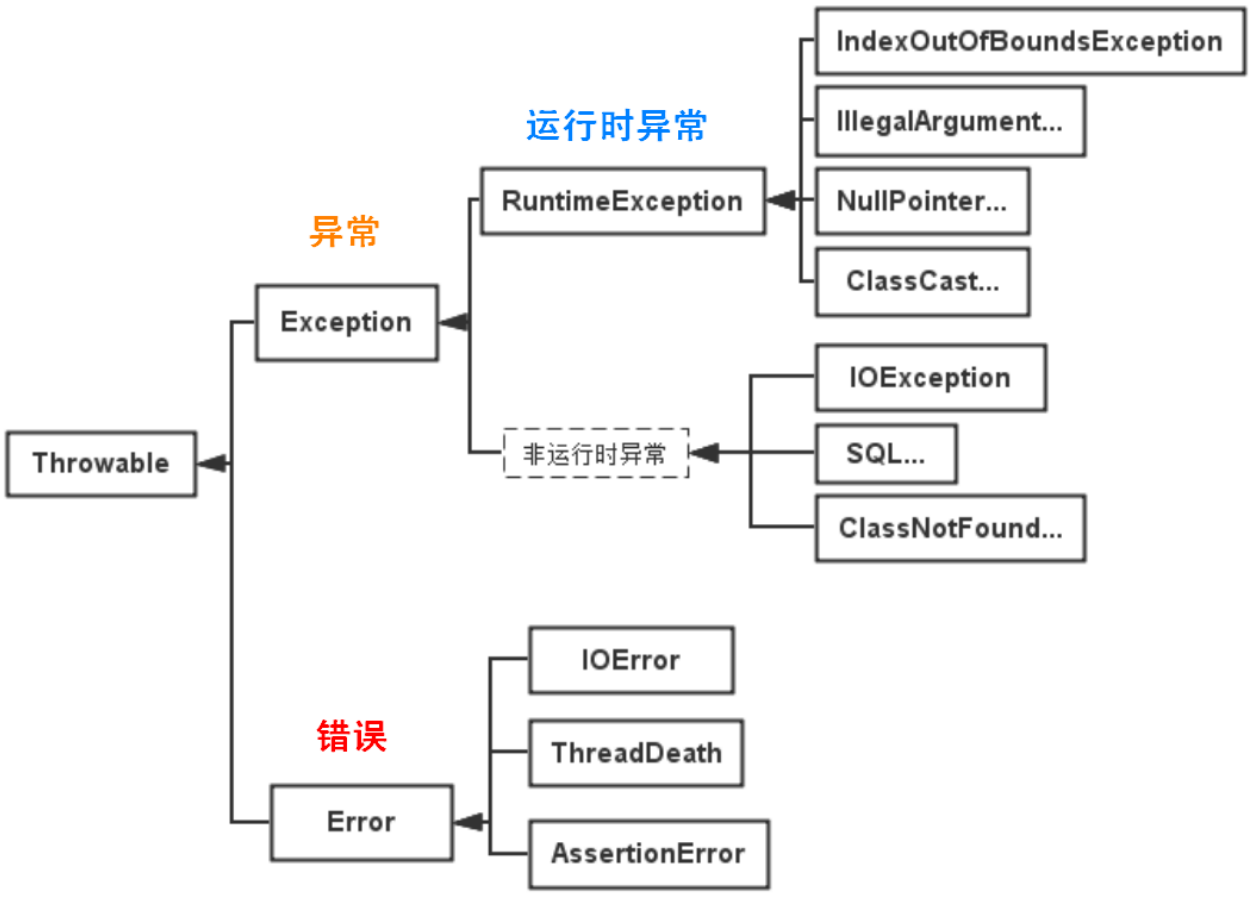
Java 基础 - 异常机制详解

Java异常是Java提供的一种识别及响应错误的一致性机制，java异常机制可以使程序中异常处理代码和正常业务代码分离，保证程序代码更加优雅，并提高程序健壮性。

异常的层次结构

异常指不期而至的各种状况，如：文件找不到、网络连接失败、非法参数等。异常是一个事件，它发生在程序运行期间，干扰了正常的指令流程。Java通 过API中Throwable类的众多子类描述各种不同的异常。因而，Java异常都是对象，是Throwable子类的实例，描述了出现在一段编码中的 错误条件。当条件生成时，错误将引发异常。

Java异常类层次结构图：



l

Throwable

Throwable 是 Java 语言中所有错误与异常的超类。

Throwable 包含两个子类：Error（错误）和 Exception（异常），它们通常用于指示发生了异常情况。

Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获取堆栈跟踪数据等信息。

Error（错误）

Error 类及其子类：程序中无法处理的错误，表示运行应用程序中出现了严重的错误。

此类错误一般表示代码运行时 JVM 出现问题。通常有 Virtual MachineError（虚拟机运行错误）、NoClassDefFoundError（类定义错误）等。比如 OutOfMemoryError：内存不足错误；StackOverflowError：栈溢出错误。此类错误发生时，JVM 将终止线程。

这些错误是不受检异常，非代码性错误。因此，当此类错误发生时，应用程序不应该去处理此类错误。按照Java惯例，我们是不应该实现任何新的Error子类的！

Exception（异常）

程序本身可以捕获并且可以处理的异常。Exception 这种异常又分为两类：运行时异常和编译时异常。

- **运行时异常**

都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch语句捕获它，也没有用throws子句声明抛出它，也会编译通过。

- **非运行时异常** （编译异常）

是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

可查的异常（*checked exceptions*）和不可查的异常（*unchecked exceptions*）

- **可查异常**（编译器要求必须处置的异常）：

正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

- **不可查异常**(编译器不要求强制处置的异常)

包括运行时异常（RuntimeException与其子类）和错误（Error）。

异常基础

TIP

接下来我们看下异常使用的基础。

异常关键字

- **try** – 用于监听。将要被监听的代码(可能抛出异常的代码)放在try语句块之内，当try语句块内发生异常时，异常就被抛出。
- **catch** – 用于捕获异常。catch用来捕获try语句块中发生的异常。
- **finally** – finally语句块总是会被执行。它主要用于回收在try块里打开的物力资源(如数据库连接、网络连接和磁盘文件)。只有finally块，执行完成之后，才会回来执行try或者catch块中的return或者throw语句，如果finally中使用了return或者throw等终止方法的语句，则就不会跳回执行，直接停止。
- **throw** – 用于抛出异常。
- **throws** – 用在方法签名中，用于声明该方法可能抛出的异常。

异常的申明*(throws)*

在Java中，当前执行的语句必属于某个方法，Java解释器调用main方法执行开始执行程序。若方法中存在检查异常，如果不对其捕获，那必须在方法头中显式声明该异常，以便于告知方法调用者此方法有异常，需要进行处理。在方法中声明一个异常，方法头中使用关键字throws，后面接上要声明的异常。若声明多个异常，则使用逗号分割。如下所示：

```
public static void method() throws IOException, FileNotFoundException{
    //something statements
}
```

注意：若是父类的方法没有声明异常，则子类继承方法后，也不能声明异常。

通常，应该捕获那些知道如何处理的异常，将不知道如何处理的异常继续传递下去。传递异常可以在方法签名处使用 throws 关键字声明可能会抛出的异常。

```java
private static void readFile(String filePath) throws IOException {
    File file = new File(filePath);
    String result;
    BufferedReader reader = new BufferedReader(new FileReader(file));
    while((result = reader.readLine())!=null) {
        System.out.println(result);
    }
    reader.close();
}
```

Throws抛出异常的规则：

- 如果是不可查异常（unchecked exception），即Error、RuntimeException或它们的子类，那么可以不使用throws关键字来声明要抛出的异常，编译仍能顺利通过，但在运行时会被系统抛出。
- 必须声明方法可抛出的任何可查异常（checked exception）。即如果一个方法可能出现受可查异常，要么用try-catch语句捕获，要么用throws子句声明将它抛出，否则会导致编译错误
- 仅当抛出了异常，该方法的调用者才必须处理或者重新抛出该异常。当方法的调用者无力处理该异常的时候，应该继续抛出，而不是囫囵吞枣。
- 调用方法必须遵循任何可查异常的处理和声明规则。若覆盖一个方法，则不能声明与覆盖方法不同的异常。声明的任何异常必须是被覆盖方法所声明异常的同类或子类。

异常的抛出*(throw)*

如果代码可能会引发某种错误，可以创建一个合适的异常类实例并抛出它，这就是抛出异常。如下所示：

```java
public static double method(int value) {
    if(value == 0) {
        throw new ArithmeticException("参数不能为0"); //抛出一个运行时异常
    }
    return 5.0 / value;
}
```

大部分情况下都不需要手动抛出异常，因为Java的大部分方法要么已经处理异常，要么已声明异常。所以一般都是捕获异常或者再往上抛。

有时我们会从 catch 中抛出一个异常，目的是为了改变异常的类型。多用于在多系统集成时，当某个子系统故障，异常类型可能有多种，可以用统一的异常类型向外暴露，不需暴露太多内部异常细节。

```java
private static void readFile(String filePath) throws MyException {
    try {
        // code
    } catch (IOException e) {
        MyException ex = new MyException("read file failed.");
        ex.initCause(e);
        throw ex;
    }
}
```

异常的自定义

习惯上，定义一个异常类应包含两个构造函数，一个无参构造函数和一个带有详细描述信息的构造函数（Throwable 的 toString 方法会打印这些详细信息，调试时很有用），比如上面用到的自定义MyException：

```
public class MyException extends Exception {
    public MyException(){ }
    public MyException(String msg){
        super(msg);
    }
    // ...
}
```

异常的捕获

异常捕获处理的方法通常有：

- try-catch
- try-catch-finally
- try-finally
- try-with-resource

try-catch

在一个 try-catch 语句块中可以捕获多个异常类型，并对不同类型的异常做出不同的处理

```
private static void readFile(String filePath) {
    try {
        // code
    } catch (FileNotFoundException e) {
        // handle FileNotFoundException
    } catch (IOException e){
        // handle IOException
    }
}
```

同一个 catch 也可以捕获多种类型异常，用 | 隔开

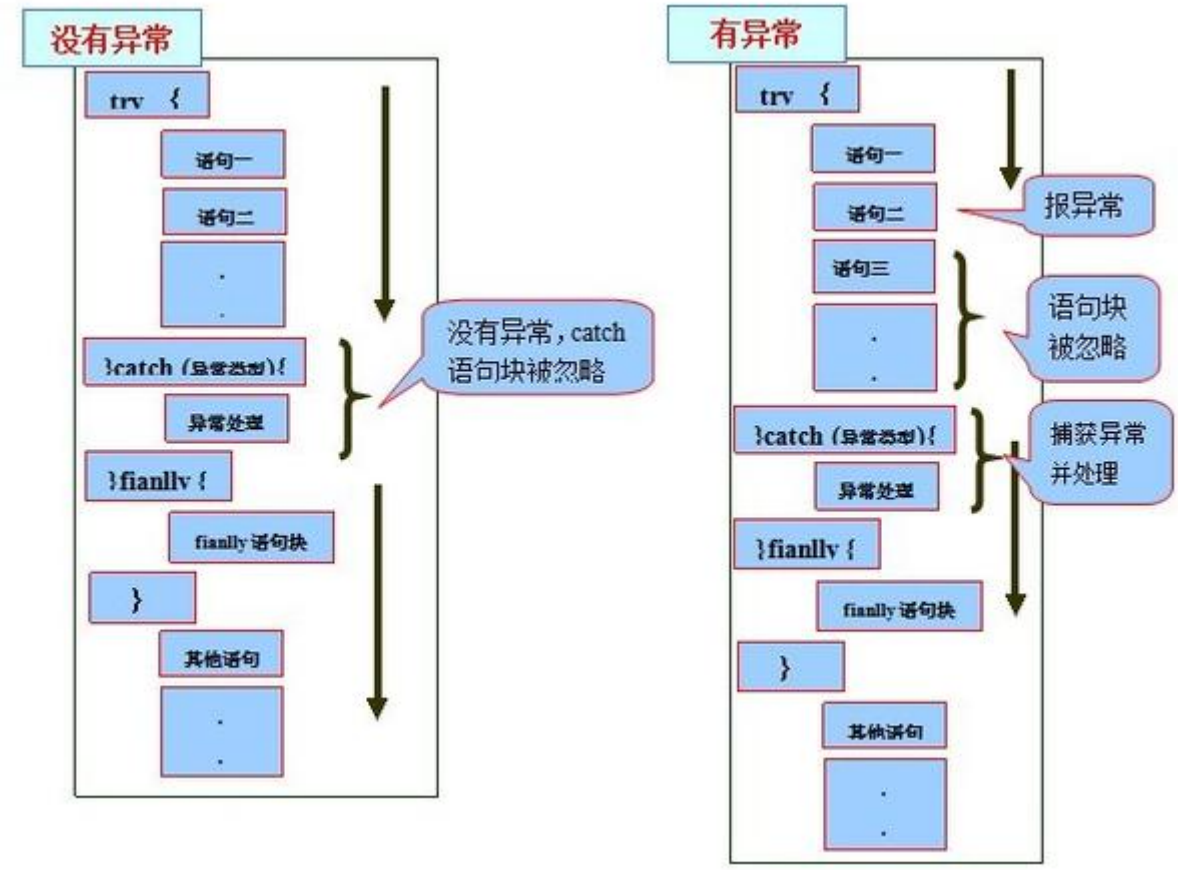
```
private static void readFile(String filePath) {
    try {
        // code
    } catch (FileNotFoundException | UnknownHostException e) {
        // handle FileNotFoundException or UnknownHostException
    } catch (IOException e){
        // handle IOException
    }
}
```

try-catch-finally

- 常规语法

```
try {
    //执行程序代码，可能会出现异常
} catch(Exception e) {
    //捕获异常并处理
} finally {
    //必执行的代码
}
```

- 执行的顺序
  - 当try没有捕获到异常时：try语句块中的语句逐一被执行，程序将跳过catch语句块，执行finally语句块和其后的语句；
  - 当try捕获到异常，catch语句块里没有处理此异常的情况：当try语句块里的某条语句出现异常时，而没有处理此异常的catch语句块时，此异常将会抛给JVM处理，finally语句块里的语句还是会被执行，但finally语句块后的语句不会被执行；
  - 当try捕获到异常，catch语句块里有处理此异常的情况：在try语句块中是按照顺序来执行的，当执行到某一条语句出现异常时，程序将跳到catch语句块，并与catch语句块逐一匹配，找到与之对应的处理程序，其他的catch语句块将不会被执行，而try语句块中，出现异常之后的语句也不会被执行，catch语句块执行完后，执行finally语句块里的语句，最后执行finally语句块后的语句；



- 一个完整的例子

```
private static void readFile(String filePath) throws MyException {
    File file = new File(filePath);
    String result;
```

```
    BufferedReader reader = null;
    try {
        reader = new BufferedReader(new FileReader(file));
        while((result = reader.readLine())!=null) {
            System.out.println(result);
        }
    } catch (IOException e) {
        System.out.println("readFile method catch block.");
        MyException ex = new MyException("read file failed.");
        ex.initCause(e);
        throw ex;
    } finally {
        System.out.println("readFile method finally block.");
        if (null != reader) {
            try {
                reader.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

try-finally

> 可以直接用try-finally吗？ 可以。

try块中引起异常，异常代码之后的语句不再执行，直接执行finally语句。 try块没有引发异常，则执行完try块就执行finally语句。

try-finally可用在不需要捕获异常的代码，可以保证资源在使用后被关闭。例如IO流中执行完相应操作后，关闭相应资源；使用Lock对象保证线程同步，通过finally可以保证锁会被释放；数据库连接代码时，关闭连接操作等等。

```
//以Lock加锁为例，演示try-finally
ReentrantLock lock = new ReentrantLock();
try {
    //需要加锁的代码
} finally {
    lock.unlock(); //保证锁一定被释放
}
```

finally遇见如下情况不会执行

- 在前面的代码中用了System.exit()退出程序。
- finally语句块中发生了异常。
- 程序所在的线程死亡。
- 关闭CPU。

try-with-resource

> try-with-resource是Java 7中引入的，很容易被忽略。

上面例子中，finally 中的 close 方法也可能抛出 IOException, 从而覆盖了原始异常。JAVA 7 提供了更优雅的方式来实现资源的自动释放，自动释放的资源需要是实现了 AutoCloseable 接口的类。

- 代码实现

```
private  static void tryWithResourceTest(){
    try (Scanner scanner = new Scanner(new FileInputStream("c:/abc"),"UTF-8")){
        // code
    } catch (IOException e){
        // handle exception
    }
}
```

- 看下Scanner

```
public final class Scanner implements Iterator<String>, Closeable {
  // ...
}
public interface Closeable extends AutoCloseable {
    public void close() throws IOException;
}
```

try 代码块退出时，会自动调用 scanner.close 方法，和把 scanner.close 方法放在 finally 代码块中不同的是，若 scanner.close 抛出异常，则会被抑制，抛出的仍然为原始异常。被抑制的异常会由 addSusppressed 方法添加到原来的异常，如果想要获取被抑制的异常列表，可以调用 getSuppressed 方法来获取。

异常基础总结

- try、catch和finally都不能单独使用，只能是try-catch、try-finally或者try-catch-finally。
- try语句块监控代码，出现异常就停止执行下面的代码，然后将异常移交给catch语句块来处理。
- finally语句块中的代码一定会被执行，常用于回收资源 。
- throws：声明一个异常，告知方法调用者。
- throw ： 抛出一个异常，至于该异常被捕获还是继续抛出都与它无关。

Java编程思想一书中，对异常的总结。

- 在恰当的级别处理问题。（在知道该如何处理的情况下了捕获异常。）
- 解决问题并且重新调用产生异常的方法。
- 进行少许修补，然后绕过异常发生的地方继续执行。
- 用别的数据进行计算，以代替方法预计会返回的值。
- 把当前运行环境下能做的事尽量做完，然后把相同的异常重抛到更高层。
- 把当前运行环境下能做的事尽量做完，然后把不同的异常抛到更高层。
- 终止程序。
- 进行简化（如果你的异常模式使问题变得太复杂，那么用起来会非常痛苦）。
- 让类库和程序更安全。

常用的异常

在Java中提供了一些异常用来描述经常发生的错误，对于这些异常，有的需要程序员进行捕获处理或声明抛出，有的是由Java虚拟机自动进行捕获处理。Java中常见的异常类:

- **RuntimeException**
  - java.lang.ArrayIndexOutOfBoundsException 数组索引越界异常。当对数组的索引值为负数或大于等于数组大小时抛出。
  - java.lang.ArithmeticException 算术条件异常。譬如：整数除零等。

- java.lang.NullPointerException 空指针异常。当应用试图在要求使用对象的地方使用了null时，抛出该异常。譬如：调用null对象的实例方法、访问null对象的属性、计算null对象的长度、使用throw语句抛出null等等
- java.lang.ClassNotFoundException 找不到类异常。当应用试图根据字符串形式的类名构造类，而在遍历CLASSPAH之后找不到对应名称的class文件时，抛出该异常。
- java.lang.NegativeArraySizeException 数组长度为负异常
- java.lang.ArrayStoreException 数组中包含不兼容的值抛出的异常
- java.lang.SecurityException 安全性异常
- java.lang.IllegalArgumentException 非法参数异常

- **IOException**
  - IOException：操作输入流和输出流时可能出现的异常。
  - EOFException 文件已结束异常
  - FileNotFoundException 文件未找到异常
- **其他**
  - ClassCastException 类型转换异常类
  - ArrayStoreException 数组中包含不兼容的值抛出的异常
  - SQLException 操作数据库异常类
  - NoSuchFieldException 字段未找到异常
  - NoSuchMethodException 方法未找到抛出的异常
  - NumberFormatException 字符串转换为数字抛出的异常
  - StringIndexOutOfBoundsException 字符串索引超出范围抛出的异常
  - IllegalAccessException 不允许访问某类异常
  - InstantiationException 当应用程序试图使用Class类中的newInstance()方法创建一个类的实例，而指定的类对象无法被实例化时，抛出该异常

异常实践

TIP

在 Java 中处理异常并不是一个简单的事情。不仅仅初学者很难理解，即使一些有经验的开发者也需要花费很多时间来思考如何处理异常，包括需要处理哪些异常，怎样处理等等。这也是绝大多数开发团队都会制定一些规则来规范进行异常处理的原因。

当你抛出或捕获异常的时候，有很多不同的情况需要考虑，而且大部分事情都是为了改善代码的可读性或者 API 的可用性。

异常不仅仅是一个错误控制机制，也是一个通信媒介。因此，为了和同事更好的合作，一个团队必须要制定出一个最佳实践和规则，只有这样，团队成员才能理解这些通用概念，同时在工作中使用它。

这里给出几个被很多团队使用的异常处理最佳实践。

只针对不正常的情况才使用异常

> 异常只应该被用于不正常的条件，它们永远不应该被用于正常的控制流。《阿里手册》中：【强制】Java 类库中定义的可以通过预检查方式规避的RuntimeException异常不应该通过catch 的方式来处理，比如：NullPointerException，IndexOutOfBoundsException等等。

比如，在解析字符串形式的数字时，可能存在数字格式错误，不得通过catch Exception来实现

- 代码1

```
if (obj != null) {
  //...
}
```

- 代码2

```
try {
  obj.method();
} catch (NullPointerException e) {
  //...
}
```

主要原因有三点：

- 异常机制的设计初衷是用于不正常的情况，所以很少会会JVM实现试图对它们的性能进行优化。所以，创建、抛出和捕获异常的开销是很昂贵的。
- 把代码放在try-catch中返回阻止了JVM实现本来可能要执行的某些特定的优化。
- 对数组进行遍历的标准模式并不会导致冗余的检查，有些现代的JVM实现会将它们优化掉。

在 *finally* 块中清理资源或者使用 *try-with-resource* 语句

当使用类似InputStream这种需要使用后关闭的资源时，一个常见的错误就是在try块的最后关闭资源。

- 错误示例

```
public void doNotCloseResourceInTry() {
    FileInputStream inputStream = null;
    try {
        File file = new File("./tmp.txt");
        inputStream = new FileInputStream(file);
        // use the inputStream to read a file
        // do NOT do this
        inputStream.close();
    } catch (FileNotFoundException e) {
        log.error(e);
    } catch (IOException e) {
        log.error(e);
    }
}
```

问题就是，只有没有异常抛出的时候，这段代码才可以正常工作。try 代码块内代码会正常执行，并且资源可以正常关闭。但是，使用 try 代码块是有原因的，一般调用一个或多个可能抛出异常的方法，而且，你自己也可能会抛出一个异常，这意味着代码可能不会执行到 try 代码块的最后部分。结果就是，你并没有关闭资源。

所以，你应该把清理工作的代码放到 finally 里去，或者使用 try-with-resource 特性。

- 方法一：使用 finally 代码块

与前面几行 try 代码块不同，finally 代码块总是会被执行。不管 try 代码块成功执行之后还是你在 catch 代码块中处理完异常后都会执行。因此，你可以确保你清理了所有打开的资源。

```
public void closeResourceInFinally() {
    FileInputStream inputStream = null;
    try {
        File file = new File("./tmp.txt");
```

```
            inputStream = new FileInputStream(file);
            // use the inputStream to read a file
        } catch (FileNotFoundException e) {
            log.error(e);
        } finally {
            if (inputStream != null) {
                try {
                    inputStream.close();
                } catch (IOException e) {
                    log.error(e);
                }
            }
        }
    }
```

- 方法二：Java 7 的 try-with-resource 语法

如果你的资源实现了 AutoCloseable 接口，你可以使用这个语法。大多数的 Java 标准资源都继承了这个接口。当你在 try 子句中打开资源，资源会在 try 代码块执行后或异常处理后自动关闭。

```
    public void automaticallyCloseResource() {
        File file = new File("./tmp.txt");
        try (FileInputStream inputStream = new FileInputStream(file);) {
            // use the inputStream to read a file
        } catch (FileNotFoundException e) {
            log.error(e);
        } catch (IOException e) {
            log.error(e);
        }
    }
```

尽量使用标准的异常

> 代码重用是值得提倡的，这是一条通用规则，异常也不例外。

重用现有的异常有几个好处：

- 它使得你的API更加易于学习和使用，因为它与程序员原来已经熟悉的习惯用法是一致的。
- 对于用到这些API的程序而言，它们的可读性更好，因为它们不会充斥着程序员不熟悉的异常。
- 异常类越少，意味着内存占用越小，并且转载这些类的时间开销也越小。

Java标准异常中有几个是经常被使用的异常。如下表格：

异常	使用场合
IllegalArgumentException	参数的值不合适
IllegalStateException	参数的状态不合适
NullPointerException	在null被禁止的情况下参数值为null
IndexOutOfBoundsException	下标越界
ConcurrentModificationException	在禁止并发修改的情况下，对象检测到并发修改
UnsupportedOperationException	对象不支持客户请求的方法

虽然它们是Java平台库迄今为止最常被重用的异常，但是，在许可的条件下，其它的异常也可以被重用。例如，如果你要实现诸如复数或者矩阵之类的算术对象，那么重用ArithmeticException和NumberFormatException将是非常合适的。如果一个异常满足你的需要，则不要犹豫，使用就可以，不过你一定要确保抛出异常的条件与该异常的文档中描述的条件一致。这种重用必须建立在语义的基础上，而不是名字的基础上。

最后，一定要清楚，选择重用哪一种异常并没有必须遵循的规则。例如，考虑纸牌对象的情形，假设有一个用于发牌操作的方法，它的参数(handSize)是发一手牌的纸牌张数。假设调用者在这个参数中传递的值大于整副牌的剩余张数。那么这种情形既可以被解释为IllegalArgumentException(handSize的值太大)，也可以被解释为IllegalStateException(相对客户的请求而言，纸牌对象的纸牌太少)。

对异常进行文档说明

> 当在方法上声明抛出异常时，也需要进行文档说明。目的是为了给调用者提供尽可能多的信息，从而可以更好地避免或处理异常。

在 Javadoc 添加 @throws 声明，并且描述抛出异常的场景。

```
/**
* Method description
*
* @throws MyBusinessException - businuess exception description
*/
public void doSomething(String input) throws MyBusinessException {
   // ...
}
```

同时，在抛出MyBusinessException 异常时，需要尽可能精确地描述问题和相关信息，这样无论是打印到日志中还是在监控工具中，都能够更容易被人阅读，从而可以更好地定位具体错误信息、错误的严重程度等。

优先捕获最具体的异常

> 大多数 IDE 都可以帮助你实现这个最佳实践。当你尝试首先捕获较不具体的异常时，它们会报告无法访问的代码块。

但问题在于，只有匹配异常的第一个 catch 块会被执行。 因此，如果首先捕获 IllegalArgumentException ，则永远不会到达应该处理更具体的 NumberFormatException 的 catch 块，因为它是 IllegalArgumentException 的子类。

总是优先捕获最具体的异常类，并将不太具体的 catch 块添加到列表的末尾。

你可以在下面的代码片断中看到这样一个 try-catch 语句的例子。 第一个 catch 块处理所有 NumberFormatException 异常，第二个处理所有非 NumberFormatException 异常的IllegalArgumentException 异常。

```
public void catchMostSpecificExceptionFirst() {
    try {
        doSomething("A message");
    } catch (NumberFormatException e) {
        log.error(e);
    } catch (IllegalArgumentException e) {
        log.error(e)
    }
}
```

不要捕获 *Throwable* 类

> Throwable 是所有异常和错误的超类。你可以在 catch 子句中使用它，但是你永远不应该这样做！

如果在 catch 子句中使用 Throwable ， 它不仅会捕获所有异常，也将捕获所有的错误。JVM 抛出错误，指出不应该由应用程序处理的严重问题。 典型的例子是 OutOfMemoryError 或者 StackOverflowError 。两者都是由应用程序控制之外的情况引起的，无法处理。

所以，最好不要捕获 Throwable ， 除非你确定自己处于一种特殊的情况下能够处理错误。

```
public void doNotCatchThrowable() {
    try {
        // do something
    } catch (Throwable t) {
        // don't do this!
    }
}
```

不要忽略异常

> 很多时候，开发者很有自信不会抛出异常，因此写了一个catch块，但是没有做任何处理或者记录日志。

```
public void doNotIgnoreExceptions() {
    try {
        // do something
    } catch (NumberFormatException e) {
        // this will never happen
    }
}
```

但现实是经常会出现无法预料的异常，或者无法确定这里的代码未来是不是会改动(删除了阻止异常抛出的代码)，而此时由于异常被捕获，使得无法拿到足够的错误信息来定位问题。

合理的做法是至少要记录异常的信息。

```
public void logAnException() {
    try {
        // do something
    } catch (NumberFormatException e) {
        log.error("This should never happen: " + e); // see this line
    }
}
```

不要记录并抛出异常

> 这可能是本文中最常被忽略的最佳实践。

可以发现很多代码甚至类库中都会有捕获异常、记录日志并再次抛出的逻辑。如下：

```
try {
    new Long("xyz");
} catch (NumberFormatException e) {
    log.error(e);
    throw e;
}
```

这个处理逻辑看着是合理的。但这经常会给同一个异常输出多条日志。如下：

```
17:44:28,945 ERROR TestExceptionHandling:65 - java.lang.NumberFormatException: For input string:
"xyz"
Exception in thread "main" java.lang.NumberFormatException: For input string: "xyz"
at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
at java.lang.Long.parseLong(Long.java:589)
at java.lang.Long.(Long.java:965)
at
com.stackify.example.TestExceptionHandling.logAndThrowException(TestExceptionHandling.java:63)
at com.stackify.example.TestExceptionHandling.main(TestExceptionHandling.java:58)
```

如上所示，后面的日志也没有附加更有用的信息。如果想要提供更加有用的信息，那么可以将异常包装为自定义异常。

```
public void wrapException(String input) throws MyBusinessException {
    try {
        // do something
    } catch (NumberFormatException e) {
        throw new MyBusinessException("A message that describes the error.", e);
    }
}
```

因此，仅仅当想要处理异常时才去捕获，否则只需要在方法签名中声明让调用者去处理。

包装异常时不要抛弃原始的异常

捕获标准异常并包装为自定义异常是一个很常见的做法。这样可以添加更为具体的异常信息并能够做针对的异常处理。 在你这样做时，请确保将原始异常设置为原因（注：参考下方代码 NumberFormatException e 中的原始异常 e ）。Exception 类提供了特殊的构造函数方法，它接受一个 Throwable 作为参数。否则，你将会丢失堆栈跟踪和原始异常的消息，这将会使分析导致异常的异常事件变得困难。

```
public void wrapException(String input) throws MyBusinessException {
    try {
        // do something
    } catch (NumberFormatException e) {
        throw new MyBusinessException("A message that describes the error.", e);
    }
}
```

不要使用异常控制程序的流程

不应该使用异常控制应用的执行流程，例如，本应该使用if语句进行条件判断的情况下，你却使用异常处理，这是非常不好的习惯，会严重影响应用的性能。

不要在*finally*块中使用*return*。

try块中的return语句执行成功后，并不马上返回，而是继续执行finally块中的语句，如果此处存在return语句，则在此直接返回，无情丢弃掉try块中的返回点。

如下是一个反例：

```
private int x = 0;
public int checkReturn() {
    try {
        // x等于1，此处不返回
        return ++x;
    } finally {
        // 返回的结果是2
        return ++x;
    }
}
```

深入理解异常

TIP

我们再深入理解下异常，看下底层实现。

*JVM*处理异常的机制？

提到JVM处理异常的机制，就需要提及Exception Table，以下称为异常表。我们暂且不急于介绍异常表，先看一个简单的 Java 处理异常的小例子。

```
public static void simpleTryCatch() {
   try {
       testNPE();
   } catch (Exception e) {
       e.printStackTrace();
   }
}
```

上面的代码是一个很简单的例子，用来捕获处理一个潜在的空指针异常。

当然如果只是看简简单单的代码，我们很难看出什么高深之处，更没有了今天文章要谈论的内容。

所以这里我们需要借助一把神兵利器，它就是javap,一个用来拆解class文件的工具，和javac一样由JDK提供。

然后我们使用javap来分析这段代码（需要先使用javac编译）

```
//javap -c Main
 public static void simpleTryCatch();
    Code:
       0: invokestatic  #3                  // Method testNPE:()V
       3: goto          11
       6: astore_0
       7: aload_0
       8: invokevirtual #5                  // Method java/lang/Exception.printStackTrace:()V
      11: return
    Exception table:
       from    to  target type
           0     3     6   Class java/lang/Exception
```

看到上面的代码，应该会有会心一笑，因为终于看到了Exception table，也就是我们要研究的异常表。

异常表中包含了一个或多个异常处理者(Exception Handler)的信息，这些信息包含如下

- **from** 可能发生异常的起始点
- **to** 可能发生异常的结束点
- **target** 上述from和to之前发生异常后的异常处理者的位置
- **type** 异常处理者处理的异常的类信息

那么异常表用在什么时候呢

答案是异常发生的时候，当一个异常发生时

- 1.JVM会在当前出现异常的方法中，查找异常表，是否有合适的处理者来处理
- 2.如果当前方法异常表不为空，并且异常符合处理者的from和to节点，并且type也匹配，则JVM调用位于target的调用者来处理。
- 3.如果上一条未找到合理的处理者，则继续查找异常表中的剩余条目
- 4.如果当前方法的异常表无法处理，则向上查找（弹栈处理）刚刚调用该方法的调用处，并重复上面的操作。
- 5.如果所有的栈帧被弹出，仍然没有处理，则抛给当前的Thread，Thread则会终止。
- 6.如果当前Thread为最后一个非守护线程，且未处理异常，则会导致JVM终止运行。

以上就是JVM处理异常的一些机制。

try catch -finally

除了简单的try-catch外，我们还常常和finally做结合使用。比如这样的代码

```
public static void simpleTryCatchFinally() {
   try {
       testNPE();
   } catch (Exception e) {
       e.printStackTrace();
   } finally {
       System.out.println("Finally");
   }
}
```

同样我们使用javap分析一下代码

```
public static void simpleTryCatchFinally();
    Code:
       0: invokestatic  #3                  // Method testNPE:()V
       3: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
       6: ldc           #7                  // String Finally
       8: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
```

```
      11: goto          41
      14: astore_0
      15: aload_0
      16: invokevirtual #5                  // Method java/lang/Exception.printStackTrace:()V
      19: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
      22: ldc           #7                  // String Finally
      24: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      27: goto          41
      30: astore_1
      31: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
      34: ldc           #7                  // String Finally
      36: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      39: aload_1
      40: athrow
      41: return
    Exception table:
       from    to  target type
           0     3    14   Class java/lang/Exception
           0     3    30   any
          14    19    30   any
```

和之前有所不同，这次异常表中，有三条数据，而我们仅仅捕获了一个Exception, 异常表的后两个item的type为any;上面的三条异常表item的意思为:

- 如果0到3之间，发生了Exception类型的异常，调用14位置的异常处理者。
- 如果0到3之间，无论发生什么异常，都调用30位置的处理者
- 如果14到19之间（即catch部分），不论发生什么异常，都调用30位置的处理者。

再次分析上面的Java代码，finally里面的部分已经被提取到了try部分和catch部分。我们再次调一下代码来看一下

```
public static void simpleTryCatchFinally();
    Code:
      //try 部分提取finally代码，如果没有异常发生，则执行输出finally操作，直至goto到41位置，执行返回操
作。

       0: invokestatic  #3                  // Method testNPE:()V
       3: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
       6: ldc           #7                  // String Finally
       8: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      11: goto          41

      //catch部分提取finally代码，如果没有异常发生，则执行输出finally操作，直至执行got到41位置，执行返
回操作。
      14: astore_0
      15: aload_0
      16: invokevirtual #5                  // Method java/lang/Exception.printStackTrace:()V
      19: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
      22: ldc           #7                  // String Finally
      24: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      27: goto          41
      //finally部分的代码如果被调用，有可能是try部分，也有可能是catch部分发生异常。
      30: astore_1
      31: getstatic     #6                  // Field java/lang/System.out:Ljava/io/PrintStream;
      34: ldc           #7                  // String Finally
      36: invokevirtual #8                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
```

```
      39: aload_1
      40: athrow      //如果异常没有被catch捕获，而是到了这里，执行完finally的语句后，仍然要把这个异常抛出去，传递给调用处。
      41: return
```

Catch先后顺序的问题

我们在代码中的catch的顺序决定了异常处理者在异常表的位置，所以，越是具体的异常要先处理，否则就会出现下面的问题

```
private static void misuseCatchException() {
   try {
       testNPE();
   } catch (Throwable t) {
       t.printStackTrace();
   } catch (Exception e) { //error occurs during compilings with tips Exception Java.lang.Exception has already benn caught.
       e.printStackTrace();
   }
}
```

这段代码会导致编译失败，因为先捕获Throwable后捕获Exception，会导致后面的catch永远无法被执行。

Return 和finally的问题

这算是我们扩展的一个相对比较极端的问题，就是类似这样的代码，既有return，又有finally，那么finally导致会不会执行

```
public static String tryCatchReturn() {
   try {
       testNPE();
       return  "OK";
   } catch (Exception e) {
       return "ERROR";
   } finally {
       System.out.println("tryCatchReturn");
   }
}
```

答案是finally会执行，那么还是使用上面的方法，我们来看一下为什么finally会执行。

```
public static java.lang.String tryCatchReturn();
    Code:
       0: invokestatic  #3                  // Method testNPE:()V
       3: ldc           #6                  // String OK
       5: astore_0
       6: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
       9: ldc           #8                  // String tryCatchReturn
      11: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      14: aload_0
      15: areturn       返回OK字符串，areturn意思为return a reference from a method
      16: astore_0
      17: ldc           #10                 // String ERROR
      19: astore_1
      20: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      23: ldc           #8                  // String tryCatchReturn
      25: invokevirtual #9                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
      28: aload_1
      29: areturn  //返回ERROR字符串
      30: astore_2
      31: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;
      34: ldc           #8                  // String tryCatchReturn
      36: invokevirtual #9                  // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
      39: aload_2
      40: athrow  如果catch有未处理的异常，抛出去。
```

异常是否耗时？为什么会耗时？

说用异常慢，首先来看看异常慢在哪里？有多慢？下面的测试用例简单的测试了建立对象、建立异常对象、抛出并接住异常对象三者的耗时对比：

```
public class ExceptionTest {

    private int testTimes;

    public ExceptionTest(int testTimes) {
        this.testTimes = testTimes;
    }

    public void newObject() {
        long l = System.nanoTime();
        for (int i = 0; i < testTimes; i++) {
            new Object();
        }
        System.out.println("建立对象：" + (System.nanoTime() - l));
    }

    public void newException() {
        long l = System.nanoTime();
        for (int i = 0; i < testTimes; i++) {
            new Exception();
        }
        System.out.println("建立异常对象：" + (System.nanoTime() - l));
    }

    public void catchException() {
        long l = System.nanoTime();
        for (int i = 0; i < testTimes; i++) {
            try {
                throw new Exception();
            } catch (Exception e) {
            }
        }
        System.out.println("建立、抛出并接住异常对象：" + (System.nanoTime() - l));
    }

    public static void main(String[] args) {
        ExceptionTest test = new ExceptionTest(10000);
        test.newObject();
        test.newException();
        test.catchException();
    }
}
```

运行结果：

```
建立对象：575817
建立异常对象：9589080
建立、抛出并接住异常对象：47394475
```

建立一个异常对象，是建立一个普通Object耗时的约20倍（实际上差距会比这个数字更大一些，因为循环也占用了时间，追求精确的读者可以再测一下空循环的耗时然后在对比前减掉这部分），而抛出、接住一个异常对象，所花费时间大约是建立异常对象的4倍。