异常使用规则

## 只针对异常的情况才使用异常

异常是为了在异常的情况下使用而设计的，不要将它们用于普通的控制流，也不要编写迫使它们这么做的API。

### 不要用于普通的控制流

基于异常的循环模式不仅模糊了代码的意图，降低了它的性能，而且还不能保证正常工作。如果出现了不相关的Bug，这个模式会悄悄失效，从而掩盖了这个Bug。

### 不要编写迫使它们这么做的API

如果类具有“状态相关”的方法，即只有在特定的不可预知的条件下才可以被调用的方法，这个类往往也应该有一个单独的“状态测试”方法，即指示是否可以调用这个状态相关的方法。如Iterator类的hasNext和next方法。

另一种提供单独的状态测试的方法的做法是，如果“状态相关的”方法被调用时，该对象处于不适当的状态之中，它就会返回一个可识别的值，比如null。

对于“状态测试方法”和“可识别的返回值”这两种做法，有一些可供参考的指导原则。如果对象将在缺少外部同步的情况下被并发访问，或者可被外界改变状态，使用可识别的返回值可能是很有必要的，因为在调用“状态测试”方法和调用对应的“状态相关”方法的时间间隔之中，对象的状态有可能发生变化。如果单独的“状态测试”方法必须重复“状态相关”方法的工作，从性能的角度考虑，就应该使用可识别的返回值。如果所有其他方面都是等同的，那么“状态测试”方法则略优于可识别的返回值。它提供了更好的可读性，对于使用不当的情形，可以更加易于检测和更正：如果忘了去调用状态测试方法，状态相关的方法就会抛出异常，使这个Bug变得很明显；如果忘了去检查可识别的返回值，这个Bug就很难被发现。

## 对可恢复的情况使用受检异常，对编程错误使用运行时异常

Java提供了三种可抛出结构：受检的异常、运行时异常和错误。

如果期望调用者能够适当的恢复，对于这种情况就应该使用受检的异常。通过抛出受检的异常，强迫调用者在一个catch子句中处理该异常，或者将它传播出去。调用者可以忽视这样的强制要求，只需捕获异常并忽略，但不建议这么做。

运行时异常和错误在行为上是等同的：它们都是不需要也不应该被捕获的可抛出结构。抛出它们，往往就属于不可恢复的情形，继续执行下去有害无益。运行时异常用来表明编程错误，大部分是前提违例，如数组越界。错误往往被JVM保留用于表示资源不足、约束失败或者其他程序无法继续执行的条件。这作为一个普遍惯例，因此，你实现的所有未受检的抛出结构都应该是RuntimeException的子类。

## 避免不必要的使用受检的异常

如果方法抛出一个或多个受检的异常，调用该方法的代码就必须在一个或多个catch块中处理这些异常，或者它必须声明它抛出这些异常，并让它们传播出去。无论哪种方法，都给程序员增添了不可忽视的负担。

如果正确的使用API并不能阻止这种异常条件的产生，并且一旦产生异常，使用API的程序员可以立即采用有用的动作，这种负担就被认为是应当的。除非这两个条件都成立，否则更适合使用未受检的异常。

## 优先使用标准的异常

常用的异常：

|  |  |
| --- | --- |
| 异常 | 使用场合 |
| IllegalArgumentsException | 非null的参数值不正确 |
| IllegalStateException | 对于方法调用而言，对象状态不合适 |
| NullPointerException | 在禁止使用null的情况下参数值为null |
| IndexOutOfBoundsException | 下标参数值越界 |
| ConcurrentModificationException | 在禁止并发修改的情况下，检测到对象的并发修改 |
| UnsupportedOperationException | 对象不支持用户请求的方法 |

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，而不是建立在名称的基础上。而且，如果希望稍微增加更多的失败-捕获信息，可以放心地把现有的异常进行子类化。

## 抛出与抽象相对应的异常

如果方法抛出的异常与它所执行的任务没有明显的联系，这种情形将会使人不知所措。当方法传递由底层抽象抛出的异常时，往往会发生这种情况。除了让人困惑之外，这也让实现细节污染了更高层的API。

为了避免这个问题，更高层的实现应该捕获低层的异常，同时抛出可以按照高层抽象进行解释的异常。即异常转译。

一种特殊的异常转译形式称为异常链，如果低层的异常对于调试导致高层异常的问题非常有帮助，使用异常链就很合适。低层的异常被传到高层的异常，高层的异常提供访问方法(Throwable.getCause)来获得低层的异常:

try {

… // Use lower-level abstraction to do our bidding

} catch(LowerLevelException cause) {

throw new HigherLevelException(cause);

}

高层异常的构造器将原因传到支持链的超级构造器，因此它最终将被传给Throwable的其中一个运行异常链的构造器，例如Throwable：

class HigherLevelException extends Exception {

HigherLevelException(Throwable cause) {

super(cause);

}

}

大多数标准的异常都有支持链的构造器。对于没有支持链的异常，可以利用Throwable的initCause方法设置原因。异常链不仅让你可以通过程序访问原因，还可以将原因的堆栈轨迹集成到更高层的异常中。

异常转译也不能被滥用。处理来自低层异常的最好做法是，在调用低层方法之前确保它们会成功执行，从而避免它们抛出异常。有时候，可以在给低层传递参数之前，检查更高层方法的参数的有效性，从而避免低层方法抛出异常。

如果无法避免低层异常，次选方案是，让更高层来悄悄绕开这些异常，从而将高层方法的调用者与低层的问题隔离开来。在这种情况下，可以用某种适当的记录机制将异常记录下来。这样有助于管理员调查问题，同时又将客户端代码和最终用户与问题隔离开来。

## 每个方法抛出的异常都要有文档

要为你编写的每个方法所能抛出的每个异常建立文档。对于未受检和受检的异常，以及对于抽象的和具体的方法也都一样。要为每个受检异常提供单独的throws子句，不要为未受检异常提供throws子句。如果没有为可以抛出的异常建立文档，其他人很难或者根本不可能有效的使用你的类和接口。

利用Javadoc的@throws标记，准确记录下抛出每个异常的条件。如果一个方法可能抛出多个异常类，不要使用“快捷方式”声明它会抛出这些异常类的某个超类，如“throws Exception”或“throws Throwable”。

## 在细节消息中包含能捕获失败的消息

异常的细节信息应该能捕获住失败，便于以后分析。为了捕获失败，异常的细节信息应该包含所有“对该异常有贡献”的参数和域的值。

异常的细节信息不应该与“用户层次的错误信息”混为一谈，后者对于最终用户而言必须是可理解的。与用户层次的错误信息不同，异常的字符串表示法主要是让程序员或者域服务人员用来分析失败的原因。因此，信息的内容比可理解性重要得多。

为了确保在异常的细节信息中包含足够的能捕获失败的信息，一种方法是在异常的构造器而不是字符串细节信息中引入这些信息。然后，有了这些信息，只要把它们放到信息描述中，就可以自动产生细节信息。

## 努力使失败保持原子性

一般而言，失败的方法调用应该使对象保持在被调用之前的状态。具有这种属性的方法被称为具有失败原子性。

最简单的方法是设计一个不可变的对象。如果对象是不可变的，失败原子性就是显然的。如果一个操作失败了，它可能会阻止创建新的对象，但是永远也不会使已有的对象保持在不一致的状态一种。

对于在可变对象上执行操作的方法，获得失败原子性最常见的方法是，在执行操作之前检查参数的有效性。这可以使得在对象的状态被修改之前，先抛出适当的异常。

另外一种类似的办法是，调整计算处理过程的顺序，使得任何可能会失败的计算部分都在对象状态被修改之前发生。

第三种办法是编写一段恢复代码，由他来拦截操作过程中发生的失败，以及使对象回滚到操作开始之前的状态上。

最后一个办法是，在对象的一份临时拷贝上执行操作，当操作完成之后在用临时拷贝中的结果代替对象的内容。

一般而言，作为方法规范的一部分，产生的任何异常都应该让对象保持在该方法调用之前的状态。如果违反了这条规则，API文档就应该清楚地指明对象将会处于什么样的状态。

## 不要忽略异常

使用空的catch块忽略异常会使异常达不到应有的目的。至少catch块也应该包含一条声明，解释为什么可以忽略这个异常。

有一种情形可以忽略异常，即关闭FileInputStream的时候。因为你还没有改变文件的状态，因此不必执行任何恢复动作，并且已经从文件中读取到所需的信息，因此不必终止正在进行的操作。即使在这种情况下，把异常记录下来还是明智的做法，因为如果这些异常经常发生，就可以调查异常的原因。