# [JAVA异常与异常处理详解](https://www.cnblogs.com/knightsu/p/7114914.html)

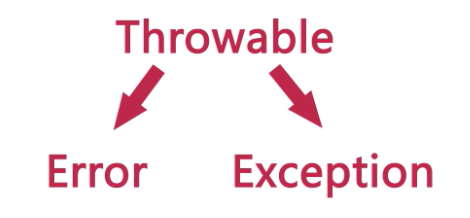
## 一、异常简介

什么是异常？

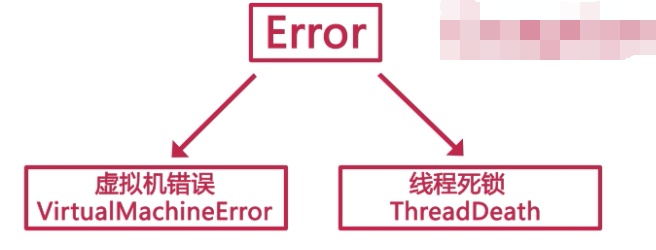
异常就是有异于常态，和正常情况不一样，有错误出错。在java中，阻止当前方法或作用域的情况，称之为异常。

java中异常的体系是怎么样的呢？

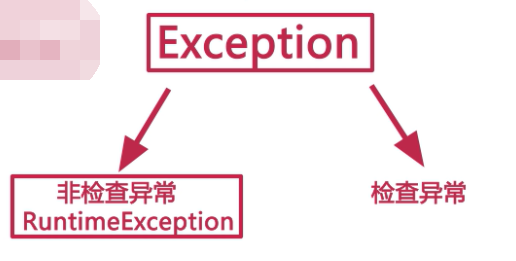
1.Java中的所有不正常类都继承于Throwable类。Throwable主要包括两个大类，一个是Error类，另一个是Exception类；



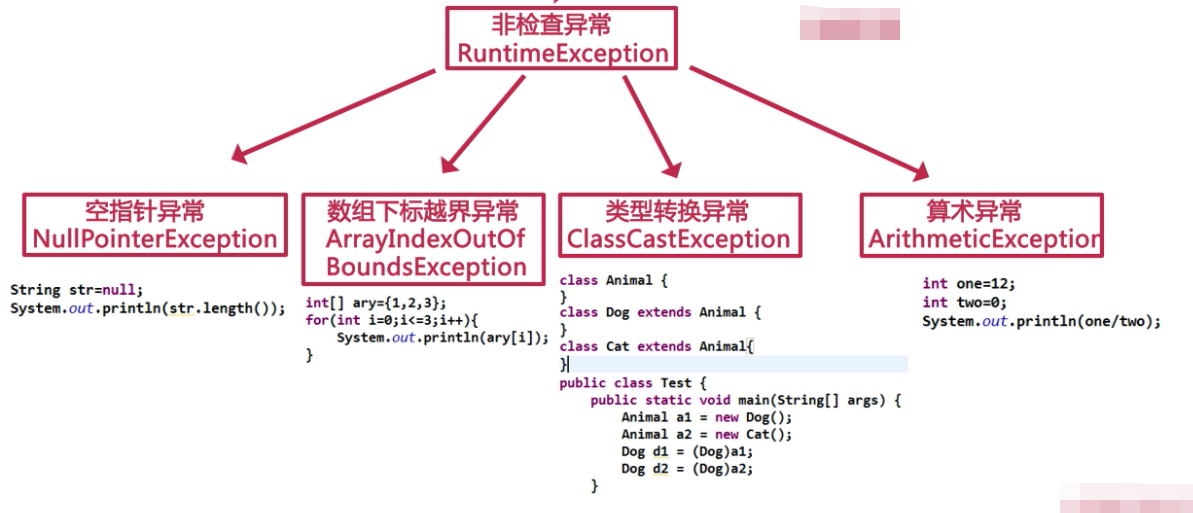
 2.其中Error类中包括虚拟机错误和线程死锁，一旦Error出现了，程序就彻底的挂了，被称为程序终结者；



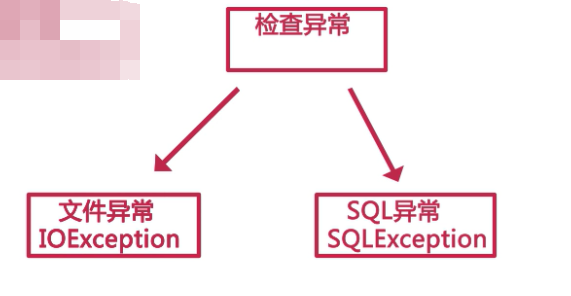
3.Exception类，也就是通常所说的“异常”。主要指编码、环境、用户操作输入出现问题，Exception主要包括两大类，非检查异常（RuntimeException）和检查异常（其他的一些异常）



4.RuntimeException异常主要包括以下四种异常（其实还有很多其他异常，这里不一一列出）：空指针异常、数组下标越界异常、类型转换异常、算术异常。RuntimeException异常会由java虚拟机自动抛出并自动捕获**（就算我们没写异常捕获语句运行时也会抛出错误！！）**，此类异常的出现绝大数情况是代码本身有问题应该从逻辑上去解决并改进代码。



5.检查异常，引起该异常的原因多种多样，比如说文件不存在、或者是连接错误等等。跟它的“兄弟”RuntimeException运行异常不同，**该异常我们必须手动在代码里添加捕获语句来处理该异常**，这也是我们学习java异常语句中主要处理的异常对象。



## 二、try-catch-finally语句

（1）try块：负责捕获异常，一旦try中发现异常，程序的控制权将被移交给catch块中的异常处理程序。

**【try语句块不可以独立存在，必须与 catch 或者 finally 块同存】**

（2）catch块：如何处理？比如发出警告：提示、检查配置、网络连接，记录错误等。**执行完catch块之后程序跳出catch块，继续执行后面的代码。**

　   【**编写catch块的注意事项：多个catch块处理的异常类，要按照先catch子类后catch父类的处理方式，因为会【就近处理】异常（由上自下）。**】

（3）finally：最终执行的代码，用于关闭和释放资源。

=======================================================================

语法格式如下：

try{

//一些会抛出的异常

}catch（Exception e）{

//第一个catch

//处理该异常的代码块

}catch（Exception e）{

//第二个catch，可以有多个catch

//处理该异常的代码块

}finally{

//最终要执行的代码

}

当异常出现时，程序将终止执行，交由异常处理程序（抛出提醒或记录日志等），异常代码块外代码正常执行。 try会抛出很多种类型的异常，**由多个catch块捕获多钟错误**。

多重异常处理代码块顺序问题：**先子类再父类**（顺序不对编译器会提醒错误），finally语句块处理最终将要执行的代码。

=======================================================================

接下来，我们用实例来巩固try-catch语句吧~

先看例子：

package com.hysum.test;

public class TryCatchTest {

/\*\*

\* divider:除数

\* result:结果

\* try-catch捕获while循环

\* 每次循环，divider减一，result=result+100/divider

\* 如果：捕获异常，打印输出“异常抛出了”，返回-1

\* 否则：返回result

\* @return

\*/

public int test1(){

int divider=10;

int result=100;

try{

while(divider>-1){

divider--;

result=result+100/divider;

}

return result;

}catch(Exception e){

e.printStackTrace();

System.out.println("异常抛出了！！");

return -1;

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

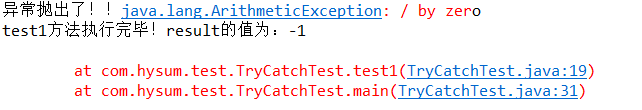
TryCatchTest t1=new TryCatchTest();

System.out.println("test1方法执行完毕！result的值为："+t1.test1());

}

}

运行结果：



结果分析：结果中的红色字抛出的异常信息是由e.printStackTrace()来输出的，它说明了这里我们抛出的异常类型是算数异常，后面还跟着原因：by zero（由0造成的算数异常），下面两行at表明了造成此异常的代码具体位置。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

在上面例子中再加上一个test2()方法来测试finally语句的执行状况：

/\*\*

\* divider:除数

\* result:结果

\* try-catch捕获while循环

\* 每次循环，divider减一，result=result+100/divider

\* 如果：捕获异常，打印输出“异常抛出了”，返回result=999

\* 否则：返回result

\* finally：打印输出“这是finally，哈哈哈！！”同时打印输出result

\* @return

\*/

public int test2(){

int divider=10;

int result=100;

try{

while(divider>-1){

divider--;

result=result+100/divider;

}

return result;

}catch(Exception e){

e.printStackTrace();

System.out.println("异常抛出了！！");

return result=999;

}finally{

System.out.println("这是finally，哈哈哈！！");

System.out.println("result的值为："+result);

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

TryCatchTest t1=new TryCatchTest();

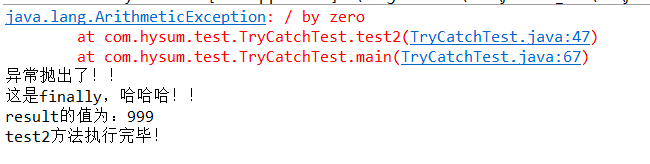
//System.out.println("test1方法执行完毕！result的值为："+t1.test1());

t1.test2();

System.out.println("test2方法执行完毕！");

}

运行结果：



结果分析：我们可以从结果看出，finally语句块是在try块和catch块语句执行之后最后执行的。finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以函数返回值是在finally执行前确定的；

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

这里有个有趣的问题，如果把上述中的test2方法中的finally语句块中加上return，编译器就会提示警告：finally block does not complete normally

public int test2(){

int divider=10;

int result=100;

try{

while(divider>-1){

divider--;

result=result+100/divider;

}

return result;

}catch(Exception e){

e.printStackTrace();

System.out.println("异常抛出了！！");

return result=999;

}finally{

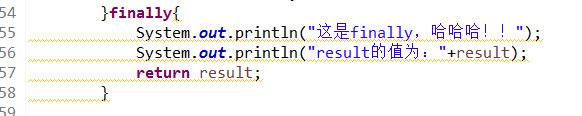
System.out.println("这是finally，哈哈哈！！");

System.out.println("result的值为："+result);

return result;//编译器警告

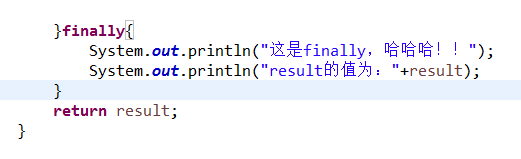
}

}



分析问题: finally块中的return语句可能会覆盖try块、catch块中的return语句;如果finally块中包含了return语句，即使前面的catch块重新抛出了异常，则调用该方法的语句也不会获得catch块重新抛出的异常，而是会得到finally块的返回值，并且不会捕获异常。

解决问题：面对上述情况，其实更合理的做法是，**既不在try block内部中使用return语句，也不在finally内部使用 return语句，而应该在 finally 语句之后使用return来表示函数的结束和返回**。如：



>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

**总结：**

　　1、不管有木有出现异常或者try和catch中有返回值return，finally块中代码都会执行；

　　2、finally中最好不要包含return，否则程序会提前退出，返回会覆盖try或catch中保存的返回值。

　　3.  e.printStackTrace()可以输出异常信息。

　　4.  return值为-1为抛出异常的习惯写法。

　　5.  如果方法中try,catch,finally中没有返回语句，则会调用这三个语句块之外的return结果。

　　6.  finally 在try中的return之后 在返回主调函数之前执行。

## 三、throw和throws关键字

java中的异常抛出通常使用throw和throws关键字来实现。

**throw ----将产生的异常抛出，是抛出异常的一个动作**。

一般会用于程序出现某种逻辑时程序员主动抛出某种特定类型的异常。如：  
　　语法：throw (异常对象)，如：

public static void main(String[] args) {

String s = "abc";

if(s.equals("abc")) {

throw new NumberFormatException();

} else {

System.out.println(s);

}

//function();

}

运行结果：

Exception in thread "main" java.lang.NumberFormatException

at test.ExceptionTest.main(ExceptionTest.java:67)

**throws----声明将要抛出何种类型的异常（声明）。**

语法格式：

1 public void 方法名（参数列表）

2 throws 异常列表{

3 //调用会抛出异常的方法或者：

4 throw new Exception（）；

5 }

当某个方法可能会抛出某种异常时用于throws 声明可能抛出的异常，然后交给上层调用它的方法程序处理。如：

public static void function() throws NumberFormatException{

String s = "abc";

System.out.println(Double.parseDouble(s));

}

public static void main(String[] args) {

try {

function();

} catch (NumberFormatException e) {

System.err.println("非数据类型不能转换。");

//e.printStackTrace();

}

}

**throw与throws的比较**1、throws出现在方法函数头；而throw出现在函数体。  
2、throws表示出现异常的一种可能性，并不一定会发生这些异常；throw则是抛出了异常，执行throw则一定抛出了某种异常对象。  
3、两者都是消极处理异常的方式（这里的消极并不是说这种方式不好），只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

来看个例子：

throws e1,e2,e3只是告诉程序这个方法可能会抛出这些异常，方法的调用者可能要处理这些异常，而这些异常e1，e2，e3可能是该函数体产生的。  
throw则是明确了这个地方要抛出这个异常。如：

void doA(int a) throws (Exception1,Exception2,Exception3){

try{

......

}catch(Exception1 e){

throw e;

}catch(Exception2 e){

System.out.println("出错了！");

}

if(a!=b)

throw new Exception3("自定义异常");

}

分析:  
1.代码块中可能会产生3个异常，(Exception1,Exception2,Exception3)。  
2.如果产生Exception1异常，则捕获之后再抛出，由该方法的调用者去处理。  
3.如果产生Exception2异常，则该方法自己处理了（即System.out.println("出错了！");）。所以该方法就不会再向外抛出Exception2异常了，void doA() throws Exception1,Exception3 里面的**Exception2也就不用写了。因为已经用try-catch语句捕获并处理了。**  
4.Exception3异常是该方法的某段逻辑出错，程序员自己做了处理，在该段逻辑错误的情况下抛出异常Exception3，则该方法的调用者也要处理此异常。这里用到了自定义异常，该异常下面会由解释。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

使用throw和throws关键字**需要注意**以下几点：

1.throws的异常列表可以是抛出一条异常，也可以是抛出多条异常，每个类型的异常中间用逗号隔开

2.方法体中调用会抛出异常的方法或者是先抛出一个异常：用throw new Exception（） throw写在方法体里，表示“抛出异常”这个动作。

3.如果某个方法调用了抛出异常的方法，那么必须添加try catch语句去尝试捕获这种异常， 或者添加声明，将异常抛出给更上一层的调用者进行处理

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

**自定义异常**

为什么要使用自定义异常，有什么**好处**？

1.我们在工作的时候，项目是分模块或者分功能开发的 ,基本不会你一个人开发一整个项目，使用自定义异常类就**统一了对外异常展示的方式**。

2.有时候我们遇到某些校验或者问题时，**需要直接结束掉当前的请求**，这时便可以通过抛出自定义异常来结束，如果你项目中使用了SpringMVC比较新的版本的话有控制器增强，可以通过@ControllerAdvice注解写一个控制器增强类来拦截自定义的异常并响应给前端相应的信息。

3.自定义异常可以在我们项目中某些**特殊的业务逻辑**时抛出异常，比如"中性".equals(sex)，性别等于中性时我们要抛出异常，而Java是不会有这种异常的。系统中有些错误是符合Java语法的，但不符合我们项目的业务逻辑。

4.使用自定义异常继承相关的异常来抛出处理后的异常信息**可以隐藏底层的异常，这样更安全，异常信息也更加的直观。**自定义异常可以抛出我们自己想要抛出的信息，可以通过抛出的信息区分异常发生的位置，根据异常名我们就可以知道哪里有异常，根据异常提示信息进行程序修改。比如空指针异常NullPointException，我们可以抛出信息为“xxx为空”定位异常位置，而不用输出堆栈信息。

说完了为什么要使用自定义异常，有什么好处，我们再来看看自定义异常的**毛病**：

毋庸置疑，我们不可能期待JVM（Java虚拟机）自动抛出一个自定义异常，也不能够期待JVM会自动处理一个自定义异常。发现异常、抛出异常以及处理异常的工作必须靠编程人员在代码中利用异常处理机制自己完成。这样就相应的增加了一些开发成本和工作量，所以项目没必要的话，也不一定非得要用上自定义异常，要能够自己去权衡。

最后，我们来看看怎么**使用**自定义异常：

在 Java 中你可以自定义异常。编写自己的异常类时需要记住下面的几点。

* 所有异常都必须是 Throwable 的子类。
* 如果希望写一个检查性异常类，则需要继承 Exception 类。
* 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

class MyException extends Exception{ }

我们来看一个实例：

package com.hysum.test;

public class MyException extends Exception {

/\*\*

\* 错误编码

\*/

private String errorCode;

public MyException(){}

/\*\*

\* 构造一个基本异常.

\*

\* @param message

\* 信息描述

\*/

public MyException(String message)

{

super(message);

}

public String getErrorCode() {

return errorCode;

}

public void setErrorCode(String errorCode) {

this.errorCode = errorCode;

}

}

使用自定义异常抛出异常信息：

package com.hysum.test;

public class Main {

public static void main(String[] args) {

// TODO Auto-generated method stub

String[] sexs = {"男性","女性","中性"};

for(int i = 0; i < sexs.length; i++){

if("中性".equals(sexs[i])){

try {

throw new MyException("不存在中性的人！");

} catch (MyException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}else{

System.out.println(sexs[i]);

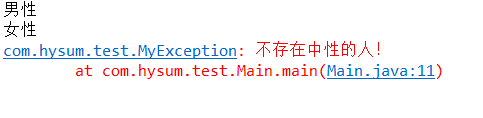
}

}

}

}

运行结果：



 就是这么简单，可以根据实际业务需求去抛出相应的自定义异常。

## 四、java中的异常链

异常需要封装，但是仅仅封装还是不够的，还需要**传递异常**。

异常链是一种面向对象编程技术，指将捕获的异常**包装**进一个**新的异常**中并**重新抛出**的异常处理方式。原异常被保存为新异常的一个属性（比如cause）。这样做的意义是一个方法应该抛出定义在相同的抽象层次上的异常，但不会丢弃更低层次的信息。

我可以这样理解异常链：

把捕获的异常包装成新的异常，在新异常里添加原始的异常，并将新异常抛出，它们就像是链式反应一样，一个导致（cause）另一个。这样在最后的顶层抛出的异常信息就包括了最底层的异常信息。

### 》场景

比如我们的JEE项目一般都又三层：持久层、逻辑层、展现层，持久层负责与[数据库](http://lib.csdn.net/base/mysql)交互，逻辑层负责业务逻辑的实现，展现层负责UI数据的处理。

有这样一个模块：用户第一次访问的时候，需要持久层从user.xml中读取数据，如果该文件不存在则提示用户创建之，那问题就来了：如果我们直接把持久层的异常FileNotFoundException抛弃掉，逻辑层根本无从得知发生任何事情，也就不能为展现层提供一个友好的处理结果，最终倒霉的就是展现层：没有办法提供异常信息，只能告诉用户“出错了，我也不知道出了什么错了”—毫无友好性而言。

正确的做法是先封装，然后传递，过程如下：

1.把FileNotFoundException封装为MyException。

2.抛出到逻辑层，逻辑层根据异常代码（或者自定义的异常类型）确定后续处理逻辑，然后抛出到展现层。

3.展现层自行确定展现什么，如果管理员则可以展现低层级的异常，如果是普通用户则展示封装后的异常。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

### 》示例

package com.hysum.test;

public class Main {

public void test1() throws RuntimeException{

String[] sexs = {"男性","女性","中性"};

for(int i = 0; i < sexs.length; i++){

if("中性".equals(sexs[i])){

try {

throw new MyException("不存在中性的人！");

} catch (MyException e) {

// TODO Auto-generated catch block

e.printStackTrace();

RuntimeException rte=new RuntimeException(e);//包装成RuntimeException异常

//rte.initCause(e);

throw rte;//抛出包装后的新的异常

}

}else{

System.out.println(sexs[i]);

}

}

}

public static void main(String[] args) {

// TODO Auto-generated method stub

Main m =new Main();

try{

m.test1();

}catch (Exception e){

e.printStackTrace();

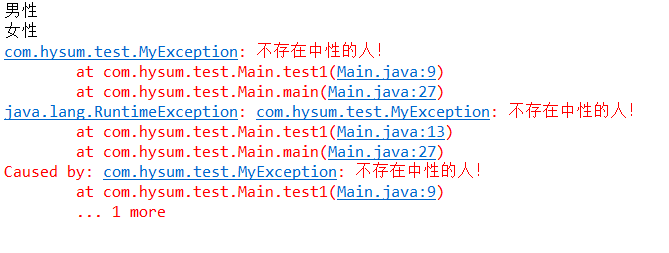
e.getCause();//获得原始异常

}

}

}

运行结果：



结果分析：我们可以看到控制台先是输出了原始异常，这是由e.getCause()输出的；然后输出了e.printStackTrace()，在这里可以看到Caused by：原始异常和e.getCause()输出的一致。这样就是形成一个异常链。**initCause()的作用是包装原始的异常，当想要知道底层发生了什么异常的时候调用getCause()就能获得原始异常**。

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

### 》建议

异常需要封装和传递，我们在进行系统开发的时候，不要“吞噬”异常，也不要“赤裸裸”的抛出异常，封装后在抛出，或者通过异常链传递，可以达到系统更健壮、友好的目的。

## 五、结束语

java的异常处理的知识点杂而且理解起来也有点困难，我在这里给大家总结了以下几点使用java异常处理的时候，**良好的编码习惯：**

1、处理运行时异常时，采用逻辑去合理规避同时辅助try-catch处理

2、在多重catch块后面，可以加一个catch（Exception）来处理可能会被遗漏的异常

3、对于不确定的代码，也可以加上try-catch，处理潜在的异常

4、尽量去处理异常，切记只是简单的调用printStackTrace（）去打印

5、具体如何处理异常，要根据不同的业务需求和异常类型去决定

6、尽量添加finally语句块去释放占用的资源