05|数组：为什么很多编程语言中数组都从0开始编号？

提到数组，我想你肯定不陌生，甚至还会自信地说，它很简单啊。

是的，在每一种编程语言中，基本都会有数组这种数据类型。不过，它不仅仅是一种编程语言中的数据类型，还是一种最基础的数据结构。尽管数组看起来非常基础、简单，但是我估计很多人都并没有理解这个基础数据结构的精髓。

在大部分编程语言中，数组都是从0开始编号的，但你是否下意识地想过，为什么数组要从**0**开始编号，而不是从**1**开始呢？ 从1开始不是更符合人类的思维习惯吗？

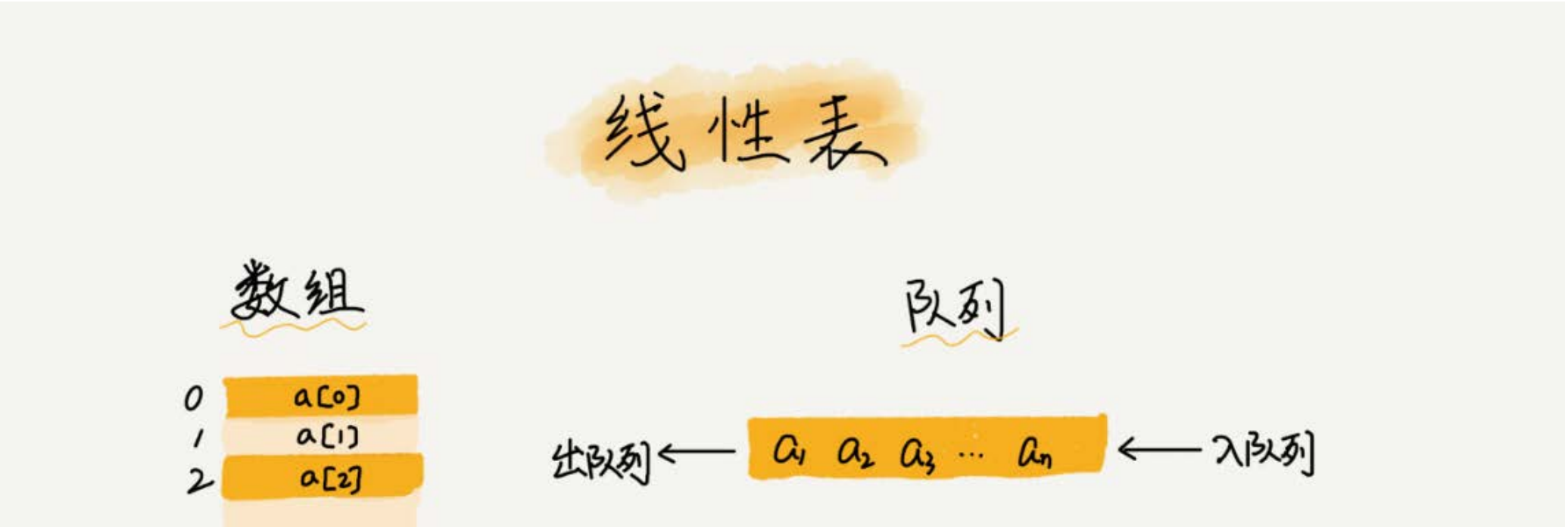
你可以带着这个问题来学习接下来的内容。

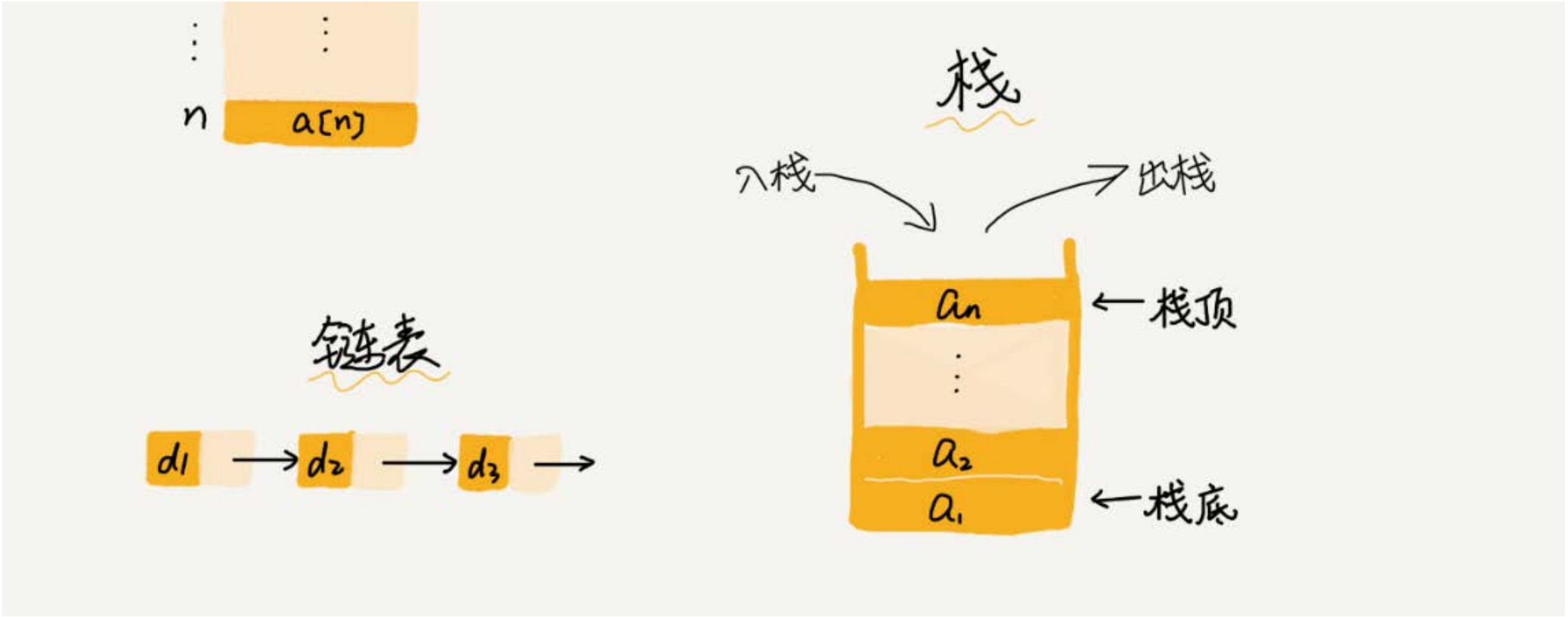
如何实现随机访问？

什么是数组？我估计你心中已经有了答案。不过，我还是想用专业的话来给你做下解释。数组（**Array**）是一种线性表数据结构。它用一组连续的内存空间，来存储一组具有相同类型的数据。

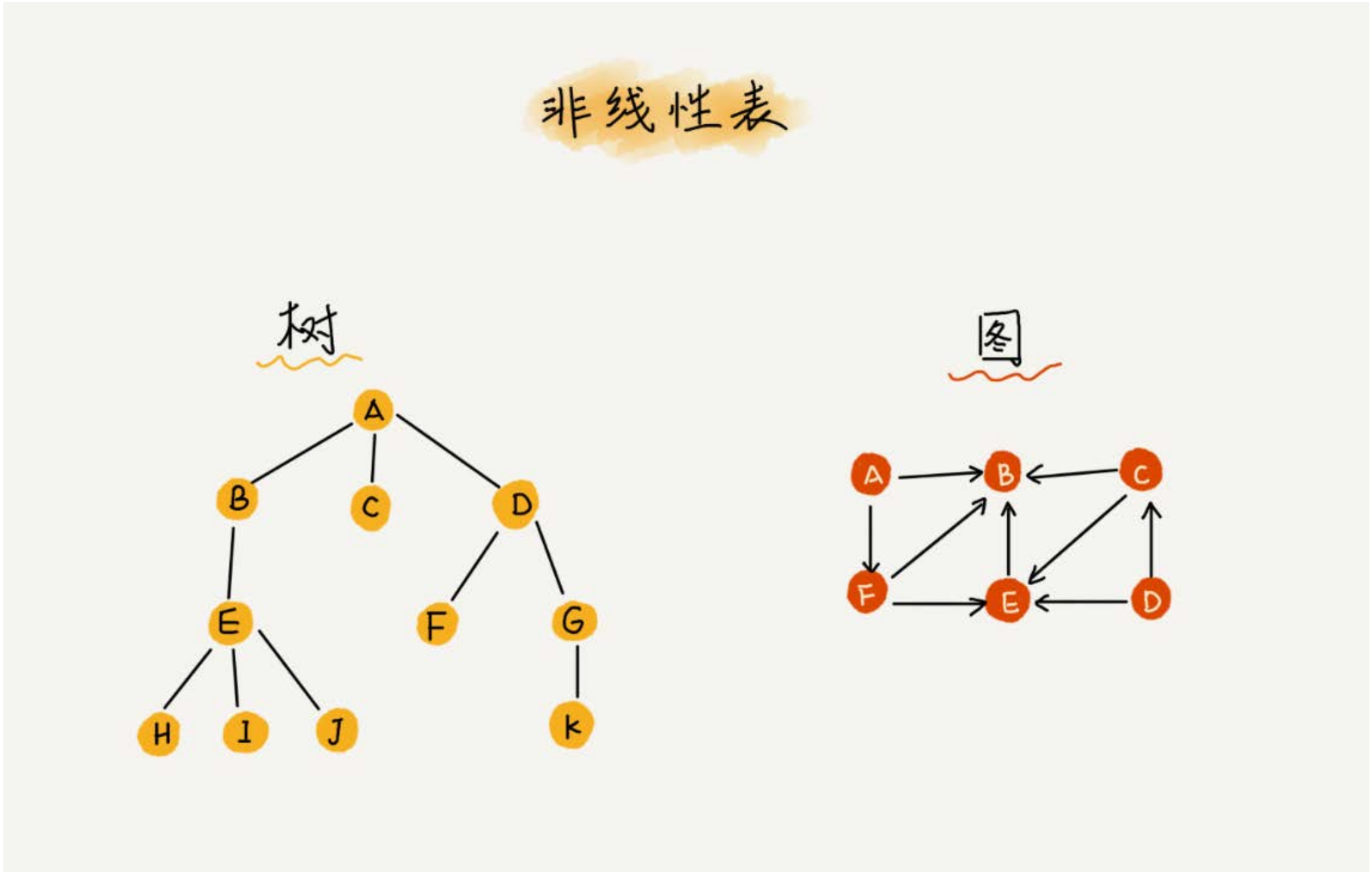
这个定义里有几个关键词，理解了这几个关键词，我想你就能彻底掌握数组的概念了。下面就从我的角度分别给你“点拨”一下。

第一是线性表（Linear List）。顾名思义，线性表就是数据排成像一条线一样的结构。每个线性表上的数据最多只有前和后两个方向。其实除了数组，链表、队列、栈等也是线性表结构。





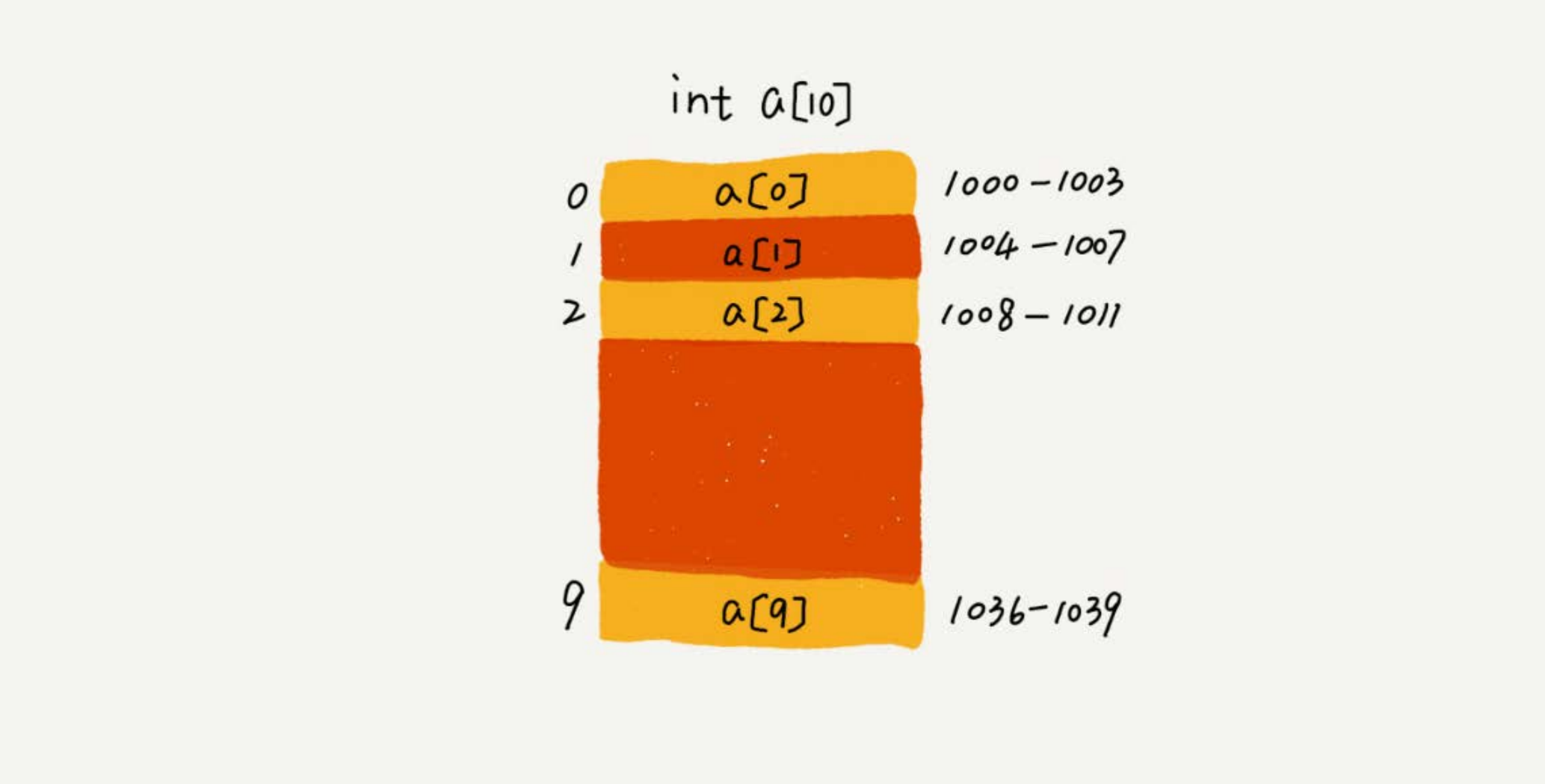
而与它相对立的概念是非线性表，比如二叉树、堆、图等。之所以叫非线性，是因为，在非线性表中，数据之间并不是简单的前后关系。



第二个是连续的内存空间和相同类型的数据。正是因为这两个限制，它才有了一个堪称“杀手锏”的特性：“随机访问”。但有利就有弊，这两个限制也让数组的很多操作变得非常低效，比如要想在数组中删除、插入一个数据，为了保证连续性，就需要做大量的数据搬移工作。

说到数据的访问，那你知道数组是如何实现根据下标随机访问数组元素的吗？

我们拿一个长度为10的int类型的数组int[] a = new int[10]来举例。在我画的这个图中，计算机给数组a[10]，分配了一块连续内存空间1000 ~ 1039，其中，内存块的首地址为base_address = 1000。



我们知道，计算机会给每个内存单元分配一个地址，计算机通过地址来访问内存中的数据。当计算机需要随机访问数组中的某个元素时，它会首先通过下面的寻址公式，计算出该元素存储的内存地址：

```
a[i]_address = base_address + i * data_type_size
```

其中data_type_size表示数组中每个元素的大小。我们举的这个例子里，数组中存储的是int类型数据，所以data_type_size就为4个字节。这个公式非常简单，我就不多做解释了。

这里我要特别纠正一个“错误”。我在面试的时候，常常会问数组和链表的区别，很多人都回答说，“链表适合插入、删除，时间复杂度O(1)；数组适合查找，查找

时间复杂度为O(1)”。

实际上，这种表述是不准确的。数组是适合查找操作，但是查找的时间复杂度并不为O(1)。即便是排好序的数组，你用二分查找，时间复杂度也是O(logn)。所以，正确的表述应该是，数组支持随机访问，根据下标随机访问的时间复杂度为O(1)。

低效的“插入”和“删除”

前面概念部分我们提到，数组为了保持内存数据的连续性，会导致插入、删除这两个操作比较低效。现在我们就来详细说一下，究竟为什么会导致低效？又有哪些改进方法呢？

我们先来看插入操作。

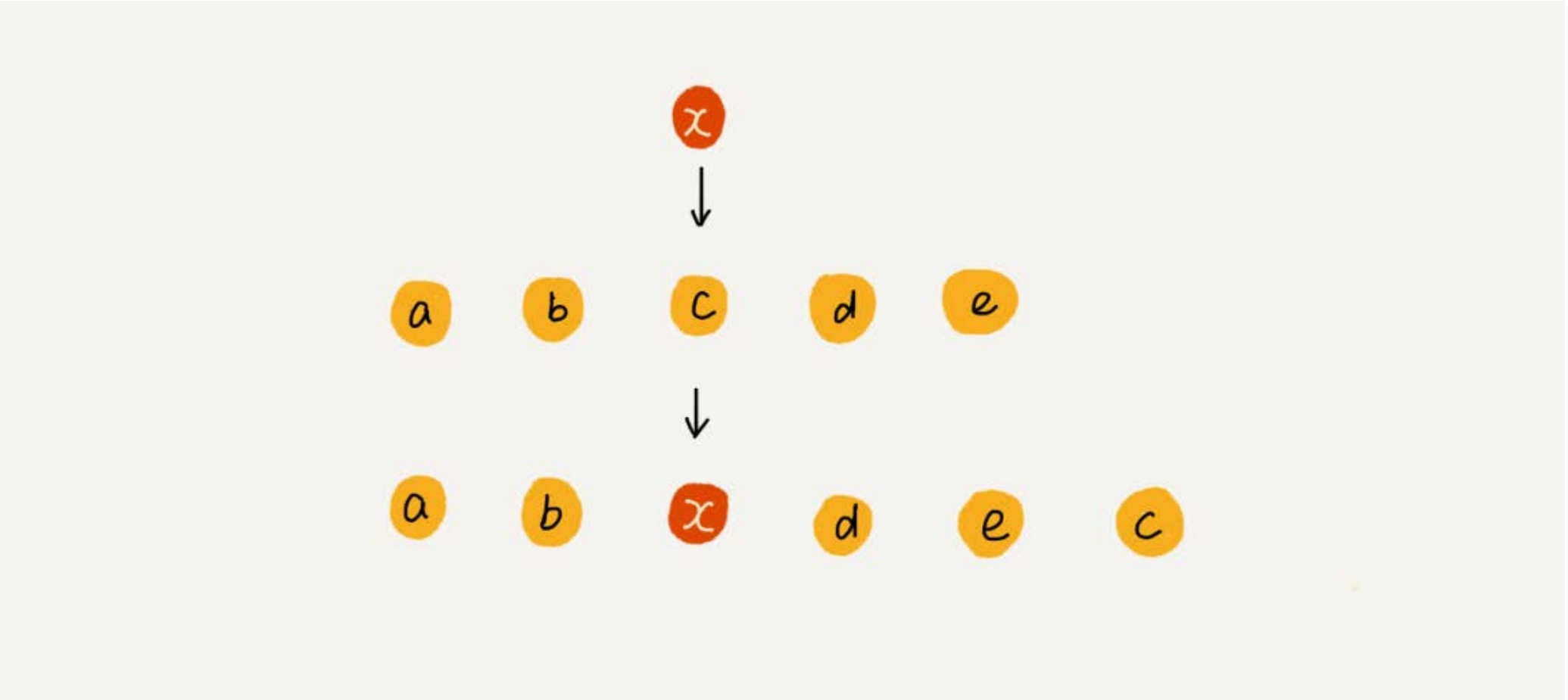
假设数组的长度为n，现在，如果我们需要将一个数据插入到数组中的第k个位置。为了把第k个位置腾出来，给新来的数据，我们需要将第k ~ n这部分的元素都顺序地往后挪一位。那插入操作的时间复杂度是多少呢？你可以自己先试着分析一下。

如果在数组的末尾插入元素，那就不需要移动数据了，这时的时间复杂度为O(1)。但如果在数组的开头插入元素，那所有的数据都需要依次往后移动一位，所以最坏时间复杂度是O(n)。因为我们在每个位置插入元素的概率是一样的，所以平均情况时间复杂度为(1+2+…n)/n=O(n)。

如果数组中的数据是有序的，我们在某个位置插入一个新的元素时，就必须按照刚才的方法搬移k之后的数据。但是，如果数组中存储的数据并没有任何规律，数组只是被当作一个存储数据的集合。在这种情况下，如果要将某个数组插入到第k个位置，为了避免大规模的数据搬移，我们还有一个简单的办法就是，直接将第k位的数据搬移到数组元素的最后，把新的元素直接放入第k个位置。

为了更好地理解，我们举一个例子。假设数组a[10]中存储了如下5个元素：a，b，c，d，e。

我们现在需要将元素x插入到第3个位置。我们只需要将c放入到a[5]，将a[2]赋值为x即可。最后，数组中的元素如下：a，b，x，d，e，c。



利用这种处理技巧，在特定场景下，在第k个位置插入一个元素的时间复杂度就会降为O(1)。这个处理思想在快排中也会用到，我会在排序那一节具体来讲，这里就说到这儿。

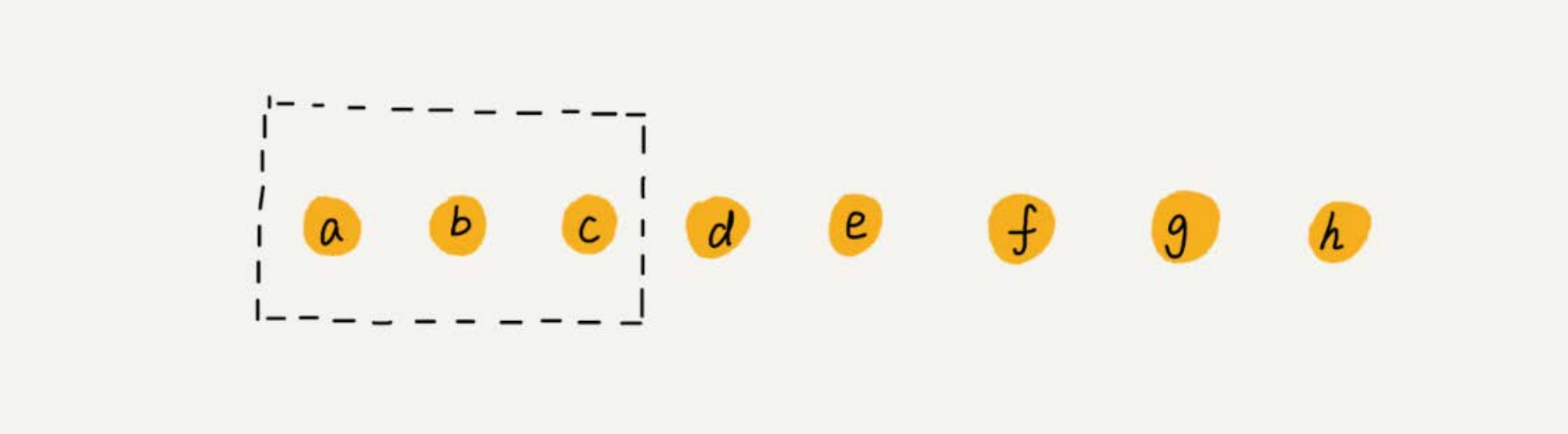
我们再来看删除操作。

跟插入数据类似，如果我们要删除第k个位置的数据，为了内存的连续性，也需要搬移数据，不然中间就会出现空洞，内存就不连续了。

和插入类似，如果删除数组末尾的数据，则最好情况时间复杂度为O(1)；如果删除开头的数据，则最坏情况时间复杂度为O(n)；平均情况时间复杂度也为O(n)。

实际上，在某些特殊场景下，我们并不一定非得追求数组中数据的连续性。如果我们将多次删除操作集中在一起执行，删除的效率是不是会提高很多呢？

我们继续来看例子。数组a[10]中存储了8个元素：a，b，c，d，e，f，g，h。现在，我们要依次删除a，b，c三个元素。



为了避免d，e，f，g，h这几个数据会被搬移三次，我们可以先记录下已经删除的数据。每次的删除操作并不是真正地搬移数据，只是记录数据已经被删除。当数组没有更多空间存储数据时，我们再触发执行一次真正的删除操作，这样就大大减少了删除操作导致的数据搬移。

如果你了解JVM，你会发现，这不就是JVM标记清除垃圾回收算法的核心思想吗？没错，数据结构和算法的魅力就在于此，很多时候我们并不是要去死记硬背某个数据结构或者算法，而是要学习它背后的思想和处理技巧，这些东西才是最有价值的。如果你细心留意，不管是在软件开发还是架构设计中，总能找到某些算法和数据结构的影子。

警惕数组的访问越界问题

了解了数组的几个基本操作后，我们来聊聊数组访问越界的问题。

首先，我请你来分析一下这段C语言代码的运行结果：

```
int main(int argc, char* argv[]){
    int i = 0;
    int arr[3] = {0};
    for(; i<=3; i++){
        arr[i] = 0;
        printf("hello world\n");
    }
    return 0;
}
```

你发现问题了吗？这段代码的运行结果并非是打印三行“hello word”，而是会无限打印“hello world”，这是为什么呢？

因为，数组大小为3，a[0]，a[1]，a[2]，而我们的代码因为书写错误，导致for循环的结束条件错写为了i<=3而非i<3，所以当i=3时，数组a[3]访问越界。

我们知道，在C语言中，只要不是访问受限的内存，所有的内存空间都是可以自由访问的。根据我们前面讲的数组寻址公式，a[3]也会被定位到某块不属于数组的内存地址上，而这个地址正好是存储变量i的内存地址，那么a[3]=0就相当于i=0，所以就会导致代码无限循环。

数组越界在C语言中是一种未决行为，并没有规定数组访问越界时编译器应该如何处理。因为，访问数组的本质就是访问一段连续内存，只要数组通过偏移计算得到的内存地址是可用的，那么程序就可能不会报任何错误。

这种情况下，一般都会出现莫名其妙的逻辑错误，就像我们刚刚举的那个例子，debug的难度非常的大。而且，很多计算机病毒也正是利用到了代码中的数组越界可以访问非法地址的漏洞，来攻击系统，所以写代码的时候一定要警惕数组越界。

但并非所有的语言都像C一样，把数组越界检查的工作丢给程序员来做，像Java本身就会做越界检查，比如下面这几行Java代码，就会抛出java.lang.ArrayIndexOutOfBoundsException。

```
int[] a = new int[3];
a[3] = 10;
```

容器能否完全替代数组？

针对数组类型，很多语言都提供了容器类，比如Java中的ArrayList、C++ STL中的vector。在项目开发中，什么时候适合用数组，什么时候适合用容器呢？

这里我拿Java语言来举例。如果你是Java工程师，几乎天天都在用ArrayList，对它应该非常熟悉。那它与数组相比，到底有哪些优势呢？

我个人觉得，ArrayList最大的优势就是可以将很多数组操作的细节封装起来。比如前面提到的数组插入、删除数据时需要搬移其他数据等。另外，它还有一个优势，就是支持动态扩容。

数组本身在定义的时候需要预先指定大小，因为需要分配连续的内存空间。如果我们申请了大小为10的数组，当第11个数据需要存储到数组中时，我们就需要重新分配一块更大的空间，将原来的数据复制过去，然后再将新的数据插入。

如果使用ArrayList，我们就完全不需要关心底层的扩容逻辑，ArrayList已经帮我们实现好了。每次存储空间不够的时候，它都会将空间自动扩容为1.5倍大小。

不过，这里需要注意一点，因为扩容操作涉及内存申请和数据搬移，是比较耗时的。所以，如果事先能确定需要存储的数据大小，最好在创建**ArrayList**的时候事先指定数据大小。

比如我们要从数据库中取出10000条数据放入ArrayList。我们看下面这几行代码，你会发现，相比之下，事先指定数据大小可以省掉很多次内存申请和数据搬移操作。

```
ArrayList<User> users = new ArrayList(10000);
for (int i = 0; i < 10000; ++i) {
  users.add(xxx);
}
```

作为高级语言编程者，是不是数组就无用武之地了呢？当然不是，有些时候，用数组会更合适些，我总结了几点自己的经验。

1.Java ArrayList无法存储基本类型，比如int、long，需要封装为Integer、Long类，而Autoboxing、Unboxing则有一定的性能消耗，所以如果特别关注性能，或者希望使用基本类型，就可以选用数组。

2.如果数据大小事先已知，并且对数据的操作非常简单，用不到ArrayList提供的大部分方法，也可以直接使用数组。

3.还有一个是我个人的喜好，当要表示多维数组时，用数组往往会更加直观。比如Object[][] array；而用容器的话则需要这样定义：ArrayList<ArrayList > array。

我总结一下，对于业务开发，直接使用容器就足够了，省时省力。毕竟损耗一丢丢性能，完全不会影响到系统整体的性能。但如果你是做一些非常底层的开发，比如开发网络框架，性能的优化需要做到极致，这个时候数组就会优于容器，成为首选。

解答开篇

现在我们来思考开篇的问题：为什么大多数编程语言中，数组要从0开始编号，而不是从1开始呢？

从数组存储的内存模型上来看，“下标”最确切的定义应该是“偏移（offset）”。前面也讲到，如果用a来表示数组的首地址，a[0]就是偏移为0的位置，也就是首地址，a[k]就表示偏移k个type_size的位置，所以计算a[k]的内存地址只需要用这个公式：

```
a[k]_address = base_address + k * type_size
```

但是，如果数组从1开始计数，那我们计算数组元素a[k]的内存地址就会变为：

```
a[k]_address = base_address + (k-1)*type_size
```

对比两个公式，我们不难发现，从1开始编号，每次随机访问数组元素都多了一次减法运算，对于CPU来说，就是多了一次减法指令。

数组作为非常基础的数据结构，通过下标随机访问数组元素又是其非常基础的编程操作，效率的优化就要尽可能做到极致。所以为了减少一次减法操作，数组选择了从0开始编号，而不是从1开始。

不过我认为，上面解释得再多其实都算不上压倒性的证明，说数组起始编号非0开始不可。所以我觉得最主要的原因可能是历史原因。

C语言设计者用0开始计数数组下标，之后的Java、JavaScript等高级语言都效仿了C语言，或者说，为了在一定程度上减少C语言程序员学习Java的学习成本，因此继续沿用了从0开始计数的习惯。实际上，很多语言中数组也并不是从0开始计数的，比如Matlab。甚至还有一些语言支持负数下标，比如Python。

内容小结

我们今天学习了数组。它可以说是最基础、最简单的数据结构了。数组用一块连续的内存空间，来存储相同类型的一组数据，最大的特点就是支持随机访问，但插入、删除操作也因此变得比较低效，平均情况时间复杂度为O(n)。在平时的业务开发中，我们可以直接使用编程语言提供的容器类，但是，如果是特别底层的开发，直接使用数组可能会更合适。

课后思考

1. 前面我基于数组的原理引出JVM的标记清除垃圾回收算法的核心理念。我不知道你是否使用Java语言，理解JVM，如果你熟悉，可以在评论区回顾下你理解的标记清除垃圾回收算法。
2. 前面我们讲到一维数组的内存寻址公式，那你可以思考一下，类比一下，二维数组的内存寻址公式是怎样的呢？

欢迎留言和我分享，我会第一时间给你反馈。

我已将本节内容相关的详细代码更新到GitHub，戳此即可查看。

精选留言：

- 杰杰 2018-10-01 04:05:27
  JVM标记清除算法：

  大多数主流虚拟机采用可达性分析算法来判断对象是否存活，在标记阶段，会遍历所有 GC ROOTS，将所有 GC ROOTS 可达的对象标记为存活。只有当标

记工作完成后，清理工作才会开始。

不足：1.效率问题。标记和清理效率都不高，但是当知道只有少量垃圾产生时会很高效。2.空间问题。会产生不连续的内存空间碎片。

二维数组内存寻址：

对于 m * n 的数组，a [i][j] (i < m,j < n)的地址为：

address = base_address + (i * n + j) * type_size

另外，对于数组访问越界造成无限循环，我理解是编译器的问题，对于不同的编译器，在内存分配时，会按照内存地址递增或递减的方式进行分配。老师的程序，如果是内存地址递减的方式，就会造成无限循环。

不知我的解答和理解是否正确，望老师解答？

[290赞]

作者回复2018-10-01 14:57:30
完全正确

- slvher 2018-10-01 01:36:12
对文中示例的无限循环有疑问的同学，建议去查函数调用的栈桢结构细节（操作系统或计算机体系结构的教材应该会讲到）。

函数体内的局部变量存在栈上，且是连续压栈。在Linux进程的内存布局中，栈区在高地址空间，从高向低增长。变量i和arr在相邻地址，且i比arr的地址大，所以arr越界正好访问到i。当然，前提是i和arr元素同类型，否则那段代码仍是未决行为。 [588赞]

作者回复2018-10-01 15:24:56
高手！

- 不诉离殇 2018-10-01 01:35:09
例子中死循环的问题跟编译器分配内存和字节对齐有关 数组3个元素 加上一个变量a 。4个整数刚好能满足8字节对齐 所以i的地址恰好跟着a2后面 导致死循环。。如果数组本身有4个元素 则这里不会出现死循环。。因为编译器64位操作系统下 默认会进行8字节对齐 变量i的地址就不紧跟着数组后面了。 [196赞]

作者回复2018-10-01 15:25:33
高手！

- Rain 2018-09-30 22:20:56

  根据我们前面讲的数组寻址公式，a[3] 也会被定位到某块不属于数组的内存地址上，而这个地址正好是存储变量 i 的内存地址，那么 a[3]=0 就相当于 i=0，所以就会导致代码无限循环。

  *而这个地址正好是存储变量 i 的内存地址*这个地方没看太懂，为什么正好就是i的内存地址呢？

  谢谢老师。 [146赞]

- Nirvanaliu 2018-10-01 00:05:18

  文章结构：

  数组看起来简单基础，但是很多人没有理解这个数据结构的精髓。带着为什么数组要从0开始编号，而不是从1开始的问题，进入主题。

  1. 数组如何实现随机访问

  1） 数组是一种线性数据结构，用连续的存储空间存储相同类型数据

  I） 线性表：数组、链表、队列、栈 非线性表：树 图

  II） 连续的内存空间、相同的数据，所以数组可以随机访问，但对数组进行删除插入，为了保证数组的连续性，就要做大量的数据搬移工作

  a) 数组如何实现下标随机访问。

  引入数组再内存种的分配图，得出寻址公式

  b) 纠正数组和链表的错误认识。数组的查找操作时间复杂度并不是O(1)。即便是排好的数组，用二分查找，时间复杂度也是O（logn）。

  正确表述：数组支持随机访问，根据下标随机访问的时间复杂度为O（1）

  2. 低效的插入和删除

  1） 插入：从最好O(1) 最坏O(n) 平均O(n)

  2） 插入：数组若无序，插入新的元素时，可以将第K个位置元素移动到数组末尾，把心的元素，插入到第k个位置，此处复杂度为O(1)。作者举例说明

  3） 删除：从最好O(1) 最坏O(n) 平均O(n)

  4） 多次删除集中在一起，提高删除效率

  记录下已经被删除的数据，每次的删除操作并不是搬移数据，只是记录数据已经被删除，当数组没有更多的存储空间时，再触发一次真正的删除操作。即JVM标记清除垃圾回收算法。

  3. 警惕数组的访问越界问题

  用C语言循环越界访问的例子说明访问越界的bug。此例在《C陷阱与缺陷》出现过，很惭愧，看过但是现在也只有一丢丢印象。翻了下书，替作者加上一句话：如果用来编译这段程序的编译器按照内存地址递减的方式给变量分配内存，那么内存中的i将会被置为0，则为死循环永远出不去。

  4. 容器能否完全替代数组

  相比于数字，java中的ArrayList封装了数组的很多操作，并支持动态扩容。一旦超过村塾容量，扩容时比较耗内存，因为涉及到内存申请和数据搬移。

  数组适合的场景：

  1） Java ArrayList 的使用涉及装箱拆箱，有一定的性能损耗，如果特别管柱性能，可以考虑数组

2） 若数据大小事先已知，并且涉及的数据操作非常简单，可以使用数组

3） 表示多维数组时，数组往往更加直观。

4） 业务开发容器即可，底层开发，如网络框架，性能优化。选择数组。

5. 解答开篇问题

1） 从偏移角度理解a[0] 0为偏移量，如果从1计数，会多出K-1。增加cpu负担。为什么循环要写成for(int i = 0;i<3;i++) 而不是for(int i = 0 ;i<=2;i++)。第一个直接就可以算出3-0 = 3 有三个数据，而后者 2-0+1个数据，多出1个加法运算，很恼火。

2） 也有一定的历史原因

[88赞]

- zyzheng 2018-10-03 10:35:22

  关于数组越界访问导致死循环的问题，我也动手实践了一下，发现结果和编译器的实现有关，gcc有一个编译选项（-fno-stack-protector）用于关闭堆栈保护功能。默认情况下启动了堆栈保护，不管i声明在前还是在后，i都会在数组之后压栈，只会循环4次；如果关闭堆栈保护功能，则会出现死循环。请参考：https://www.ibm.com/developerworks/cn/linux/l-cn-gccstack/index.html [79赞]

  作者回复2018-10-03 15:40:23

  就喜欢你这种自己动手研究的同学

- 夜下凝月 2018-10-07 09:43:04

  突然想到了垃圾桶。
  生活中，我们扔进屋里垃圾桶的垃圾，
  并没有消失，只是被 "标记" 成了垃圾，
  只有垃圾桶塞满时，才会清理垃圾桶。
  再次存放垃圾 [77赞]

- Zzzzz 2018-10-01 10:44:55

  对于死循环那个问题，要了解栈这个东西。栈是向下增长的，首先压栈的i，a[2]，a[1]，a[0]，这是我在我vc上调试查看汇编的时候看到的压栈顺序。相当于访问a[3]的时候，是在访问i变量，而此时i变量的地址是数组当前进程的，所以进行修改的时候，操作系统并不会终止进程。 [54赞]

  作者回复2018-10-01 15:19:06

- 何江 2018-10-02 01:07:55

  有个小问题，我觉得 随机访问Ramdom Acess 更应该翻译成 任意访问，更能表达数组的特性。不过国内书籍都是翻译成随机。新手朋友刚看到时会有一些理解问题，如数组怎么会是随机访问的呢(当初我就是这么想的) [38赞]

- 李朋远 2018-10-03 12:35:56

老师，您好，个人觉得您举例的内存越界的循环应该限制在　架构的小端模式，在别的架构平台上的大端模式应该不是这样的！　赞

作者回复2018-10-03 15:35:01

说的没错