**SpringCloud**

**Day2**

**课程计划**

1. 能够使用Feign [feɪn]进行远程调用

更优雅的远程调用、集成了Ribbon的负载均衡、集成了Hystrix的熔断器等功能

1. 能够搭建Spring Cloud Gateway [geɪtweɪ]

网关、路由、过滤

1. 能够配置Spring Cloud Gateway过滤器
2. 能够使用Spring Cloud Gateway默认过滤器：全局过滤、局部过滤
3. 能够搭建Spring Cloud Config配置中心服务
4. 能够使用Spring Cloud Bus 消息总线实时更新配置文件

# SpringCloudFeign

## 目标

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

## Feign简介

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

项目主页：<https://github.com/OpenFeign/feign>

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

## 快速入门

使用Feign替代RestTemplate发送Rest请求。

实现步骤：

1. 导入feign依赖

2. 编写Feign客户端接口

3. 消费者启动引导类开启Feign功能注解

4. 访问接口测试

### 导入依赖

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

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

### 创建Feign客户端

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

*/\*\*  
 \* Feign客户端入门  
 \** ***@author*** *Steven  
 \** ***@description*** *com.itheima.feign  
 \*/*@FeignClient(value = **"user-provider"**) *//声明Feign的客户端，value:填写生产者的名称***public interface** UserClient {  
  
 *//注意：此处的请求地址一定要和生产者请求的地址一样* @RequestMapping(**"/user/find/{id}"**)  
 **public** User findById(@PathVariable(**"id"**)Integer id);  
}

解释：

Feign会通过动态代理，帮我们生成实现类。

注解@FeignClient声明Feign的客户端，注解value指明服务名称

接口定义的方法，采用SpringMVC的注解。Feign会根据注解帮我们生成URL地址

注解@RequestMapping中的/user，不要忘记同时建议不要放在类的头部。因为Feign需要拼接可访问地址

### 编写控制层

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

@RestController  
@RequestMapping(**"feign"**)  
**public class** ConsumerFeignController {  
 @Autowired  
 **private** UserClient **userClient**;  
  
 @RequestMapping(**"{id}"**)  
 **public** User findById(@PathVariable(value = **"id"**)Integer id){  
 **return userClient**.findByUserId(id);  
 }  
}

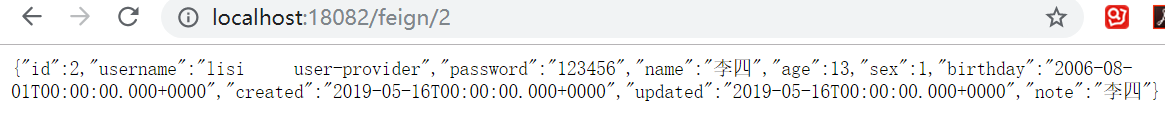
### 开启Feign

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

@SpringBootApplication  
@EnableDiscoveryClient *//开启eureka发现功能*@EnableCircuitBreaker *//开启熔断器*@EnableFeignClients *//开启Feign客户端***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();  
 }  
}

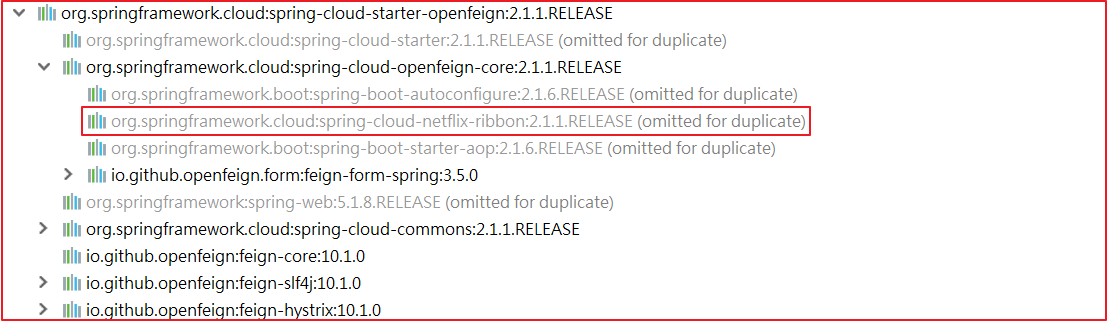
### 测试

启动相关工程，请求<http://localhost:18082/feign/2> 效果如下：



## 负载均衡

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



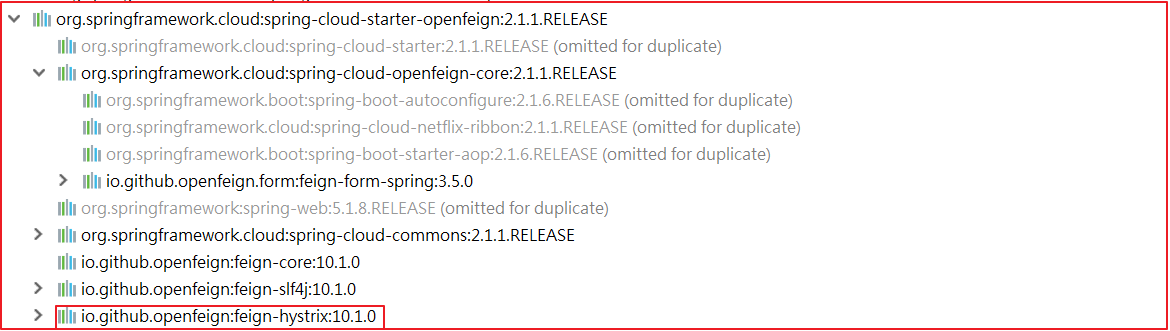
Feign内置的ribbon默认设置了请求超时时长，默认是1000，可以修改

ribbon内部有重试机制，一旦超时，会自动重新发起请求。如果不希望重试可以关闭配置：

*# 修改服务地址轮询策略，默认是轮询，配置之后变随机***user-provider**:  
 **ribbon**:  
 *#轮询* **NFLoadBalancerRuleClassName**: com.netflix.loadbalancer.RoundRobinRule  
 *#随机算法  
 #NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule* **ConnectTimeout**: 10000 *# 连接超时时间* **ReadTimeout**: 2000 *# 数据读取超时时间* **MaxAutoRetries**: 1 *# 最大重试次数(第一个服务)* **MaxAutoRetriesNextServer**: 0 *# 最大重试下一个服务次数(集群的情况才会用到)* **OkToRetryOnAllOperations**: false *# 无论是请求超时或者socket read timeout都进行重试，一般设置为false*

## 熔断器支持

Feign默认也有对Hystrix的集成!



实现步骤：

1. 在配置文件application.yml中开启feign熔断器支持

2. 编写FallBack处理类，实现FeignClient客户端

3. 在@FeignClient注解中，指定FallBack处理类。

4. 测试

### 开启Hystrix

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

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

### 熔断降级类创建

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

*/\*\*  
 \* Feign对hystrix熔断器的支持  
 \** ***@author*** *Steven  
 \** ***@description*** *com.itheima.feign.fallback  
 \*/*@Component  
**public class** UserClientFallback **implements** UserClient {  
 */\*\*  
 \* 服务降级实现  
 \** ***@param id*** *\** ***@return*** *\*/* @Override  
 **public** User findById(Integer id) {  
 User user = **new** User();  
 user.setUsername(**"Fallback，Feign服务降级。。。"**);  
 **return** user;  
 }  
}

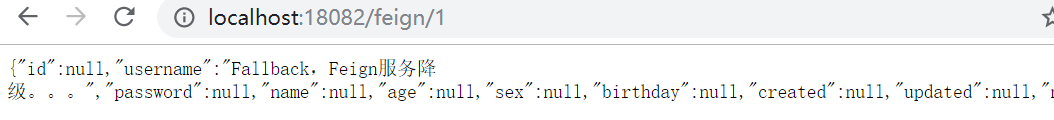
### 指定Fallback处理类

在@FeignClient注解中，指定FallBack处理类

*/\*\*  
 \* Feign客户端入门  
 \** ***@author*** *Steven  
 \** ***@description*** *com.itheima.feign  
 \*/*@FeignClient(value = **"user-provider"**,fallback = UserClientFallback.**class**) *//声明Feign的客户端，value:填写生产者的名称***public interface** UserClient {  
  
 *//注意：此处的请求地址一定要和生产者请求的地址一样* @RequestMapping(**"/user/find/{id}"**)  
 **public** User findById(@PathVariable(**"id"**)Integer id);  
}

### 测试

关闭服务生产者，请求<http://localhost:18082/feign/1> ，效果如下：



## 请求压缩

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

通过配置开启请求与响应的压缩功能：

**feign**:  
 **hystrix**:  
 **enabled**: **true** *# 开启Feign的熔断功能* **compression**:  
 **request**:  
 **enabled**: **true** *# 开启请求压缩* **response**:  
 **enabled**: **true** *# 开启响应压缩*

也可以对请求的数据类型，以及触发压缩的大小下限进行设置

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

## Feign的日志级别配置

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

实现步骤：

1. 在application.yml配置文件中开启日志级别配置

2. 编写配置类，定义日志级别bean。

3. 在接口的@FeignClient中指定配置类

4. 重启项目，测试访问

### 普通日志等级配置

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

*#配置普通日志级别***logging**:  
 **level**:  
 **com.itheima**: debug

### Feign日志等级配置

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

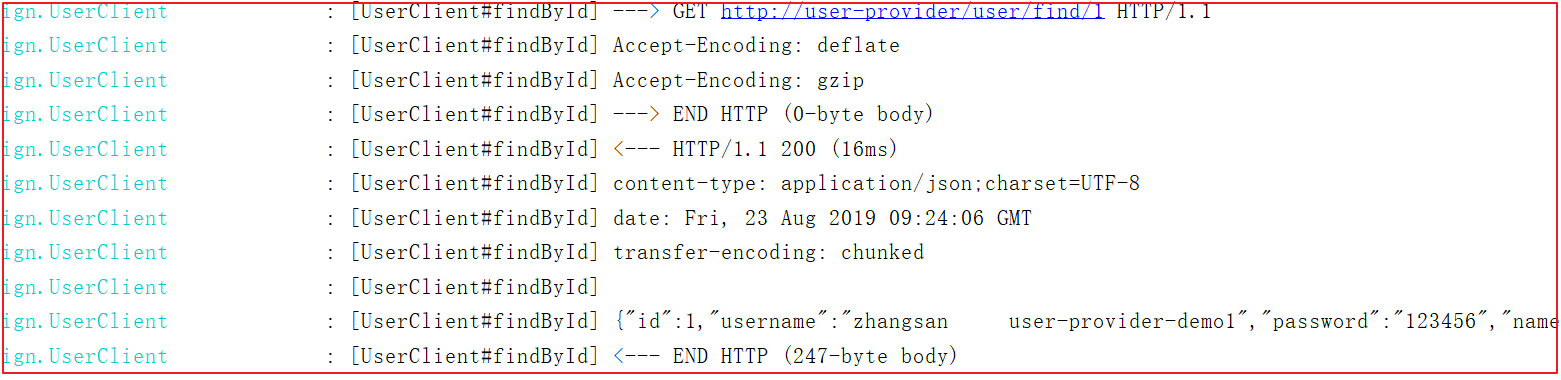
@Configuration  
**public class** FeignConfig {  
  
 */\*\*\*  
 \* 日志级别  
 \*/* @Bean  
 **public** Logger.Level feignLoggerLevel(){  
 */\*\*  
 \* NONE：不记录任何日志，默认值  
 \* BASIC：仅记录请求的方法，URL以及响应状态码和执行时间  
 \* HEADERS：在BASIC基础上，额外记录了请求和响应的头信息  
 \* FULL：记录所有请求和响应的明细，包括头信息、请求体、元数据  
 \*/* **return** Logger.Level.***FULL***;  
 }  
}

### 指定配置类

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

*/\*\*  
 \* Feign客户端入门  
 \** ***@author*** *Steven  
 \** ***@description*** *com.itheima.feign  
 \*/  
//声明Feign的客户端，value:填写生产者的名称*@FeignClient(value = **"user-provider"**,  
 fallback = UserClientFallback.**class**, *//配置熔断器* configuration = FeignConfig.**class** *//配置日志级别*)  
**public interface** UserClient {  
  
 *//注意：此处的请求地址一定要和生产者请求的地址一样* @RequestMapping(**"/user/find/{id}"**)  
 **public** User findById(@PathVariable(**"id"**)Integer id);  
}

### 重启项目，即可看到每次访问的日志



## 小结

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

properties

1.引入Feign依赖包

2.创建Feign接口,feign接口中需要指定调用的服务名字

3.使用@EnabledFeignClients启用Feign功能

* 掌握Feign的负载均衡配置

properties

在配置文件中配置

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

* 掌握Feign的熔断配置

properties

1.在application.yml中开启Hystrix

2.给Feign接口创建一个实现类

3.给Feign指定fallback类

* 掌握Feign的压缩配置

properties

在application.yml中指定压缩属性即可

* 掌握Feign的日志配置

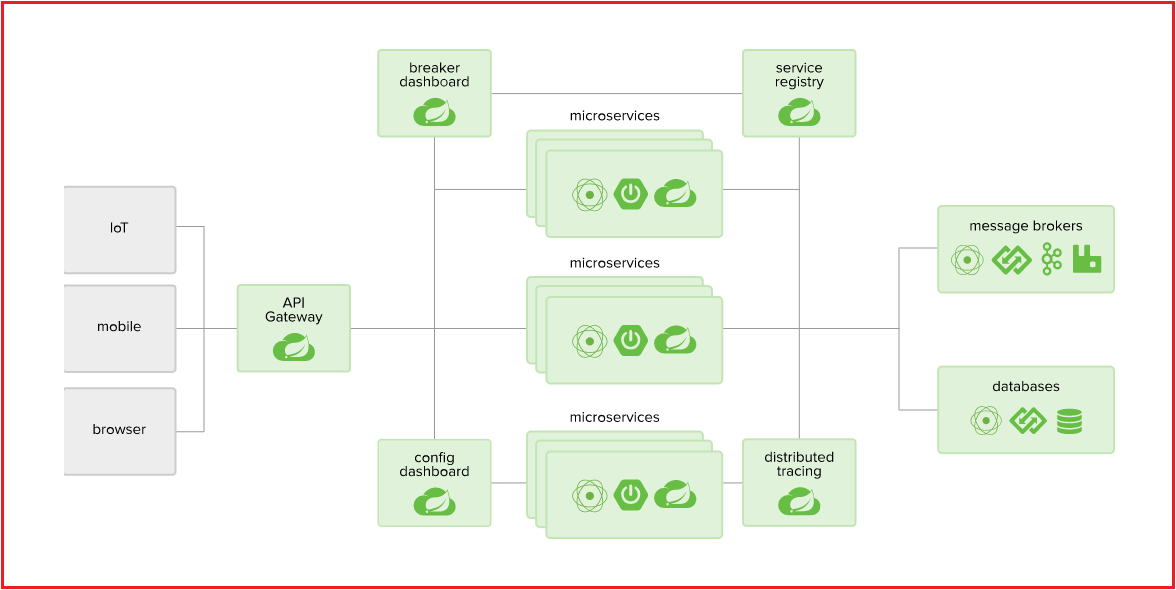
properties

1.在application.yml中开启普通日志等级

2.创建一个类，定义Feign日志等级

3.在Feign接口中指定定义日志的配置

# 网关 Spring Cloud Gateway



## 目标

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

## Gateway 简介

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

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

**本身也是一个微服务，需要注册到Eureka**。

**网关的核心功能**：过滤、路由

核心概念：通过画图解释

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

## 快速入门

实现步骤：

1. 创建gateway-service普通Maven工程

2. 编写基础配置

3. 编写路由规则，配置静态路由策略

4. 启动网关服务进行测试

### 创建一个子工程gateway\_service

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

### 创建启动引导类

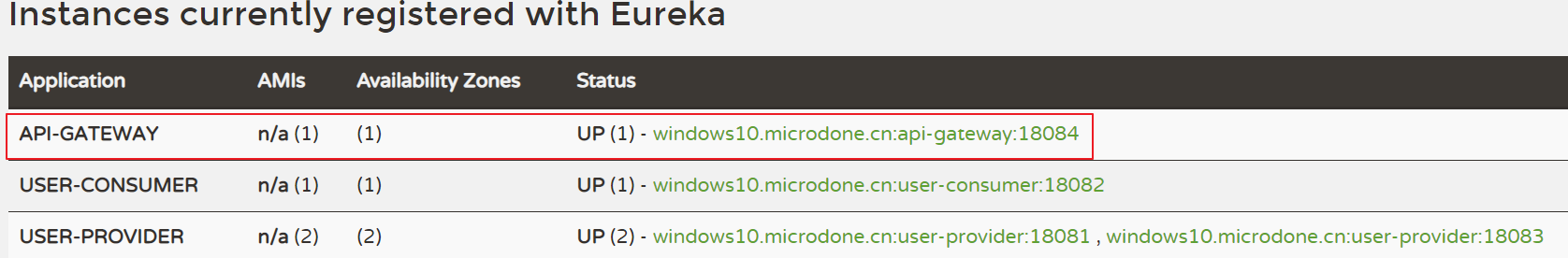
创建启动类com.itheima.GatewayApplication,代码如下：

@SpringBootApplication  
@EnableDiscoveryClient*// 开启Eureka客户端发现功能***public class** GatewayApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(GatewayApplication.**class**,args);  
 }  
}

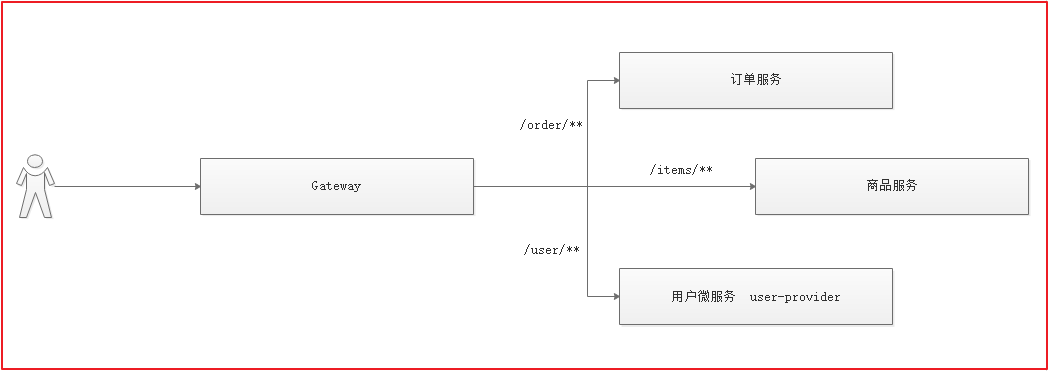
### application.yml配置

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

### 启动引导类，查看Eureka控制台



## 路由配置

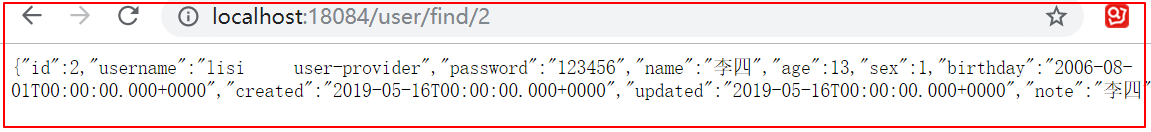


通过网关配置一个路由功能，用户访问网关的时候,如果用户请求的路径是以/user开始，则路由到user-provider服务去,修改application.yml配置即可实现，配置如下：

*# 注释版本***server**:  
 **port**: 18084  
**spring**:  
 **application**:  
 **name**: api-gateway *# 应用名* **cloud**:  
 **gateway**:  
 **routes**:  
 *#id唯一标识，可自定义* - **id**: user-service-route  
 *#路由的服务地址* **uri**: http://localhost:18081  
 *# 路由断言：路由拦截规则* **predicates**:  
 - Path=/user/\*\*  
*# Eureka服务中心配置***eureka**:  
 **client**:  
 **service-url**:  
 *# 注册Eureka Server集群* **defaultZone**: http://127.0.0.1:7001/eureka

启动GatewayApplication测试

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



## 动态路由



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

修改映射配置：通过服务名称获取，修改application.yml

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

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

## 过滤器

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

### 过滤器的分类

默认过滤器：出厂自带，实现好了拿来就用，不需要实现

全局默认过滤器

局部默认过滤器

自定义过滤器：根据需求自己实现，实现后需配置，然后才能用哦。

全局过滤器：作用在所有路由上。

局部过滤器：配置在具体路由下，只作用在当前路由上。

默认过滤器几十个，常见如下：

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

详细说明官方说明链接

<https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.1.1.RELEASE/single/spring-cloud-gateway.html#_gatewayfilter_factories>

### 默认过滤器配置

默认过滤器有两个：全局默认过滤器和局部默认过滤器。

#### 全局过滤器

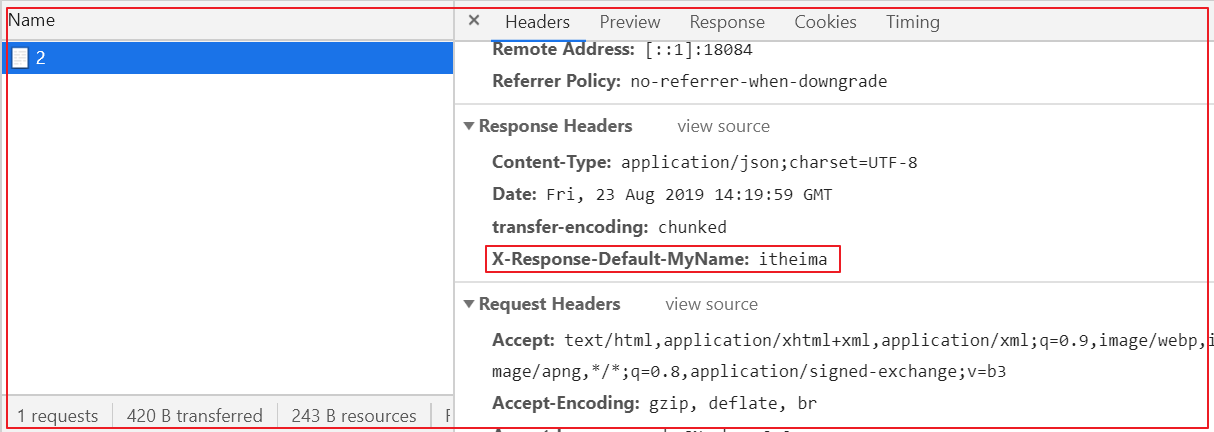
**全局过滤器：对输出响应头设置属性**

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

修改配置文件

**spring**:  
 **application**:  
 **name**: api-gateway *# 应用名* **cloud**:  
 **gateway**:  
 **routes**:  
 *#id唯一标识，可自定义* - **id**: user-service-route  
 *#路由的服务地址  
 #lb协议表示从Eureka注册中心获取服务请求地址  
 #user-provider访问的服务名称。  
 #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务* **uri**: lb://user-provider  
 *# 路由断言：路由拦截规则* **predicates**:  
 - Path=/user/\*\*  
 *#默认过滤器配置* **default-filters**:  
 *# 往响应过滤器中加入响应头信息* - AddResponseHeader=X-Response-Default-MyName,itheima

查看浏览器响应头信息!



#### 局部过滤器

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

##### 添加路径前缀

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

配置请求地址添加路径前缀过滤器

**spring**:  
 **application**:  
 **name**: api-gateway *# 应用名* **cloud**:  
 **gateway**:  
 **routes**:  
 *#id唯一标识，可自定义* - **id**: user-service-route  
 *#路由的服务地址  
 #lb协议表示从Eureka注册中心获取服务请求地址  
 #user-provider访问的服务名称。  
 #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务* **uri**: lb://user-provider  
 *# 路由断言：路由拦截规则* **predicates**:  
 *#- Path=/user/\*\*  
 #注意：要想接下来的前缀过滤器生效，此处拦截规则为所有url:/\*\** - Path=/\*\*  
 **filters**:  
 - PrefixPath=/user *# 给当前路由所有请求，添加"/user"前缀  
  
 #默认过滤器配置* **default-filters**:  
 *# 往响应过滤器中加入响应头信息* - AddResponseHeader=X-Response-Default-MyName,itheima

路由地址信息：

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

浏览器访问：<http://localhost:18084/find/2> 发现也可以查到用户信息

##### 去除路径前缀

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

配置去除路径前缀过滤器

**spring**:  
 **application**:  
 **name**: api-gateway *# 应用名* **cloud**:  
 **gateway**:  
 **routes**:  
 *#id唯一标识，可自定义* - **id**: user-service-route  
 *#路由的服务地址  
 #lb协议表示从Eureka注册中心获取服务请求地址  
 #user-provider访问的服务名称。  
 #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务* **uri**: lb://user-provider  
 *# 路由断言：路由拦截规则* **predicates**:  
 *#- Path=/user/\*\*  
 #注意：要想接下来的前缀过滤器生效，此处拦截规则为所有url:/\*\** - Path=/\*\*  
 **filters**: *#注意：PrefixPath与StripPrefix过滤器不要一起使用，一起使用等于没配置过滤器  
 #- PrefixPath=/user # 给当前路由所有请求，添加"/user"前缀* - StripPrefix=1 *# 去除路径前缀过滤器，一个"/"识别为一个路径  
 #默认过滤器配置* **default-filters**:  
 *# 往响应过滤器中加入响应头信息* - AddResponseHeader=X-Response-Default-MyName,itheima

浏览器访问：<http://localhost:18084/api/find/2> 发现也可以查到用户信息

路由地址信息：

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

### 自定义过滤器案例

自定义过滤器也有两个：全局**自定义全局过滤器**和**自定义局部过滤器**。

自定义全局过滤器的案例：模拟登陆校验。

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

#### 自定义全局过滤器

实现步骤：

1.在gateway-service工程编写全局过滤器类实现GlobalFilter,Ordered

2.编写业务逻辑代码

3.访问接口测试，加token和不加token。

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

*/\*\*  
 \* 自定义全局过滤器  
 \** ***@author*** *Steven  
 \** ***@description*** *com.itheima.filter  
 \*/*@Component  
**public class** LoginGlobalFilter **implements** GlobalFilter, Ordered {  
 */\*\*\*  
 \* 过滤拦截  
 \** ***@param exchange*** *\** ***@param chain*** *\** ***@return*** *\*/* @Override  
 **public** Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 *//获取请求参数token* String token = exchange.getRequest().getQueryParams().getFirst(**"token"**);  
 *//如果没有传入token* **if**(StringUtils.*isEmpty*(token)){  
 *//设置返回状态码-400参数异常* exchange.getResponse().setStatusCode(HttpStatus.***BAD\_REQUEST***);  
 *//结束请求* **return** exchange.getResponse().setComplete();  
 }  
 *//传入了token，放行* **return** chain.filter(exchange);  
 }  
  
 */\*\*\*  
 \* 定义过滤器执行顺序  
 \* 返回值越小，越靠前执行  
 \** ***@return*** *\*/* @Override  
 **public int** getOrder() {  
 **return** 0;  
 }  
}

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



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

#### 自定义局部过滤器(了解)

自定义局部过滤器，该过滤器在控制台输出配置文件中指定名称的请求参数及参数的值

实现步骤：

1. 在gateway-service中编写MyParamGatewayFilterFactory类

2. 实现业务代码：循环请求参数中是否包含name，如果包含则输出参数值

3. 修改配置文件

4. 访问请求测试，带name参数

在gateway\_service中编写MyParamGatewayFilterFactory类，此类不需要大家掌握，可以直接复制项目使用

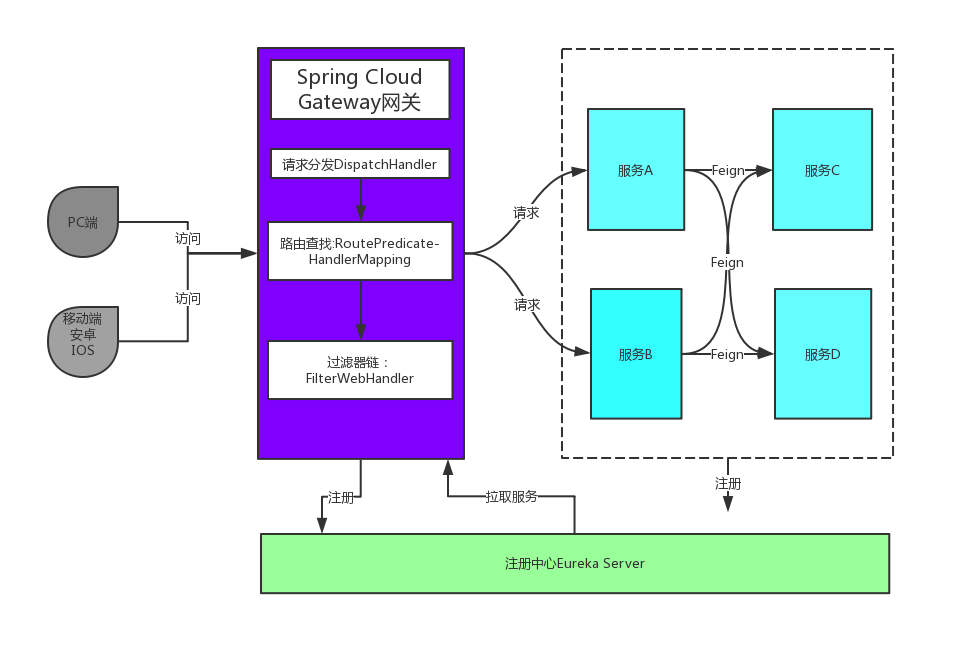
*/\*\*  
 \* 自定义局部过滤器  
 \* 注意的点：{  
 \* 类名必须是：前缀+GatewayFilterFactory  
 \* 必须有构造函数  
 \* 必须要有个内部类：Config  
 \* 处理当前排序方法：shortcutFieldOrder  
 \* 处理过滤器逻辑：apply  
 \* }  
 \*/*@Component  
**public class** MyParamGatewayFilterFactory **extends** AbstractGatewayFilterFactory<MyParamGatewayFilterFactory.Config> {  
 */\*\*  
 \* 定义需要处理的参数  
 \*/* **public static final** String ***PARAM\_NAME*** = **"name"**;  
  
 */\*\*\*\*  
 \* 处理过程  
 \** ***@return*** *\*/* @Override  
 **public** GatewayFilter apply(MyParamGatewayFilterFactory.Config config) {  
 **return new** GatewayFilter() {  
 @Override  
 **public** Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
 String name = exchange.getRequest().getQueryParams().getFirst(**"name"**);  
 **if**(!StringUtils.*isEmpty*(name)){  
 System.***out***.println(**"名字参数："**+name);  
 }  
 **return** chain.filter(exchange);  
 }  
 };  
 }  
  
 */\*\*\*  
 \* 构造函数  
 \*/* **public** MyParamGatewayFilterFactory() {  
 **super**(MyParamGatewayFilterFactory.Config.**class**);  
 }  
  
 */\*\*\*  
 \* 处理字段的排序  
 \** ***@return*** *\*/* @Override  
 **public** List<String> shortcutFieldOrder() {  
 **return** Arrays.*asList*(***PARAM\_NAME***);  
 }  
  
 */\*\*\*\*  
 \* 需要处理的参数  
 \* name：和处理的参数名字保持一致  
 \*/* **public static class** Config {  
 **private** String **name**;  
  
 **public** String getName() {  
 **return name**;  
 }  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
 }  
}

修改application.yml配置文件

**spring**:  
 **application**:  
 **name**: api-gateway *# 应用名* **cloud**:  
 **gateway**:  
 **routes**:  
 *#id唯一标识，可自定义* - **id**: user-service-route  
 *#路由的服务地址  
 #lb协议表示从Eureka注册中心获取服务请求地址  
 #user-provider访问的服务名称。  
 #路由地址如果通过lb协议加服务名称时，会自动使用负载均衡访问对应服务* **uri**: lb://user-provider  
 *# 路由断言：路由拦截规则* **predicates**:  
 *#- Path=/user/\*\*  
 #注意：要想接下来的前缀过滤器生效，此处拦截规则为所有url:/\*\** - Path=/\*\*  
 **filters**: *#注意：PrefixPath与StripPrefix过滤器不要一起使用，一起使用等于没配置过滤器  
 #- PrefixPath=/user # 给当前路由所有请求，添加"/user"前缀* - StripPrefix=1 *# 去除路径前缀过滤器，一个"/"识别为一个路径  
 #配置自定义局部过滤器，此处的名字与过滤器声明时的前缀一致* - MyParam=name  
 *#默认过滤器配置* **default-filters**:  
 *# 往响应过滤器中加入响应头信息* - AddResponseHeader=X-Response-Default-MyName,itheima

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

### 微服务架构加入Gateway后



不管是来自客户端的请求，还是服务内部调用。一切对服务的请求都可经过网关。

网关实现鉴权、动态路由等等操作。

Gateway就是我们服务的统一入口

## 小结

* 网关的作用

properties

1.为微服务架构提供一种简单有效统一的API路由管理方式

2.可以在网关中实现微服务鉴权、安全控制、请求监控、限流

* 会配置动态路由

properties

使用lb配置，能根据服务名字动态请求。

* 会配置过滤器

properties

默认过滤器：default-filters:

* 能自定义全局过滤器

properties

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

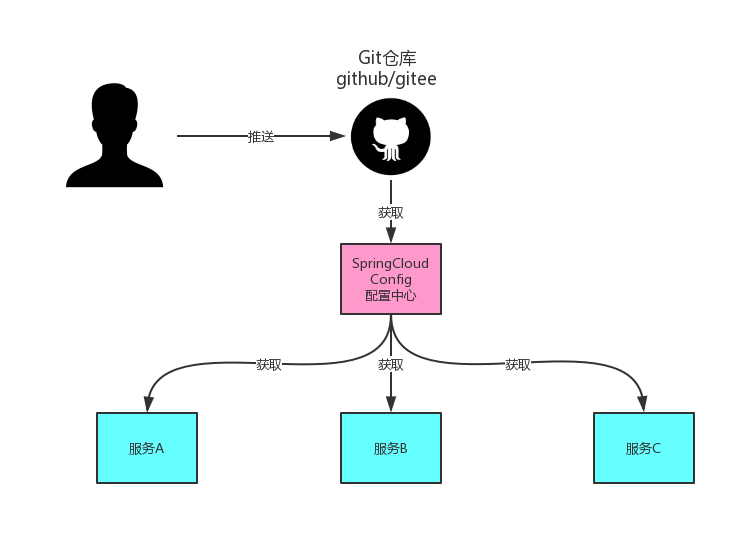
# 配置中心 Spring Cloud Config

## 目标

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

## Config简介

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



【配置中心的架构图】

## Git配置管理

### 远程Git仓库

知名的Git远程仓库有国外的GitHub和国内的码云(gitee)；

GitHub主服务在外网，访问经常不稳定，如果希望服务稳定，可以使用码云；

码云访问地址：<http://gitee.com>

测试地址：<https://gitee.com/StevenBlake/conf.git>

### 创建远程仓库

首先使用码云上的git仓库需要先注册账户。

账户注册完成，然后使用账户登录码云控制台并创建公开仓库

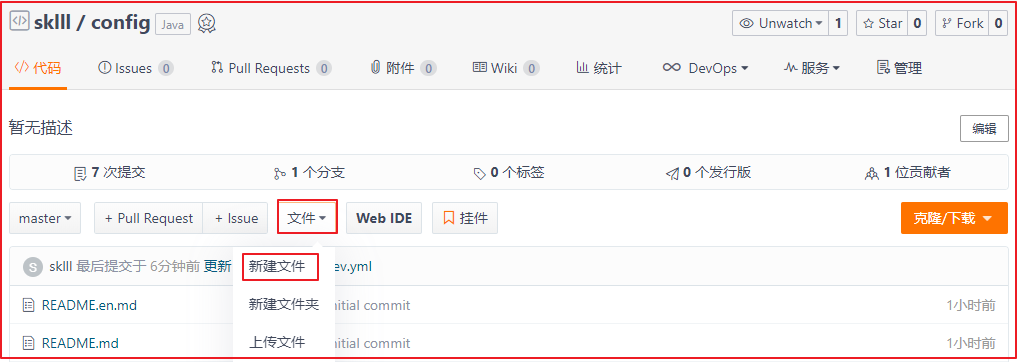


配置仓库 名称和路径



### 创建配置文件

在新建的仓库中创建需要被统一配置管理的配置文件



文件命名有规则：

配置文件的命名方式：{application}-{profile}.yml或{application}-{profile}.properties

application为应用名称

profile用于区分开发环境dev，测试环境test，生产环境pro等

开发环境 user-dev.yml

测试环境 user-test.yml

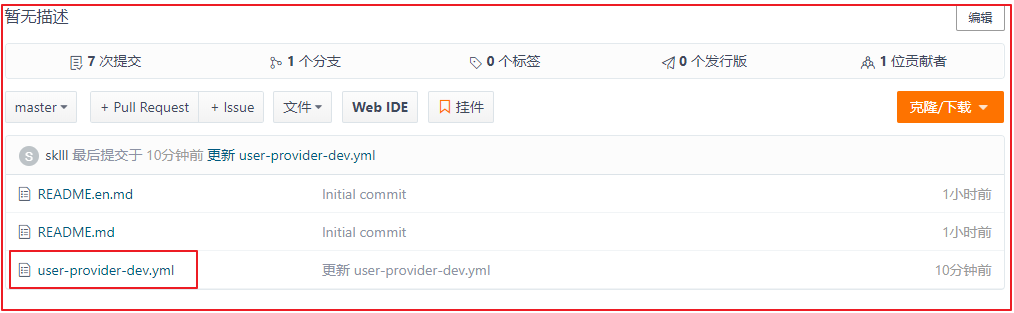
生产环境 user-pro.yml

创建一个user-provider-dev.yml文件

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



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



## 搭建配置中心微服务

实现步骤：

1. 创建配置中心SpringBoot项目config\_server

2. 配置坐标依赖

3. 启动类添加开启配置中心服务注解

4. 配置服务中心application.yml文件

5. 启动测试

### 创建工程 conf\_server

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

### 创建启动引导类

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

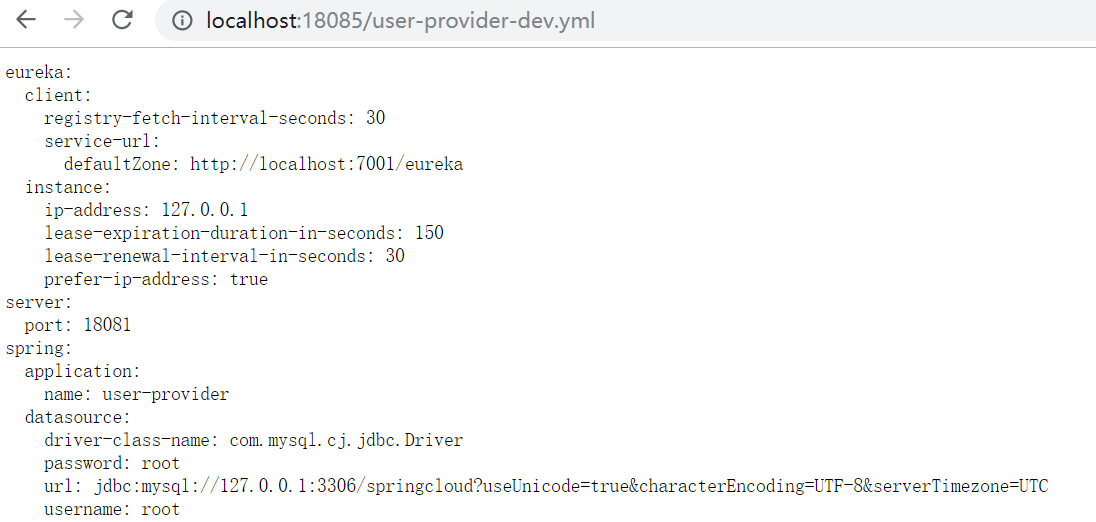
@SpringBootApplication  
@EnableDiscoveryClient*//开启Eureka客户端发现功能*@EnableConfigServer*//开启配置服务支持***public class** ConfigServerApplication {  
  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(ConfigServerApplication.**class**,args);  
 }  
}

### application.yml配置文件

*# 注释版本***server**:  
 **port**: 18085 *# 端口号***spring**:  
 **application**:  
 **name**: config-server *# 应用名* **cloud**:  
 **config**:  
 **server**:  
 **git**:  
 *# 配置gitee的仓库地址* **uri**: https://gitee.com/StevenBlake/conf.git  
*# Eureka服务中心配置***eureka**:  
 **client**:  
 **service-url**:  
 *# 注册Eureka Server集群* **defaultZone**: http://127.0.0.1:7001/eureka  
*# com.itheima 包下的日志级别都为Debug***logging**:  
 **level**:  
 **com**: debug

### 测试

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



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

## 服务去获取配置中心配置

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

实现步骤：

1. 添加配置中心客户端启动依赖

2. 修改服务提供者的配置文件

3. 启动服务

4. 测试效果

#### 添加依赖

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

#### 修改配置

删除user-provider工程的application.yml文件

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

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

关于application.yml和bootstrap.yml文件的说明：

- bootstrap.yml文件是SpringBoot的默认配置文件，而且其加载时间相比于application.yml更早。

- bootstrap.yml和application.yml都是默认配置文件，但定位不同

- bootstrap.yml可以理解成系统级别的一些参数配置，一般不会变动

- application.yml用来定义应用级别的参数

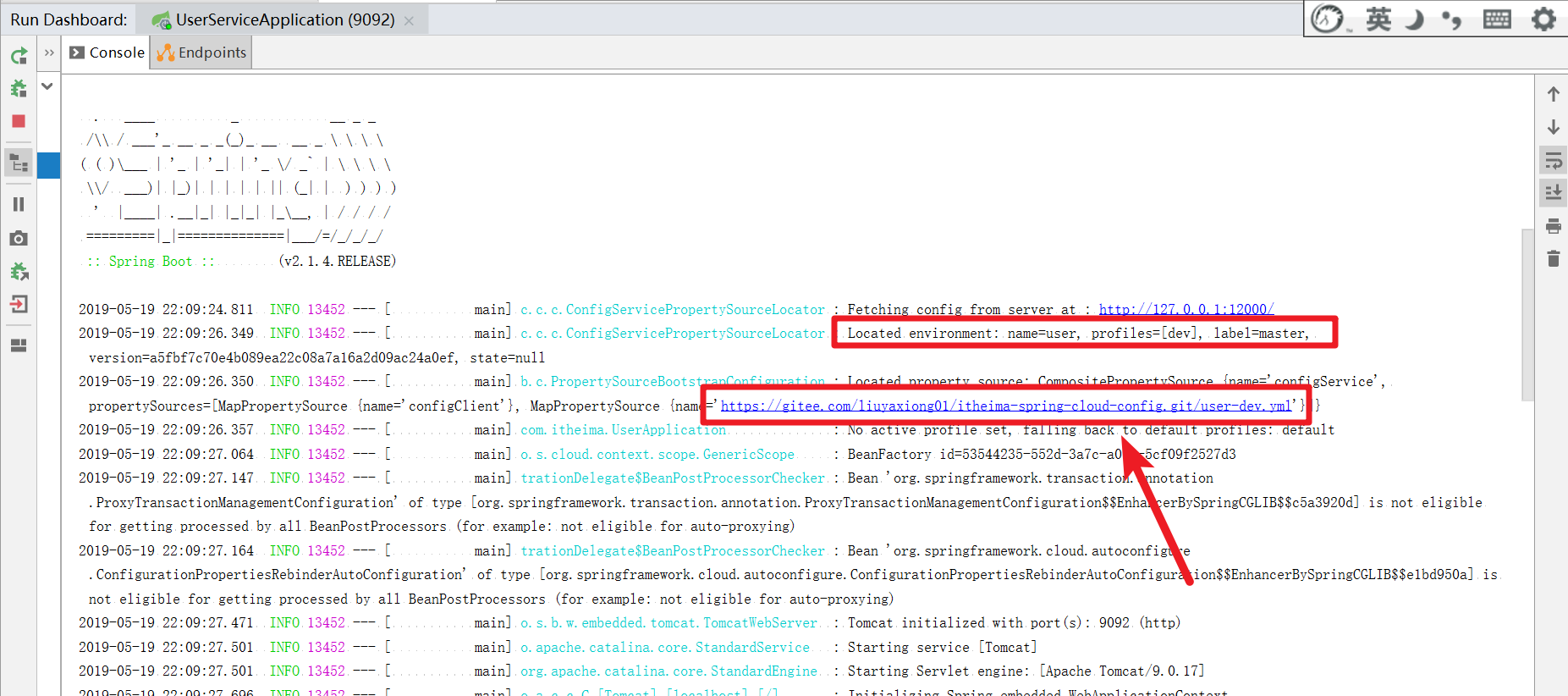
- 搭配spring-cloud-config使用application.yml的配置可以动态替换。

- bootstrap.yml相当于项目启动的引导文件，内容相对固定

- application.yml文件是微服务的常规配置参数，变化比较频繁

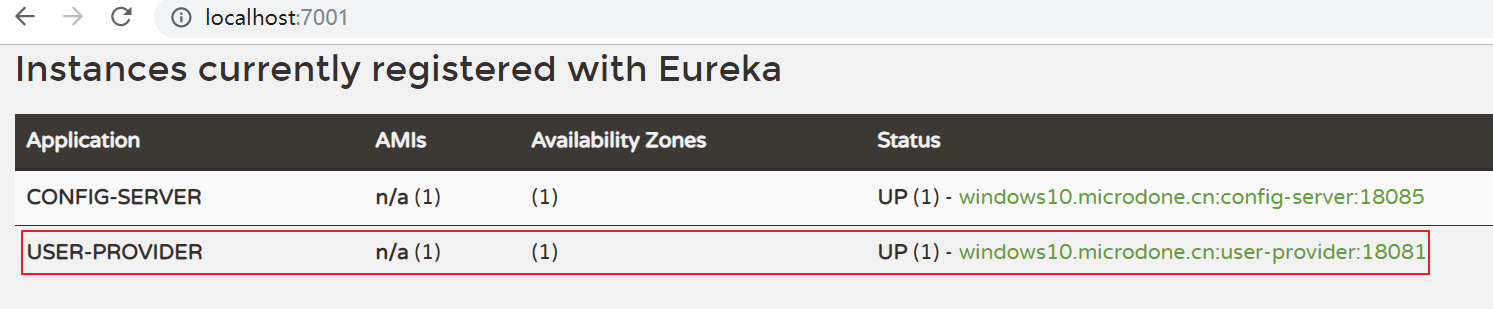
#### 启动测试

启动eurake服务中心、conf配置中心、user\_provider



如果启动没报错，其实已经使用上配置中心内容了

可以在服务中心查看也可以检验user\_service的服务



## 配置中心存在的问题

### 修改码云配置文件

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



### 读取配置文件数据

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

@RestController  
@RequestMapping(value = **"config"**)  
**public class** LoadConfigController {  
  
 @Value(**"${test.message}"**)  
 **private** String **msg**;  
  
 */\*\*\*  
 \* 读取配置文件中的数据  
 \** ***@return*** *\*/* @RequestMapping(value = **"load"**)  
 **public** String load(){  
 **return msg**;  
 }  
}

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



修改码云上的配置后，发现项目中的数据仍然没有变化,只有项目重启后才会变化。

## 小结

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

需要在application.yml配置文件中指定需要远程更新的仓库地址。

* 修改微服务，从配置中心获取修改的配置

properties

创建bootstrap.yml，并在bootstrap.yml中配置

# 注释版本

spring:

cloud:

config:

name: user-provider # 与远程仓库中的配置文件的application保持一致，{application}-{profile}.yml

profile: dev # 远程仓库中的配置文件的profile保持一致

label: master # 远程仓库中的版本保持一致

discovery:

enabled: true # 使用配置中心

service-id: config-server # 配置中心服务id

#向Eureka服务中心集群注册服务

eureka:

client:

service-url:

defaultZone: http://127.0.0.1:7001/eureka

# 消息总线 Spring Cloud Bus

SpringCloud Bus，解决上述问题，实现配置自动更新。

注意：SpringCloudBus基于RabbitMQ实现，默认使用本地的消息队列服务，所以需要提前安装并启动RabbitMQ。安装参考./04资料/安装Windows RabbitMQ.pdf

## Bus简介

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

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

广播出去的配置文件服务会进行本地缓存。

## 整合案例

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

### 改造配置中心

改造步骤：

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

#### 引入依赖

修改config-server的pom.xml引入依赖：

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

#### 修改application.yml配置文件

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

*# 注释版本***server**:  
 **port**: 18085 *# 端口号***spring**:  
 **application**:  
 **name**: config-server *# 应用名* **cloud**:  
 **config**:  
 **server**:  
 **git**:  
 *# 配置gitee的仓库地址* **uri**: https://gitee.com/StevenBlake/conf.git  
 *# rabbitmq的配置信息；如下配置的rabbit都是默认值，其实可以完全不配置* **rabbitmq**:  
 **host**: localhost  
 **port**: 5672  
 **username**: guest  
 **password**: guest  
*# 暴露触发消息总线的地址***management**:  
 **endpoints**:  
 **web**:  
 **exposure**:  
 *# 暴露触发消息总线的地址* **include**: bus-refresh  
*# Eureka服务中心配置***eureka**:  
 **client**:  
 **service-url**:  
 *# 注册Eureka Server集群* **defaultZone**: http://127.0.0.1:7001/eureka  
*# com.itheima 包下的日志级别都为Debug***logging**:  
 **level**:  
 **com**: debug

### 改造用户服务

改造步骤：

1. 在用户微服务user\_provider项目中加入Bus相关依赖

2. 修改user\_provider项目的bootstrap.yml，加入RabbitMQ的配置信息

3. UserController类上加入@RefreshScope刷新配置注解

4. 测试

#### 引入依赖

修改user-provider引入如下依赖：

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

#### 添加bootstrap.yml文件

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

*# 注释版本***spring**:  
 **cloud**:  
 **config**:  
 **name**: user-provider *# 与远程仓库中的配置文件的application保持一致，{application}-{profile}.yml* **profile**: dev *# 远程仓库中的配置文件的profile保持一致* **label**: master *# 远程仓库中的版本保持一致* **discovery**:  
 **enabled**: **true** *# 使用配置中心* **service-id**: config-server *# 配置中心服务id  
 # rabbitmq的配置信息；如下配置的rabbit都是默认值，其实可以完全不配置* **rabbitmq**:  
 **host**: localhost  
 **port**: 5672  
 **username**: guest  
 **password**: guest  
*#向Eureka服务中心集群注册服务***eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://127.0.0.1:7001/eureka

#### 添加刷新配置

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

@RestController  
@RequestMapping(value = **"config"**)  
@RefreshScope *//用于启用刷新配置文件的信息***public class** LoadConfigController {  
  
 @Value(**"${test.message}"**)  
 **private** String **msg**;  
  
 */\*\*\*  
 \* 读取配置文件中的数据  
 \** ***@return*** *\*/* @RequestMapping(value = **"load"**)  
 **public** String load(){  
 **return msg**;  
 }  
}

### 测试

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

测试步骤：

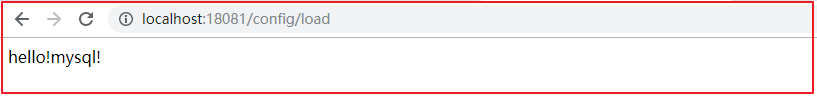
(1)启动eureka-server

(2)启动config-server

(3)启动user-provider

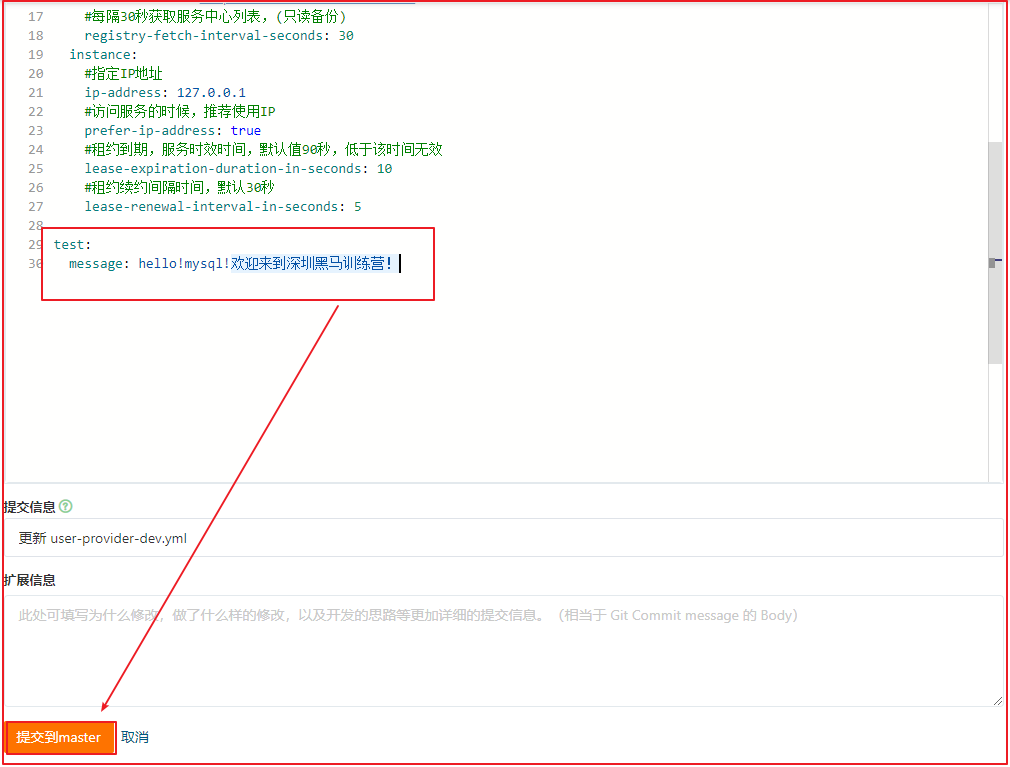
(4)访问测试

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



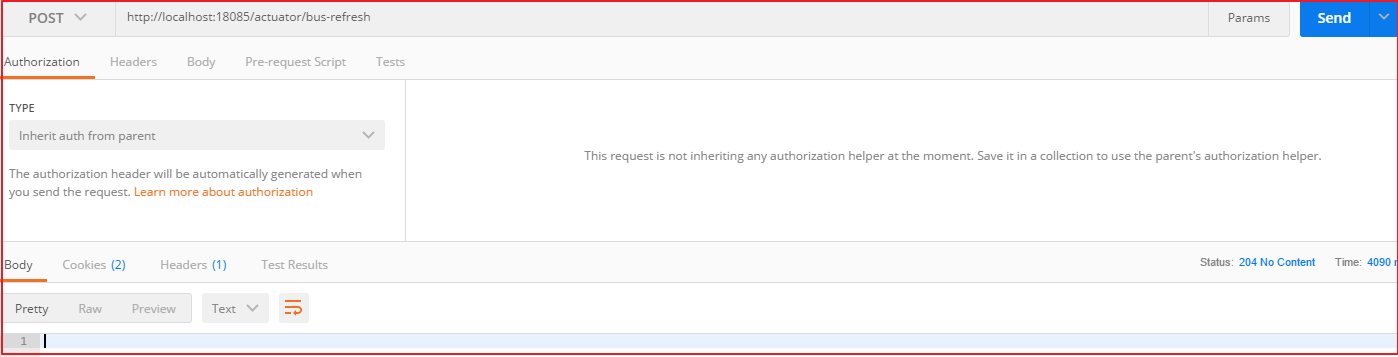
(5)修改码云配置

修改码云的配置，修改后并提交，修改如下：



(6)刷新配置

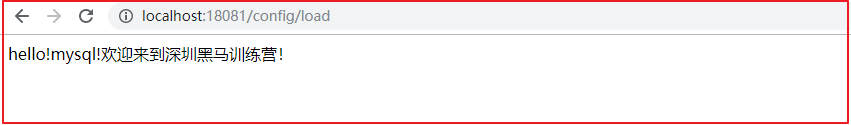
使用Postman以POST方式请求<http://localhost:18085/actuator/bus-refresh>



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

(7)刷新测试

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



消息总线实现消息分发过程：

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

## SpringCloud 总架构图

