第 5 章

异 常 处 理

本章内容

JVM 会将程序运行过程中出现导致程序中断的信息封装成异常对象返回用户。Java 官方为各类异常情况建立了丰富的异常类。基于异常对象，程序就可以通过匹配机制识别出异常信息并激活一整套异常处理的代码。异常对象的出现从根本上实现了程序中核心业务代码与防护性代码的分离。本章将介绍 Java 中异常的定义和分类，以及捕获和处理异常对象的方法，最后是自定义异常的一些规范。

学习目标

- 了解异常的定义，理解在 Java 中 Exception 和 Error 的区别。
- 掌握异常的分类，理解 RuntimeException 的特殊性，知道一些常用的 RuntimeException 子类。
- 熟练掌握异常的处理方法。
- 掌握异常的声明和抛出。
- 了解自定义异常类的方法。

5.1 异 常 基 础

5.1.1 程序出错和解决方案

异常状态是指程序在运行过程中发生的、由外部问题导致的程序运行异常事件。异常的发生往往会中断程序的运行，而这种中断出现是否就意味着程序编写出现了错误吗？不尽然，这种运行时异常情况产生的原因有很多，有的是用户错误引起的，如在计算器中除以 0；有的是程序错误引起的，如给出的数组下标为负值；有的是其他一些物理硬件错误引起的，如用户的内存容量不足。归根到底，异常状态的出现就是由于代码实际工作状态与预计的工作状态不一致。虽然不是全部，但很多异常状态都是可以通过防护性代码在程序内部解决的。

程序出现的异常状态可以分为 3 种，不同类型异常状态的解决方式也是不同的。

1. 语法错误

在编译过程中因为没有遵循语法规则而导致的程序异常状态称为语法错误或编译错误，如缺少必要的标点符号、关键字输入错误、数据类型不匹配等。对于语法错误，一般程序编译器会自动提示相应的错误地点和错误原因。因此，这类异常状态在三类异常状态中最容易发现。通过修正代码中错误的语法、查阅 API 文档或其他资料，这类异常状态基本都

可以在程序编写阶段得到修正。

2. 逻辑错误

逻辑错误是指程序没有按期望的要求、预期的逻辑顺序执行而产生的异常状态。程序的逻辑错误仅依靠编译器是无法检测的，大多数逻辑错误是程序运行时出现，甚至是特定时刻才会出现的，而产生的原因也是多样且复杂的。逻辑错误是现阶段最复杂的一类异常状态，而且没有通用的解决方案，因此人们在设计程序语言时，会尽量将逻辑错误转换为语法错误或运行错误，从代码层面以一种规范的方法，明确异常状态处理的流程，以降低程序出现逻辑错误的可能性。本章所讲的异常处理就是这一逻辑下的产物。

3. 运行错误

运行错误是指程序在运行过程中，运行环境发现了超出程序代码承载能力的情况下出现的异常状态，一个开放的尤其是与用户互动的程序在运行时不可避免会出现不符合程序主逻辑的分支，如一个读写文件内容的程序在运行时发现无法打开指定文件；一个可以运行整数加法的加法器，被赋予了两个浮点数；用户输入的用户名和密码不匹配等。

为了让程序可以在可控的条件下正常运行，一个最直观的想法就是限定并规范程序的输入内容，即建立黑白名单。然而，黑白名单的建立会大大影响核心程序的开发速度，如一个加法计算器，如果使用黑名单限制非字母的字符输入，那么科学记数法的 E、十六制的 ABCDEF 是否要加入例外？如果使用重载的方法标定白名单，用户输入整数和浮点数，甚至是虚数都是合理的，程序里需要大量的重载方法。另外，使用黑白名单的方式处理各类错误和异常的方法，需要程序员了解程序运行细节，方能定制各种情况的处理代码，这样建立的异常处理代码必然与核心代码是强耦合的，在修改时会出现牵一发动全身的情况。

> **例**
>
> 程序员送给他的女朋友一支精美的口红(主逻辑)。为了确保送口红的过程顺利进行，他可能还需要确定口红包装精美、口红是否与女朋友的造型匹配、女朋友对颜色的喜好等问题(黑白名单处理方案)……等待他的将会是一个异常状态爆炸性增长的局面。

“陷阱式”的异常生成机制为我们处理程序异常提供了一种新思路。在 Java 这个面向对象程序语言中，JVM 会根据内存状态、异常状态产生的条件或程序预先定义条件产生特定异常类对象。这个异常对象包含该异常状态生成时的栈轨迹，它包括异常栈所指向方法的名字，方法所在的类名、文件名及在代码第几行触发了错误、异常等信息。

异常对象的出现，改变了程序员应对异常的方式。通过异常对象包含的信息，程序员知道程序哪里出现了问题，出了什么类型的问题。问题的原因和类型被封装成一个独立的对象，这个对象与正常运行的代码是分离的。这意味着，程序员可以更专注于代码核心逻辑的编写，所有不适合该逻辑核心的异常状态都被封装为一个个异常对象抛出给系统。面对出现的异常对象，程序员可以选择自己处理，还是交给其他人来处理。基于异常对象的处理代码与核心逻辑是弱耦合，甚至是解耦合的。在 Java 中这种基于捕获的异常处理方案，实现了程序主体与异常处理代码的分离，简化主逻辑功能设计时的代码复杂度，也丰富了异常情况的处理手段，让专业的人员做专业的事情，十分符合现阶段大规模软件合作开发模式。

续例

“陷阱式”的异常处理机制可以将那个送口红的程序员从包装的选择、氛围的营造、喜好调查等问题的漩涡中“解救”出来，他只需要关注送口红这个主进程就行。出现包装问题就交给导购员小姐姐，出现造型问题就交给 Tony 老师……高效甩锅！成功地将所有异常情况排除到送口红这个主进程之外。

5.1.2 Error 和 Exception

Java 将程序运行过程中的异常状态分为 Error 和 Excpetion，它们都是 java.lang.Throwable 类的子类。Throwable 类型的对象在出现异常状态时由 JVM 抛出，或者可以由程序中 throw 语句主动抛出。Java 中异常类的继承关系，如图 5-1 所示。

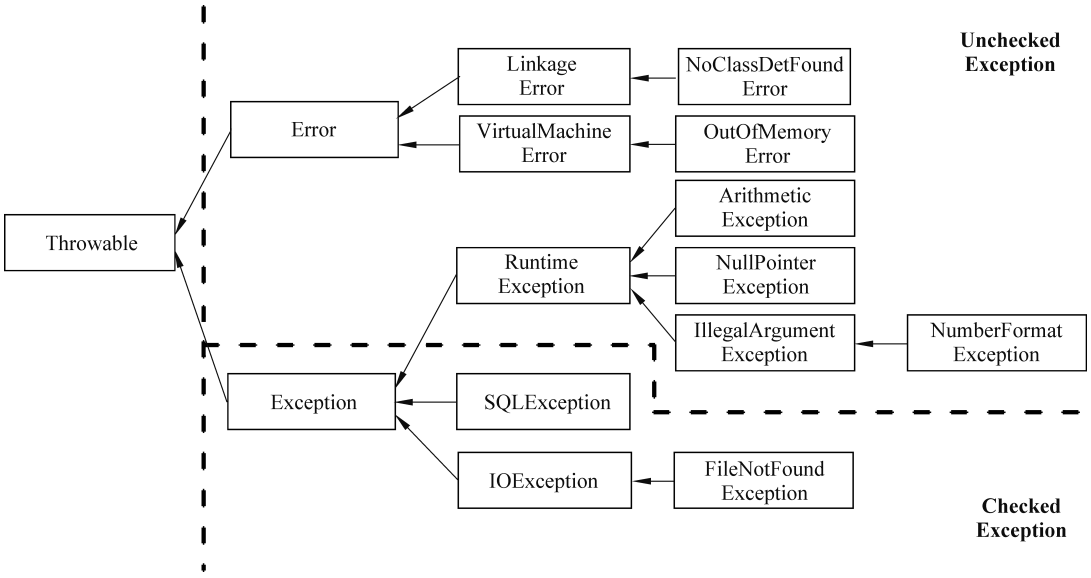

图 5-1 Java 中异常类的继承关系图（局部）

1. Error

Error 是程序无法处理的异常状态。在 Java 中，这类程序异常状态通过 Error 的子类描述，且程序在编译时也不会要求程序处理 Error，因为它们在应用程序的控制和处理能力之外。Error 是程序不可自查的，且无法自我修正的。在 Error 发生时，JVM 一般会选择线程终止，大多数 Error 与代码执行环境有关，例如：

JVM 在为程序分配内存时，发现物理内存不足时，JVM 会抛出 OutOfMemoryError 对象。

JVM 在加载类的定义时，但无法找到该类的定义时，会抛出 NoClassDefFoundError 对象。

2. Exception

Exception 是程序本身可以处理的异常，它是由程序或外部环境不匹配所引起的，程序中应当尽可能去捕获和处理这些异常。Exception 类对象可以被 Java 异常处理机制所捕获，它是异常处理的核心。Java 中的 Exception 主要分为以下两类。

(1) 非检查型异常(Unchecked Exception)。

Java语言在编译时,在程序中不要求用户给出(也可以选择给出)处理这些异常的代码。从继承结构来看,这类异常都是RuntimeException以及它们的子类异常对象。RuntimeException类异常对象出现的原因多半是代码中出现了明显的逻辑问题。例如:

当程序中用数字除以0时会抛出ArithmeticException异常对象。

当强制类型转换失败时会抛出ClassCastException类型转换异常对象。

当使用数组索引越界时就会抛出ArrayIndexOutOfBoundsException异常对象。

当程序中引用变量在未持有对象的情况下,引用变量调用成员方法或访问成员属性时就会抛出NullPointerException异常对象。

(2) 检查型异常(Checked Exception)。

Java语言强制要求程序员为这类异常对象编写异常处理代码,否则编译不会通过。Exception自身及非RuntimeException分支的子类都是检查型异常,这样的异常一般是由程序的运行环境导致的,Checked Exception是Java中异常处理的主要工作目标。

检查型异常对象的处理可以让程序更加健壮稳定,因为程序可能被运行在各种未知的环境下,而程序员无法干预用户如何使用他编写的程序,程序员就应该为这些可预见的异常(各类具体异常子类)和不可预见的异常(Exception类)准备好处理代码。例如:

程序与数据源进行交互时数据库报错时抛出的SQLException的异常对象。

程序使用I/O资源时出现异常时抛出的IOException异常。

将应用试图通过字符串名加载类,但没有找到指定名称的类的定义时,会抛出ClassNotFoundException异常对象。

续例

对于那个送口红的程序员来说,各类异常都会导致主进程中断。

(1) Error运行环境的问题。它是送口红这个程序依附的,但这个程序无法解决的问题,如程序员没有女朋友。

(2) Unchecked Exception是常识性的问题,一般人不会犯。程序员希望女友告诉他她喜欢口红颜色的RGB值。

(3) Checked Exception是程序员必须处理的问题。女友家有熊孩子,玩具必须备上,否则口红难逃被偷去练书法的下场。

所有异常都是绑定在具体的方法之上的。会抛出异常的方法有成员方法、构造方法和静态方法。例如:

- String类的成员方法substring(int beginIndex)会抛出IndexOutOfBoundsException。
- String类的构造方法String(byte[] bytes, String charsetName)会抛出UnsupportedEncodingException。
- Integer类中的静态方法parseInt(String str)会抛出NumberFormatException。

在调用相关方法时可以多看看API文档,确定程序中调用的方法是否会抛出异常。如果抛出异常,异常是什么类型的,以及抛出异常是Checked Exception还是Unchecked Exception,了解这些情况对用户编写代码都是至关重要的。

5.2 异 常 处 理

本节将以 java.lang 包中的 ArithmeticException 异常为例介绍如何编写异常处理代码。需要注意 ArithmeticException 是 Unchecked Exception，这种异常即使不附加异常处理代码，编译依然可以通过。基于这个 Unchecked Exception，可以更全面地展示各种处理异常的方法。通过比较，也可以更直观地看到 Java 中异常处理的思路和各种方案的优缺点。

5.2.1 异常出现

这是一个有潜在风险的例子：两个随机整数相除，打印它们的商。如代码 5-1 所示，x 是 1～3 的随机数，y 是 0～2 的随机数。若 y 随机到 0，因为除数不能为 0，程序会出现 ArithmeticException 异常。

```
//代码 5-1  可能出现异常的程序
public class TestArException {
    public static void computer(){
        int x = (int) (1 + (Math.random() * 2));        //x=1,2,3
        int y = (int) (Math.random() * 2);              //y=0,1,2
        int result;
        result = x/y;
        System.out.println("the result is:" + result);
    }

  public static void main(String[] args){
        computer();
    }
}
```

某一次运行结果可能显示为：

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
at * * * .TestArException.computer(TestArException.java:8)
at * * * .TestArException.main(TestArException.java:12)
```

上面的程序在运行时报告了算术异常（ArithmeticException），程序运行停止，代码中输出语句 System.out.println("the result is:"＋result)未被执行，main()主线程在异常抛出处中止。

建立黑白名单，程序可以通过添加一个 if 条件对除数的值进行测试，避免异常发生，如代码 5-2 所示。

```
//代码 5-2  使用 if 语句屏蔽异常情况出现
public class TestArithmeticException_If {
    public static void computer() {
        int x = (int) (1 + (Math.random() * 2));
```

```
            int y = (int) (Math.random() * 2);
            int result ;
            if (y != 0) {
                result = x/y;
                System.out.println("the result is:" + result);
            } else {
                System.out.println("Devisor cannot be zero");
            }
        }
        public static void main(String[] args) {
            computer();
        }
    }
```

如果 y 的值为 0，那么运行结果为：

```
Devisor cannot be zero
```

在代码 5-2 中，if 分支语句用来防止出现除数 y 为 0 的情况。这意味着程序员需要预先具备两个知识“除法中除零是一种错误，以及除数是 y”这两条先验知识，同时将错误预防代码嵌入程序当中。随着问题规模的不断扩大，这种上知天文、下知地理的要求，对于程序设计人员来说几乎是不可能实现的。

现在解决异常的主流思路是将异常情况的处理交给专业处理异常的代码去处理。Java 中采用的就是“陷阱式”异常处理机制——如果不触发异常这个陷阱，程序正常运行；一旦触发异常陷阱，生成异常对象，将异常对象转交给异常处理代码。这就是 Java 的异常处理方法的核心。

5.2.2　主动异常处理——定义异常处理代码

Java 使用 try 和 catch 关键字可以捕获程序段出现的异常，并将程序引向指定的异常处理代码段。具体来说就是：我们需要通过查阅 API 文档或测试代码确定哪些语句会抛出异常，抛出哪些类型的异常，然后按异常类型使用 catch 语句进行匹配异常，将程序导向异常处理代码块。

主动异常处理的 try-catch-finally 代码块的 Java 语法为：

```
try{
    语句块;          //可能出现异常的语句块
} catch (* * * Exception e) {
    语句块;          //该类异常对象出现后的语句块
} finally {
    语句块;          //异常无论是否发生，总是要执行的代码
}
```

在 try…catch…finally 代码块中：

try 语句块是必须存在且只能存在一块的语句块。用户将被监视运行的代码放入 try 语句块中。如果被监视运行的语句抛出异常对象，则代码在异常出现处中断，转而在 catch

中查找可以匹配异常对象的引用变量(异常类型),并激活该 catch 语句段。

注意

try 语句块中只会抛出一个异常对象,在第一个异常出现时立即中断 try 语句段的代码执行。因为代码不再执行,即使后续语句也可以抛出异常对象,后续的异常对象也不会出现。

catch 语句也是必须存在的。它可以有多个并列的语句块。当 try 语句段中出现异常对象时,catch 会捕获到发生的异常,并执行相应的 catch 块中代码。需要额外说明的是,持有异常对象的引用变量 e 是 final 的,用户不能在 catch 块中修改它的值。如果 try 段没有抛出任何异常或抛出的异常无法匹配(需要委托处理),则被略过所有 catch 语句块。

finally 语句块是可选的。finally 语句块为异常处理代码提供了统一的出口,使得控制流程在转到程序其他部分之前,能够对程序的状态做统一的管理。在 finally 语句中,通常使用 finally 可以进行资源的清除、回收物理资源工作,如关闭打开的文件、删除临时文件、变量还原默认值、网络的断开等。finally 语句块紧跟在 catch 语句之后,无论 try 语句块和 catch 语句块执行情况如何,该语句块都会被执行。

try…catch…finally 代码块的执行流程为：若 try 段语句正常执行完毕,就会跳过 catch 段,之后进入 finally 段;若 try 段中语句出现异常,则匹配 catch 语句块,之后进入 finally 段。

注意

(1) try、catch、finally 三个代码块中变量的作用域为独立代码块,彼此之间局部变量不能通用。若需要在代码块中统一使用一个变量,需要在异常处理代码块外声明,如 6.2.1 节中所示的异常处理框架,在 try 段及 finally 段都需要使用变量 fis 的情况下,将其声明在 try…catch…finally 代码块之外就是必然的选择。

(2) Java 垃圾回收机制不会回收任何物理资源,垃圾回收机制只能回收堆内存中对象所占用的内存。虽然物理资源的持有者会在其生命周期结束后自动释放,但是对于相对稀缺的物理资源还是建议使用 finally 语句段立即释放回收,以免后续程序因为物理资源访问受限出现异常。

这里使用主动异常处理技术对代码 5-2 进行改造,如代码 5-3 所示。

```
//代码 5-3  主动异常处理除 0 异常
public class TestTry {
    public static void computer(){
        try {
            int x = (int) (1 + (Math.random() * 2));
            int y = (int) (Math.random() * 2);
            int result = x/y;              //可能产生异常的语句
            System.out.println("the result is:" + result);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException 类的对象出现");
            e.printStackTrace();
        } finally {
```

```
            System.out.println("这是 finally 语句块,总会被执行的!");
        }
    }
    public static void main(String[] args){
        computer();
    }
}
```

在代码 5-3 中,程序的主体 result = x/y 放置在 try 语句块当中,被监视起来,而未对该语句的运行做任何限制(如 y! =0)。

如果 x 的值是 2,y 的值是 1,那么 try 块中程序段正常执行,就像没有使用 try 段进行包裹一样。结果如下。

```
the result is: 2
这是 finally 语句块,总会被执行的!
```

如果 y 的值是 0,那么 result = x/y 出现 ArithmeticException 异常对象。因为该语句被 try 段监视,异常对象出现后正常代码中断,语句 System.out.println("the result is:" + result)不再执行,转而执行异常处理段代码:

```
catch (ArithmeticException e) {
    System.out.println("ArithmeticException 类的对象出现");
    e.printStackTrace();
}
```

输出结果如下。

```
ArithmeticException 类的对象出现
这是 finally 语句块,总会被执行的!
java.lang.ArithmeticException:/by zero
at * * * .TestTry.computer(TestTry.java:9)
at * * * .TestTry.main(TestTry.java:21)
```

上面例子中对捕获到的异常对象的处理方式虽然仅仅是输出了异常的信息,但是有这个框架,用户就可以将更专业的异常处理代码引入其中。另外,为了方便 debug 程序,一般会使用 printStackTrace()方法打印异常对象 e 中包含的异常堆栈信息。

在主动异常处理过程中,业务逻辑与异常处理代码在一个方法内实现了分离,业务逻辑在 try 段,异常处理代码在 catch 语句群中。这种代码的分离可以让核心业务逻辑更加清晰。

5.2.3　委托异常处理——方法抛出异常

在主动异常处理的过程中,业务代码和异常处理代码的分离是在方法体内实现的。比方法体内分离更彻底的分离就是将异常处理代码分离到方法体之外,这就需要使用异常处理的第二种技术:委托异常处理。

如果方法体内出现异常对象,本方法不处理,用户可以将异常对象返回给方法的调用者

（就像发现犯罪要报警一样），主逻辑与异常处理分离到不同的方法当中了。在代码层面，只需要在可能出现异常对象的方法声明处，通过 throws 关键字声明在运行过程中方法体抛出异常类型即可。一个方法可以同时声明抛出多种异常，各异常类之间用逗号隔开，Java 的基本语法为：

```
访问控制字符 其他状态修饰符 返回值类型 方法名(参数列表) throws
        Exception1,Exception2,… {
    //语句块;
}
```

需要注意的是：

（1）throws 后罗列的异常类型必须可以覆盖所有可能抛出异常对象。这里“可能抛出”还包括方法体主动处理异常的漏网之鱼，即 catch 语句段无法匹配处理的异常类型。

（2）对于所有可能抛出的异常类型，用户可以通过简单罗列的方式或写父类异常类覆盖若干子类异常的方式编写，但一定要全面覆盖方法体中可能出现的异常对象涉及的种类。一个极端的写法就是 throws Exception，但这样的“覆盖”无助于后继程序识别异常对象的种类，因此标准 Java 代码一般会选择在 throws 语句后罗列所有的异常类。

（3）throws 中委托抛出的异常会由最精确的子类类型异常类型变量所持有，与 throws 后面登记的异常类型顺序无关。例如：

```
public void f() throws IOException, FileNotFoundException{ … }
```

FileNotFoundException 继承自 IOException，若 f()方法中出现 FileNotFoundException 类型的异常，则该异常由 FileNotFoundException 类型变量持有，而不是 IOException 类型变量，尽管 IOException 写在前面。

异常处理将程序员从庞大的异常维护 if 分支中解放出来，这也意味着 Java 的异常处理具有 if 分支的逻辑功能，在 Java 中通过 throws 关键字声明方法可能抛出的异常，这些异常对象的出现就可以理解为一个激活分支的信号，让方法调用者可以以信号为契机做一些分支操作。只不过通过 throws 语法实现的分支跨越了方法。这种向外 throws 异常对象行为，像是一种令人鄙视的甩锅操作，然而，这种行为也为用户提供了解决问题的契机。这种将解决问题的契机交给用户自行处理是一种开放的编程态度，也是 Java 中异常处理的核心设计理念，因此，我们在 Java 的 API 中经常可以看到方法会抛出异常。

使用这种方法本身不处理异常的方案，可以极大地提高程序设计的自由度。例如，一个检查密码匹配的方法只专心于“检查用户名与密码是否匹配”这个主要的业务逻辑即可，至于用户名不存在或密码为空等问题被封装为异常对象，由本方法抛出，提醒方法调用者专门处理用户名不存在和密码为空两种情况。前面除 0 问题使用 throws 技术，委托调用方法处理异常的代码如代码 5-4 所示。

```
//代码 5-4  委托处理除 0 异常
public class TestThrows {              //委托调用方法处理异常
    public static void computer() throws ArithmeticException{
```

```
        int x = (int) (1 + (Math.random() * 2));
        int y = (int) (Math.random() * 2);
        int result;
        result = x/y;
        System.out.println("the result is:" + result);
    }

    public static void exp(ArithmeticException e){
        System.out.println("ArithmeticException 类的对象出现");
        e.printStackTrace();
    }

    public static void main(String [] args){
        try {                                    //在调用方法中主动处理异常
            computer();
        }catch (ArithmeticException e) {
            exp(e);
        }
    }
}
```

如果 y 的值是 0，那么运行结果如下。

```
ArithmeticException 类的对象出现
java.lang.ArithmeticException:/by zero
at * * * .TestThrows.computer(TestThrows.java:8)
at * * * .TestThrows.main(TestThrows.java:14)
```

在上述代码中，computer()方法本地不处理 ArithmeticException 对象，而是将这个异常委托给调用它的 main()方法。在 main()方法中，程序员可以选择主动处理，如代码 5-3 所示，也可以不处理，将其抛出，委托给调用 main()方法的 JVM，而 JVM 使用默认处理异常对象的方法 printStackTrace()打印异常堆栈。例如，main()方法抛出 ArithmeticException 异常的写法如下。

```
public static void main(String [] args) throws ArithmeticException {
    computer();
}
```

如果 y 的值是 0，那么运行结果如下。

```
Exception in thread "main" java.lang.ArithmeticException: /by zero
at * * * .TestThrows.computer(TestThrows.java:9)
at * * * .TestThrows.main(TestThrows.java:22)
```

与代码 5-1 的错误显示一致。这就是默认的 Java 异常处理 RuntimeException 的方法，从出现 RuntimeException 对象后，逐层上报到 main()方法、JVM，然后打印异常堆栈信息。