# Spring Cloud

**一站式微服务架构的解决方案**

# 课程安排

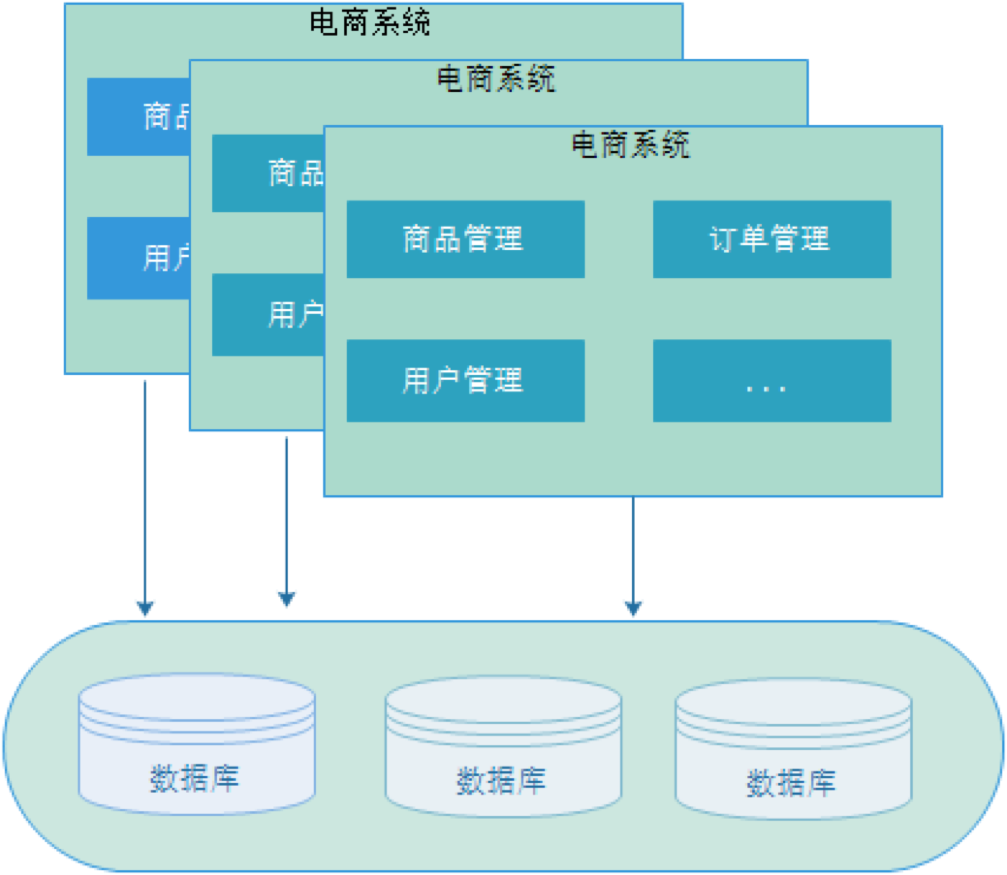
1. 理解微服务(能说出微服务架构的特点)
2. Spring cloud技术栈(能说出微服务技术栈的构成)
3. 服务治理(理解spring cloud Eureka服务治理流程)
4. 负载均衡(理解Spring cloud Ribbon 负载均衡的应用方法)
5. 容错保护(理解spring cloud Hystrix 断路保护的应用方法)
6. 服务网关(理解Spring cloud zuul 网关的作用与方法)
7. 了解spring cloud子项目(了解Spring Cloud Config /Spring Cloud Bus /Spring Cloud Sleuth等其它子项目的应用场景)

# 理解微服务

## 软件架构的发展

软件架构的发展:单体架构—》垂直架构—》SOA架构—》微服务架构

### 单体架构



特点：

1. 所有的功能都集中在一个工程中
2. 所有的功能打成一个War包部署运行，功能不可分割
3. 应用与数据库是分开部署的
4. 通过部署应用集群和数据库集群来提交性能

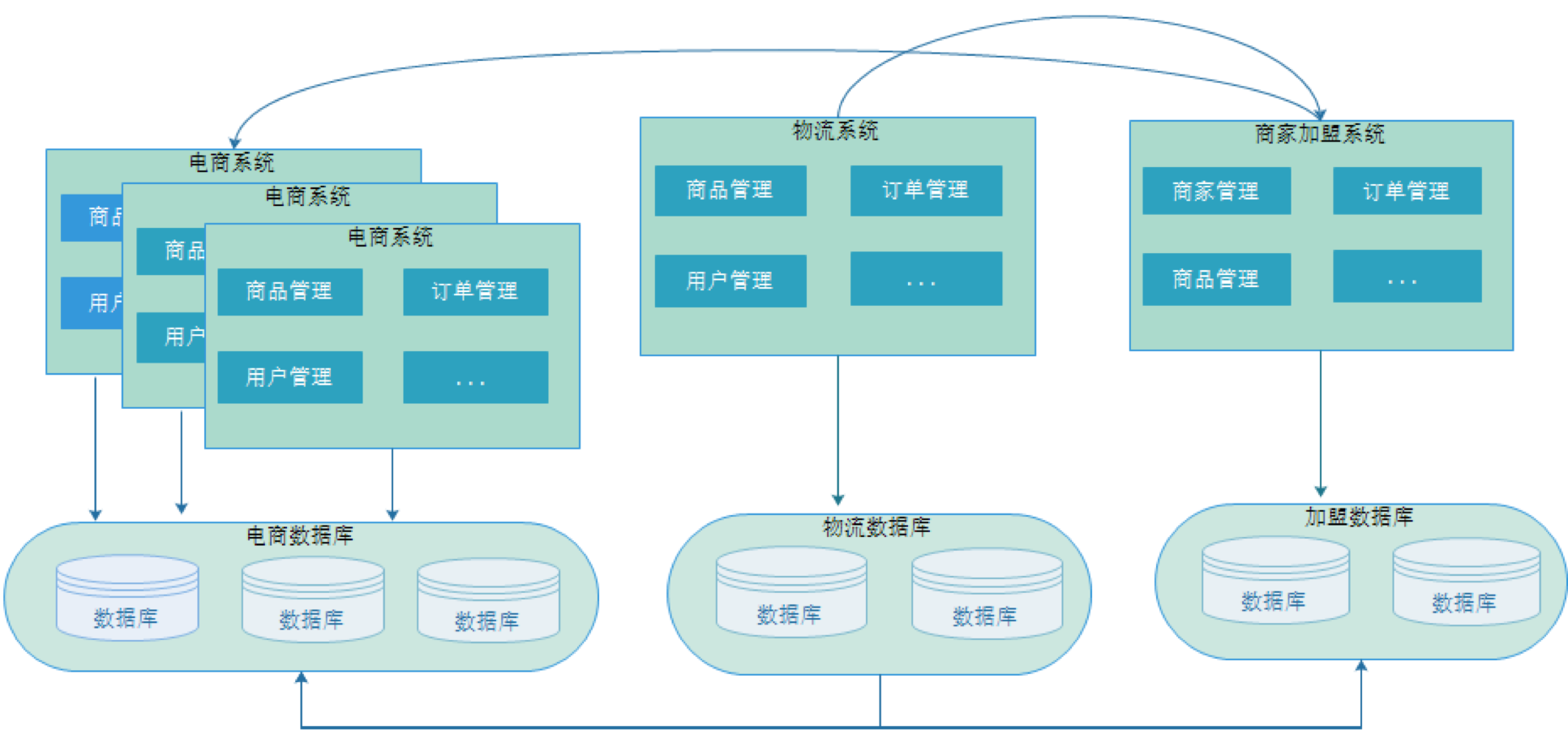
优点：

项目架构简单，前期开发成本低，开发周期短，小型项目首选

缺点：

1. 全部功能都集成在一个系统中，对于大型项目，不容易开发，不容易扩展与维护
2. 系统的扩展只能通过集群的结点扩展，成本高，有瓶颈
3. 技术栈受限

### 垂直架构



特点：

1. 以单体架构为项目单位进行功能划分，把大功能的项目划分成一个个小功能的项目
2. 项目与项目之间存在数据冗余，耦合度高，存在很多相同的功能模块。
3. 接口比较多，因此我们要对接口中的数据进行同步，接口调用次数变高，通过网络实现接口调用。

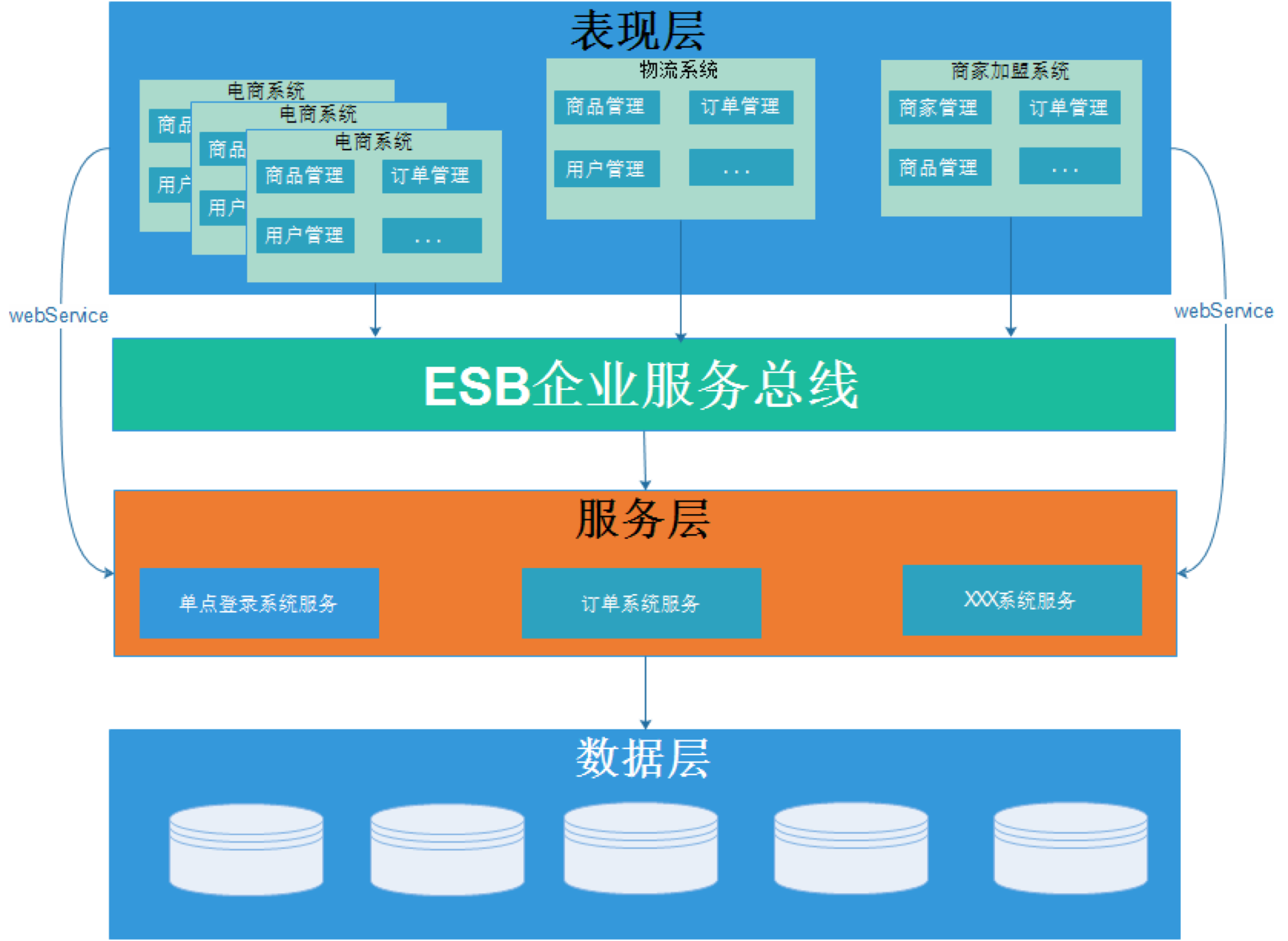
优点：

1. 项目架构简单，前期开发成本低，周期短，中型项目首选。
2. 通过垂直划分，原来的单体项目不会很大。
3. 不同的项目可以采用不同的技术实现。

缺点：

1. 全部抽取的功能都集中在一个系统中，对于大型项目，也是不容易开发和扩展的。
2. 系统的扩展只能通过集群的结点扩展，成本高，有瓶颈

### SOA架构



特点：

1. 基于SOA的思想将重复的功能利用SOA思想抽取为组件，以服务的方式为各系统提供服务。
2. 各个系统（项目）与服务之间的调用使用ESB企业服务总线进行（WebService/RPC）调用。
3. 把ESB企业服务总线进行抽取，在ESB企业服务总线中解决调用问题。

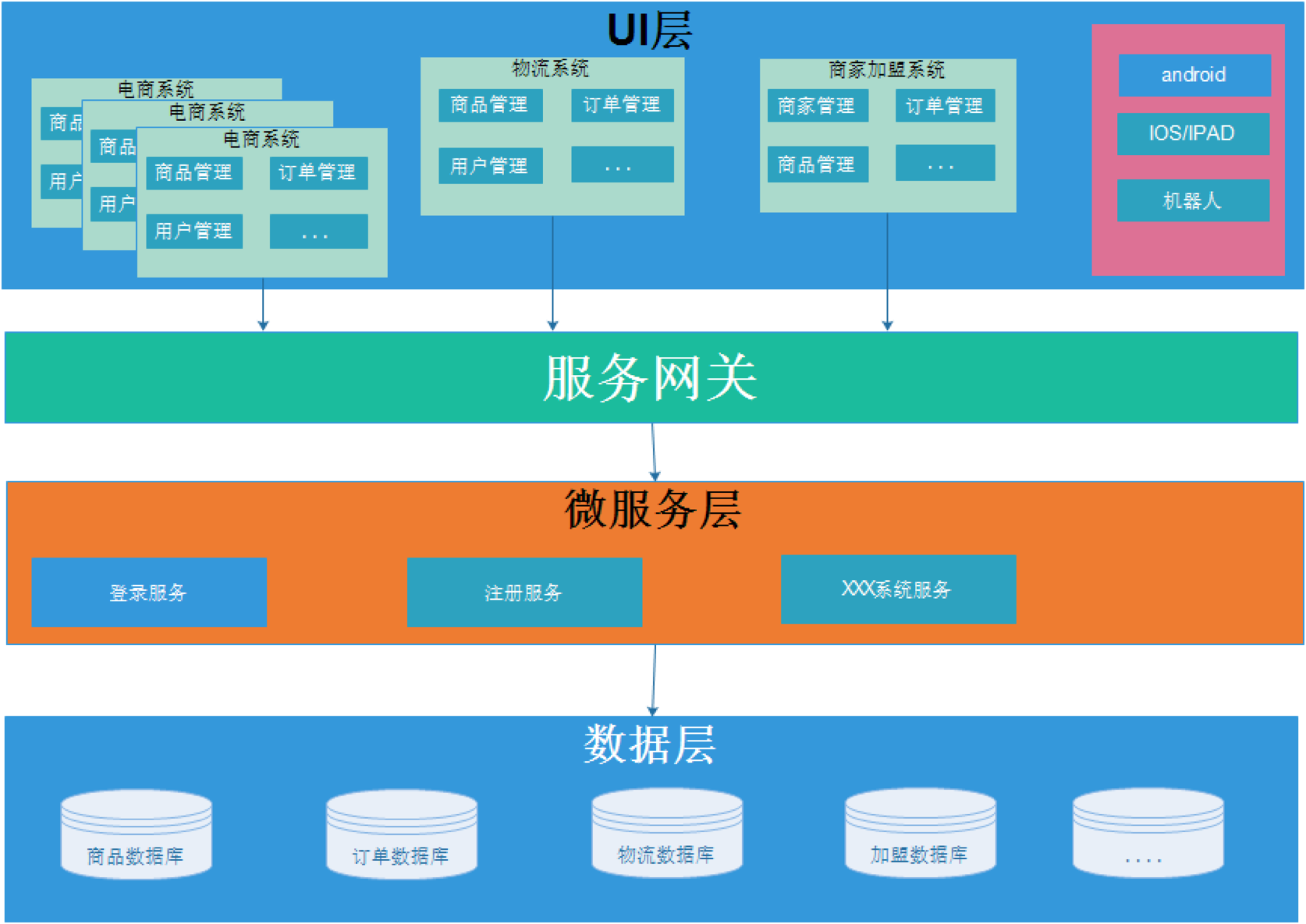
优点：

1. 将重复的服务进行了抽取，提高了开发效率，提高了系统的代码和功能的重用性、维护性
2. 可以针对不同的服务定制集群
3. 采用ESB企业服务总线减少系统之的耦合度。

缺点：

1. 系统与服务之间的界限模糊，不利于开发和维护
2. 虽然使用ESB,但是服务接口的协议不固定，种类很多，不利于维护
3. 抽取服务的粒度过大，系统与服务之间存在一定的耦合度。

### 微服务架构



特点：

1. 将系统的服务完全独立出来，将服务层抽取了一个个微服务。
2. 微服务单一原则
3. 微服务之间使用统一restful等轻量级协议进行传输

缺点：

1. 微服务服务众多，服务的治理成本变高，不利于系统维护
2. 分布式系统开发成本变高（容错、分布式事务处理等），对团队的挑战高。

优点：

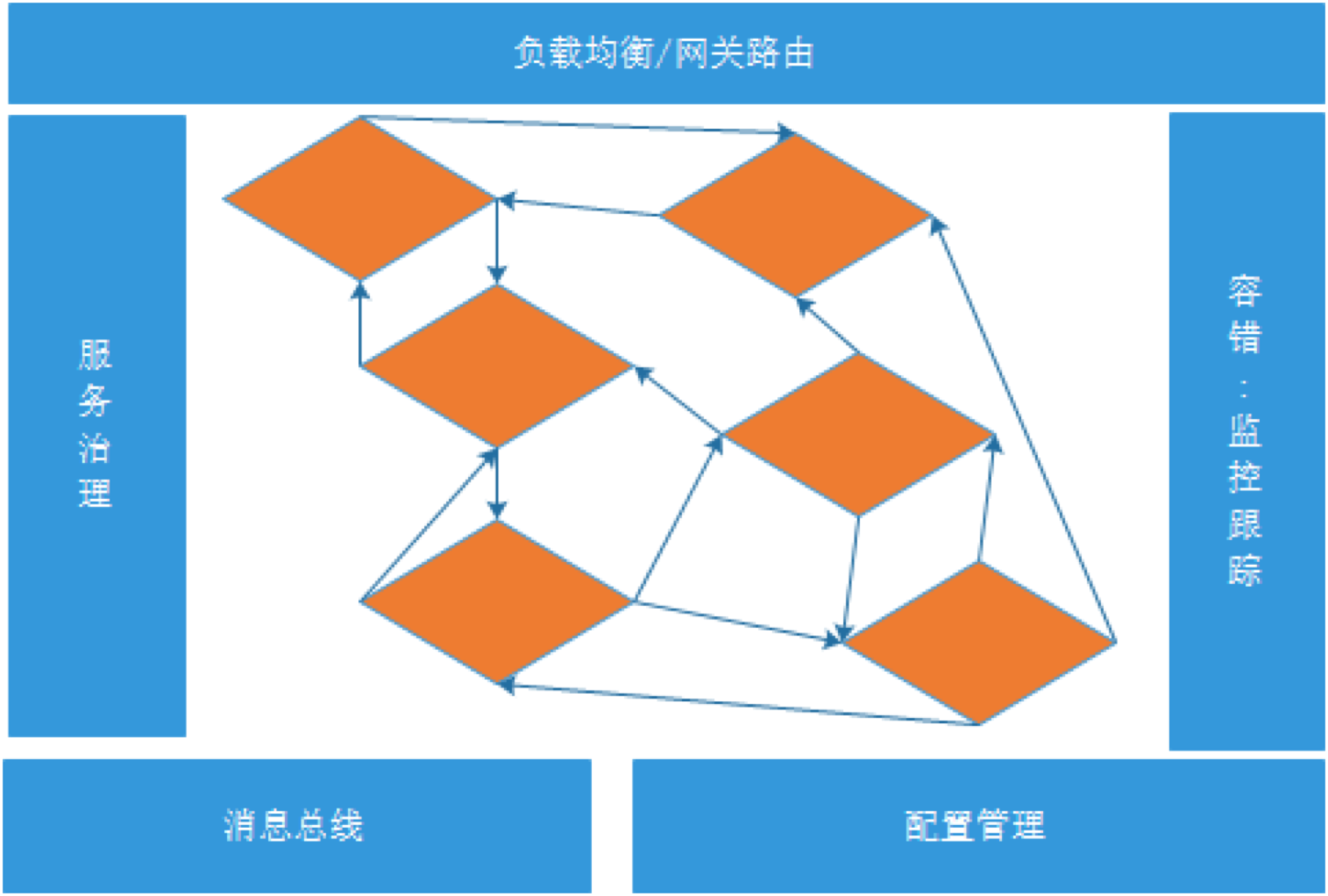
1. 服务的拆分粒度变细，有利于资源的重复利用，提高开发效率
2. 可以精准定制优化的方案，提高系统的可维护性
3. 微服务采用中心化思想，微服务之间使用统一restful等轻量级协议进行传输,相比ESB企业服务总线更轻量级。
4. 更适用于互联网项目的迭代开发。

## 什么是微服务

为了适应企业开发业务的发展，提高软件研发的生产力，降低软件的开发成本，软件的架构做了升级和优化，将独立的系统拆分成若干的服务，每个服务的运行在不同的系统中，服务与服务之间使用轻量级的Http协议（比喻Restful协议）进行数据传输，每个服务的独立性很强，耦合度低的特点，这种设置使用高内聚，低耦合的效果的软件设置，称为微服务架构。

# Spring Cloud技术栈

## 微服务技术栈



负载均衡、网关路由：高可用，集群部署，校验，请求转发，服务集成。

服务治理：服务注册与发现

容错：避免雪崩

监控跟踪：监控资源的利用，服务响应，容器资源的利用情况

消息总线：消息队列，异步通信。

## Spring Cloud是什么？

Spring Cloud 是为开发人员对微服务架构提供完整的一站式解决方案，Spring Cloud 是若干技术的集合，包括：spring cloud config/bus/..等20多种子项目构成。它提供服务治理、服务网关、智能路由、负载均衡、断路器、监控跟踪等领域的解决方案。

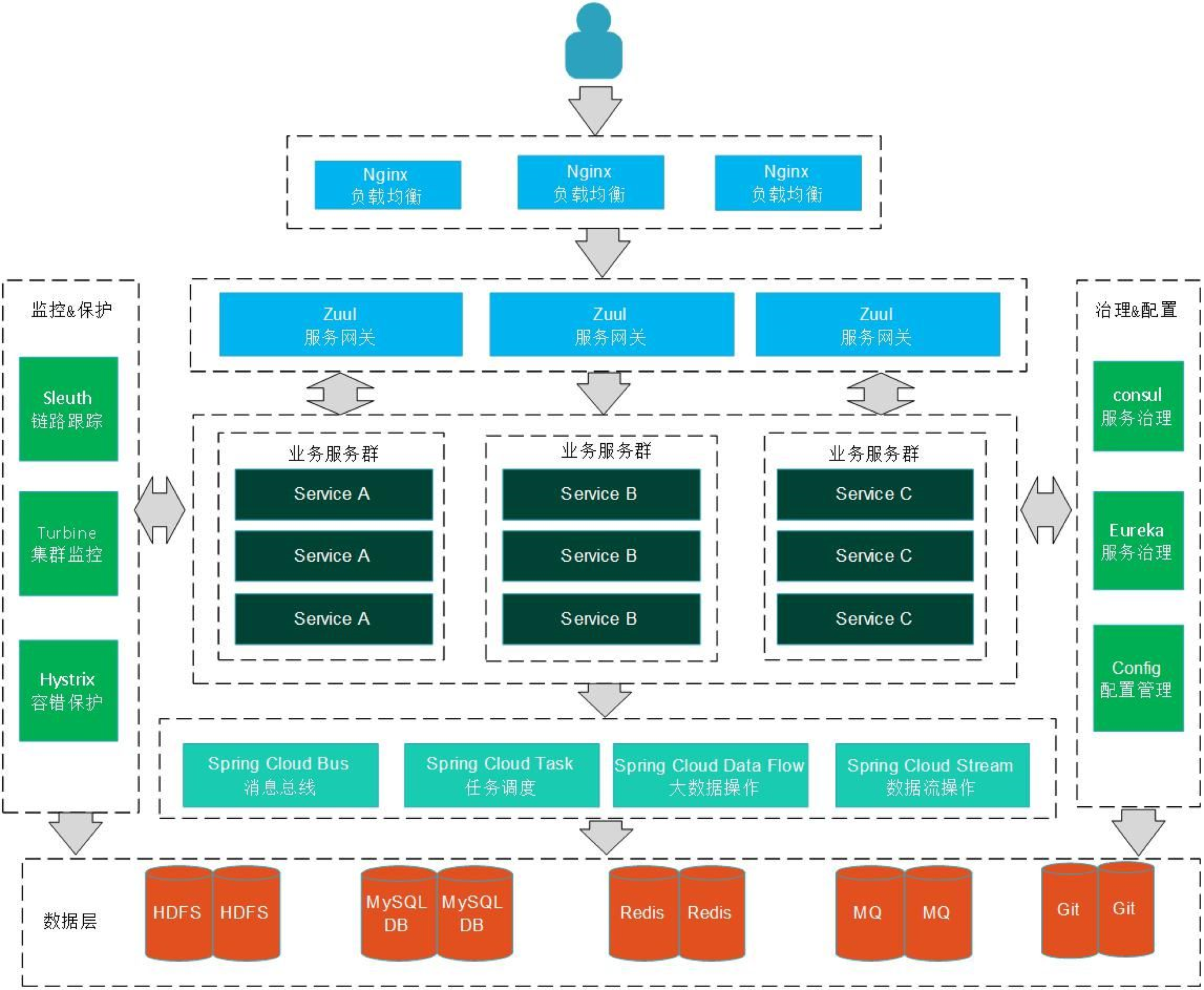
## Spring Cloud技术栈

服务治理：Dubbo（阿里巴巴）、Dubbox(当当网)、Eureka(Netfix)等

配置治理：DisConf(百度)、Qconf(360)、diamond(淘宝)等

服务跟踪：Hydra(京东)、Zipkin(Twitter)、Sleuth(Spring Cloud)等

Spring Cloud提供一站式微服务的解决方案：



## 为什么要使用Spring Cloud

通过微服务架构的优点，我们清楚可以提高生产力，但是分布式系统本身存在很多技术上的问题要解决，为了让大部分公司能够快速解决微服务产生的问题，spring公司推出spring cloud为了降低开发成本，它提供了一站式的微服务解决方案，之后很多公司就开始使用spring cloud，我们相信在不久spring cloud一定会一统江湖。

# 开发环境

Jdk8/maven3.x/spring boot最新/Spring cloud最新

# 服务治理

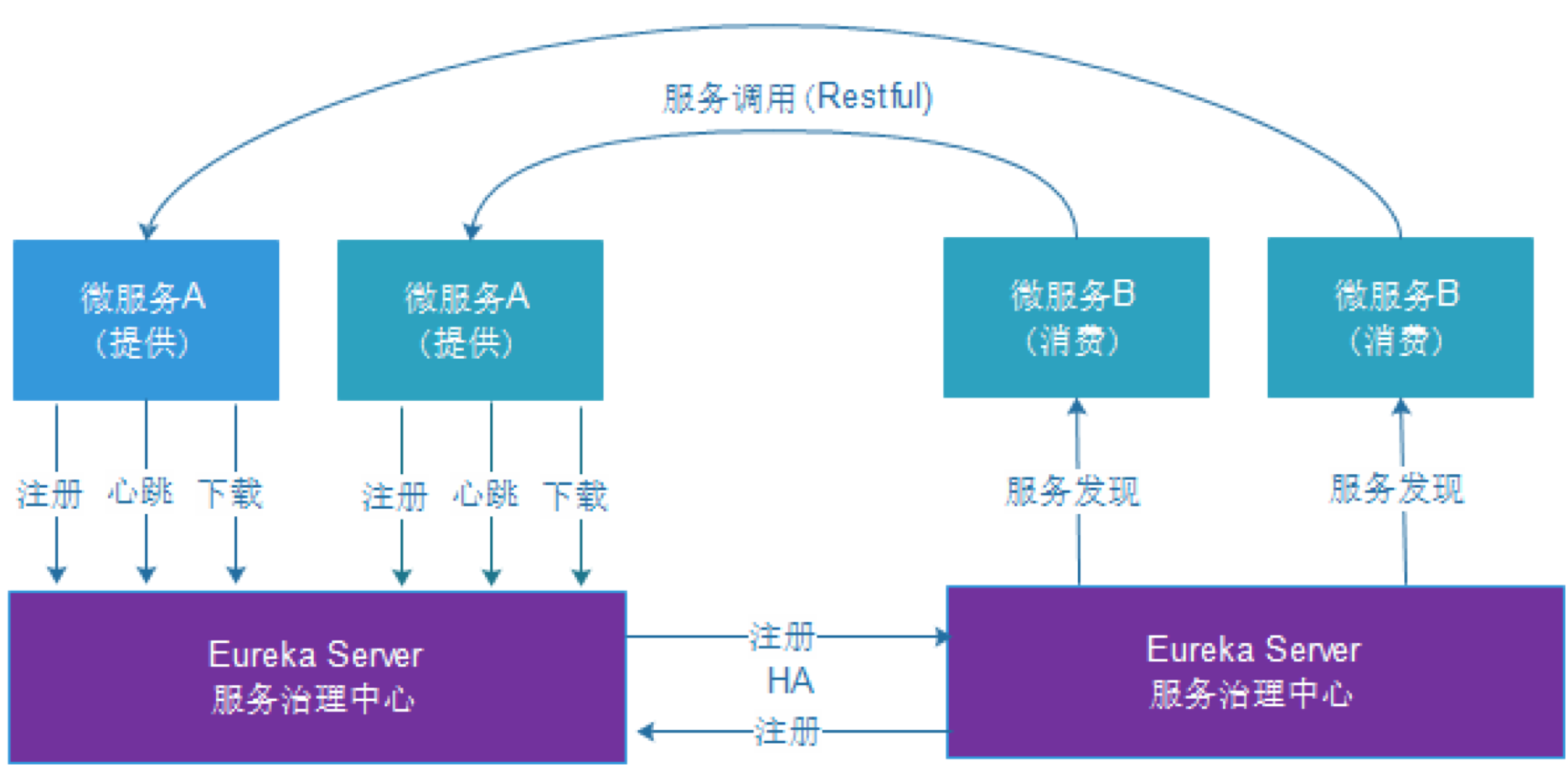
## 什么是服务治理

微服务架构中服务数量众多，导致维护服务的成本变高，服务治理为了解决服务治理的问题而产生。服务治理作用是让维护人员从人工治理中解放出来，由服务自已来维护，微服务为服务的提供方主动向治理中心注册，服务调用方可以通过治理中心查询所需要的服务，并且调用所需要的服务。（类似于Dubbo的注册中心）

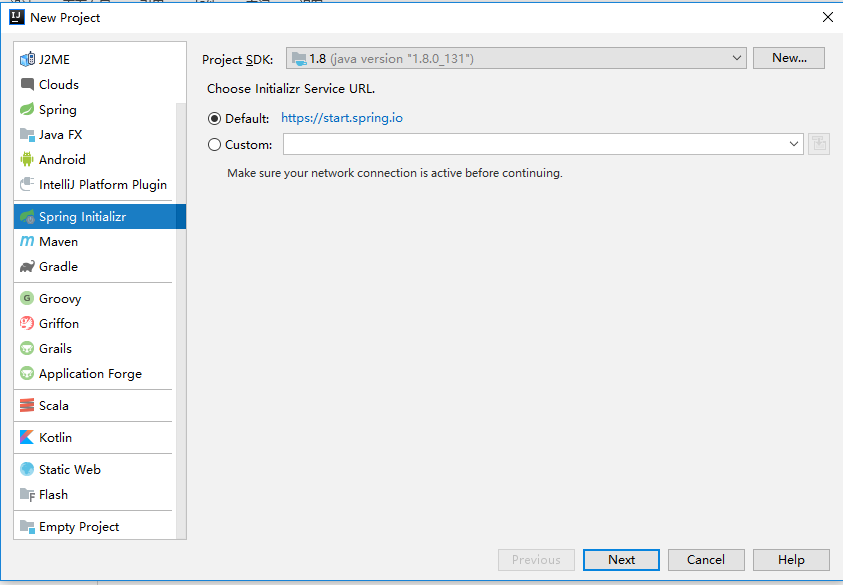
## Spring Cloud Eureka介绍

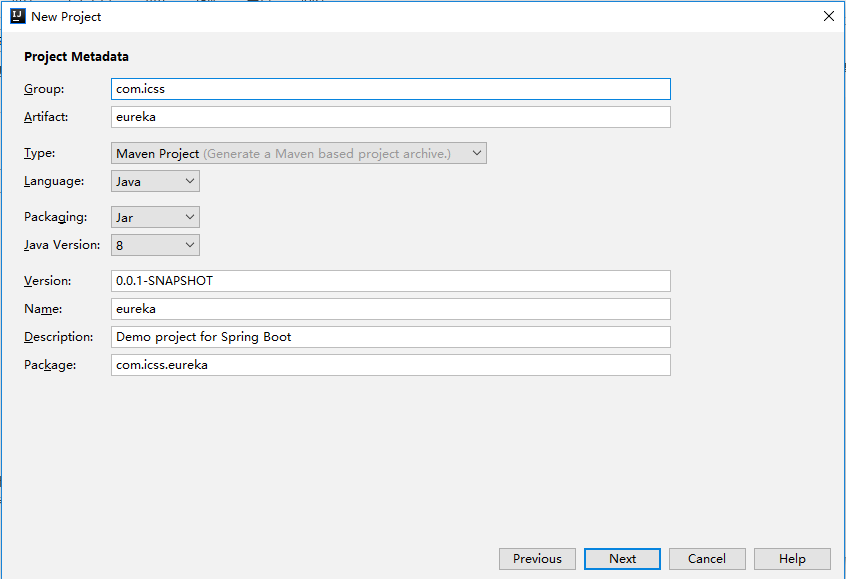
Spring Cloud Eureka 是对netflix Eureka做了二次包装，由两部分组成：Eureka Server(注册中心)和Eureka Client（服务注册）。

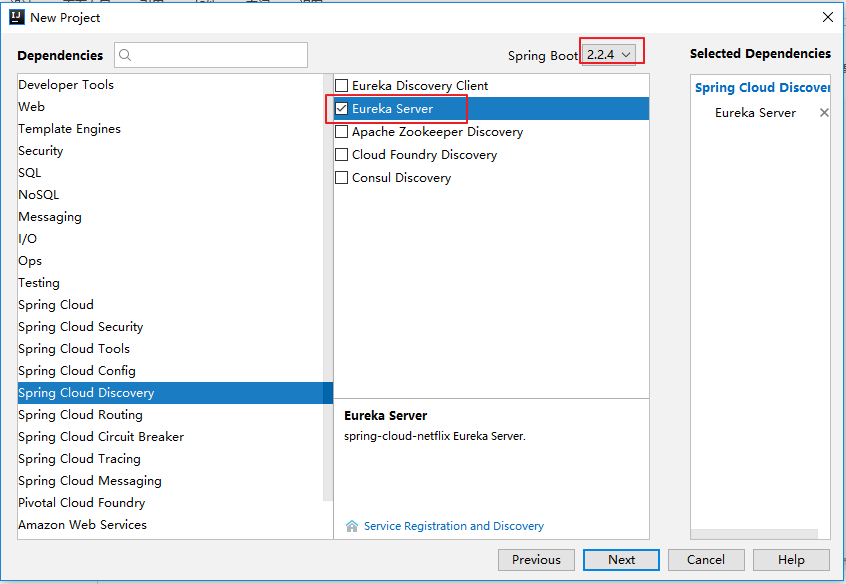
## 架构



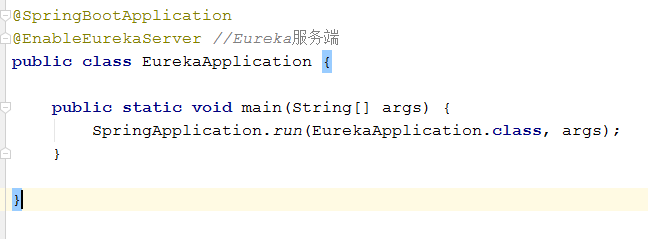
## 创建Eureka服务端工程





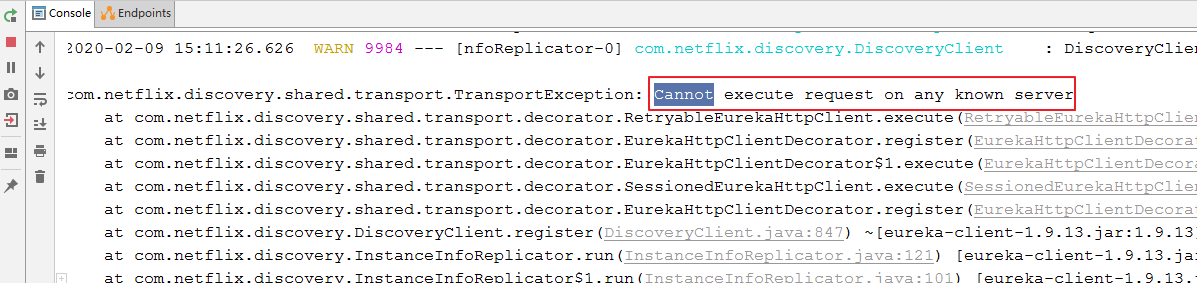


## 设置Eureka服务端

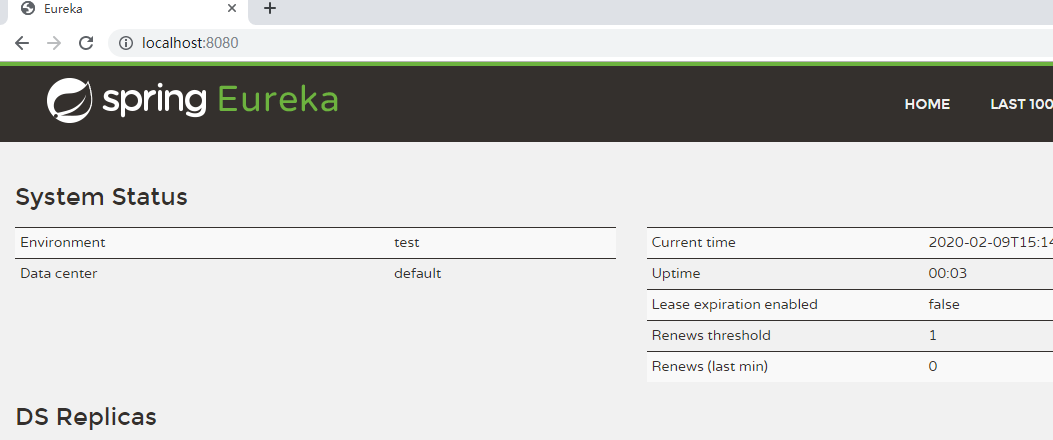


## 报错

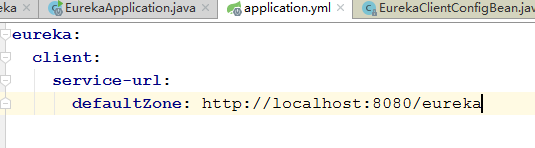
原因：自已也要注册自已，所以报这个错误是正常的。



## 访问Eureka服务器



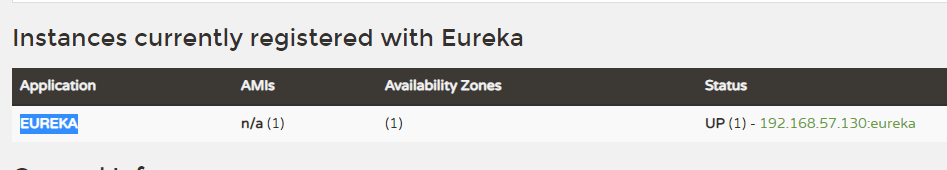
## 设置注册中心地址



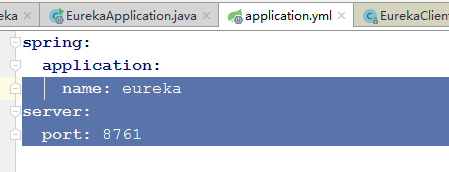
## 设置应用名称



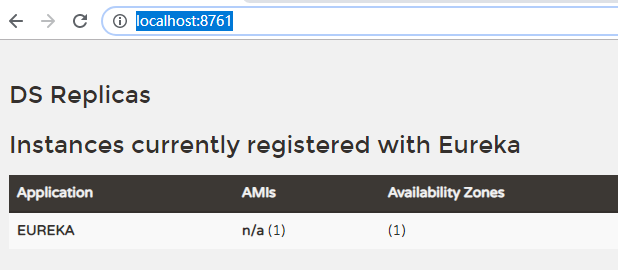
### 测试效果



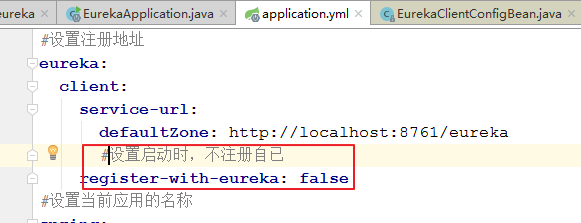
## 修改应用的端口号



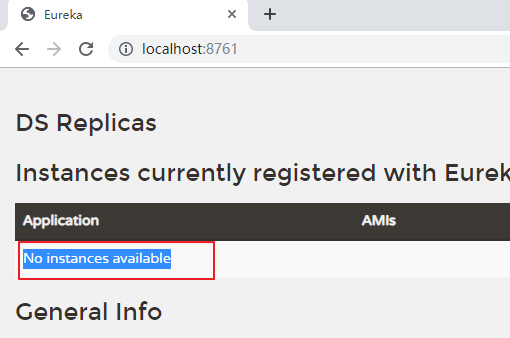
### 测试效果



## 设置启动时，不注册自已

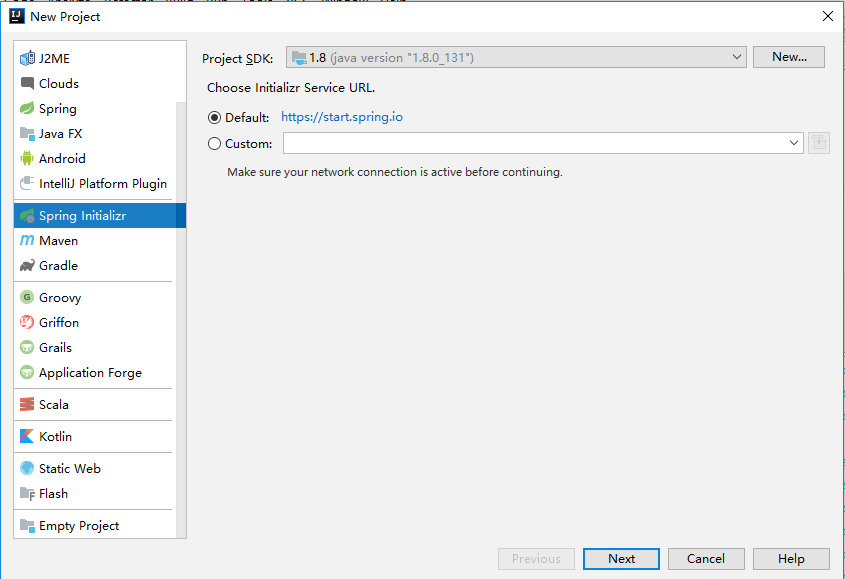


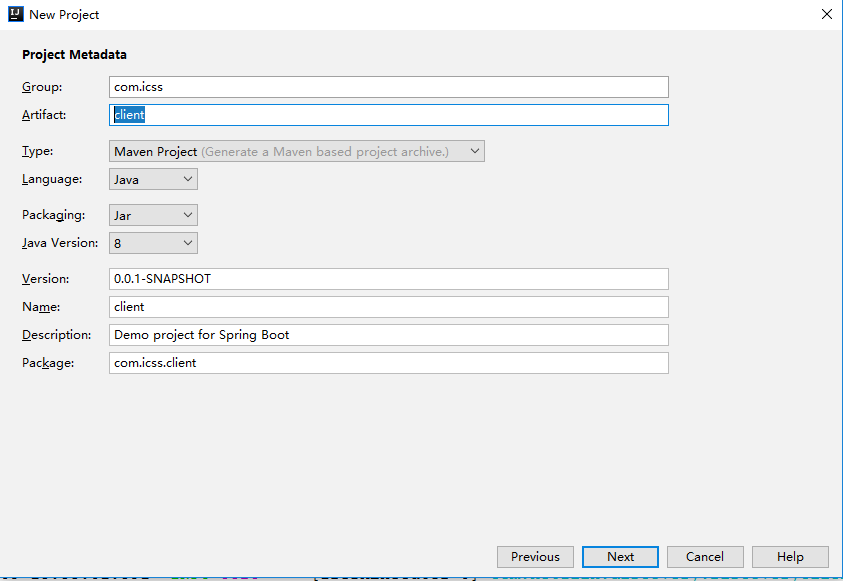
### 测试效果

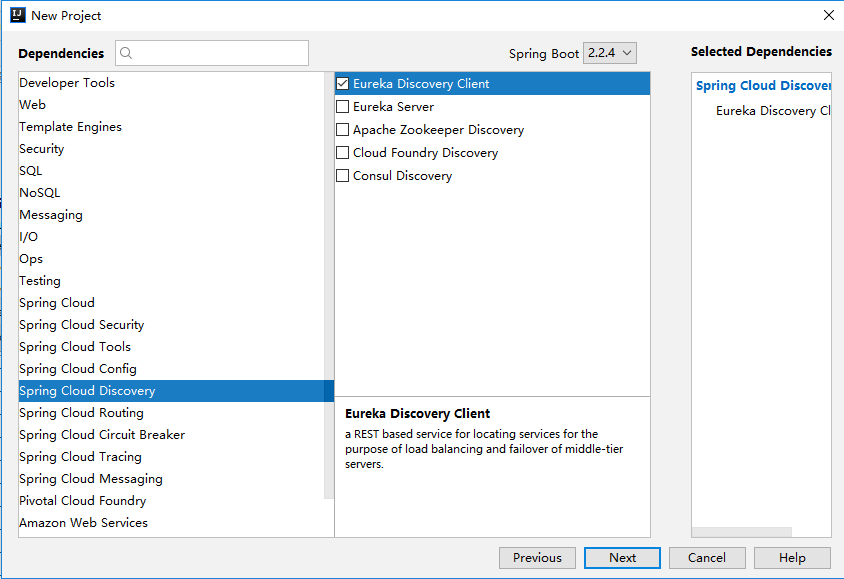


# 服务中心客户端（Spring Cloud Eureka）

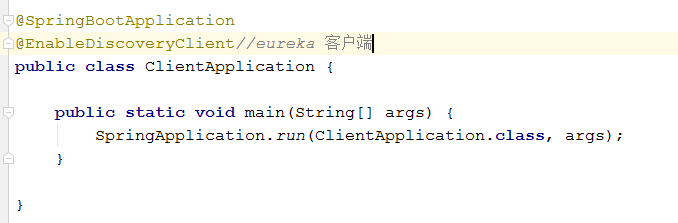
## 创建客户端工程



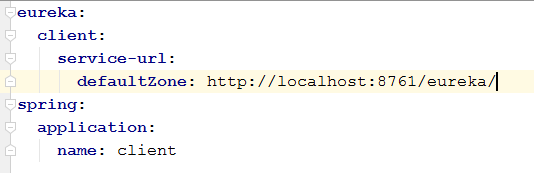




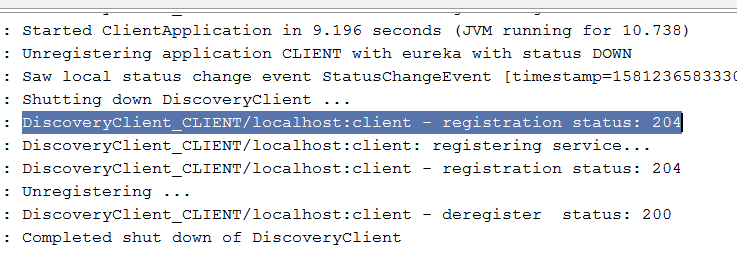
## 设置客户端应用



## 设置连接注册中心



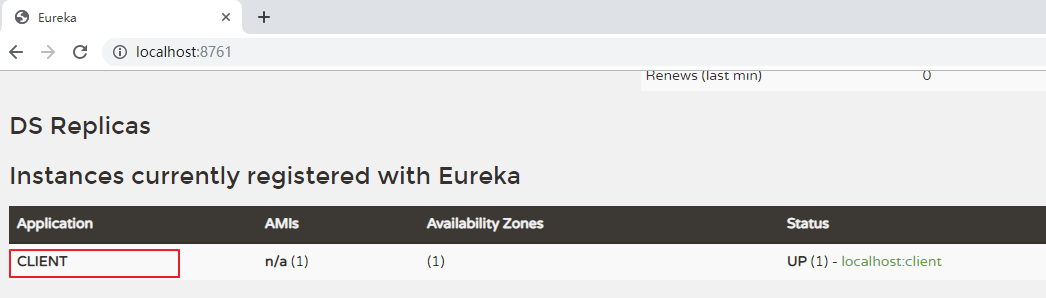
## 报错



## 导入依赖

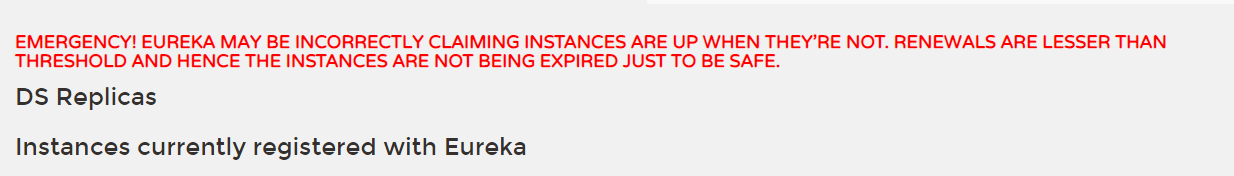
|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-web</**artifactId**> </**dependency**> |

## 测试效果



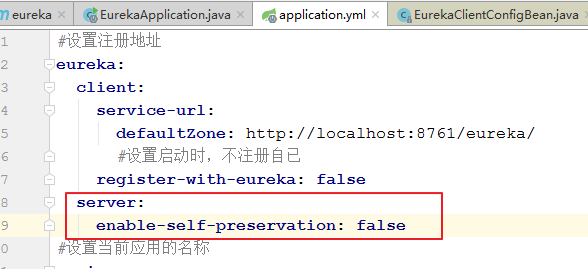
## Eureka服务端自我保护

Eureka服务端会通过心跳连接来实现健康检查，宁可信其有，不可信其无的机制，所以出现一些客户端下线了，不稳定而已。

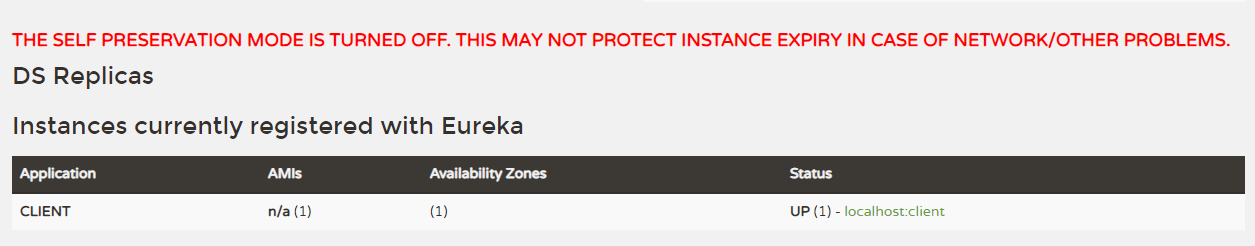


## 可以关闭健康检查

在测试环境下可以关闭，但是在生产环境中，建议开启

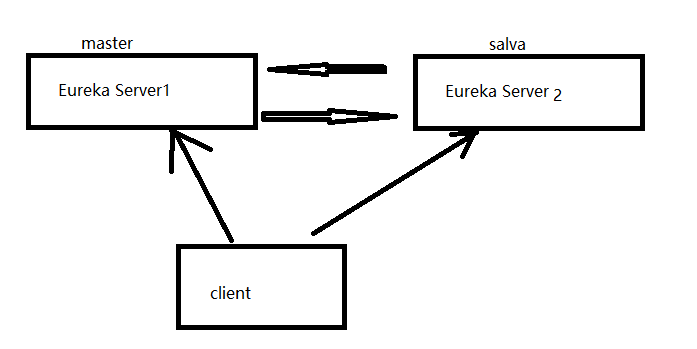


## 关闭后的效果

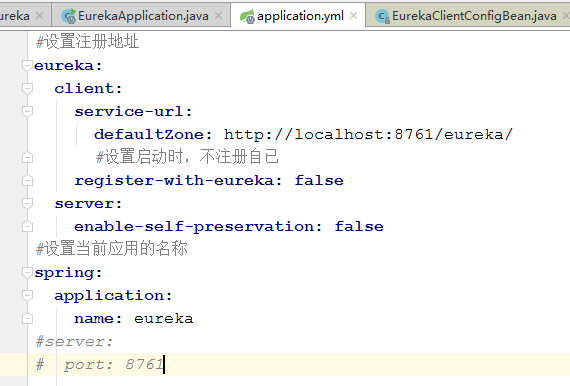


# 服务注册中心高可用

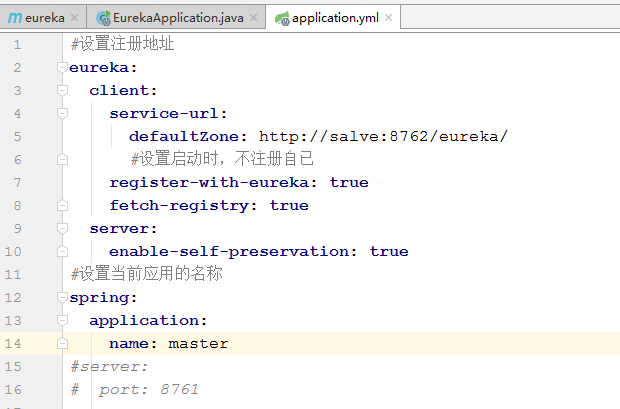
## Eureka高可用



## 注掉端口号

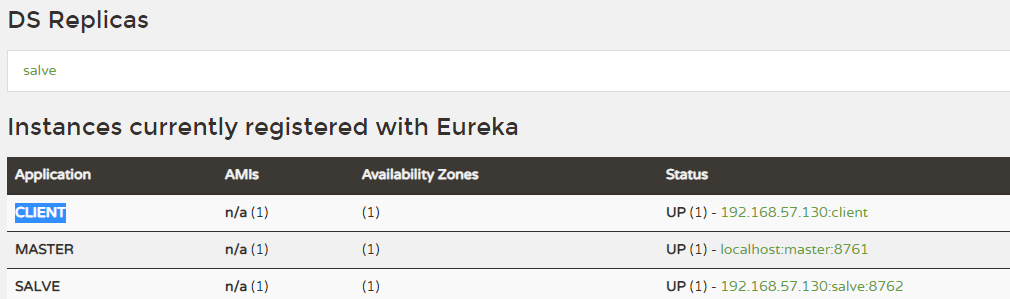


## 配置两台服务器





## 使用Euraka客户端测试



# Euraka微服务注册实践

需求：根据用户名查询用户信息

## 分析使用技术

Spring boot –springmvc

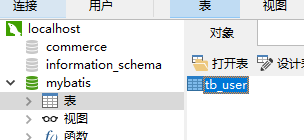
Spring cloud

Mybatis

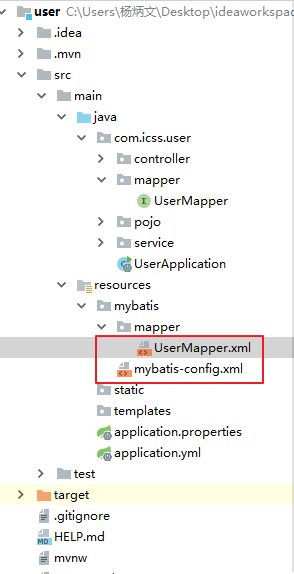
## 创建微服务注册实践工程

略。。。

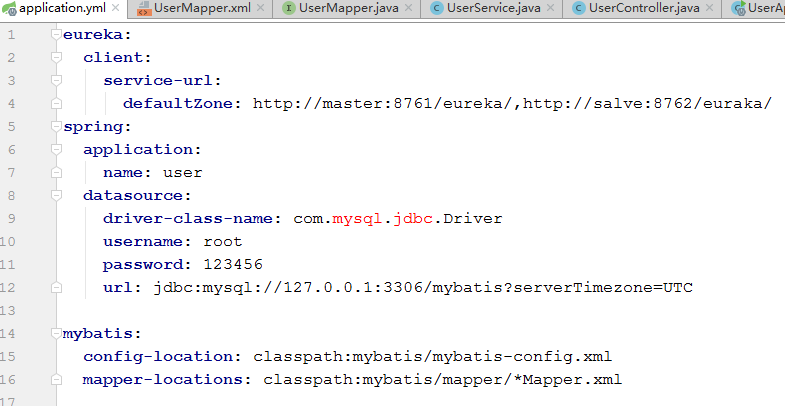
## 创建数据库并导入sql



## 整合框架



## Application.yml



## UserApplication启动类

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient @MapperScan(**"com.icss.user.mapper"**) **public class** UserApplication {   **public static void** main(String[] args) {  SpringApplication.*run*(UserApplication.**class**, args);  }  } |

## UserController

|  |
| --- |
| @Controller **public class** UserController {   @Autowired  **private** UserService **userService**;   @GetMapping(**"/user/{username}"**)  @ResponseBody  **public** ResponseEntity<User> queryUserByUsername(@PathVariable(**"username"**) String username){   **try** {  User user = **this**.**userService**.queryUserByUsername(username);  **return** ResponseEntity.*status*(HttpStatus.***OK***).body(user);  } **catch** (Exception e) {  e.printStackTrace();  }  **return** ResponseEntity.*status*(HttpStatus.***INTERNAL\_SERVER\_ERROR***).body(**null**);  } } |

## UserService

|  |
| --- |
| @Service **public class** UserService {   @Autowired  **private** UserMapper **userMapper**;   **public** User queryUserByUsername(String username){  **return this**.**userMapper**.queryUserByUsername(username);  } } |

## UserMapper

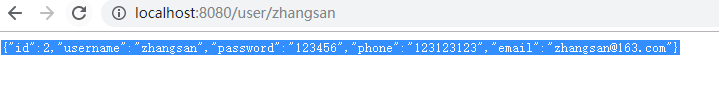
|  |
| --- |
| @Component(**"userMapper"**) **public interface** UserMapper {  **public** User queryUserByUsername(String username); } |

## User实体类

|  |
| --- |
| **package** com.icss.user.pojo;  **public class** User {  **private** Long **id**;  **private** String **username**;  **private** String **password**;  **private** String **phone**;  **private** String **email**;   **public** Long getId() {  **return id**;  }   **public void** setId(Long id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getPassword() {  **return password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   **public** String getPhone() {  **return phone**;  }   **public void** setPhone(String phone) {  **this**.**phone** = phone;  }   **public** String getEmail() {  **return email**;  }   **public void** setEmail(String email) {  **this**.**email** = email;  }   **public** User() {  }   **public** User(Long id, String username, String password, String phone, String email) {  **this**.**id** = id;  **this**.**username** = username;  **this**.**password** = password;  **this**.**phone** = phone;  **this**.**email** = email;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", password='"** + **password** + **'\''** +  **", phone='"** + **phone** + **'\''** +  **", email='"** + **email** + **'\''** +  **'}'**;  } } |

## 测试效果





# 负载均衡

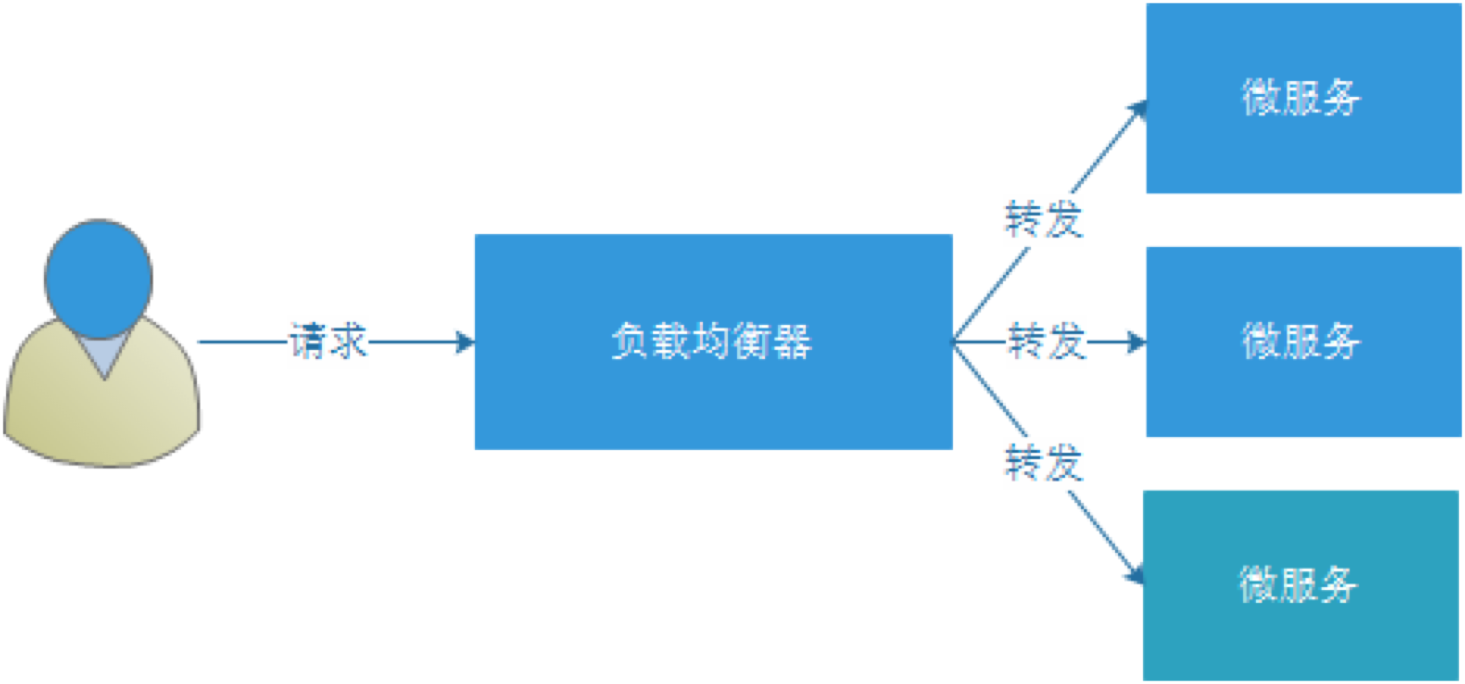
## 什么是负载均衡

负载均衡是在微服务架构中必须使用的一种技术，通过负载均衡实现高可用，扩容性。

负载均衡有两种方式实现：

硬件负载：F5、netscaler,A10等负载设备来实现负载

软件负载：nginx/lvs等



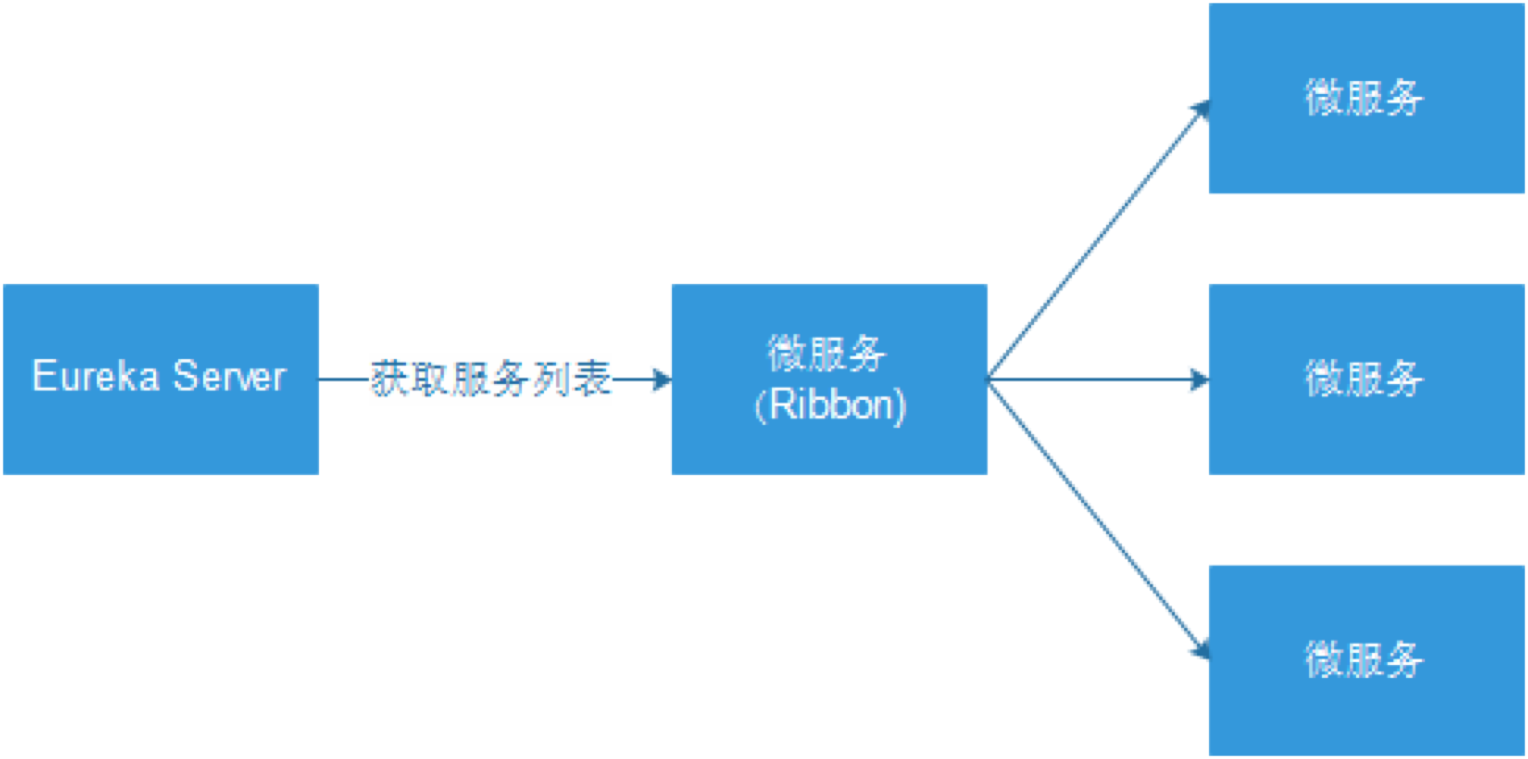
用户首先请求负载均衡器，负载均衡器会根据用户请求负载量按照一定的算法把请求分配到不同服务器。

负载均衡算法：轮转、随机、加权算法、加权随机等。

负载均衡器都会有一份服务器列表并维护服务列表，根据算法把用户请求分发到不同的业务上。

## Spring Cloud ribbon

Spring Cloud ribbon是基于客户端的负载均衡工具，负载均衡分为服务端负载和客户端负载，客户端的负载与服务端的负载的区别是客户端要维护一份服务列表。



Ribbon从服务端去获取服务列表，Ribbon根据负载均衡的算法直接请求到具体的微服务。中间省去了负载均衡服务。

执行流程：

1. 在消费微服务时，Ribbin先从Eureka server中获取服务列表
2. Ribbon根据负载均衡的算法直接请求到具体的微服务.

## 负载均衡Ribbon实践

开发单点登录系统

需求：

1. 单点登录系统服务提供用户登录功能
2. 单点登录系统要调用查询用户信息服务
3. 单点登录系统采用spring boot –springmvc开发。

## Ribbon编程步骤

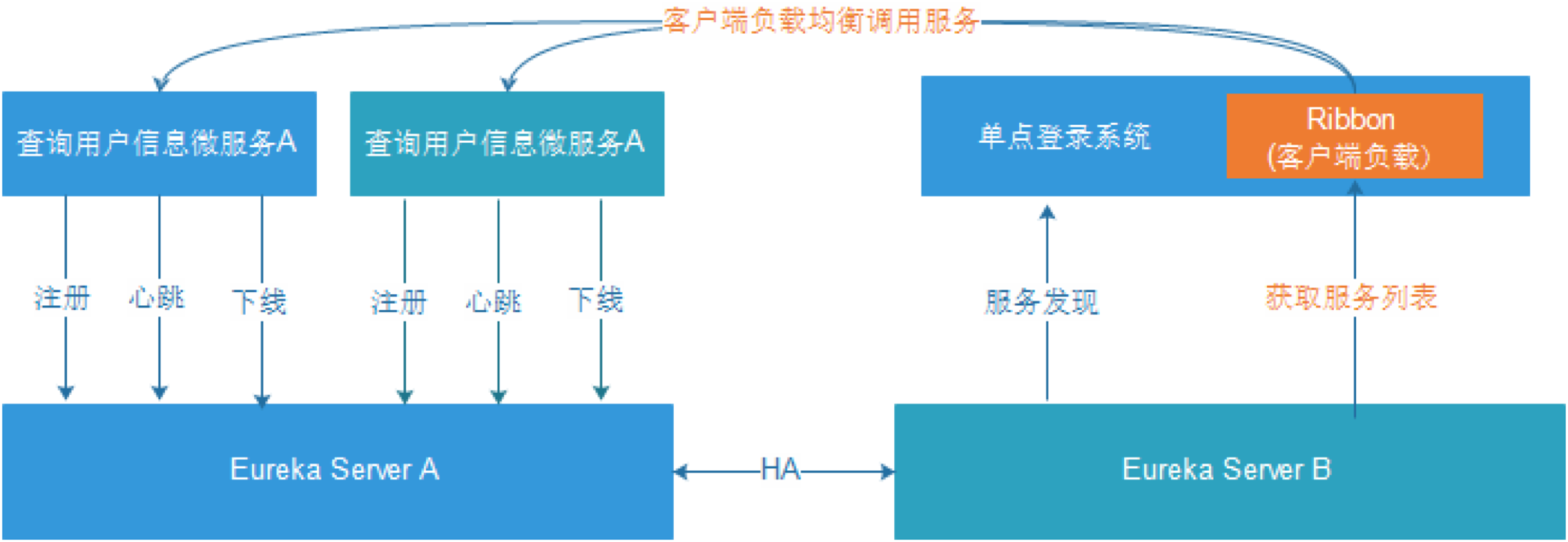
步骤：

1. 定义restTemplate对象
2. 配置@LoadBanlanced

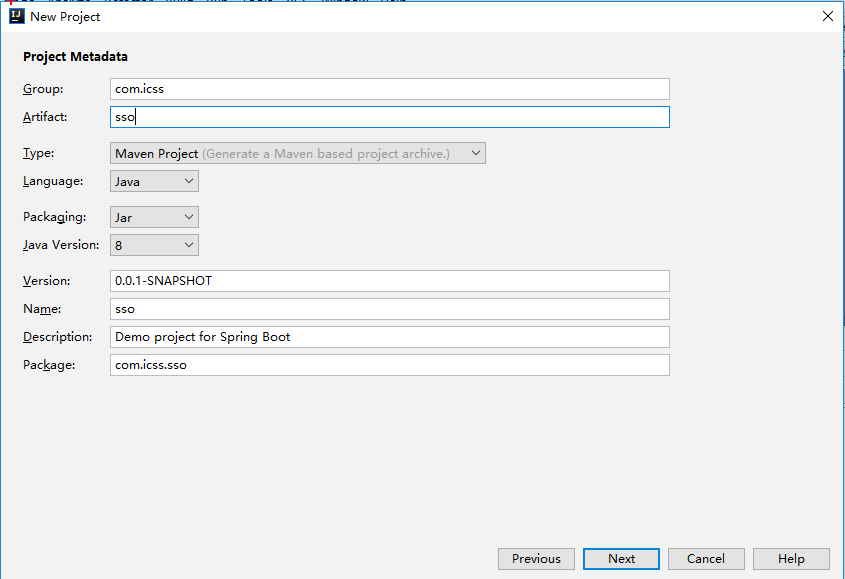
3. 设置负载均衡算法

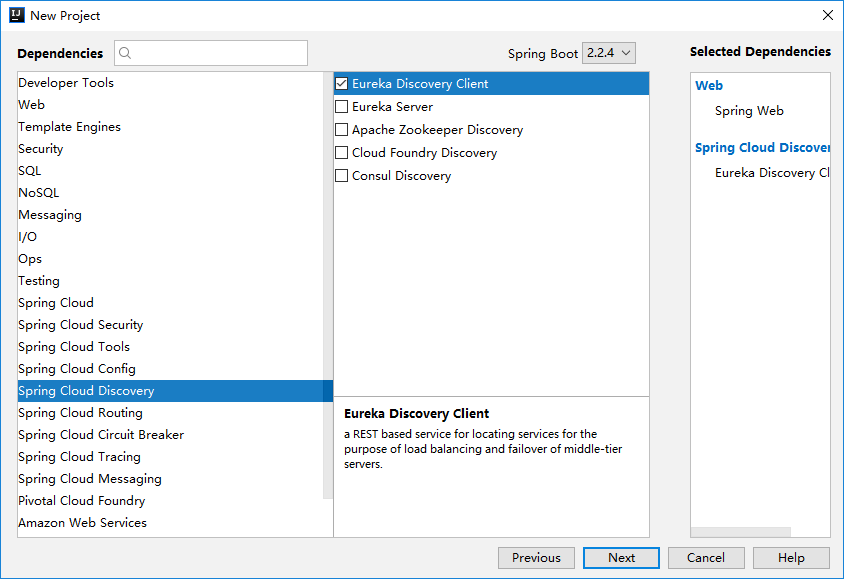
4． 使用restTemplate调用微服务

## Ribbon客户端负载调用流程



## 创建单点登录系统

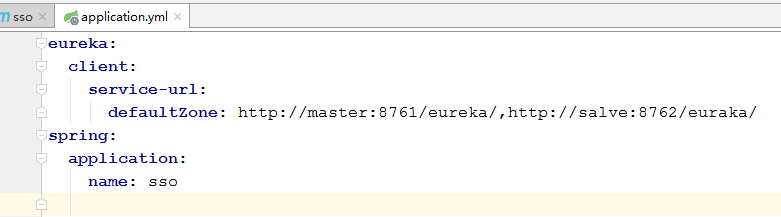




## 导入依赖

|  |
| --- |
| *<!--ribbon -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-feign</**artifactId**>  <**version**>1.4.7.RELEASE</**version**> </**dependency**>  *<!-- ok3Http-->* <**dependency**>  <**groupId**>com.squareup.okhttp3</**groupId**>  <**artifactId**>okhttp</**artifactId**>  <**version**>3.9.0</**version**> </**dependency**>  *<!--启动连接管理失败重试-->* <**dependency**>  <**groupId**>org.springframework.retry</**groupId**>  <**artifactId**>spring-retry</**artifactId**> </**dependency**> |

## 编写配置文件



## 编写启动类

|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient @EnableFeignClients *//启用ribbon客户端负载* @EnableHystrix **public class** SsoApplication {   @Bean *//向spring容器注入RestTemplate* @LoadBalanced *//取出服务列表，执行默认负载均衡算法* **public** RestTemplate restTemplate(){  **return new** RestTemplate(**new** OkHttp3ClientHttpRequestFactory());  }   **public static void** main(String[] args) {  SpringApplication.*run*(SsoApplication.**class**, args);  }  } |

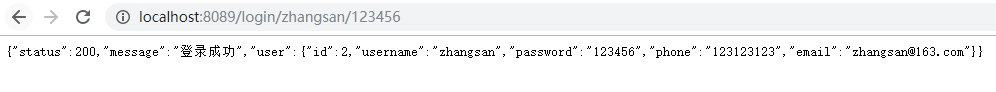
## 编写SsoController

|  |
| --- |
| @Controller @RefreshScope *//动态刷配置* **public class** SsoController {   @Autowired  **private** SsoService **ssoService**;    @RequestMapping(value = **"login/{username}/{password}"**,method = RequestMethod.***GET***)  @ResponseBody  **public** SsoResult login(@PathVariable(**"username"**) String username,  @PathVariable(**"password"**) String password){  User user = **this**.**ssoService**.checkUser(username, password);  **if** (user==**null** || user.getId()==**null**){  **return new** SsoResult(400,**"登录失败"**,user);  }**else**{  **return new** SsoResult(200,**"登录成功"**,user);  }  } } |

## 编写SsoService

|  |
| --- |
| @Service **public class** SsoService {   @Autowired  **private** RestTemplate **restTemplate**;   **public** User checkUser(String username,String password){  String serverId =**"USER"**;*//调用的服务名称* User user = **this**.**restTemplate**.getForObject(**"http://"** + serverId + **"/user/"** + username, User.**class**);  **if** (user!=**null**){  String password1 = user.getPassword();  **if** (password1.endsWith(password)){  **return** user;  }  }  **return null**;  } } |

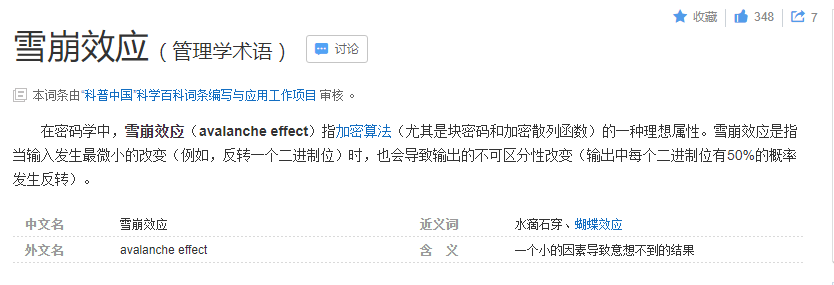
## 测试效果



# 容错保护

## 什么是容错保护

容错保护指的是微服务在执行的过程中出现错误从错误中恢复的能力，微服务如果容错保护不好，容易出现雪崩效应。



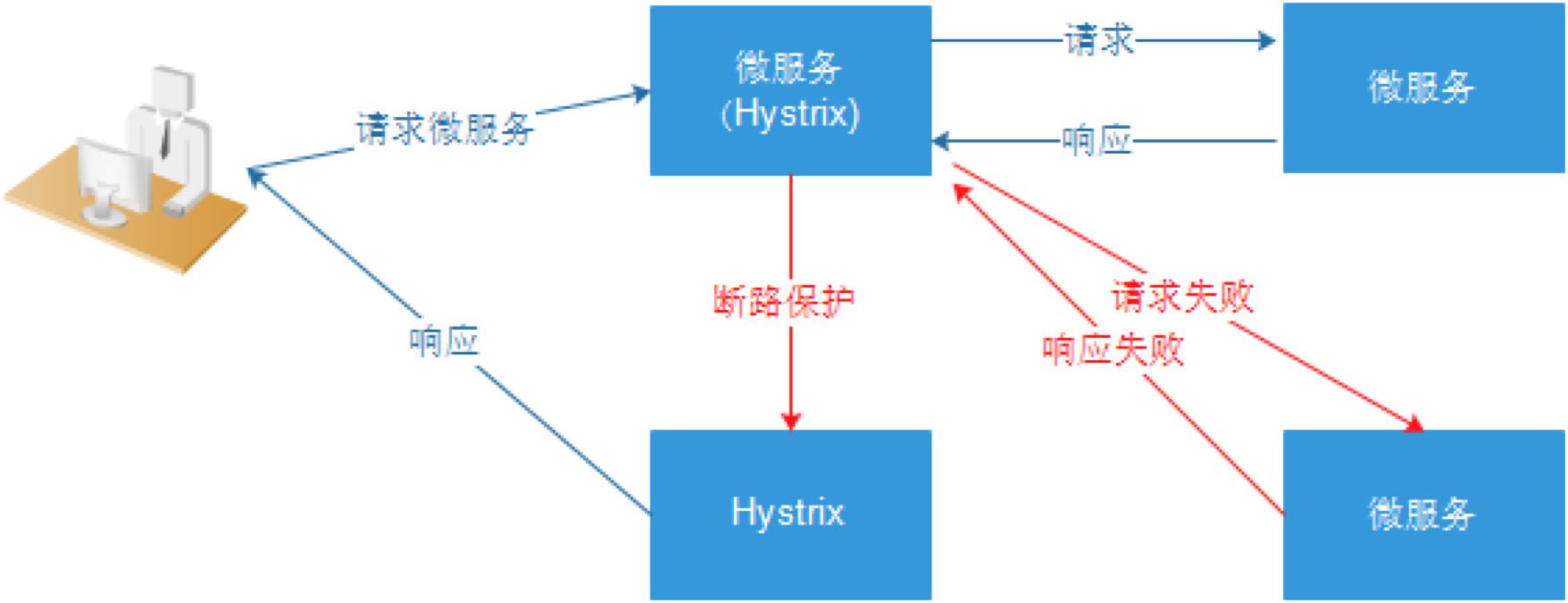
微服务中的雪崩效应指的是服务与服务之间的调用，当其中一个服务无法提供服务时，导致整个服务都死掉。

比如：用户单点登录系统会调用到查询用户信息微服务，当查询用户信息微服务不可用时，导致单点登录系统一直在等待，导致用户登录功能不可使用。像这样由一个服务不可用，导致整个系统不可用，这种就是雪崩效应。

## Spring Cloud Hystrix

Spring Cloud Hystrix是基于netflix开源框架开发的，它实现断路器、线程隔离、信号隔离等容错保护功能。

Spring Cloud Hystrix实现原理：



Spring Cloud Hystrix 使用注意：在使用方启用容错保护。（谁调用谁保护）

## Hystrix保护示例

### 导入容错保护依赖

|  |
| --- |
| *<!--容错保护依赖-->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**> </**dependency**> |

### 在启动类中开启容错保护

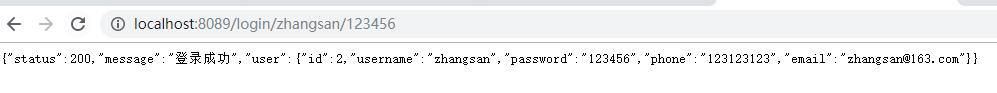
|  |
| --- |
| @SpringBootApplication @EnableDiscoveryClient @EnableFeignClients *//启用ribbon客户端负载* @EnableHystrix *//启用容错保护* **public class** SsoApplication {   @Bean *//向spring容器注入RestTemplate* @LoadBalanced *//取出服务列表，执行默认负载均衡算法* **public** RestTemplate restTemplate(){  **return new** RestTemplate(**new** OkHttp3ClientHttpRequestFactory());  }   **public static void** main(String[] args) {  SpringApplication.*run*(SsoApplication.**class**, args);  }  } |

### 使用容错保护

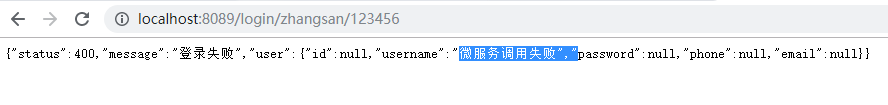
|  |
| --- |
| @Service **public class** SsoService {   @Autowired  **private** RestTemplate **restTemplate**;   @HystrixCommand(fallbackMethod = **"checkUserFeid"**)*//进行容错保护* **public** User checkUser(String username,String password){  String serverId =**"USER"**;*//调用的服务名称* User user = **this**.**restTemplate**.getForObject(**"http://"** + serverId + **"/user/"** + username, User.**class**);  **if** (user!=**null**){  String password1 = user.getPassword();  **if** (password1.endsWith(password)){  **return** user;  }  }  **return null**;  }   **public** User checkUserFeid(String username,String password){  **return new** User(**null**,**"微服务调用失败"**,**null**,**null**,**null**);  } } |

### 测试效果

正常时：



查询用户信息的微服务死机：



# 服务网关

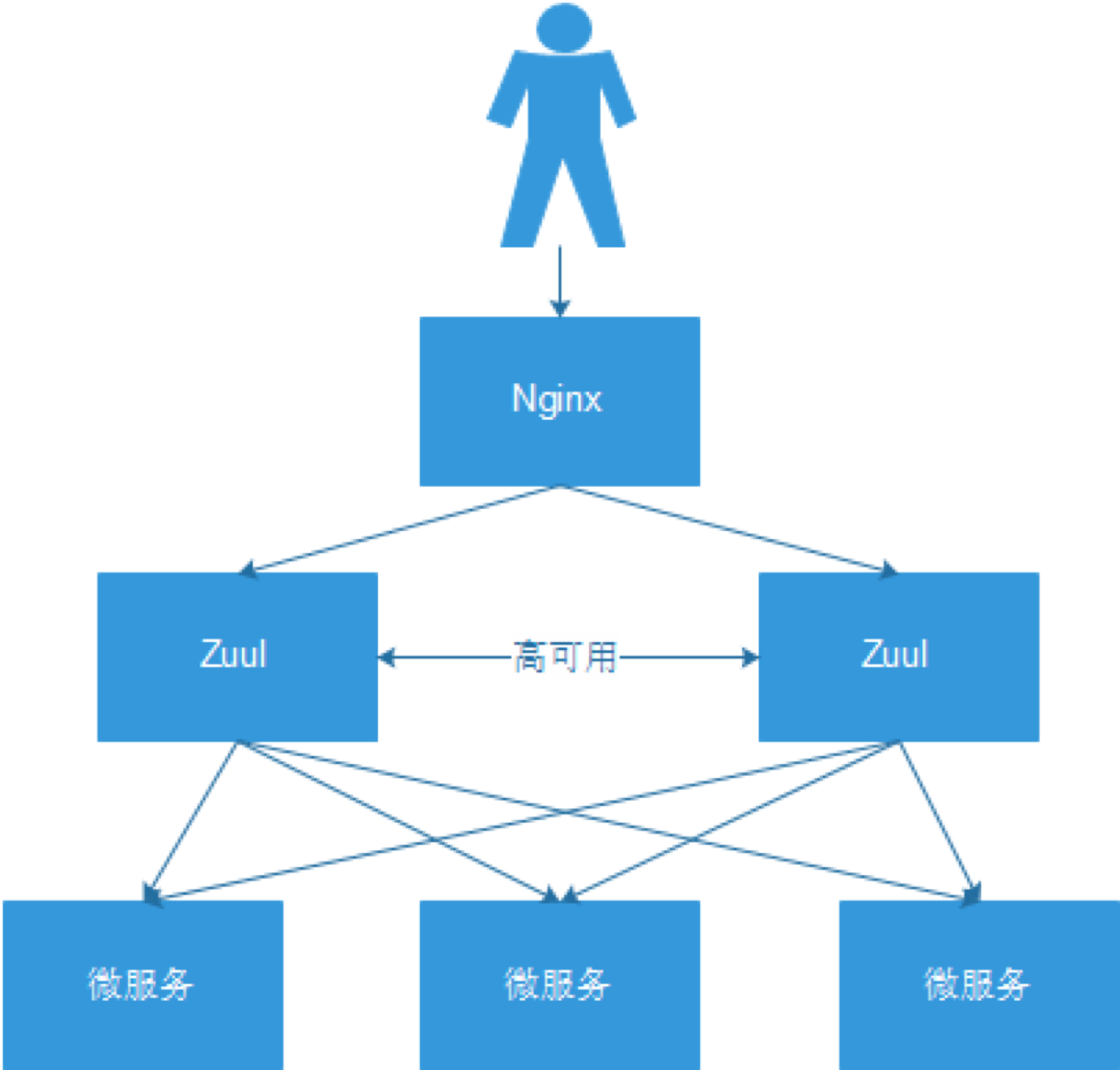
## 什么是服务网关

服务网关是在微服务前边设置一道屏障，请求先通过网关，网关会对请求进行过滤、校验、路由等处理。

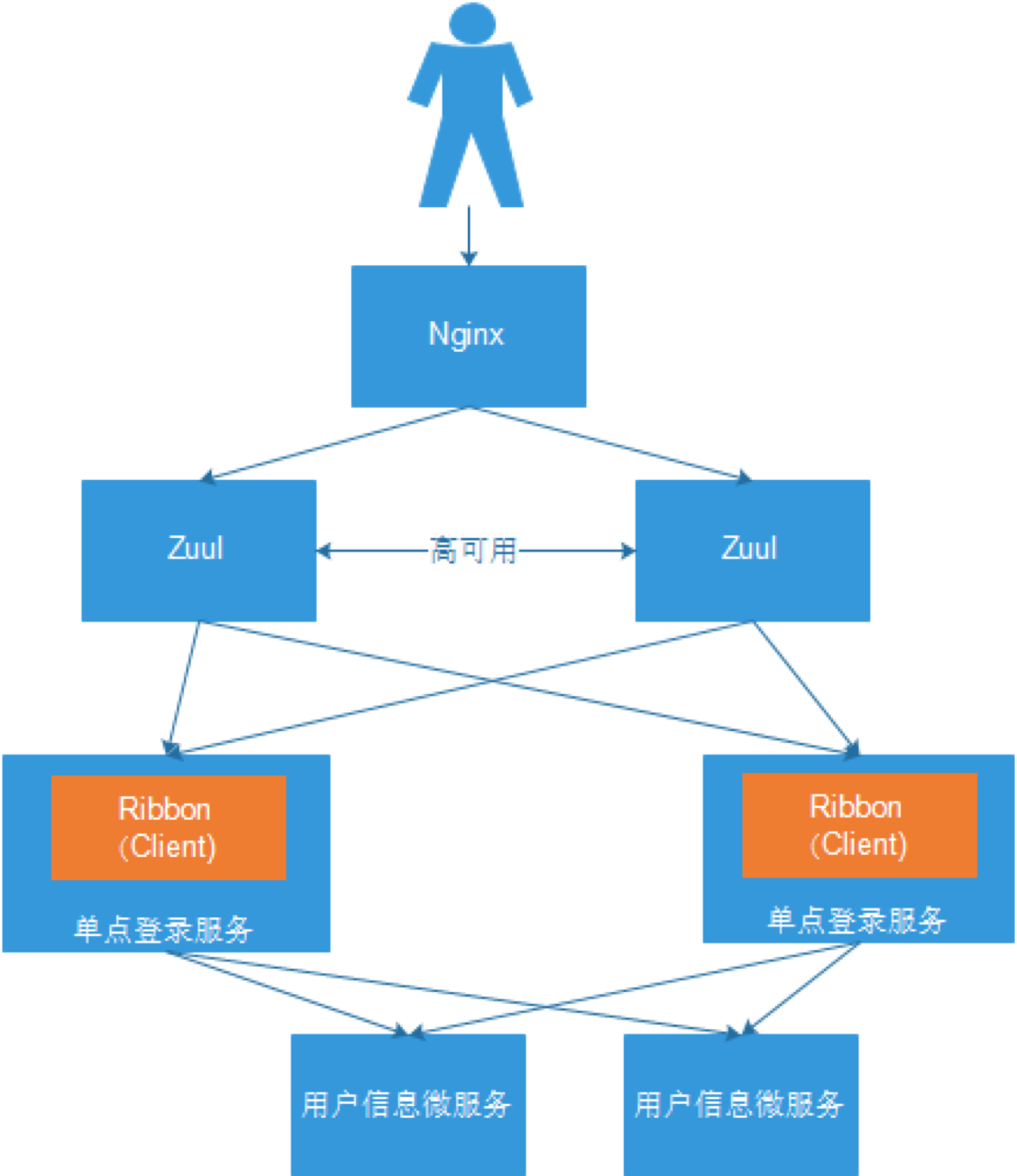
有了服务网关可以提高微服务的安全性。

## Spring cloud Zuul

Spring cloud Zuul是基于netflix公司zuul开源项目实现的，它实现了请求路由、负载均衡、校验、过滤等功能。



## Spring Cloud Zuul实践



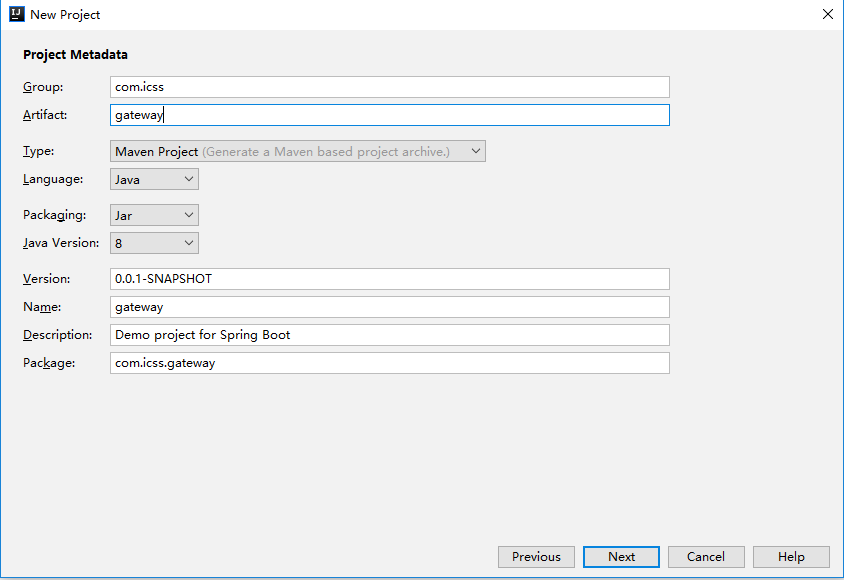
## 实现步骤

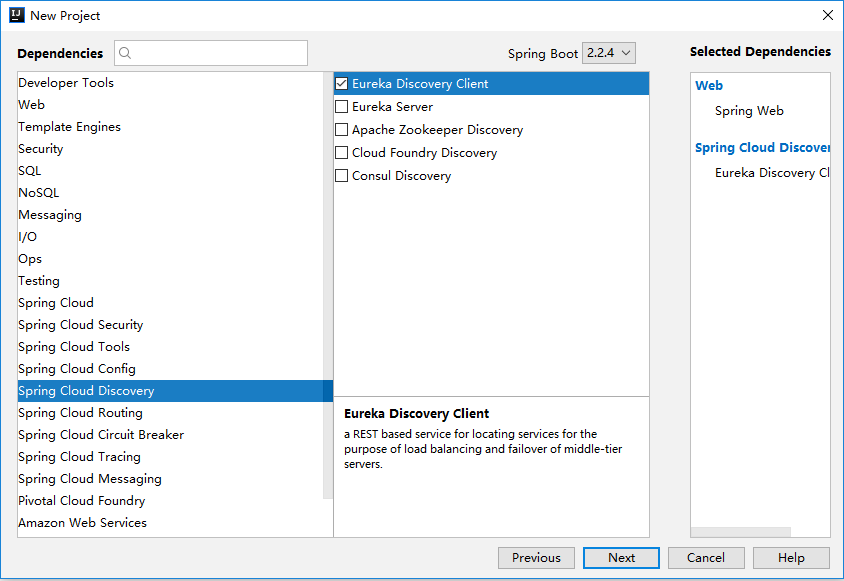
1. 已经部署查询用户信息微服务和单点登录系统服务
2. 将查询用户信息微服务和单点登录系统服务注册到Eureka Server中
3. 开发zuul模块
   1. 导入zuul依赖

/sso/\*

/xxx/\*

## 创建zuul路由工程





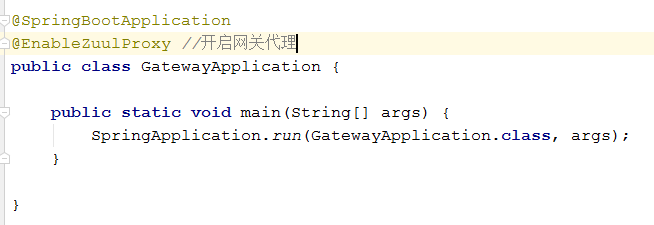
## 导入zuul依赖

|  |
| --- |
| *<!-- zuul 服务网关依赖 -->* <**dependency**>  <**groupId**>org.springframework.cloud</**groupId**>  <**artifactId**>spring-cloud-starter-zuul</**artifactId**>  <**version**>1.4.7.RELEASE</**version**> </**dependency**> |

## Application.yml



## 开启网关代理

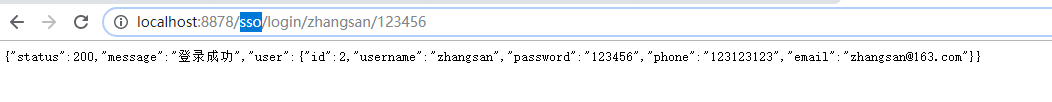


## 定义过滤器UserLoginZuulFilter

|  |
| --- |
| @Component **public class** UserLoginZuulFilter **extends** ZuulFilter {  @Override  **public** String filterType() {  **return "pre"**;*//设置过滤类型* }   @Override  **public int** filterOrder() {  **return** 0;*//设置执行顺序* }   @Override  **public boolean** shouldFilter() {  **return true**;*//设置当前过滤是否要执行* }   @Override *//编写业务* **public** Object run() **throws** ZuulException {  *//做各种业务处理。。。。* **return null**;  } } |

## 测试访问请求

凡是通过/sso/\*进来的业务都可以通过网关，其它的请求没法通过。



# 了解spring cloud子项目

## Spring cloud config 统一配置管理中心

Spring cloud config 统一配置管理中心是spring cloud解决管理配置文件管理。因为微服务很多，并且有很多的配置文件，这些文件要进行统一管理方便各个集群使用。

## Spring cloud bus 消息总线

主要是用于微服务与微服务之间的消息同步，可以解耦合实现消息同步。

## Spring cloud sleuth 链路跟踪

多个微服务之间的调用，调用节点很多，很难跟踪到哪个节点出错，使用Spring cloud sleuth 链路跟踪方便找出问题所在。

# 课程总结

