Java异常

## 何时使用异常

程序出错时至少应该做的处理是：

1. 向用户通报错误；
2. 保存所有工作结果；
3. 以妥善的形式推出程序。

异常处理主要用于运行时错误（也就是程序执行时发生的错误）的处理方面。这些错误可能是数学运算溢出，磁盘不足或者文件损坏等非正常情况。当这些错误发生时，程序应该要么去尝试恢复程序的状态，要么就直接通知用户发生了异常，异常处理的任务就是将控制权从错误产生的地方转移给能够处理这种情况的错误处理器。

使用异常的基本前提是要考虑周全某段程序可能会出现的错误，只对那些不在程序正常功能所能管理的范围之内的问题（I/O错误等）使用异常机制处理。

异常应于何时抛出？答案归于一条原则：

如果方法遇到一个不知道如何处理的意外情况（abnormal condition），那么它应该抛出异常。不幸的是，虽然这条原则易于记忆和引用，但是它并不十分清晰。实际上，它引出了另一个的问题：什么是意外情况？  
　　一个更有用的经验法则是：  
　　在有充足理由将某情况视为该方法的典型功能（typical functioning ）部分时，避免使用异常。因此，意外情况就是指方法的“正常功能”（normal functioning）之外的情况。

## 异常表示没有遵守契约

从另一个角度来看待异常，视之为“没有遵守契约”，你可能对应当怎样使用异常有更深层的理解。面向对象程序设计中经常讨论的一个设计方法是契约设计，它指出方法是客户（方法的调用者）和声明方法的类之间的契约。这个契约包括客户必须满足的前置条件（precondition）和方法本身必须满足的后置条件（postcondition）。  
**前置条件**　　String类的charAt(int index)方法是一个带有前置条件的方法。这个方法规定客户传入的index参数的最小取值是0，最大取值是在该String对象上调用length()方法的结果减去1。也就是说，如果字符串长度为5，那么index参数的取值限于0、1、2、3、4。  
**后置条件**  
　　String类的charAt(int index)方法的后置条件要求返回值必须是该字符串对象在index位置上的字符数据，而且该字符串对象必须保持不变。  
　　如果客户调用charAt()并传入-1、和length()一样大或者更大的值，那就认为客户没有遵守契约。这种情况下，charAt()方法是不能正确执行的，它将抛出异常StringIndexOutOfBoundsException。该异常指出客户程序中存在某种缺陷或String类使用不当。  
　　如果charAt()方法接收的输入没有问题（客户遵守了契约），但是由于某种原因它无法返回指定的索引上的字符数据（没有满足后置条件），它将抛出异常来指示这种情况。这种异常指出方法的实现中包含缺陷或者方法在获得运行时资源上存在问题。  
　　因此，如果一个事件表示了“异常条件”或者“没有遵守契约”，那么，Java程序所要做的就是抛出异常。

## 运行时异常和受检查异常

运行时异常表示无法让程序恢复运行的异常，导致这种异常的原因通常是由于执行了错误的操作（错误的类型转换、数组访问越界、访问空指针）。一旦出现运行时异常，建议让程序终止。

受检查异常表示程序本身没有问题，但是由于像I/O错误等导致的可以处理的异常（试图在文件尾部后面读取数据、试图打开一个不存在的文件、试图用一个字符串查找一个不存在的类）。如果抛出异常的方法本身不处理或者不能处理它，那么方法的调用者就必须去处理该异常，否则调用会出错，连编译也无法通过。

**最佳解决方案:**对于运行时异常，我们不要用try...catch来捕获处理，而是在程序开发调试阶段，尽量去避免这种异常，一旦发现该异常，正确的做法就会改进程序设计的代码和实现方式，修改程序中的错误，从而避免这种异常。捕获并处理运行时异常是好的解决办法，因为可以通过改进代码实现来避免该种异常的发生。

对于受检查异常，没说的，老老实实去按照异常处理的方法去处理，要么用try...catch捕获并解决，要么用throws抛出！

## Java异常处理的原则和技巧

**异常设计的几个原则：   
1.如果方法遭遇了一个无法处理的意外情况，那么抛出一个异常。   
2.避免使用异常来指出可以视为方法的常用功能的情况。   
3.如果发现客户违反了契约（例如，传入非法输入参数），那么抛出非检查型异常。   
4.如果方法无法履型契约，那么抛出检查型异常，也可以抛出非检查型异常。   
5.如果你认为客户程序员需要有意识地采取措施，那么抛出检查型异常。   
6.异常类应该给客户提供丰富的信息，异常类跟其它类一样，允许定义自己的属性和方法。**

**7.异常类名和方法遵循JAVA类名规范和方法名规范 。  
8.跟JAVA其它类一样，不要定义多余的方法和变量。**

1. **避免过大的try块，不要把不会出现异常的代码放到try块里面，尽量保持一个try块对应一个或多个异常。**
2. **细化异常的类型，不要不管什么类型的异常都写成Excetpion。**
3. **catch块尽量保持一个块捕获一类异常，不要忽略捕获的异常，捕获到后要么处理，要么转译，要么重新抛出新类型的异常。**
4. **不要把自己能处理的异常抛给别人。**
5. **不要用try...catch参与控制程序流程，异常控制的根本目的是处理程序的非正常情况。**

## ****异常处理的最佳实践****

**1）为可恢复的错误使用检查型异常，为编程错误使用非检查型错误。**

选择检查型还是非检查型异常，对于Java编程人员来说，总是让人感到困惑。检查型异常保证你对错误条件提供异常处理代码，这是一种从语言到强制你编写健壮的代码的一种方式，但同时会引入大量杂乱的代码并导致其不可读。当然，如果你有替代品和恢复策略的话，捕捉异常并做些什么看起来似乎也在理。

**2）在finally程序块中关闭或者释放资源**

这在Java编程中，是一个广为人知的最佳实践，在处理网络和IO类的时候，相当于一个标准。在finally块中关闭资源， 在正常和异常执行的情况下，保证之前和稀缺资源的合理释放，这由finally块保证。强烈建议独立使用try/catch子句和try/finally子句，这样有利于提高代码清晰度：

try

{

try

{

}

Finally

{

}

}

catch (Exception e)

{

}

**3）在堆栈跟踪中包含引起异常的原因**

很多时候，当一个由另一个异常导致的异常被抛出的时候，日志记录和打印根异常就变得非常重要。Java异常类提供了 getCause()方法来检索导致异常的原因，这些（原因）可以对异常的根层次的原因提供更多的信息。该Java实践对在进行调试或排除故障大有帮助。时刻记住，如果你将一个异常包装成另一种异常时，构造一个新异常要传递源异（调用initCause()方法）。

**4）始终提供关于异常的有意义的完整的信息**

异常信息是最重要的地方，因为这是程序员首先看到的第一个地方，这里你能找到问题产生的根本原因。这里始终提供精确的真实的信息。例如，对比IllegalArgumentException 异常的两条异常信息：

消息 1: "Incorrect argument for method"

消息 2: "Illegal value for ${argument}: ${value}

第一条消息仅说明了参数是非法的或者不正确，但第二条消息包括了参数名和非法值，而这对于找到错误的原因是很重要的。在用Java编程中编写异常处理代码的时候，始终遵循该Java最佳实践。

**5）避免过度使用检查型异常**

检查型异常在强制执行方面有一定的优势，但同时它也破坏了代码，通过掩盖业务逻辑使代码可读性降低。只要你不过度使用检查型异常，你可以最大限度的减少这类情况，这样做的结果是你会得到更清洁的代码。

**6）将检查型异常转为运行时异常**

这是在像Spring之类的多数框架中用来限制使用检查型异常的技术之一，大部分出自于JDBC的检查型异常，都被包装进DataAccessException中，而（DataAccessException）异常是一种非检查型异常。这是Java最佳实践带来的好处，特定的异常限制到特定的模块，像 SQLException 放到DAO层，将意思明确的运行时异常抛到客户层。

**7）记住对性能而言，异常代价高昂**

需要记住的一件事是异常代价高昂，同时让你的代码运行缓慢。假如你有方法从ResultSet（结果集）中进行读取，这时常会抛出SQLException异常而不会移到下一元素，这将会比不抛出异常的正常代码执行的慢的多。因此最大限度的减少不必要的异常捕捉和移动，那里没有什么固定的原因。不要仅仅是抛出和捕捉异常，如果你能使用boolean变量去表示执行结果，可能会得到更整洁，更高性能的解决方案。修正错误的根源，避免不必须要的异常捕捉。

**8）避免catch块为空**

没有什么比空的catch块更糟糕的了，因为它不仅隐藏了错误和异常，同时可能导致你的对象处于不可使用或者脏的状态。空的catch块只能变得无意义，如果你非常肯定异常不会继续以任何方式影响对象状态，但在程序执行期间，用日志记录错误依然是最好的（方法）。对于在Java编程中编写异常处理代码，这不仅仅是一个Java最佳实践，而是一个最通用的实践。

**9）使用标准异常**

我们的第九条最佳实践建议使用标准和内置的Java异常。使用标准异常而不是每次创建我们自己的异常，对于维护性和一致性，不管是现在还是以后，都是最好的选择。重用标准异常使代码更具可读性，因为大部分Java开发人员对标准的像源自于JDK的RuntimeException 异常，IllegalStateException 异常，IllegalArgumentException 异常或者 NullPointerException异常，（开发者）他们能一眼就知道每种异常的目的，而不是在代码里查找或者在文档里查找用户定义的异常的目的。

**10）记录任何方法抛出的异常**

Java提供了throw和throws关键字来抛出异常，在javadoc中用@throw记录任何方法可能会抛出的异常。如果你编写API或者公共接口，这就变得非常重要。任何方法抛出的异常都有相应的文档记录，这样你就能下意识的提醒任何使用（该方法）的人。