Java 程序设计基础

方法与数组

方法与数组主要内容

- **4.1 Java 方法（重点）**
- **4.2 一维数组（重点）**
- **4.3 简单的排序算法（重点、难点）**
- 4.4 二维数组

本章教学目标

- 能理解 int 型数组的内存演变过程
- 能理解 String 型数组的内存演变过程
- 能准确的使用递归方法解决常见问题
- 能理解冒泡排序、插入排序和快速排序的算法逻辑
- 能准确的编写冒泡排序、插入排序和快速排序的实现代码
- 能区分单向扫描法和双向扫描法两种方式的快速排序
- 能理解二维数组的存储结构
- 能理解二维数组的存储结构
- 能准确的为二维数组元素赋值、遍历二维数组元素

- 能理解 int 型数组的内存演变过程
- 能理解 String 型数组的内存演变过程
- 能准确的使用递归方法解决常见问题
- 能理解冒泡排序、插入排序和快速排序的算法逻辑
- 能准确的编写冒泡排序、插入排序和快速排序的实现代码
- 能区分单向扫描法和双向扫描法两种方式的快速排序
- 能准确的为二维数组元素赋值、遍历二维数组元素

什么是 Java 方法

- 方法是 Java 中一个命名的代码块，我们一直在使用的 main 就是一个方法
- 方法可以提高代码的复用度，减少重复
- 方法可以使程序结构更加清晰
- **Java 方法声明的语法形式如下（重点）**

```
[修饰符] 返回值类型 方法名([形参列表]){
    方法体
}
```

大括号前面的内容称为方法头，大括号中的内容称为方法体

方法声明后如何调用

- 本节介绍类内部方法的调用
- **在类内部调用方法，只需给出方法名以及方法的实际参数列表（实参列表的数据类型必须与形参列表一致或可以自动转换成形参列表的格式），如下**

```
int x = returnAdd(3 + 5);
drawStar(8);
```

- 使用方法调用的形式，代码结构清晰，方法声明可以被复用
- JDK 本身也提供了很多的方法，我们也一直都在使用。例如，System.out.println()就是在调用 JDK 提供的方法向向控制台输出，nextInt() 方法（Scanner 类）从控制台获取用户输入的整数等

方法声明里的元素：修饰符和返回值类型

- 修饰符：用来规定方法的可见范围等特征

  例如，我们一直在使用的 main 方法，其中的 public static 就是修饰符，

  public 表示这个方法的可见范围，static 表示 main 方法是一个静态方法

- 返回值类型：表示该方法返回什么类型的值

  特殊的，如果方法无需返回值，这时需要用 void 表示

  一旦一个方法需要返回值时，那么方法体中就必须使用 return 语句返回此类型的值

方法声明里的元素：修饰符和返回值类型

- 示例

```
public void drawCircular()      //该方法的返回值为空
{...}

public int returnInt(){              //该方法返回值为int类型
        int x = 10;
        ...
        return x;
}
```

- 一个方法只能有一个返回值，因此也只能有一个返回值类型

  如果逻辑上确实需要返回多个值，则可以将需要返回的“多个值”先转换为一个数组或一个对象，

  然后再返回转换后的这“一个值”

方法声明里的元素：方法名和形参列表

- 示例

```
public int returnAdd(int x,int y){      //x、y是形参
    return x + y;
}

// 以下是方法调用，后续讲解；这里仅用于区分形参和实参
public int test(){
    returnAdd(1,2);      //1、2是实参
}
```

一个特殊的方法(递归)

- 递归调用是指一个方法在它的方法体内调用它自身
- **Java 语言允许方法的递归调用，在递归调用中，主调方法同时也是被调方法。执行递归方法将反复调用其自身，每调用一次就再进入一次本方法**
- **递归调用最容易出现的问题是，如果递归调用没有退出的条件，则递归方法将无休止地调用其自身，这显然是不正确的。为了防止递归调用无休止地进行，必须在方法内有终止递归调用的手段。通常的做法就是增加条件判断，满足某条件后就不再进行递归调用，然后逐层返回**

一个特殊的方法(递归)

- 示例：使用递归调用计算整数n的阶乘

```
//求n的阶乘的方法
long factorial(int n) {
    if (n == 1) {                //判断条件，一旦满足就不再递归，逐层返回
        return 1;
    }
    long sum = factorial(n - 1);    //递归调用
    return sum * n;              //逐层返回求阶乘
}
```

什么是数组

- 如果一个系统中存储的是一个 Java 工程师信息，假设这个系统需要存储 100 个 Java 工程师信息，如何便捷的存储这些信息？使用数组
- Java 提供了一种称为数组的数据类型，数组不是基本数据类型，而是引用数据类型
- **数组是把相同类型的多个变量按一定顺序组织起来，这些按序排列的同类型数据元素的集合称为数组**
- **数组中的元素在内存中是连续存储的**
- 数组中的数据元素既可以是基本类型，也可以是引用类型

使用数组的三个步骤(声明、创建和赋值)

- **使用数组时，需要声明、创建、赋值并使用三个步骤**
- 声明数组的语法形式如下，推荐使用前一种

  数据类型[] 数组名; 或 数据类型 数组名[];
- 声明数组就是告诉内存，该数组中元素是什么类型的，如下

  int engNo[];

  double[] engSalary;

  String[] engName;//String 字符串是引用类型，engName 数组里存放的是引用类型元素

  必须注意，Java 语言中声明数组的时候不可以指定数组长度，例如int engNo[100]是非法的

使用数组的三个步骤(声明、创建和赋值)

- **创建数组，就是要为数组分配内存空间，不分配内存是不能存放数组元素的，创建数组就是在内存中划分出几个连续的空间用于依次存储数组中的数据元素，其语法形式如下**

  数组名 = new 数据类型[数组长度];

  也可以把数组声明和数组创建合并，其语法形式为：

  数据类型[] 数组名 = new 数据类型[数组长度];

  其中数组长度就是数组中存放的元素个数，必须是正整数，如下

  int[] engNo = new int[5];

  String[] engName = new String[5];

使用数组的三个步骤(声明、创建和赋值)

- 赋值并使用数组。在使用数组时，主要通过下标来访问数组元素
- 给数组赋值的语法形式如下

  数组名[数组下标] = 数值;

- 数组下标从 0 开始编号，数组名[0]代表数组中第 1 个元素，数组名[1]代表数组中第 2 个元素...
  数组下标的最大值为数组长度减 1，如果下标值超过最大值会出现数组下标越界问题，如下

```
engNo[0] = 1001;

engNo[1] = 1002;`

engName[4] = "张三";

engName[5] = "李四";//错误，数组的最大长度是 5，因此最后一个数组元素是 engName[4]
```

一维数组的静态初始化

- **本节介绍一维数组的静态初始化，即在声明、创建的时候同时直接为元素赋值**

  int[] engNo = new int[]{1001,1002,1003,1004,1005};

  String[] engName = new String[]{"柳海龙","孙传杰","孙悦"};

  甚至可以直接写成：

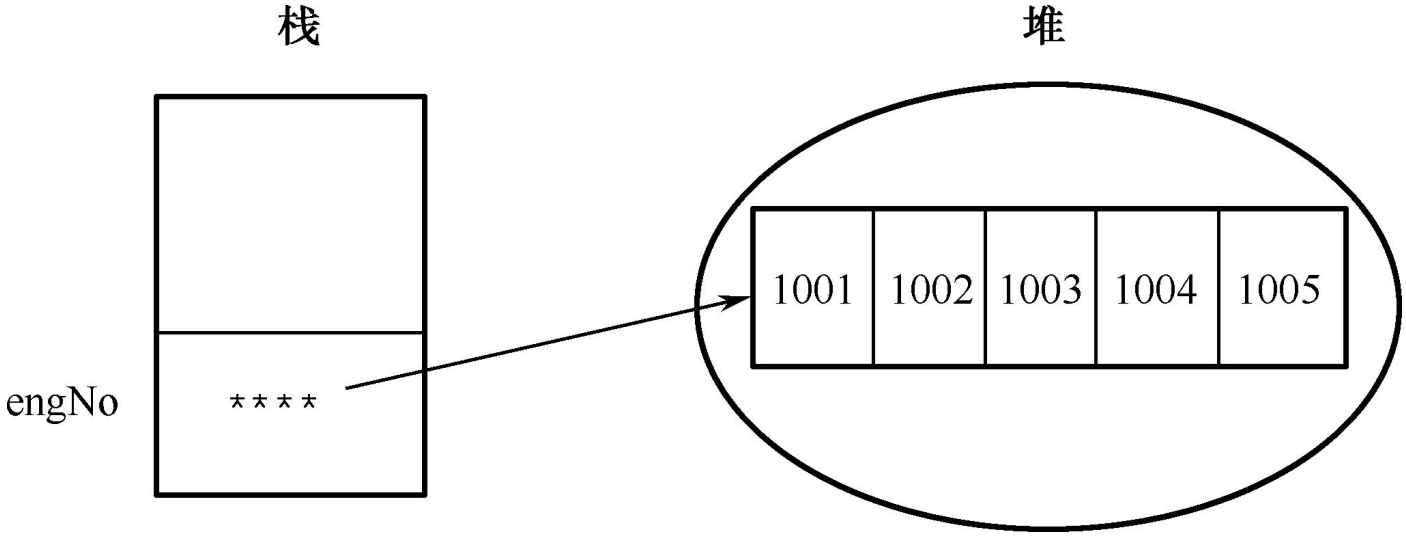
  int[] engNo = {1001,1002,1003,1004,1005};

  String[] engName = {"柳海龙","孙传杰","孙悦"};

- 静态初始化适用于一开始就知道数组内容的情况

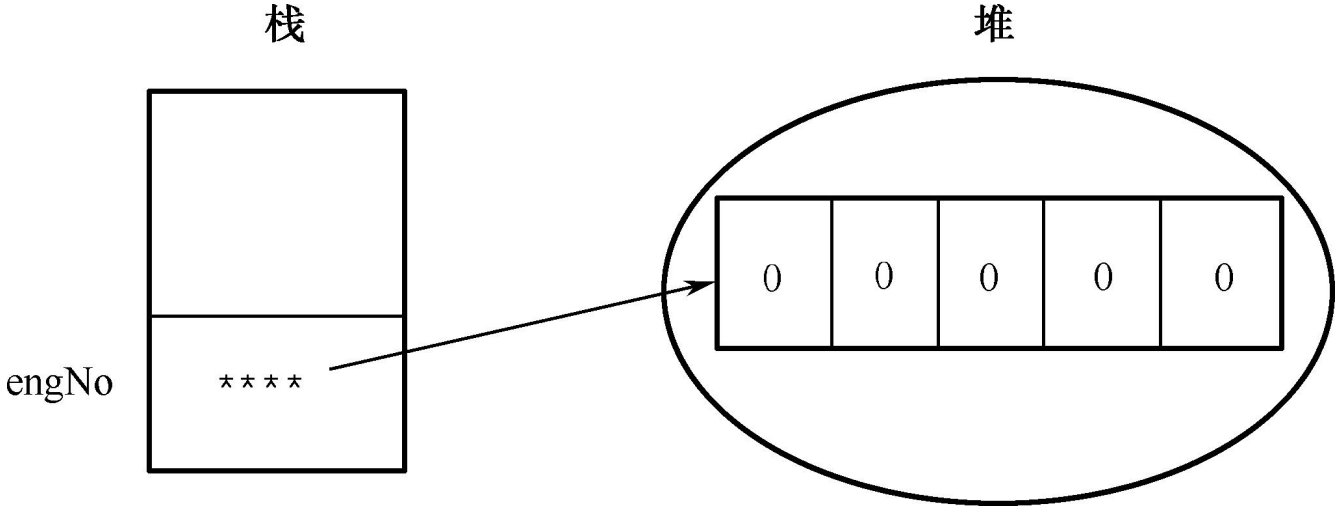
数组的内存结构

- 数组不属于基本数据类型（虽然其内部存储的可能是基本数据类型），只能是引用数据类型。因此语句 int[] engNo = new int[]{1001,1002,1003,1004,1005}; 中，数组本身是在堆内存中开辟的空间，该空间的地址存储在栈内存中并命名为 engNo，如图



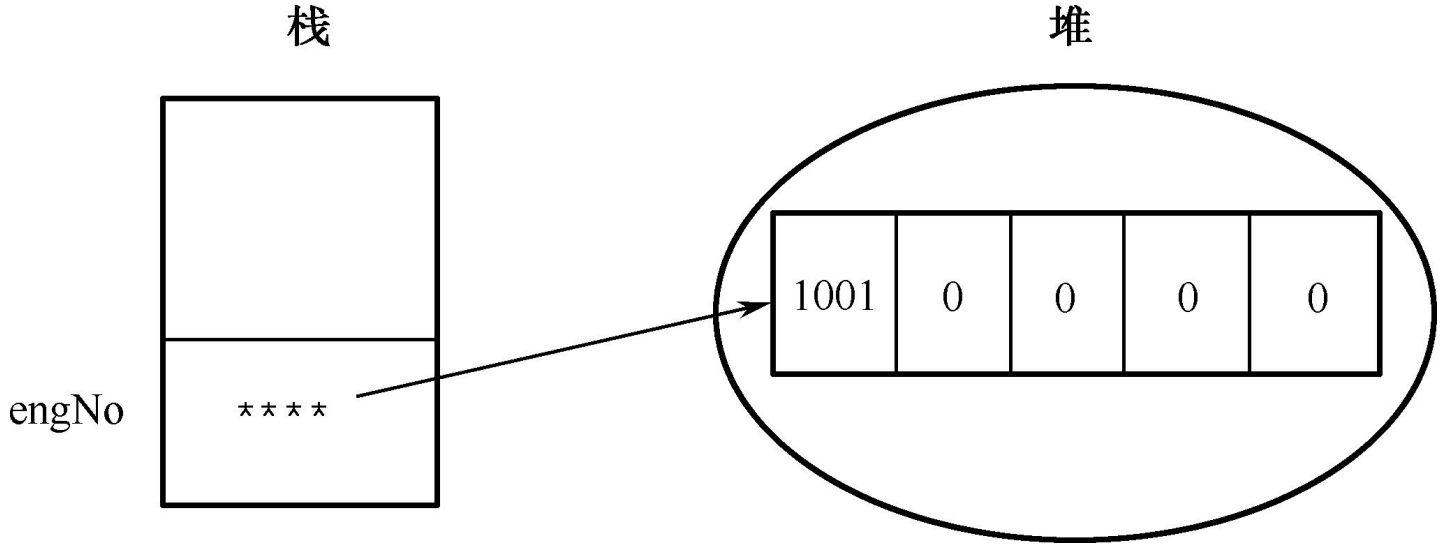
int 型数组的内存演变过程

- 声明，int[] engNo; 在栈内存中分配 1 个空间命名为 engNo，用于存放整型数组的地址（现在这个地址还不存在，默认是 null）
- 创建， int[] engNo = new int[5]; 在堆内存中分配 5 个连续空间，并把空间地址赋给 engNo ，使 engNo 指向这 5 个连续的内存空间，这 5 个空间里存放的默认初始值为 0（整数的默认值），如图所示



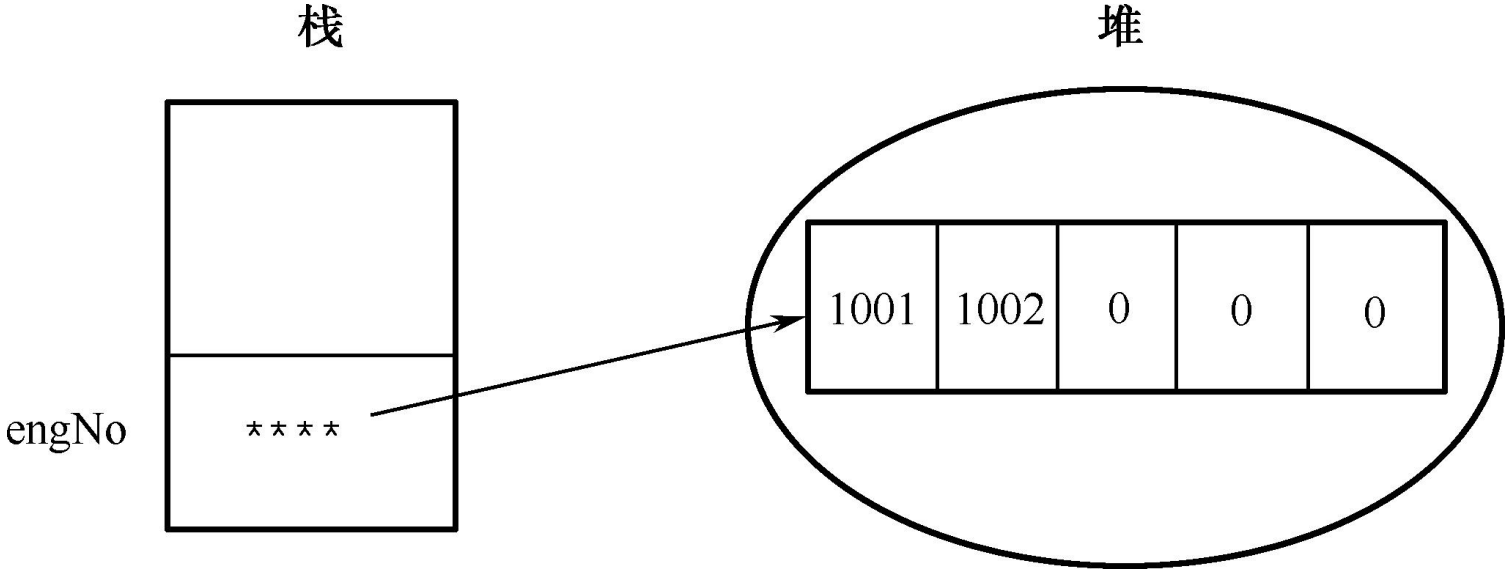
int 型数组的内存演变过程

- 赋值，engNo[0] = 1001; 将整数 1001 放到 engNo 数组的第 1 个元素空间里，如图所示



int 型数组的内存演变过程

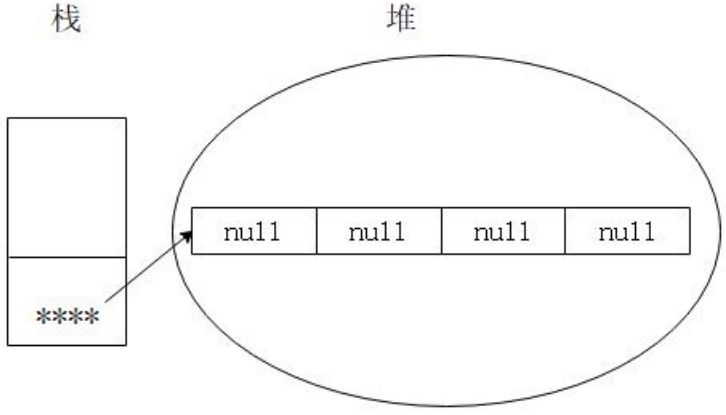
- 赋值，engNo[1] = 1002; 将整数 1002 放到 engNo 数组的第 2 个元素空间里，如图所示



String 型数组的内存演变过程

- 引用的名称实际代表的是存放数据的地址。因此，如果数组中的元素是 String 类型，那么数组元素存放的就是 String 类型的引用即数据的地址。

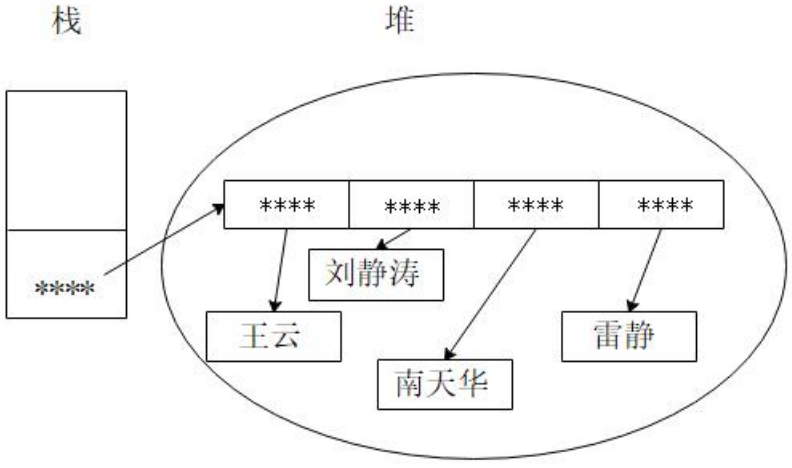
- 在使用 String[] names = new String[4]; 创建字符串数组时，会在堆内存中分配 4 个连续空间，并把空间地址赋给 names，使 names 指向这 4 个连续的内存空间。这 4 个空间里存放的默认初始值为 null（String 类型数据的默认值），如图所示。



String 型数组的内存演变过程

- names[0] = "王云"; 因为数组的元素是引用类型，因此会将"王云"的地址放到 names 数组的第 1 个元素空间里

- 之后，依次将"刘静涛", "南天华", "雷静"的地址存放到 names 中相应的位置，如图所示



为什么要学习排序算法

- 从实际编程的角度看，使用 Java 的一些工具类中的排序方法，就可以实现排序的功能。但作为学习数组的强化，编写一些简单算法是很有益处的
- 所谓排序，就是使一串记录（通常存储在数组中），按照某种算法，递增或递减地排列起来的操作
- 排序的算法有很多，各种算法对空间的要求及时间效率也各有差别

冒泡排序

- 算法描述
  - 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
  - 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
  - 针对所有的元素重复以上的步骤，除了最后一个；
  - 重复步骤1~3，直到排序完成。
- 通过上面的分析可以看出，假设需要排序的序列的个数是 n，则需要经过 n-1 轮，最终完成排序。在第一轮中，比较的次数是 n-1 次，之后每轮减少 1 次。冒泡排序的核心是元素交换。

选择排序

- 工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。。

插入排序

- 插入排序将待排序的数据分为两个部分，第一部分中的数据是已经排好序的（初始时，第一个数据划入第一部分），第二部分中的数据是无序的。之后，每次从第二部分取出头部（即第一个）元素，把它插入到第一部分的合适位置，使插入后的第一部分仍然有序。
- 直接插入排序的具体流程如下

第一轮：以下标 1 的元素（记为 a1，后面类似）为头部向前找插入位置，如果 a1 ⩾a0 ，则维持现状；否则 a1 向前插（a0 后移，位置 0 处将 a1 放入）。

第二轮：以 a2 为头部向前找插入位置，如果 a2 ⩾ a1 ，则维持现状；否则 a1 向后挪。

继续与 a0 比较，此时如果 a2 ⩾a0 ，则在位置 1 处放入 a2 ，如果 a0 ＞ a2 ，将 a0 后挪一位，在位置 0 处将 a2 放入。

总的来说，第 i 轮时，ai 为待定位的数据，通过与前序数据比较并将更大的数据后挪（如有必要）的方式找到合适的位置插入 ai，使得数组的前 i+1 个元素有序。一共要进行 n-1 轮。

快速排序法

- 快速排序是对冒泡排序的一种改进，是通过每一趟排序，将要排序的数组（或后续讲解的集合）分割成两个独立的部分。其中，一部分的所有数据比另一部分的所有数据都要小。然后通过递归重复这种操作，对分割后的两部分数据分别进行快速排序，最终达到整个数据都是有序排列的。
- 本节将会介绍单向扫描法和双向扫描法两种方式的快速排序。

快速排序法

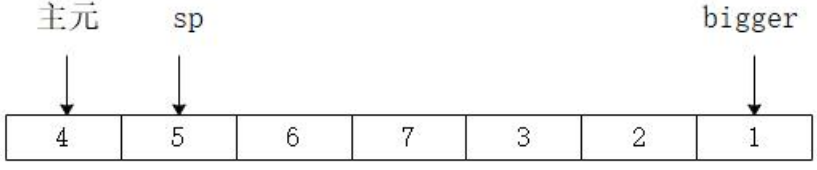
- 快速排序之单向扫描法

选定数组的一个元素，将之称为“主元”。之后，扫描一趟数组，将大于或等于主元的元素放在主元的右边，把小于或等于主元的元素放在主元的左边，这个过程被称为用主元分割数组。

具体做法是：

1.选定数组的第一个元素（即 nums 数组中的元素 4）作为主元。

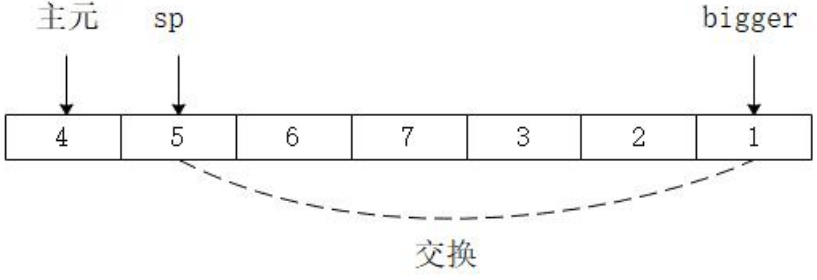
2.定义两个标记变量 sp 和 bigger，它们都是数组下标。其中 sp 表示 1 中，在从左往右扫描一趟数组的过程中，当前正在扫描的位置，它会向右移动；bigger 是边界，其右侧的数据大于或等于主元。初始时，sp 指向数组的第二个元素，bigger 指向数组的最后一个元素，如图所示



快速排序法

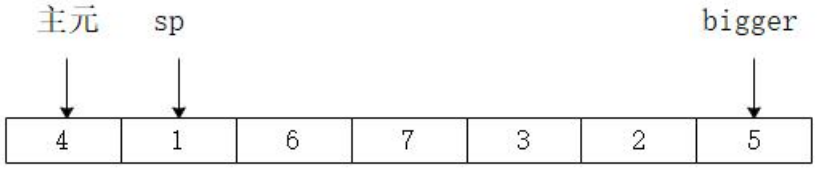
3.假设数组名是 arr，第一趟的比较流程是：在 sp≤bigger 的情况下循环，比较 arr[sp]<=主元是否成立，如果是，sp 右移一位；否则，就交换 arr[sp] 和 arr[bigger]，并将 bigger 的位置左移一位（注意 sp 原地不动），第一次比较过程如图所示。

（1）数据交换前

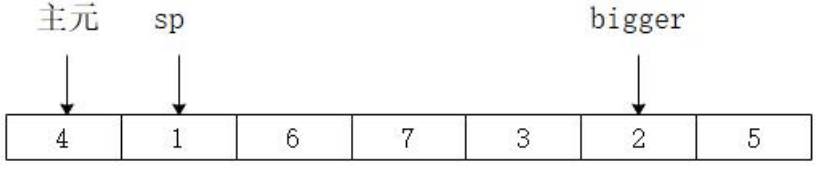


快速排序法

（2）数据交换后



（3）bigger 左移一位



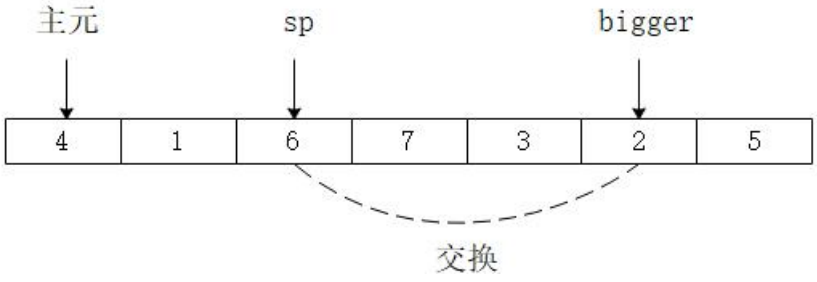
快速排序法

4.继续循环，重复 c 中描述的过程，如下。

（1）（续接上图)因为当前的arr[sp]<=主元（1<4）成立，所以 sp 右移一位即 sp++；

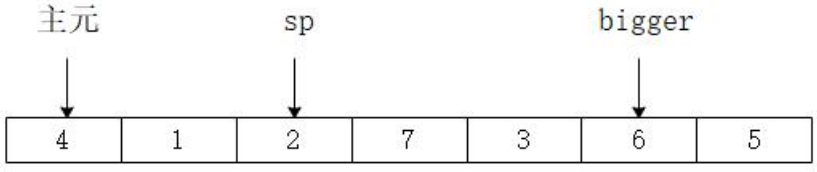
（2） 此时 arr[sp]<=主元（6>4）不成立，所以交换 arr[sp] 和 arr[bigger]，再将 bigger 左移一位，如下图所示。

数据交换前

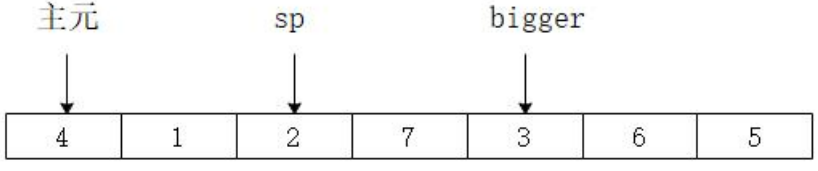


快速排序法

数据交换后



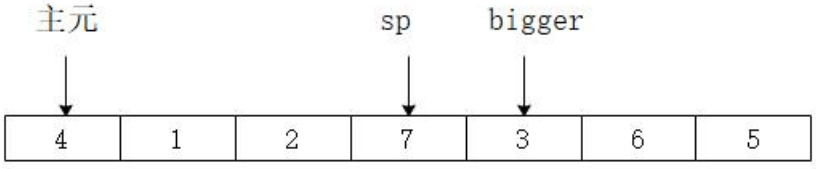
bigger 左移一位



快速排序法

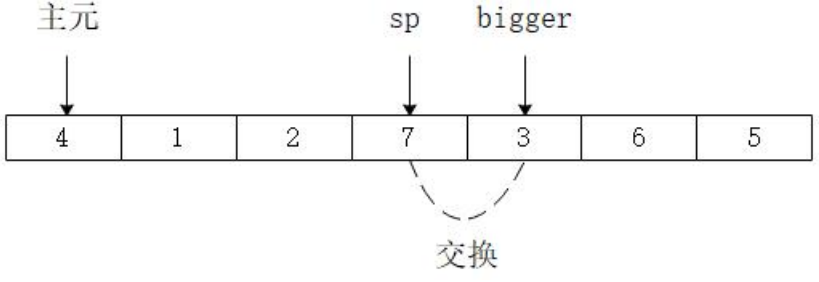
（1）因为arr[sp]<=主元（2 < 4）成立，sp 右移一位，如图所示

sp 右移一位



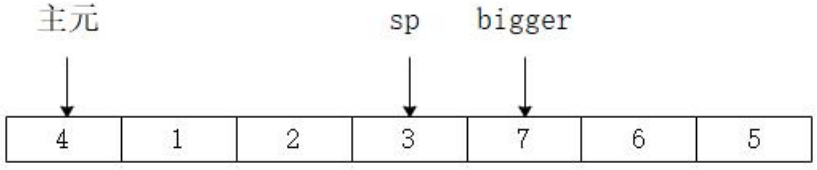
（2）因为 arr[sp]<=主元（7>4）不成立，所以交换 arr[sp] 和 arr[bigger]，再将 bigger 左移一位，如下图所示

数据交换前

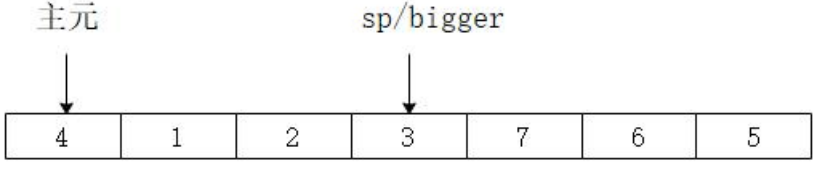


快速排序法

数据交换后



bigger 左移一位



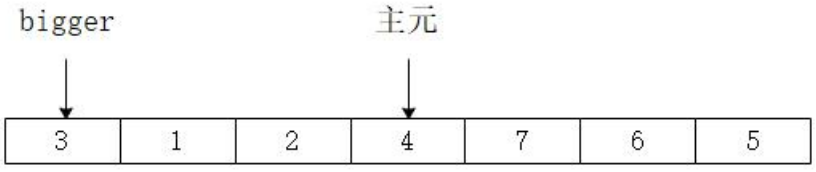
快速排序法

此时，sp==bigger，仍要继续判断，此处因为 arr[sp]<=主元（3<4）成立，所以 sp 还会右移一次变为大于 bigger，而 bigger 保持不动

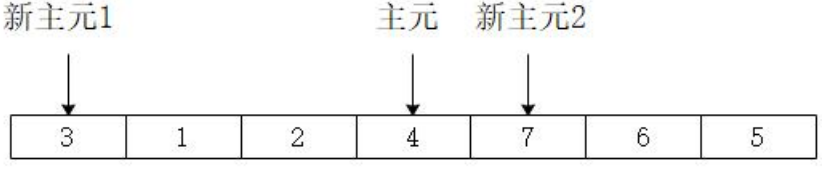
至此，循环结束，bigger 右侧的数据全部大于或等于主元，注意 bigger 本身指向的数据是小于主元的，下面通过交换 arr[bigger] 和主元就可以完成以主元分割数组的任务

快速排序法

5.交换 arr[bigger] 和主元，如图所示。



此时，主元左边的元素都小于主元，主元右边的元素都大于或等于主元。即主元已经是处在排好序的位置了。将此时的数组以主元为界，分割为两部分，分别对两部分递归处理，即从a. 开始——选取新的主元（图中的新主元 1 和新主元 2），如图所示。



用同样的方法，将新的主元也放置到排好序的位置

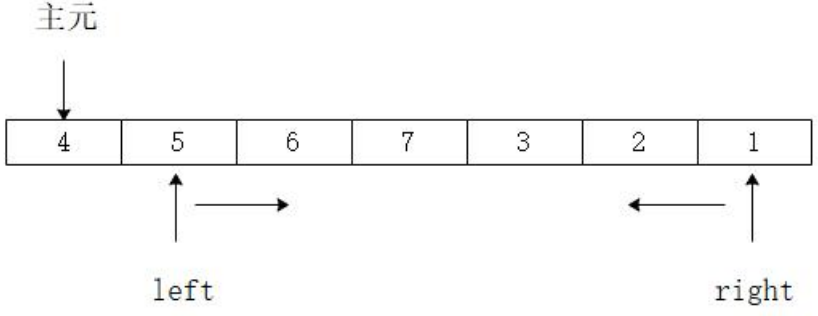
快速排序法

递归的结束条件是：子数组只有 1 个元素或者 0 个元素

快速排序法

- 快速排序之双向扫描法

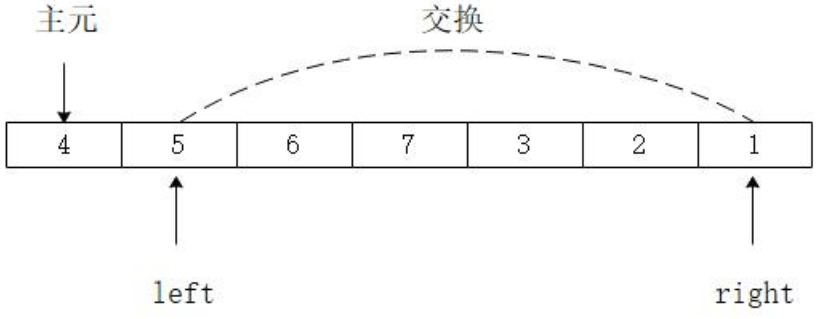
双向扫描法仍然是选取第一个元素为主元，然后在主元以外的元素里，从左右两侧同时扫描，如下图中的 left、right



快速排序法

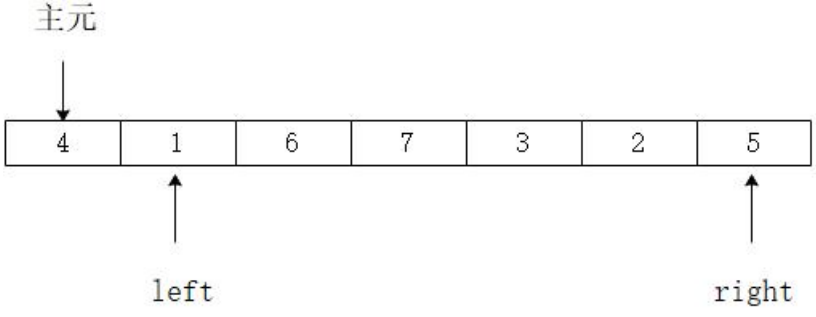
left 在向右扫描（移动）的过程中，如果 arr[left]<=主元 成立，则 left 右移，否则停止移动；right 向左扫描的过程中，如果 arr[right]>=主元 成立，则 right 左移，否则停止移动。当 left 和 right 都停止移动时，如果这时 left<=right，则交换左右 arr[left] 和 arr[right]，如下图所示。

扫描停止并准备交换数据：



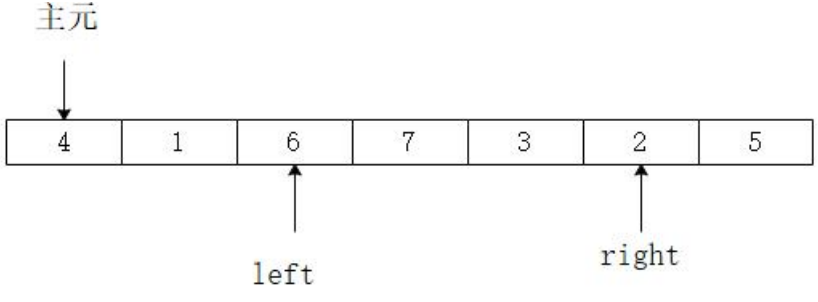
快速排序法

交换后的数据：



之后继续向中间遍历，直到 left> right，如下。

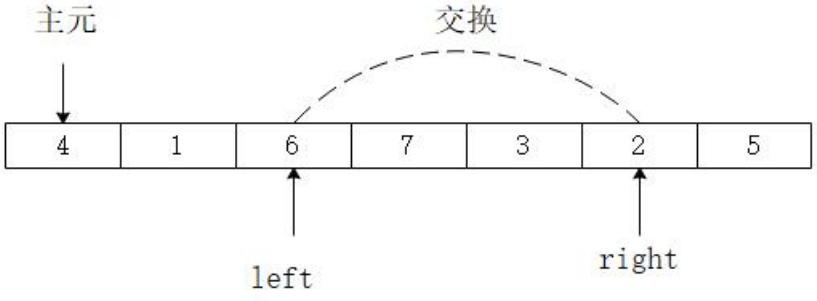
1.上一个图中，arr[left]<=主元 成立，所以 left 右移；arr[right]>=主元 成立，所以 right 左移，如图所示



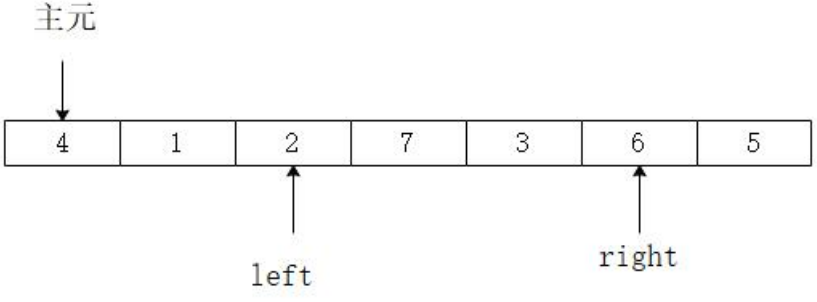
快速排序法

此时，left<right，因此需要交换 arr[left]和 arr[right]，如图所示。

right 左移数据交换前:

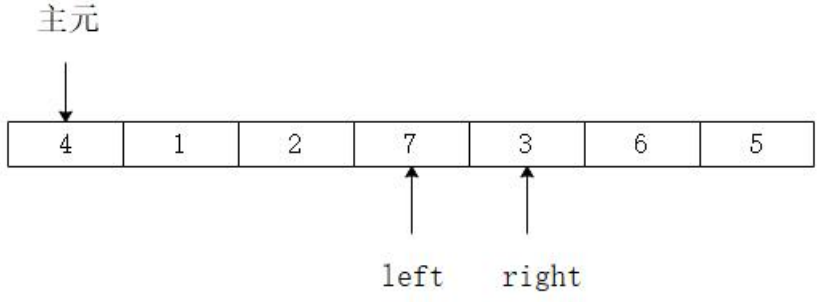


right 左移数据交换后:

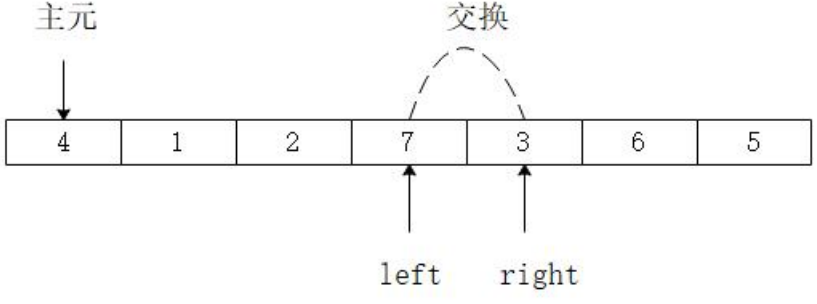


快速排序法

2.上一个图中，arr[left]<=主元成立，所以 left 右移；arr[right]>=主元成立，所以 right 左移，如图所示

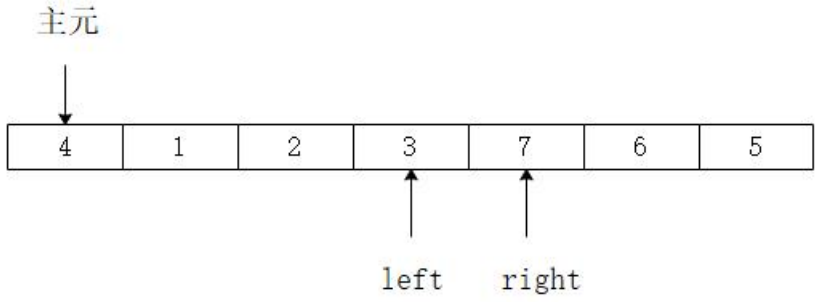


此时，left<=right 成立，且 left 和 right 满足停止条件，因此需要交换 arr[left] 和 arr[right]，如下图所示。 数据交换前：



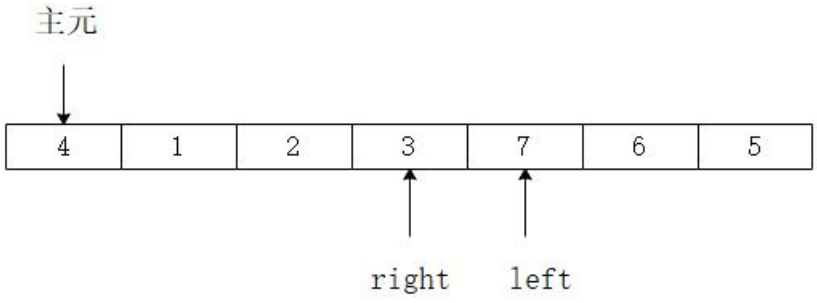
快速排序法

数据交换后：



快速排序法

3.上一个图中，arr[left]<=主元 成立，所以 left 右移；arr[right]>=主元 成立，所以 right 左移，如图所示

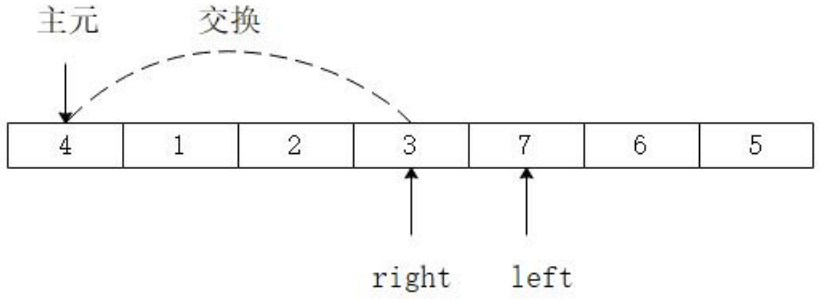


此时,已达到了循环退出条件 left> right，因此循环终止

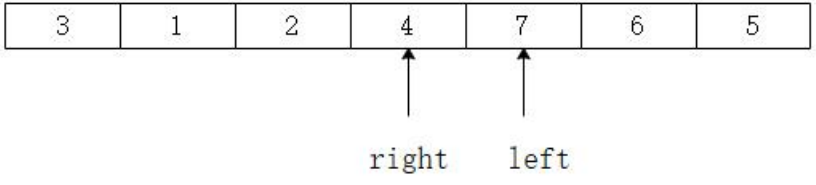
快速排序法

4. 再交换主元与 arry[right]，如下图所示。

主元与 arry[right]交换前：



主元与 arry[right]交换后：



快速排序法

至此，主元也处在了合适的位置上，一趟排序结束。

再用递归，将主元左侧和右侧的子数组视为两个需要排序的数组，重复以上步骤即可实现对整个数组的排序。

二维数组

- Java 语言允许构造多维数组并存储多维数据
- 多维数组的数组元素有多个下标，以标识它在数组中的位置
- 声明并创建二维数组的语法形式如下

  数据类型[][] 数组名；或 数据类型 数组名[][];

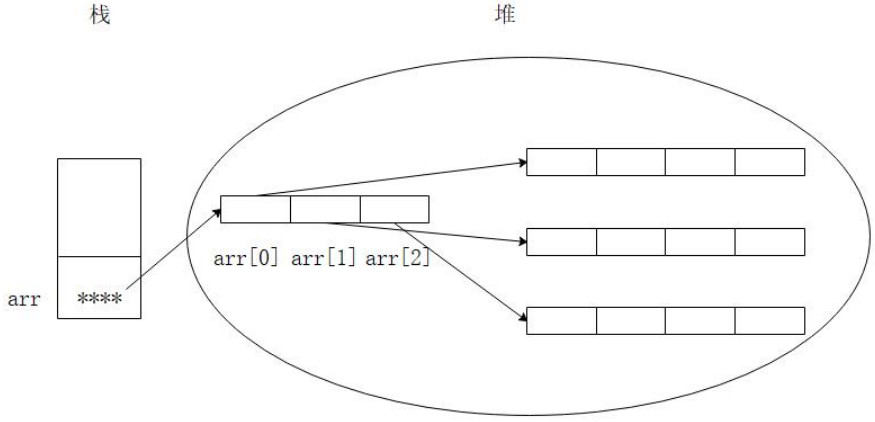
  数组名 = new 数据类型[第一维长度] [第二维长度];

- 创建的时候，可以同时设置第一维长度和第二维长度，也可以只设置第一维长度，但不可以只设置第二维长度

```
int[][] arr = new int[3][4];
int[][] arr = new int[3][];
```

二维数组

- 二维数组本质是一维数组的叠加，二维数组的所有元素都是引用类型，这些元素分别指向不同的一维数组，其内存结构和之前介绍的一维数组类似。例如，二维数组 arr 指向了由 arr[0]、arr[1]和 arr[2] 组成的数组，而 arr[0]、arr[1]和 arr[2] 自身也是一维数组。
- arr[0]、arr[1]和arr[2] 是二维数组的三个元素，这三个钥匙盒是并排的。
- arr[0]、arr[1]和arr[2] 所引用的是三个独立的数组，由于 new 操作符总是开辟新的空间且无法保证连续，arr[0]、arr[1]和arr[2] 中的“钥匙”（即地址）不是连续的，如图所示。



二维数组

- 二维数组的赋值和使用与一维数组类似，都是通过下标访问数组元素，不同的是一维数组只有一个下标，而二维数组有两个下标，分别表示该元素所在数组的行数和列数。例如 arr[0][3]，其表示的是数组 arr 中第 1 行第 4 列的元素
- 如何遍历二维数组？二维数组是二维的，是由“行”和“列”构成的。因此，只需要先遍历每一行，然后再遍历每一行中的每一列即可。其中的“行”，就是构成二维数组的元素：一维数组。遍历二维数组的伪代码如下所示

```
//遍历二维数组的每一行
for (int i = 0; i < 二维数组.length; i++) {
    //遍历每一行中的每一列
    for (int j = 0; j < 二维数组[i].length; j++) {
        //二维数组[i][j]
    }
}
```

笔试题

- 以下哪些代码是正确的（）

（1）int[] nums = new int[3];

（2）int[3] nums = new int[];

（3）int[] nums = new int[]{1,2,3};

（4）int[] nums = new int[3]{1,2,3};

（5）int[] nums ={1,2,3};

（6）int[] nums ;

nums = {1,2,3};

（7）int[] nums = new int[3];

nums = {1,2,3};

笔试题

- 以下关于test()方法的定义，哪一个是错误的（　）？

A. void test(int a, int b) {...}

B. static test(int a, int b) {...}

C. public static void test(int a, int b) {...}

D. public void test(int a, int b) {...}

笔试题

- nums是一个已经按照从小到大的顺序，排好序的int数组。nums数组中最小的数字值是？最大的数字值是？

笔试题

- 请使用方法和数组编写程序，实现以下功能

  某公司为新员工设置以下抽奖活动：员工入职时会登记姓名和年龄，登记完毕后，系统会为该员工自动分配一个四位随机数，作为该员工的员工编号。之后，系统会生成6个四位随机数作为当日的幸运数字。如果员工编号正好是6个幸运数字之一，则此员工中奖。

笔试题

示例输出：

假设员工输入如下：

```
Copy Code

请输入员工姓名:
张三
请输入员工年龄:
25
员工 张三 的员工编号是: 2345
今天的幸运数字是:
1234 5678 2345 9876 4321 8765
恭喜 张三 中奖!
```

如果员工编号 `2345` 正好是幸运数字之一，则输出中奖信息。否则输出“很遗憾，没有中奖”。