**Error：程序中无法处理的错误（JVM）**

|  |  |  |  |
| --- | --- | --- | --- |
| 虚拟机运行错误 | Virtual MachineError | 类定义错误 | NoClassDefFoundError |
| 内存不足 | OutOfMemoryError | 栈溢出错误 | StackOverflowError |

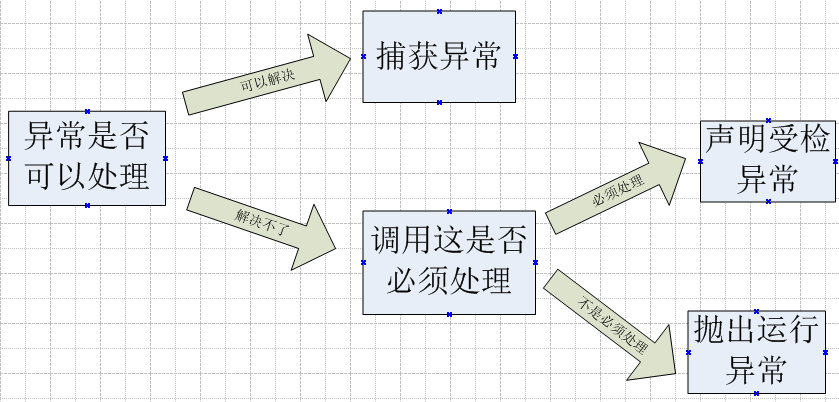
这些错误是不受检异常，非代码性错误

**Exception：运行时异常和编译异常**

|  |  |  |  |
| --- | --- | --- | --- |
| 运行时异常（非受异常） | RuntimeException | 如果没有通过throws声明抛出也没有用try catch捕获则会编译通过。但在运行时Java虚拟机自动抛出自动捕获 | NullPointPointException空指针异常  ArrayIndexOutBoundException数组下标越界异常  ClassCastException类型转换异常  ArithmeticException算数异常  ArrayStoreException数组存储异常  BufferOverflowException IO异常 |
| 编译时异常（受检异常） | 处运行时异常外其他都是 | 编译器会检查 ，只有用throws抛出或者try catch捕获才能编译通过 | ClassNotFoundException 没有找到指定的类  IOException IO流异常 |

**NoClassDefFoundError：**引起该异常的原因是JVM或ClassLoader尝试加载某类是在内存中找不到该类的定义，该动作发生在运行期间，即编译时该类存在，但在运行时却找不到了。

**ClassNotFoundException：**(1)当使用Class.forName，ClassLoader.loadClass或ClassLoader.findSystemClass动态加载类到内存的时候，通过传入的类路径参数没有找到该类，就会抛出异常；(2)某个类已经由一个类加载器加载到内存中，另一个加载器又尝试去加载它。



**finally：**主要用于回收在try块里打开的物力资源（如数据库连接、网络连接和磁盘文件）

private static void readFile(String filePath)throws MyException{

File file=new File(filePath);

String result;

BufferedReader reader=null;

try{

reader=new BufferedReader(new FileReader(file));

while((result=reader.readLine())!=null){

System.out.println(result);

}

}catch(IOException e){

System.out.println(“readFile method catch block”);

MyException ex=new MyException(“read file failed”);

ex.initCause(E);

throw ex;

}finally{

Ssytem.out.println(“readFile method finally block”);

if(null !=reader){

try{

reader.close();

}catch(IOException e){

e.printStackTrace(); //覆盖了原始异常

}

}

}

}

jdk1.7之后自动释放资源使用**AutoCloseable接口的类**

private static void tryWithResourceTest(){

try(Scanner sc=new Scanner(new FileInputStream(“c:/abc”),”UTF-8”){

//code  
}catch(IOException e){

//handle exception

}

}

try代码块退出时，会自动调用scanner.close()和把该方法放在finally代码块不同的是，若该方法抛出异常，则会被抑制，抛出的仍然为原始异常,被抑制的异常会由addSusppressed方法添加到原来的异常，如果想要获取被抑制的异常列表，可以调用getSuppressed()

\***执行完finally块才会回来执行try或者ctach块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句则不会跳回执行，直接停止。finally中的return会覆盖前面的return**

|  |  |  |
| --- | --- | --- |
| public static int getInt(){  int a=10;  try{  System.out.println(a/0);  a=20;  }catch(ArithmeticException e){  a=30;  return a; //return 30记录下返回值待//finally执行完后再返回a就成了常量30  }finally{  a=40;  }  return a; //a=40  }  执行结果：30 | Public static int getInt(){  Int a=10;  try{  System.out.println(a/0);  a=20;  }catch(ArithmeticException e){  a=30;  return a;  }finally{  a=40;  return a;//只能通过一个return返回  }  }  执行结果：40 | public static int val() {  int num = 5;  try {  num = num / 0;  } catch (Exception e) {  num = 10;  } finally {  num = 15;  }  return num;  }  执行结果：15 |

情况1： try{} catch(){}finally{} return;

显然程序按顺序执行。

情况2: try{ return; }catch(){} finally{} return;

先执行try块中return 语句（包括return语句中的表达式运算），但不返回；

执行finally语句中全部代码，最后执行try中return 返回

finally块之后的语句return不执行，因为程序在try中已经return。

情况3: try{ } catch(){return;} finally{} return;

1、程序先执行try，如果遇到异常执行catch块，

有异常：执行catch中return语句，但不返回，执行finally语句中全部代码，最后执行catch块中return返回。 finally块后的return语句不再执行。

无异常：执行完try再finally再return…

情况4: try{ return; }catch(){} finally{return;}

执行try块return语句（包括return语句中的表达式运算），但不返回；

再执行finally块，

执行finally块，有return，从这里返回。

此时finally块的return值，就是代码执行完后的值

情况5: try{} catch(){return;}finally{return;}

程序执行catch块中return语句（包括return语句中的表达式运算），但不返回；

再执行finally块，

执行finally块，有return，从这里返回。

情况6: try{ return;}catch(){return;} finally{return;}

1、程序执行try块中return语句（包括return语句中的表达式运算），但不返回；

有异常：

执行catch块中return语句（包括return语句中的表达式运算），但不返回；

再执行finally块

执行finally块，有return，从这里返回。

无异常：

再执行finally块

执行finally块，有return，从这里返回。。

-------------------------------------------------------------------------------------------------------

原文链接https://blog.csdn.net/xx326664162/article/details/50266995

发表于 2019-09-18 10:22:15

回复(4)

**1.JVM是如何处理异常的？**

在一个方法中如果发生异常，这个方法会创建一个异常对象，并转交给JVM，该异常对象包含异常名称，异常描述以及异常发生时应用程序的状态。创建异常对象并转交给JVM的过程称为抛出异常。可能有一系列的方法调用，最终才能进入抛出异常的方法，这一系列方法调用的有序列表叫调用栈。

JVM会顺着调用栈去查找看是否有可以处理异常的代码，有则调用异常处理代码。当JVM发现可以处理异常的代码时，会把发生的异常传递给它。如果JVM没有找到可以处理该异常的代码块，JVM就会将该异常转交给默认的异常处理器，并打印出异常信息并终止应用程序

**2.try-catch-finally**

try只适合处理运行时异常，try+catch适合处理运行时异常+普通异常。也就是说，如果只用try去处理普通异常编译时通不过的，因为编译器硬性规定，普通异常如果选择捕获则必须用catch显示声明以便进一步处理。而运行时异常在编译时没有如此硬性规定。

**3.java常见异常**

IIIegalAccessError:违法访问错误，当一个应用试图访问、修改某个类的域（Field）或者调用其方法，但是又违反域或方法的可见性声明，则会抛出此异常。

InstantiationError:实例化错误。用new实例化抽象类或接口时

OutOfMmeoryError：内存不足，内存不足以让JVM分配一个对象

StackOverflowError:堆栈溢出。当一个应用递归调用的层次太深或者陷入死循环

ClassCastException:强制类型转换异常

ClassNotFoundException：找不到类异常。

InstantiationException：实例化异常，同行newInstance方法创建某个抽象类或接口时。

NoSuchFieldException：属性不存在异常

NoSuchMethodException：方法不存在异常

NullPointerException：空指针异常，要求使用对象的地方使用了null，调用null对象的实例化方法、访问空对象的属性、计算空对象的长度、使用throw语句抛出null

NumberFormatException：数字格式异常。当试图吧一个String转换为制定的数字格式而该字符串却不满足数字类型要求的格式。

|  |
| --- |
| 不论什么异常，只会抛出一个。一个 try 块可能有多个 catch 块。若如此，则执行第一个匹配块。即Java虚拟机会把实际抛出的异常对象依次和各个catch代码块声明的异常类型匹配，如果异常对象为某个异常类型或 其子类的实例，就执行这个catch代码块，不会再执行其他的 catch代码块。除了下列情况，总将执行 finally 做为结束： JVM 过早终止（调用 System.exit(int)）；在 finally 块中抛出一个未处理的异常；计算机断电、失火、或遭遇病毒攻击 |
| public void getCustomerInfo() {  try {  // do something that may cause an Exception  } catch (java.io.FileNotFoundException ex) {  System.out.print("FileNotFoundException!");  } catch (java.io.IOException ex) {  System.out.print("IOException!");  } catch (java.lang.Exception ex) {  System.out.print("Exception!");  }  } |

**OutOfMemoryError**

A：属于运行时常量池导致的溢出，设置-XX：MaxPermSize可以解决这个问题，

B：Requested array size exceeds VM limit当你正准备创建一个超过虚拟机允许的大小的数组时，这条错误将会出现属于堆空间不足导致的错误，问题比较少见，解决方式和C相同，

C：Java heap space属于java堆内存问题，一般的手段是通过内存映像分析工具，对Dump出来的堆转储存快照进行分析，重点是确认内存中的对象是否是有必要的，也就是要判断是出现了内存泄漏，还是出现了内存溢出，如果是内存列楼，通过工具检查泄露对象打GC Roots的引用链信息，可以准确的确定出泄露代码的位置，不存在泄露，就应该检查虚拟机的堆参数，如果可以继续调大，可以设置-Xmx解决问题

D：java.lang.OutOfMemoryError: nativeGetNewTLA指当虚拟机不能分配新的线程本地空间(Thread Local Area）的时候错误信息，此错误是线程申请一个新的TLA时产生的，这个异常一般只会发生在jRockit虚拟机，只有过于绝对