Java 异常的处理和设计及深入理解

一、 异常的定义

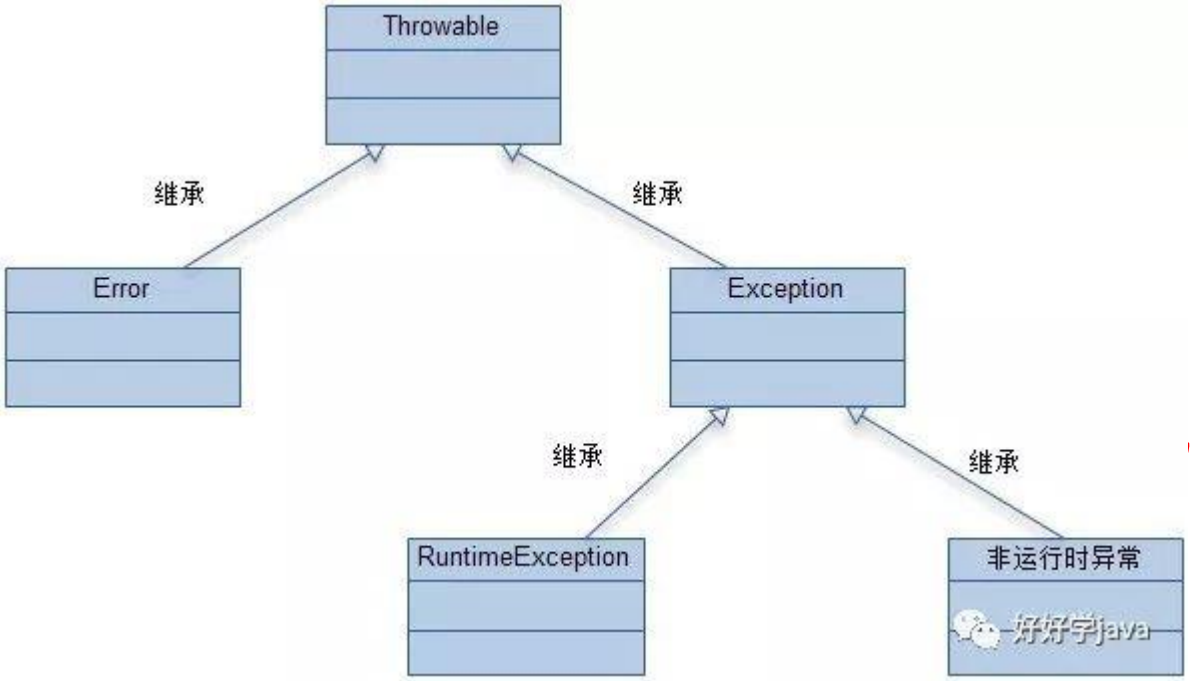
在《java 编程思想》中这样定义 异常：阻止当前方法或作用域继续执行的问题。虽然 java 中有异常处理机制，但是要明确一点，决不应该用"正常"的态度来看待异常。绝对一点说异常就是某种意义上的错误，就是问题，它可能会导致程序失败。之所以 java 要提出异常处理机制，就是要告诉开发人员，你的程序出现了不正常的情况，请注意。

```
public class Calculator {
    public int devide(int num1, int num2) {
        //判断除数是否为 0
        if(num2 == 0) {
            throw new IllegalArgumentException("除数不能为零");
        }
        return num1/num2;
    }
}
```

看一下这个类中关于除运算的方法，如果你是新手你可能会直接返回计算结果，根本不去考虑什么参数是否正确，是否合法（当然可以原谅，谁都是这样过来的）。但是我们应尽可能的考虑周全，把可能导致程序失败的"苗头"扼杀在摇篮中，所以进行参数的合法性检查就很有必要了。其中执行参数检查抛出来的那个参数非法异常，这就属于这个方法的不正常情况。正常情况下我们会正确的使用计算器，但是不排除粗心大意把除数赋值为 0。如果你之前没有考虑到这种情况，并且恰巧用户数学基础不好，那么你完了。但是如果你之前考虑到了这种情况，那么很显然错误已在你的掌控之中。

二、异常扫盲行动

首先来熟悉一下 java 的异常体系：



Throwable 类是 Java 语言中所有错误或异常的超类（这就是一切皆可抛的东西）。它有两个子类：Error 和 Exception。

Error：用于指示合理的应用程序不应该试图捕获的严重问题。这种情况是很大的问题，大到你不能处理了，所以听之任之就行了，你不用管它。比如说 VirtualMachineError：当 Java 虚拟机崩溃或用尽了它继续操作所需的资源时，抛出该错误。好吧，就算这个异常的存在了，那么应该何时，如何处理它呢？？交给 JVM 吧，没有比它更专业的了。

Exception：它指出了合理的应用程序想要捕获的条件。Exception 又分为两类：一种是 CheckedException，一种是 UncheckedException。这两种 Exception 的区别主要是 CheckedException 需要用 try...catch...显示的捕获，而 UncheckedException 不需要捕获。通常 UncheckedException 又叫做 RuntimeException。《effective java》指出：对于可恢复的条件使用被检查的异常（CheckedException），对于程序错误（言外之意不可恢复，大错已经酿成）使用运行时异常（RuntimeException）。

我们常见的 RuntimeExcepiton 有 IllegalArgumentException、IllegalStateException、NullPointerException、IndexOutOfBoundsException 等等。对于那些 CheckedException 就不胜枚举了，我们在编写程序过程中 try...catch...捕捉的异常都是 CheckedException。io 包中的 IOException 及其子类，这些都是 CheckedException。

三、 java 中异常如何处理

在 Java 中如果需要处理异常，必须先对异常进行捕获，然后再对异常情况进行处理。如何对可能发生异常的代码进行异常捕获和处理呢？使用 try 和 catch 关键字即可，如下面一段代码所示：

```
try {
```

```
  File file = new File("d:/a.txt");
  if(!file.exists())
    file.createNewFile();
} catch (IOException e) {
  // TODO: handle exception
}
```

被 try 块包围的代码说明这段代码可能会发生异常，一旦发生异常，异常便会被 catch 捕获到，然后需要在 catch 块中进行异常处理。

这是一种处理异常的方式。在 Java 中还提供了另一种异常处理方式即抛出异常，顾名思义，也就是说一旦发生异常，我把这个异常抛出去，让调用者去进行处理，自己不进行具体的处理，此时需要用到 throw 和 throws 关键字。

```
public class Main {
    public static void main(String[] args) {
        try {
            createFile();
        } catch (Exception e) {
            // TODO: handle exception
        }
    }
    public static void createFile() throws IOException{
        File file = new File("d:/a.txt");
        if(!file.exists())
            file.createNewFile();
    }
}
```

这段代码和上面一段代码的区别是，在实际的 createFile 方法中并没有捕获异常，而是用 throws 关键字声明抛出异常，即告知这个方法的调用者此方法可能会抛出 IOException。那么在 main 方法中调用 createFile 方法的时候，采用 try...catch 块进行了异常捕获处理。

当然还可以采用 throw 关键字手动来抛出异常对象。下面看一个例子：

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] data = new int[]{1,2,3};
            System.out.println(getDataByIndex(-1,data));
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
```

```
    }
    public static int getDataByIndex(int index,int[] data) {
        if(index<0||index>=data.length)
            throw new ArrayIndexOutOfBoundsException("数组下标越界");
        return data[index];
    }
}
```

然后在 catch 块中进行捕获。

也就说在 Java 中进行异常处理的话，对于可能会发生异常的代码，可以选择三种方法来进行异常处理：

1）对代码块用 try..catch 进行异常捕获处理；

2）在该代码的方法体外用 throws 进行抛出声明，告知此方法的调用者这段代码可能会出现这些异常，你需要谨慎处理。此时有两种情况：

- 如果声明抛出的异常是非运行时异常，此方法的调用者必须显示地用 try..catch 块进行捕获或者继续向上层抛出异常。
- 如果声明抛出的异常是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

3）在代码块用 throw 手动抛出一个异常对象，此时也有两种情况，跟 2）中的类似：

- 如果抛出的异常对象是非运行时异常，此方法的调用者必须显示地用 try..catch 块进行捕获或者继续向上层抛出异常。
- 如果抛出的异常对象是运行时异常，此方法的调用者可以选择地进行异常捕获处理。

（如果最终将异常抛给 main 方法，则相当于交给 jvm 自动处理，此时 jvm 会简单地打印异常信息）

四、 深刻理解 try,catch,finally,throws,throw

1. try,catch,finally

try 关键字用来包围可能会出现异常的逻辑代码，它单独无法使用，必须配合 catch 或者 finally 使用。Java 编译器允许的组合使用形式只有以下三种形式：

```
try...catch...;      try....finally.....;    try....catch...finally...
```

当然 catch 块可以有多个，注意 try 块只能有一个,finally 块是可选的（但是最多只能有一个 finally 块）。

三个块执行的顺序为 try—>catch—>finally。

当然如果没有发生异常，则 catch 块不会执行。但是 finally 块无论在什么情况下都是会

执行的（这点要非常注意，因此部分情况下，都会将释放资源的操作放在 finally 块中进行）。

在有多个 catch 块的时候，是按照 catch 块的先后顺序进行匹配的，一旦异常类型被一个 catch 块匹配，则不会与后面的 catch 块进行匹配。

在使用 try..catch..finally 块的时候，注意千万不要在 finally 块中使用 return，因为 finally 中的 return 会覆盖已有的返回值。下面看一个例子：

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
public class Main {
    public static void main(String[] args) {
        String str = new Main().openFile();
        System.out.println(str);
    }
    public String openFile() {
        try {
            FileInputStream inputStream = new FileInputStream("d:/a.txt");
            int ch = inputStream.read();
            System.out.println("aaa");
            return "step1";
        } catch (FileNotFoundException e) {
            System.out.println("file not found");
            return "step2";
        }catch (IOException e) {
            System.out.println("io exception");
            return "step3";
        }finally{
            System.out.println("finally block");
            //return "finally";
        }
    }
}
```

这段程序的输出结果为：

可以看出，在 try 块中发生 FileNotFoundException 之后，就跳到第一个 catch 块，打印"file not found"信息，并将"step2"赋值给返回值，然后执行 finally 块，最后将返回值返回。

从这个例子说明，**无论 try 块或者 catch 块中是否包含 return 语句，都会执行 finally 块。**

如果将这个程序稍微修改一下，将 finally 块中的 return 语句注释去掉，运行结果是：

最后打印出的是"finally"，返回值被重新覆盖了。

因此如果方法有返回值，切忌不要再 finally 中使用 return，这样会使得程序结构变得混乱。

2. throws 和 thow 关键字

throws 出现在方法的声明中，表示该方法可能会抛出的异常，然后交给上层调用它的方法程序处理，允许 throws 后面跟着多个异常类型；

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。throw 只会出现在方法体中，当方法在执行过程中遇到异常情况时，将异常信息封装为异常对象，然后 throw 出去。throw 关键字的一个非常重要的作用就是异常类型的转换（会在后面阐述道）。

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。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，但是不会由方法去处理异常，真正的处理异常由此方法的上层调用处理。

五、在类继承的时候，方法覆盖时如何进行异常抛出声明

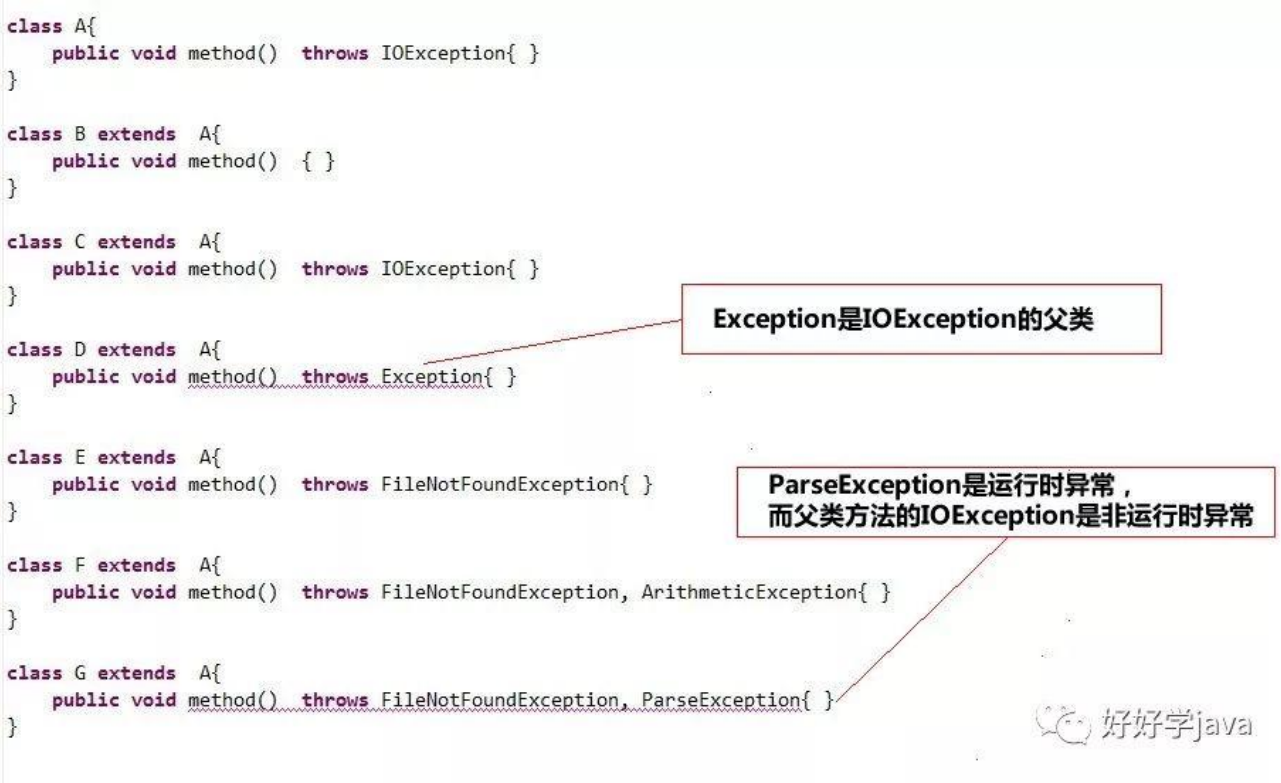
本小节讨论子类重写父类方法的时候，如何确定异常抛出声明的类型。下面是三点原则：

1）父类的方法没有声明异常，子类在重写该方法的时候不能声明异常；

2）如果父类的方法声明一个异常 exception1，则子类在重写该方法的时候声明的异常不能是 exception1 的父类；

3）如果父类的方法声明的异常类型只有非运行时异常（运行时异常），则子类在重写该

方法的时候声明的异常也只能有非运行时异常（运行时异常），不能含有运行时异常（非运行时异常）。



六、 异常的深入理解

例 1. 这个例子主要通过两个方法对比来演示一下有了异常以后代码的执行流程。

```
public static void testException1() {
        int[] ints = new int[] { 1, 2, 3, 4 };
        System.out.println("异常出现前");
        try {
            System.out.println(ints[4]);
            System.out.println("我还有幸执行到吗");// 发生异常以后，后面的代码不能被执行
        } catch (IndexOutOfBoundsException e) {
            System.out.println("数组越界错误");
        }
        System.out.println("异常出现后");
    }
    /*output:
    异常出现前
    数组越界错误
    4
```

```
    异常出现后
    */
public static void testException2() {
        int[] ints = new int[] { 1, 2, 3, 4 };
       System.out.println("异常出现前");
       System.out.println(ints[4]);
        System.out.println("我还有幸执行到吗");// 发生异常以后，他后面的代码不能被执行


 }
```

首先指出例子中的不足之处，IndexOutofBoundsException 是一个非受检异常，所以不用 try...catch...显示捕捉，但是我的目的是对同一个异常用不同的处理方式，看它会有什么不同的而结果（这里也就只能用它将就一下了）。异常出现时第一个方法只是跳出了 try 块，但是它后面的代码会照样执行的。但是第二种就不一样了直接跳出了方法，比较强硬。从第一个方法中我们看到，try...catch...是一种"事务性"的保障，它的目的是保证程序在异常的情况下运行完毕，同时它还会告知程序员程序中出错的详细信息（这种详细信息有时要依赖于程序员设计）。

例 2. 重新抛出异常

```
public class Rethrow {
    public static void readFile(String file) throws FileNotFoundException {
        try {
            BufferedInputStream in = new BufferedInputStream(new
FileInputStream(file));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.err.println("不知道如何处理该异常或者根本不想处理它，但是不做处理又
不合适，这是重新抛出异常交给上一级处理");
            //重新抛出异常
            throw e;
        }
    }
    public static void printFile(String file) {
        try {
            readFile(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    public static void main(String[] args) {
```

```
        printFile("D:/file");
    }
}
```

异常的本意是好的，让我们试图修复程序，但是现实中我们修复的几率很小，我们很多时候就是用它来记录出错的信息。如果你厌倦了不停的处理异常，重新抛出异常对你来说可能是一个很好的解脱。原封不动的把这个异常抛给上一级，抛给调用这个方法的人，让他来费脑筋吧。这样看来，java 异常（当然指的是受检异常）又给我们平添很多麻烦，尽管它的出发点是好的。

例 3. 异常链的使用及异常丢失

定义三个异常类：ExceptionA,ExceptionB,ExceptionC

```
public class ExceptionA extends Exception {
    public ExceptionA(String str) {
        super();
    }
}

public class ExceptionB extends ExceptionA {
    public ExceptionB(String str) {
        super(str);
    }
}

public class ExceptionC extends ExceptionA {
    public ExceptionC(String str) {
        super(str);
    }
}
```

异常丢失的情况：

```
public class NeverCaught {
    static void f() throws ExceptionB{
        throw new ExceptionB("exception b");
    }
    static void g() throws ExceptionC {
        try {
            f();
        } catch (ExceptionB e) {
            ExceptionC c = new ExceptionC("exception a");
```

```
            throw c;
        }
    }
    public static void main(String[] args) {
            try {
                g();
            } catch (ExceptionC e) {
                e.printStackTrace();
            }
    }
}
```

```
/*
exception.ExceptionC
at exception.NeverCaught.g(NeverCaught.java:12)
at exception.NeverCaught.main(NeverCaught.java:19)
*/
```

为什么只是打印出来了 ExceptionC 而没有打印出 ExceptionB 呢？这个还是自己分析一下吧！

上面的情况相当于少了一种异常，这在我们排错的过程中非常的不利。那我们遇到上面的情况应该怎么办呢？这就是异常链的用武之地：保存异常信息，在抛出另外一个异常的同时不丢失原来的异常。

```
public class NeverCaught {
    static void f() throws ExceptionB{
        throw new ExceptionB("exception b");
    }
    static void g() throws ExceptionC {
        try {
            f();
        } catch (ExceptionB e) {
            ExceptionC c = new ExceptionC("exception a");
            //异常连
            c.initCause(e);
            throw c;
        }
   }
    public static void main(String[] args) {
            try {
```

```
                g();
            } catch (ExceptionC e) {
                e.printStackTrace();
            }
    }
}

/*
exception.ExceptionC
at exception.NeverCaught.g(NeverCaught.java:12)
at exception.NeverCaught.main(NeverCaught.java:21)
Caused by: exception.ExceptionB
at exception.NeverCaught.f(NeverCaught.java:5)
at exception.NeverCaught.g(NeverCaught.java:10)
... 1 more
*/
```

这个异常链的特性是所有异常均具备的，因为这个 initCause()方法是从 Throwable 继承的。

例 4. 清理工作

清理工作对于我们来说是必不可少的，因为如果一些消耗资源的操作，比如 IO,JDBC。如果我们用完以后没有及时正确的关闭，那后果会很严重，这意味着内存泄露。异常的出现要求我们必须设计一种机制不论什么情况下，资源都能及时正确的清理。这就是 finally。

```
public void readFile(String file) {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new InputStreamReader(
                    new FileInputStream(file)));
            // do some other work
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

例子非常的简单，是一个读取文件的例子。这样的例子在 JDBC 操作中也非常的常见。（所以，我觉得对于资源的及时正确清理是一个程序员的基本素质之一。）

Try...finally 结构也是保证资源正确关闭的一个手段。如果你不清楚代码执行过程中会发生什么异常情况会导致资源不能得到清理，那么你就用 try 对这段"可疑"代码进行包装，然后在 finally 中进行资源的清理。举一个例子：

```
public void readFile() {
        BufferedReader reader = null;
        try {
            reader = new BufferedReader(new InputStreamReader(
                    new FileInputStream("file")));
            // do some other work
            //close reader
            reader.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

我们注意一下这个方法和上一个方法的区别，下一个人可能习惯更好一点，及早的关闭 reader。但是往往事与愿违，因为在 reader.close()以前异常随时可能发生，这样的代码结构不能预防任何异常的出现。因为程序会在异常出现的地方跳出，后面的代码不能执行（这在上面已经用实例证明过）。这时我们就可以用 try...finally 来改造：

```
public void readFile() {
        BufferedReader reader = null;
        try {
            try {
                reader = new BufferedReader(new InputStreamReader(
                        new FileInputStream("file")));
                // do some other work
                // close reader
            } finally {
                reader.close();
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }
    }
```

及早的关闭资源是一种良好的行为，因为时间越长你忘记关闭的可能性越大。这样在配合上 try...finally 就保证万无一失了(不要嫌麻烦，java 就是这么中规中矩)。

再说一种情况，假如我想在构造方法中打开一个文件或者创建一个 JDBC 连接，因为我们要在其他的方法中使用这个资源，所以不能在构造方法中及早的将这个资源关闭。那我们是不是就没辙了呢？答案是否定的。看一下下面的例子：

```
public class ResourceInConstructor {
    BufferedReader reader = null;
    public ResourceInConstructor() {
        try {
            reader = new BufferedReader(new InputStreamReader(new
FileInputStream("")));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
    public void readFile() {
        try {
            while(reader.readLine()!=null) {
                //do some work
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public void dispose() {
        try {
            reader.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

七、 异常的误用

对于异常的误用着实很常见，上一部分中已经列举了几个，大家仔细的看一下。下面再说两个其他的。

例 1.用一个 Exception 来捕捉所有的异常，颇有"一夫当关万夫莫开"的气魄。不过这也是最傻的行为。

```
public void readFile(String file) {
        BufferedReader reader = null;
        Connection conn = null;
        try {
            reader = new BufferedReader(new InputStreamReader(
                    new FileInputStream(file)));
            // do some other work
            conn = DriverManager.getConnection("");
            //...
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                reader.close();
                conn.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

从异常角度来说这样严格的程序确实是万无一失，所有的异常都能捕获。但是站在编程人员的角度，万一这个程序出错了我们该如何分辨是到底是那引起的呢，IO 还是 JDBC...所以，这种写法很值得当做一个反例。大家不要以为这种做法很幼稚，傻子才会做。我在公司实习时确实看见了类似的情况：只不过是人家没有用 Exception 而是用了 Throwable。

例 2. 这里就不举例子了，上面的程序都是反例。异常是程序处理意外情况的机制，当程序发生意外时，我们需要尽可能多的得到意外的信息，包括发生的位置，描述，原因等等。这些都是我们解决问题的线索。但是上面的例子都只是简单的 printStackTrace()。如果我们自己写代码，就要尽可能多的对这个异常进行描述。比如说为什么会出现这个异常，什么情况下会发生这个异常。如果传入方法的参数不正确，告知什么样的参数是合法的参数，或者给出一个 sample。

例 3. 将 try block 写的简短，不要所有的东西都扔在这里，我们尽可能的分析出到底哪几行程序可能出现异常，只是对可能出现异常的代码进行 try。尽量为每一个异常写一个 try...catch，避免异常丢失。在 IO 操作中，一个 IOException 也具有"一夫当关万夫莫开"的气魄。

八、 异常处理和设计的几个建议

1. 只在必要使用异常的地方才使用异常，不要用异常去控制程序的流程

谨慎地使用异常，异常捕获的代价非常高昂，异常使用过多会严重影响程序的性能。如果在程序中能够用 if 语句和 Boolean 变量来进行逻辑判断，那么尽量减少异常的使用，从而避免不必要的异常捕获和处理。比如下面这段经典的程序：

```
public void useExceptionsForFlowControl() {
  try {
  while (true) {
    increaseCount();
    }
  } catch (MaximumCountReachedException ex) {
  }
  //Continue execution
}
public void increaseCount() throws MaximumCountReachedException {
  if (count >= 5000)
    throw new MaximumCountReachedException();
}
```

上边的 useExceptionsForFlowControl()用一个无限循环来增加 count 直到抛出异常，这种做法并没有说让代码不易读，而是使得程序执行效率降低。

2. 切忌使用空 catch 块

在捕获了异常之后什么都不做，相当于忽略了这个异常。千万不要使用空的 catch 块，空的 catch 块意味着你在程序中隐藏了错误和异常，并且很可能导致程序出现不可控的执行结果。如果你非常肯定捕获到的异常不会以任何方式对程序造成影响，最好用 Log 日志将该异常进行记录，以便日后方便更新和维护。

3. 检查异常和非检查异常的选择

一旦你决定抛出异常，你就要决定抛出什么异常。这里面的主要问题就是抛出检查异常还是非检查异常。

检查异常导致了太多的 try...catch 代码，可能有很多检查异常对开发人员来说是无法合理地进行处理的，比如 SQLException，而开发人员却不得不去进行 try...catch，这样就会导致经常出现这样一种情况：逻辑代码只有很少的几行，而进行异常捕获和处理的代码却有很多行。这样不仅导致逻辑代码阅读起来晦涩难懂，而且降低了程序的性能。

我个人建议尽量避免检查异常的使用，如果确实该异常情况的出现很普遍，需要提醒调用者注意处理的话，就使用检查异常；否则使用非检查异常。

因此，在一般情况下，我觉得尽量将检查异常转变为非检查异常交给上层处理。

4. 注意 catch 块的顺序

不要把上层类的异常放在最前面的 catch 块。比如下面这段代码：

```
try {
        FileInputStream inputStream = new FileInputStream("d:/a.txt");
        int ch = inputStream.read();
        System.out.println("aaa");
        return "step1";
    } catch (IOException e) {
          System.out.println("io exception");
         return "step2";
    }catch (FileNotFoundException e) {
        System.out.println("file not found");
        return "step3";
    }finally{
        System.out.println("finally block");
        //return "finally";
    }
```

第二个 catch 的 FileNotFoundException 将永远不会被捕获到，因为 FileNotFoundException 是 IOException 的子类。

5. 不要将提供给用户看的信息放在异常信息里

```
public class Main {
    public static void main(String[] args) {
```

```
        try {
            String user = null;
            String pwd = null;
            login(user,pwd);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
    public static void login(String user,String pwd) {
        if(user==null||pwd==null)
            throw new NullPointerException("用户名或者密码为空");
        //...
    }
}
```

展示给用户错误提示信息最好不要跟程序混淆一起，比较好的方式是将所有错误提示信息放在一个配置文件中统一管理。

6. 避免多次在日志信息中记录同一个异常

只在异常最开始发生的地方进行日志信息记录。很多情况下异常都是层层向上跑出的，如果在每次向上抛出的时候，都 Log 到日志系统中，则会导致无从查找异常发生的根源。

7. 异常处理尽量放在高层进行

尽量将异常统一抛给上层调用者，由上层调用者统一之时如何进行处理。如果在每个出现异常的地方都直接进行处理，会导致程序异常处理流程混乱，不利于后期维护和异常错误排查。由上层统一进行处理会使得整个程序的流程清晰易懂。

8. 在 finally 中释放资源

如果有使用文件读取、网络操作以及数据库操作等，记得在 finally 中释放资源。这样不仅会使得程序占用更少的资源，也会避免不必要的由于资源未释放而发生的异常情况。