

JavaScript基础语法

- `HTML`：标记语言
- `JavaScript`：编程语言

序言

JavaScript发展历史 (JS)

1. 1994年, 网景公司(Netscape)发布了Navigator浏览器0.9版, 这是世界上第一款比较成熟的网络浏览器, 轰动一时。但是这是一款名副其实的浏览器--只能浏览页面, 浏览器无法与用户互动, 当时解决这个问题有两个办法, 一个是采用现有的语言, 许它们直接嵌入网页。另一个是发明一种全新的语言。

liveScript ==> javaScript ==> ECMAScript

2. 1995年Sun公司将Oak语言改名为Java, 正式向市场推出。Sun公司大肆宣传, 许诺这种语言可以"一次编写, 到处运行"(Write Once, Run Anywhere), 它看上去很可能成为未来的主宰。

3. 网景公司动了心, 决定与Sun公司结成联盟

4. 34岁的系统程序员Brendan Eich登场了。1995年4月, 网景公司录用了他, 他只用10天时间就把Javascript设计出来了。(多肽语言)

5. (1)借鉴C语言的基本语法; (2)借鉴Java语言的数据类型和内存管理; (3)借鉴Scheme语言, 将函数提升到"第一等公民"(first class)的地位; (4)借鉴Self语言, 使用基于原型(prototype)的继承机制。

JavaScript是什么

JavaScript(Java脚本)是一种基于对象 (Object) 和事件驱动 (Event Driven) 并具有安全性能的脚本语言, 使用JavaScript可以轻松的实现与HTML的互操作, 并且完成丰富的页面交互效果

基于对象:

指的是程序的内部已经为用户提供好了若干个对象, 用户直接使用这些对象即可。

面向对象:

java属于面向对象的语言, 面向对象是指用户自己定义类, 对象需要用户自己产生。

JavaScript能干什么

1. 常见的网页效果【表单验证, 轮播图。。。】
2. 与H5配合实现游戏【水果忍者: <http://www.jq22.com/demo/html5-fruit-ninja/>】
3. 实现应用级别的程序【<http://naotu.baidu.com>】
4. 实现统计效果【<http://echarts.baidu.com/examples/>】
5. 地理定位等功能【http://lbsyun.baidu.com/jsdemo.htm#i4_5】
6. 在线学编程【<https://codecombat.163.com/play/>】
7. js可以实现人工智能【面部识别】
8.

JavaScript特点

- (1)脚本语言。JavaScript是一种解释型的脚本语言,C、C++等语言先编译后执行,而JavaScript是在程序的运行过程中逐行进行解释。
- (2)基于对象。JavaScript是一种基于对象的脚本语言,它不仅可以创建对象,也能使用现有的对象。
- (3)简单。JavaScript语言中采用的是弱类型的变量类型,对使用的数据类型未做出严格的要求,是基于Java基本语句和控制的脚本语言,其设计简单紧凑。
- (4)动态性。JavaScript是一种采用事件驱动的脚本语言,它不需要经过Web服务器就可以对用户的输入做出响应。在访问一个网页时,鼠标在网页中进行鼠标点击或上下移、窗口移动等操作JavaScript都可直接对这些事件给出相应的响应。
- (5)跨平台性。JavaScript脚本语言不依赖于操作系统,仅需要浏览器的支持。因此一个JavaScript脚本在编写后可以带到任意机器上使用,前提上机器上的浏览器支持JavaScript脚本语言,目前JavaScript已被大多数的浏览器所支持。

JavaScript的组成

1. ECMAScript: 定义了javascript的语法规则,描述了语言的基本语法和数据类型
2. BOM (Browser Object Model): 浏览器对象模型
 - 有一套成熟的可以操作浏览器的 API, 通过 BOM 可以操作浏览器。比如: 弹出框、浏览器跳转、获取分辨率等
3. DOM (Document Object Model): 文档对象模型
 - 有一套成熟的可以操作页面元素的 API, 通过 DOM 可以操作页面中的元素。比如: 增加个 div, 减少个 div, 给div 换个位置等

总结: JS 就是通过固定的语法去操作 浏览器 和 标签结构 来实现网页上的各种效果

JavaScript代码的书写位置

- 和 `css` 一样, 我们的 `js` 也可以有多种方式书写在页面上让其生效
- `js` 也有多种方式书写, 分为 行内式, 内嵌式, 外链式

行内式 JS 代码 (不推荐)

- 写在标签上的 js 代码需要依靠事件 (行为) 来触发

```
<!-- 写在 a 标签的 href 属性上 -->
<a href="javascript:alert('我是一个弹出层');">点击一下试试</a>

<!-- 写在其他元素上 -->
<div onclick="alert('我是一个弹出层')">点一下试试看</div>

<!--
    注: onclick 是一个事件（点击事件），当点击元素的时候执行后面的 js 代码
-->
```

内嵌式 JS 代码

- 内嵌式的 js 代码会在页面打开的时候直接触发

```
<!-- 在 html 页面书写一个 script 标签，标签内部书写 js 代码 -->
<script type="text/javascript">
    alert('我是一个弹出层')
</script>

<!--
    注: script 标签可以放在 head 里面也可以放在 body 里面
-->
```

外链式 JS 代码（推荐）

- 外链式 js 代码只要引入了 html 页面，就会在页面打开的时候直接触发
- 新建一个 `.js` 后缀的文件，在文件内书写 js 代码，把写好的 js 文件引入 html 页面

```
// 我是 index.js 文件
alert('我是一个弹出层')
```

```
<!-- 我是一个 html 文件 -->

<!-- 通过 script 标签的 src 属性，把写好的 js 文件引入页面 -->
<script src="index.js"></script>

<!-- 一个页面可以引入多个 js 文件 -->
<script src="index1.js"></script>
<script src="index2.js"></script>
<script src="index3.js"></script>
```

常用输出/调试方法

- `alert()` 浏览器弹窗，弹出的内容就是 `()` 括号中的内容

- `document.write()` 向文档写入字符串、html 或 javascript 代码
- `console.log()` 在控制台打印相关信息
- 注意：调试代码应当从最终的产品代码中删除掉

JS 中的注释

- 学习一个语言，先学习一个语言的注释，因为注释是给我们自己看的，也是给开发人员看的
- 写好一个注释，有利于我们以后阅读代码

单行注释

- 一般就是用来描述下面一行代码的作用
- 可以直接写两个 `/`，也可以按 `ctrl + /`

```
// 我是一个单行注释

// 下面代码表示在浏览器里面出现一个弹出层
alert('我是一个弹出层')
```

多行注释

- 一般用来写一大段话，或者注释一段代码
- 可以直接写 `/**/` 然后在两个星号中间写注释，也可以按 `shift + alt + a`

```
/*
    我是一个多行注释
*/

/*
    注释的代码不会执行
    alert('我是一个弹出层')
    alert('我是一个弹出层')
*/
alert('我是一个弹出层')
```

变量（重点）

- 变量指的是在程序中保存数据的一个容器
- 变量是计算机内存中存储数据的标识符，根据变量名称可以获取到内存中存储的数据
- 也就是说，我们向内存中存储了一个数据，然后要给这个数据起一个名字，为了是我们以后再次找到他
- 语法： `var 变量名 = 值`

定义变量及赋值

```
// 定义一个变量
var num;

// 给一个变量赋值
num = 100;

// 定义一个变量的同时给其赋值
var num2 = 200;
```

- 注意：
 1. 一个变量名只能存储一个值
 2. 当再次给一个变量赋值的时候，前面一次的值就没有了
 3. 变量名称区分大小写（JS 区分大小写）

变量的命名规则和命名规范

- 规则：必须遵守的，不遵守就是错
 1. 一个变量名称可以由 **数字、字母、英文下划线（_）、美元符号（\$）** 组成
 2. 严格区分大小写
 3. 不能由数字开头，不要使用中文汉字命名
 4. 不能是 **保留字** 或者 **关键字**
 5. 不要出现空格
- 规范：建议遵守的（开发者默认），不遵守不会报错
 1. 变量名尽量有意义（语义化）
 2. 遵循驼峰命名规则，由多个单词组成的时候，从第二个单词开始首字母大写

JavaScript关键字

关键字可用于表示控制语句的开始或结束，或用于执行特定操作等。
按照规则，关键字也是语言保留的，不能用作标识符！

break	do	try	typeof
case	else	new	instanceof
catch	in	return	var
continue	for	switch	while
function	this	with	default
if	throw	delete

JavaScript保留字

保留字有可能在将来被用作关键字来使用，不能用作标识符！

abstract	int	short	boolean
export	interface	static	byte
extends	long	super	char
final	native	class	float
throws	const	goto	private
double	import	public

数据类型（重点）

- 是指我们存储在内存中的数据的数据类型
- 我们通常分为两大类 **基本数据类型** 和 **复杂数据类型**

基本数据类型

1. 数值类型（number）
 - 一切数字都是数值类型（包括二进制，十进制，十六进制等）
 - NaN（not a number），一个非数字
2. 字符串类型（string）
 - 被引号包裹的所有内容（可以是单引号也可以是双引号）
3. 布尔类型（boolean）
 - 只有两个（true 或者 false）
4. null类型（null）
 - 只有一个，就是 null，表示空的意思
5. undefined类型（undefined）
 - 只有一个，就是 undefined，表示没有值的意思

复杂数据类型（暂时先不讲）

1. 对象类型（object）
2. 函数类型（function）
3. ...

判断数据类型

- 既然已经把数据分开了类型，那么我们就要知道我们存储的数据是一个什么类型的数据
- 使用 `typeof` 关键字来进行判断

```
// 第一种使用方式
var n1 = 100;
console.log(typeof n1);

// 第二种使用方式
var s1 = 'abcdefg';
console.log(typeof(s1));
```

判断一个变量是不是数字

- 可以使用 `isNaN` 这个方法来判断一个变量是不是数字
- `isNaN` : is not a number

```
// 如果变量是一个数字
var n1 = 100;
console.log(isNaN(n1)); //=> false

// 如果变量不是一个数字
var s1 = 'Jack'
console.log(isNaN(s1)); //=> true
```

数据类型转换

- 数据类型之间的转换，比如数字转成字符串，字符串转成布尔，布尔转成数字等

其他数据类型转成数值

1. `Number(变量)`
 - 可以把一个变量强制转换成数值类型
 - 可以转换小数，会保留小数
 - 可以转换布尔值
 - 遇到不可转换的都会返回 NaN
2. `parseInt(变量)`
 - 从第一位开始检查，是数字就转换，知道一个不是数字的内容
 - 开头就不是数字，那么直接返回 NaN
 - 不认识小数点，只能保留整数
3. `parseFloat(变量)`
 - 从第一位开始检查，是数字就转换，知道一个不是数字的内容
 - 开头就不是数字，那么直接返回 NaN
 - 认识一次小数点
4. 除了加法以外的数学运算

- 运算符两边都是可运算数字才行
- 如果运算符任何一遍不是一个可运算数字，那么就会返回 NaN
- 加法不可以用

5. Number.toFixed(n) 方法可把 Number 四舍五入保留n位小数

其他数据类型转成字符串

1. 变量.toString()
 - 有一些数据类型不能使用 toString() 方法，比如 undefined 和 null
2. String(变量)
 - 所有数据类型都可以
3. 使用加法运算
 - 在 JS 里面，+ 由两个含义
 - 字符串拼接：只要 + 任意一边是字符串，就会进行字符串拼接
 - 加法运算：只有 + 两边都是数字的时候，才会进行数学运算

其他数据类型转成布尔

1. Boolean(变量)
 - 在 js 中，只有 ''、0、null、undefined、NaN，这些是 false，其余都是 true

运算符

- 就是在代码里面进行运算的时候使用的符号，不光只是数学运算，我们在 js 里面还有很多的运算方式

数学运算符

1. +
 - 只有符号两边都是数字的时候才会进行加法运算
 - 只要符号任意一边是字符串类型，就会进行字符串拼接
2. -
 - 会执行减法运算
 - 会自动把两边都转换成数字进行运算
3. *
 - 会执行乘法运算
 - 会自动把两边都转换成数字进行运算
4. /
 - 会执行除法运算

- 会自动把两边都转换成数字进行运算

5. %

- 会执行取余运算
- 会自动把两边都转换成数字进行运算

赋值运算符

1. =

- 就是把 = 右边的赋值给等号左边的变量名
- `var num = 100`
- 就是把 100 赋值给 num 变量
- 那么 num 变量的值就是 100

2. +=

```
var a = 10;  
a += 10;  
console.log(a); // => 20
```

- `a += 10` 等价于 `a = a + 10`

3. -=

```
var a = 10;  
a -= 10;  
console.log(a); // => 0
```

- `a -= 10` 等价于 `a = a - 10`

4. *=

```
var a = 10;  
a *= 10;  
console.log(a); // => 100
```

- `a *= 10` 等价于 `a = a * 10`

5. /=

```
var a = 10;  
a /= 10;  
console.log(a); // => 1
```

- `a /= 10` 等价于 `a = a / 10`

6. %=

```
var a = 10;  
a %= 10;  
console.log(a); // => 0
```

- o `a %= 10` 等价于 `a = a % 10`

比较运算符

1. `==`

- o 比较符号两边的值是否相等，不管数据类型
- o `1 == '1'`
- o 两个的值是一样的，所以得到 true

2. `===`

- o 比较符号两边的值和数据类型是否都相等
- o `1 === '1'`
- o 两个值虽然一样，但是因为数据类型不一样，所以得到 false

3. `!=`

- o 比较符号两边的值是否不等
- o `1 != '1'`
- o 因为两边的值是相等的，所以比较他们不等的时候得到 false

4. `!==`

- o 比较符号两边的数据类型和值是否不等
- o `1 !== '1'`
- o 因为两边的数据类型确实不一样，所以得到 true

5. `>=`

- o 比较左边的值是否 大于或等于 右边的值
- o `1 >= 1` true
- o `1 >= 0` true
- o `1 >= 2` false

6. `<=`

- o 比较左边的值是否 小于或等于 右边的值
- o `1 <= 2` true
- o `1 <= 1` true
- o `1 <= 0` false

7. `>`

- o 比较左边的值是否 大于 右边的值
- o `1 > 0` true
- o `1 > 1` false
- o `1 > 2` false

8. `<`

- o 比较左边的值是否 小于 右边的值

- `1 < 2` true
- `1 < 1` false
- `1 < 0` false

逻辑运算符

1. `&&`

- 进行 且 的运算
- 符号左边必须为 true 并且右边也是 true, 才会返回 true
- 只要有一边不是 true, 那么就会返回 false
- `true && true` true
- `true && false` false
- `false && true` false
- `false && false` false

2. `||`

- 进行 或 的运算
- 符号的左边为 true 或者右边为 true, 都会返回 true
- 只有两边都是 false 的时候才会返回 false
- `true || true` true
- `true || false` true
- `false || true` true
- `false || false` false

3. `!`

- 进行 取反 运算
- 本身是 true 的, 会变成 false
- 本身是 false 的, 会变成 true
- `!true` false
- `!false` true

自增自减运算符（一元运算符）

1. `++`

- 进行自增运算
- 分成两种, **前置++** 和 **后置++**
- 前置++, 会先把值自动 +1, 在返回

```
var a = 10;
console.log(++a);
// 会返回 11, 并且把 a 的值变成 11
```

- 后置++, 会先把值返回, 在自动+1

```
var a = 10;
console.log(a++);
// 会返回 10, 然后把 a 的值变成 11
```

2. --

- 进行自减运算
- 分成两种，前置-- 和 后置--
- 和 ++ 运算符道理一样

进制介绍

进制也就是进位计数制，是人为定义的带进位的计数方法。

十六进制是逢十六进一，十进制是逢十进一，八进制是逢八进一，二进制就是逢二进一 ...

在JavaScript中进制之间的转换提供了两个非常好用的方法：toString()、parseInt()。

使用 toString() 方法把十进制转为其他进制：

```
var x = 28; // 10进制
console.log(x.toString(2)); // 把十进转为2进制
console.log(x.toString(8)); // 把十进转为8进制
console.log(x.toString(16)); // 把十进转为16进制
```

使用 parseInt() 方法把其他进制转为十进制：

```
var x = "110"; // 二进制的字符串
console.log(parseInt(x, 2)); // 把这个字符串当做二进制，转为十进制
```

```
var x = "070"; // 八进制的字符串
console.log(parseInt(x, 8)); // 把这个字符串当做八进制，转为十进制
```

```
var x = "0x1c"; // 十六进制的字符串
console.log(parseInt(x, 16)); // 把这个字符串当做十六进制，转为十进制
```

parseInt() 方法，第一个参数为要转换的字符串，第二个参数指定字符串的进制，默认为十进制

其他进制的相互转换，先使用parseInt变为十进制，在利用toString变为其他进制。

在JavaScript中，八进制以 0 开头，十六进制以 0x 开头，可省略。

运算符优先级

优先级	运算类型	关联性	运算符
20	圆括号	n/a	(...)
19	成员访问	从左到右
	需计算的成员访问	从左到右	... [...]
	new (带参数列表)	n/a	new ... (...)
	函数调用	从左到右	... (...)
18	new (无参数列表)	从右到左	new ...
17	后置递增 (运算符在后)	n/a	... ++
	后置递减 (运算符在后)		... --
16	逻辑非	从右到左	! ...
	按位非		~ ...
	一元加法		+ ...

	一元减法		<code>- ...</code>
	前置递增		<code>++ ...</code>
	前置递减		<code>-- ...</code>
	typeof		<code>typeof ...</code>
	void		<code>void ...</code>
	delete		<code>delete ...</code>
	await		<code>await ...</code>
15	幂	从右到左	<code>... ** ...</code>
14	乘法	从左到右	<code>... * ...</code>
	除法		<code>... / ...</code>
	取模		<code>... % ...</code>
13	加法	从左到右	<code>... + ...</code>

	减法		<code>... - ...</code>
12	按位左移	从左到右	<code>... << ...</code>
	按位右移		<code>... >> ...</code>
	无符号右移		<code>... >>> ...</code>
11	小于	从左到右	<code>... < ...</code>
	小于等于		<code>... <= ...</code>
	大于		<code>... > ...</code>
	大于等于		<code>... >= ...</code>
	in		<code>... in ...</code>
	instanceof		<code>... instanceof ...</code>
10	等号	从左到右	<code>... == ...</code>
	非等号		<code>... != ...</code>

	全等号		... == ...
	非全等号		... != ...
9	按位与	从左到右	... & ...
8	按位异或	从左到右	... ^ ...
7	按位或	从左到右
6	逻辑与	从左到右	... && ...
5	逻辑或	从左到右
4	条件运算符	从右到左	... ? ... : ...
3	赋值	从右到左	... = ...
			... += ...
			... -= ...
			... *= ...

			<code>... /= ...</code>
			<code>... %= ...</code>
			<code>... <<= ...</code>
			<code>... >>= ...</code>
			<code>... >>>= ...</code>
			<code>... &= ...</code>
			<code>... ^= ...</code>
			<code>... = ...</code>
2	<code>yield</code>	从右到左	<code>yield ...</code>
	<code>yield*</code>		<code>yield* ...</code>
1	展开运算符	n/a	<code>... ...</code>
0	逗号	从左到右	<code>... , ...</code>