JavaScript简介

1、JavaScript发展历史

最初是由Netscape公司的程序员设计，最初命名为LiveScript，为了赶上Java的潮流，与开发出Java的Sun公司合作，共同推出JavaScript。

后来逐渐成为了全球常见浏览器必备的**脚本语言**，用途早就不局限于简单的数据验证，最终发展成为具备与浏览器窗口、内容等几乎所有的交互能力，今天的JavaScript已经成为一门功能全面的编程语言，能够处理复杂的计算和交互。

2、JavaScript的组成

虽然 JavaScript 和 ECMAScript 通常都被人们用来表达相同的含义，但 JavaScript 的含义却比 ECMA-262 中规定的要多得多。一个完整的 JavaScript 实现应该由下列三个不同的部分组成（如图1-1）

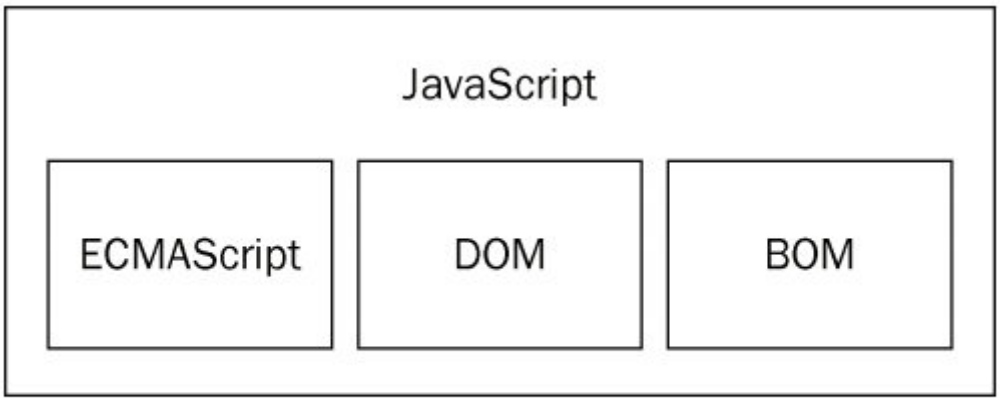


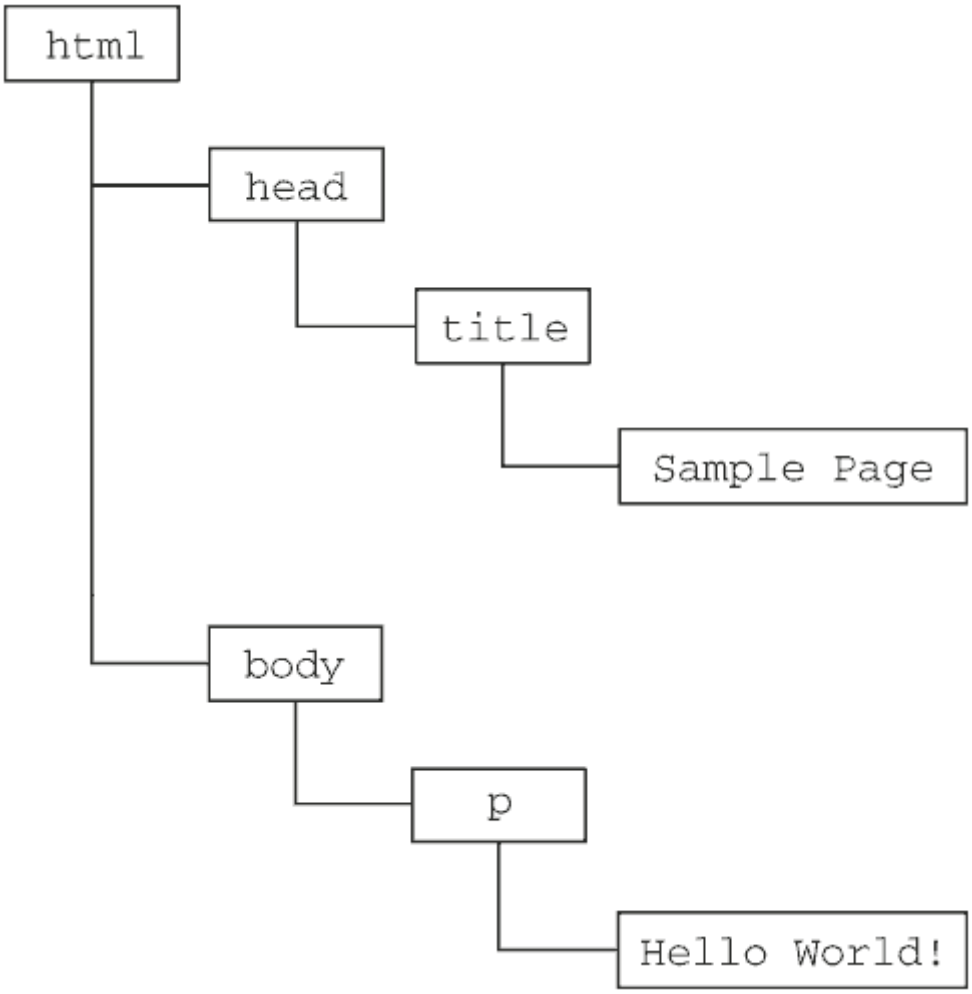
图 1-1

- 核心（ECMAScript）
  - ECMA：欧洲计算机制造商协会；
  - ES3：ECMAScript；
  - ES5：前一个版本；
  - ES6：最新的规范，大胆的使用，浏览器的支持已经非常完善；
  - ECMAScript中规定了语言的**语法、类型、语句、关键字、保留字、操作符、对象**

- 文档对象模型（DOM）
  - 文档对象模型（DOM， Document Object Model）是针对 XML 但经过扩展用于 HTML 的应用程序编程接口（API， Application Programming Interface）。 DOM 把整个页面映射为一个多层节点结构。 HTML或 XML 页面中的每个组成部分都是某种类型的节点，这些节点又包含着不同类型的数据。

```
<html>
    <head>
        <title>Sample Page</title>
    </head>

    <body>
        <p>Hello World!</p>
    </body>
</html>
```

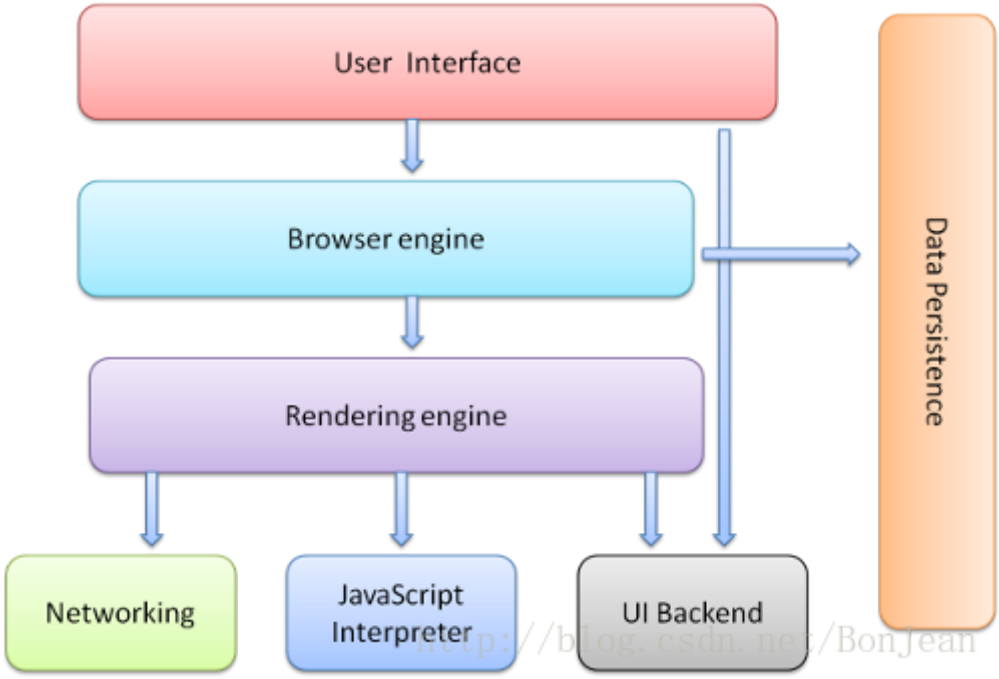


- 浏览器对象模型（BOM，Browser Object Model）
  - 可以控制浏览器显示的页面以外的部分，如弹出新浏览器窗口的功能。

3、JavaScript特点

- JavaScript是一种**脚本编程语言**：不用编译，只需要利用解释器就可以解释执行
- **面向对象**的语言：JavaScript本身也可以创建对象，以及调用对象的操作。因此，JavaScript的诸多功能可以来自于脚本环境中各种对象的调用。

- **简单性**：没有像其它需要编译的高级语言那样使用严格的数据类型。弱类型语言 使用**var**来声明变量。
- **安全性**：不允许访问本地的硬盘，并且不允许把数据存入到服务器上，还不允许对网络文档进行修改和删除，只允许通过浏览器实现信息浏览和动态交互，这样确保了对数据的安全化操作。
- **动态性**：可以直接对用户或客户的输入操作做出响应，而不必经过web服务器或web服务器程序。
- **跨平台性**：JavaScript同Java一样是与操作环境无关的，它只依赖于浏览器，只要客户的计算机浏览器支持JavaScript，它就可以被正确解释执行。从而实现一次编写，到处运行。



4、JavaScript使用

- **属性（行内）**：给标签设置对应的属性，属性值是要执行的JavaScript代码，比较少用

```
<a href="javaScript:alert('你已经领取过了')">领取奖品</a>

<input type="button" value="点击有惊喜" onclick="javascript:alert('哈哈哈')">
```

- **嵌入式**：使用script标签，标签需要闭合，标签内容是要执行的JavaScript代码

```
<script>
    //JavaScript 语言代码;
    console.log("哈哈哈");
</script>
```

  - 可以将JavaScript代码嵌入到head中或body中的任何地方。
  - 含在`<script>`元素内部的JavaScript代码将被从上至下依次解释。
  - 在使用`<script>`嵌入JavaScript代码时，记住不要在代码中的任何地方出现`</script>`字符串。因为按照解析嵌入式代码的规则，当浏览器遇到字符串`</script>`时，就会认为那是结束的`</script>`标签。通过字符串拼接可以解决这个问题。

- **引入**：使用script标签，标签需要闭合，设置属性src，src的值是js文件路径

- ```
  <script src="./js/my.js"></script>
  ```
- js代码直接写在一个独立的文件里面，该文件就是js文件，后缀是**.js** ，在文件中不需要在写`<script>`标签；
- 使用script 标签引入外部js文件时（标明src属性，不管有没有赋值），标签内容部分再填写js语句是不能执行的；
- 嵌入和引入的数量不受限制

5、JavaScript输出

JavaScript 可以通过不同的方式来输出数据：

- 使用 **window.alert()** 弹出警告框。
- 使用 **document.write()** 方法将内容写到 HTML 文档中。
- 使用 **innerHTML** 写入到 HTML 元素。
- 使用 **console.log()** 写入到浏览器的控制台。

（1）使用 window.alert()

弹出一个有确定按钮的信息框，多用于信息警告。可简写作 alert()。

```
<body>
    <h1>我的第一个页面</h1>
    我的第一个段落。
    <script>
        window.alert(5 + 6);
    </script>
</body>
```

（2）使用 document.write()

```
 <body>
    <h1>我的第一个 web 页面</h1>
    我的第一个段落。
    <script>
        document.write(Date());
    </script>
 </body>
```

注：将内容输出到HTML文档中，如果文档加载完成后执行，则会覆盖掉 所有原文档信息。

```
<script>
    window.onload = function () {
        document.write("你的文档被覆盖咯！");
    }
 </script>
```
```

（3）innerHTML

如需从 JavaScript 访问某个 HTML 元素，您可以使用 document.getElementById(id) 方法。
请使用 "id" 属性来标识 HTML 元素，并使用 innerHTML 来获取或插入元素内容。

```
<script>
    document.getElementById("demo").innerHTML = "段落已修改。";
</script>
```

（4）使用 confirm()

弹出一个可选择的警告框，有确定和取消按钮

```
<script>
    var result = confirm("确定执行该操作？");
    alert(result);
</script>
```

点击“确定”按钮返回true；点击“取消”返回 false。

（5）console.log()

- 使用 console.log() 打印日志信息到浏览器的控制台;

```
<h1>我的第一个 Web 页面</h1>
<p>
    浏览器中(Chrome, IE, Firefox) 使用 F12 来启用调试模式， 在调试窗口中点击 "Console" 菜单。
</p>
<script>
    a = 5;
    b = 6;
    c = a + b;
    console.log(c);
</script>
```

- 使用 console.error("打印'错误'信息到控制台");
- 使用 console.warn("打印'警告'到控制台");
- 使用 console.table({name:'华清远见' , age:14}) ；注：index部分不需要引号，value部分的字符串须加上引号;
- 清空浏览器控制台消息：console.clear()；需取消勾选浏览器控制台右上角"设置"项中的"Preserve log"。

（6）获取用户输入prompt

使用window.prompt()可以获取到用户输入

```
var shuru=window.prompt();
console.log(shuru);
```

6、JS获取和修改页面元素

（1）获取页面元素

- **document.getElementById("id")**：根据id找到指定的一个元素

```
document.getElementById("id").innerHTML = "新内容";
```

- **document.querySelector('选择器')**：根据选择器找到元素，如果该选择器对应多个元素时，则返回第一个元素

```
document.querySelector("选择器").innerHTML = "新内容"
```

（2）改变元素样式

- 语法是:document.querySelector('选择器').style.属性 = '值';

  – 属性是CSS样式中的属性，如display、color、width、height等；

  – 如果属性有横线(-)，如background-color、font-size、border-radius、font-weight，则把横线去掉，同时横线后面的第一个字母大写，
  如：backgroundColor、fontSize、borderRadius、fontWeight；

```
隐藏元素：document.querySelector('选择器').style.display = 'none';
改变字体颜色：document.querySelector('选择器').style.color = '#FF0000';
改变背景颜色：document.querySelector('选择器').style.backgroundColor = '#000000';
字体加粗：document.querySelector('选择器').style.fontWeight = 'bolder'
```

7、JavaScript事件

- 事件的概念

事件是指可以被JS监测到的网页行为；如：鼠标点击、鼠标移动、鼠标移入/离开、键盘按键、页面加载等；

- JavaScript事件的三要素：**事件源**、**事件**、**事件处理**

  结合现实事件——小王，把灯打开一下：

  –事件源：操作对象，名词，对应：开关;

  –事件：做什么动作，动词，对应：摁一下；

  –事件处理：背后要做哪些工作，具体要干什么，这里就是我们要写代码具体实现的功能了，对应：接通火线，把灯点亮；

- 我们学习JS就是找到“事件源”并给他绑定“事件”，在事件发生时启动“事件处理”程序；

```
<body>
    <script>
        function changeImage() {
            element=document.getElementById('myimage')
            if (element.src.match("bulbon"))
            {
```

```
                element.src="/images/pic_bulboff.gif";
            }
            else
            {
                element.src="/images/pic_bulbon.gif";
            }
        }
    </script>
    <img id="myimage" onclick="changeImage()" src="/images/pic_bulboff.gif"
width="100" height="180">
    <p>点击灯泡就可以打开或关闭这盏灯</p>
</body>
```

JavaScript语法

ECMAScript 的语法大量借鉴了 C 及其他类 C 语言（如 Java 和 Perl）的语法。因此，熟悉这些语言的开发人员在接受 ECMAScript 更加宽松的语法时，一定会有一种轻松自在的感觉。

1、语法规范

（1）区分大小写： JavaScript是严格区分大小写的

- JavaScript区分大小写，包括**关键字**、**变量**、**函数名**、**所有标识符**；
- querySelector的S是大写，你写成小写就会报错；
- alert()全部是小写，你写一个大写字母就会提示你该函数不存在；
- myname、myName、mynamE、MyName他们真不是同一个东西；

（2）空格： JavaScript会忽略标识符前后的空格

- 空格是为了让代码有整齐一致的缩进，形成统一的编码风格，让代码更具可读性；
- 你可以 document . querySelector('选择器')这样；
- 你还可以document. querySelector　('选择器')这样；
- 但是你不可以document.query　Selector('选择器')这样；
- 所以，你要搞清楚JavaScript是忽略标识符前后的空格；
- 在标识里面加空格，是把一个标识符分割成了两个或多个标识符；
- 一般加空格是为了代码排版，不要乱加空格

（3）注释： JavaScript支持两种注释方式

```
// 单行注释

/*
* 这是一个多行（块级）注释
*/
```

- 注释部分不会执行，合理的注释能显著提高代码的可读性；

- 可以通过浏览器源文件看到注释内容，所以什么该注释什么不该注释要注意；

（4）语句

ECMAScript 中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾，如下例所示：

```
var sum = a + b // 即使没有分号也是有效的语句——不推荐
var diff = a - b; // 有效的语句——推荐
```

虽然语句结尾的分号不是必需的，换行也可以表示一个语句结束，但我们建议任何时候都不要省略它。因为加上这个分号可以避免很多错误（例如不完整的输入），开发人员也可以放心地通过删除多余的空格来压缩 ECMAScript 代码（代码行结尾处没有分号会导致压缩错误）。另外，加上分号也会在某些情况下增进代码的性能，因为这样解析器就不必再花时间推测应该在哪里插入分号了。

（5）直接量（字面量）

JavaScript中直接使用的数据值叫做直接量。

```
12306;//数值

'我要学习JavaScript';//字符串类型

true;//布尔类型

null;//null类型

[1,2,3,4,5];//数组
```

（6）标识符

所谓标识符，就是指变量、函数、属性、参数的名字，或者用做某些循环语句中的跳转位置的标记。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：

- 由**字母、数字、下划线、美元符号（$）**组成
- 数字不能开头。
- 标识符采用**见名知意**原则
- 不能使用**关键字**、**保留字**、 **true**、 **false** 和 **null** 用作标识符。
- 变量、函数、属性、参数采用小驼峰法（第一个单词全小写，从第二个单词开始首字母大写）

```
//变量
var muNum = 123;
//属性
(new Object).myValue = 'test';
//函数及参数
function MyFunc(myParams){};
//跳转标记
breakOut:
for(var i = 0; i < 5; i++){
    if(i == 3){
        break breakOut;
    }
}
```

按照惯例， ECMAScript 标识符采用驼峰大小写格式，也就是第一个字母小写，剩下的每个单词的首字母大写，例如：

```
firstSecond
myCar
doSomethingImportant
```

2、关键字和保留字

ECMA-262 描述了一组具有特定用途的关键字，这些关键字可用于表示控制语句的开始或结束，或者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。以下就是 ECMAScript的全部关键字（带*号上标的是第 5 版新增的关键字）：

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

ECMA-262 还描述了另外一组不能用作标识符的保留字。尽管保留字在这门语言中还没有任何特定的用途，但它们有可能在将来被用作关键字。以下是 ECMA-262 第 3 版定义的全部保留字：

abstract	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

第 5 版把在非严格模式下运行时的保留字缩减为下列这些：

class	enum	extends	super
const	export	import	

在严格模式下，第 5 版还对以下保留字施加了限制：

implements	package	public
interface	private	static
let	protected	yield

3、变量

变量在JavaScript中就是用一个变量名表示。

- 变量名是大小写英文、数字、$和_的组合，且不能用数字开头。
- 变量名也不能是JavaScript的关键字，如if、while等。
- 变量名称对大小写敏感（y 和 Y 是不同的变量）
- 用var关键字申明一个或多个变量，比如：

```
var a; // 申明了变量a，此时a的值为undefined
var $b = 1; // 申明了变量$b，同时给$b赋值，此时$b的值为1
var s_007 = '007'; // s_007是一个字符串
var Answer = true; // Answer是一个布尔值true
var t = null; // t的值是null

var name = "小明";
var age = 18;
//打印多个内容
console.log("姓名：", name, ", 年龄：", age);
console.log(name, age);
```

变量名也可以用中文，但是，请不要给自己找麻烦。

在JavaScript中，使用等号=对变量进行赋值。可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，但是要注意只能用var申明一次，例如：

```
var a = 123; // a的值是整数123
a = 'ABC'; // a变为字符串
```

可以在一条语句中声明很多变量。该语句以 var 开头，并使用逗号分隔变量即可：

```
var lastname="Doe", age=30, job="carpenter";

//可以写成多行：可读性更强
var lastname="Doe",
    age=30,
    job="carpenter";
```

JavaScript数据类型

值类型(基本类型)：字符串（String）、数字(Number)、布尔(Boolean)、对空（Null）、未定义（Undefined）、Symbol。

引用数据类型：对象(Object)、数组(Array)、函数(Function)。

1、typeof 操作符

可以使用 typeof 操作符来检测变量的数据类型。

可能的字符串有："number"、"string"、"boolean"、"object"、"function" 和 "undefined"。

- **运算数为数字 typeof(x) = "number"**
- **字符串 typeof(x) = "string"**
- **布尔值 typeof(x) = "boolean"**
- **对象,数组和null typeof(x) = "object"**
- **函数 typeof(x) = "function"**
- **运算数未定义或undefined typeof(x) = "undefined"**

```
//数值
var num = 123;
console.log(typeof num); //number

//字符串
var str = "abc";
console.log(typeof str); //string

//布尔值
var bool = true;
console.log(typeof bool); //boolean

//对象
var obj = {};
console.log(typeof obj); //object
//数组
var arr = [1,2,3];
console.log(typeof arr); //object
// null值
var nu = null;
console.log(typeof nu); //object

//函数
var func = function show(){};
console.log(typeof func); //function

//变量未赋初值
var define;
```

```
console.log(typeof define); //undefined
console.log(typeof undefined); //undefined
```

2、Number类型

JavaScript 只有一种数字类型。数字可以带小数点，也可以不带。

```
var x1=34.00;      //使用小数点来写
var x2=34;         //不使用小数点来写
```

极大或极小的数字可以通过科学（指数）计数法来书写：

```
var y=123e5;      // 12300000
var z=123e-5;     // 0.00123
```

（1）整数类型

- **十进制**：最基本的数值字面量

```
var intNum = 55; // 整数
```

- **八进制**：以（0）开头，后跟0~7表示

```
var octalNum1 = 070; // 八进制的 56
var octalNum2 = 079; // 无效的八进制数值--解析为 79
var octalNum3 = 08; // 无效的八进制数值--解析为 8
```

- **十六进制**：以（0x）开头，后跟0~9及A~F表示

```
var hexNum2 = 0x1f; // 十六进制的 31
```

注意：在进行算术计算时，所有以八进制和十六进制表示的数值最终都将被转换成十进制数值。

- **进制转换**：默认情况下，JavaScript 数字为十进制显示。但可以使用 **toString()** 方法输出16进制、8进制、2进制。

```
var myNumber=128;
myNumber.toString(16);   // 返回 80
myNumber.toString(8);    // 返回 200
myNumber.toString(2);    // 返回 10000000
```

（2）浮点类型

所谓浮点数值，就是该数值中必须包含一个小数点，并且小数点后面必须至少有一位数字。虽然小数点前面可以没有整数，但我们不推荐这种写法。

```
var floatNum1 = 1.1;
var floatNum2 = 0.1;
var floatNum3 = .1; // 有效，但不推荐
```

对于那些极大或极小的数值，可以用 e 表示法（即科学计数法）表示的浮点数值表示。用 e 表示法表示的数值等于 e 前面的数值乘以 10 的指数次幂。

```
var floatNum = 3.125e7; // 等于 31250000
```

（3）精度

JS只能用浮点数表示其中有限的实数， 因为JS采用的是二进制表示法，如：1/2,1/4,1/8, 1/256， 我们常用的是十进制，如1/10,1/100,1/10000，

那么如何精准的表示0.1这种非常简单的数字那？答案是并不能；

在JS中使用实数的时候，通常只是真实值的一个近似表示；

如何解决：可以先全部转成整数，运算完后再转回；

```
var a=0.2, b= 0.3;
console.log((b*1000000-a*1000000)/1000000);
```

（4）NaN

- **NaN**，即非数值（Not a Number）是一个特殊的数值，这个数值用于表示一个本来要返回数值的操作数未返回数值的情况（这样就不会抛出错误了）。例如，在其他编程语言中，任何数值除以 0 都会导致错误，从而停止代码执行。但在 ECMAScript 中，任何数值除以 0 会返回 NaN，因此不会影响其他代码的执行。

```
console.log(typeof 1/0); //NaN

var a = "123abc";
console.log(typeof a*1); //NaN
```

- **isNaN()** 函数用于检查其参数是否是非数字值。如果参数值为 NaN 或字符串、对象、undefined 等非数字值则返回 true, 否则返回 false。

```
console.log(isNaN(123)); //false
console.log(isNaN(-1.23)); //false
console.log(isNaN(5 - 2)); //false
console.log(isNaN(0)); //false
console.log(isNaN("Hello")); //true
console.log(isNaN("2005/12/12")); //true
```

（5）字符串转为数字

- **转换函数**：
  - **parseInt()**：将值转为整数，只保留整数部分

```
parseInt("1234blue"); //returns 1234
parseInt("0xA"); //returns 10
parseInt("22.5"); //returns 22
parseInt("blue"); //returns NaN
```

parseInt()方法还有**基模式**，可以把二进制、八进制、十六进制或其他任何进制的字符串转换成整数。基是由parseInt()方法的第二个参数指定的

```
parseInt("AF", 16); //returns 175
parseInt("10", 2); //returns 2
parseInt("10", 8); //returns 8
parseInt("10", 10); //returns 10
```

如果十进制数包含前导0，那么最好采用基数10，这样才不会意外地得到八进制的值

```
parseInt("010", 8); //returns 8
parseInt("010", 10); //returns 10
```

- **parseFloat()**：将值转为小数，保留小数部分

```
parseFloat("1234blue"); //returns 1234.0
parseFloat("0xA"); //returns 0
parseFloat("22.5"); //returns 22.5
parseFloat("22.34.5"); //returns 22.34
parseFloat("0908"); //returns 908
parseFloat("blue"); //returns NaN
```

注意：只有对String类型的值，这两个函数才能正确运行；对其他类型返回的都是NaN(Not a Number)。

- **强制类型转换**
  - **Number(value)**：把给定的值转换成数字（可以是整数或浮点数）

```
Number(false) //0
Number(true) //1
Number(undefined) //NaN
Number(null) //0
Number( "5.5 ") //5.5
Number( "56 ") //56
Number( "5.6.7 ") //NaN
Number(new Object()) //NaN
Number(100) //100
```

- **算数运算**：
  - ***　/　-**：待运算的值必须是数值型的字符串

```
var str = '012.345 ';
var x = str - 0;
console.log(x); //12.345

var y = str/1;
console.log(y); //12.345

var z = str * 1;
console.log(z); //12.345

var a = '123abc';
console.log(a*1);//NaN
```

- +：当加号两边有字符串时，加号当成连接符使用，加号两边都是数字时才做加法运算

```
var a = "18";
console.log(a + 0); //180  字符串连接符
console.log(a + 0 + 100 + 2); //1801002 字符串连接符
console.log(a * 1 + 0 + 100 + 2); //120  算数运算

//变量可以转换
var y = "5";       // y 是一个字符串
var x = +y;        // x 是一个数字

//变量不能转换，它仍然会是一个数字，但值为 NaN
var y = "John";    // y 是一个字符串
var x = +y;        // x 是一个数字 (NaN(值不是一个数字))
```

- **区别**
  - **Number()**转换的是字符串的**整个值**，parseInt()和parseFloat()转换时是**遇到非数字的字符时停止**。
  - 对于以‘0’、‘0x’开头的字符串parseInt()可以转换为相应的八进制和十六进制整形数字,parseFloat()不可以。
  - Number可以将“0x”开头的字符串转为对应的十六进制整形数，不能转换0开头的

3、String类型

JavaScript的字符串类型用于表示文本数据。它是一组16位的无符号整数值的“元素”。在字符串中的每个元素占据了字符串的位置。第一个元素的索引为0，下一个是索引1，依此类推。字符串的长度是它的元素的数量。

符串可以是引号中的任意文本。您可以使用单引号或双引号。

（1）转义字符

String 数据类型包含一些特殊的字符字面量，也叫转义序列，用于表示非打印字符，或者具有其他用途的字符。这些字符字面量如下表所示：

字面量	含义
\n	换行
\t	制表
\b	空格
\r	回车
\f	进纸
\	斜杠
'	单引号（'），在用单引号表示的字符串中使用。例如： 'He said, 'hey.''
"	双引号（"），在用双引号表示的字符串中使用。例如： "He said, "hey.""
\xnn	以十六进制代码nn表示的一个字符（其中n为0～F）。例如， \x41表示"A"（ASCII码为65）
\unnnn	以十六进制代码nnnn表示的一个Unicode字符（其中n为0～F）。例如， \u03a3表示希腊字符Σ

这些字符字面量可以出现在字符串中的任意位置，而且也将被作为一个字符来解析。

（2）字符串的特点

ECMAScript 中的字符串是不可变的，也就是说，字符串一旦创建，它们的值就不能改变。要改变某个变量保存的字符串，首先要销毁原来的字符串，然后再用另一个包含新值的字符串填充该变量。

```
var lang = "Java";
lang = lang + "Script";
//加号:+，只要有一个是字符串，则全部作为字符串
```

（3）转换为字符串

要把一个值转换为一个字符串有3种方式。

- 使用 **toString()**方法

```
var age = 11;
var ageAsString = age.toString(); // 字符串"11"
var found = true;
var foundAsString = found.toString(); // 字符串"true"
```

数值、布尔值、对象和字符串值变量都有 toString()方法。但 null 和 undefined 值没有这个方法。

- 使用**String()**构造器

```
var num2 = 50;
var str1 = String(num2);
console.log(typeof str1); //返回string
```

- 使用**字符串连接符+**

```
var num3 = 60;
var str1s = num3 +'';
console.log(typeof str1s); //返回string
```

（4）多行字符串

由于多行字符串用\n写起来比较费事，所以最新的ES6标准新增了一种多行字符串的表示方法，用反引号 `...` 表示：

```
console.log(`多行
    字符串
    测试`
);
```

（5）模板字符串

要把多个字符串连接起来，可以用+号连接

```
var name = '小明'; var age = 20;
var message = '你好, ' + name + ', 你今年' + age + '岁了!';
alert(message);
```

如果有很多变量需要连接，用+号就比较麻烦。ES6新增了一种模板字符串，表示方法和上面的多行字符串一样，但是它会自动替换字符串中的变量：

```
var name = '小明'; var age = 20;
var message = `你好, ${name}, 你今年${age}岁了!`;
alert(message);
```

（6）字符串操作

```
var str = "abc";
console.log(str.length); //3
console.log(str[0]); //a
```

字符串与数组的相互转换：split、join

字符串和json的转换：JSON.parse、JSON.stringify

4、Boolean类型

（1）取值

布尔（逻辑）只能有两个值：true 或 false。

```
var x = true;
var y = false;
```

布尔常用在条件测试中。

```
var flag = true;
if(flag) {
    //相关代码
}
```

（2）其他类型转为Boolean

能够转为false的值有：**0**、**0.0**、**-0**、**“**、**null**、**undefined**、**false**

```
var a = 100;
console.log(Boolean(a)); //true

// 数字0转成boolean时是false
var b = 0;
console.log(Boolean(b)); //false
var c = 0.0;
console.log(Boolean(c)); //false
var d = -0;
console.log(Boolean(d)); //false

// 字符串转boolean
var str1 = '你好';
console.log(Boolean(str1)); //true
var str2 = ''; //空字符
console.log(Boolean(str2)); //false
var str3 = ' '; //字符 空格
console.log(Boolean(str3)); //true

// null和undefined
console.log(Boolean(null)); //false
console.log(Boolean(undefined)); //false

// [] 空数组  {} 空对象
console.log(Boolean([])); //true
console.log(Boolean({})); //true

// false
console.log(Boolean(false)); //false
```

5、Undefined类型

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，例如：

```
var message;
console.log(message); //undefined
```

6、Null类型

Null类型是只有一个值的数据类型，这个特殊的值是 null。从逻辑角度来看， null 值表示一个空对象指针，而这也正是使用 typeof 操作符检测 null 值时会返回"object"的原因，如下面的例子所示：

```
var car = null;
console.log(car); //null
```

如果定义的变量准备在将来用于保存对象，那么最好将该变量初始化为 null 而不是其他值。这样一来，只要直接检查 null 值就可以知道相应的变量是否已经保存了一个对象的引用，如下面的例子所示：

```
if (car != null){
 // 对 car 对象执行某些操作
}
```

注意：null和undefined区别

- **null**
  - 是保留字，常用来描述空值;
  - typeof null：返回的是object类型，也就是说可以把null看成一个特殊的对象；

- **undefined**
  - undefined表明变量没有初始化；
  - 如果函数没有返回值，则返回undefined；
  - typeof undefined：返回的是字符串undefined；
  - ==比较的是值相等，认为NULL和undefined是相等的；
  - ===比较的是地址相等，返回false；

```
console.log(null == undefined); //true
console.log(null === undefined); //false
```

7、Array类型

多个数据的集合，更多的时候放的是同类型的数据，每个数据称为是元素

（1）数组创建

- 动态创建：数组声明及赋值分开操作

```
var myCars = new Array();
myCars[0] = "Saab";
myCars[1] = "Volvo";
myCars[2] = "BMW";
```

- 静态创建：数组声明及赋值一个语句完成

```
//方式1：使用new关键字
var myCars = new Array("Saab","Volvo","BMW");
//方式2：直接使用[]
var myCars = ["Saab","Volvo","BMW"];
```

（2）数组访问

数组中元素通过下标来进行管理，下标从0开始。最后一个元素的下标为“数组长度-1”

数组的长度可以通过length获取。

```
var arr = ["张三", "李四", "王五"];
console.log(arr.length); //3
console.log(arr[1]); //李四
```

（3）数组元素类型

数组中元素可以是任意类型

```
var arr = [123, "abc", true, null, undefined, {}, function show(){}];
console.log(arr);
```

（4）二维数组、多维数组、稀疏数组

数组元素的类型为数组

```
//二维数组
var two = [[1, 2, 3], ["a", "b"]];

//多维数组
var more = [[1,2],[3,4],[5,[1,[3,[5,6]]]]];

//稀疏数组:最后一个空元素会被抛弃,只抛弃最后一个
var less = [1,2,,4,,,,7,,,,5,6,,];
```

（5）数组转换

- 数组转为字符串： **join()**

```
var arr = [1,2,3];
console.log(arr.join(" ")); //1 2 3
```

- 字符串转为数组： **split()**

```
var str = "abcd";
console.log(str.split("")); //["a", "b", "c", "d"]
```

8、Object类型

在计算机科学中，对象是指内存中的可以被 ***标识符*** 引用的一块区域。

在 Javascript 里，对象可以被看作是一组属性的集合。用**对象字面量**语法来定义一个对象时，会自动初始化一组属性。（也就是说，你定义一个var a = "Hello"，那么a本身就会有a.substring这个方法，以及a.length这个属性，以及其它；如果你定义了一个对象，var a = {}，那么a就会自动有a.hasOwnProperty及a.constructor等属性和方法。）

属性的值可以是任意类型，包括具有复杂数据结构的对象。属性使用键来标识，它的键值可以是一个字符串或者符号值（Symbol）。

对象使用**花括号{}**括起来，里面是**键值对**，键（属性）和值使用冒号分开，多个键值对之间使用逗号分开；

```
var obj = {
    age: 20,
    username: '小明'
};
console.log(obj);

//通过点“对象.键名”访问来访问键对应的值，直接写键名，不需要使用引号
console.log(obj.username);
//通过“对象['键名']”访问对应的值，键名必须使用引号
console.log(obj['age']);
```

- 更多时候是将数组与对象结合起来使用：JSON
- 判断两个数组是否相等时，不是根据值，是根据引用的地址来判断

```
var arr1 = [1,2,3];
var arr2 = [1,2,3];
console.log(arr1 == arr2); //false

var arr3 = arr1;
console.log(arr1 == arr3); //true
```

JavaScript函数

函数是由事件驱动的或者当它被调用时执行的可重复使用的代码块。

1、函数声明

函数就是包裹在花括号中的代码块，前面使用了关键词 function

```
//定义函数
function functionname() {
    // 执行代码
}

//调用函数
functionname();
```

当调用该函数时，会执行函数内的代码。

可以在某事件发生时直接调用函数（比如当用户点击按钮时），并且可由 JavaScript 在任何位置进行调用。

2、函数的参数

- 在调用函数时，您可以向其传递值，这些值被称为参数；
- 参数可以在函数中使用；
- 可以传递任意多的参数，由逗号 (,) 分隔；
- 声明函数时，把参数作为变量来声明。

```
function myFunction(var1,var2) {
    //执行代码，可以使用参数var1和var2
}
```

- 声明函数时的变量称为形参，调用函数时传递的参数称为实参。
- 形参和实参是一一对应的，如果实参个数大于形参，多余的实参不使用。

```
function show(name, age) {
    console.log("名字是${name}, 今年${age}岁了");
}
show("小明",  18);


function add(a, b, c) {
    var sum = a + b + c;
    console.log(sum);
}
add(1,2,3,4,5); //6   后面多余的参数不参与计算
```

3、函数的返回值

有时，我们会希望函数将值返回调用它的地方。

- 通过使用 **return** 语句就可以实现。
- 在使用 return 语句时，函数会停止执行，并返回指定的值。

```
function myFunction() {
    var x=5;
    return x;
}

console.log(myFunction()); //5
```

- 如果仅仅是退出函数执行，并且不需要返回数据，可以直接使用return语句。在return语句之后不能再写代码。

```
function show() {
    console.log("return语句之前的代码");
    return;
    //不可达代码，写出来会报错
    //console.log("return语句之后的代码，不会被执行");
}
```

- 函数的分类
    - 没有参数没有返回值
    - 有参数没有返回值
    - 没有参数有返回值
    - 有参数有返回值

JavaScript作用域

1、删除变量

Javascript声明变量的时候，虽然用var关键字声明和不用关键字声明，很多时候运行并没有问题，但是这两种方式还是有区别的。可以正常运行的代码并不代表是合适的代码。

使用**delete**关键字可以删除没用var声明的变量。使用var声明的变量不能删除。

```
var a = 100;
console.log(delete a); //false

b = "abc";
console.log(delete b); //true
console.log(b); //输出错误   b is not defined
```

注意：实际开发中不使用delete删除数组元素。

2、this和window

单独使用 this，它指向全局(Global)对象window。

```
var a = 100;
console.log(window.a); //100
console.log(this.a); //100

this.b = 200;
console.log(delete b);
//console.log(b); //报错 b is not defined

console.log(this === window); //true
```

3、变量作用域

作用域就是变量与函数的可访问范围，即作用域控制着变量与函数的可见性和生命周期。在JavaScript中，变量的作用域有**全局作用域**和**局部作用域**两种。

（1）局部作用域

局部作用域一般只在固定的代码片段内可访问到，最常见的是**函数内部**，所以也有人称之为函数作用域。

```
function method() {
    var a = 100;
}
console.log(a); //报错  a is not defined
```

由于局部变量只在定义它的函数内部有效，那么在不同的函数中就可以使用相同的变量名称

（2）全局作用域

在代码中任何地方都能访问到的对象拥有全局作用域，一般来说以下几种情形拥有全局作用域：

- 在函数外部定义的变量

```
var a = 100;
function aMethod() {
    console.log(a); //100
}
function bMethod() {
    console.log(a); //100
}
console.log(a); //100
```

- 所有未定义直接赋值的变量自动声明为拥有全局作用域：没有使用var声明的变量

```
a = 100;
function aMethod() {
    console.log(a); //100
    b = 200;
}
console.log(a); //100
console.log(b); //200
```

- 所有window对象的属性拥有全局作用域

```
window.name = "abc";
window.a = 100;
console.log(window.name, window.a); //abc 100
```

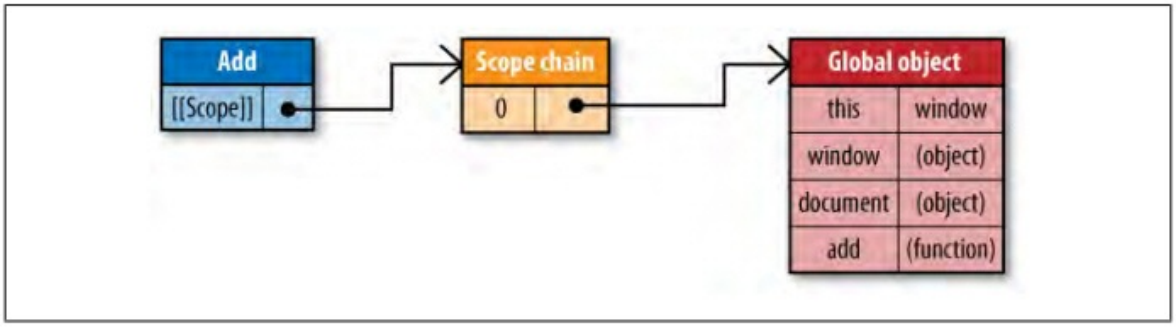
（3）作用域链

在JavaScript中，函数也是对象，实际上，JavaScript里一切都是对象。函数对象和其它对象一样，拥有可以通过代码访问的属性和一系列仅供JavaScript引擎访问的内部属性。其中一个内部属性是[[Scope]]，由ECMA-262标准第三版定义，该内部属性包含了函数被创建的作用域中对象的集合，这个集合被称为函数的作用域链，它决定了哪些数据能被函数访问。

当一个函数创建后，它的作用域链会被创建此函数的作用域中可访问的数据对象填充。例如定义下面这样一个函数：

```
function add(num1,num2) {
    var sum = num1 + num2;
    return sum;
}
```

在函数add创建时，它的作用域链中会填入一个全局对象，该全局对象包含了所有全局变量，如下图所示（注意：图片只例举了全部变量中的一部分）：

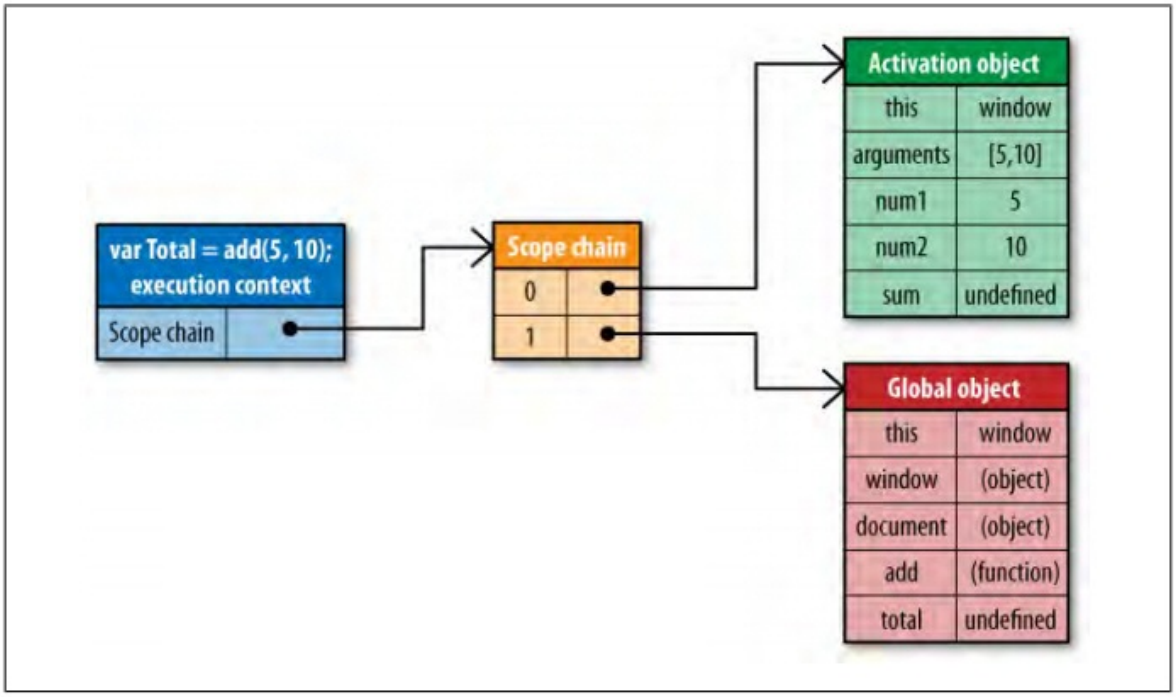


函数add的作用域将会在执行时用到。例如执行如下代码：

```
var total = add(5,10);
```

执行此函数时会创建一个称为“运行期上下文(execution context)”的内部对象，运行期上下文定义了函数执行时的环境。每个运行期上下文都有自己的作用域链，用于标识符解析，当运行期上下文被创建时，而它的作用域链初始化为当前运行函数的[[Scope]]所包含的对象。

这些值按照它们出现在函数中的顺序被复制到运行期上下文的作用域链中。它们共同组成了一个新的对象，叫“活动对象(activation object)”，该对象包含了函数的所有局部变量、命名参数、参数集合以及this，然后此对象会被推入作用域链的前端，当运行期上下文被销毁，活动对象也随之销毁。新的作用域链如下图所示:



在函数执行过程中，每遇到一个变量，都会经历一次标识符解析过程，以决定从哪里获取和存储数据。该过程从作用域链头部，也就是从活动对象开始搜索，查找同名的标识符，如果找到了就使用这个标识符对应的变量，如果没找到继续搜索作用域链中的下一个对象，如果搜索完所有对象都未找到，则认为该标识符未定义。函数执行过程中，每个标识符都要经历这样的搜索过程。

（4）作用域链代码优化

从作用域链的结构可以看出，在运行期上下文的作用域链中，标识符所在的位置越深，读写速度就会越慢。如上图所示，因为全局变量总是存在于运行期上下文作用域链的最末端，因此在标识符解析的时候，查找全局变量是最慢的。所以，在编写代码的时候应尽量少使用全局变量，尽可能使用局部变量。一个好的经验法则是：如果一个跨作用域的对象被引用了一次以上，则先把它存储到局部变量里再使用。例如下面的代码：

```
function changeColor(){
    document.querySelector("button").onclick=function(){
        document.querySelector("div").style.backgroundColor="red";
    };
}
```

这个函数引用了两次全局变量document，查找该变量必须遍历整个作用域链，直到最后在全局对象中才能找到。这段代码可以重写如下：

```
function changeColor(){
    var doc=document;
    doc.querySelector("button").onclick=function(){
        doc.querySelector("div").style.backgroundColor="red";
    };
}
```

这段代码比较简单，重写后不会显示出巨大的性能提升，但是如果程序中有大量的全局变量被从反复访问，那么重写后的代码性能会有显著改善。

（5）改变作用域链

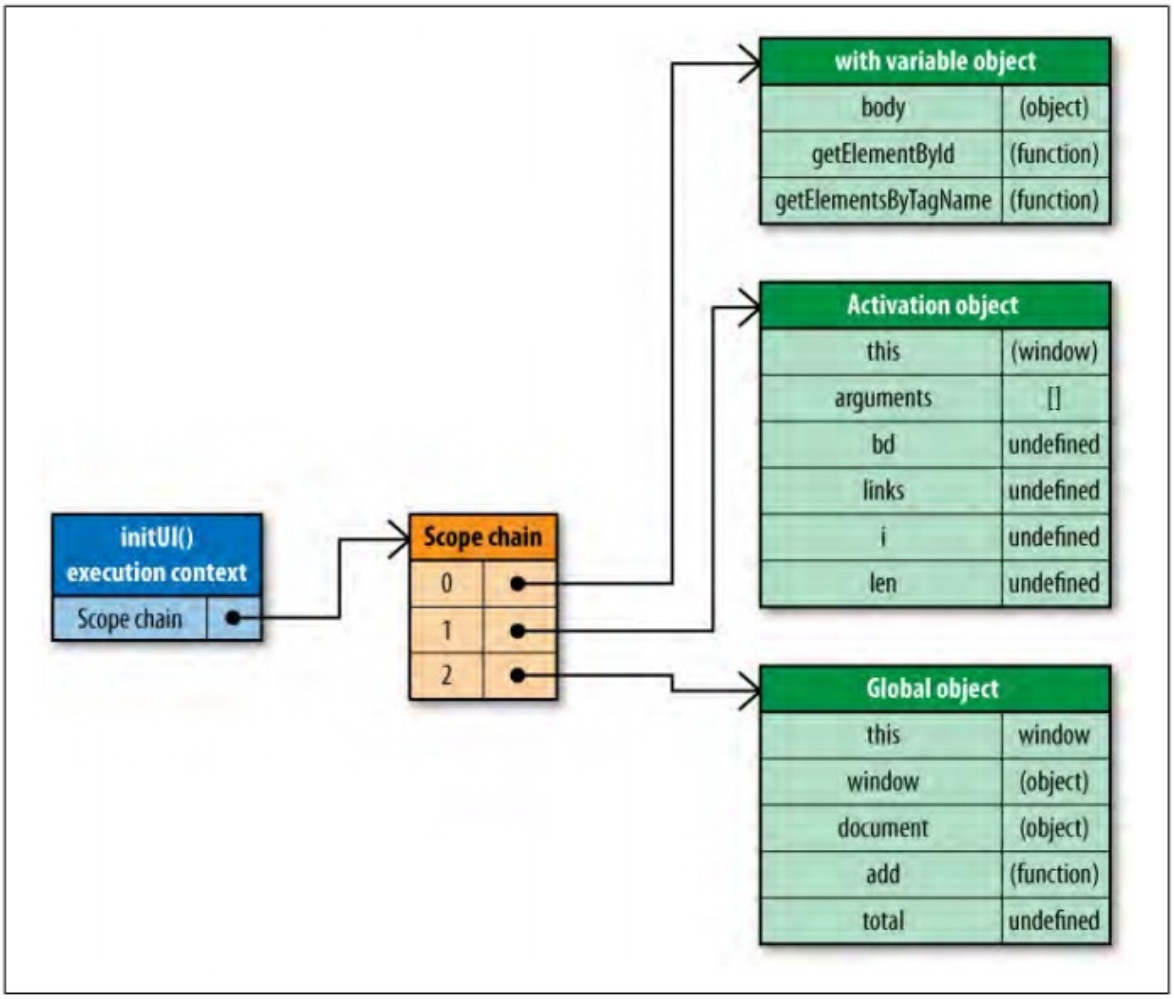
函数每次执行时对应的运行期上下文都是独一无二的，所以多次调用同一个函数就会导致创建多个运行期上下文，当函数执行完毕，执行上下文会被销毁。每一个运行期上下文都和一个作用域链关联。一般情况下，在运行期上下文运行的过程中，其作用域链只会被 with 语句和 catch 语句影响。

with语句是对象的快捷应用方式，用来避免书写重复代码。例如：

```
function initUI(){
    with(document){
        var bd=body,
            links=getElementsByTagName("a"),
            i=0,
            len=links.length;
        while(i < len){
            update(links[i++]);
        }
        getElementById("btnInit").onclick=function(){
            doSomething();
        };
    }
}
```

这里使用width语句来避免多次书写document，看上去更高效，实际上产生了性能问题。

当代码运行到with语句时，运行期上下文的作用域链临时被改变了。一个新的可变对象被创建，它包含了参数指定的对象的所有属性。这个对象将被推入作用域链的头部，这意味着函数的所有局部变量现在处于第二个作用域链对象中，因此访问代价更高了。如下图所示：



因此在程序中应避免使用with语句，在这个例子中，只要简单的把document存储在一个局部变量中就可以提升性能。

另外一个会改变作用域链的是try-catch语句中的catch语句。当try代码块中发生错误时，执行过程会跳转到catch语句，然后把异常对象推入一个可变对象并置于作用域的头部。在catch代码块内部，函数的所有局部变量将会被放在第二个作用域链对象中。示例代码：

```
try{
    doSomething();
}catch(ex){
    alert(ex.message); //作用域链在此处改变
}
```

请注意，一旦catch语句执行完毕，作用域链就会返回到之前的状态。try-catch语句在代码调试和异常处理中非常有用，因此不建议完全避免。你可以通过优化代码来减少catch语句对性能的影响。一个很好的模式是将错误委托给一个函数处理，例如：

```
try{
    doSomething();
}catch(ex){
    handleError(ex); //委托给处理器方法
}
```

优化后的代码，handleError方法是catch子句中唯一执行的代码。该函数接收异常对象作为参数，这样你可以更加灵活和统一的处理错误。由于只执行一条语句，且没有局部变量的访问，作用域链的临时改变就不会影响代码性能了。

（6）变量生命周期

JavaScript 变量生命周期在它声明时初始化。

- 局部变量在函数执行完毕后销毁。
- 全局变量在页面关闭后销毁。

4、变量提升

JavaScript 中，函数及变量的声明都将被提升到函数的最顶部。变量可以在使用后声明，也就是**变量可以先使用再声明。**

- **变量提升**：函数声明和变量声明总是会被解释器悄悄地被"提升"到方法体的最顶部。

```
x = 5; //为变量x赋值
console.log(x); //5
var x; //声明变量x
```

- **初始化的变量不会提升**

```
var x = 1; //初始化x
var y; //声明y
console.log(x + "---" + y); // 1 --- undefined
y = 7; //为变量y初始化
```

运算符

1、算数运算符

（1）+、-

- 加减法：数学运算

```
var a = 10, b= 20;
console.log(a + b); //30
console.log(a - b); //-10
```

- 正负号

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

- 连接符+：当加号两边任意一边为字符串时，+被当成连接符使用。当有多个+使用时，按照从左到右的顺序进行计算。

- 字符串和数字相加，数字转成字符串
- 数字和布尔值相加，布尔值 false 转成 0，true 转成 1
- 字符串与布尔值相加，布尔值转化成字符串。
- 数字与 null(空值) 相加，null 转化为数字 0
- 字符串与 null(空值) 相加，null 转化为字符串
- 数字和布尔值与undefined相加，将会进行加法计算，true转为1，false转为0，结果都为NaN

```
console.log(10 + 20); //30
console.log(10 + 20 + ""); //"30"
console.log("" +10 +20); //"1020"

//字符串和数字相加，数字转成字符串
var one="This is a test";
var two=123;
console.log(one + two); //"This is a test123"

//数字和布尔值相加，布尔值 false 转成 0，true 转成 1
var one=13;
var two=true;
console.log(one+two); //14

//字符串与布尔值相加，布尔值转化成字符串。
var one = "abc";
var two = true;
console.log(one +two); //abctrue

//数字与 null(空值) 相加，null 转化为数字 0
var x = 10;
console.log(x + null); //10

//字符串与 null(空值) 相加，null 转化为字符串
var y = "abc";
console.log(y + null); //abcnull

console.log(10 + undefined); //NaN
console.log(true + undefined); //NaN  true转为1后进行加法运算
console.log("abc" + undefined); //abcundefined
```

(2)*、/、%

- *****：乘法运算，还可以用于将字符串转为数值
- **/**：取除法运算之后的商。结果整数相除不能整除，得到的结果为浮点数。
- **%**：取整除之后的余数。整数取余为整数，浮点数取余认为浮点数。

```
var a = 10, b = 3;
console.log(a * b); //30
console.log(a / b); //3.3333333333333335
console.log(a % b); //1

var x = 3.8, y = 1.2;
console.log(x * y); //4.56
console.log(x / y); //3.1666666666666665
console.log(x % y); //0.19999999999999996
```

注意：取模运算的结果符号只与左边值的符号有关

```
var x = 7 % 3;
var y = 7 % (-3);
var z = (-7) % 3;
console.log(x, y, z); // 1  1  -1
```

（3）++、-- 单目运算符

- 写在变量之前：表示先对变量进行加1或减1操作，然后再使用改变之后的变量值
- 写在变量之后： 表示先使用变量的值，再进行加1或减1操作

```
var i = 0;
console.log(i++); //0
console.log(++i); //2
```

小练习：

```
var a = 2;
console.log(--a/2+(++a*2));
console.log(++a*2+--a/2);
```

2、赋值运算符

赋值运算符用于给 JavaScript 变量赋值，将等号右边的值赋给等号左边的变量

（1）=

（2）+=、-=、*=、/=、%=

x+=y 等同于x=x+y

```
var a = 10;
console.log(a += 2); //12
```

3、比较运算符

（1）>、 >=、 <、 <=

（2）\=\=、\=\=\=、 !\=、 !\=\=\=

- 对于 string、number 等基础类型，== 和 === 是有区别的
  - 不同类型间比较，== 之比较 "转化成同一类型后的值" 看 "值" 是否相等，=== 如果类型不同，其结果就是不等。
  - 同类型比较，直接进行 "值" 比较，两者结果一样。
- 对于 Array,Object 等高级类型，== 和 === 是没有区别的， 都是进行 "指针地址" 比较。

- 基础类型与高级类型，== 和 === 是有区别的
  - 对于 ==，将高级转化为基础类型，进行 "值" 比较
  - 因为类型不同，=== 结果为 false

```
var a = 123;
var b = "123";
console.log(a == b); //true
console.log(a === b); //false

var arr1 = [1,2,3];
var arr2 = [1,2,3];
console.log(arr1 == arr2); //false
console.log(arr1 === arr2); //false

console.log(a == arr1); //false
console.log(a == arr1); //false

var x = 1;
var y = [1];
console.log(x == y); //true
console.log(x === y); //false
```

4、逻辑运算符

逻辑运算符用于测定变量或值之间的逻辑。

（1）&&：表示and

JavaScript 中逻辑与和其他语言不太一样，如果第一个操作数是 true(或者能够转为 true)，计算结果就是第二个操作数，如果第一个操作数是 false，结果就是 false（短路计算），对于一些特殊数值不遵循以上规则。

可以理解为：如果运算的第一个操作数为true,则返回第二个操作数,反之则返回第一个操作数

undefined、null、NaN、空字符、0在进行短路与后认为其值。

```
var a = [1,2,3];
var b = "hello";
var obj = new Object();
var d;

console.log(true && 10);            //第一个操作数是true，结果是第二个操作，也就是10
console.log(false && b);            //第一个操作数是false，结果flase
console.log(100 && false);          //第一个操作数是100，结果flase
console.log(undefined && false);    //第一个操作数是undefined，结果undefined
console.log(NaN && false);          //第一个操作数是NaN，结果NaN
console.log(null && false);         //第一个操作数是null，结果null
console.log('' && false);           //第一个操作数是空串，结果空串
console.log(0 && 100);              //结果是0
console.log(5 && 100);              //100
console.log(a && b);                //hello
console.log(obj && 200);            //200
console.log(d && a); //undefined
```

（2）||：表示or

如果第一个操作数是true或者能够转为true，结果就是第一个操作数，否则结果是第二个操作数。

可以理解为：如果运算的第一个操作数为 true,则返回第一个操作数,反之则返回第二个操作数。

undefined、null、NaN、0、空字符将会被转为false。

```
var a = [1, 2, 3];
var b = "hello";
var obj = new Object();
var d;

console.log(true || 10); //第一个操作数是true，结果是第一个操作，也就是true
console.log(false || b); //第一个操作数是false，结果是第二个操作数b
console.log(100 || false); //第一个操作数是100，结果100
console.log(undefined || 9); //第一个操作数是undefined转false，结果9
console.log(NaN || false); //第一个操作数是NaN转false，结果第二个操作数
console.log(null || a); //第一个操作数是null转false，结果a
console.log('' || false); //第一个操作数是空串转false，结果第二操作数
console.log(0 || 100); //结果是100
console.log(5 || 100); //5
console.log(a || b); //a
console.log(obj || 200); //obj
```

（3）!：表示not

将表达式强制转换成true或false。表达式可以为字面量，函数，对象。

```
var flag = false;
if(!flag) {
    console.log("代码被执行了");
}
```

5、条件运算符（三目运算符）

x?y:z

如果表达式`x`为`true`，运算符就会返回 表达式`y`的值；否则， 就会返回 表达式`z`的值。可以将其改写为if...else语句。

```
var isRed = true;
console.log(isRed ? "red" : "green"); //red
console.log(!isRed ? "red" : "green"); //green

//嵌套运算
var a = 10;
console.log(a >= 0 ? (a%2 == 0 ? "a是偶数" : "a是奇数") : "a是负数");
```

6、运算符优先级

非、算、关、与、或、赋

非： ！、单目运算符（+、-、++、--）

算： 算数运算符

关：关系运算符

与： &&

或： ||

三目运算符

赋值： =

条件语句

条件语句用于基于不同的条件来执行不同的动作。

在 JavaScript 中，我们可使用以下条件语句：

- **if 语句** - 只有当指定条件为 true 时，使用该语句来执行代码
- **if...else 语句** - 当条件为 true 时执行代码，当条件为 false 时执行其他代码
- **if...else if....else 语句**- 使用该语句来选择多个代码块之一来执行
- **switch 语句** - 使用该语句来选择多个代码块之一来执行

1、if...else语句

（1）if语句

只有当指定条件为 true 时，该语句才会执行代码。

语法：

```
if (condition) {
    //当条件为 true 时执行的代码
}
```

condition可以是字面量、变量、表达式，都来表示true或false的值

```
var a = 10;
if(a > 0) {
    console.log("a是正数");
}
```

（2）if...else语句

请使用 if....else 语句在条件为 true 时执行代码，在条件为 false 时执行其他代码。

语法：

```
if (condition) {
    //当条件为 true 时执行的代码
} else {
    //当条件不为 true 时执行的代码
}
```

```
var time = window.prompt();
if (time < 20) {
    x="Good day";
} else {
    x="Good evening";
}
console.log(x); //Good day
```

（3）if...else if...else 语句

使用 if....else if...else 语句来选择多个代码块之一来执行。

语法：

```
if (condition1) {
    //当条件 1 为 true 时执行的代码
} else if (condition2) {
    //当条件 2 为 true 时执行的代码
} else {
   //当条件 1 和 条件 2 都不为 true 时执行的代码
}
```

```
var time = window.prompt();
if (time < 10) {
    document.write("<b>早上好</b>");
} else if (time>=10 && time<16) {
    document.write("<b>今天好</b>");
} else  {
    document.write("<b>晚上好!</b>");
}
```

如果if与else语句中只有一行代码时，{}可以省略

2、switch...case语句

语法：

```
switch(n) {
    case 1:
        执行代码块 1
        break;
    case 2:
        执行代码块 2
        break;
    default:
        与 case 1 和 case 2 不同时执行的代码
}
```

工作原理：

- 首先设置表达式 *n*（通常是一个变量）。随后表达式的值会与结构中的每个 case 的值做比较。如果存在匹配，则与该 case 关联的代码块会被执行。
- 使用 **break** 来阻止代码自动地向下一个 case 运行。
- 使用 default 关键词来规定匹配不存在时做的事情。

```
var d = new Date().getDay();
switch(d) {
    case 0:
        x = "今天是星期日";
        break;
    case 1:
        x = "今天是星期一";
        break;
    case 2:
        x = "今天是星期二";
        break;
    case 3:
        x = "今天是星期三";
        break;
    case 4:
        x = "今天是星期四";
        break;
    case 5:
        x = "今天是星期五";
        break;
    case 6:
        x = "今天是星期六";
        break;
    default:
        x="期待周末";
        break;
}
```

注意：

- break语句用于结束该分支的执行。如果没有break语句，那么程序会接着往下执行，知道遇到break为止。
- switch 中 case的判断是===的判断，即数据类型和值的双重判断。
- switch的判断条件可以是String 、Number、Boolean、null、undefined。

循环语句

循环可以将代码块执行指定的次数。

JavaScript 支持不同类型的循环：

- **for** - 循环代码块一定的次数
- **for/in** - 循环遍历对象的属性
- **while** - 当指定的条件为 true 时循环指定的代码块
- **do/while** - 同样当指定的条件为 true 时循环指定的代码块

1、for循环

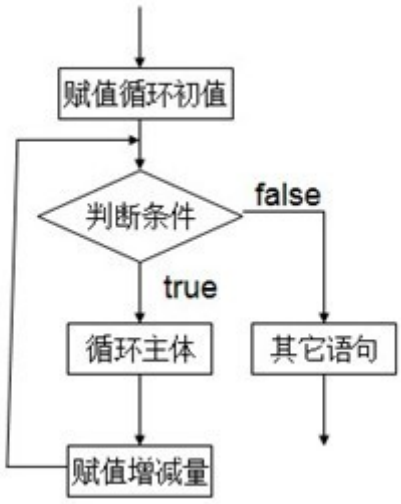
（1）语法

```
for (表达式1; 表达式2; 表达式3) {
    循环语句;
}
```

- 表达式1：初始化表达式，负责完成变量的初始化。
- 表达式2：循环条件表达式，值为boolean型的表达式，指定循环条件。
- 表达式3：循环后操作表达式，负责修整变量，改变循环条件
- 3个表达式之间使用分号分隔

```
for (var i = 0; i < 5; i++) {
      console.log(i);
}
```

（2）执行流程



示例1：使用for循环输出1到10

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

示例2：使用for循环计算1+2+3+…+100的值

```
var sum = 0;
for( var i = 0; i <= 100; i++){
    sum += i; //sum = sum + i;
}
console.log("和为：" + sum);
```

（3）嵌套循环

就是在循环体中再嵌套一层for循环，主要有两种形式：

a.内外循环独立进行：内部的循环条件是固定的

示例：使用双层for循环打印一个5行5列的矩阵。

```
for(var i = 0; i < 5; i++) {
    for(var j = 0; j < 5; j++) {
        document.write("#");
    }
    document.write("<br />");
}
```

b.内部循环依赖于外部循环：内部循环的条件**每次都是变化**的,变化是依赖于外部条件

示例：

```
for(var i = 0; i < 5; i++) {
    for(var j = 0; j <= i; j++) {
        document.write("#");
    }
    document.write("<br />");
}
```

2、for/in

循环遍历对象的属性或数组中的元素

```
var person={fname:"John",lname:"Doe",age:25};

for (var x in person) { // x 为键名
    console.log(person[x]); //访问值时语法是对象[键名]
}

var arr = ["abc", "123", "www"];
for(x in arr) {
    console.log(arr[x]);
}
```

for 循环除了使用 in 方式来循环数组，还提供了一个方式： of，遍历数组时更加方便。

for...of 是 ES6 新引入的特性。它既比传统的for循环简洁，同时弥补了forEach和for-in循环的短板。

语法：

```
for (var value of myArray) {
  console.log(value);
}
```

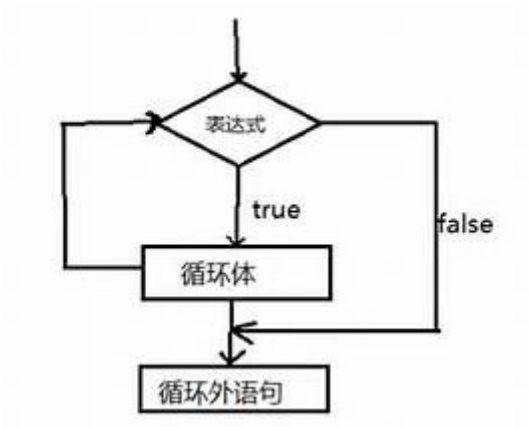
3、while循环

while 循环会在指定条件为真时循环执行代码块。只要指定条件为 true，循环就可以一直执行代码块。如果指定条件的值为false，那么会结束循环的执行。

（1）语法：

```
while (条件) {
    需要执行的代码
}
```

（2）执行流程



示例：使用while循环计算2~200之间所有偶数的和。.

```
var sum = 0;
var i = 2;
while(i <= 200) {
    if(i%2 == 0) {
        sum += i;
    }
    i++;
}
console.log("和为：" + sum);
```

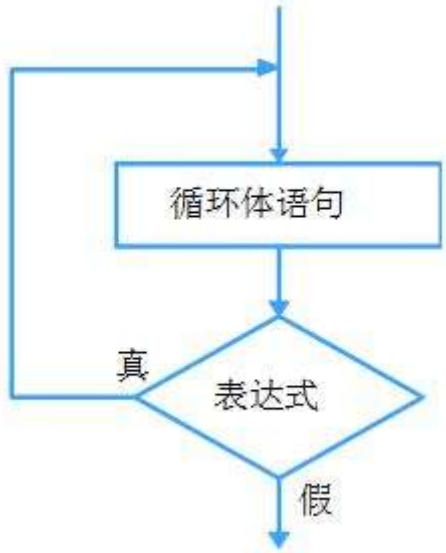
4、do...while循环

与while语句类似，都是用于循环。区别在于while语句是先判断条件是否成立再执行循环体，而do…while语句则是**先执行循环体**，再来判断条件是否成立。也就是说在条件都不满足的情况下，do…while至少执行了一次循环体。

（1）语法：

```
do {
    循环体
} while(条件表达式)
```

（2）执行流程



示例：使用do...while循环计算2~200之间所有偶数的和。

```
var sum = 0;
var i = 2;
do {
    if(i%2 == 0) {
        sum += i;
    }
    i++;
} while(i <= 200);
console.log("和为：" + sum);
```

5、break与continue

（1）break

用于跳出循环。可以与标签label结合使用来指定跳出循环

```
for (i=0;i<10;i++) {
    if (i==3) {
        break;
    }
    x = x + "The number is " + i + "<br>";
}

out: //标签
    for(var i = 1; i <= 9; i++) {
        for(var j = 1; j <= i; j++) {
            document.write(j + "*" + i + "=" + (i * j) + "\t");
            if(j >= 5) {
                break out; //根据标签结束指定的for循环
            }
        }
        document.write("<br />");
    }
```

（2）continue

用来略过循环中剩下的语句，重新开始下一次新的循环。可以和标签label结合使用。

```
for (i=0;i<=10;i++) {
    if (i==3) {
        continue;
    }
    x = x + "The number is " + i + "<br>";
}

for(var i = 1; i <= 10; i++) {
    if(i % 2 != 0) { //表示奇数，遇到奇数就跳过，只打印偶数
        continue; //略过循环中剩下的语句，重新开始下一次新的循环
    }
    console.log(i);
}

out: //标签
    for(var i = 1; i <= 9; i++) {
        for(var j = 1; j <= i; j++) {
            if(j >= 5) {
                continue out; //根据标签指定循环的层级来略过剩下的语句，重新开始新的循环
            }
            document.write(j + "*" + i + "=" + (i * j) + "\t");
        }
        document.write("<br />");
    }
```

其他语句

1、eval()函数

eval() 函数计算 JavaScript 字符串，并把它作为脚本代码来执行。

如果参数是一个表达式，eval() 函数将执行表达式。如果参数是Javascript语句，eval()将执行 Javascript 语句。

- **语法**：

```
eval(string)。
参数：string：必需的，表示要计算的字符串，其中含有要计算的 JavaScript 表达式或要执行的语句。
返回值：通过计算 string 得到的值（如果有的话）。
```

该方法只接受原始字符串作为参数，如果 string 参数不是原始字符串，那么该方法将不作任何改变地返回。因此请不要为 eval() 函数传递 String 对象来作为参数。

如果试图覆盖 eval 属性或把 eval() 方法赋予另一个属性，并通过该属性调用它，则 ECMAScript 实现允许抛出一个 EvalError 异常。

- **示例**：

```
eval("x=10;y=20;document.write(x*y)");
console.log(eval("2+2")); //4
var x = 10
console.log(eval(x + 17)); //27


eval("2+3"); // 返回 5
var myeval = eval; // 可能会抛出 EvalError 异常
myeval("2+3"); // 可能会抛出 EvalError 异常
```

2、with

（1）定义及使用

with 语句的原本用意是为逐级的对象访问提供命名空间式的速写方式. 也就是在指定的代码区域, 直接通过节点名称调用对象。

with 通常被当做重复引用同一个对象中的多个属性的快捷方式，可以不需要重复引用对象本身。

比如，目前现在有一个这样的对象：

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};
```

如果想要改变 obj 中每一项的值，一般写法可能会是这样：

```
// 重复写了3次的“obj”
obj.a = 2;
obj.b = 3;
obj.c = 4;
```

而用了 with 的写法，会有一个简单的快捷方式:

```
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

在这段代码中，使用了 with 语句关联了 obj 对象，这就意味着在 with 代码块内部，每个变量首先被认为是一个局部变量，如果局部变量与 obj 对象的某个属性同名，则这个局部变量会指向 obj 对象属性。

（2）with的弊端：不推荐使用with

- **导致数据泄漏**

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
}

foo(o1);
console.log(o1.a);  //2

foo(o2);
console.log(o2.a);  //undefined
console.log(a);     //不好，a被泄漏到全局作用域上了
```

当我们传递 o2 给 with 时，with 所声明的作用域是 o2, 从这个作用域开始对 a 进行 LHS查询。o2 的作用域、foo(...) 的作用域和全局作用域中都没有找到标识符 a，因此在**非严格模式**下，会自动在全局作用域创建一个全局变量），在严格模式下，会抛出ReferenceError 异常。

另一个不推荐 with 的原因是。在严格模式下，with 被完全禁止

- **性能下降**

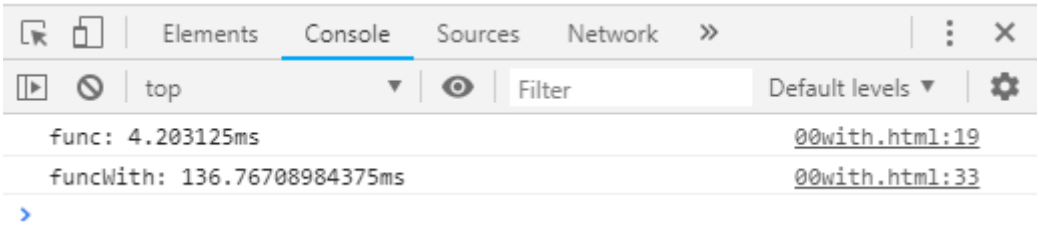
with 会在运行时修改或创建新的作用域，以此来欺骗其他在书写时定义的词法作用域。

```
function func() {
    console.time("func");
```

```
    var obj = {
        a: [1, 2, 3]
    };
    for(var i = 0; i < 100000; i++)
    {
        var v = obj.a[0];
    }
    console.timeEnd("func");
}
func();

function funcWith() {
    console.time("funcWith");
    var obj = {
        a: [1, 2, 3]
    };
    with(obj) {
        for(var i = 0; i < 100000; i++) {
            var v = a[0];
        }
    }
    console.timeEnd("funcWith");
}
funcWith();
```

测试效果：



处理相同逻辑的代码，没有使用with的运行时间比使用了with的运行时间要短很多。

原因是 **JavaScript 引擎会在编译阶段进行数项的性能优化**。其中有些优化依赖于能够根据代码的词法进行静态分析，并预先确定所有变量和函数的定义位置，才能在执行过程中快速找到标识符。

但如果引擎在代码中发现了 with，它只能简单地假设关于标识符位置的判断都是无效的，因为无法知道传递给 with 用来创建新词法作用域的对象的内容到底是什么。

最悲观的情况是如果出现了 with ，所有的优化都可能是无意义的。因此引擎会采取最简单的做法就是**完全不做任何优化**。如果代码大量使用 with 或者 eval()，那么运行起来一定会变得非常慢。无论引擎多聪明，试图将这些悲观情况的副作用限制在最小范围内，也无法避免如果没有这些优化，代码会运行得更慢的事实。

3、debugger

- **debugger** 关键字用于停止执行 JavaScript，并调用调试函数。
- 这个关键字与在调试工具中设置断点的效果是一样的。
- 如果没有调试可用，debugger 语句将无法工作。

```
<p id="demo"></p>
<script>
    var x = 15 * 5;
    debugger;
    document.getElementById("demo").innerHTML = x;
</script>
```

4、use strict

- "use strict" 指令在 JavaScript 1.8.5 (ECMAScript5) 中新增。
- 它不是一条语句，但是是一个字面量表达式，在 JavaScript 旧版本中会被忽略。
- "use strict" 的目的是指定代码在严格条件下执行。
- 严格模式下你不能使用未声明的变量。

（1）严格模式声明

严格模式通过在脚本或函数的头部添加 "**use strict**" 表达式来声明。

- 在脚本中声明表示全局作用域：

```
<script type="text/javascript">
    "use strict";
    x = 3.14; // 报错 (x 未定义)
</script>
```

- 在函数内部声明是局部作用域 (只在函数内使用严格模式):

```
x = 3.14;       // 不报错
myFunction();

function myFunction() {
   "use strict";
    y = 3.14;   // 报错 (y 未定义)
}
```

（2）为什么使用严格模式

消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为：

- 消除代码运行的一些不安全之处，保证代码运行的安全；
- 提高编译器效率，增加运行速度；
- 为未来新版本的Javascript做好铺垫。

"严格模式"体现了Javascript更合理、更安全、更严谨的发展方向，包括IE 10在内的主流浏览器，都已经支持它，许多大项目已经开始全面拥抱它。

另一方面，同样的代码，在"严格模式"中，可能会有不一样的运行结果；一些在"正常模式"下可以运行的语句，在"严格模式"下将不能运行。掌握这些内容，有助于更细致深入地理解Javascript，让你变成一个更好的程序员。

（3）严格模式的限制

- 不允许使用未声明的变量。对象也是一个变量

```
"use strict";
x = 3.14;                // 报错 (x 未定义)
y = {p1:10, p2:20};      // 报错 (y 未定义)
```

- 不允许删除变量或对象

```
"use strict";
var x = 3.14;
delete x;                // 报错
```

- 不允许删除函数

```
"use strict";
function x(p1, p2) {};
delete x;                // 报错
```

- 不允许变量名重名

```
"use strict";
function x(p1, p1) {};   // 报错
```

- 不允许使用八进制

```
"use strict";
var x = 010;             // 报错
```

- 不允许使用转义字符

```
"use strict";
var x = \010;            // 报错
```

- 不允许对只读属性赋值

```
"use strict";
var obj = {};
Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14;            // 报错
```

- 不允许对一个使用getter方法读取的属性进行赋值

```
"use strict";
var obj = {get x() {return 0} };

obj.x = 3.14;            // 报错
```

- 不允许删除一个不允许删除的属性

```
"use strict";
delete Object.prototype; // 报错
```

- 禁止this关键字指向全局对象

```
function f(){
    return !this;
}
// 返回false，因为"this"指向全局对象，"!this"就是false

function f(){
    "use strict";
    return !this;
}
// 返回true，因为严格模式下，this的值为undefined，所以"!this"为true。
```

因此，使用构造函数时，如果忘了加new，this不再指向全局对象，而是报错。

```
function f(){
    "use strict";
    this.a = 1;
};
f();// 报错，this未定义
```

5、try...catch...finally、throw

- **try** ：检测代码块的错误。
- **catch** ：当 try 代码块发生错误时，所执行的代码块，用来处理错误。JavaScript 语句 **try** 和 **catch** 是成对出现的。
- **finally** ：在 try 和 catch 语句之后，无论是否有触发异常，该语句都会执行。
- **throw** ：当错误发生时，当事情出问题时，JavaScript 引擎通常会停止，并生成一个错误消息。描述这种情况的技术术语是：JavaScript 将**抛出**一个错误。

（1）try...finally

- **语法**：

```
try {
    ...     //异常的抛出
} catch(e) {
    ...     //异常的捕获与处理
} finally {
    ...     //结束处理
}
```

- **实例**：

```
var txt = "";

function message() {
    try {
        adddlert("Welcome guest!");
    } catch(err) {
        txt = "本页有一个错误。\n\n";
        txt += "错误描述：" + err.message + "\n\n";
        txt += "点击确定继续。\n\n";
        alert(txt);
    }
}
message();
```

（2）finally语句块

finally 语句不论之前的 try 和 catch 中是否产生异常都会执行该代码块。

```
<p>请输入 5 ~ 10 之间的数字：</p>
<input id="demo" type="text">
<button type="button" onclick="myFunction()">点我</button>
<p id="p01"></p>

function myFunction() {
    var message, x;
    message = document.getElementById("p01");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "") throw "值是空的";
        if(isNaN(x)) throw "值不是一个数字";
        x = Number(x);
        if(x > 10) throw "太大";
        if(x < 5) throw "太小";
    } catch(err) {
        message.innerHTML = "错误：" + err + ".";
    } finally {
        document.getElementById("demo").value = "";
    }
}
```

（3）throw 语句

throw 语句允许我们创建自定义错误。

正确的技术术语是：创建或**抛出异常**（exception）。

如果把 throw 与 try 和 catch 一起使用，那么您能够控制程序流，并生成自定义的错误消息。

语法：

```
throw exception
```

对象

1、对象的定义

对象是JavaScript的一个引用数据类型，是一种复合值，它将很多值（原始值或者其他对象）聚合在一起，可通过名字访问这些值。即属性的无序集合。

2、对象基础

（1）对象创建

- **对象直接量 / 字面量**

```
var person = {
    name: '小明',
    age: 18,
    say: function(data){
        console.log(`${this.name}说了${data}`);
    }
}
console.log(person.name); // 小明
person.say("hello"); //小明说了hello
```

对象字面量可以用来创建单个对象，但如果要创建多个对象，会产生大量的重复代码。

- **工厂模式**

为了解决上述问题，人们开始使用工厂模式。该模式抽象了创建具体对象的过程，用函数来封装以特定接口创建对象的细节

```
function createPerson(name,age,job){
    var o = new Object(); //创建一个具体的对象
    o.name = name;
    o.age = age;
    o.job = job;
    o.say = function(){
        console.log(`${this.name}说了${data}`);
    }
    return o;
}

var person1 = createPerson('小明', 20, 'hello');
var person2 = createPerson('小花', 18, 'javascript is so easy');
```

- **构造函数模式**
    - 系统自带的构造函数： 如：new Object(), Array(), Number(), String(), Boolean()...

```
var obj = new Object();
obj.name = '小明';
console.log(obj.name); //小明
```

- 可以通过创建自定义的**构造函数**，来定义自定义对象类型的属性和方法。创建自定义的构造函数意味着可以将它的实例标识为一种特定的类型，而这正是构造函数模式胜过工厂模式的地方。该模式没有显式地创建对象，直接将属性和方法赋给了this对象，且**没有return语句。为了和普通函数区分，首字母大写，采用大驼峰式写法。**

```
function Person(name,age,job){
    this.name = name;
    this.age = age;
    this.jog = job;
    this.say = function(){
        console.log(`${this.name}说了${data}`);
    };
}
var person1 = new Person('小明', 20, 'hello');
var person2 = new Person('小花', 18, 'javascript is so easy');
//具有相同作用的sayName()方法在person1和person2这两个实例中却占用了不同的内存空间
console.log(person1.say === person2.say);//false
```

使用构造函数的主要问题是每个方法都要在每个实例上重新创建一遍。

- **new创建**

new后面跟一个函数表示创建对象；这里的函数是构造函数。

```
var obj1 = new Object();
console.log(obj1);
var obj2 = new Date();
console.log(obj2);
var obj3 = new Array();
console.log(obj3);
```

- **Object.create()创建**

用于创建一个新对象，参数是这个**对象的原型**（后续章节会详细介绍原型的概念）

```
var b = Object.create(new Array(1,4,5,6,2,3,7));
console.log(b.length);
var c = b.sort();//数组排序
console.log(c);
console.log(c.slice(2, 3));//数组截取[3]

var nu = Object.create(new String('null'));
console.log(nu[2]);//字符串->l

var nu = Object.create(null);
console.log(nu[2]);//undefined
```

（2）对象方法

- **valueOf()**：返回当前对象原始值

```
var o = new Object();
o.valueOf() === o // true

console.log(new Date().valueOf())//得到XX ms（毫秒）
```

object 引用是任何内部 JavaScript 对象，将通过不同的方式为每个内部 JavaScript 对象定义 valueOf 方法。**Math** 和 **Error** 对象没有 valueOf 方法。

对象	返回值
数组Array	返回数组实例
布尔值Boolean	布尔值
日期Date	从UTC1970年1月1日0时0分0秒开始到目前为止的时间毫秒值
函数Function	函数本身
数值Number	数字值
对象Object	对象本身，默认值
字符串String	字符串值

- **toString()**：返回当前对象对应的字符串形式

```
var o1 = new Object();
console.log(o1.toString()); // "[object Object]"

var o2 = {a:1};
console.log(o2.toString()); // "[object Object]"

//函数调用该方法返回的是函数本身的代码
function show(argument) {
    // body...
}
console.log(show.toString());

var a = [1,2,3];
console.log(a.toString());//"1,2,3"

console.log(new Date().toString())//日期字符串
```

toString 方法是一个所有内置的 JavaScript 对象的成员。 它的行为取决于对象的类型：

Object	行为
数组Array	将Array的元素转换为字符串，结果字符串被连接起来，用逗号分隔
布尔值 Boolean	如果布尔值为true，则返回"true",否则返回"false"
日期Date	返回日期的文本表示形式
错误Error	返回一个包含相关错误信息的字符串
函数 Function	返回如下格式的字符串，其中functionName是函数的名称 function functionName() { [native code] }
数值 Number	返回数字的文字表示形式
字符串 String	返回String对象的值
默认{}	返回"object Object"

附注：valueOf偏向于运算，toString偏向于显示。

1. 在进行强转字符串类型时将优先调用toString方法，强转为数字时优先调用valueOf。
2. 在有运算操作符的情况下，valueOf的优先级高于toString。

（3）对象属性

- **定义和修改属性**：
  - **Object.defineProperty()**：此方法用于直接在一个对象上定义一个新属性，或者修改一个已经存在的属性， 并返回这个对象。
  - **语法**：

```
Object.defineProperty(obj, prop, descriptor)
```

  - **参数说明**：

```
obj 需要定义属性的对象。
prop 需被定义或修改的属性名。
descriptor 需被定义或修改的属性的描述符，是一个对象形式。
```

  - **数据描述符和存取描述符均具有以下可选键值**：

```
- configurable: 仅当该属性的 configurable 为 true 时，该属性才能够被改变，也能够被删除。默认为 false
- enumerable: 仅当该属性的 enumerable 为 true 时，该属性才能够出现在对象的枚举属性中。默认为 false
- value: 该属性对应的值。可以是任何有效的 JavaScript 值（数值，对象，函数等）。默认为 undefined
- writable: 仅当仅当该属性的writable为 true 时，该属性才能被赋值运算符改变。默认为 false
- get: 一个给属性提供 getter 的方法，如果没有 getter 则为 undefined。该方法返回值被用作属性值。undefined
- set: 一个给属性提供 setter 的方法，如果没有 setter 则为 undefined。该方法将接受唯一参数，并将该参数的新值分配给该属性。默认为undefined。
```

```
var person = {
    name: 'yourname',
    age: 10
};
Object.defineProperty(person, "sex", {
    value: "male",
    enumerable: false           //不可枚举
});
```

- **检测属性**：
  - **in**：检查一个属性是否属于某个对象，包括继承来的属性；

```
var person = {
    name:'yourname',
    age:10
};
person.__proto__.LastName = "deng"; //让person继承一个LastName属性
Object.defineProperty(person, "sex", {
            value: "male",
            enumerable: false   //不可枚举
});
console.log('name' in person);     //true
console.log('sex' in person);      //true
console.log('toString' in person); //true
console.log('LastName' in person); //true
```

  - **hasOwnProperty()**：检查一个属性是否属于某个对象自有属性，不包括继承来的属性；

```
var person = {
    name:'yourname',
    age:10
};
person.__proto__.LastName = "deng"; //让person继承一个LastName属性
Object.defineProperty(person, "sex", {
            value: "male",
            enumerable: false           //不可枚举
});
console.log(person.hasOwnProperty('name'));     //true
console.log(person.hasOwnProperty('sex'));      //true
console.log(person.hasOwnProperty('toString')); //false
console.log(person.hasOwnProperty('LastName')); //false
```

- **propertyIsEnumerable()**：是hasOwnProperty()的增强版，检查一个属性是否属于某个对象自有属性，不包括继承来的属性，且该属性可枚举。

```
var person = {
    name:'yourname',
    age:10
};
person.__proto__.LastName = "deng"; //让person继承一个LastName属性
Object.defineProperty(person, "sex", {
            value: "male",
            enumerable: false   //不可枚举
});
console.log(person.propertyIsEnumerable('name'));     //true
console.log(person.propertyIsEnumerable('sex'));      //false
console.log(person.propertyIsEnumerable('toString')); //false
console.log(person.propertyIsEnumerable('LastName')); //false
```

- **存取器属性getter和setter**

属性值可以由一个或两个方法替代，这两个方法就是getter和setter

（a）由getter和setter定义的属性，称为“存取器属性”；

（b）一般的只有一个值的属性称为“数据属性”；

```
var myObj = {
    a: 2,
    get b(){
        return 3;
    }
};

console.log(myObj.a);//2，属性a称为数据属性
console.log(myObj.b);//3，属性b称为存取器属性，属性名称与函数名相同
```

与存取器属性同名的函数定义没有使用function关键字，而是使用get或set，也没有使用冒号将属性名和函数体分开，但函数体的结束和下一个方法之间有逗号隔开。

（c）查询存取器属性的值，用getter；拥有getter则该属性可读；

（d）设置存取器属性的值，用setter；拥有setter则该属性可写；

```
```

```
myObj.b = 5;
console.log(myObj.b); //3,属性打印仍然为3

var myObj = {
    a: 2,
    get b() {
        if(this._b_ == undefined) return 3;
        else return this._b_;
    },
    set b(val) {
        this._b_ = val;
    }
};
console.log(myObj.b); //3
myObj.b = 100;
console.log(myObj.b); //100
```

```
var myobj = {
    //数据属性
    myname:'yourname',
    birthday:'1983-05-17',
    get myage(){
        return (new Date().getFullYear()) - new
Date(this.birthday).getFullYear();
    },
    set myage(value){
        this.birthday = value;
    }
};
myobj.myage = '1998-08-22';//相当于调用了set myage(value)方法
console.log(myobj.myage);//相当于调用了个get myage()方法
```

- **删除属性**：使用delete关键字可以删除对象属性

```
var o = {
    a : 1
};
console.log(o.a);//1
console.log('a' in o);//true
console.log(delete o.a);//true
console.log(o.a);//undefined
console.log('a' in o);//false
```

注意：

> -- 只能删除自有属性，不能删除继承属性。
>
> -- 删除成功或不存在的属性时，返回true。

（4）序列化对象

序列化对象是指将对象的状态转成字符串，也可以将字符串还原为对象；

(a) 转成字符串：JSON.stringify()；

(b) 还原为对象：JSON.parse()；

```
var hqyj = { name:'华清远见', add:'科华北路99', tel:['0900', 8304, '0910']};
var str = JSON.stringify(hqyj);
console.log(str,  typeof str);

var obj1 = JSON.parse(str);
console.log(obj1,  typeof obj1);
```

3、对象的增删改查

（1）增

所谓增添一个对象的属性，就是直接对该属性进行赋值操作即可，这就相当于为该对象添加了一个新属性，而打印未添加的属性，浏览器不会报错，而是会打印出undefined

```
var obj = {};
console.log(obj.name); //undefined （不会报错）
obj.name = 'xx';
console.log(obj.name); // xx
```

（2）删

我们通过delete操作符来删除一个对象的属性

```
var obj = {
    name : 'xx'
};
console.log(obj.name); //xx
delete obj.name;
console.log(obj.name); //undefined
```

（3）改

修改一个对象的属性是最简单的了，直接通过赋值操作赋予其其他的值即可

```
var obj = {
    name: 'xx'
};
console.log(obj.name); // xx
obj.name = 'yy';
console.log(obj.name); // yy
```

（4）查

查询一个对象的属性值有两种方法

```
var obj = {
    name: 'xx'
};
// 第一种方法
console.log(obj['name']); //xx

//  第二种方法
console.log(obj.name); // xx
  //最本质的是第一种方法，因为在使用第二种方法时，后台自动将其转换为第一种字符串的形式来查询
```

注意：

- 以上的增、删、改三种操作都只是针对当前对象的属性进行操作，而不会影响到当前对象的原型的属性。
- 而查询是先看看当前对象本身是否设置了该属性，如果当前对象未设置该属性，则再看该对象的原型中是否设置了该属性，若两者都没有，则返回undefined

4、包装类

- 五个原始值：number, string , boolean, undefined, null。其中number, string, boolean是分别拥有自己的包装类，而undefined和null是没有自己的包装类的
- 原始值不是对象，无法拥有自己的属性，但因为的包装类的存在，原始值就好似可以拥有自己的属性了，但其拥有的属性又有点特殊之处，如下用string来举例：

```
//  str是string类型的，非对象，不能拥有属性，为什么能打印出str.length?
var str = 'abcd';
console.log(str.length); //4
 //因为每次执行完一条完整js语句后该类型对象的包装类就会将该语句包装，所以也就不会导致报错了，这些都是后台自己写的
```

5、原型

```
1. 定义：原型是function对象的一个属性，它定义了构造函数制造出的对象的公共祖先。通过该构造函数产生的对象，可以继承该原型的属性和方法。原型也是对象。

2. 利用原型特点和概念，可以提取共有属性。将一类对象的共有属性提取出来，放到该类对象的原型中，从而不需要每次用new操作符时都重新定义一遍该共有属性。

3. 对象如何查看对象的构造函数-->constructor

4. 对象如何查看原型-->隐式属性__proto__
```

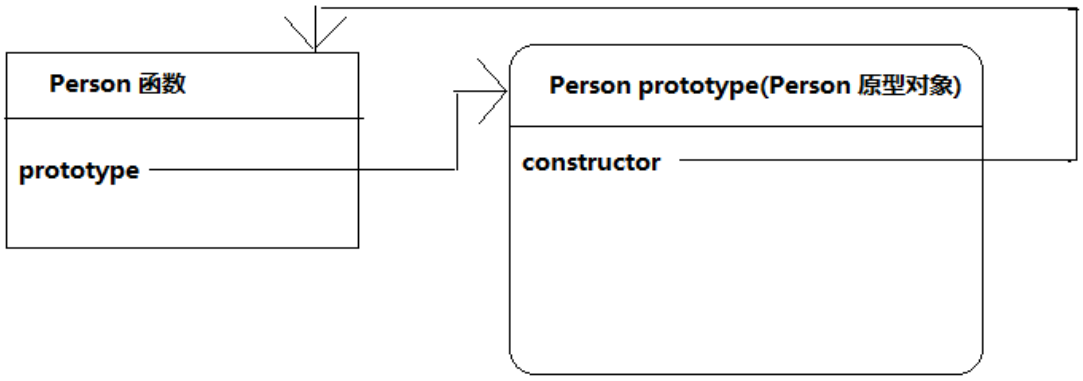
（1）函数的原型对象

在JavaScript中，我们创建一个函数A,那么浏览器就会在内存中创建一个对象B，而且每个函数都默认会有一个属性 prototype 指向了这个对象(即：prototype的属性的值是这个对象)。这个对象B就称作是函数A的原型对象，简称函数的原型。这个原型对象B 默认会有一个属性constructor指向了这个函数A (意思就是说：constructor属性的值是函数A)。

```
 /*
    声明一个函数，则这个函数默认会有一个属性叫 prototype 。
    而且浏览器会自动按照一定的规则创建一个对象，
    这个对象就是这个函数的原型对象，prototype属性指向这个原型对象。
    这个原型对象有一个属性叫constructor 执行了这个函数

    注意：原型对象默认只有属性--constructor。
    其他都是从Object继承而来，暂且不用考虑。
*/
function Person() {

}
```

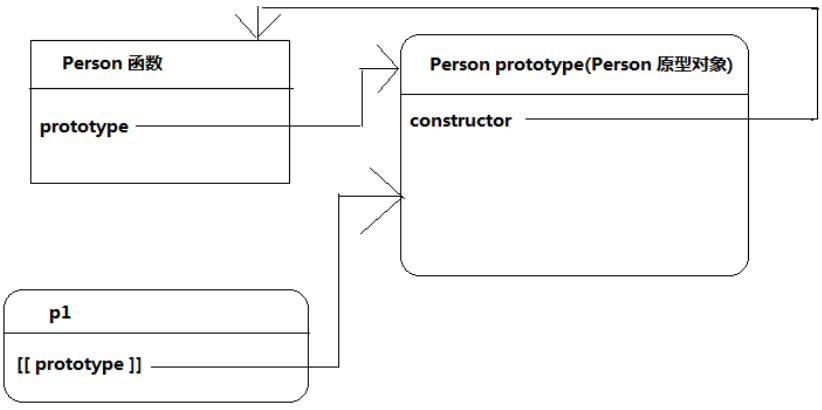


（2）使用构造函数创建对象

当把一个函数作为构造函数 使用new创建对象的时候，那么这个对象就会存在一个默认的不可见的属性，来指向了构造函数的原型对象。这个不可见的属性我们一般用[[prototype]]来表示，只是这个属性没有办法直接访问到。

```
function Person () {

}
/*
 利用构造函数创建一个对象，则这个对象会自动添加一个不可见的属性 [[prototype]]
 而且这个属性指向了构造函数的原型对象。
*/
var p1 = new Person();
```



- 从上面的图示中可以看到，创建p1对象虽然使用的是Person构造函数，但是对象创建出来之后，这个p1对象其实已经与Person构造函数没有任何关系了，p1对象的[[prototype]]属性指向的是

Person构造函数的原型对象。

- 如果使用new Person()创建多个对象，则多个对象都会同时指向Person构造函数的原型对象。
- 我们可以手动给这个原型对象添加属性和方法，那么p1,p2,p3...这些对象就会共享这些在原型中添加的属性和方法。
- 如果我们访问p1中的一个属性name，如果在p1对象中找到，则直接返回。如果p1对象中没有找到，则直接去p1对象的[[prototype]]属性指向的原型对象中查找，如果查找到则返回。(如果原型中也没有找到，则继续向上找原型的原型—原型链。 后面再讲)。
- 如果通过p1对象添加了一个属性name，则p1对象来说就屏蔽了原型中的属性name。 换句话说：在p1中就没有办法访问到原型的属性name了。
- 通过p1对象只能读取原型中的属性name的值，而不能修改原型中的属性name的值。 p1.name = “李四”; 并不是修改了原型中的值，而是在p1对象中给添加了一个属性name。

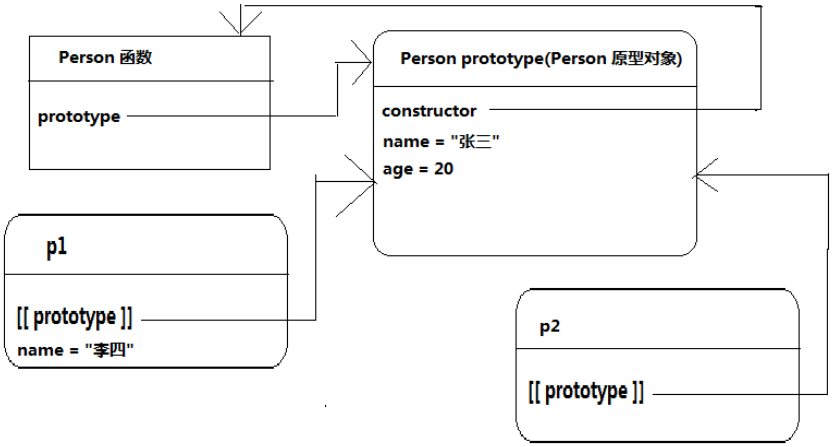
```
function Person () {
}
// 可以使用Person.prototype 直接访问到原型对象
//给Person函数的原型对象中添加一个属性 name并且值是 "张三"
Person.prototype.name = "张三";
Person.prototype.age = 20;

var p1 = new Person();
/*
  访问p1对象的属性name，虽然在p1对象中我们并没有明确的添加属性name，但是
  p1的 [[prototype]] 属性指向的原型中有name属性，所以这个地方可以访问到属性name的值。
  注意：这个时候不能通过p1对象删除name属性，因为只能删除在p1中删除的对象。
*/
alert(p1.name);  // 张三

var p2 = new Person();
alert(p2.name);  // 张三  都是从原型中找到的，所以一样。

alert(p1.name === p2.name);  // true

// 由于不能修改原型中的值，则这种方法就直接在p1中添加了一个新的属性name，然后在p1中无法再访问到
//原型中的属性。
p1.name = "李四";
alert("p1: " + p1.name);
// 由于p2中没有name属性，则对p2来说仍然是访问的原型中的属性。
alert("p2:" + p2.name);  // 张三
```



（3）查看原型

直接通过new操作符创建的对象访问隐式属性__proto__即可

```
//原型
Person.prototype = {
    eat: function(food) {
        console.log('I have eated ' + food);
    },
    sleep: function() {
        console.log("I am sleeping");
    }
}

// 构造函数
function Person(name, age) {
    this.name = name;
    this.age = age;
}
var person1 = new Person('xx', 18);
console.log(person1.__proto__); //查看原型
```

（4）查看对象的构造器

通过属性constructor来查看

contructor属性位于构造函数的原型中，其中存储的是构造函数信息，所以在不知道原型的情况下，由原型继承原理，我们可以用实例对象来直接访问constructor，即获取创建该实例的构造函数

```
//构造函数
function Person() {
    this.name = 'myName';
    this.age = 18;
}
var person = new Person();
console.log(person.constructor);
```

6、原型链

每一个对象都从原型继承属性，直到null结束。

所有的内置构造函数都有一个继承自Object.prototype的原型，我们可以看下原型链：

```
var arr1 = new Array(1,2,3);
//arr1.__proto__-->Array.prototype的原型__proto__-->Object.prototype的原型
__proto__
//-->null; 形成链，到null结束，完美；
var date1 = new Date();
//date1.__proto__-->Date.prototype的原型__proto__-->Object.prototype的原型
__proto__
//-->null; 形成链，到null结束，完美；
```

原型链就是将一个个原型串连起来，形成一条原型继承的链子。原型链的顶端是Object

```
//原型链： Child -> new Parent() -> new GrandParent() -> new Object();
function GrandParent() {
    this.name = 'GrandParent';
    this.a = 3;
}
Parent.prototype = new GrandParent();
function Parent() {
    this.name = 'parent';
    this.b = 2;
}
Child.prototype = new Parent();
function Child() {
    this.name = 'child';
    this.c = 1;
}

var child = new Child();
console.log(child); // Child {name: "child", c: 1}

console.log(child.a); // 3
console.log(child.b); //2
console.log(child.c); //1
```

简单总结原型链：

```
一、构造函数、原型和实例的关系
    a,构造函数都有一个属性prototype，这个属性值是一个对象，是Object的实例；
    b,原型对象prototype里有一个constructor属性，该属性指向原型对象所属的构造函数；
    c,实例对象都有一个__proto__属性，该属性指向构造函数的原型对象；
二、prototype与_proto_的关系
    a,prototype是构造函数的属性；
    b,__proto__是实例对象的属性；
    c,两者都指向同一个对象（原型）；
```

原型链作为继承的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

每个构造函数都有一个原型对象prototype，原型对象都包含一个指向构造函数的指针constructor，而实例都包含一个指向原型对象的内部指针 [[prototype]]。

如果让原型对象等于另一个类型的实例，那么原型对象将包含一个指向另一个原型的指针，相应地，另一个原型中也包含着一个指向另一个构造函数的指针。

```
function Person() {
    this.getName=function () {
          return this.name;
    }
}

function Student(name) {
   this.name=name;
}

Student.prototype=new Person();
var xiaoGane =new Student("小刚");
console.log(xiaoGane.getName());
```

通过实现原型链，本质上扩展了原型搜索机制，当以读取模式访问一个实例属性时，首先会在实例中搜索该属性。如果没有找到该属性，则会继续搜索实例的原型。在通过原型链实现继承的情况下，搜索过程就得以沿着原型链继续向上。

7、new

使用new创建对象时，发生了什么：

- 创建新对象，调用构造函数
- 将函数的上下文对象（作用域）中的this指向了该对象.。谁创建对象，this就表示谁
- 执行构造函数中的代码，通过this来添加属性和方法
- 返回对象

```
function Person() {
    this.name = "张三",
    this.say = function(word) {
        console.log(word);
    }
}

var p = new Person();
var p2 = new Person();
```

内置对象

1、Number

Number 对象是原始数值的包装对象。创建方式 **new Number()**。

（1）属性

- **prototype**：允许向对象添加属性和方法
- **constructor**：返回对创建此对象的 Number 函数的引用。
- **NaN**：非数字值。

- **MAX_VALUE**：可表示的最大的数。
- **MIN_VALUE** ：可表示最小的数

ES6新增的属性：

- **EPSILON**: 表示 1 和比最接近 1 且大于 1 的最小 Number 之间的差别
- **MIN_SAFE_INTEGER**: 表示在 JavaScript中最小的安全的 integer 型数字 (`-(2^53 - 1)`)。
- **MAX_SAFE_INTEGER**: 表示在 JavaScript 中最大的安全整数 (`2^53 - 1`) 。

```
console.log(Number.MAX_VALUE); //1.7976931348623157e+308
console.log(Number.MIN_VALUE); //5e-324
console.log(Number.NaN); //NaN

console.log(Number.EPSILON); //2.220446049250313e-16
console.log(Number.MIN_SAFE_INTEGER); //-9007199254740991
console.log(Number.MAX_SAFE_INTEGER); //9007199254740991

var num = new Number(123);
console.log(num.constructor); //ƒ Number() { [native code] }


Number.prototype.getPart = function() {
    this.myProp = this.valueOf()/2;
}
var num = new Number(20);
num.getPart();
console.log(num.myProp); //10
```

（2）方法

- **toString()**：把数字转换为字符串，使用指定的基数。
- **valueOf()**：返回一个 Number 对象的基本数字值。

ES6新增的方法

- **Number.isInteger()**: 用来判断给定的参数是否为整数。
- **Number.isSafeInteger()**: 判断传入的参数值是否是一个"安全整数"。安全整数范围为 `-(2^53 - 1)到 2^53 - 1之间的整数，包含 -(2^53 - 1)和 2^53 - 1`。

```
var num = new Number(255);
console.log(num.toString(16)); //77
console.log(num.toString(8)); //377
console.log(num.toString(2)); //1111 1111
console.log(num.valueOf()); 255

Number.isInteger(10);        // 返回 true
Number.isInteger(10.5);      // 返回 false

Number.isSafeInteger(10);    // 返回 true
Number.isSafeInteger(12345678901234567890);  // 返回 false
```

2、String

String对象中有一些属性和函数可以用来处理字符串。

（1）属性

- **length**：字符串的长度
- **prototype**：允许您向对象添加属性和方法
- **constructor**：对创建该对象的函数的引用

```
var txt = "hello world";
console.log(txt.length); //11
console.log(txt.constructor); //ƒ String() { [native code] }

function Person(name, job) {
    this.name = name;
    this.job = job;
}
var p = new Person("张三", "JavaScript");
Person.prototype.salary = null; //给对象添加属性salary
console.log(p.salary); //null
p.salary = 5000;
console.log(p.salary);
```

（2）方法

- **charAt()**：返回指定位置的字符。
- **indexOf()**：返回某个指定的字符串值在字符串中首次出现的位置。
- **concat()**：连接两个或更多字符串，并返回新的字符串。
- **match()**：查找找到一个或多个正则表达式的匹配。
- **replace()**：在字符串中查找匹配的子串，并替换与正则表达式匹配的子串。
- **split()**：把字符串分割为字符串数组。
- **substring()**：提取字符串中两个指定的索引号之间的字符。
- **valueOf()**：返回某个字符串对象的原始值。

```
var str = "hello java script";
console.log(str.charAt(4)); //o
console.log(str.indexOf("j")); //6
console.log(str.substring(2, 5)); //llo
console.log(str.match("s")); //["s", index: 11, input: "hello java script",
groups: undefined]
console.log(str.replace(" ", ", ")); //hello, java script
console.log(str.split(" ")); //(3) ["hello", "java", "script"]
console.log(str.valueOf()); //hello java script

var str2 = "你好啊"
console.log(str.concat(str2)); //hello java script你好啊
```

3、Date

用于处理时间和日期。

创建Date对象的四种方式：

- **var d = new Date();** 以当前时间创建对象
- **var d = new Date(milliseconds);** //返回从 1970 年 1 月 1 日至今的毫秒数
- **var d = new Date(dateString);**
- **var d = new Date(year, month, day, hours, minutes, seconds, milliseconds);**

```
var a = new Date();
console.log(a);

var b = new Date("5 13 2018 11:13:00"); //月 日 年  时：分：秒
console.log(b)

var c = new Date("2019-4-1 08:08:50");
console.log(c);

var d = new Date(2019, 4, 1, 12, 20, 30);
console.log(d);
```

（1）属性

- **prototype**：使您有能力向对象添加属性和方法。
- **constructor**：返回对创建此对象的 Date 函数的引用。

```
var date = new Date();
console.log(date.constructor); //ƒ Date() { [native code] }


Date.prototype.myMet = function() {
    if(this.getMonth() == 0) { this.myProp = "January" };
    if(this.getMonth() == 1) { this.myProp = "February" };
    if(this.getMonth() == 2) { this.myProp = "March" };
    if(this.getMonth() == 3) { this.myProp = "April" };
    if(this.getMonth() == 4) { this.myProp = "May" };
    if(this.getMonth() == 5) { this.myProp = "June" };
    if(this.getMonth() == 6) { this.myProp = "July" };
    if(this.getMonth() == 7) { this.myProp = "August" };
    if(this.getMonth() == 8) { this.myProp = "Spetember" };
    if(this.getMonth() == 9) { this.myProp = "October" };
    if(this.getMonth() == 10) { this.myProp = "November" };
    if(this.getMonth() == 11) { this.myProp = "December" };
}
var d = new Date();
d.myMet();
console.log(d.myProp);
```

（2）方法

- **getFullYear()**：从 Date 对象以四位数字返回年份。
- **getMonth()**：从 Date 对象返回月份 (0 ~ 11)。
- **getDate()**：从 Date 对象返回一个月中的某一天 (1 ~ 31)。
- **getDay()**：从 Date 对象返回一周中的某一天 (0 ~ 6)。
- **getHours()**：返回 Date 对象的小时 (0 ~ 23)。

- **getMinutes()**：返回 Date 对象的分钟 (0 ~ 59)。
- **getSeconds()**：返回 Date 对象的秒数 (0 ~ 59)。
- **getTimezoneOffset()**：返回本地时间与格林威治标准时间 (GMT) 的分钟差。
- **getTime()**：返回1970年1月1日至今的毫秒数

```
var d = new Date();
console.log(d.getFullYear(), d.getMonth(), d.getDate(), d.getDay());
console.log(d.getHours(), d.getMinutes(), d.getSeconds());
```

4、Array

（1）属性

- **length**：获得数组的长度
- **constructor**：返回创建数组对象的原型函数。
- **prototype**：向数组对象添加属性或方法。

```
var arr = ["abc", "hello", "apple"];
console.log(arr.length); //3

Array.prototype.myUpper = function() {
    for(var i = 0; i < this.length; i++) {
        this[i] = this[i].toUpperCase();
    }
}
arr.myUpper();
console.log(arr); //(3) ["ABC", "HELLO", "APPLE"]

console.log(arr.constructor); //ƒ Array() { [native code] }
```

（2）方法

- **concat()**：连接两个或更多的数组，并返回结果。
- **join()**：把数组的所有元素放入一个字符串。
- **pop()**：删除数组的最后一个元素并返回删除的元素。
- **push()**：向数组的末尾添加一个或更多元素，并返回新的长度。
- **slice()**：选取数组的的一部分，并返回一个新数组。
- **splice()**：从数组中添加或删除元素。
- **sort()**：对数组的元素进行排序。
- **reverse()**：反转数组的元素顺序。

```
var arr1 = [1,2,3];
var arr2 = [6,5,4];
console.log(arr1.concat(arr2)); //(6) [1, 2, 3, 6, 5, 4]

var arr = [3, 6, 2, 9, 4];
console.log(arr.join("-")); //3-6-2-9-4

var arr = ["aa", "qq", "yy"];
arr.pop();
console.log(arr); //(2) ["aa", "qq"]
```

```
arr.push("www");
console.log(arr); //(3) ["aa", "qq", "www"]

console.log(arr.slice(1, 2)); //["qq"]

arr.splice(1, 2)
console.log(arr); //(2) ["aa", "www"]

var arr = [5,2,7,9,1];
arr.sort();
console.log(arr); //(5) [1, 2, 5, 7, 9]

arr.reverse();
console.log(arr); //(5) [9, 7, 5, 2, 1]
```

5、Boolean

Boolean 对象用于转换一个不是 Boolean 类型的值转换为 Boolean 类型值 (true 或者false).

（1）属性

- **constructor**：返回对创建此对象的 Boolean 函数的引用
- **prototype**：使您有能力向对象添加属性和方法。

```
var bool = new Boolean(1);
console.log(bool.constructor); //ƒ Boolean() { [native code] }

Boolean.prototype.myColor = function() {
    if(this.valueOf() == true) {
        this.color = "green";
    } else {
        this.color = "red";
    }
}
bool.myColor();
console.log(bool.color); //green
```

（2）方法

- **toString()**：把布尔值转换为字符串，并返回结果。
- **valueOf()**：返回 Boolean 对象的原始值。

```
var b = new Boolean("abc");
console.log(b.toString()); //true
console.log(b.valueOf()); //true
```

6、Math

Math 对象用于执行数学任务。Math没有构造函数，因此不能使用new来创建对象

（1）属性

- **E**：返回算术常量 e，即自然对数的底数（约等于2.718）。
- **PI**：返回圆周率（约等于3.14159）。

```
console.log(Math.E); //2.718281828459045
console.log(Math.PI); //3.141592653589793
```

（2）方法

- **random()**：返回 0 ~ 1 之间的随机数。
- **floor(x)**：对 x 进行向下取整。
- **ceil(x)**：对 x 进行向上取整。
- **pow(x, y)**：返回 x 的 y 次幂。
- **sqrt(x)**：返回x的平方根

```
console.log(Math.floor(3.14)); //3
console.log(Math.ceil(5.5)); //6
console.log(Math.pow(2, 3)); //8
console.log(Math.sqrt(16)); //4

console.log(Math.random()); //获取0~1之间的随机浮点数
console.log(Math.random()*10);  //获取0~10之间的随机浮点数
console.log(parseInt(Math.random()*10)); //获取0~10之间的随机整数
console.log(10 + parseInt(Math.random()*10)); //获取10~20之间的随机整数
```

函数

函数对任何一门语言来说都是核心的概念。通过函数可以封装任意多条语句，而且可以在任何地方、任何时候调用执行。在javascript里，函数即对象，程序可以随意操控它们。函数可以嵌套在其他函数中定义，这样它们就可以访问它们被定义时所处的作用域中的任何变量，它给javascript带来了非常强劲的编程能力。

- 一处定义，处处调用；
- 如果把函数作为一个对象的属性，则称为方法；
- 每次调用函数会产生一个this：谁调用这个函数或者方法，this就指向谁；
- 函数就是对象，可以给他设置属性或方法；

1、函数定义

总共有三种函数定义的方式：函数声明语句、函数表达式、构造函数。

（1）函数声明语句

```
function functionName(parameters) {
    //执行的代码
}
```

函数声明后不会立即执行，会在我们需要的时候调用到。

示例1：计算两个坐标点之间的距离。

```
function distance (x1, y1, x2, y2) {
    var x = x2 - x1;
    var y = y2 - y1;
    return Math.sqrt(x*x + y*y).toFixed(2);
}
console.log(distance(3, 5, 8, 20));
```

示例2：定义一个求阶乘的函数。

```
//求阶乘n! 即，1×2×3×4×5×……×n（使用递归算法）
function loop(x) {
    if(x < 2) {
        return 1;
    } else {
         return x * loop(x - 1);
    }
}
//调用函数
var result = loop(5);
console.log(result); //120
```

（2）函数表达式

```
var functionName = function(parameters) {
    //执行的代码
};
//函数以分号结尾，因为它实际上是一个执行语句
```

以上函数实际上是一个 **匿名函数** (函数没有名称)。函数存储在变量中，不需要函数名称，通常通过变量名来调用。

```
var square = function(x) {
    if(x === 1) {
        return 1;
    } else {
        return x * square(x - 1);
    }
}; //函数表达式是一个语句，因此结尾可以加上分号
console.log(square(3)); //6
```

（3）Function()构造函数

在以上实例中，函数通过关键字 **function** 定义。

函数同样可以通过内置的 JavaScript 函数构造器（Function()）定义。

```
var functionName  = new Function("parameters","执行的代码");
//注意引号不可省略
```

示例1：

```
var test = new Function("a", "b", "return a * b");

var x = test(4, 3);
console.log(x); //12
```

参数和返回值都需要使用引号。

示例2：

```
var y = "global";

function myFunc() {
    var y = "local";

    return new Function("return y"); // 无法获取局部变量,作用域始终是全局作用域
}
console.log(myFunc()()); // 执行子函数，输出 “global”
```

用Function()构造函数创建一个函数时并不遵循典型的作用域，它一直把它当作是顶级函数来执行。所以，在 JavaScript 中，很多时候，你需要避免使用 **new** 关键字。**使用构造函数无法实现递归。**

示例3：

```
function distance(x1, y1, x2, y2) {
    var h = 10;

    function square(h) {
        return h * h;
    }
    return Math.sqrt(square(x2 - x1) + square(y2 - y1)).toFixed(2);
}
console.log(distance(1,5,2,4)); //1.41
```

函数可以嵌套在其他函数里面，也就是在函数里面可以定义函数。

内嵌的函数可以访问外部函数的变量和参数。

参数和函数内部使用var声明的变量都是局部变量，拥有局部作用域

（4）函数提升

提升（Hoisting）是 JavaScript 默认将当前作用域提升到前面去的的行为；

函数可以在声明之前调用。**使用表达式定义的函数无法提升。**

```
//调用函数
myFunction(5);

//定义函数
function myFunction(y) {
    return y * y;
}
```

（5）自调用函数

- 函数表达式可以 "**自调用**"。
- 如果表达式后面紧跟 () ，则会自动调用。
- 不能自调用声明的函数。
- 对于函数自调用，必须通过**在函数表达式外面添加括号**(来说明它是一个函数表达式)再调用，否则会报错

```
(function () {
    var x = "Hello!!";      // 我将调用自己
})();
```

以上函数实际上是一个 **匿名自我调用的函数** (没有函数名)

- 如果把函数表达式赋给一个变量则不需要添加括号也可以直接调用：

```
var a = function () {
    document.write("Hello! 我是自己调用的" + "<br />");
    return '返回的东西';
}();  // 因为函数自调了，变量a不再指向一个函数，而是“返回的东西”这个字符串

document.write(a);  //输出字符串



var add = (function() {
    var counter = 0;
    return function() {return counter += 1;}  //这里return了一个内嵌方法，即add指向一个方法

})();
console.log(add()); //1
```

- 函数名后的多个括号

```
f()意思是执行f函数，返回子函数

f()()执行子函数，返回孙函数

f()()()执行孙函数
```

需注意，如果想这样执行，函数结构必须是这样，f的函数体里要return 子函数，子函数里要return 孙函数，如果没有return关键字，是不能这样连续执行的，会报错。

2、函数调用

javascript一共有4种调用模式：函数调用模式、方法调用模式、构造器调用模式和间接调用模式。

每种方式的不同在于 **this** 的初始化。

（1）函数调用模式

```
var myfunc = function(a,b){
    console.log(this); //Window
    return a+b;
}

alert(myfunc(3,4)); //7
```

函数调用模式中：

- this是指向Window的
- 返回值是由return语句决定的，如果没有return则表示没有返回值

（2）方法调用模式

先定义一个对象，然后在对象的属性中定义方法，通过myobject.property来执行方法。

```
var name = "张三";
var obj = {
    name: "李四",
    getName: function() {
        console.log(this); //{name: "李四", getName: ƒ}
        console.log(this.name); //李四
    }
};
obj.getName();
```

方法调用模式中：

- this 是指向调用该方法的对象
- 返回值还是由return语句决定，如果没有return表示没有返回值

（3）构造函数调用模式

如果函数或者方法调用之前带有关键字new，它就当成构造函数调用。

```
function Person() {
    this.name = "james";
    this.age = 32;
    console.log(this);
};
var person = new Person();    // 在调用这段代码的时候，输出的是 Person{name: "james", age: 32}
console.log(person);      // 同样输出的是Person{name: "james", age: 32}
```

通过上面的代码结果分析，会得到以下结论（构造函数调用模式中）：

- this是指向构造函数的实例
- 如果没有添加返回值的话，默认的返回值是this

但是如果手动添加返回值之后呢？

```
function Fn1 () {
   this.name = "james";
   return "wade" ;           //定义返回一个字符串
};
var fn1 = new Fn1();         //仍然是接收返回的this对象
console.log(fn1);            //Fn1 {name: "james"}
console.log(fn1.name);       // 这段代码输出的是 james;


function Fn2 () {
   this.name = "james";
   return [1,2,3];           //定义返回一个数组
};
var fn2 = new Fn2();         //fn2接收的是返回的数组
console.log(fn2);            //[1, 2, 3]
console.log(fn2.name);       // 而这段代码输出的是undefined
```

通过上面的代码结果分析，优化上面的结论：

- this是指向构造函数的实例
- 如果没有添加返回值的话，默认的返回值是this
- 如果有返回值，且返回值是简单数据类型（Number,String,Boolean··）的话，最后仍回返回this
- 如果有返回值，且返回值是复杂数据类型（对象）的话，最终返回该对象，所以上面的fn2是指向数组，所以fn2.name为undefined

（4）间接调用模式

也称之为“**apply、call调用模式**”或“**上下文调用模式**”。

每个函数都包含两个非继承而来的方法：**call()**方法和**apply()**方法。这两个方法的作用都是一样的。都是在特定的作用域中调用函数，想当于设置函数体内this对象的值，以扩充函数赖以运行的作用域。

```
var myobject={};
var getSum = function(a,b){
    return a+b;
};
var sum1 = getSum.call(myobject, 10, 30); //call是调用函数的同时传实例对象+几个实参
console.log(sum1);

var sum2 = getSum.apply(myobject,[10,30]); //apply是调用函数的同时传实例对象+[几个实参]
console.log(sum2);
```

由之前所学，this指向由传入的第一个参数决定。

再看看，下面函数调用中this指向如何呢？

```
function f1(){
   console.log(this);
}
f1.call(null);          // Window
f1.call(undefined);     // Window
f1.call(123);          // Number的实例
f1.call("abc");        // String的实例
f1.call(true);         // Boolean的实例
f1.call([1,2,3]);      // Array的实例
```

通过上面的代码结果分析，得出以下结论（上下文调用模式中）：

- 传递的参数不同，this的指向不同，this会指向传入参数的数据类型
- 返回值是由return决定，如果没有return表示没有返回值。

3、函数参数

javascript函数的参数与大多数其他语言的函数的参数有所不同。函数不介意传递进来多少个参数，也不在乎传进来的参数是什么数据类型，甚至可以不传参数。

```
function add(x){
    return x+1;
}
console.log(add());//NaN
```

（1）同名形参

在非严格模式下，函数中可以出现同名形参，且只能访问最后出现的该名称的形参。

```
function add(x,x,x){
    return x;
}
console.log(add(1,2,3));//3
```

在严格模式下，出现同名形参会抛出语法错误。

（2）参数个数

当实参比函数声明指定的形参个数要少，剩下的形参都将设置为undefined值。当实参多于形参，则只使用有效的实参，多出部分没影响。

```
function add(x,y){
    console.log(x,y);//1 undefined
}
add(1);
```

（3）给形参设置默认值

在定义函数的时候，可以使用=直接给形参设置一个默认值，需要设置默认值的形参放在最后面；

```
function add(x, y = 10) {
    return x + y;
}
console.log(add(1)); //11
```

（4）显式参数(Parameters)

函数显式参数在函数定义时列出（即形参）。

```
function functionName(param1, param2, param3) {
  // 要执行的代码……
}
```

函数调用未传参时，参数会默认设置为： undefined。有时这是可以接受的，但是建议最好为参数设置一个默认值：

```
function myFunction(x, y) {
  if (y === undefined) {
    y = 0;
  }
}
```

或者，更简单的方式：

```
function myFunction(x, y) {
  y = y || 0;
}
```

（5）隐式参数(Arguments)

JavaScript 函数有个内置的对象 arguments 对象。argument 对象包含了函数调用的参数数组（实参数组）。

```
function show() {
    console.log(arguments.length);
}
show("a", "b"); //2
show(1,2,3); //3
```

通过这种方式你可以很方便的找到最大的一个参数的值

```
function findMax() {
    var max = arguments[0];
    if(arguments.length < 2) return max;
    for(var i = 0; i < arguments.length; i++) {
        if(arguments[i] > max) {
            max = arguments[i];
        }
    }
    return max;
}

x = findMax(1, 123, 500, 115, 44, 88);
console.log(x); //返回传入参数的最大值500
```

练习：求传入参数的累加和。

arguments对象与传入参数的映射规则：

```
function sum(a,b){
  arguments[1]=4;

  console.log(arguments[0], arguments[1], b);
}
sum(1); //1 4 undefined
sum(1,2); //1 4 4

//arguments对象与形参是相互独立的，但又存在映射规则：
//当传入参数与形参个数相等时，arguments对象与形参才是一一对应的；
//当传入参数与形参个数不等时，arguments对象与有传入参数的形参才存在映射规则。
```

练习一道阿里巴巴2013年的一道笔试题：

```
下面代码中console.log的结果是[1,2,3,4,5]的选项是（）ACD

//A
function foo(x){
    console.log(arguments)
    return x;
}

foo(1,2,3,4,5)

//B
function foo(x){
    console.log(arguments)
    return x;
}
(1,2,3,4,5)

//C
(function foo(x){
    console.log(arguments)
    return x;
})(1,2,3,4,5)

//D
```

```
function foo(){
    bar.apply(null,arguments);
}
function bar(x){
    console.log(arguments);
}
foo(1,2,3,4,5)
```

4、闭包

官方对闭包的解释是：一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

```
面试题：闭包是什么？
答：闭包是可访问上一层函数作用域里变量的函数，即便上一层函数已经关闭。
```

示例1：

```
function a() {
    var num=100;
    function b() {
        num++;
        console.log(num);
    }
    return b;
}
var demo=a();
demo();
demo();
```

示例2：再来玩玩闭包

```
function fun(n, o) {
    console.log(o);//undefined   0  0
    return {
        fun: function (m) {
                return fun(m, n);
        }
    };
}

var a = fun(0);//打印什么？

a.fun(1);
a.fun(2);
a.fun(3);

var b = fun(0).fun(1).fun(2).fun(3);

var c = fun(0).fun(1);
```

```
c.fun(2);
c.fun(3);
```

示例3：

```
function Person(name,age,sex){
    var a=0;
    this.name=name;
    this.sex=sex;
    this.say= function() {
        a++;
        document.write(a);
    };
}

var oPerson = new Person();
oPerson.say();
oPerson.say();
var oPerson1= new Person();
oPerson1.say();
```

（1）闭包的特点

- 作为一个函数变量的一个引用，当函数返回时，其处于激活状态。
- 一个闭包就是当一个函数返回时，一个没有释放资源的栈区。

简单的说，Javascript允许使用内部函数---即函数定义和函数表达式位于另一个函数的函数体内。而且，这些内部函数可以访问它们所在的外部函数中声明的所有局部变量、参数和声明的其他内部函数。当其中一个这样的内部函数在包含它们的外部函数之外被调用时，就会形成闭包。

优点：可以访问局部变量。

缺点：局部变量一直占用内存，内存占用严重，还容易造成内存泄漏（内存被占用，剩余的内存变少，程序加载、处理速度变慢）。

（2）闭包的几种写法

- 写在原型对象的方法上

```
function Person() {
}

Person.prototype.type="人类";
Person.prototype.getType=function() {
     return this.type;
}

var person = new Person();
console.log(person.getType());
```

- 内部函数语句访问外部函数的变量，将内部函数写在外部函数的return中

```
var Circle = function() {
   var obj = new Object();
   obj.PI = 3.14159;

   obj.area = function( r ) {
       return this.PI * r * r;
       //this访问了外部函数的变量obj
   }
   return obj;
}

var c = new Circle();
alert( c.area( 1.0 ) );
```

- 通过表达式写在对象的方法上

```
var circle = new Object();
circle.PI = 3.14159;
circle.area = function( r ) {
       return this.PI * r * r;
}

alert( circle.area( 1.0 ) );
```

- 通过属性创建写在对象的方法上

```
var Circle={
   "PI":3.14159,
   "area":function(r){
          return this.PI * r * r;
    }
};
alert( Circle.area(1.0) );
```

- 通过全局变量赋值(类似于第二种写法的原理)

```
var demo;
function test(){
    var aaa=100;
    function b(){
        console.log(aaa)
    }
    //return b;
    demo=b;
}
//demo=test()
demo();
demo();
```

（3）闭包的用途

- **实现公有变量**

  示例：函数累加器

```
function add() {
    var counter = 0;
    return counter += 1;
}

add();
add();
add();
// 本意是想输出 3，但事与愿违，输出的都是 1
```

  你可以使用全局变量，函数设置计数器递增：

```
var counter = 0;

function add() {
   return counter += 1;
}

add();
add();
add();
// 计数器现在为 3
```

  但问题来了，页面上的任何脚本都能改变计数器，即便没有调用 add() 函数。

  这时我们需要闭包。

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();

add();
add();
add();

// 计数器为 3
```

- **可以做缓存**

  示例：

```
function eater() {
    var food="";
    var obj={
        eat: function(){
            console.log("i am eating"+food);
            food="";
        },
        push: function(myfood){
```

```
            food=myfood;
        }
    }
    return obj;
}
var eater1 = eater();
eater1.push("banana")
eater1.eat();//打印i am eating banana

eater2=eater();
eater2.push("apple");
eater2.eat();
```

暂时不必要理解"缓存"代表的是什么，先把这种闭包调用过程理解了就行（多个方法都可以在外部对同一个局部变量进行操作），以后讲到闭包高级应用的时候会再提出来的。

- **可以实现封装，属性私有化**

  示例：

```
var person = function(){
    //变量作用域为函数内部，外部无法访问
    var name = "default";

    return {
       getName : function(){
           return name;
       },
       setName : function(newName){
           name = newName;
       }
    }
}();

print(person.name);//直接访问，结果为undefined
print(person.getName()); //default
person.setName("abruzzi");//   abruzzi
print(person.getName());    //abruzzi
```

- **模块化开发，防止污染全局变量**

```
var a = (function(j) {
    return function() { console.log(j) }
    //模块内的函数变量不会被外部函数所污染，且执行完立即被销毁，不会浪费空间
})(10);
a();
```

- **实现类和继承**

```
function Person(){
    var name = "default";
```

```
    return {
       getName : function(){
           return name;
       },
       setName : function(newName){
           name = newName;
       }
    }
};

var p = new Person();
p.setName("Tom");
alert(p.getName());//Tom

var Jack = function(){};
//修改Jack这个构造函数的属性prototype的引用，让其构造对象可以继承自Person
Jack.prototype = new Person();
//添加私有方法
Jack.prototype.Say = function(){
    alert("Hello,my name is Jack");
};
var j = new Jack();
j.setName("Jack");
j.Say();
alert(j.getName());
```

（4）闭包使用注意点

- 滥用闭包，会造成内存泄漏：由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄漏。解决方法是，在退出函数之前，将不使用的局部变量全部删除。
- 会改变父函数内部变量的值。所以，如果你把父函数当做对象（object）使用，把闭包当做它的公有方法（Public Method），把内部变量当做它的私有属性（Private Value）。这是一定要小心，不要随便改变父函数内部变量的值。

5、高阶函数

高阶函数是指操作函数的函数，一般地，有以下两种情况

- 函数可以**作为参数**被传递
- 函数可以**作为返回值**输出

（1）参数传递

```
function addnum (a, b, fn) {
    return fn(a) + fn(b);
}

var a = addnum(20, -30, Math.abs());//50
```

（2）返回值输出

下面是使用Object.prototype.toString方法判断数据类型的三个isType函数：

```
var isString = function( obj ){
  return Object.prototype.toString.call( obj ) === '[object String]';
};

var isArray = function( obj ){
  return Object.prototype.toString.call( obj ) === '[object Array]';
};

var isNumber = function( obj ){
  return Object.prototype.toString.call( obj ) === '[object Number]';
};
```

实际上，这些函数的大部分实现都是相同的，不同的只是Object.prototype.toString.call(obj)返回的字符串。为了避免多余的代码，可以把这些字符串作为参数提前传入isType函数。代码如下：

```
var isType = function( type ){
  return function( obj ){
    return Object.prototype.toString.call( obj ) === '[object '+ type +']';
  }
};

var isString = isType( 'String' );
var isArray = isType( 'Array' );
var isNumber = isType( 'Number' );

console.log( isArray( [ 1, 2, 3 ] ) );    // 输出：true
```

复杂点的高阶函数：

```
//检查一个值是否为偶数
function even (a) {
    return a%2 === 0;   //返回true或false
}
//对函数判断进行反转
function not (f) {
    return function () {
        var result = f.apply(this, arguments);
        console.log(!result);//这里会打印什么那？我很好奇
        return !result;
    }
}

var odd = not(even);
console.log(odd);
/*odd变量所存储的函数如下：
function () {
    var result = even.apply(this, arguments);
    console.log(!result);//这里会打印什么那？我很好奇：true true true false
```

```
    return !result;
}
*/
var arr = [1,3,5,4];
var r1 = arr.every(odd);//false
var r2 = arr.some(odd);//true
console.log(r1);
console.log(r2);
```

call()与apply()

call()允许为不同的对象分配和调用属于一个对象的函数/方法。`call()`提供新的 **this** 值给当前调用的函数/方法。

call()和 **apply()** 方法类似，只有一个区别，就是`call()`方法接受的是**若干个参数的列表**，而`apply()`方法接受的是**一个包含多个参数的数组**。

（1）语法

```
fun.call(thisArg, arg1, arg2, ...)
```

- **thisArg**：在*fun*函数运行时指定的`this`值。需要注意的是，指定的`this`值并不一定是该函数执行时真正的`this`值，如果这个函数处于`non-strict mode`，则指定为`null`和`undefined`的`this`值会自动指向全局对象(浏览器中就是window对象)，同时值为原始值(数字，字符串，布尔值)的`this`会指向该原始值的自动包装对象。
- **arg1, arg2, ...**：指定fun函数的参数列表。
- **返回值**：使用调用者提供的`this`值和参数调用该函数的返回值。若该方法没有返回值，则返回`undefined`。

（2）示例

使用`call`方法调用函数并且指定上下文的`this`

```
function greet() {
  var reply = this.name + "--" + this.age;
  console.log(reply);
}

var obj = {
    name: '张三',
    age: '20'
};

greet.call(obj);
```

JavaScript 计时事件

-

正则表达式

1、概念及用途

正则表达式（英语：Regular Expression，在代码中常简写为regex、regexp或RE），使用单个字符串来描述、匹配一系列符合某个句法规则的字符串搜索模式。

- 正则表达式是由一个字符序列形成的搜索模式。
- 当你在文本中搜索数据时，你可以用搜索模式来描述你要查询的内容。
- 正则表达式可以是一个简单的字符，或一个更复杂的模式。
- 正则表达式可用于所有文本搜索和文本替换的操作。

2、语法规则

（1）创建方法

- **直接量**
  - 语法：

```
/正则表达式主体/修饰符(可选)

1. 正则表达式的主体放在//之间

2. 修饰符：可选
    - i：执行对大小写不敏感的匹配。
    - g：执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
    - m：执行多行匹配。
```

  - 示例1：

```
var  reg=/abc/;
var  str="abcd";
reg.test(str);  //true，检查在字符串str中有没有符合reg规则的字符
```

  - 示例2：

```
var  reg=/abce/i;
var  str="ABCEd";
reg.test(str); //true
```

- **构造方法RegExp()**

```
//使用new操作符，new RegExp();
var reg = new RegExp("abc");
var str = "abcd";
reg.test(str);

//在new RegExp("abc")函数里边也可以添加属性i、g、m
var reg=new RegExp("abc","im");
var str="abcd";
reg.test(str);
```

使用new操作符，可以将已经存在的正则表达式用来给函数RegExp()传参，构造新的正则表达式.

```
//reg与reg1值相同，但两个两个值相互独立,即reg!=reg1
var reg = /abce/m;
var reg1 = new RegExp(reg);
console.log(reg == reg1);
console.log(reg === reg1);
```

若去除new操作符，将已经存在的正则表达式用来给函数RegExp()传参，只是传递引用，不能构建新的正则表达式，极少的用法。

```
var reg = /abce/m;
var reg1 = RegExp(reg);
console.log(reg == reg1); //true
console.log(reg === reg1); //true
reg.abc = 3;
console.log(reg1.abc); //3
```

（2）三个属性i，g，m

正则表达式的属性（也称修饰符），可以在全局搜索中不区分大小写：

修饰符	描述
i	ignoreCase，执行对大小写不敏感的匹配。
g	global，执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	multiline，执行多行匹配。

```
var reg = /ab/;
var str = "abababababab";
str.match(reg); //["ab"]，只查找到第一个匹配值便返回
reg = /ab/g;
str.match(reg); //["ab","ab","ab","ab","ab"]，全局查找，把所有匹配值均返回
```

```
var reg = /a/g;
var str = "abcdea";
str.match(reg); //["a","a"]
reg = /^a/g; //插入符^指的是以字母a为开头
str.match(reg); //["a"]

str = "abcde\na";
str.match(reg); //["a"]，还没有多行匹配属性
reg = /^a/gm;
str.match(reg); //["a","a"]
```

（3）方括号

方括号用于查找某个范围内的字符：

表达式	描述
[abc]	查找方括号之间的任何字符。
[0-9]	查找任何从 0 至 9 的数字。
(x|y)	查找任何以 | 分隔的选项。注意 | 前后不要乱加空格，空格也是规则

- 一个**中括号**代表一位，中括号里边的内容代表的是这一位可以取值的范围
  - 示例1：

```
var  reg=/[1234567890][1234567890][1234567890]/g;
var  str="12309u98723zpoixcuypiouqwer";
str.match(reg);//["123","987"]
```

  - 示例2：

```
var reg=/[ab][cd][d]/g;
var str="abcd";
str.match(reg);//["bcd"];
```

  - 示例3：

```
var reg=/[0-9A-Za-z][cd][d]/g;//相当于var reg=/[0-9A-z][cd][d]/g
var str="ab1cd";
str.match(reg);//["1cd"];
```

- 插入符 ^ 放到 [ ] 里边表示"**非**"的意思

```
var reg=/[^a][^b]/g;//插入符^放到[]里边表示"非"的意思
var str="ab1cd";
str.match(reg);//["b1","cd"];
```

- 在**圆括号**里可以加入"|"表示"**或**"的意思，"|"操作符两边放匹配规则

```
var reg=/(abc|bcd)/g;
var str="abc";
str.match(reg);//["abc"];//该规则既能匹配出字符串"abc"

str="bcd";
str.match(reg);//["bcd"];//该规则又能匹配出字符串"bcd"

reg=/(abc|bcd)[0-9]/g;  //匹配规则可以任意组合
str="bcd2";
str.match(reg);//["bcd2"]
```

（4）元字符

元字符是拥有特殊含义的字符，元字符也可以组合放进中括号里去使用，一个元字符代表一位。

元字符	描述
\w	查找单词字符(字母+数字+下划线)， [0-9A-z]
\W	查找非单词字符
\d	查找数字，[0-9]
\D	查找非数字字符
\s	查找空白字符，包括空格符[ ]、制表符\t、回车符\r、换行符\n、垂直换行符\v、换页符\f
\S	查找非空白字符
\b	匹配单词边界
\B	匹配非单词边界
\t	查找制表符
\n	查找换行符
\f	查找换页符
\v	查找垂直制表符
\uXXXX	查找以十六进制规定的Unicode字符，\u4e00~\u9fa5
.	（点号）查找单个字符，除了换行和行结束符

- **"\w"---->[0-9A-z_]**，(word)单词字符，字母字符

```
var reg=/\wcd2/g;
var str="bcd2";
str.match(reg);//["bcd2"]

//"\W"---->[^\w]，即\W是\w的补集
reg=/\wcd2/g;//匹配规则换成\W就不能匹配了，因为字符串里的字符均为\w所包括的字符
str.match(reg);//null

str="b*cd2";//这样"*"就能符合规则\W了
str.match(reg);//["*cd2"]

//若要在正则表达式里匹配反斜杠\，直接写/\/是不行的，需要加上转义字符/\\/
```

- **"\d"---->[0-9]**

```
//"\D"---->[^\d]
var reg=/\d\d\d/g;
var str="123";
str.match(reg);//["123"]
```

- **"\b"---->匹配单词边界（border）**

```
var reg=/\bcde/g;//单词边界后边是cde字符串
var str="abc cde fgh";
str.match(reg);//["cde"]

str="abc cde fgh";
reg=/\bcde\b/g;//该规则也能够匹配出来
str.match(reg);//["cde"]

reg=/\bcde\b/g;
str="abc cdefgh";//这种字符串就不能匹配出来了
str.match(reg);//null

str="abc cdefgh";
reg=/\bcde\B/g;//匹配规则换成\B就可以了
str.match(reg);//["cde"]
```

- **"\t"---->匹配制表符**

```
var reg=/\tc/g;
var str="ab cde";//c字母前有一个tab键
str.match(reg);//null，无法匹配视觉效果上的一个tab

str="ab\tcde";
str.match(reg);//[" c"]
//即"\t"只能匹配字符"\t"，控制台对\t的打印以转义序列制表符"\t"的方式打印
//所以最后结果是[" c"]

//其他"\n"、"\f"、"\v"类似于"\t"的使用
//"."---->[^/r/n]，匹配"非"行结束符和换行符
```

- unicode编码，\uXXXX，一般为4位16进制码

  不需要记住哪个字符对应是哪个Unicode编码，要用到的时候可借助“Unicode在线编码转换器”


```
var reg=/\u8eab\u4f53\u597d/g;
var str="身体好";
str.match(reg);//["身体好"]

reg=/[\u4000-\u9000]/g;//也可使用区间去定义规则
str="身体好";
str.match(reg);//["身","体","好"]

var reg=/[\u0000-\uffff]/g;//能够代表几乎一切字符
reg=/[\s\S]/g;//这才能够代表匹配一切
```

（5）量词

量词，代表数量的词（下面表达式的n代表的是一个匹配规则，n后边符号的符号定义量词规则）。

贪婪匹配。

量词	描述
n+	匹配任何包含至少一个 *n* 的字符串。1~多
n*	匹配任何包含零个或多个 *n* 的字符串。 0~多
n?	匹配任何包含零个或一个 *n* 的字符串。 0~1
n{X}	匹配包含X 个n 的序列的字符串。刚好是X次
n{X, Y}	匹配任何包含X 个至Y 个n 的序列的字符串。x~y
n{X, }	匹配包含至少X 个n 的序列的字符串。x~多
n$	匹配任何结尾为n 的字符串
^n	匹配任何开头为n 的字符串
S(?=n)	匹配任何其后紧接指定字符串n 的字符串S
S(?!n)	匹配任何其后没有紧接指定字符串n 的字符串S

- n+ ----> {1,Infinity}个
- n* ----> {0,Infinity}个

```
var reg=/\w+/g;//匹配\w类的字符出现一次或多次的字符串
var str="abc";
str.match(reg);//["abc"]，贪婪匹配原则

reg=/\w*/g;//匹配\w类的字符出现零次或多次的字符串
str="abc";
str.match(reg);//["abc",""]
//全局匹配到"abc"之后，逻辑上从那以后还有段距离
//这段距离符合规则"出现零次或多次的字符串"，所以匹配出来一个空字符串
//贪婪匹配原则

var reg=/\d*/g;
var str="abc";
str.match(reg);//["","","",""]
//该原则对字母a/b/c是无法匹配出来的，所以匹配出空字符串""
//最后再在最后一个光标位匹配出一个空字符串来，所以一共4个空字符串
```

- n? ---->0个或1个

```
var reg=/\w?/g;
var str="aaaaa";
str.match(reg);//["a","a","a","a","a",""]，最后还会有一个空字符串""
//若要在正则表达式里匹配问号?，直接写/?/是不行的，需要加上转义字符，写成/\?/
```

- ^n ----> 匹配以n 为开头的字符串n
- n$ ----> 匹配以n 为结尾的字符串n

```
var reg=/ed$/g;//以ed为结尾时匹配
var str="abcded";
str.match(reg);//["ed"]

var reg=/^abc$/g;//以该abc开头，且以同一abc为结尾的字符串
var str="abcabc";
str.match(reg);//null

var reg=/^abc$/g;//以该abc开头，且以同一abc为结尾的字符串
var str="abc";
str.match(reg);//["abc"]
```

- ?=n -----\>n只参与条件限定，限定其后紧接指定字符串n 的字符串，但并不参与选择
- ?!n -----\>n只参与条件限定，限定其后没有紧接指定字符串n 的字符串，但并不参与选择

```
//正向预查  正向断言
var str="abaaaa";
var reg=/a(?=b)/g;//b只参与条件限定，并不参与选择
str.match(reg);//["a"],此时只有一个a 它的后面有紧跟字符b

var str="abaaaa";
var reg=/a(?!b)/g;
str.match(reg);//["a","a","a","a"],此时有4个a 它的后面没有紧跟字符b
```

练习1：写一个正则表达式，检验字符串首尾是否**含有**数字。

分析题目：字符串首部或尾部有数字

```
var reg=/^\d|\d$/g;
var str="123abc";
str.match(reg);//["1"]
```

练习2：写一个正则表达式，检验字符串首尾是否**都有**数字。

分析题目：字符串首部和尾部都有数字

```
var reg=/^\d[\s\S]*\d$/g; //[\s\S]指开始与结束字符中间为任意字符，*指出现0到多次
var str="123abc123";
str.match(reg);//["123abc123"]
```

3、正则表达式对象

在 JavaScript 中，RegExp 对象是一个预定义了属性和方法的正则表达式对象。

（1）方法

- **test()**：用于检测一个字符串是否匹配某个模式，如果字符串中含有匹配的文本，则返回 true，否则返回 false。

```
var patt = /e/;
patt.test("The best things in life are free!");
```

- **exec()**：executive，用于检索字符串中的正则表达式的匹配。该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。

```
var reg=/ab/g;
var str="abababab";
reg.exec(str);//["ab"]，尽管具有global属性，仍然只返回一个"ab"

//再继续
reg.exec(str);//["ab"]
reg.exec(str);//["ab"]
reg.exec(str);//["ab"]
reg.exec(str);//null
reg.exec(str);//["ab"]
reg.exec(str);//["ab"]

//……

//用console.log来试一遍
console.log(reg.exec(str));//["ab",index:0,input:"abababab"]，类数组
console.log(reg.exec(str));//["ab",index:2,input:"abababab"]
//第二次匹配index变为2，是第二个"ab"字符串在原字符串中的起始位置
console.log(reg.exec(str));//["ab",index:4,input:"abababab"]
console.log(reg.exec(str));//["ab",index:6,input:"abababab"]
console.log(reg.exec(str));//null
```

- **toString()**：返回正则表达式的字符串值

```
var patt = new RegExp("hello", "g");
var res = patt.toString();
console.log(res); ///hello/g
```

（2）属性

- constructor：返回一个函数，该函数是一个创建 RegExp 对象的原型。

```
var patt = new RegExp("RUNOOB", "g");
var res = patt.constructor;
console.log(res); //f RegExp() { [native code] }
```

- global：判断是否设置了 "g" 修饰符

```
var str = "www.baidu.com";
var patt1 = /baidu/g;
if(patt1.global) {
    document.write("g 模式有设置!");
} else {
    document.write("g 模式没有设置!");
}
```

- ignoreCase：判断是否设置了 "i" 修饰符

```
var patt2 = /www/i;
if(patt2.global) {
    console.log("i 模式有设置!");
} else {
    console.log("i 模式没有设置!");
}
```

- multiline：判断是否设置了 "m" 修饰符

```
var str = "www.baidu.com";
var patt1 = /baidu/m;
if(patt1.multiline) {
    console.log("m 模式有设置!");
} else {
    console.log("m 模式没有设置!");
}
```

- source：返回正则表达式的匹配模式

```
var patt1 = /hello/g;
console.log("The text of the RegExp is: " + patt1.source);
```

- lastIndex：用于规定下次匹配的起始位置，该属性只有设置标志 g 才能使用。

上次匹配的结果是由方法 RegExp.exec() 和 RegExp.test() 找到的，它们都以 lastIndex 属性所指的位置作为下次检索的起始点。这样，就可以通过反复调用这两个方法来遍历一个字符串中的所有匹配文本。

```
var str = "The rain in Spain stays mainly in the plain";
var patt1 = /ain/g;

while(patt1.test(str) == true) {
    document.write("'ain' found. Index now at: " + patt1.lastIndex);
    document.write("<br>");
}
```

注意：该属性是可读可写的。只要目标字符串的下一次搜索开始，就可以对它进行设置。当方法 exec() 或 test() 再也找不到可以匹配的文本时，它们会自动把 lastIndex 属性重置为 0。

4、支持正则表达式的 String 对象的方法

（1）search

用于检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串，并返回匹配字符串第一次出现的下标。如果没有找到任何匹配的子串，则返回 -1。

```
var str="Mr. Blue has a blue house";
console.log(str.search("blue")); //15

var str="Mr. Blue has a blue house";
console.log(str.search(/blue/i)); //4
```

（2）match

可在字符串内检索指定的值，或找到一个或多个正则表达式的匹配

match() 方法将检索字符串 String Object，以找到一个或多个与 regexp 匹配的文本。这个方法的行为在很大程度上有赖于 regexp 是否具有标志 g。如果 regexp 没有标志 g，那么 match() 方法就只能在 stringObject 中执行一次匹配。如果没有找到任何匹配的文本， match() 将返回 null。否则，它将返回一个数组，其中存放了与它找到的匹配文本有关的信息。

```
var str = "The rain in SPAIN stays mainly in the plain";
var n = str.match(/ain/gi);
console.log(n); //(4) ["ain", "AIN", "ain", "ain"]
```

（3）replace

在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串。

注意：该方法不会改变原始字符串。

```
var str = "Mr Blue has a blue house and a blue car";
var n = str.replace(/blue/g,"red");
console.log(n); //Mr Blue has a red house and a red car
```

通过 prototype 为 JavaScript 的 String 对象添加方法，来实现将所有 "Microsoft" 替换为 "JavaScript"：

```
String.prototype.replaceAll = function(search, replacement) {
    var target = this;
    return target.replace(new RegExp(search, 'g'), replacement);
};

var str = "Visit Microsoft!Visit Microsoft!Visit Microsoft!";
str.replaceAll("Microsoft", "JavaScript");
```

（4）split

用于把一个字符串分割成字符串数组。如果把空字符串 ("") 用作 separator，那么 stringObject 中的每个字符之间都会被分割。

注意：split() 方法不改变原始字符串。

```
var str = "How are you doing today?";
var n = str.split(" ");
console.log(n); //(5) ["How", "are", "you", "doing", "today?"]
```

5、正则表达式表单验证实例

```
/*是否带有小数*/
function isDecimal(strValue )  {
   var objRegExp= /^\d+\.\d+$/;
```

```
    return objRegExp.test(strValue);
}

/*校验是否中文名称组成 */
function ischina(str) {
    var reg=/^[\u4E00-\u9FA5]{2,4}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验是否全由8位数字组成 */
function isStudentNo(str) {
    var reg=/^[0-9]{8}$/;   /*定义验证表达式*/
    return reg.test(str);     /*进行验证*/
}

/*校验电话码格式 */
function isTelCode(str) {
    var reg= /^((0\d{2,3}-\d{7,8})|(1[3584]\d{9}))$/;
    return reg.test(str);
}

/*校验邮件地址是否合法 */
function IsEmail(str) {
    var reg=/^\w+@[a-zA-Z0-9]{2,10}(\.[a-z]{2,4}){1,3}$/;
    return reg.test(str);
}
```

window对象

1、浏览器对象模型

浏览器对象模型 (BOM) 使 JavaScript 有能力与浏览器"对话"。

（1）window对象

所有浏览器都支持 window 对象。它表示浏览器窗口。

所有 JavaScript 全局对象、函数以及变量均自动成为 window 对象的成员。

全局变量是 window 对象的属性。

全局函数是 window 对象的方法。

甚至 HTML DOM 的 document 也是 window 对象的属性之一。

```
window.document.getElementById("header");
```

（2）window尺寸

对于Internet Explorer、Chrome、Firefox、Opera 以及 Safari：

- window.innerHeight - 浏览器窗口的内部高度(包括滚动条)
- window.innerWidth - 浏览器窗口的内部宽度(包括滚动条)

```
console.log(window.innerWidth, window.innerHeight);
```

（3）其他window方法

- **window.open()** - 打开新窗口

```
function open_win() {
    window.open("http://www.baidu.com", "_blank");
}
```

- **window.close()** - 关闭当前窗口

```
function close_win() {
    window.close();
}
```

2、window对象-定时器

通过使用 JavaScript，我们有能力做到在一个设定的时间间隔之后来执行代码，而不是在函数被调用后立即执行。我们称之为计时事件。

在 JavaScritp 中使用计时事件是很容易的，两个关键方法是:

- **setInterval()** - 间隔指定的毫秒数不停地执行指定的代码。
- **setTimeout()** - 在指定的毫秒数后执行指定代码。

注意: setInterval() 和 setTimeout() 是 HTML DOM Window对象的两个方法。

示例1：显示当前时间

```
var myVar=setInterval(function(){myTimer()},1000);

function myTimer(){
    var d=new Date();
    var t=d.toLocaleTimeString();
    document.getElementById("demo").innerHTML=t;
}
```

示例2：等待3秒，然后弹出 "Hello"

```
setTimeout(function(){alert("Hello")},3000);
```

- **clearInterval()** 方法用于停止 setInterval() 方法执行的函数代码。
- **clearTimeout()** 方法用于停止执行setTimeout()方法的函数代码。

3、window对象-Location

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。

window.location 对象在编写时可不使用 window 这个前缀。

- **location.href** 属性返回当前页面的 URL。
- **location.hostname** 返回 web 主机的域名
- **location.port** 返回 web 主机的端口 （80 或 443）
- **location.pathname** 返回当前页面的路径和文件名
- **location.protocol** 返回所使用的 web 协议（http:// 或 https://）

```
console.log(location.href); //http://127.0.0.1:8020/MyJS/02location.html
console.log(location.hostname); //127.0.0.1
console.log(location.port); //8020
console.log(location.protocol); //http:
console.log(location.pathname); //MyJS/02location.html
```

- **assign()**：载入一个新的文档，浏览器可以后退回之前的文档
- **reload()**：重新载入当前文档
- **replace()**：用新的文档替换当前文档，浏览器不能后退

```
<button onclick="assignDoc()">载入新文档</button>
<button onclick="reloadDoc()">重新加载文档</button>
<button onclick="replaceDoc()">用新文档替换当前文档</button>

function assignDoc() {
    location.assign("http://www.baidu.com")
}
function reloadDoc() {
    location.reload();
}
function replaceDoc() {
    location.replace("http://www.qq.com");
}
```

4、window对象-History

History 对象包含用户（在浏览器窗口中）访问过的 URL。

History 对象是 window 对象的一部分，可通过 window.history 属性对其进行访问。

- **length**：返回历史列表中的网址数
- **back()**：加载 history 列表中的前一个 URL，与在浏览器点击后退按钮相同
- **forward()**：加载 history 列表中的下一个 URL，与在浏览器中点击向前按钮相同
- **go()**：加载 history 列表中的某个具体页面。go(1)表示前进一个页面，等同于forward()；go(-1)表示后退一个页面，等同于back()
- 示例：

- **one.html**

```
<button onclick="assignDoc()">载入新文档</button>
<button onclick="forwardWin()">前进</button>

<script type="text/javascript">
    console.log(history.length);

    function assignDoc() {
        location.assign("./two.html")
    }

    function forwardWin() {
        history.forward();
    }
</script>
```

- **two.html**

```
<button onclick="backWin()">后退</button>

<script type="text/javascript">
    function backWin() {
        history.back();
    }
</script>
```