<script type='application/javascript' src=''></script>

text/javascript 过时了

null转换成boolean是false

{} []转换成boolean是true

变量名以字母、下划线或者$开头

JS里单引号里边可以套双引号，双引号里也可以套单引号。里边的引号被当作是字符串的一部分。

document.getElementById("myDIV").classList.add("mystyle");

document.getElementById("myDIV").classList.remove("mystyle");

myDiv.classList.contains('myCssClass'); //return true or false

全局变量：函数外声明的变量，称为全部变量

局部变量：函数内部使用var声明的变量，称为局部变量

在JS中，只有函数作用域，没有块级作用域！也就是说，if/for等有{}的结构体，并不能具备自己的作用域。

## foreach

let array=[2, 4, 23, {}]

for(let index **in** array) {

console.log(index,array[index]);

};

0 2

1 4

2 23

3 {}

for(let v **of** array) {

console.log(v);

};

2

4

23

{}

## Number

var x=0.2+0.1; //0.30000000000000004

0.4-0.3; //0.10000000000000003

整数最大2^53-1 9007199254740992

不能精确表示十进制小数

2.445.toFixed(2); //IE 2.45 chrome 2.44

判断obj是否为一个整数

function isInteger(obj) {

return Math.floor(obj) === obj

}

避免小数精度问题，要将小数转换成整数进行运算

## 定时器

每隔1000ms执行一次

setInterval(function(){

console.log("alf"):

}, 1000);

停1000毫秒执行，只执行一次

setTimeout(function(){

console.log("asd");

}, 1000);

window的方法,内部的this指向window

var time1=setInterval(function(){}, 1000);

clearInterval(time1);

time1是1,2,3这种值，代表一个定时器的编号

clearTimeout(time2)

setTimeout(func1, 1000, 'hello')

第三个及以后的参数将作为参数传递给func1

var progress=0;

function render() { //回调函数

a.style.marginLeft=progress+'px'; //修改图像的位置

console.log(progress); //以屏幕刷新的频率运动

if (progress < 100) {

window.requestAnimationFrame(render);

}

}

window.requestAnimationFrame(render); //第一帧渲染

与setTimeout相比，requestAnimationFrame最大的优势是由系统来决定回调函数的执行时机。具体一点讲，如果屏幕刷新率是60Hz,那么回调函数就每16.7ms被执行一次，如果刷新率是75Hz，那么这个时间间隔就变成了1000/75=13.3ms。

它能保证回调函数在屏幕每一次的刷新间隔中只被执行一次，这样就不会引起丢帧现象，也不会导致动画出现卡顿的问题。

setTimeout看起来时间差严格，其实也不准，做出来也就看个样子。不如直接硬规定频率用requestAnimationFrame做出来依据效果再改动

## json

var obj={

name : "liu",

age : 12

};

JSON.stringify(obj);

{

"name" : "liu",

"age" : 19

}

json属性名必须加双引号

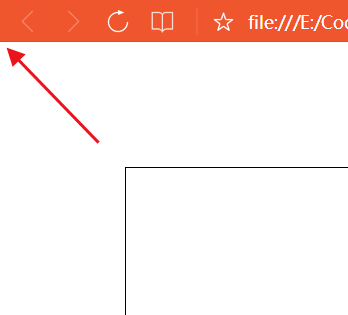
* string🡪json

JSON.parse()

* json🡪string

JSON.stringify()

## Event事件

* clientX,clientY：鼠标相对于浏览器窗口可视文档区域的左上角的位置；可视区域不包括工具栏和滚动条。就是相对于这个点的位置
* pageX,pageY (x,y)：鼠标相对于文档左上角的位置，而不是可视区域。因为有可能有滚动条，所以pageX>clientX。经测试page=x，但x, y不推荐使用
* offsetX,offsetY：鼠标相对于srcElement左上角的位置
* layerX,layerY：鼠标相对于最近的有定位的元素左上角的距离
* screenX、screenY：鼠标相对于显示器左上角的位置
* type：事件的类型，如click；
* srcElement/target：事件源，就是发生事件的元素；

jquery只能用target

* button：声明被按下的鼠标键，整数，1代表左键，2代表右键，4代表中键，如果按下多个键，酒把这些值加起来，所以3就代表左右键同时按下；（firefox中 0代表左键，1代表中间键，2代表右键）(没看出来)
* altKey,ctrlKey,shiftKey等：返回一个布尔值；
* keyCode：返回keydown何keyup事件发生的时候按键的代码，以及keypress 事件的Unicode字符；(firefox2不支持 event.keycode，可以用 event.which替代 )
* fromElement,toElement：前者指mouseover事件中鼠标移动过的文档元素，后者指代mouseout事件中鼠标移动到的文档元素；
* cancelBubble：一个布尔属性，把它设置为true的时候，将停止事件进一步起泡到包容层次的元素；(e.cancelBubble = true; 相当于 e.stopPropagation();)
* returnValue：一个布尔属性，设置为false的时候可以组织浏览器执行默认的事件动作；(e.returnValue = false; 相当于 e.preventDefault();)
* attachEvent(),detachEvent()/addEventListener(),removeEventListener：为制定 DOM对象事件类型注册多个事件处理函数的方法，它们有两个参数，第一个是事件类型，第二个是事件处理函数。在attachEvent()事件执行的时候，this关键字指向的是window对象，而不是发生事件的那个元素。attachEvent()在IE11以下版本使用

firefox里的event跟IE里的不同，IE里的是全局变量，随时可用；firefox里的要用参数引导才能用，是运行时的临时变量。

在IE/Opera中是window.event，在Firefox中是event；而事件的对象，在IE中是 window.event.srcElement，在Firefox中是event.target，Opera中两者都可用。

var evt = evt || window.event; // firefox下window.event为null, IE下event为null

事件先捕获后冒泡

捕获：从根节点（body）到div

冒泡：从div到根节点

e.stopPropagation()阻止冒泡

* && 操作符特点：只要一个是false就返回，都是true，返回最后一个
* || 操作符特点：只要一个是true就返回，都是false，返回最后一个。

在js逻辑运算中，0、""、null、false、undefined、NaN都会被判断为false

&&和||都是：当运算到某一个变量就得出最终结果之后，就返回哪个变量。

注意：返回的不是True或false，是变量的值

* 立即执行函数( function(){…} )()和( function (){…} () )两种写法都行

通过定义一个匿名函数，创建了一个新的函数作用域，相当于创建了一个“私有”的命名空间，该命名空间的变量和方法，不会破坏污染全局的命名空间

**let** 所声明的变量，只在let命令所在的代码块内有效

for (let i = 0; i < 10; i++) {

// 以后for循环用let吧

}

var b=[];

for(var i=0; i<10; i++){

b[i]=function(){

console.log(i);

}

}

b[6]()输出10

var a = [];

for (let i = 0; i < 10; i++) {

a[i] = function () {

console.log(i);

};

}

a[6]()输出6

和这样写效果一样

var b=[];

for(var i=0; i<10; i++){

(function(m){

b[i]=function(){

console.log(m);

}

})(i);

}

var t1;

let t2;

console.log(t1);

console.log(t2);

输出

undefined

undefined

console.log(t1);

报错

Uncaught ReferenceError: t1 is not defined at ttt.html:13

console.log(t1);

console.log(t2);

var t1;//var如果没赋值，定义在前面或者后面没有关系

let t2;//let定义在后边会报错

输出

undefined

ttt.html:14 Uncaught ReferenceError: t2 is not defined at ttt.html:14

<h1 id="nihao">

<p>hello world</p>

</h1>

var a=document.getElementById('nihao');

* console.log(a)

<h1 id="nihao">

<p>hello world</p>

</h1>

* a.innerText

"hello world"

a.innerText='<em>你好世界</em>'

改变内容

* a.innerHTML

"<p>hello world</p>"

* parentNode

返回父元素

节点类型

* 元素节点
* 属性节点
* 文本节点
* 注释节点

<h1 id="a">

<p>hello world</p>

</h1>

* a.childNodes

输出这个数组NodeList(3) [text, p, text]

0:text

1:p

2:text

length:3

\_\_proto\_\_:NodeList

* a.nextSibling
* a.previousSibling

a里 两个文本节点（空文本节点）一个元素节点

以上三个方法返回节点

* a.parentElement
* a.children
* a.firstElementChild
* a.lastElementChild
* a.nextElementSibling
* a.previousElementSibling

以上六个方法返回元素节点

javascript对象是基于原型prototye-based的，而java C++是基于类class-based的

js通过一个prototype对象创建更多的对象

{

f1: function(){

console.log('aaa')

}

}

和

{

f1(){

console.log('aaa');

}

}

是一种写法

## prototype 原型

函数（除了一些内建函数）都有一个名为prototype（原型）的属性，这个属性是一个指针，指向一个对象，而这个对象的用途是，包含可以有特定类型的所有实例共享的属性和方法。prototype是通过调用构造函数而创建的对象实例的原型对象。

hasOwnProperty()判断指定属性是否为自有属性；in操作符对原型属性和自有属性都返回true。

var obj = {a: 1};

obj.hasOwnProperty("a"); // true

obj.hasOwnProperty("toString"); // false

"a" in obj; // true

"toString" in obj; // true

示例：鉴别原型属性

function hasPrototypeProperty(obj, name){

return name in obj && !obj.hasOwnProperty(name);

}

\_\_proto\_\_

对象具有属性\_\_proto\_\_，可称为隐式原型，指向构造该对象的构造函数的原型，这也保证了实例能够访问在构造函数原型中定义的属性和方法。

var Boo = {name: "Boo"};

function Foo(){}

Foo.prototype = Boo;

var f = new Foo();

Object.getPrototypeOf(obj)

使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法

obj.isPrototypeOf(obj) 方法检查某个对象是否是另一个对象的原型对象

function F1(){

}

var obj1=new F1();

* F1.prototype

返回 一个对象

{

constructor:ƒ F1()

\_\_proto\_\_:Object

}

这就是典型的，一个由构造函数创建的对象的prototype。除了自带一个constructor属性之外就是个空对象。现在我们可以往这个prototype里面加东西A。以后所有F1构造出来的对象都有东西A

* obj1.\_\_proto\_\_

返回 上边那个对象

对象.\_\_proto\_\_=构造器.prototype

* obj1.\_\_proto\_\_.\_\_proto\_\_

返回null

* obj1.constructor

返回函数F1

实际上返回的是obj1.\_\_proto\_\_的属性

* 对象没有prototype，函数的\_\_proto\_\_没有意义

F1.\_\_proto\_\_

返回ƒ () { [native code] }

出问题了。函数的\_\_proto\_\_也是有意义的

F1.\_\_proto\_\_=Function.prototype

而这个Function.prototype 是一个function

基于原型创建多个对象

* 定义原型对象
* 定义对象的构造函数
* 将构造函数关联到原型
* 实例化对象

var person={

name: "aaa",

age: 12

}

function Person(a, b){

this.name=a;

this.age=b;

}

Person.prototype=person;

var p1=new Person('nihao', 23);

这个new看起来违背了基于原型的思想，来自原型的Object.create登场

var p2 = Object.create(person);

p2.name="nihao";

p2.age=23;

这样写又重复了，于是

function cons(a, b){

var obj=Object.create(person);

obj.name=a;

obj.age=2;

return obj

}

var p3=cons("nihao", 23);

变量和函数**提升**

javascript引擎进入作用域时对代码进行两轮处理

1.

* 声明并初始化函数参数
* 声明局部变量(undefined)
* 声明并初始化函数

2.执行代码

所以会出现

情况1

var a=11;

function f1(){

console.log(a);

var a;

}

输出undefined，因为里边的a在第一轮处理时就声明，但并未赋值

情况2

var a=11;

function f1(aa){

console.log(aa);//11

var aa;//这行相当于啥也没干。如果是var aa=22, 则arguments[0]也变成22

console.log(aa);//11

}

f1(a);

* 任何数据类型+字符串都=字符串
* 1+1+"a"+1+1="2a11" 从左向右执行
* 1 \* "3" = 3
* "6" - "4" = 2
* 1/0 Infinity
* 0/0 NaN
* NaN==NaN false

Undefined===Undefined true

null===null true

Undefined==null true

Undefined===null false

* undefined, NaN, null, "", 0, false转换成布尔值都是false
* window.prompt 可提示用户进行输入的对话框
* typeof null 🡪 object

typeof [213, 3] 🡪 object

* 沿着函数作用域链向上查找变量
* 沿着对象原型链向上查找属性

js执行会阻塞DOM树的解析和渲染

css加载不会阻塞DOM树的解析

css加载会阻塞DOM树的渲染

css加载会阻塞后面js语句的执行

## 变量类型

* 基本数据类型 number string boolean undefined null

保存在栈内存中

var a= 1;

操作的是变量实际保存的值。

基本类型变量的复制：在栈中创建一个新值，然后把值复制到为新变量分配的位置上。

* 引用数据类型：object array function

保存在堆内存中

也就是，变量中保存的实际上的只是一个指针，这个指针指向堆中的一个位置，该位置保存着对象。访问方式是按引用访问。

操作时，需要先从栈中读取内存地址，然后再延指针找到保存在堆内存中的值再操作。

引用类型变量的复制：复制的是存储在栈中的指针，将指针复制到栈中未新变量分配的空间中，而这个指针副本和原指针指向存储在堆中的同一个对象；复制操作结束后，两个变量实际上将引用同一个对象。改变其中的一个变量的值，将影响另一个变量。

声明object

var aaa={name: "xiaolin", age: 18}

var aaa=new Object();

aaa.age=18;

* symbol

JavaScript的第七种数据类型

* 对象的属性名现在可以有两种类型，一种是原来的字符串，另一种就是新增的 Symbol 类型。
* 凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

let s=**Symbol**(); //通过Symbol()函数生成symbol，注意没有new

let s1=Symbol();

typeof s; // 'symbol'

s==s1; //false

Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分

let s2 = Symbol('foo');

想读取的话显式toString()

s2.toString() // 'Symbol(foo)'

作为属性名

a[s] = 'Hello!';

let a = { [s]: 'Hello!' };

let a={ [s](param){console.log(param)} }; //结合增强对象写法的花式写法

这玩意可以当枚举使？

const log = {};

log.levels = {

DEBUG: Symbol('debug'),

INFO: Symbol('info'),

WARN: Symbol('warn')

};

console.log(log.levels.DEBUG, 'debug message');

console.log(log.levels.INFO, 'info message');

Symbol 作为属性名，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个Object.getOwnPropertySymbols方法，可以获取指定对象的所有 Symbol 属性名

Object.getOwnPropertySymbols(a); //[Symbol()]

Object.getOwnPropertyNames(a); //["rq"]

Reflect.ownKeys(a); //["rq", Symbol()] 这个可以返回所有，包括Symbol的属性名

Symbol.for方法它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

let s1 = Symbol.**for**('foo');

let s2 = Symbol.for('foo');

s1 === s2 // true

## 类型检查

**typeof** true; //'boolean'

typeof null; //'object'

typeof undefined; //'undefined'

typeof (()=>{}); //'function'

typeof Symbol(); //'symbol'

* object instanceof constructor

instanceof运算符用来检测 constructor.prototype 是否存在于 object 的原型链上

非object不能用instance检测

let x1=3;

let x2=new Number(3);

typeof x2; //"object"

x1 instanceof Object; //false

x2 **instanceof** Number; //true

x2.\_\_proto\_\_ instanceof Object; //true

Number instanceof Object; //true 不是因为number是object的实例，而是因为Number是一个函数，函数是Object的实例

[] instanceof Object; //true 总之右边如果是Object，则总为true。

理解这句

F1.\_\_proto\_\_==Function.\_\_proto\_\_ == Function.prototype

Function.\_\_proto\_\_.\_\_proto\_\_==Function.prototype.\_\_proto\_\_==Object.prototype

F1.\_\_proto\_\_.\_\_proto\_\_== Object.prototype

Function instanceof Object; //true

Object instanceof Function; //true

let F1=()=>{};

F1 instanceof F1; //false 右边F1的prototype是一个对象。左边F1找的是\_\_proto\_\_,是一个函数，等于Function.prototype，所以不再原型链上

F1 instanceof Function; //true

js 在底层存储变量的时候，会在变量的机器码的低位1-3位存储其类型信息

* 000：对象
* 010：浮点数
* 100：字符串
* 110：布尔
* 1：整数

对于 undefined 和 null 来说，较为特殊

* null：所有机器码均为0
* undefined：用 -2^30 整数来表示

所以，typeof 在判断 null 的时候就出现问题了，由于 null 的所有机器码均为0，因此直接被当做了对象来看待。

**Object.prototype.toString**.call(1) // "[object Number]"

Object.prototype.toString.call('hi') // "[object String]"

Object.prototype.toString.call({a:'hi'}) // "[object Object]"

Object.prototype.toString.call([1,'a']) // "[object Array]"

Object.prototype.toString.call(true) // "[object Boolean]"

Object.prototype.toString.call(() => {}) // "[object Function]"

Object.prototype.toString.call(null) // "[object Null]"

Object.prototype.toString.call(undefined) // "[object Undefined]"

Object.prototype.toString.call(Symbol(1)) // "[object Symbol]"

## 类型转换

var m= Number('456');

var m2=parseInt('123abc234feaw');//123

var m3=Boolean('{}'); //true

## window.onload

window.onload=function(){

alert('hello');

}

$(window).load(function(){});

$(document).ready(function(){

})

$(function(){}) //上一种方法的简写

等待文档加载完毕后执行

等待document只需要整个DOM结构加载完，等待window要等待图片等等

ducument.querySelector()

返回选中的第一个

document.querySelectorAll()

返回一个数组

## Date

GMT Greenwich Mean Time 格林尼治标准时间/世界时

UTC Coordinated Universal Time 协调世界时/世界协调时间

UTC是经过平均太阳时(以格林威治时间GMT为准)、地轴运动修正后的新时标以及以「秒」为单位的国际原子时所综合精算而成的时间，计算过程相当严谨精密，因此若以「世界标准时间」的角度来说，UTC比GMT更加精准。其误差值必须保持在0.9秒以内，若大于0.9秒则由位于巴黎的国际地球自转事务中央局发布闰秒，使UTC与地球自转周期一致。

大多数情况，UTC和GMT一样

地球二十四个时区，经度15度，差1小时。北京东八区

北京时间=GMT时间+8小时

如果GMT 2019.05.13 0:00，那么北京时间2019.05.13 8:00, 东十一区2019.05.13 11:00, 西十一区2019.05.12 13:00，按这样算，东十二区是5月13号中午12点，西十二区是5月12号中午12点，真的就是这样。其实，东西十二区是一个时区，加起来15度，时刻相同但是日期不同。中间有条线叫日界线，从东到西日期加一天，从西到东日期减一天。

var aaa=new Date();

console.log(aaa);

Mon May 13 2019 11:24:06 GMT+0800 (中国标准时间)

getDate() 1 ~ 31 13

getDay() 0 ~ 6 周日是1！ 1

getMonth() 0 ~ 11 4

getFullYear() 2019

getHours() 0 ~ 23 11

getMinutes() 0 ~ 59 24

getSeconds() 0 ~ 59 6

getMilliseconds() 0 ~ 999 963，也就是当前时间24分06秒963毫秒

getTime() 自1970.01.01的毫秒数 1557718435471

getTimezoneOffset() 与GMT的分钟差 -480

aaa.toDateString() "Mon May 13 2019"

aaa.toTimeString() "11:40:35 GMT+0800 (中国标准时间)"

aaa.toUTCString() "Mon, 13 May 2019 03:40:35 GMT"

aaa.toLocaleString() "2019/5/13 上午11:40:35"

aaa.toLocaleTimeString() "上午11:40:35"

aaa.toLocaleDateString() "2019/5/13"

## Array

var a = Array(); // []

var a = Array(3); // [undefined,undefined,undefined]

var a = Array(3,11,8); // [ 3,11,8 ]

console.log(a.length) // 3

遍历

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

console.log(item, index);

});

item是值

index是索引

array是数组本身

a.push(12)

向末尾添加元素，返回新的长度。

a.unshift('Strawberry')

向头部添加元素，返回新的长度。

var last = a.pop();

删除末尾元素。返回删除的值

var first = a.shift()

删除第一个元素。返回

var pos = a.indexOf(11);

查找位置

指定位置删除/替换/添加

let a = [5, 12, 132, 'nihao']

var removedItem = a.splice(1)

// a = [5], removedItem = [12, 132, "nihao"]

删除a[1]-a[a.length-1]的元素，包括a[1]，返回删除了的元素

var removedItem = a.splice(1,2) // a=[5, "nihao"]

从a[1]开始，删两个

a.splice(1, 2, 'shijie', 321) // a = [5, "shijie", 321, "nihao"]

从a[1]开始，删两个，再添加后边的元素

提取/复制

let a = [5, 12, 'nihao']

let newArr = a.slice(1) //a不变，newArr = [12, 'nihao']

复制a[1]-a[a.length-1]，包括a[1]，返回新数组

a.slice(1, 4)

复制a[1]-a[4]，包括a[1]不包括a[4]

连接

let a=[1, 2, 3]

a.concat([8], ['12', 're']) // a不变，返回[1, 2, 3, 8, "12", "re"]

会把数组拆成一项一项的连

slice和concat都是浅拷贝。map和filter也是

对于对象，只复制引用。改变原数组，还会影响新数组里的元素

对于string，number，boolean，复制值过去

let a=[1, 2]

a.map(item => item\*2)

创建一个新数组，由回调函数的结果组成

map也是浅拷贝，如果item是一个对象或者引用类型，新数组会影响原数组。所以其实新数组的对象和老数组的对象是一样的，都是修改后的结果

const words = ['spray', 'limit', 'elite', 'exuberant', 'present'];

const result = words.filter(word => word.length > 6);

filter 为数组中的每个元素调用一次 callback 函数，返回所有使得 callback 返回 true 或等价于 true 的值的元素组成的新数组。

callback 三个参数，后两个可选：元素的值element，元素索引index，被遍历的数组本身array

* join() 把数组的所有元素放入一个字符串。元素通过指定的分隔符进行分隔。
* reverse() 颠倒顺序
* sort() 排序

字符串不需要写排序函数，数字需要

function sortNumber(a,b){

return a - b

}

var arr = []

arr[0] = "10"

arr[1] = "5"

arr[2] = "40"

document.write(arr.sort(sortNumber))

* valueOf() 返回数组对象的原始值

字符串🡪数组

**Array.from**('hello') // ["h", "e", "l", "l", "o"]

类数组🡪数组

function(x, y){

return Array.from(arguments);

}

## String

let str = 'nihao';

str.includes('hao') //true

判断一个字符串是否包含另一个字符串，返回 true 或 false。

str.includes('hao', 3) //false

从str.charAt(3)开始搜索

## 立即执行函数

var mmm = [];

for (var i = 0; i < 6; i++) {

mmm[i] = function() {

console.log(i)

}

}

这样写，mmm[3](100)也输出6，因为这个函数里边的i相当于是找全局的i

for (var i = 0; i < 6; i++) {

mmm[i] = function(i) {

console.log(i)

}

}

这样写，mmm[3](100)输出100

for (var i = 0; i < 6; i++) {

mmm[i] = (function() {

console.log(i)

})()

}

这样写，直接打印012345，而且mmm=[undefined, undefined, undefined, undefined, undefined, undefined]

## 深拷贝

function deepClone(obj){

let objClone = Array.isArray(obj)?[]:{};

if(obj && typeof obj==="object"){

for(key in obj){

if(obj.hasOwnProperty(key)){

//判断ojb子元素是否为对象，如果是，递归复制

if(obj[key]&&typeof obj[key] ==="object"){

objClone[key] = deepClone(obj[key]);

}else{

//如果不是，简单复制

objClone[key] = obj[key];

}

}

}

}

return objClone;

}

.concat(), .slice()方法都是浅拷贝，string，number，boolean复制一份副本，object，array，function指向同一个对象。

## defer async

<script src="path/to/myModule.js" **defer**></script>

<script src="path/to/myModule.js" **async**></script>

defer要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行；async一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer是“渲染完再执行”，async是“下载完就执行”。另外，如果有多个defer脚本，会按照它们在页面出现的顺序加载，而多个async脚本是不能保证加载顺序的。

## 装箱

* 基本类型转换成对应的引用类型称为装箱
* 引用类型转换为对应的基本类型称为拆箱

每当读取一个基本类型的时候，后台就会创建一个对应的基本包装类型对象，从而能够调用一些方法来操作这些数据。这就是**装箱**

var str='nihao';

str.substring(2); //'hao'

相当于这个过程

var n1 = new Number(23);

var n2 = n1.toFixed(2); //"23.00"

n1 = null;

而Number的prototype有以下方法

* toExponential: ƒ toExponential()
* toFixed: ƒ toFixed()
* toPrecision: ƒ toPrecision()
* toString: ƒ toString()
* valueOf: ƒ valueOf()

所以number基本类型也能使用这些方法

相比起来string基本类型可以使用的方法就有很多

* substr: ƒ substr()
* substring: ƒ substring()
* toUpperCase: ƒ toUpperCase()
* trim: ƒ trim()
* ...

**拆箱**

* 手动拆箱

numObj.valueOf();

strObj.toString();

* 隐式拆箱或自动拆箱

拆箱过程内部调用了ToPrimitive抽象操作

该操作接收两个参数，要转变的对象和PreferredType被期待转成的类型

PreferredType不是必须的，默认number

先调用valueOf方法看有没有返回值，如果没有则调用toString方法，如果toString也没有返回值则报错TypeError

let obj={

valueOf: ()=>{return 'invoke valueOf first'},

toString: ()=>{return 'invoke toString first'}

}

console.log(obj+' aaaa');//invoke valueOf first aaaa

Number(obj); //Uncaught TypeError: Cannot convert object to primitive value

如果使用String(obj)方法，PreferredType参数会传入string，则先调用toString()方法，后调用valueOf()，都不返回基本类型时报错

String(obj)

String(obj)

显示"invoke toString first"

但是不能理解为String(x)返回x.toString()

let obj={

valueOf: ()=>{return 23},

toString: ()=>{return {nihao:3}}

}

String(obj)

"23"

obj.toString()

{nihao: 3}

## canvas

<canvas id="tutorial" width="300" height="300"></canvas>

* ​<canvas>默认 width为300、height 为 150，单位都是 px。
* 建议永远不要使用 css 属性来设置 <canvas> 的宽高。

<canvas>

如果你的浏览器不支持 canvas，就会出现这一段话。

</canvas>

支持 <canvas> 的浏览器会只渲染 <canvas> 标签，而忽略其中的内容。不支持 <canvas> 的浏览器会直接渲染替代内容。

* fillRect(x, y, width, height)：**绘制矩形**。
* strokeRect(x, y, width, height)：绘制一个矩形的边框。
* clearRect(x, y, widh, height)：清除指定的矩形区域，然后这块区域会变的完全透明。

function draw(){

var canvas = document.getElementById('tutorial');

if(!canvas.getContext) return;

var ctx = canvas.getContext("2d");

ctx.fillRect(10, 10, 100, 50); // 绘制矩形，填充的默认颜色为黑色

ctx.strokeRect(10, 70, 100, 50); // 绘制矩形边框

}

draw();

只有矩形能够通过给定的函数画出，其他形状都需要绘制路径完成

**绘制路径**

* 一个路径，甚至一个子路径，都是闭合的。但是绘制下一个路径之前必须要闭合上一个路径。。。

function draw(){

var canvas = document.getElementById('tutorial');

if (!canvas.getContext) return;

var ctx = canvas.getContext("2d");

ctx.beginPath(); //新建一条path

ctx.moveTo(50, 50); //把画笔移动到指定的坐标

ctx.lineTo(200, 50); //绘制一条从当前位置到指定坐标(200, 50)的直线.同时移动画笔

ctx.lineTo(125, 150);

ctx.closePath(); //闭合路径，把当前画笔位置移到起点

ctx.stroke();

}

draw();

moveTo(x, y)把画笔移动到指定的坐标(x, y)。相当于设置路径的起始点坐标。

closePath()闭合路径之后，图形绘制命令又重新指向到上下文中

stroke()通过线条绘制图形轮廓。stroke不会自动闭合路径，试了一下好像fill会

fill()填充路径的内容区域生成实心的图形

**绘制圆弧**

有两个方法可以绘制圆弧：

* arc(x, y, r, startAngle, endAngle, anticlockwise)

以(x, y)为圆心，以r为半径，从 startAngle弧度开始到endAngle弧度结束。

anticlosewise是布尔值，true表示逆时针，false表示顺时针。默认顺时针。

这里的度数都是弧度。

radians=(Math.PI/180)\*degrees //角度转换成弧度

ctx.beginPath();

ctx.arc(10, 10, 100, 0, Math.PI);

ctx.fill();

其实是画了一条路径，然后描边或填充。

有时候画的超出去canvas的范围了，但还是画了，填充也填了，只不过超出范围看不到。

* arcTo(x1, y1, x2, y2, radius): 根据给定的控制点和半径画一段圆弧，最后再以直线连接两个控制点。

这个方法可以这样理解。绘制的弧形是由两条切线所决定。

第 1 条切线：起始点和控制点1决定的直线。

第 2 条切线：控制点1 和控制点2决定的直线。

**绘制贝塞尔曲线**

* 二次贝塞尔曲线:

quadraticCurveTo(cp1x, cp1y, x, y)

​参数 1 和 2：控制点坐标

​参数 3 和 4：结束点坐标

* 三次贝塞尔曲线:

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)

​参数 1 和 2：控制点 1 的坐标

​参数 3 和 4：控制点 2 的坐标

​参数 5 和 6：结束点的坐标

添加颜色

fillStyle = color 设置图形的填充颜色

strokeStyle = color 设置图形轮廓的颜色

color 可以是表示 css 颜色值的字符串、渐变对象或者图案对象。

一旦设置了 strokeStyle 或者 fillStyle 的值，那么这个新值就会成为新绘制的图形的默认值。如果要给每个图形上不同的颜色，需要重新设置 fillStyle 或 strokeStyle 的值。

ctx.fillStyle='#f40';

ctx.strokeStyle='rgba(20, 344, 125, 1)';

* ctx.lineWidth = 20; 线宽。默认1
* ctx.lineCaps = "butt" 线段末尾样式，

"butt" / "round" / "square"

方形/圆形/方形但是增加了一个宽度和线段相同，高度是线段厚度一半的矩形区域

* ctx.lineJoin = 'round'; 同一个 path 内，线条与线条间接合处的样式。

'round' / 'bevel' / 'miter

round 通过填充一个额外的，圆心在相连部分末端的扇形，绘制拐角的形状。圆角的半径是线段的宽度。

bevel 在相连部分的末端填充一个额外的以三角形为底的区域， 每个部分都有各自独立的矩形拐角。

miter(默认) 通过延伸相连部分的外边缘，使其相交于一点，形成一个额外的菱形区域。

* 虚线

setLineDash 方法接受一个数组，来指定线段与间隙的交替

lineDashOffset属性设置起始偏移量

ctx.setLineDash([20, 5]); // [实线长度, 间隙长度]

ctx.lineDashOffset = -0;

**绘制文本**

fillText(text, x, y [, maxWidth])填充文本，绘制的最大宽度是可选的。

strokeText(text, x, y [, maxWidth]) 描边文本，绘制的最大宽度是可选的。

ctx.font = "100px sans-serif"

ctx.fillText("天若有情", 10, 100);

ctx.strokeText("天若有情", 10, 200)

* font字体。这个字符串使用和 CSS font 属性相同的语法。 默认 10px sans-serif。
* textAlign = value 文本对齐选项。 可选的值包括：start, end, left, right or center。 默认 start。
* textBaseline = value 基线对齐选项，可选的值包括：top, hanging, middle, alphabetic, ideographic, bottom。默认 alphabetic。。
* direction = value 文本方向。可能的值包括：ltr, rtl, inherit。默认 inherit。

strokeText效果如下：

**伤过的心就像玻璃碎片**

**绘制图片**

var img = new Image(); // 创建一个<img>元素

img.src = 'myImage.png'; // 设置图片源地址

参数 1：要绘制的 img

参数 2、3：绘制的 img 在 canvas 中的坐标

ctx.drawImage(img,0,0);

注意：考虑到图片是从网络加载，如果 drawImage 的时候图片还没有完全加载完成，则什么都不做，个别浏览器会抛异常。所以我们应该保证在 img 绘制完成之后再 drawImage。

var img = new Image(); // 创建img元素

img.onload = function(){

ctx.drawImage(img, 0, 0)

}

img.src = 'myImage.png'; // 设置图片源地址

img 先加载，然后调用onload()

绘制 img 标签元素中的图片

​img 可以 new 也可以来源于我们页面的 <img>标签。

<img src="./美女.jpg" alt="" width="300"><br>

<canvas id="tutorial" width="600" height="400"></canvas>

function draw(){

var canvas = document.getElementById('tutorial');

if (!canvas.getContext) return;

var ctx = canvas.getContext("2d");

var img = document.querySelector("img");

ctx.drawImage(img, 0, 0);

}

document.querySelector("img").onclick = function (){

draw();

}

drawImage(image, x, y, width, height)

​这个方法多了 2 个参数：width 和 height，这两个参数用来控制当像 canvas 画入时应该缩放的大小。

切片

drawImage(image, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight)

在sx,sy坐标绘制sWidth, sHeight大小的图片，在图片内部的dx, dy坐标处裁剪出dWidth, dHeight大小的子图片

**保存状态**

Saving and restoring state 是绘制复杂图形时必不可少的操作。

save 和 restore 方法是用来保存和恢复 canvas 状态的，都没有参数。

​Canvas 的状态就是当前画面应用的所有样式和变形的一个快照。

* 关于 save() ：Canvas状态存储在栈中，每当save()方法被调用后，当前的状态就被推送到栈中保存。

一个绘画状态包括：

* 当前应用的变形（即移动，旋转和缩放）
* strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffsetX, shadowOffsetY, shadowBlur, shadowColor, globalCompositeOperation 的值
* 当前的裁切路径（clipping path）
* 可以调用任意多次 save方法(类似数组的push())。
* 关于restore()：每一次调用 restore 方法，上一个保存的状态就从栈中弹出，所有设定都恢复(类似数组的 pop())

var ctx;

function draw(){

var canvas = document.getElementById('tutorial');

if (!canvas.getContext) return;

var ctx = canvas.getContext("2d");

ctx.fillRect(0, 0, 150, 150); // 使用默认设置绘制一个黑矩形

ctx.save(); // 保存默认状态

ctx.fillStyle = 'red' // 在原有配置基础上对颜色做改变

ctx.fillRect(15, 15, 120, 120); // 使用新的设置绘制一个小红矩形

ctx.restore(); // 重新加载之前的颜色状态

ctx.fillRect(45, 45, 60, 60); // 使用上一次的配置绘制一个更小的黑矩形

}

draw();

translate(x, y) 用来移动 canvas 的原点到指定的位置

​translate 方法接受两个参数。x 是左右偏移量，y 是上下偏移量

ctx.strokeRect(0, 0, 100, 100)

ctx.translate(100, 100);

ctx.fillRect(0, 0, 100, 100)

移动不影响移动前已经绘制出的图形

rotate(angle) ​旋转坐标轴。

旋转的中心是坐标原点。

scale(x, y) 缩放

x,y 分别是横轴和纵轴的缩放因子

transform矩阵变形，有点复杂，不看了

transform(a, b, c, d, e, f)

**合成**

两个图像摞在一起

ctx.fillStyle = "blue";

ctx.fillRect(0, 0, 200, 200);

ctx.globalCompositeOperation = "source-over"; //全局合成操作

ctx.fillStyle = "red";

ctx.fillRect(100, 100, 200, 200);

type 是下面 13 种字符串值之一：

* source-over这是默认设置，新图像会覆盖在原有图像。
* source-in仅仅会出现新图像与原来图像重叠的部分，其他区域都变成透明的。(包括其他的老图像区域也会透明)
* source-out仅仅显示新图像与老图像没有重叠的部分，其余部分全部透明。(老图像也不显示)
* source-atop新图像仅仅显示与老图像重叠区域。老图像仍然可以显示。
* destination-over新图像会在老图像的下面。
* destination-in仅仅新老图像重叠部分的老图像被显示，其他区域全部透明。
* destination-out仅仅老图像与新图像没有重叠的部分。 注意显示的是老图像的部分区域。
* destination-atop老图像仅仅仅仅显示重叠部分，新图像会显示在老图像的下面。
* lighter新老图像都显示，但是重叠区域的颜色做加处理。
* darken保留重叠部分最黑的像素。(每个颜色位进行比较，得到最小的)
* lighten保证重叠部分最量的像素。(每个颜色位进行比较，得到最大的)
* xor重叠部分会变成透明。
* copy只有新图像会被保留，其余的全部被清除(边透明)。

**裁剪路径**

clip() ​把已经创建的路径转换成裁剪路径。

裁剪路径的作用是遮罩。只显示裁剪路径内的区域，裁剪路径外的区域会被隐藏。

**动画**

配合这三个函数

setInterval()

setTimeout()

requestAnimationFrame()

模拟时钟

init();

function init(){

let canvas = document.querySelector("#solar");

let ctx = canvas.getContext("2d");

draw(ctx);

}

function draw(ctx){

requestAnimationFrame(function step(){

drawDial(ctx); //绘制表盘

drawAllHands(ctx); //绘制时分秒针

requestAnimationFrame(step);

});

}

/\*绘制时分秒针\*/

function drawAllHands(ctx){

let time = new Date();

let s = time.getSeconds();

let m = time.getMinutes();

let h = time.getHours();

let pi = Math.PI;

let secondAngle = pi / 180 \* 6 \* s; //计算出来s针的弧度

let minuteAngle = pi / 180 \* 6 \* m + secondAngle / 60; //计算出来分针的弧度

let hourAngle = pi / 180 \* 30 \* h + minuteAngle / 12; //计算出来时针的弧度

drawHand(hourAngle, 60, 6, "red", ctx); //绘制时针

drawHand(minuteAngle, 106, 4, "green", ctx); //绘制分针

drawHand(secondAngle, 129, 2, "blue", ctx); //绘制秒针

}

/\*绘制时针、或分针、或秒针

\* 参数1：要绘制的针的角度

\* 参数2：要绘制的针的长度

\* 参数3：要绘制的针的宽度

\* 参数4：要绘制的针的颜色

\* 参数4：ctx

\* \*/

function drawHand(angle, len, width, color, ctx){

ctx.save();

ctx.translate(150, 150); //把坐标轴的远点平移到原来的中心

ctx.rotate(-Math.PI / 2 + angle); //旋转坐标轴。 x轴就是针的角度

ctx.beginPath();

ctx.moveTo(-4, 0);

ctx.lineTo(len, 0); // 沿着x轴绘制针

ctx.lineWidth = width;

ctx.strokeStyle = color;

ctx.lineCap = "round";

ctx.stroke();

ctx.closePath();

ctx.restore();

}

/\*绘制表盘\*/

function drawDial(ctx){

let pi = Math.PI;

ctx.clearRect(0, 0, 300, 300); //清除所有内容

ctx.save();

ctx.translate(150, 150); //一定坐标原点到原来的中心

ctx.beginPath();

ctx.arc(0, 0, 148, 0, 2 \* pi); //绘制圆周

ctx.stroke();

ctx.closePath();

for (let i = 0; i < 60; i++){//绘制刻度。

ctx.save();

ctx.rotate(-pi / 2 + i \* pi / 30); //旋转坐标轴。坐标轴x的正方形从 向上开始算起

ctx.beginPath();

ctx.moveTo(110, 0);

ctx.lineTo(140, 0);

ctx.lineWidth = i % 5 ? 2 : 4;

ctx.strokeStyle = i % 5 ? "blue" : "red";

ctx.stroke();

ctx.closePath();

ctx.restore();

}

ctx.restore();

}

## this

普通函数中，this对象是在运行时基于函数的执行环境绑定的：在全局函数中，this指向的是window；当函数被作为某个对象的方法调用时，this就等于那个对象。

var name='Mike';

var a={

name:'Bob',

f1: function(){

return this.name;

},

f2: function(){

return function(){

return this.name;

}

},

f3: function(){

return name;

}

}

console.log(a.name);

console.log(a.f1());

console.log(a.f2()());

console.log(a.f3());

仔细读上面这段代码，我觉得很经典。

* 如果在浏览器里执行，返回Bob Bob Mike Mike，在node里执行返回Bob Bob undefined Mike。
* 浏览器里全局对象是window，node里没有window对象，全局的this是一个空对象。即，在浏览器中，直接conosle.log(this)输出window，在node中，直接conosle.log(this)输出{}.
* 匿名函数的执行环境是全局。所以对于f2，浏览器里this.name返回window.name，即Mike。node里返回{}.name，即undefined。
* 函数里不是this.name，而是直接一个name，那么即使通过对象调用，也去全局范围内找。

不对不对不对不对，不能这么解释。node全局并不是一个空对象，对于f2，node虽然返回undefined，但是却不是因为this是{}。相反，this里边有很多很多属性，像是node的window，但是这个this里边却没有name。现在把这个node里边一大堆属性的对象叫NoWindow。

(function(){

console.log(this)

})()

console.log(this)

在浏览器里运行，都是window，OK。在node里运行，前面输出NoWindow，后面输出{}。

好，现在我知道了不管node还是浏览器里都有全局对象。现在再看这个问题，f2在node下执行为什么是undefined。想不通，放弃了。

node里也别在函数外卵用this

val=1;

var obj={

val: 2,

f:function(){

console.log(this.val);

}

}

var func=obj.f;

func();//1

不通过对象调用，而是var func=obj.f;用一个变量把它提取出来了，那么因为this在运行时基于函数的执行环境绑定，所以绑定了window，this.val=1。

关键的问题来了，这一段在node中运行还是输出1.

针对以上所有问题，唯一合理的解释就是，匿名函数也在定义时绑定this，和箭头函数一样。所以匿名函数绑定的全局对象没有name属性，而正常函数都是在运行时绑定this。然而，至今没有看到任何和我猜想相同的说法。

至于箭头函数中的this，等然后再说吧。我服了。

## 事件循环event loop

JavaScript有一个基于事件循环的并发模型

* 栈stack

函数调用形成了一个栈，栈由若干帧frame组成。

function foo(b) {

let a = 10;

return a + b + 11;

}

function bar(x) {

return foo(x);

}

console.log(bar(7));

* 当调用 bar 时，一个包含了 bar 的参数和局部变量的帧被创建。
* 当 bar 调用 foo 时，第二个帧被创建并被压入栈中，放在第一个帧之上，帧中包含 foo 的参数和局部变量。参数就是b，局部变量就是a
* 当 foo 执行完毕然后返回时，第二个帧就被弹出栈。当 bar 也执行完毕然后返回时，第一个帧也被弹出，栈就被清空了。
* 堆Heap

对象被分配在堆中

* 队列Queue

一个 JavaScript 运行时JavaScript runtime包含了一个消息队列message queue。每一个消息都关联着一个用以处理这个消息的回调函数。

在事件循环期间的某个时刻，运行时会开始处理队列中的消息，从最先进入队列的消息开始。

处理过程：消息关联的函数被调用，消息被作为输入参数传入。被处理的消息会被移出队列。调用一个函数总是会为其创造一个新的栈帧。

然后事件循环将会处理队列中的下一个消息。

消息好像由监听点击事件产生，还有setTimeOut这种产生。具体不清楚，然后再看看吧

# ES6

## Promise

A Promise is a proxy for a value not necessarily known when the promise is created.

Promise是一个值得代理，这个值在Promise创建时未知

The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

Promise对象代表一个异步操作的最终完成或失败，以及操作结果

### 创建Promise

const promiseA = new Promise(function(resolve, reject){

});

* 构造函数接受一个名为executor的函数
* executor函数接受两个参数resolve和reject，它们都是函数。
* reject一般返回错误原因
* new Promise()之后executor函数立即执行，即使不then()也会执行

let p1 = new Promise(function(resolve, reject){

resolve('p1666')

})

console.log(p1)

Promise {<fulfilled>: "p1666"}

[[PromiseState]]: "fulfilled"

[[PromiseResult]]: "p1666"

Promise在声明之后会立即执行，所以此时Promise的状态是fulfilled，已经resolve了p1666

这应该就是Observable比Promise的先进之处，Observable声明之后不会立即执行

let p1 = new Promise(function(resolve, reject){

reject('p1000')

})

console.log(p1);

Promise {<rejected>: "p1000"}

[[PromiseState]]: "rejected"

[[PromiseResult]]: "p1000"

表示Promise的状态是rejected

let p = new Promise(function (resolve, reject) {

setTimeout(() => {

resolve("nihao");

}, 10000);

});

console.log(p);

Promise {<pending>}

[[PromiseState]]: "pending"

[[PromiseResult]]: undefined

设置一个setTimeout()，这样就能看到Promise的初始状态是pending

在调用resolve或reject函数之后，就变为fulfilled或者rejected状态

* 一个Promise有以下状态

1. nothing happened yet

2. "lock in" to another promise

3. fulfilled

4. rejected

其中{1, 2}为pending，{3, 4}为settled，{2, 3, 4}为resolved，{1}为unresolved

所以也可以说有pending, fulfilled, rejected三种状态

也可以说有pending、resolved两种状态

有篇教程能console.log出resolve，但是我试不出来

### 原型方法then, catch, finally

promise.then(successCallback, failureCallback);

let p = new Promise(function (resolve, reject) {

resolve(200);

});

p.then((a) => {console.log(a)})

200

* then、catch、finally都应用于Promise的一个实例，所以不管是通过new Promise()还是别人已经创建好的函数返回的Promise对象，都需要先获得一个Promise实例
* 此例中then方法没有failureCallBack方法，当Promise里是reject(200)时，控制台会报错，但是程序可以运行
* 对于successCallback，接受的参数就是Promise的结果，返回值有两种情况
* 返回一个Promise，比如return new Promise()，则then()的返回值就是这个Promise，所以可以继续链式调用
* 不显示指定return的Promise，例如return 0. 则自动将这个返回值包装成Promise.resolve(0)返回，所以还是可以链式调用
* 如果没有return语句，则返回一个Promise.resolve(undefined)

let p = new Promise(function (resolve, reject) {

reject(200);

});

p.then(() => {}).catch(a => console.log(a));

当Promise中是reject(200)时，有两种解决方法

* 给then()方法提供第二个参数，即failureCallback，failureCallback的参数就是reject的值。
* 指定.catch()方法，.catch也提供failureCallback。即使then()不返回Promise，catch和finally也会生效

p.finally(function(){

})

finally()方法不能带参数。它接收不到resolve或者reject传出的值

### 静态方法

let p = Promise.reject("ok");

等效于

let p = new Promise((resolve, reject) => {

reject('ok')

}

Promise.resolve(value)同理

* Promise.All

let generatePromise = function(x){

return new Promise(function(resolve, reject){

resolve(x);

})

}

let promiseList = [generatePromise(1), generatePromise(2), generatePromise('nihao')];

let promisesHandler = Promise.all(promiseList);

promisesHandler.then(x => {

console.log(x);

})

显示[1, 2, "nihao"]

generatePromises()生成Promise

promiseList是一个Promise的数组

Promise.all()接收一个参数，参数是由Promise组成的数组

Promise.all()返回一个Promise，此Promise在参数数组中所有Promise resolve之后resolve

返回的Promise resolve会得到一个数组，值为PromiseList所有Promise resolve的值组成的list

如果有其中一个Promise reject，则返回第一个失败Promise的结果

* Promise.race

All是等所有Promise resolve完毕后返回所有结果的数组，race是返回最先结束的Promise的结果

## async和await

这已经是ES2017的内容了

async关键字代表后面的函数中有异步操作，await表示等待一个异步方法执行完成。

let generatePromise = function () {

console.log("firstline in generatePromise()!");

let m = new Promise(function (resolve, reject) {

setTimeout(() => {

reject;

resolve("promise resovled");

}, 3000);

});

return m;

};

let completePromise = async function () {

let v = await generatePromise();

console.log(v);

console.log("last output in complete");

};

completePromise();

console.log("outside2");

显示

firstline in generatePromise()!

outside2

promise resovled

last output in complete

* 有一个返回Promise的函数generatePromise()。这个函数声明很正常，啥都没有
* 有一个要执行Promise的函数completePromise()，这个函数带有async
* completePromise返回一个Promise，如果不返回Promise或者没有return语句，立刻包装一个Promise。但是感觉这个Promise没啥用
* await表示等待一个Promise。如果是await 200这种不是等待Promise对象的，立刻包装
* let value = await generatePromise()相当于

let p = generatePromise();

let value;

p.then(a => value = a)

* 但还不是完全等价，因为await会造成阻塞，阻止后边代码。但是p.then()不会阻止后边代码的异步执行。比如对比下列代码

let completePromise = async function () {

let v = generatePromise();

v.then(a => console.log(a));

console.log("last output in complete");

};

completePromise();

console.log("outside2");

显示

firstline in generatePromise()!

last output in complete

outside2

promise resovled

1234变成了1423

* 总结：async函数调用不会造成代码的阻塞，但是await会引起async函数内部代码的阻塞。

## const

const保证变量指向的那个内存地址不得改动。

对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。

对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针固定。至于它指向的数据结构是不是可变的，就完全不能控制了。

const foo = {};

// 为 foo 添加一个属性，可以成功

foo.prop = 123;

foo.prop // 123

// 将 foo 指向另一个对象，就会报错

foo = {}; // TypeError: "foo" is read-only

const a = [];

a.push('Hello'); // 可执行

a.length = 0; // 可执行

a = ['Dave']; // 报错

如果真的想将对象冻结，应该使用Object.freeze方法。

const foo = Object.freeze({});

// 常规模式时，下面一行不起作用；

// 严格模式时，该行会报错

foo.prop = 123;

## 变量的解构赋值

**字符串的解构赋值**

let [a, b, c, d]='xyz4';

console.log(a); // x

类似数组的对象都有一个length属性，因此还可以对这个属性解构赋值。

let {length : len} = 'hello';

len // 5

**对象的解构赋值**

let {p1, p2}={p1: 'nihao', p2: 12};

console.log(p1); //nihao

console.log(p2); //12

const { log } = console;

log('hello') // hello

妙啊

let { first, second, third } = { second: "aaa", third: "bbb" };

first: undefined 没有对应的同名属性，则等于undefined。

second: "aaa"

third: "bbb"

如果变量名与对象的属性名不一致，但还想用解构赋值，则必须这么写

let { p1 : value } = { p1: 'aaa'};

value: 'aaa'

p1 is not defined

实际上是想给value赋值，并且要和右面模式匹配

**数组的解构赋值**

let [a, b, c, d] = [3, 5, 9, 12];

console.log(c) // 9

本质是'模式匹配'，只要等号两边模式相同，左边的变量就会被赋予相应的值。

可以用嵌套数组进行解构。虽然我觉得这样写没有任何实际意义

let [foo, [[bar], baz]] = [1, [[2], 3]];

foo // 1

bar // 2

baz // 3

所有解构不成功的都被赋予undefined

还有各种各样的默认值，我实在是不像看了

* 写成这样会报错，所以要非常注意！

let x;

{x} = {x: 1};

大括号写在行首会解析为一个代码块，导致语法错误

迭代器

next()方法返回一个结果对象

{

value: 444, //value返回下一个值

done: false //done是否没有更多数据可以返回。最后一个值返回后再next()，done=true

}

对象字面量

当属性与值的变量同名时。

const name = 'Jane';

const age = 20

// es6

const person = {

name,

age

}

// es5

var person = {

name: name,

age: age

};

计算对象键

var heat = '50%';

var field = 'rock';

var music = {

[field]: heat

}

console.log(music); // Object {rock: "50%"}

## unicode

'\u0061' 'a'

a的ascii码97, 16进制61

'\u5F62' "形"

但是，这种表示法只限于在\u0000~\uFFFF之间的字符，超过FFFF，需要连续两个\u

"\uD842\uDFB7" "𠮷" 55362 57271

Unicode应该可以避免这种的编码，前4位是其他字符的编码吧，比如\uD842是？

ES6之前: \u20BB7 "₻7"

ES6: \u{20BB7} "𠮷" 134071

D83D 55327 DE80 56960 1F680 128640

'\u{1F680}' === '\uD83D\uDE80' //true

## 遍历字符串

let text = String.fromCodePoint(0x20BB7); // "𠮷"

for (let i = 0; i < text.length; i++) {

console.log(text[i]);

}

text.length=2 打印 � �

for (let i of text) {

console.log(i);

}

打印 "𠮷"

for of可以识别大于0xFFFF的码点，传统for循环不能。当然，传统的字就没有超过0xFFFF的

'你好世界'.length 4

'𠮷𠮷𠮷𠮷'.length 8

## 模板字符串

var aaa=12;

document.querySelector('#result').innerHTML=`<div>nihao, ${aaa}</div>`;

<div>你好，12</div>

## 默认参数

function aaa(x, y=6){

}

aaa(3);

aaa()

aaa({})

aaa({x:'fefe'})

aaa({x:'fefe', y:123})

参数和解构赋值一同使用

function aaa({x, y=3}){

console.log(x, y);

}

error

undefined 3

fefe 3

fefe 123

function aaa({x, y=3}={}){

console.log(x, y);

}

undefined 3

undefined 3

fefe 3

fefe 123

function aaa({x, y}={x: 1, y:3}){

console.log(x, y)

}

1 3

undefined undefined

fefe undefined

fefe 123

function aaa({x, y=3}={x: 3, y:6}){

console.log(x, y);

}

3 6

undefined 3

fefe 3

fefe 123

以上有4种写法

* 只要在参数中用到了对象的解构赋值，想要正确的传递一个参数，就必须记住这个形参的名字，然后传一个对象进去。对象的格式为 { 形参名 : 实参 }
* 不管哪种写法，只要包含了所有的参数，就按传进去的为准，不管默认的是啥。比如只要有{x: 12, y:'13'}，x, y就必定是指定的这两个
* 函数接收参数的时候，首先判断是否有默认参数，然后如果传进了参数，再解构赋值
* 一旦发现没有传进参数，就不管结构赋值是什么样的，统一使用默认参数
* 同理，一旦传入了参数，就不管默认参数是什么样的，开始解构赋值。
* 开始解构赋值后，还有可能属性仍然有默认参数。如果传进来的这个参数没有这个属性，而且这个属性有默认参数，则还可以使用默认参数。
* 以写法4为例，如果aaa()，也就是不传参数，则使用默认参数{x: 3, y:6}；如果传了参数，但属性不全，比如传了个{x:'fefe'}，则y=3；如果传个{}，则x=undefined, y=3
* 只要函数参数使用了默认值，解构赋值，扩展运算符，函数内部就不能'use strict'
* 因为函数运行时先执行参数，后函数体，但是执行参数时不知道函数体有没有使用严格模式，所以矛盾。
* 解决方法：

1. 全局'use strict'

2. 用一个立即执行函数包起来

aaa=(function(){

'use strict'

return function(...ar){

//...

return ar;

}

})()

## rest 参数

let m=(var1, ...var2)=>var2;

是一个真正的数组

而arguments不是

## 箭头函数

let m= nihao => nihao;

等同于

let m=function(nihao){

return nihao;

}

箭头函数参数的解构赋值

let m=({var1, var2}) => var1+var2

* 箭头函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。
* 不能当做构造函数
* 没有arguments对象

## 数组

扩展运算符spread ...

相当于rest参数逆运算，把数组转换成逗号分割的参数序列

var aaa=(var1, var2) => console.log(var1, var2);

aaa(...[12, 'qq']); //用作参数时，放在一个数组前面

[...[1, '2'], 3] //[1, 2, 3]

[...document.querySelectorAll('div')] //也可以转换NodeList等任何实现了Iterator的对象

也可以跟表达式

...(x > 0 ? ['a'] : [])

将字符串转化成数组

[...'hello world'] //["h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d"]

三种写法效果相同。apply()可以废弃了

Math.max.apply(null, [14, 3, 77])

Math.max(...[14, 3, 77])

Math.max(14, 3, 77);

* 克隆数组

ES5方法

const a1 = [1, 2];

const a2 = a1.concat();

ES6方法

const a1 = [1, 2];

const a2 = [...a1];

或

const [...a2] = a1;

都是浅拷贝

注意，即使是浅拷贝也是拷贝。和a2=a1是有区别的。

## 模块

export

**export** var m1=1;

var m2='nihao';

var m3=(a)=>console.log(a);

export { m2 as NihaoString, m3}

两种export方法，推荐第二种

as 关键词指定别名

<script type="**module**" src="./foo.js"></script>

默认defer属性开启

代码是在模块作用域之中运行，而不是在全局作用域运行。模块内部的顶层变量外部不可见。

模块脚本自动采用严格模式

## 正则

# webpack

安装最新版本

npm install --save-dev webpack

-D/--save-dev webpack将出现在package.json的devDependencies中

安装 CLI

npm install --save-dev webpack-cli

npx webpack

打包，项目目录生成了一个dist 目录并且创建了一个main.js

* entry：入口文件
* output：出口文件
* module：模块
* plugins：插件
* devServer：服务器（webpack提供的本地服务器）
* mode：模式

两种模式production/development

如果没有创建webpack.config.js 配置文件，指定mode为生产模式或开发模式，打包运行会直接默认生产模式并且进行压缩。

webpack配置文件的默认名称是webpack.config.js或webpackfile.js

手动配置webpack

项目根目录创建webpack.config.js 配置文件

const path = require("path"); //node.js path模块

module.exports = {

mode: "development", //打包模式 development开发模式

entry: "./src/index.js", //入口文件指定

output: {

//出口文件配置 配置出口打包路径

filename: "build.js", //打包后的文件名称

path: path.resolve(\_\_dirname, "build") //resolve绝对路径引入

}

};

path.resolve() 方法将路径或路径片段的序列解析为绝对路径。

\_\_dirname E:\tempDirectory\www 项目的绝对路径

在项目文件里设置配置文件后，再运行

npx webpack

就在创建了一个build目录，生成了build/build.js。

全局安装webpack后，能使用webpack命令，好像效果一样

build.js文件中，包含以下内容

* 一个匿名函数
* installedModules 先定一个缓存，如果当前模块加载完成没有必要再进行加载
* webpack\_require 实现了一个require方法，因为浏览器无法直接执行node的require方法
* return \_\_webpack\_require\_\_(\_\_webpack\_require\_\_.s = "./src/index.js"); 接受入口模块
* 大致意思就是创建了一个module，前面检查了是否是installed/在缓存中，如果不是就记录moduleID

var module = installedModules[moduleId] = {

i: moduleId,

l: false,

exports: {}

};

tm的还能这么写，连等

重命名webpack.config.js => webpack.test.js

npx webpack --config webpack.test.js

另一种执行webpack打包的方法

安装webpck内置服务 webpack-dev-server

npm i webpack-dev-server -d -save

安装完成可以执行 npx webpack-dev-server 按提示打开http://localhost:8080/

如何配置开发服务运行目录可以在配置文件中添加在webpack.config.js添加devServer

输入webpack命令后

1. 先打开根目录下的webpack.config.js
2. 找entry（入口）属性的值
3. 进入到main.js里，看到它又依赖show.js，再找到show.js
4. 把main.js与show.js合并成一个js文件
5. 在webpack.config.js里找到output（出口）属性
6. 解析output里的path与filename属性的值
7. 把第4步合并成的js文件放到dist文件夹里，并起个名字叫bundle.js

## plugins

插件

* html-webpack-plugin

第一步：安装

npm i html-webpack-plugin -D

第二步：在webpack.config.js里引入模块

const HtmlWebpackPlugin=require('html-webpack-plugin');

第三步：在webpack.config.js=>module.exports中添加

plugins:[

new HtmlWebpackPlugin()

]

此时webpack.config.js的内容如下：

const path=require('path');

const HtmlWebpackPlugin=require('html-webpack-plugin');

module.exports={

entry:{

one:'./src/1.js',

two:'./src/2.js'

},

output:{

path:path.resolve(\_\_dirname,'dist'),

filename:'[name].bundle.js'

},

plugins:[

new HtmlWebpackPlugin()

]

}

1.js的内容如下：

console.log('这是第一个入口文件');

2.js的内容如下

console.log('这是第二个入口文件！');