异常处理规范

V1.0

**京东生活旅行**

**版本修改记录**

| **版本号** | **版本修改说明** | **作者/日期** | **审批人/日期** | **生效日期** |
| --- | --- | --- | --- | --- |
| 1.0 |  | 架构师/20190226 | 架构师/20190226 | 20190226 |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

说明： C：Create，初始创建；A：Add，增加内容；M：Mod，修改；D：Del，删除

目 录

[1. 概述 5](#_Toc2876247)

[1.1. 适用范围 5](#_Toc2876248)

[2. 异常处理规范 5](#_Toc2876249)

[2.1 尽量少用检查型异常，推荐使用运行时异常 5](#_Toc2876252)

[2.2 适合使用检查异常的原则 6](#_Toc2876253)

[2.3 异常转换 6](#_Toc2876254)

[2.4 禁止捕获 Throwable 类 7](#_Toc2876255)

[2.5 禁止直接抛出Exception，推荐抛出其子类。 8](#_Toc2876256)

[2.6 优先抛出明确异常 8](#_Toc2876257)

[2.7 尽量使用标准的异常 9](#_Toc2876258)

[2.8 每个方法抛出的异常都要有文档 10](#_Toc2876259)

[2.9 异常的输出 11](#_Toc2876260)

[2.10 优先捕获最具体的异常 11](#_Toc2876261)

[2.11 统一异常处理 12](#_Toc2876262)

[2.12 异常消息中必须含业务信息 12](#_Toc2876263)

[2.13 不要忽略异常 13](#_Toc2876264)

[2.14 禁止记录日志和抛出错误同时出现 13](#_Toc2876265)

[2.15 分层异常处理 13](#_Toc2876266)

[2.16 资源正确释放与关闭 18](#_Toc2876267)

[3. JSF异常处理模板参考 18](#_Toc2876268)

[3.1 统一响应格式(使用RpcResult类) 18](#_Toc2876273)

[3.2 业务错误码统一继承BaseErrorCodeEnum. 19](#_Toc2876274)

[3.3 统一的基础异常类RpcException 20](#_Toc2876275)

[3.4 统一的异常处理 23](#_Toc2876276)

[3.5 JSF的参数校验（Validation）功能 24](#_Toc2876277)

[3.6 其他原则 24](#_Toc2876278)

## 概述

旨在建立研发异常处理标准，有效预防异常的再次发生，进而提升整体研发质量及效率，特订定本规范。

### 适用范围

生旅研发所有后台应用系统。

## 异常处理规范



### 尽量少用检查型异常，推荐使用运行时异常

使用运行时异常还是非运行时异常关键在于：非正常的分支流程应该由方法调用者检查还是方法本身检查。如果检查非正常流程分支是调用者的责任。那么出现非正常流程分支时则是一个编程错误，会导致当前线程终止，所以应该抛出运行时异常。如

public void takeMoney(int amount) {

if (account < amount) {

//抛出运行时异常

throw new IllegalArgumentException("账户余额不足");

}

account -= amount;

}

### 适合使用检查异常的原则

如果检查非正常流分支是函数本身的责任，那么我们则必须在方法中声明可能会抛出这种异常，希望调用者有意识地采取措施，那么抛出检查型异常，提醒了调用者注意这个异常并采取相应的处理措施。如：

//BalanceException extends Exception

public void takeMoney(int amount) throws BalanceException {

if(account < amount) throw new BalanceException();

account -= amount;

}

如果不清楚到底该不该让调用者采取措施，那么抛出非检查型异常

### 异常转换

如果需要在异常中添加其他信息;

高层的实现应该捕获低层的异常，同时抛出一个可以按高层抽象进行解释的对象。

以上两种+情况，都需要用到异常转换。需要注意的是，进行转换时一定要在构造方法中传入原异常的throwable对象，这样可以保留原异常栈信息，而不是把原异常用另一个异常完全替换掉。如：

public void doSomeThing(String input) {

try{

…

} catch (IoException ex) {

throw new BizException(“输入参数格式错误！”);//错误

log.error(“error occurred.”,ex);//catch之后封装成新异常抛出的时候, 不要记录日志. 因为会有上层来处理记录日志

throw new BizException(“输入参数格式错误！“,ex);//正确

}

} catch(BizExeption ex){

throw new RpcException(“输入参数格式错误！”);//错误

log.error(“error occurred.”,ex);//catch之后封装成新异常抛出的时候, 不要记录日志. 因为会有上层来处理记录日志

throw new RpcException(“输入参数格式错误！“,ex);//正确

### 禁止捕获 Throwable 类

Try {

…

} catch(Throwable t){ //错误

}

### 禁止直接抛出Exception，推荐抛出其子类。

throw new Exception("该用户不存在, userid:"+userid); //错误

throw new BizException("该用户不存在, userid:"+userid);//正确

### 优先抛出明确异常

抛出的异常越明确越好。例如，抛出一个NumberFormatException来替换一个IllegalArgumentException。避免抛出一个不明确的异常。

public void doNotDoThis() throws Exception { //错误

…

}

public void doSomeThing() throws IllegalArgumentException{ //其次

…

}

public void doThis() throws NumberFormatException { //最好

…

}

### 尽量使用标准的异常

Java标准库提供了很多异常类，这些类覆盖了程序执行过程中可能发生的大多数错误。应用最广泛的如下表。

|  |  |
| --- | --- |
| 异常类 | 设计意图 |
| NullPointerException | 当应用程序试图在需要对象的地方使用 null 时，抛出该异常 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数 |
| IllegalStateException | 在非法或不适当的时间调用方法时产生的信号。换句话说，即应用程序没有处于请求操作所要求的适当状态下 |
| UnsupportedOperationException | 当不支持请求的操作时，抛出该异常 |
| EnumConstantNotPresentException | 枚举常量不存在异常。当应用试图通过名称和枚举类型访问一个枚举对象，但该枚举对象并不包含常量时，抛出该异常。 |
| IOException | 输入输出异常 |

public static PayCodeEnum getByCode(String code) {  
 for (PayCodeEnum em : values()) {  
 if (em.code.equals(code)) {  
 return em;  
 }  
 }  
 return null; //错误

throw new EnumConstantNotPresentException(PayCodeEnum.class, "Unknown value ["+code +"]"); //正确

}

### 每个方法抛出的异常都要有文档

方法中抛出异常（未被捕获），请确保向Javadoc添加@throws声明，准确记录每个异常被抛出的条件，这样方法的调用者会意识到可能会抛出异常，从一开始就实施适当的异常处理和恢复（回退）逻辑。

注意不要使用throws关键字将运行时异常包含在方法的声明中。

/\*\*

\* 方法描述

\*

\* @param input

\* @throws IOException if an I/O error occurs. //正确

\* @throws BizException if … occurs. //正确

\*

\*\*/

//BalanceException extends Exception

public void doSomeThing(String input) throws IOException {

…

throw BizException(“…”);

}

### 异常的输出

打印异常的logger，推荐用error级别。避免线上将日志级别调到ERROR级别后无法打出INFO级别的异常logger信息。

try {

…

} catch (NumberFormatException ex) {

log.error(“error”+ e)； //错误：能打印出异常信息

log.error(“error”,e)； //正确，打印出异常的堆栈信息，便于排查问题

}

### 优先捕获最具体的异常

try {

…

} catch (NumberFormatException ex) {

Log.error(ex);

} catch (IllegalArgumentException ex) {

Log.error(ex);

}

### 统一异常处理

统一目的是确保出去的异常是可控的， 调用方能够明白的异常信息。

自定义异常基类BaseException。

BaseException推荐继承自RuntimeException。

WebException、BizException、DaoException分别映射到Controller层、业务逻辑层、数据访问层。都继承自BaseException。

### 异常消息中必须含业务信息

异常的字符串应该包含所有“对该异常有贡献“的参数和域的值；

User user = userservice.getById(userid);

if (user == null) {

throw new BizException("该用户不存在"); //错误

throw new BizException("该用户不存在, userid:"+userid); //正确

}

### 不要忽略异常

禁止异常截获后不做任何处理，空的catch块会使异常达不到应有的目的。

try {

....

}

catch (BizException ex) { //禁止此类没有任何处理

}

### 禁止记录日志和抛出错误同时出现

try {

…

} catch (NumberFormatException ex) {

log.error(ex); //错误示例统一异常处理框架会打印异常，避免重复打印。

throw e; //错误示例

}

### 分层异常处理

#### Dao层

推荐不做任何异常处理，直接交给manager层处理；

#### Manager层

@Transactional 注解应该加在 Manager层上。

对于一些关键问题, 比如 Checked Exception 或者数据的问题, 应该及时 抛出DAOException 异常, 以确保事务完整.

throw new DaoException时的日志, 为了避免日志重复, 不需要 log 日志输出.

#### Service层

推荐该层抛出BizException异常

#### controller层

用spring的@ControllerAdvice和@ExceptionHandler注解定义类似拦截异常的类，实现对异常的拦截并处理。

@ControllerAdvice

@Slf4j

public class GlobalExceptionHandler {

/\*\*

\* @param e

\* @description 处理所有不可知的异常

\*/

@ExceptionHandler({Exception.class})

@ResponseBody

public ResponseResult globalExceptionHandler(Exception e) {

log.error(e.getMessage(), e);

return new ResponseResultUtil().error(ResponseCodeEnum.OPERATE\_FAIL);

}

/\*\*

\* @param e 异常

\* @description 处理所有业务异常

\*/

@ExceptionHandler({BizException.class})

@ResponseBody

public ResponseResult errorBizExceptionHandler(BizException e) {

log.error(“Biz Exception!”,e);

return new ResponseResultUtil().error(e.getCode(), e.getMessage());

}

}

/\*\*

\* @description 请求响应对象

\*/

@Setter

@Getter

public final class ResponseResult<T> {

/\*\*

\* @description 响应码

\*/

private int code;

/\*\*

\* @description 响应消息

\*/

private String message;

/\*\*

\* @description 分页对象 （如果用不到，这个可以不写）

\*/

private PageVO page;

/\*\*

\* @description 数据

\*/

private Object data;

}

public enum BaseResponseCodeEnum {

// 系统通用

SUCCESS(200, "操作成功"),

private String code;

private String message;

private String subcode;

private String submessage;

ResponseCodeEnum(String code, String message) {

this.code = code;

this.message = message;

}

ResponseCodeEnum(String code, String message,String subCode,String subMessage) {

this.code = code;

this.message = message;

}

public final Integer getCode() {

return this.code;

}

public final String getMessage() {

return this.message;

}

}

### 资源正确释放与关闭

使用try…finally结构保证发生异常时，资源也能正确释放、关闭。

## JSF异常处理模板参考



### 统一响应格式(使用RpcResult类)

@Getter

@Setter

@ToString

@Builder

public class RpcResult<T> {

private String code;

private String message;

private T data;

}

### 业务错误码统一继承BaseErrorCodeEnum.

public interface IErrorCode {

public String getErrorCode();

public String getErrorMessage();

}

public enum BaseErrorCodeEnum implements IErrorCode {

//\*\*\*\*\*\*\*\*\*\* 成功 \*\*\*\*\*\*\*\*\*\*\*\*

CODE\_200("200", "业务系统调用成功！"),

CODE\_201("201", "交易成功！"),

//\*\*\*\*\*\*\*\*\*\* 异常 \*\*\*\*\*\*\*\*\*\*\*\*

CODE\_500("500", "系统异常！"),

CODE\_501("501", "参数处理异常！"),

CODE\_502("502", "业务系统调用异常！")

private BaseErrorCodeEnum (String code, String message) {

this.code = code;

this.message = message;

}

}

### 统一的基础异常类RpcException

public class RpcException extends RuntimeException {

private static final long serialVersionUID = 1L;

private IErrorCode iErrorCode;

private String code;

private String message;

public RpcException() {

super();

}

public RpcException(IErrorCode iErrorCode) {

super();

this.iErrorCode = iErrorCode;

this.code = iErrorCode.getCode();

this.message = iErrorCode.getMessage();

}

public RpcException(String message, Throwable cause) {

super(message, cause);

}

public RpcException(String message) {

super(message);

}

public RpcException(Throwable cause) {

super(cause);

}

public RpcException(int code) {

super();

this.code = code;

}

public RpcException(int code, String message, Throwable cause) {

super(message, cause);

this.code = code;

}

public RpcException(int code, String message) {

super(message);

this.code = code;

}

public RpcException(int code, Throwable cause) {

super(cause);

this.code = code;

}

//其他get、set、构造方法

}

### 统一的异常处理

实现JsfExceptionHandler，业务类不需关注异常处理，只需处理业务逻辑；

@Slf4j  
public class JsfExceptionHandler extends AbstractFilter {

@Override  
 public ResponseMessage invoke(RequestMessage requestMessage) {

// 执行JSF 调用，获得响应消息  
 ResponseMessage responseMessage = getNext().invoke(requestMessage);

if (!responseMessage.isError()) {  
 return responseMessage;  
 }  
  
 Throwable t = responseMessage.getException();

If (ex == null) {

throw new IllegalArgumentException("ExceptionHandler.Parameter.BeNull");

}

if(ex instanceof RpcException) {

JsfResult jsfResult = new JsfResult();

jsfResult.setCode(ex.getCode());

jsfResult.setMessage(ex.getMessage());

}

}

}

### [JSF的参数校验（Validation）功能](http://jpcloud.jd.com/pages/viewpage.action?pageId=10671618)

推荐优先开启Consumer参数校验功能，详细参考[JSF的参数校验（Validation）功能](http://jpcloud.jd.com/pages/viewpage.action?pageId=10671618)：http://jpcloud.jd.com/pages/viewpage.action?pageId=10671618

### 其他原则

对于公司外的http/api开放接口必须使用“错误码”

RpcException 是对外的唯一异常类型，所有内部异常，如果向外抛出，必须转为 RpcException。

RpcException 所有code信息可用 BaseErrorCodeEnum 标识，以便保持兼容。