JavaScript简介

javascript简称是js，可以嵌入到html中，是基于对象和事件驱动的脚本语言。

特点

- 解释性：是一种脚本语言，是小程序段实现编程，也是解释性的编程语言。基于对象：是基于对象的语言，应用自己创建对象，许多功能都来自脚本环境中对象的方法和脚本的相互作用。
- 事件驱动：是以事件的方式和客户端的输入进行响应，不需要经过服务端。（鼠标点击，选择菜单）这种操作就是事件，就是事件，随后计算机做出的响应就是事件响应。
- 安全性：不允许访问本地磁盘，不能将数据写入服务器，不允许对网络文档进行修改删除，只能通过浏览器实现信息交互，防止信息丢失。跨平台{依赖于浏览器本身，与操作系统无关。只要浏览器支持就可以执行

javascript能做什么

能动态的修改、增加、删除html和css的代码

能动态的校验用户输入的表单数据

javascript历史

95年由Netscape公司发布，最初被命名为livescript，后来netscape公司与sun公司合作之后，将其重新命名为javascript，其实跟java语言没有什么关系，就像雷锋和雷峰塔一样，两者之间只不过有一些名称相同而已。后来微软等公司推出了类似的script语言，由于script语言众多，为了给这些语言定制一些标准，ECMA（欧洲计算机制造商协会）牵头制定了ECMA-262标准即ECMAScript ,该标准是以javascript为基础制定的，因此javascript有时也被叫做ECMAScript。 BOM是Browser Object Model的缩写，即浏览器对象模型，主要用来获取浏览器的属性和行为，比如获取浏览器的版本，获取浏览器中的历史记录等等。 DOM是Document Object Model的缩写，即文档对象模型，主要用来获取文档中标签的属性，例如获取html中某个input的value的值。

HTML/CSS/JS 的关系

- **HTML/CSS 标记语言--描述类语言**

  - HTML 决定网页结构和内容(决定看到什么)，相当于人的身体
  - CSS 决定网页呈现给用户的模样(决定好不好看)，相当于给人穿衣服、化妆
- **JS 脚本语言--编程类语言**
  - 实现业务逻辑和页面控制(决定功能)，相当于人的各种动作

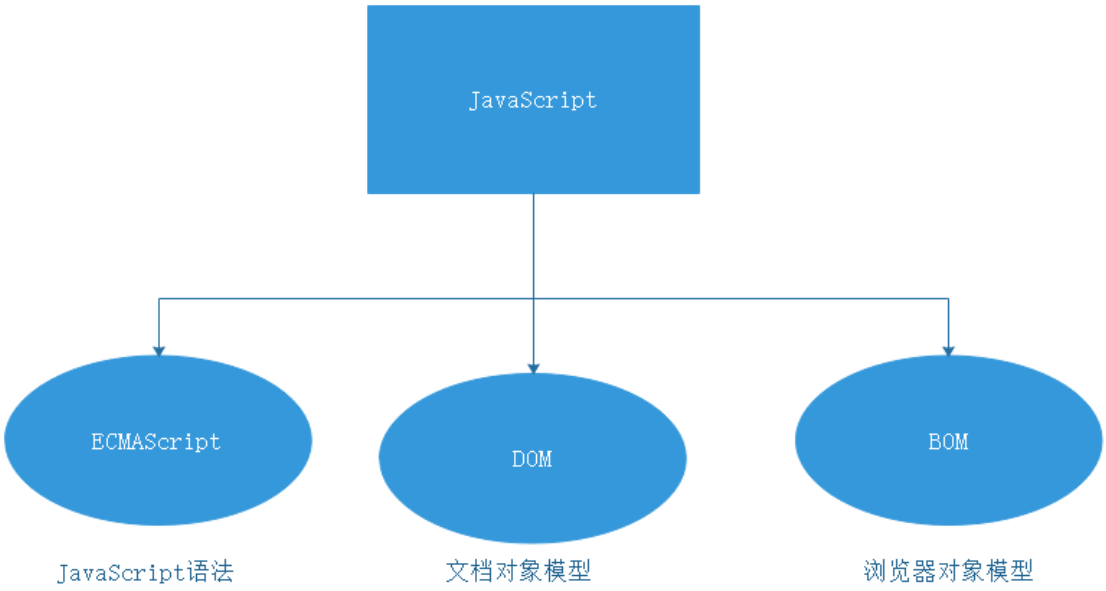
浏览器执行 JS 简介

浏览器分成两部分：渲染引擎和 JS 引擎

- 渲染引擎：用来解析HTML与CSS，俗称内核，比如 chrome 浏览器的 blink ，老版本的 webkit
- JS引擎：也称为 JS 解释器。 用来读取网页中的JavaScript代码，对其处理后运行，比如 chrome 浏览器的 V8

浏览器本身并不会执行JS代码，而是通过内置 JavaScript 引擎(解释器) 来执行 JS 代码 。JS 引擎执行代码时逐行解释每一句源码（转换为机器语言），然后由计算机去执行，所以 JavaScript 语言归为脚本语言，会逐行解释执行。

JS的组成



引入js的方式

内嵌脚本

```
<input type="button" value="button"
onclick="alert('hello world')" />
```

内部脚本

```
<script type="text/javascript">
    alert("hello world");
</script>
```

外部脚本

首先先创建一个js文件,其次在html中引入

```
<script type="text/javascript"
  src="helloworld.js"></script>
```

helloworld.js中的内容:

```
alert("hello world");
```

JavaScript的基本语法

JavaScript和Java的语法上很类似。但是也有不同。

- JavaScript区分大小写。JavaScript 语言是区分大小写的，不管是命名变量还是使用关键字的时候。alert 弹出提示框，如果将 alert 命令改为 ALERT 或者 alerT 等:
- 每行后面的分号可有可无。JavaScript并不需要一定要写;作为结束。如果没有，JavaScript会自动加上。一般都加上这个;

- JavaScript变量是弱类型：(JavaScript 都使用 var来定义你的变量)定义的时候只使用 var关键字。就可以将变量初始化为任意的值。

```
var user="javaScript";
var age=23;
```

- 使用大括号标记代码块。内容封装在大括号里面。

注释

- 单行//
- 多行注释 /*...*/

```
// 定义变量
//字符串
//定义变量
var name = '叶赫那拉';
//使用变量
alert(name);
var i = 12;
alert(i);
```

js输入输出语句

方法	说明	归属
alert(msg)	浏览器弹出警示框	浏览器
console.log(msg)	浏览器控制台打印输出信息	浏览器
prompt(info)	浏览器弹出输入框，用户可以输入	浏览器

变量的定义和使用

变量是指程序中已经命名好的一个存储单元（超市塑料袋），主要的作用为数据提供存放的信息容器。在使用前必须先明确变量的命名规则，变量的声明方法，以及作用域。变量的规则：字母，数字，下划线，,必须以字母或下划线,开始。变量名中不能有空格，加号，减号或者逗号和其他符号不能使用关键字严格区分大小写定义关键字 ：

```
1 var 变量名字=值;
```

定义变量的方式

- 第一种方式

```
1 var 变量名字=值;
```

- 第二种方式

```
1 var name;
```

案例

1.弹出一个输入框，提示用户输入姓名，弹出一个对话框，输出用户刚才输入的姓名。

变量命名规范

- 由字母(A-Za-z)、数字(0-9)、下划线(_)、美元符号($)组成，如：usrAge, num01, _name
- 严格区分大小写。var app; 和 var App; 是两个变量
- 不能 以数字开头。 18age 是错误的
- 不能 是关键字、保留字。例如：var、for、while
- 变量名必须有意义。 MMD BBD nl → age
- 遵守驼峰命名法。首字母小写，后面单词的首字母需要大写。 myFirstName

流程控制

在一个程序执行的过程中，各条代码的执行顺序对程序的结果是有直接影响的。很多时候我们要通过控制代码的执行顺序来实现我们要完成的功能。

简单理解： 流程控制就是来控制我们的代码按照什么结构顺序来执行

流程控制主要有三种结构，分别是顺序结构、分支结构和循环结构，这三种结构代表三种代码执行的顺序。

顺序结构

分支结构

循环结构

练习

1. 求1-100之间所有数的总和与平均值
2. 求1-100之间所有偶数的和

3. 求100以内7的倍数的总和
4. 使用for 循环打印矩形，要求每次只能输出一个☆

   ☆☆☆☆☆

   ☆☆☆☆☆

   ☆☆☆☆☆

   ☆☆☆☆☆
5. 使用for循环打印三角形

   ☆

   ☆☆

   ☆☆☆

   ☆☆☆☆

   ☆☆☆☆☆
6. 使用for循环打印99乘法表
7. 接收用户输入的用户名和密码，若用户名为“admin”，密码为“123456”，则提示用户登录成功! 否则，让用户一直输入。
8. 求整数1～100的累加值，但要求跳过所有个位为3的数【用continue实现】。

数组

创建数组

利用 new 创建数组

```
var 数组名 = new Array() ;
```

利用数组字面量创建数组

```
//1. 使用数组字面量方式创建空的数组
var  数组名 = [];
//2. 使用数组字面量方式创建带初始值的数组
var  数组名 = ['小白','小黑','大黄','瑞奇'];
```

获取数组元素

数组的元素是通过数组的索引来获取

```
// 定义数组
var arrays = ['星期一','星期二','星期三','星期四','星期五','星期六','星期日',];
//获取数组中的第四个元素
alert(arrays[3]);
```

遍历数组

```
1  var arrays = ['星期一','星期二','星期
   三','星期四','星期五','星期六','星期日',];
2  for (var i = 0; i < arrays.length; i++)
   {
3      console.log("fori遍历数组" +
   arrays[i]);
4  }
5
6  for (var arrayKey in arrays) {
7      console.log("forin遍历数组" +
   arrays[arrayKey]);
8  }
9
10 for (var array of arrays) {
11     console.log("forof遍历数组" + array);
12 }
13
14 var i = 0;
15 while (i < arrays.length){
16     console.log("while遍历数组" +
   arrays[i]);
17     i++;
18 }
```

数组新增一个元素

```
var arrays = ['星期一','星期二','星期三','星期四','星期五','星期六','星期日',];
//第一种方式通过修改数组的length的长度，这种方式后面的位置是没有存放元素，故是undefined
arrays.length = 9;
for (var array of arrays) {
    console.log("forof遍历数组" + array);
}
```

```
var arrays = ['星期一','星期二','星期三','星期四','星期五','星期六','星期日',];
//第二种方式直接通过下标增加。js的数组不会报数组下标越界
arrays[7] = 12;
for (var array of arrays) {
    console.log("forof遍历数组" + array);
}
```

函数

在 JS 里面，可能会定义非常多的相同代码或者功能相似的代码，这些代码可能需要大量重复使用。虽然 for 循环语句也能实现一些简单的重复操作，但是比较具有局限性，此时我们就可以使用 JS 中的函数。

函数：就是封装了一段可被重复调用执行的代码块。通过此代码块可以实现大量代码的重复使用

函数的定义

```
// 声明函数
function 函数名() {
    //函数体代码
}
```

函数的调用

```
// 调用函数
函数名();  // 通过调用函数名来执行函数体代码
```

> **注意：**声明函数本身并不会执行代码，只有调用函数时才会执行函数体代码。

带参函数

在声明函数时，可以在函数名称后面的小括号中添加一些参数，这些参数被称为**形参**，而在调用该函数时，同样也需要传递相应的参数，这些参数被称为**实参**。

函数形参和实参个数不匹配问题

参数个数	说明
实参个数等于形参个数	输出正确结果
实参个数多于形参个数	值取到形参的个数
实参个数小于形参个数	多的形参定义为undefined结果为NaN

```
function sum(num1, num2) {
    console.log(num1);
    console.log(num2);
    console.log(num1 + num2);
}
sum(100, 200);                  // 形参和实参个数相等，输出正确结果
sum(100, 400, 500, 700);    // 实参个数多于形参，只取到形参的个数
sum(200);
```

有返回值的函数

通过return来实现返回值

对象

自定义对象

对象的定义

```
//定义对象
var person = {
    //给对象定义属性
    name: '张三',
    age: 12,
    sex: '男',
    //给对象定义函数
    display:function () {
        console.log("这是对象的方法");
    }
}
```

访问属性

```
1 //访问属性存在两种方法
2 //第一种方法对象名.属性名
3 console.log(person.name);
4 //第二种方法是对象名['属性名'];
5 console.log(person['name']);
```

访问函数

```
1 //访问函数
2 //对象名.函数名();
3 person.display();
```

内置对象

查看文档：https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects

JavaScript事件

单、双击事件

单击事件：onclick=""; 双击事件：ondblclick="";

鼠标事件

鼠标在监听区域移动就会触发：onmousemove="" 鼠标离开侦听区域时触发：onmouseout="" 鼠标从外面进入侦听区域会触发：onmouseover=""

键盘事件

键盘的按下： 所有键盘按下操作都会触发事件：onkeydown="" shift键、backpace键按下时不能触发：onkeypress="" 键盘的释放：

按下键盘释放时触发事件：onkeyup=""

焦点事件

获取焦点：onfocus="" 失去焦点：onblur=""

加载事件 网页正在执行的时候，是将html文件加载到浏览器中 页面加载完成来触发事件：onload=""

事件的委派

将事件统一绑给元素共同的祖先元素，这样当后代元素上的事件触发，会冒泡到祖先，从而通过祖先元素的响应函数来处理事件，事件委派利用冒泡，减少绑定次数提高程序性能

```
<body>

<button id="btn">
    添加链接
</button>
<ul style="background: #ccc;" id="ul">
    <li><a href="javascript:;"class="link">链接1</a></li>
    <li><a href="javascript:;"class="link">链接2</a></li>
    <li><a href="javascript:;"class="link">链接3</a></li>
    <li><a href="javascript:;"class="link">链接4</a></li>
</ul>

<script>
```

```
14
15      var
    ul=document.getElementById("ul");
16      var
    btn=document.getElementById("btn");
17      btn.onclick=function(){
18          var
    li=document.createElement("li");
19          li.innerHTML="<a
    href='javascript:;' class=\"link\">新链接
    </a>";
20          ul.appendChild(li);
21      }
22
23      // 绑定单击响应函数
24      // 点击a链接冒泡到li 再冒泡到ul再响应ul单
    机事件
25      ul.onclick=function (event) {
26          event=event || window.event;
27          if(event.target.className ===
    "link" ){
28              alert("hi,我是ul单击响应函
    数");
29          }
30      };
31
32  </script>
```

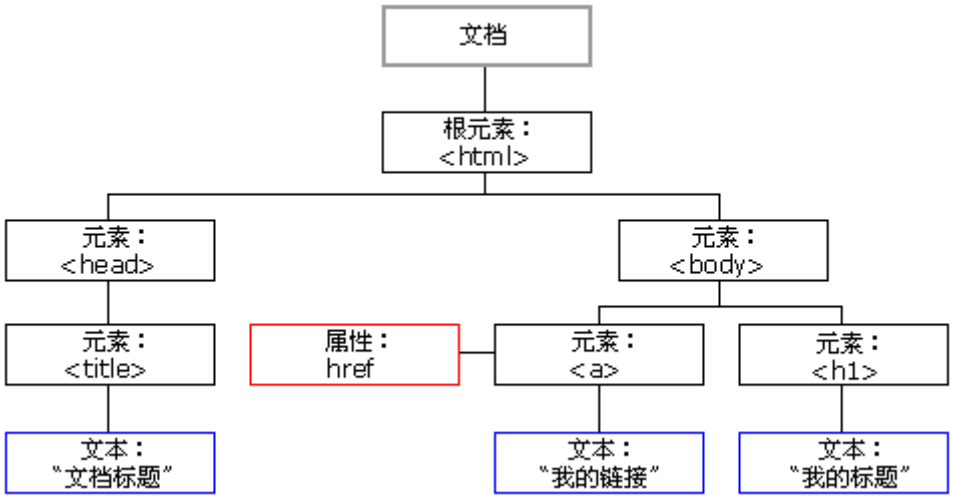
```
33
34 </body>
```

Document 对象

通过 HTML DOM，JavaScript 能够访问和改变 HTML 文档的所有元素。

当网页被加载时，浏览器会创建页面的文档对象模型（*D*ocument *O*bject *M*odel）。

HTML DOM 模型被结构化为*对象树*：



获取节点

获取所有节点

查询是否有子节点： `hasChildNodes()`

获取第一个子节点： `firstChild`

查询所有子节点： `childNodes`

节点名称： `nodeName`

- 元素 -- 元素标签名
- 文本节点 -- #text
- 注释 -- #comment

节点类型： `nodeType`

> 节点类型：
>
> - **1 ELEMENT_NODE 元素节点**
> - **2 ATTRIBUTE_NODE 属性节点**
> - **3 TEXT_NODE 文本节点**
> - 4 CDATA_SECTION_NODE CDATA区段
> - 5 ENTITY_REFERENCE_NODE 实体引用元素
> - 6 ENTITY_NODE 实体
> - 7 PROCESSING_INSTRUCTION_NODE 表示处理指令
> - **8 COMMENT_NODE 注释节点**

> - 9 DOCUMENT_NODE 最外层的Root element，包括所有其他子节点
> - **10 DOCUMENT_TYPE_NODE <!DOCTYPE>**
> - 11 DOCUMENT_FRAGMENT_NODE 文档碎片节点
> - 12 NOTATION_NODE DTD中声明的符号节点

节点值： `nodeValue`

- 元素 -- null
- 文本节点 -- 文本本身内容
- 注释 -- 注释的内容

获取兄弟节点

- **上一个兄弟节点：** `previousSibling`
- **下一个兄弟节点：** `nextSibling`

获取指定节点

- **通过ID获取：** `document.getElementById('id名')`
- **通过class获取：** `document.getElementsByClassName('class名')`
- **通过标签获取：** `document.getElementsByTagName('标签名')`
- **通过name获取：** `document.getElementsByName('name值')`

属性节点

获取所有属性节点： `getAttributeNames()`

获取指定属性节点的属性值： `getAttribute('属性名')`

设置属性节点的属性名： `setAttribute('属性名', '属性值')`

节点操作

添加节点

创建元素节点： `createElement('标签名')`

创建文本节点： `createTextNode('文本内容')`

添加节点： `父节点.appendChild(子节点)`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>添加节点</title>
</head>
<body>
<div id="myDiv" class="myClass">
    <h1>添加节点</h1>
```

```
</div>
<button onclick="add()">添加</button>
<script>
  function add() {
      //获取到父节点
      var myDiv = document.getElementById("myDiv");
      //创建子节点
     var h1 = document.createElement("h1");
      //创建文本节点
      var text = document.createTextNode("这个是新添加的h1节点")
      //把文件节点添加到子结点中
      h1.appendChild(text);
      //再把子节点添加到父节点中
      myDiv.appendChild(h1);
  }
</script>
</body>
</html>
```

移动节点

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>移动节点</title>
</head>
<body>
<div id="myDiv" class="myClass">
    <h1>移动加节点</h1>
</div>
<h1 id="h">这个是一个h1标签</h1>
<button onclick="move()">移动</button>
<script>
    function move() {
        //获取到父节点
        var myDiv = document.getElementById("myDiv");
        //获取需要移动的节点
        var h = document.getElementById("h");
        myDiv.appendChild(h);
    }
</script>
</body>
</html>
```

删除节点

删除元素节点： `父节点.removeChild(子节点)`

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>删除节点</title>
</head>
<body>
<div id="myDiv" class="myClass">
    <h1>删除节点</h1>
    <p>这个是一个p标签</p>
    <p>这个是一个p标签</p>
    <p>这个是一个p标签</p>
    <p>这个是一个p标签</p>
    <p>这个是一个p标签</p>
    <p>这个是一个p标签</p>
</div>

<button onclick="remove()">删除</button>
<script>
    function remove() {
        //获取父节点
         var div = document.getElementById("myDiv");
         var ps = document.getElementsByTagName("p");
         //这样删除不完全
         // div.removeChild(ps[0]);
```

```
26          // div.removeChild(ps[1]);
27          // div.removeChild(ps[2]);
28          // div.removeChild(ps[3]);
29          // div.removeChild(ps[4]);
30          // div.removeChild(ps[5]);
31          //以下方式虽然可以完全删除，但是这样处理含繁琐
32          // div.removeChild(p[0]);
33          // div.removeChild(p[0]);
34          // div.removeChild(ps[0]);
35          // div.removeChild(ps[0]);
36          // div.removeChild(ps[0]);
37          // div.removeChild(ps[0])
38
39          //for循环 i++方式 也是删除不完全
40          // for (var i = 0; i < p.length; i++) {
41          //     div.removeChild(ps[i]);
42          // }
43          //for循环 i--方式 完全删除
44          for (var i = p.length - 1; i >= 0 ; i--) {
45              div.removeChild(ps[i]);
46          }
47      }
48  </script>
49  </body>
```

```
50 </html>
```

替换节点

替换成新节点

- 获取父节点
- 创建新节点
- 获取被替换的节点
- 父节点.replaceChild('新节点','被替换的节点')

替换成存在的节点

- 获取父节点
- 获取替换节点
- 获取被替换的节点
- 父节点.replaceChild('替换的节点','被替换的节点')

插入节点

插入节点： `A.insertBefore(B,C)` 在A节点中把B节点插入到C节点前面

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>插入节点</title>
6 </head>
```

```
<body>
<div id="myDiv" class="myClass">
    <h1 id="after">你今天学习了吗?</h1>
</div>

<button onclick="insert()">插入</button>
<script>
    function insert() {
        //获取父节点
        var div = document.getElementById("myDiv");
        //创建节点
        var newH = document.createElement("h1");
        var text = document.createTextNode("这个是我要插入的内容");
        newH.appendChild(text);
        var after = document.getElementById("after");
        div.insertBefore(newH,after);
    }
</script>
</body>
</html>
```

复制节点

复制节点： `复制的节点.cloneNode(boolean)` boolean的值为true：复制自己包括其所有的子节点；false：只复制自己，不复制其子节点

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>复制节点</title>
</head>
<body>
<div id="myDiv" class="myClass">
    <h1 id="after">你今天学习了吗?</h1>
    <h1 id="after1">你今天学习了吗?</h1>
    <h1 id="after2">你今天学习了吗?</h1>
    <h1 id="after3">你今天学习了吗?</h1>
</div>
<div id="two"></div>

<button onclick="copy()">复制</button>
<script>
    function copy() {
        //获取父节点
         var div =
document.getElementById("myDiv");
```

```
        var clone = div.cloneNode(true);
        var two = document.getElementById("two");
        two.appendChild(clone);

    }
</script>
</body>
</html>
```

文本节点操作

添加： 文本节点.appendData(文本内容)

指定位置添加： 文本节点.insertData(位置下标,文本内容)

删除： `文本节点.deleteData(下标,删除多少个)

替换： `文本节点.replaceData(下标,替换的字符数目,文本内容)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>文本节点操作</title>
</head>
```

```
<body>
<h1 id="font">这是一个标题标签</h1>
<button onclick="add()">添加</button>
<button onclick="insert()">插入</button>
<button onclick="del()">删除</button>
<button onclick="replace()">替换
</button>
<script>
    function add() {
        //获取到需要操作的文本节点，通过父节点
获取文本节点
        var h1 =
document.getElementById('font');
        var text = h1.firstChild;
        text.appendData("这个是新添加内
容");
    }

    function insert() {
        //获取到需要操作的文本节点，通过父节点
获取文本节点
        var h1 =
document.getElementById('font');
        var text = h1.firstChild;
        text.insertData(1, '《在第二个字符
位置添加内容》');
    }
```

```
27
28      function del() {
29          //获取到需要操作的文本节点，通过父节点获取文本节点
30          var h1 = document.getElementById('font');
31          var text = h1.firstChild;
32          text.deleteData(1, 3);
33      }
34      function replace() {
35          //获取到需要操作的文本节点，通过父节点获取文本节点
36          var h1 = document.getElementById('font');
37          var text = h1.firstChild;
38          text.replaceData(1,3,'这个是替换的字符');
39      }
40
41  </script>
42  </body>
43  </html>
```

节点操作CSS样式

设置CSS样式： `节点.style.ssss属性='新值'`

获取CSS样式： `节点.style.csss属性` 获取的是内联样式

获取最终样式：

`document.defaultView.getComputedStyle(参数1，参数2)`

参数1：想要回去样式的元素，参数2：可选 伪类，不需要获取伪类的样式，直接传null

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>CSS操作</title>
    <style>
        div{
            height: 100px;
            background-color: #8d3019;
        }
    </style>
</head>
<body>
<div id="myDiv"></div>
<button onclick="change()">修改CSS样式</button>
<button onclick="get()">获取CSS样式</button>
<button onclick="getAll()">获取最终样式</button>
```

```
<script>
    var myDiv = document.getElementById('myDiv');
    function change() {
        myDiv.style.backgroundColor= "red";
        myDiv.style.width= "100px";
    }
    function get() {

        if (myDiv.style.backgroundColor == ''){
            myDiv.style.backgroundColor= "#8d3019";
        }

 console.log(myDiv.style.backgroundColor);
    }

    function getAll() {

 console.log(document.defaultView.getComputedStyle(myDiv,null).backgroundColor);
    }
</script>
</body>
```

```
37 </html>
```

innerHTML/innerText/outterHTML

```
 1 <!DOCTYPE html>
 2 <html lang="en">
 3 <head>
 4     <meta charset="UTF-8">
 5     <title>Title</title>
 6 </head>
 7 <body>
 8 <div id="myDiv">
 9     <ul>
10         <li>aaaa</li>
11         <li>bbbb</li>
12         <li>cccc</li>
13         <li>eeee</li>
14     </ul>
15 </div>
16 <button onclick="innerHtml()">获取/设置
   innerHTML</button>
17 <button onclick="innerTEXT()">获取/设置
   innerText</button>
```

```
18 <button onclick="outerHtml()">获取/设置
   outerHTML</button>
19 <script>
20     var div =
   document.getElementById("myDiv");
21     function innerHtml() {
22         //获取元素内容
23         console.log(div.innerHTML);
24         //删除元素
25         //div.innerHTML="";
26         //替换
27         //div.innerHTML = '<h1>替换后的元
   素</h1>';
28         //添加元素
29         div.innerHTML += '<h1>替换后的元素
   </h1>';
30     }
31
32     function innerTEXT() {
33         //只是获取元素里面的文本内容
34         console.log(div.innerText);
35         //删除元素
36         //div.innerText="";
37         //替换
38         //div.innerText = '<h1>替换后的元
   素</h1>';
39         //添加元素
```

```
40          div.innerText += '<h1>替换后的元素</h1>';
41      }
42
43      function outerHtml() {
44          //只是获取元素里面的文本内容
45          console.log(div.outerHTML);
46          //删除元素
47          //div.outerHTML="";
48          //替换
49          //div.outerHTML = '<h1>替换后的元素</h1>';
50          //添加元素
51         // div.outerHTML += '<h1>替换后的元素</h1>';
52      }
53  </script>
54  </body>
55  </html>
```

window对象