第5章 chapter 5

OpenFeign的原理与使用

本章学习目标

- 理解OpenFeign的概念和作用
- 掌握OpenFeign的基本用法和配置
- 熟悉OpenFeign的请求拦截器和错误处理机制
- 了解OpenFeign的内置负载均衡支持及其原理
- 掌握OpenFeign的高级用法，如动态URL、文件上传、并发访问等

本章准备工作

开发人员需要提前准备的开发环境和开发工具包括IDEA、JDK 11+、Maven 3.0+、Nacos 2.1.0+、MySQL 5.6.5+。

本章主要介绍了OpenFeign，它是一种基于接口的声明式Web服务客户端，可以使编写Web服务客户端变得更加容易。OpenFeign的使用方式类似Spring MVC的使用方式，用户只需定义一个接口并注解它，OpenFeign就会自动为这个接口创建一个实现类。同时，本章还介绍了OpenFeign的一些高级特性，包括发现和注册服务、请求超时和重试、文件上传和下载、并发访问和线程池配置、请求拦截器，以及内置负载均衡等内容。

5.1 OpenFeign介绍

5.1.1 服务间调用

在微服务架构中，服务之间需要进行相互调用和通信，以处理业务逻辑和共享数据。服务间调用问题是微服务架构的一个重要问题，主要包括以下方面。

1. 发现和注册服务

服务需要被注册到注册中心，以便其他服务可以发现并调用它们。发现和注册服务

通常需要由第三方工具实现，如Consul、ZooKeeper、Eureka和Nacos等。

2. 服务路由和负载均衡

服务之间的相互调用需要服务路由和负载均衡辅助，以实现请求的高可用和负载均衡。常用的负载均衡算法有轮询、随机和加权轮询等。

3. 请求超时和重试

由于服务间调用可能存在网络延迟或服务不可用等问题，因此需要实现请求超时和重试机制，以提高服务的可用性和稳健性。

4. 错误处理和日志记录

服务之间的相互调用可能存在各种错误，如网络异常、服务不可用等，因此需要实现错误处理和日志记录，以便进行故障排查和定位问题。

综上所述，开发者急需一种简单、方便、可靠的方式解决以上问题，作为一种轻量级的RESTful客户端工具，Feign提供了一种简单、方便的方式以进行服务之间的相互调用，为开发者带来了很大的便利。随着OpenFeign的出现，它更是提供了丰富的功能和配置选项，包括发现服务、负载均衡、请求超时和重试、错误处理和日志记录等，为开发人员在服务间调用方面提供了更加完善的解决方案。

5.1.2 Feign与OpenFeign

Feign是一个基于Java的RESTful客户端工具，它可以帮助开发者更加方便地实现服务间调用。由于Feign采用自动生成接口的方式，开发者只需要定义服务接口和相应的注解即可实现对远程服务调用。在使用Feign时，开发者只需要配置服务接口和调用方式，无须关心底层的实现细节，这可以使Feign具有较高的开发效率和易用性。

OpenFeign是一个开源的、基于Java的HTTP客户端，是Netflix公司开源的项目之一，主要用于简化基于HTTP的微服务间通信。它是Feign的升级版本(8.18.0之后，Nefflix将其捐赠给Spring Cloud社区，并更名为OpenFeign，OpenFeign的第一个版本就是9.0.0)，是一个基于Feign的RESTful客户端工具，并且具有更加强大的功能和配置选项。OpenFeign支持多种协议和编解码器，可以进行服务发现、负载均衡、请求超时和重试、错误处理和日志记录等操作。OpenFeign还支持自定义配置和扩展，开发人员可以根据自己的需求定制化配置，以适应不同的业务场景。OpenFeign的出现进一步提高了服务间调用的效率和可靠性，为开发者在服务间调用方面提供了更加完善的解决方案。

OpenFeign使开发者可以在不了解底层HTTP的情况下访问其他服务的接口，它通过自动生成RESTful API的方式隐藏底层HTTP细节。这样，开发者可以更专注于业务逻辑的实现，而不需要过多关注服务间通信细节。


OpenFeign的原理

5.2 OpenFeign的原理

5.2.1 动态代理技术

OpenFeign的原理是基于接口的动态代理技术自动生成RESTful API的客户端代码。其基本的功能如下。

1. 使用Java接口定义RESTful API

开发者在Java接口中定义服务端提供的RESTful API，包括接口的路径、请求方法、请求参数、返回值等信息。

2. 使用Feign注解标记接口

开发者使用Feign注解标记Java接口，以告知OpenFeign生成对应RESTful API的客户端代码。

3. Feign动态代理

OpenFeign使用基于接口的动态代理技术自动生成Java客户端代码，该客户端代码包含了与服务端交互所需的HTTP请求参数和请求头等信息。

4. 发送HTTP请求

在客户端代码中，OpenFeign封装了发送HTTP请求的过程，包括请求头、请求体等信息，并使用Apache HttpClient或OkHttp等HTTP客户端库发送HTTP请求。

5. 解析HTTP响应

当服务端响应HTTP请求时，OpenFeign将自动解析响应内容，并将其转换为Java对象，以便开发者在应用程序中使用。

通过这种方式，OpenFeign可以使开发者在不了解底层HTTP的情况下直接使用Java接口调用远程服务，从而简化微服务架构中的服务间通信。

5.2.2 请求拦截器

OpenFeign提供了请求拦截器(request interceptor)的机制，可以在请求发出前和响应返回后对请求和响应进行拦截和处理。这个机制在一些场景下非常有用，如下所示。

(1) 统一添加请求头，如用户鉴权、TraceId等。

(2) 对请求记录日志，便于排查问题。

(3) 对请求签名、加密。

(4) 处理返回结果的错误，如处理全局异常、包装结果等。

OpenFeign的请求拦截器是一个接口，名为RequestInterceptor，它有两个方法，如下所示。

```
public interface RequestInterceptor {
    void apply(RequestTemplate template);
    boolean match(RequestTemplate template);
}
```

其中，apply()方法是必须实现的，它接收一个RequestTemplate对象，该对象包含了请求的所有信息，如请求方法、请求地址、请求头、请求体等。开发者可以在这个方法中对RequestTemplate进行修改，如添加请求头、修改请求地址等。

match()方法是可选实现的，它用于对请求进行过滤，只有返回true的请求才会被拦截器处理。这个方法可被用于一些特殊的场景，如只拦截某个特定的URL。

下面是一个简单的示例，它演示了如何使用RequestInterceptor对请求进行拦截和处理。

```
@Component
public class AuthInterceptor implements RequestInterceptor {
    @Override
    public void apply(RequestTemplate template) {
        // 在请求头中添加 Authorization
        template.header("Authorization", "Bearer " +getToken());
    }
    private String getToken() {
        // 获取用户 Token
        // ...
    }
}
```

这个示例定义了一个AuthInterceptor拦截器，它在请求头中添加了一个Authorization字段，字段值为用户的Token。这个Token是由调用getToken()方法获取的。

要使用拦截器，只需要在Feign客户端接口添加一个@FeignClient注解，指定interceptor属性为一个拦截器列表即可，如下所示。

```
@FeignClient(name ="example-service", configuration =ExampleConfiguration.
class, interceptor ={AuthInterceptor.class})
public interface ExampleFeignClient {
    // ...
}
```

这个示例指定了一个AuthInterceptor拦截器，它将被应用到ExampleFeignClient客户端的所有请求中。

5.2.3 内置的负载均衡支持

在 Spring Cloud 2020 版本发布之后，当开发者使用 Feign 调用服务时，内部实现默认使用的是 Spring Cloud LoadBalancer 进行负载均衡，并且默认的 HTTP 客户端是 Spring WebClient，而不是之前的 RestTemplate。在没有手动显式地写@LoadBalanced 注解的情况下，Feign 会在发送请求时使用 Spring WebClient，并通过 Spring Cloud LoadBalancer 实现负载均衡。这是因为在 Spring Cloud 2020 版本中，RestTemplate 的使用已经被 Spring 官方宣布为过时，并且官方建议使用 Spring WebClient 代替之。因此，使用 Feign 调用服务时，官方建议显式地在配置文件中指定 HTTP 客户端，并使用@LoadBalanced 注解声明客户端支持负载均衡。

Spring Cloud 2020 采用了 Spring Cloud LoadBalancer 作为其内置的负载均衡组件。Spring Cloud LoadBalancer 是 Spring Cloud 社区开发的负载均衡组件，其实现了与 Spring Cloud 集成的自动化配置和默认配置，并提供了一些定制化的负载均衡策略。

Spring Cloud LoadBalancer 提供了以下几个核心组件。

（1）LoadBalancerClient：用于从负载均衡器中获取服务的地址。

（2）ServiceInstanceListSupplier：用于获取服务实例列表。

（3）LoadBalancerInterceptor：用于拦截 RestTemplate 和 WebClient 发送的请求，从而实现自动化负载均衡。

Spring Cloud LoadBalancer 默认采用轮询的负载均衡策略，支持开发者通过自定义实现 LoadBalancerClient 接口扩展其他的负载均衡策略，如随机、最小并发等。

当使用 Spring Cloud OpenFeign 调用服务时，开发者可以在 FeignClient 上添加@LoadBalanced 注解以启用 Spring Cloud LoadBalancer 的负载均衡功能，从而实现自动化的负载均衡。

在使用 RestTemplate 和 WebClient 调用服务时，开发者可以不在创建 RestTemplate 和 WebClient 时添加 @LoadBalanced 注解，而是直接使用 Spring Cloud LoadBalancer 提供的拦截器 LoadBalancerInterceptor 以实现自动化负载均衡。这样，即使更改了服务实例的地址，开发者也不需要修改客户端的代码，因为 LoadBalancerInterceptor 会在每次发送请求时从负载均衡器中获取最新的服务实例地址。

5.3 使用 OpenFeign

1. 定义服务端 RESTful API

编写一个 Java 接口，定义服务端提供的 RESTful API，其中，@FeignClient 注解指定了服务端的名称和地址，告知 OpenFeign 生成对应的客户端代码。

```
@FeignClient(name ="example-service", url ="http://localhost:8080")
public interface ExampleServiceClient {
    @GetMapping("/example/{id}")
    ExampleDto getExampleById(@PathVariable("id") Long id);
    @PostMapping("/example")
    ExampleDto createExample(@RequestBody ExampleDto example);
}
```

2. 调用服务端 RESTful API

在应用程序中，开发者可以通过自动注入 ExampleServiceClient 接口的方式调用服务端提供的 RESTful API。

```
@RestController
public class ExampleController {
    @Autowired
    private ExampleServiceClient exampleServiceClient;
    @GetMapping("/example/{id}")
    public ExampleDto getExampleById(@PathVariable Long id) {
        return exampleServiceClient.getExampleById(id);
    }
    @PostMapping("/example")
    public ExampleDto createExample(@RequestBody ExampleDto exampleDto) {
        return exampleServiceClient.createExample(exampleDto);
    }
}
```

通过这种方式，开发者可以直接使用 Java 接口调用远程服务端提供的 RESTful API，而无须了解底层 HTTP 的细节。同时，OpenFeign 提供了灵活的配置方式，使开发者可以自定义 HTTP 请求参数、请求头、请求体等信息，以满足不同的需求。

5.4 OpenFeign 的使用场景


OpenFeign的使用和扩展

下文将结合具体的场景讲解 OpenFeign 的使用场景。

使用一个开源的在线仿冒 API 地址 https://jsonplaceholder.typicode.com 作为提供者（被调用服务），可以使用的测试 API 如下。

```
GET     /posts
GET     /posts/1
GET     /posts/1/comments
GET     /comments?postId=1
POST    /posts
```

```
PUT      /posts/1
PATCH    /posts/1
DELETE   /posts/1
```

1. 引入项目依赖

新建一个 POM 文件(可以使用 spring.io 官方初始化项目的在线工具配置所需要的依赖,也可以直接使用如下 POM 文件内容)。

```
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <parent>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-parent</artifactId>
      <version>2.7.8</version>
      <relativePath/><!--lookup parent from repository -->
   </parent>
   <groupId>com.etoak.tutorial.openfeign</groupId>
   <artifactId>client</artifactId>
   <version>0.0.1-SNAPSHOT</version>
   <name>client</name>
   <description>OpenFeign 示例代码的客户端(消费端)</description>
   <properties>
      <java.version>11</java.version>
      <spring-cloud.version>2021.0.5</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-openfeign</artifactId>
      </dependency>

      <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-test</artifactId>
```

```
            <scope>test</scope>
        </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>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

2. 定义数据传递结构

上述 API 中的示例返回的数据如下。

```
{
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi optio
reprehenderit",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et
    cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem
    sunt rem eveniet architecto"
  }
```

所以基于上面响应的数据可以定义如下结构。

```
public class Post {
    private Long id;
    private String userId;
```

```
    private String title;
    private String body;
  }
```

3. 定义 FeinClient

```
@FeignClient(name ="jsonplaceholder", url ="https://jsonplaceholder.
typicode.com", contextId ="jsonplaceholder")
public interface Api {
   @GetMapping("/posts")
   List<Post>getAllPosts();

   @GetMapping("/posts/{postId}")
   Post getPostById(@PathVariable Long postId);

   @GetMapping("/posts")
   List<Post>getPostByUserId(@RequestParam Long userId);

   @PostMapping("/posts")
   Post createPost(Post post);

   @PutMapping("/posts")
   Post updatePost(Post post);

   @DeleteMapping("/posts/{postId}")
   Post deletePost(@PathVariable Long postId);
}
```

在上面代码中，@FeignClient 的参数使用了 name 和 url，url 指定了调用服务的全路径，url 属性经常被用于本地测试。如果同时指定 name/value 和 url 属性，则以 url 属性为准，name/value 属性指定的值便被当作微服务里的服务名称。

4. 启动入口

```
@SpringBootApplication
@EnableFeignClients
public class ClientApplication {
   public static void main(String[] args) {
         ConfigurableApplicationContext context = SpringApplication. run
         (ClientApplication.class, args);

      Api api =context.getBean(Api.class);
```

```
        //调用 FeignClient
        List<Post>allPosts =api.getAllPosts();

        //输出以下响应结果
        for (Post post : allPosts) {
          System.out.println("id:" +post.getId() +"," +
                "userId:" +post.getUserId() +"\n" +
                "title:" +post.getTitle() +"\n\n" +
                post.getBody());
          System.out.println("--------分隔线 --------");
        }
    }
}
```

执行完之后，控制台输出核心部分日志截取如下。

```
sunt aut facere repellat provident occaecati excepturi optio
reprehenderitquia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto
--------分隔线 --------
2  1  qui est esse est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla
--------分隔线 --------
3  1  ea molestias quasi exercitationem repellat qui ipsa sit aut et iusto sed
quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut
--------分隔线 --------
4  1  eum et est occaecati ullam et saepe reiciendis voluptatem adipisci
sit amet autem assumenda provident rerum culpa
quis hic commodi nesciunt rem tenetur doloremque ipsam iure
quis sunt voluptatem rerum illo velit
```

5.5 配置属性的解析

OpenFeign 的核心主要分为两部分：注解处理器和动态代理。

注解处理器是 OpenFeign 的核心之一，它通过处理 Java 接口中的注解生成对应的 HTTP 请求代码。当开发者在 Java 接口中使用@FeignClient、@RequestMapping、