# Java异常总结

Throwable类是Java语言中所有错误或异常的超类。只有当对象是此类（或其子类之一）的实例时，才能通过 Java 虚拟机或者Java的throw语句抛出。类似地，只有此类或其子类之一才可以是catch子句中的参数类型。

在Throwable的下一层，立即分解为两个分支：Error和Exception。

Error类层次结构描述了Java运行时系统的内部错误和资源耗尽错误，应用程序不应该抛出该种类型的对象。如果出现这样的内部错误，除了通知用户，并尽力使程序安全中止外，再也无能为力了。不过这种情况很少出现。

Exception下又分为两个分支：一个分支是派生于RuntimeException的异常；另一个分支包含其他异常。划分两个分支的规则是：由于程序错误导致的异常属于RuntimeException；曾经能够正确运行，而由于某些情况（例如，I/O错误）导致的异常不属于RuntimeException。

派生于RuntimeException的异常包含下面几种情况：A.错误的类型转换；B.数组访问越界；C.访问空指针。不派生于RuntimeException的异常包括：A.试图在文件尾部后面读取数据；B.试图打开一个错误格式的URL；C.试图根据给定的字符串查找Class对象，而这个字符串表示的类并不存在。

“如果出现RuntimeException异常，就一定是你的问题”这是一条相当有道理的规则。

Java语言规范将派生于RuntimeException类或Error类的所有异常称为“未检查（unchecked）异常”，其他的异常称为“已检查（checked）异常”。编译器将检查是否为所有已检查异常提供异常处理器。（RuntimeException这个名字很容易让人混淆，实际上，我们现在讨论的所有错误都发生在运行时刻）

Java中的异常层次结构图

在自己编写方法时，不必将所有可能抛出的对象都进行声明。至于什么时候需要在方法中用throws子句声明异常，什么异常必须使用throws子句声明，需要记住在遇到下面4中情况时才会抛出异常：

1. 调用一个抛出已检查异常的方法。
2. 在程序运行过程中发现错误，并且利用throw语句抛出一个已检查异常。
3. 程序出现错误，例如，a[-1] = 0 会抛出ArrayIndexOutOfBoundsException异常。
4. Java虚拟机和运行时库出现的内部异常。

如果出现前两种情况之一，则必须告诉调用这个方法的程序员有可能抛出异常。

在异常抛出声明throws语句后面不需要什么Java的内部错误，就是从Error继承的那些异常。任何程序代码都具有抛出这类异常的潜能，而我们对它们却没有任何控制能力。总之，一个方法必须什么所有可能抛出的“已检查”异常，而“未检查”异常要么不可控制（Error），要么就应该避免它们的发生（RuntimeException）。

注意，如果在子类中覆盖了超类的一个方法，那么子类方法中声明的已检查异常不能超过超类方法中声明的异常范围。（也就是说，子类方法中抛出的异常范围更加小，或者根本不抛出任何异常。）如果超类方法没有抛出任何已检查异常，那么子类也不能抛出任何已检查异常。

如果某个异常发生的时候没有在任何地方捕获它，程序就会终止执行，并在控制台上显示异常信息，其中包括异常的类型和堆栈的内容。

try语句可以只有finally子句，而没有catch子句。不管是否有异常被捕获，finally子句中的代码都会被执行。

强烈建议独立使用try/catch和try/finally语句块。这样可以提高代码清晰度。例如：

InputStream in = **new** FileInputStream(**new** File(FILE\_PATH));

**try** {

**try** {

// 这里包含可能会抛出异常的代码

**throw** **new** IOException();

} **finally** {

in.close();

}

}**catch** (Throwable e) {

// 这里可以捕获在finally语句中的异常

StackTraceElement[] stackArray = e.getStackTrace();

// StackTraceElement类中包含有获取文件名和当前执行代码行号的方法，

// 同时还含有获取类名和方法名的方法。

**for**(StackTraceElement stackElement : stackArray) {

System.*out*.println("异常发生在类：" + stackElement.getClassName() + "#" +

stackElement.getMethodName() +"， 行号：" + stackElement.getLineNumber());

}

}

在内层的try语句块只有一个职责，就是确保关闭输入流。在外层的try语句块也只有一个职责，就是保证报告出现的错误。这种解决方案不仅清楚，而且还具有一个功能，就是报告finally子句中出现的错误。

当finally子句包含return语句时，将会出现一种意想不到的结果。假设利用return语句从try语句块中退出。在方法返回之前，finally语句块的内容将会被执行。如果finally语句块中也包含一个return语句，那么这个返回值将会掩盖原始的返回值。

有时候，finally子句也会带来麻烦。例如，回收资源的方法也有可能抛出异常。一个典型的例子是关闭流。假设希望能够确保在流处理代码中遇到异常时将流关闭。

InputStream in = **new** FileInputStream(**new** File(FILE\_PATH));

**try** {

// 这里包含可能会抛出异常的代码

**throw** **new** IOException();

}**catch** (IOException e) {

// 这里可以捕获在finally语句中的异常

System.*out*.println(e.getStackTrace());

} **finally** {

in.close();

}

现在假设在try语句块中的代码抛出了一些非IOException的异常,这些异常只有这个方法的调用者才能够给与处理，那么将执行finally语句块，调用close方法。而close方法本身也有可能抛出IOException异常。当出现这种情况时，原始的异常会丢失，转而抛出IOException异常。这并不是异常处理机制所希望的结果。解决这个问题的最好方法是在finally子句中执行清除操作时不要抛出异常，但是InputStream类的设计者并没有这样做。

使用异常的几点建议：

1).不要只抛出RuntimeException异常，应该寻找更加适当的子类或创建自己的异常类。2).不要只捕获Throwable异常，否则会使程序代码更加难读、更难维护。

3).研究一下已检查异常与未检查异常的区别。已检查异常本来就很庞大，这里不会抛出由于逻辑错误而造成的异常。

4).将一种异常转换成另一种更加合适的异常时不要犹豫。例如，在解析某个文件中的一个整数时捕获到NumberFormatException异常，然后将他转换成IOException或其他自定义异常的子类