**任务一：零基础JavaScript编码（一）**

一、JS中innerHTML、outerHTML、innerText 、outerText、value的区别与联系？  
jS中设置或者获取所选内容的值：  
①innerHTML :属性设置或返回该标签内的HTML。也就是从对象的起始位置到终止位置的全部内容,包括Html标签。  
②outerHTML：属性设置或返回该标签及标签内的HTML。也就是从对象的该标签起始到终止位置的全部内容,包括Html标签。  
如果要输出不含HTML标签的内容，可以使用innerHTML取得包含HTML标签的内容后，再用正则表达式去除HTML标签，  
③inneText：从起始位置到终止位置的内容，但它去除html标签。（只能在IE和chrome下使用）  
④outerText 设置(包括标签)或获取(不包括标签)对象的文本 <br><br>  
⑤value：属性可设置或返回密码域的默认值。获取文本框的值  
二、jQuery中的text()、html()和val()  
jQuery中设置或者获取所选内容的值：  
①text();设置或者获取所选元素的文本内容；  
②html();设置或者获取所选元素的内容（包括html标记）；  
③val();设置或者获取表单字段的值（前提是表单设置了value属性）；  
首先，html属性中有两个方法，一个有参，一个无参

1. 无参html（）：取得第一个匹配元素的html内容。这个函数不能用于XML文档。但可以用于XHTML文档，返回的是一个String

例子：

html页面代码：<div><p>Hello</p></div>

jquery代码：$("div").html();

结果：<p>Hello</p>

2.有参html（val）：设置每一个匹配元素的html内容。这个函数不能用于XML文档。但可以用于XHTML文档。返回一个jquery对象

html页面代码：<div></div>

jquery代码：$("div").html("<p>Nice to meet you</p>");

结果：[ <div><p> Nice to meet you</p></div> ]

其次，text属性中有两个方法，一个有参，一个无参

1. 无参text（）：取得所有匹配元素的内容。结果是由所有匹配元素包含的文本内容组合起来的文本。返回的是一个String

例子：

html页面代码：<p><b>Hello</b> fine</p>

<p>Thank you!</p>

jquery代码：$("p").text();

结果：HellofineThankyou!

2.有参text（val）：设置所有匹配元素的文本内容,与 html() 类似, 但将编码 HTML (将 "<" 和 ">" 替换成相应的HTML实体).返回一个jquery对象

html页面代码：<p>Test Paragraph.</p>

jquery代码：$("p").text("<b>Some</b> new text.");

结果:[ <p><b>Some</b> new text.</p> ]

最后，val（）属性中也有两个方法，一个有参，一个无参。  
1.无参val（）：获得第一个匹配元素的当前值。在 jQuery 1.2 中,可以返回任意元素的值了。包括select。如果多选，将返回一个数组，其包含所选的值。 返回的是一个String、 array  
jquery代码：$("p").append( "<b>Single:</b> " + $("#single").val() + " <b>Multiple:</b> " + $("#multiple").val().join(", "));  
结果:[ <p><b>Single:</b>Single<b>Multiple:</b>Multiple, Multiple3</p>]  
2.有参val（val）：设置每一个匹配元素的值。在 jQuery 1.2, 这也可以为check,select,radio元件赋值,返回一个jquery对象  
html页面代码：<input type="text"/>  
jquery代码：$("input").val("hello world!");  
结果：hello world!

\*

欢迎大家和我一起完成学习，[点击访问我的学习](https://github.com/CleverFan/baidu_2017_frontEnd_study/)

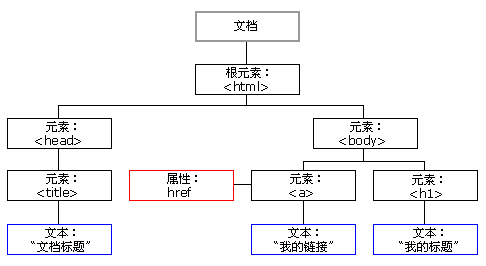
## 总结

### DOM

js的一个很重要的功能就是解析DOM元素，所以，我们有必要了解一下，什么是DOM

DOM = Document Object Model，文档对象模型，DOM可以以一种独立于平台和语言的方式访问和修改一个文档的内容和结构。换句话说，这是表示和处理一个HTML或XML文档的常用方法。

一个html页面会被浏览器解析为一个DOM树:



而js就是一种可以操作这种树的语言。我们可以用js来获取DOM树的一个节点，然后修改它，也可以添加或者删除一个节点，从而达到修改html页面的目的。

接下来我们看一下，如何获得一个DOM节点。

### 使用【document.getElementById】来获取DOM元素

HTML DOM 定义了多种查找元素的方法，除了 getElementById() 之外，还有 getElementsByName() 和 getElementsByTagName()。

不过，如果您需要查找文档中的一个特定的元素，最有效的方法是 getElementById()。

*> 后面两个我们暂时先不管，因为这道题没用到，用到再说。*

在操作文档的一个特定的元素时，最好给该元素一个 id 属性，为它指定一个（在文档中）唯一的名称，然后就可以用该 ID 查找想要的元素。

举个例子：

<html>

<head>

<script type="text/javascript">

**function** **getValue**(){

**var** x = document.getElementById("test")

alert(x.innerHTML)

}

</script>

</head>

<body>

<h1 id="test" onclick="getValue()">This is a test sentence</h1>

<p>Click on the sentence to alert its value</p>

</body>

</html>

点击那句话，就会alert它的值。代码里的【x】就代表【id = "test"】的这个节点。

我们拿到这个节点以后，就可以对这个节点进行操作，比如上述代码中，我们通过这个节点的innerHTML属性得到了这个节点的内容，然后alert它。

那么问题来了，一个节点都有什么属性呢？

### DOM节点的几个常用属性

* innerHTML 属性 : 获取元素内容
* nodeName 属性 : 节点的名称
  + nodeName 是只读的
    - 元素节点的 nodeName 与标签名相同
    - 属性节点的 nodeName 与属性名相同
    - 文本节点的 nodeName 始终是 #text
    - 文档节点的 nodeName 始终是 #document

注释：nodeName 始终包含 HTML 元素的大写字母标签名

* nodeValue 属性 : 节点的值
  + 元素节点的 nodeValue 是 undefined 或 null
    - 文本节点的 nodeValue 是文本本身
    - 属性节点的 nodeValue 是属性值
* nodeType 属性 : 节点的类型

|:-----:|:-----:|  
|元素类型|NodeType|  
|元素|1|  
|属性|2|  
|文本|3|  
|注释|4|  
|文档|5|

### 一些奇怪的问题

* 假如我html中，有两个相同的id，执行document.getElementById会出现什么情况呢？

虽然id叫做唯一标识符，但是由于html并不像java，c++那样会检查你的代码，所以，还是会出现两个id相同的情况的，而document.getElementById只是取一个节点，所以这个时候会发生什么呢？

答案是：它会选择第一个查找到的节点：可以看testGetElementById.html中的代码。当然，哪个会被先查到也是一个值得考虑的问题，不过，还是尽量不要做这种你不好，我也不好的事吧。

* 上面用过innerHTML，然后还用了一个value，有什么区别么？

事实上，除了这两个，还有一个叫做innerText的东西，也是获取值的。我们来看一下这三个有什么区别。

* + innerHTML : 它的作用是获得标签中的值，是节点的一个属性。标签大部分是这样的 ==》 【<xxx>aabb</xxx>】 ，而innerHTML获取的就是这个【aabb】，也就是被标签包裹的值。
  + value : 有些标签会有value值，比如input，你可以这样定义一个input ==> <input id="aqi-input" value="input">,这样的话，你的input标签的默认值就是input，当你修改输入框中的值被你修改之后，这个value属性值也就被改变了，我们就可以通过【节点.value】来获得这个值。
  + innerText : 这个我们没用到，但是还是要说一下，有的时候，我们输入的内容是一串html代码，这个时候，我们通过innerHTML取出来的被转换的代码，但是我们有的时候希望取出来的就是我们输入的html，这个时候，就需要用innerText来实现了。

### 后记

还有很多不完善的地方，之后会再改进，先去撸下一道题了。

**基础！是最重要的！不要好高骛远！**

本文内容大部分来自[w3school](http://www.w3school.com.cn/),大家可以自行查阅。

## innerHTML nodeValue innerText textContent

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Node.innerHTML[**[**doc**](https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML)**]** | **Node.nodeValue[**[**doc**](https://developer.mozilla.org/zh-CN/docs/Web/API/Node/nodeValue)**]** | **Node.innerText[**[**doc**](https://developer.mozilla.org/en-US/docs/Web/API/Node/innerText)**]** | **Node.textContent**[**[doc]**](https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent) |
| Summary | parses content as HTML and takes longer | 对于文本节点返回null,对于 text,comment,CDATA返回该节点的文本内容,对于 attribute 节点返回该属性的属性值 | Takes styles into consideration.It won't get hidden text for instance. | use straight text, doesn't parase HTML,and is faster |

### Node.textContent 与 Node.innerText 的区别

1. textContent会获取所有元素的内容，包括<script>与<style>元素，但 innerText不会。
2. innerText 会受到样式的影响。不返回隐藏元素的文本[**[疑惑]** i.e,display: none;？]，但 textContent 返回
3. 由于 innerText 受 CSS 样式的影响，会触发重排(reflow),但 textContent不会。
4. 在小于 IE11 中对 innerText 进行修改不仅会移除当前元素的子节点，而且会永久性的销毁所有内部文本节点。

### Node.textContent 与 Node.innerHTML 的区别

innerHTML 会返回HTML 文本，以下例作为解释再好不过

// 给定如下**HTML**:

// <**div** id="divA">**This** **is** <span>some</span> text</**div**>

// 获得文本内容:

**var** text = document.getElementById("divA").textContent;

// |text| **is** set to "This is some text".

// 设置文本内容:

document.getElementById("divA").textContent = "This is some text";

// divA的**HTML**现在是这样的:

// <**div** id="divA">**This** **is** some text</**div**>

因此，在 MDN 中并不推荐当需要往一个元素里面写纯文本的时候采用innerHTML,使用textContent使得文本不会被解析为 HTML ，可能会在性能上表现更好。这里提一句：回车(<br>)和空格都是会产生文本节点的。

那什么时候采用innerHTML呢？比如当你需要<em>等之类的标签的时候。

Q： 为什么 firstChild 会返回 null?

firstChild 属性返回被选节点的第一个子节点。 如果选定的节点没有子节点，则该属性返回 NULL。[[doc](http://www.w3school.com.cn/xmldom/prop_element_firstchild.asp)]

* 注意不要写onClick，JS 对大小写有严格的规定
* [addEventListener vs onclick Stackoverflow](http://stackoverflow.com/questions/6348494/addeventlistener-vs-onclick)
* [JAVASCRIPT ANONYMOUS FUNCTIONS](http://blog.scottlogic.com/2011/06/10/javascript-anonymous-functions.html)
* [【原创翻译】生动详细解释javascript的冒泡和捕获，包懂包会](http://www.cnblogs.com/hh54188/archive/2012/02/08/2343357.html)
* [JavaScript 详说事件机制之冒泡、捕获、传播、委托](http://www.cnblogs.com/bfgis/p/5460191.html)
* [词法作用域 VS 动态作用域](http://www.jianshu.com/p/70b38c7ab69c)
* [nodeValue vs innerHTML and textContent. How to choose? Ask Question](http://stackoverflow.com/questions/21311299/nodevalue-vs-innerhtml-and-textcontent-how-to-choose)

## 任务二：零基础JavaScript编码（二）

### 解题思路:

1. 使用filter方法获取空气质量指数大于60的城市；
2. 使用sort方法将空气质量指数大于60的几个城市进行排序；
3. 使用map方法对排好序的数组添加相应格式和相关文字，名次顺序可以使用数组的indexOf方法获得；
4. 使用forEach方法将数组元素遍历添加到<ul>的innerHTML中。

## forEach

**示例:**

**const** colors = ['red', 'green', 'blue'];

*// 不使用forEach*

*// 遍历数组*

**for**(**var** i = 0; i < colors.length; i++){

console.log(colors[i]);

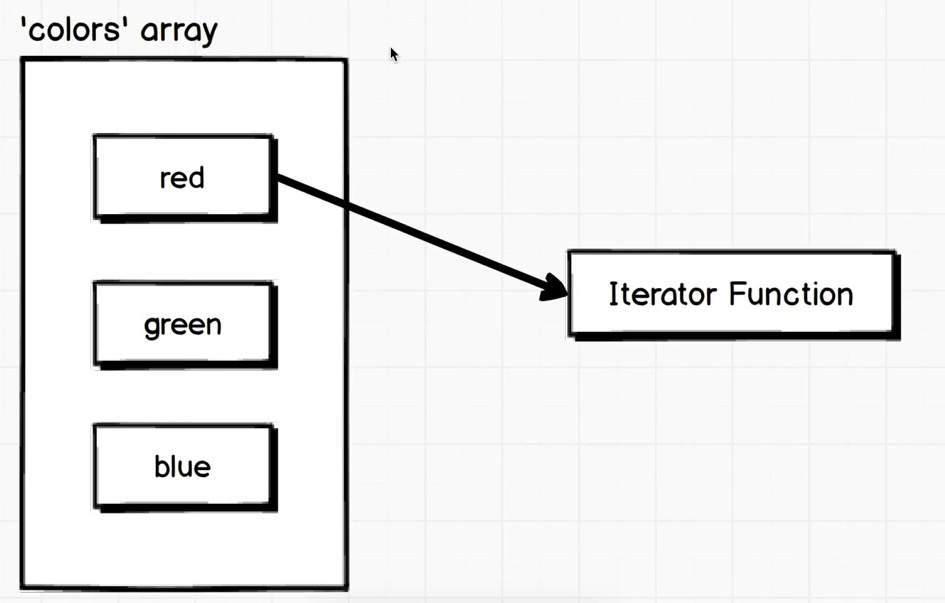
}

*// 使用forEach*

colors.forEach(color => {

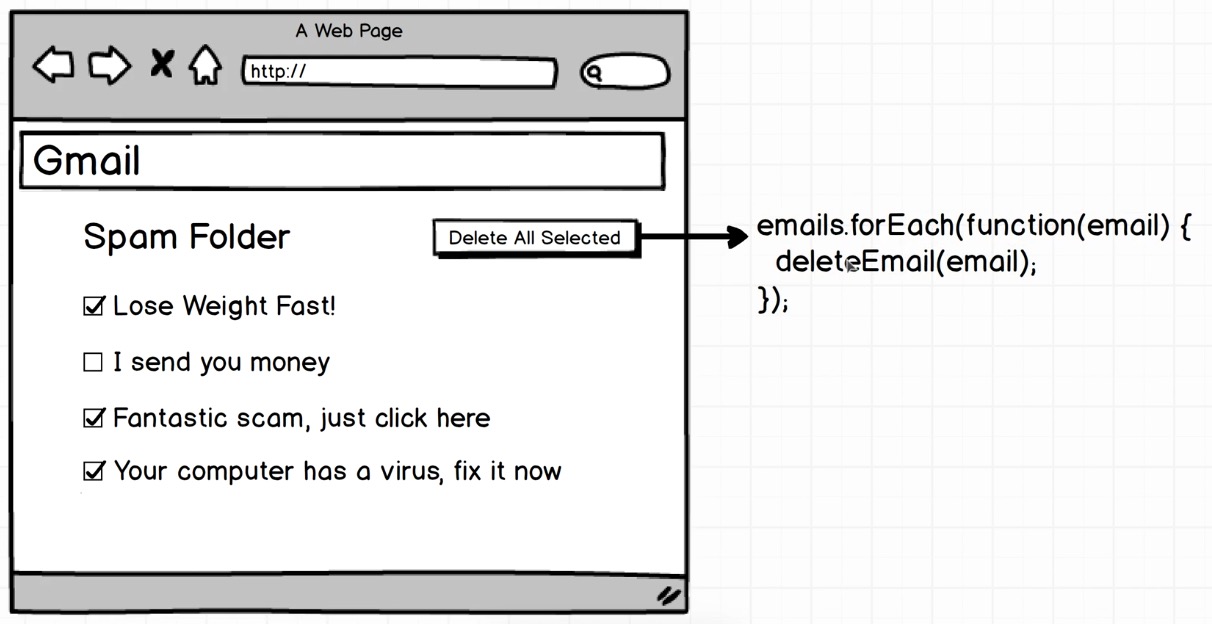
console.log(color);

});



原数组的元素被遍历传入Iterator Function并执行.

**更实际的应用:**



const numbers = [1, 2, 3, 4];

let sum = 0;

numbers.forEach(**number** => {

sum += **number**;

});

console.log(sum);

## map

**示例:**

**const** numbers = [1, 2, 3, 4];

*// 将原数组中各个元素X2输出*

*// 不使用map实现*

**let** doubledNumbers = []; *// 创建一个新数组来存储数据, 为了不改变原数组 (avoid mutating)*

**for**(**var** i = 0; i < numbers.length; i++){

doubledNumbers.push(numbers[i] \* 2)

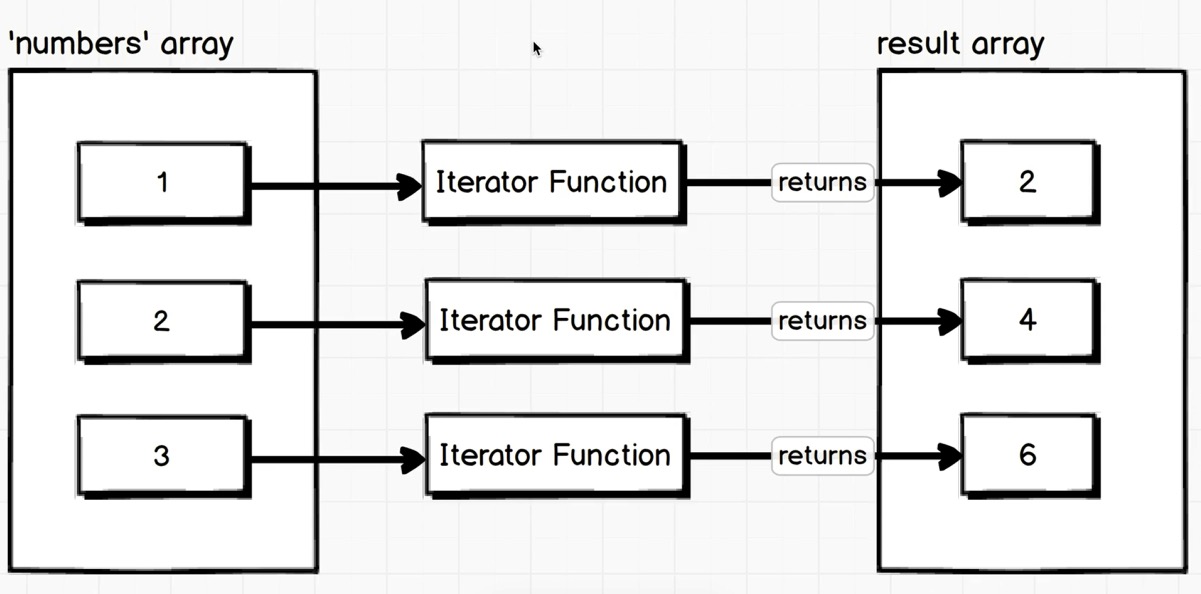
}

console.log(doubledNumbers); *// [2, 4, 6, 8]*

*// 使用map实现*

**let** doubledMap = numbers.map(number => number \* 2)

console.log(doubledMap); *// [2, 4, 6, 8]*



原数组的元素被遍历传入Iterator Function并执行, 执行后的结果被返回并存入新数组.

map可以用于汇总数组内各个元素的属性值, 并输出相关信息. 例子:

**const** cars = [

{model: 'Buick', price: 'cheap'},

{model: 'Camaro', price: 'expensive'}

];

**const** model = cars.map(car => {

**return** car.model;

});

console.log(model); *// ['Buick', 'Camaro']*

**const** price = cars.map(car => {

**return** car.price;

});

console.log(price); *// ['cheap', 'expensive']*

渲染数据列表:

可在浏览器控制台输入以下代码片段查看效果

**const** cars = [

{model: 'Buick', price: 'cheap'},

{model: 'Camaro', price: 'expensive'}

];

**const** carsList = document.createElement('ul');

**const** aboutCars = cars.map(car => {

**return** `<li>${car.model} is ${car.price}</li>`;

});

aboutCars.forEach(car => {

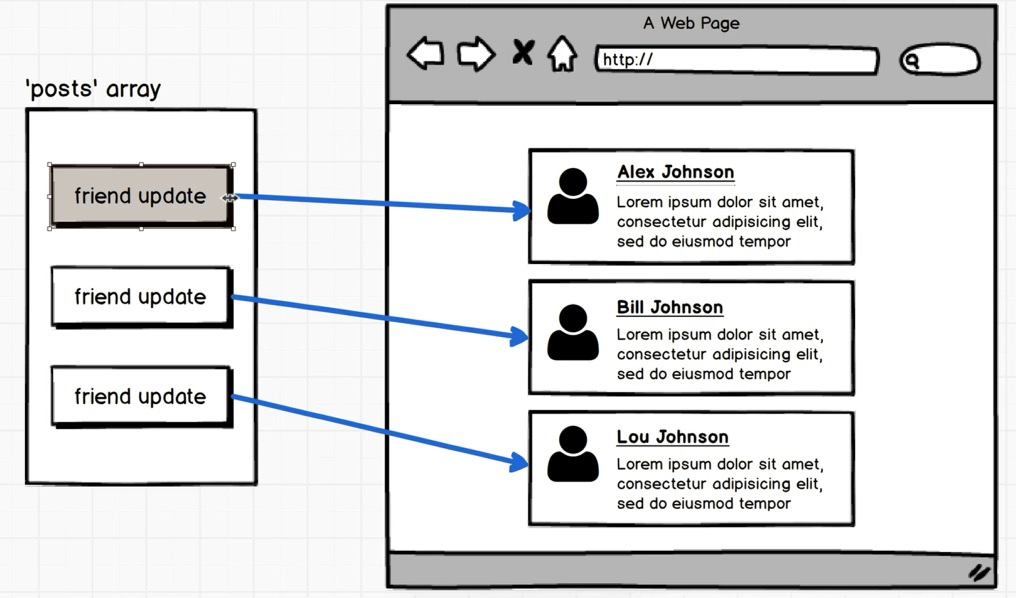
carsList.innerHTML += car;

})

document.body.innerHTML = '';

document.body.appendChild(carsList);

**更实际的应用:**



## filter

**示例:**

const products = [

{ name: 'cucumber', type: 'vegetable', quantity: 10, price: 1 },

{ name: 'banana', type: 'fruit', quantity: 8, price: 15 },

{ name: 'cucumber', type: 'vegetable', quantity: 25, price: 12 },

{ name: 'orange', type: 'fruit', quantity: 30, price: 8 },

]

*// 不使用filter实现*

*// 筛选出类型为水果的产品*

let filteredProducts = [];

**for**(var i = 0; i < products.length; i++){

**if**(products[i].type === 'fruit'){

filteredProducts.push(products[i])

}

}

console.table(filteredProducts);

*// 使用filter实现*

*// 筛选出类型为水果的产品*

const fruit = products.filter(product => product.type === 'fruit');

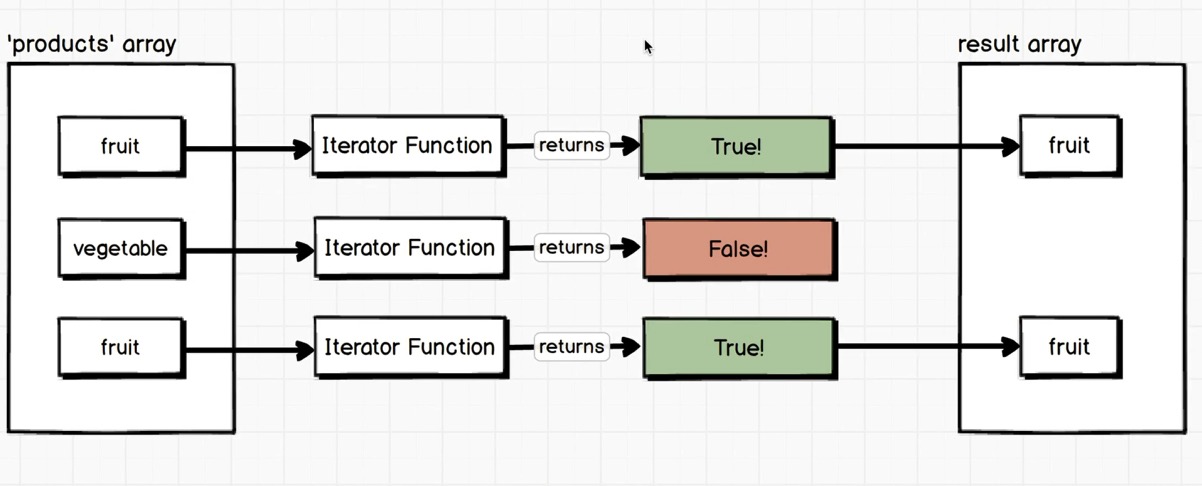
console.table(fruit);

*// 筛选出类型为蔬菜, 且数量大于0, 且价格小于10的产品*

const veg = products.filter(product => product.type === 'vegetable' && product.quantity > 0 && product.price < 10);

console.table(veg);

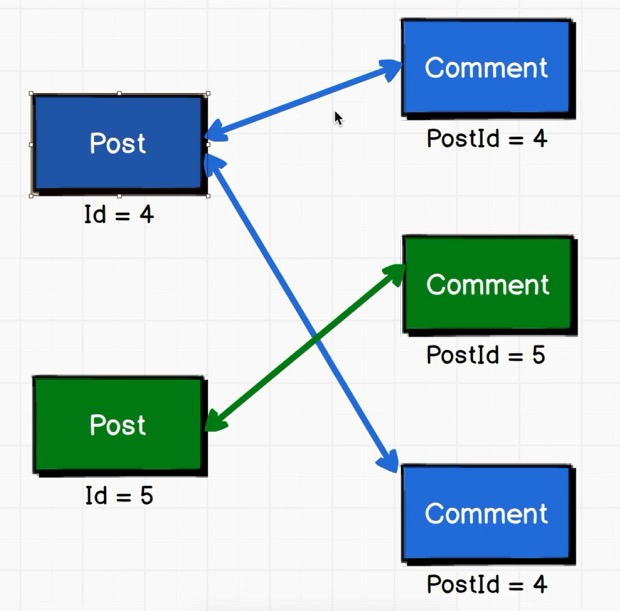
console.table将数组以表格的形式显示在控制台中, 可读性更强.



原数组的元素被遍历传入Iterator Function, Iterator Function返回值的类型为布尔值, 如果返回true, 则该元素被存入结果数组.

**更实际的应用:**

筛选出指定post的comment内容:



**const** post = { id: 4, title: 'New Post'};

**const** comments = [

{ postId: 4, content: 'Awesome Post'},

{ postId: 3, content: 'It was ok'},

{ postId: 4, content: 'neat'}

];

**function** **commentsForPost**(post, comments){

**return** comments.filter(comment => comment.postId === post.id);

}

**const** filteredComments = commentsForPost(post, comments);

console.table(filteredComments);

*## sort*

```js

**function** **compareFunction**(a, b){

*// return -1, a出现在b之前*

*// return 1, a出现在b之后*

*// return 0, 顺序不变*

}

arr.sort(compareFunction);

**示例:**

**var** numbers = [4, 20, 5, 1, 3];

numbers.sort(**function**(a, b) {

**return** a - b;

});

console.log(numbers); *// [1, 3, 4, 5, 20]*

如果没有comparisonFunction, 先将数列中的内容转化成字符串再根据Unicode顺序排序.

var scores = [1, 10, 21, 2];

scores.sort();

console.log(scores); *// [1, 10, 2, 21]*

*// 10在 2之前,*

*// 因为在Unicode码中'10'在'2'之前*

const inventors = [

{ first: 'Albert', last: 'Einstein', year: 1879, passed: 1955 },

{ first: 'Isaac', last: 'Newton', year: 1643, passed: 1727 },

{ first: 'Galileo', last: 'Galilei', year: 1564, passed: 1642 },

{ first: 'Marie', last: 'Curie', year: 1867, passed: 1934 },

{ first: 'Johannes', last: 'Kepler', year: 1571, passed: 1630 },

{ first: 'Nicolaus', last: 'Copernicus', year: 1473, passed: 1543 }

];

*// 生辰从早到晚*

inventors.sort((a, b) => a.year - b.year ? 1 : -1);

console.table(inventors);

*// 活得长的发明家排在后*

inventors.sort((a, b) => {

**return** (a.passed - a.year) - (b.passed - b.year);

});

console.table(inventors);

任务二的内容是这样的：  
**任务描述  
参考以下示例代码，页面加载后，将提供的空气质量数据数组，按照某种逻辑（比如空气质量大于60）进行过滤筛选，最后将符合条件的数据按照一定的格式要求显示在网页上。**

**<!DOCTYPE html>**

<html>

<head>

<meta charset="utf-8">

<title>IFE JavaScript Task 01</title>

</head>

<body>

<h3>污染城市列表</h3>

<ul id="aqi-list">

*<!--*

*<li>第一名：福州（样例），10</li>*

*<li>第二名：福州（样例），10</li> -->*

</ul>

<script type="text/javascript">

**var** aqiData = [

["北京", 90],

["上海", 50],

["福州", 10],

["广州", 50],

["成都", 90],

["西安", 100]

];

(**function** () {

*/\**

*在注释下方编写代码*

*遍历读取aqiData中各个城市的数据*

*将空气质量指数大于60的城市显示到aqi-list的列表中*

*\*/*

})();

</script>

</body>

</html>

**先说一下解题思路：  
1、筛选出值大于60的城市赋值给一个新的数组。  
2、对这个新的数组进行由大到小的排序。  
3、动态创建li标签并打印名次和城市及其空气质量值。**  
结果如下：

**<!DOCTYPE html>**

<html>

<head>

<meta charset="utf-8">

<title>IFE JavaScript Task 01</title>

<style>

**li**{list-style: none;}

</style>

</head>

<body>

<h3>污染城市列表</h3>

<ul id="aqi-list">

*<!--*

*<li>第一名：福州（样例），10</li>*

*<li>第二名：福州（样例），10</li> -->*

</ul>

<script type="text/javascript">

**var** aqiData = [

["北京", 90],

["上海", 50],

["福州", 10],

["广州", 50],

["成都", 90],

["西安", 100]

];

(**function** () {

*/\**

*在注释下方编写代码*

*遍历读取aqiData中各个城市的数据*

*将空气质量指数大于60的城市显示到aqi-list的列表中*

*\*/*

**var** aqiul=document.getElementById('aqi-list');

**var** b=aqiData.filter(**function**(a){**return** a[1]>60;});*//选择出空气质量大于60的城市，赋值给数组b*

*//对空气质量指数由大到小排序*

b.sort(**function**(a,c){

**return** c[1]-a[1];

});

(**function** **wirte**(){ *//输出*

**for**(**var** i=0;i<b.length;i++){

**var** r;

**var** li=document.createElement('li');

aqiul.append(li);

li.innerHTML="第"+(i+1)+"名："+b[i];

}

})();

})();

</script>

</body>

</html>

根据上边的代码和解题思路总结：  
1、筛选出值大于60的城市赋值给一个新的数组：

**var** b=aqiData.**filter**(**function**(a){**return** a[1]>60;});//选择出空气质量大于60的城市，赋值给数组b

//ECMAScript5中的数组方法**filter**()

//**filter**()方法返回的数组元素是调用数组的一个子集，传递的函数是用来逻辑判定的：该函数返回true或false；如果返回值为true或能转化为true的值，那么传递给判定函数的元素就是这个子集的成员，他将被添加到一个作为返回值得函数中。

2、对这个新的数组进行由大到小的排序：

b.sort(**function**(a,c){

**return** c[1]-a[1];

});*//对空气质量指数由大到小排序*

*//javascript数组方法Array.sort()方法将中的元素排序并返回排序后的数组。当不带参数调用sort()时，数组数组元素以字母表顺序排序（如有必要将临时转化为字符串进行比较）。为了按照其他方式而非字母表顺序进行排序，必须给sort（）传递一个比较函数。*

3、动态创建li标签并打印名次和城市及其空气质量值：

(function wirte(){ //输出

for(var i=0*;i<b.length;i++){*

var r*;*

var li=document.createElement('li')*;*

aqiul.append(li)*;*

li.innerHTML="第"+(i+1)+"名："+b[i]*;*

}

})()*;*

//这里不是今天要写的重要内容，暂不做赘述。

在这里总结一下JS的数组方法：  
1、join()  
Array.join()方法将数组中所有元素都转化为字符串并连接在一起，返回最后剩成的字符串。Array.join()方法是String.split()方法的逆向操作，后者是将字符串分割为若干块来创建一个数组。

**var** a=[1,2,3];*//创建一个包含三个元素的数组*

a.**join**();*//=>"1,2,3"*

a.**join**(" ");*//=>"1 2 3"*

a.**join**("");*//=>"123"*

**var** b=**new** Array(10);*//长度为十的空数组*

b.**join**('-');*//=>'---------':9个-组成的字符串*

2、reverse()  
Array.reverse()方法将数组中的元素颠倒顺序，返回逆向数组。他不重新创建一个新的数组，只是在原来的数组里将元素重新排列。

var a=[1,2,3];

a.reverse().join();*//=>"3,2,1"*

3、sort()  
javascript数组方法Array.sort()方法将中的元素排序并返回排序后的数组。当不带参数调用sort()时，数组数组元素以字母表顺序排序（如有必要将临时转化为字符串进行比较）。为了按照其他方式而非字母表顺序进行排序，必须给sort()传递一个比较函数。

var a=new Array("b","c","a");

a.sort();

var s=a.**join**(",");*//s=="a,b,c"*

*//默认方法*

*//由大到小排序，可以参照上面解题答案中。*

4、concat()  
Array.concat()方法创建并返回一个新数组，他的元素包括原始数组的元素和concat()的所有参数。若参数是数组，则链接的事数组元素。

var a=[1,2,3];

a.concat(4,5);*//返回[1,2,3,4,5]*

a.concat([4,5]);*//返回[1,2,3,4,5]*

a.concat([4,5],[6,7]);*//返回[1,2,3,4,5,6,7]*

a.concat(4,[5,[6,7]]);*//返回[1,2,3,4,5,[6,7]]*

5、slice()  
Array.slice()方法返回指定数组的一个片段或一个子数组，他的两个参数分别是指定的片段的开始和介绍位置。返回的数组包含第一个参数指定的位置到第二参数的指定位置间所有元素但不包含第二个参数所指的位置的元素，如果只有一个参数，则返回从开始位置到最后一个元素。如果参数是负数，表示相对于数组中最后一个元素的位置，即-1指定了最后一个元素，-3指定倒数第三个元素。注意，slice()不会修改调用的数组。

var a=[1,2,3,4,5];

a.slice(0,3);*//返回[1,2,3]*

a.slice(3);*//返回[4,5]*

a.slice(1,-1);*//返回[2,3,4]*

a.slice(-3,-2);*//返回[3]*

6、splice()  
Array.splice()方法是在数组中插入或删除元素的通用方法。splice()会修改调用的数组。splice()的第一个参数指定的插入或（和）删除的起始位置，第二个参数指定的应删除的个数，若第二个参数为空，则从起始位置到数组末尾。splice()前两个参数指定了需要删除的数组元素，紧随其后的若干个参数为需要插入到数组的元素，从第一个参数指定的位置开始插入。

var a=[1,2,3,4,5,6,7,8];

a.splice(4);*//返回[5,6,7,8],a是[1,2,3,4]*

a.splice(1,2);*//返回[2,3],a是[1,4]*

a.splice(1,1);*//返回[4],a是[1]*

*//*

var a=[1,2,3,4,5,6,7,8];

a.splice(1,0,"a","c",[1,2]);*//返回[],a是[1,2,'a','c',[1,2],3,4,5,6,7,8]*

7、push()和pop()  
push()和pop()方法允许将数组当做栈来使用，push()方法在数组的尾部添加一个或多个元素，并返回数组新的长度。pop()删除最后一个元素，减少数组长度，并返回它删除的值。两个方法都修改并替换原始数组。

**var** **a**[];

**a**.push(1,2);*//返回2，a:[1,2]*

**a**.pop();*//返回2，a:[1]*

**a**.push([1,2]);*//返回2，a:[1,[1,2]]*

**a**.pop();*//返回[1,2]，a:[1]*

9、toString()和toLocaleString()  
toString()方法将数组每个元素转化为字符并输出用逗号分隔的字符串列表，输出不包括方括号和其他形式的包裹数组值得分隔符。toLocaleString()是toString()的本地化版本。

[1,2,3].toString() //生成'1,2,3'

["a","b",3].toString() //生成'a,b,3'

[1,[2,3]].toString() //生成'1,2,3'

ECMAScript 5中的数组方法：  
1、forEach()  
forEach()方法从头至尾遍历数组，并为每个元素调用指定的函数。

**var** a=[1,2,3,4,5];

**var** sum=0;

a.**forEach**(**function**(value){sum+=value;});*//将每个元素累加到sum*

sum *//=>15*

a.**forEach**(**function**(v,i,a){a[i]=v+1;});*//元素自加1*

a *//=>[2,3,4,5,6]*

2、map()  
map()方法将调用的数组每个元素传递给指定的函数，并返回一个数组，它包含该函数的返回值。

a=[1,2,3];

b=a.map(**function**(x){**return** x\*x;});

3、filter()  
filter()方法返回的数组元素是调用数组的一个子集，传递的函数是用来逻辑判定的：该函数返回true或false；如果返回值为true或能转化为true的值，那么传递给判定函数的元素就是这个子集的成员，他将被添加到一个作为返回值得函数中。

a=[5,4,3,2,1];

b=a.filter(function(x){return x<3});*//[2,1]*

4、every()和some()  
every()和some()方法是数组的逻辑判定：他们对数组元素应用指定的函数进行判定，返回true或false。every()方法只有说有元素返回true时，返回true，相当于&；some()方法只有所有元素返回false时返回false，相当于|。

a=[1,2,3,4,5];

a.every(**function**(x){**return** x<10;});*//=>true：全都小于10*

a.every(**function**(x){**return** x%2===0;});*//=>flase:不是所有的元素都是偶数*

a.some(**function**(x){**return** x%2===0;});*//=>true：a里面含有偶数*

a.some(**function**(x){**return** x>10;});*//=>false:全都不大于10*

5、reduce()和reduceRight()  
reduce()和reduceRight()方法使用指定的函数将数组元素进行组合，生成单个值。reduce()需要两个参数，第一个是其使用的方法，第二个是可选参数，是一个传递给函数的初始值，当第二个参数为空时，意味着第一次调用函数就使用了第一个和第二个数组元素作为其第一个第二个参数，按照数组索引从低到高。reduceRight()的工作原理一样，不同的是它按照数组索引由高到低处理数组，即从数组右侧开始。

**var** a=[1,2,3,4,5];

**var** sum=a.reduce(**function**(x,y){**return** x+y},0);*//数组求和*

**var** p=a.reduce(**function**(x,y){**return** x\*y},1);*//数组求积*

**var** max=a.reduce(**function**(x){**return** (x>y)?x:y;});*//求最大值*

6、indexOf()和lastIndexOf()  
indexOf()和lastIndexOf()搜索整个数组中具有给定值得元素，返回找到的第一个元素的索引或者没有找到就返回-1。indexOF()是从头至尾，而lastIndexOf()则反向搜索。indexOf()和lastIndexOf()方法不接受一个函数作为其参数。第一个参数是需要搜索的值，第二个参数是可选的，他指定数组中一个索引，从哪开始搜索。若第二个参数为负，他代表相对数组的末尾元素的偏移量。

a=[0,1,2,1,0];

a.indexOf(1);*//=>1 a[1]=1*

a.lastIndexOf(1);*//=>3 a[3]=1*

a.indexOf(3);*//=>-1*

*//*

*//在数组中查找所有的x，并返回一个包含匹配索引ed数组*

function findall(a,x){

var results=[];*//将返回的数组*

len=a.length;*//待搜索数组的长度*

pos=0;*//开始搜索的位置*

while(pos<length){

pos=a.indexOf(x,pos);

if(pos===-1) break;

results.push(pos);

pos+=1;

}

return results;

}

赞26

# 欢迎和我一起学习，[链接](https://github.com/CleverFan/baidu_2017_frontEnd_study)

## 总结

### js中的数组

#### 数组的创建

##### 显示的创建

var arrayObj = **new** Array();　*//创建一个数组*

var arrayObj = **new** Array([size]);　*//创建一个数组并指定长度，注意不是上限，是长度*

var arrayObj = **new** Array([element0[, element1[, ...[, elementN]]]]);　*//创建一个数组并赋值*

要说明的是，虽然第二种方法创建数组指定了长度，但实际上所有情况下数组都是变长的，也就是说即使指定了长度为5，仍然可以将元素存储在规定长度以外的，注意：这时长度会随之改变。

##### 直接创建

var arr = [];

var arr = [1,2,3];

var arr = [[1,2],[3,4]];

推荐使用这种方式

#### 数组的访问与操作

直接通过下标进行访问，这个大家应该都知道。

##### 取值

`**var** testGetArrValue=arrayObj[1]; *//获取数组的元素值*`

##### 赋值

`arrayObj[1]= "这是新值"; *//给数组元素赋予新的值`*

##### 数组元素的添加

arrayObj. push([item1 [item2 [. . . [itemN ]]]]);// 将一个或多个新元素添加到数组结尾，并返回数组新长度

arrayObj.unshift([item1 [item2 [. . . [itemN ]]]]);// 将一个或多个新元素添加到数组开始，数组中的元素自动后移，返回数组新长度

arrayObj.splice(insertPos,0,[item1[, item2[, . . . [,itemN]]]]);//将一个或多个新元素插入到数组的指定位置，插入位置的元素自动后移，返回""。

##### 数组元素的删除

arrayObj.pop*()*; *//移除最后一个元素并返回该元素值*

arrayObj.shift*()*; *//移除最前一个元素并返回该元素值，数组中元素自动前移*

arrayObj.splice*(deletePos,deleteCount)*; *//删除从指定位置deletePos开始的指定数量deleteCount的元素，数组形式返回所移除的元素*

##### 数组的截取和合并

arrayObj.slice(**start**, [**end**]); //以数组的形式返回数组的一部分，注意不包括 **end** 对应的元素，如果省略 **end** 将复制 **start** 之后的所有元素

arrayObj.concat([item1[, item2[, . . . [,itemN]]]]); //将多个数组（也可以是字符串，或者是数组和字符串的混合）连接为一个数组，返回连接好的新的数组

##### 数组的拷贝

**arrayObj**.slice(0); *//返回数组的拷贝数组，注意是一个新的数组，不是指向*

**arrayObj**.concat(); *//返回数组的拷贝数组，注意是一个新的数组，不是指向*

##### 数组的反转

arrayObj.**reverse**(); *//反转元素（最前的排到最后、最后的排到最前），返回数组地址*

##### 数组的字符串化

arrayObj.**join**(separator); *//返回字符串，这个字符串将数组的每一个元素值连接在一起，中间用 separator 隔开。*

可以看做split的逆向操作

### 数组的遍历

一般我们使用 for-in 方式进行数组的遍历

**for** (**var** index **in** arrayObj){

**var** obj = arrayObj[index];

}

当然，使用普通的for循环也是可以的：

for (var index = 0； index < arrayObj.length; index++){

var obj = arrayObj[index];

}

### 数组的排序

一般我们使用sort函数进行数组的排序

#### 普通数组的排序

sort()方法有一个可选参数，是用来确定元素顺序的函数。如果这个参数被省略，那么数组中的元素将按照ASCII字符顺序进行排序。如：

var arr = ["a", "b", "A", "B"];

arr.sort();

console.log(arr);//["A", "B", "a", "b"]

如果数组元素是数字呢，结果会是怎样？

var arr = [15, 8, 25, 3];

arr.sort();

console.log(arr);*//[15, 25, 3, 8]*

结果是 [15, 25, 3, 8] 。其实，sort方法会调用每个数组项的toString()方法，得到字符串，然后再对得到的字符串进行排序。虽然数值15比3大，但在进行字符串比较时"15"则排在"3"前面。显然，这种结果不是我们想要的，这时，sort()方法的参数就起到了作用，我们把这个参数叫做比较函数。

比较函数接收两个参数，如果第一个参数应该位于第二个之前则返回一个负数，如果两个参数相等则返回0，如果第一个参数应该位于第二个之后则返回一个正数。例子：

**var** arr = [23, 9, 4, 78, 3];

arr.sort(**function**(a,b){

**return** b-a;

});

console.log(arr); *// [78,23,9,4,3]*

a,b表示数组中的任意两个元素，若return > 0 b前a后；reutrn < 0 a前b后；a=b时存在浏览器兼容

> 简化一下：**a**-b输出从小到大排序，b-**a**输出从大到小排序。

#### 对象数组的排序

如果数组项是对象，我们需要根据数组项的某个属性对数组进行排序

**var** objectList = [];

**function** **Persion**(name,age){

this.name=name;

this.age=age;

}

objectList.push(**new** **Persion**('jack',20));

objectList.push(**new** **Persion**('tony',25));

objectList.push(**new** **Persion**('stone',26));

objectList.push(**new** **Persion**('mandy',23));

*//按年龄从小到大排序*

objectList.sort(**function**(a,b){

**return** a.age-b.age

});

**for**(**var** i=0;i<objectList.length;i++){

document.writeln('<br />age:'+objectList[i].age+' name:'+objectList[i].name);

}

### js中 "+" 的使用

"+" 不仅是四则运算中的加法，在处理字符串的时候，它还是字符串连接符。

看下面几个例子：

var a = 1 + 2;

var b = 1 + "b";

var c = 1 + 2 + "c";

var d = "d" + 1 + 2;

var e = "e" + (1 + 2);

console.log("a=" + a + ",b=" + b + ",c=" + c + ",d=" + d + ",e=" + e)

*//a=3,b=1b,c=3c,d=d12,e=e3*

看了这些，你大概已经知道怎么用了。

解题的时候，有这样一段代码：

**for** (**var** **d** **in** doms){

**var** **li** = document.createElement("li");

**li**.innerHTML = "第" + (**d** + 1) + "名: " + doms[**d**][0] + "," + doms[**d**][1]; *//看这里*

ul.appendChild(**li**);

}

我想输出 【第1名】【第2名】这样的数据，但是如果这样写的话，最后输出的是【第01名】【第12名】

为什么会这样呢？我不是加了括号了么？

问题就出现在d的定义上，[var d],d可不一定就是数字，所以要这样写：

**function** **addLiToUl**(doms) {

**var** ul = document.getElementById("aqi-list");

**for** (**var** d **in** doms){

**var** li = document.createElement("li");

li.innerHTML = "第" + (parseInt(d) + 1) + "名: " + doms[d][0] + "," + doms[d][1];

ul.appendChild(li);

}

}

### 后记

还有很多不完善的地方，之后会再改进，先去撸下一道题了。

**基础！是最重要的！不要好高骛远！**

引用：  
<http://www.cnblogs.com/xljzlw/p/3694861.html>  
<http://www.cnblogs.com/longze/archive/2012/11/27/2791230.html>  
<http://www.cnblogs.com/tianguook/archive/2010/09/30/1839648.html>

## 任务四：基础JavaScript练习（一）

写在开头：

本人Js小白一枚！虽然近两年看了不少关于Js的文档，例如最近在系统地看一本书叫：《JavaScript高级程序设计（第3版）》，但由于工作内容基本都是围绕着html和css，很少需要去写Js，所以知识点基本也是一边看一边忘的状态。

颇为忐忑的发表了这篇笔记，里面很多应该都是最基本应该知道的，只是觉得这些最基本的曾经学习过的方法在实际写代码的时候却一点想不起来，实在是不应该。所以干脆列举出来，毕竟好记性不如烂笔头！

如果下面有理解错误的，望大家能指出，多谢！

## 1.判断值为数字类型的方法

### 需要考虑数字输入的合法性

可以使用 !isNaN(val) 来进行判断value值是否位数字类型，

NAN顾名思义：not a number，

具体例子如下：

**HTML**

<**input** id="num" type="text" />

<**button** id="left\_in">左侧入</button>

<**button** id="right\_in">右侧入</button>

<**button** id="left\_out">左侧出</button>

<**button** id="right\_out">右侧出</button>

<**ul** id="num\_wrap">

</ul>

**JS**

**var** num=document.getElementById("num");

**var** leftIn=document.getElementById("left\_in");

**var** rightIn=document.getElementById("right\_in");

**var** leftOut=document.getElementById("left\_out");

**var** rightOut=document.getElementById("right\_out");

**var** numList=document.getElementById("num\_wrap");

function insertNum(){

**if**(num.value==""){

alert("请输入一个数字");

}**else** **if**(!isNaN(num.value)){

**var** addLi=document.createElement('li');

addLi.innerHTML=num.value;

numList.insertBefore(addLi,numList.firstChild);

num.value="";

}**else**{

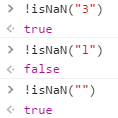
alert("格式不对，请您重新输入一个数字！")

}

}

抛开html部分，以及元素定义的部分，直接到判断数字类型的代码

刚开始我只是考虑到了输入的值为非数值的操作，后来测试的过程中发现，如果在未输入任何数据的情况下点击按钮，依然能够插入一个值为空的li标签，



测试发现!isNAN()方法会在不输入任何值的情况下，依旧返回true，所以我在代码里增加了一个 num.value="" 的判断。

## 2.子节点的插入

* appendChild() 方法：可以向节点的子节点列表的末尾添加新的子节点。语法：appendChild(newchild)
* insertBefore() 方法：可以在已有子节点前插入一个新的子节点。语法：insertChild(newchild,refchild)

## 3.childNodes

点击"左侧出"，读取并删除队列左侧第一个元素，并弹窗显示元素中数值；

点击"右侧出"，读取并删除队列又侧第一个元素，并弹窗显示元素中数值；

写插入数字的需求时，我立刻想到的就是firstChild() 和 lastChild() 这两个方法，事实证明他们也确实分别完成了从左侧和从右侧插入数字的需求。

但是实现读取值并移除的需求时候却出现了问题，childNodes返回的集合中除了包含了子节点li，同时还包含了空格等一些意想不到的东西，因此会在操作过程中造成错误。

解决方法：利用 nodeType 对childNodes返回的集合进行过滤。

但是也有别的方法，除了childNodes以外，还可以使用 querySelectorAll() 方法。

* querySelector()
* querySelectorAll()

以上两种方法都是接收一个CSS选择符，querySelector() 会返回匹配的第一个元素；querySelectorAll() 则是返回一个NodeList实例，避免了使用childNodes的firstChild() 和 lastChild()会出现的问题

但是我最终用的是getElementsByTagName("li")方法。具体代码如下：

*/\**

*\*点击左侧出、右侧出分别读取并删除队列左右第一个元素，并弹窗显示元素中数值*

*\* \*/*

**var** liList=numList.getElementsByTagName("li");

**function** **removeLeft**(){

*//var liNode=numList.querySelectorAll('li');*

*//if(confirm("该值为"+numList.firstChild.innerHTML+"你确定要删除？")){*

*//numList.firstChild.remove();*

*//避免childNodes返回除节点以外其他的值，使用下面两行代码*

**if**(confirm("该值为"+liList[0].innerHTML+"你确定要删除？")){

liList[0].remove();

}**else**{

alert("取消删除")

}

}

## 4.对比querySelectorAll 和getElementsBy系列

### 1.W3C标准

* querySelectorAll 属于 W3C 中的 Selectors API 规范
* getElementsBy 系列则属于 W3C 的 DOM 规范

### 2.浏览器兼容

* querySelectorAll：IE 8+、FF 3.5+、Safari 3.1+、Chrome 和 Opera 10+
* getElementsBy系列：IE 9+、FF 3 +、Safari 3.1+、Chrome 和 Opera 9+

### 3.接收参数

* querySelectorAll：CSS 选择符，如：querySelectorAll('.a .c')、querySelectorAll('li')、querySelectorAll('#a li')
* getElementsBy系列：单一的className、tagName 和 name，如：getElementsByClassName('c')、getElementsBytagName('li')

### 4.返回值

* querySelectorAll返回的是一个 Static Node List（静态的）
* getElementsBy系列返回的是一个 Live Node List（动态的）

### 5.性能

getElementsByTagName()要比querySelectorAll()快很多

原因是因为动态和静态NodeList的区别

关于两种方法的对比，下面是我的参考：  
[点我](https://www.zhihu.com/question/24702250)、[还有我](http://www.cnblogs.com/childsplay/p/5527999.html)

## 5.事件代理

点击队列中任何一个元素，则该元素会被从队列中删除

因为li标签是动态增加和减少的，所以解决这个问题的时候，我首先想到的就是利用**事件代理**，对li的父级标签绑定事件。

**事件代理**主要用到了JavaScript的两个特性：**事件冒泡** 和 **目标元素**。就不展开说了，因为我自己也还没有理解太透彻。呵呵！（尴尬的笑）

关于最后一个问题的具体实现代码如下：

*/\*点击队列中任何一个元素，则该元素会被从队列中删除\*/*

numList.addEventListener("click",**function**(e){

**if**(event.target.nodeName.toLowerCase()=="li"){

*// console.log(event.target);*

numList.removeChild(event.target);

}

})

## appendChild

node.appendChild(newnode);

在父节点node下添加一个子节点newnode, 添加位置在原有子节点的末尾, 返回新添加的节点;

## insertBefore

node.insertBefore(newnode, existingchild);

在父节点node下, 已存在的子节点existingchild前添加一个子节点newnode, 返回插入的节点;

**将新建子节点置于父节点下首位:**  
(父节点下需至少含有一个子节点)

parent.insertBefore(new, parent.firstChild);

**将新建子节点置于父节点下末尾:**  
(相当于appendChild)

parent.insertBefore(new, null);

## nodeType

var type = node.nodeType;

节点类型, 用数字标识, 共有12种节点类型, 其中Element(元素)的nodeType为1, Attr(属性) = 2, Text(文本) = 3.

## childNodes

var kids = node.childNodes;

获取父节点node的第一层所有类型的子节点集合, 返回的是NodeList, 并非数组, 如果要转化成数组需要用Array.from()或者...(spread operator).

## children

var children = node.children;

获取父节点node下的所有第一层元素子节点.

## firstChild

var child = node.firstChild;

获取父节点node的第一层第一个子节点, 若不存在则返回null.

## firstElementChild

var element = node.firstElementChild;

获取父节点node的第一层第一个元素子节点.

## lastElementChild

var element = node.lastElementChild;

获取父节点node的第一层最后一个元素子节点.

## childElementCount

var count = node.childElementCount;

获取父节点node的第一层元素子节点个数, 相当于 node.children.length

## removeChild

node.removeChild(oldnode);

从父节点node中移除某个子节点oldnode, 返回被移除的节点.

## replaceChild

node.replaceChild(newchild, existingchild);

在父节点node下用newchild替换已存在子节点existingchild, 返回被移除的节点;

## createElement

document.createElement([tagname]);

在document下创建元素;

## createTextNode

document.createTextNode('some text');

在document下创建一个内容为'some text'的文字节点;

**用法:**

**var** element = document.createElement('h1');

element.appendChild(document.createTextNode('some text'));

*// 结果: <h1>some text</h1>*

**参考资料:**

[DOM Core](http://reference.sitepoint.com/javascript/domcore)

## 任务五：基础JavaScript练习（二）

在计算机科学所使用的排序算法通常被分类为：

* 计算的 **时间复杂度**（最差、平均、和最好性能），依据列表（list）的大小(n)。一般而言，好的性能是O(n log n)，且坏的性能是O(n^2)。对于一个排序理想的性能是O(n)。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要O(n log n)。
* **存储器使用量**（以及其他电脑资源的使用）
* **稳定性**：稳定排序算法会让原本有相等键值的纪录维持相对次序。也就是如果一个排序算法是稳定的，当有两个相等键值的纪录R和S，且在原本的列表中R出现在S之前，在排序过的列表中R也将会是在S之前。
* **依据排序的方法**：插入、交换、选择、合并等等。

依据排序的方法分类的三种排序算法:

冒泡排序

冒泡排序对一个需要进行排序的数组进行以下操作:

1. 比较第一项和第二项;
2. 如果第一项应该排在第二项之后, 那么两者交换顺序;
3. 比较第二项和第三项;
4. 如果第二项应该排在第三项之后, 那么两者交换顺序;
5. 以此类推直到完成排序;

**实例说明:**

将数组[3, 2, 4, 5, 1]以从小到大的顺序进行排序:

1. 3应该在2之后, 因此交换, 得到[2, 3, 4, 5, 1];
2. 3, 4顺序不变, 4, 5也不变, 交换5, 1得到[2, 3, 4, 1, 5];
3. 第一次遍历结束, 数组中最后一项处于正确位置不会再有变化, 因此下一次遍历可以排除最后一项;
4. 开始第二次遍历, 最后结果为[2, 3, 1, 4, 5], 排除后两项进行下一次遍历;
5. 第三次遍历结果为[2, 1, 3, 4, 5];
6. 最后得到[1, 2, 3, 4, 5], 排序结束;

**代码实现:**

function **swap(items,** firstIndex, secondIndex){

var temp = items[firstIndex]*;*

items[firstIndex] = items[secondIndex]*;*

items[secondIndex] = temp*;*

}*;*

function **bubbleSort(items){**

var len = items.length, i, **j,** stop*;*

for (i = 0*; i < len; i++){*

for (**j** = 0, stop = len-i*; j < stop; j++){*

if (items[**j]** > items[**j+1]){**

**swap(items, j, j+1);**

}

}

}

return items*;*

}

外层的循环决定需要进行多少次遍历, 内层的循环负责数组内各项的比较, 还通过外层循环的次数和数组长度决定何时停止比较.

冒泡排序极其低效, 因为处理数据的步骤太多, 对于数组中的每n项, 都需要n^2次操作来实现该算法(实际比n^2略小, 但可以忽略, 具体原因见⤵️), 即时间复杂度为O(n^2).

对于含有n个元素的数组, 需要进行(n-1)+(n-2)+...+1次操作, 而(n-1)+(n-2)+...+1 = n(n-1)/2 = n^2/2 - n/2, 如果n趋于无限大, 那么n/2的大小对于整个算式的结果影响可以忽略, 因此最终的时间复杂度用O(n^2)表示

选择排序

选择排序对一个需要进行排序的数组进行以下操作:

1. 假定数组中的第一项为最小值(min);
2. 比较第一项和第二项的值;
3. 若第二项比第一项小, 则假定第二项为最小值;
4. 以此类推直到排序完成.

**实例说明:**

将数组["b", "a", "d", "c", "e"]以字母a-z的顺序进行排序:

1. 假定数组中第一项"b"(index0)为min;
2. 比较第二项"a"与第一项"b", 因"a"应在"b"之前的顺序, 故"a"(index1)为min;
3. 然后将min与后面几项比较, 由于"a"就是最小值, 因此min确定在index1的位置;
4. 第一次遍历结束后, 将假定的min(index0), 与真实的min(index1)进行比较, 真实的min应该在index0的位置, 因此将两者交换, 第一次遍历交换之后的结果为["a", "b", "d", "c", "e"];
5. 然后开始第二次遍历, 遍历从第二项(index1的位置)开始, 这次假定第二项为最小值, 将第二项与之后几项逐个比较, 因为"b"就在应该存在的位置, 所以不需要进行交换, 这次遍历之后的结果为"a", "b", "d", "c", "e"];
6. 之后开始第三次遍历, "c"应为这次遍历的最小值, 交换index2("d"), index3("c")位置, 最后结果为["a", "b", "c", "d", "e"];
7. 最后一次遍历, 所有元素在应有位置, 不需要进行交换.

**代码实现:**

function swap(items, firstIndex, secondIndex){

var temp = items[firstIndex];

items[firstIndex] = items[secondIndex];

items[secondIndex] = temp;

};

function selectionSort(){

let items = [...document.querySelectorAll('.num-queue span')].map(num => +num.textContent);

let len = items.length, min;

**for** (i = 0; i < len; i++){

min = i;

**for**(j = i + 1; j < len; j++){

**if**(items[j] < items[min]){

min = j;

}

}

**if**(i != min){

swap(items, i, min);

}

}

return items;

};

外层循环决定每次遍历的初始位置, 从数组的第一项开始直到最后一项. 内层循环决定哪一项元素被比较.

选择排序的时间复杂度为O(n^2).

插入排序

与上述两种排序算法不同, 插入排序是稳定排序算法(stable sort algorithm), 稳定排序算法指不改变列表中相同元素的位置, 冒泡排序和选择排序不是稳定排序算法, 因为排序过程中有可能会改变相同元素位置. 对简单的值(数字或字符串)排序时, 相同元素位置改变与否影响不是很大. 而当列表中的元素是对象, 根据对象的某个属性对列表进行排序时, 使用稳定排序算法就很有必要了.

一旦算法包含交换(swap)这个步骤, 就不可能是稳定的排序算法. 列表内元素不断交换, 无法保证先前的元素排列为止一直保持原样. 而插入排序的实现过程不包含交换, 而是提取某个元素将其插入数组中正确位置.

插入排序的实现是将一个数组分为两个部分, 一部分排序完成, 一部分未进行排序. 初始状态下整个数组属于未排序部分, 排序完成部分为空. 然后进行排序, 数组内的第一项被加入排序完成部分, 由于只有一项, 自然属于排序完成状态. 然后对未完成排序的余下部分的元素进行如下操作:

1. 如果这一项的值应该在排序完成部分最后一项元素之后, 保留这一项在原有位置开始下一步;
2. 如果这一项的值应该排在排序完成部分最后一项元素之前, 将这一项从未完成部分暂时移开, 将已完成部分的最后一项元素移后一个位置;
3. 被暂时移开的元素与已完成部分倒数第二项元素进行比较;
4. 如果被移除元素的值在最后一项与倒数第二项的值之间, 那么将其插入两者之间的位置, 否则继续与前面的元素比较, 将暂移出的元素放置已完成部分合适位置. 以此类推直到所有元素都被移至排序完成部分.

**实例说明:**

现在需要将数组var items = [5, 2, 6, 1, 3, 9];进行插入排序:

1. 5属于已完成部分, 余下元素为未完成部分. 接下来提取出2, 因为5比2大, 于是5被移至靠右一个位置, 覆盖2, 占用2原本存在的位置. 这样本来存放5的位置(已完成部分的首个位置)就被空出, 而2在比5小, 因此将2置于这个位置, 此时结果为[2, 5, 6, 1, 3, 9];
2. 接下来提取出6, 因为6比5大, 所以不操作提取出1, 1与已完成部分各个元素(2, 5, 6)进行比较, 应该在2之前, 因此2, 5, 6各向右移一位, 1置于已完成部分首位, 此时结果为[1, 2, 5, 6, 3, 9];
3. 对余下未完成元素进行类似操作, 最后得出结果[1, 2, 3, 5, 6, 9];

**代码实现:**

function **insertionSort(items)** {

let len = items.length, value, i, **j;**

for (i = 0*; i < len; i++) {*

value = items[i]*;*

for (**j** = i-1*; j > -1 && items[j] > value; j--) {*

items[**j+1]** = items[**j];**

}

items[**j+1]** = value*;*

}

return items*;*

}*;*

外层循环的遍历顺序是从数组的第一位到最后一位, 内层循环的遍历则是从后往前, 内层循环同时负责元素的移位.

插入排序的时间复杂度为O(n^2)

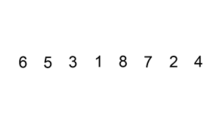
**以上三种排序算法都十分低效, 因此实际应用中不要使用这三种算法, 遇到需要排序的问题, 应该首先使用JavaScript内置的方法Array.prototype.sort();**

**参考:**

1. [2015-CS50-week-算法](https://www.youtube.com/watch?v=IEOO5UToo6A&list=PLhQjrBD2T383Xfn0zECHrOTpfOSlptPAB&index=7)
2. [排序算法-JavaScript描述](https://www.nczonline.net/blog/tag/computer-science/)

计算的 时间复杂度（最差、平均、和最好性能），依据列表（list）的大小(n)。一般而言，好的性能是O(n log n)，且坏的性能是O(n^2)。对于一个排序理想的性能是O(n)。仅使用一个抽象关键比较运算的排序算法总平均上总是至少需要O(n log n)。

## 插入排序(insertion sort)

插入排序应该算是最简单和容易理解的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。具有n个元素时它需要经过n-1趟排序。对于p = 1到p = n-1趟，插入排序保证从位置0到位置p上的元素为已排序状态。它就是基于这个事实来排序的。  


**function** **sort**(arr) {

**if**(arr.length <= 1) {

**return** arr

}

**for**(**var** i=0; i<arr.length; i++) {

**for**(**var** j=i-1; j>=0; j--) {

**if**(arr[j+1] < arr[j]) {

**var** temp = arr[j+1];

arr[j+1] = arr[j];

arr[j] = temp

}

}

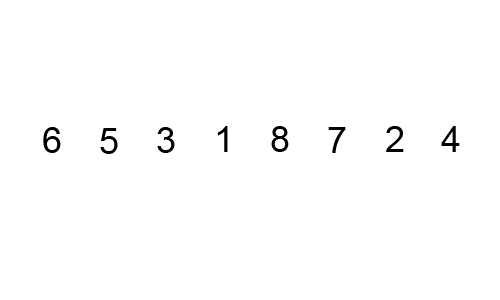
}

**return** arr

}

如果目标是把n个元素的序列升序排列，那么采用插入排序存在最好情况和最坏情况。最好情况就是，序列已经是升序排列了，在这种情况下，需要进行的比较操作需(n-1)次即可。最坏情况就是，序列是降序排列，那么此时需要进行的比较共有n(n-1)/2次。插入排序的赋值操作是比较操作的次数减去(n-1)次。平均来说插入排序算法复杂度为O(n^2)。因而，插入排序不适合对于数据量比较大的排序应用。但是，如果需要排序的数据量很小，例如，量级小于千，那么插入排序还是一个不错的选择。 插入排序在工业级库中也有着广泛的应用，在STL的sort算法和stdlib的qsort算法中，都将插入排序作为快速排序的补充，用于少量元素的排序（通常为8个或以下）

## 冒泡排序(bubble sort)

冒泡排序是与插入排序拥有相等的运行时间，但是两种算法在需要的交换次数却很大地不同。在最好的情况，冒泡排序需要O(n^2)次交换，而插入排序只要最多O(n)交换。冒泡排序的实现（类似下面）通常会对已经排序好的数列拙劣地运行O(n^2)，而插入排序在这个例子只需要O(n)个运算。因此很多现代的算法教科书避免使用冒泡排序，而用插入排序替换之。冒泡排序如果能在内部循环第一次运行时，使用一个旗标来表示有无需要交换的可能，也可以把最好的复杂度降低到O(n)。在这个情况，已经排序好的数列就无交换的需要。若在每次走访数列时，把走访顺序反过来，也可以稍微地改进效率。有时候称为鸡尾酒排序，因为算法会从数列的一端到另一端之间穿梭往返。  
  
冒泡排序算法的运作如下：

* 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
* 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。
* 针对所有的元素重复以上的步骤，除了最后一个。
* 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

由于它的简洁，冒泡排序通常被用来对于程序设计入门的学生介绍算法的概念。

**function** **bubbleSort**(arr) {

**if**(arr.length <= 1) {

**return** arr;

}

**for**(**var** j=0; j<arr.length; j++) {

**for**(**var** i=0; i<arr.length-j; i++) {

**if**(arr[i] > arr[i+1]) {

**var** tmp = arr[i];

arr[i] = arr[i+1];

arr[i+1] = tmp;

}

}

}

**return** arr;

}

## 选择排序(selection sort)

选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。  
选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终位置上，因此对n个元素的表进行排序总共进行至多n-1次交换。在所有的完全依靠交换去移动元素的排序方法中，选择排序属于非常好的一种。

#### 复杂度分析

选择排序的交换操作介于 0 和(n-1)次之间。选择排序的比较操作为n(n-1)/2次之间。选择排序的赋值操作介于0和3(n-1)次之间。比较次数O(n^2)，比较次数与关键字的初始状态无关，总的比较次数 N=(n-1)+(n-2)+...+1=n(n-1)/2。交换次数O(n),最好情况是，已经有序，交换0次；最坏情况是，逆序，交换n-1次。交换次数比冒泡排序较少，由于交换所需CPU时间比比较所需的CPU时间多, n值较小时，选择排序比冒泡排序快。  
原地操作几乎是选择排序的唯一优点，当空间复杂度要求较高时，可以考虑选择排序；实际适用的场合非常罕见。

function selectionSort(arr) {

**if**(arr.length <= 1) {

return arr

}

var i, j, min;

var temp;

**for** (i = 0; i < arr.length - 1; i++) {

min = i;

**for** (j = i + 1; j < arr.length; j++) {

**if** (arr[min] > arr[j])

min = j;

temp = arr[min];

arr[min] = arr[i];

arr[i] = temp;

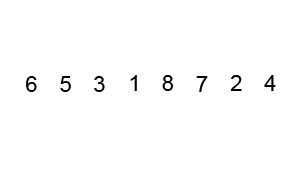
}

}

return arr

}

## 快速排序(quick sort)

快速排序使用分治法（Divide and conquer）策略来把一个序列（list）分为两个子序列（sub-lists）。  
  
步骤为：

1. 从数列中挑出一个元素，称为"基准"（pivot），
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区结束之后，该基准就处于数列的中间位置。这个称为分区（partition）操作。
3. 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。  
   递归的最底部情形，是数列的大小是零或一，也就是永远都已经被排序好了。虽然一直递归下去，但是这个算法总会结束，因为在每次的迭代（iteration）中，它至少会把一个元素摆到它最后的位置去。

正如它的名字，快速排序是在时间中最快的已知排序算法，它的平均运行时间是O(NlogN)。快速排序也是一种分治的递归算法。将数组S排序的基本算法由下列简单的四步组成：

1. 如果S中元素个数是0或1，则返回
2. 取S中任一元素v，称之为枢纽元
3. 将S - {v}分成两个不相交的集合：S1 = {x∈S - {v} | x ≤ v}和S2 = {x∈S - {v} | x ≥ v}
4. 返回{quicksort(S1)}，继续v，继而quicksort(S2)  
   由于对枢纽元的处理会导致第三步中的分割不唯一，因此，我们希望把等于枢纽元的大约一半的关键字分到S1中，而另外一半分到S2中，那怎么去选择一个好的枢纽元呢？  
   选取枢纽元  
   一种错误的方法  
   通常的，没有经过充分考虑的选择是将第一个元素用作枢纽元。如果输入是随机的，那么这是可以接受的，但是如果输入是预排序或是反序的，那么这样的枢纽元就会产生一个劣质的分割，因为所有的元素不是都被划入S1就是被划入S2。  
   一种安全的作法  
   一种安全的方针是随机选取枢纽元。但是另一方面，随机数的生成一般是昂贵的，根本减少不了算法奇遇部分的平均运行时间。  
   三数中值分割法  
   一组N个数的中值是第Math.ceil(N/2)个最大的数。枢纽元的最好的选择是数组的中值。不幸的是，这很难算出，且会减慢快速排序的速度。因此一般的做法是使用左端、右端和中心位置上的三个元素的中值作为枢纽元。例如，输入为8, 1, 4, 9, 6, 3, 5, 2, 7, 0，它的左边元素是8，右边元素是0，中心位置为Math.floor((left + right) / 2)上的元素是6，于是枢纽元v=6。

function quickSort(arr) {

**if** (arr.length <= 1) {

**return** arr.slice(0);

}

**var** **left** = [];

**var** **right** = [];

**var** mid = [arr[0]]; *//first number as a pivot*

**for** (**var** i = 1; i < arr.length; i++) {

**if** (arr[i] < mid[0]) {

**left**.push(arr[i]);

} **else** {

**right**.push(arr[i]);

}

}

**return** quickSort(**left**).concat(mid.concat(quickSort(**right**)));

}

## 任务七：JavaScript和树（一）

### 树的简介

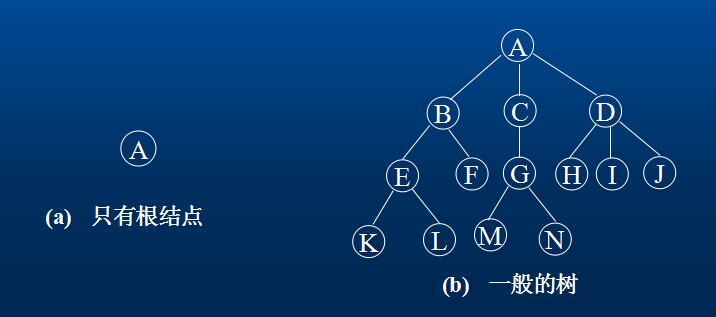
栈、队列、链表等数据结构，都是顺序数据结构。而树是非顺序数据结构。直观地，树型结构是以分支关系定义的层次结构。

#### 基础概念

##### 树的定义

树（Tree）是n（n>=0）个结点的有限集。  
在一颗非空树里：

* 有且只有一个根节点（root）
* 当n>1时，其余结点可分为m(m>0)个互不相交的有限集T1,T2,T3,...Tm，其中每一个集合本身又是一棵树，并且称为根的子树（Subtree）。



(a) 图只有一个根节点A ，（b）A是根节点，13个子节点，13个子节点分成3个互不相交的子集：T1={B,E,F,K,L},t2={D,H,I,J,M};T1,T2和T3都是根A的子树，且本身也是一棵树。

##### 树的节点

树中的每个元素，都叫做节点。  
树的结点包含一个数据元素及若干指向其子树的分支。该节点的子树被称为该节点的孩子并且该节点同时也是孩子的双亲

度：节点拥有的子树数目被称为度

例如（b）B节点的度为2，D节点的度为3，度为0的节点称为叶子或者终端结点,反之称为非终端结点或分支结点。该树的度为拥有节点的度的最大值

##### 节点的层次

节点的层次（Level）从根开始定义起，根为第一层，然后为第二层，依次顺序下去为n层。

深度： 树中节点的最大层数被称为树的深度或高度

例如 （b）树的深度为4

##### 有序和无序树

如果将树中节点的各子树看成从左至右是有次序的（即不能交换），则称该树为有序树，否则称为无序树。在有序树中最左边的子树的根称为第一个孩子，最右边的称为最后一个孩子。

##### 森林

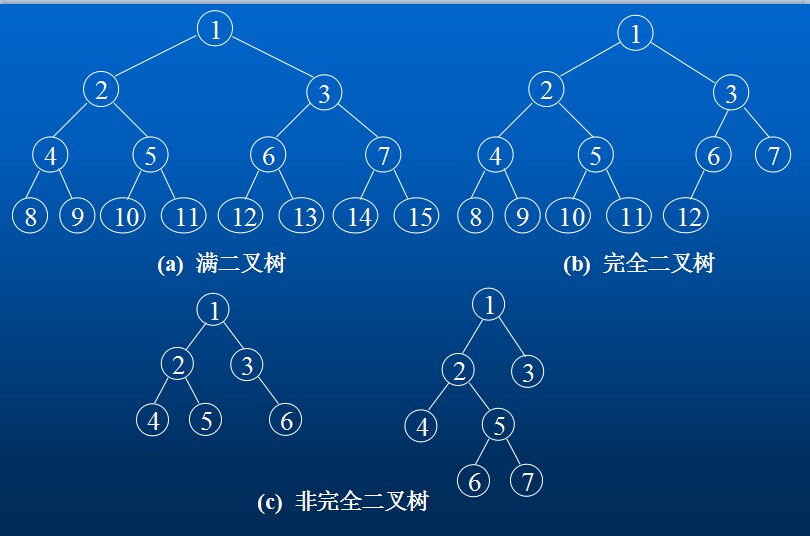
森林（Forest）是m（m>=0）棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为森林。

### 二叉树

二叉树（Binary Tree）是另一种树型结构，它的特点是每个结点至多只有两棵子树（即二叉树中不存在度大于2的结点），并且，二叉树的子树有左右之分（其次序不能任意颠倒。）

#### 二叉树的性质

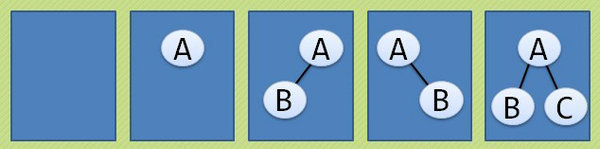
* 二叉树的第i层上最多有2的(i-1)方个节点。（i>=1）
* 深度为k的树最多有2的k次方-1个节点。（k>=1）
* 对任何一棵二叉树T，如果其终端结点数为n0，度为2的结点数为n2，则n0 = n2 + 1;
  + 一棵深度为k且有2的k次方-1个结点的二叉树称为满二叉树。
  + 深度为k的，有n个结点的二叉树，当且仅当其每一个结点都与深度为k的满二叉树中编号从1至n的结点一一对应时，称之为完全二叉树。  
    (我的理解是有限的节点个数里从左到右的编号要一一对应，否则不能称为完全二叉树)



注意：满二叉树一定是完全二叉树，但完全二叉树不一定是满二叉树。

#### 二叉树的5个基本形态

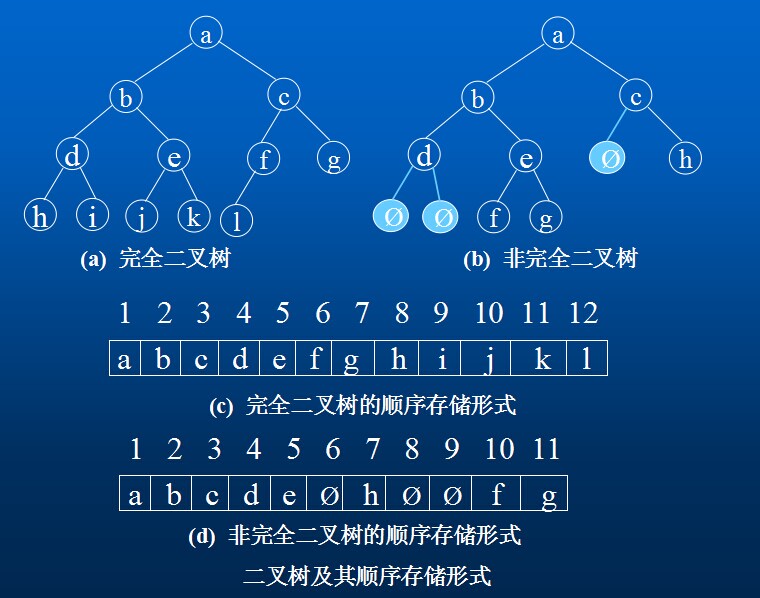
* 空二叉树
* 只有一个根结点
* 根结点只有左子树
* 根结点只有右子树
* 根结点既有左子树又有右子树



#### 二叉树的存储结构

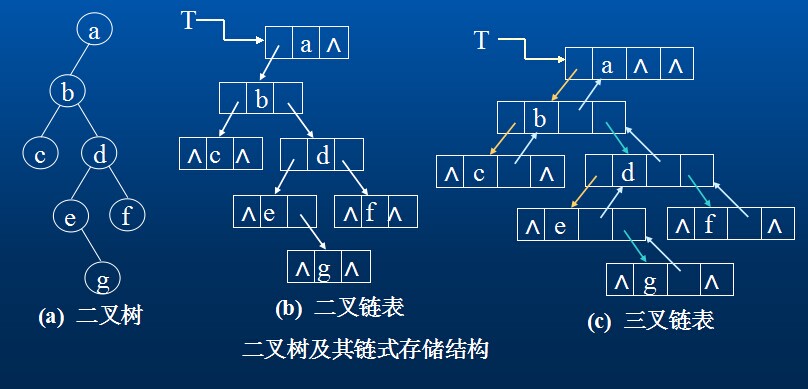
* 顺序存储结构

用一维数组依照顺序存储树的节点，编号为i节点依次存储在数组下标i-1的元素里。i为0意味不存在此节点，这样的顺序存储适合完全二叉树。但是最坏的情况下，深刻为k并且只有k个节点的二叉树（每个节点的度小于2），那么意味着需要长度为2的k次方-1的一维数组来存储，非常浪费存储空间。



* 链式存储结构

二叉树的结点由一个数据元素和分别指向其左右子树的两个分支构成，则表示二叉树的链表中的结点至少包含三个域：数据域和左右指针域。有时，为了便于找到结点的双亲，则还可在结点结构中增加一个指向其双亲结点的指针域。利用这两种结构所得的二叉树的存储结构分别称之为二叉链表和三叉链表。  
在含有n个结点的二叉链表中有n+1个空链域，我们可以利用这些空链域存储其他有用信息，从而得到另一种链式存储结构---线索链表。

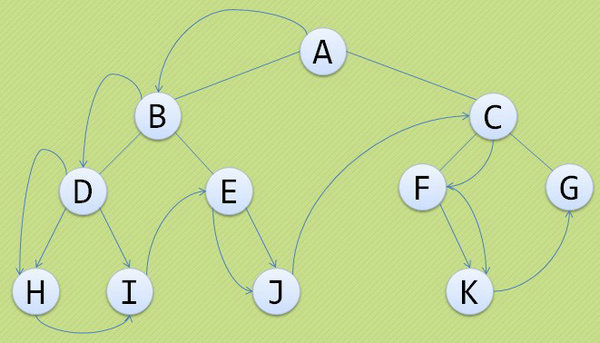


#### 二叉树的遍历

二叉树的遍历(traversing binary tree)是指从根结点出发，按照某种次序依次访问二叉树中所有结点，使得每个结点被访问一次且仅被访问一次。

二叉树的遍历有三种方式，如下：

* （1）前序遍历（DLR），首先访问根结点，然后遍历左子树，最后遍历右子树。简记根-左-右。
* （2）中序遍历（LDR），首先遍历左子树，然后访问根结点，最后遍历右子树。简记左-根-右。
* （3）后序遍历（LRD），首先遍历左子树，然后遍历右子树，最后访问根结点。简记左-右-根。



##### 前序遍历（DLR）

**算法思路**  
若二叉树为空，则遍历结束；否则

　　⑴ 访问根结点；

　　⑵ 先序遍历左子树(递归调用本算法)；

　　⑶ 先序遍历右子树(递归调用本算法)。

遍历的顺序为：A B D H I E J C F K G

**算法实现**

//先序遍历

function preOrder(**node){**

**if**(!**node** **== null**){

putstr(**node**.**show**()+ " ");

preOrder(**node**.**left**);

preOrder(**node**.**right**);

}

}

##### 中序遍历（LDR）

**算法思路**  
若二叉树为空，则遍历结束；否则

　　⑴ 中序遍历左子树(递归调用本算法)；

　　⑵ 访问根结点；

　　⑶ 中序遍历右子树(递归调用本算法)。

遍历的顺序为：A B D H I E J C F K G

**算法实现**

//使用递归方式实现中序遍历

function inOrder(**node){**

**if**(!(**node** **== null**)){

inOrder(**node**.**left**);//先访问左子树

putstr(**node**.**show**()+ " ");//再访问根节点

inOrder(**node**.**right**);//最后访问右子树

}

}

##### 后序遍历（LRD）

**算法思路**  
若二叉树为空，则遍历结束；否则

　　⑴ 后序遍历左子树(递归调用本算法)；

　　⑵ 后序遍历右子树(递归调用本算法) ；

　　⑶ 访问根结点 。

遍历的顺序为：H I D J E B K F G C A

**算法实现**

//后序遍历

function postOrder(**node){**

**if**(!**node** **== null**){

postOrder(**node**.**left**);

postOrder(**node**.**right**);

putStr(**node**.**show**()+ " ");

}

}

个人水平有限，如有错误，还望指正，感激不尽！！！

代码地址：<https://github.com/chunge16/IFE-Task-2017/tree/dev/JavaScript/task-07>  
demo地址：<https://chunge16.github.io/IFE-Task-2017/JavaScript/task-07/>

参考链接

[javascript实现数据结构： 树和二叉树,二叉树的遍历和基本操作](http://www.cnblogs.com/webFrontDev/p/3865719.html)

[JavaScript数据结构和算法之二叉树详解](http://www.jb51.net/article/61021.htm)