8-统一全局异常处理

1.设计一个优秀的异常处理机制

一、异常处理的乱象例举

乱象一：捕获异常后只输出到控制台

前端js-ajax代码

```javascript
$.ajax({
  type: "GET",
  url: "/user/add",
  dataType: "json",
  success: function(data){
    alert("添加成功");
  }
});
```

后端业务代码

```java
try {
    // do something
} catch (XyyyyException e) {
    e.printStackTrace();
}
```

问题：

1. 后端直接将异常捕获，而且只做了日志打印。用户体验非常差，一旦后台出错，用户没有任何感知，页面无状态。
2. 后端只给出前端异常结果，没有给出异常的原因的描述。用户不知道是自己操作输入错误，还是系统bug。用户无法判断自己需要等一下再操作？还是继续下一步？
3. 如果没有人去经常关注服务端日志，不会有人发现系统出现异常。

乱象二：混乱的返回方式

前端代码

JavaScript | 复制代码

```javascript
$.ajax({
    type: "GET",
    url: "/goods/add",
    dataType: "json",
    success: function(data) {
        if (data.flag) {
            alert("添加成功");
        } else {
            alert(data.message);
        }
    },
    error: function(data){
        alert("添加失败");
    }
});
```

后端代码

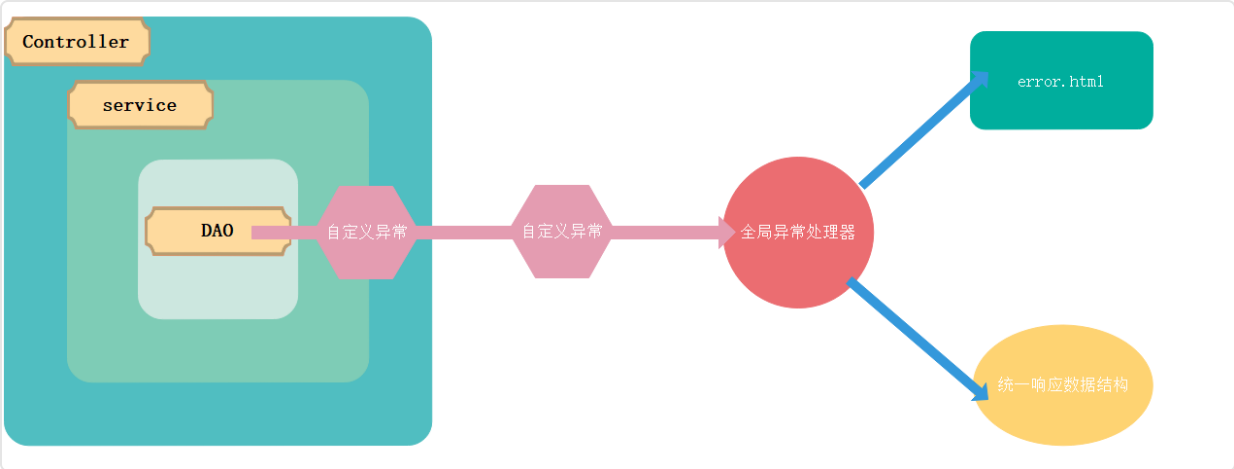
Java | 复制代码

```java
@RequestMapping("/goods/add")
@ResponseBody
public Map add(Goods goods) {
    Map map = new HashMap();
    try {
        // do something
        map.put(flag, true);
    } catch (Exception e) {
        e.printStackTrace();
        map.put("flag", false);
        map.put("message", e.getMessage());
    }
    reutrn map;
}
```

问题：

1. 每个人返回的数据有每个人自己的规范，你叫flag他叫isOK，你的成功code是0，它的成功code是0000。这样导致后端书写了大量的异常返回逻辑代码，前端也随之每一个请求一套异常处理逻辑。很多重复代码。
2. 如果是前端后端一个人开发还勉强能用，如果前后端分离，这就是系统灾难。

二、该如何设计异常处理



面向相关方友好

1. 后端开发人员职责单一，只需要将异常捕获并转换为自定义异常一直对外抛出。不需要去想页面跳转404，以及异常响应的数据结构的设计。
2. 面向前端人员友好，后端返回给前端的数据应该有统一的数据结构，统一的规范。不能一个人一个响应的数据结构。而在此过程中不需要后端开发人员做更多的工作，交给全局异常处理器去处理“异常”到“响应数据结构”的转换。
3. 面向用户友好，用户能够清楚的知道异常产生的原因。这就要求自定义异常，全局统一处理，ajax接口请求响应统一的异常数据结构，页面模板请求统一跳转到404页面。
4. 面向运维友好，将异常信息合理规范的持久化，以日志的形式存储起来，以便查询。

为什么要将系统运行时异常捕获，转换为自定义异常抛出?

因为用户不认识ConnectionTimeOutException类似这种异常是什么东西，但是转换为自定义异常就要求程序员对运行时异常进行一个翻译，比如：自定义异常里面应该有message字段，后端程序员应该明确的在message字段里面用面向用户的友好语言，说明服务端发生了什么。

三、开发规范

1. Controller、Service、DAO层拦截异常转换为自定义异常，不允许将异常私自截留。必须对外抛出。
2. 统一数据响应代码，使用http状态码，不要自定义。自定义不方便记忆，HTTP状态码程序员都知道。但是太多了程序员也记不住，在项目组规定范围内使用几个就可以。比如：200请求成功，400用户输入错误导致的异常，500系统内部异常，999未知异常。
3. 自定义异常里面有message属性，用对用户友好的语言描述异常的发生情况，并赋值给message.

4. 不允许对父类Exception统一catch，要分小类catch，这样能够清楚地将异常转换为自定义异常传递给前端。

2.自定义异常和相关数据结构

一、该如何设计数据结构

1. CustomException 自定义异常。核心要素包含异常错误编码（400,500）、异常错误信息message。
2. ExceptionTypeEnum 枚举异常分类，将异常分类固化下来，防止开发人员思维发散。
3. AjaxResponse 用于响应HTTP 请求的统一数据结构。

二、枚举异常的类型

为了防止开发人员大脑发散，每个开发人员都不断的发明自己的异常类型，我们需要规定好异常的类型(枚举)。比如：系统异常、用户（输入）操作导致的异常、其他异常等。

Java | 复制代码

```java
package com.mqxu.boot.exception.enums;

/**
 * @description: 异常类型枚举
 * @author: mqxu
 * @date: 2022-04-11
 **/
public enum CustomExceptionType {
    /**
     * 客户端异常
     */
    USER_INPUT_ERROR(400, "您输入的数据错误或您没有权限访问资源！"),

    /**
     * 服务器异常
     */
    SYSTEM_ERROR(500, "系统出现异常，请您稍后再试或联系管理员！"),

    /**
     * 未知异常
     */
    OTHER_ERROR(999, "系统出现未知异常，请联系管理员！");

    CustomExceptionType(int code, String desc) {
        this.code = code;
        this.desc = desc;
    }

    /**
     * 异常类型状态码
     */
    private final int code;

    /**
     * 异常类型中文描述
     */
    private final String desc;


    public String getDesc() {
        return desc;
    }

    public int getCode() {
        return code;
```

```
46          }
47      }
```

- 最好不要超过5个，否则开发人员将会记不住，也不愿意去记。
- 这里的code表示异常类型的唯一编码，为了方便大家记忆，就使用Http状态码400、500
- 这里的desc是通用的异常描述，在创建自定义异常的时候，为了给用户更友好的回复，通常异常信息描述应该更具体更友好。

三、自定义异常

- 自定义异常有两个核心内容，一个是code。使用CustomExceptionType 来限定范围。
- 另外一个是message，这个message信息是要最后返回给前端的，所以需要用友好的提示来表达异常发生的原因或内容

Java | 复制代码

```java
package com.mqxu.boot.exception.exception;

import com.mqxu.boot.exception.enums.CustomExceptionType;

/**
 * @description: 自定义异常
 * @author: mqxu
 * @date: 2022-04-11
 **/
public class CustomException extends RuntimeException {
    /**
     * 异常错误编码
     */
    private int code;
    /**
     * 异常信息
     */
    private String message;

    private CustomException() {
    }

    public CustomException(CustomExceptionType customExceptionType) {
        this.code = customExceptionType.getCode();
        this.message = customExceptionType.getDesc();
    }

    public CustomException(CustomExceptionType customExceptionType, String message) {
        this.code = customExceptionType.getCode();
        this.message = message;
    }

    public int getCode() {
        return code;
    }

    @Override
    public String getMessage() {
        return message;
    }
}
```

四、请求接口统一响应数据结构

为了解决不同的开发人员使用不同的结构来响应给前端，导致规范不统一，开发混乱的问题。我们使用如下代码定义统一数据响应结构

- isok表示该请求是否处理成功（即是否发生异常）。true表示请求处理成功，false表示处理失败。
- code对响应结果进一步细化，200表示请求成功，400表示用户操作导致的异常，500表示系统异常，999表示其他异常。与CustomExceptionType枚举一致。
- message：友好的提示信息，或者请求结果提示信息。如果请求成功这个信息通常没什么用，如果请求失败，该信息需要展示给用户。
- data：通常用于查询数据请求，成功之后将查询数据响应给前端。

Java | 复制代码

```java
package com.mqxu.boot.exception.utils;

import com.mqxu.boot.exception.enums.CustomExceptionType;
import com.mqxu.boot.exception.exception.CustomException;
import lombok.Data;

/**
 * @description: 请求接口统一响应数据结构
 * @author: mqxu
 * @date: 2022-04-11
 **/
@Data
public class AjaxResponse {
    /**
     * 请求响应状态码
     */
    private int code;
    /**
     * 请求结果描述信息
     */
    private String message;
    /**
     * 请求结果数据（通常用于查询操作）
     */
    private Object data;

    private AjaxResponse() {
    }

    /**
     * 请求出现异常时的响应数据封装
     *
     * @param e e
     * @return AjaxResponse
     */
    public static AjaxResponse error(CustomException e) {
        AjaxResponse resultBean = new AjaxResponse();
        resultBean.setCode(e.getCode());
        resultBean.setMessage(e.getMessage());
        return resultBean;
    }

    /**
     * 请求出现异常时的响应数据封装
     *
```

```
46          * @param customExceptionType customExceptionType
47          * @param errorMessage        errorMessage
48          * @return AjaxResponse
49          */
50         public static AjaxResponse error(CustomExceptionType
    customExceptionType, String errorMessage) {
51             AjaxResponse resultBean = new AjaxResponse();
52             resultBean.setCode(customExceptionType.getCode());
53             resultBean.setMessage(errorMessage);
54             return resultBean;
55         }
56
57         /**
58          * 请求成功的响应，不带查询数据（用于删除、修改、新增接口）
59          *
60          * @return AjaxResponse
61          */
62         public static AjaxResponse success() {
63             AjaxResponse ajaxResponse = new AjaxResponse();
64             ajaxResponse.setCode(200);
65             ajaxResponse.setMessage("请求响应成功!");
66             return ajaxResponse;
67         }
68
69         /**
70          * 请求成功的响应，带有查询数据（用于数据查询接口）
71          *
72          * @param obj obj
73          * @return AjaxResponse
74          */
75         public static AjaxResponse success(Object obj) {
76             AjaxResponse ajaxResponse = new AjaxResponse();
77             ajaxResponse.setCode(200);
78             ajaxResponse.setMessage("请求响应成功!");
79             ajaxResponse.setData(obj);
80             return ajaxResponse;
81         }
82
83         /**
84          * 请求成功的响应，带有查询数据（用于数据查询接口）
85          *
86          * @param obj     obj
87          * @param message message
88          * @return AjaxResponse
89          */
90         public static AjaxResponse success(Object obj, String message) {
91             AjaxResponse ajaxResponse = new AjaxResponse();
92             ajaxResponse.setCode(200);
```

```
93            ajaxResponse.setMessage(message);
94            ajaxResponse.setData(obj);
95            return ajaxResponse;
96        }
97
98    }
```

对于不同的场景，提供了四种构建AjaxResponse 的方法。

- 当请求成功的情况下，可以使用AjaxResponse.success()构建返回结果给前端。
- 当查询请求等需要返回业务数据，请求成功的情况下，可以使用AjaxResponse.success(data)构建返回结果给前端。携带结果数据。
- 当请求处理过程中发生异常，需要将异常转换为CustomException ，然后在控制层使用AjaxResponse error(CustomException)构建返回结果给前端。
- 在某些情况下，没有任何异常产生，我们判断某些条件也认为请求失败。这种使用AjaxResponse error(customExceptionType,errorMessage)构建响应结果。

五、使用示例如下：

例如：更新操作，Controller无需返回额外的数据

Java | 复制代码

```
1    return AjaxResponse.success();
```

查询接口，Controller需返回结果数据(data可以是任何类型数据)

Java | 复制代码

```
1     return AjaxResponse.success(data);
```

3.通用全局异常处理逻辑

一、通用异常处理逻辑

程序员的异常处理逻辑要十分的单一：无论在Controller层、Service层还是什么其他位置，程序员只负责一件事：**那就是捕获异常，并将异常转换为自定义异常。使用用户友好的信息去填充**

CustomException的message,并将CustomException抛出去。

Java | 复制代码

```java
package com.mqxu.boot.exception.service;

import com.mqxu.boot.exception.consts.MsgConsts;
import com.mqxu.boot.exception.enums.CustomExceptionType;
import com.mqxu.boot.exception.exception.CustomException;
import org.springframework.stereotype.Service;

/**
 * @description: 通用异常处理逻辑
 * @author: mqxu
 * @date: 2022-04-11
 **/
@Service
public class ExceptionService {

    /**
     * 服务层，模拟系统异常
     */
    public void systemBizError() {
        try {
            Class.forName("com.mysql.jdbc.cj.Driver");
        } catch (ClassNotFoundException e) {
            throw new CustomException(
                    CustomExceptionType.SYSTEM_ERROR,
                    "在XXX业务，myBiz()方法内，出现ClassNotFoundException，请将该信息告知管理员");
        }
    }

    /**
     * 服务层，模拟用户输入数据导致的校验异常
     *
     * @param input 用户输入
     */
    public void userBizError(int input) {
        //模拟业务校验失败逻辑
        if (input < 0) {
            throw new CustomException(CustomExceptionType.USER_INPUT_ERROR, MsgConsts.INPUT_ERROR);
        }
    }

}
```

二、全局异常处理器

通过团队内的编码规范的要求，我们已经知道了：不允许程序员截留处理Exception，必须把异常转换为自定义异常CustomException全都抛出去。那么程序员把异常跑出去之后由谁来处理？那就是ControllerAdvice。

ControllerAdvice注解的作用就是监听所有的Controller，一旦Controller抛出CustomException，就会在@ExceptionHandler(CustomException.class)注解的方法里面对该异常进行处理。处理方法很简单就是**使用AjaxResponse.error(e)包装为通用的接口数据结构返回给前端。**

Java | 复制代码

```java
package com.mqxu.boot.exception.handler;

import com.mqxu.boot.exception.enums.CustomExceptionType;
import com.mqxu.boot.exception.exception.CustomException;
import com.mqxu.boot.exception.utils.AjaxResponse;
import org.springframework.validation.BindException;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * @description: 全局异常处理器
 * @author: mqxu
 * @date: 2022-04-11
 **/
@ControllerAdvice
public class WebExceptionHandler {

    /**
     * 处理程序员主动转换的自定义异常
     *
     * @param e 异常
     * @return AjaxResponse
     */
    @ExceptionHandler(CustomException.class)
    @ResponseBody
    public AjaxResponse customerException(CustomException e) {
        if (e.getCode() == CustomExceptionType.SYSTEM_ERROR.getCode()) {
            //400异常不需要持久化，将异常信息以友好的方式告知用户就可以
            //将500异常信息持久化处理，方便运维人员处理
        }
        return AjaxResponse.error(e);
    }

    /**
     * 处理程序员在程序中未能捕获（遗漏的）异常
     *
     * @param e 异常
     * @return AjaxResponse
     */
    @ExceptionHandler(Exception.class)
    @ResponseBody
    public AjaxResponse exception(Exception e) {
```

```
46            //TODO 将异常信息持久化处理，方便运维人员处理
47            return AjaxResponse.error(new
    CustomException(CustomExceptionType.OTHER_ERROR));
48        }
49
50        @ExceptionHandler(MethodArgumentNotValidException.class)
51        @ResponseBody
52        public AjaxResponse
    handleBindException(MethodArgumentNotValidException ex) {
53            FieldError fieldError = ex.getBindingResult().getFieldError();
54            assert fieldError != null;
55            return AjaxResponse.error(new
    CustomException(CustomExceptionType.USER_INPUT_ERROR,
    fieldError.getDefaultMessage()));
56        }
57
58        @ExceptionHandler(BindException.class)
59        @ResponseBody
60        public AjaxResponse handleBindException(BindException ex) {
61            FieldError fieldError = ex.getBindingResult().getFieldError();
62            assert fieldError != null;
63            return AjaxResponse.error(new
    CustomException(CustomExceptionType.USER_INPUT_ERROR,
    fieldError.getDefaultMessage()));
64        }
65
66
67        @ExceptionHandler(IllegalArgumentException.class)
68        @ResponseBody
69        public AjaxResponse
    handleIllegalArgumentException(IllegalArgumentException e) {
70            return AjaxResponse.error(new
    CustomException(CustomExceptionType.USER_INPUT_ERROR, e.getMessage()));
71        }
72    }
```

三、测试一下

随便找一个API，注入ExceptionService访问测试一下

四、业务状态与HTTP协议状态一致

不知道大家有没有注意到一个问题(看上图)？这个问题就是我们的AjaxResponse的code是400，但是真正的HTTP协议状态码是200。

- AjaxResponse的code是400代表的是业务状态，也就是说用户的请求业务失败了
- 但是HTTP请求是成功的，也就是说数据是正常返回的。

在很多的公司开发RESTful服务时，要求HTTP状态码能够体现业务的最终执行状态，所以说：我们有必要让业务状态与HTTP协议Response状态码一致。

```java
@Component
@ControllerAdvice
public class GlobalResponseAdvice implements ResponseBodyAdvice {
    @Override
    public boolean supports(MethodParameter returnType, Class converterType) {
        //return returnType.hasMethodAnnotation(ResponseBody.class);
        return true;
    }
    @Override
    public Object beforeBodyWrite(Object body,
                                  MethodParameter returnType,
                                  MediaType selectedContentType,
                                  Class selectedConverterType,
                                  ServerHttpRequest request,
                                  ServerHttpResponse response) {

        //如果响应结果是JSON数据类型
        if(selectedContentType.equalsTypeAndSubtype(
                MediaType.APPLICATION_JSON)){
                //为HTTP响应结果设置状态码，状态码就是AjaxResponse的code，二者达到统一
                response.setStatusCode(
                        HttpStatus.valueOf(((AjaxResponse) body).getCode())
                );
                return body;
        }
        return body;
    }
}
```

- 实现ResponseBodyAdvice 接口的作用是：在将数据返回给用户之前，做最后一步的处理。也就是说，ResponseBodyAdvice 的处理过程在全局异常处理的后面。

五、进一步优化

我们已经知道了，ResponseBodyAdvice 接口的作用是：在将数据返回给用户之前，做最后一步的处理。将上文的GlobalResponseAdvice 中beforeBodyWrite方法代码优化如下。

```java
//如果响应结果是JSON数据类型
if(selectedContentType.equalsTypeAndSubtype(
        MediaType.APPLICATION_JSON)){
    //如果响应结果是AjaxResponse
    if(body instanceof AjaxResponse){
        response.setStatusCode(
                HttpStatus.valueOf(((AjaxResponse) body).getCode())
        );
        return body;
    }else{//如果响应结果不是AjaxResponse
        response.setStatusCode(HttpStatus.OK);
        return AjaxResponse.success(body);
    }
}
return body;
```

- 如果Controller或全局异常处理响应的结果body是AjaxResponse，就直接return给前端。
- 如果Controller或全局异常处理响应的结果body不是AjaxResponse，就将body封装为AjaxResponse之后再return给前端。

我们之前的代码是这样写的，比如：某个controller方法返回值

Java | 复制代码

```java
return AjaxResponse.success(objList);
```

现在就可以这样写了，因为在GlobalResponseAdvice 里面会统一再封装为AjaxResponse。

Java | 复制代码

```java
return objList;
```

最终代码如下：

Java | 复制代码

```java
package com.mqxu.boot.exception.advice;

import com.mqxu.boot.exception.utils.AjaxResponse;
import org.springframework.core.MethodParameter;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.bind.annotation.ControllerAdvice;
import
org.springframework.web.servlet.mvc.method.annotation.ResponseBodyAdvice;

/**
 * @description: 全局业务状态通知
 * @author: mqxu
 * @date: 2022-04-11
 **/
@Component
@ControllerAdvice
public class GlobalResponseAdvice implements ResponseBodyAdvice {

    @Override
    public boolean supports(MethodParameter methodParameter, Class
aClass) {
        return true;
    }

    @Override
    public Object beforeBodyWrite(Object body,
                                  MethodParameter methodParameter,
                                  MediaType mediaType,
                                  Class aClass,
                                  ServerHttpRequest serverHttpRequest,
                                  ServerHttpResponse serverHttpResponse)
{
        //如果响应结果是JSON数据类型
        if (mediaType.equalsTypeAndSubtype(
                MediaType.APPLICATION_JSON)) {
            if (body instanceof AjaxResponse ajaxResponse) {
                //999 不是标准的HTTP状态码，特殊处理
                if (ajaxResponse.getCode() != 999) {
                    serverHttpResponse.setStatusCode(HttpStatus.valueOf(
                            ajaxResponse.getCode()
                    ));
```

```
43                      }
44                      return body;
45                  } else {
46                      serverHttpResponse.setStatusCode(HttpStatus.OK);
47                      return AjaxResponse.success(body);
48                  }
49
50              }
51              return body;
52          }
53      }
```

4.服务端数据校验异常处理逻辑

一、异常校验的规范及常用注解

在Web开发时，对于请求参数，一般上都需要进行参数合法性校验的，原先的写法是一个个字段一个个去判断，这种方式太不通用了，Java的JSR 303: Bean Validation规范就是解决这个问题的。
JSR 303只是个规范，并没有具体的实现，目前通常是用 hibernate-validator进行统一参数校验。

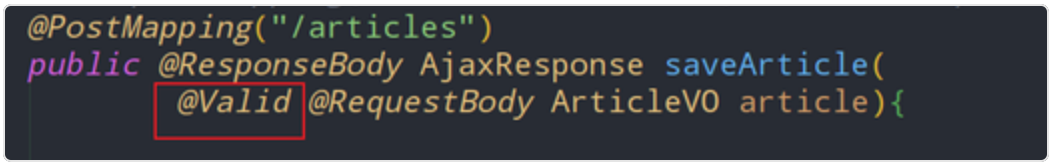
JSR303定义的校验类型

Constraint	详细信息
@Null	被注释的元素必须为 null
@NotNull	被注释的元素必须不为 null
@AssertTrue	被注释的元素必须为 true
@AssertFalse	被注释的元素必须为 false
@Min(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@Max(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@DecimalMin(value)	被注释的元素必须是一个数字，其值必须大于等于指定的最小值
@DecimalMax(value)	被注释的元素必须是一个数字，其值必须小于等于指定的最大值
@Size(max, min)	被注释的元素的大小必须在指定的范围内
@Digits (integer, fraction)	被注释的元素必须是一个数字，其值必须在可接受的范围内
@Past	被注释的元素必须是一个过去的日期
@Future	被注释的元素必须是一个将来的日期
@Pattern(value)	被注释的元素必须符合指定的正则表达式

Hibernate Validator 附加的 constraint

Constraint	详细信息
@Email	被注释的元素必须是电子邮箱地址
@Length	被注释的字符串的大小必须在指定的范围内
@NotEmpty	被注释的字符串的必须非空
@Range	被注释的元素必须在合适的范围内

用法:把以上注解加在ArticleVO的属性字段上，然后在参数校验的方法上加@Valid注解 如:



当用户输入参数不符合注解给出的校验规则的时候，会抛出BindException或MethodArgumentNotValidException。

二、Assert断言与IllegalArgumentException

之前给大家讲通用异常处理的时候，用户输入异常判断是这样处理的。这种方法也是可以用的，但是我们学了这么多的知识，可以优化一下。

Java | 复制代码

```java
//服务层，模拟用户输入数据导致的校验异常
public void userBizError(int input)  {
    if(input < 0){ //模拟业务校验失败逻辑
        throw new CustomException(
                CustomExceptionType.USER_INPUT_ERROR,
                "您输入的数据不符合业务逻辑，请确认后重新输入！");
    }

    //…… 其他的业务
}
```

更好的写法是下面这样的，使用org.springframework.util.Assert断言input >= 0，如果不满足条件就抛出IllegalArgumentException，参数不合法的异常。

```java
//服务层，模拟用户输入数据导致的校验异常
public void userBizError(int input)  {
    Assert.isTrue(input >= 0,"您输入的数据不符合业务逻辑，请确认后重新输入！");

    //…… 其他的业务
}
```

org.springframework.util.Assert断言提供了大量的断言方法，针对各种数据类型进行数据合法性校验，使用它我们编写代码更方便。

三、友好的数据校验异常处理（用户输入异常的全局处理）

我们已知当数据校验失败的时候，会抛出异常BindException或MethodArgumentNotValidException。所以我们对这两种异常做全局处理，防止程序员重复编码带来困扰。

```java
@ExceptionHandler(MethodArgumentNotValidException.class)
@ResponseBody
public AjaxResponse handleBindException(MethodArgumentNotValidException ex) {
  FieldError fieldError = ex.getBindingResult().getFieldError();
  return AjaxResponse.error(new CustomException(CustomExceptionType.USER_INPUT_ERROR,
          fieldError.getDefaultMessage()));
}


@ExceptionHandler(BindException.class)
@ResponseBody
public AjaxResponse handleBindException(BindException ex) {
  FieldError fieldError = ex.getBindingResult().getFieldError();
  return AjaxResponse.error(new CustomException(CustomExceptionType.USER_INPUT_ERROR,
          fieldError.getDefaultMessage()));
}
```

我们已知使用org.springframework.util.Assert断言，如果不满足条件就抛出IllegalArgumentException。可以使用下面的全局异常处理函数

Java | 复制代码

```java
@ExceptionHandler(IllegalArgumentException.class)
@ResponseBody
public AjaxResponse
handleIllegalArgumentException(IllegalArgumentException e) {
  return AjaxResponse.error(
          new CustomException(CustomExceptionType.USER_INPUT_ERROR,
                  e.getMessage())
  );
}
```