

# 8. 数组的定义与使用

## 【本节目标】

1. 理解数组基本概念
2. 掌握数组的基本用法
3. 数组与方法互操作
4. 熟练掌握数组相关的常见问题和代码

## 1. 数组的基本概念

### 1.1 为什么要使用数组

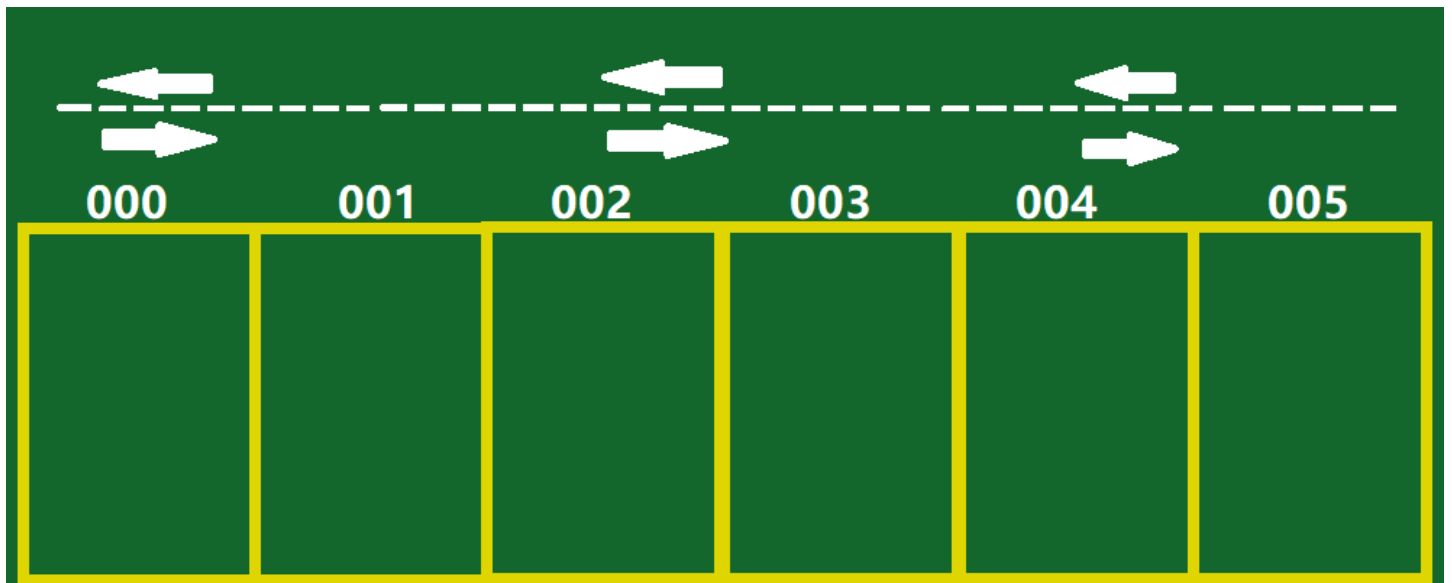
假设现在要存5个学生的javaSE考试成绩，并对其进行输出，按照之前掌握的知识点，我们会写出如下代码：

```
1  public class TestStudent{
2      public static void main(String[] args){
3          int score1 = 70;
4          int score2 = 80;
5          int score3 = 85;
6          int score4 = 60;
7          int score5 = 90;
8
9          System.out.println(score1);
10         System.out.println(score2);
11         System.out.println(score3);
12         System.out.println(score4);
13         System.out.println(score5);
14     }
15 }
```

上述代码没有任何问题，但不好是：如果有20名同学成绩呢，需要创建20个变量吗？有100个学生的成绩那不得要创建100个变量。仔细观察这些学生成绩发现：**所有成绩的类型都是相同的**，那Java中存在可以存储多个相同类型的数据吗？这就是本节要讲的数组。

### 1.2 什么是数组

数组：可以看成是**相同类型元素的一个集合**。在内存中是一段连续的空间。比如现实中的车库：



在Java中，包含6个整型类型元素的数组，就相当于上图中连在一起的6个车位，从上图中可以看到：

1. 数组的空间是连在一起的
2. 每个空间有自己的编号，起始位置的编号为0，即数组的下标。

那在程序中如何创建数组呢？

### 1.3 数组的创建及初始化

#### 数组的创建

```
1 T[] 数组名 = new T[N];
```

T：表示数组中存放元素的类型

T[]：表示数组的类型

N：表示数组的长度

```
1 int[] array1 = new int[10]; // 创建一个可以容纳10个int类型元素的数组
2 double[] array2 = new double[5]; // 创建一个可以容纳5个double类型元素的数组
3 String[] array3 = new double[3]; // 创建一个可以容纳3个字符串元素的数组
```

#### 数组的初始化

数组的初始化主要分为**动态初始化**以及**静态初始化**。

1. **动态初始化**：在创建数组时，直接指定数组中元素的个数

```
1 int[] array = new int[10];
```

## 2. 静态初始化：在创建数组时不直接指定数据元素个数，而直接将具体的数据内容进行指定

语法格式：`T[] 数组名称 = {data1, data2, data3, ..., datan};`

```
1  int[] array1 = new int[]{0,1,2,3,4,5,6,7,8,9};
2  double[] array2 = new double[]{1.0, 2.0, 3.0, 4.0, 5.0};
3  String[] array3 = new String[]{"hell", "Java", "!!!"};
```

### 【注意事项】

- 静态初始化虽然没有指定数组的长度，编译器在编译时会根据{}中元素个数来确定数组的长度。
- 静态初始化时, {}中数据类型必须与[]前数据类型一致。
- 静态初始化可以简写，省去后面的new T[]。

```
1  // 注意：虽然省去了new T[]，但是编译器编译代码时还是会还原
2  int[] array1 = {0,1,2,3,4,5,6,7,8,9};
3  double[] array2 = {1.0, 2.0, 3.0, 4.0, 5.0};
4  String[] array3 = {"hell", "Java", "!!!"};
```

- 数组也可以按照如下C语言方式创建，不推荐

```
1  /*
2  该种定义方式不太友好，容易造成数组的类型就是int的误解
3  []如果在类型之后，就表示数组类型，因此int[]结合在一块写意思更清晰
4  */
5  int arr[] = {1, 2, 3};
```

- 静态和动态初始化也可以分为两步，但是省略格式不可以。

```
1  int[] array1;
2  array1 = new int[10];
3
4  int[] array2;
5  array2 = new int[]{10, 20, 30};
6
7  // 注意省略格式不可以拆分，否则编译失败，这种方式只能在定义的同时初始化
8  // int[] array3;
9  // array3 = {1, 2, 3};
```

- 如果没有对数组进行初始化，数组中元素有其默认值
  - 如果数组中存储元素类型为基类类型，默认值为基类类型对应的默认值，比如：

类型	默认值
byte	0
short	0
int	0
long	0
float	0.0f
double	0
char	/u0000
boolean	FALSE

- 如果数组中存储元素类型为引用类型，默认值为null

```
1 String[] words = new String[3];
```

## 1.4 数组的基本使用

### 数组中元素访问

数组在内存中是一段连续的空间，空间的编号都是从0开始的，依次递增，该编号称为数组的下标，数组可以通过下标访问其任意位置的元素。比如：

```
1 int[] array = new int[]{10, 20, 30, 40, 50};
2 System.out.println(array[0]);
3 System.out.println(array[1]);
4 System.out.println(array[2]);
5 System.out.println(array[3]);
6 System.out.println(array[4]);
7
8 // 也可以通过[]对数组中的元素进行修改
9 array[0] = 100;
10 System.out.println(array[0]);
```

### 【注意事项】

1. 数组是一段连续的内存空间，因此**支持随机访问**，即通过下标访问快速访问数组中任意位置的元素
2. 下标从0开始，介于[0, N) 之间不包含N，N为元素个数，不能越界，否则会报出下标越界异常。

```
1  int[] array = {1, 2, 3};
2  System.out.println(array[3]); // 数组中只有3个元素，下标一次为：0 1 2，array[3]
   下标越界
3
4  // 执行结果
5  Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100
6      at Test.main(Test.java:4)
```

抛出了 `java.lang.ArrayIndexOutOfBoundsException` 异常. 使用数组一定要**注意下标**的访问谨防越界.

### 遍历数组

所谓 "遍历" 是指将数组中的所有元素都访问一遍, 访问是指对数组中的元素进行某种操作, 比如: 打印。

```
1  int[] array = new int[]{10, 20, 30, 40, 50};
2  System.out.println(array[0]);
3  System.out.println(array[1]);
4  System.out.println(array[2]);
5  System.out.println(array[3]);
6  System.out.println(array[4]);
```

上述代码可以起到对数组中元素遍历的目的，但问题是：

1. 如果数组中增加了一个元素，就需要增加一条打印语句
2. 如果输入中有100个元素，就需要写100个打印语句
3. 如果现在要把打印修改为给数组中每个元素加1，修改起来非常麻烦。

通过观察代码可以发现，对数组中每个元素的操作都是相同的，则可以使用循环来进行打印。

```
1  int[] array = new int[]{10, 20, 30, 40, 50};
2  for(int i = 0; i < 5; i++){
3      System.out.println(array[i]);
4  }
```

改成循环之后，上述三个缺陷可以全部2和3问题可以全部解决，但是无法解决问题1。那能否获取到数组的长度呢？

注意：在数组中可以通过 `数组对象.length` 来获取数组的长度

```
1  int[] array = new int[]{10, 20, 30, 40, 50};
2  for(int i = 0; i < array.length; i++){
3      System.out.println(array[i]);
4  }
```

也可以使用 for-each 遍历数组

```
1  int[] array = {1, 2, 3};
2  for (int x : array) {
3      System.out.println(x);
4  }
```

for-each 是 for 循环的另外一种使用方式. 能够更方便的完成对数组的遍历. 可以避免循环条件和更新语句写错.

## 2. 数组是引用类型

### 2.1 初始JVM的内存分布

内存是一段连续的存储空间，主要用来存储程序运行时数据的。比如：

1. 程序运行时代码需要加载到内存
2. 程序运行产生的中间数据要存放在内存
3. 程序中的常量也要保存
4. 有些数据可能需要长时间存储，而有些数据当方法运行结束后就要被销毁

如果对内存中存储的数据不加区分的随意存储，那对内存管理起来将会非常麻烦。比如：



因此JVM也对所使用的内存按照功能的不同进行了划分：



- **程序计数器 (PC Register):** 只是一个很小的空间, 保存下一条执行的指令的地址
- **虚拟机栈 (JVM Stack):** 与方法调用相关的一些信息, 每个方法在执行时, 都会先创建一个栈帧, 栈帧中包含有: 局部变量表、操作数栈、动态链接、返回地址以及其他的一些信息, 保存的都是与方法执行时相关的一些信息。比如: 局部变量。当方法运行结束后, 栈帧就被销毁了, 即栈帧中保存的数据也被销毁了。
- **本地方法栈 (Native Method Stack):** 本地方法栈与虚拟机栈的作用类似. 只不过保存的内容是 Native 方法的局部变量. 在有些版本的 JVM 实现中 (例如 HotSpot), 本地方法栈和虚拟机栈是一起的
- **堆 (Heap):** JVM 所管理的最大内存区域. 使用 `new` 创建的对象都是在堆上保存 (例如前面的 `new int[]{1, 2, 3}`), 堆是随着程序开始运行时而创建, 随着程序的退出而销毁, 堆中的数据只要还有在使用, 就不会被销毁。
- **方法区 (Method Area):** 用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据. 方法编译出的字节码就是保存在这个区域

现在我们只简单关心堆 和 虚拟机栈这两块空间, 后续JVM中还会更详细介绍。

## 2.2 基本类型变量与引用类型变量的区别

基本数据类型创建的变量, 称为基本变量, 该变量空间中直接存放的是其所对应的值;

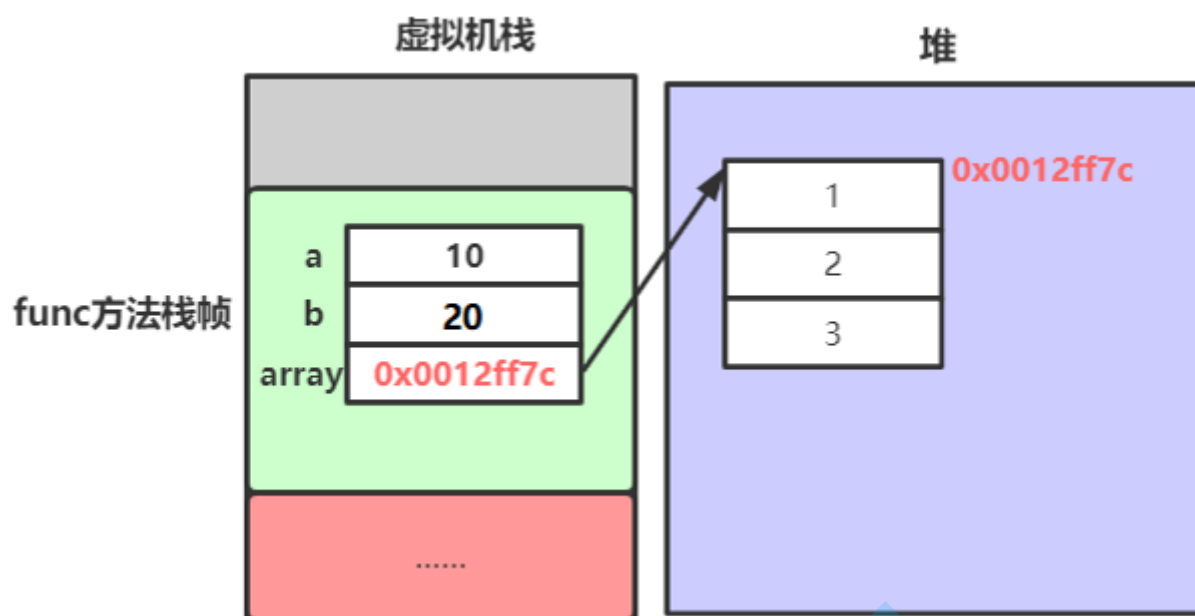
而引用数据类型创建的变量, 一般称为对象的引用, 其空间中存储的是对象所在空间的地址。

```
1 public static void func() {  
2     int a = 10;  
3     int b = 20;  
4     int[] arr = new int[]{1,2,3};  
5 }
```

在上述代码中, a、b、arr, 都是函数内部的变量, 因此其空间都在main方法对应的栈帧中分配。

a、b是内置类型的变量, 因此其空间中保存的就是给该变量初始化的值。

array是数组类型的引用变量，其内部保存的内容可以简单理解成是数组在堆空间中的首地址。

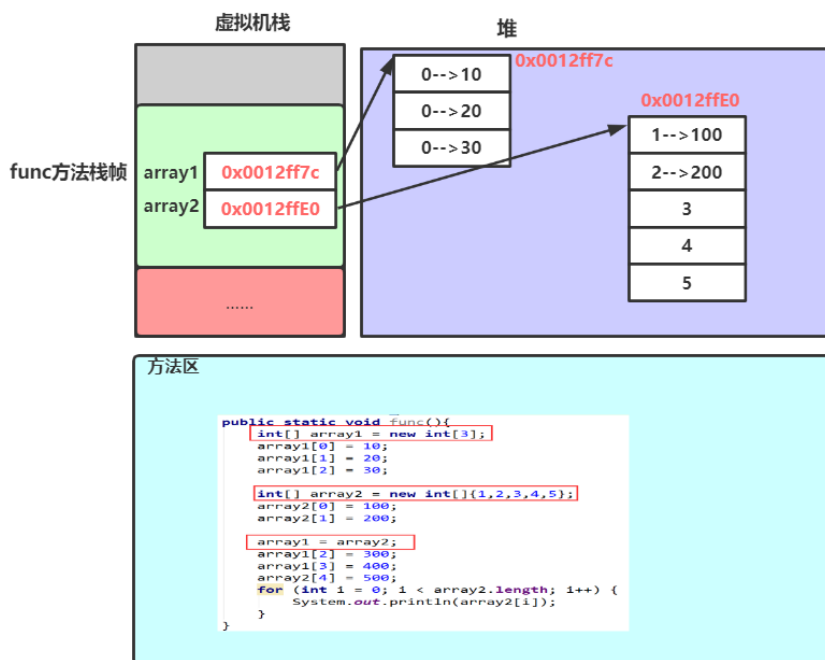


从上图可以看到，引用变量并不直接存储对象本身，可以简单理解成存储的是对象在堆中空间的起始地址。通过该地址，引用变量便可以去操作对象。有点类似C语言中的指针，但是Java中引用要比指针的操作更简单。

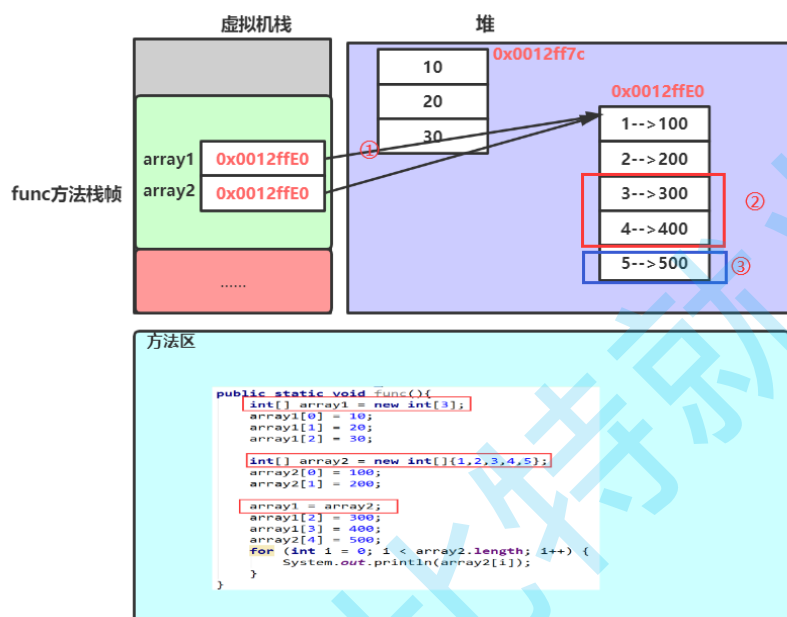
## 2.3 再谈引用变量

```
1 public static void func() {
2     int[] array1 = new int[3];
3     array1[0] = 10;
4     array1[1] = 20;
5     array1[2] = 30;
6
7     int[] array2 = new int[]{1,2,3,4,5};
8     array2[0] = 100;
9     array2[1] = 200;
10
11     array1 = array2;
12     array1[2] = 300;
13     array1[3] = 400;
14     array2[4] = 500;
15     for (int i = 0; i < array2.length; i++) {
16         System.out.println(array2[i]);
17     }
18 }
```





1. 创建数组array1，没有给数组元素设置初始值，因此每个位置都是0
2. 通过下标的方式将数组中每个元素修改成10、20、30
3. 创建数组arra2，并将其中元素设置为1、2、3、4、5
4. 通过下标方式将数组中前两个元素设置为100、200



1. array1 = array2，即让array1去引用array2引用的数组的空间，此时array1和array2实际是一个数组
2. 通过array1将数组2和3号位置元素修改为300、400，此时array2也能看到数组中修改的结果，因为array1和array2引用的是同一个数组
3. 通过array2将数组4号位置元素修改为500，此时array1也能看到数组中修改的结果，因为array1和array2引用的是同一个数组
4. 通过array2对数组中元素进行打印，输出100、200、300、400、500

## 2.4 认识 null

null 在 Java 中表示 "空引用", 也就是一个不指向对象的引用。

```

1  int[] arr = null;
2  System.out.println(arr[0]);
3
4  // 执行结果
5  Exception in thread "main" java.lang.NullPointerException
6      at Test.main(Test.java:6)

```

null 的作用类似于 C 语言中的 NULL (空指针), 都是表示一个无效的内存位置. 因此不能对这个内存进行任何读写操作. 一旦尝试读写, 就会抛出 NullPointerException.

注意: Java 中并没有约定 null 和 0 号地址的内存有任何关联.

## 3. 数组的应用场景

### 3.1 保存数据

```
1 public static void main(String[] args) {
2     int[] array = {1, 2, 3};
3
4     for(int i = 0; i < array.length; ++i){
5         System.out.println(array[i] + " ");
6     }
7 }
```

### 3.2 作为函数的参数

#### 参数传基本数据类型

```
1 public static void main(String[] args) {
2     int num = 0;
3     func(num);
4     System.out.println("num = " + num);
5 }
6
7 public static void func(int x) {
8     x = 10;
9     System.out.println("x = " + x);
10 }
11
12 // 执行结果
13 x = 10
14 num = 0
```

1. 发现在func方法中修改形参 x 的值, 不影响实参的 num 值.

#### 参数传数组类型(引用数据类型)

```
1 public static void main(String[] args) {
2     int[] arr = {1, 2, 3};
3     func(arr);
4     System.out.println("arr[0] = " + arr[0]);
5 }
```

```

6
7 public static void func(int[] a) {
8     a[0] = 10;
9     System.out.println("a[0] = " + a[0]);
10 }
11
12 // 执行结果
13 a[0] = 10
14 arr[0] = 10

```

发现在func方法内部修改数组的内容, 方法外部的数组内容也发生改变。

因为数组是引用类型, 按照引用类型来进行传递, 是可以修改其中存放的内容的。

**总结:** 所谓的 "引用" 本质上只是存了一个地址. Java 将数组设定成引用类型, 这样的话后续进行数组参数传参, 其实只是将数组的地址传入到函数形参中. 这样可以避免对整个数组的拷贝(数组可能比较长, 那么拷贝开销就会很大).

### 3.3 作为函数的返回值

比如: 获取斐波那契数列的前N项

```

1 public class TestArray {
2     public static int[] fib(int n){
3         if(n <= 0){
4             return null;
5         }
6
7         int[] array = new int[n];
8         array[0] = array[1] = 1;
9         for(int i = 2; i < n; ++i){
10             array[i] = array[i-1] + array[i-2];
11         }
12
13         return array;
14     }
15
16     public static void main(String[] args) {
17         int[] array = fib(10);
18         for (int i = 0; i < array.length; i++) {
19             System.out.println(array[i]);
20         }
21     }
22 }

```

## 4. 操作数据工具类Arrays与数组练习

### 4.1 数组转字符串

```
1  import java.util.Arrays
2
3  int[] arr = {1,2,3,4,5,6};
4
5  String newArr = Arrays.toString(arr);
6  System.out.println(newArr);
7
8  // 执行结果
9  [1, 2, 3, 4, 5, 6]
```

使用这个方法后续打印数组就更方便一些。

Java 中提供了 `java.util.Arrays` 包, 其中包含了一些操作数组的常用方法。

### 4.2 数组拷贝

```
1  import java.util.Arrays;
2
3  public static void func(){
4      // newArr和arr引用的是同一个数组
5      // 因此newArr修改空间中内容之后, arr也可以看到修改的结果
6      int[] arr = {1,2,3,4,5,6};
7      int[] newArr = arr;
8      newArr[0] = 10;
9      System.out.println("newArr: " + Arrays.toString(arr));
10
11     // 使用Arrays中copyOf方法完成数组的拷贝:
12     // copyOf方法在进行数组拷贝时, 创建了一个新的数组
13     // arr和newArr引用的不是同一个数组
14     arr[0] = 1;
15     newArr = Arrays.copyOf(arr, arr.length);
16     System.out.println("newArr: " + Arrays.toString(newArr));
17
18     // 因为arr修改其引用数组中内容时, 对newArr没有任何影响
19     arr[0] = 10;
20     System.out.println("arr: " + Arrays.toString(arr));
21     System.out.println("newArr: " + Arrays.toString(newArr));
22
23     // 拷贝某个范围.
24     int[] newArr2 = Arrays.copyOfRange(arr, 2, 4);
```

```
25     System.out.println("newArr2: " + Arrays.toString(newArr2));
26 }
```

**注意：**数组当中存储的是基本类型数据时，不论怎么拷贝基本都不会出现什么问题，但如果存储的是引用数据类型，拷贝时需要考虑深浅拷贝的问题，关于深浅拷贝在后续详细给大家介绍。

### 实现自己版本的拷贝数组

```
1  public static int[] copyOf(int[] arr) {
2      int[] ret = new int[arr.length];
3      for (int i = 0; i < arr.length; i++) {
4          ret[i] = arr[i];
5      }
6      return ret;
7  }
```

## 4.3 查找数组中指定元素(顺序查找)

给定一个数组, 再给定一个元素, 找出该元素在数组中的位置.

### 代码示例

```
1  public static void main(String[] args) {
2      int[] arr = {1,2,3,10,5,6};
3      System.out.println(find(arr, 10));
4  }
5
6  public static int find(int[] arr, int data) {
7      for (int i = 0; i < arr.length; i++) {
8          if (arr[i] == data) {
9              return i;
10         }
11     }
12     return -1; // 表示没有找到
13 }
14
15 // 执行结果
16 3
```

## 4.4 查找数组中指定元素(二分查找)

针对**有序数组**, 可以使用更高效的二分查找.

啥叫有序数组?

有序分为 "升序" 和 "降序"

如 1 2 3 4, 依次递增即为升序.

如 4 3 2 1, 依次递减即为降序.

以升序数组为例, 二分查找的思路是先取中间位置的元素, 然后使用待查找元素与数组中间元素进行比较:

- 如果相等, 即找到了返回该元素在数组中的下标
- 如果小于, 以类似方式到数组左半侧查找
- 如果大于, 以类似方式到数组右半侧查找

### 代码示例

```
1  public static void main(String[] args) {
2      int[] arr = {1,2,3,4,5,6};
3      System.out.println(binarySearch(arr, 6));
4  }
5
6  public static int binarySearch(int[] arr, int toFind) {
7      int left = 0;
8      int right = arr.length - 1;
9      while (left <= right) {
10         int mid = (left + right) / 2;
11         if (toFind < arr[mid]) {
12             // 去左侧区间找
13             right = mid - 1;
14         } else if (toFind > arr[mid]) {
15             // 去右侧区间找
16             left = mid + 1;
17         } else {
18             // 相等, 说明找到了
19             return mid;
20         }
21     }
22     // 循环结束, 说明没找到
23     return -1;
24 }
25
26 // 执行结果
27 5
```

可以看到, 针对一个长度为 10000 个元素的数组查找, 二分查找只需要循环 14 次就能完成查找. 随着数组元素个数越多, 二分的优势就越大.

## 4.5 数组排序(冒泡排序)

给定一个数组, 让数组升序 (降序) 排序.

### 算法思路

假设排升序:

1. 将数组中相邻元素从前往后依次进行比较, 如果前一个元素比后一个元素大, 则交换, 一趟下来后最大元素就在数组的末尾
2. 依次从上上述过程, 直到数组中所有的元素都排列好

```
1  public static void main(String[] args) {
2      int[] arr = {9, 5, 2, 7};
3      bubbleSort(arr);
4      System.out.println(Arrays.toString(arr));
5  }
6
7  public static void bubbleSort(int[] arr) {
8      for (int i = 0; i < arr.length; i++) {
9          for (int j = 1; j < arr.length-i; j++) {
10             if (arr[j-1] > arr[j]) {
11                 int tmp = arr[j - 1];
12                 arr[j - 1] = arr[j];
13                 arr[j] = tmp;
14             }
15         }
16     } // end for
17 } // end bubbleSort
18
19 // 执行结果
20 [2, 5, 7, 9]
```

冒泡排序性能较低. Java 中内置了更高效的排序算法

```
1  public static void main(String[] args) {
2      int[] arr = {9, 5, 2, 7};
3      Arrays.sort(arr);
4      System.out.println(Arrays.toString(arr));
5  }
```

关于 Arrays.sort 的具体实现算法, 我们在后面的排序算法课上再详细介绍. 到时候我们会介绍很多种常见排序算法.

## 4.6 数组逆序

给定一个数组, 将里面的元素逆序排列.

### 思路

设定两个下标, 分别指向第一个元素和最后一个元素. 交换两个位置的元素.

然后让前一个下标自增, 后一个下标自减, 循环继续即可.

### 代码示例

```
1  public static void main(String[] args) {
2      int[] arr = {1, 2, 3, 4};
3      reverse(arr);
4      System.out.println(Arrays.toString(arr));
5  }
6
7  public static void reverse(int[] arr) {
8      int left = 0;
9      int right = arr.length - 1;
10     while (left < right) {
11         int tmp = arr[left];
12         arr[left] = arr[right];
13         arr[right] = tmp;
14         left++;
15         right--;
16     }
17 }
```

## 5. 二维数组

### 5.1 普通的二维数组

二维数组本质上也就是一维数组, 只不过每个元素又是一个一维数组.

### 基本语法

```
1  数据类型[][] 数组名称 = new 数据类型 [行数][列数] { 初始化数据 };
```

行不可以省略, 列可以省略

### 代码示例



```

1  int[][] arr = {
2      {1, 2, 3, 4},
3      {5, 6, 7, 8},
4      {9, 10, 11, 12}
5  };
6
7  for (int row = 0; row < arr.length; row++) {
8      for (int col = 0; col < arr[row].length; col++) {
9          System.out.printf("%d\t", arr[row][col]);
10     }
11     System.out.println("");
12 }
13
14 // 执行结果
15 1      2      3      4
16 5      6      7      8
17 9      10     11     12

```

- 二维数组是特殊的一维数组，一维数组的每个元素又是一个数组
- arr.length 代表二维数组的行数
- arr[row].length 代表二维数组的列数

## 5.2 不规则的二维数组

不规则的二维数组指的是，二维数组的列在定义的时候，没有确定。

```

1  int[][] array = new int[2][];
2  array[0] = new int[3];
3  array[1] = new int[5];

```

上述二维数组就不是一个规则的二维数组。第1行有3列，第2行有5列。

