Java 异常处理

\*【强制】Java 类库中定义的可以通过预检查方式规避的 RuntimeException 异常不应该通过 catch 的方式来处理，比如:NullPointerException，IndexOutOfBoundsException 等等。 说明:无法通过预检查的异常除外，比如，在解析字符串形式的数字时，不得不通过 catch NumberFormatException 来实现。

\*正例:if (obj != null) {...}

\*反例:try { obj.method(); } catch (NullPointerException e) {...}

\*【强制】异常不要用来做流程控制，条件控制。 说明:异常设计的初衷是解决程序运行中的各种意外情况，且异常的处理效率比条件判断方式 要低很多。

\*【强制】catch 时请分清稳定代码和非稳定代码，稳定代码指的是无论如何不会出错的代码。 对于非稳定代码的 catch 尽可能进行区分异常类型，再做对应的异常处理。 说明:对大段代码进行 try-catch，使程序无法根据不同的异常做出正确的应激反应，也不利 于定位问题，这是一种不负责任的表现。 正例:用户注册的场景中，如果用户输入非法字符，或用户名称已存在，或用户输入密码过于 简单，在程序上作出分门别类的判断，并提示给用户。

\*【强制】捕获异常是为了处理它，不要捕获了却什么都不处理而抛弃之，如果不想处理它，请 将该异常抛给它的调用者。最外层的业务使用者，必须处理异常，将其转化为用户可以理解的 内容。

\*【强制】有 try 块放到了事务代码中，catch 异常后，如果需要回滚事务，一定要注意手动回 滚事务。

\*【强制】finally 块必须对资源对象、流对象进行关闭，有异常也要做 try-catch。 说明:如果 JDK7 及以上，可以使用 try-with-resources 方式。

\*【强制】不要在 finally 块中使用 return。

\*说明:finally 块中的 return 返回后方法结束执行，不会再执行 try 块中的 return 语句。

\*【强制】捕获异常与抛异常，必须是完全匹配，或者捕获异常是抛异常的父类。 说明:如果预期对方抛的是绣球，实际接到的是铅球，就会产生意外情况。

\*【推荐】方法的返回值可以为 null，不强制返回空集合，或者空对象等，必须添加注释充分 说明什么情况下会返回 null 值。

\*说明:本手册明确防止 NPE 是调用者的责任。即使被调用方法返回空集合或者空对象，对调用者来说，也并非高枕无忧，必须考虑到远程调用失败、序列化失败、运行时异常等场景返回 null 的情况。

\*【推荐】防止 NPE，是程序员的基本修养，注意 NPE 产生的场景:

\*1) 返回类型为基本数据类型，return 包装数据类型的对象时，自动拆箱有可能产生 NPE。

\* 反例:public int f() { return Integer 对象}， 如果为 null，自动解箱抛 NPE。

\*2) 数据库的查询结果可能为null。

\*3) 集合里的元素即使isNotEmpty，取出的数据元素也可能为null。

\*4) 远程调用返回对象时，一律要求进行空指针判断，防止NPE。

\*5) 对于Session中获取的数据，建议NPE检查，避免空指针。

\*6) 级联调用obj.getA().getB().getC();一连串调用，易产生NPE。

\* 正例:使用 JDK8 的 Optional 类来防止 NPE 问题。

\*【推荐】定义时区分unchecked/checked 异常，避免直接抛出newRuntimeException()， 更不允许抛出 Exception 或者 Throwable，应使用有业务含义的自定义异常。推荐业界已定义 过的自定义异常，如:DAOException / ServiceException等。

\*【参考】对于公司外的 http/api 开放接口必须使用“错误码”;而应用内部推荐异常抛出; 跨应用间 RPC 调用优先考虑使用 Result 方式，封装 isSuccess()方法、“错误码”、“错误简 短信息”。

\* 说明:关于 RPC 方法返回方式使用 Result 方式的理由:

\*1) 使用抛异常返回方式，调用方如果没有捕获到就会产生运行时错误。

\*2) 如果不加栈信息，只是new自定义异常，加入自己的理解的error message，对于调用 端解决问题的帮助不会太多。如果加了栈信息，在频繁调用出错的情况下，数据序列化和传输 的性能损耗也是问题。

\*【参考】避免出现重复的代码(Don’t Repeat Yourself)，即DRY原则。

\* 说明:随意复制和粘贴代码，必然会导致代码的重复，在以后需要修改时，需要修改所有的副本，容易遗漏。必要时抽取共性方法，或者抽象公共类，甚至是组件化。

\* 正例:一个类中有多个 public 方法，都需要进行数行相同的参数校验操作，这个时候请抽取: private boolean checkParam(DTO dto) {...}

\*

# 最佳实践

\*

## 1. 在Finally块中清理资源或者使用try-with-resource语句

\*当使用类似InputStream这种需要使用后关闭的资源时，一个常见的错误就是在try块的最后关闭资源。

public void doNotCloseResourceInTry() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

// do NOT do this

inputStream.close();

} catch (FileNotFoundException e) {

log.error(e);

} catch (IOException e) {

log.error(e);

}

}

\*上述代码在没有任何exception的时候运行是没有问题的。但是当try块中的语句抛出异常或者自己实现的代码抛出异常，那么就不会执行最后的关闭语句，从而资源也无法释放。

\*合理的做法则是将所有清理的代码都放到finally块中或者使用try-with-resource语句。

public void closeResourceInFinally() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

log.error(e);

}

}

}

}

public void automaticallyCloseResource() {

File file = new File("./tmp.txt");

try (FileInputStream inputStream = new FileInputStream(file);) {

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} catch (IOException e) {

log.error(e);

}

}

## 2. 指定具体的异常

\*尽可能的使用最具体的异常来声明方法，这样才能使得代码更容易理解。

public void doNotDoThis() throws Exception {

...

}

public void doThis() throws NumberFormatException {

...

}

\*如上，NumberFormatException字面上即可以看出是数字格式化错误。

## 3. 对异常进行文档说明

\*当在方法上声明抛出异常时，也需要进行文档说明。和前面的一点一样，都是为了给调用者提供尽可能多的信息，从而可以更好地避免/处理异常。

\*在Javadoc中加入throws声明，并且描述抛出异常的场景。

/\*\*

\* This method does something extremely useful ...

\*

\* @param input

\* @throws MyBusinessException if ... happens

\*/

public void doSomething(String input) throws MyBusinessException {

...

}

## 4. 抛出异常的时候包含描述信息

\*在抛出异常时，需要尽可能精确地描述问题和相关信息，这样无论是打印到日志中还是监控工具中，都能够更容易被人阅读，从而可以更好地定位具体错误信息、错误的严重程度等。

\*但这里并不是说要对错误信息长篇大论，因为本来Exception的类名就能够反映错误的原因，因此只需要用一到两句话描述即可。

try {

new Long("xyz");

} catch (NumberFormatException e) {

log.error(e);

}

\*NumberFormatException即告诉了这个异常是格式化错误，异常的额外信息只需要提供这个错误字符串即可。当异常的名称不够明显的时候，则需要提供尽可能具体的错误信息。

## 5. 首先捕获最具体的异常

\*现在很多IDE都能智能提示这个最佳实践，当你试图首先捕获最笼统的异常时，会提示*不能达到的代码*。

\*当有多个catch块中，按照捕获顺序只有第一个匹配到的catch块才能执行。因此，如果先捕获IllegalArgumentException，那么则无法运行到对NumberFormatException的捕获。

public void catchMostSpecificExceptionFirst() {

try {

doSomething("A message");

} catch (NumberFormatException e) {

log.error(e);

} catch (IllegalArgumentException e) {

log.error(e)

}

}

## 6. 不要捕获Throwable

\*Throwable是所有异常和错误的父类。你可以在catch语句中捕获，但是永远不要这么做。

\*如果catch了throwable，那么不仅仅会捕获所有exception，还会捕获error。而error是表明无法恢复的jvm错误。因此除非绝对肯定能够处理或者被要求处理error，不要捕获throwable。

public void doNotCatchThrowable() {

try {

// do something

} catch (Throwable t) {

// don't do this!

}

}

## 7. 不要忽略异常

\*很多时候，开发者很有自信不会抛出异常，因此写了一个catch块，但是没有做任何处理或者记录日志。

public void doNotIgnoreExceptions() {

try {

// do something

} catch (NumberFormatException e) {

// this will never happen

}

}

\*但现实是经常会出现无法预料的异常或者无法确定这里的代码未来是不是会改动(删除了阻止异常抛出的代码)，而此时由于异常被捕获，使得无法拿到足够的错误信息来定位问题。

\*合理的做法是至少要记录异常的信息。

public void logAnException() {

try {

// do something

} catch (NumberFormatException e) {

log.error("This should never happen: " + e);

}

}

## 8. 不要记录并抛出异常

\*可以发现很多代码甚至类库中都会有捕获异常、记录日志并再次抛出的逻辑。如下：

try {

new Long("xyz");

} catch (NumberFormatException e) {

log.error(e);

throw e;

}

\*这个处理逻辑看着是合理的。但这经常会给同一个异常输出多条日志。如下：

17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatException: For input string: "xyz"

Exception in thread "main" java.lang.NumberFormatException: For input string: "xyz"

at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

at java.lang.Long.parseLong(Long.java:589)

at java.lang.Long.(Long.java:965)

at com.stackify.example.TestExceptionHandling.logAndThrowException(TestExceptionHandling.java:63)

at com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:58)

\*如上所示，后面的日志也没有附加更有用的信息。如果想要提供更加有用的信息，那么可以将异常包装为自定义异常。

public void wrapException(String input) throws MyBusinessException {

try {

// do something

} catch (NumberFormatException e) {

throw new MyBusinessException("A message that describes the error.", e);

}

}

\*因此，仅仅当想要处理异常时才去捕获，否则只需要在方法签名中声明让调用者去处理。

## 9. 包装异常时不要抛弃原始的异常

\*捕获标准异常并包装为自定义异常是一个很常见的做法。这样可以添加更为具体的异常信息并能够做针对的异常处理。

\*需要注意的是，包装异常时，一定要把原始的异常设置为cause(Exception有构造方法可以传入cause)。否则，丢失了原始的异常信息会让错误的分析变得困难。

public void wrapException(String input) throws MyBusinessException {

try {

// do something

} catch (NumberFormatException e) {

throw new MyBusinessException("A message that describes the error.", e);

}

}