详解 JavaScript 实现快速排序(Quick Sort)算法

目录(Catalog)

生词(New Wrods)

- **swap [swɔp] --n.交换. --vt&vi.交换**
  - Will you swap(vt) places with me? 你愿意和我换位置吗?
  - do [make] a swap(n). 交换.
- **pivot ['pɪvət] --n.枢纽, 中心点**
  - pivot point: 枢轴点, 旋转点, 支点
- **partition [pɑ:'tɪʃ(ə)n]--n.分开, 分割 --vt.分开, 分割**
- **logarithmic [ˌlɔgə'riðmik] --adj.对数的**
  - Logarithmic functions 对数函数
- **computational [ˌkɑmpju'teʃənl] --adj.计算的**
  - computational errors. 计算错误
  - He turns to computational aids for help. 他运用计算工具.
- **complexity [kəm'pleksətɪ] --n.复杂, 复杂性**
  - complexity analysis 复杂度分析
  - There was lots of complexity built into his being. 有很多复杂因素构成了他这个人.
  - the complexities of family life. 家庭生活中的复杂因素.

内容(Content)

1. 快速排序介绍

- 快速排序也许是最常用的排序算法了. 它的复杂度为 $O(n\log(n))$, 且性能通常比其他复杂度为 $O(n\log(n))$ 的排序算法要好. 和归并排序一样, 快速排序也使用分而治之的方法, 将原始数组分为较小的数组 (但它并没有像归并排序那样将他们分割开). 快速排序比目前学过的其他排序算法要复杂一些.

  快速排序一般分下面 3 个步骤:

- (1) 首先, 从数组中选择一个子项作为 `主元/基准(pivot)` , 一般常用的是数组的中间项.
- (2) **创建 2 个指针, left 指向数组第一项的索引, right 指向数组最后一项的索引. 移动左指针直到找到一个比主元大的项, 接着移动右指针直到找到一个比主元小的项, 然后交换这两项, 重复这个过程, 直到左指针超过了右指针.** 这个过程将使得比主元小的项都排在主元之前, 而比主元大的项都排在主元之后. 这一步叫作 `划分(partition)` 操作.
- (3) 接着, 算法对划分后的小数组(较主元小的项组成的子数组, 以及较主元大的项组成的子数组) 重复之前的 2 个步骤, 直至数组完全排序.

2. 快速排序源码及讲解

- 实现代码来自《学习JavaScript数据结构与算法》

```
// - 定义两个数值比较结果的常量
const Compare = {
    LESS_THAN: -1,
    BIGGER_THAN: 1,
    EQUALS: 0
};
// - 辅助函数: 比较两个数的大小.
function defaultCompare(a, b) {
    if (a === b) {
        return Compare.EQUALS;
    }
    return a < b ? Compare.LESS_THAN : Compare.BIGGER_THAN;
}

// - swap 交换数组中的 2 个子项(值交换).
function swap(array, a, b) {
    const temp = array[a];
    array[a] = array[b];
    array[b] = temp;
    // - ES6 的方式
    // [array[a], array[b]] = [array[b], array[a]]
}

function quick(array, left, right, compareFn) {
    let index;                                          // {1}
    if (array.length > 1) {                             // {2}
        index = partition(array, left, right, compareFn);   // {3}
        if (left < index - 1) {                         // {4}
            quick(array, left, index - 1, compareFn);       // {5}
        }
        if (index < right) {                            // {6}
            quick(array, index, right, compareFn);          // {7}
        }
    }
    // console.log('array: ', array);
    return array;
}
// - partition 划分操作
function partition(array, leftIndex, rightIndex, compareFn) {
```

```
    const pivot = array[Math.floor((rightIndex + leftIndex) / 2)]; // {8}
    console.log('pivot: ', pivot);
    while (leftIndex <= rightIndex) {                              // {9}
        while (compareFn(array[leftIndex], pivot) === Compare.LESS_THAN) { //
{10}
            leftIndex++;                                           // {11}
            console.log('while leftIndex++', leftIndex);
        }
        while (compareFn(array[rightIndex], pivot) === Compare.BIGGER_THAN) {
// {12}
            rightIndex--;                                          // {13}
            console.log('while rightIndex--', rightIndex);
        }
        if (leftIndex <= rightIndex) {                             // {14}
            swap(array, leftIndex, rightIndex);                    // {15}
            leftIndex++;                                           // {16}
            rightIndex--;                                          // {17}
        }
    }
    console.log('leftIndex: ', leftIndex);
    console.log('rightIndex: ', rightIndex);
    console.log('\n');
    return leftIndex;                                             // {18}
}

function quickSort(array, compareFn = defaultCompare) {
    return quick(array, 0, array.length - 1, compareFn);
}

const array = [9, 6, 4, 3, 5, 7, 2];
const arr = quickSort(array);
console.log('arr:', arr);  // arr: [2, 3, 4, 5, 6, 7, 9]
```

- 代码讲解:
  1. 我们先来看一下, 最上面的两个函数 `defaultCompare()` 和 `swap()`, 它们都很简单,
     - `defaultCompare()` 用来比较两个数的大小, 例如: `num1`, `num2`, 如果 `num1 < num2` 返回 `-1` 反之则返回 `1`, 若两数相等返回 `0`; `Compare` 为自定义辅助对象.
     - `swap()`: 为交换数组中的两个子项.
  2. 接着来看调用示例, 首先调用 `quickSort()` 函数, 传入一个数组和 `deafultCompare` 函数指针, 此函数内部调用 `quick()` 函数, `quick()` 函数接受 4 个参数:
     - (1) 要排序的数组;
     - (2) `left` 指向数组第一个值的索引;
     - (3) `right` 指向数组最后一个值的索引;
     - (4) 第 4 个为我们自定义的 `defaultCompare` 函数的指针.
  3. 现在我们看 `行{1}`, 此处定义了一个变量 `index`, 这个变量的作用是帮助我们将子数组分离为较小值数组和较大值数组. 这样就能再次递归地调用 `quick` 函数了. 不明白? 先别着急, 下面还会讲到.
  4. `行{2}` 判断要排序的数组长度必须大于 1; 接着看 `行{3}`, 调用 `partition()` 函数, 把返回值赋值给 `index`, 现在进入到 `partition` 函数内部, 去看一下代码的逻辑;

5. `partition` (划分)函数实现的就是上面快速排序步骤中的的第 2 步, 根据左右指针所指项的大小和主元的值对比来进行排序, 先贴出 `partition` 函数的代码注解, 在下面讲代码执行步骤时可以当做参考:

```
  // - 划分过程: 第一件要做的事情是选择主元, 有好几种方式。最简单的一种是选择
  //   数组的第一个值(最左边的值). 然而, 研究表明对于几乎已排序的数组, 这不是一个
  //   好的选择, 它将导致该算法的最差表现. 另外一种方式是随机选择数组的一个值
  //   或是选择中间的值. 在本示例中, 我们选择中间值作为主元(行{8}).
  // - 只要 leftIndex 和 rightIndex 指针没有相互交错(行{9}), 就执行划分操作.
  //   首先, 移动 leftIndex 指针直到找到一个比主元大的元素(行{10}). 对 right 指针,
  //   我们做同样的事情, 移动 rightIndex 指针直到我们找到一个比主元小的元素(行{12}).
  // - 当左指针指向的元素比主元大且右指针指向的元素比主元小, 并且此时左指针索引没有
  //   右指针索引大时(行{14}), 意思是左项比右项大(值比较), 我们交换它们(行{15}),
  //   然后移动 2 个指针, 并重复此过程(从 行{9} 再次开始).
  // - 在划分操作结束后, 返回左指针的索引, 用来在 行{3} 处创建子数组.
  function partition(array, leftIndex, rightIndex, compareFn) {
      const pivot = array[Math.floor((rightIndex + leftIndex) / 2)]; // {8}
      console.log('pivot: ', pivot);
      while (leftIndex <= rightIndex) {                              // {9}
          while (compareFn(array[leftIndex], pivot) === Compare.LESS_THAN) {
// {10}
              leftIndex++;                                           // {11}
              console.log('while leftIndex++', leftIndex);
          }
          while (compareFn(array[rightIndex], pivot) === Compare.BIGGER_THAN)
{ // {12}
              rightIndex--;                                          // {13}
              console.log('while rightIndex--', rightIndex);
          }
          if (leftIndex <= rightIndex) {                             // {14}
              swap(array, leftIndex, rightIndex);                    // {15}
              leftIndex++;                                           // {16}
              rightIndex--;                                          // {17}
          }
      }
      console.log('leftIndex: ', leftIndex);
      console.log('rightIndex: ', rightIndex);
      console.log('\n');
      return leftIndex;                                            // {18}
  }
```

6. 再次熟悉快速排序步骤 2: **创建 2 个指针, left 指向数组第一项的索引, right 指向数组最后一项的索引. 移动左指针直到找到一个比主元大的项, 接着移动右指针直到找到一个比主元小的项, 然后交换这两项, 重复这个过程, 直到左指针超过了右指针.这个过程将使得比主元小的项都排在主元之前, 而比主元大的项都排在主元之后. 这一步叫作 *划分(partition)* 操作.**

   虽然上面 `partition` 函数有注解, 但看起来并不明朗, 我们执行调用示例, 对照着来讲解:

   先来看 `行{8}`, 我们**第一次传入的 array 是整个数组**, `leftIndex = 0`, `rightIndex = 6`, `pivot = array[3] = 3`

   - (1.) `行{9}` 第 1 轮循环: `leftIndex = 0` <= `rightIndex = 6` 为 `true`, 我们在要排序的数组上方添加上左右指针, 方便查看:

```
    L        P        R
[9, 6, 4, 3, 5, 7, 2]
```

(**Notice:**, *写成函数的形式是方便查看比较效果, 并没有其他意义.*)

$$f(1) = \begin{cases} 9 > 3, & \text{左指针的项大于主元, 行10 判断条件为 false. leftIndex 仍然等于 0} \\ 2 < 3, & \text{右指针的项小于主元, 行12 判断条件为 false. rightIndex 仍然等于数组的索引 6} \end{cases}$$

因为 `0 < 6` 所以 `行{14}` 的判断条件为 `true`, 执行 `行{15}` 交换 `9` 和 `2`; 接着

执行 `行{16}` `leftIndex + 1 = 1`, 然后向前移动左指针(即向右);

执行 `行{17}` `rightIndex - 1 = 5`, 然后向前移动右指针(即向左). 此时数组变为:

```
      L     P     R
[2, 6, 4, 3, 5, 7, 9]
```

此时, 代码执行到 `行{18}`, 返回 `leftIndex = 1`.

- (2.) `行{9}` 第 2 论循环: `leftIndex = 1` <= `rightIndex = 5` 为 `true`:

```
      L     P     R
[2, 6, 4, 3, 5, 7, 9]
```

$$f(2) = \begin{cases} 6 > 3, & \text{L 项大于主元, 行10 判断条件为 false. leftIndex 仍然等于 1} \\ 7 > 3, & \text{R 项不小于主元, 行12 判断条件为 true. 执行 行13, rightIndex = 5 - 1 = 4} \end{cases}$$

`行{13}` 执行完 `5 - 1 = 4` 后, 我们先把右指针向前移动一项, 接着仍执行此 `while` 语句, 目前数组指针更新为:

```
      L     P  R
[2, 6, 4, 3, 5, 7, 9]
```

我们接着比较,

$$f(3) = \begin{cases} 6 > 3, & \text{左指针无移动原地等待..., leftIndex 等于 1} \\ 5 > 3, & \text{R 项大于主元, 行12 判断条件为 true. 执行 行13, rightIndex = 4 - 1 = 3} \end{cases}$$

`行{13}` 执行后 `rightIndex = 3`, 我们先把右指针向前移动一项, 接着执行此 `while` 语句, 数组指针为:

```
      L    P/R
[2, 6, 4, 3, 5, 7, 9]
```

继续...

$$f(4) = \begin{cases} 6 > 3, & \text{左指针无移动原地等待..., leftIndex 仍然等与 1} \\ 3 = 3, & \text{右指针的项等于主元, 行12 判断条件为 false. rightIndex 仍然等于 3} \end{cases}$$

因为 `1 < 3` 所以 `行{14}` 的判断条件为 `true`, 执行 `行{15}`, 交换 `6` 和 `3`, 接着

执行 `行{16}` `leftIndex + 1 = 2`, 然后向前移动左指针(即向右);

执行 `行{17}` `rightIndex - 1 = 2`, 然后向前移动右指针(即向左). 此时数组和指针都变更为:

```
      P L/R
[2, 3, 4, 6, 5, 7, 9]
```

此时, 代码执行到 `行{18}`, 返回 `leftIndex = 2`.

- (3.) `行{9}` 第 3 论循环: `leftIndex = 2` <= `rightIndex = 2` 为 `true`:

```
      P L/R
[2, 3, 4, 6, 5, 7, 9]
```

$$f(5) = \begin{cases} 4 > 3, & \text{左指针的项大于主元, 行10 判断条件为 false. leftIndex 仍然等于 2} \\ 4 > 3, & \text{R 大于主元, 行12 判断条件为 true. rightIndex = 2 - 1 =1} \end{cases}$$

因为 `2` $\not<$ `1`, 所以 `行{14}` 判断条件为 `false`, 代码直接跳到 `行{18}` 返回 `leftIndex = 2`

- (4.) 执行完第 (3.) 步后, 代码又回到 `行{9}`, 现在 `leftIndex = 2`, `rightIndex = 1`, 判断条件为 `false`, 所以退出 `while` 循环, 至此整个 `partition` 函数执行完毕, 返回数字 `2` 给 `行{3}` 的 `index`.

7. 经过第 6 步后, `行{3}` 的 `index` 值为 `2`, 而且调整后的数组为:

```
[2, 3, 4, 6, 5, 7, 9]
```

现在我们接第 4 步的逻辑, 回到 `quick()` 函数内, 接着看 `行{4}` 和 `行{5}`:

```
if (left < index - 1) {                             // {4}
    quick(array, left, index - 1, compareFn);       // {5}
}
```

`left = 0`, `index - 1 = 2 - 1 = 1`. 判断条件为 `true`, 进入 `行{5}` 调用 `quick` 函数自身, 传入参数为 `(array, 0, 1, compareFn)`. 接下来执行 `行{1}` --> `行{2}` --> `行{3}`, 再次调用 `partition(array, 0, 1, compareFn);`, 因再次调用 `partition` 函数的执行流程和上面的 第 6 步是一样的, 这里直接给出执行流程,不再叙述:

- (1.) `行{9}` 第一轮循环: `leftIndex(0)` <= `rightIndex(1)` 为 `true`:

```
   L/P R
[2, 3, 4, 6, 5, 7, 9]
```

$$f(1) = \begin{cases} 2 = 2, & \text{L 项等于主元, 行10 判断条件为 false. leftIndex 仍等于 0} \\ 3 > 2, & \text{R 项大于主元, 行12 判断条件为 true. 执行 行13, rightIndex = 1 - 1 = 0} \end{cases}$$

`行{13}` 执行完毕 `rightIndex(0)`, 我们先把右指针向前移动一项, 接着仍执行此 `while` 语句, 目前数组的指针更新为:

```
  L/P/R
[2, 3, 4, 6, 5, 7, 9]
```

接着比较,

$$f(2) = \begin{cases} 2 = 2, & \text{左指针无移动原地等待... leftIndex 仍等于 } 0 \\ 2 = 2, & \text{R 项不大于主元, 行12 判断条件为 false. rightIndex 等于 } 0 \end{cases}$$

现在 `leftIndex(0)` = `rightIndex(0)`, 所以 `行{14} 的判断条件为` true `, 执行` 行{15} `, 交换 2 和 2`, 接着

执行 `行{16}` `leftIndex + 1 = 1`, 然后向前移动左指针(即向右);

执行 `行{17}` `rightIndex - 1 = -1`, 然后向前移动右指针(即向左). 此时数组和指针都变更为:

```
    P  L
[2, 3, 4, 6, 5, 7, 9]
```

来到 `行{18}` 返回 `leftIndex` (即: `1`).

- (2.) 到此, `行{4}` 和 `行{5}` 执行完毕, 我们接着往下看 `行{6}` 和 `行{7}`.

8. 先来看一下 `行{6}` 和 `行{7}` 的代码:

```
if (index < right) {                                // {6}
    quick(array, index, right, compareFn);          // {7}
}
```

`index = 2`, `right = 6`, 所以 `行{6}` 判断为 `true`, 执行 `行{7}` `quick` 函数再次调用自身 `quick(array, 2, 6, compareFn)`;

再次执行 `行{1}` --> `行{2}` --> `行{3}`;

再次调用 `partition(array, 2, 6, compareFn);` 执行流程和第 7 步一模一样;

`pivot: array[Math.floor((2 + 6) / 2)] = array[4] = 5`;

至此快速排序的整体执行流程算是完全分析完了, 后续的代码执行就省略所有的执行步骤, 只给出代码中的 `console` 输出:

```
pivot:  5
while leftIndex++ 3
while rightIndex-- 5
while rightIndex-- 4
leftIndex:  4
rightIndex:  3
array:  (7) [2, 3, 4, 5, 6, 7, 9]
```

9. 步骤 9

```
pivot:  4
while rightIndex-- 2
leftIndex:  3
rightIndex:  1
array:  (7) [2, 3, 4, 5, 6, 7, 9]
```

10. 步骤 10

```
array:  (7) [2, 3, 4, 5, 6, 7, 9]
pivot:  7
while leftIndex++ 5
while rightIndex-- 5
leftIndex:  6
rightIndex:  4
array:  (7) [2, 3, 4, 5, 6, 7, 9]
```

11. 步骤 11

```
pivot:  6
while rightIndex-- 2
leftIndex:  5
rightIndex:  3
array:  (7) [2, 3, 4, 5, 6, 7, 9]
```