**JavaScript**

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

**基本语法**

1. JS中严格区分大小写

2. JS中每一条语句以分号(;)结尾

- 如果不写分号，浏览器会自动添加，但是会消耗一些系统资源，而且有些时候，浏览器会加错分号，所以在开发中分号必须写

1. JS中会忽略多个空格和换行，所以我们可以利用空格和换行对代码进行格式化

**字面量**，都是一些不可改变的值

比如 ：1 2 3 4 5

字面量都是可以直接使用，但是我们一般都不会直接使用字面量

**变量**

-变量可以用来保存字面量，而且变量的值是可以任意改变的

-变量更加方便我们使用，所以在开发中都是通过变量去保存一个字面量，而很少直接使用字面量

-可以通过变量对字面量进行描述

//声明变量

var a; //在js中使用var关键字来声明一个变量

a = 123; //为变量赋值,其中123为字面量

**标识符**

- 在JS中所有的可以由我们自主命名的都可以称为是标识符

- 例如：变量名、函数名、属性名都属于标识符

- 命名一个标识符时需要遵守如下的规则：

1.标识符中可以含有字母、数字、\_、$

2.标识符不能以数字开头

3.标识符不能是ES中的关键字或保留字

4.标识符一般都采用驼峰命名法(首字母小写，每个单词的开头字母大写，其余字母小写)

helloWorld xxxYyyZzz

JS底层保存标识符时实际上是采用的Unicode编码，所以理论上讲，所有的utf-8(中文)中含有的内容都可以作为标识符，[但是一般不这么用。](https://www.cnblogs.com/yyzyxy/p/7184417.html)

**数据类型，**指的就是字面量的类型

在JS中一共有六种数据类型：

String 字符串

Number 数值

Boolean 布尔值

Null 空值

Undefined 未定义

Object 对象(Data、function、Array)

其中String Number Boolean Null Undefined属于基本数据类型,而Object属于引用数据类型

最新的 ECMAScript 标准定义了 8种数据类型:

String

Number

Boolean

Null

Undefined

BigInt(安全存储、操作大整数)

Symbol(标记,永远不相等，解决属性名冲突问题) Symbol('a')===Symbol('a') //false

Object (Data、function、Array)

**String类型**

- 在JS中字符串需要使用引号引起来

- 使用双引号或单引号都可以，但是不要混着用

- 单引号和双引号之间可以相互嵌套

1、单引号内只能嵌套双引号。

2、双引号内只能嵌套单引号。

3、如果想在双引号内，再嵌套双引号，里面的双引号，一定要进行转义（\”）

在 JavaScript 中，有三种包含字符串的方式。

双引号："Hello".

单引号：'Hello'.

反引号：`Hello`. 反引号是功能扩展引号。允许将变量和表达式包装在 ${…} 中

在字符串中我们可以使用\作为转义字符，当表示一些特殊符号时可以使用\进行转义

| **字符** | **描述** |
| --- | --- |
| \n | 换行 |
| \r | 回车：不单独使用。Windows 文本文件使用两个字符 \r\n 的组合来表示换行。 |
| \', \" | 引号 |
| \\ | 反斜线 |
| \t | 制表符 |
| \b, \f, \v | 退格，换页，垂直标签 —— 为了兼容性，现在已经不使用了。 |
| \xXX | 具有给定十六进制 Unicode XX 的 Unicode 字符，例如：'\x7A' 和 'z' 相同。 |
| \uXXXX | 以 UTF-16 编码的十六进制代码 XXXX 的 unicode 字符，  例如 \u00A9 —— 是版权符号 © 的 unicode。它必须正好是 4 个十六进制数字。 |
| \u{X…XXXXXX}  （1 到 6 个十六进制字符） | 具有给定 UTF-32 编码的 unicode 符号。  一些罕见的字符用两个unicode符号编码，占用4个字节。这样我们就可以插入长代码了。 |

**字符串的相关方法**

在底层字符串是以字符数组的形式保存的 ["H","e","l","l","0"]

length属性 - 用来获取字符串的长度

charAt() - 返回字符串中指定位置的字符,根据索引获取指定的字符(索引都是0开始）

var result = str.charAt(6)=str[6];

concat() - 用来连接两个或多个字符串,作用和+一样

var result=str.concat("A","B")

var result=str+"A"+"B"

indexof() - 检索一个字符串中是否含有指定内容,第二个参数，指定开始查找的位置

①如果字符串中含有该内容，则会返回其第一次出现的索引

②如果没有找到指定的内容，则返回-1

lastIndexOf() -用法和indexOf()一样，不同的是indexOf是从前往后找，而lastIndexOf是从后往前找

截取字符串 - 可以从字符串中截取指定的内容,不会影响原字符串，而是将截取到内容返回

- 参数： ①开始位置的索引（包括开始位置）②结束位置的索引（不包括结束位置）

- 如果省略第二个参数，则会截取到后边所有的

- 传递一个负数作为参数，负数的话将会从后边计算

①substr(开始索引，返回个数)

②substring(开始索引，结束索引)

③slice(开始索引，结束索引) 参数可以是负数

split() - 将一个字符串拆分为一个数组

- 参数：需要一个字符串作为参数，将会根据该字符串去拆分数组如果传递一个空串作为参数，

则会将每个字符都拆分为数组中的一个元素

toUpperCase() -将一个字符串转换为大写并返回

toLowerCase() -将一个字符串转换为小写并返回

replace("正则/字符串","新内容") -将匹配的内容替换，返回新的字符串

match("正则/字符串","新内容") -将匹配的内容封装到数组，返回数组

search("正则/字符串") -返回首次出现的索引，没有返回-1

查找子字符串时，使用 indexOf 或 includes/startsWith/endsWith 进行简单检查。

[includes(substr,pos)](https://developer.mozilla.org/zh/docs/Web/JavaScript/Reference/Global_Objects/String/includes)  判断是否给定的substr值，返回boolean

[startsWith(](https://zh.javascript.info/string" \l "includesstartswithendswith)[substr)](https://zh.javascript.info/string" \l "includesstartswithendswith) [判断是否已substr值开头，返回boolean](https://zh.javascript.info/string" \l "includesstartswithendswith)

[endsWith](https://zh.javascript.info/string" \l "includesstartswithendswith)(substr) 判断是否已substr值结束，返回boolean

length 属性表示字符串长度:

length 是一个属性,str.length 是一个数字属性，而不是函数。后面不需要添加括号。

'abcd'.length //4

**[访问字符](https://zh.javascript.info/string" \l "fang-wen-zi-fu)**

要获取在 pos 位置的一个字符，

①使用方括号 [pos]

②调用 [str.charAt(pos)](https://developer.mozilla.org/zh/docs/Web/JavaScript/Reference/Global_Objects/String/charAt) 方法。

let str = `Hello`;

// 第一个字符

str[0] // H

str.charAt(0) // H

// 最后一个字符

str[str.length - 1] // o

方括号是获取字符的一种现代化方法，而 charAt 是历史原因才存在的。

它们之间的唯一区别是：如果没有找到字符，[] 返回 undefined，而 charAt 返回一个空字符串：

let str = `Hello`;

str[1000] // undefined

str.charAt(1000) ); // ''（空字符串）

str[0]='A' //不起作用,需要用到字符串相关的方法来操作字符串

使用 for..of 遍历字符：

for (let char of "Hello") {

alert(char); // H,e,l,l,o（char 变为 "H"，然后是 "e"，然后是 "l" 等）

}

JavaScript 中没有 character 类型。

在一些语言中，单个字符有一个特殊的 “character” 类型，在 C 语言和 Java 语言中被称为 “char”。

在 JavaScript 中没有这种类型。只有一种 string 类型，一个字符串可以包含一个或多个字符。

**数字类型**

在现代 JavaScript 中，数字（number）有两种类型：

①JavaScript 中的常规数字以 64 位的格式 [IEEE-754](https://en.wikipedia.org/wiki/IEEE_754-2008_revision) 存储，也被称为“双精度浮点数”。这是我们大多数时候所使用的数字，。

②BigInt 数字，用于表示任意长度的整数。有时会需要它们，因为常规数字不能超过 253 或小于 -253。由于仅在少数特殊领域才会用到 BigInt

**[Number 类型](https://zh.javascript.info/types" \l "number-lei-xing)**

JS中所有的数值都是Number类型，包括整数和浮点数（小数）

数字可以有很多操作，比如，乘法 \*、除法 /、加法 +、减法 - 等等

除了常规的数字，还包括所谓的“特殊数值”也属于这种类型：Infinity、-Infinity 和 NaN

特殊的数值属于 “number” 类型。当然，对“特殊的数值”这个词的一般认识是，它们并不是数字。

-Infinity 代表数学概念中的 [无穷大](https://en.wikipedia.org/wiki/Infinity) ∞。是一个比任何数字都大的特殊值。

alert( 1 / 0 ); // Infinity

alert( Infinity ); // Infinity

-NaN 代表一个计算错误。它是一个不正确的或者一个未定义的数学操作所得到的结果：

alert( "not a number" / 2 ); // NaN，这样的除法是错误的

NaN 是粘性的。任何对 NaN 的进一步操作都会返回 NaN：

alert(NaN+1) //NaN

使用typeof检查一个NaN也会返回number

数字的最大值: Number.MAX\_VALUE //1.7976931348623157e+308

大于0的最小值:Number.MIN\_VALUE //5e-324

**舍入**（rounding）是使用数字时最常用的操作之一。

Math.floor(Num) 向下舍入：3.1 变成 3，-1.1 变成 -2。

Math.ceil(Num) 向上舍入：3.1 变成 4，-1.1 变成 -1。

Math.round(Num) 向最近的整数舍入：3.1 变成 3，3.6 变成 4，-1.1 变成 -1。

Math.trunc(Num)（IE 不支持）移除小数点后的所有内容而没有舍入：3.1 变成 3，-1.1 变成 -1。

**Num.****[toFixed(n)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed)** 将数字舍入到小数点后 n 位(有四舍五入)，并以字符串形式返回结果。

let num = 12.5;

Math.floor(num) //12

Math.ceil(num) //13

Math.ceil(num) //13

num.[toFixed(2)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) //12.50

可以使用一个运算符 typeof,来检查一个变量的类型

语法：typeof 变量

-检查字符串时，会返回string

-检查数值时，会返回number

在JS中整数的运算基本可以保证精确

如果使用JS进行浮点运算，可能得到一个不精确的结果，千万不要使用JS进行对精确度要求比较高的运算

0.1 + 0.2 //0.30000000000000004

解决：①借助方法 [toFixed(n)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number/toFixed) 对结果进行舍入： (0.1+0.2).toFixed(2)

②将数字临时乘以 100（或更大的数字），将其转换为整数，进行数学运算，然后再除回。当我们使用整数进行数学运算时，误差会有所减少，但仍然可以在除法中得到：

(0.1 \* 10 + 0.2 \* 10) / 10 //0.3

(0.28 \* 100 + 0.14 \* 100) / 100 // 0.4200000000000001

不仅仅是 JavaScript许多其他编程语言也存在同样的问题。

PHP，Java，C，Perl，Ruby 给出的也是完全相同的结果，因为它们基于的是相同的数字格式。

数字内部表示的另一个有趣结果是存在两个零：0 和 -0。

0===-0 //true

9999999999999999 // 显示 10000000000000000

9999999999999999 ===10000000000000000

length：16 length：17

精度损失。有 64 位来表示该数字，其中 52 位可用于存储数字，但这还不够。所以最不重要的数字就消失了。

**[isFinite](https://zh.javascript.info/number" \l "ce-shi-isfinite-he-isnan)** [和](https://zh.javascript.info/number" \l "ce-shi-isfinite-he-isnan) **[isNaN](https://zh.javascript.info/number" \l "ce-shi-isfinite-he-isnan)** 函数

-Infinity（和 -Infinity）是一个特殊的数值，比任何数值都大（小）。

-NaN 代表一个 error。

它们属于 number 类型，但不是“普通”数字，因此，这里有用于检查它们的特殊函数：

isNaN(value) 将其参数转换为数字，然后测试它是否为 NaN：

isNaN(NaN) // true

isNaN("str") // true

值 “NaN” 是独一无二的，它不等于任何东西，包括它自身，所以不能使用 === NaN 比较：

NaN === NaN // false

isFinite(value)将其参数转换为数字，如果是常规数字，则返回 true，而不是 NaN/Infinity/-Infinity：

有时 isFinite 被用于验证字符串值是否为**常规**数字：

let num = +prompt("Enter a number", '');

alert( isFinite(num) );

// 结果会是 true，除非你输入的是 Infinity、-Infinity 或不是数字

请注意，在所有数字函数中，包括 isFinite，空字符串或仅有空格的字符串均被视为 0。

isNaN('') //false

**===与 Object.is() 进行比较**

有一个特殊的内建方法 [Object.is](https://developer.mozilla.org/zh/docs/Web/JavaScript/Reference/Global_Objects/Object/is)，它类似于 === 一样对值进行比较，但它对于两种边缘情况更可靠：

①用于 NaN： Object.is(NaN,NaN)=== true

②值 0 和 -0 是不同的：Object.is（0，-0）=== false，

从技术上讲这是对的，因为在内部，数字的符号位可能会不同，即使其他所有位均为零。

在所有其他情况下，Object.is(a，b) 与 a === b 相同。

这种比较方式经常被用在 JavaScript 规范中。当内部算法需要比较两个值是否完全相同时，它使用 Object.is（内部称为 [SameValue](https://tc39.github.io/ecma262/" \l "sec-samevalue)）。

**[parseInt() 和 parseFloat](https://zh.javascript.info/number" \l "parseint-he-parsefloat)()**

使用加号 + 或 Number() 的数字转换是严格的。如果一个值不完全是一个数字，就会失败：

+"100px" //NaN

唯一的例外是字符串开头或结尾的空格，因为它们会被忽略。

但在现实生活中，我们经常会有带有单位的值，例如 CSS 中的 "100px" 或 "12pt"。并且，在很多国家，货币符号是紧随金额之后的，所以我们有 "19€"，并希望从中提取出一个数值。

parseInt 和 parseFloat 的作用:可以从字符串中“读取”数字，直到无法读取为止。如果发生 error，则返回收集到的数字。函数 parseInt 返回一个整数，而 parseFloat 返回一个浮点数：

parseInt('100px') //100

parseFloat('12.5em') // 12.5

当没有数字可读时会发生这种情况, parseInt/parseFloat 会返回 NaN：

parseInt('a123') // NaN，第一个符号停止了读取

**[其他数学函数](https://zh.javascript.info/number" \l "qi-ta-shu-xue-han-shu)**

JavaScript 有一个内建的 [Math](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Math) 对象，它包含了一个小型的数学函数和常量库。

Math.random() 返回一个从 0 到 1 的随机数（不包括 1）

Math.max(a, b, c...) / Math.min(a, b, c...) 从任意数量的参数中返回最大/最小值。

Math.pow(n, power) 返回 n 的给定（power）次幂

A+=B //A=A+B JS加法运算

A=+B //B转位数字，赋值给A

**[BigInt 类型](https://zh.javascript.info/types" \l "bigint-lei-xing)**

在 JavaScript 中，“number” 类型无法代表大于 253（或小于 -253）的整数，这是其内部表示形式导致的技术限制。这大约是 16 位的十进制数字，因此在大多数情况下，这个限制不是问题，但有时我们需要很大的数字，例如用于加密或微秒精度的时间戳。

BigInt 类型是最近被添加到 JavaScript 语言中的，是一种特殊的数字类型，用于表示任意长度的整数。

创建 bigint 的方式有两种：

①在一个整数字面量后面加 n

②调用 BigInt 函数，该函数从字符串、数字等中生成 bigint。

const bigint = 1234567890123456789012345678901234567890n;// 尾部的 "n"表示这是一个 BigInt 类型

const sameBigint = BigInt("1234567890123456789012345678901234567890");

const bigintFromNumber = BigInt(10); // 与 10n 相同

**[数学运算符](https://zh.javascript.info/bigint" \l "shu-xue-yun-suan-fu),**BigInt 大多数情况下可以像常规数字类型一样使用，例如：

1n + 2n // 3n

typeof(3n) //"bigint"

对 bigint 的所有操作，返回的结果也是 bigint。

3n/2n // 1n bigint 的除法总是返回 bigint

不可以把 bigint 和常规数字类型混合使用：

1n + 2 // Error: Cannot mix BigInt and other types

如果有需要，我们应该显式地转换它们：使用 **BigInt()** 或者 **Number()**，像这样：

let bigint = 1n;

let number = 2;

// 将 number 转换为 bigint

bigint + BigInt(number) // 3

// 将 bigint 转换为 number

Number(bigint) + number// 3

转换操作始终是静默的，绝不会报错，但是如果 bigint 太大而数字类型无法容纳，则会截断多余的位，因此我们应该谨慎进行此类转换。

BigInt 不支持一元加法

一元加法运算符 +value，是大家熟知的将 value 转换成数字类型的方法。

+1n //error

**比较运算符**，例如 < 和 >，使用它们来对 bigint 和 number 类型的数字进行比较没有问题：

2n > 1n // true

2n > 1 // true

由于 number 和 bigint 属于不同类型:

① == 比较时相等，

② ===（严格相等）比较时不相等

10n==10 //true

10n===10 //false

**[布尔运算](https://zh.javascript.info/bigint" \l "bu-er-yun-suan)**,当在 if 或其他布尔运算中时，bigint 的行为类似于 number。

在 if 中，bigint 0n 为 false，其他值为 true：

if (0n) {

// 永远不会执行

}

||，&& 和其他运算符，处理 bigint 的方式也类似于 number：

1n || 2 // 1（1n 被认为是 true）

0n || 2 // 2（0n 被认为是 false）

兼容性问题: 目前 Firefox 和 Chrome 已经支持 BigInt 了，但 Safari/IE/Edge 还没有。

**[Boolean 类型](https://zh.javascript.info/types" \l "boolean-lei-xing-luo-ji-lei-xing)**[（逻辑类型）](https://zh.javascript.info/types" \l "boolean-lei-xing-luo-ji-lei-xing) 布尔值

boolean 类型仅包含两个值：true 和 false。

使用typeof检查一个布尔值时，会返回boolean

typeof true //"boolean"

**[null 类型](https://zh.javascript.info/types" \l "null-zhi)** 的值只有一个，就是null

特殊的 null 值不属于上述任何一种类型，它构成了一个独立的类型，只包含 null 值

null这个值专门用来表示一个为空的对象

使用typeof检查一个null值时，会返回object

typeof null //"object"

typeof null 的结果是 "object"。这其实是不对的。官方也承认了这是 typeof 运算符的问题，现在只是为了兼容性而保留了下来。当然，null 不是一个 object。null 有自己的类型，它是一个特殊值。再次强调，这是 JavaScript 语言的一个错误。

相比较于其他编程语言，JavaScript 中的 null 不是一个“对不存在的 object 的引用”或者 “null 指针”。

JavaScript 中的 null 仅仅是一个代表“无”、“空”或“值未知”的特殊值。

**[undefined 类型](https://zh.javascript.info/types" \l "undefined-zhi)** 的值只有一个，就undefind

特殊值 undefined 和 null 一样自成类型。undefined 的含义是 未被赋值。

如果一个变量已被声明，但未被赋值，那么它的值就是 undefined：

let x

console.log(x) //undefined

使用typeof检查一个undefined时也会返回undefined

let x

typeof x //"undefined"

typeof undefined //"undefined"

①null 表示一个“空”或者“未知”的值

②undefined 只是用来检查像变量是否被赋值了这类操作

**[typeof 运算符](https://zh.javascript.info/types" \l "type-typeof)**

typeof 运算符返回参数的类型

①作为运算符：typeof x。

②函数形式：typeof(x)。

换言之，有括号和没有括号，得到的结果是一样的。

对 typeof x 的调用会以字符串的形式返回数据类型：

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)

①typeof 判断类型

②instanceof 判断对象 A instanceof B （A是否B 的实例，B为对象）

①函数 A(x)

②方法 x.A()

**类型转换**

**显式转换**：使用Number、String和Boolean等构造函数，手动将各种类型的值，转换成数字、字符串或者布尔值。

**隐式转换**：JavaScript中的某些运算符会做隐式的类型转换

x+"" // 等价于 String(x) 如88 + '6' => ’886’

+x // 等价于 Number(x).也可以写成x-0 如+'886' => number类型的886

!!x // 等价于Boolean(x) 如 !!'886' => true

**[字符串转换](https://zh.javascript.info/type-conversions" \l "zi-fu-chuan-zhuan-huan)：**

①toString()方法，不会影响到原变量，会将转换的结果返回（null和undefined不适用）

②String()函数，不会影响到原变量，会将转换的结果返回

**数字型转换**：

①使用Number()函数

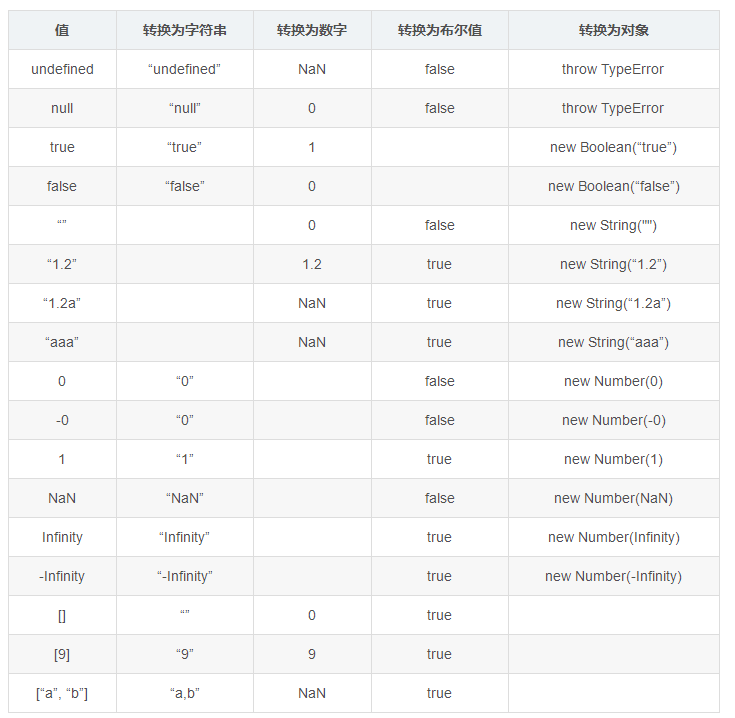
字符串 --> 数字

1.纯数字的字符串，则直接将其转换为数字

2.字符串中有非数字的内容，则转换为NaN

3.字符串是一个空串或者是一个全是空格的字符串，则转换为0

Number("123z") // NaN（从字符串“读取”数字，读到 "z" 时出现错误）

②专门转换字符串parseInt()和parseFloat()，如果对非String使用parseInt()或parseFloat()

它会先将其转换为String然后在操作

对 undefined 进行数字型转换时，输出结果为 NaN，而非 0。

**布尔型转换：**

使用 Boolean()函数：

①数字 ---> 布尔

除了0和NaN，其余的都是true

②字符串 ---> 布尔

除了空串，其余的都是true null和undefined都会转换为false 对象也会转换为true

对 "0" 和只有空格的字符串（比如：" "）进行布尔型转换时，输出结果为 true。

**运算符**也叫操作符,通过运算符可以对一个或多个值进行运算,并获取运算结果

typeof就是运算符，可以来获得一个值的类型

它会将该值的类型以字符串的形式返回 number string boolean undefined object

**算数运算符,**当对非Number类型的值进行运算时，会将这些值转换为Number然后在运算

任何值和NaN做运算都得NaN

**+**

①对两个值进行加法运算，并将结果返回

②对两个字符串进行加法运算，则会做拼串，会将两个字符串拼接为一个字符串，并返回

③任何的值和字符串做加法运算，都会先转换为字符串，然后再和字符串做拼串的操作

123 + "4" //"1234"

利用这一特点，来将一个任意的数据类型转换为String

我们只需要为任意的数据类型 + 一个 "" 即可将其转换为String

这是一种隐式的类型转换，由浏览器自动完成，实际上它也是调用String()函数

123+ "" // "123"

**-** 可以对两个值进行减法运算，并将结果返回

**\*** 可以对两个值进行乘法运算

**/** 可以对两个值进行除法运算

**%**  取模运算（取余数）

任何值做 - \* / 运算时都会自动转换为Number,我们可以利用这一特点做隐式的类型转换

通过为一个值 -0 \*1 /1 来将其转换为Number,原理和Number()函数一样，使用起来更加简单

"123"-0 // 123

typeof ("123"-0 ) // "number"

A+=B //A=A+B JS加法运算

A=+B //B转位数字，赋值给A

**一元运算符**，只需要一个操作数

+ 正号,正号不会对数字产生任何影响

- 负号,负号可以对数字进行负号的取反

对于非Number类型的值，它会将先转换为Number，然后在运算,可以对一个其他的数据类型使用+,

来将其转换为number,它的原理和Number()函数一样

A=+B //B转位数字，赋值给A

**++ 自增**

通过自增可以使变量在自身的基础上增加1, 对于一个变量自增以后，原变量的值会立即自增1

自增分成两种：①后++ (a++) ②前++ (++a)

同： a++ 还是 ++a，都会立即使原变量的值自增1

不同：a++ 和 ++a的值不同

① a++的值等于原变量的值（自增前的值）

② ++a的值等于新值 （自增后的值）

let a =1

a++ //1

console.log(a) //2

++a //3

console.log(a) //3

**--自减**

通过自减可以使变量在自身的基础上减1

自减分成两种：①后--(a--)

②前--(--a)

同： 都会立即使原变量的值自减1

不同：值不同

① a-- 是变量的原值 （自减前的值）

② --a 是变量的新值 （自减以后的值）

**逻辑运算符**

JavaScript 里有三个逻辑运算符：||（或），&&（与），!（非）。

**! 非**

!可以用来对一个值进行非运算，对一个布尔值进行取反操作， true变false，false变true

运算符接受一个参数，并按如下运作：

①将操作数转化为布尔类型：true/false。

②返回相反的值。

如果对一个值进行两次取反，它不会变化

如果对非布尔值进行元素，则会将其转换为布尔值，然后再取反

所以我们可以利用该特点，来将一个其他的数据类型转换为布尔值

可以为一个任意数据类型取两次反，来将其转换为布尔值，原理和Boolean()函数一样

**&& 与**

&&可以对符号两侧的值进行与运算并返回结果

运算规则：

①两个值中只要有一个值为false就返回false，只有两个值都为true时，才会返回true

②JS中的“与”属于短路的与，如果第一个值为false，则不会看第二个值

**|| 或**

||可以对符号两侧的值进行或运算并返回结果

运算规则：

①两个值中只要有一个true，就返回true，如果两个值都为false，才返回false

②JS中的“或”属于短路的或，如果第一个值为true，则不会检查第二个值

**&& || 非布尔值的情况：**

对于非布尔值进行与或运算时，会先将其转换为布尔值，然后再运算，并且返回原值

&&与运算： ①如果第一个值为true，则必然返回第二个值

②如果第一个值为false，则直接返回第一个值

|| 或运算： ①如果第一个值为true，则直接返回第一个值

②如果第一个值为false，则返回第二个值

**== 相等运算符**用来比较两个值是否相等，如果相等会返回true，否则返回false

使用 == 来做相等运算，当使用==来比较两个值时，如果值的类型不同，则会自动进行类型转换，将其转换为相同的类型，然后在比较。

**!= 不相等**用来判断两个值是否不相等，如果不相等返回true，否则返回false

使用 != 来做不相等运算，不相等也会对变量进行自动的类型转换，如果转换后相等它也会返回false

**===全等**

用来判断两个值是否全等，不会做自动的类型转换，如果两个值的类型不同，直接返回false

**!== 不全等**

用来判断两个值是否不全等，不会做自动的类型转换，如果两个值的类型不同，直接返回true

**三元运算符** (条件运算符)

语法： 条件表达式?语句1:语句2;

执行的流程：条件运算符在执行时，首先对条件表达式进行求值，如果该值为true，则执行语句1，并返回执行结果,如果该值为false，则执行语句2，并返回执行结果,如果条件的表达式的求值结果是一个非布尔值，会将其转换为布尔值然后在运算

a > b ? alert("a大"):alert("b大");

**运算符的优先级**

**,** 运算符 可以分割多个语句，一般可以在声明多个变量时使用,

①使用,运算符同时声明多个变量

var a , b , c;

②可以同时声明多个变量并赋值

var a=1 , b=2 , c=3;

就和数学中一样，在JS中运算符也有优先级，比如：先乘除 后加减

在JS中有一个运算符优先级的表，在表中越靠上优先级越高，优先级越高越优先计算，如果优先级一样，则从左往右计算。

但是这个表我们并不需要记忆，如果遇到优先级不清楚，可以使用()来改变优先级

**流程控制语句**

JS中的程序是从上到下一行一行执行的, 通过流程控制语句可以控制程序执行流程，使程序可以根据一定的条件来选择执行

语句的分类：1.条件判断语句 2.条件分支语句 3.循环语句

**条件判断语句(**if语句**)**：使用条件判断语句可以在执行某个语句之前进行判断，如果条件成立才会执行语句，条件不成立则语句不执行。

语法一：

if(条件表达式){

语句...

}

if语句在执行时，会先对条件表达式进行求值判断，如果条件表达式的值为true，则执行if后的语句，如果条件表达式的值为false，则不会执行if后的语句。

语法二:

if(条件表达式){

语句...

}else{

语句...

}

if...else...语句,当该语句执行时，会先对if后的条件表达式进行求值判断，如果该值为true，则执行if后的语句,如果该值为false，则执行else后的语句

语法三：

if(条件表达式){

语句...

}else if(条件表达式){

语句...

}else if(条件表达式){

语句...

}else{

语句...

}

if...else if...else当该语句执行时，会从上到下依次对条件表达式进行求值判断,如果值为true，则执行当前语句。如果值为false，则继续向下判断。如果所有的条件都不满足，则执行最后一个else后的语句

该语句中，只会有一个代码块被执行，一旦代码块执行了，则直接结束语句

**条件分支语句**（switch语句）

语法：

switch(条件表达式){

case 表达式:

语句...

break;

case 表达式:

语句...

break;

default:

语句...

break;

}

执行流程：switch...case..语句

在执行时会依次将case后的表达式的值和switch后的条件表达式的值进行全等比较， 如果比较结果为true，则从当前case处开始执行代码。当前case后的所有的代码都会执行，我们可以在case的后边跟着一个break关键字，这样可以确保只会执行当前case后的语句，而不会执行其他的case。如果比较结果为false，则继续向下比较。如果所有的比较结果都为false，则只执行default后的语句。

switch语句和if语句的功能实际上有重复的，使用switch可以实现if的功能，同样使用if也可以实现switch的功能，所以我们使用时，可以根据自己的习惯选择。

**循环语句**：通过循环语句可以反复的执行一段代码多次

跳出循环有三种方法：break 、 continue 和 return

**while循环**

语法：

while(条件表达式){

语句...

}

while语句在执行时，先对条件表达式进行求值判断，如果值为true，则执行循环体，循环体执行完毕以后，继续对表达式进行判断，如果为true，则继续执行循环体，以此类推，如果值为false，则终止循环

**do...while循环**

语法：

do{

语句...

}while(条件表达式)

执行流程：do...while语句在执行时，会先执行循环体，循环体执行完毕以后，在对while后的条件表达式进行判断，如果结果为true，则继续执行循环体，执行完毕继续判断以此类推，如果结果为false，则终止循环

实际上这两个语句功能类似，不同的是while是先判断后执行，而do...while会先执行后判断，

do...while可以保证循环体至少执行一次，而while不能

**for循环**

for语句，也是一个循环语句，也称为for循环

在for循环中，三个表达式：1.初始化表达式2.条件表达式3.更新表达式

for循环的语法：

for(①初始化表达式;②条件表达式;④更新表达式){

③语句...

}

for循环的执行流程：

①执行初始化表达式，初始化变量（初始化表达式只会执行一次）

②执行条件表达式，判断是否执行循环。如果为true，则执行循环

③如果为false，终止循环

④执行更新表达式，更新表达式执行完毕继续重复②

JS中数据类型：String、 Number、 Boolean、Null、Undefined （Symbol、BigInt）

以上这五种类型属于基本数据类型，以后我们看到的值，只要不是上边的5种，全都是对象

**Object 对象**

它是一种复合值，它将很多值聚合到一起，可以通过名字访问这些值。

• 对象也可以看做是属性的无序集合，每个属性都是一个键值对。

• 对象除了可以创建自有属性，还可以通过从 原型的对象 继承属性。

\* 如果使用基本数据类型的数据，我们所创建的变量都是独立，不能成为一个整体。

\* 对象属于一种复合的数据类型，在对象中可以保存多个不同数据类型的属性。

对象的分类：

1.内建对象： 由ES标准中定义的对象，在任何的ES的实现中都可以使用

比如：Math String Number Boolean Function Object Array Date....

2.宿主对象：由JS的运行环境提供的对象，目前来讲主要指由浏览器提供的对象

比如： BOM DOM

3.自定义对象：由开发人员自己创建的对象

类的实例、prototype、\_\_proto\_\_、函数也是对象（万物皆对象）

**创建对象**

①使用new关键字调用的函数，是构造函数constructor，构造函数是专门用来创建对象的函数

使用typeof检查一个对象时，会返回object

var obj = new Object();

typeof obj //"object"

②使用对象字面量来创建一个对象

   var obj = {};

**对象的属性**:在对象中保存的值

向对象添加属性：

①对象.属性名 = 属性值 （如果要使用特殊的属性名，不能采用.的方式来操作）

②对象["属性名"]=属性值 （推荐）

③Object.assign(obj1,obj2)

④...扩展运算符

let obj ={}

obj.name="a"

obj['old']=10

Object.assign(obj,{sex:"男"})

读取对象中的属性:

①对象.属性名

②对象["属性名"]

如果读取对象中没有的属性，不会报错而是会返回undefined

修改对象的属性值：对象.属性名 = 新值

删除对象的属性： delete 对象.属性名

遍历对象：for(let key in obj) 循环。

in 运算符，可以检查一个对象中是否含有指定的属性，如果有则返回true，没有则返回false

语法："属性名" in 对象

console.log("test2" in obj);

**Map和Set**

JavaScript的默认对象表示方式{},即一组键值对,键必须是字符串。但实际上Number或者其他数据类型作为键也是非常合理的。为了解决这个问题，最新的ES6规范引入了新的数据类型Map

**Map**

Map是一组键值对的结构，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用Array实现，需要两个Array：

var names = ['Michael', 'Bob', 'Tracy'];

var scores = [95, 75, 85];

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，Array越长，耗时越长。

如果用Map实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。

用JavaScript写一个Map如下：

var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);

m.get('Michael'); // 95

初始化Map需要一个二维数组，或者直接初始化一个空Map。

Map具有以下方法：

var m = new Map(); // 空Map

m.set('Adam', 67); // 添加新的key-value

m.set('Bob', 59);

m.has('Adam'); // 是否存在key 'Adam': true

m.get('Adam'); // 67

m.delete('Adam'); // 删除key 'Adam'

m.get('Adam'); // undefined

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

var m = new Map();

m.set('Adam', 67);

m.set('Adam', 88);

m.get('Adam'); // 88

**Set**

Set和Map类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在Set中，没有重复的key。(可以利用这个特性 数组去重)

要创建一个Set，需要提供一个Array作为输入，或者直接创建一个空Set：

var s1 = new Set(); // 空Set

var s2 = new Set([1, 2, 3]); // 含1, 2, 3

重复元素在Set中自动被过滤：

var s = new Set([1, 2, 3, 3, '3']);

s; // Set {1, 2, 3, "3"}

注意数字3和字符串'3'是不同的元素。

通过add(key)方法可以添加元素到Set中，可以重复添加，但不会有效果：

s.add(4);

s; // Set {1, 2, 3, 4}

s.add(4);

s; // 仍然是 Set {1, 2, 3, 4}

通过delete(key)方法可以删除元素：

var s = new Set([1, 2, 3]);

s; // Set {1, 2, 3}

s.delete(3);

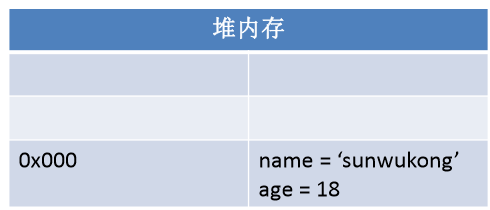
s; // Set {1, 2}

**栈和堆**

• JavaScript在运行时数据是保存到栈内存和堆内存当中的。

• 简单来说栈内存用来保存变量和基本类型。堆内存用来保存对象。

• 我们在声明一个变量时实际上就是在栈内存中创建了一个空间用来保存变量。

• 如果是基本类型则在栈内存中直接保存，

• 如果是引用类型则会在堆内存中保存，变量中保存的实际上对象在堆内存中的地址

var a = 123;

var b = true;

var c = "hello";

var d = {name:'sunwukong',age:18};

JS中的变量都是保存到栈内存中的，基本数据类型的值直接在栈内存中存储，值与值之间是独立存在，修改一个变量不会影响其他的变量。

对象是保存到堆内存中的，每创建一个新的对象，就会在堆内存中开辟出一个新的空间，而变量保存的是对象的内存地址（对象的引用），如果两个变量保存的是同一个对象引用，当一个通过一个变量修改属性时，另一个也会受到影响。

当比较两个基本数据类型的值时，就是比较值。而比较两个 引用数据类型(Object) 时，它是比较的对象的内存地址，如果两个对象是一摸一样的，但是地址不同，它也会返回false

①变量(栈存值)

②对象(栈存内存地址)

创建一个对象：

①使用new关键字创建对象

var obj = new Object();

②使用对象字面量来创建一个对象

var obj = {};

使用对象字面量，可以在创建对象时，直接指定对象中的属性

语法：{属性名:属性值,属性名:属性值....}

对象字面量的属性名可以加引号也可以不加，建议不加,如果要使用一些特殊的名字，则必须加引号

属性名和属性值是一组一组的 键值对 结构，名和值之间使用 : 连接，多个键值对之间使用,隔开

如果一个属性之后没有其他的属性了，就不要写,

var obj2 = {

name:"猪八戒",

age:13,

gender:"男",

test:{name:"沙僧"}

};

**函数** （①新参赋值 ②变量提升）

通常情况下，JavaScript 代码是自上而下执行的，不过函 数体内部的代码则不是这样。如果只是对函数进行了声明， 其中的代码并不会执行。只有在调用函数时才会执行函数 体内部的代码。

**函数 function**

- 函数也是一个对象

- 函数中可以封装一些功能（代码），在需要时可以执行这些功能（代码）

- 函数中可以保存一些代码在需要的时候调用

- 使用typeof检查一个函数对象时，会返回function

**创建函数：**

①函数声明

function 函数名([形参1,形参2...形参N]){

语句...

}

②函数表达式

var 函数名 = function([形参1,形参2...形参N]){

语句....

}

我们在实际开发中很少使用构造函数来创建一个函数对象

①通过new关键字创建一个函数对象

②可以将要封装的代码以字符串的形式传递给构造函数

var fun = new Function("console.log('Hello 这是我的第一个函数');");

//封装到函数中的代码不会立即执行

//函数中的代码会在函数调用的时候执行

调用函数 语法：**函数对象()**

//当调用函数时，函数中封装的代码会按照顺序执行

fun();

**传递参数**

JS中的所有的参数传递都是按值传递的。 也就是说把函数外部的值赋值给函数内部的参数，就和把值从一个变量赋值给另一个变量是一样的

可以在函数的()中来指定一个或多个形参（形式参数）,多个形参之间使用","隔开,声明形参就相当于在函数内部声明了对应的变量,但是并不赋值

定义一个用来求两个数和的函数:

function sum(a,b){

console.log(a+b);

}

在调用函数时，可以在()中指定实参（实际参数）,实参将会赋值给函数中对应的形参

sum(1,2);

调用函数时解析器不会检查实参的类型,所以要注意，是否有可能会接收到非法的参数，如果有可能则需要对参数进行类型的检查

函数的实参可以是任意的数据类型

sum(123,"hello");

调用函数时，解析器也不会检查实参的数量,多余实参不会被赋值,如果实参的数量少于形参的数量，则没有对应实参的形参将是undefined

sum(123,456,"hello",true,null);

**返回值,**可以使用 return 来设置函数的返回值

语法： return 值

在函数中return后的语句都不会执行，使用return可以结束整个函数

**立即执行函数**

函数定义完，立即被调用，这种函数叫做立即执行函数

立即执行函数往往只会执行一次

(function(){

alert("我是一个匿名函数~~~");

}) ( );

函数也可以称为对象的属性，如果一个函数作为一个对象的属性保存，那么我们称这个函数是这个对象的方法

调用这个函数就说调用对象的方法（method）

var obj = {

name:"猪八戒",

age:18,

sayName:function(){

console.log(obj.name);

}

};

obj.sayName(); //猪八戒

**作用域**

作用域指一个变量的作用的范围

在JS中一共有两种作用域：

1.**全局作用域**

- 直接编写在script标签中的JS代码，都在全局作用域

- 全局作用域在页面打开时创建，在页面关闭时销毁

- 在全局作用域中有一个全局对象window，它代表的是一个浏览器的窗口，它由浏览器创建我们可以直接使用

- 在全局作用域中：

①创建的变量都会作为window对象的属性保存

②创建的函数都会作为window对象的方法保存

- 全局作用域中的变量都是全局变量，在页面的任意的部分都可以访问的到

2.**函数作用域**

变量的声明提前：

- 使用var关键字声明的变量，会在所有的代码执行之前被声明（但是不会赋值，只是声明），但是如果声明变量时不使用var关键字，则变量不会被声明提前

函数的声明提前：

- 使用函数声明形式创建的函数 function 函数(){}，它会在所有的代码执行之前就被创建，所以我们可以在函数声明前来调用函数

-使用函数表达式创建的函数，不会被声明提前，所以不能在声明前调用

 //函数声明，会被提前创建：

fun();

            function fun(){

                console.log("我是一个fun函数");

            }

//函数表达式，不会被提前创建：

            var fun2 = function(){

                console.log("我是fun2函数");

            };

            fun2();

函数作用域

- 调用函数时创建函数作用域，函数执行完毕以后，函数作用域销毁

- 每调用一次函数就会创建一个新的函数作用域，他们之间是互相独立的

- 在函数作用域中可以访问到全局作用域的变量，在全局作用域中无法访问到函数作用域的变量

（- 当在函数作用域操作一个变量时，它会先在自身作用域中寻找，如果有就直接使用，如果没有则向上一级作用域中寻找，直到找到全局作用域，如果全局作用域中依然没有找到，则会报错ReferenceError）

- 在函数中要访问全局变量可以使用window对象在函数作用域也有声明提前的特性，使用var关键字声明的变量，会在函数中所有的代码执行之前被声明，函数声明也会在函数中所有的代码执行之前执行

-在函数中，不使用var声明的变量都会成为全局变量 { i=0;}

function fun(){

  //A没有使用var关键字，则会设置为全局变量

A = 10;

console.log(A); //10

console.log(window.A) //10

      }

 fun();

定义形参就相当于在函数作用域中声明了变量

function fun(e){

                alert(e); //undefined

 }

fun();

var a=123;

function fun(){

alert(a); //undfinded (首先在自身作用域里面找a变量)

var a=456; //var 变量提前声明

}

fun();

alert(a;) //123

解析器在调用函数每次都会向函数内部传递进一个隐含的参数 this ，this指向的是一个对象，这个对象我们称为函数执行的 上下文对象，根据函数的调用方式的不同，this会指向不同的对象:

1.以函数的形式调用时，this永远都是window // XX()

2.以方法的形式调用时，this就是调用方法的那个对象 // A.XX()

3.构造函数式调用，this就是新创建的对象

4.箭头函数，数会捕获其所在上下文的this值，作为自己的this值

(由其外部的函数决定，外部函数this就是箭头函数的this)

5.改变函数this的指向： call、apply

function.call(thisArg, arg1, arg2, ...)

function.apply(thisArg, [argsArray])

第一个参数就是this的指向

在严格模式下和非严格模式下this表现不同:

'use strict';

var a = 1;

function fun() {

var a = 2;

return this.a;

}

fun(); //报错 Cannot read property 'a' of undefined

说明：严格模式下，this指向undefined;

var a = 1;

function fun() {

var a = 2;

return this.a;

}

fun();//1

说明：非严格模式下this指向window;

结论：当函数独立调用的时候，在严格模式下它的this指向undefined，在非严格模式下，当this指向undefined的时候，自动指向全局对象(浏览器中就是window)

**当this不在函数中用的时候：**

var a = 1000;

var obj = {

a: 1,

b: this.a + 1

}

console.log(obj.b);//1001

当obj在全局声明的时候，obj内部属性中的this指向全局对象

var a = 1;

var obj = {

a: 2,

b: function() {

return this.a;

}

}

console.log(obj.b())//2

说明：当函数以方法形式调用时，this指向调用的对象

**箭头函数this的指向（外部函数的this就是箭头函数的this）：**

var a = 1;

var obj = { a: 2 };

function fun() {

var a = 3;

let f = () => console.log(this.a);

//箭头函数在fun函数里面，（箭头函数的this=fun函数的this）

f();

};

fun(); //1 以函数调形式用this是window,

fun.call(obj); //2 this指向obj

**工厂方法**

工厂模式，即：把实现同一事情的相同代码，放到一个函数中，以后如果再想实现这个功能，就不需要重新编写这些代码了，只要执行当前的函数即可，这就是**函数的封装**，体现了**高内聚、低耦合**的思想：减少页面的中的冗余代码，提高代码的重复利用率

使用工厂方法创建对象，通过该方法可以大批量的创建对象

function createPerson(name , age ,gender){

//①创建一个新的对象

var obj = new Object();

//②向对象中添加属性

obj.name = name;

obj.age = age;

obj.gender = gender;

obj.sayName = function(){

alert(this.name);

};

//③将新的对象返回

return obj;

}

var obj = createPerson("A",28,"男"); //{name: "A", age: 28, gender: "男", sayName: ƒ}

obj.sayName() //"A"

使用工厂方法创建的对象，使用的构造函数都是Object,所以创建的对象都是Object这个类型，就导致我们无法区分出多种不同类型的对象

**回调函数**（一个函数被作为参数传递给另一个函数）

就是一个参数，将这个函数作为参数传到另一个函数里面，当那个函数执行完之后，再执行传进去的这个函数。这个过程就叫做回调。

回调，就是回头调用的意思。主函数的事先干完，回头再调用传进来的那个函数

①你定义了 ②你没调用 ③但最终执行了

**构造函数**

构造函数就是一个普通的函数，创建方式和普通函数没有区别, 不同的是构造函数习惯上首字母大写

构造函数和普通函数的区别就是调用方式的不同：

普通函数是直接调用，而构造函数需要使用new关键字来调用

构造函数的执行流程：

1.立刻创建一个新的对象（一出现new）

2.将新建的对象设置为函数中this,在构造函数中可以使用this来引用新建的对象

3.逐行执行函数中的代码

4.将新建的对象作为返回值返回（后台直接返回）

function Person(name , age , gender){

this.name = name;

this.age = age;

this.gender = gender;

this.sayName = function(){

alert(this.name);

};

}

var per = new Person("孙悟空",18,"男");

使用同一个构造函数创建的对象，我们称为一类对象，也将一个构造函数称为一个类。

我们将通过一个构造函数创建的对象，称为是该类的实例

**instanceof** 判断一个对象是否是一个类的实例

语法： 对象 instanceof 构造函数 (返回布尔值)

console.log(per instanceof Person);

**in** 判断属性是否在对象中/下标是否在数组中

语法： 属性名 in 对象

**obj.hasOwnProperty()** 判断对象自身是否有该属性

**原型 prototype**

\_\_proto\_\_ 原型链是任何对象(除null)都有的属性

prototype 函数对象才有

我们所创建的每一个函数，解析器都会向函数中添加一个属性prototype,这个属性对应着一个对象，这个对象就是原型对象

- 如果函数作为普通函数调用prototype没有任何作用

- 当函数以构造函数的形式调用时，它所创建的对象中都会有一个隐含的属性，指向该构造函数的

原型对象，我们可以通过\_\_proto\_\_来访问该属性

\_\_proto\_\_ == constuctor.prototype

原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，我们可以将对象中共有的内容，统一设置到原型对象中。

以后我们创建构造函数时，可以将这些对象共有的属性和方法，统一添加到构造函数的原型对象中，

这样不用分别为每一个对象添加，也不会影响到全局作用域，就可以使每个对象都具有这些属性和方法了

function MyClass() {

}

//向MyClass的原型中添加属性a

MyClass.prototype.a = 123;

/向MyClass的原型中添加一个方法

MyClass.prototype.sayHello = function () {

alert("hello");

};

var mc = new MyClass();

var mc2 = new MyClass();

原型对象也是对象，所以它也有原型，当我们使用一个对象的属性或方法时，会现在自身中寻找，自身中如果有，则直接使用， 如果没有则去原型对象中寻找，如果原型对象中有，则使用，如果没有则去原型的原型中寻找,直到找到Object对象的原型，

 Object对象的原型没有原型，如果在Object原型中依然没有找到，则返回undefined

**[从原型到原型链](https://segmentfault.com/a/1190000008959943)**

- \_\_proto\_\_ 原型链是任何对象(除null)都有的属性

- prototype 函数对象才有

构造函数创建对象

function Person() {

}

var person = new Person();

person.name = 'Kevin';

console.log(person.name) // Kevin

Person 就是一个构造函数，我们使用 new 创建了一个实例对象 person。

**prototype**

每个函数都有一个 prototype 属性

function Person() {

}

// prototype是函数才会有的属性

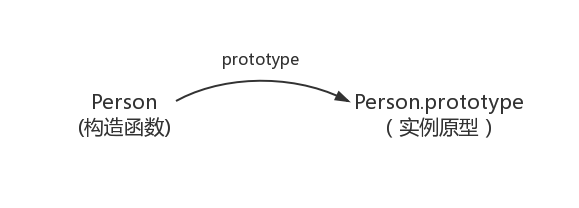
Person.prototype.name = 'Kevin';

var person1 = new Person();

var person2 = new Person();

console.log(person1.name) // Kevin

console.log(person2.name) // Kevin



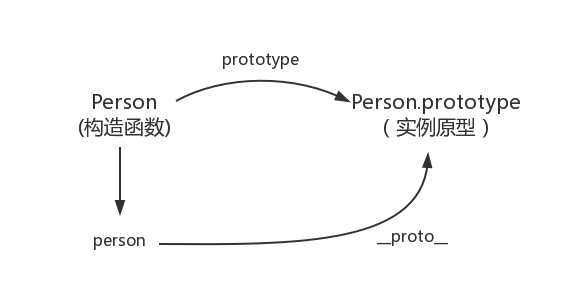
**\_\_proto\_\_**

对象(除了 null )都具有的一个属性，叫\_\_proto\_\_，这个属性会指向该对象的原型

function Person() {

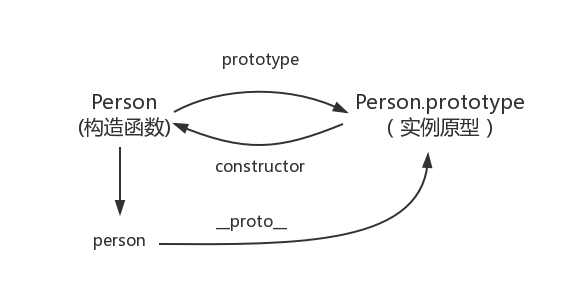
}

var person = new Person();

console.log(person.\_\_proto\_\_ === Person.prototype); // true

**constructor**

每个原型都有一个 constructor 属性指向关联的构造函数

function Person() {

}

var person = new Person();

console.log(person.\_\_proto\_\_ == Person.prototype)// true

console.log(Person.prototype.constructor == Person) // true

// 顺便学习一个ES5的方法,可以获得对象的原型

console.log(Person === Person.prototype.constructor); // true

**实例与原型**

当我们访问实例(对象)的一个属性或方法时，①它会先在实例(对象)自身中寻找，如果有则直接使用，

②如果没有则会去原型对象( \_\_proto\_\_)中寻找，如果找到则直接使用，直到找到为止。

function Person() {

}

Person.prototype.name = 'Kevin';

var person = new Person();

person.name = 'Daisy';

console.log(person.name) // Daisy

delete person.name;

console.log(person.name) // Kevin

在这个例子中，我们给实例对象 person 添加了 name 属性，当我们打印 person.name 的时候，结果自然为 Daisy。

但是当我们删除了 person 的 name 属性时，读取 person.name，从 person 对象中找不到 name 属性就会从 person 的原型也就是 person.\_\_proto\_\_ ，也就是 Person.prototype中查找，幸运的是我们找到了 name 属性，结果为 Kevin。

**原型的原型**

原型也是一个对象，既然是对象，我们就可以用最原始的方式创建它

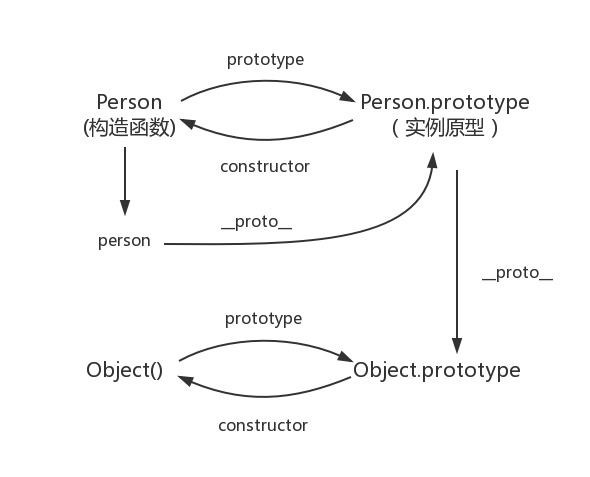
var obj = new Object();

obj.name = 'Kevin'

console.log(obj.name) // Kevin

所以原型对象是通过 Object 构造函数生成的，结合之前所讲，

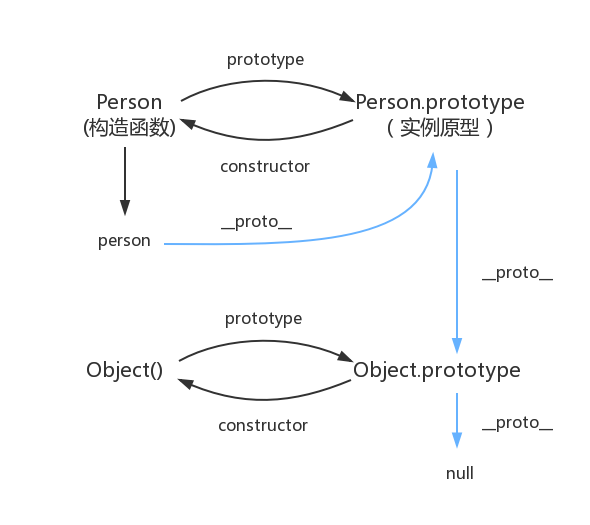
实例的 \_\_proto\_\_ 指向构造函数的 prototype ，所以我们再更新下关系图：

**原型链**

Object.prototype 的原型是 null

console.log(Object.prototype.\_\_proto\_\_ === null) // true

所以查到属性的时候查到 Object.prototype 就可以停止查找了。

相互关联的原型组成的链状结构就是原型链：

**垃圾回收（GC）**

- 当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象，此时这种对象就是一个垃圾，这种对象过多会占用大量的内存空间，导致程序运行变慢，所以这种垃圾必须进行清理。

- 在JS中拥有自动的垃圾回收机制，会自动将这些垃圾对象从内存中销毁，我们不需要也不能进行垃圾回收的操作

- 我们需要做的只是要将不再使用的对象设置null即可

**数组（Array）**

- 数组也是一个对象

- 它和我们普通对象功能类似，也是用来存储一些值的

- 不同的是普通对象是使用字符串作为属性名的，而数组时使用数字来作为索引操作元素

- 索引：从0开始的整数就是索引

- 数组的存储性能比普通对象要好，在开发中我们经常使用数组来存储一些数据

①使用构造函数创建数组,创建数组对象

var arr = new Array();

②使用字面量来创建数组

var arr = [];

元素之间使用,隔开

向数组中添加元素： 数组[索引] = 值

获取数组的长度： 数组.length

可以使用length属性来获取数组的长度(元素的个数)

修改length：①如果修改的length大于原长度，则多出部分会空出来

            ②如果修改的length小于原长度，则多出的元素会被删除

向数组的最后一个位置添加元素：数组[数组.length] = 值;

数组中的元素可以是任意的数据类型

   arr = ["hello",1,true,null,undefined];

//对象

   arr = [{name:"AA"},{name:"BB"},{name:"CC"}];

//函数

   arr = [function(){alert(1)},function(){alert(2)}];

多维数组

数组里的项也可以是数组。我们可以将其用于多维数组，例如存储矩阵：

let matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

];

alert( matrix[1][1] ); // 最中间的那个数

**数组的方法**

join() 数组=>字符串

split() 字符串=>数组

push() (从后添加)接收任意数量的参数，逐个添加至数组末尾，返回修改后数组的长度。

pop() （从后删除）从数组末尾删除最后一项并返回该项

shift() （从前删除）与pop()类似 美 [ʃɪft] 转移; 变换

unshift() （从前添加）与push()类似

sort() 排序，需要传入callback函数 (修改原数组，返回数组的引用)

arr.sort( (a,b)=>(a-b) ) 从小到大排序

arr.sort( (a,b)=>(a-b) ) 从大到小排序

对象数组排序： arr.sort( (a,b)=>(a.xx-b.xx) ) 从小到大排序

reverse() 反转数组项的顺序。

concat() 将参数添加到原数组中。（参数可以是 单个值/数组）

截取字符串: （3个中只有slice()可以用在数组的截取）

substr(start [, length ]) 返回一个从指定位置开始的指定长度的子字符串。

substring(开始索引，结束索引) 返回起始和结束位置之间的项（不包括结束位置的项）

slice(开始索引，结束索引) 返回起始和结束位置之间的项（不包括结束位置的项）参数可以是负数

splice(索引，删除数量，插入元素) 数组增/删/改，返回被删除/替换的元素。

indexOf()和 lastIndexOf() （ES5新增）

数组遍历方法：

for循环 性能最高但不够优雅

forEach() 没有返回值，对原数组没有影响（可以用数组方法改变数组，）（ES5新增）

map() 有返回值，返回新的数组，对原数组没有影响（原数组克隆一份来遍历）（ES5新增）

for...in 会枚举原型属性，就是说会遍历来自继承对象的可枚举属性(数组别用)

for...of 只能对数组使用

filter() 过滤器 保留为true的项 有返回值，返回一个新数组（ES5新增）

find() 找到第一个符合条件的数组成员返回**,**没有则返回undefined

some() 检测任一元素是否满足条件，并返回一个Boolean值（ES5新增）

every() 检测所有元素是否满足条件，并返回一个Boolean值。（ES5新增）

reduce()和 reduceRight() 汇总 一堆出来一个 (合并为一个值)（ES5新增）

数组的3个方法，均返回 Array的迭代器对象

Array.keys() //遍历数组键名

Array.values() //遍历数组键值

Array.entries() //遍历数组键名和键值

在调用函数时，浏览器每次都会传递进两个隐含的参数：

1.函数的上下文对象 this

2.封装实参的对象 arguments

- arguments是一个类数组对象,它也可以通过索引来操作数据，也可以获取长度

- 在调用函数时，我们所传递的实参都会在arguments中保存

- arguments.length可以用来获取实参的长度

- 我们即使不定义形参，也可以通过arguments来使用实参，只不过比较麻烦

arguments[0] 表示第一个实参

arguments[1] 表示第二个实参 。。。

- 它里边有一个属性叫做callee，这个属性对应一个函数对象，就是当前正在指向的函数的对象

arguments.callee

  function fun(a,b){

typeof arguments // object

           arguments instanceof Array // false

           Array.isArray(arguments) // false

           arguments[0] // hello

           arguments.length // 2

           arguments.callee == fun // true

      }

    fun("hello",true);

**Date 对象**

Date 对象用于处理日期和时间。

创建 Date 对象的语法：

var myDate=new Date()

注释：Date 对象会自动把当前日期和时间保存为其初始值。

Date 对象属性:

[constructor](https://www.w3school.com.cn/jsref/jsref_constructor_date.asp) 返回对创建此对象的 Date 函数的引用

[prototype](https://www.w3school.com.cn/jsref/jsref_prototype_date.asp) 使您有能力向对象添加属性和方法。

Date 对象方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| [Date()](https://www.w3school.com.cn/jsref/jsref_Date.asp) | 返回当日的日期和时间。 |
| [getDate()](https://www.w3school.com.cn/jsref/jsref_getDate.asp) | 从 Date 对象返回一个月中的某一天 (1 ~ 31)。 |
| [getDay()](https://www.w3school.com.cn/jsref/jsref_getDay.asp) | 从 Date 对象返回一周中的某一天 (0 ~ 6)。 |
| [getMonth()](https://www.w3school.com.cn/jsref/jsref_getMonth.asp) | 从 Date 对象返回月份 (0 ~ 11)。 |
| [getFullYear()](https://www.w3school.com.cn/jsref/jsref_getFullYear.asp) | 从 Date 对象以四位数字返回年份。 |
| [getYear()](https://www.w3school.com.cn/jsref/jsref_getYear.asp) | 请使用 getFullYear() 方法代替。 |
| [getHours()](https://www.w3school.com.cn/jsref/jsref_getHours.asp) | 返回 Date 对象的小时 (0 ~ 23)。 |
| [getMinutes()](https://www.w3school.com.cn/jsref/jsref_getMinutes.asp) | 返回 Date 对象的分钟 (0 ~ 59)。 |
| [getSeconds()](https://www.w3school.com.cn/jsref/jsref_getSeconds.asp) | 返回 Date 对象的秒数 (0 ~ 59)。 |
| [getMilliseconds()](https://www.w3school.com.cn/jsref/jsref_getMilliseconds.asp) | 返回 Date 对象的毫秒(0 ~ 999)。 |
| [getTime()](https://www.w3school.com.cn/jsref/jsref_getTime.asp) | 返回 1970 年 1 月 1 日至今的毫秒数(时间戳) |

**包装类**

基本数据类型String Number Boolean Null Undefined 、(BingInt 、Symbol)

引用数据类型 Object

在JS中为我们提供了三个包装类，通过这三个包装类可以,将基本数据类型的数据转换为对象

String() - 可以将基本数据类型字符串转换为String对象

Number() - 可以将基本数据类型的数字转换为Number对象

Boolean() - 可以将基本数据类型的布尔值转换为Boolean对象

但是注意：我们在实际应用中不会使用基本数据类型的对象，如果使用基本数据类型的对象，在做一些比较时可能会带来一些不可预期的结果

  var str = new String("hello");

  var str2 = new String("hello");

console.log(str === str2); //false

var num = new Number(3);

num.hello = "abcdefg";

console.log(typeof num); //object

console.log(num); //Number {3, hello: "abcdefg"}

方法和属性只能添加给对象，不能添加给基本数据类型

当我们对一些基本数据类型的值去调用属性和方法时，浏览器会临时使用包装类将其转换为对象，然后在调用对象的属性和方法,调用完以后，在将其转换为基本数据类型

var AA = 123;

AA.hello = "你好";

console.log(AA.hello); //undefined

console.log(typeof AA); //number

**正则表达式**

正则表达式用于定义一些字符串的规则，计算机可以根据正则表达式，来检查一个字符串是否符合规则，

获取将字符串中符合规则的内容提取出来

创建正则表达式的对象:

①使用构造函数创建

var 变量 = new RegExp("正则表达式","匹配模式");

在构造函数中可以传递一个匹配模式作为第二个参数:

i 忽略大小写

g 全局匹配模式

②使用字面量来创建正则表达式

var 变量 = /正则表达式/匹配模式

使用typeof检查正则对象，会返回object

正则表达式的测试方法：test() 正则.test(str)

- 使用这个方法可以用来检查一个字符串是否符合正则表达式的规则，如果符合则返回true，否则返回false

var reg=/a/

var str="abcd"

var str2='efg'

reg.test(str) //true

reg.test(str2) //false

**|** 或

var reg = /a|b|c/;

**[]**里的内容也是或的关系

[ab] == a|b

[a-z] 任意小写字母

[A-Z] 任意大写字母

[A-z] 任意字母

[0-9] 任意数字

检查一个字符串中是否含有 abc 或 adc 或 aec

reg = /a[bde]c/;

**[^ ]** 除了以外

reg = /[^0-9]/; // 除了数字以外

根据任意字母来将字符串拆分

var result = str.split(/[A-z]/);

**search()** 返回首次出现的索引，没有返回-1

- 它可以接受一个正则表达式作为参数，然后会根据正则表达式去检索字符串

- serach()只会查找第一个，即使设置全局匹配也没用

搜索字符串中是否含有abc 或 aec 或 afc

result = str.search(/a[bef]c/);

**match()** 会将匹配到的内容封装到一个数组中返回，即使只查询到一个结果

- 可以根据正则表达式，从一个字符串中将符合条件的内容提取出来

- 默认情况下我们的match只会找到第一个符合要求的内容，找到以后就停止检索

我们可以设置正则表达式为全局匹配模式（g），这样就会匹配到所有的内容

可以为一个正则表达式设置多个匹配模式，且顺序无所谓

var str="abcd"

var result = str.match(/[a-z]/i ); //["a", index: 0, input: "abcd", groups: undefined]

var result = str.match(/[a-z]/ig); //["a", "b", "c", "d"]

**replace("正则","新内容")** 可以将字符串中指定内容替换为新的内容

- 默认只会替换第一个

var str="abcd11abcd"

var result = str.replace(/[a-z]/ , ""); //bcd11abcd

var result = str.replace(/[a-z]/gi , ""); //11

**split()** 字符串 =>数组

- 方法中可以传递一个正则表达式作为参数，这样方法将会根据正则表达式去拆分字符串

- 这个方法即使不指定全局匹配，也会全都拆分

**量词**

- 通过量词可以设置一个内容出现的次数

- 量词只对它前边的一个内容起作用

{n} 正好出现n次

{m,n} 出现m-n次

{m,} m次以上

+ 至少一个，相当于{1,} /ab+c/

\* 0个或多个，相当于{0,} /ab\*c/

? 0个或1个，相当于{0,1} /ab?c/

**^**表示开头

reg = /^a/; //匹配开头的a

**$** 表示结尾

reg = /a$/; //匹配结尾的a

如果在正则表达式中同时使用^ $则要求字符串必须完全符合正则表达式:

reg = /^a$/; 只能是一个a 开头结尾

reg= /^a|a$/; 开头是a 结尾也是a

创建一个正则表达式，用来检查一个字符串是否是一个合法手机号:

手机号的规则： 1 3 567890123 （11位）

1. 以1开头

2. 第二位3-9任意数字

3. 三位以后任意数字9个

^1 [3-9] [0-9]{9}$

var phoneReg = /^1[3-9][0-9]{9}$/;

**.** 表示任意字符

在正则表达式中使用\作为转义字符: \. 来表示.

//检查一个字符串中是否含有 .

  var reg = /\./;

var reg = new RegExp("\\.");

使用构造函数时，由于它的参数是一个字符串，而\是字符串中转义字符，如果要使用\则需要使用\\来代替

转义字符：

\w - 任意字母、数字、\_ [A-z0-9\_]

\W - 除了字母、数字、\_ [^A-z0-9\_]

\d - 任意的数字 [0-9]

\D - 除了数字 [^0-9]

\s - 空格

\S - 除了空格

\b - 单词边界 reg = /\bchild\b/;

\B - 除了单词边界

接收一个用户的输入：

var str = prompt("请输入你的用户名:");

去除开头的空格：

str = str.replace(/^\s\*/, "");

去除结尾的空格：

str = str.replace(/\s\*$/, "");

/^\s\*|\s\*$/g 匹配开头和结尾的空格：

str = str.replace(/^\s\*|\s\*$/g,"");

电子邮件：

任意字母数字下划线 .任意字母数字下划线 @ 任意字母数字 .任意字母（2-5位）. 任意字母（2-5位）

/^ \w{3,} (\.\w+)\* @ [A-z0-9]+ (\.[A-z]{2,5}){1,2}$/

**DOM**，全称Document Object Model文档对象模型。

• JS中通过DOM来对HTML文档进行操作。只要理解DOM就可以随心所欲的操作WEB页面。

• 文档 – 文档表示的就是整个的HTML网页文档

• 对象 – 对象表示将网页中的每一个部分都转换为了一个对象。

• 模型 – 使用模型来表示对象之间的关系，这样方便我们获取对象。

节点

• 节点Node，是构成我们网页的最基本的组成部分，网页中的 每一个部分都可以称为是一个节点。

• 比如：html标签、属性、文本、注释、整个文档等都是一个节 点。

• 虽然都是节点，但是实际上他们的具体类型是不同的。

• 比如：标签我们称为元素节点、属性称为属性节点、文本称为 文本节点、文档称为文档节点。

• 节点的类型不同，属性和方法也都不尽相同。

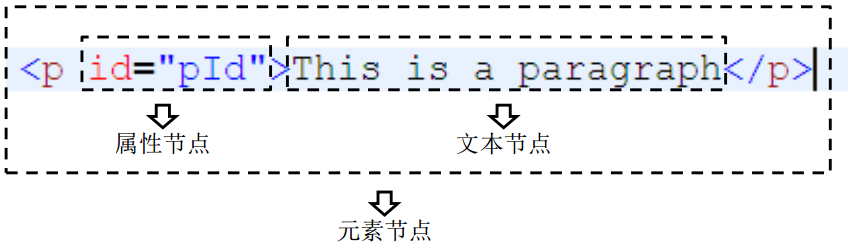
• 节点：Node——构成HTML文档最基本的单元。

• 常用节点分为四类

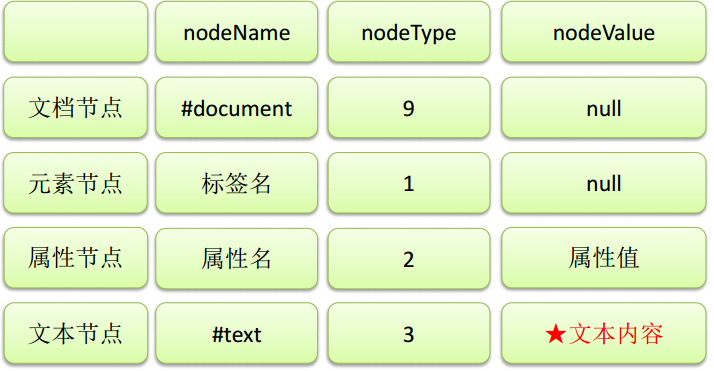
– 文档节点：整个HTML文档

– 元素节点：HTML文档中的HTML标签

– 属性节点：元素的属性

– 文本节点：HTML标签中的文本内容

节点的属性



**文档节点（document）**

• 文档节点document，代表的是整个HTML文档，网页中的所有节点都是它的子节点。

• document对象作为window对象的属性存在的，我们不用获取可以直接使用。~~window.document~~

• 通过document对象我们可以在整个文档访问内查找节点对象，并可以通过该对象创建各种节点对象。

\* 浏览器已经为我们提供文档节点对象这个对象是window属性,可以在页面中直接使用，文档节点代表的是整个网页

**元素节点（****Element）**

• HTML中的各种标签都是元素节点，这也是我们最常用 的一个节点。

• 浏览器会将页面中所有的标签都转换为一个元素节点， 我们可以通过document的方法来获取元素节点。

document.getElementById() – 根据id属性值获取一个元素节点对象。

**文本节点（Text）**

• 文本节点表示的是HTML标签以外的文本内容，任意非HTML的文本 都是文本节点。

• 它包括可以字面解释的纯文本内容。

• 文本节点一般是作为元素节点的子节点存在的。

• 获取文本节点时，一般先要获取元素节点。在通过元素节点获取文本节点。

• 例如： – 元素节点.firstChild; – 获取元素节点的第一个子节点，一般为文本节点

**属性节点（Attr）**

• 属性节点表示的是标签中的一个一个的属性，这里要注意的是属性节点并非是元素节点的子节点，而是元素节点的一部分。

• 可以通过元素节点来获取指定的属性节点。

• 例如： – 元素节点.getAttributeNode("属性名");

• 注意：我们一般不使用属性节点。

**事件**

• 事件，就是文档或浏览器窗口中发生的一些特定的交互瞬间。

• JavaScript 与 HTML 之间的交互是通过事件实现的。

• 对于 Web 应用来说，有下面这些代表性的

事件：点击某个元素、将鼠标移动至某个 元素上方、按下键盘上某个键，等等。

//获取按钮对象

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

//绑定一个单击事件

//像这种为单击事件绑定的函数，我们称为单击响应函数

btn.onclick = function(){

alert("你还点~~~");

};

**文档的加载**

浏览器在加载一个页面时，是按照自上向下的顺序加载的，读取到一行就运行一行,如果将script标签写到页面的上边，在代码执行时，页面还没有加载，页面没有加载DOM对象也没有加载，会导致无法获取到DOM对象，onload事件会在整个页面加载完成之后才触发

\* 为window绑定一个onload事件

\* 该事件对应的响应函数将会在页面加载完成之后执行，

\* 这样可以确保我们的代码执行时所有的DOM对象已经加载完毕了

window.onload = function(){

//获取id为btn的按钮

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

//为按钮绑定一个单击响应函数

btn.onclick = function(){

alert("hello");

};

};

JQuery页面载入写法：

$(document).ready(function(){

//dosomething

})

简写：

$(function(){

//dosomething

})

将js代码编写到页面的下部就是为了，可以在页面加载完毕以后再执行js代码

**获取元素（标签）节点**

• 通过document对象调用

document.getElementById(idName) –通过id属性获取一个元素节点对象，返回一个对象

document.getElementsByTagName(tagName) –通过标签名获取一组元素节点对象,返回元素对象数组

document.getElementsByName(name) –通过name属性获取一组元素节点对象，返回袁术对象数组

document.getElementsByClassName(className)-可以根据class属性值获取一组元素节点对象(IE8)

innerHTML 通过这个属性可以获取到元素内部的html代码

（innerHTML 属性设置或返回表格行的开始和结束标签之间的 HTML。对于自结束标签，这个属性没有意义）

**读取元素节点属性**： 元素.属性名

例子：①元素.id

②元素.name

③元素.value

注意：class属性不能采用这种方式

读取class属性时需要使用： 元素.className

**获取元素节点的子节点**

• 通过具体的元素节点调用

document.getElementsByTagName() – 方法，返回当前节点的指定标签名后代节点

(document.getElementsByTagName()==element)

element.childNodes – 属性，表示当前节点的所有子节点 （包括空白文本节点）

element.firstChild – 属性，表示当前节点的第一个子节点（包括空白文本节点）

element.lastChild – 属性，表示当前节点的最后一个子节点

childNodes属性会获取包括文本节点在内的所有节点

\* 根据DOM标签标签间空白也会当成文本节点

\* 注意：在IE8及以下的浏览器中，不会将空白文本当成子节点

element.children 属性可以获取当前元素的所有子元素对象，只返回HTML节点

**获取父节点和兄弟节点**

• 通过具体的节点调用

element.parentNode – 属性，表示当前节点的父节点

element.previousSibling–属性，表示当前节点的前一个兄弟节点（可能获取到空白的文本）

element.nextSibling – 属性，表示当前节点的后一个兄弟节点

element.innerText - 该属性可以获取到元素内部的文本内容

- 它和innerHTML类似，不同的是它会自动将html去除

**元素节点的属性**

• 获取:元素对象.属性名

例：element.value

element.id

element.className

• 设置:元素对象.属性名=新的值

例：element.value = “hello”

element.id = “id01”

element.className = “newClass”

其他属性

• node.nodeValue – 文本节点可以通过nodeValue属性获取和设置文本节点的内容

• element.innerHTML – 元素节点通过该属性获取和设置标签内部的 html代码

• element.innerText - 该属性可以获取到元素内部的文本内容,会自动将html去除

在事件的响应函数中，响应函数是给谁绑定的this就是谁

document.body 保存的是body的引用

document.documentElement 保存的是html根标签

document.all 代表页面中所有的元素

**使用CSS选择器进行查询**

• document.querySelector()

• document.querySelectorAll()

相同:都是传递一个选择器字符串作为参数，方法会自动根据选择器 字符串去网页中查找元素。

不同:querySelector()只会返回找到的第一个元素，而 querySelectorAll()会返回所有符合条件的元素。返回数组

- IE8中没有getElementsByClassName()但是可以使用document.querySelector()代替

var div = document.querySelector(".box1 div");

document.querySelectorAll()

- 该方法和querySelector()用法类似，不同的是它会将符合条件的元素封装到一个数组中返回

- 即使符合条件的元素只有一个，它也会返回数组

var box1 = document.querySelectorAll(".box1");

**Dom增删改（Dom方法）**

document.createElement()

用于创建一个元素节点对象，它需要一个标签名作为参数，将会根据该标签名创建元素节点对象，并将创建好的对象作为返回值返回

var li = document.createElement("li");

document.createTextNode()

用来创建一个文本节点对象，需要一个文本内容作为参数，将会根据该内容创建文本节点，并将新的节点返回

var gzText = document.createTextNode("广州");

**element.appendChild(子节点)** - 向一个父节点中添加一个新的子节点

**element.insertBefore(新节点,旧节点)** - 可以在指定的子节点前插入新的子节点

**element.replaceChild(新节点,旧节点)** - 可以使用指定的子节点替换已有的子节点

**element.removeChild(子节点)/element.parentNode.removeChild(子节点)** - 可以删除一个子节点

使用innerHTML也可以完成DOM的增删改的相关操作, 一般我们会两种方式结合使用

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

city.innerHTML += "<li>广州</li>";

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

var li = document.createElement("li");

li.innerHTML = "广州";

city.appendChild(li);

**DOM 操作CSS**

**修改元素的样式：**

语法： element.style.样式名 = 样式值

注意：如果CSS的样式名中含有-，这种名称在JS中是不合法的比如background-color,需要将这种样式名修改为驼峰命名法，去掉-，然后将-后的字母大写 backgroundColor

box1.style.width = "300px";

我们通过style属性设置的样式都是内联样式，而内联样式有较高的优先级，所以通过JS修改的样式往往会立即显示,但是如果在样式中写了!important，则此时样式会有最高的优先级，即使通过JS也不能覆盖该样式，此时将会导致JS修改样式失效,所以尽量不要为样式添加!important

**读取元素的样式:**

语法：element.style.样式名

通过style属性设置和读取的都是内联样式,无法读取样式表中的样式

**获取元素的当前显示的样式**

语法：element.currentStyle.样式名

它可以用来读取当前元素正在显示的样式，如果当前元素没有设置该样式，则获取它的默认值，

currentStyle只有IE浏览器支持，其他的浏览器都不支持。

在其他浏览器中可以使用getComputedStyle() 这个方法来获取元素当前的样式

这个方法是window的方法，可以直接使用，需要两个参数

第一个：要获取样式的元素

第二个：可以传递一个伪元素，一般都传null

该方法会返回一个对象，对象中封装了当前元素对应的样式,可以通过 对象.样式名 来读取样式

getComputedStyle(box1,null).width

如果获取的样式没有设置，则会获取到真实的值，而不是默认值

比如：没有设置width，它不会获取到auto，而是一个长度

但是该方法不支持IE8及以下的浏览器

通过currentStyle和getComputedStyle()读取到的样式都是只读的，不能修改，如果要修改必须通过style属性

定义一个函数，用来获取指定元素的当前的样式

参数：obj 要获取样式的元素,name 要获取的样式名

function getStyle(obj , name){

if(window.getComputedStyle){

//正常浏览器的方式，具有getComputedStyle()方法

return getComputedStyle(obj , null)[name];

}else{

//IE8的方式，没有getComputedStyle()方法

return obj.currentStyle[name];

}

//return

window.getComputedStyle?getComputedStyle(obj , null)[name]:obj.currentStyle[name]; }

其他样式操作属性:

clientWidth / clientHeight

- 这两个属性可以获取元素的可见宽度和高度

- 这些属性都是不带px的，返回都是一个数字，可以直接进行计算

- 会获取元素宽度和高度，包括内容区和内边距（padding）

- 这些属性都是只读的，不能修改

alert(box1.clientWidth);

alert(box1.clientHeight);

offsetWidth / offsetHeight

- 获取元素的整个的宽度和高度，包括内容区、内边距和边框（border）

offsetParent

- 可以用来获取当前元素的定位父元素

- 会获取到离当前元素最近的开启了定位（position值不是static）的祖先元素

如果所有的祖先元素都没有开启定位，则返回body

offsetLeft - 当前元素相对于其定位（position值不是static）父元素的水平偏移量

offsetTop - 当前元素相对于其定位父元素的垂直偏移量

scrollWidth / scrollHeight - 可以获取元素整个滚动区域的宽度和高度

scrollLeft - 可以获取水平滚动条滚动的距离

scrollTop - 可以获取垂直滚动条滚动的距离

垂直滚动条滚动到底:

scrollHeight - scrollTop == clientHeight

水平滚动条滚动到底:

scrollWidth - scrollLeft == clientWidth

onscroll - 该事件会在元素的滚动条滚动时触发

disabled 属性可以设置一个元素是否禁用，如果设置为true，则元素禁用,如果设置为false，则元素可用

**事件**

• 事件，指的就是 用户与浏览器交互的一瞬间。

• 我们通过为指定事件绑定回调函数的形式来处理事件，当指定事件触发以后我们的回调函数就会被调用，这样我们的页面就可以完成和用户的交互了。

事件处理程序

• 我们可以通过两种方式为一个元素绑定事件处理程序：

①通过HTML元素指定事件属性来绑定

②通过DOM对象指定的属性来绑定

• 这两种方式都是我们日常用的比较多的，但是更推荐使 用第二种方式。

• 还有一种方式比较特殊我们称为设置事件监听器。使用 如下方式：

③element.addEventListener()

① 通过HTML属性来绑定事件处理程序是最简单的方式。

<button onclick="alert('hello,world');">按钮</button>

• 这种方式我们直接将代码编写到了onclick属性中，可以编写多行js代码，也可以事先在外部定义好函数。

• 这种方式的优点在于，设定步骤非常简单，并且能够确保事件 处理程序会在载入时被设定。

• 如果在函数的最后return false则会取消元素的默认行为。

②通过DOM对象的属性绑定

• 但是其实上面的写法虽然简单，但却将JS和HTML 的代码编写到了一起，并不推荐使用

var btn = document.getElementById('btn');

btn.onclick = function(){

alert("hello,world");

};

• 这种写法将HTML代码和JS写在不同的位置，维护起来更加容易。

③设置事件监听器

elementObject.addEventListener(eventName,handle,useCapture);

| 参数 | 说明 |
| --- | --- |
| elementObject | DOM对象（即DOM元素）。 |
| eventName | 事件名称。注意，这里的事件名称没有“ on ”，如鼠标单击事件 click ，  鼠标双击事件 doubleclick ，鼠标移入事件 mouseover，鼠标移出事件 mouseout 等。 |
| handle | 事件句柄函数，即用来处理事件的函数。 |
| useCapture | Boolean类型，是否在捕获阶段就触发事件，一般用false 。 |

elementObject.attachEvent(eventName,handle);（ie8以下的浏览器）

| 参数 | 说明 |
| --- | --- |
| elementObject | DOM对象（即DOM元素）。 |
| eventName | 事件名称。注意，与addEventListener()不同，这里的事件名称有“ on ”  如鼠标单击事件 onclick ，鼠标双击事件 ondoubleclick ，  鼠标移入事件 onmouseover，鼠标移出事件 onmouseout 等。 |
| handle | 事件句柄函数，即用来处理事件的函数。 |

• 也可以使用removeEventListener()和detachEvent()移除事件。

  //定义一个函数，用来为指定元素绑定响应函数

            /\*

             \* addEventListener()中的this，是绑定事件的对象

             \* attachEvent()中的this，是window

             \* 需要统一两个方法this

             \*/

            /\*

             \* 参数：

             \*  obj 要绑定事件的对象

             \*  eventStr 事件的字符串(不要on)

             \* callback 回调函数

             \*/

            function bind(obj , eventStr , callback){

                if(obj.addEventListener){

                    //大部分浏览器兼容的方式

                    obj.addEventListener(eventStr , callback , false);

                }else{

                    /\*

                     \* this是谁由调用方式决定

                     \* callback.call(obj)

                     \*/

                    //IE8及以下

                    obj.attachEvent("on"+eventStr , function(){

                        //在匿名函数中调用回调函数

                        callback.call(obj);

                    });

                }

            }

//调用

bind(btn01 , "click" , function(){

                    alert(this);

                });

事件处理中的this

• 在事件处理程序内的 this 所引用的对象即是设定了该事件处理程序的元素。

• 也就是事件是给那个对象绑定的this就是哪个对象。

**事件对象**

• 在DOM对象上的某个事件被触发时，会产生一个事件对象Event，这个对象中包含着所有事件有关的信息。包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。

比如：鼠标的坐标 键盘哪个按键被按下 鼠标滚轮滚动的方向。。。

• 所有浏览器都支持 event 对象，但支持方式不同。

• DOM标准的浏览器会将一个event对象传入到事件的处理程序当中。无论事件处理程序是什么都会传入一个event对象。

• 可以通过这种方式获取：

var btn=document.getElementById("btn")

btn.onclick = function(event){ alert(event.type); };

• Event对象包含与创建它的特定事件有关的属性和方法。

触发的事件类型不一样，可用的属性和方法也不一样。

**事件的传播**

• 在网页中标签与标签之间是有嵌套关系的，比如这样一个页面：

<html>

<body>

<div id="foo">

<button id="bar">sample</button>

</div>

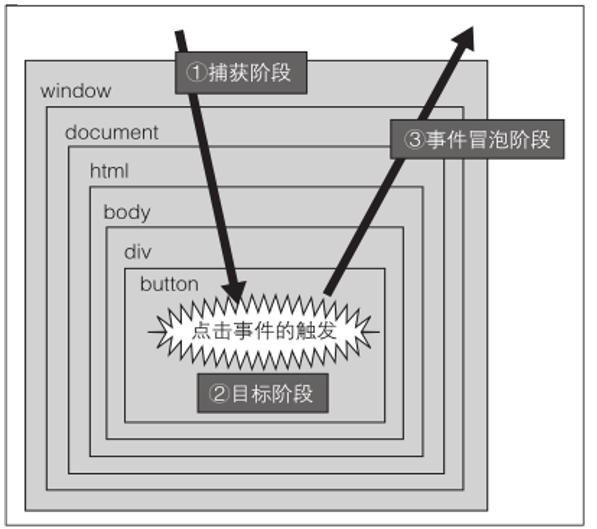
</body>

</html>

• 如果这时用户点击了sample按钮，则会以该按钮作为事件目标触发一次点击事件。

• 这时，事件的处理将会分为捕获阶段、目标阶段、事件冒泡这三个阶段。

事件的传播流程：



①捕获阶段

这一阶段会从window对象开始向下一直遍历到目标对象，如果发现有对象绑定了响应事件则做相应的处理。

②目标阶段

这一阶段已经遍历结束，则会执行目标对象上绑定的响应函数。

③事件冒泡阶段

这一阶段，事件的传播方式和捕获阶段正好相反，会从事件目标一直向上遍历，直至window对象结束，这时对象上绑定的响应函数也会执行。（事件从目标元素向他的祖先元素传递，依次触发祖先元素上的事件）

如果希望在捕获阶段就触发事件，可以将addEventListener()的第三个参数设置为true

一般情况下我们不会希望在捕获阶段触发事件，所以这个参数一般都是false

IE8及以下的浏览器中没有捕获阶段

取消事件传播

• 我们可以使用event对象的两个方法完成：

– stopPropagation()

– stopImmediatePropagation()

• 取消默认行为： preventDefault()

**事件的冒泡（Bubble）**

- 所谓的冒泡指的就是事件的向上传导，当后代元素上的事件被触发时，其祖先元素的相同事件也会被触发

- 在开发中大部分情况冒泡都是有用的,如果不希望发生事件冒泡可以通过事件对象来取消冒泡

将事件对象的cancelBubble设置为true，即可取消冒泡

var s1 = document.getElementById("s1");

s1.onclick = function(event){

event = event || window.event;

alert("我是span的单击响应函数");

//取消冒泡

event.cancelBubble = true;

};

**事件的委派**

- 指将事件统一绑定给元素的共同的祖先元素，这样当后代元素上的事件触发时，会一直冒泡到祖先元素，从而通过祖先元素的响应函数来处理事件。

- 事件委派是利用了冒泡，通过委派可以减少事件绑定的次数，提高程序的性能

事件委托的优点：

1、 可以大量节省内存占用，减少事件注册，比如在ul上代理所有li的click事件就非常棒

如上面代码所示，如果给每个li列表项都绑定一个函数，那对内存的消耗是非常大的，因此较好的解决办法就是将li元素的点击事件绑定到它的父元素ul身上，执行事件的时候再去匹配判断目标元素。

2、可以实现当新增子对象时无需再次对其绑定（动态绑定事件）

假设上述的例子中列表项li就几个，我们给每个列表项都绑定了事件；

在很多时候，我们需要通过 AJAX 或者用户操作动态的增加或者删除列表项li元素，那么在每一次改变的时候都需要重新给新增的元素绑定事件，给即将删去的元素解绑事件；

如果用了事件委托就没有这种麻烦了，因为事件是绑定在父层的，和目标元素的增减是没有关系的，执行到目标元素是在真正响应执行事件函数的过程中去匹配的；所以使用事件在动态绑定事件的情况下是可以减少很多重复工作的。

事件委托需要注意的地方：

因为 事件冒泡 的过程也要耗时，越接近顶层，事件的「事件传播链」就越长，也就越耗时。

target - event中的target表示的触发事件的对象

//如果触发事件的对象是我们期望的元素，则执行否则不执行

if(event.target.className == "link"){

alert("我是ul的单击响应函数");

}

**BOM**

• BOM 提供了很多对象，用于访问浏览器的功能，这些功能与任何网页内容无关。

• BOM将浏览器中的各个部分转换成了一个一个的对象,我们通过修改这些对象的属性，调用他们的方法，从而 控制浏览器的各种行为。

**BOM对象:**

Window - 代表的是整个浏览器的窗口，同时window也是网页中的全局对象

Navigator - 代表的当前浏览器的信息，通过该对象可以来识别不同的浏览器

Location - 代表当前浏览器的地址栏信息，通过Location可以获取地址栏信息，或者操作浏览器跳转页面

History - 代表浏览器的历史记录，可以通过该对象来操作浏览器的历史记录由于隐私原因，该对象不能获取到具体的历史记录，只能操作浏览器向前或向后翻页,而且该操作只在当次访问时有效

Screen - 代表用户的屏幕的信息，通过该对象可以获取到用户的显示器的相关的信息

BOM对象在浏览器中都是作为window对象的属性保存的，可以通过window对象来使用，也可以直接使用

Navigator

- 代表的当前浏览器的信息，通过该对象可以来识别不同的浏览器

-由于历史原因，Navigator对象中的大部分属性都已经不能帮助我们识别浏览器

- 一般我们只会使用userAgent来判断浏览器的信息，userAgent是一个字符串，这个字符串中包含有用来描述浏览器信息的内容，不同的浏览器会有不同的userAgent

var ua = navigator.userAgent;

if(/firefox/i.test(ua)){

alert("你是火狐！！！");

}else if(/chrome/i.test(ua)){

alert("你是Chrome");

}else if(/msie/i.test(ua)){

alert("你是IE浏览器~~~");

}else if("ActiveXObject" in window){

alert("你是IE11，枪毙了你~~~");

}

如果通过UserAgent不能判断，还可以通过一些浏览器中特有的对象，来判断浏览器的信息

比如：ActiveXObject

**History对象** - 包含用户（在浏览器窗口中）访问过的 URL

length - 属性，可以获取到当成访问的链接数量 //alert(history.length);

window.history.go()

– 使用 go() 方法可以在用户的历史记录中任意跳转，可以向后也可以向前。

- 它需要一个整数作为参数

1:表示向前跳转一个页面 相当于forward()

2:表示向前跳转两个页面

-1:表示向后跳转一个页面

-2:表示向后跳转两个页面

window.history.back() – 向后跳转

window.history.forward() – 向前跳转 可以跳转下一个页面，作用和浏览器的前进按钮一样

**Location对象**

• location对象包含有关当前 URL 的信息。

属性：

hash 设置或返回从井号 (#) 开始的 URL（锚）。

host 设置或返回主机名和当前 URL 的端口号。

hostname 设置或返回当前 URL 的主机名。

href 设置或返回完整的 URL。

pathname 设置或返回当前 URL 的路径部分。

port 设置或返回当前 URL 的端口号。

protocol 设置或返回当前 URL 的协议。

search 设置或返回从问号 (?) 开始的 URL（查询部分）。

ancestorOrigins: DOMStringList {length: 0}

assign: ƒ assign()

fragmentDirective: FragmentDirective {}

hash: "#/smz"

host: "localhost:9004"

hostname: "localhost"

href: "http://localhost:9004/#/smz"

origin: "http://localhost:9004"

pathname: "/"

port: "9004"

protocol: "http:"

reload: ƒ reload()

replace: ƒ replace()

search: ""

toString: ƒ toString()

valueOf: ƒ valueOf()

Symbol(Symbol.toPrimitive): undefined

\_\_proto\_\_: Location

方法：

• assign() 方法

- 用来跳转到其他的页面，作用和直接修改location一样

– 所用和href一样，使浏览器跳转页面，新地址错误参数传递到assign ()方中

• replace()方法

– 功能一样，只不过使用replace方法跳转地址不会体现到历史记录中。

- 可以使用一个新的页面替换当前页面，调用完毕也会跳转页面,不会生成历史记录，不能使用回退按钮回退

• reload() 方法

– 用于强制刷新当前页面

- 用于重新加载当前页面，作用和刷新按钮一样

- 如果在方法中传递一个true，作为参数，则会强制清空缓存刷新页面

location.reload(true);

**window对象**

• window对象是BOM的核心，它表示一个浏览器的实例。

• 在浏览器中我们可以通过window对象来访问操作浏览器，同时window也是作为全局对象存在的。

• 全局作用域： window对象是浏览器中的全局对象，因此所有在全局作用域中声明的变量、对象、函数都会变成window对象的属性 和方法。

窗口大小

• 浏览器中提供了四个属性用来确定窗口的大小：

– 网页窗口的大小

• innerWidth

• innerHeight

– 浏览器本身的尺寸

• outerWidth

• outerHeight

打开窗口

• 使用 window.open() 方法既可以导航到一个特定的URL，也可以打开一个新的浏览器窗口。

• 这个方法需要四个参数：

– 需要加载的url地址

– 窗口的目标

– 一个特性的字符串

– 是否创建新的历史记录

超时调用（延时调用）： **setTimeout()**

–按照指定的周期（以毫秒计）来调用函数或计算表达式，而且**只会执行一次**

–需要2个参数： ①要执行的内容 ②超过的时间

• 取消超时调用： clearTimeout()

• 超时调用都是在全局作用域中执行的。

间歇调用(定时调用)：  **setInterval()**

– 每隔一段时间执行指定代码

– 需要两个参数：①要执行的代码(回调函数，每隔一段时间调用一次) ②间隔的时间 (单位是毫秒)

• 取消间隔调用： clearInterval()

var timer = setInterval(function () {

count.innerHTML = num++;

if (num == 11) {

//关闭定时器 clearInterval(timer);

}

}, 1000);

延时调用和定时调用的区别，定时调用会执行多次，而延时调用只会执行一次

延时调用和定时调用实际上是可以互相代替的，在开发中可以根据自己需要去选择

系统对话框

• 浏览器通过 alert() 、 confirm() 和 prompt() 方法可以调用系统对话框向用户显示消息。

• 它们的外观由操作系统及（或）浏览器设置决 定，而不是由 CSS 决定。

• 显示系统对话框时会导致程序终止，当关闭对话框程序会恢复执行

Window 对象属性

|  |  |
| --- | --- |
| 属性 | 描述 |
| [closed](https://www.w3school.com.cn/jsref/prop_win_closed.asp) | 返回窗口是否已被关闭。 |
| [defaultStatus](https://www.w3school.com.cn/jsref/prop_win_defaultstatus.asp) | 设置或返回窗口状态栏中的默认文本。 |
| [document](https://www.w3school.com.cn/jsref/dom_obj_document.asp) | 对 Document 对象的只读引用。请参阅 [Document 对象](https://www.w3school.com.cn/jsref/dom_obj_document.asp)。 |
| [history](https://www.w3school.com.cn/jsref/dom_obj_history.asp) | 对 History 对象的只读引用。请参数 [History 对象](https://www.w3school.com.cn/jsref/dom_obj_history.asp)。 |
| [innerheight](https://www.w3school.com.cn/jsref/prop_win_innerheight_innerwidth.asp) | 返回窗口的文档显示区的高度。 |
| [innerwidth](https://www.w3school.com.cn/jsref/prop_win_innerheight_innerwidth.asp) | 返回窗口的文档显示区的宽度。 |
| length | 设置或返回窗口中的框架数量。 |
| [location](https://www.w3school.com.cn/jsref/dom_obj_location.asp) | 用于窗口或框架的 Location 对象。请参阅 [Location 对象](https://www.w3school.com.cn/jsref/dom_obj_location.asp)。 |
| [name](https://www.w3school.com.cn/jsref/prop_win_name.asp) | 设置或返回窗口的名称。 |
| [Navigator](https://www.w3school.com.cn/jsref/dom_obj_navigator.asp) | 对 Navigator 对象的只读引用。请参数 [Navigator 对象](https://www.w3school.com.cn/jsref/dom_obj_navigator.asp)。 |
| [opener](https://www.w3school.com.cn/jsref/prop_win_opener.asp) | 返回对创建此窗口的窗口的引用。 |
| [outerheight](https://www.w3school.com.cn/jsref/prop_win_outerheight.asp) | 返回窗口的外部高度。 |
| [outerwidth](https://www.w3school.com.cn/jsref/prop_win_outerwidth.asp) | 返回窗口的外部宽度。 |
| pageXOffset | 设置或返回当前页面相对于窗口显示区左上角的 X 位置。 |
| pageYOffset | 设置或返回当前页面相对于窗口显示区左上角的 Y 位置。 |
| parent | 返回父窗口。 |
| [Screen](https://www.w3school.com.cn/jsref/dom_obj_screen.asp) | 对 Screen 对象的只读引用。请参数 [Screen 对象](https://www.w3school.com.cn/jsref/dom_obj_screen.asp)。 |
| [self](https://www.w3school.com.cn/jsref/prop_win_self.asp) | 返回对当前窗口的引用。等价于 Window 属性。 |
| [status](https://www.w3school.com.cn/jsref/prop_win_status.asp) | 设置窗口状态栏的文本。 |
| [top](https://www.w3school.com.cn/jsref/prop_win_top.asp) | 返回最顶层的先辈窗口。 |
| window | window 属性等价于 self 属性，它包含了对窗口自身的引用。 |
| screenLeft  screenTop  screenX  screenY | 只读整数。声明了窗口的左上角在屏幕上的的 x 坐标和 y 坐标。  IE、Safari 和 Opera 支持 screenLeft 和 screenTop，  而 Firefox 和 Safari 支持 screenX 和 screenY。 |

## Window 对象方法

|  |  |
| --- | --- |
| 方法 | 描述 |
| [alert()](https://www.w3school.com.cn/jsref/met_win_alert.asp) | 显示带有一段消息和一个确认按钮的警告框。 |
| [blur()](https://www.w3school.com.cn/jsref/met_win_blur.asp) | 把键盘焦点从顶层窗口移开。 |
| [clearInterval()](https://www.w3school.com.cn/jsref/met_win_clearinterval.asp) | 取消由 setInterval() 设置的 timeout。 |
| [clearTimeout()](https://www.w3school.com.cn/jsref/met_win_cleartimeout.asp) | 取消由 setTimeout() 方法设置的 timeout。 |
| [close()](https://www.w3school.com.cn/jsref/met_win_close.asp) | 关闭浏览器窗口。 |
| [confirm()](https://www.w3school.com.cn/jsref/met_win_confirm.asp) | 显示带有一段消息以及确认按钮和取消按钮的对话框。 |
| [createPopup()](https://www.w3school.com.cn/jsref/met_win_createpopup.asp) | 创建一个 pop-up 窗口。 |
| [focus()](https://www.w3school.com.cn/jsref/met_win_focus.asp) | 把键盘焦点给予一个窗口。 |
| [moveBy()](https://www.w3school.com.cn/jsref/met_win_moveby.asp) | 可相对窗口的当前坐标把它移动指定的像素。 |
| [moveTo()](https://www.w3school.com.cn/jsref/met_win_moveto.asp) | 把窗口的左上角移动到一个指定的坐标。 |
| [open()](https://www.w3school.com.cn/jsref/met_win_open.asp) | 打开一个新的浏览器窗口或查找一个已命名的窗口。 |
| [print()](https://www.w3school.com.cn/jsref/met_win_print.asp) | 打印当前窗口的内容。 |
| [prompt()](https://www.w3school.com.cn/jsref/met_win_prompt.asp) | 显示可提示用户输入的对话框。 |
| [resizeBy()](https://www.w3school.com.cn/jsref/met_win_resizeby.asp) | 按照指定的像素调整窗口的大小。 |
| [resizeTo()](https://www.w3school.com.cn/jsref/met_win_resizeto.asp) | 把窗口的大小调整到指定的宽度和高度。 |
| [scrollBy()](https://www.w3school.com.cn/jsref/met_win_scrollby.asp) | 按照指定的像素值来滚动内容。 |
| [scrollTo()](https://www.w3school.com.cn/jsref/met_win_scrollto.asp) | 把内容滚动到指定的坐标。 |
| [setInterval()](https://www.w3school.com.cn/jsref/met_win_setinterval.asp) | 按照指定的周期（以毫秒计）来调用函数或计算表达式。 |
| [setTimeout()](https://www.w3school.com.cn/jsref/met_win_settimeout.asp) | 在指定的毫秒数后调用函数或计算表达式。 |

**JSON**

- JS中的对象只有JS自己认识，其他的语言都不认识

- JSON就是一个特殊格式的字符串，这个字符串可以被任意的语言所识别，并且可以转换为任意语言中的对象，JSON在开发中主要用来数据的交互

- JSON (JavaScript Object Notation) JS对象表示法

- JSON和JS对象的格式一样，只不过JSON字符串中的属性名必须加双引号,其他的和JS语法一致

JSON分类：1.对象 {}

2.数组 []

JSON中允许的值：1.字符串 2.数值 3.布尔值 4.null 5.对象 6.数组

var arr = '[1,2,3,"hello",true]'; //数组

var obj2 = '{"arr":[1,2,3]}'; //对象

在JS中，为我们提供了一个工具类，就叫JSON

这个对象可以帮助我们将一个JSON转换为JS对象，也可以将一个JS对象转换为JSON

JSON.parse() json --> js对象

- 可以将以JSON字符串转换为js对象

- 它需要一个JSON字符串作为参数，会将该字符串转换为JS对象并返回

var o = JSON.parse(arr);

JSON.stringify() JS对象 ---> JSON

- 可以将一个JS对象转换为JSON字符串

- 需要一个js对象作为参数，会返回一个JSON字符串

var str = JSON.stringify(obj3);

[js中const,var,let区别](https://www.cnblogs.com/ksl666/p/5944718.html)

①const定义的变量不可以修改，而且必须初始化。

②var定义的变量可以修改，如果不初始化会输出undefined，不会报错。

③let是块级作用域，函数内部使用let定义后，对函数外部无影响。