Java 应用程序设计
异常处理

王晓东
wxd2870@163.com

中国海洋大学

November 17, 2017

参考书目

1. 张利国、刘伟 [编著], Java SE 应用程序设计, 北京理工大学出版社, 2007.10.

本章学习目标

1. Java 异常的概念和分类
2. Java 异常处理机制
3. 用户自定义异常
4. 断言

大纲

接下来…

异常的概念及分类

Java 异常处理机制
- 捕获异常
- 声明抛出异常
- 人工抛出异常

用户自定义异常

断言

什么是异常

在 Java 语言中，程序运行出错被称为出现异常。异常（Exception）是程序运行过程中发生的事件，该事件可以中断程序指令的正常执行流程。

Java 异常分为两大类：

1. **错误**（Error）是指 JVM 系统内部错误、资源耗尽等严重情况。
2. **违例**（Exception）则是指其他因编程错误或偶然的外在因素导致的一般性问题，例如对负数开平方根、空指针访问、试图读取不存在的文件以及网络连接中断等。

什么是异常

Java 运行时异常示例

CODE ▸ TestException.java

```
1 public class TestException {
2   public static void main(String[] args) {
3     String friends[] = {"Lisa", "Bily", "Kessy"};
4     for(int i = 0; i < 5; i++) {
5       System.out.println(friends[i]);
6     }
7     System.out.println("\n this is the end");
8   }
9 }
```

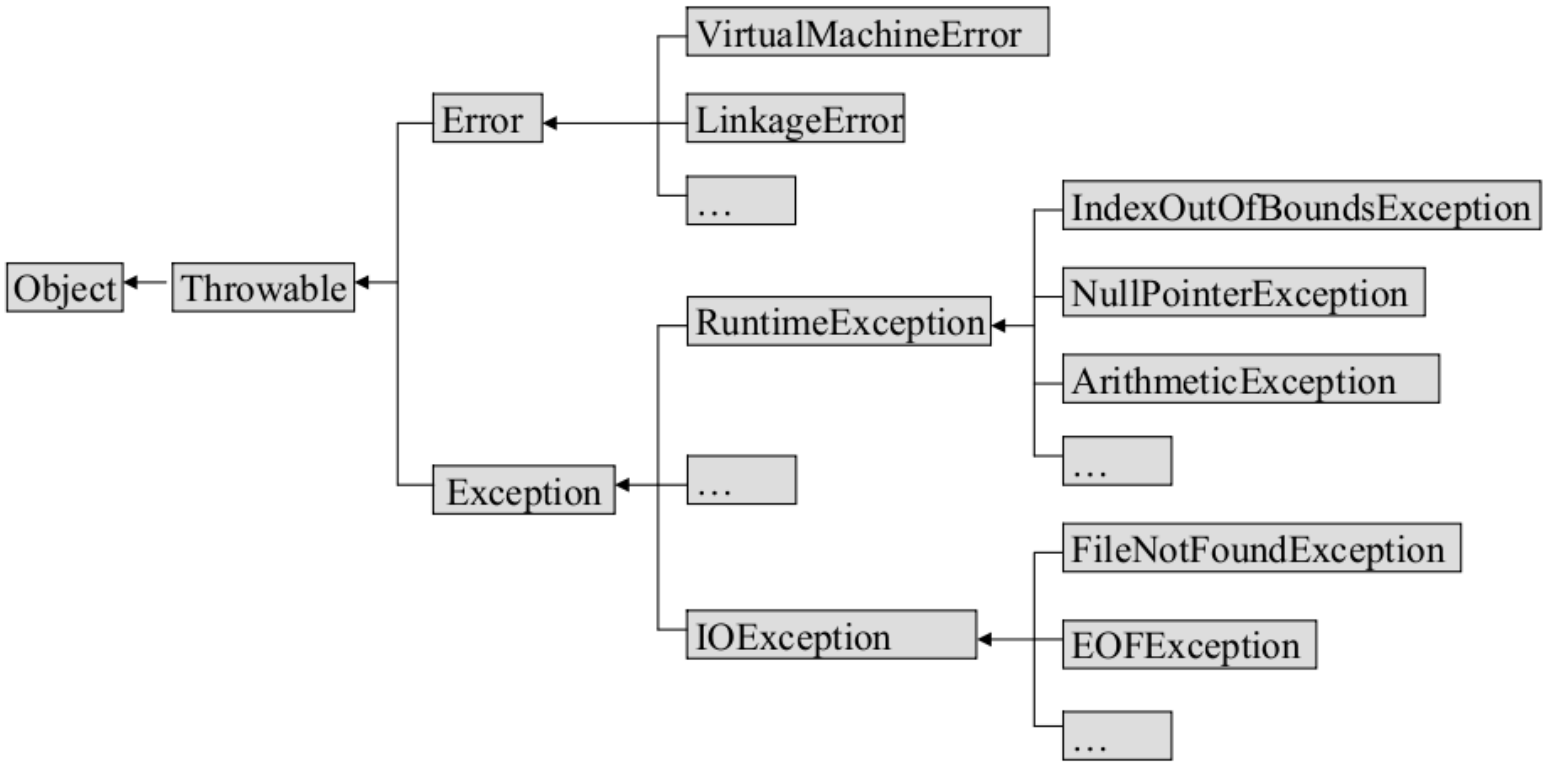
程序编译通过，运行时出错。

output

```
Lisa
Bily
Kessy
Exception in thread "main" java.lang.
ArrayIndexOutOfBoundsException: 3
at TestException.main(TestException.java:6)
```

Java 异常分类

Throwable 类是 Java 语言中所有异常类的父类。



常见异常（错误）

❖ 链接错误（LinkageError）

LinkageError 是指程序链接错误。例如，一个类中用到另外一个类，在编译前一个类之后，后一个类发生了不相容的改变时，再使用前一个类则会出现链接错误。
最常见的就是后一个类的.class 文件被误删除。

常见异常（错误）

❖ 虚拟机错误（VirtualMachineError）

当 Java 虚拟机崩溃或用尽了它继续操作所需的资源时，会抛出该错误。其中比较有代表性的是 StackOverflowError，当应用程序递归太深而导致栈内存溢出时会出现该异常。

```
1  public class TestVMError {
2    public static void main(String[] args) {
3      TestVMError t = new TestVMError();
4      t.f(100000);
5    }
6    public int f(int n) {
7      if (n <= 0) {
8        return 0;
9      }
10     int k = n * this.f(n-1);
11     return k;
12   }
13 }
```

output

```
Exception in thread "main" java.lang.StackOverflowError
at TestException.f(TestException.java:7)
at TestException.f(TestException.java:10)
```

常见异常

❖ RuntimeException

- 错误的类型转换
- 数组下标越界
- 空指针访问

空指针异常（NullPointerException）

如果试图访问不指向任何对象的引用变量的成员，将会产生空指针异常。例如：

```
1 Person p = null;
2 System.out.println(p.age);
```

常见异常

❖ IOException

- 从一个不存在的文件中读取数据
- 越过文件结尾继续读取
- 连接一个不存在的 URL

IOException 示例 下述代码无法编译！

CODE ◗ TestIOException

```
1  import java.io.*;
2  public class TestIOException {
3    public static void main(String[] args) {
4      FileInputStream in = new FileInputStream("myfile.txt");
5      int b;
6      b = in.read();
7      while (b != -1) {
8        System.out.print((char) b);
9        b = in.read();
10     }
11     in.close();
12   }
13 }
```

常见异常

❖ 对上述代码无法编译的解释

只要是有可能出现 IOException 的 Java 代码，在编译时就会出错，而不会等到运行时才发生。
编译出错信息大致如下：

output

```
TestIOException.java:4: 未报告的异常 java.io.FileNotFoundException;
必须对其进行捕捉或声明以便抛出
    FileInputStream in = new FileInputStream("myfile.txt");
... ...
```

接下来…

异常的概念及分类

Java 异常处理机制
- 捕获异常
- 声明抛出异常
- 人工抛出异常

用户自定义异常

断言

Java 异常处理机制

Java 异常处理宗旨：

- 返回到一个安全和已知的状态；
- 能够使用户执行其他的命令；
- 如果可能, 则保存所有的工作；
- 如果有必要，可以退出以避免造成进一步的危害。

Java 异常处理机制：

- Java 程序执行过程中如出现异常, 系统会监测到并自动生成一个相应的异常类对象，然后再将它交给运行时系统；
- 运行时系统再寻找相应的代码来处理这一异常。如果 Java 运行时系统找不到可以处理异常的代码，则运行时系统将终止，相应的 Java 程序也将退出。
- 程序员通常对错误（Error）无能为力，因而一般只处理违例 (Exception)。

接下来…

捕获异常

```
try {
  ... //可能产生异常的代码
} catch (ExceptionName1 e) {
  ... //当产生 ExceptionName1 型异常时的处置措施
} catch (ExceptionName2 e) {
  ... //当产生 ExceptionName2 型异常时的处置措施
} finally {
  ... //无条件执行的语句
}
```

```
public class Test {
  public static void main(String[] args) {
    String friends[]={"Lisa","Billy","Kessy"};
    try {
      for(int i = 0; i < 5; i++) {
        System.out.println(friends[i]);
      }
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("index err");
    }
    System.out.println("\nthis is the end");
  }
}
```

使用 finally 语句

finally 语句

finally 语句是可选的，其作用是为异常处理提供一个统一的出口，使得在控制流转到程序的其他部分以前，能够对程序的状态作统一的管理。

```
1  public class Test {
2    public static void main(String[] args) {
3      String friends[]={"Lisa","Billy","Kessy"};
4      try {
5        for (int i = 0; i < 5; i++) {
6          System.out.println(friends[i]);
7        }
8      } catch(ArrayIndexOutOfBoundsException e) {
9        System.out.println("index␣err");
10       return;
11     } finally {
12       System.out.println("in␣finally␣block!");
13     }
14     System.out.println("this␣is␣the␣end");
15   }
16 }
```

使用 finally 语句

不论 try 代码块中是否发生了异常事件，finally 块中的语句都会被执行。当 catch 语句块中出现 return 语句时，finally 语句块同样会执行。
上述代码的输出：

output

```
Lisa
Billy
Kessy
index err
in finally block!
```

操纵异常对象

发生异常时，系统将自动创建异常类对象，并将作为实参传递给匹配的 catch 语句块的形参，这样我们就可以在语句块中操纵该异常对象了。主要使用异常类的父类 Throwable 中定义的两个成员方法：

- public String getMessage() 返回描述当前异常的详细消息字符串；
- public void printStackTrace() 用来跟踪异常事件发生时运行栈的内容，并将相关信息输出到标准错误输出设备。本方法比较常用，在没有找到适合的异常处理代码时，系统也会自动调用该方法输出错误信息。

追踪运行栈信息

CODE ▸ A.java

```
1  public class A {
2    public void work(int[] a) {
3      String s = this.contain(a, 3);
4      System.out.println("Result:␣" + s);
5    }
6
7    public String contain(int[] source, int dest) {
8      String result = "no!";
9      try {
10       for (int i = 0; i < source.length; i++) {
11         if (source[i] == dest)
12         result = "yes!";
13       }
14     } catch(Exception e) {
15       System.out.println("异常信息: " + e.getMessage());
16       System.out.println("运行栈信息: ");
17       e.printStackTrace();
18       result = "error!";
19     }
20     return result;
21   }
22 }
```

追踪运行栈信息

CODE ▸ MyTest.java

```
1 public class MyTest {
2   public static void main(String[] args) {
3     A tst = new A();
4     tst.work(null);
5   }
6 }
```

程序输出结果为:

output

```
Exception Message: null
Stack Trace:
java.lang.NullPointerException
    at A.contain(A.java:9)
    at A.work(A.java:3)
    at MyTest.main(MyTest.java:4)
Result: error!
```

捕获和处理 IOException

```
1  import java.io.*;
2  public class TestIOException {
3    public static void main(String[] args) {
4      try {
5        FileInputStream in = new FileInputStream("myfile.txt");
6        int b;
7        b = in.read();
8        while(b != -1) {
9          System.out.print((char) b);
10         b = in.read();
11       }
12       in.close();
13     } catch (FileNotFoundException e) {
14       System.out.println("File␣is␣missing!");
15     } catch (IOException e) {
16       e.printStackTrace();
17     }
18     System.out.println("It's␣ok!");
19   }
20 }
```

FileNotFoundException 是 IOException 的子类，基于多态性机制，后一个 catch 语句也可以处理 FileNotFoundException，因此前一个 catch 语句块可以取消，但这样就无法区分“文件不存在”或其他 I/O 异常了。

异常处理知识点

- 对于只可能产生 RuntimeException 的代码可以不使用 try-catch 语句进行处理，如果对于这些相对安全的代码仍然采用了 try 语句块的形式，则 try 后可以省略 catch 语句块或 finally 语句块，但不能同时省略。
- 如果试图捕获和处理代码中根本不可能出现的异常，编译器也会指出这种不当行为。

接下来…

声明抛出异常

声明抛弃异常是 Java 中处理违例的第二种方式如果。

抛出异常

一个方法中的代码在运行时可能生成某种异常，但在本方法中不必、或者不能确定如何处理此类异常时，则可以声明抛弃该异常；此时方法中将不对此类异常进行处理, 而是由该方法的调用者负责处理。

语法格式：

```
1 [< 修饰符 >] < 返回值类型 > < 方法名 > (< 参数列表 >) [throws < 异常类型 >
2 [,< 异常类型 >]*] {
3   [< Java语句 >]*
4 }
```

声明抛出异常

CODE ▸ TestThrowsException.java

```
1  import java.io.*;
2  public class TestThrowsException {
3    public static void main(String[] args) {
4      TestThrowsException t = new TestThrowsException();
5      try {
6        t.readFile();
7      } catch (IOException e) {
8        System.out.println(e);
9      }
10   }
11   public void readFile() throws IOException {
12     FileInputStream in = new FileInputStream("myfile.txt");
13     int b;
14     b = in.read();
15     while (b != -1) {
16       System.out.print((char) b);
17       b = in.read();
18     }
19     in.close();
20   }
21 }
```

声明抛出异常

注意

- 除非事先约定，否则在开发过程中不要在自己编写的方法中采用抛出异常的方式。
- **重写方法不允许抛出比被重写方法范围更大的异常类型。**例如 IOException 重写后抛出 FileNotFoundException 和 EOFException 被允许，而抛出 Exception 则不被允许。

接下来…

人工抛出异常

Java 异常类对象除了在程序运行出错时由系统自动生成并抛出之外，也可根据需要人工创建并抛出：

```
1 IOException e = new IOException(); // 创建异常类对象
2 throw e; // 抛出操作，即将该异常对象提交给Java运行环境
```

被抛出的必须是 Throwable 或其子类类型的对象，下述语句在编译时会产生语法错误：

```
1 throw new String("want to throw");
```

人工抛出异常

CODE ▶ TextThrowException.java

```
1  import java.util.Scanner;
2
3  public class TestThrowException {
4    public static void main(String[] args) {
5      TestThrowException t = new TestThrowException();
6      System.out.print("Please␣input␣your␣age:␣");
7      System.out.print("Your␣age:␣" + t.inputAge());
8    }
9    public int inputAge() {
10     int result = -1;
11     Scanner scan = new Scanner(System.in);
12     while (true) {
13       try {
14         result = scan.nextInt();
15         if (result < 0 || result > 130) {
16           Exception me = new Exception("You␣come␣from␣Mars?␣");
17           throw me;
18         }
19         break;
20       } catch (Exception e1) {
21         System.out.println(e1.getMessage() + "Please␣input␣your␣age␣again:␣");
22         continue;
23       }
24     }
25     return result;
26   }
27 }
```

人工抛出异常

上述情况下利用异常处理机制实现数据取值范围的检查并不太合适。原则如下：

- 当明确知道可能出错的地方或能够通过简单的检查而有效防止错误发生，就应该使用 if-else 语句来预防错误发生；
- 只有当我们无法明确知道错误发生之处或无法完全避免异常，才不得不通过异常处理的方式来捕获和处理异常。

接下来…

异常的概念及分类

Java 异常处理机制
- 捕获异常
- 声明抛出异常
- 人工抛出异常

用户自定义异常

断言

用户自定义异常

- Java 语言针对常见异常状况已事先定义了相应的异常类型，并在程序运行出错时由系统自动创建相应异常对象并进行抛出、捕获和处理，因此一般不需要用户人工抛出异常对象或定义新的异常类型，但针对特殊的需要也可以这样做。
- 一般通过**继承 Exception 类**来实现异常自定义，由于用户自定义的异常不会由系统自动检测并抛出，所以只能靠人工触发并抛出。

用户自定义异常

CODE ▸ MyException.java

```
1  public class MyException extends Exception {
2    private int idnumber;
3    public MyException(String message, int id) {
4      super(message);
5      this.idnumber = id;
6    }
7    public int getId() {
8      return idnumber;
9    }
10 }
```

构造方法中使用 super 调用其父类 Exception 的有参构造方法，以在创建异常对象时将用户的定制的报错信息传递给父类中定义的 private 属性 Message（该属性由 Throwable 类定义），将来在捕获和处理异常时就可以通过调用该对象的 getMessage() 方法访问到该信息了。

用户自定义异常

CODE ▶ TestCustomizingException.java

```
1  public class TestCustomizingException {
2    public void regist(int num) throws MyException {
3      if (num < 0) {
4        throw new MyException("人数为负值，不合理", 3);
5      }
6      System.out.println("登记人数: " + num);
7    }
8    public void manager() {
9      try {
10       regist(-100);
11     } catch (MyException e) {
12       System.out.println("登记失败，出错种类" + e.getId());
13       e.printStackTrace();
14     }
15     System.out.print("本次登记操作结束");
16   }
17   public static void main(String args[]) {
18     new TestCustomizingException().manager();
19   }
20 }
```

程序输出结果：

output

```
登记失败，出错种类 3
MyException: 人数为负值，不合理 ...
```

接下来…

异常的概念及分类

Java 异常处理机制
- 捕获异常
- 声明抛出异常
- 人工抛出异常

用户自定义异常

断言

断言

从 JDK1.4 版本开始，Java 语言中引入了断言（Assert）机制，允许 Java 开发者在代码中加入一些检查语句，主要用于程序调试目的。

- 断言机制在用户定义的 boolean 表达式（判定条件）结果为 false 时抛出一个 Error 对象，其类型为 AssertionError；
- 当我们需要在约定的条件不成立时中断当前操作的话，可以使用断言；
- 作为 Error 的一种，断言失败也不需捕获处理或者声明抛出，一旦出现了则终止程序、不必进行补救和恢复。

启用和禁用断言

❖ 开启断言功能

Java 运行时环境默认设置为关闭断言功能，因此在使用断言以前，需要在运行 Java 程序时先开启断言功能。

```
1 >java -ea MyAppClass
```

或者：

```
1 >java -enableassertions MyAppClass
```

❖ 关闭断言功能

```
1 >java -da MyAppClass
```

或者：

```
1 java -disableassertions MyAppClass
```

启用和禁用断言

❖ Eclipse IDE 开启断言

在项目上点击右键 ➠ Run As ➠ Run Configurations ➠ Arguments，在 VM arguments 中，加入 -enableassertions 或 -ea 即可。

使用断言

❖ 断言的语法格式❶

```
1 assert <boolean 表达式>;
```

CODE ◗ TestAssertion.java

```
1  public class TestAssertion {
2    public static void main(String[] args) {
3      new TestAssertion().process(-12);
4    }
5    public void process(int age) {
6      assert age >= 0;
7      System.out.println("您的年龄: " + age);
8      // ---
9    }
10 }
```

output

```
Exception in thread "main" java.lang.AssertionError
at TestAssertion.process(TestAssertion.java:8)
at TestAssertion.main(TestAssertion.java:4)
```

使用断言

❖ 断言的语法格式❷

```
1 assert < boolean 表达式 >:< 表达式 2 >;
```

CODE ◗ TestAssertion2.java

```
1  public class TestAssertion2 {
2    public static void main(String[] args) {
3      new TestAssertion2().process(-12);
4    }
5    public void process(int age) {
6      assert age >= 0: "年龄值不合理";
7      System.out.println("您的年龄: " + age);
8      //---
9    }
10 }
```

输出结果:

output

```
Exception in thread "main" java.lang.AssertionError: 年龄值不合理
at TestAssertion.process(TestAssertion.java:8)
at TestAssertion.main(TestAssertion.java:4)
```

使用断言

断言失败时，系统会自动将表达式 2 的值传递给新创建的 AssertionError 对象，进而将其转换为一个消息字符串保存起来，这样就可以在获得更多/更有针对性的检查失败细节信息。因此，其中的表达式 2 可以是任何基本数据类型或引用数据类型，但必须提供一个值，即不能为 void 值。

使用断言是为了在测试阶段确定程序内部出错位置和出错信息，而不是控制程序流程。

THE END

wxd2870@163.com