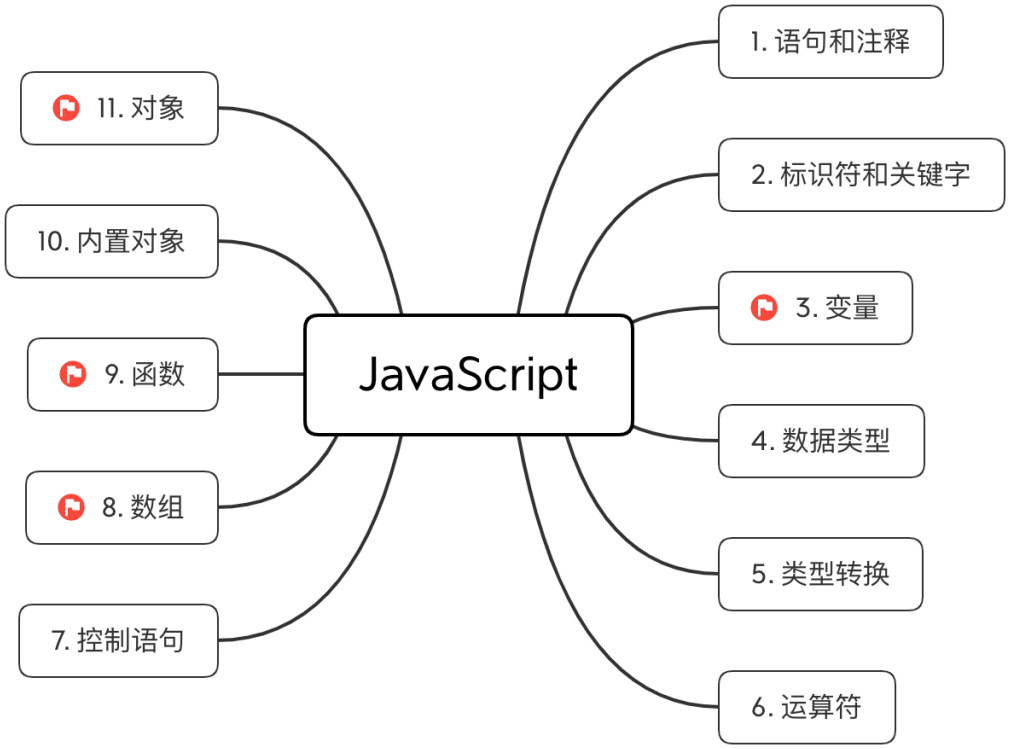
JavaScript基础语法

1. 主要内容

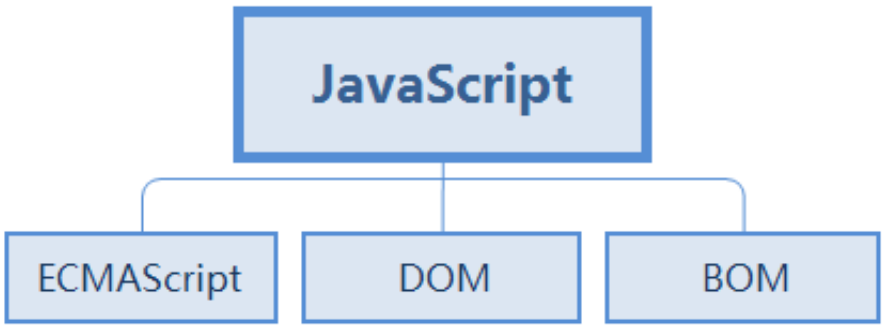


2. JavaScript

2.1. 简介

JavaScript 是一种具有面向对象能力的、解释型的程序设计语言。更具体一点，它是基于对象和事件驱动并具有相对安全性的客户端脚本语言。它的主要目的是，验证发往服务器端的数据、增加 Web 互动、加强用户体验度等。

2.1.1. JavaScript 的组成



ECMAScript定义的只是这门语言的基础，与Web浏览器没有依赖关系，而在基础语法上可以构建更完善的脚本语言。JavaScript的运行需要一定的环境，脱离了环境JavaScript代码是不能运行的，JavaScript只能够寄生在某个具体的环境中才能够工作。JavaScript运行环境一般都由宿主环境和执行期环境共同构成，其中宿主环境是由外壳程序生成的，如Web浏览器就是一个外壳程序，它提供了 一个可控制浏览器窗口的宿主环境。执行期环境则由嵌入到外壳程序中的JavaScript引擎（或称为JavaScript解释器）生成，在这个环境中 JavaScript能够生成内置静态对象，初始化执行环境等。

Web浏览器自定义的DOM组件，以面向对象方式描述的文档模型。DOM定义了表示和修改**文档**所需的对象、这些对象的行为和属性以及这些对象之间的关系。DOM对象，是我们用传统的方法(javascript)获得的对象。DOM属于浏览器，而不是JavaScript语言规范里的规定的核心内容。

前面的DOM是为了操作浏览器中的文档，而为了控制浏览器的行为和操作，浏览器还提供了BOM（浏览器对象模型）。

ECMAScript(基础语法)

JavaScript的核心语法ECMAScript描述了该语言的语法和基本对象

DOM(文档对象模型)

文档对象模型（DOM）—— 描述了处理网页内容的方法和接口

BOM(浏览器对象模型)

浏览器对象模型（BOM）—— 描述了与浏览器进行交互的方法和接口

2.1.2. 开发工具

1. 浏览器：Chrome
2. 开发工具：Hbuilder X
3. 进入浏览器控制台 Console：F12

   控制台的作用:

   console对象代表浏览器的JavaScript控制台，用来运行JavaScript命令，常常用来显示网页运行时候的错误信息。Elements用来调试网页的html和css代码。

2.2. 基本用法

JS需要和HTML一起使用才有效果，我们可以通过直接或间接的方式将JS代码嵌入在HTML页面中。

行内JS : 写在标签内部的js代码

内部JS : 定义在script标签内部的js代码

外部JS : 单独的js文件，在HTML中通过script标签引入

我们可以将JavaScript代码放在html文件中任何位置，但是我们一般放在网页的head或者body部分。由于页面的加载方式是从上往下依次加载的，而这个对我们放置的js代码运行是有影响的。

放在部分，最常用的方式是在页面中head部分放置元素，浏览器解析head部分就会执行这个代码，然后才解析页面的其余部分。

放在部分，JavaScript代码在网页读取到该语句的时候就会执行。

行内 JS：

```
<button onclick="alert('you clicked hered!!!')">click here</button>
```

内部 JS：

```
<script type="text/javascript" charset="utf-8">
  alert('this is inner js code')
</script>
```

外部 JS 文件：

hello.js

```
alert('this is a outter js document');
```

hello.html

```
<!-- 在需要使用js的html页面中引入 -->
<script src="js/hello.js" type="text/javascript" charset="utf-8"></script>
```

3. JavaScript基础语法

3.1. 语句和注释

JavaScript程序的执行单位为行（line），也就是一行一行地执行。一般情况下，每一行就是一个语句。

语句（statement）是为了完成某种任务而进行的操作，语句以分号结尾，一个分号即表示一个语句结束。多个语句可以写在一行内（不建议这么写代码），但是一行写多条语句时，语句必须以分号结尾。

表达式不需要分号结尾。一旦在表达式后面添加分号，则JavaScript引擎就将表达式视为语句，这样会产生一些没有任何意义的语句。

```
单行注释：用//起头；
多行注释：放在/* 和 */之间。
兼容html注释方式：<!-- -->
```

3.2. 标识符和关键字

标识符就是一个名字，用来给变量和函数进行命名，有特定规则和规范

规则：

```
由Unicode字母、_、$、数字组成、中文组成
（1）不能以数字开头
（2）不能是关键字和保留字
（3）严格区分大小写
```

规范：

```
（1）见名知意
（2）驼峰命名或下划线规则
```

关键字也称保留字，是被JavaScript征用来有特殊含义的单词

```
arguments、break、case、catch、class、const、continue、debugger、default、delete、do、
else、enum、eval、export、extends、false、finally、for、function、if、implements、import、
in、instanceof、interface、let、new、null、package、private、protected、public、return、
static、super、switch、this、throw、true、try、typeof、var、void、while、with、yield、
Infinity、NaN、undefined
```

3.3. 变量

变量即一个带名字的用来存储数据的内存空间，数据可以存储到变量中，也可以从变量中取出数据。

3.3.1. 变量的声明

JavaScript是一种弱类型语言，在声明变量时不需要指明数据类型，直接用**var**修饰符进行声明。

变量声明和赋值：

```
// 先声明再赋值
var a ;
a = 10;
// 声明同时赋值
var b = 20;
```

3.3.2. 变量的注意点

（1）若只声明而没有赋值，则该变量的值为undefined。

```
var box;
console.log(box);
```

（2）变量要有定义才能使用，若变量未声明就使用，JavaScript会报错，告诉你变量未定义。

```
console.log(box2);
```

（3）可以在同一条var命令中声明多个变量。

```
var a, b, c = 10;
console.log(a,b,c);
```

（4）若使用var重新声明一个已经存在的变量，是无效的。

```
var box = 10
var box;
```

（5）若使用var重新声明一个已经存在的变量且赋值，则会覆盖掉前面的值

```
var box = 10;
var box = 25;
```

（6）JavaScript是一种动态类型、弱类型语言，也就是说，变量的类型没有限制，可以赋予各种类型的值。

```
var box = 'hello world';
box = 10;
```

3.3.3. 变量提升

JavaScript 引擎的工作方式是，先解析代码，获取所有被声明的变量，然后再一行一行地运行。这造成的结果，就是所有的变量的声明语句，都会被提升到代码的头部，这就叫做变量提升。

```
console.log(msg);
var msg = "so easy";

// 变量提升，相当于下面的代码
var msg;
console.log(msg);
msg = "so easy";

// 说明：  最后的结果是显示undefined，表示变量msg已声明，但还未赋值。
```

注意：变量提升只对 var 命令声明的变量有效，如果变量不是用 var 命令声明的，就不会发生变量提升。

```
console.log(msg);
msg = "error";
```

3.4. 数据类型

虽说 JS 是弱类型语言，变量没有类型，但数据本身是有类型的。针对不同的类型，我们可以进行不同的操作。

JavaScript 中有 6 种数据类型，其中有五种简单的数据类型：**Undefined**、**Null**、**布尔**、**数值**和**字符串**。一种复杂数据类型**Object**。

```
数　值（Number）：　整数和小数（比如 1 和 3.14）
字符串（String）：　字符组成的文本（比如"Hello World"）
布尔值（Boolean）：true（真）和 false（假）两个特定值
Undefined:　　表示“未定义”或不存在，即此处目前没有任何值
Null:　　表示空缺，即此处应该有一个值，但目前为空
对象（object）（引用）：　各种值组成的集合
    1)、对象（object）{name:”zhangsan”,age:”18”}
    2)、数组（array）[1,2,3]
    3)、函数（function）function test() {}
```

3.4.1. undefined

undefined 类型的值是 undefined。

undefined 是一个表示"无"的原始值，表示值不存在。

出现undefined的常见情况：

（1）当声明了一个变量而没有初始化时，这个变量的值就是 undefined

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

（2）调用函数时，该函数有形参，但未提供实参，则该参数为 undefined。

```
function noData(str) { // js函数形参只需要变量名即可
  console.log(str); // undefined
}
noData(); // 调用方法时，未传递参数
```

（3）函数没有返回值时，默认返回 undefined。

```
// 方法没有返回值
function noData() {
  console.log("Hello");
}
var re = noData();// 定义变量接收无返回值的方法
console.log(re);
```

3.4.2. null

null类型是只有一个值的数据类型，即特殊的值null。它表示空值，即该处的值现在为空，它表示一个空对象引用。

使用Null类型值时注意以下几点：

1）使用 typeof 操作符测试null返回object字符串。

2）undefined 派生自 null，所以等值比较返回值是true。未初始化的变量和赋值为null的变量相等。

```
console.log(undefined == null);
var box = null; // 赋值为null的变量
var a; // 未初始化的变量
console.log(a == box);  // 两个的值相等
```

3.4.3. 布尔类型

布尔类型有两个值：true、false。常用来做判断和循环的条件

3.4.4. 数值型

数值型包含两种数值：整型和浮点型。

1）所有数字（整型和浮点型）都是以 64 位浮点数形式储存。所以，JS中1 与 1.0 相等，而且 1 加上 1.0 得到的还是一个整数。浮点数最高精度是17位小数，由于浮点数运算时可能不精确，尽量不要使用浮点数做判断。

2）在存储数值型数据时自动将可以转换为整型的浮点数值转为整型。

```
console.log(1 == 1.0); // true
console.log(1 + 1.0); // 2
var num = 8.0; // 自动将可以转换为整型的浮点数转为整型
console.log(num); // 8
```

3.4.5. 字符串

使用 ' ' 或 " "引起来，如：'hello'，"good"。

使用加号 '+' 进行字符串的拼接，如：console.log('hello' + ' everybody');

3.4.6. 对象

对象是一组数据和功能的集合。

说明：

{}：表示使用对象字面量方式定义的对象。空的大括号表示定义包含默认属性和方法的对象。

3.5. 类型转换

3.5.1. 自动类型转换

值	字符串	数字	布尔值
undefined	"undefined"	NaN	false
null	"null"	0	false
true	"true"	1	true
false	"false"	0	false
""（空字符串）	""	0	false
"1.5"	"1.5"	1.5	true
"one"	"one"	NaN	true

值	字符串	数字	布尔值
0	"0"	0	false
-0	"0"	0	false
NaN	"NaN"	NaN	false
Infinity	"Infinity"	Infinity	true
-Infinity	"-Infinity"	-Infinity	true
1	"1"	1	true

3.5.2. 函数转换（String to Number）

JS 提供了 **parseInt()**和 **parseFloat()**两个全局转换函数。前者把值转换成整数，后者把值转换成浮点数。只有对 String 类型调用这些方法，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)。

3.5.2.1. parseInt()

在转换之前，首先会分析该字符串，判断位置为0处的字符，判断它是否是个有效数字，如果不是，则直接返回NaN，不再继续，如果是则继续，直到找到非字符

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

3.5.2.2. parseFloat()

该方法与 parseInt() 方法的处理方式相似，从位置 0 开始查看每个字符，直到找到第一个非有效的字符为止，然后把该字 符之前的字符串转换成数字。不过，对于这个方法来说，第一个出现的小数点是有效字符。如果有两个小数点，第二个小数点将被看作无效的，parseFloat()方法会把这个小数点之前的字符串转换成数字。

```
parseFloat("1234blue"); // returns 1234.0
parseFloat("22.5"); // returns 22.5
parseFloat("22.34.5"); // returns 22.34
parseFloat("blue"); //returns NaN
```

3.5.3. 显示转换

几乎每个数对象都提供了**toString()**函数将内容转换为字符串形式，其中Number提供的**toString**()函数可以将数字转换为字符串。

Number还提供了**toFixed()**函数将根据小数点后指定位数将数字转为字符串，四舍五入

```
// 将内容转换为字符串形式
var data = 10
console.log(data.toString())

// 根据小数点后指定位数将数字转为字符串，四舍五入
data = 1.4;
console.log(data.toFixed(0));
data = 1.49;
console.log(data.toFixed(1));


// 不能对null和undefined使用
data = null
console.log(data.toString())
data = undefined
console.log(data.toString())
```

JS 为 Number、Boolean、String 对象提供了构造方法，用于强制转换其他类型的数据。此时操作的是整个数据，而不是部分。

```
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

Boolean("");      // false - empty string
Boolean("hi");      // true - non-empty string
Boolean(100);       // true - non-zero number
Boolean(null);      // false - null
Boolean(0);       // false - zero
```

```
Boolean(new Object());  // true - object
```

最后一种强制类型转换方法 String() 是最简单的，因为它可把任何值转换成字符串。要执行这种强制类型转换，只需要调用作为参数传递进来的值的 toString() 方法，即把 1 转换成"1 "，把 true转换成 "true "，把 false 转换成 "false "，依此类推。强制转换成字符串和调用 toString() 方法的唯一不同之处在于，对 null 或 undefined 值强制类型转换可以生成字符串而不引发错误：

```
var s1 = String(null); // "null"
var oNull = null;
var s2 = oNull.toString(); // won't work, causes anerror
```

最为简单的一种转换为字符串的方式，直接在任意数据后面 + "" 即可。

3.6. 运算符

运算符用于执行程序代码运算，会针对一个及其以上操作数来进行运算。

3.6.1. 算数运算符

运算符	描述	例子	结果
+	加	x=y+2	x=7
-	减	x=y-2	x=3
*	乘	x=y*2	x=10
/	除	x=y/2	x=2.5
%	求余数	x=y%2	x=1
++	自增（前导加、后导加）	x=++y	x=6
--	自减（前导减、后导减）	x=--y	x=4

3.6.2. 赋值和扩展运算符

运算符	例子	等价于	结果
=	x=y		x=5
+=	x+=y	x=x+y	x=15
-=	x-=y	x=x-y	x=5
*=	x*=y	x=x*y	x=50
/=	x/=y	x=x/y	x=2
%=	x%=y	x=x%y	x=0

3.6.3. 比较运算符

运算符	描述	例子
==	等于	x==8 为 false
===	全等（值和类型）	x===5 为 true；x==="5" 为 false
!=	不等于	x!=8 为 true
>	大于	x>8 为 false
<	小于	x<8 为 true
>=	大于或等于	x>=8 为 false
<=	小于或等于	x<=8 为 true

3.6.4. 逻辑运算符

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

3.6.5. 三目运算符

运算符	描述	例子
?:	如果…否则…	3>5?3:5

3.7. 控制语句

我们写的 JavaScript 代码都是按照从上到下依次执行，很多时候我们希望代码按照我们的意愿去执行，比如有选择性地执行某些代码，或者重复地执行某些代码，这就需要使用到流程控制语句。

流程控制语句一共有三种：

1. 流程执行:从上到下，从左到右
2. 选择执行:分支选择
3. 循环执行:重复执行

3.7.1. 选择

3.7.1.1. 单选择

```
if (条件){
    语句体;
}
```

首先执行条件

如果结果为true，则执行语句体；

如果结果为false，则结束if语句。

注意：若语句体只有一条语句，可以省略大括号，但不建议省略

3.7.1.2. 双选择

```
if (条件){
    语句体1;
}else {
    语句体2;
}
```

首先执行条件

如果结果为true，则执行语句体1；

如果结果为false，则执行语句体2。

3.7.1.3. 多选择

```
if(比较表达式1) {
  语句体1;
}else if(比较表达式2){
  语句体2;
}else if(比较表达式3){
  语句体3;
}
  ...
[else {
  语句体n+1;
}]
```

3.7.1.4. switch结构

多个 if ...else 且值为定值时（即=== 在比较运行结果时，采用的是严格相等运算符（===），而不是相等运算符（==），这意味着比较时不会发生类型转换。），可以使用 switch 替换：

```
switch(表达式) {
  case 值1:
    语句体1;
    break;
  case 值2:
    语句体2;
    break;
  ...
  default:
    语句体n+1;
    [break;]
}
```

break 防止穿透，如果没有 break,则继续执行后面的代码，直到遇到 break 或全部执行完毕，但是有些时候会利用穿透。

3.7.2. 循环

循环结构用于重复执行某个操作 简单理解就是重复执行同类型的代码，它有多种形式。

3.7.2.1. while

先判断后执行

```
基本格式
  while(判断条件语句) {
    循环体语句;
  }

扩展格式:
  初始化语句;
  while(判断条件语句){
    循环体语句;
    控制条件语句; //    少了它很容易形成死循环
    }
```

3.7.2.2. do...while

先执行后判断，至少执行一次

```
基本格式
  do {
    循环体语句;
  }while(判断条件语句);

扩展格式:
  初始化语句;
  do {
    循环体语句;
    控制条件语句;
  } while(判断条件语句);
```

3.7.2.3. for

```
for(初始化语句;判断条件语句;控制条件语句){
   循环体语句;
}
```

3.7.2.4. 死循环

条件永远成立，永远为 true,则会产生死循环，下面是最简单的死循环

```
while(true){}
for(;;){}
```

3.7.2.5. break 与 continue

break: 停止本层循环

continue:暂停本次循环，继续下一次

3.8. 数组

数组是按次序排列的一组数据，每个值的位置都有编号（从0开始），整个数组用方括号表示。

3.8.1. 数组定义

JS 中定义数组的三种方式如下（也可先声明再赋值）：

```
var arr = [值1,值2,值3];  // 隐式创建

var arr = new Array(值1,值2,值3); // 直接实例化

var arr = new Array(size); // 创建数组并指定长度
```

3.8.2. 基本操作

数组的长度可以通过length属性来获取，并可以任意更改

```
数组名.length
数组名.length = 新长度
```

数组中的每一个元素都可以被访问和修改，甚至是不存在的元素，无所谓越界

```
数组名[下标]
数组名[下标] = 新值
```

3.8.3. 数组遍历

数组的遍历即依次访问数组的每一个元素 ， JS提供三种遍历数组的方式：

3.8.3.1. 普通的for循环遍历

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

}
如:
for(var idx=0;idx<arr.length;idx++){
    console.log(arr[idx]);
}
```

3.8.3.2. for ... in

```
for(var 下标(名称任意) in 数组名){
  数组名[下标]是获取元素
} // 下标(名称任意)
如:
for(var idx in arr){
    console.log(arr[idx]);
}
```

3.8.3.3. forEach

```
数组名.forEach(function(element,index){
    // element(名称任意): 元素, index(名称任意): 下标
})
如:
arr.forEach(function(elem,idx){
    console.log(idx + "-->" + elem);
});
```

3.8.3.4. 了解

```
数组在使用的时候建议大家规矩来用。在存放数据时，从下标0开始顺序的存放数组元素。
如果下标:
  1.为非负整数(包括整数字符串): 自动从0开始,不存在添加 undefined
  2.为负数、小数、非数字符串: 这些内容不计算在长度内，当成"属性"处理，相当于自定义属性。

数组非常灵活，使用数组元素
  1.下标: 非负整数(包括整数字符串):
    数组.下标
    数组[下标]
  2.下标: 负数、小数、非数字字符串:
      数组[属性]

* for --> 不遍历属性
* foreach -->不遍历属性和索引中的undefined
* for in -->不遍历索引中的undefined
```

3.8.4. 数组提供的操作方法

Array对象为我们提供了一些方法，可以很方便地操作数组

```
push            添加元素到最后
unshift         添加元素到最前
pop             删除最后一项
shift           删除第一项
reverse         数组翻转
join            数组转成字符串
indexOf         数组元素索引
slice           截取（切片）数组，原数组不发生变化
splice          剪接数组，原数组变化，可以实现前后删除效果
concat          数组合并
```

```
var arr = ['1','a',5,'3'];
console.log(arr);
arr.push(10);
console.log(arr);
arr.unshift('b');
console.log(arr);
arr.pop();
console.log(arr);
arr.shift();
console.log(arr);
arr.reverse();
console.log(arr);
console.log(arr.join('\''));
console.log(arr);
console.log(arr.indexOf('a'));
console.log(arr.slice(2,5));
console.log(arr);
arr.splice(1,1,'一','二');
console.log(arr);
var arr1 = [0,'100'];
console.log(arr.concat(arr1));
console.log(arr);
console.log(arr1);
console.log(arr1.(arr));
```

3.9. 函数

函数，即方法。就是一段预先设置的功能代码块，可以反复调用，根据输入参数的不同，返回不同的值。函数也是对象。

3.9.1. 函数的定义

有三种函数定义的方式：函数声明语句、函数定义表达式、Function构造函数

3.9.1.1. 函数声明语句

```
function 函数名([参数列表]){

}
例如:
function foo(){
    console.log(1);
}
foo();
```

该种方式定义的函数具有声明提升的效果

```
foo();
function foo(){
        console.log(1);
}
// 变量声明提升
console.log( a );
var a = 2;
```

3.9.1.2. 函数定义表达式

以表达式方式定义的函数，函数的名称是可以不需要的

```
var 变量名 = function ([参数列表]) {

}
变量名();
例如:
var fun = function(){
    console.log("Hello");
}
fun();
```

这种写法将一个匿名函数赋值给变量。这时，这个匿名函数又称函数表达式，因为赋值语句的等号右侧只能放表达式。

3.9.1.3. Function构造函数

Function构造函数接收任意数量的参数，但最后一个参数始终都被看成是函数体，而前面的参数则列举出了新函数的参数。

```
var add = new Function('x','y','return (x + y)');
// 等同于
function add(x, y) {
    return (x + y);
}
add();
```

注意：

1. js中的函数没有重载，同名的函数，会被后面的函数覆盖。
2. js中允许有不定数目的参数，后面介绍arguments对象

3.9.2. 函数的参数、调用和return语句

3.9.2.1. 参数

函数运行的时候，有时需要提供外部数据，不同的外部数据会得到不同的结果，这种外部数据就叫参数，定义时的参数称为形参，调用时的参数称为实参

- 实参可以省略，那么对应形参为undefined
- 若函数形参同名（一般不会这么干）：在使用时以最后一个值为准。
- 可以给参数默认值：当参数为特殊值时，可以赋予默认值。
- 参数为值传递，传递副本 ；引用传递时传递地址，操作的是同一个对象。

```
// 调用函数时，实参可以省略，则对应形参为undefined
function add(a , b) {
  console.log(a + "+" + b + "=" + (a + b));
}
add(3,4,5)//3+4=7
add(1);//1+undefined=NaN
add();//undefined+undefined=NaN

// 若函数形参同名（一般不会这么干）：在使用时以最后一个值为准
function add2(a , a) {
  console.log(a);
}
add2(1,2);

// 给参数默认值
function defaultValue(a){
  a = a || "a";
  return a;
}
console.log(defaultValue());
function f(a){
  //若参数a不为undefined或null，则取本身的值，否则给一个默认值
  (a !== undefined && a !== null) ? a = a : a = 1;
  return a;
}
console.log(f());

// 值传递
var num = 12;
function change(n) {
  n = 30;
}
change(num);
console.log(num);
// 引用传递
var obj = {name: "tom"};
function paramter(o) {
  o.name = 2;
}
paramter(obj);
```

```
console.log(obj.name);
// 给形参o赋予了新的数组
var obj2 = [1, 2, 3];
function paramter2(o){
  o = [2, 3, 4];
  o[1] = 3;
}
paramter2 (obj2);
console.log(obj2)
```

3.9.2.2. 函数的调用

1. 常用调用方式

```
函数名([实参]);
```

存在返回值可以变量接收，若接收无返回值函数则为undefined。

2. 函数调用模式

```
function add(a,b){
   return a+b;
}
var sum = add(1,2)
console.log(sum);
```

3. 方法调用模式

```
var o = {
    m: function(){
         console.log(1);
    }
};
o.m();
```

3.9.2.3. return

函数的执行可能会有返回值，需要使用return语句将结果返回。return 语句不是必需的，如果没有的话，该函数就不返回任何值，或者说返回 undefined。

作用：

1. 在没有返回值的方法中，用来结束方法。

2. 有返回值的方法中，一个是用来结束方法，一个是将值带给调用者。

3.9.3. 函数的作用域

函数作用域：全局 (global variable) 和 局部 (local variable)

1. 全局变量与局部变量同名问题

```
var box =1; // 全局变量
function display(box){
  var box = 3; // 此处box与全局变量box没有关系，这里的box为传递的参数，相当于新声明的局部变量
  var b = 2; // 局部变量
  console.log("box-->" + box);
}
display();
// b 不能访问
console.log("b-->" + b);
```

2. 在函数中定义变量时，若没有加var关键字，使用之后自动变为全局变量

```
function fun(){
  a = 100;
}
fun();
alert(a);
```

3.10. 内置对象

```
Arguments          只在函数内部定义，保存了函数的实参
Array              数组对象
Date               日期对象，用来创建和获取日期
Math               数学对象
String             字符串对象，提供对字符串的一系列操作
```

3.10.1. String

```
○ charAt(idx)     返回指定位置处的字符
○ indexOf(Chr)      返回指定子字符串的位置，从左到右。找不到返回-1
○ substr(m,n)     返回给定字符串中从m位置开始，取n个字符，如果参数n省略，则意味着取到字符串末尾。
○ substring(m,n)    返回给定字符串中从m位置开始，到n位置结束，如果参数n省略，则意味着取到字符串末尾。
○ toLowerCase()     将字符串中的字符全部转化成小写。
○ toUpperCase()     将字符串中的字符全部转化成大写。
○ length        属性，不是方法，返回字符串的长度。
```

3.10.2. Math

```
○ Math.random()     随机数
○ Math.ceil()       向上取整，大于最大整数
○ Math.floor()      向小取整，小于最小整数String
```

3.10.3. Date

```
// 获取日期
○ getFullYear()     年
○ getMonth()      月
○ getDate()       日
```

- getHours()　　时
- getMinutes()　　分
- getSeconds()　　秒

```
// 设置日期
```

- setYear()
- setMonth()
- setDate()
- setHours()
- setMinutes()
- setSeconds()
- toLoacaleString() 转换成本地时间字符串

说明：

1. getMonth()：得到的值：0~11（1月~12月）
2. setMonth()：设置值时0~11
3. toLocaleString()：可根据本地时间把 Date 对象转换为字符串，并返回结果。

3.11. 对象

对象（object）是 JavaScript 的核心概念，也是最重要的数据类型。JavaScript 的所有数据都可以被视为对象。JavaScript 提供多个内建对象，比如 String、Date、Array 等等。对象是带有属性和方法的特殊数据类型。

简单说，所谓对象，就是一种无序的数据集合，由若干个"键值对"（key-value）构成。通过JavaScript我们可以创建自己的对象。 JavaScript对象满足的这种"键值对"的格式我们称为JSON格式，以后会见得非常多，即伟大的JSON对象。

{键:值, 键2:值2,...}

3.11.1. 对象的创建

JS 创建自定义对象，主要通过三种方式：字面量形式创建对象、通过new Object对象创建 、通过Object对象的create方法创建对象。

3.11.1.1. 字面量形式创建

```
var 对象名 = {};//创建一个空的对象
var 对象名 = {键:值,键2:值2,...}

var obj = {
            'name' : 'hello',
            age : 12,
            sayHello : function () {
               console.log("我是对象中的方法");
            },
          courses : {
               javase : 4,
                 javascript : 3
          },
            isLike : true,
            members : [
                 {name : "小红",age : 20},
                 {name : "小绿",age : 22},
                 {name : "小蓝",age : 27},
                 {name : "小黄"}
            ]
};
```

3.11.1.2. 通过new Object创建

```
var 对象名 = new Object(); // 创建一个空的对象

var obj = new Object();
obj.name = 'zs';
obj.age = 18;
console.log(obj);
```

3.11.1.3. 通过Object对象的create方法创建

```
var 对象名 = Object.create(null);
var obj = Object.create(null);
obj.name = 'ls';
obj.gender = true
console.log(obj);

var objn = Object.create(obj);
objn.age = 18;
console.log(objn);
console.log(objn.gender)
```

3.11.2. 对象的序列化和反序列化

序列化即将JS对象序列化为字符串，反序列化即将字符串反序列化为JS对象。JS中通过调用JSON方法，可以将对象序列化成字符串，也可以将字符串反序列化成对象 。

```
// 序列化对象，将对象转为字符串
JSON.stringify(object);
```

```
// 反序列化，将一个Json字符串转换为对象。
JSON.parse(jsonStr);
```

3.11.3. this

this是JavaScript语言的一个关键字。

它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用。

随着函数使用场合的不同，this的值会发生变化。但是有一个总的原则，那就是this指的是，调用函数的那个对象。

3.11.3.1. 在函数中使用this

在函数中使用this属于全局性调用，代表全局对象，通过window对象来访问。

```
function test () {
  this.x = 1;
  console.log(this.x);
}
test();
console.log(x); // 相当于定义在全局对象上的属性

var x = 10;
console.log(x)  // 10
function test (){
    console.log(this.x)  // 10
    this.x = 1;
    console.log(this.x) // 1
    console.log(this)
}

test();
console.log(x); // 1
console.log(this);
```

3.11.3.2. 在对象中使用this

在对象中的函数使用this，代表当前的上级对象。

```
var obj = {
  name : '张三',
  age : 20,
  sayHello : function () {
    console.log(this.name)
    console.log(this)
  }
}
obj.sayHello();
```