Java 中的每一个数组存储的数据类型是一致的。

Java 的数组的确是放在一块连续内存里的，否则不可能做到在 O(1) 时间复杂度内存取元素。

基本类型的数据都非常小，可以直接放在数组里，这跟 C 里面的数组是一样的；

但引用类型的对象就不一样了，存在数组里的都只是引用，不是真正的对象数据。我们通过数组拿到的还是引用，真正的对象分散地存在堆里，并不是连续的。

数组是静态的

Java 语言是典型的静态语言，因此 Java 数组是静态的，即当数组被初始化之后，该数组 所占的内存空间、 数组长度 都是不可变的。Java 程序中的数组必须经过初始化才可使用。所谓初始化，即创建实际的数组对象，也就是在内存中为数组对象分配内存空间，并为每个数组 元素指定初始值。

数组的初始化有以下两种方式。

- **静态初始化：初始化时由程序员显式指定每个数组元素的初始值，由系统决定数组长度。**
- **动态初始化：初始化时程序员只指定数组长度，由系统为数组元素分配初始值。**

不管采用哪种方式初始化Java 数组，一旦初始化完成，该数组的长度就不可改变，

Java 语言允许通过数组的length 属性来访问数组的长度。示例如下。

```
1. 公共类数组测试
2. {
3. 公共静态无效主要（String [] args）
4. {
5. // 采用静态初始化方式初始化第一个数组
6. String[] 书籍 = 新 String[]
7. { “1”, “2”, “3”, “4”
8. };
9. // 采用静态初始化的简化形式初始化第二个数组
10. 字符串[] 名称 =
11. {
12. "孙悟空",
13. "猪八戒",
14. "白骨精"
15. };
16. // 采用动态初始化的语法初始化第三个数组
17. 字符串[] strArr = 新字符串[5];
18. // 访问三个数组的长度
19. System.out.println("第一个数组的长度：" + books.length);
```

```
20. System.out.println("第二个数组的长度：" + names.length);
21. System.out.println("第三个数组的长度：" + strArr.length);
22. }
23. }
```

上面程序中的粗体字代码声明并初始化了三个数组。这三个数组的长度将会始终不变，程 序输出三个数组的长度依次为4 、3 、5 。

前面已经指出，Java 语言的数组变量是引用类型的变量，books、names 、strArr 这三个变量，以及各自引用的数组在内存中的分配示意图如图1.1 所示。

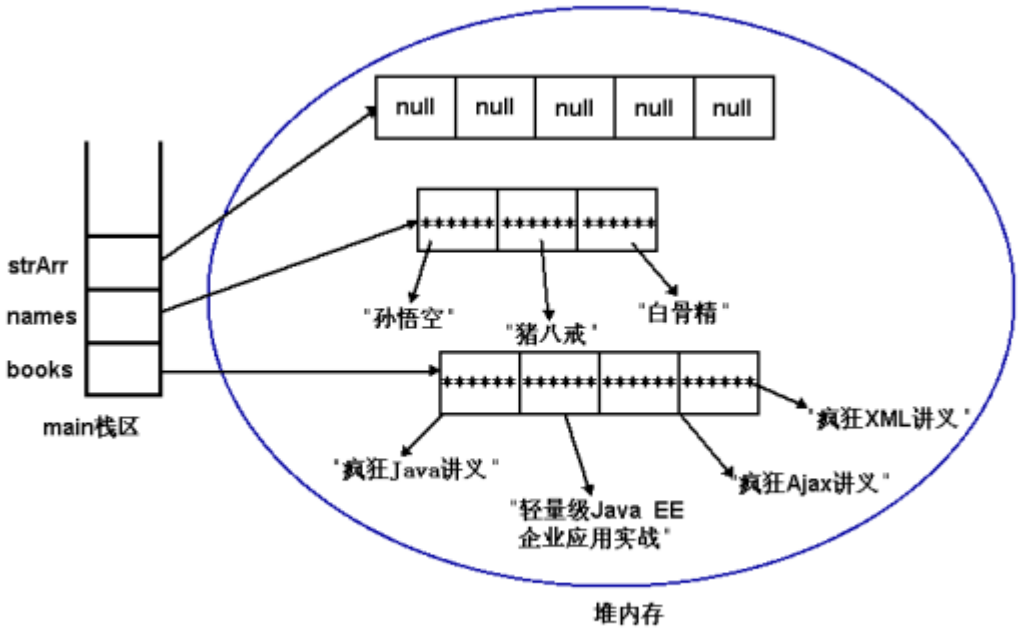

图 1.1　数组在内存中的分配示意图 1

从图1.1可以看出，对于静态初始化方式而言，程序员无须指定数组长度，指定该数组的 数组元素，由系统来决定该数组的长度即可。例如 books 数组，为它指定了四个数组元素，它 的长度就是4；对于names 数组，为它指定了三个元素，它的长度就是3 。

执行动态初始化时，程序员只需指定数组的长度，即为每个数组元素指定所需的内存空间， 系统将负责为这些数组元素分配初始值。指定初始值时，系统将按如下规则分配初始值。

- 数组元素的类型是基本类型中的整数类型（byte 、short、int 和long ），则数组元素的值是0 。
- 数组元素的类型是基本类型中的浮点类型（float 、double ），则数组元素的值是0.0。
- 数组元素的类型是基本类型中的字符类型（char ），则数组元素的值是'\u0000'。
- 数组元素的类型是基本类型中的布尔类型（boolean），则数组元素的值是false 。
- 数组元素的类型是引用类型（类、接口和数组），则数组元素的值是null 。
  Java 数组是静态的，一旦数组初始化完成，数组元素的内存空间分配即结束，程序只能改变数组元素的值，而无法改变数组的长度。

需要指出的是，**Java 的数组变量是一种引用类型的变量，数组变量并不是数组本身，它只是指向堆内存中的数组对象。**因此，可以改变一个数组变量所引用的数组，这样可以造成数 组长度可变的假

象。假设，在上面程序的后面增加如下几行。

```
1. // 让books 数组变量、strArr 数组变量指向names 所引用的数组
2. 书籍 = 名称;
3. strArr = 名称;
4. 系统.out.println(“----------”);
5. System.out.println("books 数组的长度：" + books.length);
6. System.out.println("strArr 数组的长度：" + strArr.length);
7. // 改变books 数组变量所引用的数组的第二个元素值
8. books[1] = "唐僧";
9. System.out.println("names 数组的第二个元素是：" + books[1]);
```

上面程序中粗体字代码将让books 数组变量、strArr 数组变量都指向names 数组变量所引 用的数组，这样做的结果就是books、strArr、names 这三个变量引用同一个数组对象。此时， 三个引用变量和数组对象在内存中的分配示意图如图1.2 所示。

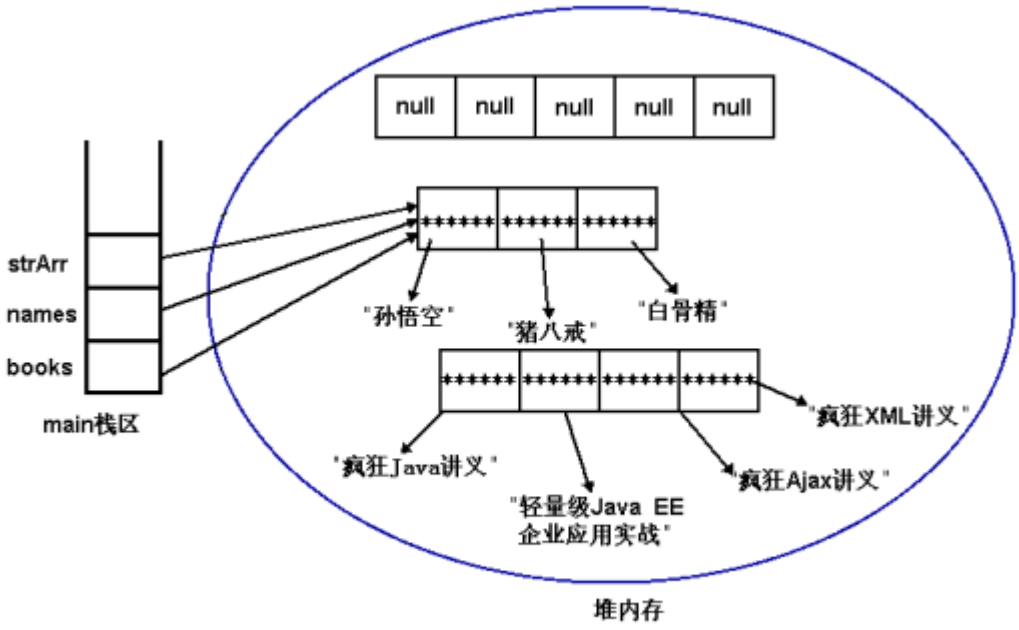

图 1.2　数组在内存中的分配示意图 2

从图1.2可以看出，此时 strArr、names 和books 数组变量实际上引用了同一个数组对象。 因此，当访问 books 数组、strArr 数组的长度时，将看到输出 3。这很容易造成一个假象：books 数组的长度从4 变成了3。实际上，数组对象本身的长度并没有发生改变，只是 books 数组变量发生了改变。books 数组变量原本指向图 1.2下面的数组，当执行了books = names;语句之后，books 数组将改为指向图1.2 中间的数组，而原来books 变量所引用的数组的长度依然是4 。

从图1.2 还可以看出，原来 books 变量所引用的数组的长度依然是 4 ，但不再有任何引用 变量引用该数组，因此它将会变成垃圾，等着垃圾回收机制来回收。此时，程序使用books、 names 和strArr 这三个变量时，将会访问同一个数组对象，因此把 books 数组的第二个元素赋 值为“唐僧”时，names 数组的第二个元素的值也会随之改变。

基本类型数组的初始化

对于基本类型数组而言，数组元素的值直接存储在对应的数组元素中，因此基本类型 数组的初始化比较简单：程序直接先为数组分配内存空间，再将数组元素的值存入对应内 存里。

下面程序采用静态初始化方式初始化了一个基本类型的数组对象。

```
1. 公共类 PrimitiveArrayTest
2. {
3. 公共静态无效主要（String [] args）
4. {
5. // 定义一个int[] 类型的数组变量
6. int[] iArr;
7. // 静态初始化数组，数组长度为4
8. iArr = new int[]{2, 5, -12, 20};
9. }
10. }
```

上面代码的执行过程代表了基本类型数组初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行第一行代码int[] iArr;时，仅定义一个数组变量，此时内存中的存储示意图如图1.4所示。

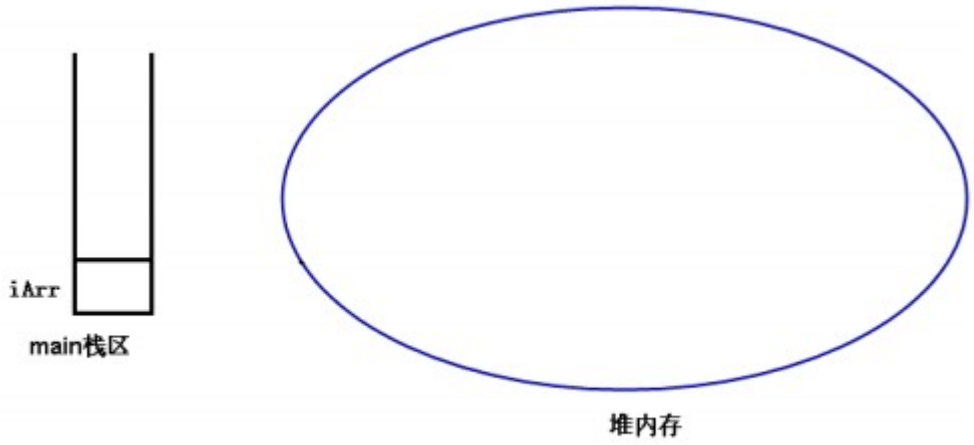

图 1.4　定义 iArr 数组变量后的存储示意图

执行了int[] iArr; 代码后，仅在 main 方法栈中定义了一个 iArr 数组变量，它是一个引用类 型的变量，并未指向任何有效的内存，没有真正指向实际的数组对象。此时还不能使用该数组 对象。

当执行iArr = new int[]{2,5,-12,20}; 静态初始化后，系统会根据程序员指定的数组元素来决 定数组的长度。此时指定了四个数组元素，系统将创建一个长度为4 的数组对象，一旦该数组 对象创建成功，该数组的长度将不可改变，程序只能改变数组元素的值。此时内存中的存储示 意图如图1.5 所示。

静态初始化完成后，iArr 数组变量引用的数组所占用的内存空间被固定下来，程序员只能 改变各数组元素内的值。既不能移动该数组所占用的内存空间，也不能扩大该数组对象所占用 的内存，或缩减该数组对象所占用的内存。

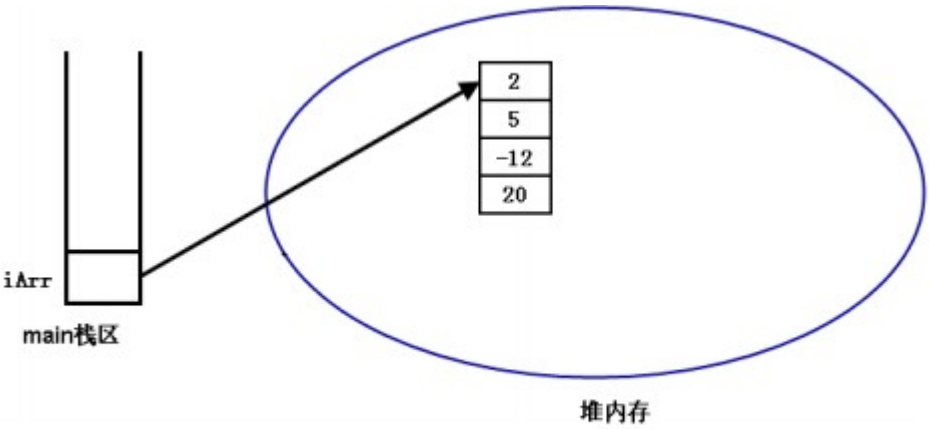

图 1.5　静态初始化 iArr 数组后的存储示意图

对于程序运行过程中的变量，可以将它们形容为具体的瓶子——瓶子可以存储 水，而变量用于存储值，也就是数据。对于强类型语言如Java ，它有一个要求： 怎样的瓶子只能装怎样的水，也就是说，指定类型的变量只能存储指定类型的值。

所有局部变量都是放在栈内存里保存的，不管其是基本类型的变量，还 是引用类型的变量，都是存储在各自的方法栈内存中的；但引用类型的变量所引用的对象（包 括数组、普通的Java 对象）则总是存储在堆内存中。

对于Java 语言而言，堆内存中的对象（不管是数组对象，还是普通的 Java 对象）通常不 允许直接访问，为了访问堆内存中的对象，通常只能通过引用变量。这也是很容易混淆的地方。 例如，iArr 本质上只是main 栈区的引用变量，但使用 iArr.length 、iArr[2] 时，系统将会自动变 为访问堆内存中的数组对象。

对于很多Java 程序员而言，他们最容易混淆的是：引用类型的变量何时只是栈内存中的 变量本身，何时又变为引用实际的Java 对象。其实规则很简单：引用变量本质上只是一个指 针，只要程序通过引用变量访问属性，或者通过引用变量来调用方法，该引用变量就会由它所 引用的对象代替。

```
1. 公共类 PrimitiveArrayTest2
2. {
3. 公共静态无效主要（String [] args）
4. {
5. // 定义一个int[] 类型的数组变量
6. int[] iArr = null;
7. // 只要不访问iArr 的属性和方法，程序完全可以使用该数组变量
8. System.out.println(iArr);//①
9. // 动态初始化数组，数组长度为5
10. iArr = 新 int[5];
```

```
11. // 只有当iArr 指向有效的数组对象后，下面才可访问iArr 的属性
12. System.out.println(iArr.length);//②
13. }
14. }
```

上面程序中两行粗体字代码两次访问iArr 变量。对于①行代码而言，虽然此时的iArr 数 组变量并未引用到有效的数组对象，但程序在①行代码处并不会出现任何问题，因为此时并未 通过iArr 访问属性或调用方法，因此程序只是访问iArr 引用变量本身，并不会去访问iArr 所 引用的数组对象。对于②行代码而言，此时程序通过iArr 访问了length 属性，程序将自动变 为访问iArr 所引用的数组对象，这就要求iArr 必须引用一个有效的对象。

有过一些编程经验，应该经常看到一个Runtime 异常： NullPointerException （空指针异常）。当通过引用变量来访问实例属性，或者调 用非静态方法时，如果该引用变量还未引用一个有效的对象，程序就会引发 NullPointerException 运行时异常。

引用类型数组的初始化

引用类型数组的数组元素依然是引用类型的，因此数组元素里存储的还是引用，它指向另一块内存，这块内存里存储了该引用变量所引用的对象（包括数组和Java 对象）。

为了说明引用类型数组的运行过程，下面程序先定义一个Person 类，然后定义一个 Person[]数组，并动态初始化该Person[]数组，再显式地为数组的不同数组元素指定值。该程序代码如下。

```
1. 人员类
2. {
3. // 年龄
4. 公共 int 年龄；
5. // 身高
6. 公共双层高度；
7. // 定义一个info 方法
8. 公共无效信息（）
9. {
10. System.out.println("我的年龄是：" + age
11. + "，我的身高是：" + height);
12. }
13. }
14. 公共类 ReferenceArrayTest
15. {
16. 公共静态无效主要（String [] args）
17. {
```

```
18. // 定义一个students 数组变量，其类型是Person[]
19. 人[]学生；
20. // 执行动态初始化
21. 学生=新人[2];
22. System.out.println("students所引用的数组的长度是："
23. +学生.长度); //①
24. // 创建一个Person 实例，并将这个Person 实例赋给zhang 变量
25. Person zhang = new Person();
26. // 为zhang 所引用的Person 对象的属性赋值
27. 张.年龄=15;
28. 张.身高=158；
29. // 创建一个Person 实例，并将这个Person 实例赋给lee 变量
30. Person lee = new Person();
31. // 为lee 所引用的Person 对象的属性赋值
32. 李.年龄=16；
33. 李.身高=161；
34. // 将zhang 变量的值赋给第一个数组元素
35. 学生[0] = 张;
36. // 将lee 变量的值赋给第二个数组元素
37. 学生[1] = 李；
38. // 下面两行代码的结果完全一样，
39. // 因为lee 和students[1]指向的是同一个Person 实例
40. lee. 信息();
41. 学生[1].信息()；
42. }
43. }
```

上面代码的执行过程代表了引用类型数组的初始化的典型过程。下面将结合示意图详细介绍这段代码的执行过程。

执行Person[] students ;代码时，这行代码仅仅在栈内存中定义了一个引用变量，也就是一个指针，这个指针并未指向任何有效的内存区。此时内存中的存储示意图如图1.6 所示。

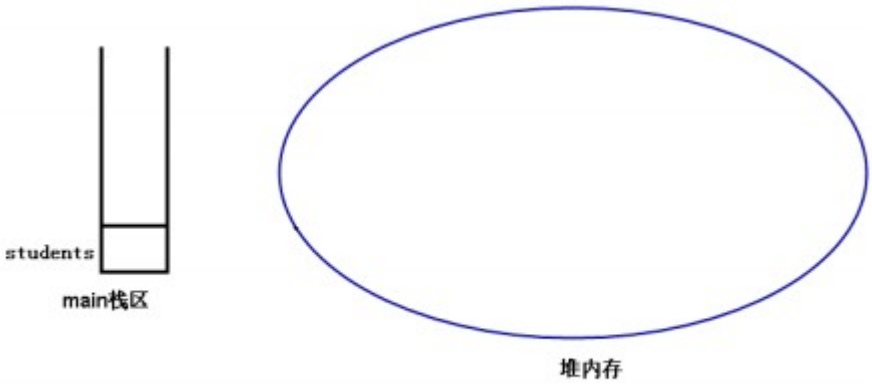

图 1.6　定义一个 students 数组变量后的存储示意图

在图1.6中的栈内存中定义了一个 students 变量，它仅仅是一个空引用，并未指向任何有 效的内存，直到执行初始化，本程序对 students 数组执行动态初始化。动态初始化由系统为数 组元素分配默认的初始值null ，即每个数组元素的值都是 null 。执行动态初始化后的存储示意 图如图1.7 所示。

从图1.7 中可以看出，students 数组的两个数组元素都是引用，而且这两个引用并未指 向任何有效的内存，因此，每个数组元素的值都是 null 。此时，程序可以通过students 来 访问它所引用的数组的属性，因此在①行代码处通过 students 访问了该数组的长度，此时 将输出2 。

students 数组是引用类型的数组，因此 students[0] 、students[1] 两个数组元素相当于两个引 用类型的变量。如果程序只是直接输出这两个引用类型的变量，那么程序完全正常。但程序依 然不能通过students[0] 、students[1] 来调用属性或方法，因此它们还未指向任何有效的内存区， 所以这两个连续的Person 变量（students 数组的数组元素）还不能被使用。

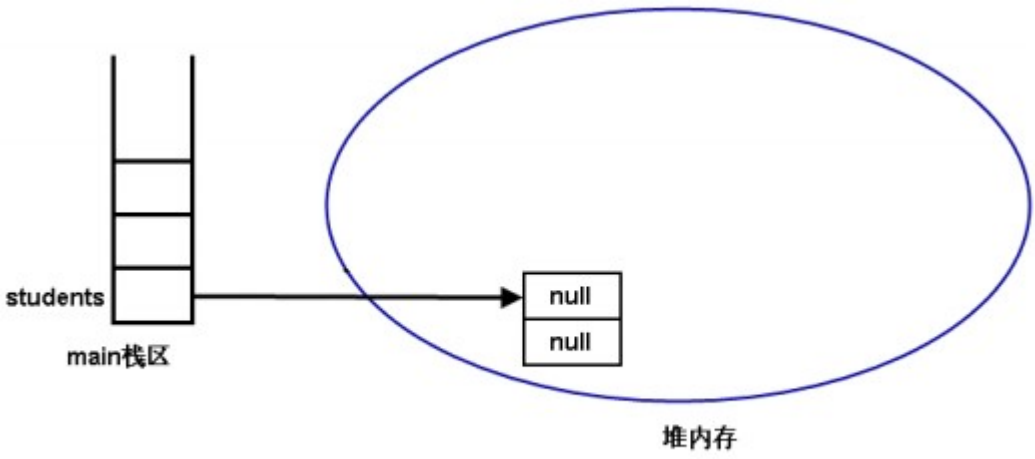

图 1.7　动态初始化 students 数组后的存储示意图

接着，程序定义了zhang 和lee 两个引用变量，并让它们指向堆内存中的两个Person 对象，此时的 zhang、lee 两个引用变量存储在 main 方法栈区中，而两个 Person 对象则存储在堆内存中。此时的内存存储示意图如图1.8 所示。

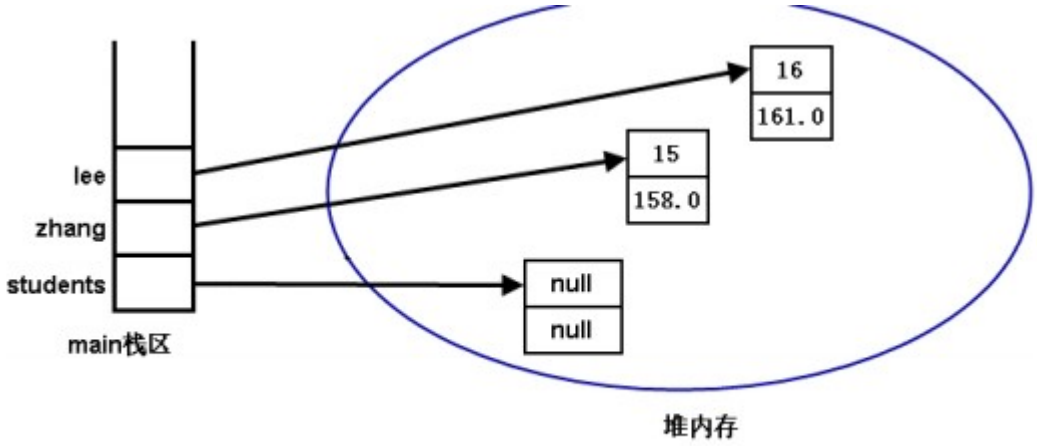

图 1.8　创建两个 Person 实例后的存储示意图

对于zhang、lee 两个引用变量来说，它们可以指向任何有效的Person 对象，而students[0] 、students[1] 也可以指向任何有效的Person 对象。从本质上来看，zhang、lee、students[0] 、students[1] 能够存储的内容完全相同。接着，程序执行students[0] = zhang;和students[1] = lee; 两行代码， 也就是让zhang 和students[0] 指向同一个 Person 对象，让 lee 和students[1] 指向同一个 Person 对象。此时的内存存储示意图如图1.9 所示。

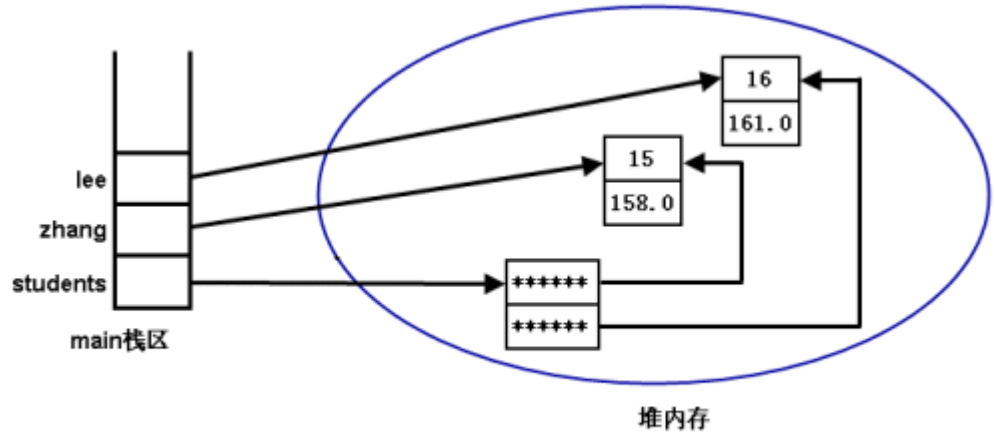

图 1.9　为数组元素赋值后的存储示意图

从图1.9 中可以看出，此时 zhang 和students[0] 指向同一个内存区，而且它们都是引用类 型的变量，因此通过 zhang 和students[0] 来访问Person 实例的属性和方法的效果完全一样。不 论修改 students[0] 所指向的 Person 实例的属性，还是修改 zhang 变量所指向的 Person 实例的 属性，所修改的其实是同一个内存区，所以必然互相影响。同理，lee 和students[1] 也是引用 到同一个 Person 对象，也有相同的效果。

前面已经提到，对于引用类型的数组而言，它的数组元素其实就是一个引用类型的变量， 因此可以指向任何有效的内存——此处“有效”的意思是指强类型的约束。比如，对 Person[] 类型的数组而言，它的每个数组元素都相当于Person 类型的变量，因此它的数组元素只能指 向Person 对象。