# 目录

- Java异常

# 为什么要使用异常

> 首先我们可以明确一点就是异常的处理机制可以确保我们程序的健壮性，提高系统可用率。虽然我们不是特别喜欢看到它，但是我们不能不承认它的地位，作用。

在没有异常机制的时候我们是这样处理的：通过函数的返回值来判断是否发生了异常（这个返回值通常是已经约定好了的），调用该函数的程序负责检查并且分析返回值。虽然可以解决异常问题，但是这样做存在几个缺陷：

> 1、 容易混淆。如果约定返回值为-11111时表示出现异常，那么当程序最后的计算结果真的为-1111呢?
>
> 2、 代码可读性差。将异常处理代码和程序代码混淆在一起将会降低代码的可读性。
>
> 3、 由调用函数来分析异常，这要求程序员对库函数有很深的了解。

在OO中提供的异常处理机制是提供代码健壮的强有力的方式。使用异常机制它能够降低错误处理代码的复杂度，如果不使用异常，那么就必须检查特定的错误，并在程序中的许多地方去处理它。

而如果使用异常，那就不必在方法调用处进行检查，因为异常机制将保证能够捕获这个错误，并且，只需在一个地方处理错误，即所谓的异常处理程序中。

这种方式不仅节约代码，而且把“概述在正常执行过程中做什么事”的代码和“出了问题怎么办”的代码相分离。总之，与以前的错误处理方法相比，异常机制使代码的阅读、编写和调试工作更加井井有条。（摘自《Think in java 》）。

该部分内容选自http://www.cnblogs.com/chenssy/p/3438130.html

## 异常基本定义

> 在《Think in java》中是这样定义异常的：异常情形是指阻止当前方法或者作用域继续执行的问题。在这里一定要明确一点：异常代码某种程度的错误，尽管Java有异常处理机制，但是我们不能以“正常”的眼光来看待异常，异常处理机制的原因就是告诉你：这里可能会或者已经产生了错误，您的程序出现了不正常的情况，可能会导致程序失败！

> 那么什么时候才会出现异常呢？只有在你当前的环境下程序无法正常运行下去，也就是说程序已经无法来正确解决问题了，这时它所就会从当前环境中跳出，并抛出异常。抛出异常后，它首先会做几件事。

> 首先，它会使用new创建一个异常对象，然后在产生异常的位置终止程序，并且从当前环境中弹出对异常对象的引用，这时。异常处理机制就会接管程序，并开始寻找一个恰当的地方来继续执行程序，这个恰当的地方就是异常处理程序。

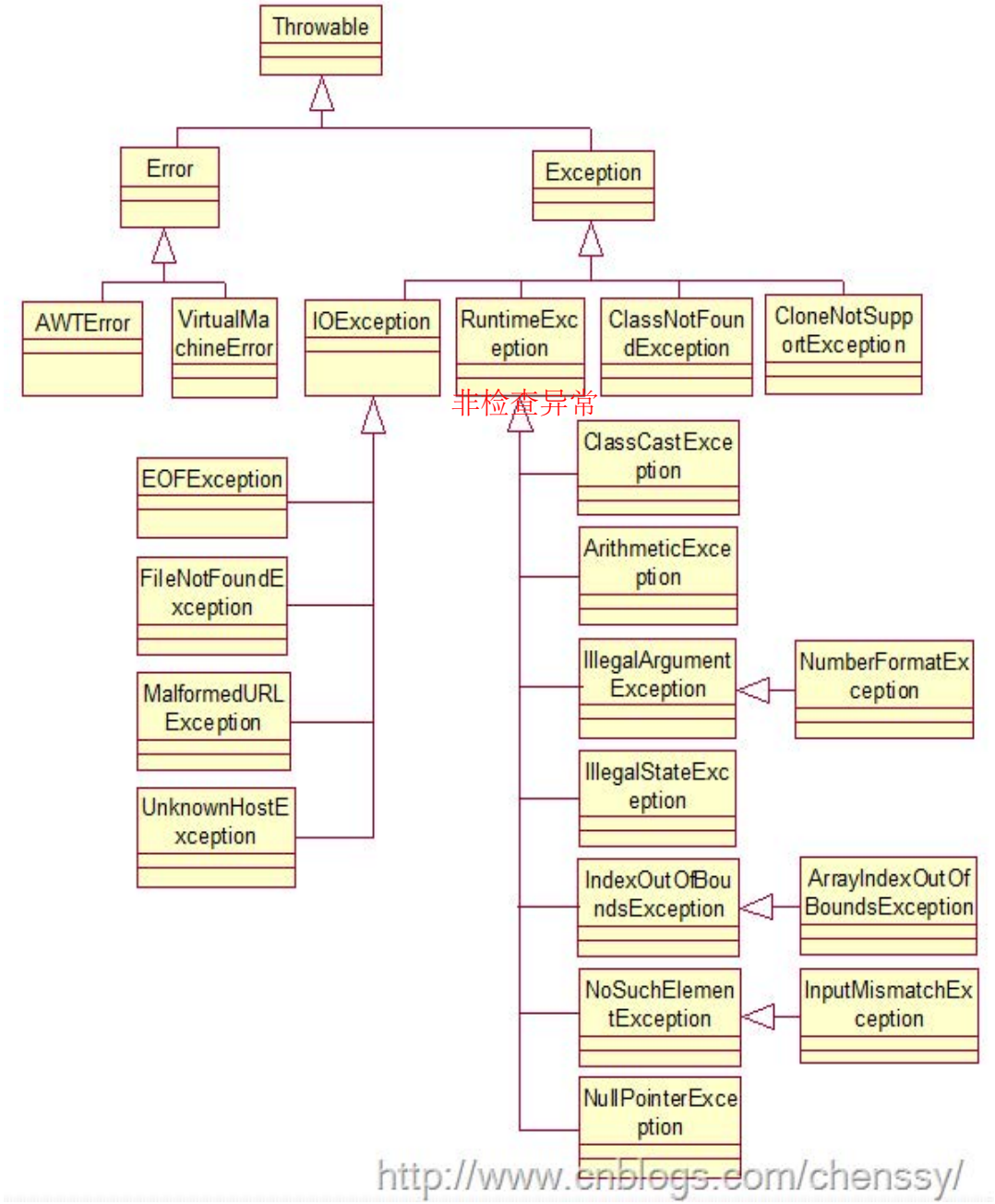
> 总的来说异常处理机制就是当程序发生异常时，它强制终止程序运行，记录异常信息并将这些信息反馈给我们，由我们来确定是否处理异常。

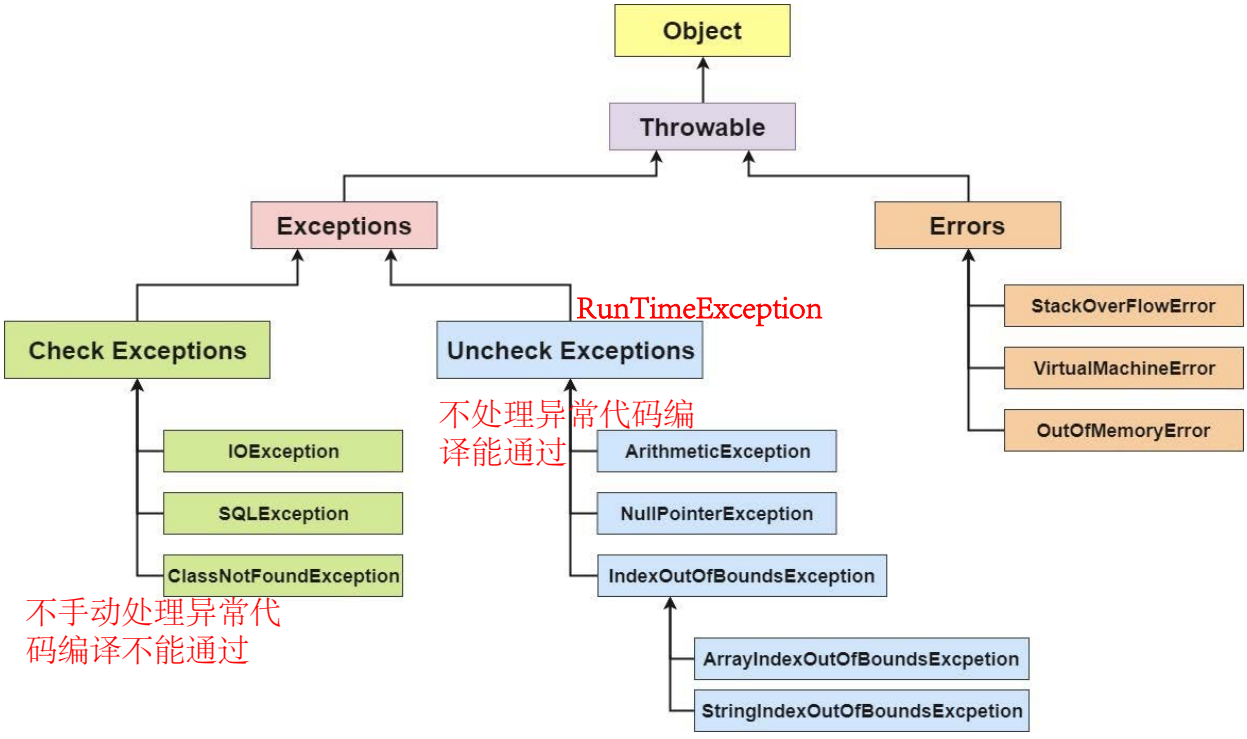
## 异常体系

从上面这幅图可以看出，Throwable是java语言中所有错误和异常的超类（万物即可抛）。它有两个子类：Error、Exception。

Java标准库内建了一些通用的异常，这些类以Throwable为顶层父类。

Throwable又派生出Error类和Exception类。


Throwable
Error
Exception
AWTError
VirtualMachineError
IOException
RuntimeException
ClassNotFoundException
CloneNotSupportException
非检查异常
EOFException
FileNotFoundException
MalformedURLException
UnknownHostException
ClassCastException
ArithmeticException
IllegalArgumentException
NumberFormatException
IllegalStateException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
NoSuchElementException
InputMismatchException
NullPointerException
http://www.cnblogs.com/chenssy/


Object
Throwable
Exceptions
Errors
RunTimeException
Check Exceptions
Uncheck Exceptions
StackOverFlowError
VirtualMachineError
OutOfMemoryError
不处理异常代码编译能通过
IOException
SQLException
ClassNotFoundException
ArithmeticException
NullPointerException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsExcpetion
StringIndexOutOfBoundsExcpetion
不手动处理异常代码编译不能通过

错误：Error类以及他的子类的实例，代表了JVM本身的错误。错误不能被程序员通过代码处理，Error很少出现。因此，程序员应该关注Exception为父类的分支下的各种异常类。

异常：Exception以及他的子类，代表程序运行时发送的各种不期望发生的事件。可以被Java异常处理机制使用，是异常处理的核心。

总体上我们根据Javac对异常的处理要求，将异常类分为2类。

> 非检查异常（unckecked exception）：Error 和 RuntimeException 以及他们的子类。javac在编译时，不会提示和发现这样的异常，不要求在程序处理这些异常。所以如果愿意，我们可以编写代码处理（使用try...catch...finally）这样的异常，也可以不处理。

> 对于这些异常，我们应该修正代码，而不是去通过异常处理器处理。这样的异常发生的原因多半是代码写的有问题。如除0错误ArithmeticException，错误的强制类型转换错误ClassCastException，数组索引越界ArrayIndexOutOfBoundsException，使用了空对象NullPointerException等等。

> 检查异常（checked exception）：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try...catch...finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。

> Connection conn = DriverManager.getConnection(...)，必须处理该代码的异常: java.sql.SQLException，否则javac编译都不能通过
>
> 这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException , IOException,ClassNotFoundException 等。

需要明确的是：检查和非检查是对于javac来说的，这样就很好理解和区分了。这部分内容摘自http://www.importnew.com/26613.html

## 初识异常

异常是在执行某个函数时引发的，而函数又是层级调用，形成调用栈的，因为，只要一个函数发生了异常，那么他的所有的caller都会被异常影响。当这些被影响的函数以异常信息输出时，就形成的了异常追踪栈。

异常最先发生的地方，叫做异常抛出点。

```
public class 异常 {
    public static void main (String [] args )
    {
        System . out. println( "----欢迎使用命令行除法计算器----" ) ;
        CMDCalculate ();
    }
    public static void CMDCalculate ()
```

```
    {
        Scanner scan = new Scanner ( System. in );
        int num1 = scan .nextInt () ;
        int num2 = scan .nextInt () ;
        int result = devide (num1 , num2 ) ;
        System . out. println( "result:" + result) ;
        scan .close () ;
    }
    public static int devide (int num1, int num2 ){
        return num1 / num2 ;
    }

//    ----欢迎使用命令行除法计算器----
//            1
//            0
//    Exception in thread "main" java.lang.ArithmeticException: / by zero
//    at com.javase.异常.异常.devide(异常.java:24)
//    at com.javase.异常.异常.CMDCalculate(异常.java:19)
//    at com.javase.异常.异常.main(异常.java:12)
```

// ----欢迎使用命令行除法计算器---- // r // Exception in thread "main" java.util.InputMismatchException // at java.util.Scanner.throwFor(Scanner.java:864) // at java.util.Scanner.next(Scanner.java:1485) // at java.util.Scanner.nextInt(Scanner.java:2117) // at java.util.Scanner.nextInt(Scanner.java:2076) // at com.javase.异常.异常.CMDCalculate(异常.java:17) // at com.javase.异常.异常.main(异常.java:12)

[外链图片转存失败(img-9rqUQJQj-1569073569354)(http://incdn1.b0.upaiyun.com/2017/09/0b3e4ca2f4cf8d7116c7ad354940601f.png)]

从上面的例子可以看出，当devide函数发生除0异常时，devide函数将抛出ArithmeticException异常，因此调用他的CMDCalculate函数也无法正常完成，因此也发送异常，而CMDCalculate的caller——main 因为CMDCalculate抛出异常，也发生了异常，这样一直向调用栈的栈底回溯。

这种行为叫做异常的冒泡，异常的冒泡是为了在当前发生异常的函数或者这个函数的caller中找到最近的异常处理程序。由于这个例子中没有使用任何异常处理机制，因此异常最终由main函数抛给JRE，导致程序终止。

> 上面的代码不使用异常处理机制，也可以顺利编译，因为2个异常都是非检查异常。但是下面的例子就必须使用异常处理机制，因为异常是检查异常。

代码中我选择使用throws声明异常，让函数的调用者去处理可能发生的异常。但是为什么只throws了IOException呢？因为FileNotFoundException是IOException的子类，在处理范围内。

## 异常和错误

下面看一个例子

```
//错误即error一般指jvm无法处理的错误
//异常是Java定义的用于简化错误处理流程和定位错误的一种工具。
public class 错误和错误 {
    Error error = new Error();

    public static void main(String[] args) {
        throw new Error();
    }

    //下面这四个异常或者错误有着不同的处理方法
    public void error1 (){
        //编译期要求必须处理，因为这个异常是最顶层异常，包括了检查异常，必须要处理
        try {
            throw new Throwable();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        }
    }
    //Exception也必须处理。否则报错，因为检查异常都继承自exception，所以默认需要捕
捉。
    public void error2 (){
        try {
            throw new Exception();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //error可以不处理，编译不报错,原因是虚拟机根本无法处理，所以啥都不用做
    public void error3 (){
        throw new Error();
    }

    //runtimeexception众所周知编译不会报错
    public void error4 (){
        throw new RuntimeException();
    }
//    Exception in thread "main" java.lang.Error
//    at com.javase.异常.错误.main(错误.java:11)

}
```

# 异常的处理方式

在编写代码处理异常时，对于检查异常，有2种不同的处理方式：

> 使用try…catch…finally语句块处理它。

> 或者，在函数签名中使用throws 声明交给函数调用者caller去解决。

下面看几个具体的例子，包括error，exception和throwable

上面的例子是运行时异常，不需要显示捕获。 下面这个例子是可检查异常需，要显示捕获或者抛出。

```
@Test
public void testException() throws IOException
{
 //FileInputStream的构造函数会抛出FileNotFoundException

  FileInputStream fileIn = new FileInputStream("E:\\a.txt"); 这属于检查型异常，必须处理

 int word;
 //read方法会抛出IOException
 while((word =  fileIn.read())!=-1)
 {
   System.out.print((char)word);
 }
 //close方法会抛出IOException
}fileIn.close();
```

一般情况下的处理方式 try catch finally

```
public class 异常处理方式 {

@Test
public void main() {
    try{
        //try块中放可能发生异常的代码。
        InputStream inputStream = new FileInputStream("a.txt");

        //如果执行完try且不发生异常，则接着去执行finally块和finally后面的代码（如果有的话）。
        int i = 1/0;
        //如果发生异常，则尝试去匹配catch块。
        throw new SQLException();
        //使用1.8jdk同时捕获多个异常，runtimeexception也可以捕获。只是捕获后虚拟机也无法处理，所以不建议捕获。
    }catch(SQLException | IOException | ArrayIndexOutOfBoundsException exception){
        System.out.println(exception.getMessage());
        //每一个catch块用于捕获并处理一个特定的异常，或者这异常类型的子类。Java7中可以将多个异常声明在一个catch中。

        //catch后面的括号定义了异常类型和异常参数。如果异常与之匹配且是最先匹配到的，则虚拟机将使用这个catch块来处理异常。

        //在catch块中可以使用这个块的异常参数来获取异常的相关信息。异常参数是这个catch块中的局部变量，其它块不能访问。

        //如果当前try块中发生的异常在后续的所有catch中都没捕获到，则先去执行finally，然后到这个函数的外部caller中去匹配异常处理器。
```

```
        //如果try中没有发生异常，则所有的catch块将被忽略。

    }catch(Exception exception){
        System.out.println(exception.getMessage());
        //...
    }finally{
        //finally块通常是可选的。
        //无论异常是否发生，异常是否匹配被处理，finally都会执行。

        //finally主要做一些清理工作，如流的关闭，数据库连接的关闭等。
    }
```

一个try至少要跟一个catch或者finally

```
    try {
        int i = 1;
    }finally {
        //一个try至少要有一个catch块，否则， 至少要有1个finally块。但是finally不
是用来处理异常的，finally不会捕获异常。
    }
}
```

异常出现时该方法后面的代码不会运行，即使异常已经被捕获。这里举出一个奇特的例子，在catch里再次使用try catch finally

```
@Test
public void test() {
    try {
        throwE();
        System.out.println("我前面抛出异常了");
        System.out.println("我不会执行了");
    } catch (StringIndexOutOfBoundsException e) {
        System.out.println(e.getCause());
    }catch (Exception ex) {
    //在catch块中仍然可以使用try catch finally
        try {
            throw new Exception();
        }catch (Exception ee) {

        }finally {
            System.out.println("我所在的catch块没有执行，我也不会执行的");
        }
    }
}
//在方法声明中抛出的异常必须由调用方法处理或者继续往上抛，
// 当抛到jre时由于无法处理终止程序
public void throwE (){
//        Socket socket = new Socket("127.0.0.1", 80);

        //手动抛出异常时，不会报错，但是调用该方法的方法需要处理这个异常，否则会出
```

```
错。
//        java.lang.StringIndexOutOfBoundsException
//        at com.javase.异常.异常处理方式.throwE(异常处理方式.java:75)
//        at com.javase.异常.异常处理方式.test(异常处理方式.java:62)
        throw new StringIndexOutOfBoundsException();
    }
```

其实有的语言在遇到异常后仍然可以继续运行

> 有的编程语言当异常被处理后，控制流会恢复到异常抛出点接着执行，这种策略叫做：resumption model of exception handling（恢复式异常处理模式）
>
> 而Java则是让执行流恢复到处理了异常的catch块后接着执行，这种策略叫做：termination model of exception handling（终结式异常处理模式）

## "不负责任"的throws

throws是另一种处理异常的方式，它不同于try...catch...finally，throws仅仅是将函数中可能出现的异常向调用者声明，而自己则不具体处理。

采取这种异常处理的原因可能是：方法本身不知道如何处理这样的异常，或者说让调用者处理更好，调用者需要为可能发生的异常负责。

```
public void foo() throws ExceptionType1 , ExceptionType2 ,ExceptionTypeN
{
     //foo内部可以抛出 ExceptionType1 , ExceptionType2 ,ExceptionTypeN 类的异常，或者他们的子类的异常对象。
}
```

## 纠结的finally

finally块不管异常是否发生，只要对应的try执行了，则它一定也执行。只有一种方法让finally块不执行：System.exit()。因此finally块通常用来做资源释放操作：关闭文件，关闭数据库连接等等。

良好的编程习惯是：在try块中打开资源，在finally块中清理释放这些资源。

需要注意的地方:

1、finally块没有处理异常的能力。处理异常的只能是catch块。

2、在同一try...catch...finally块中 ，如果try中抛出异常，且有匹配的catch块，则先执行catch块，再执行finally块。如果没有catch块匹配，则先执行finally，然后去外面的调用者中寻找合适的catch块。

3、在同一try...catch...finally块中，try发生异常，且匹配的catch块中处理异常时也抛出异常，那么后面的finally也会执行：首先执行finally块，然后去外围调用者中寻找合适的catch块。

```
public class finally使用 {
    public static void main(String[] args) {
        try {
            throw new IllegalAccessException();
        }catch (IllegalAccessException e) {
            // throw new Throwable();
            //此时如果再抛异常，finally无法执行，只能报错。
            //finally无论何时都会执行
            //除非我显示调用。此时finally才不会执行
            System.exit(0);

        }finally {
            System.out.println("算你狠");
        }
    }
}
```

## throw：JRE也使用的关键字

throw exceptionObject

程序员也可以通过throw语句手动显式的抛出一个异常。throw语句的后面必须是一个异常对象。

throw 语句必须写在函数中，执行throw 语句的地方就是一个异常抛出点，==它和由JRE自动形成的异常抛出点没有任何差别。==

```
public void save(User user)
{
      if(user  == null)
          throw new IllegalArgumentException("User对象为空");
      //......

}
```

后面开始的大部分内容都摘自http://www.cnblogs.com/lulipro/p/7504267.html

该文章写的十分细致到位，令人钦佩，是我目前为之看到关于异常最详尽的文章，可以说是站在巨人的肩膀上了。

## 异常调用链

异常的链化

在一些大型的，模块化的软件开发中，一旦一个地方发生异常，则如骨牌效应一样，将导致一连串的异常。假设B模块完成自己的逻辑需要调用A模块的方法，如果A模块发生异常，则B也将不能完成而发生异常。

==但是B在抛出异常时，会将A的异常信息掩盖掉，这将使得异常的根源信息丢失。异常的链化可以将多个模块的异常串联起来，使得异常信息不会丢失。==

> 异常链化:以一个异常对象为参数构造新的异常对象。新的异对象将包含先前异常的信息。这项技术主要是异常类的一个带Throwable参数的函数来实现的。这个当做参数的异常，我们叫他根源异常（cause）。

查看Throwable类源码，可以发现里面有一个Throwable字段cause，就是它保存了构造时传递的根源异常参数。这种设计和链表的结点类设计如出一辙，因此形成链也是自然的了。

```
public class Throwable implements Serializable {
    private Throwable cause = this;

    public Throwable(String message, Throwable cause) {
        fillInStackTrace();
        detailMessage = message;
        this.cause = cause;
    }
     public Throwable(Throwable cause) {
        fillInStackTrace();
        detailMessage = (cause==null ? null : cause.toString());
        this.cause = cause;
    }

    //........
}
```

下面看一个比较实在的异常链例子哈

```
public class 异常链 {
    @Test
    public void test() {
        C();
    }
    public void A () throws Exception {
        try {
            int i = 1;
            i = i / 0;
            //当我注释掉这行代码并使用B方法抛出一个error时，运行结果如下
//            四月 27, 2018 10:12:30 下午
org.junit.platform.launcher.core.ServiceLoaderTestEngineRegistry
loadTestEngines
//            信息: Discovered TestEngines with IDs: [junit-jupiter]
//            java.lang.Error: B也犯了个错误
//            at com.javase.异常.异常链.B(异常链.java:33)
```

```
//            at com.javase.异常.异常链.C(异常链.java:38)
//            at com.javase.异常.异常链.test(异常链.java:13)
//            at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
//            Caused by: java.lang.Error
//            at com.javase.异常.异常链.B(异常链.java:29)

        }catch (ArithmeticException e) {
            //这里通过throwable类的构造方法将最底层的异常重新包装并抛出，此时注入了A方法的信息。最后打印栈信息时可以看到caused by
            A方法的异常。
            //如果直接抛出，栈信息打印结果只能看到上层方法的错误信息，不能看到其实是A发生了错误。
            //所以需要包装并抛出
            throw new Exception("A方法计算错误", e);
        }

    }
    public void B () throws Exception,Error {
        try {
            //接收到A的异常，
            A();
            throw new Error();
        }catch (Exception e) {
            throw e;
        }catch (Error error) {
            throw new Error("B也犯了个错误", error);
        }
    }
    public void C () {
        try {
            B();
        }catch (Exception | Error e) {
            e.printStackTrace();
        }

    }

    //最后结果
//    java.lang.Exception: A方法计算错误
//    at com.javase.异常.异常链.A(异常链.java:18)
//    at com.javase.异常.异常链.B(异常链.java:24)
//    at com.javase.异常.异常链.C(异常链.java:31)
//    at com.javase.异常.异常链.test(异常链.java:11)
//    省略
//    Caused by: java.lang.ArithmeticException: / by zero
//    at com.javase.异常.异常链.A(异常链.java:16)
//            ... 31 more
}
```

# 自定义异常

如果要自定义异常类，则扩展Exception类即可，因此这样的自定义异常都属于检查异常（checked exception）。如果要自定义非检查异常，则扩展自RuntimeException。

按照国际惯例，自定义的异常应该总是包含如下的构造函数：

一个无参构造函数 一个带有String参数的构造函数，并传递给父类的构造函数。 一个带有String参数和Throwable参数，并都传递给父类构造函数 一个带有Throwable 参数的构造函数，并传递给父类的构造函数。 下面是IOException类的完整源代码，可以借鉴。

```
public class IOException extends Exception
{
    static final long serialVersionUID = 7818375828146090155L;

    public IOException()
    {
        super();
    }

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

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

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

https://blog.csdn.net/peanutwzk/article/details/78934392

# 异常的注意事项

异常的注意事项

> 当子类重写父类的带有 throws声明的函数时，其throws声明的异常必须在父类异常的可控范围内——用于处理父类的throws方法的异常处理器，必须也适用于子类的这个带throws方法 。这是为了支持多态。
>
> 例如，父类方法throws 的是2个异常，子类就不能throws 3个及以上的异常。父类throws IOException，子类就必须throws IOException或者IOException的子类。

至于为什么？我想，也许下面的例子可以说明。

```
class Father
{
```

```
    public void start() throws IOException
    {
        throw new IOException();
    }
}

class Son extends Father
{
    public void start() throws Exception
    {
        throw new SQLException();
    }
}
```

/**********假设上面的代码是允许的（实质是错误的）**********/

```
class Test
{
    public static void main(String[] args)
    {
        Father[] objs = new Father[2];
        objs[0] = new Father();
        objs[1] = new Son();

        for(Father obj:objs)
        {
        //因为Son类抛出的实质是SQLException，而IOException无法处理它。
        //那么这里的try。。catch就不能处理Son中的异常。
        //多态就不能实现了。
            try {
                 obj.start();
            }catch(IOException)
            {
                 //处理IOException
            }
         }
   }
}
```

==Java的异常执行流程是线程独立的，线程之间没有影响==

> Java程序可以是多线程的。每一个线程都是一个独立的执行流，独立的函数调用栈。如果程序只有一个线程，那么没有被任何代码处理的异常 会导致程序终止。如果是多线程的，那么没有被任何代码处理的异常仅仅会导致异常所在的线程结束。
>
> 也就是说，Java中的异常是线程独立的，线程的问题应该由线程自己来解决，而不要委托到外部，也不会直接影响到其它线程的执行。

下面看一个例子

```
public class 多线程的异常 {
    @Test
    public void test() {
        go();
    }
    public void go () {
        ExecutorService executorService = Executors.newFixedThreadPool(3);
        for (int i = 0;i <= 2;i ++) {
            int finalI = i;
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            executorService.execute(new Runnable() {
                @Override
                //每个线程抛出异常时并不会影响其他线程的继续执行
                public void run() {
                    try {
                        System.out.println("start thread" + finalI);
                        throw new Exception();
                    }catch (Exception e) {
                        System.out.println("thread" + finalI + " go wrong");
                    }
                }
            });
        }
//        结果:
//        start thread0
//        thread0 go wrong
//        start thread1
//        thread1 go wrong
//        start thread2
//        thread2 go wrong
    }
}
```

## 当finally遇上return：

很多try finally return不用死记硬背，看字节码就能理解执行顺序

首先一个不容易理解的事实：
在 try块中即便有return，break，continue等改变执行流的语句，finally也会执行。
执行顺序：try中先执行，try中return语句表达式先执行(++i)这种，但是数值暂时不返回，执行finally中语句，最后返回return表达式数值

```
public static void main(String[] args)
{
    int re = bar();
    System.out.println(re);
}
private static int bar()
{
    try{
```

0 iconst_5  try块中的常量5入操作数栈
1 istore_0  操作数栈中5存入局部变量表的0位置
2 bipush 10  finally块中的常量10入操作数栈
4 istore_1  操作数栈中10存入局部变量表的0位置
5 iload_0 局部变量表0位置数值5放入操作数栈
6 ireturn  try中return返回给调用者数值5

根据字节码异常表0-2之间的是可能发生异常的字节码，如果0-2字节码之间发生异常跳到字节码7位置

7 astore_2  创建异常对象存放到局部变量表2位置
8 bipush 10 finally块中的常量5入操作数栈
10 istore_3 操作数栈中10存入局部变量表的3位置
11 aload_2  局部变量表2位置异常对象放入操作数栈
12 athrow 抛出异常到调用者

```
        return 5;
    } finally{
       int i=10;
    }
}
/*输出:
finally
*/
```

很多人面对这个问题时，总是在归纳执行的顺序和规律，不过我觉得还是很难理解。我自己总结了一个方法。用如下GIF图说明。

[外链图片转存失败(img-SceF4t85-1569073569354)
(http://incdn1.b0.upaiyun.com/2017/09/0471c2805ebd5a463211ced478eaf7f8.gif)]

也就是说：try...catch...finally中的return 只要能执行，就都执行了，他们共同向同一个内存地址（假设地址是0×80）写入返回值，后执行的将覆盖先执行的数据，而真正被调用者取的返回值就是最后一次写入的。那么，按照这个思想，下面的这个例子也就不难理解了。

finally中的return 会覆盖 try 或者catch中的返回值。

```
public static void main(String[] args)
    {
        int result;

        result  =  foo();
        System.out.println(result);    /////////9

        result = bar();
        System.out.println(result);    /////////2
    }

    @SuppressWarnings("finally")
    public static int foo()
    {
        trz{
            int a = 5 / 0;
       } catch (Exception e){
                    return 8;
                   } finally{
           return 9;
        }

    }

    @SuppressWarnings("finally")
    public static int bar()
    {
        try {
            return 1;
```

```
 0 iconst_5
 1 iconst_0
 2 idiv
 3 istore_0
 4 bipush 9
 6 ireturn
 7 astore_0
 8 bipush 8     8放入操作数栈
10 istore_1     8从操作数栈中放到局部变量表1位置
11 bipush 9     9放入操作数栈
13 ireturn      返回局部变量表中9
14 astore_2
15 bipush 9
17 ireturn
其余字节码通过异常表pc的start和end判断
```

```
0 iconst_1  1整数放到操作数栈
1 istore_0  1整数从操作数栈中放到局部变量表0位置
2 iconst_2  2整数放到操作数栈
3 ireturn   2整数从操作数栈中弹出方法返回
下面是异常处理
4 astore_1 创建异常放到局部变量表中1位置，即使有异常比如try中int a=1/0.但是下面finally中return抑制了异常，程序正常执行。
5 iconst_2 2整数放到操作数栈
6 ireturn  2整数从操作数栈中弹出方法返回
```

```
        }finally {
            return 2;
        }
    }
```

finally中的return会抑制（消灭）前面try或者catch块中的异常

```
class TestException
{
    public static void main(String[] args)
    {
        int result;
        try{
            result = foo();
            System.out.println(result);           //输出100
        } catch (Exception e){
            System.out.println(e.getMessage());    //没有捕获到异常
        }

        try{
            result  = bar();
            System.out.println(result);           //输出100
        } catch (Exception e){
            System.out.println(e.getMessage());    //没有捕获到异常
        }
    }

    //catch中的异常被抑制
    @SuppressWarnings("finally")
    public static int foo() throws Exception
    {
        try {
            int a = 5/0;
            return 1;
        }catch(ArithmeticException amExp) {
            throw new Exception("我将被忽略，因为下面的finally中使用了return");
        }finally {
            return 100;
                                                                    }
                                                                    }
```

```
  //try中的异常被抑制 ： 虽然产生了异常对象，被放入局部变量表中，但是不用字节码指令athrow进行抛出，而是通过return语句返回操作数栈中数值。
  @SuppressWarnings("finally")
  public static int bar() throws Exception  {

        try {
          int a = 5/0;
        }finally {

            return 100;
        }
    }
}
```

```
 0 iconst_5 操作数栈中方5
 1 iconst_0 操作数栈中方0
 2 idiv 弹出两个栈顶元素，相除结果存入栈顶
 3 istore_0 栈顶结果存入局部变量表中
 4 bipush 100 100入操作数栈
 6 ireturn  返回100
0-4之间可能抛出异常，7处字节码处理
 7 astore_1  异常对象放入局部变量表1位置
 8 bipush 100  100放入操作数栈
10 ireturn 没有向上抛出局部变量表中异常对象，
而是直接弹出操作数栈中100返回。
```

finally中的异常会覆盖（消灭）前面try或者catch中的异常

```
class TestException
{
    public static void main(String[] args)
    {
        int result;
        try{
            result = foo();
        } catch (Exception e){
            System.out.println(e.getMessage());    //输出：我是finaly中的
Exception
        }

        try{
            result  = bar();
        } catch (Exception e){
            System.out.println(e.getMessage());    //输出：我是finaly中的
Exception
        }
    }

    //catch中的异常被抑制
    @SuppressWarnings("finally")
    public static int foo() throws Exception
    {
        try {
            int a = 5/0;
            return 1;
        }catch(ArithmeticException amExp) {
            throw new Exception("我将被忽略，因为下面的finally中抛出了新的异
常");
        }finally {
            throw new Exception("我是finaly中的Exception");
        }
    }

    //try中的异常被抑制
    @SuppressWarnings("finally")
    public static int bar() throws Exception
    {
        try {
            int a = 5/0;
            return 1;
        }finally {
            throw new Exception("我是finaly中的Exception");
        }

    }
}
```

上面的3个例子都异于常人的编码思维，因此我建议：

> 不要在fianlly中使用return。

> 不要在finally中抛出异常。

> 减轻finally的任务，不要在finally中做一些其它的事情，finally块仅仅用来释放资源是最合适的。

> 将尽量将所有的return写在函数的最后面，而不是try … catch … finally中。

## JAVA异常常见面试题

下面是我个人总结的在Java和J2EE开发者在面试中经常被问到的有关Exception和Error的知识。在分享我的回答的时候，我也给这些问题作了快速修订，并且提供源码以便深入理解。我总结了各种难度的问题，适合新手码农和高级Java码农。如果你遇到了我列表中没有的问题，并且这个问题非常好，请在下面评论中分享出来。你也可以在评论中分享你面试时答错的情况。

**1) Java中什么是Exception?** 这个问题经常在第一次问有关异常的时候或者是面试菜鸟的时候问。我从来没见过面高级或者资深工程师的时候有人问这玩意，但是对于菜鸟，是很愿意问这个的。简单来说，异常是Java传达给你的系统和程序错误的方式。在java中，异常功能是通过实现比如Throwable，Exception，RuntimeException之类的类，然后还有一些处理异常时候的关键字，比如throw，throws，try，catch，finally之类的。 所有的异常都是通过Throwable衍生出来的。Throwable把错误进一步划分为 java.lang.Exception 和 java.lang.Error. java.lang.Error 用来处理系统错误，例如 java.lang.StackOverFlowError 之类的。然后 Exception用来处理程序错误，请求的资源不可用等等。

**2) Java中的检查型异常和非检查型异常有什么区别?**

这又是一个非常流行的Java异常面试题，会出现在各种层次的Java面试中。检查型异常和非检查型异常的主要区别在于其处理方式。检查型异常需要使用try, catch和finally关键字在编译期进行处理，否则会出现编译器会报错。对于非检查型异常则不需要这样做。Java中所有继承自java.lang.Exception类的异常都是检查型异常，所有继承自RuntimeException的异常都被称为非检查型异常。

**3) Java中的NullPointerException和ArrayIndexOutOfBoundException之间有什么相同之处?**

在Java异常面试中这并不是一个很流行的问题，但会出现在不同层次的初学者面试中，用来测试应聘者对检查型异常和非检查型异常的概念是否熟悉。顺便说一下，该题的答案是，这两个异常都是非检查型异常，都继承自RuntimeException。该问题可能会引出另一个问题，即Java和C的数组有什么不同之处，因为C里面的数组是没有大小限制的，绝对不会抛出ArrayIndexOutOfBoundException。

**4)在Java异常处理的过程中，你遵循的那些最好的实践是什么?**

这个问题在面试技术经理是非常常见的一个问题。因为异常处理在项目设计中是非常关键的，所以精通异常处理是十分必要的。异常处理有很多最佳实践，下面列举集中，它们提高你代码的健壮性和灵活性：

1) 调用方法的时候返回布尔值来代替返回null，这样可以 NullPointerException。由于空指针是java异常里最恶心的异常

2) catch块里别不写代码。空catch块是异常处理里的错误事件，因为它只是捕获了异常，却没有任何处理或者提示。通常你起码要打印出异常信息，当然你最好根据需求对异常信息进行处理。

3)能抛受控异常（checked Exception）就尽量不抛受非控异常(checked Exception)。通过去掉重复的异常处理代码，可以提高代码的可读性。

4) 绝对不要让你的数据库相关异常显示到客户端。由于绝大多数数据库和SQLException异常都是受控异常，在Java中，你应该在DAO层把异常信息处理，然后返回处理过的能让用户看懂并根据异常提示信息改正操作的异常信息。

5) 在Java中，一定要在数据库连接，数据库查询，流处理后，在finally块中调用close()方法。

**5) 既然我们可以用RuntimeException来处理错误，那么你认为为什么Java中还存在检查型异常?**

这是一个有争议的问题，在回答该问题时你应当小心。虽然他们肯定愿意听到你的观点，但其实他们最感兴趣的还是有说服力的理由。我认为其中一个理由是，存在检查型异常是一个设计上的决定，受到了诸如C++等比Java更早编程语言设计经验的影响。绝大多数检查型异常位于java.io包内，这是合乎情理的，因为在你请求了不存在的系统资源的时候，一段强壮的程序必须能够优雅的处理这种情况。通过把IOException声明为检查型异常，Java 确保了你能够优雅的对异常进行处理。另一个可能的理由是，可以使用catch或finally来确保数量受限的系统资源（比如文件描述符）在你使用后尽早得到释放。 Joshua Bloch编写的 Effective Java 一书 中多处涉及到了该话题，值得一读。

**6) throw 和 throws这两个关键字在java中有什么不同?**

一个java初学者应该掌握的面试问题。 throw 和 throws乍看起来是很相似的尤其是在你还是一个java初学者的时候。尽管他们看起来相似，都是在处理异常时候使用到的。但在代码里的使用方法和用到的地方是不同的。throws总是出现在一个函数头中，用来标明该成员函数可能抛出的各种异常，你也可以申明未检查的异常，但这不是编译器强制的。如果方法抛出了异常那么调用这个方法的时候就需要将这个异常处理。另一个关键字 throw 是用来抛出任意异常的，按照语法你可以抛出任意 Throwable (i.e. Throwable 或任何Throwable的衍生类) , throw可以中断程序运行，因此可以用来代替 return . 最常见的例子是用 throw 在一个空方法中需要return的地方抛出 UnSupportedOperationException 代码如下：

```
123 private``static  void  show() {``throw``new
UnsupportedOperationException(``"Notyet implemented"``);``}
```

可以看下这篇 文章查看这两个关键字在java中更多的差异 。

**7) 什么是“异常链”?**

“异常链”是Java中非常流行的异常处理概念，是指在进行一个异常处理时抛出了另外一个异常，由此产生了一个异常链条。该技术大多用于将“ 受检查异常” （ checked exception）封装成为“非受检查异常”（unchecked exception)或者RuntimeException。顺便说一下，如果因为因为异常你决定抛出一个新的异常，你一定要包含原有的异常，这样，处理程序才可以通过getCause()和initCause()方法来访问异常最终的根源。

**) 你曾经自定义实现过异常吗？怎么写的?**

很显然，我们绝大多数都写过自定义或者业务异常，像AccountNotFoundException。在面试过程中询问这个Java异常问题的主要原因是去发现你如何使用这个特性的。这可以更准确和精致的去处理异常，当然这也跟你选择checked还是unchecked exception息息相关。通过为每一个特定的情况创建一个特定的异常，你就为调用者更好的处理异常提供了更好的选择。相比通用异常（general exception），我更倾向更为精确的异常。大量的创建自定义异常会增加项目class的个数，因此，在自定义异常和通用异常之间维持一个平衡是成功的关键。

**9) JDK7中对异常处理做了什么改变?**

这是最近新出的Java异常处理的面试题。JDK7中对错误(Error)和异常(Exception)处理主要新增加了2个特性，一是在一个catch块中可以出来多个异常，就像原来用多个catch块一样。另一个是自动化资源管理(ARM), 也称为try-with-resource块。这2个特性都可以在处理异常时减少代码量，同时提高代码的可读性。对于这些特性了解，不仅帮助开发者写出更好的异常处理的代码，也让你在面试中显的更突出。我推荐大家读一下Java 7攻略，这样可以更深入的了解这2个非常有用的特性。

**10) 你遇到过 OutOfMemoryError 错误嘛？你是怎么搞定的?**

这个面试题会在面试高级程序员的时候用，面试官想知道你是怎么处理这个危险的OutOfMemoryError错误的。必须承认的是，不管你做什么项目，你都会碰到这个问题。所以你要是说没遇到过，面试官肯定不会买账。要是你对这个问题不熟悉，甚至就是没碰到过，而你又有3、4年的Java经验了，那么准备好处理这个问题吧。在回答这个问题的同时，你也可以借机向面试秀一下你处理内存泄露、调优和调试方面的牛逼技能。我发现掌握这些技术的人都能给面试官留下深刻的印象。

**11) 如果执行finally代码块之前方法返回了结果，或者JVM退出了，finally块中的代码还会执行吗?**

这个问题也可以换个方式问："如果在try或者finally的代码块中调用了System.exit()，结果会是怎样"。了解finally块是怎么执行的，即使是try里面已经使用了return返回结果的情况，对了解Java的异常处理都非常有价值。只有在try里面是有System.exit(0)来退出JVM的情况下finally块中的代码才不会执行。

**12)Java中final,finalize,finally关键字的区别**

这是一个经典的Java面试题了。我的一个朋友为Morgan Stanley招电信方面的核心Java开发人员的时候就问过这个问题。final和finally是Java的关键字，而finalize则是方法。final关键字在创建不可变的类的时候非常有用，只是声明这个类是final的。而finalize()方法则是垃圾回收器在回收一个对象前调用，但也Java规范里面没有保证这个方法一定会被调用。finally关键字是唯一一个和这篇文章讨论到的异常处理相关的关键字。在你的产品代码中，在关闭连接和资源文件的是时候都必须要用到finally块。

# 参考文章


https://www.xuebuyuan.com/3248044.html https://www.jianshu.com/p/49d2c3975c56
http://c.biancheng.net/view/1038.html
https://blog.csdn.net/Lisiluan/article/details/88745820
https://blog.csdn.net/michaelgo/article/details/82790253