如果方法很多，可以将FallBack配置加在类上，实现默认FallBack

@DefaultProperties(defaultFallback=”defaultFailBack“)，在类上，指明统一的失败降级方法；

### 案例

1. 在user-consumer的com.itheima.controller.UserController类中添加一个全局熔断方法
2. 在queryById方法上将原来的@HystrixCommand相关去掉，并添加@HystrixCommand注解
3. 在user-consumer的com.itheima.controller.UserController类上添加@DefaultProperties(defaultFallback = "defaultFailBack")
4. 测试访问<http://localhost:18082/consumer/1>

Controller全部代码

@RestController  
@RequestMapping(value = **"consumer"**)  
@DefaultProperties(defaultFallback = **"defaultFailBack"**)  
**public class** UserController {  
  
 @Autowired  
 **private** RestTemplate **restTemplate**;  
 @Autowired  
 **private** DiscoveryClient **discoveryClient**; *//此对象用于向注册中心获取服务列表  
  
 /\*\*\*\*  
 \* 全局的服务降级处理方法  
 \** ***@return*** *\*/* **public** User defaultFailBack(){  
 User user = **new** User();  
 user.setUsername(**"Default-服务降级,默认处理！"**);  
 **return** user;  
 }  
  
 */\*\*\*\*  
 \* 在user-consumer服务中通过RestTemplate调用user-provider服务  
 \** ***@param id*** *\** ***@return*** *\*/* @GetMapping(value = **"/{id}"**)  
 *//@HystrixCommand(fallbackMethod = "failBack") //方法如果处理出问题，就调用降级处理方法* @HystrixCommand  
 **public** User queryById(@PathVariable(value = **"id"**)Integer id){  
 */\*String url = "http://localhost:18081/user/find/"+id;  
 return restTemplate.getForObject(url,User.class);\*/  
 //获取指定生产者的实例列表  
  
 /\* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");  
 //获取第一个实例对象  
 ServiceInstance instance = instances.get(0);  
 String url = "http://"+instance.getHost()+":"+instance.getPort()+"/user/find/"+id;\*/* String url = **"http://user-provider/user/find/"**+id;  
 **return restTemplate**.getForObject(url,User.**class**);  
 }  
  
 */\*\*\*\*  
 \* 服务降级处理方法  
 \** ***@return*** *\*/* **public** User failBack(Integer id){  
 User user = **new** User();  
 user.setUsername(**"服务降级,默认处理！"**);  
 **return** user;  
 }  
}

## 小结

* Hystrix的作用:用于隔离访问远程服务、第三方库、防止出现级联失败也就是雪崩效应。
* 理解雪崩效应:

properties

1.微服务中，一个请求可能需要多个微服务接口才能实现，会形成复杂的调用链路。

2.如果某服务出现异常，请求阻塞，用户得不到响应，容器中线程不会释放，于是越来越多用户请求堆积，越来越多线程阻塞。

3.单服务器支持线程和并发数有限，请求如果一直阻塞，会导致服务器资源耗尽，从而导致所有其他服务都不可用，从而形成雪崩效应；

* 知道熔断器的3个状态以及3个状态的切换过程

properties

1.Closed：关闭状态，所有请求正常访问

2.Open：打开状态，所有请求都会被降级。

Hystrix会对请求情况计数，当一定时间失败请求百分比达到阈(yu：四声)值(极限值)，则触发熔断，断路器完全关闭

默认失败比例的阈值是50%，请求次数最低不少于20次

3.Half Open：半开状态

Open状态不是永久的，打开一会后会进入休眠时间(默认5秒)。休眠时间过后会进入半开状态。

半开状态：熔断器会判断下一次请求的返回状况，如果成功，熔断器切回closed状态。如果失败，熔断器切回open状态。

threshold reached 到达阈(yu：四声)值

under threshold 阈值以下

* 能理解什么是线程隔离，什么是服务降级

properties

1.线程隔离：是指Hystrix为每个依赖服务调用一个小的线程池，如果线程池用尽，调用立即被拒绝，默认不采用排队。

2.服务降级(兜底方法)：优先保证核心服务，而非核心服务不可用或弱可用。触发Hystrix服务降级的情况：线程池已满、请求超时。

* 能实现一个局部方法熔断案例

properties

1.定义一个局部处理熔断的方法failBack()

2.在指定方法上使用@HystrixCommand(fallbackMethod = "failBack")配置调用

* 能实现全局方法熔断案例

properties

1.定义一个全局处理熔断的方法defaultFailBack()

2.在类上使用@DefaultProperties(defaultFallback = "defaultFailBack")配置调用

3.在指定方法上使用@HystrixCommand