[Java异常 2](#_Toc71210223)

[1. 异常的概述 2](#_Toc71210224)

[2. 异常发生的过程 2](#_Toc71210225)

[3. 异常的处理 3](#_Toc71210226)

[4. throw 和 throws 5](#_Toc71210227)

[5. 异常处理原则 5](#_Toc71210228)

[6. 自定义异常 6](#_Toc71210229)

[7. 异常转换 7](#_Toc71210230)

[8. finally 代码块 9](#_Toc71210231)

[9. 覆盖中异常的使用 10](#_Toc71210232)

# Java异常

我们都知道，Java程序执行分为两个过程：

1. 编译：检查语法错误。如果没有语法错误，编译通过，生成.class文件。
2. 运行：加载并执行.class文件，这时如果出现问题，就会抛出异常。

所以我们说的异常，是指发生在运行时期的一些问题。

## 1. 异常的概述

那么每种问题都得有对应的名字吧？然后对这个问题是不是得进行一些描述？比如错误发生在哪行代码、错误大概的原因是什么等等。所以Java就按照面向对象的思想对这些问题进行描述和对象的封装。

比如：int[] arr = new int[1]; arr[1]; ——ArrayIndexOutOfBoundsException

arr = null; arr[0]; ——NullPointerException

arr = new int[1024 \* 1024 \* 100]; ——OutOfMemoryError

上面这几个例子中，我们发现，有些问题是Exception，有些则是Error。这其实是java中对问题的分类。

Java中的问题分为两种：

1. Error：一般是虚拟机（JVM）调用到了系统底层，由系统底层发生的错误，系统底层告诉JVM，JVM再告诉调用者。一般很严重，一旦发生，程序直接中断。
2. Exception：是虚拟机在运行过程当中发生的错误，然后告诉调用者，可以进行针对性的处理。

## 2. 异常发生的过程

无论是Exception还是Error，它们都有共性，比如错误的位置，错误的信息等，所以java把他们抽离出了一个父类，那就是Throwable.

要理解异常的体系，我们还是得先了解异常发生的过程，以ArrayIndexOutOfBoundsException为例：

1. 发生问题时，虚拟机肯定是知道是什么问题的，那么虚拟机就根据这个已知的问题，把它封装成一个对象：new ArrayIndexOutOfBoundsException(1);
2. 封装对象后，得把这个问题告诉调用者啊，那么就要用 throw 关键字抛出：throw new ArrayIndexOutOfBoundsException(1); 此时的调用者是main方法。
3. main方法知道这个异常后，它没有处理方式，所以继续往外抛。抛给调用者JVM。
4. JVM就使用了默认的处理方式，将问题的名称+信息+位置在控制台上显示出来，让调用者看到信息并结束程序。此时的调用者就是用户，因为是用户启动的虚拟机。

## 3. 异常的处理

通过了解异常发生的过程，我们知道了程序运行过程中可能会有问题，举例：

class Operator{

public int divide(int a, int b){

return a/b;

}

}

class Main{

public static void main(String[] args){

Operator op = new Operator();

int value = op.divide(4,0);

SOUT(value);

}

}

比如上述的代码，编译时没有问题，因为没有语法错误，但是运行时会报错：ArthimeticException.这是因为进行除法运算时，不允许除以0。所以虽然你没有写任何异常处理的代码，运行时还是会报错，但是这是一种很不好的方式，因为别人使用你的程序时，你的代码里没有任何提醒，别人自然以为这是个好的程序，但是当他运行这个程序时，却会被强制中断而得不到想要的结果。

所以为了避免这种被强制中断程序的尴尬，我们需要在程序中处理异常，处理异常有两种方式：

1. 声明可能遇到的异常，不对异常进行处理，而是把问题继续抛给调用者，希望调用者进行处理。

比如上面的那个例子，开发人员在编写divide方法时，就应该要想到，除法运算可能会出问题，所以可以在定义方法时就声明可能出现的问题：

class Operator{

public int divide(int a, int b) throws Exception{

return a/b;

}

}

一旦方法声明了异常，那么调用该方法的调用者必须要针对该异常进行处理，当然调用者也可以选择继续抛出这个异常：

class Main{

public static void main(String[] args) throws Exception{

Operator op = new Operator();

int value = op.divide(4,0);

SOUT(value);

}

}

如果上面的main方法没有声明异常的话，编译时就会报错，因为编译过程检查完语法错误后，最后会检查异常是否被处理。

总结：不处理异常而继续抛给调用者的这种方式其实就是在函数上通过throws关键字声明异常，告诉调用者进行处理。

1. 针对问题进行捕获，用try catch 进行捕获：

try{

// 有可能发生异常的代码。

}catch (异常类 变量){

// 捕获异常类对应的问题，可以针对问题进行相应的处理

}finally{

// 一定会被执行的代码

}

还是用上面的代码为例：

class Operator{

public int divide(int a, int b) throws Exception{

return a/b;

}

}

divide方法声明异常，那么调用该方法的方法必须对可能发生的异常进行捕获或者声明，前面我们已经试了继续声明，这里我们试一下用try catch 捕获：

class Main{

public static void main(String[] args) {

Operator op = new Operator();

try{

int value = op.divide(4,0);

SOUT(value);

}catch(Exception err){

SOUT(“发生异常！”);

}

SOUT(“over”); // 如果发生异常，这个语句执行吗？执行的，因为catch代码块已经处理了异常，所以不影响下面代码运行。

}

}

上面我们捕获异常后，在catch代码块中直接打印了SOUT(“发生异常！”);

但是其实我们已经捕获到了Exception err，那么为什么不根据这个err对象的信息进行处理呢？当然这么做之前要先知道err对象都有什么属性和方法。

异常类常见的方法有：

1. getMessage()：获取异常 信息
2. toString()：异常名称+异常信息
3. printStackTrace()：直接打印异常名字+异常信息+异常位置。

这个也是JVM的默认处理方式，就是说如果你不捕获异常，那么最终JVM就会默认用这个方式处理异常。

## 4. throw 和 throws

我们上面讲到的问题，比如divide(4,0)这个问题，最初是由JVM自己自动化包装出的异常对象，他是这样的：ArithmeticException(“/ by zero”);

除了让JVM自动创建异常对象外，我们自己也可以创建异常对象，并在发生问题时抛出，那就是用throw关键字：

class Operator{

public int divide(int a, int b) throws Exception{ //这里的throws是声明异常

if(b == 0){

// 自己创建对象时，里面的错误信息可以自定义

throw new ArithMeticException(“除法除0了！”); // 这里的throw抛出异常对象

}

return a/b;

}

}

那么throw和throws有什么区别呢？

1. 位置不同：

Throws用在函数上，后面跟的是异常类，可以跟多个。

Throw用在函数内，后面跟的是异常对象。

1. 功能不同：

Throws用来声明可能出现的异常，让调用者知道这个函数可能会有问题并预先做出对应的处理。

Throw是抛出具体的问题对象，执行到throw意味着当前函数真的出现问题啦，会直接结束当前函数并跳转到调用者的异常处理代码。

异常体系最大的特点是：

体系中的类及类产生的对象都具有可抛性，可抛性的意思是可以被throw和throws所操作。

这个特点就为我们后面自己定义异常类打下基础。

## 5. 异常处理原则

1. 通常情况下，如果函数内部有throw抛出的异常，那么函数上必须用throws声明，或者用try catch代码块捕获，否则编译失败。

如果该函数用throws声明了异常，那么调用者必须对该异常进行处理：throws继续抛出或者try catch代码块捕获。

1. 特殊情况：

当函数内用throw抛出的异常是RuntimeException或者它的子类时，函数上不用throws进行声明。不声明的目的就是不让调用者进行处理，就要让调用者的程序强制中断，要修改代码。

Exception分为两种：

1. 编译时会检测的异常。

编译时的检测有两步：第一步检测语法问题，第二部检测安全性问题（throw 和 throws）

1. 运行时异常（RuntimeException），编译时不检测。

## 6. 自定义异常

我们都知道java中已经预先定义了异常体系，对一些特定的问题做出了封装，但是有些时候我们需要定义自己特定的异常，这种情况下，我们可以自己封装我们的问题成一个类，然后把它加入到java的异常体系中。如何加入异常体系？必须继承Throwable 类或者它的子类。

举例：我们要实现除法除了不允许除以0，也不允许除以负数。

思路：

1. 除数不能为0的问题java已经定义对应的异常类了，那么我们直接用就可以。但是除数不能为负数java没有对应的异常类，那么我们就要自己定义一个。

2. 自己定义异常类需要继承Throwable 类或者它的子类，我们就要思考继承什么。这里很明显不是底层错误，所以不会是继承Error类，那么它应该是继承Exception类，更深入的思考，编译时需要检查这个问题吗？它的情况应该与除数为0时类似，既然除数为0的情况编译时不检查，那么除数为负数的情况也不用在编译时检查，就应该是运行时异常，那么继承的类是RuntimeException.

package day11\_exception;  
  
  
*/\*\*  
 \* 自定义的异常类  
 \*/*class NegativeNumberException extends RuntimeException {  
  
 public NegativeNumberException() {  
 }  
  
 public NegativeNumberException(String message) {  
 super(message);  
 }  
}  
  
*/\*\*  
 \* 除法功能类  
 \*/*class DivideDemo {  
  
 public int divide(int a, int b) {  
 if (b < 0) {  
 throw new NegativeNumberException("除数不能为负数。");  
 }  
 if (b == 0) {  
 throw new ArithmeticException("除数不能为0");  
 }  
 return a / b;  
 }  
}  
  
*/\*\*  
 \* 主函数类  
 \*/*public class CustomizedExceptionDemo {  
  
 public static void main(String[] args) {  
 DivideDemo divideDemo = new DivideDemo();  
 int result = divideDemo.divide(10, -2);  
 System.*out*.println(result);  
 }  
}

总结：

异常其实就是将问题封装成对象，然后抛给调用者。

如果声明了异常，就需要调用者处理，处理方式有两种：throws继续声明，try catch捕获异常。

什么时候捕获异常？如果功能内部可以解决这个问题，就捕获。

什么时候继续声明？不能解决就继续声明，或者捕获后需要告诉调用者也要声明。

## 7. 异常转换

在自定义异常类时，需要结合实际来暴露异常信息，只暴露调用者能明白能处理的信息。

举例：

Java开发中有个典型的功能类，就是数据处理操作类：

class DBTool{

public String getData(){

// 数据库连接

// 获取数据

//关闭 数据库连接

}

}

上面示例中，获取数据方法很有可能发生问题，那么我们首先想到的是声明异常，让调用者处理。

这里需要注意，你把异常抛给调用者，调用者知道怎么处理DB的异常吗？显然不知道，甚至调用者可能都不知道数据库的存在，所以很明显这里应该是DBTool自己来处理DB的异常:

class DBTool{

public String getData(){

// 数据库连接

try{

// 获取数据

// if no problem, return data

// if has problem, throw new SQLException()

}catch(SQLException e){

// 处理异常

}

//关闭数据库连接

}

}

上面这么写就可以了吗？表面上看是可以了，但是你操作数据库的目的是返回数据给调用者，如果数据库没有发生异常，那么你顺利拿到数据把数据返回给了调用者。

但是一旦发生异常了，你虽然处理了异常但是依然需要给调用者一个结果，这时你没拿到数据，显然没法返回数据，那么就得抛出一个异常让用户知道因为有问题出现没拿到数据。此时你还能抛数据库异常吗？不能，调用者又不知道数据库异常是什么，所以你得抛出一个没拿到数据的异常:

class DBTool{

public String getData() throw NoValueException{

// 数据库连接

try{

// 获取数据

// if no problem, return data

// if has problem, throw new SQLException()

}catch(SQLException e){

// 处理异常

throw new NoValueException(“没拿到数据，原因是：” + e.getMessage());

}

//关闭数据库连接

}

}

上面你用SQLException 的内容组装了一个新的异常：NoValueException();这就是异常转换。这种转换在开发中经常用到。

## 8. finally 代码块

finally 代码块跟在try catch 代码块之后，是一块一定会被执行的代码。

只有一种情况下finally不被执行，那就是它前面有System.exit(0)，这句代码意味着强行退出JVM。

什么代码会写在finally里呢？

比如我们前面的例子，对数据库进行操作后关闭数据库连接，无论异常是否发生，这个数据库连接是必须关闭的。

也就是说，finally代码块主要的作用是，无论是否有异常发生都要对资源进行释放。

注意：

只要是代码里用throw抛出的非运行时的异常，必须被处理，处理有两种方式：1. 在方法上用throws声明。2.用catch捕获。

问：下面这个异常方法上要用throws声明吗？

try{

throw new Exception(); // 这个异常方法上要声明吗？

}finally{

}

答：要的，因为没有catch捕获，如果有catch捕获它，就不用声明了。

问：下面这个异常方法上要用throws声明吗？

try{

throw new Exception(); // 这个异常方法上要声明吗？

} catch(Exception e) {

throw e;

} finally{

}

答：要的，catch捕获了第一个异常，但是同时又抛了一个异常，这个异常没有捕获，所以要声明。

## 9. 覆盖中异常的使用

在异常当中，子类覆盖父类方法时，子类只能抛出父类方法异常类或者它的子类。如果父类方法抛出多个异常，子类只能抛出父类异常的子集。

原则：子类抛出的异常必须要在父类异常的处理控制中。

有一种情况，只能try catch捕获不能用throws声明：

被覆盖的方法没有发生异常，那么子类覆盖时，子类方法中发生了异常只能捕获进行处理，而不能通过throws声明。那么如果子类处理不了呢？可以将编译时异常转换为运行时异常，这样就不用声明，JVM会自动中断程序让调用者修改代码。