JS+ES6

**********JS**********
**********变量**********
定义变量

```
var x=5;
var y=6;
var z=x+y;
var lastname="Doe", age=30, job="carpenter"; //可以一行代码多个变量定义

var lastname="Doe",
age=30,
job="carpenter";    //跨多行定义
```

与代数一样，JavaScript 变量可用于存放值(比如 x = 5)和表达式（比如 z=x+y）。
…变量必须以字母开头
…变量也能以 $ 和 _ 符号开头（不过我们不推荐这么做）
…变量名称对大小写敏感（y 和 Y 是不同的变量）

无值的变量，未使用值来声明的变量，其值实际上是 undefined

JavaScript变量均为对象。当您声明一个变量时，就创建了一个新的对象

**********数据类型及其常用方法**********
数据类型
…基本类型： Number String Boolean **null** undefined Symbol
…引用数据类型： Object Array Function
JavaScript 拥有动态类型，相同的变量可用作不同的类型

对象属性：键值对在 JavaScript 对象通常称为对象属性
对象方法：对象的方法定义了一个函数，并作为对象的属性存储

String 创建：
1. JavaScript 字符串是原始值，可以使用字符创建
   `var firstName = "John"`
2. 也可以使用 **new** 关键字将字符串定义为一个对象
   `var firstName = new String("John")` //不要创建 String 对象，拖慢执行速度，并可能产生其他副作用

String 常用方法：

.length()	返回长度
.trim()	移除空白
.trimleft()	移除左边空白
.trimright()	移除右边空白
.charAt(n)	返回第n个字符
.concat(value, ...)	拼接
.indexOf(substring, start)	子序列位置
.substring(from, to)	根据索引取子序列
.slice(start, end)	
.toLowerCase()	小写
.toUpperCase()	大写
.split(delimiter, limit)	分割

Array 定义数组的三种方式，等价：

```
1. var cars=new Array();
   cars[0]="Saab";
   cars[1]="Volvo";
2. var cars=new Array("Saab","Volvo");
3. var cars=["Saab","Volvo"];
```

Array 常用方法:

.length	数组的大小
.push(ele)	尾部追加元素
.pop()	获取尾部的元素
.unshift(ele)	头部插入元素
.shift()	头部移除元素
.slice(start, end)	切片
.reverse()	反转
.join(seq)	将数组元素连接成字符串
.concat(val, ...)	连接数组
.sort()	排序
.forEach()	将数组的每个元素传递给回调函数
.splice()	删除元素，并向数组添加新元素。
.map()	返回一个数组元素调用函数处理后的值的新数组

.forEach(function(currentValue, index, arr), thisValue)

currentValue	必需	当前元素
index	可选	当前元素的索引值
arr	可选	当前元素所属的数组对象
thisValue	可选	传递给函数的值一般用 "this" 值 如果这个参数为空， "undefined" 会传递给 "this" 值

.splice(index,howmany,item1,.....,itemX)

index	必需	规定从何处添加/删除元素。 该参数是开始插入或删除的数组元素的下标，必须是数字。
howmany	必需	规定应该删除多少元素。必须是数字，但可以是 "0"。 未规定此参数，则删除从 index 开始到原数组结尾的所有元素。
item1, ..., itemX	可选	要添加到数组的新元素

.map(function(currentValue,index,arr), thisValue)

function(currentValue, index,arr)	必须	函数，数组中的每个元素都会执行这个函数
currentValue	必须	当前元素的值
index	可选	当期元素的索引值
arr	可选	当期元素属于的数组对象
thisValue	可选	对象作为该执行回调时使用，传递给函数，用作 "this" 的值。 如果省略了 thisValue ， "this" 的值为 "undefined"

array.filter(function(currentValue,index,arr), thisValue)

function(currentValue, index,arr)	必须	函数，数组中的每个元素都会执行这个函数
currentValue	必须	当前元素的值
index	可选	当期元素的索引值
arr	可选	当期元素属于的数组对象
thisValue	可选	对象作为该执行回调时使用，传递给函数，用作 "this" 的值 如果省略了 thisValue ， "this" 的值为 "undefined"

function 函数定义:

```
function functionname() //定义了一个函数名为functionname的函数
{
  // 执行代码
}
// 匿名函数方式
var sum = function(a, b){
  return a + b;
}
```

局部 JavaScript 变量
在 JavaScript 函数内部声明的变量（使用 var）是局部变量
您可以在不同的函数中使用名称相同的局部变量
只要函数运行完毕，函数内部局部变量就会被删除

函数中的arguments参数

```
function add(a,b){
 console.log(a+b);
 //arguments相当于将出传入的参数全部包含，这里取得就是第一个元素1
 console.log(arguments.length);   console.log(arguments[0]);
}
add(1,2)    //3，2，1
```

函数只能返回一个值，如果要返回多个值，只能将其放在数组或对象中返回

JavaScript代码运行前有一个类似编译的过程即词法分析，词法分析主要有三个步骤：
…分析参数
…函数接收形式参数，添加到AO的属性，并且这个时候值为undefine,即AO.age=undefine
接收实参，添加到AO的属性，覆盖之前的undefine
…再分析变量的声明
…如var age;或var age=18;
如果上一步分析参数中AO还没有age属性，则添加AO属性为undefine，即AO.age=undefine
如果AO上面已经有age属性了，则不作任何修改
…分析函数声明
…如果有function age(){}把函数赋给AO.age ,覆盖上一步分析的值

```
function func(age) {
  console.log(age);
  var age = 25;
  console.log(age);
 function age() {
 }
 console.log(age);
 }
 func(18);  //function age(){}    25   25
```

function(){}.bind()方法主要就是将函数绑定到某个对象
bind()会创建一个函数，函数体内的this对象的值会被绑定到传入bind()第一个参数的值
例如，f.bind(obj)，实际上可以理解为obj.f()，这时，f函数体内的this自然指向的是obj

```
var a = {
  b : function(){
    var that = this;
    var func = function(){
      console.log(that.c);
    }
    func();
  },
  c : 'Hello!'
}
a.b();
//Hello!
```

注意：可以通过赋值的方式将this赋值给that

```
var a = {
  b : function(){
    var func = function(){
      console.log(this.c);
    }.bind(this);
    func();
```

```
    },
    c : 'Hello!'
}
a.b();
//Hello!

function f(y, z){
    return this.x + y + z;
}
var m = f.bind({x : 1}, 2);
console.log(m(3));
//6      bind方法会把它的第一个实参绑定给f函数体内的this
//所以这里的this即指向{x : 1}对象，从第二个参数起，会依次传递给原始函数
//这里的第二个参数2，即是f函数的y参数，最后调用m(3)的时候
//这里的3便是最后一个参数z了，所以执行结果为1 + 2 + 3 = 6
```

Object 自定义方式：对象由花括号分隔。在括号内部

对象的属性以名称和值对的形式 (name : value) 来定义

JavaScript 对象是变量的容器

```
var person = {firstname: "John", lastname: "Doe", id: 5566};    //可跨行
```

对象的两种寻址的方式：

1. name = person.lastname;
2. name = person["lastname"];

遍历对象中的内容

```
var a = {"name": "Alex", "age": 18};
for (var i in a){
  console.log(i, a[i]);
}
```

已有的对象 Object ：

数据类型：Number String Boolean

组合对象：Array Math Date

高级对象：Object Error Function RegExp Global

```
var person=new Object();          // 创建一个person对象
person.name="Alex";               // 添加person对象的name属性
person.age=18;                  // 添加person对象的age属性
```

创建 Date 对象

```
//方法1：不指定参数
var d1 = new Date();
console.log(d1.toLocaleString());                 // 2020/10/8 下午6:33:12
//方法2：参数为日期字符串
var d2 = new Date("2004/3/20 11:12");     //2004/3/20 上午11:12:00
console.log(d2.toLocaleString());
var d3 = new Date("04/03/20 11:12");          //2004/3/20 上午11:12:00
console.log(d3.toLocaleString());
//方法3：参数为毫秒数
var d3 = new Date(5000);
console.log(d3.toLocaleString());                 //1970/1/1 上午8:00:05
console.log(d3.toUTCString());                 //Thu, 01 Jan 1970 00:00:05 GMT
//方法4：参数为年月日小时分钟秒毫秒
var d4 = new Date(2004,2,20,11,12,0,300);
console.log(d4.toLocaleString());  //毫秒并不直接显示
```

Date 对象常用方法：

.getDate()　　获取日
.getDay ()　　获取星期
.getMonth ()　　获取月（0-11）
.getFullYear ()　　获取完整年份
.getYear ()　　获取年
.getHours ()　　获取小时
.getMinutes ()　　获取分钟
.getSeconds ()　　获取秒
.getMilliseconds ()　　获取毫秒
.getTime ()　　返回累计毫秒数(从1970/1/1午夜)

JSON对象

```
var str1 = '{"name": "Alex", "age": 18}';
var obj1 = {"name": "Alex", "age": 18};
// JSON字符串转换成对象  parse
var obj = JSON.parse(str1);
// 对象转换成JSON字符串  stringify
var str = JSON.stringify(obj1);
```

RegExp 对象

语法：/正则表达式主体/修饰符(可选)

修饰符：　i　执行对大小写不敏感的匹配。
　　g　执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
　　m　执行多行匹配。

定义正则表达式两种方式

```
1. var reg1 = new RegExp("^[a-zA-Z][a-zA-Z0-9]{5,11}");
2. var reg2 = /^[a-zA-Z][a-zA-Z0-9]{5,9}$/;

// 正则校验数据  test返回值为布尔值
reg1.test('jason666')
reg2.test('jason666')

/*第一个注意事项，正则表达式中不能有空格*/

// 全局匹配
var s1 = 'egondsb dsb dsb';
s1.match(/s/)
s1.match(/s/g)
var reg2 = /^[a-zA-Z][a-zA-Z0-9]{5,9}$/g
reg2.test('egondsb');
reg2.test('egondsb');
reg2.lastIndex;
/*第二个注意事项，全局匹配时有一个lastIndex属性*/
```

.exec() 方法

用于检索字符串中的正则表达式的匹配。
　该函数返回一个数组，其中存放匹配的结果，如果未找到匹配，则返回值为 null

```
/e/.exec("The best things in life are free!");    //e

// 校验时不传参数
var reg2 = /^[a-zA-Z][a-zA-Z0-9]{5,9}$/
reg2.test();
```

```
    reg2.test(undefined);

    var reg3 = /undefined/;
    reg3.test();
```

Math 对象常用方法：

abs(x)	返回数的绝对值。
exp(x)	返回 e 的指数。
floor(x)	对数进行下舍入。
log(x)	返回数的自然对数（底为e）。
max(x,y)	返回 x 和 y 中的最高值。
min(x,y)	返回 x 和 y 中的最低值。
pow(x,y)	返回 x 的 y 次幂。
random()	返回 0 ~ 1 之间的随机数。
round(x)	把数四舍五入为最接近的整数。
sin(x)	返回数的正弦。
sqrt(x)	返回数的平方根。
tan(x)	返回角的正切。

**********typeof**********

typeof是一个一元运算符（就像++，--，！，- 等一元运算符），不是一个函数，也不是一个语句。

- undefined - 如果变量是 undefined 类型的
- **boolean** - 如果变量是 **boolean** 类型的
- number - 如果变量是 Number 类型的
- string - 如果变量是 String 类型的
- object - 如果变量是一种引用类型或 **null** 类型的

**********运算符**********

和C语言类似除了：

```
1 == “1”          // true  弱等于
1 === "1"         // false 强等于
//上面这张情况出现的原因在于JS是一门弱类型语言(会自动转换数据类型)
//所以当你用两个等号进行比较时，JS内部会自动先将数值类型的1转换成字符串类型的1再进行比较
//以后写JS涉及到比较时尽量用三等号来强制限制类型防止判断错误
```

**********异常捕获**********

```
try {
    ...   //异常的抛出
} catch(e) {
    ...   //异常的捕获与处理
} finally {
    ...   //结束处理
}
```

异常主动抛出

```
throw exception   //exception   可以是 JavaScript 字符串、数字、逻辑值或对象

function myFunction() {
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "")  throw "值为空";
        if(isNaN(x)) throw "不是数字";
```

```
        x = Number(x);
        if(x < 5)   throw "太小";
        if(x > 10)  throw "太大";
    }
    catch(err) {
        message.innerHTML = "错误: " + err;
    }
}
```

**********常用HTML事件**********

onchange	HTML 元素改变
onclick	用户点击 HTML 元素
onmouseover	用户在一个HTML元素上移动鼠标
onmouseout	用户从一个HTML元素上移开鼠标
onkeydown	用户按下键盘按键
onload	浏览器已完成页面的加载

**********BOM**********

JavaScript分为 ECMAScript，DOM，BOM

BOM （Browser Object Model）是指浏览器对象模型，它使 JavaScript 有能力与浏览器进行“对话”

DOM （Document Object Model）是指文档对象模型，通过它，可以访问HTML文档的所有元素

Window对象是客户端JavaScript最高层对象之一，由于window对象是其它大部分对象的共同祖先

…在调用window对象的方法和属性时，可以省略window对象的引用

例如：window.**document**.write()可以简写成：document.write()

…所有浏览器都支持 **window** 对象，它表示浏览器窗口

…如果文档包含框架（frame 或 iframe 标签），浏览器会为 HTML 文档创建一个 **window** 对象

并为每个框架创建一个额外的 **window** 对象

…没有应用于 **window** 对象的公开标准，不过所有浏览器都支持该对象

…所有 JavaScript 全局对象、函数以及变量均自动成为 **window** 对象的成员

…全局变量是 **window** 对象的属性，全局函数是 **window** 对象的方法

Window 常用方法：

window.innerHeight	浏览器窗口的内部高度
window.innerWidth	浏览器窗口的内部宽度
window.open()	打开新窗口
window.close()	关闭当前窗口

window的子对象

navigator对象（了解即可）

浏览器对象，通过这个对象可以判定用户所使用的浏览器，包含了浏览器相关信息

以下都是window.**navigate**.appName简写

```
navigator.appName       // Web浏览器全称
navigator.appVersion    // Web浏览器厂商和版本的详细字符串
navigator.userAgent     // 客户端绝大部分信息
navigator.platform      // 浏览器运行所在的操作系统
```

screen对象（了解即可）

screen.availWidth	可用的屏幕宽度
screen.availHeight	可用的屏幕高度

history对象（了解即可）

…浏览历史对象，包含了用户对当前页面的浏览历史，但我们无法查看具体的地址

可以简单的用来前进或后退一个页面

window.history 对象包含浏览器的历史。

```
history.forward()       // 前进一页
history.back()          // 后退一页
```

location对象

```
window.location       对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面
location.href             //获取URL
location.href="URL"       // 跳转到指定页面
location.reload()         //重新加载页面
```

弹出框

JavaScript 中创建三种消息框：警告框、确认框、提示框

警告框：警告框经常用于确保用户可以得到某些信息

当警告框出现后，用户需要点击确定按钮才能继续进行操作

```
alert("你看到了吗？");
```

确认框（了解即可）：确认框用于使用户可以验证或者接受某些信息

当确认框出现后，用户需要点击确定或者取消按钮才能继续进行操作

如果用户点击确认，那么返回值为 **true** ，如果用户点击取消，那么返回值为 **false**

```
confirm("你确定吗？")
```

提示框（了解即可）：提示框经常用于提示用户在进入页面前输入某个值

当提示框出现后，用户需要输入某个值，然后点击确认或取消按钮才能继续操纵

如果用户点击确认，那么返回值为 输入的值，如果用户点击取消，那么返回值为 **null**

```
prompt("请在下方输入","你的答案")
```

计时相关

通过使用 JavaScript，在一定时间间隔之后来执行代码，称之为计时事件

```
setTimeout()
  var t=setTimeout("JS语句",毫秒)
```

setTimeout() 方法会返回某个值。在上面的语句中，值被储存在名为 t 的变量中

假如你希望取消这个 **setTimeout()**，你可以使用这个变量名来指定它

…第一个参数是含有 JavaScript 语句的字符串

这个语句可能诸如 "alert('5 seconds!')"，或者对函数的调用，例如 alertMsg()"

…第二个参数指示从当前起多少毫秒后执行第一个参数

```
clearTimeout()
clearTimeout(setTimeout_variable)
  // 在指定时间之后执行一次相应函数
  var timer = setTimeout(function(){alert(123);}, 3000)
  // 取消setTimeout设置
  clearTimeout(timer);
```

```
setInterval()
setInterval("JS语句",时间间隔)
```

setInterval() 方法可按照指定的周期（以毫秒计）来调用函数或计算表达式

setInterval() 方法会不停地调用函数，直到 **clearInterval()** 被调用或窗口被关闭

…由 **setInterval()** 返回的 ID 值可用作 **clearInterval()** 方法的参数。

```
clearInterval()
clearInterval(setinterval返回的ID值)
```

clearInterval() 方法可取消由 **setInterval()** 设置的 timeout。

clearInterval() 方法的参数必须是由 **setInterval()** 返回的 ID 值。

```
  // 每隔一段时间就执行一次相应函数
  var timer = setInterval(function(){console.log(123);}, 3000)
  // 取消setInterval设置
  clearInterval(timer);
```

**********DOM**********

当网页被加载时，浏览器会创建页面的文档对象模型（Document Object Model）
HTML DOM 模型被构造为对象的树
DOM标准规定HTML文档中的每个成分都是一个节点(node)：
文档节点(document 对象)：代表整个文档
元素节点(element 对象)：代表一个元素（标签）
文本节点(text 对象)：代表元素（标签）中的文本
属性节点(attribute 对象)：代表一个属性，元素（标签）才有属性
注释是注释节点(comment 对象)

JavaScript 可以通过DOM创建动态的 HTML：
JavaScript 能够改变页面中的所有 HTML 元素
JavaScript 能够改变页面中的所有 HTML 属性
JavaScript 能够改变页面中的所有 CSS 样式
JavaScript 能够对页面中的所有事件做出反应

查找标签
直接查找
document.getElementById()　　根据ID获取一个标签
document.getElementsByClassName()　　根据class属性获取
document.getElementsByTagName()　　根据标签名获取标签合集
涉及到DOM操作的JS代码应该放在文档的哪个位置。

间接查找
parentElement　　父节点标签元素
children　　所有子标签
firstElementChild　　第一个子标签元素
lastElementChild　　最后一个子标签元素
nextElementSibling　　下一个兄弟标签元素
previousElementSibling　　上一个兄弟标签元素

节点操作
创建节点
createElement(标签名)

```
var divEle = document.createElement("div");
```

添加节点
追加一个子节点（作为最后的子节点）

```
somenode.appendChild(newnode);
```

把增加的节点放到某个节点的前边。

```
somenode.insertBefore(newnode,某个节点);

var imgEle=document.createElement("img");
imgEle.setAttribute("src", "http://image11.m1905.cn/uploadfile/s2010/0205/20100205083613178.jpg");
var d1Ele = document.getElementById("d1");
d1Ele.appendChild(imgEle);
```

删除节点：
获得要删除的元素，通过父元素调用该方法删除

```
somenode.removeChild(要删除的节点)
```

替换节点：

```
somenode.replaceChild(newnode, 某个节点);
```

属性节点
获取文本节点的值：

```
        var divEle = document.getElementById("d1")
        divEle.innerText
        divEle.innerHTML
    设置文本节点的值：
        var divEle = document.getElementById("d1")
        divEle.innerText="1"
        divEle.innerHTML="<p>2</p>"
    attribute操作
        var divEle = document.getElementById("d1");
        divEle.setAttribute("age","18")
        divEle.getAttribute("age")
        divEle.removeAttribute("age")
        // 自带的属性还可以直接.属性名来获取和设置
        imgEle.src
        imgEle.src="..."

  获取值操作
    elementNode.value
      适用于以下标签：
        .input
        .select
        .textarea
          var iEle = document.getElementById("i1");
          console.log(iEle.value);
          var sEle = document.getElementById("s1");
          console.log(sEle.value);
          var tEle = document.getElementById("t1");
          console.log(tEle.value);


class的操作
  className                获取所有样式类名(字符串)
  classList.remove(cls)      删除指定类
  classList.add(cls)         添加类
  classList.contains(cls)    存在返回true，否则返回false
  classList.toggle(cls)      存在就删除，否则添加

指定CSS操作
  obj.style.backgroundColor="red"
  JS操作CSS属性的规律：
    1.对于没有中横线的CSS属性一般直接使用style.属性名即可
      obj.style.margin
      obj.style.width
      obj.style.left
      obj.style.position
    2.对含有中横线的CSS属性，将中横线后面的第一个字母换成大写即可
      obj.style.marginTop
      obj.style.borderLeftWidth
      obj.style.zIndex
      obj.style.fontFamily


事件
  onclick              当用户点击某个对象时调用的事件句柄
  ondblclick       当用户双击某个对象时调用的事件句柄

  onfocus          元素获得焦点
  onblur
```

元素失去焦点//用于表单验证,用户离开某个输入框时,代表已经输入完了,我们可以对它进行验证.
onchange 域的内容被改变//通常用于表单元素,当元素内容被改变时触发.（select联动）

onkeydown 某个键盘按键被按下//当用户在最后一个输入框按下回车按键时,表单提交.
onkeypress 某个键盘按键被按下并松开
onkeyup 某个键盘按键被松开
onload 一张页面或一幅图像完成加载
onmousedown 鼠标按钮被按下
onmousemove 鼠标被移动
onmouseout 鼠标从某元素移开
onmouseover 鼠标移到某元素之上

onselect 在文本框中的文本被选中时发生
onsubmit 确认按钮被点击，使用的对象是form

绑定方式：
1.方式一

```
<div id="d1" onclick="changeColor(this);">点我</div>
<script>
  function changeColor(ths) {
    ths.style.backgroundColor="green";
  }
</script>
```

2，方式二

```
<div id="d2">点我</div>
<script>
  var divEle2 = document.getElementById("d2");
  divEle2.onclick=function () {
    this.innerText="呵呵";
  }
</script>
```

1.定时器实例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
<body>
<input type="text" id="i1">
<button id="b1">开始</button>
<button id="b2">结束</button>

<script>
    var t;
    function showTime() {
        var i1Ele = document.getElementById('i1');
        var time = new Date();
        i1Ele.value = time.toLocaleString();
    }
    showTime();
    var b1Ele = document.getElementById('b1');
    b1Ele.onclick = function (ev) {
        if (!t){
            t = setInterval(showTime,1000)
        }
```

```
    };
    var b2Ele = document.getElementById('b2');
    b2Ele.onclick = function (ev) {
      clearInterval(t);
      t = undefined
    };

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

2.搜索框实例

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>搜索框示例</title>

</head>
<body>
    <input id="d1" type="text" value="请输入关键字" onblur="blur()" onfocus="focus()">

<script>
function focus(){
    var inputEle=document.getElementById("d1");
    if (inputEle.value==="请输入关键字"){
        inputEle.value="";
    }
}

function blur(){
    var inputEle=document.getElementById("d1");
    var val=inputEle.value;
    if(!val.trim()){
        inputEle.value="请输入关键字";
    }
}
</script>
</body>
</html>
```

3.select联动

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="x-ua-compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>select联动</title>
</head>
<body>
<select id="province">
  <option>请选择省:</option>
</select>

<select id="city">
  <option>请选择市:</option>
</select>
```

```
<script>
  data = {"河北省": ["廊坊", "邯郸"], "北京": ["朝阳区", "海淀区"], "山东": ["威海市", "烟台市"]};

  var p = document.getElementById("province");
  var c = document.getElementById("city");

  for (var i in data) {
    var optionP = document.createElement("option");
    optionP.innerHTML = i;
    p.appendChild(optionP);
  }
  p.onchange = function () {
    var pro = (this.options[this.selectedIndex]).innerHTML;
    var citys = data[pro];
    // 清空option
    c.innerHTML = "";

    for (var i=0;i<citys.length;i++) {
      var option_city = document.createElement("option");
      option_city.innerHTML = citys[i];
      c.appendChild(option_city);
    }
  }
</script>
</body>
</html>
```

window.onload

当我们给页面上的元素绑定事件的时候，必须等到文档加载完毕

…因为我们无法给一个不存在的元素绑定事件

window.onload事件在文件加载过程结束的时候触发

…此时，文档中的所有对象都位于DOM中，并且所有图像，脚本，链接和子框架都已完成加载

注意：**.onload()**函数存在覆盖现象

```
**********************************ES6**********************************
*************let 命令*************
  let 声明的变量只在let命令在的代码块内部有效，只能声明一次
     {
       let a = 0
       a // 0
     }
     a //报错

     let a = 1;
     let a = 2;
     a  // Identifier 'a' has already been declared

*************var 命令*************
  var 声明变量在全局范围有效，可以声明多次
       {
      let a = 0;
      var b = 1;
     }
     a  // ReferenceError: a is not defined
     b  // 1

     var b = 3;
     var b = 4;
     b  // 4

*************const 命令*************
const 只声明一个只读变量，声明之后就不准改变了，否则会报错
     const PI = "3.1415926";
     PI  // 3.1415926

// const 其实保证的不是变量的值不变，而是保证变量指向的内存地址所保存的数据不允许改动
//对于简单类型（数值 number、字符串 string 、布尔值 boolean）,值就保存在变量指向的那个内存地址
//因此 const 声明的简单类型变量等同于常量

//复杂类型（对象 object，数组 array，函数
function），变量指向的内存地址其实是保存了一个指向实际数据的指针
//所以 const 只能保证指针是固定的，至于指针指向的数据结构变不变就无法控制了
//所以使用 const 声明复杂类型对象时要慎重。

*************实例演示*************

for (var i = 0; i < 10; i++) { //i在全局范围内都有效，整个循环过程中只有一个i，循环里的十个 setTimeout
                               //是在循环结束后才执行，所以此时的 i 都是 10
  setTimeout(function(){
    console.log(i);
  })
  }
// 输出十个 10
for (let j = 0; j < 10; j++) {// j 只在本轮循环中有效，每次循环的 j 其实都是一个新的变量
                              //所以 setTimeout 定时器里面的 j 其实是不同的变量
 setTimeout(function(){
  console.log(j);
 })
}
// 输出 0123456789
```

```
console.log(a);  //ReferenceError: a is not defined
console.log(b);  //undefined
//变量 b 用 var 声明存在变量提升
//所以当脚本开始运行的时候，b 已经存在了，但是还没有赋值，所以会输出 undefined。
//变量 a 用 let 声明不存在变量提升，在声明变量 a 之前，a 不存在，所以会报错。



**********数据的解构**********
**********数组数据的解构**********
基本解构
   let [a, b, c] = [1, 2, 3];
   // a = 1
   // b = 2
   // c = 3

嵌套
   let [a, [[b], c]] = [1, [[2], 3]];
   // a = 1
   // b = 2
   // c = 3

忽略
   let [a, , b] = [1, 2, 3];
   // a = 1
   // b = 3

不完全解构
   let [a = 1, b] = []; // a = 1, b = undefined

可以使用剩余运算符
   let [a, ...b] = [1, 2, 3];
   //a = 1
   //b = [2, 3]

对于可迭代对象（可遍历对象）皆可进行解构
   let [a, b, c, d, e] = 'hello';
   // a = 'h'
   // b = 'e'
   // c = 'l'
   // d = 'l'
   // e = 'o'

解构有默认值
   let [a = 2] = [undefined]; // a = 2
      //当解构模式有匹配结果，且匹配结果是 undefined 时，会触发默认值作为返回结果

   let [a = 3, b = a] = [];     // a = 3, b = 3         a/b匹配结果为 undefined 触发默认值返回
   let [a = 3, b = a] = [1];    // a = 1, b = 1         a正常解构，b=a 默认解构
   let [a = 3, b = a] = [1, 2]; // a = 1, b = 2         a,b都正常解构

**********对象模型的解构**********
基本解构
   let { foo, bar } = { foo: 'aaa', bar: 'bbb' };
   // foo = 'aaa'
   // bar = 'bbb'
```

```
let { baz : foo } = { baz : 'ddd' };
// foo = 'ddd'
```

可嵌套可忽略

```
let obj = { p: ['hello', {y: 'world'}] };
let {p: [x, { y }] } = obj;
// x = 'hello'
// y = 'world'
let obj = {p: ['hello', {y: 'world'}] };
let {p: [x, {  }] } = obj;
// x = 'hello'
```

不完全解构

```
let obj = {p: [{y: 'world'}] };
let {p: [{ y }, x ] } = obj;
// x = undefined
// y = 'world'
```

剩余运算符

```
let {a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40};
// a = 10
// b = 20
// rest = {c: 30, d: 40}
```

解构有默认值

```
let {a = 10, b = 5} = {a: 3};
// a = 3; b = 5;
let {a: aa = 10, b: bb = 5} = {a: 3};
// aa = 3; bb = 5;
```

**********Symbol**********

Symbol 表示独一无二的值，最大的用法是用来定义对象的唯一属性名

基本用法：

```
let sy = Symbol("KK");
console.log(sy);   // Symbol(KK)
typeof(sy);        // "symbol"

// 相同参数 Symbol() 返回的值不相等
let sy1 = Symbol("kk");
sy === sy1;        // false

//由于每一个 Symbol 的值都是不相等的，所以 Symbol 作为对象的属性名
//可以保证属性不重名
  let sy = Symbol("key1");
```

写法1

```
  let syObject = {};
  syObject[sy] = "kk";
  console.log(syObject);    // {Symbol(key1): "kk"}
```

写法2

```
  let syObject = {
    [sy]: "kk"
  };
  console.log(syObject);    // {Symbol(key1): "kk"}
```

写法3

```
let syObject = {};
Object.defineProperty(syObject, sy, {value: "kk"});
console.log(syObject);   // {Symbol(key1): "kk"}

//Symbol 作为对象属性名时不能用.运算符，要用方括号
//因为.运算符后面是字符串，所以取到的是字符串 sy 属性，而不是 Symbol 值 sy 属性
let syObject = {};
syObject[sy] = "kk";

syObject[sy];  // "kk"
syObject.sy;   // undefined

/*Symbol 值作为属性名时，该属性是公有属性不是私有属性
可以在类的外部访问，但是不会出现在 for...in 、 for...of 的循环中
也不会被 Object.keys() 、 Object.getOwnPropertyNames() 返回

如果要读取到一个对象的 Symbol 属性
可以通过 Object.getOwnPropertySymbols() 和 Reflect.ownKeys() 取到*/
let syObject = {};
syObject[sy] = "kk";
console.log(syObject);

for (let i in syObject) {
  console.log(i);
}    // 无输出

Object.keys(syObject);                          // []
Object.getOwnPropertySymbols(syObject);      // [Symbol(key1)]
Reflect.ownKeys(syObject);                      // [Symbol(key1)]


Symbol.for()
Symbol.for() 类似单例模式会在全局搜索
被登记的 Symbol 中是否有该字符串参数作为名称的 Symbol 值
如果有即返回该 Symbol 值
没有则新建并返回一个以该字符串参数为名称的 Symbol 值
并登记在全局环境中供搜索

let yellow = Symbol("Yellow");
let yellow1 = Symbol.for("Yellow");
yellow === yellow1;     // false

let yellow2 = Symbol.for("Yellow");
yellow1 === yellow2;    // true

Symbol.keyFor()
返回一个已登记的 Symbol 类型值的 key
用来检测该字符串参数作为名称的 Symbol 值是否已被登记
let yellow1 = Symbol.for("Yellow");
Symbol.keyFor(yellow1);   // "Yellow"
```

********** Map & Set **********
**********Map**********

Map对象是“键控集合（keyed collections）”
与普通对象的区别：

1. 常规JavaScript对象的键必须是String或Symbol，Map允许你使用函数、对象和其它简单的类型（包括NaN）作为键

1） key 是字符串

```
var myMap = new Map();
var keyString = "a string";

myMap.set(keyString, "和键'a string'关联的值");

myMap.get(keyString);     // "和键'a string'关联的值"
myMap.get("a string");    // "和键'a string'关联的值"
                          // 因为 keyString === 'a string'
```

2） key 是对象

```
var myMap = new Map();
var keyObj = {},

myMap.set(keyObj, "和键 keyObj 关联的值");

myMap.get(keyObj);        // "和键 keyObj 关联的值"
myMap.get({});            // undefined, 因为 keyObj !== {}
```

3） key 是函数

```
var myMap = new Map();
var keyFunc = function () {},    // 函数

myMap.set(keyFunc, "和键 keyFunc 关联的值");

myMap.get(keyFunc);              // "和键 keyFunc 关联的值"
myMap.get(function() {})         // undefined, 因为 keyFunc !== function () {}
```

4） key是NaN

```
var myMap = new Map();
myMap.set(NaN, "not a number");

myMap.get(NaN);               // "not a number"

var otherNaN = Number("foo");
myMap.get(otherNaN);          // "not a number"
                              //虽然 NaN 和任何值甚至和自己都不相等(NaN !== NaN 返回true)
                              //NaN作为Map的键来说是没有区别的
```

2. 遍历方法：

常规对象中，为了遍历keys、values和entries，你必须将它们转换为数组
如使用Object.keys()、Object.values()和Object.entries()，或者使用for ... in循环
因为常规对象不能直接遍历，另外for ... in循环还有一些限制：它仅仅遍历可枚举属性、非Symbol属性
并且遍历的顺序是任意的
Map可以直接遍历，并且由于它是键控集合，遍历的顺序和插入键值的顺序是一致的
可以使用for ... of循环或forEach方法来遍历Map的entries

for ... of循环

```
var myMap = new Map();
myMap.set(0, "zero");
myMap.set(1, "one");
```

```
// 将会显示两个 log。 一个是 "0 = zero" 另一个是 "1 = one"
for (var [key, value] of myMap) {
  console.log(key + " = " + value);
}
for (var [key, value] of myMap.entries()) {
  console.log(key + " = " + value);
}
/* 这个 entries 方法返回一个新的 Iterator 对象
它按插入顺序包含了 Map 对象中每个元素的 [key, value] 数组。 */

// 将会显示两个log。 一个是 "0" 另一个是 "1"
for (var key of myMap.keys()) {
  console.log(key);
}
/* 这个 keys 方法返回一个新的 Iterator 对象， 它按插入顺序包含了 Map 对象中每个元素的键。
*/

// 将会显示两个log。 一个是 "zero" 另一个是 "one"
for (var value of myMap.values()) {
  console.log(value);
}
/* 这个 values 方法返回一个新的 Iterator 对象，它按插入顺序包含了 Map
对象中每个元素的值。 */
```

.forEach 方法

```
var myMap = new Map();
myMap.set(0, "zero");
myMap.set(1, "one");

// 将会显示两个 logs。 一个是 "0 = zero" 另一个是 "1 = one"
myMap.forEach(function(value, key) {
  console.log(key + " = " + value);
}, myMap)
```

3. 可以调用map.size属性来获取键值数量
   而对于常规对象，为了做到这样你必须先转换为数组，然后获取数组长度
   如：Object.keys({}).length

Map 和 Array 的转换：

```
var kvArray = [["key1", "value1"], ["key2", "value2"]];

// Map 构造函数可以将一个 二维 键值对数组转换成一个 Map 对象
var myMap = new Map(kvArray);

// 使用 Array.from 函数可以将一个 Map 对象转换成一个二维键值对数组
var outArray = Array.from(myMap);
```

Map的合并：

```
var first = new Map([[1, 'one'], [2, 'two'], [3, 'three'],]);
var second = new Map([[1, 'uno'], [2, 'dos']]);

// 合并两个 Map 对象时，如果有重复的键值，则后面的会覆盖前面的，对应值即 uno， dos， three
var merged = new Map([...first, ...second]);
```

Map 数据的克隆：

```
var myMap1 = new Map([["key1", "value1"], ["key2", "value2"]]);
var myMap2 = new Map(myMap1);
```

```
console.log(original === clone);
// 打印 false。 Map 对象构造函数生成实例，迭代出新的对象
```

**********Set**********

Set 对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用

Set 对象存储的值总是唯一的，所以需要判断两个值是否恒等

几个特殊值需要特殊对待：

- … +0 与 -0 在存储判断唯一性的时候是恒等的，所以不重复
- … undefined 与 undefined 是恒等的，所以不重复
- … NaN 与 NaN 是不恒等的，但是在 Set 中只能存一个，不重复

```
let mySet = new Set();

mySet.add(1); // Set(1) {1}
mySet.add(5); // Set(2) {1, 5}
mySet.add(5); // Set(2) {1, 5} 这里体现了值的唯一性
mySet.add("some text");
// Set(3) {1, 5, "some text"} 这里体现了类型的多样性
var o = {a: 1, b: 2};
mySet.add(o);
mySet.add({a: 1, b: 2});
// Set(5) {1, 5, "some text", {…}, {…}}  {…}指的是Object
// 这里体现了对象之间引用不同不恒等，即使值相同，Set 也能存储
```

Set 与 Array 类型转换

```
// Array 转 Set
var mySet = new Set(["value1", "value2", "value3"]);
// 用...操作符，将 Set 转 Array
var myArray = [...mySet];

// String 转 Set
var mySet = new Set('hello');  // Set(4) {"h", "e", "l", "o"}
// 注：Set 中 toString 方法是不能将 Set 转换成 String
```

数组去重

```
var mySet = new Set([1, 2, 3, 4, 4]);
[...mySet]; // [1, 2, 3, 4]
```

并集

```
var a = new Set([1, 2, 3]);
var b = new Set([4, 3, 2]);
var union = new Set([...a, ...b]); // {1, 2, 3, 4}
```

交集

```
var a = new Set([1, 2, 3]);
var b = new Set([4, 3, 2]);
var intersect = new Set([...a].filter(x => b.has(x))); // {2, 3}
```

差集

```
var a = new Set([1, 2, 3]);
var b = new Set([4, 3, 2]);
var difference = new Set([...a].filter(x => !b.has(x))); // {1}
```