

14 声明式服务调用 Feign

更新时间：2019-06-21 16:20:28



“你若喜爱你自己的价值，你就得给世界创造价值。

——歌德”

在前面几篇文章中，主要和大家介绍了服务的注册与消费。在介绍过程中，我们从最最原始的手动利用 `DiscoveryClient` 发现服务开始，手动实现负载均衡，再到最后的自动化配置。相信经过前面几篇文章的学习，大家对微服务之间的调用应该有了一个基本的认知。但是我们前面的所有服务调用都是手动写 `RestTemplate` 来实现的，大家可能已经发现这样写有点麻烦，每次都要写请求 `Url`、配置响应数据类型，最后还要组装参数，更重要的是这些都是一些重复的工作，代码高度相似，每个请求只有 `Url` 不同，请求方法不同、参数不同，其它东西基本都是一样的，既然如此，那有没有办法简化请求呢？有！这就是本文我们要聊的声明式微服务调用 `Feign`。不对，严格来说，应该叫 `OpenFeign`，为什么这么说呢？早期我们用的是叫 `Netflix Feign`，不过这个东西的最近一次更新还停留在 2016年7月，`OpenFeign` 则是 `Spring Cloud` 团队在 `Netflix Feign` 基础上开发出来的声明式服务调用组件，`OpenFeign`也一直在维护，具体的迁移工作，大家参考[这个 Issue](#)。

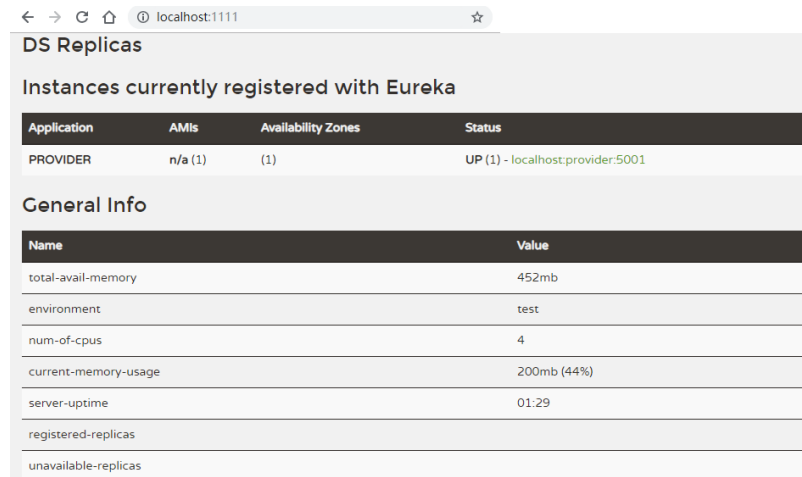
准备工作

和前面几篇文章的步骤一样，首先我们还是要先搭建一个父工程，然后创建一个服务注册中心。OK，那就开始吧！首先创建一个名为 `Feign` 的 `Maven` 父工程，然后在父工程中创建一个 `eureka` 工程充当我们的服务注册中心，具体步骤我这里就不再赘述了，大家如果忘了如何搭建服务注册中心，可以参考前面的文章。

服务注册中心搭建成功后，接下来我们还要再搭建一个 `provider` 用来提供服务。这个 `provider` 和前面 `provider` 的搭建也是基本一致的。`provider` 搭建成功后，依然提供一个 `HelloController` 接口，里边配上一个 `/hello` 接口，像下面这样：

```
@RestController
public class HelloController {
    @GetMapping("/hello")
    public String hello(String name) {
        return "hello " + name + " !";
    }
}
```

然后分别启动服务注册中心 **eureka** 以及服务提供者 **provider** ，然后浏览器中看到如下页面，表示我们的准备工作已经 OK 啦：



The screenshot shows the Eureka DS Replicas page in a browser. The page title is "DS Replicas". Below the title, it says "Instances currently registered with Eureka". There is a table with the following data:

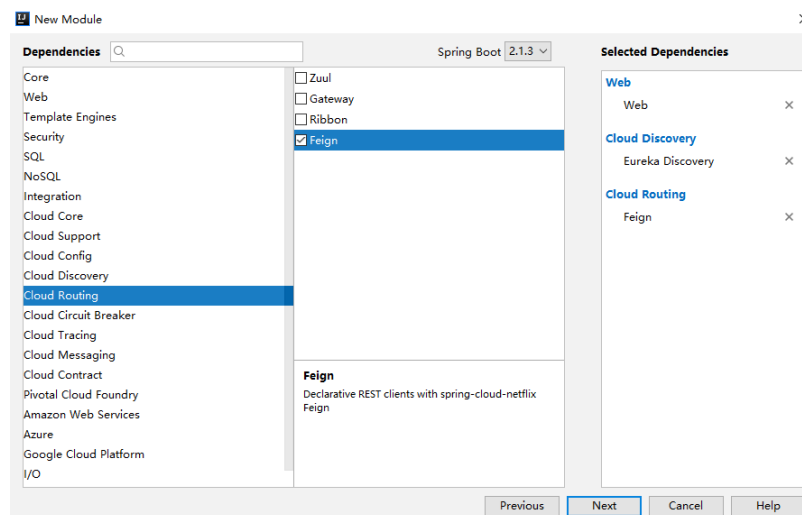
Application	AMIs	Availability Zones	Status
PROVIDER	n/a (1)	(1)	UP (1) - localhost:provider:5001

Below the table, there is a section titled "General Info" with a table of system metrics:

Name	Value
total-avail-memory	452mb
environment	test
num-of-cpus	4
current-memory-usage	200mb (44%)
server-uptime	01:29
registered-replicas	
unavailable-replicas	

基本使用

准备工作完成后，接下来我们创建一个 **feign-consumer** 微服务，不同于前面创建的 **consumer** ，这里的 **feign-consumer** 要多添加一个依赖，如下图：



项目创建成功后，添加的依赖如下：

```

<properties>
  <spring-cloud.version>Greenwich.SR1</spring-cloud.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
  </dependency>
</dependencies>
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifactId>
      <version>${spring-cloud.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

创建完成后，首先在 `application.properties` 文件上添加配置，将 `feign-consumer` 注册到 `eureka` 上，如下：

```

spring.application.name=feign-consumer
server.port=5002
eureka.client.service-url.defaultZone=http://localhost:1111/eureka

```

要使用 `Feign`，首先在项目启动类上添加 `@EnableFeignClients` 注解表示开启 `Feign` 的支持，如下：

```

@SpringBootApplication
@EnableFeignClients
public class FeignConsumerApplication {

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

}

```

接下来创建一个 `HelloService` 接口，用来绑定 `provider` 提供的服务，如下：

```

@FeignClient("provider")
public interface HelloService {
    @GetMapping("/hello")
    String hello(@RequestParam("name") String name);
}

```

这里声明了一个接口，然后做了两件事：

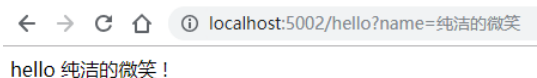
1. 使用 `@FeignClient("provider")` 注解将当前接口和 `provider` 服务绑定，`provider` 是服务名，大小写不敏感；
2. 然后使用 `SpringMVC` 的 `@GetMapping("/hello")` 注解将 `hello` 方法和 `provider` 中的 `hello` 接口绑定在一起。需要注意的是，在 `SpringMVC` 中，在需要给参数设置默认值或者要求参数必填的情况下才需要用到 `@RequestParam` 注解，而在这里，这个注解一定要加。

经过这样的步骤之后，我们就可以在一个 **Controller** 中注入 **HelloService** 接口并使用它了，而 **HelloService** 接口也会去调用相关的服务。我的 **Controller** 如下：

```
@RestController
public class HelloController {
    @Autowired
    HelloService helloService;

    @GetMapping("/hello")
    public String hello(String name) {
        return helloService.hello(name);
    }
}
```

此时，启动 **feign-consumer**，然后在浏览器中访问 <http://localhost:5002/hello?name=纯洁的微笑>，显示效果如下：



hello 纯洁的微笑！

这样，我们就成功在 **feign-consumer** 中调用了 **provider** 提供的服务了，是不是比之前手写 **RestTemplate** 调用要方便很多？

参数传递

上面和大家展示了 **Feign** 的基本使用。接下来和大家说说 **Feign** 中的参数传递问题，相信对于大多数人而言，在开发中，参数传递无非就是 **key/value** 形式的参数、放在 **body** 中的参数、放在 **Url** 路径上的参数以及放在请求头上的参数，这四种是较为常见的四种传参方式，因此，这里就重点和大家分享下这四种不同的传参方式在 **Feign** 中要如何使用。

首先，在 **Feign** 工程中再创建一个 **Maven** 子工程叫做 **commons**，在 **commons** 中定义一个 **User** 类，如下：

```
public class User {
    private Long id;
    private String username;
    private String address;
    //省略getter/setter
}
```

然后在 **provider** 和 **feign-consumer** 的 **pom.xml** 文件中分别引用该工程，如下：

```
<dependency>
  <groupId>com.justdojava</groupId>
  <artifactId>commons</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>
```

接下来，在 **provider** 中添加 **UserController** 类，并添加如下接口，各个接口分别使用不同的方式来接收参数：

```

@RestController
public class UserController {
    @DeleteMapping("/user/{id}")
    public void deleteUserById(@PathVariable Long id) {
        System.out.println(id);
    }
    @GetMapping("/user")
    public User getUserByName(@RequestParam String name) {
        User user = new User();
        user.setUsername(name);
        return user;
    }
    @PostMapping("/user")
    public User addUser(@RequestBody User user) {
        return user;
    }
    @PutMapping("/user")
    public void updateUserById(@RequestHeader String name, @RequestHeader Long id) {
        System.out.println("name:" + name + ";id:" + id);
    }
}

```

关于上面这一段代码，我说如下几点：

1. 简单起见，这里我在 **provider** 中并未调用数据库，只是将调用的模拟结果打印出来或者返回去，证明接口调通了即可；
2. 在第一个删除方法中，将参数放在 **Url** 路径中；
3. 在第二个查询方法中，将参数以 **key/value** 的形式传递；
4. 在第三个添加数据的方法中，参数将以请求体的方式传递；
5. 在第四个更新的方法中，参数以请求头的方式传递，这种传递方式注意不能传递中文，如果是中文参数，在传递之前，可以使用 `URLEncoder.encode()` 方法对参数进行编码，在接收到参数之后，使用 `URLDecoder.decode()` 方法对参数进行解码。

provider 定义完成后，接下来重新启动 **provider**，然后我们继续在 **feign-consumer** 中添加调用代码，如下：

```

@FeignClient("provider")
public interface HelloService {
    @GetMapping("/hello")
    String hello(@RequestParam("name") String name);
    @DeleteMapping("/user/{id}")
    void deleteUserById(@PathVariable("id") Long id);
    @GetMapping("/user")
    User getUserByName(@RequestParam("name") String name);
    @PostMapping("/user")
    User addUser(@RequestBody User user);
    @PutMapping("/user")
    void updateUserById(@RequestHeader("name") String name, @RequestHeader("id") Long id);
}

```

这里的调用，基本也和前面一样，无需赘述，但是有一个地方需要强调，那就是不同于 **provider**，这里的参数如果是 **key/value** 形式的，一定要在 `@RequestParam` 注解中指明 **name** 属性，如果是在 **header** 中传递的，则一定要在 `@RequestHeader` 注解中添加 **name** 属性，如果参数放在 **Url** 路径中，那么一定需要在 `@PathVariable` 注解中添加 **name** 属性指明参数名称。

配置完成后，在 **feign-consumer** 的 **HelloController** 中添加一个接口用来测试，如下：

```

@GetMapping("/hello2")
public void hello2() {
    helloService.deleteUserById(99L);
    User u2 = helloService.getUserByName("纯洁的微笑");
    System.out.println(u2);
    u2.setId(98L);
    u2.setAddress("深圳");
    User u3 = helloService.addUser(u2);
    System.out.println(u3);
    helloService.updateUserById("lenve", 99L);
}

```

调用该接口，在 `provider` 和 `feign-consumer` 中都会有日志打印出来，如下图：

```

2019-03-29 22:23:56.663 INFO
99
name:lenve;id:99

```

```

User {id=null, username='纯洁的微笑', address='null'}
User {id=98, username='纯洁的微笑', address='深圳'}
2019-03-29 22:28:49.065 INFO 11172 --- [main-executor

```

看到如上两张图，表示我们的接口已经调用成功啦！

小结

本节主要通过两个案例向读者介绍了声明式服务调用的一个基本用法。相信有一部分读者在读完本节后，会很容易想到 `MyBatis`，很多人在刚开始学习 `MyBatis` 的时候，并不是一上来就使用 `Mapper`，而是先从简单的 `SqlSessionFactory` 开始，在使用的过程中，发现了大量的模板化代码，于是才有了 `Mapper`，使用了动态代理启动生成调用逻辑，开发者只需要生命最最关键的部分。`MyBatis` 中的这一演变过程和我们这里的演变如出一辙，直接使用 `RestTemplate` 也带来了大量的模板化代码，通过 `Feign`，我们只需要定义一下方法中最关键的部分，就能实现调用。那么有人要问了，用了 `Feign`，我们在 `RestTemplate` 中使用的负载均衡还能继续用吗？答案当然是可以继续使用，关于这个问题以及 `Feign` 的一些高级用法，我将在下篇文章中和大伙儿分享。

本文作者：纯洁的微笑、江南一点雨