数据结构

第 3 周 – 数组：一维和多维数组、存储方式、动态数组

Koczka Ferenc 博士 · 埃斯特哈齐·卡罗伊大学 · 2026

复习

- **大O记号：**步数的阶 – O(1)、O(log n)、O(n)、$O(n^2)$
- **顺序查找：**O(n)，适用于任何数组
- **二分查找：**O(log n)，适用于有序数组，需要**直接访问**

今天的问题

- 机器如何存储数组？为什么访问是 O(1) 的？
- 插入或删除一个元素的代价是多少？
- 静态数组怎样“扩展”？

数组的概念

数组是由相同类型的元素组成的有限序列，通过**下标**访问元素。

性质	数组
元素类型	**同构**
大小	**静态** – 创建时确定
存储	**顺序** – 元素紧挨着依次存放
访问	**直接** – 任一元素 O(1) 时间
关系	**线性**

又称：**向量**（一维）、**矩阵**（二维）

Java 和 Go 中的数组

Java

```java
int[] a = new int[5];          // 全部为 0
int[] p = {2, 3, 5, 7, 11};
p[0] = 1;
int n = p.length;              // 5

for (int x : p)
    System.out.print(x + " ");

int[] b = a;                   // 同一个数组
int[] c = Arrays.copyOf(p, p.length);
```

- 数组是**对象**，变量是引用
- 默认值：`0`、`0.0`、`false`、`null`

Go

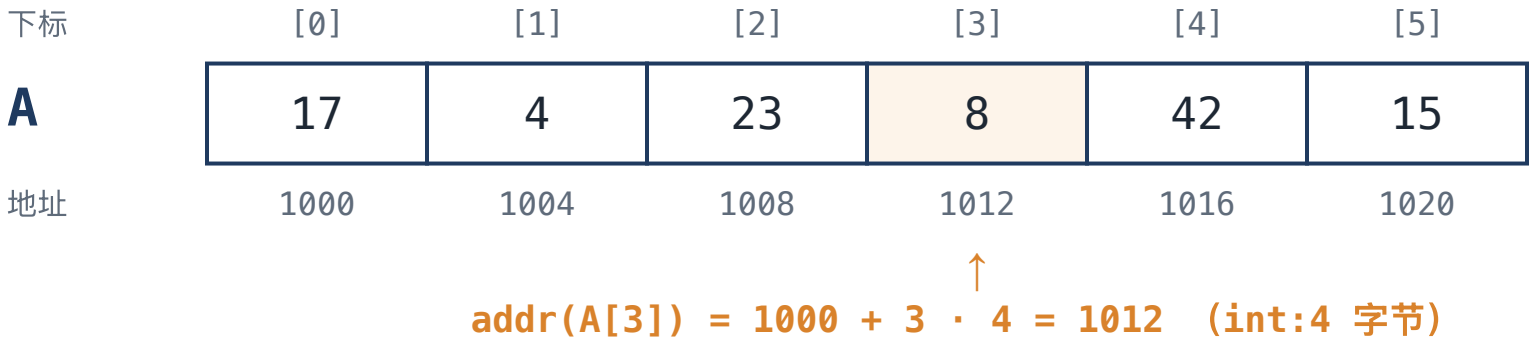
```go
var a [5]int                   // 全部为 0
p := [5]int{2, 3, 5, 7, 11}
p[0] = 1
n := len(p)                    // 5

for _, x := range p {
    fmt.Print(x, " ")
}

b := a                         // 一份副本！
s := []int{2, 3, 5}            // 切片
```

- 大小是**类型的一部分**：`[5]int` ≠ `[6]int`
- 数组是**值** – 实际中我们使用切片（`[]int`）

在内存中的存储



第 **i** 个元素的地址通过**一次计算**得到：

> addr(A[i]) = 基地址 + i · 元素大小　　（下标从 0 开始）
> addr(A[i]) = 基地址 + (i – 1) · 元素大小　　（下标从 1 开始）

所以访问是 **O(1)** 的 – 不需要经过前面的元素。

为什么下标从 0 开始?

- 下标从 0 开始时，下标就是元素相对数组起点的**偏移量**
- `A[0]` 位于基地址，`A[3]` 在它后面三个元素处
- C、Java、Go、Python：从 0 开始 · Pascal、Fortran、Lua、数学：从 1 开始

伪代码中下标**从 1 开始**，程序中**从 0 开始**。翻译时：
`For i ← 1 to n` → `for (int i = 0; i < n; i++)`

边界检查

- Java: `ArrayIndexOutOfBoundsException` · Go: `panic: index out of range`
- C 不做检查：程序会悄悄**改写**相邻的内存 – 这是严重安全漏洞的根源之一

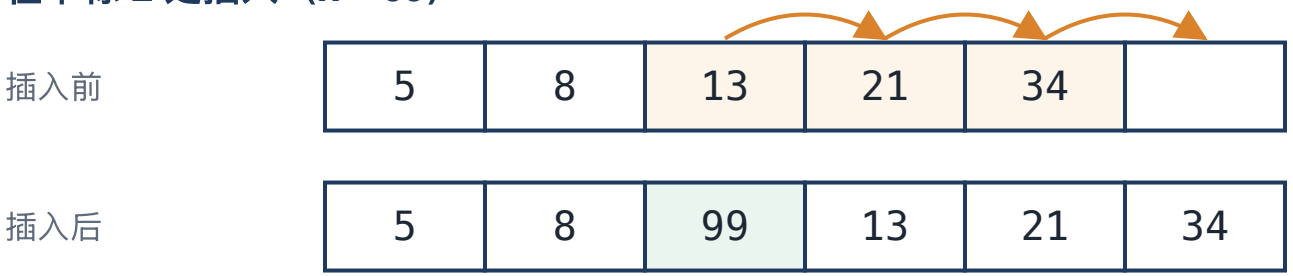
操作的代价

操作	步数	原因
读/写第 i 个元素	**O(1)**	地址计算
查找（无序）	O(n)	顺序查找
查找（有序）	O(log n)	二分查找
在末尾插入（有空位时）	O(1)	
在开头/中间插入	**O(n)**	需要移动元素
在开头/中间删除	**O(n)**	需要移动元素

> 数组**读取快**、**修改慢** – 如果需要在中间频繁地插入和删除，**链表**（第 9 周）可能是更好的选择。

插入与删除

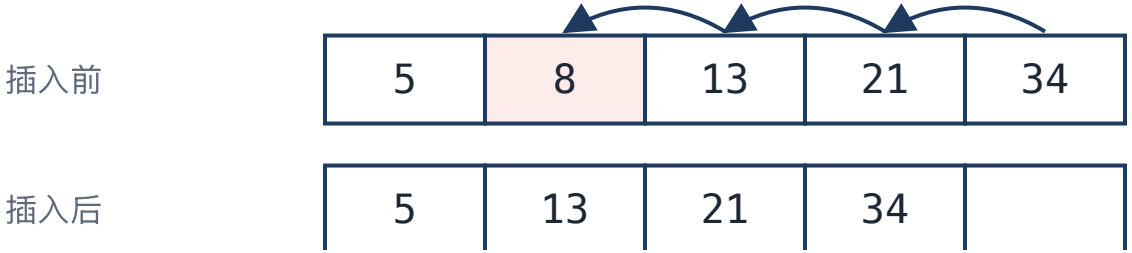
在下标 2 处插入（x = 99）



3 个元素右移

（从后往前！）

删除下标 1 处的元素



3 个元素左移

（从前往后）

插入与删除 – 算法

在位置 k 插入

```
Procedure Insert(A, n, k, x)
  // 前提:n < 容量
  For i ← n down to k
    A[i + 1] ← A[i]
  End for
  A[k] ← x
  n ← n + 1
End procedure
```

删除位置 k 的元素

```
Procedure Delete(A, n, k)
  For i ← k to n − 1
    A[i] ← A[i + 1]
  End for
  n ← n − 1
End procedure
```

Java

```java
static int insert(int[] a, int n,
                  int pos, int x) {
    for (int i = n - 1; i >= pos; i--)
        a[i + 1] = a[i];
    a[pos] = x;
    return n + 1;
}
```

插入时必须**从后往前**移动！如果从前往后移，每个元素都会被第一个移动的值覆盖。

原地逆置与循环移位

用两个下标**逆置** – O(n),O(1) 空间

```
Procedure Reverse(A, n)
  i ← 1 ; j ← n
  While i < j
    swap(A[i], A[j])
    i ← i + 1 ; j ← j − 1
  End while
End procedure
```

`[1 2 3 4 5]` → `[5 4 3 2 1]`

循环左移 k 位 – 三次逆置

`[1 2 | 3 4 5]`,k = 2

1. 逆置前 k 个元素：`[2 1 | 3 4 5]`
2. 逆置其余部分：`[2 1 | 5 4 3]`
3. 逆置整个数组：`[3 4 5 1 2]` ✓

不需要辅助数组，O(n) 步。

多维数组

二维数组（矩阵）

3 × 4 矩阵

	0	1	2	3
0	a	b	c	d
1	e	f	g	h
2	i	j	k	l

`A[1][2] = g`

存储

内存中按行优先存储

a	b	c	d	e	f	g	h	i	j	k	l
0	1	2	3	4	5	6	7	8	9	10	11

`addr(A[i][j]) = 基地址 + (i · 列数 + j) · 元素大小`

`A[1][2]:1 · 4 + 2 = 第 6 个元素`

- 内存是**一维**的 – 矩阵必须“拉平”成一个序列
- **行优先**存储（C、Java、Go）：先存第一行，再存第二行 …
- **列优先**存储（Fortran、MATLAB）：先存第一列 …

地址计算：一般公式

N × **M** 矩阵（N 行、M 列），元素大小 **s**，基地址 **k**：

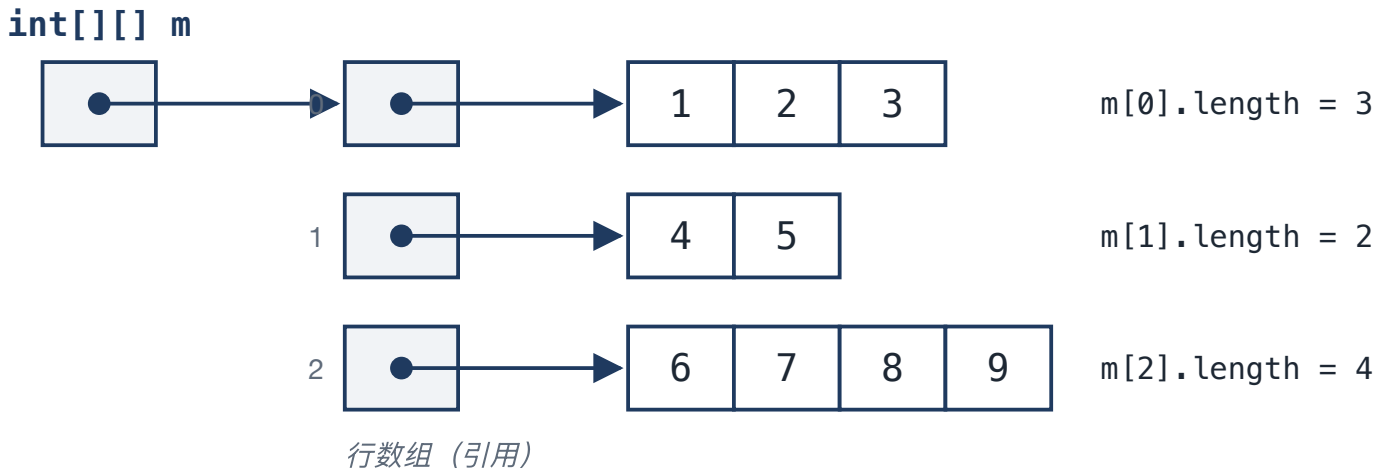
	下标从 0 开始	下标从 1 开始
行优先	$k + (i \cdot M + j) \cdot s$	$k + ((i - 1) \cdot M + (j - 1)) \cdot s$
列优先	$k + (j \cdot N + i) \cdot s$	$k + ((j - 1) \cdot N + (i - 1)) \cdot s$

例： `int A[10][20]`（元素 4 字节），基地址 2000，行优先，下标从 0 开始。

$\text{addr}(A[3][5]) = 2000 + (3 \cdot 20 + 5) \cdot 4 = 2000 + 260 = \mathbf{2260}$

三维数组 `A[i][j][l]`，维数 N × M × L：$k + ((i \cdot M + j) \cdot L + l) \cdot s$

Java 中的二维数组：数组的数组




```
int[][] m = new int[3][4];           // 3 行,每行长度 4
int[][] jagged = { {1, 2, 3}, {4, 5}, {6, 7, 8, 9} };
```

- 每一行都是**单独的对象** → 各行的长度可以不同（**锯齿数组**）
- `m.length` 是行数，`m[i].length` 是第 `i` 行的长度

Go 中的二维数组

真正的二维数组 – 一整块连续内存

```
var grid [3][4]int
grid[1][2] = 7
```

- 大小在编译时已知
- 严格按行优先存储

切片的切片 – 与 Java 类似

```
m := make([][]int, rows)
for i := range m {
    m[i] = make([]int, cols)
}
```

- 大小可以在运行时给出
- 各行的长度可以不同

遍历矩阵时，**按行**的顺序（外层循环遍历行）更快，因为它能更好地利用处理器缓存。

矩阵运算

主对角线和副对角线 – O(n)

```
For i ← 1 to n
  main ← main + A[i, i]
  anti ← anti + A[i, n − i + 1]
End for
```

转置 – O(n · m)

```
For i ← 1 to n
  For j ← 1 to m
    T[j, i] ← A[i, j]
  End for
End for
```

矩阵乘法 – (n×k) · (k×m),**O(n · k · m)**

```
For i ← 1 to n
  For j ← 1 to m
    C[i, j] ← 0
    For x ← 1 to k
      C[i, j] ← C[i, j] + A[i, x] · B[x, j]
    End for
  End for
End for
```

方阵为 **$O(n^3)$** – 三重嵌套循环。

动态数组

问题：数组的大小是固定的

- 我们常常**事先不知道**需要存放多少个元素（例如从输入读取的数据）
- 数组太小 → 放不下 · 数组太大 → 浪费

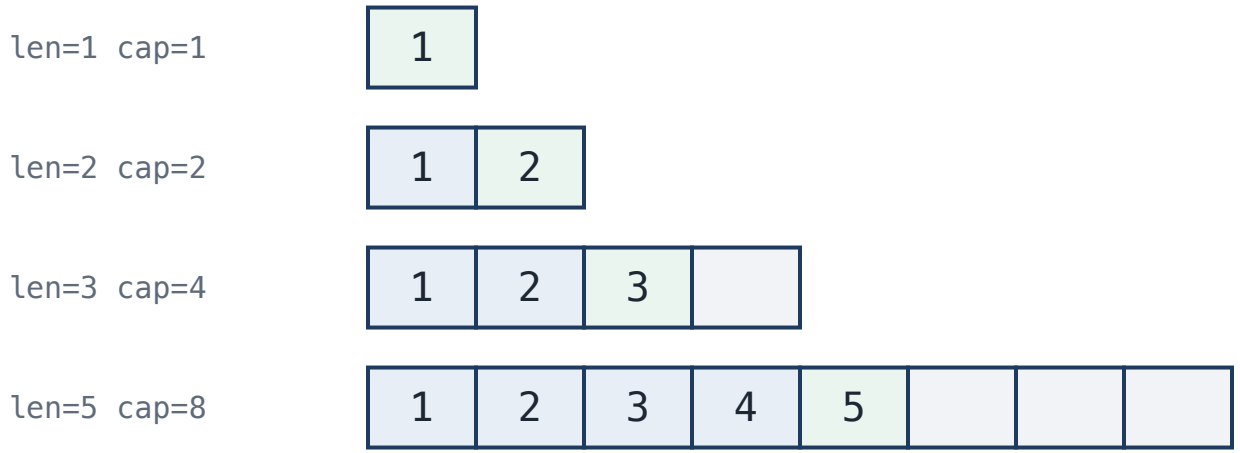
解决办法：动态数组 – 必要时**换成一个更大的数组**

- `size`（长度）：实际包含多少个元素
- `capacity`（容量）：底层数组的大小

Java: `ArrayList` · Go：切片 + `append` · C++: `vector` · Python: `list`

动态数组的工作原理

append：满时分配两倍大小的数组并复制元素



复制次数：1 + 2 + 4 + … < 2n

→ **每次 append 平均 O(1)**

（均摊代价）

- 每次增加一个：1 + 2 + 3 + … + n = 总共 **O(n²)** 次复制
- **每次加倍：**1 + 2 + 4 + … + n < 2n → 总共 O(n)，每个元素 **O(1)**

自己实现动态数组

Java

```java
class DynamicArray {
    private int[] data = new int[1];
    private int size = 0;

    void add(int x) {
        if (size == data.length) {
            int[] bigger =
                new int[2 * data.length];
            for (int i = 0; i < size; i++)
                bigger[i] = data[i];
            data = bigger;
        }
        data[size++] = x;
    }
    int get(int i) { return data[i]; }
}
```

Go

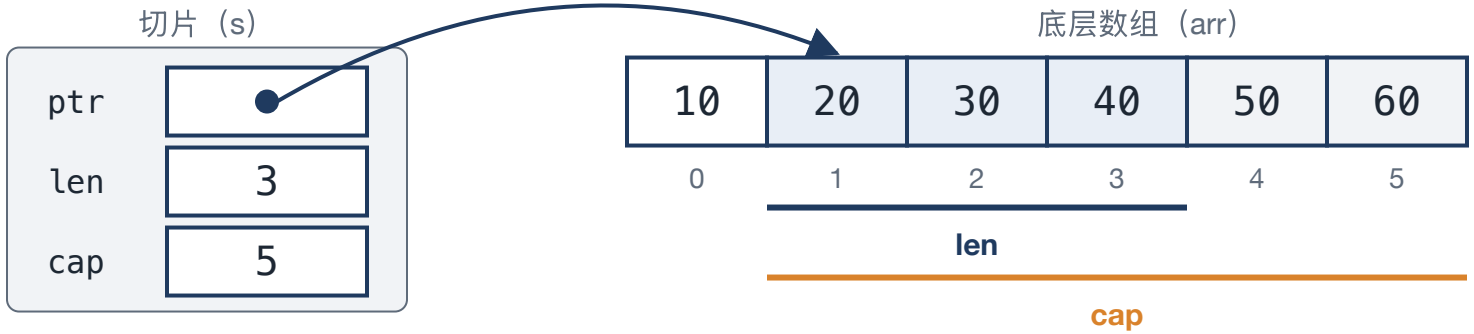
```go
type DynamicArray struct {
    data []int
    size int
}

func (d *DynamicArray) Add(x int) {
    if d.size == len(d.data) {
        bigger := make([]int, 2*len(d.data))
        copy(bigger, d.data[:d.size])
        d.data = bigger
    }
    d.data[d.size] = x
    d.size++
}
```

实测：1 000 000 次 `add` → **1 048 575** 次复制，每个元素 **1.05** 次

Go 切片：长度、容量、底层数组

```
s := arr[1:4] // len 3, cap 5
```



`s[0] = 99 → arr[1] 也变成 99！（共享底层数组）`

- 切片由三个值组成：指向底层数组的**指针**、**长度**（`len`）和**容量**（`cap`）
- 只有容量用完时，`append` 才会分配新的数组
- 因此总是写成：`s = append(s, x)` — 结果必须赋值回去！

Java 的 `ArrayList`

```java
ArrayList<Integer> list = new ArrayList<>();
list.add(5);          // 在末尾 — 均摊 O(1)
list.add(0, 3);       // 在开头插入 — O(n)
list.get(1);          // 读 — O(1)
list.set(1, 9);       // 写 — O(1)
list.remove(0);       // 按下标删除 — O(n)
list.size();          // 元素个数
```

- 只能存放**对象**：用 `Integer` 代替 `int`（自动装箱和拆箱）
- 内部是一个 `Object[]` 数组，满时扩大为 1.5 倍

小结

- 数组是**同构、静态、顺序存储、直接访问**的数据结构
- 地址计算：`基地址 + 下标 · 元素大小` → 访问 **O(1)**
- 在中间插入和删除是 **O(n)** – 需要移动元素
- 矩阵：**行优先**存储，地址 = k + (i · M + j) · s
- Java：二维数组是数组的数组 · Go: `[N][M]T` 或 `[][]T`
- **动态数组：**容量加倍 → 追加操作均摊 **O(1)**

下周：字符串 – 字符序列、编码、操作、模式匹配。

复习题

1. 判断对错：数组的元素在内存中是分散存放的。
2. 删除 **n** 个元素的数组的第一个元素需要多少步？
3. 矩阵 `int A[5][8]` 从地址 100 开始按行优先存储，元素 4 字节，下标从 0 开始。`A[2][3]` 的地址是多少？
4. 为什么插入时必须从后往前移动？
5. 为什么动态数组的容量要加倍，而不是每次加一？
6. 输出什么？ `s := []int{1,2,3}; t := s; t[0] = 9; fmt.Println(s[0])`

实践课

✎ 示例程序

程序	Java	Go	演示内容
基础	`ArrayBasics.java`	`w03/arraybasics`	创建、默认值、越界、复制
操作	`ArrayOps.java`	`w03/arrayops`	插入、删除、逆置、循环移位
矩阵	`Matrix.java`	`w03/matrix`	对角线、转置、乘法、锯齿数组
动态数组	`DynamicArray.java`	`w03/dynarray`	容量加倍、复制次数、`append`

在 Go 程序 `dynarray` 中观察内置 `append` 的容量是怎样增长的，以及共享底层数组时会发生什么！

✎ 练习

1. **平均值**：计算整数数组元素的平均值！
2. **包含**：判断数组中是否包含给定的值！
3. **第二大**：**只遍历一次**，找出第二大的（不同的）元素！
4. **去重**：在**有序**数组中原地删除重复元素，并返回新长度！（`[1,1,2,3,3,3,5]` → `[1,2,3,5]`）
5. **合并**：用 O(n + m) 步把两个有序数组合并成一个有序数组！
6. **行和与列和**：计算矩阵每一行和每一列的和！
7. **螺旋**：按螺旋顺序输出矩阵的元素！（`1 2 3 6 9 8 7 4 5`）

参考答案：`Exercises03.java`、`w03/exercises`

✎ 课后作业

1. 为我们自己的 `DynamicArray` 类增加 `insert(i, x)` 和 `remove(i)` 操作！它们需要多少步？
2. 编写 `remove`，使数组只剩**四分之一**满时把容量减半！为什么不在半满时就减半？
3. 编写程序，把 **n** × **n** 矩阵**原地**顺时针旋转 90 度！（提示：转置 + 逆置每一行。）
4. 计算：`double T[50][40]`（元素 8 字节），列优先，下标从 1 开始，基地址 5000。`T[12][7]` 的地址是多少？