SPRINGCLOUD

# 概述

# 入门案例

## 需求

对Student的增删改查 使用SpringCloud方式实现

CURD操作作为服务对外暴漏，前端通过RestTemplate进行远程调用。

* StudentProvider 对外提供服务
* StudentConsumer 通过远程调用服务 完成CRUD操作
* 项目采用MVEN 统一管理模式

## 父模块

springcloud\_student\_parent

该模块是用来统一管理其他子模块的所有依赖及编码方式

父模块中只需要添加所有子模块通用的依赖及定义编译版本即可

pom.xml

|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <druid.version>1.1.10</druid.version>  </properties>    <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>Edgware.SR4</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>1.5.13.RELEASE</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <!-- 连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>${druid.version}</version>  </dependency>  </dependencies>  </dependencyManagement> |

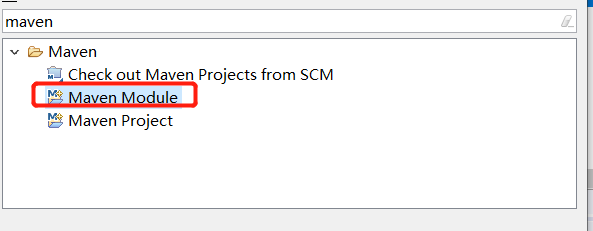
## 公共模块

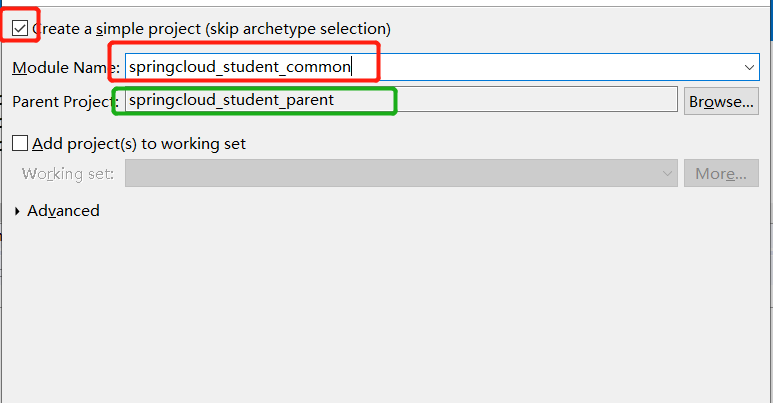
springcloud\_student\_common

公共模块是将公用的一些工具类，JavaBean等在其它子模块中抽离出来 使用时只需要引入即可 不需要每个子模块中在去编写重复的内容

### 创建方式

选中父项目 右键 新建maven module





### 添加springboot依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  </dependencies> |

### 新建JavaBean

JavaBean 需要实现序列化接口 因为远程调用时需要先序列化

|  |
| --- |
| package com.etjava.entity;  import java.io.Serializable;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.Id;  import javax.persistence.Table;  @Entity  @Table(name = "t\_student")  public class Student implements Serializable{  /\*\*  \*  \*/  private static final long serialVersionUID = 1L;  @Id  @GeneratedValue  private Integer id; // 编号  @Column(length=50)  private String name; // 姓名  @Column(length=50)  private String grade; // 年级  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getGrade() {  return grade;  }  public void setGrade(String grade) {  this.grade = grade;  }  } |

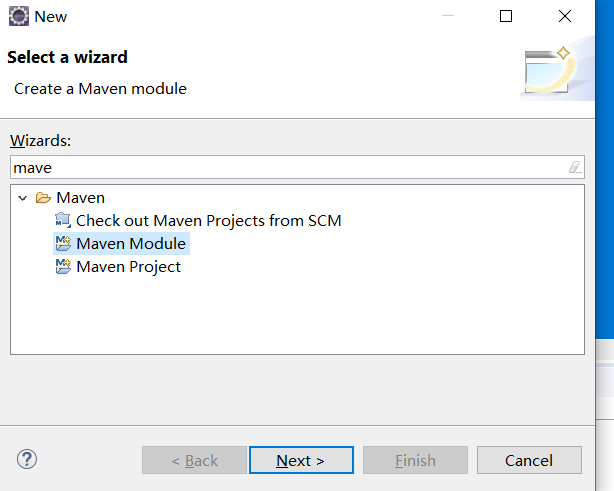
## 服务提供者

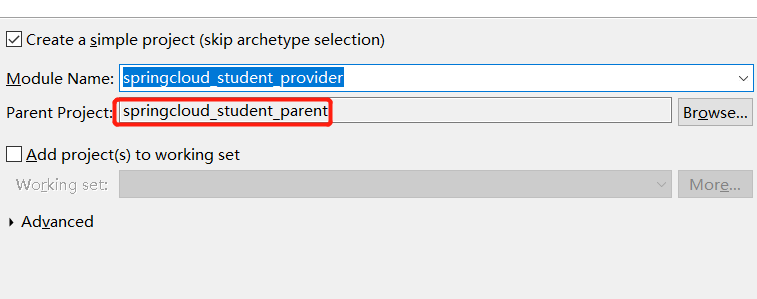
springcloud\_student\_provider

该模块为提供服务的子模块 需要对外暴漏服务 另外需要操作数据库进行数据的读写操作

### 创建方式

选中父项目右键 新建maven module





### 添加依赖

该模块中使用的JavaBean是来自公共模块的 因此这里还需要引入公共模块

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### Repository接口

Repository接口是SpringDataJPA的规范，Mybatis叫做Mapper

目的都是用来操作数据库的

|  |
| --- |
| **public** **interface** StudentRepository **extends** JpaRepository<Student, Integer>,JpaSpecificationExecutor<Student>{  } |

### Contorller

controller是对外提供服务的入口，消费者最终调用的是该入口的方法来实现CRUD操作 (RestTemplate方式)

注意：提供者在接收实体类型参数时需要通过@RequestBody 否则接收不到

|  |
| --- |
| /\*\*  \* 服务提供者Controller  \* **@author** etjav  \*  \*/  @RestController  @RequestMapping("/student")  **public** **class** StudentController {  @Resource  **private** StudentService studentService;    /\*\*  \* 添加或者修改学生信息  \* 服务提供者在接收实体数据时需要使用@RequestBody 否则接收不到数据  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/save")  **public** **boolean** save(@RequestBody Student student){  **try**{  studentService.save(student);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }    /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/list")  **public** List<Student> list(){  **return** studentService.list();  }    /\*\*  \* 根据id查询学生信息  \* **@return**  \*/  @GetMapping(value="/get/{id}")  **public** Student get(@PathVariable("id") Integer id){  **return** studentService.findById(id);  }  /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id){  **try**{  studentService.delete(id);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }  } |

### 启动类

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

### springboot配置文件

application.yml

需要配置数据源及访问的端口

|  |
| --- |
| server:  port: 1001  context-path: /    # 数据源配置  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://192.168.199.108:3306/db\_springcloud  username: root  password: Karen@1234  jpa:  hibernate:  ddl-auto: update  show-sql: true |

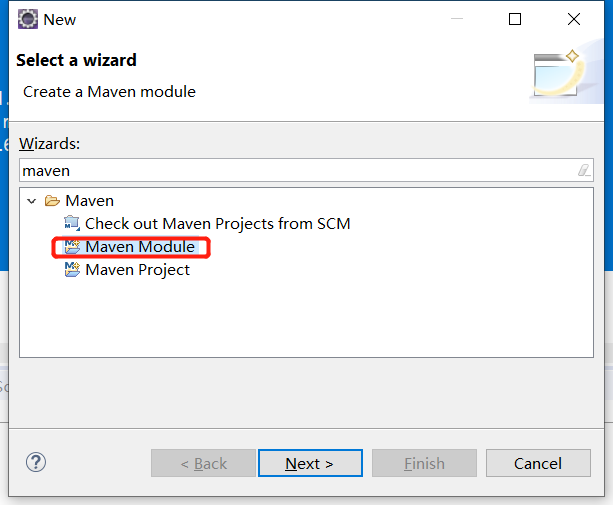
## 服务消费者

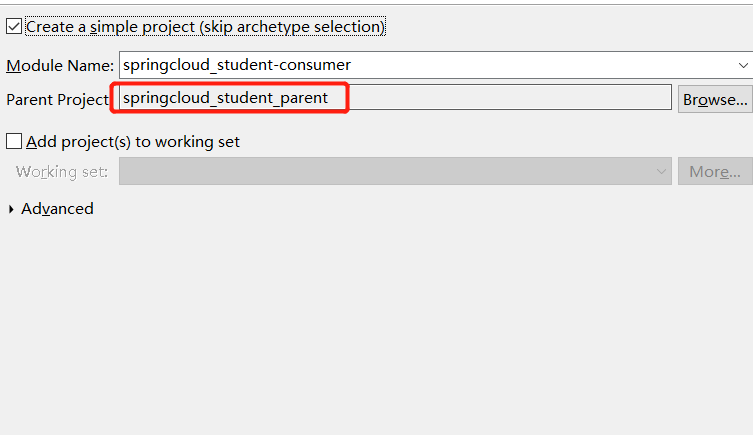
springcloud\_student-consumer

服务消费者模块是用来远程调用服务提供者的服务 实现CRUD操作

### 创建方式

选中父项目 右键 新建maven module





### 添加依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### SpringClooudConfig

需要通过SpringCloud的配置类 获取RestTemplate模板，然后在通过模板去远程调用服务

|  |
| --- |
| package com.etjava.config;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.web.client.RestTemplate;  /\*\*  \* Spring Cloud 相关配置  \* @author etjava  \*  \*/  @Configuration  public class SpringCloudConfig {  /\*\*  \* 调用服务模版  \* 通过RestTemplate 调用远程服务  \* @return  \*/  @Bean  public RestTemplate getRestTemplate(){  return new RestTemplate();  }  } |

### Contoller

消费者这边的controller是对外提供访问的 与服务提供者的controller区别在于这里的是调用服务提供者的服务 不需要自己去操作数据库

|  |
| --- |
| package com.etjava.controller;  import java.util.List;  import javax.annotation.Resource;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.web.bind.annotation.GetMapping;  import org.springframework.web.bind.annotation.PathVariable;  import org.springframework.web.bind.annotation.PostMapping;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RestController;  import org.springframework.web.client.RestTemplate;  import com.etjava.entity.Student;  /\*\*  \* 服务消费者 controller  \* @author etjav  \*  \*/  @RestController  @RequestMapping("/student")  public class StudentConsumerController {    /\*\*  \* restTemplate 调用远程服务端  \*/  @Resource  private RestTemplate restTemplate;    @Value("${provider\_host}")  private String HOST;    /\*\*  \* 添加或者修改学生信息  \* @param student  \* @return  \*/  @PostMapping(value="/save")  public boolean save(Student student){  return restTemplate.postForObject(HOST+"/student/save", student, Boolean.class);  }    /\*\*  \* 查询学生信息  \* @return  \*/  @GetMapping(value="/list")  public List<Student> list(){  return restTemplate.getForObject(HOST+"/student/list", List.class);  }    /\*\*  \* 根据id查询学生信息  \* @return  \*/  @GetMapping(value="/get/{id}")  public Student get(@PathVariable("id") Integer id){  return restTemplate.getForObject(HOST+"/student/get/"+id, Student.class);  }    /\*\*  \* 根据id删除学生信息  \* @return  \*/  @GetMapping(value="/delete/{id}")  public boolean delete(@PathVariable("id") Integer id){  return restTemplate.getForObject(HOST+"/student/delete/"+id,Boolean.class);  }  } |

### 启动类

启动类中需要注意，在springboot启动时会自动注入数据源，我们这里的服务消费者不需要操作数据库 因此也就不需要配置数据源，这里需要在启动时告诉springboot 不要加载数据源信息

|  |
| --- |
| package com.etjava;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;  import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;  /\*\*  \* exclude 表示不包含  \* 由于服务消费者模块不需要直接操作数据库 因此不需要注入数据源  \* @author etjav  \*  \*/  @SpringBootApplication(exclude={DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class})  public class StudentConsumerApp {  public static void main(String[] args) {  SpringApplication.run(StudentConsumerApp.class, args);  }  } |

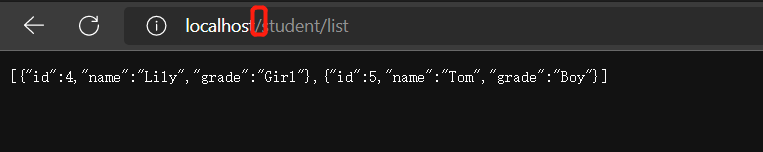
### springboot配置文件

|  |
| --- |
| server:  port: 80  context-path: /    provider\_host: http://localhost:1001 |

测试

80端口是服务消费者的

1001端口是服务提供者的



# Eureka注册中心搭建

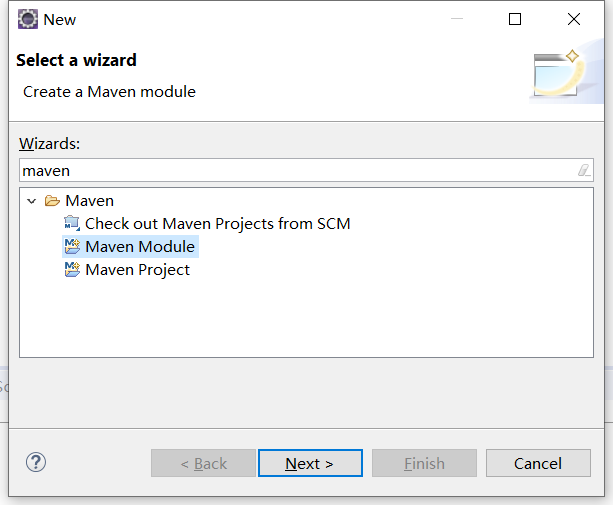
eureka采用的是C/S模式，服务的提供者是Server,其它都是Client

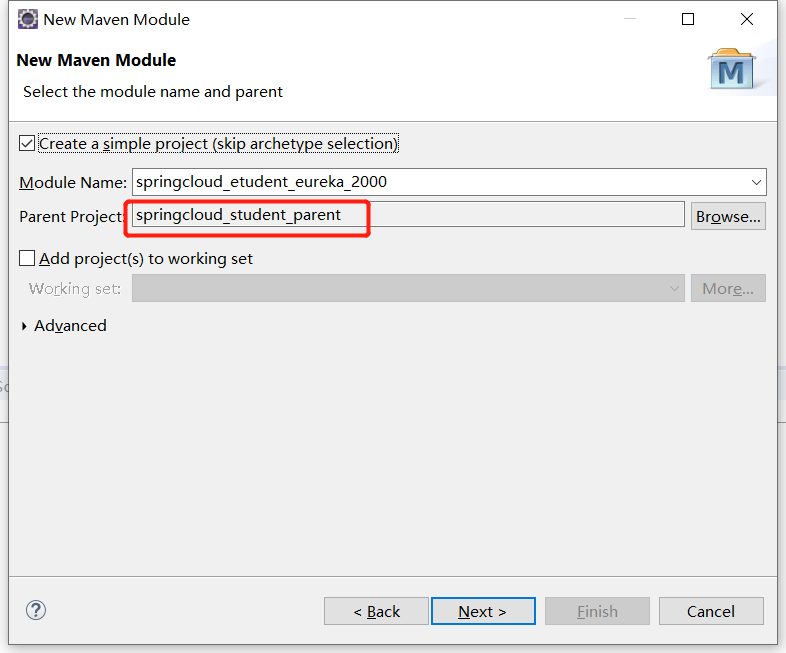
服务端用来管理所有服务，客户端通过注册中心来调用具体的服务

## 搭建eureka

springcloud\_etudent\_eureka\_2000

选中父项目 右键 新建maven module





## 添加Eureka依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

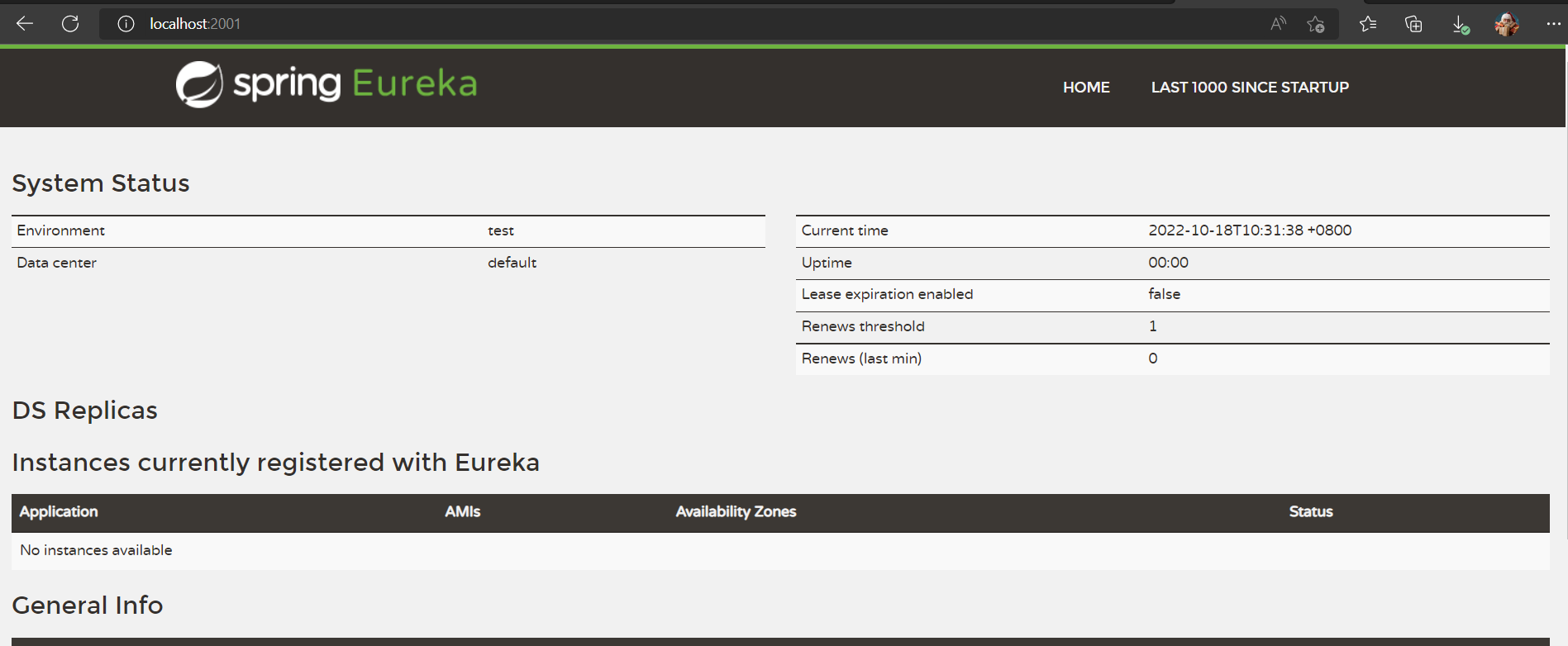
## application.yml 配置文件

|  |
| --- |
| server:  port: 2001  context-path: /    eureka:  instance:  hostname: localhost #eureka注册中心实例名称 - 访问地址  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索自己的服务，所以也设置为false(不需要检查自己是否存活状态)  service-url:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka注册中心交互的地址，查询服务和注册服务用到(对外提供服务的访问地址) |

## 启动类

|  |
| --- |
| package com.etjava;  import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  @EnableEurekaServer  @SpringBootApplication  public class EurekaApp\_2001 {  public static void main(String[] args) {  SpringApplication.run(EurekaApp\_2001.class, args);  }  } |

## 测试



# 注册服务到Eureka注册中心

## 修改服务提供者

修改之前的student\_provider 添加eureka客户端依赖

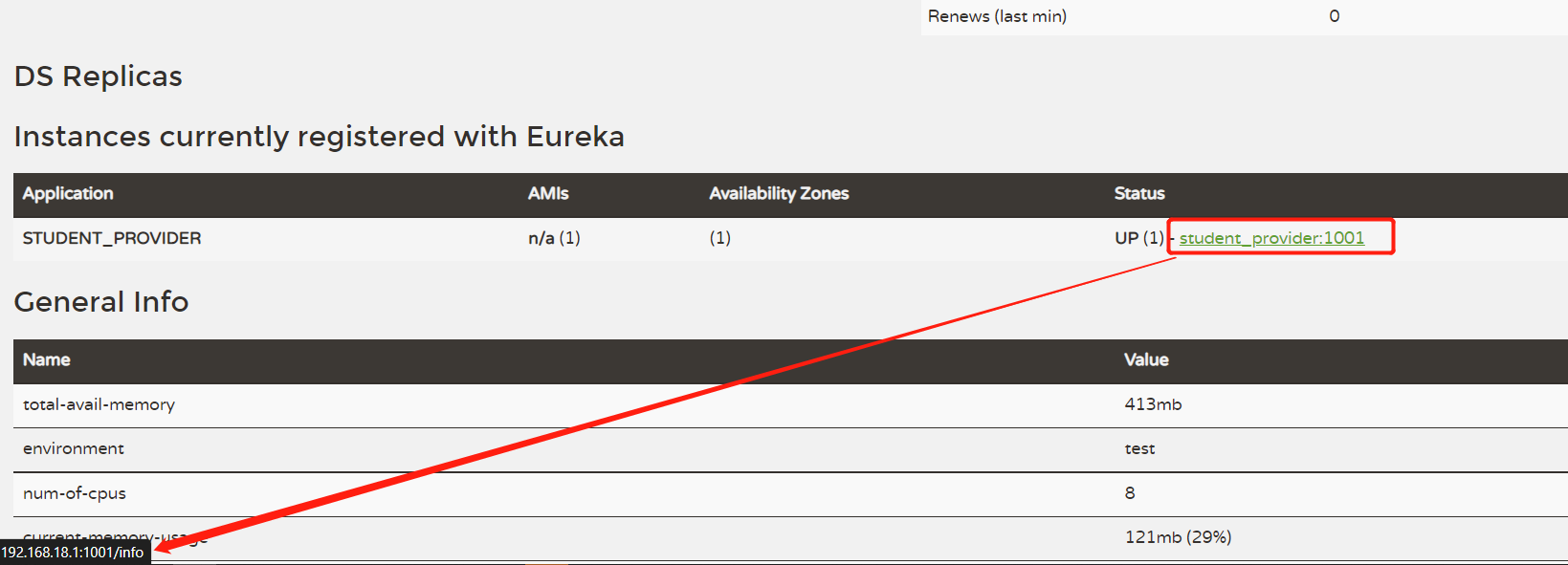
|  |
| --- |
| <!-- eureka注册中心依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency> |

## 启动类添加注解

启动类添加开启eureka注解

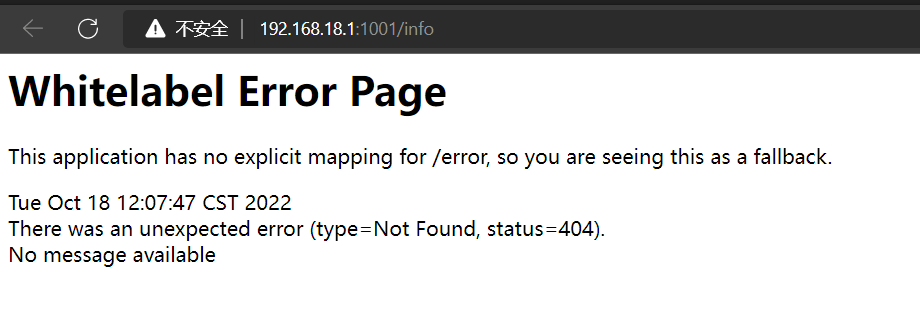
|  |
| --- |
| @EnableEurekaClient  @SpringBootApplication  **public** **class** StudentProviderApp {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentProviderApp.**class**, args);  }  } |

## 测试



# 解决Eureka的info访问问题

在点击注册中心具体服务的状态时 信息页面404



解决思路：

服务提供者模块中需要添加actuator监控的依赖 然后在application.yml中添加当前模块相关的描述信息

最后还需要在父项目中添加扫描所有子模块resource下的配置文件的插件 这样在启动注册中心时(注册中心也是作为子模块的)就会自动扫描所有子模块中的配置文件

## 服务提供者方面修改

在服务提供者的pom中引入actuator监控的依赖

|  |
| --- |
| <!-- actuator监控引入 -->  <**dependency**>      <**groupId**>org.springframework.boot</**groupId**>      <**artifactId**>spring-boot-starter-actuator</**artifactId**>  </**dependency**> |

## application.yml添加信息

在服务提供者的模块中添加当前模块的相关说明信息

|  |
| --- |
| info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 张三  联系电话: 123456 |

## 父项目中修改

父项目pom.xml里加上构建插件配置，主要是为了再构建的时候扫描子项目配置文件，解析配置用的

|  |
| --- |
| <!-- 构建的时候 解析 src/main/resources 下的配置文件  其实就是application.yml 解析以$开头和结尾的信息 -->  <**build**>      <**finalName**>microservice</**finalName**>      <**resources**>          <**resource**>              <**directory**>src/main/resources</**directory**>              <**filtering**>true</**filtering**>          </**resource**>      </**resources**>      <**plugins**>          <**plugin**>              <**groupId**>org.apache.maven.plugins</**groupId**>              <**artifactId**>maven-resources-plugin</**artifactId**>              <**configuration**>                  <**delimiters**>                      <**delimit**>$</**delimit**>                  </**delimiters**>              </**configuration**>          </**plugin**>      </**plugins**>  </**build**> |

## 测试



# Eureka集群配置

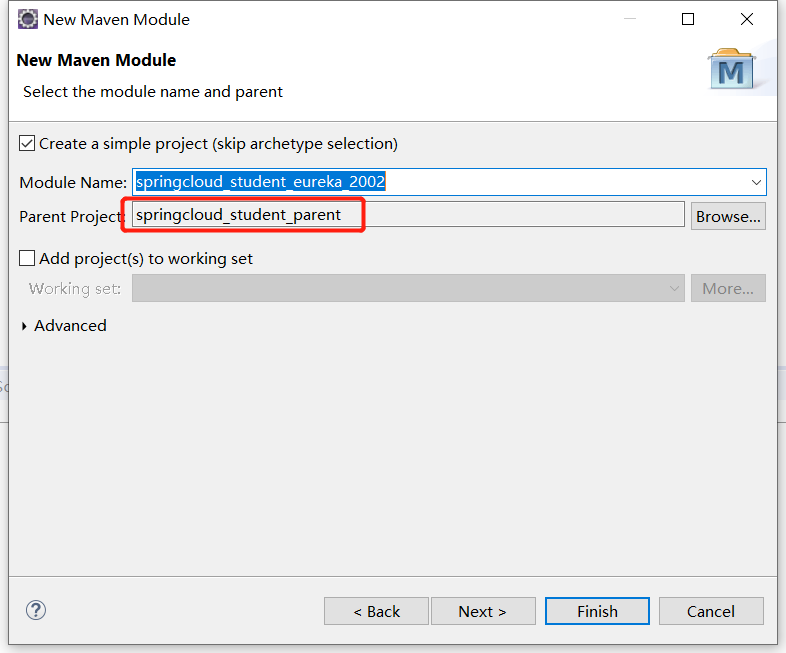
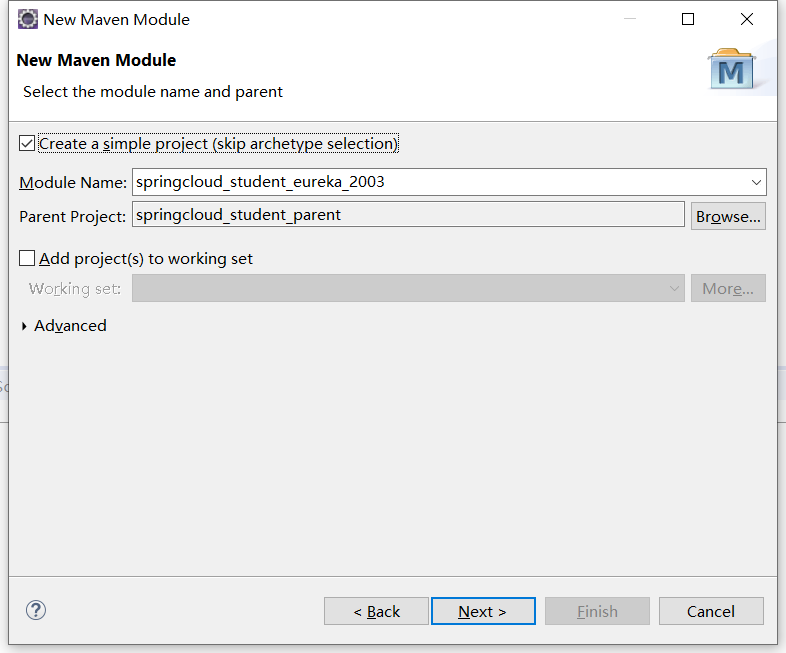
集群是用来解决访问压力问题，当消费者频繁访问注册中心获取服务时 单个注册中心压力会较大，此时可通过集群方式来分担注册中心的访问压力

## 创建集群

### 新建两个eureka的module

选中父项目 右键 新建maven module

springcloud\_student\_eureka\_2002, springcloud\_student\_eureka\_2003

### 添加依赖

两个新建的注册中心 分别添加如下依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### 创建启动类

两个注册中心分别创建启动类

|  |
| --- |
| **@EnableEurekaServer**  **@SpringBootApplication**  **public class EurekaApp\_2002 {**  **public static void main(String[] args) {**  **SpringApplication.run(EurekaApp\_2002.class, args);**  **}**  } |
| @EnableEurekaServer  @SpringBootApplication  **public** **class** EurekaApp\_2003 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(EurekaApp\_2002.**class**, args);  }  } |

### 修改本地映射文件 HOSTS

使用注册中心时需要指定访问的路径 但多个注册中心不能使用同一个访问的地址 这里使用映射模拟多台服务器 实际部署时 是部署到多台服务器的 访问地址自然就不相同了

hosts文件所在目录 C:\Windows\System32\drivers\etc



### 添加配置文件

为两个eureka注册中心添加配置文件

#### eureka2001

注册中心配置文件需要修改两个地方

一个是访问注册中心的地址 - hostname

一个是defaultZone 地址配置的是另外两个注册中心的访问地址带端口

|  |
| --- |
| server:  port: 2001  context-path: /    eureka:  instance:  # hostname: localhost 单机eureka注册中心实例名称 - 访问地址  hostname: eureka2001.etjava.com #eureka注册中心实例名称 - 访问地址  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索自己的服务，所以也设置为false(不需要检查自己是否存活状态)  service-url:  # 单机 设置与Eureka注册中心交互的地址，查询服务和注册服务用到(对外提供服务的访问地址)  # defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  # 集群 注意 不能注册到自己  defaultZone: http://eureka2002.etjava.com:2002/eureka/,http://eureka2003.etjava.com:2003/eureka/ |

#### eureka2002

注册中心配置文件需要修改两个地方

一个是访问注册中心的地址 - hostname

一个是defaultZone 地址配置的是另外两个注册中心的访问地址带端口

|  |
| --- |
| server:  port: 2002  context-path: /    eureka:  instance:  # hostname: localhost 单机eureka注册中心实例名称 - 访问地址  hostname: eureka2002.etjava.com #eureka注册中心实例名称 - 访问地址  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索自己的服务，所以也设置为false(不需要检查自己是否存活状态)  service-url:  # 单机 设置与Eureka注册中心交互的地址，查询服务和注册服务用到(对外提供服务的访问地址)  # defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  # 集群 注意 不能注册到自己  defaultZone: http://eureka2001.etjava.com:2001/eureka/,http://eureka2003.etjava.com:2003/eureka/ |

#### eureka2003

注册中心配置文件需要修改两个地方

一个是访问注册中心的地址 - hostname

一个是defaultZone 地址配置的是另外两个注册中心的访问地址带端口

|  |
| --- |
| server:  port: 2003  context-path: /    eureka:  instance:  # hostname: localhost 单机eureka注册中心实例名称 - 访问地址  hostname: eureka2003.etjava.com #eureka注册中心实例名称 - 访问地址  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索自己的服务，所以也设置为false(不需要检查自己是否存活状态)  service-url:  # 单机 设置与Eureka注册中心交互的地址，查询服务和注册服务用到(对外提供服务的访问地址)  # defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  # 集群 注意 不能注册到自己  defaultZone: http://eureka2001.etjava.com:2001/eureka/,http://eureka2002.etjava.com:2002/eureka/ |

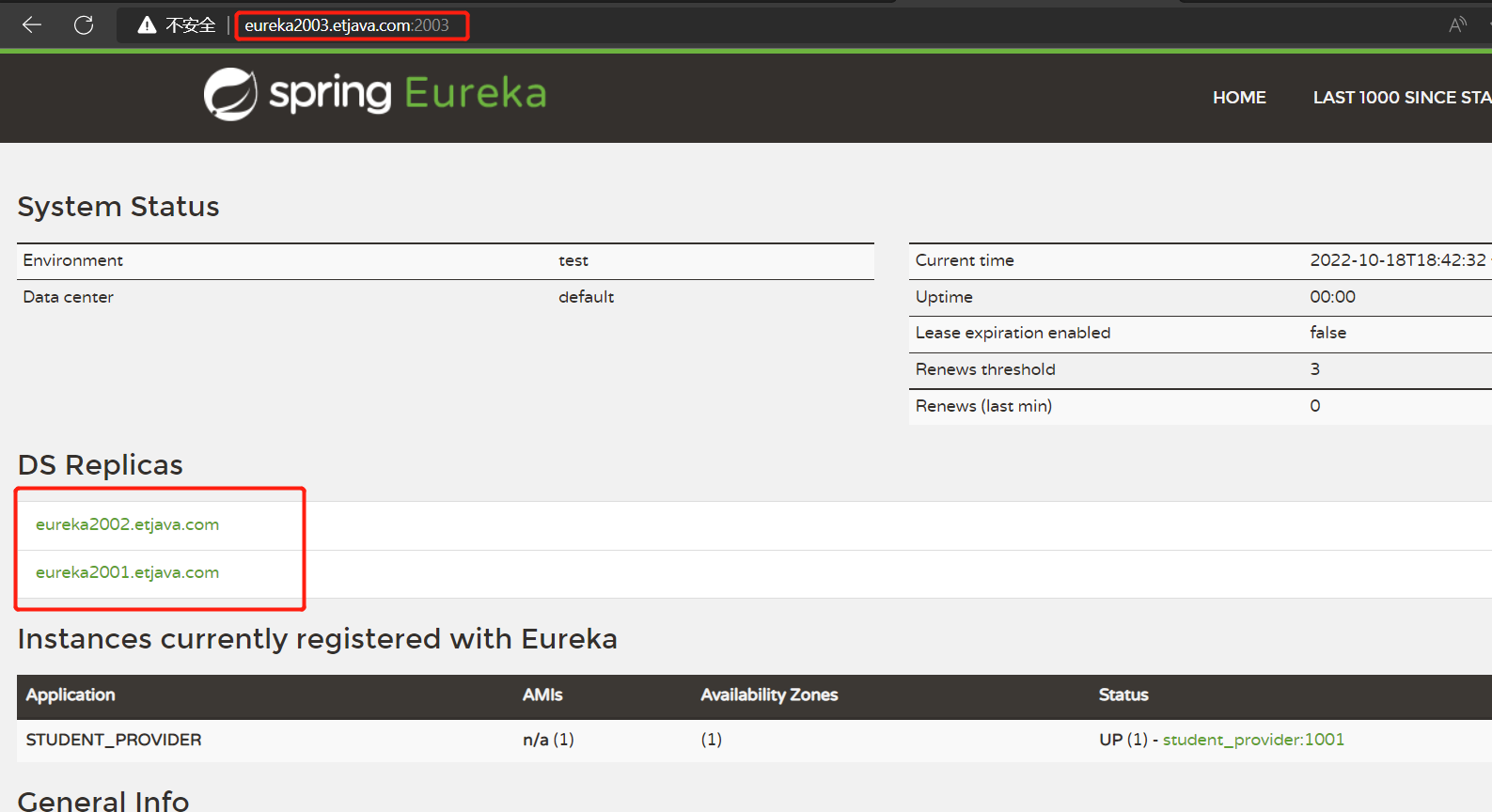
### 修改服务提供者

服务提供者配置文件从单个注册中心改为向多个注册中心注册服务

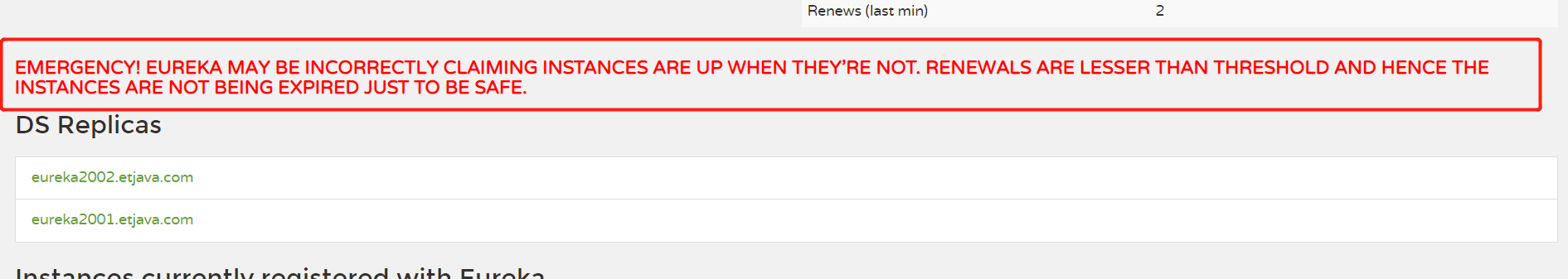
|  |
| --- |
| server:  port: 1001  context-path: /    # 数据源配置  spring:  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://192.168.199.108:3306/db\_springcloud  username: root  password: Karen@1234  jpa:  hibernate:  ddl-auto: update  show-sql: true  eureka:  instance:  hostname: localhost #eureka客户端主机实例名称  appname: student\_provider #客户端服务名  instance-id: student\_provider:1001 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  # 单机 - 把服务注册到eureka注册中心 这里的defaultZone要和前面服务注册中心的暴露地址一致  # defaultZone: http://localhost:2001/eureka  # 集群  defaultZone: http://eureka2001.etjava.com:2001/eureka,http://eureka2002.etjava.com:2002/eureka,http://eureka2002.etjava.com:2002/eureka      info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 张三  联系电话: 123456 |

### 测试

启动三个注册中心 然后在启动服务提供者



# Eureka自我保护机制



在我们长时间不访问服务或变更服务名称时就会出现该警告

这是由于注册中心在一段时间内(默认90秒)没有接收到某个服务的实例的心跳 那么注册中心就会注销该实例。

在生产环境如遇到网络延迟或网络故障等 导致服务于注册中心不能正常通讯 但此时服务是正常的 不应该注销该服务，此时eureka会通过自我保护模式来解决问题，当短时间内和服务失去通讯时 保留该服务 当通讯恢复后会退出自我保护机制

eureka通过自我保护机制可以使eureka集群更加健壮和稳定。

也可以通过配置关闭自我保护 但会提示另个警告(关闭了自我保护的警告)

# 服务提供者集群

创建多个服务提供者注册服务到eureka注册中心(集群模式)

这里采用三个服务提供者 之前已经有一个1001 在创建1002和1003

springcloud\_student\_provider

springcloud\_student\_provider\_1002

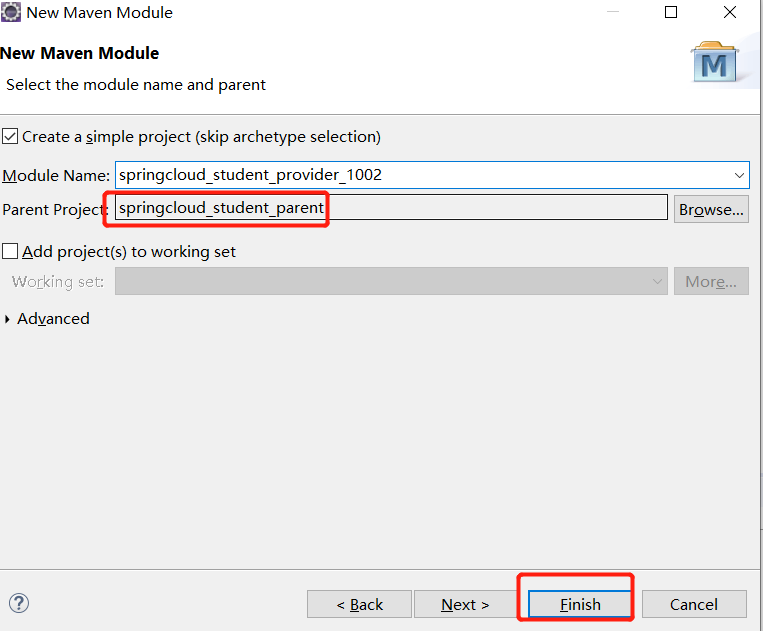
springcloud\_student\_provider\_1003

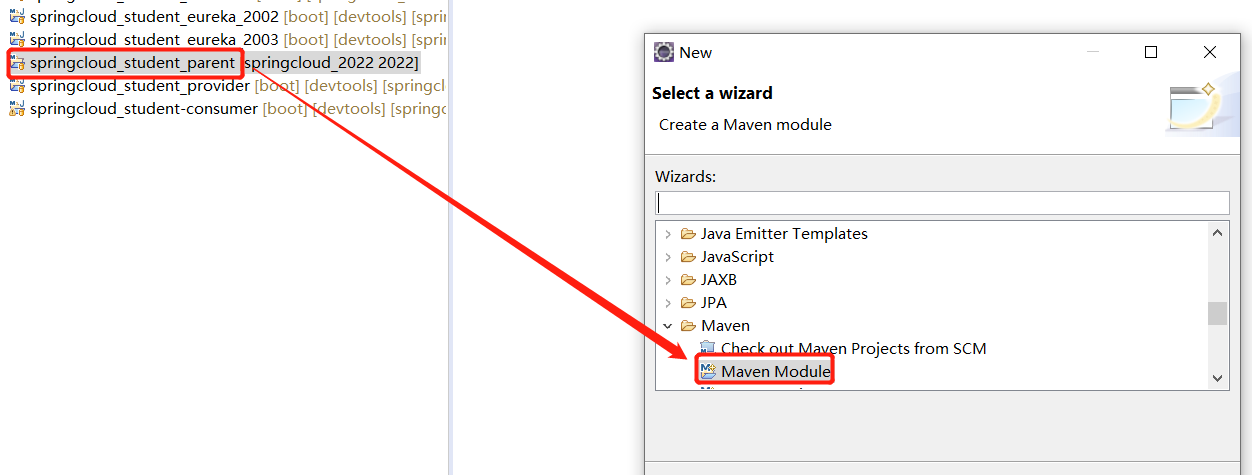
## 创建服务提供者

springcloud\_student\_provider\_1003

选中父项目 右键maven module

1002，1003创建方式相同 这里只演示一个项目 区别在于模块名称





## 1002添加依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>    <!-- eureka注册中心依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>    <!-- actuator监控引入 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>      <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

## 1002添加启动类

|  |
| --- |
| @EnableEurekaClient  @SpringBootApplication  **public** **class** StudentProviderApp\_1002 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentProviderApp\_1002.**class**, args);  }  } |

## 1002添加Repository接口

|  |
| --- |
| **public** **interface** StudentRepository **extends** JpaRepository<Student, Integer>,JpaSpecificationExecutor<Student>{  } |

## 1002 添加Service接口及实现类

### Service接口

|  |
| --- |
| **public** **interface** StudentService {  /\*\*  \* 添加或者修改学生信息  \* **@param** student  \*/  **public** **void** save(Student student);    /\*\*  \* 根据id查找学生信息  \* **@param** id  \* **@return**  \*/  **public** Student findById(Integer id);    /\*\*  \* 查询学生信息  \* **@return**  \*/  **public** List<Student> list();    /\*\*  \* 根据id删除学生信息  \* **@param** id  \*/  **public** **void** delete(Integer id);  } |

### ServiceImpl

|  |
| --- |
| @Service  **public** **class** StudentServiceImpl **implements** StudentService{  @Resource  **private** StudentRepository studentRepository;  @Override  **public** **void** save(Student student) {  studentRepository.save(student);  }  @Override  **public** Student findById(Integer id) {  **return** studentRepository.findOne(id);  }  @Override  **public** List<Student> list() {  **return** studentRepository.findAll();  }  @Override  **public** **void** delete(Integer id) {  studentRepository.delete(id);  }  } |

## 1002添加application.yml配置文件

配置文件中端口号要修改为1002

eureka的instance-id: 修改为 student-provider:1002

|  |
| --- |
| server:  port: 1002  context-path: /    # 数据源配置  spring:  application:  # 应用名称 这里消费者使用Ribbon在调用时需要根据该名称调用 这里不能使用下环线 否则消费者那边使用同样的名称 调用不到  name: student-provider22  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://192.168.199.108:3306/db\_springcloud  username: root  password: Karen@1234  jpa:  hibernate:  ddl-auto: update  show-sql: true  eureka:  instance:  hostname: localhost #eureka客户端主机实例名称  appname: student-provider #客户端服务名  instance-id: student-provider:1002 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  # 单机 - 把服务注册到eureka注册中心 这里的defaultZone要和前面服务注册中心的暴露地址一致  # defaultZone: http://localhost:2001/eureka  # 集群  defaultZone: http://eureka2001.etjava.com:2001/eureka,  http://eureka2002.etjava.com:2002/eureka,  http://eureka2002.etjava.com:2002/eureka      info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 李四  联系电话: 123456 |

## 1002 添加Controller

controller与1001完全相同的 这里只是模拟同一个模块使用集群方式发送到eureka中

|  |
| --- |
| /\*\*  \* 服务提供者Controller  \* **@author** etjav  \*  \*/  @RestController  @RequestMapping("/student")  **public** **class** StudentController {  @Resource  **private** StudentService studentService;    /\*\*  \* 添加或者修改学生信息  \* 服务提供者在接收数据时需要使用@RequestBody 否则接收不到数据  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/save")  **public** **boolean** save(@RequestBody Student student){  **try**{  studentService.save(student);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }    /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/list")  **public** List<Student> list(){  System.***out***.println("provider ------------------------------ 1002");  **return** studentService.list();  }    /\*\*  \* 根据id查询学生信息  \* **@return**  \*/  @GetMapping(value="/get/{id}")  **public** Student get(@PathVariable("id") Integer id){  **return** studentService.findById(id);  }    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id){  **try**{  studentService.delete(id);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }  } |

## 1003配置说明

1003与1002配置大致相同

区别1 启动类名称改为1003

区别2 配置文件中 当前模块的端口改为1003， instance-id 跟着的端口改为1003

区别3 controller中为了模拟查看是具体哪个服务被调用 在list请求中添加了控制台输出标记当前模块的信息

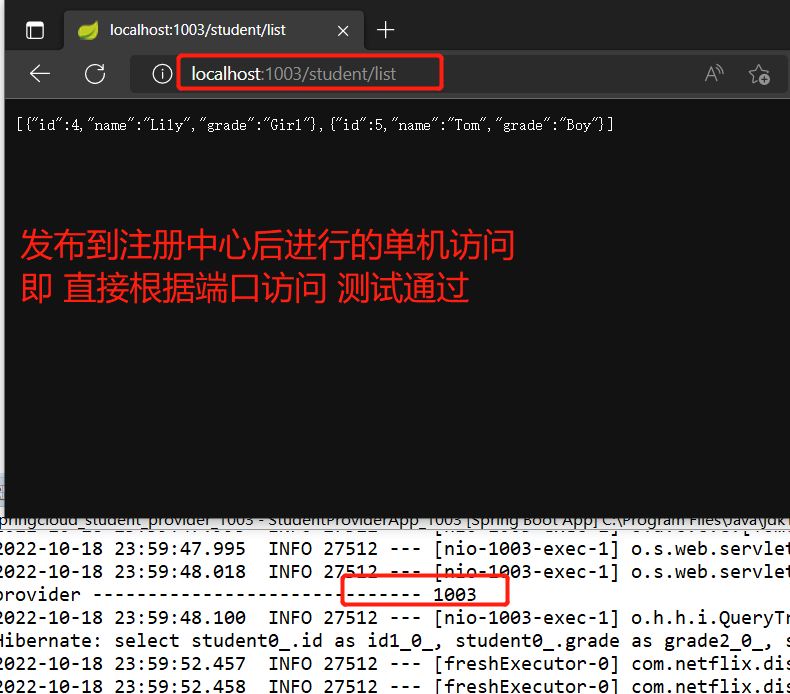
其它都与1001，1002完全相同的

## 测试服务提供者集群

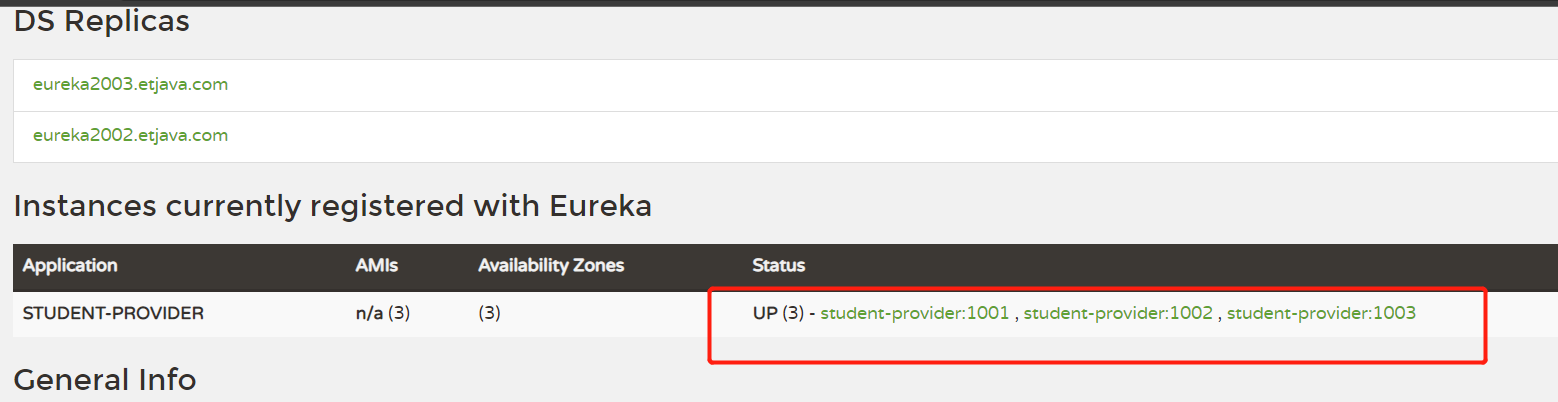
先启动三个eureka注册中心

在启动三个服务提供者

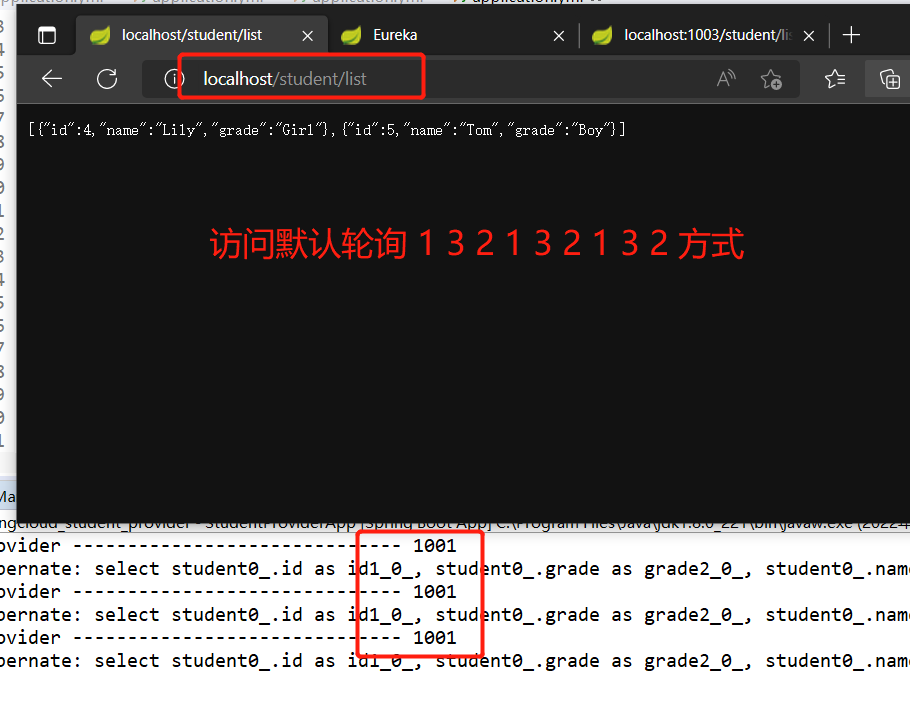
最后启动服务消费者进行测试



eureka注册中心查看



启动服务消费者测试



# Ribbon

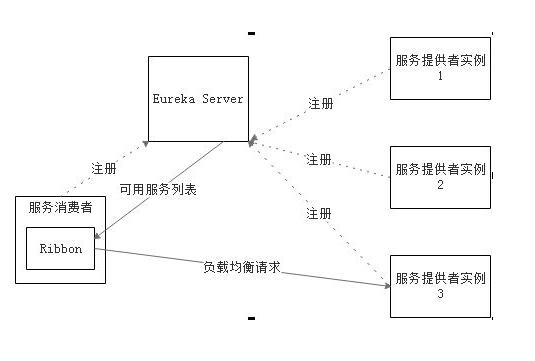
## 概述

eureka是服务注册与发现的注册中心，可以采用集群模式对外提供服务

Ribbon用来结合eureka注册中心集群调用服务的

Ribbon可自动从Eureka Server获取服务提供者地址列表，并基于负载均衡算法，请求其中一个服务提供者实例

简单理解 eureka是服务提供者的集群，ribbon是协助服务消费者来访问服务集群 ribbon提供了7种访问方式 默认是轮询访问



## 修改student\_consumer

## 添加ribbon相关依赖

|  |
| --- |
| <**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.cloud</**groupId**>      <**artifactId**>spring-cloud-starter-config</**artifactId**>  </**dependency**> |

## 修改application.yml

|  |
| --- |
| # ribbon 相关配置  eureka:  client:  #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  register-with-eureka: false  service-url:  defaultZone: http://eureka2001.etjava.com:2001/eureka/,  http://eureka2002.etjava.com:2002/eureka/,  http://eureka2003.etjava.com:2003/eureka/ |

## SPringCloudConfig配置类修改

在方法上添加@LoadBalanced注解即可

|  |
| --- |
| /\*\*  \* Spring Cloud 相关配置  \* **@author** etjava  \*  \*/  @Configuration  **public** **class** SpringCloudConfig {  /\*\*  \* 调用服务模版  \* 通过RestTemplate 调用远程服务  \* **@return**  \*/  @Bean  @LoadBalanced // 引入ribbon负载均衡  **public** RestTemplate getRestTemplate(){  **return** **new** RestTemplate();  }  } |

## 启动类修改

因为和eureka整合，所以启动类StudentConsumerApplication\_80 加个注解 @EnableEurekaClient

|  |
| --- |
| /\*\*  \* exclude 表示不包含  \* 由于服务消费者模块不需要直接操作数据库 因此不需要注入数据源  \*  \* **@EnableEurekaClient** 开启eureka客户端  \* **@author** etjav  \*  \*/  @EnableEurekaClient  @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,HibernateJpaAutoConfiguration.**class**})  **public** **class** StudentConsumerApp {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentConsumerApp.**class**, args);  }  } |

## 服务提供者修改

服务提供者需要修改访问的应用名称 添加application.name

在消费者调用服务时 会根据这个name属性值找到对应的服务

注意：这里的application.name不能使用下环线 否则消费者访问不到(消费者配置相同带有下划线的名称也无法访问)

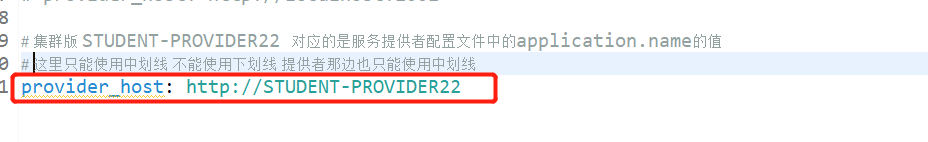
通常application.name与下边的eureka中的appname相同 这里只是区分测试

|  |
| --- |
| application:  # 应用名称 这里消费者使用Ribbon在调用时需要根据该名称调用 这里不能使用下环线 否则消费者那边使用同样的名称 调用不到  name: student-provider22 |

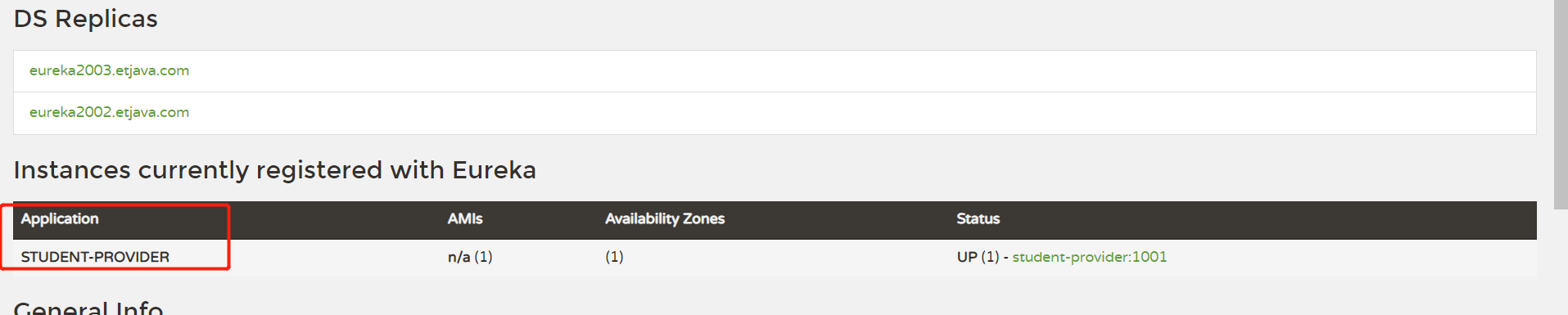
## controller修改

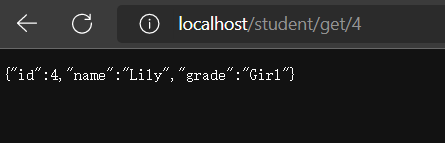
消费者的controller中需要修改访问地址为服务的实例名称

controller中RestTemplate中使用的URL 在消费者的配置文件中进行了配置 因此只需要修改配置文件中的内容就可以了



## 测试





## 配置消费者轮询访问服务

Ribbon默认不配置时采用轮询方式去访问服务，例如有三个服务提供者 那么如果有一台宕机 默认的轮询方式会有三分之一的概率由于访问不到服务而报错，因此需要指定轮询方式

访问策略：

|  |  |  |  |
| --- | --- | --- | --- |
| 策略名 | 策略声明 | 策略描述 | 实现说明 |
| BestAvailableRule | public class BestAvailableRule extends ClientConfigEnabledRoundRobinRule | 选择一个最小的并发请求的server | 逐个考察Server，如果Server被tripped了，则忽略，在选择其中ActiveRequestsCount最小的server |
| AvailabilityFilteringRule | public class AvailabilityFilteringRule extends PredicateBasedRule | 过滤掉那些因为一直连接失败的被标记为circuit tripped的后端server，并过滤掉那些高并发的的后端server（active connections 超过配置的阈值） | 使用一个AvailabilityPredicate来包含过滤server的逻辑，其实就就是检查status里记录的各个server的运行状态 |
| WeightedResponseTimeRule | public class WeightedResponseTimeRule extends RoundRobinRule | 根据响应时间分配一个weight，响应时间越长，weight越小，被选中的可能性越低。 | 一个后台线程定期的从status里面读取评价响应时间，为每个server计算一个weight。Weight的计算也比较简单responsetime 减去每个server自己平均的responsetime是server的权重。当刚开始运行，没有形成status时，使用roubine策略选择server。 |
| RetryRule | public class RetryRule extends AbstractLoadBalancerRule | 对选定的负载均衡策略机上重试机制。 | 在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server |
| RoundRobinRule | public class RoundRobinRule extends AbstractLoadBalancerRule | roundRobin方式轮询选择server | 轮询index，选择index对应位置的server |
| RandomRule | public class RandomRule extends AbstractLoadBalancerRule | 随机选择一个server | 在index上随机，选择index对应位置的server |
| ZoneAvoidanceRule | public class ZoneAvoidanceRule extends PredicateBasedRule | 复合判断server所在区域的性能和server的可用性选择server | 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。 |

springcloudconfig中配置

|  |
| --- |
| /\*\*  \* 自定义轮询算法 默认5次轮询 5次访问仍热失败则不在去访问失败的节点  \* **@return**  \*/  @Bean  **public** IRule myRule(){  **return** **new** RetryRule();  } |



# Feign

## 概述

Ribbon调用服务提供者，通过restTemplate调用，缺点是多个地方调用，同一个请求要写多次，不方便统一维护，这时候Feign来了，Feign直接把请求统一搞一个service作为FeignClient，然后其他Controller需要用到的，直接注入service，然后调用service方法即可；同时Feign整合了Ribbon和Eureka，所以要配置负载均衡的话，直接配置Ribbon即可，无其他特殊地方；当然Fiegn也整合了服务容错保护，断路器Hystrix

## 公共包定义FeignClient接口

在common模块里定义Feign的Client接口 这样做的目的是可以让其它服务消费者直接通过注解方式引用接口

### common中添加Feign的依赖

|  |
| --- |
| <!-- Feign的依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency> |

### 新建service接口

|  |
| --- |
| /\*\*  \* 基于feign方式的客户端访问时需要的service接口  \* **@FeignClient**(value="STUDENT-PROVIDER") 该接口需要指定访问的实例名称 即 application.name 不需要填写http:// 底层封装了  \* 底层实现由feign帮着实现了 类似MybatisPlus  \*  \* **@author** etjav  \*  \*/  @FeignClient(value="STUDENT-PROVIDER")  **public** **interface** StudentServiceWithFeign {  /\*\*  \* 根据id查询学生信息  \* **@param** id  \* **@return**  \*/  @GetMapping(value="/student/get/{id}")  **public** Student get(@PathVariable("id") Integer id);  /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/student/list")  **public** List<Student> list();  /\*\*  \* 添加或者修改学生信息  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/student/save")  **public** **boolean** save(Student student);    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/student/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id);    } |

## 创建基于feign的模块

### 操作步骤

1 创建模块 将另个consumer模块的内容复制一份过来

2配置文件不需要修改

3 启动类添加开启Feign客户端的注解 @EnableFeignClients

4 controller中将RestTemplate换成Service接口方式调用

4.1 service接口方式是在common包中定义的

4.1.1 在commom公共包中添加feign的依赖

4.1.2 添加service接口

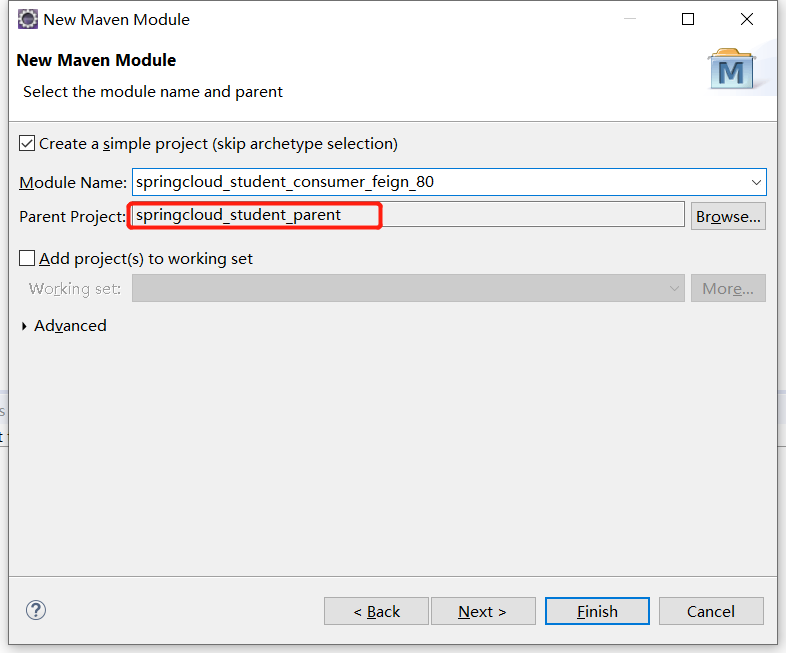
接口类上通过@FeignClient(value="STUDENT-PROVIDER")指定访问的地址

5 测试 启动eureka集群，启动多个provider向eureka集群中注册服务，启动consumer(带有Feign的consumer)进行测试

springcloud\_student\_consumer\_feign\_80

### 创建模块

选中父项目 右键 新建maven module



### 添加依赖

Feign是基于Ribbo 因此Feign依赖了Ribbon 这里只需要在引入Feign就可以了 不需要重复引入ribbon

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>    <!-- ribbon相关依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>    <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### 添加启动类

启动类中需要开启Feign 通过@EnableFeignClients

|  |
| --- |
| /\*\*  \* exclude 表示不包含  \* 由于服务消费者模块不需要直接操作数据库 因此不需要注入数据源  \*  \* **@EnableEurekaClient** 开启eureka客户端  \* **@EnableFeignClients** 开启feign的注解  \* **@author** etjav  \*  \*/  @EnableFeignClients  @EnableEurekaClient  @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,  HibernateJpaAutoConfiguration.**class**})  **public** **class** StudentConsumerFeignApp {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentConsumerFeignApp.**class**, args);  }  } |

### 配置SpringCloudConfig

主要用来配置Ribbon的方式服务集群的方式

这里的RestTemplate不需要配置了 因为他与Feign是两种访问机制

|  |
| --- |
| @Configuration  **public** **class** SpringCloudConfig {  /\*\*  \* 自定义轮询算法 默认5次轮询 5次访问仍热失败则不在去访问失败的节点  \* **@return**  \*/  @Bean  **public** IRule myRule(){  **return** **new** RetryRule();  }  } |

Controller的调整

将之前的RestTemplate改为Feign的service接口方式

此方式更接近于SpringMVC的引用方式

|  |
| --- |
| **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.beans.factory.annotation.Value;  **import** org.springframework.web.bind.annotation.GetMapping;  **import** org.springframework.web.bind.annotation.PathVariable;  **import** org.springframework.web.bind.annotation.PostMapping;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** com.etjava.entity.Student;  **import** com.etjava.service.StudentServiceWithFeign;  /\*\*  \* 服务消费者 controller  \* **@author** etjav  \*  \*/  @RestController  @RequestMapping("/student")  **public** **class** StudentConsumerController {      @Autowired  **private** StudentServiceWithFeign studentServiceWithFeign;    @Value("${provider\_host}")  **private** String HOST;    /\*\*  \* 添加或者修改学生信息  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/save")  **public** **boolean** save(Student student){  **return** studentServiceWithFeign.save(student);  }    /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/list")  **public** List<Student> list(){  **return** studentServiceWithFeign.list();  }    /\*\*  \* 根据id查询学生信息  \* **@return**  \*/  @GetMapping(value="/get/{id}")  **public** Student get(@PathVariable("id") Integer id){  **return** studentServiceWithFeign.get(id);  }    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id){  **return** studentServiceWithFeign.delete(id);  }  } |

### application.yml配置文件

该配置文件与Ribbon方式访问服务集群一致 这里没有做改动

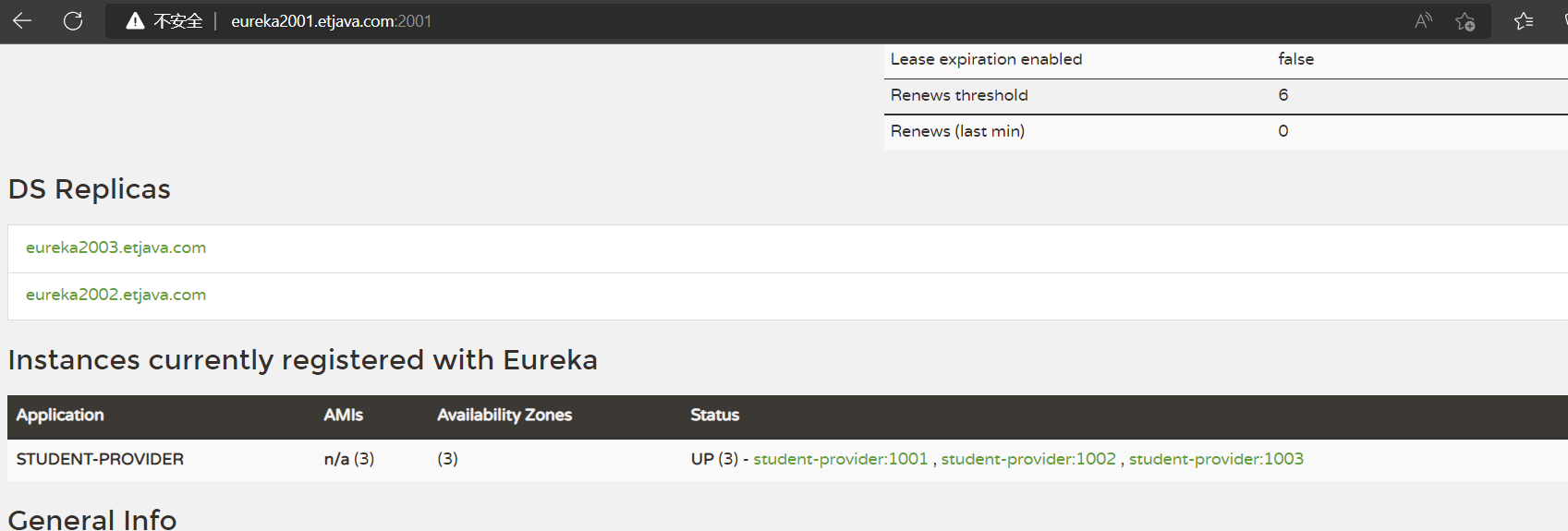
|  |
| --- |
| server:  port: 80  context-path: /  # ribbon 相关配置  eureka:  client:  #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  register-with-eureka: false  service-url:  defaultZone: http://eureka2001.etjava.com:2001/eureka/,  http://eureka2002.etjava.com:2002/eureka/,  http://eureka2003.etjava.com:2003/eureka/  # 单机 需要指定服务提供者的IP+端口  # provider\_host: http://localhost:1001  # 集群版 STUDENT-PROVIDER22 对应的是服务提供者配置文件中的application.name的值  # 这里只能使用中划线 不能使用下划线 提供者那边也只能使用中划线  provider\_host: http://STUDENT-PROVIDER |

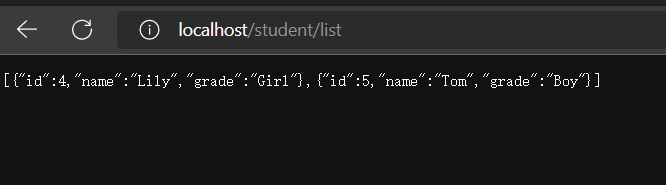
## 测试

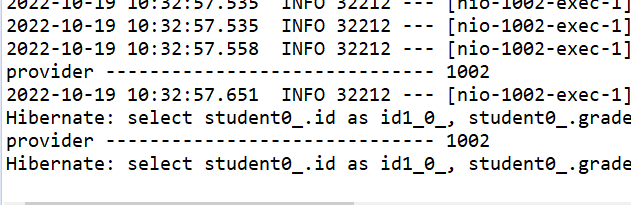
启动三个eureka

启动三个服务提供者

启动feign方式的服务消费者







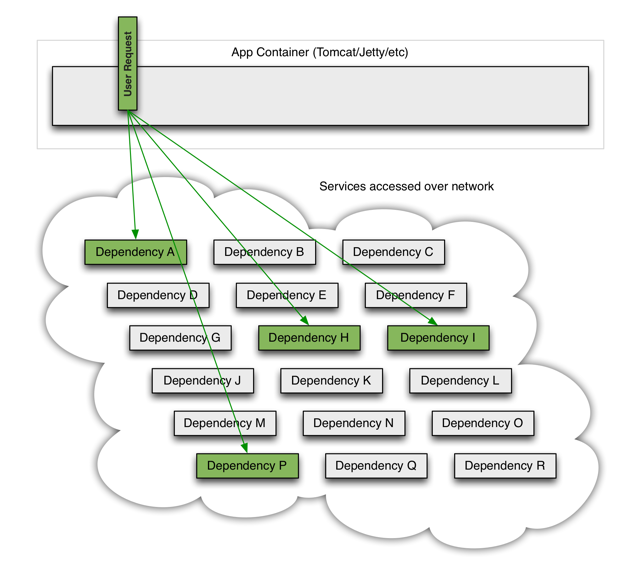
# Hystrix

## 概述

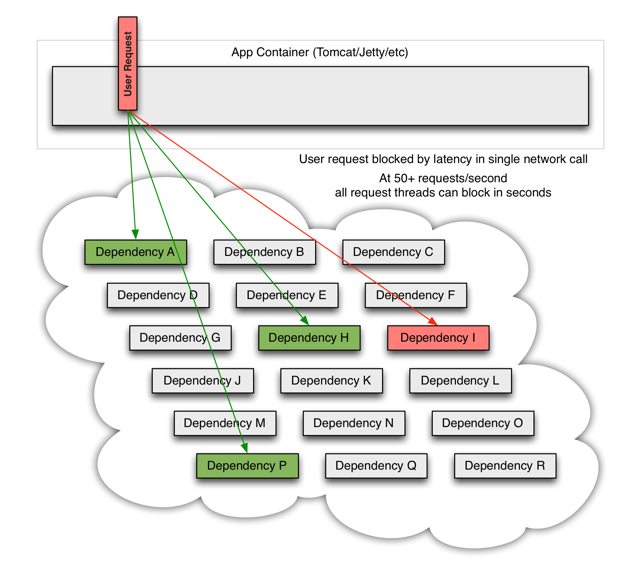
hystrix 是断路器 ，在一个分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，如何能够保证在一个依赖出问题的情况下，不会导致整体服务失败，这个就是Hystrix需要做的事情。Hystrix提供了熔断、隔离、Fallback、cache、监控等功能，能够在一个、或多个依赖同时出现问题时保证系统依然可用。

## 雪崩效应

正常情况下用户请求访问服务

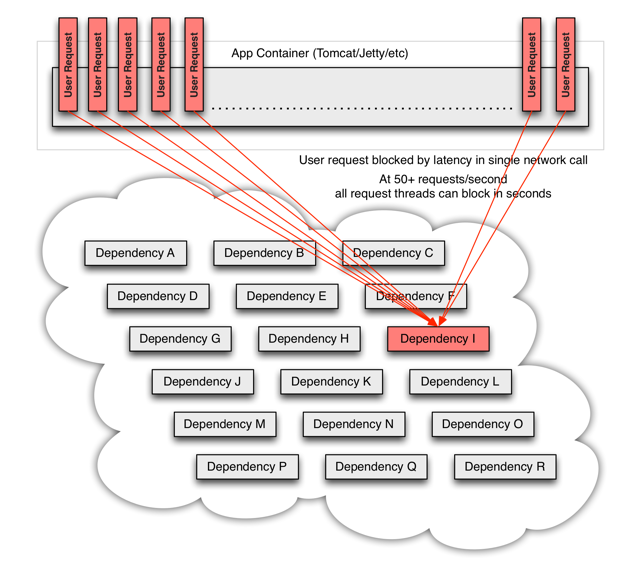


当请求的服务中出现无法访问、异常、超时等问题时（图中的I），那么用户的请求将会被阻塞



如果多个用户的请求中，都存在无法访问的服务，那么他们都将陷入阻塞的状态中 此时就是服务的雪崩效应

hystrix的作用就是用来解决这个问题的(通过服务熔断和降级)



## hystrix熔断和降级

### 操作步骤

1 创建一个新的服务提供者 带有hystrix断路器的

2 添加依赖 依赖与其它服务提供者相同 这里多出hystrix的依赖

3 修改配置文件 将端口及实例名称和端口改为1004，hystrix-1004

4 controller中新增模拟访问雪崩效应的方法

hystrix的熔断机制默认是1秒 方法请求时间只要超出1秒就会触发熔断自我保护机制 然后通过fallbackMethod指定的回调方法返回指定的错误提示

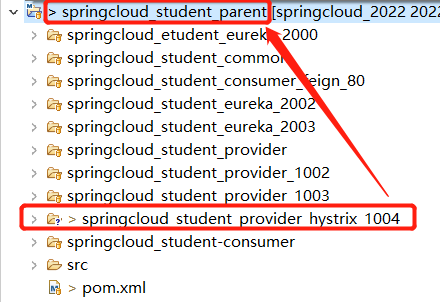
5 修改启动类 添加是开启断路器的注解 @EnableCircuitBreaker

6 修改基于RestTemplate的服务消费者端的controller 添加getInfo方法 来模拟测试访问超时时hystrix的处理

### 新建带有hystrix的服务提供者模块

springcloud\_student\_provider\_hystrix\_1004

在父项目中新建 新建方式与之前相同



### 添加Hystrix及其它相关的依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>    <!-- eureka注册中心依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>    <!-- actuator监控引入 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- Hystrix 断路器依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>    <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### 修改application.yml

修改访问的端口及实例名称

|  |
| --- |
| server:  port: 1004  context-path: /    # 数据源配置  spring:  application:  # 应用名称 这里消费者使用Ribbon在调用时需要根据该名称调用 这里不能使用下环线 否则消费者那边使用同样的名称 调用不到  name: student-provider  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://192.168.199.108:3306/db\_springcloud  username: root  password: Karen@1234  jpa:  hibernate:  ddl-auto: update  show-sql: true  eureka:  instance:  hostname: localhost #eureka客户端主机实例名称  appname: student-provider #客户端服务名  instance-id: student-provider-hystrix:1004 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  # 单机 - 把服务注册到eureka注册中心 这里的defaultZone要和前面服务注册中心的暴露地址一致  # defaultZone: http://localhost:2001/eureka  # 集群  defaultZone: http://eureka2001.etjava.com:2001/eureka,  http://eureka2002.etjava.com:2002/eureka,  http://eureka2002.etjava.com:2002/eureka      info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 王五  联系电话: 123456 |

### 修改启动类

添加@EnableCircuitBreaker 开启断路器

|  |
| --- |
| @EnableCircuitBreaker // 开启熔断保护机制  @EnableEurekaClient  @SpringBootApplication  **public** **class** StudentProviderHystrixApp\_1004 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentProviderHystrixApp\_1004.**class**, args);  }  } |

### controller中新增方法

|  |
| --- |
| /\*\*  \* 服务提供者Controller  \* **@author** etjav  \*  \*/  @RestController  @RequestMapping("/student")  **public** **class** StudentController {  @Resource  **private** StudentService studentService;    /\*\*  \* 添加或者修改学生信息  \* 服务提供者在接收数据时需要使用@RequestBody 否则接收不到数据  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/save")  **public** **boolean** save(@RequestBody Student student){  **try**{  studentService.save(student);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }    /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/list")  **public** List<Student> list(){  System.***out***.println("provider ------------------------------ 1004");  **return** studentService.list();  }    /\*\*  \* 根据id查询学生信息  \* **@return**  \*/  @GetMapping(value="/get/{id}")  **public** Student get(@PathVariable("id") Integer id){  **return** studentService.findById(id);  }    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id){  **try**{  studentService.delete(id);  **return** **true**;  }**catch**(Exception e){  **return** **false**;  }  }    /\*\*  \* 获取信息  \* **@return**  \* **@throws** InterruptedException  \*/  @ResponseBody  @GetMapping(value="/getInfo")  @HystrixCommand(fallbackMethod="getInfoFallback")  **public** Map<String,Object> getInfo() **throws** InterruptedException{  Thread.*sleep*(2000);// 模拟访问超时 hystrix默认1秒  Map<String,Object> map=**new** HashMap<String,Object>();  map.put("code", 200);  map.put("info", "业务数据xxxxx");  **return** map;  }    **public** Map<String,Object> getInfoFallback() **throws** InterruptedException{  Map<String,Object> map=**new** HashMap<String,Object>();  map.put("code", 500);  map.put("info", "系统出错，稍后重试");  **return** map;  }  } |

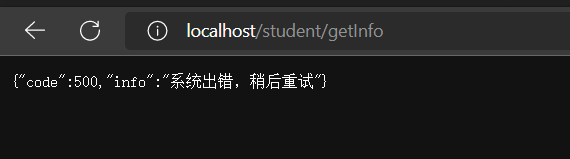
### 调整服务消费者

springcloud\_student-consumer

conller中添加远程调用getInfo的方法



### 测试



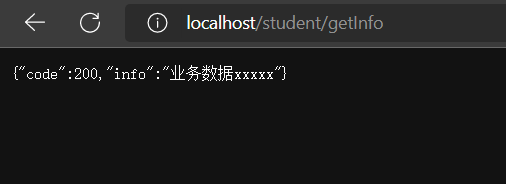
## hystrix默认超时时间配置

服务提供者中进行配置hystrix的超时时间

springcloud\_student\_provider\_hystrix\_1004

在application.yml中添加超时配置即可

|  |
| --- |
| hystrix:  command:  default:  execution:  isolation:  thread:  timeoutInMilliseconds: 3000 |



## hystrix服务监控Dashboard

dashboard是hystrix的一个监控平台 用来实时监控hystrix发起请求的执行情况

### 实现步骤

1 创建带有dashborad的服务提供者模块

2 添加依赖

3 修改端口号及添加启动类

4 启动该模块 检查是否配置成功  
 浏览器访问ip+端口/hystrix

5 启动eureka集群，启动带有hystrix的服务消费者

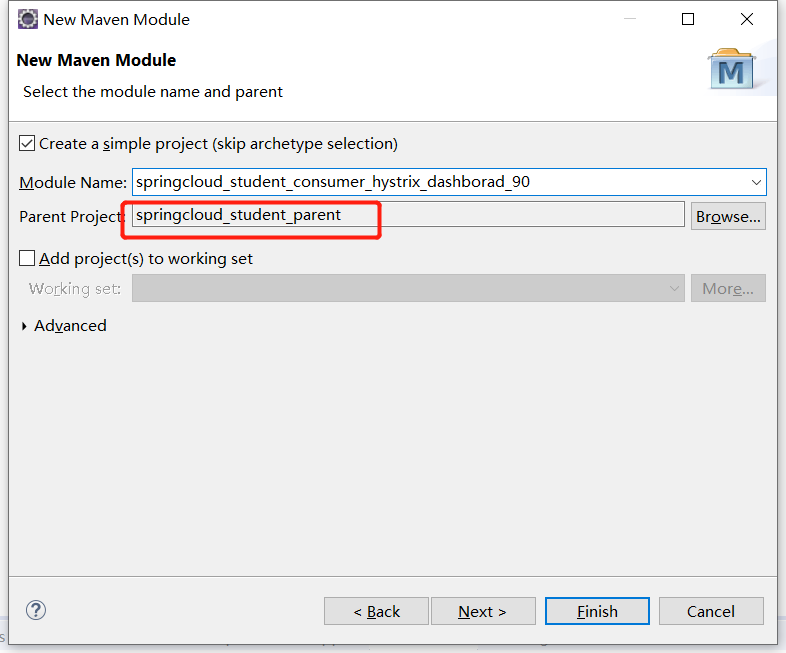
6 浏览器直接通过服务IP加端口来进行访问服务

7 配置dashborad平台图形化显示(在页面直接配置)

8 查看近10秒hystrix执行请求的情况(dashborad默认值)

### 创建hystrix的dashborad监控模块

父项目右键 新建maven module



### 添加依赖

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

### 修改配置文件中的端口

|  |
| --- |
|  |

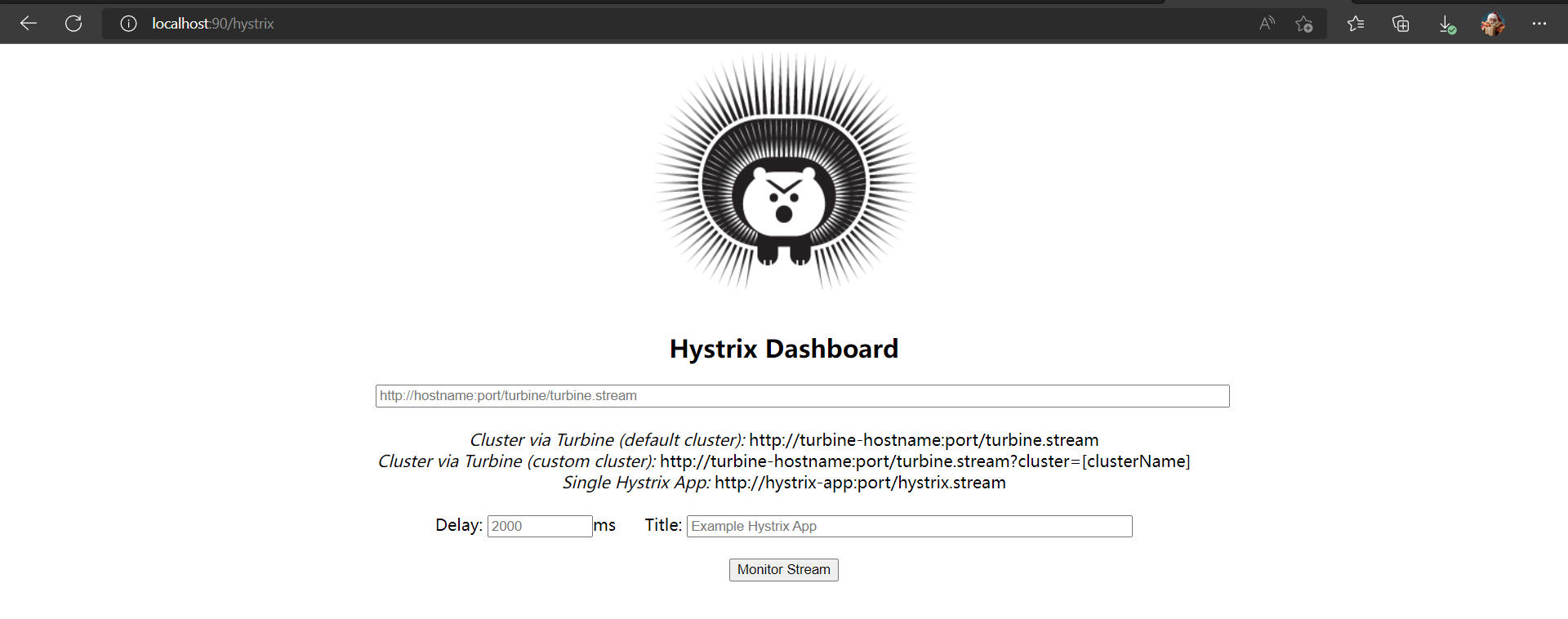
### 添加启动类

启动类中需要通过**@EnableHystrixDashboard**标记开启hystrixDashborad 实时监控平台

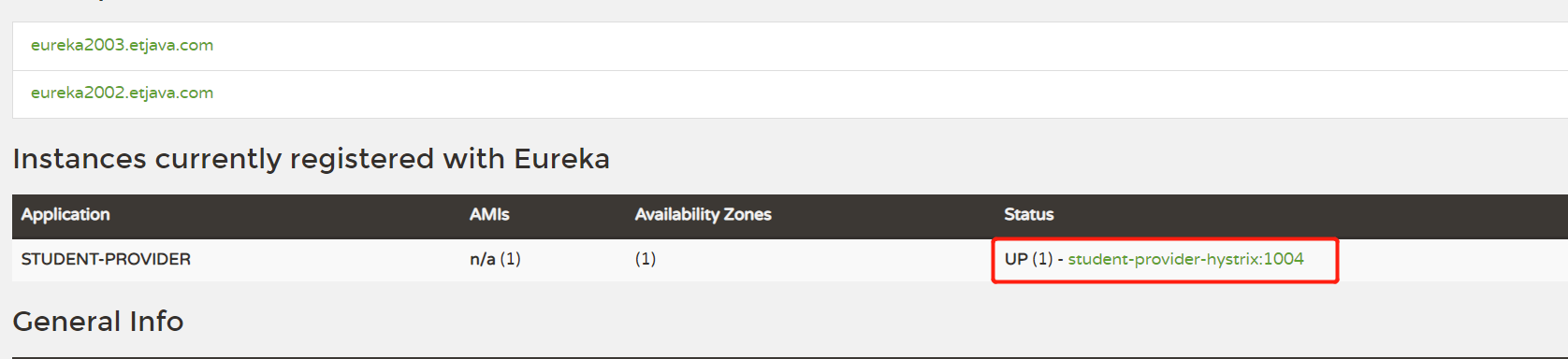
|  |
| --- |
| @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,  HibernateJpaAutoConfiguration.**class**})  @EnableHystrixDashboard  **public** **class** StudentConsumerDashBoardApp\_90 {    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentConsumerDashBoardApp\_90.**class**, args);  }  } |

### 测试

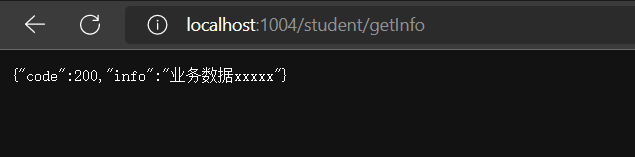
直接启动当前项目进行测试 dashboard是否配置成功



接下来在启动三个eureka及带有hystrix的provider



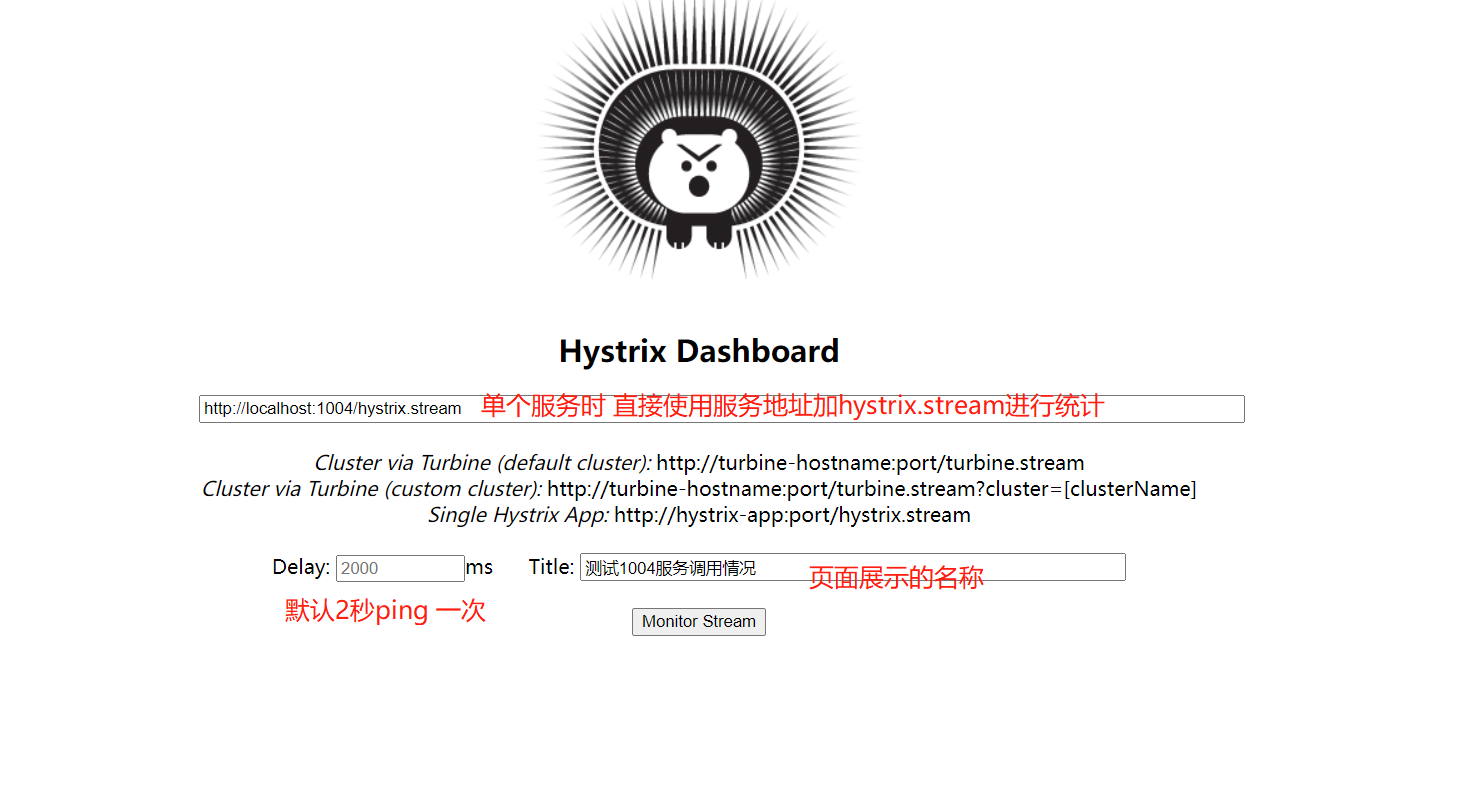
浏览器直接访问1004服务 然后查看控制台实时监控情况

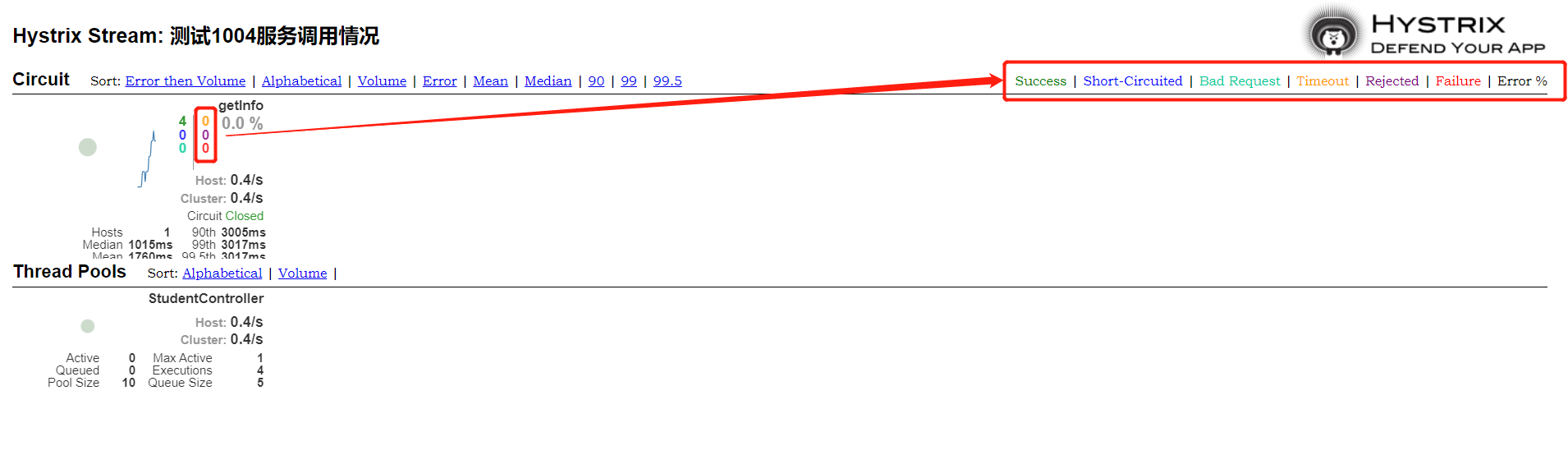


可以直接通过浏览器访问服务请求地址:端口/hystrix.stream



可以配置图形化显示





参数说明



## hystrix集群监控turbine

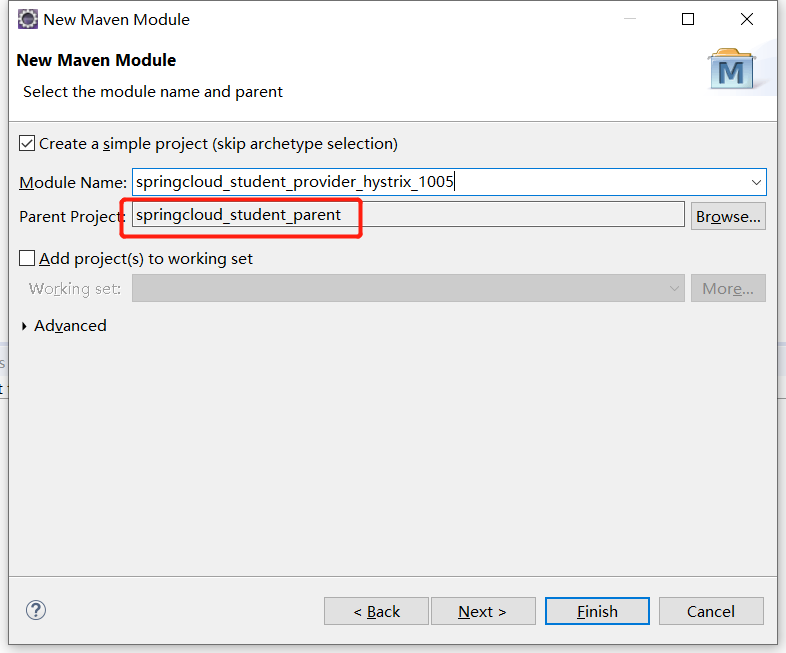
turbine 是基于Dashboard的 实际操作还是Dashboard

实现步骤

### 创建带有hystrix的服务提供者

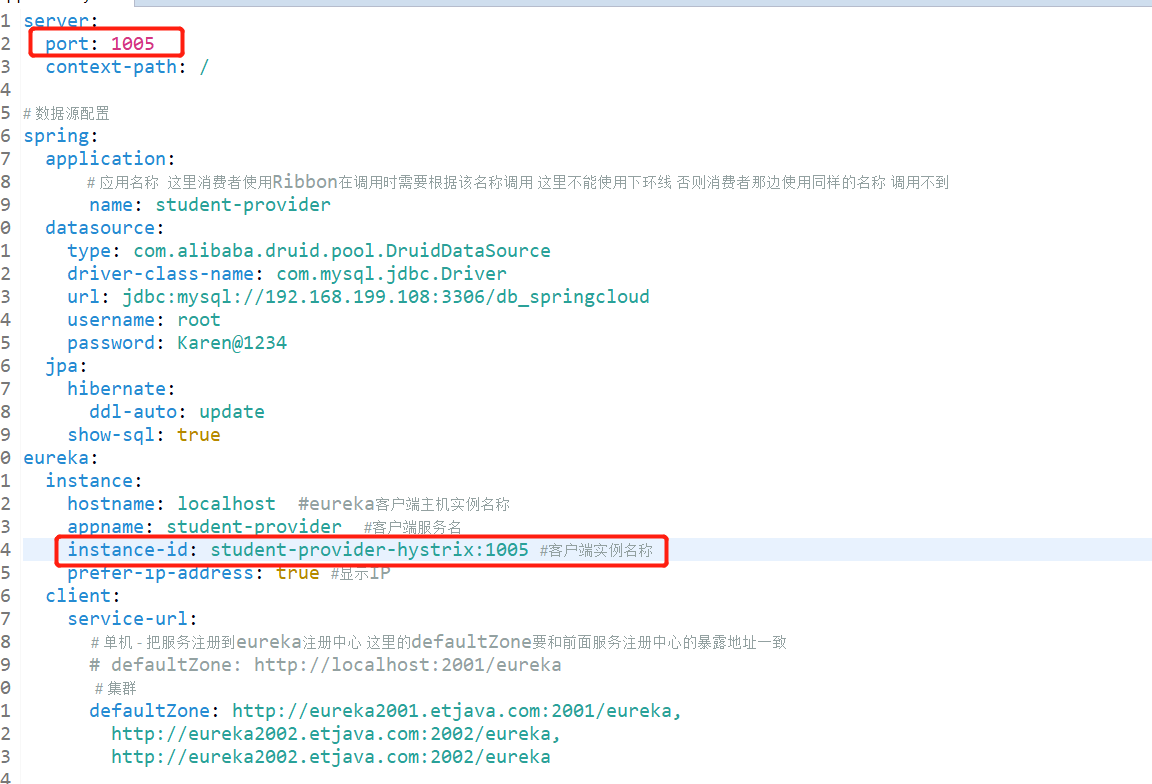
springcloud\_student\_provider\_hystrix\_1005

父项目右键 新建maven module



将provider\_hystrix\_1004中完全复制一份到1005项目

### 修改配置文件



### 修改启动类名称

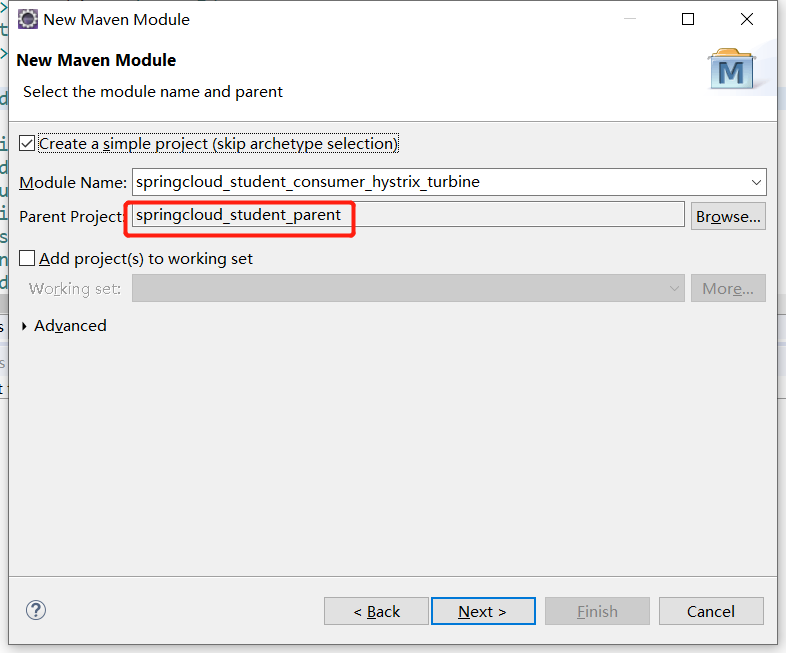


这里我们就有了两个带有熔断机制的服务提供者了

在测试集群之前还需要创建一个带有turbine的服务消费者

### 创建带有turbine的服务消费者

springcloud\_student\_consumer\_hystrix\_turbine



#### 添加依赖

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-turbine</artifactId>

</dependency>

</dependencies>

#### 创建配置文件

虽然是服务消费者 但作为集群的监控(turbine)也是需要注测到eureka中心的

|  |
| --- |
| server:  port: 91  context-path: /    # 这里需要监控的是集群 因此需要配置eureka  eureka:  client:  #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  register-with-eureka: false  service-url:  defaultZone: http://eureka2001.etjava.com:2001/eureka/,  http://eureka2002.etjava.com:2002/eureka/,  http://eureka2003.etjava.com:2003/eureka/    turbine:  app-config: student-provider # 指定要监控的应用名称  #表示集群的名字为default 如果不使用default 则在查看集群是需要带 参数  clusterNameExpression: "'default'"  spring:  application:  name: turbine # 应用名称  provider\_host: http://STUDENT-PROVIDER |

#### 创建启动类

需要添加开启Turbine注解 @EnableTurbine

|  |
| --- |
| import org.springframework.boot.SpringApplication;  import org.springframework.boot.autoconfigure.SpringBootApplication;  import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;  import org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration;  import org.springframework.cloud.netflix.turbine.EnableTurbine;  @SpringBootApplication(exclude={DataSourceAutoConfiguration.class,HibernateJpaAutoConfiguration.class})  @EnableTurbine  public class StudentConsumerTurbineApp\_91 {  public static void main(String[] args) {  SpringApplication.run(StudentConsumerTurbineApp\_91.class, args);  }  } |

### 测试

启动三个eureka注册中心

启动两个带有hystrix的服务提供者

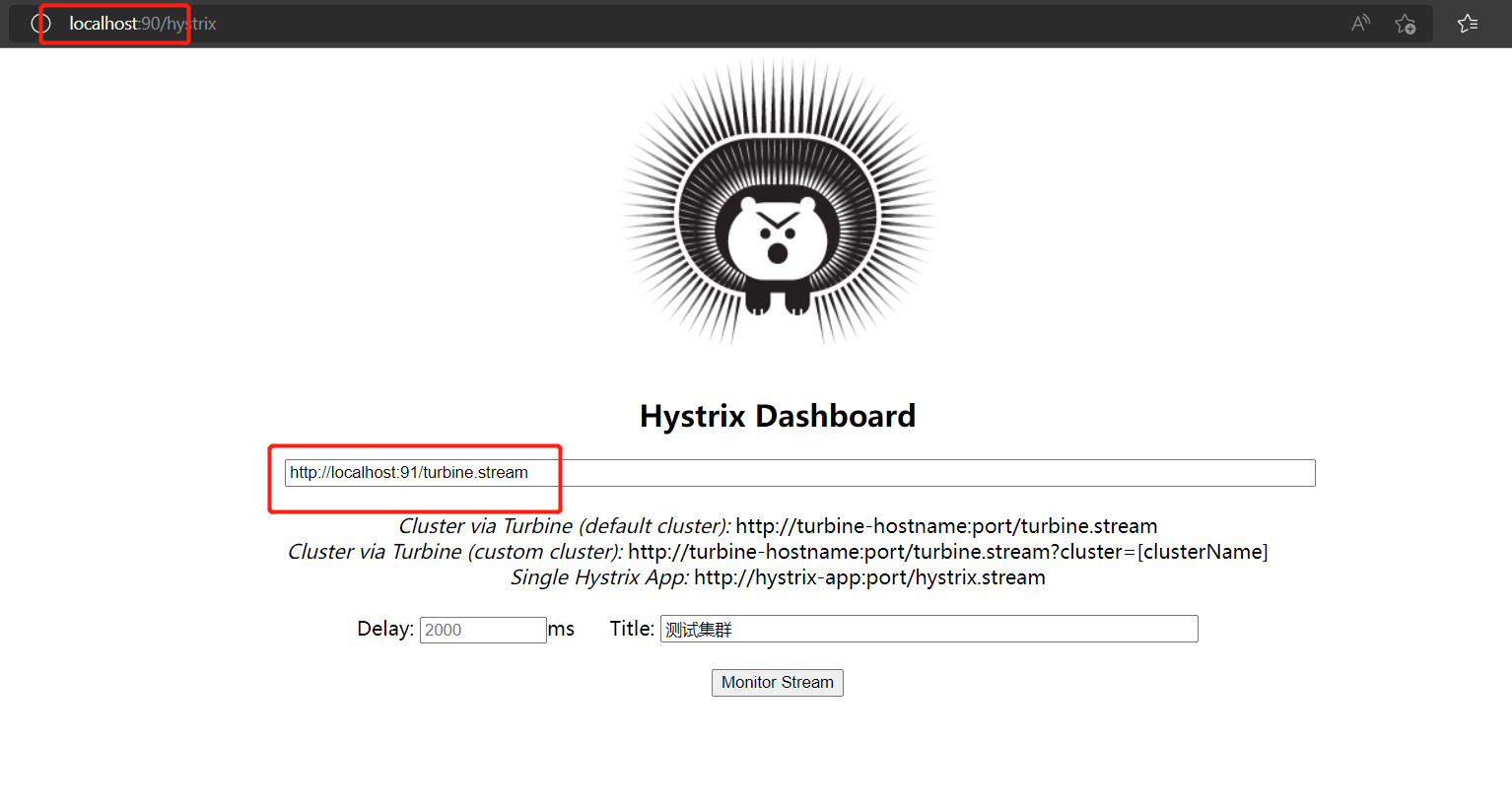
启动带有turbine的服务消费者

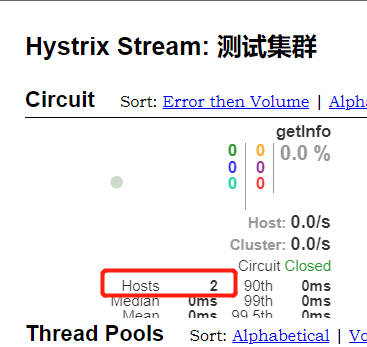
启动带有Dashboard服务消费者(严格讲 不属于消费者端 只是一个监控平台)

启动普通的服务消费者 使用RestTemplate远程访问

注意：监控台还是使用90的 即 Dashborad

91的是作为集群模式监控 最后显示到仪表盘上





## Feign与Hystrix整合实现业务彻底解耦

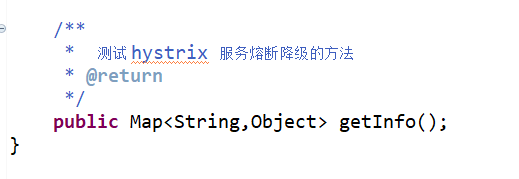
上述案例中我们为了测试hystrix的熔断与服务降级机制 将模拟异常的方法直接放在了provider的controller中 正常情况下业务流程应该是在service中，这里可以通过Feign与Hystrix整合实现业务的解耦

### 配置方式

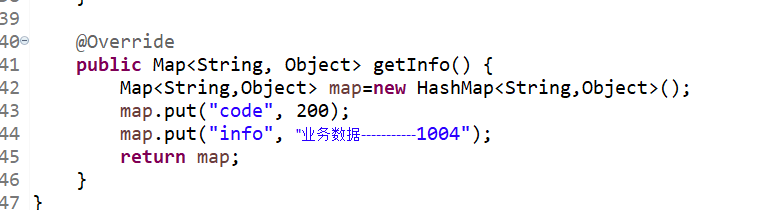
#### 修改服务提供者 - 带有hystrix

springcloud\_student\_provider\_hystrix\_1004 springcloud\_student\_provider\_hystrix\_1005

将里面controller中的getInfo方法提取到service层



impl



#### consumer配置文件中添加hystrix的支持

springcloud\_student\_consumer\_feign\_80



#### consumer的Controller修改

controller中的getInfo方法直接通过service调用即可(前提是先配置好Common模块)

|  |
| --- |
| /\*\*  \* 模拟Hystrix熔断和降级处理  \* **@return**  \*/  @GetMapping(value="/getInfo")  @ResponseBody  **public** Map<String,Object> getInfo(){  **return** studentServiceWithFeign.getInfo();  } |

#### common公共模块修改

service中添加getInfo方法

@FeignClient注解中添加降级处理的回调类

springcloud\_student\_common

|  |
| --- |
| @FeignClient(value="STUDENT-PROVIDER",fallbackFactory=StudentClientFallbackFactory.**class**)  **public** **interface** StudentServiceWithFeign {  /\*\*  \* 根据id查询学生信息  \* **@param** id  \* **@return**  \*/  @GetMapping(value="/student/get/{id}")  **public** Student get(@PathVariable("id") Integer id);    /\*\*  \* 查询学生信息  \* **@return**  \*/  @GetMapping(value="/student/list")  **public** List<Student> list();    /\*\*  \* 添加或者修改学生信息  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/student/save")  **public** **boolean** save(Student student);    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/student/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id);    /\*\*  \* 模拟hystrix熔断降级  \* **@return**  \*/  @GetMapping(value="/student/getInfo")  **public** Map<String,Object> getInfo();    } |

编写降级处理类

实现FallbackFactory接口 通过该接口中的create方法实现对service中所有方法的降级处理（方service中任何一个方法出现异常时都会有对应的降级处理方法）

|  |
| --- |
| /\*\*  \* 解耦服务熔断服务降级 - 配置服务降级  \* 当出现调用异常(超时，网络延迟等)会调用这里实现的方法  \* **@author** etjav  \*  \*/  @Component  **public** **class** StudentClientFallbackFactory **implements** FallbackFactory<StudentServiceWithFeign>{    // 当出现调用异常(超时，网络延迟等)会调用这里实现的方法  @Override  **public** StudentServiceWithFeign create(Throwable cause) {  **return** **new** StudentServiceWithFeign() {  @Override  **public** Student get(Integer id) {  // **TODO** Auto-generated method stub  **return** **null**;  }  @Override  **public** List<Student> list() {  // **TODO** Auto-generated method stub  **return** **null**;  }  @Override  **public** **boolean** save(Student student) {  // **TODO** Auto-generated method stub  **return** **false**;  }  @Override  **public** **boolean** delete(Integer id) {  // **TODO** Auto-generated method stub  **return** **false**;  }  // 当调用getInfo出现异常时 会调用这里的降级方法  @Override  **public** Map<String, Object> getInfo() {  Map<String,Object> map=**new** HashMap<String,Object>();  map.put("code", 500);  map.put("info", "系统出错，稍后重试");  **return** map;  }    };  }  } |

### 测试

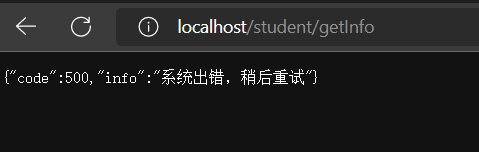
启动三个eureka

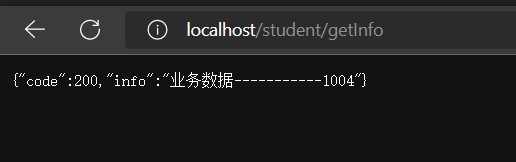
启动两个带有hystrix的服务提供者

启动带有feign的consumer

浏览器中通过对consumer中getInfo请求调用 查看是否实现配置解耦成功 不抛异常出来就表示成功了

两个provider中 getInfo方法一个休眠不足一秒 一个休眠两秒 调用请求会发现有时会返回降级处理的消息，有时则正常返回





## Feign与Hystrix整合之超时时间处理

在进行feign与hystrix整合时 发现之前在服务端配置的hystrix的超时时间不起作用了，这是因为feign的原因导致的

feign的底层封装了ribbon 而ribbon本身也有一个轮询的超时时间 默认也是1秒钟，因此 ribbon的超时时间要大于hystrix的超时时间，否则 hystrix自定义的超时时间毫无意义

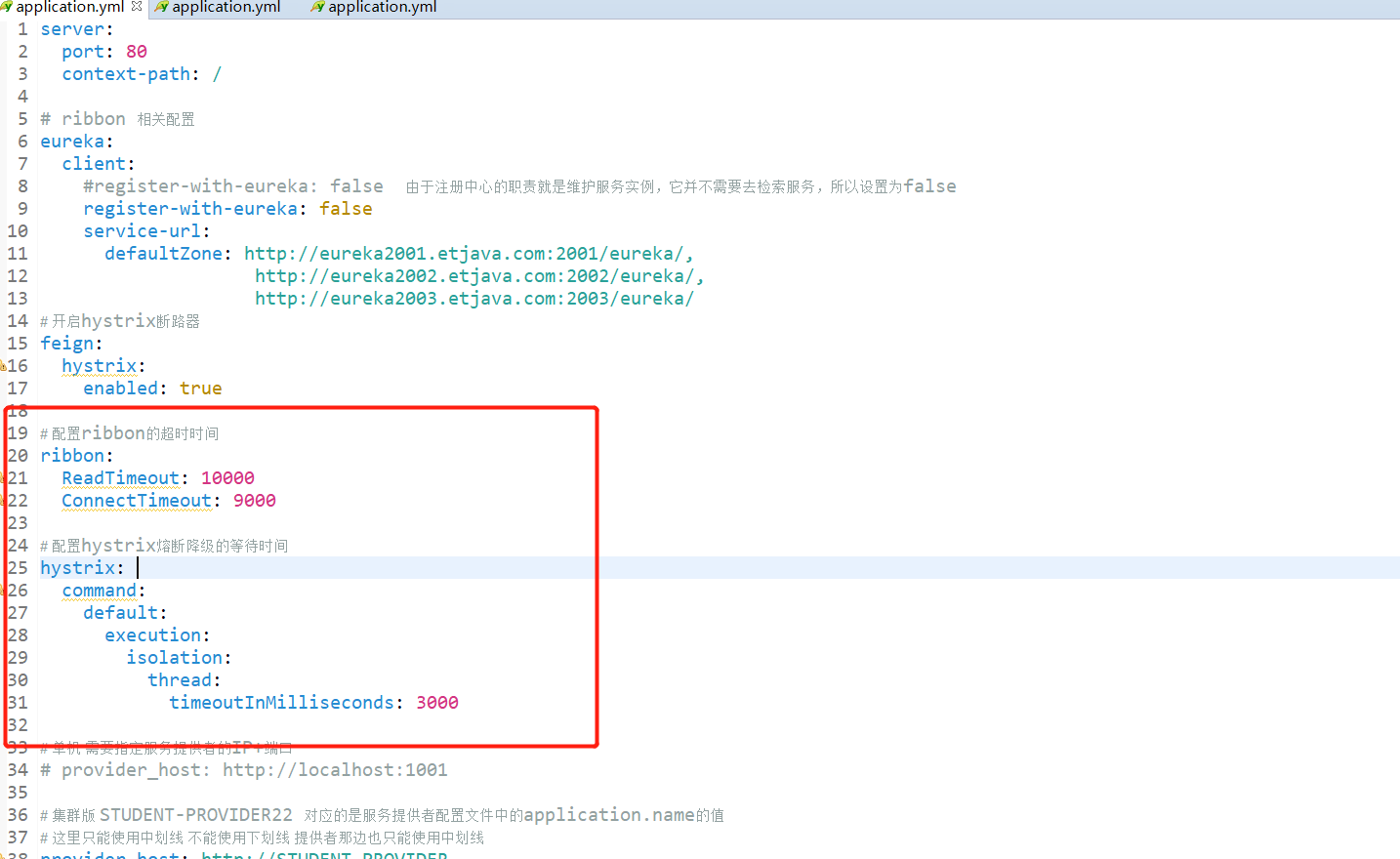
我们只需要将ribbon的默认超时时间调整就可以了。

ribbon的配置是在服务消费者端进行配置的 可以将provider中的hystrix的超时时间及ribbon的超时时间统一配置到服务的消费者中 不需要在从provider中进行配置了

改动的地方有两个 ：

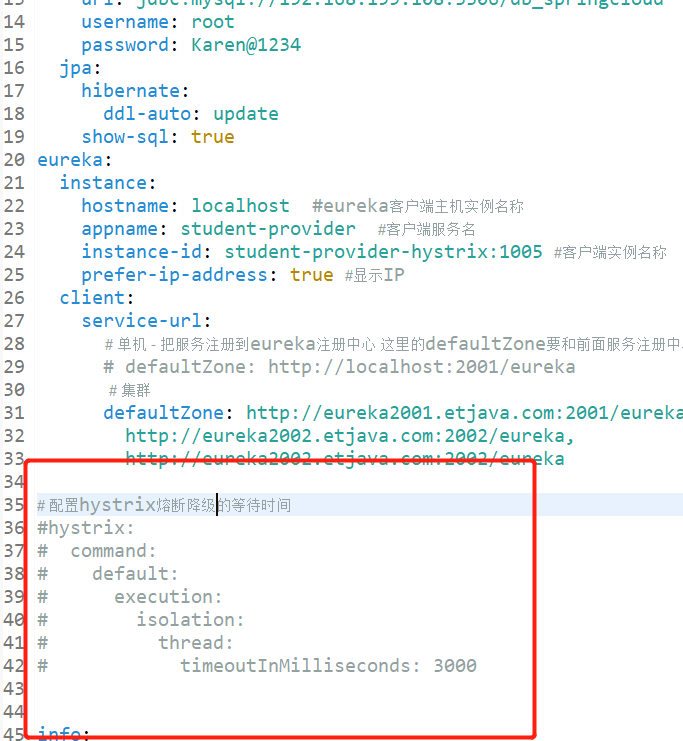
一个是在两个provider的配置文件中移除hystrix超时时间的配置

一个是在consumer的配置文件中添加ribbon和hystrix的超时时间配置



|  |
| --- |
| # 配置ribbon的超时时间  ribbon:  ReadTimeout: 10000  ConnectTimeout: 9000    # 配置hystrix熔断降级的等待时间  hystrix:  command:  default:  execution:  isolation:  thread:  timeoutInMilliseconds: 3000 |

provider中移除掉就可以了



再次测试 发现一切按照预期执行

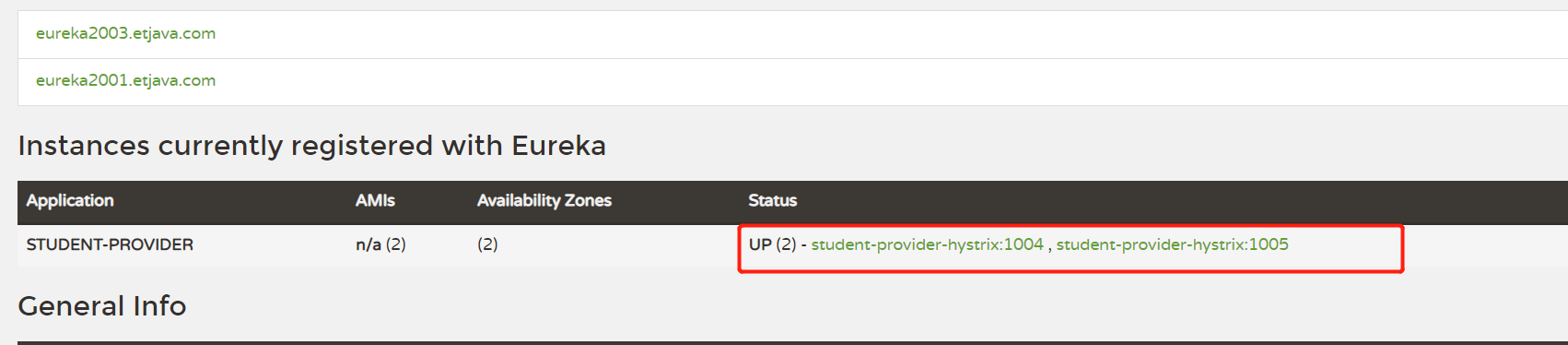
provider的controller 中getInfo方法 一个不足一秒的休眠，一个两秒 但都没达到我们配置的三秒的超时设置 因此这里的测试两个如果都返回正常的业务信息则表示配置成功

测试：

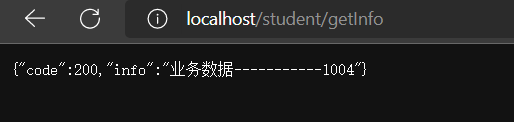
启动三个eureka

启动两个provider 带有hystrix

启动consumer 带feign



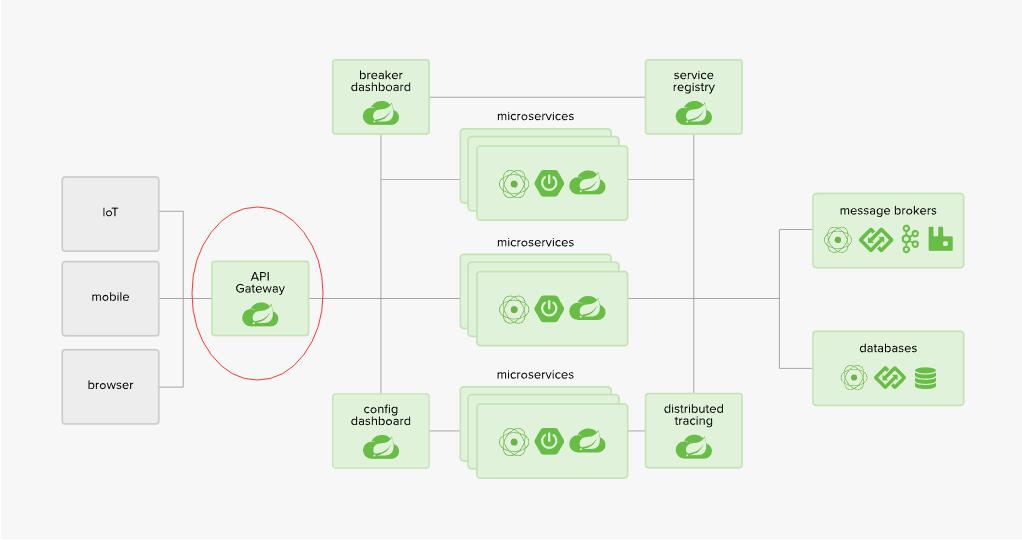




# Zuul 网关路由

## 概述

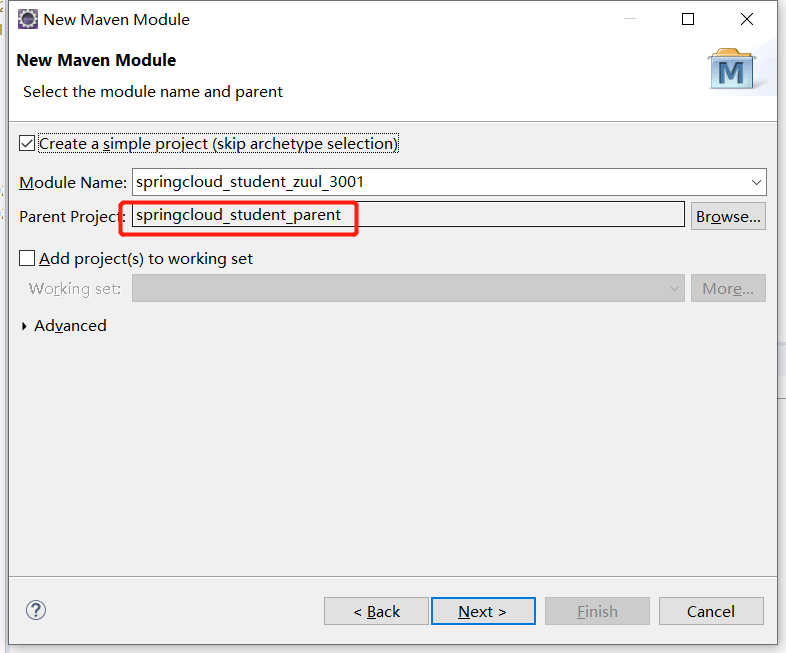
zuul是路由网关 作用主要是对外提供服务接口的时候起到了请求的路由 和 过滤的作用，也因此能够隐藏服务内部的接口细节 从而有利于保护系统的安全性



## zuul路由配置

### 新建路由模块

springcloud\_student\_zuul\_3001



### 添加依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- zuul路由网关 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <!-- actuator监控 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <!-- hystrix容错 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

### 添加springboot的配置文件

application.yml

|  |
| --- |
| server:  port: 3001  context-path: /  # 当前应用名称  spring:  application:  name: student-zuul  # 配置注册中心  eureka:  instance:  instance-id: student-zuul:3001 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  defaultZone: http://eureka2001.etjava.com:2001/eureka,  http://eureka2002.etjava.com:2002/eureka,  http://eureka2002.etjava.com:2002/eureka  info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 王五  联系电话: 123456 |

创建启动类

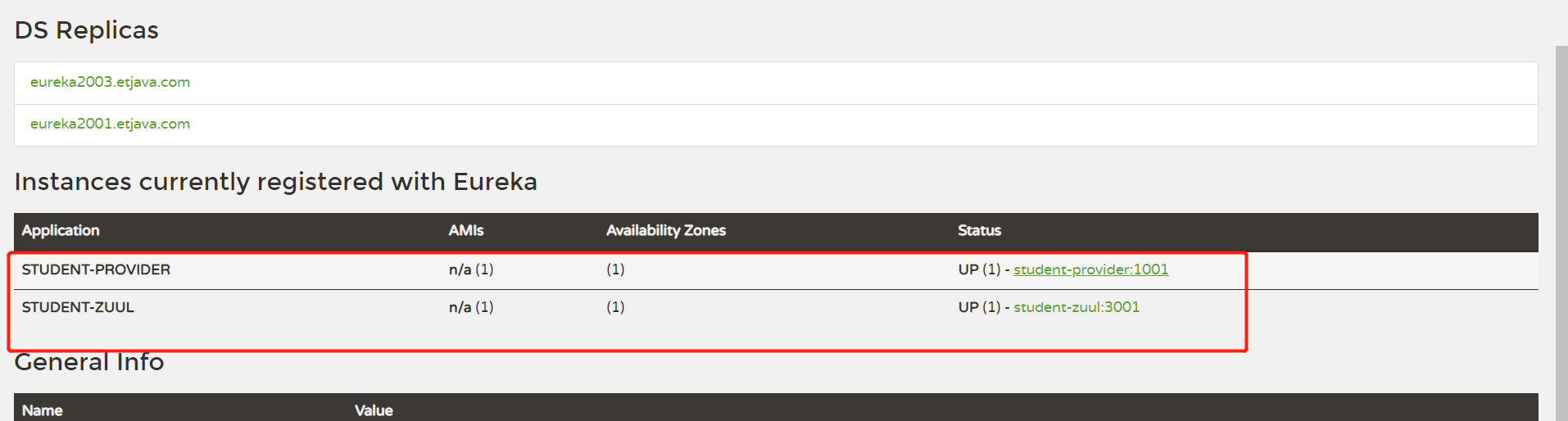
|  |
| --- |
| @EnableZuulProxy // 开启路由网关  @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,HibernateJpaAutoConfiguration.**class**})  **public** **class** ZuulApp\_3001 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ZuulApp\_3001.**class**, args);  }  } |

### 测试

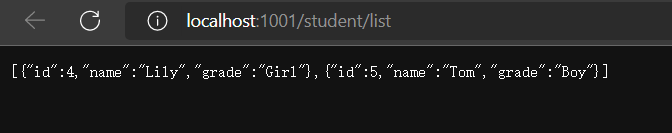
启动三个eureka

启动一个普通的服务提供则 1001

启动网关 zuul



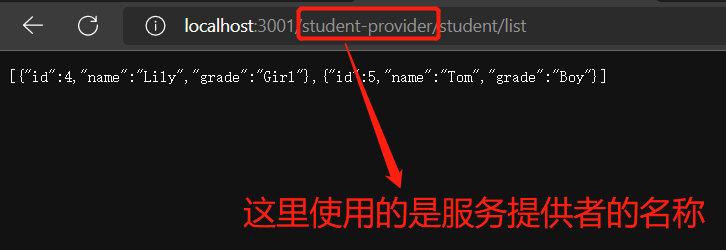
直接通过端口进行访问服务



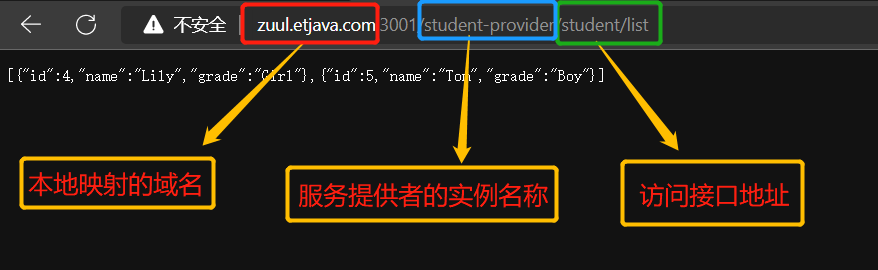
通过网关进行访问服务

访问格式 网关的IP+端口+服务提供者的名称(application.name值)/要访问的接口

例如 127.0.0.1:3001/student-provider/student/list



还可以通过本地映射域名进行访问即 修改hosts文件映射域名

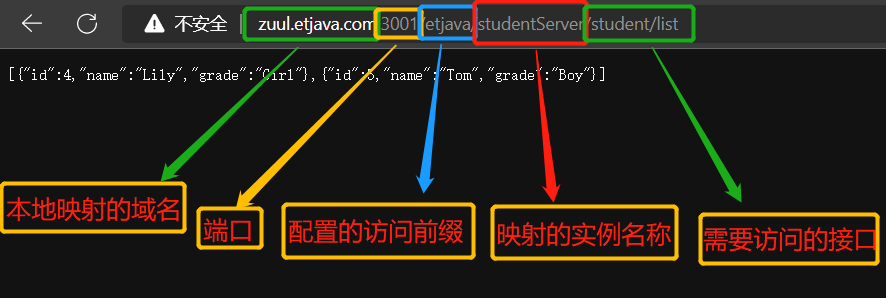


## zuul路由映射规则配置

上边的路由配置很容易暴漏接口细节 所以我们需要再次配置下 屏蔽掉访问接口的细节 提高系统的安全性

在路由模块配置文件中添加如下内容

|  |
| --- |
| # 配置路由规则 屏蔽直接通过应用实例名称直接调用服务  zuul:  # 屏蔽通过服务实例名称进行访问 如果屏蔽单个服务实例 直接写服务实例名称，多个使用星号表示  # 只屏蔽某一个服务 ignored-services: "student-provider"  # 屏蔽某一个服务后访问时404代替  # 屏蔽多个 即 ignored-services: "\*" 会自动映射成指定的path访问路径 不会使用404  ignored-services: "\*"  prefix: /etjava # 指定请求的前缀 如果指定了访问时的前缀 在访问时必须带上 否则404  routes:  # studentServer是随意取的 但后边的serviceId及path是固定写法  # serviceId 指定要进行映射的服务名称 path是映射成的访问路径  studentServer.serviceId: student-provider # 服务提供者的实例名称  studentServer.path: /studentServer/\*\* # 映射后的请求服务名称 隐藏原有的服务实例名称 |



## Zuul 请求过滤配置

在请求网关时可以进行身份验证配置 保证系统的安全性

### 创建请求过滤器

在zuul模块中添加请求过滤

创建一个AccessFilter请求过滤器 需要继承并实现ZuulFilter过滤器

|  |
| --- |
| /\*\*  \* zuul 请求过滤  \* **@author** etjav  \*  \*/  **public** **class** AccessFilter **extends** ZuulFilter {  // 该过滤器是否要被执行  @Override  **public** **boolean** shouldFilter() {  // **TODO** Auto-generated method stub  **return** **true**;  }  // 过滤器的具体执行逻辑  @Override  **public** Object run() **throws** ZuulException {  RequestContext ctx = RequestContext.*getCurrentContext*();  HttpServletRequest request = ctx.getRequest();  String parameter = request.getParameter("accessToken");  System.***out***.println(request.getRequestURL().toString()+" 请求访问");  **if**(parameter==**null**) {  System.***out***.println("accessToken为空");  ctx.setSendZuulResponse(**false**);  ctx.setResponseStatusCode(401);  ctx.setResponseBody("{\"result\":\"accessToken is empty!\"}");  }**else** {  System.***out***.println(request.getRequestURL().toString()+" 请求成功");  }    **return** **null**;  }  // 请求过滤的类型 pre为请求之前，post 表示使用post请求  @Override  **public** String filterType() {  // **TODO** Auto-generated method stub  **return** "pre";  }  // 过滤器的执行顺序  @Override  **public** **int** filterOrder() {  **return** 0;  }  } |

### 创建ZuulConfig

该config是用来配置请求过滤器生效的

|  |
| --- |
| /\*\*  \* zuul 配置  \* **@author** etjav  \*  \*/  @Configuration  **public** **class** ZuulConfig {  /\*\*  \* 将AccessFilter配置成Bean 就可以在每次访问时执行了  \* **@return**  \*/  @Bean  **public** AccessFilter accessFilter() {  **return** **new** AccessFilter();  }  } |

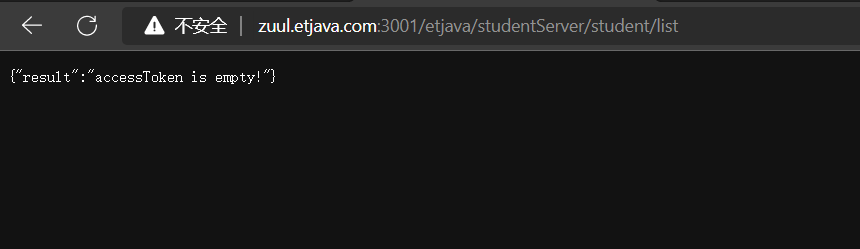
测试：

启动三个eureka

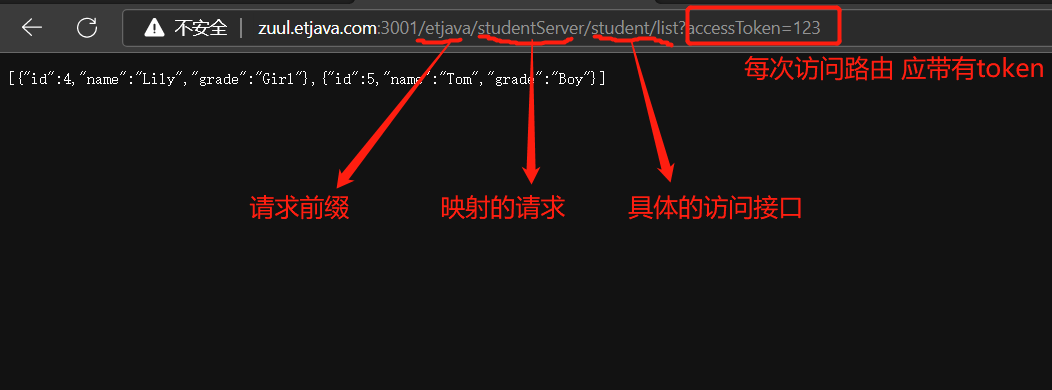
启动一个普通的provider

启动zuul路由网关

不带访问token时



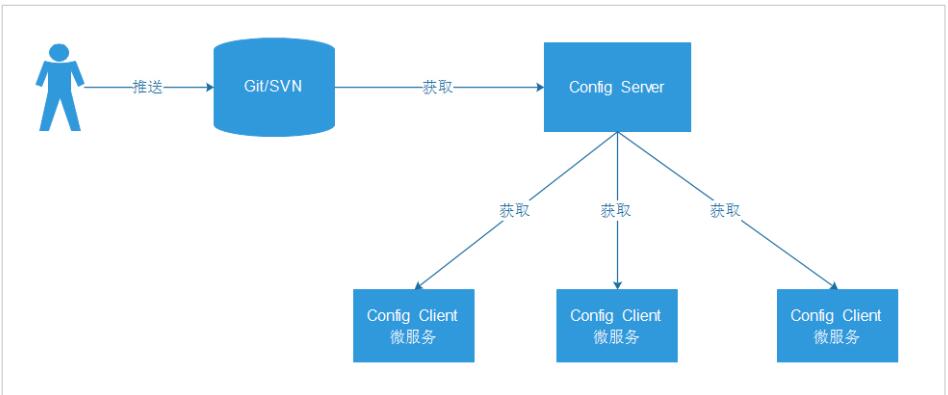
带有访问token时



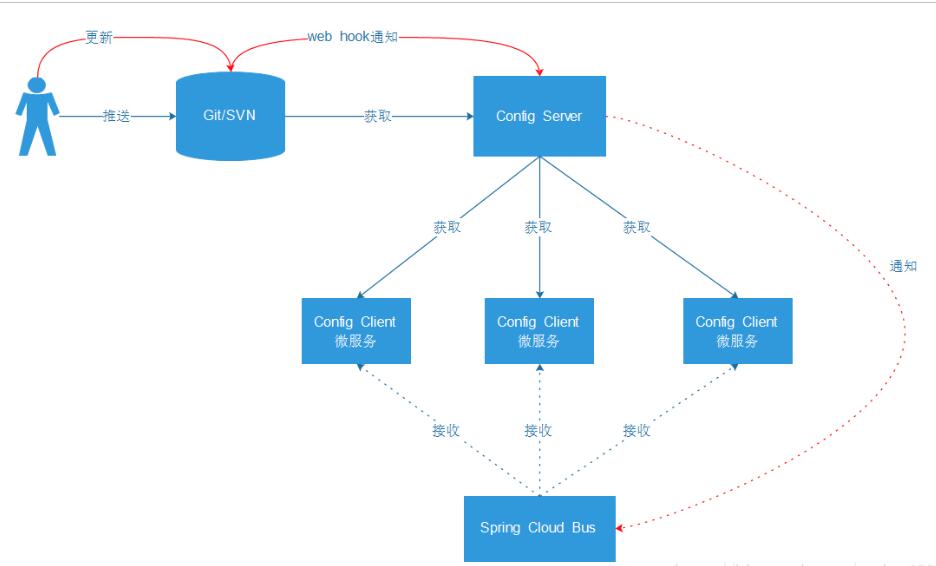
# SpringCloud Config

## 概述

Spring Cloud Config 是 Spring Cloud 团队创建的一个全新项目，用来为分布式系统中的基础设施和微服务应用提供集中化的**外部配置**支持，它分为服务端与客户端两个部分。其中服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置仓库并为客户端提供获取配置信息、加密 / 解密信息等访问接口；而客户端则是微服务架构中的各个微服务应用或基础设施，它们通过指定的配置中心来管理应用资源与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。Spring Cloud Config 实现了对服务端和客户端中环境变量和属性配置的抽象映射，所以它除了适用于 Spring 构建的应用程序之外，也可以在任何其他语言运行的应用程序中使用。由于 Spring Cloud Config 实现的配置中心默认采用 Git 来存储配置信息，所以使用 Spring Cloud Config 构建的配置服务器，天然就支持对微服务应用配置信息的版本管理，并且可以通过 Git 客户端工具来方便的管理和访问配置内容。当然它也提供了对其他存储方式的支持，比如：GIT仓库、SVN 仓库、本地化文件系统。



如果用上SpringCloud Bus，能实现无需重启，自动感知配置变化以及应用新配置

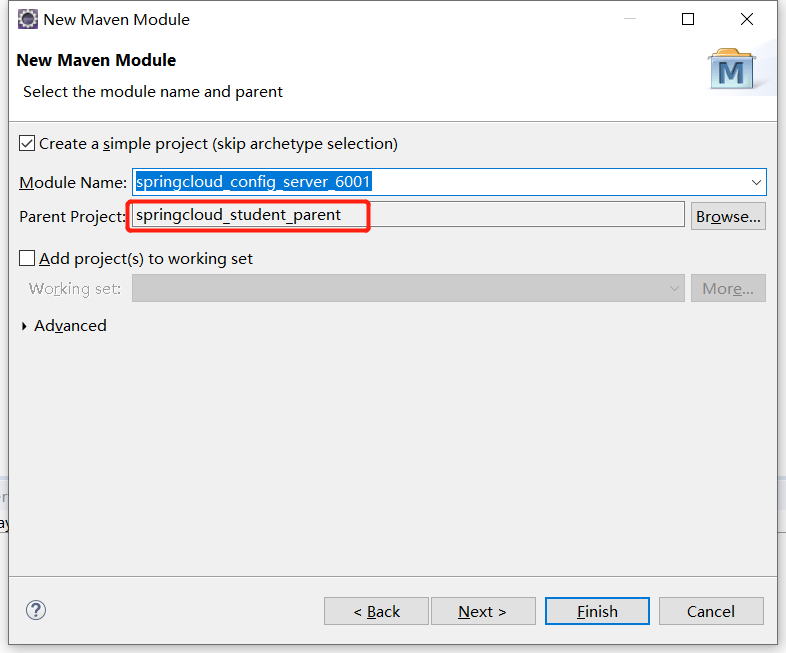


## 搭建config server

### 新建config server模块

springcloud\_config\_server\_6001

选中父项目 右键新建maven module



### 添加依赖

|  |
| --- |
| <dependencies>  <!-- spring cloud config 依赖 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-config-server</artifactId>  </dependency>  </dependencies> |

### 创建启动类

@EnableConfidServer 开启configServer

|  |
| --- |
| @SpringBootApplication  @EnableConfigServer // 开启configServer  **public** **class** SprngCloudConfigApp\_6001 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SprngCloudConfigApp\_6001.**class**, args);  }  } |

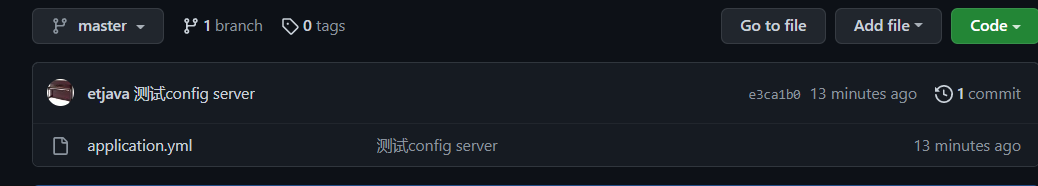
### 添加配置文件

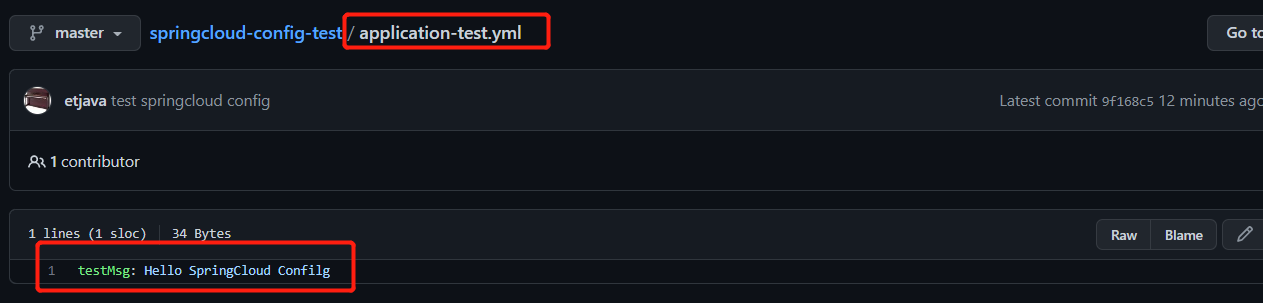
会根据配置的configserver地址去github读取文件

|  |
| --- |
| server:  port: 6001  context-path: /    spring:  application:  name: student-config  # 配置springcloud的config server  cloud:  config:  server:  git:  # uri指向的是git仓库中的地址 注意不能带有后缀 即 .git  uri: https://github.com/etjava/springcloud-config-test |

### 测试

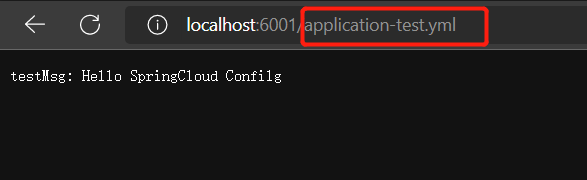
首先在github中添加一个application.xxx.yml的配置文件

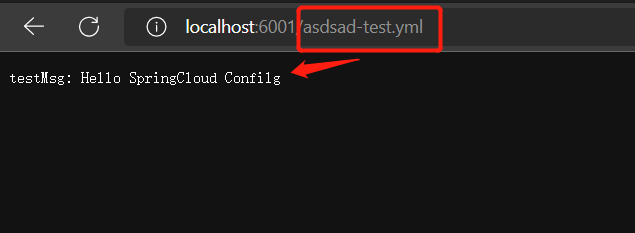




然后启动config-server模块

浏览器访问时后缀带上application-asdsads.yml



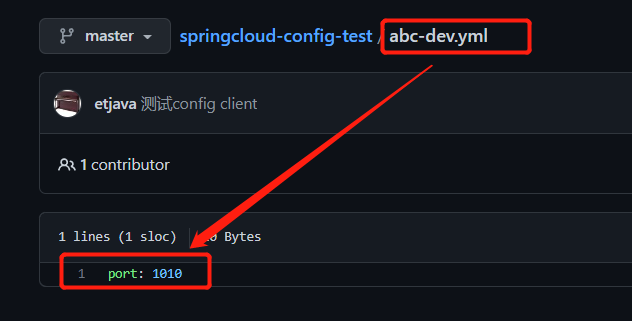


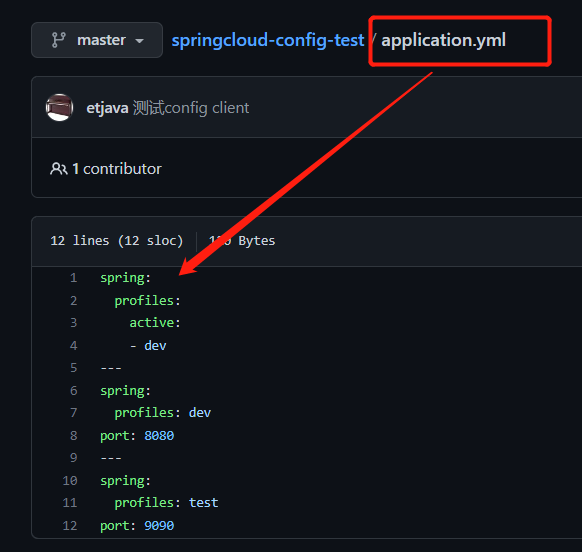
## 搭建config client

### 准备测试数据

在github远程库分别添加如下文件及对应内容



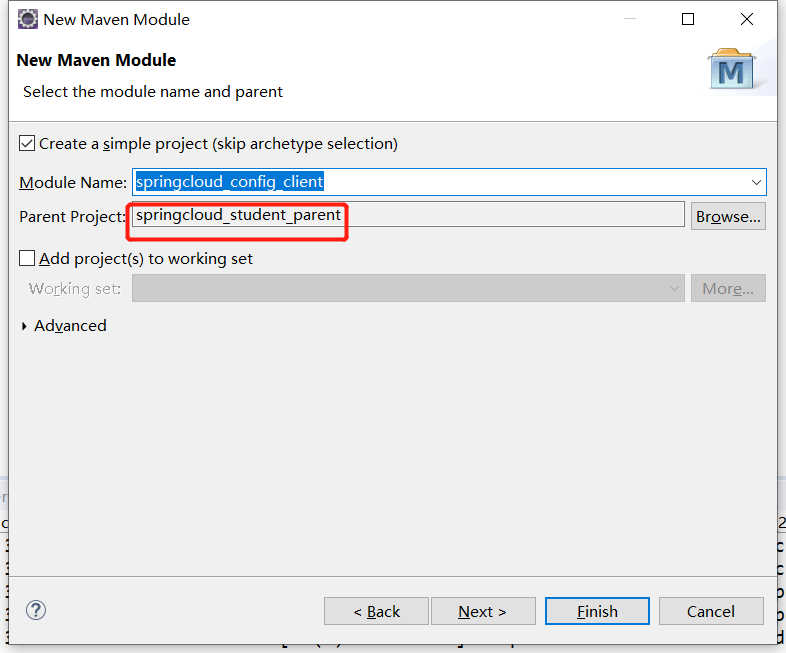
 



### 新建ConfigClient模块

选中父项目 右键新建maven module

springcloud\_config\_client



添加依赖

注意：服务端引入的是spring-cloud-config-server

客户端引入的是spring-cloud-starter-config

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies> |

### 启动时调用config server

在springboot中优先级最高的是bootstrap.yml配置文件 因此需要在bootstrap.yml中进行配置启动时加载的信息

application.yml也是需要用到的

bootstrap.yml

|  |
| --- |
| spring:  application:  name: application-dev  cloud:  config:  # name和profile 分别对应文件的abc-dev.yml  name: abc  uri: http://localhost:6001 # 配置configserver地址  profile: dev # 级别  label: master # 分支 git中 默认master  fail-fast: true |

application.yml

|  |
| --- |
| server:  port: 7001  context-path: / |

### 创建controller

动态获取外部配置文件中的数据

|  |
| --- |
| @RestController  **public** **class** ConfigClientController {  @Value("${port}")  **private** String port;  // 模拟获取外部配置文件中的信息  @GetMapping("/getPort")  **public** String getPort() {  **return** port;  }  **public** **void** setPort(String port) {  **this**.port = port;  }  } |

### 创建启动类

|  |
| --- |
| @SpringBootApplication  **public** **class** SpringCloudConfigClientApp\_7001 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(SpringCloudConfigClientApp\_7001.**class**, args);  }  } |

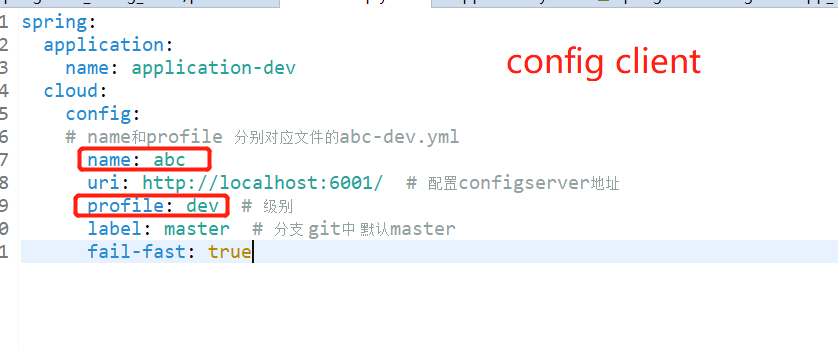
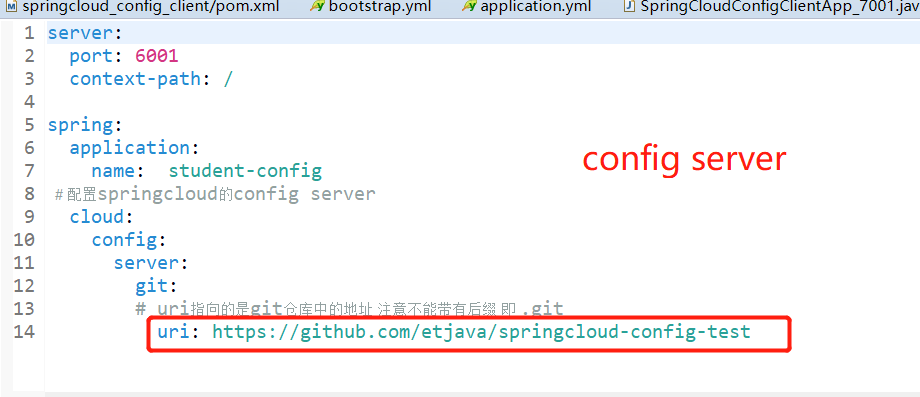
### 测试

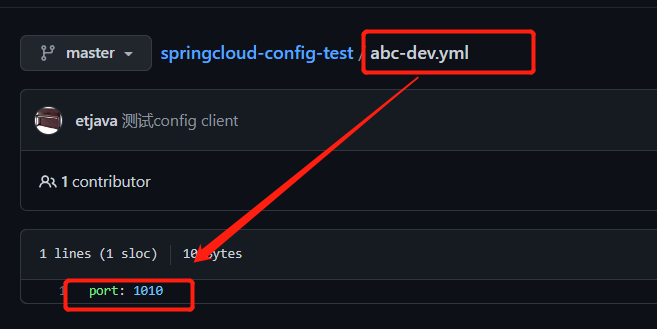
先启动configServer 在启动configClient

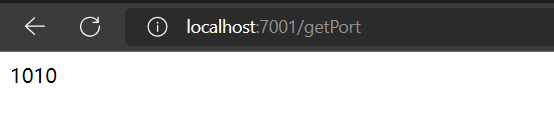
说明：启动时会根据cloud.confg的配置去加载configServer，而configServer去github读取了指定的文件

图示：

启动时



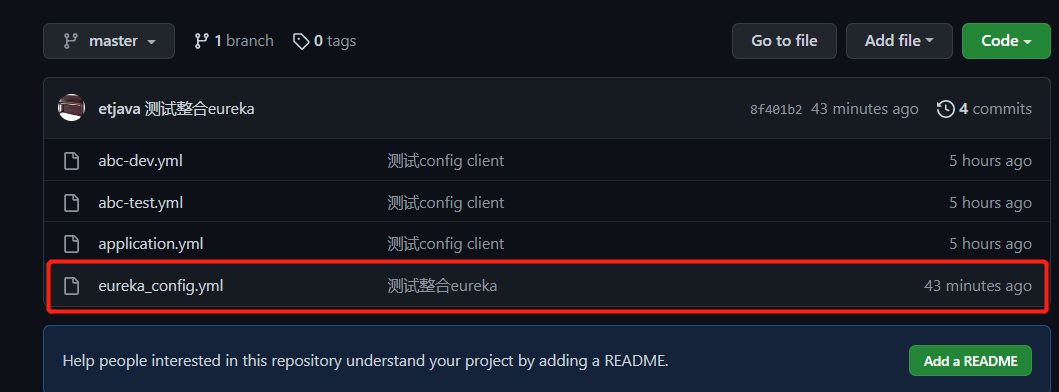


# Eureka 整合SpringCloud Config

将eureka的配置文件存放到github中，在项目启动时动态去加载外部的配置文件 然后在进行服务提供者的注册

## 创建外部文件

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 2004  context-path: /  spring:  profiles: dev  eureka:  instance:  hostname: localhost  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以也设置为false  service-url:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka注册中心交互的地址，查询服务和注册服务用到 #设置与Eureka注册中心交互的地址，查询服务和注册服务用到  ---  server:  port: 2004  context-path: /  spring:  profiles: test  eureka:  instance:  hostname: localhost  client:  register-with-eureka: false #false 由于该应用为注册中心，所以设置为false,代表不向注册中心注册自己。  fetch-registry: false #false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以也设置为false  service-url:  defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与Eureka注册中心交互的地址，查询服务和注册服务用到 #设置与Eureka注册中心交互的地址，查询服务和注册服务用到 |



## 创建整合的模块

springcloud\_student\_eureka\_server\_config\_2004

父项目右键新建maven module

## 添加依赖

## bootstrap.yml

该配置文件优先级高于application.yml

|  |
| --- |
| spring:  application:  name: microservice-eureka-server-config  cloud:  config:  name: eureka\_config  uri: http://localhost:6001 # 配置configserver地址  profile: dev # 级别  label: master # 分支 git中 默认master |

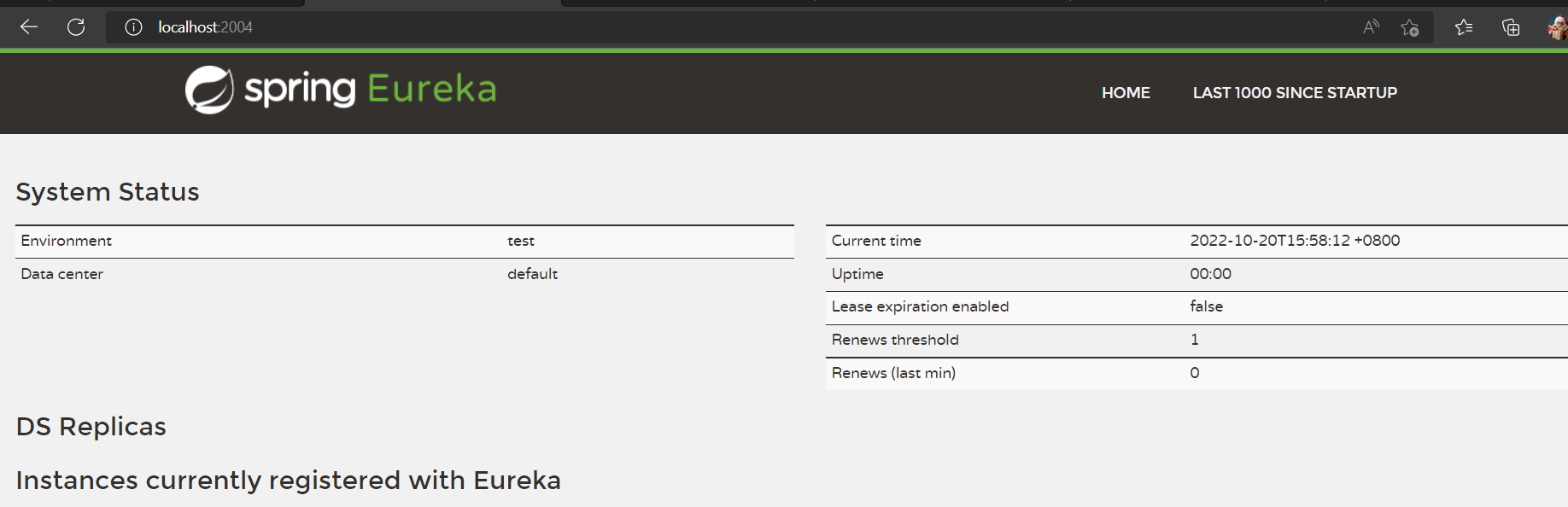
## application.yml

|  |
| --- |
| spring:  application:  name: eureka-server-config |

## 创建启动类

|  |
| --- |
| @SpringBootApplication  @EnableEurekaServer  **public** **class** EurekaApplication\_2004 {    **public** **static** **void** main(String[] args) {  SpringApplication.*run*(EurekaApplication\_2004.**class**, args);  }  } |

测试是否配置成功



# 服务提供者整合SpringCloud Config

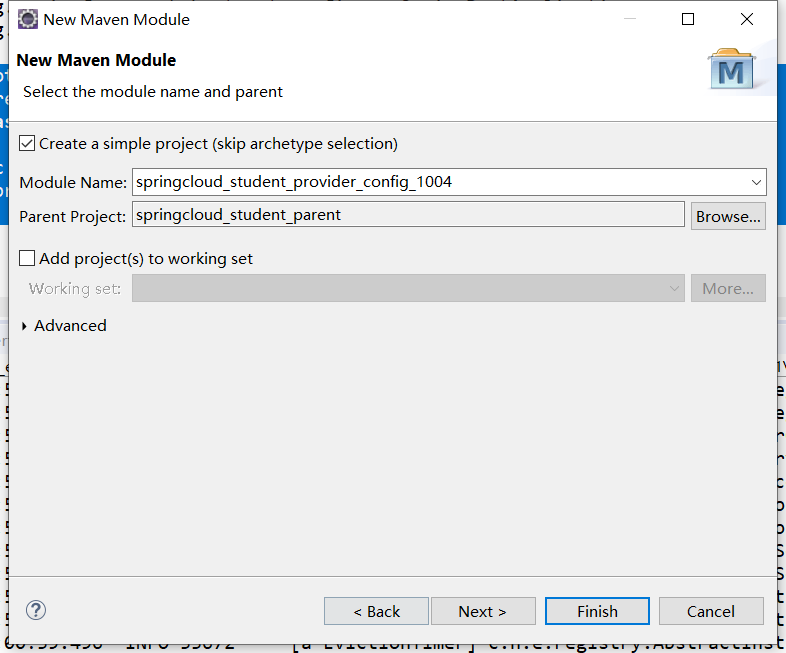
上边整合了注册中心，这里再把服务提供者整合进来 就可以实现配置文件单独管理了

## 新建provider\_config.yml

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 1004  context-path: /  # 数据源配置  spring:  profiles: dev  application:  name: microservice-student  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://192.168.199.108:3306/db\_springcloud  username: root  password: Karen@1234  jpa:  hibernate:  ddl-auto: update  show-sql: true  eureka:  instance:  hostname: localhost #eureka客户端主机实例名称  appname: microservice-student #客户端服务名  instance-id: microservice-student:1004 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  defaultZone: http://localhost:2004/eureka #把服务注册到eureka注册中心  info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 张三  联系电话: 123456  ---  server:  port: 1004  context-path: /  # 数据源配置  spring:  profiles: test  application:  name: microservice-student  datasource:  type: com.alibaba.druid.pool.DruidDataSource  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://localhost:3306/db\_springcloud2  username: root  password: 123456  jpa:  hibernate:  ddl-auto: update  show-sql: true  eureka:  instance:  hostname: localhost #eureka客户端主机实例名称  appname: microservice-student #客户端服务名  instance-id: microservice-student:1004 #客户端实例名称  prefer-ip-address: true #显示IP  client:  service-url:  defaultZone: http://localhost:2004/eureka #把服务注册到eureka注册中心  info:  groupId: $project.groupId$  artifactId: $project.artifactId$  version: $project.version$  负责人: 张三  联系电话: 123456 |

## 新建服务提供者

springcloud\_student\_provider\_config\_1004



## 添加依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>com.etjava</groupId>  <artifactId>springcloud\_student\_common</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <!-- actuator监控引入 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <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-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  </dependency>  <!-- 修改后立即生效，热部署 -->  <dependency>  <groupId>org.springframework</groupId>  <artifactId>springloaded</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  </dependency>  </dependencies> |

## bootstrap.yml

|  |
| --- |
| spring:  application:  name: student-provider-config  cloud:  config:  name: provider\_config  uri: http://localhost:6001 # 配置configserver地址  profile: dev # 级别  label: master # 分支 git中 默认master |

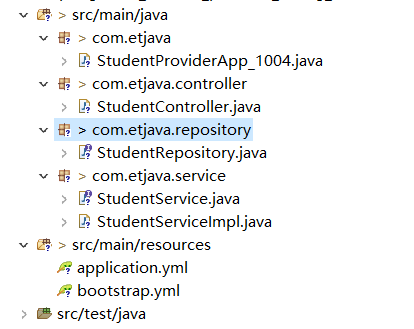
## application.yml

|  |
| --- |
| spring:  application:  name: student-provider-config |

## 复制其它类

在其它provider中复制一份其它类到当前的模块中

例如 启动类，repository，service,controller



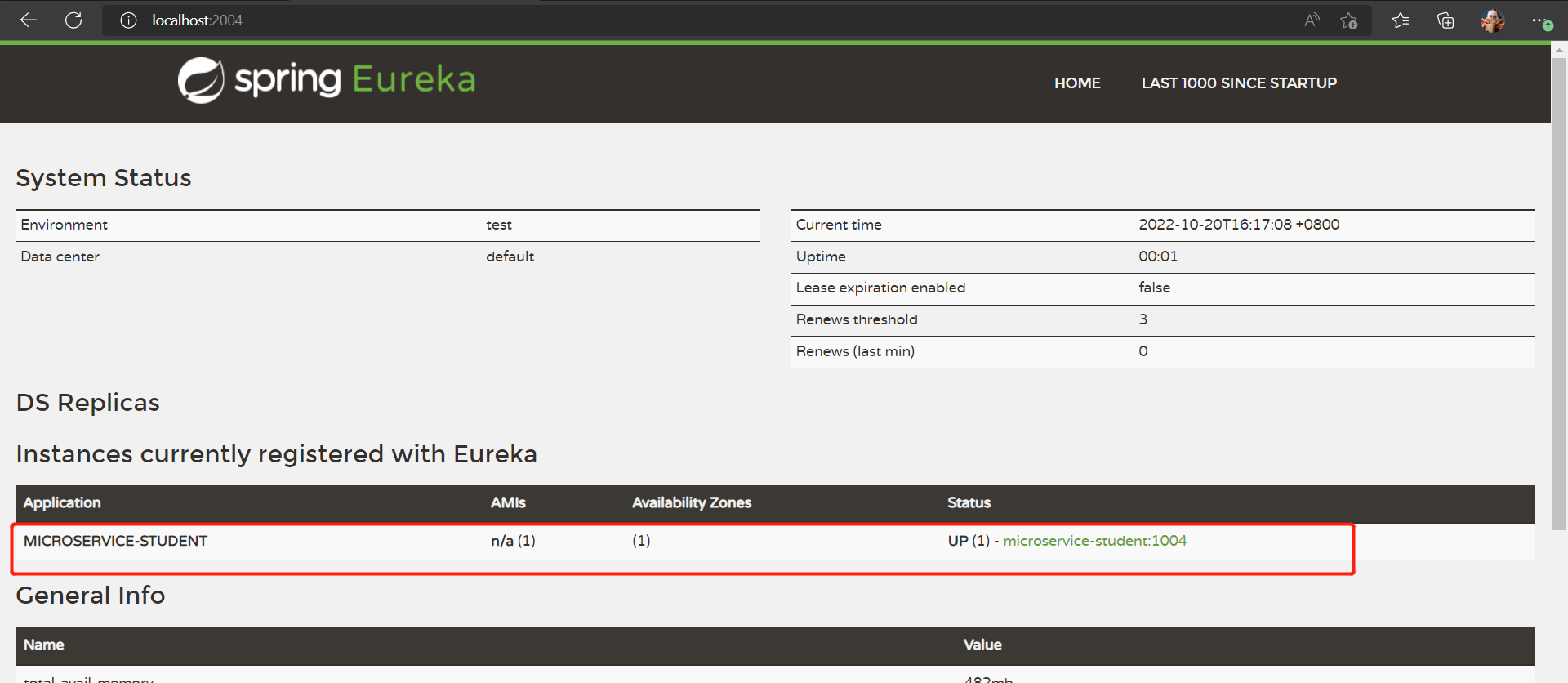
然后启动测试是否从远程读取了配置文件

如果读取到了配置文件那么就会注册到上边带有config的注册中心

启动 config-server

启动带有config的eureka

启动带有config的provider

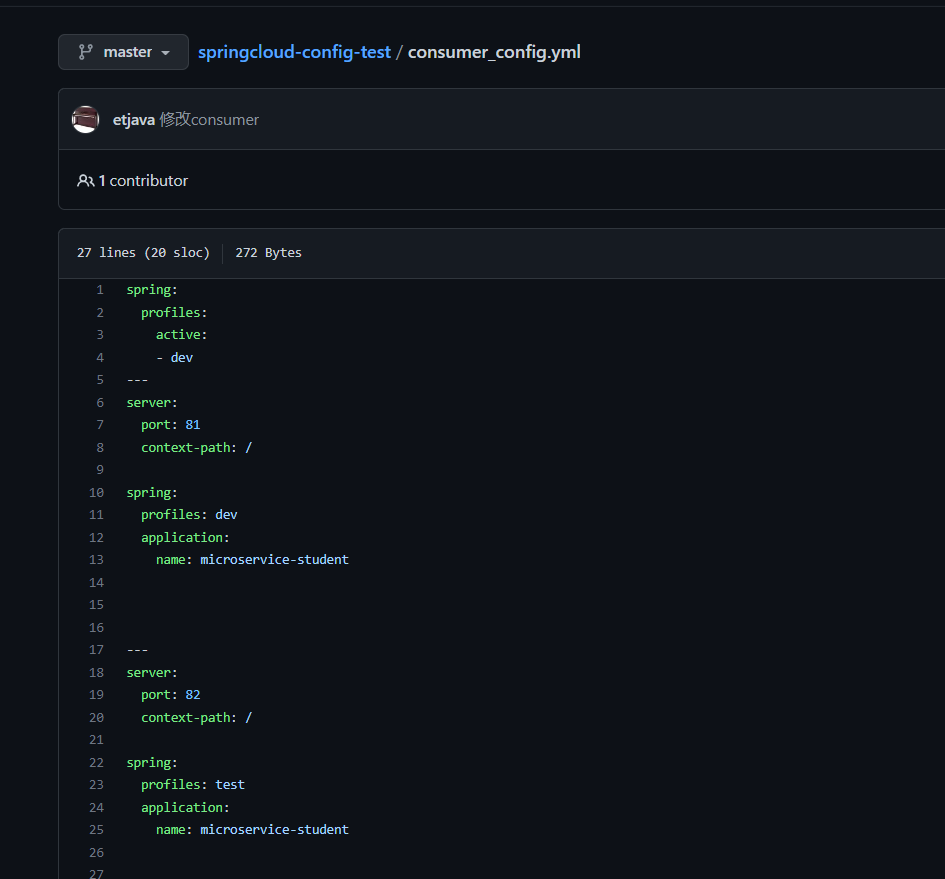


# [远程端口本地注册]服务消费者整合springcloud config

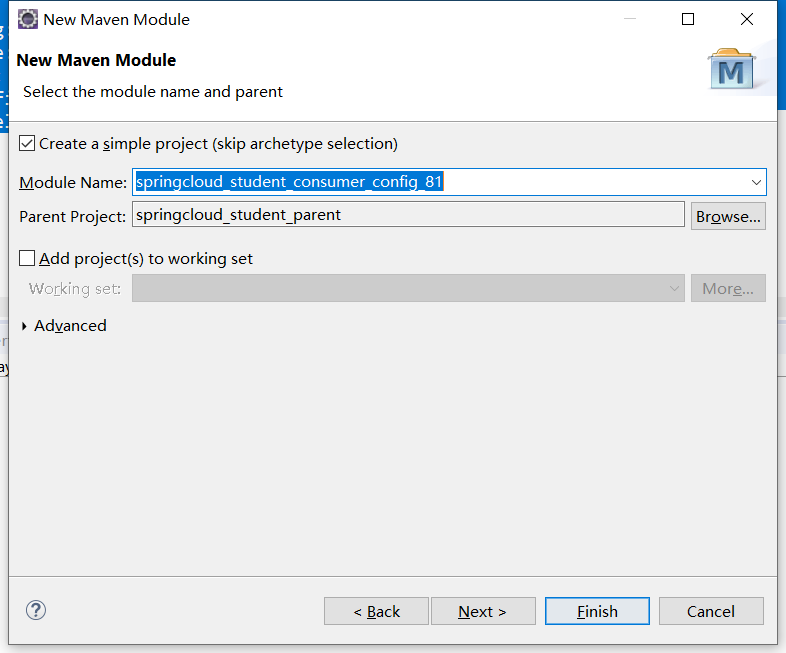
## 新建comsumer-config.yml

并上传至github

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 81  context-path: /  spring:  profiles: dev  application:  name: microservice-student  ---  server:  port: 82  context-path: /  spring:  profiles: test  application:  name: microservice-student |



## 新建服务消费者



## bootstrap.yml

spring:

application:

name: consumer-dev

cloud:

config:

# name和profile

name: consumer\_config

uri: http://localhost:6001/ # 配置configserver地址

profile: test # 级别

label: master # 分支 git中 默认master

fail-fast: true

## application.yml

spring:

application:

name: student-consumer-config

# ribbon 相关配置

eureka:

client:

#register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false

register-with-eureka: false

service-url:

defaultZone: http://localhost:2004/eureka/

# 单机 需要指定服务提供者的IP+端口

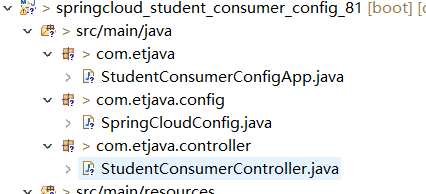
# provider\_host: http://localhost:1001

# 集群版 STUDENT-PROVIDER22 对应的是服务提供者配置文件中的application.name的值

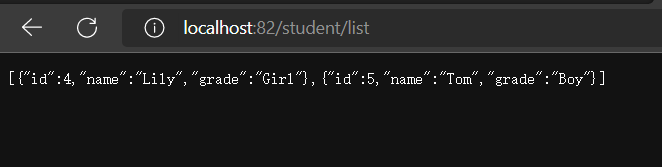
# 这里只能使用中划线 不能使用下划线 提供者那边也只能使用中划线

provider\_host: <http://MICROSERVICE-STUDENT>

## 复制其它消费者的类



## 启动测试



# [远程端口远程注册]服务消费者整合springcloud config

与[远程端口本地注册] 不同的仅在于两个配置文件

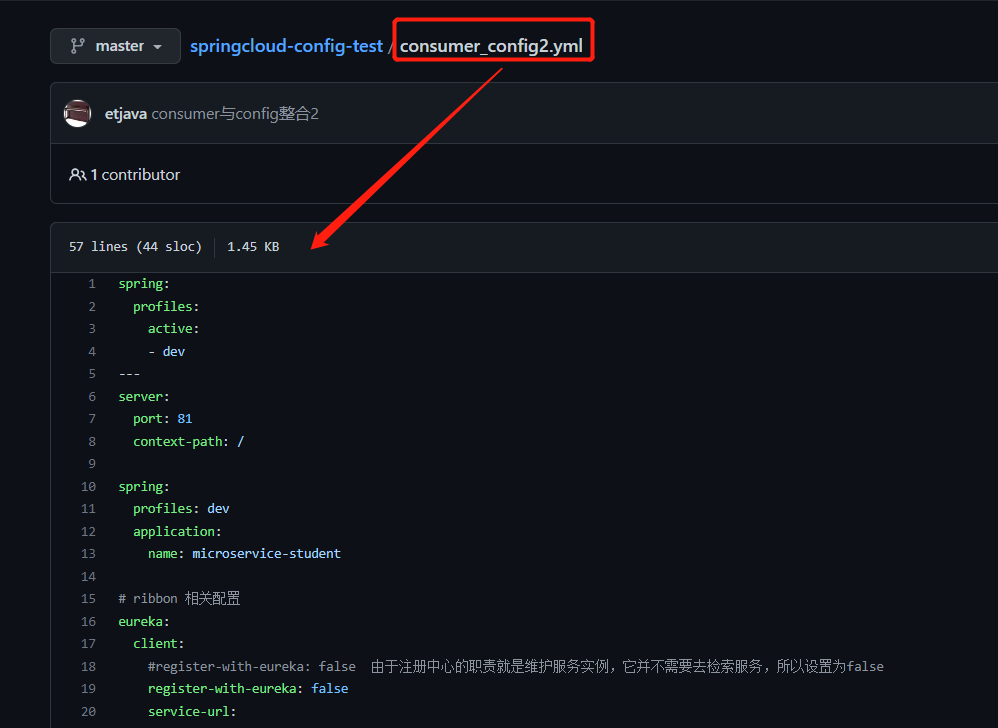
bootstrap.yml中cloud.config.name 改为consumer-config2

application.yml中仅保留application.name 和controller中用到的provider\_host

## 新建consumer-config2.yml

并上传至github

|  |
| --- |
| spring:  profiles:  active:  - dev  ---  server:  port: 81  context-path: /  spring:  profiles: dev  application:  name: microservice-student  # ribbon 相关配置  eureka:  client:  #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  register-with-eureka: false  service-url:  defaultZone: http://localhost:2004/eureka/  # 单机 需要指定服务提供者的IP+端口  # provider\_host: http://localhost:1001  # 集群版 STUDENT-PROVIDER22 对应的是服务提供者配置文件中的application.name的值  # 这里只能使用中划线 不能使用下划线 提供者那边也只能使用中划线  # provider\_host: http://MICROSERVICE-STUDENT  ---  server:  port: 82  context-path: /  spring:  profiles: test  application:  name: microservice-student  # ribbon 相关配置  eureka:  client:  #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  register-with-eureka: false  service-url:  defaultZone: http://localhost:2004/eureka/ |

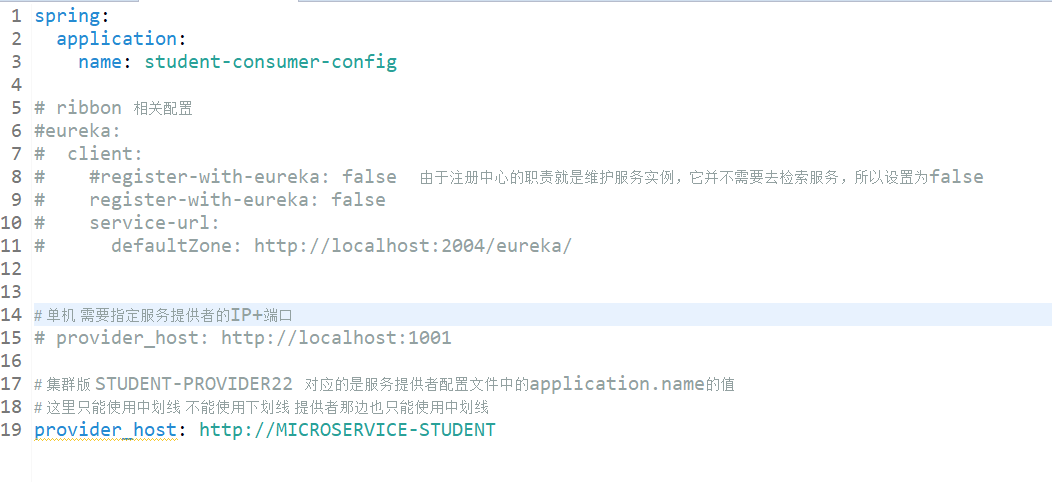


## bootstarp.yml

这里定义远程文件名称改为consumer\_config2 需要找consumer\_config2.yml

|  |
| --- |
| spring:  application:  name: consumer-dev  cloud:  config:  # name和profile  name: consumer\_config2  uri: http://localhost:6001/ # 配置configserver地址  profile: dev # 级别  label: master # 分支 git中 默认master  fail-fast: true |

## application.yml



|  |
| --- |
| spring:  application:  name: student-consumer-config    # ribbon 相关配置  #eureka:  # client:  # #register-with-eureka: false 由于注册中心的职责就是维护服务实例，它并不需要去检索服务，所以设置为false  # register-with-eureka: false  # service-url:  # defaultZone: http://localhost:2004/eureka/  # 单机 需要指定服务提供者的IP+端口  # provider\_host: http://localhost:1001  # 集群版 STUDENT-PROVIDER22 对应的是服务提供者配置文件中的application.name的值  # 这里只能使用中划线 不能使用下划线 提供者那边也只能使用中划线  provider\_host: http://MICROSERVICE-STUDENT |

## 创建启动类

|  |
| --- |
| @EnableEurekaClient  @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,  HibernateJpaAutoConfiguration.**class**})  **public** **class** StudentConsumerConfigApp {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StudentConsumerConfigApp.**class**, args);  }  } |

创建SpringCloud配置

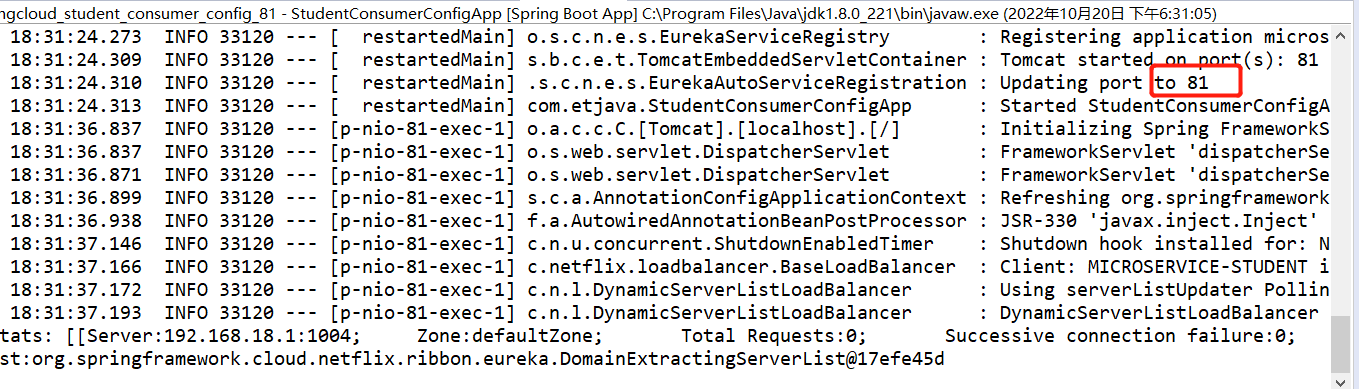
该类配置了ribbon的轮询方式

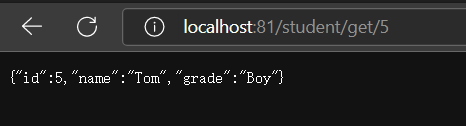
|  |
| --- |
| /\*\*  \* Spring Cloud 相关配置  \* **@author** etjava  \*  \*/  @Configuration  **public** **class** SpringCloudConfig {  /\*\*  \* 调用服务模版  \* 通过RestTemplate 调用远程服务  \* **@return**  \*/  @Bean  @LoadBalanced // 引入ribbon负载均衡  **public** RestTemplate getRestTemplate(){  **return** **new** RestTemplate();  }    /\*\*  \* 自定义轮询算法 默认5次轮询 5次访问仍热失败则不在去访问失败的节点  \* **@return**  \*/  @Bean  **public** IRule myRule(){  **return** **new** RetryRule();  }  } |

## controller

|  |
| --- |
| /\*\*  \* 服务消费者 controller  \* **@author** etjav  \*  \*/  @RestController  @RequestMapping("/student")  **public** **class** StudentConsumerController {    /\*\*  \* restTemplate 调用远程服务端  \*/  @Resource  **private** RestTemplate restTemplate;    @Value("${provider\_host}")  **private** String HOST;      /\*\*  \* 模拟hystrix处理雪崩效应  \* **@return**  \*/  @SuppressWarnings("unchecked")  @GetMapping(value="/getInfo")  @ResponseBody  **public** Map<String,Object> getInfo(){  **return** restTemplate.getForObject(HOST+"/student/getInfo/", Map.**class**);  }    /\*\*  \* 添加或者修改学生信息  \* **@param** student  \* **@return**  \*/  @PostMapping(value="/save")  **public** **boolean** save(Student student){  **return** restTemplate.postForObject(HOST+"/student/save", student, Boolean.**class**);  }    /\*\*  \* 查询学生信息  \* **@return**  \*/  @SuppressWarnings("unchecked")  @GetMapping(value="/list")  **public** List<Student> list(){  **return** restTemplate.getForObject(HOST+"/student/list", List.**class**);  }    /\*\*  \* 根据id查询学生信息  \* **@return**  \*/  @GetMapping(value="/get/{id}")  **public** Student get(@PathVariable("id") Integer id){  **return** restTemplate.getForObject(HOST+"/student/get/"+id, Student.**class**);  }    /\*\*  \* 根据id删除学生信息  \* **@return**  \*/  @GetMapping(value="/delete/{id}")  **public** **boolean** delete(@PathVariable("id") Integer id){  **return** restTemplate.getForObject(HOST+"/student/delete/"+id,Boolean.**class**);  }  } |

## 启动并测试





# config配置搜索路径

config用来配置引用外部yml配置文件的工具，可以将github中指定的文件获取到然后在动态添加到application中进行配置项目

通常我们项目中不会把所有的配置文件都存放到根目录下 此时就需要按照分类的目录进行检索需要的配置文件

**在检索时要注意文件命名格式 aa-bb.yml**

server:

port: 6001

context-path: /

spring:

application:

name: student-config

# 配置springcloud的config server

cloud:

config:

server:

git:

uri: https://github.com/etjava/springcloud-config-test # uri指向的是git仓库中的地址 注意不能带有后缀 即 .git

search-paths:

- aaa

- bbb