## 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)

function(){}也能写成字符串形式

## json

{

"name" : "liu",

"age" : 19

}

json属性名必须加双引号

string🡪json

JSON.parse()

json🡪string

JSON.stringify()

var obj={

"name" : "liu",

"age" : 12

};

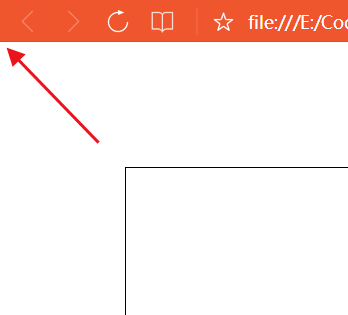
JSON.stringify(obj);

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。

## Event事件

* type：事件的类型，如onlick中的click；
* srcElement/target：事件源，就是发生事件的元素；
* 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)
* 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
* offsetX,offsetY/layerX,layerY：鼠标相对于源元素srcElement左上角的位置；

也比实际情况小……服了

* x,y/pageX,pageY：鼠标相对于文档左上角的位置（没太弄明白）
* screenX、screenY：鼠标相对于显示器左上角的位置

经测试这台电脑screenX比实际小了80，screenY比实际小了120px

* 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：前者是指代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；

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

* && 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

## 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，一个对象的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(f) === f.\_\_proto\_\_; // true

Object.getPrototypeOf(obj)

一个对象实例通过内部属性[[Prototype]]跟踪其原型对象。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。可以调用对象的Object.getPrototypeOf()方法读取[[Prototype]]属性的值，也可以使用isPrototypeOf()方法检查某个对象是否是另一个对象的原型对象。大部分JavaScript引擎在所有对象上都支持一个名为\_\_proto\_\_的属性，该属性可以直接读写[[Prototype]]属性

function f1(){

}

var obj1=new f1();

* f1.prototype

返回

{

constructor:ƒ f1()

\_\_proto\_\_:Object

}

* obj1.constructor

返回函数f1

对象.构造器=函数

* obj1.\_\_proto\_\_

返回{constructor: f}

对象.\_\_proto\_\_=构造器.prototype

* 对象没有prototype，函数的\_\_proto\_\_没有意义

f1.\_\_proto\_\_

返回ƒ () { [native code] }

* Object.getPrototypeOf(obj1)

返回f1.prototype

obj1的constructor是f1，f1的prototype是一个对象p，p的constructor又是f1。但是p的\_\_proto\_\_是object，obj1的\_\_proto\_\_是p

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

变量名以字母、下划线或者$开头

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

## 类型转换

var m= Number('456');

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

var m3=Boolean('fff');

# ES6

**Promise**

我们先看一个最简单的Promise例子：生成一个0-2之间的随机数，如果小于1，则等待一段时间后返回成功，否则返回失败：

function test(resolve, reject) { //接收两个函数作为参数

var timeOut = Math.random() \* 2;

log('set timeout to: ' + timeOut + ' seconds.');

setTimeout(function () {

if (timeOut < 1) {

log('call resolve()...');

resolve('200 OK');//想当于把'200 OK' return给then当参数

}

else {

log('call reject()...');

reject('timeout in ' + timeOut + ' seconds.'); //return给catch作为参数

}

}, timeOut \* 1000);

}

有了执行函数，我们就可以用一个Promise对象来执行它，并在将来某个时刻获得成功或失败的结果：

var p1 = new Promise(test);

var p2 = p1.then(function (result) {

console.log( result);

}); //then的参数是一个函数，这个函数的参数是resolve的参数，resolve在Promise对象绑定的函数里

var p3 = p2.catch(function (reason) {

console.log( reason);

});

变量p1是一个Promise对象，它负责执行test函数。由于test函数在内部是异步执行的，当test函数遇到resolve(param1), 执行then(function(param1))

Promise对象可以串联起来，所以上述代码可以简化为：

new Promise(test).then(function (result) {

console.log('成功：' + result);

}).catch(function (reason) {

console.log('失败：' + reason);

});

(x) => x + 6

相当于

function(x){

return x + 6;

};

(obj) =>{

const a = 999;

return 99;

}

对象的解构赋值

let { foo, bar } = { foo: "aaa", bar: "bbb" };

foo // "aaa"

bar // "bbb"

let { baz } = { foo: "aaa", bar: "bbb" };

baz // undefined

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz // "aaa"

let obj = { first: 'hello', last: 'world' };

let { first: f, last: l } = obj;

f // 'hello'

l // 'world'

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%"}