第5章 异　常

正常(normal)的程序沿着控制流执行。在执行过程中可能会发生某事件导致程序无法继续沿着控制流执行,这种不期望发生的事件称为异常(exception)。异常是“异常事件”(exceptional event)的简称。比如使用索引 3 访问长度为 3 的数组就会抛出“数组越界”异常。

5.1 异常的抛出与捕获

程序把异常事件创建为“异常对象”并传递给 Java 虚拟机,这个过程称为“抛出”(throw)。异常对象中包含了错误信息以及程序在异常出现时的状态。异常是“少而不同”的事件,并不总会发生。如果程序需要对可能发生的异常事件按预案进行处理,那么程序就设计一段代码首先从虚拟机中取出异常对象,这个过程称为“捕获”(catch),然后再进行处理。图 5-1 展示了异常从正常流程语句中抛出到虚拟机,用于处理异常的程序块从虚拟机中捕获异常再处理。从示意图可以看出,Java 的异常机制把程序的正常流程和异常处理流程分离,提高了源代码的可读性。

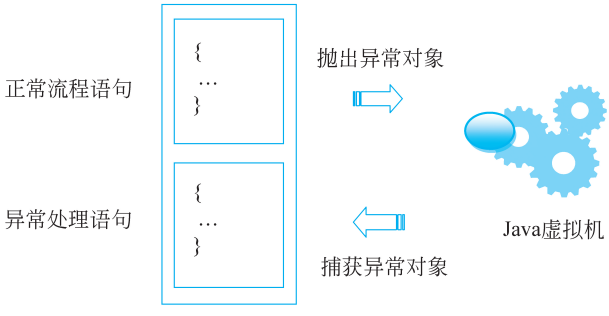

图 5-1　异常的抛出与捕获

数组越界异常 ArrayIndexOutOfBoundsException 是在 java.lang 包中定义的类。源文件 5-1 演示了数组越界异常的产生。在第 3 行首先创建了一个只具有三个元素的数组对象，即数组的长度为 3，按照 Java 语言对数组的访问规则，应分别通过 a[0]、a[1]和 a[2]访问这三个数组对象。然而第 4 行的语句却试图访问 a[3]。当程序执行到第 4 行的语句时，println 方法就会向虚拟机抛出 ArrayIndexOutOfBoundsException 类型的异常对象，并在控制台输出运行时刻错误消息：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3
out of bounds for length 3
at ArrayIndexOutOfBoundsExceptionDemo.main
(ArrayIndexOutOfBoundsExceptionDemo.java:4)
```

意思是在源文件 ArrayIndexOutOfBoundsExceptionDemo.java 第 4 行 main 方法中出现 java.lang.ArrayIndexOutOfBoundsException 类型的异常：索引 3 超出了长度为 3 的数组的索引范围。

由于程序没有设计任何预案来应对该异常，程序终止执行。

源文件 5-1　ArrayIndexOutOfBoundsExceptionDemo.java

```
1 public class ArrayIndexOutOfBoundsExceptionDemo {
2     public static void main(String[] args) {
3         int[] a = {10, 20, 30};
4         System.out.println(a[3]);
5     }
6 }
```

除了数组越界异常，常见异常还有 NullPointerException，如源文件 5-2 第 4 行试图访问静态变量 a；而 a 没有引用任何数组对象，其初值为 null。所以当程序运行到第 4 行会抛出空指针异常，并在控制台输出如下消息后终止执行：

```
Exception in thread "main" java.lang.NullPointerException: Cannot load from int
array because "NullPointerExceptionDemo.a" is null
    at NullPointerExceptionDemo.main(NullPointerExceptionDemo.java:4)
```

这段消息说，在源文件 NullPointerExceptionDemo.java 第 4 行出现 java.lang.NullPointerException 类型的异常：因为 a 的值是 null，所以无法从整型数组读取。

源文件 5-2　NullPointerExceptionDemo.java

```
1 public class NullPointerExceptionDemo {
2     static int[] a;
3     public static void main(String[] args) {
4         System.out.println(a[0]);
5     }
6 }
```

java.lang.NullPointerException 的父类是运行时刻异常 java.lang.RuntimeException；java.lang.RuntimeException 的父类是 java.lang.Exception；java.lang.Exception 的父类是 java.lang.Throwable。RuntimeException 是 ArrayIndexOutOfBoundsException 的祖先类。JDK 中的异常继承关系如图 5-2 所示。

从图 5-2 中看到，异常是可抛出的(Throwable)对象。运行时刻异常 RuntimeException 也

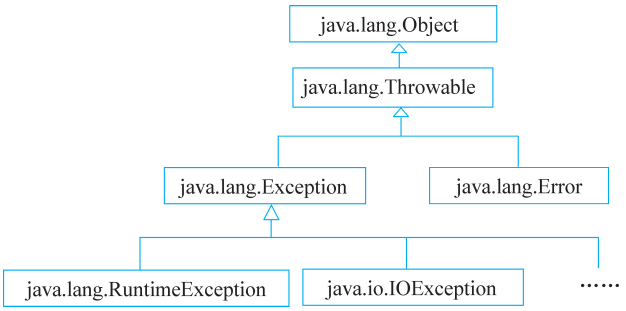

图 5-2　JDK 中的异常

称为免检(unchecked)异常,而 Exception 类的其他子类都是受检(checked)异常。所谓“免检”,指这类异常导致程序无法继续执行下去,只能终止,不必为这类异常设计处理程序;所谓“受检”,指程序必须安排预案,一旦异常出现如何处置。IOException 是典型的受检异常。FileNotFoundException 是一种 IOException。源文件 5-3 演示了编译器如何强制程序处理可能出现的 IOException 对象。

源文件 5-3　FileNotFoundExceptionDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 public class FileNotFoundExceptionDemo {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(new File("input.txt"));
6         System.out.println(sc.nextLine());
7         sc.close();
8     }
9 }
```

源文件 5-3 试图从文本文件 input.txt 中读取一行。第 5 行从当前路径中读取文本文件 input.txt,所以以文件名 input.txt 为实参创建 File 类型的对象以准备访问磁盘文件 input.txt,然后把 File 对象作为实参创建 Scanner 对象以从该文件中读写数据。第 5 行会出现编译错误:

```
FileNotFoundExceptionDemo.java:5: error:
unreported exception FileNotFoundException; must be caught or declared to
be thrown
```

意思是在源文件 FileNotFoundExceptionDemo.java 第 5 行有错误:找不到文件异常 FileNotFoundException 要么被捕获、要么声明被抛出。

这样,编译器强制在程序中必须对可能出现的 IOException 进行处理。处理的方法有两种:捕获和声明抛出。声明抛出是较为简单的处理方法:如果在方法体中可能抛出某类型的异常,那么只需把该异常类型在方法头部使用保留字 throws 声明抛出即可。该方法的调用者看到这个声明,再进行处理。如源文件 5-4 第 5 行所示。

源文件 5-4　FileNotFoundExceptionDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
```

```
4 public class FileNotFoundExceptionDemo {
5     public static void main(String[] args)
          throws FileNotFoundException {
6         Scanner sc = new Scanner(new File("input.txt"));
7         System.out.println(sc.nextLine());
8         sc.close();
9     }
10 }
```

File 和 FileNotFoundException 都是 java.io 包中的类，注意使用 import 保留字导入。

另外一种处理方式就是使用保留字 catch 捕获异常，然后在 catch 后面的块中处理该异常。源文件 5-5 使用 try-catch 捕获和处理受检异常 FileNotFoundException。

源文件 5-5　CatchDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4 public class CatchDemo {
5     public static void main(String[] args) {
6         Scanner sc = null;
7         try {
8             sc = new Scanner(new File("input.txt"));
9             System.out.println(sc.nextLine());
10            sc.close();
11         } catch (FileNotFoundException e) {
12            e.printStackTrace();
13            System.out.println(e.getMessage());
14         }
15     }
16 }
```

把可能抛出异常的程序片段放在保留字 try 后的一对花括号“{}”中。这对花括号括起来的语句序列称为 try 块(try block)。然后把对异常处理的程序片段放在保留字 catch 后面的花括号中，这对花括号“{}”括起来的语句序列称为 catch 块。一旦 try 块中的某条语句抛出异常对象，就不再继续执行该语句以及其后随语句，而是转移到 catch 块执行。

为了在 catch 块中引用 try 块中抛出的异常对象，在保留字 catch 其后的 catch 块之间使用圆括号“()”括起来了异常对象引用变量声明。执行 catch 块时，通过声明的异常对象引用变量(此例中是 e)可以访问异常对象的具体内容。第 12 行调用了实例方法 printStackTrace 输出异常对象的栈跟踪信息，这是给程序员看，用来调试程序的；第 13 行调用了实例方法 getMessage 输出异常提示消息，这是给最终用户看的。执行 catch 块完毕，继续执行 try-catch 语句的后随语句。程序的输出结果为：

```
java.io.FileNotFoundException: input.txt (系统找不到指定的文件。)
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:158)
    at java.base/java.util.Scanner.<init>(Scanner.java:645)
    at CatchDemo.main(CatchDemo.java:8)
input.txt (系统找不到指定的文件。)
```

最后一行是实例方法 getMessage 输出。图 5-3 展示 try 块、catch 块、异常对象和 JVM 之间的关系。

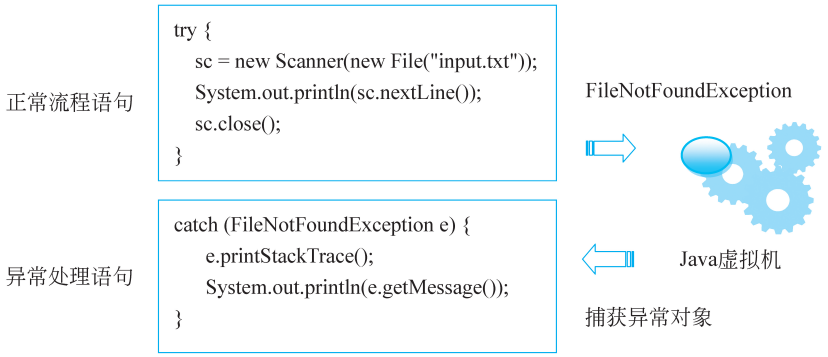

图 5-3 try 块和 catch 块

在某次运行中，try 块中的某条语句可能会抛出异常对象，此例中是 FileNotFoundException 类型的异常对象；这个异常对象交给了 JVM；程序的控制发生转移，转到 catch 块执行；catch 块通过声明的 FileNotFoundException 类型的引用变量 e 访问 try 块中抛出的异常对象，一旦 e 能够引用这个异常对象，则称捕捉到了该异常。

catch 块只能捕获 Exception 类及其子类的对象。但是 RuntimeException 及其后代类是免检异常，如 ArithmeticException、ClassCastException、IllegalArgumentException、IndexOutOfBoundsException、NegativeArraySizeException、NoSuchElementException、NullPointerException 等。除了 RuntimeException 以外，其他 Exception 类的后代类都是受检异常。对于所有受检异常，程序必须显式地声明如何进行处理。

5.2 处理异常

源文件 5-5 中的 try-catch 语句在没有任何异常出现的时候能够正确运行，但是如果 try 块中第 8 行的语句抛出异常，那么就不会执行第 10 行的关闭语句 close()，从而无法释放资源。合理的做法是将所有清理现场和释放资源的语句都放到 finally 块中或者使用 try-with-resource 语句。源文件 5-6 演示了 finally 块的用法。

源文件 5-6 FinallyDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class FinallyDemo {
6     public static void main(String[] args) {
7         Scanner sc = null;
8         try {
9             sc = new Scanner(new File("input.txt"));
10            System.out.println(sc.nextLine());
11        } catch (FileNotFoundException e) {
```

```
12              e.printStackTrace();
13              System.out.println(e.getMessage());
14          } finally {
15              if (sc != null) {
16                  sc.close();
17              }
18          }
19      }
20 }
```

在源文件 5-6 中，如果在 try 块中没有异常抛出，那么执行 finally 块，关闭 Scanner 对象 sc；如果在 try 块中抛出了异常，那么不再继续执行 try 块中抛出异常的语句及其后面的语句，转入 catch 块执行，再继续执行 finally 块，也会关闭 Scanner 对象 sc。所以，无论异常发生与否，finally 块都会执行。

try-catch 语句的一般形式是：

```
try{
   ……
}catch(Exception <引用变量名>){
   ……
}finally{
   ……
}
```

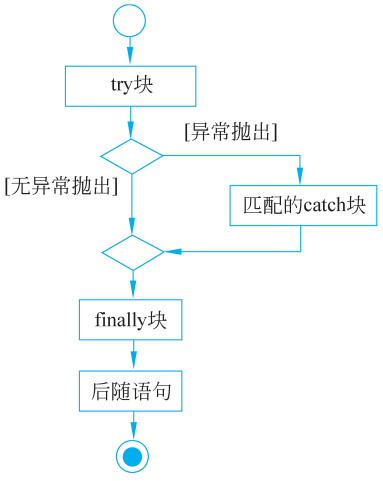

图 5-4　try-catch-finally 块

如果有 try 块，则至少与之关联一个 catch 块。另外，还可以至多与 try 块关联一个 finally 块。因为 try-catch 语句的执行有两种可能：异常不发生，仅执行 try 块；异常发生，还需执行 catch 块。当有一个 finally 与 try 块相关联时，在这两种情形下，都会执行 finally 块。也就是说，无论异常发生与否，finally 块总会执行。因此，finally 块是进行一些清除现场操作的理想位置。图 5-4 是展示 try-catch-finally 语句控制流程的 UML 活动图。

源文件 5-7 演示了 try-with-resource 的用法：把源文件 5-4 中第 6 行声明及创建 Scanner 对象的语句移到保留字 try 与花括号间的圆括号中。这样无须使用 finally 块。

源文件 5-7　WithResourceDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class WithResourceDemo {
6     public static void main(String[] args) {
7         try (Scanner sc = new Scanner(new File("input.txt"));) {
8             System.out.println(sc.nextLine());
9         } catch (FileNotFoundException e) {
```

```
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }
}
```

可以通过分号分隔声明的多个资源。例如源文件 5-8 试图把当前文件夹中的 input.txt 文件逐行复制到 output.txt 中。在第 8 行和第 9 行声明了 scanner 和 writer 引用两个变量分别引用 Scanner 对象和 PrintWriter 对象，这两个对象都会占用操作系统资源。

源文件 5-8　WithMultiResourceDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.PrintWriter;
4 import java.io.FileNotFoundException;
5
6 public class WithMultiResourceDemo {
7     public static void main(String[] args) {
8         try (Scanner scanner = new Scanner(new File("input.txt"));
9                PrintWriter writer = new PrintWriter(new File("output.txt"))) {
10            while (scanner.hasNext()) {
11                writer.print(scanner.nextLine());
12            }
13        } catch (FileNotFoundException e) {
14            e.printStackTrace();
15            System.out.println(e.getMessage());
16        }
17    }
18 }
```

catch 块中处理异常的代码称为异常处理器(exception handler)。一个 try 块可以与一个或多个 catch 块相关联。例如源文件 5-9 试图从当前文件夹的文本文件 input.txt 中读取一行写入 output.txt 中。如果当前文件夹中没有 input.txt 则抛出 FileNotFoundException 异常；如果有 input.txt 但里面是空的，一行也没有，则抛出 NoSuchElementException 异常；如果试图读取 input.txt 时使用的对象已经关闭，则抛出 IllegalStateException 异常。程序期望捕获这三种异常就可通过多个 catch 块与 try 块关联来实现，如源文件 5-9 所示。

源文件 5-9　MultiCatchDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.PrintWriter;
4 import java.io.FileNotFoundException;
5 import java.util.NoSuchElementException;
6
7 public class MultiCatchDemo {
8     public static void main(String[] args) {
9         try (Scanner scanner = new Scanner(new File("input.txt"));
10                PrintWriter writer = new PrintWriter(new File("output.txt"))){
11            writer.print(scanner.nextLine());
12        } catch (FileNotFoundException e) {
13            e.printStackTrace();
```

```
14              System.out.println(e.getMessage());
15          } catch (NoSuchElementException e) {
16              e.printStackTrace();
17              System.out.println(e.getMessage());
18          } catch (IllegalStateException e) {
19              e.printStackTrace();
20              System.out.println(e.getMessage());
21          }
22      }
23 }
```

在源文件 5-9 中，如果 try 块中抛出异常，Java 的异常处理系统就自上而下依次将该异常的类型与 catch 中声明的类型进行匹配(match)。一旦匹配，就执行相应的 catch 块，catch 块执行完毕，则整个 try-catch 语句执行完毕，继续执行该语句的后随语句。匹配的意思是：抛出的异常对象的引用能够合法地赋值异常处理器声明的参数(e)。匹配并不要求抛出异常的类型与异常处理器声明的异常类型严格相同。如果抛出的异常是子类，而异常处理器声明的类型是父类，也认为匹配。

在源文件 5-9 中，一个 try 块与三个 catch 块相关联。自上而下，catch 块声明的异常类型分别是 FileNotFoundException、NoSuchElementException 和 IllegalStateException。如果在 try 块中抛出了 FileNotFoundException 类型的异常对象，由于该类型与 FileNotFoundException 匹配，那么执行的是第一个 catch 块；如果在 try 块中抛出了 NoSuchElementException 类型的异常对象，由于该类型与 NoSuchElementException 匹配，那么执行的是第二个 catch 块；如果在 try 块中抛出了 IllegalStateException 类型的异常对象，由于该类型与 IllegalStateException 匹配，那么执行的是第三个 catch 块。

也可以把期望捕获的异常通过“|”声明在一起，如源文件 5-10 第 12 行所示。

源文件 5-10　MultiCatchInOneDemo.java

```
1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.PrintWriter;
4 import java.io.FileNotFoundException;
5 import java.util.NoSuchElementException;
6
7 public class MultiCatchInOneDemo {
8     public static void main(String[] args) {
9         try (Scanner scanner = new Scanner(new File("input.txt"));
10                 PrintWriter writer = new PrintWriter(new File("output.txt"))){
11             writer.print(scanner.nextLine());
12         } catch (FileNotFoundException | NoSuchElementException
                         | IllegalStateException e) {
13             e.printStackTrace();
14             System.out.println(e.getMessage());
15         }
16     }
17 }
```

注意，由于子类的对象总能够上转型为祖先类的引用变量，下面的 try-catch 语句中所有期望的异常都能赋值给第一个 catch 块中的 e，导致其他 catch 块不会被执行：

```
try {
   ...
}
catch(Exception e) {
   System.out.println(e.getMessage());
}
catch(IOException e) {
   System.out.println(e.getMessage());
}
catch (FileNotFoundException e){
   System.out.println(e.getMessage());
}
```

如果抛出异常的语句 S 没有在任何 try 块内，那么 Java 会去包含该语句的方法的调用者中寻找相应的调用语句 R，检查 R 是否在 try 块中。如果是，则执行与 try 块相伴的 catch 块，catch 块执行完毕后，继续执行包含语句 R 的 try-catch 块的后随语句；如果在调用者中发现调用语句 R 没有在 try 块中，那么 Java 继续去语句 R 所在方法的调用者中寻找相应的调用语句。如果寻找到 main 方法也没有找到处理该异常的 try-catch 块，Java 的异常处理系统输出打印关于该异常的一些信息并终止程序。

总之，Java 异常处理系统的任务就是当程序中的异常发生后，为其寻找异常处理器。这个寻找是沿着方法的调用反向进行的，直到 main 方法。如果把一个方法看作一个节点，方法的调用是节点之间的有向边，那么 main 方法调用方法 methodA、methodA 方法又调用方法 methodB，等等，就形成了一个方法调用链。对异常处理器的寻找就是沿着方法调用链逆向进行的。如果直到 main 方法也没有找到异常处理器，那么异常处理系统向控制台输出一些消息，并终止程序执行。

如果抛出异常的语句在 try 块中，但是该异常不能与 try-catch 语句中的任何 catch 块声明的异常类型相匹配，那么该异常就沿着方法调用链向调用者方向传播。

使用异常要注意以下 6 点。

(1) 一个 try 块至少有一个 catch 块关联，可与多个 catch 块关联。

(2) finally 块可有可无。

(3) 必须声明如何处理受检异常。

(4) 异常对象由程序中某个方法在执行期间抛出给 JVM，而 catch 块则声明要求 JVM 交递的异常。

(5) 如果 JVM 存在某个异常对象，但没有找到等待该异常的任何 catch 块，则 JVM 中止程序运行。

(6) 不能在 finally 块中使用 return。在 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。

5.3 自定义异常

前面介绍了如何捕获和处理异常，但没有解释如何抛出异常。通过保留字 throw 抛出异常。例如源文件 5-11 是 JDK 中 java.lang.Math 类的类方法 addExact，该方法把给定的

两个 int 类型参数求和，如果加法运算的结果溢出，超出了 int 类型整数的范围，那么抛出 ArithmeticException 异常。

源文件 5-11　AddExact.java

```
class AddExact {
 /**
  * 返回两个整型参数的和
  * 如果相加的结果溢出，则抛出异常
  *
  * @param x 第一个整数
  * @param y 第二个整数
  * @return 相加的结果
  * @throws ArithmeticException 如果相加的结果整数溢出
  * @since 1.8
  */
  public static int addExact(int x, int y) {
      int r = x + y;
      //当且仅当两个参数的符号都和结果的符号相反
      if (((x ^ r) & (y ^ r)) < 0) {
          throw new ArithmeticException("integer overflow");
      }
      return r;
  }
}
```

当被加数与结果的符号相反而且加数也与结果的符号相反时发生溢出。整数的补码最高位是符号位：1 表示该整数为负数；是 0 则表示该整数为非负数。x^r 表示将 x 的每个二进制位与 r 的对应位进行异或操作，如果两个位相同则结果为 0，不同则结果为 1。y^r 同理，对 y 和 r 进行异或操作。(x^r)&(y^r)将上述两个异或结果进行按位与运算，只有当两个对应的位都是 1 时，结果的该位才是 1，否则为 0。如果被加数最高位和结果的最高位不同，那么异或的结果是 1；同理，如果加数最高位和结果的最高位不同，那么异或的结果也是 1；1 和 1 进行按位逻辑与运算，结果为 1。最高位为 1 的整数表示负数，负数小于 0，所以表达式((x^r)&(y^r))<0 为真，抛出异常。

源文件 5-12　MathDemo.java

```
public class MathDemo {
    public static void main(String[] args) {
        try {
            System.out.println(addExact(2, 3));
            System.out.println(addExact(2, Integer.MAX_VALUE));
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        }
    }

    public static int addExact(int x, int y) {
        int r = x + y;
        //Overflow iff
        //both arguments have the opposite sign of the result
        if (((x ^ r) & (y ^ r)) < 0) {
            throw new ArithmeticException("integer overflow");
```