链滴

异常知识点总结

作者：zwxbest
原文链接：https://ld246.com/article/1545542758786
来源网站：链滴

运行时和检查时异常

以下

检查时异常（Checked Exception）简称为CE

运行时异常（Runtime Exception）`简称为RE

关系

CE与RE并不是矛盾关系，而是交叉关系，因为他们有共同的父类Exception，所以按照矛盾关系去思对比这两者间的关系是错误的想法。

语法

从语法上说，编译器会在编译时检查CE，如果没有try ...catch或者throws编译就不会通过。编译器不检查RE，所以没有try ...catch或者throws也没问题。

语义

从语义上说，CE是在本层无法处理，这时需要通知给上层方法或者客户端。比如IOException,在本层现IO失败，是源错误，自己try ...catch并通知上层或者直接throws。

而RE，可能是在本层无法处理,比如传入参数为Null但不应该为Null，此时应该通知上层。这是和CE同的地方。

大部分情况是你的程序逻辑本身有问题，比如数组越界，此时做一下边界限制，在本层处理即可。

设计

从设计角度来讲，C++所有的异常都是CE，Java如果也是那样就烦死了，几乎每个方法都要throws。

最佳实践

系统提示友好

对于可能会给用户看的异常信息，封装一层，写明错误，自定义错误码。

异常分类

不建议的写法

```
try {
   //xxx
}catch (Exception e){
   log.error(e);
}
```

建议的写法

```
try {
   //xxx
}catch (FileNotFoundException e){
   log.error("file not found");
}catch (Exception e){
   log.error(e);
}
```

这就比较直观，节省时间，不需要一层层看代码在分析了。

一次抛出多个异常

在注册页面，可能会出现一次抛出多个异常的情况，比如邮箱存在，密码少于6位等。

这种情况需要将可能出现的异常放到集合中，最后统一抛出。

```
public static void doStuff() throws MyException {
      List list = new ArrayList();
      // 第一个逻辑片段
  try {
         // Do Something
  } catch (Exception e) {

         list.add(e);
      }
      // 第二个逻辑片段
  try {
         // Do Something
  } catch (Exception e) {
         list.add(e);
      }

      if (list.size() > 0) {
         throw new MyException(list);
      }

   }
}

class MyException extends Exception {
   // 容纳所有的异常
  private List causes = new ArrayList();

   // 构造函数，传递一个异常列表
  public MyException(Listextends Throwable> _causes) {
      causes.addAll(_causes);
   }

   // 读取所有的异常
  public List getExceptions() {
      return causes;
   }
}
```

异常链传递

由service包装后上抛，最外层的Contrller使用@ControllerAdvice根据异常的类型来决定返回自定错误信息还是sever error

检查时异常尽量转换为运行时异常

我们实现接口方法，接口方法并没有用throws修饰，此时如果实现类抛出了CE,此时要不修改接口，不try..catch转换为RE，修改接口肯定是最差的方案，因为所有继承的都需要修改，同时也破坏了迪特法则，因为接口的其他实现并不认识这个异常类。如果我们每个CE都try,catch，又得加好多代码，读性变差。最好的方案是包装CE为RE。

那什么情况下需要包装CE为RE呢?

受检异常转换为非受检异常是需要根据项目的场景来决定的，例如同样是刷卡， 员工拿 着自己的工卡到考勤机上打考勤，此时如果附近有磁性物质干扰 ，则考勤机可以把这种受 检异常转化为非受检异常，黄灯闪烁后不做任何记录登记，因为考勤 失败 这种 情景 不是 “致命"的 业务逻辑，出错了，重新刷一下即可。但是到银行网点取钱就不一样了，拿着银行卡到银行取钱，同样有磁性物质干扰，刷不出来，那这种异常就必须登记处理，否则会成为威胁银行卡安全的事件。汇总成一句话 ： 当受检异常威胁到了系统的安全性、稳定性、可靠性、正确性时，则必须处理，不能转化为非受检异常，其他情况则可以转换为非受检异常。

不要在finally中处理返回值

```
public static void main(String[] args) {
    try {
        doStuff(-1);
        doStuff(100);
    } catch (Exception e) {
        System.out.println("这里是永远都不会到达的");
    }
}

// 该方法抛出受检异常
public static int doStuff(int _p) throws Exception {
    try {
        if (_p < 0) {
            throw new DataFormatException(" 数提格式错误");
        } else {
            return _p;
        }
    } catch (Exception e) {
        //异常处理
  throw e;
    } finally {
        return -1;
    }
}
```

doStuff(-l)的值是-1，doStuff(100)的值也是-1，调用doStuff方法永远都不会抛出异常。

为什么明明把异常throw出去了，但main方法却捕捉不到呢？这是因为异常线程在监视到有异常发生时，就会登记当前的异常类型为DataFormatException，但是当执行器执行finally代码块时，则会重新为doStuff方法赋值，也就是告诉调用者“该方法执行正确，没有产生异常，返回值是1”，于是乎，异常神奇的消失了。

会覆盖try代码块中的return

```
public static int doStuff() {
    int a = 1;
    try {
        return a;
    } catch (Exception e) {
    } finally {
   //重新修改一下返回值
  a = 1;
    }
    return 0;
}
```

我们知道方法是在栈内存中运行的，并且会按照“先进后出”的原则执行，main方法调用了doStuff方法，则main方法在下层，doStuff在上层，当doStuff方法执行完“returna”时，此方法的返回值已经确定是in类型1(a变量的值，注意基本类型都是值拷贝，而不是引用)，此后finally代码块再修改a的值已经与doStuff返回者没有任何关系了，因此该方法永远都会返回1。

```
public static Person doStuff() {
    Person person = new Person();
    person.setName("张 三 ");
    try {
        return person;
    } catch (Exception e) {
    } finally {
        //重新修改一下返回值
  person.setName("李 四 ");
    }
    person.setName(" 王 五 ");
    return person;
}

@Getter
@Setter
static  class Person {
    private String name;
}
```

此方法的返回值永远都是name为李四的Person对象，原因是Person是一个引用对象，在try代码块中的返回值是Person对象的地址

会屏蔽异常

与return语句相似，System.exit(O)或Runtime.getRuntime().exit(O)出现在异常代码块中也会产生非常多的错误假象，增加代码的复杂性

多使用异常，把性能问题放一边

Java的异常处理机制确实比较慢，这个“比较慢”是相对于诸如String、Integer等对象来说的，单单从对象的创建上来说，new一个IOException会比String慢5倍，这从异常的处理机制上也可以解释：因为它要执行fillInStackTrace方法，要记录当前栈的快照，而String类则是直接申请一个内存创建对象，异常类慢一筹也就在所难免了。

但是异常也不会经常出现的，所以相比代码可读性而言，性能考虑可以微乎其微了。