第十一章

异常

当程序违反 Java 编程语言的语义约束时，Java 虚拟机会将该错误作为异常通知给该程序。

这种冲突的一个例子是试图在数组边界之外使用索引。一些编程语言及其实现通过强制终止程序来应对此类错误；其他编程语言允许实现以任意或不可预测的方式做出反应。这两种方法都不符合 Java SE 平台的设计目标：提供可移植性和健壮性。

相反，Java 编程语言指定当违反语义约束时将抛出异常，并将导致从异常发生的点到程序员可以指定的点的控制的非本地转移。

异常被认为是从它发生的点引发的，并且被认为是在控制权转移到的点处被捕获的。

程序也可以使用 throw 语句显式抛出异常(第 14.18 节)。

显式使用 throw 语句提供了另一种处理错误条件的老式风格，方法是返回有趣的值，例如整数值-1，在这种情况下，负值通常不会被期望。经验表明，调用者经常忽略或不检查这些滑稽的值，从而导致程序不健壮、表现出不良行为或两者兼而有之。

每个异常都由 Throwable 类或其一个子类的实例表示(§11.1)。这样的对象可用于将信息从发生异常的点传递给捕获该对象的处理程序。处理程序由 try 语句的 catch 子句建立(§14.20)。

在抛出异常的过程中，Java 虚拟机突然一个接一个地完成当前线程中已经开始但尚未完成执行的任何表达式、语句、方法和构造函数调用、初始化器以及字段初始化表达式。这个过程一直持续到找到一个处理程序，该处理程序通过命名异常的类或异常的类的超类来指示它处理该特定异常(§11.2)。如果找不到这样的处理程序，则可以由未捕获的异常处理程序层次结构中的一个处理程序来处理异常(§11.3)-因此会尽一切努力避免让异常不被处理。

Java SE 平台的异常机制与其同步模型(§17.1)集成在一起，因此监视器在同步语句(§14.19)和同步方法的调用(§8.4.3.6、§15.12)突然完成时被解锁。

11.1 异常的种类和原因

11.1.1 异常的种类

异常由 Throwable 类(Object 的直接子类)或其子类之一的实例表示。

Throwable 及其所有子类统称为异常类。

类 Exception 和 Error 是 Throwable 的直接子类:

- Exception 是普通程序可能希望恢复的所有异常的超类。

  类 RounmeException 是 Except 的直接子类。RuntimeException 是所有异常的超类，这些异常可能在表达式求值过程中因多种原因而引发，但仍有可能从中恢复。

  RuntimeException 及其所有子类统称为运行时异常类。

- Error 是普通程序通常不会恢复的所有异常的超类。

  Error 及其所有子类统称为错误类。

未检查异常类是运行时异常类和错误类。

检查异常类是除了未检查异常类之外的所有异常类。也就是说，检查的异常类是 Throwable 及其所有子类，而不是 RuntimeException 及其子类和 Error 及其子类。

> 程序可以在 throw 语句中使用 Java SE 平台 API 的预先存在的异常类，或者根据需要将其他异常类定义为 Throwable 或其任何子类的子类。为了利用异常处理程序的编译时检查(§11.2)，通常将大多数新的异常类定义为已检查的异常类，也就是说，定义为 Exception 的子类，这些子类不是 RuntimeException 的子类。
>
> 类 Error 是 Throwable 的一个单独的子类，与类层次结构中的 Exception 不同，它允许程序使用惯用法"} catch (Exception e) {"(§11.2.3)来捕获所有可能恢复的异常，而不捕获通常无法恢复的错误。
>
> 注意 Throwable 的子类不能是泛型(§8.1.2)。

11.1.2 异常的原因

抛出异常的原因有以下三种:

- 执行 throw 语句(§14.18)。
- Java 虚拟机同步检测到异常的执行条件，即:
  - 表达式的求值违反了 Java 编程语言的正常语义(§15.6)，例如整数被零除。
  - 在加载、链接或初始化部分程序时发生错误(§12.2、§12.3、§12.4);在这种情况下，抛出一个 LinkageError 子类的实例。
  - 内部错误或资源限制阻止 Java 虚拟机实现 Java 编程语言的语义;在这种情况下，抛出 VirtualMachineError 子类的一个实例。

  这些异常不是在程序中的任意点抛出的，而是在它们被指定为表达式求值或语句执行的可能结果时抛出的。
- 发生异步异常 (§11.1.3)。

11.1.3 异步异常

大多数异常都是由于发生异常的线程的操作而同步发生的，并且发生在程序中指定可能导致此类异常的某个点。相比之下，异步异常是一种可能发生在程序执行过程中的任何时刻的异常。

异步异常只会在以下情况下发生:

- 调用类 Thread 或 ThreadGroup 的（废弃的）stop 方法。

  一个线程可以调用(废弃的)stop 方法来影响另一个线程或指定线程组中的所有线程。它们是异步的，因为它们可能发生在其他线程或线程组执行的任何时刻。

- Java 虚拟机中的内部错误或资源限制，使其无法实现 Java 编程语言的语义。在这种情况下，引发的异步异常是 VirtualMachineError 的子类的实例。

  请注意，StackOverflowError 是 VirtualMachineError 的一个子类，它可以通过方法调用(§15.12.4.5)同步抛出，也可以由于 native 方法执行或 Java 虚拟机资源限制而异步抛出。
  类似地，VirtualMachineError 的另一个子类 OutOfMemoyError 可以在类实例创建(§15.9.4，§12.5)、数组创建(§15.10.2，§10.6)、类初始化(§12.4.2)和装箱转换(§5.1.7)期间同步抛出，也可以异步抛出。

Java SE 平台允许在引发异步异常之前进行少量但有限度的执行。

异步异常很少见，但如果要生成高质量的机器代码，就必须正确理解它们的语义。

上面提到的延迟允许优化代码在符合 Java 编程语言语义的情况下检测并抛出这些异常。一个简单的实现可能会在每个控制转移指令点轮询异步异常。由于程序的大小有限，因此在检测异步异常时提供了总延迟的界限。由于控制传输之间不会出现异步异常，因此代码生成器可以灵活地对控制传输之间的计算进行重新排序，以获得更高的性能。论文《*Polling Efficiently on Stock Hardware*》由 Marc Feeley，Proc.1993 年发表于关于函数式程序设计和计算机体系结构的会议，丹麦哥本哈根，179-187 页，建议作为进一步阅读。

11.2 异常的编译时检查

Java 编程语言要求程序包含用于检查异常的处理程序，检查异常可能是由方法或构造函数的执行导致的(§8.4.6、§8.8.5)。这种由于存在异常处理程序的编译时检查旨在减少未正确处理的异常数量。对于作为可能结果的每个已检查异常，方法或构造函数的 throws 子句必须提及该异常的类或该异常类的一个超类（§11.2.3）。

throws 子句中指定的检查异常类（§11.1.1）是方法或构造函数的实现者和用户之间契约的一部分。重写方法的 throws 子句不能指定该方法将导致抛出被重写方法不允许抛出的任何检查异常(§8.4.8.3)。当涉及接口时，一个重写声明可以重写多个方法声明。在这种情况下，重写声明必须有一个与所有重写声明兼容的 throws 子句(§9.4.1)。

未检查的异常类(§11.1.1)不受编译时检查。

错误类被免除，因为它们可能在程序中的许多点发生，并且很难或不可能从它们中恢复。一个声明这种异常的程序将是杂乱无章的，毫无意义的。复杂的程序可能仍希望捕获并尝试从其中一些情况中恢

复。

运行时异常类被免除，因为在 Java 编程语言的设计者看来，必须声明这样的异常不会对确定程序的正确性有很大帮助。Java 编程语言的许多操作和构造都可能在运行时导致异常。Java 编译器可用的信息和编译器执行的分析级别通常不足以确定不会发生这种运行时异常，即使这对程序员来说可能是显而易见的。要求声明这样的异常类只会让程序员感到恼火。

例如，某些代码可能实现了一个循环数据结构，通过构造，它永远不会涉及空引用；然后，程序员可以确定不会发生 NullPointerException 异常，但 Java 编译器很难证明这一点。建立这种数据结构的全局属性所需的定理证明技术超出了本规范的范围。

我们说，如果根据§11.2.1 和§11.2.2 中的规则，语句或表达式可能抛出 E 类异常，则该语句或表达式可以引发 E 类异常。

我们说 catch 子句可以捕获其可捕获的异常类：

- 单一 catch 子句的可捕获异常类是其异常参数的声明类型(§14.20)。
- 多 catch 子句的可捕获异常类是联合中的替代类，用于指示其异常参数的类型。

11.2.1 表达式异常分析

类实例创建表达式(§15.9)可以抛出异常类 E 当且仅当以下有一个条件为真：

- 该表达式是限定类实例创建表达式，并且限定表达式可以抛出 E; 或者
- 参数列表的某些表达式可能会抛出 E; 或者
- E 是所选构造函数的调用类型的异常类型之一(§15.12.2.6)；或者
- 类实例创建表达式包含一个 ClassBody，ClassBody 中的某个实例初始化器或实例变量初始化器可以抛出 E。

方法调用表达式 (§15.12)可以抛出异常类 E 当且仅当以下之一为真：

- 方法调用表达式的形式为 Primary . [TypeArguments] Identifier 并且 Primary 表达式可以抛出 E; 或者
- 参数列表的某些表达式可能会抛出 E; 或者
- E 是所选方法的调用类型的异常类型之一 (§15.12.2.6)。

lambda 表达式(§15.27)不能抛出异常类。

switch 表达式(§15.28)可能引发异常类 E 当且仅当以下之一为真:

- 选择器表达式可以抛出 E; 或者
- switch 块中的某些 switch 规则表达式、switch 规则块、switch 规则抛出语句或标记为 switch 的语句组可以抛出 E。

对于任何其他类型的表达式，该表达式都可以引发异常类 E 当且仅当它的一个直接子表达式可以抛出 E。

注意方法引用表达式(§15.13)形式为 Primary :: [TypeArguments] Identifier 可以抛出异常类如果 Primary 子表达式可以抛出异常类。相反，lambda 表达式不能抛出任何东西，并且没有可用于执行异常分析的直接子表达式。它是 lambda 表达式的主体，包含可以抛出异常类的表达式和语句。

11.2.2 语句的异常分析

被抛出的表达式具有静态类型 E 并且不是 final 的或实际 final 的异常参数的 throw 语句(§14.18)可以抛出 E 或被抛出的表达式可以抛出的任何异常类。

例如，语句 throw new java.io.FileNotFoundException();只能抛出 java.io.FileNotFoundException。从形式上讲，它并不是“可以抛出”java.io.FileNotFoundException 的子类或超类。

抛出的表达式是 catch 子句 C 的 final 或实际 final 的异常参数的 thrown 表达式可以抛出异常类 E 当且仅当：

- E 是声明 C 的 try 语句的 try 块可以引发的异常类；以及
- E 与 C 的任何可捕获异常类赋值兼容；以及
- E 与同一 try 语句中 C 的左侧声明的 catch 子句的任何可捕获异常类的赋值不兼容。

try 语句(§14.20)可以抛出异常类 E 当且仅当以下之一为真:

- try 块可以抛出 E，或者用于初始化资源的表达式(在 try-with-resources 语句中)可以抛出 E，或者自动调用资源的 close()方法 (在 try-with-resources 语句中)可以抛出 E，并且 E 与 try 语句的任何 catch 子句的任何可捕获异常类不赋值兼容，并且不存在 finally 块或 finally 块可以正常完成；或者
- try 语句的某些 catch 块可能抛出 E，或者不存在 finally 块，或者 finally 块可以正常完成；或者
- 出现了 finally 块，可以抛出 E。

显式构造函数调用语句(§8.8.7.1)可能引发异常类 E 当且仅当以下之一为真:

- 构造函数调用的参数列表的某些表达式可能抛出 E; 或者
- E 被确定为被调用的构造函数的 throws 子句的异常类(§15.12.2.6)。

switch 语句(§14.11)可以引发异常类 E 当且仅当以下之一为真:

- 选择器表达式可以抛出 E; 或者
- switch 块中的某些 switch 规则表达式、switch 规则块、switch 规则抛出语句或标记为 switch 的语句组可以抛出 E。

任何其他语句 S 都可以引发异常类 E 当且仅当直接包含在 S 中的表达式或语句可以抛出 E。

11.2.3 异常检查

如果方法或构造函数体可以抛出某个异常类 E，而 E 是一个检查的异常类，并且 E 不是该

方法或构造函数的 throws 子句中声明的某个类的子类，则这是编译时错误。

如果当 E 是检查异常类，并且 E 不是 lambda 表达式所针对的函数类型的 throws 子句中声明的某个类的子类时，lambda 主体可以抛出某个异常类，则这是编译时错误。

如果命名类或接口的类变量初始化器(§8.3.2)或静态初始化器(§8.7)可以抛出检查的异常类，则为编译时错误。

如果命名类的实例变量初始化器(§8.3.2)或实例初始化器(§8.6)可以抛出检查的异常类，则它是编译时错误，除非命名类至少有一个显式声明的构造函数，并且在每个构造函数的 throws 子句中显式声明了异常类或其超类之一。

> 请注意，如果匿名类的实例变量初始化器或实例初始化器(§15.9.5)可以抛出异常类，则不会出现编译时错误。在命名类中，程序员负责传播有关初始化器可以引发哪些异常类的信息，方法是在任何显式构造函数声明上声明适当的 throws 子句。类的初始化器抛出的检查异常类和类的构造函数声明的检查异常类之间的这种关系对于匿名类声明是有保证的，因为没有显式的构造函数声明是可能的，并且 Java 编译器总是基于其初始化器可以抛出的检查异常类为匿名类声明生成一个带有合适的 throws 子句的构造函数。

如果 catch 子句可以捕获已检查异常类 E_1，则这是编译时错误，而与 catch 子句对应的 try 块不能抛出作为 E_1 的子类或超类的已检查异常类，除非 E_1 是异常或异常的超类。

如果 catch 子句可以捕获异常类 E_1，而直接封闭的 try 语句的前一个 catch 子句可以捕获 E_1 或 E_1 的超类，则是编译时错误。

> 如果 catch 子句可以捕获已检查异常类 E_1，并且与 catch 子句对应的 try 块可以抛出已检查异常类 E_2，其中 $E_2 <: E_1$，并且直接封闭的 try 语句的前一个 catch 子句可以捕获已检查异常类 E_3，其中 $E_2 <: E_3 <: E_1$，则鼓励 Java 编译器发出警告。

例子 11.2.3-1. 捕获检查异常

```
import java.io.FileNotFoundException;
import java.io.IOException;

class StaticallyThrownExceptionsIncludeSubtypes {
    public static void main(String[] args) { try {
            throw new FileNotFoundException();
        } catch (IOException ioe) {
            // "catch IOException" catches IOException
            // and any subtype.
        }

        try {
            throw new FileNotFoundException();
              // Statement "can throw" FileNotFoundException.
              // It is not the case that statement "can throw"
              // a subtype or supertype of FileNotFoundException.
        } catch (FileNotFoundException fnfe) {
            // ... Handle exception ...
        } catch (IOException ioe) {
            // Legal, but compilers are encouraged to give
            // warnings as of Java SE 7, because all subtypes of
            // IOException that the try block "can throw" have
```

```
            // already been caught by the prior catch clause.
        }

        try {
            m();
              // m's declaration says "throws IOException", so
              // m "can throw" IOException. It is not the case
              // that m "can throw" a subtype or supertype of
              // IOException (e.g. Exception).
        } catch (FileNotFoundException fnfe) {
            // Legal, because the dynamic type of the exception
            // might be FileNotFoundException.
        } catch (IOException ioe) {
            // Legal, because the dynamic type of the exception
            // might be a different subtype of IOException.
        } catch (Throwable t) {
            // Can always catch Throwable.
        }
    }

    static void m() throws IOException { throw new
        FileNotFoundException();
    }
}
```

根据上面的规则，多 catch 子句中的每个备选方案(§14.20)必须能够捕获由 try 块抛出的某个异常类，和之前的 catch 子句没有捕获的异常类。例如，下面的第二个 catch 子句将导致编译时错误，因为异常分析确定 SubclassOfFoo 已被第一个 catch 语句捕获：

```
try { ... }
catch (Foo f) { ... }
catch (Bar | SubclassOfFoo e) { ... }
```


11.3 运行时异常处理

当抛出异常时(§14.18)，控制权从导致异常的代码转移到可以处理该异常的 try 语句(§14.20)中最近的动态封闭 catch 子句(如果有的话)。

如果某个语句或表达式出现在 catch 子句所属的 try 语句的 try 块内，或者该语句或表达式的调用方被 catch 子句动态包围，则该语句或表达式将被 catch 子句动态包围。

语句或表达式的调用方取决于它发生的位置：

- 如果在方法内，则调用方是为调用该方法而执行的方法调用表达式(§15.12)。
- 如果在构造函数、实例初始化器或实例变量的初始化器中，则调用方是类实例创建表达式(§15.9)或为了创建对象而执行的 newInstance 的方法调用。
- 如果在静态初始化器或静态变量的初始化器中，则调用方是使用类或接口以使其被初始化的表达式(§12.4)。

通过将抛出的对象的类与 catch 子句的可捕获异常类进行比较来确定特定的 catch 子句是

否可以处理异常。如果 catch 子句的某个可捕获异常类是该异常的类或该异常的类的超类，则 catch 子句可以处理该异常。

> 同样，catch 子句将捕获作为其可捕获异常类之一的 instanceof(§15.20.2)的任何异常对象。

抛出异常时发生的控制转移导致表达式(§15.6)和语句(§14.1)突然完成，直到遇到可以处理该异常的 catch 子句；然后通过执行该 catch 子句的块来继续执行。导致异常的代码永远不会恢复。

所有异常(同步和异步)都是精确的：当控制转移发生时，在抛出异常的点之前执行的语句和计算的表达式的所有影响必须看起来已经发生。在引发异常的点之后出现的任何表达式、语句或其部分可能看起来都没有计算过。

> 如果优化代码推测性地执行了异常发生点之后的某些表达式或语句，则必须准备好对程序的用户可见状态隐藏这种推测性执行。

如果找不到可以处理异常的 catch 子句，则终止当前线程(遇到异常的线程)。在终止之前，所有 finally 子句都被执行，未捕获的异常按照以下规则处理：

- 如果当前线程设置了未捕获的异常处理程序，则执行该处理程序。
- 否则，将为作为当前线程的父级的 ThreadGroup 调用方法 uncaughtException。 如果 ThreadGroup 及其父 ThreadGroups 没有重写 uncaughtException，则调用默认处理程序的 uncaughtException 方法。

> 在需要确保一个代码块总是在另一个代码块之后执行的情况下，即使另一个代码块突然完成，也可以使用带有 finally 子句的 try 语句(§14.20.2)。
>
> 如果一个 try 或 catch 块在一个 try-finally 或 try-catch-finally 语句里突然完成，那么 finally 子句在异常传播的过程中执行，即使最后没有发现匹配的 catch 子句。
>
> 如果由于 try 块的突然完成而执行 finally 子句，并且 finally 子句本身突然完成，则会丢弃 try 块突然完成的原因，并从那里传播突然完成的新原因。
>
> 在§14(块、语句和模式)和§15(表达式)(特别是§15.6)中的每条语句的规范中详细规定了突然完成和捕获异常的确切规则。
>
> **例子 11.3-1. 抛出和捕获异常**
>
> 下面的程序声明了异常类 TestException。Test 类的 main 方法调用了 thrower 方法四次，导致异常在四次中的三次被引发。方法 main 中的 try 语句捕获 thrower 抛出的每个异常。无论 thrower 的调用是正常完成还是突然完成，都会打印一条消息来描述发生的事情。

```
class TestException extends Exception {
  TestException() { super(); }
  TestException(String s) { super(s); }
}

class Test {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                thrower(arg);
```

```
                System.out.println("Test \"" + arg +
                                   "\" didn't throw an exception");
            } catch (Exception e) {
                System.out.println("Test \"" + arg +
                                   "\" threw a " + e.getClass() +
                                   "\n with message: " + e.getMessage());
            }
        }
    }
    static int thrower(String s) throws TestException {
      try {
            if (s.equals("divide")) {
                int i = 0;
                return i/i;
            }
            if (s.equals("null")) {
                s = null;
                return s.length();
            }
            if (s.equals("test")) {
                throw new TestException("Test message");
            }
            return 0;
        } finally {
            System.out.println("[thrower(\"" + s + "\") done]");
        }
    }
}
```

如果我们执行该程序，将参数传递给它：

```
divide null not test
```

它产生以下输出：

```
[thrower("divide") done]
Test "divide" threw a class java.lang.ArithmeticException with
message: / by zero [thrower("null") done]
Test "null" threw a class java.lang.NullPointerException with
message: null [thrower("not") done]
Test "not" didn't throw an exception
[thrower("test") done]
Test "test" threw a class TestException
    with message: Test message
```

方法 thrower 的声明必须有一个 throws 子句，因为它可以抛出 TestException 的实例, 这是一个已检查的异常类(§11.1.1)。如果省略 throws 子句，则会发生编译时错误。

请注意，无论是否发生异常，每次调用 thrower 时都会执行 finally 子句，如每次调用时出现的输出“[thrower(...) done]”所示。