一面 4：从容应对算法题目

由冯·诺依曼机组成我们知道：数据存储和运算是计算机工作的主要内容。`程序=数据结构+算法`，所以计算机类工程师必须掌握一定的数据结构和算法知识。

知识点梳理

- 常见的数据结构
  - 栈、队列、链表
  - 集合、字典、散列集
- 常见算法
  - 递归
  - 排序
  - 枚举
- 算法复杂度分析
- 算法思维
  - 分治
  - 贪心
  - 动态规划
- 高级数据结构
  - 树、图
  - 深度优先和广度优先搜索

本小节会带领大家快速过一遍数据结构和算法，重点讲解一些常考、前端会用到的算法和数据结构。

数据结构

数据结构决定了数据存储的空间和时间效率问题，数据的写入和提取速度要求也决定了应该选择怎样的数据结构。

根据对场景需求的不同，我们设计不同的数据结构，比如：

- 读得多的数据结构，应该想办法提高数据的读取效率，比如 IP 数据库，只需要写一次，剩下的都是读取；
- 读写都多的数据结构，要兼顾两者的需求平衡，比如 LRU Cache 算法。

算法是数据加工处理的方式，一定的算法会提升数据的处理效率。比如有序数组的二分查找，要比普通的顺序查找快很多，尤其是在处理大量数据的时候。

数据结构和算法是程序开发的通用技能，所以在任何面试中都可能会遇见。随着近几年 AI、大数据、小游戏越来越火，Web 前端职位难免会跟数据结构和算法打交道，面试中也会出现越来越多的算法题目。学习数据结构和算法也能够帮助我们打开思路，突破技能瓶颈。

前端常遇见的数据结构问题

现在我来梳理下前端常遇见的数据结构：

- 简单数据结构（必须理解掌握）
  - 有序数据结构：栈、队列、链表，有序数据结构省空间（存储空间小）
  - 无序数据结构：集合、字典、散列表，无序数据结构省时间（读取时间快）
- 复杂数据结构
  - 树、堆
  - 图

对于简单数据结构，在 ES 中对应的是数组（`Array`）和对象（`Object`）。可以想一下，数组的存储是有序的，对象的存储是无序的，但是我要在对象中根据`key`找到一个值是立即返回的，数组则需要查找的过程。

这里我通过一个真实面试题目来说明介绍下数据结构设计。

> 题目：使用 ECMAScript（JS）代码实现一个事件类`Event`，包含下面功能：绑定事件、解绑事件和派发事件。

在稍微复杂点的页面中，比如组件化开发的页面，同一个页面由两三个人来开发，为了保证组件的独立性和降低组件间耦合度，我们往往使用「订阅发布模式」，即组件间通信使用事件监听和派发的方式，而不是直接相互调用组件方法，这就是题目要求写的`Event`类。

这个题目的核心是一个事件类型对应回调函数的数据设计。为了实现绑定事件，我们需要一个`_cache`对象来记录绑定了哪些事件。而事件发生的时候，我们需要从`_cache`中读取出来事件回调，依次执行它们。一般页面中事件派发（读）要比事件绑定（写）多。所以我们设计的数据结构应该尽量地能够在事件发生时，更加快速地找到对应事件的回调函数们，然后执行。

经过这样一番考虑，我简单写了下代码实现：

``` class Event { constructor() { // 存储事件的数据结构 // 为了查找迅速，使用了对象（字典） this.*cache = {}; } // 绑定 on(type, callback) { // 为了按类查找方便和节省空间， // 将同一类型事件放到一个数组中 // 这里的数组是队列，遵循先进先出 // 即先绑定的事件先触发 let fns = (this.*cache[type] = this._cache[type] || []); if (fns.indexOf(callback) === -1) { fns.push(callback); } return this; } // 触发 trigger(type, data) { let fns = this.*cache[type]; if (Array.isArray(fns)) { fns.forEach((fn) => { fn(data); }); } return this; } // 解绑 off(type, callback) { let fns = this.*cache[type]; if (Array.isArray(fns)) { if (callback) { let index = fns.indexOf(callback); if (index !== -1) { fns.splice(index, 1); } } else { //全部清空 fns.length = 0; } } return this; } } // 测试用例 const event = new Event(); event.on('test', (a) => { console.log(a); }); event.trigger('test', 'hello world');

event.off('test'); event.trigger('test', 'hello world');

```

类似于树、堆、图这些高级数据结构，前端一般也不会考查太多，但是它们的查找方法却常考，后面介绍。高级数据应该平时多积累，好好理解，比如理解了堆是什么样的数据结构，在面试中遇见的「查找最大的 K 个数」这类算法问题，就会迎刃而解。

算法的效率是通过算法复杂度来衡量的

算法的好坏可以通过算法复杂度来衡量，算法复杂度包括时间复杂度和空间复杂度两个。时间复杂度由于好估算、好评估等特点，是面试中考查的重点。空间复杂度在面试中考查得不多。

常见的时间复杂度有：

- 常数阶 `O(1)`
- 对数阶 `O(logN)`
- 线性阶 `O(n)`
- 线性对数阶 `O(nlogN)`
- 平方阶 `O(n^2)`
- 立方阶 `O(n^3)`
- !k次方阶 `O(n^k)`
- 指数阶`O(2^n)`

随着问题规模 n 的不断增大，上述时间复杂度不断增大，算法的执行效率越低。

一般做算法复杂度分析的时候，遵循下面的技巧：

1. 看看有几重循环，一般来说一重就是`O(n)`，两重就是 `O(n^2)`，以此类推
2. 如果有二分，则为`O(logN)`
3. 保留最高项，去除常数项

> 题目：分析下面代码的算法复杂度（为了方便，我已经在注释中加了代码分析）

``` let i =0; // 语句执行一次 while (i < n) { // 语句执行 n 次 console.log(`Current i is ${i}`); //语句执行 n 次 i++; // 语句执行 n 次 }

```

根据注释可以得到，算法复杂度为`1 + n + n + n = 1 + 3n`，去除常数项，为`O(n)`。

``` let number = 1; // 语句执行一次 while (number < n) { // 语句执行 logN 次 number *= 2; // 语句执行 logN 次 }

```

上面代码`while`的跳出判断条件是`number<n`，而循环体内`number`增长速度是`(2^n)`，所以循环代码实际执行`logN`次，复杂度为：`1 + 2 * logN = O(logN)`

``` for (let i = 0; i < n; i++) {// 语句执行 n 次 for (let j = 0; j < n; j++) {// 语句执行 n^2 次 console.log('I am here!'); // 语句执行 n^2 次 } }
```

```

上面代码是两个`for`循环嵌套，很容易得出复杂度为：`O(n^2)`

# 人人都要掌握的基础算法

枚举和递归是最最简单的算法，也是复杂算法的基础，人人都应该掌握！枚举相对比较简单，我们重点说下递归。

递归由下面两部分组成：

1. 递归主体，就是要循环解决问题的代码
2. 递归的跳出条件，递归不能一直递归下去，需要完成一定条件后跳出

关于递归有个经典的面试题目是：

> 实现 JS 对象的深拷贝

**什么是深拷贝？**

「深拷贝」就是在拷贝数据的时候，将数据的所有**引用结构**都拷贝一份。简单的说就是，在内存中存在两个数据结构完全相同又相互独立的数据，将引用型类型进行复制，而不是只复制其引用关系。

分析下怎么做「深拷贝」：

1. 首先假设深拷贝这个方法已经完成，为 deepClone
2. 要拷贝一个数据，我们肯定要去遍历它的属性，如果这个对象的属性仍是对象，继续使用这个方法，如此往复

``` function deepClone(o1, o2) { for (let k in o2) { if (typeof o2[k] === 'object') { o1[k] = {}; deepClone(o1[k], o2[k]); } else { o1[k] = o2[k]; } } } // 测试用例 let obj = { a: 1, b: [1, 2, 3], c: {} }; let emptyObj = Object.create(null); deepClone(emptyObj, obj); console.log(emptyObj.a == obj.a); console.log(emptyObj.b == obj.b);

```

递归容易造成爆栈，尾部调用可以解决递归的这个问题，Chrome 的 V8 引擎做了尾部调用优化，我们在写代码的时候也要注意尾部调用写法。递归的爆栈问题可以通过将递归改写成枚举的方式来解决，就是通过`for`或者`while`来代替递归。

我们在使用递归的时候，要注意做优化，比如下面的题目。

> 题目：求斐波那契数列（兔子数列） 1,1,2,3,5,8,13,21,34,55,89...中的第 n 项

下面的代码中`count`记录递归的次数，我们看下两种差异性的代码中的`count`的值：

``` let count = 0; function fn(n) { let cache = {}; function _fn(n) { if (cache[n]) { return cache[n]; } count++; if (n == 1 || n == 2) { return 1; } let prev = _fn(n - 1); cache[n - 1] = prev; let next = _fn(n - 2); cache[n - 2] = next; return prev + next; } return _fn(n); }

let count2 = 0; function fn2(n) { count2++; if (n == 1 || n == 2) { return 1; } return fn2(n - 1) + fn2(n - 2); }

console.log(fn(20), count); // 6765 20 console.log(fn2(20), count2); // 6765 13529

```

快排和二分查找

前端中面试排序和查找的可能性比较小，因为 JS 引擎已经把这些常用操作优化得很好了，可能项目中你费劲写的一个排序方法，都不如`Array.sort`速度快且代码少。因此，掌握快排和二分查找就可以了。

快排和二分查找都基于一种叫做「分治」的算法思想，通过对数据进行分类处理，不断降低数量级，实现`O(logN)`（对数级别，比`O(n)`这种线性复杂度更低的一种，快排核心是二分法的`O(logN)`，实际复杂度为`O(N*logN)`）的复杂度。

快速排序

快排大概的流程是：

1. 随机选择数组中的一个数 A，以这个数为基准
2. 其他数字跟这个数进行比较，比这个数小的放在其左边，大的放到其右边
```

3. 经过一次循环之后，A 左边为小于 A 的，右边为大于 A 的
4. 这时候将左边和右边的数再递归上面的过程

具体代码如下：

```

// 划分操作函数 function partition(array, left, right) { // 用index取中间值而非splice const pivot = array[Math.floor((right + left) / 2)] let i = left let j = right

```
while (i <= j) {
    while (compare(array[i], pivot) === -1) {
        i++
    }
    while (compare(array[j], pivot) === 1) {
        j--
    }
    if (i <= j) {
        swap(array, i, j)
        i++
        j--
    }
}
return i
```

}

// 比较函数 function compare(a, b) { if (a === b) { return 0 } return a < b ? -1 : 1 }

function quick(array, left, right) { let index if (array.length > 1) { index = partition(array, left, right) if (left < index - 1) { quick(array, left, index - 1) } if (index < right) { quick(array, index, right) } } return array } function quickSort(array) { return quick(array, 0, array.length - 1) }

// 原地交换函数，而非用临时数组 function swap(array, a, b) { ;[array[a], array[b]] = [array[b], array[a]] } const Arr = [85, 24, 63, 45, 17, 31, 96, 50]; console.log(quickSort(Arr)); // 本版本来自：https://juejin.im/post/5af4902a6fb9a07abf728c40#heading-12

```

二分查找

二分查找法主要是解决「在一堆有序的数中找出指定的数」这类问题，不管这些数是一维数组还是多维数组，只要有序，就可以用二分查找来优化。

二分查找是一种「分治」思想的算法，大概流程如下：

1. 数组中排在中间的数字 A，与要找的数字比较大小
2. 因为数组是有序的，所以： a) A 较大则说明要查找的数字应该从前半部分查找 b) A 较小则说明应该从查找数字的后半部分查找
3. 这样不断查找缩小数量级（扔掉一半数据），直到找完数组为止

> 题目：在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

``` function Find(target, array) { let i = 0; let j = array[i].length - 1; while (i < array.length && j >= 0) { if (array[i][j] < target) { i++; } else if (array[i][j] > target) { j--; } else { return true; } } return false; }

//测试用例 console.log(Find(10, [ [1, 2, 3, 4], [5, 9, 10, 11], [13, 20, 21, 23] ]) );

```

另外笔者在面试中遇见过下面的问题：

> 题目：现在我有一个 1~1000 区间中的正整数，需要你猜下这个数字是几，你只能问一个问题：大了还是小了？问需要猜几次才能猜对？

拿到这个题目，笔者想到的就是电视上面有个「猜价格」的购物节目，在规定时间内猜对价格就可以把实物抱回家。所以问题就是让面试官不停地回答我猜的数字比这个数字大了还是小了。这就是二分查找！

猜几次呢？其实这个问题就是个二分查找的算法时间复杂度问题，二分查找的时间复杂度是`O(logN)`，所以求`log1000`的解就是猜的次数。我们知

道`2^10=1024`，所以可以快速估算出：`log1000`约等于 10，最多问 10 次就能得到这个数！

面试遇见不会的算法问题怎么办

面试的时候，在遇见算法题目的时候，应该揣摩面试官的意图，听好关键词，比如：有序的数列做查找、要求算法复杂度是`O(logN)`这类一般就是用二分的思想。

一般来说算法题目的解题思路分以下四步：

1. 先降低数量级，拿可以计算出来的情况（数据）来构思解题步骤
2. 根据解题步骤编写程序，优先将特殊情况做好判断处理，比如一个大数组的问题，如果数组为两个数长度的情况
3. 检验程序正确性
4. 是否可以优化（由浅到深），有能力的话可以故意预留优化点，这样可以体现个人技术能力

正则匹配解题

很多算法题目利用 ES 语法的特性来回答更加简单，比如正则匹配就是常用的一种方式。笔者简单通过几个真题来汇总下正则的知识点。

> 题目：字符串中第一个出现一次的字符

请实现一个函数用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符「go」时，第一个只出现一次的字符是「g」。当从该字符流中读出前六个字符「google」时，第一个只出现一次的字符是「l」。

这个如果用纯算法来解答需要遍历字符串，统计每个字符出现的次数，然后按照字符串的顺序来找出第一次出现一次的字符，整个过程比较繁琐，如果用正则就简单多了。

``` function find(str){ for (var i = 0; i < str.length; i++) { let char = str[i] let reg = new RegExp(char, 'g'); let l = str.match(reg).length if(l===1){ return char } } }

```

当然，使用`indexOf/lastIndexOf`也是一个取巧的方式。再来看一个千分位问题。

> 题目：将`1234567` 变成 `1,234,567`，即千分位标注

这个题目可以用算法直接来解，如果候选人使用正则来回答，这样主动展现了自己其他方面的优势，即使不是算法解答出来的，面试官一般也不会太难为他。这道题目可以利用正则的「零宽断言」`(?=exp)`，意思是它断言自身出现的位置的后面能匹配表达式 exp。数字千分位的特点是，第一个逗号后面数字的个数是3的倍数，正则：`/(\d{3})+$/`；第一个逗号前最多可以有 1~3 个数字，正则：`/\d{1,3}/`。加起来就是`/\d{1,3}(\d{3})+$/`，分隔符要从前往后加。

对于零宽断言的详细介绍可以阅读「零宽断言」这篇文章。

``` function exchange(num) { num += ''; //转成字符串 if (num.length <= 3) { return num; }

```
num = num.replace(/\d{1,3}(?=(\d{3})+$)/g, (v) => {
    console.log(v)
    return v + ',';
});
return num;
```

}

console.log(exchange(1234567));

```

当然上面讲到的多数是算法题目取巧的方式，下面这个题目是纯正则考查，笔者在面试的过程中碰见过，这里顺便提一下。

> 题目，请写出下面的代码执行结果

``` var str = 'google'; var reg = /o/g; console.log(reg.test(str)) console.log(reg.test(str)) console.log(reg.test(str))

```

代码执行后，会发现，最后一个不是为`true`，而是`false`，这是因为`reg`这个正则有个`g`，即`global`全局的属性，这种情况下`lastIndex`就发挥作用了，可以看下面的代码执行结果就明白了。

``` console.log(reg.test(str), reg.lastIndex) console.log(reg.test(str), reg.lastIndex) console.log(reg.test(str), reg.lastIndex)

```

实际开发中也会犯这样的错误，比如为了减少变量每次都重新定义，会把用到的变量提前定义好，这样在使用的时候容易掉进坑里，比如下面代码：

``` (function(){ const reg = /o/g; function isHasO(str){ // reg.lastIndex = 0; 这样就可以避免这种情况 return reg.test(str) } var str = 'google'; console.log(isHasO(str)) console.log(isHasO(str)) console.log(isHasO(str)) }())

```

小结

本小节介绍了数据结构和算法的关系，作为普通的前端也应该学习数据结构和算法知识，并且顺带介绍了下正则匹配。具体来说，本小节梳理了以下几部分数据结构和算法知识点：

1. 经常用到的数据结构有哪些，它们的特点有哪些
2. 递归和枚举是最基础的算法，必须牢牢掌握
3. 排序里面理解并掌握快速排序算法，其他排序算法可以根据个人实际情况大概了解
4. 有序查找用二分查找
5. 遇见不会的算法问题，先缩小数量级，然后分析推导

当然算法部分还有很多知识，比如动态规划这些算法思想，还有图和树常用到的广度优先搜索和深度优先搜索。这些知识在前端面试和项目中遇见得不多，感兴趣的读者可以在梳理知识点的时候根据个人情况自行决定是否复习。