

第1天 SpringCloud

学习目标

- 能够理解SpringCloud作用
 - 用来做微服务架构的技术解决方案
 - SpringCloud基于SpringBoot开发的， SpringCloud整合了很多优秀的第三方微服务开源框架
- 能够使用RestTemplate发送请求
 - 封装了基于Rest的Http请求
 - 可以实现Java对象序列化与反序列化
- ==能够搭建Eureka注册中心==
 - 用于管理服务、监控服务、服务路由
- 能够使用==Ribbon负载均衡==
 - 用来实现负载均衡(实现消费方负载均衡)
- 能够使用==Hystrix==熔断器
 - 做服务降级，防止程序发生雪崩

1 初识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，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；

1.1 目标

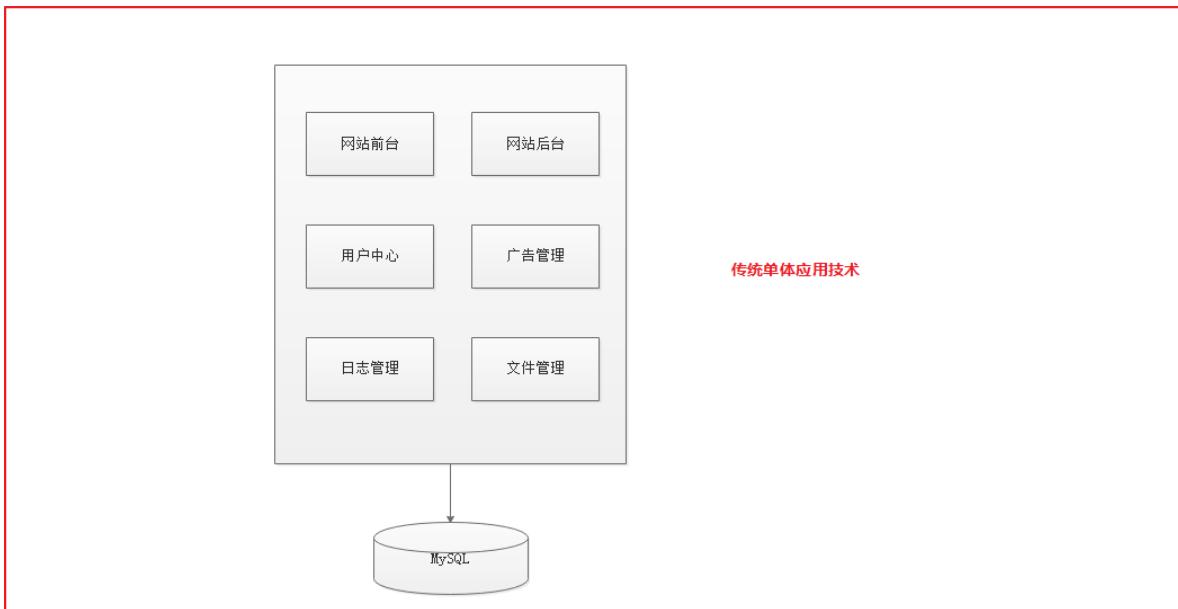
- 了解微服务架构
- 了解SpringCloud技术

1.2 讲解

1.2.1 技术架构演变

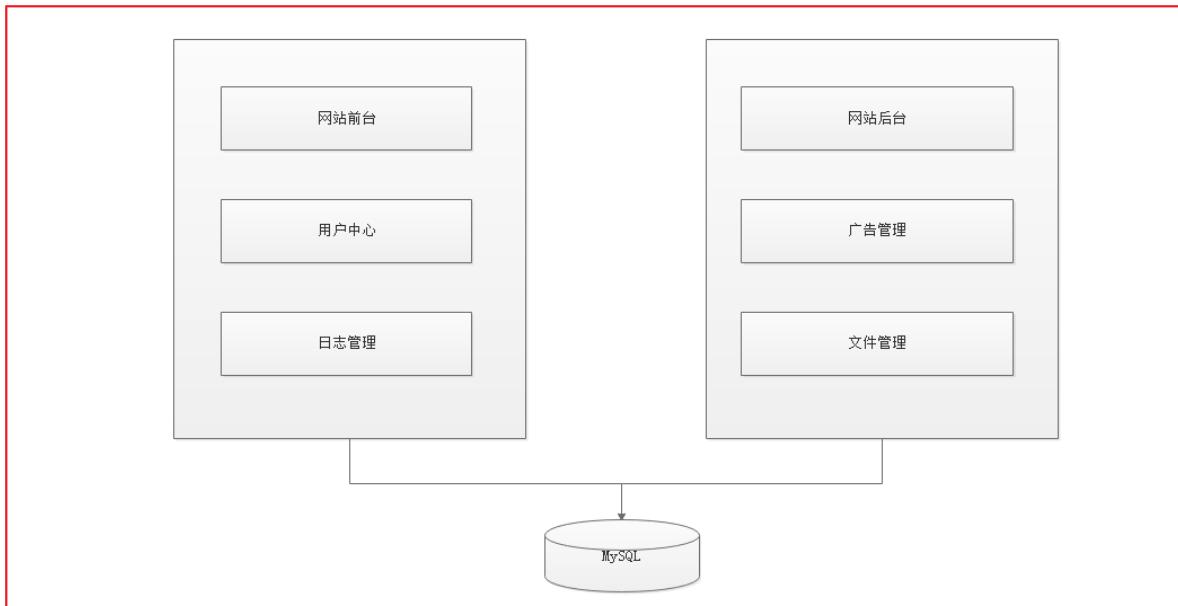
(1)单一应用架构

当网站流量很小时，只需要一个应用，所有功能部署在一起，减少部署节点成本的框架称之为集中式框架。此时，用于简化增删改查工作量的数据访问框架(ORM)是影响项目开发的关键。



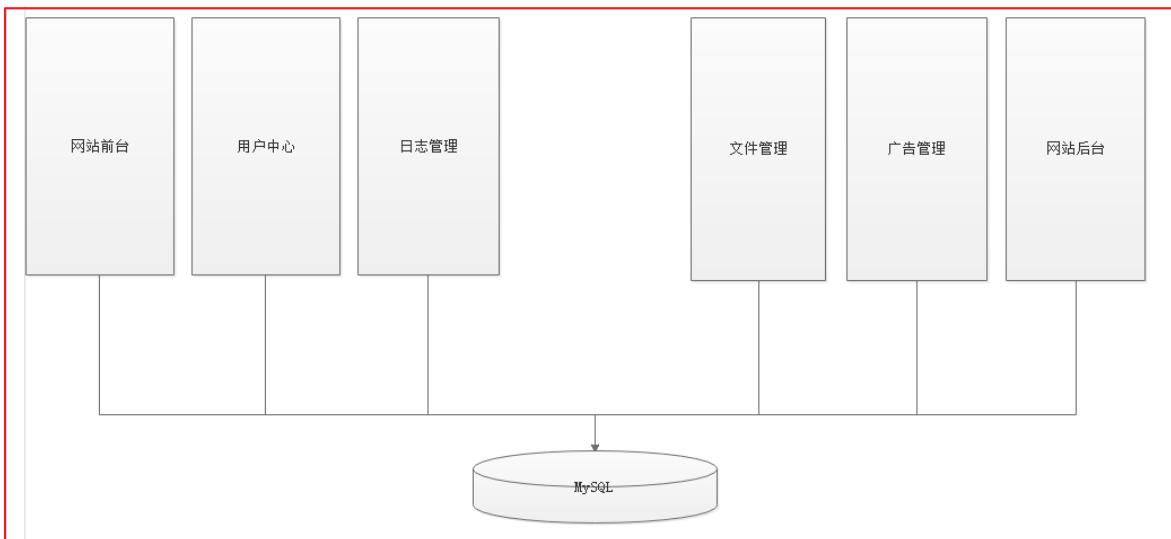
(2) 垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。



(3) 分布式服务架构

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。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。



(4)面向服务(SOA)架构

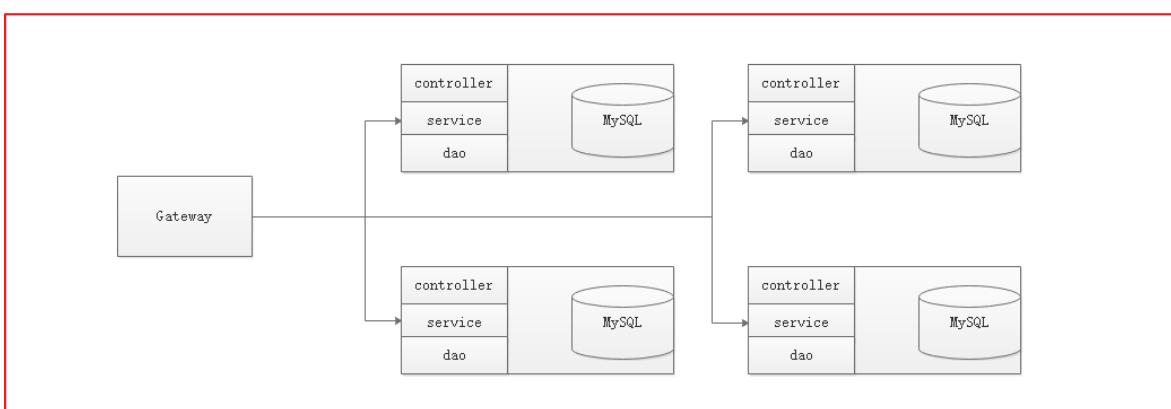
典型代表有两个：流动计算架构和微服务架构；

流动计算架构：

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。流动计算架构的最佳实践阿里的Dubbo。

微服务架构

与流动计算架构很相似，除了具备流动计算架构优势外，微服务架构中的微服务可以独立部署，独立发展。且微服务的开发不会限制于任何技术栈。微服务架构的最佳实践是SpringCloud。



1.2.2 SpringCloud简介

(1)SpringCloud介绍

Spring Boot擅长的是集成，把世界上最好的框架集成到自己项目中

==Spring Cloud本身也是基于SpringBoot开发而来，SpringCloud是一系列框架的有序集合,也是把非常流行的微服务的技术整合到一起，是属于微服务架构的一站式技术解决方案。==

Spring Cloud包含了：

注册中心：Eureka、consul、Zookeeper

负载均衡：Ribbon

熔断器：Hystrix

服务通信：Feign

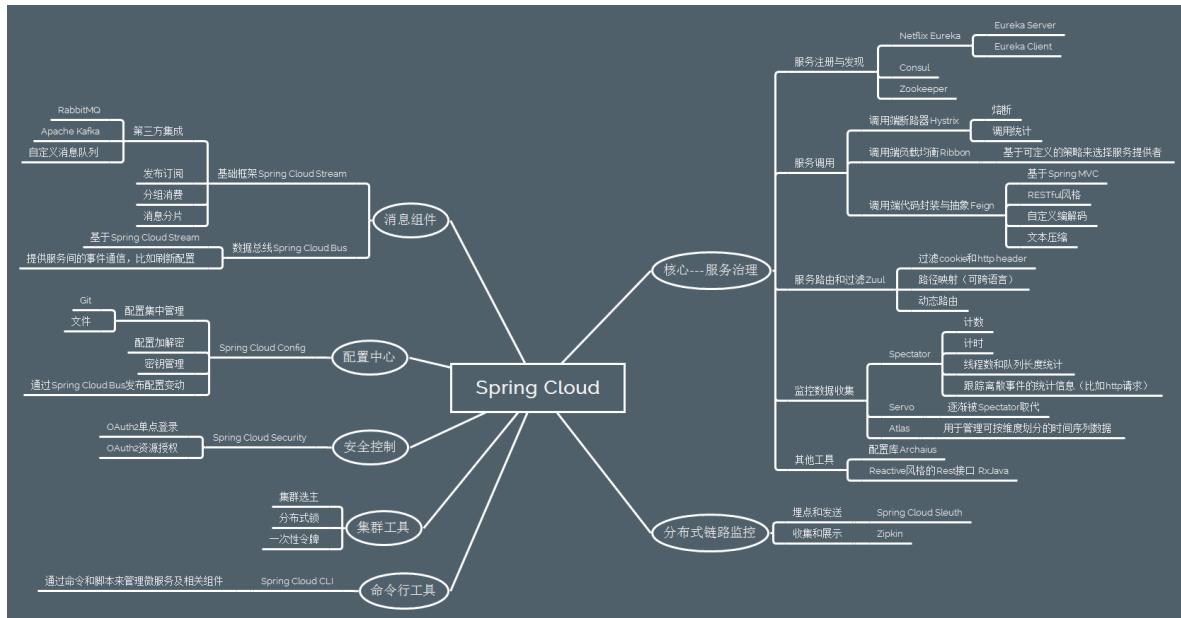
网关：Gateway

配置中心：config

消息总线：Bus

集群状态等等....功能。

Spring Cloud协调分布式环境中各个微服务，为各类服务提供支持。



(2)Spring Cloud的版本

The screenshot shows the official Spring Cloud project page. The left sidebar lists various projects under the 'Spring Cloud' category, including Spring Cloud Stream, Spring Cloud Azure, Spring Cloud for AWS, Spring Cloud Bus, Spring Cloud CLI, Spring Cloud for Cloud Foundry, Spring Cloud - Cloud Foundry Service Broker, Spring Cloud Cluster, and the main Spring Cloud project. The right side features the 'Learn' section with tabs for Overview, Learn, and Samples. Below this is a 'Documentation' section with a note about each project having its own documentation. A table lists the available versions for each project, with Greenwich SR2 being the current version for most.

Project	Version	Status	Reference Doc.	API Doc.
Greenwich SR2	CURRENT	GA	Reference Doc.	API Doc.
Hoxton	SNAPSHOT		Reference Doc.	API Doc.
Greenwich	SNAPSHOT		Reference Doc.	API Doc.
Finchley SR3	GA		Reference Doc.	API Doc.
Finchley	SNAPSHOT		Reference Doc.	API Doc.
Edgware SR6	GA		Reference Doc.	API Doc.
Edgware	SNAPSHOT		Reference Doc.	API Doc.
Dalston SR5	GA		Reference Doc.	API Doc.

版本说明：

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)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)

鉴于SpringBoot与SpringCloud关系，SpringBoot建议采用2.1.x版本

1.3 小结

- 微服务架构：就是将相关的功能独立出来，单独创建一个项目，并且连数据库也独立出来，单独创建对应的数据库。
- Spring Cloud本身也是基于SpringBoot开发而来，SpringCloud是一系列框架的有序集合，也是把非常流行的微服务的技术整合到一起。

2 服务调用方式

2.1 目标

- 理解RPC和HTTP的区别
- 能使用RestTemplate发送请求

2.2 讲解

2.2.1 RPC和HTTP

常见远程调用方式：

RPC:(Remote Produce Call)远程过程调用

- 1. 基于Socket
- 2. 自定义数据格式
- 3. 速度快，效率高
- 4. 典型应用代表：Dubbo, WebService, Elasticsearch集群间互相调用

HTTP：网络传输协议

- 1. 基于TCP/IP
- 2. 规定数据传输格式
- 3. 缺点是消息封装比较臃肿、传输速度比较慢
- 4. 优点是对服务提供和调用方式没有任何技术限定，自由灵活，更符合微服务理念

RPC和HTTP的区别：RPC是根据语言API来定义，而不是根据基于网络的应用来定义。

Http客户端工具

常见Http客户端工具：HttpClient、OKHttp、URLConnection。

2.2.2 Spring的RestTemplate

(1)RestTemplate介绍

- RestTemplate是Rest的HTTP客户端模板工具类
- 对基于Http的客户端进行封装
- 实现对象与JSON的序列化与反序列化
- 不限定客户端类型，目前常用的3种客户端都支持：HttpClient、OKHttp、JDK原生URLConnection(默认方式)

(2)RestTemplate入门案例



我们可以使用RestTemplate实现上图中的请求，`springcloud-day1-resttemplate`通过发送请求，请求`springcloud-day1-provider`的`/user/list`方法。

(1)搭建`springcloud-day1-provider`

这里不演示详细过程了，大家直接使用IDEA搭建一个普通的SpringBoot工程即可。

坐标

```
<groupId>com.itheima</groupId>
<artifactId>springcloud-day1-provider</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

pom.xml依赖

```
<!--父工程-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <!--web起步依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

创建 com.itheima.domain.User

```
public class User implements Serializable {
    private String name;
    private String address;
    private Integer age;

    public User() {
    }

    public User(String name, String address, Integer age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }

    //..get set tostring 略
}
```

application.properties

```
server.port=18081
```

创建 com.itheima.controller.UserController,代码如下:

```
@RestController
@RequestMapping(value = "/user")
public class UserController {

    /**
     * 提供服务
     * @return
     */
    @RequestMapping(value = "/list")
    public List<User> list(){
        List<User> users = new ArrayList<User>();
```

```

        users.add(new User("王五", "深圳", 25));
        users.add(new User("李四", "北京", 23));
        users.add(new User("赵六", "上海", 26));
        return users;
    }
}

```

创建启动类，并启动工程

```

@SpringBootApplication
public class SpringcloudDay1ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringcloudDay1ProviderApplication.class, args);
    }
}

```

访问：<http://localhost:18081/user/list> 效果如下：



(2) 创建 `springcloud-day1-resttemplate`

创建的详细过程也不讲解了，直接使用IDEA创建一个SpringBoot工程即可。

pom.xml依赖

```

<!--父工程-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <!--web起步依赖-->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    
```

```
</dependency>

<!--测试包-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>
```

创建启动类，并在启动类中创建RestTemplate对象

```
@SpringBootApplication
public class SpringcloudDay1ResttemplateApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringcloudDay1ResttemplateApplication.class,
args);
    }

    /**
     * @Bean:创建一个对象实例，并将对象交给Spring容器管理
     * <bean class="restTemplate"
class="org.springframework.web.client.RestTemplate" />
     * @return
     */
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

测试

在测试类HttpDemoApplicationTests中@Autowired注入RestTemplate

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

代码如下：

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringcloudDay1ResttemplateApplicationTests {
    @Autowired
    private RestTemplate restTemplate;

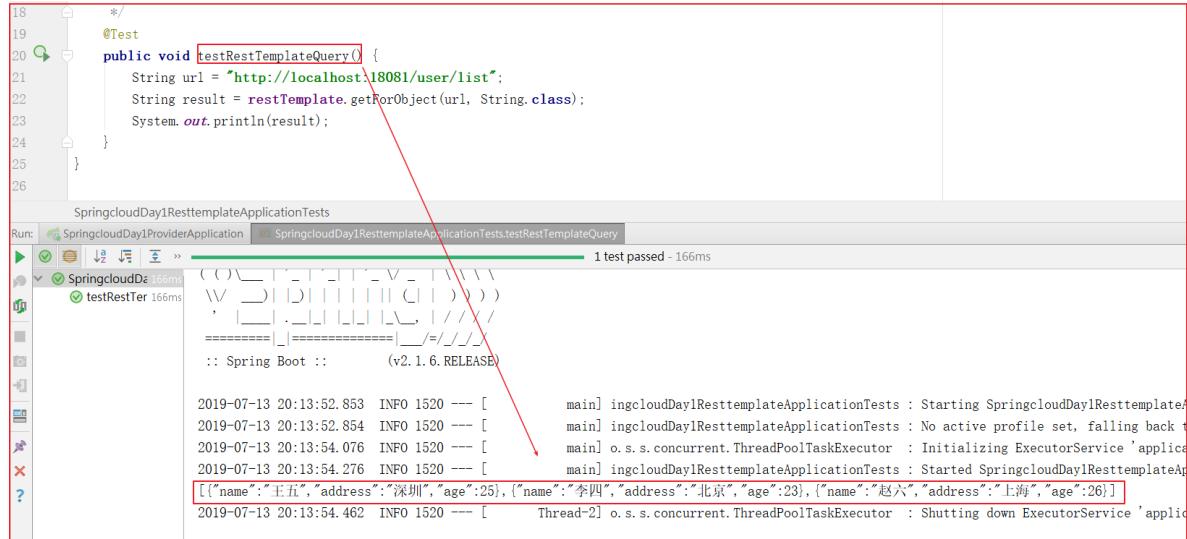
    /**
     * RestTemplate远程调用
     */
    @Test
```

```

public void testRestTemplateQuery() {
    String url = "http://localhost:18081/user/list";
    String result = restTemplate.getForObject(url, String.class);
    System.out.println(result);
}
}

```

运行测试方法，效果如下：



2.3 小结

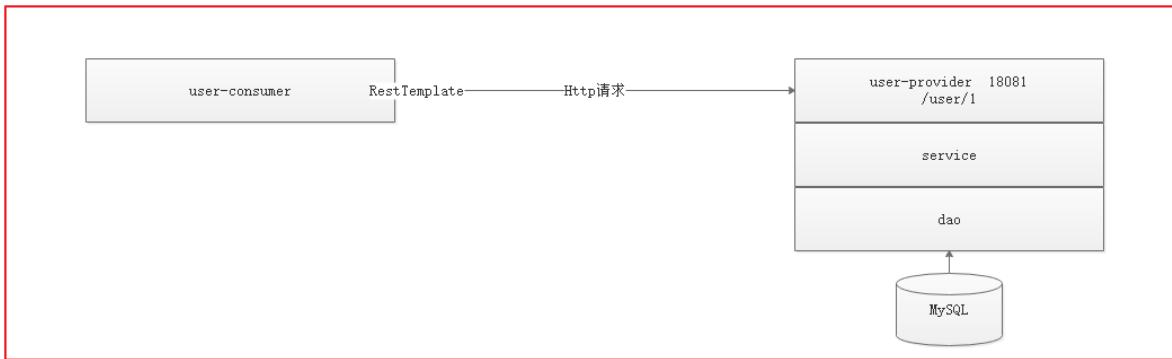
- RPC和HTTP的区别：RPC是根据语言API来定义，而不是根据基于网络的应用来定义。
- RestTemplate：
 - ①RestTemplate是Rest的HTTP客户端模板工具类。
 - ②对基于Http的客户端进行封装。
 - ③实现对象与JSON的序列化与反序列化。
 - ④不限定客户端类型

3 模拟微服务业务场景

模拟开发过程中的服务间关系。抽象出来，开发中的微服务之间的关系是生产者和消费者关系。

总目标：模拟一个最简单的服务调用场景，场景中保护微服务提供者(Producer)和微服务调用者(Consumer)，方便后面学习微服务架构

注意：实际开发中，每个微服务为一个独立的SpringBoot工程。



3.1 目标

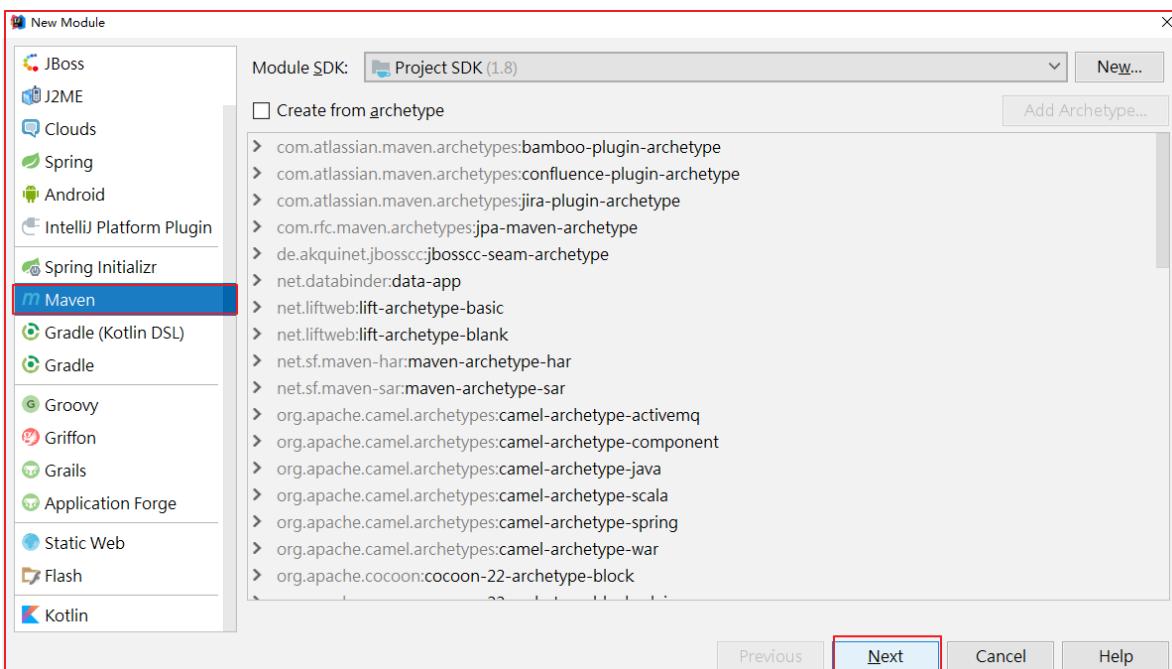
- 创建父工程
- 搭建服务提供者
- 搭建服务消费者
- 服务消费者使用RestTemplate调用服务提供者

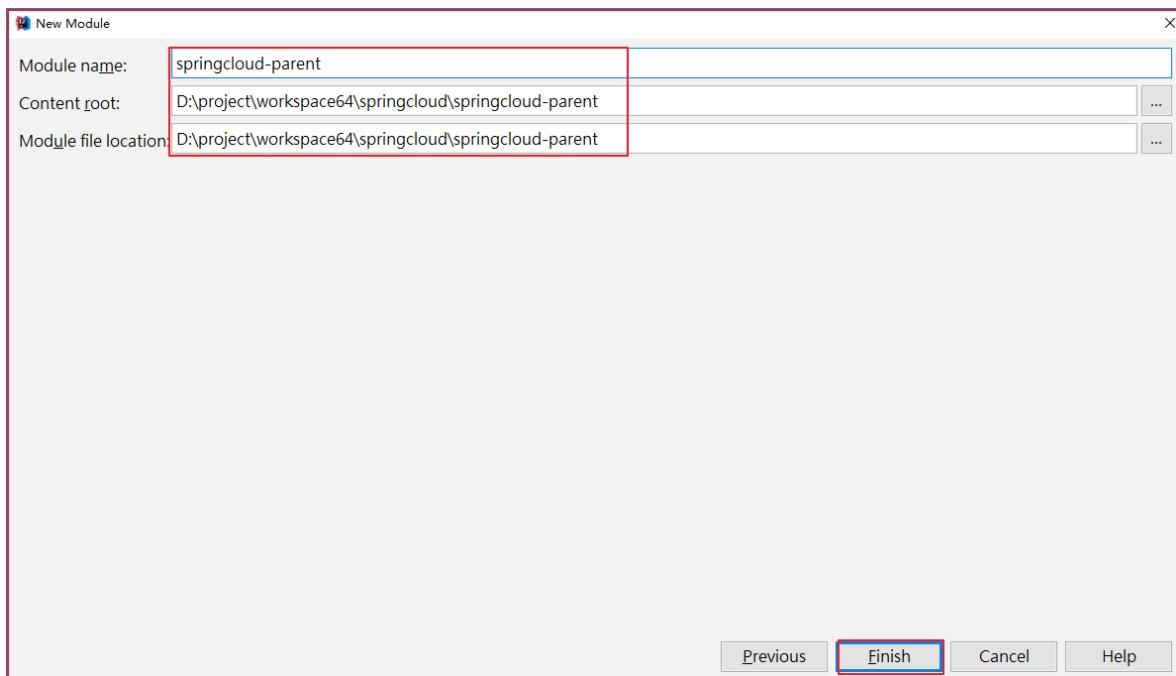
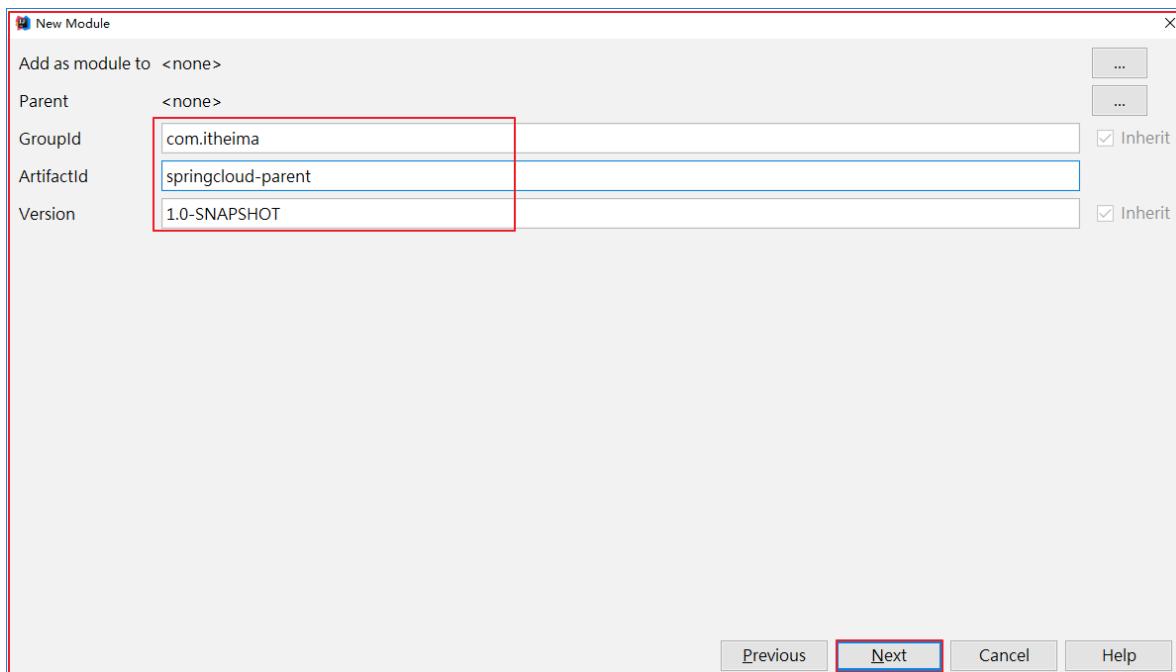
3.2 讲解

3.2.1 创建父工程

(1) 新建工程

新建一个Maven父工程 `springcloud-parent`, 创建步骤如下:





(2) 引入依赖

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

    <groupId>com.itheima</groupId>
    <artifactId>springcloud-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>user-provider</module>
        <module>user-consumer</module>
    </modules>
```

```

<!--父工程-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.6.RELEASE</version>
</parent>
<!--SpringCloud包依赖管理-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Greenwich.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
</project>

```

3.2.2 创建服务提供者(producer)工程

每个微服务工程都是独立的工程，连数据库都是独立的，所以我们一会要单独为该服务工程创建数据库。

工程创建步骤：

- 1.准备表结构
- 2.创建工程
- 3.引入依赖
- 4.创建Pojo，需要配置JPA的注解
- 5.创建Dao，需要继承JpaRepository<T, ID>
- 6.创建Service，并调用Dao
- 7.创建Controller，并调用Service
- 8.创建application.yml文件
- 9.创建启动类
- 10.测试

(1)建表

producer工程是一个独立的微服务，一般拥有独立的controller、service、dao、数据库，我们在springcloud数据库新建表结构信息，如下：

```

-- 使用springcloud数据库
USE springcloud;
-----
-- Table structure for tb_user
-----
CREATE TABLE `tb_user` (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `username` varchar(100) DEFAULT NULL COMMENT '用户名',

```

```

`password` varchar(100) DEFAULT NULL COMMENT '密码',
`name` varchar(100) DEFAULT NULL COMMENT '姓名',
`age` int(11) DEFAULT NULL COMMENT '年龄',
`sex` int(11) DEFAULT NULL COMMENT '性别, 1男, 2女',
`birthday` date DEFAULT NULL COMMENT '出生日期',
`created` date DEFAULT NULL COMMENT '创建时间',
`updated` date DEFAULT NULL COMMENT '更新时间',
`note` varchar(1000) DEFAULT NULL COMMENT '备注',
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8 COMMENT='用户信息表';

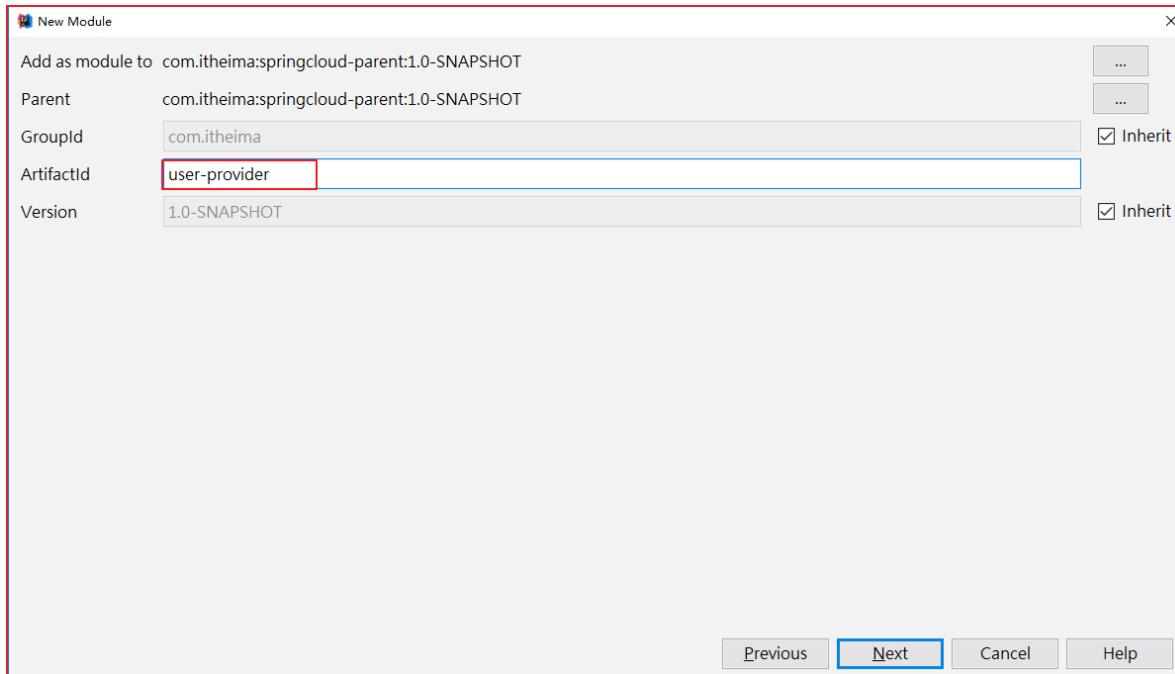
-- 
-- Records of tb_user
-- 

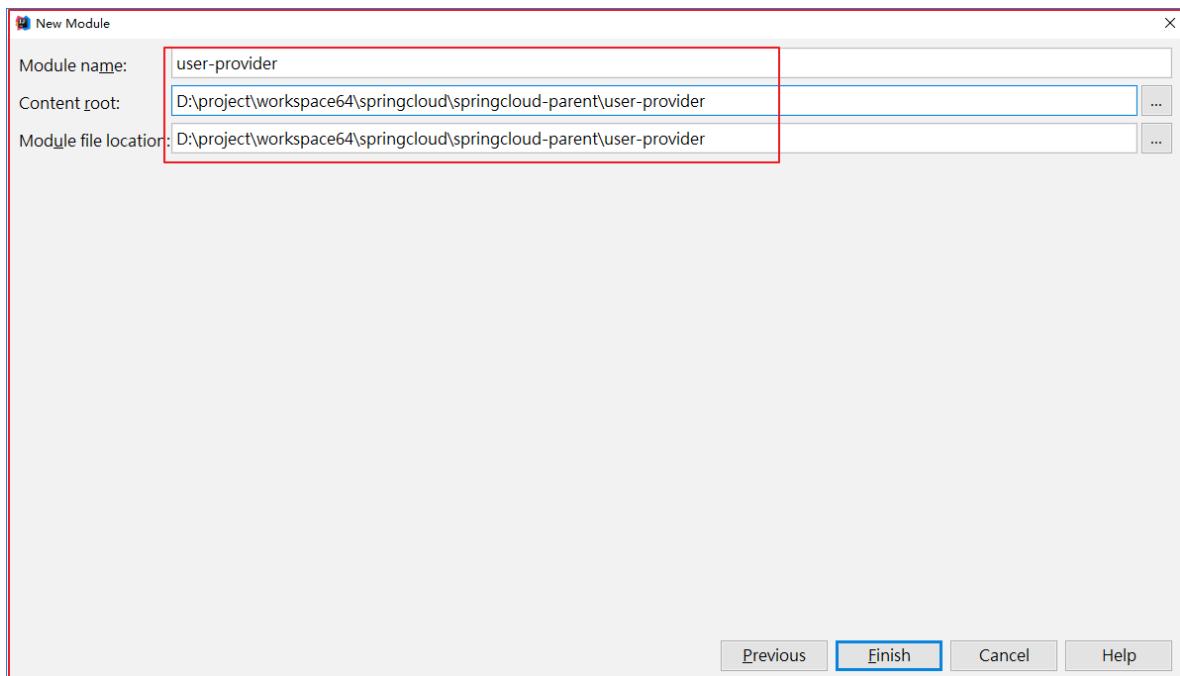
INSERT INTO `tb_user` VALUES ('1', 'zhangsan', '123456', '张三', '13', '1',
'2006-08-01', '2019-05-16', '2019-05-16', '张三');
INSERT INTO `tb_user` VALUES ('2', 'lisi', '123456', '李四', '13', '1', '2006-08-01',
'2019-05-16', '2019-05-16', '李四');

```

(2)新建user-provider工程

选中springcloud-parent工程->New Modul->Maven->输入坐标名字, 如下步骤:





引入pom.xml依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>springcloud-parent</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>user-provider</artifactId>

    <!--依赖包-->
    <dependencies>
        <!--JPA包-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!--web起步包-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!--MySQL驱动包-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <!--测试包-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-test</artifactId>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

(3)User对象创建

创建 `com.itheima.domain.User`，代码如下：

```
@Entity
@Table(name = "tb_user")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;//主键id
    private String username;//用户名
    private String password;//密码
    private String name;//姓名
    private Integer age;//年龄
    private Integer sex;//性别 1男性, 2女性
    private Date birthday; //出生日期
    private Date created; //创建时间
    private Date updated; //更新时间
    private String note;//备注

    //...set get toString 略
}
```

(4)dao

创建 `com.itheima.dao.UserDao`，代码如下：

```
public interface UserDao extends JpaRepository<User, Integer> { }
```

(5)Service层

创建 `com.itheima.service.UserService` 接口，代码如下：

```
public interface UserService {
    /**
     * 根据ID查询用户信息
     * @param id
     * @return
     */
    User findByUserId(Integer id);
}
```

创建 `com.itheima.service.impl.UserServiceImpl` 代码如下：

```

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    /**
     * 根据ID查询用户信息
     * @param id
     * @return
     */
    @Override
    public User findById(Integer id) {
        return userDao.findById(id).get();
    }
}

```

(6)控制层

创建 com.itheima.controller.UserController，代码如下：

```

@RestController
@RequestMapping(value = "/user")
public class UserController {

    @Autowired
    private UserService userService;

    /**
     * 根据ID查询用户信息
     * @param id
     * @return
     */
    @RequestMapping(value = "/find/{id}")
    public User findById(@PathVariable(value = "id") Integer id){
        return userService.findById(id);
    }
}

```

(7)application.yml配置

```

server:
  port: 18081
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: itcast
    url: jdbc:mysql://127.0.0.1:3306/springcloud?
    useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC

```

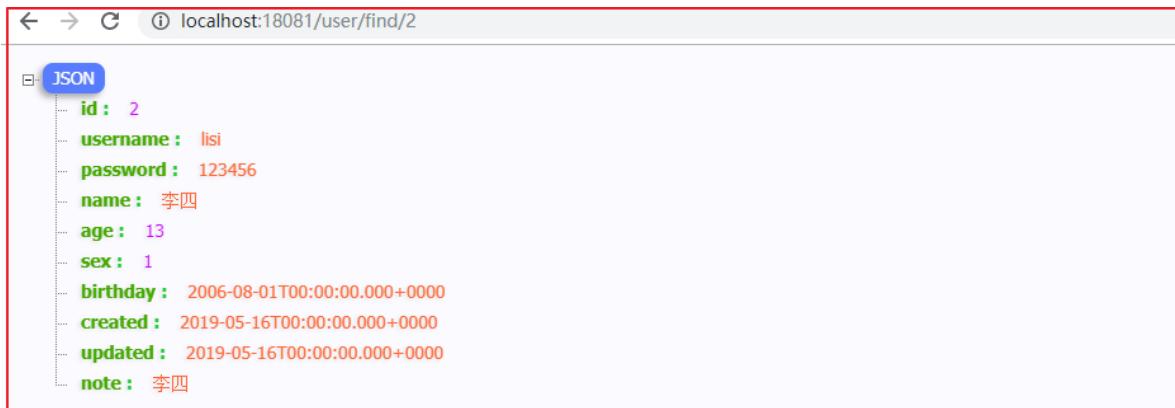
(8)启动类创建

创建 `com.itheima.UserProviderApplication` 启动类，并启动

```
@SpringBootApplication
public class UserProviderApplication {

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

测试：`<http://localhost:18081/user/find/2>`



3.2.3 创建服务消费者(consumer)工程

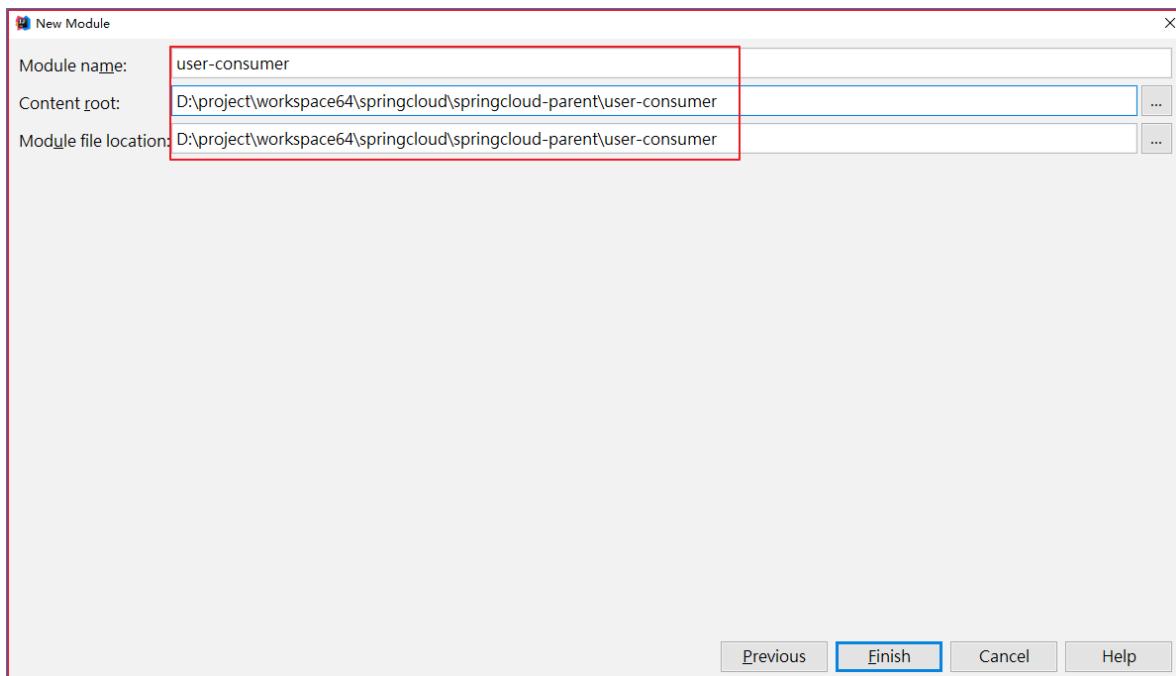
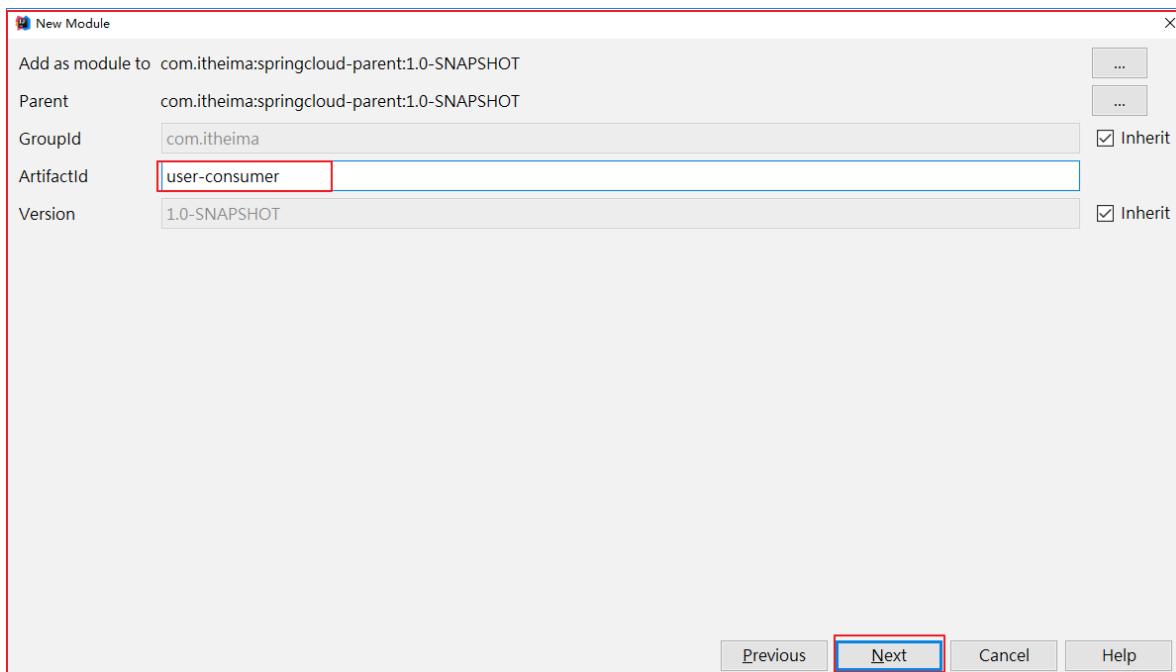
在该工程中使用`RestTemplate`来调用`user-provider`微服务。

实现步骤：

1. 创建工程
2. 引入依赖
3. 创建Pojo
4. 创建启动类，同时创建`RestTemplate`对象，并交给`SpringIOC`容器管理
5. 创建`application.yml`文件，指定端口
6. 编写`Controller`，在`Controller`中通过`RestTemplate`调用`user-provider`的服务
7. 启动测试

(1)工程搭建

选中`springcloud-parent`工程->`New Modul`->`Maven`->输入坐标名字，如下步骤：



pom.xml依赖如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>springcloud-parent</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>user-consumer</artifactId>

    <!--依赖包-->
    <dependencies>
```

```
<!--web起步依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>
</project>
```

(2)创建User对象

在src下创建 com.itheima.domain.User,代码如下:

```
public class User {
    private Integer id;//主键id
    private String username;//用户名
    private String password;//密码
    private String name;//姓名
    private Integer age;//年龄
    private Integer sex;//性别 1男性, 2女性
    private Date birthday; //出生日期
    private Date created; //创建时间
    private Date updated; //更新时间
    private String note;//备注

    //...set、get、toString 略
}
```

(3)创建启动引导类

在src下创建 com.itheima.UserConsumerApplication,代码如下:

```
@SpringBootApplication
public class UserConsumerApplication {

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

    /**
     * 将RestTemplate的实例放到spring容器中
     * @return
     */
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

创建application.yml,并配置端口为18082

```
server:  
port: 18082
```

(4) 创建控制层，在控制层中调用user-provider

在src下创建 `com.itheima.controller UserController`，代码如下：

```
@RestController  
@RequestMapping(value = "/consumer")  
public class UserController {  
  
    @Autowired  
    private RestTemplate restTemplate;  
  
    /**/  
     * 在user-consumer服务中通过RestTemplate调用user-provider服务  
     * @param id  
     * @return  
     */  
    @GetMapping(value = "/{id}")  
    public User queryById(@PathVariable(value = "id") Integer id){  
        String url = "http://localhost:18081/user/find/" + id;  
        return restTemplate.getForObject(url, User.class);  
    }  
}
```

启动测试：

请求地址：`<http://localhost:18082/consumer/1>`



3.2.4 思考问题

user-provider：对外提供用户查询接口

user-consumer：通过RestTemplate访问接口查询用户数据

存在的问题：

1. 在服务消费者中，我们把url地址硬编码到代码中，不方便后期维护
2. 在服务消费者中，不清楚服务提供者的状态(user-provider有可能没有宕机了)

3. 服务提供者只有一个服务，即便服务提供者形成集群，服务消费者还需要自己实现负载均衡
4. 服务提供者的如果出现故障，是否能够及时发现：

其实上面说的问题，概括一下就是微服务架构必然要面临的问题

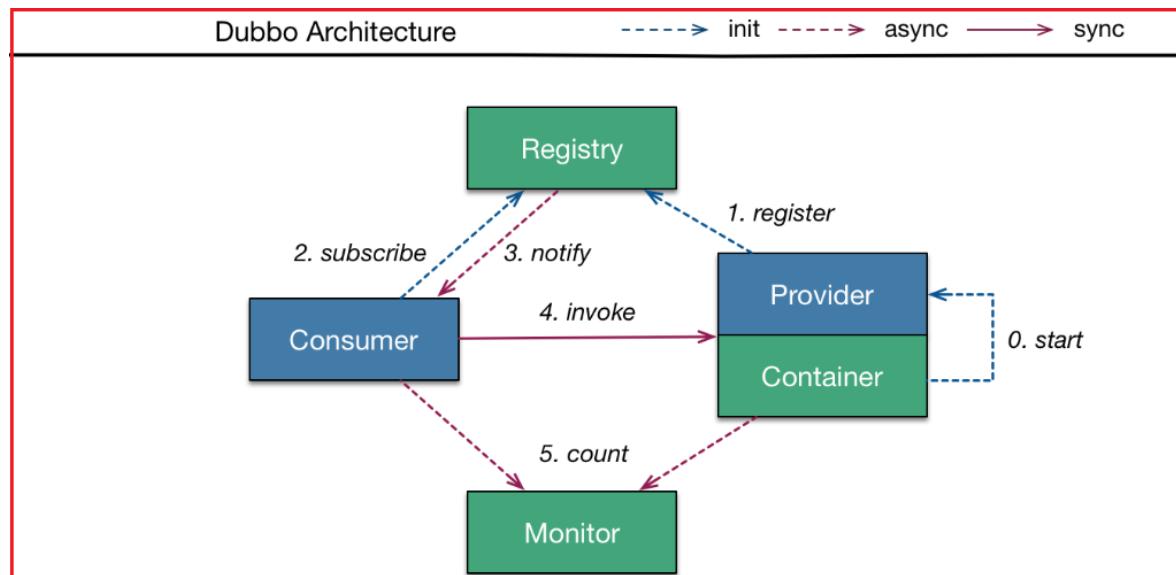
- 服务管理：自动注册与发现、状态监管
- 服务负载均衡
- 熔断器

3.3 小结

- 服务消费者使用RestTemplate调用服务提供者, 使用RestTemplate调用的时候，需要先创建并注入到SpringIOC容器中
- 在服务消费者中，我们把url地址硬编码到代码中，不方便后期维护
- 在服务消费者中，不清楚服务提供者的状态(user-provider有可能没有宕机了)
- 服务提供者只有一个服务，即便服务提供者形成集群，服务消费者还需要自己实现负载均衡
- 服务提供者的如果出现故障，不能及时发现。

4 注册中心 Spring Cloud Eureka

前面我们学过Dubbo，关于Dubbo的执行过程我们看如下图片：



执行过程：

1. Provider: 服务提供者，异步将自身信息注册到Register（注册中心）
2. Consumer: 服务消费者，异步去Register中拉取服务数据
3. Register异步推送服务数据给Consumer，如果有新的服务注册了，Consumer可以直接监控到新的服务
4. Consumer同步调用Provider
5. Consumer和Provider异步将调用频率信息发给Monitor监控

4.1 目标

- 理解Eureka的原理图

- 能实现Eureka服务的搭建
- 能实现服务提供者向Eureka注册服务
- 能实现服务消费者向Eureka注册服务
- 能实现消费者通过Eureka访问服务提供者
- 能掌握Eureka的详细配置

4.2 讲解

4.2.1 Eureka 简介

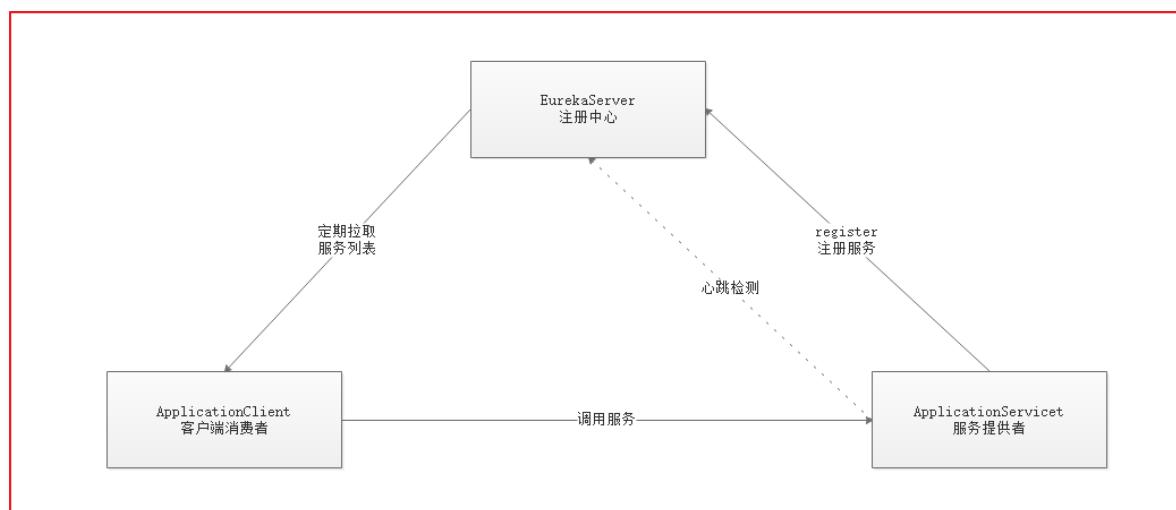
Eureka解决了第一个问题：服务的管理，注册和发现、状态监管、动态路由。

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；

4.2.2 原理图

基本架构图



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

服务提供者：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：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

4.2.3 入门案例

目标：搭建Eureka Server环境，创建一个eureka_server工程。

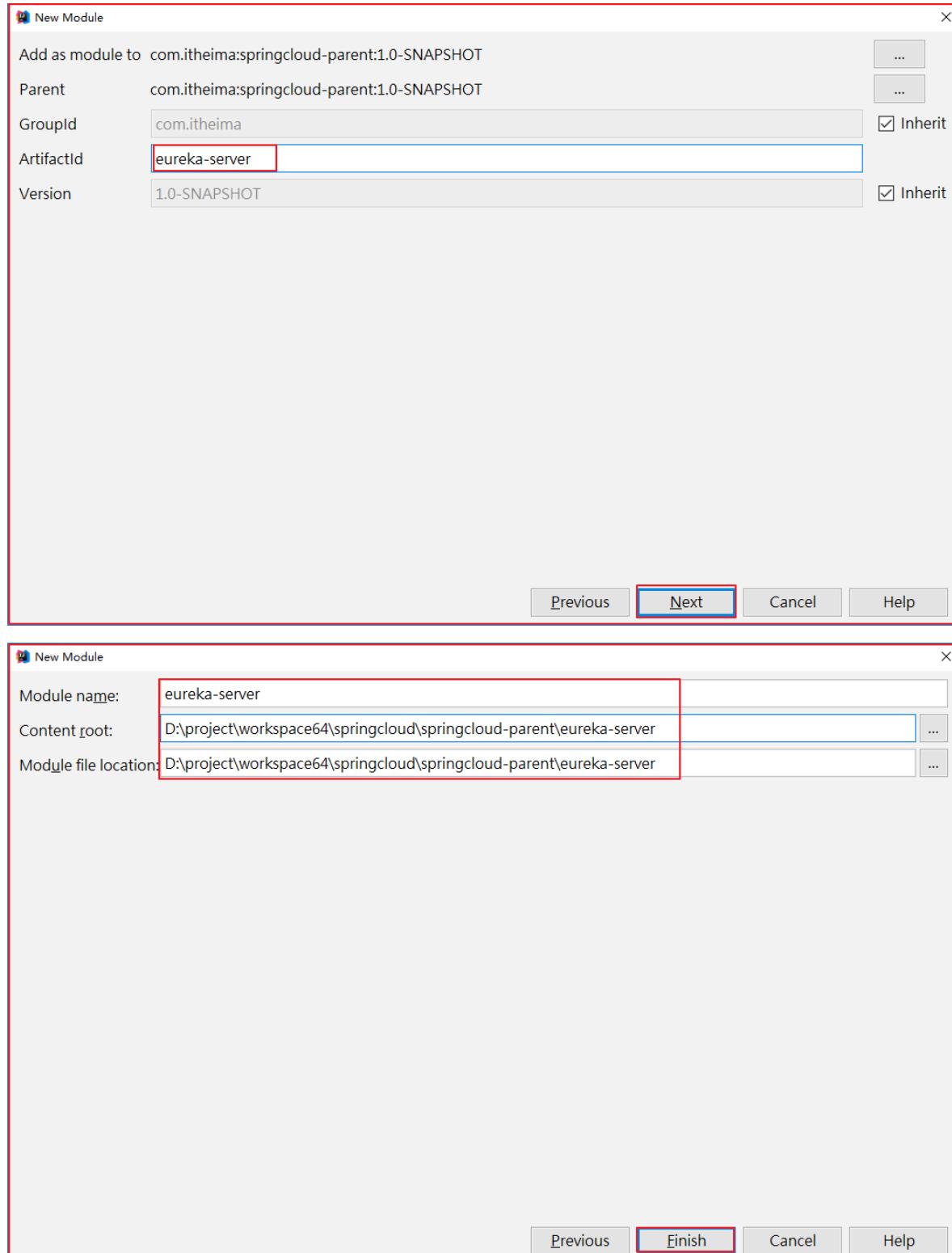
步骤：分三步

- 1: eureka-server搭建工程eureka-server
- 2: 服务提供者-注册服务， user-provider工程
- 3: 服务消费者-发现服务， user-consumer工程

4.2.3.1 搭建eureka-server工程

(1)工程搭建

选中springcloud-parent工程->New Modul->Maven->输入坐标名字，如下步骤：



(2)pom.xml引入依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<parent>
    <artifactId>springcloud-parent</artifactId>
    <groupId>com.itheima</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>
<artifactId>eureka-server</artifactId>

<!--依赖包-->
<dependencies>
    <!--eureka-server依赖-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    </dependency>
</dependencies>
</project>

```

(3)application.yml配置

```

server:
  port: 7001      #端口号
spring:
  application:
    name: eureka-server # 应用名称，会在Eureka中作为服务的id标识（serviceId）
eureka:
  client:
    register-with-eureka: false    #是否将自己注册到Eureka中
    fetch-registry: false        #是否从eureka中获取服务信息
    service-url:
      defaultzone: http://localhost:7001/eureka # EurekaServer的地址

```

(4)启动类创建

在src下创建 `com.itheima.EurekaServerApplication`, 在类上需要添加一个注解 `@EnableEurekaServer` , 用于开启Eureka服务, 代码如下:

```

@SpringBootApplication
@EnableEurekaServer //开启Eureka服务
public class EurekaServerApplication {

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

```

(5)启动访问

启动后, 访问 `<http://127.0.0.1:7001/>` , 效果如下:

The screenshot shows the Spring Eureka dashboard at 127.0.0.1:7001. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main sections are:

- System Status**: Displays environment (test), data center (default), current time (2019-07-14T16:35:58 +0800), uptime (00:00), lease expiration enabled (false), renew threshold (1), and renew count (0).
- DS Replicas**: Shows a single instance at localhost.
- Instances currently registered with Eureka**: A table with columns Application, AMIs, Availability Zones, and Status. It shows "No instances available".
- General Info**: A table with columns Name and Value. It includes entries for total-avail-memory (438mb), environment (test), and num-of-cpus (12).

4.2.3.2 服务提供者-注册服务

我们的user-provider属于服务提供者，需要在user-provider工程中引入Eureka客户端依赖，然后在配置文件中指定Eureka服务地址，然后在启动类中开启Eureka服务发现功能。

步骤：

1. 引入eureka客户端依赖包
2. 在application.yml中配置Eureka服务地址
3. 在启动类上添加@EnabledDiscoveryClient或者@EnableEurekaClient

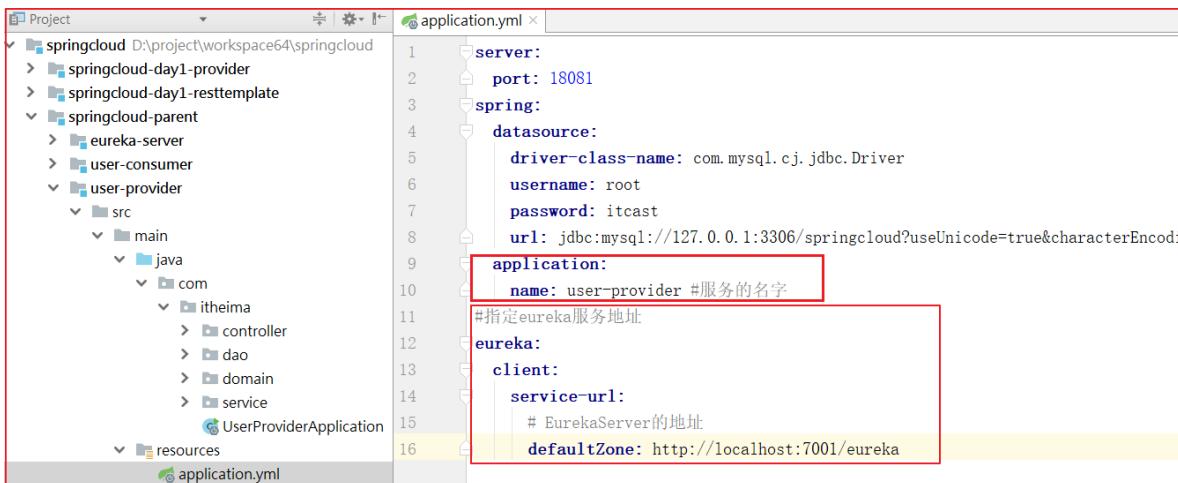
(1)引入依赖

在user-provider的pom.xml中引入如下依赖

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

(2)配置Eureka服务地址

修改user-provider的application.yml配置文件，添加Eureka服务地址，代码如下：



上图代码如下：

```

server:
  port: 18081
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: itcast
    url: jdbc:mysql://127.0.0.1:3306/springcloud?
      useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
  application:
    name: user-provider #服务的名字,不同的应用,名字不同,如果是集群,名字需要相同
  #指定eureka服务地址
eureka:
  client:
    service-url:
      # EurekaServer的地址
    defaultZone: http://localhost:7001/eureka

```

(3)开启Eureka客户端发现功能

在user-provider的启动类 `com.itheima.UserProviderApplication` 上添加 `@EnableDiscoveryClient` 注解或者 `@EnableEurekaClient`，用于开启客户端发现功能。

```

@SpringBootApplication
@EnableDiscoveryClient //开启Eureka客户端发现功能
@EnableEurekaClient //开启Eureka客户端发现功能,注册中心只能是Eureka
public class UserProviderApplication {

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

```

区别：

`@EnableDiscoveryClient` 和 `@EnableEurekaClient` 都用于开启客户端的发现功能，但 `@EnableEurekaClient` 的注册中心只能是Eureka。

(4)启动测试

启动eureka-server，再启动user-provider。

访问Eureka地址 <http://127.0.0.1:7001/>，效果如下：

The screenshot shows the Spring Eureka dashboard at `http://127.0.0.1:7001`. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into sections: System Status and DS Replicas.

System Status: Displays environment information (Environment: test, Data center: default), current time (2019-07-14T17:46:23 +0800), uptime (00:00), lease expiration enabled (false), renew threshold (3), and renew counts (0).

DS Replicas: Shows instances currently registered with Eureka. A red arrow points from the text "spring.application.name" to the "Application" column of the table. The table has columns: Application, AMIs, Availability Zones, and Status. One instance is listed: USER-PROVIDER with n/a (1) AMIs, (1) Availability Zones, and Status UP (1) - DESKTOP-79U5AQM:user-provider:18081.

4.2.3.3 服务消费者-注册服务中心

消费方添加Eureka服务注册和生产方配置流程一致。

步骤：

1. 引入eureka客户端依赖包
2. 在application.yml中配置Eureka服务地址
3. 在启动类上添加@EnabledDiscoveryClient或者@EnableEurekaClient

(1)pom.xml引入依赖

修改user-consumer的pom.xml引入如下依赖

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

(2)application.yml中配置eureka服务地址

修改user-consumer工程的application.yml配置，添加eureka服务地址，配置如下：

```

Project: user-consumer
File: application.yml

server:
  port: 18082
spring:
  application:
    name: user-consumer #服务名字
#指定eureka服务地址
eureka:
  client:
    service-url:
      # EurekaServer的地址
      defaultZone: http://localhost:7001/eureka

```

上图配置如下：

```

server:
  port: 18082
spring:
  application:
    name: user-consumer #服务名字
#指定eureka服务地址
eureka:
  client:
    service-url:
      # EurekaServer的地址
      defaultZone: http://localhost:7001/eureka

```

(3)在启动类上开启Eureka服务发现功能

修改user-consumer的 `com.itheima.UserConsumerApplication` 启动类，在类上添加 `@EnableDiscoveryClient` 注解，代码如下：

```

Project: user-consumer
File: UserConsumerApplication.java

@SpringBootApplication
@EnableDiscoveryClient //开启Eureka客户端发现功能
public class UserConsumerApplication {
  public static void main(String[] args) {
    SpringApplication.run(UserConsumerApplication.class, args);
  }
  /**
   * 将RestTemplate的实例放到Spring容器中
   * @return
   */
  @Bean
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }
}

```

上图代码如下：

```

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

  //...略
}

```

(4)测试

启动user-consumer，然后访问Eureka服务地址 <http://127.0.0.1:7001/> 效果如下：

The screenshot shows the Eureka service registration interface. At the top, it says 'DS Replicas' and 'localhost'. Below that, it lists 'Instances currently registered with Eureka'.

Application	AMIs	Availability Zones	Status
USER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-79U5AQM:user-consumer:18082
USER-PROVIDER	n/a (1)	(1)	UP (1) - DESKTOP-79U5AQM:user-provider:18081

4.2.3.4 消费者通过Eureka访问提供者

之前消费者 user-consumer 访问服务提供者 user-provider 是通过 `http://localhost:18081/user/find/1` 访问的，这里是具体的路径，没有从Eureka获取访问地址，我们可以让消费者从Eureka那里获取服务提供者的访问地址，然后访问服务提供者。

修改user-consumer的 `com.itheima.controller.UserController`，代码如下：

```
@RestController
@RequestMapping(value = "/consumer")
public class UserController {

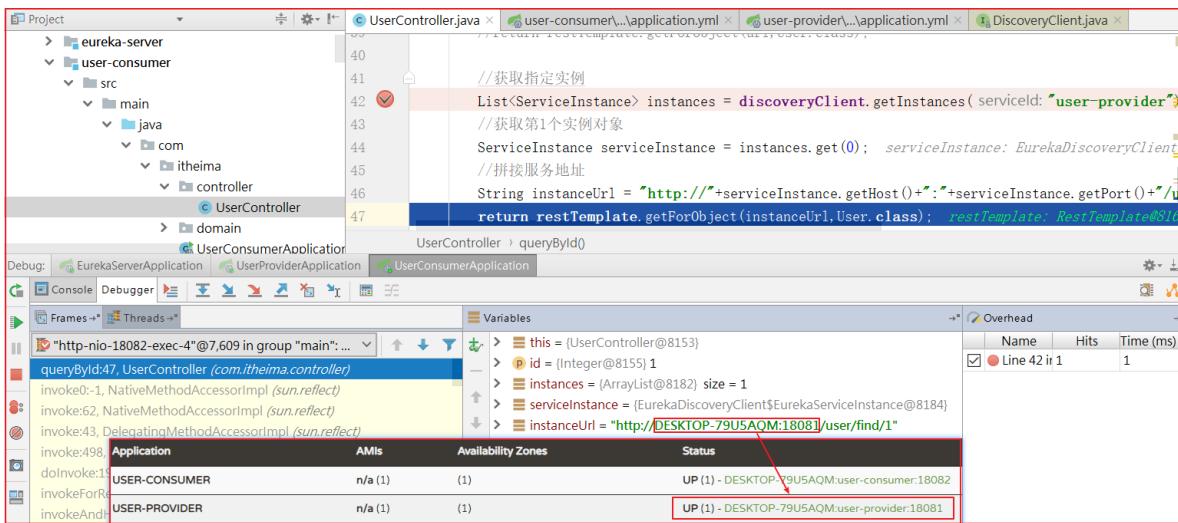
    @Autowired
    private RestTemplate restTemplate;

    //可以用来发现服务
    @Autowired
    private DiscoveryClient discoveryClient; 注入该对象，可用来发现当前注册中心中的服务对象

    *****
    * 在user-consumer服务中通过RestTemplate调用user-provider服务
    * @param id
    * @return
    */
    @GetMapping(value = "/{id}")
    public User queryById(@PathVariable(value = "id") Integer id) {
        //String url = "http://localhost:18081/user/find/" + id; 原来的配置，注释掉
        //return restTemplate.getForObject(url, User.class);

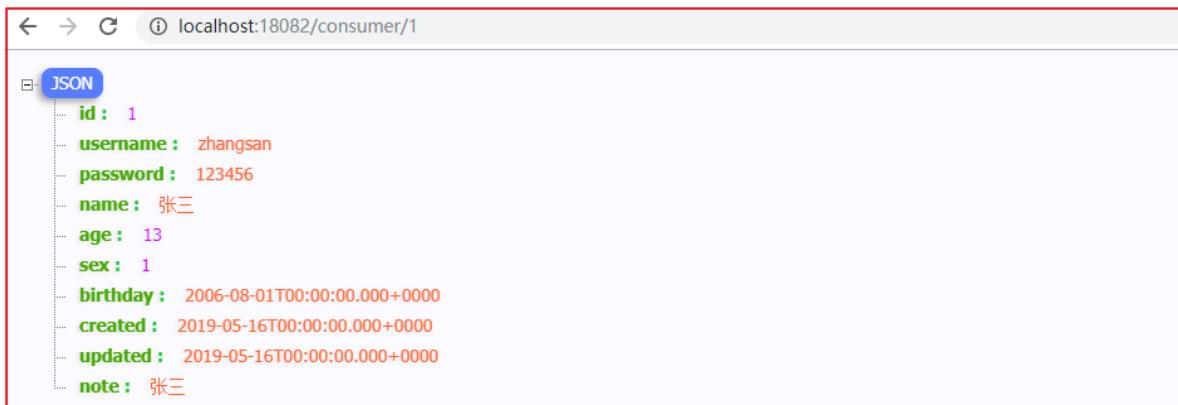
        //获取指定实例
        List<ServiceInstance> instances = discoveryClient.getInstances(servicename: "user-provider");
        //获取第1个实例对象
        ServiceInstance serviceInstance = instances.get(0);
        //拼接服务地址
        String instanceUrl = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/find/" + id; 服务的主机 服务的端口号
        return restTemplate.getForObject(instanceUrl, User.class);
    }
}
```

Debug跟踪运行，访问 <http://localhost:18082/consumer/1>，效果如下：



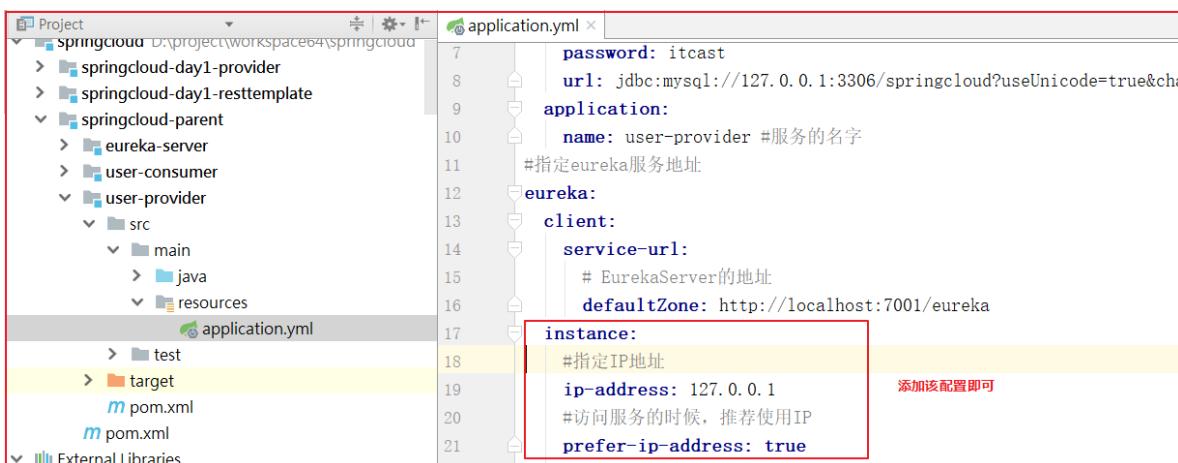
跟踪运行后，我们发现，这里的地址就是服务注册中的状态名字。

浏览器结果如下：



(2) 使用IP访问配置

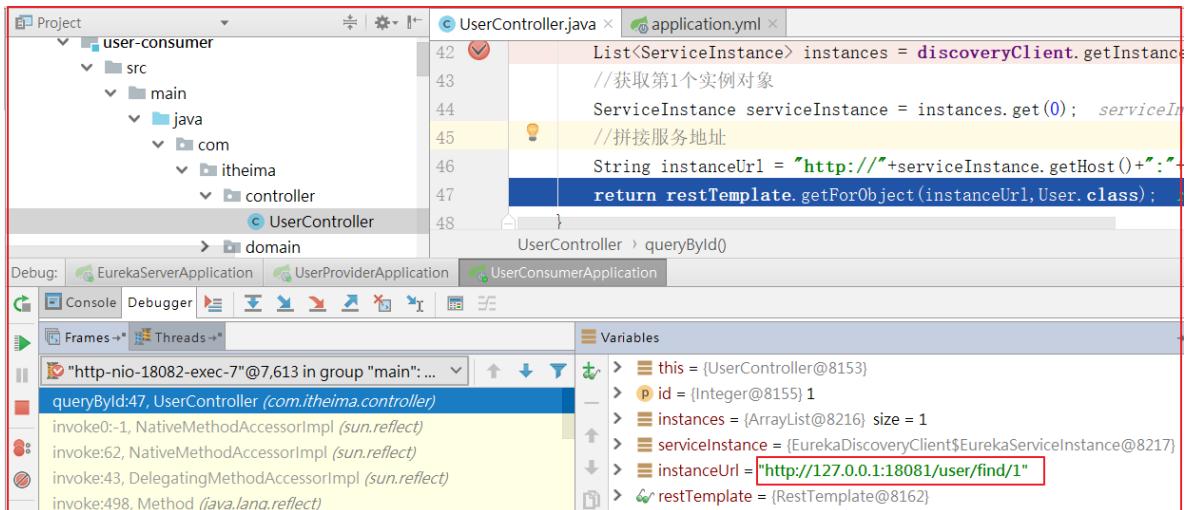
上面的请求地址是服务状态名字，其实也是当前主机的名字，可以通过配置文件，将它换成IP，修改 application.yml 配置文件，代码如下：



上图配置如下：

```
instance:
#指定IP地址
ip-address: 127.0.0.1
#访问服务的时候，推荐使用IP
prefer-ip-address: true
```

重新启动 user-provider，并再次测试，测试效果如下：



4.2.4 Eureka详解

4.2.4.1 基础架构

Eureka架构中的三个核心角色

1. 服务注册中心：Eureka服务端应用，提供服务注册发现功能，**eureka-server**
2. 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：**user-provider**
3. 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，**user-consumer**

4.2.4.2 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：

1. 当我们开启了客户端发现注解@**DiscoveryClient**。同时导入了**eureka-client**依赖坐标
2. 同时配置Eureka服务注册中心地址在配置文件中
3. 服务在启动时，检测是否有@**DiscoveryClient**注解和配置信息
4. 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(IP、端口等)
5. Eureka注册中心会把服务的信息保存在**Map**中。

服务续约：

服务注册完成以后，服务提供者会维持一个**心跳**，保存服务处于存在状态。这个称之为服务续约(renew)。

```

Project
└ springcloud D:\project\workspace64\springcloud
  └ springcloud-parent
    └ user-provider
      └ src
        └ main
          └ resources
            └ application.yml
  └ External Libraries

user-provider\...\application.yml
10   name: user-provider #服务的名字
11   #指定eureka服务地址
12   eureka:
13     client:
14       service-url:
15         # EurekaServer的地址
16         defaultZone: http://localhost:7001/eureka
17   instance:
18     #指定IP地址
19     ip-address: 127.0.0.1
20     #访问服务的时候，推荐使用IP
21     prefer-ip-address: true
22     #租约到期，服务时效时间，默认值90秒
23     lease-expiration-duration-in-seconds: 150
24     #租约续约间隔时间，默认30秒
25     lease-renewal-interval-in-seconds: 30

```

上图配置如下：

```

#租约到期，服务时效时间，默认值90秒
lease-expiration-duration-in-seconds: 150
#租约续约间隔时间， 默认30秒
lease-renewal-interval-in-seconds: 30

```

参数说明：

- 两个参数可以修改服务续约行为
 - `lease-renewal-interval-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 [前提是`EurekaServer`关闭了自我保护]

获取服务列表：

```

Project
└ springcloud D:\project\workspace64\springcloud
  └ springcloud-parent
    └ user-provider
      └ src
        └ main
          └ resources
            └ application.yml
  └ External Libraries

application.yml
8   url: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterSetResults=utf8
9   application:
10    name: user-provider #服务的名字
11    #指定eureka服务地址
12    eureka:
13      client:
14        service-url:
15          # EurekaServer的地址
16          defaultZone: http://localhost:7001/eureka
17          #每隔30秒获取服务中心列表，(只读备份)
18          registry-fetch-interval-seconds: 30
19      instance:
20        #指定IP地址
21        ip-address: 127.0.0.1
22        #访问服务的时候，推荐使用IP
23        prefer-ip-address: true
24        #租约到期，服务时效时间，默认值90秒
25        lease-expiration-duration-in-seconds: 150
26        #租约续约间隔时间， 默认30秒
27        lease-renewal-interval-in-seconds: 30

```

上图配置如下：

```
registry-fetch-interval-seconds: 30
```

说明：

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

4.2.4.3 失效剔除和自我保护

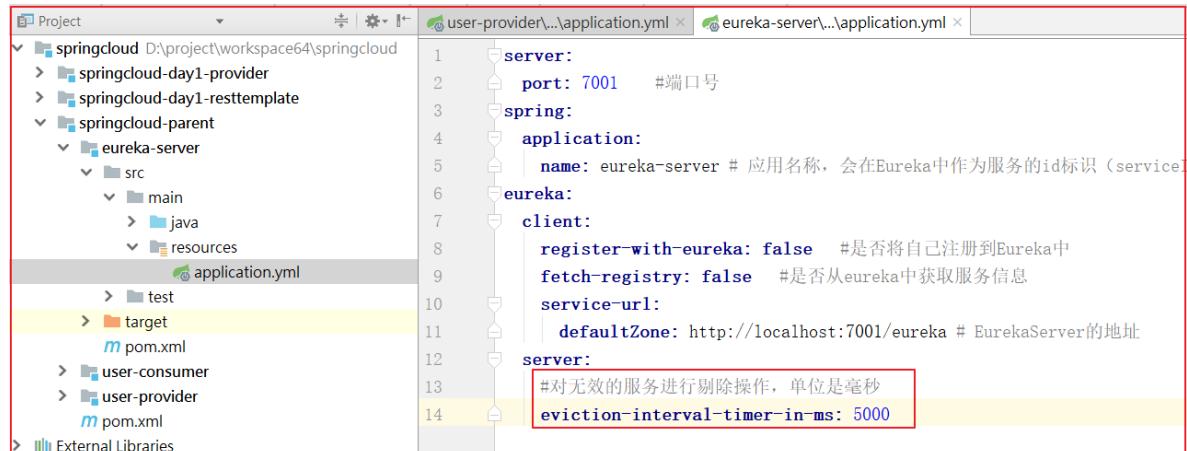
服务下线：

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：

服务中心每隔一段时间(默认60秒)将清单中没有续约的服务剔除。
通过eviction-interval-timer-in-ms配置可以对其进行修改，单位是毫秒

剔除时间配置



```
server:
  port: 7001 #端口号
spring:
  application:
    name: eureka-server # 应用名称，会在Eureka中作为服务的id标识 (serviceId)
  eureka:
    client:
      register-with-eureka: false #是否将自己注册到Eureka中
      fetch-registry: false #是否从eureka中获取服务信息
      service-url:
        defaultZone: http://localhost:7001/eureka # EurekaServer的地址
    server:
      #对无效的服务进行剔除操作，单位是毫秒
      eviction-interval-timer-in-ms: 5000
```

上图代码如下：

```
eviction-interval-timer-in-ms: 5000
```

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 dashboard at localhost:10086. The top navigation bar includes links for HOME and LAST 1000 SINCE STARTUP. The main section is titled 'System Status' and displays various system metrics:

Environment	test	Current time	2019-05-17T09:18:37 +0800
Data center	default	Uptime	00:06
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

A red box highlights a warning message: "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。" (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 one instance registered under '127.0.0.1':

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。

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

关闭自我保护

The screenshot shows the IntelliJ IDEA interface with the project structure and the contents of the `application.yml` file for the Eureka server.

```

server:
  port: 7001 #端口号
spring:
  application:
    name: eureka-server # 应用名称，会在Eureka中作为服务的id标识 (serviceName)
  eureka:
    client:
      register-with-eureka: false #是否将自己注册到Eureka中
      fetch-registry: false #是否从eureka中获取服务信息
      service-url:
        defaultZone: http://localhost:7001/eureka # EurekaServer的地址
    server:
      #关闭自我保护模式（缺省为打开）
      enable-self-preservation: false
      #对无效的服务进行剔除操作，单位是毫秒
      eviction-interval-timer-in-ms: 5000
  
```

上图配置如下：

```
enable-self-preservation: false
```

4.3 小结

- 理解Eureka的原理图:

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
服务提供者: 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: 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

- 能实现Eureka服务的搭建:引入依赖包, 配置配置文件, 在启动类上加`@EnableEurekaServer`。
- 能实现服务提供者向Eureka注册服务

- 引入eureka客户端依赖包
- 在`application.yml`中配置Eureka服务地址
- 在启动类上添加`@EnableDiscoveryClient`或者`@EnableEurekaClient`

- 能实现服务消费者向Eureka注册服务

- 引入eureka客户端依赖包
- 在`application.yml`中配置Eureka服务地址
- 在启动类上添加`@EnableDiscoveryClient`或者`@EnableEurekaClient`

- 能实现消费者通过Eureka访问服务提供者

5 负载均衡 Spring Cloud Ribbon

Ribbon主要解决集群服务中, 多个服务高效率访问的问题。

5.1 目标

- 理解Ribbon的负载均衡应用场景
- 能实现Ribbon的轮询、随机算法配置
- 理解源码对负载均衡的切换

5.2 讲解

5.2.1 Ribbon 简介

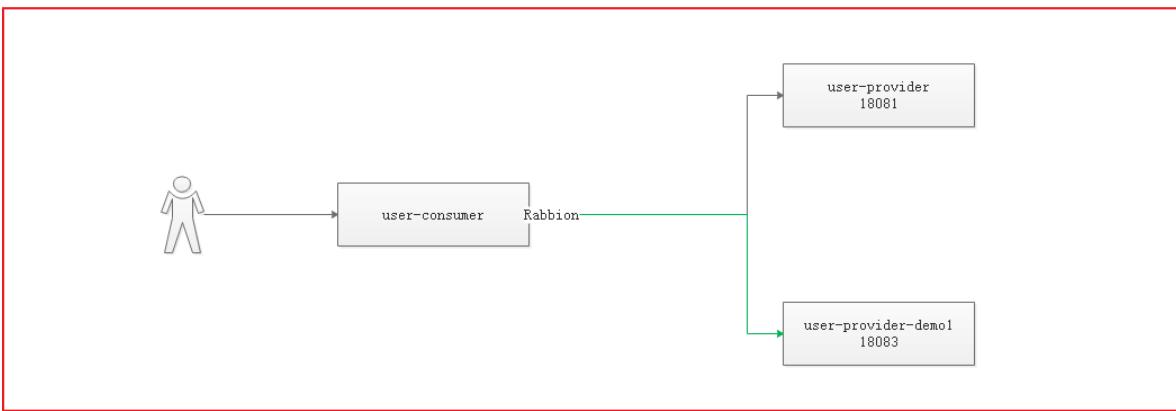
什么是Ribbon?

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。

Ribbon默认提供的负载均衡算法: 轮询, 随机, 重试法, 加权。当然, 我们可用自己定义负载均衡算法

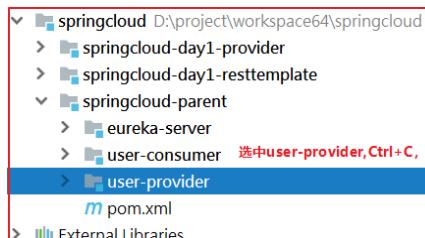
5.2.2 入门案例

5.2.2.1 多个服务集群

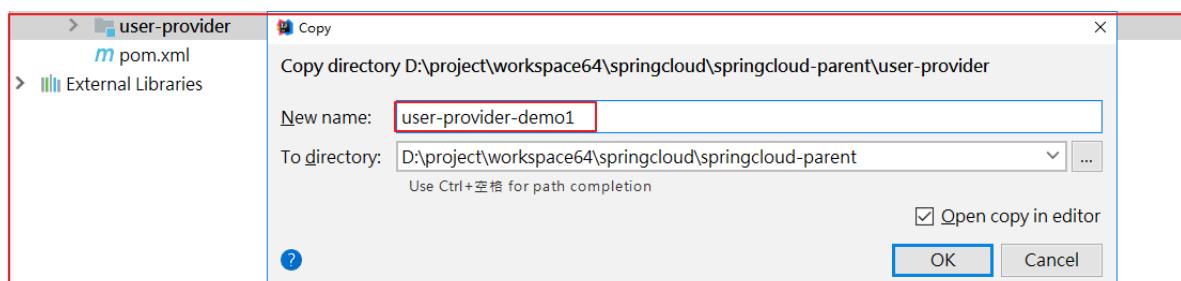


如果想要做负载均衡，我们的服务至少2个以上，为了演示负载均衡案例，我们可以复制2个工程，分别为 `user-provider` 和 `user-provider-demo1`，可以按照如下步骤拷贝工程：

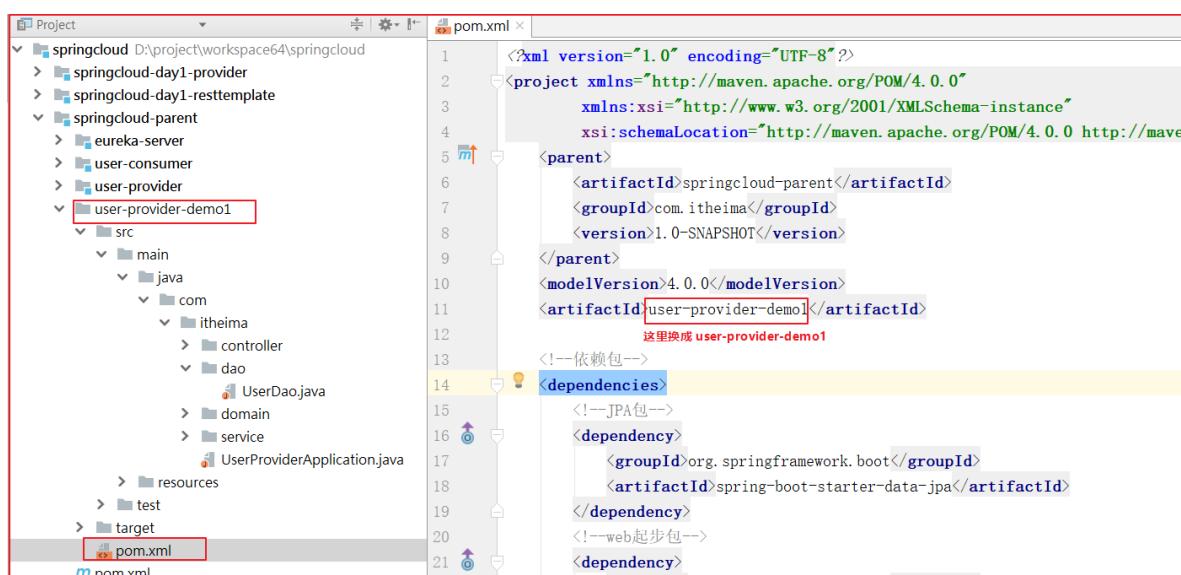
①选中 `user-provider`,按 `Ctrl+C`，然后 `Ctrl+V`



②名字改成 `user-provider-demo1`,点击OK



③将 `user-provider-demo1` 的 `artifactId` 换成 `user-provider-demo1`



④在 `springcloud-parent` 的 `pom.xml` 中添加一个 `<module>user-provider-demo1</module>`

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/pom-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>springcloud-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>user-provider</module>
        <module>user-provider-demo1</module> 添加该模块
        <module>user-consumer</module>
        <module>eureka-server</module>
    </modules>

    <!--父工程-->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.6.RELEASE</version>
    </parent>

```

⑤将 user-provider-demo1 的 application.yml 中的端口改成18083

```

server:
  port: 18083
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: itcast
    url: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterSet=utf8
    application:
      name: user-provider #服务的名字
      #指定eureka服务地址
  eureka:

```

为了方便测试，将2个工程对应的 com.itheima.controller.UserController 都修改一下：

user-provider:

```

@RequestMapping(value = "/find/{id}")
public User findById(@PathVariable(value = "id") Integer id){
    User user = userService.findById(id);
    user.setUsername(user+"      user-provider");
    return user;
}

```

user-provider-demo1:

```

@RequestMapping(value = "/find/{id}")
public User findById(@PathVariable(value = "id") Integer id){
    User user = userService.findById(id);
    user.setUsername(user+"      user-provider-demo1");
    return user;
}

```

⑥启动 eureka-server 和 user-provider、user-provider-demo1、user-consumer，启动前先注释掉 eureka-server 中的自我保护和剔除服务配置。

```

server:
  port: 7001 #端口号
spring:
  application:
    name: eureka-server # 应用名称，会在Eureka中作为服务的id标识 (serviceId)
  eureka:
    client:
      register-with-eureka: false #是否将自己注册到Eureka中
      fetch-registry: false #是否从eureka中获取服务信息
      service-url:
        defaultZone: http://localhost:7001/eureka # EurekaServer的地址
    server:
      #关闭自我保护模式（缺省为打开）
      enable-self-preservation: false
      #对无效的服务进行剔除操作，单位是毫秒 为了不影响程序运行，注释掉
      eviction-interval-timer-in-ms: 5000

```

访问 eureka-server 地址 <http://127.0.0.1:7001/> 效果如下：

Application	AMIs	Availability Zones	Status
USER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-79U5AQM:user-consumer:18082
USER-PROVIDER	n/a (2)	(2)	UP (2) - DESKTOP-79U5AQM:user-provider:18081 DESKTOP-79U5AQM:user-provider:18083

5.2.2.2 开启负载均衡

(1) 客户端开启负载均衡

Eureka已经集成Ribbon，所以无需引入依赖，要想使用Ribbon，直接在RestTemplate的配置方法上添加 @LoadBalanced 注解即可。

修改 user-consumer 的 com.itheima.UserConsumerApplication 启动类，在 restTemplate() 方法上添加 @LoadBalanced 注解，代码如下：

```

* @Date: 2019/7/13 22:14
* @Description: com.itheima
*/
@SpringBootApplication
@EnableDiscoveryClient //开启Eureka客户端发现功能
public class UserConsumerApplication {

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

  /**
   * 将RestTemplate的实例放到Spring容器中
   * @return
   */
  @Bean
  @LoadBalanced //开启负载均衡
  public RestTemplate restTemplate() {
    return new RestTemplate();
  }
}

```

(2) 采用服务名访问配置

修改 user-consumer 的 com.itheima.controller.UserController 的调用方式，不再手动获取ip和端口，而是直接通过服务名称调用，代码如下：

The screenshot shows the IntelliJ IDEA interface with the UserController.java file open. The code implements a REST endpoint to query a user by ID, utilizing a Eureka client to discover the provider service. Below the code, a table from the Eureka UI shows two instances registered under the application 'USER-PROVIDER': one at port 18081 and another at port 18080.

```

    @GetMapping(value = "/{id}")
    public User queryById(@PathVariable(value = "id") Integer id) {
        String url = "http://localhost:18081/user/find/" + id;
        //return restTemplate.getForObject(url, User.class);

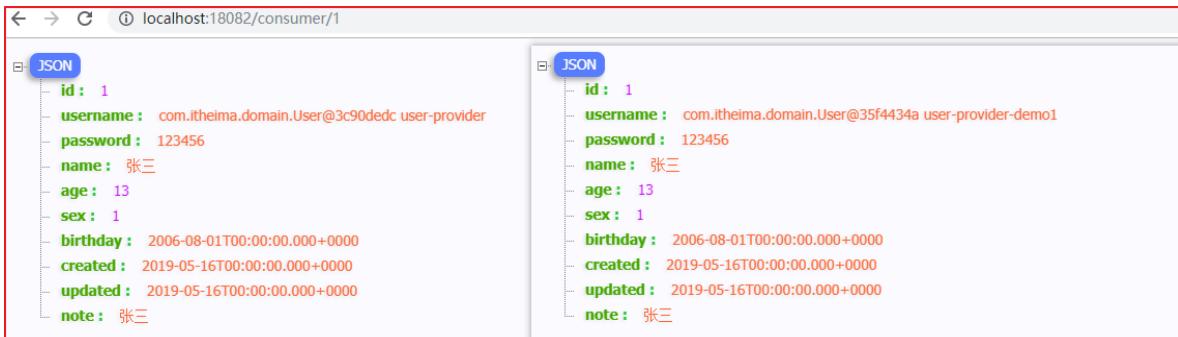
        //获取指定实例
        List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
        //获取第1个实例对象
        ServiceInstance serviceInstance = instances.get(0);
        //拼接服务地址
        String instanceUrl = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/find/" + id;
        //return restTemplate.getForObject(instanceUrl, User.class);
    }

```

Application	AMIs	Availability Zones	Status
USER-CONSUMER	n/a (1)	(1)	UP (1) - DESKTOP-79U5AQM:user-consumer:18082
USER-PROVIDER	n/a (2)	(2)	UP (2) - DESKTOP-79U5AQM:user-provider:18081, DESKTOP-79U5AQM:user-provider:18080

(3) 测试

启动并访问测试 `<http://localhost:18082/consumer/1>`, 可以发现, 数据会在2个服务之间轮询切换。



5.2.2.3 其他负载均衡策略配置

配置修改轮询策略: Ribbon默认的负载均衡策略是轮询, 通过如下

```

# 修改服务地址轮询策略, 默认是轮询, 配置之后变随机
user-provider:
  ribbon:
    #轮询
    #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
    #随机算法
    #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
    #重试算法, 该算法先按照轮询的策略获取服务, 如果获取服务失败则在指定的时间内会进行重试, 获取可用的服务
    #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RetryRule
    #加权法, 会根据平均响应时间计算所有服务的权重, 响应时间越快服务权重越大被选中的概率越大。刚
    #启动时如果同统计信息不足, 则使用轮询的策略, 等统计信息足够会切换到自身规则。
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.ZoneAvoidanceRule

```

SpringBoot可以修改负载均衡规则, 配置为 `ribbon.NFLoadBalancerRuleClassName`

格式`{服务名称}.ribbon.NFLoadBalancerRuleClassName`

5.2.3 负载均衡源码跟踪探究

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：

打开LoadBalancerInterceptor类，断点打入intercept方法中

```
51     @Override
52     public ClientHttpResponse intercept(final HttpRequest request, final byte[] body, final InterceptingClientHttpRequest execution) throws IOException {
53         final URI originalUri = request.getURI(); // originalUri: "http://user-provider/user/find/1"
54         String serviceName = originalUri.getHost(); // 取得服务名字
55         Assert.state(serviceName != null, "serviceName: " + serviceName);
56         message("Request URI does not contain a valid hostname: " + originalUri);
57         return this.loadBalancer.execute(serviceName,
58             this.requestFactory.createRequest(request, body, execution));
59     }
60 }
61
62 }
```

继续跟入execute方法：发现获取了18081发端口的服务

```
112     */
113     public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) throws IOException {
114         ILoadBalancer loadBalancer = getLoadBalancer(serviceId); // 负载均衡器
115         Server server = getServer(loadBalancer, hint); // server: "127.0.0.1:18081" 负载均衡器
116         if (server == null) { // server: "127.0.0.1:18081"
117             throw new IllegalStateException("No instances available for " + serviceId);
118         }
119         RibbonServer ribbonServer = new RibbonServer(serviceId, server,
120             isSecure(server, serviceId),
121             serverIntrospector(serviceId).getMetadata(server));
122     }
123
124 }
```

再跟下一次，发现获取的是18081和18083之间切换

```
112     */
113     public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) throws IOException {
114         ILoadBalancer loadBalancer = getLoadBalancer(serviceId); // 负载均衡器
115         Server server = getServer(loadBalancer, hint); // server: "127.0.0.1:18083" 负载均衡器
116         if (server == null) { // server: "127.0.0.1:18083"
117             throw new IllegalStateException("No instances available for " + serviceId);
118         }
119         RibbonServer ribbonServer = new RibbonServer(serviceId, server,
120             isSecure(server, serviceId),
121             serverIntrospector(serviceId).getMetadata(server));
122
123 }
```

通过代码断点内容判断，果然是实现了负载均衡

5.3 小结

- Ribbon的负载均衡算法应用在客户端，只需要提供服务列表，就能帮助消费端自动访问服务端，并通过不同算法实现负载均衡。
- Ribbon的轮询、随机算法配置：在application.yml中配置 `{服务名}.ribbon.NFLoadBalancerRuleClassName`
- 负载均衡的切换：在LoadBalancerInterceptor中获取服务的名字，通过调用 RibbonLoadBalancerClient的execute方法，并获取ILoadBalancer负载均衡器，然后根据

ILoadBalancer负载均衡器查询出要使用的节点，再获取节点的信息，并实现调用。

6 熔断器 Spring Cloud Hystrix

6.1 目标

- 理解Hystrix的作用
- 理解雪崩效应
- 知道熔断器的3个状态以及3个状态的切换过程
- 能理解什么是线程隔离，什么是服务降级
- 能实现一个局部方法熔断案例
- 能实现全局方法熔断案例

6.2 讲解

6.2.1 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。

6.2.2 雪崩效应

什么是雪崩效应？

1. 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。
2. 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。
3. 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；

6.2.3 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：

1. **Closed**: 关闭状态，所有请求正常访问

2. **Open**: 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(yu: 四声)值(极限值)，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

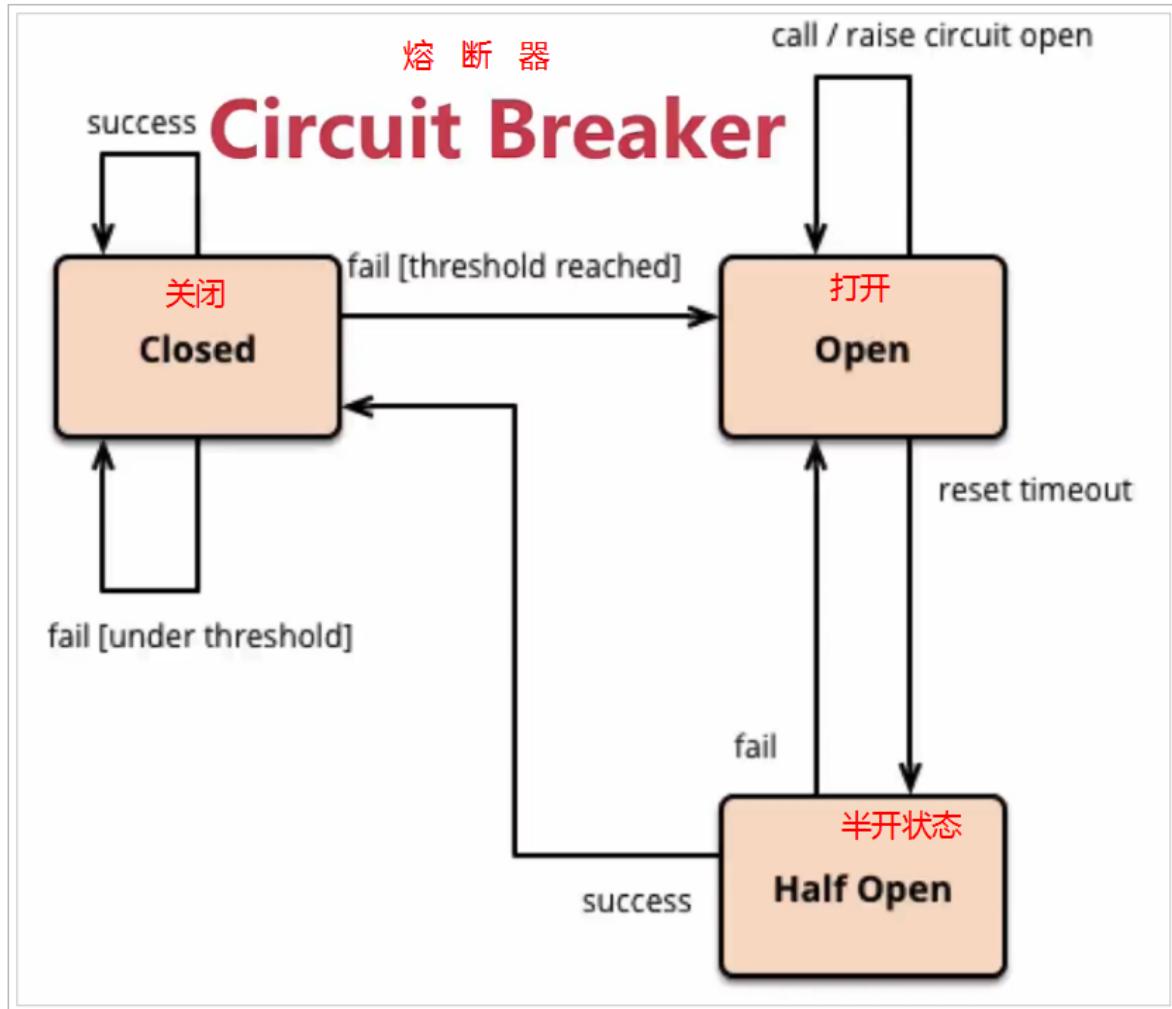
3. **Half Open**: 半开状态

Open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closed状态。如果失败，熔断器切回open状态。

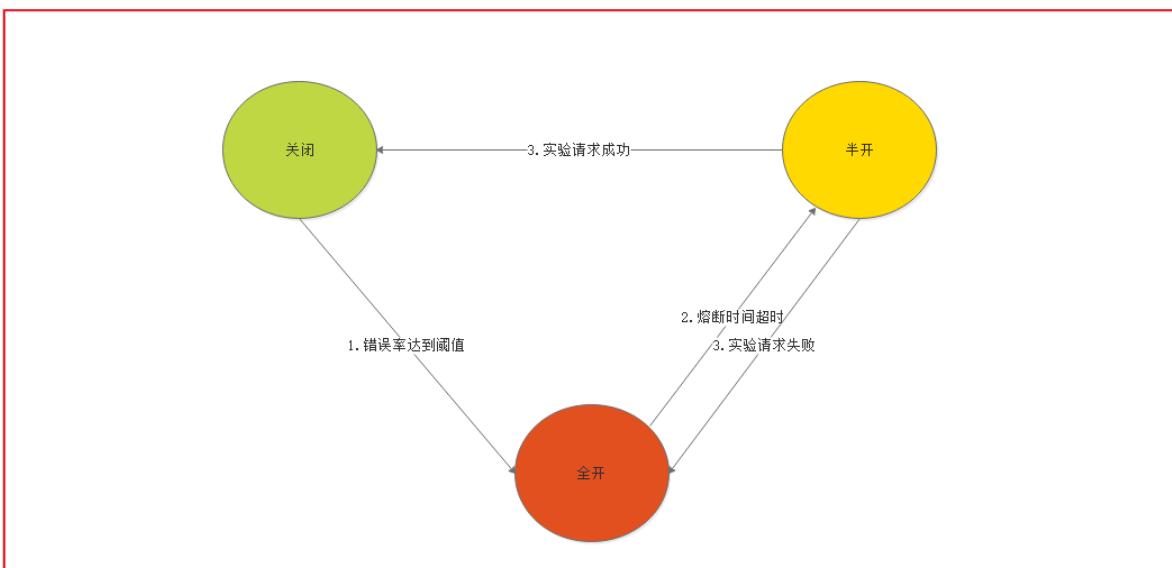
threshold reached 到达阈(yu: 四声)值

under threshold 阈值以下



【Hystrix熔断状态机模型：配图】

翻译之后的图：



熔断器的核心：线程隔离和服务降级。

1. 线程隔离：是指Hystrix为每个依赖服务调用一个小的线程池，如果线程池用尽，调用立即被拒绝，默认不采用排队。
2. 服务降级(兜底方法)：优先保证核心服务，而非核心服务不可用或弱可用。触发Hystrix服务降级的情况：线程池已满、请求超时。

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)。

6.2.4 局部熔断案例

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)引入熔断的依赖坐标：

在 user-consumer 中加入依赖

```
<!--熔断器-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

(2)开启熔断的注解

修改 user-consumer 的 com.itheima.UserConsumerApplication，在该类上添加
@EnableCircuitBreaker，代码如下：

```
@SpringBootApplication
@EnableDiscoveryClient //开启Eureka客户端发现功能
@EnableCircuitBreaker //开启熔断
public class UserConsumerApplication {

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

    /**
     * 将RestTemplate的实例放到Spring容器中
     * @return
     */
    @Bean
    @LoadBalanced //开启负载均衡
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

注意：这里也可以使用 @SpringCloudApplication，写了 @SpringCloudApplication 后，其他注解需要全部去掉。

(3)服务降级处理

在 user-consumer 的 com.itheima.controller.UserController 中添加降级处理方法，方法如下：

```

*****
 * 服务降级处理方法
 * @return
 */
public User failBack(Integer id){
    User user = new User();
    user.setUsername("服务降级,默认处理！");
    return user;
}

```

在有可能发生问题的方法上添加降级处理调用，例如在 queryById 方法上添加降级调用，代码如下：

```

@HystrixCommand(fallbackMethod = "failBack") //方法如果发生问题，则调用降级处理方法
@GetMapping(value = "/{id}")
public User queryById(@PathVariable(value = "id") Integer id){
    //String url = "http://localhost:18081/user/find/" + id;
    //return restTemplate.getForObject(url, User.class);

    //获取指定实例
    //List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
    //获取第1个实例对象
    //ServiceInstance serviceInstance = instances.get(0);
    //拼接服务地址
    //String instanceUrl = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/find/" + id;
    //return restTemplate.getForObject(instanceUrl, User.class);
    //测试负载均衡      服务的名字
    String url = "http://user-provider/user/find/" + id;
    return restTemplate.getForObject(url, User.class);
}

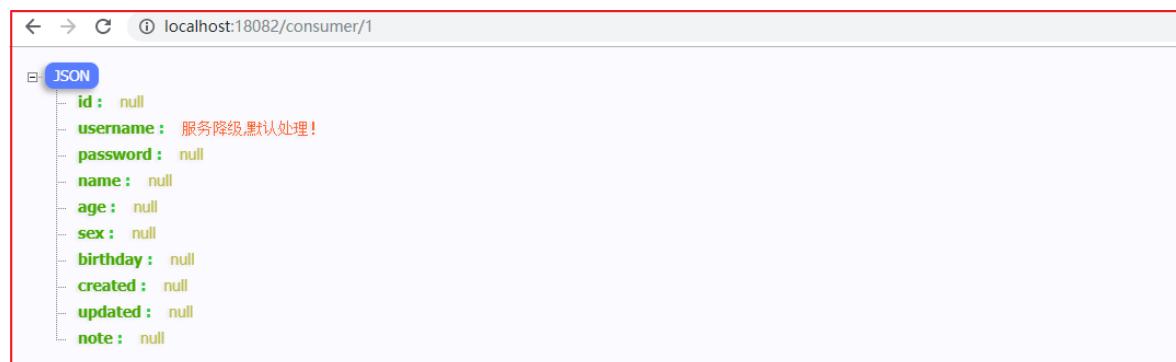
*****
 * 服务降级处理方法
 * @return
 */
public User failBack(Integer id){ //...
    User user = new User();
    user.setUsername("服务降级,默认处理！");
    return user;
}

```

(4) 测试

将服务全部停掉，启动 eureka-server 和 user-consumer，然后请求

<<http://localhost:18082/consumer/1>> 测试效果如下：



6.2.5 其他熔断策略配置

1. 熔断后休眠时间: `sleepWindowInMilliseconds`
2. 熔断触发最小请求次数: `requestVolumeThreshold`
3. 熔断触发错误比例阈值: `errorThresholdPercentage`
4. 熔断超时时间: `timeoutInMilliseconds`

配置如下:

```
# 配置熔断策略:
hystrix:
  command:
    default:
      circuitBreaker:
        # 强制打开熔断器 默认false关闭的。测试配置是否生效
        forceOpen: false
        # 触发熔断错误比例阈值, 默认值50%
        errorThresholdPercentage: 50
        # 熔断后休眠时长, 默认值5秒
        sleepWindowInMilliseconds: 10000
        # 熔断触发最小请求次数, 默认值是20
        requestVolumeThreshold: 10
      execution:
        isolation:
          thread:
            # 熔断超时设置, 默认为1秒
            timeoutInMilliseconds: 2000
```

(1)超时时间测试

- a.修改 `user-provider` 的 `com.itheima.controller.UserController` 的 `findById` 方法, 让它休眠3秒钟。
- b.修改 `user-consumer` 的 `application.yml`, 设置超时时间5秒, 此时不会熔断。

```
hystrix:
  command:
    default:
      circuitBreaker:
        # 强制打开熔断器 默认false关闭的。测试配置是否生效
        forceOpen: true
        # 触发熔断错误比例阈值, 默认值50%
        errorThresholdPercentage: 100
        # 熔断后休眠时长, 默认值5秒
        sleepWindowInMilliseconds: 30000
        # 熔断触发最小请求次数, 默认值是20
        requestVolumeThreshold: 2
      execution:
        isolation:
          thread:
            # 熔断超时设置, 默认为1秒
            timeoutInMilliseconds: 5000
```

- c.如果把超时时间改成2000, 此时就会熔断。

(2)熔断触发最小请求次数测试

- a.修改 `user-provider` 的 `com.itheima.controller.UserController`, 在方法中制造异常, 代码如下:

```

@RequestMapping(value = "/find/{id}")
public User findById(@PathVariable(value = "id") Integer id) {
    //如果id==1，则抛出异常
    if(id==1) {
        throw new RuntimeException("");
    }

    User user = userService.findById(id);
    user.setUsername(user+"      user-provider");
    return user;
}

```

b.3次并发请求 `<http://localhost:18082/consumer/1>`，会触发熔断

再次请求 `<http://localhost:18082/consumer/2>` 的时候，也会熔断，5秒钟会自动恢复。

并发请求建议使用 `jmeter` 工具。

6.2.5 扩展-服务降级的fallback方法：

两种编写方式：编写在类上，编写在方法上。在类的上边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="failBack")` 用来声明一个降级逻辑的 `failBack` 兜底方法

(2)类上默认服务降级的fallback兜底方法

刚才把 `failback` 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="defaultFailBack")`，在类上，指明统一的失败降级方法；

(3)案例

a. 在 `user-consumer` 的 `com.itheima.controller.UserController` 类中添加一个全局熔断方法：

```

*****
 * 全局的服务降级处理方法
 * @return
 */
public User defaultFailBack(){
    User user = new User();
    user.setUsername("Default-服务降级，默认处理！");
    return user;
}

```

b. 在 `queryById` 方法上将原来的 `@HystrixCommand` 相关去掉，并添加 `@HystrixCommand` 注解：

```
@HystrixCommand
@GetMapping(value = "/{id}")
public User queryById(@PathVariable(value = "id") Integer id){
    //...略
    return user;
}
```

c.在 user-consumer 的 com.itheima.controller.UserController 类上添加
@DefaultProperties(defaultFallback = "defaultFallback")

d.测试访问 <http://localhost:18082/consumer/1>，效果如下：



com.itheima.controller.UserController 完整代码：

```

@RestController
@DefaultProperties(defaultFallback = "defaultFailBack")
@RequestMapping(value = "/consumer")
public class UserController {

    /**
     * 全局的服务降级处理方法
     * @return
     */
    public User defaultFailBack() {
        User user = new User();
        user.setUsername("Default-服务降级,默认处理!");
        return user;
    }

    @Autowired
    private RestTemplate restTemplate;

    //可以用来发现服务
    @Autowired
    private DiscoveryClient discoveryClient;

    /**
     * 在user-consumer服务中通过RestTemplate调用user-provider服务
     * @param id
     * @return
     */
    //@HystrixCommand(fallbackMethod = "failBack")
    @HystrixCommand
    @GetMapping(value = "/{id}")
    public User queryById(@PathVariable(value = "id") Integer id) {
        //String url = "http://localhost:18081/user/find/"+id;
        //return restTemplate.getForObject(url,User.class);

        //获取指定实例
        //List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
        //获取第1个实例对象
        //ServiceInstance serviceInstance = instances.get(0);
        //拼接服务地址
        //String instanceUrl = "http://"+serviceInstance.getHost()+":"+serviceInstance.getPort()+"/user/find/"+id;
        //return restTemplate.getForObject(instanceUrl,User.class);
        //测试负载均衡      服务的名字
        String url = "http://user-provider/user/find/"+id;
        return restTemplate.getForObject(url,User.class);
    }

    /**
     * 服务降级处理方法
     * @return
     */
    public User failBack(Integer id) {
        User user = new User();
        user.setUsername("服务降级,默认处理!");
        return user;
    }
}

```

6.3 小结

- Hystrix的作用: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。
- 理解雪崩效应:

1. 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。
2. 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。
3. 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；

- 知道熔断器的3个状态以及3个状态的切换过程

1. **Closed**: 关闭状态，所有请求正常访问

2. **Open**: 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(yu: 四声)值(极限值)，则触发熔断，断路器完全关闭

默认失败比例的阈值是50%，请求数量最少不少于20次

3. **Half Open**: 半开状态

Open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closed状态。如果失败，熔断器切回open状态。

threshold reached 到达阈(yu: 四声)值

under threshold 阈值以下

- 能理解什么是线程隔离，什么是服务降级

1. 线程隔离：是指Hystrix为每个依赖服务调用一个小的线程池，如果线程池用尽，调用立即被拒绝，
默认不采用排队。

2. 服务降级(兜底方法)：优先保证核心服务，而非核心服务不可用或弱可用。触发Hystrix服务降级的情况：
线程池已满、请求超时。

- 能实现一个局部方法熔断案例

1. 定义一个局部处理熔断的方法**fallback()**

2. 在指定方法上使用@HystrixCommand(fallbackMethod = "fallback")配置调用

- 能实现全局方法熔断案例

1. 定义一个全局处理熔断的方法**defaultFallback()**

2. 在类上使用@DefaultProperties(defaultFallback = "defaultFallback")配置调用

3. 在指定方法上使用@HystrixCommand

第2天 SpringCloud

学习目标

- ==能够使用Feign进行远程调用==

1. **feign**的使用->解决远程请求中硬编码问题

2. 负载均衡配置

3. 支持熔断配置

4. 请求压缩

5. 日志配置

- 能够搭建==Spring Cloud Gateway==

1. 微服务网关

2. 路由

3. 过滤配置

- 能够配置Spring Cloud Gateway过滤器

- 能够使用Spring Cloud Gateway默认过滤器：全局过滤、局部过滤
- 能够搭建Spring Cloud Config配置中心服务

1. 集中管理配置文件

- 能够使用Spring Cloud Bus 消息总线实时更新配置文件

1. 每个微服务的通知消息管理服务

1 Spring Cloud Feign

1.1 目标

- 了解Feign的作用
- 掌握Feign的使用过程
- 掌握Feign的负载均衡配置
- 掌握Feign的熔断配置
- 掌握Feign的压缩配置
- 掌握Feign的日志配置

1.2 讲解

1.2.1 Feign简介

Feign [feɪn] 译文 伪装。Feign是一个声明式WebService客户端.使用Feign能让编写WebService客户端更加简单,它的使用方法是定义一个接口，然后在上面添加注解。不再需要拼接URL，参数等操作。项目主页：<https://github.com/OpenFeign/feign>。

- 集成Ribbon的负载均衡功能
- 集成了Hystrix的熔断器功能
- 支持请求压缩
- 大大简化了远程调用的代码，同时功能还增强啦
- Feign以更加优雅的方式编写远程调用代码，并简化重复代码

1.2.2 快速入门

使用Feign替代RestTemplate发送Rest请求。

实现步骤：

1. 导入feign依赖
2. 编写Feign客户端接口
3. 消费者启动引导类开启Feign功能注解
4. 访问接口测试

实现过程：

(1)导入依赖

在 user-consumer 中添加 spring-cloud-starter-openfeign 依赖

```
<!--配置feign-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

(2)创建Feign客户端

在 user-consumer 中创建 com.itheima.feign.UserClient 接口，代码如下：

```
@FeignClient(value = "user-provider") #服务的名字
public interface UserClient {
    /**
     * 根据ID获取用户信息
     * @param id
     * @return
     */
    @RequestMapping(value = "/user/find/{id}")
    User findById(@PathVariable("id") Integer id);
}
```

```
spring:
  datasource: user-provider 的 application.yml
  driver-class-name: com.mysql.cj.jdbc.Driver
  username: root
  password: itcast
  url: jdbc:mysql://127.0.0.1:3306/springcloud?
application:
  name: user-provider #服务的名字
```

解释：

Feign会通过动态代理，帮我们生成实现类。
注解@FeignClient声明Feign的客户端，注解value指明服务名称
接口定义的方法，采用SpringMVC的注解。Feign会根据注解帮我们生成URL地址
注解@RequestMapping中的/user，不要忘记。因为Feign需要拼接可访问地址

(3)编写控制层

在 user-consumer 中创建 com.itheima.controller.ConsumerFeignController，在Controller中使用@Autowired注入FeignClient,代码如下：

```
@RestController
@RequestMapping(value = "/feign")
public class ConsumerFeignController {

    @Autowired
    private UserClient userClient;

    *****
    * 使用Feign调用user-provider的方法
    */
    @RequestMapping(value = "/{id}")
    public User queryById(@PathVariable("id") Integer id){
        return userClient.findById(id);
    }
}
```

```
    }  
}
```

(4)开启Feign

修改 user-consumer 的启动类，在启动类上添加 `@EnableFeignClients` 注解，开启Feign,代码如下：

```
@SpringBootApplication  
@EnableDiscoveryClient //开启Eureka客户端发现功能  
@EnableCircuitBreaker //开启熔断  
@EnableFeignClients //开启Feign  
public class UserConsumerApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(UserConsumerApplication.class, args);  
    }  
  
    /**/  
     * 将RestTemplate的实例放到Spring容器中  
     * @return  
     */  
    @Bean  
    @LoadBalanced //开启负载均衡  
    public RestTemplate restTemplate(){  
        return new RestTemplate();  
    }  
}
```

(5)测试

请求 `<http://localhost:18082/feign/2>`，效果如下：



1.2.3 负载均衡

Feign自身已经集成了Ribbon，因此使用Feign的时候，不需要额外引入依赖。

```
✓ org.springframework.cloud:spring-cloud-starter-openfeign:2.1.1.RELEASE  
  ✓ org.springframework.cloud:spring-cloud-starter:2.1.1.RELEASE (omitted for duplicate)  
✓ org.springframework.cloud:spring-cloud-openfeign-core:2.1.1.RELEASE  
  ✓ org.springframework.boot:spring-boot-autoconfigure:2.1.6.RELEASE (omitted for duplicate)  
  ✓ org.springframework.cloud:spring-cloud-netflix-ribbon:2.1.1.RELEASE (omitted for duplicate)  
  ✓ org.springframework.boot:spring-boot-starter-aop:2.1.6.RELEASE (omitted for duplicate)  
  > io.github.openfeign.form:feign-form-spring:3.5.0  
  ✓ org.springframework.web:5.1.8.RELEASE (omitted for duplicate)  
  > org.springframework.cloud:spring-cloud-commons:2.1.1.RELEASE  
  ✓ io.github.openfeign:feign-core:10.1.0  
  > io.github.openfeign:feign-slf4j:10.1.0  
  ✓ io.github.openfeign:feign-hystrix:10.1.0
```

Feign内置的ribbon默认设置了请求超时时长， 默认是1000， 可以修改
ribbon内部有重试机制， 一旦超时， 会自动重新发起请求。如果不希望重试可以关闭配置：

```
# 修改服务地址轮询策略， 默认是轮询， 配置之后变随机
user-provider:
  ribbon:
    #轮询
    NLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
    ConnectTimeout: 10000 # 连接超时时间
    ReadTimeout: 2000 # 数据读取超时时间
    MaxAutoRetries: 1 # 最大重试次数(第一个服务)
    MaxAutoRetriesNextServer: 0 # 最大重试下一个服务次数(集群的情况才会用到)
    OkToRetryOnAllOperations: false # 无论是请求超时或者socket read timeout都进行重试
```

1.2.4 熔断器支持

feign整合Hystrix熔断器

Feign默认也有对Hystrix的集成！

```
✓ org.springframework.cloud:spring-cloud-starter-openfeign:2.1.1.RELEASE
  ✓ org.springframework.cloud:spring-cloud-starter:2.1.1.RELEASE (omitted for duplicate)
  ✓ org.springframework.cloud:spring-cloud-openfeign-core:2.1.1.RELEASE
    ✓ org.springframework.boot:spring-boot-autoconfigure:2.1.6.RELEASE (omitted for duplicate)
    ✓ org.springframework.cloud:spring-cloud-netflix-ribbon:2.1.1.RELEASE (omitted for duplicate)
    ✓ org.springframework.boot:spring-boot-starter-aop:2.1.6.RELEASE (omitted for duplicate)
    > io.github.openfeign.form:feign-form-spring:3.5.0
    ✓ org.springframework:spring-web:5.1.8.RELEASE (omitted for duplicate)
  ✓ org.springframework.cloud:spring-cloud-commons:2.1.1.RELEASE
  ✓ io.github.openfeign:feign-core:10.1.0
  > io.github.openfeign:feign-slf4j:10.1.0
  > io.github.openfeign:feign-hystrix:10.1.0
```

实现步骤：

1. 在配置文件application.yml中开启feign熔断器支持
2. 编写Fallback处理类， 实现FeignClient客户端
3. 在@FeignClient注解中， 指定Fallback处理类。
4. 测试

(1)开启Hystrix

在配置文件application.yml中开启feign熔断器支持：默认关闭

```
feign:
  hystrix:
    enabled: true # 开启Feign的熔断功能
```

(2)熔断降级类创建

修改 user-consumer, 创建一个类 com.itheima.feign.fallback.UserClientFallback， 实现刚才编写的UserClient， 作为FallBack的处理类, 代码如下：

```
@Component
public class UserClientFallback implements UserClient{
```

```

    /**
     * 服务降级处理方法
     * @param id
     * @return
     */
    @Override
    public User findById(Integer id) {
        User user = new User();
        user.setUsername("Fallback, 服务降级。。。");
        return user;
    }
}

```

(3)指定Fallback处理类

在@FeignClient注解中，指定FallBack处理类

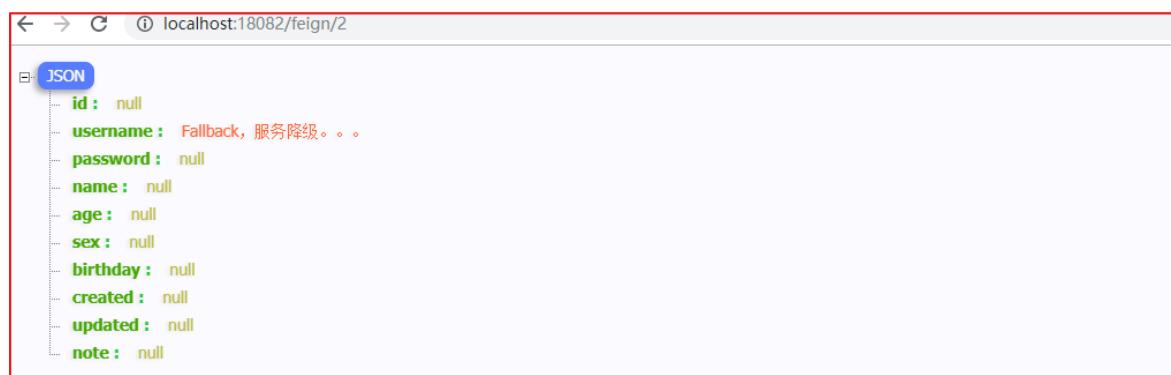
```

@FeignClient(value = "user-provider", fallback = UserClientFallback.class)
public interface UserClient {
    /**
     * 根据ID获取用户信息
     * @param id
     * @return
     */
    @RequestMapping(value = "/user/find/{id}")
    User findById(@PathVariable(value = "id") Integer id);
}

```

(4)测试

关闭服务消费方，请求 `<http://localhost:18082/feign/2>`，效果如下：



1.2.5 请求压缩(了解)

SpringCloudFeign支持对请求和响应进行GZIP压缩，以减少通信过程中的性能损耗。

通过配置开启请求与响应的压缩功能：

```
feign:  
  compression:  
    request:  
      enabled: true # 开启请求压缩  
    response:  
      enabled: true # 开启响应压缩
```

也可以对请求的数据类型，以及触发压缩的大小下限进行设置

```
# Feign配置  
feign:  
  compression:  
    request:  
      enabled: true # 开启请求压缩  
      mime-types: text/html,application/xml,application/json # 设置压缩的数据类型  
      min-request-size: 2048 # 设置触发压缩的大小下限  
    #以上数据类型，压缩大小下限均为默认值
```

1.2.6 Feign的日志级别配置

通过`loggin.level.xx=debug`来设置日志级别。然而这个对Feign客户端不会产生效果。因为`@FeignClient`注解修饰的客户端在被代理时，都会创建一个新的`Feign.Logger`实例。我们需要额外通过配置类的方式指定这个日志的级别才可以。

实现步骤：

1. 在`application.yml`配置文件中开启日志级别配置
2. 编写配置类，定义日志级别`bean`。
3. 在接口的`@FeignClient`中指定配置类
4. 重启项目，测试访问

实现过程：

(1)普通日志等级配置

在`user-consumer`的配置文件中设置`com.itheima`包下的日志级别都为`debug`

```
# com.itheima 包下的日志级别都为Debug  
logging:  
  level:  
    com.itheima: debug
```

(2)Feign日志等级配置

在`user-consumer`中创建`com.itheima.feign.util.FeignConfig`,定义日志级别

```

@Configuration
public class FeignConfig {

    /**
     * 日志级别
     * @return
     */
    @Bean
    public Logger.Level feignLoggerLevel(){
        return Logger.Level.FULL;
    }
}

```

日志级别说明：

Feign支持4中级别：

NONE: 不记录任何日志， 默认值

BASIC: 仅记录请求的方法， URL以及响应状态码和执行时间

HEADERS: 在**BASIC**基础上， 额外记录了请求和响应的头信息

FULL: 记录所有请求和响应的明细， 包括头信息、 请求体、 元数据

(3)指定配置类

修改 user-consumer 的 `com.itheima.feign.UserClient` 指定上面的配置类， 代码如下：

```

@FeignClient(value = "user-provider", fallback = UserClientFallback.class, configuration = FeignConfig.class)
public interface UserClient {

    /**
     * 根据ID获取用户信息
     * @param id
     * @return
     */
    @RequestMapping(value = "/user/find/{id}")
    User findById(@PathVariable(value = "id") Integer id);
}

```

重启项目， 即可看到每次访问的日志

```

2019-07-16 17:07:07.746 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] ---> GET http://user-provider/user/find/2 HTTP/1.1
2019-07-16 17:07:07.746 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] Accept-Encoding: deflate
2019-07-16 17:07:07.746 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] Accept-Encoding: gzip
2019-07-16 17:07:07.746 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] ---> END HTTP (0-byte body)
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] <--- HTTP/1.1 200 (6ms)
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] content-type: application/json;charset=UTF-8
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] date: Tue, 16 Jul 2019 09:07:07 GMT
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] transfer-encoding: chunked
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById]
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] {"id":2,"username":"com.itheima.domain.User@157aedf8"
2019-07-16 17:07:07.753 DEBUG 15496 --- [user-provider-7] com.itheima.feign.UserClient      : [UserClient#findById] <--- END HTTP (265-byte body)

```

1.3 小结

- Feign的作用:不再使用拼接URL的方式实现远程调用，以接口调用的方式实现远程调用，简化了远程调用的实现方式，增强了远程调用的功能，例如：增加了负载均衡、熔断、压缩、日志启用。
- 掌握Feign的使用过程

- 1. 引入Feign依赖包
- 2. 创建Feign接口, feign接口中需要指定调用的服务名字
- 3. 使用@EnabledFeignClients启用Feign功能

- 掌握Feign的负载均衡配置

在配置文件中配置

{spring.application.name}: ribbon: 负载均衡属性配置

- 掌握Feign的熔断配置

1. 在application.yml中开启Hystrix

2. 给Feign接口创建一个实现类

3. 给Feign指定fallback类

- 掌握Feign的压缩配置

在application.yml中指定压缩属性即可

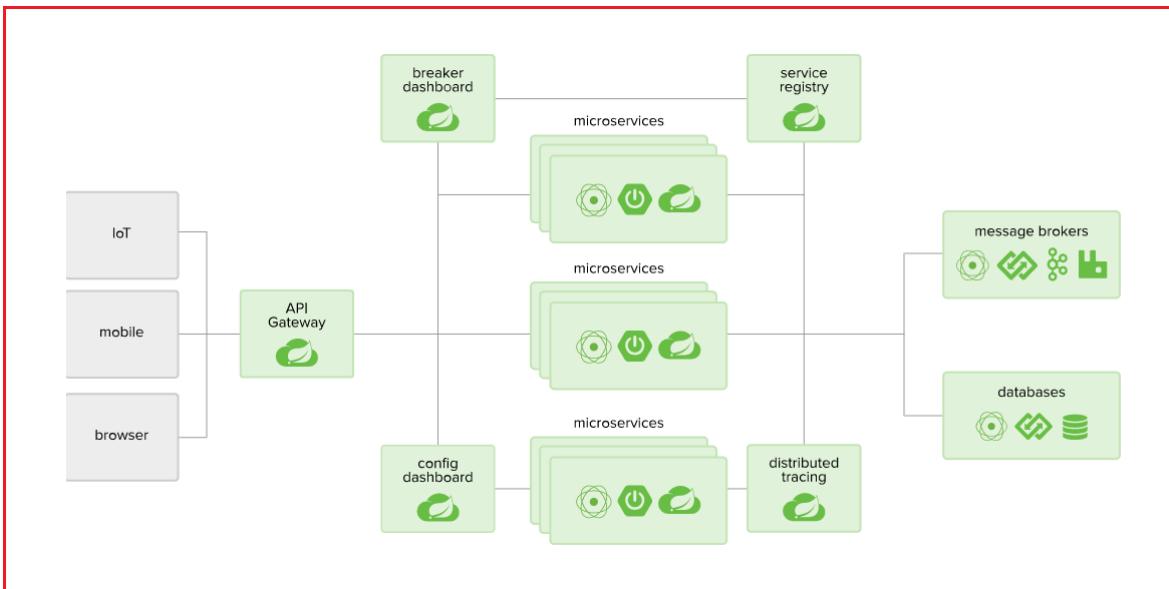
- 掌握Feign的日志配置

1. 在application.yml中开启普通日志等级

2. 创建一个类, 定义Feign日志等级

3. 在Feign接口中指定定义日志的配置

2 网关 Spring Cloud Gateway



2.1 目标

- 网关的作用
- 会配置动态路由
- 会配置过滤器
- 能自定义全局过滤器

==网关作用：为微服务提供统一的路由管理，可以在路由管理基础上进行一系列的过滤，可以做一系列的监控操作，限流。 ==

2.2 讲解

2.2.1 Gateway 简介

Spring Cloud Gateway 是Spring Cloud团队的一个全新项目，基于Spring 5.0、SpringBoot2.0、Project Reactor 等技术开发的网关。 ==旨在为微服务架构提供一种简单有效统一的API路由管理方式。
==

Spring Cloud Gateway 作为SpringCloud生态系统中的网关，目标是替代Netflix Zuul。Gateway不仅提供统一路由方式，并且==基于Filter链的方式提供网关的基本功能。例如：安全，监控/指标，和限流。 ==

本身也是一个微服务，需要注册到Eureka

网关的核心功能：过滤、路由

核心概念：通过画图解释

- **路由(route):**
- **断言Predicate函数：**路由转发规则
- **过滤器(Filter):**

2.2.2 快速入门

实现步骤：

1. 创建gateway-service工程SpringBoot
2. 编写基础配置
3. 编写路由规则，配置静态路由策略
4. 启动网关服务进行测试

实现过程：

(1)创建一个子工程gateway-service

工程坐标：

```
<artifactId>gateway-service</artifactId>
<groupId>com.itheima</groupId>
<version>1.0-SNAPSHOT</version>
```

(2)pom.xml依赖

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

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>springcloud-parent</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>gateway-service</artifactId>

    <dependencies>
        <!--网关依赖-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
        <!-- Eureka客户端 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
    </dependencies>
</project>

```

(3)启动类

创建启动类 `com.itheima.GatewayApplication`, 代码如下:

```

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

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

```

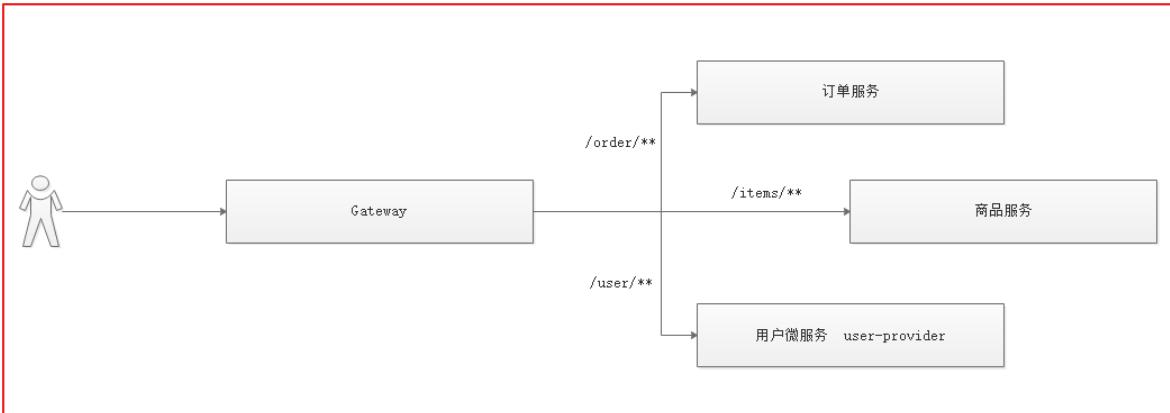
(4)application.yml配置

```

# 注释版本
server:
  port: 18084
spring:
  application:
    name: api-gateway # 应用名
# Eureka服务中心配置
eureka:
  client:
    service-url:
      # 注册Eureka Server集群
      defaultZone: http://127.0.0.1:7001/eureka

```

2.2.3 路由配置

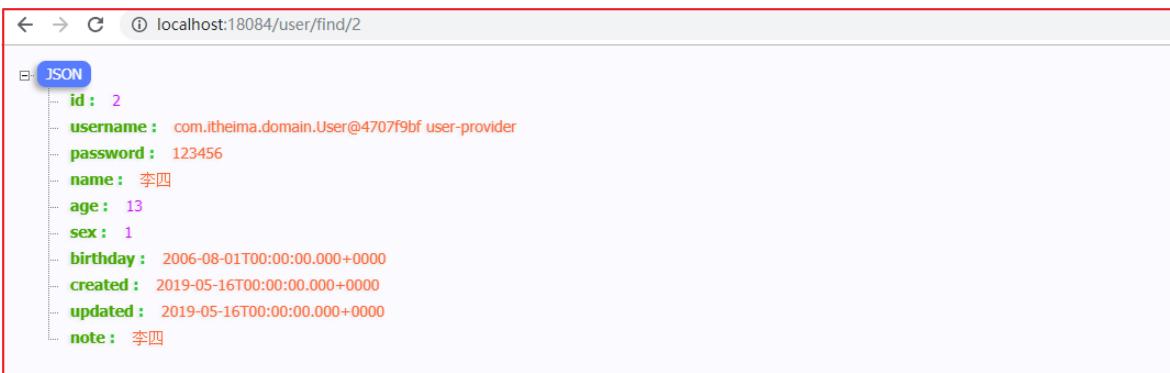


通过网关配置一个路由功能，用户访问网关的时候，如果用户请求的路径是以 /user 开始，则路由到 user-provider 服务去，修改 application.yml 配置即可实现，配置如下：

```
spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          uri: http://localhost:18081
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/user/**
```

启动 GatewayApplication 测试

访问 <http://localhost:18084/user/find/2> 会访问 user-provider 服务，效果如下：



2.2.4 动态路由

```

spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          uri: http://localhost:18081   写死了
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/user/**

```

刚才路由规则中，我们把路径对应服务地址写死了！如果服务提供者集群的话，这样做不合理。应该是根据服务名称，去Eureka注册中心查找服务对应的所有实例列表，然后进行动态路由！

修改映射配置：通过服务名称获取：

修改application.yml

因为已经配置了Eureka客户端，可以从Eureka获取服务的地址信息，修改application.yml文件如下：

```

spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。
          #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
          uri: lb://user-provider
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/user/**

```

上图代码如下：

```

spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。
          #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
          uri: lb://user-provider
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/user/**

```

路由配置中uri所用的协议为lb时，gateway将把user-provider解析为实际的主机和端口，并通过Ribbon进行负载均衡。

2.2.5 过滤器

过滤器作为Gateway的重要功能。常用于请求鉴权、服务调用时长统计、修改请求或响应header、限流、去除路径等等...

2.2.5.1 过滤器的分类

默认过滤器：出厂自带，实现好了拿来就用，不需要实现

全局默认过滤器

局部默认过滤器

自定义过滤器：根据需求自己实现，实现后需配置，然后才能用哦。

全局过滤器：作用在所有路由上。

局部过滤器：配置在具体路由下，只作用在当前路由上。

默认过滤器几十个，常见如下：

过滤器名称	说明
AddRequestHeader	对匹配上的请求加上Header
AddRequestParameters	对匹配上的请求路由
AddResponseHeader	对从网关返回的响应添加Header
StripPrefix	对匹配上的请求路径去除前缀

详细说明官方[链接](#)

2.2.5.2 默认过滤器配置

默认过滤器有两个：全局默认过滤器和局部默认过滤器。

全局过滤器：对输出响应头设置属性

对输出的响应设置其头部属性名称为X-Response-Default-MyName,值为itheima

修改配置文件

```
spring:  
  cloud:  
    gateway:  
      # 配置全局默认过滤器  
      default-filters:  
        # 往响应过滤器中加入信息  
        - AddResponseHeader=X-Response-Default-MyName,itheima
```

查看浏览器响应头信息!

The screenshot shows the Network tab of a browser's developer tools. A request to 'http://localhost:18084/user/find/2' is selected. In the Response Headers section, the 'X-Response-Default-MyName' header is highlighted with a red box, containing the value 'itheima'.

局部过滤器：通过局部默认过滤器，修改请求路径。局部过滤器在这里介绍两种：添加路径前缀、去除路径前缀。

第一：添加路径前缀：

在gateway中可以通过配置路由的过滤器PrefixPath 实现映射路径中的前缀

配置请求地址添加路径前缀过滤器

```
spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
  gateway:
    routes:
      #id唯一标识，可自定义
      - id: user-service-route
        #路由的服务地址
        #uri: http://localhost:18081
        #lb协议表示从Eureka注册中心获取服务请求地址
        #user-provider访问的服务名称。
        #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
        uri: lb://user-provider
        # 路由拦截的地址配置（断言）
        predicates:
          - Path=/**
        filters:
          - PrefixPath=/user
    default-filters:
      - AddResponseHeader=X-Response-Default-MyName, itheima
```

上图配置如下：

```
spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
  gateway:
    routes:
      #id唯一标识，可自定义
      - id: user-service-route
        #路由的服务地址
        #uri: http://localhost:18081
        #lb协议表示从Eureka注册中心获取服务请求地址
        #user-provider访问的服务名称。
        #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
        uri: lb://user-provider
        # 路由拦截的地址配置（断言）
        predicates:
          - Path=/**
```

```

filters:
  # 请求地址添加路径前缀过滤器
  - PrefixPath=/user

default-filters:
  - AddResponseHeader=X-Response-Default-MyName,itheima

```

路由地址信息：

配置	访问地址	路由地址
PrefixPath=/user	http://localhost:18084/find/2	http://localhost:18081/user/find/2

第二：去除路径前缀：

在gateway中通过配置路由过滤器StripPrefix，实现映射路径中地址的去除。通过StripPrefix=1来指定路由要去掉的前缀个数。如：路径/api/user/1将会被路由到/user/1。

配置去除路径前缀过滤器

```

spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。
          #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
          uri: lb://user-provider
          # 路由拦截的地址配置（断言）
          predicates:
            - Path=/**

          filters:
            # 请求地址添加路径前缀过滤器
            #- PrefixPath=/user
            # 去除路径前缀过滤器
            - StripPrefix=1          去掉请求地址第一个/对应的路径， http://localhost:18084/api/user/find/2 路由地址 http://localhost:18081/user/find/2
      default-filters:
        - AddResponseHeader=X-Response-Default-MyName,itheima

```

上图配置如下：

```

spring:
  application:
    # 应用名
    name: api-gateway
  cloud:
    gateway:
      routes:
        #id唯一标识，可自定义
        - id: user-service-route
          #路由的服务地址
          #uri: http://localhost:18081
          #lb协议表示从Eureka注册中心获取服务请求地址
          #user-provider访问的服务名称。

```

```

#路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务
uri: lb://user-provider
# 路由拦截的地址配置（断言）
predicates:
    - Path=/**
filters:
    # 请求地址添加路径前缀过滤器
    #- PrefixPath=/user
    # 去除路径前缀过滤器
    - StripPrefix=1
default-filters:
    - AddResponseHeader=X-Response-Default-MyName,itheima

```

路由地址信息：

配置	访问地址	路由地址
StripPrefix=1	http://localhost:18084/api/user/find/2	http://localhost:18081/user/find/2
StripPrefix=2	http://localhost:18084/api/r/user/find/2	http://localhost:18081/user/find/2

2.2.5.3 自定义过滤器案例

自定义过滤器也有两个：全局自定义过滤器，和局部自定义过滤器。

自定义全局过滤器的案例，自定义局部过滤器的案例。

自定义全局过滤器的案例：模拟登陆校验。

基本逻辑：如果请求中有Token参数，则认为请求有效放行，如果没有则拦截提示授权无效

2.2.5.3.1 全局过滤器自定义：

实现步骤：

1. 在gateway-service工程编写全局过滤器类GlobalFilter,ordered
2. 编写业务逻辑代码
3. 访问接口测试，加token和不加token。

实现过程：

在 gateway-service 中创建 com.itheima.filter.LoginGlobalFilter 全局过滤器类,代码如下：

```

@Component
public class LoginGlobalFilter implements GlobalFilter, ordered {

    /**
     * 过滤拦截
     * @param exchange
     * @param chain
     */
    @Override
    public void filter(ServerWebExchange exchange, NextFunction<ServerWebExchange> chain) {
        // ...
    }
}

```

```

    * @return
    */
@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
    //获取请求参数
    String token = exchange.getRequest().getQueryParams().getFirst("token");

    //如果token为空, 则表示没有登录
    if(StringUtils.isEmpty(token)){
        //没登录, 状态设置403
        exchange.getResponse().setStatusCode(HttpStatus.PAYLOAD_TOO_LARGE);
        //结束请求
        return exchange.getResponse().setComplete();
    }

    //放行
    return chain.filter(exchange);
}

/**
 * 定义过滤器执行顺序
 * 返回值越小, 越靠前执行
 * @return
 */
@Override
public int getOrder() {
    return 0;
}
}

```

测试：不携带token <<http://localhost:18084/api/user/find/2>> 效果如下：



测试：携带token <<http://localhost:18084/api/user/find/2?token=abc>> 此时可以正常访问。

2.2.5.3.2 局部过滤器定义

自定义局部过滤器，该过滤器在控制台输出配置文件中指定名称的请求参数及参数的值,以及判断是否携带请求中参数,打印。

实现步骤：

1. 在gateway-service中编写MyParamGatewayFilterFactory类
2. 实现业务代码：循环请求参数中是否包含name，如果包含则输出参数值，并打印在第三步配置的参数名和值
3. 修改配置文件
4. 访问请求测试，带name参数

实现过程：

在gateway_service中编写MyParamGatewayFilterFactory类

```

@Component
public class MyParamGatewayFilterFactory extends
AbstractNameValueGatewayFilterFactory {
    /**
     * 处理过程 默认需要在配置配置文件中配置 NAME ,VALUE
     *
     * @param config
     * @return
     */
    public GatewayFilter apply(NameValueConfig config) {

        return new GatewayFilter() {
            // - MyParam=name,value
            @Override
            public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
                String name111 = config.getName(); //获取参数名name的值
                String value111 = config.getValue(); //获取参数名value的值
                System.out.println("获取配置中的参数的NAME值：" + name111);
                System.out.println("获取配置中的参数的VALUE值：" + value111);
                //获取参数值
                String name =
exchange.getRequest().getQueryParams().getFirst("name");
                if (!StringUtils.isEmpty(name)) {
                    System.out.println("哈哈：" + name);
                }
                //添加到头信息或者作为参数传递等等.
                return chain.filter(exchange);
            }
        };
    }
}

```

修改application.yml配置文件

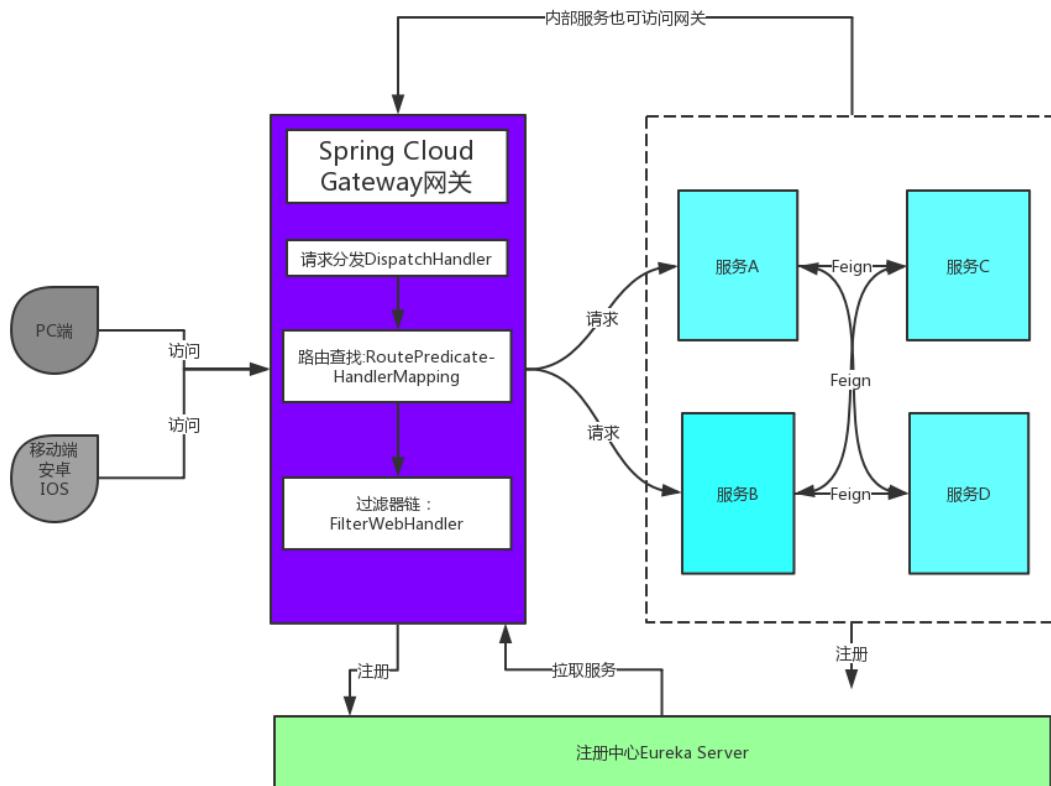
```

1   server:
2     port: 8083      #端口号
3
4   spring:
5     application:
6       name: gateway
7
8   cloud:
9
10  gateway:
11    routes:
12      - id: host_router
13        uri: http://localhost:8080
14        predicates:
15          - Path=/user/**
16
17        filters:
18          - MyParam=name, itheima
19
20    default-filters:
21      - AddResponseHeader=X-Response-Default-Foo, Default-Bar
22
23  eureka:

```

测试访问，检查后台是否输出name和itheima；访问`<http://localhost:18084/api/user/find/2?name=itheima&tomen=aaa>`会输出。

2.2.6 微服务架构加入Gateway后



- 不管是来自客户端的请求，还是服务内部调用。一切对服务的请求都可经过网关。
- 网关实现鉴权、动态路由等等操作。

- Gateway就是我们服务的统一入口

2.3 小结

- 网关的作用

1. 为微服务架构提供一种简单有效统一的API路由管理方式
2. 可以在网关中实现微服务鉴权、安全控制、请求监控、限流

- 会配置动态路由

使用**lb**配置，能根据服务名字动态请求。

- 会配置过滤器

默认过滤器：**default-filters**：

- 能自定义全局过滤器

编写过滤器类，实现**GlobalFilter**和**Ordered**，在**filter**方法中实现过滤。

3 配置中心 Spring Cloud Config

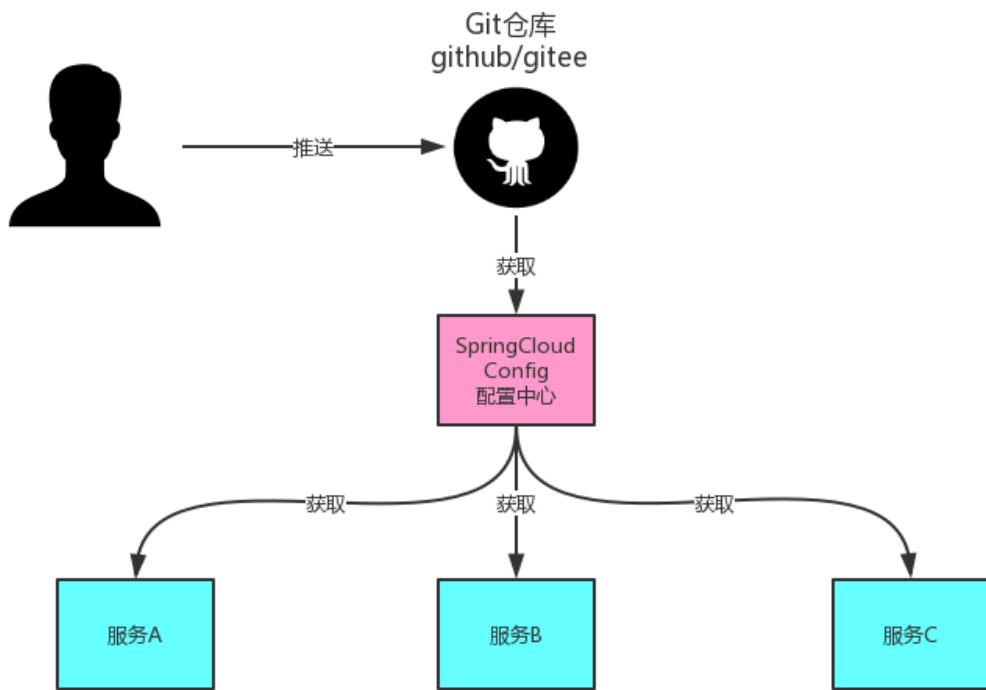
3.1 目标

- 了解配置中心的作用
- 能配置Git仓库
- 能搭建配置中心
- 修改微服务，从配置中心获取修改的配置

3.2 讲解

3.2.1 Config简介

分布式系统中，由于服务数量非常多，配置文件分散在不同微服务项目中，管理极其不方便。为了方便配置文件集中管理，需要分布式配置中心组件。在Spring Cloud中，提供了Spring Cloud Config，它支持配置文件放在配置服务的本地，也支持配置文件放在远程仓库Git(GitHub、码云)。配置中心本质上是一个微服务，同样需要注册到Eureka服务中心！



【配置中心的架构图】

3.2.2 Git配置管理

3.2.2.1 远程Git仓库

- 知名的Git远程仓库有国外的GitHub和国内的码云(gitee);
- GitHub主服务在外网，访问经常不稳定，如果希望服务稳定，可以使用码云；
- 码云访问地址：<http://gitee.com>

测试地址：

<https://gitee.com/sklll/config.git>

3.2.2.2 创建远程仓库

首先使用码云上的git仓库需要先注册账户

账户注册完成，然后使用账户登录码云控制台并创建公开仓库



配置仓库名称和路径

新建仓库

仓库名称 <input checked="" type="text"/> config	仓库名字 <input type="text"/>
归属 <input type="text"/> skill	路径 <input type="text"/> config
仓库地址: https://gitee.com/skill/config	
仓库介绍 非必填 <input type="text"/>	
是否开源 <input type="radio"/> 私有 <input checked="" type="radio"/> 公开 <input type="button" value="选择公开"/>	
任何人都可以访问该仓库的代码和其他任何形式的资源	
选择语言 <input type="text"/> Java	添加 .gitignore <input type="text"/> 请选择 .gitignore 模板
添加开源许可证 <input type="text"/> 请选择开源许可证	
<input checked="" type="checkbox"/> 使用 README 文件初始化这个仓库 <input type="checkbox"/> 使用 ISSUE 模板文件初始化这个仓库 <input type="checkbox"/> 使用 Pull Request 模板文件初始化这个仓库	
选择分支模型 (仓库初始化后将根据所选分支模型创建分支) <input type="text"/> 单分支模型 (只创建 master 分支)	
<input type="button" value="导入已有仓库"/>	
<input type="button" value="创建"/>	

尝试码云企业版?

- 专业研发管理平台
- 有序规划和管理软件研发全流程

与他们一起提升研发效能

已有超过 60000 企业客户



[了解更多](#)

[码云企业版介绍](#)

[社区版与企业版功能对比](#)

3.2.2.3 创建配置文件

在新建的仓库中创建需要被统一配置管理的配置文件

skill / config Java

代码 Issues 0 Pull Requests 0 附件 0 Wiki 0 统计 DevOps 服务 管理

暂无描述

7 次提交	1 个分支	0 个标签	0 个发行版	1 位贡献者
master <input type="button" value="+ Pull Request"/>	<input type="button" value="+ Issue"/>	<input type="button" value="文件"/>	<input type="button" value="Web IDE"/>	<input type="button" value="挂件"/>
<input type="button" value="克隆/下载"/>				
skill 最后提交于 6 分钟前 <input type="button" value="更新"/>	新建文件 <input type="text" value="av.yml"/>			
README.en.md	新建文件夹 <input type="text" value="initial commit"/>	1小时前		
README.md	上传文件 <input type="text" value="initial commit"/>	1小时前		

文件命名有规则:

配置文件的命名方式: `{application}-{profile}.yml` 或 `{application}-{profile}.properties`
application 为应用名称
profile 用于区分开发环境 `dev`, 测试环境 `test`, 生产环境 `pro` 等
 开发环境 `user-dev.yml`
 测试环境 `user-test.yml`
 生产环境 `user-pro.yml`

创建一个 `user-provider-dev.yml` 文件

将 `user-provider` 工程里的配置文件 `application.yml` 内容复制进去。

```
4 datasource:
5   driver-class-name: com.mysql.cj.jdbc.Driver
6   username: root
7   password: itcast
8   url: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
9 application:
10  name: user-provider #服务的名字
11 #指定eureka服务地址
12 eureka:
13   client:
14     service-url:
15       # EurekaServer的地址
16       defaultZone: http://localhost:7001/eureka
17       #每隔30秒获取服务中心列表, (只读备份)
18       registry-fetch-interval-seconds: 30
19   instance:
20     #指定IP地址
21     ip-address: 127.0.0.1
22     #访问服务的时候, 推荐使用IP
23     prefer-ip-address: true
24     #租约到期, 服务时效时间, 默认值90秒, 低于该时间无效
25     lease-expiration-duration-in-seconds: 10
26     #租约续约间隔时间, 默认30秒
27     lease-renewal-interval-in-seconds: 5
```

创建完user-provider-dev.yml配置文件之后，gitee中的仓库如下：

暂无描述

7 次提交 1 个分支 0 个标签 0 个发行版 1 位贡献者

master + Pull Request + Issue 文件 Web IDE 挂件 克隆/下载

S skll 最后提交于 10分钟前 更新 user-provider-dev.yml

文件	操作	时间
README.en.md	Initial commit	1小时前
README.md	Initial commit	1小时前
user-provider-dev.yml	更新 user-provider-dev.yml	10分钟前

3.2.3 搭建配置中心微服务

实现步骤：

1. 创建配置中心SpringBoot项目config_server
2. 配置坐标依赖
3. 启动类添加开启配置中心服务注解
4. 配置服务中心application.yml文件
5. 启动测试

实现过程：

(1) 创建工程

工程坐标

```
<artifactId>config-server</artifactId>
<groupId>com.itheima</groupId>
<version>1.0-SNAPSHOT</version>
```

(2)pom.xml依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <artifactId>springcloud-parent</artifactId>
        <groupId>com.itheima</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>
    <artifactId>config-server</artifactId>

    <dependencies>
        <!-- Eureka客户端 -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
        </dependency>
        <!--配置中心-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-config-server</artifactId>
        </dependency>
    </dependencies>
</project>
```

(3)创建启动类

在 config-server 工程中创建启动类 com.itheima.ConfigServerApplication,代码如下:

```
@SpringBootApplication
@EnableDiscoveryClient//开启Eureka客户端发现功能
@EnableConfigServer//开启配置服务支持
public class ConfigServerApplication {

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

(4)application.yml配置文件

```
# 注释版本
server:
  port: 18085 # 端口号
spring:
  application:
    name: config-server # 应用名
  cloud:
    config:
```

```

server:
  git:
    # 配置gitee的仓库地址
    uri: https://gitee.com/sk1111/config.git
# Eureka服务中心配置
eureka:
  client:
    service-url:
      # 注册Eureka Server集群
      defaultZone: http://127.0.0.1:7001/eureka
# com.itheima 包下的日志级别都为Debug
logging:
  level:
    com: debug

```

注意：上述spring.cloud.config.server.git.uri是在码云创建的仓库地址

(5)测试

启动 config-server，访问 `<http://localhost:18085/user-provider-dev.yml>`，效果如下：

```

← → ⌂ ⓘ localhost:18085/user-provider-dev.yml

eureka:
  client:
    registry-fetch-interval-seconds: 30
    service-url:
      defaultZone: http://localhost:7001/eureka
  instance:
    ip-address: 127.0.0.1
    lease-expiration-duration-in-seconds: 10
    lease-renewal-interval-in-seconds: 5
    prefer-ip-address: true
server:
  port: 18081
spring:
  application:
    name: user-provider
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: itcast
    url: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
    username: root

```

可以看到码云上的文件数据，并且可以在gitee上修改user-dev.yml，然后刷新上述测试地址也能及时更新数据

3.2.4 服务去获取配置中心配置

目标：改造user-provider工程，配置文件不再由微服务项目提供，而是从配置中心获取。

实现步骤：

1. 添加配置中心客户端启动依赖
2. 修改服务提供者的配置文件
3. 启动服务
4. 测试效果

实现过程：

(1)添加依赖

```
<!--spring cloud 配置中心-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

(2)修改配置

删除user-provider工程的application.yml文件

创建user-provider工程bootstrap.yml配置文件，配置内容如下

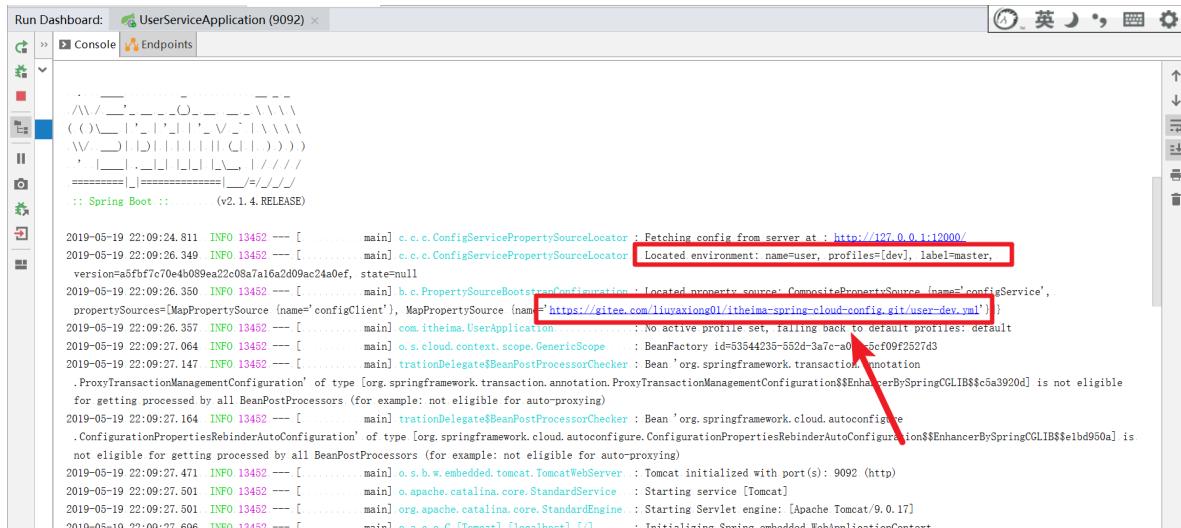
```
# 注释版本
spring:
  cloud:
    config:
      name: user-provider # 与远程仓库中的配置文件的application保持一致,
{application}-{profile}.yml
      profile: dev # 远程仓库中的配置文件的profile保持一致
      label: master # 远程仓库中的版本保持一致
      discovery:
        enabled: true # 使用配置中心
        service-id: config-server # 配置中心服务id
#向Eureka服务中心集群注册服务
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:7001/eureka
```

关于application.yml和bootstrap.yml文件的说明：

- bootstrap.yml文件是SpringBoot的默认配置文件，而且其加载时间相比于application.yml更早。
- bootstrap.yml和application.yml都是默认配置文件，但定位不同
 - bootstrap.yml可以理解成系统级别的一些参数配置，一般不会变动
 - application.yml用来定义应用级别的参数
- 搭配spring-cloud-config使用application.yml的配置可以动态替换。
- bootstrap.yml相当于项目启动的引导文件，内容相对固定
- application.yml文件是微服务的常规配置参数，变化比较频繁

启动测试：

启动服务中心、配置中心、用户中心user_service

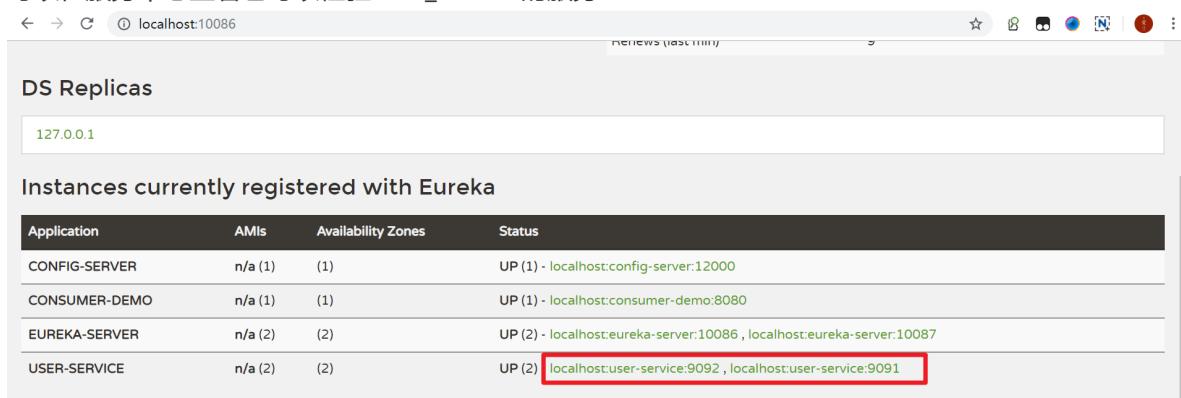


The screenshot shows the application logs in the terminal. A red box highlights a log entry where the application successfully retrieves configuration from the server at `http://127.0.0.1:12000/`. Another red box highlights the URL `https://gitcode.net/gitee.com/liuyaxiong01/itheima-spring-cloud-config.git/user-dev.yaml` which is used as a property source.

```
2019-05-19 22:09:24.811 INFO 13452 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://127.0.0.1:12000/
2019-05-19 22:09:26.349 INFO 13452 --- [           main] c.c.c.ConfigServicePropertySourceLocator : Located environment: name=user, profiles=[dev], label=master, version=a5bf7c70e4b099ea22c08a716e2d09ac24a0ef, state=null
2019-05-19 22:09:26.359 INFO 13452 --- [           main] b.c.PropertySourceBootstrapConfiguration : Located property source: CompositePropertySource [name='configService', propertySources=[MapPropertySource [name='configClient'], MapPropertySource [name='https://gitcode.net/gitee.com/liuyaxiong01/itheima-spring-cloud-config.git/user-dev.yaml']]}
2019-05-19 22:09:26.359 INFO 13452 --- [           main] com.itheima.UserServiceApplication : No active profile set, falling back to default profiles: default
2019-05-19 22:09:27.064 INFO 13452 --- [           main] o.s.cloud.context.scope.GenericScope : BeanFactory id=53544235-552d-3a7c-a055-5cf09f2527d3
2019-05-19 22:09:27.147 INFO 13452 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration' of type [org.springframework.transaction.annotation.ProxyTransactionManagementConfiguration$$EnhancerBySpringCGLIB$$c5a3920d] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2019-05-19 22:09:27.164 INFO 13452 --- [           main] trationDelegate$BeanPostProcessorChecker : Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebindAutoConfiguration' of type [org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebindAutoConfiguration$$EnhancerBySpringCGLIB$$e1bd950a] is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)
2019-05-19 22:09:27.471 INFO 13452 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 9092 (http)
2019-05-19 22:09:27.501 INFO 13452 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2019-05-19 22:09:27.501 INFO 13452 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.17]
2019-05-19 22:09:27.604 INFO 13452 --- [           main] o.a.n.u.s.t.s.s.TomcatEmbeddedWebContainer : Initializing Service embedded WebContainer
```

如果启动没报错，其实已经使用上配置中心内容了

可以在服务中心查看也可以检验user_service的服务



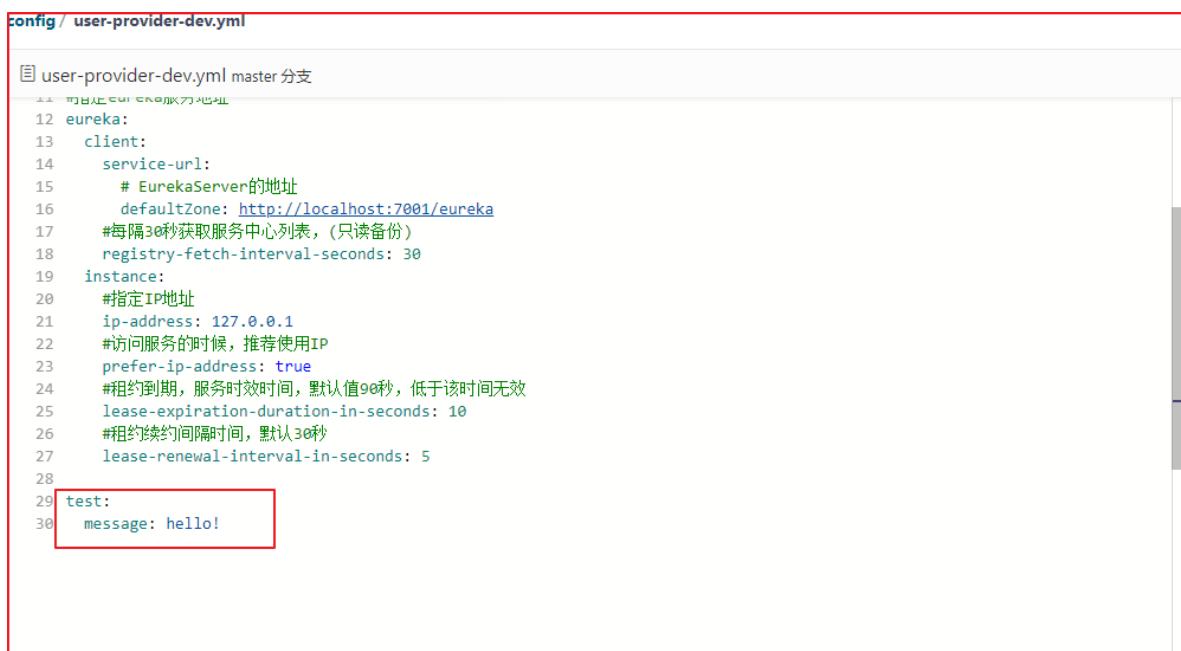
The screenshot shows the Eureka UI displaying registered instances. A red box highlights the instance `USER-SERVICE` with the URL `localhost:user-service:9092,localhost:user-service:9091`.

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - localhost:config-server:12000
CONSUMER-DEMO	n/a (1)	(1)	UP (1) - localhost:consumer-demo:8080
EUREKA-SERVER	n/a (2)	(2)	UP (2) - localhost:eureka-server:10086,localhost:eureka-server:10087
USER-SERVICE	n/a (2)	(2)	UP (2) <code>localhost:user-service:9092,localhost:user-service:9091</code>

3.2.5 配置中心存在的问题

(1)修改码云配置文件

修改在码云上的user-provider-dev.yml文件，添加一个属性test.message,如下操作：



The screenshot shows the GitHub code editor with the file `config/user-provider-dev.yml`. A red box highlights the new configuration section `test:` with the value `message: hello!`.

```
config / user-provider-dev.yml
@ user-provider-dev.yml master 分支
12 eureka:
13   client:
14     service-url:
15       # EurekaServer的地址
16       defaultZone: http://localhost:7001/eureka
17       #每隔30秒获取服务中心列表, (只读备份)
18       registry-fetch-interval-seconds: 30
19     instance:
20       #指定IP地址
21       ip-address: 127.0.0.1
22       #访问服务的时候, 推荐使用IP
23       prefer-ip-address: true
24       #租约到期, 服务时效时间, 默认值90秒, 低于该时间无效
25       lease-expiration-duration-in-seconds: 10
26       #租约续约间隔时间, 默认30秒
27       lease-renewal-interval-in-seconds: 5
28
29 test:
30   message: hello!
```

(2)读取配置文件数据

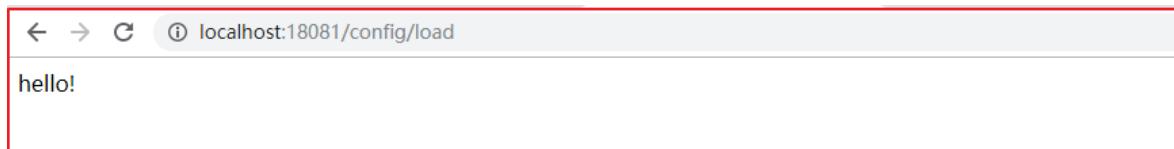
在 user-provider 工程中创建一个 com.itheima.controller.LoadConfigController 读取配置文件信息，代码如下：

```
@RestController
@RequestMapping(value = "/config")
public class LoadConfigController {

    @Value("${test.message}")
    private String msg;

    /**
     * 响应配置文件中的数据
     * @return
     */
    @RequestMapping(value = "/load")
    public String load(){
        return msg;
    }
}
```

启动运行 user-provider，访问 <http://localhost:18081/config/load>



修改码云上的配置后，发现项目中的数据仍然没有变化，只有项目重启后才会变化。

3.3 小结

- 配置中心的作用：将各个微服务的配置文件集中到一起进行统一管理。
- 能搭建配置中心

需要在 application.yml 配置文件中指定需要远程更新的仓库地址。

- 修改微服务，从配置中心获取修改的配置

```
创建bootstrap.yml，并在bootstrap.yml中配置
# 注释版本
spring:
  cloud:
    config:
      name: user-provider # 与远程仓库中的配置文件的application保持一致,
{application}-{profile}.yml
      profile: dev # 远程仓库中的配置文件的profile保持一致
      label: master # 远程仓库中的版本保持一致
    discovery:
      enabled: true # 使用配置中心
      service-id: config-server # 配置中心服务id
# 向Eureka服务中心集群注册服务
eureka:
```

```
client:  
  service-url:  
    defaultZone: http://127.0.0.1:7001/eureka
```

4 消息总线 Spring Cloud Bus

SpringCloud Bus，解决上述问题，实现配置自动更新。

注意：SpringCloudBus基于RabbitMQ实现，默认使用本地的消息队列服务，所以需要提前安装并启动 RabbitMQ。

4.1 Bus简介

Bus是用轻量的消息代理将分布式的节点连接起来，可以用于**广播配置文件的更改**或者服务的监控管理。

Bus可以为微服务做监控，也可以实现应用程序之间互相通信。Bus可选的消息代理**RabbitMQ**和 Kafka。

广播出去的配置文件服务会进行本地缓存。

4.2 整合案例

目标：消息总线整合入微服务系统，实现配置中心的配置自动更新。不需要重启微服务。

4.2.1 改造配置中心

改造步骤：

1. 在config-server项目中加入Bus相关依赖
2. 修改application.yml，加入RabbitMQ的配置信息，和暴露触发消息总线地址

实现过程：

(1)引入依赖

修改 config-server 的pom.xml引入依赖：

```
<!--消息总线依赖-->  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-bus</artifactId>  
</dependency>  
<!--RabbitMQ依赖-->  
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  
</dependency>
```

(2)修改application.yml配置文件

修改 config-server 的application.yml，如下配置的rabbit都是默认值，其实可以完全不配置，代码如下：

```

server:
  port: 18085 # 端口号
spring:
  application:
    name: config-server # 应用名
cloud:
  config:
    server:
      git:
        # 配置gitee的仓库地址
        uri: https://gitee.com/skllll/config.git
# rabbitmq的配置信息; 如下配置的rabbit都是默认值, 其实可以完全不配置
rabitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
# 暴露触发消息总线的地址
management:
  endpoints:
    web:
      exposure:
        # 暴露触发消息总线的地址
        include: bus-refresh

```

上图配置如下:

```

# 注释版本
server:
  port: 18085 # 端口号
spring:
  application:
    name: config-server # 应用名
cloud:
  config:
    server:
      git:
        # 配置gitee的仓库地址
        uri: https://gitee.com/skllll/config.git
# rabbitmq的配置信息; 如下配置的rabbit都是默认值, 其实可以完全不配置
rabitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
# 暴露触发消息总线的地址
management:
  endpoints:
    web:
      exposure:
        # 暴露触发消息总线的地址
        include: bus-refresh

# Eureka服务中心配置
eureka:
  client:
    service-url:
      # 注册Eureka Server集群
      defaultZone: http://127.0.0.1:7001/eureka
# com.itheima 包下的日志级别都为Debug
logging:
  level:

```

4.2.2 改造用户服务

改造步骤：

1. 在用户微服务user_service项目中加入Bus相关依赖
2. 修改user_service项目的bootstrap.yml，加入RabbitMQ的配置信息
3. UserController类上加入@RefreshScope刷新配置注解
4. 测试

实现过程：

(1)引入依赖

修改 user-provider 引入如下依赖：

```
<!--消息总线依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-bus</artifactId>
</dependency>
<!--RabbitMQ依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<!--健康监控依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

(2)添加bootstrap.yml文件

在 user-provider 的resources目录下添加bootstrap.yml，添加rabbitmq配置，代码如下：

```
# 注释版本
spring:
  cloud:
    config:
      name: user-provider # 与远程仓库中的配置文件的application保持一致,
{application}-{profile}.yml
      profile: dev # 远程仓库中的配置文件的profile保持一致
      label: master # 远程仓库中的版本保持一致
    discovery:
      enabled: true # 使用配置中心
      service-id: config-server # 配置中心服务id
# rabbitmq的配置信息；如下配置的rabbit都是默认值，其实可以完全不配置
  rabbitmq:
```

```
host: localhost
port: 5672
username: guest
password: guest
#向Eureka服务中心集群注册服务
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:7001/eureka
```

(3)添加刷新配置

修改 user-provider 的 `com.itheima.controller.LoadConfigController`, 添加一个 `@RefreshScope` 注解刷新配置信息, 代码如下:

```
@RestController
@RefreshScope
@RequestMapping(value = "/config")
public class LoadConfigController {

  @Value("${test.message}")
  private String msg;

  /**
   * 响应配置文件中的数据
   * @return
   */
  @RequestMapping(value = "/load")
  public String load() {
    return msg;
  }
}
```

`@RefreshScope`: 用于启用刷新配置文件的信息。

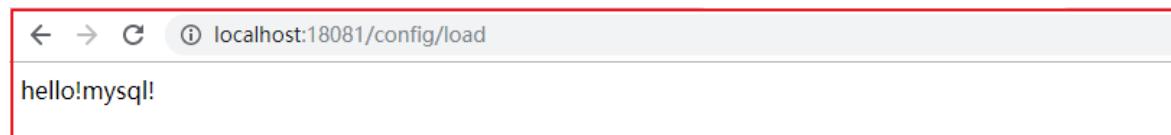
4.3 测试

目标: 当我们修改Git仓库的配置文件, 用户微服务是否能够在不重启的情况下自动更新配置文件信息。

测试步骤:

- (1)启动 `eureka-server`
- (2)启动 `config-server`
- (3)启动 `user-provider`
- (4)访问测试

访问 `<http://localhost:18081/config/load>`,效果如下:



(5)修改码云配置

修改码云的配置，修改后并提交，修改如下：

```
17 #每隔30秒获取服务中心列表, (只读备份)
18 registry-fetch-interval-seconds: 30
19 instance:
20     #指定IP地址
21     ip-address: 127.0.0.1
22     #访问服务的时候, 推荐使用IP
23     prefer-ip-address: true
24     #租约到期, 服务时效时间, 默认值90秒, 低于该时间无效
25     lease-expiration-duration-in-seconds: 10
26     #租约续约间隔时间, 默认30秒
27     lease-renewal-interval-in-seconds: 5
28
29 test:
30     message: hello!mysql!欢迎来到深圳黑马训练营!
```

提交信息

更新 user-provider-dev.yml

扩展信息

此处可填写为什么修改, 做了什么样的修改, 以及开发的思路等更加详细的提交信息。 (相当于 Git Commit message 的 Body)

提交到master 取消

(6)刷新配置

使用Postman以POST方式请求 <http://localhost:18085/actuator/bus-refresh>

POST http://localhost:18085/actuator/bus-refresh

Authorization Headers Body Pre-request Script Tests

TYPE Inherit auth from parent

This request is not inheriting any authorization helper at the moment. Save it in a collection to use the parent's authorization helper.

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Body Cookies (2) Headers (1) Test Results

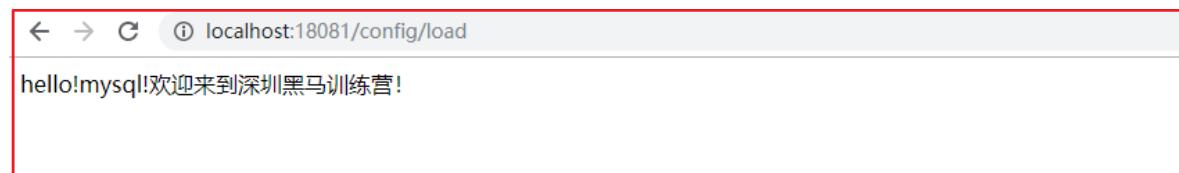
Pretty Raw Preview Text

Status: 204 No Content Time: 4090ms

请求地址中actuator是固定的, bus-refresh对应的是配置中心的config-server中的application.yml文件的配置项include的内容

(7)刷新测试

访问 <<http://localhost:18081/config/load>>,效果如下:



消息总线实现消息分发过程：

- 请求地址访问配置中心的消息总线
- 消息总线接收到请求
- 消息总线向消息队列发送消息
- user-service微服务会监听消息队列
- user-service微服务接到消息队列中消息后
- user-service微服务会重新从配置中心获取最新配置信息

SpringCloud 总架构图

1558286268166