JS

前言

- 运行在浏览器的脚本语言（目标程序以文本格式打开）
- 用于网页和用户之间的交互，比如提交的时候，进行用户名是否为空的判断
- Java运行在JVM当中，JavaScript运行在浏览器的内存当中
- 完整的javascript由语言基础,BOM,DOM组成

简介

组成部分

组成部分	作用
ECMA Script	构成了JS核心的语法基础（ES规范 / ECMA-262标准）
BOM Browser Object Model	浏览器对象模型，用来操作浏览器上的对象
DOM Document Obiect Model	文档对象模型，用来操作网页中的元素

特点

1. 开发工具简单，记事本即可
2. 无需编译，直接由数据 JS引擎负责执行

3. 弱类型语言 由数据来决定数据类型
4. 面向对象

作用

1. 嵌入动态文本与HTML页面
2. 对浏览器时间做出响应
3. 读写HTML元素
4. 在数据被提交到服务器之前验证数据
5. 检测访客的浏览器信息
6. 控制cookies，包括创建和修改等。
7. 基于node.js技术进行服务器端编程。

JS三种添加方式

行内式

- 事件句柄=“js代码”，把这段代码注册到onclick之后，有操作后，js代码会在浏览器被自动调用
- 弹窗消息的用法是：window.alert(“消息”)
- JS中的一条语句结束之后可以使用分号“;”，也可以不用
- JS中的字符串可以使用双引号，也可以使用单引号

```
<button onclick="alert('行内js')">单击试试</button>

<!--
1、JS是一门事件驱动型的编程语言，依靠事件去驱动，然后执行对应的程序。
在JS中有很多事件，其中有一个事件叫做：鼠标单击，单词：click。并且任何
事件都会对应一个事件句柄叫做：onclick。【注意：事件和事件句柄的区别是：
事件句柄是在事件单词前添加一个on。】，而事件句柄是以HTML标签的属性存在的。

2、onclick="js代码"，执行原理是什么?
页面打开的时候，js代码并不会执行，只是把这段JS代码注册到按钮的click事件上了。
等这个按钮发生click事件之后，注册在onclick后面的js代码会被浏览器自动调用。

3、怎么使用JS代码弹出消息框?
在JS中有一个内置的对象叫做window，全部小写，可以直接拿来使用，window代表的是浏览器对象。
window对象有一个函数叫做:alert，用法是：window.alert("消息");这样就可以弹窗了。
-->
```

内嵌式

通过脚本块的方式，页面打开的时候执行，并且遵守自上而下的顺序依次逐行执行。（这个代码的执行不需要事件）。（CSS为样式块）

- javascript的脚本块在一个页面当中可以出现多次。没有要求
- javascript的脚本块出现位置也没有要求，随意
- alert有阻塞当前页面加载的作用。（阻挡，直到用户点击确定按钮。）

```
<script>
alert('内嵌js');
</script>
```

外链式

外部引入js文件

- 同一个js文件可以被引入多次，但实际开发中这种需求很少
- 引入js文件的同时再写代码，文件会被执行，但代码块不会被执行。但分别两个script，一个引入一个写代码是可以的

```
<script src="js文件路径地址">这里不能写js语句</script>
```

变量

变量的声明与赋值

- java中要求变量声明的时候是什么类型，不可变。编译期强行固定变量的数据类型称为强类型语言。数据类型 变量名;
- 对比javascript，javascript是一种弱类型语言，没有编译阶段，一个变量可以随意赋值，赋什么类型的值都行，var 变量名;
- 当系统没有赋值的时候，会默认给undefined，undefined是系统的一个存在值
- 当系统直接没声明直接调用一个值，会报错

```
<script type="text/javascript">
        var a, b, c = 200;
        alert("a = " + a);
        alert("b = " + b);
        alert("c = " + c);

        a = false;
        alert(a);

        a = "abc";
        alert(a);

        a = 1.2;
        alert(a);
</script>
```

变量命名规范

1. 只能由字母、数字、_（下划线）、$（美元符号）组成。
2. 不能以数字开头。
3. 命名中不能出现-（js会理解成减号进行减法的操作），不能和关键字冲突。

数据类型

数据类型有：原始类型、引用类型

- 原始类型：undefined、number、string、boolean、null
- 引用类型：object以及object的子类

```
var i;
alert(typeof i); // "undefined"

var k = 10;
alert(typeof k); // "number"

var f = "abc";
alert(typeof f); // "string"

var d = null;
alert(typeof d); // "object"  null属于Null类型,但是typeof运算符的结果是"object"

var flag = false;
alert(typeof flag); // "boolean"

var obj = new Object();
alert(typeof obj); // "object"
```

```
// sayHello是一个函数.
function sayHello(){

}
alert(typeof sayHello); // "function"
```

检测数据类型

typeof(value); 或 typeof value; 返回这个变量的类型

说明：同一个变量, 可以进行不同类型的数据赋值。

在JS当中比较字符串是否相等使用“==”完成。没有equals

```
typeof运算符的语法格式:
        typeof 变量名

function sum(a, b){
   if(typeof a == "number" && typeof b == "number"){
           return a + b;
   }
   alert(a + "," + b + "必须都为数字! ");
}
```

变量提升

javascript并不是严格的自上而下执行的语言 。它会将当前作用域的所有变量的声明提升到程序的顶部。

细讲数据类型

undefined

当一个变量没有手动赋值，系统默认赋值undefined，或者也可以给一个变量手动赋值undefined

```
var i; // undefined
var k = undefined; // undefined
```

```
alert(i == k); // true

var y = "undefined"; // "undefined"是一个字符串类型
alert(y == k); // false
```

number

Number类型包括整数、小数、正数、负数、不是数字、无穷大等。比如 -1 0 1 2 2.3 3.14 100 … NaN Infinity等都属于Number

```
// Infinity (当除数为0的时候，结果为无穷大)
alert(10 / 0);
```

补充关于NaN（(表示Not a Number，不是一个数字，但属于Number类型)） 运算结果本来应该是一个数字,最后算完不是一个数字的时候,结果是NaN

isNaN函数 （is Not a Number）

isNaN(数据)，结果是true表示不是一个数字， 结果是false表示是一个数字

```
function sum(a, b){
   if(isNaN(a) || isNaN(b)){
            alert("参与运算的必须是数字！");
            return;
   }
   return a + b;
}
sum(100, "abc");
alert(sum(100, 200));
```

parseInt()或者parseFloat()函数

可以将字符串自动转换成数字，并且取整数位，同理parseFloat也一样

```
alert(parseInt("3.9999")); // 3
alert(parseInt(3.9999)); // 3

// parseFloat():可以将字符串自动转换成数字.
alert(parseFloat("3.14") + 1); // 4.14
alert(parseFloat("3.2") + 1); // 4.2
```

Math.ceil()函数

向上取整

```
// Math.ceil()
alert(Math.ceil("2.1")); // 3
```

boolean

对应true和false

```
// 规律:“有"就转换成true,"没有"就转换成false.
alert(Boolean(1)); // true
alert(Boolean(0)); // false
alert(Boolean("")); // false
alert(Boolean("abc")); // true
alert(Boolean(null)); // false
alert(Boolean(NaN)); // false
alert(Boolean(undefined)); // false
alert(Boolean(Infinity)); // true
```

Boolean()函数

Boolean()函数的作用是将非布尔类型转换成布尔类型

```
语法格式:
                Boolean(数据)
```

null

Null类型是第二个只有一个值的类型，这个特殊值就是null，从逻辑的角度看，null值表示一个空对象指针，而这也正是typeof检测null值时会返回object的原因。

```
// Null类型只有一个值,null
alert(typeof null); // "object"
```

String

在JS当中字符串可以使用单引号，也可以使用双引号

```
var s1 = 'abcdef';
var s2 = "test";
```

两种创建对象的方式，但创建的对象的类型有所不同 String是一个内置的类，可以直接用，String的父类是Object

```
//第一种:
var s = "abc";
//第二种（使用JS内置的支持类String）:
var s2 = new String("abc");

// 小string(属于原始类型String)
var x = "king";
alert(typeof x); // "string"

// 大String(属于Object类型)
var y = new String("abc");
alert(typeof y); // "object"
```

函数名	功能
indexOf	获取指定字符串在当前字符串中第一次出现处的索引
lastIndexOf	获取指定字符串在当前字符串中最后一次出现处的索引
replace	替换
substr	截取子字符串（下标，长度）
substring	截取子字符串【开始下标，结束下标）
toLowerCase	转换小写
toUpperCase	转换大写
split	拆分字符串

Object

Object类型是所有类型的超类，自定义的任何类型，默认继承Object

js定义类的方式

```
定义类的语法:
第一种方式:
function 类名(形参){
```

```
}
第二种方式:
类名 = function(形参){

}
第三种方式:
class Persion{
      constructor(name){
        this.name =name;
      }
      fn() {
        alert('Persion');
      }
    }
class User extends Persion{
      constructor(name){
        super(name);
      }
      ufn() {
        //super为People原型对象
        //People.prototype.fn.call(this);
        super.fn();
      }
    }
创建对象的语法:
new 构造方法名(实参); // 构造方法名和类名一致。
```

js的类定义以及探讨

```
// 定义一个学生类
function Student(){
   alert("Student.....");
}

// 当做普通函数调用，方法三不适用
Student();

// 当做类来创建对象
var stu = new Student();
alert(stu); // [object Object]
```

JS中的类的定义，同时又是一个构造函数的定义 在JS中类的定义和构造函数的定义是放在一起来完成的 js的函数形参调用可以有一个或者多个都可以执行，因为是弱类型

```
function User(a, b, c){ // a b c是形参,属于局部变量.
   // 声明属性 (this表示当前对象)
   // User类中有三个属性:sno/sname/sage
```

```
    this.sno = a;
    this.sname = b;
    this.sage = c;
}

// 创建对象
var u1 = new User(111, "zhangsan", 30);
// 访问对象的属性
alert(u1.sno);
alert(u1.sname);
alert(u1.sage);

var u2 = new User(222, "jackson", 55);
alert(u2.sno);
alert(u2.sname);
alert(u2.sage);

// 访问一个对象的属性,还可以使用这种语法
alert(u2["sno"]);
alert(u2["sname"]);
alert(u2["sage"]);
```

也可以换种方式定义类

```
Product = function(pno,pname,price){
    // 属性
    this.pno = pno;
    this.pname = pname;
    this.price = price;
    // 函数
    this.getPrice = function(){
            return this.price;
    }
}

var xigua = new Product(111, "西瓜", 4.0);
var pri = xigua.getPrice();
alert(pri); // 4.0
```

属性有：prototype属性（常用的，主要是这个）：作用是给类动态的扩展/挂载属性和函数。constructor属性

```
// 可以通过prototype这个属性来给类动态扩展属性以及函数
Product.prototype.getPname = function(){
    return this.pname;
}

// 调用后期扩展的getPname()函数
```

```
var pname = xigua.getPname();
alert(pname)

// 给String扩展一个函数
String.prototype.suiyi = function(){
   alert("这是给String类型扩展的一个函数，叫做suiyi");
}

"abc".suiyi();
```

对比一下java类型定义与js类型定义

java语言怎么定义类，怎么创建对象？（强类型）

```
public class User{
        private String username;
        private String password;
        public User(){

        }
        public User(String username,String password){
                this.username = username;
                this.password = password;
        }
}
User user = new User();
User user = new User("lisi","123");
```

JS语言怎么定义类，怎么创建对象？（弱类型）

```
User = function(username,password){
        this.username = username;
        this.password = password;
}
var u = new User();
var u = new User("zhangsan");
var u = new User("zhangsan","123");

或

class Persion{
      constructor(name){
        this.name =name;
      }
      fn() {
        alert('Persion');
      }
    }
```

null、NaN和undefined

- 数据类型不一致
- ==(等同运算符：只判断值是否相等) ===(全等运算符：既判断值是否相等，又判断数据类型是否相等)

```
// null NaN undefined 数据类型不一致.
alert(typeof null); // "object"
alert(typeof NaN); // "number"
alert(typeof undefined); // "undefined"

// null和undefined可以等同.
alert(null == NaN); // false
alert(null == undefined); // true
alert(undefined == NaN); // false

// 在JS当中有两个比较特殊的运算符
alert(null === NaN); // false
alert(null === undefined); // false
alert(undefined === NaN); // false
```

函数（特殊的对象）

具名函数的定义与使用

函数类似java中的方法，java中定义方法的格式是

```
[修饰符列表] 返回值类型 方法名(形式参数列表){
        方法体;
}

public static boolean login(String username,String password){
        ...
        return true;
}

boolean loginSuccess = login("admin","123");
```

而js是一种弱类型，js中的函数不需要 指定返回值类型，返回什么类型都行，函数的定义格式是

```
//第一种方式:
function 函数名(形式参数列表){
         函数体;
}
//第二种方式:
函数名 = function(形式参数列表){
        函数体;
}

function sum(a, b){
// a和b都是局部变量,他们都是形参(a和b都是变量名，变量名随意。)
        alert(a + b);
}

sum(10,20);//函数必须调用才能执行

//或者第二个格式
// 定义函数sayHello
sayHello = function(username){
        alert("hello " + username);
}

// 调用函数
sayHello("zhangsan");
```

配合单击按钮框的逻辑完整代码如下

```
<!DOCTYPE html>
<html>
        <head>
                <meta charset="utf-8">
                <title>JS函数初步</title>
        </head>
        <body>
                <script type="text/javascript">

                    function sum(a, b){
                            // a和b都是局部变量,他们都是形参(a和b都是变量名，变量名随
意。)
                            alert(a + b);
                    }

                    sayHello = function(username){
                            alert("hello " + username);
                    }

                    // 调用函数
                    sayHello("zhangsan");
```

```
                </script>

                <input type="button" value="hello" onclick="sayHello('jack');" />
                <input type="button" value="计算10和20的求和" onclick="sum(10, 20);"
/>

        </body>
</html>
```

匿名函数(与匿名内部类相似)

定义：即没有名字的函数

注意：

1. var init = function() { }是函数表达式，函数在代码执行的到当前行的时候才被执行，init 才被赋值。
2. function init() {} 是函数的声明，和 var 一样，会被提前到代码最前面定义。
3. 添加后再移除事件要用具名函数

```
function(形式参数){函数体}
定义函数并赋值给变量：var fn = function(形式参数){函数体}

调用方式：将匿名函数赋值给一个变量，通过变量名调用函数
调用函数：fn(实际参数);

<script type="text/javascript">

    // 匿名函数 ：即没有名称的函数
    var func = function(i, u) {
        alert(i + " 喜欢 " + u);
    }

    // 调用函数 ：
   func("我", "你");//显示 我喜欢你

</script>
```

自执行函数

没有名字的函数会报错，这时候用一个括号把它包起来就不会报错了,然后在最后面加一个括号就可以马上执行这个函数了—— **直接调用** ， 也叫 **自执行函数** 。

作用： 避免多人开发时变量冲突，自执行函数只能调用一次。

```
//参数直接写在括号里
//写法1
(function (sum){
        console.log("JavaScript"+sum);
})("你好")
//输出内容为  JavaScript你好

//写法2
~function(){
        console.log('立即执行匿名函数! ')
}()

//写法3
!function(){
        console.log('立即执行匿名函数! ');
}()
```

箭头函数（跟lambda相似）

箭头函数表面上相当于匿名函数，并且简化了函数定义。

```
(x,y) => {
        return x + y;
}
```

函数参数

JavaScript 函数对参数的值没有进行任何的检查。

函数显式参数(Parameters)和隐式参数(Arguments)

函数显式参数在函数定义时列出。

函数隐式参数在函数调用时传递给函数真正的值。

```
function Name(parameter1, parameter2, parameter3) {
    // 要执行的代码……
}
```

参数规则

JavaScript 函数定义显式参数时没有指定数据类型。

JavaScript 函数对隐式参数没有进行类型检测。

JavaScript 函数对隐式参数的个数没有进行检测。

默认参数

ES5 中如果函数在调用时未提供隐式参数，参数会默认设置为： **undefined**

有时这是可以接受的，但是建议最好为参数设置一个默认值：

```
function myFunction(x, y) {
    y = y || 0;
}
//如果 y 已经定义，y || 0 返回 y，因为 y 是 true，否则返回 0，因为 undefined 为 false。
```

ES6 函数可以自带参数

ES6 支持函数带有默认参数，就判断 undefined 和 || 的操作：

```
function myFunction(x, y = 10) {
    // y is 10 if not passed or undefined
    return x + y;
}

myFunction(0, 2) // 输出 2
myFunction(5); // 输出 15, y 参数的默认值
```

arguments 对象

JavaScript 函数有个内置的对象 arguments 对象。

arguments 对象包含了函数调用的参数数组。

通过这种方式你可以很方便的找到最大的一个参数的值：

```
x = sumAll(1, 123, 500, 115, 44, 88);

function sumAll() {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i++) {
        sum += arguments[i];
```

```
    }
    return sum;
}
```

通过值传递参数（基本数据类型）

在函数中调用的参数是函数的隐式参数。

JavaScript 隐式参数通过值来传递：函数仅仅只是获取值。

如果函数修改参数的值，不会修改显式参数的初始值（在函数外定义）。

隐式参数的改变在函数外是不可见的。

通过对象传递参数（地址）

在JavaScript中，可以引用对象的值。

因此我们在函数内部修改对象的属性就会修改其初始的值。

修改对象属性可作用于函数外部（全局变量）。

修改对象属性在函数外是可见的。

什么是重载?

相同函数名，不同形参列表的多个函数，可以在调用时根据传入的实参值的不同，执行不同的逻辑。

好处：减少函数名的个数，减轻调用者的负担

但在JS当中，函数的名字不能重名，当函数重名的时候，后声明的函数会将之前声明的同名函数覆盖，无法像java一样进行直接重载

第一种重载实现

这种方法比较简单，给一个思路，大家肯定都能理解，就是函数内部用if或switch语句，根据传入参数的个数调用不同的case语句，从而功能上达到重载的效果。

```
<script language="JavaScript">
function f(length,width)
{
```

```
    var len= arguments.length;
    if(2== len)
    {
        var width = arguments[1];
        alert("高为: "+length+", 宽为: "+width);
    }
    else
    {
        alert("高为: "+length);
    }
}
</srcipt>
```

这种方法简单粗暴。但是对于一个正在学习js的人来说，这种方法未免太敷衍了。

第二种重载实现

是第一种方法的增强

addMethod方法用来给一个对象添加自定义方法，能够接收三个参数：

1. 需要添加方法的对象
2. 自定义方法名
3. 定义方法具体要实现的功能，通过回调实现

```
function addMethod(object, name, fn) {
        var old = object[name];
        object[name] = function () {
          console.log(arguments);
          console.log(fn.length);
          if (fn.length === arguments.length) {
            return fn.apply(this, arguments);
          } else if (typeof old === "function") {
            return old.apply(this, arguments);
          }
        }
        console.log(object[name]);
      }


      var people = {
        values: ["Dean Edwards", "Alex Russell", "Dean Tom"]
      };

      /* 下面开始通过addMethod来实现对people.find方法的重载 */

      // 不传参数时，返回peopld.values里面的所有元素
      addMethod(people, "find", function () {
        return this.values;
      });
```

```
// 传一个参数时，按first-name的匹配进行返回
addMethod(people, "find", function (firstName) {
  var ret = [];
  for (var i = 0; i < this.values.length; i++) {
    if (this.values[i].indexOf(firstName) === 0) {
      ret.push(this.values[i]);
    }
  }
  return ret;
});

// 传两个参数时，返回first-name和last-name都匹配的元素
addMethod(people, "find", function (firstName, lastName) {
  var ret = [];
  for (var i = 0; i < this.values.length; i++) {
    if (this.values[i] === (firstName + " " + lastName)) {
      ret.push(this.values[i]);
    }
  }
  return ret;
});

// 测试:
console.log(people.find()); //["Dean Edwards", "Alex Russell", "Dean Tom"]
console.log(people.find("Dean")); //["Dean Edwards", "Dean Tom"]
console.log(people.find("Dean", "Edwards")); //["Dean Edwards"]*/
```

全局变量和局部变量

- 在函数体之外声明的变量属于全局变量；在函数体当中声明的变量，包括一个函数的形参都属于局部变量
- 全局变量的生命周期是浏览器打开时声明，浏览器关闭时销毁，尽量少用。因为全局变量会一直在浏览器的内存当中，耗费内存空间
- 局部变量的生命周期是：函数开始执行时局部变量的内存空间开辟，函数执行结束之后，局部变量的内存空间释放，局部变量生命周期较短

```
// 全局变量
var username = "jack";
function accessUsername(){
// 局部变量
 var username = "lisi";
// 就近原则:访问局部变量
 alert("username = " + username);
 }
 // 调用函数
 accessUsername();
// 访问全局变量
```

```
 alert("username = " + username);
```

先输出lisi后输出jack 因为局部变量结束后便释放了，所以局部变量没有值 而如果单纯没有定义全局变量，全在全局中输出局部变量那会报错

```
function accessAge(){
var age = 20;
alert("年龄 = " + age);
}

accessAge();

// 报错(语法不对)
 alert("age = " + age);
```

如果一个变量在声明的时候没有定义var，默认是全局变量（即使在局部变量中声明）

```
// 以下语法是很奇怪的.
function myfun(){
// 当一个变量声明的时候没有使用var关键字,那么不管这个变量是在哪里声明的,都是全局变量.
myname = "dujubin";
 }

// 访问函数
myfun();

alert("myname = " + myname); // myname = dujubin
```

JavaScript 没有块级作用域（ES6 之前）。

在 ES6 之前，是没有块级作用域的概念的。ES6 可以使用 let 关键字来实现块级作用域。

这一点也是JavaScript相比其它语言较灵活的部分。仔细观察下面的代码，你会发现变量i、j、k作用域是相同的，他们在整个rain函数体内都是全局的。

对于有块级作用域的语言来说，for语句初始化变量的表达式所定义的变量，只会存在于循环的环境中。而对于JavaScript来说，for语句创建的变量 i 即使在for循环执行结束之后，依旧存在于循环外部的执行环境之中。

```
<script type="text/javascript">
    function rainman(){
        // rainman函数体内存在三个局部变量 i j k
        var i = 0;
        if ( 1 ) {
            var j = 0;
            for(var k = 0; k < 3; k++) {
                alert( k );    //分别弹出 0 1 2
            }
            alert( k );        //弹出3
        }
        alert( j );            //弹出0
    }
</script>
```

var、let和const的区别

1. 使用var声明的变量，其作用域为该语句所在的函数内，且存在变量提升现象；
2. 使用let声明的变量，其作用域为该语句所在的代码块（所在花括号里）内，不存在变量提升；
3. const 用于声明一个或多个常量，声明时必须进行初始化，且初始化后值不可再修改：
4. let不允许在相同作用域内重复声明同一个变量。
5. const定义常量（必须初始化）与使用 let 定义的变量相似：二者都是块级作用域，都不能和它所在作用域内的其他变量或函数拥有相同的名称
6. const 的本质: const 定义的变量并非常量，并非不可变，它定义了一个常量引用一个值。使用 const 定义的对象或者数组，其实是可变的。

经典案例

```
<script type="text/javascript">
        //试验代码
        for(var i=0;i<5;i++){
            console.log(i);
        }
        console.log(i);
    </script>
//0 1 2 3 4 5

<script type="text/javascript">
        //试验代码
        for(let i=0;i<5;i++){
            console.log(i);
        }
```

```
        console.log(i);
    </script>
//报错

const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"];    // 错误

// 创建常量对象
const car = {type:"Fiat", model:"500", color:"white"};
// 修改属性:
car.color = "red";
// 添加属性
car.owner = "Johnson";
```

运算符

逗号运算符

使用逗号可以在一条语句中执行多次操作

使用逗号运算符分隔的语句会 从左到右 依次执行

```
var age1=16,age2=17,age3=18;
```

算术运算符

```
 + -    *       /       %       ++      --

<script>

    alert(1234 / 1000 * 1000); // 1234

    var s = "12";
     s -= 10;
    alert(s);  // 2

    var s = "aa";
    s -= 10;
    alert(s);  // NaN       Not a Number 不是一个数字

    var s = "12";
    s += 10;
    alert(s);       // 1210
```

```
</script>
```

注意：
- js中的小数和整数都是number类型，不存在整数除以整数还是整数的结论。
  - 字符串和其他的数据使用+号运算，会连接成一个新的字符串。
  - 字符串使用除了+以外的运算符：如果字符串本身是一个数字，那么会自动转成number进行运算，否则就会返回一个NaN的结果，表示这不是一个数字。NaN：not a number

关系运算符

```
>               >=              <               <=      !=

<script>

    // 请问：2 > 5，结果为 ？
    alert(2 > 5);   // false

    // 请问：“22” == 22  结果为 ？
    alert("22" == 22); // true  （仅仅判断数值）

    // 请问：“22” === 22  结果为 ？
    alert("22" === 22);  // false  （恒等于，数值和类型都要相等）

</script>
```

逻辑运算符

```
&&              逻辑与          true&&false             ====>false
||              逻辑或          true||false             ====>true
!               逻辑非          !true                   ====>false

针对 && ：有一个假即为假。
针对 || ：有一个真即为真。

true（理解）：  true，  非0，  非null，  非undefined
false（理解）：false，  0，   null，    undefined

<script>
    // 请问 1:  8 < 7 && 3 < 4,结果为 ？
    alert(8 < 7 && 3 < 4);  // false

    // 请问 2:  -2 && 6 + 6 && null 结果为 ？
    alert(-2 && 6 + 6 && null); // null

    // 请问 3:  1 + 1 && 0 && 5  结果为 ？
    alert(1 + 1 && 0 && 5); // 0
```

```
    // 请问1 :  0 || 23 结果为 ?
    alert(0 || 23); // 23

    // 请问2 :  0 || false || true  结果为 ?
    alert(0 || false || true); // true

    // 请问3 :  null || 10 < 8 || 10 + 10 结果为 ?
    alert(null || 10 < 8 || 10 + 10);  // 20

    // 请问4 :  null || 10 < 8 || false 结果为 ?
    alert(null || 10 < 8 || false); // false

</script>
```

三目运算符

```
条件?表达式1:表达式=2
如果条件为true，返回表达式1的结果
如果条件为false，返回表达式2的结果
<script>
    // 请问1 :  var score=80 >= 60 ? "合格" : "不合格"  结果为 ?
    alert(var score=80 >= 60 ? "合格" : "不合格");  // 合格

    // 请问2 :  1 > 5 ? "是的" : "不是" 结果为 ?
    alert(1 > 5 ? "是的" : "不是");       // 不是
</script>
```

赋值运算符

```
运算符     x=10 y=5       结果
=             x=y                  x=5
+=          x+=y     x=x+y        x=15
-=          x-=y     x=x-y        x=5
*=          x*=y     x=x*y        x=50
/=          x/=y     x=x/y        x=2
%=          x%=y     x=x%y        x=0
```

void运算符

立即调用的函数表达式

```
void function fn(){
    console.log(123)
}()
```

在a链接中阻止跳转

```
<a href="javascript:void(0);"></a>
```

void(0)会返回undefined，这个链接点击之后不会做任何事情，此时就 禁止了页面跳转。但是这样不是所有浏览器都兼容的，很多时候我们用 href="#"

两者都是阻止页面跳转，href="#"执行的时候会在地址栏后面添加#号，还会让页面的滚动条滚动到 页面的最上面。

注意：利用 javascript: 伪协议来执行js代码是不推荐的，利用 href="#"也有地址栏添加 # 号的问题，推荐的做法是为链接元素 绑定事件。

在箭头函数中

箭头函数标准中，当函数返回值是一个不会被使用到的时候，应该使用 void运算符，来确保返回 undefined。

```
const fn = () => void doSomething();
```

判断值是否为undefined中

在实际开发中，我们判断一个值为 undefined的时候，会第一时间想到 txt === undefined的例子，但是这样做其实是有bug的，如果一个全局变量也叫 undefined，那么此时会发生判断错误，正确的写法应该是 txt === void(0)

```
if(txt === undefined)
if(txt === void(0))
```

控制语句

与java结构一致，不赘述了

js事件

常用事件

事件	描述
onchange	HTML 元素改变
onclick	用户点击 HTML 元素
onmouseover	鼠标指针移动到指定的元素上时发生
onmouseout	用户从一个 HTML 元素上移开鼠标时发生
onkeydown	用户按下键盘按键
onload	浏览器已完成页面的加载

注册事件方式

回调函数的特点:由其他程序负责调用该函数

第一种注册方式： 直接在标签中使用事件句柄

```
<script>
function sayHello(){
   alert("hello js!");
}
</script>

<!--以下代码的含义是：将sayHello函数注册到按钮上，等待click事件发生之后，该函数被浏览器调用。我们称这个函数为回调函数。-->
<input type="button" value="hello" onclick="sayHello()"/>
```

第二种注册方式： 使用js代码完成事件注册

```
<input type="button" value="hello4" id="mybtn2" />
<script type="text/javascript">
        function doSome(){
                alert("do some!");
        }
 // 第一步:先获取这个按钮对象(document是全部小写，内置对象，可以直接用，document就代表整个HTML页面)
```

```
var btnObj = document.getElementById("mybtn");
// 第二步:给按钮对象的onclick属性赋值
btnObj.onclick = doSome; // 注意:别加小括号. btnObj.onclick = doSome();这是错误的写法.
// 这行代码的含义是,将回调函数doSome注册到click事件上.
</script>
```

第三种注册方式：使用匿名函数（非必须，期望使用前两种）

```
var mybtn1 = document.getElementById("mybtn");
mybtn1.onclick = function(){ // 匿名函数,这个匿名函数也是一个回调函数.
   alert("test........."); // 这个函数在页面打开的时候只是注册上,不会被调用,在click事件发生之后才会调用.
}
```

代码执行顺序

```
<script type="text/javascript">

// 第一步:根据id获取节点对象
var btn = document.getElementById("btn"); // 返回null(因为代码执行到此处的时候id="btn"的元素还没有加载到内存)

// 第二步:给节点对象绑定事件
btn.onclick = function(){
        alert("hello js");
}



</script>

<input type="button" value="hello" id="btn" />
```

执行错误，返回了null，因为还未获取到id元素，但将顺序反过来偶尔会忘记，所以我们添加上面的一个函数load()，页面加载完的时候才会发生

```
<script>
window.onload = function(){
        document.getElementById("id属性").onclick = function(){
                //获得属性后的 利用属性执行函数
        }
}
```

```
</script>
<input type="button" value="框中的值" id="id属性" />
```

设置节点的属性

通过点击一个按钮狂，将其变为复选框

```
<script type="text/javascript">
        window.onload = function(){
                document.getElementById("btn").onclick = function(){
                        var mytext = document.getElementById("mytext");
                        // 一个节点对象中只要有的属性都可以"."
                        mytext.type = "checkbox";
                }
        }
</script>

<input type="text" id="mytext"/>

<input type="button" value="将文本框修改为复选框" id="btn"/>
```

捕捉回车键

回车键的键值是13、ESC键的键值是27，按钮键是onclick，而回车键onkeydown，回调函数的参数可以有，有与没有都会调用回调函数

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<input type="text" id="mytext">

<script type="text/javascript">
    window.onload = function (){
        document.getElementById("mytext").onkeydown = function (event){
            if (event.keyCode == 13)
            alert("点击了回车！");
        }
    }
</script>
```

```
</body>
</html>
```

严格模式

JavaScript 严格模式（strict mode）即在严格的条件下运行（严格遵守语法规则）。

严格模式下你不能使用未声明的变量。

严格模式通过在脚本或函数的头部添加 **use strict**; 表达式来声明。

```
x = 3.14;       // 不报错
myFunction();

function myFunction() {
   "use strict";
    y = 3.14;   // 报错 (y 未定义)
}
```

js 表单

HTML 表单验证可以通过 JavaScript 来完成。

以下实例代码用于判断表单字段(fname)值是否存在， 如果不存在，就弹出信息，阻止表单提交：

```
<script>
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == null || x == "") {
        alert("需要输入名字。");
        return false;
    }
}
</script>
</head>
<body>

<form name="myForm" action="demo_form.php" onsubmit="return validateForm()"
method="post">
名字: <input type="text" name="fname">
```

```
<input type="submit" value="提交">
</form>
```

属性	描述
disabled	规定输入的元素不可用
max	规定输入元素的最大值
min	规定输入元素的最小值
pattern	规定输入元素值的模式（正则表达式）
required	规定输入元素字段是必需的
type	规定输入元素的类型

this

面向对象语言中 this 表示当前对象的一个引用。

但在 JavaScript 中 this 不是固定不变的，它会随着执行环境的改变而改变。

- 在方法中，this 表示该方法所属的对象。
- 如果单独使用，this 表示全局对象。
- 在函数中，this 表示全局对象。
- 在函数中，在严格模式下，this 是未定义的(undefined)。
- 在事件中，this 表示接收事件的元素。
- 类似 call() 和 apply() 方法可以将 this 引用到任何对象。

同名全局变量和局部变量

出现同名全局变量和局部变量时，调用全局变量，需要将全局this赋给this_(重命名)，在局部使用this_调用即可

```
<script>
var lastName = "Doe123";
this_ = this
// 创建一个对象
var person = {
  firstName: "John",
  lastName : "Doe",
  id     : 5566,
```

```
  fullName : function() {
    return this.firstName + " " + this_.lastName;
  }
};

// 显示对象的数据
document.getElementById("demo").innerHTML = person.fullName();
</script>
```

事件中的this

```
<button onclick="this.style.display='none'">点我后我就消失了</button>
```

显式函数绑定

在 JavaScript 中函数也是对象，对象则有方法，apply （参数为数组）和 call（参数为列表项） 就是函数对象的方法。这两个方法异常强大，他们允许切换函数执行的上下文环境（context），即 this 绑定的对象。（将一个对象的方法，提供给其他对象使用）

在下面实例中，当我们使用 person2 作为参数来调用 person1.fullName 方法时, **this** 将指向 person2, 即便它是 person1 的方法：

```
var person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}
var person2 = {
  firstName:"John",
  lastName: "Doe",
}
person1.fullName.call(person2);  // 返回 "John Doe"
```

js异步编程

在前端编程中（甚至后端有时也是这样），我们在处理一些简短、快速的操作时，例如计算 1 + 1 的结果，往往在主线程中就可以完成。主线程作为一个线程，不能够同时接受多方面的请求。所以，当一个事件没有结束时，界面将无法处理其他请求。

现在有一个按钮，如果我们设置它的 onclick 事件为一个死循环，那么当这个按钮按下，整个网页将失去响应。

为了避免这种情况的发生，我们常常用子线程来完成一些可能消耗时间足够长以至于被用户察觉的事情，比如读取一个大文件或者发出一个网络请求。因为子线程独立于主线程，所以即使出现阻塞也不会影响主线程的运行。但是子线程有一个局限：一旦发射了以后就会与主线程失去同步，我们无法确定它的结束，如果结束之后需要处理一些事情，比如处理来自服务器的信息，我们是无法将它合并到主线程中去的。

回调函数

回调函数就是一个函数，它是在我们启动一个异步任务的时候就告诉它：等你完成了这个任务之后要干什么。这样一来主线程几乎不用关心异步任务的状态了，他自己会善始善终。

```
setTimeout(function () {
    document.getElementById("demo").innerHTML="RUNOOB!";
}, 3000);
```

这段程序中的 setTimeout 就是一个消耗时间较长（3 秒）的过程，它的第一个参数是个回调函数，第二个参数是毫秒数，这个函数执行之后会产生一个子线程，子线程会等待 3 秒，然后执行回调函数 "print"，在命令行输出 "RUNOOB!"。

异步 AJAX

除了 setTimeout 函数以外，异步回调广泛应用于 AJAX 编程。（简单演示）

```
//js
var xhr = new XMLHttpRequest();

xhr.onload = function () {
    // 输出接收到的文字数据
    document.getElementById("demo").innerHTML=xhr.responseText;
}

xhr.onerror = function () {
    document.getElementById("demo").innerHTML="请求出错";
}

// 发送异步 GET 请求
xhr.open("GET", "https://www.runoob.com/try/ajax/ajax_info.txt", true);
```

```
xhr.send();

//使用jq库
$.get("https://www.runoob.com/try/ajax/demo_test.php",function(data,status){
    alert("数据: " + data + "\n状态: " + status);
});
```

JS Promise

Promise 是一个 ECMAScript 6 提供的类，目的是更加优雅地书写复杂的异步任务。

构造 Promise

现在我们新建一个 Promise 对象：

```
new Promise(function (resolve, reject) {
    // 要做的事情...
});
```

如果我想分三次输出字符串，第一次间隔 1 秒，第二次间隔 4 秒，第三次间隔 3 秒：

失败的写法

```
setTimeout(function () {
    console.log("First");
    setTimeout(function () {
        console.log("Second");
        setTimeout(function () {
            console.log("Third");
        }, 3000);
    }, 4000);
}, 1000);
```

这段程序实现了这个功能，但是它是用 "函数瀑布" 来实现的。可想而知，在一个复杂的程序当中，用 "函数瀑布" 实现的程序无论是维护还是异常处理都是一件特别繁琐的事情，而且会让缩进格式变得非常冗赘。

成功的写法

```
new Promise(function (resolve, reject) {
    setTimeout(function () {
        console.log("First");
        resolve();
    }, 1000);
}).then(function () {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log("Second");
            resolve();
        }, 4000);
    });
}).then(function () {
    setTimeout(function () {
        console.log("Third");
    }, 3000);
});
```

Promise 的构造函数

Promise 构造函数是 JavaScript 中用于创建 Promise 对象的内置构造函数。

Promise 构造函数接受一个函数作为参数，该函数是同步的并且会被立即执行，所以我们称之为**起始函数**。起始函数包含两个参数 resolve 和 reject，分别表示 Promise 成功和失败的状态。

起始函数执行成功时，它应该调用 resolve 函数并传递成功的结果。当起始函数执行失败时，它应该调用 reject 函数并传递失败的原因。

Promise 构造函数返回一个 Promise 对象，该对象具有以下几个方法：

- then：用于处理 Promise 成功状态的回调函数。
- catch：用于处理 Promise 失败状态的回调函数。
- finally：无论 Promise 是成功还是失败，都会执行的回调函数。

下面是一个使用 Promise 构造函数创建 Promise 对象的例子：

当 Promise 被构造时，起始函数会被同步执行：

```
new Promise(function (resolve, reject) {
    var a = 0;
    var b = 1;
    if (b == 0)
        reject("Divide zero");
    else
```

```
        resolve(a / b);
}).then(function (value) {
    console.log("a / b = " + value);
}).catch(function (err) {
    console.log(err);
}).finally(function () {
    console.log("End");
});
```

resolve() 中可以放置一个参数用于向下一个 then 传递一个值，then 中的函数也可以返回一个值传递给 then。但是，如果 then 中返回的是一个 Promise 对象，那么下一个 then 将相当于对这个返回的 Promise 进行操作，这一点从刚才的计时器的例子中可以看出来。

reject() 参数中一般会传递一个异常给之后的 catch 函数用于处理异常。

但是请注意以下两点：

- resolve 和 reject 的作用域只有起始函数，不包括 then 以及其他序列；
- resolve 和 reject 并不能够使起始函数停止运行，别忘了 return。

Promise 函数

上述的 "计时器" 程序看上去比函数瀑布还要长，所以我们可以将它的核心部分写成一个 Promise 函数：

```
function print(delay, message) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log(message);
            resolve();
        }, delay);
    });
}

print(1000, "First").then(function () {
    return print(4000, "Second");
}).then(function () {
    print(3000, "Third");
});
```

异步函数(async function)

```
function print(delay, message) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            console.log(message);
            resolve();
        }, delay);
    });
}
async function asyncFunc() {
    await print(1000, "First");
    await print(4000, "Second");
    await print(3000, "Third");
}
asyncFunc();
```

异步函数 async function 中可以使用 await 指令，await 指令后必须跟着一个 Promise，异步函数会在这个 Promise 运行中暂停，直到其运行结束再继续运行。

异步函数实际上原理与 Promise 原生 API 的机制是一模一样的，只不过更便于程序员阅读。

带返回值的async await

```
async function asyncFunc() {
    let value = await new Promise(
        function (resolve, reject) {
            resolve("Return value");
        }
    );
    console.log(value);
}
asyncFunc();
```

闭包

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();

add();
add();
add();
```

```
// 计数器为 3
```

1. 变量 **add** 指定了函数自我调用的返回字值。
2. 自我调用函数只执行一次。设置计数器为 0。并返回函数表达式。
3. add变量可以作为一个函数使用。非常棒的部分是它可以访问函数上一层作用域的计数器。
4. 这个叫作 JavaScript 闭包。 它使得函数拥有私有变量变成可能。
5. 计数器受匿名函数的作用域保护，只能通过 add 方法修改。

js 类

示例

```
class Runoob {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age(x) {
    return x - this.year;
  }
}

let date = new Date();
let year = date.getFullYear();

let runoob = new Runoob("菜鸟教程", 2020);
document.getElementById("demo").innerHTML=
"菜鸟教程 " + runoob.age(year) + " 岁了。";
```

类方法

方法	描述
constructor()	构造函数，用于创建和初始化类

类关键字

关键字	描述
extends	继承一个类
static	在类中定义一个静态方法
super	调用父类的构造方法

继承

super() 方法引用父类的构造方法。

通过在构造方法中调用 **super()** 方法，我们调用了父类的构造方法，这样就可以访问父类的属性和方法。

继承对于代码可复用性很有用。

```
class Site {
  constructor(name) {
    this.sitename = name;
  }
  present() {
    return '我喜欢' + this.sitename;
  }
}

class Runoob extends Site {
  constructor(name, age) {
    super(name);
    this.age = age;
  }
  show() {
    return this.present() + ', 它创建了 ' + this.age + ' 年。';
  }
}

let noob = new Runoob("菜鸟教程", 5);
document.getElementById("demo").innerHTML = noob.show();
```

getter 和 setter

类中我们可以使用 getter 和 setter 来获取和设置值，getter 和 setter 都需要在严格模式下执行。getter 和 setter 可以使得我们对属性的操作变的很灵活。类中添加 getter 和 setter 使用的是 get 和 set 关键字。

以下实例为 sitename 属性创建 getter 和 setter：

```
class Runoob {
  constructor(name) {
    this.sitename = name;
  }
  get s_name() {
    return this.sitename;
  }
  set s_name(x) {
    this.sitename = x;
  }
}

let noob = new Runoob("菜鸟教程");

document.getElementById("demo").innerHTML = noob.s_name;
```

提升

函数声明和类声明之间的一个重要区别在于, 函数声明会提升，类声明不会。

你首先需要声明你的类，然后再访问它，否则代码将抛出 ReferenceError：

静态方法

静态方法是使用 static 关键字修饰的方法，又叫类方法，属于类的，但不属于对象，在实例化对象之前可以通过 **类名.方法名** 调用静态方法。

静态方法不能在对象上调用，只能在类中调用。

```
class Runoob {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

let noob = new Runoob("菜鸟教程");

// 可以在类中调用 'hello()' 方法
document.getElementById("demo").innerHTML = Runoob.hello();

// 不能通过实例化后的对象调用静态方法
```

```
// document.getElementById("demo").innerHTML = noob.hello();
// 以上代码会报错

//如果你想在对象 noob 中使用静态方法，可以作为一个参数传递给它：
class Runoob {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
let noob = new Runoob("菜鸟教程");
document.getElementById("demo").innerHTML = Runoob.hello(noob);
```