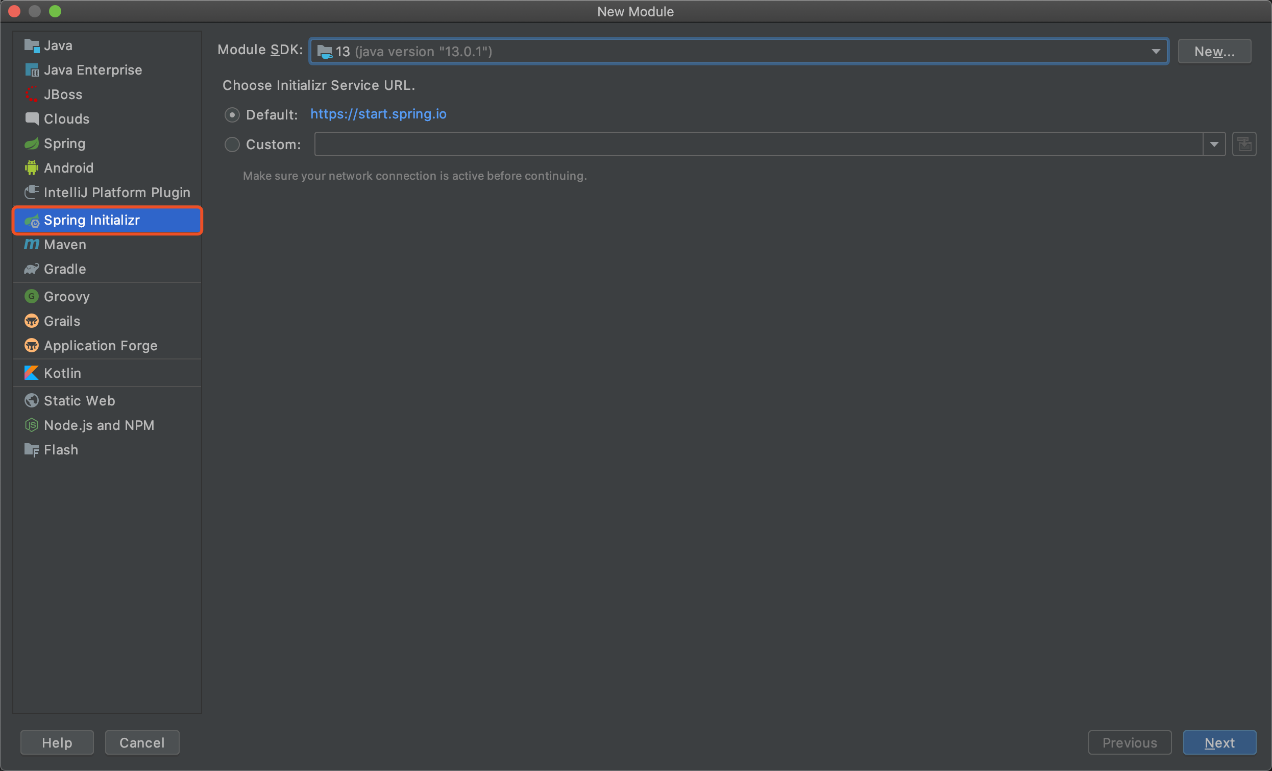
# 简介

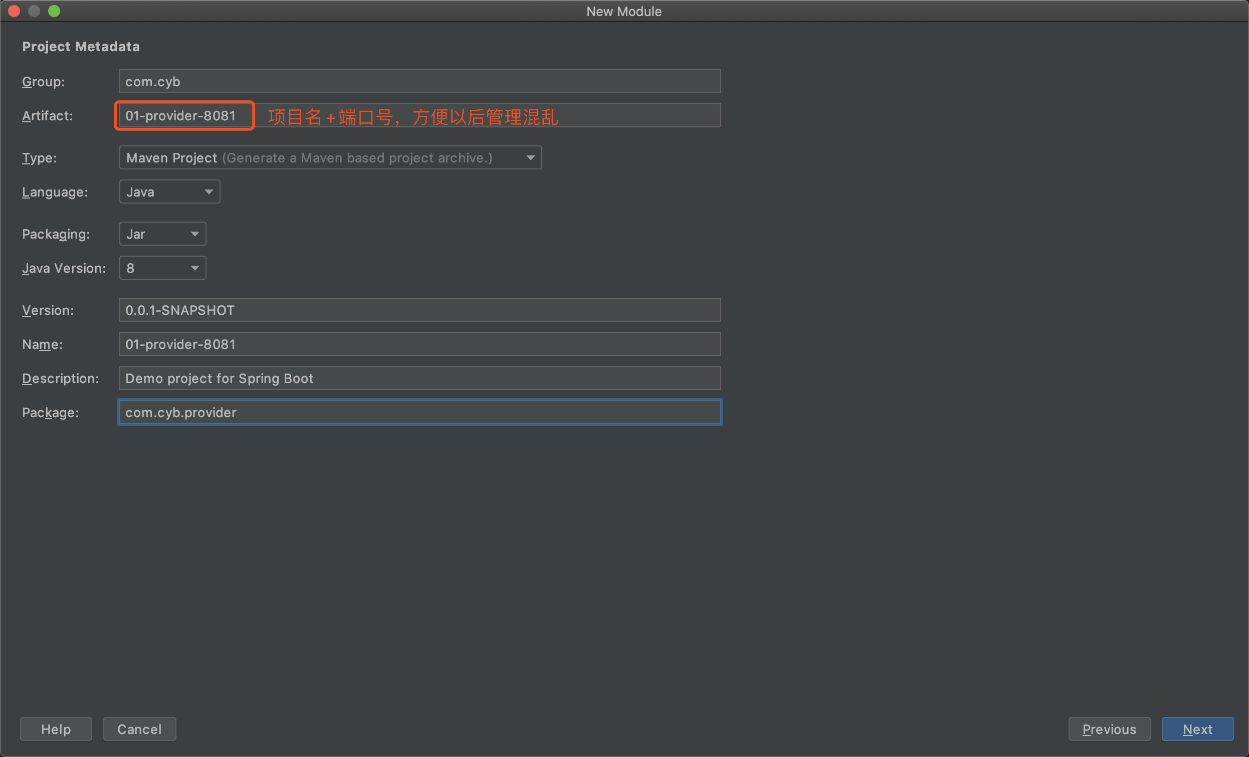
Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽了复杂的配置和实现原理，最终给开发者流出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

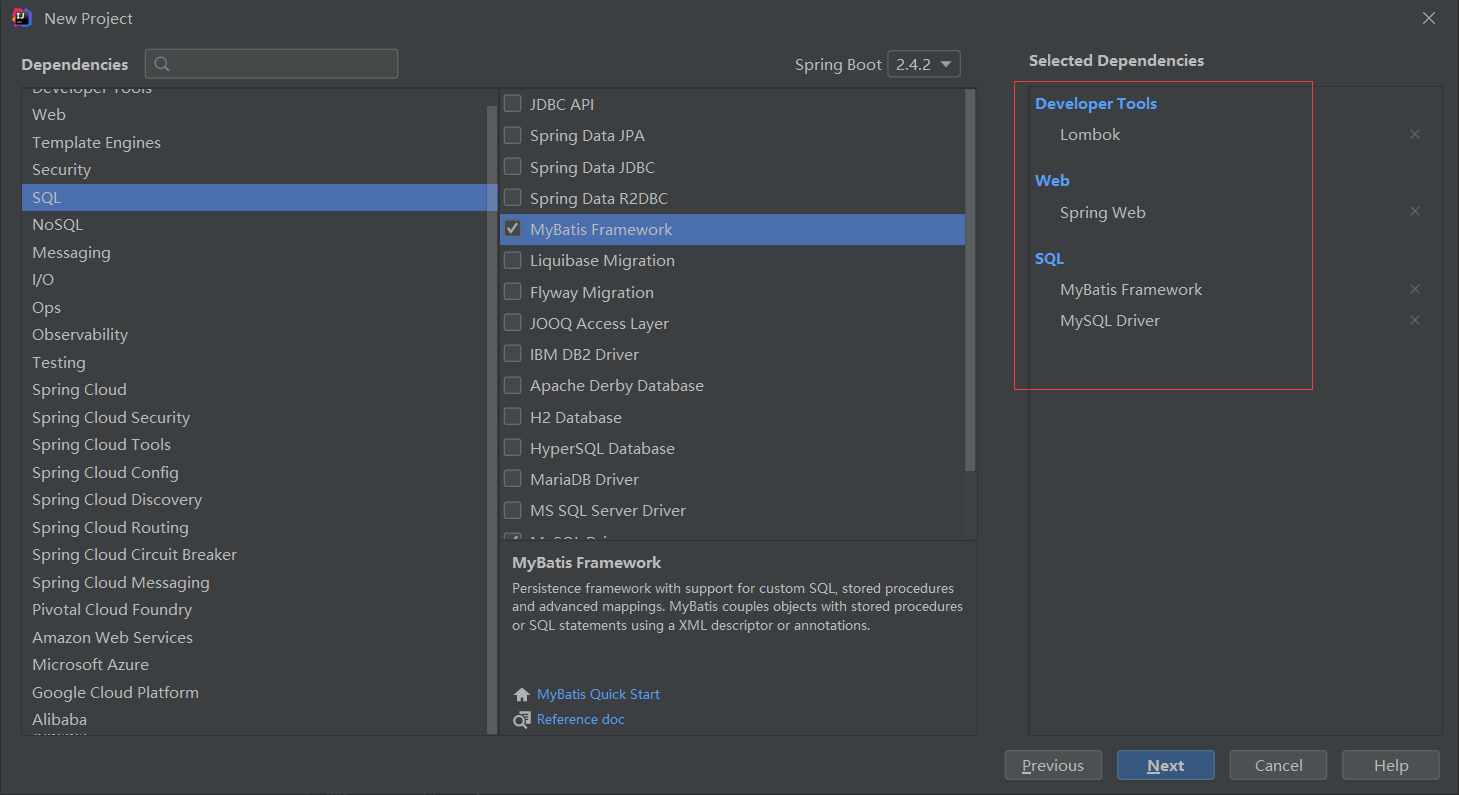
## 服务提供者项目

本示例使用Spring的RestTemplate实现消费者对提供者的调用，并未使用到Spring Cloud，但其为后续Spring Cloud的运行测试环境。使用MySql数据库，使用Spring Data JPA作为持久层技术。

### 创建工程







### 添加Druid依赖

pom.xml

<!--Druid依赖-->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.10</version>

</dependency>

### 定义bean包

Controller处理器方法的返回值是作为JSON数据响应给浏览器的；这个数据转换工作是由SpringMvc的HttpMessageConverter接口完成的。

Depart.java

1. package com.cyb.provider.bean;
2. import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
3. import lombok.Data;
4. @Data
5. public class Depart {
6. private Integer id;
7. private String name;
8. private String dbase;
9. }

### 定义Dao包

DepartDao.java

1. package com.cpb.provider.dao;
2. import com.cpb.provider.bean.Depart;
3. import org.apache.ibatis.annotations.Mapper;
4. import java.util.List;
5. @Mapper
6. public interface DepartDao {
7. */\*\**
8. \* 增加
9. \* @param depart
10. \* @return
11. \*/
12. Integer save(Depart depart);
13. */\*\**
14. \* 删除
15. \* @param id
16. \* @return
17. \*/
18. Integer removeDepartById(int id);
19. */\*\**
20. \* 修改
21. \* @param depart
22. \* @return
23. \*/
24. Integer modifyDepart(Depart depart);
25. */\*\**
26. \* 查询id
27. \* @param id
28. \* @return
29. \*/
30. Depart getDepartById(int id);
31. */\*\**
32. \* 查询所有
33. \* @return
34. \*/
35. List<Depart> listAllDeparts();
36. }

### 定义service包

DepartService.java

1. package com.cyb.provider.Service;
2. import com.cyb.provider.bean.Depart;
3. import java.util.List;
4. */\*\**
5. \* 业务接口
6. \*/
7. public interface DepartService {
8. */\*\**
9. \* 增加
10. \* @param depart
11. \* @return
12. \*/
13. boolean saveDepart(Depart depart);
14. */\*\**
15. \* 删除
16. \* @param id
17. \* @return
18. \*/
19. boolean removeDepartById(int id);
20. */\*\**
21. \* 修改
22. \* @param depart
23. \* @return
24. \*/
25. boolean modifyDepart(Depart depart);
26. */\*\**
27. \* 查询id
28. \* @param id
29. \* @return
30. \*/
31. Depart getDepartById(int id);
32. */\*\**
33. \* 查询所有
34. \* @return
35. \*/
36. List<Depart> listAllDeparts();
37. }

DepartServiceImpl.java

1. package com.cyb.provider.Service;
2. import com.cyb.provider.bean.Depart;
3. import com.cyb.provider.repository.DepartRepository;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.stereotype.Service;
6. import java.util.List;
7. @Service
8. public class DepartServiceImpl implements DepartService {
9. @Autowired
10. private DepartDao departDao;
11. @Override
12. public boolean saveDepart(Depart depart) {
13. *//返回结果有值，操作成功；没值失败，操作失败*
14. return departDao.save(depart) == null ? false : true;
15. }
16. @Override
17. public boolean removeDepartById(int id) {
18. *//对于deleteById方法，若DB中存在该id，一定能删除；不存在该id，抛异常*
19. if (departDao.existsById(id)) {
20. departDao.deleteById(id);
21. return true;
22. }
23. return false;
24. }
25. @Override
26. public boolean modifyDepart(Depart depart) {
27. *//返回结果有值，操作成功；没值失败，操作失败*
28. return departDao.save(depart) == null ? false : true;
29. }
30. @Override
31. public Depart getDepartById(int id) {
32. *//getOne()方法：若其指定的id不存在，该方法将抛出异常*
33. if (departDao.existsById(id)){
34. return departDao.getOne(id);
35. }
36. Depart depart=new Depart();
37. depart.setName("not this depart");
38. return depart;
39. }
40. @Override
41. public List<Depart> listAllDeparts() {
42. return departDao.findAll();
43. }
44. }

### 定义controller包

DepartController.java

1. package com.cyb.provider.controller;
2. import com.cyb.provider.Service.DepartService;
3. import com.cyb.provider.bean.Depart;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.web.bind.annotation.\*;
6. import java.util.List;
7. @RequestMapping("/provider/depart")
8. @RestController
9. public class DepartController {
10. @Autowired
11. private DepartService service;
12. @RequestMapping("/save")
13. public boolean saveHandle(@RequestBody Depart depart) {
14. return service.saveDepart(depart);
15. }
16. @RequestMapping("/del/{id}")
17. public boolean deleteHandle(@PathVariable("id") int id) {
18. return service.removeDepartById(id);
19. }
20. @RequestMapping("/update")
21. public boolean updateHandle(@RequestBody Depart depart) {
22. return service.modifyDepart(depart);
23. }
24. @RequestMapping("/get/{id}")
25. public Depart getHandle(@PathVariable("id") int id) {
26. return service.getDepartById(id);
27. }
28. @RequestMapping("/list")
29. public List<Depart> listHandle() {
30. return service.listAllDeparts();
31. }
32. }

### 配置文件

Departmapper.xml

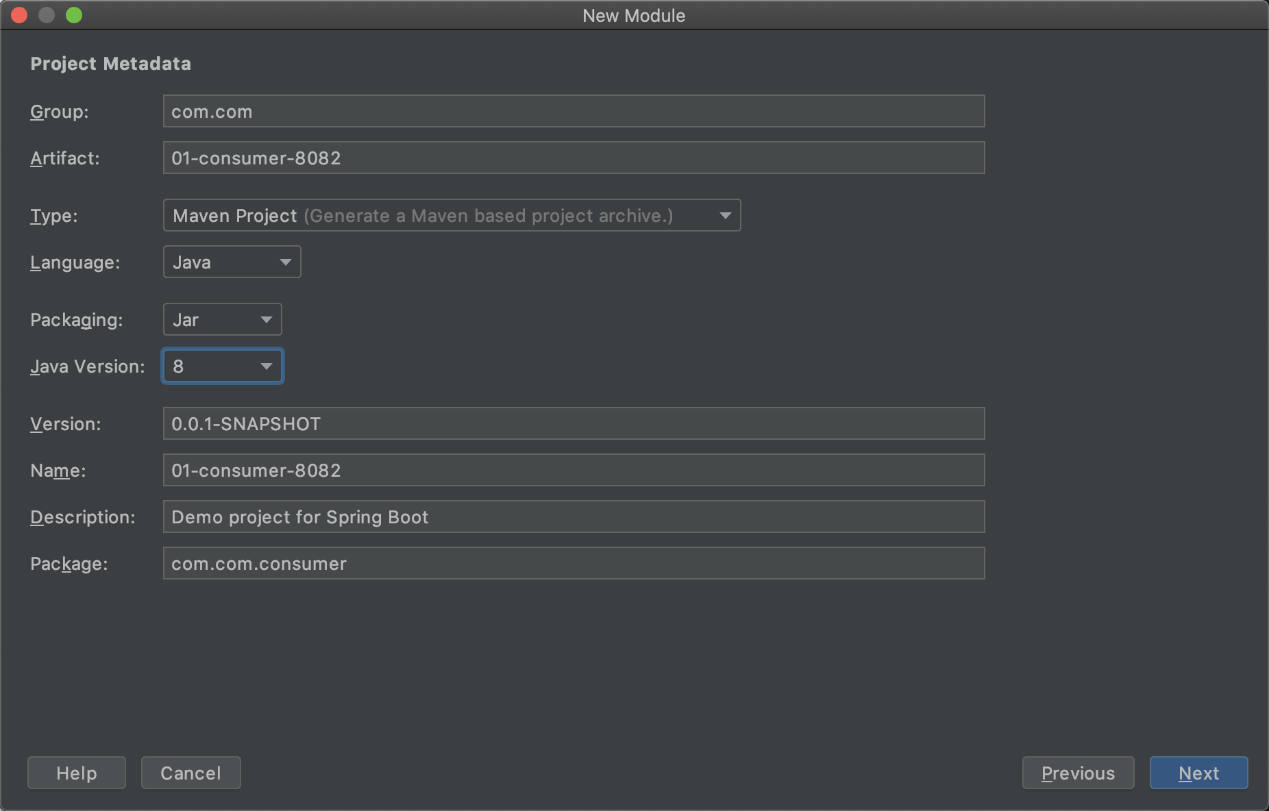
1. <?xml version="1.0" encoding="UTF-8" ?>
2. <!DOCTYPE mapper
3. PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
4. "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
5. <mapper namespace="com.cpb.provider.dao.DepartDao">
6. </mapper>

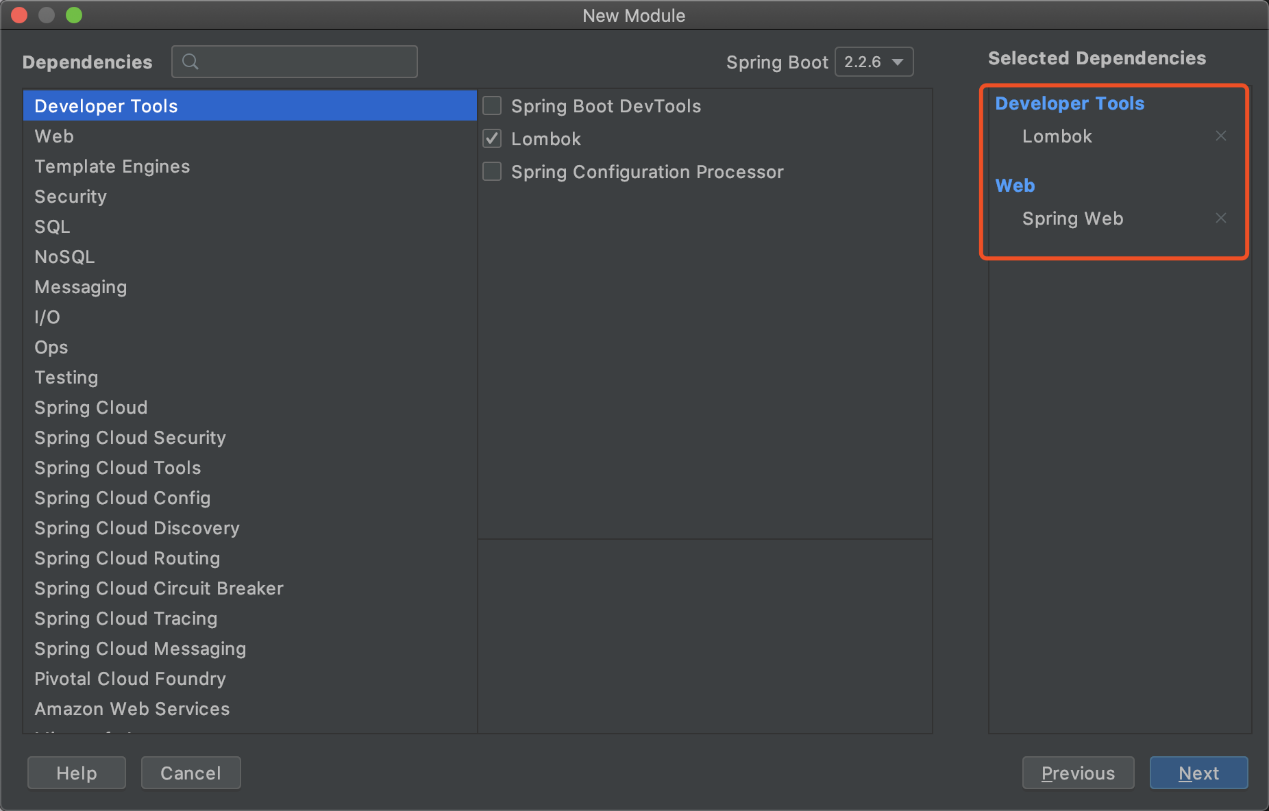
application.properties

1. *# 端口号*
2. server.port=8081
3. *# 应用启动是否自动创建表，默认为false*
4. spring.jpa.generate-ddl=true
5. *# 是否在控制台显示sql语句，默认为false*
6. spring.jpa.show-sql=true
7. *# 应用启动时设置不重新建表*
8. spring.jpa.hibernate.ddl-auto=none
9. *# 数据类型*
10. spring.datasource.type=
11. spring.datasource.url=
12. spring.datasource.username=root
13. spring.datasource.password=
14. *# 设置日志输出格式*
15. logging.pattern.console=level-%level%msg%n
16. *# Spring Boot启动时的日志级别*
17. logging.level.root=info

## 服务消费者项目

### 创建工程





### 定义bean包

Depart.java

1. package com.com.consumer.bean;
2. import lombok.Data;
3. @Data
4. public class Depart {
5. private Integer id;
6. private String name;
7. private String dbase;
8. }

### 定义codeconfig包

DepartCodeConfig.java

1. package com.com.consumer.codeconfig;
2. import org.springframework.context.annotation.Bean;
3. import org.springframework.context.annotation.Configuration;
4. import org.springframework.web.client.RestTemplate;
5. @Configuration
6. public class DepartCodeConfig {
7. @Bean
8. public RestTemplate restTemplate(){
9. return new RestTemplate();
10. }
11. }

### 定义controller包

DepartController.java

1. package com.com.consumer.controller;
2. import com.com.consumer.bean.Depart;
3. import org.springframework.beans.factory.annotation.Autowired;
4. import org.springframework.web.bind.annotation.\*;
5. import org.springframework.web.client.RestTemplate;
6. import java.util.List;
7. @RestController
8. @RequestMapping("/consumer/depart")
9. public class DepartController {
10. @Autowired
11. private RestTemplate restTemplate;
12. @RequestMapping("/save")
13. public boolean saveHandle(Depart depart) {
14. String url="http://localhost:8081/provider/depart/save";
15. return restTemplate.postForObject(url,depart,Boolean.class);
16. }
17. @RequestMapping("/del/{id}")
18. public void deleteHandle(@PathVariable("id") int id) {
19. String url="http://localhost:8081/provider/depart/del/"+id;
20. restTemplate.delete(url);
21. }
22. @RequestMapping("/update")
23. public void updateHandle(@RequestBody Depart depart) {
24. String url="http://localhost:8081/provider/depart/update";
25. restTemplate.put(url,depart);
26. }
27. @RequestMapping("/get/{id}")
28. public Depart getHandle(@PathVariable("id") int id) {
29. String url="http://localhost:8081/provider/depart/get/"+id;
30. return restTemplate.getForObject(url,Depart.class);
31. }
32. @RequestMapping("/list")
33. public List<Depart> listHandle() {
34. String url="http://localhost:8081/provider/depart/list";
35. return restTemplate.getForObject(url,List.class);
36. }
37. }

### application.properties

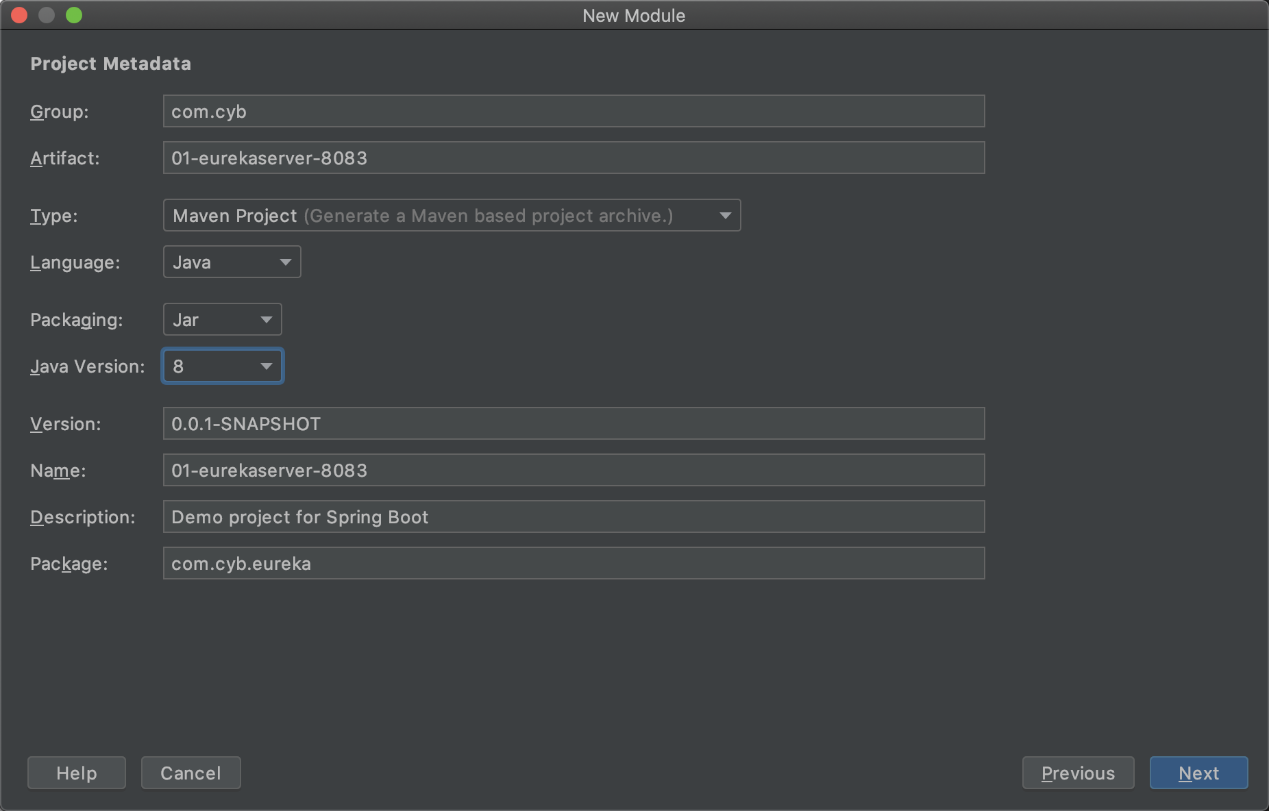
1. server.port=8082

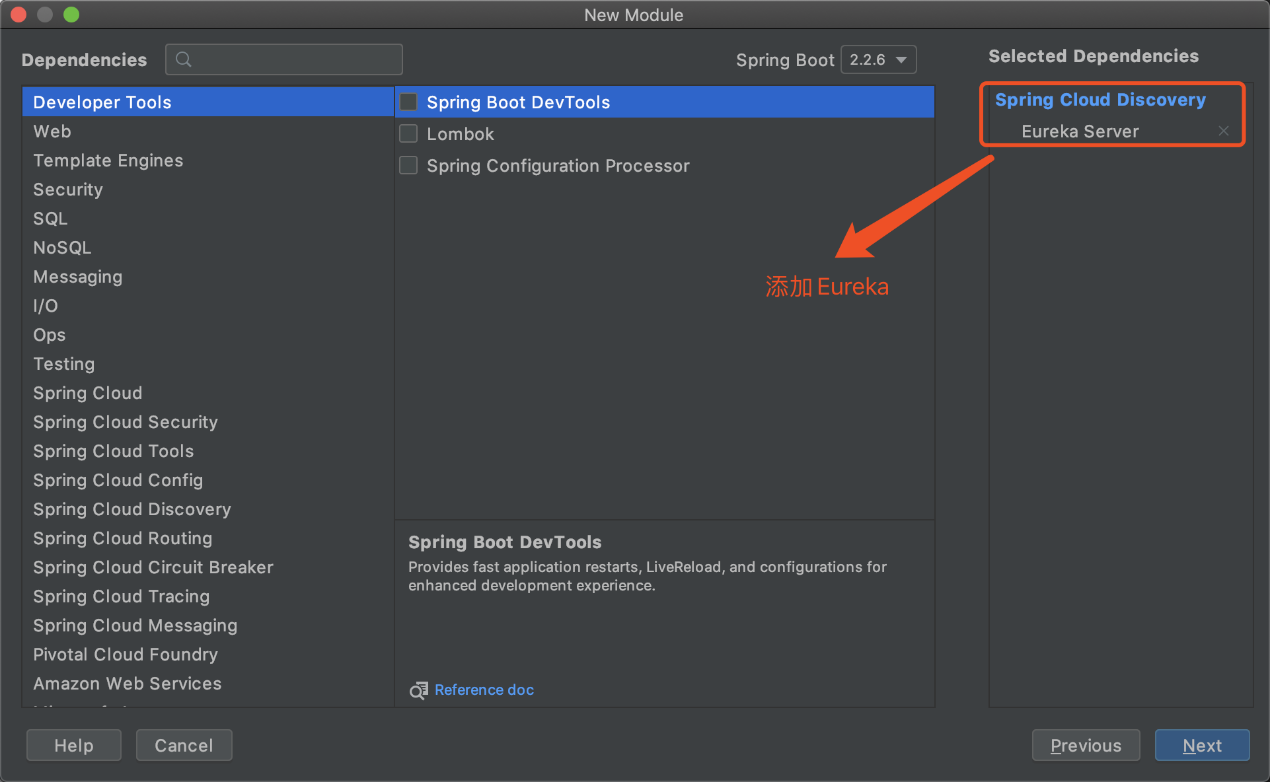
# 微服务中心Eureka

## 创建Eureka服务中心

* 导入Eureka依赖
* 在配置文件中配置EurekaServer
* 在启动类中添加注解@EnableEurekaServer开启Eureka

### 创建工程





### 导入依赖

这里要导入的依赖并非Spring Cloud工程直接的依赖。而是由Eureka Server所依赖的，JDK9之前包含其所需要的依赖，JDK9之后，Eureka所需的依赖被踢出了,需要单独添加依赖。JDK9之前的不需要以下依赖。

<!--Eureka添加依赖开始-->

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>com.sun.xml.bind</groupId>

<artifactId>jaxb-core</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>com.sun.xml.bind</groupId>

<artifactId>jaxb-impl</artifactId>

<version>2.3.0</version>

</dependency>

<dependency>

<groupId>javax.activation</groupId>

<artifactId>activation</artifactId>

<version>1.1.1</version>

</dependency>

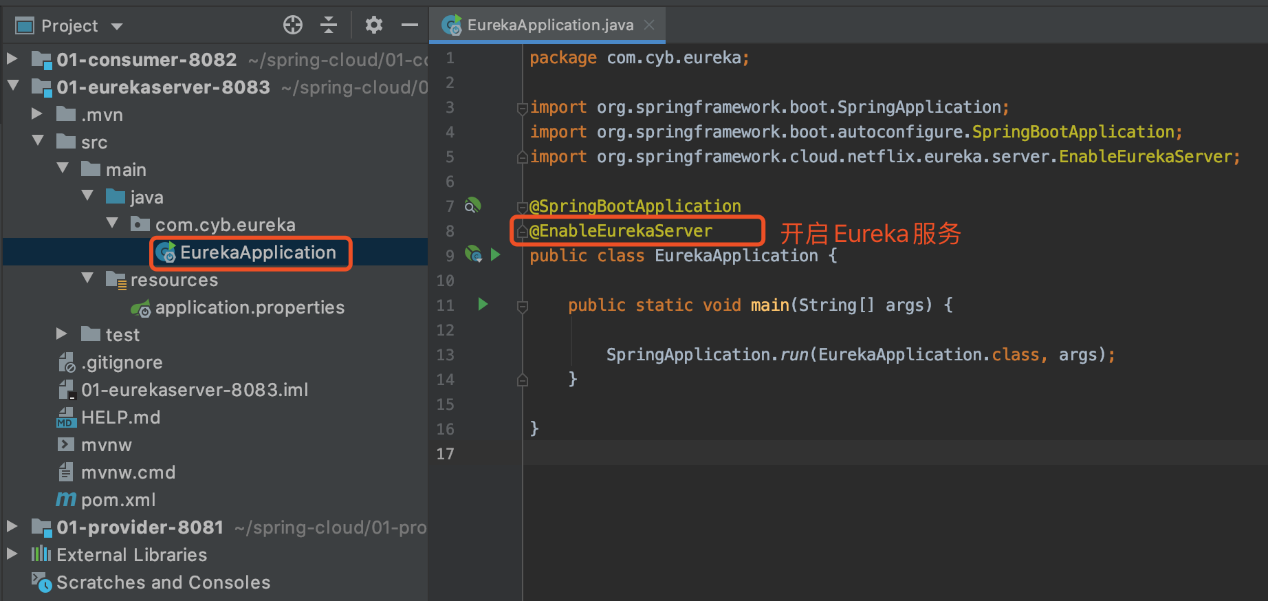
<!--Eureka添加依赖结束-->

### 设置配置文件

application.properties

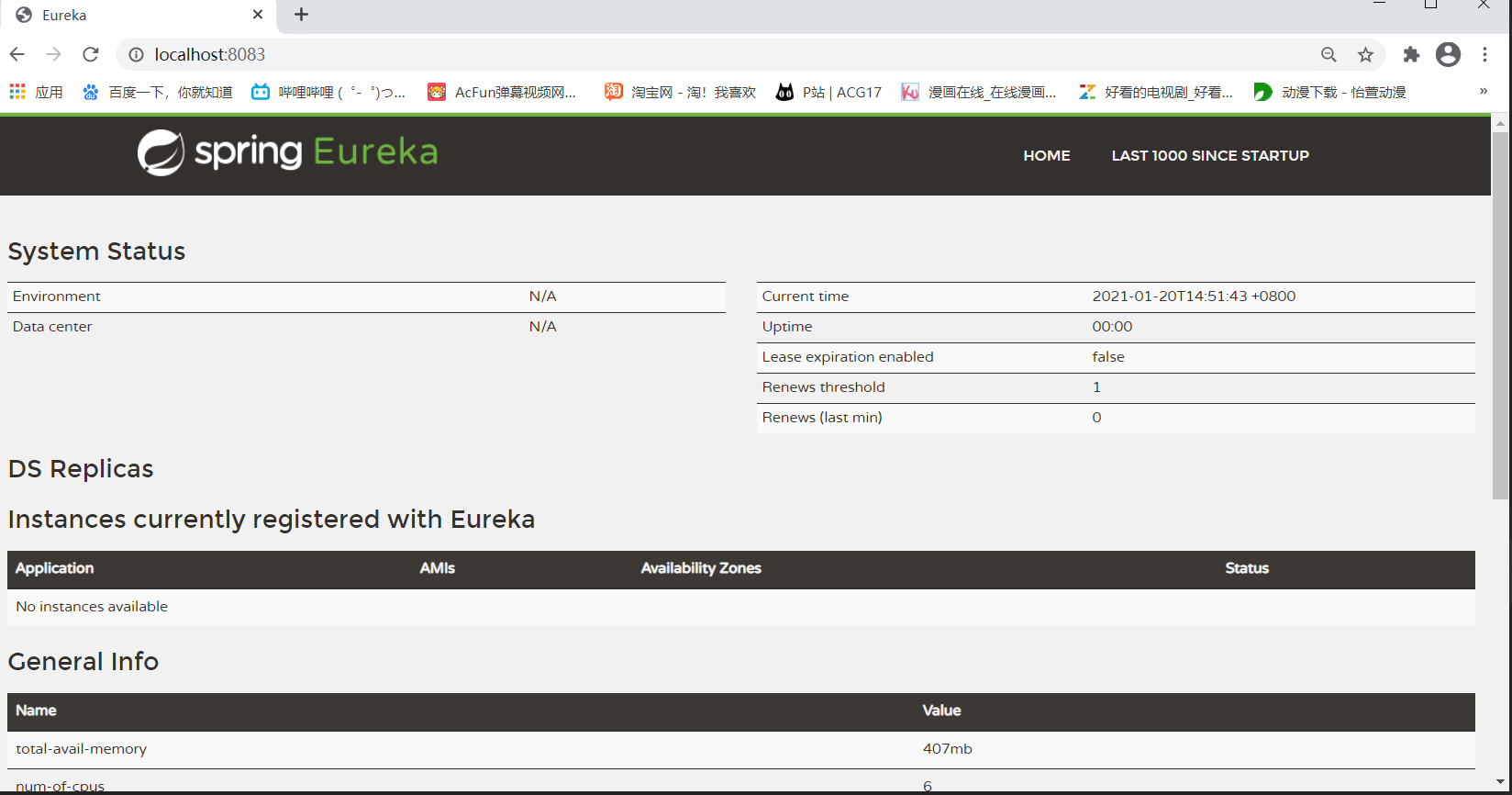
1. server.port=8083
2. # 配置Eureka，开始
3. # 配置Eureka主机名
4. eureka.instance.hostname=localhost
5. # 指定当前主机是否需要向注册中心注册(不用，因为当前主机是Server，不是Client)
6. eureka.client.register-with-eureka=false
7. # 指定当前主机是否需要获取注册信息(不用，因为当前主机是Server，不是Client)
8. eureka.client.fetch-registry=false
9. # ${eureka.instance.hostname}和${server.port}，动态引入变量的值
10. # 暴露服务中心地址
11. eureka.client.service-url.defaultZone=http:*//${eureka.instance.hostname}:${server.port}/eureka*

### 启动类上添加注解



### 项目启动测试

可以看到eurekaserver服务界面。



## 创建提供者工程

* 添加Eureka客户端依赖
* 在配置文件中指定要注册的Eureka注册中心
* 在启动类上添加@EnableEurekaClient注解

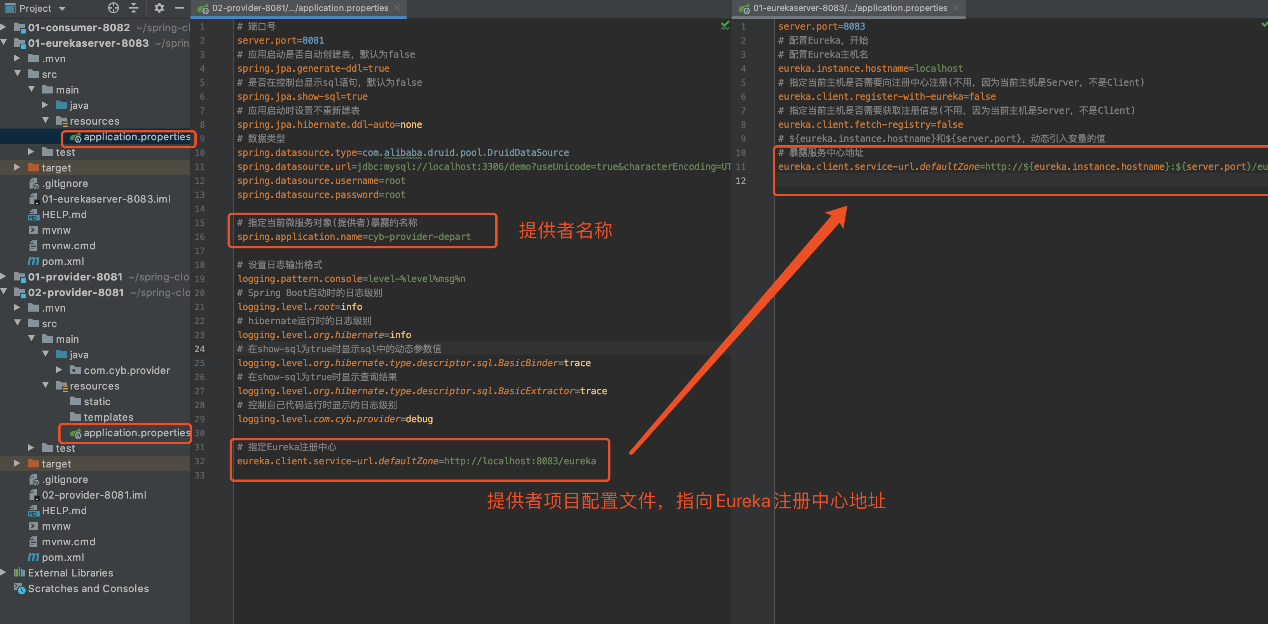
### 创建工程

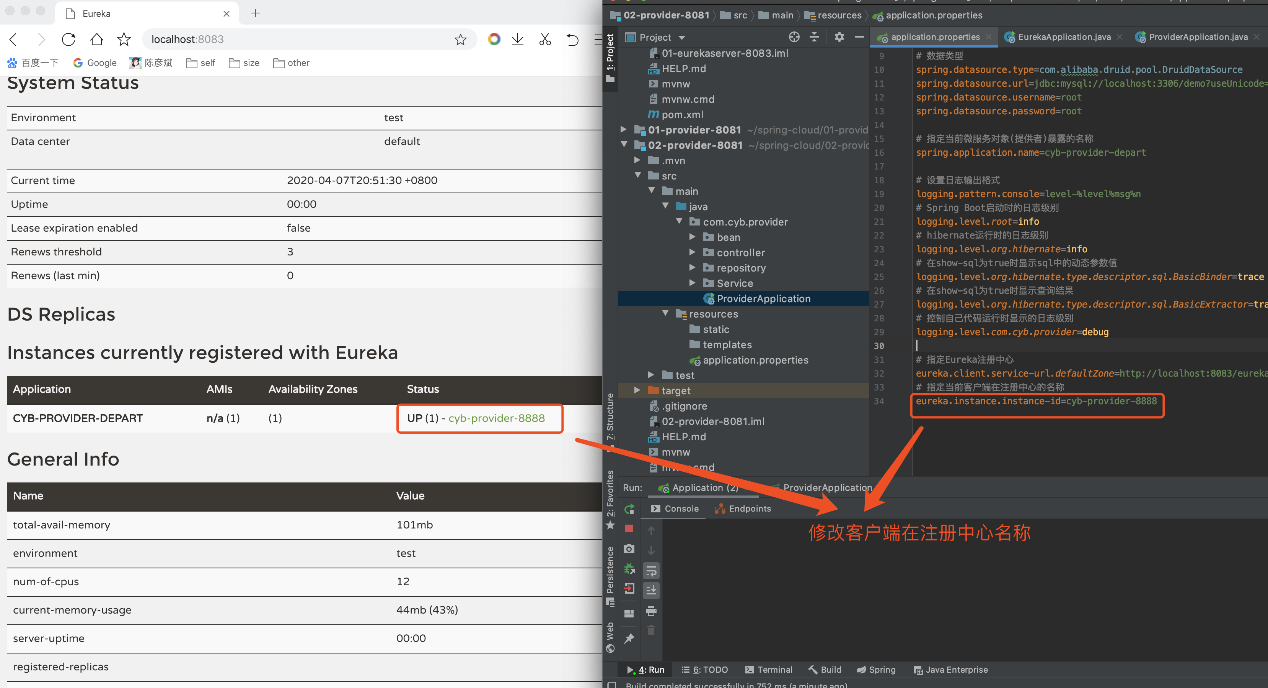
拷贝一份：01-provider-8081，重命名为：02-provider-8081

### 添加依赖

1. <dependencies>
2. *<!--Eureka客户端依赖-->*
3. <dependency>
4. <groupId>org.springframework.cloud</groupId>
5. <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
6. <version>2.2.2.RELEASE</version>
7. </dependency>
8. </dependencies>
9. *<!-- Eureka依赖管理模块 -->*
10. <dependencyManagement>
11. <dependencies>
12. <dependency>
13. <groupId>org.springframework.cloud</groupId>
14. <artifactId>spring-cloud-dependencies</artifactId>
15. <version>Finchley.SR1</version>
16. <type>pom</type>
17. </dependency>
18. </dependencies>
19. </dependencyManagement>

### 修改配置文件





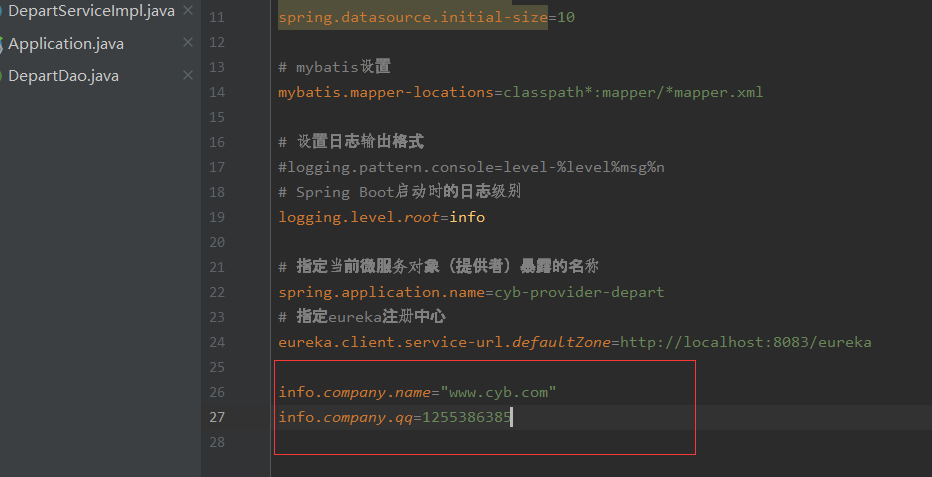
## actuator完善微服务info

在提供者配置文件中，设置actuator的info监控终端。

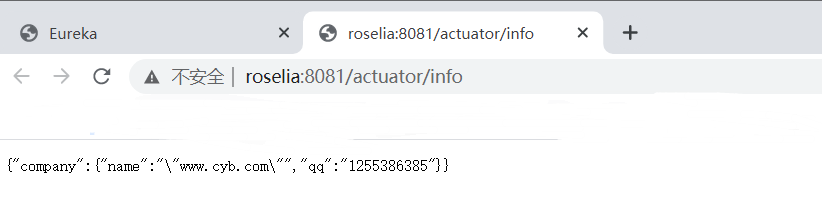
### 添加提供者依赖

1. *<!-- actuator依赖 -->*
2. <dependency>
3. <groupId>org.springframework.boot</groupId>
4. <artifactId>spring-boot-starter-actuator</artifactId>
5. </dependency>

### 添加info配置文件



### 测试



## 创建消费者工程2

消费者将使用提供者暴露的服务名称(spring.application.name)来消费服务。

* 添加Eureka客户端依赖
* 在配置文件中指定Eureka注册中心
* 在DepartCodeConfig类中添加@LoadBalanced注解
* 在启动类上添加@EnableEurekaClient注解

### 创建工程

复制01-consumer-8082，重复名为02-consumer-8082

### 添加依赖

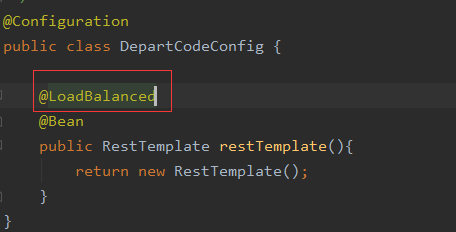
pom.xml

1. <dependencies>
2. *<!--Eureka客户端依赖-->*
3. <dependency>
4. <groupId>org.springframework.cloud</groupId>
5. <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
6. <version>2.2.2.RELEASE</version>
7. </dependency>
8. </dependencies>
9. *<!-- Eureka依赖管理模块 -->*
10. <dependencyManagement>
11. <dependencies>
12. <dependency>
13. <groupId>org.springframework.cloud</groupId>
14. <artifactId>spring-cloud-dependencies</artifactId>
15. <version>Finchley.SR1</version>
16. <type>pom</type>
17. </dependency>
18. </dependencies>
19. </dependencyManagement>
20. <!-- actuator依赖 -->
21. <dependency>
22. <groupId>org.springframework.boot</groupId>
23. <artifactId>spring-boot-starter-actuator</artifactId>
24. </dependency>

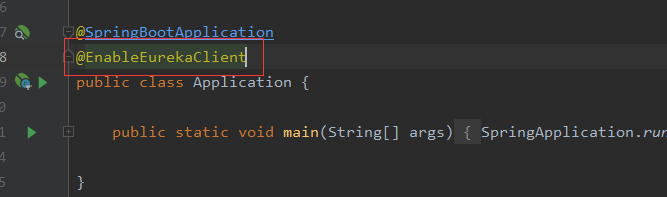
### 修改配置文件

1. *# 指定当前微服务对外(消费者)暴露的名称*
2. spring.application.name=cyb-consumer-depart
3. *# 指定Eureka注册中心*
4. eureka.client.service-url.defaultZone=http://localhost:8083/eureka

添加@LoadBalanced注解



启动类上添加注解



## 服务发现Discovery

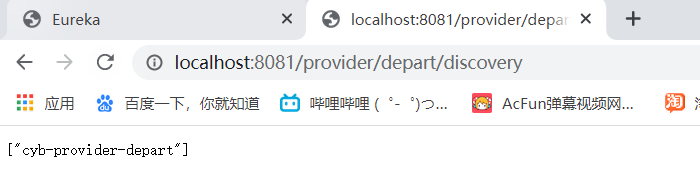
即通过“服务发现客户端”读取EurekaServer中的服务列表，获取指定名称的微服务详情。

### 修改处理器

在任何微服务的提供者或消费者处理器中，只要获取到“服务发现Client”，即可读取到Eureka Server的微服务列表。

1. @RequestMapping("/discovery")
2. public Object discoveryHandle(){
3. *// 获取服务注册列表中所有的微服务名称*
4. List<String> springApplicationNames = client.getServices();
5. for (String name:springApplicationNames){
6. *// 获取提供指定微服务名称的所有提供者主机*
7. List<ServiceInstance> instances = client.getInstances(name);
8. for (ServiceInstance instance:instances){
9. String host = instance.getHost();
10. int port = instance.getPort();
11. System.out.println(MessageFormat.format("host:{0},port:{1}",host,port));
12. }
13. }
14. return springApplicationNames;
15. }

### 测试



## EurekaServer集群

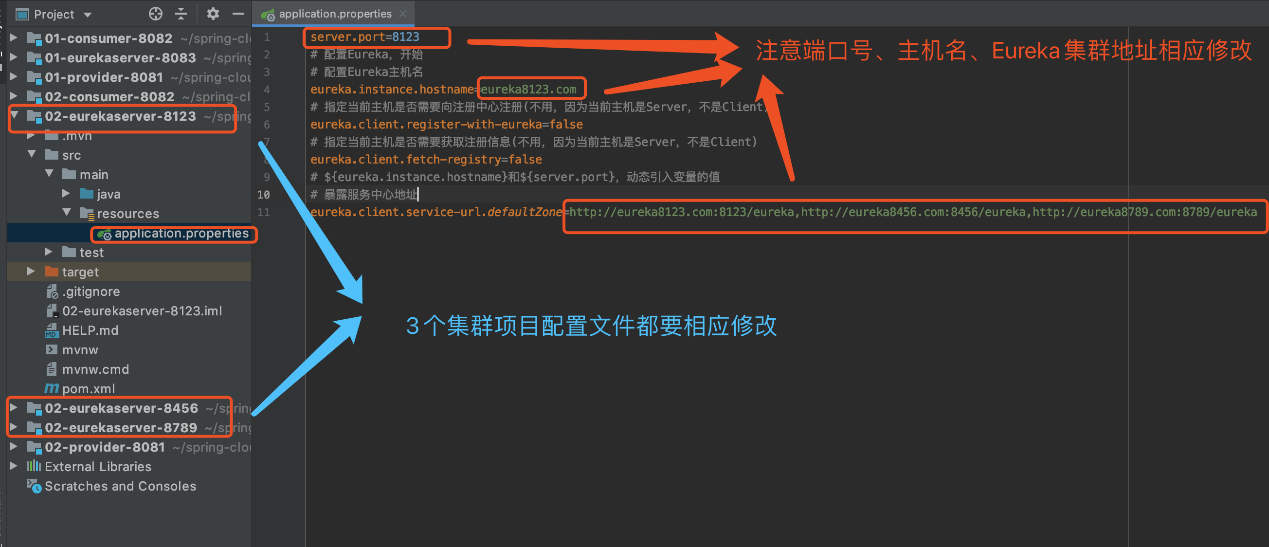
单个EurekaServer 不仅吞吐量有限，还存在单点问题，所以我们会使用EurekaServer集群，这里要搭建的EurekaServer集群中包含3个EurekaServer节点，其端口号分别为8123，8456，8789。

### 设置域名

由于这些Eureka 在这里都是运行在当前的这一台主机，而Eureka管理页面中显示的仅仅是Eureka主机的域名，不显示端口号，所以为了在Eureka管理页面可以区分Eureka集群中各个主机，我们这里先为每一个Eureka节点设置一个不同的域名。修改host文件为：

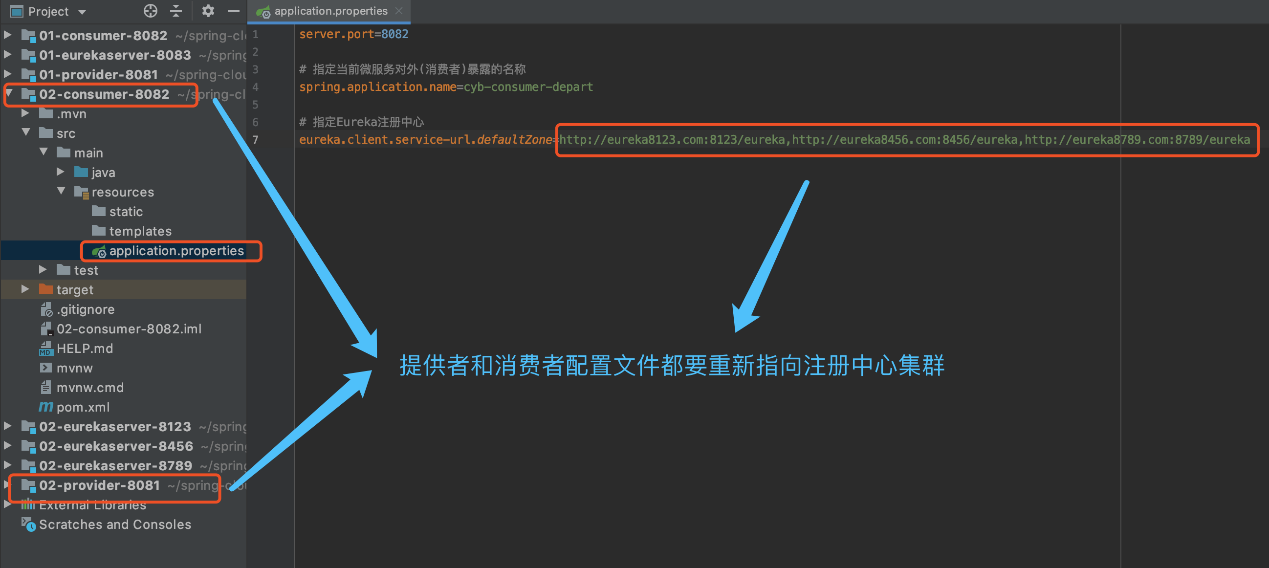
* 127.0.0.1 eureka8123.com
* 127.0.0.1 eureka8456.com
* 127.0.0.1 eureka8789.com

### 复制并修改EurekaServer

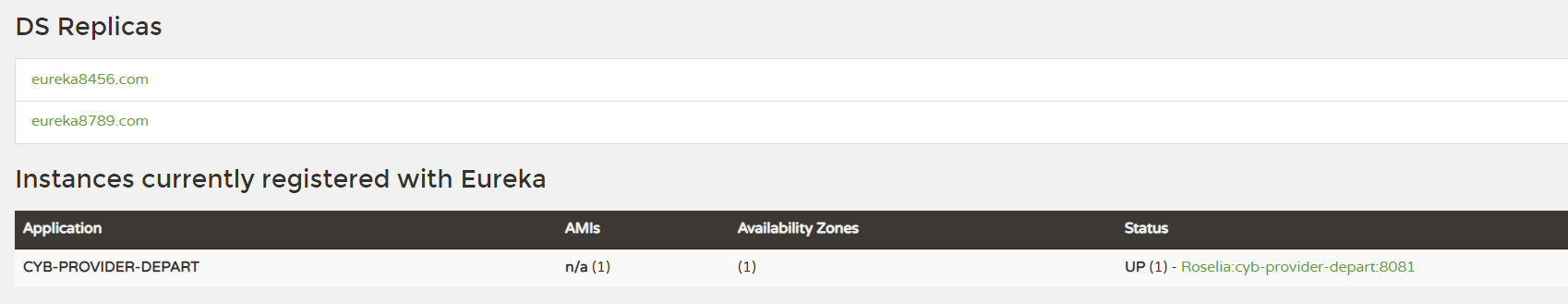


注：“,”隔开的中间不能有空格！！！集群中3个项目都要相应修改！！！

### 修改客户端配置



### 运行访问



# 声明式REST客户端OpenFeign

## 创建消费者工程

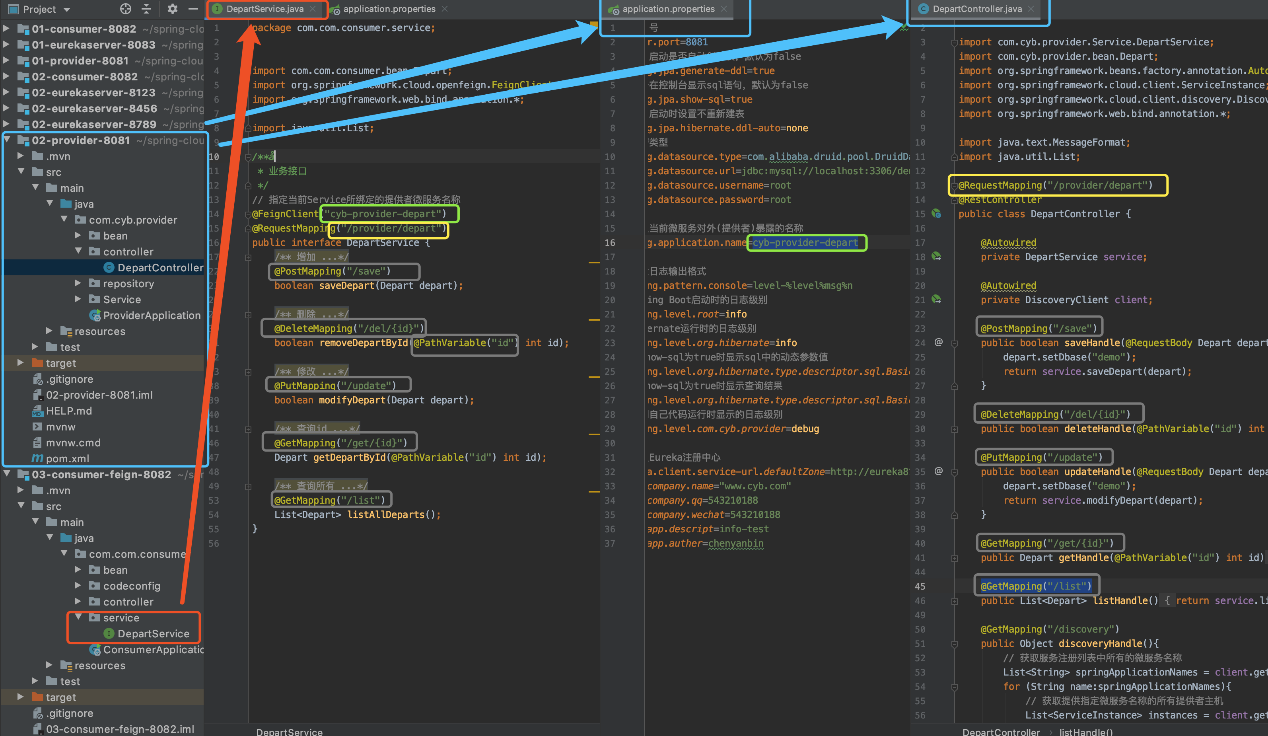
无需修改提供者工程，只需修改消费者工程即可。复制02-consumer-8082，并重命名为03-consumer-feign-8082。

* 添加openfeign依赖
* 定义Service接口，并指定其所绑定的微服务
* 修改处理器，通过Service接口消费微服务
* 在启动类上添加@EnableFeignClients注解

### 添加依赖

1. *<!-- openfeign依赖 -->*
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-openfeign</artifactId>
5. <version>2.2.2.RELEASE</version>
6. </dependency>

### 定义Service

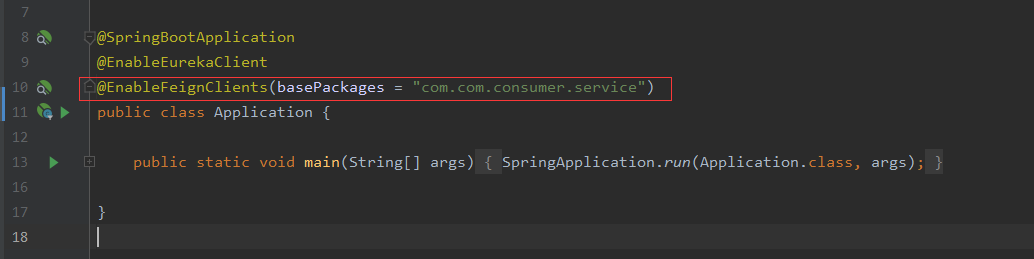


1. package com.com.consumer.server;
2. import com.com.consumer.bean.Depart;
3. import org.springframework.cloud.openfeign.FeignClient;
4. import org.springframework.web.bind.annotation.PathVariable;
5. import org.springframework.web.bind.annotation.RequestMapping;
6. import java.util.List;
7. */\*\**
8. \* 业务接口
9. \*/
10. *// 指定当前Service所绑定的提供者微服务名称*
11. @FeignClient("cyb-provider-depart")
12. @RequestMapping("/provider/depart")
13. public interface DepartService {
14. */\*\**
15. \* 增加
16. \* @param depart
17. \* @return
18. \*/
19. @RequestMapping("/save")
20. boolean saveDepart(Depart depart);
21. */\*\**
22. \* 删除
23. \* @param id
24. \* @return
25. \*/
26. @RequestMapping("/del/{id}")
27. boolean removeDepartById(@PathVariable("id") int id);
28. */\*\**
29. \* 修改
30. \* @param depart
31. \* @return
32. \*/
33. @RequestMapping("/update")
34. boolean modifyDepart(Depart depart);
35. */\*\**
36. \* 查询id
37. \* @param id
38. \* @return
39. \*/
40. @RequestMapping("/get/{id}")
41. Depart getDepartById(@PathVariable("id") int id);
42. */\*\**
43. \* 查询所有
44. \* @return
45. \*/
46. @RequestMapping("/list")
47. List<Depart> listAllDeparts();
48. }

### 修改处理器

1. package com.com.consumer.controller;
2. import com.com.consumer.bean.Depart;
3. import com.com.consumer.server.DepartService;
4. import org.springframework.beans.factory.annotation.Autowired;
5. import org.springframework.web.bind.annotation.PathVariable;
6. import org.springframework.web.bind.annotation.RequestBody;
7. import org.springframework.web.bind.annotation.RequestMapping;
8. import org.springframework.web.bind.annotation.RestController;
9. import java.util.List;
10. */\*\**
11. \* @author CrimsonTyrant
12. \* @version 1.0.0
13. \* @data 1/20/2021 14:00
14. \*/
15. @RestController
16. @RequestMapping("/consumer/depart")
17. public class DepartController {
18. @Autowired(required = false)
19. private DepartService departService;
20. @RequestMapping("/save")
21. public boolean saveHandle(Depart depart) {
22. return departService.saveDepart(depart);
23. }
24. @RequestMapping("/get/{id}")
25. public Depart getHandle(@PathVariable("id") int id) {
26. return departService.getDepartById(id);
27. }
28. @RequestMapping("/list")
29. public List<Depart> listHandle() {
30. return departService.listAllDeparts();
31. }
32. @RequestMapping("/update")
33. public boolean updateHandle(@RequestBody Depart depart) {
34. return departService.modifyDepart(depart);
35. }
36. @RequestMapping("/del/{id}")
37. public boolean deleteHandle(@PathVariable("id") int id) {
38. return departService.removeDepartById(id);
39. }
40. }

### 启动类上添加注解



### 测试

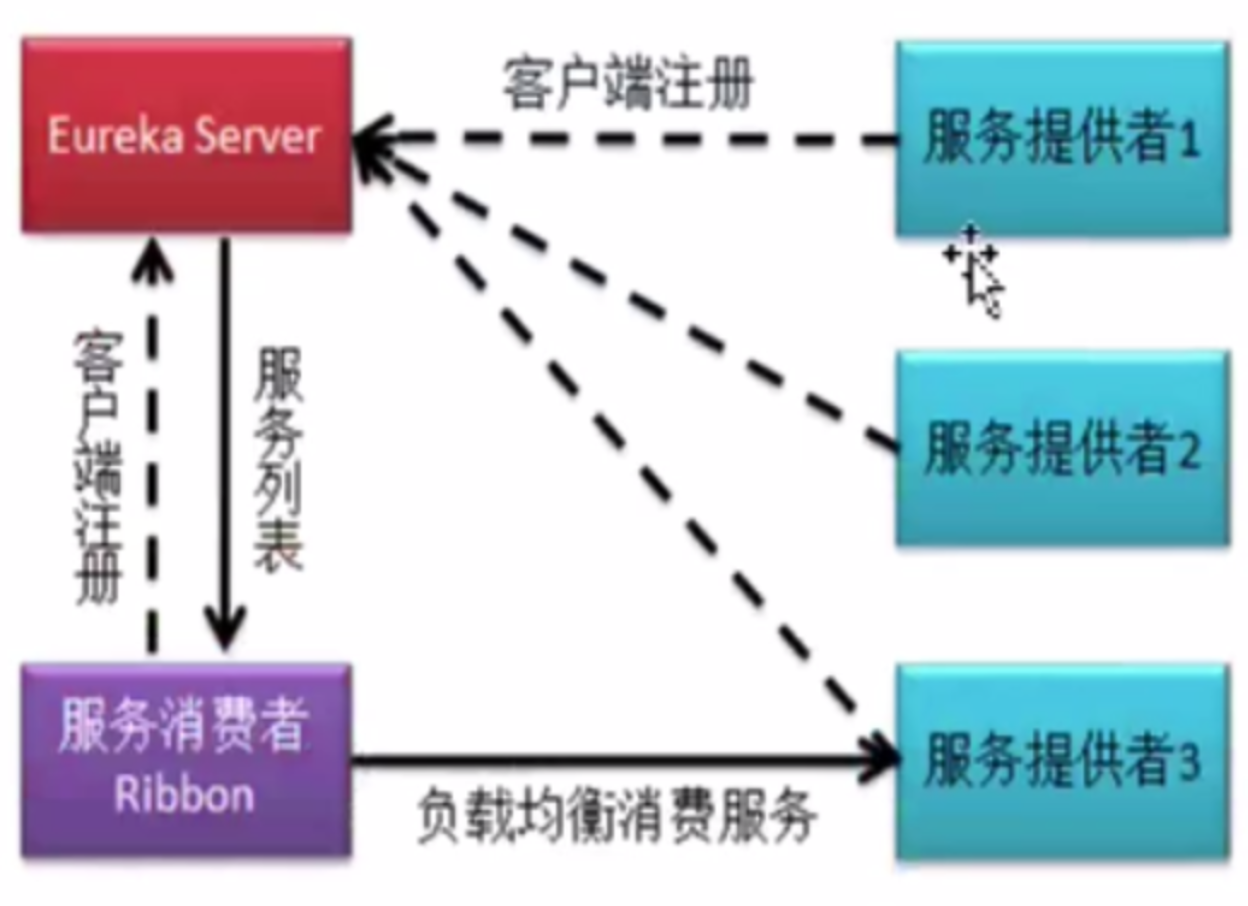
分别启动：01-eurekaserver-8083；02-provider-8081(需修改eureka注册中心地址)；03-consumer-feign-8082(需修改eureka注册中心地址)

# Ribbon负载均衡

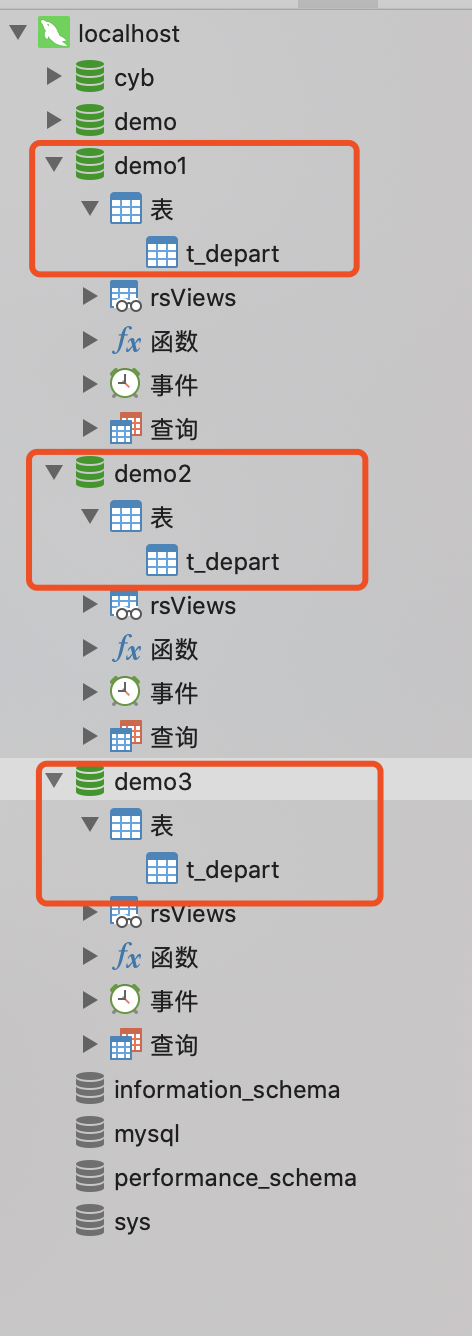
## Ribbo负载均衡演示

### 系统结构

负载均衡需要搭建出多个服务提供者，搭建系统如下：一个微服务由3个提供者提供，而消费者使用Ribbon对这3个提供者进行负载均衡访问。Ribbon首先会选择同一区域访问量较少的EurekaService，然后再从该EurekaServer中获取到服务列表，然后再根据用户指定的负载均衡策略选择一个服务提供者。

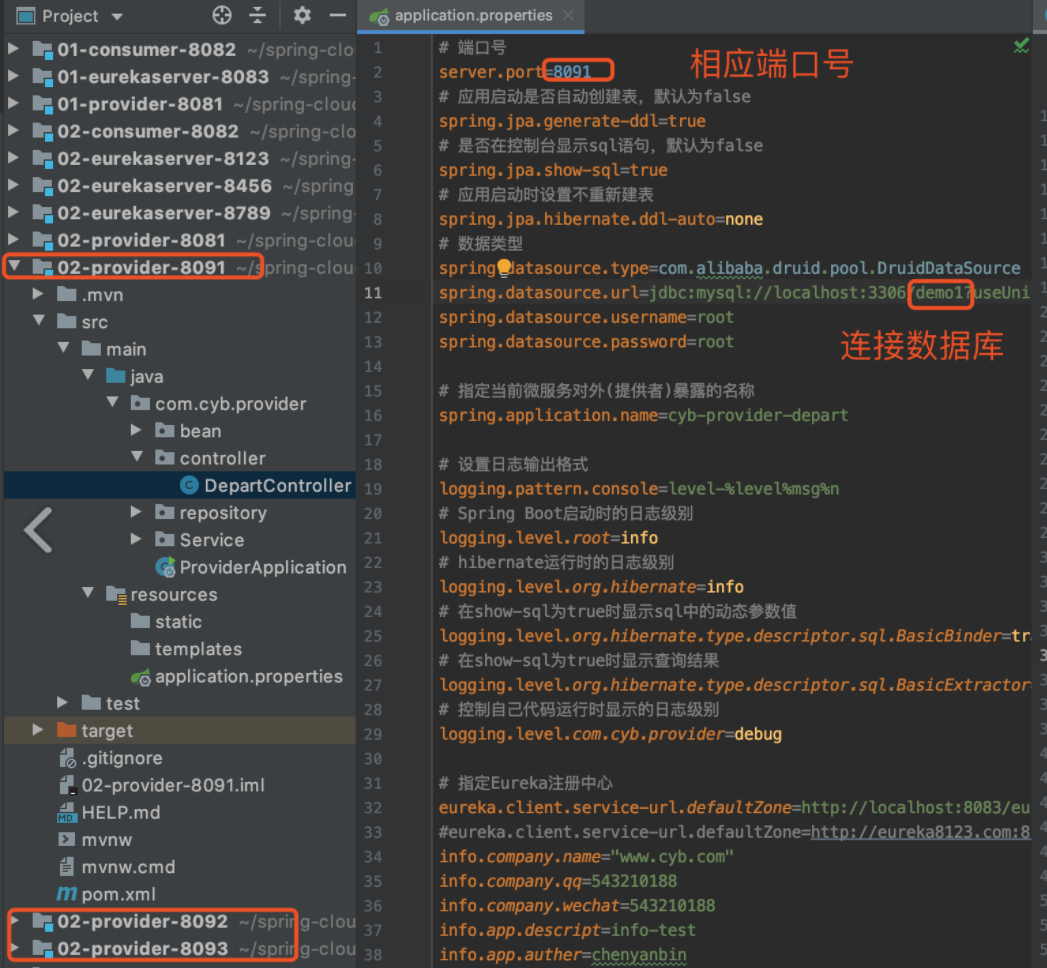


### 创建3个数据库



### 创建3个提供者

复制02-provider-8081，并重命名：02-provider-8091；02-provider-8092；02-provider-8093，修改相应端口号，连接的数据库等信息



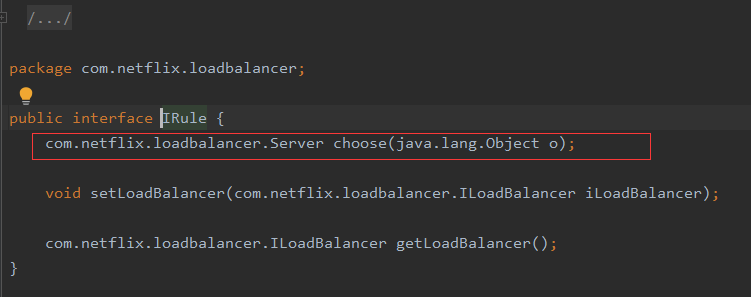
### 测试

启动依次启动：01-eurekaserver-8083；02-provider-8091；02-provider-8092；02-provider-8093；03-consumer-feign-8082。我们发现调用消费者的时候，消费者依次调用提供者1、提供者2、提供者3，这是因为默认采用负载均衡算法是轮询，他还支持其他的算法。

## 负载均衡算法IRule

Ribbon提供了多种负载均衡策略算法，例如轮询算法、随机算法、响应时间加权算法等。默认采用的是轮询算法，也可以指定Ribbon默认算法。

### IRule接口



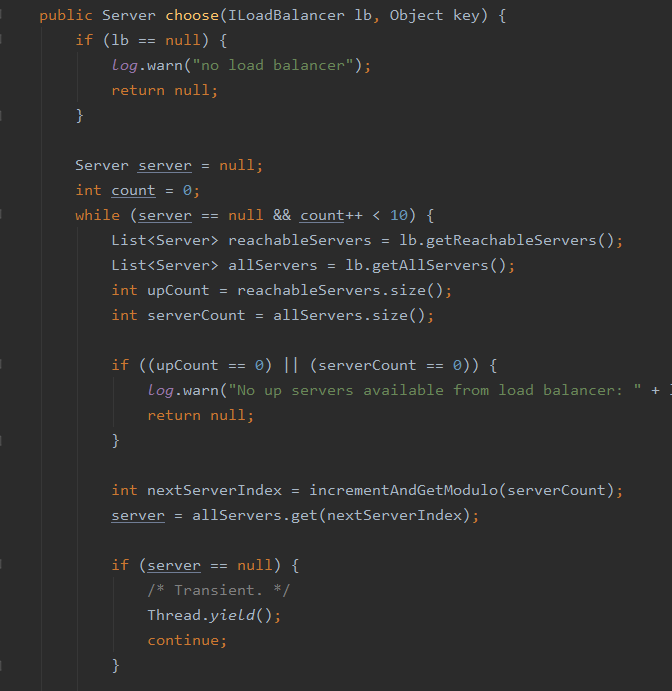
choose()方法：Ribbon的负载均衡算法需要实现IRule接口，而该接口中的核心方法即choose()方法，即对提供者的选择方式就是在该方法中体现的。

### Ribbon自带算法

Ribbon的内置可用负载均衡算法有七种。

#### RoundRobinRule

轮询策略。Ribbon默认采用的策略



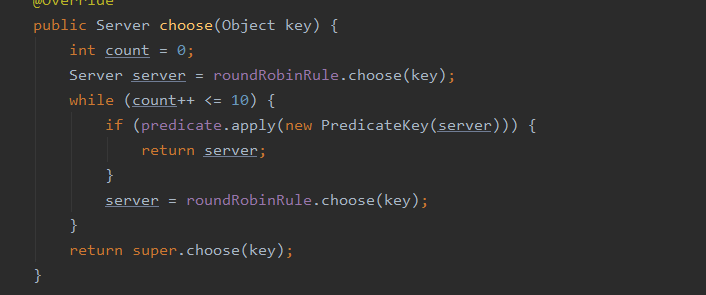
#### BestAvailableRule

选择并发量最小的provider，即连接的消费者数量最少的provider。其会遍历服务列表中的每一个provider，选择当前连接数量minimalConcurrentConnections最小的provider。



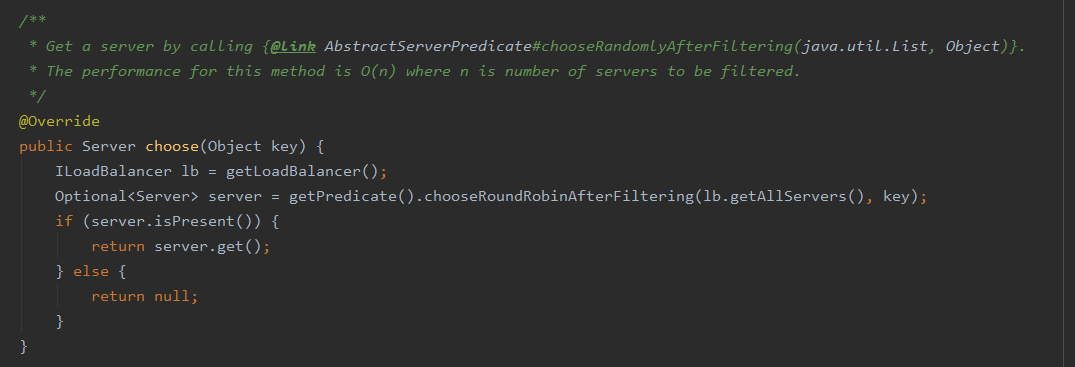
#### AvailabilityFilteringRule

过滤掉由于连续连接或读故障而处于短路器跳闸状态的provider，或已经超过连接极限的provider，对剩余provider采用轮询策略。



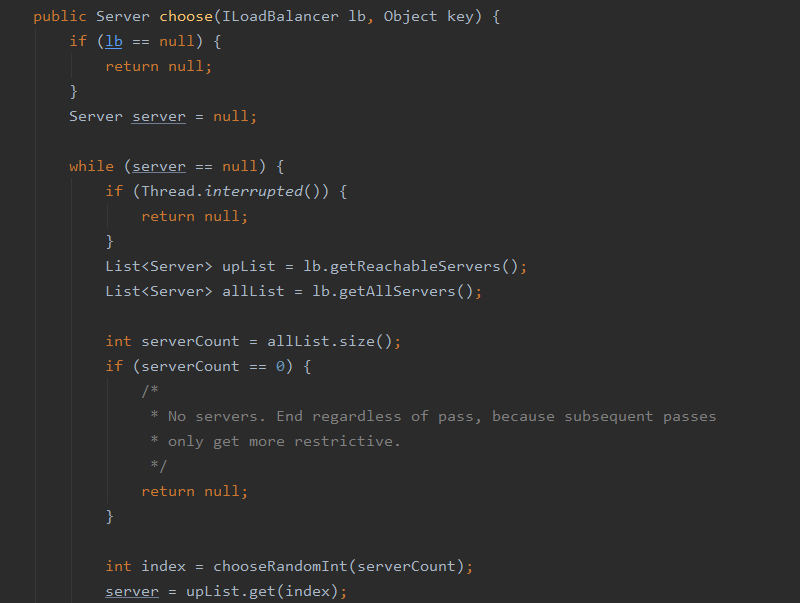
#### ZoneAvoidanceRule

复合判断provider所在区域的性能及provider的可用性选择服务器



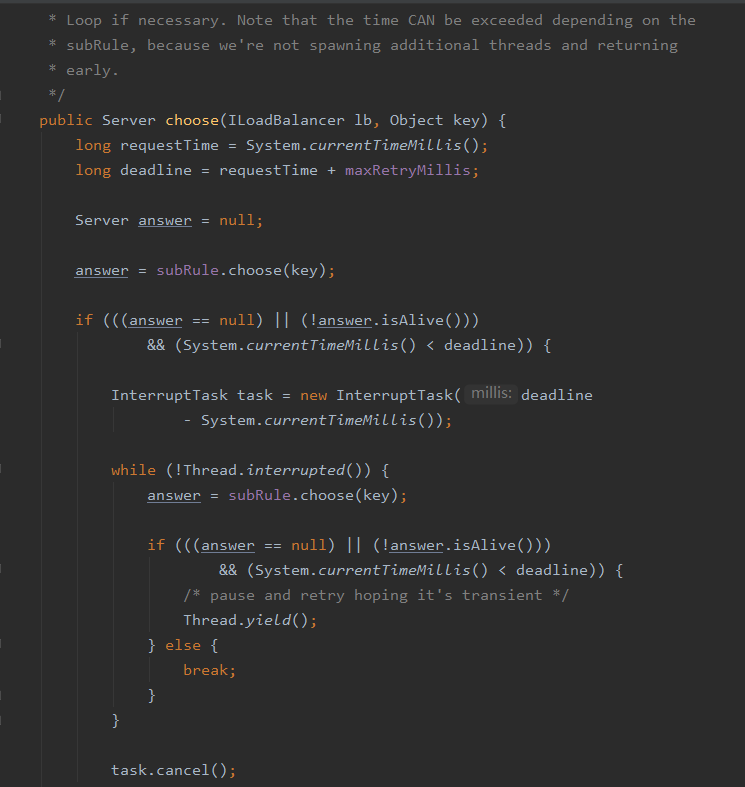
#### RandomRule

随机策略，从所有可用的provider中随机选一个。



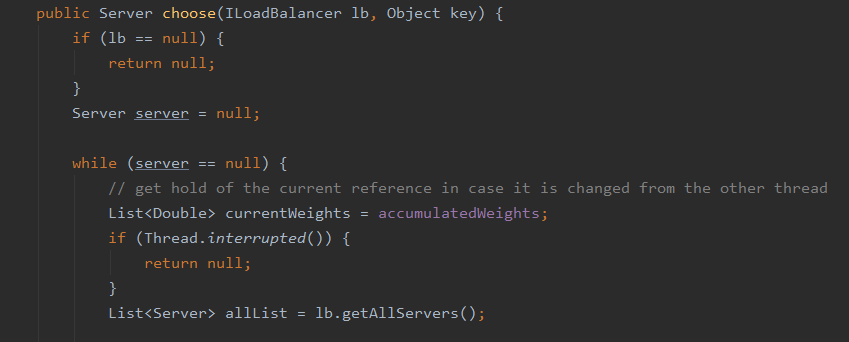
#### RetryRule

先按照RoundRobinRule策略获取provider，若后去失败，则在指定的时限内重试。默认的时限为500毫秒。



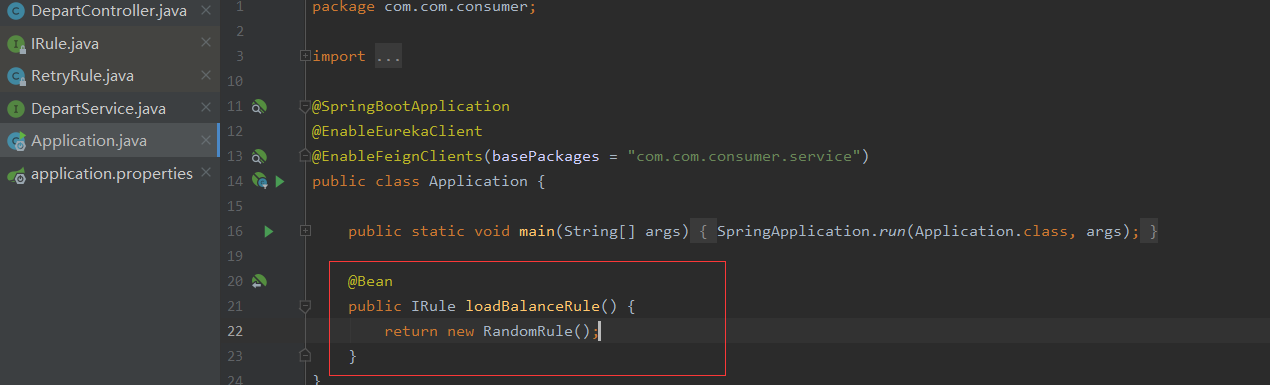
#### WeightedResponseTimeRule

权重响应时间策略，根据每个provider的平均响应时间计算其权重，响应时间越快权重越大，被选中的几率就越高，在刚启动时采用轮询策略，后面就会根据权重重新进行选择。



### 更改默认策略

Ribbon默认采用的是RoundRobinRule，即轮询策略。只需要在启动类中添加如下代码即可



### 自定义负载均衡策略

示例，该负载均衡策略的思路是：从所有可用的provider中排出掉指定端口号的provider，剩余provider进行随机选择。

CustomRule.java

1. package com.com.consumer.irule;
2. import com.netflix.loadbalancer.ILoadBalancer;
3. import com.netflix.loadbalancer.IRule;
4. import com.netflix.loadbalancer.Server;
5. import java.util.ArrayList;
6. import java.util.List;
7. import java.util.Random;
8. */\*\**
9. \* 自定义负载均衡算法
10. \* 从所有可用provider中排出掉指定端口号的provider，剩余provider进行随机选择
11. \*/
12. public class CustomRule implements IRule {
13. private ILoadBalancer lb;
14. */\*\**
15. \* 要排除提供者端口号集合
16. \*/
17. private List<Integer> excludePorts;
18. public CustomRule() {
19. }
20. public CustomRule(List<Integer> excludePorts) {
21. this.excludePorts = excludePorts;
22. }
23. @Override
24. public Server choose(Object key) {
25. *// 获取所有可用的提供者*
26. List<Server> servers = lb.getReachableServers();
27. *// 获取所有排出了指定端口号的提供者*
28. List<Server> availableServers = this.getAvailableServers(servers);
29. *// 从剩余的提供者中随机获取可用的提供者*
30. return this.getAvailableRandomServers(availableServers);
31. }
32. *// 获取所有排出了指定端口号的提供者*
33. private List<Server> getAvailableServers(List<Server> servers) {
34. *// 没有要排除的Server，则直接将所有可用Servers返回*
35. if (excludePorts == null || excludePorts.size() == 0) return servers;
36. *// 定义一个集合，用于存放排出了指定端口号的Server*
37. List<Server> aservers = new ArrayList<>();
38. boolean flag;
39. for (Server server : servers) {
40. flag = true;
41. for (Integer port : excludePorts) {
42. if (server.getPort() == port) {
43. flag = false;
44. break;
45. }
46. }
47. *// 若flag为false，说明上面的for循环执行了break，说明当前遍历的Server是要排除掉的*
48. if (flag) aservers.add(server);
49. }
50. return aservers;
51. }
52. *// 从剩余的提供者中随机获取可用的提供者*
53. private Server getAvailableRandomServers(List<Server> availableServers) {
54. *// 获取一个[0，availableServers.size()]的随机整数*
55. int index = new Random().nextInt(availableServers.size());
56. return availableServers.get(index);
57. }
58. @Override
59. public void setLoadBalancer(ILoadBalancer lb) {
60. this.lb = lb;
61. }
62. @Override
63. public ILoadBalancer getLoadBalancer() {
64. return lb;
65. }
66. }

# Hystrix 熔断机制与服务降级

## 简介

需要先了解雪崩效应与服务雪崩。

### 雪崩效应

分布式系统中很容易出现雪崩效应。在IO型服务中，假设服务A依赖服务B和服务C，而B服务和C服务有可能依赖其他的服务，继续下去会使得调用链路过长，技术上称1->N扇出。



如果在A的链路上某个或几个被调用的子服务不可用或延迟较高，则会导致调用A服务的请求被堵住。堵住的A请求会消耗占用系统的进程、IO等资源，当对A服务的请求越来越多，占用的计算机资源越来越多，会导致系统瓶颈出现，造成其他的请求同样不可用，最终导致业务系统崩溃，这种现象称为雪崩效应。

例如一个汽车生产线，生产不同的汽车，需要使用不同的零件。如果某个零件因为种种原因无法及时供给，而没有该零件，则后续的好多已经到货的零件也无法安装。一个零件的缺失造成整台车无法装配，陷入等待零件的状态，直到零件到位，才能继续组装。此时如果有很多个车型都需要这个零件，那么整个工厂都将陷入等待的状态，而前述已经生成好多的汽车部件，暂不能安装的其他零件，将由于等待而占用大量资金、场地等资源。一个零件最终导致所有生产陷入瘫痪，这就是雪崩效应。

### 服务雪崩

雪崩效应发生在分布式SOA(Service-Oriented Architecture，面向服务的架构)系统中，则称为服务雪崩。大量用户请求出现异常全部陷入阻塞的情况，即服务发生雪崩的情况。举个例子，一个依赖30个微服务的系统，每个服务99.99%可用。则整个系统的可用性为99.99%的30次方，约为99.7%。为什么是30次方呢？若系统依赖于2个微服务，一个微服务的可用率为99.99%，那么，两个微服务的组合的可用率为99.99%\*99.99%，同理，30个微服务，每个微服务的可用率为99.99%，则这30个微服务组合后的可用性为99.99%的30次方。也就是说，整个系统会存在0.3%的失败率。若存在一亿次请求，那么将会有30万次失败。

### 熔断机制

熔断机制是服务雪崩的一种有效解决方案。当服务消费者所请求的提供者暂不能提供服务时，消费者会被阻塞，且长时间占用请求链路。为了防止这种情况的发生，当在设定阈值限制到达时，仍未获得提供者的服务，则系统将通过断路器直接将此请求链路断开。这种像熔断“保险丝”一样的解决方案称为熔断机制。

## Hystrix简介

https://github.com/Netflix/Hystrix

## 服务降级简介

在访问分布式系统中，经常会发生以下两种情况：

* 当整个微服务架构整体的负载超出了预设的上限阈值，或即将到来的流量预计将会超过预设的阈值时，为了保证重要或基本的服务能正常运行，我们可以将一些不重要或不紧急的服务进行延迟使用或暂停使用。这就是服务熔断，类似于主动拉电闸的服务熔断。此时，若有消费者消费这些延迟/暂停使用的服务则会出现阻塞，等待提供者的响应。
* 当消费者访问某微服务时，由于网络或其他原因，提供者向消费者响应过慢，出现服务超时或根本就没有响应时，这也是一种服务熔断，类似于保险丝自动熔断的服务熔断。此时消费者会被迫阻塞，等待提供者的响应。

在发生服务熔断时，不仅用户体验很差，其还占用了大量的系统资源。为了解决这个问题，在编写消费者端代码时就设置了预案：在消费者端给出一种默认的、临时的预处理方案，能够给出消费者一个可以接受的结果。即，对于用户(指的是人，并非指消费者端)来说，其所消费的服务并非由应当提供服务的提供者端给出，而是由服务消费者临时给出，服务质量降级了。提供者端的“服务熔断”与消费者端的“本地服务”，共同构成了“服务降级”。

## Hystrix服务降级

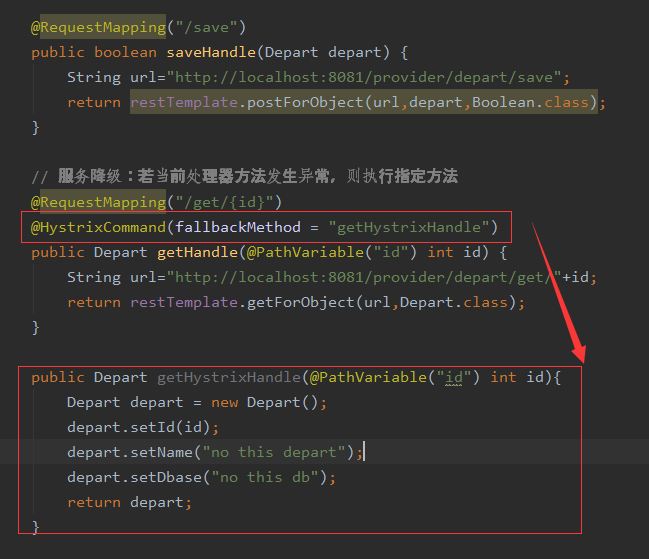
* 添加hystrix依赖
* 修改处理器，在处理器方法上添加@HystrixCommond注解，并添加处理方法
* 在启动类上添加@EnableCircuitBreaker注解

### 创建消费者工程

复制02-consumer-8082，并重命名04-consumer-hystrix-8082，添加依赖：

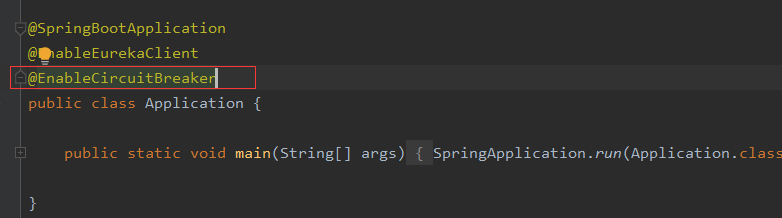
1. *<!-- hystrix依赖 -->*
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
5. <version>2.2.2.RELEASE</version>
6. </dependency>

修改处理器：



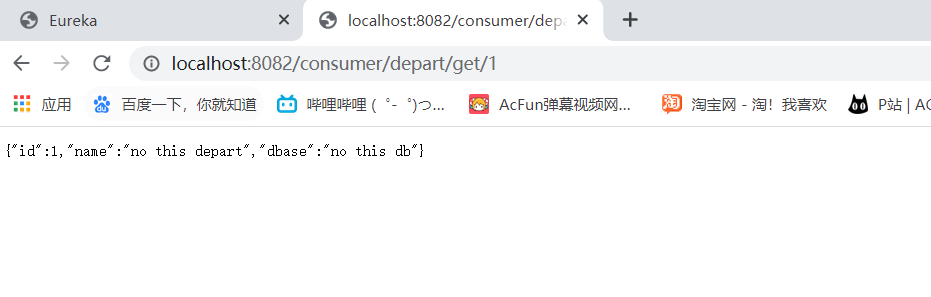
注：实际中一个方法对应一个处理函数

启动类上添加注解：



### 测试

为了演示方法，只开启eureka注解中心和消费者



## Hystrix+Feign服务降级

* 在Feign接口所在包下定义降级处理类
* 在Feign接口中指定要使用的降级处理类
* 在配置文件中开启Feign对Hystrix的支持

### 创建消费者工程

复制03-consumer-feign-8082并重命名04-consumer-feign-hystrix-8082

### 定义降级处理类

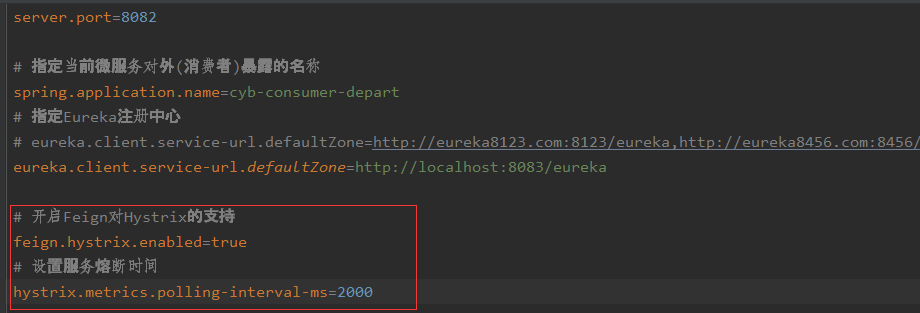
降级处理类需要实现FallbackFactory接口，该接口的泛型为Feign接口。该类可以定义在任意包下，不过，一般会与Feign解口定义在同一包下。该类需要使用@Component注解，表示要将其交给Spring容器来管理。

1. package com.com.consumer.service;
2. import com.com.consumer.bean.Depart;
3. import feign.hystrix.FallbackFactory;
4. import org.springframework.stereotype.Component;
5. import java.util.List;
6. */\*\**
7. \* 降级处理类
8. \*/
9. @Component
10. public class DepartFallbackFactory implements FallbackFactory<DepartService> {
11. @Override
12. public DepartService create(Throwable throwable){
13. return new DepartService() {
14. @Override
15. public boolean saveDepart(Depart depart) {
16. System.out.println("执行saveDepart的服务降级处理方法");
17. return false;
18. }
19. @Override
20. public boolean removeDepartById(int id) {
21. System.out.println("执行removeDepartById的服务降级处理方法");
22. return false;
23. }
24. @Override
25. public boolean modifyDepart(Depart depart) {
26. System.out.println("执行modifyDepart的服务降级处理方法");
27. return false;
28. }
29. @Override
30. public Depart getDepartById(int id) {
31. System.out.println("执行getDepartById的服务降级处理方法");
32. return null;
33. }
34. @Override
35. public List<Depart> listAllDeparts() {
36. System.out.println("执行listAllDeparts的服务降级处理方法");
37. return null;
38. }
39. };
40. }
41. }

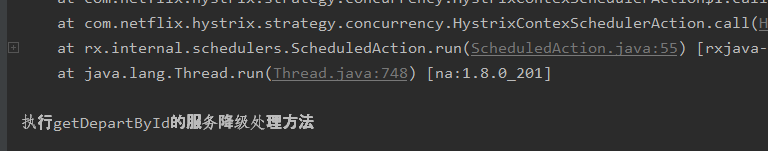
### 接口类上指定降级处理器类



### 修改配置文件



### 测试



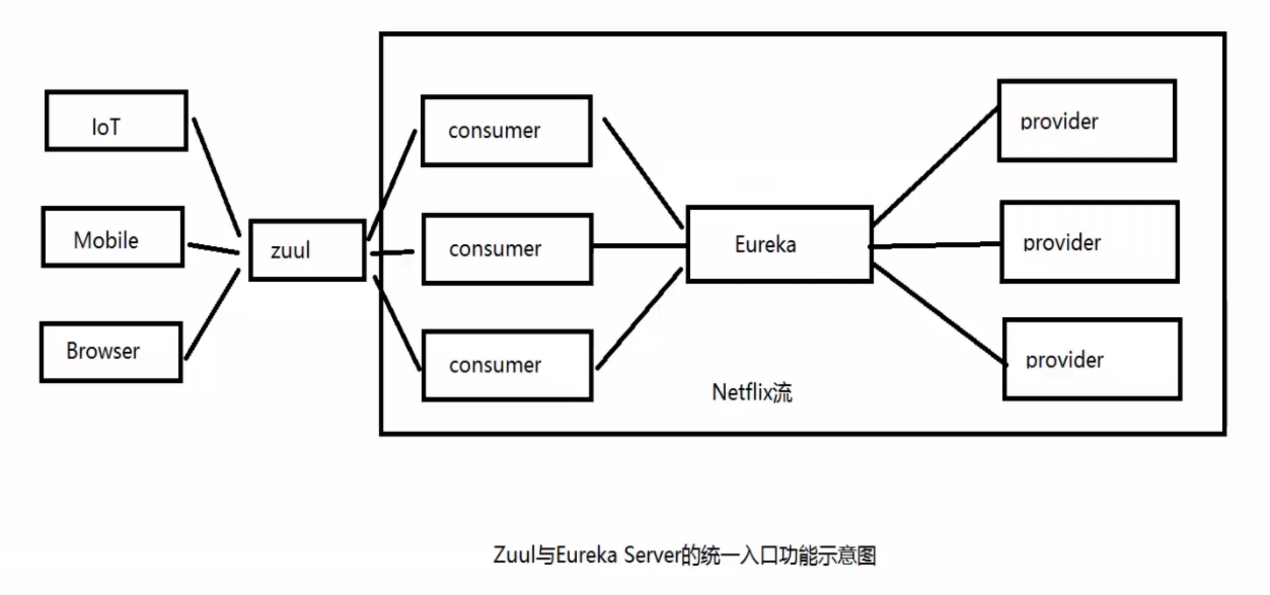
注：方法级别的优先级小于类级别的优先级

# 网关服务Zuul

<https://github.com/Netflix/zuul>

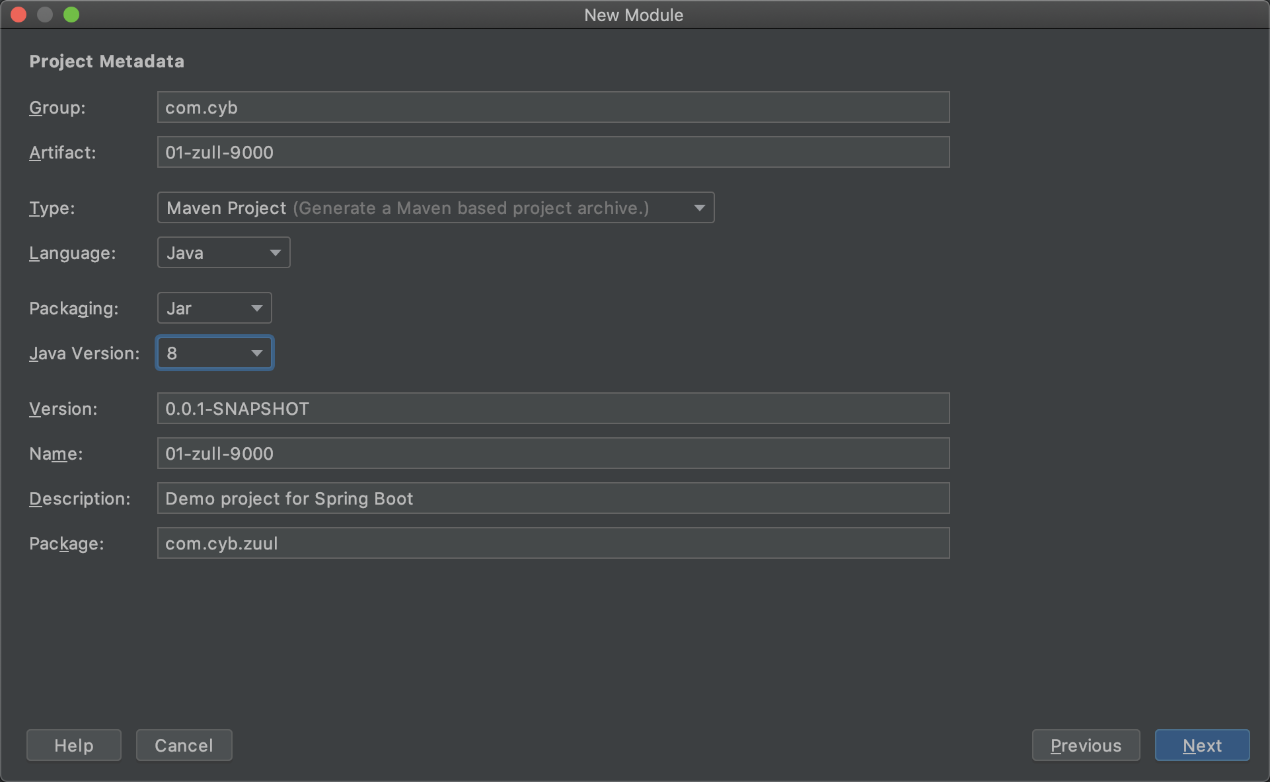
## 简介

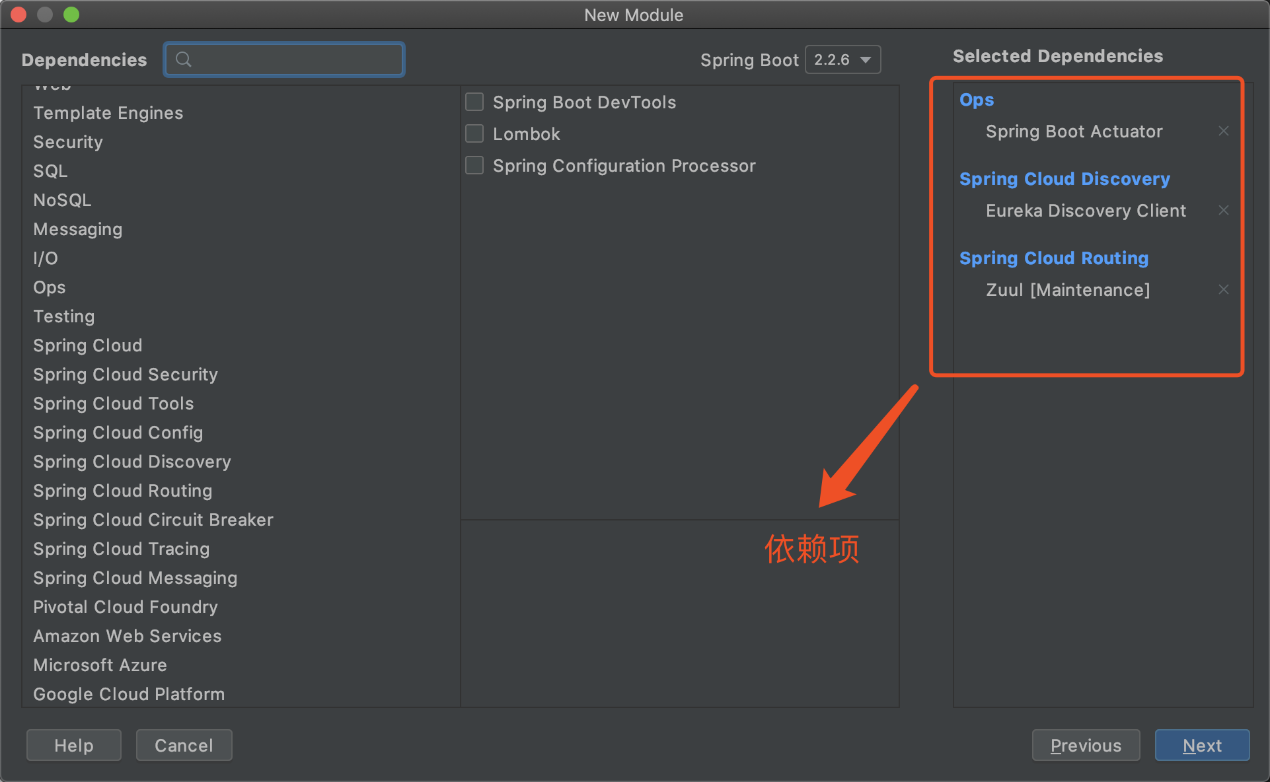
Zuul主要提供了对请求的路由有过滤功能。路由功能主要指，将外部请求转发到具体的微服务实例上，是外部访问微服务的统一入口。过滤功能主要指，对请求的处理过程进行干预，对请求进行校验、服务聚合等处理。Zuul与Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，从Eureka Server中获取到其他微服务信息，使外部对于微服务的访问都是通过Zull进行转发的。



## 基本用法

### 创建zuul网关服务器

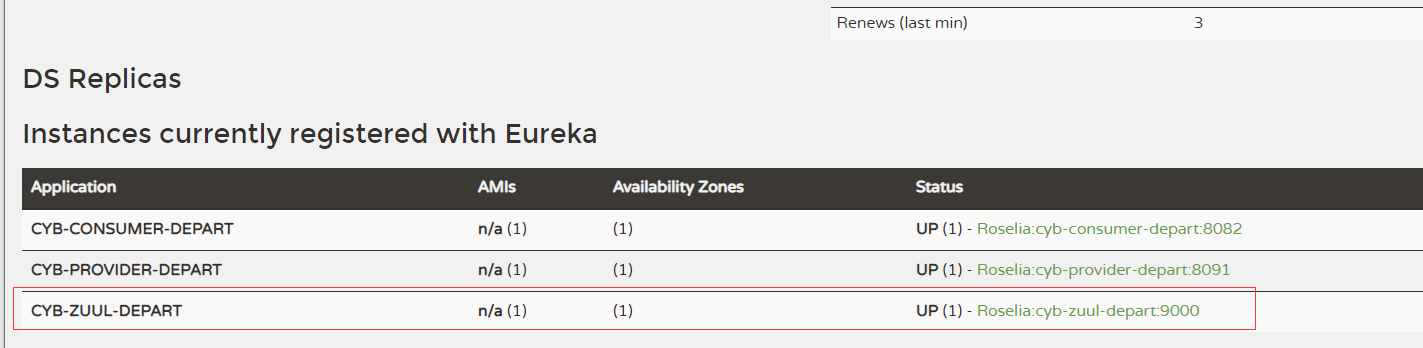


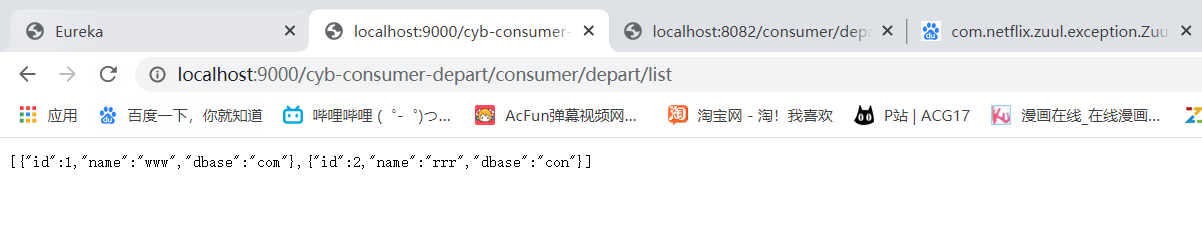


### 修改配置文件



### 启动测试





### 设置zull路由映射规则

为了保护和隐藏服务名称，可以为其配置一个映射路径，将这个映射路径暴露给消费者即可。

1. server.port=9000
2. *# 指定Eureka注册中心*
3. eureka.client.service-url.defaultZone=http://localhost:8083/eureka
4. spring.application.name=cyb-zuul-depart
5. *# zuul：设置zuul路由规则*
6. *# somedepart.service-id：指定要替换的微服务名称*
7. zuul.routes.somedepart.service-id=cyb-consumer-depart
8. *# 指定替换使用的路径*
9. zuul.routes.somedepart.path=/cyb/\*\*

对于该配置需要注意以下几点：

* somedepart：可以随意命名，但service-id与path是关键字，不能更改
* somedepart.service-id：指定要被替换掉的微服务名称
* somedepart.path：指定用于替换指定微服务名称的路径

设置过zuul路由规则后，两种方式，一样可以访问。



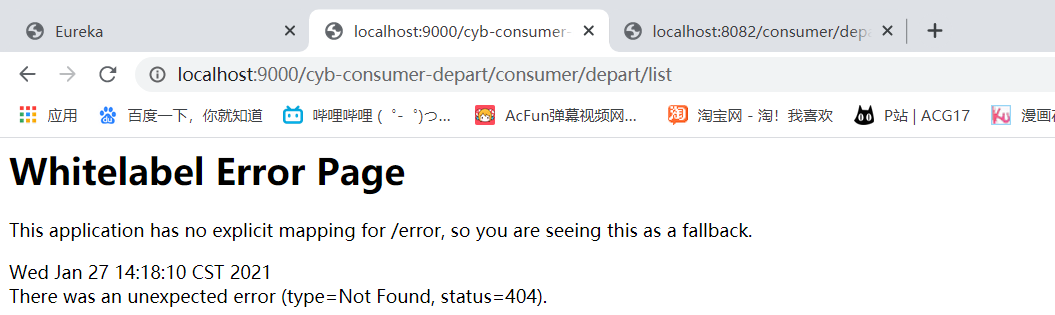
## 忽略服务名称

以上配置虽然可以使用映射路径访问微服务，但是通过原来的服务名称仍可以访问到微服务，即以上配置并没有隐藏和保护了原来的微服务名称。可以在配置文件中设置忽略微服务属性，替换原有的微服务名称使用。两种方式：1、忽略指定微服务；2、忽略所有微服务

### 忽略指定微服务名称

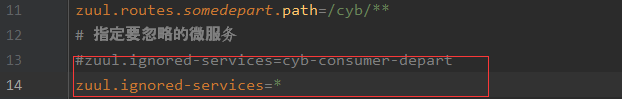
在配置文件中指定要忽略的微服务







### 忽略所有微服务名称



注：效果和上面忽略指定的微服务是一样的！

### 为映射路径配置统一前缀

一般情况下我们会在映射路径前添加一个前缀用于表示模块信息或公司名称等，而该前缀对于各个微服务来说一般都是需要的，所以我们可以为映射路径统一配置前缀。

1. server.port=9000
2. *# 指定Eureka注册中心*
3. eureka.client.service-url.defaultZone=http://localhost:8083/eureka
4. spring.application.name=cyb-zuul-depart
5. *# zuul：设置zuul路由规则*
6. *# somedepart.service-id：指定要替换的微服务名称*
7. zuul.routes.somedepart.service-id=cyb-consumer-depart
8. *# 指定替换使用的路径*
9. zuul.routes.somedepart.path=/cyb/\*\*
10. *# 指定要忽略的微服务*
11. *# zuul.ignored-services=cyb-consumer-depart*
12. *# 忽略所有的微服务*
13. zuul.ignored-services=\*
14. *# 指定访问的统一前缀*
15. zuul.prefix=/test

