

# 我接触过的前端数据结构与算法

我们已经讨论过了前端与计算机基础的很多话题，诸如[SQL](#)、[面向对象](#)、[多线程](#)，本篇将讨论数据结构与算法，以我接触过的一些例子做为说明。

## 1. 递归

递归就是自己调自己，递归在前端里面算是一种比较常用的算法。假设现在有一堆数据要处理，要实现上一次请求完成了，才能去调下一个请求。一个是可以用 Promise，就像《[前端与SQL](#)》这篇文章里面提到的。但是有时候并不想引入 Promise，能简单处理先简单处理。这个时候就可以用递归，如下代码所示：

```
var ids = [34112, 98325, 68125];
(function sendRequest(){
    var id = ids.shift();
    if(id){
        $.ajax({url: "/get", data: {id}}).always(function(

            console.log("finished");
            sendRequest();

        });
    }
});
```

```
    } else {  
        console.log("finished");  
    }  
}());
```

复制代码

上面代码定义了一个`sendRequest`的函数，在请求完成之后再调一下自己。每次调之前先取一个数据，如果数组已经为空，则说明处理完了。这样就用简单的方式实现了串行请求不堵塞的功能。

再来讲另外一个场景：DOM树。

由于DOM是一棵树，而树的定义本身就是用的递归定义，所以用递归的方法处理树，会非常地简单自然。例如用递归实现一个查DOM的功能  
`document.getElementById`。

```
function getElementById(node, id){  
    if(!node) return null;  
    if(node.id === id) return node;  
    for(var i = 0; i < node.childNodes.length; i++){  
        var found = getElementById(node.childNodes[i], id)  
        if(found) return found;  
    }  
    return null;  
}  
getElementById(document, "d-cal");复制代码
```

`document`是DOM树的根结点，一般从`document`开始往下找。在`for`循环里面先找`document`的所有子结点，对所有子结点递归查找他们的子结点，一层一层地往下查找。如果已经到了叶子结点还没有找到，则在第二行代码的判断里面返回`null`，返回之后`for`循环的`i`加1，继续下一个子结点。如果当前结点的`id`符合查找条件，则一层层地返回。所以这是一个深度优先的遍历，每次都先从根结点一直往下直到叶子结点，再从下往上返回。

最后在控制台验证一下，执行结果如下图所示：

```
> getElementById(document, "d-cal")  
< ▶ <button id="d-cal">...</button>
```

使用递归的优点是代码简单易懂，缺点是效率比不上非递归的实现。**Chrome**浏览器的查DOM是使用非递归实现。非递归要怎么实现呢？

如下代码：

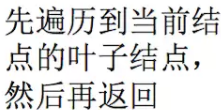
```
function getByElementId(node, id){  
  
    while(node){  
        if(node.id === id) return node;  
        node = nextElement(node);  
    }  
    return null;  
}复制代码
```

还是依次遍历所有的DOM结点，只是这一次改成一个`while`循环，函数`nextElement`负责找到下一个结点。所以关键在于这个`nextElement`如何非递归实现，如下代码所示：

```
function nextElement(node){
    if(node.children.length) {
        return node.children[0];
    }
    if(node.nextElementSibling){
        return node.nextElementSibling;
    }
    while(node.parentNode){
        if(node.parentNode.nextElementSibling) {
            return node.parentNode.nextElementSibling;
        }
        node = node.parentNode;
    }
    return null;
}复制代码
```

还是用深度遍历，先找当前结点的子结点，如果它有子结点，则下一个元素就是它的第一个子结点，否则判断它是否有相邻元素，如果有则返回它的下一个相邻元素。如果它既没有子结点，也没有下一个相邻元素，则要往上返回它的父结点的下一个相邻元素，相当于上面递归实现里面的for循环的i加1。

在控制台里面运行这段代码，同样也可以正确地输出结果。不管是非递归还是递归，它们都是深度优先遍历，这个过程如下图所示。



上面是单个选择器的查找，按id，按class等，多个选择器应该如何查找呢？

## 如实现一个document.querySelector:

<https://juejin.im/post/5958bac35188250d892f5c91>

首先把复杂选择器做一个解析，序列为以下格式：

```
var selectors = [  
  {relation: "descendant", matchType: "class", value: "copy"},  
  {relation: "child", matchType: "tag", value: "div"},  
  {relation: "subSelector", matchType: "class", value: "mls-"}]
```

从右往左，第一个selector是copyright-content，它是一个类选择器，所以它的matchType是class，它和第二个选择器是祖先和子孙关系，因此它的relation是descendant；同理第二个选择器的matchType是tag，而relation是child，表示是第三个选择器的直接子结点；第三个选择器也是class，但是它没有下一个选择器了，relation用subSelector表示。

matchType的作用就在于用来比较当前选择器是否match，如下代码所示：

```
function match(node, selector){  
  if(node === document) return false;  
  switch(selector.matchType){  
  
    case "class":  
      return node.className.trim().split(/ +/).indexOf(selector.value) !== -1;  
  
    case "tag":  
      return node.tagName.toLowerCase() === selector.value;  
  
    case "subSelector":  
      return match(node, selector.value) === true;  
  
    default:  
      return false;  
  }  
}
```

```
        default:
            throw "unknown selector match type";
    }
}复制代码
```

根据不同的matchType做不同的匹配。

在匹配的时候，从右往左，依次比较每个选择器是否match. 在比较下一个选择器的时候，需要找到相应的DOM结点，如果当前选择器是下一个选择器的子孙时，则需要比较当前选择器所有的祖先结点，一直往上直到document；而如果是直接子元素的关系，则比较它的父结点即可。所以需要有一个找到下一个目标结点的函数：

```
function nextTarget(node, selector){
    if(!node || node === document) return null;
    switch(selector.relation){
        case "descendant":
            return {node: node.parentNode, hasNext: true};
        case "child":
            return {node: node.parentNode, hasNext: false};
        case "sibling":
            return {node: node.previousSibling, hasNext: true};
        default:
            throw "unknown selector relation type";
    }
}复制代码
```

有了nextTarge和match这两个函数就可以开始遍历DOM，如下代码所示：

```
function querySelector(node, selectors){
  while(node){
    var currentNode = node;
    if(!match(node, selectors[0])){
      node = nextElement(currentNode);
      continue;
    }
    var next = null;
    for(var i = 0; i < selectors.length - 1; i++){
      var matchIt = false;
      do{
        next = nextTarget(node, selectors[i]);
        node = next.node;
        //没有相关节点，当前node匹配失败
        if(!node) break;
        if(match(node, selectors[i + 1])){
          matchIt = true;
          break;
        }
      }while(next.hasNext());
      if(!matchIt) break;
    }
    //已经把所有选择器匹配完，并且都是成功的，说明匹配成功
    if(matchIt && i === selectors.length - 1){
      return currentNode;
    }
    node = nextElement(currentNode);
  }
  return null;
}
```

遍历所有的DOM节点

如果匹配不上第一个选择器，则继续下一个节点

循环下一个选择器所有命中的节点，如果有一个match了，则继续下一个选择器

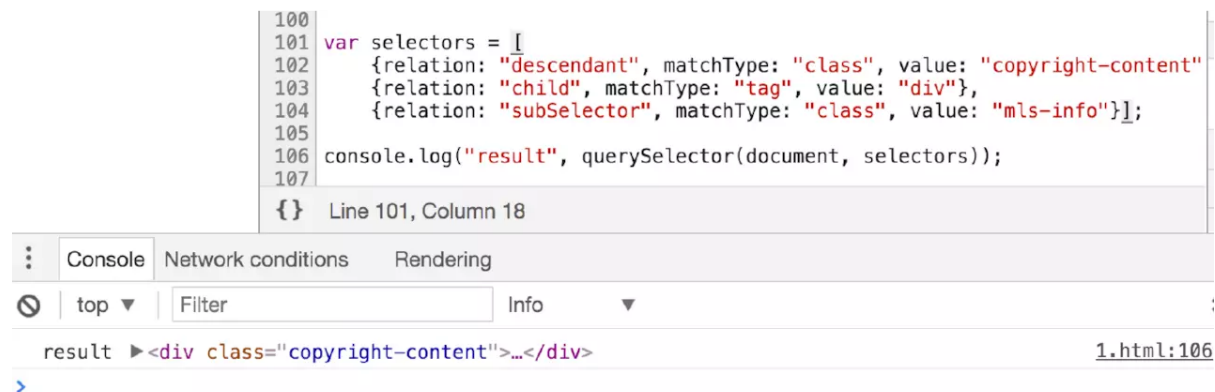
直到全部都匹配完了

最外层的while循环和简单选择器一样，都是要遍历所有DOM结点。对于每个结点，先判断第一个选择器是否match，如果不match的话，则继续下一个结点，如果不是标签选择器，对于绝大多数结点将会在这里判断不通过。如果第一个选择器match了，则根据第一个选择器的relation，找到下一个target，判断下一个targe是否match下一个selector，只要有一个target匹配上了，则退出里层的while循环，继续下一个选择器，如果所有的selector都能匹配上说明匹配成功。如果有一个selecotr的所有target都没有match，则说明匹配失败，退出selector的for循环，直接从头开始对下一个DOM结点进行匹配。



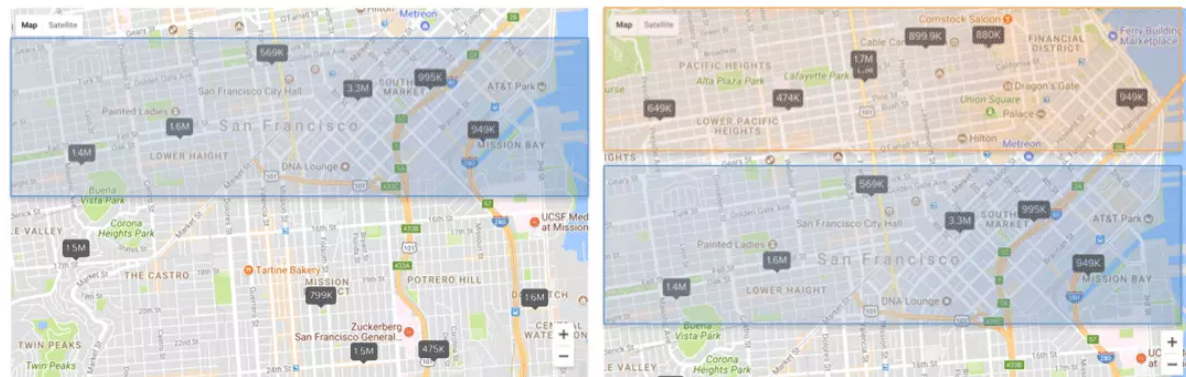
这样就实现了一个复杂选择器的查DOM。写这个的目的并不是要你自己写一个查DOM的函数拿去用，而是要明白查DOM的过程是怎么样的，可以怎么实现，浏览器又是怎么实现的。还有可以怎么遍历DOM树，当明白这个过程的时候，遇到类似的问题，就可以举一反三。

最后在浏览器上运行一下，如下图所示：



### 3. 重复值处理

现在有个问题，如下图所示：



当地图往下拖的时候要更新地图上的房源标签数据，上图绿框表示不变的标签，而黄框表示新加的房源。

后端每次都会把当前地图可见区域的房源返回给我，当用户拖动的时候需要知道哪些是原先已经有的房源，哪些是新加的。把新加的房源画上，而把超出区域的房源删掉，已有的房源保持不动。因此需要对比当前房源和新的结果哪些是重复的。因为如果不这样做的话，改成每次都是全部删掉再重新画，已有的房源标签就会闪一下。因此为了避免闪动做一个增量更新。

把这个问题抽象一下就变成：给两个数组，需要找出第一个数组里面的重复值和非重复值。即有一个数组保存上一次状态的房源，而另一个数组是当前状态的新房源数据。找到的重复值是需要保留，找到非重复值是要删掉的。

最直观的方法是使用双重循环。

## (1) 双重循环

如下代码所示：

```
var lastHouses = [];  
filterHouse: function(houses){  
    if(lastHouses === null){  
        lastHouses = houses;  
        return {  
            remainsHouses: [],  
            newHouses: houses  
        };  
    }  
    var remainsHouses = [],  
        newHouses = [];
```

```
    for(var i = 0; i < houses.length; i++){
```

```
var isNewHouse = true;
for(var j = 0; j < lastHouses.length; j++){
    if(houses[i].id === lastHouses[j].id){
        isNewHouse = false;
        remainsHouses.push(lastHouses[j]);
        break;
    }
}
if(isNewHouse){
    newHouses.push(houses[i]);
}
}
lastHouses = remainsHouses.concat(newHouses);
return {
    remainsHouses: remainsHouses,
    newHouses: newHouses
};
}
```

复制代码

上面代码有一个双重for循环，对新数据的每个元素，判断老数据里面是否已经有了，如果有的话则说明是重复值，如果老数据循环了一遍都没找到，则说明是新数据。由于用到了双重循环，所以这个算法的时间复杂度为 $O(N^2)$ ，对于百级的数据还好，对于千级的数据可能会有压力，因为最坏情况下要比较1000000次。

## (2) 使用Set

如下代码所示:

```
var lastHouses = new Set();
function filterHouse(houses){
    var remainsHouses = [],
        newHouses = [];
    for(var i = houses.length - 1; i >= 0; i--){
        if(lastHouses.has(houses[i].id)){
            remainsHouses.push(houses[i]);
        } else {
            newHouses.push(houses[i]);
        }
    }
    for(var i = 0; i < newHouses.length; i++){
        lastHouses.add(newHouses[i].id);
    }
    return {remainsHouses: remainsHouses,
            newHouses: newHouses};
}复制代码
```

老数据的存储lastHouses从数组改成set，但如果一开始就是数组呢，就像问题抽象里面说的给两个数组？那就用这个数组的数据初始化一个Set.

使用Set和使用Array的区别在于可以减少一重循环，调用Set.prototype.has的函数。Set一般是使用红黑树实现的，红黑树是一种平衡查找二叉树，它的查找时间复杂度为 $O(\log N)$ 。所以时间上进行了改进，从 $O(N)$ 变成 $O(\log N)$ ，而总体时间从 $O(N^2)$ 变成 $O(N \log N)$ 。实际上，Chrome V8的Set是用哈希实现的，它是一个哈希Set，查找时间复杂度为 $O(1)$ ，所以总体的时间复杂度是 $O(N)$ 。

不管是 $O(N\log N)$ 还是 $O(N)$ ，表面上看它们的时间要比 $O(N^2)$ 的少。但实际上需要注意的是它们前面还有一个系数。使用Set在后面更新lastHouses的时候也是需要时间的：

```
for(var i = 0; i < newHouses.length; i++){
    lastHouses.add(newHouses[i].id);
}
```

复制代码

如果Set是用树的实现，这段代码是时间复杂度为 $O(N\log N)$ ，所以总的时间为 $O(2N\log N)$ ，但是由于大O是不考虑系数的， $O(2N\log N)$ 还是等于 $O(N\log N)$ ，当数据量比较小的时候，这个系数会起到很大的作用，而数据量比较大的时候，指数级增长的 $O(N^2)$ 将会远远超过这个系数，哈希的实现也是同样道理。所以当数据量比较小时，如只有一两百可直接使用双重循环处理即可。

上面的代码有点冗长，我们可以用ES6的新特性改写一下，变得更加的简洁：

```
function filterHouse(houses){
    var remainsHouses = [],
        newHouses = [];
    houses.map(house => lastHouses.has(house.id) ? remains
        : newHouses.push(house));
    newHouses.map(house => lastHouses.add(house.id));
    return {remainsHouses, newHouses};
}
```

复制代码

代码从16行变成了8行，减少了一半。

### (3) 使用Map

使用Map也是类似的，代码如下所示：

```
var lastHouses = new Map();
function filterHouse(houses){
    var remainsHouses = [],
        newHouses = [];
    houses.map(house => lastHouses.has(house.id) ? remains
        : newHouses.push(house));
    newHouses.map(house => lastHouses.set(house.id, house))
    return {remainsHouses, newHouses};
}复制代码
```

哈希的查找复杂度为 $O(1)$ ，因此总的时间复杂度为 $O(N)$ ，Set/Map都是这样，代价是哈希的存储空间通常为数据大小的两倍

### (4) 时间比较

最后做下时间比较，为此得先造点数据，比较数据量分别为 $N = 100, 1000, 10000$ 的时间，有 $N/2$ 的id是重复的，另外一半的id是不一样的。用以下代码生成：

```
var N = 1000;
var lastHouses = new Array(N);
var newHouses = new Array(N);
var data = new Array(N);
```

```

for(var i = 0; i < N / 2; i++){
    var sameNumId = i;
    lastHouses[i] = newHouses[i] = {id: sameNumId};
}
for(; i < N; i++){
    lastHouses[i] = {id: N + i};
    newHouses[i] = {id: 2 * N + i};
}复制代码

```

然后将重复的数据随机分布，可用以下函数把一个数组的元素随机分布：

```

function randomIndex(arr){
    for(var i = 0; i < arr.length; i++){
        var swapIndex = parseInt(Math.random() * (arr.leng
        var tmp = arr[i];
        arr[i] = arr[swapIndex];
        arr[swapIndex] = tmp;
    }
}
randomIndex(lastHouses);
randomIndex(newHouses);复制代码

```

Set/Map的数据：

```

var lastHousesSet = new Set();
for(var i = 0; i < N; i++){

```

```
        lastHousesSet.add(lastHouses[i].id);
    }

    var lastHousesMap = new Map();
    for(var i = 0; i < N; i++){
        lastHousesMap.set(lastHouses[i].id, lastHouses[i]);
    }复制代码
```

分别重复100次，比较时间：

```
console.time("for time");
for(var i = 0; i < 100; i++){
    filterHouse(newHouses);
}
console.timeEnd("for time");
```

```
console.time("Set time");
for(var i = 0; i < 100; i++){
    filterHouseSet(newHouses);
}
console.timeEnd("Set time");
```

```
console.time("Map time");
for(var i = 0; i < 100; i++){
    filterHouseMap(newHouses);
}
console.timeEnd("Map time");复制代码
```



使用Chrome 59，当N = 100时，时间为：for < Set < Map，如下图所示，执行三次：

for time: 1.05615234375ms	for time: 1.021240234375ms	for time: 1.02392578125ms
Set time: 1.301025390625ms	Set time: 1.314697265625ms	Set time: 1.319091796875ms
Map time: 1.2138671875ms	Map time: 1.179931640625ms	Map time: 1.223876953125ms

当N = 1000时，时间为：Set = Map < for，如下图所示：

for time: 81.8388671875ms	for time: 84.0830078125ms	for time: 80.078125ms
Set time: 14.708984375ms	Set time: 14.0859375ms	Set time: 14.62890625ms
Map time: 14.941162109375ms	Map time: 15.011962890625ms	Map time: 14.260986328125ms

当N = 10000时，时间为Set = Map << for，如下图所示：

for time: 14067.343017578125ms	for time: 11048.724853515625ms	for time: 14571.531005859375ms
Set time: 150.600830078125ms	Set time: 161.98291015625ms	Set time: 174.516845703125ms
Map time: 151.918212890625ms	Map time: 158.296875ms	Map time: 169.633056640625ms

可以看出，Set和Map的时间基本一致，当数据量小时，for时间更少，但数据量多时Set和Map更有优势，因为指数级增长还是挺恐怖的。这样我们会有一个问题，究竟Set/Map是怎么实现的。

## 4. Set和Map的V8哈希实现

我们来研究一下Chrome V8对Set/Map的实现，源码是在[chrome/src/v8/src/js/collection.js](https://chromium.googlesource.com/v8/v8/src/js/collection.js)这个文件里面，由于Chrome一直在更新迭代，所以有可能以后Set/Map的实现会发生改变，我们来看一下现在是怎么实现的。

如下代码初始化一个Set：

```
var set = new Set();
```

```
var data = [3, 62, 38, 42, 14, 4, 14, 33, 56, 20, 21, 63,  
for(var i = 0; i < data.length; i++){  
    set.add(data[i]);  
}
```

复制代码

这个Set的数据结构到底是怎样的呢，是怎么进行哈希的呢？

哈希的一个关键的地方是哈希算法，即对一堆数或者字符串做哈希运算得到它们的随机值，V8的数字哈希算法是这样的：

```
function ComputeIntegerHash(key, seed) {  
    var hash = key;  
    hash = hash ^ seed;  
    hash = ~hash + (hash << 15);  
    hash = hash ^ (hash >>> 12);  
    hash = hash + (hash << 2);  
    hash = hash ^ (hash >>> 4);  
    hash = (hash * 2057) | 0;  
    hash = hash ^ (hash >>> 16);  
    return hash & 0x3fffffff;  
}
```

复制代码

把数字进行各种位运算，得到一个比较随机的数，然后对这个数对行散射，如下所示：

```
var capacity = 64;
var indexes = [];
for(var i = 0; i < data.length; i++){
    indexes.push(ComputeIntegerHash(data[i], seed)
                  & (capacity - 1));
}
console.log(indexes)复制代码
```

散射的目的是得到这个数放在数组的哪个index。

由于有20个数，容量capacity从16开始增长，每次扩大一倍，到64的时候，可以保证 $\text{capacity} > \text{size} * 2$ ，因为只有容量是实际存储大小的两倍时，散射结果重复值才能比较低。

计算结果如下：

(index)	data	hash	index
0	9	966578110	3
1	33	1040156529	62
2	68	555376181	38
3	57	960921035	42
4	56	517763097	14
5	15	984906997	4
6	72	216206785	14
7	91	846706203	33
8	31	268270873	56
9	52	337660910	20
10	32	449031721	21
11	97	846706203	63
12	0	858319754	49
13	48	984906997	41
14	39	178506403	10
15	46	137068084	14
16	49	846706203	24
17	12	1057677827	59
18	78	555376181	49
19	10	852843840	29

可以看到散射的结果还是比较均匀的，但是仍然会有重复值，如14重复了3次。

然后进行查找，例如现在要查找key = 56是否存在这个Set里面，先把56进行哈希，然后散射，按照存放的时候同样的过程：

```
function SetHas(key){  
    var index = ComputeIntegerHash(56, seed) & this.capacity;  
  
    return setArray[index] !== null  
        && setArray[index] === key;
```

```
}
```

复制代码

上面是哈希存储结构的一个典型实现，但是Chrome的V8的Set/Map并不是这样实现的，略有不同。

哈希算法是一样的，但是散射的时候用来去掉高位的并不是用capacity，而是用capacity的一半，叫做buckets的数量，用下面的data做说明：

```
var data = [9, 33, 68, 57];
```

复制代码

由于初始化的buckets = 2，计算的结果如下：

(index)	data	hash	bucket
0	9	966578110	0
1	33	1040156529	1
2	68	555376181	1
3	57	960921035	1

由于buckets很小，所以散射值有很多重复的，4个数里面1重复了3次。

现在一个个的插入数据，观察Set数据结构的变化。

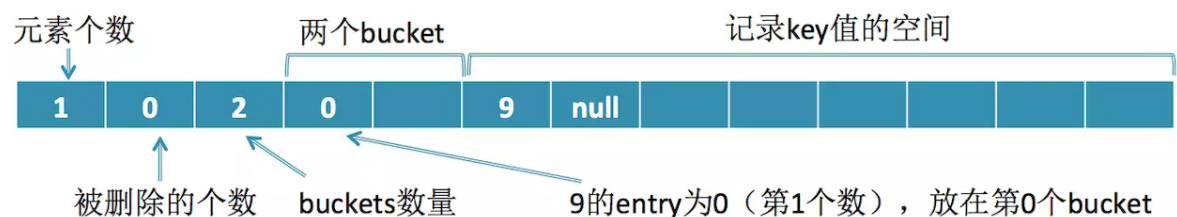
## (1) 插入过程

### a) 插入9

如下图所示，Set的存储结构分成三部分，第一部分有3个元素，分别表示有效元素个数、被删除的个数、buckets的数量，前两个数相加就表示总的元素个数，

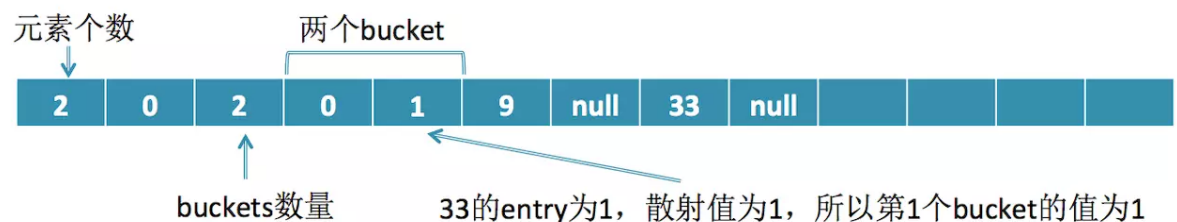
插入9之后，元素个数加1变成1，初始化的buckets数量为2. 第二部分对应

buckets, buckets[0]表示第1个bucket所存放的原始数据的index, 源码里面叫做entry, 9在data这个数组里面的index为0, 所以它在bucket的存放值为0, 并且bucket的散射值为0, 所以bucket[0] = 0. 第三部分是记录key值的空间, 9的entry为0, 所以它放在了 $3 + \text{buckets.length} + \text{entry} * 2 = 5$ 的位置, 每个key值都有两个元素空间, 第一个存放key值, 第二个是keyChain, 它的作用下面将提到。



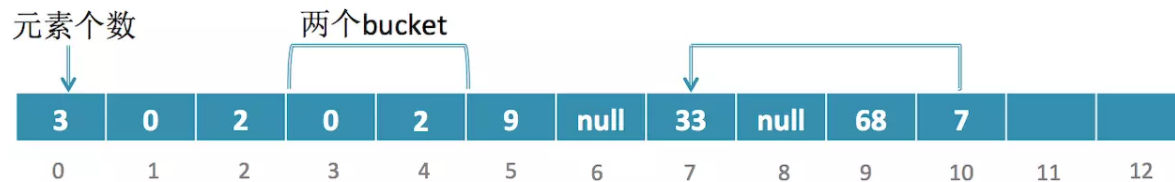
## b) 插入33

现在要插入33, 由于33的bucket = 1, entry = 1, 所以插入后变成这样:



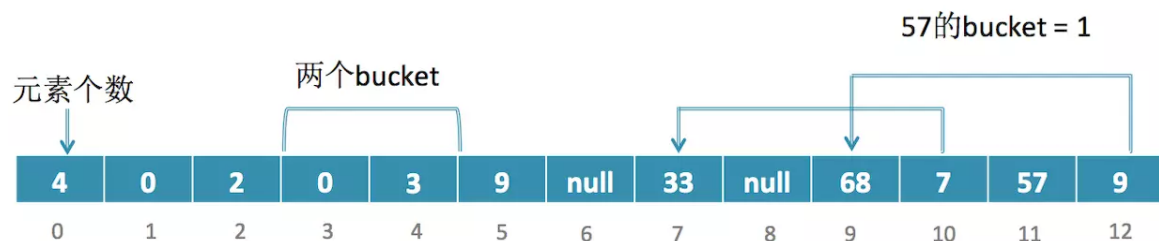
## c) 插入68

68的bucket值也为1, 和33重复了, 因为entry = buckets[1] = 1, 不为空, 说明之前已经存过了, entry为1指向的数组的位置为 $3 + \text{buckets.length} + \text{entry} * 2 = 7$ , 也就是说之前的那个数是放在数组7的位置, 所以68的相邻元素存放值keyChain为7, 同时bucket[1]变成68的entry为2, 如下图所示:



#### d) 插入57

插入57也是同样的道理，57的bucket值为1，而bucket[1] = 2，因此57的相邻元素存放 $3 + 2 + 2 * 2 = 9$ ，指向9的位置，如下图所示：



#### (2) 查找

现在要查找33这个数，通过同样的哈希散射，得到33的bucket = 1，bucket[1] = 3，3指向的index位置为11，但是11放的是57，不是要找的33，于是查看相邻的元素为9，非空，可继续查找，位置9存放的是68，同样不等于33，而相邻的index = 10指向位置7，而7存放的就是33了，通过比较key值相等，所以这个

Set里面有33这个数。

```
entry = bucket[1] = 3
index = 3 + bucketCount + 2 * entry = 11
candidateKey = array[11] = 57 ← 不等于要查找的33

chainEntry[11 + 1] = 9 ← 9不是一个null，可继续往下找
array[9] = 68 同样不等于33
array[9 + 1] = 7
array[7] = 33 ← 最后找到
```

这里的数据总共是4个数，但是需要比较的次数比较多，key值就比较了3次，key值的相邻keyChain值比较了2次，总共是5次，比直接来个for循环还要多。所以数据量比较小时，使用哈希存储速度反而更慢，但是当数据量偏大时优势会比较明显。

### (3) 扩容

再继续插入第5个数的时候，发现容量不够了，需要继续扩容，会把容量提升为2倍，bucket数量变成4，再把所有元素再重新进行散射。

Set的散射容量即bucket的值是实际元素的一半，会有大量的散射冲突，但是它的存储空间会比较小。假设元素个数为N，需要用来存储的数组空间为： $3 + N / 2 + 2 * N$ ，所以占用的空间还是挺大的，它用空间换时间。

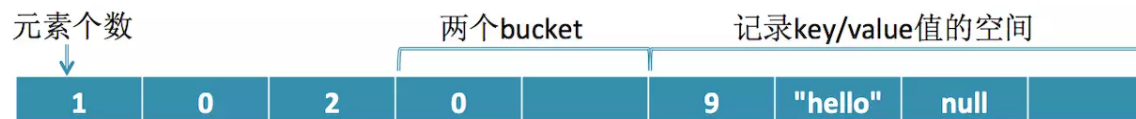
### (4) Map的实现

和Set基本一致，不同的地方是，map多了存储value的地方，如下代码：

```
var map = new Map();
map.set(9, "hello");复制代码
```



生成的数据结构为：



当然它不是直接存的字符串“hello”，而是存放hello的指针地址，指向实际存放hello的内存位置。

## (5) 和JS Object的比较

JSObject主要也是采用哈希存储，具体我在《[从Chrome源码看JS Object的实现](#)》这篇文件章里面已经讨论过。

和JS Map不一样的地方是，JSObject的容量是元素个数的两倍，就是上面说的哈希的典型实现。存储结构也不一样，有一个专门存放key和一个存放value的数组，如果能找到key，则拿到这个key的index去另外一个数组取出value值。当发生散列值冲突时，根据当前的index，直接计算下一个查找位置：

```
inline static uint32_t FirstProbe(uint32_t hash, uint32_t
    return hash & (size - 1);
}
```

```
inline static uint32_t NextProbe(
    uint32_t last, uint32_t number, uint32_t size) {
    return (last + number) & (size - 1);
}复制代码
```

同样地，查找的时候在下一个位置也是需要比较key值是否相等。

上面讨论的都是数字的哈希，字符串如何做哈希计算呢？

## (6) 字符串的哈希计算

如下所示，依次对字符串的每个字符的unicode编码做处理：

```
uint32_t AddCharacterCore(uint32_t running_hash, uint16_t
    running_hash += c;
    running_hash += (running_hash << 10);
    running_hash ^= (running_hash >> 6);
    return running_hash;
}
```

```
uint32_t running_hash = seed;
char *key = "hello";
for(int i = 0; i < strlen(key); i++){
    running_hash = AddCharacterCore(running_hash, key[i]);
}复制代码
```

接着讨论一个经典话题

## 5. 数组去重

如下，给一个数组，去掉里面的重复值：

```
var a = [3, 62, 3, 38, 20, 42, 14, 5, 38, 29, 42];
```

输出

```
[3, 62, 38, 20, 42, 14, 5, 29];
```

复制代码

## 方法1: 使用Set + Array

如下代码所示:

```
function uniqueArray(arr){  
    return Array.from(new Set(arr));  
}
```

复制代码

在控制台上运行:

```
var a = [3, 62, 3, 38, 20, 42, 14, 5, 38, 29, 42];  
function uniqueArray(arr){  
    return Array.from(new Set(arr));  
}  
uniqueArray(a)  
► (8) [3, 62, 38, 20, 42, 14, 5, 29]
```

优点: 代码简洁, 速度快, 时间复杂度为 $O(N)$

缺点: 需要一个额外的Set和Array的存储空间, 空间复杂度为 $O(N)$

## 方法2: 使用splice

如下代码所示：

```
function uniqueArray(arr){  
    for(var i = 0; i < arr.length - 1; i++){  
        for(var j = i + 1; j < arr.length; j++){  
            if(arr[j] === arr[i]){  
                arr.splice(j--, 1);  
            }  
        }  
    }  
    return arr;  
}  
复制代码
```

优点：不需要使用额外的存储空间，空间复杂度为 $O(1)$

缺点：需要频繁的内存移动，双重循环，时间复杂度为 $O(N^2)$

注意splice删除元素的过程是这样的，这个我在《[从Chrome源码看JS Array的实现](#)》已做过详细讨论：

它是用的内存移动，并不是写个for循环一个个复制。内存移动的速度还是很快的，最快1s可以达到30GB，如[下图所示](#)：

### 方法3：只用Array

如下代码所示：

```
function uniqueArray(arr){  
    var retArray = [];  
    for(var i = 0; i < arr.length; i++){  
        if(retArray.indexOf(arr[i]) < 0){  
            retArray.push(arr[i]);  
        }  
    }  
}
```

```
    }  
    return retArray;  
}复制代码
```

时间复杂度为 $O(N^2)$ ，空间复杂度为 $O(N)$

## 方法4：使用Object + Array

下面代码是[goog.array](#)的去重实现：

和方法三的区别在于，它不再是使用`Array.indexOf`判断是否已存在，而是使用`Object[key]`进行哈希查找，所以它的时间复杂度为 $O(N)$ ，空间复杂为 $O(N)$ 。

最后做一个执行时间比较，对 $N = 100/1000/10000$ ，分别重复1000次，得到下面的表格：

`Object + Array`最省时间，`splice`的方式最耗时（它比较省空间），`Set + Array`的简洁方式在数据量大的时候时间将明显少于需要 $O(N^2)$ 的`Array`，同样是 $O(N^2)$ 的`splice`和`Array`，`Array`的时间要小于经常内存移动操作的`splice`。



实际编码过程中1、2、4都是可以可取的

方法1 一行代码就可以搞定

方法2 可以用来添加一个Array.prototype.unique的函数

方法4 适用于数据量偏大的情况

上面已经讨论了哈希的数据结构，再来讨论下栈和堆

## 6. 栈和堆

### (1) 数据结构的栈

栈的特点是先进后出，只有push和pop两个函数可以操作栈，分别进行压栈和弹栈，还有top函数查看栈顶元素。栈的一个典型应用是做开闭符号的处理，如构建DOM。有以下html：

```
hello, world
goodbye, world
```

复制代码

将会构建这么一个DOM:

上图省略了document结点，并且这里我们只关心DOM父子结点关系，省略掉兄弟节点关系。

首先把html序列化成一个个的标签，如下所示：

```
1 html ( 2 head ( 3 head ) 4 body ( 5 div ( 6 text 7 div ) 8 p ( 9 text 10 p ) 11  
body ) 12 html)
```

其中左括号表示开标签，右括号表示闭标签。

如下图所示，处理html和head标签时，它们都是开标签，所以把它们都压到栈里面去，并实例一个HTMLHtmlElement和HTMLHeadElement对象。处理head标签时，由于栈顶元素是html，所以head的父元素就是html。

处理剩余其它元素如下图所示：

遇到第三个标签是head的闭标签，于是进行弹栈，把head标签弹出来，栈顶元素变成了html，所以在遇到第一个标签body的时候，html元素就是body标签的父结点。其它节点类似处理。

上面的过程，我在《[从Chrome源码看浏览器如何构建DOM树](#)》已经做过讨论，这里用图表示意，可能会更加直观。

## (2) 内存栈

函数执行的时候会把局部变量压到一个栈里面，如下函数：

```
function foo(){  
    var a = 5,  
        b = 6,  
        c = a + b;  
}  
foo();复制代码
```

a, b, c三个变量在内存栈的结构如下图所示：

先遇到a把a压到栈里面，然后是b和c，对函数里面的局部变量不断地压栈，内存向低位增长。栈空间大小8MB（可使用8MB / (8B \* 10) = 80万次就会发生栈溢出stackoverflow

这个在《[WebAssembly与程序编译](#)》这篇文章里面做过讨论。

### (3) 堆

数据结构里的堆通常是指用数组表示的二叉树，如大堆排序和小堆排序。内存里的堆是指存放new出来动态创建变量的地方，和栈相对，如下图所示：

讨论完了栈和堆，再分析一个比较实用的技巧。

## 6. 节流

节流是前端经常会遇到的一个问题，就是不想让resize/mousemove/scroll等事件触发得太快，例如说最快每100ms执行一次回调就可以了。如下代码不进行节流，直接兼听resize事件：

```
$(window).on("resize", adjustSlider);复制代码
```

由于adjustSlider是一个非常耗时的操作，我并不想让它执行得那么快，最多500ms执行一次就好了。那应该怎么做呢？如下图所示，借助setTimeout和一个tId的标志位：

最后再讨论下图像和图形处理相关的。

## 7. 图像处理

假设要在前端做一个滤镜，如用户选择了本地的图片之后，点击某个按钮就可以把图片置成灰色的：



效果如下:

一个方法是使用CSS的`filter`属性，它支持把图片置成灰图的：

```
{  
    filter: grayscale(100%);  
}
```

复制代码

由于需要把真实的图片数据传给后端，因此需要对图片数据做处理。我们可以用`canvas`获取图片的数据，如下代码所示：

```
<canvas id="my-canvas"></canvas>
```

复制代码

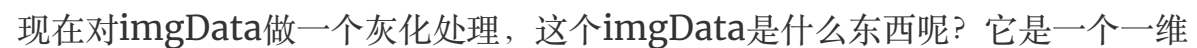

JS处理如下:

```
var img = new Image();
img.src = "/test.jpg";
img.onload = function(){

    ctx.drawImage(this, 10, 10);
}

function blackWhite() {
    var imgData = ctx.getImageData(10, 10, 31, 30);
    ctx.putImageData(imgData, 50, 10);
    console.log(imgData, imgData.data);
}复制代码
```

这个的效果是把某张图片原封不动地再画一个，如下图所示：

现在对做一个灰化处理，这个是什么呢？它是一个一维数组，存放了从左到右，从上到下每个像素点的**rgba**值，如下图所示：

这张图片尺寸为 $31 * 30$ ，所以数组的大小为 $31 * 30 * 4 = 3720$ ，并且由于这张图片没有透明通道，所以a的值都为255。

常用的灰化算法有以下两种：

(1) 平均值

$$\text{Gray} = (\text{Red} + \text{Green} + \text{Blue}) / 3$$

(2) 按人眼对三原色的感知度：绿 > 红 > 蓝

$$\text{Gray} = (\text{Red} * 0.3 + \text{Green} * 0.59 + \text{Blue} * 0.11)$$

第二种方法更符合客观实际，我们采用第二种方法，如下代码所示：

```
function blackWhite() {  
    var imgData = ctx.getImageData(10, 10, 31, 30);  
    var data = imgData.data;  
    var length = data.length;  
    for(var i = 0; i < length; i += 4){  
        var grey = 0.3 * data[i] + 0.59 * data[i + 1] + 0.11 * data[i + 2];  
        data[i] = data[i + 1] = data[i + 2] = grey;  
    }  
    ctx.putImageData(imgData, 50, 10);  
}复制代码
```

执行的效果如下图所示：

其它的滤镜效果，如模糊、锐化、去斑等，读者有兴趣可继续查找资料。

还有一种是图形算法

## 8. 图形算法

如下需要计算两个多边形的交点：

这个就涉及到图形算法，可以认为图形算法是对矢量图形的处理，和图像处理是对全部的rgba值处理相对。这个算法也多种多样，其中一个可参考《[A new algorithm for computing Boolean operations on polygons](#)》

综合以上，本篇讨论了几个话题：

1. 递归和查DOM
2. Set/Map的实现
3. 数组去重的几种方法比较
4. 栈和堆



## 5. 节流

## 6. 图像处理

本篇从前端的角度对一些算法做一些分析和总结，只列了一些我认为比较重要，其它的还有很多没有提及。算法和数据结构是一个永恒的话题，它的目的是用最小的时间和最小的空间解决问题。但是有时候不用太拘泥于一定要最优的答案，能够合适地解决问题就是好方法，而且对于不同的应用场景可能要采取不同的策略。反之，如果你的代码里面动不动就是三四重循环，还有嵌套了很多if-else，你可能要考虑下采用合适的的数据结构和算法去优化你的代码。

---

Viewed using [Just Read](#)