

# Chp12 异常处理

## 本章导读

异常处理，处理的是程序的错误。对于我们来说，程序出错并不是最可怕的事情，最可怕的是程序出错为用户带来损失。例如，如果我们去 ATM 机上取钱，假设输入密码和金额之后，ATM 机没有吐出钱来，而是程序崩溃了。这个时候，我们关心的不是程序是否还能正常运行，这台机器有多大的几率能够修复。我们关心的是，我们的账户上的余额是否被修改了，关心的是程序崩溃了是否给我们带来了损失。

因此，异常处理不是为了让程序不出错，而是为了一旦程序出错，能够有一个相关的机制让程序执行一些代码来减少损失。这些代码是事先写好的，只有在错误发生的时候才会运行。就好像生活中的医院：开设医院并不能阻止人们生病，而是在人们生病之后，能有一个地方处理人的病情，通过各种手段来让人们恢复健康从而减少健康方面的损失。

## 1 异常的概念和分类

首先我们来介绍一下 Java 中所有错误的分类。在面向对象的概念中，一个错误也是一个对象，犯了一个错误，也就是创建了一个错误对象。在 Java 中，有一个 `java.lang.Throwable` 类，这个类是所有错误的父类。Java 中所有的错误类都是 `Throwable` 的子类。

`Throwable` 有两个子类，一个叫做 `Error`，一个叫做 `Exception`。其中，`Error` 指的是非常严重的错误。这种错误往往来源于 Java 底层，一旦发生这种错误，我们连减少损失的机会都没有。例如，虚拟机运行时崩溃，这种错误就是非常典型的 `Error`。因为虚拟机一旦崩溃，我们完全没有办法再执行任何代码，因此也没有任何机会来做一些减少损失的操作。这就好比当一个人如果生病了，可以去医院看病，从而减少损失。但是如果这个人死了，那无论做什么，都已经没有挽回损失的余地了。因此，对于这种严重的底层错误，我们的态度是：不做处理。并不是我们不想对这些错误做处理，而是我们根本没有机会对这种严重的底层错误进行处理。

相对于 `Error` 来说，`Exception` 就是指的，还不那么严重，有挽回余地的错误，这个单词被翻译成“异常”，在 Java 中的异常处理，指的就是 `Exception` 的处理。对于 `Exception` 而言，这个类有很多很多的子类，其中有一个类叫做 `RuntimeException`，这个类也有很多的子类。这样，所有 `Exception` 的子类就被 `RuntimeException` 分为两大部分：一种是 `Exception` 的子类，但不是 `RuntimeException` 的子类，被称为“已检查异常”；另一种是 `Exception` 子类但不是 `RuntimeException` 的子类，被称为“未检查异常”。

如果拿到一个异常类，如何分辨其是已检查异常还是未检查异常呢？只要看这个类的继承体系：如果这个类的直接或者间接父类中有 `RuntimeException`，则这个类是一个未检查异常；如果没有的话，则这个类是一个已检查异常。

那这两种异常有什么区别呢？所谓的“未检查异常”，指的是在写程序过程中可以避免的异常。可以这么来理解，之所以发生“未检查异常”，原因就是程序员写完程序以后没有好好检查；如果程序员能够好好检查自己的代码，则这些异常都可以避免发生。下面我们就为大家介绍一些常见的未检查异常。

```
import java.util.Scanner;
```

```

public class TestRuntimeException {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        int b = sc.nextInt();
        System.out.println(a/b);
    }
}

```

这段代码读入两个整数，输出他们的商。乍看之下，这段代码没什么问题，但是这段代码有一个隐患：当读入的整数 **b** 为 0 时，这段代码会输出什么？

当 **b** 为 0 时，这段代码就会产生一个异常，异常信息如下：

```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at TestRuntimeException.main(TestRuntimeException.java:7)

```

这段代码里面有几个重要的信息。一个，是产生的异常类的名字，当除数为 0 时，会产生一个 `ArithmeticException` 的异常，产生这个异常的原因则写在类名后面：“/ by zero”。另外，显示产生这个异常的原因代码位置是 `TestRuntimeException.java` 文件的第 7 行。

在上面的代码中，`ArithmeticException` 异常是一个非常典型的未检查异常，完全可以通过编程时检查代码，通过各种手段加以避免。例如，我们可以在输出语句之前加一句判断：

```

if (b != 0) System.out.println(a/b);

```

这样，就能避免我们的输出语句产生异常。

因此，产生的 `ArithmeticException` 就是一个未检查异常，通过分析这个类的父类，我们同样可以得出这个结论。通过查阅 API 文档，我们发现这个类的父类是：`RuntimeException`。

`java.lang`

## 类 `ArithmeticException`

[`java.lang.Object`](#)

└ [`java.lang.Throwable`](#)

└ [`java.lang.Exception`](#)

└ [`java.lang.RuntimeException`](#)

└ `java.lang.ArithmeticException`

除了 `ArithmeticException` 之外，还有一些其他异常的例子，例如下面的代码：

```

int[] n = new int[3];
n[3] = 10;

```

这段代码会产生一个 `ArrayIndexOutOfBoundsException`，也就是典型的数组下标越界异常。这个异常同样是一个未检查异常，可以通过细心的编写数组代码来避免出现这样的异常。

再例如，下面的代码：

```

class Animal{}
class Dog extends Animal{}
class Cat extends Animal{}
public class TestAnimal{
    public static void main(String args[]){
        Animal a = new Dog();
        Cat c = (Cat) a
    }
}

```

```
}
```

上面的代码会在运行时抛出一个 `ClassCastException`。这个异常也是一个未检查异常，原因是类型转换失败。我们可以通过在强制类型转换之前调用 `instanceof` 判断，从而来避免产生这样的类型转换异常。

最后，还有一个常见的未检查异常：`NullPointerException`。这个异常在我们对 `null` 引用使用属性或者调用方法时产生。这个异常也能够通过仔细编写代码来避免。

对于未检查异常，我们都可以通过仔细检查，在程序中加入适当的 `if` 语句，从而避免这种异常。因此，对于这个异常来说，系统并不要求我们必须处理它们。再好的异常处理，也不如不让异常发生，所谓“防范胜于救灾”，就是这个道理。

未检查异常的对应的是已检查异常。所谓的“已检查异常”，指的是一类无法避免的异常。可以这么来记忆：“已检查异常”是程序员已经仔细把代码检查过了之后，依然会发生的异常。这种异常因为程序员无法避免，因此必须要处理。如果一个程序可能发生已检查异常，而程序中缺少处理异常的代码，那么编译时我们会得到编译错误。

例如，假设程序员在 `Java` 中写了一段用来连接网络服务器的程序。对于程序员来说，无论在编程方面多么小心，都可能面临这样的异常：网络不通。一个 `java` 程序员有再大的本事，也无法保证运行程序时网络一定是通的。因此，在这种情况下，程序员必须要处理这个异常，也就是说，程序员必须要写下一个预案，向 `JVM` 说明：如果网络不通，程序应当如何处理。当网络不通时，系统会抛出 `java.net.SocketException`。通过查阅 `API` 文档可知，这个类是 `Exception` 的子类，但不是 `RuntimeException` 的子类，因此是已检查异常。

`java.net`

## 类 `SocketException`

`java.lang.Object`

└ `java.lang.Throwable`

└ `java.lang.Exception`

└ `java.io.IOException`

└ `java.net.SocketException`

举个生活中的例子。地震，就是一个典型的已检查异常。地震这种异常，任何人都无法预防。那各地的政府会做什么呢？一方面是根据当地的实际情况，用来设定当地建筑物的抗震标准，这样一旦发生地震时，建筑物能够在一定程度上抵抗这个异常，从而让人有逃生的机会，减少损失；一方面是由地震监测部门，监测并预报地震的发生，另一方面，是一旦发生地震之后，进行救灾和援助的组织工作。以上工作皆可认为是政府部门对地震这种已检查异常的处理。

而生活中未检查异常的例子，比较典型的是吸烟引起的火灾。根据资料统计，人在劳累时在床上或者沙发上抽烟，因为烟头点燃纺织物而造成的火灾，是产生火灾的重要原因之一。由于这种原因而产生的异常（火灾），就是一种完全能够避免的异常。这种异常就属于未检查异常。也许我们的家中很少会事先配备灭火器，来及时扑灭抽烟引起的大火。取而代之的，是加强防范措施，来避免这种火灾的发生。

对于已检查异常来说，由于无法避免，因此应该做好充分的预案，因此已检查异常必须要处理；对于未检查异常来说，因为这种异常可以避免，因此可处理可不处理，在实际编程过程中，未检查异常应当以避免为主。

下面是对 `Java` 中异常分类的总结：

**`Throwable` 类：**所有错误的父类

|-- Error: 严重的底层错误, 无法处理  
|-- Exception: 异常, 异常处理的主要对象  
    |-- RuntimeException 的子类: 未检查异常, 可以避免, 可处理可不处理  
    |-- 非 RuntimeException 的子类: 已检查异常, 无法避免, 必须处理

## 2 异常对象的产生和传递

看下面的代码:

```
import java.util.Scanner;
public class TestException{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("main 1");
        ma(i);
        System.out.println("main 2");
    }
    static void ma(int i) {
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }
    static void mb(int i){
        System.out.println("mb 1");
        mc(i);
        System.out.println("mb 2");
    }
    static void mc(int i){
        System.out.println("mc 1");
        if (i==0) throw new NullPointerException();
        System.out.println("mc 2");
    }
}
```

在这段代码中, 主方法调用 `ma`, `ma` 调用 `mb`, `mb` 调用 `mc`。注意 `mc` 方法, 当 `i==0` 时这个语句的语法:

```
throw new NullPointerException();
```

这是一个 `throw` 语句, `throw` 语句表明要抛出一个错误。要注意的是 `throw` 语句的语法:

`throw + Throwable 对象`

也就是说, `throw` 语句后面跟的是一个对象, 这个对象代表了发生的错误。例如, 在 `mc` 方法中, 我们创建了一个 `NullPointerException` 对象, 并把这个对象抛出。`throw` 语句的作用类似于 `return` 语句, 表示将一个异常对象作为方法的返回值返回。

需要注意的是, 如果我们没有使用 `throw`, 而仅仅是创建了一个对象, 就只是在 JVM 中分配了一块内存空间而已, 和创建一个普通对象一样。而只有使用了 `throw`, 才能表明真正抛出异常对象。

由于 `NullPointerException` 是一个未检查异常，因此可处理可不处理。在这个程序中，我们没有对 `NullPointerException` 进行了处理。

当程序正常时（既在命令行上读入的整数不为 0 时），输出如下：

```
main 1
ma 1
mb 1
mc 1
mc 2
mb 2
ma 2
main 2
```

上面的代码很容易理解，主方法输出 `main1`，然后调用 `ma` 方法；`ma` 输出 `ma1`，然后调用 `mb` 方法；`mb` 方法输出 `mb1`，然后调用 `mc` 方法；`mc` 方法输出 `mc1` 和 `mc2`；返回 `mb`；`mb` 方法输出 `mb2`，返回 `ma`；`ma` 方法输出 `ma2`，返回主方法；最后主方法输出 `main2`。

当我们在命令行上输入 0 时，运行结果如下：

```
main 1
ma 1
mb 1
mc 1
Exception in thread "main" java.lang.NullPointerException
    at TestException.mc(TestException.java:23)
    at TestException.mb(TestException.java:18)
    at TestException.ma(TestException.java:13)
    at TestException.main(TestException.java:8)
```

上面的运行结果是怎么来的呢？首先依然是 `main → ma → mb → mc` 这个调用过程。然后，在 `mc` 方法中产生了一个异常对象，并且向上抛出。`mc` 方法产生异常之后，后面的正常代码就不执行了，程序从 `throw` 语句处把异常向上抛出。`mc` 方法抛出异常之后，这个异常对象就到了 `mc` 的调用者：`mb` 方法中，即在 `mb` 方法的

```
mc(i);
```

这个语句处产生一个异常对象。对于 `mb` 方法而言，也就是调用 `mc` 方法时产生了一个异常。由于 `mb` 方法中没有任何处理异常的代码，因此 `mb` 方法后面的代码也停止执行，而直接把这个异常抛出，向上抛给了 `ma` 方法。`ma` 方法获得了这个异常之后，也没有处理，抛给了 `main` 方法，`main` 方法也没有处理，于是这个异常就抛到了 JVM 中。JVM 获得这个异常之后，会打印这个异常的信息，并且让程序终止。

也就是说，方法调用时，是 `main → ma → mb → mc` 的链状结构，这叫做方法调用链。而当产生异常的时候，函数的代码会在产生异常的地方终止，然后把异常对象返回给函数的调用者。对于我们这个例子来说，产生异常之后，异常传递的方式是 `main ← ma ← mb ← mc`。这个结论就是说：当函数产生并抛出一个异常时，异常会沿着方法调用链反向传递。

### 3 异常对象的处理

在上一小节代码的基础上，我们修改一下，让 `mc` 方法有更多的选择：

```

static void mc(int i){
    System.out.println("mc 1");
    if (i==0) throw new NullPointerException();
    if (i==1) throw new java.io.FileNotFoundException();
    if (i==2) throw new java.io.EOFException();
    if (i==3) throw new java.sql.SQLException();
    System.out.println("mc 2");
}

```

首先我们来看一下这几个异常类。我们可以查看 [JDK 文档](#)，在文档中显示的 `FileNotFoundException` 和 `EOFException` 这两个类的继承关系如下：

`FileNotFoundException`:

```

java.io
Class FileNotFoundException

java.lang.Object
└─ java.lang.Throwable
    └─ java.lang.Exception
        └─ java.io.IOException
            └─ java.io.FileNotFoundException

```

`EOFException`:

```

java.io
Class EOFException

java.lang.Object
└─ java.lang.Throwable
    └─ java.lang.Exception
        └─ java.io.IOException
            └─ java.io.EOFException

```

可以看出，这两个异常都是 `IOException` 的子类，并且，由于继承树中不存在 `RuntimeException` 这个类，因此这两个类都是已检查异常。

而下面是 `java.sql.SQLException` 的继承关系：

`SQLException`:

```

java.sql
Class SQLException

java.lang.Object
└─ java.lang.Throwable
    └─ java.lang.Exception
        └─ java.sql.SQLException

```

从它的继承关系可以看出，这个类同样是一个已检查异常。

在 `mc` 方法抛出 `NullPointerException` 的时候，由于这个异常是一个未检查异常，可处理可不处理，因此我们的程序中没有任何处理 `NullPointerException` 的代码。而对于后面的代码，由于抛出的三个异常都是已检查异常，因此必须要处理。如果不处理的话，就会产生编译时错误，信息如下：

```

D:\book>javac TestException.java
TestException.java:23: 未报告的异常 java.io.FileNotFoundException; 必须对其进行
捕捉或声明以便抛出
        if (i==1) throw new java.io.FileNotFoundException();
        ^
TestException.java:24: 未报告的异常 java.io.EOFException; 必须对其进行捕捉或声明
以便抛出
        if (i==2) throw new java.io.EOFException();
        ^
TestException.java:25: 未报告的异常 java.sql.SQLException; 必须对其进行捕捉或声
明以便抛出
        if (i==3) throw new java.sql.SQLException();
        ^
3 错误

```

接下来我们要介绍的是，如何处理异常。

### 3.1 throws 声明抛出异常

让我们想想这样的场景，JVM 和 main、ma、mb、mc 这四个方法坐在一起开会。首先，JVM 问 mc：

JVM: 你可能会抛出 NullPointerException? 你打算怎么处理?

mc: ..., 我没有任何要说的，要注意，它是未检查异常，我有不处理它的权利!

JVM: OK, 那好，那你还可能抛出 FileNotFoundException, EOFException, SQLException, 关于这些异常，你打算怎么办?

mc: 我不知道，让我想一下……

JVM: 你不知道??!! 这些都是已检查异常，如果你不处理的话，我想编译器都不会让你过关的!

为了解决这个尴尬的局面，mc 方法决定回答 JVM。

mc: 你说的那些异常，我作为一个小小的被调用的方法，实在是处理不了。如果发生这些问题的话，我会向上级领导反映……

mc 这种处理异常的方式，体现在代码上，就是下面的 throws 语句：

```

import java.io.*;
import java.sql.*;
...
static void mc(int i)
    throws FileNotFoundException, EOFException, SQLException{
    System.out.println("mc 1");
    if (i==0) throw new NullPointerException();
    if (i==1) throw new FileNotFoundException();
    if (i==2) throw new EOFException();
    if (i==3) throw new SQLException();
    System.out.println("mc 2");
}

```

在方法的参数表最后，写上“throws”加上一系列异常的名字，表示声明抛出。如果要抛出多个异常的话，多个异常之间用逗号隔开。

注意，一定要区分“throws”和“throw”这两个关键字。throw 是一个动作，如果执行时遇到 throw，这表明在这个语句的地方真的会抛出一个异常。例如，在 mc 方法内部的

**throw**，每个 **throw** 都会真的抛出一个异常。而 **throws** 表示的是一个声明，这个声明表示这个方法有可能抛出异常。例如，**mc** 方法声明 **throws FileNotFoundException, EOFException, SQLException**，这表明调用 **mc** 方法有可能会抛出这些异常。也可以这么理解：如果你要调用 **mc** 方法的话，**mc** 方法告诉其他函数：调用我可以，但是我可能会出“**FileNotFoundException, EOFException, SQLException**”这些错，如果出了这些错，我不会管，而由调用我的函数负责。

简单来说：**throw** 是一个动作，表示抛出；而 **throws** 是一个声明，表示本方法一旦发生这些异常，本方法不作处理，异常由调用这个方法的方法来处理。

OK，回到 JVM 和四个方法的会议。**mc** 声明了这些异常它不管，接下来 JVM 就开始问另一个方法：**mb**。如果 **mb** 不处理 **mc** 声明抛出的异常，则会产生编译时的错误。

JVM：**mb**，**mc** 说他要 **throws** 那三个异常。他说调用他的函数要对那三个异常负责。所以，现在我问你，你对这三个异常打算怎么处理？

**mb**：首先我想说，**FileNotFoundException** 和 **EOFException**，这都属于 **IOException**。只要是 **IOException**，我就不管。

JVM：……好吧，那 **SQLException** 呢？

**mb**：我同样不想管。

这段对话翻译成代码如下：

```
static void mb(int i) throws IOException, SQLException{
    System.out.println("mb 1");
    mc(i);
    System.out.println("mb 2");
}
```

**mb** 方法声明抛出 **IOException** 和 **SQLException**。需要注意的是，由于 **FileNotFoundException** 和 **EOFException** 与 **IOException** 有父子类的关系，因此声名抛出 **IOException**，就包含了声明抛出所有 **IOException** 的子类。这是把多态用在声明抛出异常上面。

到了这一步，**mb** 同样把异常往上抛：**mb** 声明抛出 **IOException** 和 **SQLException**。作为调用 **mb** 的方法，**ma** 难辞其咎。于是 JVM 与 **ma** 有了下面的对话：

JVM：**ma**，**ma**！醒醒！我们正在开会呢！

**ma**：啊？哦……你们刚刚说了什么？

JVM：**mb** 方法说他不处理 **IOException** 和 **SQLException**，你作为他的调用者，你应该……

**ma**：啥都别说了，有问题，找我的调用者！我接着睡觉去了……

JVM：……

把这段场景翻译成代码如下：

```
static void ma(int i) throws Exception{
    System.out.println("ma 1");
    mb(i);
    System.out.println("ma 2");
}
```



也就是说，只要是异常，**ma** 方法都会往上抛出！

终于到了最后，JVM 面对着 **main** 方法……

JVM: **main** 方法，我们相处这么多年了，我启动以后第一个寻找的就是你，你不会让我失望吧……

**main**: 唉，大哥不好当啊，小弟如果出了问题，我也很难办……

JVM: 你的意思是？

**main**: 如果出了问题，我也拦不住，所以……

JVM: 难道……

是的，在主方法后面，同样可以加上 **throws** 语句。换句话说，主方法同样可以抛出异常！

JVM: 我没了问了，所有的方法都选择了逃避。那么一旦 **mc** 方法抛出了异常，**mc** 推给 **mb**，**mb** 推给 **ma**，**ma** 推给 **main**，**main** 方法又推给了我，那么我，只好选择停止这个程序的运行了。

完整的代码如下：

```
import java.util.*;
import java.io.*;
import java.sql.*;
public class TestException{
    public static void main(String[] args) throws Exception{
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("main 1");
        ma(i);
        System.out.println("main 2");
    }
    static void ma(int i) throws Exception{
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }
    static void mb(int i) throws IOException, SQLException{
        System.out.println("mb 1");
        mc(i);
        System.out.println("mb 2");
    }
    static void mc(int i) throws FileNotFoundException,
EOFException, SQLException{
        System.out.println("mc 1");
        if (i==0) throw new NullPointerException();
        if (i==1) throw new java.io.FileNotFoundException();
        if (i==2) throw new java.io.EOFException();
        if (i==3) throw new java.sql.SQLException();
        System.out.println("mc 2");
    }
}
```

```
    }  
}
```

当我们在命令行上输入 1 时，运行结果如下：

```
main 1
```

```
ma 1
```

```
mb 1
```

```
mc 1
```

```
Exception in thread "main" java.io.FileNotFoundException  
    at TestException.mc(TestException.java:24)  
    at TestException.mb(TestException.java:18)  
    at TestException.ma(TestException.java:13)  
    at TestException.main(TestException.java:8)
```

与抛出 `NullPointerException` 的情况类似，异常对象由 `mc` 方法抛出后，被逐层传递至 JVM，最终导致程序的中止运行。

通过 `throws` 关键字，函数可以声明抛出异常。虽然这样处理异常比较“消极”，但是同样是一种处理方式。

虽然我们在任何一个函数后面都没有声明抛出 `NullPointerException`，但是由于这个异常是未检查异常，因此任何一个函数都可以向外抛出这个异常。换句话说，我们可以认为每个函数都默认 `throws RuntimeException`。

### 3.2 try-catch 捕获异常

除了通过 `throws` 进行消极处理之外，我们同样可以进行一些积极处理。我们可以采用 `try-catch` 的语法来捕获可能抛出的异常。例如，现在 `ma` 方法不向抛出异常，而决定自己来捕获和处理 `mb` 有可能发生的异常。`try-catch` 的语法结构如下：

```
try{  
    可能发生异常的代码;  
}catch(捕获的异常类型 1 e){  
    异常处理 1...  
}catch(捕获的异常类型 2 e){  
    异常处理 2...  
} catch(捕获的异常类型 n e){  
    异常处理 n...  
}
```

如果在 `try` 语句块中没有出现异常，那么任何一个 `catch` 语句块都不会得到执行。但一旦 `try` 块中出现了异常，程序的流程会马上跳出 `try` 块，根据异常类型的不同，进入某一个 `catch` 语句块。随后，这个异常被宣告处理完毕，程序将继续向下正常运行。

例如，我们可以把 `ma` 方法按照上面的语法结构改成如下形式：

```
static void ma(int i) {  
    try{  
        System.out.println("ma 1");  
        mb(i);  
        System.out.println("ma 2");  
    }
```

```

    }catch(IOException ioe){
        System.out.println("IOException");
    }catch(SQLException sqle){
        System.out.println("SQLException");
    }catch(Exception e){
        System.out.println("Exception");
    }
}

```

注意上面的代码，我们把 `ma` 方法签名后面的 `throws Exception` 声明给去掉了，并把 `ma` 方法中原有的三行代码都放入了 `try` 块中。由于这其中 `mb` 方法有可能抛出异常，我们这么做也就是把可能抛出异常的代码放入了 `try` 块中。

在 `try` 块之后，是三个 `catch` 子句。这三个 `catch` 语句分别捕获不同的异常类型，三个分别为 `IOException`、`SQLException` 和 `Exception` 类型。`catch` 语句表示，当 `try` 块中的代码抛出异常的时候，会根据抛出异常类型的不同，而进行对应的不同的处理。例如，当 `mb` 方法抛出 `SQLException` 时，程序会进入 `SQLException` 的 `catch` 语句块。

当我们在 `ma` 方法中处理完异常之后，就可以把 `main` 方法中的 `throws Exception` 语句给去掉。完整代码如下：

```

import java.util.*;
import java.io.*;
import java.sql.*;
public class TestException{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int i = sc.nextInt();
        System.out.println("main 1");
        ma(i);
        System.out.println("main 2");
    }
    static void ma(int i) {
        try{
            System.out.println("ma 1");
            mb(i);
            System.out.println("ma 2");
        }catch(IOException ioe){
            System.out.println("IOException");
        }catch(SQLException sqle){
            System.out.println("SQLException");
        }catch(Exception e){
            System.out.println("Exception");
        }
    }

    static void mb(int i) throws IOException, SQLException{
        System.out.println("mb 1");
    }
}

```

```

        mc(i);
        System.out.println("mb 2");
    }
    static void mc(int i) throws FileNotFoundException,
EOFException, SQLException{
        System.out.println("mc 1");
        if (i==0) throw new NullPointerException();
        if (i==1) throw new java.io.FileNotFoundException();
        if (i==2) throw new java.io.EOFException();
        if (i==3) throw new java.sql.SQLException();
        System.out.println("mc 2");
    }
}

```

下面我们分析一下，当正常情况以及异常情况时程序执行的流程。当输入  $i$  不等于  $0 \sim 4$  之间的值时，`mc` 方法不抛出异常，也就意味着 `ma` 中的 `try` 块中没有发生异常。在这种情况下，`ma` 方法执行时，会顺利执行完 `try` 块中的所有语句，然后返回 `main` 方法中。

那如果发生异常情况呢？假设给定  $i == 3$ ，根据 `mc` 方法中的代码，`mc` 会抛出一个 `SQLException`。`mb` 调用 `mc`，由于 `mb` 中没有处理这个异常的代码，因此会把这个异常抛给 `ma` 方法。也就是说，在 `ma` 调用 `mb` 方法的这个 `try` 块中，产生了一个异常。当产生异常之后，程序会马上跳出 `try` 块，因此在 `try` 块中最后一句输出“`ma2`”会被跳过。然后，由于 `try` 块中产生了一个 `SQLException`，得到这个异常之后会依次跟后面的 `catch` 语句中的异常类型进行比较，因此会进入捕获 `SQLException` 的语句块中，输出“`SQLException`”。当 `catch` 语句执行完毕之后，`ma` 方法正常返回 `main` 方法中。因此输出结果如下：

```

main 1
ma 1
mb 1
mc 1
SQLException
main 2

```

假设给定  $i == 2$ 。此时程序中产生的异常是 `EOFException`。当这个异常从 `mc` 传递到 `ma` 时，相当于在 `try` 块中产生了一个 `EOFException`。这个异常被抛出之后，会依次对 `catch` 语句进行匹配。由于 `EOFException` 是 `IOException` 的子类，因此在遇到第一个 `catch` 语句：

```
catch(IOException ioe) {...}
```

时，`EOFException` 会作为 `IOException` 的子类被捕获。这是非常典型的多态用在异常的捕获上。类似的，当  $i == 1$  时抛出的 `FileNotFoundException` 也会被捕获 `IOException` 的 `catch` 语句捕获。

当  $i == 0$  时，程序中抛出的是 `NullPointerException`。由于 `NullPointerException` 既不是 `IOException`，也不是 `SQLException`，因此这个异常无法被前两个 `catch` 语句所捕获。但是由于 `NullPointerException` 是 `Exception` 的子类，因此它能够被第三个 `catch` 语句捕获。

上面这个例子说明了 1、未检查异常同样可以被捕获、被处理。2、如果在一系列 `catch` 语句中存在一个捕获 `Exception` 的语句，就意味着任何类型的异常都能够被这个 `catch` 语句

所捕获。换句话说，由于 `Exception` 是所有异常类的父类，根据多态，捕获 `Exception` 类也就是捕获其任何一种子类异常。

由于多态可以用在异常的捕获上面，因此上面的代码还有一个细节值得注意。如果修改上述 `ma` 方法中的代码：

```
static void ma(int i) {
    try{
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }
    //! 下面的代码编译出错!
    catch(Exception e){
        System.out.println("Exception");
    }catch(IOException ioe){
        System.out.println("IOException");
    }catch(SQLException sqle){
        System.out.println("SQLException");
    }
}
```

注意，上述代码就是把捕获 `Exception` 类型的 `catch` 语句放到了前面。如果代码写成这样的话，编译就会失败，原因在于：由于任何一种异常都能被当做 `Exception` 来捕获，因此 `try` 块中产生的任何一种异常都会被第一个 `catch` 语句捕获。这也就意味着，即使产生了 `IOException` 或者 `SQLException`，后面的两个 `catch` 语句也不会被执行。由于这两个 `catch` 语句永远无法被执行到，因此会产生一个编译时的错误。

为了避免上述的问题，往往我们写捕获异常的代码时，会遵循这样一个原则：捕获子类异常的 `catch` 语句写在前面，捕获父类异常的 `catch` 语句写在后面。

此外，还要注意的一点是，由于 `try` 块中的代码表示的是可能会出错的代码，因此对于 `java` 来说，`try` 块中的代码是有可能不执行的代码。于是，就可能会衍生出一些特别的语法现象。例如，定义下面的函数：

```
public static int m1(){
    try{
        return 100;
    }catch(Exception e){}
}
```

上面的函数无法编译通过，原因在于，上面的函数要求必须返回一个 `int` 类型的值，但是由于 `return` 语句是在 `try` 块中，因此 `return` 语句有可能不执行，这样就会使得这个函数不返回任何值，从而产生编译错误。

再看下面这段代码的例子：

```
public static void m2(){
    int n;
    try{
```

```

        n = 10;
    } catch (Exception e) {}
    System.out.println(n);
}

```

上面的代码同样编译出错。在 `m2` 方法中定义了一个局部变量 `n`，这个局部变量在 `try` 块中被赋予了初始值。但是由于 `java` 认为 `try` 块中的语句不一定会被执行，所以在 `try` 块外输出 `n` 的值时，会提示“可能尚未初始化变量 `n`”，因此编译出错。

### 3.3 finally

`finally` 语句是 `try-catch` 语句的一个补充。在 `try-catch` 语句块之后，可以加上一个 `finally` 语句块，这个语句块中的代码，无论程序执行时是否发生异常，最终都会被执行。例如，我们可以为 `ma` 方法的 `try-catch` 语句增加一个 `finally` 代码块：

```

static void ma(int i) {
    try{
        System.out.println("ma 1");
        mb(i);
        System.out.println("ma 2");
    }catch(IOException ioe){
        System.out.println("IOException");
    }catch(SQLException sqle){
        System.out.println("SQLException");
    }catch(Exception e){
        System.out.println("Exception");
    }finally{
        System.out.println("in finally of ma");
    }
}

```

如上面代码所示，我们为 `try-catch` 语句增加了一个 `finally` 语句块。当程序正常运行时（也就是调用 `mb` 方法时没有产生异常），此时 `try` 块正常执行结束。在程序跳出 `try-catch-finally` 而返回主方法之前，`finally` 语句会得到执行。体现在输出结果中，就是在 `ma2` 和 `main2` 之间输出“in finally of ma”。输出结果如下：

```

main 1
ma 1
mb 1
mc 1
mc 2
mb 2
ma 2
in finally of ma //当程序正常时会运行 finally
main 2

```

当产生异常时（例如 `i == 1`），此时由于 `try` 块中抛出异常，因此跳出 `try` 块并且执行相应的 `catch` 语句块。当 `catch` 语句块执行之后，同样会执行 `finally` 语句块中的内容之后，然后才会返回主方法。当 `i == 1` 时，输出结果如下：

```
main 1
ma 1
mb 1
mc 1
IOException //进入 catch IOException 的 catch 语句
in finally of ma //产生异常也要运行 finally
main 2
```

因此，**finally** 语句块中的代码，意味着无论程序是否发生异常，都一定要执行。关于这一点，我们还可以看下面的代码

```
static int ma2(int i) {
    try{
        mb(i);
        return 100;
    }catch(Exception e){
        System.out.println("Exception");
        return 200;
    }finally{
        System.out.println("in finally of ma");
        return 300;
    }
}
```

上述代码，在 **try** 块中有一个 **return 100**，在 **catch** 块中有一个 **return 200**，在 **finally** 中有一个 **return 300**。如果在其他方法中调用这个 **ma2** 方法，返回值是什么呢？

如果 **mb** 方法不抛出异常，程序正常执行的话，这样执行到 **try** 块最后的 **return 100** 时，程序不会在 **try** 块内返回，而会先执行 **finally** 语句块中的内容，再执行 **return 100**。而由于 **finally** 语句块中存在一个 **return 300**，因此 **ma2** 方法的返回值为 **300**。

如果 **mb** 方法抛出异常，则程序会跳出 **try** 块，进入 **catch** 子句。在 **catch** 子句中有一个 **return** 语句。这个 **return** 语句在执行之前，同样要先执行 **finally** 语句块中的内容，执行完之后才能返回。由于 **finally** 语句块中同样具有 **return** 语句，因此在执行 **finally** 语句过程中，程序就会返回，返回值为 **300**。

也就是说，如果 **finally** 语句块中存在 **return** 语句，最终函数一定会从 **finally** 中返回，而不是 **try** 块或者 **catch** 语句中返回。这也进一步说明：**finally** 块中的代码一定会执行。

正因为 **finally** 语句块中的代码一定会被执行，因此 **finally** 语句块中我们往往会放上一些释放资源的代码。例如，有下面一些步骤：

1. 申请数据库连接资源
2. 通过数据库的验证，得到数据库连接
3. 使用数据连接，完成数据库的操作
4. 释放数据库连接

在上面的四个步骤中，释放数据库连接就是一个非常典型的释放资源的过程。在申请到了数据库资源之后，无论是第二步进行数据库验证，还是第三步完成数据库操作，都有可能发生异常。但是无论发生异常与否，数据库连接的释放都应该执行，因此这一步也就是所谓的一定要执行的一步。为了保证数据库连接的释放一定被执行，因此这一步应当放在 **finally**

语句块中。

## 4 异常与面向对象

上面介绍了一些异常的基本操作和语法，接下来这部分内容，将结合面向对象与异常处理，对异常进行进一步的介绍。

### 4.1 异常与方法覆盖

我们首先介绍一下异常与方法覆盖相关的内容。方法覆盖对于方法声明抛出的异常也有要求，具体的来说，要求：子类的覆盖方法不能比父类的被覆盖方法抛出更多的异常。

如何来理解所谓“不能抛出更多的异常”呢？考虑下面的代码：

```
class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m() throws IOException{}
}
```

上面两个类中，**Sub** 类继承自 **Super** 类，并且覆盖了 **Super** 类的 **m** 方法。在 **Super** 类中的 **m** 方法抛出 **IOException**，而子类的 **m** 方法也同样抛出 **IOException**。这样，子类的覆盖方法与父类的被覆盖方法抛出的异常相同，这样能够编译通过。

同样的，如果父类方法中 **throws IOException**，而子类方法中不抛出任何异常，这样代码同样可以编译通过，这样也不算抛出更多的异常。示例代码如下：

```
class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m(){}
}
```

但是，如果父类方法中没有抛出 **IOException**，而子类方法抛出这个异常，则会产生问题。例如：

//此处代码编译出错！

```
class Super{
    public void m(){}
}

class Sub extends Super{
    public void m() throws IOException{}
}
```

上述代码就是子类比父类抛出了更多的异常。

此外，异常的抛出还有多态的问题。例如下面的代码：



```

class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m() throws FileNotFoundException, EOFException{}
}

```

在上面的代码中，父类只抛出一个异常，子类抛出了两个异常，但是由于子类抛出的异常 `FileNotFoundException` 和 `EOFException`，是父类抛出的异常的子类，因此并不能算子类抛出了更多的异常。与之对应的是下面这个例子：

//此处代码编译出错！

```

class Super{
    public void m() throws IOException{}
}

class Sub extends Super{
    public void m() throws SQLException{}
}

```

这个例子同样编译出错。因为父类抛出的是 `IOException`，而子类抛出的 `SQLException`，这两个异常类之间完全没有任何的父子类之间的关系。

换句话说，子类不能比父类抛出更多的异常，我们可以这么来理解：子类要么抛出跟父类相同的异常，要么不抛出异常，要么抛出的异常是父类抛出异常的子类。只有这样三种选择。

## 4.2 异常类介绍

`Exception` 类作为 `java` 中异常处理的核心，下面我们就研究一下这个类的相关方法和属性。

### 4.2.1 message 属性

在 `Exception` 类中，有一个 `getMessage` 方法。该方法签名如下：

```
public String getMessage()
```

这个方法是在 `Throwable` 中定义的，因此能够被 `Exception` 子类继承，并且也就意味着所有的异常类都包含这个方法。这个方法返回一个字符串，JDK 的文档中，称这个方法返回的是“详细消息”。那么什么是这个详细消息呢？

从 `getMessage` 这个方法的签名来看，非常类似 `getXXX` 方法，我们有理由相信，在 `Throwable` 这个类中包含一个私有的 `message` 属性，而这个 `getMessage()` 方法就是用来获得那个 `message` 属性的。

虽然在 `java` 中提供了 `getMessage` 方法用来获得这个属性，但是却没有提供 `setMessage` 方法来设置这个属性的值。那么如何来设置属性值呢？这就只能在创建异常的时候，调用异常的构造方法来完成。在 `Exception` 类中包含这样一个构造方法：

```
public Exception(String message)
```

这个构造方法接受一个字符串参数，从 JDK 文档的形参名我们就可以猜测出，这个构造方法的参数能够用来设置 `Throwable` 中的 `message` 属性。也就是说，如果我们创建异常时给定一个字符串参数，则在后来捕获异常之后调用 `getMessage` 方法时就能获得这个字符串的

值。例如下面的例子：

```
import java.util.*;
import java.io.*;
import java.sql.*;
public class TestExceptionArgs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(Exception e){
            System.out.println(e.getMessage());
        }
    }

    public static void mb(int n) throws Exception{
        if (n == 0) throw new SQLException("n==0");
        if (n == 1) throw new EOFException("n==1");
        if (n == 2) throw new FileNotFoundException("n==2");
    }
}
```

上面的代码就是非常典型的使用 **message** 属性的例子：在创建异常时，给定一个字符串参数，而当捕获异常时，利用 **getMessage** 方法获得当时传递的参数。

当输入 **n** 为 1 时，输出结果为：

**n==1**

也就是创建异常时，给定的 **message** 参数。

**message** 属性往往被用来对异常进行一些解释和说明，使得程序员在得到异常的同时，能够得到关于该异常的更多信息，方便程序员调试。就像我们到银行取款，如果出现了异常，我们总是希望更多的了解这个异常的信息，究竟为什么取款失败，是因为密码错误，还是因为余额不足呢？

### 4.2.2 printStackTrace

除了 **getMessage** 方法之外，在异常类中还有一个常用方法：**printStackTrace**。这个方法签名如下：

```
public void printStackTrace()
```

这个方法同样是 **Exception** 类从 **Throwable** 继承来的方法，同样也是所有异常类都具有的方法。这个方法的作用是：在标准错误输出上打印出产生异常时的方法调用栈的信息。怎么来理解呢？所谓的标准错误，在默认情况下往往指的就是程序员的屏幕。而所谓的方法调用栈的信息是什么意思呢？我们可以试验一下打印的信息。把上面 **TestExceptionArgs.java** 程序进行修改如下：

```
import java.util.*;
```

```

import java.io.*;
import java.sql.*;
public class TestExceptionArgs{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void mb(int n) throws Exception{
        if (n == 0) throw new SQLException("n==0");
        if (n == 1) throw new EOFException("n==1");
        if (n == 2) throw new FileNotFoundException("n==2");
    }
}

```

在 `catch` 语句中，我们调用 `printStackTrace` 来打印方法调用栈的信息。当 `n` 输入为 1 时，输出结果如下：

```

java.io.EOFException: n==1
    at TestExceptionArgs.mb(TestExceptionArgs.java:21)
    at TestExceptionArgs.ma(TestExceptionArgs.java:13)
    at TestExceptionArgs.main(TestExceptionArgs.java:8)

```

我们可以看到，打印的内容分为了三个部分。首先，是抛出异常的类型名：`java.io.EOFException`。随后，在异常类名之后，是一个冒号，冒号后面的内容，就是我们创建异常时给定的“详细信息”，也可以理解为就是异常的 `message` 属性。

之后，有三行信息，这三行说明了异常产生时的方法调用关系。异常是在 21 行，`mb` 方法中产生的，调用 `mb` 方法的是 13 行的 `ma` 方法，调用 `ma` 方法的是第 8 行的 `main` 方法。通过这样的格式，打印出了产生异常时程序运行的状态和方法调用的关系。

在实际开发过程中，在开发和调试阶段，`printStackTrace` 方法往往会被程序员用来做异常处理，因为这个方法能够为程序员提供非常多的信息，能够帮助程序员更好的找到错误并改正错误。

## 5 自定义异常

我们除了可以使用 Sun 公司已经提供好的异常之外，也可以创建自己的异常体系和异常结构，从而完善自己的软件系统。在 Java 中，自定义异常是相当简单的事情，下面我们就

对自定义异常进行一下介绍。

## 5.1 创建异常类

在 Java 中自定义一个异常类非常简单,只要保证这个异常类继承自 `Exception` 类就可以;而如果想要自定义一个未检查异常,则只要创建一个类继承自 `RuntimeException` 就可以了。例如下面的代码:

```
//自定义已检查异常
class MyException1 extends Exception{
}
//自定义未检查异常
class MyException2 extends RuntimeException{
}
```

定义了这两个自定义异常之后,使用的方式跟 Sun 公司定义的异常类相比,没有任何区别。例如下面的例子:

```
public class TestMyException{
    public static void main(String args[]){
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        ma(n);
    }

    public static void ma(int n) {
        try{
            mb(n);
        }catch(MyException1 e){
            e.printStackTrace();
        }catch(Exception e){
            e.printStackTrace();
        }
    }

    public static void mb(int n) throws MyException1{
        if (n == 0) throw new MyException1();
        if (n == 1) throw new MyException2();
    }
}
```

从上面的代码我们可以看出,创建 `MyException1` 和 `MyException2` 的对象,以及抛出这两个异常,以及 `throws` 语句和 `try-catch` 语句,无论是异常的哪一个方面,我们创建的自定义异常都和 Sun 公司提供的异常完全一样。

也就是说,只要继承自 `Exception` 类或者 `RuntimeException` 类,自定义异常最基本的部分就完成了。

## 5.2 设定 message 属性

但是,如果仅仅让 `MyException` 继承 `Exception` 或 `RuntimeException` 类的话,这样的功

能还不够完善。我们应当让自定义异常和 Sun 公司提供的异常一样,能够设置 `message` 属性。由于 `message` 属性在 `Throwable` 类中,并且没有提供 `setMessage` 的方法,我们只能让异常在起构造方法中对 `message` 属性进行赋值。

由于我们的 `MyException` 本身并没有定义 `message` 属性,因此为了设置 `message` 属性,我们必须为 `MyException` 提供接受字符串作为参数的构造方法。例如下面的代码:

```
public MyException(String msg){
    ...
}
```

在构造方法中应当做哪些事情呢?在构造方法中,需要做的就只有一件事情,那就是调用父类(`Exception` 类或者 `RuntimeException` 类)的带字符串参数的构造方法。修改后的 `MyException1` 和 `MyException2` 代码如下:

```
//自定义已检查异常
class MyException1 extends Exception{
    public MyException1(){}
    public MyException1(String str){
        super(str);
    }
}

//自定义未检查异常
class MyException2 extends RuntimeException{
    public MyException2(){}
    public MyException2(String str){
        super(str);
    }
}
```

代码非常简单。当修改过后,就可以使用带字符串参数的构造方法以及 `getMessage` 方法来使用自定义异常类的 `message` 属性了。