

# 0. 学习目标

---

- 能够使用RestTemplate发送请求
- 能够说出SpringCloud的作用
- 能够搭建Eureka注册中心
- 能够使用Robbin负载均衡
- 能够使用Hystrix熔断器

## 1. 系统架构演变

---

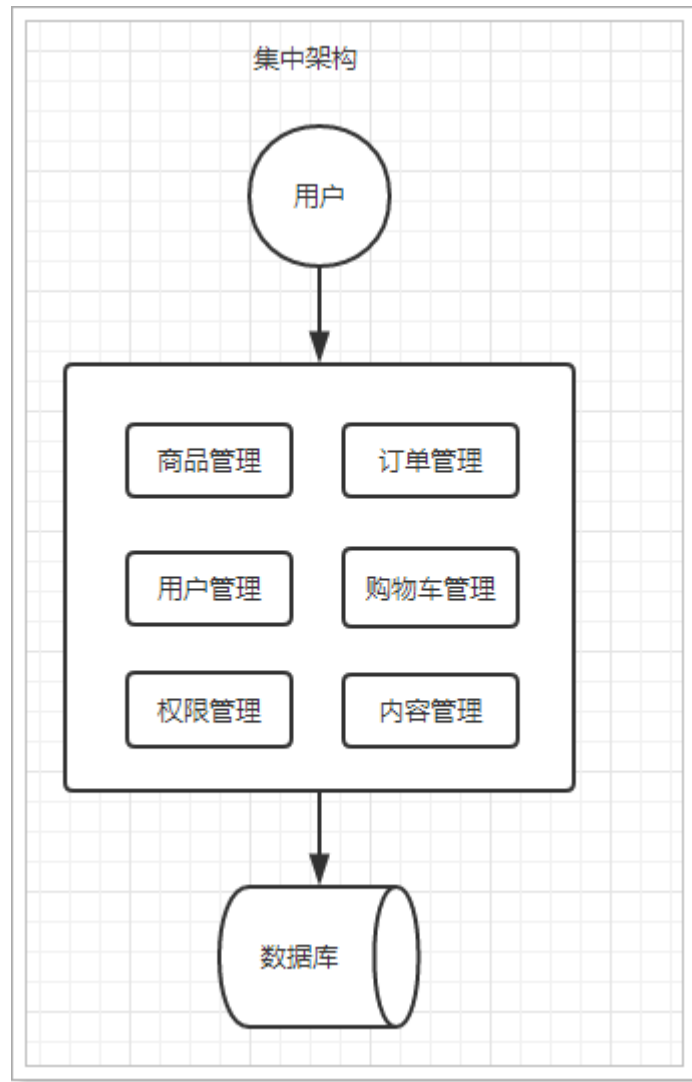
随着互联网的发展，网站应用的规模不断扩大。需求的激增，带来的是技术上的压力。系统架构也因此也不断的演进、升级、迭代。从单一应用，到垂直拆分，到分布式服务，到SOA，以及现在火热的微服务架构，还有在Google带领下来势汹涌的Service Mesh。我们到底是该乘坐微服务的船只驶向远方，还是偏安逸得过且过？

其实生活不止眼前的苟且，还有诗和远方。所以我们今天就回顾历史，看一看系统架构演变的历程；把握现在，学习现在最火的技术架构；展望未来，争取成为一名优秀的Java工程师。

### 1.1. 集中式架构

---

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。



优点：

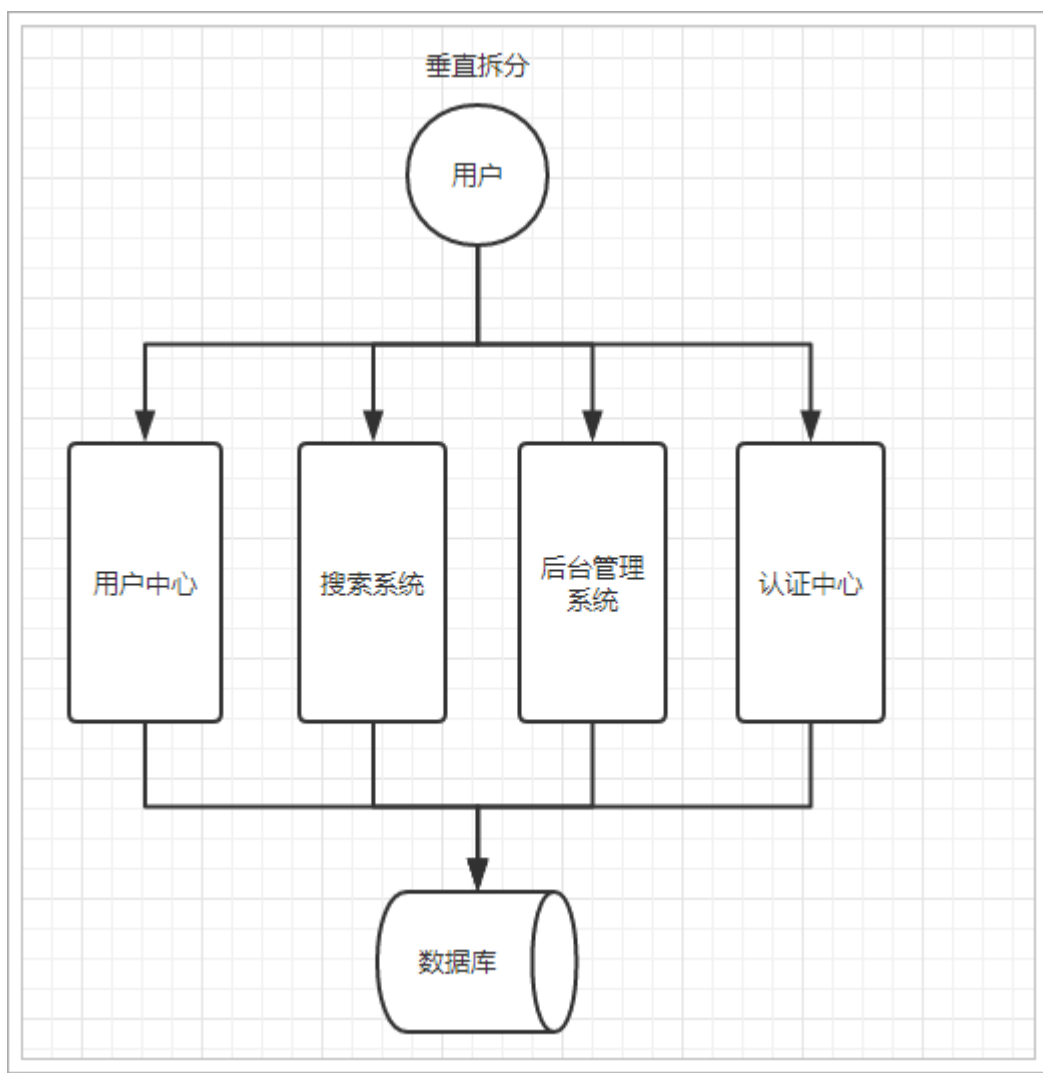
- 系统开发速度快
- 维护成本低
- 适用于并发要求较低的系统

缺点：

- 代码耦合度高，后期维护困难
- 无法针对不同模块进行针对性优化
- 无法水平扩展
- 单点容错率低，并发能力差

## 1.2. 垂直拆分

当访问量逐渐增大，单一应用无法满足需求，此时为了应对更高的并发和业务需求，我们根据业务功能对系统进行拆分：



优点：

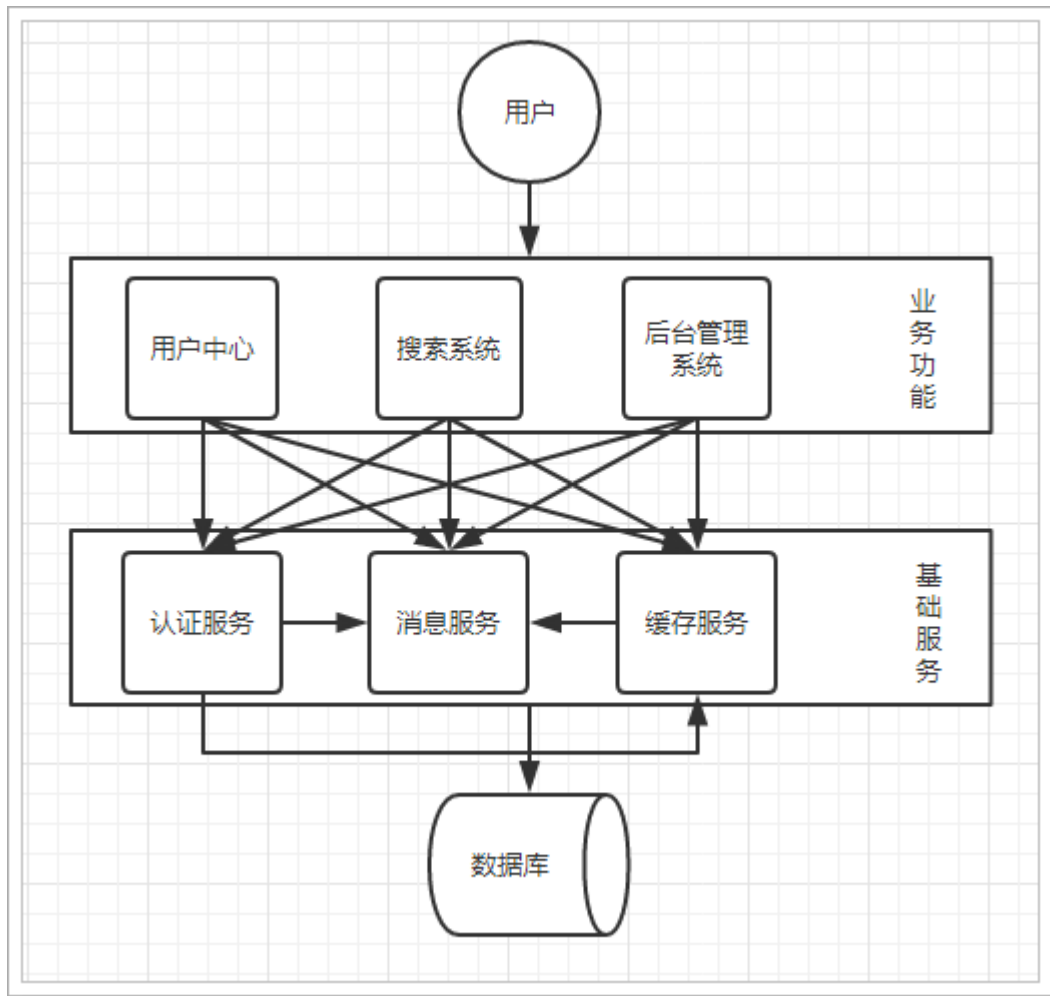
- 系统拆分实现了流量分担，解决了并发问题
- 可以针对不同模块进行优化
- 方便水平扩展，负载均衡，容错率提高

缺点：

- 系统间相互独立，会有很多重复开发工作，影响开发效率

## 1.3. 分布式服务

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。



优点:

- 将基础服务进行了抽取，系统间相互调用，提高了代码复用和开发效率

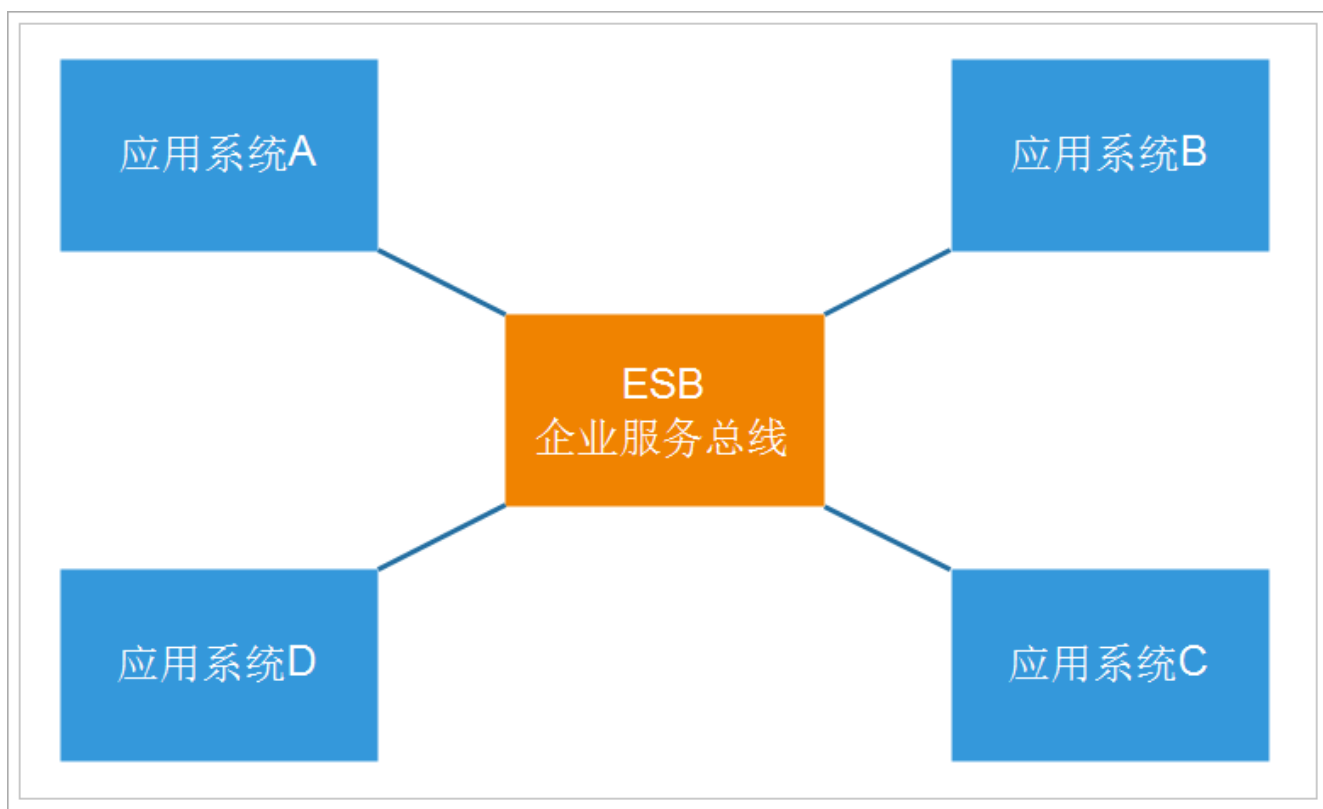
缺点:

- 系统间耦合度变高，调用关系错综复杂，难以维护

## 1.4. 面向服务架构 (SOA)

SOA (Service Oriented Architecture) 面向服务的架构：它是一种设计方法，其中包含多个服务，服务之间通过相互依赖最终提供一系列的功能。一个服务通常以独立的形式存在与操作系统进程中。各个服务之间通过网络调用。

SOA结构图：



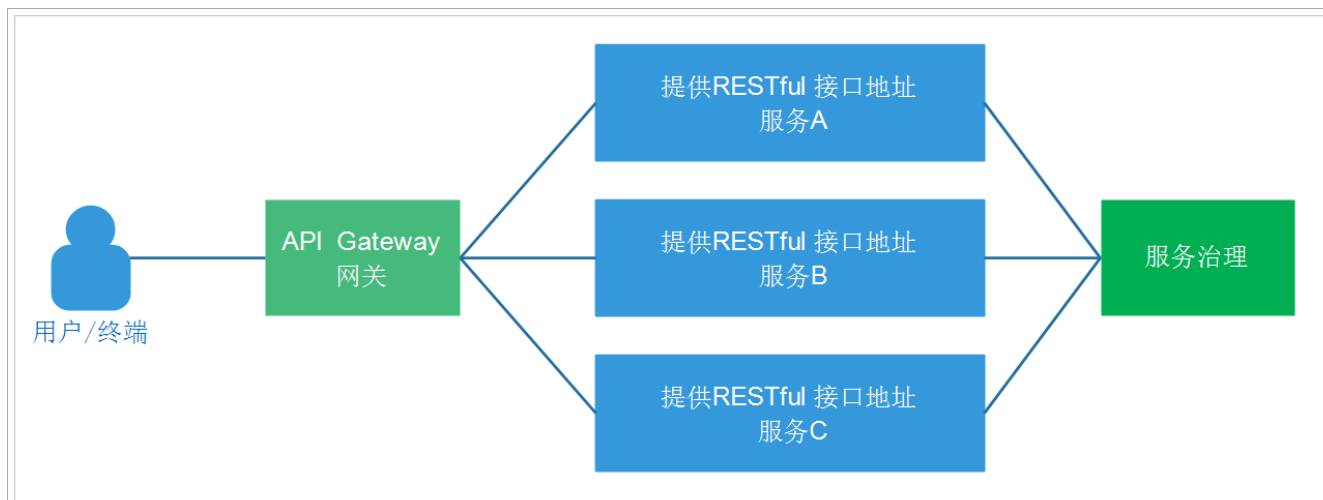
ESB（企业服务总线），简单来说 ESB 就是一根管道，用来连接各个服务节点。为了集成不同系统，不同协议的服务，ESB 做了消息的转解释和路由工作，让不同的服务互联互通。

**SOA缺点：**每个供应商提供的ESB产品有偏差，自身实现较为复杂；应用服务粒度较大，ESB集成整合所有服务和协议、数据转换使得运维、测试部署困难。所有服务都通过一个通路通信，直接降低了通信速度。

## 1.5. 微服务架构

**微服务架构**是使用一套小服务来开发单个应用的方式或途径，每个服务基于单一业务能力构建，运行在自己的进程中，并使用轻量级机制通信，通常是HTTP API，并能够通过自动化部署机制来独立部署。这些服务可以使用不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理。

微服务结构图：



API Gateway网关是一个服务器，是系统的唯一入口。为每个客户端提供一个定制的API。API网关核心是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。如它还可以具有其它职责，如身份验证、监控、负载均衡、缓存、请求分片与管理、静态响应处理。通常，网关提供RESTful/HTTP的方式访问服务。而服务端通过服务注册中心进行服务注册和管理。

微服务的特点：

- 单一职责：微服务中每一个服务都对应唯一的业务能力，做到**单一职责**
- 微：微服务的服务拆分**粒度很小**，例如一个用户管理就可以作为一个服务。每个服务虽小，但“五脏俱全”。
- 面向服务：**面向服务**是说每个服务都要对外暴露Rest风格服务接口API。并不关心服务的技术实现，做到与平台和语言无关，也不限定用什么技术实现，只要提供Rest的接口即可。
- 自治：自治是说**服务间互相独立**，互不干扰
  - 团队独立：每个服务都是一个独立的开发团队，人数不能过多。
  - 技术独立：因为是面向服务，提供Rest接口，使用什么技术没有别人干涉
  - 前后端分离：采用前后端分离开发，提供统一Rest接口，后端不用再为PC、移动端开发不同接口
  - 数据库分离：每个服务都使用自己的数据源
  - 部署独立，服务间虽然有调用，但要做到服务重启不影响其它服务。有利于持续集成和持续交付。每个服务都是独立的组件，可复用，可替换，降低耦合，易维护

微服务架构与SOA都是对系统进行拆分；微服务架构基于SOA思想，可以把微服务当做去除了ESB的SOA。ESB是SOA架构中的中心总线，设计图形应该是星形的，而微服务是去中心化的分布式软件架构。两者比较类似，但其实也有一些差别：

功能	SOA	微服务
组件大小	大块业务逻辑	单独任务或小块业务逻辑
耦合	通常松耦合	总是松耦合
管理	着重中央管理	着重分散管理
目标	确保应用能够交互操作	易维护、易扩展、更轻量级的交互

## 2. 服务调用方式

### 2.1. RPC和HTTP

无论是微服务还是SOA，都面临着服务间的远程调用。那么服务间的远程调用方式有哪些呢？

常见的远程调用方式有以下2种：

- **RPC**：Remote Produce Call远程过程调用，**RPC基于Socket，工作在会话层。自定义数据格式**，速度快，效率高。早期的webservice，现在热门的dubbo，都是RPC的典型代表
- **Http**：http其实是一种**网络传输协议，基于TCP，工作在应用层，规定了数据传输的格式**。现在客户端浏览器与服务端通信基本都是采用Http协议，也可以用来进行远程服务调用。缺点是消息封装臃肿，优势是对服务的提供和调用方没有任何技术限定，自由灵活，更符合微服务理念。

现在热门的Rest风格，就可以通过http协议来实现。

**区别：**RPC的机制是根据语言的API（language API）来定义的，而不是根据基于网络的应用来定义的。

如果你们公司全部采用Java技术栈，那么使用Dubbo作为微服务架构是一个不错的选择。

相反，如果公司的技术栈多样化，而且你更青睐Spring家族，那么Spring Cloud搭建微服务是不二之选。在我们的项目中，会选择Spring Cloud套件，因此会使用Http方式来实现服务间调用。

## 2.2. Http客户端工具

既然微服务选择了Http，那么我们就需要考虑自己来实现对请求和响应的处理。不过开源世界已经有很多的http客户端工具，能够帮助我们做这些事情，例如：

- HttpClient
- OKHttp
- URLConnection

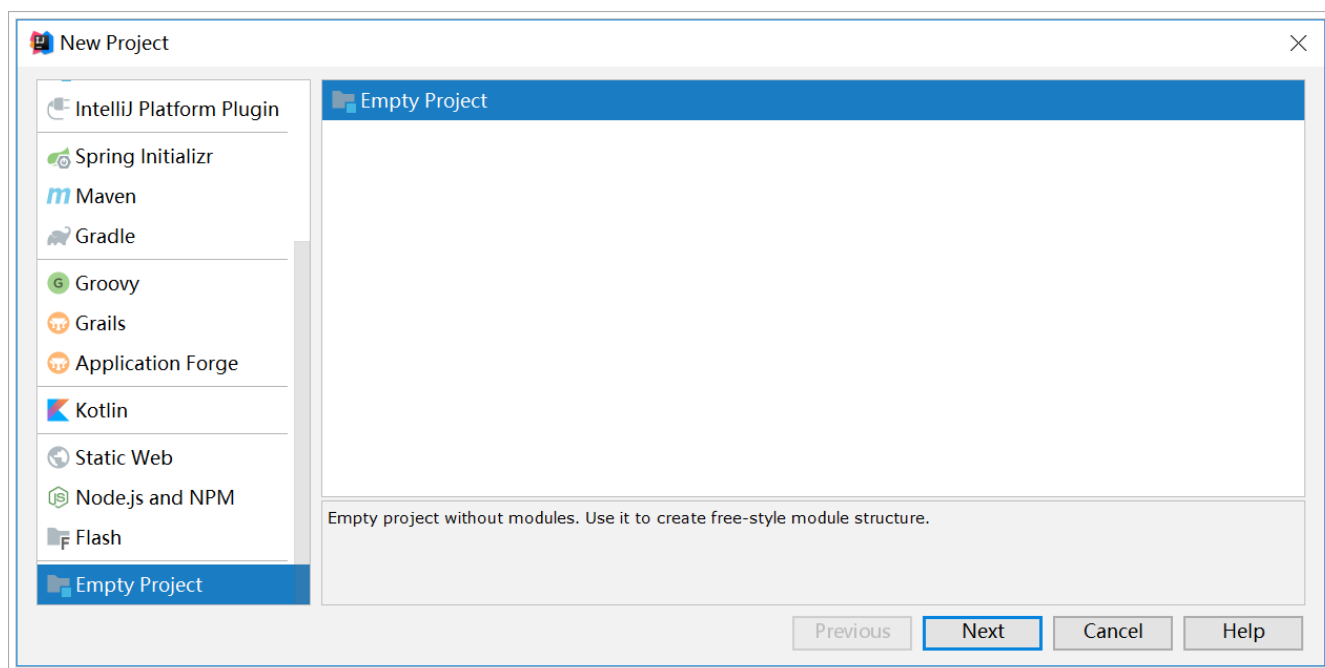
不过这些不同的客户端，API各不相同。而Spring也有对http的客户端进行封装，提供了工具类叫RestTemplate。

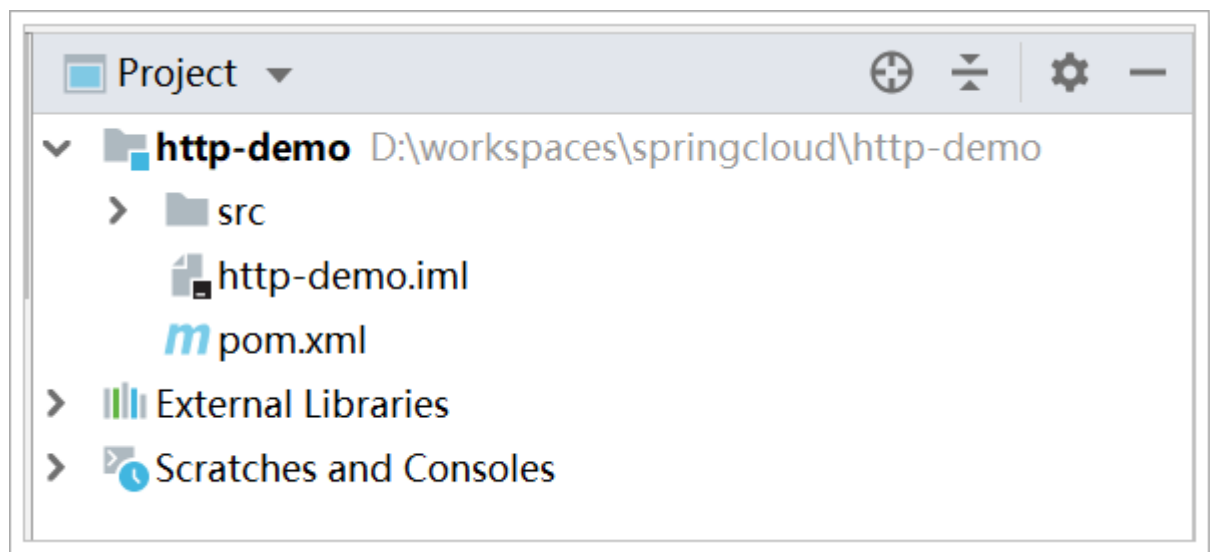
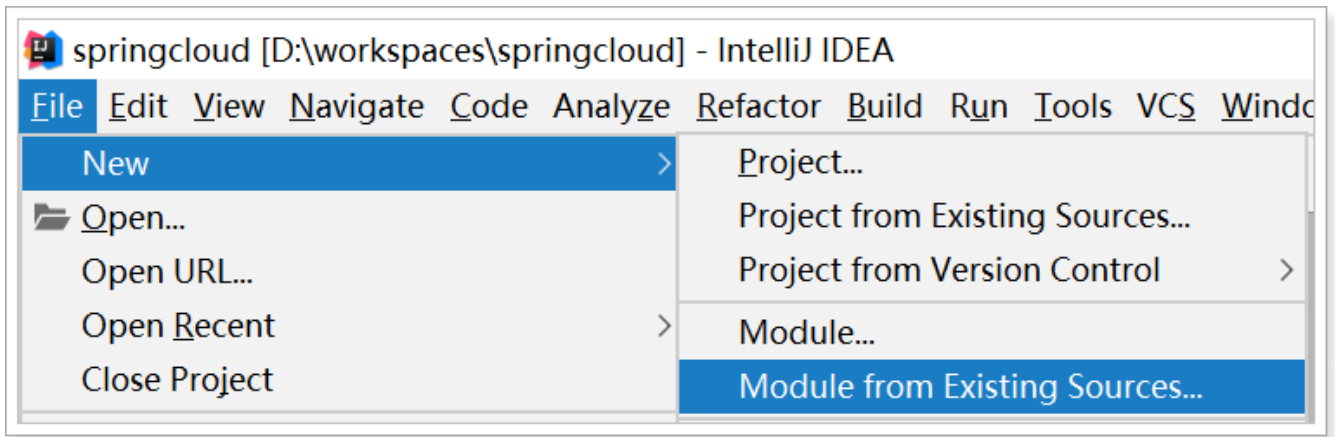
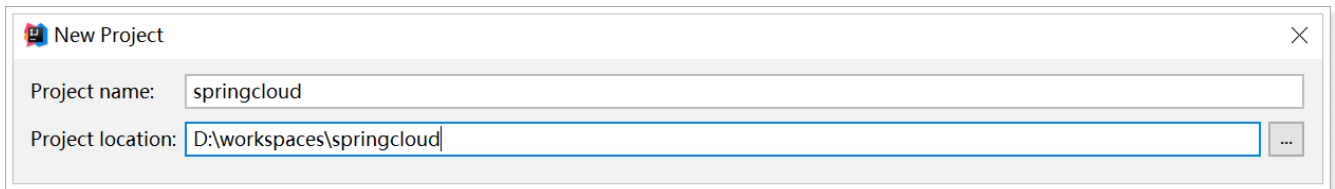
## 2.3. Spring的RestTemplate

Spring提供了一个RestTemplate模板工具类，对基于Http的客户端进行了封装，并且实现了对象与json的序列化和反序列化，非常方便。RestTemplate并没有限定Http的客户端类型，而是进行了抽象，目前常用的3种都有支持：

- HttpClient
- OkHttp
- JDK原生的URLConnection（默认的）

导入资料\http-demo 工程；





已经在导入的项目中的 `HttpDemoApplication` 注册一个 `RestTemplate` 对象，可以在启动类位置注册：

```
1 package com.itheima;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.web.client.RestTemplate;
7
8 @SpringBootApplication
9 public class HttpDemoApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(HttpDemoApplication.class, args);
12     }
13
14     @Bean
15     public RestTemplate restTemplate(){
```



```

16         return new RestTemplate();
17     }
18 }

```

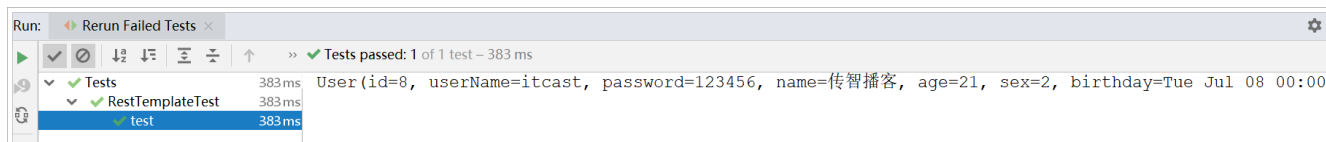
启动springboot项目，在项目中的测试类中直接 @Autowired 注入：

```

1  package com.itheima.test;
2
3  import com.itheima.pojo.User;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.boot.test.context.SpringBootTest;
8  import org.springframework.test.context.junit4.SpringRunner;
9  import org.springframework.web.client.RestTemplate;
10
11 @RunWith(SpringRunner.class)
12 @SpringBootTest
13 public class RestTemplateTest {
14
15     @Autowired
16     private RestTemplate restTemplate;
17
18     @Test
19     public void test(){
20         //如果要测试需要启动spring boot项目，以便获取数据
21         String url = "http://localhost/user/8";
22         User user = restTemplate.getForObject(url, User.class);
23         System.out.println(user);
24     }
25 }
26

```

- 通过RestTemplate的getForObject()方法，传递url地址及实体类的字节码，RestTemplate会自动发起请求，接收响应，并且帮我们对响应结果进行反序列化。



了解完Http客户端工具，接下来就可以正式学习微服务了。

## 4. 初识Spring Cloud

微服务是一种架构方式，最终肯定需要技术架构去实施。

微服务的实现方式很多，但是最火的莫过于Spring Cloud了。为什么？

- 后台硬：作为Spring家族的一员，有整个Spring全家桶靠山，背景十分强大。
- 技术强：Spring作为Java领域的前辈，可以说是功力深厚。有强力的技术团队支撑，一般人还真比不了
- 群众基础好：可以说大多数程序员的成长都伴随着Spring框架，试问：现在有几家公司开发不用Spring？Spring Cloud与Spring的各个框架无缝整合，对大家来说一切都是熟悉的配方，熟悉的味道。
- 使用方便：相信大家都体会到了SpringBoot给我们开发带来的便利，而Spring Cloud完全支持Spring Boot的开发，用很少的配置就能完成微服务框架的搭建

## 4.1. 简介

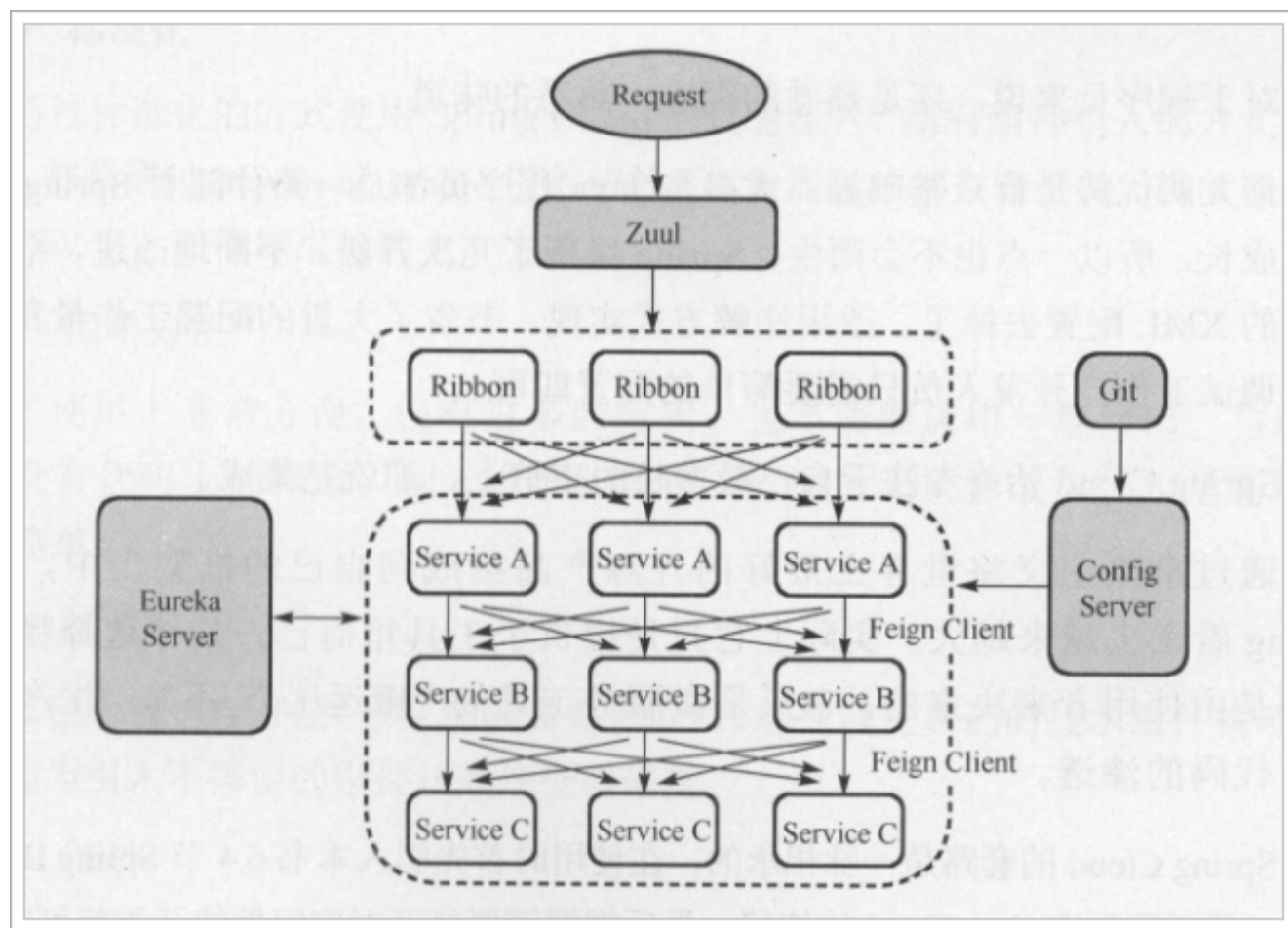
Spring Cloud是Spring旗下的项目之一，[官网地址：http://projects.spring.io/spring-cloud/](http://projects.spring.io/spring-cloud/)

Spring最擅长的就是集成，把世界上最好的框架拿过来，集成到自己的项目中。

Spring Cloud也是一样，它将现在非常流行的一些技术整合到一起，实现了诸如：配置管理，服务发现，智能路由，负载均衡，熔断器，控制总线，集群状态等功能；协调分布式环境中各个系统，为各类服务提供模板性配置。其主要涉及的组件包括：

- Eureka：注册中心
- Zuul、Gateway：服务网关
- Ribbon：负载均衡
- Feign：服务调用
- Hystrix或Resilience4j：熔断器

以上只是其中一部分，架构图：



## 4.2. 版本

Spring Cloud不是一个组件，而是许多组件的集合；它的版本命名比较特殊，是以A到Z的为首字母的一些单词（其实是伦敦地铁站的名字）组成：

Hoxton	SNAPSHOT
Greenwich SR1	GA
Greenwich	SNAPSHOT
Finchley SR3	GA
Finchley	SNAPSHOT
Edgware SR5	GA
Edgware	SNAPSHOT
Dalston SR5	GA

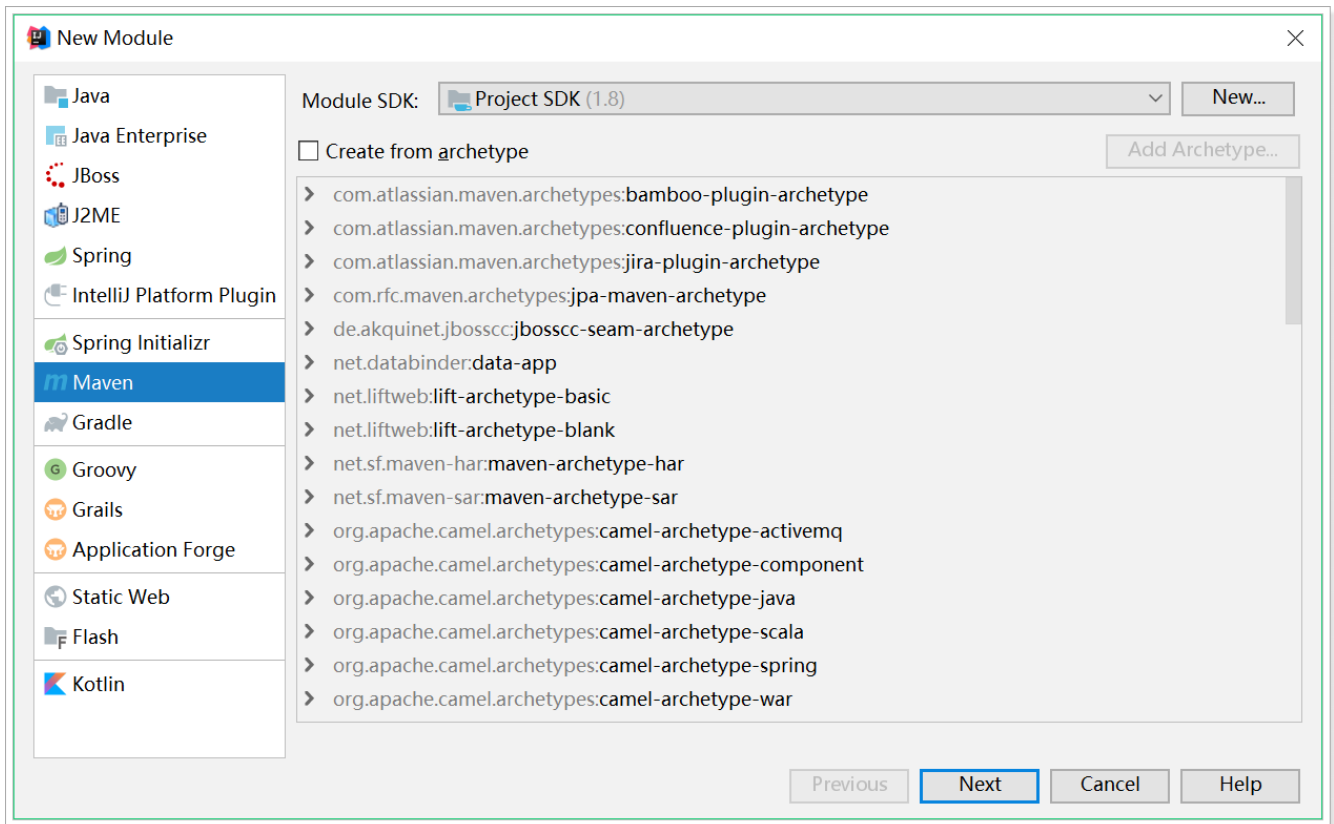
我们在项目中，使用最新稳定的Greenwich版本。

## 5. 微服务场景模拟

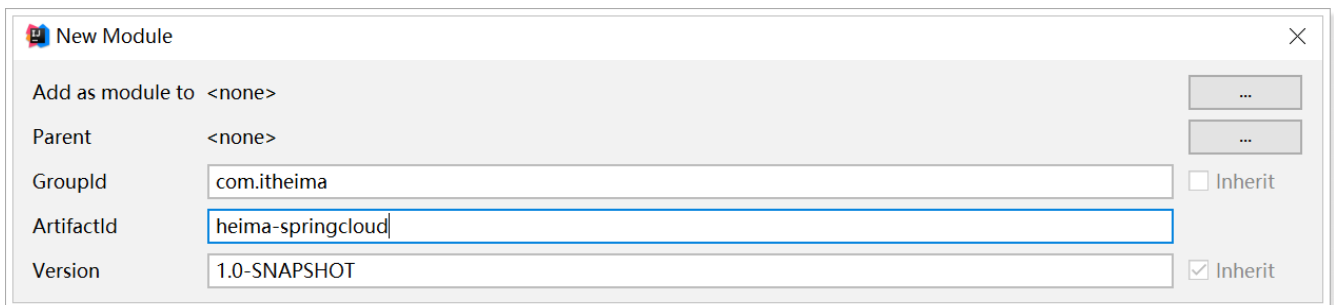
首先，我们需要模拟一个服务调用的场景。方便后面学习微服务架构

### 5.1. 创建父工程

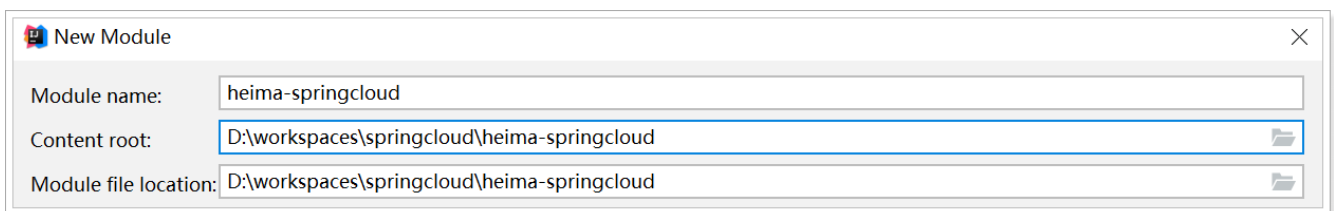
微服务中需要同时创建多个项目，为了方便课堂演示，先创建一个父工程，然后后续的工程都以这个工程为父，实现maven的聚合。这样可以在一个窗口看到所有工程，方便讲解。**在实际开发中，每个微服务可独立一个工程。**



编写项目信息：



编写保存位置：



然后将 pom.xml 修改成如下（请从 资料\heima-springcloud.xml 文件中复制）：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5       http://maven.apache.org/xsd/maven-4.0.0.xsd">
6       <modelVersion>4.0.0</modelVersion>

```

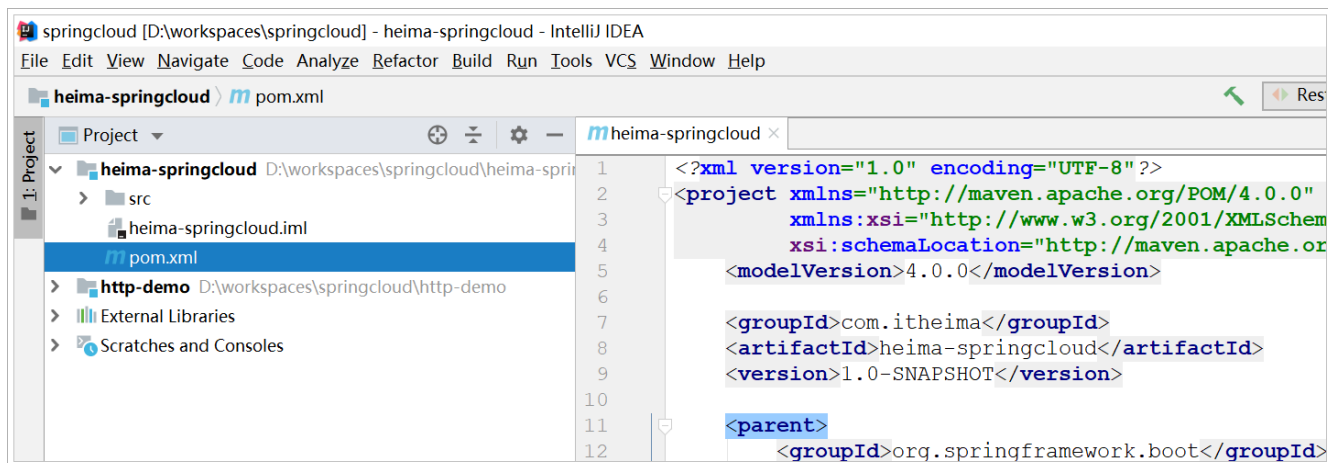
```
7 <groupId>com.itheima</groupId>
8 <artifactId>heima-springcloud</artifactId>
9 <version>1.0-SNAPSHOT</version>
10
11
12 <parent>
13     <groupId>org.springframework.boot</groupId>
14     <artifactId>spring-boot-starter-parent</artifactId>
15     <version>2.1.5.RELEASE</version>
16     <relativePath/>
17 </parent>
18
19 <properties>
20     <java.version>1.8</java.version>
21     <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
22     <mapper.starter.version>2.1.5</mapper.starter.version>
23     <mysql.version>5.1.46</mysql.version>
24 </properties>
25
26 <dependencyManagement>
27     <dependencies>
28         <!-- springCloud -->
29         <dependency>
30             <groupId>org.springframework.cloud</groupId>
31             <artifactId>spring-cloud-dependencies</artifactId>
32             <version>${spring-cloud.version}</version>
33             <type>pom</type>
34             <scope>import</scope>
35         </dependency>
36         <!-- 通用Mapper启动器 -->
37         <dependency>
38             <groupId>tk.mybatis</groupId>
39             <artifactId>mapper-spring-boot-starter</artifactId>
40             <version>${mapper.starter.version}</version>
41         </dependency>
42         <!-- mysql驱动 -->
43         <dependency>
44             <groupId>mysql</groupId>
45             <artifactId>mysql-connector-java</artifactId>
46             <version>${mysql.version}</version>
47         </dependency>
48     </dependencies>
49 </dependencyManagement>
50 <dependencies>
51     <dependency>
52         <groupId>org.projectlombok</groupId>
53         <artifactId>lombok</artifactId>
54     </dependency>
55 </dependencies>
56
57 <build>
58     <plugins>
59         <plugin>
```

```

60         <groupId>org.springframework.boot</groupId>
61         <artifactId>spring-boot-maven-plugin</artifactId>
62     </plugin>
63 </plugins>
64 </build>
65 </project>

```

这里已经对大部分要用到的依赖的版本进行了管理，方便后续使用

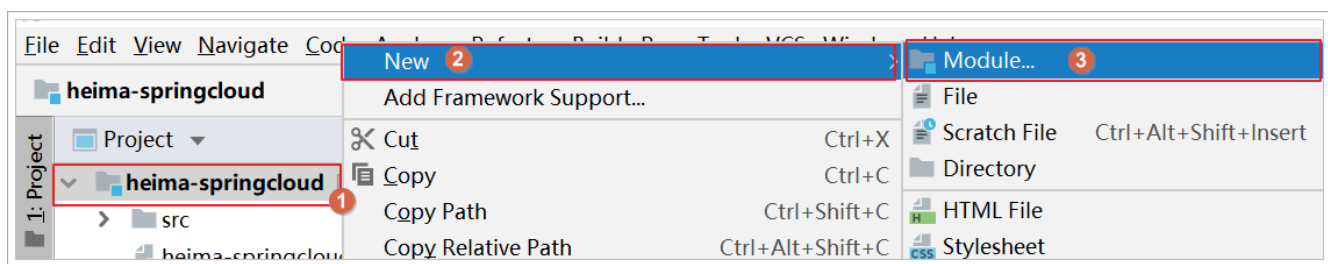


## 5.2. 服务提供者

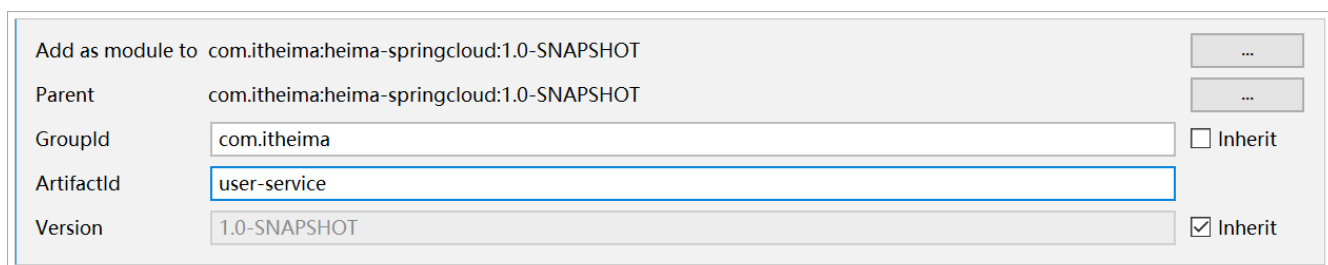
新建一个项目user-service，对外提供查询用户的服务。

### 5.2.1. 创建module



选中父工程：heima-springcloud



填写module信息：



注意，子模块要在父工程的下级目录：

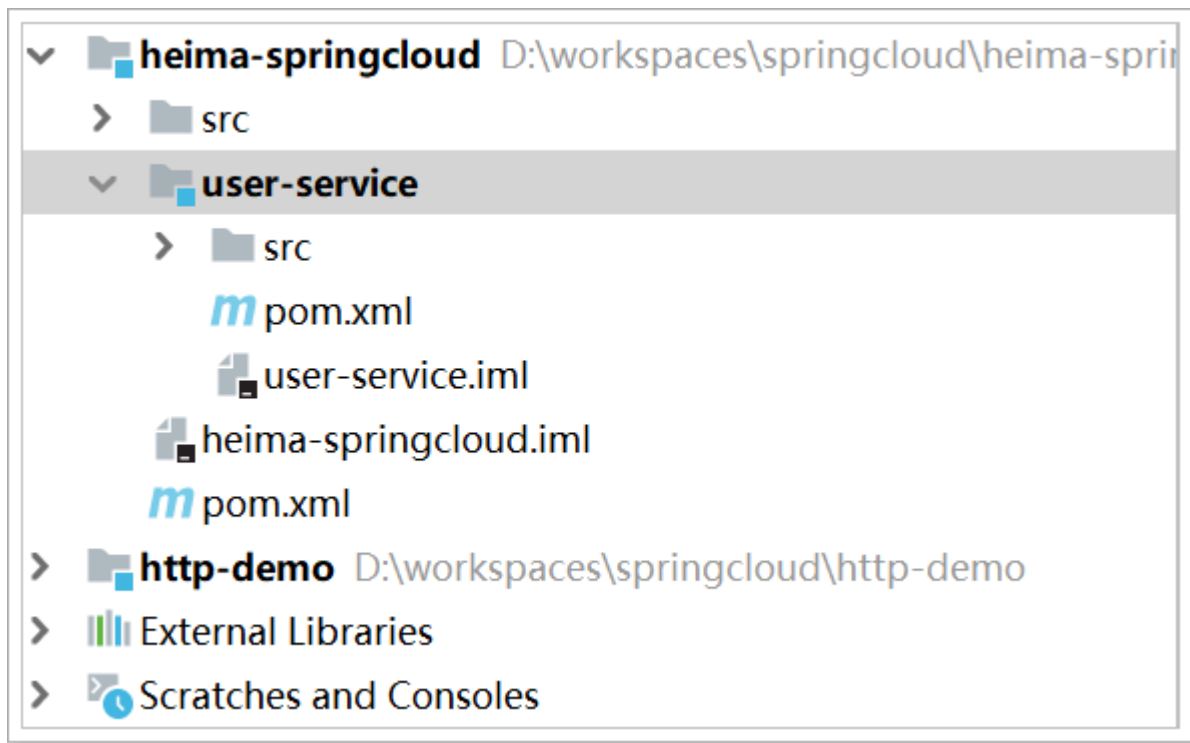
Module name:	<input type="text" value="user-service"/>
Content root:	<input type="text" value="D:\workspaces\springcloud\heima-springcloud\user-service"/> 
Module file location:	<input type="text" value="D:\workspaces\springcloud\heima-springcloud\user-service"/> 

## 5.2.2. 添加依赖

pom.xml 文件中的内容如下：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5      <parent>
6          <artifactId>heima-springcloud</artifactId>
7          <groupId>com.itheima</groupId>
8          <version>1.0-SNAPSHOT</version>
9      </parent>
10     <modelVersion>4.0.0</modelVersion>
11
12     <groupId>com.itheima</groupId>
13     <artifactId>user-service</artifactId>
14
15     <dependencies>
16         <dependency>
17             <groupId>org.springframework.boot</groupId>
18             <artifactId>spring-boot-starter-web</artifactId>
19         </dependency>
20         <dependency>
21             <groupId>mysql</groupId>
22             <artifactId>mysql-connector-java</artifactId>
23         </dependency>
24         <dependency>
25             <groupId>tk.mybatis</groupId>
26             <artifactId>mapper-spring-boot-starter</artifactId>
27         </dependency>
28     </dependencies>
29 </project>
```

项目结构：



### 5.1.3. 编写配置文件

创建 `user-service\src\main\resources\application.yml` 属性文件,这里我们采用了yaml语法,而不是properties:

```
1 server:
2   port: 9091
3 spring:
4   datasource:
5     driver-class-name: com.mysql.jdbc.Driver
6     url: jdbc:mysql://localhost:3306/springcloud
7     username: root
8     password: root
9 mybatis:
10  type-aliases-package: com.itheima.user.pojo
```

使用mysql图形界面工具创建 `springcloud` 数据库, 将 `资料\tb_user.sql` 导入;

### 5.1.4. 编写代码

- 编写 `user-service\src\main\java\com\itheima\user\UserApplication.java` 启动类:

```
1 package com.itheima.user;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import tk.mybatis.spring.annotation.MapperScan;
6
7 @SpringBootApplication
```



```

8  @MapperScan("com.itheima.user.mapper")
9  public class UserApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(UserApplication.class, args);
12     }
13 }
14

```

- 编写 `user-service\src\main\java\com\itheima\user\pojo\User.java` 实体类:

```

1  package com.itheima.user.pojo;
2
3  import lombok.Data;
4  import tk.mybatis.mapper.annotation.KeySql;
5
6  import javax.persistence.Id;
7  import javax.persistence.Table;
8  import java.util.Date;
9
10 @Table(name = "tb_user")
11 @Data
12 public class User {
13
14     @Id
15     @KeySql(useGeneratedKeys = true)
16     private Long id;
17
18     private String userName; // 用户名
19
20     private String password; // 密码
21
22     private String name; // 姓名
23
24     private Integer age; // 年龄
25
26     private Integer sex; // 性别, 1男性, 2女性
27
28     private Date birthday; // 出生日期
29
30     private Date created; // 创建时间
31
32     private Date updated; // 更新时间
33
34     private String note; // 备注
35 }

```

- 编写 `user-service\src\main\java\com\itheima\user\mapper\UserMapper.java`

```

1 package com.itheima.user.mapper;
2
3 import com.itheima.user.pojo.User;
4 import tk.mybatis.mapper.common.Mapper;
5
6 public interface UserMapper extends Mapper<User> {
7 }
8

```

- 编写 `user-service\src\main\java\com\itheima\user\service\UserService.java`

```

1 package com.itheima.user.service;
2
3 import com.itheima.user.mapper.UserMapper;
4 import com.itheima.user.pojo.User;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.stereotype.Service;
7
8 @Service
9 public class UserService {
10
11     @Autowired
12     private UserMapper userMapper;
13
14     public User queryById(Long id){
15         return userMapper.selectByPrimaryKey(id);
16     }
17 }
18

```

- 添加一个对外查询的接口处理器

`user-service\src\main\java\com\itheima\user\controller\UserController.java`

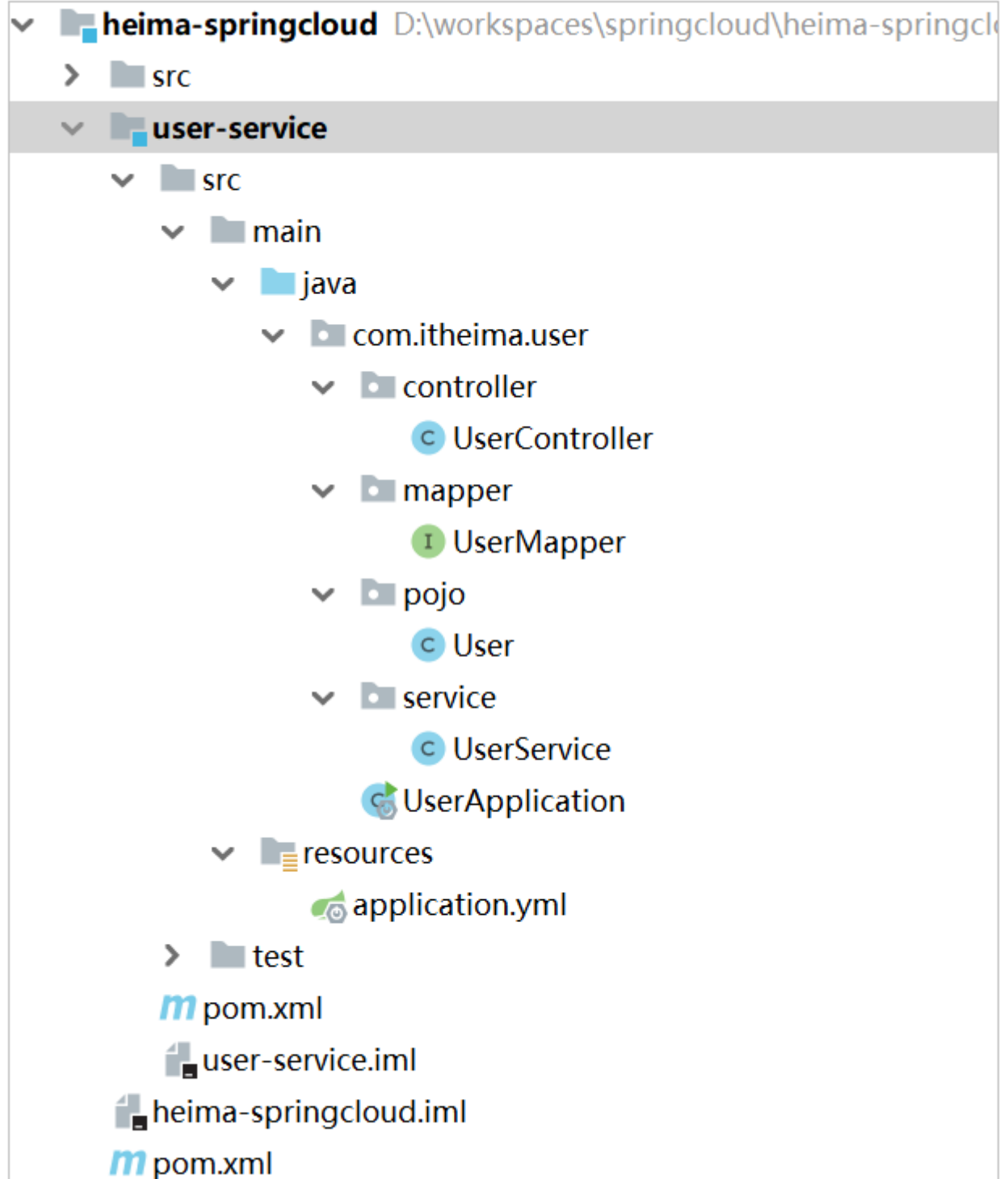
```

1 package com.itheima.user.controller;
2
3 import com.itheima.user.pojo.User;
4 import com.itheima.user.service.UserService;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.web.bind.annotation.GetMapping;
7 import org.springframework.web.bind.annotation.PathVariable;
8 import org.springframework.web.bind.annotation.RequestMapping;
9 import org.springframework.web.bind.annotation.RestController;
10
11 @RestController
12 @RequestMapping("/user")
13 public class UserController {
14
15     @Autowired
16     private UserService userService;
17
18 }
19

```

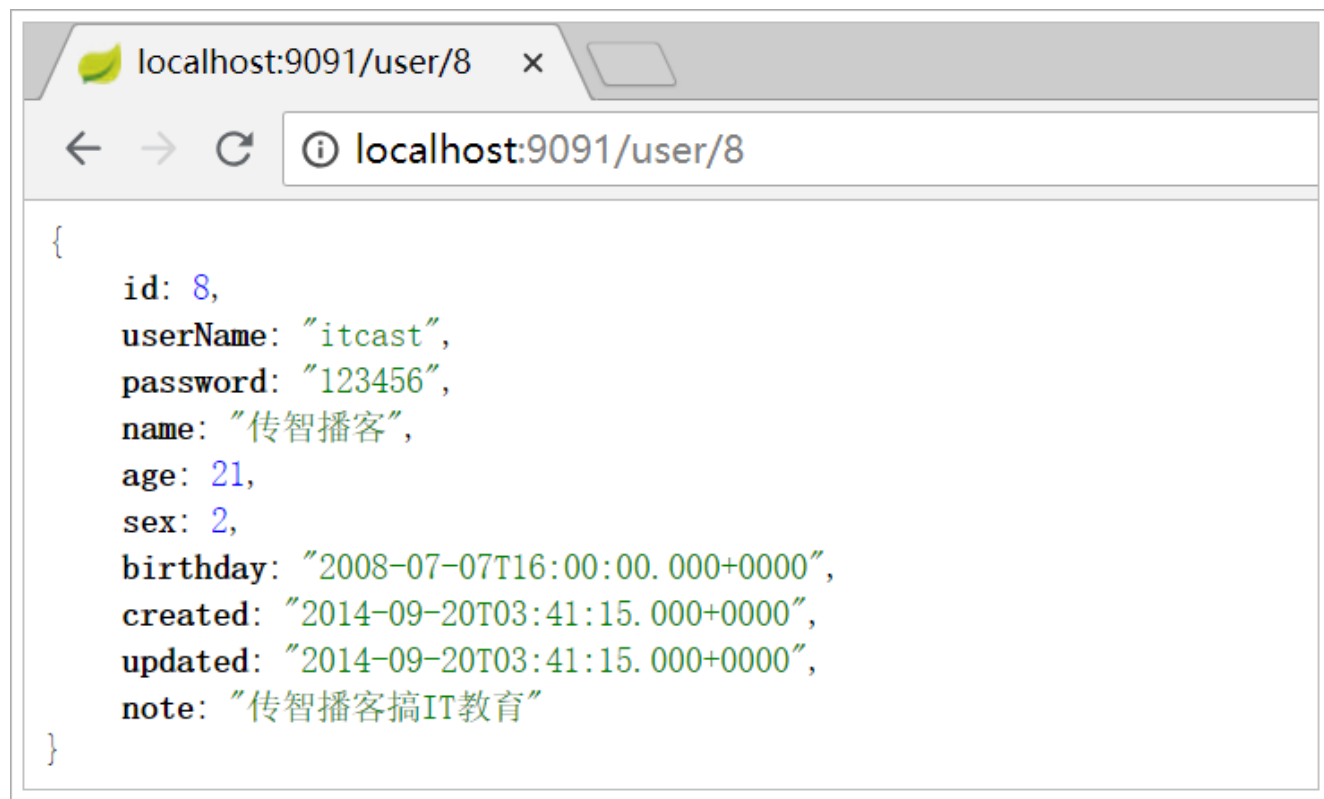
```
17
18     @GetMapping("/{id}")
19     public User queryById(@PathVariable Long id){
20         return userService.queryById(id);
21     }
22 }
23
```

创建完上述代码后项目结构：



## 5.1.5. 启动并测试

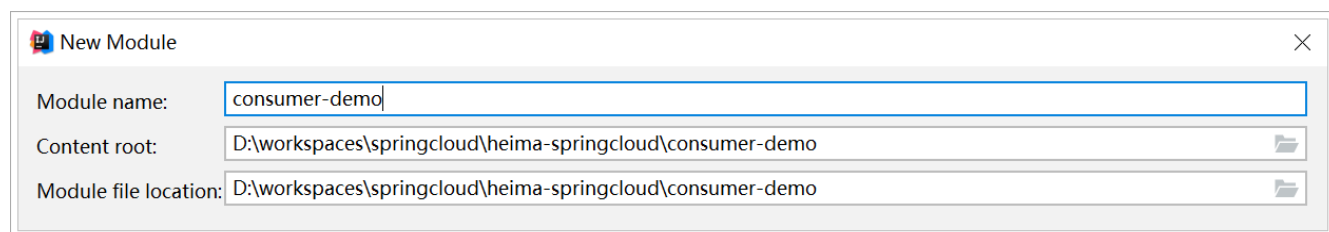
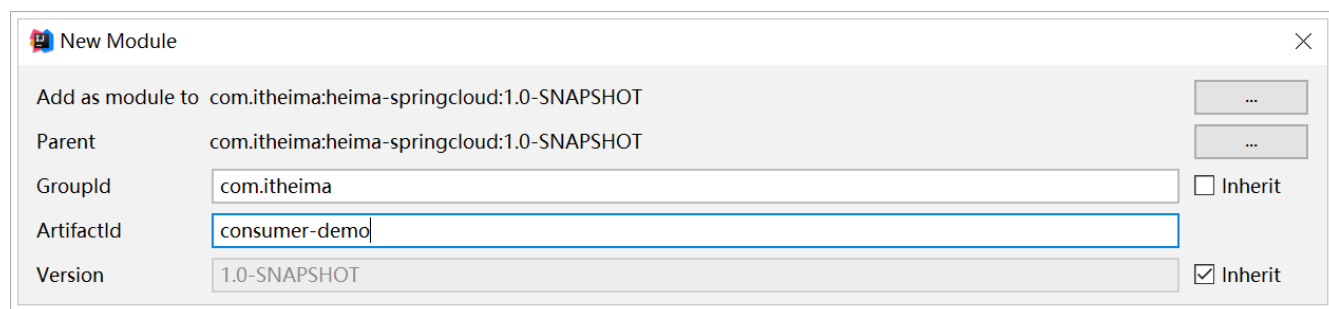
启动 `user-service` 项目，访问接口：<http://localhost:9091/user/8>



## 5.2. 服务调用者

### 5.2.1. 创建工程

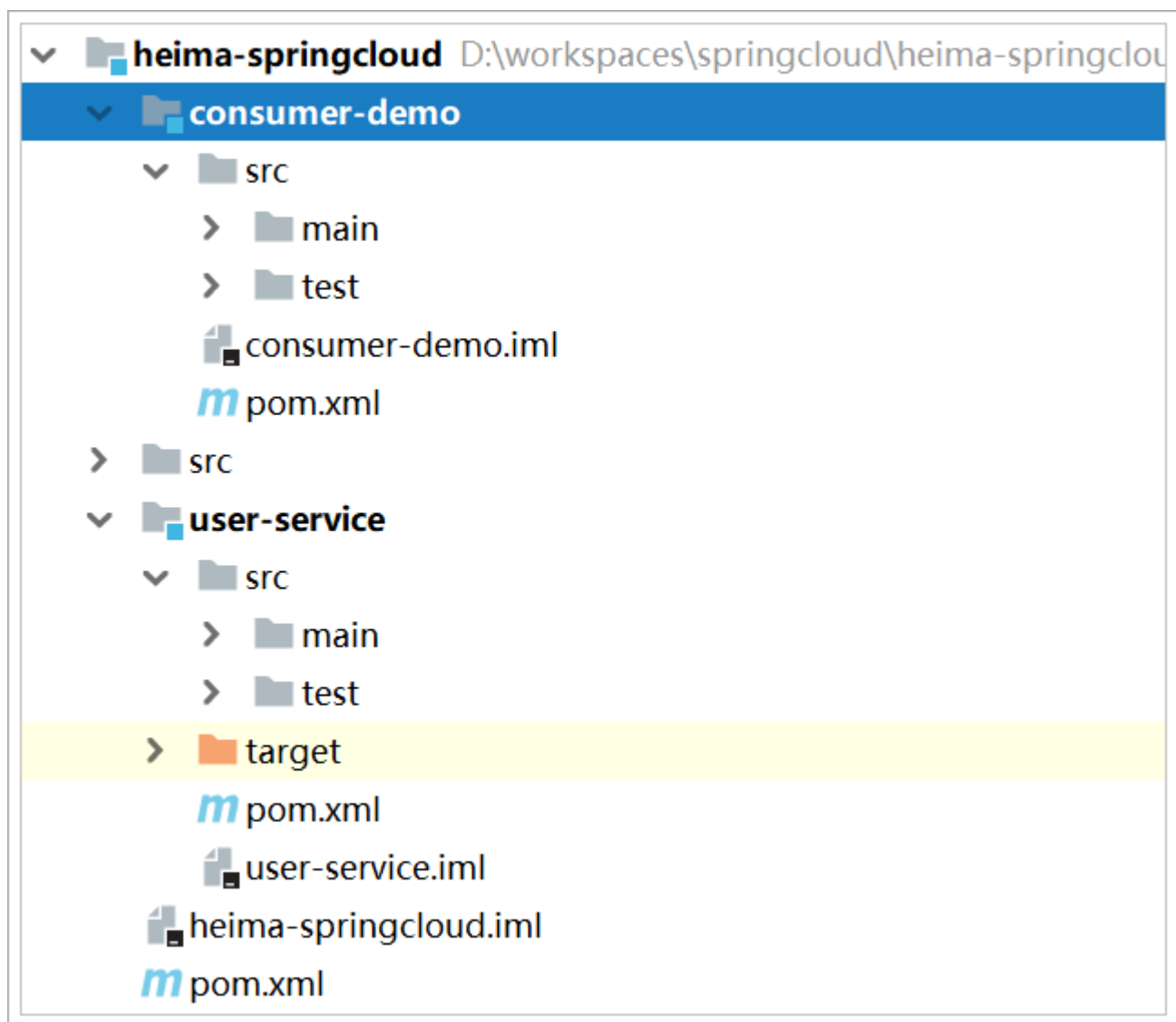
与上面类似，这里不再赘述，需要注意的是，我们调用 `user-service` 的功能，因此不需要 `mybatis` 相关依赖了。



`pom.xml` 文件内容如下：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <parent>
6         <artifactId>heima-springcloud</artifactId>
7         <groupId>com.itheima</groupId>
8         <version>1.0-SNAPSHOT</version>
9     </parent>
10    <modelVersion>4.0.0</modelVersion>
11
12    <groupId>com.itheima</groupId>
13    <artifactId>consumer-demo</artifactId>
14
15    <dependencies>
16        <dependency>
17            <groupId>org.springframework.boot</groupId>
18            <artifactId>spring-boot-starter-web</artifactId>
19        </dependency>
20    </dependencies>
21 </project>
```

项目结构如下：



### 5.2.2. 编写代码

编写启动类 `consumer-demo\src\main\java\com\itheima\consumer\ConsumerApplication.java` 并在其中注册 `RestTemplate` 具体如下：

```
1 package com.itheima.consumer;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.web.client.RestTemplate;
7
8 @SpringBootApplication
9 public class ConsumerApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(ConsumerApplication.class, args);
12     }
13
14     @Bean
15     public RestTemplate restTemplate(){
```

```
16         return new RestTemplate();
17     }
18 }
19
```

创建实体类 `consumer-demo\src\main\java\com\itheima\consumer\pojo\User.java`

```
1 package com.itheima.consumer.pojo;
2
3 import lombok.Data;
4
5 import java.util.Date;
6
7 @Data
8 public class User {
9
10     private Long id;
11
12     private String userName; // 用户名
13
14     private String password; // 密码
15
16     private String name; // 姓名
17
18     private Integer age; // 年龄
19
20     private Integer sex; // 性别, 1男性, 2女性
21
22     private Date birthday; // 出生日期
23
24     private Date created; // 创建时间
25
26     private Date updated; // 更新时间
27
28     private String note; // 备注
29
30 }
```

编写 `consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java`,

在controller中直接调用RestTemplate, 远程访问 user-service 的服务接口:

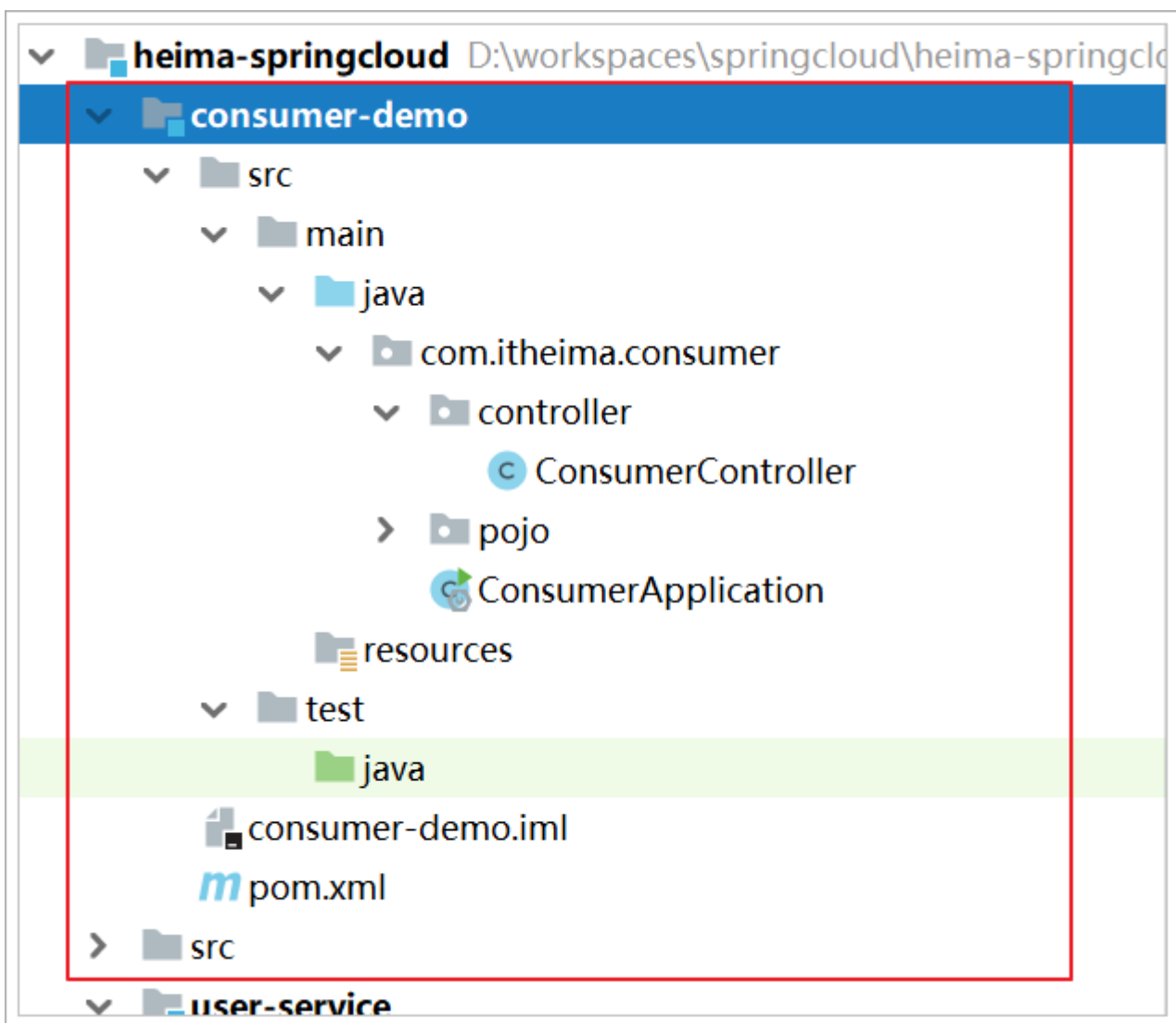
```
1 package com.itheima.consumer.controller;
2
3 import com.itheima.consumer.pojo.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7 import org.springframework.web.bind.annotation.RequestMapping;
8 import org.springframework.web.bind.annotation.RestController;
9 import org.springframework.web.client.RestTemplate;
```

```

10
11 @RestController
12 @RequestMapping("/consumer")
13 public class ConsumerController {
14
15     @Autowired
16     private RestTemplate restTemplate;
17
18     @GetMapping("/{id}")
19     public User queryById(@PathVariable Long id){
20         String url = "http://localhost:9091/user/" + id;
21         return restTemplate.getForObject(url, User.class);
22     }
23
24 }
25

```

服务调用者项目结构：

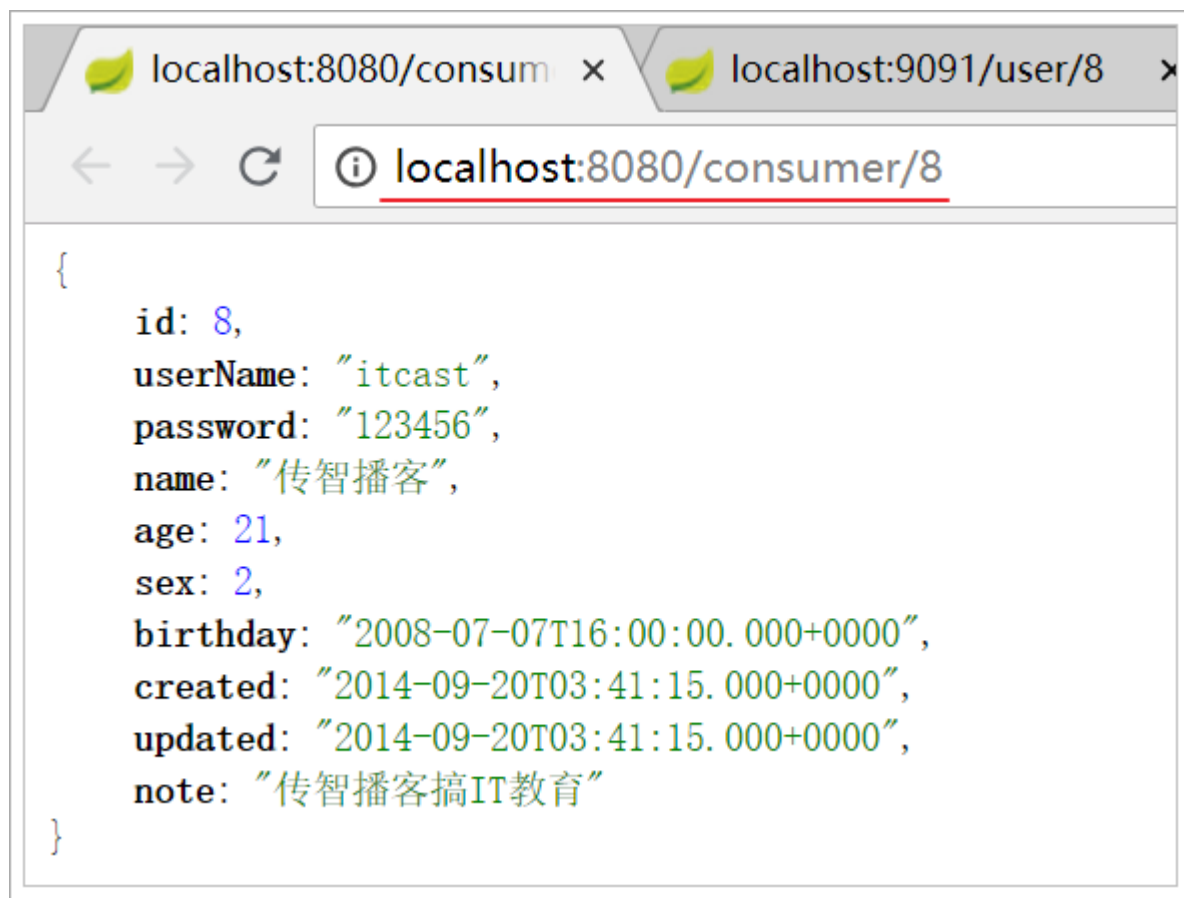


### 5.2.3. 启动测试

启动 consumer-demo 引导启动类；因为 consumer-demo 项目没有配置端口，那么默认就是8080，



我们访问: <http://localhost:8080/consumer/8>



一个简单的远程服务调用案例就实现了。

## 5.3. 思考问题

简单回顾一下，刚才我们写了什么：

**user-service**：对外提供了查询用户的接口

**consumer-demo**：通过RestTemplate访问 <http://localhost:9091/user/{id}> 接口，查询用户数据

存在什么问题？

- 在consumer中，我们把url地址硬编码到了代码中，不方便后期维护
- consumer需要记忆user-service的地址，如果出现变更，可能得不到通知，地址将失效
- consumer不清楚user-service的状态，服务宕机也不知道
- user-service只有1台服务，不具备高可用性
- 即便user-service形成集群，consumer还需自己实现负载均衡

其实上面说的问题，概括一下就是分布式服务必然要面临的问题：

- 服务管理
  - 如何自动注册和发现
  - 如何实现状态监管
  - 如何实现动态路由
- 服务如何实现负载均衡
- 服务如何解决容灾问题

- 服务如何实现统一配置

以上的问题，都将在SpringCloud中得到答案。

## 6. Eureka注册中心

### 6.1. 认识Eureka

首先我们来解决第一问题，服务的管理。

#### 问题分析

在刚才的案例中，user-service对外提供服务，需要对外暴露自己的地址。而consumer-demo（调用者）需要记录服务提供者的地址。将来地址出现变更，还需要及时更新。这在服务较少的时候并不觉得有什么，但是在现在日益复杂的互联网环境，一个项目可能会拆分出十几，甚至几十个微服务。此时如果还人为管理地址，不仅开发困难，将来测试、发布上线都会非常麻烦，这与DevOps的思想是背道而驰的。

DevOps的思想是系统可以通过一组过程、方法或系统；提高应用发布和运维的效率，降低管理成本。

#### 网约车

这就好比是 网约车出现以前，人们出门叫车只能叫出租车。一些私家车想做出租却没有资格，被称为黑车。而很多人想要约车，但是无奈出租车太少，不方便。私家车很多却不敢拦，而且满大街的车，谁知道哪个才是愿意载人的。一个想要，一个愿意给，就是缺少引子，缺乏管理啊。

此时滴滴这样的网约车平台出现了，所有想载客的私家车全部到滴滴注册，记录你的车型（服务类型），身份信息（联系方式）。这样提供服务的私家车，在滴滴那里都能找到，一目了然。

此时要叫车的人，只需要打开APP，输入你的目的地，选择车型（服务类型），滴滴自动安排一个符合需求的车到你面前，为你服务，完美！

#### Eureka做什么？

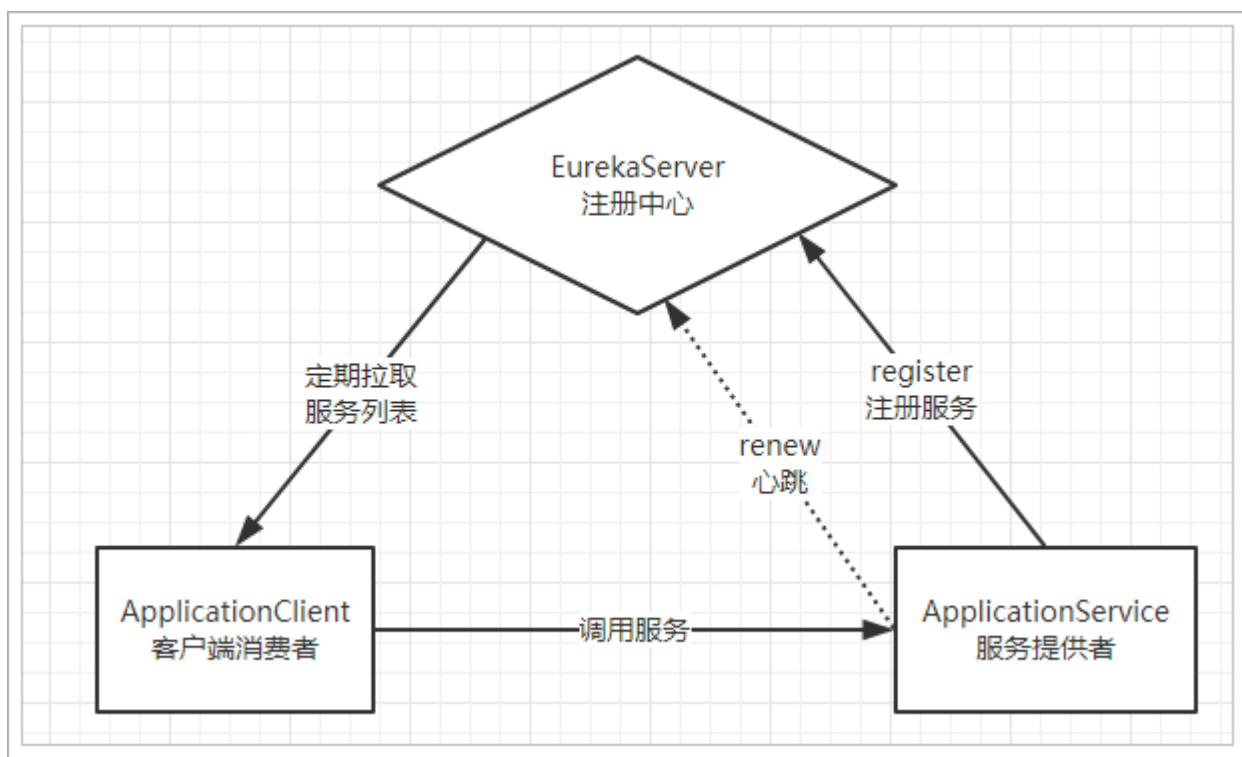
Eureka就好比是滴滴，负责管理、记录服务提供者的信息。服务调用者无需自己寻找服务，而是把自己的需求告诉Eureka，然后Eureka会把符合你需求的服务告诉你。

同时，服务提供方与Eureka之间通过“心跳”机制进行监控，当某个服务提供方出现问题，Eureka自然会把它从服务列表中剔除。

这就实现了服务的自动注册、发现、状态监控。

### 6.2. 原理图

#### 基本架构：



- Eureka：就是服务注册中心（可以是一个集群），对外暴露自己的地址
- 提供者：启动后向Eureka注册自己信息（地址，提供什么服务）
- 消费者：向Eureka订阅服务，Eureka会将对应服务的所有提供者地址列表发送给消费者，并且定期更新
- 心跳(续约)：提供者定期通过http方式向Eureka刷新自己的状态

## 6.3. 入门案例

### 6.3.1. 搭建eureka-server工程

接下来创建一个项目 `eureka-server`，启动一个Eureka Server Application服务注册中心。

The "New Module" dialog box shows the configuration for a new module. The "Add as module to" field is set to "com.itheima:heima-springcloud:1.0-SNAPSHOT". The "Parent" field is also set to "com.itheima:heima-springcloud:1.0-SNAPSHOT". The "GroupId" field is set to "com.itheima". The "ArtifactId" field is set to "eureka-server". The "Version" field is set to "1.0-SNAPSHOT". The "Inherit" checkbox is checked.

The "New Module" dialog box shows the configuration for a new module. The "Module name" field is set to "eureka-server". The "Content root" field is set to "D:\workspaces\springcloud\heima-springcloud\eureka-server". The "Module file location" field is set to "D:\workspaces\springcloud\heima-springcloud\eureka-server".

项目中的 `pom.xml` 文件如下：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <parent>
7         <artifactId>heima-springcloud</artifactId>
8         <groupId>com.itheima</groupId>
9         <version>1.0-SNAPSHOT</version>
10    </parent>
11    <modelVersion>4.0.0</modelVersion>
12
13    <groupId>com.itheima</groupId>
14    <artifactId>eureka-server</artifactId>
15
16    <dependencies>
17        <dependency>
18            <groupId>org.springframework.cloud</groupId>
19            <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
20        </dependency>
21    </dependencies>
22 </project>

```

## 1) 编写启动类

eureka-server\src\main\java\com\itheima\EurekaServerApplication.java

```

1 package com.itheima;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;
6
7 //声明当前应用为eureka服务
8 @EnableEurekaServer
9 @SpringBootApplication
10 public class EurekaServerApplication {
11     public static void main(String[] args) {
12         SpringApplication.run(EurekaServerApplication.class);
13     }
14 }
15

```

## 2) 编写配置文件

eureka-server\src\main\resources\application.yml

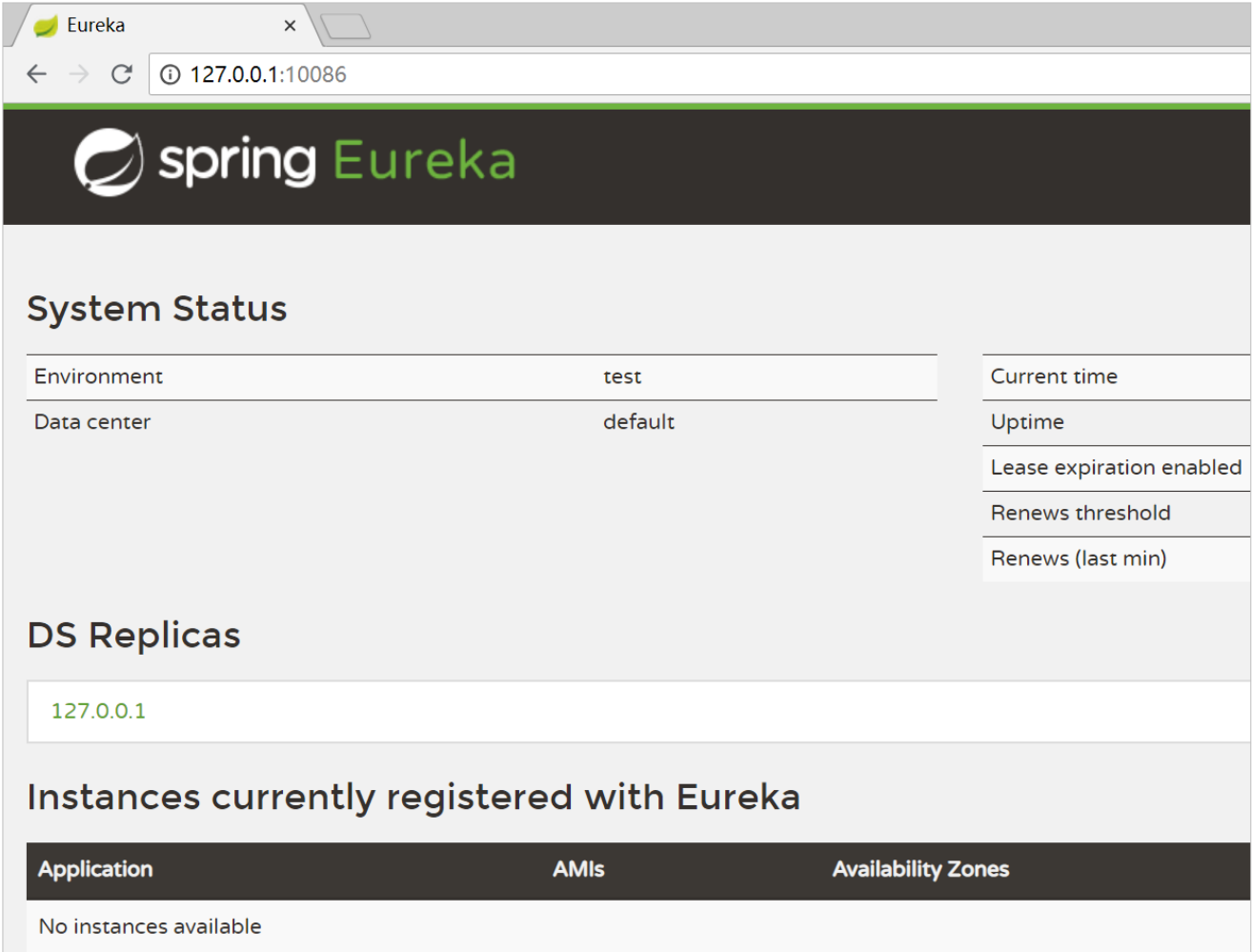
```

1  server:
2    port: 10086
3  spring:
4    application:
5      name: eureka-server # 应用名称, 会在Eureka中作为服务的id标识 (serviceId)
6  eureka:
7    client:
8      service-url: # EurekaServer的地址, 现在是自己的地址, 如果是集群, 需要写其它Server的地址。
9      defaultZone: http://127.0.0.1:10086/eureka
10     register-with-eureka: false # 不注册自己
11     fetch-registry: false #不拉取
12

```

### 3) 启动服务

启动 eureka-server 访问: <http://127.0.0.1:10086>



The screenshot shows the Spring Eureka web interface in a browser window. The address bar shows `127.0.0.1:10086`. The page has a dark header with the "spring Eureka" logo. Below the header, the "System Status" section contains two tables. The first table shows "Environment" as "test" and "Data center" as "default". The second table shows "Current time", "Uptime", "Lease expiration enabled", "Renews threshold", and "Renews (last min)". Below this, the "DS Replicas" section shows a single replica at `127.0.0.1`. The "Instances currently registered with Eureka" section shows a table with headers "Application", "AMIs", and "Availability Zones", and a message "No instances available".

System Status	
Environment	test
Data center	default

Current time
Uptime
Lease expiration enabled
Renews threshold
Renews (last min)

**DS Replicas**

127.0.0.1
-----------

**Instances currently registered with Eureka**

Application	AMIs	Availability Zones
No instances available		

## Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (1)	(1)	UP (1) - tp4jbl:eureka-server:10086

当前服务，现在只有eureka服务本身

## General Info

Name	Value
total-avail-memory	558mb
environment	test
num-of-cpus	8
current-memory-usage	239mb (42%)
server-uptime	00:00
registered-replicas	http://127.0.0.1:10086/eureka/
unavailable-replicas	http://127.0.0.1:10086/eureka/,
available-replicas	

## 6.3.2. 服务注册

注册服务，就是在服务上添加Eureka的客户端依赖，客户端代码会自动把服务注册到EurekaServer中。

### 1) 添加依赖

我们在user-service中添加Eureka客户端依赖：

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

### 2) 修改启动类

在启动类上开启Eureka客户端功能

通过添加 `@EnableDiscoveryClient` 来开启Eureka客户端功能

```

1 @SpringBootApplication
2 @MapperScan("com.itheima.user.mapper")
3 @EnabledDiscoveryClient // 开启Eureka客户端发现功能
4 public class UserServiceDemoApplication {
5     public static void main(String[] args) {
6         SpringApplication.run(UserServiceDemoApplication.class, args);
7     }
8 }

```

### 3) 修改配置文件

编写user-service\src\main\resources\application.yml配置文件为如下：

```

1 server:
2     port: 9091
3 spring:
4     datasource:
5         driver-class-name: com.mysql.jdbc.Driver
6         url: jdbc:mysql://localhost:3306/springcloud
7         username: root
8         password: root
9     application:
10        #应用名
11        name: user-service
12 mybatis:
13     type-aliases-package: cn.itcast.user.pojo
14 eureka:
15     client:
16         service-url:
17             defaultZone: http://localhost:10086/eureka
18

```

注意：

- 这里我们添加了spring.application.name属性来指定应用名称，将来会作为服务的id使用。

### 4) 测试

重启 user-service 项目，访问[Eureka监控页面](#)

#### DS Replicas

127.0.0.1

#### Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
USER-SERVICE	n/a (1)	(1)	UP (1) - localhost:user-service:9091

我们发现user-service服务已经注册成功了

### 6.3.3. 服务发现

接下来我们修改 `consumer-demo`，尝试从EurekaServer获取服务。

方法与服务提供者类似，只需要在项目中添加EurekaClient依赖，就可以通过服务名称来获取信息了！

#### 1) 添加依赖

找到 `consumer-demo\pom.xml` 添加如下依赖

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

#### 2) 修改启动类

修改 `consumer-demo\src\main\java\com\itheima\consumer\ConsumerApplication.java` 开启Eureka客户端

```
1 package com.itheima.consumer;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6 import org.springframework.context.annotation.Bean;
7 import org.springframework.web.client.RestTemplate;
8
9 @SpringBootApplication
10 @EnableDiscoveryClient
11 public class ConsumerApplication {
12     public static void main(String[] args) {
13         SpringApplication.run(ConsumerApplication.class, args);
14     }
15
16     @Bean
17     public RestTemplate restTemplate(){
18         return new RestTemplate();
19     }
20 }
21
```

#### 3) 新增配置文件

新增 `consumer-demo\src\main\resources\application.yml` 配置文件



```

1 server:
2   port: 8080
3 spring:
4   application:
5     name: consumer-demo # 应用名称
6 eureka:
7   client:
8     service-url: # EurekaServer地址
9     defaultZone: http://127.0.0.1:10086/eureka

```

#### 4) 修改处理器

修改 `consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java` 代码，用DiscoveryClient类的方法，根据服务名称，获取服务实例。

```

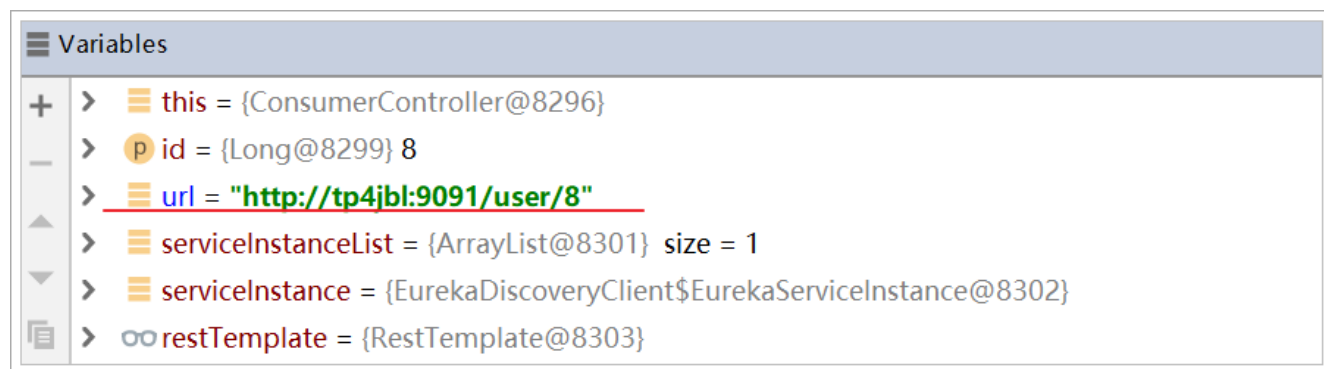
1 package com.itheima.consumer.controller;
2
3 import com.itheima.consumer.pojo.User;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.cloud.client.ServiceInstance;
6 import org.springframework.cloud.client.discovery.DiscoveryClient;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11 import org.springframework.web.client.RestTemplate;
12
13 import java.util.List;
14
15 @RestController
16 @RequestMapping("/consumer")
17 public class ConsumerController {
18
19     @Autowired
20     private RestTemplate restTemplate;
21
22     @Autowired
23     private DiscoveryClient discoveryClient;
24
25     @GetMapping("/{id}")
26     public User queryById(@PathVariable Long id){
27         String url = "http://localhost:9091/user/" + id;
28
29         //获取eureka中注册的user-service实例列表
30         List<ServiceInstance> serviceInstanceList =
31         discoveryClient.getInstances("user-service");
32         ServiceInstance serviceInstance = serviceInstanceList.get(0);
33
34         url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort()
35         + "/user/" + id;

```

```
34
35     return restTemplate.getForObject(url, User.class);
36 }
37
38 }
39
```

## 5) Debug跟踪运行

重启 `consumer-demo` 项目；然后再浏览器中再次访问 <http://localhost:8080/consumer/8>；在代码中debug跟进查看最终拼接要访问的URL：



## 6.4. Eureka详解

### 6.4.1. 基础架构

Eureka架构中的三个核心角色：

- 服务注册中心  
Eureka的服务端应用，提供服务注册和发现功能，就是刚刚我们建立的eureka-server
- 服务提供者  
提供服务的应用，可以是SpringBoot应用，也可以是其它任意技术实现，只要对外提供的是Rest风格服务即可。本例中就是我们实现的user-service
- 服务消费者  
消费应用从注册中心获取服务列表，从而得知每个服务方的信息，知道去哪里调用服务方。本例中就是我们实现的consumer-demo

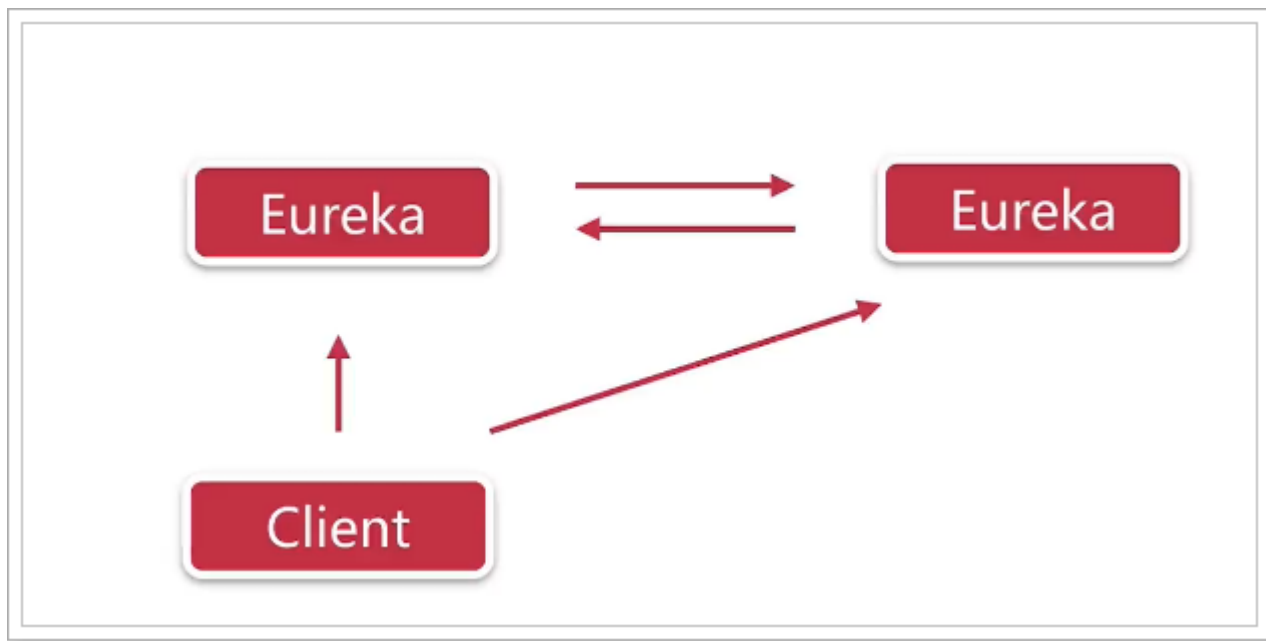
### 6.4.2. 高可用的Eureka Server

Eureka Server即服务的注册中心，在刚才的案例中，我们只有一个EurekaServer，事实上EurekaServer也可以是一个集群，形成高可用的Eureka中心。

#### 服务同步

多个Eureka Server之间也会互相注册为服务，当服务提供者注册到Eureka Server集群中的某个节点时，该节点会把服务的信息同步给集群中的每个节点，从而实现**数据同步**。因此，无论客户端访问到Eureka Server集群中的任意一个节点，都可以获取到完整的服务列表信息。

而作为客户端，需要把信息注册到每个Eureka中：



如果有三个Eureka，则每一个EurekaServer都需要注册到其它几个Eureka服务中，例如：有三个分别为10086、10087、10088，则：

- 10086要注册到10087和10088上
- 10087要注册到10086和10088上
- 10088要注册到10086和10087上

动手搭建高可用的EurekaServer

我们假设要搭建两台EurekaServer的集群，端口分别为：10086和10087

1) 修改原来的EurekaServer配置；修改 `eureka-server\src\main\resources\application.yml` 如下：

```
1  server:
2    port: ${port:10086}
3
4  spring:
5    application:
6      # 应用名称，会在eureka中作为服务的id(serviceId)
7      name: eureka-server
8  eureka:
9    client:
10     service-url:
11       # eureka服务地址；如果是集群则是其它服务器地址，后面要加/eureka
12       defaultZone: ${defaultZone:http://127.0.0.1:10086/eureka}
13     # 是否注册自己，自身不提供服务所以不注册
14     #register-with-eureka: false
15     # 是否拉取服务
16     #fetch-registry: false
```

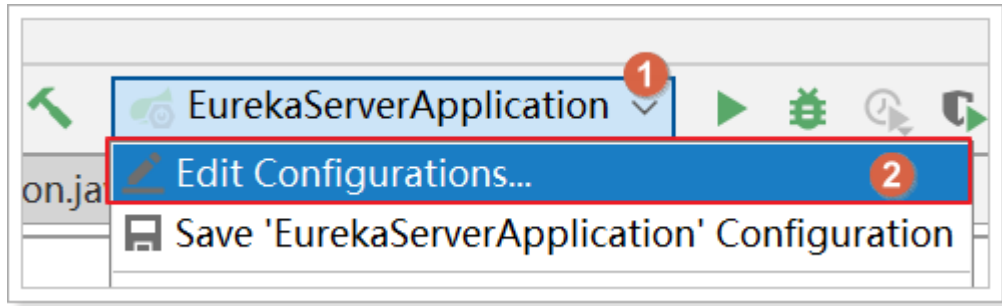
所谓的高可用注册中心，其实就是把EurekaServer自己也作为一个服务，注册到其它EurekaServer上，这样多个EurekaServer之间就能互相发现对方，从而形成集群。因此我们做了以下修改：

注意把register-with-eureka和fetch-registry修改为true或者注释掉

在上述配置文件中的\${}表示在jvm启动时候若能找到对应port或者defaultZone参数则使用，若无则使用后面的默认值

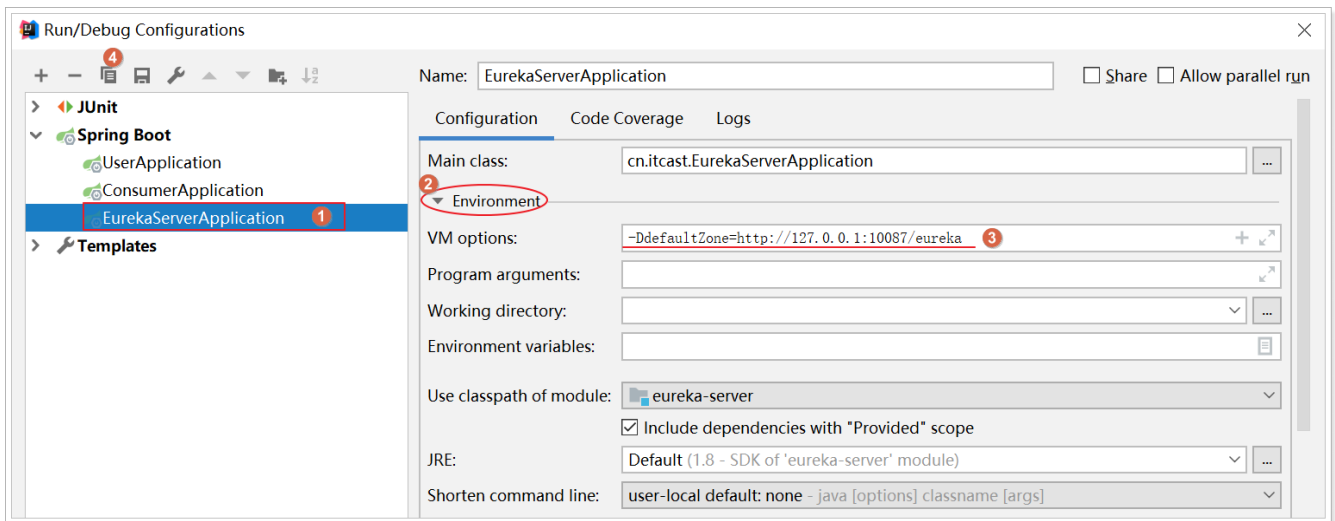
- 把service-url的值改成了另外一台EurekaServer的地址，而不是自己

2) 另外一台在启动的时候可以指定端口port和defaultZone配置：



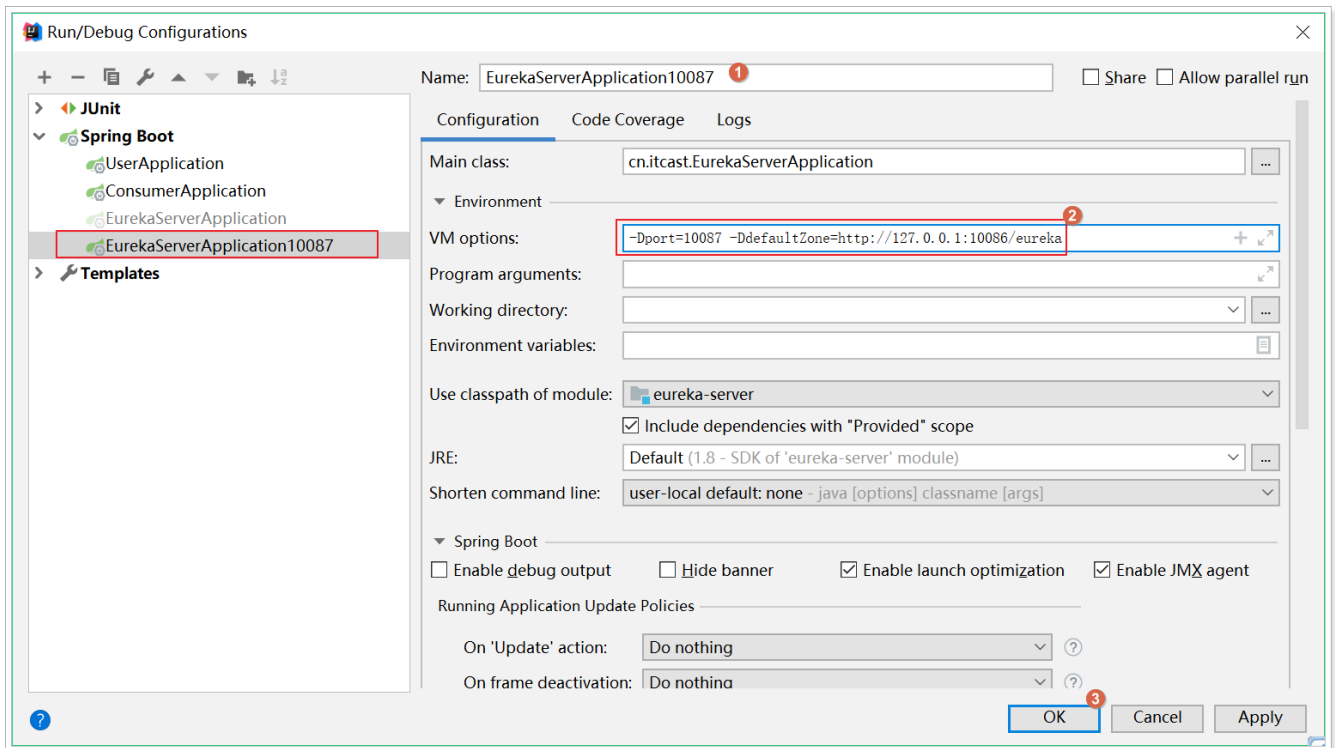
修改原来的启动配置组件；在如下界面中的 `VM options` 中

设置 `-DdefaultZone=http:127.0.0.1:10087/eureka`

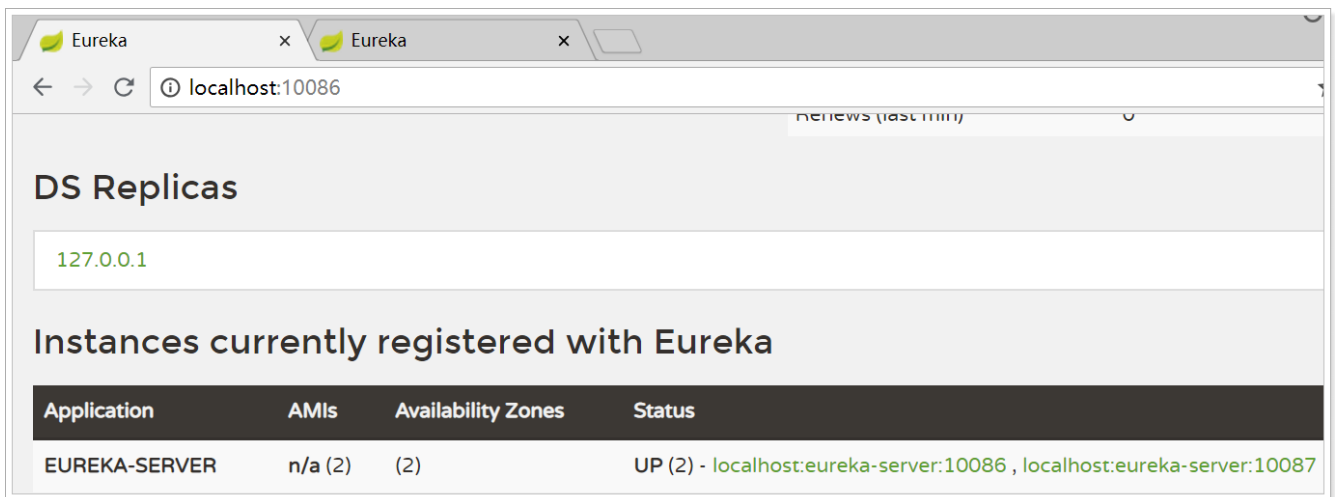


复制一份并修改；在如下界面中的 `VM options` 中

设置 `-Dport=10087 -DdefaultZone=http:127.0.0.1:10086/eureka`



### 3) 启动测试；同时启动两台eureka server



### 4) 客户端注册服务到集群

因为EurekaServer不止一个，因此user-service项目注册服务或者consumer-demo获取服务的时候，service-url参数需要修改为如下：

```

1 eureka:
2   client:
3     service-url: # EurekaServer地址,多个地址以','隔开
4     defaultZone: http://127.0.0.1:10086/eureka,http://127.0.0.1:10087/eureka

```

为了方便上课和后面内容的修改，在测试完上述配置后可以再次改回单个eureka server的方式。

### 6.4.3. Eureka客户端

服务提供者要向EurekaServer注册服务，并且完成服务续约等工作。

#### 服务注册

服务提供者在启动时，会检测配置属性中的：`eureka.client.register-with-eureka=true` 参数是否正确，事实上默认就是true。如果值确实为true，则会向EurekaServer发起一个Rest请求，并携带自己的元数据信息，Eureka Server会把这些信息保存到一个双层Map结构中。

- 第一层Map的Key就是服务id，一般是配置中的 `spring.application.name` 属性
- 第二层Map的key是服务的实例id。一般host+ serviceId + port，例如：`localhost:user-service:8081`
- 值则是服务的实例对象，也就是说一个服务，可以同时启动多个不同实例，形成集群。

默认注册时使用的是主机名或者localhost，如果想用ip进行注册，可以在 `user-service` 中添加配置如下：

```
1 eureka:
2   instance:
3     ip-address: 127.0.0.1 # ip地址
4     prefer-ip-address: true # 更倾向于使用ip，而不是host名
```

修改完后先后重启 `user-service` 和 `consumer-demo`；在调用服务的时候就已经变成ip地址；需要注意的是：*不是在eureka中的控制台服务实例状态显示。*

#### 服务续约

在注册服务完成以后，服务提供者会维持一个心跳（定时向EurekaServer发起Rest请求），告诉EurekaServer：“我还活着”。这个我们称为服务的续约（renew）；

有两个重要参数可以修改服务续约的行为；可以在 `user-service` 中添加如下配置项：

```
1 eureka:
2   instance:
3     lease-expiration-duration-in-seconds: 90
4     lease-renewal-interval-in-seconds: 30
```

- `lease-renewal-interval-in-seconds`：服务续约(renew)的间隔，默认为30秒
- `lease-expiration-duration-in-seconds`：服务失效时间，默认值90秒

也就是说，默认情况下每隔30秒服务会向注册中心发送一次心跳，证明自己还活着。如果超过90秒没有发送心跳，EurekaServer就会认为该服务宕机，会定时（`eureka.server.eviction-interval-timer-in-ms`设定的时间）从服务列表中移除，这两个值在生产环境不要修改，默认即可。

#### 获取服务列表

当服务消费者启动时，会检测 `eureka.client.fetch-registry=true` 参数的值，如果为true，则会从Eureka Server服务的列表拉取只读备份，然后缓存在本地。并且每隔30秒会重新拉取并更新数据。可以在 `consumer-demo` 项目中通过下面的参数来修改：

```
1 eureka:
2   client:
3     registry-fetch-interval-seconds: 30
```

### 6.4.5. 失效剔除和自我保护

如下的配置都是在Eureka Server服务端进行：

#### 服务下线

当服务进行正常关闭操作时，它会触发一个服务下线的REST请求给Eureka Server，告诉服务注册中心：“我要下线了”。服务中心接受到请求之后，将该服务置为下线状态。

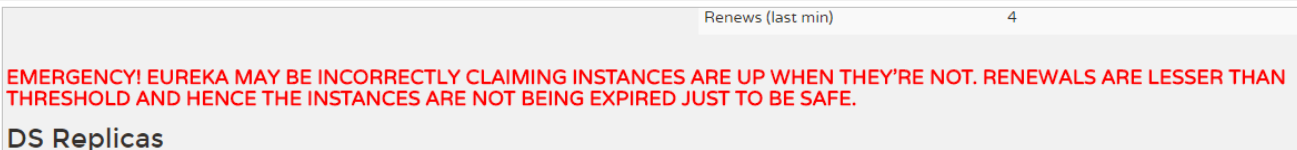
#### 失效剔除

有时我们的服务可能由于内存溢出或网络故障等原因使得服务不能正常的工作，而服务注册中心并未收到“服务下线”的请求。相对于服务提供者的“服务续约”操作，服务注册中心在启动时会创建一个定时任务，默认每隔一段时间（默认为60秒）将当前清单中超时（默认为90秒）没有续约的服务剔除，这个操作被称为失效剔除。

可以通过 `eureka.server.eviction-interval-timer-in-ms` 参数对其进行修改，单位是毫秒。

#### 自我保护

我们关停一个服务，很可能在Eureka面板看到一条警告：



The screenshot shows a console window with a title bar that says "Renews (last min)" and a close button. The main content area has a red warning message: "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 the warning, the text "DS Replicas" is visible.

这是触发了Eureka的自我保护机制。当服务未按时进行心跳续约时，Eureka会统计服务实例最近15分钟心跳续约的比例是否低于了85%。在生产环境下，因为网络延迟等原因，心跳失败实例的比例很有可能超标，但是此时就把服务剔除列表并不妥当，因为服务可能没有宕机。Eureka在这段时间内不会剔除任何服务实例，直到网络恢复正常。生产环境下这很有效，保证了大多数服务依然可用，不过也有可能获取到失败的服务实例，因此服务调用者必须做好服务的失败容错。

可以通过下面的配置来关停自我保护：

```
1 eureka:
2   server:
3     enable-self-preservation: false # 关闭自我保护模式（缺省为打开）
```

## 7. 负载均衡Ribbon

在刚才的案例中，我们启动了一个 `user-service`，然后通过DiscoveryClient来获取服务实例信息，然后获取ip和端口来访问。

但是实际环境中，往往会开启很多个 `user-service` 的集群。此时获取的服务列表中就会有多个，到底该访问哪一个呢？

一般这种情况下就需要编写负载均衡算法，在多个实例列表中进行选择。

不过Eureka中已经集成了负载均衡组件：Ribbon，简单修改代码即可使用。

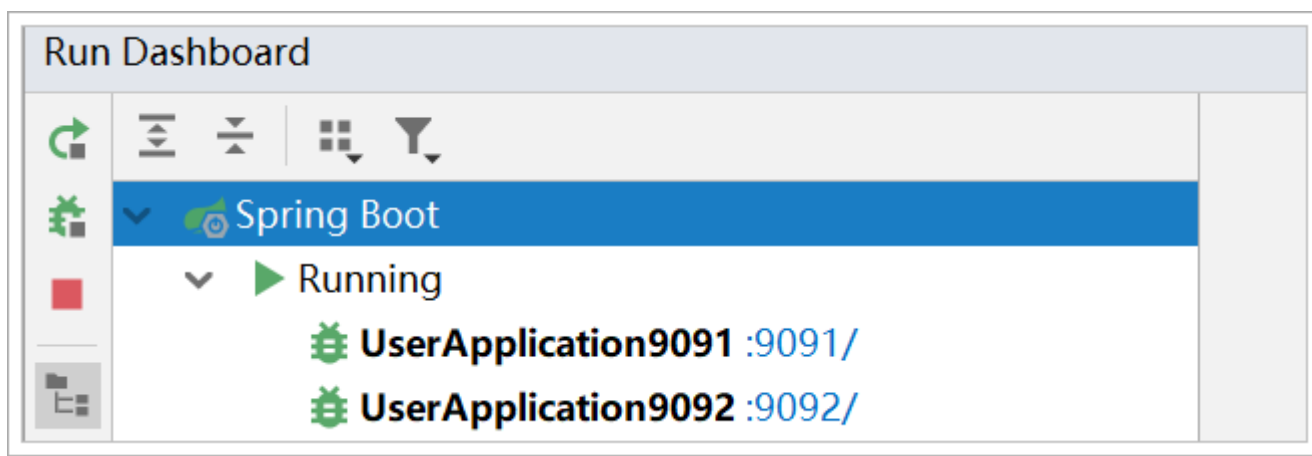
什么是Ribbon：

Ribbon 是 Netflix 发布的负载均衡器，它有助于控制 HTTP 和 TCP 客户端的行为。为 Ribbon 配置服务提供者地址列表后，Ribbon 就可基于某种负载均衡算法，自动地帮助服务消费者去请求。Ribbon 默认为我们提供了很多的负载均衡算法，例如轮询、随机等。当然，我们也可为 Ribbon 实现自定义的负载均衡算法。

接下来，我们就来使用Ribbon实现负载均衡。

## 7.1. 启动两个服务实例

首先我们配置启动两个 `user-service` 实例，一个9091，一个9092。



Eureka监控面板：

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
USER-SERVICE	n/a (2)	(2)	UP (2) - localhost:user-service:9092 , localhost:user-service:9091

## 7.2. 开启负载均衡

因为Eureka中已经集成了Ribbon，所以我们无需引入新的依赖。

直接修改 `consumer-demo\src\main\java\com\itheima\consumer\ConsumerApplication.java`

在RestTemplate的配置方法上添加 `@LoadBalanced` 注解：



```
1 @Bean
2 @LoadBalanced
3 public RestTemplate restTemplate() {
4     return new RestTemplate();
5 }
```

修改 `consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java` 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；

```
1 @GetMapping("/{id}")
2 public User queryById(@PathVariable("id") Long id){
3     String url = "http://user-service/user/" + id;
4     User user = restTemplate.getForObject(url, User.class);
5     return user;
6 }
```

访问页面，查看结果；并可以在9091和9092的控制台查看执行情况：

了解：Ribbon默认的负载均衡策略是轮询。SpringBoot也帮提供了修改负载均衡规则的配置入口在consumer-demo的配置文件中添加如下，就变成随机的了：

```
1 user-service:
2     ribbon:
3         NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

格式是：{服务名称}.ribbon.NFLoadBalancerRuleClassName

## 7.3. 源码跟踪

为什么只输入了service名称就可以访问了呢？之前还要获取ip和端口。

显然是有组件根据service名称，获取到了服务实例的ip和端口。因为 consumer-demo 使用的是RestTemplate，spring的负载均衡自动配置类 `LoadBalancerAutoConfiguration.LoadBalancerInterceptorConfig` 会自动配置负载均衡拦截器（在spring-cloud-commons-\*.jar包中的spring.factories中定义的自动配置类），它就是 `LoadBalancerInterceptor`，这个类会在对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实的服务地址信息，替换服务id。

我们进行源码跟踪：

```

public class LoadBalancerInterceptor implements ClientHttpRequestInterceptor {

    private LoadBalancerClient loadBalancer; loadBalancer: RibbonLoadBalancerClient@8374
    private LoadBalancerRequestFactory requestFactory; requestFactory: LoadBalancerRequestFactory@8373

    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer, LoadBalancerRequestFactory requestFactory) {
        this.loadBalancer = loadBalancer;
        this.requestFactory = requestFactory;
    }

    public LoadBalancerInterceptor(LoadBalancerClient loadBalancer) {
        // for backwards compatibility
        this(loadBalancer, new LoadBalancerRequestFactory(loadBalancer));
    }

    @Override
    public ClientHttpResponse intercept(final HttpRequest request, final byte[] body, request: InterceptingClientHttpRequestExecution execution) throws IOException {
        final ClientHttpRequestExecution execution) throws IOException {
            execution: InterceptingClientHttpRequestExecution
            final URI originalUri = request.getURI(); originalUri: "http://user-service/user/8" request: InterceptingClientHttpRequestExecution
            String serviceName = originalUri.getHost(); originalUri: "http://user-service/user/8"
            Assert.state(expression: serviceName != null, message: "Request URI does not contain a valid hostname: " + originalUri.getHost());
            return this.loadBalancer.execute(serviceName, requestFactory.createRequest(request, body, execution));
        }
    }
}

```

继续跟入execute方法：发现获取了9092端口的服务

```

public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) throws IOException {
    ILoadBalancer loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer"
    Server server = getServer(loadBalancer, hint); loadBalancer: "DynamicServerListLoadBalancer"
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }
    RibbonServer ribbonServer = new RibbonServer(serviceId, server, isSecure(server, serviceId), serverIntrospector(serviceId).getMetadata(server));

    return execute(serviceId, ribbonServer, request);
}

```

再跟下一次，发现获取的是9091、9092之间切换：

```

@Override
public <T> T execute(String serviceId, ServiceInstance serviceInstance, LoadBalancerRequest<T> request) throws IOException {
    Server server = null; server: "127.0.0.1:9092"
    if (serviceInstance instanceof RibbonServer) {
        server = ((RibbonServer) serviceInstance).getServer(); serviceInstance: "RibbonServer{server: 127.0.0.1:9092}"
    }
    if (server == null) {
        throw new IllegalStateException("No instances available for " + serviceId);
    }

    RibbonLoadBalancerContext context = this.clientFactory
        .getLoadBalancerContext(serviceId);
    RibbonStatsRecorder statsRecorder = new RibbonStatsRecorder(context, server);

    try {
        T returnVal = request.apply(serviceInstance);
        statsRecorder.recordStats(returnVal);
        return returnVal;
    }
}

```

多次访问 consumer-demo 的请求地址；然后跟进代码，发现其果然实现了负载均衡。

## 8. 熔断器Hystrix

### 8.1. 简介

Hystrix 在英文里面的意思是 豪猪，它的logo 看下面的图是一头豪猪，它在微服务系统中是一款提供保护机制的组件，和eureka一样也是由netflix公司开发。

主页: <https://github.com/Netflix/Hystrix/>

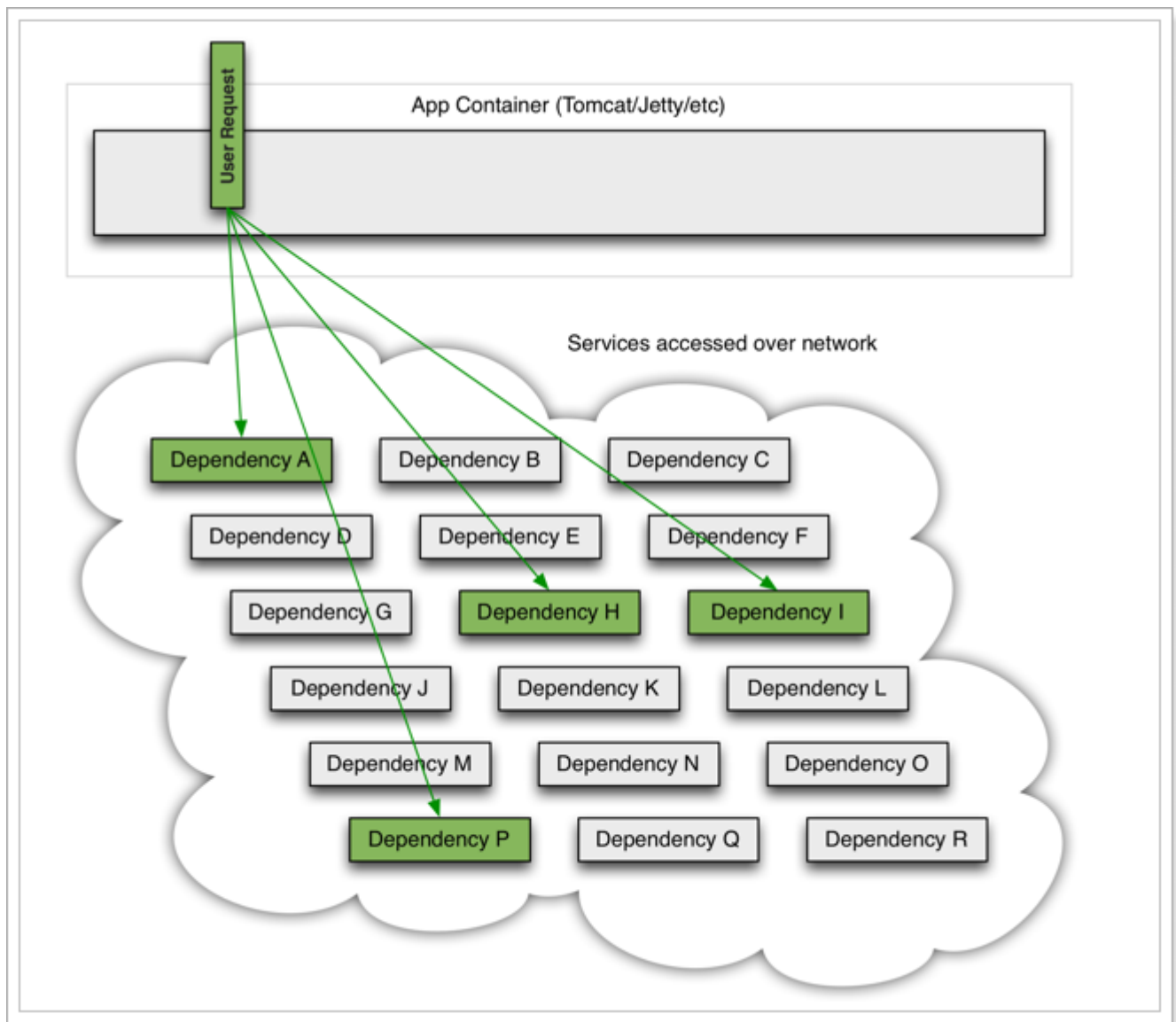


那么Hystrix的作用是什么呢？具体要保护什么呢？

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。

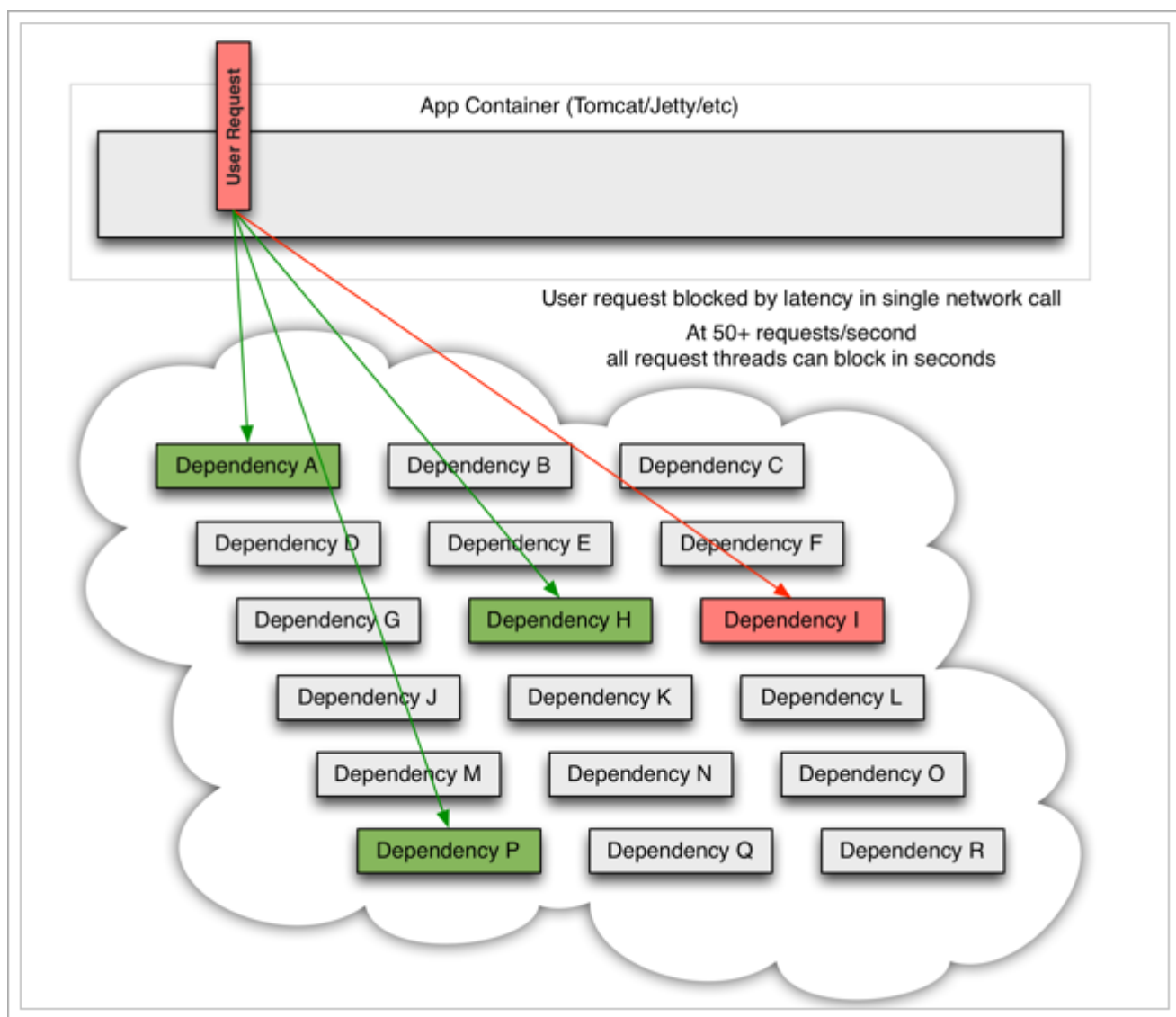
## 8.2. 雪崩问题

微服务中，服务间调用关系错综复杂，一个请求，可能需要调用多个微服务接口才能实现，会形成非常复杂的调用链路：

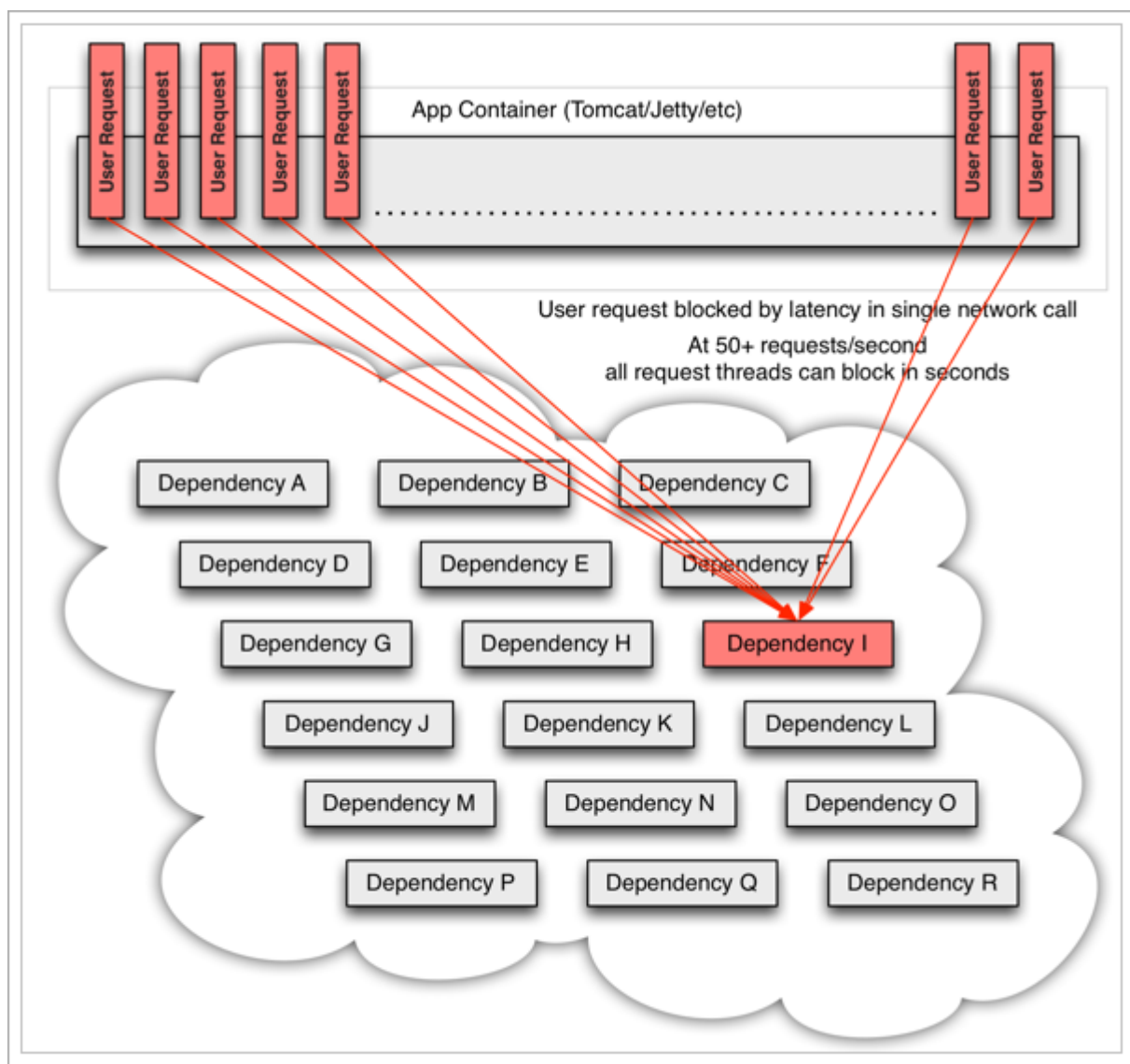


如图，一次业务请求，需要调用A、P、H、I四个服务，这四个服务又可能调用其它服务。

如果此时，某个服务出现异常：



例如：微服务I 发生异常，请求阻塞，用户请求就不会得到响应，则tomcat的这个线程不会释放，于是越来越多的用户请求到来，越来越多的线程会阻塞：



服务器支持的线程和并发数有限，请求一直阻塞，会导致服务器资源耗尽，从而导致所有其它服务都不可用，形成雪崩效应。

这就好比，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使用，那么就会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，导致所有生产都陷入瘫痪。一个零件的波及范围不断扩大。

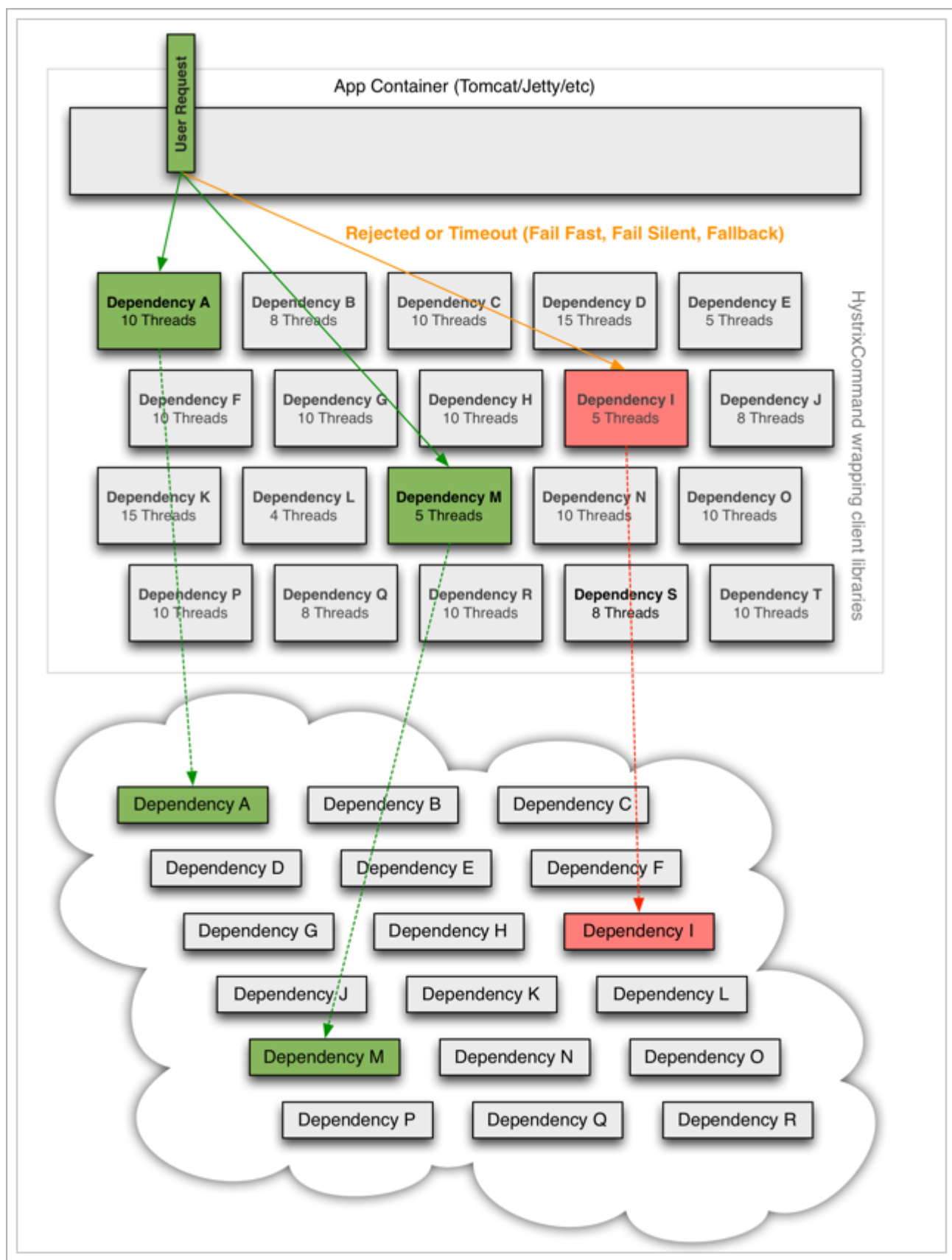
Hystrix解决雪崩问题的手段主要是服务降级，包括：

- 线程隔离
- 服务熔断

## 8.3. 线程隔离&服务降级

### 8.3.1. 原理

线程隔离示意图：



解读：

- Hystrix为每个依赖服务调用分配一个小的线程池，如果线程池已满调用将被立即拒绝，默认不采用排队，**加速失败判定时间**。

- 用户的请求将不再直接访问服务，而是通过线程池中的空闲线程来访问服务，如果**线程池已满**，或者**请求超时**，则会进行降级处理，什么是服务降级？

服务降级：优先保证核心服务，而非核心服务不可用或弱可用。

用户的请求故障时，不会被阻塞，更不会无休止的等待或者看到系统崩溃，至少可以看到一个执行结果（例如返回友好的提示信息）。

**服务降级虽然会导致请求失败，但是不会导致阻塞，而且最多会影响这个依赖服务对应的线程池中的资源，对其它服务没有响应。**

触发Hystrix服务降级的情况：

- 线程池已满
- 请求超时

## 8.3.2. 动手实践

### 1) 引入依赖

在 `consumer-demo` 消费端系统的pom.xml文件添加如下依赖：

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
4 </dependency>
```

### 2) 开启熔断

在启动类 `ConsumerApplication` 上添加注解：`@EnableCircuitBreaker`

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 @EnableCircuitBreaker
4 public class ConsumerApplication {
5     // ...
6 }
```

可以看到，我们类上的注解越来越多，在微服务中，经常会引入上面的三个注解，于是Spring就提供了一个组合注解：`@SpringCloudApplication`

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootApplication
@EnableDiscoveryClient
@EnableCircuitBreaker
public @interface SpringCloudApplication {
}
```



因此，我们可以使用这个组合注解来代替之前的3个注解。

```
1 @SpringCloudApplication
2 public class ConsumerApplication {
3     // ...
4 }
```

### 3) 编写降级逻辑

当目标服务的调用出现故障，我们希望快速失败，给用户一个友好提示。因此需要提前编写好失败时的降级处理逻辑，要使用HystrixCommand来完成。

改造 `consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java` 处理器类，如下：

```
1 package com.itheima.consumer.controller;
2
3 import com.itheima.consumer.pojo.User;
4 import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
5 import lombok.extern.slf4j.Slf4j;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.cloud.client.ServiceInstance;
8 import org.springframework.cloud.client.discovery.DiscoveryClient;
9 import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RestController;
13 import org.springframework.web.client.RestTemplate;
14
15 import java.util.List;
16
17 @RestController
18 @RequestMapping("/consumer")
19 @Slf4j
20 public class ConsumerController {
21
22     @Autowired
23     private RestTemplate restTemplate;
24
25     @Autowired
26     private DiscoveryClient discoveryClient;
27
28     @GetMapping("/{id}")
29     @HystrixCommand(fallbackMethod = "queryByIdFallback")
30     public String queryById(@PathVariable Long id){
31         String url = "http://localhost:9091/user/" + id;
32
33         //获取eureka中注册的用户-service实例列表
34         /*List<ServiceInstance> serviceInstanceList =
35         discoveryClient.getInstances("user-service");
36         ServiceInstance serviceInstance = serviceInstanceList.get(0);
```

```

36
37     url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort()
+ "/user/" + id;*/
38     url = "http://user-service/user/" + id;
39
40     return restTemplate.getForObject(url, String.class);
41 }
42
43 public String queryByIdFallback(Long id){
44     log.error("查询用户信息失败。id: {}", id);
45     return "对不起，网络太拥挤了! ";
46 }
47
48 }

```

要注意；因为熔断的降级逻辑方法必须跟正常逻辑方法保证：**相同的参数列表和返回值声明**。

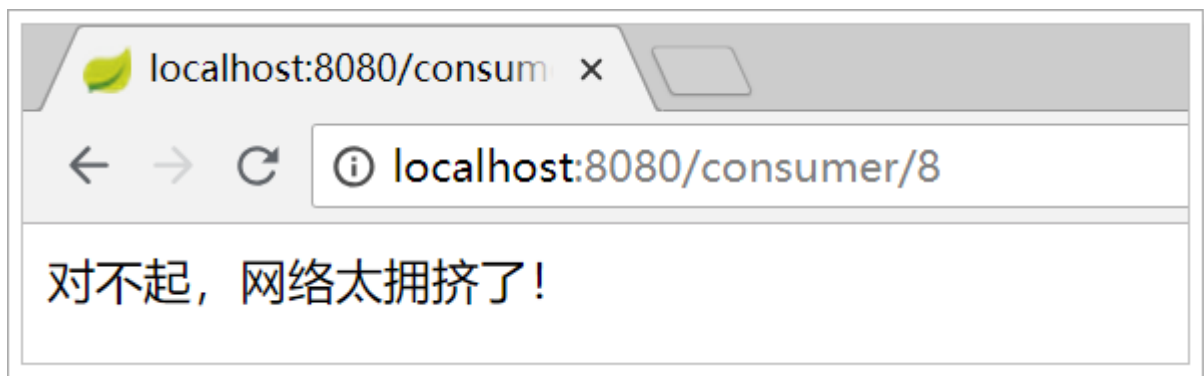
失败逻辑中返回User对象没有太大意义，一般会返回友好提示。所以把queryById的方法改造为返回String，反正也是Json数据。这样失败逻辑中返回一个错误说明，会比较方便。

说明：

- @HystrixCommand(fallbackMethod = "queryByIdFallBack")：用来声明一个降级逻辑的方法

测试：

当 user-service 正常提供服务时，访问与以前一致。但是当将 user-service 停机时，会发现页面返回了降级处理信息：



#### 4) 默认的Fallback

刚才把fallback写在了某个业务方法上，如果这样的方法很多，那岂不是要写很多。所以可以把Fallback配置加在类上，实现默认fallback；

再次改造 consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java

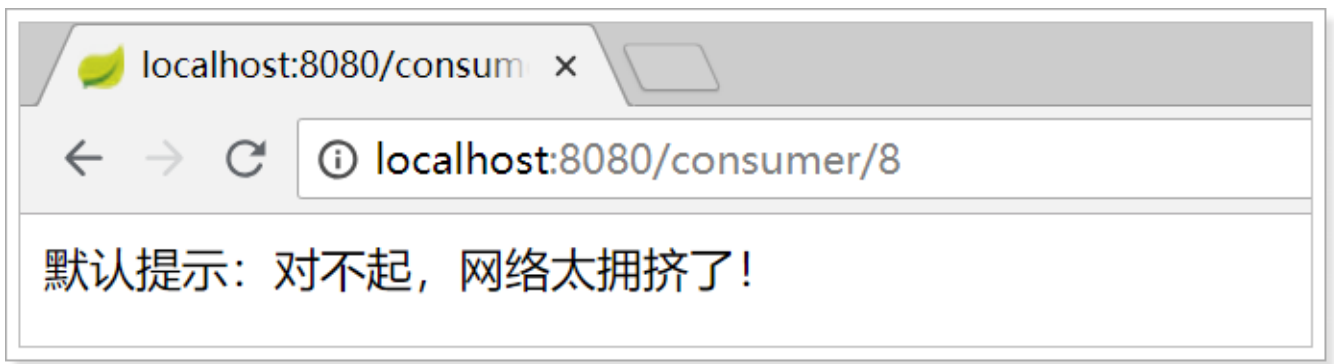
```

1 package com.itheima.consumer.controller;
2
3 import com.itheima.consumer.pojo.User;
4 import com.netflix.hystrix.contrib.javanica.annotation.DefaultProperties;
5 import com.netflix.hystrix.contrib.javanica.annotation.HystrixCommand;
6 import lombok.extern.slf4j.Slf4j;
7 import org.springframework.beans.factory.annotation.Autowired;

```

```
8 import org.springframework.cloud.client.ServiceInstance;
9 import org.springframework.cloud.client.discovery.DiscoveryClient;
10 import org.springframework.web.bind.annotation.GetMapping;
11 import org.springframework.web.bind.annotation.PathVariable;
12 import org.springframework.web.bind.annotation.RequestMapping;
13 import org.springframework.web.bind.annotation.RestController;
14 import org.springframework.web.client.RestTemplate;
15
16 import java.util.List;
17
18 @RestController
19 @RequestMapping("/consumer")
20 @Slf4j
21 @DefaultProperties(defaultFallback = "defaultFallback")
22 public class ConsumerController {
23
24     @Autowired
25     private RestTemplate restTemplate;
26
27     @Autowired
28     private DiscoveryClient discoveryClient;
29
30     @GetMapping("/{id}")
31     // @HystrixCommand(fallbackMethod = "queryByIdFallback")
32     @HystrixCommand
33     public String queryById(@PathVariable Long id){
34         String url = "http://localhost:9091/user/" + id;
35
36         // 获取eureka中注册的用户-service实例列表
37         /*List<ServiceInstance> serviceInstanceList =
38         discoveryClient.getInstances("user-service");
39         ServiceInstance serviceInstance = serviceInstanceList.get(0);
40
41         url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort()
42         + "/user/" + id;*/
43         url = "http://user-service/user/" + id;
44
45         return restTemplate.getForObject(url, String.class);
46     }
47
48     public String queryByIdFallback(Long id){
49         log.error("查询用户信息失败。id: {}", id);
50         return "对不起，网络太拥挤了! ";
51     }
52
53     public String defaultFallback(){
54         return "默认提示：对不起，网络太拥挤了! ";
55     }
56 }
```

- @DefaultProperties(defaultFallback = "defaultFallBack"): 在类上指明统一的失败降级方法；该类中所有方法返回类型要与处理失败的方法的返回类型一致。



## 5) 超时设置

在之前的案例中，请求在超过1秒后都会返回错误信息，这是因为Hystrix的默认超时时长为1，我们可以通过配置修改这个值；修改 `consumer-demo\src\main\resources\application.yml` 添加如下配置：

```
1 hystrix:
2   command:
3     default:
4       execution:
5         isolation:
6           thread:
7             timeoutInMilliseconds: 2000
```

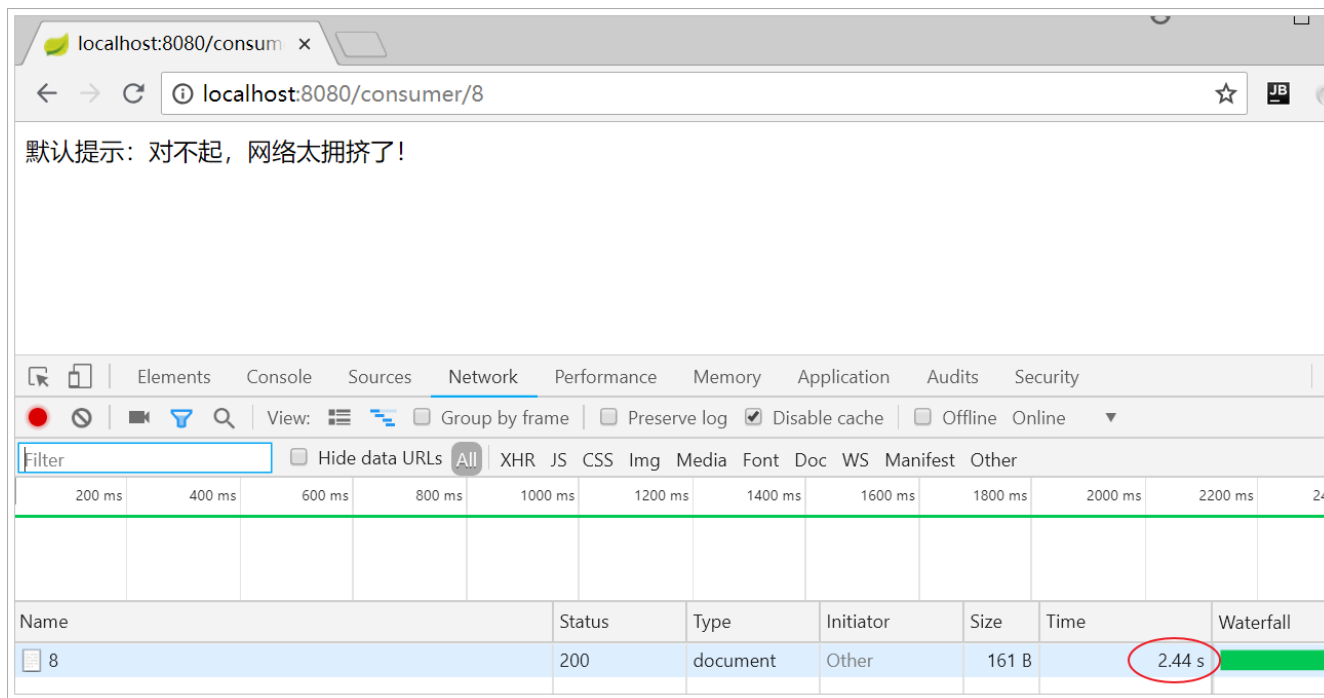
这个配置会作用于全局所有方法。为了方便复制到yml配置文件中，可以复制

`hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=2000` 到yml文件中会自动格式化后再进行修改。

为了触发超时，可以在 `user-service\src\main\java\com\itheima\user\service\UserService.java` 的方法中休眠2秒；

```
1 @Service
2 public class UserService {
3
4     @Autowired
5     private UserMapper userMapper;
6
7     public User queryById(Long id) {
8         try {
9             Thread.sleep(2000);
10        } catch (InterruptedException e) {
11            e.printStackTrace();
12        }
13        return userMapper.selectByPrimaryKey(id);
14    }
15 }
```

测试：



可以发现，请求的时长已经到了2s+，证明配置生效了。如果把修改时间修改到2秒以下，又可以正常访问。

## 8.4. 服务熔断

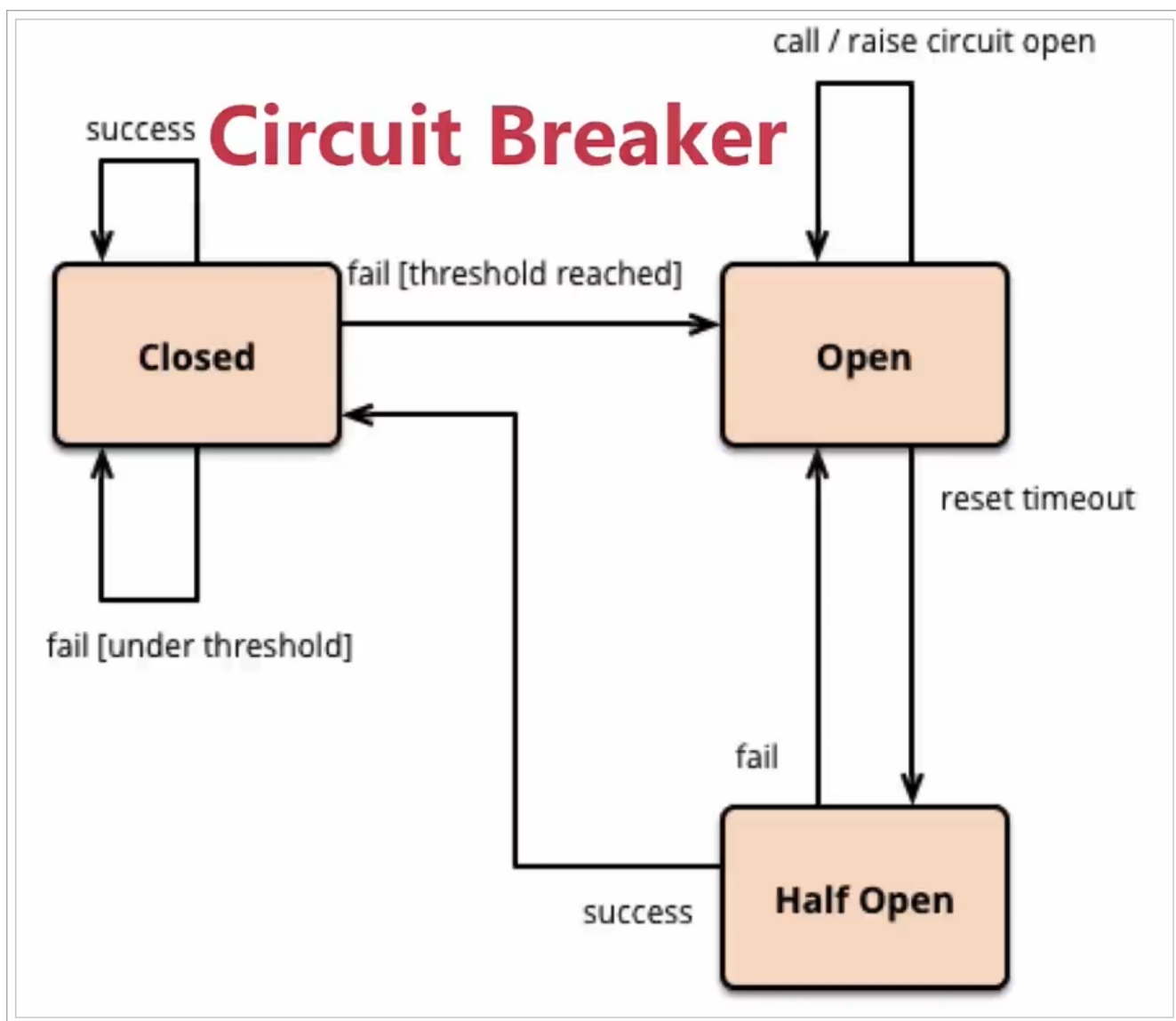
### 8.4.1. 熔断原理

在服务熔断中，使用的熔断器，也叫断路器，其英文单词为：Circuit Breaker

熔断机制与家里使用的电路熔断原理类似；当如果电路发生短路的时候能立刻熔断电路，避免发生灾难。在分布式系统中应用服务熔断后；服务调用方可以自己进行判断哪些服务反应慢或存在大量超时，可以针对这些服务进行主动熔断，防止整个系统被拖垮。

Hystrix的服务熔断机制，可以实现弹性容错；当服务请求情况好转之后，可以自动重连。通过断路的方式，将后续请求直接拒绝，一段时间（默认5秒）之后允许部分请求通过，如果调用成功则回到断路器关闭状态，否则继续打开，拒绝请求的服务。

Hystrix的熔断状态机模型：



状态机有3个状态：

- Closed：关闭状态（断路器关闭），所有请求都正常访问。
- Open：打开状态（断路器打开），所有请求都会被降级。Hystrix会对请求情况计数，当一定时间内失败请求百分比达到阈值，则触发熔断，断路器会完全打开。默认失败比例的阈值是50%，请求次数最少不低于20次。
- Half Open：半开状态，不是永久的，断路器打开后会进入休眠时间（默认是5S）。随后断路器会自动进入半开状态。此时会释放部分请求通过，若这些请求都是健康的，则会关闭断路器，否则继续保持打开，再次进行休眠计时

### 8.4.2. 动手实践

为了能够精确控制请求的成功或失败，在 `consumer-demo` 的处理器业务方法中加入一段逻辑；

修改 `consumer-demo\src\main\java\com\itheima\consumer\controller\ConsumerController.java`

```

1 @GetMapping("{id}")
2 @HystrixCommand
3 public String queryById(@PathVariable("id") Long id){
4     if(id == 1){
5         throw new RuntimeException("太忙了");
6     }
7     String url = "http://user-service/user/" + id;
8     String user = restTemplate.getForObject(url, String.class);
9     return user;
10 }

```

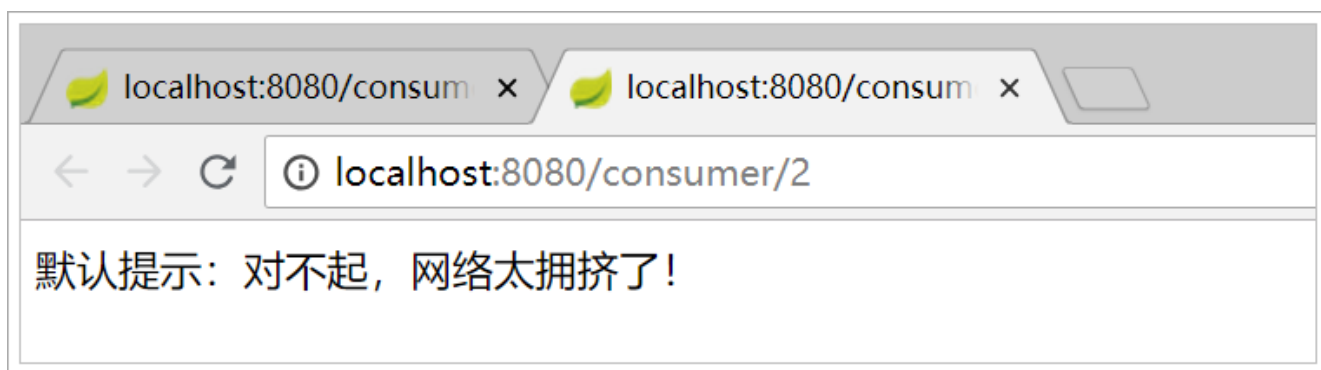
这样如果参数是id为1，一定失败，其它情况都成功。（不要忘了清空user-service中的休眠逻辑）

我们准备两个请求窗口：

- 一个请求：<http://localhost:8080/consumer/1>，注定失败
- 一个请求：<http://localhost:8080/consumer/2>，肯定成功

当我们疯狂访问id为1的请求时（超过20次），就会触发熔断。断路器会打开，一切请求都会被降级处理。

此时你访问id为2的请求，会发现返回的也是失败，而且失败时间很短，只有20毫秒左右；因进入半开状态之后2是可以的。



不过，默认的熔断触发要求较高，休眠时间窗较短，为了测试方便，我们可以通过配置修改熔断策略：

```

1 # 配置熔断策略:
2 hystrix:
3   command:
4     default:
5       circuitBreaker:
6         errorThresholdPercentage: 50 # 触发熔断错误比例阈值，默认值50%
7         sleepWindowInMilliseconds: 10000 # 熔断后休眠时长，默认值5秒
8         requestVolumeThreshold: 10 # 熔断触发最小请求次数，默认值是20
9       execution:
10        isolation:
11          thread:
12            timeoutInMilliseconds: 2000 # 熔断超时设置，默认为1秒

```

为了方便复制上述配置，可以使用如下格式复制到yaml文件中会自动格式化：

```
1 | hystrix.command.default.circuitBreaker.requestVolumeThreshold=10
2 | hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds=10000
3 | hystrix.command.default.circuitBreaker.errorThresholdPercentage=50
4 | hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=2000
```

上述的配置项可以参考 `HystrixCommandProperties` 类中。