

课时 01

程序世界中的基石：数组

1. 数组的内存模型
2. “高效”的访问与“低效”的插入删除

数组的内存模型

在计算机科学中

数组可以被定义为是一组被保存在连续存储空间中，并且具有相同类型的数据元素集合



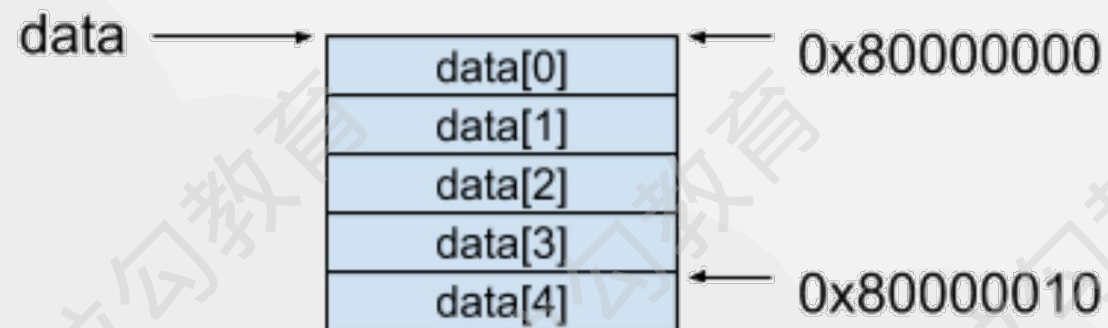
数组的内存模型

一维数组

数组的内存模型举例

定义 5 个元素的 int 数组: `int[] data = new int[5];`

整个 data 数组在计算机内存中分配的模型如图所示



数组的内存模型

一维数组

这种分配连续空间的内存模型同时也揭示了数组在数据结构中的另外一个特性，即随机访问（Random Access）

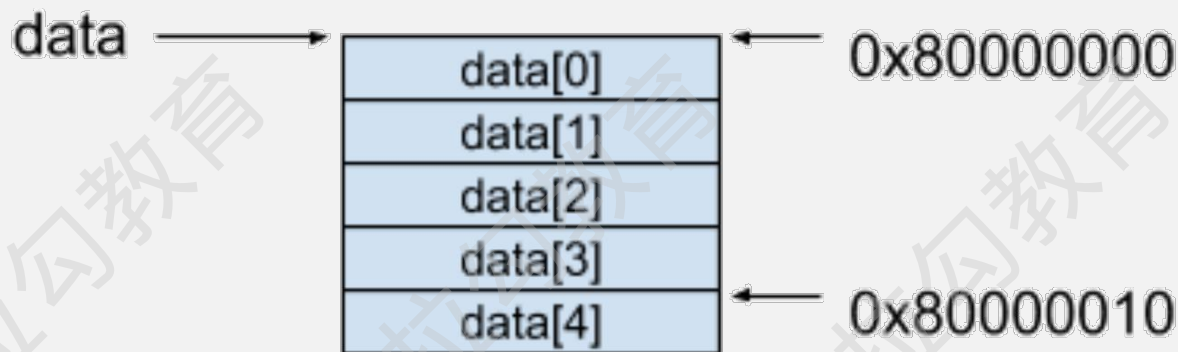
随机访问这个概念在计算机科学中被定义为：可以用同等的时间访问到一组数据中的任意一个元素

疑惑：为什么在访问数组中的第一个元素时，程序一般都是表达成这样的：`data[0]`

是因为获取数组元素的方式是按照这个公式进行获取的： $\text{base_address} + \text{index} \times \text{data_size}$

数组的内存模型

一维数组



`data` 这个数组被分配到的起始地址是 `0x80000000`，是因为 `int` 类型数据占据了 4 个字节的空间

如果我们要访问第 5 个元素 `data[4]` 的时候，按照上面的公式 (`base_address + index × data_size`)

只需要取得 `0x80000000 + 4 × 4 = 0x80000010` 这个地址的内容就可以了

数组的内存模型

二维数组

声明一个二维数组：`int[][] data = new int[2][3];`

这个二维数组在内存中的寻址方式又是怎样的呢？

这其实涉及到计算机内存到底是以行优先（Row-Major Order）还是以列优先（Column-Major Order）存储的

	0	1	2
0	1	2	3
1	4	5	6

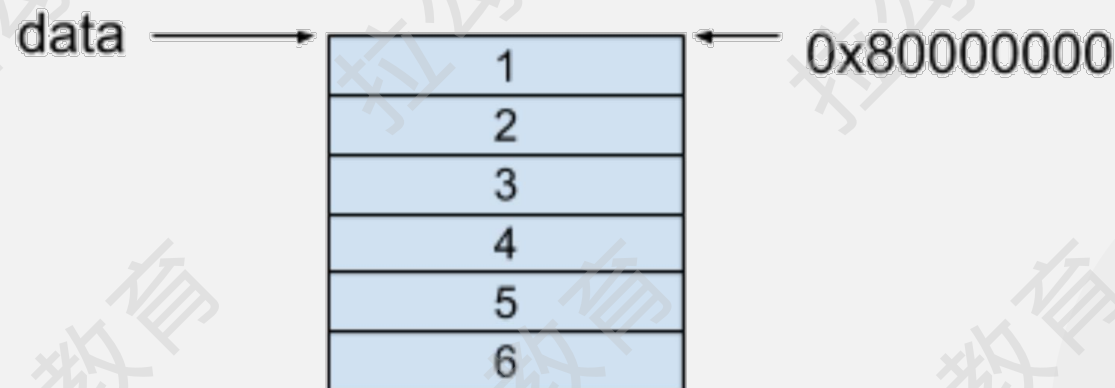
数组的内存模型

二维数组——行优先

行优先的内存模型保证了每一行的每个相邻元素都保存在了相邻的连续内存中

	0	1	2
0	1	2	3
1	4	5	6

二维数组的内存模型



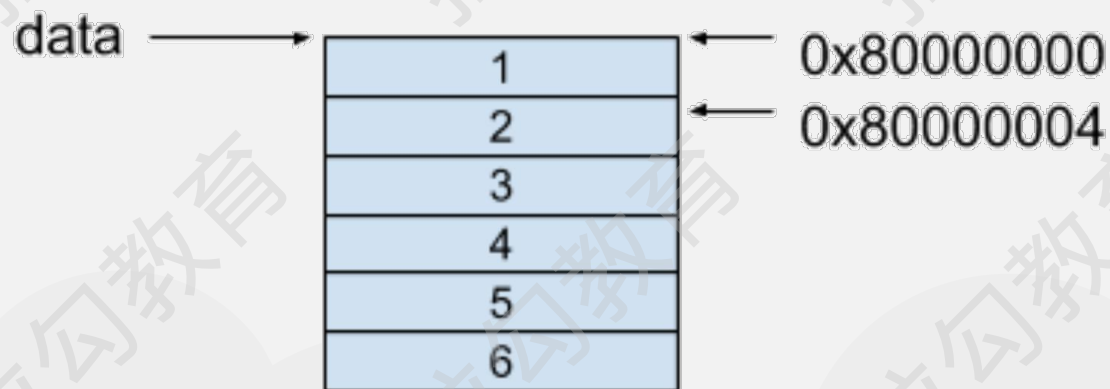
数组的内存模型

二维数组——行优先

获取数组元素的方式是按照以下的公式进行获取的：

$$\text{base_address} + \text{data_size} \times (i \times \text{number_of_column} + j)$$

当访问 `data[0][1]` 这个值时，套用上面的公式，得到的值，即 `0x80000004` 地址的值，也就是 2



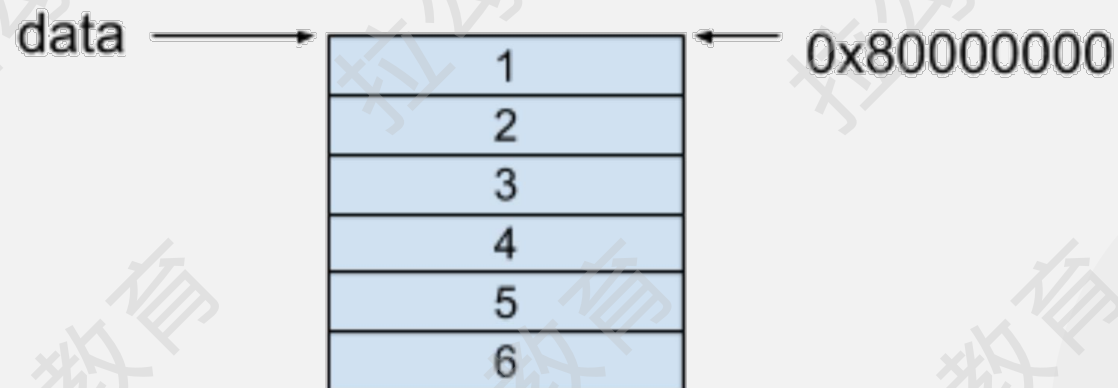
数组的内存模型

二维数组——列优先

列优先的内存模型保证了每一行的每个相邻元素都保存在了相邻的连续内存中

	0	1	2
0	1	2	3
1	4	5	6

二维数组的内存模型



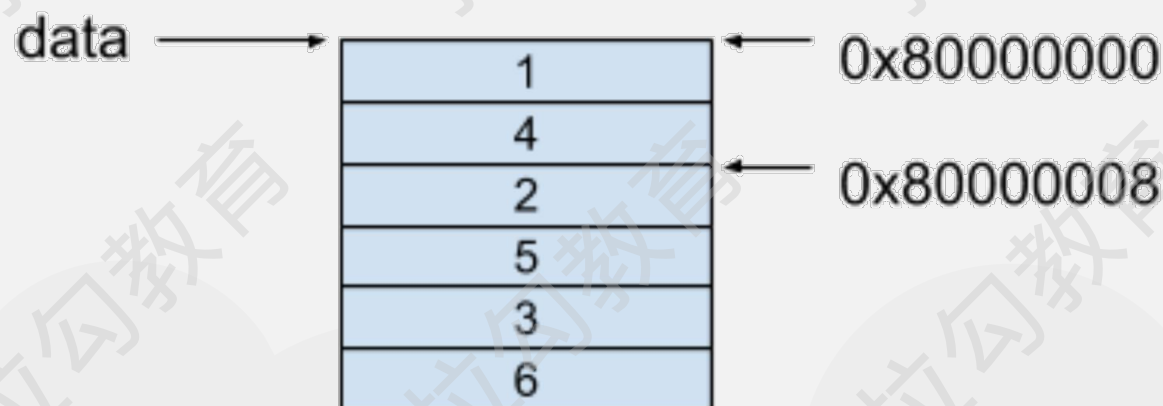
数组的内存模型

二维数组——列优先

获取数组元素的方式是按照以下的公式进行获取的：

$$\text{base_address} + \text{data_size} \times (\text{i} + \text{number_of_row} \times \text{j})$$

当访问 `data[0][1]` 这个值时，套用上面的公式，得到的值，即 `0x80000008` 地址的值，也就是 2



数组的内存模型

多维数组

声明一个三维数组: `int[][][] data = new int[2][3][4];`

行优先: $\text{base_address} + \text{data_size} \times (D_n + S_n \times (D_{n-1} + S_{n-1} \times (D_{n-2} + S_{n-2} \times (\dots + S_2 \times (D_1) \dots))))$

列优先: $\text{base_address} + \text{data_size} \times (D_1 + (S_1 \times (D_2 + S_2 \times (D_3 + S_3 \times (\dots + S_{n-1} \times D_n) \dots))))$

数组的内存模型

在 CPU 读取程序指定地址的数值时，CPU 会把和它地址相邻的一些数据也一并读取并放到更高一级的缓存中

比如类似 L1 或者 L2 缓存

当数据存放在这种缓存上的时候，读取的速度有可能会比直接读取内存的速度快 10 倍以上

如果知道了数据存放的内存模型是行优先的话，在设计数据结构的时候，会更倾向于读取每一行上的数据

因为每一行的数据在内存中都是保存在相邻位置的，它们更有可能被一起读取到 CPU 缓存中

反之，我们更倾向于读取每一列上的数据



“高效”的访问与“低效”的插入删除

访问一个数组中的元素采用的是随机访问的方式

只要按照上面提到的寻址方式来获取相应位置的数值便可，所以访问数组元素的时间复杂度是 $O(1)$

对于保存基本类型的数组来说，它们的内存大小在一开始就已经确定好了，称为静态数组 (Static Array)

静态数组的大小无法改变，所以无法对这种数组进行插入或者删除操作

但在使用高级语言的时候，如 Java，Java 中的 ArrayList 这种 Collection 是提供了像 add 和 remove 这样的

API 来进行插入和删除操作，这样的数组称为动态数组 (Dynamic Array)



“高效”的访问与“低效”的插入删除

add(int index, E element) 函数源码

```
public void add(int index, E  
element) {  
    System.arraycopy(elementData,  
        index,
```

```
        elementData, index +  
            1,
```

```
        index);
```

```
    elementData[index] =
```

add 函数调用了一个 System.arraycopy 的函数进行内存操作

s 在这里代表了 ArrayList 的 size

当我们调用 add 函数的时候

函数在实现的过程中到底发生了什么呢


“高效”的访问与“低效”的插入删除

add(int index, E element) 函数源码

Index	0	1	2	3	4	5
ElementData	1	2	3	0	0	0



Index	0	1	2	3	4	5
ElementData	1	2	2	3	0	0



Index	0	1	2	3	4	5
ElementData	1	4	2	3	0	0

“高效”的访问与“低效”的插入删除

remove(int index) 函数源码

```
public void remove(int index) {  
    ...  
    fastRemove(es, index);  
    ...  
}  
private void fastRemove(Object[]  
    es, int i) {  
    ...  
    System.arraycopy(es, i + 1,  
        es, i, newSize - i);  
    ...  
}
```


“高效”的访问与“低效”的插入删除

remove(int index) 函数源码

Index	0	1	2	3	4	5
ElementData	1	2	3	0	0	0



Index	0	1	2	3	4	5
ElementData	1	3	0	0	0	0

小结

- 读取数组的时间复杂度为 $O(1)$
- 通过分析 Java Openjdk-jdk11，知道了插入和删除数组元素的时间复杂度为 $O(n)$

