异常的深入研究与分析（1）

栏目: Java基础 作者: admin 日期: 2015-05-30 评论: 0 点击: 3,381 次

前言

本文是异常内容的集大成者，力求全面，深入的异常知识研究与分析。本文由金丝燕网独家撰写，参考众多网上资源，经过内容辨别取舍，文字格式校验等步骤编辑而成，以飨读者。对于本文的内容，建议小白需要多多思考力求掌握，对于老手只需意会温故知新。对于本文的内容，属于基础知识研究范畴，切勿以为读完此文就能将异常知识掌握到家。切记：操千曲而后晓声，观千剑而后识器，所以我觉得没有大量的源码阅读经验，你很难知道什么时候需要自定义异常，什么时候需要抛出异常。

异常机制概述

异常机制是指当程序出现错误后，程序如何处理。具体来说，异常机制提供了程序退出的安全通道。当出现错误后，程序执行的流程发生改变，程序的控制权转移到异常处理器。

异常处理的流程

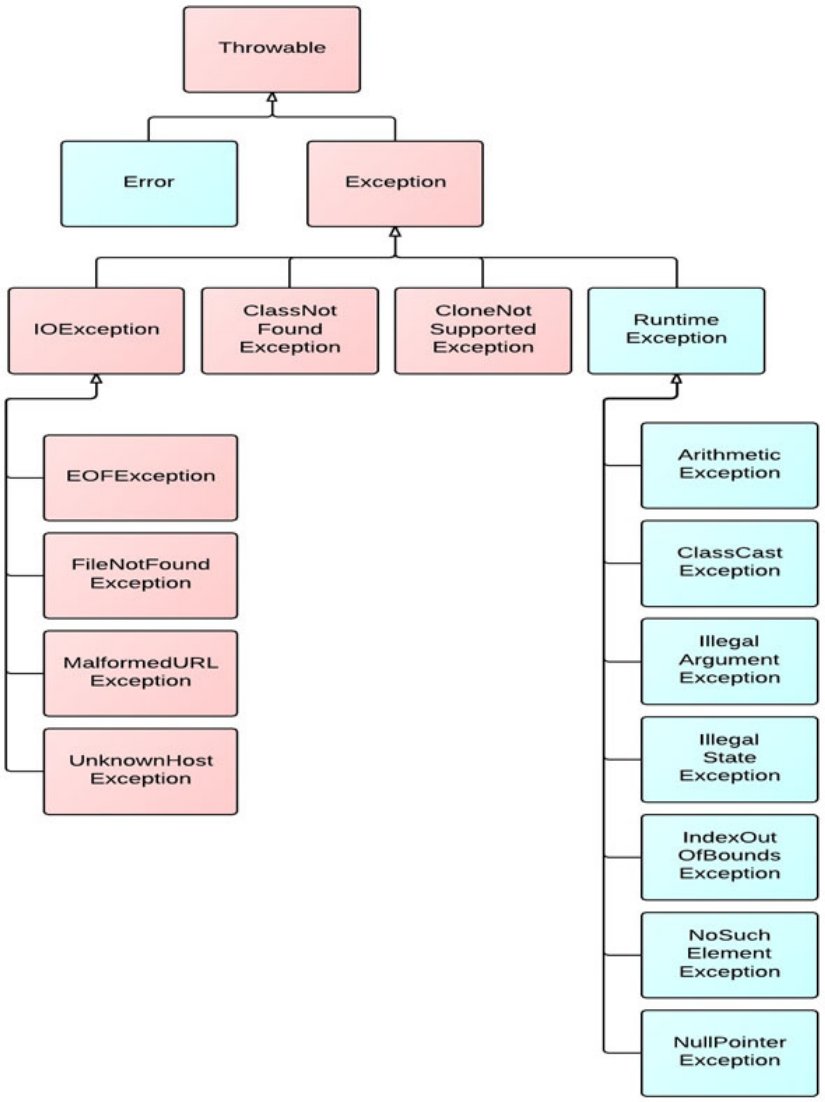
当程序中抛出一个异常后，程序从程序中导致异常的代码处跳出，java虚拟机检测寻找和try关键字匹配的处理该异常的catch块，如果找到，将控制权交到catch块中的代码，然后继续往下执行程序，try块中发生异常的代码不会被重新执行。如果没有找到处理该异常的catch块，在所有的finally块代码被执行和当前线程的所属的ThreadGroup的uncaughtException方法被调用后，遇到异常的当前线程被中止。

异常的结构

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）。

最新文章

- 关于域名的若干想法
- 公司的伙食
- 基于人工智能的问答类型APP介绍
- 来年寄语
- 经验决定行动
- 主机迁移公告
- UTF-16编码和Java，UTF-8编码...
- UTF-8编码
- Big Endian和Little Endian
- UTF-16编码



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。

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。

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。

在声明方法时候抛出异常

语法：throws（略）
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。

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。

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。

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

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。

丢失的异常

请看下面的代码：

```
public void method2()
{
try
{
    ……
    method1();  //method1进行了数据库操作
}
catch(SQLException e)
{
    ……
    throw new MyException("发生了数据库异常:"+e.getMessage);
}
}
public void method3()
{
    try
{
    method2();
}
catch(MyException e)
{
    e.printStackTrace();
    ……
}
}
```

上面method2的代码中，try块捕获method1抛出的数据库异常SQLException后，抛出了新的自定义异常MyException。这段代码是否并没有什么问题，但看一下控制台的输出：
MyException:发生了数据库异常：对象名称'MyTable' 无效。
at MyClass.method2(MyClass.java:232)
at MyClass.method3(MyClass.java:255)
原始异常SQLException的信息丢失了，这里只能看到method2里面定义的MyException的堆栈情况；而method1中发生的数据库异常的堆栈则看不到，如何排错呢，只有在method1的代码行中一行行去寻找数据库操作语句了。
JDK的开发者们也意识到了这个情况，在JDK1.4.1中，Throwable类增加了两个构造方法,public Throwable(Throwable cause)和public Throwable(String message,Throwable cause)，在构造函数中传入的原始异常堆栈信息将会在printStackTrace方法中打印出来。但对于还在使用JDK1.3的程序员，就只能自己实现打印原始异常堆栈信息的功能了。实现过程也很简单，只需要在自定义的异常类中增加一个原始异常字段，在构造函数中传入原始异常，然后重载printStackTrace方法，首先调用类中保存的原始异常的printStackTrace方法，然后再调用super.printStackTrace方法就可以打印出原始异常信息了。可以这样定义前面代码中出现的MyException类：

```java
import java.io.PrintStream;
import java.io.PrintWriter;
public class MyException extends Exception
{

    private static final long serialVersionUID = 1L;
    //原始异常
    private Throwable cause;
    //构造函数
    public MyException(Throwable cause)
    {
        this.cause = cause;
    }
    public MyException(String s,Throwable cause)
    {
        super(s);
        this.cause = cause;
    }
    //重载printStackTrace方法，打印出原始异常堆栈信息
    public void printStackTrace()
    {
        if (cause != null)
        {
            cause.printStackTrace();
        }
        super.printStackTrace();
    }

    public void printStackTrace(PrintStream s)
    {
        if (cause != null)
        {
            cause.printStackTrace(s);
        }
        super.printStackTrace(s);
    }

    public void printStackTrace(PrintWriter s)
    {
        if (cause != null)
        {
            cause.printStackTrace(s);
        }
        super.printStackTrace(s);
    }
}
```


上一篇：Java门派研究分析报告
下一篇：异常的深入研究与分析（2）

异常的深入研究与分析（1）：等您坐沙发呢！

发表评论

昵称 *

邮箱 *

网址

提交[Ctrl+Enter] 重写