第一次测验考点

目录

[快速排序 1](#_Toc487220337)

[冒泡排序 7](#_Toc487220338)

[异常机制 8](#_Toc487220339)

[单例和多例 10](#_Toc487220340)

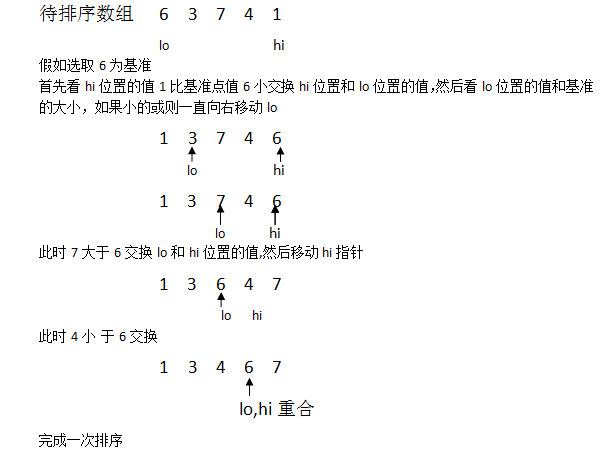
[自定义异常 17](#_Toc487220341)

[【Java学习笔记】自定义Java异常 24](#_Toc487220342)

## 快速排序

算法思想：基于分治的思想，是冒泡排序的改进型。首先在数组中选择一个基准点（该基准点的选取可能影响快速排序的效率，后面讲解选取的方法），然后分别从数组的两端扫描数组，设两个指示标志（lo指向起始位置，hi指向末尾)，首先从后半部分开始，如果发现有元素比该基准点的值小，就交换lo和hi位置的值，然后从前半部分开始扫秒，发现有元素大于基准点的值，就交换lo和hi位置的值，如此往复循环，直到lo>=hi,然后把基准点的值放到hi这个位置。一次排序就完成了。以后采用递归的方式分别对前半部分和后半部分排序，当前半部分和后半部分均有序时该数组就自然有序了。

排序过程：



算法实现：

public static int partition(int []array,int lo,int hi){

//固定的切分方式

int key=array[lo];

while(lo<hi){

while(array[hi]>=key&&hi>lo){//从后半部分向前扫描

hi--;

}

array[lo]=array[hi];

while(array[lo]<=key&&hi>lo){从前半部分向后扫描

lo++;

}

array[hi]=array[lo];

}

array[hi]=key;

return hi;

}

public static void sort(int[] array,int lo ,int hi){

if(lo>=hi){

return ;

}

int index=partition(array,lo,hi);

sort(array,lo,index-1);

sort(array,index+1,hi);

}

快速排序的时间复杂度为O(NlogN).

快速排序的优化

对于基准位置的选取一般有三种方法：固定切分，随机切分和三取样切分。固定切分的效率并不是太好，随机切分是常用的一种切分，效率比较高，最坏情况下时间复杂度有可能为O(N2).对于三数取中选择基准点是最理想的一种。

三数取中切分：

public static int partition(int []array,int lo,int hi){

//三数取中

int mid=lo+(hi-lo)/2;

if(array[mid]>array[hi]){

swap(array[mid],array[hi]);

}

if(array[lo]>array[hi]){

swap(array[lo],array[hi]);

}

if(array[mid]>array[lo]){

swap(array[mid],array[lo]);

}

int key=array[lo];

while(lo<hi){

while(array[hi]>=key&&hi>lo){

hi--;

}

array[lo]=array[hi];

while(array[lo]<=key&&hi>lo){

lo++;

}

array[hi]=array[lo];

}

array[hi]=key;

return hi;

}

public static void swap(int a,int b){

int temp=a;

a=b;

b=temp;

}

public static void sort(int[] array,int lo ,int hi){

if(lo>=hi){

return ;

}

int index=partition(array,lo,hi);

sort(array,lo,index-1);

sort(array,index+1,hi);

}

快速排序在序列中元素很少时，效率将比较低，不然插入排序，因此一般在序列中元素很少时使用插入排序，这样可以提高整体效率。

public static void quick(int []array ,int lo,int hi){

if(hi-lo+1<10){

insertSort(array);

}else{

quickSort(array,lo,hi);

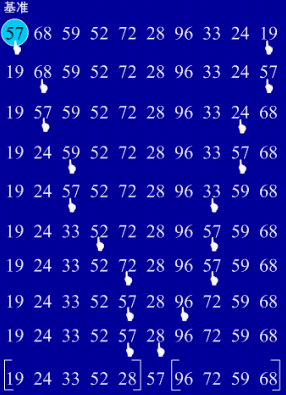
}

}

**6.快速排序**

（1）基本思想：选择一个基准元素,通常选择第一个元素或者最后一个元素,通过一趟扫描，将待排序列分成两部分,一部分比基准元素小,一部分大于等于基准元素,此时基准元素在其排好序后的正确位置,然后再用同样的方法递归地排序划分的两部分。

（2）实例：



（3）用java实现

public class quickSort {

inta[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

public quickSort(){

quick(a);

for(int i=0;i<a.length;i++)

System.out.println(a[i]);

}

public int getMiddle(int[] list, int low, int high) {

int tmp = list[low]; //数组的第一个作为中轴

while (low < high) {

while (low < high && list[high] >= tmp) {

high--;

}

list[low] = list[high]; //比中轴小的记录移到低端

while (low < high && list[low] <= tmp) {

low++;

}

list[high] = list[low]; //比中轴大的记录移到高端

}

list[low] = tmp; //中轴记录到尾

return low; //返回中轴的位置

}

public void \_quickSort(int[] list, int low, int high) {

if (low < high) {

int middle = getMiddle(list, low, high); //将list数组进行一分为二

\_quickSort(list, low, middle - 1); //对低字表进行递归排序

\_quickSort(list, middle + 1, high); //对高字表进行递归排序

}

}

public void quick(int[] a2) {

if (a2.length > 0) { //查看数组是否为空

\_quickSort(a2, 0, a2.length - 1);

}

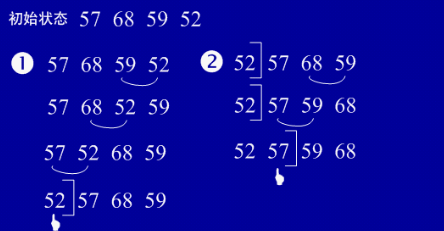
}

}

## 冒泡排序

（1）基本思想：在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

（2）实例：



（3）用java实现

public class bubbleSort {

public bubbleSort(){

int a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

int temp=0;

for(int i=0;i<a.length-1;i++){

for(int j=0;j<a.length-1-i;j++){

if(a[j]>a[j+1]){

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

}

}

}

for(int i=0;i<a.length;i++)

System.out.println(a[i]);

}

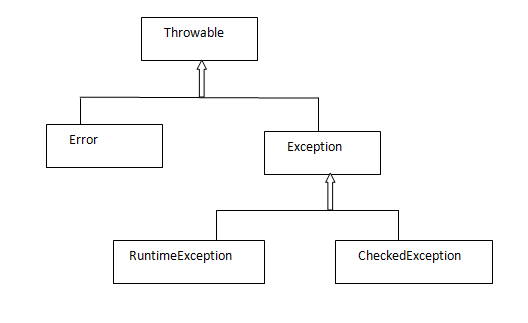
}

## 异常机制

概述:异常是java程序运行时出现的一些意外情况。

异常的分类：所有的异常都继承自java.lang.Trowable类，Trowable类有两个直接子类，Exception和error。

异常的层次结构图：



error是java程序本身无法恢复的严重错误，一般不进行捕获，处理，一般是通知用户并终止程序执行。

检查时异常一般是由通过throws抛出的，该方法的调用者必须在调用该方法时进行捕获、处理，否则将不能进行编译。

异常的处理：使用try{

                    }catch(Exception e){

　　　　　　　}finally{

　　　　　　　}

示例：

public class Test {

public static void main(String[] args) {

try{

String [] s={"l","u","n"};

for(int i=0;i<=s.length;i++){

System.out.println(s[i]);

}

System.out.println("gg");

}catch(ArrayIndexOutOfBoundsException e){

e.printStackTrace();

}catch(Exception e){

System.out.println("程序出错！");

System.out.println(e.getMessage());

e.printStackTrace();

}finally{

System.out.println("very good");

}

}

}

## 单例和多例

单例模式的关键有两点：

1.构造方法为私有，这样外界就不能随意调用。

2.get的方法为静态，由类直接调用

多例模式（Multiton）

1 、多例类可以有多个实例   
2 、多例类必须能够自我创建并管理自己的实例，并向外界提供自己的实例。

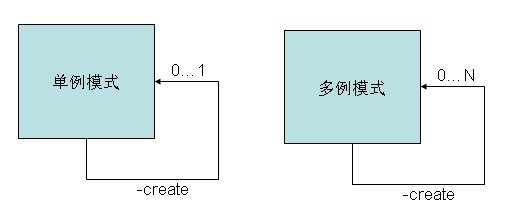
一、单例模式和多例模式说明：

1.         单例模式和多例模式属于对象模式。

2.         单例模式的对象在整个系统中只有一份，多例模式可以有多个实例。

3.         它们都不对外提供构造方法，即构造方法都为私有。

4.         单例模式和多例模式的结构图如下所示：



单例模式的三种形式：

第一种形式:懒汉式

public class SingletonClass{

    private static SingletonClass instance=null;

    public static　synchronized　SingletonClass getInstance()

    {

        if(instance==null)

        {

           instance=new SingletonClass();

        }

        return instance;

    }

    private SingletonClass(){

    }

}

第二种形式:饿汉式

//对第一行static的一些解释

// java允许我们在一个类里面定义静态类。比如内部类（nested class）。

//把nested class封闭起来的类叫外部类。

//在java中，我们不能用static修饰顶级类（top level class）。

//只有内部类可以为static。

public class Singleton{

    //在自己内部定义自己的一个实例，只供内部调用

   private static final Singleton instance = new Singleton();

   private Singleton(){

        //do something

    }

    //这里提供了一个供外部访问本class的静态方法，可以直接访问

   public static Singleton getInstance(){

        return instance;

    }

}

第三种形式: 双重锁的形式

public class Singleton{

    private static Singleton instance=null;

    private Singleton(){

        //do something

    }

    public static Singleton getInstance(){

        if(instance==null){

            synchronized(Singleton.class){

                if(instance==null)

{

  instance=new Singleton();

                }

            }

        }

        return instance;

    }

}

二、应用举例

1.         单例模式举例:

package com.solid.pattern;

import **[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "_blank)**.util.Locale;

import java.util.ResourceBundle;

/\*\*

 \* 单例模式

 \* @author solid

 \*

 \*/

public class Singleton {

    private static Singleton singleton = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {

       return singleton;

    }

    /\*\*

     \* 获取配置文件的值

     \* @param key

     \*/

    public void getMessage(String key) {

       Locale locale = new Locale(key);

       ResourceBundle res = ResourceBundle.getBundle("res\_zh\_CN");

       String message = res.getString(key);

       System.out.println(message);

    }

}

**package** com.solid.pattern;

/\*\*

 \* 单例模式**[测试](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "_blank)**

 \* **@author** solid

 \*

 \*/

**public** **class** TestSingleton {

**private** **static** Singleton singleton;

**public** **static** **void** main(String[] args) {

       singleton = Singleton.getInstance();

       singleton.getMessage("title");

    }

}

2.         多例模式举例:

**package** com.solid.pattern;

/\*\*

 \* 多例模式

 \* **@author** solid

 \*

 \*/

**public** **class** Multiton {

**private** **static** Multiton multi1 = **new** Multiton();

**private** **static** Multiton multi2 = **new** Multiton();

**private** Multiton() {}

**public** **static** Multiton getInstance(**int** key) {

**if**(key == 1) {

**return** multi1;

       } **else** {

**return** multi2;

       }

    }

    /\*\*

     \* 获取1—6之间的随机数

     \*/

**public** **void** getValue() {

**int** value = (**int**)(Math.random()\*6+1);

       System.out.println(value);

    }

}

**package** com.solid.pattern;

/\*\*

 \* 多例模式测试

 \* **@author** solid

 \*

 \*/

**public** **class** TestMultiton {

**private** **static** Multiton multi1;

**private** **static** Multiton multi2;

**public** **static** **void** main(String[] args) {

       multi1 = Multiton.getInstance(1);

       multi2 = Multiton.getInstance(2);

       multi1.getValue();

       multi2.getValue();

    }

}

 单例和多例的详细描述：

1. 什么是单例多例:  
所谓单例就是所有的请求都用一个对象来处理，比如我们常用的service和dao层的对象通常都是单例的，而多例则指每个请求用一个新的对象来处理，比如action;   
2. 如何产生单例多例:  
    在通用的SSH中，单例在**[spring](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)**中是默认的，如果要产生多例，则在配置文件的bean中添加scope="prototype";   
3. 为什么用单例多例：  
    之所以用单例，是因为没必要每个请求都新建一个对象，这样子既浪费CPU又浪费内存；  
   之所以用多例，是为了防止并发问题；即一个请求改变了对象的状态，此时对象又处理另一个请求，而之前请求对对象状态的改变导致了对象对另一个请求做了错误的处理；  
    用单例和多例的标准只有一个：  
    当对象含有可改变的状态时（更精确的说就是在实际应用中该状态会改变），则多例，否则单例；  
4. 何时用单例？何时用多例？  
    对于struts2来说，action必须用多例，因为action本身含有请求参数的值，即可改变的状态；  
  而对于STRUTS1来说，action则可用单例，因为请求参数的值是放在actionForm中，而非action中的；  
   另外要说一下，并不是说service或dao一定是单例，标准同第3点所讲的，就曾见过有的service中也包含了可改变的状态，同时执行方法也依赖该状态，但一样用的单例，这样就会出现隐藏的BUG,而并发的BUG通常很难重现和查找；

## 自定义异常

1. 1.自定义异常:
3. class 异常类名 extends Exception
4. {
5. public 异常类名(String msg)
6. {
7. super(msg);
8. }
9. }
11. 2.标识可能抛出的异常:
13. throws 异常类名1,异常类名2
15. 3.捕获异常:
16. try{}
17. catch(异常类名 y){}
18. catch(异常类名 y){}
20. 4.方法解释
21. getMessage() //输出异常的信息
22. printStackTrace() //输出导致异常更为详细的信息

25. \*/
26. /\*自定义异常\*/
27. **class** ChushulingException **extends** Exception
28. {
29. **public** ChushulingException(String msg)
30. {
31. **super**(msg);
32. }
33. }
35. **class** ChushufuException **extends** Exception
36. {
37. **public** ChushufuException(String msg)
38. {
39. **super**(msg);
40. }
41. }
43. /\*自定义异常 End\*/
45. **class** Numbertest
46. {
47. **public** **int** shang(**int** x,**int** y) **throws** ChushulingException,ChushufuException
48. {
49. **if**(y<0)
50. {
51. **throw** **new** ChushufuException("您输入的是"+y+",规定除数不能为负数!");//抛出异常
52. }
53. **if**(y==0)
54. {
55. **throw** **new** ChushulingException("您输入的是"+y+",除数不能为0!");
56. }
58. **int** m=x/y;
59. **return** m;
60. }
61. }




67. **class** Rt001
68. {
69. **public** **static** **void** main(String[]args)
70. {
71. Numbertest n=**new** Numbertest();
73. //捕获异常
74. **try**
75. {
76. System.out.println("商="+n.shang(1,-3));
77. }
78. **catch**(ChushulingException yc)
79. {
80. System.out.println(yc.getMessage());
81. yc.printStackTrace();
82. }
83. **catch**(ChushufuException yx)
84. {
85. System.out.println(yx.getMessage());
86. yx.printStackTrace();
87. }
88. **catch**(Exception y)
89. {
90. System.out.println(y.getMessage());
91. y.printStackTrace();
92. }
94. **finally**{ System.out.println("finally!");} ////finally不管发没发生异常都会被执行
96. }
97. }
98. /\*

为什么要使用自定义异常，有什么好处？

1.我们在工作的时候，项目是分模块或者分功能开发的 ,基本不会你一个人开发一整个项目，使用自定义异常类就统一了对外异常展示的方式。

2.有时候我们遇到某些校验或者问题时，需要直接结束掉当前的请求，这时便可以通过抛出自定义异常来结束，如果你项目中使用了SpringMVC比较新的版本的话有控制器增强，可以通过@ControllerAdvice注解写一个控制器增强类来拦截自定义的异常并响应给前端相应的信息（关于springMVC控制器增强的知识有空再和大家分享）。

3.自定义异常可以在我们项目中某些特殊的业务逻辑时抛出异常，比如"中性".equals(sex)，性别等于中性时我们要抛出异常，而Java是不会有这种异常的。系统中有些错误是符合Java语法的，但不符合我们项目的业务逻辑。

4.使用自定义异常继承相关的异常来抛出处理后的异常信息可以隐藏底层的异常，这样更安全，异常信息也更加的直观。自定义异常可以抛出我们自己想要抛出的信息，可以通过抛出的信息区分异常发生的位置，根据异常名我们就可以知道哪里有异常，根据异常提示信息进行程序修改。比如空指针异常NullPointException，我们可以抛出信息为“xxx为空”定位异常位置，而不用输出堆栈信息。

我们再来看看自定义异常的毛病：

毋庸置疑，我们不可能期待JVM（Java虚拟机）自动抛出一个自定义异常，也不能够期待JVM会自动处理一个自定义异常。发现异常、抛出异常以及处理异常的工作必须靠编程人员在代码中利用异常处理机制自己完成。这样就相应的增加了一些开发成本和工作量，所以项目没必要的话，也不一定非得要用上自定义异常，要能够自己去权衡

 编写自己的异常类时需要记住下面的几点。

* 所有异常都必须是 Throwable 的子类。
* 如果希望写一个检查性异常类，则需要继承 Exception 类。
* 如果你想写一个运行时异常类，那么需要继承 RuntimeException 类。

可以像下面这样定义自己的异常类：

class MyException extends Exception{ }

我们来看一个完整的实例：

[复制代码](javascript:void(0);)

1 package com.czgo.exception;

2

3 /\*\*

4 \* 自定义异常类(继承运行时异常)

5 \* @author AlanLee

6 \* @version 2016/11/26

7 \*/

8 public class MyException extends RuntimeException {

9

10 private static final long serialVersionUID = 1L;

11

12 /\*\*

13 \* 错误编码

14 \*/

15 private String errorCode;

16

17 /\*\*

18 \* 消息是否为属性文件中的Key

19 \*/

20 private boolean propertiesKey = true;

21

22 /\*\*

23 \* 构造一个基本异常.

24 \*

25 \* @param message

26 \* 信息描述

27 \*/

28 public MyException(String message)

29 {

30 super(message);

31 }

32

33 /\*\*

34 \* 构造一个基本异常.

35 \*

36 \* @param errorCode

37 \* 错误编码

38 \* @param message

39 \* 信息描述

40 \*/

41 public MyException(String errorCode, String message)

42 {

43 this(errorCode, message, true);

44 }

45

46 /\*\*

47 \* 构造一个基本异常.

48 \*

49 \* @param errorCode

50 \* 错误编码

51 \* @param message

52 \* 信息描述

53 \*/

54 public MyException(String errorCode, String message, Throwable cause)

55 {

56 this(errorCode, message, cause, true);

57 }

58

59 /\*\*

60 \* 构造一个基本异常.

61 \*

62 \* @param errorCode

63 \* 错误编码

64 \* @param message

65 \* 信息描述

66 \* @param propertiesKey

67 \* 消息是否为属性文件中的Key

68 \*/

69 public MyException(String errorCode, String message, boolean propertiesKey)

70 {

71 super(message);

72 this.setErrorCode(errorCode);

73 this.setPropertiesKey(propertiesKey);

74 }

75

76 /\*\*

77 \* 构造一个基本异常.

78 \*

79 \* @param errorCode

80 \* 错误编码

81 \* @param message

82 \* 信息描述

83 \*/

84 public MyException(String errorCode, String message, Throwable cause, boolean propertiesKey)

85 {

86 super(message, cause);

87 this.setErrorCode(errorCode);

88 this.setPropertiesKey(propertiesKey);

89 }

90

91 /\*\*

92 \* 构造一个基本异常.

93 \*

94 \* @param message

95 \* 信息描述

96 \* @param cause

97 \* 根异常类（可以存入任何异常）

98 \*/

99 public MyException(String message, Throwable cause)

100 {

101 super(message, cause);

102 }

103

104 public String getErrorCode()

105 {

106 return errorCode;

107 }

108

109 public void setErrorCode(String errorCode)

110 {

111 this.errorCode = errorCode;

112 }

113

114 public boolean isPropertiesKey()

115 {

116 return propertiesKey;

117 }

118

119 public void setPropertiesKey(boolean propertiesKey)

120 {

121 this.propertiesKey = propertiesKey;

122 }

123

124 }

使用自定义异常抛出异常信息：

[复制代码](javascript:void(0);)

1 package com.czgo.exception;

2

3 public class MyExceptionTest {

4

5 public static void main(String[] args) {

6

7 String[] sexs = {"男性","女性","中性"};

8 for(int i = 0; i < sexs.length; i++){

9 if("中性".equals(sexs[i])){

10 throw new MyException("你全家都是中性！");

11 }else{

12 System.out.println(sexs[i]);

13 }

14 }

15 }

16 }

# [【Java学习笔记】自定义Java异常](http://www.cnblogs.com/gnuhpc/archive/2013/01/04/2843675.html)

作者：[gnuhpc](http://www.cnblogs.com/gnuhpc/)   
出处：<http://www.cnblogs.com/gnuhpc/>

1.前言：

你的程序总有一天会崩溃掉，在崩溃掉的时候我们要知道它在哪，为了什么而崩溃掉，数据的保存或者丢失情况如何等问题。我们可以通过继承类java.lang.Throwable的子类：Exception来设计我们自己的Java异常。Exception类用于描述程序能够捕获的异常，如ClassNotFoundException。要注意的是自定义异常类之间也可以有继承关系，同时也需要为自定义异常类设计构造方法，以方便构造自定义异常对象。

2.设计实例分析：   
这是个比较完整的自定义异常类的设计，其实是比较模板化的东西。

package playground;   
import java.io.\*;

public class MyException extends Exception {   
  private int id; // a unique id   
  private String classname; // the name of the class   
  private String method; // the name of the method   
  private String message; // a detailed message   
  private MyException previous =   
   null; // the exception which was caught   
  private String separator = "/n"; // line separator   
  public MyException(int id, String classname, String method,   
    String message, MyException previous) {   
    this.id        = id;   
    this.classname = classname;   
    this.method    = method;   
    this.message   = message;   
    this.previous  = previous;   
  }    
  public String traceBack() {   
    return traceBack("/n");   
  }

  public String traceBack(String sep) {   
    this.separator = sep;   
    int level = 0;   
    MyException e = this;   
    String text = line("Calling sequence (top to bottom)");   
    while (e != null) {   
      level++;   
      text += line("--level " + level + "--------------------------------------");   
      text += line("Class/Method: " + e.classname + "/" + e.method);   
      text += line("Id          : " + e.id);   
      text += line("Message     : " + e.message);   
      e = e.previous;   
    }    
    return text;   
  }

  private String line(String s) {   
    return s + separator;   
  }

}

我们来一个个看看这些东西：   
在这个继承了Exception类的自定义异常类中，我们定义了如下变量：   
id：独立标示符，这个是用来进行标示类中什么地方出现了错误被捕捉到。   
classname：捕捉到这个错误的类的名字。   
method：捕捉到这个错误的方法的名字。   
message：用来描述整个事件的情况。   
previous：是MyException得一个实例，若在链表中它是第一个的话，那么它就是null。

我们再看看都定义了什么方法：   
traceBack()：产生一个包含在异常类中存储的数据的一个回溯。使用newline标示作为分隔符   
traceBack(String sep)：和上一个其实是相同的东西，使用的分隔符可以自己定义。   
line(String s)：traceBack使用的一个方法。

步骤一：先扔出第一个异常：   
这有两种情况：   
a.若程序判断异常的某个地方主动抛出异常，那么你就会主动抛出一个错误，例如：   
throw new MyException(  1, "Person", "getSalary",   
  "Trying to get a salary, but no person was specified",null);

b.在另一个异常出现时抛出该异常，例如：   
catch (Exception e) {   
  throw new MyException(4, "Person", "read", e.toString(), null);   
}

要是某一个方法要抛出MyException异常，那么在方法定义的时候形式如下：   
public void read() throws MyException . . .

步骤二：接着我们扔出来的异常，例如：   
Person p = new Person();   
p.setPersonId(id);   
try {   
  p.read();   
  s += p.getSalary();   
}    
catch (MyException e) {   
  throw new MyException(1, "Stats", "getAllSalaries",   
    "Could not get salary for " + id, e);   
}

我们的策略是这样的：将我们可能扔出异常的方法置于我们的监控之下（try{}），然后出了事情我们在处理区（catch{}）把这个异常处理，至于处理的方式可以是当场处理，也可以处理不了交予其他地方处理（throw()）。需要注意的是，我们看到最后一个参数e，我们要在每个新的异常处将这个异常加入到异常类的链表中。

步骤三：回溯异常   
我们上边设计的这个异常最后得出的一个异常链如下：   
Calling sequence (top to bottom)   
--level 1--------------------------------------   
Class/Method: SalaryServlet/doGet   
Id : 7   
Message : Trying to get the total salary for employees   
--level 2--------------------------------------   
Class/Method: Stats/getAllSalaries   
Id : 1   
Message : Could not get salary for Lincoln   
--level 3--------------------------------------   
Class/Method: Person/read   
Id : 3   
Message : java.sql.SQLException: [HANSEN]Invalid object name 'xEMPLOYEE'.   
--level 4--------------------------------------   
Class/Method: Person/read   
Id : 999   
Message : SQLException. State/error code: S0002/208

我们可以分析得知在这个场景上，一个servlet被触发了，它调用了getAllSalaries，然后又调用了read方法，在read方法中，出现了一个SQLException

3.总结：   
自定义异常类给了我们自己喜欢的异常的出现,其他的也没什么。哈哈~

作者：[gnuhpc](http://www.cnblogs.com/gnuhpc/)   
出处：<http://www.cnblogs.com/gnuhpc/>

结束语：现在搞软件开发的人越来越多了，高手数不胜数，菜鸟也数不胜数，把这行的行情搅的比较混乱。所以初入这行的哥们一定要有危机意识。公司都想要工资开的不高，又能做一大把事情的人，搞开发很累，但是在当前的环境下，一定要不断的去提高自己。到了洗牌的时候，可能被淘汰的人里面就有你，除非你去做别的，不然就硬着头皮干几年把技术搞扎实，该复习的知识就多多复习，该学习的知识就多多学习，基础扎实了就去看看底层的东西，比如理解理解Java虚拟机，研究研究Spring源码。多关注关注前沿技术的发展，因为你现在所用的东西或许某一天大部分公司就不用了，或者根本满足不了项目的需求，这都是有可能的。希望大家都能混出自己的一片天地，小Alan现在也还在吃土中，所以大家都在一起前行，这世界终究是属于年轻人的，别气馁。