

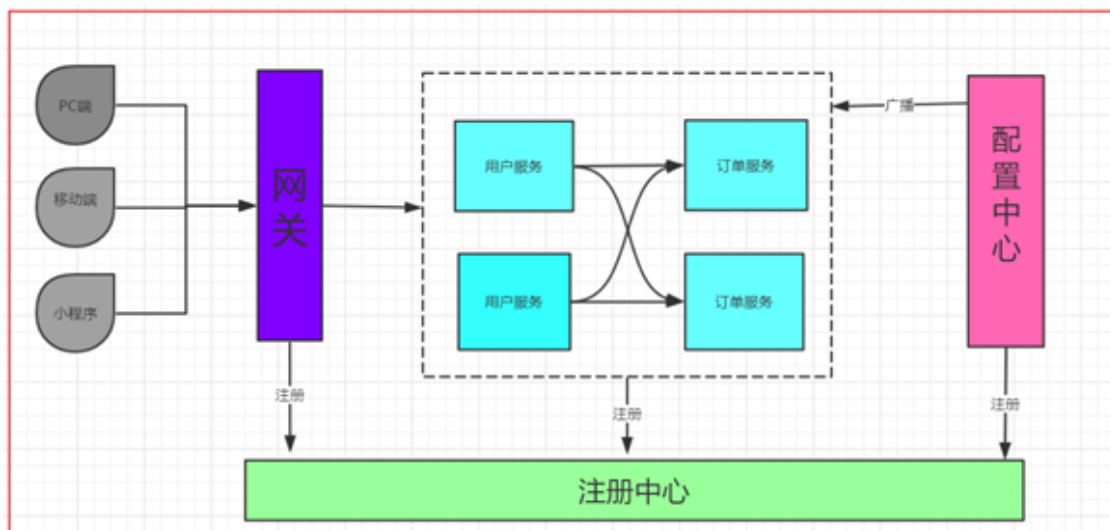
- 学习目标
- SpringCloud 总架构图
- 一、远程调用 Spring Cloud Feign
 - 1.1 简介
 - 1.2 入门案例
 - 1.3 负载均衡
 - 1.4 熔断器支持
 - 1.5 请求压缩和响应压缩
 - 1.6 配置日志级别
- 二、网关 Spring Cloud Gateway
 - 2.1 简介
 - 功能特性
 - 术语解释
 - 2.2 快速入门
 - 2.3 动态路由
 - 2.4 路由前缀
 - 2.5 过滤器
 - 2.5.1 简介
 - 2.5.2 过滤器配置
 - 2.6 自定义全局过滤器【重点】
- 三、配置中心 Spring Cloud Config
 - 3.0 Config 简介
 - 3.1 配置中心整合步骤：
 - 3.2 Git配置管理
 - 3.2.1 远程Git仓库
 - 3.2.2 创建远程仓库
 - 3.2.3 创建配置文件
 - 3.3 搭建配置中心微服务
 - 3.4 服务去获取配置中心配置
 - 3.5 配置中心存在的问题
- 四、消息总线 Spring Cloud Bus
 - 4.1 简介
 - 4.2 整合案例

- 4.2.1 改造配置中心
- 4.2.2 改造生产者服务
- 4.3 测试

学习目标

- 理解什么是远程调用Feign【重点】
- 理解什么是网关Gateway【重点】
- 能够搭建网关微服务
- 理解什么是配置中心Config【重点】
- 能够搭建配置中心微服务

SpringCloud 总架构图



一、远程调用 Spring Cloud Feign

前面学习中，使用RestTemplate大大简化了远程调用的代码：

```
String baseUrl = "http://user-service/user/findById?id=1"+ id;  
User user = restTemplate.getForObject(baseUrl, User.class);
```

如果就学到这里，你可能以后需要编写类似的大量重复代码，格式基本相同，无非参数不一样。有没有更优雅的方式，来对这些代码再次优化呢

这就是接下来要学的Feign的功能了。

1.1 简介

Feign 的英文表意为“假装，伪装，变形”，是一个http请求调用的轻量级框架，是以Java接口的方式发送Http请求，而不用像Java中通过封装HTTP请求url的方式直接调用。Feign通过处理注解，将请求模板化，当实际调用的时候，传入参数，根据参数再应用到请求上，进而转化成真正的请求，这种请求相对而言比较直观。

Feign被广泛应用在Spring Cloud 的解决方案中，是学习基于Spring Cloud 微服务架构不可或缺的重要组件。

封装了Http调用流程，更符合面向接口化的编程习惯。类似Dubbo服务调用。

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

1.2 入门案例

目标：使用Feign替代RestTemplate发送Rest请求。使之更符合面向接口化的编程习惯。

实现步骤：

1. 导入依赖feign的starter
2. 启动引导类加@EnableFeignClients注解
3. 编写FeignClient接口，使用SpringMVC的注解
4. 在Controller中注入Feign接口，直接调用，无需实现类
5. 访问接口测试

实现过程：

1. 导入依赖feign的starter

```

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

```

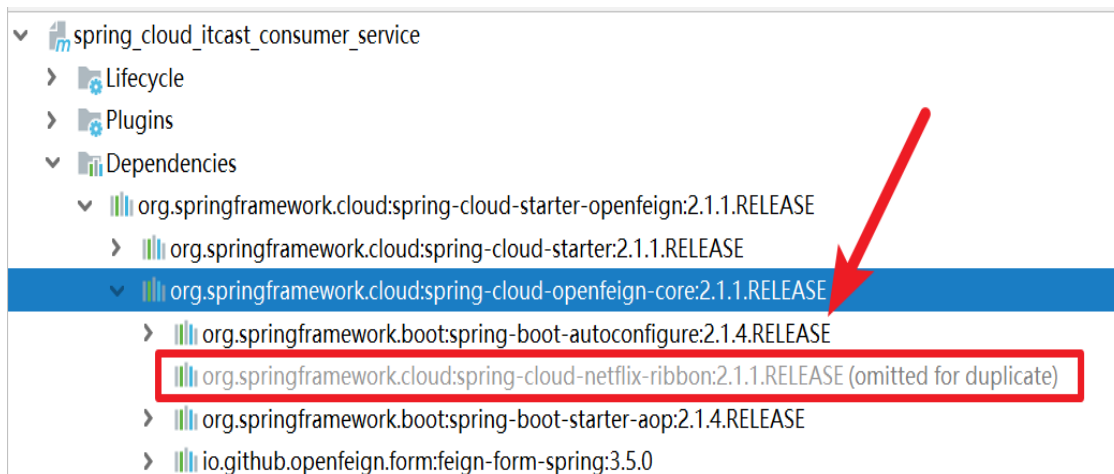
2. 启动引导类加@EnableFeignClients注解

```

@SpringBootApplication
@EnableFeignClients// 开启Feign功能
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class,args);
    }
}

```

Feign中已经自动集成Ribbon负载均衡



1. 编写FeignClient接口，使用SpringMVC的注解

- 在consumer_service中编写Feign客户端接口UserService

```

@FeignClient("user-service")// 指定feign调用的服务
public interface UserService {

    @GetMapping("/user/{id}")
    UserVO findById(@PathVariable("id") Integer id);
}

```

- Feign会通过动态代理，帮我们生成实现类。
- 注解@FeignClient声明Feign的客户端接口，需指明服务名称
- 接口定义的方法，采用SpringMVC的注解。Feign会根据注解帮我们逆向生成URL地址然后请求

2. 在Controller中注入UserService接口，直接调用，无需实现类

```
@RestController
public class FeignConsumerController {
    /**
     * 注入Feign客户端接口：优雅的像一首诗！
     * 三个优点：
     * 可复用：哪里需要注入哪里
     * 可读性：有层次结构感觉
     * 易于管理：如果接口地址发送变化，只需要改一次
     */
    @Autowired
    private UserService userService;

    @RequestMapping("/feignconsumer/{id}")
    public UserVO hellofeign(@PathVariable Integer id){
        return userService.findById(id);
    }
}
```

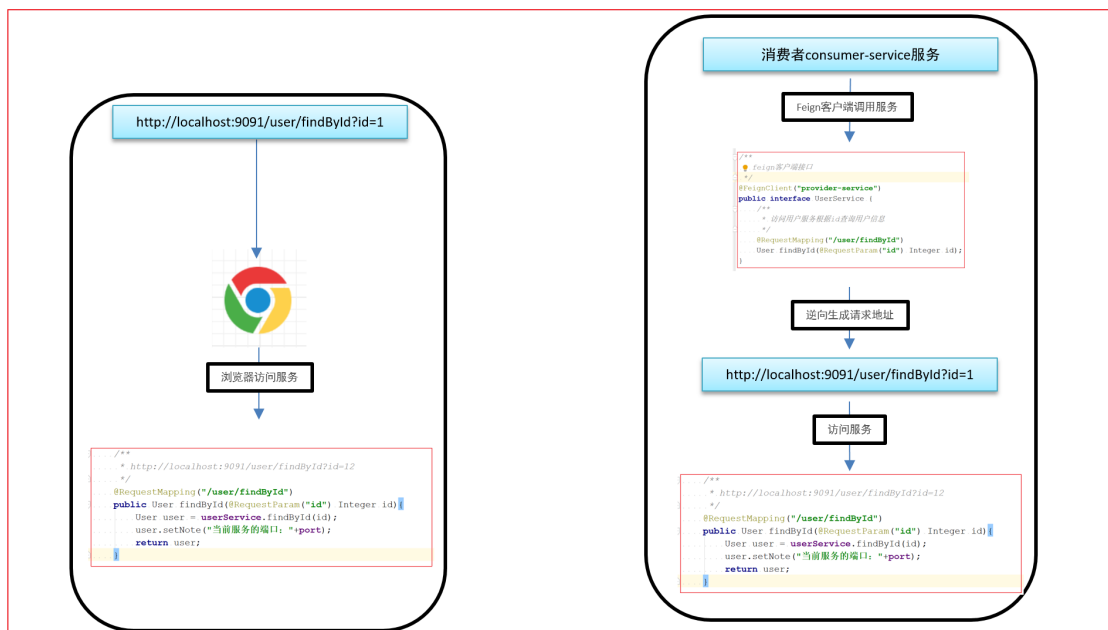
3. 启动测试：访问接口 <http://localhost:8080/feignconsumer/1>，正常获取结果

```
GET http://localhost:8080/feignconsumer/1 Send

Pretty Raw Preview Visualize JSON

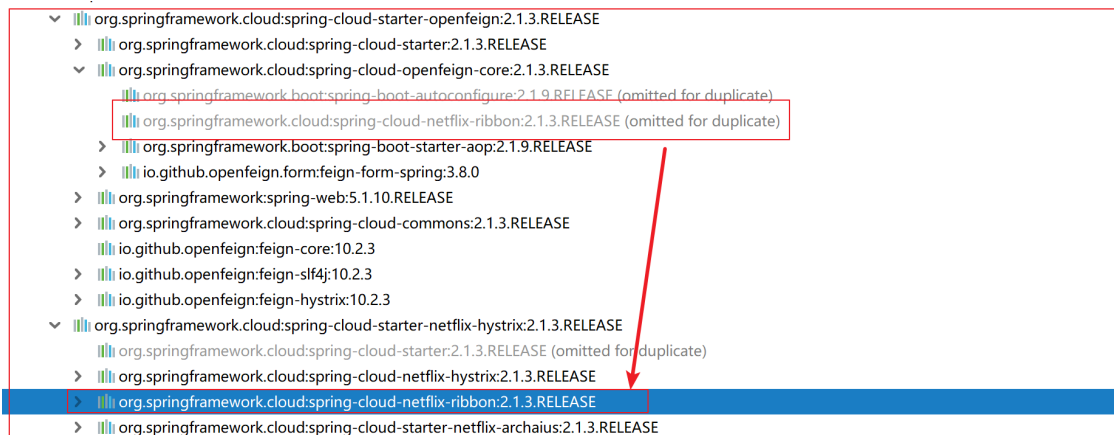
1 {
2   "id": 1,
3   "username": "zhangsan",
4   "password": "123456",
5   "name": "张三",
6   "age": 13,
7   "sex": 1,
8   "birthday": "2006-07-31T16:00:00.000+0000",
9   "created": "2019-05-15T16:00:00.000+0000",
10  "updated": "2019-05-15T16:00:00.000+0000",
11  "note": "张三"
12 }
```

Feign实现原理简单分析：



1.3 负载均衡

负载均衡是远程过程调用必备的要素。Feign本身集成了Ribbon，因此不需要额外引入依赖，也不需要再注册RestTemplate对象。即可，无感知使用负载均衡这一特性。



Feign内置Ribbon，默认设置了连接超时时间是2000毫秒(2秒)，和读取超时时间是5000毫秒(5秒)。我们可以通过手动配置来修改。

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。

```
# 配置熔断器超时时间
# 连接超时时长
ribbon.ConnectTimeout: 2000
# 读取数据超时时长
ribbon.ReadTimeout: 5000
# 当前服务器的重试次数【针对请求】
ribbon.MaxAutoRetries: 0
# 重试多少次服务【针对服务】
ribbon.MaxAutoRetriesNextServer: 0
# 是否对所有的请求方式都重试
ribbon.OkToRetryOnAllOperations: false
```

1.4 熔断器支持

Feign本身也集成Hystrix熔断器，starter内查看。

服务降级方法实现步骤：

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客户端接口
3. 在@FeignClient注解中，指定Fallback处理类。
4. 测试服务降级效果

实现过程：

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

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

2. 定义一个类UserServiceFallBack，实现刚才编写的UserFeignClient，作为FallBack的处理类

```
@Component
public class UserServiceFallBack implements UserService{

    @Override
    public UserVO findById(Integer id) {
        UserVO user = new UserVO();
        user.setId(id);
        user.setUsername("用户不存在!!!");
        return user;
    }
}
```

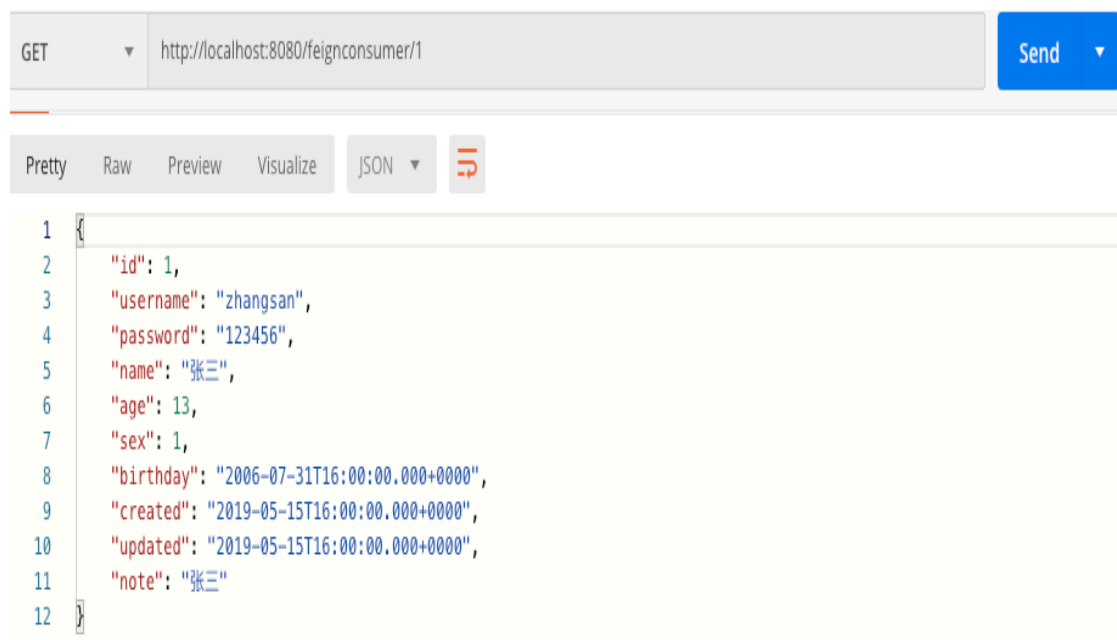
3. 在@FeignClient注解中，指定FallBack处理类。。

```
/**
 * Feign客户端接口：接口的作用，将请求调用封装到当前的接口中
 * 使用的时候，直接注入Controller即可使用！
 * @FeignClient() 在注解的作用，声明当前接口为feign客户端接口
 * value属性：设置访问服务的名称
 * fallback属性：设置服务降级的处理方法
 * configuration属性：设置Feign的配置类
 * @author by SangJiacun
 * @Date 2020/8/14 10:33
 */
@FeignClient(value = "provider-service", fallback =
UserServiceFallBack.class)// 指定feign调用的服务
public interface UserService {

    @GetMapping("/user/{id}")
    UserVO findById(@PathVariable("id") Integer id);
}
```

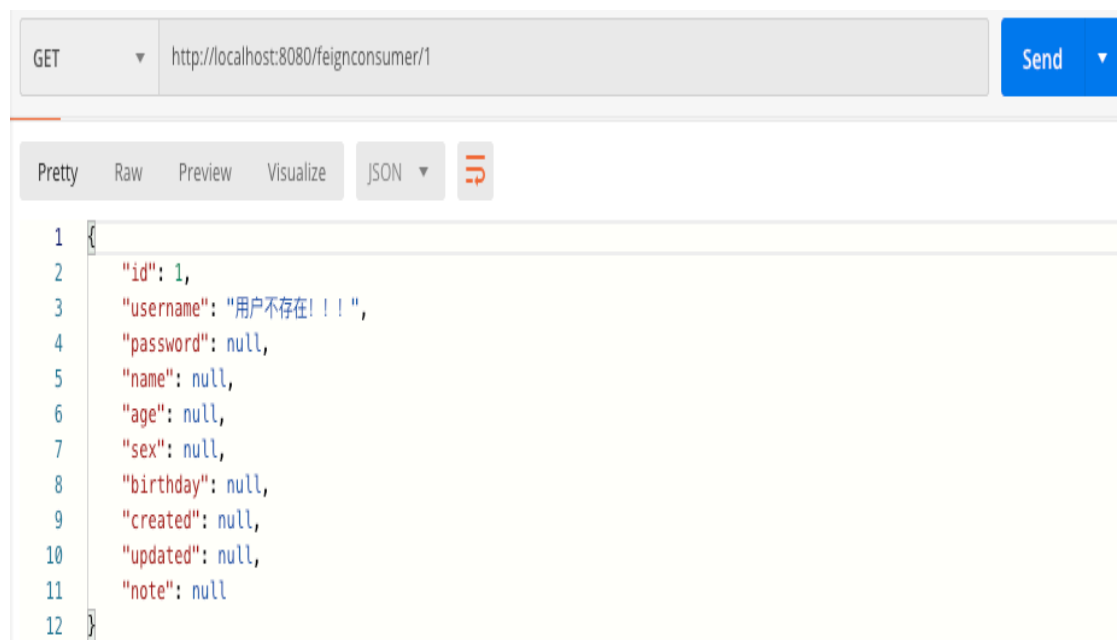

4. 重启测试：关闭provider-service服务，然后在页面访问； <http://localhost:8080/feignConsumer/2>

关闭前：



```
GET http://localhost:8080/feignconsumer/1 Send
Pretty Raw Preview Visualize JSON
1 {
2   "id": 1,
3   "username": "zhangsan",
4   "password": "123456",
5   "name": "张三",
6   "age": 13,
7   "sex": 1,
8   "birthday": "2006-07-31T16:00:00.000+0000",
9   "created": "2019-05-15T16:00:00.000+0000",
10  "updated": "2019-05-15T16:00:00.000+0000",
11  "note": "张三"
12 }
```

关闭后：



```
GET http://localhost:8080/feignconsumer/1 Send
Pretty Raw Preview Visualize JSON
1 {
2   "id": 1,
3   "username": "用户不存在!!!",
4   "password": null,
5   "name": null,
6   "age": null,
7   "sex": null,
8   "birthday": null,
9   "created": null,
10  "updated": null,
11  "note": null
12 }
```

1.5 请求压缩和响应压缩

SpringCloudFeign支持对请求和响应进行GZIP压缩，以提升通信过程中的传输速度。

- 为什么RPC远程调用的方式性能更高？传输的数据量小
- Http的优势是什么？跨编程语言

通过配置开启请求与响应的压缩功能：

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

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

```
# 设置压缩的数据类型
feign.compression.request.mime-types:
text/html,application/xml,application/json
# 设置触发压缩的大小下限
feign.compression.request.min-request-size: 2048
```

1.6 配置日志级别

在发送和接收请求的时候，Feign定义了日志的输出定义了四个等级：这里我们配置测试一下。

级别	说明
NONE	不做任何记录
BASIC	只记录输出Http 方法名称、请求URL、返回状态码和执行时间
HEADERS	记录输出Http 方法名称、请求URL、返回状态码和执行时间 和 Header 信息
FULL	记录Request 和Response的Header， Body和一些请求元数据

实现步骤：

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

实现过程：

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

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

2. 在consumer_service编写配置类，定义日志级别

```
/**
 * feign客户端的配置类
 * 配置日志级别:
 * 动态的注入配置信息到Spring的容器中
 * @author by SangJiacun
 * @Date 2020/8/14 11:01
 */
@Configuration
public class FeignConfiguration {
    /**
     * 动态的注入配置信息到Spring的容器中
     * 四种日志级别:
     * None: 没有日志
     * Basic: 基本的日志, 请求方法, 请求URL地址, 响应的状态码, 请求执行时间
     * Headers: 与BASIC的日志基本一样, 多请求的头部信息Header
     * Full: 全部的日志信息, 请求的体, 响应的体
     */
    @Bean
    public Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

3. 在consumer_service的FeignClient中指定配置类

```
@FeignClient(value = "provider-service",
            fallback = UserServiceFallBack.class,
            configuration = FeignConfiguration.class)
public interface UserService {

    @GetMapping("/user/{id}")
    UserVO findById(@PathVariable("id") Integer id);
}
```

4. 重启项目，即可看到每次访问的日志

```
2020-08-14 11:04:41.572 INFO 7747 --- [nio-8080-exec-6] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-08-14 11:04:41.572 INFO 7747 --- [nio-8080-exec-6] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-08-14 11:04:41.588 INFO 7747 --- [nio-8080-exec-6] o.s.web.servlet.DispatcherServlet : Completed initialization in 16 ms
2020-08-14 11:04:41.916 DEBUG 7747 --- [vider-service-1] com.sjc.api.UserService : [UserService#findById] --> GET http://provider-service/user/1 HTTP/1.1
2020-08-14 11:04:41.916 DEBUG 7747 --- [vider-service-1] com.sjc.api.UserService : [UserService#findById] Accept-Encoding: deflate
2020-08-14 11:04:41.916 DEBUG 7747 --- [vider-service-1] com.sjc.api.UserService : [UserService#findById] Accept-Encoding: gzip
2020-08-14 11:04:41.916 DEBUG 7747 --- [vider-service-1] com.sjc.api.UserService : [UserService#findById] --> END HTTP (0-byte body)
2020-08-14 11:04:42.157 INFO 7747 --- [vider-service-1] c.netflix.config.ChainedDynamicProperty : Flipping property: provider-service.ribbon.ActiveConnectionsLimit to use NEXT prc
2020-08-14 11:04:42.210 INFO 7747 --- [vider-service-1] c.n.u.concurrent.ShutdownEnabledTimer : Shutdown hook installed for: NFLoadBalancer-PingTimer-provider-service
2020-08-14 11:04:42.211 INFO 7747 --- [vider-service-1] c.netflix.loadbalancer.BaseLoadBalancer : Client: provider-service instantiated a LoadBalancer: DynamicServerListLoadBalanc
2020-08-14 11:04:42.216 INFO 7747 --- [vider-service-1] c.n.l.DynamicServerListLoadBalancer : Using serverListUpdater PollingServerListUpdater
2020-08-14 11:04:42.220 INFO 7747 --- [vider-service-1] c.n.l.DynamicServerListLoadBalancer : DynamicServerListLoadBalancer for client provider-service initialized: DynamicSer
```

二、网关 Spring Cloud Gateway

2.1 简介

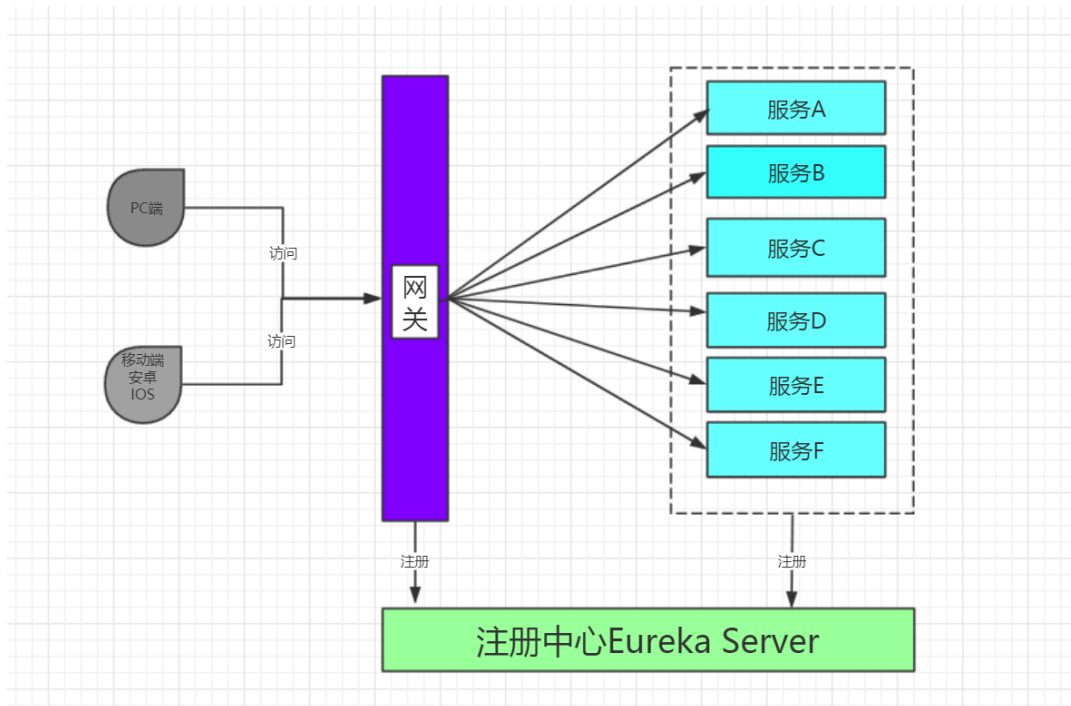
Gateway网关是我们服务的守门神，所有微服务的统一入口。Spring Cloud Gateway 是 Spring Cloud的一个全新项目，该项目是基于 Spring 5.0，Spring Boot 2.0 和 Project Reactor等技术开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

在Gateway之前，SpringCloud并不自己开发网关，可能是觉得Netflix公司的Zuul不行吧，然后自己就写了一个，也是替代Netflix Zuul。其不仅提供统一的路由方式，并且基于 Filter链的方式提供了网关基本的功能，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

功能特性

- 动态路由
- Predicates 和 Filters 作用于特定路由
- 集成 Hystrix 断路器
- 简单好用的 Predicates 和 Filters
- 限流
- 路径重写



- 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是我们服务的统一入口

术语解释

- **Route（路由）**：这是网关的基本模块。它由一个ID，一个目标URI，一组断言和一组过滤器定义。如果断言为真，则路由匹配。
- **Predicate（断言）**：本质就是拦截的规则，这是一个 Java 8 的 Predicate。输入类型是一个 ServerWebExchange。我们可以使用它来匹配来自 HTTP 请求的任何内容，例如 headers 或参数。
- **Filter（过滤器）**：这是 `org.springframework.cloud.gateway.filter.GatewayFilter` 的实例，我们可以使用它修改请求和响应。

2.2 快速入门

搭建网关微服务，实现服务路由分发。

<http://访问网关的服务/user/findById?id=11> ==>
<http://127.0.0.1:9091/user/findById?id=11>

实现步骤：

1. 创建SpringBoot工程gateway_server
2. 勾选starter：网关、Eureka客户端
3. 编写基础配置：端口，应用名称，注册中心地址
4. 编写路由规则：唯一表示id，路由url地址，路由限定规则[拦截请求的规则]
 - i. <http://localhost:10010/user/findById?id=11> ==>
<http://127.0.0.1:9091/user/findById?id=11>
5. 启动网关服务进行测试

实现过程：

1. 创建SpringBoot工程gateway_server

New Project

Project Metadata

Group: com.itheima

Artifact: gateway_server

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

Language: Java ▾

Packaging: Jar ▾

Java Version: 8 ▾

Version: 0.0.1-SNAPSHOT

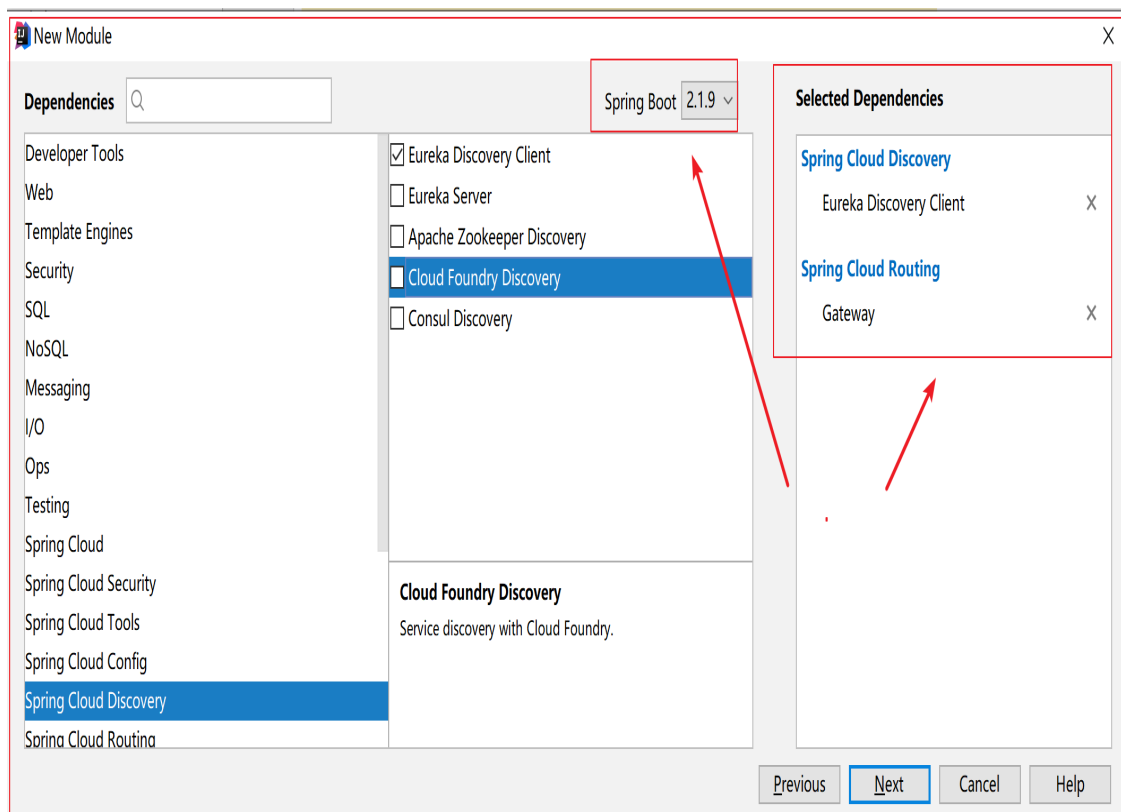
Name: gateway_server

Description: Demo project for Spring Boot

Package: com.itheima

Previous Next Cancel Help

2. 勾选Starter：网关、Eureka客户端



3. 启动引导类开启注册中心Eureka客户端发现

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

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

4. 编写基础配置

- 在gateway_server中创建application.yml文件，配置

```
# 端口
server.port: 10010
# 应用名
spring.application.name: api-gateway
# 注册中心地址
eureka.client.service-url.defaultZone:
http://eureka1:8761/eureka,http://eureka2:8762/eureka
```


5. 编写路由规则

- 需要用网关来路由provier_service服务，查看服务ip和端口

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

eureka2

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.1.103:8080
EUREKA-SERVER	n/a (2)	(2)	UP (2) - 192.168.1.103:eureka-server:8761, 192.168.1.103:eureka-server:8762
PROVIDER-SERVICE	n/a (2)	(2)	UP (2) - 192.168.1.103:provider-service:9091, 192.168.1.103:provider-service:9092

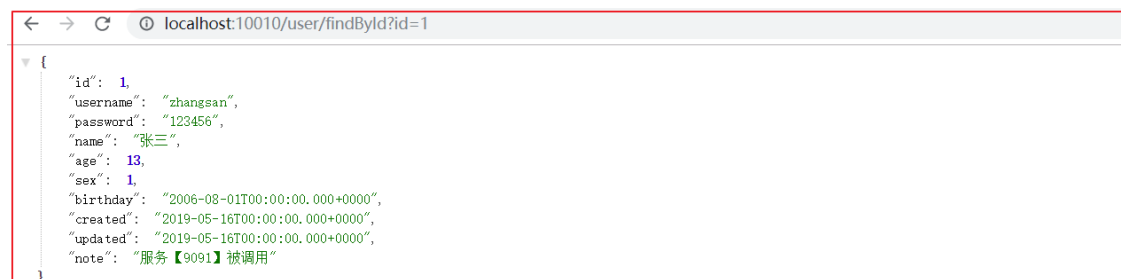
- 修改gateway_server的配置文件application.yml，配置网关内容

```
spring:
  cloud:
    gateway:
      # 路由si(集合)
      routes:
        # id唯一标识
        - id: user-service-route
          # 路由服务地址
          uri: http://127.0.0.1:9091
          # 断言
          predicates:
            - Path=/user/**
```

- 将符合 path 规则的请求，路由到 uri 参数指定地址。
- 举例：<http://localhost:10010/user/findById?id=1> 路由转发到 <http://localhost:9091/user/findById?id=1>

2. 启动GatewayApplication进行测试

- 访问路径中，必须包含路由规则的映射路径/user才会被路由



2.3 动态路由

```
# 路由si(集合)
routes:
  # id唯一标识
  - id: user-service-route
    # 路由地址
    uri: http://127.0.0.1:9091
    # 路由拦截地址(断言)
  predicates:
    - Path=/user/**
```



刚才路由规则中，我们把路径对应服务地址写死了！如果服务提供者是集群的话，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
 - 因为已经配置了Eureka客户端，可以从Eureka获取服务的地址信息，修改application.yml文件如下
 - 路由配置中uri所用的协议为lb时，gateway将把provider-service解析为实际的主机和端口，并通过Ribbon进行负载均衡。

```
# 注解版
spring:
  cloud:
    gateway:
      # 路由si(集合)
      routes:
        # id唯一标识
        - id: user-service-route
          # 路由地址
          uri: http://127.0.0.1:9091
          # 采用lb协议，会从Eureka注册中心获取服务请求地址
          # 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
          # 规则: lb协议+服务名称
          uri: lb://user-service
          # 路由拦截地址(断言)
      predicates:
        - Path=/user/**
```

- 启动GatewayApplication测试
 - 这次gateway进行路由时，会利用Ribbon进行负载均衡访问。日志中可以看到使用了负载均衡器。

GET

http://localhost:10010/user/1

Send

Pretty

Raw

Preview

Visualize

JSON

1

2

3

4

5

6

7

8

9

10

11

12

"id": 1,

"username": "zhangsan",

"password": "123456",

"name": "张三",

"age": 13,

"sex": 1,

"birthday": "2006-07-31T16:00:00.000+00:00",

"created": "2019-05-15T16:00:00.000+00:00",

"updated": "2019-05-15T16:00:00.000+00:00",

"note": "张三"

2020-08-14 11:18:02.025 INFO --- [elastic-2] c.netflix.config.ChainedDynamicProperty : Flipping property: provider-service.ribbon.ActiveConnectionsLimit to use NEXT property: nides.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

2020-08-14 11:18:02.046 INFO --- [elastic-2] c.n.c.concurrent.ShutdownNotifier : Shutdown hook installed for: NFLoadBalancer-PingPong-provider-service

2020-08-14 11:18:02.046 INFO --- [elastic-2] c.netflix.loadbalancer.BaseLoadBalancer : Client: provider-service instantiated a LoadBalancer: DynamicServerListLoadBalancer: (NFLoadBalancer:name=provider-service,current list of Servers=[],Load balancer stats=Zone status: using serverListUpdater: PollingServerListUpdater

2020-08-14 11:18:02.052 INFO --- [elastic-2] c.n.l.DynamicServerListLoadBalancer : Flipping property: provider-service.ribbon.ActiveConnectionsLimit to use NEXT property: nides.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

2020-08-14 11:18:02.066 INFO --- [elastic-2] c.netflix.config.ChainedDynamicProperty : Flipping property: provider-service.ribbon.ActiveConnectionsLimit to use NEXT property: nides.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

2020-08-14 11:18:02.068 INFO --- [elastic-2] c.n.l.DynamicServerListLoadBalancer : DynamicServerListLoadBalancer for client provider-service initialized: DynamicServerListLoadBalancer: (NFLoadBalancer:name=provider-service,current list of Servers=[192.168.1.183:9991

13)Server stats: ([Server:192.168.1.183:9991; Zone=defaultZone; Total Requests:0; Successful connection failures:0; Total blackout seconds:0; Last connection made:Thu Jan 01 00:00:00 CST 1970; First connection made:Thu Jan 01 00:00:00 CST 1970; Active Connections:0; Total failure

14 1]ServerList:org.springframework.cloud.netflix.ribbon.eureka.DomainExtractingServerList@60a0b8

2020-08-14 11:18:03.009 INFO --- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty : Flipping property: provider-service.ribbon.ActiveConnectionsLimit to use NEXT property: nides.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

2.4 路由前缀

第一：添加前缀：

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添加。可以起到隐藏接口地址的作用，避免接口地址暴露。

1. 配置请求地址添加路径前缀过滤器

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service-route # 路由id, 可以随意写
          uri: lb://user-service
          # 路由断言, 配置映射路径
        predicates:
          - Path=/**
          # 请求地址添加路径前缀过滤器
        filters:
          - PrefixPath=/user
```

2. 重启GatewayApplication

3. 配置完成的效果：

配置	访问地址	路由地址
PrefixPath=/user	localhost:10010/1	localhost:9091/user/findById/1
PrefixPath=/user/abc	localhost:10010/findById?id=1	localhost:9091/user/abc/findById?id=1

第二：去除前缀：

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。

1. 配置去除路径前缀过滤器

```
spring:
  cloud:
    gateway:
      routes:
        - id: user-service-route # 路由id, 可以随意写
          # 代理服务地址; lb表示从Eureka中获取具体服务
          uri: lb://user-service
          # 路由断言, 配置映射路径
          predicates:
            - Path=/**
          # 去除路径前缀过滤器
          filters:
            - StripPrefix=1
```

2. 重启GatewayApplication

3. 访问查看效果

配置	访问地址	路由地址
StripPrefix=1	localhost:10010/api/user/findById?id=1	localhost:9091/user/findById?id=1
StripPrefix=2	localhost:10010/aa/api/user/findById?id=1	localhost:9091/user/findById?id=1

2.5 过滤器

2.5.1 简介

过滤器作为网关的其中一个重要功能，就是实现请求的鉴权。前面的 [路由前缀](#) 章节中的功能也是使用过滤器实现的。

Gateway自带过滤器有几十个，常见自带过滤器有：

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
PrefixPath	对匹配上的请求路径添加前缀

详细说明官方[链接](#)

使用场景：

- 请求鉴权：如果没有访问权限，直接进行拦截
- 异常处理：记录异常日志
- 服务调用时长统计

2.5.2 过滤器配置

过滤器类型： Gateway有两种过滤器

- 局部过滤器：只作用在当前配置的路由上。
- 全局过滤器：作用在所有路由上。

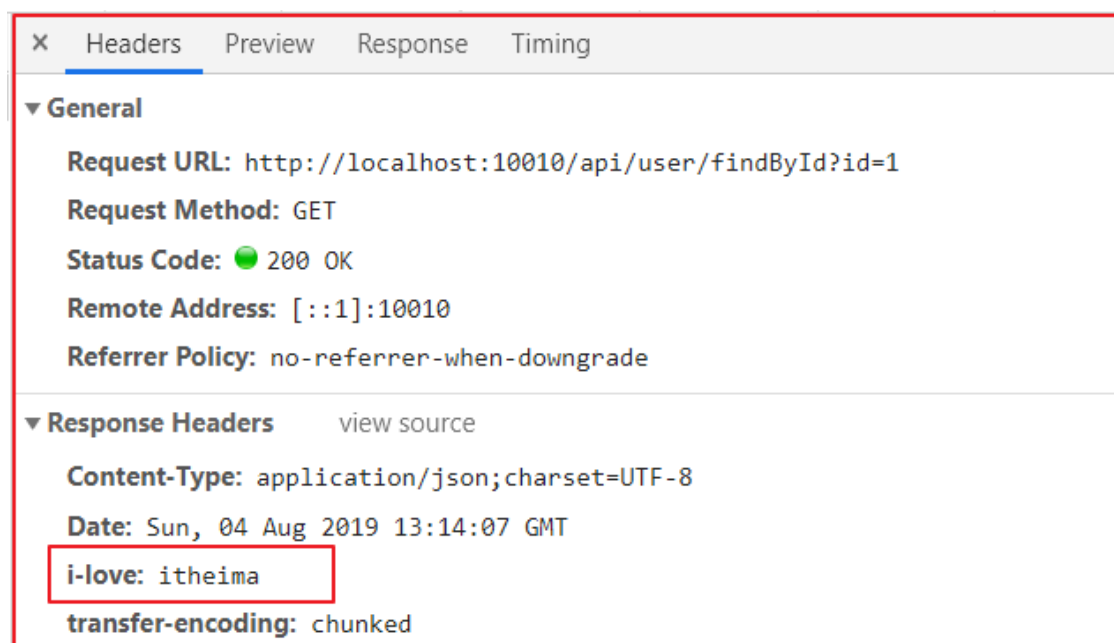
配置全局过滤器：

对输出的响应设置其头部属性名称为i-love,值为itheima

1. 修改配置文件

```
spring:
  cloud:
    gateway:
      # 配置全局默认过滤器
      default-filters:
        # 往响应过滤器中加入信息
        # 不能写入空格，输入特殊字符
        - AddResponseHeader=i-love,sjc
```

2. 查看浏览器响应头信息



2.6 自定义全局过滤器【重点】

需求： 模拟登录校验

拦截器，业务逻辑完全一样，把所有经过网关的请求地址进行过滤

token就是一把钥匙

实现步骤：

1. 在gateway_server中，定义全局过滤器：
 - 名称自定义
 - 实现GlobalFilter和 Ordered接口
2. 编写业务逻辑代码判断：
 - 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未授权。
3. 访问接口测试，加token和不加token。

实现过程：

1. 在gateway_server中，全局过滤器类MyGlobalFilter，实现GlobalFilter和 Ordered接口
2. 编写业务逻辑代码判断：
 - i. 如果请求中有token参数，则认为请求有效，放行
 - ii. 如果没有则拦截提示未授权

```

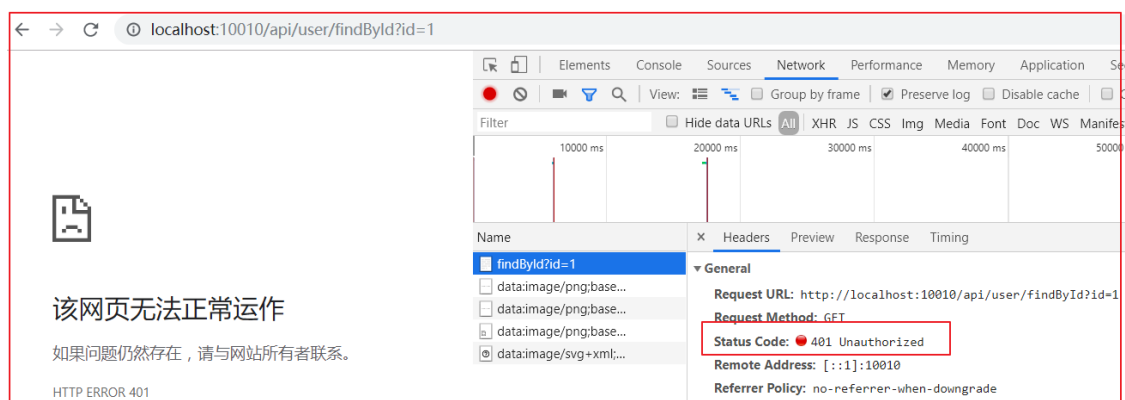
@Component
public class MyGlobalFilter implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
        System.out.println("-----全局过滤
器MyGlobalFilter-----");
        //1、获取参数中的token，以及token的值
        String token =
exchange.getRequest().getQueryParams().getFirst("token");
        //2、如果token的值为空，则拦截
        if (StringUtils.isBlank(token)) {

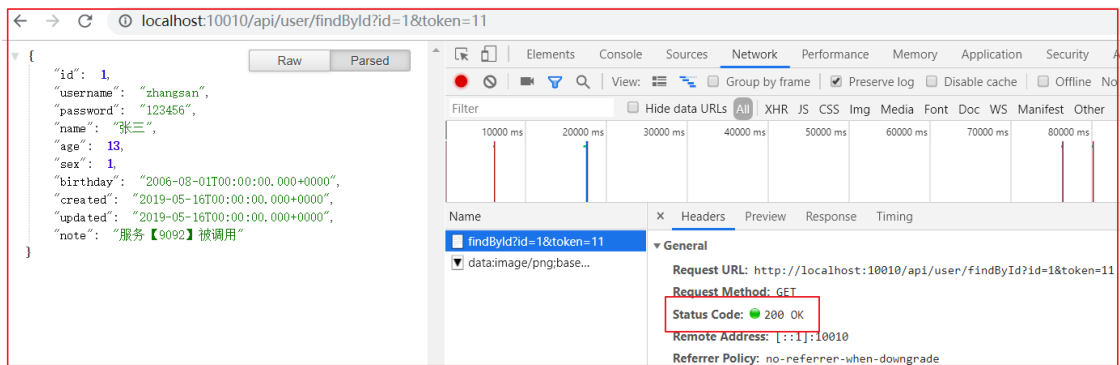
exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
        return chain.filter(exchange);
    }
    /**
     * 定义过滤器执行顺序
     * 返回值越小，越靠前执行
     * @return
     */
    @Override
    public int getOrder() {
        return 0;
    }
}

```

3. 访问：<http://localhost:10010/api/user/findById?id=1>



4. 访问：<http://localhost:10010/api/user/findById?id=1&token=11>

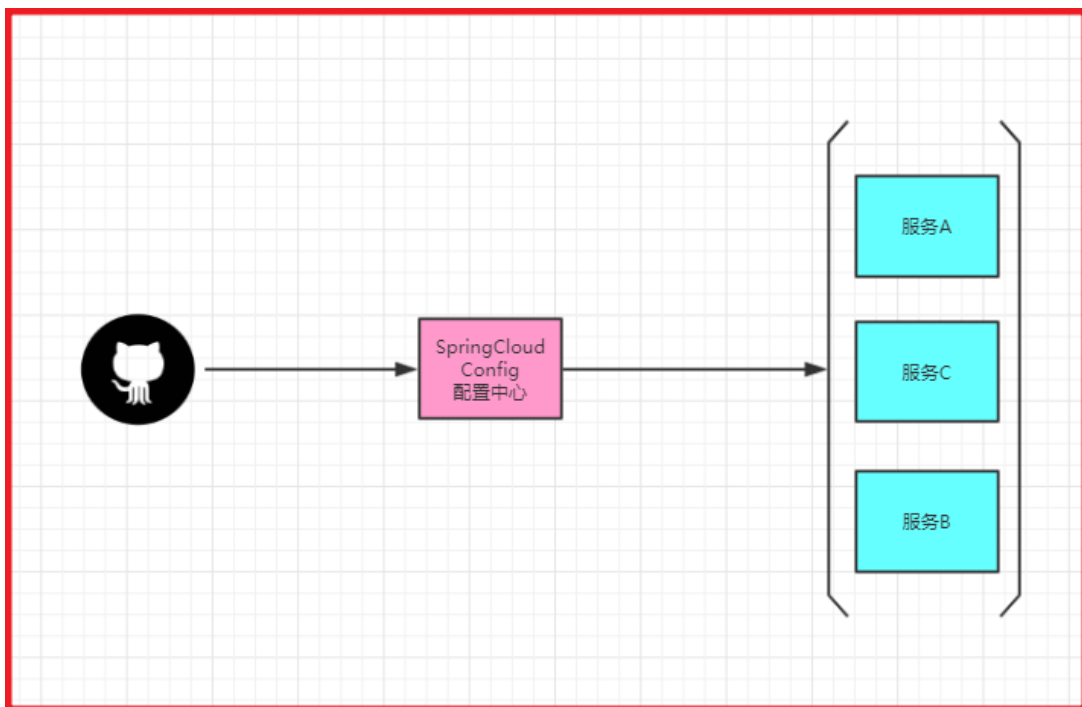


三、配置中心 Spring Cloud Config

3.0 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！

一句话概括：统一管理众多微服务配置文件的一个微服务



3.1 配置中心整合步骤：

1. 配置文件集中放在码云
2. 配置中心获取码云配置文件
3. 用户服务获取配置中心文件

3.2 Git配置管理

3.2.1 远程Git仓库

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

3.2.2 创建远程仓库

1. 首先使用码云上的git仓库需要先注册账户
2. 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



3. 配置仓库 名称和路径


新建仓库

1、配置仓库名称

仓库名称 

itheima-spring-cloud-config

2、配置路径

归属  / 路径

itheima-spring-cloud-config

仓库地址: <https://gitee.com/liuyaxiong01/itheima-spring-cloud-config>

仓库介绍 非必填

用简短的语言来描述一下吧

是否开源

☐ 私有

☒ 公开

3、选择公开仓库

任何人都可以访问该仓库的代码和其他任何形式的资源

选择语言

Java

添加 .gitignore

请选择 .gitignore 模板

添加开源许可证 

请选择开源许可证

4、选择Java语言

☒ 使用Readme文件初始化这个仓库

☐ 使用Issue模板文件初始化这个仓库 

☐ 使用Pull Request模板文件初始化这个仓库 

选择分支模型 (仓库初始化后将根据所选分支模型创建分支)

单分支模型 (只创建 master 分支)

 导入已有仓库

创建

5、创建

尝试码云企业版?

- ✓ 不止更多成员、更高配额
- ✓ 灵活支撑协作研发、代码交付、IT 教育培训等场景代码托管需求

他们都在用

已有超过 60000 企业研发团队选择码云企业版, 60% 客户来自口碑推荐。



了解更多

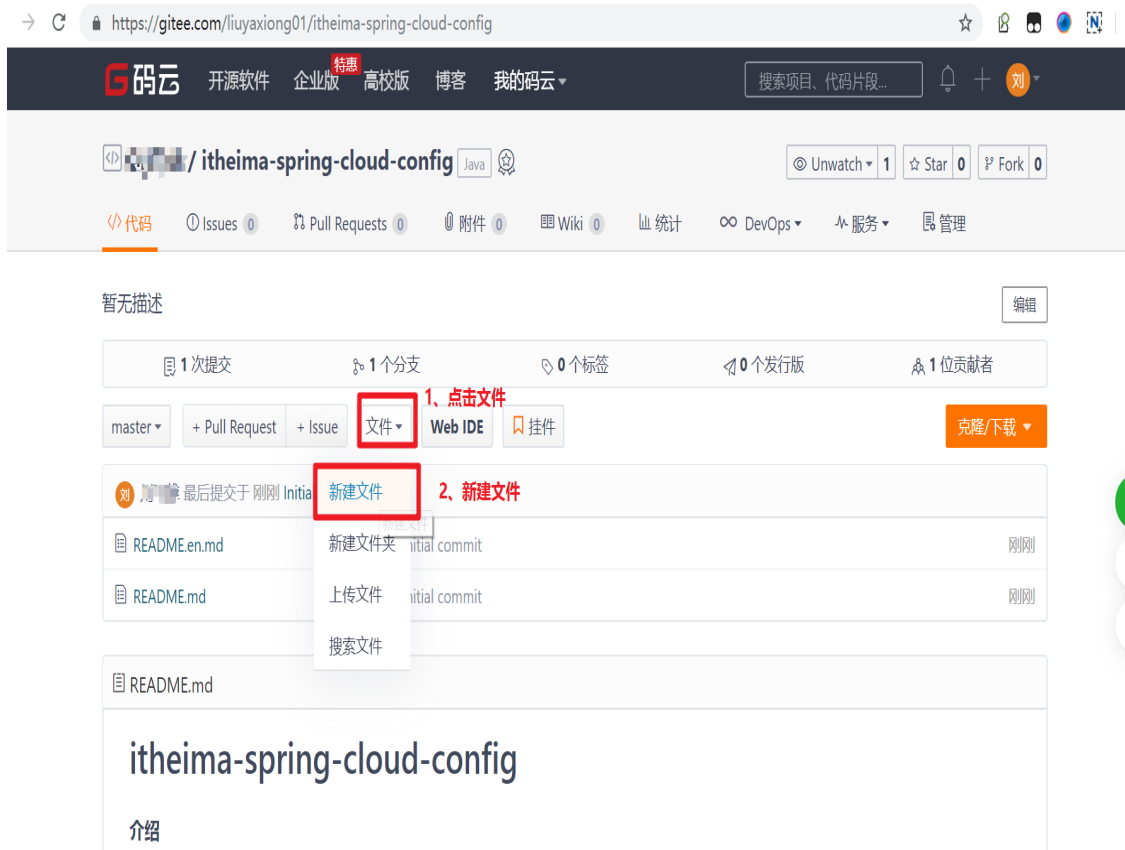
企业版介绍

社区版与企业版功能对比

3.2.3 创建配置文件

1. 在新建的仓库中创建需要被统一配置管理的配置文件

- 配置文件的命名方式：{application}-{profile}.yaml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.yaml
 - 测试环境 user-test.yaml
 - 生产环境 user-pro.yaml



2. 将user-service工程里的配置文件application.yml内容复制作为user-dev.yaml文件内容

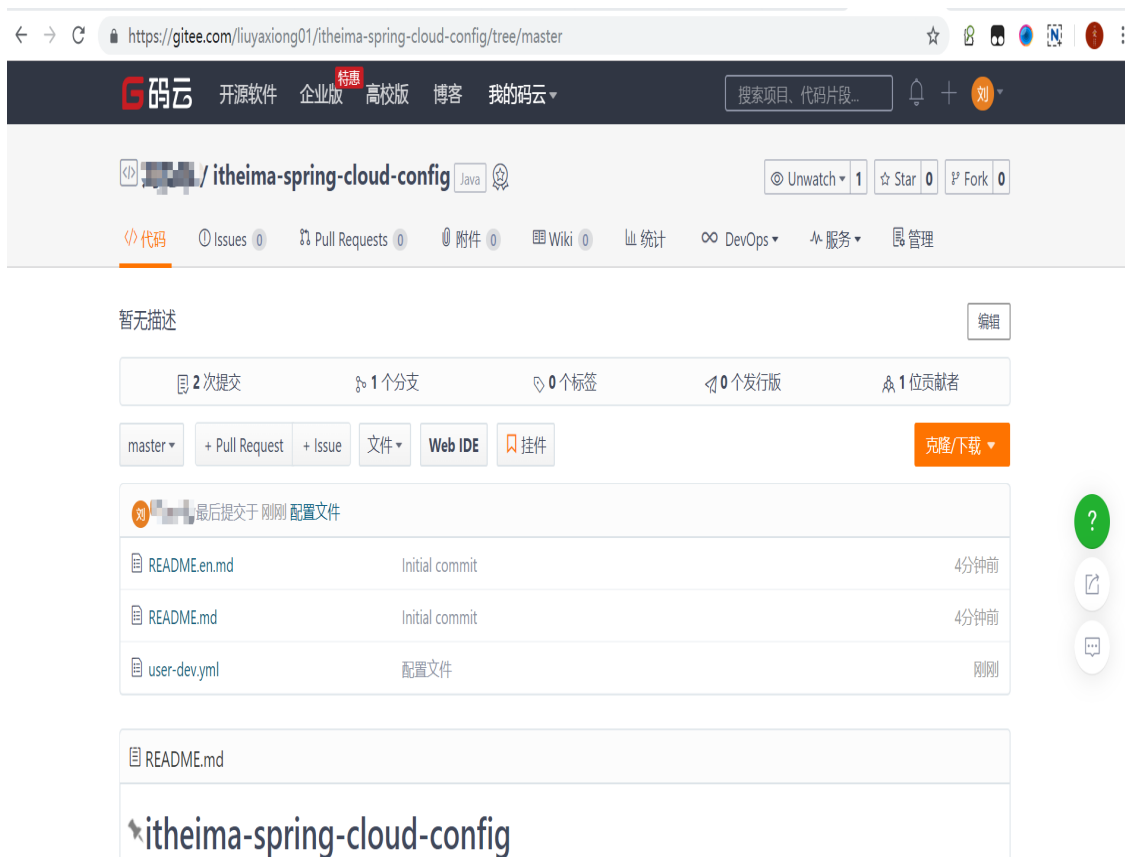
user-dev.yml 594 Bytes

一键复制 编辑 Web IDE 原始数据 按行查看 历史

刘亚雄 提交于 11分钟前 · [配置文件](#)

```
1 # 端口
2 server.port: ${port:9091}
3 # 应用名称
4 spring.application.name: user-service
5 # 注册中心地址
6 eureka.client.service-url.defaultZone: http://127.0.0.1:10086/eureka
7
8 # DB 配置
9 spring.datasource.driver-class-name: com.mysql.cj.jdbc.Driver
10 spring.datasource.url: jdbc:mysql://127.0.0.1:3306/springcloud?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
11 spring.datasource.password: root
12 spring.datasource.username: root
13
14 # 扫描Mapper.xml文件
15 mybatis.mapper-locations: mapper/*Mapper.xml
16 # 扫描实体类
17 mybatis.type-aliases-package: com.itheima.domain
```

3. 创建完user-dev.yml配置文件之后，gitee中的仓库如下：



3.3 搭建配置中心微服务

实现步骤：

1. 创建配置中心SpringBoot项目config_server
2. 勾选Starter：配置中心，Eureka客户端
3. 在启动引导类上加@EnableConfigServer注解
4. 修改配置文件：端口，应用名称，注册中心地址，码云仓库地址
5. 启动测试，测试配置文件实时同步

实现过程：

1. 创建配置中心SpringBoot项目config_server

New Module

Project Metadata

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

Name:

Description:

Package:

2. 勾选Starter坐标依赖：配置中心starter，Eureka客户端starter

New Module

Dependencies

Spring Boot

Selected Dependencies

☐ Config Client

☒ Config Server

☐ Vault Configuration

☐ Apache Zookeeper Configuration

☐ Consul Configuration

Spring Cloud

Config Server

Central management for configuration via Git, SVN, or HashiCorp Vault.

Spring Cloud Config

Config Server

Spring Cloud Discovery

Eureka Discovery Client

3. 启动类：创建配置中心工程config_server的启动类ConfigServerApplication

```

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

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

```

4. 配置文件：创建配置中心工程config_server的配置文件application.yml
 - 注意：上述spring.cloud.config.server.git.uri是在码云创建的仓库地址

```

# 端口
server.port: 12000
# 应用名称
spring.application.name: config-server
# git仓库地址
spring.cloud.config.server.git.uri:
https://gitee.com/sangjiacun/sjc-spring-cloud-config.git
# 注册中心地址
eureka.client.service-url.defaultZone:
http://127.0.0.1:10086/eureka

```

5. 启动测试：启动eureka注册中心和配置中心；
 - 访问<http://localhost:12000/user-dev.yml>查看能否输出码云存储管理的user-dev.yml文件
 - 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


```
GET http://localhost:12000/user-dev.yml Send
body Cookies Headers (5) Test Results Status: 200 OK Time: 6.51 s Size: 727 B Sav
Pretty Raw Preview Visualize Text ↻
1 eureka:
2   client:
3     service-url:
4       defaultZone: http://eureka2:8762/eureka,http://eureka1:8761/eureka
5   instance:
6     instance-id: ${spring.cloud.client.ip-address}:9091
7     prefer-ip-address: true
8   mybatis:
9     mapper-locations: classpath:mapper/*Mapper.xml
10    type-aliases-package: com.sjc.entity
11  server:
12    port: 9091
13  spring:
14    application:
15      name: provider-service
16    datasource:
17      driver-class-name: com.mysql.cj.jdbc.Driver
18      password: root
19      url: jdbc:mysql://127.0.0.1:3306/itheima?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
20      username: root
21
```

3.4 服务去获取配置中心配置

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相当于项目启动的引导文件
 - application.yml文件是微服务的常规配置参数，变化比较频繁
- 搭配spring-cloud-config使application.yml的配置可以动态替换。

目标：改造user_service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. 在 provider_service 服务中，添加Config的starter依赖
2. 删除application.yml配置文件，新增bootstrap.yml配置文件
3. 配置bootstrap.yml配置文件：
 - 配置中心相关配置(配置文件前缀、后缀，仓库分支，是否开启配置中心)
 - 注册中心地址
4. 启动服务，测试效果

实现过程：

1. 添加依赖

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

2. 修改配置

- 删除 provider_service 工程的 application.yml 文件
- 创建 provider_service 工程 bootstrap.yml 配置文件，配置内容如下

```
# 注册中心地址
eureka.client.service-url.defaultZone:
http://127.0.0.1:10086/eureka

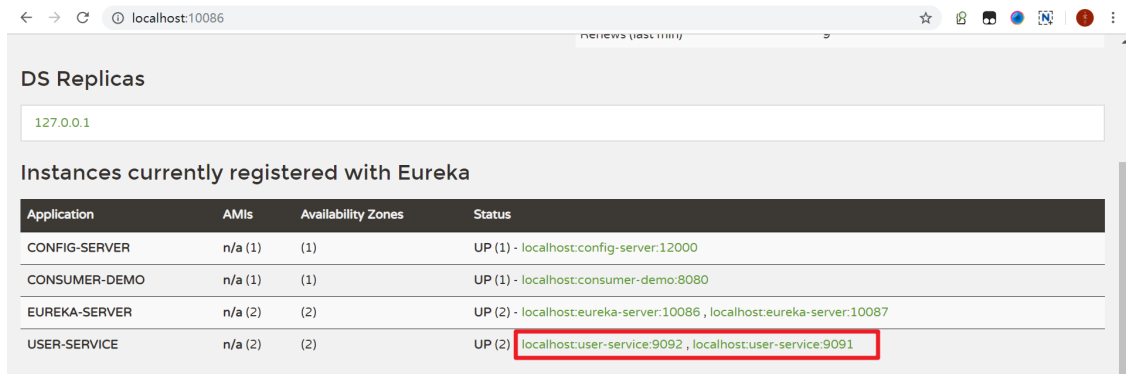
# 配置中心相关配置
# 使用配置中心
spring.cloud.config.discovery.enabled: true
# 配置中心服务id
spring.cloud.config.discovery.service-id: config-server
# 与远程仓库中的配置文件的application和profile保持一致, {application}-{profile}.yml
spring.cloud.config.name: user
spring.cloud.config.profile: dev
# 远程仓库中的分支保持一致
spring.cloud.config.label: master
```

3. 启动测试：

- 依次启动：注册中心、配置中心、用户中心 provider_service

```
Run: EurekaServerApplication-8761, EurekaServerApplication-8762, ProviderServiceApplication-9091, ProviderServiceApplication-9092, ConsumerServiceApplication, GatewayServerApplication, ConfigServerApplication
c.c.c.ConfigServicePropertySourceLocator: Fetching config from server at: http://127.0.0.1:8888/
c.c.c.ConfigServicePropertySourceLocator: Located environment: name=user, profiles=[dev], label=master, version=99ea80df68efcc9bee9d8867ed5c891a9, state=null
b.c.PropertySourceBootstrapConfiguration: Located property source: CompositePropertySource {name='configService', propertySources=[MapPropertySource {name='configClient'}, MapPropertySource {name='https://gitlab.com/samelecu/sic-spring-cloud-config-suit/user-dev.yml'}]}
com.xjc.ProviderServiceApplication: No active profile set, falling back to default profiles: default
e.s.cloud.context.scope.dynamicScope: BeanFactory [JdkSelfDestructingScope] Sub-Scope [JdkSelfDestructingScope]
org.springframework.beans.factory.support.DefaultListableBeanFactory: Bean 'org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration$$EnhancerBySpringCGLIB$$5d59f461' of type [org.springframework.cloud.autoconfigure.ConfigurationPropertiesRebinderAutoConfiguration$$EnhancerBySpringCGLIB$$5d59f461] is not eligible for getting instances from the BeanFactory
o.s.b.w.embedded.tomcat.TomcatWebServer: Tomcat initialized with port(s): 9091 (http)
```

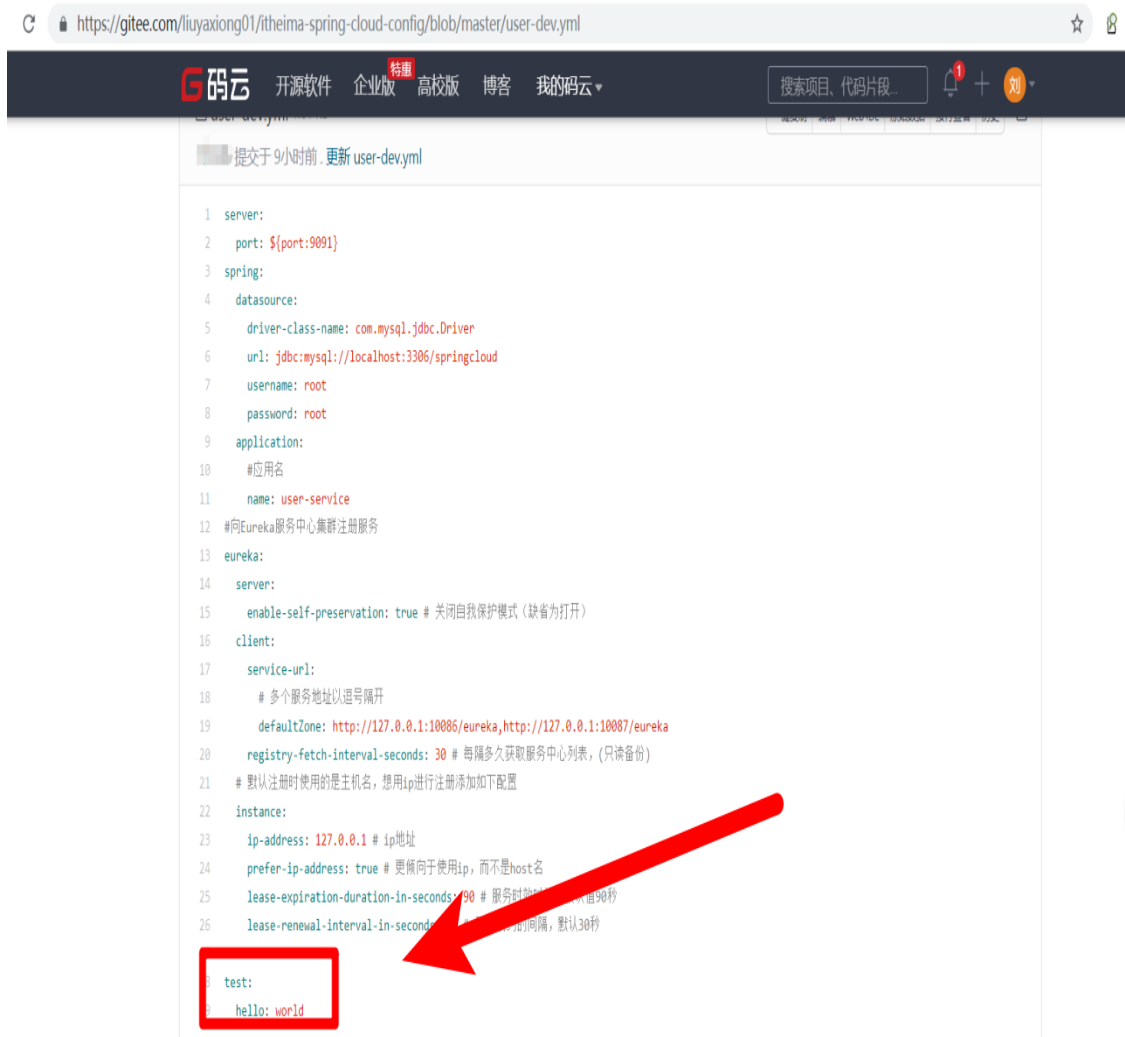
- 如果启动没报错，其实已经使用上配置中心内容了
- 可以在服务中心查看也可以检验 provider_service 的服务



3.5 配置中心存在的问题

复现问题步骤：

1. 修改远程Git配置
 - 修改在码云上的user-dev.yml文件，添加一个属性test.name



2. 修改UserController

```

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

    @Value("${server.port}")
    private String port;

    @Value("${test.hello}")
    private String name;

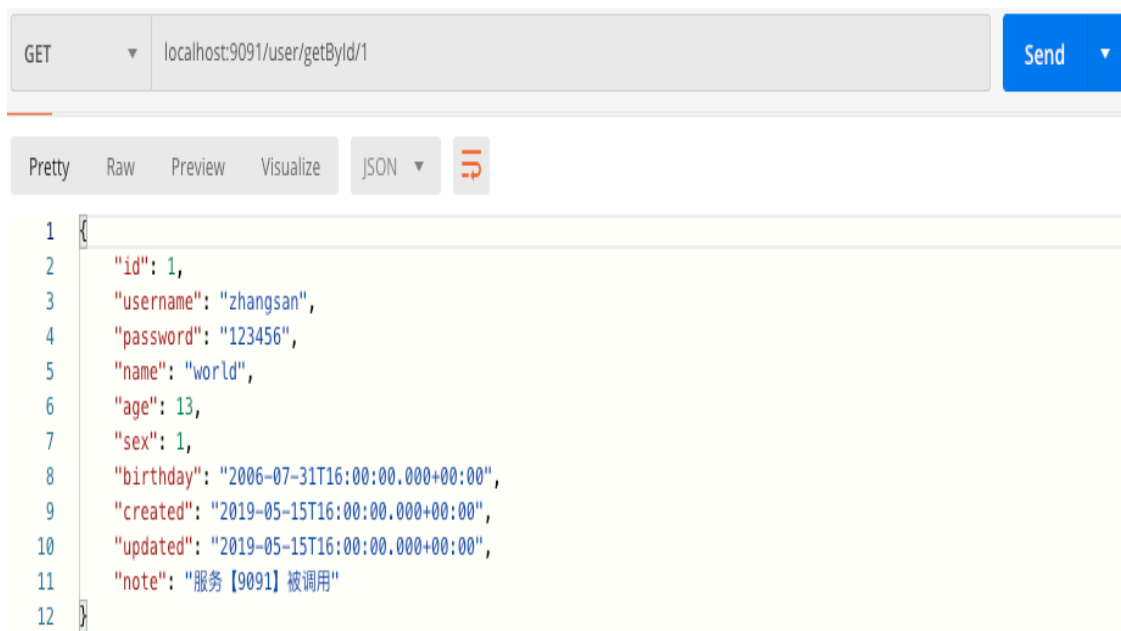
    @Autowired
    UserService userService;
    // 查询所有
    @RequestMapping("/findAll")
    public List<User> findAll() {
        return userService.findAll();
    }

    // 根据id查询
    @RequestMapping("/findById")
    public User findById(Integer id) {
        System.out.println("服务 [" + port + "] 被调用");
        User user = userService.findById(id);
        user.setNote("服务 [" + port + "] 被调用");
        user.setName(name);
        return user;
    }
}

```

2. 测试：

- 依次启动Eureka，配置中心，用户微服务；
- 访问用户微服务，查看输出内容。我们修改的user-dev.yml并没有发生立即发生变化。但是配置中心的配置文件内容发生了变化。



The screenshot shows a REST client interface. At the top, a GET request is made to `localhost:9091/user/getById/1`. Below the request bar, there are tabs for 'Pretty', 'Raw', 'Preview', and 'Visualize', with 'JSON' selected. The response is a JSON object displayed in a pretty-printed format:

```
1 {
2   "id": 1,
3   "username": "zhangsan",
4   "password": "123456",
5   "name": "world",
6   "age": 13,
7   "sex": 1,
8   "birthday": "2006-07-31T16:00:00.000+00:00",
9   "created": "2019-05-15T16:00:00.000+00:00",
10  "updated": "2019-05-15T16:00:00.000+00:00",
11  "note": "服务 [9091] 被调用"
12 }
```

结论：通过浏览器输出结果发现，我们对于Git仓库中的配置文件的修改，并没有及时更新到user-service微服务，只有重启用户微服务才能生效。

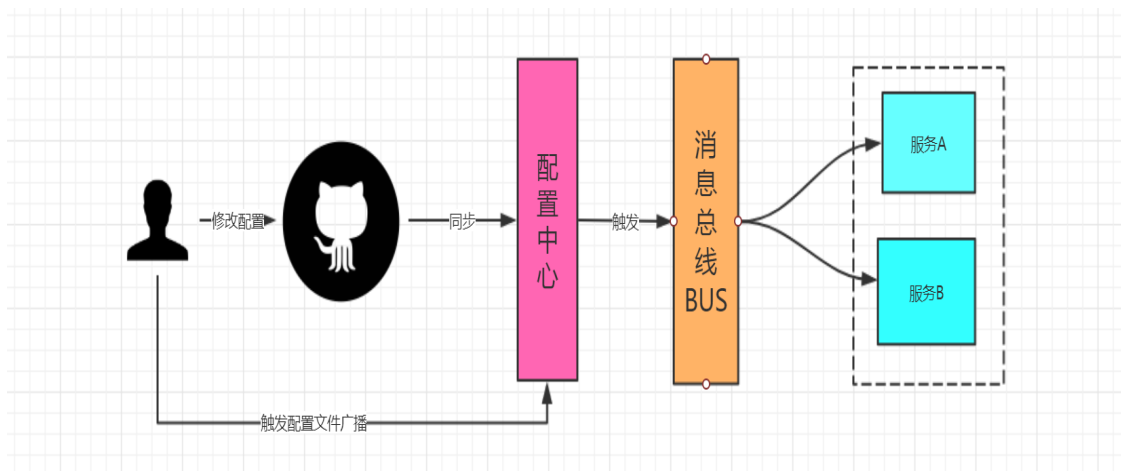
SpringCloud Bus，解决上述问题，实现配置自动更新。

四、消息总线 Spring Cloud Bus

4.1 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(消息队列)**RabbitMQ**和**Kafka**。

广播出去的配置文件服务会进行本地缓存。



4.2 整合案例

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

4.2.1 改造配置中心

改造步骤：

1. 在config_server中，加入Bus和RabbitMQ的依赖
2. 修改配置文件：RabbitMQ服务地址，触发配置文件更改接口

实现过程：

1. 在config_server项目中加入Bus相关依赖

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

2. 在config_server项目中修改application.yml

```
# RabbitMQ的服务地址
spring.rabbitmq.host: 127.0.0.1
spring.rabbitmq.port: 5672
spring.rabbitmq.username: guest
spring.rabbitmq.password: guest

# 触发配置文件广播的地址actuator的endpoint
management.endpoints.web.exposure.include: bus-refresh
```

4.2.2 改造生产者服务

改造步骤：

1. 在 provider-service 中，加入Bus和RabbitMQ的依赖
2. 修改配置文件：RabbitMQ服务地址
3. 在需要刷新配置的类上加@RefreshScope注解
4. 测试效果

实现过程：

1. 在用户微服务 provider_service 项目中加入Bus相关依赖

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

2. 修改 provider_service 项目的bootstrap.yml


```
# RabbitMQ的服务地址
spring.rabbitmq.host: 192.168.200.128
spring.rabbitmq.port: 5672
spring.rabbitmq.username: guest
spring.rabbitmq.password: guest
```

3. 改造用户微服务 provider_service 项目的UserController

```
@RestController
@RequestMapping("/user")
@RefreshScope //刷新配置
public class UserController {

    @Value("${server.port}")
    private String port;

    @Value("${test.hello}")
    private String name;

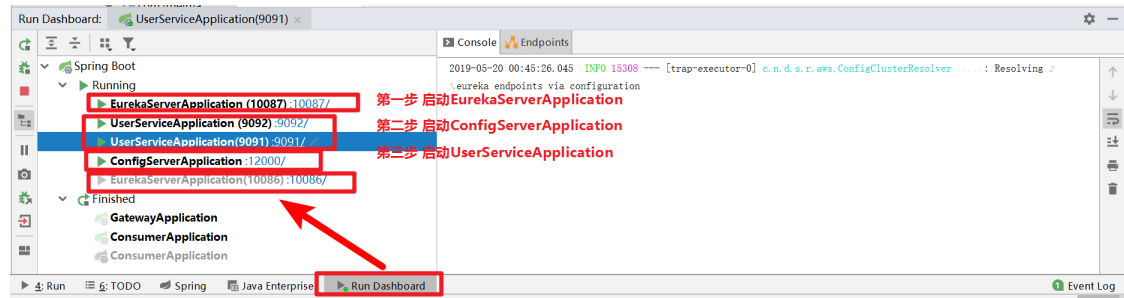
    @Autowired
    UserService userService;
    // 查询所有
    @RequestMapping("/findAll")
    public List<User> findAll() {
        return userService.findAll();
    }
    // 根据id查询
    @RequestMapping("/findById")
    public User findById(Integer id) {
        System.out.println("服务 [" + port + "] 被调用");
        User user = userService.findById(id);
        user.setNote("服务 [" + port + "] 被调用");
        user.setName(name);
        return user;
    }
}
```

4.3 测试

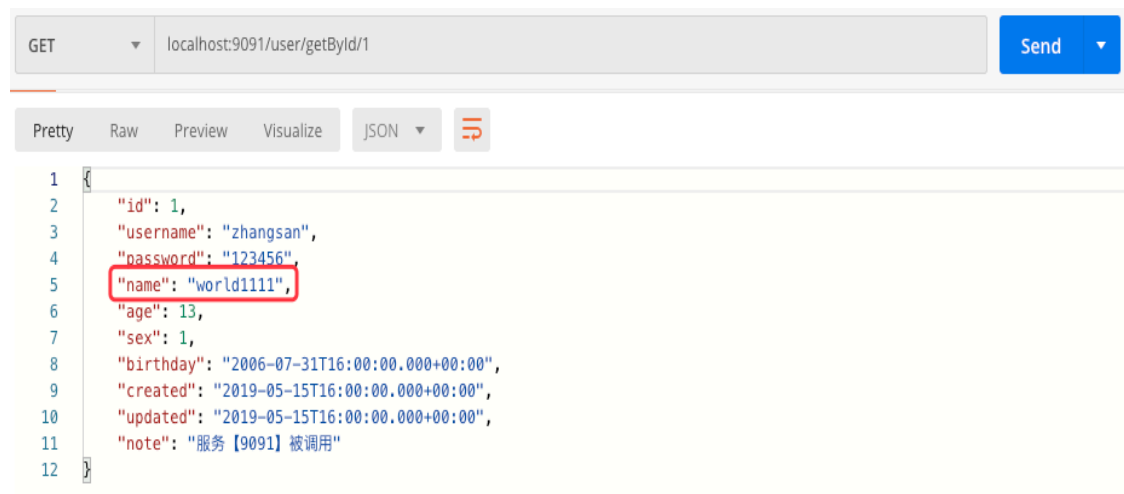
目标：当我们修改Git仓库的配置文件，用户微服务是否能够在不重启的情况下自动更新配置文件信息。

测试步骤：

1. 依次启动Eureka注册中心，配置中心，用户微服务



2. 访问用户微服务查看输出结果



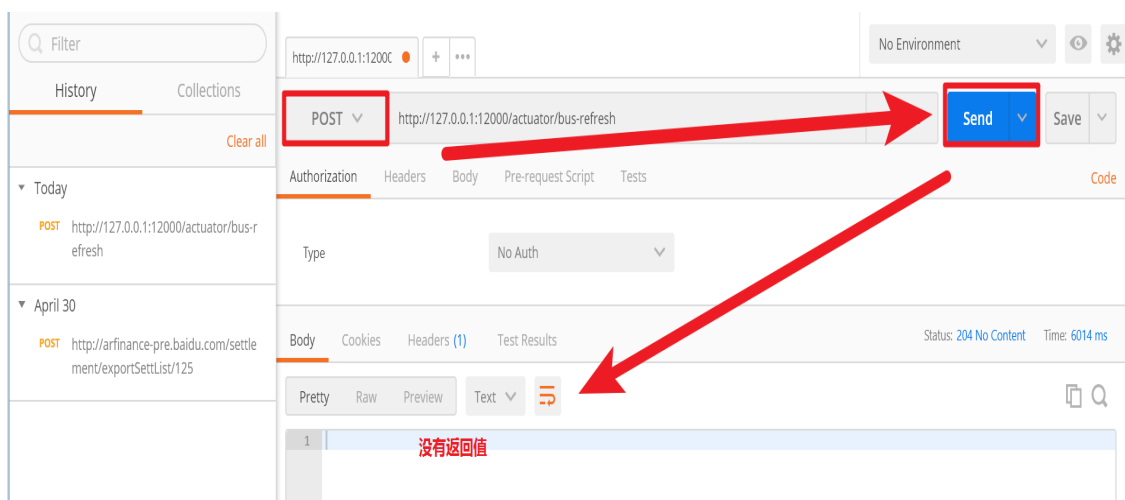
3. 修改Git仓库中配置文件内容

```

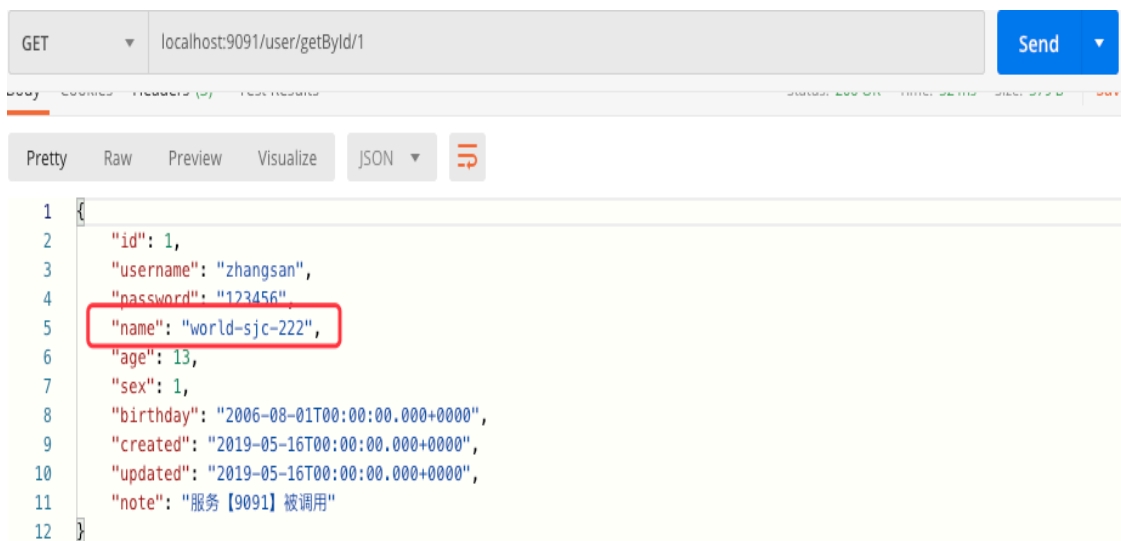
5  application:
6    name: provider-service
7  datasource:
8    url: jdbc:mysql://127.0.0.1:3306/itheima?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC
9    username: root
10   password: sang,1230
11   driver-class-name: com.mysql.cj.jdbc.Driver
12 #mybatis配置
13 mybatis:
14   #实体位置
15   type-aliases-package: com.sjc.domain
16   #mapper.xml位置
17   mapper-locations: classpath:mapper/*Mapper.xml
18
19 # 配置eurekaServer
20 eureka:
21   client:
22     service-url:
23       defaultZone: http://eureka1:8761/eureka,http://eureka2:8762/eureka
24   instance:
25     metadata-map:
26       my-metadata: zhangsan #自定义元数据
27     test.hello: world-sjc-222

```

4. 使用Postman工具发送POST请求，地址：<http://127.0.0.1:12000/actuator/bus-refresh>。刷新配置



5. 访问服务接口，浏览器查看输出结果



```
GET localhost:9091/user/getById/1 Send
Pretty Raw Preview Visualize JSON
1 {
2   "id": 1,
3   "username": "zhangsan",
4   "password": "123456",
5   "name": "world-sjc-222",
6   "age": 13,
7   "sex": 1,
8   "birthday": "2006-08-01T00:00:00.000+0000",
9   "created": "2019-05-16T00:00:00.000+0000",
10  "updated": "2019-05-16T00:00:00.000+0000",
11  "note": "服务 [9091] 被调用"
12 }
```

- 注意：所有的模块的spring cloud版本一致，Spring boot版本一致。且不建议最新版本
 - spring cloud版本：Greenwich.SR2
 - spring boot版本：2.1.9.RELEASE

说明：

- Postman或者REStClient是一个可以模拟浏览器发生各种请求的工具
- 请求地址 <http://127.0.0.1:12000/actuator/bus-refresh> 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
- <http://127.0.0.1:12000/actuator/bus-refresh>

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