手写实现

二叉树遍历

- 前序遍历：先遍历根结点，然后左子树，再右子树
- 中序遍历：先遍历左子树，然后根结点，再右子树
- 后续遍历：先遍历左子树，然后右子树，再根结点

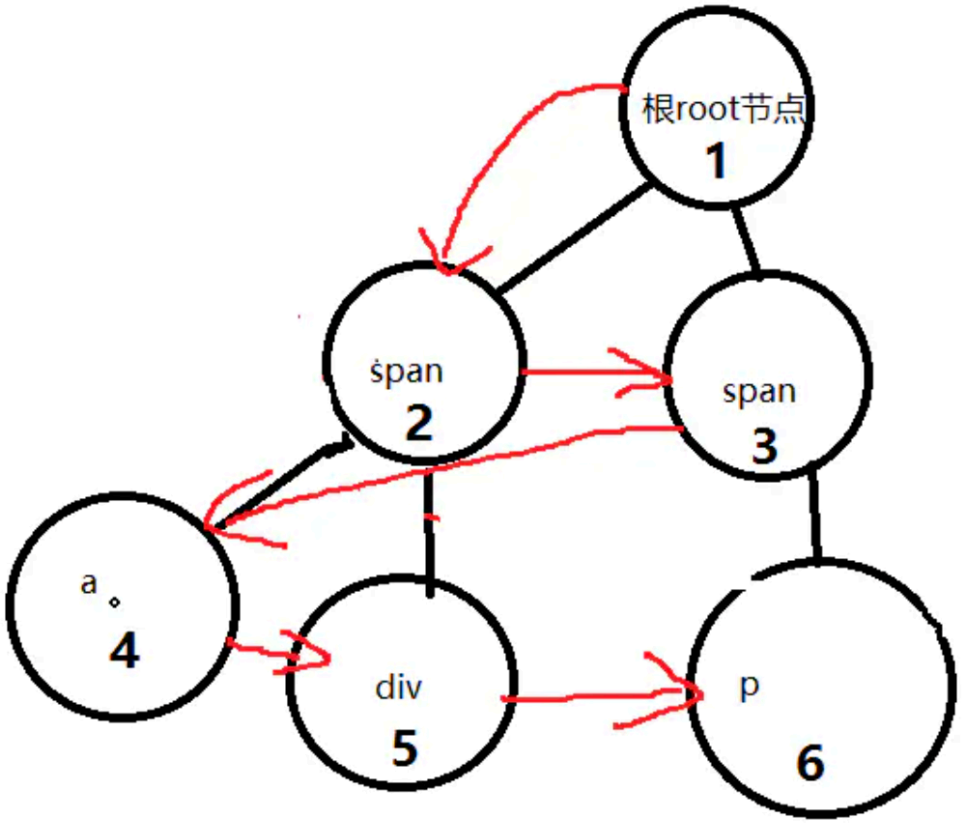
```
function CreatePreOrderTree(arr) {
  arr.sort((a,b) => a - b);
  function createNode(root, left, rgiht) {
    this.root = root;
    this.left = left ? left : null;
    this.right = right ? right : null;
  }

  function preOrderTree(node, val) {
    if(!node) {
      preOrderTree(new createNode(null, null, null), arr.pop());
    }

    if(val > node.root) {

    }
  }

  return preOrderTree(null, 0);
}
```

```
function TreeCode() {
    let BiTree = function (ele) {
        this.data = ele;
        this.lChild = null;
        this.rChild = null;
    }

    this.createTree = function () {
        let biTree = new BiTree('A');
        biTree.lChild = new BiTree('B');
        biTree.rChild = new BiTree('C');
        biTree.lChild.lChild = new BiTree('D');
        biTree.lChild.lChild.lChild = new BiTree('G');
        biTree.lChild.lChild.rChild = new BiTree('H');
        biTree.rChild.lChild = new BiTree('E');
        biTree.rChild.rChild = new BiTree('F');
        biTree.rChild.lChild.rChild = new BiTree('I');
        return biTree;
```

```
    }
}

//前序遍历
function ProOrderTraverse(biTree) {
    if (biTree == null) return;
    console.log(biTree.data);
    ProOrderTraverse(biTree.lChild);
    ProOrderTraverse(biTree.rChild);
}

//中序遍历
function InOrderTraverse(biTree) {
    if (biTree == null) return;
    InOrderTraverse(biTree.lChild);
    console.log(biTree.data);
    InOrderTraverse(biTree.rChild);
}

//后续遍历
function PostOrderTraverse(biTree) {
    if (biTree == null) return;
    PostOrderTraverse(biTree.lChild);
    PostOrderTraverse(biTree.rChild);
    console.log(biTree.data);
}

let myTree = new TreeCode();
console.log(myTree.createTree());
console.log('前序遍历')
ProOrderTraverse(myTree.createTree());
console.log('中序遍历')
InOrderTraverse(myTree.createTree());
console.log('后续遍历')
PostOrderTraverse(myTree.createTree());
```

广度优先遍历二叉树

广度优先遍历二叉树，也就是按层次的去遍历。依次遍历根节点，然后是左孩子和右孩子。所以要遍历完当前节点的所有孩子，。根据左右孩子的顺序来输出，所以就是先进先出的原则，那么我们当然就想到了队列这个数据结构：




```
/**
 *             1
 *           2   7
 *          3   8 9
 *         4       10
 *        5 6     11 12
 */
const treeData = {
  value: 1,
  left: {
    value: 2,
    left: {
      value: 3,
      left: {
        value: 4,
        left: {
          value: 5
        },
        right: {
          value: 6
        },
      },
    },
  },
  right: {
    value: 7,
    right: {
      value: 9,
      right: {
        value: 10,
```

```
      left: {
        value: 11,
      },
      right: {
        value: 12
      },
    },
  },
  left: {
    value: 8
  }
 }
};

/**
 * 广度优先遍历
 */
const bfs = tree => {
    const stack = [tree];
    const res = [];

    while(stack.length > 0) {
        const head = stack.shift();
        res.push(head.value);

        if(head.left) {
            stack.push(head.left);
        }

        if(head.right) {
            stack.push(head.right);
        }
    }

    return res;
}

const result = bfs(treeData);
console.log(result); //1 2 7 3 8 9 4 10 5 6 11 12
```

快速排序

```
function quickSort(arr) {
  if(arr.length <= 1) {
    return arr;
  }

  const mid = arr.splice(Math.floor(arr.length/2), 1)[0];
  const left = [];
  const right = [];

  for(i = 0; i < arr.length; i++) {
    arr[i] < mid ? left.push(arr[i]) : right.push(arr[i]);
  }
```

```
13
14    return [...quickSort(left), mid, ...quickSort(right)];
15  }
```

冒泡排序

```
1  function bubbleSort(arr) {
2    if(arr.length <= 1) {
3      return arr;
4    }
5
6    for(i = 0; i < arr.length; i++) {
7      for(j = 0; j < arr.length - i - 1; j++) {
8        if(arr[j] > arr[j+1]) {
9          const temp = arr[j];
10         arr[j] = arr[j+1];
11         arr[j+1] = temp;
12       }
13     }
14   }
15
16   return arr;
17 }
```

二分法遍历

```
1  // 二分法主要用于在已排序的数组里寻找合适解
2  function findTar(arr, target) {
3    if(arr[0] === target) {
4      return target;
5    } else if(arr.length <= 1) {
6      return;
7    }
8    arr.sort((a,b) => a - b);
9    const mid = Math.floor(arr.length/2);
10
11   if(arr[mid] > target) {
12     return findMid(arr.slice(0, mid - 1), target);
13   } else {
14     return findMid(arr.slice(mid, arr.length - 1), target);
15   }
16 }
```

二分法找index

```
1  function findIndex(arr, tar) {
2    if (!arr || arr.length < 1) {
3      return -1;
4    }
5
```

```
  let start = 0;
  let end = arr.length - 1;

  while (start <= end) {
    const mid = Math.floor((start + end) / 2);
    if (arr[mid] > tar) {
      start = mid;
    } else if (arr[mid] < tar) {
      end = mid;
    } else if (arr[mid] === tar) {
      return mid;
    } else {
      return -1;
    }
  }
}
```

递归（深度优先）

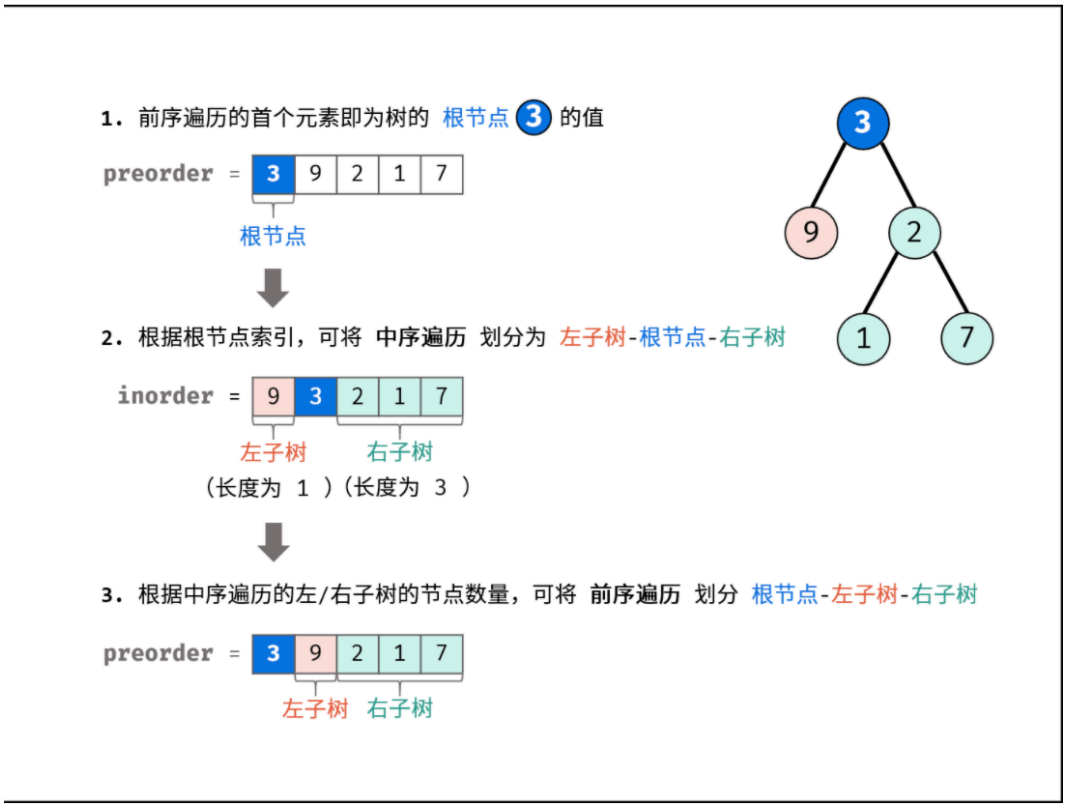
```
function recur(n) {
  const ans = [];
  const s = 'abc';

  function dfs(curStr, depth) {
    if(curStr.length === n) {
      ans.push(curStr);
      return;
    }

    s.split('').forEach((c) => {
      dfs(curStr + c, depth + 1);
    });
  }

  dfs('', 0);
  return ans;
}
```

重建二叉树

输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。

解题思路：

前序遍历的第一个值为根的值，在拿到根的值之后可以在中序遍历中寻找根的index。根的index会将中序遍历分割成两部分，前半部（0到i）为左子树，后半部（i+1到结尾）为右子树。同时index也代表了左子树的长度，用这个长度可以把前序遍历除了根值以外的部分再分成两块，1到i+1为左子树，i+1到结尾为右子树



。

```
var buildTree = function(preorder, inorder) {
    if(!preorder.length) {
        return null;
    }
    const root = new TreeNode(preorder[0]);
    const i = inorder.indexOf(root.val);
    root.left = buildTree(preorder.slice(1, i + 1), inorder.slice(0, i));
    root.right = buildTree(preorder.slice(i + 1), inorder.slice(i + 1));

    return root;
};
```

Promise.all

```
function PromiseAll(prmoises) {
    const res = new Array(promises.length);
    let count = 0;

    return new Promise((resolve, reject) => {
      for(let i = 0; i < promises.length; i++) {
        if (p && p.then) {
          Prmoise.resolve(promises[i]).then((data) => {
            res[i] = data;

            if(count++ === promises.length) {
              resolve(res);
            }
          }).catch((error) => {
            reject(error);
          });
        } else {
```

```
            res[i] = p;
          }
        }
      });
}
```

防抖函数

```
function debounce(fn, delay) {
    let timer; // 维护一个 timer
    return function () {
        let _this = this; // 取debounce执行作用域的this
        let args = arguments;
        if (timer) {
            clearTimeout(timer);
        }
        timer = setTimeout(function () {
            fn.apply(_this, args); // 用apply指向调用debounce的对象，相当于_this.fn(args);
        }, delay);
    };
}
```

节流函数

```
function throttle(fn, delay) {
    let timer;
    return function () {
        let _this = this;
        let args = arguments;
        if (timer) {
            return;
        }
        timer = setTimeout(function () {
            fn.apply(_this, args);
            timer = null; // 在delay后执行完fn之后清空timer，此时timer为假，throttle触发可以进入计时器
        }, delay)
    }
}
```

手写bind

```
Function.prototype.myBind = function(thisArg) {
  // 只有函数可以调用bind
  if (typeof this !== 'function') {
    return
  }

  var _self = this
  var args = Array.prototype.slice.call(arguments, 1)
  var voidFn = function () {} // 定义一个空函数
  var fnBound = function () {
    // bind需指向它的调用者
```

```
    var _this = this instanceof _self ? this : thisArg

    return _self.apply(_this, args.concat(Array.prototype.slice.call(arguments)))
  }
  // 维护原型关系
  if (this.prototype) {
    voidFn.prototype = this.prototype;
  }

  fnBound.prototype = new voidFn();

  return fnBound;
}
```

Array.flat

```
function flat(arr, depth) {
    let depthArg = depth || 1;
    let depthCount = 0;
    let ans = [];

    flatArr = (arr) => {
      arr.map((element, i, self) =>  {
        if (element instanceof Array) {
          if (depthCount < depthArg) {
            depthCount++;
            flatArr(element);
          } else {
            ans.push(element);
          }
        } else {
          ans.push(element);
          if (i === self.lengh - 1) {
            depthCount = 0;
          }
        }
      }
    })

    return ans;
}
```

利用空函数实现继承

这种情况下修改Student的prototype就不会影响到Person的prototype对象了，并且，因为直接将Person的prototype赋给Empty的prototype，所以不会存在特权属性(实例属性)浪费资源的问题。这样利用空函数就能很好的解决共有方法的继承问题了。当然这时Student.prototype中的constructor是Person，所以最好加上Student.prototype.constructor = Student转换过来。

```
function Person(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype = {
```

```
    constructor:Person,
    sayHi:function(){
        alert('hi');
    }
}

function Student(name,age,grade){
    Person.call(this,name,age);
    this.grade = grade;
}

function Empty(){}
Empty.prototype = Person.prototype;

Student.prototype = new Empty();
Student.prototype.constructor = Student;

var p1 = new Person('xiaoming',10);
var s1 = new Student('xiaohong',9,3);
console.log(p1);//Person { name="xiaoming", age=10, sayHi=function()}
console.log(s1);//Student {name="xiaohong", age=9, grade=3, 更多...}
console.log(p1.constructor);//Person(name,age) 父类的实例指向仍是父类
console.log(s1.constructor);//Student(name,age,grade) //子类的实例指向仍是子类
```

循环拷贝实现继承

这种方法直接将父类的共有方法利用遍历的模式拷贝到子类中去。这样就避免了子类实例直接指向父类的问题，也不会出现修改子类的共有方法，对父类产生了影响。也算一种比较完美的继承。

```
function Person(name,age){
    this.name = name;
    this.age = age;
}
Person.prototype = {
    constructor:Person,
    sayHi:function(){
        alert('hi');
    }
}

function Student(name,age,grade){
    Person.call(this,name,age);
    this.grade = grade;
}

for(var i in Person.prototype){Student.prototype[i] = Person.prototype[i]}
Student.prototype.constructor = Student;
Student.prototype.study = function(){
    alert('study');
}
var p1 = new Person('xiaoming',10);
var s1 = new Student('xiaohong',9,3);
console.log(p1);//Person { name="xiaoming", age=10, sayHi=function()}
console.log(s1);//Student { name="xiaohong", age=9, grade=3, 更多...}
console.log(p1.constructor);//Person(name,age) 父类的实例指向仍是父类
```

```
console.log(s1.constructor);//Student(name,age,grade) //子类的实例指向仍是子类
```

多继承

javascript是可以利用call方法和prototype属性来实现多继承的。继承方法与单继承相似，只是将需要继承的多个父类依次实现，另外对于属性或共有方法重命的时候，以最后继承的属性和方法为主。因为会覆盖前面的继承。由于在空函数继承时，会将子类的prototype指向空函数。当用空函数的方法继承复数个父类时，子类的prototype会被最新的空函数重写以致丧失之前继承的熟悉，所以在多继承时无法使用空函数的方法。

```
function Parent1(name,age){
    this.name = name;
    this.age = age;
    this.height=180;
}
Parent1.prototype.say = function(){
    alert('hi...');
}
function Parent2(name,age,weight){
    this.name = name;
    this.age = age;
    this.weight = weight;
    this.height = 170;
    this.skin='yellow';
}
Parent2.prototype.walk = function(){
    alert('walk...');
}

function Child(name,age,weight){
    Parent1.call(this,name,age);
    Parent2.call(this,name,age,weight);
}

for(var i in Parent1.prototype){Child.prototype[i] = Parent1.prototype[i]}
for(var i in Parent2.prototype){Child.prototype[i] = Parent2.prototype[i]}
Child.prototype.constructor = Child;
```

通用的事件监听器

```
const EventUtils = {
  // 视能力分别使用dom0||dom2||IE方式 来绑定事件
  // 添加事件
  addEvent: function(element, type, handler) {
    if (element.addEventListener) {
      element.addEventListener(type, handler, false);
    } else if (element.attachEvent) {
      element.attachEvent("on" + type, handler);
    } else {
      element["on" + type] = handler;
    }
  },

  // 移除事件
  removeEvent: function(element, type, handler) {
```

```
    if (element.removeEventListener) {
      element.removeEventListener(type, handler, false);
    } else if (element.detachEvent) {
      element.detachEvent("on" + type, handler);
    } else {
      element["on" + type] = null;
    }
  },

  // 获取事件目标
  getTarget: function(event) {
    return event.target || event.srcElement;
  },

  // 获取 event 对象的引用，取到事件的所有信息，确保随时能使用 event
  getEvent: function(event) {
    return event || window.event;
  },

  // 阻止事件（主要是事件冒泡，因为 IE 不支持事件捕获）
  stopPropagation: function(event) {
    if (event.stopPropagation) {
      event.stopPropagation();
    } else {
      event.cancelBubble = true;
    }
  },

  // 取消事件的默认行为
  preventDefault: function(event) {
    if (event.preventDefault) {
      event.preventDefault();
    } else {
      event.returnValue = false;
    }
  }
};
```