【推荐】 UniHttp 开源第三方接口对接框架

个人开源框架矩阵

百万级任务重试框架 Fast-Retry

stream流太难用了看看JDFrame

spring-smart-di 动态切换实现类框架

UniHttp 第三方接口对接框架

前言

从企业级项目来说，如果你项目里还在用传统的编程式Http客户端比如HttpClient、Okhttp去直接对接第三方Http接口，
那么你项目一定充斥着大量的对接逻辑和代码， 并且针对不同的对接渠道方需要每次封装一次调用的简化，
一旦封装不好系统将会变得难以维护，难以阅读， 甚至不同的开发同学会用自己的方式用不同的Http客户端用不同的封装逻辑去对接接口，
这种情况一般发生于项目换了维护者，技术负责人也没把控代码质量和规范所导致

如果你的项目里也存在这样的问题或者需要解决这样的问题, 那么`UniHttp`就是你的版本答案。

1、简介

一个`声明式的Http接口对接框架`，能以极快的方式完成对一个第三方Http接口的对接和使用，之后就像调用本地方法一样自动去发起Http请求，不需要开发者去如何发送一个请求，如何去传递Http请求参数，以及如何对请求结果进行处理和反序列化，这些框架都帮你一一实现
就像配置 `Spring的Controller` 那样简单，只不过相当于是反向配置而已

该框架`更注重于`如何保持高内聚和可读性高的代码情况下`与快速第三方渠道接口进行对接和集成`，而非像传统编程式的Http请求客户端（比如

HttpClient、Okhttp）那样`专注于如何去发送Http请求`，虽然底层也是用的Okhttp去发送请求。 与其说的是对接的Http接口，不如说是对接的第三方渠道，UniHttp可支持自定义接口渠道方HttpAPI注解以及一些自定义的对接和交互行为 ，为此扩展了发送和响应和反序列化一个`Http请求的各种生命周期钩子`，开发者可自行去扩展实现。

2、快速开始

2.1、引入依赖

```
    <dependency>
      <groupId>io.github.burukeyou</groupId>
      <artifactId>uniapi-http</artifactId>
      <version>0.0.4</version>
    </dependency>
```

2.2、对接接口

首先随便创建一个接口，然后在接口上标记@HttpApi注解，然后指定请求的域名url, 然后就可以在方法上去配置对接哪个接口。

比如下面两个方法的配置则对接了以下两个接口

GET [http://localhost:8080/getUser和](http://cxyroad.com/”http://localhost:8080/getUser%E5%92%8C”)

POST [http://localhost:8080/addUser](http://cxyroad.com/”http://localhost:8080/addUser”)

`方法返回值定义成Http响应body对应的类型即可，默认会使用fastjson反序列化Http响应body的值为该类型对象。`

```
@HttpApi(url = ”http://localhost:8080”)
interface UserHttpApi {

   @GetHttpInterface(”/getUser”)
   BaseRsp<String> getUser(@QueryPar(”name”) String
param,@HeaderPar(”userId”) Integer id);

   @PostHttpInterface(”/addUser”)
   BaseRsp<Add4DTO> addUser(@BodyJsonPar Add4DTO req);

}
```

@QueryPar 表示将参数值放到Http请求的查询参数内

@HeaderPar 表示将参数值放到Http请求的请求头里

@BodyJsonPar 表示将参数值放到Http请求body内，并且content-type是application/json

1、getUser方法最终构建的Http请求报文为

```
GET http://localhost:8080/getUser?name=param
Header:
    userId: id
```

2、addUser最终构建的Http请求报文为

```
        POST:  http://localhost:8080/addUser
        Header:
            Content-Type:   application/json
        Body:
            {”id”:1,”name”:”jay”}
```

```

### 2.3、声明定义的HttpAPI的包扫描路径

在spring的配置类上使用`@UniAPIScan注解`标记定义的@HttpAPI的包扫描路径,会自动为标记了@HttpApi接口生成代理对象并且注入到Spring容器中，
之后只需要像使用Spring的其他bean一样，依赖注入使用即可

```
@UniAPIScan(”com.xxx.demo.api”)
@SpringBootApplication
public class DemoApplication {

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

### 2.4 依赖注入使用即可

```
@Service
class UserAppService {

    @Autowired
    private UserHttpApi userHttpApi;

    public void doSomething(){
        userHttpApi.getUser(”jay”,3);
    }
}
```

3、说明介绍

3.1、@HttpApi注解
```

用于标记接口上，该接口上的方法会被代理到对应的Http请求接口，可指定请求的域名，也可指定自定义的Http代理逻辑等等。

3.2、@HttpInterface注解

用于配置一个接口的参数，包括请求方式、请求路径、请求头、请求cookie、请求查询参数等等

并且内置了以下请求方式的@HttpInterface，不必再每次手动指定请求方式

* @PostHttpInterface
* @PutHttpInterface
* @DeleteHttpInterface
* @GetHttpInterface

```
    @PostHttpInterface(
            // 请求路径
            path = "/getUser",
            // 请求头
            headers = {"clientType:sys-app","userId:99"},
            // url查询参数
            params = {"name=周杰伦","age=1"},
            // url查询参数拼接字符串
            paramStr = "a=1&b=2&c=3&d=哈哈
&e=%E7%89%9B%E9%80%BC",
            // cookie 字符串
            cookie = "name=1;sessionId=999"
    )
    BaseRsp<String> getUser();
```

3.3、@Par注解

以下各种Par后缀的注解，主要用于方法参数上，用于指定在发送请求时将参数

值放到Http请求体的哪部分上。

为了方便描述，下文描述的普通值就是表示String，基本类型、基本类型的包装类型等类型.

简单复习下Http协议报文

@QueryPar注解

标记Http请求url的查询参数

支持以下方法参数类型的标记: 普通值、普通值集合、对象、Map

```
    @PostHttpInterface
    BaseRsp<String> getUser(@QueryPar(”id”)  String id,  //  普通值
                            @QueryPar(”ids”) List<Integer> idsList, //  普通值集合
                            @QueryPar User user,  // 对象
                            @QueryPar Map<String,Object> map); // Map
```

如果类型是普通值或者普通值集合需要手动指定参数名，因为是当成单个查询参数传递

如果类型是对象或者Map是当成多个查询参数传递，字段名或者map的key名就是参数名，字段值或者map的value值就是参数值。

* `如果是对象，参数名默认是字段名，由于用的是fastjson序列化可以用@JSONFieId指定别名`

@PathPar注解

标记Http请求路径变量参数，仅支持标记普通值类型

```
    @PostHttpInterface("/getUser/{userId}/detail")
    BaseRsp<String> getUser(@PathPar("userId") String id); // 普通值
```

@HeaderPar注解

标记Http请求头参数

支持以下方法参数类型： 对象、Map、普通值

```
    @PostHttpInterface
    BaseRsp<String> getUser(@HeaderPar("id") String id, // 普通值
                            @HeaderPar User user, // 对象
                            @HeaderPar Map<String,Object> map); // Map
```

如果类型是普通值类型需要手动指定参数名，当成单个请求头参数传递. 如果是对象或者Map当成多个请求头参数。

@CookiePar注解

用于标记Http请求的cookie请求头

支持以下方法参数类型: Map、Cookie对象、字符串

```
    @PostHttpInterface
    BaseRsp<String> getUser(@CookiePar(”id”）String cookiePar，//  普通值 （指定name）当成单个cookie键值对处理
                            @CookiePar String cookieString，// 普通值 （不指定name），当成完整的cookie字符串处理
                            @CookiePar com.burukeyou.uniapi.http.support.Cookie cookieObj，// 单个Cookie对象
                            @CookiePar List<com.burukeyou.uniapi.http.support.Cookie> cookieList // Cookie对象列表
                            @CookiePar Map<String,Object> map); // Map
```

如果类型是字符串时，`当指定参数名时`，当成单个cookie键值对处理，如果`不指定参数名时`当成完整的cookie字符串处理比如a=1;b=2;c=3 这样

如果是Map当成多个cookie键值对处理。

如果类型是内置的 `com.burukeyou.uniapi.http.support.Cookie`对象当成单个cookie键值对处理

@BodyJsonPar注解

用于标记Http请求体内容为json形式: 对应content-type为 `application/json`

支持以下方法参数类型: 对象、对象集合、Map、普通值、普通值集合

```
    @PostHttpInterface
    BaseRsp<String> getUser(@BodyJsonPar  String id,                // 普通值
                            @BodyJsonPar  String[] id               // 普通值集合
                            @BodyJsonPar List<User> userList,       // 对象集合
                            @BodyJsonPar User user,                 // 对象
                            @BodyJsonPar Map<String,Object> map);   // Map
```

```

序列化和反序列化默认用的是fastjson，所以如果想指定别名，可以在字段上标记 @JSONField 注解取别名

### @BodyFormPar注解

用于标记Http请求体内容为普通表单形式: 对应content-type为 `application/x-www-form-urlencoded`

支持以下方法参数类型： 对象、Map、普通值

```
    @PostHttpInterface
    BaseRsp<String> getUser(@BodyFormPar(”name”) String value,
// 普通值
                            @BodyFormPar User user,   // 对象
                            @BodyFormPar Map<String,Object> map);   // Map
```

如果类型是普通值类型需要手动指定参数名，当成单个请求表单键值对传递

### BodyMultiPartPar注解

用于标记Http请求体内容为复杂形式: 对应content-type为 `multipart/form-data`

支持以下方法参数类型: 对象、Map、普通值、File对象

```
    @PostHttpInterface
    BaseRsp<String> getUser(@BodyMultiPartPar(”name”) String value,
    // 单个表单文本值
                            @BodyMultiPartPar User user,   // 对象
                            @BodyMultiPartPar Map<String,Object> map,   //
Map
```
```

@BodyMultiPartPar(”userImg”) File file); // 单个表单文件值

```

如果参数类型是普通值或者File类型，当成单个表单键值对处理，需要手动指定参数名。

如果参数类型是对象或者Map，当成多个表单键值对处理。 如果字段值或者map的value参数值是File类型，则自动当成是文件表单字段传递处理

### @BodyBinaryPar注解

用于标记Http请求体内容为二进制形式: 对应content-type为 `application/octet-stream`

支持以下方法参数类型: InputStream、File、InputStreamSource

```
    @PostHttpInterface
    BaseRsp<String> getUser(@BodyBinaryPar InputStream value,
                            @BodyBinaryPar File user,
                            @BodyBinaryPar InputStreamSource map);
```

### @ComposePar注解

这个注解本身不是对Http请求内容的配置，仅用于标记一个对象，然后会对该对象内的所有标记了其他@Par注解的字段进行嵌套解析处理，
目的是减少方法参数数量，支持都内聚到一起传递

支持以下方法参数类型: 对象

```
    @PostHttpInterface
    BaseRsp<String> getUser(@ComposePar UserReq req);
```
```

```

比如UserReq里面的字段可以嵌套标记其他@Par注解，具体支持的标记类型和处理逻辑与前面一致

```
class UserReq {

    @QueryPar
    private Long id;

    @HeaderPar
    private String name;

    @BodyJsonPar
    private Add4DTO req;

    @CookiePar
    private String cook;
}
```

### 3.4、原始的HttpResponse

HttpResponse表示Http请求的原始响应对象，如果业务需要拿到完整的Http响应，只需要在方法返回值包装返回即可。
如下面所示，此时`HttpResponse<Add4DTO>`里的泛型Add4DTO才是代表接口实际返回的响应内容，后续可直接手动获取

```
    @PostHttpInterface("/user-web/get")
    HttpResponse<Add4DTO> get();
```

通过它我们就可以拿到响应的Http状态码、响应头、响应cookie等等，当然也可以拿到我们的响应body的内容通过getBodyResult方法

3.5、处理文件下载接口
```

对于若是下载文件的类型的接口，可将方法返回值定义为 HttpBinaryResponse、HttpFileResponse、HttpInputStreamResponse 的任意一种，
这样就可以拿到下载后的文件。

* `HttpBinaryResponse`: 表示下载的文件内容以二进制形式返回，如果是大文件请谨慎处理，因为会存放在内存中
* `HttpFileResponse`: 表示下载的文件内容以File对象返回，这时文件已经被下载到了本地磁盘
* `HttpInputStreamResponse`: 表示下载的文件内容输入流的形式返回，这时文件其实还没被下载到客户端，调用者可以自行读取该输入流进行文件的下载
*

3.6、HttpApiProcessor 生命周期钩子

HttpApiProcessor是一个Http请求接口的各种生命周期钩子，开发者可以实现它在里面自定义编写各种对接逻辑。 然后可以配置到@HttpApi注解或者@HttpInterface注解上, 然后框架内部默认会从SpringContext获取，获取不到则手动new一个。

* 通常一个Http请求需要经历 构建请求参数、发送Http请求时，Http响应后获取响应内容、反序列化Http响应内容成具体对象。

`目前提供了4种钩子,执行顺序流程如下:`

```
                postBeforeHttpMetadata                (请求发送前)在发送请求之前，对Http请求体后置处理
                        |
                        V
                postSendingHttpRequest                (请求发送时)在Http请求发送时处理
                        |
                        V
            postAfterHttpResponseBodyString           (请求响应后)对响应body文本字符串进行后置处理
                        |
```

```
                    V
          postAfterHttpResponseBodyResult          (请求响应后)对响应body反序列化后的结果进行后置处理
                    |
                    V
          postAfterMethodReturnValue               (请求响应后)对代理的方法的返回值进行后置处理，类似aop的后置处理
```

.

1、`postBeforeHttpMetadata`: 可在发送http请求之前对请求体进行二次处理，比如加签之类

2、`postSendHttpRequest`: Http请求发送时会回调该方法，可以在该方法执行自定义的发送逻辑或者打印发送日志

3、`postAfterHttpResponseBodyString`： Http请求响应后，对响应body字符串进行进行后置处理，比如如果是加密数据可以进行解密

4、`postAfterHttpResponseBodyResult`： Http请求响应后，对响应body反序列化后的对象进行后置处理，比如填充默认返回值

5、`postAfterMethodReturnValue`： Http请求响应后，对代理的方法的返回值进行后置处理，类似aop的后置处理

.

回调参数说明:

* `HttpMetadata`: 表示此次Http请求的请求体，包含请求url，请求头、请求方式、请求cookie、请求体、请求参数等等。
* `HttpApiMethodInvocation`: 继承自MethodInvocation， 表示被代理的方法调用上下文，可以拿到被代理的类，被代理的方法，被代理的HttpAPI注解、HttpInterface注解等信息

3.7、配置自定义的Http客户端

默认使用的是Okhttp客户端，如果要重新配置Okhttp客户端,注入spring的bean即可,如下

```
@Configuration
public class CusotmConfiguration {

    @Bean
    public OkHttpClient myOHttpClient(){
        return new OkHttpClient.Builder()
                .readTimeout(50, TimeUnit.SECONDS)
                .writeTimeout(50, TimeUnit.SECONDS)
                .connectTimeout(10, TimeUnit.SECONDS)
                .connectionPool(new ConnectionPool(20,10,
TimeUnit.MINUTES))
                .build();
    }
}
```

4、企业级渠道对接实战

案例背景:

* 假设现在需要对接一个某天气服务的所有接口，需要在请求cookie带上一个token字段和sessionId字段， 这两个字段的值需要每次接口调用前先手动调渠道方的一个特定的接口申请获取，token值在该接口返回值中返回，sessionId在该接口的响应头中返回。然后还需要在请求头上带上一个sign签名字段， 该sign签名字段生成规则需要用渠道方提供的公钥对所有请求体和请求参数进行加签生成。
然后还需要在每个接口的查询参数上都带上一个渠道方分配的客户端appId。

4.1 在application.yml中配置对接渠道方的信息

```
```

```
channel:
  mtuan:
    # 请求域名
    url: http://127.0.0.1:8999
    # 分配的渠道appId
    appId: UUU-asd-01
    # 分配的公钥
    publicKey: fajdkf9492304jklfahqq
```

4.2 自定义该渠道方的HttpAPI注解

假设现在对接的是某团，所以自定义注解叫`@MTuanHttpApi`吧，然后需要在该注解上标记@HttpApi注解，并且需要配置processor字段，需要去自定义实现一个HttpApiProcessor这个具体实现后续讲。

有了这个注解后就可以自定义该注解与对接渠道方相关的各种字段配置，当然也可以不定义。 注意这里url的字段是使用 @AliasFor(annotation = HttpApi.class),
这样构建的HttpMetadata中会默认解析填充要请求体，不标记则也可自行处理。

```
@Inherited
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@HttpApi(processor = MTuanHttpApiProcessor.class)
public @interface MTuanHttpApi {

    /**
     * 渠道方域名地址
     */
    @AliasFor(annotation = HttpApi.class)
    String url() default "${channel.mtuan.url}";

    /**
     * 渠道方分配的appId
     */
    String appId() default "${channel.mtuan.appId}";
}
```

```

```
@Slf4j
@Component
public class MTuanHttpApiProcessor implements
HttpApiProcessor<MTuanHttpApi> {

}
```

注意实现的HttpApiProcessor泛型要指定为刚才定义的注解@MTuanHttpApi类型，因为这个HttpApiProcessor配置到它上面，如果需要通用处理可以定义为Annocation类型

## 4.3 对接接口

有了@MTuanHttpApi注解之后就可以开始对接接口了，比如假设有两个接口要对接。一个就是前面说的获取令牌的接口。 一个是获取天气情况的接口。

* 为什么getToken方法返回值是 `HttpResponse`, 这是UniHttp内置的原始Http响应对象，方便我们去拿到原始Http响应体的一些内容（比如响应状态码、响应cookie）。
其中的泛型BaseRsp才是实际的Http响应体反序列化后的内容。 而getCityWeather方法没有使用HttpResponse包装,
BaseRsp只是单纯Http响应体反序列化后的内容，这是两者的区别。 前面介绍过 `HttpResponse`，其实大部份接口是不HttpResponse的可以不用去配置。

```
@MTuanHttpApi
public interface WeatherApi {

    /**
     * 根据城市名获取天气情况
     */
    @GetHttpInterface("/getCityByName")
    BaseRsp<WeatherDTO> getCityWeather(@QueryPar("city") String
cityName);

    /**
```
```

```
     * 根据appId和公钥获取令牌
     */
    @PostHttpInterface("/getToken")
    HttpResponse<BaseRsp<TokenDTO>> getToken(@HeaderPar("appId")
String appId, @HeaderPar("publicKey")String publicKey);

}
```

4.4、自定义HttpApiProcessor

在之前我们自定义了一个@MTuanHttpApi注解上指定了一个MTuanHttpApiProcessor，接下来我们去实现他的具体内容为了实现我们案例背景里描述的功能。

```
@Slf4j
@Component
public class MTuanHttpApiProcessor implements
HttpApiProcessor<MTuanHttpApi> {

    /**
     * 渠道方分配的公钥
     */
    @Value("${channel.mtuan.publicKey}")
    private String publicKey;

    @Value("${channel.mtuan.appId}")
    private String appId;

    @Autowired
    private Environment environment;

    @Autowired
    private WeatherApi weatherApi;

    /** 实现-postBeforeHttpMetadata： 发送Http请求之前会回调该方法
，可对Http请求体的内容进行二次处理
     *
     * @param httpMetadata              原来的请求体
     * @param methodInvocation          被代理的方法
     * @return                          新的请求体
```

```
     */
    @Override
    public HttpMetadata postBeforeHttpMetadata(HttpMetadata
httpMetadata, HttpApiMethodInvocation<MTuanHttpApi>
methodInvocation) {
        /**
         * 在查询参数中添加提供的appId字段
         */
        // 获取MTuanHttpApi注解
        MTuanHttpApi apiAnnotation =
methodInvocation.getProxyApiAnnotation();

        // 获取MTuanHttpApi注解的appId，由于该appId是环境变量所以我们
从environment中解析取出来
        String appIdVar = apiAnnotation.appId();
        appIdVar = environment.resolvePlaceholders(appIdVar);

        // 添加到查询参数中
        httpMetadata.putQueryParam("appId",appIdVar);

        /**
         *  生成签名sign字段
         */
        // 获取所有查询参数
        Map<String, Object> queryParam =
httpMetadata.getHttpUrl().getQueryParam();

        // 获取请求体参数
        HttpBody body = httpMetadata.getBody();

        // 生成签名
        String signKey = createSignKey(queryParam,body);

        // 将签名添加到请求头中
        httpMetadata.putHeader("sign",signKey);

        return httpMetadata;
    }

    private String createSignKey(Map<String, Object> queryParam,
HttpBody body) {
        // todo 伪代码
        // 1、将查询参数拼接成字符串
        String queryParamString = queryParam.entrySet()
                .stream().map(e -> e.getKey() + "="+e.getValue())
                .collect(Collectors.joining(";"));

        // 2、将请求体参数拼接成字符串
```

```
        String bodyString = "";
        if (body instanceof HttpBodyJSON){
            // application/json  类型的请求体
            bodyString = body.toStringBody();
        }else if (body instanceof HttpBodyFormData){
            // application/x-www-form-urlencoded 类型的请求体
            bodyString = body.toStringBody();
        }else if (body instanceof HttpBodyMultipart){
            // multipart/form-data 类型的请求体
            bodyString =  body.toStringBody();
        }

        // 使用公钥publicKey 加密拼接起来
        String sign = publicKey + queryParamString + bodyString;
        try {
            MessageDigest md = MessageDigest.getInstance("SHA-256");
            byte[] digest = md.digest(sign.getBytes());
            return new String(digest);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 实现-postBeforeHttpMetadata： 发送Http请求时，可定义发送请求的行为 或者打印请求和响应日志。
     */
    @Override
    public HttpResponse<?> postSendHttpRequest(HttpSender httpSender, HttpMetadata httpMetadata) {
        //  忽略 weatherApi.getToken的方法回调，否则该方法也会回调此方法会递归死循环。 或者该接口指定自定义的HttpApiProcessor重写 postSendingHttpRequest
        Method getTokenMethod = ReflectionUtils.findMethod(WeatherServiceApi.class, "getToken",String.class,String.class);
        if (getTokenMethod == null || getTokenMethod.equals(methodInvocation.getMethod())){
            return httpSender.sendHttpRequest(httpMetadata);
        }

        // 1、动态获取token和sessionId
        HttpResponse<String> httpResponse = weatherApi.getToken(appId, publicKey);

        // 从响应体获取令牌token
        String token = httpResponse.getBodyResult();
        // 从响应头中获取sessionId
```

```
        String sessionId = httpResponse.getHeader("sessionId");

        // 把这两个值放到此次的请求cookie中
        httpMetadata.addCookie(new Cookie("token",token));
        httpMetadata.addCookie(new Cookie("sessionId",sessionId));

        log.info("开始发送Http请求 请求接口:{} 请求体
:{}",httpMetadata.getHttpUrl().toUrl(),httpMetadata.toHttpProtocol());

        // 使用框架内置工具实现发送请求
        HttpResponse<?> rsp =
httpSender.sendHttpRequest(httpMetadata);

        log.info("开始发送Http请求 响应结果:{}",rsp.toHttpProtocol());

        return rsp;
    }

    /**
     * 实现-postAfterHttpResponseBodyResult： 反序列化后Http响应体的
内容后回调，可对该结果进行二次处理返回
     * @param bodyResult                Http响应体反序列化后的结果
     * @param rsp                       原始Http响应对象
     * @param method                    被代理的方法
     * @param httpMetadata              Http请求体
     */
    @Override
    public Object postAfterHttpResponseBodyResult(Object bodyResult,
HttpResponse<?> rsp, Method method, HttpMetadata httpMetadata) {
        if (bodyResult instanceof BaseRsp){
            BaseRsp baseRsp = (BaseRsp) bodyResult;
            // 设置
            baseRsp.setCode(999);
        }

        return bodyResult;
    }
}


```

上面我们分别重写了postBeforeHttpMetadata、postSendHttpRequest、postAfterHttpResponseBodyResult三个生命周期的钩子方法去完成我们的需求，在发送请求前对请求体进行加签、在发送请求时动态获取令牌重新构建请求体和打印日志、在发送请求后给响应对象设置code为999。

最后
==

[gitHub代码地址](http://cxyroad.com/
”https://github.com/burukeYou/UniAPI”) 具体使用案例见uniapi-test-http模块

如果觉得项目有用，可以star下感谢!

原文链接: https://juejin.cn/post/7389925676519948297