

数据结构层面,大家需要掌握以下几种:

- 数组
- 栈
- 队列
- 链表
- 树(这里我们着重讲二叉树)

对于这些数据结构,各位如果没有大量的可支配时间可以投入,那么其实不建议找厚厚的大学教材来刷。此时此刻,时间为王,我们追求的是效率的最大化。

不同的数据结构教材,对数据结构有着不同的划分、不同的解读、不同的编码实现。在这里,我们面向 JavaScript,面向前端面试,只针对大家后续做题、答题时会用到的最贴合实战的数据结构特性&编码技能作讲解。

保姆式教学の温情提示:

这两节我们所提及的基础知识细节,很可能会成为你后面写代码的关键线索。

不要因为乍一看觉得简单,就急着跳读急着做题。

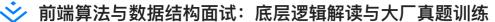
不然你很可能做题做到一半,会不知道自己到底为什么就卡了壳。 到时候万一又因为懒得回头看,而原地卡死,那就更做不下去了orz。

注:由于 JavaScript 中字符串和数组关联紧密,关键知识点重复度较高,故我们在数据结构部分,不再单独为字符串保留篇幅。字符串相关的知识点,我们直接带到后续的解题技巧归纳专题里去看。

数组

数组是各位要认识的第一个数据结构。

作为最简单、最基础的数据结构,大多数的语言都天然地对数组有着原生的表达, JavaScript 亦然。这意味着我们可以对数组做到"开箱即用",而不必自行模拟实现, 非常方便。







需要它帮忙的地方会非常多。

数组的创建

大家平时用的最多的创建方式想必就是直接方括号+元素内容这种形式:

不过在算法题中,很多时候我们初始化一个数组时,并不知道它内部元素的情况。这 种场景下, 要给大家推荐的是构造函数创建数组的方法:

```
js
const arr = new Array()
```

当我们以构造函数的形式创建数组时,若我们像楼上这样,不传任何参数,得到的就 会是一个空数组。等价于:

```
js
const arr = []
```

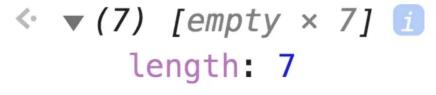
不过咱们使用构造函数,可不是为了创建空数组这么无聊。

我们需要它的时候,往往是因为我们有"创造指定长度的空数组"这样的需求。需要多 长的数组,就给它传多大的参数:

```
js
const arr = new Array(7)
```

这样的写法就可以得到一个长度为7的数组:

> arr







一个1. 只需给它 fill 一个1:

```
js
const arr = (new Array(7)).fill(1)
```

如此便可以得到一个长度为7,且每个元素都初始化为1的数组:

> arr

数组的访问和遍历

访问数组中的元素, 我们直接在中括号中指定其索引即可:

```
js
arr[0] // 访问索引下标为0的元素
```

而遍历数组,这个方法就多了,不过目的往往都是一致的——访问到数组中的每个元 素,并且知道当前元素的索引。这里我们讲三个方法:

1. for 循环

这个是最最基础的操作。我们可以通过循环数组的下标,来依次访问每个值:

```
js
// 获取数组的长度
const len = arr.length
for(let i=0;i<len;i++) {</pre>
   // 输出数组的元素值,输出当前索引
   console.log(arr[i], i)
}
```

2. forEach 方法

通过取 forEach 方法中传入函数的第一个入参和第二个入参,我们也可以取到数组每 个元素的值及其对应索引:





```
console.log(item, index)
})
```

3. map 方法

map 方法在调用形式上与 forEach 无异,区别在于 map 方法会根据你传入的函数逻 辑对数组中每个元素进行处理、进而返回一个全新的数组。

所以其实 map 做的事情不仅仅是遍历,而是在遍历的基础上"再加工"。当我们需要 对数组内容做批量修改、同时修改的逻辑又高度一致时,就可以调用 map 来达到我 们的目的:

```
js
const newArr = arr.map((item, index)=> {
   // 输出数组的元素值,输出当前索引
   console.log(item, index)
   // 在当前元素值的基础上加1
   return item+1
})
```

这段代码就通过 map 来返回了一个全新的数组,数组中每个元素的值都是在其现有 元素值的基础上+1后的结果。

这里给个小建议:个人推荐如果没有特殊的需要,那么统一使用 for 循环来实现遍 历。因为**从性能上看,for 循环遍历起来是最快的**。

二维数组

初学编程的同学基础如果比较薄弱,会对二维数组完全没有概念。这里咱们先简单介 绍下:二维数组其实就是数组套数组,也就是每个元素都是数组的数组。 说起来有点绕口,咱们直接上图来看:

```
js
const arr = [1,2,3,4,5]
```

这个数组在逻辑上的分布就是这样式儿的:







@稀土掘金技术社区

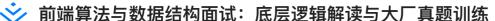
js

像图上这样,数组的元素是数字而非数组。整个数组的结构看上去宛如一条"线",这 就是一维数组。

而"每个元素都是数组的数组",代码里看是这样:

```
const arr = [
  [1,2,3,4,5],
  [1,2,3,4,5],
  [1,2,3,4,5],
  [1,2,3,4,5],
  [1,2,3,4,5]
]
```

直接把它的逻辑结构画出来看,是这样:







0	1	2	3	4	5
1	1	2	3	4	5
2	1	2	3	4	5
3	1	2	3	4	5
4	1	2	3	4	5

@稀土掘金技术社区

图中的每一行,就代表着一个数组元素。比如第 0 行,就代表着数组中 arr[0] 这个 数组元素, 其内容是 [1,2,3,4,5]。

每一行中的每一列,则代表一个确切的"坑"。比如第0行第1列,就代表着数组中 arr[0][1] 这个元素,其值为2,是一个确切的 number。

明白了二维数组的索引规律,现在我们来看一下二维数组的特点:从形状上看,相对 于一维数组一条"线"一般的布局,二维数组更像是一个"面"。拿咱们这个例子来说, 这里的二维数组逻辑分布图就宛如一个正方形。当然啦,如果我们稍微延长一下其中 的一边,它也可以是一个矩形。

在数学中,形如这样**长方阵列排列的复数或实数集合**,被称为"矩阵"。因此二维数组 的别名就叫"矩阵"。

讲到这里,如果有对"矩阵"的定义一脸懵逼的同学,也不用怕——不知道"矩阵"是 啥,一点不打紧(所以快停下你复制粘贴到 Google 的手哈哈),但你必须要记住 **"矩阵"和"二维数组"之间的等价关系**。在算法题目中,见到"矩阵"时,能够立刻反 射出它说的是二维数组,不被别名整懵逼,这就够了。



fill 的局限性

有同学用 fill 方法用顺了手,就本能地想用 fill 解决所有的问题,比如初始化一个二维数组:

```
const arr =(new Array(7)).fill([])
```

js

乍一看没啥毛病,7个坑都被乖乖地填上了数组元素:

```
> arr
```

(7) [Array(0), Array(0), Array(0), Array(0), Array(0), Array(0)]

▶ 0: []

▶ 1: []

▶2: []

▶3: []

▶4: []

▶5: []

6: []

length: 7

▶ __proto__: Array(0)

@掘金技术社区

但是当你想修改某一个坑里的数组的值的时候:

```
arr[0][0] = 1
```

js

你会发现一整列的元素都被设为了 1:







```
* (1), Array(1), Array(1), Array(1)]
▶ 0: [1]
▶ 1: [1]
▶ 2: [1]
▶ 3: [1]
▶ 4: [1]
▶ 5: [1]
▶ 6: [1]
▶ 6: [1]
▶ proto : Array(0)
```

这是什么骚操作???

这就要从 fill 的工作机制讲起了。各位要清楚,当你给 fill 传递一个入参时,**如果这个入参的类型是引用类型,那么 fill 在填充坑位时填充的其实就是入参的引用**。也就是说下图中虽然看似我们给7个坑位各初始化了一个数组:

```
> arr
```

```
    (7) [Array(0), Array(0), Array(0), Array(0), Array(0), Array(0)] 
    ▶ 0: []
    ▶ 1: []
    ▶ 2: []
    ▶ 3: []
    ▶ 4: []
    ▶ 5: []
    ▶ 6: []
    ▶ 6: []
    ▶ 6: []
```

其实这7个数组对应了同一个引用、指向的是同一块内存空间,**它们本质上是同一个数组**。因此当你修改第0行第0个元素的值时,第1-6行的第0个元素的值也都会跟着发生改变。







js

js

```
const len = arr.length
for(let i=0;i<len;i++) {
    // 将数组的每一个坑位初始化为数组
    arr[i] = []
}</pre>
```

for 循环中,每一次迭代我们都通过"[]"来创建一个新的数组,这样便不会有引用指向问题带来的尴尬。

二维数组的访问

访问二维数组和访问一维数组差别不大,区别在于我们现在需要的是两层循环:

```
// 缓存外部数组的长度
const outerLen = arr.length
for(let i=0;i<outerLen;i++) {
    // 缓存内部数组的长度
    const innerLen = arr[i].length
    for(let j=0;j<innerLen;j++) {
        // 输出数组的值,输出数组的索引
        console.log(arr[i][j],i,j)
    }
}</pre>
```

一维数组用 for 循环遍历只需一层循环,二维数组是两层,三维数组就是三层。依次 类推,**N 维数组需要 N 层循环来完成遍历**。

数组小结

关于数组的基础知识,咱们整整用掉了一节的篇幅来介绍,可见其重要性。

在本节,我们仅仅围绕数组最基本的操作进行介绍,这远不是数组的全部。关于数组,还有太多太多的故事要讲——实际上,单就其重要的方法的使用:如concat、some、slice、join、sort、pop、push 等等这些,就足以说上个把钟头。





关于数组的更多特性和技巧,会被打散到后续的章节中去。各位在真题解读的环节、 包括在其它数据结构的讲解中,都会不可避免地再见到数组的身影。彼时数组的每一 个方法都会和它对应的应用场景一起出现,相信你会有更深刻的记忆。

事实上,在 JavaScript 数据结构中,数组几乎是"基石"一般的存在。这一点,大家 在下一节就会有所感触。

(阅读过程中有任何想法或疑问,或者单纯希望和笔者交个朋友啥的,欢迎大家添加 我的微信xyalinode与我交流哈~)

留言

输入评论(Enter换行、 器 + Enter发送)

发表评论

全部评论(91)



喂喂喂321 9分钟前

99

△ 点赞 🖵 回复



喂喂喂321 11分钟前

99.6

△ 点赞 □ 回复



剁椒鱼头666 № 前端开发 @ 铼 1月前

打卡

△ 点赞 🖵 回复



yanwuhc 前端白菜工程师 1月前

打卡







mark

△ 点赞 🖵 回复



笑笑X ■ 3月前

数组的创建、遍历

△ 点赞 🖃 回复



暖阳之时 前端 @ 联通支付 3月前

看到二维数组想到了之前的一个面试题: 给定一个m*n的二维数组,要求斜向输入每一个 元素

△ 点赞 □ 3



Rita命 2月前

斜向输入每一个元素是什么意思啊

△ 点赞 🖵 回复



咖妃 回复 Rita命 1月前

可能是(1,1),(2,2),(3,3)这种 🙀

"斜向输入每一个元素是什么意思啊"

△ 点赞 🖵 回复

查看更多回复 ~



敲代码的小提琴手 **□** 前端开发练习生 @ JDT 4月前

分享一个创建 m行n列 所有元素为6 的二维数组 的方法 const arr = new Array(m).fill(0).map(() => new Array(n).fill(6))

△1 □回复

bensonLI 5月前

打卡学习

△ 点赞 🖵 回复



漫无目的的悲观主... ☑ 前端开发 6月前



△ 点赞 □ 回复

遇之则安伴之则暖 6月前





muyiyr 🛂 前端 6月前

打卡

△ 点赞 🖵 回复



是阿正啊 □ 前端 @ 梦想 7月前

终于知道为啥fill二维数组再赋值的错误原因了

1分2 三2

漫无目的的...

因为 你赋值了一个指针

16 2 回复

帅得乱七八糟 🚾 4月前

因为你在一维层面所有元素赋值的是同一个数组的引用

△ 点赞 🖵 回复

Jason_cc 前端工程师 7月前

打卡

△ 点赞 🖵 回复

月圆之夜 7月前

冲啊~

△ 点赞 □ 回复



空白处 、 前端 7月前

打卡 🎁

△ 点赞 □ 回复

Mr.HoTwo 7月前

冲!!!

△ 点赞 🖵 回复



奔小猪 □ 前端工程师 8月前

打卡

△ 点赞 □ 1



海染蓝了天 8月前







锐来锐好 前端 @ 群接龙 8月前

let a = Array.from(new Array(5), () => new Array(5).fill(1)) 这样来创建好不好呢? <u>1</u>2 □ 1



敲代码的小... □ 4月前

之前刷题遇到的题解用的map方法 感觉那种要权威并普适一些吧!

△ 点赞 □ 回复



冰淇淋先生 前端开发 8月前

打卡

△ 点赞 □ 回复

查看全部 91 条回复 ~



