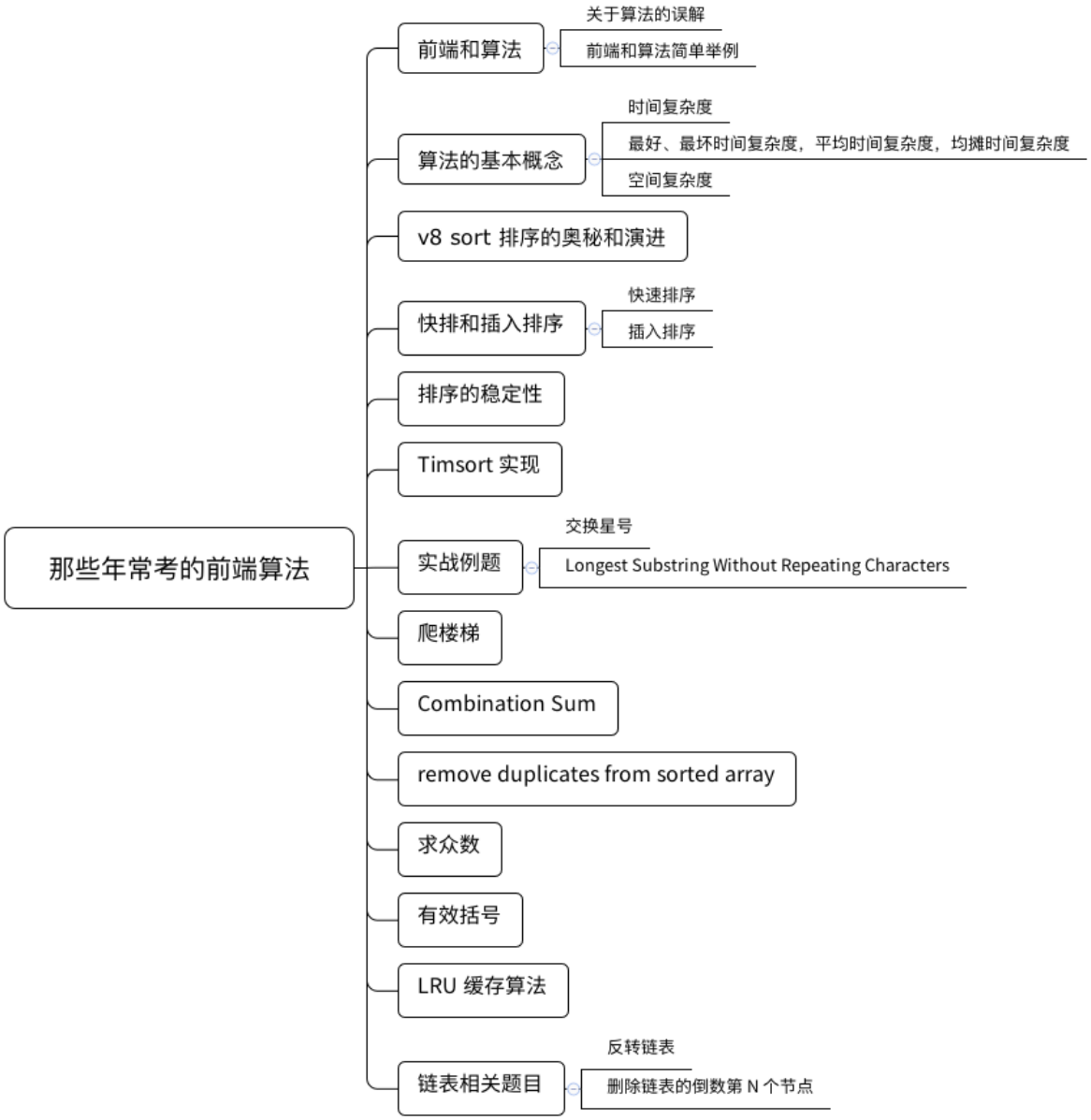
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

那些年常考的前端算法（1）

在上一讲中，我们全面梳理了重要的数据结构，并用 JavaScript 对各种数据结构进行了实现和方法模拟。数据结构常与算法一起出现，两者相互依存密不可分。这一讲，我们来研究一下「那些年常考的前端算法」。

主要内容如下：



我们将通过三讲的内容来剖析算法问题。本讲先「轻松」一下，主要介绍前端和算法的关系，以及算法中的一些基本概念。

前端和算法

前端和算法有什么关系呢？我想先纠正两个常见的错误认知。

关于算法的误解

前端没有算法?

「前端没有算法」这种说法往往出自算法岗甚至后端读者，这种认知是错误的。前端不仅有算法，而且算法在前端开发中占据的地位也越来越重要。我们常提到的 Virtual dom diff、webpack 实现、React fiber、React hooks、响应式编

程、浏览器引擎工作方式等都有算法的影子。在业务代码中，哪怕写一个抽奖游戏，写一个混淆函数都离不开算法。

算法重要不重要?

有读者认为，前端中算法只是提供了一些偏底层的能力和实现支持，我在业务开发中真正使用到算法的场景也很有限。事实上，不仅单纯的前端业务，哪怕对于后端业务来说，真正让你「徒手」实现一段算法的场景也不算多。但是据此得出算法不重要的说法还是太片面了。为什么高阶面试中总会问到算法呢？因为算法很好地反应了候选者编程思维和计算机素养；另一方面，如果我们想进阶，算法也是必须要攻克的一道难关。

前端和算法简单举例

我就先举一个例子作为引子，一起先热热身，看看算法应用在前端开发中的一个小细节。

想必不少读者写过「抽奖」代码，或者「老虎机」转盘。其中可能会涉及到一个问题，就是：

「如何将一个 JavaScript 数组打乱顺序？」

事实上乱序一个数组不仅仅是前端课题，那么这个问题在前端的背景下，有哪些特点呢？可能有读者首先想到使用数组的 sort API，再结合 Math.random 实现：

```
[12,4,16,3].sort(function() {
   return .5 - Math.random();
})
```

这样的思路非常自然，但也许你不知道：这不是真正意义上的完全乱序。

为此我们进行验证，对数组

```
let letters = ['A','B','C','D','E','F','G','H','I','J']
```

使用 array.sort 方法进行了 10000 次乱序处理，并对乱序之后得到的新数组中，每一个字母出现的位置进行统计，并可视化输出：

Results	slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7	slot 8	slot 9
A's	0	0	0	0	0	0	0	0	0	0
B's	0	0	0	0	0	0	0	0	0	0
C's	0	0	0	0	0	0	0	0	0	0
D's	0	0	0	0	0	0	0	0	0	0
E's	0	0	0	0	0	0	0	0	0	0
F's	0	0	0	0	0	0	0	0	0	0
G's	0	0	0	0	0	0	0	0	0	0
H's	0	0	0	0	0	0	0	0	0	0
I's	0	0	0	0	0	0	0	0	0	0
J's	0	0	0	0	0	0	0	0	0	0

recalculate

得到结果：

Results	slot 0	slot 1	slot 2	slot 3	slot 4	slot 5	slot 6	slot 7	slot 8	slot 9
A's	2943	2767	1956	1158	568	336	141	74	39	18
B's	2852	2937	1932	1126	567	306	145	81	43	11
C's	1978	1965	2208	1642	1071	576	296	159	69	36
D's	1109	1117	1710	2222	1755	1011	530	319	142	85
E's	567	587	1083	1691	2225	1722	1078	591	292	164
F's	273	334	558	1039	1739	2366	1744	1043	606	298
G's	149	148	294	600	1018	1787	2493	1820	1094	597
H's	75	73	144	299	616	1029	1825	2632	2023	1284
I's	28	51	70	141	291	568	1116	2100	3145	2490
J's	26	21	45	82	150	299	632	1181	2547	5017

不管点击按钮几次，你都会发现整体乱序之后的结果绝对不是「完全随机」。

比如，A 元素大概率出现在数组的头部，J 元素大概率出现在数组的尾部，所有元素大概率停留在自己初始位置。

这是为什么呢?

究其原因，在 Chrome v8 引擎源码中，可以清晰看到：

v8 在处理 sort 方法时，使用了插入排序和快排两种方案。当目标数组长度小于 10（不同版本有差别）时，使用插入排序；反之，使用快排。

其实不管用什么排序方法，大多数排序算法的时间复杂度介于 O(n) 到 O(n2) 之间，元素之间的比较次数通常情况下要远小于 n(n-1)/2，也就意味着有一些元素之间根本就没机会相比较（也就没有了随机交换的可能），这些 sort 随机排序的算法自然也不能真正随机。

通俗地说，其实我们使用 array.sort 进行乱序，理想的方案或者说纯乱序的方案是：数组中每两个元素都要进行比较，这个比较有 50% 的交换位置概率。如此

一来，总共比较次数一定为 n(n-1)。

而在 sort 排序算法中，大多数情况都不会满足这样的条件，因此当然不是完全随机的结果了。

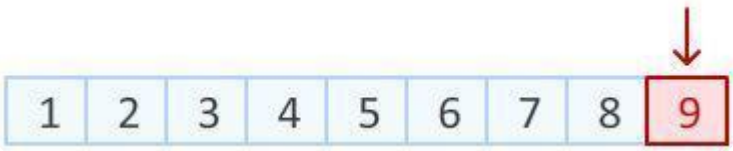
那为了满足乱序一个数组的需求，我们应该怎么做呢?

Fisher–Yates shuffle 洗牌算法——会是一个更好的选择。这里，我们简单借助图形来理解，非常简单直观。接下来就会明白为什么这是理论上的完全乱序（图片来源于网络）。

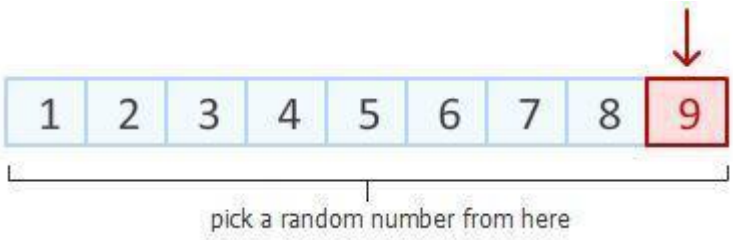
首先我们有一个已经排好序的数组：

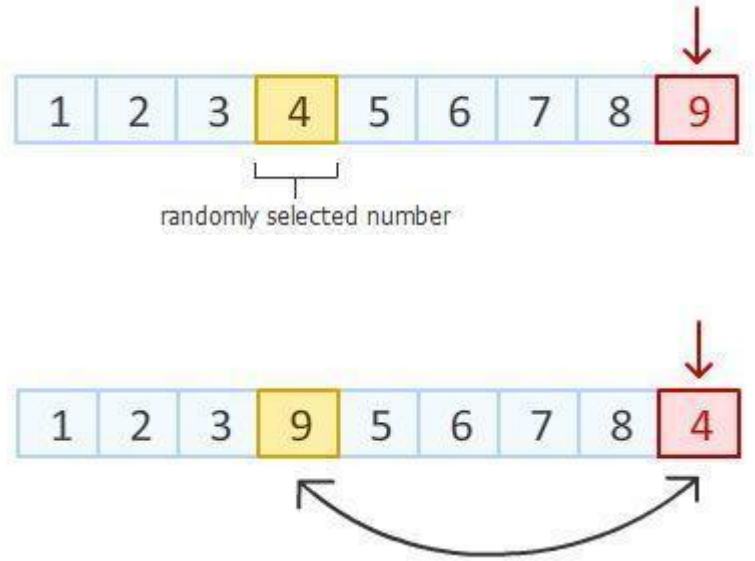
Step1：

这一步需要做的就是，从数组末尾开始，选取最后一个元素。



在数组一共 9 个位置中，随机产生一个位置，该位置元素与最后一个元素进行交换。

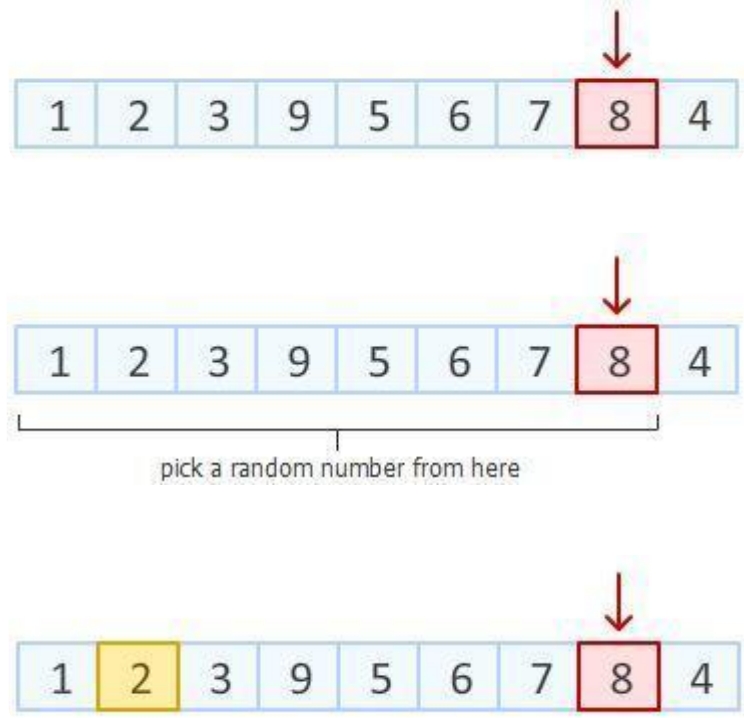




Step2：

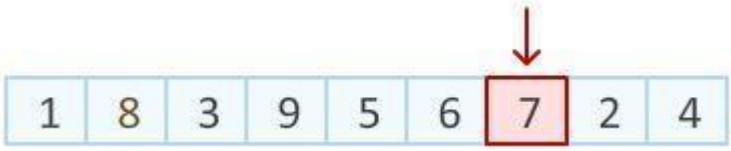
在上一步中，我们已经把数组末尾元素进行随机置换。

接下来，对数组倒数第二个元素动手。在除去已经排好的最后一个元素位置以外的 8 个位置中，随机产生一个位置，该位置元素与倒数第二个元素进行交换。



Step3：

理解了前两步，接下来就是依次进行，如此简单。



明白了原理，代码实现也很简单：

```
Array.prototype.shuffle = function() {
    var array = this;
    var m = array.length,
        t, i;
    while (m) {
        i = Math.floor(Math.random() * m--);
        t = array[m];
        array[m] = array[i];
        array[i] = t;
    }
    return array;
}
```

算法的基本概念

在具体讲解各种算法前，我们有必要先掌握基本概念。搞定算法，需要读者优先了解数据结构以及各种结构的相关方法，这些内容上一讲中已经进行了梳理。另外一个重要概念就是算法复杂度了，它是评估一个算法优秀程度的重要考证。我们常说的时间复杂度和空间复杂度该如何理解呢?

时间复杂度

我们先看一下时间复杂度的概念：

一个算法的时间复杂度反映了程序运行从开始到结束所需要的时间。把算法中基本操作重复执行的次数（频度）作为算法的时间复杂度。

但是时间复杂度的计算既可以「有理可依」，又可以靠「主观感觉」。通常我们认为：

没有循环语句，时间复杂度记作 O(1)，我们称为常数阶；

只有一重循环，那么算法的基本操作的执行频度与问题规模 n 呈线性增大关系，记作 O（n），也叫线性阶。

那么如何让时间复杂度的计算「有理可依」呢？来看几个原则：

只看循环次数最多的代码

加法法则：总复杂度等于量级最大的那段代码的复杂度

乘法法则：嵌套代码的复杂度等于嵌套内外复杂度的乘积

我们来逐一分析：

```
const cal = n => {
  let sum = 0
  let i = 1
  for (; i <= n; ++i) {
    sum = sum + i
  }
  return sum
}
```

执行次数最多的是 for 循环及里面的代码，执行了 n 次，应该「只看循环次数最多的代码」原则，因此时间复杂度为 O(n)。

```
const cal = n => {
  let sum1 = 0
  let p = 1

  for (; p < 100; ++p) {
    sum1= sum1 + p
```

```
  }

  let sum2 = 0
  let q = 1
  for (; q < n; ++q) {
    sum2 = sum2 + q
  }

  let sum3 = 0
  let i = 1
  let j = 1
  for (; i <= n; ++i) {
    j = 1
    for (; j <= n; ++j) {
      sum3 = sum3 +  i * j
    }
  }

  return sum1 + sum2 + sum3
}
```

上述代码分别对 sum1、sum2、sum3 求和：

对于 sum1 求和，循环 100 次，常数执行时间，时间复杂度为 O(1)；

对于 sum2 求和，循环规模为 n，时间复杂度为 O(n)；

对于 sum3 求和，两层循环，时间复杂度为 $O(n^2)$。

因此 $O(1) + O(n) + O(n^2)$，取三段代码的最大量级，上面例子最终的时间复杂度为 $O(n^2)$。

对于代码：

```
const cal = n => {
  let ret = 0
```

```
  let i = 1
  for (; i < n; ++i) {
    ret = ret + f(i); // 注意  f(i)
  }
}

const f = n => {
 let sum = 0
 let i = 1
 for (; i < n; ++i) {
   sum = sum + i
 }
 return sum
}
```

方法 cal 循环里面调用 f 方法，而 f 方法里面也有循环，这时应用第三个原则——乘法原则，得到时间复杂度 $O(n^2)$。

最后我们再看一个对数阶的概念：

```
const aFun = n => {
 let i = 1;
 while (i <= n)  {
    i = i * 2
 }
 return i
}

const cal = n => {
  let sum = 0
  for (let i = 1; i <= n; ++i) {
    sum = sum + aFun(n)
  }
  return sum
}
```

这里的不同之处是 aFun 每次循环，i = i * 2，那么自然不再是全遍历。想想高中学过的等比数列：

```
2^0 * 2^1 * 2^2 * 2^k * 2^x = n
```

因此，我们只要知道 x 值是多少，就知道这行代码执行的次数了，通过 2x = n 求解 x，数学中求解得 x = log2n 。即上面代码的时间复杂度为 O(log2n)。

但是不知道读者有没有发现：不管是以 2 为底，还是以 K 为底，我们似乎都把所有对数阶的时间复杂度都记为 O(logn)。这又是为什么呢?

事实上，基本的数学概念告诉我们：对数之间是可以互相转换的，log3n = log32 *log2n，因此 O(log3n) = O(C* log2n)，其中 C=log32 是一个常量。所以全部以 2 为底，并没有什么问题。

总之，需要读者准确理解：由于**时间复杂度**描述的是**算法执行时间与数据规模的增长变化趋势**，因而常量、低阶、系数实际上对这种增长趋势不产生决定性影响，所以在做时间复杂度分析时忽略这些项。

最好、最坏时间复杂度，平均时间复杂度，均摊时间复杂度

我们来看一段代码：

```
const find = (array, x) => {
   let pos = -1
   let len = array.length
   for (let i = 0 ; i < n; ++i) {
       if (array[i] === x) {
           pos = i
       }
   }
   return pos
}
```

上面的代码有一层循环，循环规模和 n 成线性关系。因此时间复杂度为 O(n)，我们改动代码为：

```
const find = (array, x) => {
   let pos = -1
   let len = array.length
   for (let i = 0 ; i < n; ++i) {
       if (array[i] === x) {
           pos = i
       }
   }
   return pos
}
```

在找到第一个匹配元素后，循环终止，那么时间复杂度就不一定是 O(n) 了，因此就有了最好时间复杂度、最坏时间复杂度的区别。针对上述代码最好时间复杂度就是 O(1)、最坏时间复杂度还是 O(n)。

最好时间复杂度、最坏时间复杂度其实都是极端情况，我们可以从统计学角度给出一个平均时间复杂度。在上述代码中，平均时间复杂度的计算方式应该是：

```
(1/(n+1)) * 1 + (1/(n+1)) * 2 + ... + (1/(n+1)) * n +
(1/(n+1)) * n
```

得到结果为：n(n+3)/2(n+1)

因为变量 x 出现在数组中的位置分别有 0 —— n－1 种情况，对应需要遍历的次数；除此之外，还有变量 x 不出现在数组中，这种情况仍然后遍历完数组。

上述结果简化之后仍然得到 O(n)。

我们再来看一段代码：

```
let array = new Array(n)
let count = 0
function insert(val) {
   let len = array.length
   if (count === len) {
       let sum = 0
```

```
        for (let i = 0; i < len; i++) {
            sum = sum + array[i]
        }
        array[0] = sum
        count = 1
    }
    array[count] = val
    ++count
}
```

这段代码逻辑很简单：我们实现了一个往数组中插入数据的功能。但是多了些判断：当数组满了之后，即 count === len 时，采用 for 循环对数组进行求和，求和完毕之后：先清空数组，然后将求和之后的结果放到数组的第一个位置，最后再将新的数据插入。

这是一段非常典型的代码，我们来看它的时间复杂度：

最好时间复杂度

数组中有空闲，count !== len，直接执行插入操作，复杂度为 O(1)。

最好时间复杂度

数组已满，count === len，需要先遍历一遍再求和，复杂度为 O(n)。

平均时间复杂度

假设数组长度为 n，数组空闲时，复杂度为 O(1)；数组已满，复杂度为 O(n)。采用平均加权方式：

```
(1/(n+1)) * 1 + (1/(n+1)) * 1 + ... + (1/(n+1)) * n
```

公式求和仍为 O(1)，主观上想：我们的操作是在进行了 n 个 O(1) 的插入操作后，此时数组满了，执行一次 O(n) 的求和和清空操作。这样一来，其实前面的 n 个 O (1) 和最后的 1 个 O(n) 其实是可以抵消掉的，这是一种均摊时间复杂度的概念。

这种均摊的概念是有实际应用场景的。例如，C++ 里的 vector 动态数组的自动扩容机制，每次往 vector 里 push 值的时候会判断当前 size 是否等于 capacity，一旦元素超过容器限制，则再申请扩大一倍的内存空间，把原来 vector 里的值复制到新的空间里，触发扩容的这次 push 操作的时间复杂度是 O(n)，但均摊到前面 n 个元素后，可以认为时间复杂度是 O(1) 常数。

最后总结一下，常见时间复杂度：

O(1)：基本运算 +、-、*、/、%、寻址

O(logn)：二分查找，跟分治（Divide & Conquer）相关的基本上都是 logn

O(n)：线性查找

O(nlogn)：归并排序，快速排序的期望复杂度，基于比较排序的算法下界

$O(n^2)$：冒泡排序，插入排序，朴素最近点对

$O(n^3)$：Floyd 最短路，普通矩阵乘法

$O(2^n)$：枚举全部子集

O(n!)：枚举全排列

O(logn) 近似于是常数的时间复杂度，当 n 为 2^{32} 的规模时 logn 也只是 32 而已； 对于顺序执行的语句或者算法，总的时间复杂度等于其中最大的时间复杂度。例如，$O(n^2) + O(n)$ 可直接记做 $O(n^2)$。

空间复杂度

空间复杂度表示算法的存储空间与数据规模之间的增长关系。常见的空间复杂度：O(1)、O(n)、$O(n^2)$，像 O(logn)、O(nlogn) 这样的对数阶复杂度平时都用不到。有的题目在空间上要求 in-place（原地），是指使用 O(1) 空间，在输入的空间上进行原地操作，比如字符串反转。但 in-place 又不完全等同于常数的空间复杂度，比如数组的快排认为是 in-place 交换，但其递归产生的堆栈的空间是可以不考虑的，因此 in-place 相对 O(1) 空间的要求会更宽松一点。

对于时间复杂度和空间复杂度，开发者应该有所取舍。在设计算法时，可以考虑「牺牲空间复杂度，换取时间复杂度的优化」，反之依然。空间复杂度我们不再过多介绍。

总结

本讲我们介绍了算法的基本概念，重点就是时间复杂度和空间复杂度分析，同时剖出了一个「乱序数组」算法进行热身。算法的大门才刚刚打开，请读者继续保持学习。

点击查看下一节

那些年常考的前端算法