Feign核心流程源码分析

Mr_1214 (/u/983b1f8da680) + 关注
2018.05.19 19:49 字数 408 阅读 1214 评论 0 喜欢 0
(/u/983b1f8da680)

Feign是简化Java HTTP客户端开发的工具（java-to-httpclient-binder），它的灵感来自于Retrofit、JAXR WebSocket。Feign的初衷是降低统一绑定Denominator到HTTP API的复杂度。

下来我们通过简单用例来分析工作核心原理以及流程

```
interface GitHub {
  @RequestLine("GET /repos/{owner}/{repo}/contributors")
  List<Contributor> contributors(@Param("owner") String owner, @Param("repo") String repo);
}

static class Contributor {
  String login;
  int contributions;
}

public static void main(String... args) {
  GitHub github = Feign.builder()
                       .decoder(new GsonDecoder())
                       .target(GitHub.class, "https://api.github.com");

  // 获取贡献者列表，并打印其登录名以及贡献次数
  List<Contributor> contributors = github.contributors("netflix", "feign");
  for (Contributor contributor : contributors) {
    System.out.println(contributor.login + " (" + contributor.contributions + ")");
  }
}
```

- ==通过上面样例我们看到首先定义了一个 GitHub接口，然后通过Feign.builder().target()创建了一个GitHub接口的实例。
  那么Feign.builder().target()这里应该就是Feign的核心部分了，接下来我们跟踪源码，查看Feign都为我们做了什么事情==

1. Feign.builder()创建Builder实例对象

```
   //1.创建Builder实例对象
   public static Feign.Builder builder() {
        //调用创建Builder构建函数
        return new Feign.Builder();
    }


  //2.Builder构建函数，初始化组件信息，当然我们也可以自定义组件
    public Builder() {
           //日志级别
            this.logLevel = Level.NONE;
            //注解解析组件
            this.contract = new Default();
            //http发送组件
            this.client = new feign.Client.Default((SSLSocketFactory)null, (HostnameVerifier)null);
            //重试机制组件
            this.retryer = new feign.Retryer.Default();
            //日志
            this.logger = new NoOpLogger();
            //编码解码器组件
            this.encoder = new feign.codec.Encoder.Default();
            this.decoder = new feign.codec.Decoder.Default();
            this.errorDecoder = new feign.codec.ErrorDecoder.Default();
            this.options = new Options();
            //默认的反射InvocationHandlerFactory
            // Feign 使用的jdk自带的动态代理
            this.invocationHandlerFactory = new feign.InvocationHandlerFactory.Default();
        }
```

2. Builder.target()创建目标实例对象

```
    //1. 调用target传入目标接口以及请求URL
    public <T> T target(Class<T> apiType, String url) {

         return this.target(new HardCodedTarget(apiType, url));
    }

    //2.根据Target包装对象创建目标接口的实例对象
    public <T> T target(Target<T> target) {
            //通过3构建一个Feign对象，然后通过newInstance方法获取目标实例对象
            return this.build().newInstance(target);
    }

    // 3. 构建Feign对象
    public Feign build() {
            //创建方法代理类工厂
            Factory synchronousMethodHandlerFactory = new Factory(this.client, this.retryer, this.requestInterceptors, this.logger,
            //
            ParseHandlersByName handlersByName = new ParseHandlersByName(this.contract, this.options, this.encoder, this.decoder, t
            //这里返回真实的Feign对象
            return new ReflectiveFeign(handlersByName, this.invocationHandlerFactory);
    }
```

3. Feign.newInstance()(通过2我们可以看出实际是通过ReflectiveFeign对象的newInstance方法创建)

```
//1. 这里是创建目录接口实例对象的真正地方
// 这里可以看到是使用了jdk自带的动态代理实现
//可以很清楚的看到返回的是目标接口的代理对象
 public <T> T newInstance(Target<T> target) {
        Map<String, MethodHandler> nameToHandler = this.targetToHandlersByName.apply(target);
        Map<Method, MethodHandler> methodToHandler = new LinkedHashMap();
        List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList();

        InvocationHandler handler = this.factory.create(target, methodToHandler);
        //这里可以看到是使用了jdk自带的动态代理实现的
        //那么我们知道jdk动态代理真正执行的是InvocationHandler接口中的invoke方法，我们再跟踪invoke，看下执行目标接口方法时具体逻辑。
        T proxy = Proxy.newProxyInstance(target.type().getClassLoader(), new Class[]{target.type()}, handler);
        return proxy;
    }

    //2. 执行目标接口方法带来具体实现（FeignInvocationHandler）
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
     //在此我们可以看出目标函数除了equals，hashCode，toString方法外都会调用this.dispatch.get(method)).invoke(args)
     //dispatch是目标函数代理类集合，目标接口中每个函数都会对应有一个MethodHandler类，至于怎么得到的有兴趣可以查看源码
            if(!"equals".equals(method.getName())) {
                return "hashCode".equals(method.getName())?Integer.valueOf(this.hashCode()):("toString".equals(method.getName())?th
            } else {

            }
        }
```

4. 通过3我们看出了目录接口每个函数的执行其实是执行其MethodHandler类的invoke方法那么下来我们看下这里具体逻辑

   MethodHandler默认实现SynchronousMethodHandler

```
//1. 接口方法执行都会调用其对应的invoke方法
 public Object invoke(Object[] argv) throws Throwable {
        RequestTemplate template = this.buildTemplateFromArgs.create(argv);
        //重试组件
        Retryer retryer = this.retryer.clone();

        while(true) {
            try {
                //执行请求并解码
                return this.executeAndDecode(template);
            } catch (RetryableException var5) {
                retryer.continueOrPropagate(var5);
                if(this.logLevel != Level.NONE) {
                    this.logger.logRetry(this.metadata.configKey(), this.logLevel);
                }
            }
        }
    }

    //2 构建request请求并执行和解码
    Object executeAndDecode(RequestTemplate template) throws Throwable {
        //1. 获取request请求
        Request request = this.targetRequest(template);

        long start = System.nanoTime();
        Response response;
        try {
           //通过http组件发送请求
            response = this.client.execute(request, this.options);
            response.toBuilder().request(request).build();
        } catch (IOException var15) {

        }
        //解码操作调用解码组件进行解码
        //这里省略
        return var9;
    }

    //3组装request请求，这里同时完成了拦截器调用的逻辑
     Request targetRequest(RequestTemplate template) {
        //获取当前请求的所有拦截器
        Iterator var2 = this.requestInterceptors.iterator();

        while(var2.hasNext()) {
            RequestInterceptor interceptor = (RequestInterceptor)var2.next();
            //依次调用拦截器进行拦截操作
            interceptor.apply(template);
        }

        //返回Request对象
        return this.target.apply(new RequestTemplate(template));
    }
```

通过上面四步我们可以清晰的看出我们通过定义目标接口是怎么一步一步的完成了http请求发送与接收。
在项目中我们使用Feign是简化我们的http操作，同时我们理解整个http请求响应是怎么通过Feign来完成的。这样后期不管是定制还是问题定位，我们的分析。

后记

本文记录了使用Feign后想了解Feign实现原理然后跟踪源码的一些流程和心得。