2、具体内容

Java语言提供的最为强大的支持就在于异常的处理操作上。

■认识异常对程序的影响

异常指的是导致程序中断执行的一种指令流，那么下面首选in来观察没有异常产生的程序执行结构。

范例：没有异常产生

```
public class JavaDemo {
	public static void main(String args[]) {
		System.out.println("【1】****** 程序开始执行 ******") ;
		System.out.println("【2】****** 数学计算：" + (10 / 2)) ;
		System.out.println("【3】****** 程序执行完毕 ******") ;
	}
}
```

```
【1】****** 程序开始执行 ******
【2】****** 数学计算：5
【3】****** 程序执行完毕 ******
```

在程序执行正常的过程里面会发现，所有的程序会按照既定的结构从头到尾开始执行。

范例：产生异常

```
public class JavaDemo {
	public static void main(String args[]) {
		System.out.println("【1】****** 程序开始执行 ******") ;
		System.out.println("【2】****** 数学计算：" + (10 / 0)) ;
		System.out.println("【3】****** 程序执行完毕 ******") ;
	}
}
```

```
【1】****** 程序开始执行 ******
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at JavaDemo.main(JavaDemo.java:4)
```

在出现错误之后，整个的程序将不会按照既定的方式进行执行，而是中断了执行，那么为了保证程序出现了非致命错误之后程序依然可以正常完成，所以就需要有一个完善的异常处理机制，以保证程序的顺利执行。

■处理异常

在Java之中如果要进行异常的处理，可以使用try、catch、finally这几个关键字来完成，其基本的处理结构如下：

```
try {
	// 可能出现异常的语句
} [catch (异常类型 异常对象) {
	// 异常处理
} catch (异常类型 异常对象) {
	// 异常处理
} catch (异常类型 异常对象) {
	// 异常处理
} ... ] [finally {
	不管异常是否处理都要执行 ;
}]
```

在此格式之中可以使用的组合为：try……catch、try……catch……finally、try……finally。

范例：处理异常

```
public class JavaDemo {
	public static void main(String args[]) {
		System.out.println("【1】****** 程序开始执行 ******") ;
		try {
			System.out.println("【2】****** 数学计算：" + (10 / 0)) ;
		} catch (ArithmeticException e) {
			System.out.println("【C】处理异常：" + e) ;  // 处理异常
		}
		System.out.println("【3】****** 程序执行完毕 ******") ;
	}
}
```

```
【1】****** 程序开始执行 ******
【C】处理异常：java.lang.ArithmeticException: / by zero
【3】****** 程序执行完毕 ******
```

此时可以发现现在即便出现了异常，程序也可以正常执行完毕，所以此时的设计属于一个合理设计，但是有一个问题出现了。此时在进行异常处理的时候直接输出的是一个异常类的对象，那么对于此对象如果直接打印（调用toString()）所得到的的异常信息并不完整，那么如果想要获得非常完整的异常信息，则可以使用异常类中提供的printStackTrace()方法。

范例：获取完整异常信息

```
public class JavaDemo {
```

```
    public static void main(String args[]) {
        System.out.println("【1】****** 程序开始执行 ******") ;
        try {
            System.out.println("【2】****** 数学计算：" + (10 / 0)) ;
        } catch (ArithmeticException e) {
            e.printStackTrace() ;
        }
        System.out.println("【3】****** 程序执行完毕 ******") ;
    }
}
```

对于异常的处理格式也可以在最后追加有一个finally程序块，表示异常处理后的出口，不管是否出现异常都执行。

范例：使用finally语句

```
public class JavaDemo {
    public static void main(String args[]) {
        System.out.println("【1】****** 程序开始执行 ******") ;
        try {
            System.out.println("【2】****** 数学计算：" + (10 / 0)) ;
        } catch (ArithmeticException e) {
            e.printStackTrace() ;
        } finally {
            System.out.println("【F】不管是否出现异常，我都会执行。") ;
        }
        System.out.println("【3】****** 程序执行完毕 ******") ;
    }
}
```

此时程序中有异常执行finally，没有异常也执行finally。

■处理多个异常

很多时候在程序执行的过程之中可能会产生若干个异常，那么这种情况下可以使用多个catch进行异常的捕获。现在假设通过初始化参数来进行两个数学计算数字的设置。

```
public class JavaDemo {
    public static void main(String args[]) {
        System.out.println("【1】****** 程序开始执行 ******") ;
        try {
            int x = Integer.parseInt(args[0]) ;
            int y = Integer.parseInt(args[1]) ;
            System.out.println("【2】****** 数学计算：" + (x / y)) ;
        } catch (ArithmeticException e) {
            e.printStackTrace() ;
        } finally {
            System.out.println("【F】不管是否出现异常，我都会执行。") ;
        }
        System.out.println("【3】****** 程序执行完毕 ******") ;
    }
}
```

那么对于此时的程序就有可能产生三类异常：

- 【未处理】程序执行的时候没有输入初始化参数（java JavaDemo）：

java.lang.ArrayIndexOutOfBoundsException:

程序执行结果:	【1】****** 程序开始执行 ****** 【F】不管是否出现异常，我都会执行。 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0 at JavaDemo.main(JavaDemo.java:5)

- 【未处理】输入的数据不是数字（java JavaDemo a b）：

java.lang.NumberFormatException: For input string:

程序执行结果:	【1】****** 程序开始执行 ****** 【F】不管是否出现异常，我都会执行。 Exception in thread "main" java.lang.NumberFormatException: For input string: "a" at java.lang.NumberFormatException.forInputString(Unknown Source) at java.lang.Integer.parseInt(Unknown Source) at java.lang.Integer.parseInt(Unknown Source) at JavaDemo.main(JavaDemo.java:5)

- 【已处理】输入的被除数为0（java JavaDemo 10 0）：

java.lang.ArithmeticException:

程序执行结果:	【1】****** 程序开始执行 ****** java.lang.ArithmeticException: / by zero at JavaDemo.main(JavaDemo.java:7) 【F】不管是否出现异常，我都会执行。 【3】****** 程序执行完毕 ******

如果即便有了异常处理语句，但是如果没有进行正确的异常捕获，那么程序也会导致中断（finally的代码依然执行），所以在这样的情况下就必须进行多个异常的捕获。

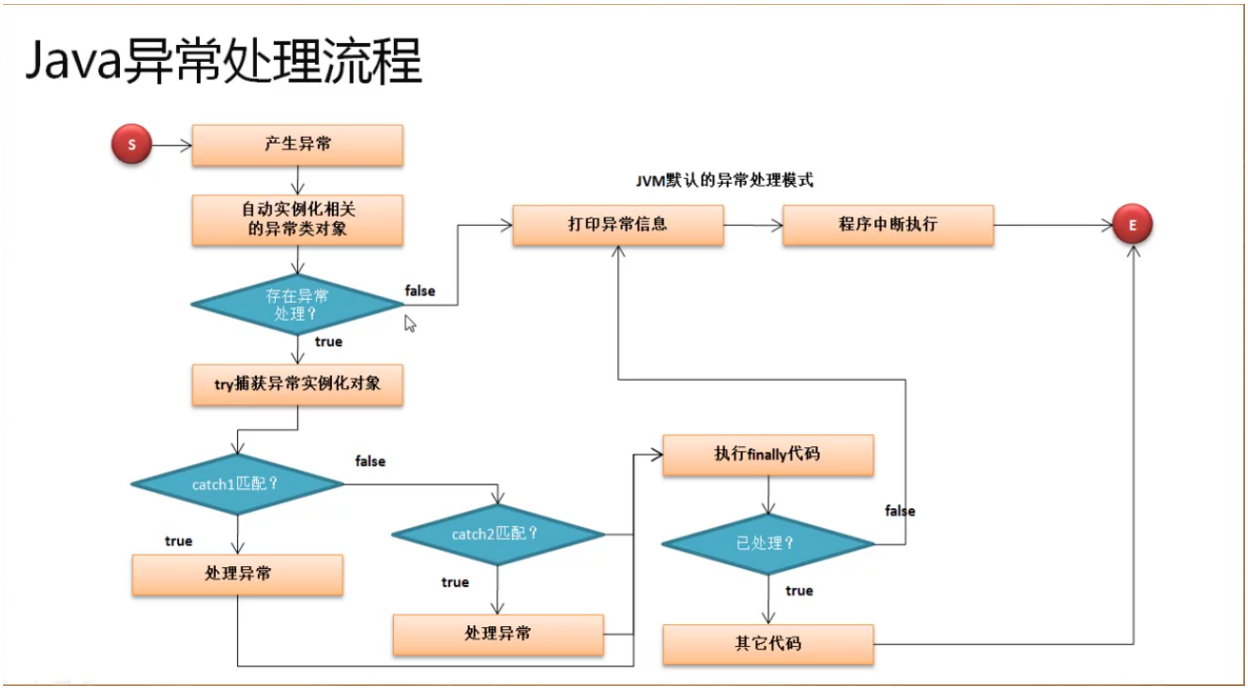
范例：修改程序代码

```
public class JavaDemo {
    public static void main(String args[]) {
        System.out.println("【1】****** 程序开始执行 ******") ;
        try {
            int x = Integer.parseInt(args[0]) ;
            int y = Integer.parseInt(args[1]) ;
            System.out.println("【2】****** 数学计算: " + (x / y)) ;
        } catch (ArithmeticException e) {
            e.printStackTrace() ;
        } catch (NumberFormatException e) {
            e.printStackTrace() ;
        } catch (ArrayIndexOutOfBoundsException e) {
            e.printStackTrace() ;
        } finally {
            System.out.println("【F】不管是否出现异常，我都会执行。") ;
        }
        System.out.println("【3】****** 程序执行完毕 ******") ;
    }
}
```

此时我们开发者都已经明确的知道了有那些异常了，那么又何必非要用个异常处理呢？直接多写点判断不就可以了。

■异常处理流程

在进行异常处理的时候如果将所有可能已经明确知道要产生的异常都进行了捕获，虽然你可以得到非常良好的代码结构，但是这种代码编写是非常麻烦的，所以现在要想进行合理异常就必须清楚在异常产生之后程序到底做了哪些处理。



1、在程序运行的过程之中才会产生异常，而一旦程序执行中产生了异常之后将自动进行指定类型的异常类对象实例化处理。

2、如果此时程序之中并没有提供有异常处理的支持，则会采用JVM默认异常处理方式，首先进行异常信息的打印，而后直接退出当前的程序。

3、此时程序中如果存在有异常处理，那么这个产生的异常类的实例化对象将会被try语句所捕获。

4、try捕获到异常之后与其匹配的catch中的异常类型进行依次的比对，如果此时与catch中的捕获异常类型相同，则认为应该使用此catch进行异常处理；如果不匹配则继续匹配后续的catch类型，如果现在没有任何的catch匹配成功，那么就表示该异常无法进行处理。

5、不管异常是否处理最终都要执行finally语句，但是当执行完成finally的程序之后会进一步判断当前的异常是否已经处理过了，如果处理过了，则继续向后执行其它代码；如果没有处理则交由JVM进行默认的处理。

通过分析可以发现在整个的异常处理流程之中实际上操作的还是一个异常类的实例化对象，那么这个异常类的实例化对象的类型就成为了理解异常处理的核心关键所在，在之前接触过了两种异常：

ArithmeticException	**ArrayIndexOutOfBoundsException**

java.lang.Object
  |- java.lang.**Throwable**
  |- java.lang.**Exception**
  |- java.lang.Runtime**Exception**
  |- java.lang.ArithmeticException
	java.lang.Object
	  |- java.lang.**Throwable**
	  |- java.lang.**Exception**
	  |- java.lang.Runtime**Exception**
	  |- java.lang.IndexOutOfBoundsException
	  |- java.lang.ArrayIndexOutOfBoundsException

可以发现在成之中可以处理的异常的最大的类型就是Throwable，而打开Throwable可以观察在此类中提供有两个子类：

- Error：此时程序还未执行出现的错误，开发者无法处理；
- Exception：程序中出现的异常，开发者可以处理，真正在开发之中所需要关注的就是Exception；

通过分析可以发现异常产生的时候回产生异常的实例化对象，那么按照对象的引用原则，可以自动向父类转型，那么如果按照这样的逻辑，实际上所有的异常都可以使用Exception来处理。

范例：简化异常处理

```
public class JavaDemo {
    public static void main(String args[]) {
        System.out.println("【1】****** 程序开始执行 ******") ;
        try {
            int x = Integer.parseInt(args[0]) ;
            int y = Integer.parseInt(args[1]) ;
            System.out.println("【2】****** 数学计算：" + (x / y)) ;
        } catch (Exception e) {
            e.printStackTrace() ;
        } finally {
            System.out.println("【F】不管是否出现异常，我都会执行。") ;
        }
        System.out.println("【3】****** 程序执行完毕 ******") ;
    }
}
```

当你不确定可能产生哪些异常的时候，这种处理方式是最方便的。但是如果这样处理也会产生一个问题。这种异常的处理方式虽然方便，但是它描述的错误信息不明确，所以分开处理异常是一种可以更加明确的处理方式。

在以后进行多个异常同时处理的时候要把捕获范围大的异常放在捕获范围小的异常之后。

■throws关键字

通过之前的程序可以发现，在执行程序的过程之中有可能会产生异常，但是如果说现在假设你定义了一个方法，实际上就应该明确的告诉使用者，这个方法可能会产生何种异常，

那么此时就可以在方法的声明上使用throws关键字来进行异常类型的标注。

范例：观察throws的使用

```
class MyMath {
	// 这个代码执行的时候可能会产生异常，如果产生异常调用处处理
	public static int div(int x,int y) throws Exception {
		return x / y ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		try {
			System.out.println(MyMath.div(10,0)) ;
		} catch (Exception e) {
			e.printStackTrace() ;
		}
	}
}
```

主方法本身也还是一个方法，那么实际上主方法也可以继续向上抛出。

范例：在主方法上继续抛出异常

```
class MyMath {
	// 这个代码执行的时候可能会产生异常，如果产生异常调用处处理
	public static int div(int x,int y) throws Exception {
		return x / y ;
	}
}
public class JavaDemo {
	public static void main(String args[]) throws Exception {
		System.out.println(MyMath.div(10,0)) ;
	}
}
```

如果主方法继续向上抛出异常，那么就表示此异常将交由JVM负责处理。

■throw关键字

与throws对应的还有throw关键字，此关键字的主要作用在于表示手工进行异常的抛出，即：此时将手工产生一个异常类的实例化对象，并且进行异常的抛出处理。

范例：观察throw的使用

```
public class JavaDemo {
	public static void main(String args[]) {
		try{  //异常对象不再是由系统生成的，而是由手工定义的
			throw new Exception("自己抛着玩的对象");
		}catch (Exception e){
			e.printStackTrace();
		}
	}
}
```

面试题：请解释throw与throws区别？

- throw：是在代码中使用的，主要是手工进行异常对象的抛出；
- throws：是在方法定义上使用的，表示将此方法中可能产生的异常明确告诉给调用处，有调用处进行处理；

■异常处理的标准格式

现在已经学习完成了大部分的异常处理格式：try、catch、finally、throw、throws，那么这些关键字在实际的开发之中往往会一起进行使用，下面通过一个具体的程序来进行分析。

现在要求定义一个可以实现除法计算的方法，在这个方法之中开发要求如下：

- 在进行数学计算开始于结束的时候进行信息提示；
- 如果在进行计算的过程之中产生了异常，则要交给调用处来处理。

```
class MyMath {
	// 异常交给被调用处处理则一定要在方法上使用throws
	public static int div(int x,int y) throws Exception {
		int temp = 0 ;
		System.out.println("*** 【START】除法计算开始。") ;
		try {
			temp = x / y ;
		} catch (Exception e) {
			throw e ; // 向上抛异常对象
		} finally {
			System.out.println("*** 【END】除法计算结束。") ;
		}
		return temp ;
	}
}
public class JavaDemo {
	public static void main(String args[]) {
		try {
			System.out.println(MyMath.div(10,0)) ;
		} catch (Exception e) {
			e.printStackTrace() ;
		}
	}
}
```

对于此类操作实际上可以简化，省略掉catch与throw的操作。

```
class MyMath {
	// 异常交给被调用处处理则一定要在方法上使用throws
	public static int div(int x,int y) throws Exception {
		int temp = 0 ;
		System.out.println("*** 【START】网络资源连接。") ;
		try {
			temp = x / y ;
```

```
            } finally {
                System.out.println("*** 【END】网络连接关闭。") ;
            }
            return temp ;
        }
}
public class JavaDemo {
        public static void main(String args[]) {
            try {
                System.out.println(MyMath.div(10,0)) ;
            } catch (Exception e) {
                e.printStackTrace() ;
            }
        }
}
```

在以后实际开发过程之中，这种异常的处理格式是最为重要的，尤其是当与一些资源进行访问操作的时候尤其重要。

■RuntimeException

通过之前的分析可以发现只要方法后面带有throws往往都是告诉用户本方法可能产生的异常是什么，所以这个时候来观察一段代码

```
public class JavaDemo {
        public static void main(String args[]) {
            int num = Integer.parseInt("123") ;
            System.out.println(num) ;
        }
}
```

打开Integer类中parseInt()方法的定义来观察：public static int parseInt(String s) throws NumberFormatException;

这个方法上明确的抛出了一个异常，但是在处理的时候并没有强制性要求处理，观察一下NumberFormatException类的继承结构，同时也观察数学类异常类的继承结构：

ArithmeticException	**NumberFormatException**

java.lang.Object
  |- java.lang.**Throwable**
  |- java.lang.**Exception**
  |- **java.lang.RuntimeException**
  |- java.lang.Arithmetic**Exception**
	java.lang.Object
	  |- java.lang.Throwable
	  |- java.lang.Exception
	  |- **java.lang.RuntimeException**
	  |- java.lang.IllegalArgumentException
	  |- java.lang.NumberFormatException

如果现在所有的程序执行上只要使用了throws定义的方法都必须要求开发者进行手工处理，那么这个代码的编写就太麻烦了，所以在设计的过程之中，考虑到代表编写的方便，所以提供有一个灵活的可选的异常处理父类“RuntimeException”，这个类的异常子类可以不需要强制性处理。

面试题：请解释RuntimeException与Exception的区别？请列举几个常见的RuntimeException

- RuntimeException是Exception的子类；
- RuntimeException标注的异常可以不需要强制性处理，而Exception异常必须强制性处理；
- 常见的RuntimeException异常：NumberFormatException 、ClassCastException、NullPointerException

---

■自定义异常类

在JDK之中提供有大量的异常类型，但是在实际的开发之中可能这些异常类型未必够你所使用，你不可能所有的设计里面都只是抛出Exception，所以这个时候就需要考虑进行自定义异常类。但是对于自定义异常也有两种实现方案：继承Exception、继承RuntimeException。

范例：实现自定义异常

```
class BombException extends Exception {
	public BombException(String msg) {
		super(msg) ;
	}
}
class Food {
	public static void eat(int num) throws BombException {
		if (num > 10) {
			throw new BombException("吃太多了，肚子爆了。") ;
		} else {
			System.out.println("正常开始吃，不怕吃胖。") ;
		}
	}
}
public class JavaDemo {
	public static void main(String args[]) throws Exception {
		Food.eat(11) ;
	}
}
```

在以后的项目开发之中会接触到大量的自定义异常处理，如果遇见了不不清楚的异常，最简单的方式就是通过搜索引擎查询一下异常可能产生的原因。

■assert断言

从JDK1.4开始追加有一个断言的功能，确定代码执行到某行之后一定是所期待的结果。在实际的开发之中，对于断言而言，并不一定是准确的，也有可能出现偏差，但是这种偏差

不应该影响程序的正常执行。

范例：断言的使用

```
public class JavaDemo {
    public static void main(String args[]) throws Exception {
        int x = 10 ;
        // 中间会经过许多的x变量的操作步骤
        assert x == 100 : "x的内容不是100" ;
        System.out.println(x) ;
    }
}
```

如果现在要想执行断言，则必须在程序执行的时候加入参数：

java -ea JavaDemo	Exception in thread "main" java.lang.AssertionError: x的内容不是100at JavaDemo.main(JavaDemo.java:5)

所以在Java里面并没有将断言设置为一个程序必须执行的步骤，需要特点环境下才可以开启