## 深入了解 Java 中的异常处理 + 面试题

在程序开发中，异常处理也是我们经常使用到的模块，只是平常很少去深究异常模块的一些知识点。比如，try-catch 处理要遵循的原则是什么，finally 为什么总是能执行，try-catch 为什么比较消耗程序的执行性能等问题，我们本讲内容都会给出相应的答案，当然还有面试中经常被问到的异常模块的一些面试题，也是我们本篇要讲解的重点内容。

### 异常处理基础介绍

先来看看 **异常处理的语法格式** ：

try{ … } catch(Exception e){ … } finally{ … }

其中，

* **try** ：是用来监测可能会出现异常的代码段。
* **catch** ：是用来捕获 try 代码块中某些代码引发的异常，如果 try 里面没有异常发生，那么 catch 也一定不会执行。在 Java 语言中，try 后面可以有多个 catch 代码块，用来捕获不同类型的异常，需要注意的是前面的 catch 捕获异常类型一定不能包含后面的异常类型，这样的话，编译器会报错。
* **finally** ：不论 try-catch 如何执行，finally 一定是最后执行的代码块，所有通常用来处理一些资源的释放，比如关闭数据库连接、关闭打开的系统资源等。

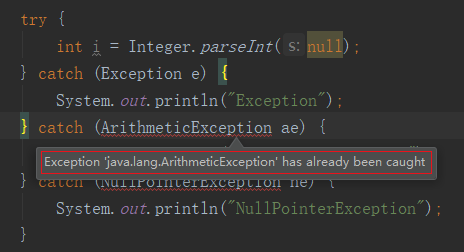
**异常处理的基本使用** ，具体可以参考下面的代码段：

try {  
 int i = 10 / 0;  
} catch (ArithmeticException e) {  
 System.out.println(e);  
} finally {  
 System.out.println("finally");  
}

**多 catch 的使用** ，具体可以参考下面的代码段：

try {  
 int i = Integer.parseInt(null);  
} catch (ArithmeticException ae) {  
 System.out.println("ArithmeticException");  
} catch (NullPointerException ne) {  
 System.out.println("NullPointerException");  
} catch (Exception e) {  
 System.out.println("Exception");  
}

需要注意的是 Java 虚拟机会从上往下匹配错误类型，因此前面的 catch 异常类型不能包含后面的异常类型。比如上面的代码如果把 Exception 放在最前面编译器就会报错，具体可以参考下面的图片。



enter image description here

### 异常处理的发展

随着 Java 语言的发展，JDK 7 的时候引入了一些更加便利的特性，用来更方便的处理异常信息，如 try-with-resources 和 multiple catch，具体可以参考下面的代码段：

try (FileReader fileReader = new FileReader("");  
 FileWriter fileWriter = new FileWriter("")) { // try-with-resources  
 System.out.println("try");  
} catch (IOException | NullPointerException e) { // multiple catch  
 System.out.println(e);  
}

### 异常处理的基本原则

先来看下面这段代码，有没有发现一些问题？

try {  
 // ...  
 int i = Integer.parseInt(null);  
} catch (Exception e) {  
}

以上的这段代码，看似“正常”，却违背了异常处理的两个基本原则：

* 第一，尽量不要捕获通用异常，也就是像 Exception 这样的异常，而是应该捕获特定异常，这样更有助于你发现问题；
* 第二，不要忽略异常，像上面的这段代码只是加了 catch，但没有进行如何的错误处理，信息就已经输出了，这样在程序出现问题的时候，根本找不到问题出现的原因，因此要切记不要直接忽略异常。

### 异常处理对程序性能的影响

异常处理固然好用，但一定不要滥用，比如下面的代码片段：

// 使用 com.alibaba.fastjson  
JSONArray array = new JSONArray();  
String jsonStr = "{'name':'laowang'}";  
try {  
 array = JSONArray.parseArray(jsonStr);  
} catch (Exception e) {  
 array.add(JSONObject.parse(jsonStr));  
}  
System.out.println(array.size());

这段代码是借助了 try-catch 去处理程序的业务逻辑，通常是不可取的，原因包括下列两个方面。

* try-catch 代码段会产生额外的性能开销，或者换个角度说，它往往会影响 JVM 对代码进行优化，因此建议仅捕获有必要的代码段，尽量不要一个大的 try 包住整段的代码；与此同时，利用异常控制代码流程，也不是一个好主意，远比我们通常意义上的条件语句（if/else、switch）要低效。
* Java 每实例化一个 Exception，都会对当时的栈进行快照，这是一个相对比较重的操作。如果发生的非常频繁，这个开销可就不能被忽略了。

以上使用 try-catch 处理业务的代码，可以修改为下列代码：

// 使用 com.alibaba.fastjson  
JSONArray array = new JSONArray();  
String jsonStr = "{'name':'laowang'}";  
if (null != jsonStr && !jsonStr.equals("")) {  
 String firstChar = jsonStr.substring(0, 1);  
 if (firstChar.equals("{")) {  
 array.add(JSONObject.parse(jsonStr));  
 } else if (firstChar.equals("[")) {  
 array = JSONArray.parseArray(jsonStr);  
 }  
}  
System.out.println(array.size());

### 相关面试题

#### 1. try 可以单独使用吗？

答：try 不能单独使用，否则就失去了 try 的意义和价值。

#### 2. 以下 try-catch 可以正常运行吗？

try {  
 int i = 10 / 0;  
} catch {  
 System.out.println("last");  
}

答：不能正常运行，catch 后必须包含异常信息，如 catch (Exception e)。

#### 3. 以下 try-finally 可以正常运行吗？

try {  
 int i = 10 / 0;  
} finally {  
 System.out.println("last");  
}

答：可以正常运行。

#### 4. 以下代码 catch 里也发生了异常，程序会怎么执行？

try {  
 int i = 10 / 0;  
 System.out.println("try");  
} catch (Exception e) {  
 int j = 2 / 0;  
 System.out.println("catch");  
} finally {  
 System.out.println("finally");  
}  
System.out.println("main");

答：程序会打印出 finally 之后抛出异常并终止运行。

#### 5. 以下代码 finally 里也发生了异常，程序会怎么运行？

try {  
 System.out.println("try");  
} catch (Exception e) {  
 System.out.println("catch");  
} finally {  
 int k = 3 / 0;  
 System.out.println("finally");  
}  
System.out.println("main");

答：程序在输出 try 之后抛出异常并终止运行，不会再执行 finally 异常之后的代码。

#### 6. 常见的运行时异常都有哪些？

答：常见的运行时异常如下：

* java.lang.NullPointerException 空指针异常；出现原因：调用了未经初始化的对象或者是不存在的对象；
* java.lang.ClassNotFoundException 指定的类找不到；出现原因：类的名称和路径加载错误，通常是程序

试图通过字符串来加载某个类时引发的异常；

* java.lang.NumberFormatException 字符串转换为数字异常；出现原因：字符型数据中包含非数字型字符；
* java.lang.IndexOutOfBoundsException 数组角标越界异常，常见于操作数组对象时发生；
* java.lang.ClassCastException 数据类型转换异常；
* java.lang.NoClassDefFoundException 未找到类定义错误；
* java.lang.NoSuchMethodException 方法不存在异常；
* java.lang.IllegalArgumentException 方法传递参数错误。

#### 7. Exception 和 Error 有什么区别？

答：Exception 和 Error 都属于 Throwable 的子类，在 Java 中只有 Throwable 及其之类才能被捕获或抛出，它们的区别如下：

* Exception（异常）是程序正常运行中，可以预期的意外情况，并且可以使用 try/catch 进行捕获处理的。Exception 又分为运行时异常（Runtime Exception）和受检查的异常（Checked Exception），运行时异常编译能通过，但如果运行过程中出现这类未处理的异常，程序会终止运行；而受检查的异常，要么用 try/catch 捕获，要么用 throws 字句声明抛出，否则编译不会通过。
* Error（错误）是指突发的非正常情况，通常是不可以恢复的，比如 Java 虚拟机内存溢出，诸如此类的问题叫做 Error。

#### 8. throw 和 throws 的区别是什么？

答：它们的区别如下：

* throw 语句用在方法体内，表示抛出异常由方法体内的语句处理，执行 throw 一定是抛出了某种异常；
* throws 语句用在方法声明的后面，该方法的调用者要对异常进行处理，throws 代表可能会出现某种异常，并不一定会发生这种异常。

#### 9. Integer.parseInt(null) 和 Double.parseDouble(null) 抛出的异常一样吗？为什么？

答：Integer.parseInt(null) 和 Double.parseDouble(null) 抛出的异常类型不一样，如下所示：

* Integer.parseInt(null) 抛出的异常是 NumberFormatException；
* Double.parseDouble(null) 抛出的异常是 NullPointerException。

至于为什么会产生不同的异常，其实没有特殊的原因，主要是由于这两个功能是不同人开发的，因而就产生了两种不同的异常信息。

#### 10. NoClassDefFoundError 和 ClassNoFoundException 有什么区别？

* NoClassDefFoundError 是 Error（错误）类型，而 ClassNoFoundExcept 是 Exception（异常）类型；
* ClassNoFoundExcept 是 Java 使用 Class.forName 方法动态加载类，没有加载到，就会抛出 ClassNoFoundExcept 异常；
* NoClassDefFoundError 是 Java 虚拟机或者 ClassLoader 尝试加载类的时候却找不到类订阅导致的，也就是说要查找的类在编译的时候是存在的，运行的时候却找不到，这个时候就会出现 NoClassDefFoundError 的错误。

#### 11. 使用 try-catch 为什么比较耗费性能？

答：这个问题要从 JVM（Java 虚拟机）层面找答案了。首先 Java 虚拟机在构造异常实例的时候需要生成该异常的栈轨迹，这个操作会逐一访问当前线程的栈帧，并且记录下各种调试信息，包括栈帧所指向方法的名字，方法所在的类名、文件名，以及在代码中的第几行触发该异常等信息，这就是使用异常捕获耗时的主要原因了。

#### 12. 常见的 OOM 原因有哪些？

答：常见的 OOM 原因有以下几个：

* 数据库资源没有关闭；
* 加载特别大的图片；
* 递归次数过多，并一直操作未释放的变量。

#### 13. 以下程序的返回结果是？

public static int getNumber() {  
 try {  
 int number = 0 / 1;  
 return 2;  
 } finally {  
 return 3;  
 }  
}

A：0

B：2

C：3

D：1

答：3

题目解析：程序最后一定会执行 finally 里的代码，会把之前的结果覆盖为 3。

#### 14. finally、finalize 的区别是什么？

答：finally、finalize 的区别如下：

* finally 是异常处理语句的一部分，表示总是执行；
* finalize 是 Object 类的一个方法，子类可以覆盖该方法以实现资源清理工作，垃圾回收之前会调用此方法。

#### 15. 为什么 finally 总能被执行？

答：finally 总会被执行，都是编译器的作用，因为编译器在编译 Java 代码时，会复制 finally 代码块的内容，然后分别放在 try-catch 代码块所有的正常执行路径及异常执行路径的出口中，这样 finally 才会不管发生什么情况都会执行。

更多java学习资料如下：

[Java高级架构师课程(总共122门课程，1460GB)](https://www.consultdog.com/index.php/front/44/)

[最强java面试视频课程(21门课程，126GB)](https://www.consultdog.com/index.php/front/78/)