java 中 try 与 catch 的使用

作者：huihui
原文链接：https://ld246.com/article/1496716457930
来源网站：链滴

转自 CSDN
try{
//代码区
}catch(Exception e){
//异常处理
}
代码区如果有错误，就会返回所写异常的处理。
首先要清楚，如果没有 try 的话，出现异常会导致程序崩溃。
而 try 则可以保证程序的正常运行下去，比如说：
try{
int i = 1/0;
}catch(Exception e){
.....
}
一个计算的话，如果除数为 0，则会报错，如果没有 try 的话，程序直接崩溃。用 try 的话，则可以程序运行下去，并且输出为什么出错！
try catch 是捕捉 try 部分的异常，当你没有 trycatch 的时候，如果出现异常则程序报错，加上 tycatch，出现异常程序正常运行，只是把错误信息存储到 Exception 里，所以 catch 是用来提取异信息的，你可以在 Catch 部分加上一句 System.out.println(e.ToString());，如果出现异常可以把异打印出来
Java 的异常处理机制(try...catch...finally)
1 引子
try...catch...finally 恐怕是大家再熟悉不过的语句了，而且感觉用起来也是很简单，逻辑上似乎也是容易理解。不过，我亲自体验的“教训”告诉我，这个东西可不是想象中的那么简单、听话。不信？你看看下面的代码，“猜猜”它执行后的结果会是什么？不要往后看答案、也不许执行代码看真正答哦。如果你的答案是正确，那么这篇文章你就不用浪费时间看啦。
public class TestException
{
public TestException()
{
}
boolean testEx() throws Exception
{
boolean ret = true;
try
{
ret = testEx1();
}
catch (Exception e)
{
System.out.println("testEx, catch exception");
ret = false;
throw e;
}
finally
{
System.out.println("testEx, finally; return value=" + ret);
return ret;
}
}

```
boolean testEx1() throws Exception<br>
{<br>
boolean ret = true;<br>
try<br>
{<br>
ret = testEx2();<br>
if (!ret)<br>
{<br>
return false;<br>
}<br>
System.out.println("testEx1, at the end of try");<br>
return ret;<br>
}<br>
catch (Exception e)<br>
{<br>
System.out.println("testEx1, catch exception");<br>
ret = false;<br>
throw e;<br>
}<br>
finally<br>
{<br>
System.out.println("testEx1, finally; return value=" + ret);<br>
return ret;<br>
}<br>
}<br>
boolean testEx2() throws Exception<br>
{<br>
boolean ret = true;<br>
try<br>
{<br>
int b = 12;<br>
int c;<br>
for (int i = 2; i &gt;= -2; i--)<br>
{<br>
c = b / i;<br>
System.out.println("i=" + i);<br>
}<br>
return true;<br>
}<br>
catch (Exception e)<br>
{<br>
System.out.println("testEx2, catch exception");<br>
ret = false;<br>
throw e;<br>
}<br>
finally<br>
{<br>
System.out.println("testEx2, finally; return value=" + ret);<br>
return ret;<br>
}<br>
}<br>
public static void main(String[] args)<br>
{<br>
TestException testException1 = new TestException();<br>
```

try
{
testException1.testEx();
}
catch (Exception e)
{
e.printStackTrace();
}
}
}
你的答案是什么？是下面的答案吗？
i=2
i=1
testEx2, catch exception
testEx2, finally; return value=false
testEx1, catch exception
testEx1, finally; return value=false
testEx, catch exception
testEx, finally; return value=false
如果你的答案真的如上面所说，那么你错啦。^_^，那就建议你仔细看一看这篇文章或者拿上面的代
按各种不同的情况修改、执行、<a href="https://ld246.com/forward?goto=http%3A%2F%2Flib.
sdn.net%2Fbase%2Fsoftwaretest" title="软件测试知识库" target="_blank" rel="nofollow ugc"
测试，你会发现有很多事情不是原来想象中的那么简单的。
现在公布正确答案：
i=2
i=1
testEx2, catch exception
testEx2, finally; return value=false
testEx1, finally; return value=false
testEx, finally; return value=false
2 基础知识
2.1 相关概念
例外是在程序运行过程中发生的异常事件，比如除 0 溢出、数组越界、文件找不到等，这些事件的发
将阻止程序的正常运行。为了加强程序的鲁棒性，程序设计时，必须考虑到可能发生的异常事件并做
相应的处理。<a href="https://ld246.com/forward?goto=http%3A%2F%2Flib.csdn.net%2Fbas
%2Fc" title="C语言知识库" target="_blank" rel="nofollow ugc">C 语言中，通过使用 if 语
来判断是否出现了例外，同时，调用函数通过被调用函数的返回值感知在被调用函数中产生的例外事
并进行处理。全程变量 ErroNo 常常用来反映一个异常事件的类型。但是，这种错误处理机制会导致
少问题。
Java 通过面向对象的方法来处理例外。在一个方法的运行过程中，如果发生了例外，则这个方法生成
表该例外的一个对象，并把它交给运行时系统，运行时系统寻找相应的代码来处理这一例外。我们把
成例外对象并把它提交给运行时系统的过程称为抛弃(throw)一个例外。运行时系统在方法的调用栈
查找，从生成例外的方法开始进行回朔，直到找到包含相应例外处理的方法为止，这一个过程称为捕获
catch)一个例外。
2.2 Throwable 类及其子类
　用面向对象的方法处理例外，就必须建立类的层次。类 Throwable 位于这一类层次的最顶层，只
它的后代才可以做为一个例外被抛弃。图 1 表示了例外处理的类层次。
从图中可以看出，类 Throwable 有两个直接子类：Error 和 Exception。Error 类对象（如动态连接
误等），由 Java 虚拟机生成并抛弃（通常，Java 程序不对这类例外进行处理）；Exception 类对象是
Java 程序处理或抛弃的对象。它有各种不同的子类分别对应于不同类型的例外。其中类 RuntimeExc
ption 代表运行时由 Java 虚拟机生成的例外，如算术运算例外 ArithmeticException(由除 0 错等导致
、数组越界例外 ArrayIndexOutOfBoundsException 等；其它则为非运行时例外，如输入输出例外
OException 等。Java 编译器要求 Java 程序必须捕获或声明所有的非运行时例外，但对运行时例外
以不做处理。

2.3　异常处理关键字
Java 的异常处理是通过 5 个关键字来实现的：try，catch，throw，throws，finally。JB 的在线帮中对这几个关键字是这样解释的：
Throws: Lists the exceptions a method could throw.
Throw: Transfers control of the method to the exception handler.
Try: Opening exception-handling statement.
Catch: Captures the exception.
Finally: Runs its code before terminating the program.
2.3.1 try 语句　
try 语句用大括号{}指定了一段代码，该段代码可能会抛弃一个或多个例外。
2.3.2 catch 语句　
catch 语句的参数类似于方法的声明，包括一个例外类型和一个例外对象。例外类型必须为 Throwabl 类的子类，它指明了 catch 语句所处理的例外类型，例外对象则由运行时系统在 try 所指定的代码块生成并被捕获，大括号中包含对象的处理，其中可以调用对象的方法。
catch 语句可以有多个，分别处理不同类的例外。Java 运行时系统从上到下分别对每个 catch 语句处的例外类型进行检测，直到找到类型相匹配的 catch 语句为止。这里，类型匹配指 catch 所处理的例类型与生成的例外对象的类型完全一致或者是它的父类，因此，catch 语句的排列顺序应该是从特殊一般。
也可以用一个 catch 语句处理多个例外类型，这时它的例外类型参数应该是这多个例外类型的父类，序设计中要根据具体的情况来选择 catch 语句的例外处理类型。　
2.3.3 finally 语句　
try 所限定的代码中，当抛弃一个例外时，其后的代码不会被执行。通过 finally 语句可以指定一块代。无论 try 所指定的程序块中抛弃或不抛弃例外，也无论 catch 语句的例外类型是否与所抛弃的例外类型一致，finally 所指定的代码都要被执行，它提供了统一的出口。通常在 finally 语句中可以进行源的清除工作。如关闭打开的文件等。
2.3.4 throws 语句　
throws 总是出现在一个函数头中，用来标明该成员函数可能抛出的各种异常。对大多数 Exception 类来说，Java 编译器会强迫你声明在一个成员函数中抛出的异常的类型。如果异常的类型是 Error 或 untimeException， 或它们的子类，这个规则不起作用， 因为这在程序的正常部分中是不期待出现。 如果你想明确地抛出一个 RuntimeException，你必须用 throws 语句来声明它的类型。
2.3.5 throw 语句　
throw 总是出现在函数体中，用来抛出一个异常。程序会在 throw 语句后立即终止，它后面的语句行不到，然后在包含它的所有 try 块中（可能在上层调用函数中）从里向外寻找含有与其匹配的 catch 子句的 try 块。
3 关键字及其中语句流程详解
3.1 try 的嵌套
你可以在一个成员函数调用的外面写一个 try 语句，在这个成员函数内部，写另一个 try 语句保护其代码。每当遇到一个 try 语句，异常的框架就放到堆栈上面，直到所有的 try 语句都完成。如果下一的 try 语句没有对某种异常进行处理，堆栈就会展开，直到遇到有处理这种异常的 try 语句。下面是个 try 语句嵌套的例子。
class MultiNest {
static void procedure() {
try {
int a = 0;
int b = 42/a;
} catch(java.lang.ArithmeticException e) {
System.out.println("in procedure, catch ArithmeticException: " + e);
}
}
public static void main(String args[]) {
try {
procedure();
} catch(java.lang. Exception e) {
System.out.println("in main, catch Exception: " + e);

}
}
}
这个例子执行的结果为：
in procedure, catch ArithmeticException: java.lang.ArithmeticException: / by zero
成员函数 procedure 里有自己的 try/catch 控制，所以 main 不用去处理 ArrayIndexOutOfBounds xception；当然如果如同最开始我们做测试的例子一样，在 procedure 中 catch 到异常时使用 throw e;语句将异常抛出，那么 main 当然还是能够捕捉并处理这个 procedure 抛出来的异常。例如在 proc dure 函数的 catch 中的 System.out 语句后面增加 throw e;语句之后，执行结果就变为：
in procedure, catch ArithmeticException: java.lang.ArithmeticException: / by zero
in main, catch Exception: java.lang.ArithmeticException: / by zero
3.2 try-catch 程序块的执行流程以及执行结果
相对于 try-catch-finally 程序块而言，try-catch 的执行流程以及执行结果还是比较简单的。
首先执行的是 try 语句块中的语句，这时可能会有以下三种情况：
1．如果 try 块中所有语句正常执行完毕，那么就不会有其他的“动做”被执行，整个 try-catch 程序正常完成。
2．如果 try 语句块在执行过程中碰到异常 V，这时又分为两种情况进行处理：
--> 如果异常 V 能够被与 try 相应的 catch 块 catch 到，那么第一个 catch 到这个异常的 catch（也是离 try 最近的一个与异常 V 匹配的 catch 块）将被执行；如果 catch 块执行正常，那么 try-cath 程序块的结果就是“正常完成”；如果该 catch 块由于原因 R 突然中止，那么 try-catch 程序块的果就是“由于原因 R 突然中止（completes abruptly）”。
--> 如果异常 V 没有 catch 块与之匹配，那么这个 try-catch 程序块的结果就是“由于抛出异常 V 而突然中止（completes abruptly）”。
3． 如果 try 由于其他原因 R 突然中止（completes abruptly），那么这个 try-catch 程序块的结果是“由于原因 R 突然中止（completes abruptly）”。
3.3 try-catch-finally 程序块的执行流程以及执行结果
try-catch-finally 程序块的执行流程以及执行结果比较复杂。
首先执行的是 try 语句块中的语句，这时可能会有以下三种情况：
1．如果 try 块中所有语句正常执行完毕，那么 finally 块的居于就会被执行，这时分为以下两种情况
--> 如果 finally 块执行顺利，那么整个 try-catch-finally 程序块正常完成。
--> 如果 finally 块由于原因 R 突然中止，那么 try-catch-finally 程序块的结局是“由于原因 R 突中止（completes abruptly）”
2．如果 try 语句块在执行过程中碰到异常 V，这时又分为两种情况进行处理：
--> 如果异常 V 能够被与 try 相应的 catch 块 catch 到，那么第一个 catch 到这个异常的 catch（也是离 try 最近的一个与异常 V 匹配的 catch 块）将被执行；这时就会有两种执行结果：
--> 如果 catch 块执行正常，那么 finally 块将会被执行，这时分为两种情况：
--> 如果 finally 块执行顺利，那么整个 try-catch-finally 程序块正常完成。
--> 如果 finally 块由于原因 R 突然中止，那么 try-catch-finally 程序块的结局是“由于原因 R 突中止（completes abruptly）”
--> 如果 catch 块由于原因 R 突然中止，那么 finally 模块将被执行，分为两种情况：
--> 如果如果 finally 块执行顺利，那么整个 try-catch-finally 程序块的结局是“由于原因 R 突然止（completes abruptly）”。
--> 如果 finally 块由于原因 S 突然中止，那么整个 try-catch-finally 程序块的结局是“由于原因 S 突然中止（completes abruptly）”，原因 R 将被抛弃。
（注意，这里就正好和我们的例子相符合，虽然我们在 testEx2 中使用 throw e 抛出了异常，但是由 testEx2 中有 finally 块，而 finally 块的执行结果是 complete abruptly 的（别小看这个用得最多的 eturn，它也是一种导致 complete abruptly 的原因之一啊——后文中有关于导致 complete abruptl 的原因分析），所以整个 try-catch-finally 程序块的结果是“complete abruptly”，所以在 testEx1 中调用 testEx2 时是捕捉不到 testEx1 中抛出的那个异常的，而只能将 finally 中的 return 结果获取。
如果在你的代码中期望通过捕捉被调用的下级函数的异常来给定返回值，那么一定要注意你所调用的级函数中的 finally 语句，它有可能会使你 throw 出来的异常并不能真正被上级调用函数可见的。当这种情况是可以避免的，以 testEx2 为例：如果你一定要使用 finally 而且又要将 catch 中 throw 的 e

在 testEx1 中被捕获到，那么你去掉 testEx2 中的 finally 中的 return 就可以了。
这个事情已经在 OMC2.0 的 MIB 中出现过啦：服务器的异常不能完全被反馈到客户端。）
--> 如果异常 V 没有 catch 块与之匹配，那么 finally 模块将被执行，分为两种情况：
--> 如果 finally 块执行顺利，那么整个 try-catch-finally 程序块的结局就是“由于抛出异常 V 而然中止（completes abruptly）”。
--> 如果 finally 块由于原因 S 突然中止，那么整个 try-catch-finally 程序块的结局是“由于原因 S 突然中止（completes abruptly）”，异常 V 将被抛弃。
3．如果 try 由于其他原因 R 突然中止（completes abruptly），那么 finally 块被执行，分为两种情：
--> 如果 finally 块执行顺利，那么整个 try-catch-finally 程序块的结局是“由于原因 R 突然中止 completes abruptly）”。
--> 如果 finally 块由于原因 S 突然中止，那么整个 try-catch-finally 程序块的结局是“由于原因 S 突然中止（completes abruptly）”，原因 R 将被抛弃。
3.4 try-catch-finally 程序块中的 return
从上面的 try-catch-finally 程序块的执行流程以及执行结果一节中可以看出无论 try 或 catch 中发生什么情况，finally 都是会被执行的，那么写在 try 或者 catch 中的 return 语句也就不会真正的从该数中跳出了，它的作用在这种情况下就变成了将控制权（语句流程）转到 finally 块中；这种情况下定要注意返回值的处理。
例如，在 try 或者 catch 中 return false 了，而在 finally 中又 return true，那么这种情况下不要期你的 try 或者 catch 中的 return false 的返回值 false 被上级调用函数获取到，上级调用函数能够获到的只是 finally 中的返回值，因为 try 或者 catch 中的 return 语句只是转移控制权的作用。
3.5 如何抛出异常
如果你知道你写的某个函数有可能抛出异常，而你又不想在这个函数中对异常进行处理，只是想把它出去让调用这个函数的上级调用函数进行处理，那么有两种方式可供选择：
第一种方式：直接在函数头中 throws SomeException，函数体中不需要 try/catch。比如将最开始例子中的 testEx2 改为下面的方式，那么 testEx1 就能捕捉到 testEx2 抛出的异常了。
boolean testEx2() throws Exception{
boolean ret = true;
int b=12;
int c;
for (int i=2;i>=-2;i--){
c=b/i;
System.out.println("i="+i);
}
return true;
}
第二种方式：使用 try/catch，在 catch 中进行一定的处理之后（如果有必要的话）抛出某种异常。如上面的 testEx2 改为下面的方式，testEx1 也能捕获到它抛出的异常：
boolean testEx2() throws Exception{
boolean ret = true;
try{
int b=12;
int c;
for (int i=2;i>=-2;i--){
c=b/i;
System.out.println("i="+i);
}
return true;
}catch (Exception e){
System.out.println("testEx2, catch exception");
Throw e;
}
}
第三种方法：使用 try/catch/finally，在 catch 中进行一定的处理之后（如果有必要的话）抛出某种

常。例如上面的 testEx2 改为下面的方式，testEx1 也能捕获到它抛出的异常：
boolean testEx2() throws Exception{
boolean ret = true;
try{
int b=12;
int c;
for (int i=2;i>=-2;i--){
c=b/i;
System.out.println("i="+i);
throw new Exception("aaa");
}
return true;
}catch (java.lang.ArithmeticException e){
System.out.println("testEx2, catch exception");
ret = false;
throw new Exception("aaa");
}finally{
System.out.println("testEx2, finally; return value="+ret);
}
}
4 关于 abrupt completion
前面提到了 complete abruptly（暂且理解为“突然中止”或者“异常结束”吧），它主要包含了两大的情形：abrupt completion of expressions and statements，下面就分两种情况进行解释。<br

4.1 Normal and Abrupt Completion of Evaluation
每一个表达式（expression）都有一种使得其包含的计算得以一步步进行的正常模式，如果每一步计都被执行且没有异常抛出，那么就称这个表达式“正常结束（complete normally）”；如果这个表式的计算抛出了异常，就称为“异常结束（complete abruptly）”。异常结束通常有一个相关联的因（associated reason），通常也就是抛出一个异常 V。
与表达式、操作符相关的运行期异常有：
-->A class instance creation expression, array creation expression , or string concatenation operatior expression throws an OutOfMemoryError if there is insufficient memory available.<r>
-->An array creation expression throws a NegativeArraySizeException if the value of any dmension expression is less than zero.
-->A field access throws a NullPointerException if the value of the object reference expresson is null.
-->A method invocation expression that invokes an instance method throws a NullPointerxception if the target reference is null.
-->An array access throws a NullPointerException if the value of the array reference expresion is null.
-->An array access throws an ArrayIndexOutOfBoundsException if the value of the array inex expression is negative or greater than or equal to the length of the array.
-->A cast throws a ClassCastException if a cast is found to be impermissible at run time.<b>
-->An integer division or integer remainder operator throws an ArithmeticException if thealue of the right-hand operand expression is zero.
-->An assignment to an array component of reference type throws an ArrayStoreException when the value to be assigned is not compatible with the component type of the array.
4.2 Normal and Abrupt Completion of Statements
正常情况我们就不多说了，在这里主要是列出了 abrupt completion 的几种情况：
-->break, continue, and return 语句将导致控制权的转换，从而使得 statements 不能正常地、整地执行。
--> 某些表达式的计算也可能从 java 虚拟机抛出异常，这些表达式在上一小节中已经总结过了；

个显式的的 throw 语句也将导致异常的抛出。抛出异常也是导致控制权的转换的原因（或者说是阻止 tatement 正常结束的原因）。
如果上述事件发生了，那么这些 statement 就有可能使得其正常情况下应该都执行的语句不能完全执行到，那么这些 statement 也就是被称为是 complete abruptly.
导致 abrupt completion 的几种原因：
-->A break with no label
-->A break with a given label
-->A continue with no label
-->A continue with a given label
-->A return with no value
-->A return with a given value A
-->throw with a given value, including exceptions thrown by the Java virtual machine
5 关于我们的编程的一点建议
弄清楚 try-catch-finally 的执行情况后我们才能正确使用它。
如果我们使用的是 try-catch-finally 语句块，而我们又需要保证有异常时能够抛出异常，那么在 finall 语句中就不要使用 return 语句了（finally 语句块的最重要的作用应该是释放申请的资源），因为 finaly 中的 return 语句会导致我们的 throw e 被抛弃，在这个 try-catch-finally 的外面将只能看到 finall 中的返回值（除非在 finally 中抛出异常）。（我们需要记住：不仅 throw 语句是 abrupt completio 的原因，return、break、continue 等这些看起来很正常的语句也是导致 abrupt completion 的原。）</p>