

- - 学习目标
  - 三、Spring Cloud
    - 3.1 Spring Cloud简介
    - 3.2 Spring Cloud的版本
    - 3.3 SpringCloud与SpringBoot版本匹配关系
    - 3.4 SpringCloud 架构
      - 3.4.1 SpringCloud 组件
      - 3.4.2 SpringCloud常用架构
  - 四、注册中心 Spring Cloud Eureka
    - 4.1 Eureka 简介
    - 4.2 常见的注册中心
    - 4.3 架构图
    - 4.4 整合注册中心Eureka
      - 4.4.1 搭建eureka-server工程
      - 4.4.2 服务提供者-注册服务中心
      - 4.4.3 服务消费者-注册服务中心
      - 4.4.4 Eureka元数据
        - 标准元数据
        - 自定义元数据(了解)
      - 4.4.5 消费者通过Eureka访问提供者
    - 4.5 Eureka详解
      - 4.5.1 基础架构
      - 4.5.2 Eureka客户端
        - 服务注册过程:
        - 服务续约过程:
        - 获取服务列表:
      - 4.5.3 Eureka服务端
        - 服务下线:
        - 失效剔除:
        - 自我保护:
    - 4.6 Eureka高可用集群
    - 4.7 Eureka服务端源码
  - 五、负载均衡 Spring Cloud Ribbon

- 5.1 Ribbon 简介
- 5.2 入门案例
- 5.3 负载策略
- 5.4 Ribbon源码解析
  - 5.4.1 Ribbon关键组件
  - 5.4.2 源码分析
- 六、熔断器 Spring Cloud Hystrix
  - 6.1 Hystrix 简介
  - 6.2 雪崩效应
  - 6.3 熔断案例
  - 6.4 熔断原理分析
  - 6.5 扩展-服务降级的fallback方法：
  - 6.6 Hystrix Dashboard监控平台
- 总结：

## 学习目标

---

1. 能够理解SpringCloud作用
2. 能够搭建Eureka注册中心服务
3. 能够理解Ribbon负载均衡的作用
4. 能够测试Ribbon负载均衡访问效果：随机和轮询
5. 能够理解Hystrix熔断器的作用
6. 能够写出Hystrix熔断器服务降级方法

## 三、Spring Cloud

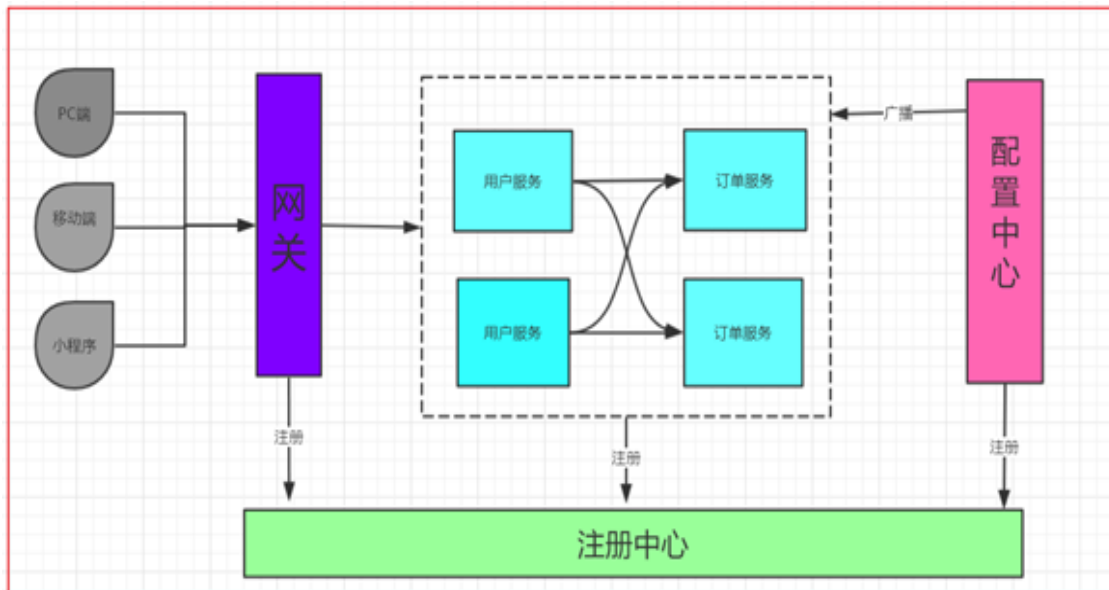
---

大家谈起的微服务，大多来讲说的只不过是种架构方式。其实现方式很多种：Spring Cloud，Dubbo，华为的Service Combo(已经捐给Apache，成为Apache的顶级项目)，Istio.....。那么这么多的微服务架构产品中，我们为什么要用Spring Cloud？因为它后台硬、技术强、群众基础好，使用方便；

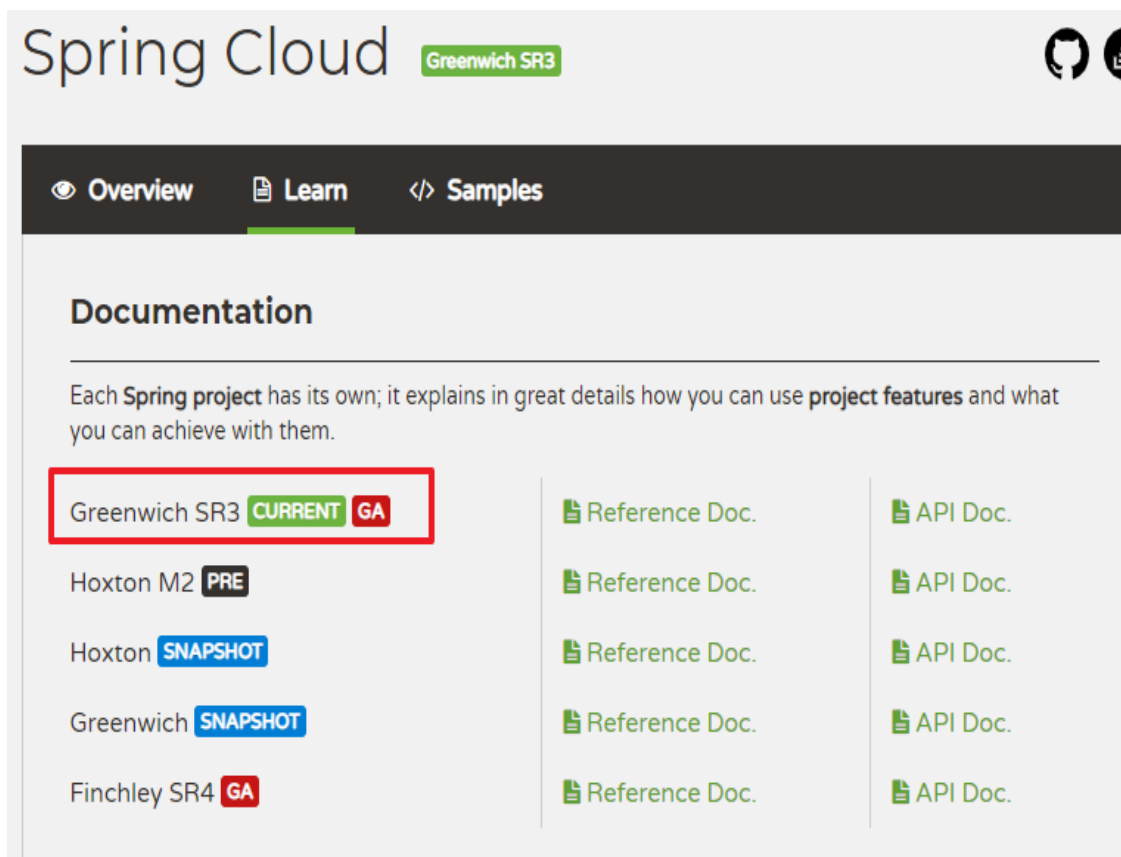
### 3.1 Spring Cloud简介

**Spring Cloud**从技术架构上降低了对大型系统构建的要求和难度，使我们以非常低的成本（技术或者硬件）搭建一套高效、分布式、容错的平台，但Spring Cloud也不是没有缺点，小型独立的项目不适合使用。

Spring Cloud是一系列分布式微服务技术的有序整合，把非常流行的微服务的技术整合到一起。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。



### 3.2 Spring Cloud的版本



- SpringCloud是一系列框架组合，为了避免与框架版本产生混淆，采用新的版本命名方式，形式为大版本名+子版本名称
- 大版本名用伦敦地铁站名：
- 子版本名称三种
  - SNAPSHOT：快照版本，尝鲜版，随时可能修改
  - M版本，Milestone，M1表示第一个里程碑版本，一般同时标注PRE，表示预览版
  - SR，Service Release，SR1表示第一个正式版本，同时标注GA(Generally Available)，稳定版

### 3.3 SpringCloud与SpringBoot版本匹配关系

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

## 3.4 SpringCloud 架构

### 3.4.1 SpringCloud 组件

Spring Cloud的本质是在 Spring Boot 的基础上，增加了一堆微服务相关的规范，并对应用上下文（Application Context）进行了功能增强。既然 Spring Cloud 是规范，那么就需要去实现，目前 Spring Cloud 规范已有 Spring官方，Spring Cloud Netflix，Spring Cloud Alibaba等实现。通过组件化的方式，Spring Cloud将这些实现整合到一起构成全家桶式的微服务技术栈。

#### Spring Cloud Netflix组件

组件名称	作用
Eureka	服务注册中心
Ribbon	客户端负载均衡
Feign	声明式服务调用
Hystrix	熔断器
Zuul	API服务网关

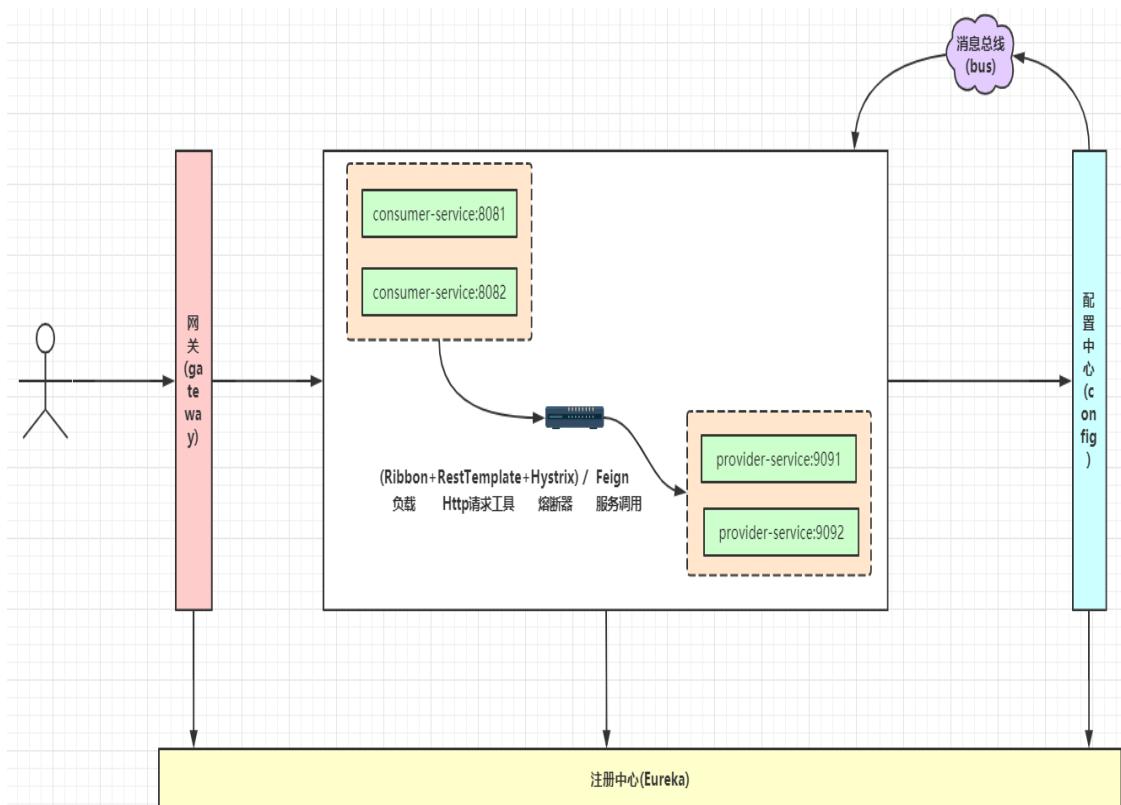
#### Spring Cloud Alibaba组件

组件名称	作用
Nacos	配置中心、注册中心
Sentinel	客户端容错保护

### Spring Cloud原生及其他组件

组件	作用
Consul	服务注册中心
Config	分布式配置中心
Gateway	API服务网关
Sleuth/Zipkin	分布式链路追踪

### 3.4.2 SpringCloud常用架构



- 网关：负责转发所有的对外请求服务;
- 注册中心：负责服务的注册与发现，很好的将服务连接起来;
- 配置中心：提供统一的配置信息管理服务，可以实时的通知各个服务获取最新的配置信息;

## 四、注册中心 Spring Cloud Eureka

---

### 4.1 Eureka 简介

Eureka解决了第一个问题：服务的管理，注册和发现、状态监管、动态路由。

Eureka负责管理记录服务提供者的信息。服务调用者无需自己寻找服务，Eureka自动匹配服务给调用者。

Eureka与服务之间通过 **心跳 机制** 进行监控。

### 4.2 常见的注册中心

#### Zookeeper

zookeeper它是一个分布式服务框架，是Apache Hadoop 的一个子项目，它主要是用来解决分布式应用中经常遇到的一些数据管理问题，如：统一命名服务、状态同步服务、集群管理、分布式应用配置项的管理等。简单来说 **zookeeper=文件系统+监听通知机制**。

#### Eureka

Eureka是在Java语言上，基于Restful Api开发的服务注册与发现组件，Springcloud Netflix中的重要组件。

#### Consul

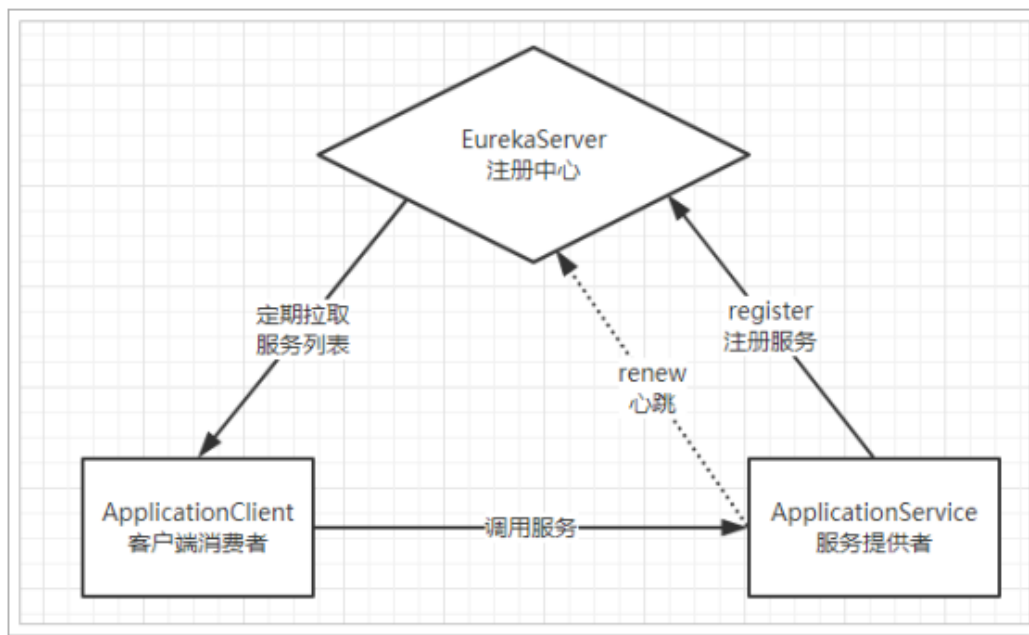
Consul是由HashiCorp基于Go语言开发的支持多数据中心分布式高可用的服务发布和注册服务软件，采用Raft算法保证服务的一致性，且支持健康检查。

#### Nacos

Nacos是一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。简单来说 Nacos 就是注册中心 + 配置中心的组合，提供简单易用的特性集，帮助我们解决微服务开发必会涉及到的服务注册与发现，服务配置，服务管理等问题。Nacos 还是 Spring Cloud Alibaba 组件之一，负责服务注册与发现。

## 4.3 架构图

基本架构图



Eureka：就是**服务注册中心**

- 服务提供者(ApplicationService)：启动后向Eureka注册自己的信息(地址，提供什么服务)。
- 服务消费者(ApplicationClient)：向Eureka订阅服务，Eureka会将对应服务的所有提供者地址列表发送给消费者，并且定期更新。
- 心跳(续约)：提供者定期通过http方式向Eureka刷新自己的状态。

## 4.4 整合注册中心Eureka

步骤：



- 第一步：搭建工程eureka-server
- 第二步：改造服务提供者provider-service,注册到eureka-server服务
- 第三步：改造服务消费者consumer-service,注册到eureka-server服务.并调用服务

#### 4.4.1 搭建eureka-server工程

步骤：

1. 创建eureka-server的springboot工程
2. 勾选坐标
3. 编写配置文件application.yml
4. 在启动类EurekaServerApplication声明当前应用为Eureka服务使用 `@EnableEurekaServer` 注解
5. 启动EurekaServerApplication
6. 测试访问地址http://127.0.0.1:8761

过程：

1. 创建eureka-server的springboot工程。

New Project X

**Project Metadata**

Group: com.itheima

Artifact: eureka-server

Type: Maven Project (Generate a Maven based project archive) ▾

Language: Java ▾

Packaging: Jar ▾

Java Version: 8 ▾

Version: 0.0.1-SNAPSHOT

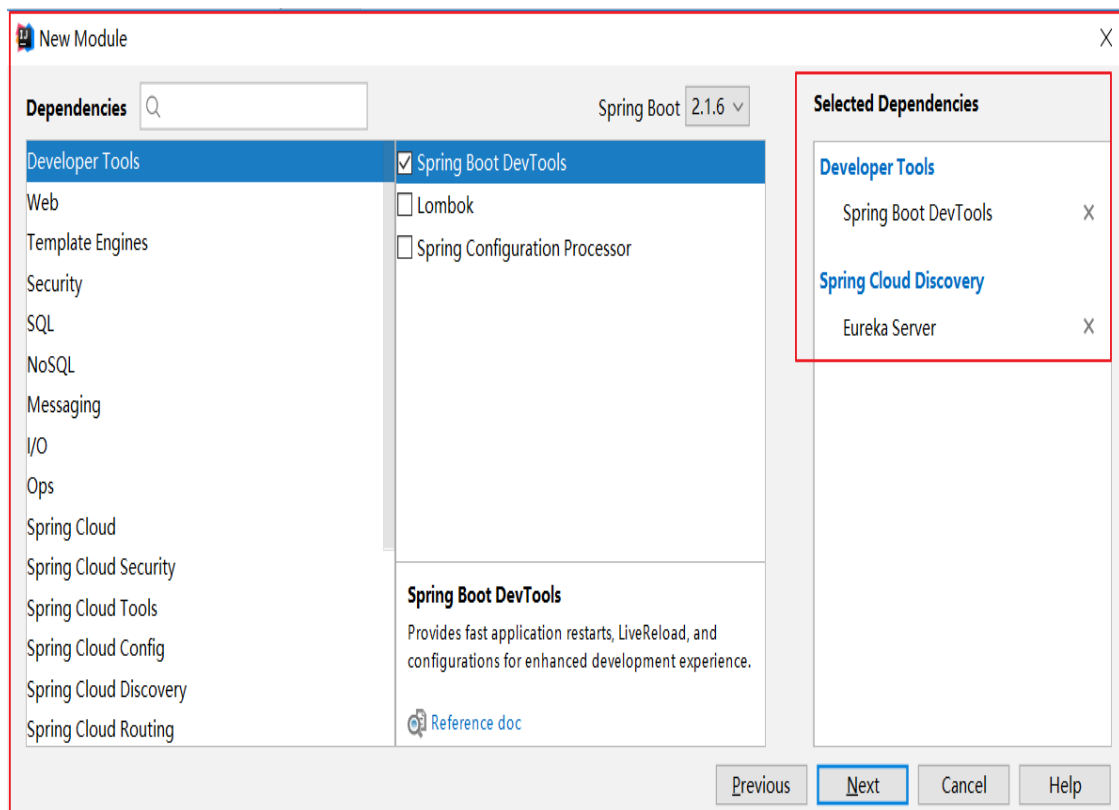
Name: eureka-server

Description: Demo project for Spring Boot

Package: com.itheima

Previous Next Cancel Help

## 2. 勾选坐标



### 3. 编写配置文件application.yml

```
# 端口
server:
  port: 8761
# 应用名称, 会在Eureka中作为服务的id标识 (serviceId)
spring:
  application:
    name: eureka-server
# EurekaServer的地址, 现在自己的地址, 如果是集群, 需要写其它Server的地址。
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:8761/eureka
    fetch-registry: false #是否抓取注册列表
    register-with-eureka: false #是否注册服务中心Eureka
```

4. 在启动类EurekaServerApplication声明当前应用为Eureka服务使用 `@EnableEurekaServer` 注解
5. 启动EurekaServerApplication
6. 测试访问地址http://127.0.0.1:8761, 如下信息代表访问成功

System Status

Environment	test	Current time	2020-08-13T15:56:32 +0800
Data center	default	Uptime	00:00
	租约到期启用	Lease expiration enabled	false
	期望每分钟收到的续约次数	Renews threshold	1
	上一分钟收到的续约次数	Renews (last min)	0

DS Replicas

Discovery Server Replicas  
发现服务集群节点

127.0.0.1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory 总有效内存	746mb
environment	test
num-of-cpus	12
current-memory-usage	416mb (55%)
server-uptime	00:00
registered-replicas 注册的副本	http://127.0.0.1:8761/eureka/
unavailable-replicas 无效副本	http://127.0.0.1:8761/eureka/
available-replicas 有效副本	

Instance Info

Name	Value
ipAddr	192.168.1.103
status	UP

## 4.4.2 服务提供者-注册服务中心

步骤：

1. 在服务提供者provider-service工程中添加springcloud依赖管理和Eureka客户端依赖坐标
2. 在启动类上开启Eureka客户端发现功能 @EnableDiscoveryClient
3. 修改配置文件：spring.application.name指定应用名称，作为服务ID使用，并配置eureka注册中心地址

过程：

1. 在服务提供者provider-service工程中添加springcloud依赖管理和Eureka客户端依赖坐标

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

<!--SpringCloud所有依赖管理的坐标-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. 在启动类上开启Eureka客户端发现功能 @EnableDiscoveryClient

```
@SpringBootApplication
@EnableDiscoveryClient // 开启Eureka客户端发现功能
public class UserApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserApplication.class,args);
    }
}
```

3. 修改配置文件：spring.application.name指定应用名称，作为服务ID使用，并配置eureka注册中心地址

```

server:
  port: 9091
#db配置
spring:
  application:
    name: provider-service
  datasource:
    url: jdbc:mysql://127.0.0.1:3306/itheima?
    useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
    username: root
    password: sang,1230
    driver-class-name: com.mysql.cj.jdbc.Driver
#mybatis配置
mybatis:
  #实体位置
  type-aliases-package: com.sjc.domain
  #mapper.xml位置
  mapper-locations: classpath:mapper/*Mapper.xml

# 配置eurekaServer
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:8761/eureka

```

4. 完成之后重启项目
5. 客户端代码会自动把服务注册到EurekaServer中
6. 在Eureka监控页面可以看到服务注册成功信息

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PROVIDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.103:provider-service:9091

### 4.4.3 服务消费者-注册服务中心

步骤：

1. 在服务消费者consumer-service工程中添加springcloud依赖管理和Eureka客户端依赖坐标
2. 在启动类上开启Eureka客户端发现功能 @EnableDiscoveryClient
3. 修改配置文件：spring.application.name指定应用名称，作为服务ID使用，并配置eureka注册中心地址

过程：

1. 在服务消费者consumer-service工程中添加springcloud依赖管理和Eureka客户端依赖坐标

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

<!--SpringCloud所有依赖管理的坐标-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR2</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2. 在启动类上开启Eureka客户端发现功能 @EnableDiscoveryClient

```

@SpringBootApplication
@EnableDiscoveryClient // 开启服务发现
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class,args);
    }
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}

```

3. 修改配置文件：spring.application.name指定应用名称，作为服务ID使用，并配置eureka注册中心地址

```

# 配置应用基本信息
server:
  port: 8080
spring:
  application:
    name: consumer-service
# 配置eureka server
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:8761/eureka

```

4. 启动服务，在服务中心查看是否注册成功

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.103:consumer-service:8080
PROVIDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.103:provider-service:9091

#### 4.4.4 Eureka元数据

Eureka的元数据有两种：标准元数据和自定义元数据。



- 标准元数据：主机名、IP地址、端口号、状态页和健康检查等信息，这些信息都会被发布在服务注册表中，用于服务之间的调用。
- 自定义元数据：可以使用eureka.instance.metadata-map配置，符合KEY/VALUE的存储格式。这些元数据可以在远程客户端中访问。

在程序中可以使用DiscoveryClient 获取指定微服务的所有元数据信息 。

## 标准元数据

```
import org.springframework.cloud.client.discovery.DiscoveryClient;

@Autowired
private DiscoveryClient discoveryClient;

/**
 * 获取元数据信息
 */
@GetMapping("/getMetaData")
public List<ServiceInstance> getMetaData(@PathVariable Integer id){
    List<ServiceInstance> instances =
    discoveryClient.getInstances("provider-service");
    return instances;
}
```

```
GET http://192.168.1.103:8080/consumer/getMetaData Send

Pretty Raw Preview Visualize JSON

1 [
2   {
3     "host": "192.168.1.103",
4     "port": 9091,
5     "uri": "http://192.168.1.103:9091",
6     "instanceId": "192.168.1.103:provider-service:9091",
7     "instanceInfo": {
8       "instanceId": "192.168.1.103:provider-service:9091",
9       "app": "PROVIDER-SERVICE",
10      "appGroupName": null,
11      "ipAddr": "192.168.1.103",
12      "sid": "na",
13      "homePageUrl": "http://192.168.1.103:9091/",
14      "statusPageUrl": "http://192.168.1.103:9091/actuator/info",
15      "healthCheckUrl": "http://192.168.1.103:9091/actuator/health",
16      "secureHealthCheckUrl": null,
17      "vipAddress": "provider-service",
18      "secureVipAddress": "provider-service",
19      "countryId": 1,
20      "dataCenterInfo": {
21        "@class": "com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo",
22        "name": "MyOwn"
23      },
24      "hostName": "192.168.1.103",
25      "status": "UP",
26      "overriddenStatus": "UNKNOWN",
27      "leaseInfo": {
28        "renewalIntervalInSecs": 30,
29        "durationInSecs": 90,
30        "registrationTimestamp": 1597306340286,
31        "lastRenewalTimestamp": 1597308080577,
32        "evictionTimestamp": 0,
33        "serviceUpTimestamp": 1597306339775
34      },
35      "isCoordinatingDiscoveryServer": false,
36      "metadata": {
37        "management.port": "9091"
38      },
39      "lastUpdatedTimestamp": 1597306340286,
40      "lastDirtyTimestamp": 1597306339725,
41      "actionType": "ADDED",
42      "asgName": null
43    },
44    "secure": false,
45    "serviceId": "PROVIDER-SERVICE",
46    "metadata": {
47      "management.port": "9091"
48    },
49    "scheme": null
50  }
51 ]
```

自定义元数据(了解)

```
# 配置eurekaserver
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:8761/eureka
  instance:
    metadata-map:
      my-metadata: zhangsan #自定义元数据
```

```
31      "lastRenewalTimestamp": 1597308243699,
32      "evictionTimestamp": 0,
33      "serviceUpTimestamp": 1597308243184
34    },
35    "isCoordinatingDiscoveryServer": false,
36    "metadata": {
37      "my-metadata": "zhangsan",
38      "management.port": "9091"
39    },
40    "lastUpdatedTimestamp": 1597308243699,
41    "lastDirtyTimestamp": 1597308243144,
42    "actionType": "ADDED",
43    "asgName": null
44  },
45  "secure": false,
46  "serviceId": "PROVIDER-SERVICE",
47  "metadata": {
48    "my-metadata": "zhangsan",
49    "management.port": "9091"
50  },
51  "scheme": null
52 }
53 }
```

标准元数据中包含了微服务注册到Eureka的基本信息，我们可以从uri中获取到该服务的ip地址和端口号。那么接下来我们去优化服务的调用方式。

#### 4.4.5 消费者通过Eureka访问提供者

使用 `DiscoveryClient` 获取服务提供者的host和port


```

@RestController
@RequestMapping("/consumer")
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private DiscoveryClient discoveryClient;

    @GetMapping("/queryById/{id}")
    public UserVO queryById(@PathVariable Integer id){
        //1、获取Eureka中注册的provider-service实例列表
        List<ServiceInstance> serviceInstanceList =
discoveryClient.getInstances("provider-service");
        //2、获取实例
        ServiceInstance serviceInstance =
serviceInstanceList.get(0);
        //3、根据实例的信息拼接的请求地址
        String url = serviceInstance.getUri()+ "/user/" + id;
        System.out.println(url);
        UserVO user = restTemplate.getForObject(url, UserVO.class);
        return user;
    }
}

```

打印日志查看url地址



```

Run: EurekaServerApplication x ProviderServiceApplication x ConsumerServiceApplication x
2020-08-13 16:52:33.098 INFO 19135 --- [nio-8080-exec-2] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
2020-08-13 16:52:33.099 INFO 19135 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-08-13 16:52:33.104 INFO 19135 --- [nio-8080-exec-2] o.s.web.servlet.DispatcherServlet : Completed initialization in 5
http://192.168.1.103:9091/user/1
http://192.168.1.103:9091/user/2
http://192.168.1.103:9091/user/2
http://192.168.1.103:9091/user/2

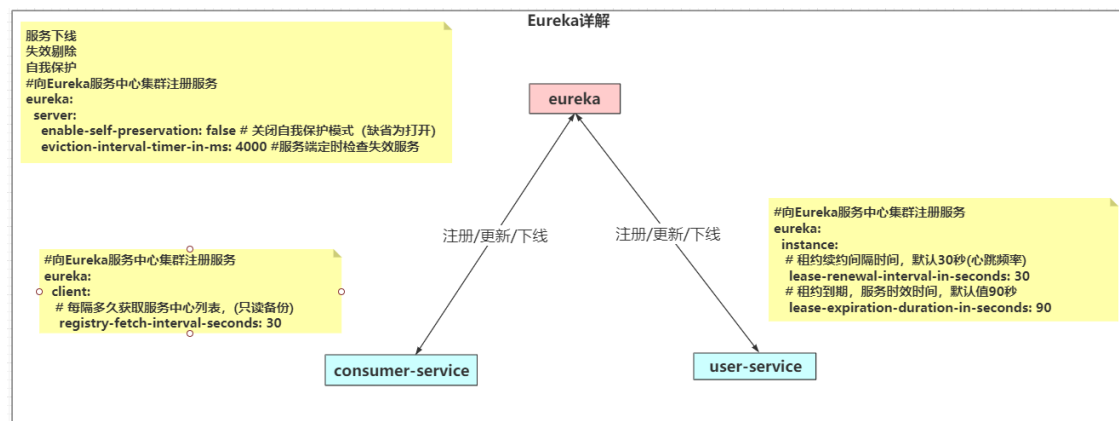
```

这里服务的host地址有什么问题！？

```
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:8761/eureka
  instance:
    prefer-ip-address: true # 默认注册时使用的主机名，想用ip进行注册添加
    instance-id: ${spring.cloud.client.ip-address}:${server.port} #
    管控制台展示服务ip+port
```

## 4.5 Eureka详解

### 4.5.1 基础架构



#### Eureka架构中的三个核心角色

- 服务注册中心：Eureka服务端应用，提供服务注册发现功能，eureka-server
- 服务提供者：提供服务的应用
  - 要求统一对外提供Rest风格服务即可
  - 本例子：provider-service
- 服务消费者：从注册中心获取服务列表，知道去哪调用服务方，consumer-service

### 4.5.2 Eureka客户端

服务提供者要向EurekaServer注册服务，并完成服务续约等工作。

服务注册过程：

1. 当我们导入了eureka-client依赖坐标，配置Eureka服务注册中心地址；
2. 服务在启动时，会检测是否有@DiscoveryClient注解和配置信息；
3. 如果有，则会向注册中心发起注册请求，携带服务元数据信息；
4. Eureka注册中心会把服务的信息保存在Map中。

### 服务续约过程：

服务每隔30秒会向注册中心续约(心跳)一次，如果没有续约，租约在90秒后到期，然后服务会被失效。**每隔30秒的续约操作我们称之为：心跳检测。**

```
#向Eureka服务中心集群注册服务
eureka:
  instance:
    lease-renewal-interval-in-seconds: 30    # 租约续约间隔时间，默认30秒
    lease-expiration-duration-in-seconds: 90 # 租约到期，服务时效时间，默认值90秒
```

- 两个参数可以修改服务续约行为
  - lease-renewal-interval-in-seconds:90，租约到期时效时间，默认90秒
  - lease-expiration-duration-in-seconds:30，租约续约间隔时间，默认30秒
- 服务超过90秒没有发生心跳，EurekaServer会将服务从列表移除

### 获取服务列表：

每隔30秒服务会从注册中心中拉取一份服务列表；

这个时间可以通过配置修改；

```
#向Eureka服务中心集群注册服务
eureka:
  client:
    registry-fetch-interval-seconds: 30 # 每隔多久获取服务中心列表，(只读备份)
```

- 服务消费者启动时，会检测是否获取服务注册信息配置；
- 如果是，则会从 EurekaServer服务列表获取只读备份，缓存到本地；
- 每隔30秒，会重新获取并更新数据；
- 每隔30秒的时间可以通过配置 `registry-fetch-interval-seconds` 修改；

### 4.5.3 Eureka服务端

服务下线：

- 当服务正常关闭操作时，会发送服务下线的REST请求给EurekaServer。
- 服务中心接受到请求后，将该服务置为下线状态；

失效剔除：

Eureka Server会定时（间隔值是`eureka.server.eviction-interval-timer-in-ms`，默认值为60）进行检查，如果发现实例在一定时间（此值由客户端设置的`eureka.instance.lease-expiration-duration-in-seconds`定义，默认值为90s）内没有收到心跳，则会注销此实例。

自我保护：

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

服务中心页面会显示如下提示信息

The screenshot shows the Spring Eureka web interface at localhost:10086. The 'System Status' section contains two tables. The left table shows 'Environment: test' and 'Data center: default'. The right table shows 'Current time: 2019-05-17T09:18:37 +0800', 'Uptime: 00:06', 'Lease expiration enabled: false', 'Renews threshold: 1', and 'Renews (last min): 0'. A red box highlights an emergency message in Chinese: '译文：紧急情况！Eureka可能错误地声称实例已经启动，而事实并非如此。续约低于阈值，因此实例不会为了安全而过期。' Below this is the English translation: 'EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.' The 'DS Replicas' section shows '127.0.0.1'. The 'Instances currently registered with Eureka' table shows one instance: 'EUREKA-SERVER' with 'n/a (1)' AMIs, '(1)' Availability Zones, and 'UP (1) - localhost:eureka-server:10086' status.

Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (1)	(1)	UP (1) - localhost:eureka-server:10086

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

- 自我保护模式下，不会剔除任何服务实例；
- 自我保护模式保证了大多数服务依然可用；
- 通过 `enable-self-preservation` 配置可用关停自我保护，默认值是打开；

```
#向Eureka服务中心集群注册服务
eureka:
  server:
    enable-self-preservation: false # 关闭自我保护模式（缺省为打开）
```

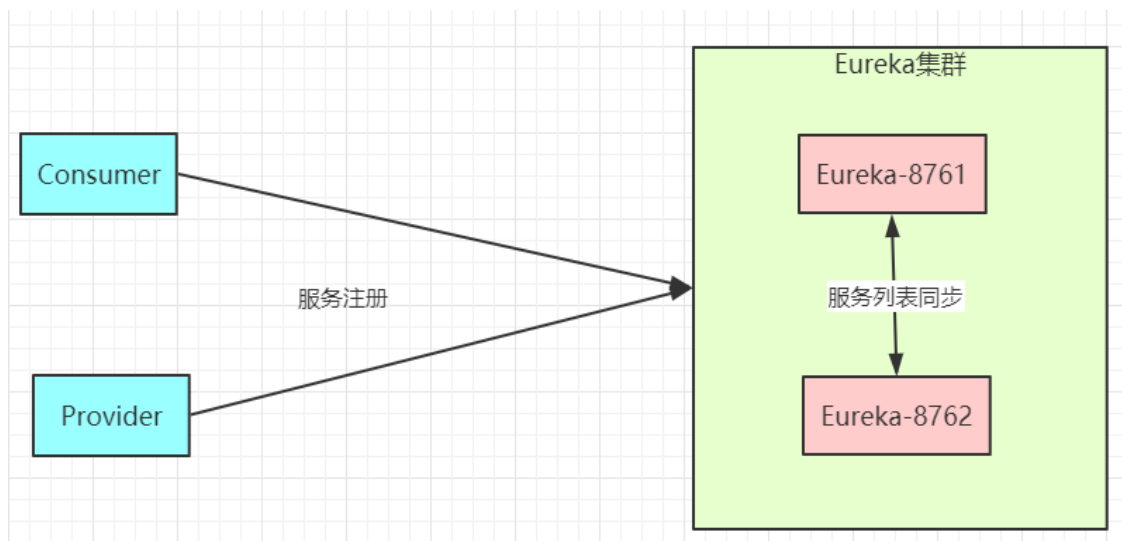
## 4.6 Eureka高可用集群

在上面我们实现了单节点的Eureka Server的服务注册与服务发现功能。Eureka Client会定时连接Eureka Server，获取注册表中的信息并缓存到本地。微服务在消费远程API时总是使用本地缓存中的数据。



因此一般来说，即使Eureka Server发生宕机，也不会影响到服务之间的调用。但如果Eureka Server宕机时，某些微服务也出现了不可用的情况，Eureka Server中的缓存若不被刷新，就可能会影响到微服务的调用，甚至影响到整个应用系统的高可用。因此，在生成环境中，通常会部署一个高可用的Eureka Server集群。

Eureka Server可以通过运行多个实例并相互注册的方式实现高可用部署，Eureka Server实例会彼此增量地同步信息，从而确保所有节点数据一致。事实上，节点之间相互注册是Eureka Server的默认行为。



目标： 搭建Eureka高可用集群，并使生产者服务和消费者服务注册到Eureka集群

步骤：

1. 修改本机host,位置： C:\Windows\System32\Drivers\etc
2. 将Eureka配置改为多文件配置，相互注册并关闭是否注册和是否拉取注册列表
3. 复制Eureka启动类，指定启动时加载的配置文件
4. 改造生产者和消费者的配置文件
5. 启动测试

过程：

- 修改本机host,位置： C:\Windows\System32\Drivers\etc

```
127.0.0.1    eureka1 eureka2
```

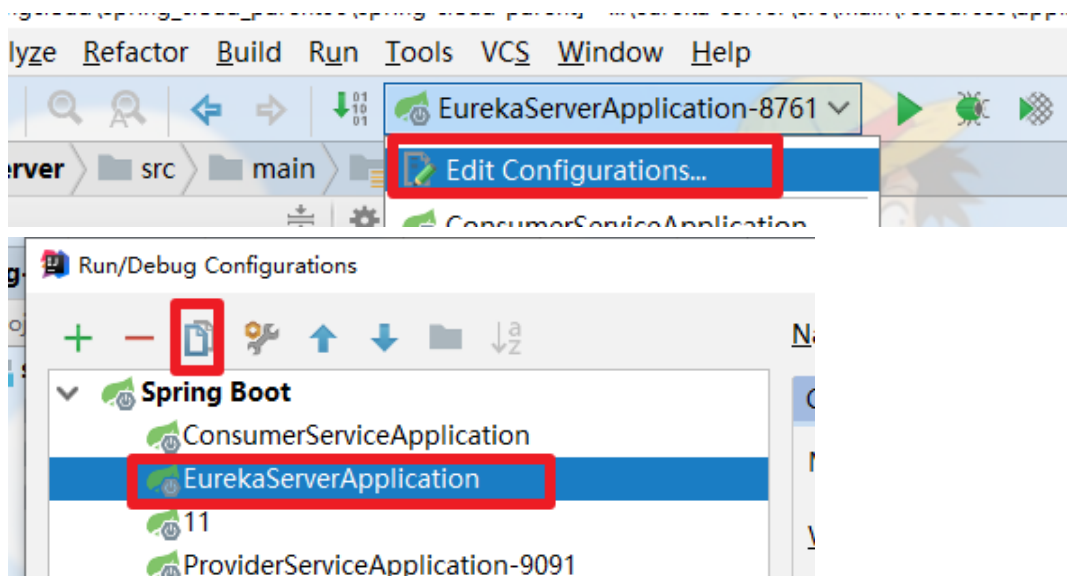
- 将Eureka配置改为多文件配置，相互注册并关闭是否注册和是否拉取注册列表

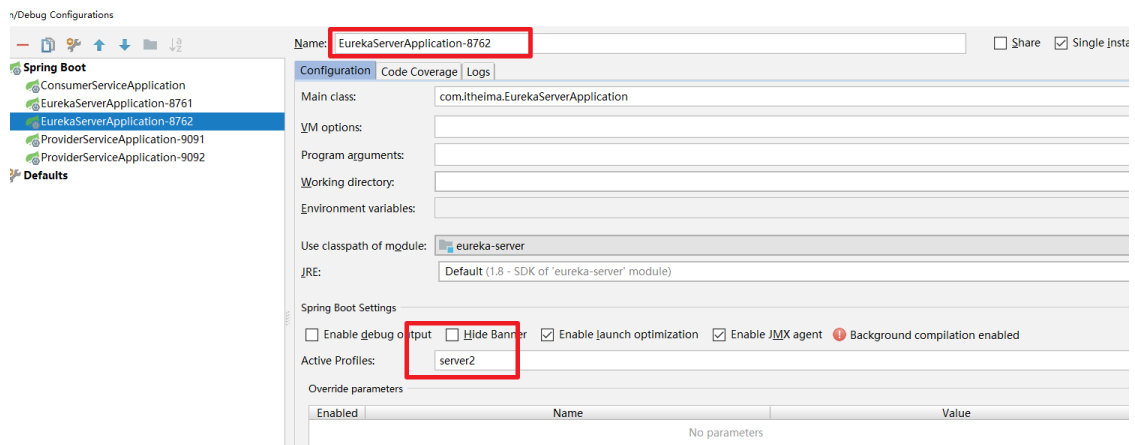
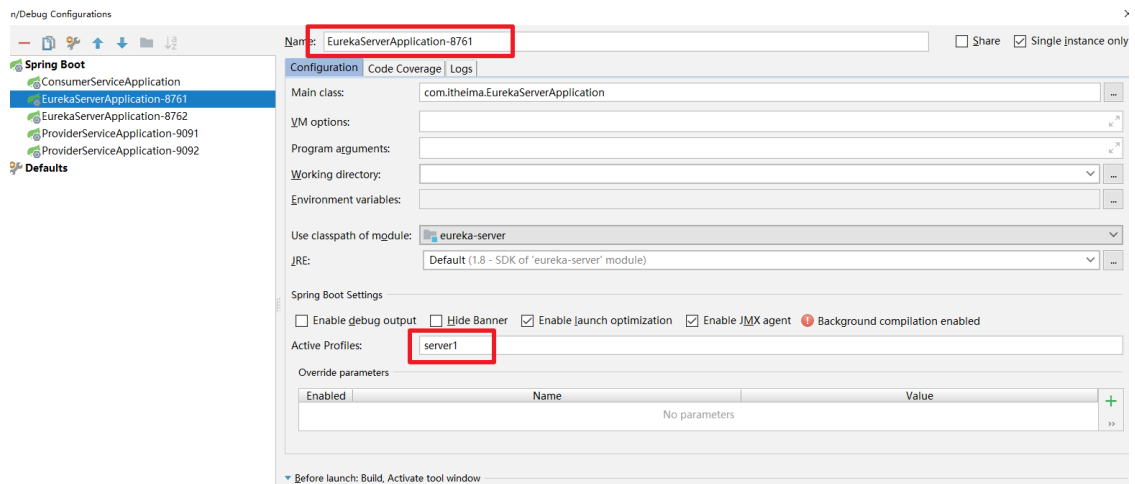
```

spring:
  application:
    name: eureka-server
---
spring:
  profiles: server1
server:
  port: 8761
eureka:
  client:
    service-url:
      defaultZone: http://eureka2:8762/eureka  #只填写其它eureka地址
      #即可, 多个用逗号分隔
    instance:
      hostname: eureka1
---
spring:
  profiles: server2
server:
  port: 8762
eureka:
  client:
    service-url:
      defaultZone: http://eureka1:8761/eureka  #只填写其它eureka地址
      #即可, 多个用逗号分隔
    instance:
      hostname: eureka2

```

- 复制Eureka启动类，指定启动时加载的配置文件





- 改造生产者和消费者的配置文件

```
eureka:
  client:
    service-url:
      defaultZone:
http://eureka1:8761/eureka,http://eureka2:8762/eureka
    instance:
      prefer-ip-address: true #使用ip注册
      instance-id: ${spring.cloud.client.ip-address}:${server.port} #
管控制台展示服务ip+port
```

- 启动测试

localhost:8761

DS Replicas

eureka2

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.129.18081
EUREKA-SERVER	n/a (2)	(2)	UP (2) - localhost:eureka-server:8761, localhost:eureka-server:8762
PROVIDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.129.19091

General Info

Name	Value
total-avail-memory	395mb
environment	test
num-of-cpus	8
current-memory-usage	122mb (30%)
server-up-time	00:09
registered-replicas	http://eureka2:8762/eureka/
unavailable-replicas	
available-replicas	http://eureka2:8762/eureka/

localhost:8762

DS Replicas

eureka1

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.129.18081
EUREKA-SERVER	n/a (2)	(2)	UP (2) - localhost:eureka-server:8761, localhost:eureka-server:8762
PROVIDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.129.19091

General Info

Name	Value
total-avail-memory	509mb
environment	test
num-of-cpus	8
current-memory-usage	414mb (81%)
server-up-time	00:09
registered-replicas	http://eureka1:8761/eureka/
unavailable-replicas	
available-replicas	http://eureka1:8761/eureka/

Instance Info

## 4.7 Eureka服务端源码

为什么我们在启动类上添加上@EnableEurekaServer注解就可以启动一个Eureka服务了？首先我们来回顾一下学习过的spring boot自动配置源码，spring boot自动配置借助于@Enable\*注解和@Condition\*的一系列注解，完成了代码的自动配置。Eureka又是springboot框架编写的，所以他们的实现原理是一致的。

```

@EnableEurekaServer
---@Import({EurekaServerMarkerConfiguration.class})
    ---创建了一个Marker Bean对象

spring-cloud-netflix-eureka-server.jar
---META-INF:spring.factories
    ---org.springframework.cloud.netflix.eureka.server.EurekaServerAutoConfiguration
    ---@ConditionalOnBean({Marker.class})
        可以看到EurekaServerAutoConfiguration创建的判断条件是是否有Marker.class, @EnableEurekaServer这个注解为我们创建了Marker.class

    ---@Import({EurekaServerInitializerConfiguration.class})
        引入启动配置类,实现了SmartLifecycle, 也就意味着Spring容器启动时会去执行start()方法。加载所有的EurekaServer的配置

    ---@EnableConfigurationProperties({EurekaDashboardProperties.class, InstanceRegistryProperties.class})
        EurekaDashboardProperties 配置Eureka的管控台 InstanceRegistryProperties 配置期望续约数量和默认的通信数量

    ---EurekaController 管控台的Controller类
    ---PeerAwareInstanceRegistry
    ---InstanceRegistry
    ---PeerAwareInstanceRegistryImpl
        register(InstanceInfo info, boolean isReplication) 服务注册
        renew(String appName, String id, boolean isReplication) 服务更新

```

### @EnableEurekaServer

```

---@Import({EurekaServerMarkerConfiguration.class})
    ---创建了一个Marker Bean对象

```

```

spring-cloud-netflix-eureka-server.jar
    ---META-INF:spring.factories
        ---

```

```

org.springframework.cloud.netflix.eureka.server.EurekaServerAutoConfiguration
    ---@ConditionalOnBean({Marker.class})
        可以看到EurekaServerAutoConfiguration创建的判断条件是是否有Marker.class, @EnableEurekaServer这个注解为我们创建了Marker.class

```

```

---
@Import({EurekaServerInitializerConfiguration.class})
    引入启动配置类,实现了SmartLifecycle, 也就意味着Spring容器启动时会去执行start()方法。加载所有的EurekaServer的配置

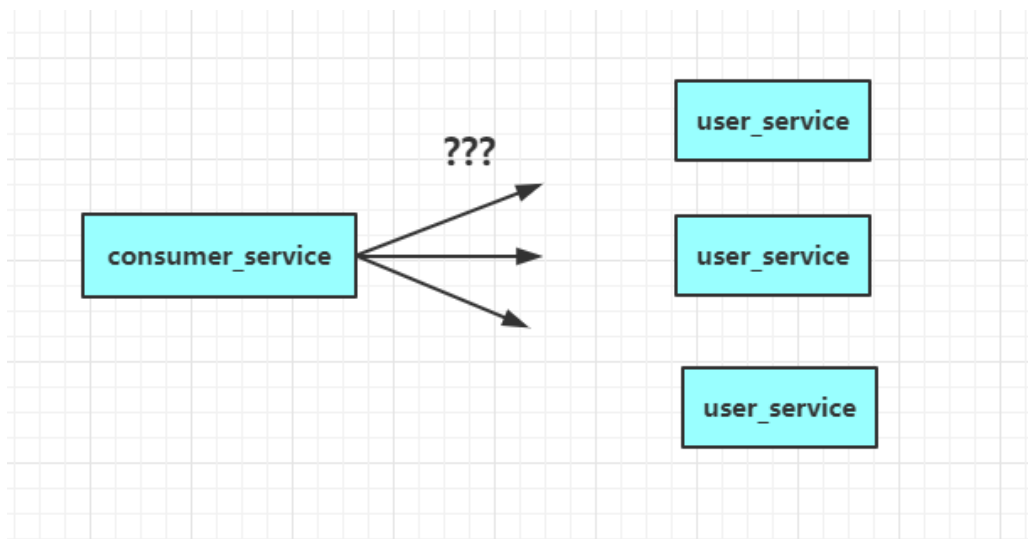
```

## 五、负载均衡 Spring Cloud Ribbon

### 5.1 Ribbon 简介

解决了集群服务中，多个服务高效率访问的问题。

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



## 5.2 入门案例

实现负载均衡访问用户服务。

如果想要做负载均衡，我们的服务至少2个以上，所以我们先来在idea中启动两个生产者。

实现步骤：

第一步：启动多个provider-service服务

1. 编辑应用启动配置
2. 选择启动类
3. 选择应用服务、填写服务名称、指定服务端口
4. 复制一个，将端口改为9092

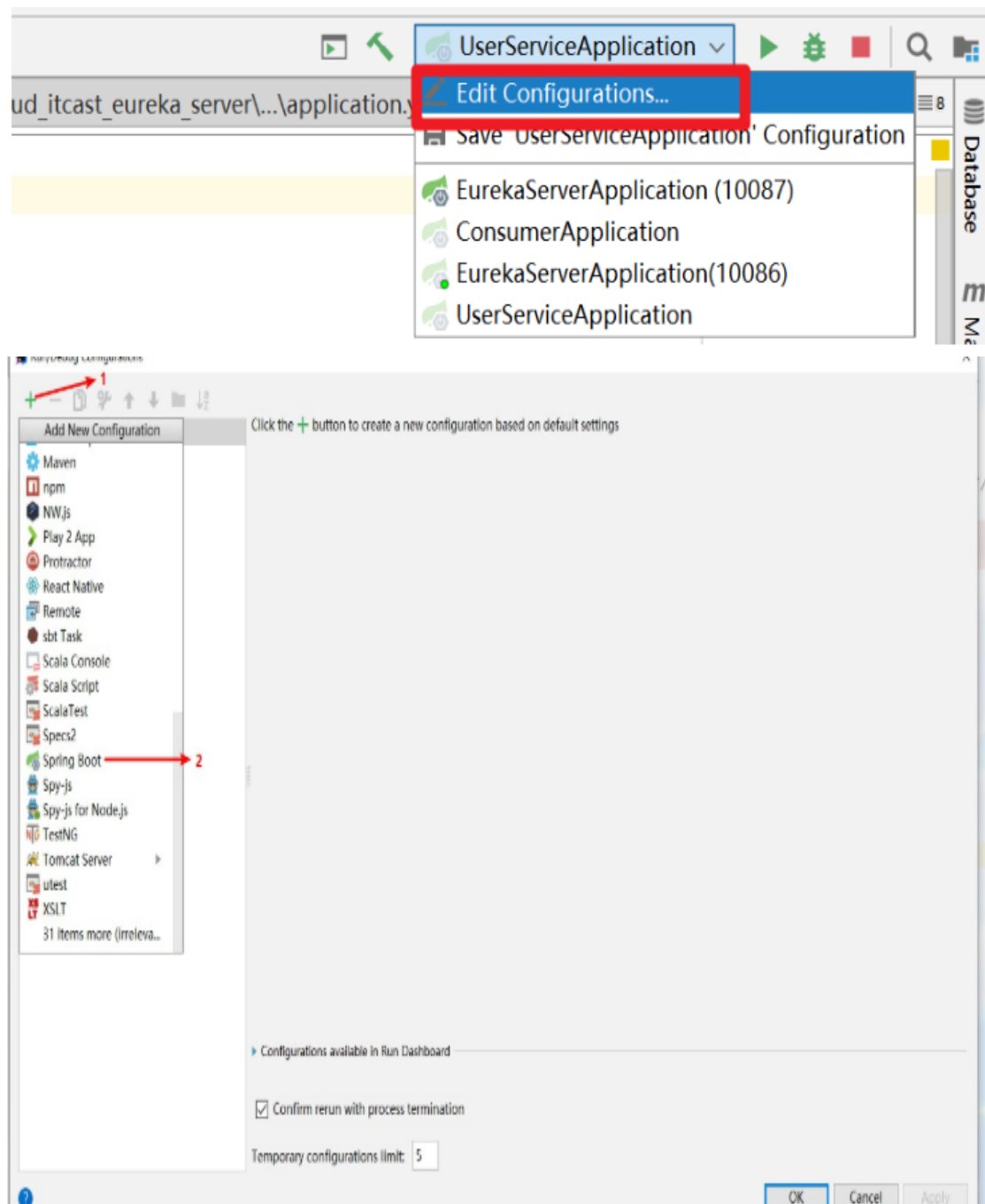
第二步：开启消费者负载均衡

1. 在RestTemplate的注入方法上加入@LoadBalanced注解
2. 修改调用请求的Url地址，改为服务名称调用
3. 访问页面查看效果

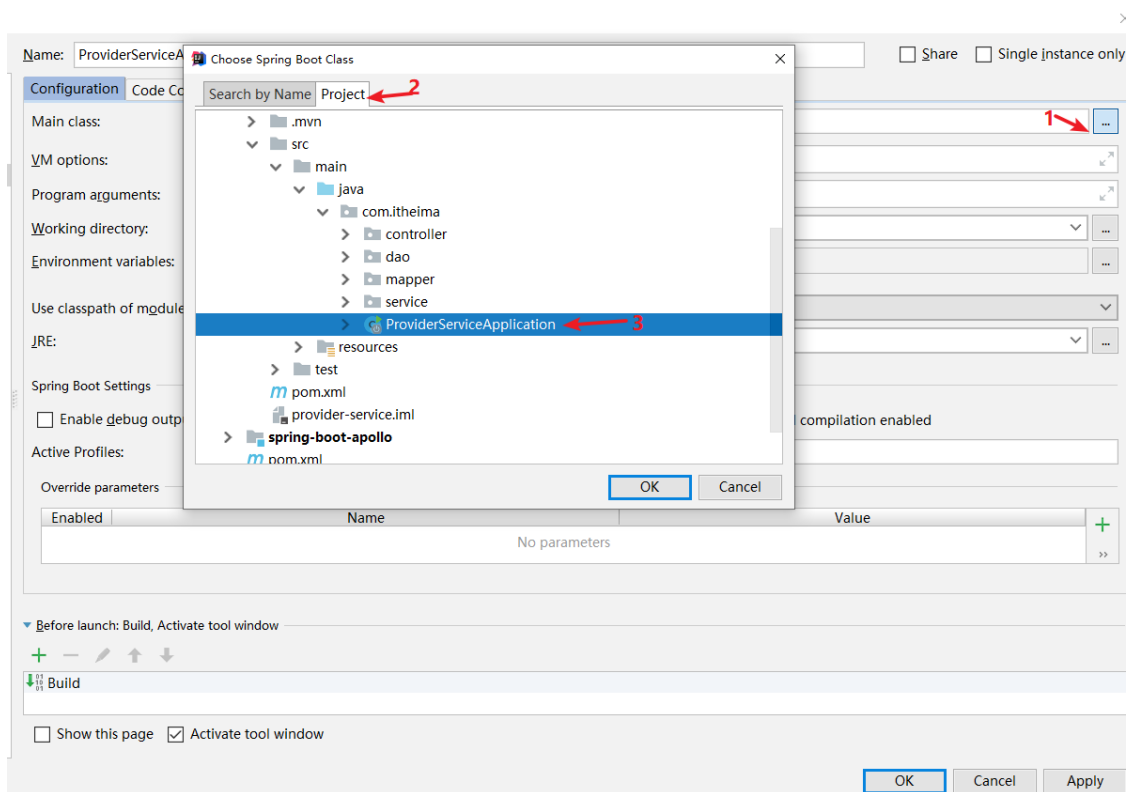
实现过程：

第一步：使用run dashboard启动两个provider-service应用

- 1.编辑应用启动配置

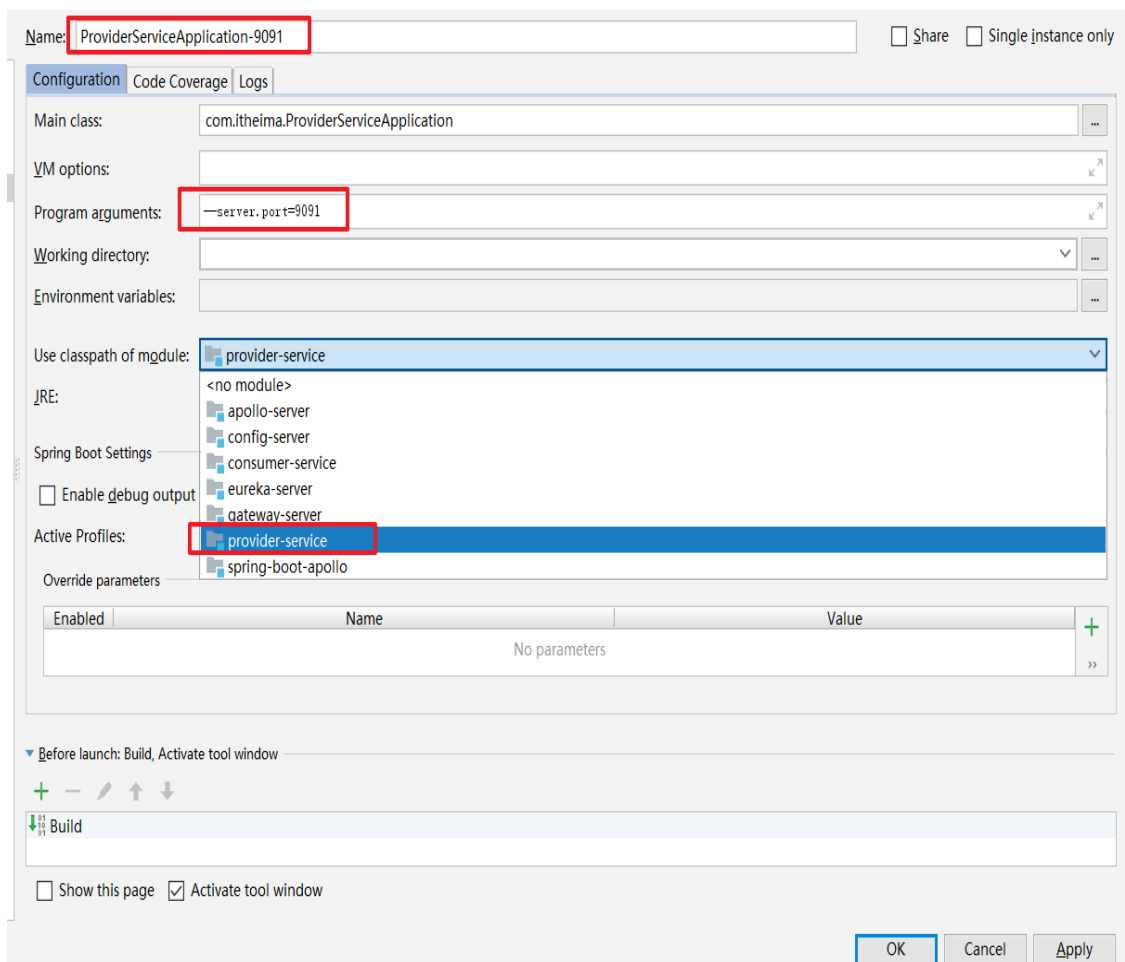


## 2. 选择启动类

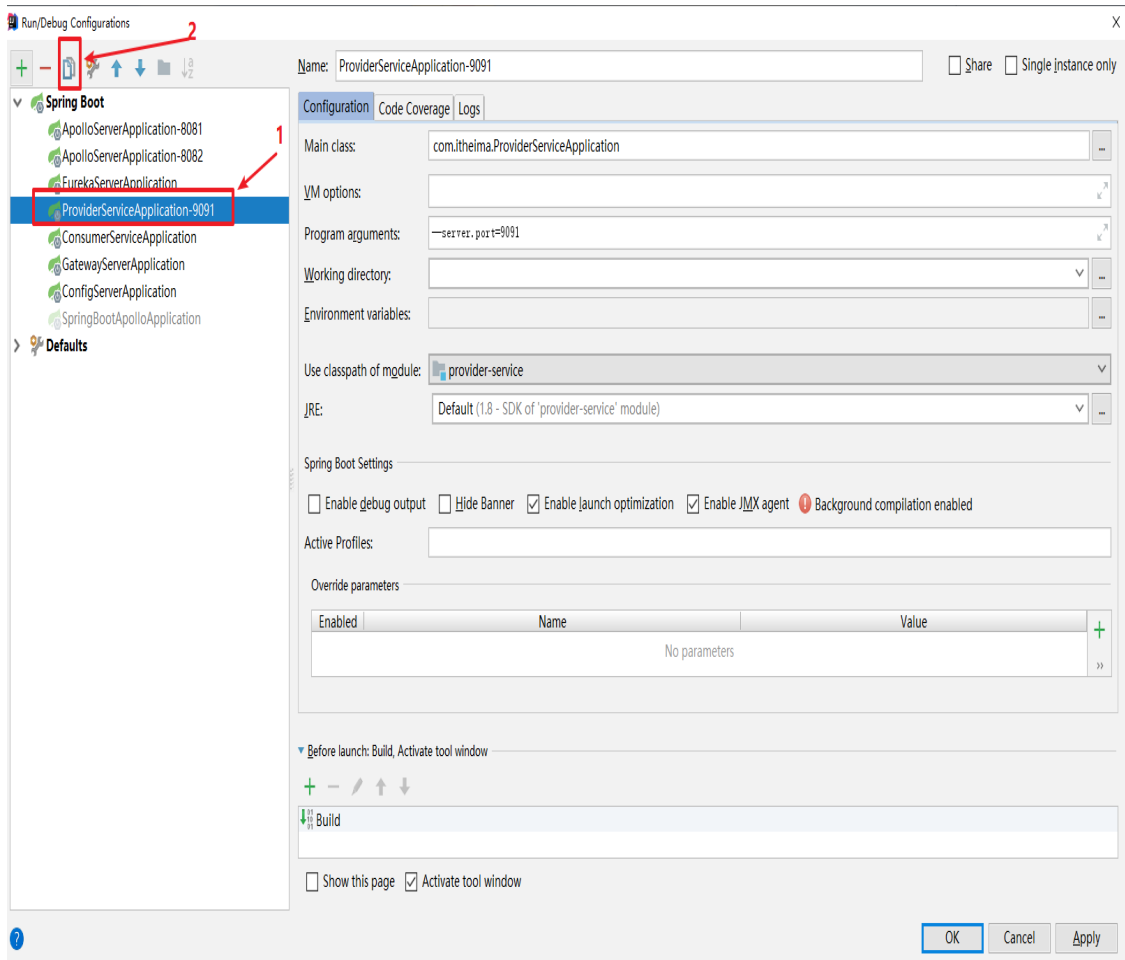


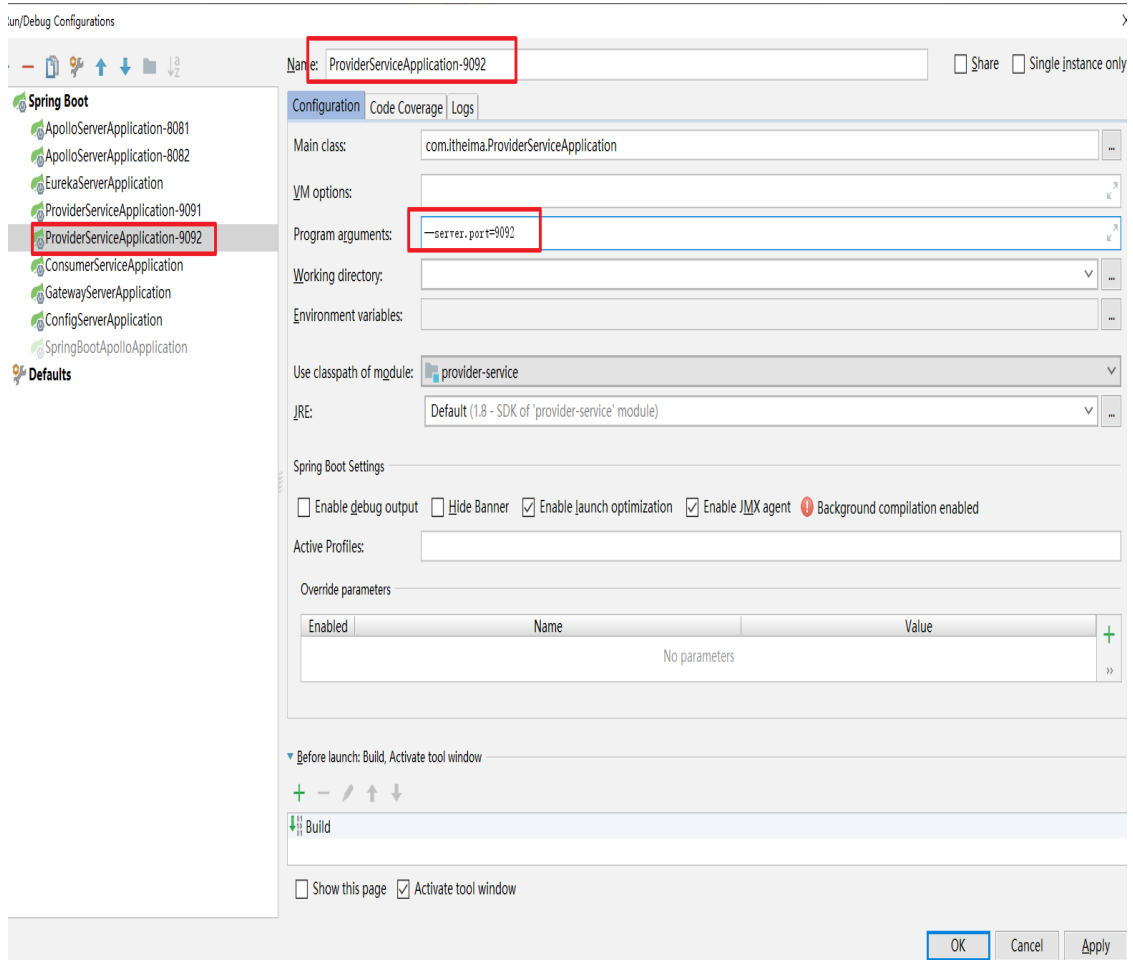
3.选择应用服务、填写服务名称、指定服务端口



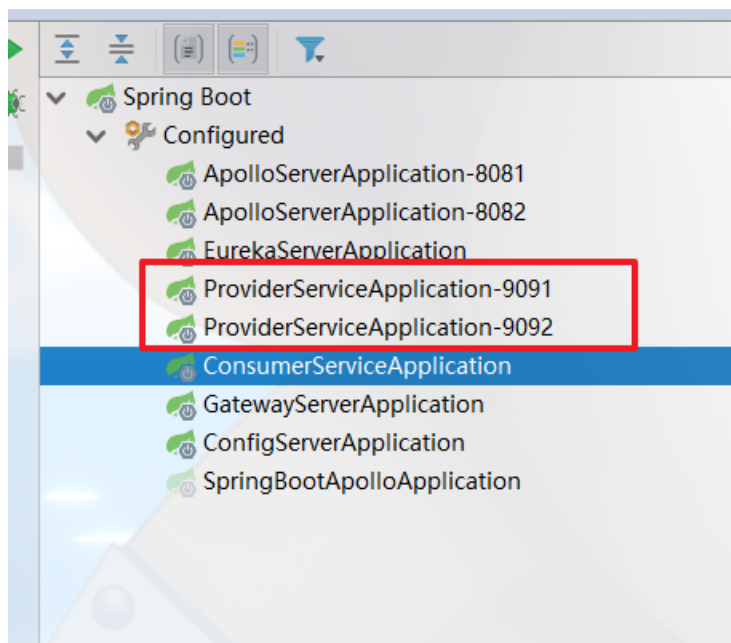


4.复制一个，将端口改为9092





## 5. 结果



第二步：开启消费者调用负载均衡

Eureka已经集成Ribbon，所以无需引入依赖。

1. 在RestTemplate的配置方法上添加@LoadBalanced注解即可

```
@Bean
@LoadBalanced// 开启负载均衡
public RestTemplate restTemplate(){
    return new RestTemplate();
}
```

2. 修改ConsumerController调用方式，不再手动获取ip和端口，而是直接通过服务名称调用

```
@GetMapping("/findUserById/{id}")
public User queryById(@PathVariable Long id){
    String url = "http://provider-service/user/findUserById/" + id;
    return restTemplate.getForObject(url, User.class);
}
```

3. 访问页面查看结果；并在9091和9092的控制台查看执行情况

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas			
eureka2			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.103:8080
EUREKA-SERVER	n/a (2)	(2)	UP (2) - 192.168.1.103:eureka-server:8761, 192.168.1.103:eureka-server:8762
PROVIDER-SERVICE	n/a (2)	(2)	UP (2) - 192.168.1.103:provider-service:9091, 192.168.1.103:provider-service:9092

## 5.3 负载策略

Ribbon内置了多种负载均衡策略，内部负责复杂均衡的顶级接口为：  
com.netflix.loadbalancer.IRule，实现方式如下：

```

c AbstractLoadBalancerRule (com.netflix.loadbalancer)
c AvailabilityFilteringRule (com.netflix.loadbalancer)
c BestAvailableRule (com.netflix.loadbalancer)
c ClientConfigEnabledRoundRobinRule (com.netflix.loadbalancer)
c PredicateBasedRule (com.netflix.loadbalancer)
c RandomRule (com.netflix.loadbalancer)
c ResponseTimeWeightedRule (com.netflix.loadbalancer)
c RetryRule (com.netflix.loadbalancer)
c RoundRobinRule (com.netflix.loadbalancer)
c WeightedResponseTimeRule (com.netflix.loadbalancer)
c ZoneAvoidanceRule (com.netflix.loadbalancer)

```

- com.netflix.loadbalancer.RandomRule：随机选择一个server
- com.netflix.loadbalancer.RetryRule：对选定的负载均衡策略机上重试机制。
- com.netflix.loadbalancer.RoundRobinRule：以轮询的方式进行负载均衡
- com.netflix.loadbalancer.WeightedResponseTimeRule：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。
- com.netflix.loadbalancer.AvailabilityFilteringRule：可用过滤策略，过滤掉故障和请求数超过阈值的服务实例，再从剩下的实例中轮询调用(默认)

在服务消费者的application.yml配置文件中修改负载均衡策略，格式：

{服务提供者名称}.ribbon.NFLoadBalancerRuleClassName

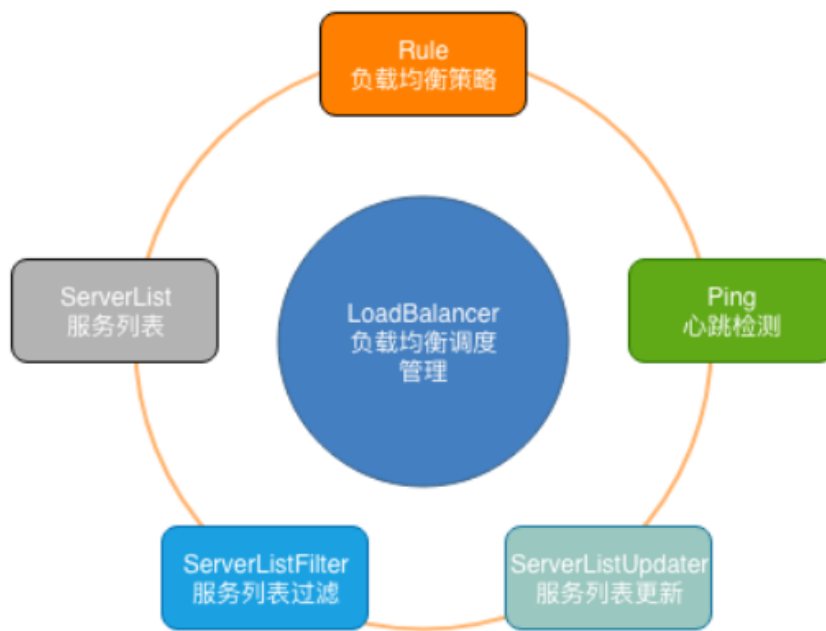
```

# 修改服务地址轮询策略，默认是轮询，配置之后变随机, RandomRule
provider-service:
  ribbon:
    NFLoadBalancerRuleClassName:
      com.netflix.loadbalancer.RandomRule

```

## 5.4 Ribbon源码解析

### 5.4.1 Ribbon关键组件



- ServerList：可以响应客户端的特定服务的服务器列表。
- ServerListFilter：可以动态获得的具有所需特征的候选服务器列表的过滤器。
- ServerListUpdater：用于执行动态服务器列表更新。
- Rule：负载均衡策略，用于确定从服务器列表返回哪个服务器。
- Ping：客户端用于快速检查服务器当时是否处于活动状态。
- LoadBalancer：负载均衡器，负责负载均衡调度的管理。

### 5.4.2 源码分析

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

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

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

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

源码跟踪步骤：

1. 打开LoadBalancerInterceptor类，断点打入intercept方法中

```
ConsumerServiceApplication.java x LoadBalancerInterceptor.java x LoadBalancerClient.java x RibbonLoadBalancerClient.java x ConsumerController.java x
33 */
34 public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {
35
36     private LoadBalancerClient loadBalancer; loadBalancer: RibbonLoadBalancerClient@8017
37
38     private LoadBalancerRequestFactory requestFactory; requestFactory: LoadBalancerRequestFactory@8018
39
40     public LoadBalancerInterceptor(LoadBalancerClient loadBalancer,
41     LoadBalancerRequestFactory requestFactory) {
42         this.loadBalancer = loadBalancer;
43         this.requestFactory = requestFactory;
44     }
45
46     public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {
47         // for backwards compatibility
48         this(loadBalancer, new LoadBalancerRequestFactory(loadBalancer));
49     }
50
51     @Override
52     public ClientHttpResponse intercept(final HttpRequest request, final byte[] body, request: InterceptingClientHttpRequest@8016 body: {})
53     final ClientHttpRequestExecution execution) throws IOException { execution: InterceptingClientHttpRequest$InterceptingRequestExecution@8020
54     final URI originalUri = request.getURI(); originalUri: "http://provider-service/user/1" request: InterceptingClientHttpRequest@8016
55     String serviceName = originalUri.getHost(); serviceName: "provider-service" originalUri: "http://provider-service/user/1"
56     Assert.state(serviceName != null, message: "Request URI does not contain a valid hostname: " + originalUri); 获取服务 name
57
58     return this.loadBalancer.execute(serviceName,
59         this.requestFactory.createRequest(request, body, execution));
60 }
61 }
```

## 2. 继续跟入execute方法：发现获取了9092发端口的服务

```
ConsumerServiceApplication.java x LoadBalancerInterceptor.java x LoadBalancerClient.java x RibbonLoadBalancerClient.java x ConsumerController.java x
187
188 * @param serviceId id of the service to execute the request to
189 * @param request to be executed request: LoadBalancerRequestFactory$Lambda@8452
190 * @param hint used to choose appropriate {@link Server} instance
191 * @return request execution result
192 * @throws IOException executing the request may result in an {@link IOException}
193 */
194 public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) serviceId: "provider-service" request: LoadBalancerRequestFactory$Lambda@8452 hint: null
195     throws IOException { 获取负载均衡器
196     LoadBalancer loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer: {NLoadBalancer: name=provider-service, current list of Servers=[192.168.1.183:9092, 192.168.1.183:9091]
197     Server server = getServer(loadBalancer, hint); server: "192.168.1.183:9092" loadBalancer: "DynamicServerListLoadBalancer: {NLoadBalancer: name=provider-service, current list of Servers=[192.168.1.183:9092,
198     if (server == null) { 通过负载均衡器获取服务调用地址 当前服务调用端口
199         throw new IllegalStateException("No instances available for " + serviceId);
200     }
201
202     RibbonServer ribbonServer = new RibbonServer(serviceId, server, serviceId: "provider-service" server: "192.168.1.183:9092"
203     }
```

- 获取负载均衡器
- 使用负载均衡器从服务列表获取服务

## 3. 再跟下一次，发现获取的是9091和9092之间切换

```
ConsumerServiceApplication.java x LoadBalancerInterceptor.java x LoadBalancerClient.java x RibbonLoadBalancerClient.java x ConsumerController.java x
107 * @param <T> returned request execution result type
108 * @param serviceId id of the service to execute the request to
109 * @param request to be executed request: LoadBalancerRequestFactory$Lambda@8577
110 * @param hint used to choose appropriate {@Link Server} instance
111 * @return request execution result
112 * @throws IOException executing the request may result in an {@Link IOException}
113 */
114 public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) serviceId: "provider-
115 throws IOException {
116     ILoadBalancer loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer.
117     Server server = getServer(loadBalancer, hint); server: "192.168.1.103:9092" loadBalancer: "DynamicSei
118     if (server == null) {
119         throw new IllegalStateException("No instances available for " + serviceId);
120     }
121     RibbonServer ribbonServer = new RibbonServer(serviceId, server, serviceId: "provider-service" server
122     isSecure(server, serviceId),
```

4. 通过代码断点内容判断，果然是实现了负载均衡

## 六、熔断器 Spring Cloud Hystrix

### 6.1 Hystrix 简介



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

Hystrix的作用是什么？

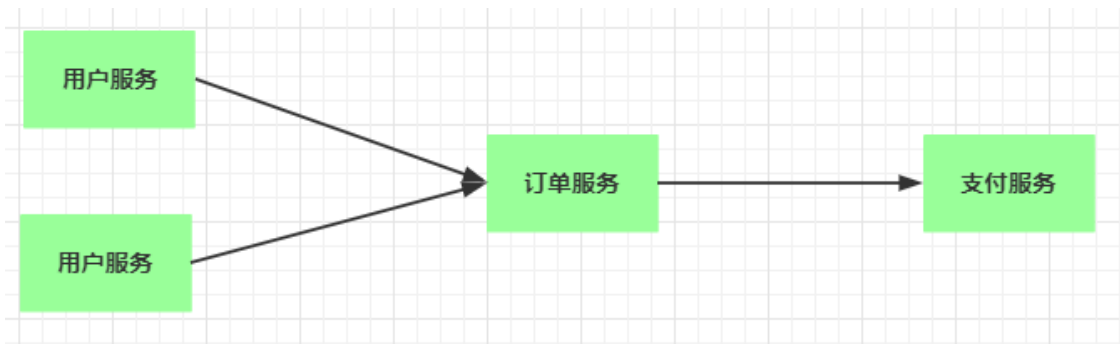
Hystrix是Netflix开源的一个延迟和容错库，用于隔离访问远程服务、第三方库、防止出现级联失败也就是雪崩效应。

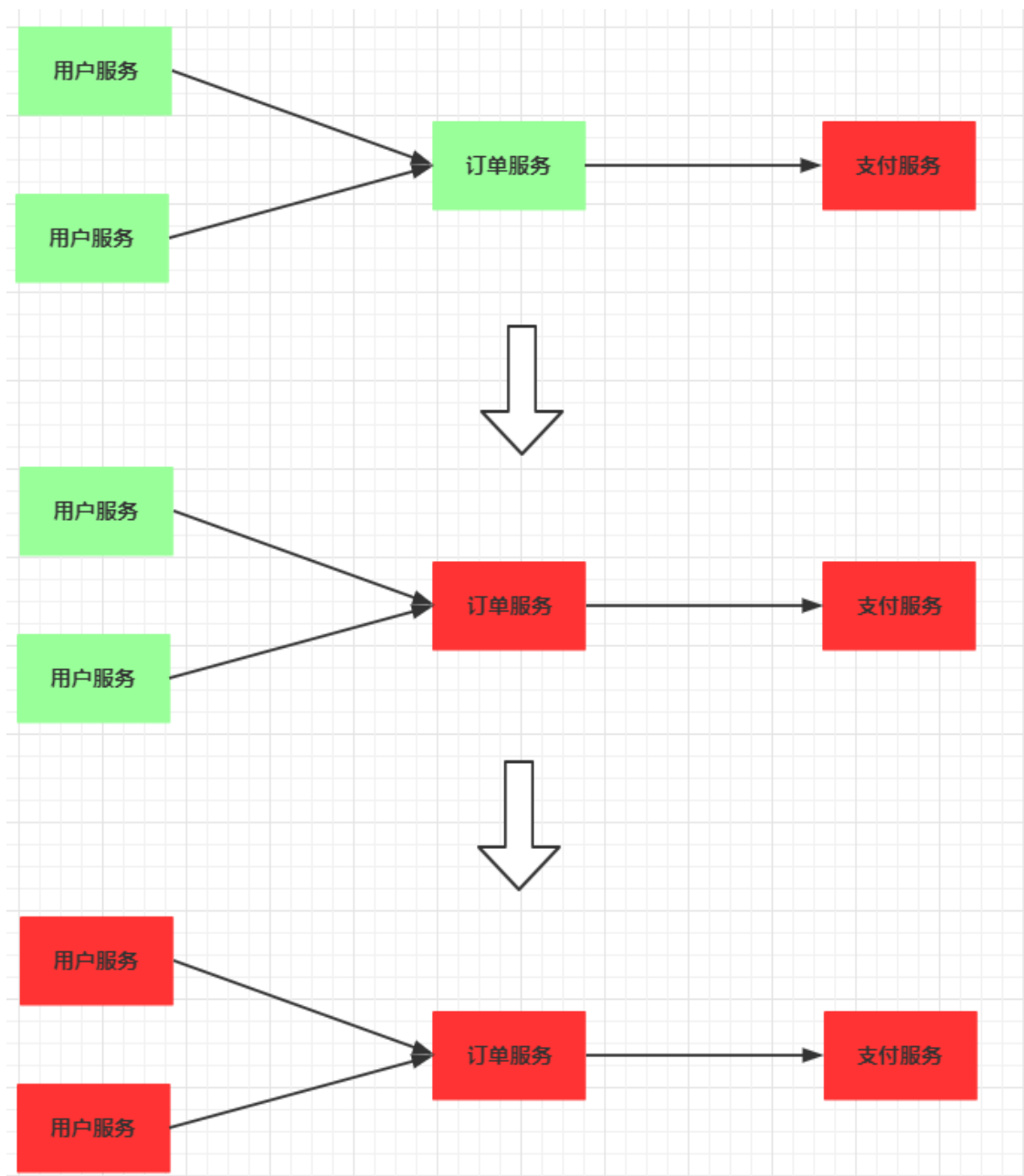




## 6.2 雪崩效应

- 微服务中，一个请求可能需要多个微服务接口才能实现，会形成复杂的调用链路。
- 如果某服务出现异常，请求阻塞，用户得不到响应，容器中线程不会释放，于是越来越多用户请求堆积，越来越多线程阻塞。
- 单服务器支持线程和并发数有限，请求如果一直阻塞，会导致服务器资源耗尽，从而导致所有其他服务都不可用，从而形成雪崩效应；





Hystrix解决雪崩问题的手段，主要是服务降级 (兜底)，线程隔离；

## 6.3 熔断案例

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

实现步骤：

1. 引入熔断的starter依赖坐标
2. 消费者服务开启熔断的注解
3. 编写服务降级处理的方法
4. 配置熔断的策略
5. 模拟异常代码
6. 测试熔断服务效果

#### 实现过程：

1. 引入熔断的依赖坐标：

- consumer-service中加入依赖

```
<!--熔断Hystrix starter-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
<!--Springcloud 新版本中不包含@hystrixcommand注解,需要引入此依赖-->
<dependency>
    <groupId>com.netflix.hystrix</groupId>
    <artifactId>hystrix-javanica</artifactId>
</dependency>
```

2. 开启熔断的注解

```

// 注解简化写法：微服务中，注解往往引入多个，简化注解可以使用组合注解。@SpringCloudApplication = 等同于
@SpringBootApplication+@EnableDiscoveryClient+@EnableCircuitBreaker
/*
@SpringBootApplication
@EnableDiscoveryClient// 开启服务发现
@EnableCircuitBreaker// 开启熔断
*/
@SpringCloudApplication
public class ConsumerApplication {
    @Bean
    @LoadBalanced// 开启负载均衡
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class,args);
    }
}

```

3. 编写服务降级处理方法：使用@HystrixCommand定义fallback方法。

注意：熔断服务降级方法必须保证与被降级方法相同的参数列表和返回值。

```

@GetMapping("/findUserById/{id}")
@HystrixCommand(fallbackMethod ="fallBackMethod")
public User findUserById(@PathVariable Integer id){
    String url = "http://provider-service/user/findUserById/" + id;
    return restTemplate.getForObject(url,User.class);
}

// 熔断方法
public User fallBackMethod(Integer id){
    User user = new User();
    user.setId(id);
    user.setName("熔断方法");
    return user;
}

```

#### 4. 配置熔断策略

- i. 常见熔断策略配置
- ii. 熔断后休眠时间: `sleepWindowInMilliseconds`
- iii. 熔断触发最小请求次数: `requestVolumeThreshold`
- iv. 熔断触发错误比例阈值: `errorThresholdPercentage`
- v. 熔断超时时间: `timeoutInMilliseconds`

```
# 配置熔断策略:
hystrix:
  command:
    default:
      circuitBreaker:
        forceOpen: false # 强制打开熔断器 默认false关闭的
        errorThresholdPercentage: 50 # 触发熔断错误比例阈值, 默认值50%
        sleepWindowInMilliseconds: 5000 # 熔断后休眠时长, 默认值5秒
        requestVolumeThreshold: 20 # 熔断触发最小请求次数, 默认值是20
      execution:
        isolation:
          thread:
            timeoutInMilliseconds: 1000 # 熔断超时设置, 默认为1秒
```

#### 5. 停止生产者服务, 模拟服务异常, 然后访问消费者会发现走了熔断方法。

- 停止1个生产者, 在短时间内, 会访问已经停止的生产者服务,

GET http://localhost:8080/consumer/findUserById/1 Send

Body Cookies Headers (5) Test Results Status: 200 OK Time: 19 ms Size: 306 B Save

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "username": null,
4   "password": null,
5   "name": "熔断方法",
6   "age": null,
7   "sex": null,
8   "birthday": null,
9   "created": null,
10  "updated": null,
11  "note": null
12 }
```

#### 6. 测试访问超时: 服务提供者线程休眠超过5秒, 访问消费者触发fallback方法。

```

@Service
public class UserServiceImpl implements UserService {
    private final BeanCopier beanCopier =
        BeanCopier.create(UserD0.class, UserDT0.class, false);

    /**
     * 用户管理模块的dao组件
     */
    @Autowired
    private UserDAO userDAO;

    /**
     * 根据用户ID查找用户信息
     * @param id 用户ID
     * @return 用户信息
     */
    @Override
    public UserDT0 findById(Integer id) {
        try {
            // 测试服务超时, 熔断
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            System.err.println(e.getMessage());
        }
        UserDT0 userDT0 = new UserDT0();
        beanCopier.copy(userDAO.findById(id), userDT0, null);
        return userDT0;
    }
}

```

GET http://localhost:8080/consumer/findUserById/1 Send

Body Cookies Headers (5) Test Results Status: 200 OK Time: 19 ms Size: 306 B Save

Pretty Raw Preview Visualize JSON

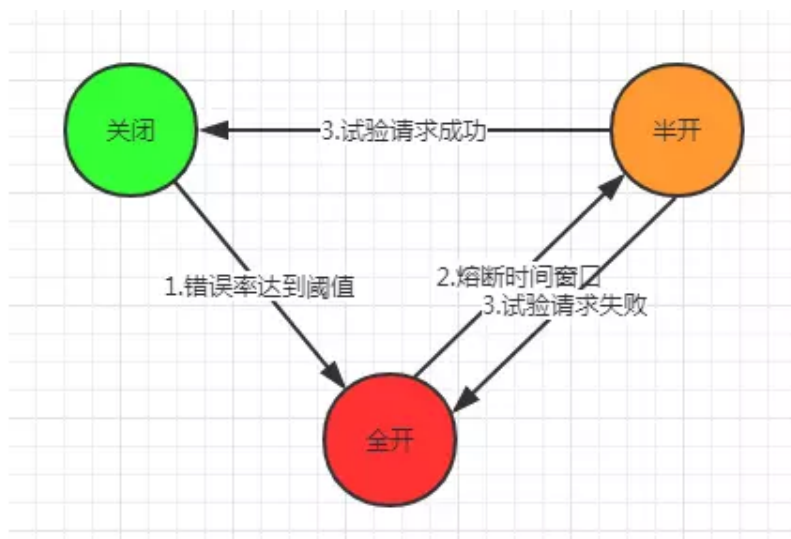
```
1 {
2   "id": 1,
3   "username": null,
4   "password": null,
5   "name": "熔断方法",
6   "age": null,
7   "sex": null,
8   "birthday": null,
9   "created": null,
10  "updated": null,
11  "note": null
12 }
```

## 6.4 熔断原理分析

熔断器的原理很简单，如同电力过载保护器。

熔断器状态机有3个状态：

- 关闭状态，所有请求正常访问
- 打开状态，所有请求都会被降级。
  - Hystrix会对请求情况计数，当一定时间失败请求百分比达到阈值，则触发熔断，断路器完全关闭
  - 默认失败比例的阈值是50%，请求次数最低不少于20次
- 半开状态
  - 打开状态不是永久的，打开一会后会进入休眠时间(默认5秒)。休眠时间过后会进入半开状态。
  - 半开状态：熔断器会判断下一次请求的返回状况，如果成功，熔断器切回关闭状态。如果失败，熔断器切回打开状态。



熔断器的核心解决方案：线程隔离和服务降级。

- 线程隔离。
- 服务降级(兜底方法)。

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

#### 什么时候熔断：

1. 访问超时
2. 服务不可用(死了)
3. 服务抛出异常(虽然有异常但还活着)
4. 其他请求导致服务异常到达阈值，所有服务都会被降级

模拟异常测试：<http://localhost:8080/consumer/1> 失败请求发送10次以上。再请求成功地址 <http://localhost:8080/consumer/2>，发现服务被熔断，会触发消费者 fallback方法。

## 6.5 扩展-服务降级的fallback方法：

将熔断方法编写在类上，对类的所有方法都生效。在方法上，仅对当前方法有效。



## 1. 方法上服务降级的fallback兜底方法

- 使用HystrixCommon注解，定义
- @HystrixCommand(fallbackMethod="fallBackMethod")用来声明一个降级逻辑的fallback兜底方法

## 2. 类上默认服务降级的fallback兜底方法

- 刚才把fallback写在了某个业务方法上，如果方法很多，可以将FallBack配置加在类上，实现默认FallBack
- @DefaultProperties(defaultFallback="defaultFallBack")，在类上，指明统一的失败降级方法；

```

@RestController
@RequestMapping("/consumer")
@DefaultProperties(defaultFallback = "defaultFallback") // 开启默认的
Fallback, 统一失败降级方法(兜底)
public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
    private DiscoveryClient discoveryClient;

    /**
     * 发送http请求调用provider服务根据id查找用户的接口
     */
    @GetMapping("/findUserById/{id}")
    @HystrixCommand
    public UserVO findUserById(@PathVariable Integer id){
        //String url = "http://127.0.0.1:9091/user/" + id;
        String url = "http://provider-service/user/" + id;
        UserVO user = restTemplate.getForObject(url, UserVO.class);
        return user;
    }

    // 熔断方法
    public UserVO fallbackMethod(Integer id){
        UserVO user = new UserVO();
        user.setId(id);
        user.setName("熔断方法");
        return user;
    }

    /**
     * 默认降级方法
     */
    public UserVO defaultFallback(){
        UserVO user = new UserVO();
        user.setName("默认提示: 对不起, 网络太拥挤了!");
        return user;
    }
}

```

```
1 {
2   "id": null,
3   "username": null,
4   "password": null,
5   "name": "默认提示: 对不起, 网络太拥挤了!",
6   "age": null,
7   "sex": null,
8   "birthday": null,
9   "created": null,
10  "updated": null,
11  "note": null
12 }
```

## 6.6 Hystrix Dashboard监控平台

Hystrix官方提供了基于图形化的DashBoard（仪表板）监控平台。Hystrix仪表板可以显示每个断路器（被@HystrixCommand注解的方法）的状态。

目标： 使用仪表板查看消费者服务请求情况

步骤：

1. 在pom中添加监控和仪表板坐标
2. 启动类开启仪表板
3. 配置文件暴露监控地址
4. 访问<http://localhost:8081/hystrix>
5. 添加需要监控的服务

过程：

- 在pom中添加监控和仪表板坐标

```

<!--监控坐标-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--仪表板坐标-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-
dashboard</artifactId>
</dependency>

```

- 启动类开启仪表板

```

@SpringBootApplication
@EnableDiscoveryClient // 声明是一个Eureka客户端
@EnableCircuitBreaker // 开启熔断器支持
@EnableHystrixDashboard // 开启仪表板支持
public class ConsumerServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerServiceApplication.class,
args);
    }

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

```

- 配置文件暴露监控地址

```

#暴露全部监控信息,可以只填写hystrix.stream, 也可以暴露全部 *, 注意在yaml文件
中特殊符号需要加引号
management.endpoints.web.exposure.include: '*'

```

- 访问<http://localhost:8080/hystrix> , 添加需要监控的服务  
<http://localhost:8080/actuator/hystrix.stream>



## Hystrix Dashboard

<http://localhost:8081/actuator/hystrix.stream>

所需监控的服务

Cluster via Turbine (default cluster): <https://turbine-hostname:port/turbine.stream>  
Cluster via Turbine (custom cluster): [https://turbine-hostname:port/turbine.stream?cluster=\[clusterName\]](https://turbine-hostname:port/turbine.stream?cluster=[clusterName])  
Single Hystrix App: <https://hystrix-app:port/actuator/hystrix.stream>

延迟

Delay:  ms

Title:

起个名称

填写完毕后点击

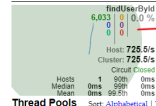
Hystrix Stream: <http://localhost:8081/actuator/hystrix.stream>



Circuit Set: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

颜色对应解释

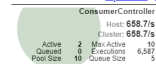
[Success](#) | [Short-Circuited](#) | [Bad Request](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | [Error %](#)



请求速率

熔断器状态

Thread Pools Set: [Alphabetical](#) | [Volume](#)



## 总结：

1. 理解SpringCloud设计初衷
  - 从架构的层面降低大型分布式系统的构建难度和要求
2. 能够搭建起微服务的业务场景，生产者消费者
  - 编写生产者【根据id查询用户信息】，消费者服务
  - 常见问题：
    - 请求url地址硬编码了
3. 理解Eureka解决的问题
  - 分布式系统的服务管理问题
  - 可以看到所有分布式系统中的微服务的状态！
4. 能够搭建Eureka注册中心服务
  - 导入Eureka的starter坐标
  - 开启Eureka的注解支持@EnableEurekaServer
  - 配置：配置应用名称，配置注册中心地址
5. 掌握Eureka心跳机制及相关参数的作用
  - 检测当前服务是否存活
  - 续约间隔：30秒
  - 服务时效剔除的时间：90秒
6. 理解Ribbon解决的问题
  - 解决集群服务负载均衡的问题
7. 能够测试Ribbon负载均衡访问效果：随机和轮询
  - 启动3个用户服务，开启@LoadBalanced
  - 负载均衡访问
8. 理解Hystrix解决的问题
  - 分布式系统中的异常捕获机制
  - 容错系统，提高分布式系统的整体弹性！
  - 解决分布式系统的雪崩效应！
9. 理解雪崩效应:PPT
  - 分布式系统的级联抛错现象
10. 了解熔断器原理分析
  - 了解
11. 能够写出Hystrix熔断器服务降级方法
  - @HystrixCommand(指定服务降级的处理方法)
  - 服务降级的处理方法：方法名称与方法的返回值必须与被降级的方法保持一致！