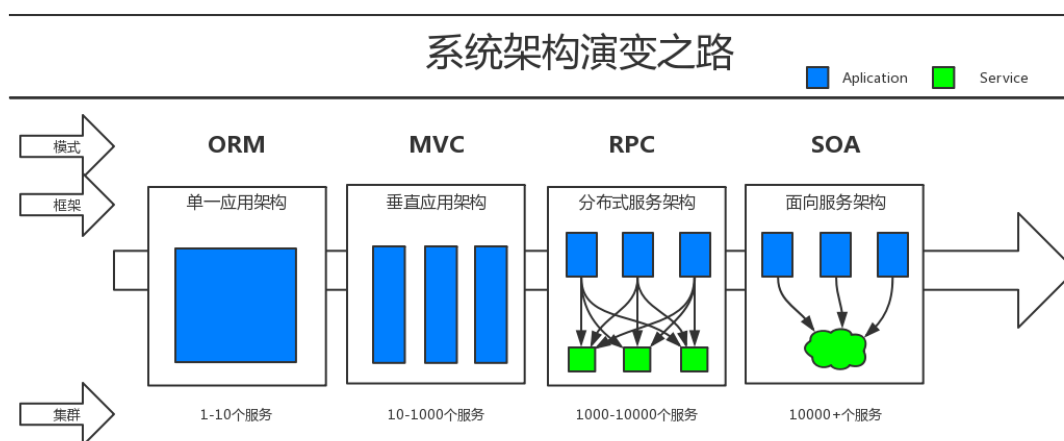


# 学习目标

1. 理解什么是SpringCloud，及其出现的意义
2. 知道微服务业务场景中，必然面临的一些问题
3. 理解什么是注册中心Eureka，及其解决的问题
  - 能够搭建注册中心为服务
4. 理解什么是负载均衡Ribbon，及其解决的问题
  - 能够搭建集群，测试负载均衡效果
5. 理解什么是熔断器Hystrix，及其解决的问题
6. 能够编写出熔断器的服务降级方法

## 一、系统架构演变之路(回顾)



### 1.1 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。

### 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。

### 1.3 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。

### 1.4 面向服务架构

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。

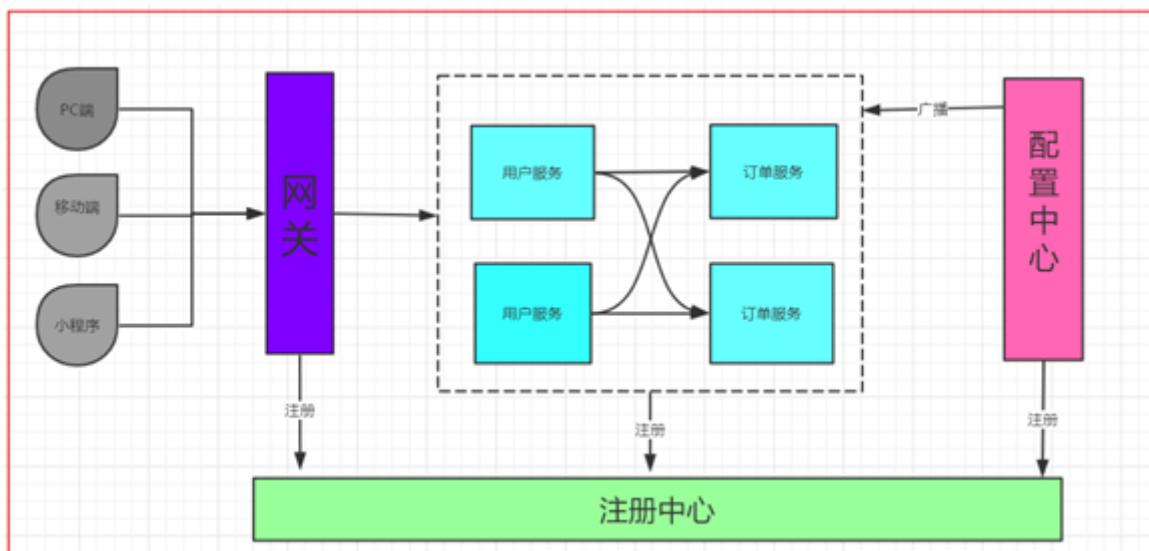
## 二、初识Spring Cloud

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；

### 2.1 Spring Cloud简介

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

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



### 2.2 Spring Cloud的版本

## Documentation

Each **Spring project** has its own; it explains in great details how you can use **project features** and what you can achieve with them.

Hoxton <b>SNAPSHOT</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Greenwich SR1 <b>GA</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Greenwich <b>SNAPSHOT</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Finchley SR3 <b>GA</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Finchley <b>SNAPSHOT</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Edgware SR5 <b>GA</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Edgware <b>SNAPSHOT</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>
Dalston SR5 <b>GA</b>	<a href="#">Reference Doc.</a>	<a href="#">API Doc.</a>

- 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

## 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

## 三、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。

**目标：**模拟一个最简单的服务调用场景，场景中包含微服务提供者(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 创建服务的父工程

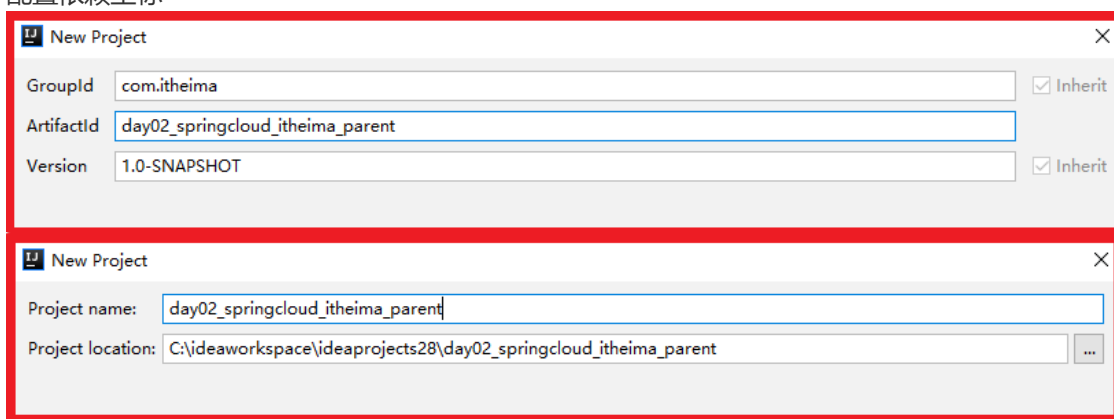
**目标：**新建一个父项目itheima\_parent

**实现步骤：**

1. 创建maven的工程
2. 配置依赖坐标，maven的parent、以及SpringCloud的依赖管理坐标

**实现过程：**

1. 创建maven工程，itheima\_parent
2. 配置依赖坐标



3. 添加起步依赖坐标：SpringBoot、SpringCloud

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

<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>
```

## 3.2 创建服务提供者(provider)工程

**目标：**新建一个项目provider\_service，对外提供查询用户的服务

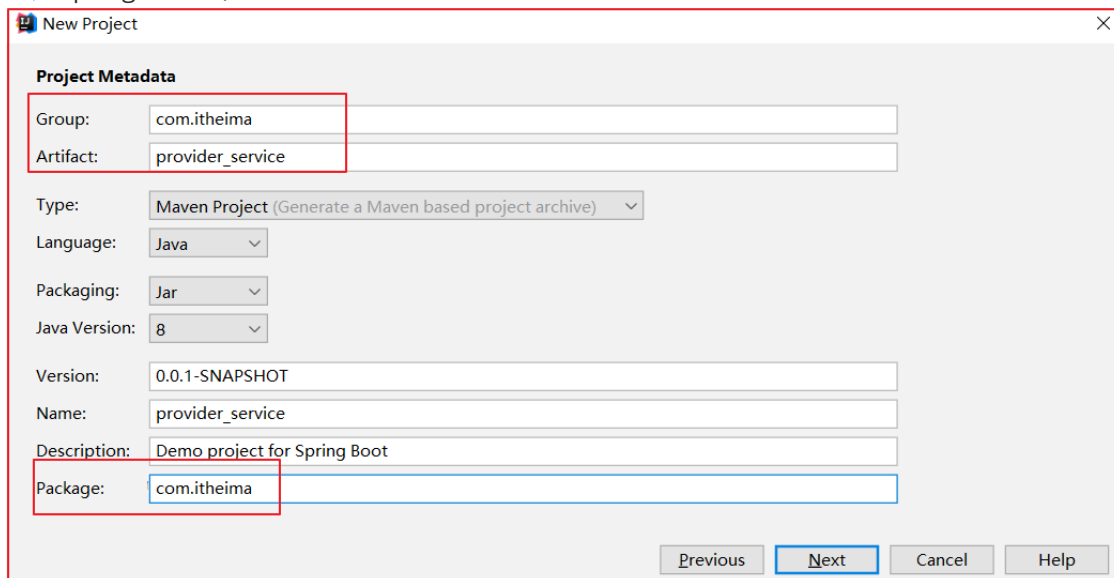
**实现步骤：**

1. 创建SpringBoot工程

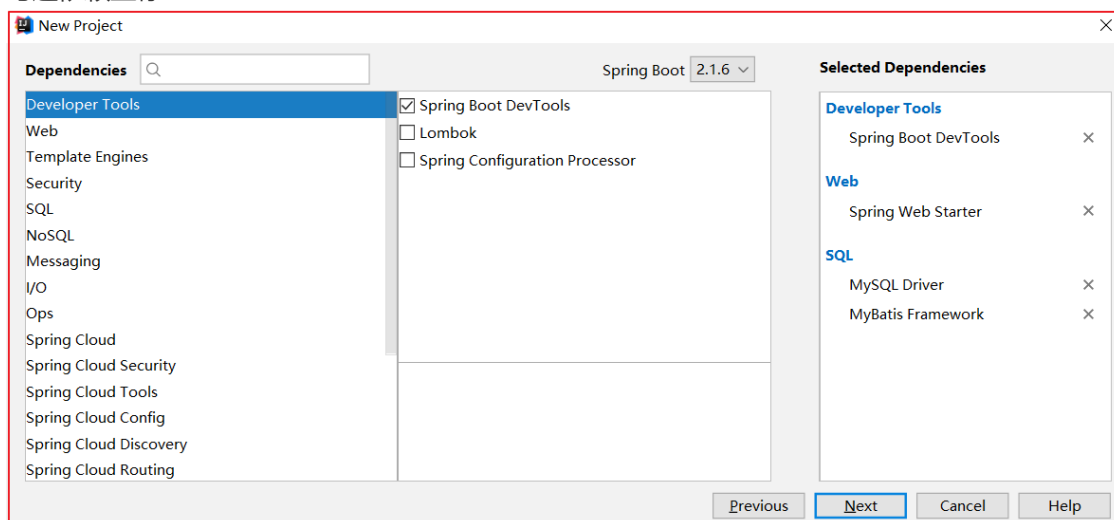
2. 勾选依赖坐标
3. 数据库连接信息
4. 创建User表、创建实体User
5. 编写三层架构: Mapper、Service、controller, 编写查询方法
6. 配置Mapper映射文件
7. 在application.properties中添加MyBatis配置, 扫描mapper.xml和mapper
8. 访问测试地址

## 实现过程:

1. 创建SpringBoot工程



2. 勾选依赖坐标



3. 数据库连接信息

4. 创建User表、创建实体User

```
-- 创建数据库
CREATE database springcloud CHARACTER SET utf8 COLLATE utf8_general_ci;
-- 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');

```

实体bean:

```

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
    //getter setter
}

```

5. 编写三层架构: Mapper、Service、controller, 编写查询所有的方法

```

@Repository
Mapper
public interface UserMapper {
    User findById();
}

```

service类:

```

@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private UserMapper userMapper;

    @Override
    public List<User> findAll() {
        return userMapper.findAll();
    }
}

```

```

@Override
public User findById(Integer id) {
    return userMapper.findById(id);
}

}

```

controller:

```

@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    UserService userService;
    /**
     * 查询所有
     * @return
     */
    @RequestMapping("/findAll")
    public List<User> findAll() {
        return userService.findAll();
    }
    /**
     * 根据id查询
     * @param id
     * @return
     */
    @RequestMapping("/findById")
    public User findById(Integer id) {
        return userService.findById(id);
    }
}

```

## 6. 配置Mapper映射文件

```

<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.iheima.mapper.UserMapper">

    <select id="findAll" resultType="user">
        select * from tb_user
    </select>
    <select id="findById" parameterType="Integer" resultType="user">
        select * from tb_user where id = #{id}
    </select>
</mapper>

```

## 7. 在application.properties中添加MyBatis配置，扫描mapper.xml和mapper

```
# 端口
server.port: 9091
# DB 配置
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.datasource.url: jdbc:mysql://127.0.0.1:3306/springcloud?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
spring.datasource.password: root
spring.datasource.username: root
# 扫描实体
mybatis.type-aliases-package: com.itheima.domain
# mapper.xml配置文件路径
mybatis.mapper-locations: classpath:mapper/*Mapper.xml
```

#### 8. 访问测试地址

<http://localhost:9091/user/findById?id=1>

## 3.3 创建服务消费者(consumer)工程

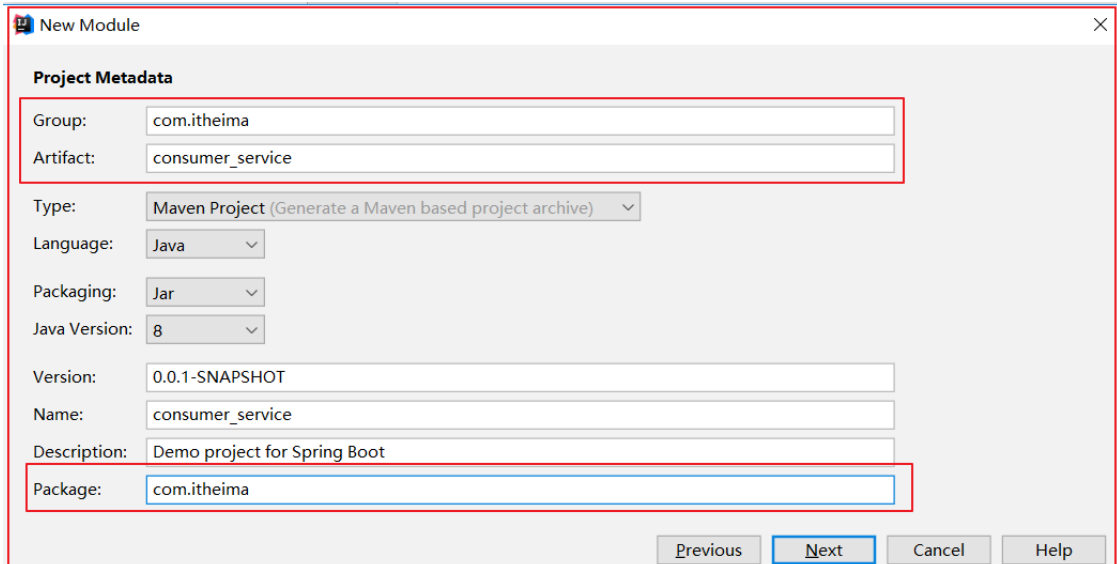
**目标：**新建一个项目consumer\_service，调用用户微服务提供的查询用户服务

**实现步骤：**

1. 创建消费者SpringBoot工程consumer\_service
2. 勾选starter：开发者工具devtools、web
3. 注册http请求客户端对象RestTemplate
4. 编写Controller，用RestTemplate访问服务提供者
5. 启动服务并测试

**实现过程：**

1. 创建consumer\_service的SpringBoot工程



New Module

**Project Metadata**

Group: com.itheima

Artifact: consumer\_service

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

Language: Java

Packaging: Jar

Java Version: 8

Version: 0.0.1-SNAPSHOT

Name: consumer\_service

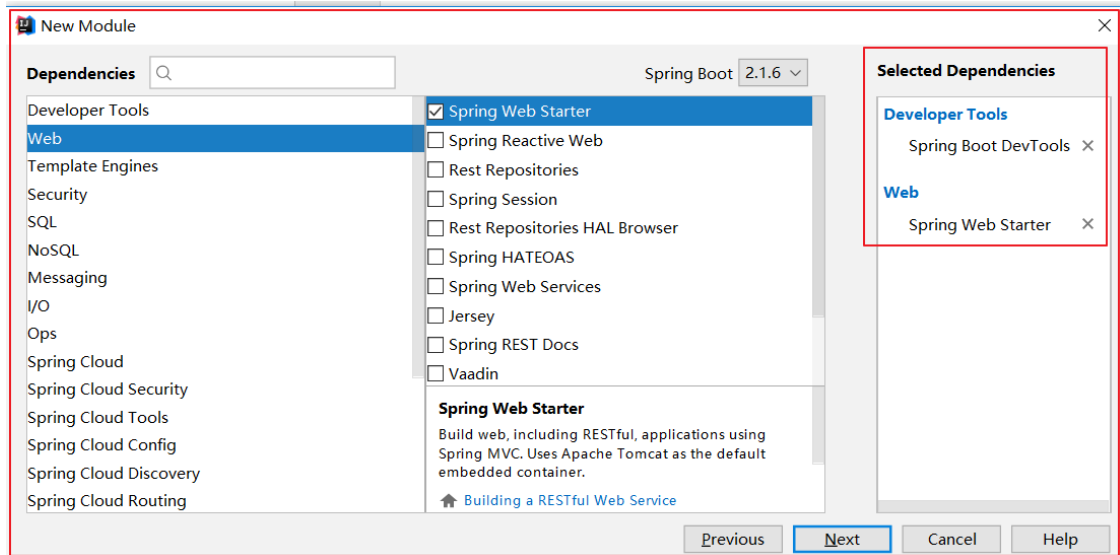
Description: Demo project for Spring Boot

Package: com.itheima

Previous Next Cancel Help



## 2. 勾选需要的相关依赖



## 3. 编写代码

### 1. 在启动类中注册RestTemplate

```
@SpringBootApplication
public class ConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
    //注册RestTemplate
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

### 2. 编写ConsumerController, 在Controller中直接调用RestTemplate, 远程访问User-service

```
@RestController

public class ConsumerController {
    @Autowired
    private RestTemplate restTemplate;
    @RequestMapping("/consumer/{id}")
    public User queryById(@PathVariable Long id){
        String url = String.format("http://localhost:9091/user/%d",
id);
        return restTemplate.getForObject(url, User.class);
    }
}
```

### 4. 启动并测试, 访问: <http://localhost:8080/consumer/1>



## 3.4 思考问题

provider\_service：对外提供用户查询接口

consumer\_service：通过RestTemplate访问接口查询用户数据

存在的问题：

1. 在消费者服务中，访问url地址硬编码了，url地址端口变化了怎么办？服务死掉了如何才能知道？
  - 在消费者服务中，是不清楚服务提供者状态的！
2. 为了增加服务并发访问量，我们搭建集群，集群的负载均衡怎么实现？
3. 服务提供者如果出现故障，会不会向用户抛出异常页面，该不该抛出错误页面？
4. RestTemplate这种请求调用方式是否还有优化空间？复用，管理，可读性角度来思考
5. 多服务权限拦截如何实现？怎么保证所有微服务服务的安全性？
6. 众多微服务的配置文件，每次都修改很多个，是不是很麻烦！？

其实上面说的部分问题，概括一下就是微服务架构必然面临的一些问题。

- 服务管理：自动注册与发现、状态监管
- 服务负载均衡
- 熔断
- 面向接口的远程调用
- 网关拦截、路由转发
- 统一配置修改

## 四、注册中心 Spring Cloud Eureka

在消费者服务中，访问url地址硬编码了，url地址端口变化了怎么办？服务死掉了才能知道？

- 在消费者服务中，是不清楚服务提供者状态的！

### 4.1 Eureka 简介

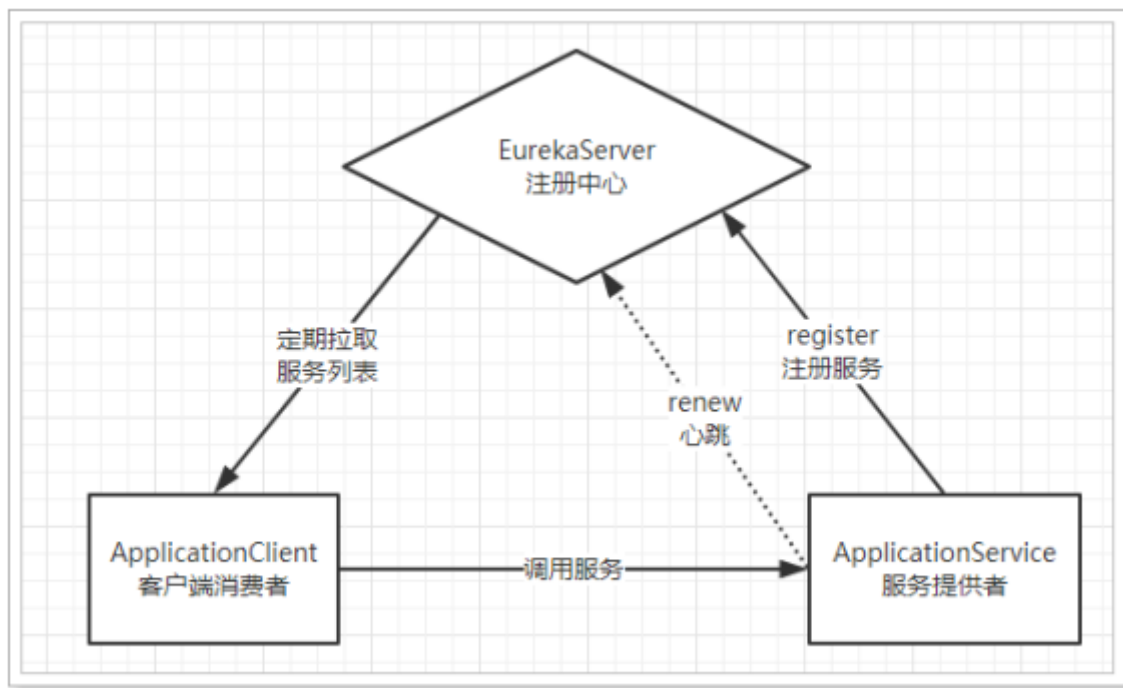
**Spring Cloud Eureka使用Netflix Eureka来实现服务注册与发现（服务治理）。**它既包含了服务端组件，也包含了客户端组件，并且服务端与客户端均采用java编写，所以Eureka主要适用于通过java实现的分布式系统，或是JVM兼容语言构建的系统。

**Eureka服务端组件：**即服务注册中心。它同其他服务注册中心一样，支持高可用配置。依托于强一致性提供良好的服务实例可用性，可以应对多种不同的故障场景。

**Eureka客户端组件：**主要处理服务的注册和发现。客户端服务通过注册和参数配置的方式，嵌入在客户端应用程序的代码中。在应用程序启动时，Eureka客户端向服务注册中心注册自身提供的服务，并周期性的发送心跳来更新它的服务租约。同时，他也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期性的刷新服务状态。

### 4.2 架构图

基本架构图



Eureka服务端组件：就是**服务注册中心**

## 4.3 整合注册中心Eureka

步骤：分三步

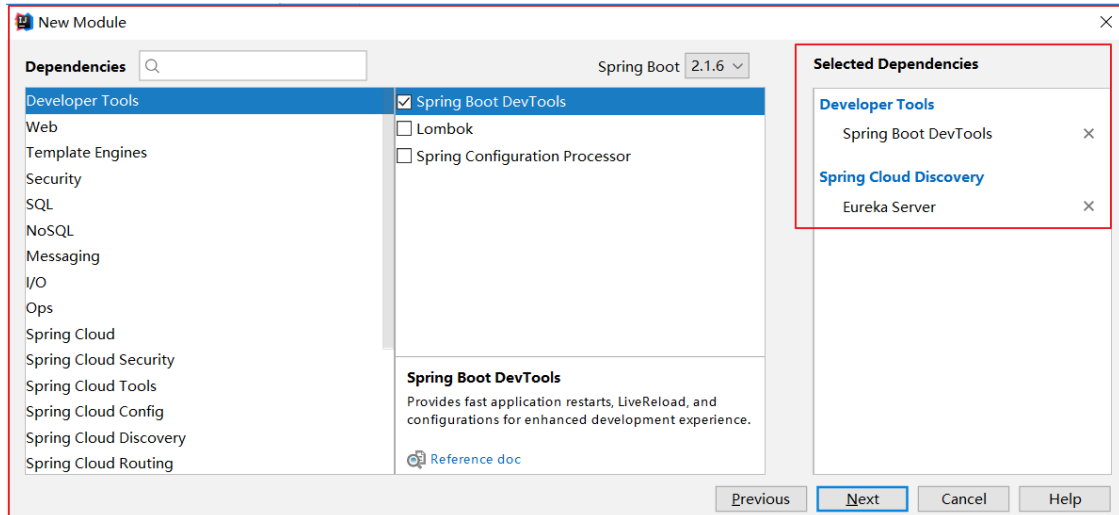
- 第一步：eureka-serve搭建工程eureka\_server
- 第二步：服务提供者-注册服务， user\_service工程
- 第三步：服务消费者-发现服务， consumer\_service工程

### 4.3.1 搭建eureka-server工程

1. 创建eureka\_server的springboot工程。

The screenshot shows the 'New Module' dialog box in an IDE. The 'Project Metadata' section is highlighted with a red box. The fields are as follows: Group: com.itheima, Artifact: eureka\_server, Type: Maven Project (Generate a Maven based project archive), Language: Java, Packaging: Jar, Java Version: 8, Version: 0.0.1-SNAPSHOT, Name: eureka\_server, Description: Demo project for Spring Boot, and Package: com.itheima. The 'Next' button is highlighted with a blue box.

## 2. 勾选坐标



3. 在启动类EurekaServerApplication声明当前应用为Eureka服务使用 @EnableEurekaServer 注解

4. 编写配置文件application.yml

```
# 端口
server.port: 10086
# 应用名称，会在Eureka中作为服务的id标识（serviceId）
spring.application.name: eureka-server
# EurekaServer的地址，现在是自己的地址，如果是集群，需要写其它Server的地址。
eureka.client.service-url.defaultZone: http://127.0.0.1:10086/eureka
```

5. 启动EurekaServerApplication

6. 测试访问地址<http://127.0.0.1:10086>，如下信息代表访问成功

7.

### System Status 系统状态

Environment <small>环境</small>	test <small>测试</small>	Current time <small>当前系统时间</small>	2019-06-28T00:27:33 +0800
Data center <small>数据中心</small>	default <small>默认</small>	Uptime <small>上线时间</small>	00:01
		Lease expiration enabled <small>租约到期启用</small>	false
		Renews threshold <small>期望每分钟收到续约次数</small>	1
		Renews (last min) <small>上一分钟收到的续约次数</small>	0

### DS Replicas

127.0.0.1

Instances currently registered with Eureka

Application <small>应用</small>	AMIs	Availability Zones <small>可用区</small>	Status <small>状态</small>
EUREKA-SERVER	n/a (1)	(1)	UP (1) - localhost:eureka-server:10086

8.

9.

General Info 普通信息		
Name		Value
total-avail-memory	总有效内存	361mb
environment	环境	test
num-of-cpus	CPU数	4
current-memory-usage	当前使用内存	149mb (41%)
server-uptime	服务上线时间	00:01
registered-replicas	注册的副本	http://127.0.0.1:10086/eureka/
unavailable-replicas	无效副本	http://127.0.0.1:10086/eureka/,
available-replicas	有效副本	
Instance Info 当前Eureka 实例信息		
Name		Value
ipAddr	IP地址	192.168.229.1
status	状态(上线)	UP

## 10. 关闭注册自己

```
# 是否抓取注册列表
eureka.client.fetch-registry: false
# 是否注册服务中心Eureka
eureka.client.register-with-eureka: false
```

## 4.3.2 服务提供者-注册服务中心

### 1. 在服务提供者user\_service工程中添加Eureka客户端依赖

```
o <!--eureka客户端starter-->
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-
client</artifactId>
  </dependency>
</dependencies>
<!--SpringCloud所有依赖管理的坐标-->
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>Greenwich.SR1</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

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

- ```

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

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

```

3. 修改配置文件：spring.application.name指定应用名称，作为服务ID使用

- ```

# 端口
server.port: 9091
# DB 配置
spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
spring.datasource.url: jdbc:mysql://127.0.0.1:3306/springcloud?
useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
spring.datasource.password: root
spring.datasource.username: root

# 扫描实体
mybatis.type-aliases-package: com.itheima.domain
# mapper.xml配置文件路径
mybatis.mapper-locations: classpath:mapper/*Mapper.xml

# 应用名称
spring.application.name: user-service
# 注册中心地址
eureka.client.service-url.defaultZone: http://127.0.0.1:10086/eureka

```

4. 完成之后重启项目

5. 客户端代码会自动把服务注册到EurekaServer中

6. 在Eureka监控页面可以看到服务注册成功信息

DS Replicas

127.0.0.1

- Instances currently registered with Eureka

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

### 4.3.3 服务消费者-注册服务中心

1. 在服务消费者spring\_cloud\_itcast\_consumer\_service工程中添加Eureka客户端依赖

- ```

<!-- Eureka客户端 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<!--SpringCloud所有依赖管理的坐标-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>

```

```

        <artifactId>spring-cloud-dependencies</artifactId>
        <version>Greenwich.SR1</version>
        <type>pom</type>
        <scope>import</scope>
    </dependency>
</dependencies>
</dependencyManagement>

```

## 2. 在启动类开启Eureka客户端 @EnabledDiscoveryClient

```

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

```

## 3. 修改配置文件：加入EurekaServer地址

```

o   # 端口
    server.port: 8080
    # 应用名称
    spring.application.name: consumer-demo
    # 注册中心地址
    eureka.client.service-url.defaultZone: http://127.0.0.1:10086/eureka

```

## 4. 启动服务，在服务中心查看是否注册成功

视频结束

## 4.3.3 消费者通过Eureka访问提供者

修改代码

```

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

    @GetMapping("/{id}")
    public User queryById(@PathVariable Long id){
        String url = String.format("http://localhost:9091/user/%d", id);

        //1、获取Eureka中注册的user-service实例列表
        List<ServiceInstance> serviceInstanceList =
discoveryClient.getInstances("user-service");
        //2、获取实例
        ServiceInstance serviceInstance = serviceInstanceList.get(0);
        //3、根据实例的信息拼接的请求地址
    }
}

```

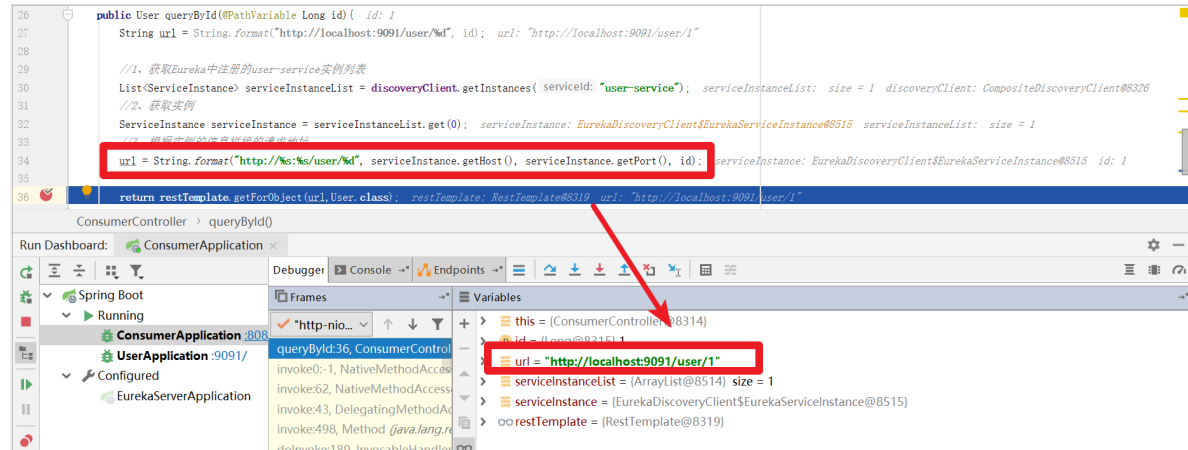
```

        url = String.format("http://%s:%s/user/%d", serviceInstance.getHost(),
        serviceInstance.getPort(), id);
        //发生请求
        return restTemplate.getForObject(url, User.class);
    }
}

```

## Debug跟踪运行

### 服务提供者地址拼接成功



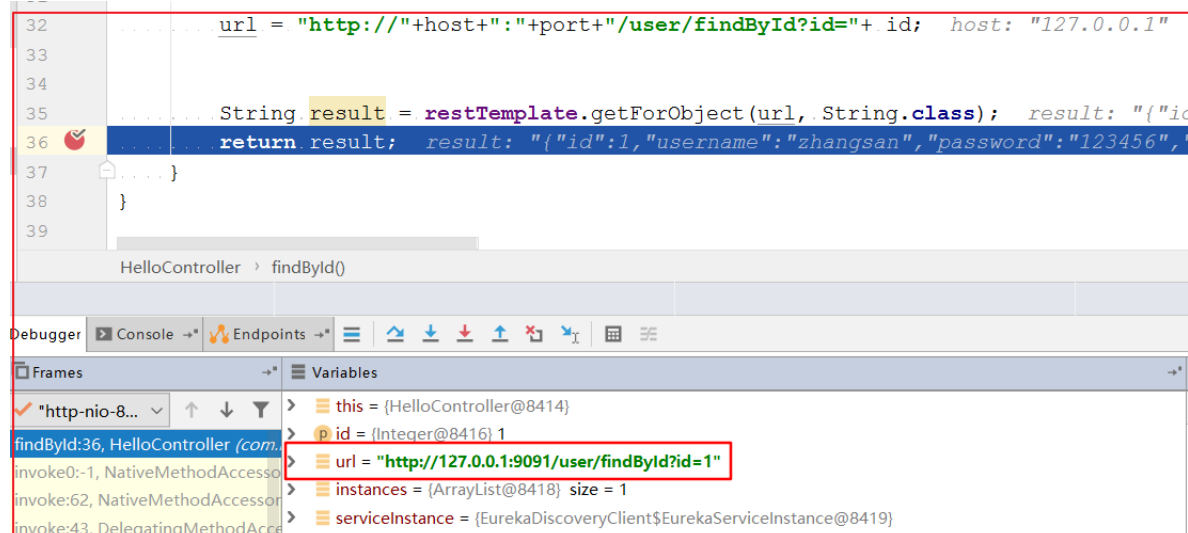
这里服务的host地址有什么问题! ?

```

# 默认注册时使用的是主机名，想用ip进行注册添加如下配置
# ip地址
eureka.instance.ip-address: 127.0.0.1
# 更倾向于使用ip，而不是host名
eureka.instance.prefer-ip-address: true

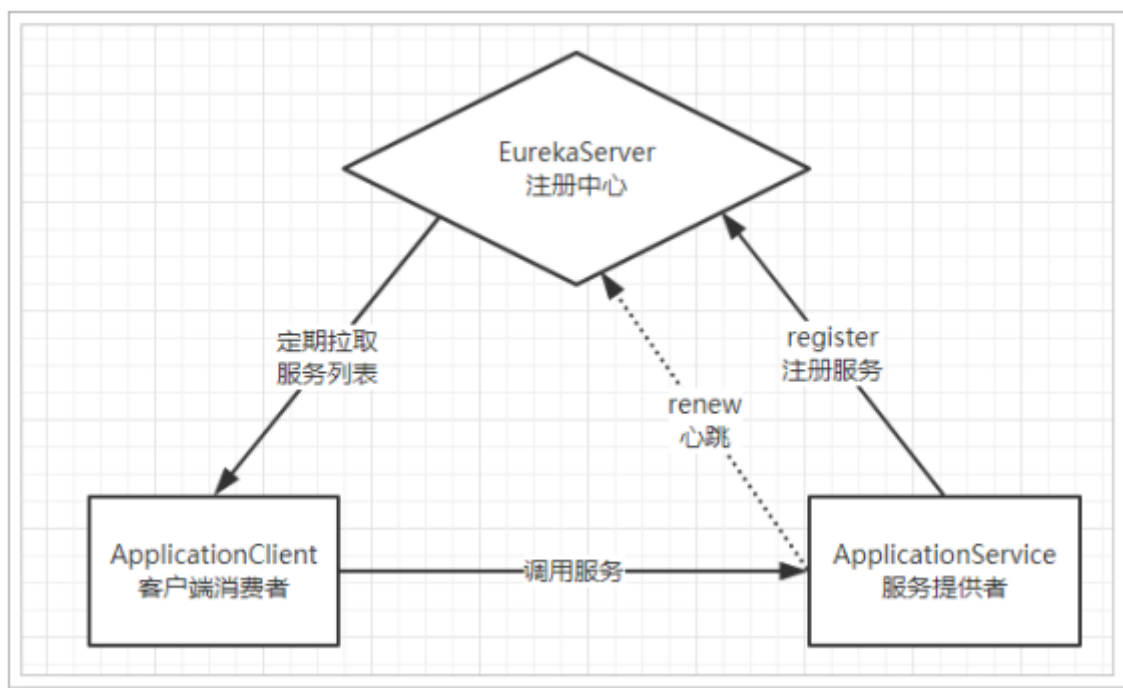
```

## 修改配置之后



## 4.4 Eureka详解





### 4.4.1 基础架构

Eureka架构中的三个核心角色

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

### 4.4.2 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 (register) :

Eureka Client会通过发送REST请求的方式向Eureka Server注册自己的服务，提供自身的元数据，比如ip地址、端口、运行状况指标的url、主页地址等信息。Eureka Server接收到注册请求后，就会把这些元数据信息存储在一个双层的Map中。

#### 服务续约 (renew) :

在服务注册后，Eureka Client会维护一个心跳来持续通知Eureka Server，说明服务一直处于可用状态，防止被剔除。默认每隔30秒 `eureka.instance.lease-renewal-interval-in-seconds` 发送一次心跳来进行服务续约。

```
# 租约续约间隔时间，默认30秒
eureka.instance.lease-renewal-interval-in-seconds: 30
```

#### 获取服务列表 (get registry) :

服务消费者 (Eureka Client) 在启动的时候，会发送一个REST请求给Eureka Server，获取上面注册的服务清单，并且缓存在Eureka Client本地，默认缓存30秒(`eureka.client.registry-fetch-interval-seconds`)。同时，为了性能虑，Eureka Server也会维护一份只读的服务清单缓存，该缓存每隔30秒更新一次。

```
# 每隔多久获取服务中心列表，（只读备份）
eureka.client.registry-fetch-interval-seconds: 30
```

## 服务调用：

服务消费者在获取到服务清单后，就可以根据清单中的服务列表信息，查找到其他服务的地址，从而进行远程调用。

## 服务下线 (cancel)：

当Eureka Client需要关闭或重启时，就不希望在这个时间段内再有请求进来，所以，就需要提前先发送REST请求给Eureka Server，告诉Eureka Server自己要下线了，Eureka Server在收到请求后，就会把该服务状态置为下线（DOWN），并把该下线事件传播出去。

## 失效剔除 (evict)：

服务实例可能会因为网络故障等原因，导致不能提供服务，而此时该实例也没有发送请求给Eureka Server来进行服务下线。所以，还需要有服务剔除的机制。Eureka Server在启动的时候会创建一个定时任务，每隔一段时间（默认60秒），从当前服务清单中把超时没有续约（默认90秒

eureka.instance.lease-expiration-duration-in-seconds）的服务剔除。

```
# 租约到期，服务时效时间，默认值90秒
eureka.instance.lease-expiration-duration-in-seconds: 90
```

## 自我保护：

既然Eureka Server会定时剔除超时没有续约的服务，那就有可能出现一种场景，网络一段时间内发生了异常，所有的服务都没能够进行续约，Eureka Server就把所有的服务都剔除了，这样显然不太合理。所以，就有了自我保护机制，当短时间内，统计续约失败的比例，如果达到一定阈值，则会触发自我保护的机制，在该机制下，Eureka Server不会剔除任何的微服务，等到正常后，再退出自我保护机制。自我保护开关(eureka.server.enable-self-preservation: false)

```
#向Eureka服务中心集群注册服务
eureka.server.enable-self-preservation: false # 关闭自我保护模式（默认值是打开）
```

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

localhost:10086

spring Eureka

HOME LAST 1000 SINCE STARTUP

### System Status

|             |         |                          |                           |
|-------------|---------|--------------------------|---------------------------|
| 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                         |

**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.

### DS Replicas

127.0.0.1

### Instances currently registered with Eureka

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

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

- 自我保护模式下，不会剔除任何服务实例
- 自我保护模式保证了大多数服务依然可用

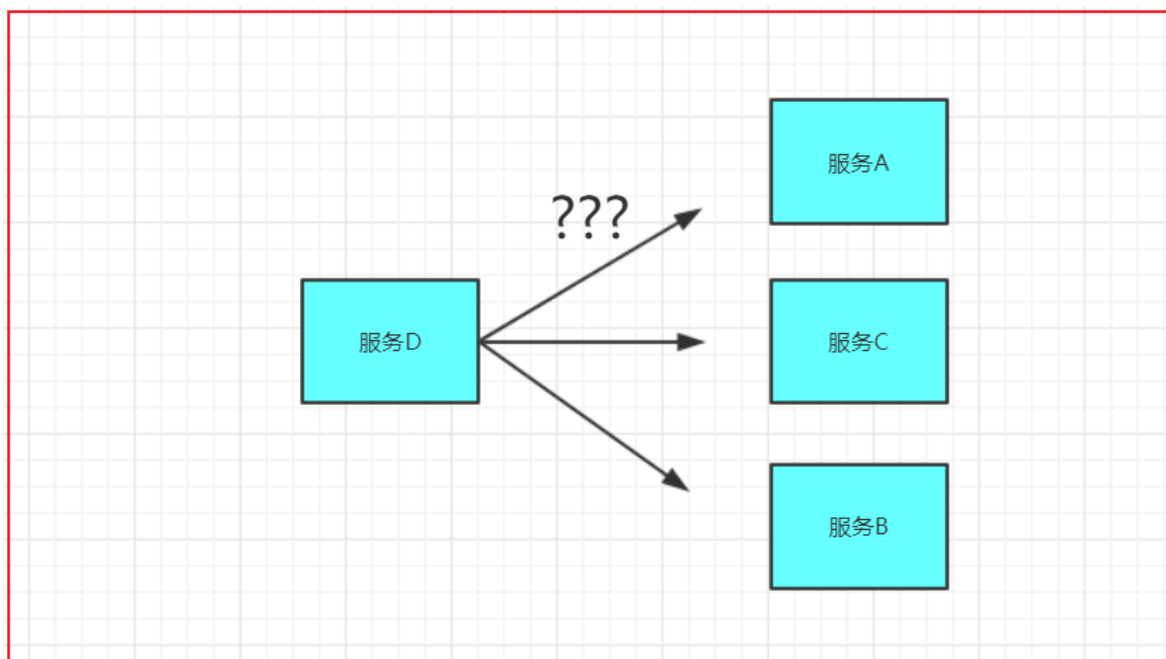
## 五、负载均衡 Spring Cloud Ribbon

为了增加服务并发访问量，我们搭建集群，集群的负载均衡怎么实现？

### 5.1 Ribbon 简介

Ribbon是一个工具框架，实现了HTTP和TCP的客户端负载均衡的工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。它不是一个微服务需要独立部署，但是它**几乎存在于每一个Spring Cloud构建的微服务和基础设施中**。因为微服务间的调用，API网关的请求转发等内容，实际上都是通过Ribbon来实现的，包括明天我们将要介绍的Feign，它也是基于Ribbon实现的工具。所以，对Spring Cloud Ribbon的理解和使用，对于我们使用Spring Cloud来构建微服务非常重要。

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



除此之外的其他算法，了解一下就行：

RoundRobinRule：轮询算法

AvailabilityFilteringRule：根据当前服务是否熔断及并发情况进行负载均衡的算法

WeightedResponseTimeRule：根据服务响应时间来负载均衡的算法

## 5.2 入门案例

实现负载均衡访问用户服务。

如果想要做负载均衡，我们的服务至少2个以上。所有第一步目标

实现步骤：

第一步：启动多个user\_service服务

1. 修改配置文件端口获取方式
2. 编辑应用启动配置
3. 启动两个提供者服务
4. 在注册中心查询是否启动成功

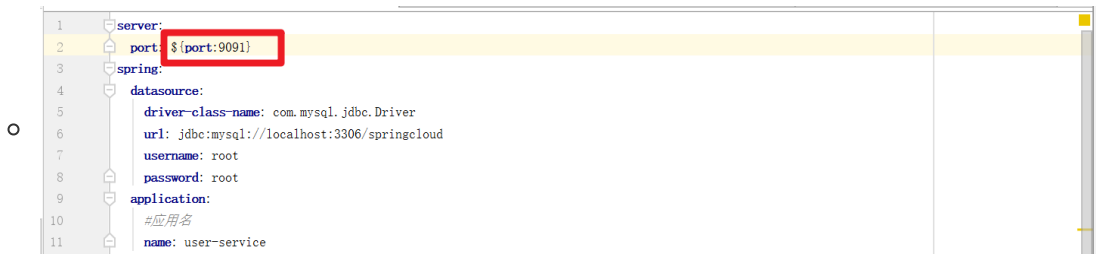
第二步：开启消费者负载均衡

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

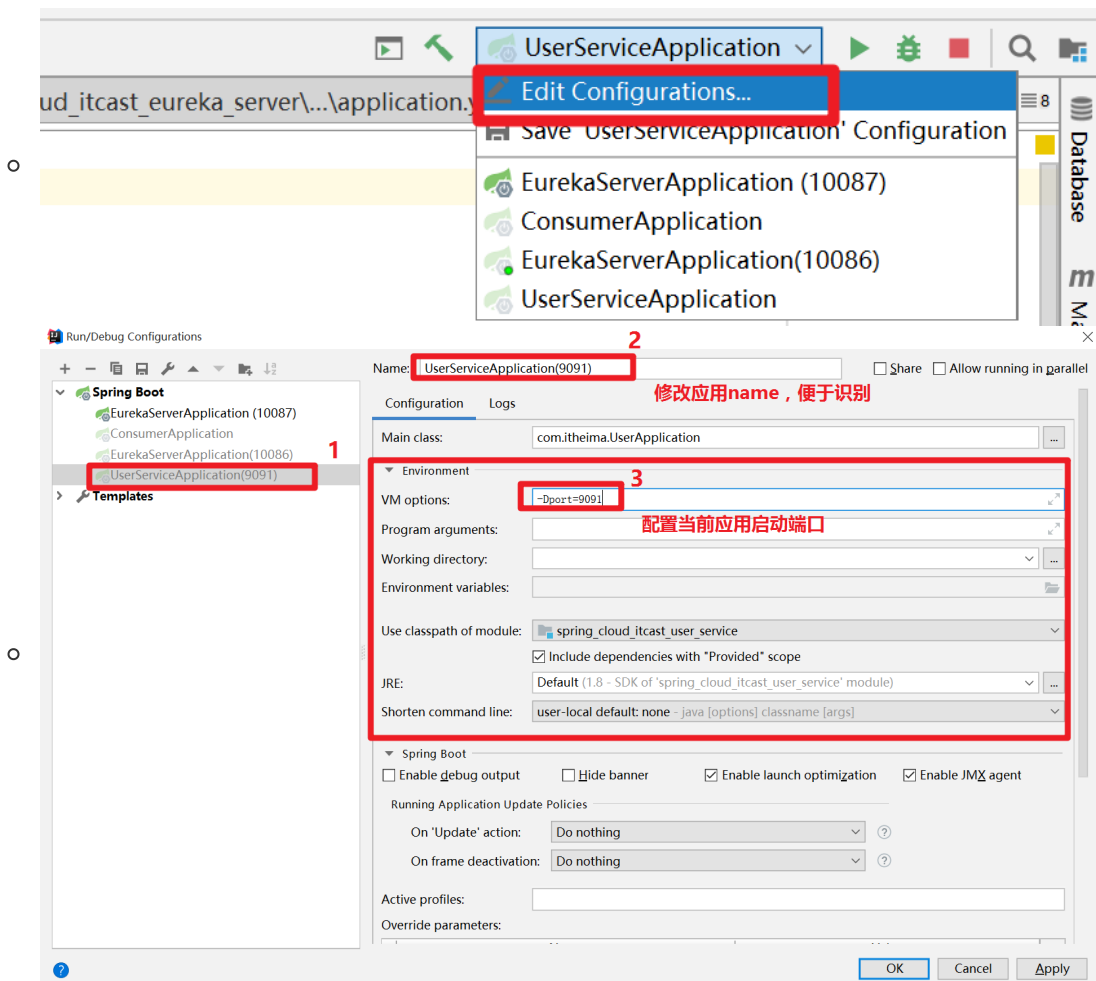
实现过程：

第一步：启动两个user\_service应用

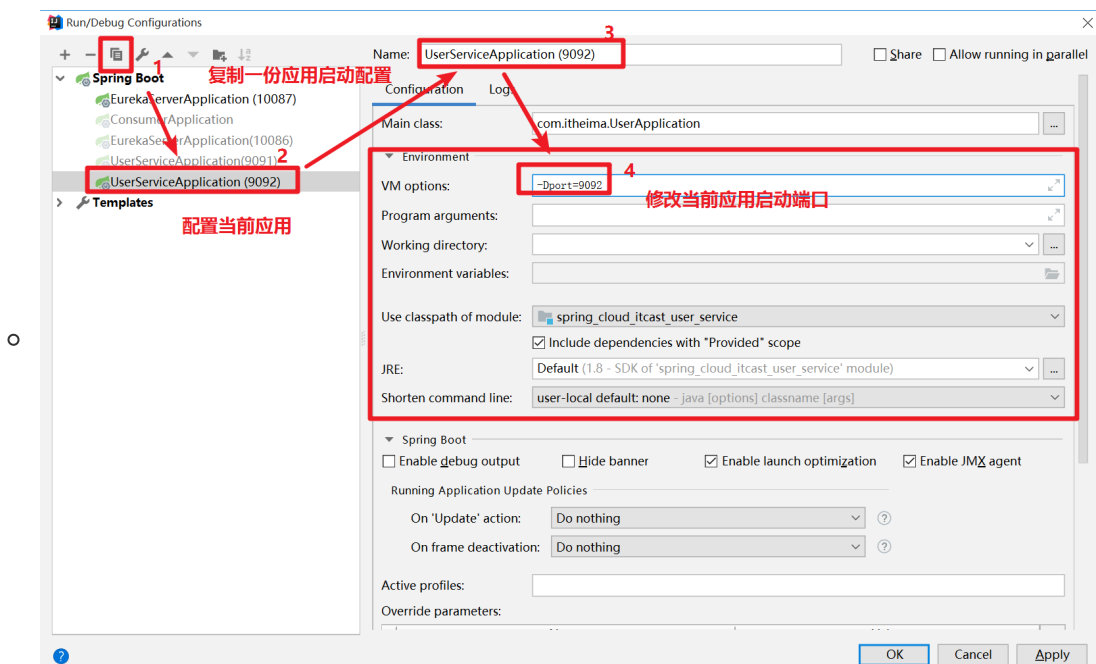
1. 修改UserServiceApplication的application.yml配置文件
  - 端口9091改为，`${port:9091}`



## 2. 编辑应用启动配置

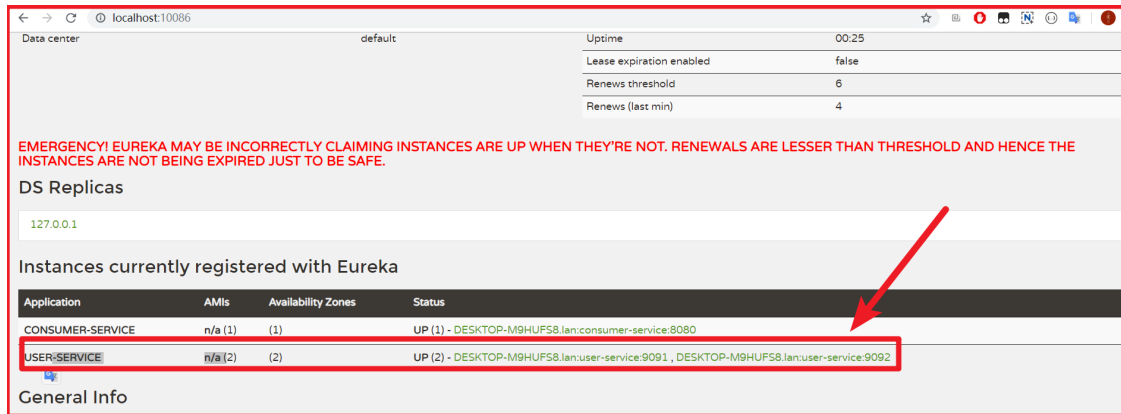


## 3. 复制一份UserServiceApplication



## 4. 启动两个UserServiceApplication应用

## 5. Eureka服务中心可查看，注册成功



第二步：开启消费者调用负载均衡

Eureka已经集成Ribbon，所以无需引入依赖。

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

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

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

```
@RequestMapping("/ribbonconsumer/{id}")
public String queryByIdByRibbon(@PathVariable Integer id){
    String url = "http://user-service/user/findById?id="+id;
    String user = restTemplate.getForObject(url, String.class);
    return user;
}
```

3. 访问页面查看结果；并在9091和9092的控制台查看执行情况

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, RoundRobinRule
user-service:
ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

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

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

## 5.3 负载均衡源码跟踪探究(了解)

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

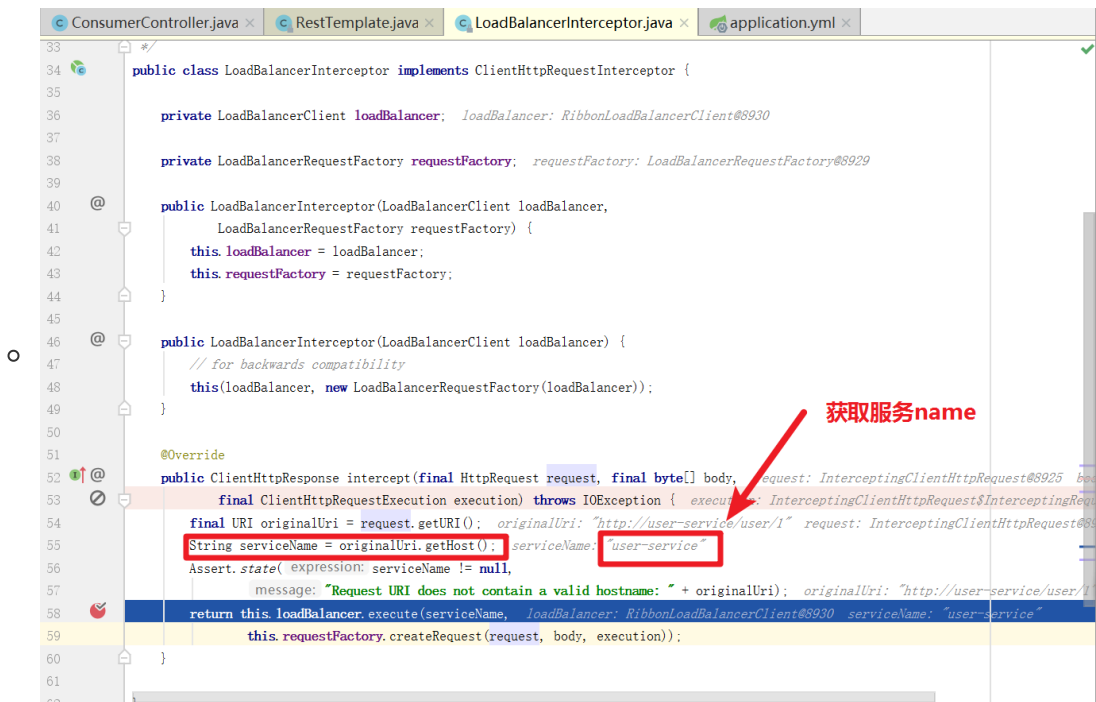
负载均衡器动态的从服务注册中心中获取服务提供者的访问地址(host、port)

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

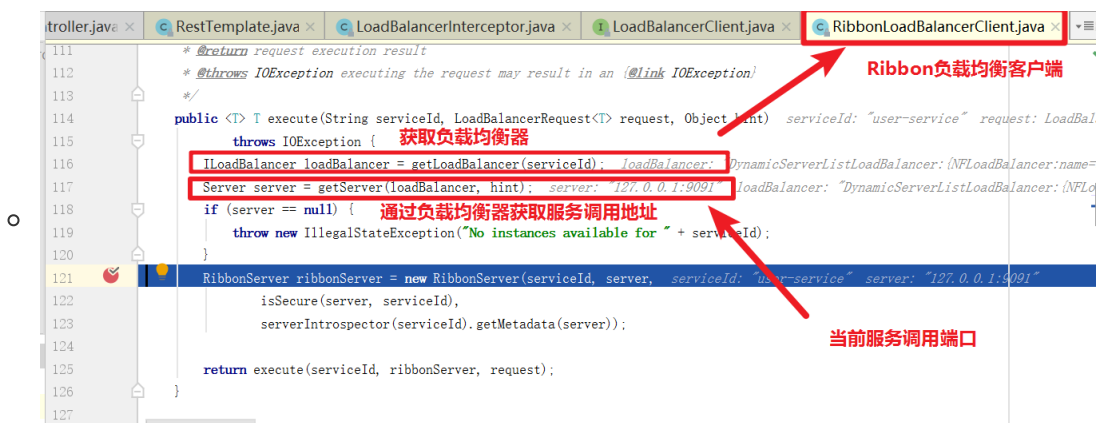
这个类会对RestTemplate的请求进行拦截，然后从Eureka根据服务id获取服务列表，随后利用负载均衡算法得到真正服务地址信息，替换服务id。

源码跟踪步骤:

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

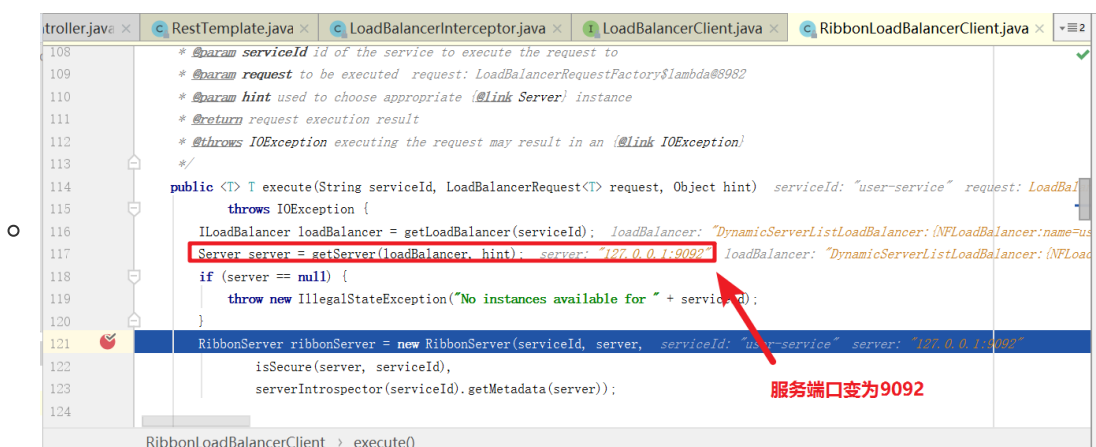


### 2. 继续跟入execute方法: 发现获取了9092端口的服务



- 获取负载均衡器
- 使用负载均衡器从服务列表获取服务

### 3. 再跟下一次, 发现获取的是9091和9092之间切换



### 4. 通过代码断点内容判断, 果然是实现了负载均衡

## 六、熔断器 Spring Cloud Hystrix

服务提供者如果出现故障, 会不会向用户抛出异常页面, 该不该抛出错误页面?



## 6.1 Hystrix 简介



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

Hystrix是什么？

在分布式环境中，许多服务依赖项中的部分服务必然有概率出现失败。Hystrix是一个库，通过添加延迟和容错逻辑，来帮助你控制这些分布式服务之间的交互。Hystrix通过隔离服务之间的访问点、停止级联失败和提供回退选项来实现防止级联出错。提高了系统的整体弹性。与Ribbon并列，也几乎存在于每个Spring Cloud构建的微服务和基础设施中。

Hystrix被设计的目标是：

- 对通过第三方客户端库访问的依赖项（通常是通过网络）的延迟和故障进行保护和控制。
- 在复杂的分布式系统中阻止雪崩效应。
- 快速失败，快速恢复。
- 回退，尽可能优雅地降级。



## 6.2 什么是雪崩效应？

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

Hystrix解决雪崩问题的手段，主要是服务降级(兜底)，线程隔离；



## 6.3 熔断案例

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

**实现步骤：**

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

**实现过程：**

1. 引入熔断的依赖坐标：

- consumer\_service中加入依赖

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

2. 开启熔断的注解

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

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

```
@RequestMapping("/ribbonconsumer/{id}")
@HystrixCommand(fallbackMethod = "queryByIdFallback")
public String queryById(@PathVariable Long id){
    String url = String.format("http://user-service/user/%d", id);
    return restTemplate.getForObject(url,String.class);
}

public String queryByIdFallback(Long id){
    return "对不起，网络太拥挤了！";
}
```

#### 4. 配置熔断策略

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

```
# 配置熔断策略:
# 强制打开熔断器 默认false关闭的。测试配置是否生效
hystrix.command.default.circuitBreaker.forceOpen: false
# 触发熔断错误比例阈值, 默认值50%
hystrix.command.default.circuitBreaker.errorThresholdPercentage: 20
# 熔断后休眠时长, 默认值5秒
hystrix.command.default.circuitBreaker.sleepWindowInMilliseconds: 60000
# 熔断触发最小请求次数, 默认值是20
hystrix.command.default.circuitBreaker.requestVolumeThreshold: 5
# 熔断超时设置, 默认为1秒
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 2000
```

#### 5. 模拟异常代码

```
@GetMapping("{id}")
@HystrixCommand(fallbackMethod = "queryByIdFallback")
public String queryById(@PathVariable Long id){
    //如果参数为1抛出异常, 否则 执行REST请求返回user对象
    if (id == 1){
        throw new RuntimeException("too busy!!!");
    }
    String url = String.format("http://user-service/user/%d", id);
    return restTemplate.getForObject(url,String.class);
}
```

#### 6. 测试熔断的情况(模拟请求超时):

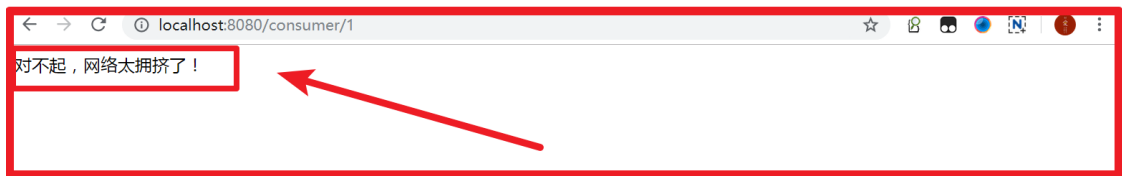
1. 访问超时: 服务提供者线程休眠超过2秒, 访问消费者触发fallback方法。

```
@Service
public class UserService {
    @Autowired
    private UserMapper userMapper;

    public User queryById(Long id){
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return userMapper.selectByPrimaryKey(id);
    }
}
```

2. 服务不可用: 停止user-service服务提供者。访问消费者触发fallback方法。

7.

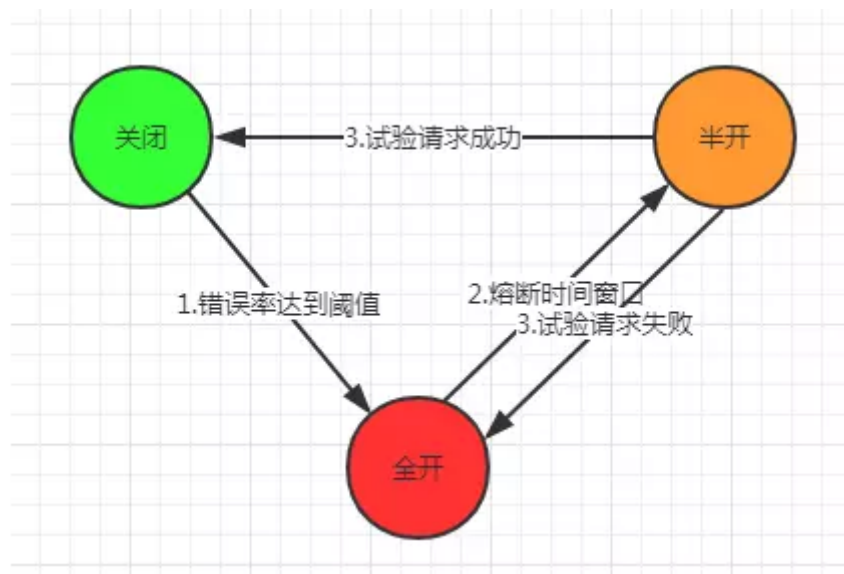


## 6.4 熔断原理分析

熔断器的原理如同电力过载保护器。

熔断器状态机有3个状态：

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



【Hystrix熔断状态机模型：配图】

熔断器的核心解决方案：线程隔离和服务降级。

- 线程隔离。
- 服务降级(兜底方法)。

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

### 什么时候熔断：

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

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

## 6.5 扩展-服务降级的fallback方法：

注意：熔断服务降级方法必须保证与被降级方法相同的参数列表和返回值。

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="queryByIdFallBack")用来声明一个降级逻辑的fallback兜底方法

### 2. 类上默认服务降级的fallback兜底方法

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

```
@RestController
@RequestMapping("/consumer")
@DefaultProperties(defaultFallback = "defaultFallBack")//开启默认的
FallBack，统一失败降级方法(兜底)
public class ConsumerController {
    @GetMapping("/{id}")
    @HystrixCommand
    public String queryById(@PathVariable Long id){
        //如果参数为1抛出异常，否则 执行REST请求返回user对象
        if (id == 1){
            throw new RuntimeException("too busy!!!");
        }
        String url = String.format("http://user-service/user/%d", id);
        return restTemplate.getForObject(url,String.class);
    }
    /**
     * queryById的降级方法
     */
    public String queryByIdFallBack(Long id){
        return "对不起，网络太拥挤了！";
    }
    /**
     * 默认降级方法
     */
    public String defaultFallBack(){
        return "默认提示：对不起，网络太拥挤了！";
    }
}
```

