知识点一：异常的好处

1、防止宕机

如果函数内部不捕获异常，则会造成异常向上传递，由调用者来负责，当传递到 main 的时候，会抛出：

Exception in thread main：xxx

这样整个程序就停止了。如果在函数内部对异常进行捕获的话，捕获之后，程序仍然进行函数的下一步执行，而不会造成整个程序的停止。

2、让代码有灵性

```
public synchronized E peek() {
    int     len = size();
    if (len == 0)
        throw new EmptyStackException();
    return elementAt(len - 1);
}
```

问：为什么 len==0 的时候不返回 null 呢，而是要抛出异常？

答：

（1）可能结果本身就是 null，返回 null 属于正确结果，返回 null 容易产生歧义，到底是真正的结果是 null 还是出现了问题导致为 null 呢？调用者对返回的结果不太明确。

（2）抛出异常说明是出现了错误，必须要处理的。调用者能明确的感知出现了问题。

（3）异常分为受检异常和非受检异常（运行时异常），此处背后的思想是：利用运行异常实现受检异常的目的。

知识点二：异常基础类 Throwable

那个类是所有异常的基础类？

A String

B Error

C Throwable

D RuntimeException

参考答案：

C

试题分析：

在 Java 的 lang 包里面有一个 Throwable 类，它是所有异常的父类或者间接父类，它有两个直接子类：Error 和 Exception。Error 及其子类是处理系统内部及程序运行环境的异常，一般与硬件有关，由系统直接处理，不需要程序员在程序中处理。Exception 又分两大类，运行时异常（RuntimeException）和非运行时异常。其中类 RuntimeException 代表运行时由 Java 虚拟机产生的异常，例如算术运算异常 ArithmeticException，数组越界异常 ArrayIndexOutOfBoundsException 等；非运行时异常，例如输入输入异常 IOException 等，Java 编译器要求 Java 程序必须捕获或声明所有非运行时异常，但对运行时异常可以不做处理，因此，在编程时非运行时异常如果不处理，编译时会出错。

1 构造函数

```
public Throwable() {
        fillInStackTrace();
}
public Throwable(String message) {
```

```
        fillInStackTrace();
        detailMessage = message;
}
public Throwable(String message, Throwable cause) {
        fillInStackTrace();
        detailMessage = message;
        this.cause = cause;
}
```

Throwable 的构造函数非常神奇，不仅仅包含了我们传入的信息：

message

cause

而且还包含了自己在哪里生成的信息，例如在哪行代码生成，在哪个文件生成的，在哪个方法生成的。

2 输出异常信息

由上面可以知道，异常包含了很多信息，如何将这些信息输出呢？见下面的例子：

```
import java.io.PrintWriter;
import java.io.StringWriter;

public class Test {
    public static void main(String[] args) throws Throwable {

        Throwable th = new Throwable("hello world");
        //获取异常的message
        System.out.println(th.getMessage());

        //获取异常的堆栈信息
        StringWriter stringWriter = new StringWriter();
        th.printStackTrace(new PrintWriter(stringWriter));
        String msg = stringWriter.toString();
        System.out.println(msg);
        //重新填充堆栈的信息，输出到控制台
        th.fillInStackTrace();
        th.printStackTrace(System.out);
    }
}
```

3 利用异常信息

一个函数，我不想被某一特定的类调用，那就可以生成异常，打印出异常信息，看看是否有这个类的调用记录，如果有的话，可以中断。

知识点三：异常声明与抛出

TimedOutException不是一个RuntimeException，下面的那些选项载入程序中，使程序可以正常运行？

A public void final()

B public void final() throws Exception

C public void final() throws TimedOutException

D public void final() throw TimedOutException
E public throw TimedOutException void final()

参考答案：

BC

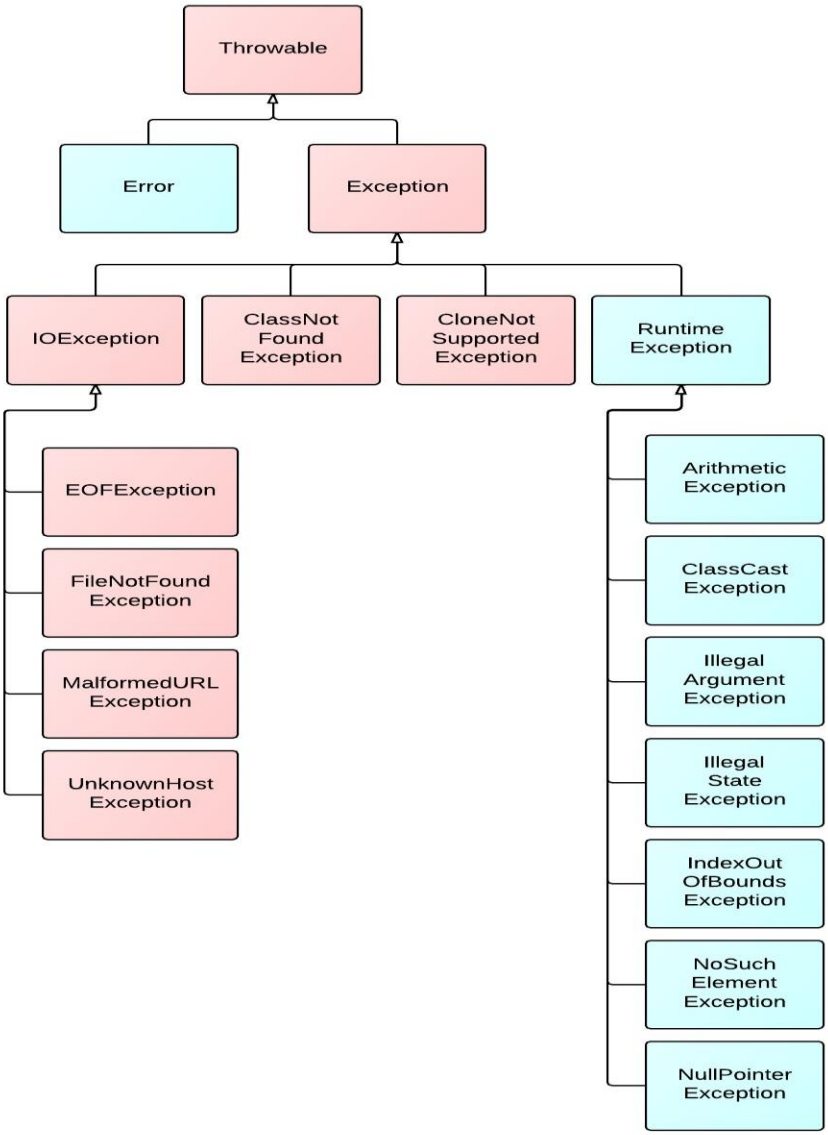
试题分析：

如果一个程序在运行时候有异常发生，而这个异常又不是RuntimeException或者Error，那么程序必须对这个异常进行捕获处理或者声明抛出该异常。捕获异常使用try-catch-finally，而声明异常则是在声明方法的同时将会发生的异常进行声明，**使用关键字throws，带s表明可以抛出多个异常。**

A项没有使用关键字声明异常，所以是错误的。由于Exception是所有异常的父类，当然也可以代表TimedOutException，所以B项是正确的。C项符合声明异常的格式，是正确的。在D项中，throw是抛出异常，而不是声明异常，关键字使用错误，所以D项是错的。E项的语法格式是错误的。

知识点四：异常的分类

异常的继承结构：Throwable 为基类，Error 和 Exception 继承 Throwable，RuntimeException 和 IOException 等继承 Exception。Error 和 RuntimeException 及其子类成为未检查异常（unchecked），其它异常成为已检查异常（checked）。



1 Error 异常

Error 表示程序在运行期间出现了十分严重、不可恢复的错误，在这种情况下应用程序只能中止运行，例如 JAVA 虚拟机出现错误。Error 是一种 Unchecked Exception，编译器不会检查 Error 是否被处理，在程序中不用捕获 Error 类型的异常。一般情况下，在程序中也不应该抛出 Error 类型的异常。

2 RuntimeException 异常

Exception 异常包括 RuntimeException 异常和其他非 RuntimeException 的异常。RuntimeException 是一种 Unchecked Exception，即表示编译器不会检查程序是否对 RuntimeException 作了处理，在程序中不必捕获 RuntimException 类型的异常，也不必在方法体声明抛出 RuntimeException 类。RuntimeException 发生的时候，表示程序中出现了编程错误，所以应该找出错误修改程序，而不是去捕获 RuntimeException。

3 Checked Exception 异常

Checked Exception 异常，这也是在编程中使用最多的 Exception，所有继承自 Exception 并且不是 RuntimeException 的异常都是 checked Exception，上图中的 IOException 和 ClassNotFoundException。JAVA 语言规定必须对 checked Exception 作处理，编译器会对此作检查，要么在方法体中声明抛出 checked Exception，要么使用 catch 语句捕获

checked Exception 进行处理，不然不能通过编译。

4 RuntimeException 和 Checked Exception 区别[掌握]

（1）生成 RuntimeException 异常之后，既不用捕获，也不用声明抛出，即使已经声明抛出，调用者也不用管。如下所示：

```
public class Test
{
    public static void main(String[] args)
    {
        throwRuntime1();
        throwRuntime2();
    }

    public static void throwRuntime1()
    {
        throw new RuntimeException("throwRuntime1:抛出runtime异常");
    }

    public static void throwRuntime2() throws RuntimeException
    {
        throw new RuntimeException("throwRuntime2:抛出runtime异常");
    }
}
```

（2）生成 Checked Exception 之后，要么是声明异常然后让调用者处理；要么是自己捕获异常，不用声明异常，当然调用者也不用处理异常（这种自产自销的方式是不合情理的）。如下所示：

```
public class Test2
{
    public static void main(String[] args)
    {
        try
        {
            throwChecked1();
        } catch (Exception e)
        {
            e.printStackTrace();
        }

        throwChecked2();
    }

    public static void throwChecked1() throws Exception
    {
        throw new Exception("throwChecked1:抛出异常");
```

```
    }

    public static void throwChecked2()
    {
        try
        {
            throw new Exception("throwChecked2:抛出异常");
        } catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

（3）无论函数是否真正的产生异常，都可以抛出RuntimeException异常和Checked Exception异常，调用者需要分别对待。

知识点五：自定义异常的编写

当 JDK 中的异常无法满足需求的时候，我们要设计自己的异常类。下面的过程首先从基本的功能开始，然后逐步的扩展这个异常类。

1 第一步：封装异常基本信息

```
public class MyException extends RuntimeException
{
    private static final long serialVersionUID = 1L;
    public MyException(String message)
    {
        super(message);
    }
}
```

测试代码如下所示：

```
public class Test1
{
    public static void main(String[] args)
    {
        throw new MyException("myException");
    }
}
```

分析：

此时 MyException 能详细的描述异常的特性。但是在下面的情况下，会造成异常信息的丢失：

```
public class Test2
{
    public static void main(String[] args)
    {
        try
        {
```

```
                throw new IllegalArgumentException("illegal arguments");
            }
            catch (Exception e)
            {
                throw new MyException("myException");
            }
        }
}
```

分析：

此时异常抛出的信息是：

Exception in thread "main" MyException: myException

这里只有 MyException 的信息，而 IllegalArgumentException 异常的信息已经丢失了。所以需要继续完善 MyException 类。

2 第二步：封装发生异常的源头

```
public class MyException extends RuntimeException
{

    private static final long serialVersionUID = 1L;

    public MyException(String message, Throwable cause)
    {
        super(message, cause);
    }
    public MyException(String message)
    {
        super(message);
    }
}
```

测试代码：

```
public class Test3
{
    public static void main(String[] args)
    {
        try
        {
            throw new IllegalArgumentException("illegal arguments");
        }
        catch (Exception e)
        {
            throw new MyException("myException", e);
        }
    }
}
```

分析：

Exception in thread "main" MyException: myException
Caused by: java.lang.IllegalArgumentException: illegal arguments

这个时候 MyException 能打印出 IllegalArgumentException 的堆栈信息。但是存在一种问题：我不想使用 JDK 的异常，我想使用自己的异常，但是我这个异常也没有需要补充的信息。那怎么办呢？见下面。

3 第三步：对已经存在异常的封装。

```
public class MyException extends RuntimeException
{

    private static final long serialVersionUID = 1L;

    public MyException()
    {
        super();
    }

    public MyException(String message, Throwable cause)
    {
        super(message, cause);
    }

    public MyException(String message)
    {
        super(message);
    }

    public MyException(Throwable cause)
    {
        super(cause);
    }
}
```

测试代码如下所示：

```
public class Test4
{
    public static void main(String[] args)
    {
        try
        {
            throw new IllegalArgumentException("illegal arguments");
        }
        catch (Exception e)
        {
            throw new MyException(e);
        }
```

```
    }
}
```

分析：

Exception in thread "main" MyException: java.lang.IllegalArgumentException: illegal arguments

Caused by: java.lang.IllegalArgumentException: illegal arguments

不仅抛出了与具体业务更密切的 MyException 异常，还告诉了异常的根源：IllegalArgumentException

知识点六：异常的使用场景

1 情景一抛出异常

自己 new 出一个异常。如果是受检异常，必须 try-catch 进行处理，或者不想处理则在方法上声明，让调用者进行处理。

代码 1：

```
try
{
    //do something    --监控区
}
catch (Exception e)
{
}
```

代码 2：

```
try
{
    throw new Exception("");    --监控区
}
catch (Exception e)
{
}
```

上面的监控区都是：在堆上创建异常对象，然后当前的执行路径被终止，并从当前的环境中弹出对异常对象的引用，此时异常处理线程接管程序，并开始寻找一个恰当的地方来继续执行程序。

2 情景二声明异常

在定义抽象类或者接口的时候，提前占个位，实际上其内部不一定会抛出异常。任何派生类或者接口实现来要么处理异常要么是抛出异常。

3 情景三捕获异常

（1）打印异常的信息（栈轨迹）：例如：

```
catch (Exception e)
{
    e.printStackTrace();
```

```
}
```

（2）重新抛出异常，例如：

```
catch (Exception e)
{
    throw e;
}
```

如果只是把当前异常对象重新抛出，那么 printStackTrace()方法显示的将是原来异常抛出点的栈调用信息，而非重新抛出点的信息，要更新这个信息，可以调用 fillInStackTrace()方法，这将返回一个 Throwable 对象，它是通过把当前调用栈信息填入原来那个异常对象而建立的。

```
public class Test
{
    public static void main(String[] args)
    {
        try
        {
            two();
        } catch (Exception e)
        {
            System.out.println("main 函数捕获 two 的异常，开始打印异常");
            e.printStackTrace(System.out);
        }

        try
        {
            three();
        } catch (Exception e)
        {
            System.out.println("main 函数捕获 three 的异常，开始打印异常");
            e.printStackTrace(System.out);
        }
    }

    public static void one() throws Exception
    {
        System.out.println("one 准备产生异常");
        throw new Exception("one 异常");
    }

    public static void two() throws Exception
    {
        try
        {
```

```
                one();
            } catch (Exception e)
            {
                System.out.println("tow 捕获 one 异常（直接抛出），开始打印异常");
                e.printStackTrace(System.out);
                throw e;
            }
        }

        public static void three() throws Exception
        {
            try
            {
                one();
            } catch (Exception e)
            {
                System.out.println("three 捕获 one 异常（重置异常再抛出），开始打印异常");
                e.printStackTrace(System.out);
                throw (Exception) e.fillInStackTrace();
            }
        }

}
```

输出结果是:

```
one 准备产生异常
tow 捕获 one 异常（直接抛出），开始打印异常
java.lang.Exception: one 异常
        at Test.one(Test.java:28)
        at Test.two(Test.java:35)
        at Test.main(Test.java:8)
main 函数捕获 two 的异常，开始打印异常
java.lang.Exception: one 异常
        at Test.one(Test.java:28)
        at Test.two(Test.java:35)
        at Test.main(Test.java:8)
one 准备产生异常
three 捕获 one 异常（重置异常再抛出），开始打印异常
java.lang.Exception: one 异常
        at Test.one(Test.java:28)
        at Test.three(Test.java:48)
        at Test.main(Test.java:17)
main 函数捕获 three 的异常，开始打印异常
java.lang.Exception: one 异常
        at Test.three(Test.java:53)
```

```
    at Test.main(Test.java:17)
```

（3）重新抛出另外一个异常，例如：

```
catch (Exception e)
{
    throw new MyException("myException");
}
```

有关原来的异常发生点的信息完全丢失，剩下的只是与新抛出点相关的信息了。

（4）异常链

4 情景四 finally 清理

需要注意：捕获异常不同于 finally 清理。

知识点七：异常的捕获和处理

1 异常捕获的结构

分析下面给出的 Java 代码，编译运行后，输出的结果是什么？

```
public class print_message {
    public static void main(String[] args) {
        print();
    }
    static void print(){
        try{
            System.out.println("thank you !");
        }finally{
            System.out.println("I am sorry !");
        }
        }
}
```

A thank you !

B I am sorry !

C thank you !

I am sorry !

D 代码不能编译

参考答案：

C

试题分析：

在Java中，try和catch可以连用，try-catch-finally可以连用，但是try，catch，finally却不能单独使用，如果在程序中只想使用try而不想使用catch也可以，但是try的后面必须有finally。在本题中，try中的打印语句并没有异常发生所以正常输出，对于finally无论有没有异常发生，总是要执行的。

2 finally 块和 return 语句

只要 Java 虚拟机不退出，不管 try 块正常结束，还是遇到异常非正常退出，finally 块总是获得执行机会。

实例一：try 和 finally

```
public class Test {
public static boolean decide(){
try{
                System.out.println("First");
                return true;
    }finally
    {
                System.out.println("Second");
                return false;
     }
}
      public static void main (String [] args){
            if(Test.decide()){
                System.out.println("True");
            }else{
                System.out.println("False");
            }
      }
}
```

结果是:

```
First
Second
False
```

当 Java 程序执行 try 块, catch 块时遇到了 return 语句, return 语句会导致该方法立即结束。系统执行完 return 语句之后，并不会立即结束该方法，而是去寻找异常处理流程中是否包含 finally 块，如果没有 finally 块，方法终止，返回相应的返回值。如果有 finally 块，系统立即开始执行 finally 块，只有当 finally 块执行完成后，系统才会再次跳回来根据 return 语句结束方法。如果 finally 块里使用使用了 return 语句来导致方法结束，则 finally 块已经结束了方法，系统将不会跳回去执行 try 块，catch 块里的代码。

实例二：try-catch和finally

```
public class Test {
public static int getLength(String s ){
      try{
          int l= s.length();
          return l;
      }catch(Exception e)
    {
          return 0;
      }finally{
           return -1;
```

```
        }
}
public static void main(String[] args) {
System.out.println(getLength("tom"));
System.out.println(getLength(null));
    }
}
```

运行结果：

```
-1
-1
```

无论 try 语句是否发生异常，getLength()方法的返回结果都是 finally 中的 return 的值。

实例三：throw和finally

```
public class FianllyFlowTest {
    public static   int test(){
        int count = 5;
        try{
            throw new RuntimeException("测试异常");
        }finally{
            System.out.println("finally块被执行");
            return count;
        }
    }
    public static void main(String[] args) {
    int a = test();
    System.out.println(a);
    }
}
```

运行结果：

finally块被执行

5

当程序执行 try 块，catch 块时遇到 throw 语句是，throw 语句会导致该方法立即结束，系统执行 throw 语句并不会立即抛出异常，而是去寻找该异常处理流程中是否包含 finally 块。如果没有 finally 块，程序立即抛出异常。如果有 finally 块，系统立即开始运行 finally 块，只有当 finally 块执行完成后，系统才会再次跳回来抛出异常。如果 finally 块里使用 return 语句来结束方法，系统将不会跳回去执行 try 块，catch 块抛出异常。

3 finally 与 system.exit()

```
public class Test {
public static int decide(){
        try{
            System.out.println("First");
            System.exit(0);
            }finally{
            System.out.println("Second");
            return 1 ;
        }
    }
    public static void main (String [] args){
        if(Test.decide()==0){
            System.out.println("True");
        }else{
            System.out.println("False");
        }
    }
}
```

结果是：`First`

4 只能 catch 可能抛出的异常

```
import java.io.IOException;
public class CatchTest {
public static void test1(){
        try
        {
            System.out.println("catch");
        }
        catch(IndexOutOfBoundsException ex)
        {
          ex.printStackTrace();
          }
}

    public static void test2(){
        try
        {
            System.out.println("catch");
        }
        catch( NullPointerException ex)
        {
          ex.printStackTrace();
        }
```

```
}

public static void test3(){
        try
        {
            System.out.println("catch");
        }
        catch(IOException ex)
        {
          ex.printStackTrace();
        }
}
public static void test4(){
        try
        {
            System.out.println("catch");
        }
        catch(ClassNotFoundException ex)
        {
          ex.printStackTrace();
        }
}

    public static void test5(){
        try
        {
            System.out.println("catch");
        }
        catch(Exception ex)
        {
          ex.printStackTrace();
        }
}

    public static void main(String[] args) {

    test1();
    test2();
    test3();
    test4();
    test5();
}
}
```

程序企图在捕捉 IOException 和 ClassNotFoundException 两个异常出错误，编译器认为

System.out.println("catch")不可能抛出这两个异常，因此企图捕捉这两个异常是有错的。但是 test1()和 test2()两个方法企图捕捉 IndexOutOfBoundsException，NullPointerException 异常却没有任何错误。

因为 IndexOutOfBoundsException，NullPointerException 是 RuntimeException 的子类，属于运行时异常，而 IOException 和 ClassNotFoundException 属于非运行时异常。根据 Java 语言规范，如果一个 catch 子句试图捕获一个类型为 XxxException 的非运行时异常(Checked)，那么它对应的 try 子句必须可以抛出 XxxException 或者其子类的异常，否则编译器将认为该程序编译错误。而 Runtime 异常是一种非常灵活的异常，它无需显式声明抛出，只要程序有需要，即可以在任何需要的地方使用 try-catch 块来捕捉 Runtime 异常。

知识点八：异常的种类

运行时异常

ArithmeticException	除以 0 等运算错误
ArrayIndexOutOfBoundsException	数组下标出界
ArrayStoreException	数组元素值与元素类型不同
ClassCastException	强制类型转换异常
IllegalArgumentException	调用方法的参数非法
IllegalMonitorStateException	非法监控操作
IllegalStateException	环境或状态错误
IllegalThreadStateException	请求操作与当前线程不兼容
IndexOutOfBoundsException	索引越界
NullPointerException	非法使用空引用
NumberFormatException	字符串非法转化数字格式
SecurityException	安全性
StringIndexOutBoundsException	字符串索引越界
UnsupportedOperationException	操作错误

非运行时异常

ClassNotFoundException	找不到相关类
CloneNotSupportedException	对象不能实现
IllegalAccessException	访问类被拒绝
InstantiationException	创建抽象对象
InsterrupteException	线程被另一个线程中断
NoSuchFieldException	请求的内容不存在
NoSuchMethodException	请求的方法不存在

知识点九：异常的继承关系

JAVA 语言规定，子类重写父类方法时，不能声明抛出比父类方法类型更多、范围更大的异常。也就是说，子类重写父类方法时，子类方法只能声明抛出父类方法所声明抛出的异常的子类。

```
interface      Type1
{
    void test()throws ClassNotFoundException;
```

```
}
interface       Type2
{
    void test()throws NoSuchMethodException;
}

public class Test implements Type1,Type2 {

    public void test()throws ClassNotFoundException,NoSuchMethodException
    {
        System.out.println("wxf");

    }
    public static void main(String[] args) {
        Test t= new Test();
        t.test();
    }
}
```

如上所说：
Test 类实现了 Type1 接口，实现 Type1 接口的 test() 方法时可以声明抛出 ClassNotFoundException异常或者该异常的子类，或者不声明抛出；
Test 类实现了 Type1 接口，实现 Type1 接口的 test() 方法时可以声明抛出 NoSuchMethodException 异常或该异常的子类，或者不声明抛出。
由于Test类同时实现了Type1、Type2两个接口，因此需要同时实现两个接口中的test()方法。只能是上面两种声明抛出的交集，不能声明抛出任何异常。

知识点十：throw 和 throws 的区别

```
public class TestThrow
{
    public static void main(String[] args)
    {
        try
        {
            //调用带throws声明的方法，必须显式捕获该异常
            //否则，必须在main方法中再次声明抛出
            throwChecked(-3);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
        //调用抛出Runtime异常的方法既可以显式捕获该异常，
        //也可不理会该异常
```

```
            throwRuntime(3);
        }
        public static void throwChecked(int a)throws Exception
        {
            if (a > 0)
            {
                //自行抛出Exception异常
                //该代码必须处于try块里，或处于带throws声明的方法中
                throw new Exception("a的值大于0，不符合要求");
            }
        }
        public static void throwRuntime(int a)
        {
            if (a > 0)
            {
                //自行抛出RuntimeException异常，既可以显式捕获该异常
                //也可完全不理会该异常，把该异常交给该方法调用者处理
                throw new RuntimeException("a的值大于0，不符合要求");
            }
        }
}
```

补充：throwChecked 函数的另外一种写法如下所示：

```
public static void throwChecked(int a)
        {
            if (a > 0)
            {
                //自行抛出Exception异常
                //该代码必须处于try块里，或处于带throws声明的方法中
                try
                {
                    throw new Exception("a的值大于0，不符合要求");
                }
                catch (Exception e)
                {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        }
```

注意：此时在 main 函数里面 throwChecked 就不用 try 异常了。

知识点十一：声明方法抛出异常和方法中抛出异常

1 在声明方法时候抛出异常

语法：throws（略）

2 为什么要在声明方法抛出异常?

方法是否抛出异常与方法返回值的类型一样重要。假设方法抛出异常却没有声明该方法将抛出异常，那么客户程序员可以调用这个方法而且不用编写处理异常的代码。那么，一旦出现异常，那么这个异常就没有合适的异常控制器来解决。

为什么抛出的异常一定是已检查异常?

RuntimeException 与 Error 可以在任何代码中产生，它们不需要由程序员显示的抛出，一旦出现错误，那么相应的异常会被自动抛出。遇到 Error，程序员一般是无能为力的；遇到 RuntimeException，那么一定是程序存在逻辑错误，要对程序进行修改；只有已检查异常才是程序员所关心的，程序应该且仅应该抛出或处理已检查异常。而已检查异常是由程序员抛出的，这分为两种情况：客户程序员调用会抛出异常的库函数；客户程序员自己使用 throw 语句抛出异常。

注意：

覆盖父类某方法的子类方法不能抛出比父类方法更多的异常，所以，有时设计父类的方法时会声明抛出异常，但实际的实现方法的代码却并不抛出异常，这样做的目的就是为了方便子类方法覆盖父类方法时可以抛出异常。

3 在方法中如何抛出异常

语法：throw（略）

抛出什么异常?

对于一个异常对象，真正有用的信息是异常的对象类型，而异常对象本身毫无意义。比如一个异常对象的类型是 ClassCastException，那么这个类名就是唯一有用的信息。所以，在选择抛出什么异常时，最关键的就是选择异常的类名能够明确说明异常情况的类。

异常对象通常有两种构造函数：一种是无参数的构造函数；另一种是带一个字符串的构造函数，这个字符串将作为这个异常对象除了类型名以外的额外说明。

为什么要创建自己的异常?

当 Java 内置的异常都不能明确的说明异常情况的时候，需要创建自己的异常。需要注意的是，唯一有用的就是类型名这个信息，所以不要在异常类的设计上花费精力。

4 应该在声明方法抛出异常还是在方法中捕获异常?

处理原则：捕捉并处理哪些知道如何处理的异常，而传递哪些不知道如何处理的异常

5 使用 finally 块释放资源

finally 关键字保证无论程序使用任何方式离开 try 块，finally 中的语句都会被执行。在以下三种情况下会进入 finally 块:

（1） try 块中的代码正常执行完毕。

（2） 在 try 块中抛出异常。

（3） 在 try 块中执行 return、break、continue。

因此，当你需要一个地方来执行在任何情况下都必须执行的代码时，就可以将这些代码放入 finally 块中。当你的程序中使用了外界资源，如数据库连接，文件等，必须将释放这些资源的代码写入 finally 块中。

必须注意的是：在 finally 块中不能抛出异常。JAVA 异常处理机制保证无论在任何情况下必须先执行 finally 块然后再离开 try 块，因此在 try 块中发生异常的时候，JAVA 虚拟机先转到 finally 块执行 finally 块中的代码，finally 块执行完毕后，再向外抛出异常。如果在 finally 块

中抛出异常，try 块捕捉的异常就不能抛出，外部捕捉到的异常就是 finally 块中的异常信息，而 try 块中发生的真正的异常堆栈信息则丢失了。

请看下面的代码：

```
Connection   con = null;
try
{
      con = dataSource.getConnection();
      ……
}
catch(SQLException e)
{
      ……
      throw e;//进行一些处理后再将数据库异常抛出给调用者处理
}
finally
{
      try
      {
            con.close();
      }
      catch(SQLException e)
      {
            e.printStackTrace();
            ……
      }
}
```

运行程序后，调用者得到的信息如下

```
java.lang.NullPointerException
at myPackage.MyClass.method1(methodl.java:266)
```

而不是我们期望得到的数据库异常。这是因为这里的 con 是 null 的关系，在 finally 语句中抛出了 NullPointerException，在 finally 块中增加对 con 是否为 null 的判断可以避免产生这种情况。