JS

字符串

\ : 转 义字符

多行字符串用反引号表示:

```
`这是一个
多行
字符串`;
```

连接字符串

1

```
let name = '小明';
let age = 20;
let message = '你好, ' + name + ', 你今年' + age + '岁了!';
alert(message);
```

2

```
let name = '小明';
let age = 20;
let message = `你好, ${name}, 你今年${age}岁了!`;
alert(message);
```

toUpperCase

`toUpperCase()`把一个字符串全部变为大写:

```
let s = 'Hello';
s.toUpperCase(); // 返回'HELLO'
```

toLowerCase

`toLowerCase()`把一个字符串全部变为小写:

```
let s = 'Hello';
let lower = s.toLowerCase(); // 返回'hello'并赋值给变量lower
lower; // 'hello'
```

indexOf

`indexOf()`会搜索指定字符串出现的位置：

```
let s = 'hello, world';
s.indexOf('world'); // 返回7
s.indexOf('World'); // 没有找到指定的子串，返回-1
```

substring

`substring()`返回指定索引区间的子串：

```
let s = 'hello, world'
s.substring(0, 5); // 从索引0开始到5（不包括5），返回'hello'
s.substring(7); // 从索引7开始到结束，返回'world'
```

数组操作

- 直接给`Array`的`length`赋一个新的值会导致`Array`大小的变化：

  `arr.length = 6; console.log(arr); // arr变为['A', 'B', 'C', undefined, undefined, undefined]`

- `Array`可以通过索引把对应的元素修改为新的值，因此，对`Array`的索引进行赋值会直接修改这个`Array`：

  `let arr = ['A', 'B', 'C']; arr[1] = 99; console.log(arr); // arr现在变为['A', 99, 'C'] console.log(arr[1]); // 99`

- **indexOf**

  ```
  let arr = [10, 20, '30', 'xyz'];
  arr.indexOf(10); // 元素10的索引为0
  ```

- **slice**

  截取`Array`的部分元素，然后返回一个新的`Array`：

  ```
  let arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
  arr.slice(0, 3); // 从索引0开始，到索引3结束，但不包括索引3: ['A', 'B', 'C']
  arr.slice(3); // 从索引3开始到结束: ['D', 'E', 'F', 'G']
  ```

xx.slice() : 截取所有元素

- **push和pop**

  `push()`向`Array`的末尾添加若干元素，`pop()`则把`Array`的最后一个元素删除掉

- **unshift和shift**

  往`Array`的头部添加若干元素，使用`unshift()`方法，`shift()`方法则把`Array`的第一个元素删掉

- **sort**

  `sort()`可以对当前`Array`进行排序

- **reverse**

  `reverse()`把整个`Array`的元素反转

- **splice**

  从指定的索引开始删除若干元素，然后再从该位置添加若干元素：

```
let arr = ['Microsoft', 'Apple', 'Yahoo', 'AOL', 'Excite', 'Oracle'];
// 从索引2开始删除3个元素,然后再添加两个元素:
arr.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素 ['Yahoo', 'AOL',
'Excite']
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
// 只删除,不添加:
arr.splice(2, 2); // ['Google', 'Facebook']
arr; // ['Microsoft', 'Apple', 'Oracle']
// 只添加,不删除:
arr.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
```

- **concat**

  `concat()`方法把当前的`Array`和另一个`Array`连接起来，并返回一个新的`Array`：

```
let arr = ['A', 'B', 'C'];
arr.concat(1, 2, [3, 4]); // ['A', 'B', 'C', 1, 2, 3, 4]
```

- **join**

  把当前`Array`的每个元素都用指定的字符串连接起来，然后返回连接后的字符串：

```
let arr = ['A', 'B', 'C', 1, 2, 3];
arr.join('-'); // 'A-B-C-1-2-3'
```

对象

JavaScript用一个{...}表示一个对象，键值对以xxx: xxx形式申明，用,隔开

```
let xiaoming = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
};
```

```
xiaoming.name; // '小明'
```

访问属性是通过.操作符完成的，但这要求属性名必须是一个有效的变量名。如果属性名包含特殊字符，就必须用''括起来

```
'middle-school': 'No.1 Middle School'
```

```
xiaohong['middle-school']; // 'No.1 Middle School'
```

检测xiaoming是否拥有某一属性，可以用in操作符：

```
'name' in xiaoming; // true
```

要判断一个属性是否是xiaoming自身拥有的，而不是继承得到的，可以用hasOwnProperty()方法：

```
let xiaoming = {
    name: '小明'
};
xiaoming.hasOwnProperty('name'); // true
xiaoming.hasOwnProperty('toString'); // false
```

条件判断

JavaScript把null、undefined、0、NaN和空字符串''视为false，其他值一概视为true

循环

for

for循环最常用的地方是利用索引来遍历数组：

```
let arr = ['Apple', 'Google', 'Microsoft'];
let i, x;
for (i=0; i<arr.length; i++) {
    x = arr[i];
    console.log(x);
}
```

for循环的3个条件都是可以省略的，如果没有退出循环的判断条件，就必须使用break语句退出循环，否则就是死循环：

```
let x = 0;
for (;;) { // 将无限循环下去
    if (x > 100) {
        break; // 通过if判断来退出循环
    }
    x ++;
}
```

for循环的一个变体是for ... in循环，它可以把一个对象的所有属性依次循环出来：

```
let o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (let key in o) {
    console.log(key); // 'name', 'age', 'city'
}
```

过滤掉对象继承的属性

```
for (let key in o) {
    if (o.hasOwnProperty(key)) {
        console.log(key); // 'name', 'age', 'city'
    }
}
```

```
for ... in`对`Array`的循环得到的是`String`而不是`Number
```

```
let a = ['A', 'B', 'C'];
for (let i in a) {
    console.log(i); // '0', '1', '2'
    console.log(a[i]); // 'A', 'B', 'C'
}
```

while

while循环只有一个判断条件，条件满足，就不断循环，条件不满足时则退出循环。比如我们要计算100以内所有奇数之和，可以用while循环实现：

```
let x = 0;
let n = 99;
while (n > 0) {
    x = x + n;
    n = n - 2;
}
```

do ... while

最后一种循环是do { ... } while()循环，它和while循环的唯一区别在于，不是在每次循环开始的时候判断条件，而是在每次循环完成的时候判断条件：

```
let n = 0;
do {
    n = n + 1;
} while (n < 100);
n; // 100
```

Map

Map是一组键值对的结构，具有极快的查找速度。

JavaScript写一个Map如下：

```
let m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);
m.get('Michael'); // 95
```

初始化Map需要一个二维数组，或者直接初始化一个空Map。Map具有以下方法：

```
let m = new Map(); // 空Map
m.set('Adam', 67); // 添加新的key-value
```

```
m.set('Bob', 59);
m.has('Adam'); // 是否存在key 'Adam': true
m.get('Adam'); // 67
m.delete('Adam'); // 删除key 'Adam'
m.get('Adam'); // undefined
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
let m = new Map();
m.set('Adam', 67);
m.set('Adam', 88);
m.get('Adam'); // 88
```

Set

Set和Map类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在Set中，没有重复的key。

要创建一个Set，需要提供一个Array作为输入，或者直接创建一个空Set：

```
let s1 = new Set(); // 空Set
let s2 = new Set([1, 2, 3]); // 含1, 2, 3
```

重复元素在Set中自动被过滤：

```
let s = new Set([1, 2, 3, 3, '3']);
s; // Set {1, 2, 3, "3"}
```

注意数字3和字符串'3'是不同的元素。

通过add(key)方法可以添加元素到Set中，可以重复添加，但不会有效果：

```
s.add(4);
s; // Set {1, 2, 3, 4}
s.add(4);
s; // 仍然是 Set {1, 2, 3, 4}
```

通过delete(key)方法可以删除元素：

```
let s = new Set([1, 2, 3]);
s; // Set {1, 2, 3}
s.delete(3);
s; // Set {1, 2}
```

iterable

用`for ... of`循环遍历集合，用法如下：

```
let a = ['A', 'B', 'C'];
let s = new Set(['A', 'B', 'C']);
let m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
for (let x of a) { // 遍历Array
    console.log(x);
}
for (let x of s) { // 遍历Set
    console.log(x);
}
for (let x of m) { // 遍历Map
    console.log(x[0] + '=' + x[1]);
}
```

`forEach()`方法

`Set`与`Array`类似，但`Set`没有索引，因此回调函数的前两个参数都是元素本身：

```
let s = new Set(['A', 'B', 'C']);
s.forEach(function (element, sameElement, set) {
    console.log(element);
});
```

`Map`的回调函数参数依次为`value`、`key`和`map`本身：

```
let m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
m.forEach(function (value, key, map) {
    console.log(value);
});
```

如果对某些参数不感兴趣，由于JavaScript的函数调用不要求参数必须一致，因此可以忽略它们。例如，只需要获得`Array`的`element`：

```
let a = ['A', 'B', 'C'];
a.forEach(function (element) {
    console.log(element);
});
```

函数定义和调用

```
function abs(x) {}
```

- function指出这是一个函数定义；
- abs是函数的名称；
- (x)括号内列出函数的参数，多个参数以,分隔；
- { ... }之间的代码是函数体，可以包含若干语句，甚至可以没有任何语句。

第二种定义函数的方式如下：

```
let abs = function (x) {};
```

注意第二种方式按照完整语法需要在函数体末尾加一个;，表示赋值语句结束。

arguments，获得调用者传入的所有参数

rest参数，函数可以写为：

```
function foo(a, b, ...rest) {
    console.log('a = ' + a);
    console.log('b = ' + b);
    console.log(rest);
}

foo(1, 2, 3, 4, 5);
// 结果:
// a = 1
// b = 2
// Array [ 3, 4, 5 ]

foo(1);
// 结果:
// a = 1
// b = undefined
// Array []
```

变量作用域与解构赋值

JavaScript的函数在查找变量时从自身函数定义开始，从“内”向“外”查找。如果内部函数定义了与外部函数重名的变量，则内部函数的变量将“屏蔽”外部函数的变量。

变量提升

JavaScript的函数定义有个特点，它会先扫描整个函数体的语句，把所有用var申明的变量“提升”到函数顶部

全局作用域

名字空间

全局变量会绑定到window上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:
let MYAPP = {};

// 其他变量:
MYAPP.name = 'myapp';
MYAPP.version = 1.0;

// 其他函数:
MYAPP.foo = function () {
    return 'foo';
};
```

把自己的代码全部放入唯一的名字空间MYAPP中，会大大减少全局变量冲突的可能。

许多著名的JavaScript库都是这么干的：jQuery，YUI，underscore等等。

局部作用域

let替代var可以申明一个块级作用域的变量

常量

const来定义常量，const与let都具有块级作用域：

```
const PI = 3.14;
```

解构赋值

如果数组本身还有嵌套，也可以通过下面的形式进行解构赋值，注意嵌套层次和位置要保持一致：

```
let [x, [y, z]] = ['hello', ['JavaScript', 'ES6']];
x; // 'hello'
y; // 'JavaScript'
z; // 'ES6'
```

解构赋值还可以忽略某些元素：

```
let [, , z] = ['hello', 'JavaScript', 'ES6']; // 忽略前两个元素，只对z赋值第三个元素
z; // 'ES6'
```

```
let person = {
    name: '小明',
    age: 20,
    gender: 'male',
    passport: 'G-12345678',
    school: 'No.4 middle school'
};
let {name, age, passport} = person;

// name, age, passport分别被赋值为对应属性:
console.log(`name = ${name}, age = ${age}, passport = ${passport}`);
```

排序以及多个属性数组对象排序（按条件排序）

原生排序

```
let arr = [5,2,1,4,9,8]
for(let i = 0 ; i < arr.length ; i ++) {
             for(let j = 0 ; j < arr.length -1 ; j ++) {
                 if(arr[j] > arr[j+1]){
                     let num = arr[j]
                     arr[j] = arr[j+1]
                     arr[j+1] = num
                     comeout.innerText = arr
                 }
             }
       // 结果 1,2,4,5,8,9
```

ES6排序

sort() 方法是最强大的数组方法之一。 默认排序顺序为按字母升序。 使用数字排序，你必须通过一个函数作为参数来调用。 比较函数两个参数a和b，a-b 升序，返回b-a 降序 **注意： 这种方法会改变原始数组!**

```
// 升序
 arr.sort(function(a,b){
                 return a - b
             })
          console.log(arr)
// 结果 1,2,4,5,8,9


// 降序
arr.sort(function(a,b){
                 return b - a
             })
```

```
            console.log(arr)
// 结果 9,8,5,4,2,1
```

根据数组中的某个属性排序

```
let arr_choice = [{id:1},{id:3},{id:2},{id:8},{id:6},{id:4}]
// 升序
arr_choice.sort(function(a,b){
                return b.id - a.id
            })
           console.log(arr_choice)
// 结果: [{"id":1},{"id":2},{"id":3},{"id":4},{"id":6},{"id":8}]


// 降序
arr_choice.sort(function(a,b){
                return b.id - a.id
            })
          console.log(arr_choice)
// 结果: [{"id":8},{"id":6},{"id":4},{"id":3},{"id":2},{"id":1}]
```

根据多个属性排序

```
let arr_multi = [{id:1,age:10},{id:3,age:5},{id:2,age:6},{id:8,age:8},
{id:6,age:5},{id:4,age:5}]
// 升序
arr_multi.sort(function(a,b){
                // 默认根据年龄排序，年龄相同则按照id排序
                if(a.age==b.age){
                    return a.id - b.id
                }
                return a.age - b.age
            })
      console.log(arr_multi)
// 结果: [{"id":3,"age":5},{"id":4,"age":5},{"id":6,"age":5},{"id":2,"age":6},
{"id":8,"age":8},{"id":1,"age":10}]


// 降序
 arr_multi.sort(function(a,b){
                // 默认根据年龄排序，相同则按照id排序
                if(a.age==b.age){
                    return b.id - a.id
                }
                return b.age - a.age
            })
            console.log(arr_multi)
```

```
// 结果: [{"id":1,"age":10},{"id":8,"age":8},{"id":2,"age":6},{"id":6,"age":5},
{"id":4,"age":5},{"
```

扩展操作符（...）可在函数调用/数组构造时, 将数组表达式或者string在语法层面展开； 还以在构造对象时, 将对象表达式按key-value的方式展开；