

0. 学习目标

- 能够使用Feign进行远程调用
- 能够搭建Spring Cloud Gateway网关服务
- 能够配置Spring Cloud Gateway路由过滤器
- 能够编写Spring Cloud Gateway全局过滤器
- 能够搭建Spring Cloud Config配置中心服务
- 能够使用Spring Cloud Bus实时更新配置

1. Feign

在前面的学习中，使用了Ribbon的负载均衡功能，大大简化了远程调用时的代码：

```
1 String url = "http://user-service/user/" + id;  
2 User user = this.restTemplate.getForObject(url, 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也叫伪装：

Feign可以把Rest的请求进行隐藏，伪装成类似SpringMVC的Controller一样。你不用再自己拼接url，拼接参数等等操作，一切都交给Feign去做。

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

1.2. 快速入门

1.2.1. 导入依赖

在 `consumer-demo` 项目的 `pom.xml` 文件中添加如下依赖

```
1 <dependency>  
2     <groupId>org.springframework.cloud</groupId>  
3     <artifactId>spring-cloud-starter-openfeign</artifactId>  
4 </dependency>
```

1.2.2. Feign的客户端

在 `consumer-demo` 中编写如下Feign客户端接口类:

```
1 package com.itheima.consumer.client;
2
3 import com.itheima.consumer.pojo.User;
4 import org.springframework.cloud.openfeign.FeignClient;
5 import org.springframework.web.bind.annotation.GetMapping;
6 import org.springframework.web.bind.annotation.PathVariable;
7
8 @FeignClient("user-service")
9 public interface UserClient {
10
11     @GetMapping("/user/{id}")
12     User queryById(@PathVariable("id") Long id);
13 }
```

- 首先这是一个接口, Feign会通过动态代理, 帮我们生成实现类。这点跟mybatis的mapper很像
- `@FeignClient`, 声明这是一个Feign客户端, 同时通过 `value` 属性指定服务名称
- 接口中的定义方法, 完全采用SpringMVC的注解, Feign会根据注解帮我们生成URL, 并访问获取结果
- `@GetMapping`中的`/user`, 请不要忘记; 因为Feign需要拼接可访问的地址

编写新的控制器类 `ConsumerFeignController`, 使用`UserClient`访问:

```
1 package com.itheima.consumer.controller;
2
3 import com.itheima.consumer.client.UserClient;
4 import com.itheima.consumer.pojo.User;
5 import lombok.extern.slf4j.Slf4j;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11
12 @RestController
13 @RequestMapping("/cf")
14 public class ConsumerFeignController {
15
16     @Autowired
17     private UserClient userClient;
18
19     @GetMapping("/{id}")
20     public User queryById(@PathVariable Long id){
21         return userClient.queryById(id);
22     }
23 }
24
```

1.2.3. 开启Feign功能

在 `ConsumerApplication` 启动类上, 添加注解, 开启Feign功能

```

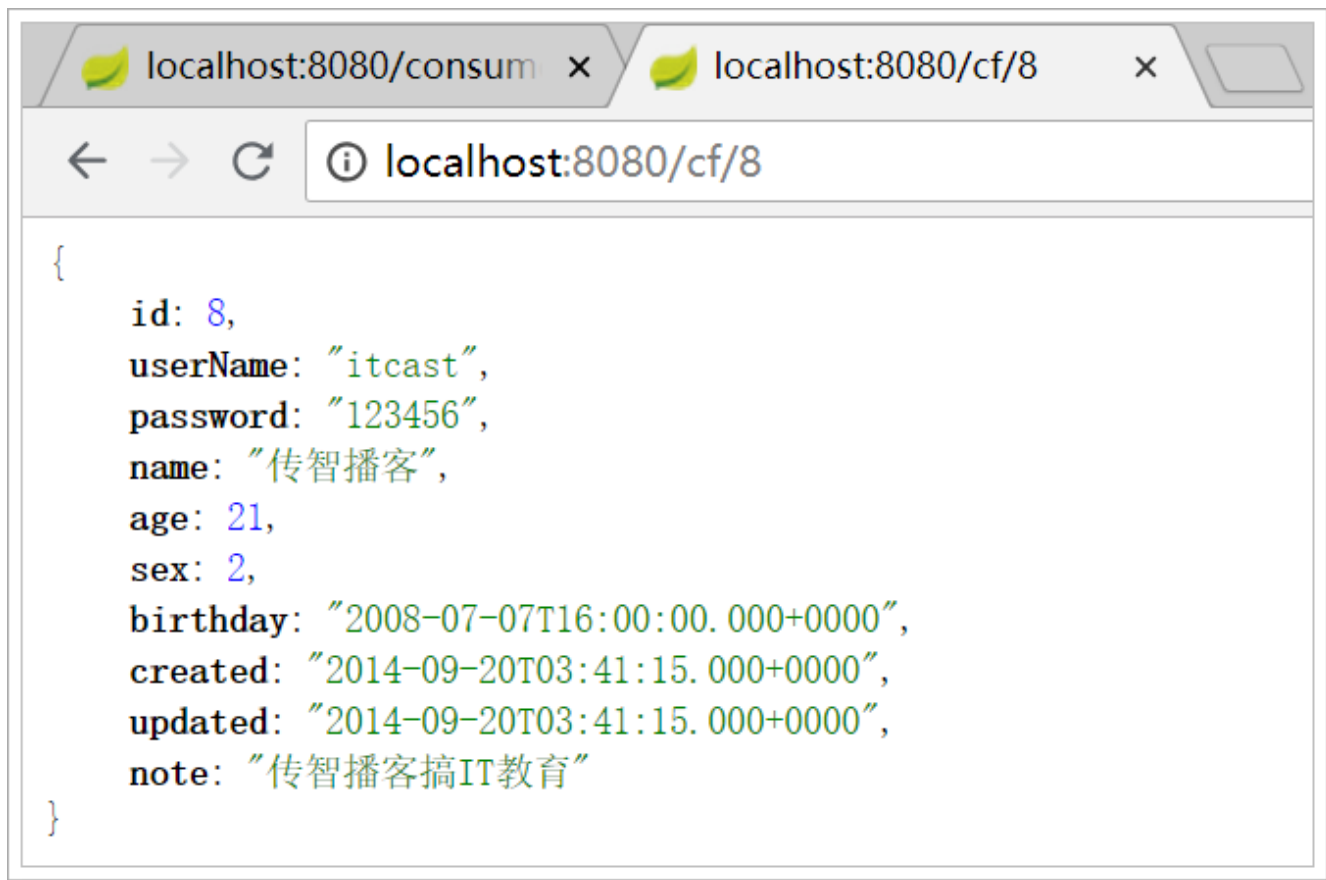
1  package com.itheima.consumer;
2
3  import org.springframework.boot.SpringApplication;
4  import org.springframework.cloud.client.SpringCloudApplication;
5  import org.springframework.cloud.client.loadbalancer.LoadBalanced;
6  import org.springframework.cloud.openfeign.EnableFeignClients;
7  import org.springframework.context.annotation.Bean;
8  import org.springframework.web.client.RestTemplate;
9
10 /*@SpringBootApplication
11 @EnableDiscoveryClient
12 @EnableCircuitBreaker*/
13 @SpringCloudApplication
14 @EnableFeignClients//开启Feign功能
15 public class ConsumerApplication {
16     public static void main(String[] args) {
17         SpringApplication.run(ConsumerApplication.class, args);
18     }
19
20     @Bean
21     @LoadBalanced
22     public RestTemplate restTemplate(){
23         return new RestTemplate();
24     }
25 }
26

```

Feign中已经自动集成了Ribbon负载均衡，因此不需要自己定义RestTemplate进行负载均衡的配置。

1.2.4. 启动测试

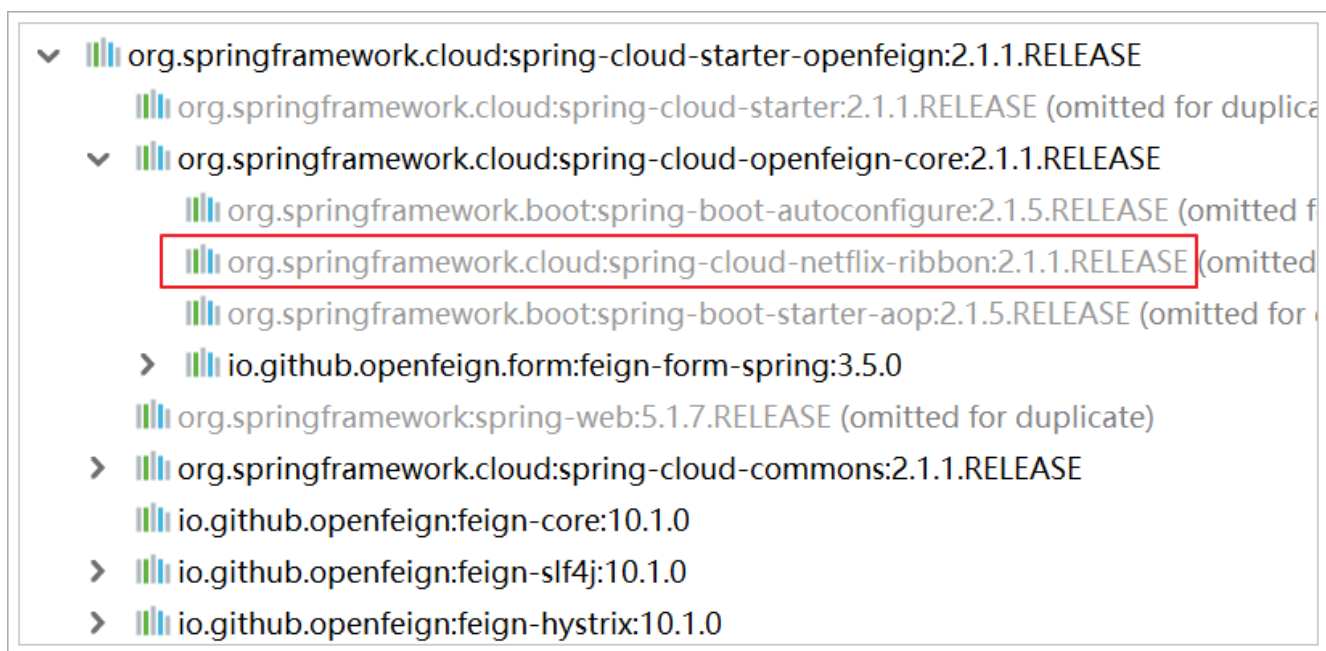
访问接口: <http://localhost:8080/cf/2>



正常获取到了结果。

1.3. 负载均衡

Feign中本身已经集成了Ribbon依赖和自动配置：



因此不需要额外引入依赖，也不需要再注册 `RestTemplate` 对象。

Feign内置的ribbon默认设置了请求超时时长，默认是1000，我们可以通过手动配置来修改这个超时时长：

```
1 ribbon:
2   ReadTimeout: 2000 # 读取超时时长
3   ConnectTimeout: 1000 # 建立链接的超时时长
```

因为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，可以添加配置：

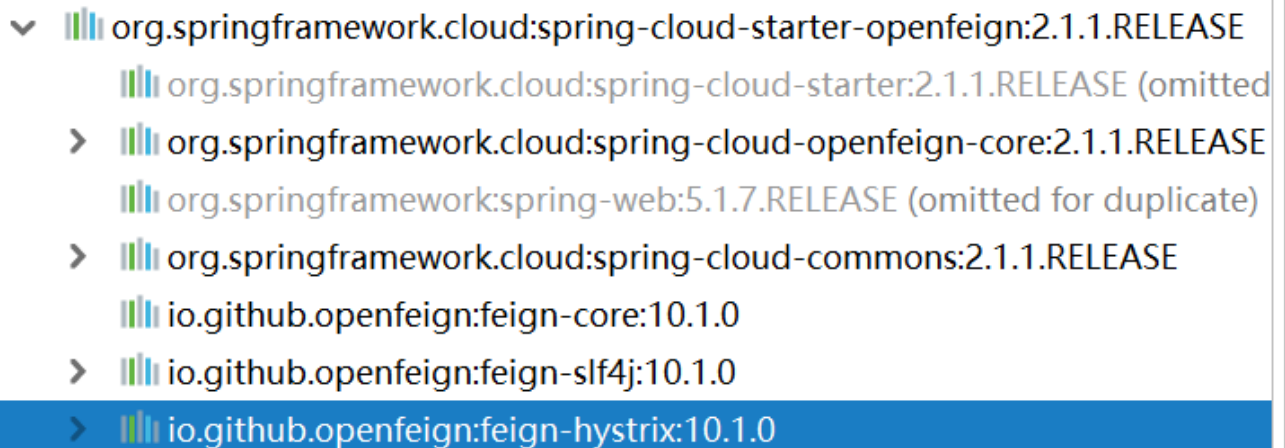
修改 `consumer-demo\src\main\resources\application.yml` 添加如下配置：

```
1 ribbon:
2   ConnectTimeout: 1000 # 连接超时时长
3   ReadTimeout: 2000 # 数据通信超时时长
4   MaxAutoRetries: 0 # 当前服务器的重试次数
5   MaxAutoRetriesNextServer: 0 # 重试多少次服务
6   okToRetryOnAllOperations: false # 是否对所有的请求方式都重试
```

重新给UserService的方法设置上线程沉睡时间2秒可以测试上述配置

1.4. Hystrix支持(了解)

Feign默认也有对Hystrix的集成：



```
▼ org.springframework.cloud:spring-cloud-starter-openfeign:2.1.1.RELEASE
  org.springframework.cloud:spring-cloud-starter:2.1.1.RELEASE (omitted)
  > org.springframework.cloud:spring-cloud-openfeign-core:2.1.1.RELEASE
    org.springframework:spring-web:5.1.7.RELEASE (omitted for duplicate)
  > org.springframework.cloud:spring-cloud-commons:2.1.1.RELEASE
    io.github.openfeign:feign-core:10.1.0
  > io.github.openfeign:feign-slf4j:10.1.0
  > io.github.openfeign:feign-hystrix:10.1.0
```

只不过，默认情况下是关闭的。需要通过下面的参数来开启；

修改 `consumer-demo\src\main\resources\application.yml` 添加如下配置：

```
1 feign:
2   hystrix:
3     enabled: true # 开启Feign的熔断功能
```

但是，Feign中的Fallback配置不像Ribbon中那样简单了。

1) 首先，要定义一个类，实现刚才编写的UserFeignClient，作为fallback的处理类

```

1 package com.itheima.consumer.client.fallback;
2
3 import com.itheima.consumer.client.UserClient;
4 import com.itheima.consumer.pojo.User;
5 import org.springframework.stereotype.Component;
6
7 @Component
8 public class UserClientFallback implements UserClient {
9     @Override
10    public User queryById(Long id) {
11        User user = new User();
12        user.setId(id);
13        user.setName("用户异常");
14        return user;
15    }
16 }
17

```

2) 然后在UserFeignClient中，指定刚才编写的实现类

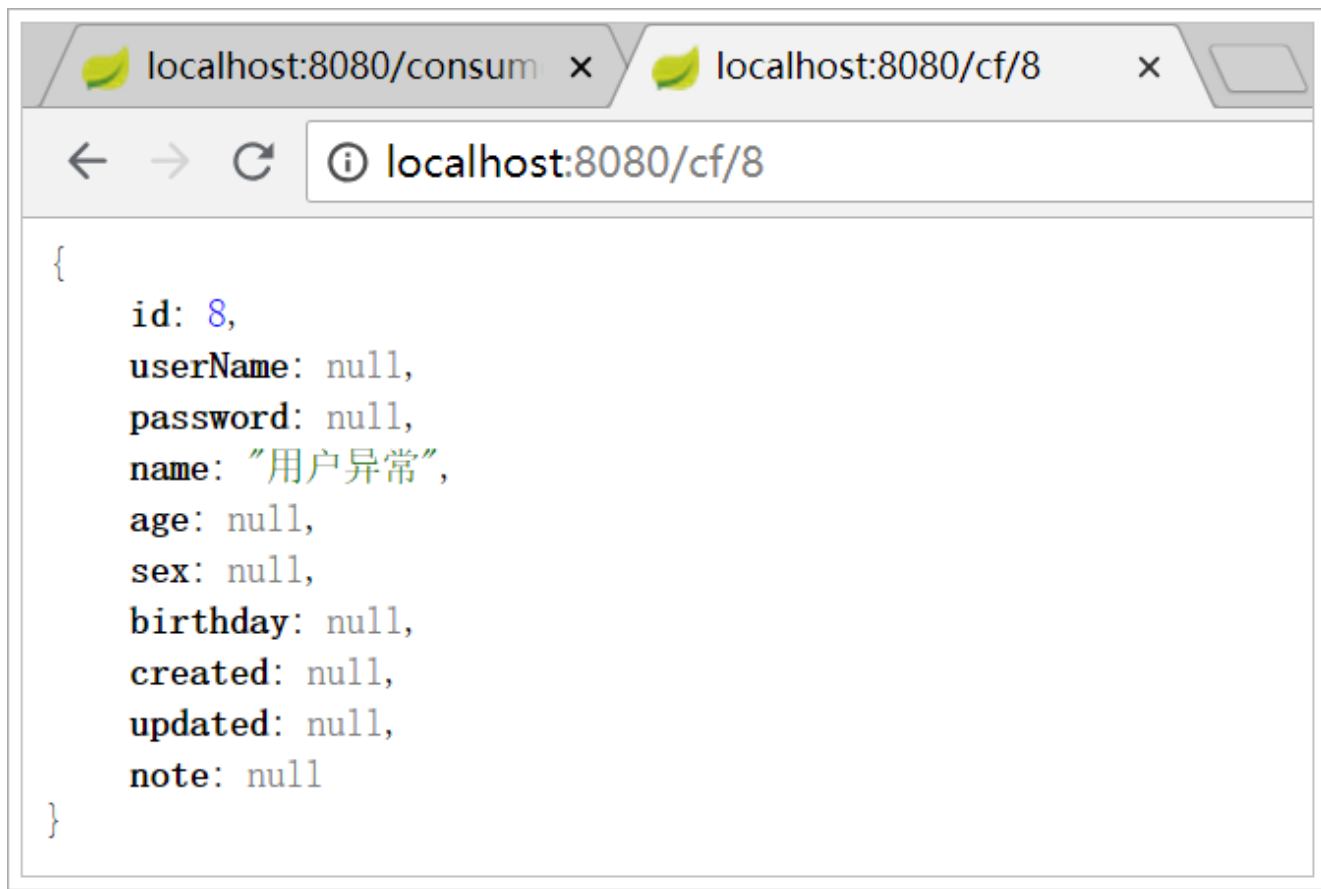
```

1 @FeignClient(value = "user-service", fallback = UserClientFallback.class)
2 public interface UserClient {
3
4     @GetMapping("/user/{id}")
5     User queryById(@PathVariable("id") Long id);
6 }
7

```

3) 重启测试

重启启动 consumer-demo 并关闭 user-service 服务，然后在页面访问：<http://localhost:8080/cf/8>



1.5. 请求压缩(了解)

Spring Cloud Feign 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。通过下面的参数即可开启请求与响应的压缩功能：

```
1 feign:
2   compression:
3     request:
4       enabled: true # 开启请求压缩
5     response:
6       enabled: true # 开启响应压缩
```

同时，我们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：

```
1 feign:
2   compression:
3     request:
4       enabled: true # 开启请求压缩
5       mime-types: text/html,application/xml,application/json # 设置压缩的数据类型
6       min-request-size: 2048 # 设置触发压缩的大小下限
```

注：上面的数据类型、压缩大小下限均为默认值。

1.6. 日志级别(了解)

前面讲过，通过 `logging.level.xx=debug` 来设置日志级别。然而这个对Feign客户端而言不会产生效果。因为 `@FeignClient` 注解修改的客户端在被代理时，都会创建一个新的Feign.Logger实例。我们需要额外指定这个日志的级别才可以。

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

修改 `consumer-demo\src\main\resources\application.yml` 添加如下配置：

```
1 logging:
2   level:
3     com.itheima: debug
```

2) 在 `consumer-demo` 编写FeignConfig配置类，定义日志级别

```
1 package com.itheima.consumer.config;
2
3 import feign.Logger;
4 import org.springframework.context.annotation.Bean;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 public class FeignConfig {
9
10     @Bean
11     Logger.Level feignLoggerLevel(){
12         //记录所有请求和响应的明细，包括头信息、请求体、元数据
13         return Logger.Level.FULL;
14     }
15 }
16
```

这里指定的Level级别是FULL，Feign支持4种级别：

- NONE：不记录任何日志信息，这是默认值。
- 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的基础上，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。

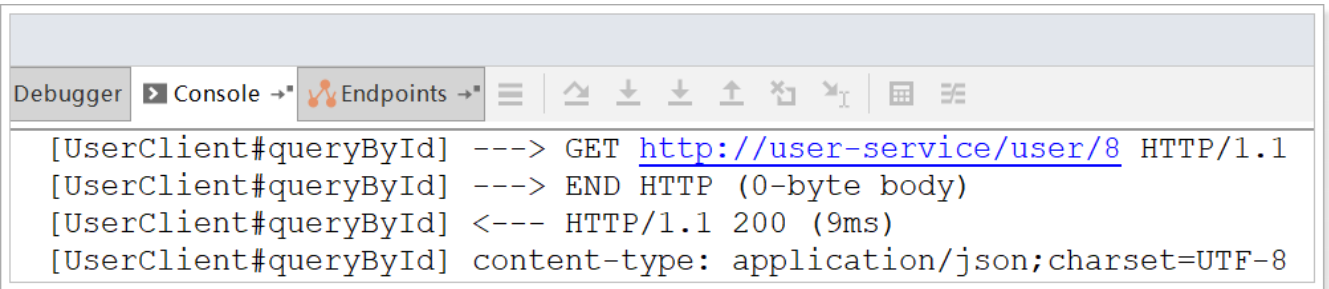
3) 在 `consumer-demo` 的 `UserClient` 接口类上的 `@FeignClient` 注解中指定配置类：

```
1 package com.itheima.consumer.client;
2
3 import com.itheima.consumer.client.fallback.UserClientFallback;
4 import com.itheima.consumer.config.FeignConfig;
5 import com.itheima.consumer.pojo.User;
6 import org.springframework.cloud.openfeign.FeignClient;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
```



```
9
10 @FeignClient(value = "user-service", fallback = UserClientFallback.class,
11 configuration = FeignConfig.class)
12 public interface UserClient {
13
14     @GetMapping("/user/{id}")
15     User queryById(@PathVariable Long id);
16 }
```

4) 重启项目，访问：<http://localhost:8080/cf/8>；即可看到每次访问的日志：



The screenshot shows a web browser's developer console with the 'Endpoints' tab selected. It displays the following log entries:

```
[UserClient#queryById] ---> GET http://user-service/user/8 HTTP/1.1
[UserClient#queryById] ---> END HTTP (0-byte body)
[UserClient#queryById] <--- HTTP/1.1 200 (9ms)
[UserClient#queryById] content-type: application/json;charset=UTF-8
```

2. Spring Cloud Gateway网关

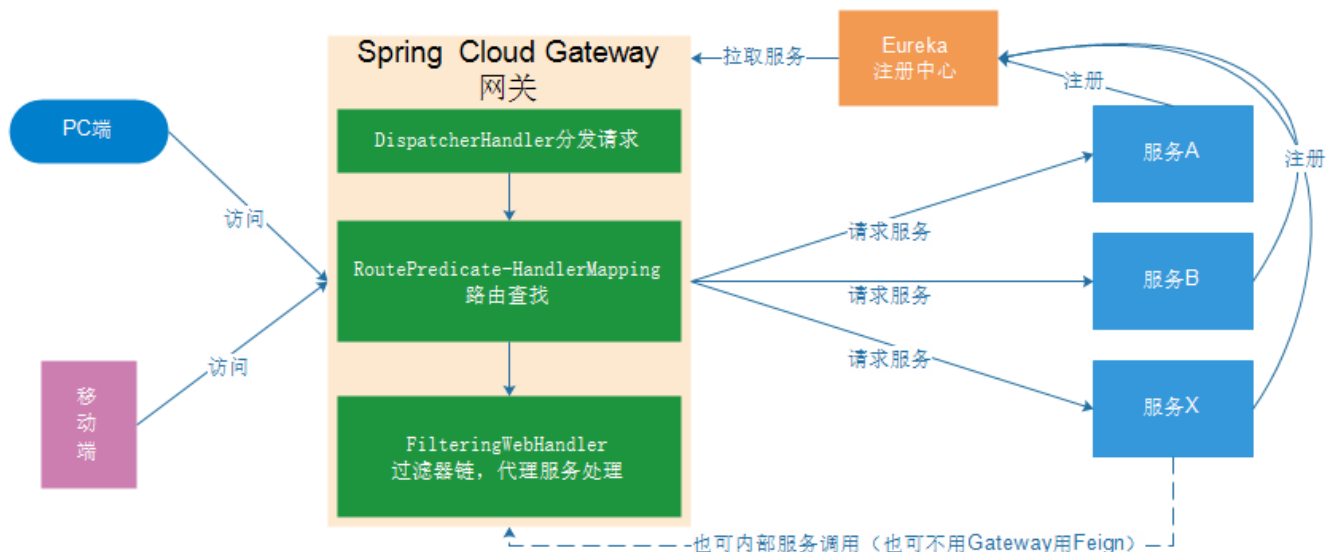
2.1. 简介

- Spring Cloud Gateway是Spring官网基于Spring 5.0、Spring Boot 2.0、Project Reactor等技术开发的网关服务。
- Spring Cloud Gateway基于Filter链提供网关基本功能：安全、监控 / 埋点、限流等。
- Spring Cloud Gateway为微服务架构提供简单、有效且统一的API路由管理方式。
- Spring Cloud Gateway是替代Netflix Zuul的一套解决方案。

Spring Cloud Gateway组件的核心是一系列的过滤器，通过这些过滤器可以将客户端发送的请求转发（路由）到对应的微服务。Spring Cloud Gateway是加在整个微服务最前沿的防火墙和代理器，隐藏微服务结点IP端口信息，从而加强安全保护。Spring Cloud Gateway本身也是一个微服务，需要注册到Eureka服务注册中心。

网关的核心功能是：**过滤和路由**

2.2. Gateway加入后的架构



- 不管是来自于客户端（PC或移动端）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，然后再由网关来实现鉴权、动态路由等等操作。Gateway就是我们服务的统一入口。

2.3. 核心概念

- **路由 (route)** 路由信息的组成：由一个ID、一个目的URL、一组断言工厂、一组Filter组成。如果路由断言为真，说明请求URL和配置路由匹配。
- **断言 (Predicate)** Spring Cloud Gateway中的断言函数输入类型是Spring 5.0框架中的ServerWebExchange。Spring Cloud Gateway的断言函数允许开发者去定义匹配来自于Http Request中的任何信息比如请求头和参数。
- **过滤器 (Filter)** 一个标准的Spring WebFilter。Spring Cloud Gateway中的Filter分为两种类型的Filter，分别是Gateway Filter和Global Filter。过滤器Filter将会对请求和响应进行修改处理。

2.4. 快速入门

2.4.1. 新建工程

填写基本信息：

New Module

Add as module to

com.itheima:heima-springcloud:1.0-SNAPSHOT

...

Parent

com.itheima:heima-springcloud:1.0-SNAPSHOT

...

GroupId

com.itheima

☐ Inherit

ArtifactId

heima-gateway

Version

1.0-SNAPSHOT

☒ Inherit

New Module

Module name:

heima-gateway

Content root:

D:\workspaces\springcloud\heima-springcloud\heima-gateway

Module file location:

D:\workspaces\springcloud\heima-springcloud\heima-gateway

打开 `heima-springcloud\heima-gateway\pom.xml` 文件修改为如下:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
5     <parent>
6         <artifactId>heima-springcloud</artifactId>
7         <groupId>com.itheima</groupId>
8         <version>1.0-SNAPSHOT</version>
9     </parent>
10    <modelVersion>4.0.0</modelVersion>
11
12    <groupId>com.itheima</groupId>
13    <artifactId>heima-gateway</artifactId>
14
15    <dependencies>
16        <dependency>
17            <groupId>org.springframework.cloud</groupId>
18            <artifactId>spring-cloud-starter-gateway</artifactId>
19        </dependency>
20        <dependency>
21            <groupId>org.springframework.cloud</groupId>
22            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
23        </dependency>
24    </dependencies>
25
26 </project>
```

2.4.2. 编写启动类

在heima-gateway中创建 `com.itheima.gateway.GatewayApplication` 启动类

```
1 package com.itheima.gateway;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
6
7 @SpringBootApplication
8 @EnableDiscoveryClient
9 public class GatewayApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(GatewayApplication.class, args);
12     }
13 }
14
```

2.4.2. 编写配置

创建 heima-gateway\src\main\resources\application.yml 文件，内容如下：

```
1 server:
2   port: 10010
3 spring:
4   application:
5     name: api-gateway
6
7 eureka:
8   client:
9     service-url:
10      defaultZone: http://127.0.0.1:10086/eureka
11   instance:
12     prefer-ip-address: true
```

2.4.4. 编写路由规则

需要用网关来代理 user-service 服务，先看一下控制面板中的服务状态：

Instances currently registered with Eureka

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

- ip为：127.0.0.1
- 端口为：9091

修改 heima-gateway\src\main\resources\application.yml 文件为：

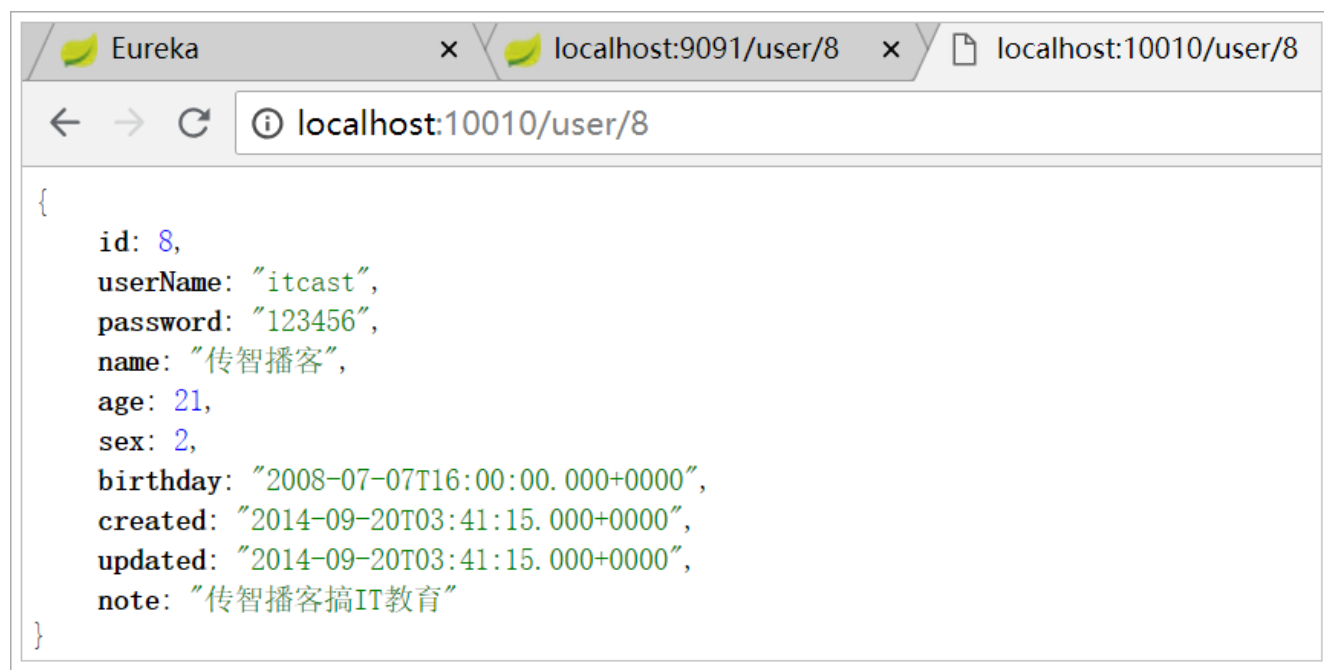
```
1 server:
2   port: 10010
3 spring:
4   application:
5     name: api-gateway
6   cloud:
7     gateway:
8       routes:
9         # 路由id, 可以随意写
10        - id: user-service-route
11          # 代理的服务地址
12          uri: http://127.0.0.1:9091
13          # 路由断言, 可以配置映射路径
14          predicates:
15            - Path=/user/**
16 eureka:
17   client:
18     service-url:
19      defaultZone: http://127.0.0.1:10086/eureka
20   instance:
21     prefer-ip-address: true
```

将符合 `Path` 规则的一切请求，都代理到 `uri` 参数指定的地址

本例中，我们将路径中包含有 `/user/**` 开头的请求，代理到<http://127.0.0.1:9091>

2.4.5. 启动测试

访问的路径中需要加上配置规则的映射路径，我们访问：<http://localhost:10010/user/8>



2.5. 面向服务的路由

在刚才的路由规则中，把路径对应的服务地址写死了！如果同一服务有多个实例的话，这样做显然不合理。

应该根据服务的名称，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！

2.5.1. 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 `heima-gateway\src\main\resources\application.yml` 文件如下：

```
1  server:
2    port: 10010
3  spring:
4    application:
5      name: api-gateway
6  cloud:
7    gateway:
8      routes:
9        # 路由id, 可以随意写
10       - id: user-service-route
11         # 代理的服务地址; lb表示从eureka中获取具体服务
```

```

12     uri: lb://user-service
13     # 路由断言, 可以配置映射路径
14     predicates:
15         - Path=/user/**
16 eureka:
17     client:
18         service-url:
19             defaultZone: http://127.0.0.1:10086/eureka
20     instance:
21         prefer-ip-address: true

```

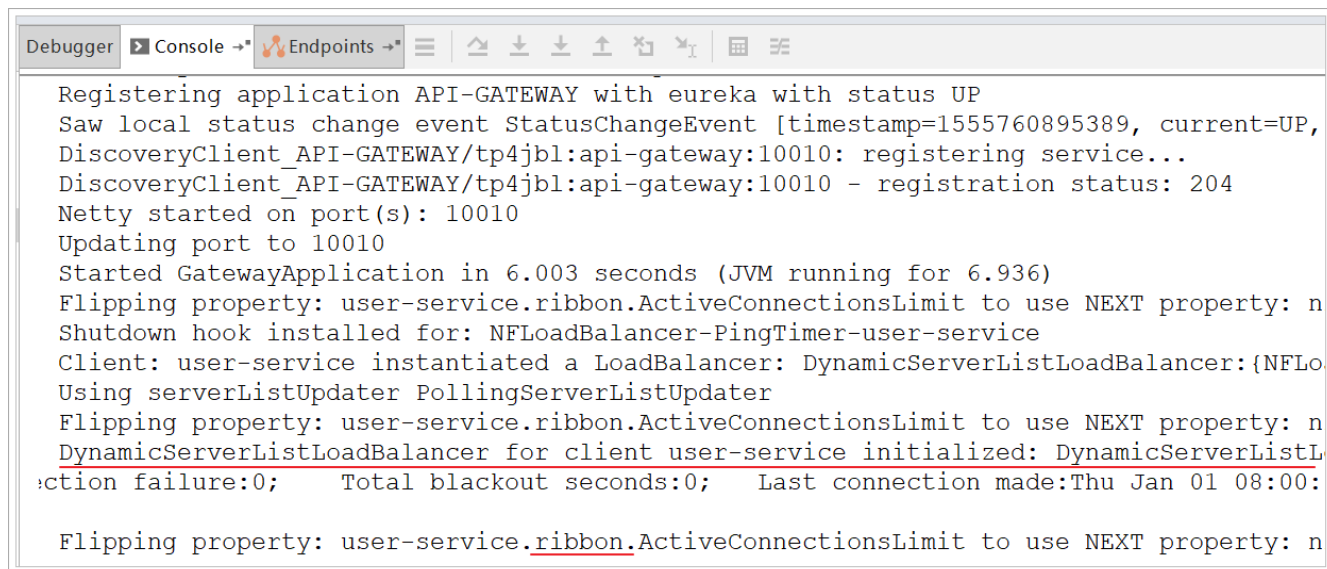
路由配置中uri所用的协议为lb时（以uri: lb://user-service为例），gateway将使用 LoadBalancerClient把 user-service通过eureka解析为实际的主机和端口，并进行ribbon负载均衡。

2.5.2. 启动测试

再次启动 heima-gateway，这次gateway进行代理时，会利用Ribbon进行负载均衡访问：

<http://localhost:10010/user/8>

日志中可以看到使用了负载均衡器：



```

Debugger Console Endpoints
Registering application API-GATEWAY with eureka with status UP
Saw local status change event StatusChangeEvent [timestamp=1555760895389, current=UP,
DiscoveryClient_API-GATEWAY/tp4jbl:api-gateway:10010: registering service...
DiscoveryClient_API-GATEWAY/tp4jbl:api-gateway:10010 - registration status: 204
Netty started on port(s): 10010
Updating port to 10010
Started GatewayApplication in 6.003 seconds (JVM running for 6.936)
Flipping property: user-service.ribbon.ActiveConnectionsLimit to use NEXT property: n
Shutdown hook installed for: NFLoadBalancer-PingTimer-user-service
Client: user-service instantiated a LoadBalancer: DynamicServerListLoadBalancer:{NFLo
Using serverListUpdater PollingServerListUpdater
Flipping property: user-service.ribbon.ActiveConnectionsLimit to use NEXT property: n
DynamicServerListLoadBalancer for client user-service initialized: DynamicServerListL
ction failure:0; Total blackout seconds:0; Last connection made:Thu Jan 01 08:00:
Flipping property: user-service.ribbon.ActiveConnectionsLimit to use NEXT property: n

```

2.6. 路由前缀

2.6.1. 添加前缀

在gateway中可以通过配置路由的过滤器PrefixPath，实现映射路径中地址的添加；

修改 heima-gateway\src\main\resources\application.yml 文件：

```

1 server:
2     port: 10010
3 spring:
4     application:
5         name: api-gateway

```

```

6   cloud:
7     gateway:
8       routes:
9         # 路由id, 可以随意写
10        - id: user-service-route
11          # 代理的服务地址; lb表示从eureka中获取具体服务
12          uri: lb://user-service
13          # 路由断言, 可以配置映射路径
14          predicates:
15            - Path=/**
16          filters:
17            # 添加请求路径的前缀
18            - PrefixPath=/user
19  eureka:
20    client:
21      service-url:
22        defaultZone: http://127.0.0.1:10086/eureka
23    instance:
24      prefer-ip-address: true

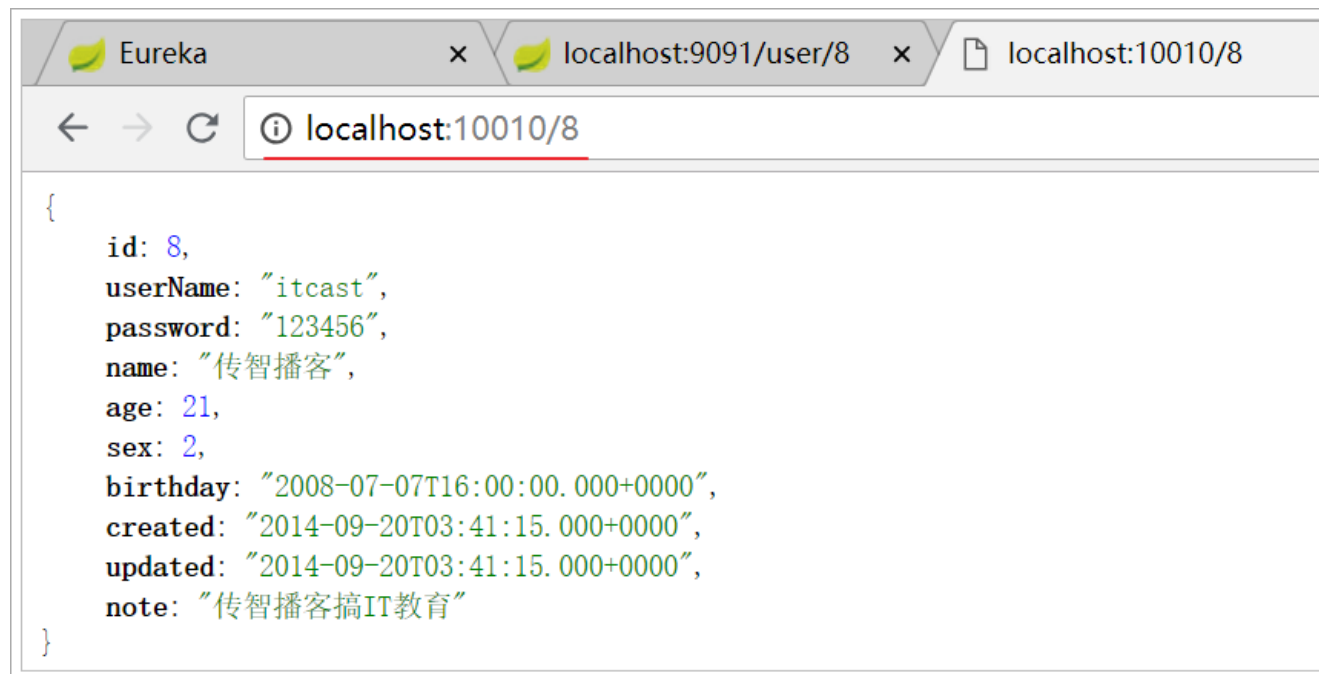
```

通过 `PrefixPath=xxx` 来指定了路由要添加的前缀。

也就是：

- `PrefixPath=/user` <http://localhost:10010/8> --> <http://localhost:9091/user/8>
- `PrefixPath=/user/abc` <http://localhost:10010/8> --> <http://localhost:9091/user/abc/8>

以此类推。



2.6.2. 去除前缀

在gateway中可以通过配置路由的过滤器`StripPrefix`，实现映射路径中地址的去除；

修改 heima-gateway\src\main\resources\application.yml 文件:

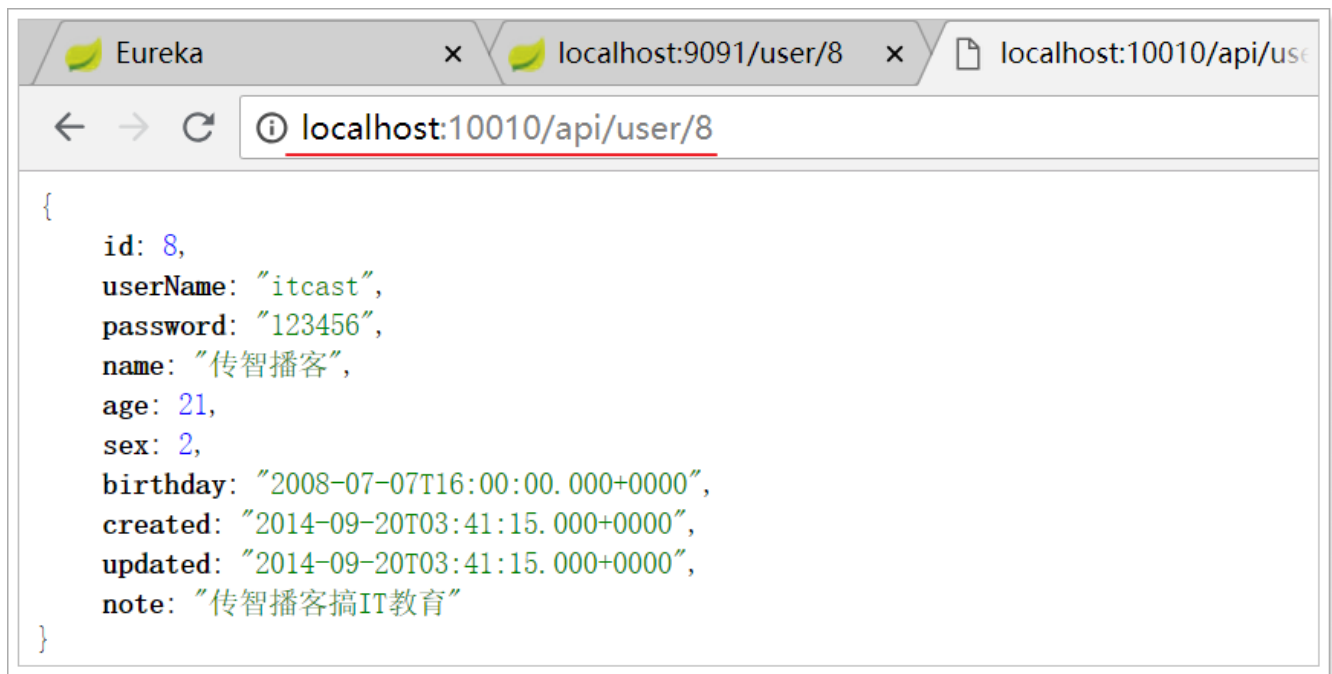
```
1  server:
2    port: 10010
3  spring:
4    application:
5      name: api-gateway
6  cloud:
7    gateway:
8      routes:
9        # 路由id, 可以随意写
10       - id: user-service-route
11         # 代理的服务地址; lb表示从eureka中获取具体服务
12         uri: lb://user-service
13         # 路由断言, 可以配置映射路径
14         predicates:
15           - Path=/api/user/**
16         filters:
17           # 表示过滤1个路径, 2表示两个路径, 以此类推
18           - StripPrefix=1
19  eureka:
20    client:
21      service-url:
22        defaultZone: http://127.0.0.1:10086/eureka
23    instance:
24      prefer-ip-address: true
```

通过 StripPrefix=1 来指定了路由要去掉的前缀个数。如: 路径 /api/user/1 将会被代理到 /user/1。

也就是:

- StripPrefix=1 <http://localhost:10010/api/user/8> --> <http://localhost:9091/user/8>
- StripPrefix=2 <http://localhost:10010/api/user/8> --> <http://localhost:9091/8>

以此类推。



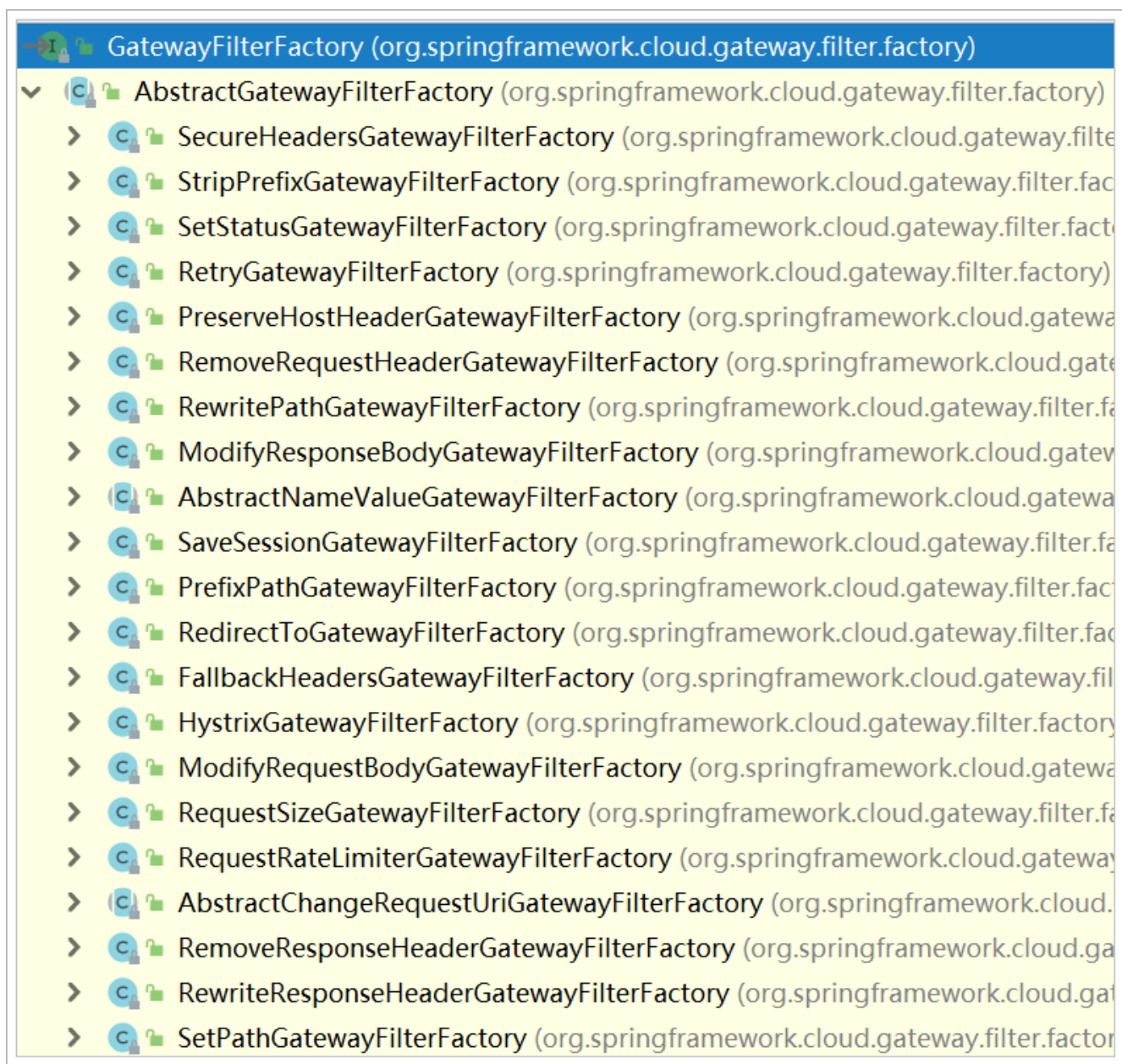
2.7. 过滤器

2.7.1. 简介

Gateway作为网关的其中一个重要功能，就是实现请求的鉴权。而这个动作往往是通过网关提供的过滤器来实现的。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添加参数
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



详细的说明在[官网链接](#)

- **配置全局默认过滤器**

这些自带的过滤器可以和使用 [路由前缀](#) 章节中的用法类似，也可以将这些过滤器配置成不只是针对某个路由；而是可以对所有路由生效，也就是配置默认过滤器：

了解如下：

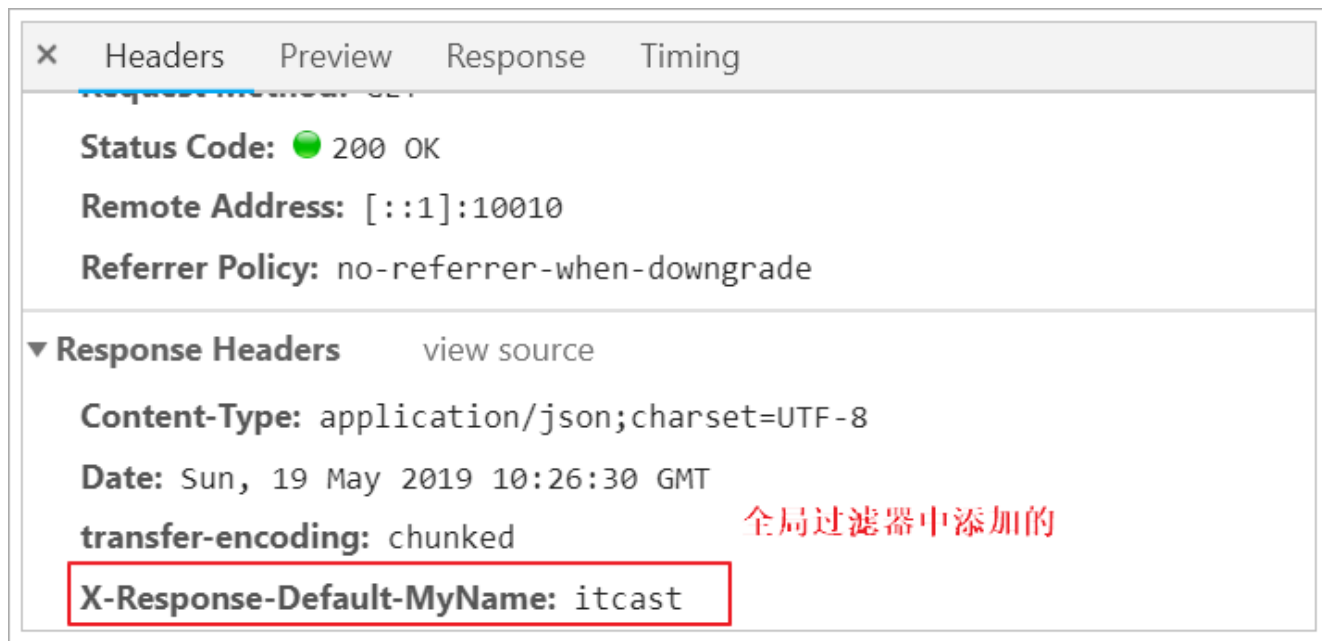
```
1  server:
2    port: 10010
3  spring:
4    application:
5      name: api-gateway
6    cloud:
7      gateway:
8        # 默认过滤器，对所有路由生效
```

```

9      default-filters:
10         # 响应头过滤器，对输出的响应设置其头部属性名称为X-Response-Default-MyName，值为itcast；
           如果有多个参数多则重写一行设置不同的参数
11         - AddResponseHeader=X-Response-Default-MyName, itcast
12      routes:
13         # 路由id，可以随意写
14         - id: user-service-route
15           # 代理的服务地址；lb表示从eureka中获取具体服务
16           uri: lb://user-service
17           # 路由断言，可以配置映射路径
18           predicates:
19             - Path=/api/user/**
20           filters:
21             # 表示过滤1个路径，2表示两个路径，以此类推
22             - StripPrefix=1

```

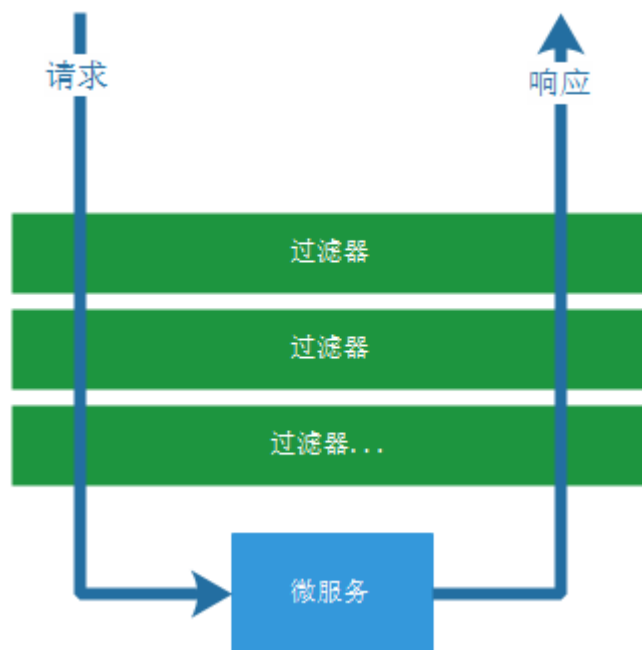
上述配置后，再访问 <http://localhost:10010/api/user/8> 的话；那么可以从其响应中查看到如下信息：



- **过滤器类型：**Gateway实现方式上，有两种过滤器；
 1. **局部过滤器：**通过 `spring.cloud.gateway.routes.filters` 配置在具体路由下，只作用在当前路由上；自带的过滤器都可以配置或者自定义按照自带过滤器的方式。如果配置 `spring.cloud.gateway.default-filters` 上会对所有路由生效也算是全局的过滤器；但是这些过滤器的实现上都是要实现GatewayFilterFactory接口。
 2. **全局过滤器：**不需要在配置文件中配置，作用在所有的路由上；实现 `GlobalFilter` 接口即可。

2.7.2. 执行生命周期

Spring Cloud Gateway 的 Filter 的生命周期也类似Spring MVC的拦截器有两个：“pre”和“post”。“pre”和“post”分别会在请求被执行前调用和被执行后调用。



这里的 `pre` 和 `post` 可以通过过滤器的 `GatewayFilterChain` 执行 `filter` 方法前后来实现。

2.7.3. 使用场景

常见的应用场景如下：

- 请求鉴权：一般 `GatewayFilterChain` 执行 `filter` 方法前，如果发现没有访问权限，直接就返回空。
- 异常处理：一般 `GatewayFilterChain` 执行 `filter` 方法后，记录异常并返回。
- 服务调用时长统计：`GatewayFilterChain` 执行 `filter` 方法前后根据时间统计。

2.8. 自定义过滤器

2.8.1. 自定义局部过滤器

需求：在 `Application.yml` 中对某个路由配置过滤器，该过滤器可以在控制台输出配置文件中指定名称的请求参数的值。

1) 编写过滤器

在 `heima-gateway` 工程编写过滤器工厂类 `MyParamGatewayFilterFactory`

```
1 package com.itheima.gateway.filter;
2
3 import org.springframework.cloud.gateway.filter.GatewayFilter;
4 import org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
5 import org.springframework.http.server.reactive.ServerHttpRequest;
6 import org.springframework.stereotype.Component;
7
8 import java.util.Arrays;
9 import java.util.List;
```

```

10
11 @Component
12 public class MyParamGatewayFilterFactory extends
AbstractGatewayFilterFactory<MyParamGatewayFilterFactory.Config> {
13
14     public static final String PARAM_NAME = "param";
15
16     public MyParamGatewayFilterFactory() {
17         super(Config.class);
18     }
19
20     @Override
21     public List<String> shortcutFieldOrder() {
22         return Arrays.asList(PARAM_NAME);
23     }
24
25     @Override
26     public GatewayFilter apply(Config config) {
27         return (exchange, chain) -> {
28             ServerHttpRequest request = exchange.getRequest();
29
30             if (request.getQueryParams().containsKey(config.param)) {
31                 request.getQueryParams().get(config.param)
32                     .forEach(value -> System.out.printf("-----局部过滤器-----%s
= %s-----",
33                                     config.param, value));
34             }
35
36             return chain.filter(exchange);
37         };
38     }
39
40     public static class Config {
41         private String param;
42
43         public String getParam() {
44             return param;
45         }
46
47         public void setParam(String param) {
48             this.param = param;
49         }
50     }
51
52 }

```

2) 修改配置文件

在heima-gateway工程修改 heima-gateway\src\main\resources\application.yml 配置文件

```

1 server:

```

```

2   port: 10010
3   spring:
4     application:
5       name: api-gateway
6     cloud:
7       gateway:
8         routes:
9           # 路由id, 可以随意写
10          - id: user-service-route
11            # 代理的服务地址; lb表示从eureka中获取具体服务
12            uri: lb://user-service
13            # 路由断言, 可以配置映射路径
14            predicates:
15              - Path=/api/user/**
16            filters:
17              # 表示过滤1个路径, 2表示两个路径, 以此类推
18              - StripPrefix=1
19              # 自定义过滤器
20              - MyParam=name
21   eureka:
22     client:
23       service-url:
24         defaultZone: http://127.0.0.1:10086/eureka
25     instance:
26       prefer-ip-address: true
27

```

注意：自定义过滤器的命名应该为：***GatewayFilterFactory

测试访问：<http://localhost:10010/api/user/8?name=itcast> 检查后台是否输出name和itcast；但是若访问<http://localhost:10010/api/user/8?name2=itcast> 则是不会输出的。

2.8.2. 自定义全局过滤器

需求：模拟一个登录的校验。基本逻辑：如果请求中有token参数，则认为请求有效，放行。

在heima-gateway工程编写全局过滤器类MyGlobalFilter

```

1   package com.itheima.gateway.filter;
2
3   import org.apache.commons.lang.StringUtils;
4   import org.springframework.cloud.gateway.filter.GatewayFilterChain;
5   import org.springframework.cloud.gateway.filter.GlobalFilter;
6   import org.springframework.core.Ordered;
7   import org.springframework.http.HttpStatus;
8   import org.springframework.stereotype.Component;
9   import org.springframework.web.server.ServerWebExchange;
10  import reactor.core.publisher.Mono;

```

```

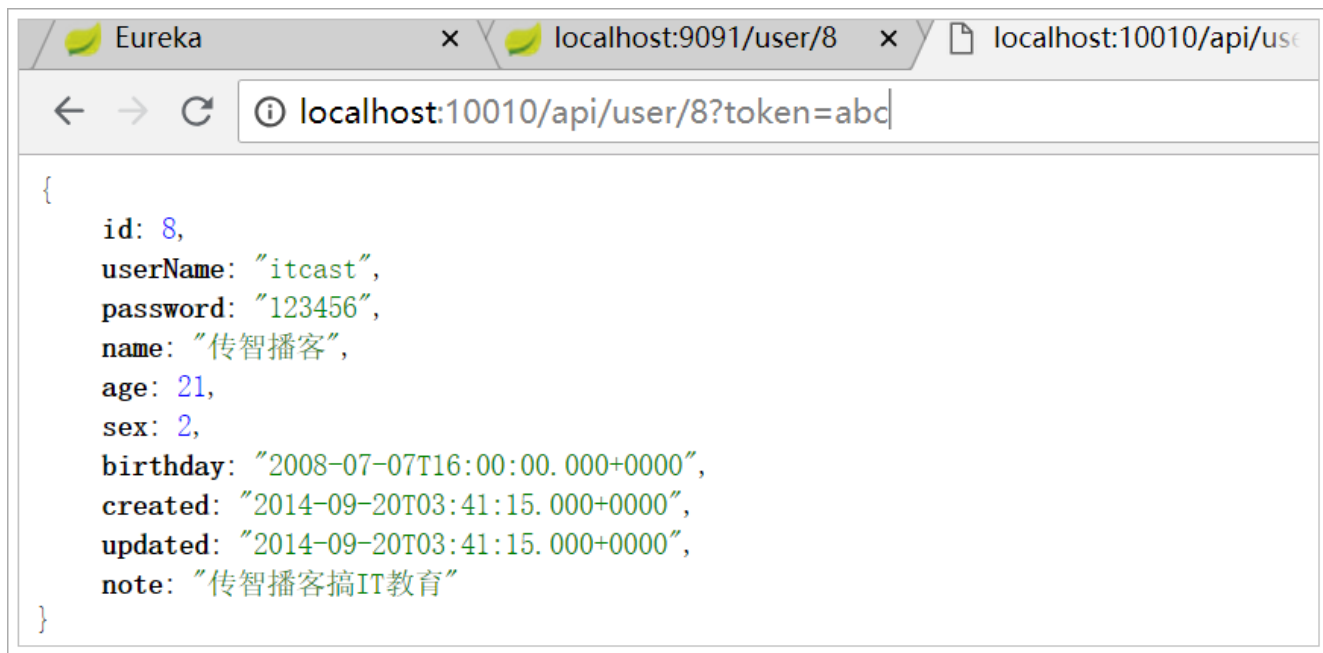
11
12 @Component
13 public class MyGlobalFilter implements GlobalFilter, Ordered {
14     @Override
15     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
16         System.out.println("-----全局过滤器MyGlobalFilter-----");
17         String token = exchange.getRequest().getQueryParams().getFirst("token");
18         if (StringUtils.isBlank(token)) {
19             exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
20             return exchange.getResponse().setComplete();
21         }
22         return chain.filter(exchange);
23     }
24
25     @Override
26     public int getOrder() {
27         //值越小越先执行
28         return 1;
29     }
30 }
31

```

访问 <http://localhost:10010/api/user/8>

The screenshot shows a web browser's developer tools interface. The address bar displays the URL `http://localhost:10010/api/user/8`. Below the address bar, the HTTP method is set to `GET`, which is circled in red. The `Raw` tab is selected. The status bar at the bottom indicates a `401 Unauthorized` response, also circled in red, with a loading time of `10 ms`. The request headers section shows the `User-Agent` and `Content-Type` information.

访问 <http://localhost:10010/api/user/8?token=abc>



2.9. 负载均衡和熔断（了解）

Gateway中默认就已经集成了Ribbon负载均衡和Hystrix熔断机制。但是所有的超时策略都是走的默认值，比如熔断超时时间只有1S，很容易就触发了。因此建议手动进行配置：

```
1 hystrix:
2   command:
3     default:
4       execution:
5         isolation:
6           thread:
7             timeoutInMilliseconds: 6000
8 ribbon:
9   ConnectTimeout: 1000
10  ReadTimeout: 2000
11  MaxAutoRetries: 0
12  MaxAutoRetriesNextServer: 0
```

2.10. Gateway跨域配置

一般网关都是所有微服务的统一入口，必然在被调用的时候会出现跨域问题。

跨域：在js请求访问中，如果访问的地址与当前服务器的域名、ip或者端口号不一致则称为跨域请求。若不解决则不能获取到对应地址的返回结果。

如：从在<http://localhost:9090>中的js访问 <http://localhost:9000>的数据，因为端口不同，所以也是跨域请求。

在访问Spring Cloud Gateway网关服务器的时候，出现跨域问题的话；可以在网关服务器中通过配置解决，允许哪些服务是可以跨域请求的；具体配置如下：

```
1  spring:
2    cloud:
3      gateway:
4        globalcors:
5          corsConfigurations:
6            '[/**]':
7              #allowedOrigins: * # 这种写法或者下面的都可以，*表示全部
8              allowedOrigins:
9                - "http://docs.spring.io"
10             allowedMethods:
11               - GET
```

上述配置表示：可以允许来自 <http://docs.spring.io> 的get请求方式获取服务数据。

allowedOrigins 指定允许访问的服务器地址，如：<http://localhost:10000> 也是可以的。

'[/**]' 表示对所有访问到网关服务器的请求地址

官网具体说明：https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.1.1.RELEASE/multi/multi__cors_configuration.html

2.11. Gateway的高可用（了解）

启动多个Gateway服务，自动注册到Eureka，形成集群。如果是服务内部访问，访问Gateway，自动负载均衡，没问题。

但是，Gateway更多是外部访问，PC端、移动端等。它们无法通过Eureka进行负载均衡，那么该怎么办？

此时，可以使用其它的服务网关，来对Gateway进行代理。比如：Nginx

2.12. Gateway与Feign的区别

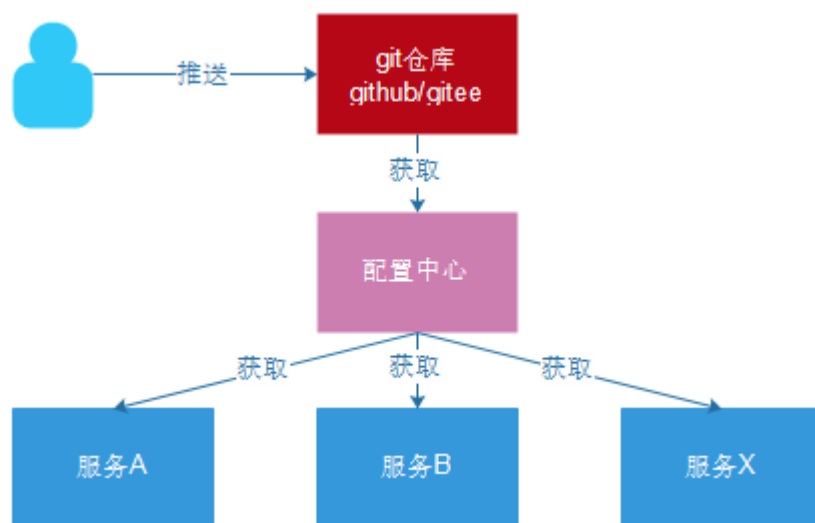
- Gateway 作为整个应用的流量入口，接收所有的请求，如PC、移动端等，并且将不同的请求转发至不同的处理微服务模块，其作用可视为nginx；大部分情况下用作权限鉴定、服务端流量控制
- Feign 则是将当前微服务的部分服务接口暴露出来，并且主要用于各个微服务之间的服务调用

3. Spring Cloud Config分布式配置中心

3.1. 简介

在分布式系统中，由于服务数量非常多，配置文件分散在不同的微服务项目中，管理不方便。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，也支持放在远程Git仓库（GitHub、码云）。

使用Spring Cloud Config配置中心后的架构如下图：



配置中心本质上也是一个微服务，同样需要注册到Eureka服务注册中心！

3.2. Git配置管理

3.2.1. 远程Git仓库

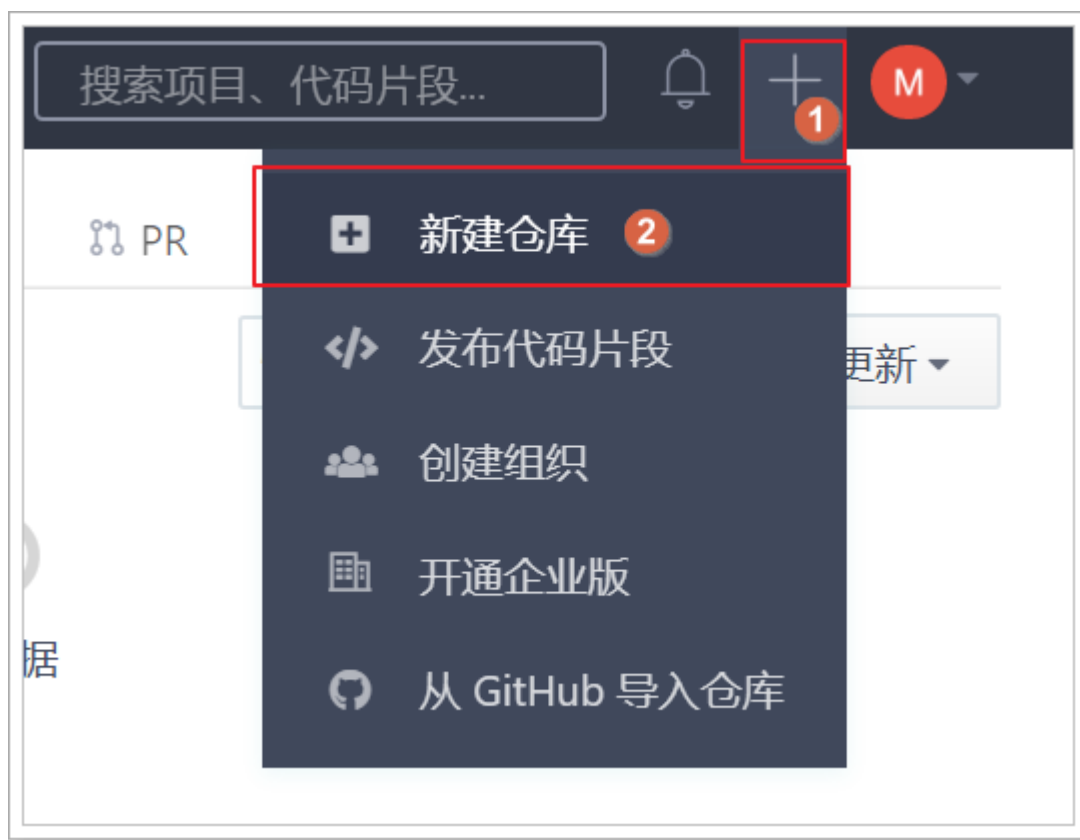
知名的Git远程仓库有国外的GitHub和国内的码云（gitee）；但是使用GitHub时，国内的用户经常遇到的问题是访问速度太慢，有时候还会出现无法连接的情况。如果希望体验更好一些，可以使用国内的Git托管服务——码云（gitee.com）。

与GitHub相比，码云也提供免费的Git仓库。此外，还集成了代码质量检测、项目演示等功能。对于团队协作开发，码云还提供了项目管理、代码托管、文档管理的服务。本章中使用的远程Git仓库是码云。

码云访问地址：<https://gitee.com/>

3.2.2. 创建远程仓库

首先要使用码云上的私有远程git仓库需要先注册帐号；请先自行访问网站并注册帐号，然后使用帐号登录码云控制台并创建公开仓库。



仓库名称 ✓

heima-config 1

归属 **路径**

M meinitcasthavefun / heima-config 2

仓库地址: <https://gitee.com/liaojianbin/heima-config>

仓库介绍 非必填

配置中心 3

是否开源 4

☐ 私有 ☒ 公开

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 **添加 .gitignore** **添加开源许可证** ⓘ

Java 5 JetBrains 6 Apache-2.0 7

☒ 8 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 ⓘ

3.2.3. 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。

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，测试环境、生产环境等

如user-dev.yaml，表示用户微服务开发环境下使用的配置文件。

这里将user-service工程的配置文件application.yaml文件的内容复制作为user-dev.yaml文件的内容，具体配置如下：



创建 `user-dev.yaml`；内容来自 `user-service\src\main\resources\application.yaml`（方便后面测试user-service项目的配置），可以如下：

```
1  server:
2    port: ${port:9091}
3  spring:
4    datasource:
5      driver-class-name: com.mysql.jdbc.Driver
6      url: jdbc:mysql://localhost:3306/springcloud
7      username: root
8      password: root
9    application:
10     #应用名
11     name: user-service
12  mybatis:
13    type-aliases-package: com.itheima.user.pojo
14  eureka:
15    client:
16      service-url:
17        defaultZone: http://127.0.0.1:10086/eureka
18    instance:
```

```
19 ip-address: 127.0.0.1
20 prefer-ip-address: true
21 lease-expiration-duration-in-seconds: 90
22 lease-renewal-interval-in-seconds: 30
```

heima-config / user-dev.yml master 分支 提示: 输入 / 可以将文件创建到新文件夹下

```
1 server:
2   port: ${port:9091}
3 spring:
4   datasource:
5     driver-class-name: com.mysql.jdbc.Driver
6     url: jdbc:mysql://localhost:3306/springcloud
7     username: root
8     password: root
9   application:
10    #应用名
11    name: user-service
12 mybatis:
13   type-aliases-package: com.itheima.user.pojo
14 eureka:
15   client:
16     service-url:
17       defaultZone: http://127.0.0.1:10086/eureka
18   instance:
19     ip-address: 127.0.0.1
```

创建完user-dev.yml配置文件之后，gitee中的仓库如下：

master + Pull Request + Issue 文件 Web IDE 挂件

M meinitcasthavefun 最后提交于 6分钟前 用户微服务配置文件user-dev

.gitignore	Initial commit
LICENSE	Initial commit
README.en.md	Initial commit
README.md	Initial commit
user-dev.yml	用户微服务配置文件user-dev

3.3. 搭建配置中心微服务

3.3.1. 创建工程

创建配置中心微服务工程：

New Module

Add as module to: com.itheima:heima-springcloud:1.0-SNAPSHOT

Parent: com.itheima:heima-springcloud:1.0-SNAPSHOT

GroupId: com.itheima

ArtifactId: config-server

Version: 1.0-SNAPSHOT

☐ Inherit

☒ Inherit

New Module

Module name: config-server

Content root: D:\workspaces\springcloud\heima-springcloud\config-server

Module file location: D:\workspaces\springcloud\heima-springcloud\config-server

添加依赖，修改 config-server\pom.xml 如下：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5         http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <parent>
7         <artifactId>heima-springcloud</artifactId>
8         <groupId>com.itheima</groupId>
9         <version>1.0-SNAPSHOT</version>
10    </parent>
11    <modelVersion>4.0.0</modelVersion>
12    <groupId>com.itheima</groupId>
13    <artifactId>config-server</artifactId>
14
15    <dependencies>
16        <dependency>
17            <groupId>org.springframework.cloud</groupId>
18            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
19        </dependency>
20        <dependency>
21            <groupId>org.springframework.cloud</groupId>
22            <artifactId>spring-cloud-config-server</artifactId>
23        </dependency>
24    </dependencies>
25 </project>
```

3.3.2. 启动类

创建配置中心工程 config-server 的启动类;

config-server\src\main\java\com\itheima\config\ConfigServerApplication.java 如下:

```
1 package com.itheima.config;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.cloud.config.server.EnableConfigServer;
6
7 @SpringBootApplication
8 @EnableConfigServer
9 public class ConfigServerApplication {
10     public static void main(String[] args) {
11         SpringApplication.run(ConfigServerApplication.class, args);
12     }
13 }
14
```

3.3.3. 配置文件

创建配置中心工程 config-server 的配置文件;

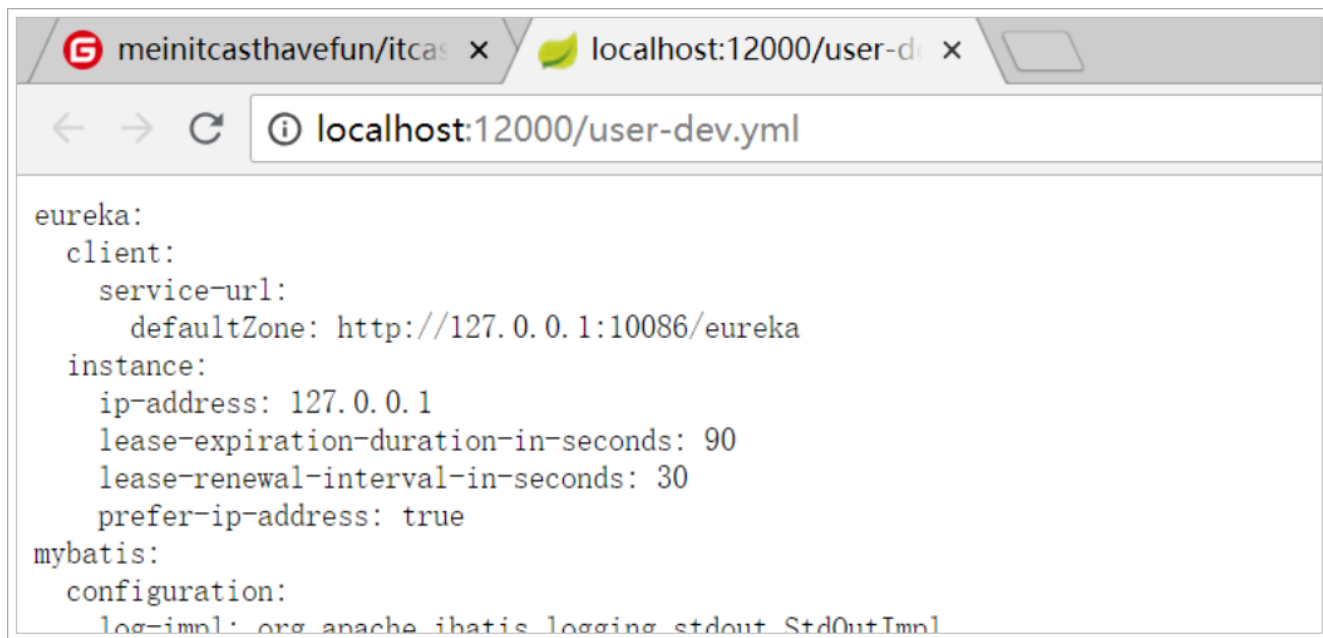
config-server\src\main\resources\application.yml 如下:

```
1 server:
2   port: 12000
3 spring:
4   application:
5     name: config-server
6   cloud:
7     config:
8       server:
9         git:
10          uri: https://gitee.com/liaojianbin/heima-config.git
11 eureka:
12   client:
13     service-url:
14       defaultZone: http://127.0.0.1:10086/eureka
15
```

注意上述的 spring.cloud.config.server.git.uri 则是在码云创建的仓库地址; 可修改为你自己创建的仓库地址

3.3.4. 启动测试

启动eureka注册中心和配置中心; 然后访问<http://localhost:12000/user-dev.yml>, 查看能否输出在码云存储管理的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到最新数据。



```
eureka:
  client:
    service-url:
      defaultZone: http://127.0.0.1:10086/eureka
  instance:
    ip-address: 127.0.0.1
    lease-expiration-duration-in-seconds: 90
    lease-renewal-interval-in-seconds: 30
    prefer-ip-address: true
mybatis:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
```

3.4. 获取配置中心配置

前面已经完成了配置中心微服务的搭建，下面我们就需要改造一下用户微服务 `user-service`，配置文件信息不再由微服务项目提供，而是从配置中心获取。如下对 `user-service` 工程进行改造。

3.4.1. 添加依赖

在 `user-service` 工程中的 `pom.xml` 文件中添加如下依赖：

```
1      <dependency>
2          <groupId>org.springframework.cloud</groupId>
3          <artifactId>spring-cloud-starter-config</artifactId>
4          <version>2.1.1.RELEASE</version>
5      </dependency>
```

3.4.2. 修改配置

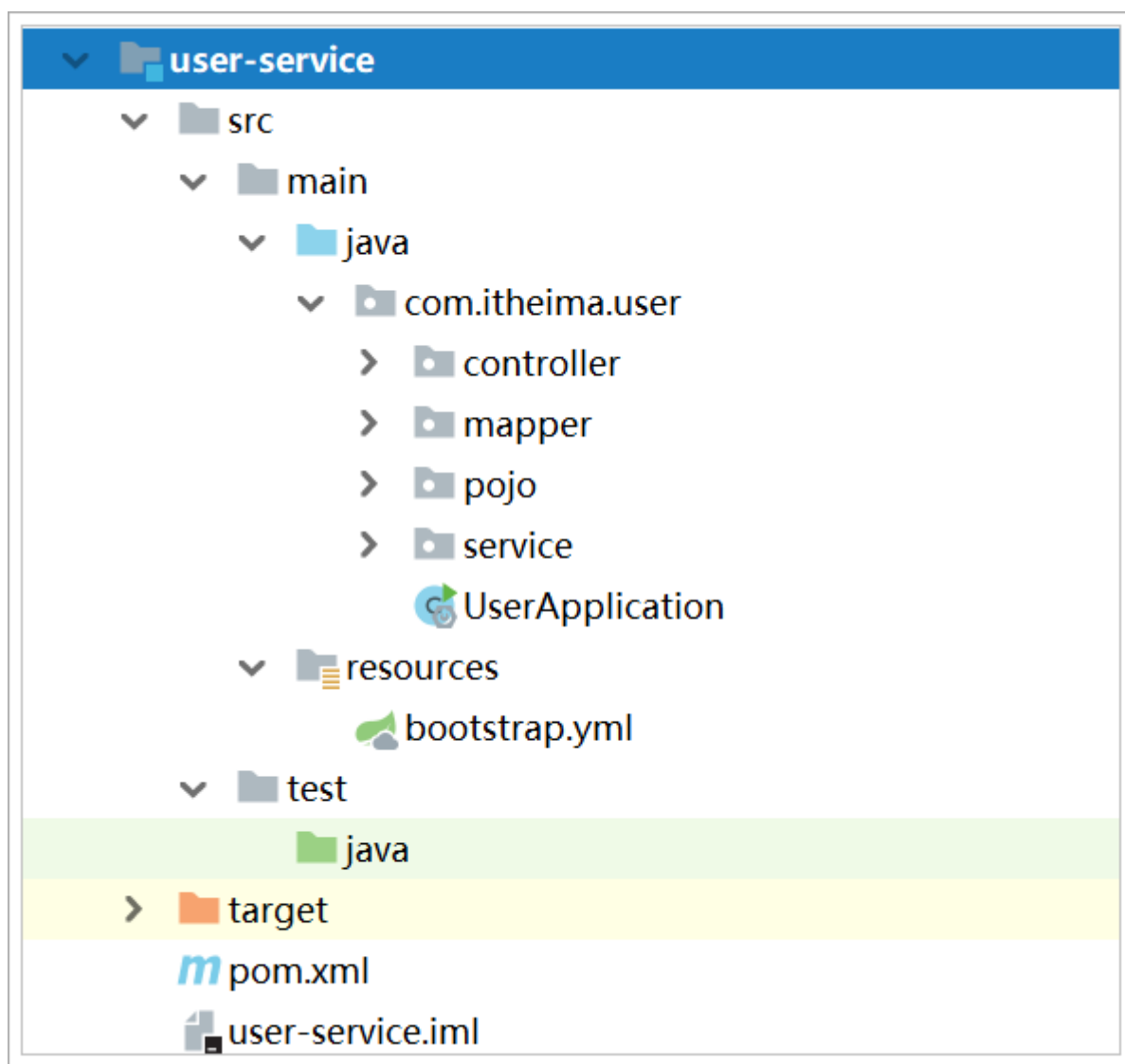
1. 删除 `user-service` 工程的 `user-service\src\main\resources\application.yml` 文件（因为该文件从配置中心获取）
2. 创建 `user-service` 工程 `user-service\src\main\resources\bootstrap.yml` 配置文件

```
1  spring:
2      cloud:
3          config:
4              # 与远程仓库中的配置文件的application保持一致
5              name: user
6              # 远程仓库中的配置文件的profile保持一致
7              profile: dev
8              # 远程仓库中的版本保持一致
```



```
9      label: master
10     discovery:
11       # 使用配置中心
12       enabled: true
13       # 配置中心服务id
14       service-id: config-server
15   eureka:
16     client:
17       service-url:
18         defaultZone: http://127.0.0.1:10086/eureka
```

user-service 工程修改后结构:



bootstrap.yml文件也是Spring Boot的默认配置文件，而且其加载的时间相比于application.yml更早。

application.yml和bootstrap.yml虽然都是Spring Boot的默认配置文件，但是定位却不相同。bootstrap.yml可以理解成系统级别的一些参数配置，这些参数一般是不会变动的。application.yml可以用来定义应用级别的参数，如果搭配 spring cloud config 使用，application.yml 里面定义的文件可以实现动态替换。

总结就是，bootstrap.yml文件相当于项目启动时的引导文件，内容相对固定。application.yml文件是微服务的一些常规配置参数，变化比较频繁。

3.4.3. 启动测试

启动注册中心 eureka-server、配置中心 config-server、用户服务 user-service，如果启动没有报错其实已经使用上配置中心内容，可以到注册中心查看，也可以检验 user-service 的服务。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - localhost:config-server:12000
USER-SERVICE	n/a (1)	(1)	UP (1) - localhost:user-service:9091

4. Spring Cloud Bus服务总线

4.1. 问题

前面已经完成了将微服务中的配置文件集中存储在远程Git仓库，并且通过配置中心微服务从Git仓库拉取配置文件，当用户微服务启动时会连接配置中心获取配置信息从而启动用户微服务。

如果我们更新Git仓库中的配置文件，那用户微服务是否可以及时接收到新的配置信息并更新呢？

4.1.1. 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。

itcast-config / user-dev.yml

user-dev.yml master 分支

```
17 client:
18     service-url:
19         defaultZone: http://127.0.0.1:10086/eureka
20 instance:
21     #提供服务时候使用ip地址而不是默认的主机名
22     ip-address: 127.0.0.1
23     #优先使用ip地址方式
24     prefer-ip-address: true
25     lease-expiration-duration-in-seconds: 90
26     lease-renewal-interval-in-seconds: 30
27
28 test:
29     name: heima
```

4.1.2. 修改UserController

修改 user-service 工程中的处理器类;

user-service\src\main\java\com\itheima\user\controller\UserController.java 如下:

```
1 package com.itheima.user.controller;
2
3 import com.itheima.user.pojo.User;
4 import com.itheima.user.service.UserService;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.beans.factory.annotation.Value;
7 import org.springframework.web.bind.annotation.GetMapping;
8 import org.springframework.web.bind.annotation.PathVariable;
9 import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11
12 @RestController
13 @RequestMapping("/user")
14 public class UserController {
15
```

```

16     @Autowired
17     private UserService userService;
18
19     @Value("${test.name}")
20     private String name;
21
22     @GetMapping("/{id}")
23     public User queryById(@PathVariable Long id){
24         System.out.println("配置文件中的test.name = " + name);
25         return userService.queryById(id);
26     }
27 }

```

4.1.3. 测试

依次启动注册中心 `eureka-server`、配置中心 `config-server`、用户服务 `user-service`；然后修改Git仓库中的配置信息，访问用户微服务，查看输出内容。

结论：通过查看用户微服务控制台的输出结果可以发现，我们对于Git仓库中配置文件的修改并没有及时更新到用户微服务，只有重启用户微服务才能生效。

如果想在重启微服务的情况下更新配置该如何实现呢？**可以使用Spring Cloud Bus来实现配置的自动更新。**

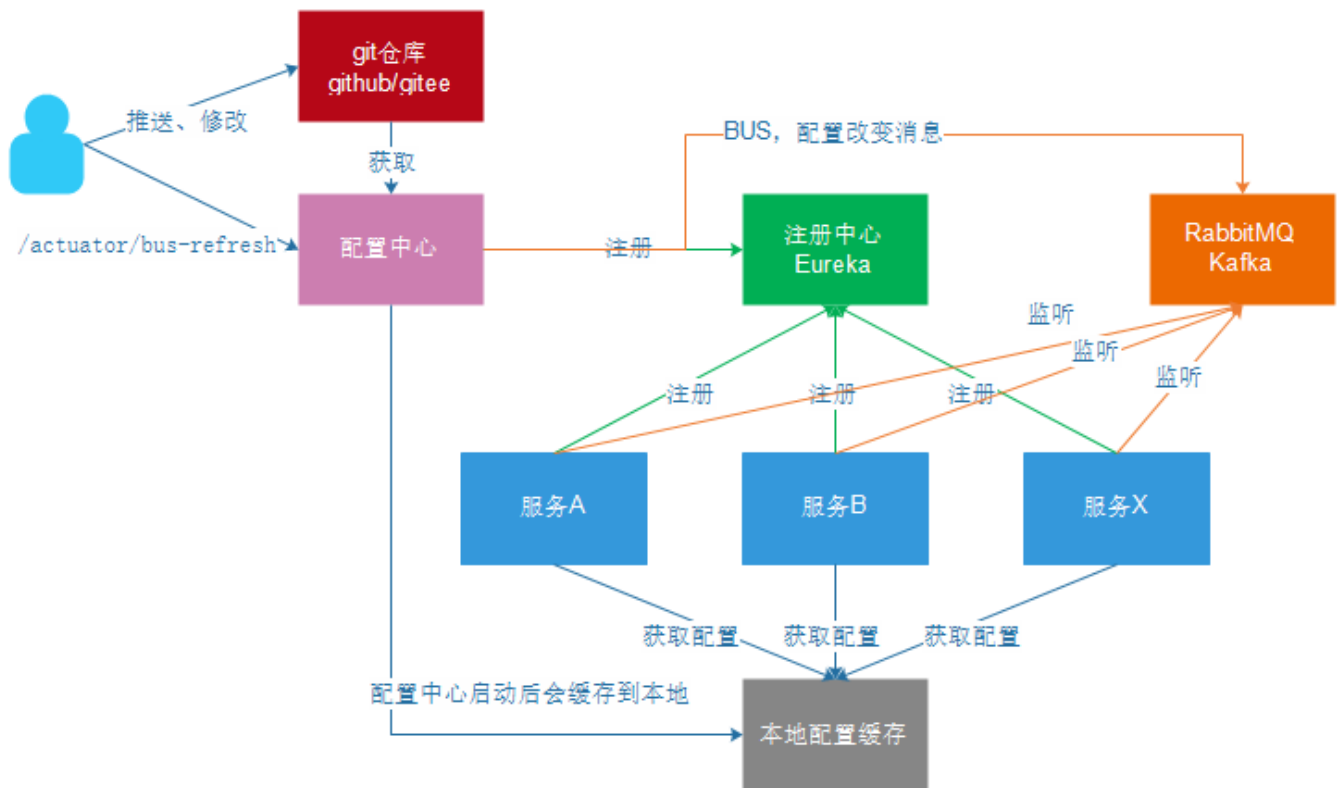
需要注意的是Spring Cloud Bus底层是基于RabbitMQ实现的，默认使用本地的消息队列服务，所以需要提前启动本地RabbitMQ服务（安装RabbitMQ以后才有），如下：

服务(本地)					
	名称	描述	状态	启动类型	登录为
RabbitMQ 停止此服务 重启此服务 描述: Multi-protocol open source messaging broker	Print Spooler	该服务在后台执行打印作业并处理与打印机的交互。...	正在运行	自动	本地系统
	Printer Extensions and Notifications	This service opens custom printer dialog boxes and...		手动	本地系统
	PrintWorkflow_a2f4e26	打印工作流		手动	本地系统
	Problem Reports and Solutions Control Pan...	此服务为查看、发送和删除“问题报告和解决方案”控...		手动	本地系统
	Program Compatibility Assistant Service	此服务为程序兼容性助手(PCA)提供支持。PCA 监视...	正在运行	手动	本地系统
	QPCore Service	腾讯安全服务	正在运行	自动	本地系统
	QQ拼音输入法基础服务	为QQ拼音输入法提供基础服务，如果禁用此服务，...		手动	本地系统
	Quality Windows Audio Video Experience	优质 Windows 音视频体验(qWave)是用于 IP 家庭...		手动	本地系统
	RabbitMQ	Multi-protocol open source messaging broker	正在运行	手动	本地系统
	Remote Access Auto Connection Manager	无论什么时候，当某个程序引用一个远程 DNS 或者 ...		手动	本地系统

4.2. Spring Cloud Bus简介

Spring Cloud 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。也就是消息总线可以为微服务做监控，也可以实现应用程序之间相互通信。Spring Cloud Bus可选的消息代理有RabbitMQ和Kafka。

使用了Bus之后：



4.3. 改造配置中心

1. 在 config-server 项目的pom.xml文件中加入Spring Cloud Bus相关依赖

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-bus</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.cloud</groupId>
7   <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
8 </dependency>
9
```

2. 在 config-server 项目修改application.yml文件如下:

```
1 server:
2   port: 12000
3 spring:
4   application:
5     name: config-server
6   cloud:
7     config:
8       server:
9         git:
10          uri: https://gitee.com/liaojianbin/heima-config.git
11 # 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
```

```

12     rabbitmq:
13         host: localhost
14         port: 5672
15         username: guest
16         password: guest
17     eureka:
18         client:
19             service-url:
20                 defaultZone: http://127.0.0.1:10086/eureka
21     management:
22         endpoints:
23             web:
24                 exposure:
25                     # 暴露触发消息总线的地址
26                     include: bus-refresh
27

```

4.4. 改造用户服务

1. 在用户微服务 `user-service` 项目的pom.xml中加入Spring Cloud Bus相关依赖

```

1     <dependency>
2         <groupId>org.springframework.cloud</groupId>
3         <artifactId>spring-cloud-bus</artifactId>
4     </dependency>
5     <dependency>
6         <groupId>org.springframework.cloud</groupId>
7         <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
8     </dependency>
9     <dependency>
10        <groupId>org.springframework.boot</groupId>
11        <artifactId>spring-boot-starter-actuator</artifactId>
12    </dependency>

```

2. 修改 `user-service` 项目的bootstrap.yml如下:

```

1     spring:
2         cloud:
3             config:
4                 # 与远程仓库中的配置文件的application保持一致
5                 name: user
6                 # 远程仓库中的配置文件的profile保持一致
7                 profile: dev
8                 # 远程仓库中的版本保持一致
9                 label: master
10            discovery:
11                # 使用配置中心
12                enabled: true

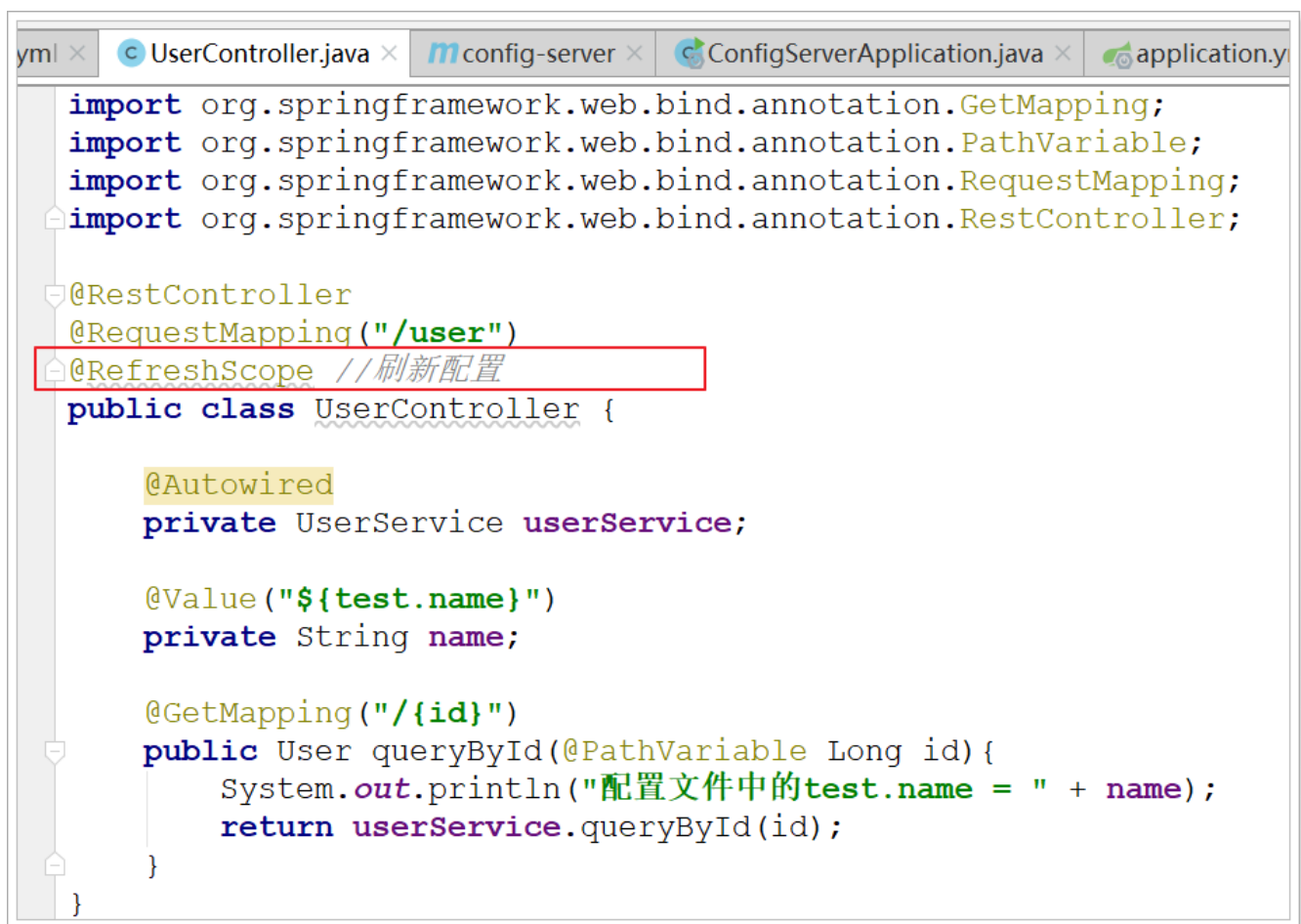
```

```

13     # 配置中心服务id
14     service-id: config-server
15     # 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
16     rabbitmq:
17         host: localhost
18         port: 5672
19         username: guest
20         password: guest
21     eureka:
22         client:
23             service-url:
24                 defaultZone: http://127.0.0.1:10086/eureka

```

3. 改造用户微服务 user-service 项目的UserController



```

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

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

    @Autowired
    private UserService userService;

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

    @GetMapping("/{id}")
    public User queryById(@PathVariable Long id) {
        System.out.println("配置文件中的test.name = " + name);
        return userService.queryById(id);
    }
}

```

4.5. 测试

前面已经完成了配置中心微服务和用户微服务的改造, 下面来测试一下, 当我们修改了Git仓库中的配置文件, 用户微服务是否能够在不重启的情况下自动更新配置信息。

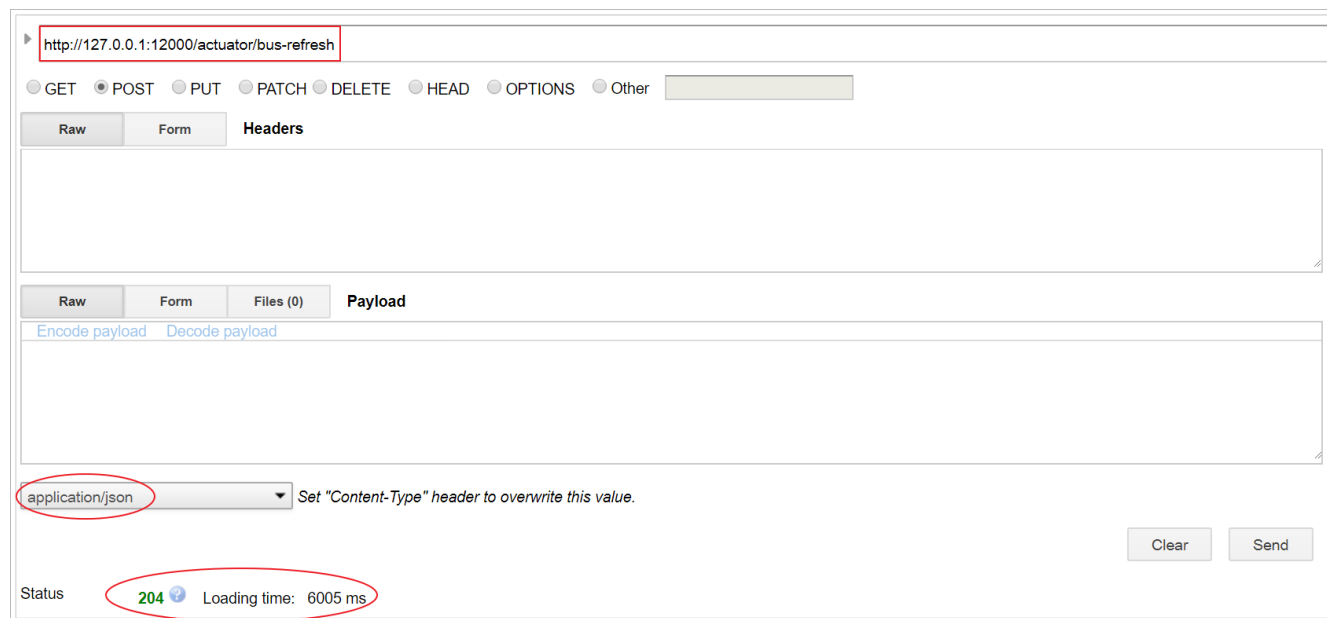
测试步骤:

第一步: 依次启动注册中心 `eureka-server`、配置中心 `config-server`、用户服务 `user-service`

第二步: 访问用户微服务 <http://localhost:9091/user/8>; 查看IDEA控制台输出结果

第三步：修改Git仓库中配置文件 `user-dev.yml` 的 `test.name` 内容

第四步：使用Postman或者RESTClient工具发送POST方式请求访问地址<http://127.0.0.1:12000/actuator/bus-refresh>



第五步：访问用户微服务系统控制台查看输出结果

说明：

- 1、Postman或者RESTClient是一个可以模拟浏览器发送各种请求（POST、GET、PUT、DELETE等）的工具
- 2、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config-server中的application.yml文件的配置项include的内容
- 3、请求<http://127.0.0.1:12000/actuator/bus-refresh>地址的作用是访问配置中心的消息总线服务，消息总线服务接收到请求后会向消息队列中发送消息，各个微服务会监听消息队列。当微服务接收到队列中的消息后，会重新从配置中心获取最新的配置信息。

4.6. Spring Cloud 体系技术综合应用概览

