Javascript是由ECMAScript规格的具体实现。每一年都会有新的ECMAScript规范，以及新的特性。

2015年为ECMA2015, 也就是ES2015，也可以称为ES6（比年代+1）.

2016年ECMA2016, 也就是ES2016, 也可以称为ES7.

2017年ECMA2017, 也就是ES2017, 也可以称为ES8.

2018年ECMA2018, 也就是ES2018, 也可以称为ES9.

2019年ECMA2019, 也就是ES2019, 也可以称为ES10.

2020年ECMA2020, 也就是ES2020, 也可以称为ES11.

🡪

JavaScript是客户端语言，一般在浏览器中运行。通过Node.js的发明，使得Javascript可以在后端运行。

🡪

你无法确定一个方法函数中 this 的指向，因为它的值跟函数的调用方式，调用者有关，而跟如何定义函数无关。除非你使用箭头函数，因为箭头函数中this的值是继承自外围作用域。

判断函数执行时，具体this指向的思路：

--函数执行，首先看函数名前面是否有“.” ，有的话， “.”前面是谁，this就是谁；没有的话this就是window

--给元素的某一事件绑定方法，当事件触发的时候，执行对应的方法，方法中的this是当前元素

箭头函数=>的语法，实现在ES6标准中。箭头函数=>的语法相当于创建了一个this不会改变的**匿名函数**。

箭头函数和一般的函数不同，箭头函数不会绑定this。 或者说箭头函数不会改变this本来的绑定. 调用箭头函数的时候，this便是调用前原来的this对象。

函数需要参数：

(参数1，参数2) => {函数体}

函数不需要参数:

() => {函数体}

另外箭头函数还可以更加的简洁：

函数只需要一个参数， 那么参数地方不需要括号：

参数1=> {函数体}

箭头函数（fat arrow function）是否存在返回值？ 首先=>后是否有花括号。

如果没有任何符号，则返回整体，常用于React中。

如果是() 则返回 () 中的所有内容。

如果是花括号则表达是方法体申明，此时此方法提返回的内容是花括号方法体中,return关键字后的值。如果没有return关键字,则不返回。

🡪

#### Javascript的三元运算符

语法：条件?代码块1：代码块2

如果条件为真则执行代码块1，反之执行代码块2.

例如：let test = 1>3? 1:3;

则test为3

🡪

Javascript的解构引入语法：

import React, {Component} from ‘react’🡪意思是从react中引入React, 以及**React中的**Component.以后便可以直接在使用Component.

🡪

ECMAScript 和 JavaScript 的关系是，**前者是后者的规格，后者是前者的一种实现**。

javascript是netscape创造的并交给了国际标准化组织ECMA来制定标准化过程。

ES6是ECMA的为JavaScript制定的第6个版本的标准，标准委员会最终决定，标准在每年的 6 月份正式发布一次，作为当年的正式版本。

**ECMAscript 2015 是在2015年6月份发布的ES6的第一个版本**。依次类推：

ECMAscript 2016 对于ES7.

ECMAscript 2017 对于ES8.

ECMAscript 2018 对于ES9.

ECMAscript 2019 对于ES10.

ECMAscript 2020 对于ES11.

ES6增加的例子有：

|  |
| --- |
| 新增关键字let：let只在代码块内有效，var在全局范围内有效。let只能声明一次，var可以声明多次。  **let关键字不存在变量提升，var关键字存在变量提升**。  例如：  console.log(a); //ReferenceError: a is not defined  let a = "apple";  console.log(b); //undefined  var b = "banana";  变量 b 用 var 声明存在变量提升，所以当脚本开始运行的时候，b 已经存在了，但是还没有赋值，所以会输出 undefined。  变量 a 用 let 声明不存在变量提升，在声明变量 a 之前，a 不存在，所以会报错。  Undefined和ReferenceError: x is not defined不一样，undefined是申明了但没有赋值具体的数值，值就是undefined, 而没有申明过直接调用则是not defined, 是错误。 |
| const 声明一个只读变量，声明之后不允许改变，不允许赋值或者重新声明。意味着，一旦声明必须初始化，否则会报错。 |
| 使用语法糖class来定义类。  不需要经历构造函数，然后通过对构造函数.prototype所指向的原型添加属性的方式构建类。 |
| 提供箭头函数=》 |
| 模板字符串语法，在**反单引号**中，以${varname}的形式插入变量  例如：  let str=” world”;  console.log (`Hello ${str}`) // 输出Hellow World |
| 扩展语法  扩展操作符 …  使用扩展操作符…, 用于将数组或者字典打开并且复制.  **…与Object.Assign方法执行的都是浅拷贝，意味着只有第一层会被复制。如果在数据结构内部还有引用类型数据，那么将不会被拷贝，而是采用相同引用。**  例子1:  var mid = [3, 4];  var arr = [1, 2, ...mid, 5, 6]; // 通过扩展操作符，mid数组将被打开成为3,4元素插入到新的数组当中  例子2:  var arr = ['a', 'b', 'c'];  var arr2 = [...arr]; // **通过扩展操作符，复制出arr2和arr有同样内容。 这样就算arr2改变，arr也不会改变。如果通过arr2 = arr,那样则会因为是同一个引用地址而同时改变。**  arr2.push('d');  例子3:  var str = "hello";  var chars = [...str]; // 通过扩展操作符， 将字符串hello拆分为数组 |

🡪

Javascript定义函数：

Javascript为动态类型语言，申明函数的时候不需要指定函数的类型

function myFunc (name, weather) {

…

}

Javascript不像Java,C#那样支持多态，Javascript不支持多态(polymorphism)。函数名一样， 不同函数参数不会因此产生多态效果，只会让后定义的函数覆盖先定义的函数。

**调用Javascript函数的时候，不需要和定义函数时需要的参数个数相同。**

**如果调用时传入的函数少于定义，则剩下的参数为undefined.**

**function** myFunc(name, weather) {  
 console.log("Hello " + name + ".");  
 console.log("It is " + weather + " today.");  
}  
myFunc("Adam"); #执行正常。只是未被传值的变量weather将是undefined.

**如果调用时传入的参数多于定义，则多余的参数会被放入arguments保留关键字数组中，或者如果定义...extraArgs，则多余参数将放入这个数组当中。**

**function** myFunc(name, weather) {  
 console.log("Hello " + name + ".");  
 console.log("It is " + weather + " today.");  
}  
myFunc("Adam", "sunny", "GGG", "AAA"); #执行正常，没有错误

多余的参数可以用...extraArgs数组来存放。

**function** myFunc(name, weather, ...extraArgs) {  
 console.log("Hello " + name + ".");  
 console.log("It is " + weather + " today.");  
 **for** (**let** i=0; i<extraArgs.length; i++){  
 console.log(i);  
 }  
}  
myFunc("Adam", "sunny", "one", "two", "three"); #执行正常，多余参数在extraArgs中

Javascript支持参数的默认值。

**function** myFunc(name, weather="Sunny") {  
 console.log("Hello " + name + ".");  
 console.log("It is " + weather + " today.");  
}  
myFunc("Adam"); #执行正常。 It is Sunny today.

Javascript将函数看作是对象，所以可以将函数作为参数进行传递，调用。这也是回调函数的根本原理。

**function** myFunc(nameFunction){  
 **return** ("Hello " + nameFunction() + ".")  
}  
  
**function** printName(nameFunction, printFunction){  
 printFunction(myFunc(nameFunction))  
}  
  
printName(**function**() {**return** "Adam"}, console.log); #输出为Hello Adam.

Javascript函数中的闭包(closure)。意思是Javascript在函数中定义的另一个函数，可以获取外层函数的本地变量。需要注意的是，**内层函数如果需要使用counter, index这种计数变量的时候，不要和外层的计数变量混在一起。**

**function** myFunc(name) {  
 **let** myLocalVar = "sunny";  
 **let** innerFunction = **function** () {  
 **return** ("Hello " + name +". Today is " + myLocalVar + ".");  
 };  
 **return** innerFunction();  
}  
console.log(myFunc("Adam")); # 内层函数可以获取外层函数的本地变量, name, myLocalVar

Javascript函数的定义方法有两种, 一种是比较熟悉的function myFunc(parameter){…}形式，在定义函数时候给与函数名。另外一种是const myFunc2 = function(parameter){…}的方式，相当于创建匿名函数然后把函数的引用赋值给一个变量作为函数名。

**function** myFunc(name) {  
 console.log ("Hello " + name)  
}  
**const** myFunc1 = **function**(name) {  
 console.log ("Hello " + name)  
};  
  
myFunc("A");  
myFunc1("B");

🡪

Javascript的六种**基本**数据类型：

Number: 数字类型

String: 字符串类型， Javascript中的字符串由Unicode表示

Boolean: 布尔类型。（0, 空字符串，NaN, null, undefined都认为是false）

Symbol

undefined: 数据已经定义但是没有赋值

null: 数据定义并且赋值，值为null表示虚无

其余的数据类型都统称为 object 数据类型，其中，包括 Array, Date, Function等

字符串类型常用自带函数：

|  |  |
| --- | --- |
| .length | 返回字符串的长度 |
| .charAt(index) | 返回在index位置的字母 |
| .concat(string) | 将字符串与传入的字符串拼接 |
| .indexOf(term, start) | 返回查找字符串的首先出现的位置，如果没有则为-1 |
| ,replace(term, newTerm) | 将字符串中的term换为newTerm.  其中term可以为字符串常量，也可以为正则表达式 |
| .slice(start, end) | 将字符串从指定开始位置截取到结束位置 |
| .split(term) | 将字符串根据term字符进行分割 |
| toUpperCase() | 将字符串全部变为大写 |
| toLowerCase() | 将字符串全部变为小写 |
| trim() | 将字符串中的开头前，结尾后的空白字符移除 |
| `${变量名}` | ES6的特性，字符串模板，将变量插入字符串的内容中 |

🡪

== 与 === 的区别。当javascript进行==比较的时候，Javascript会将变量的类型变为一致，然后比较他们的值是否相同。而如果使用===进行比较，Javascript不会进行任何转换，而是直接比较。如果相同，则需要类型，和值都相同。

🡪

Javascript在运算过程中会自动转换数据类型。有时候并不是我们需要的。

例如, Javascript中 “+” 的字符串拼接比运算加号的优先级高所以

let myData2 = 5 + “5” #将会变为55， 而不是数字类型10.这是Javascript自动转换的结果。如果我们需要变为数字10，则需要手动转换为let myData2 = 5 + parseInt (“5”)

数字转为字符串(Number to String)

|  |  |
| --- | --- |
| toString() | 返回数字在十进制下的字符串 |
| toString(2) | 返回数字在二进制下的字符串 |
| toString(8) | 返回数字在八进制下的字符串 |
| toString(16) | 返回数字在十六进制下的字符串 |
| toFixed(n) | 返回数字字符串表示，带有n位小数点 |
| toExponential(n) | 返回数字的指数型表示，带有n位小数点 |

字符串转为数字(String to Number)

|  |  |
| --- | --- |
| Number(str) | 将字符串变为整形数字或者浮点数 |
| parseInt(str, 进制) | 将字符串变为整形数字 |
| parseFloat(str) | 将字符串变为整数变为浮点数，总是转化为10进制 |

🡪

Javascript创建数组的两种方式：

let myArray = new Array (); #以构建函数的方式创建数组

let myArray = [100, “Adam”, true] # 以普通变量的方式构建数组

Javascript的数组可以**自动扩展数组长度**，并且可以存放任何数据类型到同一数组当中。

Javascript数组Built-in方法：

Immutable方法：该方法执行在数组上时，**不会改变**原数组结构和值。而是将改变以后的结果直接返回。React对state中的状态进行改动的时候，用immutable方法，最后直接this.setState()进行更新。

Mutable方法：该方法执行在数组上时，会改变原数组结构和值。

|  |  |
| --- | --- |
| .concat(otherArray)  (immutable方法) | 将原数组中的元素与参数提供的数组中的元素相拼接形成新数组，然后返回新数组，不改变原数组。 |
| .join(separator) | 将数组中的元素使用separator连接起来成为字符串并且返回 |
| .pop() | 移除数组中的最后一个元素，并且返回 |
| .shift() | 移除数组中的第一个元素，并且返回 |
| .push(item)  **（mutable方法）** | 将元素放入数组的结尾 |
| .unshift(item)  **（mutable方法）** | 将元素放入数组的开头 |
| .reverse()  **（mutable方法）** | 将数组倒序并返回。该方法会直接改变数组，而不是返回另一新的数组。 |
| .slice(start, end)  **(immutable方法)** | 将数组进行切割，从start位置到end位置.返回切割后的数组，不改变原数组。直接array.slice()则复制相当于复制一份数组。 |
| .sort()  **（mutable方法）** | 将数组进行排序，可以传入自定义的比较方法.  如果没有自定义的比较方法，将按字母顺序对数组中的元素进行排序，说得更精确点，是按照字符编码的顺序进行排序。要实现这一点，首先应把数组的元素都转换成字符串（如有必要），以便进行比较。  如果想按照其他标准进行排序，就需要提供比较函数，该函数要比较两个值，然后返回一个用于说明这两个值的相对顺序的数字。比较函数应该具有两个参数 a 和 b，其返回值如下：   * 若 a 小于 b，在排序后的数组中 a 应该出现在 b 之前，则返回一个小于 0 的值。 * 若 a 等于 b，则返回 0。 * 若 a 大于 b，则返回一个大于 0 的值 |
| .splice(index, count)  **(mutable方法)** | 将数组从index位置开始，移除count个元素，并且将移除的元素返回。移除后数组自动将后方元素补齐移除元素的空缺。 |
| .fill(value, start, end) | (ES6) 将一个固定的数组value, 从数组的start下标开始，到end下标结束, 不包括结束位置的下标，替换原来数组的元素。 |
| .every(test) | 对数组中每个元素运行test方法，如果数组中所有元素能让test函数都为true, 则返回true,否则为false |
| .some(test) | 对数组中每个元素运行test方法，如果数组中任一元素能让test函数都为true, 则返回true,否则为false |
| .filter(test)  **(immutable方法)** | 对数组中每个元素运行test方法，保留能让test函数为true的元素并且以新数组的形式返回。 |
| .find(test) | 返回第一个让test函数为true的元素 |
| .findIndex(test) | 返回第一个让test函数为true的元素下标 |
| .forEach(callback) | 对数组中的每个元素调用callback函数 |
| .map(callback)  **（Immmutable方法）** | 对数组中的每个元素调用callback函数，并且将每个元素运行后的结果放入新的数组中返回。  Callback函数第一个参数为元素本身。  第二个元素为index。  第三个元素为原数组。 |
| .reduce(callback) | 对数组中的每个元素调用callback函数，获取累计的总计数值.  Callback要求两个参数，第一个参数是在轮训中保持的数值，例如总数。第二个参数是每一个数组中的元素。 |
| .indexOf(element) | 判断element是否存在于数组当中，如果存在返回第一个元素的下标。如果不存在返回-1. （注意，直接像Python那样使用.. in 数组是不正确的）  例如：  let test = [“A,” B”];  1 in test //为true, 原因是数组在JS中为对象，赋值的时候相当于是下标0, 下标1作为数组对象的key, “A”, “B” 作为数组对象的值。所以1 in test, key=1是该对象的属性之一，所以也为true. 要判断元素是否存在数组当中为test.indexOf(1) |

🡪

创建Object的两种方式：

let myObj = new Object() #以构建函数的方式创建对象

let myObj = { #以普通变量的方式构建对象

“xx”: xxx,

xx: function () {

}

}

**P.s**

{1,2,3} 🡪 非法, 因为对象以key, value形式存在

{a,b,c} **->** 如果a,b,c是普通变量，也是非法。因为无法变为key, value形式存在

def a () {

}

def b () {

}

def c () {

}

{a,b,c} 🡪 合法，将会变为{“a”:function {}, “b”: function {}, “c”: function {}}

🡪

Javascript的Class中，constructor方法用于构建类的实例，它将会在每个类的实例化时自动调用。

🡪

Javascript的for循环四种形式：

第一种可以根据自己设置的次数，下标进行循环。

for (var i = 0; i < 5; i++) {

// Will execute 5 times

}

第二种可以对对象中的**属性**进行循环。对于数组则是下标，对于字典则是keys

for (let property in object) {

// do something with object property

}

第三种对可迭代对象中每个**属性值**进行循环。

for (let value of object) {

// do something with object value

}

第四种使用数组的.forEach方法循环

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

除此以外，Javascript还有while循环，do-while循环。

🡪

Javascript Switch语句：

switch (action) {

case 'draw':

drawIt();

break; // break语句当遇到break语句不会执行余下的条件，如果没有则继续往下执行。

case 'eat':

eatIt();

break;

default: //如果上方没有条件满足，或者满足后没有break则执行default中的语句

doNothing();

}

🡪

var obj = {

name: 'Carrot',

details: {

color: 'orange',

size: 12

}

};

读取Javascript对象中的属性值时，有两种方法。

第一种：Dot notation方式。例如 obj.name值为Carrot。使用Dot notation读取对象中的属性值时，无法使用变量读取。例如obj.name将获取obj对象属性名为name的属性值，而不是获取obj对象变量name的值，所对应的obj对象属性值

第二种bracket notation，obj[“name”] 值为Carrot.在bracket **notation中还可以使用变量读取，test = “name”, obj[test]值为Carrot.**

🡪

Javascript匿名函数。

var avg = **function () {/**/ 申明函数的时候，不写函数名称，申明函数需要的参数。在需要传入函数的地方，可以直接写入匿名函数。也可以申明匿名函数，然后作为函数引用传递给某个变量

**var sum = 0;**

**for (var i = 0, j = arguments.length; i < j; i++) {**

**sum += arguments[i];**

**}**

**return sum / arguments.length;**

**};**

🡪

Javascript的Object.assign() 方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

语法：

Object.assign(target, source1, source2, …)

例子：

let json1 = {a:1}

let json2 = {b:2}

let json3 = {c:3}

let json = Object.assign ({}, json1, json2, json3);

console.log(json);

🡪

Javascript类的实例方法, 实例属性和静态方法，静态属性。

**静态方法：**在类中如果在一个方法前， 加上static关键字， 就表示该方法静态方法。静态方法不会被实例继承，但可以被子类继承。 静态方法直接通过类来调用。 如果在实例上调用静态方法， 会抛出一个错误， 表示不存在该方法。

静态属性指的是 Class 本身的属性， 即Class.propname， 而不是定义在实例对象（ this） 上的属性。

🡪

var x = [{"score":1}, {"score":2}, {"score":3}]

console.log(x);

x.reverse();

console.log(x);

Javascript的输出将会是

[{"score":3}, {"score":2}, {“score”:1]

[{"score":3}, {"score":2}, {“score”:1]

Javascript并没有首先输出原顺序的数组然后再输出reverse () 以后的数组。这是因为console.log会判断输出的对象是否是可以变的, 如果输出的上下文在短时间内发生变化，那么console.log将会输出变化后的值。所以要达到目的需要

var x = [{"score":1}, {"score":2}, {"score":3}]

console.log (Object.assign([], x));

x.reverse();

console.log(x);

🡪

Javascript的数组是对象，数组的索引是其数组对象的属性。所以数组[1] = “AAA”, 数组[2] = “BBB”赋值语句的时候，Javascript为其数组对象添加了成员属性1,2,所以可以通过下标数组[1],数组[2]读取。但是不能像Python一样进行数组[-1]获取最后一个元素，因为-1在数组对象中并没有-1这个对象属性成员。

🡪

Javascript的解构赋值（ES6提出, Destructuring assignment）

解构赋值对数组以及对象都有效。并且解构赋值相当于拆包以后赋值给变量。而扩展符…（spread operator）只是打开数据。

对于数组：

var foo = ["one", "two", "three"];

var [one, two, three] = foo;

//JS解构赋值的时候，需要[]表示来满足原来的数据结构，依次赋值

//在Python中可以a, b, c = foo

console.log(one) //输出”one”

对于对象：

let options = {

title: "Menu",

width: 100,

height: 200

}

let { width, title, height } = options // 改变左边顺序也不会影响，以{}表示满足于原来的数据结构，width表示获取对象中属性为width的值并且赋值到变量名为 width的变量当中

console.log(title); // Menu

console.log(width); // 100

console.log(height); // 200

🡪

Javascript模块引入主要有两种标准：



**CommonJs规范定义**很多普通应用程序（主要指非浏览器的应用）使用的API，填补了一开始Javascript官方定义的API只能构建基于浏览器的应用程序. 它的终极目标是提供一个类似Python，Ruby和Java标准库。这样的话，开发者可以使用CommonJS API编写应用程序，然后这些应用可以运行在不同的JavaScript解释器和不同的主机环境中。

在兼容CommonJS的系统中，你可以使用JavaScript开发以下程序：

(1)、服务器端JavaScript应用程序  
(2)、命令行工具  
(3)、图形界面应用程序  
(4)、混合应用程序（如，Titanium或Adobe AIR）

NodeJS是CommonJS规范的实现，webpack 也是以CommonJS的形式来书写。

---在CommonJS规范中使用require关键字来导入模块，使用module.exports来导出变量。

---require关键字的导入将会导入模块中的所有内容，无法通过选择性的导入模块中个别变量。

---require关键字导入时同步导入，step by step的导入。当导入完一个require语句以后，再导入下一个语句。主要用于服务器端。

**ES6规范的导入导出：**

ES6规范模块的导入导出，使用import以及export关键字。

使用import关键字导入可以选择模块中的某些变量进行选择性导入。

import导入是异步的。主要用于浏览器端的JS应用导入。

🡪

const increment = () => ({type: 'INCREMENT’}) //函数名为increment的方法

const decrement = () => ({type: 'DECREMENT’}) //函数名为decrement的方法

const reset = () => ({type: 'RESET’}) //函数名为reset的方法

{increment, decrement, reset} //将是一个对象，属性为increment, decrement, reset. 属性值为匿名箭头函数

const test1 = {“A”:” B”};

const test2 = {“C”:” D”};

{test1, test2} //将会是一个对象，属性为test1, test2， 属性值为test1所对应得对象，test2所对对应的对象。

🡪

"use strict" 指令在 JavaScript 1.8.5 (ECMAScript5) 中新增， 让javascript工作在严格模式下。为了让消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为。

在严格模式下，JS对会更多的限制进行报错，例如不允许使用未声明的变量，不允许删除变量或对象，不允许this指向全局对象，新增保留字等。

🡪

JavaScript是单线程执行的，无法同时执行多段代码。当某一段代码正在执行的时候，所有后续的任务都必须等待，形成一个队列。一旦当前任务执行完毕，再从队列中取出下一个任务，这也常被称为 “阻塞式执行” 。所以一次鼠标点击，或是计时器到达时间点，或是Ajax请求完成触发了回调函数，这些事件处理程序或回调函数都不会立即运行，而是立即排队，一旦线程有空闲就执行。假如当前 JavaScript线程正在执行一段很耗时的代码，此时发生了一次鼠标点击，那么事件处理程序就被阻塞，用户也无法立即看到反馈，事件处理程序会被放入任务队列，直到前面的代码结束以后才会开始执行。如果代码中设定了一个 setTimeout，那么浏览器便会在合适的时间，将代码插入任务队列，如果这个时间设为 0，就代表立即插入队列，但不是立即执行，仍然要等待前面代码执行完毕。所以 setTimeout 并不能保证执行的时间，是否及时执行取决于 JavaScript 线程是拥挤还是空闲。

也就是说setTimeout只能保证在指定的时间过后将任务(需要执行的函数)插入队列等候，并不保证这个任务在什么时候执行

🡪

Javascript中 instanceof为二元操作符: A instanceof B, A实例是否为B的类。

Typeof为一元操作符， typeof A 输出变量A的类型。

🡪

Javascript立即执行匿名函数。格式：外层 () 表示函数对象，最后以 () 表示马上调用。

(function () {

alert("我是一个匿名函数~~~");

})();

🡪

Javascript的构造函数, 或者语法糖Class的首字母大写。

例如

Class Person () {

}

或者

Function Person () {

}

🡪

**Object.keys(可枚举对象)**

Object.keys方法接受一个可枚举对象，并且返回该可枚举对象的属性。例如字典对象则返回其键， 数组对象则返回其下标。

Object.entries(**可枚举对象**)

Object.entries方法返回一个给定对象自身可枚举属性的键值对，并将每一个键值对放入一个数组，然后所有键值对数组放入一个整体数组中。形成数组of数组的结构。

例如：

let test = {“A”:” B”, “C”:” D”}

Object.entries(test) //为 [[“A”,” B”], [“C”,” D”]]

对象.hasOwnProperty(field)

检查对象自身（不包含原型链查找）中是否包含field的属性。如果含有返回true,如果不含有返回false.

🡪

JS中指定对象的key时，可以使用 [] 符号，表示运算 [] 中的式子结果作为键。

例如：

{

name: “AAA”,

[1+3]: 4 //添加一个键为4, 键值为4

}

🡪

ES6(ES2015) 中规定，如果一个对象中的方法被**get关键字修饰**，则该方法名成为对象中的一个属性名。如果该对象的该属性被访问，则调用该方法。

例如：

const obj = {

log: ['a'],

get latest () {

if (this.log.length == 0) {

return undefined;

}

return this.log [this.log.length - 1];

}

}

从而obj拥有了latest属性。如果使用obj.latest访问latest属性，则latest方法被调用，返回值。

🡪

JSON.stringify() 方法用于将 对象或数组转换为 JSON 字符串。

语法：

JSON.stringify(value[, replacer[, space]])

* 必需， 要转换的 JavaScript 值（通常为对象或数组）。
* replacer:

可选。用于转换结果的函数或数组。

如果 replacer 为函数，则 JSON.stringify 将调用该函数，并传入每个成员的键和值。使用返回值而不是原始值。如果此函数返回 undefined，则排除成员。根对象的键是一个空字符串：""。

如果 replacer 是一个数组，则仅转换该数组中具有键值的成员。成员的转换顺序与键在数组中的顺序一样。

* space:

可选，文本添加缩进、空格和换行符，如果 space 是一个数字，则返回值文本在每个级别缩进指定数目的空格，如果 space 大于 10，则文本缩进 10 个空格。space 也可以使用非数字，如：\t。

🡪

Javascript正则表达式

创建正则表达式:

**方式1：**

var pattern=/pattern/modifiers;

其中modifier是RegExp修饰符，i修饰符是用来执行不区分大小写的匹配。g修饰符是用于执行全文的搜索（而不是在找到第一个就停止查找, 而是找到所有的匹配）。

**方式2：**

var patt1=new RegExp("e"，修饰符);

正则表达式的test()方法：

test()方法搜索字符串指定的值，根据结果并返回真或假。

例如：

var patt1=new RegExp("e");

document.write(patt1.test("The best things in life are free")); //输出为true

正则表达式的exec()方法：

exec() 方法检索字符串中的指定值。返回值是被找到的值。如果没有发现匹配，则返回 null。

例如：

var patt1=new RegExp("e"); document.write(patt1.exec("The best things in life are free"));

//输出为e

🡪