> 本文档为记录bilibili上关于前端算法和LeetCode教程的学习笔记

栈

LeetCode20：有效的括号

```
 s = '{}()[]()'

        var isValid = function (s) {
            var stack = []
            for (let i = 0; i < s.length; i++) {
                let start = s[i]
                if (start == "(" || start == "{" || start == "[") {
                    stack.push(s[i])
                } else {
                    const end = stack[stack.length - 1]

                    if (start == ")" && end == "(" || start == "]" && end == "["
|| start == "}" && end == "{") {
                        stack.pop()
                    } else {
                        return false
                    }
                }
            }

            return stack.length == 0
        };
```

本题利用的是栈的后进先出的原理

LeetCode1047：删除字符串中所有的相邻重复项

```
let s='abbcac'
var removeDuplicates=function(s){
    let stack=[]
    for(v of s ){
        let prev=stack.pop()
        if(prev!=v){
            stack.push(prev)
            stack.push(v)
        }
    }

    return stack.join('')
}
```

LeetCode71:简化路径

```
var simplifyPath(path){
    let stack=[]
    let res=''
    let arr=path.split('/')

    arr.forEach(val => {
        if(val&& val=='..'){
            stack.pop()
        }else if(val && val!='.'){
            stack.push(val)
        }
    });

    arr.length? res='/'+stack.join('/') : res='/'
    return res
}
```

队列

先进先出

js执行流程

1. 主线程读取js代码，此时为同步环境，形成对应的堆和执行栈
2. 主线程如果遇到异步任务会推给异步进程进行处理
3. 异步进程处理完毕，将对应的异步任务推入任务队列，任务队列分为宏任务和微任务
4. 主线程查询任务队列，执行微任务，将其按照顺序执行，全部执行完毕
5. 主线程查询任务队列，执行宏任务，取得第一个宏任务，执行完毕
6. 重复以上4，5步骤

LeetCode933： 最近的请求次数

```
var RecentCounter = function() {
this.stack=[]
};


RecentCounter.prototype.ping = function(t) {
    this.stack.push(t)
while( this.stack[0]<t-3000){
    this.stack.shift()
}
return this.stack.length
};
```

链表

链表是一个多个元素存储的列表

链表优点类似于数组，不过链表中的元素在内存中不是顺序存储的，而是通过next指针联系在一起的

js中的原型链原理就是链表结构

链表与数组的区别：

- 数组是有序存储的，在中间某个位置删除或者添加某个元素，其他元素要跟着动
- 链表中的元素在内存中不是顺序存储的，而是通过next指针联系在一起的

链表分类：

- 单向
- 双向
- 环形

instanceof 原理

利用原型链的原理

```
let myInstanceof=(target,obj)=>{
while(target){
    if(target==obj.prototype){
        return true
    }

    target=target.__proto__
}

return false
}
```

LeetCode141:环形链表

```
var hasCycle(head){
    let f=head, s=head
    while(f!==null && f.next!==null){
        s=s.next;
        f=f.next.next;
        if( s=f) return true
    }
    return false
}
```

LeetCode237:删除链表中的节点

```
var deleteNode=function (node){
    node.val=node.next.val
    node.next=node.next.next
}
```

LeetCode83：删除排序链中的重复元素

如果当前的节点有next，将当前节点的val值与next节点的val值进行对比，如果两个节点的val相同的话，就删除第一个节点，如果不同，就继续往下对比

```
var deleteDuplicates=function(head) {
    if(!head){
        return head
    }


let cur=head
while(cur.next){
    if(cur.val==cur.next.val){
        cur.next=cur.next.next
    }else{
        cur=cur.next
    }
}

  return head

}
```

LeetCode206：翻转列表

1-> 2 -> 3 -> 4 -> 5 5 -> 4 -> 3 -> 2 -> 1

```
var reverseList=function(head){
    let prev= null
    let curr=head
 while (curr){
    let next= curr.next
    curr.next=prev
    prev=curr;
    curr=next
 }
```

```
 return prev
}
```

数组和链表的区别

1. 元素之间的联系
    - 数组通过下标联系在一起
    - 链表通过next指针联系在一起
2. 数据插入
    - 数组如果在中间插入新的元素，其他元素会重新计算
    - 链表不会重新计算
3. 查找
    - 数组：通过下标进行查找
    - 链表： 每次查找都需要从头开始找

字典 & 哈希表

字典是键值对存储，有点类似于js的对象

js存在一定的问题：js的键（key）都是字符串类型，或者会转换为字符串类型

字典是以map来表示的，map的键不会转换类型

区别：

1. 寻找value
    - 字典如果要找key对应的value需要遍历key，
    - 那么想要省去遍历的过程，需要用哈希表来表示
2. 排列顺序
    - 字典是根据添加的顺序进行排列的
    - 哈希表不是添加的顺序进行排列的

LeetCode1:两数之和

```
var twoSum=function(nums,target){
    let map=new Map()
    for (let i = 0 ;i<nums.length;i++){

        let num=target-nums[i]

        if(map.has(num)){
            return [map.get(num),i]
        }else{
            map.set(nums[i],i)
        }
    }
}
```

LeetCode217:重复元素

数组中有重复元素返回true,没有返回false

```
var containsDuplicate=function(nums){
    let set=new Set()
    for (let num of nums){
        if (set.has(num)){
            return true
        }else{
            set.add(num)
        }
    }

    return false
}
```

LeetCode349:两个数组的交集

```
var intersection = function (num1,nums2){
    let set=new Set(nums2)

    return [...new Set(num1)].filter(item=>set.has(item))
}
```

字符串中出现次数最多的字符，并统计字数

```
function MostString(s){
    let maxNum=0
    let maxStr-''
    let map=new Map()

    for(let item of s){
map.set(item,(map.get(item) || 0)+1)
    }


    for (let [key,val] of map){
        if(val>maxNum){
            maxStr=key
            maxNum=val
        }
    }

    return [maxStr,maxNum]
}
```

LeetCode1207:独一无二的出现次数

```
var uniqueOccurrences = function(arr) {
    let map=new Map()

    for (let item of arr ){
        if (map.has(item)){
            map.set(item,map.get(item)+1)
        }else{
            map.set(item,1)
        }

    }



    let set =new Set()

    for (let [key,value] of map){
        set.add(value)
    }


    return set.size==map.size
}
```

LeetCode3:无重复字符的最长子串

```
// 滑动窗口思想
var lengthOfLongestSubstring=function(s){

    let map=new Map()

    let left=0 //左指针

    let num=0 //记录的最长无重复子串的数量

    for (let i =0 i<s.length; i++){
        if(map.has(s[i]) && map.get(s[i]) >=left){
            left=map.get(s[i])+1
        }else{

            num=Math.max(num,i-l+1)
            map.set(s[i],i)
        }
    }
```

```
    return num
}
```

树

树是一种分层数据的抽象模型

深度优先遍历（搜索）

从根出发，尽可能深的搜素树的节点

技巧：

1. 访问根节点
2. 对于根节点的children挨个进行深度优点遍历

广度优先遍历（搜索）

从根出发，优先访问离根节点最近的节点

技巧：

1. 新建一个队列，把根节点入队
2. 把队头出队
3. 把对头的children挨个入队
4. 重复2和3步，直到队列为空

二叉树

LeetCode144：二叉树的前序遍历

前序遍历：根左右

```
// 递归的形式
var preorderTraversal=function (root){

    let arr=[]

    let fun=(node)=>{
        if(node){
            // 把根节点放进去
            arr.push(node.val)

            // 遍历左子树
            fun(node.left)

            // 遍历右子树
            fun(node.right)
```

```
        }
    }


    fun(root)

    return arr

}



// 栈的形式
var preorderTraversal=function (root){

    if(!root) return []

    let arr=[]

    // 根节点入栈
    let stack=[root]

    while(stack.length){
        // 出栈
        let o =stack.pop()

        arr.push(o.val)

        o.right&& stack.push(o.right)
        o.left&& stack.push(o.left)


    }

    return arr

}
```

LeetCode94:二叉树的中序遍历

左根右

递归版本

```


var inorderTraversal=function(root){

    let arr=[]

    let fun=(node)=>{
```

```
        if(!node) return []

        fun(node.left)
        arr.push(node.val)
        fun(node.right)

    }

fun(root)
return arr
}
```

非递归版本

```
var inorderTraversal=function (root) {

    let arr=[]
    let stack=[]
    let o=root

    while(stack.length || o){
        wihle(o){
            stack.push(o)
            o=o.left
        }

        const n=stack.pop()
        arr.push(n.val)
        o=o.right
    }

    return arr
  }
```

后序遍历

左右根

递归版：

```
var postorderTraversal=function(root){
    let arr=[]

    let fun=(node)=>{
        if(node){
            fun(node.left)
            fun(node.right)
            arr.push(node.val)
        }
```

```
    }


    return arr

}
```

非递归版：

```
var postorderTraversal=function(root){
    if( ! root) return []
    let arr=[]
    let stack=[root]

    while(stack.length){
        const o=stack.pop()
        arr.unshift(o.val)
        o.left&& atack.push(o.left)
        o.right && stack.push(o.right)
    }
    return arr
}
```

LeetCode111:二叉树的最小深度

```
var minDepth=function(root){
    if(!root) return 0

    const stack=[[root,1]]
    while(stack.length){
        const [o,n]=stack.shift()

        if(!o.left && ! o.right){
            return n
        }

        if(o.left) stack.push([o.left,n+1])
        if(o.right) stack.push([o.right,n+1])
    }
}
```

LeetCode104:二叉树的最大深度

```
var maxDepth=function(root){
    if(!root) return 0
```

```
    const stack=[root]
    let num=0

    while(stack.length){
        let len=stack.length

        num++

        while(len--){
            const o=stack.shift()

            o.left && stack.push(o.left)
            o.right && stack.push(o.right)
        }
    }


    return num
}
```

LeetCode226:翻转二叉树

```
var inertTree=function(root){

    if(!root) reutrn

    let temp= root.left;
    root.left=root.right;
    root.right=temp;

    inertTree(root.left)
    inertTree(root.right)

    return root

}
```

LeetCode100:相同的树

```
var isSameTree=function (p,q){
    if(p===null && q=== null) return true
    if(p===null || q=== null) return false
    if(p.val!== q.val) return false

  return   isSameTree(p.left,q.left) && isSameTree(p.right,q.right)
}
```

堆

堆都能用树来表示，并且一般树的实现都是利用链表

而二叉堆是一种特殊的堆，它用完全二叉树表示，却可以利用数组实现

平时使用最多的是二叉堆，他可以用完全二叉树表示，二叉堆易于储存，并且便于索引

在堆实现时，需要注意的几点：

- 因为是数组所以父子节点的关系就不需要特殊的结构去维护了，索引之间通过计算就可以得到，省去了很多麻烦，如果是链表结构，就会复杂很多
- 完全二叉树要求叶子结点从左往右填满，才能开始填充下一层，这就保证了不需要对数组整体进行大片的移动，这也是随机存储结果（数组）的短板；删除一个元素之后，整体前移是比较费时的。这个特性也导致堆在删除元素的时候，要把最后一个叶子结点补充到树根结点的缘故

二叉树像树的样子我可以理解，但将他们安排在数组里的化，通过当前下标怎么就能找到父节点和子节点的呢?

- 寻找左子树：2*index+1
- 寻找右子树：2*index+2
- 寻找父节点：(index-1)/2

最小堆

```
class MinHead{
    constructor(){
        this.heap=[]
    }

    // 添加元素
    insert(value){
        this.heap.push(value)

    }
}
```

LeetCode215：数组中的第k个最大元素

```

```

排序算法

冒泡排序

```

```

```
function bubbleSort(arr){
    for (let i=0;i<arr.length-1;i++){
        for (let j=0;j<arr.length-1-i;j++){
            if(arr[j]>arr[j+1]){
                [arr[j],arr[j+1]]=[arr[j+1],arr[j]]
            }
        }
    }

    return arr
}
```

选择排序

```
function chooseSort(arr){

    for (let i=1;i<arr.length;i++)a[
        for (let j=i;j>0;j--){
            if(arr[j]<arr[j-1]){
                [arr[j],arr[j-1]]=[arr[j-1],arr[j]]
            }
        }

    ]

    return arr

}
```