异常

57、只针对异常的情况才使用异常

先看两段代码

```
// 基于异常的模式
String t1[]={"11","22"};
try{
    int i=0;
    while(true){
        System.out.println(t1[i]);
        i++;
    }
}catch(ArrayIndexOutOfBoundsException e){

}
//标准模式
for(String t:t1){
    System.out.println(t);
}
```

为什么有人优先使用异常模式，而不是标准模式?

因为他们企图利用 java 的错误判断机制来提高性能，因为 VM 对每次数组访问都要检查越界情况，所以正常的终止测试被认为是多余的。这种想法有三个错误:

a.异常机制的设计初衷是用于不正常的情形，所有很少有 jvm 实现试图对它们进行优化，使得与显示的测试一样快速

b.把代码放在 try-catch 中反而阻止了现代 jvm 实现本来可能要执行的特定优化

c.对数组进行变量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。

基于异常的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，例如下面的代码:

```
public static void main(String[] args) {
    String t1[]={"11","22"};
    try{
        int i=0;
        while(true){
            System.out.println(t1[i]);
            test();
            i++;
        }
        }catch(ArrayIndexOutOfBoundsException e){

        }
    }
    private static void test(){
        String t2[]={};
        t2[0].toString();
    }
```

使用了异常的模式，当产生了与这个 bug 相关的异常将会被捕捉到，并且被错误的解释为正常的循环终止条件。

设计良好的 API 不应该强迫它的客户端为了正常的控制流而使用异常，有两种良好的 API 设计

(1).状态测试，指示是否可以调用这个状态相关的方法。例如，

```
for(Iterator<String> i=collection.iterator();i.hasNext();){
        String s=i.next()
}
```

Iterator 接口有一个“状态相关”的 next 方法，和相应的状态测试方法 hasNext().这使得利用传统的 for 循环对集合进行迭代的标准模式成为可能

(2).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，返回一个可识别的值，比如 null。

58、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

(1)Java 提供了三种可以抛出的结构：

a.受检异常(checked exception)

b.运行时异常(runtime exception)

c.错误(error)

(2)运行时异常和错误都是不需要也不应该被捕获的可抛出结构。如果程序抛出运行时异常或者错误，说明出现了不可恢复的情形，继续执行下去有害无益。如果没有捕捉到这样的结构，将会导致当前线程停止，并出现适当的错误消息。

(3)使用原则：

a.如果期望调用者能够适当地恢复，对于这种情况就使用受检的异常。

b.用运行时异常来表明编程错误。

c.如果不清楚是否有可能恢复，则使用未受检异常。

59、避免不必要的使用受检异常

(1) 与返回代码不同，他们强迫程序员处理异常的条件。也就是说使用受检的异常会使 API 使用起来非常不方便

(2) 如果正确使用 API 并不能阻止这中异常条件的产生，并且一旦产生异常，使用 API 的程序员可以立即采取有用过的方法，这种负担就被认为是正当的，所以没必要使用受检异常。

(3) 添加状态测试方法，以此把受检的异常变成未受检的异常

60、优先使用标准的异常

(1)使用现有异常的好处：

a.是客户代码更加易于学习和使用

b.可读性更好

c.异常类型越少，开销越少

(2)一些常见的异常类型：

异常类型	使用场合
IllegalArgumentException	非 null 的参数值不正确
IllegalStateExcepition	对于方法调用而言，对象状态不适合
NullPointerException	在禁止使用 null 的情况下使用 null
IndexOutOfBoundsException	下表参数值越界
ConcurrentModificationException	在禁止并发修改的情况下，检测到并发修改
UnSupportedOperationException	对象不支持用户请求的方法

61、抛出与抽象相对应的异常

想想这样一种情况：方法 B 抛出了一个受检的异常 ，那么方法 A 在内部调用方法 B 时，面对方法 B 抛出的受检异常，可以选择继续抛出向上传播这个异常，也可以捕获这个异常进行处理。究竟是向上传播抛出，还是捕获处理呢？？？

有一个指导原则是：抛出与抽象相对应的异常。说明如下：

(1)例如如果方法 B 抛出了 NoSuchElementException 这个受检异常，然而在方法 A 中调用方法 B 时，根据方法 A 中的逻辑，当遇到 NoSuchElementException 异常时，抛出一个 IndexsOutOfBoundsException 异常更为合适。那么就不应该选择向上传播抛出 NoSuchElementException，而是应该选择捕获 NoSuchElementException，然后抛出 IndexsOutOfBoundsException。

(2)**更高层的实现应该捕获底层的异常，同时抛出可以按照高层抽象进行解释的异常**。这种做法称为异常转译（exception translation）。

一种特殊的异常转译形式称为异常链（exception chaining)。

(3)尽管异常转译让异常更加明确。但是如有可能，处理来自底层的异常的最好的做法是，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。

a.有时候，可以在给低层方法传递参数之前，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。

b.如果无法避免低层异常，次选方案是，让更高层的方法来悄悄地绕开这些异常（方法 C 调用方法 A，那么方法 C 就是更高层的方法）。那么在高层方法中调用低层方法时，面对低层方法抛出的受检异常，高层异常可以捕获异常，转化为非受检异常，或者利用某种适当的记录机制（日志）将异常记录下来。这样更高层的方法 C 在调用高层方法 A 是，不用再受来自低层方法的异常烦扰，而异常在高层方法中也得到了处理。

62、每个方法抛出的异常都要有文档

描述一个方法所抛出的异常，是正确使用这个方法时所需文档的重要组成部分。

(1)始终要单独声明受检的异常，并且利用 javadoc 的@throws 标记，准确地记录下来抛出的每个异常条件。

(2)不要使用 throws 关键字将未受检的异常也包含在方法的声明中。

(3)如果一个类中的许多方法出于同样的原因而抛出同一个异常，在该类的文档注释中对这个异常建立文档，这是可以接受的。

63、在细节消息中包含能捕获失败的信息

本条建议：

(1)为了捕获失败，异常的细节信息应当包含所有“对该异常有贡献”的参数和域的值。

(2)应当区分异常的细节和“用户层次的错误信息”，对于调试程序的维护人员，更重要的是信息的内容而不是可读性

(3)有些异常类（例如 IndexOutOfBoundsException，）提供了使用细节信息的构造方法，对于 IndexOutOfBoundsException 有 public IndexOutOfBoundsException(int lowerBound,int upperBound,int index)这样的构造器，来确保异常的细节信息包含足够多的能描述捕获失败的消息，应当推荐。

64、努力是失败保持原子性

(1)失败原子性：指失败的方法调用应当是对象保持在被调用之前的状态。

(2)获得失败原子性的方法：

a.检查参数的有效性

b 调整计算过程的顺序，是的任何可能会导致失败的计算部分在对象状态被修改之前发生。

c.编写恢复代码，让对象回滚到失败之前，但这中方法并不常用，主要用于持久性的数据结构

d.对对象的拷贝进行操作，例如，Collections.sort 就是在内部将输入列表转到一个数组，还提高了性能

(3)不适合实现失败原子性的情况：此时应当将其记入 API 文档中，指明失败时对象将在什么状态，是否可用

a.并发修改对象导致状态不一致：这种情况下通常是不可恢复的

b.保持失败原子性会显著增加开销或者复杂性的时候

65、不要忽略异常

不应该有空的 catch 块. 至少,catch 块也应该包含一条说明,解释为什么可以忽略这个异常