链滴

年轻人不讲武德，竟然重构出这么优雅后台 API 接口

作者：9526xu
原文链接：https://ld246.com/article/1606870555509
来源网站：链滴

Hello，早上好，我是楼下小黑哥~

最近偶然间在看到 Spring 官方文档的时候，新学到一个注解 @ControllerAdvice，并且成功使用这注解重构我们项目的对外 API 接口，去除繁琐的重复代码，使其开发更加优雅。

展示具体重构代码之前，我们先来看下原先对外 API 接口是如何开发的。

这个 API 接口主要是用来与我们 APP 交互，这个过程我们统一定义一个交互协议，APP 端与后台 API 接口统一都使用 JSON 格式。

另外后台 API 接口对 APP 返回时，统一一些错误码，APP 端需要根据相应错误码，在页面弹出一些示。

下面展示一个查询用户信息返回的接口数据：

```
{
    "code": "000000",
    "msg": "success",
    "result": {
        "id": "1",
        "name": "test"
    }
}
```

code代表对外的错误码，msg代表错误信息，result代表具体返回信息。

前端 APP 获取这个返回信息，首先判断接口返回 code是否为 **000000**，如果是代表查询成功，然后取 result 信息作出相应的展示。否则，直接弹出相应的错误信息。

重构之前

下面我们来看下，重构之前的，后台 API 层的如何编码。

```
/**
 * V1 版本
 *
 * @return
 */
@RequestMapping("testv1")
public APIResult testv1() {
    try {
        User user = new User();
        user.setId("1");
        user.setName("test");
        return APIResult.success(user);
    } catch (APPException e) {
        log.error("内部异常", e);
        return APIResult.error(e.getCode(), e.getMsg());
    } catch (Exception e) {
        log.error("系统异常", e);
        return APIResult.error(RetCodeEnum.FAILED);
```

```
    }
}
```

上面的代码其实很简单，内部统一封装了一个工具类 APIResult，然后用其包装具体的结果。

```
@Data
public class APIResult<T> implements Serializable {

    private static final long serialVersionUID = 47477745421077118 45L;

    private String code;

    private String msg;

    private T result;


    public static <T> APIResult success(T result) {
        APIResult apiResult = new APIResult();
        apiResult.setResult(result);
        apiResult.setCode("000000");
        apiResult.setMsg("success");
        return apiResult;
    }

    public static APIResult error(String code, String msg) {
        APIResult apiResult = new APIResult();
        apiResult.setCode(code);
        apiResult.setMsg(msg);
        return apiResult;
    }

    public static APIResult error(RetCodeEnum codeEnum) {
        APIResult apiResult = new APIResult();
        apiResult.setCode(codeEnum.getCode());
        apiResult.setMsg(codeEnum.getMsg());
        return apiResult;
    }
```

除了这个以外，还定义一个异常对象 APPException，用来统一包装内部的各种异常。

上面的代码很简单，但是呢可以说比较繁琐，重复代码也比较多，每个接口都需要使用 try...catch 包，然后使用 APIResult包括正常的返回信息与错误信息。

第二呢，接口对象只能返回 APIResult，真实业务对象只能隐藏在 APIResult中。这样不太优雅，另不能很直观知道真实业务对象。

重构之后

下面我们开始重构上面的代码，主要目的是去除重复的那一坨try...catch 代码。

这次重构我们需要使用Spring 注解 @ControllerAdvice以及 ResponseBodyAdvice，我们先来看下构的代码。

ps: ResponseBodyAdvice来自 Spring 4.2 API，如果各位同学需要使用这个的话，可能需要升级 Spring 版本。

改写返回信息

首先我们需要实现 ResponseBodyAdvice，实现我们自己的处理类。

```
@ControllerAdvice
public class CustomResponseAdvice implements ResponseBodyAdvice {
    /**
     * 是否需要处理返回结果
     * @param methodParameter
     * @param aClass
     * @return
     */
    @Override
    public boolean supports(MethodParameter methodParameter, Class aClass) {
        System.out.println("In supports() method of " + getClass().getSimpleName());
        return true;
    }

    /**
     * 处理返回结果
     * @param body
     * @param methodParameter
     * @param mediaType
     * @param aClass
     * @param serverHttpRequest
     * @param serverHttpResponse
     * @return
     */
    @Override
    public Object beforeBodyWrite(Object body, MethodParameter methodParameter, MediaType mediaType, Class aClass, ServerHttpRequest serverHttpRequest, ServerHttpResponse serveHttpResponse) {
        System.out.println("In beforeBodyWrite() method of " + getClass().getSimpleName());
        if (body instanceof APIResult) {
            return body;
        }
        return APIResult.success(body);
    }
}
```

实现上面的接口，我们就可以在 beforeBodyWrite方法里，修改返回结果了。

上面代码中，只是简单使用 APIResult包装了返回结果，然后返回。其实我们还可以在此增加一些额逻辑，比如说如接口返回信息由加密的需求，我们可以在这一层统一加密。

另外，这里判断一下 body 是否 APIResult类，如果是就直接返回，不做修改。

这么做一来兼容之前的老接口，这是因为默认情况下，我们自己实现的 CustomResponseAdvice类将会对所有的 Controller 生效。

如果不做判断，以前的老接返回就会被包装了两层 APIResul,影响 APP 解析。

除此之外，如果大家担心这个修改对以前的老接口有影响的话，可以使用下面的方式，只对指定的方生效。

首先自定义一个注解，比如说：

```
@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CustomResponse {
}
```

然后将其标注在需要改动的方法中，然后我们在 ResponseBodyAdvice#supports中判断具体方法有没有自定义注解 CustomResponse，如果存在，返回 true，这就代表最后将会修改返回类。如果存在，则返回 false，那么就会跟以前流程一样。

```
/**
 * 是否需要处理返回结果
 *
 * @param methodParameter
 * @param aClass
 * @return
 */
@Override
public boolean supports(MethodParameter methodParameter, Class aClass) {
    System.out.println("In supports() method of " + getClass().getSimpleName());
    Method method = methodParameter.getMethod();
    return method.isAnnotationPresent(CustomResponse.class);
}
```

全局异常处理

上面的代码重构之后，将重复代码抽取了出来，整体的代码就剩下我们的业务逻辑，这样就变得非常洁优雅。

不过，上面的重构的代码，还是存在问题，主要是异常的处理。

如果上面的业务代码抛出了异常，那么接口将会返回堆栈错误信息，而不是我们定义的错误信息。所下面我们这个，再次优化一下。

这次我们主要需要使用 @ExceptionHandler注解，这个注解需要与 @ControllerAdvice 一起使用。

```
@Slf4j
@ControllerAdvice
public class CustomExceptionHandler {

    @ExceptionHandler(Exception.class)
    @ResponseBody
    public APIResult handleException(Exception e) {
        log.error("系统异常", e);
        return APIResult.error(RetCodeEnum.FAILED);
    }

    @ExceptionHandler(APPException.class)
    @ResponseBody
```

```
    public APIResult handleAPPException(APPException e) {
        log.error("内部异常", e);
        return APIResult.error(e.getCode(), e.getMsg());
    }

}
```

使用这个 @ExceptionHandler，将会拦截相应的异常，然后将会调用的相应方法处理异常。这里我就使用 APIResult包装一些错误信息返回。

总结

我们可以使用 @ControllerAdvice加 ResponseBodyAdvice 拦截返回结果，统一做出一些修改。这就可以使用的业务代码非常简洁，优雅。

另外，针对业务代码的中，我们可以使用 @ExceptionHandler注解，统一做一个全局异常处理，这就可以无缝的跟 ResponseBodyAdvice结合。

不过这里需要一点，我们实现的 ResponseBodyAdvice 类，一定需要跟 @ControllerAdvice配合一使用哦，至于具体原因，下篇文章小黑哥分析原来的时候，再具体解释哦。敬请期待哦~