Javascript

学习JavaScript这一篇就够了_javascript学习这一篇就够了-CSDN博客

JavaScript基础（超详细全文约一万字）-CSDN博客

第一章 JavaScript简介

1.1、JavaScript的起源

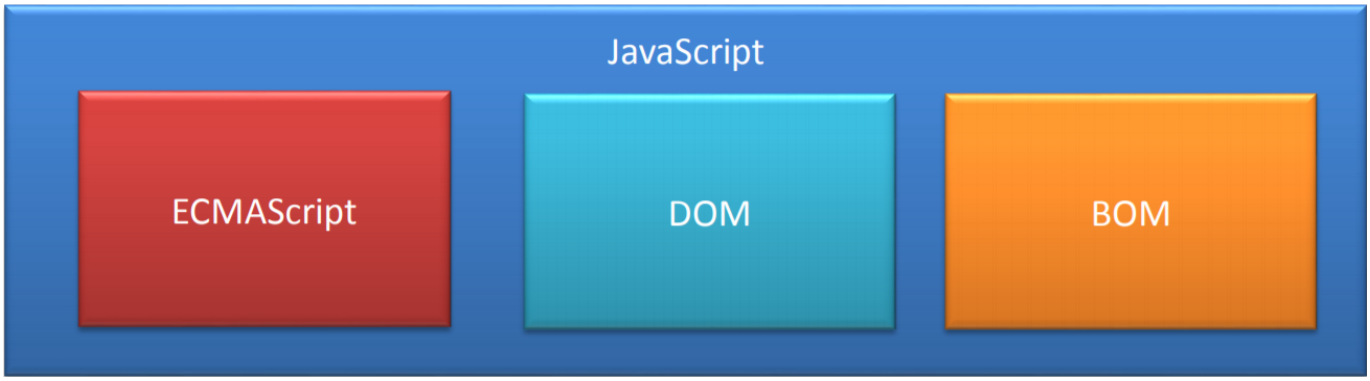
JavaScript诞生于1995年，它的出现主要是用于处理网页中的前端验证。所谓的前端验证，就是指检查用户输入的内容是否符合一定的规则。比如：用户名的长度，密码的长度，邮箱的格式等。但是，有的同学可能会有疑问，这些验证，后端不也可以进行验证吗？确实，后端程序的确可以进行这些验证，但你要清楚，在1995年那个年代，网速是非常慢的，向后端发送一个请求，浏览器很久才能得到响应，那这无疑是一种非常不好的用户体验。

1.2、JavaScript的组成

ECMAScript是一个标准，而这个标准需要由各个厂商去实现，不同的浏览器厂商对该标准会有不同的实现。

浏览器	JavaScript实现方式
FireFox	SpiderMonkey
Internet Explorer	JScript/Chakra
Safari	JavaScriptCore
Chrome	v8
Carakan	Carakan

我们已经知道ECMAScript是JavaScript标准，所以一般情况下这两个词我们认为是一个意思。但是实际上JavaScript的含义却要更大一些。一个完整的JavaScript实现应该由以下三个部分构成：



一般认为JavaScript由三部分组成

- ECMAScript: 基础语法
- DOM: 文档数据模型
- BOM: 浏览器对象模型

1.2.1 ECMAScript

ECMAScript 是由ECMA（原欧洲计算机制造商协会）进行标准化的一门编程语言, 主要规定了像变量, 数据类型, 流程控制, 函数等基础语法

1.2.2 DOM和BOM

W3C: 万维网联盟 (World Wide Web Consortium) 主要是完成HTML和CSS及浏览器标准化的研究, 是一个非盈利性的公益组织, 主要由大公司和开发人员组成

其中,

- DOM是由W3C组织制定的标准, 通过 DOM 提供的接口可以对页面上的各种元素进行操作（大小、位置、颜色、事件等）
- BOM是由各个浏览器厂商根据DOM在各自浏览器上的实现, 不同的浏览器会略有差异, 通过BOM可以操作浏览器窗口，比如弹出框、控制浏览器跳转、获取分辨率等

1.2.3 JS的写在哪里

跟CSS一样, JS也有3种书写方式

- 外部: 将JS文件单独保存

```
通过< script src="xxx.js ">引入

<script src="main.js"></script>

main.js:
alert("Hello,World!");
```

- 内嵌: 在HTML文件中, 将JS代码写在< script >标签中
- 行内: 现在几乎不用

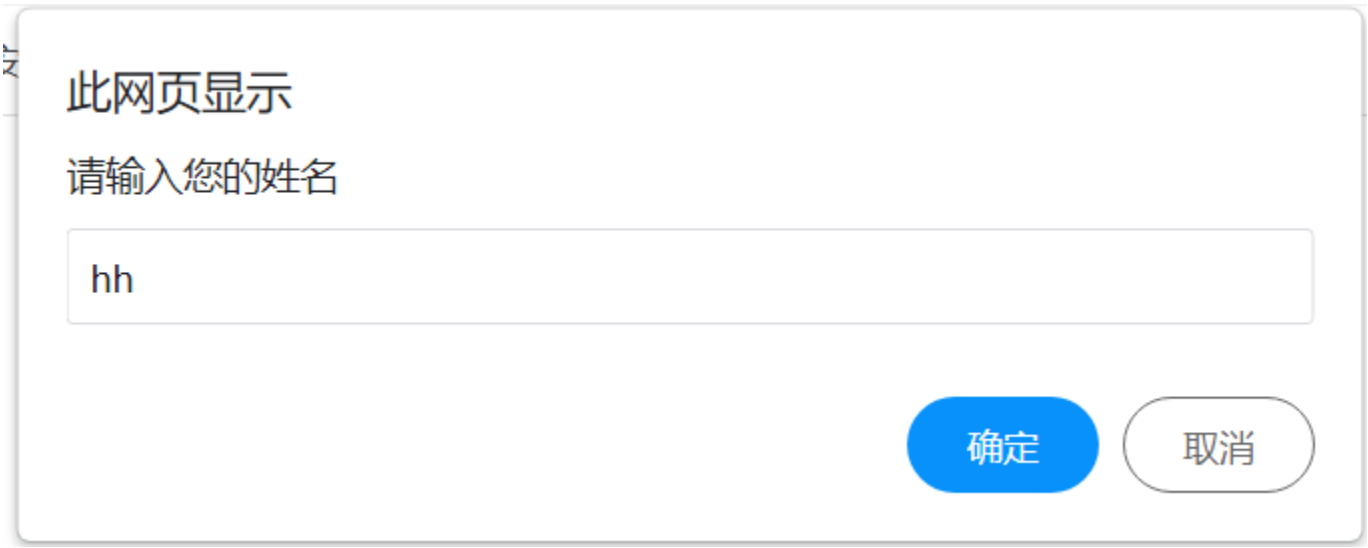
1.2.4 体验JS

为了方便信息的输入输出，JS中提供了一些输入输出语句，其常用的语句如下：

方法	说明	归属
alert(msg)	浏览器弹出警示框	浏览器BOM
console.log(msg)	浏览器控制台打印输出信息	浏览器BOM
prompt(info)	浏览器弹出输入框，用户可以输入	浏览器BOM

- 注意：alert() 主要用来显示消息给用户，console.log() 用来给程序员自己看运行时的消息

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script>
        // 这是一个输入框
        var name = prompt('请输入您的姓名')
        // alert 弹出警示框 输出的 展示给用户的
        alert(name + ' 你好!')
        // console 控制台输出 给程序员测试用的
        console.log('我是程序员能看到的:' + name)
    </script>
</body>
</html>
```



在Chrome浏览器中, 使用F12或者Ctrl+Shift+I打开调试窗口, 在console控制台中查看

在控制台中, 也可以编写JS的代码

1.2.5 JavaScript的使用

(1)页面输出

使用JavaScript向页面输出一句话

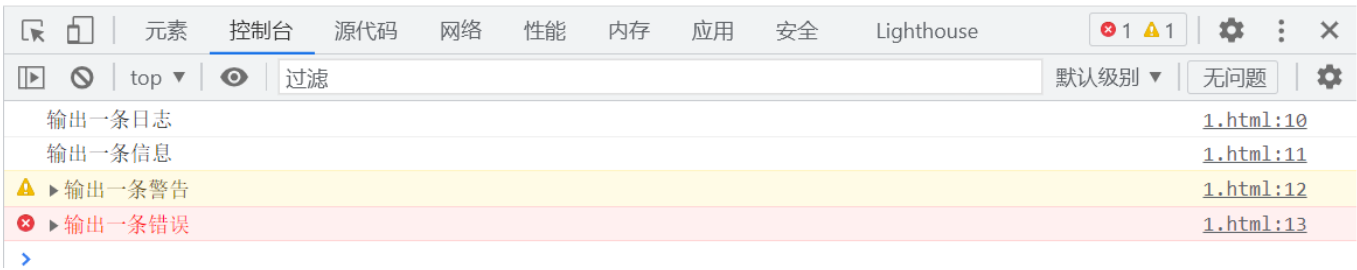
```
<script>
    document.write("Hello,World!");
</script>
```

Hello,World!

(2)**控制台输出**

使用JavaScript向控制台输出一句话

```
<script>
    console.log("输出一条日志");//最常用
    console.info("输出一条信息");
    console.warn("输出一条警告");
    console.error("输出一条错误");
</script>
```



(3)**弹出窗口输出**

使用JavaScript向弹出窗口输出一句话

```
<script>
    alert("Hello,World!");
</script>
```

1.2.6 JavaScript的注释

注释中的内容不会被解析器解析执行，但是会在源码中显示，我们一般会使用注释对程序中的内容进行解释。

JS中的注释和Java的的一致，分为两种：

- 单行注释：// 注释内容
- 多行注释：/* 注释内容 */

第二章 JavaScript基础语法

2.1、标识符

所谓标识符，就是指给变量、函数、属性或函数的参数起名字。

标识符可以是按照下列格式规则组合起来的一或多个字符：

- 第一个字符必须是一个字母、下划线（ _ ）或一个美元符号（ $ ）。
- 其它字符可以是字母、下划线、美元符号或数字。
- 按照惯例，ECMAScript 标识符采用驼峰命名法。驼峰法 (首字母小写，后面单词的首字母需要大写)
- 标识符不能是关键字和保留字符。

关键字：

break	do	instanceof	typeof	case
else	new	var	catch	finally
return	void	continue	for	switch
while	default	if	throw	delete
in	try	function	this	with
debugger	false	true	null	

保留字符：

class	enum	extends	super	const	export
import	implements	let	private	public	yield
interface	package	protected	static		

其它不建议使用的标识符：

abstract	double	goto	native	static	boolean
enum	implements	package	super	byte	export
import	private	synchronize	char	extends	int
protected	throws	class	final	interface	public
transient	const	float	long	short	volatile
arguments	encodeURI	Infinity	Number	RegExp	undefined
isFinite	Object	String	Boolean	Error	RangeError
parseFloat	SyntaxError	Date	eval	JSON	ReferenceError
TypeError	decodeURI	EvalError	Math	URIError	decodeURIComponent
Function	NaN	isNaN	parseInt	Array	encodeURICOmponent

2.2、字面量和变量

2.2.1、字面量

字面量实际上就是一些固定的值，比如：1、2 、3、true、false、null、NaN、“hello”，字面量都是不可以改变的，由于字面量不是很方便使用，所以在JavaScript中很少直接使用字面量，使用的而是变量。

2.2.2、变量

变量的作用是给某一个值或对象标注名称。比如我们的程序中有一个值123，这个值我们是需要反复使用的，这个时候 我们最好将123这个值赋值给一个变量，然后通过变量去使用123这个值。

- 变量的声明： 使用var关键字声明一个变量。

```
var a;
```

- 变量的赋值： 使用=为变量赋值。

```
a = 123;
```

- 声明和赋值同时进行：

```
var a = 123;
```

2.3、数据类型

2.3.1、类型分类

数据类型决定了一个数据的特征，比如：123和"123"，直观上看这两个数据都是123，但实际上前者是一个数字，而后者是一个字符串。

对于不同的数据类型我们在进行操作时会有很大的不同。

JavaScript中一共有5种基本数据类型：

- 字符串型（String）
- 数值型（Number）
- 布尔型（Boolean）
- undefined型（Undefined）
- null型（Null） 这5种之外的类型都称为Object，所以总的来看JavaScript中共有六种数据类型。

2.3.2、typeof运算符

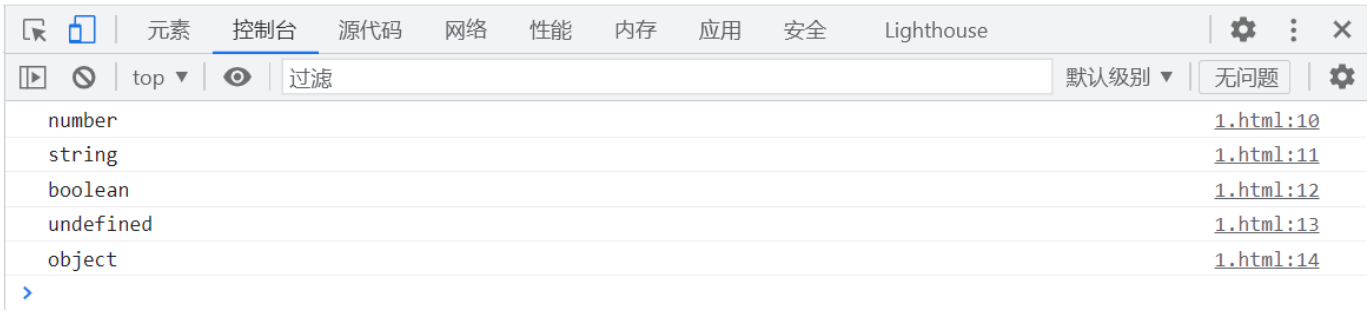
使用typeof操作符可以用来检查一个变量的数据类型。

使用方式：

```
typeof 数据
```

示例代码：

```
<script>
    console.log(typeof 123);
    console.log(typeof "Hello,World!");
    console.log(typeof true);
    console.log(typeof undefined);
    console.log(typeof null);
</script>
```



2.3.3、String

String用于表示一个字符序列，即字符串。字符串需要使用 单引号 或 双引号 括起来。

转义字符：

转义字符	含义	转义字符	含义
\n	换行	\\	斜杠
\t	制表	\’	单引号
\b	空格	\”	双引号
\r	回车		

> 注意：使用typeof运算符检查字符串时，会返回"string"。

2.3.4、Number

Number 类型用来表示整数和浮点数，最常用的功能就是用来表示10进制的整数和浮点数。

Number表示的数字大小是有限的，如果超过了这个范围，则会返回 ±Infinity。

- 最大值：+1.7976931348623157e+308
- 最小值：-1.7976931348623157e+308
- 0以上的最小值：5e-324

特殊的数字:

- Infinity：正无穷
- -Infinity：负无穷
- NaN：非法数字（Not A Number）

其它的进制：

- 二进制：0b 开头表示二进制，但是，并不是所有的浏览器都支持
- 八进制：0 开头表示八进制
- 十六进制：0x 开头表示十六进制

> 注意：使用typeof检查一个Number类型的数据时（包括NaN 和 Infinity），会返回"number"。

2.3.5、Boolean

布尔型也被称为逻辑值类型或者真假值类型。

布尔型只能够取真（true）和假（false）两种数值。除此以外， 其它的值都不被支持。

2.3.6、Undefined

Undefined 类型只有一个值，即特殊的 undefined。

在使用 var 声明变量但未对其加以初始化时，这个变量的值就是 undefined。

> 注意：使用typeof对没有初始化和没有声明的变量，会返回“undefined”。

2.3.7、Null

Null 类型是第二个只有一个值的数据类型，这个特殊的值是 null。

undefined值实际上是由null值衍生出来的，所以如果比较undefined和null是否相等，会返回 true。

> 注意：从语义上看null表示的是一个空的对象，所以使用typeof检查null会返回一个Object。

2.4、强制类型转换

强制类型转换指将一个数据类型强制转换为其它的数据类型。一般是指，将其它的数据类型转换为String、Number、Boolean。

2.4.1、转换为String类型

将其它数值转换为字符串有三种方式：toString()、String()、 拼串。

- 方式一

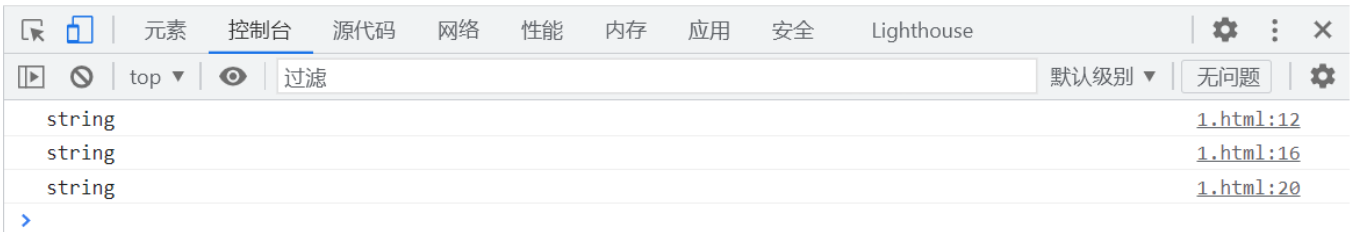
调用被转换数据类型的toString()方法，该方法不会影响到原变量，它会将转换的结果返回，但是注意：null和undefined这两个值没有toString()方法，如果调用它们的方法，会报错。

```
<script>
    var a = 123;
    console.log(a);
    console.log(typeof a);
    a = a.toString();
    console.log(a);
    console.log(typeof a);
</script>
```

- 方式二

调用String()函数，并将被转换的数据作为参数传递给函数，使用String()函数做强制类型转换时，对于Number和Boolean实际上就是调用的toString()方法，但是对于null和undefined，就不会调用toString()方法，它会将 null 直接转换为 “null”，将 undefined 直接转换为 “undefined”。

```
<script>
    var a = 123;
    a = String(a);
    console.log(typeof a);

    var b = undefined;
    b = String(b);
    console.log(typeof b);

    var c = null;
    c = String(c);
    console.log(typeof c);
</script>
```



- 方式三：为任意的数据类型 +""

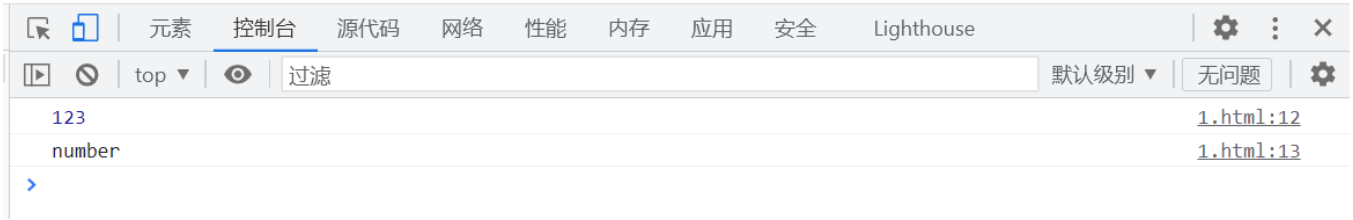
```
<script>
    var a = 123;
    a = a + "";
    console.log(a);
    console.log(typeof a);
</script>
```

2.4.2、转换为Number类型

有三个函数可以把非数值转换为数值：Number()、parseInt() 和parseFloat()。Number()可以用来转换任意类型的数据，而后两者只能用于转换字符串。parseInt()只会将字符串转换为整数，而parseFloat()可以将字符串转换为浮点数。

- 方式一：使用Number()函数
    - 字符串 --> 数字
        - 如果是纯数字的字符串，则直接将其转换为数字
        - 如果字符串中有非数字的内容，则转换为NaN
        - 如果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
    - 布尔 --> 数字
        - true 转成 1
        - false 转成 0
    - null --> 数字
        - null 转成 0
    - undefined --> 数字
        - undefined 转成 NaN
- 方式二：这种方式专门用来对付字符串，parseInt() 把一个字符串转换为一个整数

```
<script>
    var a = "123";
    a = parseInt(a);
    console.log(a);
    console.log(typeof a);
</script>
```



- 方式三：这种方式专门用来对付字符串，parseFloat() 把一个字符串转换为一个浮点数

```
<script>
    var a = "123.456";
    a = parseFloat(a);
    console.log(a);
    console.log(typeof a);
</script>
```

> 注意：如果对非String使用parseInt()或parseFloat()，它会先将其转换为String然后在操作

2.4.3、转换为Boolean类型

将其它的数据类型转换为Boolean，只能使用Boolean()函数。

使用Boolean()函数

- 数字 —> 布尔 除了0和NaN，其余的都是true
- 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

2.5、运算符

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。

比如：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）

2.5.1、算术运算符

算术运算符用于表达式计算。

y=5，下面的表格解释了这些算术运算符：

运算符	描述	例子	x 运算结果	y 运算结果
+	加法	x=y+2	7	5
-	减法	x=y-2	3	5
*	乘法	x=y*2	10	5
/	除法	x=y/2	2.5	5
%	取模（求余数）	x=y%2	1	5
++	自增	x=++y x=y++	6 5	6 6
–	自减	x=–y x=y–	4 5	4 4

```
<p>假设 y=5，计算 x=y+2，并显示结果。</p>
<button onclick="myFunction()">点击这里</button>
<p id="demo"></p>
<script>
    function myFunction() {
        var y = 5;
        var x = y + 2;
        var demoP = document.getElementById("demo");
        demoP.innerHTML = "x=" + x;
    }
</script>
```

假设 y=5，计算 x=y+2，并显示结果。

点击这里

x=7

2.5.2、关系运算符

关系运算符在逻辑语句中使用，以测定变量或值是否相等。

x=5，下面的表格解释了比较运算符：

运算符	描述	比较	返回值
>	大于	x>8	*false*
<	小于	x<8	*true*
>=	大于或等于	x>=8	*false*
<=	小于或等于	x<=8	*true*

2.5.3、赋值运算符

赋值运算符用于给 JavaScript 变量赋值。

x=10 和 y=5，下面的表格解释了赋值运算符：

运算符	例子	等同于	运算结果
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

2.5.4、逻辑运算符

逻辑运算符用于测定变量或值之间的逻辑。

给定 x=6 以及 y=3，下表解释了逻辑运算符：

运算符	描述	例子
&&	and	(x < 10 && y > 1) 为 true
||	or	(x==5 || y==5) 为 false
!	not	!(x==y) 为 true

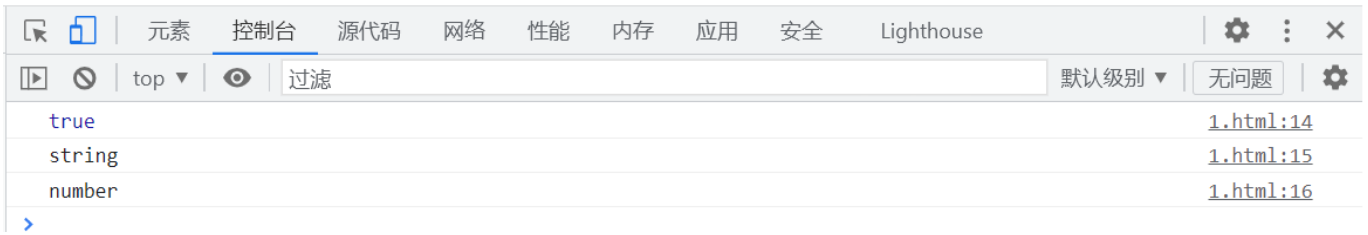
2.5.5、比较运算符

比较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
- 使用 != 来做不相等运算 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，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
- 使用 === 来做全等运算 用来判断两个值是否全等，它和相等类似，不同的是它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

- 使用 !== 来做不全等运算 用来判断两个值是否不全等，它和不等类似，不同的是它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

```
<script>
        var y = 5;
        var x = '5';
        console.log(x==y);
        console.log(typeof x);
        console.log(typeof y);
</script>
```



2.5.6、条件运算符

JavaScript 还包含了基于某些条件对变量进行赋值的条件运算符。

```
语法：variablename=(condition)?value1:value2;

举例：result=(age<18)?"年龄太小":"年龄合适";
```

执行流程：如果condition为true，则执行语句1，并返回执行结果，如果为false，则执行语句2，并返回执行结果。

2.5.7、逗号运算符

使用逗号可以在一条语句中执行多次操作。

比如：var num1=1, num2=2, num3=3;

使用逗号运算符分隔的语句会从左到右顺 序依次执行。

2.6、运算符优先级

运算符优先级由上到下依次减小，对于同级运算符，采用从左向右依次执行的方法。

- .、[]、 new
- ()
- ++、 --
- !、~、+(单目)、-(单目)、typeof、void、delete
- %、*、/
- +(双目)、-(双目)
- << 、 >>、 >>>
- <、<=、>、>=
- ==、!==、===
- &
- ^
- |
- &&
- ||
- ?:
- =、+=、-=、*=、/=、%=、<<=、>>=、>>>=、&=、^=、|=
- ,

2.7、代码块

2.7.1、语句

前边我所说表达式和运算符等内容可以理解成是我们一 门语言中的单词，短语。而语句（statement）就是我们这个语言中一句一句完 整的话了。语句是一个程序的基本单位，JavaScript的程序就是由一条一条语句构成的，每一条语句使用;结尾。

JavaScript中的语句默认是由上至下顺序执行的，但是我们也可以通过一些流程控制语句来控制语句的执行顺序。

2.7.2、代码块

代码块是在大括号 {} 中所写的语句，以此将多条语句的集合视为一条语句来使用。

例如：

```
{
    var a = 123;
    a++;
    alert(a);
}
```

我们一般使用代码块将需要一起执行的语句进行分组，需要注意的是，代码块结尾不需要加 分号。

2.8、条件语句

条件语句是通过判断指定表达式的值来决定执行还是跳过某些语句，最基本的条件语句：

- if...else
- switch...case

2.8.1、if...else

if...else语句是一种最基本的控制语句，它让JavaScript可以有条件的执行语句。

- 第一种形式：

```
if(expression)
    statement
```

```
<script>
    var age = 16;
    if (age < 18) {
        console.log("未成年");
    }
</script>
```

- 第二种形式：

```
if(expression)
    statement
else
    statement
```

```
<script>
    var age = 16;
    if (age < 18) {
        console.log("未成年");
    } else {
        console.log("已成年");
}
</script>
```

- 第三种形式

```
if(expression1)
    statement
else if(expression2)
    statement
else
    statement
```

```
<script>
    var age = 18;
    if (age < 18) {
        console.log("小于18岁了");
    } else if (age == 18) {
        console.log("已经18岁了");
    } else {
        console.log("大于18岁了")
}
</script>
```

2.8.2、switch...case

switch...case是另一种流程控制语句。

switch语句更适用于多条分支使用同一条语句的情况。

语法格式：

```
switch (语句) {
    case 表达式1:
        语句...
    case 表达式2:
        语句...
    default:
        语句...
}
```

> 注意：需要注意的是一旦符合case的条件程序会一直运行到结束，所以我们一般会在case中添加break作为语句的结束。

```
<script>
        var today = 1;
        switch (today) {
            case 1:
                console.log("星期一");
                break;
            case 2:
                console.log("星期二");
                break;
            case 3:
                console.log("星期三");
                break;
            case 4:
                console.log("星期四");
                break;
            case 5:
                console.log("星期五");
                break;
            case 6:
                console.log("星期六");
                break;
            case 7:
                console.log("星期日");
                break;
            default:
                console.log("输入错误");
        }

</script>
```

2.9、循环语句

循环语句和条件语句一样，也是基本的控制语句，只要满足一定的条件将会一直执行，最基本的循环语句：

- while
- do...while
- for

2.9.1、while

while语句是一个最基本的循环语句，while语句也被称为while循环。

语法格式：

```
while(条件表达式){
    语句...
}
```

```
<script>
    var i = 1;
    while (i <= 10) {
        console.log(i);
        i++;
}
</script>
```

2.9.2、do...while

do...while和while非常类似，只不过它会在循环的尾部而不是顶部检查表达式的值，因此，do...while循环会至少执行一次。相比于while，do...while的使用情况并不 是很多。

语法格式：

```
do{
    语句...
}while(条件表达式);
```

```
<script>
    var i = 1;
    do {
        console.log(i);
        i++;
    } while (i <= 10);
</script>
```

2.9.3、for

for语句也是循环控制语句，我们也称它为for循环。大部分循环都会有一个计数器用以控制循环执行的次数， 计数器的三个关键操作是初始化、检测和更新。for语句 就将这三步操作明确为了语法的一部分。

语法格式：

```
for(初始化表达式 ; 条件表达式 ; 更新表达式){
    语句...
}
```

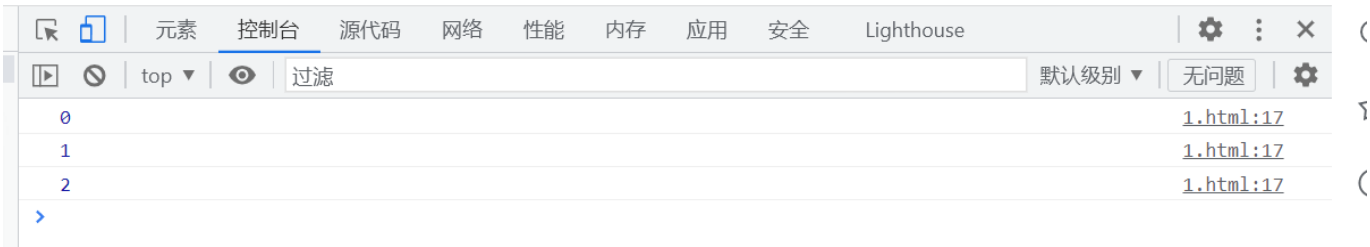
```
<script>
    for (var i = 1; i <= 10; i++) {
        console.log(i);
    }
</script>
```

2.9.4、跳转控制

- break：结束最近的一次循环，可以在循环和switch语句中使用。
- continue：结束本次循环，执行下一次循环，只能在循环中使用。

那如果我们想要跳出多层循环或者跳到指定位置该怎么办呢？可以为循环语句创建一个label，来标识当前的循环，如下例子：

```
<script>
        outer: for (var i = 0; i < 5; i++) {
                for (var j = 0; j < 5; j++) {
                    if (j == 3) {
                        break outer;
                    }
                    console.log(j);
                }
            }
</script>
```



2.10、对象基础

2.10.1、概述

Object类型，我们也称为一个对象，是JavaScript中的引用数据类型。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

个名/值对。对象除了可以创建自有属性，还可以通过从一个名为原型的对象那里继承属性。除了字符串、数字、true、false、null和undefined之外，JavaScript中的值都是对象。

2.10.2、创建对象

创建对象有两种方式：

- 第一种方式：

```
<script>
    var person = new Object();
    person.name = "孙悟空";
    person.age = 18;
    console.log(person);
</script>
```

- 第二种方式：

```
<script>
    var person = {
    name: "孙悟空",
    age: 18
};
    console.log(person);
</script>
```

2.10.3、访问属性

访问属性的两种方式：

- 第一种方式：使用 . 来访问

```
对象.属性名
```

- 第二种方式：使用 [] 来访问

```
对象['属性名']
```

2.10.4、删除属性

删除对象的属性可以使用delete关键字，格式如下：

```
delete 对象.属性名
```

```
<script>
    var person = new Object();
    person.name = "孙悟空";
    person.age = 18;
    console.log(person);
    delete person.name
    console.log(person);
</script>
```

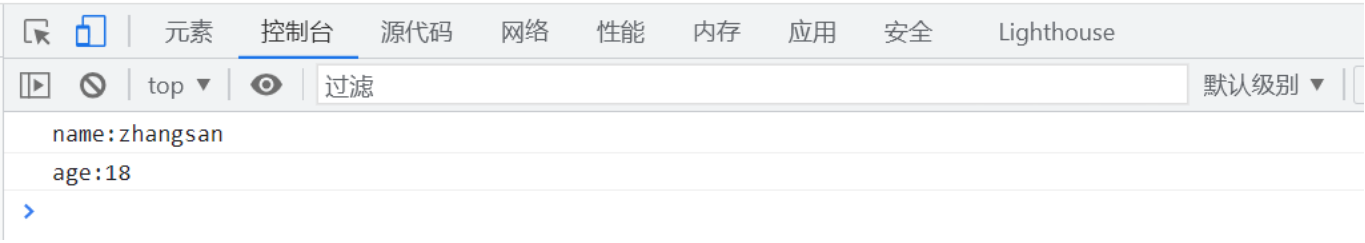
2.10.5、遍历对象

枚举遍历对象中的属性，可以使用for ... in语句循环，对象中有几个属性，循环体就会执行几次。

语法格式：

```
for (var 变量 in 对象) {

}
```

```
<script>
    var person = {
    name: "zhangsan",
    age: 18
    }
    for (var personKey in person) {
        var personVal = person[personKey];
        console.log(personKey + ":" + personVal);
    }
</script>
```



2.10.6、数据类型梳理

2.10.6.1、基本数据类型

JavaScript中的变量可能包含两种不同数据类型的值：基本数据类型和引用数据类型。

JavaScript中一共有5种基本数据类型：String、Number、 Boolean、Undefined、Null。

基本数据类型的值是无法修改的，是不可变的。

基本数据类型的比较是值的比较，也就是只要两个变量的值相等，我们就认为这两个变量相等。

2.10.6.2、引用数据类型

引用类型的值是保存在内存中的对象。

当一个变量是一个对象时，实际上变量中保存的并不是对象本身，而是对象的引用。

当从一个变量向另一个变量复制引用类型的值时，会将对象的引用复制到变量中，并不是创建一个新的对象。

这时，两个变量指向的是同一个对象。因此，改变其中一个变量会影响另一个。

2.10.7、栈和堆梳理

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，堆内存是用来保存对象。

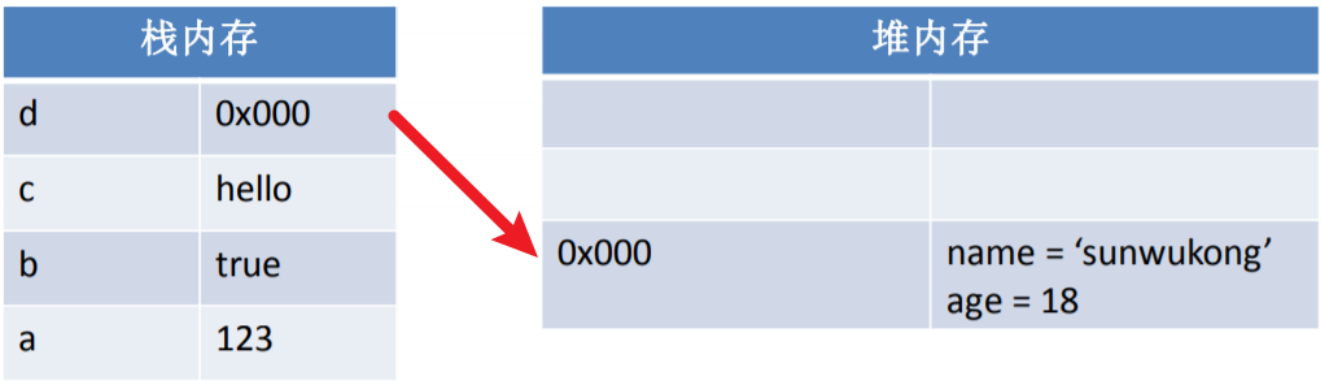
我们在声明一个变量时，实际上就是在栈内存中创建了一个空间用来保存变量。

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。

当我们写了下边这几句代码的时候，栈内存和堆内存的结构如下：

```
<script>
    var a = 123;
    var b = true;
    var c = "hello";
    var d = {name: 'sunwukong', age: 18};
</script>
```

栈的特点：先进后出，后进先出



2.11、函数

2.11.1、概述

函数是由一连串的子程序（语句的集合）所组成的，可以被外部程序调用，向函数传递参数之后，函数可以返回一定的值。

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。

这里要注意的是JavaScript中的函数也是一个对象，使用typeof检查一个函数对象时，会返回function。

2.11.2、函数创建

- 使用 函数对象 来创建一个函数（几乎不用）

语法格式：

```
var 函数名 = new Function("执行语句");
```

```
<script>
    var fun = new Function("console.log('这是我的第一个函数');");
</script>
```

- 使用 函数声明 来创建一个函数（比较常用）

语法格式：

```
function 函数名([形参1,形参2,...,形参N]) {
    语句...
}
```

```
<script>
    function fun(){
        console.log("这是我的第二个函数");
}
</script>
```

- 使用 函数表达式 来创建一个函数（比较常用）

语法格式：

```
var 函数名  = function([形参1,形参2,...,形参N]) {
    语句....
}
```

```
<script>
    var fun  = function() {
        console.log("这是我的第三个函数");
}
</script>
```

2.11.3、函数调用

- 对于无参函数调用：

```
<script>
    // 函数声明
var fun = function () {
    console.log("哈哈，我执行啦！");
}

// 函数调用
fun();
</script>
```

- 对于有参函数调用：

```
<script>
    // 函数声明
var sum = function (num1, num2) {
    var result = num1 + num2;
    console.log("num1 + num2 = " + result);
}

// 函数调用
sum(10, 20);
</script>
```

2.11.4、函数参数

- 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，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
- 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

- 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

2.11.5、函数返回值

可以使用 return 来设置函数的返回值，return后的值将会作为函数的执行结果返回，可以定义一个变量，来接收该结果

> 注意：在函数中return后的语句都不会执行，如果return语句后不跟任何值就相当于返回一个undefined，如果函数中不写return，则也会返回undefined，return后可以跟任意类型的值

```
<script>
    function sum(num1, num2) {
        return num1 + num2;
    }
    var result = sum(10, 20);
    console.log(result);
</script>
```

2.11.6、嵌套函数

嵌套函数：在函数中声明的函数就是嵌套函数，嵌套函数只能在当前函数中可以访问，在当前函数外无法访问。

案例演示：

```
<script>
    function fu() {
    function zi() {
        console.log("我是儿子")
    }

    zi();
}

fu();

</script>
```

2.11.7、匿名函数

匿名函数：没有名字的函数就是匿名函数，它可以让一个变量来接收，也就是用 “函数表达式” 方式创建和接收。

案例演示：

```
<script>
        var fun = function () {
            alert("我是一个匿名函数");
        }
    fun();
    </script>
```

2.11.8、立即执行函数

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。

案例演示：

```
<script>
        (function () {
            alert("我是一个立即执行函数");
        })();
    </script>
```

2.11.9、对象中的函数

对象的属性值可以是任何的数据类型，也可以是个函数。

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）。

注意：方法和函数只是名称上的区别，没有其它别的区别

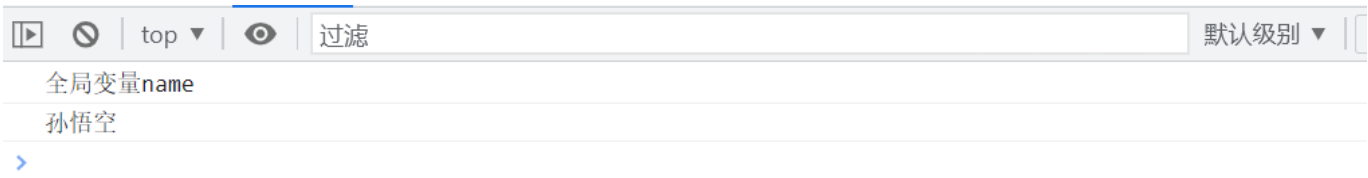
```
<script>
        var person = {
            name: "zhangsan",
            age: 18,
            sayHello: function () {
            console.log(name + " hello")
    }
}
        person.sayHello();
    </script>
```

2.11.10、this对象

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，这个隐含的参数就是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

- 以函数的形式调用时，this永远都是window
- 以方法的形式调用时，this就是调用方法的那个对象 案例演示：

```
<script>
        //创建一个全局变量name
        var name = "全局变量name";

        //创建一个函数
        function fun() {
            console.log(this.name);
        }

        fun();

        //创建一个对象
        var obj = {
            name: "孙悟空",
            sayName: fun
        };

        //我们希望调用obj.sayName()时可以输出obj的名字而不是全局变量name的名字
        obj.sayName();

    </script>
```



JavaScript中的this详解_js this-CSDN博客

> this是和执行上下文相关的对象，只有在运行的时候才能确定this的值；

this主要应用在：

- 构造函数中的this；
- 普通函数中的this；
- 对象函数中的this；
- call、apply、bind改变this
- 箭头函数中的this；

(1)构造函数中的this

```
<script>
        function Person(name) {
        this.name = name;
        }
        const p = new Person('xiaoming');
        console.log(p.name);
    </script>
```

(2)**普通函数中的**this

```
<script>
        function getVal() {
            console.log(this); //window
        }
        getVal();
    </script>
```

- 1.this总是代表它的直接调用者(js的this是执行上下文), 例如 obj.func ,那么func中的this就是obj
- 2.在默认情况(非严格模式下,未使用 ‘use strict’),没找到直接调用者,则this指的是 window （常见的window的属性和方法有: alert,location,document,parseInt,setTimeout,setInterval等）(约定俗成)
- 3.在严格模式下,没有直接调用者的函数中的this是 undefined
- 4.使用call,apply,bind(ES5新增)绑定的,this指的是 绑定的对象

```
<script>
        var factory = function () {
            this.a = 'a'
               this.b = 'b'
            this.c = {
            a: 'a+',
            b: function () {
                return this.a
            }
            }
        }
        console.log(new factory().c.b()) //a+
    </script>
```

```
<script>
        var factory = function () {
            this.a = 'a'
            this.b = 'b'
            this.c = {
            a: 'a+',
            b: () => {
                    return this.a
            }
        }
        }
        console.log(new factory().c.b())  //a
    </script>
```

(3)对象函数中的this

注意是对象函数中的this，不是对象中的this，对象中的this还属于上一级作用域中的this； 如果函数是声明式函数，则this为当前对象，如果函数是箭头函数，则this去上一级作用域链找；call并不能改变箭头函数中的this，因为箭头函数不会自己创建this；

(4)call、apply、bind改变this

多次bind也只绑定了第一次的bind的对象，因为bind内部实现里返回了一个函数，如果这个函数没有执行，那么绑定的oThis永远是第一次bind的this。

(5)箭头函数和普通函数的区别

- 普通函数：根据调用我的人（谁调用我，我的this就指向谁）
- 箭头函数：MDN官方文档里面描述箭头函数this定义的时候，描述的是"箭头函数不会创建自己的this,它只会从自己的作用域链的上一层继承this。"，因此箭头函数里的this也不能够通过

call方法来改变。

2.12、对象进阶

2.12.1、用工厂方法创建对象

```
<script>
        // 使用工厂模式创建对象
        function createPerson(name, age) {
            // 创建新的对象
            var obj = new Object();
            // 设置对象属性
            obj.name = name;
            obj.age = age;
            // 设置对象方法
            obj.sayName = function () {
                console.log(this.name);
            };
            //返回新的对象
            return obj;
        }

        for (var i = 1; i <= 1000; i++) {
            var person = createPerson("person" + i, 18);
            console.log(person);
        }

    </script>
```

```
▼Object                                                    1.html:28
    age: 18
    name: "person1"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▼Object                                                    1.html:28
    age: 18
    name: "person2"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▼Object                                                    1.html:28
    age: 18
    name: "person3"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▼Object                                                    1.html:28
    age: 18
    name: "person4"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▶ Object                                                   1.html:28
▶ Object                                                   1.html:28
```

2.12.2、用构造函数创建对象

在前一节中，我们学会了使用工厂模式创建对象，但是，你会发现我们所创建的对象类型都是Object;

每创建一个都是对象，但是在实际生活中，人应该是一个确定的类别，属于人类，对象是一个笼统的称呼，万物皆对象，它并不能确切的指明当前对象是人类，那我们要是既想实现创建对象的功能，同时又能明确所创建出来的对象是人类，那么似乎问题就得到了解决，这就用到了构造函数，每一个构造函数你都可以理解为一个类别，用构造函数所创建的对象我们也成为类的实例

```
<script>
        // 使用构造函数来创建对象
        function Person(name, age) {
            // 设置对象的属性
            this.name = name;
            this.age = age;
            // 设置对象的方法
            this.sayName = function () {
                console.log(this.name);
            };
        }

        var person1 = new Person("孙悟空", 18);
        var person2 = new Person("猪八戒", 19);
        var person3 = new Person("沙和尚", 20);

        console.log(person1);
        console.log(person2);
        console.log(person3);

    </script>
```

```
▼Person
    age: 18
    name: "孙悟空"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▼Person
    age: 19
    name: "猪八戒"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
▼Person
    age: 20
    name: "沙和尚"
  ▶ sayName: ƒ ()
  ▶ [[Prototype]]: Object
>
```

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还有一个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。

那构造函数是怎么执行创建对象的过程呢？我再来解释一下

- 1.调用构造函数，它会立刻创建一个新的对象
- 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

你会发现构造函数有点类似工厂方法，但是它创建对象和返回对象都给我们隐藏了，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。

现在，this又出现了一种新的情况，为了不让大家混淆，我再来梳理一下：

当以函数的形式调用时，this是window 当以方法的形式调用时，谁调用方法this就是谁 当以构造函数的形式调用时，this就是新创建的那个对象 我们可以使用 instanceof 运算符检查一个对象是否是一个类的实例，它返回true或false

语法格式：

```
对象 instanceof 构造函数
```

```
<script>
    console.log(person1 instanceof Person);
</script>
```

2.12.3、原型

在前一节中，我们学习了使用构造函数的方式进行创建对象，但是，它还是存在一个问题，那就是，你会发现，每一个对象的属性不一样这是一定的，但是它的方法似乎好像是一样的，如果我创建1000个对象，那岂不是内存中就有1000个相同的方法，那要是有10000个，那对内存的浪费可不是一点半点的，我们有没有什么好的办法解决，没错，我们可以把函数抽取出来，作为全局函数，在构造函数中直接引用就可以了，上代码：

```
<script>
        // 使用构造函数来创建对象
        function Person(name, age) {
            // 设置对象的属性
            this.name = name;
            this.age = age;
            // 设置对象的方法
            this.sayName = sayName
        }

        // 抽取方法为全局函数
        function sayName() {
            console.log(this.name);
        }

        var person1 = new Person("孙悟空", 18);
        var person2 = new Person("猪八戒", 19);
        var person3 = new Person("沙和尚", 20);

        person1.sayName();
        person2.sayName();
        person3.sayName();

    </script>
```

但是，在全局作用域中定义函数却不是一个好的办法，为什么呢？因为，如果要是涉及到多人协作开发一个项目，别人也有可能叫sayName这个方法，这样在工程合并的时候就会导致一系列的问题，污染全局作用域，那该怎么办呢？有没有一种方法，我只在Person这个类的全局对象中添加一个函数，然后在类中引用？答案肯定是有的，这就需要原型对象了，我们先看看怎么做的，然后在详细讲解原型对象。

```
<script>
        // 使用构造函数来创建对象
        function Person(name, age) {
            // 设置对象的属性
            this.name = name;
            this.age = age;
        }

        // 在Person类的原型对象中添加方法
        Person.prototype.sayName = function() {
            console.log(this.name);
        };

        var person1 = new Person("孙悟空", 18);
        var person2 = new Person("猪八戒", 19);
        var person3 = new Person("沙和尚", 20);

        person1.sayName();
        person2.sayName();
        person3.sayName();

    </script>
```

- prototype

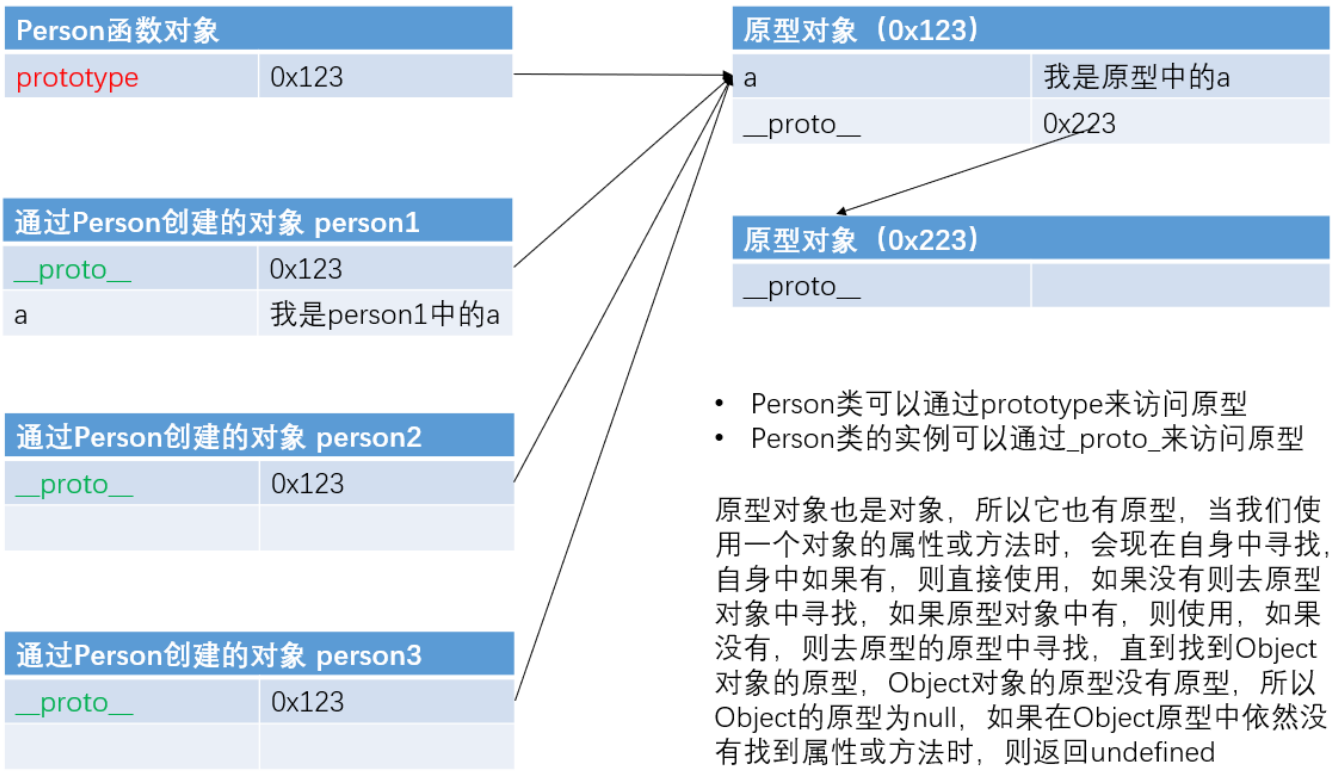
我们所创建的每一个函数，解析器都会向函数中添加一个属性prototype，这个属性对应着一个对象，这个对象就是我们所谓的原型对象，即显式原型，原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象，我们可以将对象中共有的内容，统一设置到原型对象中。

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__（隐式原型）来访问该属性。当我们访问对象的一个属性或方法时，它会先在对象自身中寻找，如果有则直接使用，如果没有则会去原型对象中寻找，如果找到则直接使用。

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。

2.12.4、原型链

访问一个对象的属性时，先在自身属性中查找，找到返回， 如果没有，再沿着__proto__这条链向上查找，找到返回，如果最终没找到，返回undefined，这就是原型链，又称隐式原型链，它的作用就是查找对象的属性(方法)。



> 注意：Object对象是所有对象的祖宗，Object的原型对象指向为null，也就是没有原型对象

2.12.5、toString方法

toString()函数用于将当前对象以字符串的形式返回。该方法属于Object对象，由于所有的对象都"继承"了Object的对象实例，因此几乎所有的实例对象都可以使用该方法，所有主流浏览器均支持该函数。

```
<script>
    // 使用构造函数来创建对象
    function Person(name, age) {
        // 设置对象的属性
        this.name = name;
        this.age = age;
    }

    //创建对象的一个实例对象
    var p = new Person("张三", 20);
    console.log(p.toString());

</script>
```

JavaScript的许多内置对象都重写了该函数，以实现更适合自身的功能需要。

类型	行为描述
String	返回 String 对象的值。
Number	返回 Number 的字符串表示。
Boolean	如果布尔值是true，则返回"true"。否则返回"false"。
Object (默认)	返回"[object ObjectName]"，其中 ObjectName 是对象类型的名称。
Array	将 Array 的每个元素转换为字符串，并将它们依次连接起来，两个元素之间用英文逗号作为分隔符进行拼接。
Date	返回日期的文本表示。
Error	返回一个包含相关错误信息的字符串。
Function	返回如下格式的字符串，其中 functionname 是一个函数的名称 此函数的 toString 方法被调用： “function functionname() { [native code] }”

2.12.6、hasOwnProperty方法

前边章节我们学过，如何遍历一个对象所有的属性和值，那我们要是判断当前对象是否包含指定的属性或方法可以使用 in 运算符来检查，如下代码演示：

```
<script>
    // 创造一个构造函数
    function MyClass() {
    }

    // 向MyClass的原型中添加一个name属性
    MyClass.prototype.name = "我是原型中的名字";

    // 创建一个MyClass的实例
    var mc = new MyClass();
    mc.age = 18;

    // 使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true
    console.log("age" in mc);
    console.log("name" in mc);

</script>
```

如果我只想要检查自身对象是否含有某个方法或属性，我们可以使用Object的hasOwnProperty()方法，它返回一个布尔值，判断对象是否包含特定的自身（非继承）属性。如下代码演示：

```
<script>
    // 创造一个构造函数
    function MyClass() {
    }

    // 向MyClass的原型中添加一个name属性
    MyClass.prototype.name = "我是原型中的名字";

    // 创建一个MyClass的实例
    var mc = new MyClass();
    mc.age = 18;

    // 使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true
    console.log("age" in mc);
    console.log("name" in mc);

    /*可以使用对象的hasOwnProperty()来检查对象自身中是否含有该属性，
    使用该方法只有当对象自身中含有属性时，才会返回true*/
    console.log(mc.hasOwnProperty("age"));
    console.log(mc.hasOwnProperty("name"));

</script>
```

hasOwnProperty这个方法就是原型中的，在执行方法的时候它会通过原型链进行查找，这个方法是Object的特有方法

```html
<script>
    // 创造一个构造函数
    function MyClass() {
    }

    // 向MyClass的原型中添加一个name属性
    MyClass.prototype.name = "我是原型中的名字";

    // 创建一个MyClass的实例
    var mc = new MyClass();
    mc.age = 18;

    // 检查当前对象
    console.log(mc.hasOwnProperty("hasOwnProperty"));
    // 检查当前对象的原型对象
    console.log(mc.__proto__.hasOwnProperty("hasOwnProperty"));
    // 检查当前对象的原型对象的原型对象
    console.log(mc.__proto__.__proto__.hasOwnProperty("hasOwnProperty"));

</script>
```

2.12.7、对象继承

前边我们一直在说继承，那什么是继承？它有什么作用？如何实现继承？将会是本章节探讨的问题。

面向对象的语言有一个标志，那就是它们都有类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。但是在JavaScript中没有类的概念，前边我们说所的类只是我们自己这么叫，大家要清楚。因此它的对象也与基于类的对象有所不同。实际上，JavaScript语言是通过一种叫做原型（prototype）的方式来实现面向对象编程的。

那实现继承有一个最大的好处就是子对象可以使用父对象的属性和方法，从而简化了一些代码。

JavaScript有六种非常经典的对象继承方式，但是我们只学习前三种：

- **原型链继承**
- **借用构造函数继承**
- **组合继承（重要）**
- 原型式继承
- 寄生式继承

- 寄生组合式继承

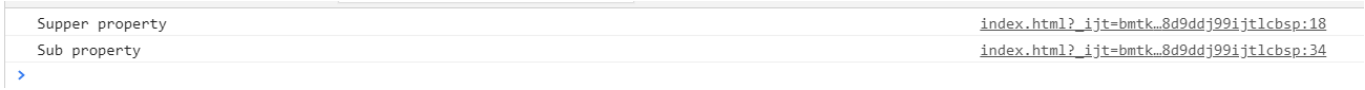
2.12.7.1、原型链继承

核心思想： 子类型的原型为父类型的一个实例对象

基本做法：

1.定义父类型构造函数 2.给父类型的原型添加方法 3.定义子类型的构造函数 4.创建父类型的对象赋值给子类型的原型 5.将子类型原型的构造属性设置为子类型 6.给子类型原型添加方法 7.创建子类型的对象: 可以调用父类型的方法 案例演示：

```
<script>
        // 定义父类型构造函数
        function SupperType() {
            this.supProp = 'Supper property';
        }

        // 给父类型的原型添加方法
        SupperType.prototype.showSupperProp = function () {
            console.log(this.supProp);
        };

        // 定义子类型的构造函数
        function SubType() {
            this.subProp = 'Sub property';
        }

        // 创建父类型的对象赋值给子类型的原型
        SubType.prototype = new SupperType();

        // 将子类型原型的构造属性设置为子类型
        SubType.prototype.constructor = SubType;

        // 给子类型原型添加方法
        SubType.prototype.showSubProp = function () {
            console.log(this.subProp)
        };

        // 创建子类型的对象: 可以调用父类型的方法
        var subType = new SubType();
        subType.showSupperProp();
        subType.showSubProp();

    </script>
```



缺点描述：

1.原型链继承多个实例的引用类型属性指向相同，一个实例修改了原型属性，另一个实例的原型属性也会被修改 2.不能传递参数 3.继承单一

2.12.7.2、借用构造函数继承

核心思想： 使用.call()和.apply()将父类构造函数引入子类函数，使用父类的构造函数来增强子类实例，等同于复制父类的实例给子类

基本做法：

1.定义父类型构造函数 2.定义子类型的构造函数 3.给子类型的原型添加方法 4.创建子类型的对象然后调用 案例演示：

借用构造函数继承的重点就在于SuperType*.*call(this, name)*，调用了SuperType构造函数，这样，SubType的每个实例都会将SuperType中的属性复制一份。

```
<script>// 定义父类型构造函数
        function SuperType(name) {
            this.name = name;
            this.showSupperName = function () {
                console.log(this.name);
            };
        }

        // 定义子类型的构造函数
        function SubType(name, age) {
            // 在子类型中调用call方法继承自SuperType
            SuperType.call(this, name);
            this.age = age;
        }

        // 给子类型的原型添加方法
        SubType.prototype.showSubName = function () {
            console.log(this.name);
        };

        // 创建子类型的对象然后调用
        var subType = new SubType("孙悟空", 20);
        subType.showSupperName();
        subType.showSubName();
        console.log(subType.name);
        console.log(subType.age);

        </script>
```

缺点描述:

只能继承父类的实例属性和方法，不能继承原型属性和方法 无法实现构造函数的复用，每个子类都有父类实例函数的副本，影响性能，代码会臃肿

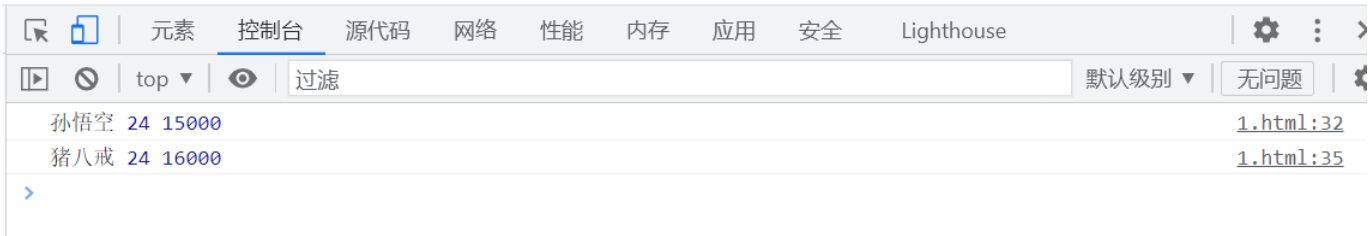
2.12.7.3、组合继承

核心思想： 原型链+借用构造函数的组合继承

基本做法:

1.利用原型链实现对父类型对象的方法继承 2.利用super()借用父类型构建函数初始化相同属性 案例演示：

```
<script>// 定义父类型构造函数
        function Person(name, age) {
            this.name = name;
            this.age = age;
        }

        Person.prototype.setName = function (name) {
            this.name = name;
        };

        function Student(name, age, price) {
            Person.call(this, name, age);
            // 为了得到父类型的实例属性和方法
            this.price = price;
            // 添加子类型私有的属性
        }

        Student.prototype = new Person();
         // 为了得到父类型的原型属性和方法
        Student.prototype.constructor = Student;
        // 修正constructor属性指向
        Student.prototype.setPrice = function (price) {
            // 添加子类型私有的方法
            this.price = price;
        };

        var s = new Student("孙悟空", 24, 15000);
        console.log(s.name, s.age, s.price);
        s.setName("猪八戒");
        s.setPrice(16000);
        console.log(s.name, s.age, s.price);

        </script>
```



缺点描述：

父类中的实例属性和方法既存在于子类的实例中，又存在于子类的原型中，不过仅是内存占用，因此，在使用子类创建实例对象时，其原型中会存在两份相同的属性和方法 。 注意：这个方法是

JavaScript中最常用的继承模式。

2.12.8、垃圾回收

垃圾回收（GC）：就像人生活的时间长了会产生垃圾一样，程序运行过程中也会产生垃圾，这些垃圾积攒过多以后，会导致程序运行的速度过慢，所以我们需要一个垃圾回收的机制，来处理程序运行过程中产生垃圾。

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。

```
<script>
        // 使用构造函数来创建对象
        function Person(name, age) {
            // 设置对象的属性
            this.name = name;
            this.age = age;
        }

        var person1 = new Person("孙悟空", 18);
        var person2 = new Person("猪八戒", 19);
        var person3 = new Person("沙和尚", 20);

        person1 = null;
        person2 = null;
        person3 = null;

    </script>
```

2.13、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：

- 全局作用域
- 函数作用域

2.13.1、声明提前

- 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），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 函数名(){} ，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

2.13.2、作用域

2.13.2.1、全局作用域

- 直接编写在script标签中的JavaScript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：
    - 创建的变量都会作为window对象的属性保存
    - 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

2.13.2.2、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，它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
- 在函数中要访问全局变量可以使用window对象
- 作用域链：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

2.13.3、作用域链

多个上下级关系的作用域形成的链，它的方向是从下向上的(从内到外)，查找变量时就是沿着作用域链来查找的。

查找一个变量的查找规则：

1.在当前作用域下的执行上下文中查找对应的属性，如果有直接返回，否则进入2 2.在上一级作用域的执行上下文中查找对应的属性，如果有直接返回，否则进入3 3.再次执行2的相同操作，直到全局作用域，如果还找不到就抛出找不到的ReferenceError异常