48 前端算法知识脉络梳理+好题精做

更新时间：2020-08-04 09:29:58

> 生活永远不像我们想像的那样好，但也不会像我们想像的那样糟。——莫泊桑

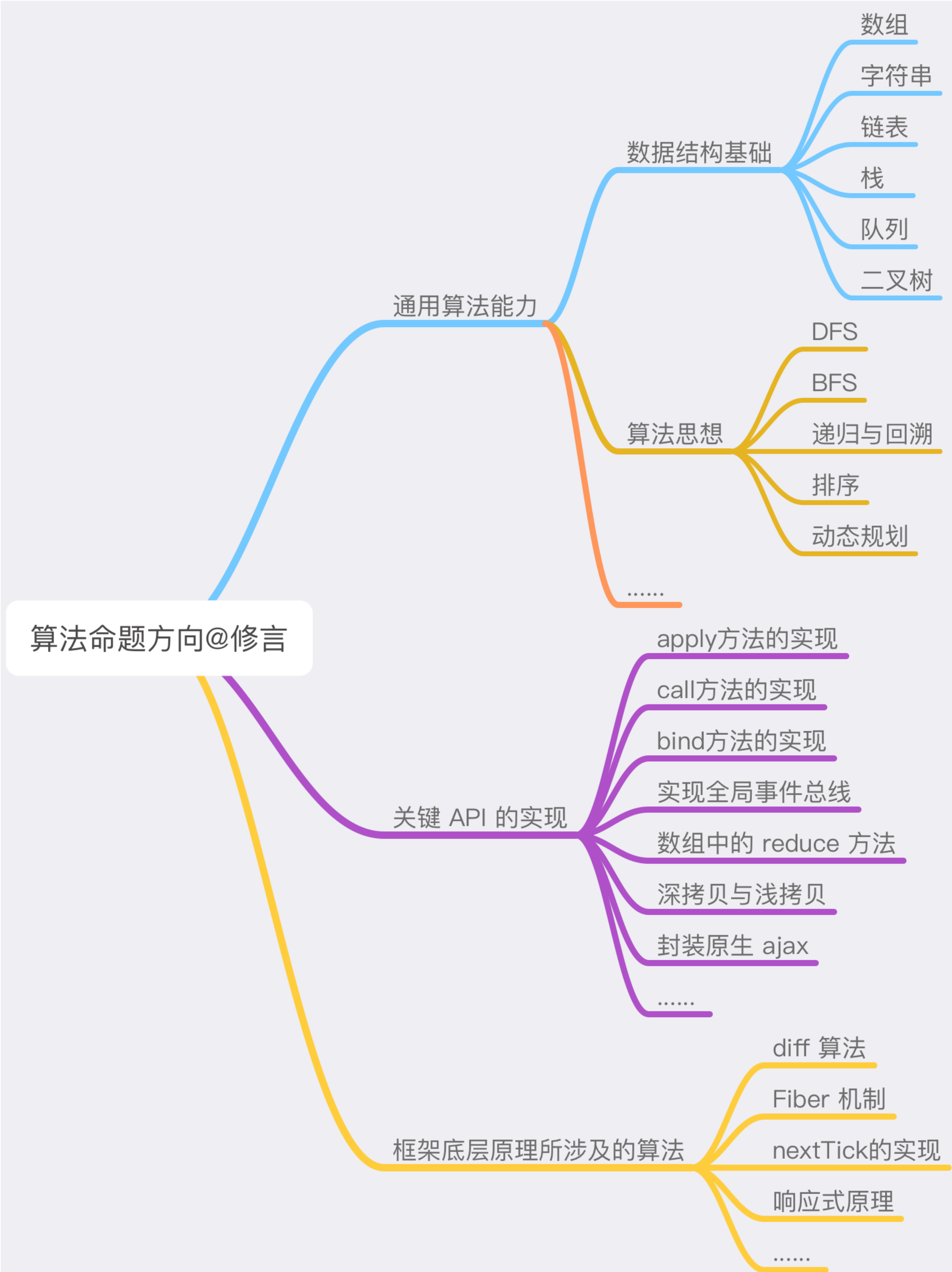
随着前端应用复杂度的提高，前端工程师的 Job-Model 渐渐与软件工程师趋同。基于此，围绕计算机通用能力的考察在前端面试中越来越普遍，前有设计模式，后有数据结构与算法。

前端算法知识脉络梳理

所谓“算法”，指的是解题方案的准确而完整的描述。算法的范畴是比较广泛的，它并不仅仅局限与 LeetCode 上面的一问一答。就前端而言，算法的考察整体上三个大的方向：

1. 通用算法能力
2. 关键 API 的实现
3. 框架底层原理所涉及的算法（重在理解）

基于这三个大的方向，可以细分出许许多多个小的命题点：



这张导图可以作为大家后续深入挖掘算法这个命题点的一个依据。这其中，2和3的内容我们在专栏的其它小节中多少已经有些涉及，而“通用算法能力”这部分目前来说还是一张白纸，因此本节会选取“通用算法能力”这个方向相关的命题热点为大家作进一步讲解。

排序

如何将一个乱序数组变得有序（有序在不经特别说明的情况下，指的都是从小到大排列）？这里我们讲三种必须掌握的方法：

冒泡排序

冒泡排序的过程，就是循环对比相邻的两个数据项。如果发现第一个比第二个大，则交换两个数据项的位置。较大的数据项不断向上移动到正确的位置，就好像是气泡浮出水面一样，因此这种排序方法被称为“冒泡排序”

冒泡排序编码实现如下（解析在注释里）：

```
function bubbleSort(arr) {
  // 缓存数组长度
  const len = arr.length
  // 外层循环，n个元素就要循环n次，每次确定的是索引为 len-1-i 这个坑位上的正确元素值
  for(let i=0;i<len;i++) {
    // 内层循环，逐个对比相邻两个数的大小
      for(let j=0; j<len-1-i; j++) {
      // 如果靠前的数字大于靠后的数字，则交换两者的位置
      if(arr[j] > arr[j+1]) {
        const temp = arr[j]
        arr[j] = arr[j+1]
        arr[j+1] = temp
      }
    }
  }
  return arr
}
```

注意，交换数组中两个数的位置，这里我给出的是 ES2015 之前比较常用的一种做法：

```
const temp = arr[j]
arr[j] = arr[j+1]
arr[j+1] = temp
```

在 ES2015 中，我们还可以用数组解构来实现同样的效果：

```
[arr[j], arr[j+1]] = [arr[j+1], arr[j]]
```

选择排序

选择排序的思路是：首先定位到数组的最小值，把它放在第一个坑位；接着排查第二个到最后一个元素，找出第二小的值，把它放在第二个坑位；循环这个过程，直至数组的所有坑位被重新填满为止。

选择排序的编码实现如下（解析在注释里）：

```
function selectSort(arr) {
  // 缓存数组长度
  const len = arr.length
  // 定义 minIndex，缓存当前区间最小值的索引，注意是索引
  let minIndex
  // 遍历数组中的前 n-1 个元素
  for(let i=0; i<len-1; i++) {
    // 初始化 minIndex 为当前区间第一个元素
    minIndex = i
    // i、j分别定义当前区间的上下界，i是左边界，j是右边界
    for(let j=i; j< len; j++) {
      // 若 j 处的数据项比当前最小值还要小，则更新最小值索引为 j
      if(arr[j] < arr[minIndex]) {
        minIndex = j
      }
    }
    // 如果 minIndex 发生过更新，则将 minIndex 置于当前排序区间的头部
    if(minIndex !== i) {
      [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]
    }
  }
  return arr
}
```

插入排序

插入排序的概念比较拗口，我们用一个例子来理解它。

看下面这个数组：

```
[5,1,1,2,0,0]
```

插入排序的思想是，从第一个数据项开始，每次插入一个靠后的邻近数据项。现在我们从 5 开始，尝试插入它后面最近的一个数据项：1。

由于 1<5，所以我们将两个元素交换位置，然后数组就会变成下面这样：

```
[1,5,1,2,0,0]
```

我们发现，此时数组的前两项已经是有序的。接着看第三个数（1），把它和第二个数（5）进行对比，发现它比5要小，因此交换第二个数和第三个数的位置：

```
[1,1,5,2,0,0]
```

交换完之后，继续试图向前对比，发现1和1是相等的，故不必再移动元素位置。

此时数组的前三项已经是有序的。其实整个插入排序的过程，就是**反复地基于前面已经有序的序列，尝试插入后一个元素，并且在有序序列中为这个新元素找到合适的位置**。

现在我们尝试插入第四个数（2），用它和最近的有序序列元素（5）做比较，发现2比5小，因此交换两者的位置：

```
[1,1,2,5,0,0]
```

交换完之后，继续试图向前对比，发现2比前面的1大，符合排序规则，所以当前位置就是2应该待的位置，不必再移动元素。

接着尝试插入第五个数（0），用它和最近的有序序列元素（5）做比较，发现0比5小，因此交换两者的位置；交换完毕后，继续向前对比，发现0比2小，交换两数；继续向前对比，发现0比1小，继续交换两数。交换到最后，我们会发现第一个坑才是0的正确位置：

```
[0,1,1,2,5,0]
```

接下来按照同样的思路，最后一个数字0也会被定位到数组的前面去：

```
[0,0,1,1,2,5]
```

以上便是插入排序的全过程。基于这个过程，我们来重新理解一下插入排序的定义：

> 插入排序每次排一个数组项，以此方式构建最后的排序数组。假定第一项已经排序了，接着用它和第二项作比较，使头两项能够正确排序，接着再和第三项比较，以此类推

插入排序的编码实现如下（解析在注释里）：

```
function insertSort(arr) {
  // 缓存数组长度
  const len = arr.length
  // temp 用来保存当前插入的新元素
  let temp
  // i用于标识每次被插入的元素的索引
  for(let i=1;i<len;i++) {
    // j用于帮助 temp 寻找自己应该有的定位
    let j=i
    temp = arr[i]
    // 判断 j 前面一个元素是否比 temp 大
    while(j>0 && arr[j-1]>temp) {
      // 如果是，则将 j 前面的一个元素后移一位，为 temp 让出位置
      arr[j] = arr[j-1]
      j--
    }
    // 循环让位，最后得到的 j 就是 temp 的正确索引
    arr[j] = temp
  }
  return arr
}
```

快速排序

以上三种排序算法，相对来说思路都比较简单，对应的整体时间复杂度也比较高（O(n^2)。接下来要介绍一种性能更好，也更常用的排序算法——快速排序。

快速排序的核心思想是“分而治之”，具体操作办法是把原始的数组筛选成较小和较大的两个子数组，然后递归地排序两个子数组。这个概念初学者可能会觉得比较抽象，我们照样是来看一个例子：

尝试排序以下数组：

```
[5,1,3,6,2,0,7]
```

首先要做的事情就选取一个基准值。基准值的选择有很多方式，这里我们选取数组中间的值：

```
[5, 1, 3, 6, 2, 0, 7]
↑     基准     ↑
```

左右指针分别指向数组的两端。接下来我们要做的，就是先移动左指针，直到找到一个比基准值大的值；然后再移动右指针，直到找到一个比主元小的值。

首先我们来看左指针，5比6小，故左指针右移一位：

```
[5, 1, 3, 6, 2, 0, 7]
  ↑   基准     ↑
```

继续对比，1比6小，继续右移左指针：

```
[5, 1, 3, 6, 2, 0, 7]
    ↑ 基准     ↑
```

继续对比，3比6小，继续右移左指针，左指针最终指向了基准值：

```
[5, 1, 3, 6, 2, 0, 7]
      基准     ↑
       ↑
```

此时由于 6=6，左指针停止移动。开始看右指针：

右指针指向7，7<6，故左移右指针：

```
[5, 1, 3, 6, 2, 0, 7]
      基准   ↑
       ↑
```

发现 0 比 6 小，停下来，交换 6 和 0，同时两个指针共同向中间走一步：

```
[5, 1, 3, 0, 2, 6, 7]
         ↑ 基准
         ↑
```

此时 2 比 6 小，故右指针不动，左指针继续前进：

```
[5, 1, 3, 0, 2, 6, 7]
         ↑ 基准
       right↑
           left
```

此时右指针所指的值小于 6，左指针所指的值满足大于等于6，故两个指针都不再移动。此时我们会发现，对左指针所指的数字来说，它左边的所有数字都比它小，右边的所有数字都比它大。接着我们以左指针为轴心，划分出两个子数组：

```
[5, 1, 3, 0, 2]
[6, 7]
```

针对两个子数组，重复执行以上操作，直到数组完全排序为止。这就是快速排序的整个过程。

快速排序的编码实现如下（解析在注释里）：

```
// 快速排序入口
function quickSort(arr, left = 0, right = arr.length - 1) {
  // 定义递归边界，若数组只有一个元素，则没有排序必要
  if(arr.length > 1) {
    // lineIndex表示下一次划分左右子数组的索引位
    const lineIndex = partition(arr, left, right)
    // 如果左边子数组的长度不小于1，则递归快排这个子数组
    if(left < lineIndex-1) {
      // 左子数组以 lineIndex-1 为右边界
      quickSort(arr, left, lineIndex-1)
    }
    // 如果右边子数组的长度不小于1，则递归快排这个子数组
    if(lineIndex<right) {
      // 右子数组以 lineIndex 为左边界
      quickSort(arr, lineIndex, right)
    }
  }
  return arr
}
// 以基准值为轴心，划分左右子数组的过程
function partition(arr, left, right) {
  // 基准值默认取中间位置的元素
  let pivotValue = arr[Math.floor(left + (right-left)/2)]
  // 初始化左右指针
  let i = left
  let j = right
  // 当左右指针不越界时，循环执行以下逻辑
  while(i<=j) {
    // 左指针所指元素若小于基准值，则右移左指针
    while(arr[i] < pivotValue) {
      i++
    }
    // 右指针所指元素大于基准值，则左移右指针
    while(arr[j] > pivotValue) {
      j--
    }

    // 若i<=j，则意味着基准值左边存在较大元素或右边存在较小元素，交换两个元素确保左右两侧有序
    if(i<=j) {
      swap(arr, i, j)
      i++
      j--
    }

  }
  // 返回左指针索引作为下一次划分左右子数组的依据
  return i
}

// 快速排序中使用 swap 的地方比较多，我们提取成一个独立的函数
function swap(arr, i, j) {
  [arr[i], arr[j]] = [arr[j], arr[i]]
}
```

动态规划

同学们现在可以回味一下快速排序的过程，它是“分治”思想的典型应用：把一个问题分解为相互独立的子问题，逐个解决子问题后，再组合子问题的答案，就得到了问题的最终解。

动态规划的思想和“分治”有点相似。不同之处在于，“分治”思想中，各个子问题之间是独立的：子数组之间的排序并不互相影响。而动态规划划分出的子问题，往往是相互依赖、相互影响的。

下面我们通过一道非常经典的动态规划题目，来认识动态规划解题中的基本要素：

爬楼梯问题

> 题目描述：假设楼梯一共有 n 层。每次只能爬 1 步 或 2 步，问有多少种爬到楼顶的方法

思路分析

在做算法题的时候，如果题目中仅仅问了“解决某个问题有多少种方法”或者“抵达某个坐标有多少条路径”，而不要求你列出方法和路径的内容，这时候要本能地想到用动态规划来做题。

我们可以用 f(n) 表示爬到第 n 层楼梯的方法数，那么爬到第 n-1 层楼梯的方法数对应的表示就是 f(n-1)，爬到第 n-2 层楼梯的方法数对应的表示就是 f(n-2)。f(n)、f(n-1) 和 f(n-2) 之间有着如下的关系：

```
f(n) = f(n-1) + f(n-2)
```

为什么会有这样一层关系？大家思考爬楼梯问题，不妨试试从后往前想：

如果我此刻就站在第 n 层楼梯上，我要往后退，有几种退法？

按照题目的要求，我每次只能后退一步或者两步。假如我后退一步，那么就来到了第 n-1 层楼梯；假如我后退了两步，那么就来到了第 n-2 层楼梯。这就意味着，如果我要抵达第 n 层楼梯，那么我只有两个可能的来路：

- 从第 n-1 层楼梯爬一步上来
- 从第 n-2 层楼梯爬两步上来

因此，爬到第 n 层楼梯的办法数，就是 f(n-1) 和 f(n-2) 相加的结果。

这一步，就是动**态规划中最关键的一步——找出递推公式，这个递推公式，学名叫“状态转移方程”，**它用于表**达不同子问题之间的关联。**

基于这个思路，我们继续倒推，会发现状态转移方程是具有通用性的：对任意的第 k(2<k<=n）层楼梯，都有以下公式：

```
f(k) = f(k-1) + f(k-2)
```

因此这道题的解法就是将 f(n) 转化为 f(n-1) + f(n-2) 两个子问题，然后再对子问题进行拆解：比如将 f(n-1) 转化为 f(n-1 -1) + f(n-1 -2)。这样依次递归，直到 n=1 或者 n=2 为止——这两种情况是特殊的，分别只有一种抵达方法。

基于这个思路，我们来写代码：

编码实现

```
// 用于存储不同楼层对应的解决办法个数
const f = []
// 定义爬楼梯方法
const climbStairs = function(n) {
  // 处理边界条件
  if(n==1) {
      return 1
  }
  if(n==2) {
      return 2
  }
  // 若 f(n) 不存在，则递归计算其值
  if(f[n]===undefined)  f[n] = climbStairs(n-1) + climbStairs(n-2)

  // 若 f(n) 已经有值，则直接返回
  return f[n]
};
```

注意：像楼上这种递归时记忆每个状态对应结果的解法，叫做“记忆化搜索”。这个过程是以“倒退”的形式从高层向低层反向推导，严格来说不能算是动态规划。我们把整个过程改为“前进”的形式，从低层向高层推导：

```
const climbStairs = function(n) {
    // 初始化结果数组
    const f = [];
    f[1] = 1;
    f[2] = 2;
    // 动态更新每一层楼梯对应的结果
    for(let i = 3;i <= n;i++){
        f[i] = f[i-2] + f[i-1];
    }
    return f[n];
};

以上便是这道题的标准动态规划解法了。
```

（注：记忆化搜索和动态规划，在思想上来说一脉相承，区别在于是“自顶向下”解决问题还是“自底向上”解决问题。）

硬币找零问题

> 题目描述：给出需要找零的钱数，你可以用指定面额的硬币来完成找零。问达成找零所需要的最少硬币个数
>
> 举例：给定的硬币面额分别是1、5、10、25（美分硬币），要求找零的钱数为36，那么我们最少可以用3个硬币（25、10、1）来完成找零。
>
> 提示：若题目无解，则返回 -1

思路分析

这道题是动态规划中一个非常有代表性的“最值”问题。求解最值问题，我们的思路和楼上非常相似，仍然是首先去思考这个状态转移方程怎么写。

状态转义方程的分析，本质上是对子问题之间关联的分析。要想明确子问题之间的关联，最快的办法是“倒推”。就像解决爬楼梯问题时，我们首先思考的是如何站在第 n 层楼梯上倒退。这道题也一样，我们可以假装此时手里已经有了 36 美分，只是不清楚硬币的个数，把“如何凑到36”的问题转化为“如何从36减到0”的问题。

硬币的英文是 coin，因此我们这里用 c1、c2、c3…cn 分别来表示题目中给到我们的第 1-n 个硬币。现在我如果从 36 美分的总额中拿走一个硬币，那么有以下几种可能：

```
拿走 c1
拿走 c2
拿走 c3
......
拿走 cn
```

假如用 f(x）表示每一个总额数字对应的最少硬币数，那么我们可以得到以下的对应关系：

```
f(36) = Math.min(f(36-c1)+1,f(36-c2)+1,f(36-c3)+1......f(36-cn)+1)
```

这套对应关系，就是本题的状态转移方程。

找出了状态转移方程，我们接下来需要思考的是递归的边界条件：在什么情况下，我的“倒退”可以停下来。这里需要考虑的是硬币总额为0的情况，这种情况对应的硬币个数毫无疑问也会是0，因而不需要任何的回溯计算。

注意：虽然我们在明确状态转移方程的过程中，不可避免会用到递归的思想，但别忘了，递归总是在“自顶向下”解决问题。而动态规划要求我们“自底向上”解决问题。同时，这道题同时涉及到了指定范围内数字的枚举（对硬币总额的枚举）、对每个硬币面额的枚举等，因此用循环遍历来做是非常舒服的：

编码实现

（解析在注释里）

```
const coinChange = function(coins, amount) {
  // 用于保存每个目标总额对应的最小硬币个数
  const f = []
  // 提前定义已知情况
  f[0] = 0
  // 遍历 [1, amount] 这个区间的硬币总额
  for(let i=1;i<=amount;i++) {
    // 求的是最小值，因此我们预设为无穷大，确保它一定会被更小的数更新
    f[i] = Infinity
    // 循环遍历每个可用硬币的面额
    for(let j=0;j<coins.length;j++) {
      // 若硬币面额小于目标总额，则问题成立
      if(i-coins[j]>=0) {
        // 状态转移方程
        f[i] = Math.min(f[i],f[i-coins[j]]+1)
      }
    }
  }
  // 若目标总额对应的解为无穷大，则意味着没有一个符合条件的硬币总数来更新它，本题无解，返回-1
  if(f[amount]===Infinity) {
    return -1
  }
  // 若有解，直接返回解的内容
  return f[amount]
};
```

}

47 设计模式知识脉络梳理+好题精做

49 临门一脚：前端面试的本质是什么？