# Javascript基础语法

## JS组成部分

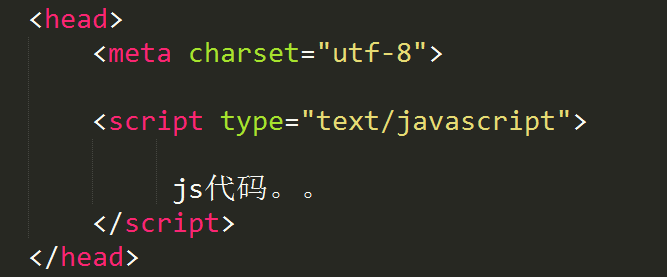
◆ecmascript js标准

◆Dom 通过js操作网页元素

◆bom 通过调用api操作浏览器

## JS书写位置

### 内嵌式写法

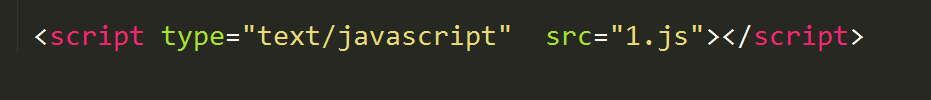


☞ 推荐将script标签放到body标签后面

因为js运行是和代码位置有关。如果js代码在head中，它会先执行js代码，然后显示页面内容。如果js代码过多，则会卡顿的效果。（以下也是可以的）



### 外链式写法



◆新建一个js文件

◆通过script标签中的src属性将外部js引入到html页面中

☞ 推荐将多个js文件合并成一个js文件去加载。

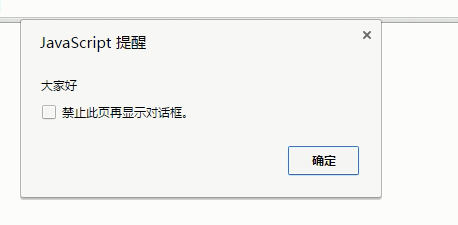
引用外部文件，标签内就不能写内容了

## 页面反馈消息（调试的方法）

### 在页面中以弹窗的形式输出消息（确定按钮）

alert(“信息”);

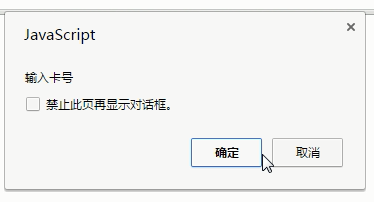
早期用来配合js代码调试。



### 页面输出消息显示 （确定和取消按钮）

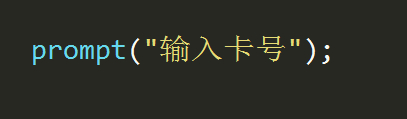
confirm(“消息”);

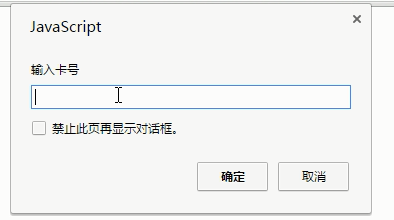
一般和条件判断配合使用



### 接收用户输入信息

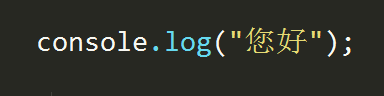
prompt(“信息”);



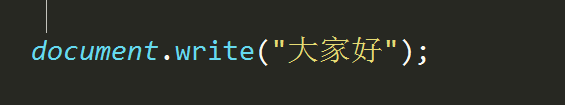


### 控制台显示消息

console.log("您好")



### 在页面中输出消息

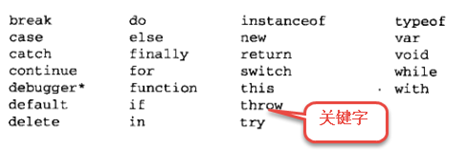
document.write("大家好");

该方法内部可以包含html标签。

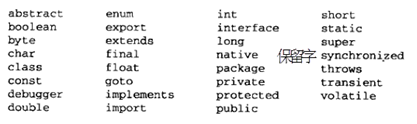
## JS语法要求

* 在js中是严格区分字母大小写
* 在写js代码过程中分号“ ; ”必须的写

### JS关键字



### JS保留字



## JS数据类型

### 简单数据类型

#### number 数字类型

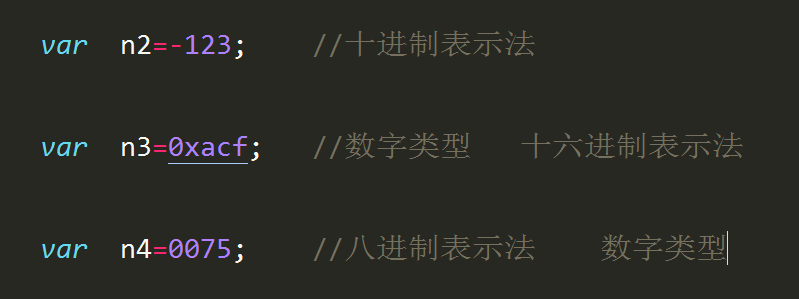
数字类型的表示方式：

◆十进制表示法

◆十六进制表示法

以0x开头 数字0-9 和 a(A)-f(F)

◆八进制表示法

以数字0开头 0-7之间的组合。

#### string 字符串类型

☞ 转义字符

\” 转双引号

\’ 转单引号

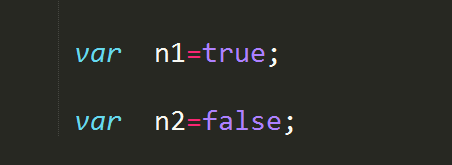
\r 回车

\n 换行

#### boolean 布尔类型

☞ true

☞ false



**使用Boolean()转换**

\* true

* 非空字符串
* 非零数值
* 非空对象

\* false

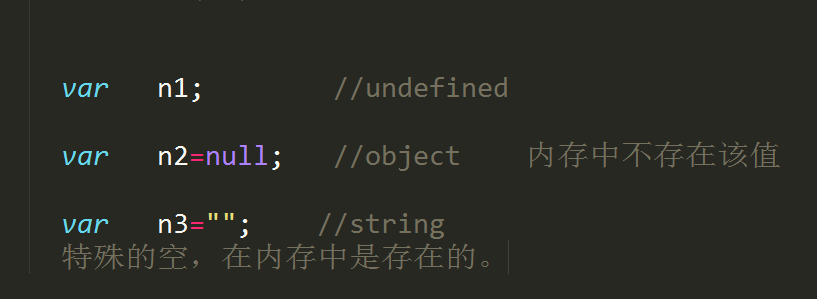
* 空字符串
* 0 以及 NaN
* 空对象,Null以及undefined

#### undefined 变量未初始化

定义了变量，但是没有过给变量进行赋值。

#### null object

该变量的引用为空



**Null 和 Undefined 区分**

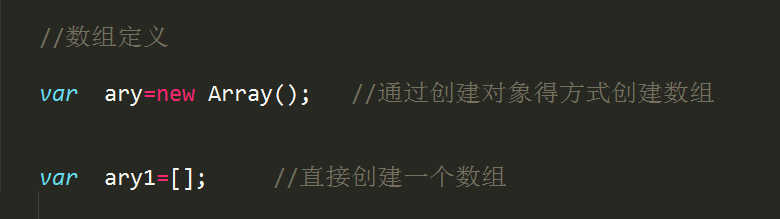
* Null表示空指针对象,Undefined表示声明变量后未初始化
* typeof: Null返回类型为Object,Undefined返回类型为Undefined
* Undefined派生于null,使用"=="比较时会返回true

### 复杂数据类型

#### object 对象

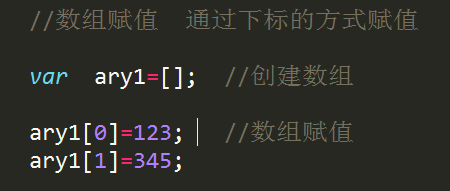
#### Array 数组

定义



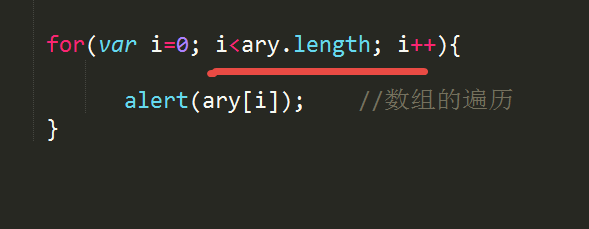
赋值

数组中通过下标的方式进行赋值。下标从0开始。



属性length

length 可以获取数组中的元素个数。



数组合并

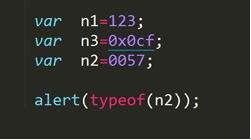


数组返回字符串



### 数据类型判断

typeof(变量)



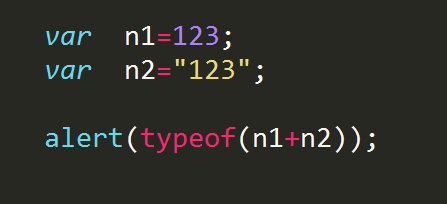
### 数据类型转换

#### 隐式类型转换

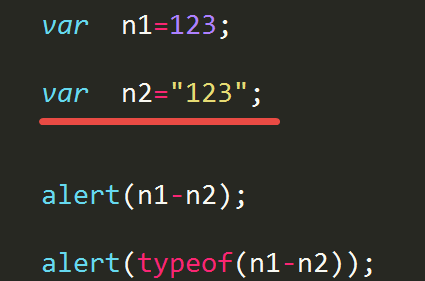
将变量参与到程序运算中，程序将变量进行类型转换

（不是程序员直接进行的类型转换）

* 一个数字类型和一个字符串类型变量相加，最后结果为字符串，加号起的是一个连接的作用



* 如果一个数字类型的变量和一个数字的字符串类型的变量相减，最后结果为数字类型



* 如果一个数字类型的变量和一个非数字类型的变量相减，最后结果为NaN

NaN: not a number 不是一个数字

NaN的数据类型属于number类型。

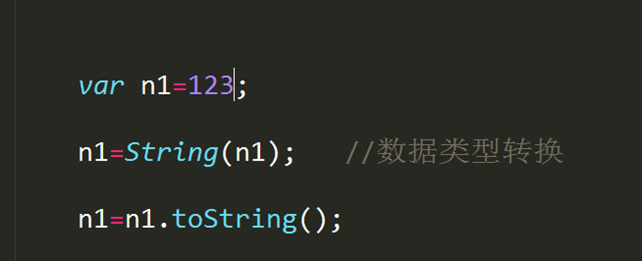
* 如果一个数字类型的变量和一个数字的字符串变量相除，最后结果为数字类型
* 如果一个将数字0作为除数，得出的结果是infinity（无穷大）

infinity的数据类型属于number类型

#### 强制类型转换

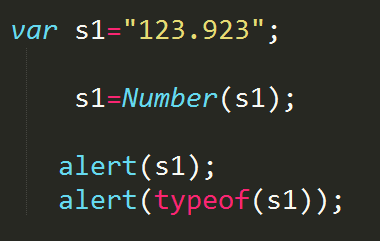
* 将数字转换为字符串

String() .toString();



* 将字符串转换为数字

☞ Number()



非数字类型的字符串最后结果为NaN

☞ parseInt()

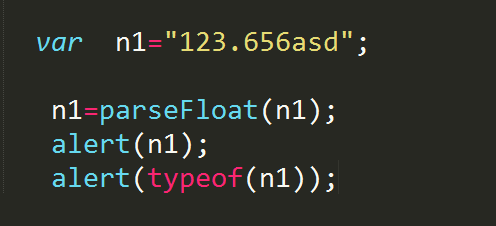
将数字类型的字符串通过该方法转换后，只保留数字的整数部分，不会进行四舍五入运算

非数字类型的字符串转换后的结果为NaN

如果在数字后加上非数字的字符，也可以转换成功

☞ parseFloat()

最后保留的元数据，不会对数字进行四舍五入运算。

如果在数字后加上非数字的字符，也可以转换成功 

* 将其他类型转换为Boolean类型

可以将数字0 转换为false

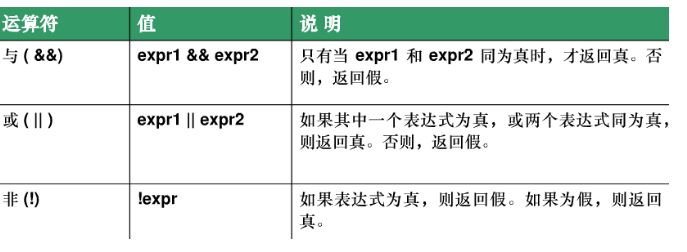
其他数据类型通过boolean强制类型转换后结果true

## 运算符

**运算符类型**

* 算术运算符: + - \* / %
* 赋值运算符: = += -= \*= %= /=
* 自增自减: ++ --
* 位运算符: <<< >>
* 关系运算符: > < >= <= == !=
* 逻辑运算符: ! && ||(优先级依次降低)

### 逻辑运算符



### 等号运算符

☞ “=” 赋值运算符

☞ “==” 判断是否相同

判断变量是否相同，只判断变量的内容是否相同，与变量的数据 类型无关。

☞ “===” 判断是否相同

判断变量是否相同，除了判断变量的内容还要判断变量的数据类型是否一致。

☞ != 不相等

判断内容是否不相同，不考虑数据类型

☞ !== 不相等

判断的是内容和数据类型是否不相同。

### 逗号运算符



逗号运算符，可以将变量进行集体声明

## 逻辑结构

### switch

Switch(变量n){

case 10:

执行的代码;

break;

case 20:

执行的代码;

break;

default:

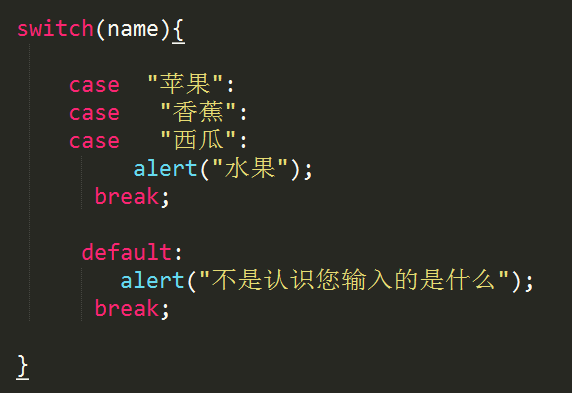
执行的代码;

break;

}

Switch语句后面的变量数据类型必须和case后面的数据类型保持一致。

Switch语句可以对变量进行集体判断：



### 自增自减

var i=123;

var n1=i++; 该段代码的含义是： 先将i的值赋值给变量n1,然后变量i的值再加1。

var n2=++i; 该段代码的含义是： 先将i的值加1,然后变量i加1后的值赋值给变量n2。

### break 与continue

在循环体内，只要代码遇到break,程序立马结束当前循环。

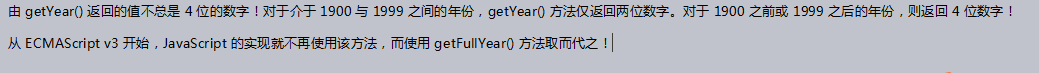
（当前循环指的是break语句所在的循环体。）

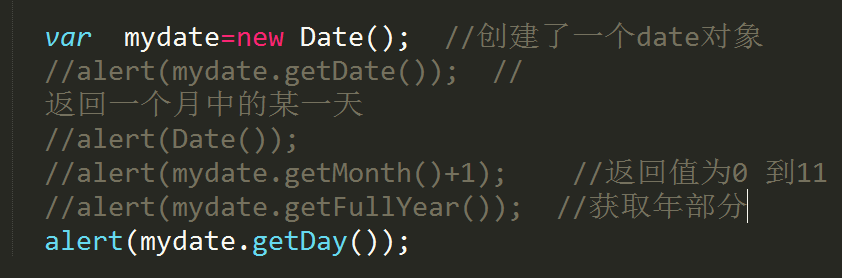
Continue语句指的是跳出本次循环，该语句后面的代码不再执行。

## 相关函数

### Date用法







### Math对象

☞ Math.round(x)

对数字进行四舍五入运算，最后的结果为整数

☞ Math.ceil() 天花板函数

返回一个数字的整数部分

对该数字进行向上舍入

（注意：该方法不会对数字进行四舍五入运算。）

☞ Math.floor() 地板函数

对该数字进行向下舍入

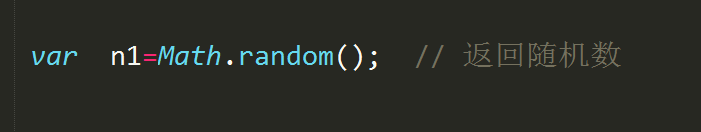
☞ Math.max(x,y)

返回x，y之间的最大值

☞ Math.min(x,y)

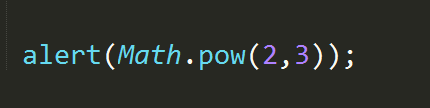
返回x，y之间的最小值

☞ Math.random()



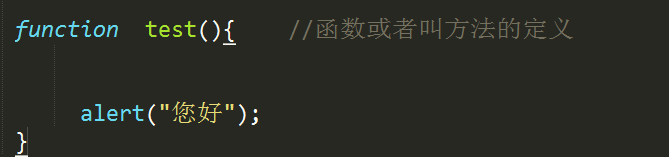
☞ Math.pow(x,y)

返回x值的y次方



### 自定义函数

通过 function关键字 和 自定义方法名 既可定义一个函数。



## JS对象

### 创建对象

三种方式：

1. var obj = {};

2. var obj1 = new Object();

3. var obj2 = Object.create();

1跟2的区别：

推荐使用第一个方式

第二种方式存在效率问题，因为要new对象，会涉及到原型查找的问题。

### 构造函数

构造函数

```js

function Person(name,age){

this.name = name;

this.age = age;

this.sayHello = function(){

alert(this.name);

}

}

// 1. 创建一个对象

// 2. this指针指向这个对象

// 3. 执行构造函数内的代码

// 4. 将构造函数返回

var p = new Person("John",18);

p.sayHello();

```

构造函数继承

更改this指向的位置

* + apply():利用数组的方式改变this指向
  + call():利用参数的方式改变this指向

```js

function Student(name,age,id){

Person.apply(this,[name,age]);

//Person.call(this,name,age);

this.id = id;

this.study = function(){

alert(this.id + "study");

}

}

var p2 = new Student("Tom",18,"#001");

p2.sayHello();

```

### 原型prototype

##### 概念

**注：一个已存在的对象构造器中是不能添加新的属性**

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法：

 Date 对象从 Date.prototype 继承。

 Array 对象从 Array.prototype 继承。

 Person 对象从 Person.prototype 继承。

所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例。

JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

Date 对象, Array 对象, 以及 Person 对象从 Object.prototype 继承。

**利用原型为对象添加属性和方法**

 有的时候我们想要在所有已经存在的对象添加新的属性或方法。

 另外，有时候我们想要在对象的构造函数中添加属性或方法。

 使用 prototype 属性就可以给对象的构造函数添加新的属性：

##### 原型

```js

function Person(){

}

Person.prototype.name = "John";

Person.prototype.age = 15;

Person.prototype.sayHello = function(){

console.log(this.name);

}

var p = new Person();

p.sayHello();

```

当访问一个对象的属性时,先在该对象的本身进行查找

如果找到就直接返回这个属性,停止查找

如果没有找到,会在原型上找,即在\_\_proto\_\_原型指向的对象上进行查找

##### 原型的继承

```js

function Person(){

}

Person.prototype.name = "John";

Person.prototype.age = 15;

Person.prototype.sayHello = function(){

console.log(this.name);

}

function Student(){

}

Student.prototype = Object.create(Person.prototype);

var s = new Student();

s.sayHello();

```

##### 混合继承

构造函数继承 + 原型链继承

```js

function Person(name,age){

this.name = name;

this.age = age;

}

Person.prototype.sayHello = function(){

console.log(this.name);

}

function Student(name,age,id){

Person.apply(this,[name,age]);

this.id = id;

}

Student.prototype = Object.create(Person.prototype);

Student.prototype.study = function(){

alert(this.id + "study");

}

var s = new Student("Bob",12);

s.sayHello();

```

# Javascript进阶

## 作用域

* 局部作用域

变量在函数内声明，变量为局部作用域。局部变量：只能在函数内部访问。

* 全局作用域

变量在函数外定义，即为全局变量。全局变量有全局作用域: 网页中所有脚本和函数均可使用。

* JavaScript 变量生命周期在它声明时初始化。局部变量在函数执行完毕后销毁。全局变量在页面关闭后销毁。

## 事件

**事件：**访问web页面的用户引起的一系列操作

### 事件类型

所有事件处理函数由两部分组成: on + 事件名称

* 鼠标事件

- onclick

- onmouseover

- onmousedown

- onmouseup

* 键盘事件

- onkeypress(只有字符键才会触发)

- onkeydown(所有键都会触发)

- onkeyup

* HTML事件

- 表单元素

- form:onsubmit

- input:onfocus,onblur(失去焦点),onchange,oninput

- 页面加载

- onload:页面加载完成

键盘事件的区别:

- onkeypress

- 这个事件在用户按下并放开任何字母数字键时发生。系统按钮（例如，箭头键和功能键）无法得到识别。

- onkeyup

- 这个事件在用户放开任何先前按下的键盘键时发生。

- onkeydown

- 这个事件在用户按下任何键盘键（包括系统按钮，如箭头键和功能键）时发生。

### 事件调用方法

```html

<!--直接执行代码-->

<button onclick='getElementById("demo").innerHTML=Date()'>

The time is?

</button>

<!--调用JS的方法-->

<button onclick="displayDate()">The time is?</button>

```

### 事件对象

触发事件时,会产生一个事件对象,包含事件的所有信息

通过事件绑定的执行函数,可以得到隐藏参数 event对象

**方式：**

```js

odiv.onclick = function(e){

consolo.log(e);

}

```

**Event对象的获取方式：**

* 鼠标事件的对应event

- e.clientX 鼠标距离当前可视区边界的坐标

- e.clientY

- e.pageX 鼠标距离当前页面的坐标(包含滚动条隐藏的部分)

- e.pageY

- e.offsetX 鼠标距离事件源对应标签的坐标

- e.offsetY

* 键盘事件的对应event

- e.keyCode

- e.altKey

- e.ctrlKey

- e.shiftKey

### 事件流

**事件流:**页面接收事件的顺序

**事件流的三个阶段:**

- 捕获:从外到内依次执行事件

- 目标

- 冒泡:从内到外依次执行事件

**阻止冒泡:**只触发内层事件,不触发外层事件

- event.cancelBubble = true

- event.stopPropagation();

**事件委托:**利用事件冒泡,把本应该给某元素的事件委托给他的父级(外层)

- 事件委托的应用,让通过js代码新产生的标签也有和同级标签一样的触发函数

### 事件委托

```html

<body>

<ul>

<li>1</li>

<li>2</li>

<li>3</li>

</ul>

<script>

var oLI = document.getElementsByTagName("li");

var oUL = document.getElementsByTagName("ul")[0];

<!-- 利用e.target,将事件委托给父级元素 -->

oUL.onclick = function(e){

var \_target = e.target;

if(\_target.nodeName.toUpperCase() == "LI"){

console.log(\_target.innerHTML);

}

}

</script>

</body>

```

### 浏览器默认行为

**标签本身具有的属性**

- a标签的跳转

- submit按钮的提交

- 右键菜单

- 文本框输入

**阻止默认行为的方式**

- event.preventDefault()

- event.returnValue = false;

- return false;

```html

<body>

<a href="">test</a>

<script>

var a = document.getElementsByTagName("a")[0];

a.onclick = function(){

console.log("我点了a");

return false;//如果没有阻止默认行为,点击事件就无法正常运行

}

</script>

</body>

```

- 如果input标签阻止默认行为，就无法进行输入

- input.onkeydown

- 阻止右键菜单

- document.oncontextmenu

### DOM2级事件处理程序

* 添加事件监听器

- addEventListener(事件名,处理函数,布尔值)

* 移除事件监听器

- removeEventListener(事件名,处理函数)

[注]

* 事件监听器中的事件名,不加on
* 布尔值:true:捕获,false:冒泡
* 事件监听器的作用:使标签可以执行多个响应函数
* 事件监听器只能通过remove来移除

## 字符串

### 转义字符串



### 字符串与字符串对象

**两种方法：**

* var firstName = "John"
* var firstName = new String("John")

**区别：**

* 两种方式创建的字符串并不是一种类型
* 不要创建 String 对象。它会拖慢执行速度，并可能产生其他副作用

字符串一旦被创建,将无法改变,要改变必须摧毁原有字符串

### 字符串的属性和方法

原始值字符串，如 "John", 没有属性和方法(因为他们不是对象)。

原始值可以使用 JavaScript 的属性和方法，因为 JavaScript 在执行方法和属性时可以把原始值当作对象。

##### 属性



##### 方法



String()和toString()的区别:

* toString()可以转化除了Null和Undefined以外的数据类型
* String()可以将null和undefined转换为字符串，但是没法转进制字符串

```javaScript

var i=1;

var j=null;

var k;

j=i.toString();

console.log(typeof(j)+":"+j);//返回string:1

// j=k.toString();

// console.log(typeof(j)+":"+j);//报错

j=String(null);

console.log(typeof(j)+":"+j);//返回string:null

j=String(k);

console.log(typeof(j)+":"+j);//返回string:undefined

```

## 数组

### 创建数组

##### 一维数组

// 方法一:new

var array = new Array(1,true,"hello");/\* 数组内容 \*/

var array2 = new Array(5);/\* 设置数组大小 \*/

console.log(array);

console.log(array2);

// 方法二:省略new

var array3 = Array(1,true,"hello");/\* 数组内容 \*/

console.log(array3);

//方法三:直接创建

var array4 = [1,true,"hello"];

console.log(array4);

注:length属性,表示只读的,可以进行设置

##### 二维数组

# 二维数组

生成二维数组

```js

// 生成二维数组

var count = 0;/\* 从1到25累加的中间变量 \*/

var arr = [];

for(var i = 0; i < 5; i++){

var newArr = [];

//每个newArr中都有五个数

for(var j = 0; j < 5; j++){

newArr.push(++count);

}

arr.push(newArr);

}

console.log(arr);

```

打印二维数组

```js

for(var i = 0; i < arr.length; i++){

for(var j = 0; j < arr[i].length; j++){

document.write(arr[i][j]+"\t");

}

document.write("<br/>");

}

```

### 遍历数组

for( in ){} 快速遍历数组,效率比普通for循环更高(不需要条件判断)

```js

var array = [1,true,"hello"];/\* 数组内容 \*/

for(var i in array){

console.log(array[i]);

}

```

### 数组函数

##### 栈方法

push()

[格式] 数组.push(元素...)

- 功能:给数组末尾添加元素

- 参数:参数个数随意

pop()

[格式] 数组.pop()

- 功能:移除末尾最后一个元素

- 参数:无

- 返回值:被移除的元素

##### 队列方法

push()

[格式] 数组.push(元素...)

- 功能:给数组末尾添加元素

- 参数:参数个数随意

shift()

[格式] 数组.shift()

- 功能:给数组头部取下一个元素

- 参数:无

- 返回值:取下的元素

unshift()

[格式] 数组.unshift(元素...)

- 功能:给数组头部添加元素

- 参数:参数个数随意

##### 其他

concat()

[格式] 数组1.concat(数组2)

- 功能:将两个数组合并为一个数组,原数组不变

- 参数:合并后的数组

slice()

[格式] 数组.slice(start,end)

- 功能:从数组中的指定位置获取元素并创建新数组,原数组不变

- 参数:从start开始,不包括end指定的下标元素

- 返回值:新数组

splice()

[格式] 数组.splice(start,length,element...)

- 功能:可以完成删除,插入,替换操作

- 参数:

- start:截取开始的下标位置

- length:截取长度

- element:从截取开始的下标位置,插入任意数量元素

- 返回值:删返插无

```js

var array = new Array(1,true,"hello");/\* 数组内容 \*/

// 删除元素

array.splice(1,1);

console.log(array);

// 插入元素

array.splice(1,0,"嚓嚓嚓");

console.log(array);

//替换元素

array.splice(1,1,"哇哇哇");

console.log(array);

```

join()

[格式] 数组.join("拼接符")

- 功能:使用拼接符,将数组元素拼接为字符串

- 参数:拼接符

- 返回值:字符串

### 数组排序

##### 系统提供的排序方法

reverse()

[格式] 数组.reverse()

- 功能:将原数组进行逆向排序,改变原数组

- 参数:无

- 返回值:无

sort()

[格式] 数组.sort()

- 功能:将原数组进行升序排序,改变原数组

- 参数:无

- 返回值:无

注: sort默认按照字符串进行排序

##### 自定义排序算法

一般情况,我们需要自己编写排序算法,系统提供的排序方法使用较少

1. 冒泡排序

```js

//冒泡排序

//原理：先遍历一遍 0-n-1 把最大的数放到最后，再遍历一遍 0-n-2找到第二大的数放到第二位，以此类推

var arr = [3,7,8,4,5,2,1,6,9]

function bubbleSort(arr){

for(var i = 0; i < arr.length; i++){

for(var j = 0; j < arr.length - i - 1; j++){

if(arr[j] > arr[j+1]){

var tmp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = tmp;

}

}

}

//return arr;//这句话可以不要,因为函数会改变复合数据类型的值

}

//arr = bubbleSort(arr);//因此这一句话也可以不要,直接执行函数

bubbleSort(arr);

// alert(arr);

```

2. 选择排序

```js

//选择排序

//原理:第一次遍历，通过每次比较,选出更小的放在第一个位置上,第二次遍历，通过每次比较,选出更小的放在第二个位置,以此类推

var arr = [3,7,8,4,5,2,1,6,9]

function selectSort(arr){

for(var i = 0; i < arr.length-1; i++){

for(var j = i + 1; j < arr.length; j++){

if(arr[j] < arr[i]){

var tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

}

}

}

return arr;

}

arr = selectSort(arr);

alert(arr);

```

### 数组引用

函数内对同名变量的操作不会影响到函数外对同名变量的操作

不同于变量的作用域，数组属于复合数据类型/引用数据类型

注:

* + 函数的运行空间是预先分配好的.一旦被分配好内存空间,就不能进行改变了.
  + 堆空间,可以随时分配内存
  + 复合数据类型存储堆空间地址,堆空间存储实际值
  + 函数会改变复合数据类型在堆空间的值,因此会影响函数外的值

```js

var arr = [1,2]

function add(arr){

arr.push(3,4)

}

add(arr);

// alert(arr);//返回 1,2,3,4

var a = 1;

function add2(a){

a = a + 1;

}

add2(a);

// alert(a);//返回 1

```

## 正则表达式

由普通字符和特殊字符组成的对字符串进行过滤的逻辑公式

对象: RegExp

### 创建正则表达式的方式

* + 字面量的方式

- var reg = /abc/;

* + 构造函数

- var reg = new RegExp("abc");

### 正则表达式修饰符

var reg = /abc/ig;

* i:忽略大小写
* g:执行全局匹配
* m:执行多行匹配

### 正则表达式语法

* 元字符

\* . :除换行符之外的所有单个字符

\* \d:查找数字(\D查找非数字)(相当于[0-9])

\* \w:查找字母,数字,下划线(相当于[0-9a-zA-Z])

\* \uXXX:查找十六进制数XXX规定的Unicode字符

* 空白字符

\* \b:空格

\* \t:制表符

\* \n:换行

\* \r:回车

\* \s:匹配空格,换行符,制表符,空白符(\S非空白符)

* 限定符

\* n+: 1-多次

\* n\*: 0-多次

\* n?: 0-1次

\* n{m,n}:m-n次

\* [abc] :查找方括号之间的任意字符

\* [0-9] :查找0-9之间的任意数字

\* ()|() :查找任何以|分隔的选项

\* ():分组,将内容作为整体进行匹配

\* $1 $2

```js

var reg = /(.\*)\s(.\*)/;

var str = "taobao baidu";

consolo.log(str.replace(reg,"$2 $1"));//返回结果: "baidu taobao"

```

* 定位符

\* ^:行首

\* $:行尾

\* \b:匹配单词边界

### 正则表达式的相关方法

* 字符串方法

- str.match(reg):返回正则表达式的匹配结果

- str.search(reg):检索匹配字符串的起始位置

- str.split(reg):根据正则表达式进行分离

- str.replace(reg,str):替换匹配的字符串

* 正则对象方法

- reg.test():测试是否和正则表达式匹配,返回true/false

- exec():返回数组存储匹配结果,没有匹配返回null

- 与 str.match() 方法不同,可以不用添加g全局匹配

- compile():

- compile() 方法用于改变 RegExp。

- compile() 既可以改变检索模式，也可以添加或删除第二个参数。

### 正则表达式实例

```html

<body>

用户名:<input type="text" id="username"/>

(数字,字母,下划线组成的6-15位数字,且不能以数字开头)

<script>

var username = document.getElementById("username");

username.onchange = function(){

var val = this.value;

var reg = /^[a-zA-Z\_]\w{5,14}$/;

if(reg.test(val)){

alert("格式正确");

}else{

alert("格式错误");

}

}

</script>

</body>

```

## 错误

try 语句测试代码块的错误。

catch 语句处理错误。

throw 语句创建自定义错误。

finally 语句在 try 和 catch 语句之后，无论是否有触发异常，该语句都会执行。

try {

... //异常的抛出

} catch(e) {

... //异常的捕获与处理

} finally {

... //结束处理

}

## 调错

* 设置断点
* debugger关键字

## 变量提升

**声明提升：**

**JavaScript 中，函数及变量的声明都将被提升到函数的最顶部。**

**JavaScript 中，变量可以在使用后声明，也就是变量可以先使用再声明。**



**初始化不会提升：**



**在头部声明你的变量**

对于大多数程序员来说并不知道 JavaScript 变量提升。

如果程序员不能很好的理解变量提升，他们写的程序就容易出现一些问题。

为了避免这些问题，通常我们在每个作用域开始前声明这些变量，这也是正常的 JavaScript 解析步骤，易于我们理解。

## 严格模式

JavaScript 严格模式（strict mode）即在严格的条件下运行。

**使用 "use strict" 指令**

"use strict" 指令在 JavaScript 1.8.5 (ECMAScript5) 中新增。

它不是一条语句，但是是一个字面量表达式，在 JavaScript 旧版本中会被忽略。

"use strict" 的目的是指定代码在严格条件下执行。

严格模式下你不能使用未声明的变量。

"use strict";

myFunction();

function myFunction() {

y = 3.14; // 报错 (y 未定义)

}

为什么使用严格模式:

* 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
* 消除代码运行的一些不安全之处，保证代码运行的安全；
* 提高编译器效率，增加运行速度；
* 为未来新版本的Javascript做好铺垫。

## 验证

<input id="id1" type="number" min="100" max="300" required>

<button onclick="myFunction()">验证</button>

<p id="demo"></p>

<script>

function myFunction() {

var inpObj = document.getElementById("id1");

if (inpObj.checkValidity() == false) {

document.getElementById("demo").innerHTML = inpObj.validationMessage;

}

}

</script>

### 验证DOM方法



### 验证DOM属性



### input -validity属性



## this关键字

面向对象语言中 this 表示当前对象的一个引用。

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

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

var person1 = {

fullName: function() {

return this.firstName + " " + this.lastName;

}

}

var person2 = {

firstName:"John",

lastName: "Doe",

}

person1.fullName.call(person2); // 返回 "John Doe"

## let 和 const

##### let

let 声明的变量只在 let 命令所在的代码块 {} 内有效，在 {} 之外不能访问。

{

let x = 2;

}

// 这里不能使用 x 变量

**let和var的区别**

使用 var 关键字声明的全局作用域变量属于 window 对象

使用 let 关键字声明的全局作用域变量不属于 window 对象

使用 var 关键字声明的变量在任何地方都可以修改

在相同的作用域或块级作用域中，不能使用 let 关键字来重置 var 关键字声明的变量

在相同的作用域或块级作用域中，不能使用 let 关键字来重置 let 关键字声明的变量

let 关键字在不同作用域，或不同块级作用域中是可以重新声明赋值的

let x = 2; // 合法

var x = 3; // 不合法

{

let x = 4; // 合法

var x = 5; // 不合法

}

JavaScript 中，var 关键字定义的变量可以在使用后声明，也就是变量可以先使用再声明

let 关键字定义的变量则不可以在使用后声明，也就是变量需要先声明再使用

##### const

const 用于声明一个或多个常量，声明时必须进行初始化，且初始化后值不可再修改

const定义常量与使用let 定义的变量相似：

* 二者都是块级作用域
* 都不能和它所在作用域内的其他变量或函数拥有相同的名称

两者还有以下两点区别：

* const声明的常量必须初始化，而let声明的变量不用
* const 定义常量的值不能通过再赋值修改，也不能再次声明。而 let 定义的变量值可以修改。

**const并非真正的常量**

const 的本质: const 定义的变量并非常量，并非不可变，它定义了一个常量引用一个值。使用 const 定义的对象或者数组，其实是可变的。

不会报错：

// 创建常量对象

const car = {type:"Fiat", model:"500", color:"white"};

// 修改属性:

car.color = "red";

// 添加属性

car.owner = "Johnson";

但是我们不能对常量对象重新赋值：

const car = {type:"Fiat", model:"500", color:"white"};

car = {type:"Volvo", model:"EX60", color:"red"}; // 错误

## javascript:void(0)

**href="#"与href="javascript:void(0)"的区别**

# 包含了一个位置信息，默认的锚是#top 也就是网页的上端。

而javascript:void(0), 仅仅表示一个死链接。

在页面很长的时候会使用 # 来定位页面的具体位置，格式为：# + id。

如果你要定义一个死链接请使用 javascript:void(0) 。

## 递归

尽量不要使用递归,效率太低,太占用内存

递归调用方法:

1. 找到临界值:无需计算就可以得到的值

2. 找这一次和上一次的关系

3. 假设当前函数可以使用,调用自身计算上一次的结果,写出这一次的运行结果

```js

// 计算1到n的和

function sum(n){

if (n == 1) {

return 1;

}

return sum(n-1) + n;

}

console.log(sum(20));

```

## 定时器

window 对象允许以指定的时间间隔执行代码。这些时间间隔称为定时事件。

通过 JavaScript 使用的有两个关键的方法：

* setTimeout(function, milliseconds)

在等待指定的毫秒数后执行函数。（执行一次）

* setInterval(function, milliseconds)

等同于 setTimeout()，但持续重复执行该函数。（执行多次）

[返回值] 返回当前定时器的唯一标识(定时器ID)

对应清除方法

* clearTimeout(定时器ID):清除对应定时器
* clearInterval(定时器ID):清除对应定时器