

# JavaScript 操作指南(支持 ES6)

作者：向仔州

## 目录

<b>JS 基本书写规则</b>	3
JS 输入输出语法	5
<b>JS 变量使用</b>	6
prompt 和变量，联合使用实现“输出用户名案例”	7
输入数据案例和 var, let 定义变量的区别	8
<b>JS 数组基本使用</b>	8
<b>JS 常量，数据类型，类型转换，算数运算</b>	8
字符串类型	10
字符串拼接	10
布尔类型 boolean, null 类型	11
<b>订单表案例实现</b>	12
<b>JS 运算符使用</b>	14
<b>JS 判断语句 if 单分支，双分支，三元运算符</b>	16
<b>数字补 0 案例</b>	18
<b>Switch 分支语句</b>	18
<b>断点调试方法，while 循环</b>	19
continue 和 break 退出循环	19
<b>简易 ATM 取款机案例</b>	20
<b>for 循环的用法</b>	21
数组操作	22
数组筛选案例	23
数组和 for 循环实现柱状图案例	24
<b>函数的使用</b>	27
匿名函数实现	30
<b>逻辑中断</b>	32
<b>JS 对象基本使用</b>	33
遍历对象，将对象里面每个变量都获取出来	34
<b>渲染信息表案例</b>	35
<b>JS 内置对象</b>	37
<b>Web API 使用</b>	38
DOM 树和 DOM 对象	40
操作标签常用属性，修改标签样式属性，修改表单标签属性	43
轮播图简单案例	46
获取页面输入框内容(获取表单的值)	49
操作表单元素，属性	50
H5 自定义属性	50
定时器间歇函数	51

轮播图定时器切换.....	52
事件监听.....	53
随机点名案例.....	54
事件监听版本及鼠标事件.....	56
轮播图点击切换案例.....	56
焦点事件.....	59
键盘事件，发布评论案例.....	60
事件对象.....	62
评论回车发布案例.....	64
环境对象 this.....	66
TAB 切换案例.....	67
全选文本框案例.....	68
事件流.....	70
事件委托.....	74
事件委托实现 tab 栏切换 .....	76
阻止冒泡(比如阻止后面的程序运行) .....	78
页面加载事件和页面滚动事件.....	79
页面尺寸事件.....	82
页面向下滚动之后探出顶部导航栏案例.....	84
日期对象使用.....	86
时间戳.....	87
DOM 节点操作(对标签进行增删改查).....	89
M 端事件(就是移动端事件，比如手机) .....	93
<b>swiper 插件使用</b> .....	94
swiper 组件查找官方库，学习定制轮播图方法 .....	95
<b>学生信息表案例</b> .....	96
<b>BOM 对象和延时函数 setTimeout</b> .....	101
<b>事件循环 eventloop(理论讲解)</b> .....	102
<b>location 对象用与 JS 中跳转页面地址使用</b> .....	103
<b>navigator 对象，用于记录当前浏览器下的信息</b> .....	104
<b>history 对象，浏览器前进后退功能</b> .....	105
<b>localStorage 本地存储</b> .....	105
<b>map 和 join 方法数据字符串拼接</b> .....	109
<b>正则表达式</b> .....	111
<b>元字符之边界符</b> .....	112
<b>元字符之量词</b> .....	112
<b>JS 新语法(ES6 标准)</b> .....	115
作用域和作用域链.....	115
JS 垃圾回收机制和闭包.....	116
函数动态参数.....	117
<b>箭头函数，JS 的 ES6 标准(重要)</b> .....	118
<b>数组解构(重要)</b> .....	119
<b>对象解构(重要)</b> .....	120
<b>forEach 遍历数组</b> .....	120

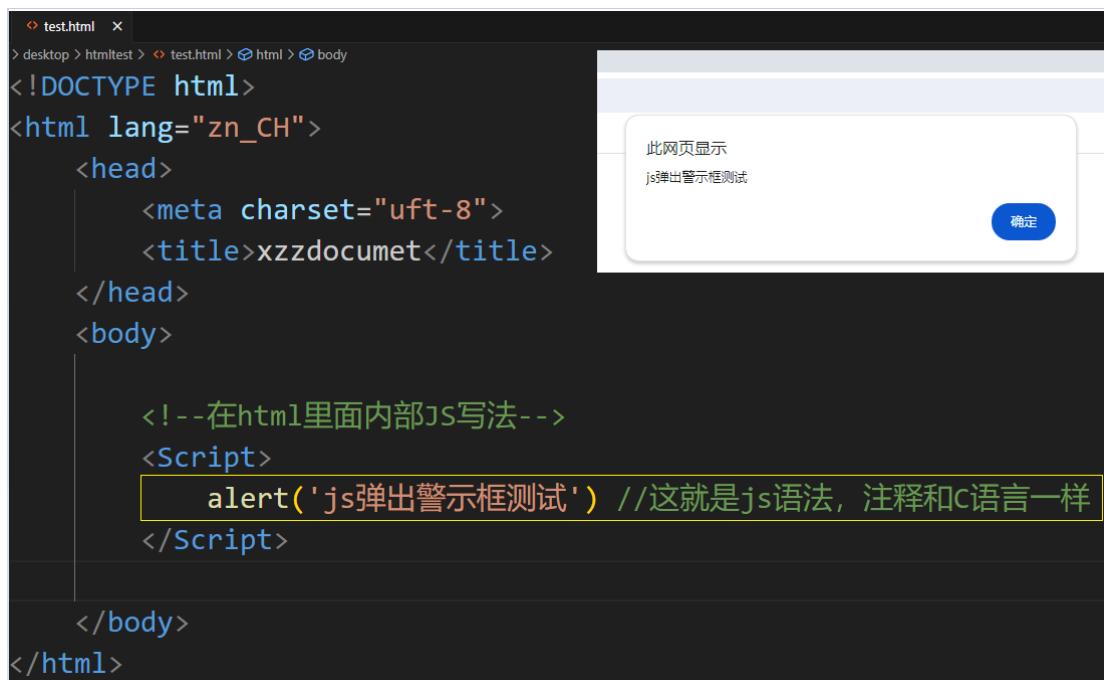
# JS 基本书写规则

## 1. 内部 JavaScript

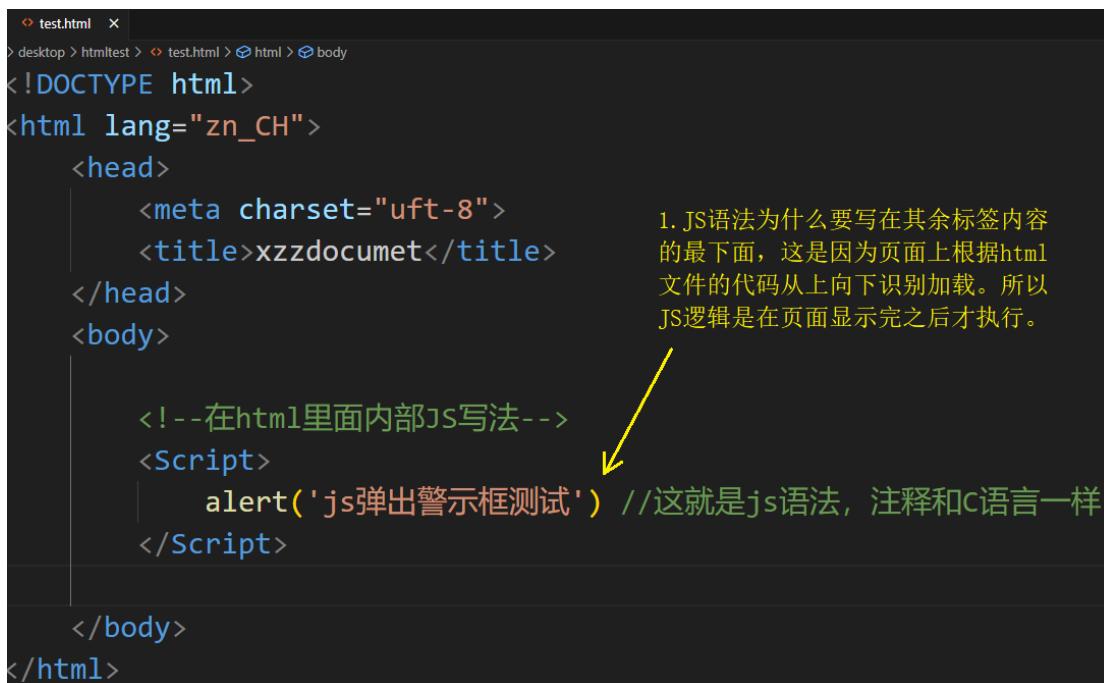
直接写在html文件里，用script标签包住

**规范：** script标签写在</body>上面

**拓展：** alert('你好，js') 页面弹出警告对话框



```
<!DOCTYPE html>
<html lang="zn_CH">
  <head>
    <meta charset="uft-8">
    <title>xzzdocument</title>
  </head>
  <body>
    <!--在html里面内部JS写法-->
    <Script>
      alert('js弹出警示框测试') //这就是js语法，注释和c语言一样
    </Script>
  </body>
</html>
```



```
<!DOCTYPE html>
<html lang="zn_CH">
  <head>
    <meta charset="uft-8">
    <title>xzzdocument</title>
  </head>
  <body>
    <!--在html里面内部JS写法-->
    <Script>
      alert('js弹出警示框测试') //这就是js语法，注释和c语言一样
    </Script>
  </body>
</html>
```

1. JS语法为什么要写在其余标签内容的最下面，这是因为页面上根据html文件的代码从上向下识别加载。所以JS逻辑是在页面显示完之后才执行。

## 2. 外部 JavaScript

代码写在以.js结尾的文件里

**语法：**通过script标签，引入到html页面中。

The screenshot shows a file explorer window with files '55.png', 'my.js', and 'test.html'. A note says '在html同级目录下单独建立个js文件, xx.js'. To the right is a browser window showing an alert box with the message '我是外部js文件'.

Below is the code for 'test.html':

```
D:\> user > xiang > desktop > htmltest > < test.html > < html
1  <!DOCTYPE html>
2  <html lang="zn_CH">
3  |   <head>
4  |   |       <meta charset="uft-8">
5  |   |       <title>xzzdocument</title>
6  |   </head>
7  |   <body>
8
9      <!--在html引入外部JS写法-->
10     <Script src="./my.js"></Script>    <!--符号./就是引入当前目录下js文件-->
11     </body>
12 </html>
```

At the bottom, there is another code block with annotations:

```
<!DOCTYPE html>
<html lang="zn_CH">
|   <head>
|   |       <meta charset="uft-8">
|   |       <title>xzzdocument</title>
|   </head>
|   <body>
|
|       <!--在html引入外部JS写法-->
|       <Script src="./my.js">
|           <AbortController>    .
|       </Script>    <!--符号./就是引入当前目录下js文件-->
|   </body>
</html>
```

Annotations explain that when you include an external JS file, you cannot write content between the Script tags, or it won't execute and will报错 (throw an error).

## 3. 内联 JavaScript

代码写在标签内部

**语法：**

注意：此处作为了解即可，但是后面vue框架会用这种模式

```
<body>
|   <button onclick="alert('逗你玩~~~')">点击我月薪过万</button>
</body>
```

就是在标签控件内部写 JS。

## 1.2 JavaScript 怎么写

- 单行注释

- 符号: //

- 作用: //右边这一行的代码会被忽略

- 快捷键: **ctrl + /**

```
JS my.js x
> desktop > htmltest > JS my.js
alert('我是外部js文件') //单行注释
```

- 块注释

- 符号: /\* \*/

- 作用: 在/\* 和 \*/之间的所有内容都会被忽略

- 快捷键: shift + alt + a

```
JS my.js x
g > desktop > htmltest > JS my.js
alert('我是外部js文件') //单行注释

/*
    这就是一个
    多行注释方法，和C语言一样
*/
```

JS代码用分号结束，和C语言一样。

```
JS my.js x
> desktop > htmltest > JS my.js
alert('第1句JS代码');
alert('第2句JS代码');
```



## JS 输入输出语法

`document.write(.....) //填入输出内容`

```
JS my.js x
> desktop > htmltest > JS my.js
alert('第1句JS代码');
document.write('输出语法内容')

1. 先执行警示
2. 点击确定后，页面输出内容
    确定
    输出语法内容
```

```
JS my.js x
> desktop > htmltest > JS my.js
document.write('输出第一行内容')
document.write('输出第二行内容') //两行字符内容连接在一起的
document.write('<h1>333333</h1>') //可以直接输出h标签，H标签可以换行
```

输出第一行内容输出第二行内容

**333333**

**语法2:**

```
alert('要出的内容')
```

**作用:** 页面弹出警告对话框

**语法3:**

```
console.log('控制台打印')
```

**作用:** 控制台输出语法，程序员调试使用

```
console.log(.....) //程序员调试输出内容函数
```

The screenshot shows the Chrome DevTools console tab. Inside, there is a code snippet and its execution results. The code is:

```
document.write('输出第一行内容')
document.write('输出第二行内容')      //两行字符内容连接在一起的
document.write('<h1>33333</h1>') //可以直接输出h标签，H标签可以换行
console.log('程序调试使用的输出方式') //这个就是程序要经常用的调试输出方式
```

The results show the output of the document.write statements followed by the console.log output:

```
输出第一行内容输出第二行内容
33333
程序调试使用的输出方式
```

A callout arrow points from the text "在页面浏览器审查元素里面console可以看到" to the word "console" in the log output.

```
prompt(.....) //输入框，请求输入内容
```

The screenshot shows the Chrome DevTools console tab. Inside, there is a code snippet and its execution results. The code is:

```
prompt('请输入内容：')
```

The results show the output of the prompt statement:

```
此网页显示
请输入内容：
```

A green arrow points from the text "页面执行到这一句，弹出输入框" to the word "prompt" in the code. Another green arrow points from the input field in the screenshot to the input field in the code.

## JS 变量使用

```
let 变量名
```

- 声明变量有两部分构成：声明关键字、变量名（标识）

The screenshots show three different ways to declare and assign a variable named 'age' in JavaScript:

- Screenshot 1:** Shows the declaration and assignment of 'age' followed by an alert call:

```
let age //声明一个年龄的变量
age = 18 //变量赋值
alert(age)
```
- Screenshot 2:** Shows the declaration and assignment of 'age' followed by an alert call:

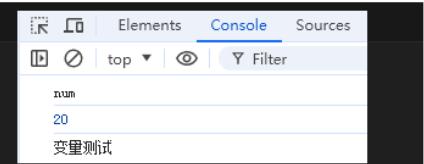
```
let age //声明一个年龄的变量
age = 18 //变量赋值
alert(age)
```
- Screenshot 3:** Shows the declaration and assignment of 'age' followed by a console.log call:

```
let age //声明一个年龄的变量
age = 18 //变量赋值
console.log(age)
```

Each screenshot includes a callout arrow pointing from the variable name 'age' in the code to the corresponding 'age' value displayed in the alert or log output window.

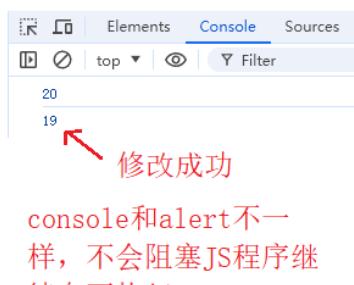
```
JS my.js x
> desktop > htmltest > JS my.js > ...
let age = 18 //声明一个年龄的变量
console.log(age)
```

```
JS my.js x
> desktop > htmltest > JS my.js > ...
let num = 20
let uname = '变量测试'
console.log('num') //变量不需要加引号，只有字符串才加引号
console.log(num)
console.log(uname)
```



### 变量值运行的时候修改

```
JS my.js x
g > desktop > htmltest > JS my.js > ...
let num = 20
console.log(num)
num = 19
console.log(num)
```



console和alert不一样，不会阻塞JS程序继续向下执行。

```
JS my.js x
g > desktop > htmltest > JS my.js > ...
let num = 20, name1 = '多变量测试'
console.log(num, name1)
```



### prompt 和变量，联合使用实现”输出用户名案例”

```
JS my.js x
g > desktop > htmltest > JS my.js > ...
let uname //name好像是JS内置名称，不建议使用
uname = prompt('输入数据') //数据输入赋值给变量
document.write(uname) //数据输出
```



```
JS my.js x
g > desktop > htmltest > JS my.js > ...
变量相互复制
let num1 = 10
let num2 = 50
let temp

temp = num2 //temp=50
num2 = num1 //num2=10
console.log(num1,num2)
console.log(temp)
```

#### 1. 规则：变量定义要求

- 不能用关键字
  - ✓ 关键字：有特殊含义的字符，JavaScript 内置的一些英语词汇。例如：let、var、if、for等
- 只能用下划线、字母、数字、\$组成，且数字不能开头
- 字母严格区分大小写，如 Age 和 age 是不同的变量

## 输入数据案例和 var, let 定义变量的区别

The screenshot illustrates the behavior of `var` and `let` in JavaScript. On the left, a code snippet demonstrates using `prompt` to input three pieces of data (character, number, symbol) and then concatenating them with `document.write`. On the right, three separate input dialogs are shown: '请输入字符' (Input Character) with 'ABCDE' and '第1次输入' (First Input), '输入数字' (Input Number) with '12345' and '第2次输入' (Second Input), and '输入符号' (Input Symbol) with '.....' and '第3次输入' (Third Input). The final output 'ABCDE12345.....' is labeled '输出结果' (Output Result) with the note '三个数据拼接在一起的' (Three data concatenated together).

### let和var区别是什么？

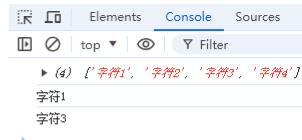
旧版本的JS是用var来声明变量的，新版本ES6的JS大部分都是用let声明变量。  
1. 我们用var声明变量不赋值直接使用

2. 我们先给变量赋值，然后再定义变量

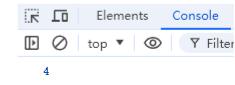
The screenshot compares `var` and `let` declarations. It shows two browser consoles. The first console shows `var num; console.log(num)` followed by `num = 10` and `console.log(num)` outputting `10`. A red arrow points from the assignment to the output. The second console shows `var num` followed by `num = 10` and `console.log(num)` outputting `10`. A red arrow points from the assignment to the output. The third part shows `let num; num = 10; console.log(num)` resulting in an error message: 'Uncaught ReferenceError: Cannot access 'num' at my\_js:1:5'. A red arrow points from the error message to the code. The fourth part shows `let num; num = 10; console.log(num)` outputting `10`.

## JS 数组基本使用

```
let strarr = ['字符1', '字符2', '字符3', '字符4', ]  
console.log(strarr) //输出数组中所有数据  
console.log(strarr[0]) //输出数组第1个元素  
console.log(strarr[2]) //输出数组第3个元素
```

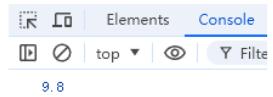


```
let strarr = ['字符1', '字符2', '字符3', '字符4', ]  
console.log(strarr.length) //计算数字元素个数
```

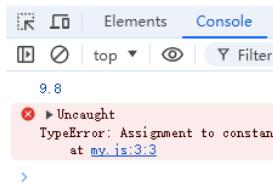


## JS 常量，数据类型，类型转换，算数运算

```
const G = 9.8 //固定不变的值定义为常量  
console.log(G)
```

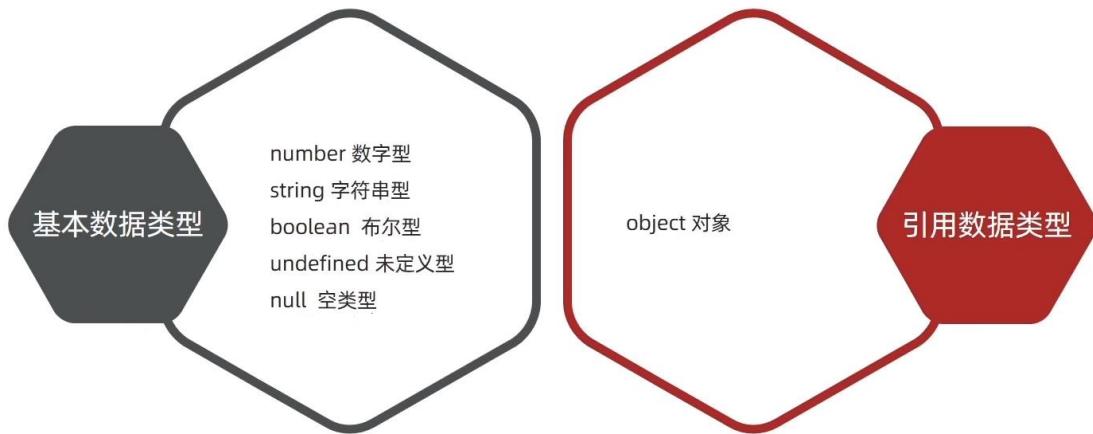


```
const G = 9.8 //固定不变的值定义为常量  
console.log(G)  
G = 20 ← 如果我去修改固定不变  
的值，运行就会报错  
console.log(G)
```



- 注意：常量不允许重新赋值，声明的时候必须赋值（初始化）
- 小技巧：不需要重新赋值的数据使用const

JS数据类型整体分为两大类



```
//int NumDec = 10 强数据类型语言如C语言，就是定义的变量都要求赋值的数据类型一致
let NumDec = '字符串类型' //弱数据类型就是，只有你赋值之后，变量才知道是什么类型
console.log(NumDec)
NumDec = 20           //你看，我给字符串变量输入整形，变量就变成整形数据类型了
console.log(NumDec)   //这就是弱数据类型
```



数字可以有很多操作，比如，乘法 \* 、除法 / 、加法 + 、减法 - 等等

- +: 求和
- -: 求差
- \*: 求积
- /: 求商
- %: 取模（取余数）

```
console.log(1 + 1)    //=2
console.log(5 - 2)    //=3
console.log(2 * 2)    //=4
console.log(4 / 2)    //=2
console.log(5 % 3)    //=2 余2
```

```
2
3
4
2
2
```

1. 算术运算符有那几个常见的？

➤ + - \* / %

2. 算术运算符优先级怎么记忆？

➤ 先乘除取余，后加减，有小括号先算小括号里面的

3. 取余运算符开发中的使用场景是？

➤ 来判断某个数字是否能被整除

```
let r = prompt('输入一个数值')
let re = 2.5 * r * r
console.log(re)
```

```

let xstr = '字符串'
temp = xstr - 2 //字符串减去数字得到NaN, 意思就是计算机也不知道是得到什么
console.log(temp)

```

## 字符串类型

通过单引号（'） 、双引号（"）或反引号(`)包裹的数据都叫字符串，

单引号和双引号没有区别，推荐使用单引号。

```

let str = '单引号字符串'
let tstr = "双引号字符串"
let Fstr = `反引号`
console.log(str)
console.log(tstr)
console.log(Fstr)

```

这三种符号都是字符串

```

let num = 11
let tstr = '11' //带引号的数字就是字符串了
let kstr = '' //空字符串
console.log(num)
console.log(tstr)
console.log(kstr)

```

2. 单引号/双引号可以互相嵌套，但是不以自己嵌套自己（口诀：外双内单，或者外单内双）

## 字符串拼接

```

console.log('pink' + '中文') //使用‘+’号拼接字符串
let unmae = "歌名" + "歌名"
let song = "忘情水" + "忘情水"
document.write(unmae + song)
console.log(unmae + song)

```

```

let age = 19
document.write('拼接数字' + age + '完成')

```

拼接数字19完成

但是这种拼接字符串有时候手残了容易少些引号什么的导致报错。

使用模板字符串来解决拼接字符串

```

let age = 19
document.write(`拼接数字${age}完成`) //模板字符串就是用${变量}

```

document.write(`大家好, 我叫\${name}, 今年\${age}岁`)

模板字符串使用起来较简单



## 布尔类型 boolean, null 类型

```
let cool = true
console.log(cool)

let obj = null
console.log(obj)
```

### null 和 undefined 区别：

- undefined 表示没有赋值
- null 表示赋值了，但是内容为空

```
console.log(null + 1)
console.log(undefined + 1)
```

虽然变量为空，但是加1之后数据放进去了。  
可是undefined系统不知道是什么东西，所以为NaN

## null 是什么类型？ 开发场景是？

- 空类型
- 如果一个变量里面确定存放的是对象，如果还没准备好对象，可以放个null

## typeof 变量 //输出数据类型，为什么要进行数据类型转换呢？

```
let num = 10
console.log(typeof num)

let str = 'PINK'
console.log(typeof str)

let boolv = false
console.log(typeof boolv)

let un
console.log(typeof un)

let obj = null
console.log(typeof obj)
```

number  
string  
boolean  
undefined  
object

注意，在JS当中，prompt, 表单，复选框，多选框，单选框，取出来的值默认都是字符串数据类型。

`Number(变量)`//将字符转换成数字，可以说小数和整数

```
let num = prompt('请输入基本工资: ') 未进行数据转换，输入的都是字符串。  
console.log(typeof num)  
let num2 = prompt('再输入奖金: ') 未进行数据转换，输入的都是字符串。  
console.log(typeof num2)  
  
console.log(num + num2) //基本工资 + 奖金相加 = 你看是不是想要的数
```



```
let num = prompt('请输入基本工资: ')  
console.log(Number(num)) //Number 字符转换成数字  
let num2 = prompt('再输入奖金: ')  
console.log(Number(num2)) //Number 字符转换成数字
```

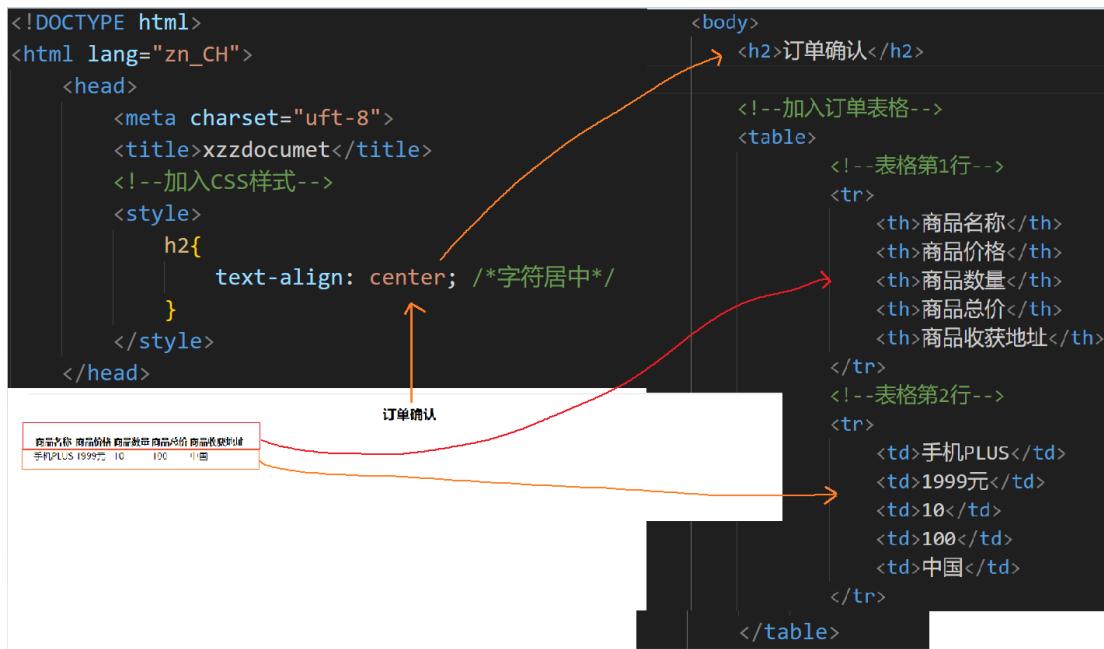
Number字符转小数
1000
3000
4000

```
console.log(Number(num) + Number(num2)) //基本工资 + 奖金相加 = 想要的结果
```

## 订单表案例实现

### 订单付款确认页面

商品名称	商品价格	商品数量	总价	收货地址
小米手机青春PLUS	1999元	3	5997元	北京顺义黑马程序员



```
<!--加入CSS样式-->
<style>
    h2{
        text-align: center; /*字符居中*/
    }
    table,th,td{ 在样式中给th, td表格加边框,
        使用table全局表格
        border: 1px solid #000; /*表格加边框*/
    }

```

商品名称	商品价格	商品数量	商品总价	商品收获地址
手机PLUS	1999元	10	100	中国

```
<style>
    h2{
        text-align: center; /*字符居中*/
    }
    table,th,td{
        border: 1px solid #000; /*表格加边框*/
    }
    table{
        border-collapse: collapse; /*单独给表格合并边框*/
    }
</style>
```

商品名称	商品价格	商品数量	商品总价	商品收获地址
手机PLUS	1999元	10	100	中国

```
<body>
    <h2>订单确认</h2>
    <!--在html中取消掉表格标签，将表格标签放入JS实现
    <!--在html引入外部JS写法-->
    <Script src="./my.js"></Script>    <!--符号./就是引入当前目录下js文件-->
</body>
```

```
let price = prompt('请输入商品价格: ')
let num = prompt('请输入商品数量: ')
let address = prompt('请输入收获地址: ')

let total = price * num //计算总额
```

document.write(` ← 模板字符串也可以输出html格式，使用`  
` 实现模板字符串`

商品名称	商品价格	商品数量	商品总价	商品收获地址
手机PLUS	\${price}元	\${num}	\${total}元	\${address}

`) ← 模板字符串中写入表格  
` 标签` ← \${} 就是给html标签用  
` 来引入JS变量的方法` ← 使用模板字符串变量导  
` 入JS变量，实现数字可  
` 显示` ← `

请输入商品价格:

**确定**

请输入商品数量:

**确定**

请输入收获地址:

**确定**

**数据被页面初始化手动写入修改**

商品名称	商品价格	商品数量	商品总价	商品收获地址
手机PLUS	1000元	50	50000元	地址显示

- 下面可能出现什么问题？如何解决？

```
let num1 = prompt('请输入第一个数值：')
let num2 = prompt('请输入第二个数值：')
alert(`两者相加的结果是: ${num1 + num2}`)
```

分析：

- 因为prompt获取过来的是字符型，所以会出现字符相加的问题
- prompt如果出现相加记得要转为数字型，可以利用+最简单

```
let num1 = +prompt('请输入第一个数值：')
let num2 = +prompt('请输入第二个数值：')
alert(`两者相加的结果是: ${num1 + num2}`)
```

## JS 运算符使用

```
let num = 1
num + 1
console.log(num) //=1

num = num + 1 //这样加num才对
console.log(num)

num +=1 //等价于num = num +1
console.log(num)
```

```
let i = 10
console.log(++i + 1) //先++i = 11, 然后再+1 = 12
```

```
let i = 10
++i 变量先自加
console.log(i)
```

```
let i = 10
console.log(i++ + 1) //i++就是后加, 那么这种情况就是代码执行完了才i++
```

你看, i++的部分就没有纳入运算, 因为i++是后加, 是运算完之后再加

```

console.log(3 > 5) //大于号
console.log(3 >= 3) //大于等于
console.log(2 == 2) //两边是否相等
console.log(2 == '2') //数字和字符串是否相等，这句绝对会认为相等
console.log(2 === '2') //全等，判断值和数据类型都一样才行，  

//以后判断两边数据是否相等，最好使用3等号

console.log(2 !== '2') //不等于

```

Elements Console  
top Filter  
false  
true  
true  
true ← 这就是==的问题  
false  
true  
>

```

console.log('a' > 'b') //字符ASCII顺序判断，a<b才对
console.log('aa' > 'bb') //字符串比较
console.log('aa' < 'bb') //字符串比较

```

Elements Console  
top Filter  
false  
false  
true  
>

字符串比较是先比较两边字符串第1个字符，如果相等  
如果第1个字符相等，就比较第2个字符

```
console.log('aa' < 'aac') //字符串比较
```

Elements Console  
top Filter  
true  
>

字符串少的比字符串多的小

```

console.log(true && true) //逻辑与，两边为真，最后为真
console.log(false && true) //逻辑与，两边不一致，最好为假
console.log(3 < 5 && 3 > 2) //两边成立，结果为真
console.log(3 < 5 && 3 < 2) //有一边不成立，结果为假

console.log(true || true) //逻辑或，有一个为真，结果为真
console.log(false || true) //逻辑或，有一个为真，结果为真

console.log(!true) //逻辑非，取反
console.log(!false) //逻辑非，取反

```

Elements Console  
top Filter  
true  
false  
true  
false  
true  
true  
false  
true  
>

```

let num = 6
console.log(num > 5 && num < 10) //结果为真

```

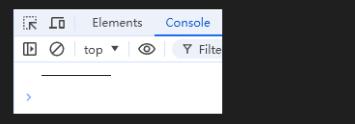
优先级	运算符	顺序
1	小括号	()
2	一元运算符	++ -- !
3	算数运算符	先 * / % 后 + -
4	关系运算符	> >= < <=
5	相等运算符	== != === !==
6	逻辑运算符	先 && 后
7	赋值运算符	=
8	逗号运算符	,

## JS 判断语句 if 单分支，双分支，三元运算符

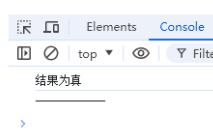
```
if(2) //结果为真
{
    console.log('结果为真')
}
console.log('-----')
```



```
if('') //空字符串为假(所以除了空字符串其它字符串都为真)
{
    console.log('结果为真')
}
console.log('-----')
```



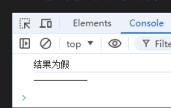
```
let score = 1000
if(score > 700)
{
    console.log('结果为真')
}
console.log('-----')
```



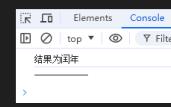
```
let uname = 'pink'
let pwd = 123456
if(uname === 'pink' && pwd === 123456) //字符串判断, 数值判断
{
    console.log('结果为真')
}
else
{
    console.log('结果为假')
}
console.log('-----')
```



```
let uname = 'pink'
let pwd = 123456
if(uname === 'pin' && pwd === 123456) //字符串判断, 数值判断
{
    console.log('结果为真')
}
else
{
    console.log('结果为假')
}
console.log('-----')
let year = 2000
```



```
if(year % 4 === 0 && year % 100 !== 0 || year % 400 === 0) //多逻辑判断
{
    console.log('结果为闰年')
}
else
{
    console.log('结果不为闰年')
}
```



```

let year = 1000

if(year % 4 === 0 && year % 100 !== 0 || year % 400 === 0) //多逻辑判断
{
    console.log('结果为闰年')
}
else
{
    console.log('结果不为闰年')
}
console.log('-----')

```

<pre> let score = 1000  if(score &gt; 90) //多逻辑判断 {     console.log('&gt;90') } else if(score &gt;= 70) {     console.log('&gt;=70') } else if(score &gt;= 60) {     console.log('&gt;=60') } else {     console.log('不成立') } console.log('----') </pre>	<pre> let score = 65  if(score &gt; 90) //多逻辑判断 {     console.log('&gt;90') } else if(score &gt;= 70) {     console.log('&gt;=70') } else if(score &gt;= 60) {     console.log('&gt;=60') } else {     console.log('不成立') } console.log('----') </pre>	<pre> let score = 40  if(score &gt; 90) //多逻辑判断 {     console.log('&gt;90') } else if(score &gt;= 70) {     console.log('&gt;=70') } else if(score &gt;= 60) {     console.log('&gt;=60') } else {     console.log('不成立') } console.log('----') </pre>
--	--	--

<pre> console.log(3 &gt; 5 ? 3:5 ) //返回5 </pre>	<pre> 3 &lt; 5 ? console.log('真'): console.log('假') //三元运算符可以这样判断 </pre>	<pre> let sum = 3 &lt; 5 ? 3 : 5 //返回3 console.log(sum) </pre>
---	--	--

## 数字补0案例

### 数字补0案例

需求：用户输入1个数，如果数字小于10，则前面进行补0，比如09 03等

①：为后期页面显示时间做铺垫

22:00 点场距结束

②：利用三元运算符 补0计算

00 : 02 : 18

```
let num = 5 //我想在5前面补0，变成05
num < 10 ? 0 + num : num //num < 10补0，不 < 10 原数不变
console.log(num)
```

我发现数字5前面没有补0

5

```
let num = 5 //我想在5前面补0，变成05
num = num < 10 ? 0 + num : num //三目运算结果返回给num
console.log(num) \可能是要运算之后最后赋值给num才行
```

最后发现还是不行

5

原来补0是字符形式才行

```
let num = '5' //我想在5前面补0，变成05
num = num < 10 ? 0 + num : num //三目运算结果返回给num
console.log(num)
```

05

## Switch 分支语句

```
switch(1)
{
    case 1:
        console.log('选择1111')
    case 2:
        console.log('选择2222')
    case 3:
        console.log('选择3333')
    default:
        console.log('没有符合条件')
}
```

1. 分支语句如果不加break，那么每一个case都会被执行  
选择1111  
选择2222  
选择3333  
没有符合条件

```
switch(3) 3. 选择分支3
{
    case 1:
        console.log('选择1111')
        break
    case 2:
        console.log('选择2222')
        break
    case 3:
        console.log('选择3333')
        break
    default:
        console.log('没有符合条件')
}
```

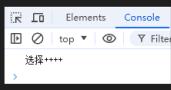
```
switch(1)
{
    case 1:
        console.log('选择1111')
        break 2. 加入break之后，符合条件的语句执行之后就会跳出
    case 2:
        console.log('选择2222')
        break
    case 3:
        console.log('选择3333')
        break
    default:
        console.log('没有符合条件')
}
```

```

let sp = '+' // Switch符号也可以做选择

switch(sp)
{
    case '+':
        console.log('选择++++')
        break
    case '-':
        console.log('选择----')
        break
    case '*':
        console.log('选择*****')
        break
    default:
        console.log('没有符合条件')
}

```



## 断点调试方法，while 循环

1. 我有3行代码

```

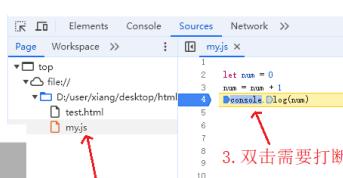
let num = 0
num = num + 1
console.log(num)

```

1.1 单击确认

5. 你发现浏览器上灰色的，这是因为程序只执行到断点位置，后面的代码没有执行。

4. 记住，这时候重新刷新浏览器。



3. 双击需要打断点的地方

2. 浏览器运行，我选择审查元素sources，选择我的js文件

5. 我们用while循环  
while(*i* <= 3) 来试试断点调试

```

let i = 1
while(i <= 3)
{
    document.write('我要循环三次<br>')
    i++
}

```

6. 正是执行3次

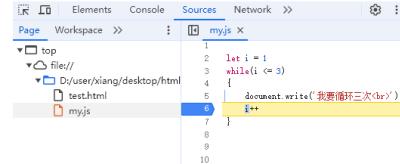
我要循环三次

我要循环三次

我要循环三次

11. 实现第2次打印  
12. 第2次打印出来了

10. 代码运行过了断点



7. 现在进入断点调试，在*i*++这一句打断点。



9. 点击浏览器符号，单步运行。



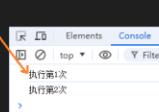
10. 代码运行过了断点

## continue 和 break 退出循环

```

let i = 1
while(i <= 5)
{
    console.log(`执行第${i}次`) // 使用``反引号哦
    i++
    if(i === 3) 执行两次就退出循环了
    {
        break // 退出整个循环
    }
}

```



当代码运行到continue的时候，下面的代码就不执行了，直接跳转到while开始的地方重新执行。

```

let i = 1
while(i <= 5)
{
    if(i === 3)
    {
        continue // 结束当前一次循环
    }
    console.log(`执行第${i}次`) // 使用``反引号哦
    i++
}

```

代码最先执行两次，然后*i*=3，出现了continue，代码就一直在while和if之间死循环。

## 简易 ATM 取款机案例

### 简易ATM取款机案例

需求：用户可以选择存钱、取钱、查看余额和退出功能

分析：

- ①：循环的时候，需要反复提示输入框，所以提示框写到循环里面
- ②：退出的条件是用户输入了 4，如果是4，则结束循环，不在弹窗
- ③：提前准备一个金额预先存储一个数额
- ④：取钱则是减法操作，存钱则是加法操作，查看余额则是直接显示金额
- ⑤：输入不同的值，可以使用switch来执行不同的操作

```
while(true) //死循环
{
    let re = +prompt(`选择操作:
        1.存钱
        2.取钱
        3.查看余额
        4.退出
    `) //在prompt前加+号，将字符串转成数字

    if(re === 4) //用户输入4，退出死循环
    {
        break
    }
}
```

选择操作:  
1.存钱  
2.取钱  
3.查看余额  
4.退出

用户输入1, 2, 3都会不停的弹出本窗口，因为是死循环

1 确定

选择操作:  
1.存钱  
2.取钱  
3.查看余额  
4.退出

用户输入4，退出死循环，窗口不见

4 确定

订单确认 退出死循环，执行后面的页面程序

```
let money = 100
while(true) //死循环
{
    let re = +prompt(`选择操作:
        1.存钱
        2.取钱
        3.查看余额
        4.退出
    `) //在prompt前加+号，将字符串转成数字

    if(re === 4) //用户输入4，退出死循环
    {
        break
    }
}

switch(re)
{
    case 1:
        let cun = +prompt('请输入存款金额')
        money = money + cun
        break
    case 2:
        let qu = +prompt('请输入取款金额')
        money = money - qu
        break
    case 3:
        alert(`你的银行卡余额度${money}`)
        break
}
```

选择操作:  
1.存钱  
2.取钱  
3.查看余额  
4.退出

存入金额

请输入存款金额  
1000 存入额度

确定

选择操作:  
1.存钱  
2.取钱  
3.查看余额  
4.退出

查询金额

你的银行卡余额度1100  
因为money变量初始值为100，所以存入1000就是1100

3 确定

## for 循环的用法

```
for (变量起始值; 终止条件; 变量变化量) {
    // 循环体
}
```

```
for(let i = 1; i <= 5; i++) {
    console.log(`i = ${i}`)
}
```

```
let arr = ['数组1', '数组2', '数组3', '数组4', '数组5'] //5个元素
for(let i = 1; i <= 5; i++) //最好遍历4次0~4
{
    console.log(`arr = ${arr[i]}`)
}
```

```
let arr = ['数组1', '数组2', '数组3', '数组4', '数组5'] //5个元素
for(let i = 0; i <= arr.length - 1; i++) //5个元素, 长度-1就是遍历0~4
{
    console.log(`arr = ${arr[i]}`)
}
```

```
let arr = ['数组1', '数组2', '数组3', '数组4', '数组5'] //5个元素
for(let i = 0; i <= arr.length - 1; i++) //5个元素, 长度-1就是遍历0~4
{
    if(i === 3) ←2. 这是因为当i 加到3就会因为continue跳转, 导致数组
    [3]执行不了, 数组[3]里面装的‘数组4’
    {
        continue //循环到第3次后面代码不执行
    }
    console.log(`arr = ${arr[i]}`)   3. i加到4就不会跳
    转, 所以‘数组5’输出显示了
}
```

```
let arr = ['数组1', '数组2', '数组3', '数组4', '数组5'] //5个元素
for(let i = 0; i <= arr.length - 1; i++) //5个元素, 长度-1就是遍历0~4
{
    if(i === 3)
    {
        break
    }
    console.log(`arr = ${arr[i]}`)
}
```

```

for(let i = 1; i <= 3; i++) 循环嵌套
{
    console.log(`第1层循环i = ${i}`)
    for(let j = 1; j <= 5; j++)
    {
        console.log(`第2层循环j = ${j}`)
    }
}

```

Elements Console

第1层循环i = 1  
第2层循环j = 1  
第2层循环j = 2  
第2层循环j = 3  
第2层循环j = 4  
第2层循环j = 5  
第1层循环i = 2  
第2层循环j = 1  
第2层循环j = 2  
第2层循环j = 3  
第2层循环j = 4  
第2层循环j = 5  
第1层循环i = 3  
第2层循环j = 1  
第2层循环j = 2  
第2层循环j = 3  
第2层循环j = 4  
第2层循环j = 5

```

for(let i = 1; i <= 5; i++)
{
    for(let j = 1; j <= 5; j++)
    {
        document.write('☆')
    }
    document.write('<br>') //换行显示
}

```

☆☆☆☆☆  
☆☆☆☆☆  
☆☆☆☆☆  
☆☆☆☆☆  
☆☆☆☆☆

## 数组操作

```

let arr = [2,6,1,7,4]
let max = arr[0] 判断数组最
let min = arr[0] 大值最小值
for(let i = 1; i <= 5; i++)
{
    if(max < arr[i])
    {
        max = arr[i]
    }

    if(min > arr[i])
    {
        min = arr[i]
    }
}

console.log(`最大值:${max}`)
console.log(`最小值:${min}`)

```

Elements Console

最大值:7  
最小值:1

```

let arr = [2,6,1,7,4]
let max = arr[0] 判断数组最
let min = arr[0] 大值最小值
for(let i = 1; i <= 5; i++)
{
    //也可以用三元运算符做大小判断
    max < arr[i] ? max = arr[i] : max
    min > arr[i] ? min = arr[i] : min
}

console.log(`最大值:${max}`)
console.log(`最小值:${min}`)

```

Elements Console Sources Network

每一项都加入了新数据

```

let arr = ['pink','red','green']
for(let i = 0; i <= 5; i++)
{
    arr[i] = arr[i] + '新数据' //pink新数据
}

console.log(arr)

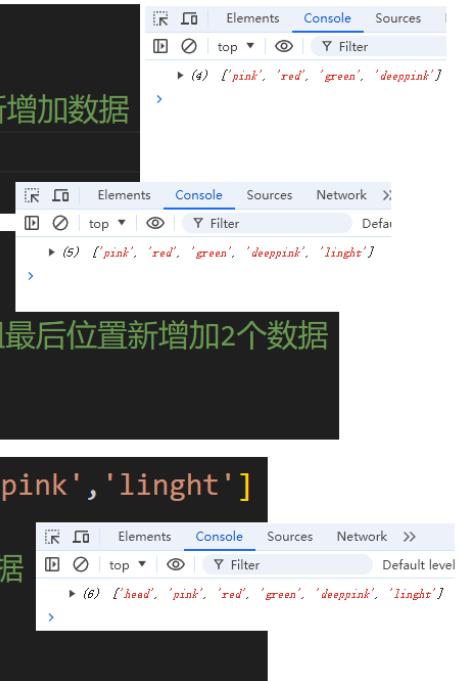
```

数组.push(新加内容)//在数组最后位置新增加数据

数组.unshift(新加内容)//在数组最前面位置增加数据

```
let arr = ['pink', 'red', 'green']
arr.push('deeppink') //在数组最后位置新增加数据
console.log(arr)
let arr = ['pink', 'red', 'green']
arr.push('deeppink', 'linght') //在数组最后位置新增加2个数据
console.log(arr)

let arr = ['pink', 'red', 'green', 'deeppink', 'linght']
arr.unshift('head') //在数组开头追加数据
console.log(arr)
```



## 数组筛选案例

### 数组筛选

需求：将数组 [2, 0, 6, 1, 77, 0, 52, 0, 25, 7] 中大于等于 10 的元素选出来，放入新数组

```
let arr = [2, 0, 6, 1, 77, 9, 54, 3, 78, 7]
let NewArr = [] // 定义新的空数组

for(let i = 0; i < arr.length; i++ )
{
    if(arr[i] >= 10)
    {
        NewArr.push(arr[i]) // 把 >=10 的数据给新数组
    }
}

console.log(NewArr)
```



数组.pop()//删除数组中最后一个元素

数组.shift()//删除数组中最开头元素

数组.splice(起始位置, 删除第几个元素)//删除数组中指定的元素

```
let arr = ['数组一', '数组二', '数组三']
console.log(arr)

arr.pop() //删除数组最后一个元素
console.log(arr)
```

```
let arr = ['数组一', '数组二', '数组三']
console.log(arr)

arr.shift() //删除数组最开头元素
console.log(arr)
```

```
let arr = ['数组一', '数组二', '数组三', '数组四', '数组五']
console.log(arr)

arr.splice(1,1) //从数组第1个元素开始删, 删1个元素
console.log(arr)
```

↑  
从下标1开始删  
刪除2个

```
let arr = ['数组一', '数组二', '数组三', '数组四', '数组五']
console.log(arr)

arr.splice(1,2) //从数组第1个元素开始删, 删2个元素
console.log(arr)
```

```
let arr = ['数组一', '数组二', '数组三', '数组四', '数组五']
console.log(arr)

arr.splice(2,2) //从数组第2个元素开始删, 删2个元素
console.log(arr)
```

数组和 for 循环实现柱状图案例



```

<body>
  <div class="box">
    <div style="height: 123px;">
      <span>123</span>
      <h4>第1季度</h4>
    </div>

    <div style="height: 156px;">
      <span>156</span>
      <h4>第2季度</h4>
    </div>

    <div style="height: 120px;">
      <span>120</span>
      <h4>第3季度</h4>
    </div>

    <div style="height: 210px;">
      <span>210</span>
      <h4>第4季度</h4>
    </div>
  </div>

```

123

第1季度

156

第2季度

120

第3季度

210

第4季度

树型排列效果

样式中设置box盒子下所有div的背景色和布局

```

<style>
  .box>div{
    display: flex;
    width: 50px;
    background-color: pink;
    flex-direction: column;
    justify-content: space-between;
  }
</style>

```

.box div span{ margin-top: -20px; }

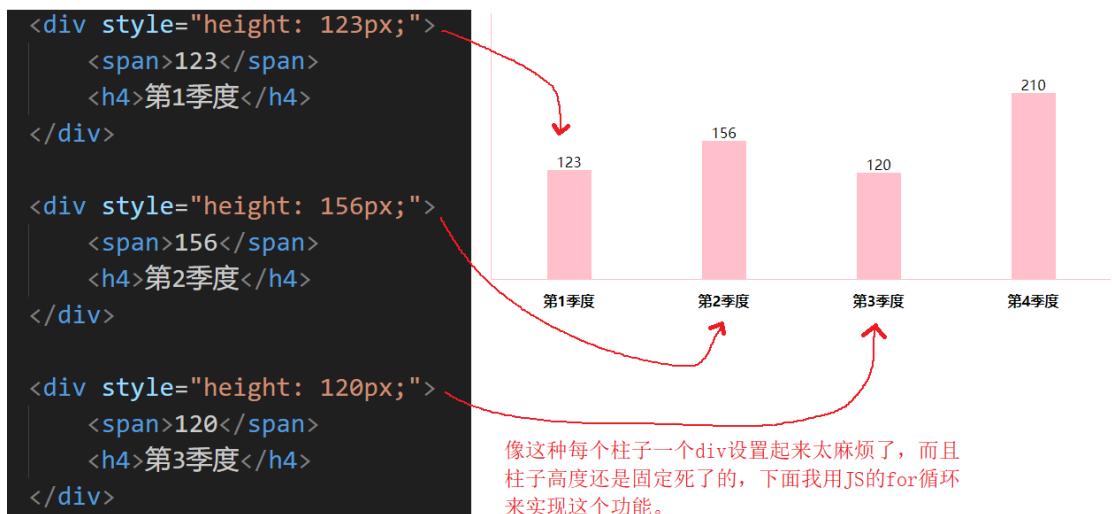
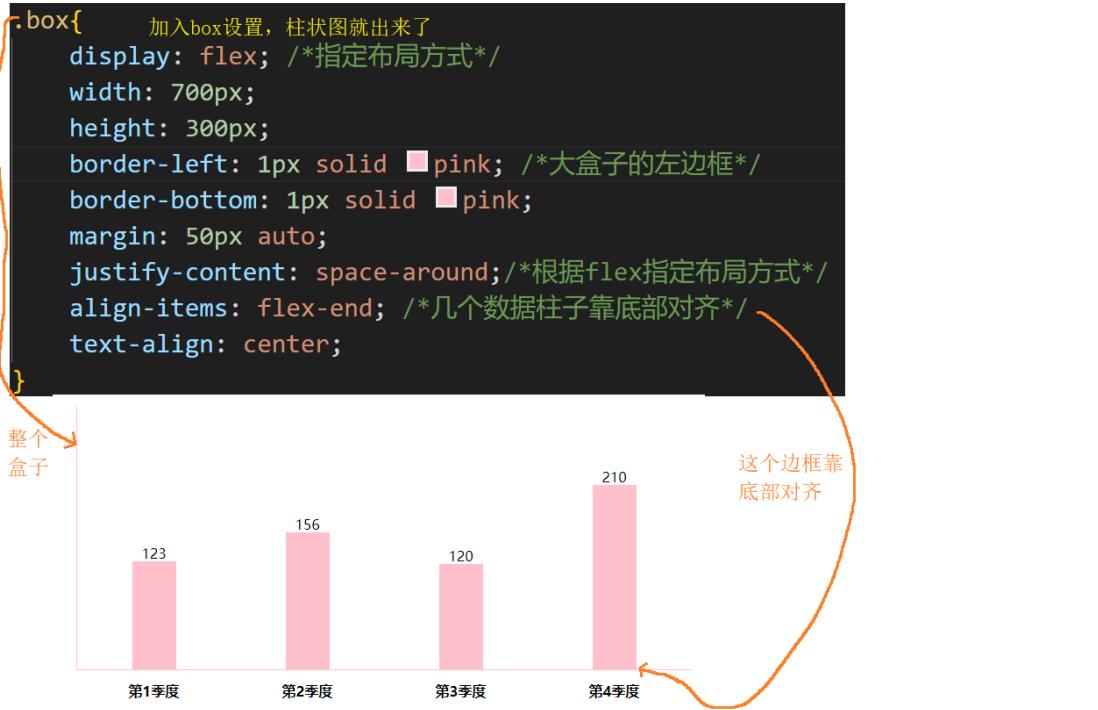
设置box盒子里面div里面的span标签向上移动20个像素

```

.box div h4{
  margin-bottom: -35px;
  width: 70px;
  margin-left: -10px;
}

```

h4标签边距调整



1. 在html文件中把css写好

```

.box{
    display: flex; /*指定布局方式*/
    width: 700px;
    height: 300px;
    border-left: 1px solid pink; /*大盒子的左边框*/
    border-bottom: 1px solid pink;
    margin: 50px auto;
    justify-content: space-around; /*根据flex指定布局方式*/
    align-items: flex-end; /*几个数据柱子靠底部对齐*/
    text-align: center;
}

```

2. 在html文件中把js写好

```

<!--在html引入外部js写法--> <!--在html引入外部js写法--> 2. 之间调用js文件
<Script src="_/my.js"></Script> <!--符号./就是引入当前目录下js文件-->

```

3. 在js文件中输出html标签

```

document.write(`<div class="box">`); //直接向页面输出html

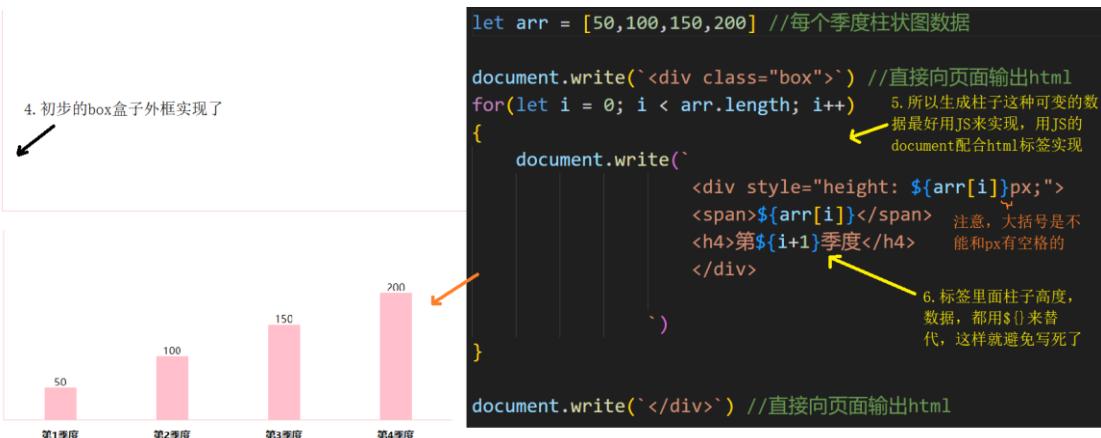
document.write(`</div>`); //直接向页面输出html

```

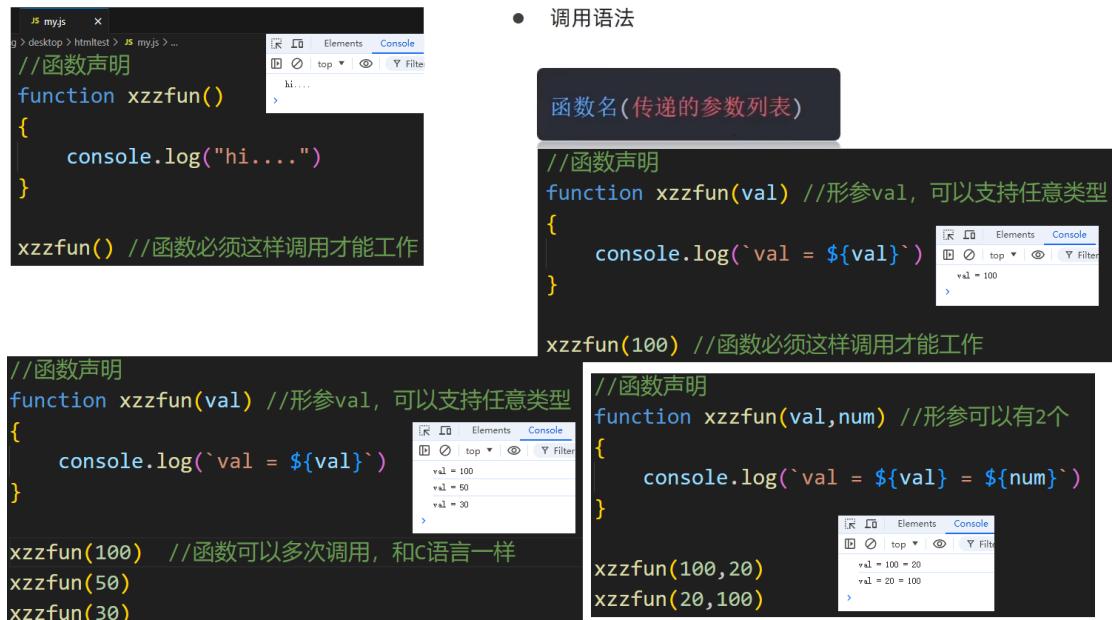
```

.box>div{
    display: flex;
    width: 50px;
    background-color: pink;
    flex-direction: column;
    justify-content: space-between;
}
.box div span{
    margin-top: -20px;
}
.box div h4{
    margin-bottom: -35px;
    width: 70px;
    margin-left: -10px;
}

```



## 函数的使用



但是如果做用户不输入实参，刚才的案例，则出现 undefined + undefined 结果是什么？

> NaN

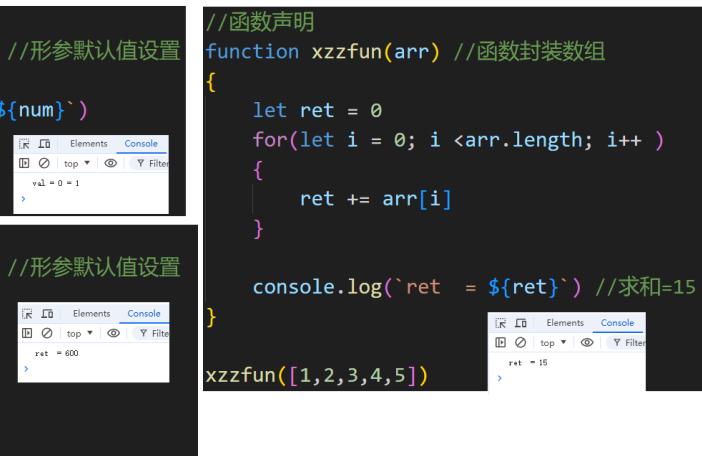
我们可以改进下，用户不输入实参，可以给 **形参默认值**，可以默认为 0，这样程序更严谨，可以如下操作：

```
//函数声明
function xzzfun(val = 0,num = 1) //形参默认值设置
{
    console.log(`val = ${val} = ${num}`)
}

xzzfun()

//函数声明
function xzzfun(val = 0,num = 1) //形参默认值设置
{
    let ret = val + num
    console.log(`ret = ${ret}`)
}

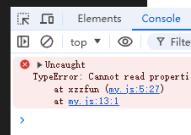
xzzfun(100,500)
```



```
//函数声明 如果函数形参不传入数组，就会运行报错
function xzzfun(arr) //函数封装数组
{
    let ret = 0
    for(let i = 0; i < arr.length; i++)
    {
        ret += arr[i]
    }

    console.log(`ret = ${ret}`) //求和=15
}

xzzfun() //但是如果函数不传入数组呢?
```



```
//函数声明
function xzzfun(arr = []) //如果把函数形参设成数组形式，设置死数据类型
{
    let ret = 0
    for(let i = 0; i < arr.length; i++)
    {
        ret += arr[i]
    }

    console.log(`ret = ${ret}`) //求和=15
}
```



程序运行不报错

```
//函数声明
function xzzfun(val) //函数返回值测试
{
    let ret = 0
    ret = val + 100
    return ret
}

let data = xzzfun()
console.log(data)
```

A screenshot of a browser developer tools console window titled "Console". It shows the output "300".

```
//函数声明
function xzzfun(val = []) //函数返回数组
{
    let first = 0;
    let end = 0;
    first = val[2]
    end = val[3]
    return [first,end] //返回数组方式[]
}

let arr = []
arr = xzzfun([1,2,3,4,5,6]) //得到返回的3和4
console.log(arr)
```

```
(2) [3, 4]
0: 3
1: 4
length: 2
[[Prototype]: Array(0)]
```

```
let xgla = 100 //这是全局变量
函数变量作用域测试

function xzzfun()
```

```
{
    let str = 'pink' //这就是局部变量
    //局部变量只能在函数内使用
    console.log(str)
}
```

```
Uncaught ReferenceError: str is not defined
at my_js_10-13
```

外部使用局部变量失败

```
console.log(str)

let xgla = 100 //这是全局变量

function xzzfun()
{
    let str = 'pink' //这就是局部变量
    //局部变量只能在函数内使用
    console.log(str)
}

xzzfun()
console.log(xgla) //使用全局变量成功
```

```
pink
100
```

```
let xgla = 100

function xzzfun()
{
    let xgla = 50 //局部变量和全局变量虽然名字相同，但是不冲突
    console.log(xgla)
}
```

```
xzzfun()
console.log(xgla) //使用全局变量成功
```

建议少这么用，还是各取变量名最好

```
let xgla = 100

function xzzfun()
{
    let xgla = 50 //局部变量和全局变量虽然名字相同，但是不冲突
    console.log(xgla)
    function fun2()
    {
        let xgla = 80 //函数套函数，变量作用域也不冲突
        console.log(xgla)
    }
    fun2()
}

xzzfun()
console.log(xgla) //使用全局变量成功
```

```
50
80
100
```

函数套函数定义方法，JS独有，C语言这样可不行。

## 匿名函数实现

### 匿名函数

没有名字的函数，无法直接使用。

使用方式：

➤ 函数表达式

➤ 立即执行函数

```
//这就是没有名字的函数，用变量赋值的表达式方式实现
let fn = function()
{
    console.log('匿名函数表达式')
}
console.log(fn) 使用变量相当于使用函数
```

```
//这就是没有名字的函数，用变量赋值的表达式方式实现
```

```
let fn = function()
{
    console.log('匿名函数表达式')
}
```

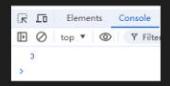
```
fn() //直接调用变量+()也可以
```



```
//这就是没有名字的函数，用变量赋值的表达式方式实现
```

```
let fn = function(x,y)
{
    console.log(x+y)
}
```

```
fn(1,2)
```



```
let btn = document.querySelector('button')
btn.onclick = function() {
    alert('我是匿名函数')
}
```

1. 匿名函数的好处就是，  
比如我获取一个按钮

2. 当点击按钮的时候，  
这个匿名函数运行

这种就不用单独定义函数去写按钮触发内容了，直接在按钮触发时候用匿名函数实现

```
let btn = document.querySelector('button')
btn.addEventListener(function() {
    alert('弹出')
}) 匿名函数也可以放在某个函数的括号中使用
```

## 2. 立即执行函数

场景介绍：避免全局变量之间的污染

```
let num = 10
let num = 20
```

如何让两个名字一样的  
全局变量不相互影响？

```
(function(){}()) //这就是立即执行函数
```

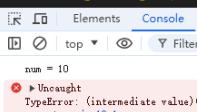
```
(function(){←立即执行函数里面必须
    写内容，不然会报错
    console.log('立即执行函数')
})()
```

```
(function(){
    let num = 10
    console.log(`num = ${num}`)
})()
```

```
(function(){
    let num = 20
    console.log(`num = ${num}`)
})()
```

如果是两个立即执行函  
数可以同时运行吗

```
num = 10
```



运行报错，其实不是函数 →  
问题，是两个立即执行函  
数需要用 ; 分割

```
(function(){
    let num = 10
    console.log(`num = ${num}`)
})(); 分号分隔，执行正常
(function(){
    let num = 20
    console.log(`num = ${num}`)
})()
```

```
(function(){})(这是匿名函数形参)
```

```
(function(x,y){  
    console.log(x,y)  
})(1,2)←形参传入
```

```
(function(){}()) //这也是立即执行函数，只是把小括号都放在一起了
```

```
(function(x,y){  
    console.log(x,y)  
}(1,2))
```

立即执行函数应用场景

```
(function flexible(window, document) {  
    var docEl = document.documentElement  
    var dpr = window.devicePixelRatio || 1  
    // adjust body font size  
    function setBodyFontSize() {  
        if (document.body) {  
            document.body.style.fontSize = (12 * dpr) + 'px'  
        }  
        else {  
            document.addEventListener('DOMContentLoaded', setBodyFontSize)  
        }  
    }  
    setBodyFontSize();
```

很多时候，写功能函数的时候都喜欢用立即函数将其包裹起来

这样函数里面的变量就不会和其它功能函数冲突

而且这个是第3方写的JS库，很可能里面的变量和自己当前文件变量名冲突。

整数=parselInt (传入字符串)//解析一个字符串并返回整数

转换时间案例 函数封装实验

计算公式：计算时分秒

小时： h = parseInt(总秒数 / 60 / 60 % 24)

分钟： m = parseInt(总秒数 / 60 % 60 )

秒数： s = parseInt(总秒数 % 60)

let second = 1000 如果传入1000

```
let second = 1000  
function getTime(t)  
{  
    let h = parseInt(t / 60 / 60 % 24)  
    let m = parseInt(t / 60 % 60)  
    let s = parseInt(t % 60)  
    h = h < 10 ? '0'+ h : h  
    m = m < 10 ? '0'+ m : m  
    s = s < 10 ? '0'+ s : s  
    return `转换完毕之后是${h}小时${m}分${s}秒`  
}  
  
let str = getTime(second)  
console.log(str)
```

## 逻辑中断

符号	短路条件
&&	左边为false就短路
	左边为true就短路

之前我们知道如果函数有形参，但是函数使用时不传入参数就是NaN

```
function fn(x,y){    console.log(x+y)}fn()
```

x没有值就是假  
x = x || 0  
y = y || 0  
console.log(x+y)

使用&&或者||实现逻辑中断

```
console.log(false && 3+5) //逻辑中断
```

```
console.log(true && 3+5) //逻辑中断
```

```
console.log(false || 3+5) //逻辑或是反过来的, false 执行右边
```

```
function fn(x,y){    x = 1 || 0 //x=1    y = 2 || 0 //y = 2    console.log(x+y)}fn()
```

### 逻辑中断案例

```
(function flexible(window, document) {    var dpr = window.devicePixelRatio || 1}(window, document))
```

如果这个为1，就返回1给dpr，如果为2就返回2给dpr

```
console.log(false && 20) // false
console.log(5 < 3 && 20) // false
console.log(undefined && 20) // undefined
console.log(null && 20) // null
console.log(0 && 20) // 0
console.log(10 && 20) // 20
```

```
console.log(false || 20) // 20
console.log(5 < 3 || 20) // 20
console.log(undefined || 20) // 20
console.log(null || 20) // 20
console.log(0 || 20) // 20 0为假，所以
console.log(10 || 20) // 10 返回20
```

# JS 对象基本使用

对象有什么好处？

```
// 这3个180是啥意思？ 你猜~  
let arr = [180, 180, 180]
```

1. 保存网站用户信息，比如姓名，年龄，电话号码... 用以前学的数据类型方便吗？

➤ 不方便，很难区分

2. 我们是不是需要学习一种新的数据类型，可以详细的描述某个事物？

- 姓名 ↗ 就是说一看到这种新定义的
- 年龄 ↗ 数据类型，就知道里面有多
- 电话 ↗ 少个变量内容

```
let obj = {  
    uname: 'pink老师',  
    age: 18,  
    gender: '女'  
} 这就算对象写法
```

## ● 1. 对象声明语法

```
let 对象名 = {}
```

```
let 对象名 = {  
    属性名: 属性值,  
    方法名: 函数  
}
```

```
let 对象名 = new Object()
```

```
let pink = {  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
console.log(pink)  
console.log(typeof pink) //检测数据类型
```

```
let pink = {  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
pink.uname = '新名字' //修改对象指定变量内容  
pink.age = 500  
console.log(pink.uname) //获取对象名字  
console.log(pink.age) //获取对象年龄
```

```
let pink = {  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
console.log(pink.uname) //获取对象名字  
console.log(pink.age) //获取对象年龄
```

```
let pink = {  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
pink.xadd = '直接新增内容'  
//其它语言都需要专门函数来给对象增加变量，但是JS可以直接增加变量，够任性  
console.log(pink)
```

**delete 对象.成员** //删除对象里面的某个成员变量

```
let pink = {  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
delete pink.age //删除对象里面指定的变量  
  
console.log(pink)
```

对象里面实现函数

```
let pink = {  
    uname : 'pink名字',  
  
    //对对象里面实现函数使用匿名函数，song是自己取的函数别名  
    song:function()  
    {  
        console.log('对象内函数使用')  
    }  
}  
  
pink.song() //这就是调用对象里面的函数
```

对象里面定义变量方式2

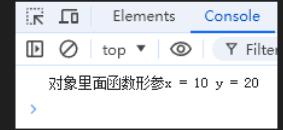
```
let pink = {  
    'goods-name': '变量的另外一种定义法', //另一种在对象里面定义变量的方法  
    uname : 'pink名字',  
    age : 18,  
    gender : '男'  
}  
  
console.log(pink['goods-name']) //使用（对象名['属性名']）这种方式获取数据
```

```

let pink = {
    uname : 'pink名字',
    //对象里面实现函数形参
    song:function(x,y)
    {
        console.log(`对象里面函数形参x = ${x} y = ${y}`)
    }
}

pink.song(10,20) //这就是调用对象里面的函数

```



```

let pink = {
    uname : 'pink名字',
    //对象里面实现函数形参
    song:function(x,y)
    {
        console.log(`对象里面函数形参x = ${x} y = ${y}`)
    },
    dance:function(){}
} //为什么返回undefined, 这是因为函数没有返回值
console.log( pink.song(10,20) ) //我这样调用

```



遍历对象，将对象里面每个变量都获取出来

for in 做循环来遍历对象

for(变量 in 变量)

声明一个变量，in 在数组变量里面去遍历

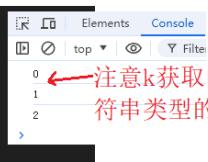
我们来说是用for遍历数组的情况

```

let arr = ['pink','red','blue']

for(let k in arr)
{
    console.log(k) //k 是获取数组下标
}

```

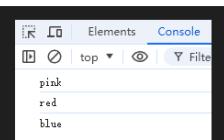


```

let arr = ['pink','red','blue']

for(let k in arr)
{
    console.log(arr[k]) //k虽然是字符串, 也可以做数组下标
} //所以我们不推荐用for in 来遍历数组, for in适合遍历对象

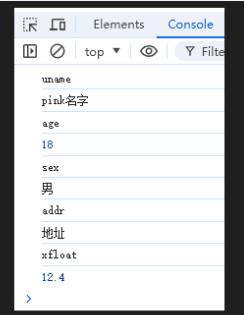
```



```

let pink = {
    uname : 'pink名字',
    age : 18,
    sex : '男',
    addr : '地址',
    xfloat : 12.4
}

```



```

//因为对象是无序的，所以之前用for(let i = 0; i < pink.length; i++)无效
for(let k in pink)
{
    console.log(k) //获取变量名(属性名)
    console.log(pink[k]) //获取变量值(属性值)
}

```

为什么对象要用 `for in`？这是因为数组有长度，但是对象是无长度的，无长度用(`for i<???`)终止条件是什么呢？对象没有长度，就没有终止条件，所以用 `for in` 最合适。

## 渲染信息表案例

```

// 定义一个存储了若干学生信息的数组
let students = [
    {name: '小明', age: 18, gender: '男', hometown: '河北省'},
    {name: '小红', age: 19, gender: '女', hometown: '河南省'},
    {name: '小刚', age: 17, gender: '男', hometown: '山西省'},
    {name: '小丽', age: 18, gender: '女', hometown: '山东省'}
]

```

需求：请把下面数据中的对象打印出来：

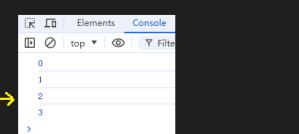
```

let students = [
    {name : '小明',age : 18, gender : '男', hometown: '河北省'}, //数组里面加一条对象
    {name : '小红',age : 19, gender : '女', hometown: '河南省'}, //数组里面加一条对象
    {name : '小刚',age : 17, gender : '男', hometown: '山西省'}, //数组里面加一条对象
    {name : '小丽',age : 20, gender : '女', hometown: '山东省'} //数组里面加一条对象
]

for(let i = 0; i < students.length; i++)
{
    console.log(i) //数组下标
}

```

这次只是打印  
了数组下标 →



```

let students = [
    {name : '小明',age : 18, gender : '男', hometown: '河北省'}, //数组里面加一条对象
    {name : '小红',age : 19, gender : '女', hometown: '河南省'}, //数组里面加一条对象
    {name : '小刚',age : 17, gender : '男', hometown: '山西省'}, //数组里面加一条对象
    {name : '小丽',age : 20, gender : '女', hometown: '山东省'} //数组里面加一条对象
]

for(let i = 0; i < students.length; i++)
{
    console.log(i) //数组下标
    console.log(students[i]) //这样就是取每条整个对象
}

```



```

let students = [
    {name : '小明',age : 18, gender : '男', hometown: '河北省'}, //数组里面加一条对象
    {name : '小红',age : 19, gender : '女', hometown: '河南省'}, //数组里面加一条对象
    {name : '小刚',age : 17, gender : '男', hometown: '山西省'}, //数组里面加一条对象
    {name : '小丽',age : 20, gender : '女', hometown: '山东省'} //数组里面加一条对象
]

for(let i = 0; i< students.length; i++)
{
    console.log(students[i].name) //取每条对象里面的指定属性值
    console.log(students[i].hometown) //取每条对象里面的指定属性值
}

```

但是这种for(i...)取法，是因为数组对象长度都是写死了的

学生列表

序号	姓名	年龄	性别	家乡
1	小明	18	男	河北省
2	小红	19	女	河南省
3	小刚	17	男	山西省
4	小丽	18	女	山东省

需求：根据以上数据渲染生成表格

```

let students = [
    {name : '小明',age : 18, gender : '男', hometown: '河北省'}, //数组里面加一条对象
    {name : '小红',age : 19, gender : '女', hometown: '河南省'}, //数组里面加一条对象
    {name : '小刚',age : 17, gender : '男', hometown: '山西省'}, //数组里面加一条对象
    {name : '小丽',age : 20, gender : '女', hometown: '山东省'} //数组里面加一条对象
]

for(let i = 0; i< students.length; i++)
{
    document.write(`
        <tr>
            <td>${i+1}</td>
            <td>${students[i].name}</td>
            <td>${students[i].age}</td>
            <td>${students[i].gender}</td>
            <td>${students[i].hometown}</td>
        </tr>
    `)
}

```

感觉排行对了，怎么没变成表格  
呢？需要加CSS↓

1 小明 18 男 河北省 2 小红 19 女 河南省 3 小刚 17 男 山西省 4 小丽 20 女 山东省

` //老方法，在document里面实现html表格标签，然后将对象填入，自动生成表格

```

<!--加入css样式-->
<style>
    table{
        width: 600px;
        text-align: center;
    }
    table,
    th,
    td{
        border: 1px solid #ccc;
        border-collapse: collapse;
    }
    caption{
        font-size: 18px;
        margin-bottom: 10px;
        font-weight: 700;
    }
    tr{
        height: 40px;
        cursor: pointer;
    }
    table tr:nth-child(1){ /*第1行背景色变色*/
        background-color: #ddd;
    }
    /*除了第1行，鼠标移动到其它行有底色*/
    table tr:not(:first-child):hover{
        background-color: #eee;
    }
</style>

```

```

document.write(`<table>` //输出表格标签 ← 表格标签不要忘了
for(let i = 0; i < students.length; i++) //这里循环按照表格标签排列
{
    document.write(`
        <tr>
            <td>${i+1}</td>
            <td>${students[i].name}</td>
            <td>${students[i].age}</td>
            <td>${students[i].gender}</td>
            <td>${students[i].hometown}</td>
        </tr>
    `) //老方法，在document里面实现html表格标签，然后将对象填入，自动生成表格
}

document.write(`</table>` //表格标签

```

1	小明	18	男	河北省
2	小红	19	女	河南省
3	小刚	17	男	山西省
4	小丽	20	女	山东省

## JS 内置对象

**Math.PI** //得到 3.14

**Math.random()** //返回 0~1 随机数

**Math.abs**//取绝对值

**Math.max**//取最大值

思考：我们之前用过内置对象吗？

- `document.write()`
- `console.log()` ↗ 这些就是内置对象(对象. 函数)

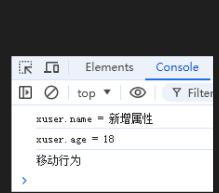
```

console.log(Math.PI) //获取圆周率
console.log(Math.ceil(1.1)) //向上取整, 1.1 =2
console.log(Math.ceil(1.5)) //向上取整, 1.5 =2
console.log(Math.floor(1.1)) //向下取整, 1.1 =1
console.log(Math.floor(1.5)) //向下取整, 1.5 =1
console.log(Math.round(1.2)) //四舍五入 = 1
console.log(Math.round(1.9)) //四舍五入 = 2
console.log(Math.round(-1.1)) //四舍五入 = -1
console.log(Math.round(-1.5)) //四舍五入 = -1
console.log(Math.max(1,2,3,4,5)) //取最大值
console.log(Math.min(1,2,3,4,5)) //取最小值
console.log(Math.abs(-1)) //取绝对值
console.log(Math.random()) //取0~1之间随机数(包含0, 不包括1)

```

## 对象动态添加函数和属性

```
let xuser = {} //声明一个空对象  
  
xuser.name = '新增属性'  
xuser.age = 18 //新增数字  
  
xuser.move = function() //给对象新增匿名函数  
{  
    console.log(`xuser.name = ${xuser.name}`)  
    console.log(`xuser.age = ${xuser.age}`)  
    console.log('移动行为')  
}  
  
xuser.move()
```



- 如何生成0-10的随机数呢？

```
Math.floor(Math.random() * (10 + 1))  
  
let arr = ['red', 'green', 'blue', 'yellow'] //定义一个数组，随机输出某个元素  
let random = Math.floor(Math.random() * arr.length) //随机小数 * 长度 = 从0~1到0~10  
console.log(arr[random]) //random得到0~10随意整数
```



- 如何生成5-10的随机数？

```
Math.floor(Math.random() * (5 + 1)) + 5
```

- 如何生成N-M之间的随机数

```
Math.floor(Math.random() * (M - N + 1)) + N
```

如生成5~10随机数，M取5, N取10

## Web API 使用

### 变量声明

- 变量声明有三个 var let 和 const
- 我们应该用那个呢？
- 首先var 先排除，老派写法，问题很多，可以淘汰掉...
- let or const？
- 建议：const 优先，尽量使用const，原因是：

在不改变变量的应用中，就用const

- const语义化更好
- 很多变量我们声明的时候就知道他不会被更改了，那为什么不用const呢？
- 实际开发中也是，比如react框架，基本const

- 请问以下的可不可以把let 改为 const？

```
document.write('我叫' + '刘德华') // 我叫刘德华  
let uname = '刘德华' // 是否可以用const ?  
let song = '忘情水' // 因为变量只用一次，可以改成const  
document.write(uname + song) // 刘德华忘情水
```

```
let num1 = +prompt('请输入第一个数值：')  
let num2 = +prompt('请输入第二个数值：')  
alert('两者相加的结果是: ${num1 + num2}')
```

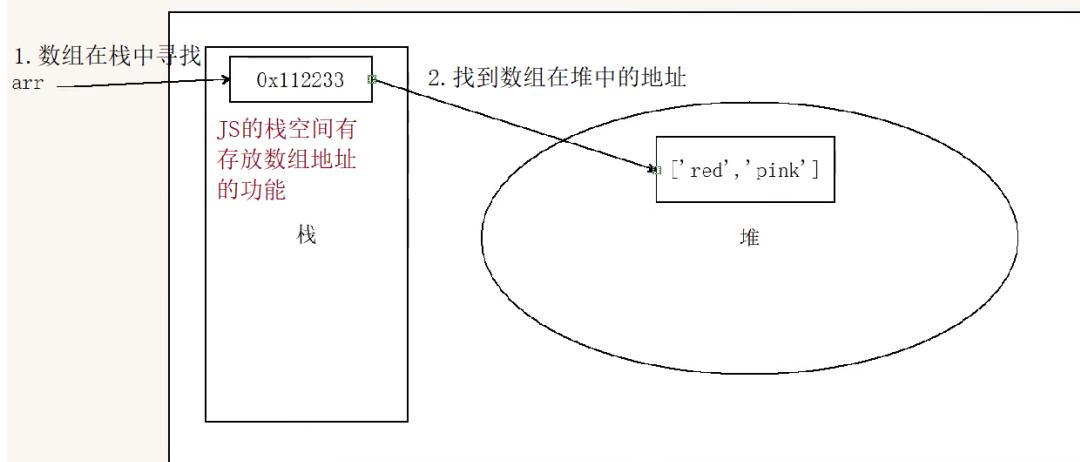
变量虽然加减，但是变量本身值没变，可以用const

- 请问以下的可不可以把let 改为 const？

```
<script>      let不能改成const  
    let num = 1  
    num = num + 1  
    console.log(num) // 结果是 2  
</script>  
</div>
```

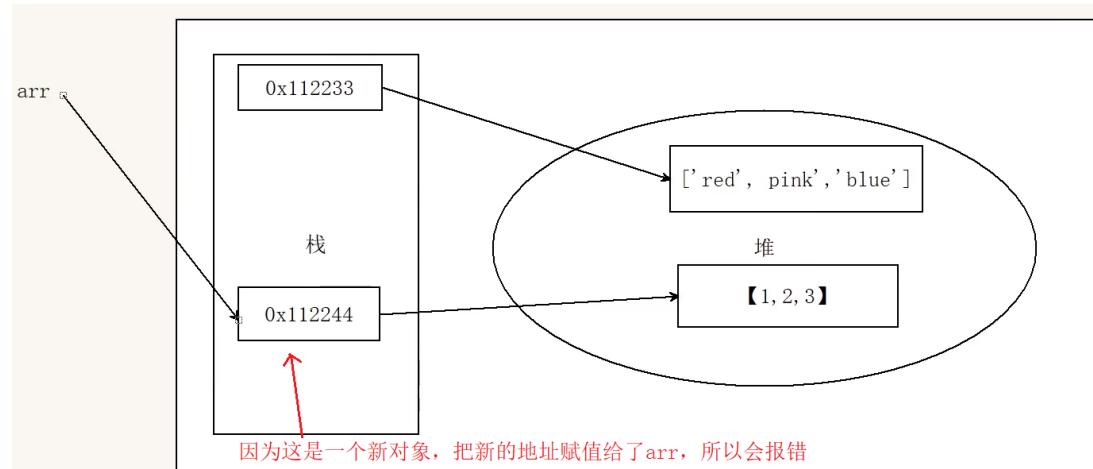
```
for (let i = 0; i < nums.length; i++) {  
    document.write(nums[i])  
}      let不能改成const
```

```
const arr = ['red', 'pink'] //数组放在堆空间 ← 为什么const 定义的arr常量可以用 push添加元素? 不是常量不能修改吗。
```



```
const arr = ['red', 'pink'] //数组放在堆空间
arr = [1, 2, 3] //如果给对象新增加数组地址
arr.push('blue') //数组添加的元素也是放在堆里面
```

为什么报错?



所以const定义的对象, 只是添加删除对象里面原有的元素, 是不会改变对象的。所以push没有问题。

```
const names = [] 报错
names = [1, 2, 3]
```

```
const names = []
names[0] = 1 改成这样不
names[1] = 2 会报错
names[2] = 3
```

```
const obj = {} 报错
obj = {
  uname: 'pink老师'
}
```

```
const obj = {} 改成这样不
obj.uname = 'pink老师'
```

## DOM 树和 DOM 对象

对象=document.querySelector(写入标签) //填入 html 标签，将标签返回给 JS 对象

console.dir(对象) //console.dir 是专门，打印对象里面内容的

DOM示例演示  
通过 JavaScript 动态更新 HTML 文档中的标签元素。  
随机点名  
张飞  
开始 结束

通过 JS 去改变标签里面的内容就算 DOM

在html里面写一个标签  
!--在html引入外部JS写法-->  
<Script src="./my.js"></Script>

const div = document.querySelector('div') //获取html指定标签的内容转成对象  
console.dir(div) //console.dir是专门，打印对象的  
console.log(typeof div) //查看数据类型

获取到对象了

样式属性  
div里面有无数的属性  
里面的属性都可以通过JS去修改从而更改html显示

<div class="box">JS的DOM测试1111</div>  
<div class="box">JS的DOM测试22222</div>

const box = document.querySelector('div') //获取html指定标签的内容转成对象  
console.dir(box) 这时候JS去获取div，它会选择第1个div还是第2个div

其实JS只会选择第1个div

<div class="box">JS的DOM测试1111</div>  
<div class="box">JS的DOM测试22222</div>  
<p id="xnav"> id选择器测试</p>

在html端加入id选择器

const box = document.querySelector('#xnav')  
console.dir(box) ID选择器必须用#获取

找到ul标签，获取第1个li内容

我想获取ul标签里第1个li标签内容

40 / 120

## 根据CSS选择器来获取DOM元素 (重点)

1. 选择匹配的第一个元素

语法:

```
document.querySelector('css选择器')
```

参数:

包含一个或多个有效的CSS选择器 **字符串**

返回值:

CSS选择器匹配的**第一个元素**,一个 **HTMLElement**对象。

返回的是对象

如果没有匹配到, 则返回null。

```
const lis = document.querySelector('ul li') //获取li标签下所有内容(也就是每个li标签)  
console.log(lis)
```

2. 选择匹配的多个元素

语法:

```
document.querySelectorAll('css选择器')
```

参数:

包含一个或多个有效的CSS选择器 **字符串**

返回值:

CSS选择器匹配的**NodeList** 对象集合



```
<p id="xnav"> p标签测试 </p>
```

```
const lis = document.querySelector('#xnav')  
lis.style.color = 'red' //修改p标签颜色
```

```
<ul>  
  <li>测试1</li>  
  <li>测试2</li>  
  <li>测试3</li>  
</ul>
```

3个li标签

```
const lis = document.querySelectorAll('ul li')  
  
for(let i = 0; i < lis.length; i++)  
{  
  console.log(lis[i]) //寻找每一个li标签对象  
}
```

每一个li标签对象用遍历方式获取

- 测试1
- 测试2
- 测试3

```
const lis = document.querySelectorAll('ul li')
```

```
lis[0].style.color = 'red' //直接使用伪数组下标获取法, 修改每个标签颜色  
lis[1].style.color = 'blue'  
lis[2].style.color = 'green'
```

- 测试1
- 测试2
- 测试3

1. 获取页面中的标签我们最终常用那两种方式?

- querySelectorAll()
- querySelector()

2. 他们两者的区别是什么?

- querySelector() 只能选择一个元素, 可以直接操作(就算有多个相同的标签, 它也只获取第1个标签)
- querySelectorAll() 可以选择多个元素, 得到的是伪数组, 需要遍历得到每一个元素

3. 他们两者小括号里面的参数有神马注意事项?

- 里面写css选择器
- 必须是字符串, 也就是必须加引号

对象.innerText //获取指定标签内容，修改指定标签内容，只解析修改文本内容

```
<div class="box">JS替换html标签内容测试</div>
const box = document.querySelector('.box') //获取标签box类对象
console.log(box.innerText) //提取指定标签里面的文本内容
JS替换html标签内容测试 提取标签内容
> 标签内容提取成功

<div class="box">JS替换html标签内容测试</div> ← 修改前的html标签
JS替换html标签内容测试

const box = document.querySelector('.box') //获取标签box类对象
box.innerText = '修改后的标签内容' //直接修改指定html标签内容
修改后的标签内容
↓
const box = document.querySelector('.box') //获取标签box类对象
box.innerText = '<strong>标签内新标签加入</strong>' //直接在标签中加入新标签可以吗?
<strong>标签内新标签加入</strong> 我们发现innerText居然只能修改文本，strong标签居然没效果。
```

对象.innerHTML //获取指定标签内容，修改指定标签内容，可以解析标签

```
<div class="box">JS替换html标签内容测试</div>
const box = document.querySelector('.box') //获取标签box类对象
console.log(box.innerHTML) //获取标签内容和innerText一样
JS替换html标签内容测试
↓
const box = document.querySelector('.box') //获取标签box类对象
box.innerHTML='<strong>我要更换</strong>' → 我要更换
```

#### 抽奖案例

- ①：声明数组: const personArr = ['周杰伦', '刘德华', '周星驰', 'Pink老师', '张学友']
- ②：一等奖:随机生成一个数字 (0~数组长度)，找到对应数组的名字
- ③：通过innerText 或者 innerHTML 将名字写入span元素内部
- ④：二等奖依次类推

```
<div class="wrapper">
  <strong> 抽奖比赛 </strong>
  <h1>一等奖: <span id="one">???</span> </h1>
  <h3>二等奖: <span id="two">???</span> </h3>
  <h5>三等奖: <span id="three">???</span> </h5>
</div>
const personArr = ['奖品111', '奖品222', '奖品333', '奖品444', '奖品555']
const random = Math.floor(Math.random() * personArr.length) //获取随机数，转成数组下标
console.log(personArr[random]) //随机得到一个名字
const one = document.querySelector('#one') //一等奖获取数组的名字赋值给一等奖标签
one.innerHTML = personArr[random] //数组值赋值给一等奖标签
personArr.splice(random,1) //获取之后，删除这个获取的奖品
console.log(personArr) //剩下的就是没获取的奖品
```

1. 我们使用每次刷新页面的方式来抽奖

2. 现在测试一等奖

抽奖比赛 3. 一等奖抽取555之后

一等奖: ???

二等奖: ???

三等奖: ???

4. 数组里面的奖品555就删除了

```
奖品555
> (4) ["奖品111", "奖品222", "奖品333", "奖品444"]
```

```

const personArr = ['奖品111', '奖品222', '奖品333', '奖品444', '奖品555']
const random = Math.floor(Math.random() * personArr.length) //获取随机数，转成数组下标
console.log(personArr[random]) //随机得到一个名字
const one = document.querySelector('#one') //一等奖获取数组的名字赋值给一等奖标签
one.innerHTML = personArr[random] //数组值赋值给一等奖标签
personArr.splice(random, 1) //获取之后，删除这个获取的奖品
console.log(personArr) //剩下的就是没获取的奖品

//二等奖操作，因为数组少了一个，所以重新计算长度
const random2 = Math.floor(Math.random() * personArr.length)
const two = document.querySelector('#two') //二等奖获取
two.innerHTML = personArr[random2]
personArr.splice(random2, 1)

//三等奖操作，和上面方式一样

```

抽奖比赛  
一等奖: 奖品333  
二等奖: 奖品111  
三等奖: ???

操作标签常用属性，修改标签样式属性，修改表单标签属性

### JS 修改 html 中 img 图片标签中的图片

The screenshot illustrates the steps to change an image source using JavaScript:

- Initial State:** An tag with `src="./129.jpg"` and `alt=""`. A note says "在标签中实现了图片加载" (The image is loaded directly from the tag).
- Image Preview:** A preview of the image shows a purple and black character.
- JavaScript Code:**

```

const img = document.querySelector('img') //直接获取img标签对象
img.src = './500.jpg' //用JS直接更换标签中的图片

```
- Result:** The image is now replaced by a yellow and orange character.
- Notes:**
  - "下面我要通过JS去修改标签的图片内容"
  - "现在JS将标签改成500.jpg图片"
  - "下面实现随机刷新图片案例"
- Implementation:**

```

//取N~M的随机整数
function getRandom(N,M){
    return Math.floor(Math.random() * (M - N + 1)) + N
}

const img = document.querySelector('img') //直接获取img标签对象
const random = getRandom(1,6) //取1~6随机整数
img.src = `./${random}.jpg` //将图片名改成序号，实现刷新页面随机生成图片

```

### JS 修改标签尺寸

The screenshot shows the following steps to change a div's dimensions:

- CSS Definition:** In the `<style>` block, there is a rule: `div { width: 200px; height: 200px; background-color: pink; }`.
- Default Dimensions:** A note says "2. div标签使用中默认尺寸就是200x200".
- JS Modification:**
  - `<div class="box"></div>`
  - `4. 先给标签加入类属性`
  - `const box = document.querySelector('.box')`
  - `box.style.width = '500px'`
  - `5. 用JS对象的.style去修改尺寸`
- Note:** "注意，JS的style...去修改尺寸，尺寸值是用字符串来写。"

```

<style>
  div {
    width: 200px;
    height: 200px;
    background-color: pink;
  }
</style>

```

```

const box = document.querySelector('.box')
box.style.width = '500px'
box.style.height = '500px'
box.style.backgroundColor = 'hotpink' //修改背景色

```

发现没有，JS中修改CSS的背景色属性，是用的驼峰命名法，把background-color的‘-’取消掉了，然后color用大写C来替代

```

const box = document.querySelector('.box')
box.style.width = '500px'
box.style.height = '500px'
box.style.backgroundColor = 'hotpink' //修改背景色
box.style.border = '2px solid black' //加入边框

```

```

box.style.borderTop = '2px solid blue' //上边框变蓝色

```

## JS 修改整个页面背景图片

```


<style>
  body{      给整个body页面标签增加背景图片，也就是整个页面加入背景图片
    background-image: url('./1.jpg') ;
  }
</style>
</head>
<body>
  
  <Script src="./my.js"></Script>  <!-- 符号 ./ 就是表示本地文件夹
</body>

```

```

<style>
  body{          no-repeat: 不平铺
    background: url('./1.jpg') no-repeat;   ↓
  }      直接使用background和background-
        image是一样的。
</style>

```

不平铺之后，页面背景只显示一张  
但是多余的页面尺寸是空着的。

```

<style>
  body{          top 水平居中
    background: url('./1.jpg') no-repeat top;   ↓
  }
</style>

```

```

<style>
  body{
    background: url('./1.jpg') no-repeat top center;
  }
</style>

```

```

document.body.style.backgroundImage = `url('./2.jpg)`

```

使用js去修改body页面的背景图，注意使用的属性是backgroundImage  
background-image在JS变成了backgroundImage



## 对象.className, //JS 使用类名的方式修改 CSS, 这样更方便

```
<style>
    .box{ 1. 之前修改CSS都是用
          style方法
          width: 200px;
          height: 200px;
          background-color: #pink;
    }
</style>
```

```
const box = document.querySelector('.box')
box.style.width = '300px'
box.style.backgroundColor = 'hotpink'
box.style.border = '2px solid blue'
box.style.borderTop = '2px solid red'
```

2. 如果我CSS里面修改的样式很多, 是不是只有每个样式都去写, 对象.style呢? 那是当然, 所以很麻烦。

```
<style> 3. 现在有个标签, 我
    div{   默认样式是这样。
          width: 200px; ↘
          height: 200px;
          background-color: #pink;
    }
</style>
</head>
<body> 4. 实现div盒子标签
      <div></div> ↘
```



5. 我现在想将div标签尺寸改大, 怎么可以方便的实现?

```
<style>
    div{  const div = document.querySelector('div') //获取div标签对象
          width: 200px;
          height: 200px;
          background-color: #pink;
    }
    .xbox{ /*自己取个类名*/
          width: 600px;
          height: 700px;
          background-color: #skyblue;
          margin: 100px auto;
          padding: 10px;
          border: 1px solid #000;
    }
</style>
</head>
<body> 6. 这里的xbox是我自
      定义的样式, 用整个
      样式修改div标签。
      <div></div>
```

```
div.className = 'xbox' //添加自定义的类名给div, 修改div标签的css样式
```

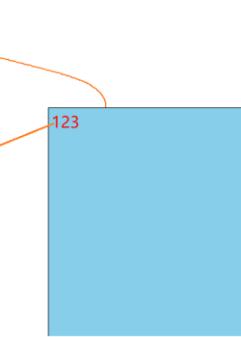


7. 直接用className就可以把自定义样式覆盖过来。

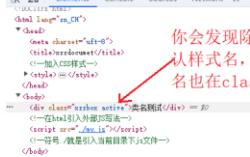
8. 这样的好处就是可以用CSS文本定义需要修改的样式, 样式代码自动补全也方便, 然后JS对象用类名去调用样式修改。

className是使用新值换旧值, 如果需要添加一个类, 需要保留之前的类名

```
<style>
    div{  const div = document.querySelector('div') //获取div标签对象
          div.className = 'xbav xbox' //加入两个样式, 既保持字体样式, 又保持背景样式
    }
    .xbox{ /*自己取个类名*/
          width: 600px;
          height: 700px;
          background-color: #skyblue;
          margin: 100px auto;
          padding: 10px;
          border: 1px solid #000;
    }
    .xbav{ /*字体样式*/
          font-size: 50px;
          color: #red;
    }
</style>
</head>
<body> 123
      <div class="xnav">123</div>
```



对象.classList.add(样式名字符串), //JS 使用类名给标签添加新样式, 就样式不会被覆盖



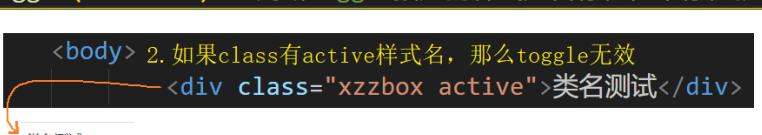
```
<style>
    .xzzbox{ /*我定义的默认样式*/
        width: 200px;
        height: 200px;
        color: #333;
    }
    .active{ /*定义新样式*/ ✓
        color: red;
        background-color: blue;
    }
</style>
</head>
<body>
    <div class="xzzbox">类名测试</div>

```

const div = document.querySelector('.xzzbox') //获取div类名  
div.classList.add('active') //添加类名

所以classList.add的好处就是新加的样式不会覆盖以前的样式

对象.classList.remove(样式名字符串), //删除新添加的样式类



```
<style>
    .xzzbox{ /*我定义的默认样式*/
        width: 200px;
        height: 200px;
        color: #333;
    }
    .active{ /*定义新样式*/
        color: red;
        background-color: blue;
    }
</style>
</head>
<body>
    <div class="xzzbox">类名测试</div>

```

const div = document.querySelector('.xzzbox') //获取div类名  
div.classList.toggle('active') //判断toggle指定的样式是否存在, 不存在就直接加上

类名测试

1. 因为标签里面没有 active样式, 所以 toggle生效

类名测试

2. 如果class有active样式名, 那么toggle无效

类名测试

轮播图简单案例



`<body>`

```

<div class="slider">
    <div class="slider-wrapper">
        
    </div>
    <div class="slider-footer">
        <p>轮播图显示</p>
        <ul class="slider-indicator">
            <li class="active"></li>
            <li></li>
            <li></li>
            <li></li>
            <li></li>
            <li></li>
        </ul>
        <div class="toggle">
            <button class="prev">&lt;</button>
            <button class="next">&lt;</button>
        </div>
    </div>

```

```

.slider-indicator li{ /*让每个'点'颜色变浅*/
    width: 8px;
    height: 8px;
    margin: 4px;
    border-radius: 50%;
    background: #fff;
    opacity: 0.4;
    cursor: pointer;
}
.slider-indicator li.active{ /*选中的'点'颜色会变深，其余'点'颜色变浅*/
    width: 12px;
    height: 12px;
    opacity: 1;
}

```

```

.slider-indicator{
    margin: 0;
    padding: 0;
    list-style: none;
    display: flex;
    align-items: center;
}
.slider-footer{
    height: 80px;
    background-color: #rgb(100, 67, 68);
    padding: 12px 12px 0 12px;
    position: relative;
}
.slider-wrapper{
    width: 100%;
    height: 320px;
}

```

小点横排摆放

增加横排div背景色

修改背景图片尺寸，导致选项‘点’div移动到图片中间

`.slider{ width: 560px; height: 400px; overflow: hidden; }`

整个盒子尺寸缩小

`.slider-footer p{ margin: 0; color: #fff; font-size: 18px; margin-bottom: 10px; }`

文字变白，选项'点上移'

`<div class="slider"> <div class="slider-wrapper">  </div>`

我们要获取图片标签，让JS可以修改图片路径

`//需要轮播的初始数据 const sliderData = [ {url: './1.jpg', title: '轮播第1111张图片', color: 'rgb(100,67,68)'}, {url: './2.jpg', title: '轮播第2222张图片', color: 'rgb(43,35,26)'}, {url: './3.jpg', title: '轮播第3333张图片', color: 'rgb(36,31,33)'}, {url: './4.jpg', title: '轮播第4444张图片', color: 'rgb(139,98,66)'}, {url: './5.jpg', title: '轮播第5555张图片', color: 'rgb(67,90,92)'} ]`

`//随机数生成函数 function getRandom(N,M){ return Math.floor(Math.random() * (M - N + 1)) + N }`

`const random = getRandom(0,4) //生成0~4随机数 console.log(sliderData[random])`

`//获取slider-wrapper下img标签对象 const img = document.querySelector('.slider-wrapper img')`

`img.src = sliderData[random].url //获取sliderData数组里面的图片地址`

我每次刷新页面，图片会变，但是  
图片下面的指示'点没有变'

`<div class="slider-footer"> <p>轮播图显示</p>`

JS获取p标签，修改里面的文字

`const random = getRandom(0,4) //生成0~4随机数 console.log(sliderData[random])`

`//获取slider-wrapper下img标签对象 const img = document.querySelector('.slider-wrapper img')`

`img.src = sliderData[random].url //获取sliderData数组里面的图片地址`

`//获取slider-footer下p标签对象 const p = document.querySelector('.slider-footer p')`

`//将数组对应的文字赋值给p标签，替换p标签原来的文字 p.innerHTML = sliderData[random].title`

不同页面，不同长条背景色

`//修改轮播图下面长条div背景色 const footer = document.querySelector('.slider-footer') footer.style.backgroundColor = sliderData[random].color`

不同的背景色

`<div class="slider-footer"> <p>轮播图显示</p> <ul class="slider-indicator"> <li class="active"></li> <li></li> <li></li> 第1个指示点高亮 <li></li> <li></li> <li></li> <li></li> </ul>`

`<div class="slider-footer"> <p>轮播图显示</p> <ul class="slider-indicator"> <li></li> <li></li> 最后一个指示点高亮 <li></li> <li></li> <li class="active"></li> </ul>`

`/*选中的'点'颜色会变深，其余'点'颜色变浅*/ .slider-indicator li.active{ width: 12px; height: 12px; opacity: 1; }`

<div class="slider-footer">  
 <p>轮播图显示</p>  
 <ul class="slider-indicator">  
 <li></li>  
 <li></li>  
 <li></li> ← 使用伪类选择器，用  
 <li></li> JS修改高亮的点  
 <li></li>  
 <li class="active"></li>  
 </ul>  
 </div>

// 使用css伪类选择器 nth-child(标签序号),  
// li:nth-child(3) 选择多个li标签里面的第3个li标签 我选择高亮第3个点  
const li = document.querySelector('.slider-indicator li:nth-child(3)')  
li.classList.add('active') 使用类名加载方式让第3个点设置active样式

// 使用css伪类选择器 nth-child(标签序号),  
// li:nth-child(3) 选择多个li标签里面的第3个li标签  
const li = document.querySelector('.slider-indicator li:nth-child(\${random+1})')  
li.classList.add('active') 直接使用`加入\${}变量就可以让高亮点随机指示

获取页面输入框内容(获取表单的值)

返回值=对象.value //获取表单中的值

对象.type='password' //表单不显示输入内容, type = 'text' 显示内容

类型固定字符      输入框提示, 自己定义

<input type="text" value="输入框测试">

const uname = document.querySelector('input') //获取输入框内容  
 console.log(uname.value) ← value就是获取输入框的值

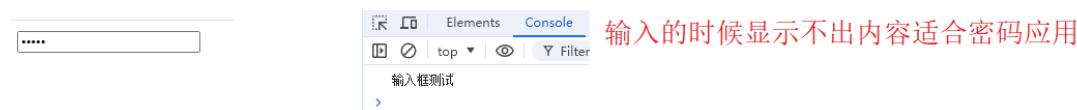


我还没有输入值, 默认值就被JS获取过来了

const uname = document.querySelector('input') //获取输入框内容  
 console.log(uname.type) //打印表单数据类型

表单类型是text, 所以无法用innerHTML获取

const uname = document.querySelector('input') //获取输入框内容  
 uname.type = 'password' //设置表单隐藏内容显示  
 console.log(uname.value)



输入的时候显示不出内容适合密码应用

## 操作表单元素，属性

```
<input type='checkbox' //勾选框
```

```
<button disabled='....' //按钮框
```

```
<input type="checkbox" name="" id="">
```

```
const ipt = document.querySelector('input')
console.log(ipt.checked) //获取是否被勾选状态
```

返回false没被勾选，返回true勾选中



```
Elements Console
top Filter
false
```

```
const ipt = document.querySelector('input')
ipt.checked = true //自动勾选上' true' 也能勾选上
console.log(ipt.checked) //获取是否被勾选状态
```

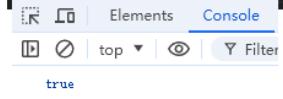
字符' true' 因为系统隐式转换成了bool值，所以也能给checked使用。

```
<button disabled="disabled">点击</button>
```



按钮的disabled属性选择disabled，就是无法点击

```
const btn = document.querySelector('button')
console.log(btn.disabled) //查看按钮disabled属性状态
```



```
Elements Console
top Filter
true
```

返回true，表示不可用disabled功能被打开

## H5 自定义属性

### 自定义属性

- **标准属性:** 标签天生自带的属性 比如class id title等，可以直接使用点语法操作比如：disabled、checked、selected

- **自定义属性:**

- 在html5中推出来了专门的data-自定义属性
- 在标签上一律以data-开头
- 在DOM对象上一律以dataset对象方式获取

```
<body>
  <div data-id="1">111</div>
  <div data-id="2">222</div>
  <div data-id="3">333</div>
  <div data-id="4">444</div>
  <div data-id="5">555</div>
```

```
const onediv = document.querySelector('div') //获取5个div对象
console.log(onediv.dataset) //使用dataset获取自定义属性data的标签
```



dataset不指定，就默认获取第1个div标签的属性

```

<div data-id="1" data-xzzs="不清楚">111</div>
<div data-id="2">222</div>
<div data-id="3">333</div>
<div data-id="4">444</div>
<div data-id="5">555</div>

```

一个标签可以自定义  
很多个属性

```

const onediv = document.querySelector('div') //获取5个div对象
console.log(onediv.dataset.id) //获取第1个自定义的属性
console.log(onediv.dataset.xzzs) //获取data后面自己新增的属性

```

这个自定义属性更多的用法，还有获取标签里面的内容，后面再讲

## 定时器间歇函数

`setInterval(函数, 间隔时间)//间隔时间单位是毫秒, 开启定时器`

`clearInterval(变量名)//传入定时器序号, 关闭定时器`

```

function xzzfun()
{
    console.log('500毫秒执行一次')
}
// 500毫秒打印一次，在计数打印次数
setInterval(xzzfun,500) //每间隔500毫秒执行一次xzzfun函数

```

```

//返回值给 n 是定时器ID号
let n = setInterval(function(){
    console.log('第1个定时器执行一次')
},500)

console.log(n) //开启1个定时器, n = 1

```

```

setInterval(function(){
    console.log('匿名函数方式500毫秒执行一次')
},500)

```

```

//返回值给 t 是定时器ID号
let t = setInterval(function(){
    console.log('第2个定时器执行一次')
},500)

console.log(t) //开启1个定时器, n = 2

```

```

//返回值给 n 是定时器ID号
let n = setInterval(function(){
    console.log('第1个定时器执行一次')
},500)

console.log(n) //开启1个定时器, n = 1

```

定时器1关闭了

```

//返回值给 t 是定时器ID号
let t = setInterval(function(){
    console.log('第2个定时器执行一次')
},500)

console.log(t) //开启1个定时器, n = 2

```

定时器1关闭了

```

clearInterval(n) //关闭第1个定时

```

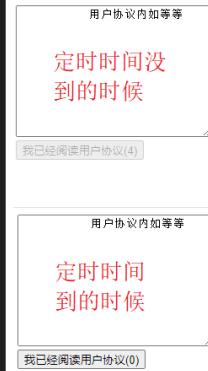
定时器1关闭了

实现一个定时器时间到，才可以操作按钮的功能

```
<textarea name="" id="" cols="30" rows="10">
    用户协议内如等等
</textarea>          先让按钮变成灰色不可选
<br>
<button class="btn" disabled>我已经阅读用户协议(60)</button>

const btn = document.querySelector('.btn')
let i = 5 //倒计时

//开启定时器
let n = setInterval(function(){
    i --
    btn.innerHTML = `我已经阅读用户协议(${i})`           让按钮标签内容改变
    //按钮比较特殊，按钮属于表单，所以只能用innerHTML去修改内容
    if(i === 0)           定时5秒时间到，按钮
    {
        clearInterval(n) //关闭定时器           可以点击
        btn.disabled = false //取消按钮禁用
    }
},1000)
```



## 轮播图定时器切换

**案例 轮播图定时器版**

需求：每隔一秒钟切换一个图片

分析：

①：准备一个数组对象，里面包含详细信息（素材包含）

②：获取元素

③：设置定时器函数

    设置一个变量++  
    找到变量对应的对象  
    更改图片、文字信息  
    激活小圆点：移除上一个高亮的类名，当前变量对应的小圆点添加类

④：处理图片自动复原从头播放（放到变量++后面，紧挨）

    如果图片播放到最后一张，就是大于等于数组的长度  
    则把变量重置为0

```
<div class="slider">
    <div class="slider-wrapper">
        
    </div>
    <div class="slider-footer">
        <p>轮播图显示</p>
        <ul class="slider-indicator">
            <li class="active"></li>
            <li></li>
            <li></li>
            <li></li>
            <li></li>
            <li></li>
        </ul>
    </div>
    <div class="toggle">
        <button class="prev">&lt;&lt;&lt;&lt;&lt;&lt;</button>
        <button class="next">&lt;&lt;&lt;&lt;&lt;&lt;</button>
    </div>
</div>
```

```

<style>
  .slider-indicator {
    margin: 0;
    padding: 0;
    list-style: none;
    display: flex;
    align-items: center;
  }
  .slider-footer {
    height: 80px;
    background-color: #fff;
    padding: 12px 12px 0 12px;
    position: relative;
  }
  .slider-wrapper {
    width: 100%;
    height: 320px;
  }
  .slider {
    width: 560px;
    height: 400px;
    overflow: hidden;
  }
  .slider-footer p {
    margin: 0;
    color: #fff;
    font-size: 18px;
    margin-bottom: 10px;
  }
  .slider-indicator li {
    width: 8px;
    height: 8px;
    margin: 4px;
    border-radius: 50%;
    background: #fff;
    opacity: 0.4;
    cursor: pointer;
  }
  .slider-indicator li.active {
    width: 12px;
    height: 12px;
    opacity: 1;
  }
</style>

```

```

const sliderData = [
  {url:'./1.jpg',title:'轮播图第1111张图片',color:'rgb(100,67,68)'},
  {url:'./2.jpg',title:'轮播图第2222张图片',color:'rgb(43,35,26)'},
  {url:'./3.jpg',title:'轮播图第3333张图片',color:'rgb(36,31,33)'},
  {url:'./4.jpg',title:'轮播图第4444张图片',color:'rgb(139,98,66)'},
  {url:'./5.jpg',title:'轮播图第5555张图片',color:'rgb(67,90,92)'}
]

```

```

//1.获取元素
const img = document.querySelector('.slider-wrapper img')
const p = document.querySelector('.slider-footer p')
let i = 0

setInterval(function(){
  i++
  img.src = sliderData[i].url //取图片地址
  p.innerHTML = sliderData[i].title // 把字写入p标签
  //可以不用返回值去，什么
  //而是直接.classList也可以
  //删除以前高亮的小圆点
  document.querySelector('.slider-indicator .active').classList.remove('active')
  //取新的小圆点
  xyd = document.querySelector('.slider-indicator li:nth-child(${i+1})')
  xyd.classList.add('active') //每翻一页添加个高亮圆点
},1000)

```



## 事件监听

事件监听就是鼠标点击按钮有对应得代码执行，鼠标移动到指定区域有指定的代码执行。

**对象.addEventListener('事件类型', 要执行的函数)//事件监听加载函数**

事件类型: 'click' 鼠标点击事件

'mouseenter' 鼠标经过事件

'mouseleave' 鼠标离开事件

'focus' 焦点事件，鼠标移动到某个标签区域(输入框)触发

```

<button>点击</button>

function xzzfun()
{
    console.log('按钮点击事件被触发')
}

const btn = document.querySelector('button') //获取按钮对象
btn.addEventListener('click',xzzfun) //鼠标点击事件，xzzfun函数被执行

```

The screenshot shows a browser's developer tools with the 'Console' tab selected. It displays the message '按钮点击事件被触发' (Button click event triggered) in the log.

```

const btn = document.querySelector('button') //获取按钮对象
btn.addEventListener('click',function(){

    console.log('按钮点击事件，匿名函数出发')

}) //鼠标点击事件，可以匿名函数实现

```

The screenshot shows a browser's developer tools with the 'Console' tab selected. It displays the message '按钮点击事件，匿名函数出发' (Button click event, anonymous function triggered) in the log.

```

<div class="box">
    我是广告
    <div class="box1">x</div>
</div>
<style>
    .box
    {
        width: 300px;
        height: 400px;
        background-color: #blanchedalmond;
        font-size: 50px;
        line-height: 200px;
        font-weight: 700;
    }

```

```

.box1{
    position: absolute;
    right: 20px;
    top: 10px;
    width: 20px;
    height: 20px;
    background-color: #rgb(188, 188, 236);
    text-align: center;
    line-height: 20px;
    font-size: 16px;
}

const box1 = document.querySelector('.box1') //获取小盒子
const box = document.querySelector('.box') //获取大盒子

box1.addEventListener('click', function(){

    box.style.display = 'none' //隐藏大盒子
})

```

The screenshot shows a browser's developer tools with the 'Console' tab selected. It displays the message '按钮点击事件，匿名函数出发' (Button click event, anonymous function triggered) in the log.

## 随机点名案例



### 步骤 • 随机点名案例

业务分析：

- ① 点击开始按钮随机抽取数组的一个数据，放到页面中
  - ② 点击结束按钮删除数组当前抽取的一个数据
  - ③ 当抽取到最后一个数据的时候，两个按钮同时禁用（写点开始里面，只剩最后一个数据不用抽了）
- 核心：利用定时器快速展示，停止定时器结束展示

```

<style>
    .box{
        display: flex;
    }
</style>

</head>
<body>
    <div class="box">
        <span>名字是 : </span>
        <div class="qs">这里显示姓名</div>
    </div>
    <button class="start">开始</button>
    <button class="end">结束</button>

```

```

const arr = ['马超', '黄忠', '赵云', '关羽', '张飞']
const start = document.querySelector('.start') //获取开始按钮

const qs = document.querySelector('.qs')

//开始按钮点击事件
start.addEventListener('click',function(){
    const random = parseInt(Math.random() * arr.length) //获得随机数
    qs.innerHTML = arr[random] //开始按钮之后随即获取名字
})

```

名字是:赵云 ← 点击开始就能随即得到一个名字

```

const arr = ['马超', '黄忠', '赵云', '关羽', '张飞']
const start = document.querySelector('.start') //获取开始按钮

const qs = document.querySelector('.qs')

let gtime = 0 //定时器全局变量，方便开启和关闭定时
//开始按钮点击事件
start.addEventListener('click',function(){

    gtime = setInterval(function(){
        const random = parseInt(Math.random() * arr.length) //获得随机数
        qs.innerHTML = arr[random] //开始按钮之后随即获取名字
    },35) //35毫秒刷新
})

const end = document.querySelector('.end')
end.addEventListener('click',function(){
    clearInterval(gtime)
})

```

点击开始，名字不停的变化，点击结束才停止变化

## 事件监听版本及鼠标事件

```
const btn = document.querySelector('button')
btn.onclick = function () {
  alert(11) 早期版本的事件监听是这样写，但是有些BUG  
主要是容易出现事件覆盖的问题
}

const btn = document.querySelector('button')
btn.onclick = function () {
  alert(11)
}
btn.onclick = function () {
  alert(22) 就是这种，后面覆盖前面的事件
}

btn.addEventListener('click', function () {
  alert(11) 使用add...是我们现在的事件写法，没有问题
})
```



文本事件

input 用户输入事件

```
<style>
  div{
    width: 200px;
    height: 200px;
    background-color: pink;
  }
</style>

</head>
<body>

<div></div>
```

```
//鼠标离开div框，mouseleave事件触发
div.addEventListener('mouseleave',function(){
  console.log('鼠标离开')
})
```



鼠标触发

click 鼠标点击  
mouseenter 鼠标经过  
mouseleave 鼠标离开



键盘触发

keydown 键盘按下触发  
keyup 键盘抬起触发



焦点事件

焦点事件就是鼠标移动到某个位置弹出其它框框

表单获得光标

focus 获得焦点  
blur 失去焦点

```
const div = document.querySelector('div') //获取div对象

//鼠标经过div框，mouseenter事件触发
div.addEventListener('mouseenter', function(){
  console.log('鼠标经过本框')
})
```

每次鼠标经过div框就  
会触发一次事件

鼠标经过

```
鼠標經過本程  
鼠標離開  
鼠標經過本程  
鼠標離開
```

鼠标进入粉色框和离开  
粉色框都有事件触发

## 轮播图点击切换案例

### 轮播图点击切换

需求：当点击左右的按钮，可以切换轮播图

分析：

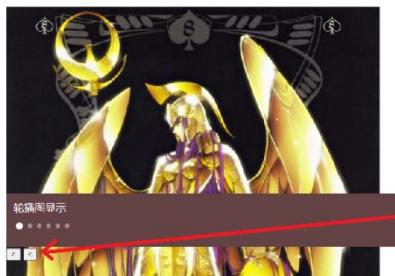
- ①：右侧按钮点击，变量++，如果大于等于8，则复原0
- ②：左侧按钮点击，变量--，如果小于0，则复原最后一张
- ③：鼠标经过暂停定时器
- ④：鼠标离开开启定时器

## 轮播图点击切换

需求：当点击左右的按钮，可以切换轮播图

分析：

- ①：右侧按钮点击，变量++，如果大于等于8，则复原0
- ②：左侧按钮点击，变量--，如果小于0，则复原最后一张
- ③：鼠标经过暂停定时器
- ④：鼠标离开开启定时器



```
<div class="slider">
  <div class="slider-wrapper">
    
  </div>
  <div class="slider-footer">
    <p>轮播图显示</p>
    <ul class="slider-indicator">
      <li class="active"></li>
      <li></li>
      <li></li>
      <li></li>
      <li></li>
    </ul>
  </div>
  <div class="toggle">
    <button class="prev">&lt;&lt;;</button>
    <button class="next">&lt;&lt;;</button>
  </div>
```

```
<style>
  .slider-indicator {
    margin: 0;
    padding: 0;
    list-style: none;
    display: flex;
    align-items: center;
  }
  .slider-footer{
    height: 80px;
    background-color: #rgb(100, 67, 68);
    padding: 12px 12px 0 12px;
    position: relative;
  }
  .slider-indicator li.active{
    width: 12px;
    height: 12px;
    opacity: 1;
  }
```

```
.slider-wrapper{
  width: 100%;
  height: 320px;
}
.slider{
  width: 560px;
  height: 400px;
  overflow: hidden;
  display: contents;
}
```

为了让按钮出来，使  
用了该布局

```
.slider-footer p{
  margin: 0;
  color: #fff;
  font-size: 18px;
  margin-bottom: 10px;
}

.slider-indicator li{
  width: 8px;
  height: 8px;
  margin: 4px;
  border-radius: 50%;
  background: #fff;
  opacity: 0.4;
  cursor: pointer;
}
```

```

const sliderData = [
    {url:'./1.jpg',title:'轮播图第111张图片',color:'rgb(100,67,68)'},
    {url:'./2.jpg',title:'轮播图第222张图片',color:'rgb(43,35,26)'},
    {url:'./3.jpg',title:'轮播图第333张图片',color:'rgb(36,31,33)'},
    {url:'./4.jpg',title:'轮播图第444张图片',color:'rgb(139,98,66)'},
    {url:'./5.jpg',title:'轮播图第555张图片',color:'rgb(67,90,92)'},
]

const img = document.querySelector('.slider-wrapper img') //获取图片标签
const p = document.querySelector('.slider-footer p') //获取文字标签
const footer = document.querySelector('.slider-footer')

let i = 0
const next = document.querySelector('.next') //获取下一步按钮

//下一步按钮事件触发
next.addEventListener('click',function(){
    i ++
    img.src = sliderData[i].url
    p.innerHTML = sliderData[i].title
    footer.style.backgroundColor = sliderData[i].color

    //更换小圆点，先移除原来的类名，当前li再添加新类名
    //先移除原来的active
    document.querySelector('.slider-indicator .active').classList.remove('active')
    //添加新小圆点
    document.querySelector(`.slider-indicator li:nth-child(${i + 1})`).classList.add('active')
})

```

下面实现图片自动切换，鼠标移动到图片上，停止切换，鼠标离开图片继续切换

```

//启用定时器切换图片
let tim = setInterval(function(){
    next.click() //强制执行下一步按钮(这种就是不用手点击，定时器帮你执行)
},500)

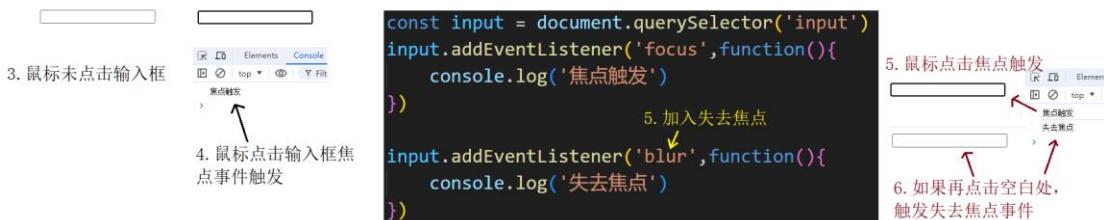
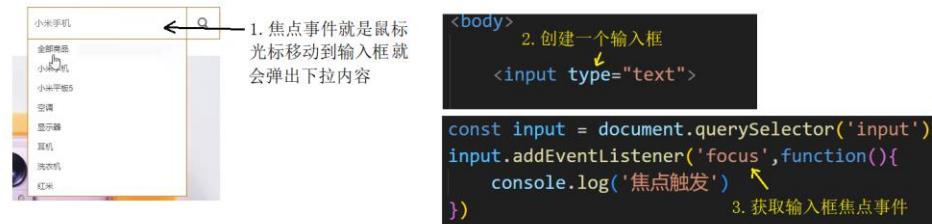
const slider = document.querySelector('.slider-footer') //获取横框

//鼠标移动到横框停止定时器
slider.addEventListener('mouseenter',function(){
    clearInterval(tim) //停止定时器
})

//鼠标离开横框启动定时，继续切换图片
slider.addEventListener('mouseleave',function(){
    //开启定时器(就是将之前的开启定时器代码再写一遍，不要let)
    tim = setInterval(function(){
        next.click()
    },500)
})

```

## 焦点事件



```
<style>
  *{
    margin: 0;
    padding: 0;
    box-sizing: border-box;
  }
  ul{
    list-style: none;
  }
  .mi{
    position: relative;
    width: 223px;
    margin: 100px auto;
  }
</style>
.mi input{
  width: 223px;
  height: 48px;
  padding: 0 10px;
  font-size: 14px;
  line-height: 48px;
  border: 1px solid #e0e0e0;
  outline: none;
}
.mi .search{
  border: 1px solid #ff6700;
}
.result-list{
  position: absolute;
  left: 0;
  top: 48px;
  width: 223px;
  border: 1px solid #ff6700;
  border-top: 0;
  background: #fff;
}
.result-list a{
  display: block;
  padding: 6px 15px;
  font-size: 12px;
  color: #424242;
  text-decoration: none;
}
```

```
<body>
  <div class="mi">
    <input type="search" placeholder="小米笔记本">
    <ul class="result-list">
      <li><a href="#">全部商品</a></li>
      <li><a href="#">小米11</a></li>
      <li><a href="#">小米10s</a></li>
      <li><a href="#">小米笔记本</a></li>
      <li><a href="#">小米手机</a></li>
      <li><a href="#">黑纱</a></li>
      <li><a href="#">空调</a></li>
    </ul>
  </div>
```

小米笔记本  
全部商品  
小米11  
小米10s  
小米笔记本  
小米手机  
黑纱  
空调

标签写完之后，下拉框提前出来了，下面我们用 JS 隐藏下来框

```

.result-list{
  display: none; // 2. 先屏蔽输入框的下拉选项
  position: absolute;
  left: 0;
  top: 48px;
  width: 223px;
  border: 1px solid #ff6700;
  border-top: 0;
  background: #fff;
}

const input = document.querySelector('[type=search]')
// [type=search] 这种写法也是获取input，就是css的属性选择器

const ul = document.querySelector('.result-list')
input.addEventListener('focus', function(){
  ul.style.display='block' // 4. 鼠标进入输入框，点击，修改CSS用block显示下拉框
})

.mi .search{ // 7. 加入输入框颜色，修改css中的输入框颜色
  border: 1px solid #ff6700;
}

小米笔记本 // 5. 本来没有
全部商品 小米11 小米10s 小米笔记本 小米手机 黑粉 空清 // 6. 鼠标点击一下，有了
小米笔记本 // 8. 输入框有颜色了 // 鼠标离开输入框，下拉菜单隐藏
全部商品 小米11 小米10s 小米笔记本 小米手机 黑粉 空清 // 9. 鼠标离开后，下拉菜单隐藏了

```

## 键盘事件，发布评论案例

```

<input type="text">

const input = document.querySelector('input')

1. 键盘按下事件
input.addEventListener('keydown', function(){
  console.log('键盘按下了')
})

2. 键盘松开事件
input.addEventListener('keyup', function(){
  console.log('键盘弹起了')
})

3. 键盘输入了两次
 // 键盘输入了两次

const input = document.querySelector('input')
4. 输入事件
input.addEventListener('input', function(){
  console.log('用户输入内容')
})

5. 用户输入内容被触发
 // 用户输入内容

6. 每次输入框输入值，我都能获取
 // 每次输入框输入值，我都能获取

```

### 案例 评论字数统计

需求：用户输入文字，可以计算用户输入的字数

121212121212121

发布

检测用户输入字数 → 15/200字

60 / 120

```

<style>
  .wrapper{
    min-width: 400px;
    max-width: 800px;
    display: flex;
    justify-content: flex-end;
  }

  .avatar{
    width: 48px;
    height: 48px;
    border-radius: 50%;
    overflow: hidden;
  }

  .wrapper textarea{
    outline: none;
    border-color: transparent;
    resize: none;
    background: #f5f5f5;
    border-radius: 4px;
    flex: 1;
    padding: 10px;
    transition: all 0.5s;
    height: 30px;
  }

  .wrapper textarea:focus{
    border-color: #e4e4e4;
    background: #fff;
    height: 50px;
  }

  .wrapper button{
    background: #00aeec;
    color: #fff;
    border: none;
    border-radius: 4px;
    margin-left: 10px;
    width: 70px;
    cursor: pointer;
  }

  .wrapper .total{
    margin-right: 80px;
    color: #999;
    margin-top: 5px;
    opacity: 0;
    transition: all 0.5s;
  }
</style>

```

```

.list {
  min-width: 400px;
  max-width: 800px;
  display: flex;
}

.list .item{
  width: 100%;
  display: flex;
}

</style>

<div class="list">
  <div class="wrapper">
    <i class="avatar"></i>
    <textarea id="tx" placeholder="发一条评论" rows="2" maxlength="50"></textarea>
    <button>发布</button>
  </div>

  <div class="list-item">
    <div class="wrapper">
      <span class="total">0/200字</span>
    </div>
  </div>
</div>

```

点击前的效果

点击后的效果

```

const tx = document.querySelector('#tx')
const total = document.querySelector('.total')

tx.addEventListener('focus',function(){ //获得输入框焦点
  total.style.opacity = 1 字数显示
})

tx.addEventListener('blur',function(){ //输入框失去焦点
  total.style.opacity = 0 字数不显示
})

```

输入前

输入后，注意字数有显示了

输入字数有统计

```

tx.addEventListener('input',function(){ //每输入一个字，就触发输入事件
  total.innerHTML = `${tx.value.length}/200字` //直接修改整个标签内容
})

```

## 事件对象

### 获取事件对象

目标：能说出什么是事件对象

- 事件对象是什么

- 语法：如何获取

- 在事件绑定的回调函数的第一个参数就是事件对象

- 一般命名为event、ev、e

- 也是个对象，这个对象里有事件触发时的相关信息

- 例如：鼠标点击事件中，事件对象就存了鼠标点在哪个位置等信息

- 使用场景

- 可以判断用户按下哪个键，比如按下回车键可以发布新闻

- 可以判断鼠标点击了哪个元素，从而做相应的操作



- 一般命名为event、ev、e

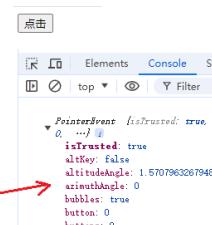
事件对象

```
元素.addEventListener('click', function (e) {  
})
```

```
<button>点击</button>
```

```
const btn = document.querySelector('button')  
这就是事件对象  
btn.addEventListener('click', function(e){  
    console.log(e) 记住，这个e是在addEventListener里面  
    才是事件对象，如果没有add...这个e就是形参  
})
```

点击按钮，e返回一堆属性变量和函数，这里就是e返回的。因为e是对象，所以有属性变量和函数



- 部分常用属性

- type

比如上一个案例，  
获取当前的事件类型 点击按钮事件



- clientX/clientY

获取光标相对于浏览器可见窗口左上角的位置  
淘宝有个页面部分位置放大效果，就是用的clientXY



- offsetX/offsetY

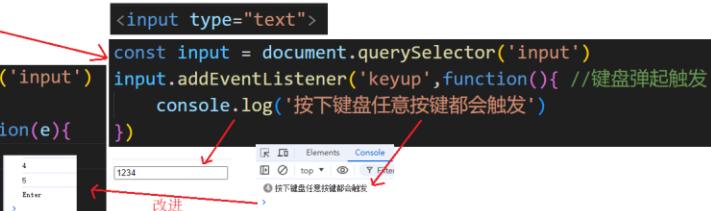
获取光标相对于当前DOM元素左上角的位置  
也就是鼠标在标签这个盒子里面的相对位置

- key

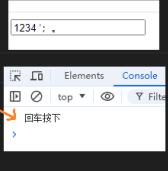
用户按下的键盘键的值

现在不提倡使用keyCode

```
const input = document.querySelector('input')
//我只想按了回车键触发
input.addEventListener('keyup',function(e){
    //获取到Enter字符就是回车
    console.log(e.key)
})
```



```
const input = document.querySelector('input')
//我只想按了回车键触发
input.addEventListener('keyup',function(e){
    //获取到Enter字符就是回车
    //console.log(e.key)
    if(e.key === 'Enter'){
        console.log('回车按下')
    }
})
```



```
<div class="wrapper">      输入框回车获取字符案例，CSS前面案例已经写过，这儿就不写了
    <i class="avatar"></i>
    <textarea id="tx" placeholder="发布一条评论" rows="2" maxlength="50">

    </textarea>
    <button>发布</button>
</div>
<div class="wrapper">
    <span class="total">0/200字</span>
</div>
```

```
const tx = document.querySelector('#tx')
const total = document.querySelector('.total')
```

```
tx.addEventListener('focus',function(){
    total.style.opacity = 1
})
tx.addEventListener('blur',function(){
    total.style.opacity = 0
})
```

输入回车



```
tx.addEventListener('keyup',function(e){
    if(e.key === 'Enter') //键盘回车触发
    {
        console.log(tx.value) //获取输入的字符
    }
})
```

## 评论回车发布案例



分析：

- ①：用到按下键盘事件 keydown 或者 keyup 都可以
- ②：如果用户按下的是回车键，则发布信息
- ③：让留言信息模块显示，把拿到的数据渲染到对应标签内部

```
<div class="wrapper">
    <i class="avatar"></i>
    <textarea id="tx" placeholder="发布一条评论" rows="2" maxlength="50">
    </textarea>
    <button>发布</button>
</div>
<div class="wrapper">
    <span class="total">0/200字</span>
</div>
<div class="list"> ← 新加入的评论框
    <div class="item" style="display: block;">
        <i class="avatar"></i>
        <div class="info">
            <p class="name">向仔用户 </p>
            <p class="text">大家辛苦了 </p>
            <p class="time">2022-10-10 20:29:21</p>
        </div>
    </div>
</div>
```

发布

0/200字

向仔用户  
大家辛苦了

2022-10-10 20:29:21

```

.list {
    min-width: 400px;
    max-width: 800px;
    display: flex;
}

.list .item {
    width: 100%;
    display: flex;
}

.list .item .info {
    flex: 1;
    border-bottom: 1px dashed #e4e4e4;
    padding-bottom: 10px;
}

.list .item p {
    margin: 0;
}

.list .item .name {
    color: #fb7299;
    font-size: 14px;
    font-weight: bold;
}

.list .item .text {
    color: #333;
}

.list .item .time {
    color: #999;
    font-size: 12px;
}

```

2. 实际上要求是输入文字，按了回车之后，评论字符才显示，所以评论框先屏蔽掉none

1. 经过以上list评论框样式修改之后，效果达到了。

```

const tx = document.querySelector('#tx')
const total = document.querySelector('.total')

tx.addEventListener('focus',function(){
    total.style.opacity = 1
})
tx.addEventListener('blur',function(){
    total.style.opacity = 0
})  

主要就是这段代码

```

const item = document.querySelector('.item') //回去评论框标签对象  
const text = document.querySelector('.text') //获取评论区域对象

//按下回车发布评论  
tx.addEventListener('keyup',function(e){
 if(e.key === 'Enter') //键盘回车触发
 {
 item.style.display = 'block' //评论区显示
 text.innerHTML = tx.value //将输入框的内容赋值给评论区
 tx.value = '' //回车之后清空输入框里面字符
 }
})



字符对象.trim() //取出字符两边的空格

```

const str = '          pink      '
console.log(str.trim())

```

pink	有空格
pink	无空格

```

//按下回车发布评论
tx.addEventListener('keyup',function(e){
    if(e.key === 'Enter') //键盘回车触发
    {
        1. 加入空字符判断之后
        if(tx.value.trim() !== '') //如果输入内容不为空，表示用户输入正常数字
        {
            item.style.display = 'block' //评论区显示
            text.innerHTML = tx.value //将输入框的内容赋值给评论区
        }
        tx.value = '' //回车之后清空输入框里面字符
    }
})

```



## 环境对象 this

### 环境对象

目标：能够分析判断函数运行在不同环境中 this 所指代的对象

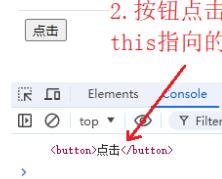
**环境对象：**指的是函数内部特殊的变量 this，它代表着当前函数运行时所处的环境

**作用：**弄清楚this的指向，可以让我们代码更简洁

```
//每个函数里面都有this环境对象
function fn(){
  console.log(this) //普通函数里面的this指向本机浏览器窗口window
}
```

```
<button>点击</button>
const btn = document.querySelector('button')
btn.addEventListener('click',function(){
  console.log(this) ← 1. 按钮事件函数里面
})
```

2. 按钮点击之后，发现 this指向的是按钮标签



- 函数的调用方式不同，this 指代的对象也不同
- 【谁调用， this 就是谁】是判断 this 指向的粗略规则

谁调用这个函数，this 就指向谁

```
<button>点击</button>
const btn = document.querySelector('button')
btn.addEventListener('click',function(){
  this.style.color = 'red'
})
```

按钮触发，调用该回调事件，那么回调事件下的this就是指向调用者，也就是按钮

this实际应用案例如下：



1. 你鼠标指向谁，谁就变成红色

2. 鼠标指向另外一个字符，该字符变成红色

## 回调函数

目标：能够说出什么是回调函数

如果将函数 A 做为参数传递给函数 B 时，我们称函数 A 为**回调函数**

简单理解：当一个函数当做参数来传递给另外一个函数的时候，这个函数就是**回调函数**

- 常见的使用场景：

```
function fn() {  
    console.log('我是回调函数...')  
}  
  
// fn传递给了setInterval, fn就是回调函数  
setInterval(fn, 1000)
```

```
box.addEventListener('click', function () {  
    console.log('我也是回调函数')  
})
```

## TAB 切换案例



The screenshot shows a tab-based interface. On the left, there's a navigation bar with five tabs labeled "精选", "美食", "百货", "个护", and "预告". The first tab, "精选", is active and highlighted with a yellow background. On the right, there's a content area displaying five images corresponding to the tabs. The first image is also active and highlighted with a yellow border. Arrows point from the active tab to the active image.

```
<div class="tab">  
  <div class="tab-nav">  
    <h3>每日特价</h3>  
    <ul>  
      <li><a href="" class="active">精选</a></li>  
      <li><a href="">美食</a></li>  
      <li><a href="">百货</a></li>  
      <li><a href="">个护</a></li>  
      <li><a href="">预告</a></li>  
    </ul>  
  </div>  
  <div class="tab-content">  
    <div class="item active"> </div>  
    <div class="item"> </div>  
    <div class="item"> </div>  
    <div class="item"> </div>  
    <div class="item"> </div>  
  </div>  
</div>  
  
.tab-nav ul li a{  
  text-decoration: none;  
  border-bottom: 2px solid transparent;  
  color: #333;  
}  
  
.tab-nav ul li a.active{  
  border-color: #e1251b;  
  color: #e1251b;  
}  
  
.tab-nav ul li a{  
  text-decoration: none;  
  border-bottom: 2px solid transparent;  
}
```

```
每日特价  
• 精选  
• 美食  
• 百货  
• 个护  
• 预告
```

```
.tab-nav ul li{  
  margin: 0 20px;  
  font-size: 14px;  
}  
  
.tab-content .item.active{  
  display: block;  
}  
.tab-content .item{  
  display: none;  
}
```



```

const as = document.querySelectorAll('.tab-nav a')
console.log(as)

for(let i = 0; i < as.length; i++){
    //给每个a标签绑定鼠标经过事件
    as[i].addEventListener('mouseenter',function(){
        //移除上一个a标签的active
        document.querySelector('.tab-nav .active').classList.remove('active')
        this.classList.add('active') //使用this, 获取当前as[i]下标函数
    })
}

```

## 全选文本框案例

### 全选文本框案例1

需求：用户点击全选，则下面复选框全部选择，取消全选则全部取消

分析：

①：全选复选框点击，可以得到当前按钮的 checked

②：把下面所有的小复选框状态checked，改为和全选复选框一致

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米手机	小米	¥ 1999
<input checked="" type="checkbox"/>	小米净水器	小米	¥ 4999
<input checked="" type="checkbox"/>	小米电视	小米	¥ 5999

```

<table>
  <tr>
    <th class="allCheck">
      <input type="checkbox" name="" id="checkAll">
      <span class="all">全选</span>
    </th>
    <th>商品</th>
    <th>商家</th>
    <th>价格</th>
  </tr>
  <tr>
    <td>
      <input type="checkbox" name="check" class="ok">
    </td>
    <td>小米净水器</td>
    <td>小米</td>
    <td>4999</td>
  </tr>
</table>

```

对象.checked //获取复选框状态，设置复选框状态

```
const checkall = document.querySelector('#checkAll') //获取大复选框
const cks = document.querySelectorAll('.ok') //获取小复选框

checkall.addEventListener('click',function(){
    console.log(this.checked) //得到当前大复选框选中状态
})
```



```
const checkall = document.querySelector('#checkAll') //获取大复选框
const cks = document.querySelectorAll('.ok') //获取小复选框
```

```
checkall.addEventListener('click',function(){
    //循环设置所有小复选框，让小复选框勾选状态checked = 大复选框的勾选状态
    for(let i = 0; i < cks.length; i++){
        {
            cks[i].checked = true //每个小复选框选中
        }
    }
})
```

商品	商家	价格
小米净水器	小米	4999
小米		2000
4999		3000

```
<table>
<tr>
    <th class="allCheck">
        <input type="checkbox" name="" id="checkAll">
        <span class="all">全选</span>
    </th>
    <th>商品</th>
    <th>商家</th>
    <th>价格</th>
</tr>
<tr>
    <td>
        <input type="checkbox" name="check" class="ok">
        小米净水器
    </td>
    <td>小米</td>
    <td>4999</td>
</tr>
<tr>
    <td>
        <input type="checkbox" name="check" class="ok">
        小米
    </td>
    <td></td>
    <td>2000</td>
</tr>
<tr>
    <td>
        <input type="checkbox" name="check" class="ok">
        4999
    </td>
    <td></td>
    <td>3000</td>
</tr>
</table>
```

```
const checkall = document.querySelector('#checkAll') //获取大复选框
const cks = document.querySelectorAll('.ok') //获取小复选框
```

```
checkall.addEventListener('click',function(){
    //循环设置所有小复选框，让小复选框勾选状态checked = 大复选框的勾选状态
    for(let i = 0; i < cks.length; i++){
        {
            cks[i].checked = checkall.checked //全选赋值给每个小复选框选中
        }
    }
})
```

利用css 复选框选择器 input:checked 我可以利用:checked选中来获取有几个按钮被选中  
如果:checked = 3 就有3个勾选框被勾选，如果:checked=2，只有两个被勾选，全选就不显示勾选

检查小复选框选中的个数，是不是等于 小复选框总的个数，

```

for(let i = 0; i < cks.length; i++){
}
    cks[i].addEventListener('click',function(){
        //判断选中的小复选框个数是不是等于总的小复选框个数
        //一定要写在点击里面，因为每次要获得最新的个数
        console.log(document.querySelectorAll('.ok:checked').length)
        //如果直接querySelector('.ok')表示把所有小复选框选出来，如果加:checked
        //得到小复选框个数('.ok:checked').length
    })
}

```

每次勾选一个，就会统计勾选框被勾选个数 → 1

全选	商品	商家	价格
<input type="checkbox"/>	小米净水器	小米	4999
<input type="checkbox"/>	小米手机	小米	2000
<input type="checkbox"/>	小米电视	小米	3000

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米净水器	小米	4999
<input checked="" type="checkbox"/>	小米手机	小米	2000
<input type="checkbox"/>	小米电视	小米	3000

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米净水器	小米	4999
<input checked="" type="checkbox"/>	小米手机	小米	2000
<input checked="" type="checkbox"/>	小米电视	小米	3000

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米净水器	小米	4999
<input checked="" type="checkbox"/>	小米手机	小米	2000
<input checked="" type="checkbox"/>	小米电视	小米	3000

勾选数量一直在更新 → 2

```

for(let i = 0; i < cks.length; i++){
}
    cks[i].addEventListener('click',function(){
        //判断小复选框是否3个都选中，3个被选中，赋值true给checkall.checked
        checkall.checked = document.querySelectorAll('.ok:checked').length === cks.length
    })
}

```

这种写法就是右边先做判断，如果...length 等于 cks.length，返回true，不然返回 false。这时候全选checkall就得到右边的计算结果。

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米净水器	小米	4999
<input checked="" type="checkbox"/>	小米手机	小米	2000
<input type="checkbox"/>	小米电视	小米	3000

全选不会被选中

全选	商品	商家	价格
<input checked="" type="checkbox"/>	小米净水器	小米	4999
<input checked="" type="checkbox"/>	小米手机	小米	2000
<input checked="" type="checkbox"/>	小米电视	小米	3000

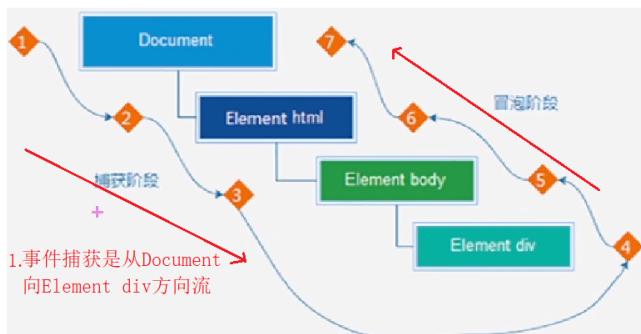
全选被选中

## 事件流

### 1.1 事件流和两个阶段说明

目标：能够说出事件流经过的2个阶段

- 事件流指的是事件完整执行过程中的流动路径



2. 当我点击div做点击事件，那么div上一级有body，所有标签都写在body里面，body还有上一级html，Document。

3. 所以捕获阶段就是，比如我要找div，就是从Document->html->body->div

4. 冒泡阶段就是从div到Document

- 说明：假设页面里有个div，当触发事件时，会经历两个阶段，分别是捕获阶段、冒泡阶段
- 简单来说：捕获阶段是 从父到子 冒泡阶段是从子到父

```

.father{
  width: 500px;
  height: 500px;
  background-color: #pink;
}
.son{
  width: 200px;
  height: 200px;
  background-color: #purple;
}

<div class="father">
  <div class="son"></div>
</div>

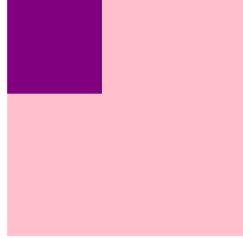
const fa = document.querySelector('.father')
const son = document.querySelector('.son')

fa.addEventListener('click',function(){
  alert('father 标签执行') //使用console.log不太合适
})

son.addEventListener('click',function(){
  alert('son 标签执行')
})

document.addEventListener('click',function(){
  alert('Document对象事件第1个执行')
})

```




所以事件流的方向是son流向->father->document

## 1.3 事件冒泡

目标：能够说出事件冒泡的执行过程

事件冒泡概念：

当一个元素的事件被触发时，同样的事件将会在该元素的所有祖先元素中依次被触发。这一过程被称为事件冒泡

- 简单理解：当一个元素触发事件后，会依次向上调用所有父级元素的 同名事件（意思就是同一种事件类型，比如

```
const father = document.querySelector('.father')
const son = document.querySelector('.son')
document.addEventListener('click', function () {
  alert('我是爷爷')
})
fa.addEventListener('click', function () {
  alert('我是爸爸')
})
son.addEventListener('click', function () {
  alert('我是儿子')
})
```

- 事件冒泡是默认存在的
- L2事件监听第三个参数是 false，或者默认都是冒泡

三个都是注册的click

## 1.4 阻止冒泡

目标：能够写出阻止冒泡的代码

- 问题：因为默认就有冒泡模式的存在，所以容易导致事件影响到父级元素
- 需求：若想把事件就限制在当前元素内，就需要阻止事件冒泡
- 前提：阻止事件冒泡需要拿到事件对象
- 语法：

事件对象.stopPropagation()

- 注意：此方法可以阻断事件流动传播，不光在冒泡阶段有效，捕获阶段也有效

e.stopPropagation() //阻止事件流传播，阻止冒泡

```
const fa = document.querySelector('.father')
const son = document.querySelector('.son')

fa.addEventListener('click', function(){
  alert('father 标签执行') //使用console.log不太合适
})

son.addEventListener('click', function(e){
  alert('son 标签执行')
  e.stopPropagation() //阻止流动传播
})

document.addEventListener('click', function(){
  alert('Document对象事件第1个执行')
})
```



2. 然后你发现son事件执行之后，后面的father和Document都没有执行，这是因为被stop...截断了事件流传播

## 1.5 解绑事件

on事件方式，直接使用null覆盖偶就可以实现事件的解绑

语法：

```
// 绑定事件 这是传统的解绑方法
btn.onclick = function () {
| alert('点击了')
}
// 解绑事件
btn.onclick = null      解绑之后再次点击
                      按钮没有反应了
```

addEventListener方式，必须使用：

removeEventListener(事件类型, 事件处理函数, [获取捕获或者冒泡阶段])

```
function fn() {
  alert('点击了')
}
// 绑定事件
btn.addEventListener('click', fn)
// 解绑事件
btn.removeEventListener('click', fn)
```

## 鼠标经过事件的区别

- 鼠标经过事件：

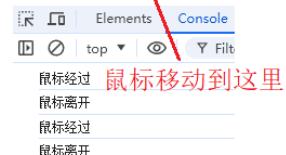
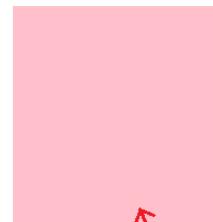
- mouseover 和 mouseout 会有冒泡效果
- mouseenter 和 mouseleave 没有冒泡效果 (推荐)

```
.dad{
  width: 400px;
  height: 400px;
  background-color: #pink;
}
```

```
<div class="dad">
  <div class="baby">
    </div>
</div>
```

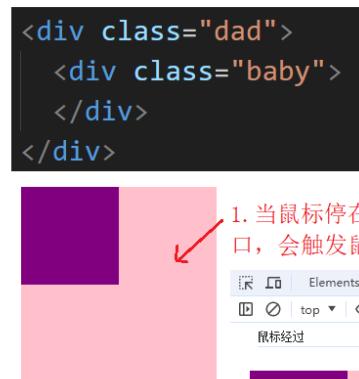
```
const dad = document.querySelector('.dad')
dad.addEventListener('mouseover',function(){
  console.log('鼠标经过')
})

dad.addEventListener('mouseout',function(){
  console.log('鼠标离开')
})
```

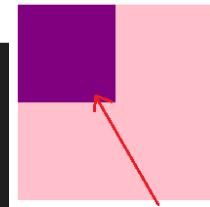
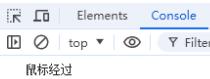


父div和子div冒泡事件问题

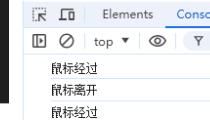
```
.dad{  
    width: 400px;  
    height: 400px;  
    background-color: #pink;  
}  
.baby{  
    width: 200px;  
    height: 200px;  
    background-color: #purple;  
}  
  
const dad = document.querySelector('.dad')  
const baby = document.querySelector('.baby')  
dad.addEventListener('mouseover',function(){  
    console.log('鼠标经过')  
})  
  
dad.addEventListener('mouseout',function(){  
    console.log('鼠标离开')  
})
```



1. 当鼠标停在dad窗口，会触发鼠标经过



2. 紧接着鼠标经过baby小窗口，出现了鼠标离开，鼠标又经过事件



这就有个难以理解的地方，`baby` 窗口没有注册 `addEvent....` 事件，怎么还是有鼠标经过事件呢？这是因为鼠标虽然移动到 `baby` 窗口了，但是父 `div` 是 `dad`, `dad` 事件会冒泡到子 `div` 上，也就是自己下一级的 `div` 上。

如果不想用冒泡，可以使用 `mouseenter`。

## 事件委托

目标：能够说出事件委托的好处

### 委托：

1. 如果班上有40个同学，买了40份快递。



2. 这时候如果快递小哥，给每一个学生都送快递，如果40个学生，小哥就要送40次，这对于快递小哥效率非常低。



3. 所以快递小哥把40份快递委托给班主任，班主任去处理快递给学生。快递小哥效率变高了。

```
<ul> 1. 我有很多个li标签
    <li>我是第1个li</li>
    <li>我是第2个li</li>
    <li>我是第3个li</li>
    <li>我是第4个li</li>
    <li>我是第5个li</li>
</ul>
```

```
const lis = document.querySelectorAll('ul li')
for (let i = 0; i < lis.length; i++) {
    lis[i].addEventListener('click', function () {
        alert('我被点击了')
    })
}
```

2. 如果我要给每个li标签实现点击事件，我就只有用for循环去遍历每个li，相当麻烦

事件委托是利用事件流的特征解决一些开发需求的知识技巧

- 优点：减少注册次数，可以提高程序性能
- 原理：事件委托其实是利用事件冒泡的特点。
  - 给父元素注册事件，当我们触发子元素的时候，会冒泡到父元素身上，从而触发父元素的事件

```
ul.addEventListener('click', function(){})
```



```
<ul>
    <li>第1个li标签</li>
    <li>第2个li标签</li>
    <li>第3个li标签</li>
    <li>第4个li标签</li>
    <li>第5个li标签</li>
</ul>
```

```
const ul = document.querySelector('ul')
ul.addEventListener('click', function(){
    console.log('ul 下 li事件触发')
})
```

直接获取ul，注册ul事件

- 第1个li标签
- 第2个li标签
- 第3个li标签
- 第4个li标签
- 第5个li标签

你发现不管点击任何一个li标签都是同一个事件

所以li冒泡冒到ul父标签身上，相当于上一节描述的ul就是老师，li就是学生。

```
const ul = document.querySelector('ul')
通过事件e.target，就能获取到当前ul下某一个子标签被点击
ul.addEventListener('click', function(e){
    e.target.style.color = 'red'
})
```

我用改变子标签颜色来演示

- 第1个li标签
  - 第2个li标签
  - 第3个li标签
  - 第4个li标签
  - 第5个li标签
- 当你点某个标签，这个标签颜色就会变

```
<ul>
    <li>第1个li标签</li>
    <li>第2个li标签</li>
    <li>第3个li标签</li>
    <li>第4个li标签</li>
    <li>第5个li标签</li>
    <p> p标签也是子标签了</p>
</ul>
```

常规情况下，我点击p标签也会触发事件

```
const ul = document.querySelector('ul')
ul.addEventListener('click', function(e){
    //<li>标签在事件包里面名字叫做LI
    if(e.target.tagName === 'LI'){
        e.target.style.color = 'red'
    }
})
```

但是我现在直接判断点击事件类型，只获取li标签

p标签也是子标签了

你会发现，虽然点击p标签也冒泡触发了，但是没有满足tagName判断要求，所以无法改变其颜色。点击li才触发改变颜色。

## 1. 事件委托的好处是什么？

- 减少注册次数，提高了程序性能

## 2. 事件委托是委托给了谁？父元素还是子元素？

- 父元素

## 3. 如何找到真正触发的元素？

- 事件对象.target.tagName

## 事件委托实现 tab 栏切换

```
<style>
    .tab-nav ul li{
        margin: 0 20px;
        font-size: 14px;
    }
    .tab-nav ul li a{
        text-decoration: none;
        border-bottom: 2px solid transparent;
        color: #333;
    }
    .tab{
        width: 500px;
        height: 340px;
        margin: 20px;
        border: 1px solid #e4e4e4;
    }
    .tab-nav ul{
        list-style: none;
        display: flex;
        justify-content: flex-end;
    }
    .tab-nav ul li a.active{
        border-color: #e1251b;
        color: #e1251b;
    }
    .tab-content .item.active{
        display: block;
    }
</style>



<div class="tab-nav">
        <h3>每日特价</h3>
        <ul>
            <li><a class="active" href="#">精选</a></li>
            <li><a href="#">美食</a></li>
            <li><a href="#">百货</a></li>
            <li><a href="#">个护</a></li>
            <li><a href="#">预告</a></li>
        </ul>
    </div>


```

```
const ul = document.querySelector('.tab-nav ul') //获取tab-nav类标签下的ul

ul.addEventListener('click',function(e){
    console.log(e.target)//获取具体事件
})
```



5. 有没有办法只指定li标签点击之后触发li事件呢？

我们可以使用tagName来判断'A'标准，因为a标签事件触发后返回'A'

```

<div class="tab">
  <div class="tab-nav">
    <h3>每日特价</h3>
    <ul>
      <li><a class="active" href="javascript:;">精选</a></li>
      <li><a href="javascript:;">美食</a></li>
      <li><a href="javascript:;">百货</a></li>
      <li><a href="javascript:;">个护</a></li>
      <li><a href="javascript:;">预告</a></li>
    </ul>
  </div>
</div>
const ul = document.querySelector('.tab-nav ul') //获取tab-nav类标签下的ul

ul.addEventListener('click',function(e){

  if(e.target.tagName === 'A') //a标签tagName是'A'
  {
    console.log('选择A')
  }
})

```

增加a标签鼠标移动到a  
标签变成‘手掌’属性

```

ul.addEventListener('click',function(e){

  if(e.target.tagName === 'A') //a标签tagName是'A'
  {
    //排他思想，现移除原来选择的a标签active属性
    document.querySelector('.tab-nav .active').classList.remove('active')

    //this指向ul，不能用this，所以只有用e
    e.target.classList.add('active')
  }
})

```

TAB选项栏标志根据鼠标点击而变动



如何让JS知道我当前选择的是哪一个a标签呢？

事件对象 `e.target.dataset.id` //获取标签自定义的 id 值，用来判断哪一个标签被触发

```

<ul>
  <li><a class="active" href="javascript:;" data-id="0">精选</a></li>
  <li><a href="javascript:;" data-id="1">美食</a></li>
  <li><a href="javascript:;" data-id="2">百货</a></li>
  <li><a href="javascript:;" data-id="3">个护</a></li>
  <li><a href="javascript:;" data-id="4">预告</a></li>
</ul>

```

```



每日特价  
精选 美食 百货 个护 预售  
每个标签对应不同的ID →



```

if(e.target.tagName === 'A') //a标签tagName是'A'
{
    //排他思想，现移除原来选择的a标签active属性
    document.querySelector('.tab-nav .active').classList.remove('active')

    //this指向ul, 不能用this, 所以只有用e
    e.target.classList.add('active')
    const id = e.target.dataset.id //获取点击的a标签ID, 判断哪一个标签被触发
    //得到id之后判断哪一个a标签被触发, 去执行对应的程序
}

```


```

阻止冒泡(比如阻止后面的程序运行)

## 阻止冒泡

我们某些情况下需要阻止默认行为的发生，比如 阻止链接的跳转，表单域跳转

- 语法：

`e.preventDefault()`

```

<form action="http://www.baidu.com">
    <input type="submit" value="提交">
</form>
<script>
    const form = document.querySelector('form')
    form.addEventListener('click', function (e) {
        // 阻止表单默认提交行为
        e.preventDefault()
    })

```

- 比如有一个婚恋网站，如果没有输入正确的信息



- 这时候我点击注册，就不允许换页，必须停止在当前页，等到信息输入正确。这就是阻止冒泡

事件对象 e.preventDefault() //阻止当前事件行为发生

```
<form action="http://www.baidu.com">
| <input type="submit" value="免费注册">
</form>
```



```
const form = document.querySelector('form')
form.addEventListener('submit', function(e){
    e.preventDefault() //阻止默认行为
})
```

在对应的触发事件中加入阻止冒泡，submit事件就不会发生。

页面加载事件和页面滚动事件

## 页面加载事件

- 加载外部资源（如图片、外联CSS和JavaScript等）加载完毕时触发的事件
- 为什么要学？
  - 有些时候需要等页面资源全部处理完了做一些事情
  - 老代码喜欢把 script 写在 head 中，这时候直接找 dom 元素找不到
- 事件名：load
- 监听页面所有资源加载完毕：
  - 给 window 添加 load 事件

```
// 页面加载事件
window.addEventListener('load', function () {
    // 执行的操作
})
```

<button>点击</button> 这是模拟的页面加载完成之后，点击按钮才有效果

```
//load:等待网络下载的图片数据等资源传输完毕，执行function内部的内容
window.addEventListener('load',function(){
    const btn = document.querySelector('button')
    btn.addEventListener('click',function(){
        alert('点击事件')
    })
})
```



```
//图片加载完毕之后采取执行里面的代码
Image.addEventListener('load',function(){
})
```

## 页面加载事件

- 当初始的 HTML 文档被完全加载和解析完成之后，`DOMContentLoaded` 事件被触发，而无需等待样式表、图像等完全加载
  - 事件名：`DOMContentLoaded`
  - 监听页面DOM加载完毕：
    - 给 `document` 添加 `DOMContentLoaded` 事件

```
document.addEventListener('DOMContentLoaded', function () {  
    // 执行的操作  
})
```

## 页面滚动事件

- 滚动条在滚动的时候持续触发的事件
  - 为什么要学?
    - 很多网页需要检测用户把页面滚动到某个区域后做一些处理, 比如固定导航栏, 比如返回顶部
  - 事件名: scroll
  - 监听整个页面滚动:

```
<style>
  *{
    margin: 0;
    padding: 0;
  }
  body{ height: 3000px;
}
</style>
```

```
//scroll: 页面滚动事件  
window.addEventListener('scroll',function(){  
    console.log('我滚了')  
})
```

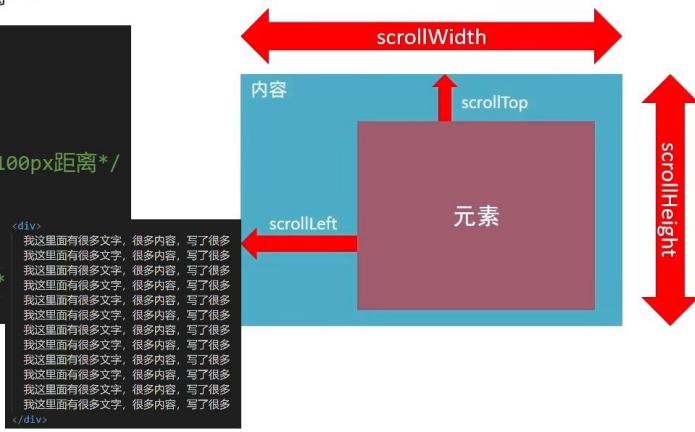


- 使用场景：
    - > 我们想要页面滚动一段距离，比如100px，就让某些元素显示出来
    - > 显示隐藏，那我们怎么知道，页面滚动了100像素呢？
    - 就可以使用scroll来检测滚动的距离~~~

- scrollLeft和scrollTop（属性）

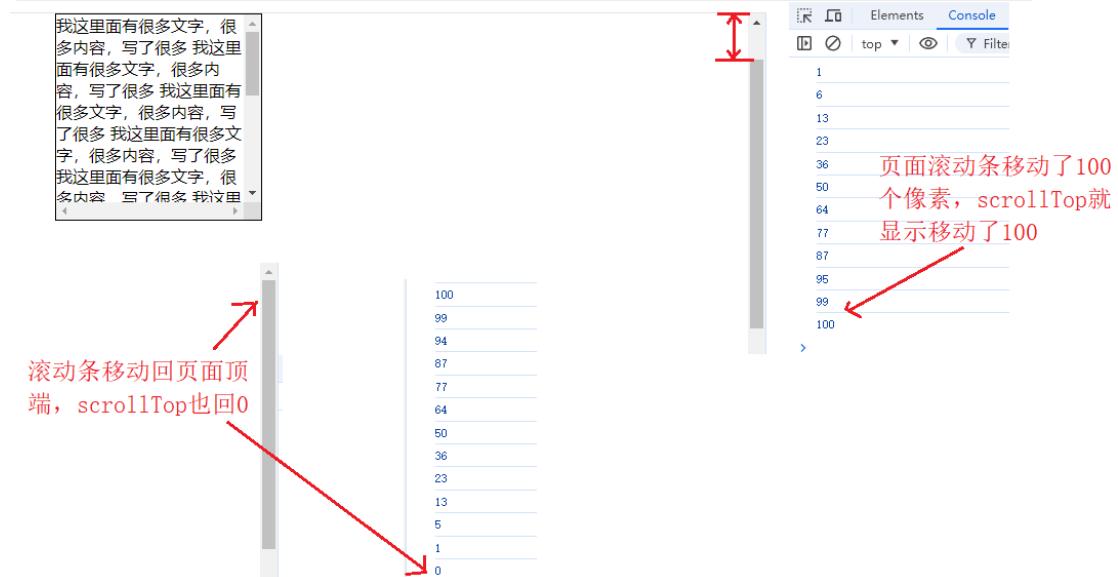
- 获取被卷去的大小
  - 获取元素内容往左、往上滚出去看不到的距离
  - 这两个值是可读写的

```
body{  
    height: 3000px;  
}  
  
div{  
    margin: 100px; /*div框离浏览器100px距离*/  
    width: 200px;  
    height: 200px;  
    border: 1px solid #000;  
    overflow: scroll; /*加入滚动条*  
}  
    
```



```
document.documentElement.scrollTop //获取页面竖向滚动条滚动得距离
```

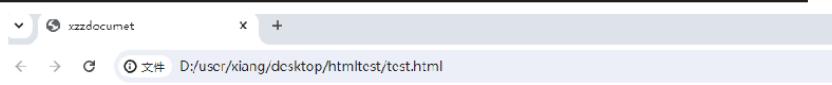
```
//scroll: 页面滚动事件  
window.addEventListener('scroll',function(){  
    //我想知道页面到底滚动了多少像素，就是被滚去了多少scrollTop  
  
    console.log(document.documentElement.scrollTop)  
})
```



## 页面尺寸事件

```
document.documentElement.clientWidth //获取浏览器当前宽度尺寸
```

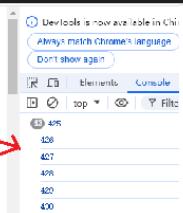
```
//resize:窗口尺寸变化事件
window.addEventListener('resize',function(e){
    console.log('窗口尺寸变化')
})
```



浏览器窗口拖大拖小，  
尺寸在变化

```
//resize:窗口尺寸变化事件
window.addEventListener('resize',function(e){
    let w = document.documentElement.clientWidth //检测屏幕尺寸
    console.log(w)
})
```

当浏览器左  
右拉伸得时  
候，宽度变  
了

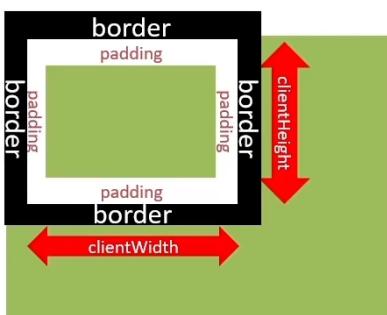


因为是检测浏  
览器当前宽  
度，所以上下  
拉伸浏  
览器，浏  
览器宽度不  
变

- 获取宽高：

- 获取元素的可见部分宽高（不包含边框，margin，滚动条等）

- clientWidth和clientHeight

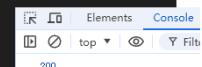


```
div{
    width: 200px;
    height: 200px;
    border: 1px solid #000;
}

<body>
    <div></div>
```

不管浏览器尺寸怎么变，div窗  
口得大小尺寸是不变得。

```
const div = document.querySelector('div')
console.log(div.clientWidth) //获取div窗口大小
//resize:窗口尺寸变化事件
window.addEventListener('resize',function(e){
    let w = document.documentElement.clientWidth //检测屏幕尺寸
    console.log(w)
})
```

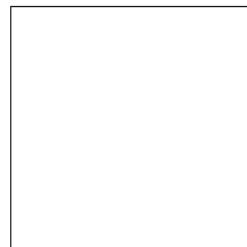


```

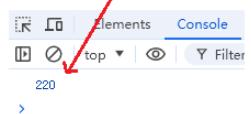
div{
    width: 200px;
    height: 200px;
    border: 1px solid #000;
    padding: 10px;
}

```

如果加入padding



使用client去获取div  
窗口尺寸，也会包含  
padding进去



```

const div = document.querySelector('div')
console.log(div.clientWidth) //获取div窗口大小

```

如下案例：

```

(function flexible(window, document) { ...
})(window, document)

```

外部传入参数到window和document

↓

```

(function flexible(window, document) {
    var docEl = document.documentElement ← 获取html
    var dpr = window.devicePixelRatio || 1 ← 获取当前dpr，确定这
                                                是PC端还是移动端
}

```

```

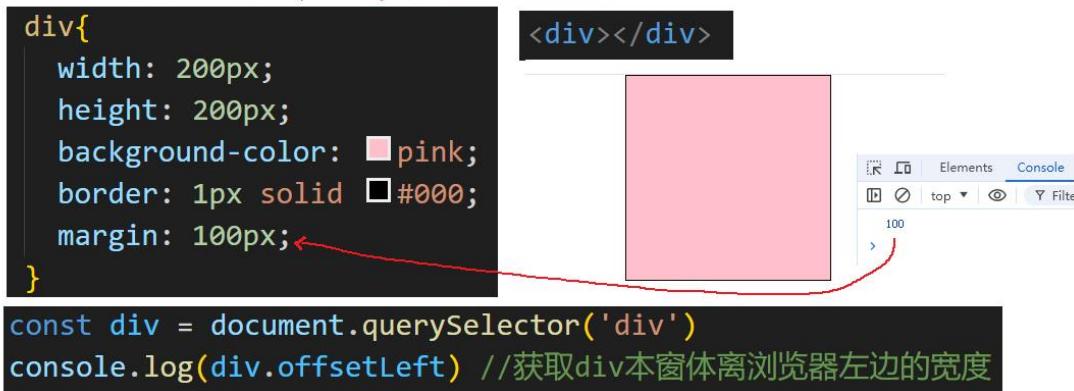
// adjust body font size
function setBodyFontSize() {
    if (document.body) { 1. 如果获取到body
        document.body.style.fontSize = (12 * dpr) + 'px'
    }
    else { 2. 我就将body里面文字大小设置成12
        document.addEventListener('DOMContentLoaded', setBodyFontSize)
    }
}
3. 如果没获取到body或者body没获取完，等待DOM加载完毕
setBodyFontSize();

// set 1rem = viewWidth / 10
function setRemUnit() {
    var rem = docEl.clientWidth / 10 ← 4. 获取整个窗口的宽度
    docEl.style.fontSize = rem + 'px' ← (不包含边框和滚动条)
}
5. 设置html文字大小

```

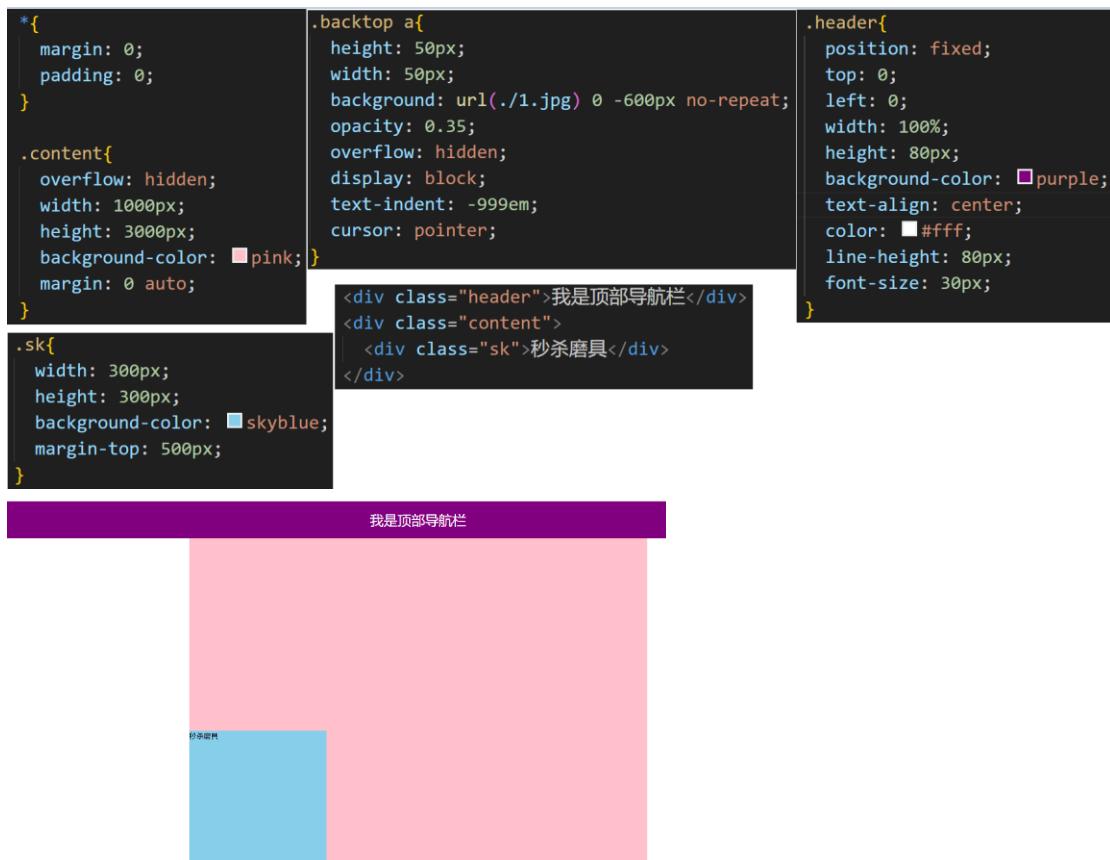
## 元素尺寸与位置-尺寸

- 获取宽高：
  - 获取元素的自身宽高、包含元素自身设置的宽高、padding、border
  - offsetWidth和offsetHeight
  - 获取出来的是数值,方便计算
  - 注意：获取的是可视宽高,如果盒子是隐藏的,获取的结果是0
- 获取位置：
  - 获取元素距离自己定位父级元素的左、上距离
  - offsetLeft和offsetTop 注意是只读属性



```
div{  
    width: 200px;  
    height: 200px;  
    background-color: #pink;  
    border: 1px solid #000;  
    margin: 100px;  
}  
  
const div = document.querySelector('div')  
console.log(div.offsetLeft) //获取div本窗体离浏览器左边的宽度
```

页面向下滚动之后探出顶部导航栏案例



```
*{  
    margin: 0;  
    padding: 0;  
}  
  
.content{  
    overflow: hidden;  
    width: 1000px;  
    height: 3000px;  
    background-color: #pink;  
    margin: 0 auto;  
}  
  
.sk{  
    width: 300px;  
    height: 300px;  
    background-color: #skyblue;  
    margin-top: 500px;  
}  
  
.header{  
    position: fixed;  
    top: 0;  
    left: 0;  
    width: 100%;  
    height: 80px;  
    background-color: #purple;  
    text-align: center;  
    color: #fff;  
    line-height: 80px;  
    font-size: 30px;  
}  
  
<div class="header">我是顶部导航栏</div>  
<div class="content">  
    <div class="sk">秒杀磨具</div>  
</div>
```

```

.header{
  position: fixed;
  top: 0;          完全显示导航栏
  left: 0;
  width: 100%;
  height: 80px;
  background-color: #purple;
  text-align: center;
  color: #ffff;
  line-height: 80px;
  font-size: 30px;
}

const sk = document.querySelector('.sk') //获取秒杀磨具div框标签
const header = document.querySelector('.header') //获取顶部导航栏标签
window.addEventListener('scroll',function(){
  //当页面滚动到秒杀磨具div位置的时候，弹出导航栏
  const n = document.documentElement.scrollTop
  if(n >= sk.offsetTop) //当页面滚动像素>当前秒杀磨具离浏览器顶部距离的像素时
  {
    header.style.top = 0 //弹出导航栏
  }
})

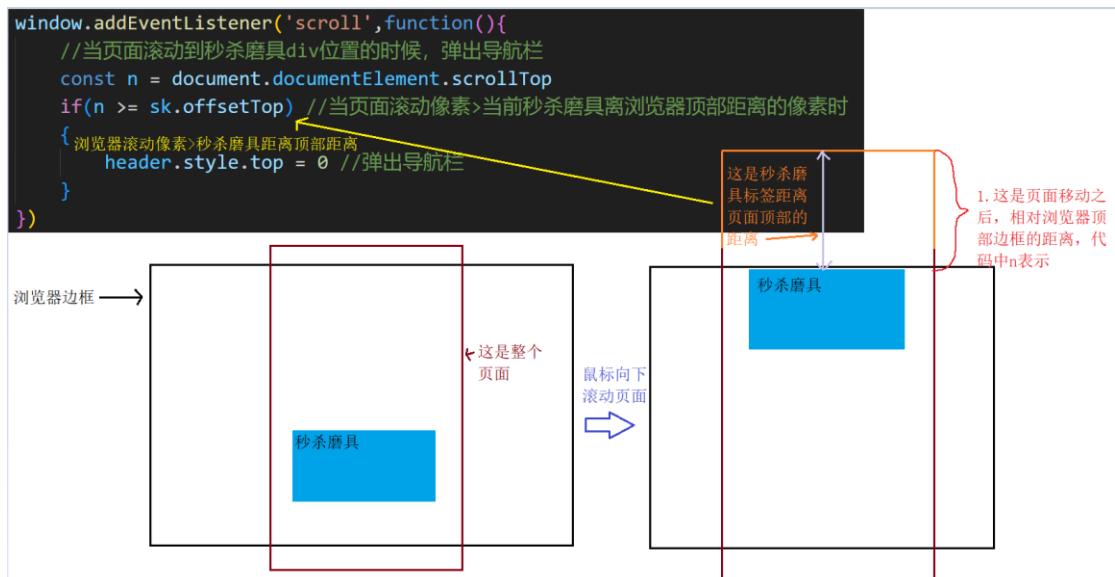
```

完全显示导航栏

我们等着JS触发，将导航栏移动回来

默认没有导航栏

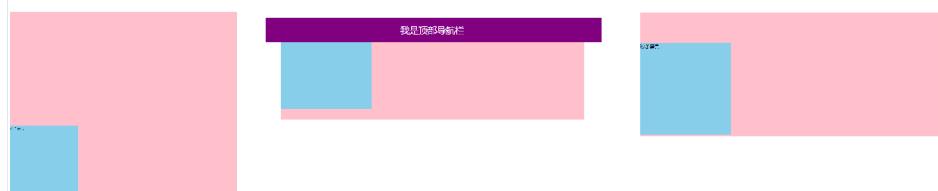
浏览器滚动到秒杀磨具框  
指定位置，弹出导航栏



```

const sk = document.querySelector('.sk') //获取秒杀磨具div框标签
const header = document.querySelector('.header') //获取顶部导航栏标签
window.addEventListener('scroll',function(){
  //当页面滚动到秒杀磨具div位置的时候，弹出导航栏
  const n = document.documentElement.scrollTop
  if(n >= sk.offsetTop) //当页面滚动像素>当前秒杀磨具离浏览器顶部距离的像素时
  {
    header.style.top = 0 //弹出导航栏
  }
  else{
    header.style.top = '-80px' //JS修改样式也要带单位，用字符串写
  }
})

```



## 日期对象使用

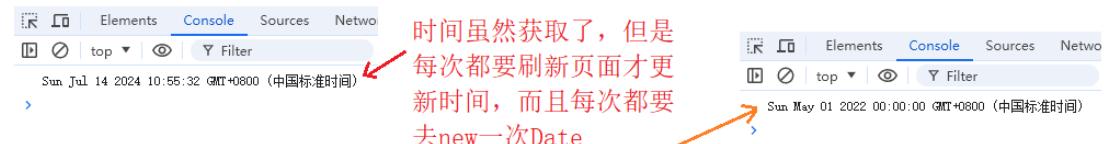
### 1.1 实例化

目标：能够实例化日期对象

- 在代码中发现了 new 关键字时，一般将这个操作称为**实例化**
- 创建一个时间对象并获取时间
  - 获得当前时间

```
const date = new Date()
```

```
const date = new Date() // 创建时间对象  
console.log(date)
```



```
const date = new Date('2022-5-1') // 设置指定时间返回  
console.log(date)
```

### 2.2 日期对象方法

目标：能够使用日期对象中的方法写出常见日期

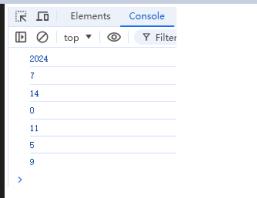
使用场景：因为日期对象返回的数据我们不能直接使用，所以需要转换为实际开发中常用的格式

方法	作用	说明
getFullYear()	获得年份	获取四位年份
getMonth()	获得月份	取值为 0 ~ 11
getDate()	获取月份中的每一天	不同月份取值也不相同
getDay()	获取星期	取值为 0 ~ 6
getHours()	获取小时	取值为 0 ~ 23
getMinutes()	获取分钟	取值为 0 ~ 59
getSeconds()	获取秒	取值为 0 ~ 59

```

const date = new Date()
console.log(date.getFullYear()) //获得时间年
console.log(date.getMonth()+1) //获得时间月
console.log(date.getDate()) //获得日
console.log(date.getDay()) //获得星期(注意星期天是0, 所以星期取值0~6)
console.log(date.getHours()) //获得小时
console.log(date.getMinutes()) //获得分钟
console.log(date.getSeconds()) //获得秒

```



## 案例 页面显示时间

需求：将当前时间以：YYYY-MM-DD HH:mm 形式显示在页面 2008-08-08 08:08

分析：

- ①：调用日期对象方法进行转换
- ②：记得数字要补0

```



```

今天是:2024年7月14日 11时13分8秒

## 时间戳

### 2.3 时间戳

目标：能够获得当前时间戳

- **使用场景：**如果计算倒计时效果，前面方法无法直接计算，需要借助于时间戳完成
- **什么是时间戳：**

➤ 是指1970年01月01日00时00分00秒起至现在的毫秒数，它是一种特殊的计量时间的方式

- **算法：**

- 将来的时间戳 - 现在的时间戳 = 剩余时间毫秒数
- 剩余时间毫秒数 转换为 剩余时间的 年月日时分秒 就是 倒计时时间
- 比如 将来时间戳 2000ms - 现在时间戳 1000ms = 1000ms
- 1000ms 转换为就是 0小时0分1秒

#### 三种方式获取时间戳：

1. 使用 `getTime()` 方法



```

const date = new Date()
console.log(date.getTime()) //获取时间戳
console.log(+new Date()) //获取时间戳第2种方式 +new Date()
console.log(Date.now()) //获取时间戳第3种方式 Date.now()

```

### 三种方式获取时间戳：

#### 1. 使用 `getTime()` 方法

```
const date = new Date()
console.log(date.getTime())

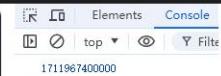
// 我要根据日期 Day() 0 ~ 6 返回的是 星期一
const arr = ['星期天', '星期一', '星期二', '星期三', '星期四', '星期五']
// const date = new Date()
console.log(arr[new Date().getDay()])
```

#### 2. 简写 `+new Date()`

- 无需实例化

```
//设置未来时间或者以前时间，返回时间戳
console.log(+new Date('2024-4-1 18:30:00'))
```

```
console.log(+new Date())
```



#### 3. 使用 `Date.now()` 如果想用未来的时间转换成时间戳做倒计时，第3种方式不合适

- 无需实例化
- 但是只能得到当前的时间戳，而前面两种可以返回指定时间的时间戳

### 案例 毕业倒计时效果

需求：计算到下课还有多少时间

分析：

- ①：用将来时间减去现在时间就是剩余的时间
- ②：核心：使用将来的时间戳减去现在的时间戳
- ③：把剩余的时间转换为 天 时 分 秒



注意：

1. 通过时间戳得到是毫秒，需要转换为秒在计算
2. 转换公式：

- `d = parseInt(总秒数 / 60/60 /24); // 计算天数`
- `h = parseInt(总秒数 / 60/60 %24) // 计算小时`
- `m = parseInt(总秒数 /60 %60 ); // 计算分钟`
- `s = parseInt(总秒数%60); // 计算当前秒数`

```
div{
  width: 300px;
  height: 40px;
  border: 1px solid pink;
  text-align: center;
  line-height: 40px;
}
```

```
<div class="countdown">
  <p class="next">今天是2022年2月22日</p>
  <p class="title">下班倒计时</p>
  <p class="clock">
    <span id="hour">00</span>
    <i>:</i>
    <span id="minutes">25</span>
    <i>:</i>
    <span id="second">20</span>
  </p>
  <p class="tips">18:30:00下课</p>
</div>
```

```

const now = +new Date() //得到当前时间戳
const last = +new Date('2024-10-1 18:30:00') //设置未来指定时间

const count = (last - now) /1000 //未来的时间-当前时间，转换成秒数

let h = parseInt(count / 60 / 60 % 24) //计算小时
h = h < 10 ? '0' + h : h //补0
let m = parseInt(count / 60 % 60) //计算分
m = m < 10 ? '0' + m : m //补0
let s = parseInt(count % 60) //计算当前秒
s = s < 10 ? '0' + s : s //补0

const hour = document.querySelector('#hour')
const min = document.querySelector('#minutes')
const scond = document.querySelector('#scond')

hour.innerHTML = h
min.innerHTML = m
scond.innerHTML = s

```

今天是2022年2月22日  
下班倒计时  
02:41:11  
18:30:00下课

今天是2022年2月22日  
下班倒计时  
02:41:04  
18:30:00下课

今天是2022年2月22日  
下班倒计时  
02:40:55  
18:30:00下课

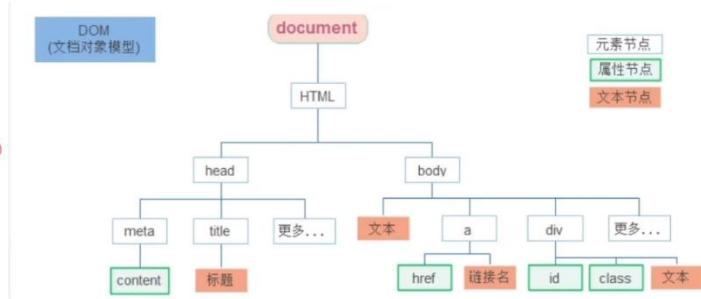
倒计时

## DOM 节点操作(对标签进行增删改查)

### DOM节点

目标：能说出DOM节点的类型

- DOM节点
  - DOM树里每一个内容都称之为节点
- 节点类型
  - **元素节点** 主要是操作元素节点(标签节点)
    - 所有的标签 比如 body、div
    - html 是根节点
  - 属性节点
    - 所有的属性 比如 href
  - 文本节点
    - 所有的文本
  - 其他



### 查找节点

目标：能够具备根据节点关系查找目标节点的能力

- 关闭二维码案例：
 

点击关闭按钮，关闭的是二维码的盒子，还要获取erweima盒子
- 思考：
  - 关闭按钮 和 erweima 是什么关系呢？
  - 父子关系
  - 所以，我们完全可以这样做：
  - 点击关闭按钮，直接关闭它的爸爸，就无需获取erweima元素了
- 节点关系：针对的找亲戚返回的都是对象



- 父节点
- 子节点
- 兄弟节点

子元素.parentNode //返回最近一级的父节点，找不到返回 null

The screenshot shows a browser developer tools console with two examples. The first example shows a simple structure: <div class="dad"><div class="baby">X</div></div>. It demonstrates that baby.parentNode returns the parent node ('dad'). The second example shows a more complex structure with multiple levels of parents. It demonstrates that baby.parentNode.parentNode returns the parent of the parent ('yeve'), and baby.parentNode.parentNode.parentNode returns the grandparent ('box'). The third example shows a click event listener on a child element ('box1') that retrieves its parent node ('box'). The fourth example shows a click event listener on a child element ('box1') that retrieves its parent node ('box'). A note indicates that this was previously done using two document.querySelectorAll calls.

```
<div class="dad"><div class="baby">X</div></div>
const baby = document.querySelector('.baby') //获取子节点
console.log(baby) //返回子节点对象
console.log(baby.parentNode) //返回父节点对象

<div class="yeve"><div class="dad">加入节点
  <div class="baby">X</div>
</div>
</div>
const baby = document.querySelector('.baby') //获取子节点
console.log(baby) //返回子节点对象
console.log(baby.parentNode) //返回父节点对象
console.log(baby.parentNode.parentNode) //获取父节点之上爷爷节点

<div class="box">
  我是广告
  <div class="box1">X</div>
</div>
const box1 = document.querySelector('.box1')
const box = document.querySelector('.box')

const box1 = document.querySelector('.box1')

box1.addEventListener('click',function(){
  this.parentNode.style.display = 'none'
}) this就是获取box1
  然后选取box1的父节点box
```

parentNode 应用案例如下：

The screenshot shows a practical application of parentNode. It includes CSS styles for a 'box' class, three separate 'box' div elements containing text, and JavaScript code that loops through all 'box1' elements, adds a click event listener to each, and sets its parent node's style to 'display: none' when clicked. A note explains that this approach is efficient because it uses the parentNode property to target the specific parent node for each click event. The bottom part of the screenshot shows three pink boxes labeled '我是广告1', '我是广告2', and '我是广告3'. Clicking the 'X' button on any box successfully closes it.

```
<style>
  .box{
    position: relative;
    width: 100px;
    height: 200px;
    background-color: pink;
    margin: 10px auto;
    text-align: center;
    font-size: 50px;
    line-height: 200px;
    font-weight: 700;
  }
  .box1{
    position: absolute;
    right: 20px;
    top: 10px;
    width: 20px;
    height: 20px;
    background-color: skyblue;
    text-align: center;
    line-height: 20px;
    font-size: 16px;
    cursor: pointer;
  }
</style>
<div class="box">
  我是广告
  <div class="box1">X</div>
</div>
<div class="box">
  我是广告
  <div class="box1">X</div>
</div>
<div class="box">
  我是广告
  <div class="box1">X</div>
</div>
const closeBtn = document.querySelectorAll('.box1') //获取三个box1属性的div
for(let i = 0; i < closeBtn.length ; i++){
  closeBtn[i].addEventListener('click', function(){
    this.parentNode.style.display = 'none'
  })
}

我是广告1
我是广告1
我是广告2
我是广告2
我是广告3
我是广告3
我是广告3
```

元素.previousElementSibling //获取上一个兄弟标签

元素.nextElementSibling //获取下一个兄弟标签

● 子节点查找：

➢ childNodes

✓ 获得所有子节点、包括文本节点（空格、换行）、注释节点等

➢ children 属性（重点）

✓ 仅获得所有元素节点

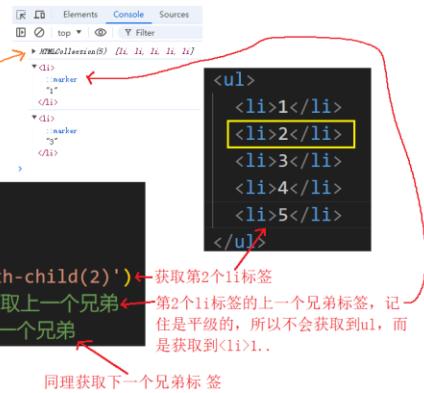
✓ 返回的还是一个伪数组

父元素.children

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ul>
```

```
const ul = document.querySelector('ul')
console.log(ul.children) // 获取ul标签子节点内容
```

```
const li2 = document.querySelector('ul li:nth-child(2)') // 获取第2个li标签
console.log(li2.previousElementSibling) // 获取上一个兄弟
console.log(li2.nextElementSibling) // 获取下一个兄弟
```



## 增加节点

### 2. 追加节点

- 要想在界面看到，还得插入到某个父元素中
- 插入到父元素的最后一个子元素：

```
// 插入到这个父元素的最后
父元素.appendChild(要插入的元素)
```

```
const ul = document.querySelector('ul')
const li = document.createElement('li')
li.innerHTML = '我是li' 将内容写入新建的li标签
ul.appendChild(li) 将li标签放入ul
```

```
// 1. 创建节点
const div = document.createElement('div')
// console.log(div)
document.body.appendChild(div)
```

```
<body>
  <script>...</script>
  <div></div> == $0 // 空加入了一个div到body标签
  <!-- Code injected by ...
  <script type="text/javascript">
```

```
const ul = document.querySelector('ul')
const li = document.createElement('li') // 创建一个li标签
ul.appendChild(li) // 把li标签赋值到ul标签下
```

```
• 我是li
• 我是老大
```

- 插入到父元素中某个子元素的前面

```
// 插入到某个子元素的前面
父元素.insertBefore(要插入的元素, 在哪个元素前面)
```

```
const ul = document.querySelector('ul')
const li = document.createElement('li')
li.innerHTML = '我是li'
```

```
• 我是li
• 我是老大
```

```
ul.insertBefore(li, ul.children[0]) // 我加的li标签永远在ul
                                         标签第0个元素的前面
```

## 2.3 增加节点

- 特殊情况下，我们新增节点，按照如下操作：
  - 复制一个原有的节点
  - 把复制的节点放入到指定的元素内部
- 克隆节点



```
// 克隆一个已有的元素节点  
元素.cloneNode(布尔值)
```

cloneNode会克隆出一个跟原标签一样的元素，括号内传入布尔值

- 若为true，则代表克隆时会包含后代节点一起克隆 表示节点子孙后代的节点都会被复制
- 若为false，则代表克隆时不包含后代节点 只复制本身这个节点
- 默认为false

```
//const li1 = document.querySelector('ul li:first-child') //选择ul第1个节点  
const ul = document.querySelector('ul')  
  
// 克隆节点 元素.cloneNode(true)  
//children[0]意思是选择ul下面第1个节点，也就是li，然后克隆  
const li1 = ul.children[0].cloneNode(true) //子孙节点都被复制  
ul.appendChild(li1) //将复制的li标签节点加在ul下面li标签最后位置
```

```
<ul>  
  <li>11111</li>  
  <li>22222</li>  
  <li>33333</li>  
  <li>44444</li>  
  <li>54444</li>  
  <li>11111</li>  
</ul>
```

```
const li1 = [ul.children[0].cloneNode()] //这种没有true就是只克隆当前标签，其余都不管  
ul.appendChild(li1) //将复制的li标签节点加在ul下面li标签最后位置
```

所以大部分我们都是用了true的

### 2.4 删除节点

目标：能够具备根据需求删除节点的能力

- 若一个节点在页面中已不需要时，可以删除它
- 在 JavaScript 原生DOM操作中，要删除元素必须通过父元素删除
- 语法

```
父元素.removeChild(要删除的元素)
```

```
<ul>  
  <li>11111</li>  
  <li>22222</li>  
  <li>33333</li>  
  <li>44444</li>  
  <li>54444</li>  
</ul>
```

```
const ul = document.querySelector('ul')  
ul.removeChild(ul.children[0]) //删除父节点下第1个li节点
```

完整的li标签  
11111  
22222  
33333  
44444  
54444

删除第1个li  
22222  
33333  
44444  
54444

```
const ul = document.querySelector('ul')  
ul.removeChild(ul.children[3]) //删除父节点下第3个li节点
```

## M 端事件(就是移动端事件，比如手机)

### 3. M端事件

目标：了解M端常见的事件

移动端也有自己独特的地方。比如**触屏事件 touch**（也称触摸事件），Android 和 IOS 都有。

- **触屏事件 touch**（也称触摸事件），Android 和 IOS 都有。
- touch 对象代表一个触摸点。触摸点可能是一根手指，也可能是一根触摸笔。触屏事件可响应用户手指（或触控笔）对屏幕或者触控板操作。
- 常见的触屏事件如下：

触屏touch事件	说明
touchstart	手指触摸到一个 DOM 元素时触发
touchmove	手指在一个 DOM 元素上滑动时触发
touchend	手指从一个 DOM 元素上移开时触发

**touchstart //触摸事件**

**touchend //手指离开触摸屏**

**touchmove//手指按着触摸屏滑动**



```
div{
  width: 300px;
  height: 300px;
  background-color: #pink;
}

<div></div>
const div = document.querySelector('div')
div.addEventListener('touchstart',function(){
  //touch触摸事件
  console.log('开始触摸事件')
})

div.addEventListener('touchend',function(){
  //touchstart离开事件
  console.log('离开触摸')
})

div.addEventListener('touchmove',function(){
  //touchmove, 一直摸着
  console.log('摸着移动')
})
```

1. 选择移动端模式

2. 点击一下再松开

3. 点击，按着一直移动，最后再松开

## swiper 插件使用

我们知道用 M 端方法去做轮播图比较麻烦，所以我们使用别人做好的 swiper 插件来做轮播图。比如 vue 框架轮播图也是用 swiper 来做。

进入 <https://www.swiper.com.cn/index.html> 中文网站

The screenshot shows the official Swiper Chinese documentation website. It highlights the download section where the latest version (11.0.3) is available for direct download. A red arrow points to the download link. Below this, a file browser window shows the contents of the downloaded zip file, including various demo HTML files and CSS/JS files. Another red arrow points to the '010-default.html' file. The page then guides the user through the setup process:

1. 直接下载最新的swiper
2. 先在demo里面找到你想要的功能例程，运行看看效果。比如我选default.html
3. 是一个拖滑的窗口
4. 然后我们打开它具体实现的html文件
5. 将官方的这两个文件拷贝进自己工程
6. 在自己工程中加入css，然后将官方例程的CSS实现拷贝过来
7. 自己例程根目录下就有这个js和css库文件
8. 注意将官方的html实现放在自己定义的div中，比如我定义的box框
9. 导入官方方案例的JS库
10. 导入自己的js实现
11. 在自己的js中启动swiper
12. 效果有了，但是感觉纵向好窄

Code snippets from the documentation and examples are shown throughout the interface, illustrating the implementation steps.

**Top Left:**

```
.box{
  position: relative;
  width: 800px;
  height: 300px;    ←
  background-color: pink;
  margin: 100px auto;
}
```

**Annotations:**

1. 给包裹这个swiper插件的div加长宽尺寸。
2. 你看, swiper图框变大了
3. 但是这里没有切换指示点了吧?

**Bottom Left:**

```
.swiper {
  overflow: hidden;
  width: 100%;
  height: 100%;    ←
}
```

**Annotation:**

4. 先让swiper组件长宽不超过包裹自己div的长宽。

## swiper 组件查找官方库，学习定制轮播图方法

**Top Left:**

1. 进入swiper官方网站 2. 找到使用方法

**Top Right:**

3. 给出了使用方法

**Middle Left:**

1. 首先加载插件, 需要用到的文件有swiper-bundle.min.js和swiper-bundle.min.css文件, 不同Swiper版本用到的文件名略有不同。可下载Swiper文件或使用CDN。

```
<!DOCTYPE html>
<html>
<head>
  ...
  <link rel="stylesheet" href="dist/css/swiper-bundle.min.css">
</head>
<body>
  ...
  <script src="dist/js/swiper-bundle.min.js"></script>
  ...
</body>
</html>
```

2. 添加HTML内容。Swiper7的默认容器是'.swiper', Swiper6之前是'.swiper-container'。

```
<div class="swiper">
  <div class="swiper-wrapper">
    <div class="swiper-slide">Slide 1</div>
    <div class="swiper-slide">Slide 2</div>
    <div class="swiper-slide">Slide 3</div>
  </div>
  <!-- 如果需要分页器 -->
  <div class="swiper-pagination"></div>
```

**Bottom Left:**

配置选项  
Swiper(4.7)的配置选项  
找到定时切换滚动  
延时选项

**Bottom Middle:**

API 文档

组件

Autoplay (自动切换)

autoplay: {  
 delay: 3000  
};

从Swiper7开始, 官方默认类名由swiper-container变为swiper。

```
<div class="swiper-slide red-slide" data-swiper-autoplay="5000">slider2</div>
<script>
var mySwiper = new Swiper('.swiper', {
  autoplay: {
    delay: 1000, //1秒切换一次
  },
});
</script>
```

Slide 1      Slide 2      Slide 7

自动切换的功能有了

**Bottom Right:**

```
var swiper = new Swiper(".mySwiper", {
  autoplay: {
    delay: 1000, //1秒切换一次
  },
});
```

在swiper中加入图片

```
<div class="box">
  <div class="swiper mySwiper">
    <div class="swiper-wrapper">
      <div class="swiper-slide">
        <a href="">
          
        </a>
      </div>
      <div class="swiper-slide">
        <a href="">
          
        </a>
      </div>
    </div>
  </div>
</div>
```



## 学生信息表案例



### 学生信息表案例

业务模块：

- ①：点击录入按钮可以录入数据
- ②：点击删除可以删除当前的数据

### 新增学员

姓名： 年龄： 性别： 薪资： 就业城市： 北京

### 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
----	----	----	----	----	------	----

说明：

本次案例，我们尽量减少dom操作，采取操作数据的形式

增加和删除都是针对数组的操作，然后根据数组数据渲染页面

```

<body>
  <h1>新增学员</h1>
  <form class="info" autocomplete="off">
    姓名
    <input type="text" class="uname" name="uname">
    年龄
    <input type="text" class="age" name="age">
    性别
    <select name="gender" class="gender">
      <option value="男">男</option>
      <option value="女">女</option>
    </select>
    薪资
    <input type="text" class="salary" name="salary">
    就业城市
    <select name="city" class="city">
      <option value="北京">北京</option>
      <option value="上海">上海</option>
      <option value="广州">广州</option>
      <option value="深圳">深圳</option>
      <option value="曹县">曹县</option>
    </select>
    <button class="add">录入</button> <!-- 提交事件 -->
  </form>

```

**新增学员**

姓名	年龄	性别	薪资
<input type="text"/>	<input type="text"/>	<input checked="" type="checkbox"/> 男 <input type="checkbox"/> 女	<input type="text"/>
		就业城市	<input type="text" value="北京"/>
<b>录入</b>			

**就业榜**

学号	姓名	年龄	性别	薪资	就业城市	操作

```

<style>
  .info input {
    margin: 0;
    padding: 0;
  }

  h1 {
    text-align: center;
  }

  .info {
    width: 900px;
    margin: 50px auto;
    text-align: center;
  }

```

```

.info input {
  width: 80px;
  height: 27px;
  outline: none;
  border-radius: 5px;
  border: 1px solid #b8daff;
  padding-left: 5px;
  box-sizing: border-box;
  margin-right: 15px;
}

th {
  border: 1px solid #ccc;
  border-collapse: collapse;
}

tbody {
  display: table-row-group;
  vertical-align: middle;
  border-color: inherit;
}

```

```

.info button{
  width: 60px;
  height: 27px;
  background-color: #004085;
  outline: none;
  border: 0;
  color: #fff;
  cursor: pointer;
  border-radius: 5px;
}

table {
  margin: 0 auto;
  width: 800px;
  border-collapse: collapse;
  color: #004085;
  text-align: center;
}

```

```

const uname = document.querySelector('.uname') //获取输入的数据
const age = document.querySelector('.age')
const gender = document.querySelector('.gender')
const salary = document.querySelector('.salary')
const city = document.querySelector('.city')
const tbody = document.querySelector('tbody') //获取页面增加行数据，纯标签不需要加'.'

const arr = [] //声明一个空数组，对增加和删除都是对这个数组进行操作

const info = document.querySelector('.info')

```

```

info.addEventListener('submit',function(e){ //点击from表里的按钮会触发submit

    e.preventDefault() //阻止默认行为，防止页面点击录入后跳转。这样才能实现本页面增加内容

    //每次点击录入，创建新的数据对象
    const obj = {
        stuID: arr.length + 1, //每次点击录入，数组长度+1
        uname: uname.value,
        age: age.value,
        gender: gender.value,
        salary: salary.value,
        city: city.value
    }

    arr.push(obj) //将新增数据写入数组
    this.reset() //重置，点击录入之后，编辑的数据在页面输入框清空
    render() //页面增加一行数据，自定义函数

})

//页面增加一行数据
function render()
{
    for(let i = 0; i<arr.length; i++)
    {
        //生成tr
        const tr = document.createElement('tr') //生成tr标签
        tr.innerHTML=`
            <th>${arr[i].stuID}</th>
            <th>${arr[i].uname}</th>
            <th>${arr[i].age}</th>
            <th>${arr[i].gender}</th>
            <th>${arr[i].salary}</th>
            <th>${arr[i].city}</th>
            <th>
                <a href = "javascript:">删除</a>
            </th>
        `

        tbody.appendChild(tr) //向页面追加元素，父元素.appendChild(子元素)
    }
}

```

## 新增学员

姓名  年龄  性别  薪资  就业城市  录入

## 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
----	----	----	----	----	------	----

## 新增学员 内容录入之后

姓名  年龄  性别  薪资  就业城市  录入

## 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
1	名字写入	50	女	222222	广州	<a href="#">删除</a> 新增了一行

## 新增学员

姓名  年龄  性别  男  女 薪资  就业城市  北京

发现一个问题，第一次  
添加数据，没有问题

### 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
1	名字测试	50	男	123	广州	<a href="#">删除</a>

## 新增学员

姓名  年龄  性别  男  女 薪资  就业城市  北京

### 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
1	名字测试	50	男	123	广州	<a href="#">删除</a>
1	名字测试	50	男	123	广州	<a href="#">删除</a>
2	名字测试2	2	男	22222	北京	<a href="#">删除</a>

第2次添加数据，发现  
把第1次的数据又多添  
加了一次。

```
//页面增加一行数据
function render()
{
    const arr = [] //声明一个空数组,
    for(let i = 0; i<arr.length; i++)
    {
        //生成tr
        const tr = document.createElement('tr') //生成tr标签
        tr.innerHTML= `
            <th>${arr[i].stuID}</th>
            <th>${arr[i].uname}</th>
            <th>${arr[i].age}</th>
            <th>${arr[i].gender}</th>
            <th>${arr[i].salary}</th>
            <th>${arr[i].city}</th>
            <th>
                <a href = "javascript:">删除</a>
            </th>
        `
        tbody.appendChild(tr) //向页面追加元素，父元素.appendChild(子元素)
    }
}
```

1. 第1次添加数据，将  
数据写入到数组

2. 第1次执行appendChild，没有问题

1..... 第1次添加的数据

1..... 第1次添加的数据

2..... 第2次添加的数据

3. 当第2次添加数据的时候，因为数组有了第一次  
的数据，所以第2次数据是追加在数组后面，但是  
for循环的时候，第一次的数据也存在在数组里  
面，所以直接加入了数组里面两个元素的数据

```
//页面增加一行数据
function render()
{
    //先清空tbody，把tbody之间新添的数据取消掉，
    tbody.innerHTML = '' //清空之后，for循环再次增加新数据
    for(let i = 0; i<arr.length; i++)
    {
        //生成tr
        const tr = document.createElement('tr') //生成tr标签
        tr.innerHTML= `
            <th>${arr[i].stuID}</th>
            <th>${arr[i].uname}</th>
            <th>${arr[i].age}</th>
            <th>${arr[i].gender}</th>
            <th>${arr[i].salary}</th>
            <th>${arr[i].city}</th>
            <th>
                <a href = "javascript:">删除</a>
            </th>
        `
        tbody.appendChild(tr) //向页面追加元素，父元素.appendChild(子元素)
    }
}
```

## 新增学员

姓名  年龄  性别  男  女 薪资  就业城市  北京

### 就业榜

学号	姓名	年龄	性别	薪资	就业城市	操作
1	名字测试	50	男	123	广州	<a href="#">删除</a>
2	名字测试2	2	男	22222	北京	<a href="#">删除</a>

没有出现重行现象

```

function render()
    //先清空tbody，把tbody之间新添的数据取消掉,
    tbody.innerHTML = '' //清空之后, for循环再次增加新数据
    for(let i = 0; i<arr.length; i++)
    {
        //生成tr
        const tr = document.createElement('tr') //生成tr标签
        tr.innerHTML= `
            <th>${arr[i].stuID}</th>
            <th>${arr[i].uname}</th>
            <th>${arr[i].age}</th>
            <th>${arr[i].gender}</th>
            <th>${arr[i].salary}</th>
            <th>${arr[i].city}</th>
            <th>
                <a href = "javascript:" data-id = ${i}>删除</a>
            </th>
        `

        // 1. 在给页面增加数据的时候, 每增加一行, 给这一行设置一个ID
        tbody.appendChild(tr) //向页面追加元素, 父元素.appendChild(子元素)
    }
}

```

**新增学员**

学号	姓名	年龄	性别	薪资	就业城市	操作
1	第1个数据	111	男	111	北京	删除
3	第3个数据	333	男	333	广州	删除
4	第4个数据	444	女	444	深圳	删除

1. 这是一个自定义的删除标签, 在JS中定义的

**删除操作**

```

tbody.addEventListener('click',function(e){
    if(e.target.tagName === 'A') //删除操作
    {
        //e.target.dataset.id 这是得到当前自定义id
        arr.splice(e.target.dataset.id,1) //删除数组里面对应的数据
        render() //重新渲染, 会发现数组里面删除的那个元素不会被显示出来了
    }
})

```

2. 获取当前点击的数据行的id, 然后删除数组1个元素, 这个元素就是当前行。

3. 重新渲染, 会发现数组里面删除的那个元素不会被显示出来了,

5. 我删除了第2行数据

**就业榜**

学号	姓名	年龄	性别	薪资	就业城市	操作
1	第1个数据	111	男	111	北京	删除
3	第3个数据	333	男	333	广州	删除
4	第4个数据	444	女	444	深圳	删除
4	第4个数据	444	男	444	北京	删除
5					北京	删除
6					北京	删除

6. 这个程序有个BUG, 就是我什么数据不写入, 点击录入, 也会录入空元素

**就业榜**

学号	姓名	年龄	性别	薪资	就业城市	操作
1	第1个数据	111	男	111	北京	删除
3	第3个数据	333	男	333	广州	删除
4	第4个数据	444	女	444	深圳	删除
4	第4个数据	444	男	444	北京	删除
5					北京	删除
6					北京	删除

```

const items = document.querySelectorAll('[name]') //我获取所有带name属性的数据

const arr = []//声明一个空数组, 对增加和删除都是对这个数组进行操作

const info = document.querySelector('.info')
info.addEventListener('submit',function(e){ //点击from表里的按钮会触发submit

    e.preventDefault() //阻止默认行为, 防止页面点击录入后跳转。这样才能实现本页面增加内容

    //进行表单验证, 如果不通过, 直接中断, 不需要添加数据
    for(let i = 0; i < items.length; i++)
    {
        if(items[i].value === '') //只要表单输入有一个值为空, 就不增加行数据, 直接跳出程序
        return //发现name属性里面标签数据为空, 就不进行下一步操作
    }
})

//每次点击录入, 创建新的数据对象
const obj = {
    stuID: arr.length + 1, //每次
    uname: uname.value,
    age: age.value,
    salary: salary.value,
    city: city.value
}

//将新的数据对象添加到数组里
arr.push(obj)

//将数组的数据渲染到页面上
render()

```

9. 因为有一个数据为空

**就业榜**

学号	姓名	年龄	性别	薪资	就业城市	操作
1	向测试	1111	男	1111	北京	删除

10. 所以数据录入没有执行

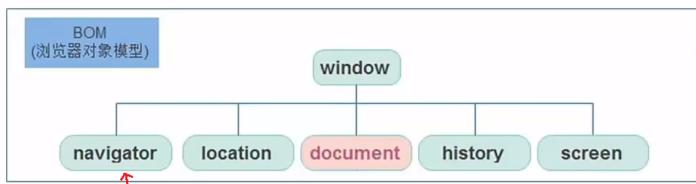
## BOM 对象和延时函数 setTimeout

### Window对象

- BOM(浏览器对象模型)
- 定时器-延时函数
- JS执行机制
- location对象
- navigator对象
- history对象

The screenshot shows a browser window with a form titled '新增学员' (Add Student) and a table titled '就业榜' (Job Ranking). The form has fields for Name, Age, Gender, Salary, and Employment City. The table lists student information with columns: 学号 (Student ID), 姓名 (Name), 年龄 (Age), 性别 (Gender), 薪资 (Salary), 就业城市 (Employment City), and 操作 (Operations). A red arrow points from the text '整个浏览器属于BOM对象' to the browser window title bar.

- BOM(Browser Object Model ) 是浏览器对象模型



```
//document.querySelector() //我们之前用的dom其实就是在window对象里面  
window.document.querySelector() //如果路径写全，就是这样写。这就说明dom在window对象里面  
console.log(document === window.document) //返回true，所以平时我们是省略了window写法
```

- window对象是一个全局对象，也可以说是JavaScript中的顶级对象
- 像document、alert()、console.log()这些都是window的属性，基本BOM的属性和方法都是window的。
- 所有通过var定义在全局作用域中的变量、函数都会变成window对象的属性和方法
- window对象下的属性和方法调用的时候可以省略window

```
function fn() {  
    console.log(11)  
}  
window.fn()  
var num = 10  
console.log(window.num)
```

像函数或者var这种都是挂在window对象下面。  
但是const, let就不是挂在window下的，它们挂在自己的作用域内

### setTimeout(回调函数, 等待毫秒数) //BOM 下的定时函数

#### 1.2 定时器-延时函数

可以用于打开浏览器弹出广告的方式应用

- JavaScript 内置的一个用来让代码延迟执行的函数，叫 setTimeout

- 语法：

```
setTimeout(回调函数, 等待的毫秒数)
```

- setTimeout 仅仅只执行一次，所以可以理解为就是把一段代码延迟执行，平时省略window

```
setTimeout(function(){  
    console.log('时间到了')  
},2000)
```

浏览器打开，运行2秒之后，执行回调。之后就再也不执行了。除非重开浏览器。

- 清除延时函数：

```
let timer = setTimeout(回调函数, 等待的毫秒数)  
clearTimeout(timer) 清除定时器
```

- 两种定时器对比：执行的次数

- 延时函数：执行一次
- 间歇函数：每隔一段时间就执行一次，除非手动清除

## 事件循环 eventloop(理论讲解)

### 经典面试题

```
console.log(1111)
setTimeout(function () {
    console.log(2222)
}, 1000)
console.log(3333)
// 问，输出的结果是什么？
```

函数内部延时，所以执行结果如下 ↴ 1111  
3333  
2222

```
console.log(1111)
setTimeout(function () {
    console.log(2222)
}, 0)
console.log(3333)
// 问，输出的结果是什么？
```

函数没延时了，怎么2222还是最后一个执行？ ↴ 1111  
3333  
2222

### 1.3 JS 执行机制

JavaScript 语言的一大特点就是单线程，也就是说，同一个时间只能做一件事。

这是因为 Javascript 这门脚本语言诞生的使命所致——JavaScript 是为处理页面中用户的交互，以及操作 DOM 而诞生的。比如我们对某个 DOM 元素进行添加和删除操作，不能同时进行。应该先进行添加，之后再删除。

单线程就意味着，所有任务需要排队，前一个任务结束，才会执行后一个任务。这样所导致的问题是：如果 JS 执行的时间过长，这样就会造成页面的渲染不连贯，导致页面渲染加载阻塞的感觉。

- 为了解决这个问题，利用多核 CPU 的计算能力，HTML5 提出 Web Worker 标准，允许 JavaScript 脚本创建多个线程。于是，JS 中出现了同步和异步。

#### 同步

前一个任务结束后再执行后一个任务，程序的执行顺序与任务的排列顺序是一致的、同步的。比如做饭的同步做法：我们要烧水煮饭，等水开了（10分钟之后），再去切菜，炒菜。

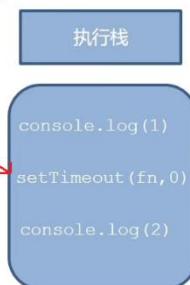
#### 异步

你在做一件事情时，因为这件事情会花费很长时间，在做这件事的同时，你还可以去处理其他事情。比如做饭的异步做法，我们在烧水的同时，利用这10分钟，去切菜，炒菜。

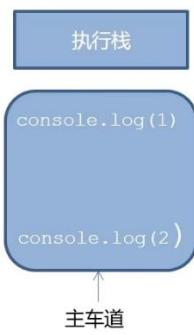
#### 同步任务

同步任务都在主线程上执行，形成一个**执行栈**。JS 的异步是通过回调函数实现的。

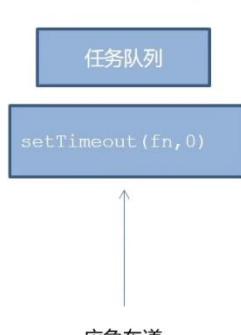
异步任务相关添加到**任务队列**中（任务队列也称为消息队列）。



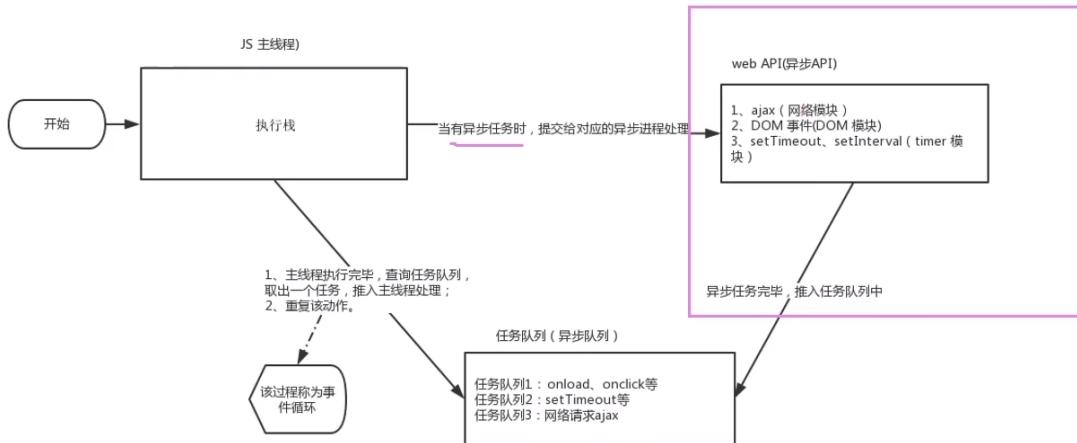
1. 先执行执行栈中的同步任务。



2. 异步任务放入任务队列中。



3. 一旦执行栈中的所有同步任务执行完毕，系统就会按次序读取任务队列中的异步任务，于是被读取的异步任务结束等待状态，进入执行栈，开始执行。



## location 对象用与 JS 中跳转页面地址使用

```
<body>
    <!--在html中引入外部JS写法-->
    <Script src="./test2.js"></Script>      <!--符号./就是引入当前目录下js文件-->
</body>
```

location.href = '<http://www.baidu.com>' JS中直接使用location的href指定跳转页面



运行代码，直接跳转到指定页面

百度一下

这种跳转页面的方式主要用于JS中，不需要点击，代码运行到location.href就跳转



### 5秒钟之后跳转的页面

需求：用户点击可以跳转，如果不点击，则5秒之后自动跳转，要求里面有秒数倒计时  
分析：

- ①：目标元素是链接
- ②：利用定时器设置数字倒计时
- ③：时间到了，自动跳转到新的页面

```
<body>
    <a href="http://www.baidu.com">支付成功<span>5</span>秒钟之后跳转首页</a>
const a = document.querySelector('a')

let num = 5 //5秒钟倒计时
let timeID = setInterval(function(){
    num --
    a.innerHTML = `支付成功<span>${num}</span>秒钟之后跳转首页` //实时显示倒计时
    if(num === 0)
    {
        clearInterval(timeID) //清除定时器
        location.href = 'http://www.baidu.com' //时间到直接跳转
    }
},1000)
```

支付成功4秒钟之后跳转首页 时间到直接跳转

百度一下

- 常用属性和方法：

- reload 方法用来刷新当前页面，传入参数 true 时表示强制刷新

```
<button>点击刷新</button>
<script>
    let btn = document.querySelector('button')
    btn.addEventListener('click', function () {
        location.reload(true)
        // 强制刷新 类似 ctrl + f5
    })
</script>
```

## navigator 对象，用于记录当前浏览器下的信息

### 1.5 navigator对象

- navigator的数据类型是对象，该对象下记录了浏览器自身的相关信息
  - 常用属性和方法：
- 通过 userAgent 检测浏览器的版本及平台

```
// 检测 userAgent (浏览器信息)
!(function () {
    const userAgent = navigator.userAgent
    // 验证是否为Android或iPhone
    const android = userAgent.match(/(Android);?[\s\S]+([\d.]+)?/)
    const iphone = userAgent.match(/(iPhone|sOS)\s([\d_]+)/)

    // 如果是Android或iPhone，则跳转至移动站点
    if (android || iphone) {
        location.href = 'http://m.itcast.cn'
    }
})()

userAgent: "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/99.0.4844.84 Safari/537.36"
```

主要是获取这个信息，就是浏览器使用的平台或者信息。

```
// 检测 userAgent (浏览器信息)
!(function () {
    const userAgent = navigator.userAgent
    // 验证是否为Android或iPhone
    const android = userAgent.match(/(Android);?[\s\S]+([\d.]+)?/)
    const iphone = userAgent.match(/(iPhone|sOS)\s([\d_]+)/)
    // 如果是Android或iPhone，则跳转至移动站点
    if (android || iphone) {
        location.href = 'http://m.itcast.cn'
    }
})()

// !(function () { })();
!function () { }()
```

## history 对象，浏览器前进后退功能

### 1.6 history对象

- history 的数据类型是对象，主要管理历史记录，该对象与浏览器地址栏的操作相对应，如前进、后退、历史记录等
- 常用属性和方法：

history对象方法	作用
back()	可以后退功能
forward()	前进功能
go(参数)	前进后退功能 参数如果是 1 前进1个页面 如果是-1 后退1个页面

```
<body>
    <button>后退</button>
    <button>前进</button>
    1. 先输入百度地址
    ↴
    baidu.com
    2. 再后退
    ↴
    3. 再点击前进，就能回到百度页面
    ↴
```

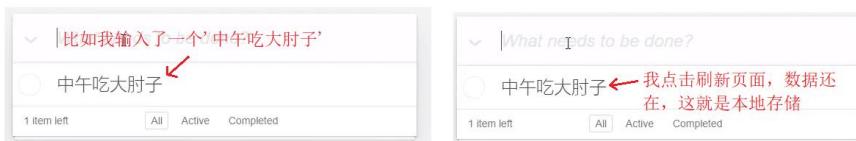
```
const back = document.querySelector('button:first-child')
const forward = back.nextElementSibling
back.addEventListener('click',function(){
    history.back() //后退一步
})

forward.addEventListener('click',function(){
    history.forward() //前进一步
})
```

## localStorage 本地存储

### 2.1 本地存储介绍

- 以前我们页面写的数据一刷新页面就没有了，是不是？
- 随着互联网的快速发展，基于网页的应用越来越普遍，同时也变的越来越复杂，为了满足各种各样的需求，会经常在本地存储大量的数据，HTML5规范提出了相关解决方案。
- 1、数据存储在用户浏览器中
- 2、设置、读取方便、甚至页面刷新不丢失数据
- 3、容量较大， sessionStorage 和 localStorage 约 5M 左右
- 常见的使用场景：
- <https://todomvc.com/examples/vanilla-es6/> 页面刷新数据不丢失



### 2.2 本地存储分类- localStorage

目标：能够使用localStorage 把数据存储的浏览器中

- 作用：可以将数据永久存储在本地(用户的电脑)，除非手动删除，否则关闭页面也会存在
- 特性：

➢ 可以多窗口（页面）共享（同一浏览器可以共享）

➢ 以键值对的形式存储使用

存储数据：

获取数据：

```
localStorage.setItem(key, value)
```

```
localStorage.getItem(key)
```

```
localStorage.setItem('uname', '数据存储')
```

Key	Value
uname	数据存储

```
localStorage.setItem('uname', '数据存储')
console.log(localStorage.getItem('uname')) // 获取存储数据
```

```
localStorage.setItem('age', 18) ← 纯数字可以  
console.log(localStorage.getItem('age')) 不加引号
```

## 2.2 本地存储分类- sessionStorage

- **特性：** sessionStorage和localStorage的区别是，  
 sessionStorage关闭浏览器数据就会消失  
 localStorage关闭浏览器数据不会消失

- 生命周期为关闭浏览器窗口
- 在同一个窗口(页面)下数据可以共享
- 以键值对的形式存储使用
- 用法跟localStorage 基本相同

所以sessionStorage用得少

```
const obj = {  
    uname: '人物名',  
    age: 18,  
    gender: '女'  
}
```

证明这种存储方式存多个数据没对

```
localStorage.setItem('xobj', obj) //一次性存入多个数据的方法  
console.log(localStorage.getItem('xobj'))
```

Key	Value
xobj	[object Object]
obj	[object Object]
uname	数据存储

发现存储的居然是对象

`JSON.stringify(对象)` //将数据对象转成 JSON 字符串

`JSON.parse(JSON 字符串)` //将 JSON 字符串转换成数据对象

## 2.3 存储复杂数据类型

- 解决：需要将复杂数据类型转换成JSON字符串,在存储到本地
- 语法：`JSON.stringify(复杂数据类型)`

```
const obj = {  
    uname: '人物名',  
    age: 18,  
    gender: '女'  
}
```

```
localStorage.setItem('xobj', JSON.stringify(obj)) //将对象转成JSON字符串, 才能存储  
console.log(localStorage.getItem('xobj')) //获取存储的数据是字符串, 因为存入的是JSON字符串
```

Key	Value
xobj	{"uname": "人物名", "age": 18, "g...}

你看, 存入的是字符串

```
localStorage.setItem('xobj', JSON.stringify(obj)) //将对象转成JSON字符串, 才能存储  
const getobj = localStorage.getItem('xobj') //获取存储的JSON字符串数据  
console.log(JSON.parse(getobj)) //将JSON字符串转换成对象打印出来
```

```
获取存储对象成功 → {uname: '人物名', age: 18, gender: '女'}
```

修改之前的学生就业统计表案例，让其增加浏览器存储功能

```
<style>
  *{
    margin: 0;
    padding: 0;
  }

  h1{
    text-align: center;
  }
  .info{
    width: 900px;
    margin: 50px auto;
    text-align: center;
  }
.info input
{
  width: 80px;
  height: 27px;
  outline: none;
  border-radius: 5px;
  border: 1px solid #b8daff;
  padding-left: 5px;
  box-sizing: border-box;
  margin-right: 15px;
}
.info button{
  width: 60px;
  height: 27px;
  background-color: #004085;
  outline: none;
  border: 0;
  color: #fff;
  cursor: pointer;
  border-radius: 5px;
}
.info table {
  margin: 0 auto;
  width: 800px;
  border-collapse: collapse;
  color: #004085;
  text-align: center;
}
.info th{
  border: 1px solid #ccc;
  border-collapse: collapse;
}

tbody{
  display: table-row-group;
  vertical-align: middle;
  border-color: inherit;
}

<body>
  <h1>新增学员</h1>
  <form class="info" autocomplete="off">
    姓名
    <input type="text" class="uname" name="uname">
    年龄
    <input type="text" class="age" name="age">
    性别
    <select name="gender" class="gender">
      <option value="男">男</option>
      <option value="女">女</option>
    </select>
    薪资
    <input type="text" class="salary" name="salary">
    就业城市
    <select name="city" class="city">
      <option value="北京">北京</option>
      <option value="上海">上海</option>
      <option value="广州">广州</option>
      <option value="深圳">深圳</option>
      <option value="曹县">曹县</option>
    </select>
    <button class="add">录入</button> <!-- 提交事件-->
  </form>
  <h1>就业榜</h1>
  <table>
    <thead>
      <tr>
        <th>学号</th>
        <th>姓名</th>
        <th>年龄</th>
        <th>性别</th>
        <th>薪资</th>
        <th>就业城市</th>
        <th>操作</th>
      </tr>
    </thead>
    <tbody></tbody>
  </table>
</body>

const uname = document.querySelector('.uname') //获取输入的数据
const age = document.querySelector('.age')
const gender = document.querySelector('.gender')
const salary = document.querySelector('.salary')
const city = document.querySelector('.city')
const tbody = document.querySelector('tbody') //获取页面增加行数据，纯标签不需要加'.'

const items = document.querySelectorAll('[name]') //我获取所有带name属性的数据

const arr = []//声明一个空数组，对增加和删除都是对这个数组进行操作

const info = document.querySelector('.info')
info.addEventListener('submit',function(e){ //点击from表里的按钮会触发submit

  e.preventDefault() //阻止默认行为，防止页面点击录入后跳转。这样才能实现本页面增加内容

  //进行表单验证，如果不通过，直接中断，不需要添加数据
  for(let i = 0; i < items.length; i++){
    if(items[i].value === '') //只要表单输入有一个值为空，就不增加行数据，直接跳出程序
      return
  }
})
```

```

//每次点击录入，创建新的数据对象
const obj = [
    stuID: arr.length + 1, //每次点击录入，数组长度+1
    uname: uname.value,
    age: age.value,
    gender: gender.value,
    salary: salary.value,
    city: city.value
}

arr.push(obj) //将新增数据写入数组
this.reset() //重置，点击录入之后，编辑的数据在页面输入框清空
render() //页面增加一行数据，自定义函数

})

//页面增加一行数据
function render()
{
    //先清空tbody，把tbody之间新添的数据取消掉，
    tbody.innerHTML = '' //清空之后，for循环再次增加新数据
    for(let i = 0; i<arr.length; i++)
    {
        //生成tr
        const tr = document.createElement('tr') //生成tr标签
        tr.innerHTML=
            <th>${arr[i].stuID}</th>
            <th>${arr[i].uname}</th>
            <th>${arr[i].age}</th>
            <th>${arr[i].gender}</th>
            <th>${arr[i].salary}</th>
            <th>${arr[i].city}</th>
            <th>
                <a href = "javascript:" data-id = ${i}>删除</a>
            </th>
        tbody.appendChild(tr) //向页面追加元素，父元素.appendChild(子元素)
    }
}

```

代码修改方式如下

```

arr = []//声明一个空数组，对增加和删除都是对这个数组进行操作
        因为要读取存储的数据放入数组  
所以数组不能有const
const info = document.querySelector('.info')
info.addEventListener('submit',function(e){ //点击form表里的按钮会触发submit

    e.preventDefault() //阻止默认行为，防止页面点击录入后跳转，这样才能实现本页面增加内容

    //进行表单验证，如果不通过，直接中断，不需要添加数据
    for(let i = 0; i < items.length; i++)
    {
        if(items[i].value === '') //只要表单输入有一个值为空，就不增加行数据，直接跳出程序
        return
    }
    //每次点击录入，创建新的数据对象
    const obj = [
        stuID: arr.length + 1, //每次点击录入，数组长度+1
        uname: uname.value,
        age: age.value,
        gender: gender.value,
        salary: salary.value,
        city: city.value
    ]
    arr.push(obj) //将新增数据写入数组
    localStorage.setItem('data',JSON.stringify(arr)) //每次增加数据，让数组向本地存入数据
    this.reset() //重置，点击录入之后，编辑的数据在页面输入框清空
    render() //页面增加一行数据，自定义函数
})

```

在页面点击添加数据的时候，  
数据就必须存入数组。

```
//删除操作
tbody.addEventListener('click',function(e){
    if(e.target.tagName === 'A') //删除操作
    {
        //e.target.dataset.id 这是得到当前自定义id
        arr.splice(e.target.dataset.id,1) //删除数组里面对应的数据
        localStorage.setItem('data',JSON.stringify(arr)) //删除的时候，要减去数组里面的数据
        render()
    }
})
arr = JSON.parse(localStorage.getItem('data')) || [] //这是每次刷新页面就要获取存储的数据
//如果存储的数据没有，就执行[]空数组
render()
```

1. 我先写入了4个数据

学号	姓名	年龄	性别	薪资	就业城市	操作
1	测试1	10	男	1111	北京	删除
2	测试2	20	男	2222	上海	删除
3	测试3	30	男	3333	北京	删除
4	测试4	40	女	4444	曹县	删除

2. 刷新页面4个数据还在

学号	姓名	年龄	性别	薪资	就业城市	操作
1	测试1	10	男	1111	北京	删除
2	测试2	20	男	2222	上海	删除
3	测试3	30	男	3333	北京	删除
4	测试4	40	女	4444	曹县	删除

3. 我再删除2个数据

学号	姓名	年龄	性别	薪资	就业城市	操作
1	测试1	10	男	1111	北京	删除
4	测试4	40	女	4444	曹县	删除

4. 刷新页面2个数据还在

学号	姓名	年龄	性别	薪资	就业城市	操作
1	测试1	10	男	1111	北京	删除
4	测试4	40	女	4444	曹县	删除

## map 和 join 方法数据字符串拼接

### 数组中map方法 迭代数组

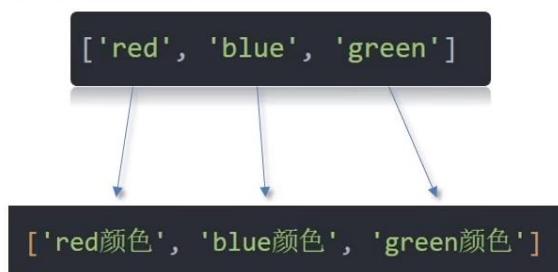
- 使用场景：

map 可以遍历数组处理数据，并且返回新的数组

```
const arr = ['red', 'blue', 'green']
const newArr = arr.map(function (ele, index) {
    console.log(ele) // 数组元素 ← 比如第1次循环，ele就是0元素，返回给新的newArr数组，第2
    console.log(index) // 数组索引号 次循环ele是1元素，以此类推，根据数组个数决定循环几次。
    return ele + '颜色'
})
console.log(newArr) // ['red颜色', 'blue颜色', 'green颜色']
```

map 也称为映射。映射是个术语，指两个元素的集之间元素相互“对应”的关系。

map重点在于有返回值，forEach没有返回值



## 数组中join方法

- 作用：

join() 方法用于把数组中的所有元素转换一个字符串

- 语法：

```
const arr = ['red颜色', 'blue颜色', 'green颜色']
console.log(arr.join('')) // red颜色blue颜色green颜色
```

```
// 小括号是空字符串，则元素之间没有分隔符
console.log(newArr.join('')) // red颜色blue颜色pink颜色
console.log(newArr.join('|')) // red颜色|blue颜色|pink颜色
```

red颜色blue颜色pink颜色  
red颜色|blue颜色|pink颜色

## 数组中map + join 方法渲染页面思路：

map遍历数组处理数据生成tr，返回一个数组

把数组转换为字符串

追加给tbody

```
// 页面增加一行数据
function render()
{
    // 先清空tbody，把tbody之间新增的数据取消掉
    tbody.innerHTML = '' // 清空之后，for循环再次增加新数据
    for(let i = 0; i < arr.length; i++) // 1. 取消掉for循环的方式
    {
        // 生成tr
        const tr = document.createElement('tr') // 生成tr标签
        tr.innerHTML =
            ` >${arr[i].stuID}</th>             <th>${arr[i].uname}</th>             <th>${arr[i].age}</th>             <th>${arr[i].gender}</th>             <th>${arr[i].salary}</th>             <th>${arr[i].city}</th>             <th>                 <a href = "javascript:" data-id = ${i}>删除</a>             </th>         `         tbody.appendChild(tr) // 向页面追加元素，父元素.appendChild(子元素)     } } |
```

新增学员

姓名  年龄  性别  女性  男性  就业城市  北京

```
// 页面增加一行数据
function render()
{
    // 先清空tbody，把tbody之间新增的数据取消掉
    tbody.innerHTML = '' // 清空之后，for循环再次增加新数据
    const nArr = arr.map(function(ele){ // 2. 直接用map替代for循环 ele就是当前循环的元素
        return `<tr>
            <th>${ele.stuID}</th>
            <th>${ele.uname}</th>
            <th>${ele.age}</th>
            <th>${ele.gender}</th>
            <th>${ele.salary}</th>
            <th>${ele.city}</th>
            <th>
                <a href = "javascript:" data-id = ${ele}>删除</a>
            </th>
        </tr>
    `})
    tbody.innerHTML = nArr.join('') // 将数组转成字符串显示
}
```

4. 数组转化成html认识的字符串

5. 测试成功

就业榜					
序号	姓名	年龄	性别	薪资	就业城市
3	迪丽热巴	23	女	12000	北京

# 正则表达式

## 1.1 什么是正则表达式

- 正则表达式（Regular Expression）是用于匹配字符串中字符组合的模式。在 JavaScript 中，正则表达式也是对象
- 通常用来查找、替换那些符合正则表达式的文本，许多语言都支持正则表达式。
- 正则表达式在 JavaScript 中的使用场景：
  - 例如验证表单：用户名表单只能输入英文字母、数字或者下划线，昵称输入框中可以输入中文（**匹配**）
    - 比如用户名： `/^[a-zA-Z_-]{3,16}$/`
  - 过滤掉页面内容中的一些敏感词（**替换**），或从字符串中获取我们想要的特定部分（**提取**）等。

- 1. 定义正则表达式语法：

```
const 变量名 = /表达式/
```

其中 / / 是正则表达式字面量

- 比如：

```
const reg = /前端/
```

### 2. 判断是否有符合规则的字符串：

**test()** 方法 用来查看正则表达式与指定的字符串是否匹配

- 语法：

```
regObj.test(被检测的字符串)
```

```
const str = '写入一个前端错误的语法'
const reg = /前端/ // 定义规则
console.log(reg.test(str)) // 匹配返回true, 不匹配返回false
```

## 1.2 语法

### 3. 检索（查找）符合规则的字符串：

**exec()** 方法 在一个指定字符串中执行一个搜索匹配

- 语法：

```
regObj.exec(被检测字符串)
```

- 如果匹配成功， exec() 方法返回一个数组，否则返回null

```
const str = '写入一个前端错误的语法'
const reg = /前端/ // 定义规则
console.log(reg.exec(str)) // 匹配返回数组, 不匹配返回空
```

```
▼ [ '前端', index: 4, input: '写入一个前端错误的语法', groups: undefined ] ← 返回一个数组
  0: '前端'
  groups: undefined
  index: 4 ← index 表示在字符串里面第几个字符开始找到'前端的'
  input: '写入一个前端错误的语法'
  length: 1
  ► [[Prototype]]: Array(0)
```

# 元字符之边界符

## 元字符

为了方便记忆和学习，我们对众多的元字符进行了分类：

- 边界符（表示位置，开头和结尾，必须用什么开头，用什么结尾）比如手机号是用1开头的，这就是边界符
- 量词（表示重复次数）比如手机号是11位，我就规定只能识别11位
- 字符类（比如 \d 表示 0~9）

边界符	说明
^	表示匹配行首的文本(以谁开始)
\$	表示匹配行尾的文本(以谁结束)

如果 ^ 和 \$ 在一起，表示必须是精确匹配。

```
console.log(/哈/.test('哈')) //定义规则, test检测前面字符是否有'哈', 有'哈'返回true  
console.log(/哈/.test('哈哈')) //定义规则, test检测前面字符是否有'哈', 有'哈'返回true  
console.log(/哈/.test('二哈')) //定义规则, test检测前面字符是否有'哈', 有'哈'返回true  
  
//1.边界符^  
console.log(/^哈/.test('哈')) //定义规则,  
console.log(/^哈/.test('哈哈')) //定义规则,  
console.log(/^哈/.test('二哈')) //定义规则, 返回false, 因为它不是以'哈'开头, 而是以'二'开头  
console.log(/^哈$/.test('哈')) //定义规则, 返回true, 以'哈'开头
```



## 元字符之量词

### 2. 量词

- 量词用来 **设定某个模式出现的次数**

量词	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

```
console.log(/^哈*$/.test('哈')) //定义规则, *表示>=0,只要有一个'哈'匹配, 就返回true  
console.log(/^哈*$/.test('哈哈哈哈')) //定义规则, *表示>=0,只要有一个'哈'匹配, 就返回true  
console.log(/^哈*$/.test('')) //定义规则, *表示>=0,没有'哈'匹配, 但是满足>=0,返回true  
console.log(/^哈*$/.test('二哈')) //定义规则, 必须以'哈'开头, '哈'结尾, 返回false  
console.log(/^哈*$/.test('哈很哈')) //'哈'可以出现很多次, 但是不能出现其它字符, 返回false
```

```
console.log(/^哈{4}$/.test('哈哈哈哈')) //定义规则, {4}表示指定的'哈'必须出现4次才返回true  
console.log(/^哈{4}$/.test('哈哈')) //{}没有出现4次, 返回false  
  
console.log(/^哈{4,6}$/.test('哈哈哈哈'))  
//定义规则, {n,m} >=n <=m 逗号', '两侧千万不能有空格, 这里表示'哈'字符必须>4个, <6个返回true  
console.log(/^哈{4,6}$/.test('哈哈哈哈哈哈哈哈')) //匹配字符超标, 返回false  
console.log(/^哈{4,6}$/.test('哈哈')) //匹配字符<4个, 返回false
```

### 3. 字符类：

(1) [ ] 匹配字符集合

- 后面的字符串只要包含 abc 中任意一个字符，都返回 true。

```
// 只要中括号里面的任意字符出现都返回为true
console.log(/[abc]/.test('andy')) // true
console.log(/[abc]/.test('baby')) // true
console.log(/[abc]/.test('cry')) // true
console.log(/[abc]/.test('die')) // false
```

(1) [ ] 里面加上 - 连字符

- 使用连字符 - 表示一个范围

```
console.log(/^[a-z]$/.test('c')) // true
```

● 比如：

- [a-z] 表示 a 到 z 26个英文字母都可以
- [a-zA-Z] 表示大小写都可以
- [0-9] 表示 0~9 的数字都可以

● 认识下：

腾讯QQ号： ^[1-9][0-9]{4,}\$( 腾讯QQ号从10000开始)

```
<input type="text">
<span></span>
<script>
    const reg = /^[a-zA-Z0-9-_]{6,16}$/
    const input = document.querySelector('input')
    input.addEventListener('blur', function () {
        // console.log(reg.test(this.value))
        if (reg.test(this.value)) {
            input.nextElementSibling.innerHTML = '输入正确'
        }
    })
}
```

pink

请输入6~16位的英文数字下划线

### 3. 字符类：

- (3) 预定义：指的是某些常见模式的简写方式。

预定类	说明
\d	匹配0-9之间的任一数字，相当于[0-9]
\D	匹配所有0-9以外的字符，相当于 [^0-9]
\w	匹配任意的字母、数字和下划线，相当于[A-Za-z0-9_]
\W	除所有字母、数字和下划线以外的字符，相当于 [^A-Za-z0-9_]
\s	匹配空格（包括换行符、制表符、空格符等），相等于[\t\r\n\f]
\S	匹配非空格的字符，相当于 [^\t\r\n\f]

日期格式：^\d{4}-\d{1,2}-\d{1,2} 用得最多的是\d和\w

↑  
出现4个数字 ↑  
取1~2位，比如1到30  
最后结果可能是2020-12-8

### 修饰符

- 修饰符约束正则执行的某些细节行为，如是否区分大小写、是否支持多行匹配等
- 语法：

/表达式/修饰符

- i 是单词 ignore 的缩写，正则匹配时字母不区分大小写
- g 是单词 global 的缩写，匹配所有满足正则表达式的结果

```
console.log(/a/i.test('a')) // true
console.log(/a/i.test('A')) // true
```

```
console.log(/^java$/i.test('java')) //返回true
console.log(/^java$/i.test('JAVA')) //返回false 不加i就要区分大小写
```

```
console.log(/^java$/i.test('java')) //返回true
console.log(/^java$/i.test('JAVA')) //返回true
```

- 替换 replace 替换

- 语法：

字符串.replace(/正则表达式/, '替换的文本')

注意：返回值才是替换的字符串

```
const str = '写出需要替换的字符'
const ret = str.replace(/\字符/, '变化')
console.log(ret)
```





## 案例

### 过滤敏感字

需求：要求用户不能输入敏感字

比如，pink老师上课很有\*\*

分析：

①：用户输入内容

②：内容进行正则替换查找，找到敏感词，进行\*\*

③：要全局替换使用修饰符 g

```
<textarea name="" id="" cols="" rows=""></textarea>
<button>发布</button>
<div></div>
const tx = document.querySelector('textarea')
const btn = document.querySelector('button')
const div = document.querySelector('div')

btn.addEventListener('click',function(){
    xtx = tx.value.replace(/激情|奋斗/, '**') // 替换激情或者奋斗为**
    div.innerHTML = xtx // 替换后字符显示
})
```



## JS 新语法(ES6 标准)

### 作用域和作用域链

#### 1. 函数作用域：

在函数内部声明的变量只能在函数内部被访问，外部无法直接访问。

```
<script>
    function getSum() {
        // 函数内部是函数作用域 属于局部变量
        const num = 10
    }
    console.log(num) // 此处报错 函数外部不能使用局部作用域变量
</script>
```

#### 2. 块作用域：

在 JavaScript 中使用 {} 包裹的代码称为代码块，代码块内部声明的变量外部将【有可能】无法被访问。

```
for (let t = 1; t <= 6; t++) {
    // t 只能在该代码块中被访问
    console.log(t) // 正常
}
// 超出了 t 的作用域
console.log(t) // 报错
```

1. let 声明的变量会产生块作用域，var 不会产生块作用域

2. const 声明的常量也会产生块作用域

## JS 垃圾回收机制和闭包

### 1.4 垃圾回收机制

#### 内存的生命周期

JS环境中分配的内存，一般有如下生命周期：

1. 内存分配：当我们声明变量、函数、对象的时候，系统会自动为他们分配内存
2. 内存使用：即读写内存，也就是使用变量、函数等
3. 内存回收：使用完毕，由垃圾回收器自动回收不再使用的内存

```
// 为变量分配内存
const age = 18

// 为对象分配内存
const obj = {
  age: 19
}

// 为函数分配内存
function fn() {
  const age = 18
  console.log(age)
}
```

#### 说明：

- 全局变量一般不会回收(关闭页面回收)
- 一般情况下局部变量的值，不用了，会被自动回收掉

**内存泄漏：**程序中分配的内存由于某种原因程序  
未释放或无法释放叫做**内存泄漏**

### 拓展-JS垃圾回收机制-算法说明

#### • 引用计数

但它却存在一个致命的问题：**嵌套引用**（循环引用）

如果两个对象相互引用，尽管他们已不再使用，垃圾回收器不会进行回收，导致内存泄露。

```
function fn() {
  let o1 = {}
  let o2 = {}
  o1.a = o2
  o2.a = o1
  return '引用计数无法回收'
}
fn()
```

### 1.5 闭包

目标：能说出什么是闭包，闭包的作用以及注意事项

概念：一个函数对周围状态的引用捆绑在一起，内层函数中访问到其外层函数的作用域

简单理解：闭包 = 内层函数 + 外层函数的变量

```

function outer() {
  const a = 1
  function f() {
    console.log(a)
  }
  f()
}
outer()

```

内层函数调用了外层函数的变量，才会产生闭包。

```

let count = 0
function fn() {
  count++
  console.log(`函数被调用${count}次`)
}
fn() // 2
fn() // 3

```

闭包作用：封闭数据，提供操作，外部也可以访问函数内部的变量      但是，这个count是个全局变量，很容易被修改

闭包的基本格式：

```

function outer() {
  let i = 1
  function fn() {
    console.log(i)
  }
  return fn
}
const fun = outer()
fun() // 1
// 外层函数使用内部函数的变量

```



```

// 简约写法
function outer() {
  let i = 1
  return function () {
    console.log(i)
  }
}
const fun = outer()
fun() // 调用fun 1
// 外层函数使用内部函数的变量

```

## 函数动态参数

### 1. 动态参数

`arguments` 是函数内部内置的伪数组变量，它包含了调用函数时传入的所有实参

```

// 求和函数，计算所有参数的和
function sum() {
  // console.log(arguments)
  let s = 0
  for(let i = 0; i < arguments.length; i++) {
    s += arguments[i]
  }
  console.log(s)
}
// 调用求和函数
sum(5, 10) // 两个参数
sum(1, 2, 4) // 三个参数

```

```

function getSum() {
  // arguments 动态参数 只存在于 函数里面
  // 是伪数组
  // console.log(arguments) [2,3,4]
  let sum = 0
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i]
  }
  console.log(sum)
}
getSum(2, 3, 4)
getSum(1, 2, 3, 4, 2, 2, 3, 4)

```

## 箭头函数，JS 的 ES6 标准(重要)

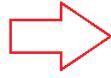
### 箭头函数（重要）

目标：能够熟悉箭头函数不同写法

目的：引入箭头函数的目的是更简短的函数写法并且不绑定this，箭头函数的语法比函数表达式更简洁

使用场景：箭头函数更适用于那些本来需要匿名函数的地方

```
//以前匿名函数是这样写  
const fn = function (){  
    console.log(123)  
}
```



```
//现在箭头函数这样写  
const fn = () =>{  
    console.log(123)  
}  
fn()
```

函数名 形参传入  
const fn = () =>{  
}

```
//箭头函数形参写法  
const fn = (x) =>{  
    console.log(x)  
}  
fn(569)
```

```
//箭头函数如果只有1个形参，小括号可以省略  
const fn = x =>{  
    console.log(x)  
}  
fn(569)
```

```
//箭头函数只有一行代码的应用可以省略大括号  
const fn = x => console.log(x)  
fn(569)
```

```
//箭头函数只有一行代码的应用可以直接运算  
const fn = x => x+x  
console.log(fn(1))
```

Console  
2

```
//箭头函数只有一行代码可以直接返回对象  
const fn = (uname) => ({name:uname}) ←直接返回的对象
```

```
//fn('传入字符数据')  
console.log(fn('传入字符数据'))
```

Console  
▶ {name: '传入字符数据'}

所以箭头函数要返回对象，必须加入()

const obj = {  
 name: 'andy',  
 fun1: function(){  
 console.log(this) //this指向obj  
 }  
}  
obj.fun1()  
  
const fun1 = () => {  
 console.log(this) //this指向浏览器window  
}  
fun1()  
所以箭头函数里面的this是指向一层作用域里面的this

在开发中【使用箭头函数前需要考虑函数中 this 的值】，事件回调函数使用箭头函数时，this 为全局的 window，因此  
DOM事件回调函数为了简便，还是不太推荐使用箭头函数

const obj = {  
 uname: 'pink老师',  
 fun1: function(){  
 let i = 10 //这个函数层有this  
 const count = () => {  
 console.log(this) //此时的this指向obj  
 }  
 count() //调用obj  
 }  
}  
obj.fun1()

## 数组解构(重要)

### 3. 解构赋值

目标：知道解构的语法及分类，使用解构简洁语法快速为变量赋值

```
const arr = [100, 60, 80]  
console.log(arr[0]) // 最大值  
console.log(arr[1]) // 最小值  
console.log(arr[2]) // 平均值
```

以前我们是用下标来拿最大值最小值的数组元素值，但是这种容易搞忘。

```
const arr = [100, 60, 80]  
const max = arr[0]  
const min = arr[1] ←  
const avg = arr[2]  
console.log(max) //最大值 100  
console.log(min) // 最小值 60  
console.log(avg) // 最小值 80
```

我可以这样写来帮助记忆，  
取最大值，最小值。  
但是这样写要写这么多行

#### 基本语法：

1. 赋值运算符 = 左侧的 [] 用于批量声明变量，右侧数组的单元值将被赋值给左侧的变量
2. 变量的顺序对应数组单元值的位置依次进行赋值操作

```
// 普通的数组  
const arr = [1, 2, 3]  
// 批量声明变量 a b c  
// 同时将数组单元值 1 2 3 依次赋值给变量 a b c  
const [a, b, c] = arr  
console.log(a) // 1  
console.log(b) // 2  
console.log(c) // 3
```

```

let a = 1
let b = 2 // 数组解构的前面必须加分号
[b, a] = [a, b] // 1, 2 变成 2, 1
console.log(a,b)

```

// 1. 立即执行函数要加两个立即执行  
`(function () { })();`  
`(function () { })();`

```

const str = 'pink'; 加分号
[1, 2, 3].map(function (item) {
  console.log(item)
})

```

```

const [hr, lx, mi, fz] = ['海尔', '联想', '小米', '方正']
console.log(hr)
console.log(lx)
console.log(mi)
console.log(fz)

```

```

// 请将最大值和最小值函数返回值解构 max 和min 两个变量
function getValue() {
  return [100, 60]
}
const [max, min] = getValue()
console.log(max)
console.log(min)

```

## 对象解构(重要)

### 1. 基本语法:

1. 赋值运算符 = 左侧的 {} 用于批量声明变量，右侧对象的属性值将被赋值给左侧的变量

```

const {uname,age} = {uname:'解构语法测试',age:100} // 对象结构
console.log(uname)
console.log(age)

```

// 解构数组对象

```

const pig = [
  {
    uname:'佩奇',
    age: 18
  }
]
const [{uname,age}] = pig
console.log(uname)
console.log(age)

```

```

const pig = {
  name: '佩奇',
  family: {
    mother: '猪妈妈',
    father: '猪爸爸',
    sister: '乔治'
  },
  age: 6
}
// 多级对象解构
const { name, family: { mother, father, sister } } = pig

```

## forEach 遍历数组

### 遍历数组 forEach 方法 (重点)

- forEach() 方法用于调用数组的每个元素，并将元素传递给回调函数
- 主要使用场景：遍历数组的每个元素
- 语法：

```

被遍历的数组.forEach(function (当前数组元素, 当前元素索引号) {
  // 函数体
})

```

```

const arr = ['red','green','pink']
arr.forEach(function(item,index){
  console.log(item) // 打印数组中的元素
  console.log(index) // 打印数组下标
})

```

