主讲老师：Fox老师

> 有道笔记：https://note.youdao.com/s/DGcP0Jpk

1. LoadBalancer+RestTemplate的缺陷

LoadBalancer+RestTemplate进行微服务调用

```
@Bean
@LoadBalanced
public RestTemplate restTemplate() {
    return new RestTemplate();
}

//调用方式
String url = "http://mall-order/order/findOrderByUserId/"+id;
R result = restTemplate.getForObject(url,R.class);
```

思考： 这种方式进行微服务调用存在什么问题?

- 代码可读性差，编程体验不统一
- 参数复杂时URL难以维护

2. 微服务调用组件Spring Cloud OpenFeign实战

2.1 什么是Spring Cloud OpenFeign

Feign是Netflix开发的声明式、模板化的HTTP客户端，Feign可帮助我们更加便捷、优雅地调用HTTP API。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。

```
//本地调用
R result = orderService.findOrderByUserId(id);
//openFeign远程调用  orderService为代理对象
R result = orderService.findOrderByUserId(id);
```

Spring Cloud OpenFeign对Feign进行了增强，使其支持Spring MVC注解，从而使得Feign的使用更加方便。

> 官方文档： https://docs.spring.io/spring-cloud-openfeign/docs/current/reference/html/

2.2 微服务快速整合OpenFeign实战

1）引入依赖

微服务调用者引入OpenFeign依赖

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

2）在启动类上添加@EnableFeignClients注解，开启openFeign功能

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

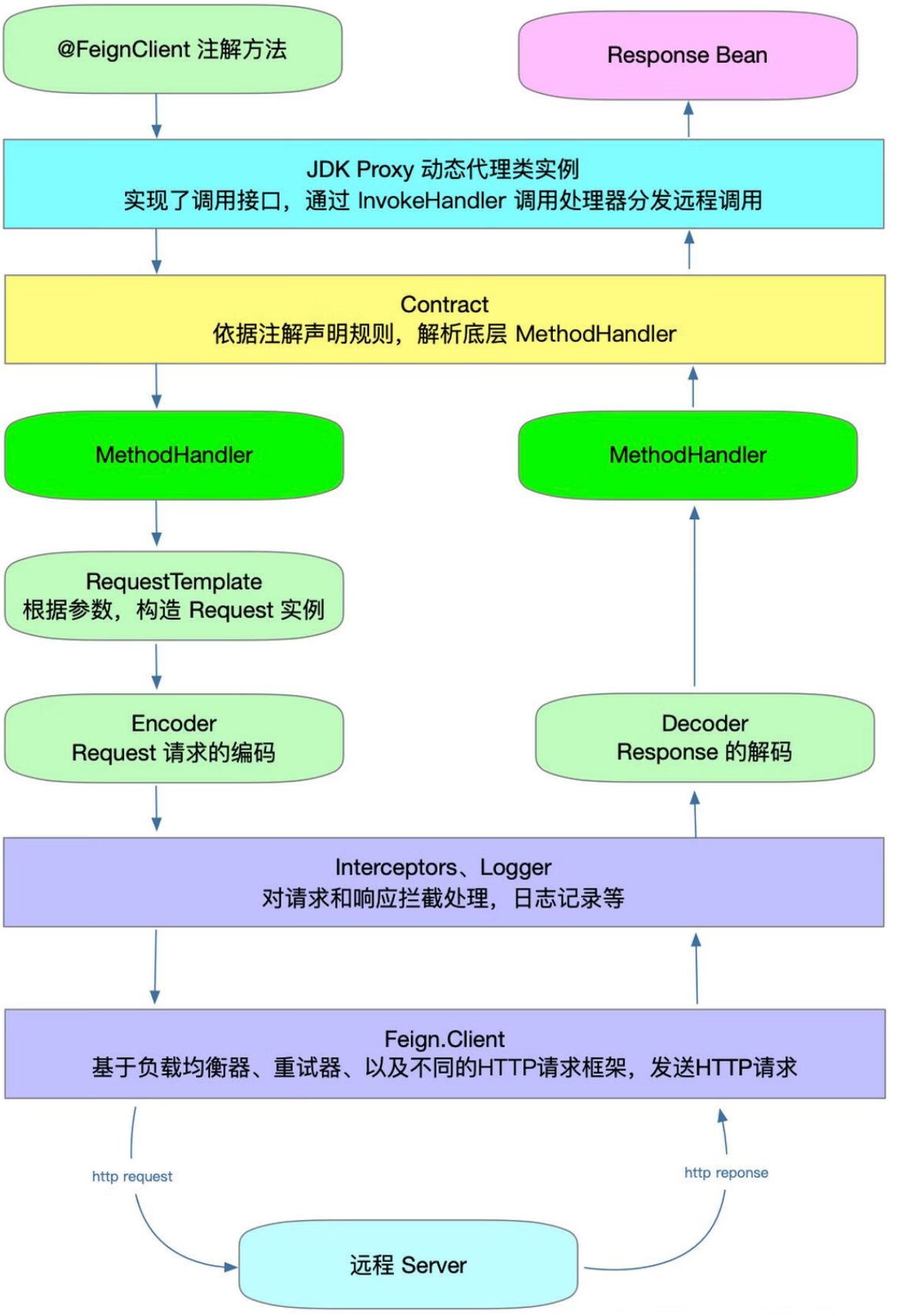
3）编写OpenFeign客户端

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

    //基于SpringMvc的注解来声明远程调用信息
    @RequestMapping("/findOrderByUserId/{userId}")
    public R findOrderByUserId(@PathVariable("userId") Integer userId);
}
```

4）微服务调用者发起调用，像调用本地方式一样调用远程微服务提供者

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

    @Autowired
    OrderFeignService orderFeignService;

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

2.3 OpenFeign的调用流程



2.4 OpenFeign扩展优化实战

Feign 提供了很多的扩展机制，让用户可以更加灵活的使用。

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。

Java Bean配置方式

方式1：利用@Configuration实现全局生效，对所有的微服务调用者都生效

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@Configuration
@Configuration
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，默认值】**：不记录任何日志。
- **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) 在application.yml配置文件中配置 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
```

3) 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)
```

方式2： 局部生效，让指定的微服务生效，在@FeignClient 注解中指定configuration

```
@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

yml配置文件配置方式

- 全局生效：配置 **{服务名}** 为 **default**
- 局部生效：配置 **{服务名}** 为 **具体服务名**

方式3：全局生效，对所有的微服务调用者都生效

```yaml
spring:
  cloud:
    openfeign:
      client:
        config:
          default:
            loggerLevel: FULL
```

方式4：局部生效，yml中对调用的微服务提供者进行配置

对应属性配置类：

org.springframework.cloud.openfeign.FeignClientProperties.FeignClientConfiguration

```yaml
spring:
  cloud:
    openfeign:
      client:
        config:
          mall-order:  #对应微服务
            loggerLevel: FULL
```

超时时间配置

OpenFeign使用两个超时参数:

- connectTimeout 可以防止由于较长的服务器处理时间而阻塞调用者。
- readTimeout 从连接建立时开始应用，当返回响应花费太长时间时触发。

Java Bean配置方式

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）；第二个是请求处理的超时时间（ms）。

```
@Bean
public Request.Options options() {
    return new Request.Options(3000, 5000);
}
```

yml配置文件配置方式

```
spring:
    cloud:
        openfeign:
            client:
                config:
                  mall-order:  #对应微服务
                    # 连接超时时间
                    connectTimeout: 3000
                    # 请求处理超时时间
                    readTimeout: 5000
```

补充说明： Feign的底层用的是Ribbon或者LoadBalancer，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"
}
```


契约配置（了解即可）

Spring Cloud 在 Feign 的基础上做了扩展，可以让 Feign 支持 Spring MVC 的注解来调用。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。

Java Bean配置方式

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);
}
```

yml配置文件配置方式

```yaml
spring:
    cloud:
        openfeign:
            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
```

客户端组件配置

Feign 中默认使用 JDK 原生的 URLConnection 发送 HTTP 请求，没有连接池，我们可以集成别的组件来替换掉 URLConnection，比如 Apache HttpClient5，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);
}
```

配置Apache HttpClient5

从Spring Cloud OpenFeign 4开始，不再支持Feign Apache HttpClient 4。我们建议使用Apache HttpClient 5。

1）引入依赖

```xml
<!-- Apache HttpClient5 -->
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-hc5</artifactId>
</dependency>

```

2）修改yml配置，启用Apache HttpClient5 ，可以忽略

```yaml
spring:
  cloud:
    openfeign:
      httpclient:  #feign client使用 Apache HttpClient5
        hc5:
          enabled: true
```

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

```
// for load balanced clients.
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(ApacheHttp5Client.class)
@ConditionalOnMissingBean(org.apache.hc.client5.http.impl.classic.CloseableHttpClient.class)
@ConditionalOnProperty(value = "spring.cloud.openfeign.httpclient.hc5.enabled", havingValue = "true",
        matchIfMissing = true)
@Import(org.springframework.cloud.openfeign.clientconfig.HttpClient5FeignConfiguration.class)
protected static class HttpClient5FeignConfiguration {
```

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

配置 OkHttp

1）引入依赖

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

2）修改yml配置，将 Feign 的 HttpClient 禁用，启用 OkHttp，配置如下：

```yaml
spring:
  cloud:
    openfeign:
      okhttp:       #feign client使用 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

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
spring:
  cloud:
    openfeign:
      compression:  # 配置 GZIP 来压缩数据
        request:
          enabled: true
          mime-types: text/xml,application/xml,application/json
          min-request-size: 1024  # 最小请求压缩阈值
        response:
          enabled: true
```

注意：当 Feign 的 HttpClient不是 okHttp的时候，压缩配置不会生效，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容器中不包含指定的 bean 时条件匹配，也就是没有启用 okhttp3 时才会进行压缩配置。

编码器解码器配置

Feign 中提供了自定义的编码解码器设置，同时也提供了多种编码器的实现，比如 Gson、Jaxb、Jackson。我们可以用不同的编码解码器来处理数据的传输。如果你想传输 XML 格式的数据，可以自定义 XML 编码解码器来实现，或者使用官方提供的 Jaxb。

扩展点：Encoder & Decoder

```
public interface Encoder {
    void encode(Object object, Type bodyType, RequestTemplate template) throws EncodeException;
}
public interface Decoder {
    Object decode(Response response, Type type) throws IOException, DecodeException, FeignException;
}
```

方式1：利用@Configuration实现全局配置，对所有的微服务调用者都生效

以配置jackson为例

1） 引入依赖

使用Jackson，需要引入依赖：

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

使用Gson，需要引入依赖：

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

2）配置编码解码器只需要在 Feign 的配置类中注册 Decoder 和 Encoder 这两个类即可

```
@Bean
public Decoder decoder() {
    return new JacksonDecoder();
}
@Bean
public Encoder encoder() {
    return new JacksonEncoder();
}
```

方式2：局部配置，yml中对调用的微服务提供者进行配置

```
spring:
  cloud:
    openfeign:
      client:
        config:
          mall-order:  #对应微服务
            # 配置编解码器
            encoder: feign.jackson.JacksonEncoder
            decoder: feign.jackson.JacksonDecoder
```

拦截器配置

通过拦截器实现参数传递

通常我们调用的接口都是有权限控制的，很多时候可能认证的值是通过参数去传递的，还有就是通过请求头去传递认证信息，比如 Basic 认证方式。

Feign 中我们可以直接配置 Basic 认证

```

@Bean
```

```
public BasicAuthRequestInterceptor basicAuthRequestInterceptor() {
    return new BasicAuthRequestInterceptor("fox", "123456");
}
```

扩展点： feign.RequestInterceptor

每次 feign 发起http调用之前，会去执行拦截器中的逻辑。

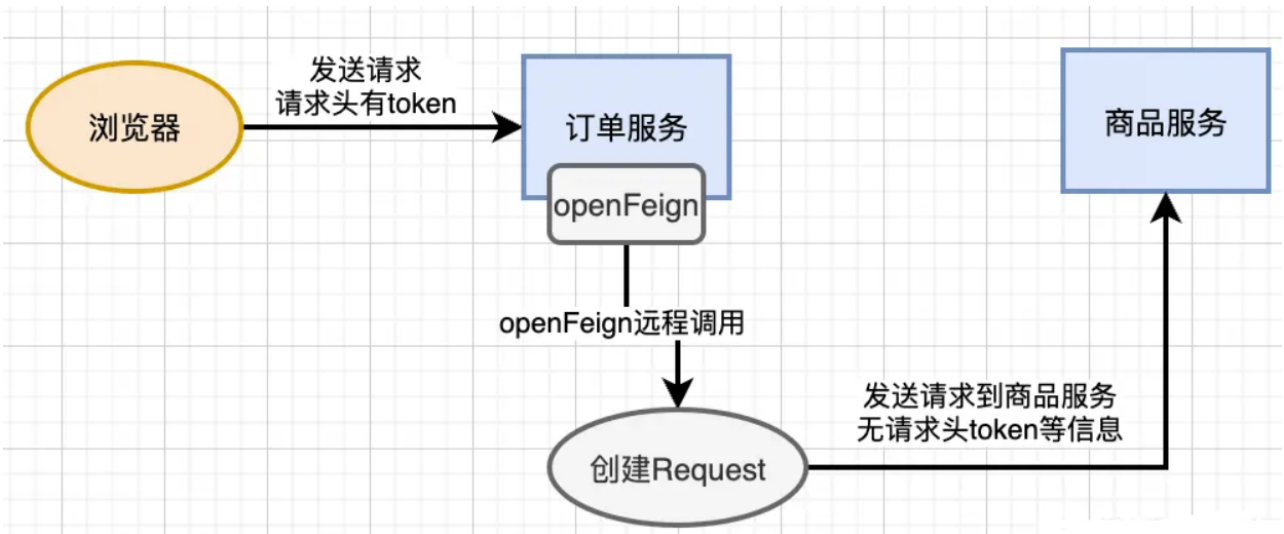
```
public interface RequestInterceptor {

  /**
   * Called for every request. Add data using methods on the supplied {@link RequestTemplate}.
   */
  void apply(RequestTemplate template);
}
```

使用场景

- 统一添加 header 信息；
- 对 body 中的信息做修改或替换；

自定义拦截器实现认证逻辑

OpenFeign作为微服务间接口的调用组件，除了需要考虑传递消息体外，还需要考虑到如何在各个服务间传递请求头信息。如果不做任何配置，直接使用openFeign在服务间进行调用就会如下图：



这样会丢失请求头，在企业级的应用中，token是非常重要的请求信息，他会携带权限、用户信息等。

解决方案：

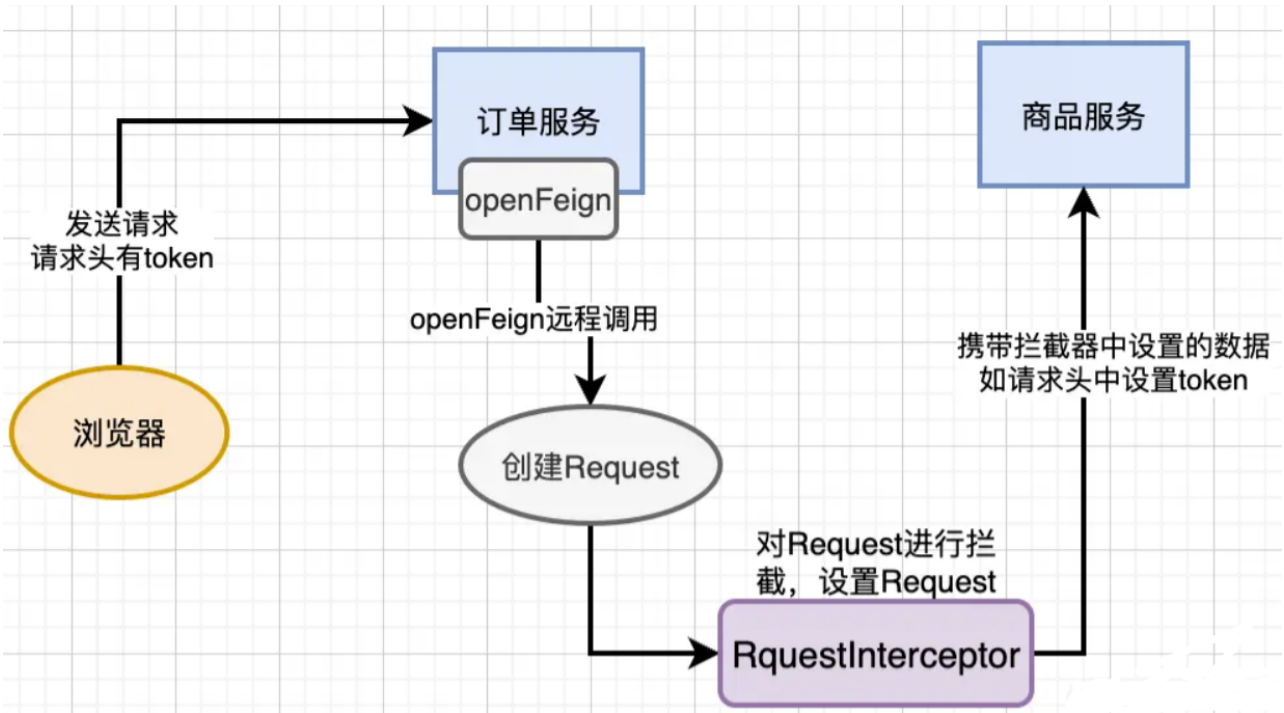
- **方案1：增加接口参数**

```
@RequestMapping(value = "/api/test", method = RequestMethod.GET)
String callApiTest(@RequestParam(value = "name") String name, @RequestHeader(value = "token") String token);
```

毫无疑问，这方案不好，因为对代码有侵入，需要开发人员每次手动的获取和添加接口参数，因此舍弃

- **方案2：添加拦截器**

openFeign在远程调用之前会遍历容器中的RequestInterceptor，调用RequestInterceptor的apply方法，创建一个新的Request进行远程服务调用。因此可以通过实现RequestInterceptor给容器中添加自定义的RequestInterceptor实现类，在这个类里面设置需要发送请求时的参数，比如请求头信息，链路追踪信息等。



代码实现：

```
@Slf4j
public class FeignAuthRequestInterceptor implements RequestInterceptor {
    @Override
    public void apply(RequestTemplate template) {
        // 业务逻辑  模拟认证逻辑
```

```
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder
                .getRequestAttributes();
        if(null != attributes){
            HttpServletRequest request = attributes.getRequest();
            String access_token = request.getHeader("Authorization");
            log.info("从Request中解析请求头:{}",access_token);
            //设置token
            template.header("Authorization",access_token);
        }

    }
}

@Configuration  // 全局生效
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

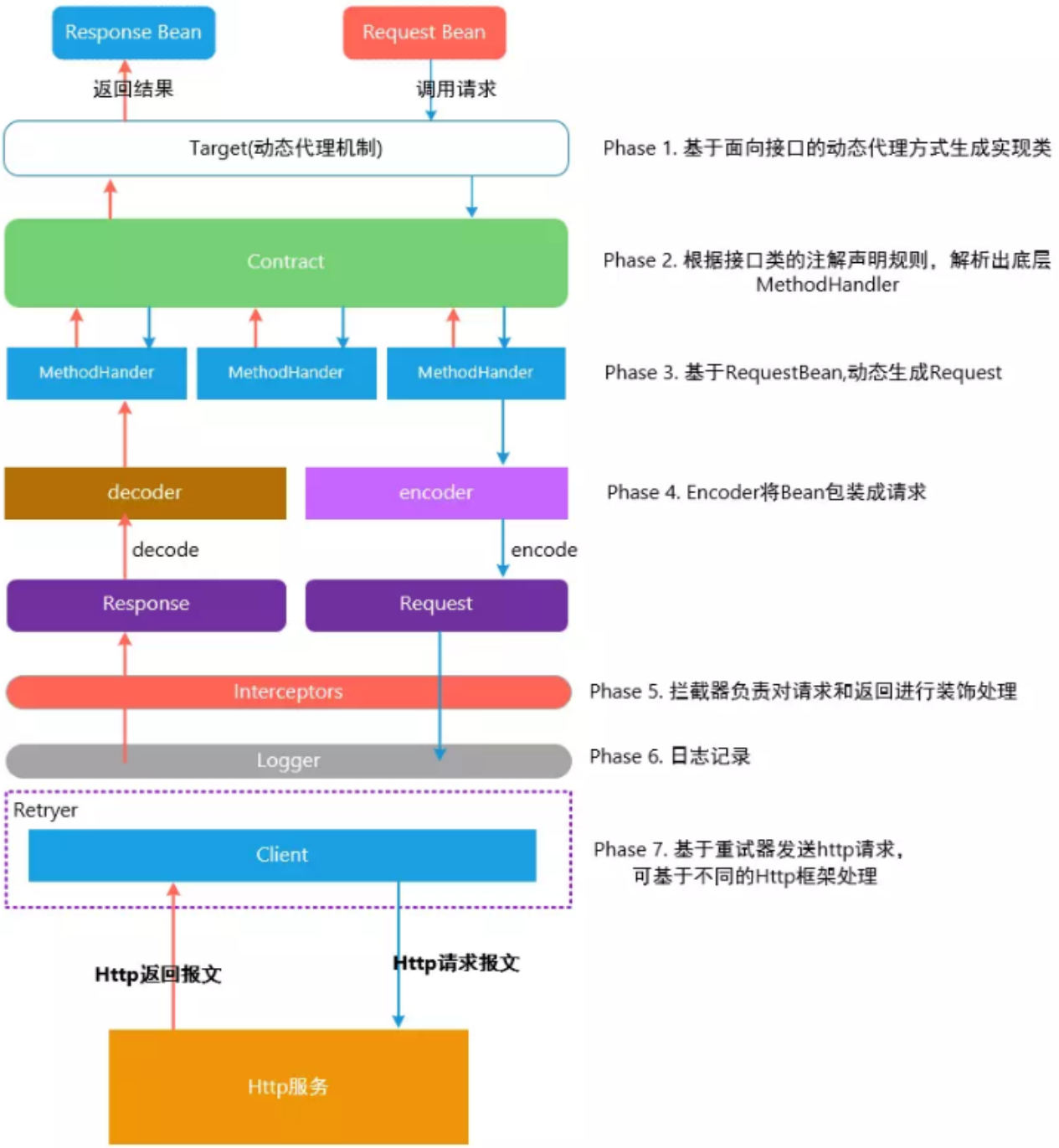
```
ervice#findOrderByUserId] ---> GET http://mall-order/order/findOrderByUserId/1 HTTP/1.1
ervice#findOrderByUserId] Accept-Encoding: gzip
ervice#findOrderByUserId] Accept-Encoding: deflate
ervice#findOrderByUserId] Authorization: bearer 541ccdeb-d5b8-41e5-8768-daf1b25bd19d
ervice#findOrderByUserId] ---> END HTTP (0-byte body)
ervice#findOrderByUserId] <--- HTTP/1.1 200  (4ms)
ervice#findOrderByUserId] connection: keep-alive
ervice#findOrderByUserId] content-type: application/json
ervice#findOrderByUserId] date: Thu, 17 Aug 2023 05:06:33 GMT
```

也可以在yml中配置

```yaml
spring:
    cloud:
        openfeign:
            client:
                config:
                  mall-order:  #对应微服务
                    requestInterceptors:  #配置拦截器
                      -
com.tuling.mall.feigndemo.interceptor.FeignAuthRequestInterceptor
```

mall-order端可以通过 @RequestHeader获取请求参数进行校验，建议在filter或者mvc interceptor中进行处理

2.5 OpenFeign设计架构



feign的源码链接： https://www.processon.com/view/link/5e80ae79e4b03b99653fe42f