异常概念

在我们日常生活中,有时会出现各种各样的异常,例如:职工小王开车去上班,在正常情况下,小王会准时到达单位。但是天有不测风云,在小王去上班时,可能会遇到一些异常情况,比如小王的车子出了故障,小王只能改为步行.

实际工作中，遇到的情况不可能是非常完美的。比如：你写的某个模块，用户输入不一定符合你的要求、你的程序要打开某个文件，这个文件可能不存在或者文件格式不对，你要读取数据库的数据，数据可能是空的等。我们的程序再跑着，内存或硬盘可能满了。等等。

软件程序在运行过程中，非常可能遇到刚刚提到的这些异常问题，我们叫异常，英文是：Exception，意思是例外。这些，例外情况，或者叫异常，怎么让我们写的程序做出合理的处理。而不至于程序崩溃。

异常指程序运行中出现的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。
异常发生在程序运行期间,它影响了正常的程序执行流程。

比如说，你的代码少了一个分号，那么运行出来结果是提示是错误`java.lang.Error`；如果你用`System.out.println(11/0)`，那么你是因为你用0做了除数，会抛出`java.lang.ArithmeticException`的异常。

异常发生的原因有很多，通常包含以下几大类：

- 用户输入了非法数据。
- 要打开的文件不存在。
- 网络通信时连接中断，或者JVM内存溢出。

这些异常有的是因为用户错误引起，有的是程序错误引起的，还有其它一些是因为物理错误引起的。

要理解Java异常处理是如何工作的，你需要掌握以下三种类型的异常：

- **检查性异常：**最具代表的检查性异常是用户错误或问题引起的异常，这是程序员无法预见的。例如要打开一个不存在文件时，一个异常就发生了，这些异常在编译时不能被简单地忽略。
- **运行时异常：** 运行时异常是可能被程序员避免的异常。与检查性异常相反，运行时异常可以在编译时被忽略。
- **错误：** 错误不是异常，而是脱离程序员控制的问题。错误在代码中通常被忽略。例如，当栈溢出时，一个错误就发生了，它们在编译也检查不到的。

异常指不期而至的各种状况，如：文件找不到、网络连接失败、除0操作、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语言在设计的当初就考虑到这些问题，提出异常处理的框架的方案，所有的异常都可以用一个异常类来表示，不同类型的异常对应不同的子类异常（目前我们所说的异常包括错误概念），定义异常处理的规范，在`JDK1.4`版本以后增加了异常链机制，从而便于跟踪异常。

Java异常是一个描述在代码段中发生异常的对象，当发生异常情况时，一个代表该异常的对象被创建并且在导致该异常的方法中被抛出，而该方法可以选择自己处理异常或者传递该异常。

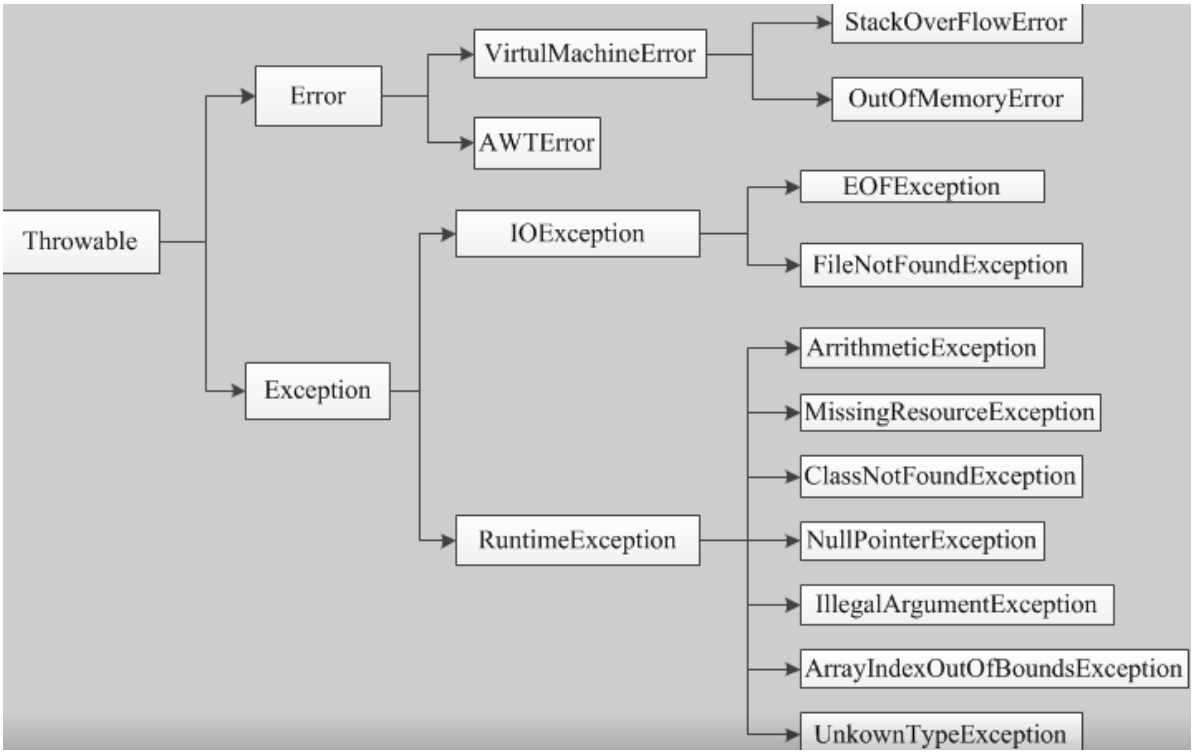
异常体系结构

Java把异常当作对象来处理，并定义一个基类`java.lang.Throwable`作为所有异常的超类。

在Java API中已经定义了许多异常类，这些异常类分为两大类，**错误Error和异常Exception。**

Java异常层次结构图：

狂神社群笔记资料，禁止外传，本人QQ：24736743



从图中可以看出所有异常类型都是内置类`Throwable`的子类，因而`Throwable`在异常类的层次结构的顶层。

接下来`Throwable`分成了两个不同的分支，**一个分支是Error，它表示不希望被程序捕获或者是程序无法处理的错误。另一个分支是Exception，它表示用户程序可能捕捉的异常情况或者说是程序可以处理的异常。**

其中异常类`Exception`又分为运行时异常(`RuntimeException`)和非运行时异常。Java异常又可以分为不受检查异常（`Unchecked Exception`）和检查异常（`Checked Exception`）。

异常之间的区别与联系

1、Error

`Error`类对象由 Java 虚拟机生成并抛出，大多数错误与代码编写者所执行的操作无关。

比如说：

Java虚拟机运行错误（`Virtual MachineError`），当JVM不再有继续执行操作所需的内存资源时，将出现`OutOfMemoryError`。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止；

还有发生在虚拟机试图执行应用时，如类定义错误（`NoClassDefFoundError`）、链接错误（`LinkageError`）。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。

对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在Java中，错误通常是使用`Error`的子类描述。

2、Exception

在`Exception`分支中有一个重要的子类`RuntimeException`（运行时异常），该类型的异常自动为你所编写的程序定义`ArrayIndexOutOfBoundsException`（数组下标越界）、`NullPointerException`（空指针异常）、

狂神社群笔记资料，禁止外传，本人QQ：24736743

`ArithmeticException`（算术异常）、`MissingResourceException`（丢失资源）、`ClassNotFoundException`（找不到类）等异常，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；而`RuntimeException`之外的异常我们统称为非运行时异常，类型上属于`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。

注意：`Error`和`Exception`的区别：`Error`通常是灾难性的致命的错误，是程序无法控制和处理的，当出现这些异常时，Java虚拟机（JVM）一般会选择终止线程；`Exception`通常情况下是可以被程序处理的，并且在程序中应该尽可能的去处理这些异常。

3、检查异常和不受检查异常

检查异常：在正确的程序运行过程中，很容易出现的、情理可容的异常状况，在一定程度上这种异常的发生是可以预测的，并且一旦发生该种异常，就必须采取某种方式进行处理。

解析：**除了RuntimeException及其子类以外，其他的Exception类及其子类都属于检查异常**，当程序中可能出现这类异常，**要么使用try-catch语句进行捕获，要么用throws子句抛出**，否则编译无法通过。

不受检查异常：包**括RuntimeException及其子类和Error。**

分析：`不受检查异常`为编译器不要求强制处理的异常，`检查异常`则是编译器要求必须处置的异常。

Java异常处理机制

java异常处理本质：抛出异常和捕获异常

1、抛出异常

要理解抛出异常，首先要明白什么是异常情形（exception condition），它是指阻止当前方法或作用域继续执行的问题。其次把异常情形和普通问题相区分，普通问题是指在当前环境下能得到足够的信息，总能处理这个错误。

对于异常情形，已经无法继续下去了，因为在当前环境下无法获得必要的信息来解决问题，你所能做的就是从当前环境中跳出，并把问题提交给上一级环境，这就是抛出异常时所发生的事情。抛出异常后，会有几件事随之发生。

首先，是像创建普通的java对象一样将使用`new`在堆上创建一个异常对象；然后，当前的执行路径（已经无法继续下去了）被终止，并且从当前环境中弹出对异常对象的引用。此时，异常处理机制接管程序，并开始寻找一个恰当的地方继续执行程序，

这个恰当的地方就是异常处理程序或者异常处理器，它的任务是将程序从错误状态中恢复，以使程序要么换一种方式运行，要么继续运行下去。

举例：

假使我们创建了一个学生对象Student的一个引用stu,在调用的时候可能还没有初始化。所以在使用这个对象引用调用其他方法之前，要先对它进行检查，可以创建一个代表错误信息的对象，并且将它从当前环境中抛出，这样就把错误信息传播到更大的环境中。

```
if(stu == null){
    throw new NullPointerException();
}
```

2、捕获异常

在方法抛出异常之后，运行时系统将转为寻找合适的异常处理器（exception handler）。潜在的异常处理器是异常发生时依次存留在调用栈中的方法的集合。当异常处理器所能处理的异常类型与方法抛出的异常类型相符时，即为合适的异常处理器。运行时系统从发生异常的方法开始，依次回查调用栈中的方法，直至找到含有合适异常处理器的方法并执行。当运行时系统遍历调用栈而未找到合适的异常处理器，则运行时系统终止。同时，意味着Java程序的终止。

注意：

对于`运行时异常`、`错误`和`检查异常`，Java技术所要求的异常处理方式有所不同

由于运行时异常及其子类的不可查性，为了更合理、更容易地实现应用程序，Java规定，**运行时异常将由Java运行时系统自动抛出，允许应用程序忽略运行时异常。**

对于方法运行中可能出现的`Error`，当运行方法不欲捕捉时，Java允许该方法不做任何抛出声明。因为，大多数`Error`异常属于永远不能被允许发生的状况，也属于合理的应用程序不该捕捉的异常。

```
对于所有的检查异常，Java规定：一个方法必须捕捉，或者声明抛出方法之外。也就是说，当一个方法选择不捕捉检查异常时，它必须声明将抛出异常。
```

3、异常处理五个关键字

分别是：`try`、`catch`、`finally`、`throw`、`throws`

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。

处理异常

1、try -catch

```
try{
    //code that might generate exceptions
}catch(Exception e){
    //the code of handling exception1
}catch(Exception e){
    //the code of handling exception2
}
```

要明白异常捕获，还要理解`监控区域`（guarded region）的概念。它是一段可能产生异常的代码，并且后面跟着处理这些异常的代码。

因而可知，上述`try-catch`所描述的即是监控区域，关键词`try`后的一对大括号将一块可能发生异常的代码包起来，即为监控区域。Java方法在运行过程中发生了异常，则创建异常对象。

将异常抛出监控区域之外，由Java运行时系统负责寻找匹配的`catch`子句来捕获异常。若有一个`catch`语句匹配到了，则执行该`catch`块中的异常处理代码，就不再尝试匹配别的`catch`块了。

匹配原则：如果抛出的异常对象属于`catch`子句的异常类，或者属于该异常类的子类，则认为生成的异常对象与`catch`块捕获的异常类型相匹配。

【演示】

```
public class TestException {
    public static void main(String[] args) {
        int a = 1;
        int b = 0;
        try { // try监控区域
            if (b == 0) throw new ArithmeticException(); // 通过throw语句抛出异常
            System.out.println("a/b的值是：" + a / b);
            System.out.println("this will not be printed!");
        }
        catch (ArithmeticException e) { // catch捕捉异常
            System.out.println("程序出现异常，变量b不能为0！");
        }
        System.out.println("程序正常结束。");
    }
}

//输出
程序出现异常，变量b不能为0！
程序正常结束。
```

注意：显示一个异常的描述，`Throwable`重载了`toString()`方法（由`Object`定义），所以它将返回一个包含异常描述的字符串。例如，将前面的`catch`块重写成：

```
catch (ArithmeticException e) { // catch捕捉异常
    System.out.println("程序出现异常"+e);
}
//输出
程序出现异常java.lang.ArithmeticException
程序正常结束。
```

算术异常属于运行时异常，因而实际上该异常不需要程序抛出，运行时系统自动抛出。如果不用try-catch程序就不会往下执行了。

【演示】

```
public class TestException {
    public static void main(String[] args) {
        int a = 1;
        int b = 0;
        System.out.println("a/b的值是：" + a / b);
        System.out.println("this will not be printed!");
    }
}


结果：
Exception in thread "main" java.lang.ArithmeticException: / by zero
at TestException.main(TestException.java:7)
```

使用多重的catch语句：很多情况下，由单个的代码段可能引起多个异常。处理这种情况，我们需要定义两个或者更多的`catch`子句，每个子句捕获一种类型的异常，当异常被引发时，每个`catch`子句被依次检查，第一个匹配异常类型的子句执行，当一个`catch`子句执行以后，其他的子句将被旁路。

编写多重catch语句块注意事项：

顺序问题：先小后大，即先子类后父类

注意：

Java通过异常类描述异常类型。对于有多个`catch`子句的异常程序而言，应该尽量将捕获底层异常类的`catch`子句放在前面，同时尽量将捕获相对高层的异常类的`catch`子句放在后面。否则，捕获底层异常类的`catch`子句将可能会被屏蔽。

嵌套try语句：`try`语句可以被嵌套。也就是说，一个`try`语句可以在另一个`try`块的内部。每次进入`try`语句，异常的前后关系都会被推入堆栈。如果一个内部的`try`语句不含特殊异常的`catch`处理程序，堆栈将弹出，下一个`try`语句的`catch`处理程序将检查是否与之匹配。这个过程将继续直到一个`catch`语句被匹配成功，或者是直到所有的嵌套`try`语句被检查完毕。如果没有`catch`语句匹配，Java运行时系统将处理这个异常。

【演示】

```
class NestTry{
    public static void main(String[] args){
        try{
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = "+ a);
            try{
                if(a == 1){
                a = a/(a-a);
                }
                if(a == 2){
                    int c[] = {1};
                    c[42] =99;
                }
            }catch(ArrayIndexOutOfBoundsException e){
                System.out.println("ArrayIndexOutOfBounds :"+e);
            }
        }catch(ArithmeticException e){
            System.out.println("Divide by 0"+ e);
        }
    }
```

```
}

//分析运行：
D:\java>java NestTry one

a = 1

Divide by 0java.lang.ArithmeticException: / by zero

D:\java>java NestTry one two

a = 2

ArrayIndexOutOfBounds :java.lang.ArrayIndexOutOfBoundsException: 42
```

分析：正如程序中所显示的，该程序在一个try块中嵌套了另一个`try`块。程序工作如下：当你在没有命令行参数的情况下执行该程序，外面的`try`块将产生一个被0除的异常。

程序在有一个命令行参数条件下执行，由嵌套的`try`块产生一个被0除的异常，由于内部的`catch`块不匹配这个异常，它将把异常传给外部的`try`块，在外部异常被处理。如果你在具有两个命令行参数的条件下执行该程序，将由内部`try`块产生一个数组边界异常。

注意：当有方法调用时，`try`语句的嵌套可以很隐蔽的发生。例如，我们可以将对方法的调用放在一个`try`块中。在该方法的内部，有另一个`try`语句。

在这种情况下，方法内部的`try`仍然是嵌套在外部调用该方法的`try`块中的。下面我们将对上述例子进行修改，嵌套的`try`块移到方法nesttry()的内部：结果依旧相同！

```
class NestTry{
    static void nesttry(int a){
        try{
            if(a == 1){
                a = a/(a-a);
            }
            if(a == 2){
                int c[] = {1};
                c[42] =99;
            }
        }catch(ArrayIndexOutOfBoundsException e){
            System.out.println("ArrayIndexOutOfBounds :"+e);
        }
    }
    public static void main(String[] args){
        try{
            int a = args.length;
            int b = 42 / a;
            System.out.println("a = "+ a);
            nesttry(a);
        }catch(ArithmeticException e){
            System.out.println("Divide by 0"+ e);
        }
    }
}
```

2、thorw

到目前为止，我们只是获取了被Java运行时系统引发的异常。然而，我们还可以用`throw`语句抛出明确的异常。

语法形式：

```
throw ThrowableInstance;
```

这里的ThrowableInstance一定是`Throwable`类类型或者`Throwable`子类类型的一个对象。简单的数据类型，例如`int`，`char`,以及非`Throwable`类，例如`String`或`Object`，不能用作异常。

有两种方法可以获取`Throwable`对象：在`catch`子句中使用参数或者使用`new`操作符创建。程序执行完`throw`语句之后立即停止；`throw`后面的任何语句不被执行，最邻近的`try`块用来检查它是否含有一个与异常类型匹配的`catch`语句。

如果发现了匹配的块，控制转向该语句；如果没有发现，次包围的`try`块来检查，以此类推。如果没有发现匹配的`catch`块，默认异常处理程序中断程序的执行并且打印堆栈轨迹。

```
class TestThrow{
    static void proc(){
        try{
            throw new NullPointerException("demo");
        }catch(NullPointerException e){
            System.out.println("Caught inside proc");
            throw e;
        }
    }

    public static void main(String [] args){
        try{
            proc();
        }catch(NullPointerException e){
            System.out.println("Recaught: "+e);
        }
    }
}
```

该程序两次处理相同的错误，首先，`main()`方法设立了一个异常关系然后调用proc()。proc()方法设立了另一个异常处理关系并且立即抛出一个`NullPointerException`实例，`NullPointerException`在`main()`中被再次捕获。

该程序阐述了怎样创建Java的标准异常对象，特别注意这一行：

```
throw new NullPointerException("demo");
```

分析：此处`new`用来构造一个`NullPointerException`实例，所有的Java内置的运行时异常有两个构造方法：一个没有参数，一个带有一个字符串参数。

当用第二种形式时，参数指定描述异常的字符串。如果对象用作`print()`或者`println()`的参数时，该字符串被显示。这同样可以通过调用getMessage()来实现，getMessage()是由`Throwable`定义的。

3、throws

如果一个方法可以导致一个异常但不处理它，它必须指定这种行为以使方法的调用者可以保护它们自己而不发生异常。要做到这点，我们可以在方法声明中包含一个`throws`子句。

一个`throws`子句列举了一个方法可能引发的所有异常类型。这对于除了`Error`或`RuntimeException`及它们子类以外类型的所有异常是必要的。一个方法可以引发的所有其他类型的异常必须在`throws`子句中声明，否则会导致编译错误。

```
public void info() throws Exception
{
   //body of method
}
```

Exception 是该方法可能引发的所有的异常,也可以是异常列表，中间以逗号隔开。

【例子】

```
class TestThrows{
    static void throw1(){
        System.out.println("Inside throw1 . ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args){
        throw1();
    }
}
```

上述例子中有两个地方存在错误，你能看出来吗?

该例子中存在两个错误，首先，throw1()方法不想处理所导致的异常，因而它必须声明`throws`子句来列举可能引发的异常即`IllegalAccessException`；其次，`main()`方法必须定义`try/catch`语句来捕获该异常。

正确例子如下:

```
class TestThrows{
    static void throw1() throws IllegalAccessException {
        System.out.println("Inside throw1 . ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String[] args){
        try {
            throw1();
        }catch(IllegalAccessException e ){
            System.out.println("Caught " + e);
        }
    }
}
```

`throws`抛出异常的规则:

- 如果是不受检查异常（`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检查异常（`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。

狂神社群笔记资料，禁止外传，本人QQ：24736743

4、finally

当异常发生时，通常方法的执行将做一个陡峭的非线性的转向，它甚至会过早的导致方法返回。例如，如果一个方法打开了一个文件并关闭，然后退出，你不希望关闭文件的代码被异常处理机制旁路。`finally`关键字为处理这种意外而设计。

`finally`创建的代码块在`try/catch`块完成之后另一个`try/catch`出现之前执行。`finally`块无论有没有异常抛出都会执行。如果抛出异常，即使没有`catch`子句匹配，`finally`也会执行。

一个方法将从一个`try/catch`块返回到调用程序的任何时候，经过一个未捕获的异常或者是一个明确的返回语句，`finally`子句在方法返回之前仍将执行。这在关闭文件句柄和释放任何在方法开始时被分配的其他资源是很有用。

注意：`finally`子句是可选项，可以有也可以无，但是每个`try`语句至少需要一个`catch`或者`finally`子句。

【例子】

```
class TestFinally{
    static void proc1(){
        try{
            System.out.println("inside proc1");
            throw new RuntimeException("demo");
        }finally{
            System.out.println("proc1's finally");
        }
    }
    static void proc2(){
        try{
            System.out.println("inside proc2");
            return ;
        } finally{
            System.out.println("proc2's finally");
        }
    }
    static void proc3(){
        try{
            System.out.println("inside proc3");
        }finally{
            System.out.println("proc3's finally");
        }
    }
    public static void main(String [] args){
        try{
            proc1();
        }catch(Exception e){
            System.out.println("Exception caught");
        }
        proc2();
        proc3();
    }
}


结果：
inside proc1
```


狂神社群笔记资料，禁止外传，本人QQ：24736743

```
proc1's finally

Exception caught

inside proc2

proc2's finally

inside proc3

proc3's finally
```

注：如果`finally`块与一个`try`联合使用，`finally`块将在`try`结束之前执行。

try, catch,finally ,return 执行顺序

1．执行try，catch， 给返回值赋值

2．执行finally

3．return

自定义异常

使用Java内置的异常类可以描述在编程时出现的大部分异常情况。除此之外，用户还可以自定义异常。用户自定义异常类，只需继承`Exception`类即可。

在程序中使用自定义异常类，大体可分为以下几个步骤:

- 创建自定义异常类。
- 在方法中通过`throw`关键字抛出异常对象。
- 如果在当前抛出异常的方法中处理异常，可以使用`try-catch`语句捕获并处理；否则在方法的声明处通过`throws`关键字指明要抛出给方法调用者的异常，继续进行下一步操作。
- 在出现异常方法的调用者中捕获并处理异常。

【举例】

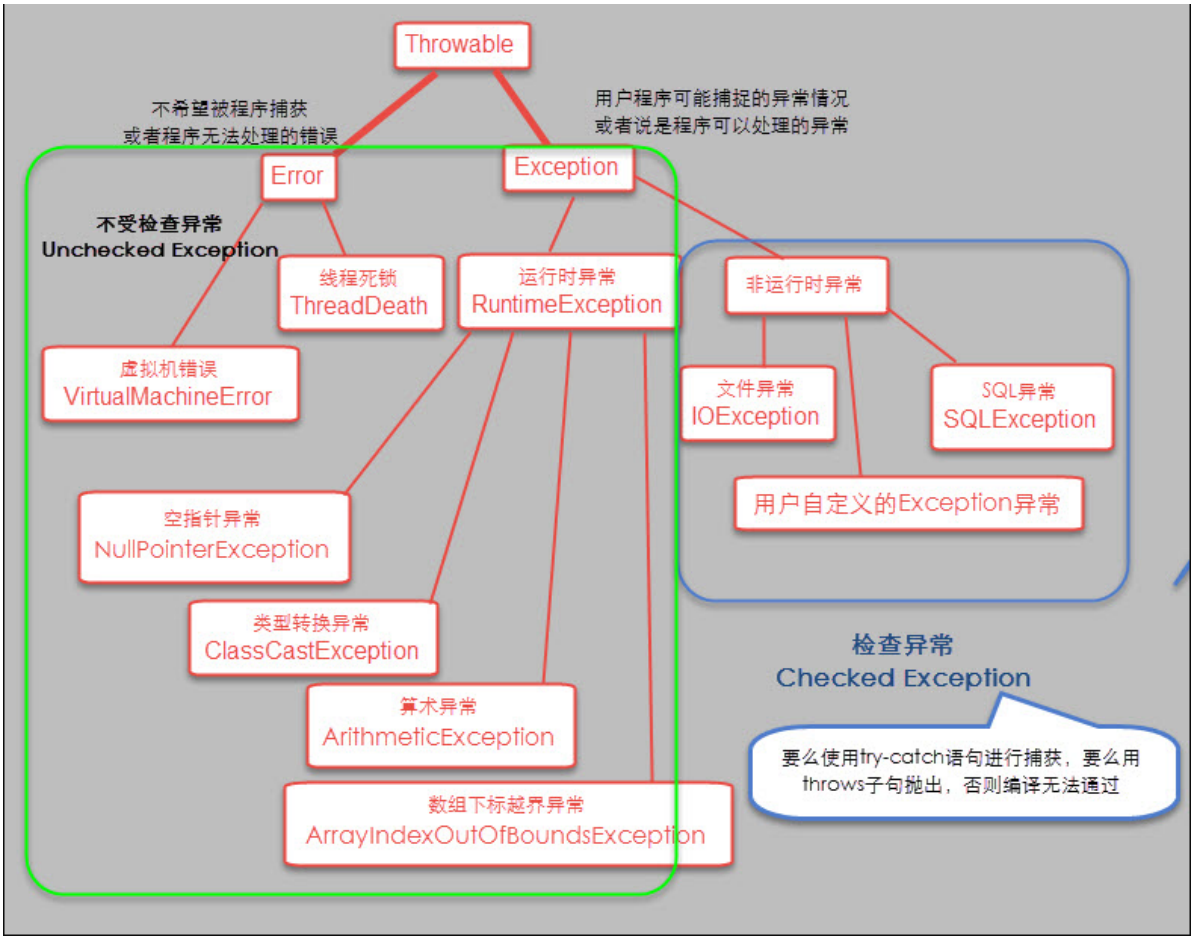
```
class MyException extends Exception {
    private int detail;
    MyException(int a){
        detail = a;
    }
    public String toString(){
        return "MyException ["+ detail + "]";
    }
}
public class TestMyException{
    static void compute(int a) throws MyException{
        System.out.println("Called compute(" + a + ")");
        if(a > 10){
            throw new MyException(a);
        }
        System.out.println("Normal exit!");
    }
    public static void main(String [] args){
```

```
19          try{
20              compute(1);
21              compute(20);
22          }catch(MyException me){
23              System.out.println("Caught " + me);
24          }
25      }
26  }
```

【结果】

```
1  Called compute(1)
2  Normal exit!
3  Called compute(20)
4  Caught MyException [20]
```

总结



开发中

狂神社群笔记资料，禁止外传，本人QQ：24736743

实际应用中的经验与总结

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，切忌只是简单的调用printStackTrace()去打印输出

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

狂神社群笔记资料，禁止外传，本人QQ：24736743