微服务调用组件Feign

JAVA 项目中如何实现接口调用?

1）Httpclient

HttpClient 是 Apache Jakarta Common 下的子项目，用来提供高效的、最新的、功能丰富的支持 Http 协议的客户端编程工具包，并且它支持 HTTP 协议最新版本和建议。HttpClient 相比传统 JDK 自带的 URLConnection，提升了易用性和灵活性，使客户端发送 HTTP 请求变得容易，提高了开发的效率。

2）Okhttp

一个处理网络请求的开源项目，是安卓端最火的轻量级框架，由 Square 公司贡献，用于替代 HttpUrlConnection 和 Apache HttpClient。OkHttp 拥有简洁的 API、高效的性能，并支持多种协议（HTTP/2 和 SPDY）。

3）HttpURLConnection

HttpURLConnection 是 Java 的标准类，它继承自 URLConnection，可用于向指定网站发送 GET 请求、POST 请求。HttpURLConnection 使用比较复杂，不像 HttpClient 那样容易使用。

4）RestTemplate WebClient

RestTemplate 是 Spring 提供的用于访问 Rest 服务的客户端，RestTemplate 提供了多种便捷访问远程 HTTP 服务的方法，能够大大提高客户端的编写效率。

上面介绍的是最常见的几种调用接口的方法，我们下面要介绍的方法比上面的更简单、方便，它就是 Feign。

1. 什么是Feign

Feign是Netflix开发的声明式、模板化的HTTP客户端，其灵感来自Retrofit、JAXRS-2.0以及WebSocket。Feign可帮助我们更加便捷、优雅地调用HTTP API。
Feign支持多种注解，例如Feign自带的注解或者JAX-RS注解等。

Spring Cloud openfeign对Feign进行了增强，使其支持Spring MVC注解，另外还整合了Ribbon和Nacos，从而使得Feign的使用更加方便

1.1 优势

Feign可以做到使用 HTTP 请求远程服务时就像调用本地方法一样的体验，开发者完全感知不到这是远程方法，更感知不到这是个 HTTP 请求。它像 Dubbo 一样，consumer 直接调用接口方法调用 provider，而不需要通过常规的 Http Client 构造请求再解析返回数据。它解决了让开发者调用远程接口就跟调用本地方法一样，无需关注与远程的交互细节，更无需关注分布式环境开发。

2. Spring Cloud Alibaba快速整合OpenFeign

1）引入依赖

```
<!-- openfeign 远程调用 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

2）编写调用接口+@FeignClient注解

```
@FeignClient(value = "mall-order",path = "/order")
public interface OrderFeignService {

    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);
}
```

3）调用端在启动类上添加@EnableFeignClients注解

```
@SpringBootApplication
@EnableFeignClients
public class MallUserFeignDemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(MallUserFeignDemoApplication.class, args);
    }
}
```

4）发起调用，像调用本地方式一样调用远程服务

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

    @Autowired
    OrderFeignService orderFeignService;

    @RequestMapping(value = "/findOrderByUserId/{id}")
    public R  findOrderByUserId(@PathVariable("id") Integer id) {
        //feign调用
        R result = orderFeignService.findOrderByUserId(id);
        return result;
    }
}
```

3. Spring Cloud Feign的自定义配置及使用

Feign 提供了很多的扩展机制，让用户可以更加灵活的使用。

3.1 日志配置

有时候我们遇到 Bug，比如接口调用失败、参数没收到等问题，或者想看看调用性能，就需要配置 Feign 的日志了，以此让 Feign 把请求信息输出来。

1）定义一个配置类，指定日志级别

```
// 注意：  此处配置@Configuration注解就会全局生效，如果想指定对应微服务生效，就不能配置
public class FeignConfig {
    /**
     * 日志级别
     *
     * @return
     */
    @Bean
    public Logger.Level feignLoggerLevel() {
        return Logger.Level.FULL;
    }
}
```

通过源码可以看到日志等级有 4 种，分别是：

- **NONE【性能最佳，适用于生产】**：不记录任何日志（默认值）。

- **BASIC【适用于生产环境追踪问题】**：仅记录请求方法、URL、响应状态代码以及执行时间。
- **HEADERS**：记录BASIC级别的基础上，记录请求和响应的header。
- **FULL【比较适用于开发及测试环境定位问题】**：记录请求和响应的header、body和元数据。

2) 局部配置，让调用的微服务生效，在@FeignClient 注解中指定使用的配置类

```
@FeignClient(value = "mall-order",path = "/order",configuration = FeignConfig.class)
public interface OrderFeignService {

    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);
}
```

指定configuration，注意：FeignConfig不能添加@Configuration

3) 在yml配置文件中执行 Client 的日志级别才能正常输出日志，格式是"logging.level.feign接口包路径=debug"

```
logging:
  level:
    com.tuling.mall.feigndemo.feign: debug
```

测试：BASIC级别日志

```
: [OrderFeignService#findOrderByUserId] ---> GET http://mall-order/order/findOrderByUserId/1 HTTP/1.1
: [OrderFeignService#findOrderByUserId] <--- HTTP/1.1 200 (11ms)
```

补充：局部配置可以在yml中配置

对应属性配置类：

org.springframework.cloud.openfeign.FeignClientProperties.FeignClientConfiguration

```
feign:
  client:
    config:
      mall-order:  #对应微服务
        loggerLevel: FULL
```

3.2 契约配置

Spring Cloud 在 Feign 的基础上做了扩展，使用 Spring MVC 的注解来完成Feign的功能。原生的 Feign 是不支持 Spring MVC 注解的，如果你想在 Spring Cloud 中使用原生的注解方式来定义客户端也是可以的，通过配置契约来改变这个配置，Spring Cloud 中默认的是 SpringMvcContract。

Spring Cloud 1 早期版本就是用的原生Fegin. 随着netflix的停更替换成了Open feign

1）修改契约配置，支持Feign原生的注解

```
/**
 * 修改契约配置，支持Feign原生的注解
 * @return
```

```
 */
@Bean
public Contract feignContract() {
    return new Contract.Default();
}
```

注意：修改契约配置后，OrderFeignService 不再支持springmvc的注解，需要使用Feign原生的注解

2）OrderFeignService 中配置使用Feign原生的注解

```
@FeignClient(value = "mall-order",path = "/order")
public interface OrderFeignService {
    @RequestLine("GET /findOrderByUserId/{userId}")
    public R findOrderByUserId(@Param("userId") Integer userId);
}
```

3）补充，也可以通过yml配置契约

```
feign:
  client:
    config:
      mall-order:  #对应微服务
        loggerLevel: FULL
        contract: feign.Contract.Default   #指定Feign原生注解契约配置
```

3.3自定义拦截器实现认证逻辑

```
public class FeignAuthRequestInterceptor implements RequestInterceptor {
    @Override
    public void apply(RequestTemplate template) {
        // 业务逻辑
        String access_token = UUID.randomUUID().toString();
        template.header("Authorization",access_token);
    }
}

@Configuration  // 全局配置
public class FeignConfig {
    @Bean
    public Logger.Level feignLoggerLevel() {
```

```
        return Logger.Level.FULL;
    }
    /**
     * 自定义拦截器
     * @return
     */
    @Bean
    public FeignAuthRequestInterceptor feignAuthRequestInterceptor(){
        return new FeignAuthRequestInterceptor();
    }
}
```

测试

```
eignService#findOrderByUserId] ---> GET http://mall-order/order/findOrderByUserId/1 HTTP/1.1
eignService#findOrderByUserId] Authorization: 09558987-0e31-409b-b808-15663176a375
eignService#findOrderByUserId] ---> END HTTP (0-byte body)
eignService#findOrderByUserId] <--- HTTP/1.1 200 (60ms)
eignService#findOrderByUserId] connection: keep-alive
eignService#findOrderByUserId] content-type: application/json
```

补充：可以在yml中配置

```
feign:
  client:
    config:
      mall-order:  #对应微服务
        requestInterceptors[0]:  #配置拦截器
          com.tuling.mall.feigndemo.interceptor.FeignAuthRequestInterceptor
```

mall-order端可以通过 @RequestHeader获取请求参数

建议在filter,interceptor中处理

3.4 超时时间配置

通过 Options 可以配置连接超时时间和读取超时时间，Options 的第一个参数是连接的超时时间（ms），默认值是 2s；第二个是请求处理的超时时间（ms），默认值是 5s。

全局配置

```
@Configuration
public class FeignConfig {
    @Bean
    public Request.Options options() {
        return new Request.Options(5000, 10000);
    }
}
```

yml中配置

```yaml
feign:
  client:
    config:
      mall-order:  #对应微服务
        # 连接超时时间，默认2s
        connectTimeout: 5000
        # 请求处理超时时间，默认5s
        readTimeout: 10000
```

补充说明： Feign的底层用的是Ribbon，但超时时间以Feign配置为准

测试超时情况：

```
java.net.SocketTimeoutException: Read timed out
    at java.net.SocketInputStream.socketRead0(Native Method) ~[na:1.8.0_181]
    at java.net.SocketInputStream.socketRead(SocketInputStream.java:116) ~[na:1.8.0_181]
    at java.net.SocketInputStream.read(SocketInputStream.java:171) ~[na:1.8.0_181]
    at java.net.SocketInputStream.read(SocketInputStream.java:141) ~[na:1.8.0_181]
```

返回结果

```
{
    "timestamp": "2021-01-30T13:24:25.589+0000",
    "status": 500,
    "error": "Internal Server Error",
    "message": "Read timed out executing GET http://mall-order/order/findOrderByUserId/1",
    "path": "/user/findOrderByUserId/1"
}
```

3.5 客户端组件配置

Feign 中默认使用 JDK 原生的 URLConnection 发送 HTTP 请求，我们可以集成别的组件来替换掉 URLConnection，比如 Apache HttpClient，OkHttp。

Feign发起调用真正执行逻辑：**feign.Client#execute （扩展点）**

```java
@Override
public Response execute(Request request, Options options) throws IOException {
  HttpURLConnection connection = convertAndSend(request, options);
  return convertResponse(connection, request);
}
```

3.5.1 配置Apache HttpClient

引入依赖

```xml
<!-- Apache HttpClient -->
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
```

```
    <artifactId>httpclient</artifactId>
    <version>4.5.7</version>
</dependency>
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-httpclient</artifactId>
    <version>10.1.0</version>
</dependency>
```

然后修改yml配置，将 Feign 的 Apache HttpClient启用：

```
feign:
  #feign 使用 Apache HttpClient  可以忽略，默认开启
  httpclient:
    enabled: true
```

关于配置可参考源码： org.springframework.cloud.openfeign.FeignAutoConfiguration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(ApacheHttpClient.class)
@ConditionalOnMissingClass("com.netflix.loadbalancer.ILoadBalancer")
@ConditionalOnMissingBean(CloseableHttpClient.class)
@ConditionalOnProperty(value = "feign.httpclient.enabled", matchIfMissing = true)
protected static class HttpClientFeignConfiguration {
```

测试：调用会进入feign.httpclient.ApacheHttpClient#execute

3.5.2 配置 OkHttp

引入依赖

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-okhttp</artifactId>
</dependency>
```

然后修改yml配置，将 Feign 的 HttpClient 禁用，启用 OkHttp，配置如下：

```
feign:
  #feign 使用 okhttp
  httpclient:
    enabled: false
  okhttp:
    enabled: true
```

关于配置可参考源码： org.springframework.cloud.openfeign.FeignAutoConfiguration

```java
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(OkHttpClient.class)
@ConditionalOnMissingClass("com.netflix.loadbalancer.ILoadBalancer")
@ConditionalOnMissingBean(okhttp3.OkHttpClient.class)
@ConditionalOnProperty("feign.okhttp.enabled")
protected static class OkHttpFeignConfiguration {

    private okhttp3.OkHttpClient okHttpClient;
```

测试：调用会进入feign.okhttp.OkHttpClient#execute

3.6 GZIP 压缩配置

开启压缩可以有效节约网络资源，提升接口性能，我们可以配置 GZIP 来压缩数据：

```yaml
feign:
  # 配置 GZIP 来压缩数据
  compression:
    request:
      enabled: true
      # 配置压缩的类型
      mime-types: text/xml,application/xml,application/json
      # 最小压缩值
      min-request-size: 2048
    response:
      enabled: true
```

注意：只有当 Feign 的 Http Client 不是 okhttp3 的时候，压缩才会生效，配置源码在 FeignAcceptGzipEncodingAutoConfiguration

```java
@Configuration(proxyBeanMethods = false)
@EnableConfigurationProperties(FeignClientEncodingProperties.class)
@ConditionalOnClass(Feign.class)
@ConditionalOnBean(Client.class)
@ConditionalOnProperty(value = "feign.compression.response.enabled",
        matchIfMissing = false)
// The OK HTTP client uses "transparent" compression.
// If the accept-encoding header is present it disable transparent compression
@ConditionalOnMissingBean(type = "okhttp3.OkHttpClient")
@AutoConfigureAfter(FeignAutoConfiguration.class)
public class FeignAcceptGzipEncodingAutoConfiguration {

    @Bean
    public FeignAcceptGzipEncodingInterceptor feignAcceptGzipEncodingInterceptor(
            FeignClientEncodingProperties properties) {
        return new FeignAcceptGzipEncodingInterceptor(properties);
    }
```

核心代码就是 @ConditionalOnMissingBean（type="okhttp3.OkHttpClient"），表示 Spring BeanFactory 中不包含指定的 bean 时条件匹配，也就是没有启用 okhttp3 时才会进行压缩配置。