

Spring Cloud

- 1 微服务是什么
- 2 Spring Cloud是什么
- 3 **Spring Cloud Eureka**
- 4 Spring Cloud Ribbon
- 5 Spring Cloud OpenFeign
- 6 Spring Cloud Hystrix
- 7 Spring Cloud Gateway
- 8 Spring Cloud Config
- 9 Spring Cloud Alibaba是什么
- 10 Spring Cloud Alibaba Nacos
- 11 Spring Cloud Alibaba Sentinel
- 12 Spring Cloud Alibaba Seata

首页 > Spring Cloud

Eureka: Spring Cloud服务注册与发现组件 (非常详细)

< 上一节

下一节 >

Eureka 一词来源于古希腊词汇，是“发现了”的意思。在软件领域，Eureka 是 Netflix 公司开发的一款开源的服务注册与发现组件。

Spring Cloud 将 Eureka 与 Netflix 中的其他开源服务组件（例如 Ribbon、Feign 以及 Hystrix 等）一起整合进 Spring Cloud Netflix 模块中，整合后的组件全称为 Spring Cloud Netflix Eureka。

Eureka 是 Spring Cloud Netflix 模块的子模块，它是 Spring Cloud 对 Netflix Eureka 的二次封装，主要负责 Spring Cloud 的服务注册与发现功能。

Spring Cloud 使用 Spring Boot 思想为 Eureka 增加了自动化配置，开发人员只需要引入相关依赖和注解，就能将 Spring Boot 构建的微服务轻松地与 Eureka 进行整合。

Eureka 两大组件

Eureka 采用 CS (Client/Server, 客户端/服务器) 架构，它包括以下两大组件：

- **Eureka Server:** Eureka 服务注册中心，主要用于提供服务注册功能。当微服务启动时，会将自己的服务注册到 Eureka Server。Eureka Server 维护了一个可用服务列表，存储了所有注册到 Eureka Server 的可用服务的信息，这些可用服务可以在 Eureka Server 的管理界面中直观看到。
- **Eureka Client:** Eureka 客户端，通常指的是微服务系统中各个微服务，主要用于和 Eureka Server 进行交互。在微服务应用启动后，Eureka Client 会向 Eureka Server 发送心跳（默认周期为 30 秒）。若 Eureka Server 在多个心跳周期内没有接收到某个 Eureka Client 的心跳，Eureka Server 将它从可用服务列表中移除（默认 90 秒）。

注：“心跳”指的是一段定时发送的自定义信息，让对方知道自己“存活”，以确保连接的有效性。大部分 CS 架构的应用程序都采用了心跳机制，服务端和客户端都可以发心跳。通常情况下是客户端向服务器端发送心跳包，服务端用于判断客户端是否在线。

Eureka 服务注册与发现

Eureka 实现服务注册与发现的原理，如下图所示。

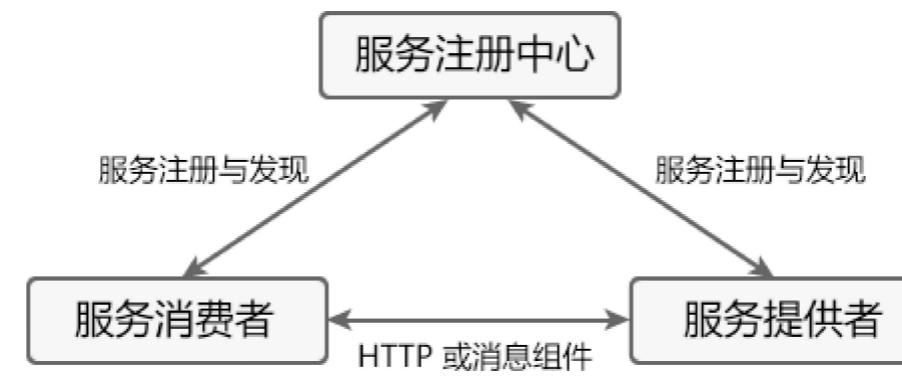


图1: Eureka 原理图

上图中共涉及到以下 3 个角色：

- **服务注册中心 (Register Service)**：它是一个 Eureka Server，用于提供服务注册和发现功能。



- **服务提供者 (Provider Service)**：它是一个 Eureka Client，用于提供服务。它将自己提供的服务注册到服务注册中心，以供服务消费者发现。
- **服务消费者 (Consumer Service)**：它是一个 Eureka Client，用于消费服务。它可以从服务注册中心获取服务列表，调用所需的服务。

Eureka 实现服务注册与发现的流程如下：

1. 搭建一个 Eureka Server 作为服务注册中心；
2. 服务提供者 Eureka Client 启动时，会把当前服务器的信息以服务名（spring.application.name）的方式注册到服务注册中心；
3. 服务消费者 Eureka Client 启动时，也会向服务注册中心注册；
4. 服务消费者还会获取一份可用服务列表，该列表中包含了所有注册到服务注册中心的服务信息（包括服务提供者和自身的信息）；
5. 在获得了可用服务列表后，服务消费者通过 HTTP 或消息中间件远程调用服务提供者提供的服务。

服务注册中心（Eureka Server）所扮演的角色十分重要，它是服务提供者和服务消费者之间的桥梁。服务提供者只有将自己的服务注册到服务注册中心才可能被服务消费者调用，而服务消费者也只有通过服务注册中心获取可用服务列表后，才能调用所需的服务。

示例 1

下面，我们通过一个案例来展示下 Eureka 是如何实现服务注册与发现的。

1. 创建主工程 (Maven Project)

由于本案例中，会涉及到多个由 Spring Boot 创建的微服务，为了方便管理，这里我们采用 Maven 的多 Module 结构（即一个 Project 包含多个 Module）来构建工程。

创建一个名为 spring-cloud-demo2 的 Maven 主工程，然后在该主工程的 pom.xml 中使用 dependencyManagement 来管理 Spring Cloud 的版本，内容如下。

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0"
03.           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
05.     <modelVersion>4.0.0</modelVersion>
06.     <packaging>pom</packaging>
07.     <modules>
08.       <module>micro-service-cloud-api</module>
09.     </modules>
10.     <parent>
11.       <groupId>org.springframework.boot</groupId>
12.       <artifactId>spring-boot-starter-parent</artifactId>
13.       <version>2.3.6.RELEASE</version>
14.       <relativePath/> <!-- lookup parent from repository -->
15.     </parent>
16.     <groupId>net.biancheng.c</groupId>
17.     <artifactId>spring-cloud-demo2</artifactId>
18.     <version>0.0.1-SNAPSHOT</version>
19.
20.     <properties>
21.       <maven.compiler.source>8</maven.compiler.source>
22.       <maven.compiler.target>8</maven.compiler.target>
23.       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
```



```

24.      <maven.compiler.source>1.8</maven.compiler.source>
25.      <maven.compiler.target>1.8</maven.compiler.target>
26.      <junit.version>4.12</junit.version>
27.      <log4j.version>1.2.17</log4j.version>
28.      <lombok.version>1.16.18</lombok.version>
29.  </properties>
30.
31.  <dependencyManagement>
32.    <dependencies>
33.      <!--在主工程中使用 dependencyManagement 声明 Spring Cloud 的版本,
34.          这样工程内的 Module 中引入 Spring Cloud 组件依赖时，就不必在声明组件的版本信息
35.          保证 Spring Cloud 各个组件一致性-->
36.      <dependency>
37.        <groupId>org.springframework.cloud</groupId>
38.        <artifactId>spring-cloud-dependencies</artifactId>
39.        <version>Hoxton.SR12</version>
40.        <type>pom</type>
41.        <scope>import</scope>
42.      </dependency>
43.    </dependencies>
44.  </dependencyManagement>
45.  <build>
46.    <finalName>microservicecloud</finalName>
47.    <resources>
48.      <resource>
49.        <directory>src/main/resources</directory>
50.        <filtering>true</filtering>
51.      </resource>
52.    </resources>
53.    <plugins>
54.      <plugin>
55.        <groupId>org.apache.maven.plugins</groupId>
56.        <artifactId>maven-resources-plugin</artifactId>
57.        <configuration>
58.          <delimiters>
59.            <delimit>$</delimit>
60.          </delimiters>
61.        </configuration>
62.      </plugin>
63.    </plugins>
64.  </build>
65. </project>

```

2. 创建公共子模块 (Maven Module)

1) 在主工程下，创建一个名为 micro-service-cloud-api 的 Maven Module: micro-service-cloud-api，其 pom.xml 配置如下。

```
01. <?xml version="1.0" encoding="UTF-8"?>
```



```

02. <project xmlns="http://maven.apache.org/POM/4.0.0"
03.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
05.   <parent>
06.     <artifactId>spring-cloud-demo2</artifactId>
07.     <groupId>net.biancheng.c</groupId>
08.     <version>0.0.1-SNAPSHOT</version>
09.   </parent>
10.   <modelVersion>4.0.0</modelVersion>
11.   <artifactId>micro-service-cloud-api</artifactId>
12.   <properties>
13.     <maven.compiler.source>8</maven.compiler.source>
14.     <maven.compiler.target>8</maven.compiler.target>
15.   </properties>
16.   <dependencies>
17.     <dependency>
18.       <groupId>org.projectlombok</groupId>
19.       <artifactId>lombok</artifactId>
20.     </dependency>
21.   </dependencies>
22. </project>

```

注： micro-service-cloud-api 是整个工程的公共子模块，它包含了一些其他子模块共有的内容，例如实体类、公共工具类、公共依赖项等。当其他子模块需要使用公共子模块中的内容时，只需要在其 pom.xml 引入公共子模块的依赖即可。

2) 在 micro-service-cloud-api 的 net.biancheng.c.entity 包下，创建一个名为 Dept 的实体类，代码如下。

```

01. package net.biancheng.c.entity;
02.
03. import lombok.Data;
04. import lombok.NoArgsConstructor;
05. import lombok.experimental.Accessors;
06.
07. import java.io.Serializable;
08.
09. @NoArgsConstructor //无参构造函数
10. @Data // 提供类的get、set、equals、hashCode、canEqual、toString 方法
11. @Accessors(chain = true)
12. public class Dept implements Serializable {
13.     private Integer deptNo;
14.     private String deptName;
15.     private String dbSource;
16. }

```

3. 搭建服务注册中心

1) 在主工程下创建一个名为 micro-service-cloud-eureka-7001 的 Spring Boot Module 作为服务注册中心，并在其 pom.xml 中引入以下依赖。



```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
04.   <modelVersion>4.0.0</modelVersion>
05.   <!--继承主工程的 POM-->
06.   <parent>
07.     <artifactId>spring-cloud-demo2</artifactId>
08.     <groupId>net.biancheng.c</groupId>
09.     <version>0.0.1-SNAPSHOT</version>
10.   </parent>
11.   <groupId>net.biancheng.c</groupId>
12.   <artifactId>micro-service-cloud-eureka-7001</artifactId>
13.   <version>0.0.1-SNAPSHOT</version>
14.   <name>micro-service-cloud-eureka-7001</name>
15.   <description>Demo project for Spring Boot</description>
16.
17.   <properties>
18.     <java.version>1.8</java.version>
19.   </properties>
20.
21.   <dependencies>
22.     <dependency>
23.       <groupId>org.springframework.boot</groupId>
24.       <artifactId>spring-boot-starter-web</artifactId>
25.     </dependency>
26.     <!--为服务注册中心引入 Eureka Server 的依赖-->
27.     <dependency>
28.       <groupId>org.springframework.cloud</groupId>
29.       <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
30.     </dependency>
31.     <!--devtools 和 lombok 均为开发辅助模块，根据需求适当选择-->
32.     <dependency>
33.       <groupId>org.springframework.boot</groupId>
34.       <artifactId>spring-boot-devtools</artifactId>
35.       <scope>runtime</scope>
36.       <optional>true</optional>
37.     </dependency>
38.     <dependency>
39.       <groupId>org.projectlombok</groupId>
40.       <artifactId>lombok</artifactId>
41.       <optional>true</optional>
42.     </dependency>
43.     <dependency>
44.       <groupId>org.springframework.boot</groupId>
45.       <artifactId>spring-boot-starter-test</artifactId>
46.       <scope>test</scope>
```



```
47.      </dependency>
48.  </dependencies>
49.
50.  <build>
51.    <plugins>
52.      <plugin>
53.        <groupId>org.springframework.boot</groupId>
54.        <artifactId>spring-boot-maven-plugin</artifactId>
55.        <configuration>
56.          <excludes>
57.            <exclude>
58.              <groupId>org.projectlombok</groupId>
59.              <artifactId>lombok</artifactId>
60.            </exclude>
61.          </excludes>
62.        </configuration>
63.      </plugin>
64.    </plugins>
65.  </build>
66. </project>
```

2) 在 micro-service-cloud-eureka-7001 的类路径 (/resources 目录) 下, 添加一个配置文件 application.yml, 配置内容如下。

```
01. server:
02.   port: 7001 #该 Module 的端口号
03.
04. eureka:
05.   instance:
06.     hostname: localhost #eureka服务端的实例名称,
07.
08. client:
09.   register-with-eureka: false #false表示不向注册中心注册自己。
10.   fetch-registry: false #false表示自己端就是注册中心, 我的职责就是维护服务实例, 并不需要去检索服务
11.   service-url:
12.     defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #单机版服务注册中心
```

3) 在 micro-service-cloud-eureka-7001 的主启动类上使用 @EnableEurekaServer 注解开启服务注册中心功能, 接受其他服务的注册, 代码如下。

```
01. package net.biancheng.c;
02.
03. import org.springframework.boot.SpringApplication;
04. import org.springframework.boot.autoconfigure.SpringBootApplication;
05. import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
06.
07. @SpringBootApplication
08. @EnableEurekaServer //开启 Eureka server, 接受其他微服务的注册
```



```

9. public class MicroServiceCloudEureka7001Application {
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(MicroServiceCloudEureka7001Application.class, args);
13.     }
14. }

```

4) 启动 micro-service-cloud-eureka-7001，使用浏览器访问 Eureka 服务注册中心主页，地址为 “<http://localhost:7001/>” , 结果如下图。

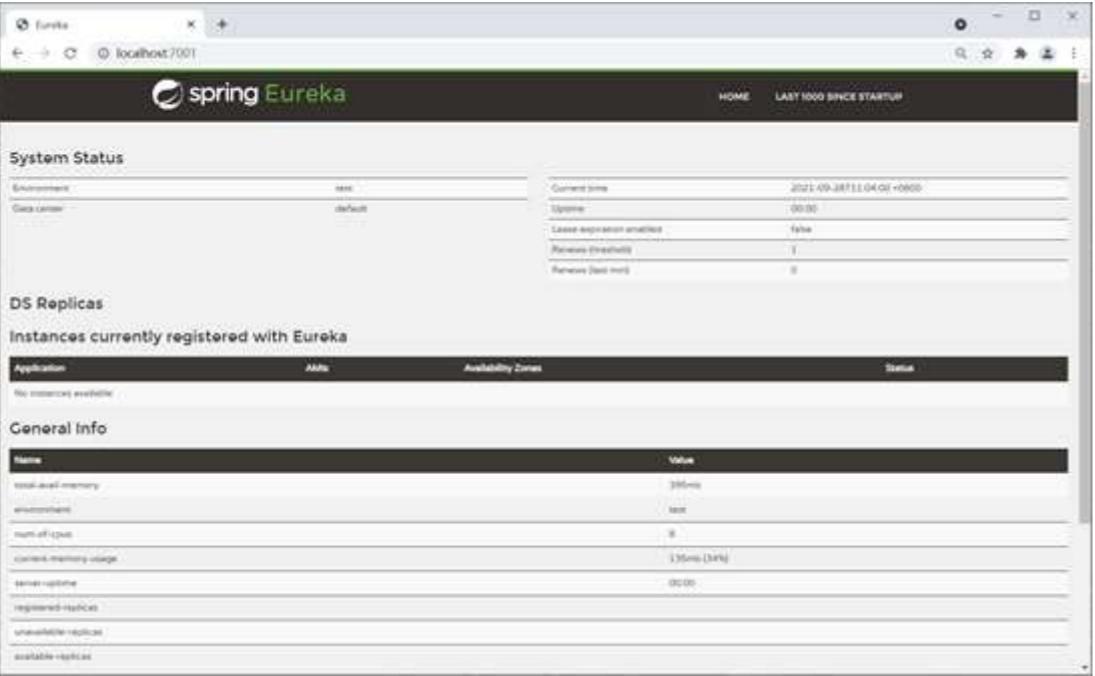


图2: Eureka 7001 服务注册中心

4. 搭建服务提供者

1) 在主工程下创建一个名为 micro-service-cloud-provider-dept-8001 的 Spring Boot Module，并在其 pom.xml 中引入以下依赖。

```

01. <?xml version="1.0" encoding="UTF-8"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.           xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
04.     <modelVersion>4.0.0</modelVersion>
05.     <!--引入父工程pom-->
06.     <parent>
07.       <artifactId>spring-cloud-demo2</artifactId>
08.       <groupId>net.biancheng.c</groupId>
09.       <version>0.0.1-SNAPSHOT</version>
10.     </parent>
11.
12.     <groupId>net.biancheng.c</groupId>
13.     <artifactId>micro-service-cloud-provider-dept-8001</artifactId>
14.     <version>0.0.1-SNAPSHOT</version>
15.     <name>micro-service-cloud-provider-dept-8001</name>
16.     <description>Demo project for Spring Boot</description>
17.

```



```
18. <properties>
19.     <java.version>1.8</java.version>
20. </properties>
21. <dependencies>
22.     <!--Spring Boot Web-->
23.     <dependency>
24.         <groupId>org.springframework.boot</groupId>
25.         <artifactId>spring-boot-starter-web</artifactId>
26.     </dependency>
27.     <!--devtools 开发工具-->
28.     <dependency>
29.         <groupId>org.springframework.boot</groupId>
30.         <artifactId>spring-boot-devtools</artifactId>
31.         <scope>runtime</scope>
32.         <optional>true</optional>
33.     </dependency>
34.     <!--Spring Boot 测试-->
35.     <dependency>
36.         <groupId>org.springframework.boot</groupId>
37.         <artifactId>spring-boot-starter-test</artifactId>
38.         <scope>test</scope>
39.     </dependency>
40.     <!--引入公共子模块-->
41.     <dependency>
42.         <groupId>net.biancheng.c</groupId>
43.         <artifactId>micro-service-cloud-api</artifactId>
44.         <version>${project.version}</version>
45.     </dependency>
46.     <!--junit 测试-->
47.     <dependency>
48.         <groupId>junit</groupId>
49.         <artifactId>junit</artifactId>
50.         <version>4.12</version>
51.     </dependency>
52.     <!--mysql 驱动-->
53.     <dependency>
54.         <groupId>mysql</groupId>
55.         <artifactId>mysql-connector-java</artifactId>
56.         <version>5.1.49</version>
57.     </dependency>
58.     <!--logback 日志-->
59.     <dependency>
60.         <groupId>ch.qos.logback</groupId>
61.         <artifactId>logback-core</artifactId>
62.     </dependency>
63.     <!--整合 mybatis -->
```



```
64.    <dependency>
65.        <groupId>org.mybatis.spring.boot</groupId>
66.        <artifactId>mybatis-spring-boot-starter</artifactId>
67.        <version>2.2.0</version>
68.    </dependency>
69.    <!-- 修改后立即生效，热部署 -->
70.    <dependency>
71.        <groupId>org.springframework</groupId>
72.        <artifactId>springloaded</artifactId>
73.        <version>1.2.8.RELEASE</version>
74.    </dependency>
75.    <!--引入 Eureka Client 的依赖，将服务注册到 Eureka Server-->
76.    <dependency>
77.        <groupId>org.springframework.cloud</groupId>
78.        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
79.    </dependency>
80.    <!-- Spring Boot 监控模块-->
81.    <dependency>
82.        <groupId>org.springframework.boot</groupId>
83.        <artifactId>spring-boot-starter-actuator</artifactId>
84.    </dependency>
85. </dependencies>
86.
87. <build>
88.     <plugins>
89.         <!--mybatis自动生成代码插件-->
90.         <plugin>
91.             <groupId>org.mybatis.generator</groupId>
92.             <artifactId>mybatis-generator-maven-plugin</artifactId>
93.             <version>1.4.0</version>
94.             <configuration>
95.                 <configurationFile>src/main/resources/mybatis-generator/generatorConfig.xml</configurationFile>
96.                 <verbose>true</verbose>
97.                 <!-- 是否覆盖，true表示会替换生成的JAVA文件，false则不覆盖 -->
98.                 <overwrite>true</overwrite>
99.             </configuration>
100.            <dependencies>
101.                <!--mysql驱动包-->
102.                <dependency>
103.                    <groupId>mysql</groupId>
104.                    <artifactId>mysql-connector-java</artifactId>
105.                    <version>5.1.49</version>
106.                </dependency>
107.                <dependency>
108.                    <groupId>org.mybatis.generator</groupId>
109.                    <artifactId>mybatis-generator-core</artifactId>
```



```

110.          <version>1.4.0</version>
111.      </dependency>
112.  </dependencies>
113. </plugin>
114. <plugin>
115.     <groupId>org.springframework.boot</groupId>
116.     <artifactId>spring-boot-maven-plugin</artifactId>
117.   </plugin>
118. </plugins>
119. </build>
120. </project>

```

2) 在 micro-service-cloud-provider-dept-8001 类路径 (/resources 目录) 下, 添加配置文件 application.yml, 配置内容如下。

```

01. server:
02.   port: 8001 #服务端口号
03. spring:
04.   application:
05.     name: microServiceCloudProviderDept #微服务名称, 对外暴露的微服务名称, 十分重要
06. ##### JDBC 通用配置 #####
07.   datasource:
08.     username: root      #数据库登陆用户名
09.     password: root      #数据库登陆密码
10.    url: jdbc:mysql://127.0.0.1:3306/bianchengbang_jdbc      #数据库url
11.    driver-class-name: com.mysql.jdbc.Driver      #数据库驱动
12.
13. ##### 不检查 spring.config.import=configserver:#####
14. # cloud:
15. #   config:
16. #     enabled: false
17. ##### MyBatis 配置 #####
18. mybatis:
19.   # 指定 mapper.xml 的位置
20.   mapper-locations: classpath:mybatis/mapper/*.xml
21.   #扫描实体类的位置,在此处指明扫描实体类的包,在 mapper.xml 中就可以不写实体类的全路径名
22.   type-aliases-package: net.biancheng.c.entity
23.   configuration:
24.     #默认开启驼峰命名法, 可以不用设置该属性
25.     map-underscore-to-camel-case: true
26. ##### Spring cloud 自定义服务名称和 ip 地址 #####
27. eureka:
28.   client: #将客户端注册到 eureka 服务列表内
29.   service-url:
30.     defaultZone: http://eureka7001.com:7001/eureka #这个地址是 7001注册中心在 application.yml 中暴露出来额注册地址 (单机
版)

```



```
31.  
32. instance:  
33.   instance-id: spring-cloud-provider-8001 #自定义服务名称信息  
34.   prefer-ip-address: true #显示访问路径的 ip 地址  
35. ##### spring cloud 使用 Spring Boot actuator 监控完善信息  
36. #####  
37. # Spring Boot 2.50对 actuator 监控屏蔽了大多数的节点，只暴露了 heath 节点，本段配置 (*) 就是为了开启所有的节点  
38. management:  
39.   endpoints:  
40.     web:  
41.       exposure:  
42.         include: "*" # * 在yaml 文件属于关键字，所以需要加引号  
43.     info:  
44.       app.name: micro-service-cloud-provider-dept  
45.       company.name: c.biancheng.net  
46.       build.aetifactId: @project.artifactId@  
47.       build.version: @project.version@
```

3) 在 net.biancheng.c.mapper 包下创建一个名为 DeptMapper 的接口，代码如下。

```
01. package net.biancheng.c.mapper;  
02.  
03. import net.biancheng.c.entity.Dept;  
04. import org.apache.ibatis.annotations.Mapper;  
05.  
06. import java.util.List;  
07.  
08. @Mapper  
09. public interface DeptMapper {  
10.   //根据主键获取数据  
11.   Dept selectByPrimaryKey(Integer deptNo);  
12.  
13.   //获取表中的全部数据  
14.   List<Dept> GetAll();  
15. }
```

4) 在 resources/mybatis/mapper/ 目录下，创建一个名为 DeptMapper.xml 的 MyBatis 映射文件，配置内容如下。

```
01. <?xml version="1.0" encoding="UTF-8"?>  
02. <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  
03. <mapper namespace="net.biancheng.c.mapper.DeptMapper">  
04.   <resultMap id="BaseResultMap" type="net.biancheng.c.entity.Dept">  
05.     <id column="dept_no" jdbcType="INTEGER" property="deptNo"/>  
06.     <result column="dept_name" jdbcType="VARCHAR" property="deptName"/>  
07.     <result column="db_source" jdbcType="VARCHAR" property="dbSource"/>  
08.   </resultMap>
```



```
09.  
10. <sql id="Base_Column_List">  
11.     dept_no  
12.     , dept_name, db_source  
13. </sql>  
14.  
15. <select id="selectByPrimaryKey" parameterType="java.lang.Integer" resultMap="BaseResultMap">  
16.     select  
17.         <include refid="Base_Column_List"/>  
18.     from dept  
19.     where dept_no = #{deptNo, jdbcType=INTEGER}  
20. </select>  
21. <select id="GetAll" resultType="net.biancheng.c.entity.Dept">  
22.     select *  
23.     from dept;  
24. </select>  
25. </mapper>
```

5) 在 net.biancheng.c.service 包下创建一个名为 DeptService 的接口，代码如下。

```
01. package net.biancheng.c.service;  
02.  
03. import net.biancheng.c.entity.Dept;  
04. import java.util.List;  
05.  
06. public interface DeptService {  
07.  
08.     Dept get(Integer deptNo);  
09.  
10.    List<Dept> selectAll();  
11. }
```

6) 在 net.biancheng.c.service.impl 包下创建 DeptService 接口的实现类 DeptServiceImpl，代码如下。

```
01. package net.biancheng.c.service.impl;  
02.  
03. import net.biancheng.c.entity.Dept;  
04. import net.biancheng.c.mapper.DeptMapper;  
05. import net.biancheng.c.service.DeptService;  
06. import org.springframework.beans.factory.annotation.Autowired;  
07. import org.springframework.stereotype.Service;  
08.  
09. import java.util.List;  
10.  
11. @Service("deptService")  
12. public class DeptServiceImpl implements DeptService {
```



```
13.     @Autowired
14.     private DeptMapper deptMapper;
15.
16.     @Override
17.     public Dept get(Integer deptNo) {
18.         return deptMapper.selectByPrimaryKey(deptNo);
19.     }
20.
21.     @Override
22.     public List<Dept> selectAll() {
23.         return deptMapper.GetAll();
24.     }
25. }
```

7) 在 net.biancheng.c.controller 包下创建一个名为 DeptController 的 Controller 类, 代码如下。

```
01. package net.biancheng.c.controller;
02.
03. import lombok.extern.slf4j.Slf4j;
04. import net.biancheng.c.entity.Dept;
05. import net.biancheng.c.service.DeptService;
06. import org.springframework.beans.factory.annotation.Autowired;
07. import org.springframework.beans.factory.annotation.Value;
08. import org.springframework.web.bind.annotation.*;
09.
10. import java.util.List;
11.
12. /**
13. * 服务提供者的控制层
14. * author:c语言中文网 c.biancheng.net
15. */
16. @RestController
17. @Slf4j
18. public class DeptController {
19.     @Autowired
20.     private DeptService deptService;
21.
22.     @Value("${server.port}")
23.     private String serverPort;
24.
25.     @RequestMapping(value = "/dept/get/{id}", method = RequestMethod.GET)
26.     public Dept get(@PathVariable("id") int id) {
27.         return deptService.get(id);
28.     }
29.
30.     @RequestMapping(value = "/dept/list", method = RequestMethod.GET)
```



```
31.     public List<Dept> list() {
32.         return deptService.selectAll();
33.     }
34. }
```

8) 在 micro-service-cloud-provider-dept-8001 的主启动类上，使用 @EnableEurekaClient 注解开启 Eureka 客户端功能，将服务注册到服务注册中心 (Eureka Server)，代码如下。

```
01. package net.biancheng.c;
02.
03. import org.springframework.boot.SpringApplication;
04. import org.springframework.boot.autoconfigure.SpringBootApplication;
05. import org.springframework.cloud.netflix.eureka.EnableEurekaClient;
06.
07. @SpringBootApplication
08. @EnableEurekaClient // Spring cloud Eureka 客户端，自动将本服务注册到 Eureka Server 注册中心中
09. public class MicroServiceCloudProviderDept8001Application {
10.
11.     public static void main(String[] args) {
12.         SpringApplication.run(MicroServiceCloudProviderDept8001Application.class, args);
13.     }
14. }
```

9) 依次启动 micro-service-cloud-eureka-7001 和 micro-service-cloud-provider-dept-8001，使用浏览器访问再次访问 Eureka 服务注册中心主页 (<http://localhost:7001/>)，如下图。

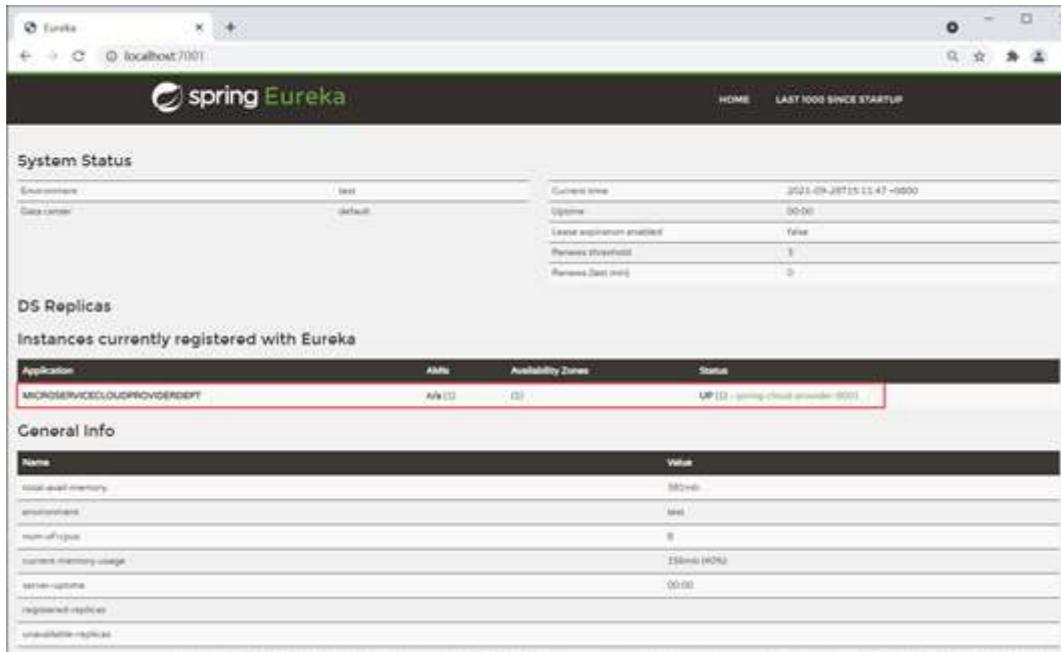


图3：服务提供者注册到服务注册中心

从图 3 可以看到， Instances currently registered with Eureka (注册到 Eureka Server 的实例) 选项中已经包含了一条服务信息，即已经有服务注册到 Eureka Server 上了。

Instances currently registered with Eureka 选项中包含以下内容：

- Application: MICROSERVICECLOUDPROVIDERDEPT, 该取值为 micro-service-cloud-provider-dept-8001 配置文件 application.yml 中 spring.application.name 的取值。
- Status: UP (1) - spring-cloud-provider-8001, UP 表示服务在线, (1) 表示有集群中服务的数量, spring-cloud-provider-8001 则是 micro-service-cloud-provider-dept-8001 配置文件 application.yml 中 eureka.instance.instance-id 的取值。

10) 在 MySQL 的 bianchengbang_jdbc 数据库中执行以下 SQL, 准备测试数据。

```
DROP TABLE IF EXISTS `dept`;  
CREATE TABLE `dept` (  
    `dept_no` int NOT NULL AUTO_INCREMENT,  
    `dept_name` varchar(255) DEFAULT NULL,  
    `db_source` varchar(255) DEFAULT NULL,  
    PRIMARY KEY (`dept_no`)  
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;  
  
INSERT INTO `dept` VALUES ('1', '开发部', 'bianchengbang_jdbc');  
INSERT INTO `dept` VALUES ('2', '人事部', 'bianchengbang_jdbc');  
INSERT INTO `dept` VALUES ('3', '财务部', 'bianchengbang_jdbc');  
INSERT INTO `dept` VALUES ('4', '市场部', 'bianchengbang_jdbc');  
INSERT INTO `dept` VALUES ('5', '运维部', 'bianchengbang_jdbc');
```

11) 使用浏览器访问 “<http://localhost:8001/dept/list>” , 结果如下图。

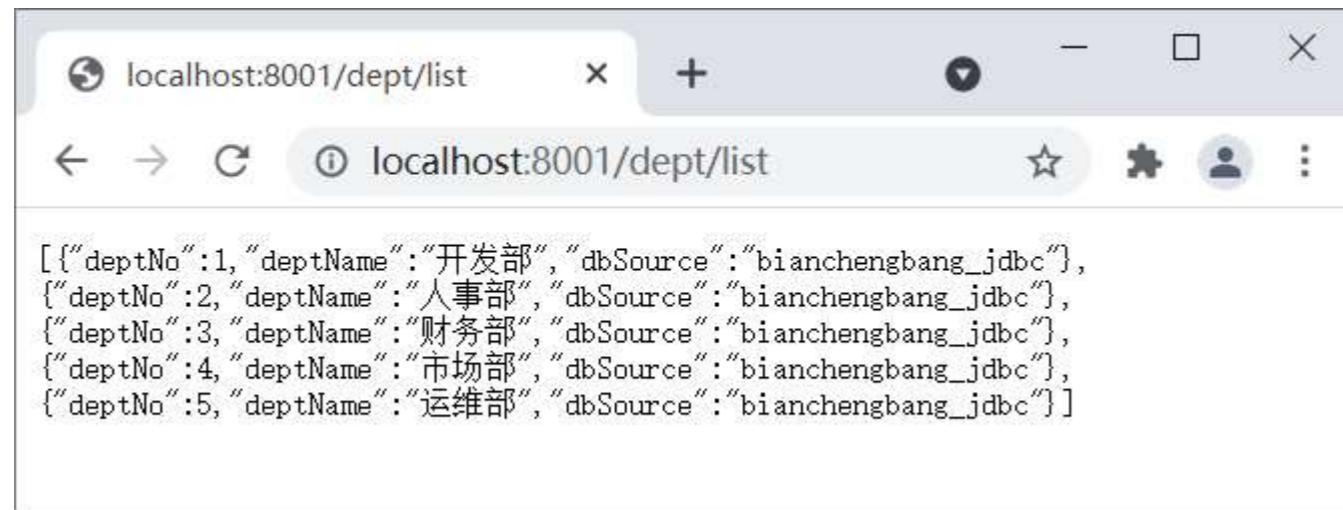


图4：服务提供者提供服务访问数据库

Eureka Server 集群

在微服务架构中, 一个系统往往由十几甚至几十个服务组成, 若将这些服务全部注册到同一个 Eureka Server 中, 就极有可能导致 Eureka Server 因不堪重负而崩溃, 最终导致整个系统瘫痪。解决这个问题最直接的办法就是部署 Eureka Server 集群。

我们知道, 在 Eureka 实现服务注册与发现时一共涉及了 3 个角色: 服务注册中心、服务提供者以及服务消费者, 这三个角色分工明确, 各司其职。但是其

在 Eureka 中，所有服务都既是服务消费者也是服务提供者，服务注册中心 Eureka Server 也不例外。

我们在搭建服务注册中心时，在 application.yml 中涉及了这样的配置：

```
01. eureka:  
02.   client:  
03.     register-with-eureka: false #false 表示不向注册中心注册自己。  
04.     fetch-registry: false #false 表示自己端就是注册中心，职责就是维护服务实例，并不需要去检索服务
```

这样设置的原因是 micro-service-cloud-eureka-7001 本身自己就是服务注册中心，服务注册中心是不能将自己注册到自己身上的，但服务注册中心是可以将自己作为服务向其他的服务注册中心注册自己的。

举个例子，有两个 Eureka Server 分别为 A 和 B，虽然 A 不能将自己注册到 A 上，B 也不能将自己注册到 B 上，但 A 是可以作为一个服务把自己注册到 B 上的，同理 B 也可以将自己注册到 A 上。

这样就可以形成一组互相注册的 Eureka Server 集群，当服务提供者发送注册请求到 Eureka Server 时，Eureka Server 会将请求转发给集群中所有与之相连的 Eureka Server 上，以实现 Eureka Server 之间的服务同步。

通过服务同步，服务消费者可以在集群中的任意一台 Eureka Server 上获取服务提供者提供的服务。这样，即使集群中的某个服务注册中心发生故障，服务消费者仍然可以从集群中的其他 Eureka Server 中获取服务信息并调用，而不会导致系统的整体瘫痪，这就是 Eureka Server 集群的高可用性。

示例 2

下面我们在示例 1 的基础上进行扩展，构建一个拥有 3 个 Eureka Server 实例的集群。

1. 参照 micro-service-cloud-eureka-7001 的搭建过程，在主工程下另外再创建两个 Eureka Server：micro-service-cloud-eureka-7002 和 micro-service-cloud-eureka-7003，此时这 3 个 Eureka Server 无论是 Maven 依赖、代码还是配置都是一模一样的。
2. 修改 micro-service-cloud-eureka-7001、micro-service-cloud-eureka-7002、micro-service-cloud-eureka-7003 中 application.yml 的配置，具体配置如下。

micro-service-cloud-eureka-7001 中 application.yml 的配置如下。

```
01. server:  
02.   port: 7001 #端口号  
03.  
04. eureka:  
05.   instance:  
06.     hostname: eureka7001.com #eureka服务端的实例名称  
07.  
08.   client:  
09.     register-with-eureka: false #false 表示不向注册中心注册自己。  
10.     fetch-registry: false #false 表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务  
11.     service-url:  
12.       #defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #单机版
```



13. defaultZone: http://eureka7002.com:7002/eureka/, http://eureka7003.com:7003/eureka/ #集群版 将当前的 Eureka Server 注册到 7003 和 7003 上, 形成一组互相注册的 Eureka Server 集群

micro-service-cloud-eureka-7002 中 application.yml 的配置如下。

```
01. server:  
02.   port: 7002 #端口号  
03.  
04. eureka:  
05.   instance:  
06.     hostname: eureka7002.com #Eureka Server 实例名称  
07.  
08.   client:  
09.     register-with-eureka: false #false 表示不向注册中心注册自己。  
10.    fetch-registry: false #false 表示自己端就是注册中心, 我的职责就是维护服务实例, 并不需要去检索服务  
11.    service-url:  
12.      defaultZone: http://eureka7001.com:7001/eureka/, http://eureka7003.com:7003/eureka/ #将这个 Eureka Server 注册到 7001 和  
7003 上
```

micro-service-cloud-eureka-7003 中 application.yml 的配置如下。

```
01. server:  
02.   port: 7003 #端口号  
03.  
04. eureka:  
05.   instance:  
06.     hostname: eureka7003.com #Eureka Server 实例名称  
07.  
08.   client:  
09.     register-with-eureka: false #false 表示不向注册中心注册自己。  
10.    fetch-registry: false #false 表示自己端就是注册中心, 我的职责就是维护服务实例, 并不需要去检索服务  
11.    service-url:  
12.      defaultZone: http://eureka7001.com:7001/eureka/, http://eureka7002.com:7002/eureka/ #将这个 Eureka Server 注册到 7001 和  
7002 上
```

3. 由于我们是在本地搭建的 Eureka Server 集群, 因此我们需要修改本地的 host 文件, Windows 操作系统的电脑在 C:/Windows/System/drivers/etc/hosts 中修改, Mac 系统的电脑则需要在 vim/etc/hosts 中修改, 修改内容如下。

```
#Spring Cloud eureka 集群  
127.0.0.1 eureka7001.com  
127.0.0.1 eureka7002.com  
127.0.0.1 eureka7003.com
```

4. 修改 micro-service-cloud-provider-dept-8001 (服务提供者) 配置文件 application.yml 中 eureka.client.service-url.defaultZone 的取值, 将服务注册到 Eureka Server 集群上, 具体配置如下。



```

01. eureka:
02.   client: #将客户端注册到 eureka 服务列表内
03.   service-url:
04.     #defaultZone: http://eureka7001.com:7001/eureka #这个地址是 7001 注册中心在 application.yml 中暴露出来的注册地址 (单机
版)
05.     defaultZone: http://eureka7001.com:7001/eureka/, http://eureka7002.com:7002/eureka/, http://eureka7003.com:7003/eureka/ #
将服务注册到 Eureka Server 集群

```

5. 启动 micro-service-cloud-eureka-7001，使用浏览器访问 “<http://eureka7001.com:7001/>” , 结果如下图。

The screenshot shows the Spring Eureka dashboard with the following sections:

- System Status:** Displays environment (test), data center (default), current time (2021-09-29T10:26:01 +0800), uptime (00:00), lease expiration enabled (false), renew threshold (3), and renew (last min) (0).
- DS Replicas:** Shows two entries: "eureka7003.com" and "eureka7002.com", both highlighted with red boxes.
- Instances currently registered with Eureka:** A table with columns Application, AMIs, Availability Zones, and Status. It shows one instance: "MICROSERVICECLOUDPROVIDERDEPT" with AMIs (n/a (1)), Availability Zones (1), and Status (UP (1) - spring-cloud-provider-8001).
- General Info:** A table of system metrics including total-available-memory (377mb), environment (test), num-of-cpus (8), current-memory-usage (254mb (67%)), server-upptime (00:00), registered-replicas (http://eureka7003.com:7003/eureka/, http://eureka7002.com:7002/eureka/), and unavailable-replicas (http://eureka7003.com:7003/eureka/, http://eureka7002.com:7002/eureka/).

图5: Eureka Server 集群 -7001

从上图可以看到，服务提供者 (micro-service-cloud-provider-dept-8001) 的服务已经注册到了 Eureka Server 7001，并且在 DS Replicas 选项中也显示了集群中的另外两个 Eureka Server: Eureka Server 7002 和 Eureka Server 7003。

6. 启动 micro-service-cloud-eureka-7002，使用浏览器访问 “<http://eureka7002.com:7002/>” , 结果如下图。

The screenshot shows the Spring Eureka web interface. At the top, it displays the current time as 2021-09-29T10:28:09 +0800 and uptime as 00:02. In the 'DS Replicas' section, three instances are listed: eureka7003.com (selected), eureka7001.com, and eureka7002.com. The 'Instances currently registered with Eureka' table shows one instance: MICROSERVICECLOUDPROVIDERDEPT with AMIs n/a (1) and Availability Zones (1), and Status UP (1) - spring-cloud-provider-8001. The 'General Info' section lists various system metrics.

Name	Value
total-avail-memory	397mb
environment	test
num-of-cpus	8
current-memory-usage	104mb (26%)
server-upptime	00:02
registered-replicas	http://eureka7003.com:7003/eureka/, http://eureka7001.com:7001/eureka/
unavailable-replicas	http://eureka7003.com:7003/eureka/, http://eureka7001.com:7001/eureka/,

图6: Eureka Server 集群 -7002

从上图可以看到，服务提供者（micro-service-cloud-provider-dept-8001）所提供的服务已经注册到了 Eureka Server 7002，并且在 DS Replicas 选项中也显示了集群中的另外两个 Eureka Server: Eureka Server 7001 和 Eureka Server 7003。

7. 启动 micro-service-cloud-eureka-7003，使用浏览器访问 “<http://eureka7003.com:7003/>” ，结果如下图。

The screenshot shows the Spring Eureka dashboard at <http://eureka7003.com:7003>. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into several sections:

- System Status:** Displays environment (test), data center (default), current time (2021-09-29T10:29:32 +0800), uptime (00:03), lease expiration enabled (false), renew threshold (3), and renew (last min) (2).
- DS Replicas:** Shows two entries: eureka7001.com and eureka7002.com.
- Instances currently registered with Eureka:** A table with columns Application, AMIs, Availability Zones, and Status. It lists one instance: MICROSERVICECLOUDPROVIDERDEPT (n/a (1)) (1) UP (1) - spring-cloud-provider-8001.
- General Info:** A table showing various system metrics and configurations, such as total avail memory (380mb), environment (test), num-of-cpus (8), current memory usage (268mb (70%)), server uptime (00:03), registered replicas (<http://eureka7001.com:7001/eureka/>, <http://eureka7002.com:7002/eureka/>), and unavailable replicas (<http://eureka7001.com:7001/eureka/>, <http://eureka7002.com:7002/eureka/>).

图7: Eureka Server 集群 -7003

从上图可以看到，服务提供者 (micro-service-cloud-provider-dept-8001) 所提供的服务已经注册到了 Eureka Server 7003，并且在 DS Replicas 选项中也显示了集群中的另外两个 Eureka Server: Eureka Server 7001 和 Eureka Server 7002。

自此我们就完成了 Eureka Server 集群的搭建和使用。

Eureka 自我保护机制

当我们在本地调试基于 Eureka 的程序时，Eureka 服务注册中心很有可能会出现如下图所示的红色警告。

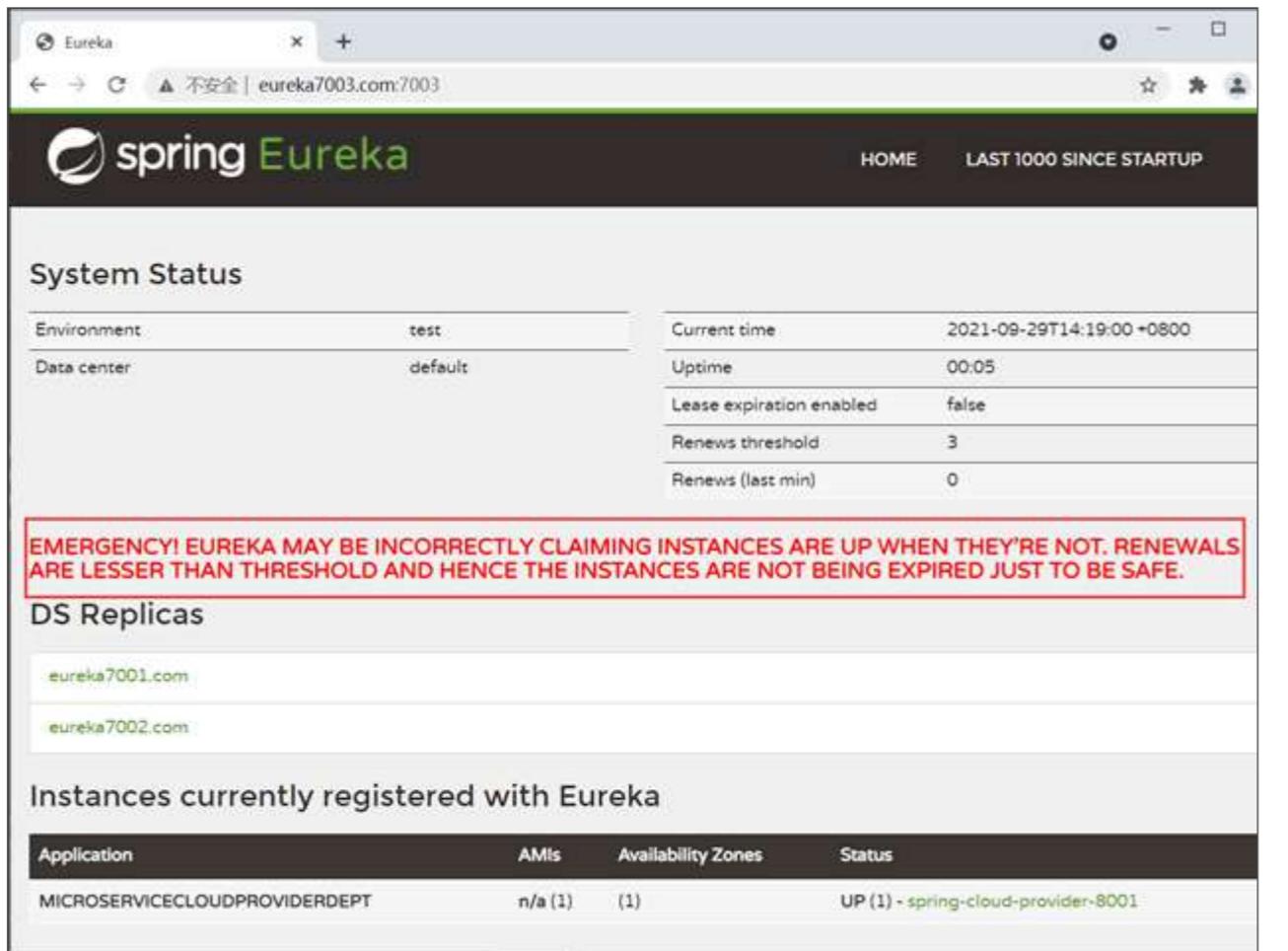


图8：Eureka 自我保护提示

实际上，这个警告是触发了 Eureka 的自我保护机制而出现的。默认情况下，如果 Eureka Server 在一段时间内（默认为 90 秒）没有接收到某个服务提供者（Eureka Client）的心跳，就会将这个服务提供者提供的服务从服务注册表中移除。这样服务消费者就再也无法从服务注册中心中获取到这个服务了，更无法调用该服务。

但在实际的分布式微服务系统中，健康的服务（Eureka Client）也有可能会由于网络故障（例如网络延迟、卡顿、拥挤等原因）而无法与 Eureka Server 正常通讯。若此时 Eureka Server 因为没有接收心跳而误将健康的服务从服务列表中移除，这显然是不合理的。而 Eureka 的自我保护机制就是来解决此问题的。

所谓“Eureka 的自我保护机制”，其中心思想就是“好死不如赖活着”。如果 Eureka Server 在一段时间内没有接收到 Eureka Client 的心跳，那么 Eureka Server 就会开启自我保护模式，将所有的 Eureka Client 的注册信息保护起来，而不是直接从服务注册表中移除。一旦网络恢复，这些 Eureka Client 提供的服务还可以继续被服务消费者消费。

综上，Eureka 的自我保护机制是一种应对网络异常的安全保护措施。它的架构哲学是：宁可同时保留所有微服务（健康的服务和不健康的服务都会保留）也不盲目移除任何健康的服务。通过 Eureka 的自我保护机制，可以让 Eureka Server 集群更加的健壮、稳定。

Eureka 的自我保护机制也存在弊端。如果在 Eureka 自我保护机制触发期间，服务提供者提供的服务出现问题，那么服务消费者就很容易获取到已经不存在的服务进而出现调用失败的情况，此时，我们可以通过客户端的容错机制来解决此问题，详情请参考 [Spring Cloud Netflix Ribbon](#) 和 [Spring Cloud Netflix Hystrix](#)。

默认情况下，Eureka 的自我保护机制是开启的，如果想要关闭，则需要在配置文件中添加以下配置。

01. eureka:
02. server:



```
03. enable-self-preservation: false # false 关闭 Eureka 的自我保护机制，默认是开启，一般不建议大家修改
```

示例 3

下面我们通过一个实例，来验证下 Eureka 的自我保护机制。

1. 在 micro-service-cloud-eureka-7001 的配置文件 application.yml 中添加以下配置，关闭 Eureka 的自我保护机制。

```
01. eureka:  
02.   server:  
03.     enable-self-preservation: false # false 关闭 Eureka 的自我保护机制，默认是开启，一般不建议大家修改
```

2. 集群中的 micro-service-cloud-eureka-7002 和 micro-service-cloud-eureka-7002 不作任何修改，即它们的自我保护机制是开启的。

3. 重启 Eureka Server 集群以及 micro-service-cloud-provider-dept-8001，使用浏览器访问 “<http://eureka7001.com:7001/>” ，结果如下图。

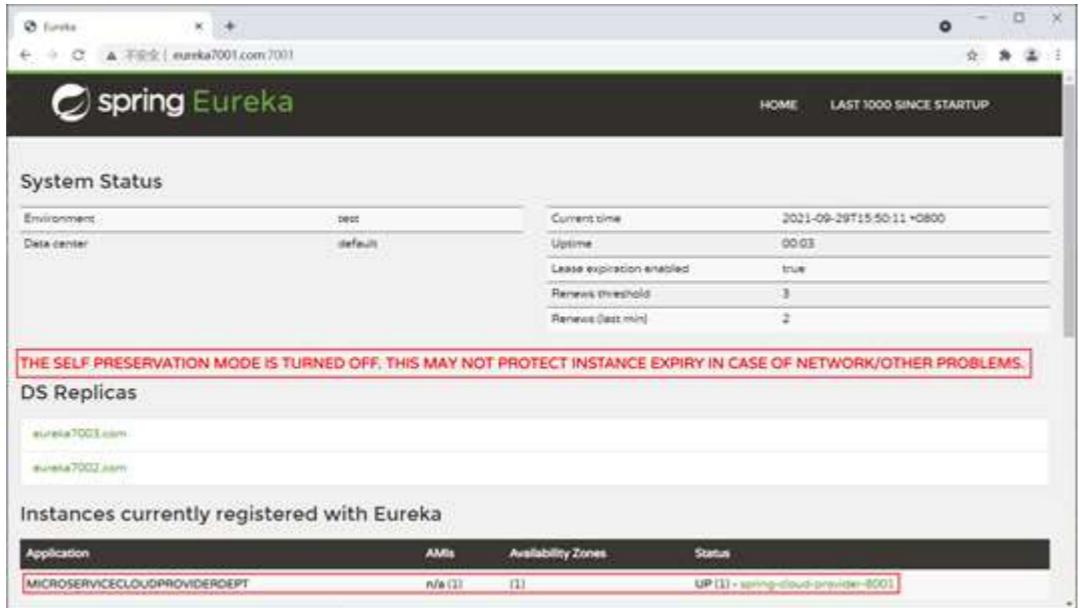


图9：Eureka 关闭自我保护机制

从图 8，您可以看到以下内容：

- 在 DS Replicas 选项上面出现了红色警告信息 “THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.”，出现该信息则表示 “Eureka 自我保护模式已关闭。”
- micro-service-cloud-provider-dept-8001 提供的服务已经注册到该 Eureka Server 中。

4. 使用浏览器访问 “<http://eureka7002.com:7002/>” ，结果如下图。

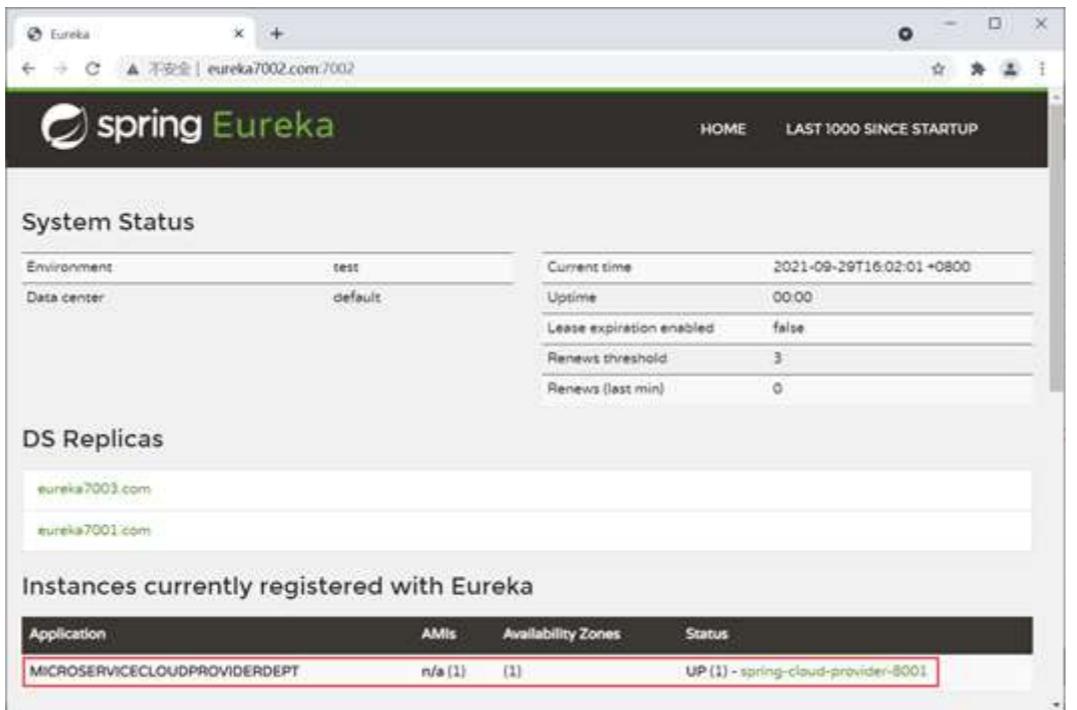


图10: Eureka 自我保护机制

从图 9 可以看出，micro-service-cloud-provider-dept-8001 提供的服务也已经注册到当前 Eureka Server 中，但 DS Replicas 选项上方没有任何警告提示。

5. 关闭 micro-service-cloud-provider-dept-8001，等待几分钟，再次访问 “<http://eureka7001.com:7001/>” ，结果如下图。

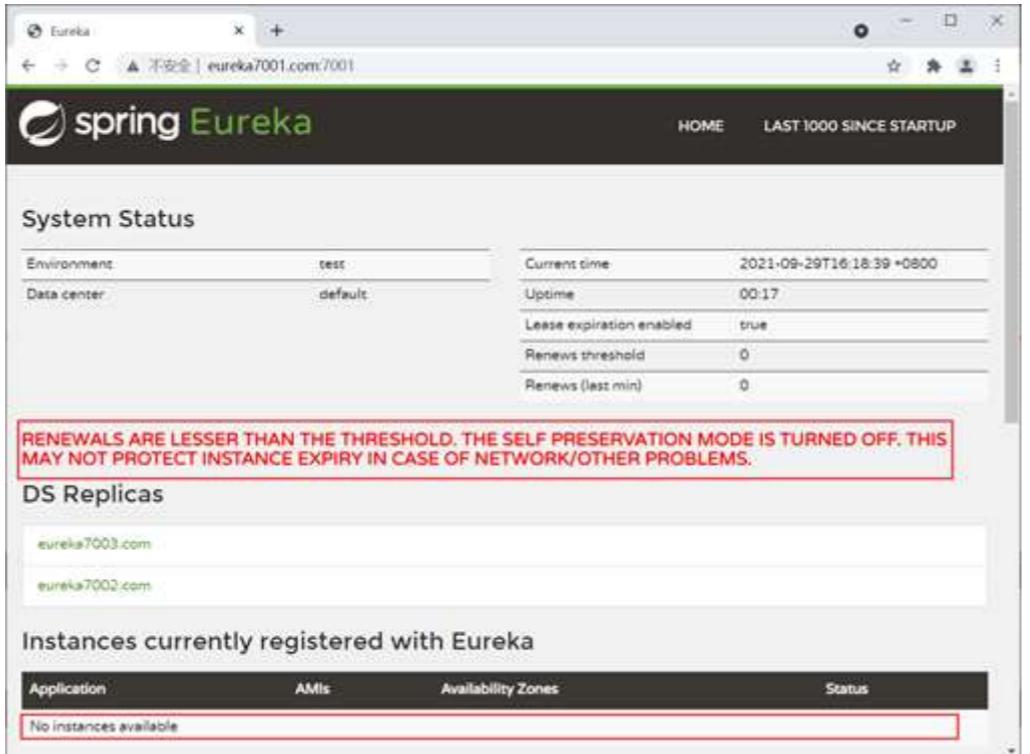


图11: Eureka 关闭自我保护机制-2

在图 10 中，我们可以看到以下内容：

- 在 DS Replicas 选项上面出现了红色警告信息 “RENEWALS ARE LESSER THAN THE THRESHOLD. THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.”，出现该信息则表示 Eureka 的自我保护模式已关闭，且已经有服务被移除。

- micro-service-cloud-provider-dept-8001 提供的服务已经从服务列表中移除。

6. 再次访问 “<http://eureka7002.com:7002/>” , 结果如下图。

The screenshot shows the Eureka Server dashboard. At the top, it displays system status information including environment (test), data center (default), current time (2021-09-29T16:34:20 +0800), uptime (00:32), lease expiration enabled (false), renew threshold (3), and renew count (0). A red warning box at the bottom states: "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." Below this, the "DS Replicas" section lists eureka7003.com and eureka7001.com. The "Instances currently registered with Eureka" section shows a table with one entry: Application: MICROSERVICECLOUDPROVIDERDEPT, AMIs: n/a (1), Availability Zones: (1), Status: UP (1) - spring-cloud-provider-8001.

图12: Eureka 自我保护机制生效

在图 11 中 , 您可以看到以下内容:

- 在 DS Replicas 选项上面出现了红色警告信息 “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.” , 出现该信息表明 Eureka 的自我保护机制处于开启状态, 且已经被触发。
- micro-service-cloud-provider-dept-8001 的服务信息依然保存 Eureka Server 服务注册表中, 并未被移除。

[< 上一节](#)

[下一节 >](#)

推荐阅读

[C++ multimap\(STL multimap\)的使用详解](#)

[C++函数的默认参数详解](#)

[MySQL修改视图 \(ALTER VIEW\)](#)

[Python while循环及用法详解](#)

[Django QueryDict对象](#)

[Python Pandas统计函数](#)

[Redis list列表](#)

[Redis SETEX命令的用法](#)

[《Web前端开发项目化教程》PDF下载 \(高清完整版\)](#)

[Linux ln命令：创建链接文件](#)

