

SpringCloud

Day1

学习目标

1. 能够理解 SpringCloud 作用
2. 能够使用 RestTemplate 发送请求
3. 能够搭建 Eureka[ju'rikə]注册中心
管理服务、监控服务、服务路由
4. 能够使用 Ribbon ['ribən]负载均衡(消费方)
5. 能够使用 Hystrix[hɪst'riks]熔断器
服务降级、防止程序雪崩

1 初识 Spring Cloud

谈起微服务，它其实是种架构方式。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

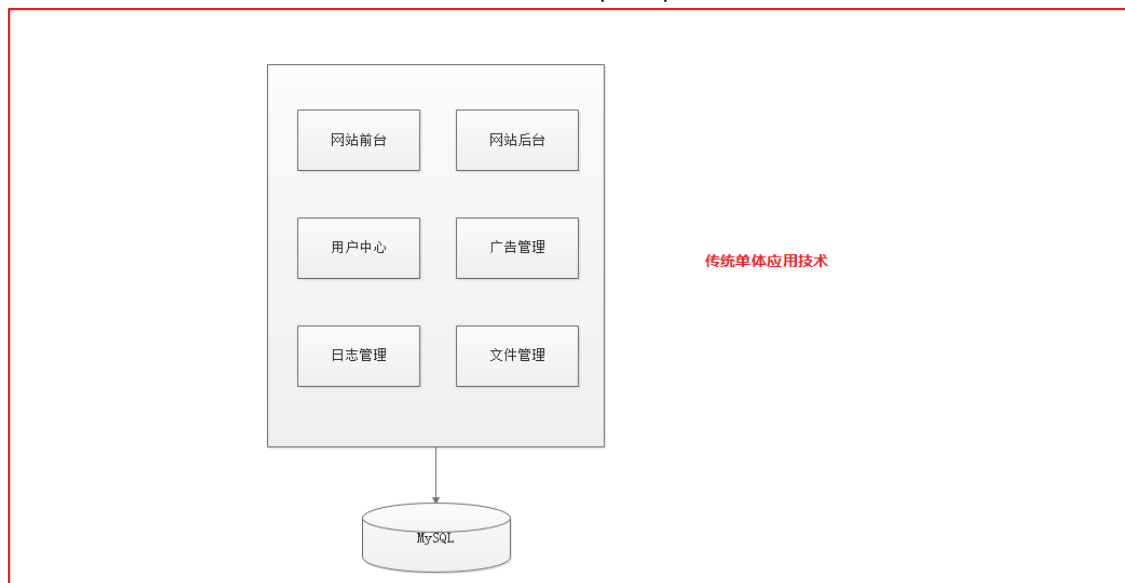
- 了解微服务架构
- 了解 SpringCloud 技术

1.2 讲解

1.2.1 技术架构演变

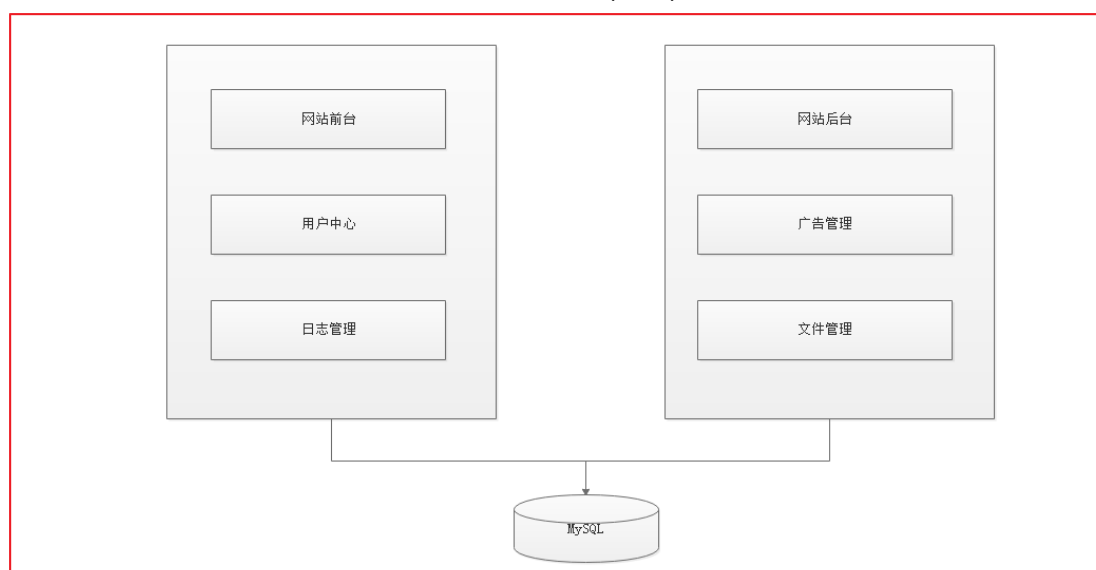
1.2.1.1 单一应用架构(集中式)

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。



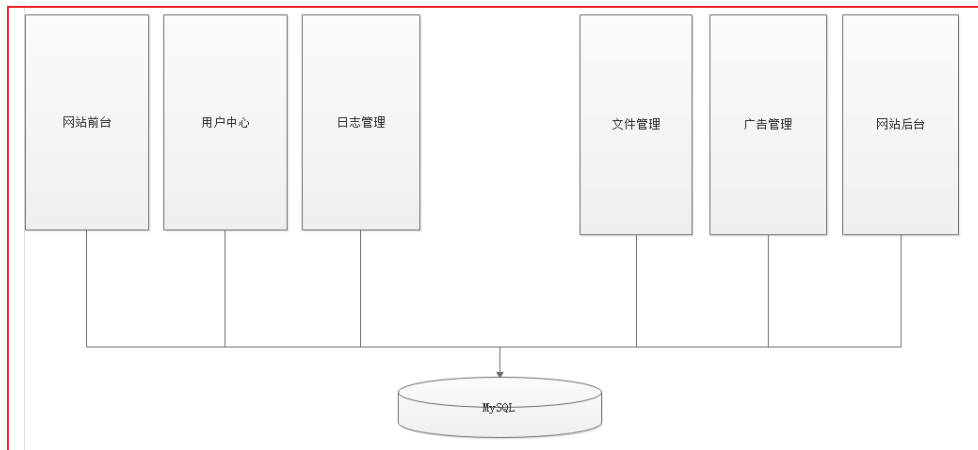
1.2.1.2 垂直应用架构

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。



1.2.1.3 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。



1.2.1.4 面向服务(SOA)架构

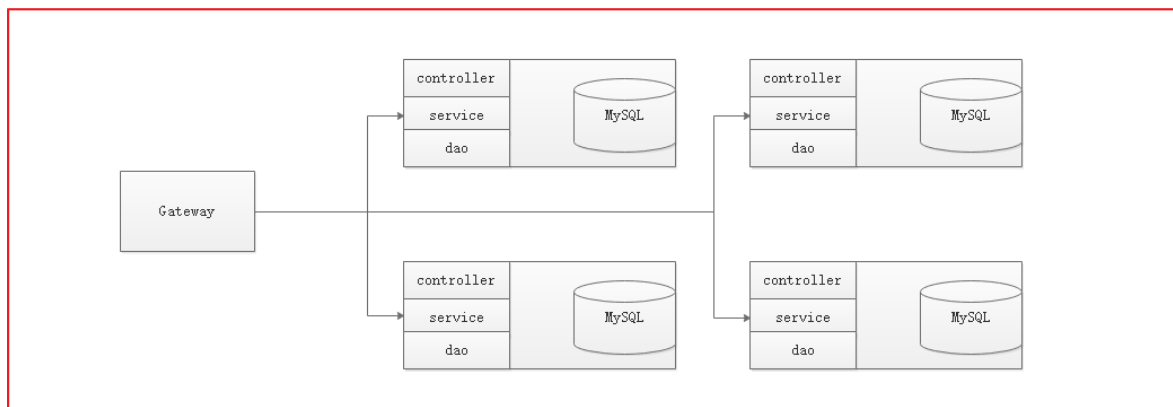
典型代表有两个：流动计算架构和微服务架构：

1.2.1.4.1 流动计算架构(SOA)

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。

1.2.1.4.2 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

1.2.2.1 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。

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

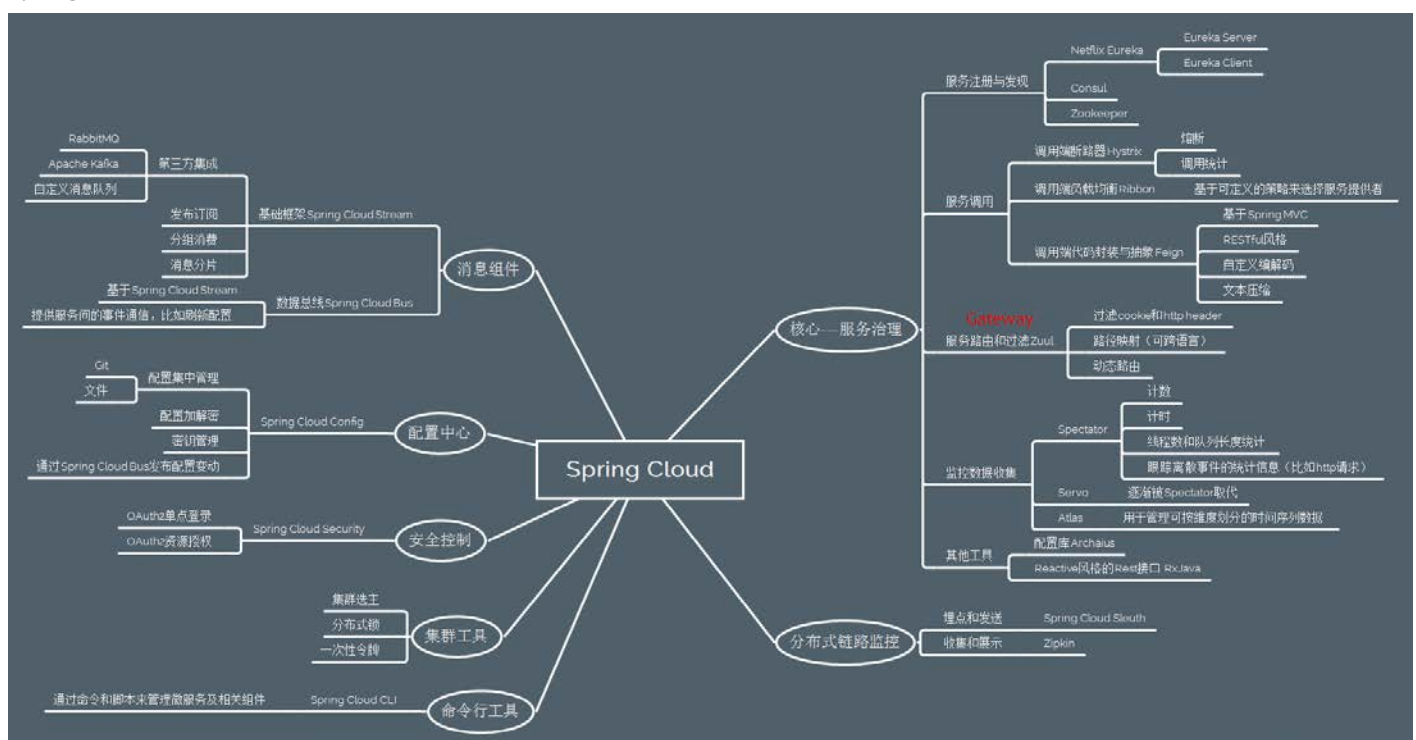
网关：Gateway

配置中心：config

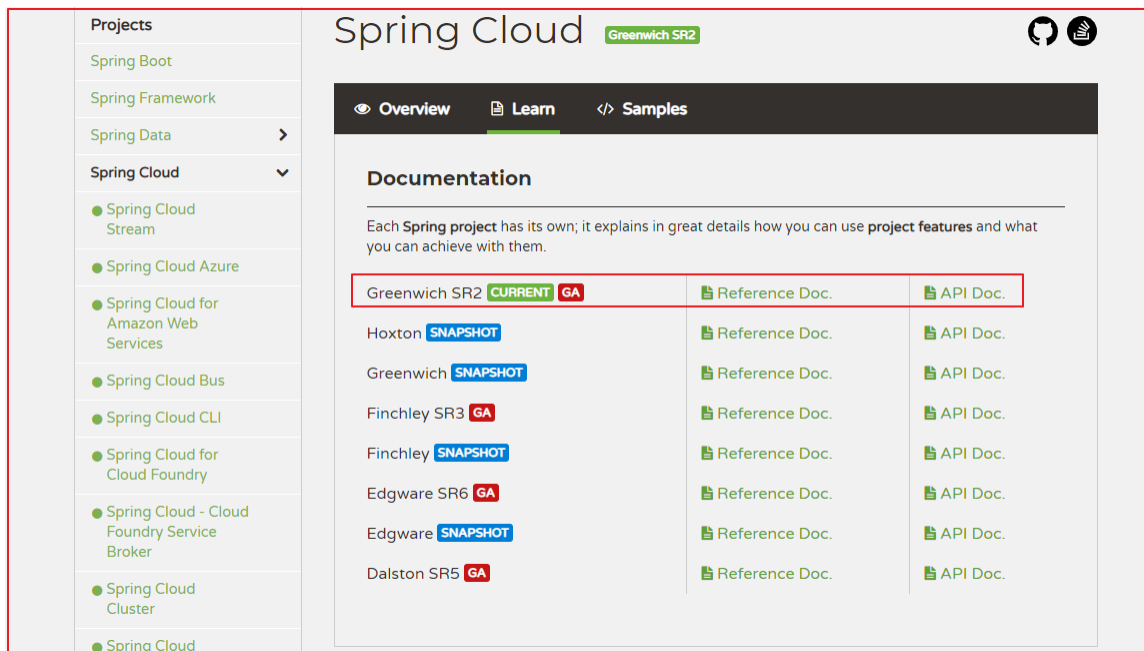
消息总线：Bus

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。



1.2.2.2 Spring Cloud 的版本



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

1.2.2.3 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 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，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。

2.3 Spring 的 RestTemplate

2.3.1 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)

2.3.2 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。

2.3.2.1 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，我们需要 web 启动器。

```
<?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
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.itheima</groupId>
  <artifactId>springcloud_day1_provider</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>springcloud_day1_provider</name>
  <description>Demo project for Spring Boot</description>
```

```
<properties>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

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

</project>
```

2.3.2.2 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
}
```

2.3.2.3 application.properties

```
server.port=18081
```


2.3.2.4 创建 com.itheima.controller.UserController

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

    /**
     * 获取用户列表
     * @return
     */
    @RequestMapping(value = "list")
    public List<User> list() {
        List<User> users = new ArrayList<User>();
        users.add(new User("张三", "深圳", 25));
        users.add(new User("李四", "北京", 26));
        users.add(new User("王五", "上海", 27));
        return users;
    }
}
```

2.3.2.5 启动与测试

启动引导类，浏览器访问：<http://localhost:18081/user/list>



2.3.3 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，我们也需要 web 启动器。

```
<?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
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.1.7.RELEASE</version>
<relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>com.itheima</groupId>
<artifactId>springcloud_day1_resttemplate</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>springcloud_day1_resttemplate</name>
<description>Demo project for Spring Boot</description>

<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

</project>
```

2.3.3.1 在启动引导类中创建 RestTemplate 对象

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

    @Bean
    public RestTemplate getRestTemplate() {
        return new RestTemplate();
    }
}
```

2.3.3.2 测试

在测试类 SpringcloudDay1ResttemplateApplication 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;

    @Test
    public void testRestTemplateGet() {
        String url = "http://localhost:18081/user/list";
        //getForObject(请求地址, 结果解析类型字节码)
        String json = restTemplate.getForObject(url, String.class);
        System.out.println(json);
    }
}
```

运行测试方法，效果如下：

```

  ____  __
 / ___/  / /_  __
/ /   / __/ / / /
/ /___/ /_ / /_/ /
/_____/_/_____/

:: Spring Boot ::      (v2.1.7.RELEASE)

2019-08-22 22:28:29.027 INFO 12976 --- [           main] ingcloudDay1ResttemplateApplicationTests : Starting Springc
2019-08-22 22:28:29.029 INFO 12976 --- [           main] ingcloudDay1ResttemplateApplicationTests : No active profil
2019-08-22 22:28:31.893 INFO 12976 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing Exe
2019-08-22 22:28:32.213 INFO 12976 --- [           main] ingcloudDay1ResttemplateApplicationTests : Started Springc
[{"name":"张三","address":"深圳","age":25}, {"name":"李四","address":"北京","age":26}, {"name":"王五","address":"上海",
2019-08-22 22:28:32.554 INFO 12976 --- [ Thread-2] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down Ex
```

2.4 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:

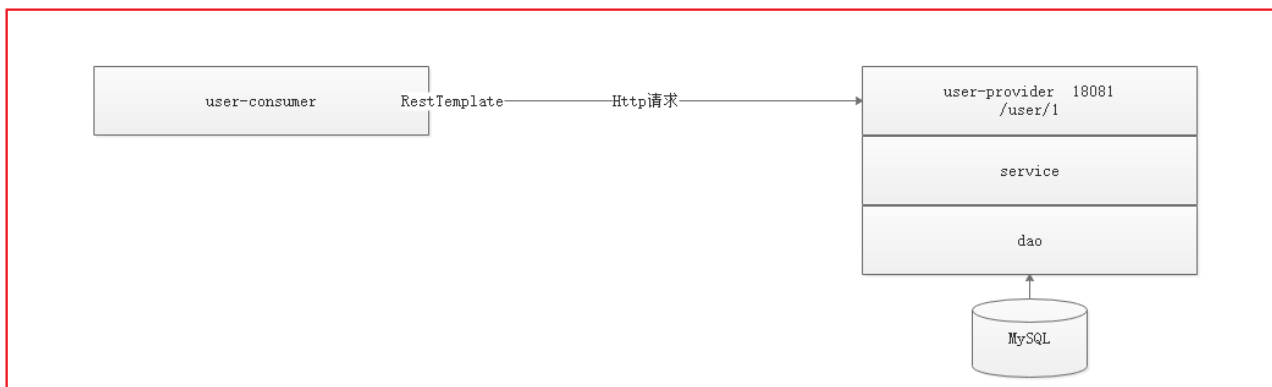
- ① RestTemplate 是 Rest 的 HTTP 客户端模板工具类。
- ② 对基于 Http 的客户端进行封装。
- ③ 实现对象与 JSON 的序列化与反序列化。
- ④ 不限定客户端类型

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

3.1.1 创建父工程

新建一个普通 Maven 父工程 springcloud_parent 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">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>springcloud_parent</artifactId>
    <version>0.0.1-SNAPSHOT</version>
```

```
<!--父工程-->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.7.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>
```

3.1.2 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

3.1.2.1 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');
```

3.1.2.2 新建 user_provider 工程

选中 springcloud-parent 工程->New Modul->Maven，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>0.0.1-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>
```

3.1.2.3 创建 com.itheima.domain.User

```
@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
}
```

3.1.2.4 创建 com.itheima.dao.UserDao

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

3.1.2.5 创建 com.itheima.service.UserService 接口

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

3.1.2.6 创建 com.itheima.service.impl.UserServiceImp

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserDao userDao;

    @Override
    public User findById(Integer id) {
        return userDao.findById(id).get();
    }
}
```


3.1.2.7 创建 com.itheima.controller.UserController

```
@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);
    }
}
```

3.1.2.8 创建 application.yml 配置

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

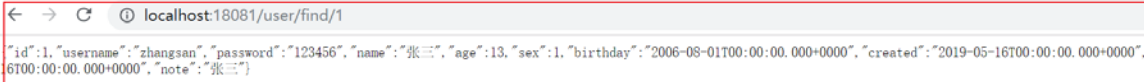
3.1.2.9 启动与测试

创建 com.itheima.UserProviderApplication 启动引导类，并启动

```
@SpringBootApplication
public class UserProviderApplication {
```

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

访问: <http://localhost:18081/user/find/1>



3.1.3 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

3.1.3.1 创建工程 user_consumer

选中 springcloud-parent 工程->New Modul->Maven, 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>0.0.1-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>
```

3.1.3.2 创建 com.itheima.domain.User

可以复制 user_provider 工程中的实体类 User 过来，注意要去掉 JAP 注解，代码略.....

3.1.3.3 创建 com.itheima.UserConsumerApplication

```
@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();
    }
}
```

3.1.3.4 创建 application.yml

```
server:
  port: 18082
```

3.1.3.5 创建 com.itheima.controller.UserController

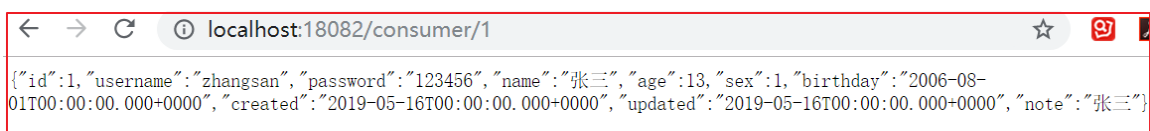
```
@RestController
@RequestMapping(value = "consumer")
public class UserController {

    @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);
    }
}
```

3.1.3.6 启动与测试

启动引导类，访问：<http://localhost:18082/consumer/1>



3.1.4 思考

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

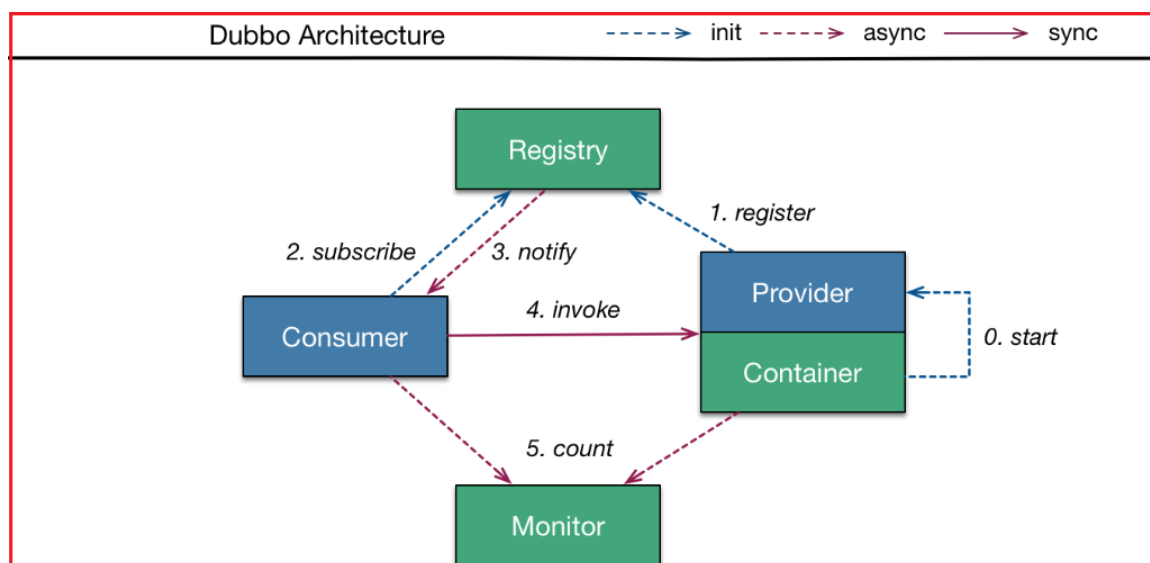
- 服务管理：自动注册与发现、状态监管
- 服务负载均衡
- 熔断器

3.2 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

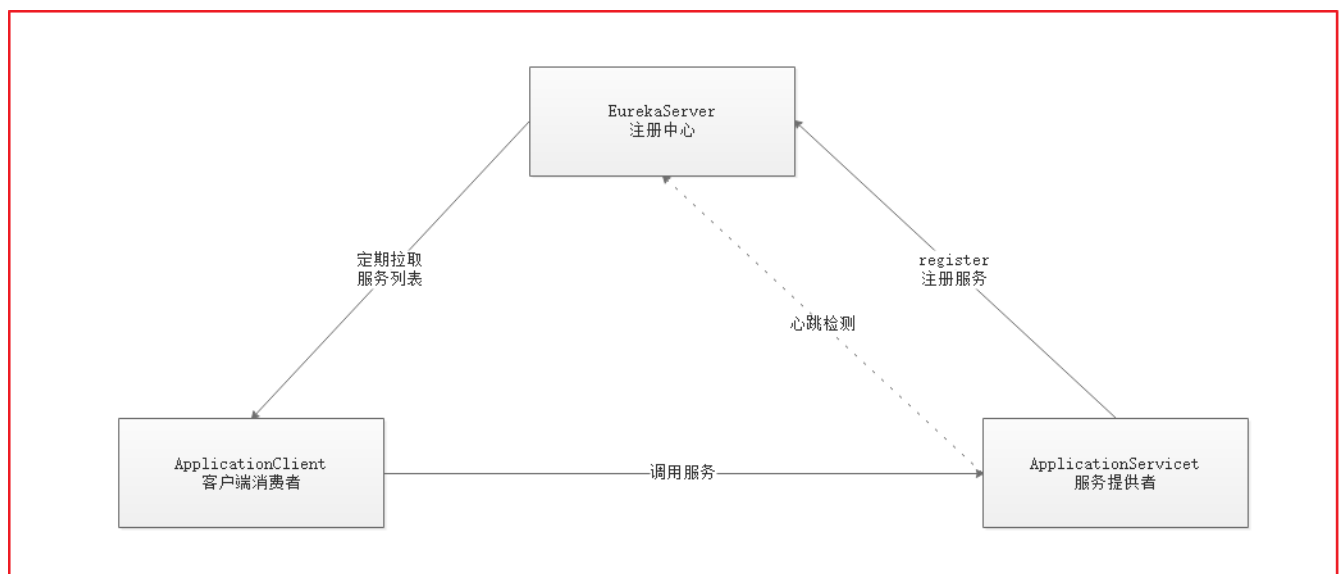
4.2 Eureka 简介

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；

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

4.3 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: 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

4.3.1 搭建 eureka_server 工程

选中 springcloud_parent 工程->New Modul->Maven, 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>0.0.1-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>
```

4.3.1.1 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
```

4.3.1.2 创建启动类

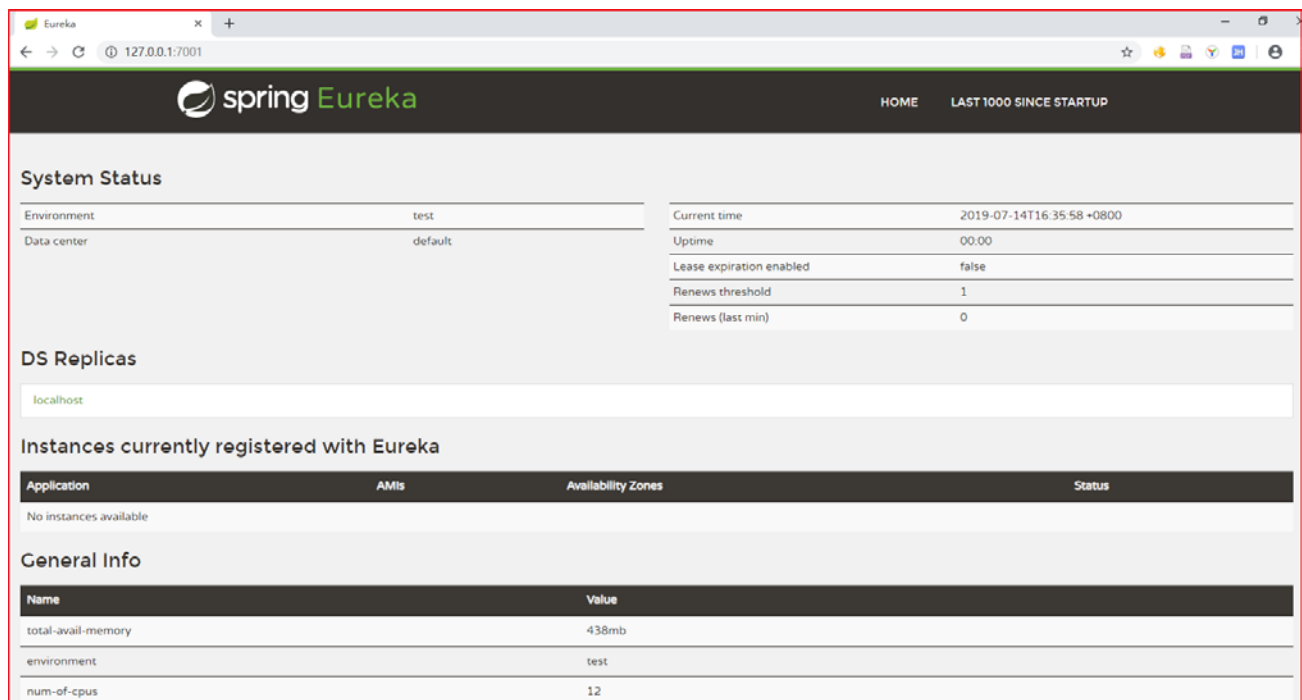
创建 `com.itheima.EurekaServerApplication`,在类上需要添加`@SpringBootApplication`、`@EnableEurekaServer` 两个注解，用于开启 Eureka 服务,代码如下：

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

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

4.3.1.3 启动与测试

启动后，访问 <http://127.0.0.1:7001>，效果如下：



The screenshot shows the Spring Eureka web interface in a browser window. The address bar shows `127.0.0.1:7001`. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'.

System Status

Environment	test	Current time	2019-07-14T16:35:58 +0800
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	1
		Renews (last min)	0

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	438mb
environment	test
num-of-cpus	12

4.3.2 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@EnableDiscoveryClient 或者@EnableEurekaClient

4.3.2.1 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>
```

4.3.2.2 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：

```
server:
  port: 18081
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    username: root
    password: root
    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
```

4.3.2.3 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。

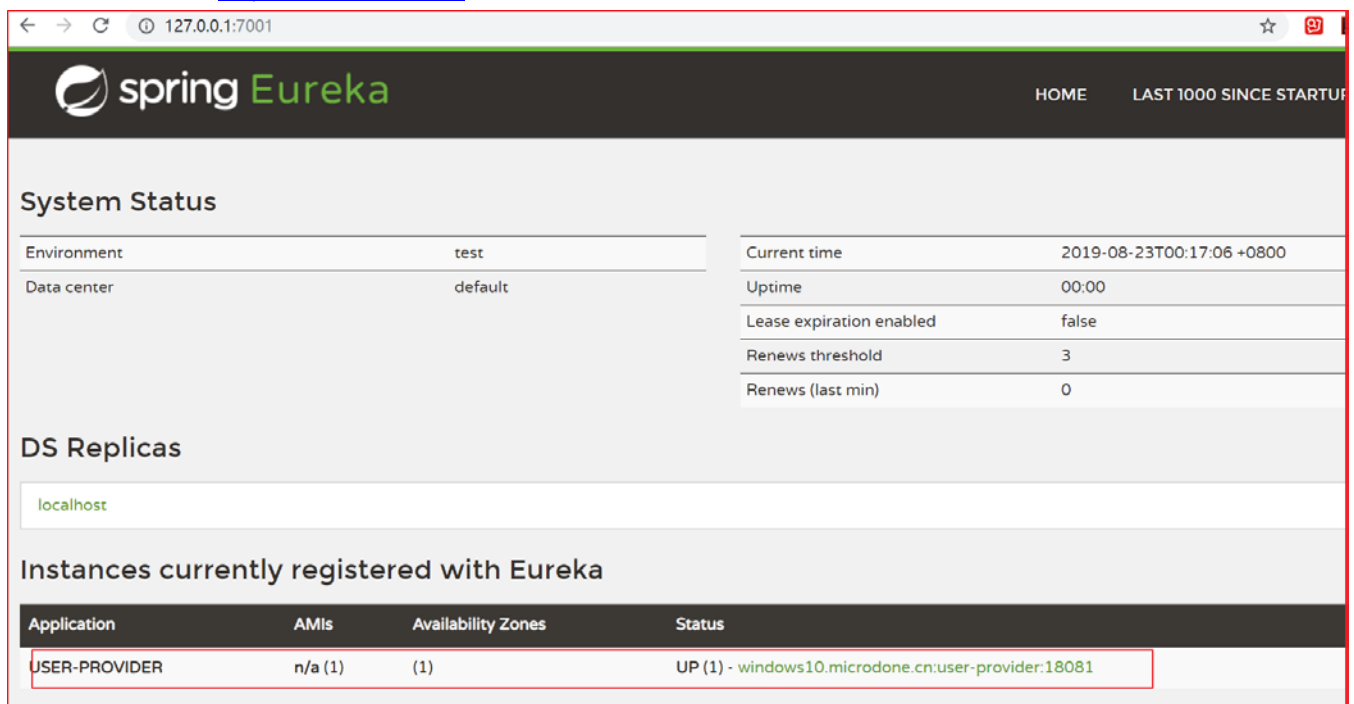
```
@SpringBootApplication
@EnableDiscoveryClient //开启 Eureka 客户端发现功能(推荐使用)
//@EnableEurekaClient //开启 Eureka 客户端发现功能，注册中心只能是 Eureka
public class UserProviderApplication {

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

4.3.2.4 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 web interface in a browser. The address bar shows the URL <http://127.0.0.1:7001>. The page has a dark header with the 'spring Eureka' logo and navigation links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is divided into sections: 'System Status', 'DS Replicas', and 'Instances currently registered with Eureka'. The 'System Status' section contains two tables. The first table lists 'Environment' as 'test' and 'Data center' as 'default'. The second table lists 'Current time' as '2019-08-23T00:17:06 +0800', 'Uptime' as '00:00', 'Lease expiration enabled' as 'false', 'Renews threshold' as '3', and 'Renews (last min)' as '0'. The 'DS Replicas' section shows 'localhost'. The 'Instances currently registered with Eureka' section contains a table with the following data:

Application	AMIs	Availability Zones	Status
USER-PROVIDER	n/a (1)	(1)	UP (1) - windows10.microdone.cn:user-provider:18081

4.3.3 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 @EnableDiscoveryClient 或者 @EnableEurekaClient

4.3.3.1 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>
```

4.3.3.2 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：

```
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
```

4.3.3.3 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：

```
@SpringBootApplication
@EnableDiscoveryClient
public class UserConsumerApplication {

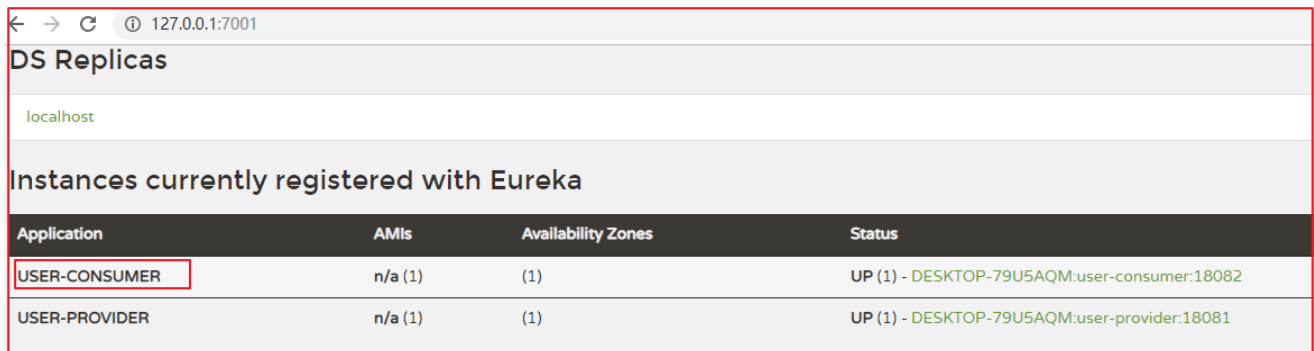
    public static void main(String[] args) {
        SpringApplication.run(UserConsumerApplication.class, args);
    }

    /**
     * 将 RestTemplate 的实例放到 Spring 容器中
     * @return
     */
}
```

```
*/  
@Bean  
public RestTemplate restTemplate() {  
    return new RestTemplate();  
}  
}
```

4.3.3.4 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：



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

4.3.3.5 消费者通过 Eureka 访问提供者

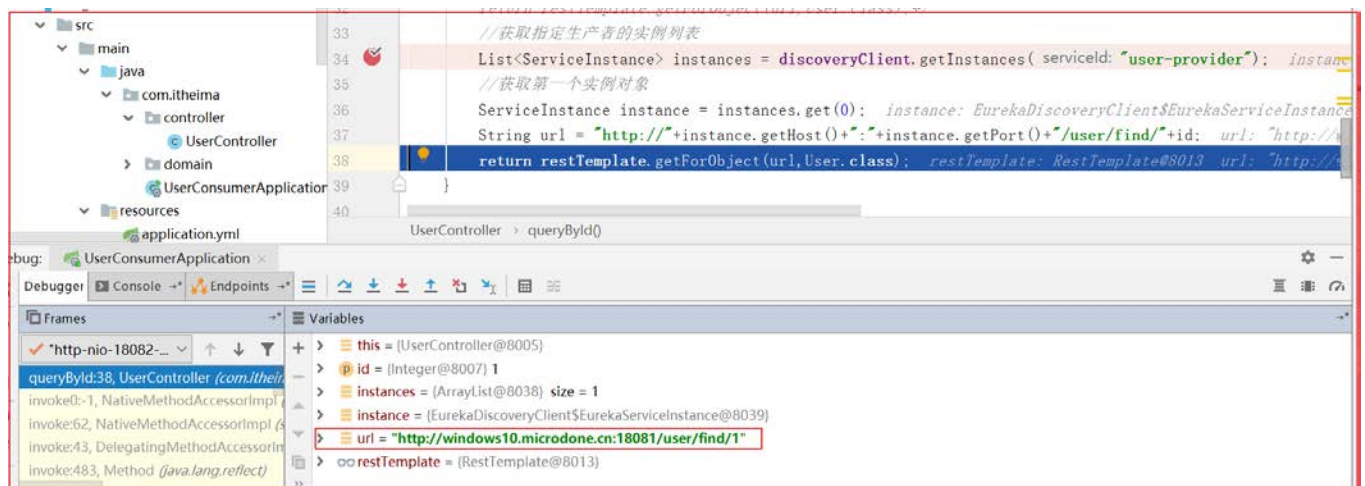
之前消费者 user-consumer 访问服务提供者 user-provider 是通过 <http://localhost:18081/user/find/1> 访问的，这里是具体的路径，没有从 Eureka 获取访问地址，我们可以让消费者从 Eureka 那里获取服务提供者的访问地址，然后访问服务提供者。

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

```
@RestController  
@RequestMapping(value = "consumer")  
public class UserController {  
  
    @Autowired  
    private RestTemplate restTemplate;  
  
    @Autowired  
    private DiscoveryClient discoveryClient; //此对象用于向注册中心获取服务列表  
  
    /**  
     * 在 user-consumer 服务中通过 RestTemplate 调用 user-provider 服务  
     * @param id  
     * @return  
     */  
}
```

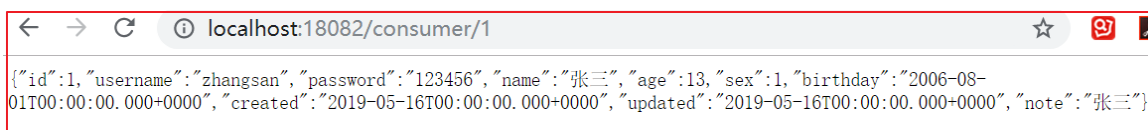
```
*/  
@GetMapping(value =("/{id}")  
public User queryById(@PathVariable(value = "id") Integer id) {  
    /*String url = "http://localhost:18081/user/find/"+id;  
    return restTemplate.getForObject(url, User.class);*/  
    //获取指定生产者的实例列表  
    List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");  
    //获取第一个实例对象  
    ServiceInstance instance = instances.get(0);  
    String url = "http://" + instance.getHost() + ":" + instance.getPort() + "/user/find/" + id;  
    return restTemplate.getForObject(url, 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：



使用 IP 访问配置

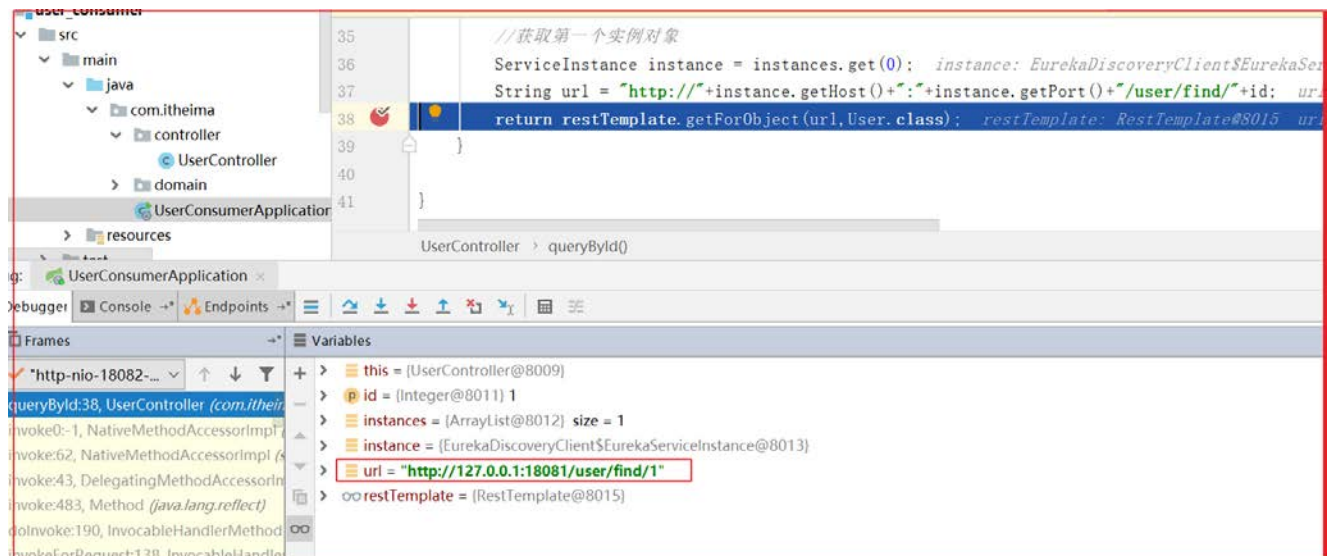
上面的请求地址是服务状态名字，其实也是当前主机的名字，可以通过配置文件，将它换成 IP，修改 application.yml 配置文件，代码如下：

```
server:  
  port: 18081  
spring:  
  datasource:  
    driver-class-name: com.mysql.cj.jdbc.Driver  
    username: root  
    password: root
```

```
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
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 与 user-consumer，并再次测试，测试效果如下：



4.3.4 Eureka 配置详解

4.3.4.1 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

4.3.4.2 Eureka 客户端

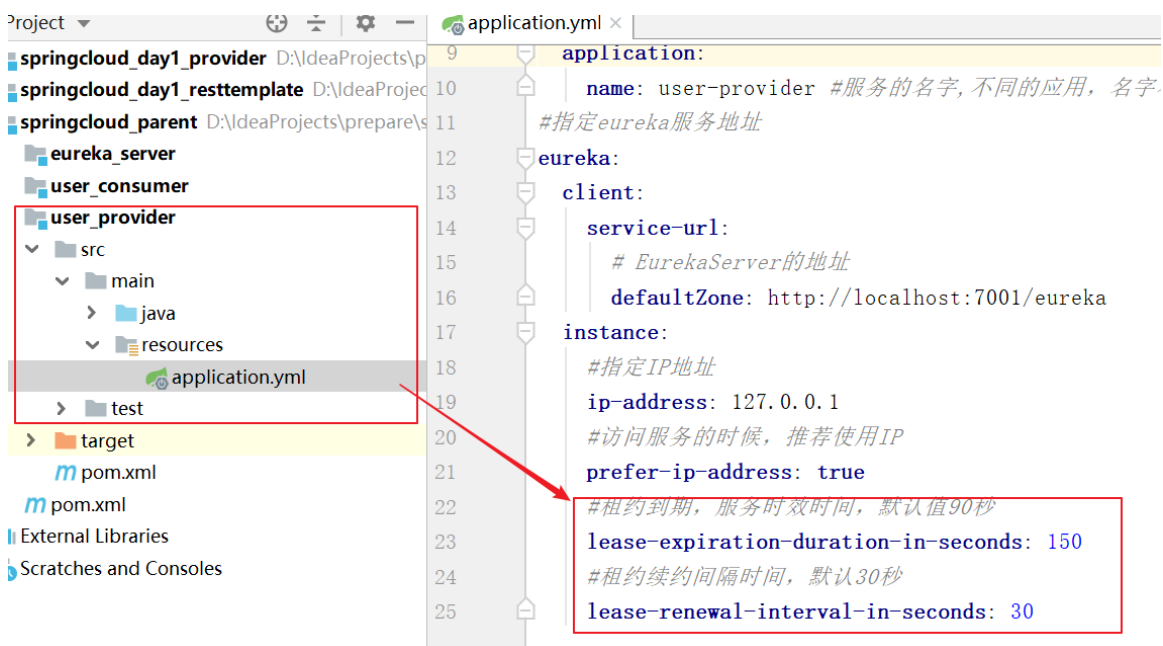
服务提供者要向 EurekaServer 注册服务，并完成服务续约等工作

4.3.4.2.1 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。

4.3.4.2.2 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)。



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:

1. 两个参数可以修改服务续约行为
lease-renewal-interval-in-seconds:90, 租约到期时效时间, 默认 90 秒
lease-expiration-duration-in-seconds:30, 租约续约间隔时间, 默认 30 秒

2.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]

4.3.4.2.3 获取服务列表



上图配置如下:

```
# 每隔 30 获取服务列表(只读备份)
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

4.3.4.3 失效剔除和自我保护

4.3.4.3.1 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

4.3.4.3.2 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

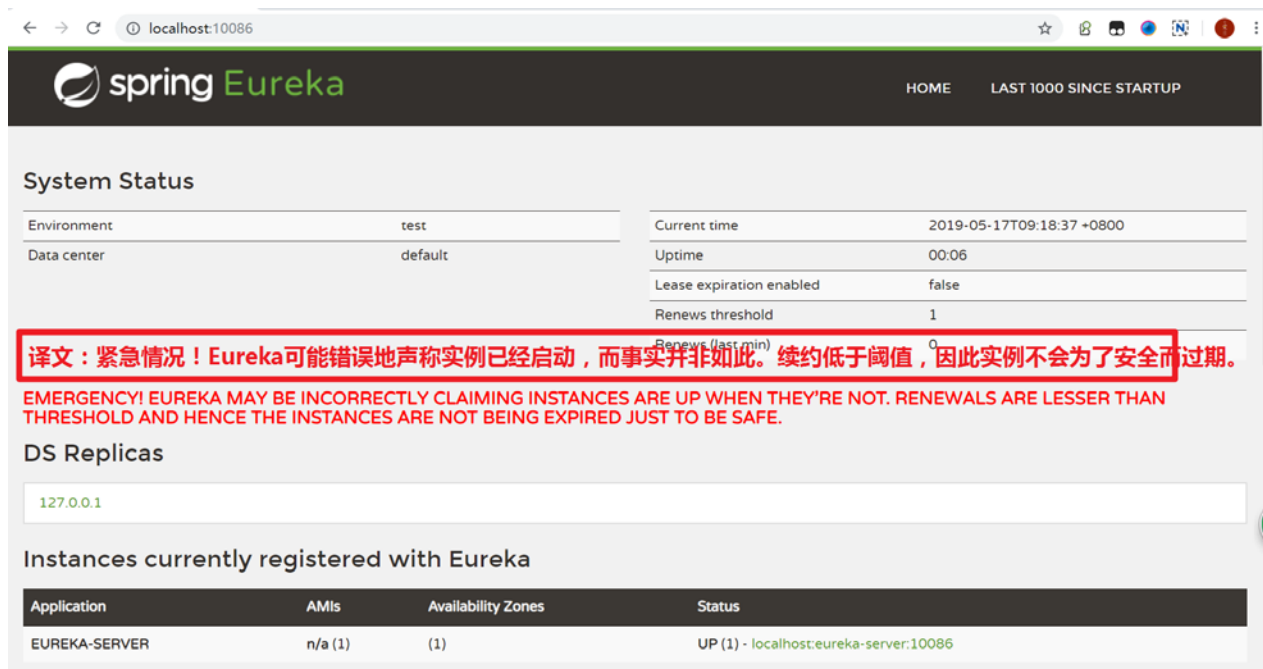


上图代码如下：

```
server:
  # 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，单位是毫秒
  eviction-interval-timer-in-ms: 5000
```

4.3.4.3.3 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。



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



上图配置如下：

```
# 关闭自我保护功能，默认是打开的
enable-self-preservation: false
```

4.4 小结

● 理解 Eureka 的原理图

properties

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

properties

- 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@EnableDiscoveryClient 或者@EnableEurekaClient

● 能实现服务消费者向 Eureka 注册服务

properties

- 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@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 Ribbon 简介

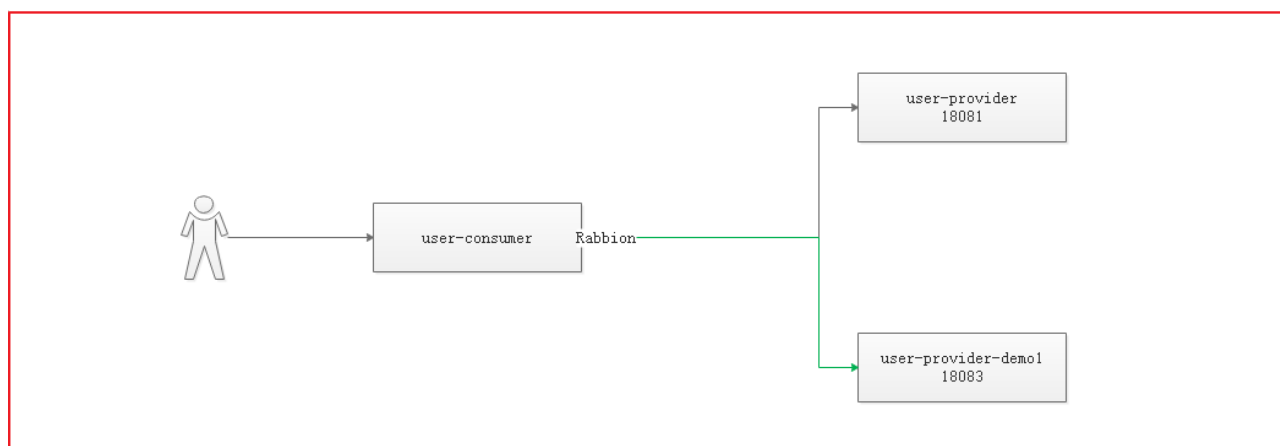
什么是 Ribbon?

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

5.2.1 入门案例

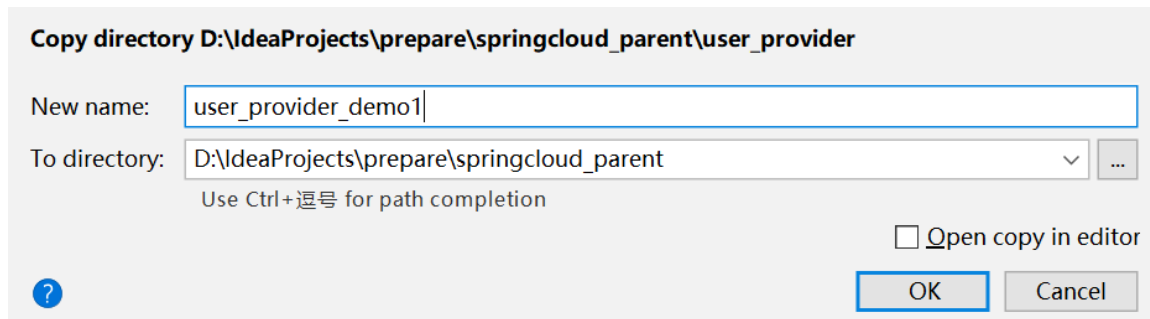
5.2.1.1 多个服务集群



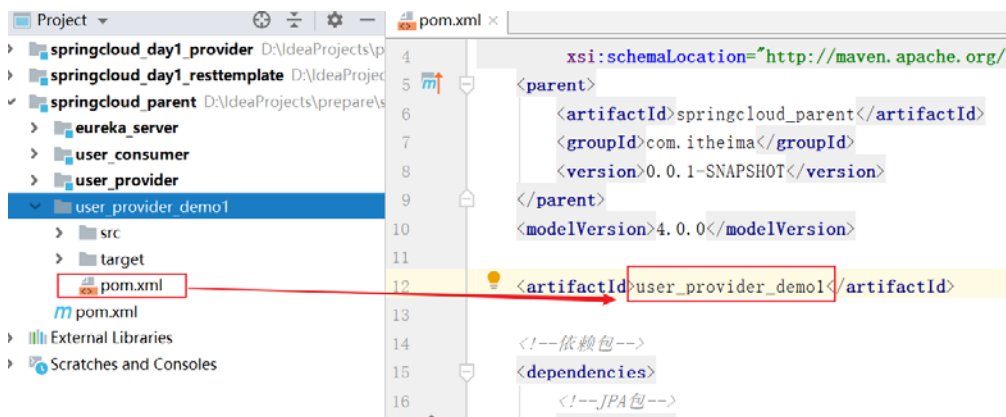
如果想要做负载均衡,我们的服务至少 2 个以上,为了演示负载均衡案例,我们可以复制 2 个工程,分别为 user-provider 和 user-provider-demo1, 可以按照如下步骤拷贝工程:

5.2.1.1.1 选中 user_provider,按 Ctrl+C, 然后 Ctrl+V

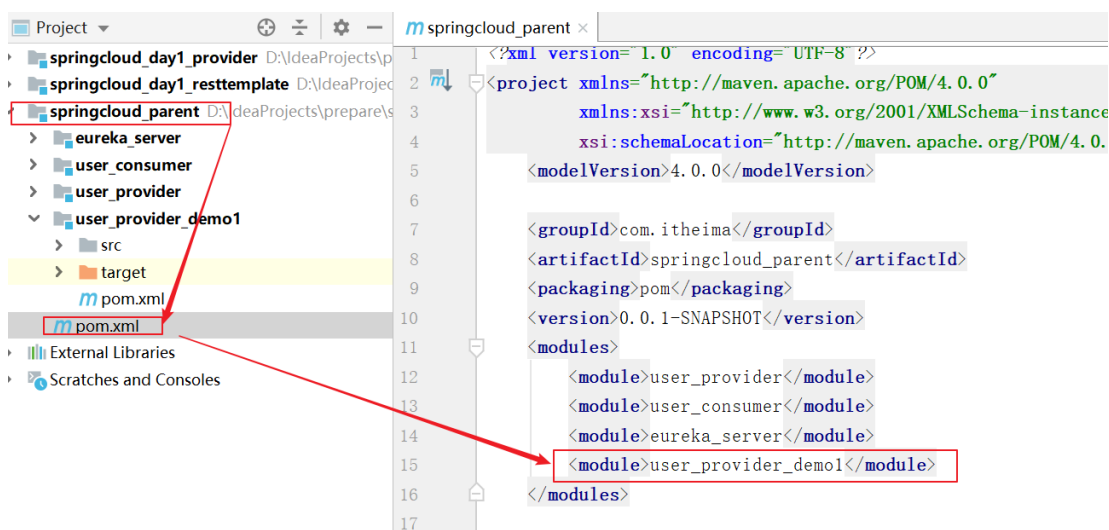
5.2.1.1.2 名字改成 user_provider_demo1,点击 OK



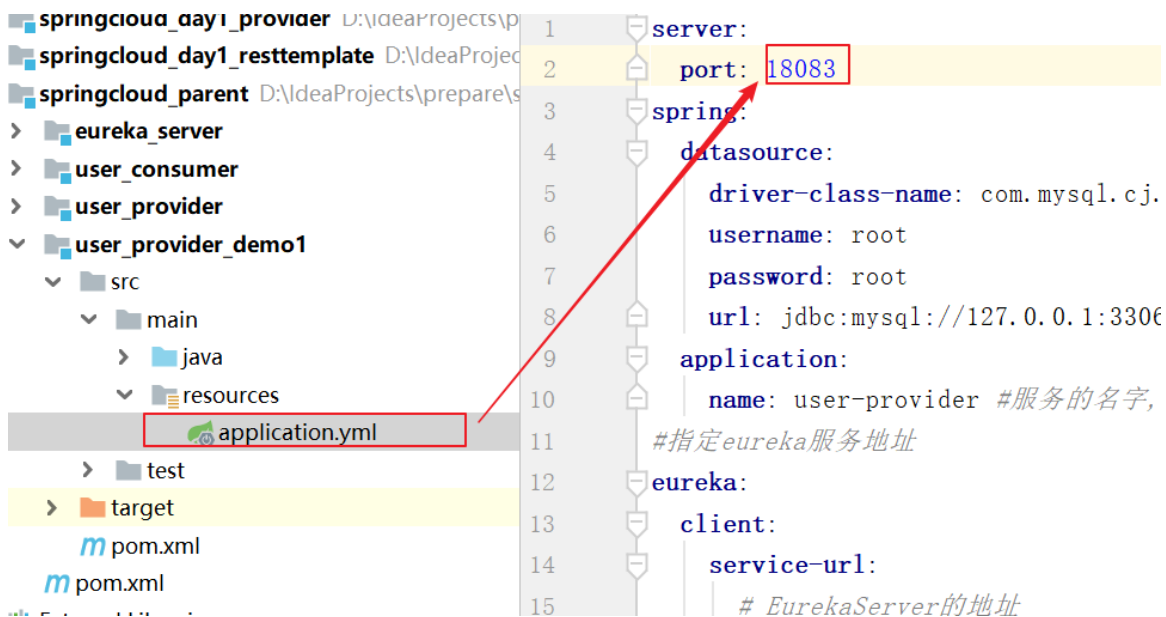
5.2.1.1.3 将 user-provider-demo1 的 artifactId 换成 user-provider-demo1



5.2.1.1.4 在 springcloud-parent 的 pom.xml 中添加一个 <module>user_provider_demo1</module>



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



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.getUsername() + "    user-provider ");
    return user;
}
```

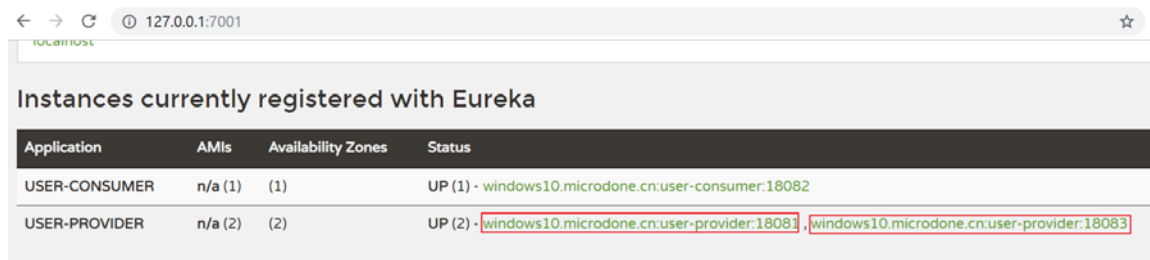
user-provider-demo1:

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

5.2.1.1.6 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。



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

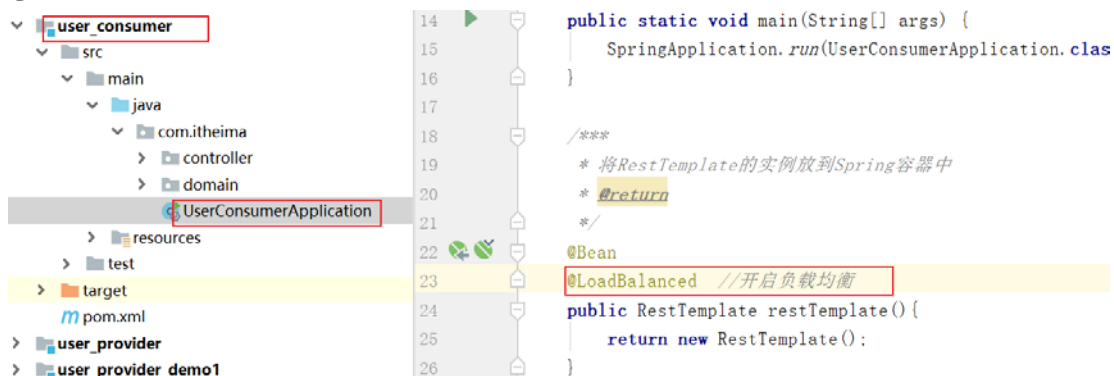


5.2.1.2 开启负载均衡

5.2.1.2.1 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：



5.2.1.2.2 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：

```
@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);*/
    //获取指定生产者的实例列表

    /* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
    //获取第一个实例对象
    ServiceInstance instance = instances.get(0);
    String url = "http://" + instance.getHost() + ":" + instance.getPort() + "/user/find/" + id;*/

    String url = "http://user-provider/user/find/" + id;
    return restTemplate.getForObject(url, User.class);
}
```

5.2.1.2.3 测试

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

```
← → ↺ ⓘ localhost:18082/consumer/1
{"id":1,"username":"zhangsan user-provider","password":"123456","name":"张三","age":13,"sex":1,"birth":16T00:00:00.000+0000,"updated":"2019-05-16T00:00:00.000+0000","note":"张三"}
```

5.2.1.3 其他负载均衡策略配置

需要在消费者中配置修改轮询策略：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

5.2.2 负载均衡源码跟踪探究

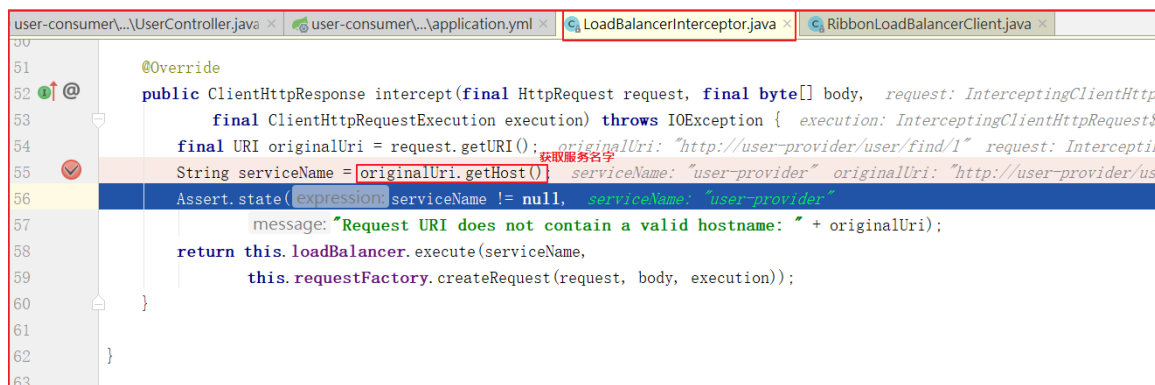
为什么只输入了 Service 名称就可以访问了呢？不应该需要获取 ip 和端口吗？

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。

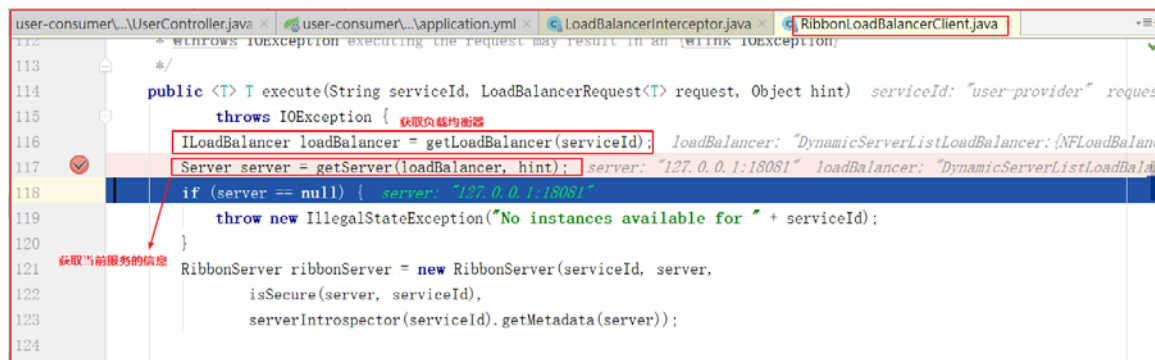
源码跟踪步骤：

打开 LoadBalancerInterceptor 类，断点打入 intercept 方法中



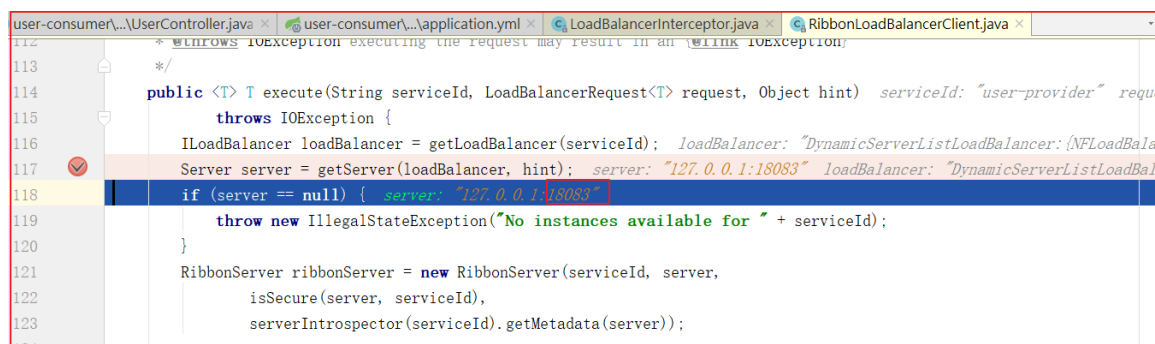
```
51 @Override
52 public ClientHttpResponse intercept(final HttpRequest request, final byte[] body, request: InterceptingClientHttp
53     final ClientHttpRequestExecution execution) throws IOException { execution: InterceptingClientHttpRequest$
54     final URI originalUri = request.getURI(); originalUri: "http://user-provider/user/find/1" request: Intercepti
55     String serviceName = originalUri.getHost(); serviceName: "user-provider" originalUri: "http://user-provider/us
56     Assert.state(expression: serviceName != null, serviceVase: "user-provider"
57         message: "Request URI does not contain a valid hostname: " + originalUri);
58     return this.loadBalancer.execute(serviceName,
59         this.requestFactory.createRequest(request, body, execution));
60 }
61
62 }
63 }
```

继续跟入 execute 方法：发现获取了 18081 发端口的服务



```
112 * throws IOException executing the request may result in an @link IOException
113 */
114 public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) serviceId: "user-provider" reques
115     throws IOException {
116     ILoadBalancer loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer: {NFLoadBala
117     Server server = getServer(loadBalancer, hint); server: "127.0.0.1:18081" loadBalancer: "DynamicServerListLoadBala
118     if (server == null) { server: "127.0.0.1:18081"
119         throw new IllegalStateException("No instances available for " + serviceId);
120     }
121     RibbonServer ribbonServer = new RibbonServer(serviceId, server,
122         isSecure(server, serviceId),
123         serverIntrospector(serviceId).getMetadata(server));
124 }
```

再跟下一次，发现获取的是 18081 和 18083 之间切换



```
112 * throws IOException executing the request may result in an @link IOException
113 */
114 public <T> T execute(String serviceId, LoadBalancerRequest<T> request, Object hint) serviceId: "user-provider" reques
115     throws IOException {
116     ILoadBalancer loadBalancer = getLoadBalancer(serviceId); loadBalancer: "DynamicServerListLoadBalancer: {NFLoadBala
117     Server server = getServer(loadBalancer, hint); server: "127.0.0.1:18083" loadBalancer: "DynamicServerListLoadBala
118     if (server == null) { server: "127.0.0.1:18083"
119         throw new IllegalStateException("No instances available for " + serviceId);
120     }
121     RibbonServer ribbonServer = new RibbonServer(serviceId, server,
122         isSecure(server, serviceId),
123         serverIntrospector(serviceId).getMetadata(server));
124 }
```

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 { 服务名称}.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 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。

6.3 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；

6.4 熔断原理分析



熔断器的原理很简单，如同电力过载保护器。

熔断器状态机有 3 个状态：

- 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

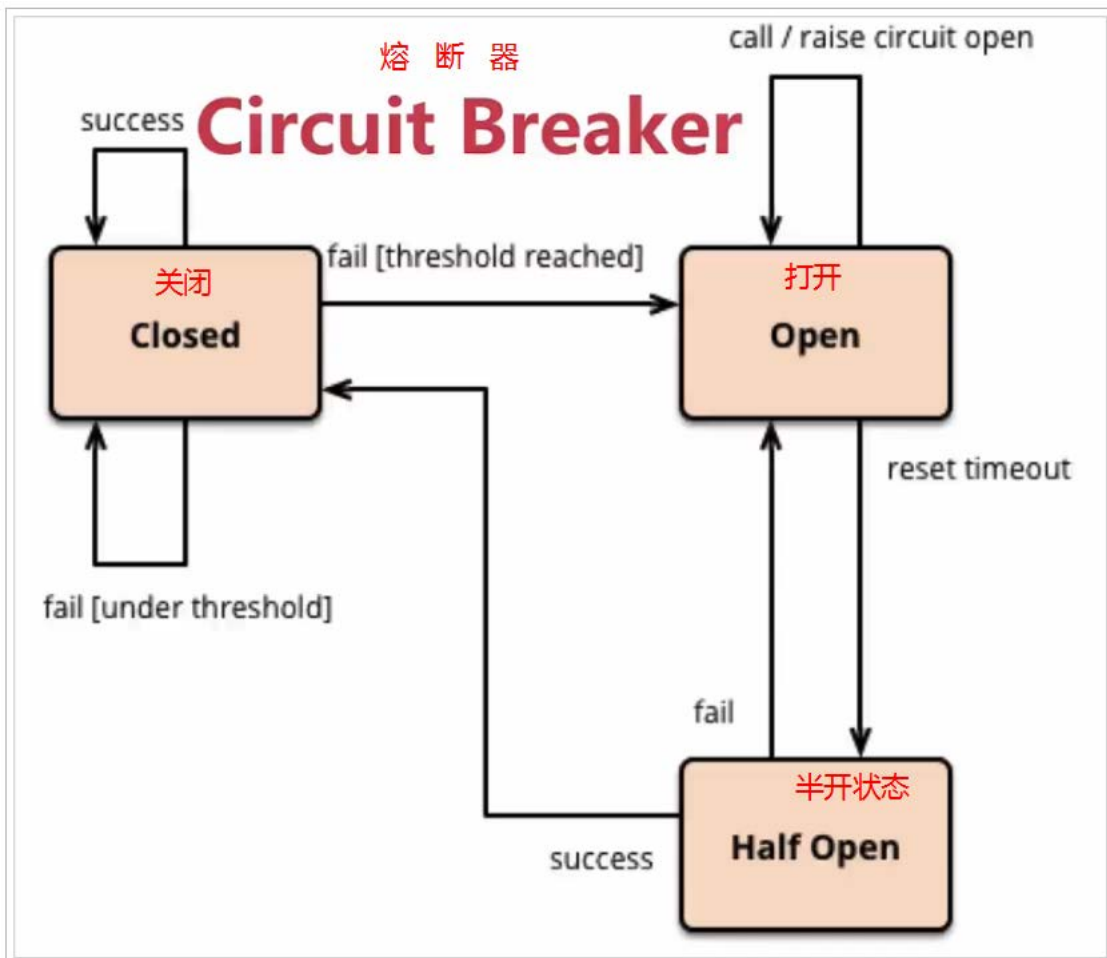
默认失败比例的阈值是 50%，请求次数最低不少于 20 次

- 3.Half Open：半开状态

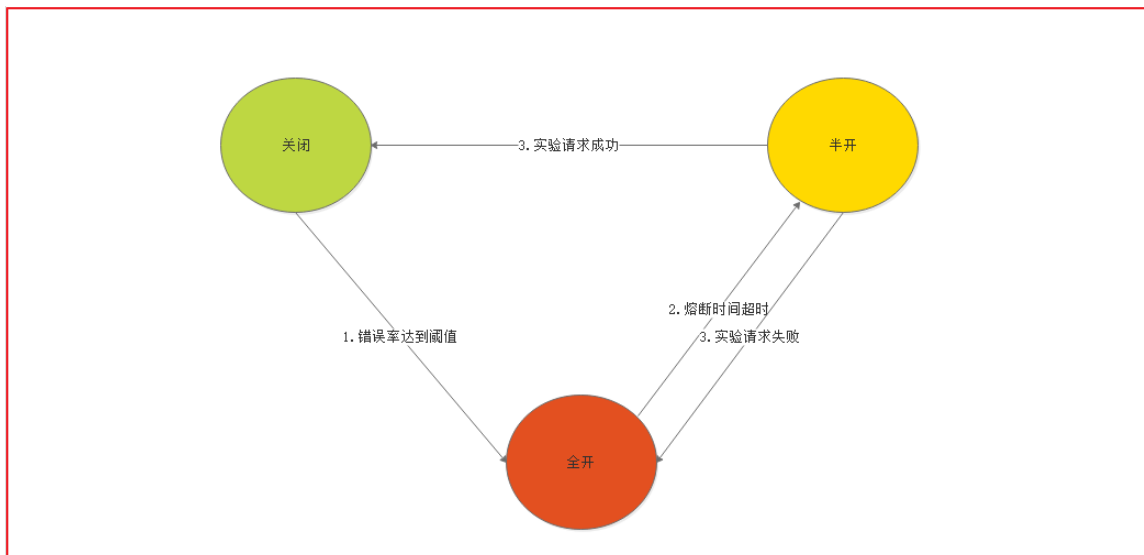
Open 状态不是永久的，打开一会后会进入休眠时间(默认 5 秒)。休眠时间过后会进入半开状态。

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】



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)。

6.5 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

6.5.1 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>
```

6.5.2 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 发现功能
@EnableCircuitBreaker //开启熔断器
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。

6.5.3 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：

```
@GetMapping(value =("/{id}")
@HystrixCommand(fallbackMethod = "failBack") //方法如果处理出问题，就调用降级处理方法
public User queryById(@PathVariable(value = "id") Integer id) {
    /*String url = "http://localhost:18081/user/find/"+id;
    return restTemplate.getForObject(url, User.class);*/
    //获取指定生产者的实例列表

    /* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
    //获取第一个实例对象
    ServiceInstance instance = instances.get(0);
    String url = "http://" + instance.getHost() + ":" + instance.getPort() + "/user/find/" + id;*/

    String url = "http://user-provider/user/find/" + id;
    return restTemplate.getForObject(url, User.class);
}
```

6.5.4 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：

```
← → ↺ ⓘ localhost:18082/consumer/1
{"id":null,"username":"服务降级, 默认处理!", "password":null, "name":null, "age":null, "sex":null, "birthday":n
```

6.6 其他熔断策略配置(了解)

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

6.6.1 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。

6.6.2 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。

6.7 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。

6.7.1 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 fallback 兜底方法

6.7.2 类上默认服务降级的 fallback 兜底方法

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

6.7.3 案例

- 在 user-consumer 的 com.itheima.controller.UserController 类中添加一个全局熔断方法
- 在 queryById 方法上将原来的 @HystrixCommand 相关去掉，并添加 @HystrixCommand 注解
- 在 user-consumer 的 com.itheima.controller.UserController 类上添加 @DefaultProperties(defaultFallback = "defaultFailBack")
- 测试访问 <http://localhost:18082/consumer/1>



Controller 全部代码

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

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private DiscoveryClient discoveryClient; //此对象用于向注册中心获取服务列表

    /**
     * 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;
    }

    /**
     * 在 user-consumer 服务中通过 RestTemplate 调用 user-provider 服务
     * @param id
     * @return
     */
    @GetMapping(value =("/{id}")
    // @HystrixCommand(fallbackMethod = "failBack") //方法如果处理出问题，就调用降级处理方法
    @HystrixCommand
    public User queryById(@PathVariable(value = "id") Integer id) {
        /*String url = "http://localhost:18081/user/find/"+id;
        return restTemplate.getForObject(url, User.class);*/
        //获取指定生产者的实例列表

        /* List<ServiceInstance> instances = discoveryClient.getInstances("user-provider");
        //获取第一个实例对象
        ServiceInstance instance = instances.get(0);
        String url = "http://" + instance.getHost() + ":" + instance.getPort() + "/user/find/" + id;*/

        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;
}
```

6.8 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:

properties

- 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

properties

- 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

- 能理解什么是线程隔离，什么是服务降级

properties

- 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

properties

- 1.定义一个局部处理熔断的方法 failBack()
- 2.在指定方法上使用 @HystrixCommand(fallbackMethod = "failBack")配置调用

- 能实现全局方法熔断案例

properties

1. 定义一个全局处理熔断的方法 `defaultFailBack()`
2. 在类上使用 `@DefaultProperties(defaultFallback = "defaultFailBack")` 配置调用
3. 在指定方法上使用 `@HystrixCommand`