**JAVA基础**

# 异常Exception

## Try….catch

执行try块里的业务逻辑代码时出现异常，系统自动生成一个异常对象，该异常对象被提交给Java运行时环境，这个过程被称为抛出（throw）异常。

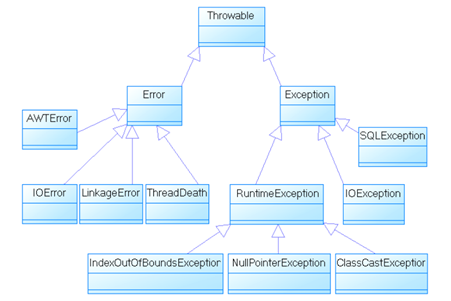
Java运行时环境收到异常对象时，会寻找能处理该异常对象的catch块，如果找到合适的catch块并把该异常对象交给该catch块处理，那这个过程被称为捕获（catch）异常；如果Java运行时环境一直到main方法时都找不到捕获异常的catch块，则运行时环境终止，Java程序也将退出

|  |
| --- |
| **public** **class** HelloException {  **public** **static** **void** main(String[] args) {  HelloException he = **new** HelloException();  he.test1();  }  **public** **void** test1() {  test2();  }  **public** **void** test2() {  test3();  }  /\*\*  \* 当发生异常时，会打印方法调用栈的信息，从发生异常的方法开始 依次打印该方法的调用者，依次类推，若异常没有被处理，  \* 直到main方法，如果main方法也不处理，应用程序将终止退出， 也就是平时所说的应用崩溃（闪退）  \*/  **public** **void** test3() {  **int**[] iArr = **null**;  // nullpointexception  // 这个异常没有处理，执行到此处，便会开始向外抛出异常，其后的  // 代码也不再起作用  // System.out.println(iArr.length);  /\*\*  \* try块中存放可能发生异常的代码 catch用于捕获发生的异常对象，在catch块负责在发生异常时 所需要做的处理工作  \*  \* NullPointerException extends RuntimeException extends Exception  \* 目前为了不必要的麻烦，大家在不知道明确的异常时，都可以使用Exception 如果明确知道可能发生的异常类型，则使用已知的异常对象  \*/  **try** {  // 发生异常，系统自动生成一个异常，  // 然后交给运行时环境，这个过程称为抛出异常  System.***out***.println(iArr.length);  // 运行时环境得到异常后，就寻找能处理该异常的catch块  // 能处理就是catch块中的异常对象能够接收所抛出的异常  // 即若将Exception替换为ClassCastException则异常无法处理  // 因为ClassCastException他无法接收此处抛出NullPointerException  } **catch** (Exception e) {  // 发生异常时，会执行这里的代码，那么这里就应该负责发生异常的后续工作  System.***out***.println(e.getMessage());  System.***out***.println(e.toString());  }  System.***out***.println("123");  }  } |

## 多个异常的捕获

|  |
| --- |
| **public** **class** MultiCatch {  **public** **static** **void** main(String[] args) {  MultiCatch mc = **new** MultiCatch();  mc.test1();  }    **public** **void** test1() {  test2();  }    **public** **void** test2() {  **try** {    String[] strArr = **new** String[2];  // java.lang.ArrayIndexOutOfBoundsException  // extends IndexOutOfBoundsException  // extends RuntimeException extends Exception  System.***out***.println(strArr[3]);    **int**[] iArr = **null**;  // 只要这里发生了异常，便会立刻抛出并寻找catch，try块中  // 其他的代码不在执行  System.***out***.println(iArr.length);  // java.lang.ArithmeticException  // extends RuntimeException extends Exception  **int** i = 5/0;  } **catch** (NullPointerException e) {  System.***out***.println("NullPointerException");  } **catch** (ArithmeticException e) {  // 只要异常在此被捕获，那么其后的catch块就不会在执行  System.***out***.println("ArithmeticException");  } **catch**(Exception e) {  System.***out***.println("Exception");  }  System.***out***.println("只要异常处理了，我就会执行！");  }    /\*\*  \* try-catch 另外一种写法，jdk 1.7及以后  \*/  **public** **void** test3() {  // int[] iArr = null;  **try** {  // 作用域仅在try块对应的大括号内部  **int**[] iArr = **null**;  System.***out***.println(iArr.length);  } **catch** (NullPointerException|ArithmeticException e) {    } **catch**(Exception e) {    }  // iArr = null;  }  } |

## 异常体系



箭头表示继承关系，Error错误，作为程序员我们不需要关心，错误是程序员无法控制的系统或编译器错误，exception是我们学习的重点。

### Try-catch-finally

|  |
| --- |
| **public** **class** FinallyDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  FinallyDemo fd = **new** FinallyDemo();  fd.test1();  }    **public** **void** test1() {  test2();  }    **public** **void** test2() {    /\*\*  \* finally只有一种情况下不会被执行，那就是在他之前调用了  \* System.exit(0)  \* finally一定是在try-catch之后执行  \*  \*/  **try** {  System.***out***.println("try");  // System.exit(0);  **int** i = 5/0;  } **catch**(Exception e) {  System.***out***.println("Exception");  } **finally** {  // 通常用来关闭资源。比如：数据库资源，IO资源，网络连接等，手动释放资源  // 为什么要关闭这些资源，因为他们需要消耗大量的  // （相对于一般的创建对象）内存    // 为了保证一定能回收try块中打开的物理资源，异常处理机制提供了finally  // 块。不管try块中的代码是否出现异常，也不管哪一个catch块被执行，  // finally块总会被执行    System.***out***.println("finally");  }    }  } |

为什么要在finally里关闭资源

finalize（）/ finally的区别

Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之

前做必要的清理工作

**一旦垃圾收集器准备好释放对象占用的存储空间，它首先调用finalize()，而且只有在下一次垃圾收集过程中，才会真正回收对象的内存。**所以如果使用finalize()，就可以在垃圾收集期间进行一些重要的清除或清扫工作**(如关闭流等操作)**。**但JVM(Java虚拟 机)不保证此方法总被调用**。

finalize()在垃圾回收之前被调用是很重要的，当一个对象脱离作用域后，他并不会马上被回收。因此程序员应该提供对象使用的其他的释放系统资源的方法（一般在finally去做）。一**般的编程操作绝对不能依赖finalize（）**。

String str = “aaa”;->脱离作用域将会被回收而不是立马回收-》回收使用gc，在gc之前会调用finalize方法完成清理工作

从str脱离作用域到真正被回收有一段时间空隔

一般文件或数据库操作都会有checked异常，所以finally与try-catch结合使用，保证在finally之前打开的资源一定会执行相应的关闭或释放操作。

## 异常的嵌套

|  |
| --- |
| **public** **class** NestedException {  **public** **static** **void** main(String[] args) {  NestedException ne = **new** NestedException();  ne.test1();  }    **public** **void** test1() {    // 异常的嵌套，三个地方都可以嵌套异常  // 通常没有必要使用超过两层的嵌套异常处理  /\*\*  \* 输出  \* nested  try  catch  finally  \*/    **try**{    **try** {  String[] strArr = **new** String[2];  System.***out***.println(strArr[3]);  } **catch** (Exception e) {  System.***out***.println("nested");  }    System.***out***.println("try");  **int** i = 5/0;        } **catch**(Exception e) {  // try {  //  // } catch (Exception e1) {  //  // }  System.***out***.println("catch");    } **finally** {  // try {  //  // } catch (Exception e1) {  //  // }    System.***out***.println("finally");    }    }  } |

## Checked/Runtime异常

Java的异常被分为两大类：Checked异常和Runtime异常（运行时异常）。所有RuntimeException类及其子类的实例被称为Runtime异常；不是RuntimeException类及其子类的异常实例则被称为Checked异常

什么场景下会发生？应该怎么做(怎么分析)？断点查看变量/从发生异常的位置向上开始查看代码。 如何尽量避免这样的运行时异常？避免就是加判断，条件怎么写？

NullPointException

ArrayIndexOutOfBoundsException

NumberForfatException

ClassCastException

ArithmeticException

#### Check异常

当前方法明确知道如何处理该异常，程序应该使用try...catch块来捕获该异常，然后在对应的catch块中处理该异常，除了捕获还可以throws出去，把它交个调用者处理

|  |
| --- |
| **public** **class** CheckedExceptionDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  *test*();  }    **public** **static** **void** test() {  File file = **new** File("/path");  // 检查型异常不需要我们去关心，编译器会自动提示，让我们去捕获  // 如果不捕获，编译就不会通过,编译不通过，程序一定无法运行  // 所以检查型异常必须处理    **try** {  file.createNewFile();  } **catch** (IOException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    }  } |

## Runtime异常

* Runtime异常则更加灵活，Runtime异常无需显式声明抛出/处理。
* 如果程序需要捕捉Runtime异常，也可以使用try...catch块来捕捉Runtime异常

RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。处理RuntimeException的原则是：如果出现RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。

|  |
| --- |
| **public** **class** RuntimeExceptionDemo {  **public** **static** **void** main(String[] args) {  // **TODO** Auto-generated method stub  *test*();  }    /\*\*  \* 检查型异常在编译期间处理，且必须处理  \* 运行时异常在程序运行的过程中发生的，这样的异常不应该处理  \* 应该将它修复，为什么？  \* 因为这是程序员脑袋糊涂了，把代码写错了，导致了异常的发生  \* 所以可以通过修改代码，解决这样的问题。  \*/    **public** **static** **void** test() {  // 会抛出运行时异常，这个异常不应该使用try-catch 捕获  // 应该修改代码，避免这样的问题发生  // 所以处理这样的问题，逻辑考虑要全面，如以后再处理对象问题上  // 你要注意判空，避免空指针问题，如数组下标越界，需要做下标小于数组  // 长度的判断，避免下标越界问题    **int** j = 0;  **if** (j != 0) {  **int** i = 5/j;  } **else** {  System.***out***.println("除数不能为0");  }    }  } |

遇到Error，程序员一般是无能为力的；遇到RuntimeException，那么一定是程序存在逻辑错误，要对程序进行修改；只有检查型异常在编译期间才是程序员所关心的，程序应该且仅应该抛出或处理检查型异常

## Throws

throws声明抛出异常的思路是：当前方法不知道应该如何这种类型的异常，该异常应该由调用者处理，如果main方法也不知道应该如何处理这种类型的异常，也可以使用throws声明抛出异常，该异常将交给JVM处理。JVM对异常的处理方法是：打印异常跟踪栈信息，并中止程序运行，这就是前面程序在遇到异常后自动结束的原因。

|  |
| --- |
| /\*\*  \* 为什么还需要throws  \* 当前方法（抛出异常的方法）不知道应该如何这种类型的异常，该异常应该由调用者处理  \* 错误处理代码与业务逻辑代码的分离，错误处理不和业务逻辑混淆（解耦 耦合 --相关性）  \*  \* **@author** tzhang  \*  \*/  **public** **class** ThrowsDemo {  /\*\*  \* 如果main方法也不知道应该如何处理这种类型的异常，  \* 也可以使用throws声明抛出异常，该异常将交给JVM处理  \*  \* JVM对异常的处理方法是：打印异常跟踪栈信息，并中止程序运行，  \* 这就是前面程序在遇到异常后自动结束的原因。  \* **@param** args  \* **@throws** IOException  \*/  **public** **static** **void** main(String[] args) **throws** IOException {  *test1*();  }      **public** **static** **void** test1() **throws** IOException {  *test2*();  }    **public** **static** **void** test2() **throws** IOException {  // 检查型异常  File file = **new** File("/path");  file.createNewFile();  }      **public** **static** **void** test3() {  // alt + shift + z 选中try块内容，生成try-catch快捷方式  // try {  *test4*();  // } catch (RuntimeException e) {  // // **TODO** Auto-generated catch block  // e.printStackTrace();  // }  }  /\*\*  \* 运行时异常不会主动提示我们去捕获异常，需要修改代码避免这样的问题发生  \* **@throws** RuntimeException  \*/  **private** **static** **void** test4() **throws** RuntimeException {    **int** i = 5/0;  }    /\*\*  \* 使用throws抛出对个异常时的写法，多个异常之间使用逗号隔开  \* throws后面不要去抛运行时异常，此处我们只是举例讲解多个异常抛出时的写法  \* 实际开发中抛出运行时异常是可以但是不允许这么做（需要取修改代码）。  \* **@throws** NullPointerException  \* **@throws** ClassCastException  \*/  **public** **static** **void** test5() **throws** NullPointerException,ClassCastException {    **int** i = 5/0;  }  } |

## Throw

* 如果需要在程序中自行抛出异常，应使用throw语句，throw语句可以单独使用，throw语句抛出的不是异常类，而是一个异常实例，而且每次只能抛出一个异常实例。throw语句的语法格式如下：
  + throw ExceptionInstance;
* 如果throw语句抛出的异常是Checked异常，则该throw语句要么处于try块里，显式捕获该异常，要么放在一个带throws声明抛出的方法中，即把该异常交给该方法的调用者处理。

|  |
| --- |
| **public** **class** ThrowDemo {  **public** **static** **void** main(String[] args) {  *test3*();  System.***out***.println("123");  }    **public** **void** test1() {    }    **public** **void** test2() **throws** IOException {  File file = **new** File("/path");  **try** {  file.createNewFile();  } **catch** (IOException e) {  // throw抛出异常  **throw** e;  }  }      **public** **static** **void** test3() {    **try** {  System.***out***.println("456");  **int** i = 5/0;  } **catch** (RuntimeException e) {  // throw抛出异常,一旦发生，就会将异常继续上抛  **throw** e;  }  }  } |

### 自定义异常

程序很少会自行抛出系统异常，因为异常的类名通常包含了该异常的有用信息。所以在选择抛出什么异常时，应该选择合适的异常类，从而可以明确地描述该异常情况。在这种情形下，应用程序常常需要抛出自定义异常

用户自定义异常都应该继承Exception基类，如果希望自定义Runtime异常，则应该继承RuntimeException基类。定义异常类时通常需要提供两种构造器：一个是无参数的构造器；另一个是带一个字符串参数的构造器，这个字符串将作为该异常对象的详细说明（也就是异常对象的getMessage方法的返回值）

|  |
| --- |
| **public** **class** SelfException **extends** Exception {  **public** SelfException() {  **super**();  }  **public** SelfException(String message) {  **super**(message);  }  } |
| **public** **class** MainSelfException {  **public** **static** **void** main(String[] args) {  **try** {  *test1*();  } **catch** (SelfException e) {  System.***out***.println(e.getMessage());  }  }    **public** **static** **void** test() **throws** SelfException {    **int** i = 5;  }      **public** **static** **void** test1() **throws** SelfException {    **int** i = 5;  **throw** **new** SelfException("i为5");  }  } |

## 异常跟踪栈

* 异常对象的printStackTrace方法用于打印异常的跟踪栈信息，根据printStackTrace方法的输出结果，我们可以找到异常的源头，并跟踪到异常一路触发的过程。
* 面向对象的应用程序运行时，经常会发生一系列方法调用，从而形成“方法调用栈”，异常的传播则与相反：只要异常没有被完全捕获（包括异常没有被捕获，或异常被处理后重新抛出了新异常），异常从发生异常的方法逐渐向外传播，首先传给该方法的调用者，该方法调用者再次传给其调用者……直至最后传到 main方法，如果main方法依然没有处理该异常，JVM会中止该程序，并打印异常的跟踪栈信息。

|  |
| --- |
| **public** **class** TransferExce {  **public** **static** **void** main(String[] args) {  TransferExce te = **new** TransferExce();  te.test1();  System.***out***.println("123");  }  **public** **void** test1() {  test2();  }  **public** **void** test2() {  **try** {  // 异常一旦真正处理之后，便不会在继续上抛，  // 如此处的NullPointerException处理不了test3中抛出的异常  // 此时异常还会继续向上抛，抛给方法的调用者  test3();  } **catch** (NullPointerException e) {    }    System.***out***.println("456");  }  **private** **void** test3() {  System.***out***.println("789");  **int** i = 5/0;  }  } |

#### 练习

编写一个Cirle类，包括构造方法，求圆的面积和周长，如果圆的半径小于等于0则抛出异常，让用户必须处理，半径必须大于0才可以

|  |
| --- |
| **public** **class** CrMinusZeroException **extends** Exception{  **public** CrMinusZeroException() {  **super**();  }  **public** CrMinusZeroException(String message) {  **super**(message);  }    } |
| **public** **class** MainCircle {  **static** Scanner *scanner* = **new** Scanner(System.***in***);  **static** **double** *radius*;      **public** **static** **void** main(String[] args) {  String value = *scanner*.nextLine();  *radius* = Double.*parseDouble*(value);    **try** {  *dealRadius*();  } **catch** (CrMinusZeroException e) {  System.***out***.println(e.getMessage());  }  }      **public** **static** **void** dealRadius() **throws** CrMinusZeroException {  **if** (*radius* <= 0) {  **throw** **new** CrMinusZeroException("半径小于等于0，输入有误");  } **else** {  *area*();  *perimeter*();  }  }    // String.format("%.2f", double)    **public** **static** **void** area() {    String a = String.*format*("%.2f", Math.***PI*** \* Math.*pow*(*radius*, 2));    System.***out***.println(a);  }    // 保留2位小数，并四舍五入  **public** **static** **void** perimeter() {  String p = String.*format*("%.3f", 2 \* Math.***PI*** \* *radius*);  System.***out***.println(p);  **double** val = Double.*parseDouble*(p) \*100;  **double** need = Math.*round*(val) / 100.0;  System.***out***.println(need );  }  } |

# 泛型

|  |
| --- |
| **public** **class** Student {  **private** String sno;  **private** **int** age;  **public** Student() {  **super**();  // **TODO** Auto-generated constructor stub  }  **public** Student(String sno, **int** age) {  **super**();  **this**.sno = sno;  **this**.age = age;  }  **public** String getSno() {  **return** sno;  }  **public** **void** setSno(String sno) {  **this**.sno = sno;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  **return** "Student [sno=" + sno + ", age=" + age + "]";  }  } |
| /\*\*  \* 带有泛型的类，  \* 普通类里有的东西他都有，  \* 除了存在一个<T>之外，就可以把它看成一个普通类  \*  \* T就像是一种引用数据类型，只是我们给他一个具体的类型，  \* 所以我们称之为泛型  \*  \* 在定义类时指定泛型，就是<T>，此时采用什么字母，那么在当前类中  \* 出现这个泛型的话，都应该使用这个字母  \* **@author** tzhang  \*  \* **@param** <T>  \*/  **public** **class** GenericClass<T> {  **private** T a;  **private** String name;  **public** GenericClass(){    }    **public** GenericClass(T a, String name) {  **super**();  **this**.a = a;  **this**.name = name;  }  **public** T getA() {  **return** a;  }  **public** **void** setA(T a) {  **this**.a = a;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "GenericClass [a=" + a + ", name=" + name + "]";  }  } |
| /\*\*  \* 泛型:  \* 在定义类时，没有指定具体的数据类型，使用一个<字母>代替某种引用类型数据  \* 在使用时我们再去指定他的类型  \*  \* 泛型只需要看得懂，不需要去写  \* **@author** tzhang  \*/  **public** **class** GenericDemo {  **public** **static** **void** main(String[] args) {  // 这样写的目的，是将在GenericClass中出现的所有的T都替换成String类型  GenericClass<String> gc = **new** GenericClass<String>("aaa","add");  System.***out***.println(gc.toString());    // 这样写的目的，是将在GenericClass中出现的所有的T都替换成Student类型  Student stu = **new** Student("123",20);  GenericClass<Student> gc1 = **new** GenericClass<Student>(stu,"张三");  System.***out***.println(gc1.toString());    *swap*("t1","t2");  *swap*(1,2);  *swap*(1.1,2.2);  }    **public** **static** <T> **void** swap(T t1, T t2) {  }    } |
| |  | | --- | | **public** **class** Generic2<T,S> {  **private** T t;  **private** S s;  } | |