单例设计模式

```java
public class SingletonTest {
    public static void main(String[] args) {
        ESingleton e = ESingleton.getInstance();
        ESingleton e1 = ESingleton.getInstance();
        System.out.println(e == e1);
        System.out.println(e.hashCode());
        System.out.println(e1.hashCode());

        LSingleton l1 = LSingleton.getInstance();
        LSingleton l2 = LSingleton.getInstance();
        System.out.println(l1 == l2);
        System.out.println(l1.hashCode());
        System.out.println(l2.hashCode());

        EnumSingleton instance = EnumSingleton.INSTANCE;
        EnumSingleton instance1 = EnumSingleton.INSTANCE;
        System.out.println(instance == instance1);
        System.out.println(instance.hashCode());
        System.out.println(instance1.hashCode());
    }
}

// 饿汉式,两种
class ESingleton {

    // 静态变量
    //    private final static  ESingleton s = new ESingleton();
    //    private ESingleton() {}
    //    public static ESingleton getInstance() {
    //        return s;
    //    }

    // 静态代码块
    private ESingleton() {
    }

    private static ESingleton s;

    static {
        s = new ESingleton();
    }

    public static ESingleton getInstance() {
        return s;
    }


}

// 懒汉式
class LSingleton {

    private LSingleton() {}
    // 线程不安全
```

```
    //  private static LSingleton s;
    //    public static LSingleton getInstance() {
    //        if (s == null) {
    //            s = new LSingleton();
    //        }
    //        return s;
    //    }

    // 双重查询
    //    private static volatile Singleton s;
    //    public static LSingleton getInstance() {
    //        if (s == null) {
    //            synchronized (LSingleton.class) {
    //                if (s == null) {
    //                    s = new LSingleton();
    //                }
    //            }
    //        }
    //        return s;
    //    }

    // 内部类
    private static class Inner{
        private static final LSingleton l = new LSingleton();
    }

    public static LSingleton getInstance(){
        return Inner.l;
    }
}

// 枚举类
enum EnumSingleton{
    INSTANCE;
}
```

代码块

Father类

```
public class Father {
    private int i = test();
    private static int j = method();

    static{
        System.out.println("1.Father的静态代码块");
    }

    Father(){
        System.out.println("2.Father的构造方法");
    }

    {
        System.out.println("3.Father的代码块");
    }
    public int test(){
        System.out.println("4.Father的普通变量");
        return 1;
```

```
    }

    public static int method(){
        System.out.println("5.Father的静态变量");
        return 1;
    }
}

```

Son类

```
public class Son extends Father {
    private int i = test();                    // 重写Father中test()方法,多态,方法运行看子类
    private static int j = method();

    static{
        System.out.println("6.Son的静态代码块");
    }

    Son(){
        System.out.println("7.Son的构造方法");
    }

    {
        System.out.println("8.Son的代码块");
    }
    public int test(){
        System.out.println("9.Son的普通变量");
        return 1;
    }

    public static int method(){
        System.out.println("10.Son的静态变量");
        return 1;
    }

    public static void main(String[] args){
        Son s1 = new Son();
        System.out.println();
        Son s2 = new Son();
    }
}

```

1.***父类静态成员变量*** 和 ***父类静态代码块*** 同级,谁在前先执行谁
2.***子类静态成员变量*** 和 ***子类静态代码块*** 同级,谁在前先执行谁
3.***父类普通成员变量*** 和 ***父类普通代码块*** 同级,谁在前先执行谁
4.父类构造方法
5.***子类普通成员变量*** 和 ***子类普通代码块*** 同级,谁在前先执行谁
6.子类构造方法

> 5.Father的静态变量
> 1.Father的静态代码块
> 10.Son的静态变量
> 6.Son的静态代码块
> 9.Son的普通变量

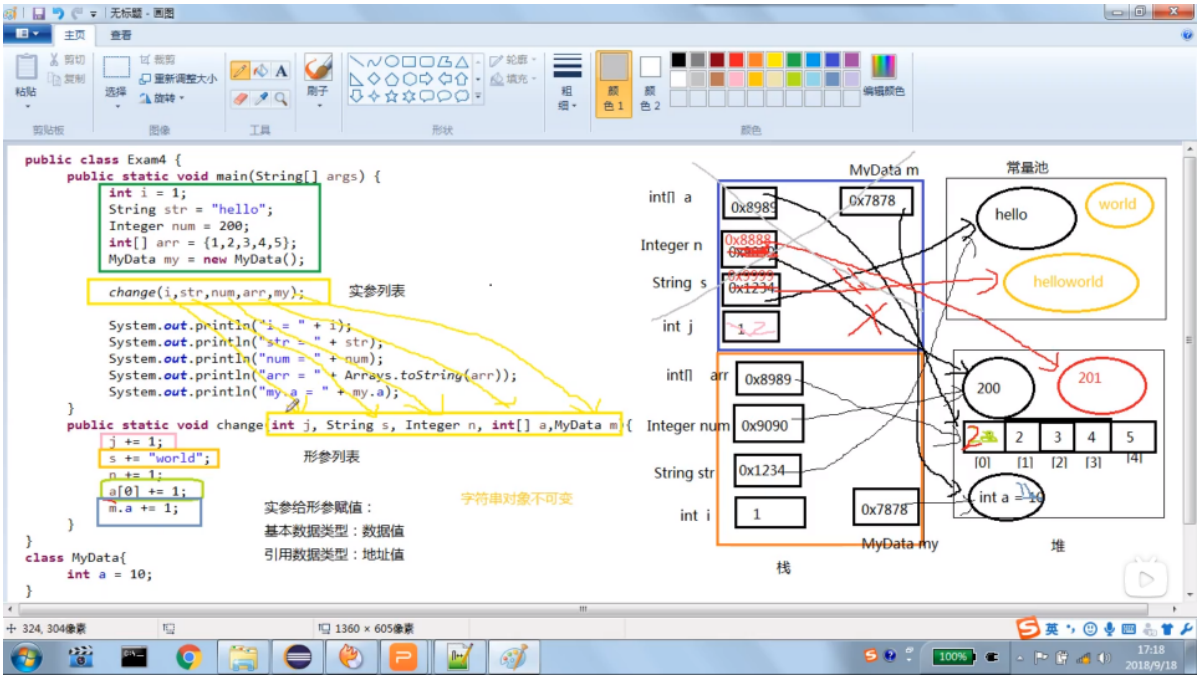
```
3.Father的代码块
2.Father的构造方法
9.Son的普通变量
8.Son的代码块
7.Son的构造方法

9.Son的普通变量
3.Father的代码块
2.Father的构造方法
9.Son的普通变量
8.Son的代码块
7.Son的构造方法
```

参数传递

```java
public class Test2 {
    public static void main(String[] args) {
        int i = 1;
        String str = "hello";
        Integer num = 2;
        int[] arr = {1, 2, 3, 4, 5};
        MyData my = new MyData();

        System.out.println("int = " + i);
        System.out.println("String = " + str);
        System.out.println("Integer = " + num);
        System.out.println("数组 = " + Arrays.toString(arr));
        System.out.println("自定义类的属性 = " + my.a);

        System.out.println("--------------");

        change(i, str, num, arr, my);

        System.out.println("int = " + i);
        System.out.println("String = " + str);
        System.out.println("Integer = " + num);
        System.out.println("数组 = " + Arrays.toString(arr));
        System.out.println("自定义类的属性 = " + my.a);
    }

    private static void change(int j, String s, Integer n, int[] a, MyData m) {
        j += 1;
        s += "world";
        n += 1;
        a[0] += 1;
        m.a += 1;
    }
}

class MyData {
    int a = 10;
}
```

```
int = 1
String = hello
Integer = 2
数组 = [1, 2, 3, 4, 5]
```

> 自定义类的属性 = 10
>
> ---
>
> int = 1
> String = hello
> Integer = 2
> 数组 = [2, 2, 3, 4, 5]
> 自定义类的属性 = 11



方法的参数传递机制

①形参是基本数据类型
　　传递数据值
②实参是引用数据类型
　　传递地址值
　　特殊的类型:String,包装类等对象不可变性

冒泡排序

两个相邻位置比较,如果前面的元素比后面的元素大,就换位置,大的在后面
每次都是以 第一个和第二个进行比较 开始
每次比较后,确定最后一个位置的数字

```
1  public class BubbleSort {
2      public static void main(String[] args) {
3          int[] arr1 = {12,58,69,10,31};
4          System.out.print("原始数组：");
5          print(arr1);                                    // 原始数组：12, 58, 69, 10,
   31
6          bubbleSort(arr1);
7          System.out.print("冒泡排序：");
8          print(arr1);                                    // 冒泡排序：10, 12, 31, 58,
   69
9      }
10
11     /*
12      *
13      *       冒泡排序
14      *       1，返回值类型，void
```

```
     *       2，参数列表，int[] arr
     *                          i和j都是从0开始
j的取值范围
     *       第一次：arr[0]与arr[1]，arr[1]与arr[2]，arr[2]与arr[3]，arr[3]与
arr[4]比较四次
             第二次：arr[0]与arr[1]，arr[1]与arr[2]，arr[2]与arr[3]比较三次
             第三次：arr[0]与arr[1]，arr[1]与arr[2]比较两次
i的取值范围    第四次：arr[0]与arr[1]比较一次
     *
     *       总长度length是五次，i是次数，j是比较次数
     *       j = length - i
     *
     *       i:0~4
     *       j:第一次循环是0~4，第二次循环是0~3，第三次是0~2，第四次是0~1
     *
     */

    public static void bubbleSort(int[] arr) {
        /*
         *       外循环控制次数
         *       内循环控制一次的比较次数
         *
         * */

        for (int i = 0; i < arr.length - 1; i++) {                    // 外循环只
需要比较arr.length-1次就可以了
            for (int j = 0; j < arr.length - 1 - i; j++) {            // - 1 为了
if语句j+1防止索引越界,-i为了提高效率(-i不要也不错)
                if (arr[j] > arr[j + 1]) {                            // 第j+1个和
第j+2个比较
                    /*int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;*/
                    swap(arr,j,j+1);
                }
            }
        }
    }


    /*
     *  打印数组
     *  1，返回值类型，void
     *  2，参数列表int[] arr
     *
     */
    public static void print(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            if(i == arr.length - 1) {
                System.out.print(arr[i]);
            }else {
                System.out.print(arr[i] + ", ");
            }
        }
    }

    /*
     *
     *       换位操作
```

```
     *       1，返回值类型，void
     *       2，参数列表int[] arr, int i ,int j
     *
     *       如果某个方法，只针对本类使用，不想让其他类使用就可以定义成私有的
     */
    private static void swap(int[] arr, int i ,int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}
```

选择排序

用一个索引位置上的元素,依次与其他索引位置上的元素比较,小的放在前面,大的放在后面
每次以 下一个数字与后面的数字进行比较 开始
每次比较,确定第一个位置,下次由后一位(下一个)进行向后比较

```
public class SelectSort {
    public static void main(String[] args) {
        int[] arr2 = {22,99,66,11,33};
        System.out.print("原始数组：");
        print(arr2);                                        // 原始数组：22, 99, 66, 11, 33
        selectSort(arr2);
        System.out.print("选择排序：");
        print(arr2);                                        // 选择排序：11, 22, 33, 66, 99
    }

    /*
     *
     *       选择排序
     *       1，返回值类型，void
     *       2，参数列表，int[] arr
     *
     *       第一次：arr[0]与arr[1], arr[0]与arr[2], arr[0]与arr[3], arr[0]与arr[4]比较四次
             第二次：arr[1]与arr[2], arr[1]与arr[3], arr[2]与arr[4]比较三次
             第三次：arr[2]与arr[3], arr[1]与arr[4]比较两次
             第四次：arr[3]与arr[4]比较一次
                                i          j              j的取值范围
             第一次：          arr[0]分别与arr[1-4]比较，比较四次
             第二次：          arr[1]分别与arr[2-4]比较，比较三次
             第三次：          arr[2]分别与arr[3-4]比较，比较两次
i的取值范围   第四次：          arr[3]与arr[4]比较，比较一次
     *
     */

    public static void selectSort(int[] arr) {
        /*
         *       外循环控制比较的数和次数
         *       内循环控制被比较数和一次的比较次数
         *
         *
         * */
        for (int i = 0; i < arr.length - 1; i++) {          //i:0~4
```

```
            for (int j = i + 1; j < arr.length; j++) {      //j:第一次循环是1~4，第二次循环是2~4，第三次是3~4，第四次是4
                if(arr[i] > arr[j]) {
                    /*int temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;*/
                    swap(arr,i,j);
                }
            }
        }
    }

    /*
     *  打印数组
     *  1，返回值类型，void
     *  2，参数列表int[] arr
     *
     */
    public static void print(int[] arr) {
        for (int i = 0; i < arr.length; i++) {
            if(i == arr.length - 1) {
                System.out.print(arr[i]);
            }else {
                System.out.print(arr[i] + ", ");
            }
        }

    }

    /*
     *
     *      换位操作
     *      1，返回值类型，void
     *      2，参数列表int[] arr, int i ,int j
     *
     *      如果某个方法，只针对本类使用，不想让其他类使用就可以定义成私有的
     */
    private static void swap(int[] arr, int i ,int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

}
```

二分查找

```
/**
 *

 *      B:注意事项
 *          如果数组无序，就不能使用二分查找
 *          因为如果你排序了，但是你排序的时候已经改变了我最原始的元素索引
 *
 *
 */
```

```java
public class Thirteen_Erfen {
    public static void main(String[] args) {
        int[] arr = { 11, 22, 33, 44, 55, 66, 77, 88 };
        System.out.println(getIndex(arr, 22));                  // 1
        System.out.println(getIndex(arr, 77));                  // 6
        System.out.println(getIndex(arr, 56));                  // -1
    }

    /*
     *
     *      二分查找
     *      1,返回值类型,int
     *      2,参数列表int[] arr ,int value
     *
     */
    public static int getIndex(int[] arr,int value) {
        int min = 0;
        int max = arr.length-1;
        int mid = (min + max) / 2;

        while(arr[mid] != value) {                              //当中间值不等于要找的值，就开始循环查找
            if(arr[mid] < value) {                              //当中间值小于要找的值，
                min = mid + 1;                                  //最小的索引改变
            }else if(arr[mid] > value) {                        //当中间值大于要找的值，
                max = mid - 1;                                  //最大的索引改变
            }

            mid = (min + max) / 2;                              //无论最大还是最小改变，中间索引都会随之改变

            if(min > max) {                                     //如果最小索引大于最大索引
                return -1;
            }
        }

        return mid;

    }
}
```

局部变量和全局变量

```java
public class Field {
    static int s;
    int i;
    int j;

    {
        int i = 1;
        i++;
        j++;
        s++;
    }

```

```
    public void test(int j) {
        j++;
        i++;
        s++;
    }

    public static void main(String[] args) {
        Field obj1 = new Field();
        Field obj2 = new Field();
        obj1.test(10);
        obj1.test(20);
        obj2.test(30);
        System.out.println(obj1.i + "," + obj1.j + "," + obj1.s);       //2,1,5
        System.out.println(obj2.i + "," + obj2.j + "," + obj2.s);       //1,1,5

    }
}
```

插入排序

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 *
 * 将比较的数值暂存temp,进行和前面的数据比较,
 *      大的话不动,
 *      小的话 (1)数值依次向后移动,(这时数组中没有该数值,且数组中有重复数值),直到合适的位置
 *               (2)将合适的位置插入temp值
 *
 * 将前面的看成有序数组,后面的看成无序数组,每次循环提取无序数组的第一个数据X,
 * 让它和有序数组的最后一个数据Y进行比较,[Z,Y]
 * 如果X>Y,那么X就插入有序数组的最后一个位置[Z,Y,X]
 * 如果X<Y,那么X再和Y的前一个数据Z进行比较,
 * Y后移,让有序数组的最后一个数据也为Y,这时有两个Y,原来位置的Y2,后移位置后的Y1[Z,Y2,Y1]
 *      如果X>Z,那么X就插入Z的后面,让Y2=X,Y1为最后一个最大的数据 ,[Z,X,Y1]
 *      如果X<Z,那么X再和Y的前一个数据A进行比较,[A,Z,Y]
 *      Z后移,让Y2=Z,这时有两个Z,原来位置的Z2,后移位置后的Z1
 *          如果同上....
 *
 * 但其实还是就一个数组
 *
 */
public class InsertSort {

    public static void main(String[] args) {
        int[] arr = {101, 34, 119, 1, -1, 89,222};
        insertSort(arr); //调用插入排序算法
        System.out.println(Arrays.toString(arr));
    }

    //插入排序
    public static void insertSort(int[] arr) {
```

```
        int insertVal = 0;
        int insertIndex = 0;
        //使用for循环来把代码简化
        for(int i = 1; i < arr.length; i++) {
            //定义待插入的数
            insertVal = arr[i];
            insertIndex = i - 1; // 即arr[1]的前面这个数的下标,从有序数组的最后一个位置开始向前找

            // 给insertVal 找到插入的位置
            // 说明
            // 1. insertIndex >= 0 保证在给insertVal 找插入位置，不越界
            // 2. insertVal < arr[insertIndex] 待插入的数，还没有找到插入位置,就向前找
            // 3. 就需要将 arr[insertIndex] 后移,让后一个数等于前一个数,腾出来前面的位置
            while (insertIndex >= 0 && insertVal < arr[insertIndex]) {
                arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
                insertIndex--;
            }
            // 当退出while循环时，说明插入的位置找到, insertIndex + 1
            // 举例：理解不了，我们一会 debug
            //这里我们判断是否需要赋值,如果insertIndex + 1 == i,相当于没有执行while语句,就是在有序数组最后一位插入
            if(insertIndex + 1 != i) {
                arr[insertIndex + 1] = insertVal;
            }
            //System.out.println("第"+i+"轮插入");
            //System.out.println(Arrays.toString(arr));
        }
    }
}
```

快速排序

第一趟排序过程如下：

[49, 38, 65, 97, 76, 13, 27, 49]

[49, 38, 65, 97, 76, 13, 27, 49]

[27, 38, 65, 97, 76, 13, 49, 49]

[27, 38, 65, 97, 76, 13, 49, 49]

[27, 38, 49, 97, 76, 13, 65, 49]

[27, 38, 13, 97, 76, 49, 65, 49]

[27, 38, 13, 49, 76, 97, 65, 49]

第一趟排序后：

[27, 38, 13]49[76, 97, 65, 49]

第二趟排序后：

[13]27[38]49[49, 65]76[97]

第三趟排序后：

13, 27, 38, 49, 49 [65] 76 , 97

最后的排序结果：

13, 27, 38, 49, 49 [65] 76 , 97

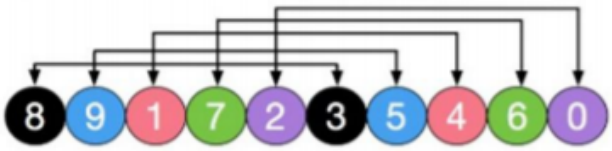
```java
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 *
 * 第二个方法
                1: 以数组的第一个为基数
                2: 循环,条件为 i < j
                2: 从后向前找比基数小的数,arr[j]
                3: 从前向后找比基数大的数,arr[i]
                4: 如果 i < j ,那么这个两数交换位置
                5: 一直循环,直到 i == j
                6: 因为第一个数一直没动,一直是除第一个数后面的数一直在交换位置
                7: 将第一个数 和 最后arr[i] 交换,这时数组 一个数 现在的位置 左边都是比这个数小的数,右边的都是比这个数大的数
                8: 然后左右递归,左递归(arr, low, i - 1),右递归(arr, i + 1, high)
```

```
 *
 */

public class QuickSort {

    public static void main(String[] args) {
        int[] arr = {0, 78, -567, -9, 88, 70, -1, 70, 4561};

        quickSort2(arr, 0, arr.length - 1);
        System.out.println("arr=" + Arrays.toString(arr));

    }

    public static void quickSort2(int[] arr, int low, int high) {
        int i, j, temp, t;
        if (low > high) {
            return;
        }
        i = low;
        j = high;
        //temp就是基准位
        temp = arr[low];

        while (i < j) {
            //先看右边，依次往左递减
            while (temp <= arr[j] && i < j) {
                j--;
            }
            //再看左边，依次往右递增
            while (temp >= arr[i] && i < j) {
                i++;
            }
            //如果满足条件则交换
            if (i < j) {
                t = arr[j];
                arr[j] = arr[i];
                arr[i] = t;
            }

        }
        // 这时 i == j
        //最后将基准为与i和j相等位置的数字交换
        arr[low] = arr[i];   // 将第一个位置的数,换成最后的 索引为i 的数值
        arr[i] = temp;       // temp存的是原来第一个数,将第一个数换到合适的地方
        //递归调用左半数组
        quickSort2(arr, low, i - 1);
        //递归调用右半数组
        quickSort2(arr, i + 1, high);
    }

}

```

希尔排序

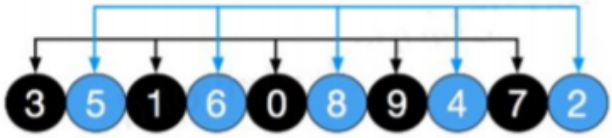
原始数组 以下数据元素颜色相同为一组

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



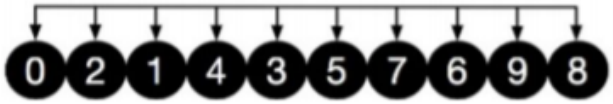
对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下

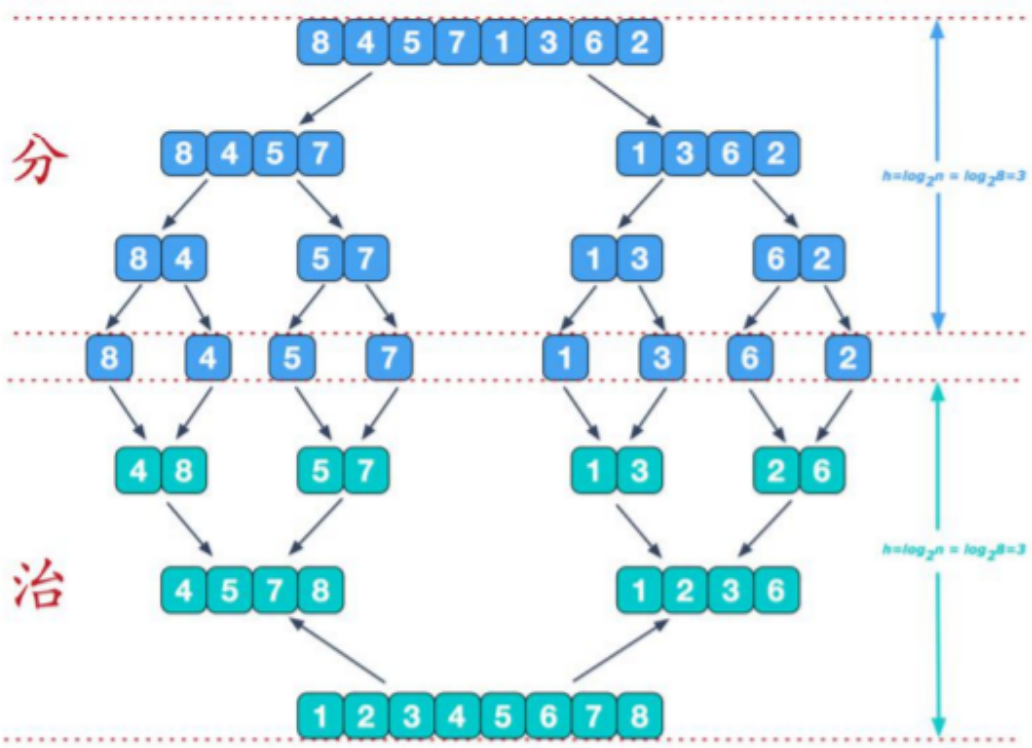



```java
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 *
 *
 * 希尔排序:
        交换
            1. 按照原数组长度的一半当为步长,原数组分组,每组两个元素,将每组进行组内排序
            2. 再将上一步的步长的一半为这一步的步长,原数组分组,每组元素变多,将每组进行组内排序
            3. 重复上一步

        移位(类似插入排序)
            1. 按照原数组长度的一半当为步长,原数组分组,
            2. 从第gap个元素，逐个对其所在的组进行直接插入排序
            3. 遍历数组
            4. 将该数值与该组的前一个数值进行比较
                (1) 如果大,不动
                (2) 如果小,那么前一个数值向后移动(组内移动,索引值±步长)
                (3) 重复上一步,直到该数值大于前一个数值,或者该数组前面不再有数值
            5. 将该数值插入该位置
 *
 */

public class ShellSort {

```

```java
    public static void main(String[] args) {
        int[] arr = {8, 9, 1, 7, 2, 3, 5, 4, 6, 0};
//        shellSort(arr); //交换式
        shellSort2(arr);//移位方式

        System.out.println(Arrays.toString(arr));
    }

    // 使用逐步推导的方式来编写希尔排序
    // 希尔排序时， 对有序序列在插入时采用交换法，
    // 思路(算法) ===> 代码
    public static void shellSort(int[] arr) {

        int temp = 0;
        int count = 0;
        // 根据前面的逐步分析，使用循环处理

        // 初始组数为数组的长度的一半,然后组数每次减半
        for (int gap = arr.length / 2; gap > 0; gap /= 2) {
            // arr[i]:后面的数;arr[j]:同一组第一个数,arr[j+gap]:同一组后一个数,arr[j]和arr[j + gap]比较
            // i++ 向后遍历数组,所有组依次进行排序,并不是一个组排序完进行下一个组
            for (int i = gap; i < arr.length; i++) {
                // 遍历各组中所有的元素(共gap组，每组有个元素)， 步长gap
                // 遍历i索引值对应数值的 组,j进行组内操作
                for (int j = i - gap; j >= 0; j -= gap) {
                    // 如果当前元素大于加上步长后的那个元素，说明交换
                    if (arr[j] > arr[j + gap]) {
                        temp = arr[j];
                        arr[j] = arr[j + gap];
                        arr[j + gap] = temp;
                    }
                }
            }
            //System.out.println("希尔排序第" + (++count) + "轮 =" + Arrays.toString(arr));
        }

    }

    //对交换式的希尔排序进行优化->移位法
    public static void shellSort2(int[] arr) {

        // 增量gap， 并逐步的缩小增量
        for (int gap = arr.length / 2; gap > 0; gap /= 2) {
            // 从第gap个元素，逐个对其所在的组进行直接插入排序
            // i++ 向后遍历数组,所有组依次进行排序,并不是一个组排序完进行下一个组
            for (int i = gap; i < arr.length; i++) {
                int j = i;
                int temp = arr[j];
                // j和gap组合 是每一组进行排序操作,i索引值对应数值的 组
                if (arr[j] < arr[j - gap]) {
                    // 如果这个数值小,那么这个数值的前一个数向后移动
                    // 索引值-步长,就是当前数值所在组的前一个数的索引
                    while (j - gap >= 0 && temp < arr[j - gap]) {
                        //移动
                        arr[j] = arr[j - gap];
                        j -= gap;
                    }
```

```
                    //当退出while后，就给temp找到插入的位置
                    arr[j] = temp;
                }

            }
        }
    }

}
```

归并排序

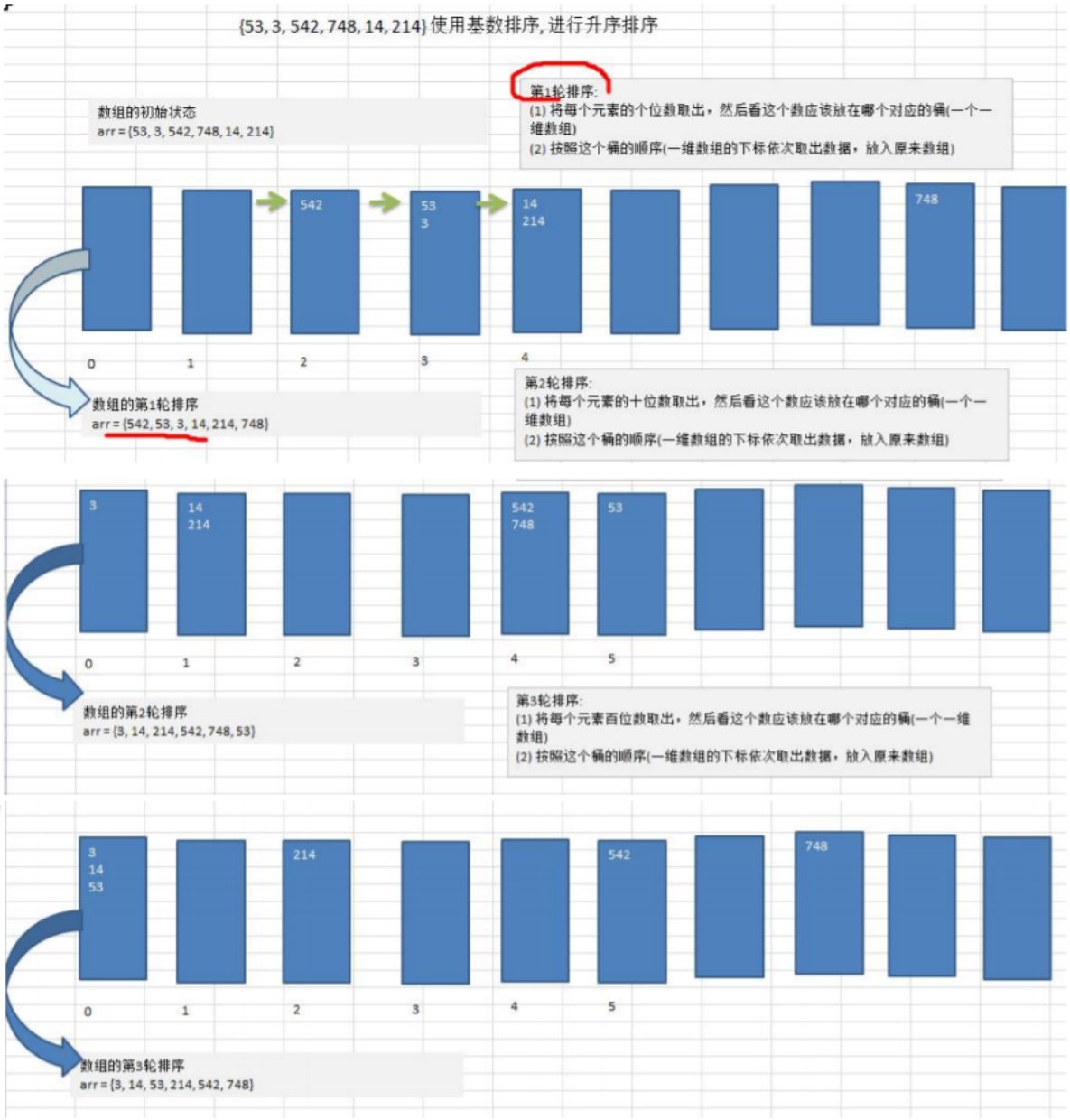



```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 *
 * 归并排序:拆过之后,排序合在一起
            1: 将原数组对半拆分,直到全部分开,递归(实际还是一个数组,只是理解为拆成若干小数组,用来第二步操作)
            2: 将拆开后的数组,两两结合进行判断
                ① 将两个数组的数据,按顺序提出并存放到一个临时数组
                ② 临时数组包含两个数组的全部数据,且是有序的
            3: 将临时数组按顺序存放到原数组
 *
 */

public class MergetSort {

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

```
        int arr[] = { 8, 4, 5, 7, 1, 3, 6, 2 }; //

        int temp[] = new int[arr.length]; //归并排序需要一个额外空间
        mergeSort(arr, 0, arr.length - 1, temp);

        System.out.println("归并排序后=" + Arrays.toString(arr));
    }


    //分+合方法
    public static void mergeSort(int[] arr, int left, int right, int[] temp)
{
        if(left < right) {
            int mid = (left + right) / 2; //中间索引
            //向左递归进行分解
            mergeSort(arr, left, mid, temp);
            //向右递归进行分解
            mergeSort(arr, mid + 1, right, temp);
            //合并
            merge(arr, left, mid, right, temp);

        }
    }

    //合并的方法
    /**
     *
     * @param arr 排序的原始数组
     * @param left 左边有序序列的初始索引
     * @param mid 中间索引
     * @param right 右边索引
     * @param temp 做中转的数组
     */
    public static void merge(int[] arr, int left, int mid, int right, int[]
temp) {

        int i = left; // 初始化i, 左边有序序列的初始索引
        int j = mid + 1; //初始化j, 右边有序序列的初始索引
        int t = 0; // 指向temp数组的当前索引

        //(一)
        //先把左右两边(有序)的数据按照规则填充到temp数组
        //直到左右两边的有序序列，有一边处理完毕为止
        while (i <= mid && j <= right) {//继续
            //如果左边的有序序列的当前元素，小于等于右边有序序列的当前元素
            //即将左边的当前元素，填充到 temp数组
            //然后 t++, i++
            if(arr[i] <= arr[j]) {
                temp[t] = arr[i];
                t += 1;
                i += 1;
            } else { //反之,将右边有序序列的当前元素，填充到temp数组
                temp[t] = arr[j];
                t += 1;
                j += 1;
            }
        }

        //(二)
```

```
        //把有剩余数据的一边的数据依次全部填充到temp
        while( i <= mid) { //左边的有序序列还有剩余的元素，就全部填充到temp
            temp[t] = arr[i];
            t += 1;
            i += 1;
        }

        while( j <= right) { //右边的有序序列还有剩余的元素，就全部填充到temp
            temp[t] = arr[j];
            t += 1;
            j += 1;
        }


        //(三)
        //将temp数组的元素拷贝到arr
        //注意，并不是每次都拷贝所有
        t = 0;
        int tempLeft = left; //
        //第一次合并 tempLeft = 0 , right = 1 //  tempLeft = 2  right = 3 //
tL=0 ri=3
        //最后一次 tempLeft = 0  right = 7
        while(tempLeft <= right) {
            arr[tempLeft] = temp[t];
            t += 1;
            tempLeft += 1;
        }

    }

}

```

基数排序




```java
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

/**
 *
 * 基数排序:桶子法
         1. 得到数组中最大的数的位数,得到最大数是几位数,用来判断遍历几次
         2. 创建一个二维数组,new int[10][arr.length],十个桶,0-9代表该位(个,十,百..)的数值
         3. 创建一个一维数组,int[10],按顺序存放每个桶的数据个数
         4. 将原数组每个元素进行循环操作
            第一次按照 个位 依次放到桶中,再按照顺序取到原数组中
            第二次按照 十位 依次放到桶中,再按照顺序取到原数组中
            ....
         5. 最终取到原数组的一次,就是排序成功的数值

 *
 */

public class RadixSort {
```

```
23
24      public static void main(String[] args) {
25          int arr[] = { 53, 3, 542, 748, 14, 214};
26
27          radixSort(arr);
28
29          System.out.println("基数排序后 " + Arrays.toString(arr));
30
31      }
32
33      //基数排序方法
34      public static void radixSort(int[] arr) {
35
36          //根据前面的推导过程，我们可以得到最终的基数排序代码
37
38          //1. 得到数组中最大的数的位数
39          int max = arr[0]; //假设第一数就是最大数
40          for(int i = 1; i < arr.length; i++) {
41              if (arr[i] > max) {
42                  max = arr[i];
43              }
44          }
45          //得到最大数是几位数
46          int maxLength = (max + "").length();
47
48
49          //定义一个二维数组，表示10个桶， 每个桶就是一个一维数组
50          //说明
51          //1. 二维数组包含10个一维数组
52          //2. 为了防止在放入数的时候，数据溢出，则每个一维数组(桶)，大小定为arr.length
53          //3. 名明确，基数排序是使用空间换时间的经典算法
54          int[][] bucket = new int[10][arr.length];
55
56          //为了记录每个桶中，实际存放了多少个数据,我们定义一个一维数组来记录各个桶的每次放入的数
    据个数
57          //可以这里理解
58          //比如：bucketElementCounts[0] ， 记录的就是  bucket[0] 桶的放入数据个数
59          // bucketElementCounts只是用来存每个桶中数据个数
60
61          // 下标索引值:第几个桶;该索引对应的值:该桶存放的数据个数
62          int[] bucketElementCounts = new int[10];
63
64
65          //这里我们使用循环将代码处理
66
67          for(int i = 0 , n = 1; i < maxLength; i++, n *= 10) {
68              //(针对每个元素的对应位进行排序处理)， 第一次是个位，第二次是十位，第三次是百位..
69              for(int j = 0; j < arr.length; j++) {
70                  //取出每个元素的对应位的值,digitOfElement就代表第几个桶,那么二维数组的第一个
    下标就是digitOfElement
71                  int digitOfElement = arr[j] / n % 10;
72                  //放入到对应的桶中
73                  bucket[digitOfElement][bucketElementCounts[digitOfElement]] =
    arr[j];
74                  bucketElementCounts[digitOfElement]++;
75              }
76              //按照这个桶的顺序(一维数组的下标依次取出数据，放入原来数组)
77              int index = 0;
78              //遍历每一桶，并将桶中是数据，放入到原数组
```

```
            for(int k = 0; k < bucketElementCounts.length; k++) {
                //如果桶中，有数据，我们才放入到原数组
                if(bucketElementCounts[k] != 0) {
                    //循环该桶即第k个桶(即第k个一维数组)，放入
                    for(int l = 0; l < bucketElementCounts[k]; l++) {
                        //取出元素放入到arr
                        arr[index++] = bucket[k][l];
                    }
                }
                //第i+1轮处理后，需要将每个 bucketElementCounts[k] = 0 ！！！！
                bucketElementCounts[k] = 0;

            }
            //System.out.println("第"+(i+1)+"轮，对个位的排序处理 arr =" +
Arrays.toString(arr));

        }
    }
}
```

中缀表达式转后缀表达式

```
 中缀表达式转后缀表达式
        1.定义两个栈,s1,s2(或者一个栈s1一个集合s2,两个栈更好理解)
            s1用来存运算符,s2用来存结果
        2.从左到右扫描中缀表达式
        3.遇到数字,直接压入s2
        4.遇到运算符时,比较 它与s1栈顶运算符的优先级
            ① 如果s1为空,或栈顶运算符为左括号"(",则直接将此运算符入栈s1
            ② 否则,若优先级比栈顶运算符的  高 ,也将运算符压入s1(不与s1中的括号进行比较)
            ③ 否则,将s1栈顶的运算符弹出并压入s2中,再次转到 4.1 与s1中新的栈顶运算符比较
                弹出一个后,继续和下一个进行比较优先级,小于等于弹出,继续下一个,直到没有或者优
先级高,入s1
        5.遇到括号时
            ① 如果是左括号"(",则直接压入栈
            ② 如果是有括号")",则一次弹出s1栈顶的运算符,并压入s2,直到遇到左括号为止,左括号弹
出但并不压入s2
                将这一对括号舍弃
        6. 重复步骤2-5,直到表达式的最右边
        7. 将s1中剩余的运算符依次弹出并压入s2
        8. 依次弹出s2中的元素并输出,结果的倒叙即为中缀表达式对应的后缀表达式
            如果s2为集合,输出结果即为中缀表达式对应的后缀表达式
```

```
package com.atguigu.stack;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

/**
 *
 * 中缀表达式转后缀表达式
        1.定义两个栈,s1,s2(或者一个栈s1一个集合s2,两个栈更好理解)
            s1用来存运算符,s2用来存结果
        2.从左到右扫描中缀表达式
```

```
        3.遇到数字,直接压入s2
        4.遇到运算符时,比较 它与s1栈顶运算符的优先级
            ① 如果s1为空,或栈顶运算符为左括号"(",则直接将此运算符入栈s1
            ② 否则,若优先级比栈顶运算符的 高 ,也将运算符压入s1(不与s1中的括号进行比较)
            ③ 否则,将s1栈顶的运算符弹出并压入s2中,再次转到 4.1 与s1中新的栈顶运算符比较
                弹出一个后,继续和下一个进行比较优先级,小于等于弹出,继续下一个,直到没有或者优先级高,入s1
        5.遇到括号时
            ① 如果是左括号"(",则直接压入栈
            ② 如果是有括号")",则一次弹出s1栈顶的运算符,并压入s2,直到遇到左括号为止,左括号弹出但并不压入s2
                将这一对括号舍弃
        6. 重复步骤2-5,直到表达式的最右边
        7. 将s1中剩余的运算符依次弹出并压入s2
        8. 依次弹出s2中的元素并输出,结果的倒叙即为中缀表达式对应的后缀表达式
            如果s2为集合,输出结果即为中缀表达式对应的后缀表达式
 *
 */
public class PolandNotation {

    public static void main(String[] args) {


        //完成将一个中缀表达式转成后缀表达式的功能
        //说明
        //1. 1+((2+3)×4)-5 => 转成  1 2 3 + 4 × + 5 -
        //2. 因为直接对str 进行操作，不方便，因此 先将  "1+((2+3)×4)-5" =》 中缀的表达式对应的List
        //   即 "1+((2+3)×4)-5" => ArrayList [1,+,(,(,2,+,3,),*,4,),-,5]
        //3. 将得到的中缀表达式对应的List => 后缀表达式对应的List
        //   即 ArrayList [1,+,(,(,2,+,3,),*,4,),-,5]  =》 ArrayList [1,2,3,+,4,*,+,5,-]

//        String expression = "1+((2+3)*4)-5";//注意表达式  16
        // [1, 2, 3, +, 4, *, +, 5, -]
//        String expression = "11/5+((5-1*5)-4*5)+2*3";//注意表达式  -12
        // [11, 5, /, 5, 1, 5, *, -, 4, 5, *, -, +, 2, 3, *, +]
        String expression = "11/5+((5*1-5)*4-5)/2-3";//注意表达式  3
        // [11, 5, /, 5, 1, *, 5, -, 4, *, 5, -, 2, /, +, 3, -]
        List<String> infixExpressionList = toInfixExpressionList(expression);
        System.out.println("中缀表达式对应的List=" + infixExpressionList); // ArrayList [1,+,(,(,2,+,3,),*,4,),-,5]
        List<String> suffixExpreesionList = parseSuffixExpreesionList(infixExpressionList);
        System.out.println("后缀表达式对应的List" + suffixExpreesionList); //ArrayList [1,2,3,+,4,*,+,5,-]

        System.out.printf("expression=%d", calculate(suffixExpreesionList)); // ?


        /*

        //先定义给逆波兰表达式
        //(30+4)×5-6  => 30 4 + 5 × 6 - => 164
        // 4 * 5 - 8 + 60 + 8 / 2 => 4 5 * 8 - 60 + 8 2 / +
        //测试
        //说明为了方便，逆波兰表达式 的数字和符号使用空格隔开
        //String suffixExpression = "30 4 + 5 * 6 -";
```

```
        String suffixExpression = "4 5 * 8 - 60 + 8 2 / +"; // 76
        //思路
        //1. 先将 "3 4 + 5 × 6 - " => 放到ArrayList中
        //2. 将 ArrayList 传递给一个方法，遍历 ArrayList 配合栈 完成计算

        List<String> list = getListString(suffixExpression);
        System.out.println("rpnList=" + list);
        int res = calculate(list);
        System.out.println("计算的结果是=" + res);

        */
    }


    //即 ArrayList [1,+,(,(,2,+,3,),*,4,),-,5]  =》 ArrayList [1,2,3,+,4,*,+,5,-]
    //方法：将得到的中缀表达式对应的List => 后缀表达式对应的List
    public static List<String> parseSuffixExpreesionList(List<String> ls) {
        //定义两个栈
        Stack<String> s1 = new Stack<String>(); // 符号栈
        //说明：因为s2 这个栈，在整个转换过程中，没有pop操作，而且后面我们还需要逆序输出
        //因此比较麻烦，这里我们就不用 Stack<String> 直接使用 List<String> s2
        //Stack<String> s2 = new Stack<String>(); // 储存中间结果的栈s2
        List<String> s2 = new ArrayList<String>(); // 储存中间结果的Lists2

        //遍历ls
        for (String item : ls) {
            //如果是一个数，加入s2
            if (item.matches("\\d+")) {
                s2.add(item);
            } else if (item.equals("(")) {
                s1.push(item);
            } else if (item.equals(")")) {
                //如果是右括号“)”，则依次弹出s1栈顶的运算符，并压入s2，直到遇到左括号为止，此时将这一对括号丢弃
                while (!s1.peek().equals("(")) {
                    s2.add(s1.pop());
                }
                s1.pop();//!!! 将 ( 弹出 s1栈， 消除小括号
            } else {
                //当item的优先级小于等于s1栈顶运算符，将s1栈顶的运算符弹出并加入到s2中，再次转到(4.1)与s1中新的栈顶运算符相比较
                //问题：我们缺少一个比较优先级高低的方法
                while (s1.size() != 0 && Operation.getValue(s1.peek()) >= Operation.getValue(item)) {
                    s2.add(s1.pop());
                }
                //还需要将item压入栈
                s1.push(item);
            }
        }

        //将s1中剩余的运算符依次弹出并加入s2
        while (s1.size() != 0) {
            s2.add(s1.pop());
        }

        return s2; //注意因为是存放到List，因此按顺序输出就是对应的后缀表达式对应的List

```

```
    }

    //方法：将 中缀表达式转成对应的List
    //  s="1+((2+3)×4)-5";
    public static List<String> toInfixExpressionList(String s) {
        //定义一个List,存放中缀表达式 对应的内容
        List<String> ls = new ArrayList<String>();
        int i = 0; //这时是一个指针，用于遍历 中缀表达式字符串
        String str; // 对多位数的拼接
        char c; // 每遍历到一个字符，就放入到c
        do {
            //如果c是一个非数字，我需要加入到ls
            if ((c = s.charAt(i)) < 48 || (c = s.charAt(i)) > 57) {
                ls.add("" + c);
                i++; //i需要后移
            } else { //如果是一个数，需要考虑多位数
                str = ""; //先将str 置成"" '0'[48]->'9'[57]
                while (i < s.length() && (c = s.charAt(i)) >= 48 && (c =
s.charAt(i)) <= 57) {
                    str += c;//拼接
                    i++;
                }
                ls.add(str);
            }
        } while (i < s.length());

        System.out.println("ls:" + ls.toString());
        return ls;//返回
    }

    //将一个逆波兰表达式， 依次将数据和运算符 放入到 ArrayList中
    public static List<String> getListString(String suffixExpression) {
        //将 suffixExpression 分割
        String[] split = suffixExpression.split(" ");
        List<String> list = new ArrayList<String>();
        for (String ele : split) {
            list.add(ele);
        }
        return list;

    }

    //完成对逆波兰表达式的运算
    /*
     * 1)从左至右扫描，将3和4压入堆栈；
        2)遇到+运算符，因此弹出4和3（4为栈顶元素，3为次顶元素），计算出3+4的值，得7，再将
7入栈；
        3)将5入栈；
        4)接下来是×运算符，因此弹出5和7，计算出7×5=35，将35入栈；
        5)将6入栈；
        6)最后是-运算符，计算出35-6的值，即29，由此得出最终结果
     */

    public static int calculate(List<String> ls) {
        // 创建给栈，只需要一个栈即可
        Stack<String> stack = new Stack<String>();
        // 遍历 ls
        for (String item : ls) {
            // 这里使用正则表达式来取出数
```

```java
            if (item.matches("\\d+")) { // 匹配的是多位数
                // 入栈
                stack.push(item);
            } else {
                // pop出两个数，并运算， 再入栈
                int num2 = Integer.parseInt(stack.pop());
                int num1 = Integer.parseInt(stack.pop());
                int res = 0;
                if (item.equals("+")) {
                    res = num1 + num2;
                } else if (item.equals("-")) {
                    res = num1 - num2;
                } else if (item.equals("*")) {
                    res = num1 * num2;
                } else if (item.equals("/")) {
                    res = num1 / num2;
                } else {
                    throw new RuntimeException("运算符有误");
                }
                //把res 入栈
                stack.push("" + res);
            }

        }
        //最后留在stack中的数据是运算结果
        return Integer.parseInt(stack.pop());
    }

}

//编写一个类 Operation 可以返回一个运算符 对应的优先级
class Operation {
    private static int ADD = 1;
    private static int SUB = 1;
    private static int MUL = 2;
    private static int DIV = 2;

    //写一个方法，返回对应的优先级数字
    public static int getValue(String operation) {
        int result = 0;
        switch (operation) {
            case "+":
                result = ADD;
                break;
            case "-":
                result = SUB;
                break;
            case "*":
                result = MUL;
                break;
            case "/":
                result = DIV;
                break;
            case "(":
                break;
            case ")":
                break;
            default:
                System.out.println("不存在该运算符" + operation);
```

```
                break;
            }
            return result;
        }

}

```