# day09【排序算法、异常、多线程基础】

## 今日内容

- 排序算法
  - 冒泡排序
  - 选择排序
- 查找算法
  - 二分查找
- 异常处理---->掌握
- 多线程基础
  - 创建线程并启动线程的三种方式---->重点\掌握

## 教学目标

- [ ] 能够理解冒泡排序的执行原理
- [ ] 能够理解选择排序的执行原理
- [ ] 能够理解二分查找的执行原理
- [ ] 能够辨别程序中异常和错误的区别
- [ ] 说出异常的分类
- [ ] 列举出常见的三个运行期异常
- [ ] 能够使用try...catch关键字处理异常
- [ ] 能够使用throws关键字处理异常
- [ ] 能够自定义并使用异常类
- [ ] 说出进程和线程的概念
- [ ] 能够理解并发与并行的区别
- [ ] 能够描述Java中多线程运行原理
- [ ] 能够使用继承类的方式创建多线程
- [ ] 能够使用实现接口的方式创建多线程
- [ ] 能够说出实现接口方式的好处

# 第一章 冒泡排序

## 知识点-- 冒泡排序

### 目标

- 能够理解冒泡排序的执行原理

### 路径

- 冒泡排序概述

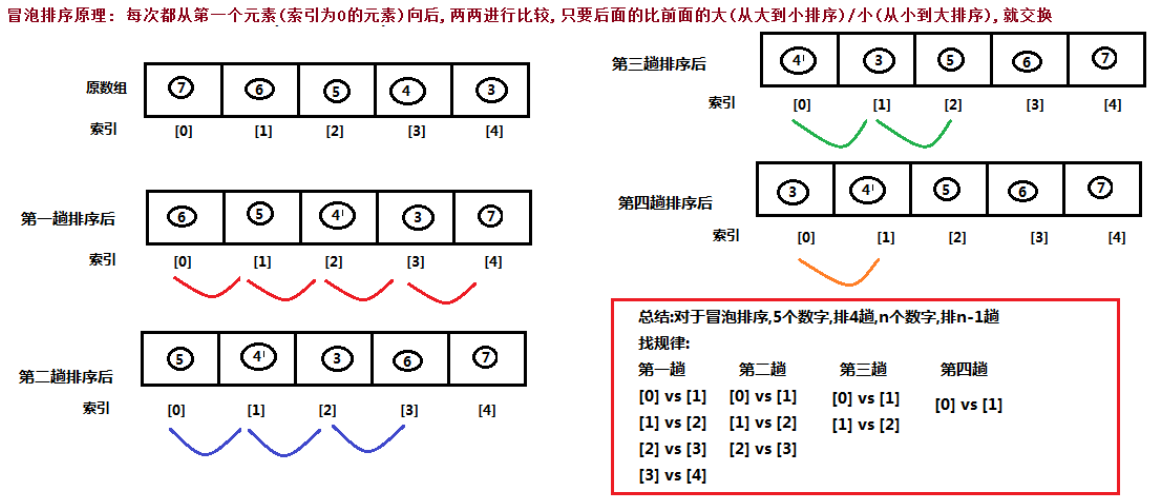
- 冒泡排序图解
- 冒泡排序代码实现

# 讲解

## 冒泡排序概述

- 一种排序的方式，对要进行排序的数据中**相邻的数据进行两两比较**，将较大的数据放在后面，依次对所有的数据进行操作，直至所有数据按要求完成排序
- 如果有n个数据进行排序，总共需要比较n-1次
- 每一次比较完毕，下一次的比较就会少一个数据参与

## 冒泡排序图解




## 冒泡排序代码实现

```
package com.itheima.demo1_冒泡排序;

import java.util.Arrays;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 9:05
 */
public class Test {
    public static void main(String[] args) {
        //定义一个数组
        int[] arr = {7, 6, 18, 5, 4, 3};
        System.out.println("排序前: " + Arrays.toString(arr));

        // 冒泡排序
        // 外层循环控制比较的轮数
        for (int i = 0; i < arr.length - 1; i++) {
            // 内层循环控制比较的次数
            for (int j = 0;j<arr.length-1-i;j++){
                // 比较判断 arr[j]与arr[j+1]
                if (arr[j] > arr[j+1]){
                    // 交换
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
```

```
            }
        }


        System.out.println("排序后：" + Arrays.toString(arr));
    }
}
```

## 小结

略

# 第二章 选择排序

## 知识点-- 选择排序

### 目标

- 能够理解选择排序的执行原理

### 路径

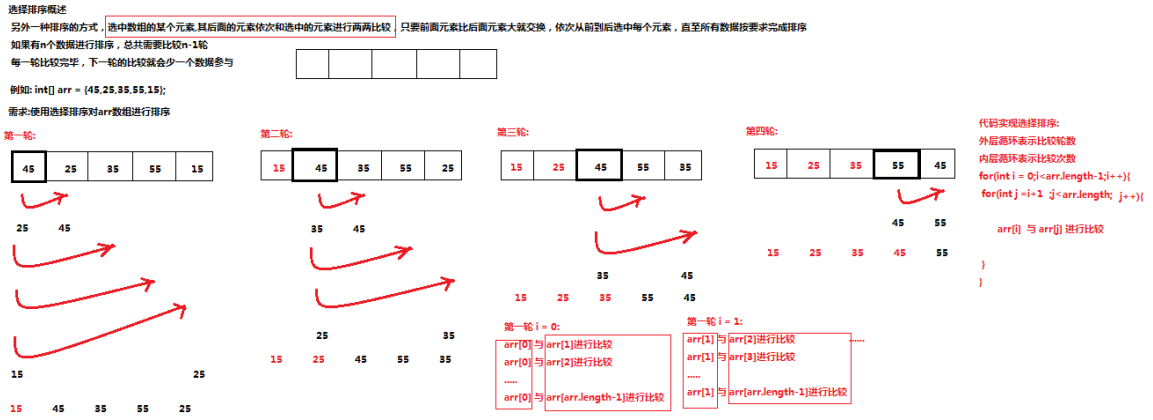
- 选择排序概述
- 选择排序图解
- 选择排序代码实现

### 讲解

#### 选择排序概述

- 另外一种排序的方式，选中数组的某个元素,与其后面的元素依次进行两两比较，将较大的数据放在后面，依次从前到后选中每个元素，直至所有数据按要求完成排序
- 如果有n个数据进行排序，总共需要比较n-1轮
- 每一轮比较完毕，下一轮的比较就会少一个数据参与

#### 选择排序图解




#### 选择排序代码实现

```java
package com.itheima.demo2_选择排序;

import java.util.Arrays;

/**
 * @Author：pengzhilin
 * @Date: 2020/9/16 9:33
 */
public class Test {
    public static void main(String[] args) {
        int[] arr = {45, 25, 35, 55, 15};
        System.out.println("排序前：" + Arrays.toString(arr));

        // 选择排序
        // 外层循环控制比较的轮数
        for (int i = 0; i < arr.length - 1; i++) {
            // 内层循环控制比较的次数
            for (int j = i + 1; j < arr.length; j++) {
                // 比较判断: arr[i]vsarr[j]
                if (arr[i] > arr[j]){
                    int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }
            }
        }

        System.out.println("排序后：" + Arrays.toString(arr));
    }
}
```

## 小结

略

# 第三章 二分查找

## 知识点-- 二分查找

### 目标

- 能够理解二分查找的执行原理

### 路径

- 普通查找和二分查找
- 二分查找图解
- 二分查找代码实现

### 讲解

#### 普通查找和二分查找

##### 普通查找

原理：遍历数组，获取每一个元素，然后判断当前遍历的元素是否和要查找的元素相同，如果相同就返回该元素的索引。如果没有找到，就返回一个负数作为标识(一般是-1)

**二分查找**

原理: 每一次都去获取数组的中间索引所对应的元素，然后和要查找的元素进行比对，如果相同就返回索引;

如果不相同，就比较中间元素和要查找的元素的值;

如果中间元素的值大于要查找的元素，说明要查找的元素在左侧，那么就从左侧按照上述思想继续查询(忽略右侧数据);

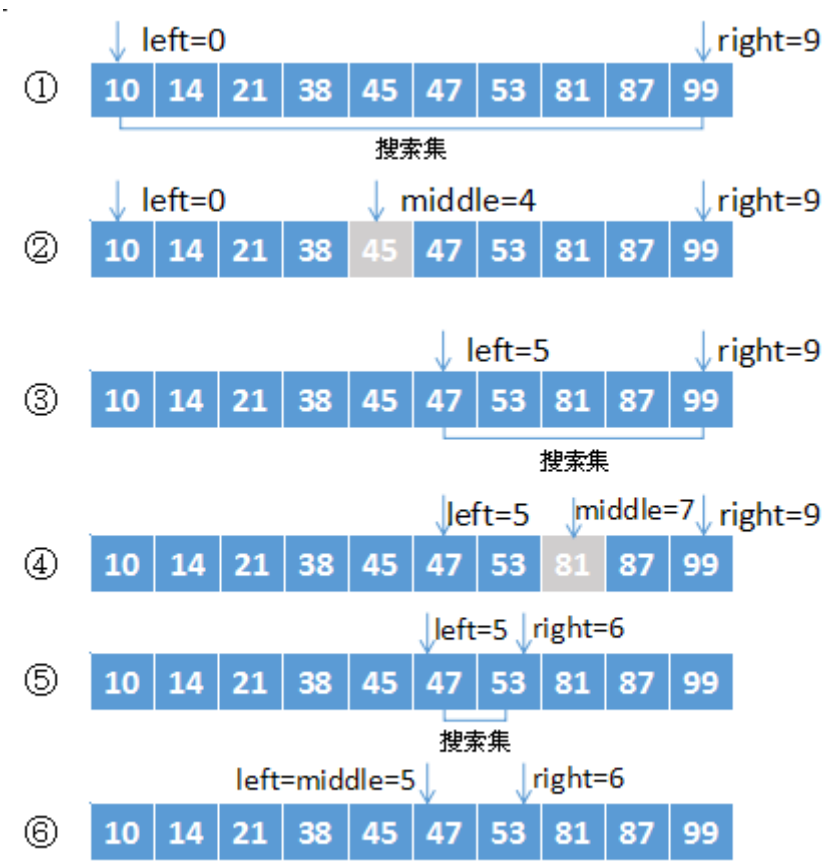
如果中间元素的值小于要查找的元素，说明要查找的元素在右侧，那么就从右侧按照上述思想继续查询(忽略左侧数据);

**二分查找对数组是有要求的,数组必须已经排好序**

## 二分查找图解

假设有一个给定有序数组(10,14,21,38,45,47,53,81,87,99),要查找50出现的索引

则查询过程如下图所示:




## 二分查找代码实现

```
    public static void main(String[] args) {
        int[] arr = {10, 14, 21, 38, 45, 47, 53, 81, 87, 99};
        int index = binarySerach(arr, 38);
        System.out.println(index);
    }
    /**
```

```
     * 二分查找方法
     * @param arr 查找的目标数组
     * @param number 查找的目标值
     * @return 找到的索引,如果没有找到返回-1
     */
    public static int binarySerach(int[] arr, int number) {
        int start = 0;
        int end = arr.length - 1;

        while (start <= end) {
            int mid = (start + end) / 2;
            if (number == arr[mid]) {
                return mid ;
            } else if (number < arr[mid]) {
                end = mid - 1;
            } else if (number > arr[mid]) {
                start = mid + 1;
            }
        }
        return -1;  //如果数组中有这个元素，则返回
    }
```

## 小结

略

# 第四章 异常

## 知识点-- 异常

### 目标

- 能够辨别程序中异常和错误的区别,并且说出异常的分类

### 路径

- 异常概念
- 异常体系
- 异常分类
- 异常的产生过程解析

### 讲解

#### 异常概念

异常，就是不正常的意思。在生活中:医生说,你的身体某个部位有异常,该部位和正常相比有点不同,该部位的功能将受影响.在程序中的意思就是：

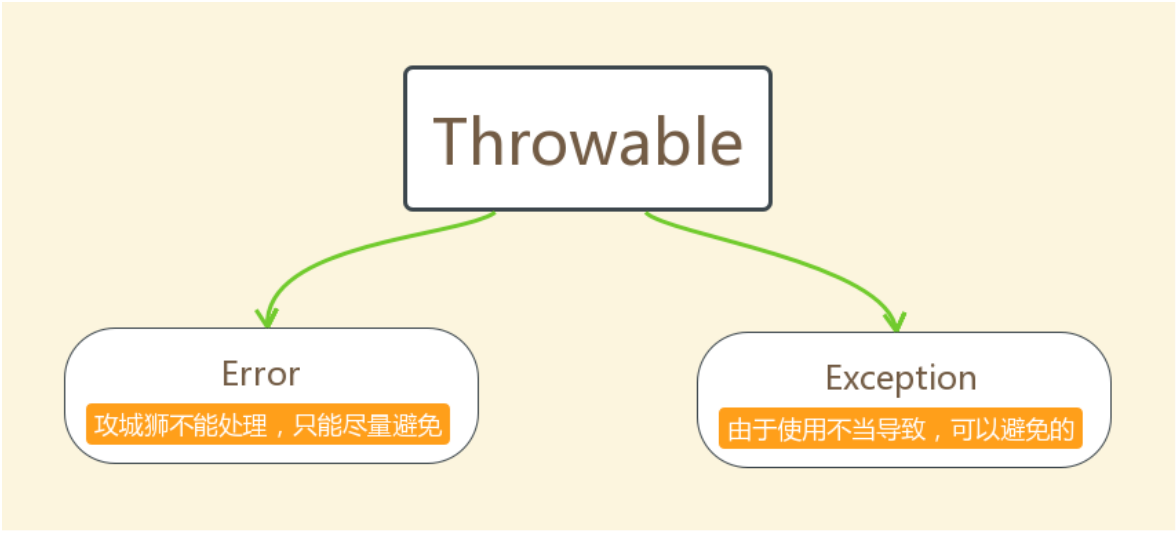
- **异常** ： 指的是程序在执行过程中，出现的非正常的情况，最终会导致JVM的非正常停止。

**注意:** 在Java等面向对象的编程语言中，**异常本身是一个类，产生异常就是创建异常对象并抛出了一个异常对象。**Java处理异常的方式是中断处理。

> 异常指的并不是语法错误,语法错了,编译不通过,不会产生字节码文件,根本不能运行.

#### 异常体系

异常机制其实是帮助我们**找到**程序中的问题，异常的根类是`java.lang.Throwable`，其下有两个子类：`java.lang.Error`与`java.lang.Exception`，平常所说的异常指`java.lang.Exception`。




**Throwable体系：**

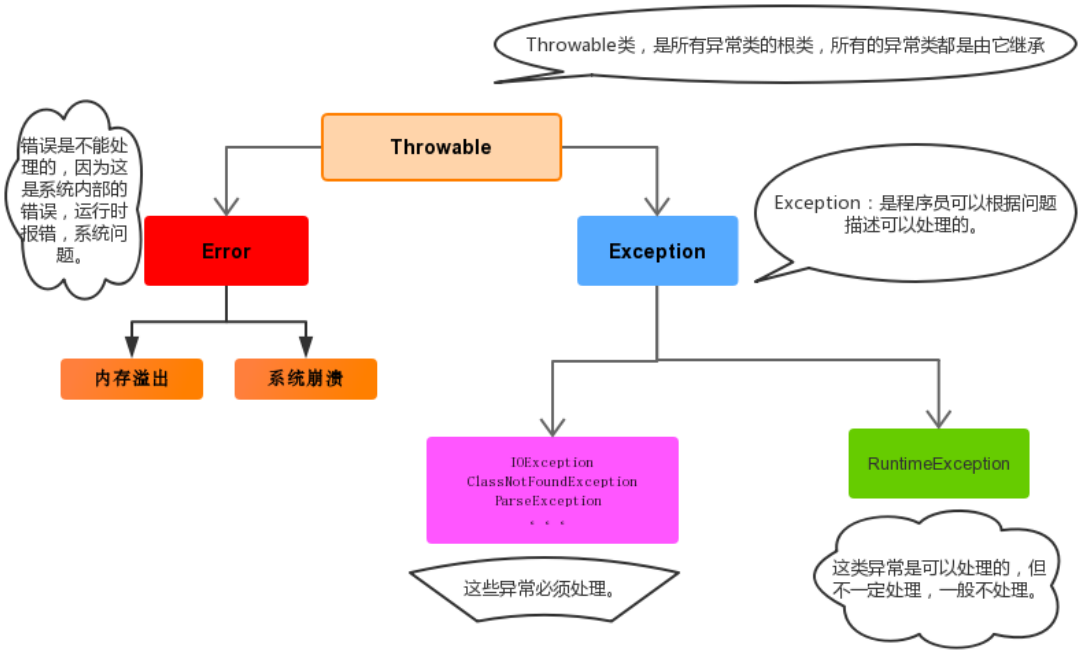
- **Error**:严重错误Error，无法通过处理的错误，只能事先避免，好比绝症。
- **Exception**:表示异常，异常产生后程序员可以通过代码的方式纠正，使程序继续运行，是必须要处理的。好比感冒、阑尾炎。

## 异常分类

我们平常说的异常就是指Exception，因为这类异常一旦出现，我们就要对代码进行更正，修复程序。

**异常(Exception)的分类**:根据在编译时期还是运行时期去检查异常?

- **编译时期异常**:checked异常。在编译时期,就会检查,如果没有处理异常,则编译失败。(如日期格式化异常)
- **运行时期异常**:runtime异常。在运行时期,检查异常.在编译时期,运行异常不会编译器检测(不报错)。(如数学异常)




## 异常的产生过程解析

先运行下面的程序，程序会产生一个数组索引越界异常ArrayIndexOfBoundsException。我们通过图解来解析下异常产生的过程。

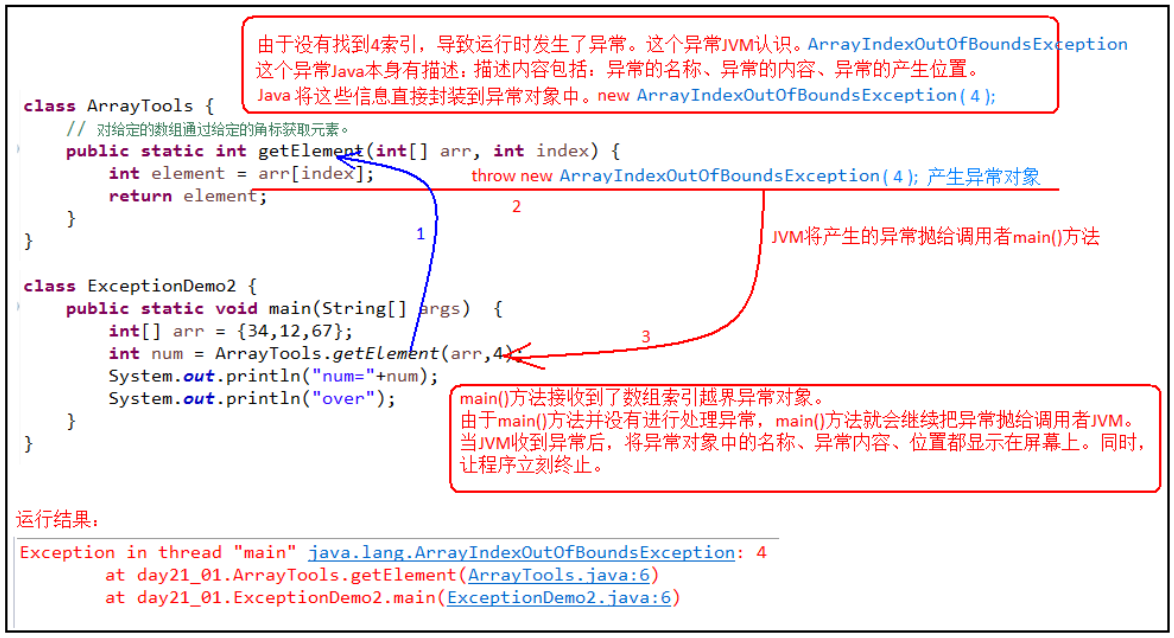
工具类

```
public class ArrayTools {
    // 对给定的数组通过给定的角标获取元素。
    public static int getElement(int[] arr, int index) {
        int element = arr[index];
        return element;
    }
}
```

测试类

```
public class ExceptionDemo {
    public static void main(String[] args) {
        int[] arr = { 34, 12, 67 };
        intnum = ArrayTools.getElement(arr, 4)
        System.out.println("num=" + num);
        System.out.println("over");
    }
}
```

上述程序执行过程图解：




## 小结

略

# 第五章 异常的产生和处理

## 知识点-- 异常的产生

## 目标

- 能够理解使用throw关键字产生异常

## 路径

- throw关键字的作用
- throw关键字的使用格式
- 案例演示

## 讲解

### throw关键字的作用

在java中，提供了一个**throw**关键字，它用来抛出一个指定的异常对象。throw**用在方法内**，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。

### throw关键字的使用格式

```
throw new 异常类名(参数);
```

例如：

```
throw new NullPointerException("要访问的arr数组不存在");
throw new ArrayIndexOutOfBoundsException("该索引在数组中不存在，已超出范围");
```

### 案例演示

```
package com.itheima.demo6_throw关键字;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 11:49
 */
public class Test {
    public static void main(String[] args) {
        /*
            throw关键字:
                作用:throw用在方法内，用来抛出一个异常对象，将这个异常对象传递到调用者处，并结束当前方法的执行。
                格式: throw 异常对象;
         */
        int[] arr = {10,20,30,40};
        method(arr,4);
    }

    /**
     * 查找指定索引位置的元素
     * @param arr
     * @param index
     */
    public static void method(int[] arr,int index){
        if (index < 0 || index > arr.length-1){
            // 索引不存在-->产生一个异常
            throw new ArrayIndexOutOfBoundsException(index+"");
        }else{
            int num = arr[index];
```

```
            System.out.println(num);
        }
    }
}
```

## 小结

略

# 知识点--声明处理异常

## 目标

- 掌握声明处理异常

## 路径

- 声明处理异常的概述
- 声明处理异常格式

## 讲解

### 声明处理异常的概述

**声明处理异常**：使用throws关键字将问题标识出来，表示当前方法不处理异常，而是提醒给调用者，让调用者来处理....最终会到虚拟机,虚拟机直接结束程序,打印异常信息。

### 声明处理异常格式

```
修饰符 返回值类型 方法名(参数) throws 异常类名1,异常类名2...{  // 可以抛出一个,也可以多个
}
```

### 案例演示

```
package com.itheima.demo7_声明处理异常;

import java.io.FileNotFoundException;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 11:58
 */
public class Test {
    // 使用throws关键字将异常标识出来，表示当前方法不处理异常，而是提醒调用者来处理
    public static void main(String[] args) throws ParseException {
        /*
            处理异常的目的: 为了让程序可以继续往下执行
            声明处理异常:
                概述:使用throws关键字将问题标识出来，表示当前方法不处理异常，而是提醒给调用者,
                    让调用者来处理....最终会到虚拟机,虚拟机直接结束程序,打印异常信息。
```

```
                格式:
                    修饰符 返回值类型 方法名(形参列表) throws 异常类名1,异常类名2…{  //可以抛出一个,也可以多个

                    }
                特点: 声明处理异常,处理完后,如果程序运行的时候出现异常,程序还是无法继续往下执行
                使用场景:  声明处理异常一般处理运行的时候不会出现异常的编译异常

         */
        //method1();

        // 举例:声明处理异常一般处理运行的时候不会出现异常的编译异常
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date date = sdf.parse("1999-10-10");
        System.out.println(date);

        // 举例:一般用来处理编译异常,让程序通过编译,但程序运行的时候出现异常,程序还是无法继续往下执行
        /*SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
        Date date = sdf.parse("1999年10月10日");
        System.out.println(date);*/

        System.out.println("结束");

    }


    // 一次声明处理一个异常
    // 使用throws关键字将异常标识出来, 表示当前方法不处理异常, 而是提醒调用者来处理
    public static void method1() throws ParseException{
        // 产生一个异常对象
        throw new ParseException("解析异常",1);
    }

    // 一次声明处理多个异常
    public static void method2(int num) throws ParseException,FileNotFoundException{
        // 产生一个异常对象
        if (num == 1){
            throw new ParseException("解析异常",1);
        }else{
            throw new FileNotFoundException("文件找不到异常");
        }
    }

}
```

## 小结

略

# 知识点--捕获处理异常try...catch

# 目标

- 掌握捕获处理异常

# 路径

- 捕获处理异常的概述
- 捕获处理异常格式
- 获取异常信息

# 讲解

## 捕获处理异常的概述

- **捕获处理异常**：对异常进行捕获处理，处理完后程序可以正常向下执行。

## 捕获处理异常格式

```
try{
      编写可能会出现异常的代码
}catch(异常类型  e){
      处理异常的代码
      //记录日志/打印异常信息
}
执行步骤:
    1.首先执行try中的代码,如果try中的代码出现了异常,那么就直接执行catch()里面的代码,执行完后,程序继续往下执行
    2.如果try中的代码没有出现异常,那么就不会执行catch()里面的代码,而是继续往下执行
```

> 注意:
>
> 1. try和catch都不能单独使用,必须连用。
> 2. try中的代码出现了异常,那么出现异常位置后面的代码就不会再执行了
> 3. 捕获处理异常,如果程序出现了异常,程序会继续往下执行
>
> 声明处理异常,如果程序出现了异常,程序就不会继续往下执行

演示如下：

```
package com.itheima.demo8_捕获处理异常;

import java.text.ParseException;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 12:24
 */
public class Test {
    public static void main(String[] args) {
        /*
            捕获处理异常: try...catch..
                概述:对异常进行捕获处理 ， 处理完后程序可以正常向下执行。
                格式:
                    try{
                          编写可能会出现异常的代码
                    }catch(异常类型  变量名){
                          处理异常的代码
```

```
                        //记录日志/打印异常信息
                    }
                执行步骤:
                    1.首先执行try中的代码,如果try中的代码出现了异常,那么就直接执行catch()里面的代码,执行完后,程序继续往下执行
                    2.如果try中的代码没有出现异常,那么就不会执行catch()里面的代码,而是继续往下执行
                 */
        method1();
        System.out.println("======================");
        // 捕获处理运行异常
        try {
            System.out.println(1/1);// 没有出现异常,1
        }catch (ArithmeticException e){
            System.out.println("出现了数学运算异常");
        }
        System.out.println("======================");

        try {
            System.out.println(1/0);// 出现了异常
            System.out.println("try...");
        }catch (ArithmeticException e){
            System.out.println("出现了数学运算异常");
        }
        System.out.println("结束");
    }

    // 捕获处理编译异常
    public static void method1(){
        try{
            throw new ParseException("解析异常",1);
        }catch (ParseException e){
            System.out.println("出现了异常");
        }
        System.out.println("method1方法结束...");
    }
}
```

## 获取异常信息

Throwable类中定义了一些查看方法:

- `public String getMessage()`:获取异常的描述信息,原因(提示给用户的时候,就提示错误原因。
- `public String toString()`:获取异常的类型和异常描述信息(不用)。
- `public void printStackTrace()`:打印异常的跟踪栈信息并输出到控制台。

*包含了异常的类型,异常的原因,还包括异常出现的位置,在开发和调试阶段,都得使用printStackTrace。*

在开发中呢也可以在catch将编译期异常转换成运行期异常处理。

```
package com.itheima.demo9_Throwable获取异常信息的方法;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 14:47
 */
```

```
public class Test {
    public static void main(String[] args) {
        /*
            Throwable获取异常信息的方法:
            - public String getMessage():获取异常的描述信息,原因(提示给用户的时候,就提
示错误原因。
            - public String toString():获取异常的类型和异常描述信息(不用)。
            - public void printStackTrace():打印异常的跟踪栈信息并输出到控制台。
         */
        System.out.println("开始");

        try {
            System.out.println(1/0);// 报异常,产生一个异常对象

        }catch (ArithmeticException e){
            /*System.out.println("出现了异常");
            System.out.println(e.getMessage());
            System.out.println(e);
            System.out.println(e.toString());*/
            e.printStackTrace();

        }

        System.out.println("结束");
    }
}
```

## 小结

略

# 知识点--finally 代码块

## 目标

- 掌握finally代码块的格式和执行流程

## 路径

- finally代码块的概述
- finally代码块的语法格式
- 案例演示

## 讲解

### finally代码块的概述

**finally**：有一些特定的代码无论异常是否发生，都需要执行。另外，因为异常会引发程序跳转，导致有些语句执行不到。而finally就是解决这个问题的，在finally代码块中存放的代码都是一定会被执行的。

### finally代码块的语法格式

```
try{
    可能会出现异常的代码

}catch(异常的类型 变量名){
    处理异常的代码或者打印异常的信息
}finally{
    无论异常是否发生,都会执行这里的代码(正常情况,都会执行finally中的代码,一般用来释放资源)
}

执行步骤:
 1.首先执行try中的代码,如果try中的代码出现了异常,那么就直接执行catch()里面的代码,执行完后会执行finally中的代码,然后程序继续往下执行
 2.如果try中的代码没有出现异常,那么就不会执行catch()里面的代码,但是还是会执行finally中的代码,然后程序继续往下执行
```

> 注意:finally不能单独使用。

## 案例演示

```
package com.itheima.demo10_finally代码块;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 15:09
 */
public class Test {
    public static void main(String[] args) {
        /*
            finally 代码块:
                概述:在finally代码块中存放的代码都是一定会被执行的
                格式:
                    try{
                        可能会出现异常的代码

                    }catch(异常的类型 变量名){
                        处理异常的代码或者打印异常的信息
                    }finally{
                        无论异常是否发生,都会执行这里的代码(正常情况,都会执行finally中的代码,一般用来释放资源)
                    }

                执行步骤:
                 1.首先执行try中的代码,如果try中的代码出现了异常,那么就直接执行catch()里面的代码,执行完后会执行finally中的代码,然后程序继续往下执行
                 2.如果try中的代码没有出现异常,那么就不会执行catch()里面的代码,但是还是会执行finally中的代码,然后程序继续往下执行
         */
        System.out.println("开始");
        /*try {
            System.out.println(1/0);
        }catch (ArithmeticException e){
            System.out.println("catch 出现了异常");
            return;
            //System.exit(0);
        }finally {
            System.out.println("finally 无论是否发生异常都会执行");
        }*/
```

```
        System.out.println("================================");
        try {
            System.out.println(1/1);// 1
            return;
        }catch (ArithmeticException e){
            System.out.println("catch 出现了异常");
        }finally {
            System.out.println("finally 无论是否发生异常都会执行");
        }
        System.out.println("结束");
    }
}
```

> 当只有在try或者catch中调用退出JVM的相关方法,此时finally才不会执行,否则finally永远会执行。


## 小结

略

# 扩展--finally经典面试题

```
package com.itheima.demo11_finally经典面试题;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 15:22
```

```
 */
public class Test {
    public static void main(String[] args) {
        System.out.println(method1());// 30
        System.out.println(method2());// 20
    }

    public static int method1() {
        int num = 10;
        try {
            System.out.println(1 / 0);
        } catch (ArithmeticException e) {
            num = 20;
            return num;
        } finally {
            num = 30;
            return num;
        }
    }

    public static int method2() {
        int num = 10;
        try {
            System.out.println(1 / 0);
        } catch (ArithmeticException e) {
            num = 20;
            // catch中的return会做2件事:1.先记录要返回的值,然后执行finally中的代码,2.最后把记录的值返回
            return num;// 记录要返回的值:20
        } finally {
            num = 30;
        }
        return num;
    }
}
```

# 知识点--异常注意事项

## 目标

- 理解异常注意事项

## 路径

- 异常注意事项

## 讲解

- 运行时异常被抛出可以不处理。即不捕获也不声明抛出。
- 如果父类的方法抛出了多个异常,子类覆盖(重写)父类方法时,只能抛出相同的异常或者是他的子集。

- 父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出
- 声明处理多个异常,可以直接声明这多个异常的父类异常
- 在try/catch后可以追加finally代码块，其中的代码一定会被执行，通常用于资源回收。
- 多个异常使用捕获又该如何处理呢?
    1. 多个异常分别处理。
    2. 多个异常一次捕获，多次处理。
    3. 多个异常一次捕获一次处理。
- 当多异常分别处理时，捕获处理，前边的类不能是后边类的父类

  一般我们是使用一次捕获多次处理方式，格式如下:

```
try{
      编写可能会出现异常的代码
}catch(异常类型A  e){  当try中出现A类型异常,就用该catch来捕获.
      处理异常的代码
      //记录日志/打印异常信息/继续抛出异常
}catch(异常类型B  e){  当try中出现B类型异常,就用该catch来捕获.
      处理异常的代码
      //记录日志/打印异常信息/继续抛出异常
}
```

> 注意:这种异常处理方式，要求多个catch中的异常不能相同，并且若catch中的多个异常之间有子父类异常的关系，那么子类异常要求在上面的catch处理，父类异常在下面的catch处理。

代码如下:

```
package com.itheima.demo12_异常注意事项;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.text.ParseException;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 15:49
 */
class Fu {
    public void show() throws ParseException, FileNotFoundException {
        // ...
    }

    public void run() {

    }
}

class Zi extends Fu {
    // 2.如果父类的方法抛出了多个异常,子类覆盖(重写)父类方法时,只能抛出相同的异常或者是他的子集。
   /* @Override
    public void show() throws ParseException, FileNotFoundException, IOException {
```

```
    }*/

   // 3.父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常,只能捕获处理
    @Override
    public void run() {
        try {
            throw new FileNotFoundException("");
        }catch (FileNotFoundException e){

        }
    }
}

public class Test {
    public static void main(String[] args) {
        /*
            异常注意事项:
                1.运行时异常被抛出可以不处理。即不捕获也不声明抛出。
                2.如果父类的方法抛出了多个异常,子类覆盖(重写)父类方法时,只能抛出相同的异常或者是他的子集。
                3.父类方法没有抛出异常，子类覆盖父类该方法时也不可抛出异常。此时子类产生该异常，只能捕获处理，不能声明抛出
                4.声明处理多个异常,可以直接声明这多个异常的父类异常
                5.在try/catch后可以追加finally代码块，其中的代码一定会被执行，通常用于资源回收。
                6.多个异常使用捕获又该如何处理呢？
                    1. 多个异常分别处理。
                    2. 多个异常一次捕获，多次处理。
                    3. 多个异常一次捕获一次处理。
                7.当多异常分别处理时，捕获处理，前边的类不能是后边类的父类
         */
    }

    // 1.运行时异常被抛出可以不处理。即不捕获也不声明抛出。
    public static void show1() {
        System.out.println(1 / 0);
    }

    // 4.声明处理多个异常,可以直接声明这多个异常的父类异常
    public static void show2(int num) throws Exception{
        if (num == 1){
            throw new FileNotFoundException("");
        }else{
            throw new ParseException("",1);
        }
    }

    // 多个异常使用捕获又该如何处理呢？
    // 1. 多个异常分别处理。
    public static void show3(int num) {
        if (num == 1){
            try {
                throw new FileNotFoundException("");
            } catch (FileNotFoundException e) {

            }
```

```
        }else{
            try {
                throw new ParseException("",1);
            } catch (ParseException e) {

            }
        }
    }
    // 2. 多个异常一次捕获，多次处理。
    public static void show4(int num) {
        try {
            if (num == 1){
                throw new FileNotFoundException("");
            }else{
                throw new ParseException("",1);
            }
        }catch (FileNotFoundException e){

        }catch (ParseException e){

        }
    }
    // 3. 多个异常一次捕获一次处理。
    public static void show5(int num) {
        try {
            if (num == 1){
                throw new FileNotFoundException("");
            }else{
                throw new ParseException("",1);
            }
        }catch (Exception e){

        }
    }
    // 7.当多异常分别处理时，捕获处理，前边的类不能是后边类的父类
    public static void show6(int num) {
        try {
            if (num == 1){
                throw new FileNotFoundException("");
            }else{
                throw new ParseException("",1);
            }
        }catch (Exception e){

        }/*catch (ParseException e){

        }*/
    }
}
```

## 小结

略

# 第六章 自定义异常

# 知识点-- 自定义异常

## 目标

- 能够自定义并使用异常类

## 路径

- 自定义异常概述
- 自定义异常练习

## 讲解

### 自定义异常概述

**为什么需要自定义异常类:**

我们说了Java中不同的异常类,分别表示着某一种具体的异常情况,那么在开发中总是有些异常情况是SUN没有定义好的,例如年龄负数问题,考试成绩负数问题.这些异常在JDK中没有定义过,此时我们根据自己业务的异常情况来定义异常类。

**什么是自定义异常类:**

在开发中根据自己业务的异常情况来定义异常类.

自定义一个业务逻辑异常: **RegisterException**。一个注册异常类。

**异常类如何定义:**

1. 自定义一个编译期异常: 自定义类 并继承于`java.lang.Exception`。
2. 自定义一个运行时期的异常类:自定义类 并继承于`java.lang.RuntimeException`。

```
package com.itheima.demo13_自定义异常;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:06
 */
// 编译异常
public class MyException1 extends Exception {

    public MyException1() {
    }

    public MyException1(String message) {
        super(message);
    }
}


package com.itheima.demo13_自定义异常;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:08
 */
// 运行异常
public class MyException2 extends RuntimeException {
```

```
    public MyException2() {
    }

    public MyException2(String message) {
        super(message);
    }
}
package com.itheima.demo13_自定义异常;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:05
 */
public class Test {
    public static void main(String[] args)  {
        /*
            异常类如何定义:
                1. 自定义一个编译期异常: 自定义类 并继承于java.lang.Exception。
                2. 自定义一个运行时期的异常类:自定义类 并继承于
java.lang.RuntimeException。

         */
        //throw new MyException1("自定义异常1");

        throw new MyException2("自定义异常2");
    }
}
```

## 自定义异常的练习

要求：我们模拟注册操作，如果用户名已存在，则抛出异常并提示：亲，该用户名已经被注册。

首先定义一个注册异常类RegisterException：

```
// 业务逻辑异常
public class RegisterException extends Exception {
    /**
     * 空参构造
     */
    public RegisterException() {
    }

    /**
     *
     * @param message 表示异常提示
     */
    public RegisterException(String message) {
        super(message);
    }
}
```

模拟登陆操作，使用数组模拟数据库中存储的数据，并提供当前注册账号是否存在方法用于判断。

```
package com.itheima.demo14_自定义异常的练习;
```

```java
import java.util.Scanner;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:10
 */
public class Test {
    public static void main(String[] args) {
        // 需求:我们模拟注册操作，如果用户名已存在，则抛出异常并提示：亲，该用户名已经被注册。
        // 1.定义一个数组,存储一些已知用户名
        String[] names = {"jack", "rose", "jim", "tom"};

        // 2.用户输入要注册的用户名
        Scanner sc = new Scanner(System.in);
        System.out.println("请输入一个用户名:");
        String name = sc.next();

        // 3.循环遍历数组
        for (String s : names) {
            // 4.在循环中,判断用户输入的用户名和已知的用户名是否相同
            if (s.equals(name)) {
                // 5.如果相同,就抛出异常对象
                try {
                    throw new RegisterException("亲，该用户名已经被注册。");
                } catch (RegisterException e) {
                    System.out.println(e.getMessage());
                    return;
                }
            }
        }

        // 6.如果不相同,就显示提示信息
        System.out.println("亲,恭喜您注册成功!");
    }
}
```

## 小结

略

# 第七章 多线程

我们在之前，学习的程序在没有跳转语句的前提下，都是由上至下依次执行，那现在想要设计一个程序，边打游戏边听歌，怎么设计?

要解决上述问题,咱们得使用多进程或者多线程来解决.

## 知识点--并发与并行

### 目标

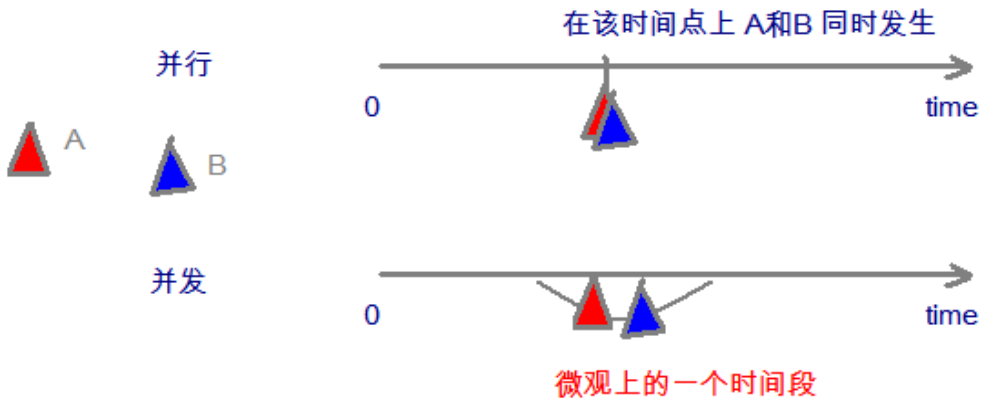
- 能够理解什么是并发和并行

## 路径

- 并行的概述
- 并发的概述

## 讲解

- **并行**：指两个或多个事件在**同一时刻**发生（同时执行）。
- **并发**：指两个或多个事件在**同一个时间段内**发生(交替执行)。




在操作系统中，安装了多个程序，并发指的是在一段时间内宏观上有多个程序同时运行，这在单 CPU 系统中，每一时刻只能有一道程序执行，即微观上这些程序是分时的交替运行，只不过是给人的感觉是同时运行，那是因为分时交替运行的时间是非常短的。

而在多个 CPU 系统中，则这些可以并发执行的程序便可以分配到多个处理器上（CPU），实现多任务并行执行，即利用每个处理器来处理一个可以并发执行的程序，这样多个程序便可以同时执行。目前电脑市场上说的多核 CPU，便是多核处理器，核越多，并行处理的程序越多，能大大的提高电脑运行的效率。

> 注意：单核处理器的计算机肯定是不能并行的处理多个任务的，只能是多个任务在单个CPU上并发运行。同理,线程也是一样的，从宏观角度上理解线程是并行运行的，但是从微观角度上分析却是串行运行的，即一个线程一个线程的去运行，当系统只有一个CPU时，线程会以某种顺序执行多个线程，我们把这种情况称之为线程调度。

## 小结

略

# 知识点-- 线程与进程

## 目标

- 能够理解什么是线程与进程

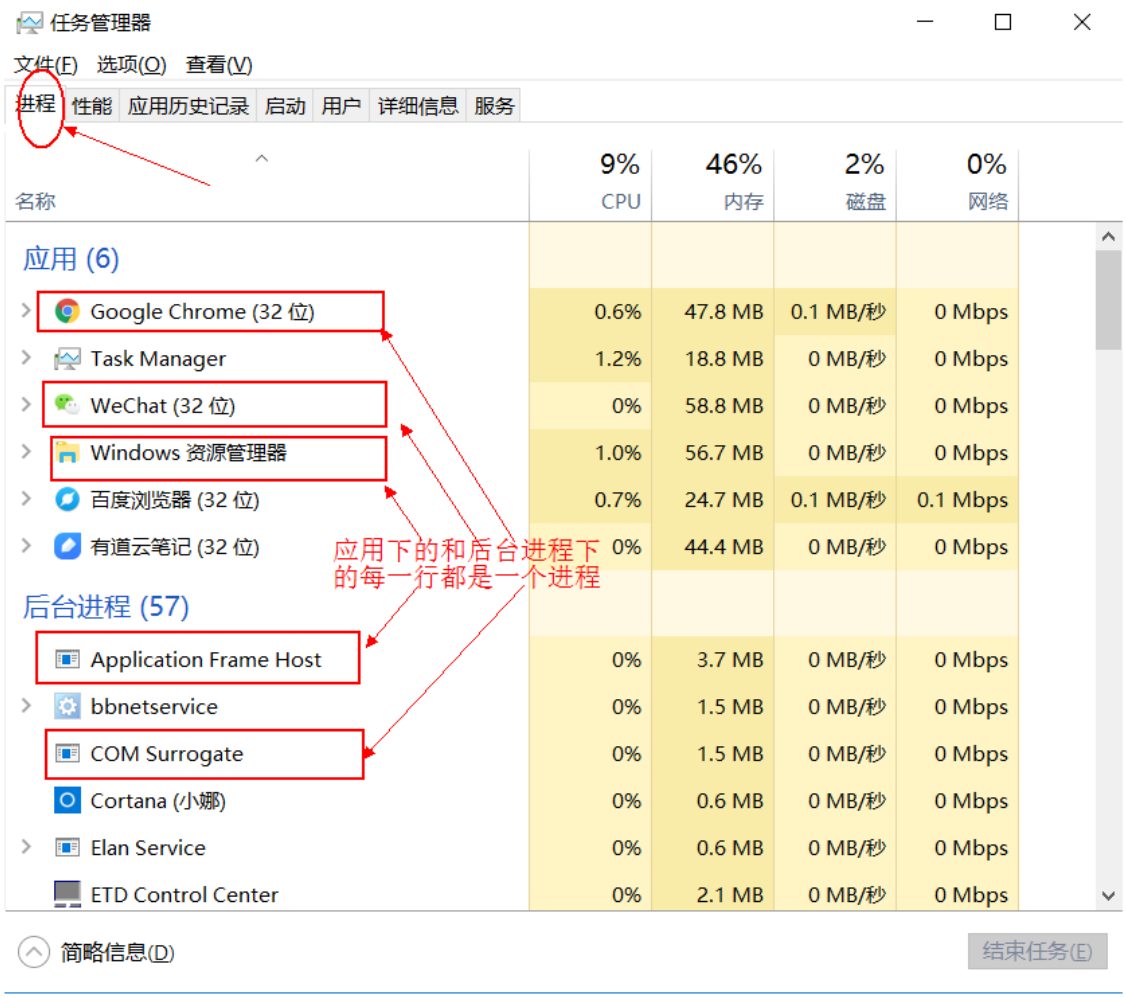
## 路径

- 线程的概述
- 进程的概述

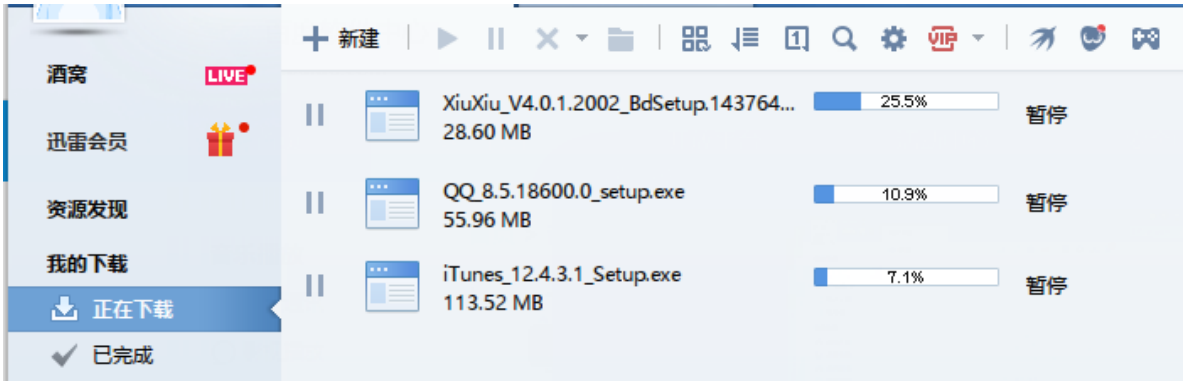
## 讲解

- **进程**：进程是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；
    - 进程:其实就是应用程序的可执行单元(.exe文件)
    - 每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；
- **线程**：是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少有一个线程。一个进程中是可以有多个线程的，这个应用程序也可以称之为多线程程序。
    - 线程:其实就是进程的可执行单元
    - 每条线程都有独立的内存空间,一个进程可以同时运行多个线程；
- 多线程并行: 多条线程在同一时刻同时执行
- **多线程并发:多条线程在同一时间段交替执行**
- **在java中线程的调度是:抢占式调度**
- 在java中只有多线程并发,没有多线程并行(高并发)

**进程**




**线程**




**进程与线程的区别**

- 进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。
- 线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

> **注意：**下面内容为了解知识点
>
> 1:因为一个进程中的多个线程是并发运行的，那么从微观角度看也是有先后顺序的，哪个线程执行完全取决于 CPU 的调度，程序员是干涉不了的。而这也就造成的多线程的随机性。
>
> 2:Java 程序的进程里面至少包含两个线程，主进程也就是 main()方法线程，另外一个是垃圾回收机制线程。每当使用 java 命令执行一个类时，实际上都会启动一个 JVM，每一个 JVM 实际上就是在操作系统中启动了一个线程，java 本身具备了垃圾的收集机制，所以在 Java 运行时至少会启动两个线程。
>
> 3:由于创建一个线程的开销比创建一个进程的开销小的多，那么我们在开发多任务运行的时候，通常考虑创建多线程，而不是创建多进程。

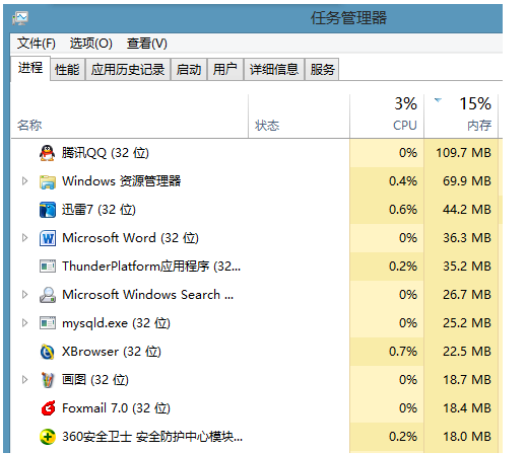
**线程调度:**

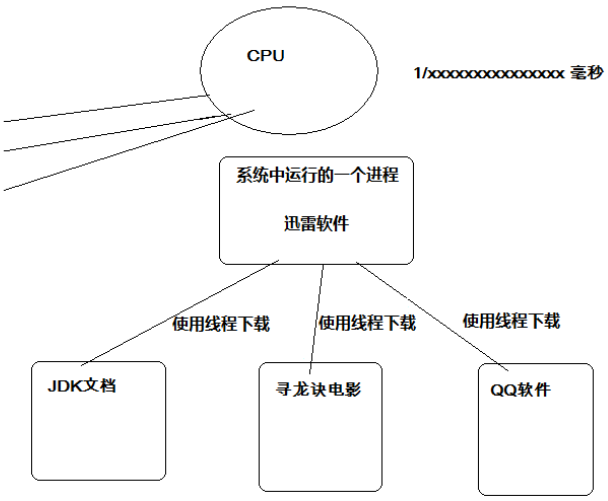
- 分时调度

  所有线程轮流使用 CPU 的使用权，平均分配每个线程占用 CPU 的时间。

- 抢占式调度

  优先让优先级高的线程使用 CPU，如果线程的优先级相同，那么会随机选择一个(线程随机性)，Java使用的为抢占式调度。




# 小结

略

# 知识点-- Thread类

## 目标

- 会使用Thread类的构造方法和常用方法

## 路径

- Thread类的构造方法
- Thread类的常用方法

## 讲解

### Thread类的概述

- 表示线程,也叫做线程类,创建该类的对象,就是创建线程对象(或者说创建线程)
- 线程的任务: 执行一段代码
- Runnable : 接口,线程任务接口

### Thread类的构造方法

线程开启我们需要用到了`java.lang.Thread`类，API中该类中定义了有关线程的一些方法，具体如下:

- `public Thread()`:分配一个新的线程对象,线程名称是默认生成的。
- `public Thread(String name)`:分配一个指定名字的新的线程对象。

- `public Thread(Runnable target)`:分配一个带有指定目标新的线程对象,线程名称是默认生成的。
- `public Thread(Runnable target,String name)`:分配一个带有指定目标新的线程对象并指定名字。

- 创建线程的方式有2种:
  - 一种是通过继承Thread类的方式
  - 一种是通过实现Runnable接口的方法

### Thread类的常用方法

- `public String getName()`:获取当前线程名称。
- `public void start()`:导致此线程开始执行; Java虚拟机调用此线程的run方法。
- `public void run()`:此线程要执行的任务在此处定义代码。
- `public static void sleep(long millis)`:使当前正在执行的线程以指定的毫秒数暂停（暂时停止执行）。
- `public static Thread currentThread()`:返回对当前正在执行的线程对象的引用。

翻阅API后得知创建线程的方式总共有两种，一种是继承Thread类方式，一种是实现Runnable接口方式，

## 小结

略

# 知识点--创建线程方式一_继承方式

## 目标

- 能够掌握创建线程方式一

## 路径

- 创建线程方式一_继承方式

## 讲解

Java使用`java.lang.Thread`类代表**线程**，所有的线程对象都必须是Thread类或其子类的实例。每个线程的作用是完成一定的任务，实际上就是执行一段程序流即一段顺序执行的代码。Java使用线程执行体来代表这段程序流。Java中通过继承Thread类来**创建**并**启动多线程**的步骤如下：

1. 定义Thread类的子类，并重写该类的run()方法，该run()方法的方法体就代表了线程需要完成的任务,因此把run()方法称为线程执行体。
2. 创建Thread子类的实例，即创建了线程对象
3. 调用线程对象的start()方法来启动该线程

代码如下：

测试类：

```
package com.itheima.demo15_创建线程方式一_继承方式;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:57
 */
public class Test {
    public static void main(String[] args) {
        /*
            补充: java程序至少有2条线程:一条为主线程,一条为垃圾回收线程
            创建线程方式一_继承方式:
                1.创建子类继承Thread类
                2.在子类中重写run方法,把线程需要执行的任务代码放在run方法中
                3.创建子类线程对象
                4.调用start()方法启动线程,执行任务代码
         */
        // 创建子类线程对象
        MyThread mt1 = new MyThread();
        // 调用start()方法启动线程,执行任务代码
        mt1.start();

        for (int j = 0; j < 100; j++) {
            System.out.println("主线程 第"+(j+1)+"次循环");
        }
    }
}
```

自定义线程类：

```
package com.itheima.demo15_创建线程方式一_继承方式;
```

```java
/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 16:58
 */
public class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println("子线程 第"+(i+1)+"次循环");
        }
    }
}
```

## 小结

```
创建线程方式一_继承方式:
    1.创建子类继承Thread类
    2.在子类中重写run方法,把线程需要执行的任务代码放在run方法中
    3.创建子类线程对象
    4.调用start()方法启动线程,执行任务代码
```

# 知识点--创建线程的方式二_实现方式

## 目标

- 能够掌握创建线程方式二

## 路径

- 创建线程方式二_实现方式

## 讲解

采用`java.lang.Runnable`也是非常常见的一种，我们只需要重写run方法即可。

步骤如下：

1. 定义Runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。
2. 创建Runnable实现类的实例，并以此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。
3. 调用线程对象的start()方法来启动线程。

代码如下：

```java
package com.itheima.demo16_创建线程的方式二_实现方式;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 17:09
 */
public class MyRunnable implements Runnable {
```

```
    @Override
    public void run() {
        // 线程需要执行的任务代码
        for (int i = 0; i < 100; i++) {
            System.out.println("子线程 第"+(i+1)+"次循环");
        }
    }
}
```

```
package com.itheima.demo16_创建线程的方式二_实现方式;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 17:07
 */
public class Test {
    public static void main(String[] args) {
        /*
            创建线程的方式二_实现方式:
                1.创建实现类实现Runnable接口
                2.在实现类中重写run方法,把线程需要执行的任务代码放入run方法中
                3.创建实现类对象
                4.创建Thread线程对象,并传入Runnable接口的实现类对象
                5.调用start()方法启动线程,执行任务
         */
        //创建实现类对象
        MyRunnable mr = new MyRunnable();

        //创建Thread线程对象,并传入Runnable接口的实现类对象
        Thread t1 = new Thread(mr);

        //调用start()方法启动线程,执行任务
        t1.start();


        for (int j = 0; j < 100; j++) {
            System.out.println("主线程 第"+(j+1)+"次循环");
        }
    }
}
```

通过实现Runnable接口，使得该类有了多线程类的特征。run()方法是多线程程序的一个执行目标。所有的多线程代码都在run方法里面。Thread类实际上也是实现了Runnable接口的类。

在启动的多线程的时候，需要先通过Thread类的构造方法Thread(Runnable target) 构造出对象，然后调用Thread对象的start()方法来运行多线程代码。

实际上所有的多线程代码都是通过运行Thread的start()方法来运行的。因此，不管是继承Thread类还是实现Runnable接口来实现多线程，最终还是通过Thread的对象的API来控制线程的，熟悉Thread类的API是进行多线程编程的基础。

> tips:Runnable对象仅仅作为Thread对象的target，Runnable实现类里包含的run()方法仅作为线程执行体。而实际的线程对象依然是Thread实例，只是该Thread线程负责执行其target的run()方法。

## 小结

略

# 知识点--匿名内部类方式

## 目标

- 能够掌握匿名内部类方式

## 路径

- 创建匿名内部类方式

## 讲解

使用线程的内匿名内部类方式，可以方便的实现每个线程执行不同的线程任务操作。

使用匿名内部类的方式实现Runnable接口，重新Runnable接口中的run方法：

```
package com.itheima.demo17_创建线程的方式三_匿名内部类方式;

/**
 * @Author: pengzhilin
 * @Date: 2020/9/16 17:14
 */
public class Test {
    public static void main(String[] args) {
        /*
            创建线程的方式三_匿名内部类方式:
                1.创建Runnable的匿名内部类
                2.在匿名内部类中重写run方法,把线程需要执行的任务代码放入run方法中
                3.创建Thread线程对象,并传入Runnable的匿名内部类
                4.调用start()方法启动线程,执行任务

            注意:
                1.主线程一定会等子线程全部执行完毕才会结束主线程
                2.子线程任务代码执行完毕,线程就会销毁
         */

        // 创建Thread线程对象,并传入Runnable的匿名内部类
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                // 线程需要执行的任务代码
                for (int i = 0; i < 100; i++) {
                    System.out.println("子线程 第"+(i+1)+"次循环");
                }
            }
        });

        // 调用start()方法启动线程,执行任务
        t.start();

        for (int j = 0; j < 100; j++) {
```

```
            System.out.println("主线程 第"+(j+1)+"次循环");
        }

    }
}
```

## 小结

略

# 总结---Thread和Runnable的区别

如果一个类继承Thread，则不适合资源共享。但是如果实现了Runable接口的话，则很容易的实现资源共享。

**总结:**

**实现Runnable接口比继承Thread类所具有的优势:**

1. 可以避免java中的单继承的局限性。
2. 增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和线程独立。
3. 适合多个相同的程序代码的线程去共享同一个资源。
4. 线程池只能放入实现Runable或Callable类线程，不能直接放入继承Thread的类。

# 总结

```
必须练习:
    1.处理异常---->alt+回车-->选择throws\try...catch
    2.创建并启动线程的三种方式-->继承\实现\匿名内部类
    3.排序\查找
    4.预习

- 能够理解冒泡排序的执行原理
    1.冒泡排序---->相邻的两个元素进行比较,较大的数据放在后面
      for(int i = 0;i<数组长度-1;i++){
            for(int j = 0;j<数组长度-1-i;j++){
              // 索引为j和索引为j+1的元素进行比较
            }
      }

- 能够理解选择排序的执行原理
    2.选择排序---->选择某个元素(从前往后选择),与该元素后面的所有元素一一进行比较,较大的放在后面
        for(int i = 0;i<数组长度-1;i++){
            for(int j = i+1;j<数组长度;j++){
              // 索引为i和索引为j的元素进行比较
            }
        }

- 能够理解二分查找的执行原理
    二分查找:
        原理: 每一次都去获取数组的中间索引所对应的元素，然后和要查找的元素进行比对，如果相同就返回索引;
如果不相同，就比较中间元素和要查找的元素的值;
```

如果中间元素的值大于要查找的元素，说明要查找的元素在左侧，那么就从左侧按照上述思想继续查询(忽略右侧数据)；
如果中间元素的值小于要查找的元素，说明要查找的元素在右侧，那么就从右侧按照上述思想继续查询(忽略左侧数据)；
二分查找对数组是有要求的,数组必须已经排好序

- 能够辨别程序中异常和错误的区别
  Error:表示错误,程序员是无法通过代码进行纠正使得程序继续往下执行,只能事先避免
  Exception:表示异常,程序员是可以通过代码进行纠正使得程序继续往下执行

- 说出异常的分类
  编译异常:在编译期间出现的异常,如果编译期间不处理,无法通过编译,程序就无法执行
  运行异常:在运行期间出现的异常,如果编译期间不处理,可以通过编译,程序可以执行

- 列举出常见的三个运行期异常
  NullPointerException
  ArrayIndexOutOfBoundsException
  ArithmeticException
  ClassCastException
  ...

- 能够使用try...catch关键字处理异常
  格式:

```
try{
    编写可能会出现异常的代码
}catch(异常类型  变量名){
    处理异常的代码
    //记录日志/打印异常信息
}
```

  执行步骤:
  1.首先执行try中的代码,如果try中的代码出现了异常,那么就直接执行catch()里面的代码,执行完后,程序继续往下执行
  2.如果try中的代码没有出现异常,那么就不会执行catch()里面的代码,而是继续往下执行

- 能够使用throws关键字处理异常
  格式:

```
修饰符 返回值类型 方法名(形参列表) throws 异常类名1,异常类名2…{  // 可以抛出一个,也可以多个

}
```

  特点: 声明处理异常,处理完后,如果程序运行的时候出现异常,程序还是无法继续往下执行;
  如果程序运行的时候不出现异常,程序就可以继续往下执行

  使用场景: 声明处理异常一般处理运行的时候不会出现异常的编译异常

- 能够自定义并使用异常类
  自定义编译异常:创建一个类继承Exception
  自定义运行异常:创建一个类继承RuntimeException

- 说出进程和线程的概念
  进程: 其实就是应用程序的可执行单元(.exe)
  线程: 其实就是进程的可执行单元
  一个应用程序可以有多个进程,一个进程可以有多条线程

- 能够理解并发与并行的区别
  并发: 多个事情在同一时间段交替发生

并行；多个事情在同一时刻同时发生

多线程并发：多条线程同时请求,但交替执行

- 能够描述Java中多线程运行原理
  抢占式

- 能够使用继承类的方式创建多线程
  1.创建子类继承Thread类
  2.在子类中重写run方法,把线程需要执行的任务代码写入run方法中
  3.创建子类线程对象
  4.调用start()方法,启动线程,执行任务

- 能够使用实现接口的方式创建多线程
  1.创建实现类实现Runnable接口
  2.在实现类中重写run方法,把线程需要执行的任务代码写入run方法中
  3.创建实现类对象
  4.创建Thread线程对象,并传入实现类对象
  5.调用start()方法,启动线程,执行任务

- 能够说出实现接口方式的好处
  1.解决单继承的弊端
  2.线程和任务代码是独立分块的,启动解耦操作
  3.任务可以被多条线程共享
  4.线程池中的线程都是通过实现Runnable或者Callable接口方式的线程,不能放入继承Thread类方式的线程