Javascript

**前端三层:**

语言 功能

结构层 HTML 搭建结构、放置部件、描述语义

样式层 CSS 美化网页、实现布局

行为层 JavaScript 实现交互效果、数据收发、表单验证等

**Js是一种运行在客户端（浏览器）的编程语言，由ECMAscript和web APIs（DOM，BOM）两部分组成**

**JS的书写位置**

1，在 <body> 中的 <script> 标签中书写 JS 代码

2，将 JS 代码单独保存为 .js 文件

**输出语句**

alert()：弹出消息框

console.log()：控制台输出

document.write():向body标签内输出，可以输出文本或html(例如<h1>)

**输入语句**

prompt(‘请输入姓名’)：会显示一个对话框，包含一个输入框和一条提示信息



**结束符：**分号;可写可不写，要么都写要么都不写，保持风格统一

学会 debugger

当我们需要调试 JavaScript 代码时，我们可以在代码的任何行上写上 debugger;，那么当执行到此处时，代码执行就会自动停止（相当于 “打断点”）

**字面量**

在计算机科学中，字面量（literal）是在计算机中描述 事/物

比如：

Ø 我们工资是： 1000 此时 1000 就是 数字字面量

Ø '黑马程序员' 字符串字面量

Ø 还有接下来我们学的 [] 数组字面量 {} 对象字面量 等等

**变量**

变量的本质:

声明变量之后占据一块内存空间

要想使用变量，第一步就是声明它，并给它赋值。

let b；

var a = 24;

var 定义变量的关键字

a 变量名

= 赋值符号

24 变量值

只能由 字母、数字、\_、$ 组成，但**不能以数字**开头（建议普通代码一律以字母开头）

不能是 关键字 或 保留字

变量名大小写敏感，a 和 A 是两个不同的变量

一个变量只定义，但没有赋初值，默认值是 undefined（未定义）

一个变量只有被 var 定义，并赋初值之后，才算正式初始化完成

一个变量不能被重复声明！

**基本数据类型:**

JS 没有字符型（char），JS 的 String 是基本类型！

Number（数字）（整数，小数，正负数）

String（字符串）

Boolean（布尔）

Undefined（未定义）

Null（空）

**复杂数据类型:**

Object（对象）

Array（数组）

Function（函数）

RegExp（正则表达式）

Date

Map

Set

Symbol

......

**typeof 运算符**

使用 typeof 运算符可以检测值或者变量的类型

例如：

var a = '5';

console.log(typeof a);

//输出number

类型名 typeof 检测结果 值举例

数字类型 number 5、2.5、-0.5

字符串类型 string '前端'、"后端"、'3.14'

布尔类型 boolean true、false

undefined 类型 undefined undefined

null 类型 object（可以理解为空对象） null

**一个特殊的数字型值 NaN**

NaN 是 “not a number” 的意思，即 “不是一个数”，但它属于数字类型。

typeof NaN; // number

0 除以 0 的结果是 NaN，事实上，在 JS 数学运算中，若结果不能得到数字，其结果往往都是 NaN

NaN 有一个 “奇怪” 的性质：不自等

再次强调：NaN 是一个值（特殊的值），不是类型。

唯一能判断 NaN 的方法是通过 isNaN() 函数：

isNaN(NaN); // true

**String（字符串）类型:**

字符串的 length 属性表示字符串的长度。

方法 功能

charAt() 得到指定位置字符

substring() 提取子串

substr() 提取子串

slice() 提取子串

toUpperCase() 将字符串变为大写

toLowerCase() 将字符串变为小写

indexOf() 检索字符串

trim() 删除首尾空格

trimStart() 删除首部空格

trimEnd() 删除尾部空格

substring()、substr() 和 slice() 方法

1. substring(a, b) 方法得到从 a 开始到 b 结束（不包括 b 处）的子串 [a, b)
2. substr(a, b) 中，将得到从 a 开始的长度为 b 的子串
3. slice(a, b) 方法得到从 a 开始到 b 结束（不包括 b 处）的子串,参数可以是负数

**数据类型的转换**

**其他值 ——> 数字**

1，使用 Number() 函数

Number() 函数是 JS 内置函数。

由于 Number() 属于内置构造函数，所以 Number() 的首字母 N 要大写。

（1）字符串 ——> 数字

Number('123'); // 123

Number('123.45'); // 123.45

Number('-123'); // -123

Number('-123.45'); // -123.45

// 字符串中不支持有非数值字符

Number('123年'); // NaN

Number('2e3'); // 2000

Number(''); // 0

// 字符串不支持是数学表达式

Number('1+1'); // NaN

// 除了字符串外，还可以直接放一个表达式

Number(1+1); // 2

（2）布尔值 ——> 数字

Number(true); // 1

Number(false); // 0

（3）undefined 和 null ——> 数字

Number(undefined); // NaN

Number(null); // 0

2，使用 parseInt() 函数

parseInt() 函数的功能是将 字符串 或 浮点数 转为 整数。

自动截掉第一个非数字字符之后的所有字符

parseInt('3.14'); // 3

parseInt('-3.14'); // -3

parseInt('3周吉瑞.14'); // 3

parseInt(3.14); // 3

parseInt(-3.14); // -3

所有文字都将被截掉

parseInt('3.14是圆周率'); // 3

如果字符串以非数字开头，则转为 NaN

parseInt('圆周率是3.14'); // NaN

parseInt('一二三'); // NaN

不存在 “四舍五入”

parseInt('3.99'); // 3

true、false、undefined、null 转为 NaN

**其他值 ——> 字符串**

1，使用 String() 函数

String() 函数是 JS 内置函数。

由于 String() 属于内置构造函数，所以 String() 的首字母 S 要大写。

（1）数字 ——> 字符串

变为 “长得相同” 的字符串。

科学计数法和非 10 进制数字会转为 10 进制的标准值。

String(123); // '123'

String(123.4); // '123.4'

String(2e3); // '2000'

Stiing(NaN); // 'NaN'

String(Infinity); // 'Infinity'

String(0xf); // '15'

（2）布尔值 ——> 字符串

变为 “长得相同” 的字符串。

String(true); // 'true'

String(false); // 'false'

（3）undefined 和 null ——> 字符串

变为 “长得相同” 的字符串。

String(undefined); // 'undefined'

String(null); // 'null'

2，使用 toString() 方法

toString() 是几乎所有基本类型都自带的方法，功能是将值转为字符串

toString() 方法依旧是返回一个字符串，并不改变变量原有的类型

**其他值 ——> 布尔值**

Boolean() 函数是 JS 内置函数。

由于 Boolean() 属于内置构造函数，所以 Boolean() 的首字母 B 要大写。

（1）数字 ——> 布尔值

0 和 NaN 转为 false，其他数字都转为 true。

Boolean(123); // true

Boolean(0); // false

Boolean(NaN); // false

Boolean(Infinity); // true

Boolean(-Infinity); // true

全局属性 Infinity 是一个数值，表示无穷大。

（2）字符串 ——> 布尔值

空字符串 '' 转为 false，其他都转为 true。

Boolean(''); // false

Boolean('abc'); // true

Boolean('false'); // true

（3）undefined 和 null ——> 布尔值

转为 false。

Boolean(undefined); // false

Boolean(null); // false

**JS表达式与操作符**

**算术表达式**

意义 运算符

加 +

减 -

乘 \*

除 /

取余 %

**幂和开根号**

JS 中没有提供幂运算、开根号的运算符，需要使用 Math 对象的相关方法进行计算。

Math.pow(a, b)：求 a 的 b 次方。

Math.sqrt(a)：求 a 的平方根。

Math.pow(2, 3); // 8

Math.pow(3, 2); // 9

Math.sqrt(81); // 9

Math.sqrt(-81); // NaN

**向上取整和向下取整**

Math.ceil()：向上取整。

Math.floor()：向下取整。

Math.round()：把一个数字舍入为最接近的整数（“四舍六入”，“五不一定”）

注意：向上、向下的标准是：X轴正方向为上！

负 ———— 0 ————> 正

注意：由于底层的关系，无论是 toFixed() 还是 Math.round() 都不是完全遵循 “四舍五入”（无论正负），而是 “四舍六入，五不一定”，为什么非要这样呢？原因是没办法！因为小数在计算机底层的实现机制导致的问题！建议对于小数位数的处理不要用原生的函数来处理！最好是手动实现想要的处理代码，或者是借助第三方库！

**关系表达式**

意义 运算符

大于 >

小于 <

大于或等于 >=

小于或等于 <=

等于 ==

不等于 !=

全等于 ===

不全等于 !==

**逻辑表达式**

意义 运算符

非 !

与 &&

或 ||

**逻辑运算的优先级**

优先级：! > && > ||

**赋值表达式**

意义 运算符

赋值 =

快捷赋值 +=、-=、\*=、/=、%=

自增运算 ++

自减运算 --

**JS流程控制**

1. **if 条件语句(同java)**

If，else

1. **switch 选择语句（同java）**

Switch，case，break，default

1. **三元运算符（同java）**

条件表达式 ? 表达式1 : 表达式2;

1. for 循环语句（同java）

【for ... in 循环】（遍历对象或数组）

var a = ['A', 'B', 'C'];

for (var i in a) {

console.log(i + ': ' + a[i]);

}

/\*

0: A

1: B

2: C

\*/

For in遍历的是对象的key和value，如果遍历的是数组，就是索引和值，如果只想提取出数组元素的值，则要用for of

【for ... of 循环】

for of 遍历的是数组元素的值，只是数组内的元素，不包括原型属性和索引！

推荐：变量普通对象用：for in，变量数组及字符串等用：for of

1. **while 循环语句**
2. **break 和 continue**

label 表达式

label 是一个标签，可以配合 break 或 continue 使程序跳转到这个标签处执行（执行 break 或 continue），从而改变程序的执行流程

例如：

break label; 表示跳出循环并执行label后面的语句

注意：label 不是一个特定的关键字，可以随便取名

**JS数组**

**数组的定义**

1，方括号定义法

Let arr = ['A', 'B', 'C', 'D'];

2，new 定义法

let arr = new Array('A', 'B', 'C', 'D');

**数组项**

注意：JS 中数组的元素可以是不同的数据类型！

数组每一项都有下标，下标从 0 开始！

可以使用 数组名[下标] 的形式，访问数组的任一项

**下标越界**

JS 规定，访问数组中不存在的项会返回 undefined，不会报错！

**数组的长度**

数组的 length 属性表示它的长度。

**更改数组项**

如果访问的数组项超过了 length-1，则会创造一个新项。

也就是说，JS 数组是可以动态扩容的！

例如：

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

arr[6] = 0;

console.log(arr); // [1, 2, 3, 4, undefined, undefined, 0]

//arr这个数组从4位扩容到了6位

**数组的遍历**

1. forEach() 方法

arr.forEach(function(item, index, array) {

// ...

});

例如：

var arr = ['Bilbo', 'Gandalf', 'Nazgul'];

arr.forEach(function (item, index, array) {

console.log(`${item} 的索引值为：${index}，在数组：${array} 中`);

});

// Bilbo 的索引值为：0，在数组：Bilbo,Gandalf,Nazgul 中

// Gandalf 的索引值为：1，在数组：Bilbo,Gandalf,Nazgul 中

// Nazgul 的索引值为：2，在数组：Bilbo,Gandalf,Nazgul 中

注意：不能用 return、break 中断循环！

1. map() 方法（生成一个新数组）

const numbers = [1, 2, 3];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // [2, 4, 6]

**数组类型的检测**

数组用 typeof 检测结果是 object。

Array.isArray() 方法可以用来检测数组，返回一个布尔值。

例如：

Array.isArray([1, 2, 3]); // true

Array.isArray([]); // true

**数组的常用方法**

**数组的头尾操作**

1，push() 在尾部插入新项

var arr = [22, 33, 44, 55];

arr.push(66);

arr.push(77, 88, 99);

console.log(arr);

// [22, 33, 44, 55, 66, 77, 88, 99]

2，pop() 在尾部删除。删除数组中的最后一项

var arr = [22, 33, 44, 55];

var item = arr.pop();

console.log(arr); // [22, 33, 44]

console.log(item); // 55

1. unshift() 在头部插入新项
2. shift() 在头部删除

**splice() 方法**

splice() 方法用于替换数组中的指定项。

由于 splice() 可以实现很多功能，所以也称为 JS 的 “多功能方法”。

**替换项**

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

// 从下标为 3 的项开始，连续替换 2 项。即将 'D', 'E' 替换为 'X', 'Y', 'Z'

arr.splice(3, 2, 'X', 'Y', 'Z');

console.log(arr);

// ['A', 'B', 'C', 'X', 'Y', 'Z', 'F', 'G']

**插入项**

var arr = ['A', 'B', 'C', 'D'];

// 从下标为 2 的项开始，连续替换 0 项，即：在 [2] 处插入。

arr.splice(2, 0, 'X', 'Y', 'Z');

console.log(arr);

// ['A', 'B', 'X', 'Y', 'Z', 'C', 'D']

**删除项**

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

// 从下标为 2 的项开始，连续替换 4 项（替换为空，即：删除）。

arr.splice(2, 4);

console.log(arr);

// ['A', 'B', 'G']

注意：splice() 方法会以数组形式返回被替换/删除的项。

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

// 从下标为 3 的项开始，连续替换 2 项。

var item = arr.splice(3, 2, 'X', 'Y', 'Z');

console.log(arr);

// ['A', 'B', 'C', 'X', 'Y', 'Z', 'F', 'G']

console.log(item);

// ['D', 'E']

var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];

// 从下标为 2 的项开始，连续替换 4 项（替换为空，即：删除）。

var item = arr.splice(2, 4);

console.log(arr);

// ['A', 'B', 'G']

console.log(item);

// ['C', 'D', 'E', 'F']

**slice() 方法**

slice() 方法用于得到子数组，类似于字符串中的 slice() 方法。

slice(a, b) 截取的子数组从下标为 a 的项开始，到下标为 b（但不包括下标为 b 的项）结束。

slice(a, b) 方法不会更改原有的数组。

slice() 如果不提供第二个参数，则表示从指定项开始，提取后续所有项作为子数组。

slice() 方法的参数允许为负数，表示数组的倒数第几项（记住不包括最后一项）

**join() 和 split() 方法**

数组的 join() 方法可以使 数组 转为 字符串。

字符串的 split() 方法可以使 字符串 转为 数组。

注意：join() 的参数表示以什么字符作为连接符，如果留空则默认以逗号分隔，如同调用 toString() 方法。

而split() 的参数表示以什么字符拆分字符串，一般不能留空。

例如：

[22, 33, 44, 55].join(); // "22,33,44,55"

[22, 33, 44, 55].toString(); // "22,33,44,55"

[22, 33, 44, 55].join(','); // "22,33,44,55"

[22, 33, 44, 55].join('-'); // "22-33-44-55"

[22, 33, 44, 55].join('~'); // "22~33~44~55"

'abcdefg'.split(); // ["abcdefg"]

'abcdefg'.split(''); // ["a", "b", "c", "d", "e", "f", "g"]

'a-b-c-d-e-f-g'.split(''); // ["a", "-", "b", "-", "c", "-", "d", "-", "e", "-", "f", "-", "g"]

'a-b-c-d-e-f-g'.split('-'); // ["a", "b", "c", "d", "e", "f", "g"]

**concat() 方法**

concat() 方法可以合并连接多个数组（以返回值的形式）。

concat() 方法不会改变原数组。

var arr1 = [1, 2, 3, 4];

var arr2 = [5, 6, 7, 8];

var arr3 = [9, 10, 11];

var arr = arr1.concat(arr2, arr3);

console.log(arr);

// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

**reverse() 方法**

reverse() 方法用来将一个数组中的全部项顺序置反。

var arr = ['A', 'B', 'C', 'D'];

arr.reverse();

console.log(arr); // ["D", "C", "B", "A"]

【小案例】

将字符串 'ABCDEFG' 逆序。

'ABCDEFG'.split('').reverse().join('');

//转换为数组

//反转

//转换回字符串

//输出 "GFEDCBA"

**indexOf() 和 includes() 方法**

indexOf() 方法的功能是搜索数组中的元素，并返回它所在的位置，如果元素不存在，则返回 -1。

includes() 方法的功能是判断一个数组是否包含一个指定的值，返回一个布尔值。

['A', 'B', 'C', 'D'].indexOf('C'); // 2

['A', 'B', 'C', 'D'].indexOf('D'); // 3

['A', 'B', 'C', 'D'].indexOf('X'); // -1

['A', 'B', 'B', 'B'].indexOf('B'); // 1

['A', 'B', 'C', 'D'].includes('D'); // true

['A', 'B', 'C', 'D'].includes('X'); // false

注意：indexOf() 及 includes() 方法的判断标准为 === 全相等！

[11, 22, 33].includes('22'); // false

[11, 22, 33].indexOf('22'); // -1

**sort() 方法**

sort() 方法用原地算法（直接改变原数组）对数组的元素进行排序，并返回数组。

默认排序顺序是将元素转换为字符串，然后根据字典序进行排序（数字 ——> 大写字母 ——> 小写字母，字符串内逐个字符进行比较，相同时比较下一位）

var arr = [3, 18, 10, 24];

console.log(arr.sort()); // [ 10, 18, 24, 3 ]

var arr = ['A', 'a', 'c', 'D', 1];

console.log(arr.sort()); // [ 1, 'A', 'D', 'a', 'c' ]

var arr = ['aa', 'a0', 'aA', 'A1', 'Aa', 'AA'];

console.log(arr.sort()); // [ 'A1', 'AA', 'Aa', 'a0', 'aA', 'aa' ]

sort() 方法可以接收一个函数作为参数，我们可以在这个函数中自定义我们的排序规则：

arr.sort(function(a, b) {

// 比较规则...

});

// 参数a：前一个用于比较的元素。

// 参数b：后一个用于比较的元素。

| **function(a, b) 返回值** | **排序顺序** |
| --- | --- |
| > 0 | a 在 b 后 |
| < 0 | a 在 b 前 |
| === 0 | 保持 a 和 b 的顺序 |

案例1：按数字大小进行排序（降序）：

var arr = [3, 18, 10, 24];

arr.sort(function(a, b) {

if (a > b) {

return -1;

} else if (a < b) {

return 1;

} else {

return 0;

}

});

console.log(arr); // [ 24, 18, 10, 3 ]

**find/findIndex/findLastIndex() 方法**

想象一下，我们有一个对象数组。我们如何找到具有特定条件的对象？

这时可以用 Array.find 方法。

语法如下：

var result = arr.find(function(item, index, array) {

// 如果返回 true，则返回 item 并停止迭代

// 如果返回 false，则继续迭代，如果一直没有返回 true，那么结束时返回 undefined

});

依次对数组中的每个元素调用该函数：

・item 是元素。

・index 是它的索引。

・array 是数组本身。

例如，我们有一个存储用户的数组，每个用户都有 id 和 name 字段。让我们找到 id === 1 的那个用户：

var users = [

{id: 1, name: "John"},

{id: 2, name: "Pete"},

{id: 3, name: "Mary"}

];

var user = users.find(function(item) {

return item.id === 1;

});

console.log(user.name); // 输出结果为 John

**arr.findIndex 方法**（与 arr.find）具有相同的语法，但它返回找到的元素的索引，而不是元素本身。如果没找到，则返回 -1。

**arr.findLastIndex 方法**类似于 findIndex，但从右向左搜索，类似于 lastIndexOf

**filter() 方法**

find 方法搜索的是使函数返回 true 的第一个（单个）元素。

如果需要匹配的有很多，我们可以使用 Array.filter(fn)。

语法与 find 大致相同，但是 filter 返回的是所有匹配元素组成的数组：

filter：过滤、过滤器

var results = arr.filter(function(item, index, array) {

// 如果返回 true，那么 item 被 push 到 results，迭代继续

// 如果什么都没找到，则返回空数组

});

例如：（重要！从后端选取数据常用）

var users = [

{id: 1, name: 'John'},

{id: 2, name: 'Pete'},

{id: 3, name: 'Jerry'},

{id: 4, name: 'Mary'}

];

// 返回前两个用户的数组

var someUsers = users.filter(function(item) {

return item.id < 3;

});

console.log(someUsers); // [ { id: 1, name: 'John' }, { id: 2, name: 'Pete' } ]

**数组方法的应用：数组去重和随机样本**

【数组去重】

题目：去掉数组中的重复项。

思路：准备一个空结果数组（空数组），遍历原数组，如果遍历到的项不在结果数组中，则推入结果数组。

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

var resultArr = [];

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

if (!resultArr.includes(arr[i])) {

resultArr.push(arr[i]);

}

}

console.log(resultArr);

【随机样本】

题目：请随机从原数组中取 3 项。

思路：准备一个空结果数组，遍历原数组，随机选择一项，推入结果数组，并且将这项在原数组中删除。

var arr = [3, 6, 10, 5, 8, 9];

var resultArr = [];

for (var i = 0; i < 3; i++) {

var n = parseInt(Math.random() \* arr.length);

resultArr.push(arr[n]);

arr.splice(n, 1);

}

console.log(resultArr);

**基本类型值和引用类型值**

**区别：**

【当 var a = b 赋值时】

基本类型值：内存中产生新的副本

引用类型值：内存中不产生新的副本，而是让新变量指向同一个对象（内存地址的副本）

// 基本类型值

var a = 3;

var b = a;

a++;

console.log(a); // 4

consloe.log(b); // 3 也就是说b还是a原先的值

// 引用类型值

var arr1 = [1, 2, 3, 4];

var arr2 = arr1;

arr1.push(5);

console.log(arr1); // [1, 2, 3, 4, 5]

console.log(arr2); // [1, 2, 3, 4, 5]

基本类型：number、boolean、string、undefined、null

引用类型：array、object、function、regexp、……

**【相等 == 判断时的区别】**

基本类型进行相等 == 判断时，会比较 “值” 是否相等

引用类型进行相等 == 判断时，会比较 “地址” 是否相等，也就是说它会比较是否为内存中的同一个对象

基本类型：==（自动转型后，比较值）；===（类型相同的情况下，比较值）

引用类型：== 和 === 效果一样！（因为当内存地址相同时，类型肯定相同）

**深克隆和浅克隆**

深克隆和浅克隆需要手写代码实现，而不是简单的调用函数。

使用 arr1 = arr2 的语法不会克隆数组（两个变量指向同一个内存地址）。

**浅克隆**：只克隆数组的第一层，如果是多维数组，或者数组中的项是其他引用类型值，则不克隆其他更深层。

**深克隆**：克隆数组的所有层，要用递归方法。

【浅克隆】

核心思想：“藕断丝连”

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

var resultArr = [];

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

resultArr.push(arr[i]);

}

console.log(resultArr); // [1, 2, 3, 4, [5, 6, 7]]

console.log(resultArr == arr); // false

console.log(resultArr[4] == arr[4]); // true

【深克隆】

使用递归思想，整体思路和浅克隆类似，但稍微进行一些改动：如果遍历到的项是基本类型值，则直接推入结果数组；如果遍历到的项是数组，则重复执行浅克隆的操作。

var arr1 = [33, 44, 11, 22, [77, 88, [33, 44]]];

function deepClone(arr) {

// 结果数组，“每一层”都有一个结果数组

var result = [];

// 遍历数组的每一项

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

// 类型判断，如果遍历到的项为数组

if (Array.isArray(arr[i])) {

// 递归

result.push(deepClone(arr[i]));

} else {

// 如果遍历到的项是基本类型，就直接推入到结果数组中

result.push(arr[i]);

}

}

// 返回结果数组

return result;

}

var arr2 = deepClone(arr1);

console.log(arr1); // [33, 44, 11, 22, [77, 88, [33, 44]]]

console.log(arr2); // [33, 44, 11, 22, [77, 88, [33, 44]]]

console.log(arr1[4] == arr2[4]); // false

**函数的定义**

和变量类似，函数必须先定义然后才能使用。

使用 function 关键字定义函数。

【方式 1 函数声明】

function fun() {

// 函数语句块

}

【方式 2 函数表达式】

var fun = function() {

// 函数语句块

};

function：定义函数的关键字

fun：函数名（必须符合 JS 标识符命名规则）

()：圆括号中是形参列表，即使没有形参，也必须书写圆括号

{}：花括号内为函数语句块

**函数的调用**

执行函数体中的所有语句，就称为 “调用函数”。

调用函数非常简单，只需要在函数名字后书写圆括号对即可。

fun(); // 调用函数

**函数的参数**

参数是函数内的一些待定值，在调用函数时，必须传入这些参数的具体值。

函数的参数可多可少，函数可以没有参数，也可以有多个参数，多个参数之间需要用逗号隔开。

// 形参

function add(a, b) {

var sum = a + b;

console.log('a + b = ' + sum);

}

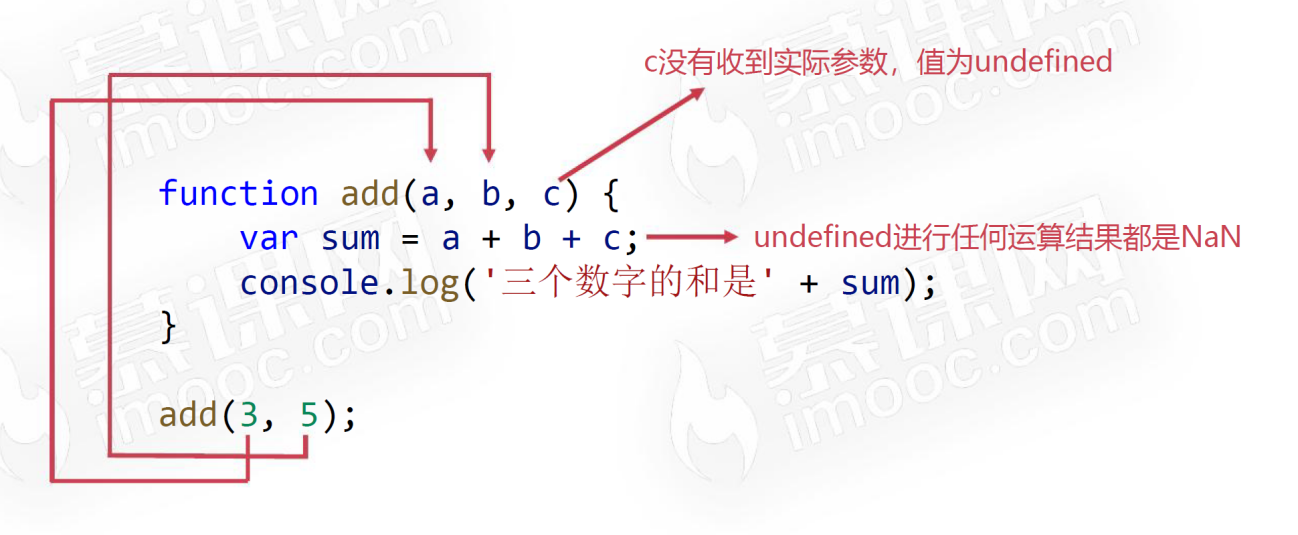
// 实参

add(3, 5);

//圆括号中定义 “形式参数”

//调用函数时传入 “实际参数”

**形参和实参个数不同的情况：**



**arguments**

函数内 arguments 表示它接收到的实参列表，它是一个类数组对象。

类数组对象：所有属性均为从 0 开始的自然数序列，并且有 length 属性，和数组类似可以用方括号书写下标访问对象的某个属性值，但是**不能调用数组的方法**。

function fun() {

console.log(arguments); // 11 22 33 44

console.log(arguments[0]); // 11

console.log(arguments[1]); // 22

console.log(arguments[9]); // undefined

}

fun(11, 22, 33, 44);

【小案例】

JS 本身没有函数的重载（函数名相同，形参个数不同），但是可以借助 arguments 模拟 “函数重载”。

以下例子是一个典型的 “函数重载”，参数个数不同形成 “重载”。

function fun() {

if (arguments.length == 0) {

console.log(0);

} else if (arguments.length == 1) {

console.log(1);

} else {

console.log(2);

}

}

fun(); // 0

fun(1); // 1

fun(1, 2); // 2

**递归**

函数的内部语句可以调用这个函数自身，从而发起对函数的一次迭代。在新的迭代中，又会执行调用函数自身的语句，从而又产生一次迭代。当函数执行到某一次时，不再进行新的迭代，函数被一层一层返回，函数被递归。

函数自己调用自己！

递归是一种较为高级的编程思想，它把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解。

【小案例：求阶乘】

// n! 的本质：n \* (n-1)!

function factorial(n) {

// 递归的出口

if (n == 1) {

return 1;

}

return n \* factorial(n - 1);

}

**变量作用域**

关键字var对应函数作用域：变量只在其所定义时所在的 function 内部有意义。

注：在 ES6 标准中，JS 引入了块级作用域概念！（关键字let和const）

**局部变量**

function fun(n) {

var a = n;

}

fun(3);

console.log(a); // 报错

console.log(n); // 报错

注意：变量 a 是在 fun 函数中被定义的，所以变量 a 只在 fun 函数内部有定义，fun 函数就是 a 的 “作用域”，变量 a 被称为 “局部变量”。函数形参（n）也属于该函数的作用域！

**全局变量**

如果不将变量定义在任何函数的内部（包括形参），此时这个变量就是全局变量，它在任何函数内部都可以被访问和更改。

**遮蔽效应**

如果函数中也定义了和全局变量同名的变量，则函数内的局部变量会将全局的变量进行 “遮蔽”。（以函数内部变量的值优先）【就近原则】

注意：不加 var 将定义全局变量

在初次给变量赋值时，如果没有加 var，则将定义全局变量。

例如：

function fun() {

a = 3;

}

fun();

console.log(a); // 3

注意：没有特殊情况，一律都要记得加 var

**闭包（Closure）概念**

在 JavaScript 中，闭包是指一个函数能够“记住”并访问定义在它外部作用域中的变量，即使在外部函数已经执行完毕的情况下。

简单来说，闭包是函数与其引用的外部变量的结合体。闭包允许函数继续访问其外部函数的变量，即使外部函数已经返回。

“相当于一个大背包（外部函数）套一个小背包（内部函数），大背包已经将拉链封死（函数作用域），但小背包的拉链没拉好（引用了外部函数的变量），留了个口子”

**闭包的形成条件**

闭包是在以下情况下形成的：

1，函数内部定义了另一个函数。

2，内部函数引用了外部函数的变量。

**闭包的工作原理**

外部函数执行时，会创建一个执行上下文，并在其中定义一些变量。

内部函数虽然在外部函数的执行上下文中，但当外部函数执行完成后，内部函数依然可以访问这些外部函数中的变量，因为这些变量被保存在闭包中。

闭包的作用是：让内部函数“记住”外部函数的变量。

**闭包的例子**

示例：

function outer() {

var outerVariable = 'I am outer variable';

function inner() {

console.log(outerVariable); // 访问外部函数中的变量

}

return inner;

}

const closureFunction = outer(); // 这里调用外部函数，返回内部函数（闭包）

closureFunction(); // 输出：I am outer variable

在上面的例子中，outer() 函数返回了 inner() 函数。即使 outer() 已经执行完毕，inner() 仍然可以访问 outer() 中的变量 outerVariable。

**闭包的实际应用：**

闭包特性往往需要将函数 “换一个地方” 执行，才能被直观的体现出来

**闭包用途1 - 模拟私有变量：**

Java中有私有变量的概念，但js中没有，可以用闭包来模拟私有变量。外部无法直接访问这些变量，而只能通过提供的函数来操作它们。

示例：封装私有变量

function createCounter() {

let count = 0; // 私有变量

return {

increment: function() {

count++;

console.log(count);

},

decrement: function() {

count--;

console.log(count);

},

getCount: function() {

return count;

}

};

}

const counter = createCounter();

counter.increment(); // 输出 1

counter.increment(); // 输出 2

console.log(counter.getCount()); // 输出 2

counter.decrement(); // 输出 1

注意：在这里，count 是一个私有变量，只能通过 increment、decrement 和 getCount 方法来操作。这是通过闭包实现的，count 被封装在 createCounter 函数内部，外部无法直接访问或修改它。

**闭包用途2 - 回调函数和异步操作：**

闭包在异步编程中也非常有用，例如在回调函数中使用外部变量。

示例：异步操作中的闭包

function fetchData(url) {

setTimeout(function() {

console.log('Data fetched from ' + url);

}, 1000);

}

fetchData('https://example.com');

注意：在上面的例子中，setTimeout 是异步的，而内部的回调函数可以访问 fetchData 函数中的 url 变量，这就是闭包的应用

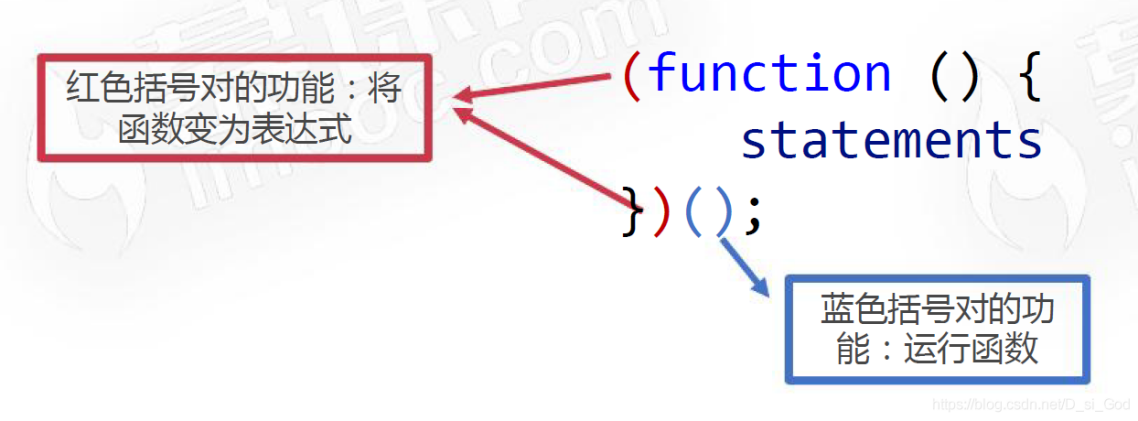
**立即执行函数 IIFE**

IIFE 立即调用函数表达式，是一种特殊的 JS 函数写法，一旦被定义，就立即被调用。

(function () {

// 函数体语句

})();



蓝色括号里写传递的参数



常用 () 来将函数转为 “函数表达式”

**IIFE的作用1 - 为变量赋值**

var age = 12;

var sex = '男';

var title = (function () {

if (age < 18) {

return '小朋友';

} else {

if (sex == '男') {

return '先生';

} else {

return '女士';

}

}

})();

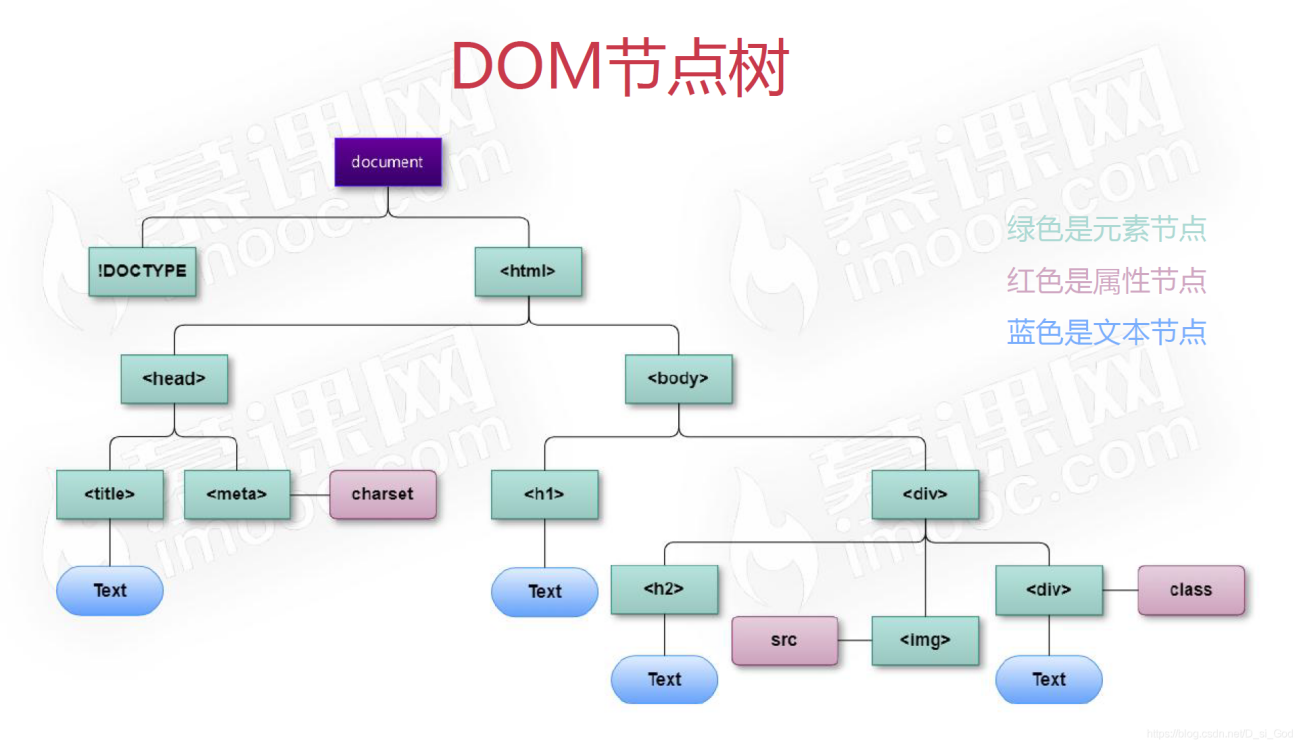
**IIFE的作用2 - 将全局变量变为局部变量**

IIFE 可以在一些场合（如 for 循环中）将全局变量变为局部变量，语法显得紧凑。

在 ES6 中，有更好的方式可以约定变量的作用域！

**DOM**

DOM基本概念



DOM（Document Object Model，文档对象模型）是 JavaScript 操作 HTML 文档的接口，使文档操作变得非常优雅、简便。

DOM 最大的特点就是将 HTML 文档表示为 “节点树”。

DOM 元素/节点：就是渲染到页面上的，一个个的 HTML 标签体（标签 + 属性 + 内容）。

**nodeType**

节点的 nodeType 属性可以显示这个节点具体的类型。

document.nodeType; // 9

| **nodeType值** | **节点类型** |
| --- | --- |
| 1 | 元素节点，例如 <p> 和 <div> |
| 3 | 文字节点 |
| 8 | 注释节点 |
| 9 | document 节点 |
| 10 | DTD 节点（文档类型声明） |

**访问元素节点的常用方法（类似于css的选择器）**

注意：以下方法的参数都是字符串类型 ''。

| **方法** | **功能** | **兼容性** |
| --- | --- | --- |
| document.getElementById() | 通过 id 得到**元素** | IE 6 |
| document.getElementsByTagName() | 通过标签名得到**元素数组** | IE 6 |
| document.getElementsByClassName() | 通过类名得到**元素数组** | IE 9 |
| document.querySelector() | 通过选择器得到**元素** | IE 8 部分兼容、IE 9 完全兼容 |
| document.querySelectorAll() | 通过选择器得到**元素数组** | IE 8 部分兼容、IE 9 完全兼容 |

注意：querySelector() 方法的功能是通过选择器得到元素。

例如：

HTML

<div id="box1">

<p>我是段落</p>

<p class="spec">我是段落</p>

<p>我是段落</p>

</div>

JS

var the\_p = document.querySelector('#box1 .spec');

**操作节点**

1. **改变元素节点中的内容**

innerHTML 属性能以 HTML 语法设置节点中的内容

innerText 属性只能以纯文本的形式设置节点中的内容

1）innerHTML 属性允许你获取或设置 HTML 元素的HTML 内容，包括嵌套的标签和文本内容。它返回或设置元素内的整个 HTML 结构（包括标签）。

获取 innerHTML：返回元素的 HTML 内容（包括所有嵌套的 HTML 标签）。

设置 innerHTML：可以用来更改元素内的 HTML 内容，包括添加 HTML 标签。

示例：使用 innerHTML

<div id="myDiv">

<p>Hello, <strong>World</strong>!</p>

</div>

<script>

// 获取 innerHTML

var divContent = document.getElementById('myDiv').innerHTML;

console.log(divContent); // 输出: <p>Hello, <strong>World</strong>!</p>

// 设置 innerHTML

document.getElementById('myDiv').innerHTML = '<h1>New Content</h1>';

console.log(document.getElementById('myDiv').innerHTML);

// 输出: <h1>New Content</h1>

</script>

注意：

innerHTML 可以用来插入新的 HTML 标签。

使用 innerHTML 时，所有的 HTML 标签都会被解析并插入到指定元素内，包括 <script> 标签，这可能会引发安全问题（例如跨站脚本攻击 XSS）。因此，使用时要小心。

2）innerText 属性用于获取或设置元素的文本内容，它返回的是该元素的纯文本，不包括 HTML 标签。innerText 会自动去掉 HTML 标签，只获取显示在页面上的文本。

获取 innerText：返回元素内的纯文本内容（不包括 HTML 标签）。

设置 innerText：可以用来设置元素的纯文本内容，HTML 标签将会被视为文本而不会被解析。

示例：使用 innerText

<div id="myDiv">

<p>Hello, <strong>World</strong>!</p>

</div>

<script>

// 获取 innerText

var divText = document.getElementById('myDiv').innerText;

console.log(divText); // 输出: Hello, World!

// 设置 innerText

document.getElementById('myDiv').innerText = 'New Content';

console.log(document.getElementById('myDiv').innerText);

// 输出: New Content

</script>

注意：

innerText 会受到浏览器样式的影响（例如 display: none 的元素不会被获取到）。

在设置 innerText 时，HTML 标签会被当作普通文本处理。

1. **改变元素节点的CSS样式**

改变元素节点的 CSS 样式需要使用这样的语句：

节点元素名.style.backgroundColor = 'red';

节点元素名.style.backgroundImage = 'url(images/1.jpg)';

节点元素名.style.fontSize = '32px';

1. **改变元素节点的HTML属性**

标准 W3C 属性，如 src、href、title、alt 等等，只需要直接打点进行更改即可。

例如：

oImg.src = 'images/2.jpg';

将图片地址改变

**节点的创建**

document.createElement() 方法用于创建一个指定 tag name 的 HTML 元素。

var oDiv = document.createElement('div');

创建了一个元素名为oDiv的div标签

此时新创建出的节点是 “孤儿节点”，这意味着它并没有被挂载到 DOM 树上，虽然你可以对它进行编辑，比如增加文本，改变样式，但在html里无法看见它。

必须继续使用 appendChild() 或 insertBefore() 方法将孤儿节点插入到 DOM 树上之后，才能看到这个节点。

appendChild()

任何已经在 DOM 树上的节点，都可以调用 appendChild() 方法，它可以将孤儿节点挂载到它的内部，成为它的最后一个子节点。

语法如下：

父节点.appendChild(孤儿节点);

insertBefore()

任何已经在 DOM 树上的节点，都可以调用 insertBefore() 方法，它可以将孤儿节点挂载到它的内部，成为它的 “标杆子节点” 之前的节点。

语法如下：

父节点.insertBefore(孤儿节点, 标杆节点);

**移动节点**

如果对已经挂载到 DOM 树上的节点成为 appendChild() 或者 insertBefore() 的参数，这个节点将会被移动。

语法如下：

新父节点.appendChild(已经有父亲的节点);

新父节点.insertBefore(已经有父亲的节点, 标杆子节点);

这意味着一个节点不能同时位于 DOM 树的两个位置。

**删除节点**

removeChild() 方法从 DOM 中删除一个子节点。

语法如下：

父节点.removeChild(要删除子节点);

节点不能主动删除自己，必须由父节点删除它。

**克隆节点**

cloneNode() 方法可以克隆节点，克隆出的节点是 “孤儿节点”。

必须继续使用 appendChild() 或 insertBefore() 方法将孤儿节点插入到 DOM 树上。

var 孤儿节点 = 老节点.cloneNode();

var 孤儿节点 = 老节点.cloneNode(true);

注意：参数是一个布尔值，表示是否采用深度克隆：如果为 true，则该节点的所有后代节点也会被克隆，如果为 false，则只克隆该节点本身

**监听事件**

1. **直接通过盒子添加监听事件**

例如：

Html：<div id="box"></div>

Js：var oBox = document.getElementById(‘box’)

oBox.onclick = function() {

// 当点击盒子时，将执行这里的语句}

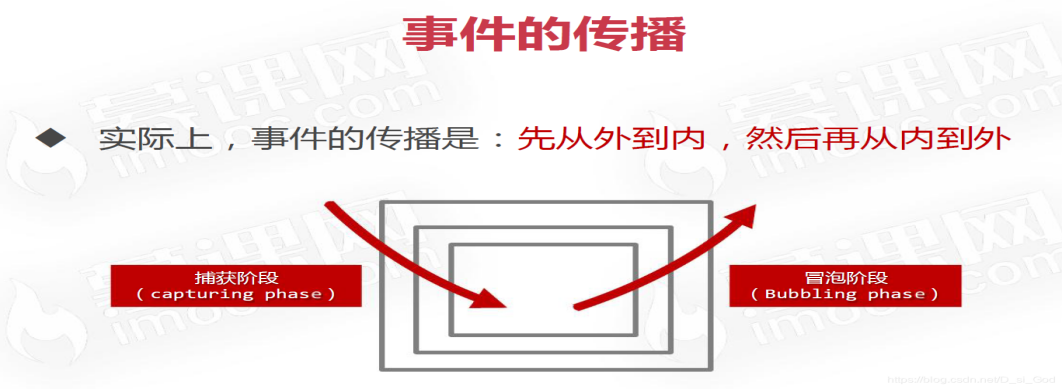
1. **addEventListener()方法**

oBox.addEventListener('click', function(){}, true);

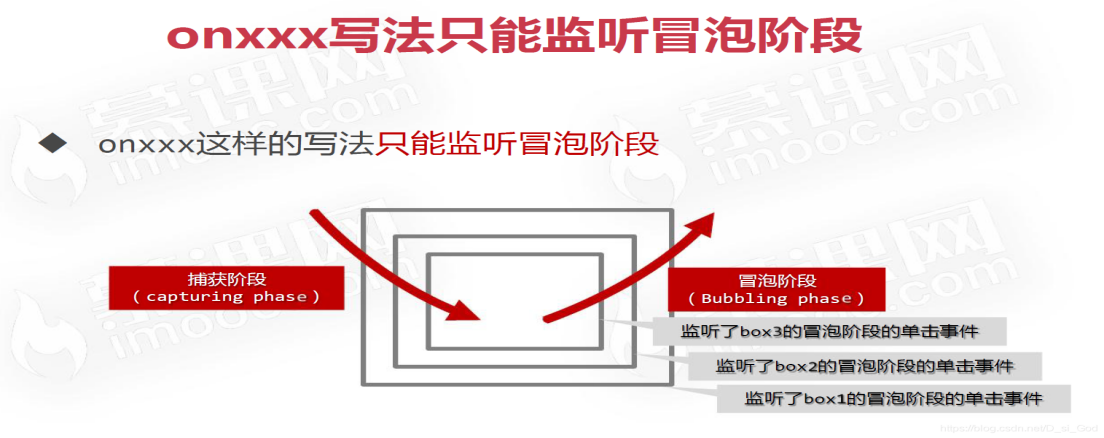
**事件传播**

当多个盒子嵌套，且每个盒子都有各自的监听事件时，就要引入事件传播的概念

捕获阶段和冒泡阶段：



而onclik写法只能监听冒泡阶段，也就是说捕获阶段的监听将不会被执行，一开始就直接从最内层的监听开始执行



如果有先执行捕获阶段，再执行冒泡阶段的需要，就需要用addEventListener()方法



阻止事件冒泡（Event Bubbling）

event.stopPropagation();

这是阻止事件继续冒泡到祖先元素（父、祖父...）的标准方式。

**常见的鼠标事件监听**

| **事件名** | **事件描述** |
| --- | --- |
| onclick | 当鼠标单击某个对象 |
| ondblclick | 当鼠标双击某个对象 |
| onmousedown | 当某个鼠标按键在某个对象上被按下 |
| onmouseup | 当某个鼠标按键在某个对象上被松开 |
| onmousemove | 当某个鼠标按键在某个对象上被移动 |
| onmouseenter | 当鼠标进入某个对象（相似事件 onmouseover） |
| onmouseleave | 当鼠标离开某个对象（相似事件 onmouseout） |

**常见的键盘事件监听**

| **事件名** | **事件描述** |
| --- | --- |
| onkeypress | 当某个键盘的键被按下（系统按钮如箭头键和功能键无法得到识别） |
| onkeydown | 当某个键盘的键被按下（系统按钮可以识别，并且会先于 onkeypress 发生） |
| onkeyup | 当某个键盘的键被松开 |

**常见的表单事件监听**

| **事件名** | **事件描述** |
| --- | --- |
| onchange | 当用户改变完成域的内容 |
| onfocus | 当某元素获得焦点（比如 tab 键或鼠标点击） |
| onblur | 当某元素失去焦点 |
| onsubmit | 当表单被提交 |
| onreset | 当表单被重置 |

**常见的页面事件监听**

| **事件名** | **事件描述** |
| --- | --- |
| onload | 当页面或图像被完成加载 |
| onunload | 当用户退出页面 |

**移除监听事件（停止监听）**

**removeEventListener()方法**

当我们 addEventListener() 后，该监听事件就会一直生效，直到关闭页面或是移除该对应的监听！

removeEventListener() 方法用来移除监听事件（只能移除具名函数的监听，且方法名称后面不能带）

**e.charCode和e.keyCode属性**

e.charCode 属性通常用于 onkeypress 事件中，表示用户输入的字符的 “字符码”。

e.keyCode 属性通常用于 onkeydown 事件和 onkeyup 中，表示用户按下的按键的 “键码”。

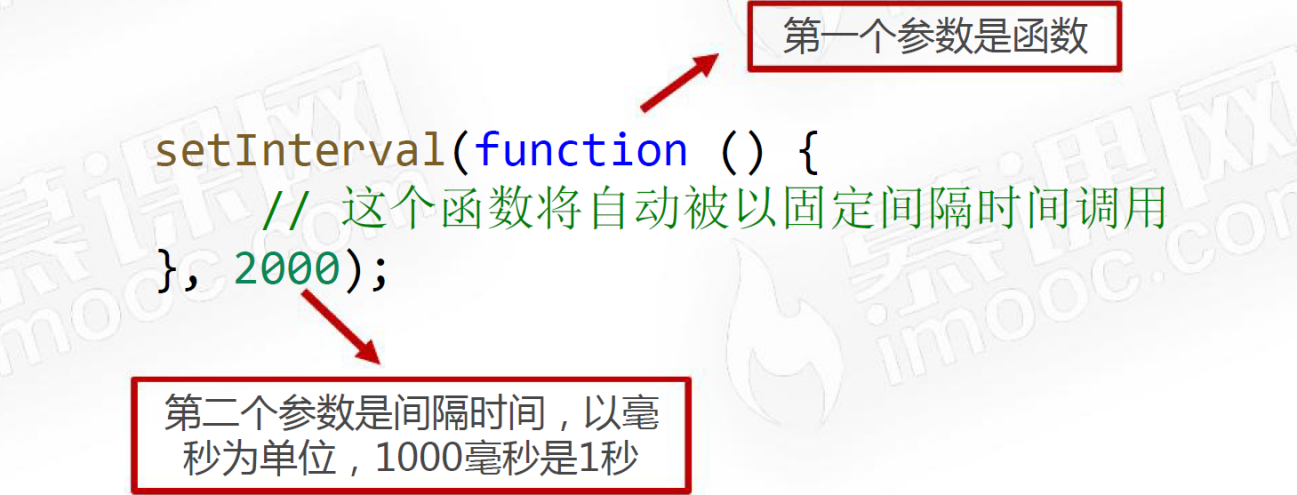
**事件委托**

**批量添加事件监听**

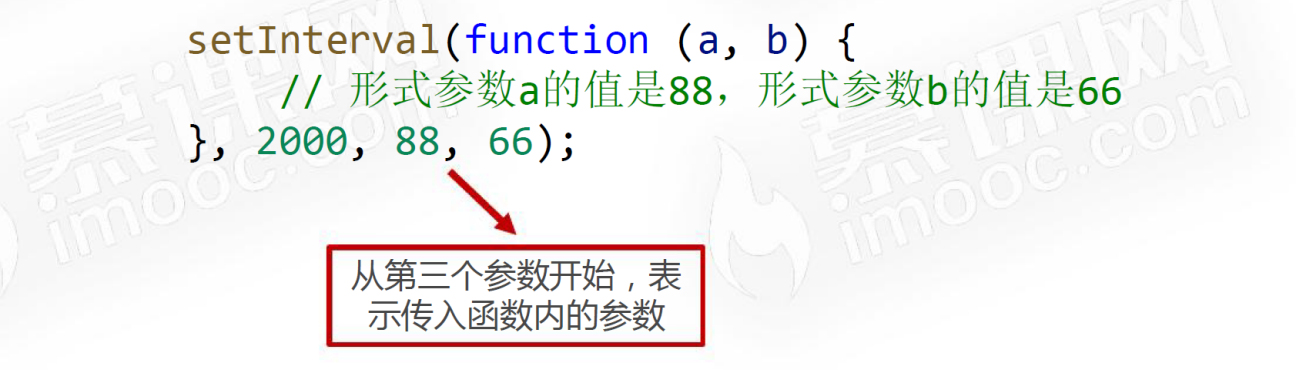
题目：页面上有一个无序列表 <ul>，它内部共有 20 个 <li> 元素，请批量给它们添加事件监听，实现效果：点击哪个 <li> 元素，哪个 <li> 元素就变红。

**定时器**

setInterval() 函数可以重复调用一个函数，在每次调用之间有固定的时间间隔。

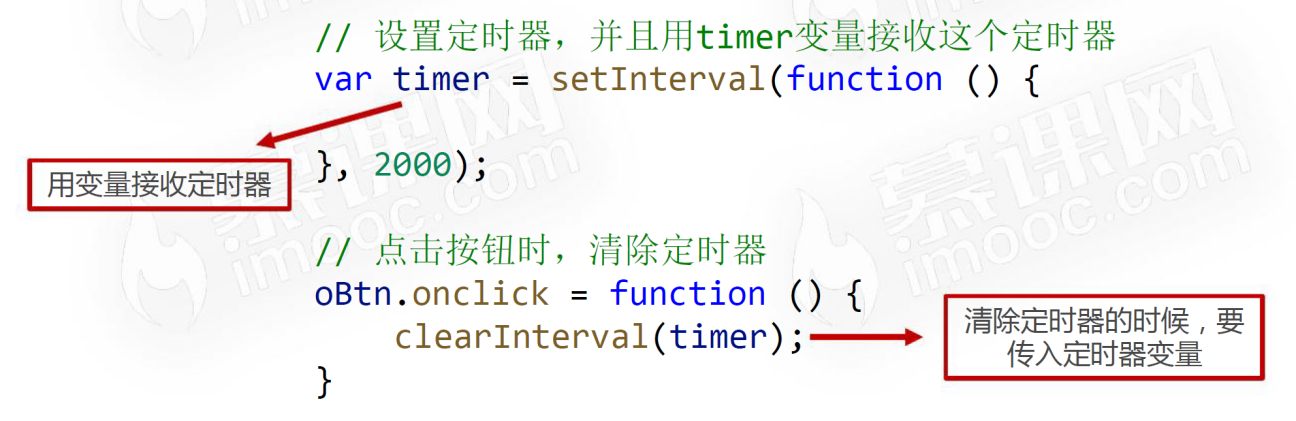


setInterval() 函数可以接收第 3、4、…… 个参数，它们将按顺序传入函数。



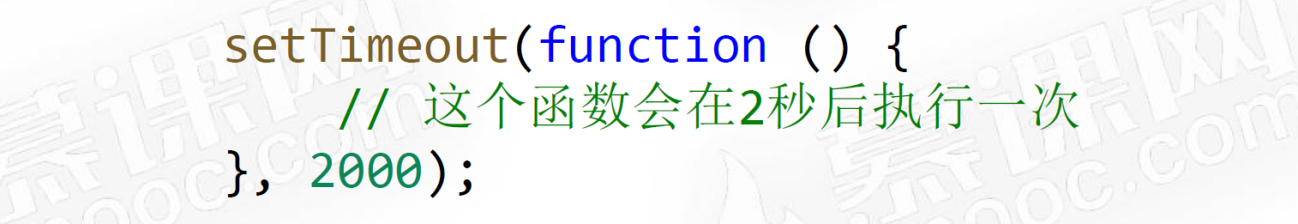
**清除定时器**

clearInterval() 函数可以清除一个定时器。



**延时器**

setTimeout() 函数可以设置一个延时器，当指定时间到了之后，会执行函数一次，不再重复执行。



**清除延时器**

clearTimeout() 函数可以清除延时器，和 clearInterval() 非常类似。

**BOM**

BOM基本概念

BOM（Browser Object Model，浏览器对象模型）是 JS 与 浏览器窗口交互的接口。

一些与浏览器改变尺寸、滚动条滚动相关的特效，都要借助 BOM 技术。

window对象

window 对象是当前 JS 脚本运行所处的窗口，而这个窗口中包含 DOM 结构，window.document 属性就是 document 对象。

注意！！全局变量是window的属性

var a = 10;

console.log(window.a == a); // true

这就意味着，多个 js 文件之间是共享全局作用域的

窗口尺寸相关属性

| **属性** | **意义** |
| --- | --- |
| innerHeight | 浏览器窗口的内容区域的高度，包含水平滚动条（如果有的话） |
| innerWidth | 浏览器窗口的内容区域的宽度，包含垂直滚动条（如果有的话） |
| outerHeight | 浏览器窗口的外部高度 |
| outerWidth | 浏览器窗口的外部宽度 |

获得不包含滚动条的窗口宽度，要用：

document.documentElement.clientWidth

**resize事件**

在窗口大小改变之后，就会触发 resize 事件，可以使用：

window.onresize 或者 window.addEventListener('resize') 来绑定事件处理函数。

**scroll事件**

在窗口被卷动之后，就会触发 scroll 事件，可以使用：

window.onscroll 或者 window.addEventListener('scroll') 来绑定事件处理函数。

**History对象**

window.history 对象提供了操作**浏览器会话历史的接口**。

常用操作就是模拟浏览器回退按钮。

history.back(); // 等同于点击浏览器的回退按钮

history.go(-1); // 等同于 history.back();

**Location对象**

window.location 标识当前所在网址，可以通过给这个属性赋值命令浏览器进行页面跳转。

window.locaiton = 'http://www.imooc.com';

window.location.href = 'http://www.imooc.com';

**重新加载当前页面**

可以调用 location 的 reload 方法以重新加载当前页面，参数 true 表示强制从服务器强制加载。

window.location.reload(true);

**面向对象**

**一，认识对象**

对象（object）是 “键值对” 的集合，表示属性和值的映射关系。

var xiaoming = {

name: '小明',

age: 12,

sex: '男',

hobbies: ['足球', '编程']

};

* 属性名（键名，key）: 属性值（value）
* JS 中，大括号表示对象
* 最后的属性后面不加逗号（某些程序员也习惯悬挂逗号，因为再次追加属性会比较方便）
* {} 后加上分号（因为这是一个赋值语句）

**属性是否加引号**

如果对象的属性键名不符合 JS 标识符命名规范，则这个键名必须用引号包裹。

例如：'favorite-book': '舒克和贝塔' （有短横不符合命名规范）

**属性的访问**

可以用 “点语法” 访问对象中指定键的值。

xiaoming.name; // '小明'

xiaoming.age; // 12

xiaoming.hobbys; // ['足球', '游泳', '编程']

如果属性名不符合 JS 标识符命名规范，则必须用方括号的写法来访问。

xiaoming['favorite-book']; // '舒克和贝塔'

xiaoming['name']; // '小明'

注意！！方括号 [] 中只能是字符串类型！

任何对象的属性名都可以通过 [] 来访问，只要把属性名写为字符串的形式。

**属性的更改**

直接使用赋值运算符重新对某属性赋值即可更改属性。

var obj = {

a: 10

};

obj.a = 30;

obj.a++;

**属性的创建**

如果对象本身没有某个属性值，则用点语法赋值时，这个属性会被创建出来。

var obj = {

a: 10

};

obj.b = 40;

var obj = {

a: 10

};

obj['zjr-b'] = 40;

**属性的删除**

如果要删除某个对象的属性，需要使用 delete 操作符。

var obj = {

a: 1,

'zjr-b': 2

};

delete obj.a;

delete obj['zjr-b'];

**二，对象的方法**

认识方法

如果某个属性值是函数，则它也被称为对象的 “方法”。

var xiaoming = {

name: '小明',

age: 12,

sex: '男',

hobbys: ['足球', '游泳', '编程'],

'favorite-book': '舒克和贝塔',

// sayHello方法

sayHello: function () {

console.log('你好我是小明，今年12岁，我是个男生');

}

};

**方法的调用**

使用 “点语法” 可以调用对象的方法。

xiaoming.sayHello();

**对象的遍历**

和遍历数组类似，对象也可以被遍历，遍历对象需要使用 for...in... 循环。

使用 for...in... 循环可以遍历对象的每个键。

// k: 循环变量，它会依次成为对象的每一个键（字符串类型）

// obj: 要遍历的对象

for (var k in obj) {

console.log('属性' + k + '的值是' + obj[k]);

}

**复习基本类型值和引用类型值**

|  | **举例** | **当 var a = b 变量传值时** | **当用 == 比较时** | **当用 === 比较时** |
| --- | --- | --- | --- | --- |
| 基本类型值 | 数字、字符串、布尔、undefined、null | 内存中产生新的副本 | 比较值是否相等 | 类型相等的前提下，比较值相等 |
| 引用类型值 | 对象、数组等 | 内存中不产生新的副本，而是让新变量指向同一个对象（内存地址的副本） | 比较内存地址是否相同，即比较是否为同一对象 | 比较内存地址是否相同，即比较是否为同一对象 |

**对象的浅克隆（藕断丝连）**

复习什么是浅克隆：只克隆对象的 “表层”，如果对象的某些属性值又是引用类型值，则不进一步克隆它们，只是传递它们的引用。

使用 for...in... 循环即可实现对象的浅克隆。

【案例】

var obj1 = {

a: 1,

b: 2,

c: [44, 55, 66]

};

// var obj2 = obj1; 这不是克隆，千万不能这样！！！！

// 实现浅克隆

var obj2 = {};

for (var k in obj1) {

// 每遍历一个 k 属性，就给 obj2 也添加一个同名的 k 属性

// 值和 obj1 的 k 属性值相同

obj2[k] = obj1[k];

}

// 为什么叫浅克隆呢？比如 c 属性的值是引用类型值，那么本质上 obj1 和 obj2 的 c 属性是内存中的同一个数组，并没有被克隆分开。

obj1.c.push(77);

console.log(obj2.c); // obj2 的 c 属性这个数组也会被增加 77

console.log(obj1.c == obj2.c); // true，证明了数组是同一个对象

**对象的深克隆（藕断丝断）**

复习什么是深克隆：克隆对象的全貌，不论对象的属性值是否又是引用类型值，都能将它们实现克隆。也就是说每次遇到一个引用数据类型，就创造一个新的内存地址

和数组的深克隆类似，对象的深克隆需要使用递归。

面试时经常会考察深克隆算法，必须掌握。

【案例】

var obj1 = {

a: 1,

b: 2,

c: [33, 44, {

m: 55,

n: 66,

p: [77, 88]

}]

};

// 深克隆

function deepClone(o) {

// 要判断 o 是对象还是数组

if (Array.isArray(o)) {

// 数组

var result = [];

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

result.push(deepClone(o[i]));

}

} else if (typeof o == 'object') {

// 对象

var result = {};

for (var k in o) {

result[k] = deepClone(o[k]);

}

} else {

// 基本类型值

var result = o;

}

return result;

}

var obj2 = deepClone(obj1);

console.log(obj2);

console.log(obj1.c == obj2.c); // false

obj1.c.push(99);

console.log(obj2); // obj2 不变的，因为没有“藕断丝连”的现象

obj1.c[2].p.push(999);

console.log(obj2); // obj2 不变的，因为没有“藕断丝连”的现象

**认识上下文**

函数的上下文

函数中可以使用 this 关键字，它表示函数的上下文。

与中文中 “这” 类似，函数中的 this 具体指代什么必须通过调用函数时的 “前言后语” 来判断。

var xiaoming = {

nickname: '小明',

age: 12,

sayHello: function () {

console.log('我是' + this.nickname + '，我' + this.age + '岁了');

}

};

xiaoming.sayHello();

// 我是小明，我12岁了

var xiaoming = {

nickname: '小明',

age: 12,

sayHello: function () {

console.log('我是' + this.nickname + '，我' + this.age + '岁了');

}

};

var sayHello = xiaoming.sayHello; // 将函数“提”出来，单独存为变量

sayHello(); // 直接圆括号调用这个函数，而不是对象打点调用了

// 我是undefined，我undefined岁了

同函数的上下文由调用方式决定，一个函数，用不同的形式调用它，则函数的上下文不同。

宏观上可以把 “谁调用，上下文就是谁” 作为评判方法，如果没有明确的调用者，那么就是 Window。

**函数如果不调用，则不能确定函数的上下文**

规则1：对象打点调用它的方法函数，则函数的上下文是这个打点的对象。

规则2：圆括号直接调用函数，则函数的上下文是 window 对象。

规则3：数组（类数组对象）枚举出函数进行调用，上下文是这个数组（类数组对象）。

规则4：IIFE 中的函数，上下文是 window 对象。

规则5：定时器、延时器回调函数，上下文是 window 对象。

规则6：事件处理函数的上下文是绑定事件的 DOM 元素。

**call和apply**

能指定函数的上下文

将 xiaoming 变为 sum() 的上下文：

sum.call(xiaoming); 或 sum.apply(xiaoming);

语法：

函数.call(上下文);

函数.apply(上下文);

function sum() {

console.log(this.chinese + this.math + this.english);

}

var xiaoming = {

chinese: 80,

math: 95,

english: 93

};

sum.call(xiaoming); // 268

sum.apply(xiaoming); // 268

**call和apply的区别（参数形式不同）**

不传参时，call和apply没有区别

传参时，call要用逗号罗列，apply要写进数组

function sum(b1, b2) {

alert(this.c + this.m + this.e + b1 + b2);

}

var xiaoming = {

c: 80,

m: 95,

e: 93

};

sum.call(xiaoming, 5, 3); // 276 call 必须要用逗号罗列参数

sum.apply(xiaoming, [5, 3]); // 276 apply 必须要把参数写到数组中

**上下文规则总结**

| **规则** | **上下文** |
| --- | --- |
| 对象.函数() | 对象 |
| 函数() | window |
| 数组[下标]() | 数组 |
| IIFE | window |
| 定时器 | window |
| DOM事件处理函数 | 绑定 DOM 的元素 |
| call和apply | 任意指定 |
| 用new调用函数 | 秘密创建出的对象 |

**用new操作符调用函数**

现在，我们学习一种新的函数调用方式：new 函数()

**用new调用函数的四步走**

JS 规定，使用 new 操作符调用函数会进行 “四步走”：

1. 函数体内会自动创建出一个空白对象
2. 函数的上下文（this）会指向这个对象
3. 函数体内的语句会执行
4. 函数会自动返回上下文对象，即使函数没有 return 语句

如下：

function fun() {

this.a = 3;

this.b = 5;

}

var obj = new fun();

console.log(obj); // fun { a: 3, b: 5 }



**构造函数**

* 用 new 调用一个函数，这个函数就被称为 “构造函数”，任何函数都可以是构造函数，只需要用 new 调用它
* 顾名思义，构造函数用来 “构造新对象”，它内部的语句将为新对象添加若干属性和方法，完成对象的初始化
* 构造函数必须用 new 关键字调用，否则不能正常工作，正因如此，开发者约定构造函数命名时首字母要大写
* 构造函数中的 this 不是这个函数本身，而是在这个函数中新创建的空白对象！

// 书写规范：构造函数首字母大写

// 接收三个参数

function People(name, age, sex) {

// this上绑定三个参数的同名属性

this.name = name;

this.age = age;

this.sex = sex;

}

// 传入三个参数

var xiaoming = new People('小明', 12, '男');

var xiaohong = new People('小红', 10, '女');

var xiaogang = new People('小刚', 13, '男');

console.log(xiaoming); // People { name: '小明', age: 12, sex: '男' }

console.log(xiaohong); // People { name: '小红', age: 10, sex: '女' }

console.log(xiaogang); // People { name: '小刚', age: 13, sex: '男' }