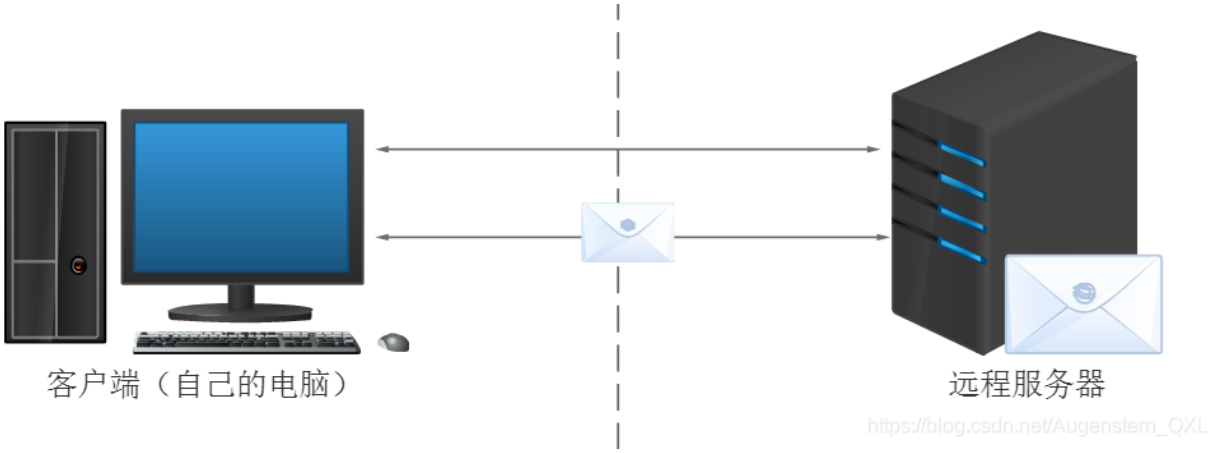
✍初识JavaScirpt

- JavaScript 是世界上最流行的语言之一，是一种运行在客户端的脚本语言 （Script 是脚本的意思）
- 脚本语言：不需要编译，运行过程中由 js 解释器(js 引擎）逐行来进行解释并执行
- 现在也可以基于 Node.js 技术进行服务器端编程



✍浏览器执行JS简介

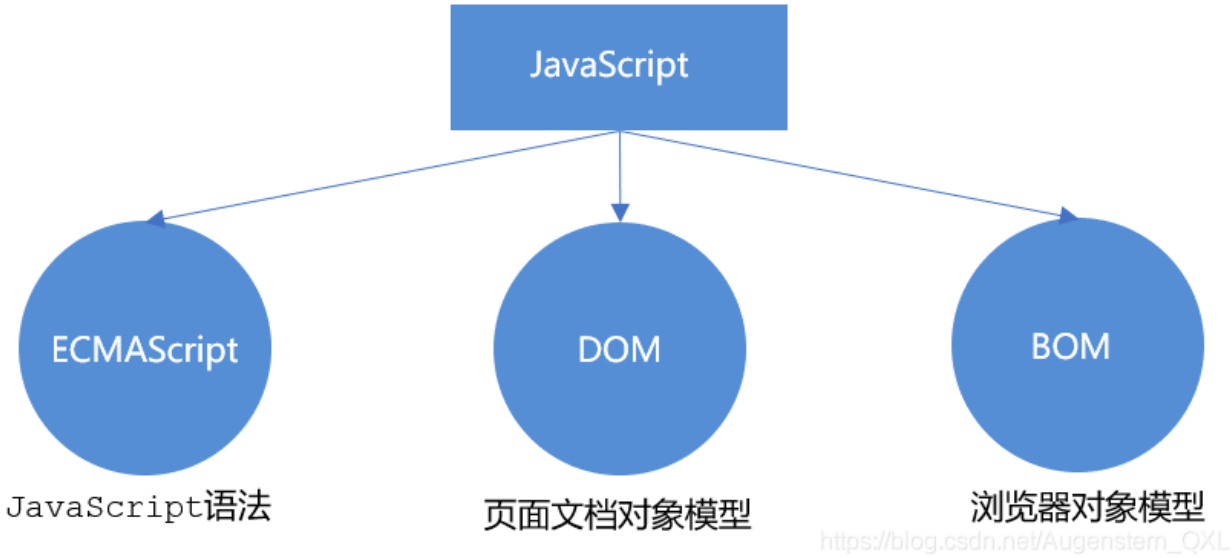
浏览器分成两部分：渲染引擎和 JS 引擎

- 渲染引擎：用来解析HTML与CSS，俗称内核，比如 chrome 浏览器的 blink ，老版本的 webkit
- JS 引擎：也称为 JS 解释器。 用来读取网页中的JavaScript代码，对其处理后运行，比如 chrome 浏览器的 V8

浏览器本身并不会执行JS代码，而是通过内置 JavaScript 引擎(解释器) 来执行 JS 代码 。JS 引擎执行代码时逐行解释每一句源码（转换为机器语言），然后由计算机去执行，所以 JavaScript 语言归为脚本语言，会逐行解释执行。

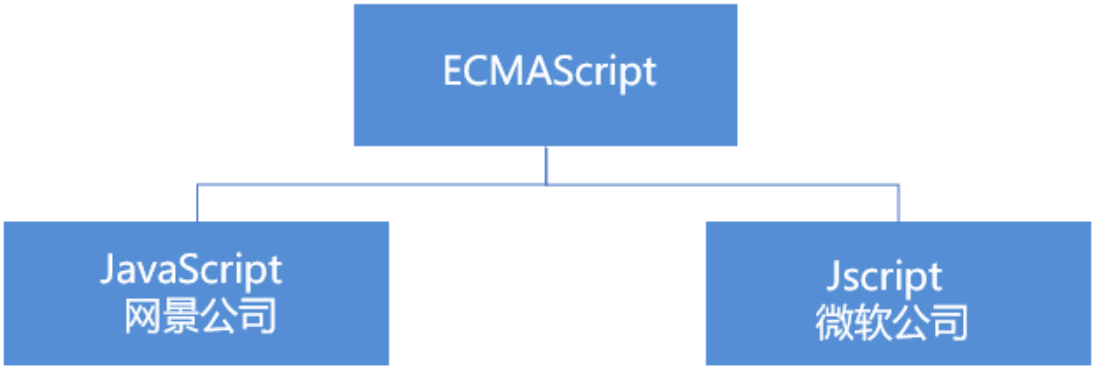
✍JS的组成

JavaScript 包括 ECMAScript、DOM、BOM



✍ECMAScript

ECMAScript 是由ECMA 国际（原欧洲计算机制造商协会）进行标准化的一门编程语言，这种语言在万维网上应用广泛，它往往被称为 JavaScript 或 JScript，但实际上后两者是 ECMAScript 语言的实现和扩展。



ECMAScript：ECMAScript 规定了JS的编程语法和基础核心知识，是所有浏览器厂商共同遵守的一套JS语法工业标准。

🔥DOM文档对象模型

文档对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展标记语言的标准编程接口。通过 DOM 提供的接口可以对页面上的各种元素进行操作（大小、位置、颜色等）。

BOM浏览器对象模型

BOM (Browser Object Model，简称BOM) 是指浏览器对象模型，它提供了独立于内容的、可以与浏览器窗口进行互动的对象结构。通过BOM可以操作浏览器窗口，比如弹出框、控制浏览器跳转、获取分辨率等。

1、JS初体验

1.1、行内式JS

```
<input type="button" value="点我试试"
onclink="javascript:alert('Hello World')" />
```

1. 可以将单行或少量JS代码写在HTML标签的事件属性中(以on开头的属性)，如： onclink
2. 注意单双引号的使用：在HTML中我们推荐使用**双引号**，JS中我们推荐使用**单引号**
3. 可读性差，在 HTML 中编入 JS 大量代码时，不方便阅读
4. 特殊情况下使用

1.2、内嵌式JS

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

- 可以将多行JS代码写到 `<script>` 标签中
- 内嵌 JS 是学习时常用的方式

1.3、外部JS

```
<script src="my.js"></script>
```

1. 利于HTML页面代码结构化，把单独JS代码独立到HTML页面之外，既美观，又方便
2. 引用外部JS文件的script标签中间不可以写代码
3. 适合于JS代码量比较大的情况

2、JS基本语法

2.1、注释

2.1.1、单行注释

```
//   单行注释
```

- 快捷键 `ctrl + /`

2.1.2、多行注释

```
/*
    多行注释
*/
```

- 快捷键 `shift + alt + a`
- vscode中修改快捷键方式：vscode➡ 首选项按钮➡ 键盘快捷方式 ➡ 查找原来的快捷键➡ 修改为新的快捷键➡ 回车确认

2.2、输入输出语句

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

- `alert()` 主要用来显示消息给用户
- `console.log()` 用来给程序员看自己运行时的消息

2.3、变量

- 变量是用于存放数据的**容器**，我们通过**变量名**获取数据，甚至数据可以修改
- **本质：*变量是程序在*内存**中申请的一块用来存放数据的空间

2.3.1、变量初始化

1. var是一个JS关键字，用来声明变量(variable变量的意思)。使用该关键字声明变量后，计算机会自动为变量分配内存空间。
2. age 是程序员定义的变量名，我们要通过变量名来访问内存中分配的空间

```
//声明变量同时赋值为18
var age = 18;
//同时声明多个变量时，只需要写一个 var， 多个变量名之间使用英文逗号隔开。

var age = 18, address ='火影村',salary = 15000;
```

2.3.2、声明变量特殊情况

情况	说明	结果
var age; console.log(age);	只声明，不赋值	undefined
console.log(age)	不声明 不赋值 直接使用	报错
age = 10;console.log(age);	不声明 只赋值	10

2.3.3、变量的命名规范

1. 由字母(A-Z,a-z)，数字(0-9)，下划线(_)，美元符号($)组成，如:usrAge,num01,__name
2. 严格区分大小写。 `var app;` 和 `var App;` 是两个变量
3. 不能以数字开头。
4. 不能是关键字，保留字。例如： `var,for,while`
5. 遵循驼峰命名法。首字母小写，后面单词的首字母需要大写。 `myFirstName`
6. 推荐翻译网站：有道 爱词霸

2.4、数据类型

JavaScript 是一种弱类型或者说动态语言。这意味着不用提前声明变量的类型，在程序运行过程中，类型会被自动确定。

```
var age = 10;             //这是一个数字型
var areYouOk = '使得';     //这是一个字符串
```

- 在代码运行时，变量的数据类型是由 JS引擎 根据 = 右边变量值的数据类型来判断 的，运行完毕之后， 变量就确定了数据类型。
- JavaScript 拥有动态类型，同时也意味着相同的变量可用作不同的类型

```
var x = 6;       //x为数字
var x = "Bill"; //x为字符串
```

JS 把数据类型分为两类：

- 基本数据类型(Number,String,Boolean,Undefined,Null)

- 复杂数据类型(Object)

2.4.1、基本数据类型

简单数据类型	说明	默认值
Number	数字型，包含整型值和浮点型值，如21，0.21	0
Boolean	布尔值类型，如true，false ，等价于1和0	false
Undefined	var a; 声明了变量a但是没有赋值，此时a=undefined	undefined（未定义的）
string	字符串类型，如“张三”	“”
Null	var a = null;声明了变量a为空值	null

2.4.2、数字型Number

JavaScript 数字类型既可以用来保存整数值，也可以保存小数(浮点数）。

```
var age = 12;        //整数
var Age = 21.3747;   //小数
```

2.4.2、数字型进制

最常见的进制有二进制、八进制、十进制、十六进制。

```
// 1.八进制数字序列范围：0~7
var num1 = 07;       //对应十进制的7
var Num2 = 019;      //对应十进制的19
var num3 = 08;       //对应十进制的8


// 2.十六进制数字序列范围：0~9以及A~F
var num = 0xA;
```

- **在JS中八进制前面加0，十六进制前面加 0x**

①数字型范围

- JS中数值的最大值：`Number.MAX_VALUE`
- JS中数值的最小值：`Number.MIN_VALUE`

```
consol.log(Number.MAX_VALUE);
consol.log(Number.MIN_VALUE);
```

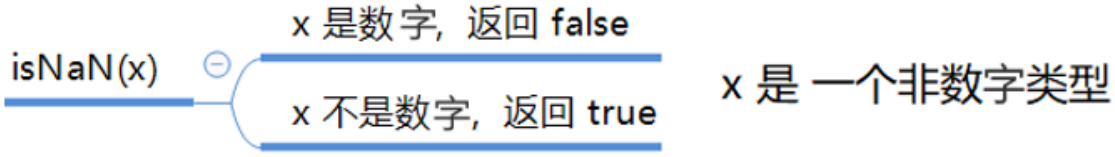
②数字型的三个特殊值

```
alert(Infinity);    //Infinity(无穷大)
alert(-Infinity);   //-Infinity(无穷小)
alert(NaN);         //NaN - Not a Number ,代表任何一个非数值
```

- Infinity ，代表无穷大，大于任何数值
- -Infinity ，代表无穷小，小于任何数值
- Nan ，Not a Number，代表一个非数值

③isNaN

这个方法用来判断非数字，并且返回一个值，如果是数字返回的是false，如果不是数字返回的是true




```
var userAge = 21;
var isOk = isNan(userAge);
console.log(isOk);      //false,21不是一个非数字

var userName = "andy";
console.log(isNan(userName));   //true,"andy"是一个非数字
```

2.4.3、字符串型String

字符串型可以是引号中的任意文本，其语法为 “**双引号**” 和 ”**单引号**”

```
var strMsg = "我爱北京天安门~";          //使用双引号表示字符串
var strMsg = '我爱北京';                 //使用单引号表示字符串
```

因为 HTML 标签里面的属性使用的是双引号，JS 这里我们更推荐**使用单引号**。

①字符串引号嵌套

JS可以用 **单引号嵌套双引号**，或者用 **双引号嵌套单引号（外双内单，外单内双）**

```
var strMsg ='我是一个“高富帅”' //可以用 ' ' 包含 " "
var strMsg2 ="我是'高富帅'" //可以用" " 包含 ''
```

②字符串转义符

类似HTML里面的特殊字符，字符串中也有特殊字符，我们称之为转义符。

转义符都是 \ 开头的，常用的转义符及其说明如下：

转义符	解释说明
\n	换行符，n是newline
\ \	斜杠\
\ ’	’ 单引号
\ “	“ 双引号
\ t	tab 缩进
\ b	空格，b是blank的意思

③字符串长度

字符串是由若干字符组成的，这些字符的数量就是字符串的长度。通过字符串的 length 属性可以获取整个字符串的长度。

```
//通过字符串的length属性可以获取整个字符串的长度
var strMsg = "我是高富帅! ";
alert(strMsg.length);     //显示6
```

④字符串的拼接

- 多个字符串之间可以使用 + 进行拼接，其拼接方式为 **字符串 + 任何类型 = 拼接之后的新字符串**
- 拼接前会把与字符串相加的任何类型转成字符串，再拼接成一个新的字符串

注意：字符串 + 任何类型 =拼接之后的新字符串

```
//1 字符串相加
alert('hello' + ' ' + 'World');  //hello World

//2 数值字符串相加
alert('100' + '100'); //100100

//3 数值字符串+数值
alert('12'+12); //1212

//4 数值+数值
alert(12+12); //24
```

- `+` 号总结口诀：数值相加，字符相连

```
var  age = 18;
console.log('我今年'+age+'岁');
console.log('我今年'+age+'岁');  //引引加加，最终也是上面的形式
```

⑤字符串拼接加强

```
console.log('Pink老师' + 18);          //只要有字符就会相连
var age = 18;
// console.log('Pink老师age岁了');       //这样不行,会输出 "Pink老师age岁了"

console.log('Pink老师' + age);          // Pink老师18
console.log('Pink老师' + age + '岁啦'); // Pink老师18岁啦
```

- 我们经常会将字符串和变量来拼接，因为变量可以很方便地修改里面的值
- 变量是不能添加引号的，因为加引号的变量会变成字符串
- 如果变量两侧都有字符串拼接，口诀==“引引加加 ”，删掉数字==变量写加中间

2.4.4、布尔型Boolean

- 布尔类型有两个值：true 和 false ，其中 true 表示真（对），而 false 表示假（错）。
- 布尔型和数字型相加的时候， true 的值为 1 ，false 的值为 0。

```
var flag = true;
console.log(flag + 1); // 2 true当加法来看当1来看，flase当0来看
```

2.4.5、undefined未定义

- 一个**声明后没有被赋值**的变量会有一个默认值 undefined (如果进行相连或者相加时，注意结果)

```
// 如果一个变量声明未赋值，就是undefined 未定义数据类型
var str;
console.log(str);                  //undefined
var variable = undefined;
console.log(variable + 'Pink'); //undefinedPink
console.log(variable + 18); //NaN
```

1.undefined 和 字符串 相加，会拼接字符串

2.undefined 和 数字相加，最后结果是**NaN**

2.4.6、空值null

- 一个声明变量给 null 值，里面存的值为空

```
var space = null;
console.log(space + 'pink'); //nullpink
console.llog(space + 1); // 1
```

2.4.7、typeof

- typeof 可用来获取检测变量的数据类型

```
var num = 18;
console.log(typeof num) // 结果 number
```

不同类型的返回值

类型	例	结果
string	typeof “小白”	“string”
number	typeof 18	“number”
boolean	typeof true	“boolean”
undefined	typeof undefined	“undefined”
null	typeof null	“object”

2.4.8、字面量

字面量是在源代码中一个固定值的表示法，通俗来说，就是字面量表示如何表达这个值。

- 数字字面量：8，9，10
- 字符串字面量：‘大前端’，‘后端’
- 布尔字面量：true、false

通过控制台的颜色判断属于哪种数据类型

黑色	字符串
蓝色	数值
灰色	undefined 和 null

2.5、数据类型转换

使用表单、prompt 获取过来的数据默认是字符串类型的，此时就不能直接简单的进行加法运算，而需要转换变量的数据类型。通俗来说，**就是把一种数据类型的变量转换成另外一种数据类型**。

我们通常会实现3种方式的转换：

- 转换为字符串类型
- 转换为数字型
- 转换为布尔型

①转换为字符串型

方式	说明	案例
toString()	转成字符串	var num = 1; alert(num.toString());
String()强制转换	转成字符串	var num = 1; alert(String(num));
加号拼接字符串	和字符串拼接的结果都是字符串	var num =1; alert(num+“我是字符串”);

```
//1.把数字型转换为字符串型 toString()  变量.toString()
var num = 10;
var str = num.toString();
console.log(str);

//2.强制转换
console.log(String(num));
```

- toString() 和 String() 使用方式不一样

- 三种转换方式，我们更喜欢用第三种加号拼接字符串转换方式，这一方式也称为隐士转换

②转换为数字型

方式	说明	案例
parseInt(string)函数	将string类型转成整数数值型	parseInt(‘78’)
parseFloat(string)函数	将string类型转成浮点数数值型	parseFloat(‘78.21’)
Number()强制转换函数	将string类型转换为数值型	Number(‘12’)
js 隐式转换(- * /)	利用算术运算隐式转换为数值型	‘12’-0

```
// 1.parseInt()
var age =prompt('请输入您的年龄');
consolo.log(parseInt(age));  //数字型18
consolo.log(parseInt('3.14'));  //3取整
consolo.log(parseInt('3.94'));  //3,不会四舍五入
consolo.log(parseInt('120px'));  //120,会去掉单位

// 2.parseFloat()
console.log(parseFloat('3.14'));  //3.14
consolo.log(parseFloat('120px'));  //120,会去掉单位


// 3.利用Number(变量)
var str ='123';
console.log(Number(str));
console.log(Number('12'));

// 4.利用了算术运算 - * /   隐式转换
console.log('12'-0);  // 12
console.log('123' - '120');  //3
console.log('123' * 1);  // 123
```

1.注意 parseInt 和 parseFloat ，这两个是重点

2.隐式转换是我们在进行算数运算的时候，JS自动转换了数据类型

③转换为布尔型

方法	说明	案例
Boolean()函数	其他类型转成布尔值	Boolean(‘true’);

- 代表空，否定的值会被转换为false，如 ’ ’ , 0, NaN , null , undefined

- 其余的值都会被被转换为true

```
console.log(Boolean('')); //false
console.log(Boolean(0));  //false
console.log(Boolean(NaN)); //false
console.log(Boolean(null)); //false
console.log(Boolean(undefined)); //false
console.log(Boolean('小白')); //true
console.log(Boolean(12));   //true
```

2.6、运算符

运算符（operator）也被称为**操作符**，是用于实现赋值、比较和执行算数运算等功能的符号

JavaScript 中常用的运算符有：

- 算数运算符
- 递增和递减运算符
- 比较运算符
- 逻辑运算符
- 赋值运算符

2.6.1、算术运算符

概念：算术运算使用的符号，用于执行两个变量或值的算术运算。

运算符	描述	实例
+	加	10 + 20 =30
-	减	10 - 20 =-10
*	乘	10 * 20 =200
/	除	10 / 20 =0.5
%	取余数（取模）	返回出发的余数 9 % 2 =1

2.6.2、浮点数的精度问题

浮点数值的最高精度是17位小数，但在进行算数计算时其精确度远远不如整数

```
var result = 0.1 +0.2; //结果不是0.3，
0.30000000000000004
console.log(0.07 * 100); //结果不是7，而是
7.000000000000001
```

所以不要直接判断两个浮点数是否相等

2.6.3、递增和递减运算符

递增（++）

递减（- -）

放在变量前面时，我们称为**前置递增(递减)运算符**

放在变量后面时，我们称为**后置递增(递减)运算符**

注意：递增和递减运算符必须和变量配合使用。

①前置递增运算符

++num num = num + 1

使用口诀:**先自加，后返回值**

```
var num = 10;
alert (++num + 10); // 21
```

先自加 10+1=11，返回11，此时num=11

②后置递增运算符

num ++ num = num +1

使用口诀:**先返回原值，后自加**

```
var num = 10;
alert(10 + num++); // 20
```

③小结

- 前置递增和后置递增运算符可以简化代码的编写，让变量的值 + 1 比以前写法更简单
- 单独使用时，运行结果相同，与其他代码联用时，执行结果会不同
- 开发时，大多使用后置递增/减，并且代码独占一行

2.6.4、比较(关系)运算符

比较运算符是**两个数据进行比较时所使用的运算符**，比较运算后，会**返回一个布尔值**(true / false)作为比较运算的结果。

运算符名称	说明	案例	结果
<	小于号	1 < 2	true
>	大于号	1 > 2	false
>=	大于等于号(大于或者等于)	2 >= 2	true
<=	小于等于号(小于或者等于)	3 <= 2	false
==	判等号(会转型)	37 == 37	true
!=	不等号	37 != 37	false
=== !==	全等 要求值和数据类型都一致	37 === '37'	false

①===== 小结

符号	作用	用法
=	赋值	把右边给左边
==	判断	判断两边值是否相等(注意此时有隐士转换)
===	全等	判断两边的值和数据类型是否完全相同

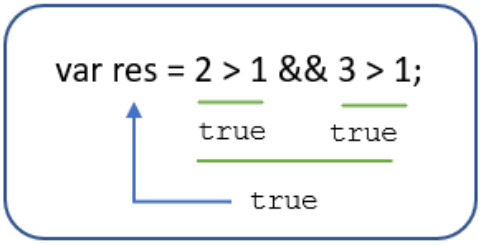
```
console.log(18 == '18');        //true
console.log(18 === '18');       //false
```

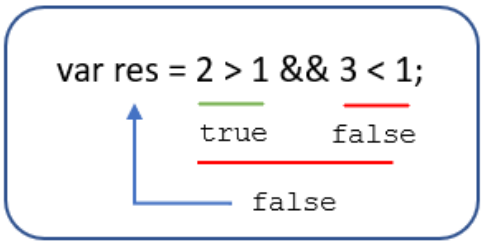
2.6.5、逻辑运算符

逻辑运算符是用来进行布尔值运算的运算符，其返回值也是布尔值

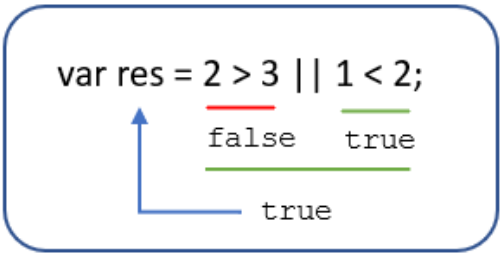
逻辑运算符	说明	案例
&&	“逻辑与”，简称"与" and	true && false
||	“逻辑或”，简称"或" or	true || false
!	“逻辑非”，简称"非" not	！true

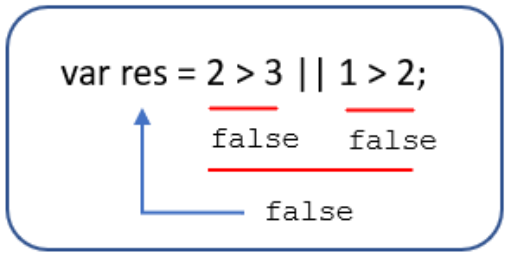
逻辑与：两边都是 true才返回 true，否则返回 false





逻辑或：两边都为 false 才返回 false，否则都为true





逻辑非：逻辑非（!）也叫作取反符，用来取一个布尔值相反的值，如 true 的相反值是 false

```
var isOk = !true;
console.log(isOk);  // false
//逻辑非（!）也叫作取反符，用来取一个布尔值相反的值，如 true 的相反值是 false
```

2.6.5.1、短路运算(逻辑中断)

短路运算的原理：当有多个表达式（值）时,左边的表达式值可以确定结果时,就不再继续运算右边的表达式的值

①逻辑与

- 语法：表达式1 && 表达式2
- 如果第一个表达式的值为真，则返回表达式2
- 如果第一个表达式的值为假，则返回表达式1

```
console.log(123 && 456);    //456
console.log(0 && 456);      //0
console.log(123 && 456 && 789);  //789
```

②逻辑或

- 语法：表达式1 || 表达式2
- 如果第一个表达式的值为真，则返回表达式1
- 如果第一个表达式的值为假，则返回表达式2

```
console.log(123 || 456); //123
console.log(0 || 456);   //456
console.log(123 || 456 || 789);  //123
123
var num = 0;
console.log(123 || num++);
// 先返回在加，相当于 (123 || 0)
console.log(num);    // 123
```

2.6.6、赋值运算符

概念：用来把数据赋值给变量的运算符。

赋值运算符	说明	案例
=	直接赋值	var usrName = ‘我是值’
+= ，-=	加，减一个数后再赋值	var age = 10； age+=5；//15
*=，/=，%=	成，除，取模后再赋值	var age = 2; age*=5; //10

```
var age = 10;
age += 5;  // 相当于 age = age + 5;
age -= 5;  // 相当于 age = age - 5;
age *= 10; // 相当于 age = age * 10;
```

2.6.7、运算符优先级

优先级	运算符	顺序
1	小括号	()
2	一元运算符	++ - ！
3	算数运算符	**先 * / 后 + -**
4	关系运算符	**>, >= , < , <=,**
5	相等运算符	，！=，=，！==
6	逻辑运算符	**先 && 后 ||（先与后或）**
7	赋值运算符	=
8	逗号运算符	,

1.一元运算符里面的**逻辑非**优先级很高

2.**逻辑与** 比 **逻辑或** 优先级高

3.练习题

```
console.log( 4 >= 6 || '人' != '阿凡达' && !(12 * 2
== 144) && true)    // true
1
var a = 3 > 5 && 2 < 7 && 3 == 4;
console.log(a);     //false

var b = 3 <= 4 || 3 > 1 || 3 != 2;
console.log(b);     //true

var c = 2 === "2";
console.log(c);     //false

var d = !c || b && a ;
console.log(d);     //true
```

2.7、流程控制

流程控制主要有三种结构，分别是顺序结构、分支结构和循环结构，这三种结构代表三种代码执行的顺序

2.7.1、分支结构

JS 语言提供了两种分支结构语句：**JS 语句 switch语句**

①if语句

```
// 条件成立执行代码，否则什么也不做
if (条件表达式) {
    //条件成立执行的代码语句
}
```

案例：进入网吧

弹出一个输入框，要求用户输入年龄，如果年龄大于等于 18 岁，允许进网吧

```
var usrAge = prompt('请输入您的年龄:');
if(usrAge >= 18)
{
      alert('您的年龄合法，欢迎来到老子网吧享受学习的乐
趣! ');
}
```

②if else 语句

```
// 条件成立，执行if里面代码，否则执行else里面的代码
if(条件表达式)
{
    //[如果]条件成立执行的代码
}
else
    {
        //[否则]执行的代码
    }
```

案例：判断闰年

接收用户输入的年份，如果是闰年就弹出闰年，否则弹出是平年

算法：能被4整除且不能整除100的为闰年（如2004年就是闰年，1901年不是闰年）或者能够被 400 整除的就是闰年

```
var year = prompt('请输入年份');

if (year % 4 == 0 && year % 100 !=0 || year % 400
==0)
{
   alert('这个年份是闰年');
}
else
{
  alert('这个年份是平年');
}
```

③if else if 语句

```
if(条件表达式1)
{
  语句1;
}
else if(条件表达式2)
{
   语句2;
}
else if(条件表达式3)
{
  语句3;
}
else
```

```
{
    //上述条件都不成立执行此处代码
}
```

案例:接收用户输入的分数，根据分数输出对应的等级字母 A、B、C、D、E

其中：

1. 90分(含)以上 ，输出：A
2. 80分(含)~ 90 分(不含)，输出：B
3. 70分(含)~ 80 分(不含)，输出：C
4. 60分(含)~ 70 分(不含)，输出：D
5. 60分(不含) 以下，输出： E

```
 var score = prompt('请您输入分数:');
        if (score >= 90) {
            alert('宝贝，你是我的骄傲');
        } else if (score >= 80) {
            alert('宝贝，你已经很出色了');
        } else if (score >= 70) {
            alert('你要继续加油喽');
        } else if (score >= 60) {
            alert('孩子，你很危险');
        } else {
            alert('可以再努力点吗，你很棒，但还不够
棒');
        }
```

2.7.2、三元表达式

- 语法结构 : 表达式1 ？表达式2 : 表达式3
- 执行思路

如果表达式1为true，则返回表达式2的值,如果表达式1为false，则返回表达式3的值

案例：数字补0

用户输入数字，如果数字小于10，则在前面补0，比如01，09，

如果数字大于10，则不需要补，比如20

```
var figuer = prompt('请输入0~59之间的一个数字');
var result = figuer < 10 ? '0' + figuer : figue
        alert(result);
```

2.7.3、switch

```
switch(表达式){
  case value1:
     //表达式等于 value1 时要执行的代码
     break;
  case value2:
     //表达式等于value2 时要执行的代码
     break;
  default:
     //表达式不等于任何一个value时要执行的代码

}
```

- switch： 开关 转换， case： 小例子 选项
- 关键字 switch 后面**括号内**可以是**表达式或值**， 通常是一个**变量**
- 关键字 case , 后跟一个选项的表达式或值，**后面跟一个冒号**
- switch 表达式的值会与结构中的 case 的值做比较
- 如果存在匹配**全等**(===) ，则与该 case 关联的代码块会被执行，并在遇到 **break 时停止**，整个 switch 语句代码执行结束
- 如果所有的 case 的值都和表达式的值不匹配，则执行 default 里的代码
- **执行case 里面的语句时，如果没有break，则继续执行下一个case里面的语句**

```
// 用户在弹出框里面输入一个水果，如果有就弹出该水果的价格，  如果没有该水果就弹出“没有此水果”
        var fruit = prompt('请您输入查询的苹果');
        switch (fruit) {
            case '苹果':
                alert('苹果的价格为3.5元/千克');
                break;
            case '香蕉':
                alert('香蕉的价格为3元/千克');
                break;
            default:
                alert('没有这种水果');
        }
```

3、断点调试

1. 浏览器中按 F12-> sources -->找到需要调试的文件->在程序的某一行设置断点(在行数点一下)
2. 刷新浏览器
3. Watch: 监视，通过watch可以监视变量的值的变化，非常的常用
4. F11: 程序单步执行，让程序一行一行的执行，这个时候，观察watch中变量的值的变化

4、循环

4.1、for循环

在程序中，一组被重复执行的语句被称之为**循环体**，能否继续重复执行，取决于循环的**终止条件**。由循环体及循环的终止条件组成的语句，被称之为**循环语句**

```
for(初始化变量;条件表达式;操作表达式)
{
    //循环体
}
```

1.输入10句"娘子晚安哈！"

```
//基本写法
for(var i = 1; i<=10; i++  )
    {
        console.log('娘子晚安哈');
    }
// 用户输入次数
var num = prompt('请输入次数:');
for(var i = 1; i<= num ;i++)
    {
        console.log('娘子晚安哈');
    }
```

2.求1-100之间所有整数的累加和

```
// 求1-100所以的整数和
var sum = 0;
for (var i = 1; i <= 100; i++) {
    var sum = sum + i;
}
console.log(sum);
```

3.求1-100之间所有数的平均值

```
 // 3.求1-100之间所有数的平均值
var sum = 0;
for (var i = 1; i <= 100; i++) {
    var sum = sum + i;
}
console.log(sum / 100);
```

4.求1-100之间所有偶数和奇数的和

```
//    4.求1-100之间所有偶数和奇数的和
var sum1 = 0;
var sum2 = 0;
for (var i = 1; i <= 100; i++) {
    if (i % 2 == 0) {
        sum1 = sum1 + i;
    } else {
        sum2 = sum2 + i;
    }
}
console.log('偶数和为' + sum1);
console.log('奇数和为' + sum2);
```

5.求1-100之间所有能被3整除的数字的和

```
// 5.求1-100之间所有能被3整除的数字的和
var sum = 0;
for (var i = 1; i <= 100; i++) {
    if (i % 3 == 0) {
        sum += i;
    }
}
console.log(sum);
```

6.要求用户输入班级人数，之后依次输入每个学生的成绩，最后打印出该班级总的成绩以及平均成绩。

```
var num = prompt('请输入班级总的人数:'); // num 班级总的人数
var sum = 0; // 总成绩
var average = 0; // 平均成绩
for (var i = 1; i <= num; i++) {
     var score = prompt('请输入第' + i + '个学生的成绩');
    //这里接收的是str，必须转换为数值
     sum = sum + parseFloat(score);
}
average = sum / num;
alert('班级总的成绩是: ' + sum);
alert('班级总的平均成绩是: ' + average);
```

7.一行打印5个星星

我们采取追加字符串的方式，这样可以打印到控制台上

```
var star = '';
for (var i = 1; i <= 5; i++) {
     star += '☆';
 }
console.log(star);
```

4.2、双重for循环

循环嵌套是指在一个循环语句中再定义一个循环语句的语法结构，例如在for循环语句中，可以再嵌套一个for 循环，这样的 for 循环语句我们称之为双重for循环。

```
for(外循环的初始;外循环的条件;外形循环的操作表达式){
    for(内循环的初始;内循环的条件;内循环的操作表达式){
        需执行的代码;
    }
}
```

- 内层循环可以看做外层循环的语句
- 内层循环执行的顺序也要遵循 for 循环的执行顺序
- 外层循环执行一次，内层循环要执行全部次数

①打印五行五列星星

核心：

- 内层循环负责一行打印五个星星
- 外层循环负责打印五行

```
var star = '';
for(var j = 1;j<=5;j++)
{
   for (var i = 1; i <= 5; i++)
   {
     star += '☆'
   }
    //每次满5个星星就加一次换行
    star +='\n'
}
console.log(star);
```

②打印n行n列的星星

要求用户输入行数和列数，之后在控制台打印出用户输入行数和列数的星星

```
var star = '';
var row = prompt('请输入行数');
var col = prompt('请输入列数');
for (var j = 1; j <= col; j++) {
    for (var i = 1; i <= row; i++) {
        star += '☆';
    }
    star += '\n';
}
console.log(star);
```

③打印倒三角形

- 一共有10行，但是每行的星星个数不一样，因此需要用到双重 for 循环
- 外层的 for 控制行数 i ，循环10次可以打印10行
- 内层的 for 控制每行的星星个数 j
- 核心算法： 每一行星星的个数： j = i ; j <= 10; j++
- 每行打印完毕后，都需要重新换一行

```
var star = '';
var row = prompt('请输入行数');
var col = prompt('请输入列数');
for (var i = 1; i <= row; i++) {
    for (var j = i; j <= col; j++) {
        star += '☆';
    }
    star += '\n';
}
console.log(star);
```

4.3、while循环

```
while(条件表达式){
  //循环体代码
}
```

执行思路：

- 先执行条件表达式，如果结果为 true，则执行循环体代码；如果为 false，则退出循环，执行后面代码
- 执行循环体代码
- 循环体代码执行完毕后，程序会继续判断执行条件表达式，如条件仍为true，则会继续执行循环体，直到循环条件为 false 时，整个循环过程才会结束

注意：

- 使用 while 循环时一定要注意，它必须要有退出条件，否则会称为死循环
- while 循环和 for 循环的不同之处在于 while 循环可以做较为复杂的条件判断，比如判断用户名和密码

①打印人的一生

从1岁到99岁

```
var age = 0;
while (age <= 100) {
    age++;
    console.log('您今年' + age + '岁了');
}
```

②计算 1 ~ 100 之间所有整数的和

```
var figure = 1;
var sum = 0;
while (figure <= 100) {
     sum += figure;
      figure++;
}
console.log('1-100的整数和为' + sum);
```

4.4、do while循环

```
do {
  //循环体代码-条件表达式为true的时候重复执行循环一代码
}while(条件表达式);
123
```

执行思路：

1. 先执行一次循环体代码

2. 再执行表达式，如果结果为true，则继续执行循环体代码，如果为false，则退出循环，继续执行后面的代码
3. 先执行再判断循环体，**所以dowhile循环语句至少会执行一次循环体代码**

需求：弹出一个提示框， 你爱我吗？ 如果输入我爱你，就提示结束，否则，一直询问

```
do {
    var love = prompt('你爱我吗？ ');
} while (love != '我爱你');
    alert('登录成功');
1234
```

4.5、continue 关键字

continue 关键字用于**立即跳出本次循环，继续下一次循环**（本次循环体中 continue 之后的代码就会少执行一次）。

例如，吃5个包子，第3个有虫子，就扔掉第3个，继续吃第4个第5个包子

```
for (var i = 1; i <= 5; i++) {
 if (i == 3) {
     console.log('这个包子有虫子，扔掉');
     continue; // 跳出本次循环，跳出的是第3次循环
  }
  console.log('我正在吃第' + i + '个包子呢');
}
67
```

4.6、break关键字

break 关键字用于**立即跳出整个循环**

例如，吃5个包子，吃到第3个发现里面有半个虫子，其余的也不吃了

```
for (var i = 1; i <= 5; i++) {
   if (i == 3) {
       break; // 直接退出整个for 循环，跳到整个for下面的语句
   }
   console.log('我正在吃第' + i + '个包子呢');
 }

67
```

5、数组

数组(Array)是指一组数据的集合，其中的每个数据被称作元素，在数组中可以存放任意类型的元素。

数组是一种将一组数据存储在单个变量名下的优雅方式。

```
//普通变量一次只能存储一个值
var num = 10;
//数组一次可以存储多个值
var arr =[1,2,3,4,5];
1234
```

5.1、创建数组

JavaScript 中创建数组有两种方式：

- 利用 new 创建数组
- 利用数组字面量创建数组

①利用 new 创建数组

```
var 数组名 = new Array();
var arr = new Array(); //创建一个新的空数组
12
```

- 这种方式暂且了解，等学完对象再看
- 注意 `Array()`，A要大写

②利用数组字面量创建数组

```
// 1.利用数组字面量方式创建空的数组
var 数组名 =[];
// 2.使用数组字面量方式创建带初始值的数组
var 数组名 =['小白','小黑','小黄','瑞奇'];
// 3.数组中可以存放任意类型的数据，例如字符串，数字，布尔值等
var arrStus =['小白', 12,true,28.9];
6
```

- 数组的字面量是方括号 `[]`
- 声明数组并赋值称为数组的初始化
- 这种字面量方式也是我们以后最多使用的方式

5.2、数组的索引（下标）

索引(下标)：用来访问数组元素的序号（数组下标从 0 开始）

```
//定义数组
var arrStus = [1,2,3];
//获取数组中的第2个元素
alert(arrStus[1]);
1234
```

5.3遍历数组

我们可以通过 for 循环索引遍历数组中的每一项

```
// 数组索引访问数组中的元素
var arr = ['red','green', 'blue'];
console.log(arr[0]) // red
console.log(arr[1]) // green
console.log(arr[2]) // blue

// for循环遍历数组
var arr = ['red','green', 'blue'];
for (var i = 0; i < arr.length; i++){
    console.log(arrStus[i]);
}
11
```

5.4、数组的长度

使用“数组名.length”可以访问数组元素的数量（数组长度）

```
var arrStus = [1,2,3];
alert(arrStus.length);  // 3
12
```

注意：

- 此处数组的长度是**数组元素的个数**，不要和**数组的索引号**混淆
- 当我们数组里面的元素个数发生了变化，这个 length 属性跟着一起变化

5.5、案例

1.请将 [“关羽”,“张飞”,“马超”,“赵云”,“黄忠”,“刘备”,“姜维”]; 数组里的元素依次打印到控制台

```
var arr = ["关羽","张飞","马超","赵云","黄忠","刘
备","姜维"];
// 遍历  从第一个到最后一个
for(var i = 0; i < arr.length; i++ )  {
   console.log( arr[i] );
}
```

2.求数组 [2,6,1,7, 4] 里面所有元素的和以及平均值

- ①声明一个求和变量 sum。
- ①遍历这个数组，把里面每个数组元素加到 sum 里面。
- ①用求和变量 sum 除以数组的长度就可以得到数组的平均值。

```
var arr = [2, 6, 1, 7, 4];
var sum = 0;
var average = 0;
for (var i = 0; i < arr.length; i++) {
    sum += arr[i];
}
average = sum / i; //此时i为5
//      average = sum / arr.length;
console.log('和为' + sum);
console.log('平均值为' + average);
```

3.求数组[2,6,1,77,52,25,7]中的最大值

- ①声明一个保存最大元素的变量 max。
- ②默认最大值可以取数组中的第一个元素。
- ③遍历这个数组，把里面每个数组元素和 max 相比较。
- ④如果这个数组元素大于max 就把这个数组元素存到 max 里面，否则继续下一轮比较。
- ⑤最后输出这个 max。

```
 var arr = [2, 6, 1, 77, 52, 25, 7];
        var max = arr[0];
        var temp;
        for (var i = 0; i < arr.length; i++) {
            if (max < arr[i]) {
                temp = max;
                max = arr[i];
                arr[i] = temp;
            }
        }
        console.log('最大值为' + max);
```

```
方法二：

var arrNum = [2,6,1,77,52,25,7];
var maxNum = arrNum[0]; // 用来保存最大元素,默认最大值是数组中的第一个元素
// 从0 开始循环数组里的每个元素
for(var i = 0;i< arrNum.length; i++){
    // 如果数组里当前循环的元素大于 maxNum，则保存这个元素和下标
    if(arrNum[i] > maxNum){
        maxNum = arrNum[i]; // 保存数值到变量 maxNum
    }
}

19202122232425
```

4.将数组 [‘red’, ‘green’, ‘blue’, ‘pink’] 里面的元素转换为字符串

思路：就是把里面的元素相加就好了，但是注意保证是字符相加

- ①需要一个新变量 str 用于存放转换完的字符串。
- ②遍历原来的数组，分别把里面数据取出来，加到字符串变量 str 里面。

```
var arr = ['red','green','blue','pink'];
var str ='';
for(var i = 0; i < arr.length; i++){
    str += arr[i];
}
console.log(str);
// redgreenbluepink
67
```

5.将数组 [‘red’, ‘green’, ‘blue’, ‘pink’] 转换为字符串，并且用 | 或其他符号分割

- ①需要一个新变量用于存放转换完的字符串 str。
- ①遍历原来的数组，分别把里面数据取出来，加到字符串里面。
- ①同时在后面多加一个分隔符。

```
var arr = ['red', 'green', 'blue', 'pink'];
var str = '';
var separator = '|';
for (var i = 0; i < arr.length; i++) {
    str += arr[i] + separator;
}
console.log(str);
// red|green|blue|pink
```

5.6、数组中新增元素

①通过修改 length 长度新增数组元素

- 可以通过修改 length 长度来实现数组扩容的目的
- length 属性是可读写的

```
var arr = ['red', 'green', 'blue', 'pink'];
arr.length = 7;
console.log(arr);
console.log(arr[4]);
console.log(arr[5]);
console.log(arr[6]);
6
```

其中索引号是 4，5，6 的空间没有给值，就是声明变量未给值，默认值就是 **undefined**

②通过修改数组索引新增数组元素

- 可以通过修改数组索引的方式追加数组元素
- 不能直接给数组名赋值，否则会覆盖掉以前的数据
- 这种方式也是我们最常用的一种方式

```
var arr = ['red', 'green', 'blue', 'pink'];
arr[4] = 'hotpink';
console.log(arr);
123
```

5.7、数组中新增元素

1.新建一个数组，里面存放10个整数（ 1~10），要求使用循环追加的方式输出：

[1,2,3,4,5,6,7,8,9,10]

- ①使用循环来追加数组。
- ②声明一个空数组 arr。

- ③循环中的计数器 i 可以作为数组元素存入。
- 由于数组的索引号是从0开始的， 因此计数器从 0 开始更合适，存入的数组元素要+1。

```
var arr = [];
for (var i = 0; i < 10; i++){
    arr[i] = i + 1;
}
console.log(arr);
```

2.将数组 [2, 0, 6, 1, 77, 0, 52, 0, 25, 7] 中大于等于 10 的元素选出来，放入新数组

- ①声明一个新的数组用于存放新数据。
- ②遍历原来的数组，找出大于等于 10 的元素。
- ③依次追加给新数组 newArr。

实现代码1：

```
var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
var newArr = [];
// 定义一个变量 用来计算 新数组的索引号
var j = 0;
for (var i = 0; i < arr.length; i++) {
    if (arr[i] >= 10) {
        // 给新数组
        newArr[j] = arr[i];
        // 索引号 不断自加
        j++;
    }
}
console.log(newArr);
```

实现代码2：

```
var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
var newArr = [];
for (var i = 0; i < arr.length; i++) {
    if (arr[i] >= 10) {
        // 给新数组
        newArr[newArr.length] = arr[i];
    }
}
console.log(newArr);
```

5.8、删除指定数组元素

将数组[2, 0, 6, 1, 77, 0, 52, 0, 25, 7]中的 0 去掉后，形成一个不包含 0 的新数组。

```
var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
var newArr = [];
for(var i = 0; i <arr.length; i++){
    if(arr[i] != 0){
        newArr[newArr.length] = arr[i];
    }
}
console.log(newArr);

//老师代码
var arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];
var newArr = [];    // 空数组的默认的长度为 0
// 定义一个变量 i 用来计算新数组的索引号
for (var i = 0; i < arr.length; i++) {
    // 找出大于 10 的数
    if (arr[i] != 0) {
        // 给新数组
        // 每次存入一个值，newArr长度都会 +1
        newArr[newArr.length] = arr[i];
    }
}
console.log(newArr);

14
```

5.9、翻转数组

将数组 [‘red’, ‘green’, ‘blue’, ‘pink’, ‘purple’] 的内容反过来存放

```
// 把旧数组索引号的第4个取过来(arr.length - 1),给新数组索引号第0个元素(newArr.length)

var arr = ['red','green','blue','pink','purple'];
var newArr = [];
for (var i = arr.length -1; i>=0; i--){
    newArr[newArr.length] = arr[i];
}
console.log(newArr);
```

5.10、数组排序

冒泡排序

将数组 [5, 4, 3, 2, 1]中的元素按照从小到大的顺序排序，输出： 1，2，3，4，5

```
var arr = [5,4,3,2,1];
for (var i = 0; i < arr.length-1; i++){ //外层循环管趟数，5个数共交换4躺
    for (var j = 0; j <= arr.length - i - 1; j++){
        //里层循环管每一趟交换的次数
        //前一个和后面一个数组元素相比较
        if(arr[j] > arr[j+1]){
            var temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
        }
    }
}
console.log(arr);
```

#

1、函数

函数：就是封装了一段**可被重复调用执行的代码块**。通过此代码块可以实现大量代码的重复使用。

1.1、函数的使用

函数在使用时分为两步：**声明函数**和**调用函数**

①声明函数

```
//声明函数
function 函数名(){
    //函数体代码
}
1234
```

- function 是声明函数的关键字,**必须小写**
- 由于函数一般是为了实现某个功能才定义的， 所以通常我们将函数名命名为动词，比如 getSum

②调用函数

```
//调用函数
函数名(); //通过调用函数名来执行函数体代码
12
```

- 调用的时候**千万不要忘记添加小括号**
- 口诀：函数不调用，自己不执行

注意：声明函数本身并不会执行代码，只有调用函数时才会执行函数体代码。

1.2、函数的封装

- 函数的封装是把一个或者多个功能通过**函数的方式**封装起来，对外只提供一个简单的函数接口

1.3、函数的参数

1.3.1、形参和实参

在声明函数时，可以在函数名称后面的小括号中添加一些参数，这些参数被称为**形参**，而在**调用该函数**时，同样也需要传递相应的参数，这些参数被称为**实参**。

参数	说明
形参	**形**式上的**参**数 **函数定义**的时候 传递的参数 当前并不知道是什么
实参	**实**际上的**参**数 **函数调用**的时候 传递的参数 实参是传递给形参的

参数的作用：在**函数内部**某些值不能固定，我们可以通过参数在**调用函数时传递不同的值**进去

```
// 带参数的函数声明
function 函数名(形参1, 形参2 , 形参3...) { // 可以定义任意多的参数，用逗号分隔
  // 函数体
}


// 带参数的函数调用
函数名(实参1, 实参2, 实参3...);
```

例如：利用函数求任意两个数的和

```
// 声明函数
function getSum(num1,num2){
    console.log(num1+num2)
}

// 调用函数
getSum(1,3) //4
getSum(6, 5) //11
```

- 函数调用的时候实参值是传递给形参的
- 形参简单理解为:**不用声明的变量**
- 实参和形参的多个参数之间用 `逗号(,)` 分隔，

1.3.2、形参和实参个数不匹配

参数个数	说明
实参个数等于形参个数	输出正确结果
实参个数多于形参个数	只取到形参的个数
实参个数小于形参个数	多的形参定义为undefined，结果为NaN

```
function sum(num1, num2) {
    console.log(num1 + num2);
}
sum(100, 200);              // 300，形参和实参个数相等，输出正确结果

sum(100, 400, 500, 700);    // 500，实参个数多于形参，只取到形参的个数

sum(200);                   // 实参个数少于形参，多的形参定义为undefined，结果为NaN
```

注意：在JavaScript中，形参的默认值是undefined

1.3.3、小结

- 函数可以带参数也可以不带参数
- 声明函数的时候，函数名括号里面的是形参，形参的默认值为 undefined
- 调用函数的时候，函数名括号里面的是实参
- 多个参数中间用逗号分隔
- 形参的个数可以和实参个数不匹配，但是结果不可预计，我们尽量要匹配

1.4、函数的返回值

1.4.1、return语句

有的时候，我们会希望函数将值返回给调用者，此时通过使用 return 语句就可以实现。

return 语句的语法格式如下：

```
// 声明函数
function 函数名 () {
    ...
    return  需要返回的值;
}
// 调用函数
函数名();    // 此时调用函数就可以得到函数体内return 后面的值
```

- 在使用 return 语句时，函数会停止执行，并返回指定的值
- 如果函数没有 return ，返回的值是 undefined

```
// 声明函数
function sum(){
    ...
    return  666;
}
// 调用函数
sum();       // 此时 sum 的值就等于666，因为 return 语句会把自身后面的值返回给调用者
```

1.4.2、return 终止函数

return 语句之后的代码不被执行

```
function add(num1, num2){
    //函数体
    return num1 + num2; // 注意：return 后的代码不执行
    alert('我不会被执行，因为前面有 return');
}
var resNum = add(21,6); // 调用函数，传入两个实参，并通过 resNum 接收函数返回值
alert(resNum);          // 27
```

1.4.3、return 的返回值

return **只能返回一个值**。如果用逗号隔开多个值，**以最后一个为准**

```
function add(num1, num2){
    //函数体
    return num1,num2;
}
var resNum = add(21,6); // 调用函数，传入两个实参，并通过 resNum 接收函数返回值
alert(resNum);          // 6
```

1.4.4、小结

函数都是有返回值的

1. 如果有 return ，则返回 return 后面的值
2. 如果没有 return，则返回 undefined

1.4.5、区别

break、continue、return 的区别

- `break` ： 结束当前循环体(如 for、while)
- `continue` ：跳出本次循环，继续执行下次循环(如for、while)
- `return` ：不仅可以退出循环，还能够返回 return 语句中的值，同时还可以结束当前的函数体内的代码

1.4.5、练习

1.利用函数求任意两个数的最大值

```
function getMax(num1, num2) {
    return num1 > num2 ? num1 : num2;
}
console.log(getMax(1, 2));
console.log(getMax(11, 2));
```

2.求数组 [5,2,99,101,67,77] 中的最大数值

```
//定义一个获取数组中最大数的函数
function getMaxFromArr(numArray){
    var maxNum = numArray[0];
    for(var i = 0; i < numArray.length;i++){
        if(numArray[i]>maxNum){
            maxNum = numArray[i];
        }
    }
    return maxNum;
}
var arrNum = [5,2,99,101,67,77];
var maxN = getMaxFromArr(arrNum);  //这个实参是个数组
alert('最大值为' + maxN);
```

3.创建一个函数，实现两个数之间的加减乘除运算，并将结果返回

```
var a = parseFloat(prompt('请输入第一个数'));
var b = parseFloat(prompt('请输入第二个数'));

function count(a,b){
    var arr = [a + b, a - b, a * b, a / b];
    return arr;
}
var result = count(a,b);
console.log(result)
```

1.5、arguments的使用

当我们不确定有多少个参数传递的时候，可以用 arguments 来获取。在 JavaScript 中，arguments 实际上它是当前函数的一个内置对象。所有函数都内置了一个 arguments 对象，arguments 对象中存储了传递的所有实参。

- **arguments**存放的是传递过来的实参
- **arguments展示形式是一个伪数组，因此可以进行遍历。伪数组具有以下特点**

①：具有 length 属性

②：按索引方式储存数据

③：不具有数组的 push , pop 等方法

```
// 函数声明
function fn() {
    console.log(arguments);  //里面存储了所有传递过来的实参
    console.log(arrguments.length); // 3
    console.log(arrguments[2]); // 3
}

// 函数调用
fn(1,2,3);
```

例如：利用函数求任意个数的最大值

```
 function maxValue() {
    var max = arguments[0];
    for (var i = 0; i < arguments.length; i++) {
        if (max < arguments[i]) {
            max = arguments[i];
        }
    }
    return max;
}
console.log(maxValue(2, 4, 5, 9)); // 9
console.log(maxValue(12, 4, 9)); // 12
```

、函数调用另外一个函数

因为每个函数都是独立的代码块，用于完成特殊任务，因此经常会用到函数相互调用的情况。具体演示在下面的函数练习中会有。

1.6、函数练习

1.利用函数封装方式，翻转任意一个数组

```
function reverse(arr) {
    var newArr = [];
    for (var i = arr.length - 1; i >= 0; i--) {
        newArr[newArr.length] = arr[i];
    }
    return newArr;
}
var arr1 = reverse([1, 3, 4, 6, 9]);
console.log(arr1);
```

2.利用函数封装方式，对数组排序 – 冒泡排序

```
function sort(arr) {
    for (var i = 0; i < arr.length - 1; i++) {
        for (var j = 0; j < arr.length - i - 1; j++)
{
            if (arr[j] > arr[j+1]) {
                var temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}
```

3.输入一个年份，判断是否是闰年（闰年：能被4整除并且不能被100整数，或者能被400整除）

```
function isRun(year) {
     var flag = false;
     if (year % 4 === 0 && year % 100 !== 0 || year
% 400 === 0) {
         flag = true;
     }
    return flag;
}
console.log(isRun(2010));
console.log(isRun(2012));
```

4.用户输入年份，输出当前年份2月份的天数，如果是闰年，则2月份是 29天， 如果是平年，则2月份是 28天

```
function backDay() {
    var year = prompt('请您输入年份:');
    if (isRun(year)) { //调用函数需要加小括号
        alert('你输入的' + year + '是闰年，2月份有29
天');
    } else {
        alert('您输入的' + year + '不是闰年，2月份有28
天');
    }
}
backDay();
//判断是否是闰年的函数
function isRun(year) {
    var flag = false;
    if (year % 4 === 0 && year % 100 !== 0 || year %
400 === 0) {
        flag = true;
```

```
    }
    return flag;
}
```

1.7、函数的两种声明方式

1.7.1、自定义函数方式(命名函数)

利用函数关键字 function 自定义函数方式。

```
// 声明定义方式
function fn() {...}

// 调用
fn();
```

1. **因为有名字，所以也被称为命名函数**
2. **调用函数的代码既可以放到声明函数的前面，也可以放在声明函数的后面**

1.7.2、函数表达式方式(匿名函数)

利用函数表达式方式的写法如下：

```
// 这是函数表达式写法，匿名函数后面跟分号结束
var fn = function(){...};

// 调用的方式，函数调用必须写到函数体下面
fn();
```

- 因为函数没有名字，所以也称为**匿名函数**
- 这个fn 里面存储的是一个函数
- **函数调用的代码必须写到函数体后面**

2、作用域

通常来说，一段程序代码中所用到的名字并不总是有效和可用的，而限定这个名字的**可用性的代码范围**就是这个名字的**作用域**。作用域的使用提高了程序逻辑的局部性，增强了程序的可靠性，减少了名字冲突。

JavaScript (ES6前) 中的作用域有两种：

- 全局作用域
- 局部作用域(函数作用域)

2.1、全局作用域

作用于所有代码执行的环境(整个 script 标签内部)或者一个独立的 js 文件

2.2、局部（函数）作用域

作用于函数内的代码环境，就是局部作用域。 因为跟函数有关系，所以也称为函数作用域

2.3、JS 没有块级作用域

- 块作用域由 `{}` 包括
- 在其他编程语言中（如 java、c#等），在 if 语句、循环语句中创建的变量，仅仅只能在本 if 语句、本循环语句中使用，如下面的Java代码：

```
if(true){
    int num = 123;
    System.out.println(num);    // 123
}
System.out.println(num);        // 报错
```

JS 中没有块级作用域(在ES6之前)

```
if(true){
    int num = 123;
    System.out.println(num);    // 123
}
System.out.println(num);        // 123
```

3、变量的作用域

在JavaScript中，根据作用域的不同，变量可以分为两种：

- 全局变量
- 局部变量

3.1、全局变量

在全局作用域下声明的变量叫做全局变量（**在函数外部定义的变量**）

- 全局变量在代码的任何位置都可以使用
- 在全局作用域下 var 声明的变量 是全局变量
- 特殊情况下，在函数内不使用 var 声明的变量也是全局变量（不建议使用）

3.2、局部变量

在局部作用域下声明的变量叫做局部变量（**在函数内部定义的变量**）

- 局部变量只能在该函数内部使用
- 在函数内部 var 声明的变量是局部变量
- 函数的**形参**实际上就是**局部变量**

3.3、区别

- 全局变量：在任何一个地方都可以使用，只有在浏览器关闭时才会被销毁，因此比较占内存
- 局部变量：只在函数内部使用，当其所在的代码块被执行时，会被初始化；当代码块运行结束后，就会被销毁，因此更节省内存空间

4、作用域链

1. 只要是代码，就至少有一个作用域
2. 写在函数内部的叫局部作用域
3. 如果函数中还有函数，那么在这个作用域中就又可以诞生一个作用域
4. 根据在内部函数可以访问外部函数变量的这种机制，用链式查找决定哪些数据能被内部函数访问，就称作作用域链

```
// 作用域链：内部函数访问外部函数的变量，采取的是链式查找的方式来决定取哪个值，这种结构我们称为作用域链表

var num = 10;
funtion fn() { //外部函数
    var num = 20;

    function fun() { //内部函数
        console.log(num);  // 20 ,一级一级访问
    }
}
```

- 作用域链：采取**就近原则**的方式来查找变量最终的值。

5、预解析

首先来看几段代码和结果：

```
console.log(num);  // 结果是多少?
//会报错 num is undefined
console.log(num);  // 结果是多少?
var num = 10;
```

```
// undefined

// 命名函数(自定义函数方式):若我们把函数调用放在函数声明上面
fn();                //11
function fn() {
    console.log('11');
}

// 匿名函数(函数表达式方式):若我们把函数调用放在函数声明上面
fn();
var  fn = function() {
    console.log('22'); // 报错
}


//相当于执行了以下代码
var fn;
fn();       //fn没赋值，没这个，报错
var  fn = function() {
    console.log('22'); //报错
}
```

JavaScript 代码是由浏览器中的 JavaScript 解析器来执行的。JavaScript 解析器在运行 JavaScript 代码的时候分为两步：**预解析和代码执行。**

- **预解析**：js引擎会把js里面所有的 **var** 还有 **function** 提升到当前作用域的最前面
- **代码执行**：从上到下执行JS语句

预解析只会发生在通过 var 定义的变量和 function 上。学习预解析能够让我们知道**为什么在变量声明之前访问变量的值是 undefined，为什么在函数声明之前就可以调用函数。**

5.1、变量预解析(变量提升)

变量预解析也叫做变量提升、函数提升

变量提升: 变量的声明会被提升到**当前作用域**的最上面，变量的赋值不会提升

```
console.log(num);  // 结果是多少?
var num = 10;
// undefined



//相当于执行了以下代码
var num;          // 变量声明提升到当前作用域最上面
console.log(num);
num = 10;         // 变量的赋值不会提升
```

5.2、函数预解析(函数提升)

函数提升： 函数的声明会被提升到**当前作用域**的最上面，但是不会调用函数。

```
fn();                //11
function fn() {
    console.log('11');
}
1234
```

5.3、解决函数表达式声明调用问题

对于函数表达式声明调用需要记住：**函数表达式调用必须写在函数声明的下面**

```
// 匿名函数(函数表达式方式):若我们把函数调用放在函数声明上面
fn();
var  fn = function() {
    console.log('22'); // 报错
}


//相当于执行了以下代码
var fn;
fn();       //fn没赋值，没这个，报错
var  fn = function() {
    console.log('22'); //报错
}
```

5.4、预解析练习

预解析部分十分重要，可以通过下面4个练习来理解。

Pink老师的视频讲解预解析：https://www.bilibili.com/video/BV1Sy4y1C7ha?p=143

```
// 练习1
var num = 10;
fun();
function fun() {
    console.log(num);   //undefined
    var num = 20;
}
// 最终结果是 undefined
```

上述代码相当于执行了以下操作

```
var num;
function fun() {
    var num;
    console.log(num);
    num = 20;
}
num = 10;
fun();
```

```
// 练习2
var num = 10;
function fn(){
    console.log(num);        //undefined
    var num = 20;
    console.log(num);        //20
}
fn();
// 最终结果是 undefined 20
```

上述代码相当于执行了以下操作

```
var num;
function fn(){
    var num;
    console.log(num);
    num = 20;
    console.log(num);
}
num = 10;
fn();
```

```
// 练习3
var a = 18;
f1();

function f1() {
    var b = 9;
    console.log(a);
    console.log(b);
    var a = '123';
}
```

上述代码相当于执行了以下操作

```
var a;
function f1() {
    var b;
    var a
    b = 9;
    console.log(a); //undefined
    console.log(b); //9
    a = '123';
}
a = 18;
f1();
11
```

```
// 练习4
f1();
```

```
console.log(c);
console.log(b);
console.log(a);
function f1() {
    var a = b = c = 9;
    // 相当于 var a = 9; b = 9;c = 9;  b和c的前面没有var声明,当全局变量看
    // 集体声明 var a = 9,b = 9,c = 9;
    console.log(a);
    console.log(b);
    console.log(c);
}
```

上述代码相当于执行了以下操作

```
function f1() {
    var a;
    a = b = c = 9;
    console.log(a); //9
    console.log(b); //9
    console.log(c); //9
}
f1();
console.log(c); //9
console.log(b); //9
console.log(a); //报错 a是局部变量
```

✍目录总览

1、对象 🔥

在 JavaScript 中，对象是一组无序的相关属性和方法的集合，所有的事物都是对象，例如字符串、数值、数组、函数等。

对象是由属性和方法组成的：

- 属性：事物的**特征，*在对象中用*属性**来表示（常用名词）
- 方法：事物的**行为，*在对象中用*方法**来表示（常用动词）

1.1、创建对象 🔥

在 JavaScript 中，现阶段我们可以采用三种方式创建对象（object）：

- 利用字面量创建对象
- 利用 new Object创建对象
- 利用构造函数创建对象

①利用字面量创建对象

对象字面量：就是花括号 `{ }` 里面包含了表达这个具体事物（对象）的属性和方法

`{ }` 里面采取键值对的形式表示

- 键：相当于属性名
- 值：相当于属性值，可以是任意类型的值（数字类型、字符串类型、布尔类型，函数类型等）

```
var star = {
    name : 'pink',
    age : 18,
    sex : '男',
    sayHi : function(){
        alert('大家好啊~');
    }
};
// 多个属性或者方法中间用逗号隔开
// 方法冒号后面跟的是一个匿名函数
```

对象的调用

- 对象里面的属性调用 : 对象.属性名 ，这个小点 . 就理解为“ **的** ”
- 对象里面属性的另一种调用方式 : 对象[‘属性名’]，注意方括号里面的属性必须**加引号**，我们后面会用
- 对象里面的方法调用：对象.方法名() ，注意这个方法名字后面**一定加括号**

```
console.log(star.name)      // 调用名字属性
console.log(star['name'])   // 调用名字属性
star.sayHi();               // 调用 sayHi 方法,注意,一定不要忘记带后面的括号
123
```

变量、属性、函数、方法总结

- 变量：单独声明赋值，单独存在
- 属性：对象里面的变量称为属性，不需要声明，用来描述该对象的特征
- 函数：单独存在的，通过==“函数名()”==的方式就可以调用
- 方法：对象里面的函数称为方法，方法不需要声明，使用==“对象.方法名()”==的方式就可以调用，方法用来描述该对象的行为和功能。

②利用 new Object 创建对象

跟之前的 `new Array()` 原理一致：`var 对象名 = new Object();`

使用的格式：对象.属性 = 值

```
var obj = new Object(); //创建了一个空的对象
obj.name = '张三丰';
obj.age = 18;
obj.sex = '男';
obj.sayHi = function() {
    console.log('hi~');
}

//1.我们是利用等号赋值的方法添加对象
//2.每个属性和方法之间用分号结束
console.log(obj.name);
console.log(obj['sex']);
obj.sayHi();
```

③利用构造函数创建对象

构造函数：是一种特殊的函数，主要用来初始化对象，即为对象成员变量赋初始值，它总与 new 运算符一起使用。我们可以把对象中一些公共的属性和方法抽取出来，然后封装到这个函数里面。

在 js 中，使用构造函数要时要注意以下两点：

- 构造函数用于创建某一类对象，其首字母要大写
- 构造函数要和 new 一起使用才有意义

```
//构造函数的语法格式
function 构造函数名() {
    this.属性 = 值;
    this.方法 = function() {}
}
new 构造函数名();
//1. 构造函数名字首字母要大写
//2. 构造函数不需要return就可以返回结果
//3. 调用构造函数必须使用 new
//4. 我们只要new Star() 调用函数就创建了一个对象
//5. 我们的属性和方法前面必须加this
function Star(uname,age,sex) {
    this.name = uname;
    this.age = age;
    this.sex = sex;
    this.sing = function(sang){
```

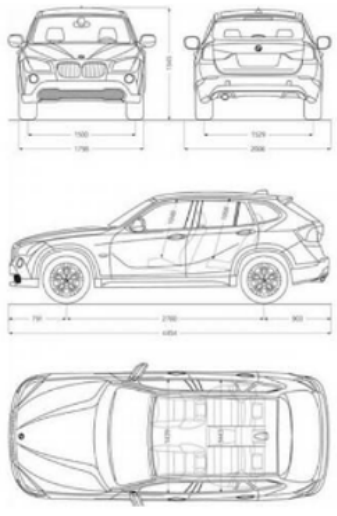
```
        console.log(sang);
    }
}
var ldh = new Star('刘德华',18,'男');
console.log(typeof ldh) // object对象，调用函数返回的是对象

console.log(ldh.name);
console.log(ldh['sex']);
ldh.sing('冰雨');
//把冰雨传给了sang

var zxy = new Star('张学友',19,'男');
```

- 构造函数名字首字母要大写
- 函数内的属性和方法前面需要添加 this ，表示当前对象的属性和方法。
- 构造函数中不需要 return 返回结果。
- 当我们创建对象的时候，必须用 new 来调用构造函数。

构造函数和对象

- 构造函数，如 Stars()，抽象了对象的公共部分，封装到了函数里面，它泛指某一大类（class）
- 创建对象，如 new Stars()，特指某一个，通过 new 关键字创建对象的过程我们也称为对象实例化

汽车设计图纸（构造函数）



一辆真宝马！(对象实例)

new关键字

new 在执行时会做四件事:

1. 在内存中创建一个新的空对象。
2. 让 this 指向这个新的对象。

3. 执行构造函数里面的代码，给这个新对象添加属性和方法
4. 返回这个新对象（所以构造函数里面不需要return）

1.2、遍历对象的属性

- `for...in` 语句用于对数组或者对象的属性进行循环操作

语法如下

```
for(变量 in 对象名字){
    // 在此执行代码
}
123
```

语法中的变量是自定义的，它需要符合命名规范，通常我们会将这个变量写为 k 或者 key。

```
for(var k in obj) {
    console.log(k);      //这里的 k 是属性名
    console.log(obj[k]);//这里的 obj[k] 是属性值
}

var obj = {
    name: '秦sir',
    age: 18,
    sex: '男',
    fn:function() {};
};
console.log(obj.name);
console.log(obj.age);
console.log(obj.sex);

//for in 遍历我们的对象
//for (变量 in 对象){}
//我们使用for in 里面的变量 我们喜欢写k 或者key
for(var k in obj){
    console.log(k); // k 变量 输出得到的是属性名
    console.log(obj[k]); // obj[k] 得到的是属性值
}
```

2、内置对象

- JavaScript 中的对象分为3种：自定义对象 、内置对象、 浏览器对象
- 内置对象就是指 JS 语言自带的一些对象，这些对象供开发者使用，并提供了一些常用的或是最基本而必要的功能
- JavaScript 提供了多个内置对象：Math、 Date 、Array、String等

2.1、查文档

学习一个内置对象的使用，只要学会其常用成员的使用即可，我们可以通过查文档学习，可以通过MDN/W3C来查询

MDN: https://developer.mozilla.org/zh-CN/

2.1.1、如何学习对象中的方法

1. 查阅该方法的功能
2. 查看里面参数的意义和类型
3. 查看返回值的意义和类型
4. 通过 demo 进行测试

3、Math对象

Math 对象不是构造函数，它具有数学常数和函数的属性和方法。跟数学相关的运算（求绝对值，取整、最大值等）可以使用 Math 中的成员。

```
// Math数学对象，不是一个构造函数，所以我们不需要new 来调用，而是直接使用里面的属性和方法即可

Math.PI                 // 圆周率
Math.floor()            // 向下取整
Math.ceil()             // 向上取整
Math.round()            // 四舍五入版 就近取整    注意 -3.5    结果是  -3
Math.abs()              // 绝对值
Math.max()/Math.min()   // 求最大和最小值
```

注意：上面的方法必须带括号

```
console.log(Math.PI);
console.log(Math.max(1,99,3)); // 99
12
```

练习：封装自己的数学对象

利用对象封装自己的数学对象，里面有PI 最大值 和最小值

```
var myMath = {
    PI: 3.141592653,
    max: function() {
        var max = arguments[0];
        for (var i = 1; i < arguments.length; i++) {
            if (arguments[i] > max) {
```

```
                max = arguments[i];
            }
        }
        return max;
    },
    min: function() {
        var min = arguments[0];
        for (var i = 1; i < arguments.length; i++) {
            if (arguments[i] < min) {
                min = arguments[i];
            }
        }
        return min;
    }
}
console.log(myMath.PI);
console.log(myMath.max(1, 5, 9));
console.log(myMath.min(1, 5, 9));
192021222324
```

3.Math绝对值和三个取整方法

- `Math.abs()` 取绝对值
- 三个取整方法：
  - `Math.floor()` ：向下取整
  - `Math.ceil()` ：向上取整
  - `Matg.round()` ：四舍五入，其他数字都是四舍五入，但是5特殊，它往大了取

```
//1.绝对值方法
console.log(Math.abs(1));  // 1
console.log(Math.abs(-1)); // 1
console.log(Math.abs('-1')); // 1 隐式转换，会把字符串-1 转换为数字型
//2.三个取整方法
console.log(Math.floor(1.1)); // 1 向下取整，向最小的取值 floor-地板
console.log(Math.floor(1.9)); //1

console.log(Math.ceil(1.1)); //2 向上取整，向最大的取值 ceil-天花板
console.log(Math.ceil(1.9)); //2

//四舍五入 其他数字都是四舍五入，但是5特殊，它往大了取

console.log(Math.round(1.1)); //1 四舍五入
```

```
console.log(Math.round(1.5)); //2
console.log(Math.round(1.9)); //2
console.log(Math.round(-1.1)); // -1
console.log(Math.round(-1.5)); // -1
```

4.随机数方法random()

- random() 方法可以随机返回一个小数，其取值范围是 [0，1)，左闭右开 0 <= x < 1
- 得到一个两数之间的随机整数，包括第一个数，不包括第二个数

```
// 得到两个数之间的随机整数，并且包含这两个整数
function getRandom(min,max) {
    return Math.floor(Math.random() * (max - min +
1)) + min;
}
console.log(getRandom(1,10));
```

1.随机点名

```
var arr = ['张三', '李四','王五','秦六'];
console.log(arr[getRandom(0,arr.length - 1)]);
12
```

2.猜数字游戏

案例：猜数字游戏

程序随机生成一个 1~ 10 之间的数字，并让用户输入一个数字，

1. 如果大于该数字，就提示，数字大了，继续猜；
2. 如果小于该数字，就提示数字小了，继续猜；
3. 如果等于该数字，就提示猜对了， 结束程序。

```
function getRandom(min,max) {
    return Math.floor(Math.random() * (max - min +
1)) + min;
}
var random = getRandom(1,10);
while(true) { //死循环 ，需要退出循环条件
     var num = prompt('请输入1~10之间的一个整数:');
     if(num > random) {
         alert('你猜大了');
     }else if (num < random) {
         alert('你猜小了');
```

```
    }else {
        alert('你猜中了');
        break; //退出整个循环
    }
}
1415
```

4、Data()日期对象

- Date 对象和 Math 对象不一样，他是一个构造函数，所以我们需要实例化后才能使用
- Date 实例用来处理日期和时间

4.1、Date()方法的使用

4.1.1、获取当前时间必须实例化

```
var now = new Date();
console.log(now);
12
```

4.1.2、Date()构造函数的参数

如果括号里面有时间，就返回参数里面的时间。例如日期格式字符串为 `‘2019-5-1’`，可以写成 `new Date('2019-5-1')` 或者 `new Date('2019/5/1')`

- 如果Date()不写参数，就返回当前时间
- 如果Date()里面写参数，就返回括号里面输入的时间

```
// 1.如果没有参数，返回当前系统的当前时间
var now = new Date();
console.log(now);


// 2.参数常用的写法 数字型 2019,10,1  字符串型 '2019-10-1 8:8:8' 时分秒
// 如果Date()里面写参数，就返回括号里面输入的时间
var data = new Date(2019,10,1);
console.log(data);  // 返回的是11月不是10月

var data2 = new Date('2019-10-1 8:8:8');
console.log(data2);
1112
```

4.2、日期格式化

我们想要 2019-8-8 8:8:8 格式的日期，要怎么办?

需要获取日期指定的部分，所以我们要手动的得到这种格式

方法名	说明	代码
getFullYear()	获取当年	dObj.getFullYear()
getMonth()	获取当月(0-11)	dObj.getMonth()
getDate()	获取当天日期	dObj.getDate()
getDay()	获取星期几(周日0到周六6)	dObj.getDay()
getHours()	获取当前小时	dObj.getHours()
getMinutes()	获取当前小时	dObj.getMinutes()
getSeconds()	获取当前秒钟	dObj.gerSeconds()

```
var date = new Date();
console.log(date.getFullYear()); // 返回当前日期的年 2019
console.log(date.getMonth() + 1);  //返回的月份小一个月 记得月份 +1
console.log(date.getDate); //返回的是几号
console.log(date.getDay());  //周一返回1 周6返回六 周日返回0



// 写一个 2019年 5月 1日 星期三
var date = new Date();
var year =  date.getFullYear();
var month = date.getMonth() + 1;
var dates = date.getDate();
console.log('今天是' + year +'年' + month + '月' + dates +'日' );

// 封装一个函数返回当前的时分秒 格式 08:08:08
function getTimer() {
    var time = new Date();
    var h = time.getHours();
    h = h < 10 ? '0' + h : h;
    var m = time.getMinutes();
    m = m < 10 ? '0' + m : m;
```

```
    var s = time.getSeconds();
    s = s < 10 ? '0' + s : s;
    return h + ':' + m + ':' + s;
}
console.log(getTimer());
1920212223242526 27
```

4.3、获取日期的总的毫秒形式

- `date.valueOf()` ：得到现在时间距离1970.1.1总的毫秒数
- `date.getTime()` ：得到现在时间距离1970.1.1总的毫秒数

```
// 获取Date总的毫秒数 不是当前时间的毫秒数 而是距离1970年1月1号过了多少毫秒数

// 实例化Date对象
var date = new Date();

// 1 .通过 valueOf()  getTime() 用于获取对象的原始值
console.log(date.valueOf());  //得到现在时间距离1970.1.1总的毫秒数
console.log(date.getTime());

// 2.简单的写法
var date1 = +new Date();  // +new Date()返回的就是总的毫秒数,
console.log(date1);

// 3. HTML5中提供的方法 获得总的毫秒数 有兼容性问题
console.log(Date.now());
1415
```

倒计时效果

做一个倒计时效果


```
function countDown(time) {
    var nowTime = +new Date(); //没有参数，返回的是当前时间总的毫秒数
    var inputTime = +new Date(time); // 有参数，返回的是用户输入时间的总毫秒数
    var times = (inputTime - nowTime) / 1000; //times就是剩余时间的总的秒数
    var d = parseInt(times / 60 / 60 / 24); //天数
    d < 10 ? '0' + d : d;
    var h = parseInt(times / 60 / 60 % 24); //小时
    h < 10 ? '0' + h : h;
    var m = parseInt(times / 60 % 60); //分
    m < 10 ? '0' + m : m;
    var s = parseInt(times % 60); //秒
    s < 10 ? '0' + s : s;
    return d + '天' + h + '时' + m + '分' + s + '秒';
}
console.log(countDown('2020-11-09 18:29:00'));
var date = new Date;
console.log(date); //现在时间
```

5、数组对象

5.1、数组对象的创建

创建数组对象的两种方式

- 字面量方式
- new Array()

5.2、检测是否为数组

- instanceof 运算符，可以判断一个对象是否属于某种类型
- `Array.isArray()` 用于判断一个对象是否为数组，isArray() 是 HTML5 中提供的方法

```
var arr = [1, 23];
var obj = {};
console.log(arr instanceof Array); // true
console.log(obj instanceof Array); // false
console.log(Array.isArray(arr));   // true
console.log(Array.isArray(obj));   // false
6
```

5.3、添加删除数组元素

方法名	说明	返回值
push(参数1...)	末尾添加一个或多个元素，注意修改原数组	并返回新的长度
pop()	删除数组最后一个元素	返回它删除的元素的值
unshift(参数1...)	向数组的开头添加一个或更多元素，注意修改原数组	并返回新的长度
shift()	删除数组的第一个元素，数组长度减1，无参数，修改原数组	并返回第一个元素

```
// 1.push() 在我们数组的末尾，添加一个或者多个数组元素
push 推
var arr = [1, 2, 3];
arr.push(4, '秦晓');
console.log(arr);
console.log(arr.push(4, '秦晓'));
console.log(arr);
// push 完毕之后，返回结果是新数组的长度


// 2. unshift 在我们数组的开头 添加一个或者多个数组元素
arr.unshift('red');
console.log(arr);

// pop() 它可以删除数组的最后一个元素，一次只能删除一个元素
arr.pop(); //不加参数
```

```
// shift() 它剋删除数组的第一个元素,一次只能删除一个元素
arr.shift(); //不加参数
```

👉 **筛选数组**

有一个包含工资的数组[1500,1200,2000,2100,1800],要求把数组中工资超过2000的删除，剩余的放到新数组里面

```
var arr = [1500, 1200, 2000, 2100, 1800];
var newArr = [];
for (var i = 0; i < arr.length; i++) {
    if (arr[i] < 2000) {
        newArr.push(arr[i]);
    }
}
console.log(newArr);
```

5.4、数组排序 🔥

方法名	说明	是否修改原数组
reverse()	颠倒数组中元素的顺序，无参数	该方法会改变原来的数组，返回新数组
sort()	对数组的元素进行排序	该方法会改变原来的数组，返回新数组

```
// 1.翻转数组
var arr = ['pink','red','blue'];
arr.reverse();
console.log(arr);

// 2.数组排序(冒泡排序)
var arr1 = [3,4,7,1];
arr1.sort();
console.log(arr1);

// 对于双位数
var arr = [1,64,9,61];
arr.sort(function(a,b) {
     return b - a;  //降序的排列
     return a - b; //升序
    }
 )
```

5.5、数组索引

方法名	说明	返回值
indexOf()	数组中查找给定元素的第一个索引	如果存在返回索引号，如果不存在，则返回-1
lastIndexOf()	在数组的最后一个索引，从后向前索引	如果存在返回索引号，如果不存在，则返回-1

```
//返回数组元素索引号方法 indexOf(数组元素)  作用就是返回该数组元素的索引号
//它只发返回第一个满足条件的索引号
//如果找不到元素，则返回-1
var arr = ['red','green','blue','pink','blue'];
console.log(arr.indexOf('blue'));  // 2

console.log(arr.lastIndexOf('blue')); // 4
```

5.5.1、数组去重

案例： 数组去重（重点案例）

有一个数组['c', 'a', 'z', 'a', 'x', 'a', 'x', 'c', 'b']，要求去除数组中重复的元素。

分析：把旧数组里面不重复的元素选取出来放到新数组中，重复的元素只保留一个，放到新数组中去重。

核心算法：我们遍历旧数组，然后拿着旧数组元素去查询新数组，如果该元素在新数组里面没有出现过，我们就添加，否则不添加。

我们怎么知道该元素没有存在？ 利用 新数组.indexOf(数组元素) 如果返回是 -1 就说明 新数组里面没有改元素

```
// 封装一个去重的函数 unique 独一无二的
function unique(arr) {
    var newArr = [];
    for (var i = 0; i < arr.length; i++) {
        if (newArr.indexOf(arr[i]) === -1) {
            newArr.push(arr[i]);
        }
    }
    return newArr;
}
var demo = unique(['c', 'a', 'z', 'a', 'x', 'a',
'x', 'c', 'b']);
console.log(demo);
```

5.6、数组转化为字符串

方法名	说明	返回值
toString()	把数组转换成字符串，逗号分隔每一项	返回一个字符串
join('分隔符')	方法用于把数组中的所有元素转换为一个字符串	返回一个字符串

```
// 1.toString() 将我们的数组转换为字符串
var arr = [1, 2, 3];
console.log(arr.toString()); // 1,2,3
// 2.join('分隔符')
var arr1 = ['green', 'blue', 'red'];
console.log(arr1.join()); // 不写默认用逗号分割
console.log(arr1.join('-')); //  green-blue-red
console.log(arr1.join('&')); // green&blue&red
```

5.7、其他方法

方法名	说明	返回值
concat()	连接两个或多个数组 不影响原数组	返回一个新的数组
slice()	数组截取slice(begin,end)	返回被截取项目的新数组
splice()	数组删除splice(第几个开始要删除的个数)	返回被删除项目的新数组，这个会影响原数组

6、字符串对象

6.1、基本包装类型

为了方便操作基本数据类型，JavaScript 还提供了三个特殊的引用类型：String、Number和Boolean。

基本包装类型就是把简单数据类型包装成为复杂数据类型，这样基本数据类型就有了属性和方法。

我们看看下面代码有什么问题吗?

```
var str = 'andy';
console.log(str.length);
```

按道理基本数据类型是没有属性和方法的，而对象才有属性和方法，但上面代码却可以执行，这是因为 js 会把基本数据类型包装为复杂数据类型，其执行过程如下：

```
// 1.生成临时变量,把简单类型包装为复杂数据类型
var temp = new String('andy');
// 2.赋值给我们声明的字符变量
str = temp;
// 3.销毁临时变量
temp = null;
```

6.2、字符串的不可变

指的是里面的值不可变，虽然看上去可以改变内容，但其实是地址变了，内存中新开辟了一个内存空间。

```
var str = 'abc';
str = 'hello';
// 当重新给 str 赋值的时候，常量'abc'不会被修改，依然在内存中
// 重新给字符串赋值，会重新在内存中开辟空间，这个特点就是字符串的不可变
// 由于字符串的不可变，在大量拼接字符串的时候会有效率问题
var str = '';
for(var i = 0; i < 10000;  i++){
    str += i;
}
console.log(str);
// 这个结果需要花费大量时间来显示，因为需要不断的开辟新的空间
```

6.3、根据字符返回位置

字符串所有的方法，都不会修改字符串本身(字符串是不可变的)，**操作完成会返回一个新的字符串**

方法名	说明
indexOf(‘要查找的字符’，开始的位置)	返回指定内容在元字符串中的位置，如果找不到就返回-1，开始的位置是index索引号
lastIndexOf()	从后往前找，只找第一个匹配的

```
 // 字符串对象  根据字符返回位置  str.indexOf('要查找的字符', [起始的位置])
        var str = '改革春风吹满地，春天来了';
        console.log(str.indexOf('春')); //默认从0开始查找 ，结果为2
        console.log(str.indexOf('春', 3)); // 从索引号是 3的位置开始往后查找，结果是8
```

6.3.1、返回字符位置

查找字符串 “abcoefoxyozzopp” 中所有o出现的位置以及次数

- 核心算法：先查找第一个o出现的位置
- 然后 只要 indexOf返回的结果不是 -1 就继续往后查找
- 因为 indexOf 只能查找到第一个，所以后面的查找，一定是当前索引加1，从而继续查找

```
var str = "oabcoefoxyozzopp";
var index = str.indexOf('o');
var num = 0;
// console.log(index);
while (index !== -1) {
    console.log(index);
    num++;
    index = str.indexOf('o', index + 1);
}
console.log('o出现的次数是: ' + num);
```

6.4、根据位置返回字符

方法名	说明	使用
charAt(index)	返回指定位置的字符(index字符串的索引号)	str.charAt(0)
charCodeAt(index)	获取指定位置处字符的ASCII码(index索引号)	str.charCodeAt(0)
str[index]	获取指定位置处字符	HTML,IE8+支持和charAt()等效

返回字符位置

判断一个字符串 “abcoefoxyozzopp” 中出现次数最多的字符，并统计其次数

- 核心算法：利用 charAt() 遍历这个字符串
- 把每个字符都存储给对象， 如果对象没有该属性，就为1，如果存在了就 +1
- 遍历对象，得到最大值和该字符

```
<script>
    // 有一个对象 来判断是否有该属性 对象['属性名']
    var o = {
        age: 18
    }
    if (o['sex']) {
        console.log('里面有该属性');

    } else {
        console.log('没有该属性');

    }

    //  判断一个字符串 'abcoefoxyozzopp' 中出现次数最多的字符，并统计其次数。
    // o.a = 1
    // o.b = 1
    // o.c = 1
    // o.o = 4
    // 核心算法：利用 charAt() 遍历这个字符串
    // 把每个字符都存储给对象， 如果对象没有该属性，就为1，如果存在了就 +1
    // 遍历对象，得到最大值和该字符
    var str = 'abcoefoxyozzopp';
    var o = {};
    for (var i = 0; i < str.length; i++) {
```

```
        var chars = str.charAt(i); // chars 是 字符串的每一个字符
        if (o[chars]) { // o[chars] 得到的是属性值
            o[chars]++;
        } else {
            o[chars] = 1;
        }
    }
    console.log(o);
    // 2. 遍历对象
    var max = 0;
    var ch = '';
    for (var k in o) {
        // k 得到是 属性名
        // o[k] 得到的是属性值
        if (o[k] > max) {
            max = o[k];
            ch = k;
        }
    }
    console.log(max);
    console.log('最多的字符是' + ch);
</script>
```

6.5、字符串操作方法 🔥

方法名	说明
concat(str1,str2,str3...) 🔥	concat() 方法用于连接两个或对各字符串。拼接字符串 🔥
substr(start,length) 🔥	从 start 位置开始(索引号), length 取的个数。 🔥
slice(start,end)	从 start 位置开始，截取到 end 位置 ，end 取不到 (两个都是索引号)
substring(start,end)	从 start 位置开始，截取到 end 位置 ，end 取不到 (基本和 slice 相同，但是不接受负)

```
<script>
    // 1. concat('字符串1','字符串2'....)
    var str = 'andy';
    console.log(str.concat('red'));

    // 2. substr('截取的起始位置', '截取几个字符');
    var str1 = '改革春风吹满地';
    console.log(str1.substr(2, 2)); // 第一个2 是索引号的2   第二个2 是取几个字符
</script>
```

6.6、replace()方法

replace() 方法用于在字符串中用一些字符替换另一些字符

其使用格式： `replace(被替换的字符,要替换为的字符串)`

```
<script>
    // 1. 替换字符 replace('被替换的字符', '替换为的字符')  它只会替换第一个字符
    var str = 'andyandy';
    console.log(str.replace('a', 'b'));
    // 有一个字符串 'abcoefoxyozzopp'  要求把里面所有的 o 替换为 *
    var str1 = 'abcoefoxyozzopp';
    while (str1.indexOf('o') !== -1) {
        str1 = str1.replace('o', '*');
    }
    console.log(str1);
</script>
11
```

6.7、split()方法

split() 方法用于切分字符串，它可以将字符串切分为数组。在切分完毕之后，返回的是一个新数组。

例如下面代码：

```
var str = 'a,b,c,d';
console.log(str.split(','));
// 返回的是一个数组 ['a', 'b', 'c', 'd']
123
<script>
// 2. 字符转换为数组 split('分隔符')    前面我们学过 join 把数组转换为字符串
    var str2 = 'red, pink, blue';
    console.log(str2.split(','));
    var str3 = 'red&pink&blue';
    console.log(str3.split('&'));
</script>
67
```

6.8、大小写转换

- `toUpperCase()` 转换大写
- `toLowerCase()` 转换小写

7、简单类型于复杂类型

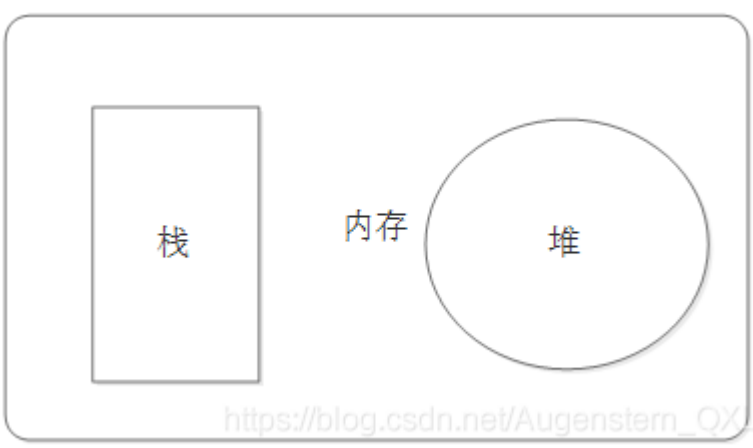
简单类型又叫做基本数据类型或者值类型，复杂类型又叫做引用类型。

- 值类型：简单数据类型/基本数据类型，在存储时变量中存储的是值本身，因此叫做值类型
  - string ，number，boolean，undefined，null
- 引用类型：复杂数据类型，在存储时变量中存储的仅仅是地址（引用），因此叫做引用数据类型
  - 通过 new 关键字创建的对象（系统对象、自定义对象），如 Object、Array、Date等

7.1、堆和栈

堆栈空间分配区别：

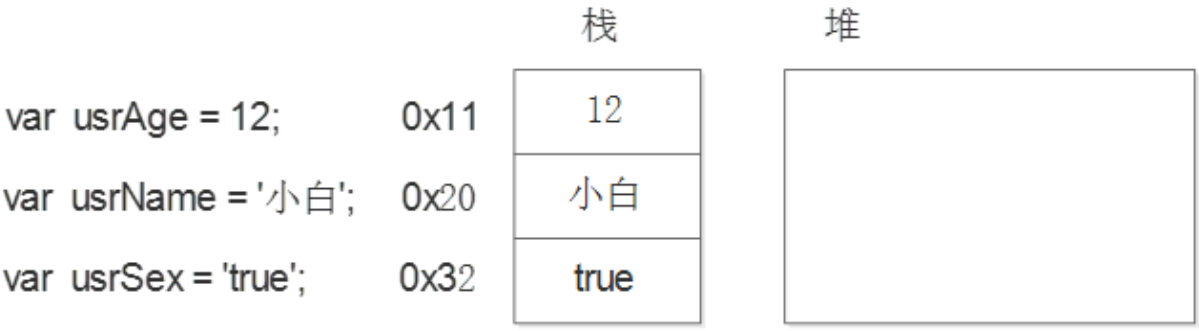
1. 栈（操作系统）：由操作系统自动分配释放存放函数的参数值、局部变量的值等。其操作方式类似于数据结构中的栈；
   - 简单数据类型存放到栈里面
2. 堆（操作系统）：存储复杂类型(对象)，一般由程序员分配释放，若程序员不释放，由垃圾回收机制回收。
   - 复杂数据类型存放到堆里面



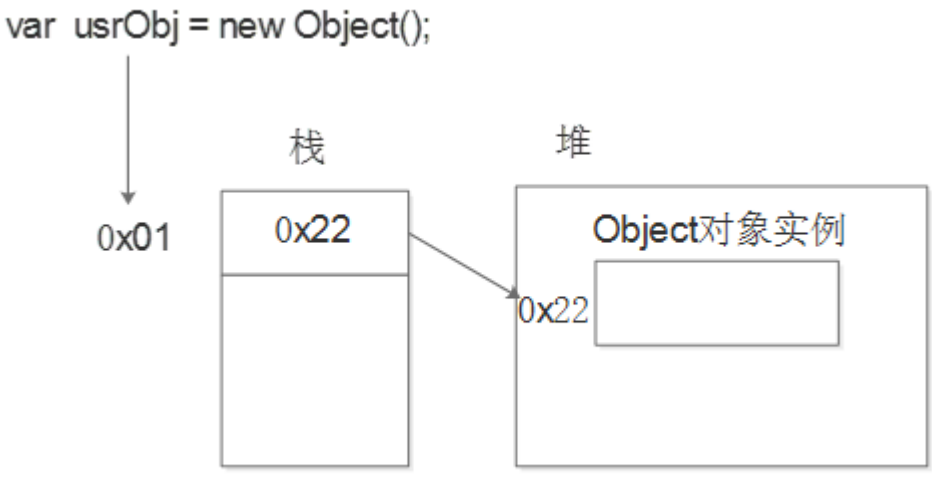
注意：JavaScript中没有堆栈的概念，通过堆栈的方式，可以让大家更容易理解代码的一些执行方式，便于将来学习其他语言。

7.2、简单类型的内存分配

- 值类型（简单数据类型）： string ，number，boolean，undefined，null
- 值类型变量的数据直接存放在变量（栈空间）中



- 引用类型（复杂数据类型）：通过 new 关键字创建的对象（系统对象、自定义对象），如 Object、Array、Date等
- 引用类型变量（栈空间）里存放的是地址，真正的对象实例存放在堆空间中



```
<script>
    // 简单数据类型 null  返回的是一个空的对象  object
    var timer = null;
    console.log(typeof timer);
    // 如果有个变量我们以后打算存储为对象，暂时没想好放啥，  这个时候就给 null
    // 1. 简单数据类型 是存放在栈里面 里面直接开辟一个空间存放的是值
    // 2. 复杂数据类型 首先在栈里面存放地址 十六进制表示 然后这个地址指向堆里面的数据
</script>
```

7.3、简单类型传参

函数的形参也可以看做是一个变量，当我们把一个值类型变量作为参数传给函数的形参时，其实是把变量在栈空间里的值复制了一份给形参，那么在方法内部对形参做任何修改，都不会影响到的外部变量。

```
<script>
    // 简单数据类型传参
    function fn(a) {
        a++;
        console.log(a);
    }
    var x = 10;
    fn(x);
    console.log(x);
</script>
```

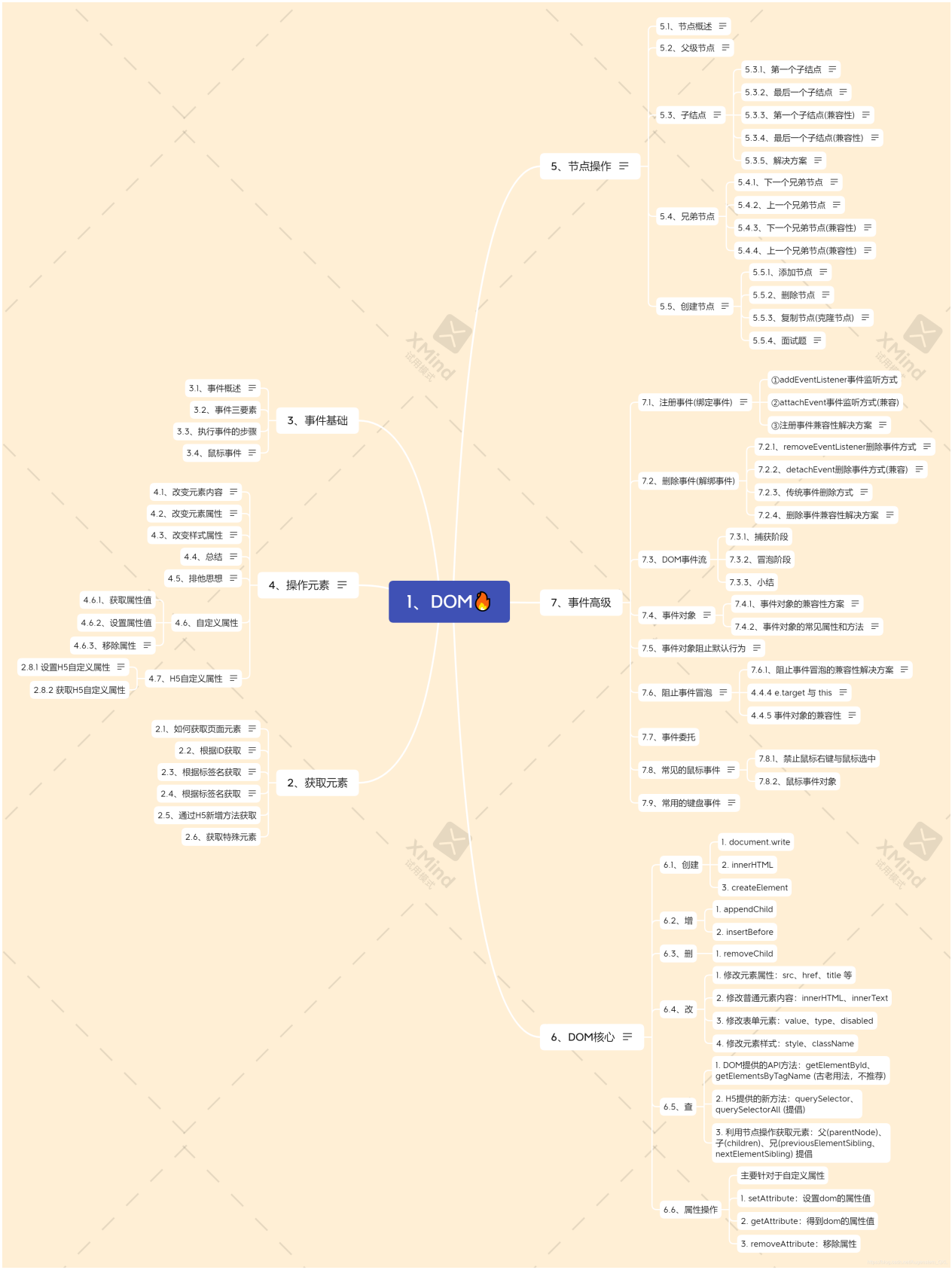
7.4、复杂类型传参

函数的形参也可以看做是一个变量，当我们把引用类型变量传给形参时，其实是把变量在栈空间里保存的堆地址复制给了形参，形参和实参其实保存的是同一个堆地址，所以操作的是同一个对象。

```
<script>
    // 复杂数据类型传参
    function Person(name) {
        this.name = name;
    }

    function f1(x) { // x = p
        console.log(x.name); // 2. 这个输出什么 ？  刘德华
```

```
        x.name = "张学友";
        console.log(x.name); // 3. 这个输出什么 ?
张学友
    }
    var p = new Person("刘德华");
    console.log(p.name); // 1. 这个输出什么 ?   刘德
华
    f1(p);
    console.log(p.name); // 4. 这个输出什么 ?   张学
友
</script>
```

✍目录总览

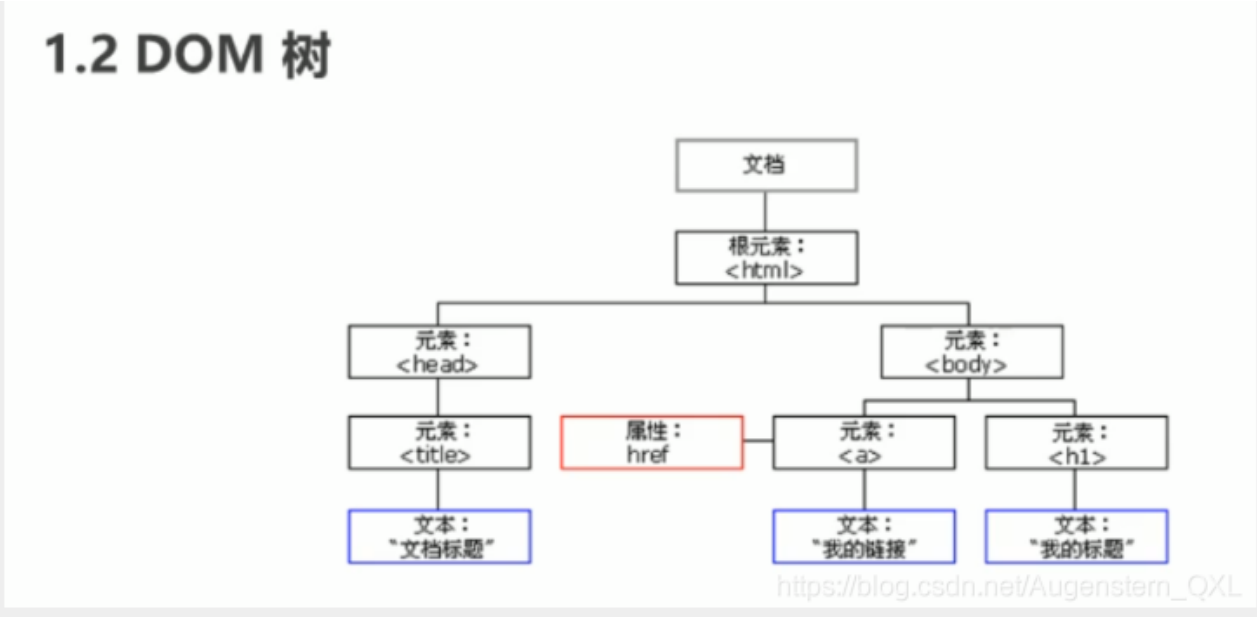


1、DOM简介

1.1、什么是DOM

文档对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展标记语言（HTML或者XML）的标准编程接口

W3C 已经定义了一系列的 DOM 接口，通过这些 DOM 接口可以改变网页的内容、结构和样式。



- 文档：一个页面就是一个文档，DOM中使用doucument来表示
- 元素：页面中的所有标签都是元素，DOM中使用 element 表示
- 节点：网页中的所有内容都是节点（标签，属性，文本，注释等），DOM中使用node表示

DOM 把以上内容都看做是对象

2、获取元素

2.1、如何获取页面元素

DOM在我们实际开发中主要用来操作元素。

我们如何来获取页面中的元素呢?

获取页面中的元素可以使用以下几种方式:

- 根据 ID 获取
- 根据标签名获取
- 通过 HTML5 新增的方法获取
- 特殊元素获取

2.2、根据ID获取

使用 `getElementByld()` 方法可以获取带ID的元素对象

```
doucument.getElementByld('id名')
```

使用 `console.dir()` 可以打印我们获取的元素对象，更好的查看对象里面的属性和方法。

示例

```
<div id="time">2019-9-9</div>
<script>
    // 1.因为我们文档页面从上往下加载，所以得先有标签，所以script写在标签下面
    // 2.get 获得 element 元素 by 通过 驼峰命名法
    // 3.参数 id是大小写敏感的字符串
    // 4.返回的是一个元素对象
    var timer = document.getElementById('time');
    console.log(timer);
    // 5. console.dir 打印我们的元素对象，更好的查看里面的属性和方法
    console.dir(timer);
</script>
```

2.3、根据标签名获取

根据**标签名**获取，使用 `getElementByTagName()` 方法可以返回带有指定标签名的**对象的集合**

```
doucument.getElementsByTagName('标签名');
```

- 因为得到的是一个对象的集合，所以我们想要操作里面的元素就需要遍历
- 得到元素对象是动态的
- 返回的是获取过来元素对象的集合，以伪数组的形式存储
- 如果获取不到元素，则返回为空的伪数组(因为获取不到对象)

```
<ul>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
    <li>知否知否，应是等你好久</li>
</ul>
<script>
    // 1.返回的是获取过来元素对象的集合 以伪数组的形式存储
    var lis = document.getElementsByTagName('li');
    console.log(lis);
    console.log(lis[0]);
    // 2.依次打印,遍历
    for (var i = 0; i < lis.length; i++) {
        console.log(lis[i]);
    }
    // 3.如果页面中只有 1 个 li，返回的还是伪数组的形式
    // 4.如果页面中没有这个元素，返回的是空伪数组
</script>
```

2.4、根据标签名获取

还可以根据标签名获取某个元素（父元素）内部所有指定标签名的子元素,获取的时候不包括父元素自己

```
element.getElementsByTagName('标签名')

ol.getElementsByTagName('li');
```

注意：父元素必须是单个对象(必须指明是哪一个元素对象)，获取的时候不包括父元素自己

```
<script>
    //element.getElementsByTagName('标签名'); 父元素必须是指定的单个元素
    var ol = document.getElementById('ol');
    console.log(ol.getElementsByTagName('li'));
</script>
```

2.5、通过H5新增方法获取

①getElementsByClassName

根据类名返回元素对象合集

- `document.getElementsByClassName('类名')`

```
document.getElementsByClassName('类名');
```

②document.querySelector

根据指定选择器返回第一个元素对象

```
document.querySelector('选择器');

// 切记里面的选择器需要加符号
// 类选择器.box
// id选择器 #nav
var firstBox = document.querySelector('.类名');
```

③document.querySelectorAll

根据指定选择器返回所有元素对象

```
document.querySelectorAll('选择器');
```

注意：

```
`querySelector` 和 `querySelectorAll` 里面的选择器需要加符号,比如: `document.querySelector('#nav');
```

④例子

```
<body>
    <div class="box">我是第一个div元素</div>
    <div class="box">我是第二个div元素</div>
    <div class="box1">我是第三个div元素</div>
    <ul>
        <li>1111</li>
        <li>2222</li>
        <li>3333</li>
        <li>4444</li>
    </ul>
</body>
<script>
    // 1. getElementsByClassName 根据类名获得某些元素集合
    var boxs = document.getElementsByClassName('box1')
    console.log("getElementsByClassName", boxs)
        //2.querySelector返回指定选择器的第一个元素对象  切记
        // 里面的选择器需要加符号  .box  #nav
    var firtBox1 = document.querySelector('.box') //获取第一个元素
    console.log("querySelector('.box')", firtBox1)

    var firtBox2 = document.querySelector('li')
    console.log(firtBox2)
        // 3. querySelectorAll()返回指定选择器的所有元素对象集合
    var all0 = document.querySelectorAll('.box')[0]
    console.log(all0.innerHTML)

</script>
```

2.6、获取特殊元素

①获取body元素

返回body元素对象

```
document.body;
```

②获取html元素

返回html元素对象

```
document.documentElement;
```

3、事件基础

3.1、事件概述

JavaScript 使我们有能力创建动态页面，而事件是可以被 JavaScript 侦测到的行为。

简单理解： 触发— 响应机制。

网页中的每个元素都可以产生某些可以触发 JavaScript 的事件，例如，我们可以在用户点击某按钮时产生一个事件，然后去执行某些操作。

3.2、事件三要素

1. 事件源(谁)
2. 事件类型(什么事件)
3. 事件处理程序(做啥)

```
<script>
    // 点击一个按钮，弹出对话框
    // 1. 事件是有三部分组成  事件源  事件类型  事件处理程序   我们也称为事件三要素
    //(1) 事件源 事件被触发的对象   谁  按钮
    var btn = document.getElementById('btn');
    //var btn = document.getElementsByClassName('btn')[0]
    console.log(btn)
    //(2) 事件类型  如何触发 什么事件 比如鼠标点击(onclick) 还是鼠标经过 还是键盘按下
    //(3) 事件处理程序  通过一个函数赋值的方式 完成
    btn.onclick = function() {
        alert('点秋香');
    }
</script>
```

3.3、执行事件的步骤

1. 获取事件源
2. 注册事件(绑定事件)
3. 添加事件处理程序(采取函数赋值形式)

```
<script>
    // 执行事件步骤
    // 点击div 控制台输出 我被选中了
    // 1. 获取事件源
    var div = document.querySelector('div');
    // 2.绑定事件 注册事件
    // div.onclick
    // 3.添加事件处理程序
    div.onclick = function() {
        console.log('我被选中了');
    }
</script>
```

3.4、鼠标事件

鼠标事件	触发条件
onclick	鼠标点击左键触发
onmouseover	鼠标经过触发
onmouseout	鼠标离开触发
onfocus	获得鼠标焦点触发
onblur	失去鼠标焦点触发
onmousemove	鼠标移动触发
onmouseup	鼠标弹起触发
onmousedown	鼠标按下触发

4、操作元素

JavaScript 的 DOM 操作可以改变网页内容、结构和样式，我们可以利用 DOM 操作元素来改变元素里面的内容 、属性等。注意以下都是属性

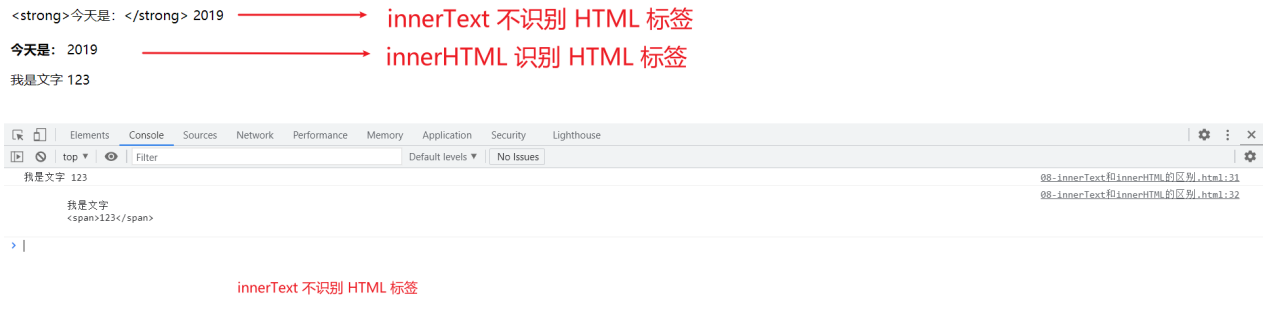
4.1、改变元素内容

从起始位置到终止位置的内容，但它去除html标签，同时空格和换行也会去掉。

```
element.innerText
```

起始位置到终止位置的全部内容，包括HTML标签，同时保留空格和换行

```
element.innerHTML
<body>
    <div></div>
    <p>
        我是文字
        <span>123</span>
    </p>

    <script>
        // innerText 和 innerHTML的区别
        // 1. innerText 不识别html标签,去除空格和换行
        var div = document.querySelector('div');
        div.innerText = '<strong>今天是：</strong>
2019';
        // 2. innerHTML 识别html标签 保留空格和换行的
        div.innerHTML = '<strong>今天是：</strong>
2019';
        // 这两个属性是可读写的  可以获取元素里面的内容
        var p = document.querySelector('p');
        console.log(p.innerText);
        console.log(p.innerHTML);
    </script>
</body>
```



4.2、改变元素属性

```
// img.属性
img.src = "xxx";

input.value = "xxx";
input.type = "xxx";
input.checked = "xxx";
input.selected = true / false;
input.disabled = true / false;
```

4.3、改变样式属性

我们可以通过 JS 修改元素的大小、颜色、位置等样式。

- 行内样式操作

```
// element.style
<style>
    div{
        width: 200px;
        height: 200px;
        background-color: rgb(21, 199, 199);
    }
</style>
<body>
    <div></div>
</body>
div.style.backgroundColor = 'pink';
div.style.width = '250px';
```

- 类名样式操作

```
// element.className
```

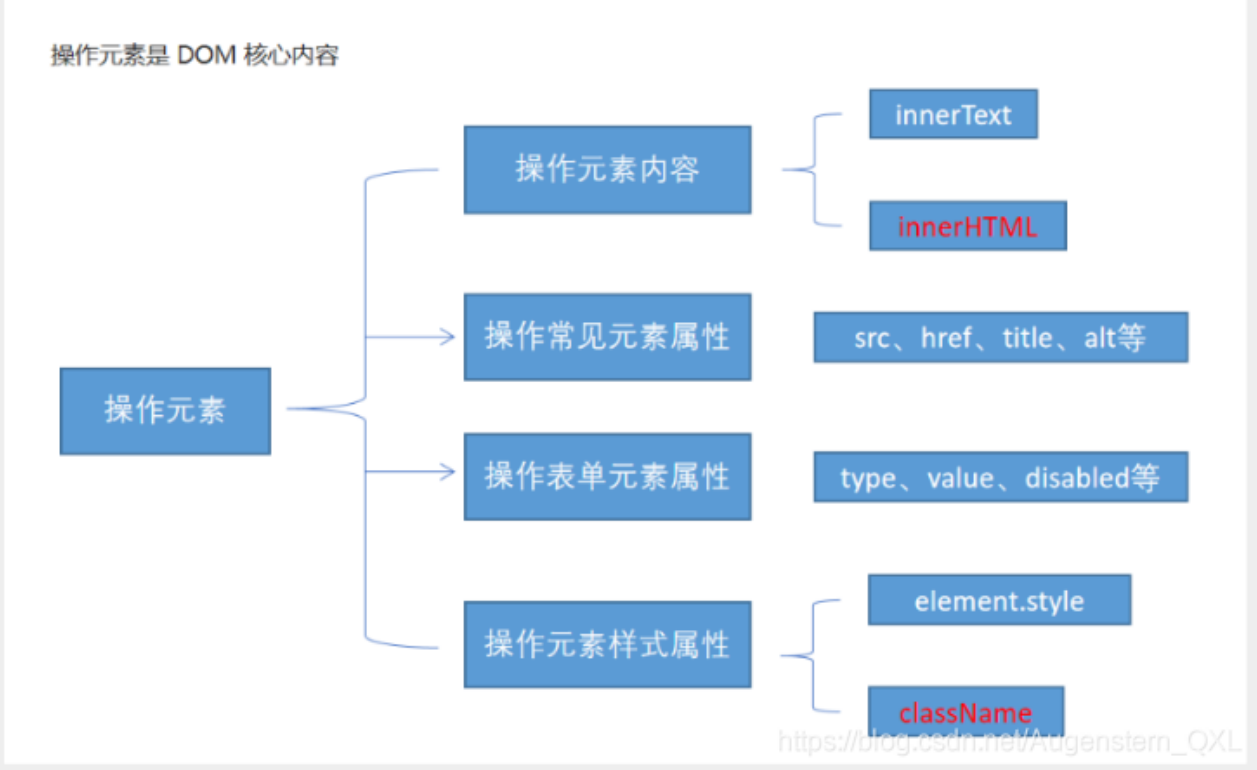
注意：

1. JS里面的样式采取驼峰命名法，比如 fontSize ，backgroundColor
2. JS 修改 style 样式操作 ，产生的是行内样式，CSS权重比较高
3. 如果样式修改较多，可以采取操作类名方式更改元素样式
4. class 因为是个保留字，因此使用 `className` 来操作元素类名属性
5. className 会直接更改元素的类名，会覆盖原先的类名

```
<body>
    <div class="first">文本</div>
    <script>
```

```
        // 1. 使用 element.style 获得修改元素样式  如果样式比较少 或者 功能简单的情况下使用
        var test = document.querySelector('div');
        test.onclick = function() {
            // this.style.backgroundColor = 'purple';
            // this.style.color = '#fff';
            // this.style.fontSize = '25px';
            // this.style.marginTop = '100px';
            // 让我们当前元素的类名改为了 change

            // 2. 我们可以通过 修改元素的className更改元素的样式 适合于样式较多或者功能复杂的情况
            // 3. 如果想要保留原先的类名，我们可以这么做 多类名选择器
            // this.className = 'change';
            this.className = 'first change';
        }
    </script>
</body>
19
```

4.4、总结

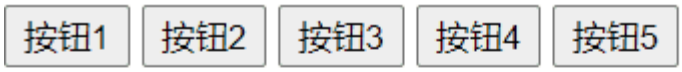


4.5、排他思想

如果有同一组元素，我们相要某一个元素实现某种样式，需要用到循环的排他思想算法：

1. 所有元素全部清除样式（干掉其他人）
2. 给当前元素设置样式 （留下我自己）
3. 注意顺序不能颠倒，首先干掉其他人，再设置自己

```
<body>
    <button>按钮1</button>
    <button>按钮2</button>
    <button>按钮3</button>
    <button>按钮4</button>
    <button>按钮5</button>
    <script>
        // 1. 获取所有按钮元素
        var btns =
document.getElementsByTagName('button');
        // btns得到的是伪数组  里面的每一个元素
btns[i]
        for (var i = 0; i < btns.length; i++) {
            btns[i].onclick = function() {
                // (1) 我们先把所有的按钮背景颜色去掉
干掉所有人
                for (var i = 0; i < btns.length;
i++) {
                    btns[i].style.backgroundColor =
'';
                }
                // (2) 然后才让当前的元素背景颜色为
pink 留下我自己
                this.style.backgroundColor = 'pink';

            }
        }
        //2. 首先先排除其他人，然后才设置自己的样式 这种
排除其他人的思想我们成为排他思想
    </script>
</body>
192021222324
```



4.6、自定义属性

4.6.1、获取属性值

- 获取内置属性值(元素本身自带的属性)

```
element.属性;
```

- 获取自定义的属性

```
element.getAttribute('属性');
```

4.6.2、设置属性值

- 设置内置属性值

```
element.属性 = '值';
```

- 主要设置自定义的属性

```
element.setAttribute('属性','值');
```

4.6.3、移除属性

```
//element.removeAttribute('属性');
<body>
    <div id="demo" index="1" class="nav"></div>
    <script>
        var div = document.querySelector('div');
        // 1. 获取元素的属性值
        // (1) element.属性
        console.log(div.id);
        //(2) element.getAttribute('属性')  get得到获取 attribute 属性的意思 我们程序员自己添加的属性我们称为自定义属性 index
        console.log(div.getAttribute('id'));
```

```
        console.log(div.getAttribute('index'));
        // 2. 设置元素属性值
        // (1) element.属性= '值'
        div.id = 'test';
        div.className = 'navs';
        // (2) element.setAttribute('属性', '值');
主要针对于自定义属性
        div.setAttribute('index', 2);
        div.setAttribute('class', 'footer'); //
class 特殊  这里面写的就是class 不是className
        // 3 移除属性 removeAttribute(属性)
        div.removeAttribute('index');
    </script>
</body>
```

4.7、H5自定义属性

自定义属性目的：

- 保存并保存数据，有些数据可以保存到页面中而不用保存到数据库中
- 有些自定义属性很容易引起歧义，不容易判断到底是内置属性还是自定义的，所以H5有了规定

4.7.1 设置H5自定义属性

H5规定自定义属性 `data-` 开头作为属性名并赋值

```
<div data-index = "1"></>
// 或者使用JavaScript设置
div.setAttribute('data-index',1);
```

4.7.2 获取H5自定义属性

- 兼容性获取 `element.getAttribute('data-index')`
- H5新增的： `element.dataset.index` 或 `element.dataset['index']` IE11才开始支持

```
<body>
    <div getTime="20" data-index="2" data-list-
name="andy"></div>
    <script>
        var div = document.querySelector('div');
        console.log(div.getAttribute('getTime'));
        div.setAttribute('data-time', 20);
        console.log(div.getAttribute('data-index'));
        console.log(div.getAttribute('data-list-
name'));
```

```
        // h5新增的获取自定义属性的方法 它只能获取data-开头的
        // dataset 是一个集合里面存放了所有以data开头的自定义属性
        console.log(div.dataset);
        console.log(div.dataset.index);
        console.log(div.dataset['index']);
        // 如果自定义属性里面有多个-链接的单词，我们获取的时候采取 驼峰命名法
        console.log(div.dataset.listName);
        console.log(div.dataset['listName']);
    </script>
</body>
```

5、节点操作

获取元素通常使用两种方式：

1.利用DOM提供的方法获取元素	2.利用节点层级关系获取元素
document.getElementById()	利用父子兄节点关系获取元素
document.getElementsByTagName()	逻辑性强，但是兼容性较差
document.querySelector 等	
逻辑性不强，繁琐	

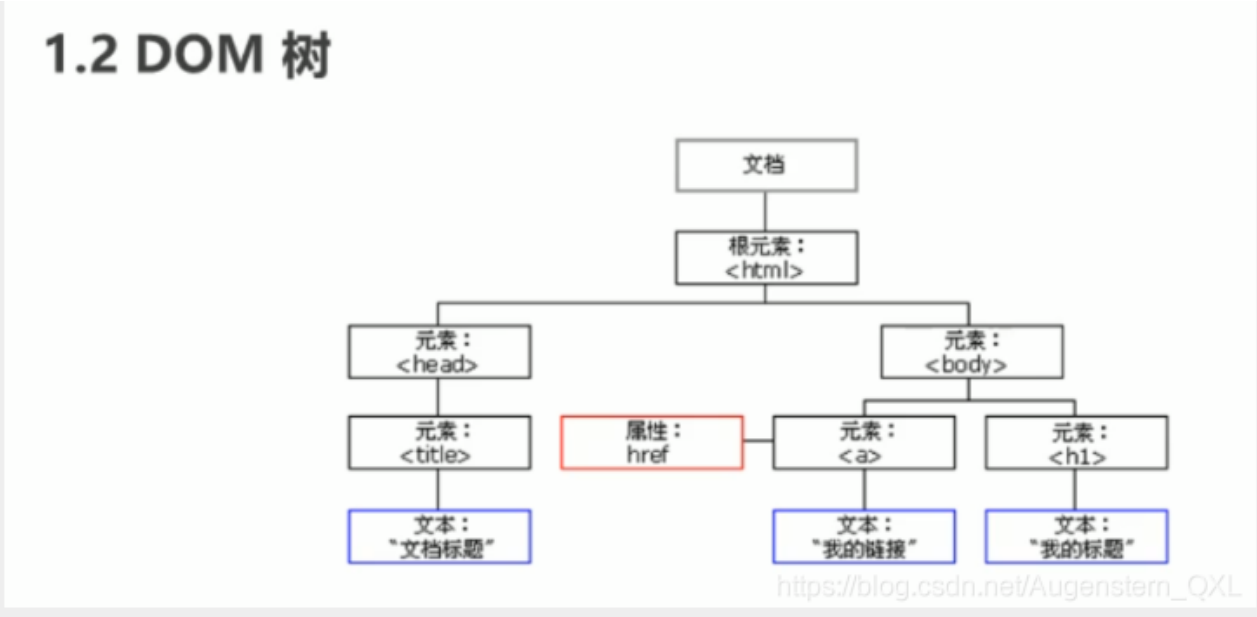
这两种方式都可以获取元素节点，我们后面都会使用，但是节点操作更简单

一般的，节点至少拥有三个基本属性

5.1、节点概述

网页中的所有内容都是节点（标签、属性、文本、注释等），在DOM 中，节点使用 node 来表示。

HTML DOM 树中的所有节点均可通过 JavaScript 进行访问，所有 HTML 元素（节点）均可被修改，也可以创建或删除。



一般的，节点至少拥有nodeType（节点类型）、nodeName（节点名称）和nodeValue（节点值）这三个基本属性。

- 元素节点：nodeType 为1
- 属性节点：nodeType 为2
- 文本节点：nodeType 为3(文本节点包括文字、空格、换行等)

我们在实际开发中，节点操作主要操作的是**元素节点**

利用 DOM 树可以把节点划分为不同的层级关系，常见的是**父子兄层级关系**。

5.2、父级节点

```
node.parentNode
```

- `parentNode`属性可以返回某节点的父结点，注意是最近的一个父结点
- 如果指定的节点没有父结点则返回null

```
<body>
    <!-- 节点的优点 -->
    <div>我是div</div>
    <span>我是span</span>
    <ul>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
    </ul>
    <div class="demo">
        <div class="box">
            <span class="erweima">×</span>
        </div>
```

```
    </div>

    <script>
        // 1. 父节点 parentNode
        var erweima =
document.querySelector('.erweima');
        // var box = document.querySelector('.box');
        // 得到的是离元素最近的父级节点(亲爸爸) 如果找不
到父节点就返回为 null
        console.log(erweima.parentNode);
    </script>
</body>
```

5.3、子结点

```
parentNode.childNodes(标准)
```

- `parentNode.childNodes` 返回包含指定节点的子节点的集合，该集合为即时更新的集合
- 返回值包含了所有的子结点，包括元素节点，文本节点等
- 如果只想要获得里面的元素节点，则需要专门处理。所以我们一般不提倡使用 `childNodes`

```
parentNode.children(非标准)
```

- `parentNode.children` 是一个只读属性，返回所有的子元素节点
- 它只返回子元素节点，其余节点不返回 （**这个是我们重点掌握的**）
- 虽然 children 是一个非标准，但是得到了各个浏览器的支持，因此我们可以放心使用

```
<body>
    <ul>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
    </ul>
    <ol>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
        <li>我是li</li>
    </ol>
    <script>
        // DOM 提供的方法（API）获取
        var ul = document.querySelector('ul');
        var lis = ul.querySelectorAll('li');
        // 1. 子节点  childNodes 所有的子节点 包含 元
素节点 文本节点等等
```

```
        console.log(ul.childNodes);
        console.log(ul.childNodes[0].nodeType);
        console.log(ul.childNodes[1].nodeType);
        // 2. children 获取所有的子元素节点 也是我们实际开发常用的
        console.log(ul.children);
    </script>
</body>
```

5.3.1、第一个子结点

```
parentNode.firstChild
```

- `firstChild` 返回第一个子节点，找不到则返回null
- 同样，也是包含所有的节点

5.3.2、最后一个子结点

```
parentNode.lastChild
```

- `lastChild` 返回最后一个子节点，找不到则返回null
- 同样，也是包含所有的节点

```
<body>
    <ol>
        <li>我是li1</li>
        <li>我是li2</li>
        <li>我是li3</li>
        <li>我是li4</li>
        <li>我是li5</li>
    </ol>
    <script>
        var ol = document.querySelector('ol');
        // 1. firstChild 第一个子节点 不管是文本节点还是元素节点
        console.log(ol.firstChild);
        console.log(ol.lastChild);
        // 2. firstElementChild 返回第一个子元素节点 ie9才支持
        console.log(ol.firstElementChild);
        console.log(ol.lastElementChild);
        // 3. 实际开发的写法  既没有兼容性问题又返回第一个子元素
        console.log(ol.children[0]);                //第一个子元素节点
```

```
        console.log(ol.children[ol.children.length - 1]);//最后一个子元素节点
    </script>
</body>
```

1. 我是li1
2. 我是li2
3. 我是li3
4. 我是li4
5. 我是li5

5.3.3、第一个子结点(兼容性)

```
parentNode.firstElementChild
```

- `firstElementChild` 返回第一个子节点，找不到则返回null
- 有兼容性问题，IE9以上才支持

5.3.4、最后一个子结点(兼容性)

```
parentNode.lastElementChild
```

- `lastElementChild` 返回最后一个子节点，找不到则返回null
- 有兼容性问题，IE9以上才支持

5.3.5、解决方案

实际开发中，firstChild 和 lastChild 包含其他节点，操作不方便，而 firstElementChild 和 lastElementChild 又有兼容性问题，那么我们如何获取第一个子元素节点或最后一个子元素节点呢?

> 解决方案

- 如果想要第一个子元素节点，可以使用 `parentNode.chilren[0]`
- 如果想要最后一个子元素节点，可以使用

```
// 数组元素个数减1 就是最后一个元素的索引号
parentNode.chilren[parentNode.chilren.length - 1]
```

- 示例：

```
<body>
    <ol>
        <li>我是li1</li>
        <li>我是li2</li>
        <li>我是li3</li>
        <li>我是li4</li>
    </ol>
    <script>
        var ol = document.querySelector('ol');
        // 1.firstChild 获取第一个子结点的，包含文本结点和元素结点
        console.log(ol.firstChild);
        // 返回的是文本结点 #text(第一个换行结点)

        console.log(ol.lastChild);
        // 返回的是文本结点 #text(最后一个换行结点)
        // 2. firstElementChild 返回第一个子元素结点
        console.log(ol.firstElementChild);
        // <li>我是li1</li>

        // 第2个方法有兼容性问题，需要IE9以上才支持
        // 3.实际开发中，既没有兼容性问题，又返回第一个子元素
        console.log(ol.children[0]);
        // <li>我是li1</li>
        console.log(ol.children[3]);
        // <li>我是li4</li>
        // 当里面li个数不唯一时候，需要取到最后一个结点时这么写
        console.log(ol.children[ol.children.length - 1]);
    </script>
</body>
```

5.4、兄弟节点

5.4.1、下一个兄弟节点

```
node.nextSibling
```

- `nextSibling` 返回当前元素的下一个兄弟元素节点，找不到则返回null
- 同样，也是包含所有的节点

5.4.2、上一个兄弟节点

```
node.previousSibling
```

- `previousSibling` 返回当前元素上一个兄弟元素节点，找不到则返回null
- 同样，也是包含所有的节点

5.4.3、下一个兄弟节点(兼容性)

```
node.nextElementSibling
```

- `nextElementSibling` 返回当前元素下一个兄弟元素节点，找不到则返回null
- 有兼容性问题，IE9才支持

5.4.4、上一个兄弟节点(兼容性)

```
node.previousElementSibling
```

- `previousElementSibling` 返回当前元素上一个兄弟元素节点，找不到则返回null
- 有兼容性问题，IE9才支持

示例

```
<body>
    <div>我是div</div>
    <span>我是span</span>
    <script>
        var div = document.querySelector('div');
        // 1.nextSibling 下一个兄弟节点 包含元素节点或者 文本节点等等
        console.log(div.nextSibling);         // #text
        console.log(div.previousSibling);     // #text
        // 2. nextElementSibling 得到下一个兄弟元素节点
        console.log(div.nextElementSibling);
//<span>我是span</span>
```

```
 console.log(div.previousElementSibling);//null
    </script>
</body>
```

如何解决兼容性问题？

答：自己封装一个兼容性的函数

```
function getNextElementSibling(element) {
    var el = element;
    while(el = el.nextSibling) {
        if(el.nodeType === 1){
            return el;
        }
    }
    return null;
}
```

5.5、创建节点

```
document.createElement('tagName');
```

- `document.createElement()` 方法创建由 tagName 指定的HTML 元素
- 因为这些元素原先不存在，是根据我们的需求动态生成的，所以我们也称为**动态创建元素节点**

5.5.1、添加节点

```
node.appendChild(child)
```

- `node.appendChild()` 方法将一个节点添加到指定父节点的子节点列表**末尾**。类似于 CSS 里面的 after 伪元素。

```
node.insertBefore(child,指定元素)
```

- `node.insertBefore()` 方法将一个节点添加到父节点的指定子节点**前面**。类似于 CSS 里面的 before 伪元素。

示例

```
<body>
    <ul>
        <li>123</li>
    </ul>
```

```
    <script>
        // 1. 创建节点元素节点
        var li = document.createElement('li');
        // 2. 添加节点 node.appendChild(child)  node 父级  child 是子级 后面追加元素  类似于数组中的push
        // 先获取父亲ul
        var ul = document.querySelector('ul');
        ul.appendChild(li);
        // 3. 添加节点 node.insertBefore(child, 指定元素);
        var lili = document.createElement('li');
        ul.insertBefore(lili, ul.children[0]);
        // 4. 我们想要页面添加一个新的元素分两步: 1. 创建元素 2. 添加元素
    </script>
</body>
```

5.5.2、删除节点

```
node.removeChild(child)
```

- `node.removeChild()` 方法从 DOM 中删除一个子节点，返回删除的节点

5.5.3、复制节点(克隆节点)

```
node.cloneNode()
```

- `node.cloneNode()` 方法返回调用该方法的节点的一个副本。 也称为克隆节点/拷贝节点
- 如果括号参数为空或者为 false ，则是浅拷贝，即只克隆复制节点本身，不克隆里面的子节点
- 如果括号参数为 true ，则是深度拷贝，会复制节点本身以及里面所有的子节点

示例

```
<body>
    <ul>
        <li>1111</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <script>
        var ul = document.querySelector('ul');
        // 1. node.cloneNode(); 括号为空或者里面是false 浅拷贝 只复制标签不复制里面的内容
        // 2. node.cloneNode(true); 括号为true 深拷贝复制标签复制里面的内容
```

```
        var lili = ul.children[0].cloneNode(true);
        ul.appendChild(lili);
    </script>
</body>
14
```

- 1111
- 2
- 3
- 1111 → 深拷贝，复制标签里面的内容

5.5.4、面试题

三种动态创建元素的区别

- doucument.write()
- element.innerHTML
- document.createElement()

区别：

- `document.write()` 是直接将内容写入页面的内容流，但是文档流执行完毕，则它会导致页面全部重绘
- `innerHTML` 是将内容写入某个 DOM 节点，不会导致页面全部重绘
- `innerHTML` 创建多个元素效率更高（不要拼接字符串，采取数组形式拼接），结构稍微复杂

```
<body>
    <div class="innner"></div>
    <div class="create"></div>
    <script>
        // 2. innerHTML 创建元素
        var inner =
document.querySelector('.inner');
        // 2.1 innerHTML 用拼接字符串方法
        for (var i = 0; i <= 100; i++) {
            inner.innerHTML += '<a href="#">百度
</a>';
        }
        // 2.2 innerHTML 用数组形式拼接
        var arr = [];
        for (var i = 0; i <= 100; i++) {
```

```
            arr.push('<a href="#">百度</a>');
        }
        inner.innerHTML = arr.join('');

        // 3.document.createElement() 创建元素
        var create =
document.querySelector('.create');
        var a = document.createElement('a');
        create.appendChild(a);
    </script>
</body>
1920212223
```

- `createElement()` 创建多个元素效率稍低一点点，但是结构更清晰

> 总结：不同浏览器下， innerHTML 效率要比 createElement 高

6、DOM核心

对于DOM操作，我们主要针对子元素的操作，主要有

- 创建
- 增
- 删
- 改
- 查
- 属性操作
- 时间操作

6.1、创建

1. document.write
2. innerHTML
3. createElement

6.2、增

1. appendChild
2. insertBefore

6.3、删

1. removeChild

6.4、改

- 主要修改dom的元素属性，dom元素的内容、属性、表单的值等

1. 修改元素属性：src、href、title 等
2. 修改普通元素内容：innerHTML、innerText
3. 修改表单元素：value、type、disabled
4. 修改元素样式：style、className

6.5、查

- 主要获取查询dom的元素

1. **DOM提供的API方法**：getElementById、getElementsByTagName (古老用法，不推荐)
2. **H5提供的新方法**：querySelector、querySelectorAll (提倡)
3. 利用节点操作获取元素：父(parentNode)、子(children)、兄(previousElementSibling、nextElementSibling) 提倡

6.6、属性操作

- 主要针对于自定义属性

1. setAttribute：设置dom的属性值
2. getAttribute：得到dom的属性值
3. removeAttribute：移除属性

7、事件高级

7.1、注册事件(绑定事件)

给元素添加事件，称为注册事件或者绑定事件。

注册事件有两种方式：传统方式和方法监听注册方式

传统注册方式	方法监听注册方式
利用 on 开头的事件 onclick	w3c 标准推荐方式
`<button onclick = "alert("hi")"></button>`	addEventListener() 它是一个方法
btn.onclick = function() {}	IE9 之前的 IE 不支持此方法，可使用 attachEvent() 代替
特点：注册事件的唯一性	特点：同一个元素同一个事件可以注册多个监听器
同一个元素同一个事件只能设置一个处理函数，最后注册的处理函数将会覆盖前面注册的处理函数	按注册顺序依次执行

①addEventListener事件监听方式

- `eventTarget.addEventListener()` 方法将指定的监听器注册到 eventTarget（目标对象）上
- 当该对象触发指定的事件时，就会执行事件处理函数

```
eventTarget.addEventListener(type,listener[,useCapture])
```

该方法接收三个参数：

- `type` :事件类型字符串，比如click,mouseover,注意这里不要带on
- `listener` ： 事件处理函数，事件发生时，会调用该监听函数
- `useCapture` ： 可选参数，是一个布尔值，默认是 false。学完 DOM 事件流后，我们再进一步学习

```
<body>
    <button>传统注册事件</button>
    <button>方法监听注册事件</button>
    <button>ie9 attachEvent</button>
    <script>
        var btns = document.querySelectorAll('button');
        // 1. 传统方式注册事件
        btns[0].onclick = function() {
            alert('hi');
        }
        btns[0].onclick = function() {
                alert('hao a u');
```

```
            }
            // 2. 事件监听注册事件 addEventListener
            // (1) 里面的事件类型是字符串 所以加引号 而且不带on
            // (2) 同一个元素 同一个事件可以添加多个侦听器（事件处理程序）
        btns[1].addEventListener('click', function() {
            alert(22);
        })
        btns[1].addEventListener('click', function() {
                alert(33);
            })
            // 3. attachEvent ie9以前的版本支持
        btns[2].attachEvent('onclick', function() {
            alert(11);
        })
    </script>
</body>
```

②attachEvent事件监听方式(兼容)

- `eventTarget.attachEvent()` 方法将指定的监听器注册到 eventTarget（目标对象） 上
- 当该对象触发指定的事件时，指定的回调函数就会被执行

```
eventTarget.attachEvent(eventNameWithOn,callback)
```

该方法接收两个参数：

- `eventNameWithOn`：事件类型字符串，比如 onclick 、onmouseover ，这里要带 on
- `callback`： 事件处理函数，当目标触发事件时回调函数被调用
- ie9以前的版本支持

③注册事件兼容性解决方案

兼容性处理的原则：首先照顾大多数浏览器，再处理特殊浏览器

```
function addEventListener(element, eventName, fn) {
      // 判断当前浏览器是否支持 addEventListener 方法
      if (element.addEventListener) {
        element.addEventListener(eventName, fn);  //
第三个参数 默认是false
      } else if (element.attachEvent) {
        element.attachEvent('on' + eventName, fn);
      } else {
        // 相当于 element.onclick = fn;
        element['on' + eventName] = fn;
 }
```

7.2、删除事件(解绑事件)

7.2.1、removeEventListener删除事件方式

```
eventTarget.removeEventListener(type,listener[,useCa
pture]);
```

该方法接收三个参数：

- `type` :事件类型字符串，比如click,mouseover,注意这里不要带on
- `listener` ： 事件处理函数，事件发生时，会调用该监听函数
- `useCapture` ： 可选参数，是一个布尔值，默认是 false。学完 DOM 事件流后，我们再进一步学习

7.2.2、detachEvent删除事件方式(兼容)

```
eventTarget.detachEvent(eventNameWithOn,callback);
```

该方法接收两个参数：

- `eventNameWithOn` ： 事件类型字符串，比如 onclick 、onmouseover ，这里要带 on
- `callback` ： 事件处理函数，当目标触发事件时回调函数被调用
- ie9以前的版本支持

7.2.3、传统事件删除方式

```
eventTarget.onclick = null;
```

事件删除示例：

```
<body>
    <div>1</div>
    <div>2</div>
```

```
    <div>3</div>
    <script>
        var divs = document.querySelectorAll('div');
        divs[0].onclick = function() {
            alert(11);
            // 1. 传统方式删除事件
            divs[0].onclick = null;
        }
        // 2.removeEventListener 删除事件
        divs[1].addEventListener('click',fn);   //里面的fn不需要调用加小括号

        function fn(){
            alert(22);
            divs[1].removeEventListener('click',fn);
        }
        // 3.IE9 中的删除事件方式
        divs[2].attachEvent('onclick',fn1);
        function fn1() {
            alert(33);
            divs[2].detachEvent('onclick',fn1);
        }
    </script>

</body>
1920212223242526 27
```

7.2.4、删除事件兼容性解决方案

```
 function removeEventListener(element, eventName, fn) {
      // 判断当前浏览器是否支持 removeEventListener 方法
      if (element.removeEventListener) {
        element.removeEventListener(eventName, fn);
 // 第三个参数 默认是false
      } else if (element.detachEvent) {
        element.detachEvent('on' + eventName, fn);
      } else {
        element['on' + eventName] = null;
 }
```

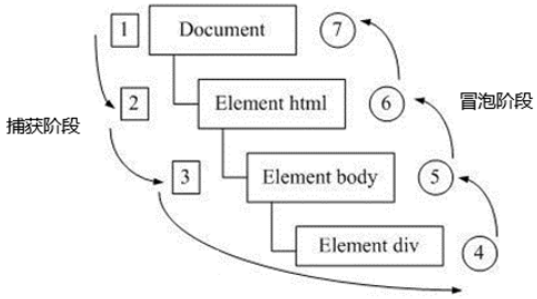
7.3、DOM事件流

- 事件流描述的是从页面中接收事件的顺序
- 事件发生时会在元素节点之间按照特定的顺序传播，这个传播过程即DOM事件流

比如我们给一个div 注册了点击事件：

DOM 事件流分为3个阶段：

1. 捕获阶段
2. 当前目标阶段
3. 冒泡阶段

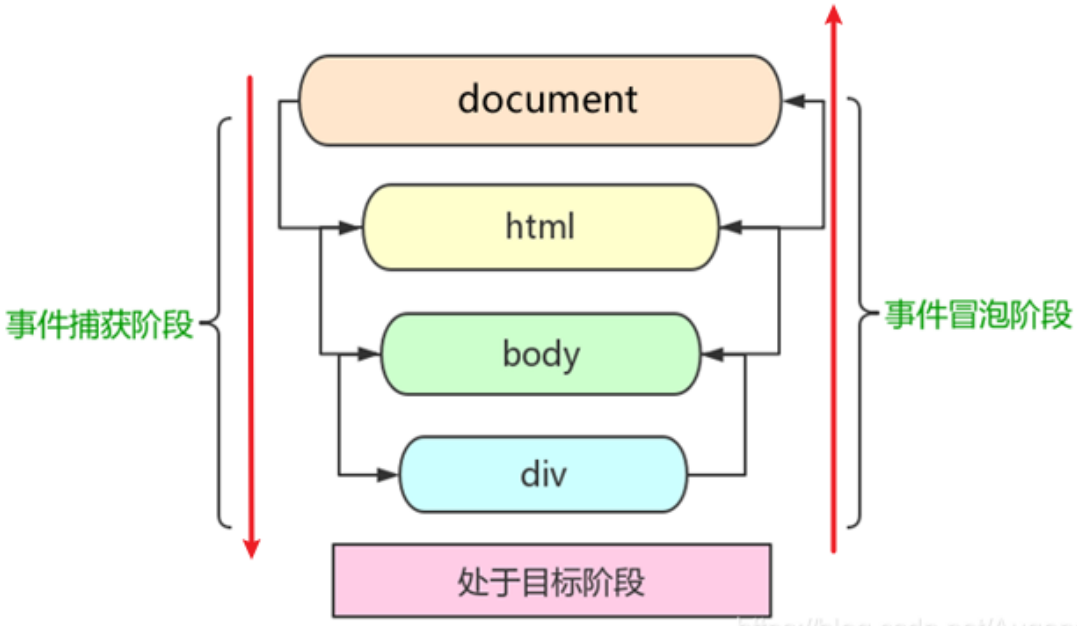


DOM事件流

- 事件冒泡： IE 最早提出，事件开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点的过程。
- 事件捕获： 网景最早提出，由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收的过程。

加深理解：

我们向水里面扔一块石头，首先它会有一个下降的过程，这个过程就可以理解为从最顶层向事件发生的最具体元素（目标点）的捕获过程；之后会产生泡泡，会在最低点（ 最具体元素）之后漂浮到水面上，这个过程相当于事件冒泡。



7.3.1、捕获阶段

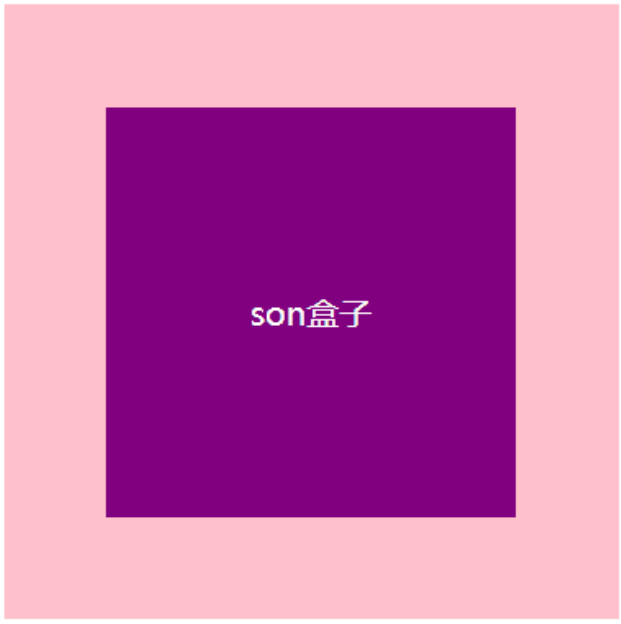
- document -> html -> body -> father -> son

两个盒子嵌套，一个父盒子一个子盒子，我们的需求是当点击父盒子时弹出 father ，当点击子盒子时弹出 son

```
<body>
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // dom 事件流 三个阶段
        // 1. JS 代码中只能执行捕获或者冒泡其中的一个阶段。
        // 2. onclick 和 attachEvent（ie） 只能得到冒泡阶段。
        // 3. 捕获阶段 如果addEventListener 第三个参数是 true 那么则处于捕获阶段  document -> html -> body -> father -> son
        var son = document.querySelector('.son');
        son.addEventListener('click', function() {
             alert('son');
        }, true);
        var father = document.querySelector('.father');
        father.addEventListener('click', function() {
            alert('father');
        }, true);
    </script>
</body>
19
```

但是因为DOM流的影响，我们点击子盒子，会先弹出 father，之后再弹出 son

三天/code/11-DOM%20事件流三个阶段.html



这是因为捕获阶段由 DOM 最顶层节点开始，然后逐级向下传播到到最具体的元素接收

- document -> html -> body -> father -> son
- 先看 document 的事件，没有；再看 html 的事件，没有；再看 body 的事件，没有；再看 father 的事件，有就先执行；再看 son 的事件，再执行。

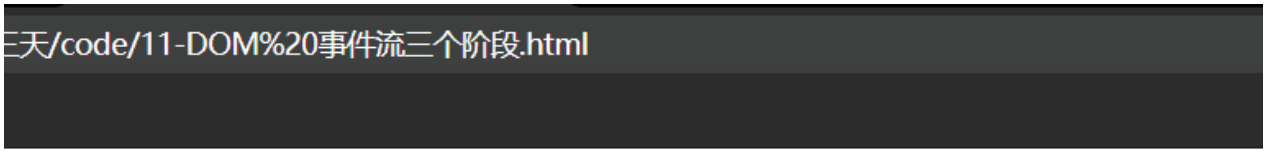
7.3.2、冒泡阶段

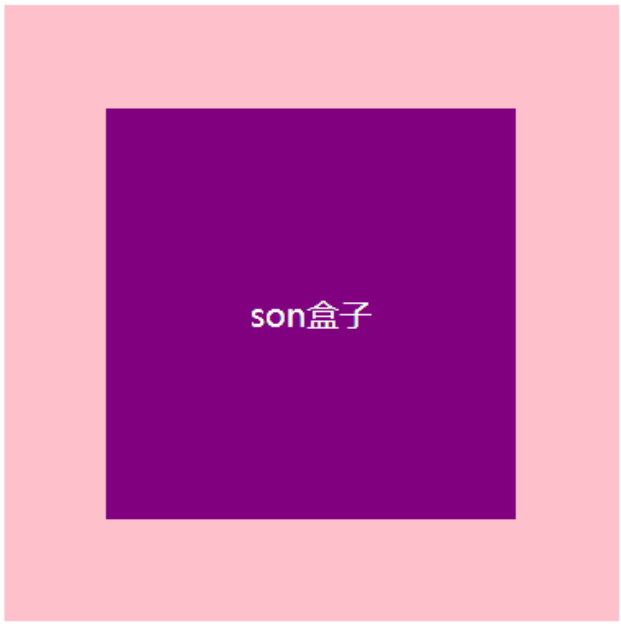
- son -> father ->body -> html -> document

```
<body>
    <div class="father">
        <div class="son">son盒子</div>
    </div>
    <script>
        // 4. 冒泡阶段 如果addEventListener 第三个参数是 false 或者 省略 那么则处于冒泡阶段  son -> father ->body -> html -> document
        var son = document.querySelector('.son');
        son.addEventListener('click', function() {
            alert('son');
        }, false);
        var father = document.querySelector('.father');
```

```
        father.addEventListener('click', function()
{
            alert('father');
        }, false);
        document.addEventListener('click',
function() {
            alert('document');
        })
    </script>
</body>
19
```

我们点击子盒子，会弹出 son、father、document





这是因为冒泡阶段开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点

- son -> father ->body -> html -> document

7.3.3、小结

- JS 代码中只能执行捕获或者冒泡其中的一个阶段
- `onclick` 和 `attachEvent` 只能得到冒泡阶段
- `addEventListener(type,listener[,useCapture])` 第三个参数如果是 true，表示在事件捕获阶段调用事件处理程序；如果是 false (不写默认就是false),表示在事件冒泡阶段调

用事件处理程序

- 实际开发中我们很少使用事件捕获，我们更关注事件冒泡。
- 有些事件是没有冒泡的，比如 onblur、onfocus、onmouseenter、onmouseleave

7.4、事件对象

```
eventTarget.onclick = function(event) {
    // 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt
}
eventTarget.addEventListener('click', function(event) {
    // 这个 event 就是事件对象，我们还喜欢的写成 e 或者 evt
})
6
```

- 官方解释：event 对象代表事件的状态，比如键盘按键的状态、鼠标的位置、鼠标按钮的状态
- 简单理解：
    - 事件发生后，跟事件相关的一系列信息数据的集合都放到这个对象里面
    - 这个对象就是事件对象 event，它有很多属性和方法，比如“
        - 谁绑定了这个事件
        - 鼠标触发事件的话，会得到鼠标的相关信息，如鼠标位置
        - 键盘触发事件的话，会得到键盘的相关信息，如按了哪个键
- 这个 event 是个形参，系统帮我们设定为事件对象，不需要传递实参过去
- 当我们注册事件时， event 对象就会被系统自动创建，并依次传递给事件监听器（事件处理函数）

```
<body>
    <div>123</div>
    <script>
        // 事件对象
        var div = document.querySelector('div');
        div.onclick = function(e) {
                // console.log(e);
                // console.log(window.event);
                // e = e || window.event;
                console.log(e);


            }
```

```
        // 1. event 就是一个事件对象 写到我们侦听函数的小括号里面 当形参来看
        // 2. 事件对象只有有了事件才会存在，它是系统给我们自动创建的，不需要我们传递参数
        // 3. 事件对象 是 我们事件的一系列相关数据的集合 跟事件相关的 比如鼠标点击里面就包含了鼠标的相关信息，鼠标坐标啊，如果是键盘事件里面就包含的键盘事件的信息 比如判断用户按下了那个键
        // 4. 这个事件对象我们可以自己命名 比如 event 、 evt、 e
        // 5. 事件对象也有兼容性问题 ie 通过 window.event 兼容性的写法  e = e || window.event;
    </script>
</body>
1920
```

7.4.1、事件对象的兼容性方案

事件对象本身的获取存在兼容问题：

1. 标准浏览器中是浏览器给方法传递的参数，只需要定义形参 e 就可以获取到。
2. 在 IE6~8 中，浏览器不会给方法传递参数，如果需要的话，需要到 window.event 中获取查找

解决：

```
e = e || window.event;
1
```

7.4.2、事件对象的常见属性和方法

事件对象属性方法	说明
e.target	返回触发事件的对象 标准
e.srcElement	返回触发事件的对象 非标准 ie6-8使用
e.type	返回事件的类型 比如 `click` `mouseover` 不带on
e.cancelBubble	该属性阻止冒泡，非标准，ie6-8使用
e.returnValue	该属性阻止默认行为 非标准，ie6-8使用
e.preventDefault()	该方法阻止默认行为 标准 比如不让链接跳转
e.stopPropagation()	阻止冒泡 标准

e.target 和 this 的区别：

- this 是事件绑定的元素， 这个函数的调用者（绑定这个事件的元素）
- e.target 是事件触发的元素。

7.5、事件对象阻止默认行为

```
<body>
    <div>123</div>
    <a href="http://www.baidu.com">百度</a>
    <form action="http://www.baidu.com">
        <input type="submit" value="提交"
name="sub">
    </form>
    <script>
        // 常见事件对象的属性和方法
        // 1. 返回事件类型
        var div = document.querySelector('div');
        div.addEventListener('click', fn);
        div.addEventListener('mouseover', fn);
        div.addEventListener('mouseout', fn);

        function fn(e) {
            console.log(e.type);

        }
        // 2. 阻止默认行为（事件） 让链接不跳转 或者让
提交按钮不提交
        var a = document.querySelector('a');
        a.addEventListener('click', function(e) {
                e.preventDefault(); //  dom 标准写法
            })
            // 3. 传统的注册方式
        a.onclick = function(e) {
            // 普通浏览器 e.preventDefault();  方法
            // e.preventDefault();
            // 低版本浏览器 ie  returnValue  属性
            // e.returnValue;
            // 我们可以利用return false 也能阻止默认行
为 没有兼容性问题 特点:  return 后面的代码不执行了， 而
且只限于传统的注册方式
            return false;
            alert(11);
        }
    </script>
</body>
1920212223242526272829303132333435
```

7.6、阻止事件冒泡

事件冒泡：开始时由最具体的元素接收，然后逐级向上传播到到 DOM 最顶层节点

事件冒泡本身的特性，会带来的坏处，也会带来的好处，需要我们灵活掌握。

- 标准写法

```
e.stopPropagation();
1
```

- 非标准写法： IE6-8 利用对象事件 cancelBubble属性

```
e.cancelBubble = true;
1
<body>
    <div class="father">
        <div class="son">son儿子</div>
    </div>
    <script>
        // 常见事件对象的属性和方法
        // 阻止冒泡  dom 推荐的标准 stopPropagation()
        var son = document.querySelector('.son');
        son.addEventListener('click', function(e) {
            alert('son');
            e.stopPropagation(); // stop 停止
Propagation 传播
            e.cancelBubble = true; // 非标准 cancel
取消 bubble 泡泡
        }, false);

        var father =
document.querySelector('.father');
        father.addEventListener('click', function()
{
            alert('father');
        }, false);
        document.addEventListener('click',
function() {
            alert('document');
        })
    </script>
</body>
1920212223
```

7.6.1、阻止事件冒泡的兼容性解决方案

```
if(e && e.stopPropagation){
      e.stopPropagation();
  }else{
      window.event.cancelBubble = true;
  }
```

4.4.4 e.target 与 this

> e.target 与 this 的区别

- `this` 是事件绑定的元素，这个函数的调用者(绑定这个事件的元素)
- `e.target` 是事件触发的元素

```
<body>
    <div>123</div>
    <ul>
        <li>abc</li>
        <li>abc</li>
        <li>abc</li>
    </ul>
    <script>
        // 常见事件对象的属性和方法
        // 1. e.target 返回的是触发事件的对象（元素）
this 返回的是绑定事件的对象（元素）
        // 区别 ： e.target 点击了那个元素，就返回那个
元素 this 那个元素绑定了这个点击事件，那么就返回谁
        var div = document.querySelector('div');
        div.addEventListener('click', function(e) {
            console.log(e.target);
            console.log(this);

        })
        var ul = document.querySelector('ul');
        ul.addEventListener('click', function(e) {
                // 我们给ul 绑定了事件  那么this 就指
向ul
                console.log(this);
                console.log(e.currentTarget);

                // e.target 指向我们点击的那个对象 谁
触发了这个事件 我们点击的是li e.target 指向的就是li
                console.log(e.target);

            })
```

```
            // 了解兼容性
            // div.onclick = function(e) {
            //     e = e || window.event;
            //     var target = e.target ||
e.srcElement;
            //     console.log(target);

        // }
        // 2. 了解 跟 this 有个非常相似的属性
currentTarget  ie不认识
    </script>
</body>
```

4.4.5 事件对象的兼容性

事件对象本身的获取存在兼容问题：

- 标准浏览器中浏览器是给方法传递的参数，只需定义形参e就可以获取到
- 在 IE6 -> 8 中，浏览器不会给方法传递参数，如果需要的话，需要到 `window.event` 中获取查找

> 解决方案

- `e = e || window.event`

```
<body>
    <div>123</div>
    <script>
        // 事件对象
        var div = document.querySelector('div');
        div.onclick = function(e) {
                // e = e || window.event;
                console.log(e);
                // 事件对象也有兼容性问题 ie 通过
window.event 兼容性的写法  e = e || window.event;

            }
</body>
1112
```

7.7、事件委托

- 事件委托也称为事件代理，在 jQuery 里面称为事件委派
- 事件委托的原理

- **不是每个子节点单独设置事件监听器，而是事件监听器设置在其父节点上，然后利用冒泡原理影响设置每个子节点**

```
<body>
    <ul>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
        <li>知否知否，点我应有弹框在手！</li>
    </ul>
    <script>
        // 事件委托的核心原理：给父节点添加侦听器， 利用事件冒泡影响每一个子节点
        var ul = document.querySelector('ul');
        ul.addEventListener('click', function(e) {
            // alert('知否知否，点我应有弹框在手！');
            // e.target 这个可以得到我们点击的对象
            e.target.style.backgroundColor = 'pink';
            // 点了谁，就让谁的style里面的backgroundColor颜色变为pink
        })
    </script>
</body>
19
```

以上案例：给 ul 注册点击事件，然后利用事件对象的 target 来找到当前点击的 li，因为点击 li，事件会冒泡到 ul 上， ul 有注册事件，就会触发事件监听器。

7.8、常见的鼠标事件

鼠标事件	触发条件
onclick	鼠标点击左键触发
onmouseover	鼠标经过触发
onmouseout	鼠标离开触发
onfocus	获得鼠标焦点触发
onblur	失去鼠标焦点触发
onmousemove	鼠标移动触发
onmouseup	鼠标弹起触发
onmousedown	鼠标按下触发

7.8.1、禁止鼠标右键与鼠标选中

- `contextmenu` 主要控制应该何时显示上下文菜单，主要用于程序员取消默认的上下文菜单
- `selectstart` 禁止鼠标选中

```
<body>
    <h1>我是一段不愿意分享的文字</h1>
    <script>
        // 1. contextmenu 我们可以禁用右键菜单
        document.addEventListener('contextmenu',
function(e) {
            e.preventDefault(); // 阻止默认行为
        })
        // 2. 禁止选中文字 selectstart
        document.addEventListener('selectstart',
function(e) {
            e.preventDefault();

        })
    </script>
</body>
14
```

7.8.2、鼠标事件对象

- **event**对象代表事件的状态，跟事件相关的一系列信息的集合
- 现阶段我们主要是用鼠标事件对象 **MouseEvent** 和键盘事件对象 **KeyboardEvent**。

鼠标事件对象	说明
e.clientX	返回鼠标相对于浏览器窗口**可视区**的X坐标
e.clientY	返回鼠标相对于浏览器窗口**可视区**的Y坐标
e.pageX（重点）	返回鼠标相对于文档页面的X坐标 IE9+ 支持
e.pageY（重点）	返回鼠标相对于文档页面的Y坐标 IE9+ 支持
e.screenX	返回鼠标相对于电脑屏幕的X坐标
e.screenY	返回鼠标相对于电脑屏幕的Y坐标

```
<body>
    <script>
        // 鼠标事件对象 MouseEvent
        document.addEventListener('click',
function(e) {
            // 1. client 鼠标在可视区的x和y坐标
            console.log(e.clientX);
            console.log(e.clientY);
            console.log('----------------------');

            // 2. page 鼠标在页面文档的x和y坐标
            console.log(e.pageX);
            console.log(e.pageY);
            console.log('----------------------');

            // 3. screen 鼠标在电脑屏幕的x和y坐标
            console.log(e.screenX);
            console.log(e.screenY);

        })
    </script>
</body>
192021
```

7.9、常用的键盘事件

键盘事件	触发条件
onkeyup	某个键盘按键被松开时触发
onkeydown	某个键盘按键被按下时触发
onkeypress	某个键盘按键被按下时触发，但是它不识别功能键，比如 ctrl shift 箭头等

- **如果使用addEventListener 不需要加 on**
- `onkeypress` 和前面2个的区别是，它不识别功能键，比如左右箭头，shift 等
- 三个事件的执行顺序是： keydown – keypress — keyup

```
<body>
    <script>
        // 常用的键盘事件
        //1. keyup 按键弹起的时候触发
        // document.onkeyup = function() {
        //         console.log('我弹起了');

        //     }
        document.addEventListener('keyup',
function() {
            console.log('我弹起了');
        })

        //3. keypress 按键按下的时候触发  不能识别功能
键 比如 ctrl shift 左右箭头啊
        document.addEventListener('keypress',
function() {
                console.log('我按下了press');
            })
            //2. keydown 按键按下的时候触发  能识别功
能键 比如 ctrl shift 左右箭头啊
        document.addEventListener('keydown',
function() {
                console.log('我按下了down');
            })
            // 4. 三个事件的执行顺序  keydown --
keypress -- keyup
    </script>
</body>
1920212223
```

7.9.1、键盘对象属性

键盘事件对象 属性	说明
keyCode	返回该键值的ASCII值

- `onkeydown` 和 `onkeyup` 不区分字母大小写， `onkeypress` 区分字母大小写。
- 在我们实际开发中，我们更多的使用keydown和keyup， 它能识别所有的键（包括功能键）
- `Keypress` 不识别功能键，但是 `keyCode` 属性能区分大小写，返回不同的ASCII值

```
```

```
<body>
    <script>
        // 键盘事件对象中的keyCode属性可以得到相应键的ASCII码值
        // 1. 我们的keyup 和keydown事件不区分字母大小写  a 和 A 得到的都是65
        // 2. 我们的keypress 事件 区分字母大小写  a 97 和 A 得到的是65
        document.addEventListener('keyup', function(e) {
            console.log('up:' + e.keyCode);
            // 我们可以利用keycode返回的ASCII码值来判断用户按下了那个键
            if (e.keyCode === 65) {
                alert('您按下的a键');
            } else {
                alert('您没有按下a键')
            }

        })
        document.addEventListener('keypress', function(e) {
            console.log('press:' + e.keyCode);
        })
    </script>
</body>
```

1、面向对象

面向对象更贴近我们的实际生活, 可以使用面向对象描述现实世界事物. 但是事物分为具体的事物和抽象的事物

面向对象的思维特点：

1. 抽取（抽象）对象共用的属性和行为组织(封装)成一个类(模板)
2. 对类进行实例化, 获取类的对象

1.1、对象

在 JavaScript 中，对象是一组无序的相关属性和方法的集合，所有的事物都是对象，例如字符串、数值、数组、函数等。

对象是由属性和方法组成的

- 属性：事物的**特征，*在对象中用*属性**来表示
- 方法：事物的**行为，*在对象中用*方法**来表示

1.2、类

在 ES6 中新增加了类的概念，可以使用 class 关键字声明一个类，之后以这个类来实例化对象。

- 类抽象了对象的公共部分，它泛指某一大类（class）
- 对象特指某一个，通过类实例化一个具体的对象

1.2.1、创建类

```
class name {
    // class body
}
123
```

- 创建实例

```
var XX = new name();
1
```

注意：类必须使用 `new` 实例化对象

1.2.2、构造函数

constructor() 方法是类的构造函数(默认方法)，用于传递参数,返回实例对象，通过 new 命令生成对象实例时，自动调用该方法。如果没有显示定义, 类内部会自动给我们创建一个constructor()

```
<script>
    // 1. 创建类 class  创建一个 明星类
    class Star {
        // constructor 构造器或者构造函数
        constructor(uname, age) {
            this.uname = uname;
            this.age = age;
        }
    }

    // 2. 利用类创建对象 new
    var ldh = new Star('刘德华', 18);
    var zxy = new Star('张学友', 20);
    console.log(ldh);
    console.log(zxy);
</script>
141516
```

- 通过 class 关键字创建类，类名我们还是习惯性**定义首字母大写**
- 类里面有个 `constructor` 函数，可以接收传递过来的参数，同时返回实例对象

- `constructor` 函数只要 new 生成实例时，就会自动调用这个函数，如果我们不写这个函数，类也会自动生成这个函数
- 最后注意语法规范
  - 创建类➡类名后面不要加小括号
  - 生成实例➡类名后面加小括号
  - 构造函数不需要加 function 关键字

1.2.3、类中添加方法

语法：

```
class Person {
  constructor(name,age) {
      // constructor 称为构造器或者构造函数
      this.name = name;
      this.age = age;
    }
   say() {
      console.log(this.name + '你好');
    }
}
var ldh = new Person('刘德华', 18);
ldh.say()
1112
```

注意： 方法之间不能加逗号分隔，同时方法不需要添加 function 关键字。

```
<script>
    // 1. 创建类 class  创建一个 明星类
    class Star {
        // 类的共有属性放到 constructor 里面
        constructor(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        sing(song) {
            console.log(this.uname + song);
        }
    }

    // 2. 利用类创建对象 new
    var ldh = new Star('刘德华', 18);
    var zxy = new Star('张学友', 20);
    console.log(ldh);
    console.log(zxy);
```

```
    // (1) 我们类里面所有的函数不需要写function
    // (2) 多个函数方法之间不需要添加逗号分隔
    ldh.sing('冰雨');
    zxy.sing('李香兰');
</script>
1920212223
```

- 类的共有属性放到 `constructor` 里面
- 类里面的函数都不需要写 `function` 关键字

1.3 、类的继承

现实中的继承：子承父业，比如我们都继承了父亲的姓。

程序中的继承：子类可以继承父类的一些属性和方法。

语法：

```
// 父类
class Father {

}
// 子类继承父类
class Son extends Father {

}
```

看一个实例：

```
<script>
    // 父类有加法方法
    class Father {
        constructor(x, y) {
            this.x = x;
            this.y = y;
        }
        sum() {
            console.log(this.x + this.y);
        }
    }
    // 子类继承父类加法方法 同时 扩展减法方法
    class Son extends Father {
        constructor(x, y) {
            // 利用super 调用父类的构造函数
            // super 必须在子类this之前调用
            super(x, y);
            this.x = x;
```

```
            this.y = y;
        }
        subtract() {
            console.log(this.x - this.y);
        }
    }
    var son = new Son(5, 3);
    son.subtract();
    son.sum();
</script>
19202122232425262728
```

1.4、super关键字

- `super` 关键字用于访问和调用对象父类上的函数，可以调用父类的构造函数，也可以调用父类的普通函数

1.4.1、调用父类的构造函数

语法：

```
// 父类
class Person {
    constructor(surname){
        this.surname = surname;
    }
}
// 子类继承父类
class Student entends Person {
    constructor(surname,firstname) {
        super(surname);                    //调用父类的constructor(surname)
        this.firstname = firstname;        //定义子类独有的属性
    }
}
```

注意：**子类在构造函数中使用super,必须放到this前面（必须先调用父类的构造方法，在使用子类构造方法）**

案例：

```
// 父类
class Father {
    constructor(surname){
        this.surname = surname;
```

```
    }
    saySurname() {
        console.log('我的姓是' + this.surname);
    }
}
// 子类继承父类
class Son entends Father {
    constructor(surname,firstname) {
        super(surname);                    //调用父类的
constructor(surname)
        this.firstname = firstname;        //定义子类独
有的属性
    }
    sayFirstname() {
        console.log('我的名字是:' + this.firstname);
    }
}

var damao = new Son('刘','德华');
damao.saySurname();
damao.sayFirstname();
1920212223
```

1.4.2、调用父类的普通函数

语法：

```
class Father {
    say() {
        return '我是爸爸';
    }
}
class Son extends Father {
    say(){
        // super.say() super调用父类的方法
        return super.say() + '的儿子';
    }
}

var damao = new Son();
console.log(damao.say());
14
```

- 多个方法之间不需要添加逗号分隔
- 继承中属性和方法的查找原则：就近原则，先看子类，再看父类

1.4、三个注意点

1. 在ES6中类没有变量提升，所以必须先定义类，才能通过类实例化对象
2. 类里面的共有属性和方法一定要加 `this` 使用
3. 类里面的

   ```
   this
   ```

   指向：

   - constructor 里面的 `this` 指向实例对象
   - 方法里面的 `this` 指向这个方法的调用者

```
<body>
    <button>点击</button>
    <script>
        var that;
        var _that;
        class Star {
            constructor(uname, age) {
                // constructor 里面的this 指向的是 创建的实例对象
                that = this;
                this.uname = uname;
                this.age = age;
                // this.sing();
                this.btn = document.querySelector('button');
                this.btn.onclick = this.sing;
            }
            sing() {
            // 这个sing方法里面的this 指向的是 btn 这个按钮,因为这个按钮调用了这个函数
                console.log(that.uname);
                // that里面存储的是constructor里面的this
            }
            dance() {
                // 这个dance里面的this 指向的是实例对象 ldh 因为ldh 调用了这个函数
                _that = this;
                console.log(this);
            }
        }
        var ldh = new Star('刘德华');
        console.log(that === ldh);
```

```
        ldh.dance();
        console.log(_that === ldh);

        // 1. 在 ES6 中类没有变量提升，所以必须先定义类，才能通过类实例化对象

        // 2. 类里面的共有的属性和方法一定要加this使用.
    </script>
</body>
19202122232425262728293031323334353 6
```

2、构造函数和原型

2.1、概述

在典型的 OOP 的语言中（如 Java），都存在类的概念，类就是对象的模板，对象就是类的实例，但在 ES6之前， JS 中并没用引入类的概念。

ES6， 全称 ECMAScript 6.0 ，2015.06 发版。但是目前浏览器的 JavaScript 是 ES5 版本，大多数高版本的浏览器也支持 ES6，不过只实现了 ES6 的部分特性和功能。

在 ES6之前 ，对象不是基于类创建的，而是用一种称为构建函数的特殊函数来定义对象和它们的特征。

- 创建对象有三种方式
    - **对象字面量**
    - **new Object()**
    - **自定义构造函数**

```
// 1. 利用 new Object() 创建对象
var obj1 = new Object();

// 2. 利用对象字面量创建对象
var obj2 = {};

// 3.利用构造函数创建对象
function Star(uname,age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
    }
}
var ldh = new Star('刘德华',18);
1415
```

注意：

1. 构造函数用于创建某一类对象，其首字母要大写
2. 构造函数要和 `new` 一起使用才有意义

2.2、构造函数

- 构造函数是一种特殊的函数，主要用来初始化对象(为对象成员变量赋初始值)，它总与 `new` 一起使用
- 我们可以把对象中的一些公共的属性和方法抽取出来，然后封装到这个函数里面

new 在执行时会做四件事

1. 在内存中创建一个新的空对象。
2. 让 this 指向这个新的对象。
3. 执行构造函数里面的代码，给这个新对象添加属性和方法。
4. 返回这个新对象（所以构造函数里面不需要 return ）。

2.2.1、静态成员和实例成员

JavaScript 的构造函数中可以添加一些成员，可以在构造函数本身上添加，也可以在构造函数内部的 `this` 上添加。通过这两种方式添加的成员，就分别称为静态成员和实例成员。

- 静态成员: 在构造函数本身上添加的成员为静态成员，只能由构造函数本身来访问
- 实例成员: 在构造函数内部创建的对象成员称为实例成员，只能由实例化的对象来访问

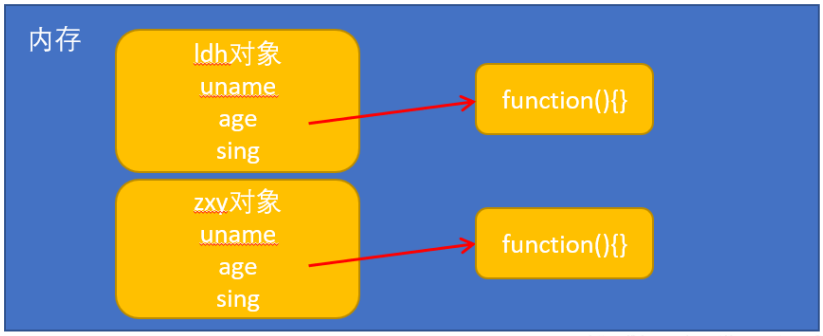
```
// 构造函数中的属性和方法我们称为成员，成员可以添加
function Star(uname,age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
    }
}
var ldh = new Star('刘德华',18);

// 实例成员就是构造函数内部通过this添加的成员  uname age sing  就是实例成员
// 实例成员只能通过实例化的对象来访问
ldh.sing();
Star.uname; // undefined     不可以通过构造函数来访问实例成员

// 静态成员就是在构造函数本身上添加的成员 sex 就是静态成员
// 静态成员只能通过构造函数来访问
```

```
Star.sex = '男';
Star.sex;
ldh.sex; // undefined  不能通过对象来访问
1920
```

2.2.2、构造函数的问题

构造函数方法很好用，但是存在浪费内存的问题。

```
function Star(uname, age) {
    this.uname = uname;
    this.age = age;
    this.sing = function() {
        console.log('我会唱歌');
}
}
var ldh = new Star('刘德华', 18);
var zxy = new Star('张学友', 19);
```



- **我们希望所有的对象使用同一个函数，这样就比较节省内存**

2.3、构造函数原型 prototype

- 构造函数通过原型分配的函数是所有对象所共享的,这样就解决了内存浪费问题
- JavaScript 规定，每一个构造函数都有一个 `prototype` 属性，指向另一个对象，注意这个 `prototype` 就是一个对象，这个对象的所有属性和方法，都会被构造函数所拥有
- 我们可以把那些不变的方法，直接定义在 `prototype` 对象上，这样所有对象的实例就可以共享这些方法

```
<body>
    <script>
        // 1. 构造函数的问题.
        function Star(uname, age) {
            //公共属性定义到构造函数里面
            this.uname = uname;
            this.age = age;
            // this.sing = function() {
            //     console.log('我会唱歌');
            // }
        }
        //公共的方法我们放到原型对象身上
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
        console.log(ldh.sing === zxy.sing);
```

```
        ldh.sing();
        zxy.sing();
        // 2. 一般情况下,我们的公共属性定义到构造函数里面, 公共的方法我们放到原型对象身上
    </script>
</body>
1920212223
```

- 一般情况下,我们的公共属性定义到构造函数里面, 公共的方法我们放到原型对象身上

问答：原型是什么?

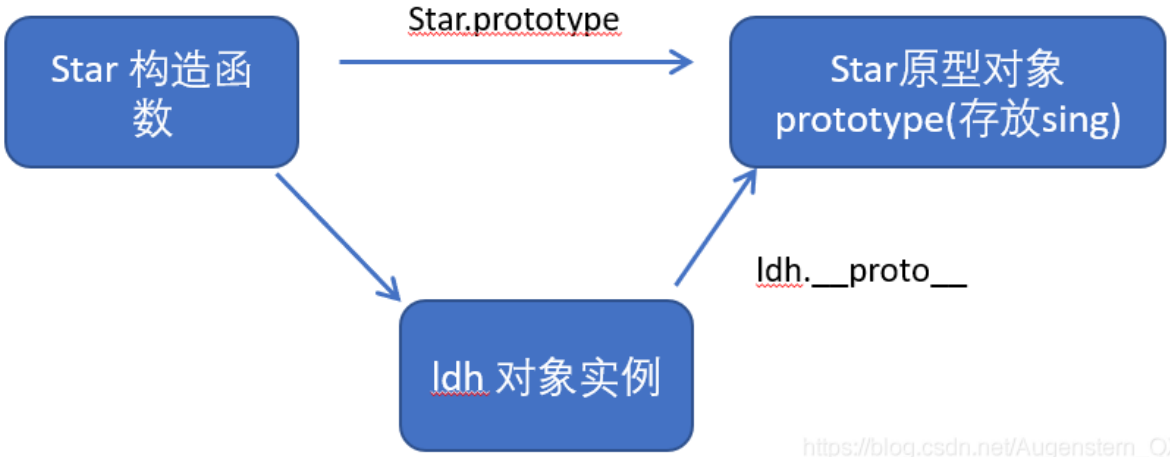
- 一个对象，我们也称为 `prototype` 为原型对象

问答：原型的作用是什么?

- 共享方法

2.4、对象原型 __ proto __

- 对象都会有一个属性 `_proto_` 指向构造函数的 `prototype` 原型对象，之所以我们对象可以使用构造函数 `prototype` 原型对象的属性和方法，就是因为对象有 `_proto_` 原型的存在。
- `_proto_` 对象原型和原型对象 `prototype` 是等价的
- `_proto_` 对象原型的意义就在于为对象的查找机制提供一个方向，或者说一条路线，但是它是一个非标准属性，因此实际开发中，不可以使用这个属性，它只是内部指向原型对象 `prototype`



- `Star.prototype 和 ldh._proto_` 指向相同

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
```

```
        }
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
        ldh.sing();
        console.log(ldh);
        // 对象身上系统自己添加一个 __proto__ 指向我们构造函数的原型对象 prototype
        console.log(ldh.__proto__ === Star.prototype);
        // 方法的查找规则: 首先先看ldh 对象身上是否有 sing 方法,如果有就执行这个对象上的sing
        // 如果没有sing 这个方法,因为有 __proto__ 的存在,就去构造函数原型对象prototype身上去查找sing这个方法
    </script>
</body>
19
```

2.5、constructor 构造函数

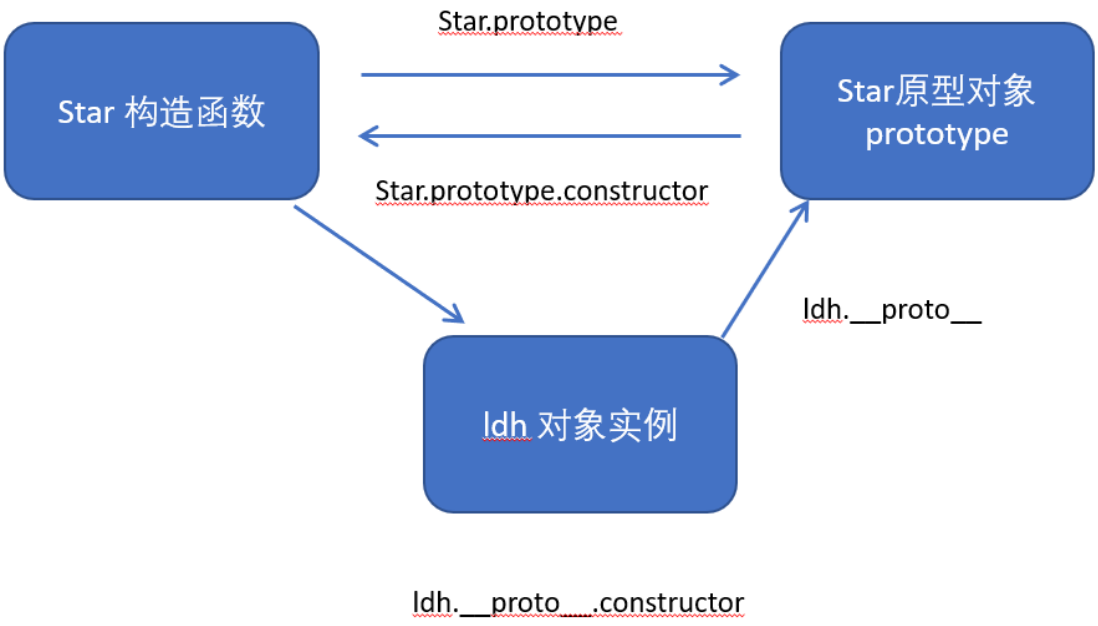
- 对象原型(__ proto __) 和构造函数(prototype)原型对象 里面都有一个属性 constructor 属性，constructor 我们称为构造函数，因为它指回构造函数本身。
- `constructor` 主要用于记录该对象引用于哪个构造函数，它可以让原型对象重新指向原来的构造函数
- **一般情况下，对象的方法都在构造函数(prototype)的原型对象中设置**
- 如果有多个对象的方法，我们可以给原型对象 `prototype` 采取对象形式赋值，但是这样会覆盖构造函数原型对象原来的内容，这样修改后的原型对象 `constructor` 就不再指向当前构造函数了。此时，我们可以在修改后的原型对象中，添加一个 `constructor` 指向原来的构造函数

具体请看实例配合理解

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        // 很多情况下,我们需要手动的利用constructor 这个属性指回 原来的构造函数
        // Star.prototype.sing = function() {
        //     console.log('我会唱歌');
        // };
```

```
        // Star.prototype.movie = function() {
        //     console.log('我会演电影');
        // }
        Star.prototype = {
            // 如果我们修改了原来的原型对象,给原型对象赋值的是一个对象,则必须手动的利用constructor指回原来的构造函数
            constructor: Star,
            sing: function() {
                console.log('我会唱歌');
            },
            movie: function() {
                console.log('我会演电影');
            }
        }
        var ldh = new Star('刘德华', 18);
        var zxy = new Star('张学友', 19);
    </script>
</body>
19202122232425262 7
```

2.6、构造函数、实例、原型对象三者关系

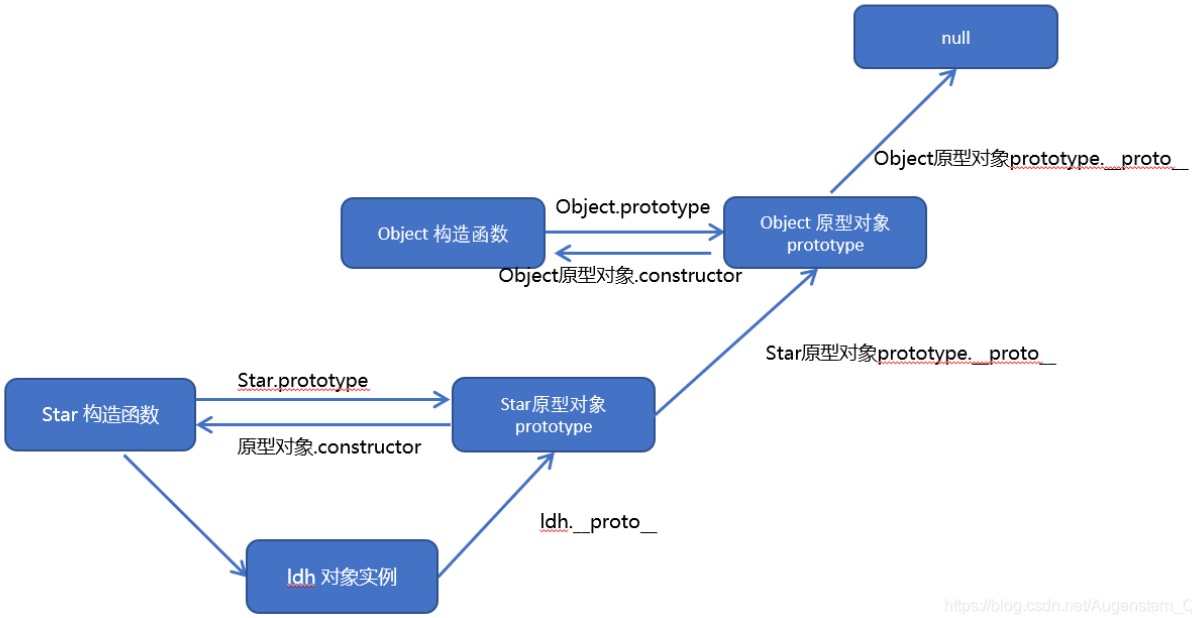


2.7、原型链查找规则

1. 当访问一个对象的属性(包括方法)时，首先查找这个对象自身有没有该属性
2. 如果没有就查找它的原型(也就是 `_proto_` 指向的 `prototype原型对象`)
3. 如果还没有就查找原型对象的原型(Object的原型对象)
4. 依次类推一直找到Object为止(null)

5. __ proto __对象原型的意义就在于为对象成员查找机制提供一个方向，或者说一条路线。

1.8 原型链




```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        Star.prototype.sing = function() {
            console.log('我会唱歌');
        }
        var ldh = new Star('刘德华', 18);
        // 1. 只要是对象就有__proto__ 原型, 指向原型对象
        console.log(Star.prototype);
        console.log(Star.prototype.__proto__ === Object.prototype);
        // 2.我们Star原型对象里面的__proto__原型指向的是 Object.prototype
        console.log(Object.prototype.__proto__);
        // 3. 我们Object.prototype原型对象里面的__proto__原型  指向为 null
    </script>
</body>
```

2.8、原型对象this指向

- 构造函数中的 `this` 指向我们的实例对象

- 原型对象里面放的是方法，这个方法里面的 `this` 指向的是这个方法的调用者，也就是这个实例对象

```
<body>
    <script>
        function Star(uname, age) {
            this.uname = uname;
            this.age = age;
        }
        var that;
        Star.prototype.sing = function() {
            console.log('我会唱歌');
            that = this;
        }
        var ldh = new Star('刘德华', 18);
        // 1. 在构造函数中,里面this指向的是对象实例 ldh
        ldh.sing();
        console.log(that === ldh);

        // 2.原型对象函数里面的this 指向的是 实例对象 ldh
    </script>
</body>
19
```

2.9、扩展内置对象

- 可以通过原型对象，对原来的内置对象进行扩展自定义的方法
- 比如给数组增加自定义求偶数和的功能

```
<body>
    <script>
        // 原型对象的应用 扩展内置对象方法

        Array.prototype.sum = function() {
            var sum = 0;
            for (var i = 0; i < this.length; i++) {
                sum += this[i];
            }
            return sum;
        };
        // Array.prototype = {
        //     sum: function() {
        //         var sum = 0;
        //         for (var i = 0; i < this.length; i++) {
```

```
        //             sum += this[i];
        //         }
        //         return sum;
        //     }

        // }
        var arr = [1, 2, 3];
        console.log(arr.sum());
        console.log(Array.prototype);
        var arr1 = new Array(11, 22, 33);
        console.log(arr1.sum());
    </script>
</body>
19202122232425262728
```

注意：

- 数组和字符串内置对象不能给原型对象覆盖操作 `Array.prototype = {}` ，只能是 `Array.prototype.xxx = function(){}` 的方式

3、继承

ES6 之前并没有给我们提供 `extends` 继承

- 我们可以通过构造函数+原型对象模拟实现继承，被称为组合继承

3.1、call()

调用这个函数，并且修改函数运行时的 this 指向

```
fun.call(thisArg,arg1,arg2,......)
1
```

- `thisArg` ： 当前调用函数 this 的指向对象
- `arg1,arg2` ： 传递的其他参数

示例

```
<body>
    <script>
        // call 方法
        function fn(x, y) {
            console.log('我希望我的希望有希望');
            console.log(this);      // Object{...}
            console.log(x + y);     // 3
        }

        var o = {
```

```
            name: 'andy'
        };
        // fn();
        // 1. call() 可以调用函数
        // fn.call();
        // 2. call() 可以改变这个函数的this指向 此时这个函数的this 就指向了o这个对象
        fn.call(o, 1, 2);
    </script>
</body>
19
```

3.2、借用构造函数继承父类型属性

- 核心原理: 通过 `call()` 把父类型的 this 指向子类型的 this，这样就可以实现子类型继承父类型的属性

```
<body>
    <script>
        // 借用父构造函数继承属性
        // 1. 父构造函数
        function Father(uname, age) {
            // this 指向父构造函数的对象实例
            this.uname = uname;
            this.age = age;
        }
        // 2 .子构造函数
        function Son(uname, age, score) {
            // this 指向子构造函数的对象实例
            Father.call(this, uname, age);
            this.score = score;
        }
        var son = new Son('刘德华', 18, 100);
        console.log(son);
    </script>
</body>
19
```

3.3、借用原型对象继承父类型方法

- 一般情况下，对象的方法都在构造函数的原型对象中设置，通过构造函数无法继承父类方法

核心原理：

1. 将子类所共享的方法提取出来，让子类的 `prototype 原型对象 = new 父类()`

2. 本质： 子类原型对象等于是实例化父类，因为父类实例化之后另外开辟空间，就不会影响原来父类原型对象
3. 将子类的 `constructor` 重新指向子类的构造函数

```
<body>
    <script>
        // 借用父构造函数继承属性
        // 1. 父构造函数
        function Father(uname, age) {
            // this 指向父构造函数的对象实例
            this.uname = uname;
            this.age = age;
        }
        Father.prototype.money = function() {
            console.log(100000);

        };
        // 2 .子构造函数
        function Son(uname, age, score) {
            // this 指向子构造函数的对象实例
            Father.call(this, uname, age);
            this.score = score;
        }
        // Son.prototype = Father.prototype;  这样直接赋值会有问题,如果修改了子原型对象,父原型对象也会跟着一起变化
        Son.prototype = new Father();
        // 如果利用对象的形式修改了原型对象,别忘了利用 constructor 指回原来的构造函数
        Son.prototype.constructor = Son;
        // 这个是子构造函数专门的方法
        Son.prototype.exam = function() {
            console.log('孩子要考试');

        }
        var son = new Son('刘德华', 18, 100);
        console.log(son);
        console.log(Father.prototype);
        console.log(Son.prototype.constructor);
    </script>
</body>
19202122232425262728293031323334
```

3.3 类的本质

1. class 本质还是 function
2. 类的所有方法都定义在类的 `prototype` 属性上
3. 类创建的实例，里面也有 `_proto_` 指向类的 `prototype` 原型对象
4. 所以 ES6 的类它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。
5. 所以 ES6 的类其实就是语法糖
6. 语法糖：语法糖就是一种便捷写法，简单理解

4、ES5新增方法

ES5 给我们新增了一些方法，可以很方便的操作数组或者字符串

- 数组方法
- 字符串方法
- 对象方法

4.1、数组方法

- 迭代(遍历)方法：foreach() ，map()，filter()，some() ，every() ;

4.1.1、forEach()

```
array.forEach(function(currentValue,index,arr))
1
```

- currentValue : 数组当前项的值
- index: 数组当前项的索引
- arr: 数组对象本身

```
<body>
    <script>
        // forEach 迭代(遍历) 数组
        var arr = [1, 2, 3];
        var sum = 0;
        arr.forEach(function(value, index, array) {
            console.log('每个数组元素' + value);
            console.log('每个数组元素的索引号' +
index);
            console.log('数组本身' + array);
            sum += value;
        })
        console.log(sum);
```

```
    </script>
</body>
14
```

4.1.2、filter()筛选数组

```
array.filter(function(currentValue,index,arr))
1
```

- `filter()` 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素，主要用于筛选数组
- 注意它直接返回一个新数组

```
<body>
    <script>
        // filter 筛选数组
        var arr = [12, 66, 4, 88, 3, 7];
        var newArr = arr.filter(function(value,
index) {
            // return value >= 20;
            return value % 2 === 0;
        });
        console.log(newArr);
    </script>
</body>
11
```

4.1.3、some()

- `some()` 方法用于检测数组中的元素是否满足指定条件（查找数组中是否有满足条件的元素）
- 注意它返回的是布尔值，如果查找到这个元素，就返回true，如果查找不到就返回false
- 如果找到第一个满足条件的元素，则终止循环，不再继续查找

```
<body>
    <script>
        // some 查找数组中是否有满足条件的元素
        var arr1 = ['red', 'pink', 'blue'];
        var flag1 = arr1.some(function(value) {
            return value == 'pink';
        });
        console.log(flag1);
        // 1. filter 也是查找满足条件的元素 返回的是一个数组 而且是把所有满足条件的元素返回回来
        // 2. some 也是查找满足条件的元素是否存在  返回的是一个布尔值 如果查找到第一个满足条件的元素就终止循环
    </script>
</body>
1112
```

4.2、字符串方法

- `trim()` 方法会从一个字符串的两端删除空白字符
- `trim()` 方法并不影响原字符串本身，它返回的是一个新的字符串

```
<body>
    <input type="text"> <button>点击</button>
    <div></div>
    <script>
        // trim 方法去除字符串两侧空格
        var str = '   an  dy   ';
        console.log(str);
        var str1 = str.trim();
        console.log(str1);
        var input = document.querySelector('input');
        var btn = document.querySelector('button');
        var div = document.querySelector('div');
        btn.onclick = function() {
            var str = input.value.trim();
            if (str === '') {
                alert('请输入内容');
            } else {
                console.log(str);
                console.log(str.length);
                div.innerHTML = str;
            }
        }
    </script>
</body>
192021222324
```

4.3、对象方法

4.3.1、Object.keys()

1. `Object.keys()` 用于获取对象自身所有的属性
2. 效果类似 `for...in`
3. 返回一个由属性名组成的数组

```
<body>
    <script>
        // 用于获取对象自身所有的属性
        var obj = {
            id: 1,
            pname: '小米',
            price: 1999,
            num: 2000
        };
        var arr = Object.keys(obj);
        console.log(arr);
        arr.forEach(function(value) {
            console.log(value);
            // id
            // pname
            // price
            // num
        })
    </script>
</body>
1920
```

4.3.2、Object.defineProperty()

- `Object.defineProperty()` 定义对象中新属性或修改原有的属性(了解)

```
Object.defineProperty(obj,prop,descriptor)
1
```

- obj : 目标对象
- prop : 需定义或修改的属性的名字
- descriptor : 目标属性所拥有的特性

```
<body>
    <script>
        // Object.defineProperty() 定义新属性或修改原有的属性
        var obj = {
```

```
            id: 1,
            pname: '小米',
            price: 1999
        };
        // 1. 以前的对象添加和修改属性的方式
        // obj.num = 1000;
        // obj.price = 99;
        // console.log(obj);
        // 2. Object.defineProperty() 定义新属性或修改原有的属性
        Object.defineProperty(obj, 'num', {
            value: 1000,
            enumerable: true
        });
        console.log(obj);
        Object.defineProperty(obj, 'price', {
            value: 9.9
        });
        console.log(obj);
        Object.defineProperty(obj, 'id', {
            // 如果值为false 不允许修改这个属性值 默认值也是false
            writable: false,
        });
        obj.id = 2;
        console.log(obj);
        Object.defineProperty(obj, 'address', {
            value: '中国山东蓝翔技校xx单元',
            // 如果只为false 不允许修改这个属性值 默认值也是false
            writable: false,
            // enumerable 如果值为false 则不允许遍历, 默认的值是 false
            enumerable: false,
            // configurable 如果为false 则不允许删除这个属性 不允许在修改第三个参数里面的特性 默认为false
            configurable: false
        });
        console.log(obj);
        console.log(Object.keys(obj));
        delete obj.address;
        console.log(obj);
        delete obj.pname;
        console.log(obj);
        Object.defineProperty(obj, 'address', {
            value: '中国山东蓝翔技校xx单元',
            // 如果值为false 不允许修改这个属性值 默认值也是false
```

```
            writable: true,
            // enumerable 如果值为false 则不允许遍历，
默认的值是 false
            enumerable: true,
            // configurable 如果为false 则不允许删除这
个属性 默认为false
            configurable: true
        });
        console.log(obj.address);
    </script>
</body>
474849505152535455
```

- 第三个参数 descriptor 说明：以对象形式{ }书写
- value：设置属性的值，默认为undefined
- writeable: 值是否可以重写 true | false 默认为false
- enumerable: 目标属性是否可以被枚举 true | false 默认为false
- configurable: 目标属性是否可以被删除或是否可以再次修改特性 true | false 默认为false

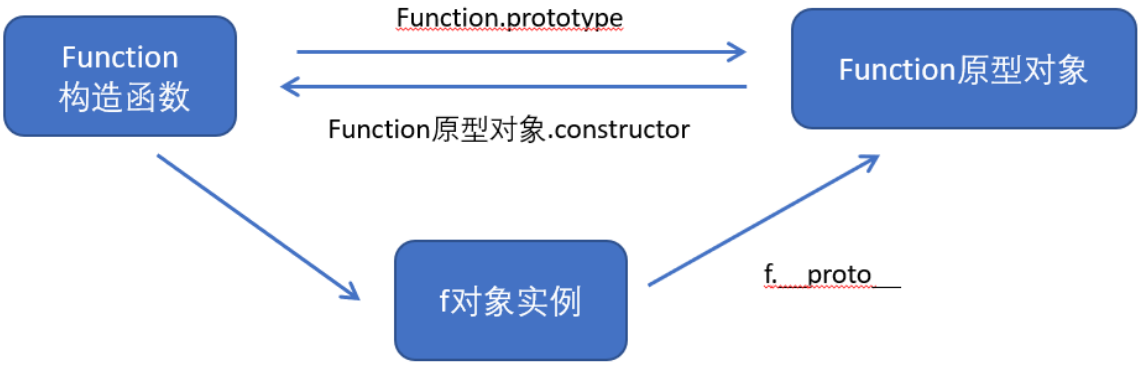
5、函数进阶

5.1、函数的定义方式

1. 函数声明方式 function 关键字(命名函数)
2. 函数表达式(匿名函数)
3. new Function()

```
var fn = new Function('参数1','参数2',.....,'函数
体');
1
```

- Function 里面参数都必须是字符串格式
- 第三种方式执行效率低，也不方便书写，因此较少使用
- 所有函数都是 Function 的实例(对象)
- 函数也属于对象




```
<body>
    <script>
        //  函数的定义方式

        // 1. 自定义函数(命名函数)

        function fn() {};

        // 2. 函数表达式 (匿名函数)

        var fun = function() {};


        // 3. 利用 new Function('参数1','参数2', '函数体');
        //             Function 里面参数都必须是字符串格式，执行效率低，较少写

        var f = new Function('a', 'b', 'console.log(a + b)');
        f(1, 2);
        // 4. 所有函数都是 Function 的实例(对象)
        console.dir(f);
        // 5. 函数也属于对象
        console.log(f instanceof Object);
    </script>
</body>
192021222324
```

5.2、函数的调用方式

1. 普通函数
2. 对象的方法
3. 构造函数

4. 绑定事件函数
5. 定时器函数
6. 立即执行函数

```
<body>
    <script>
        // 函数的调用方式

        // 1. 普通函数
        function fn() {
            console.log('人生的巅峰');

        }
        // fn();   fn.call()
        // 2. 对象的方法
        var o = {
            sayHi: function() {
                console.log('人生的巅峰');

            }
        }
        o.sayHi();
        // 3. 构造函数
        function Star() {};
        new Star();
        // 4. 绑定事件函数
        // btn.onclick = function() {};   // 点击了
按钮就可以调用这个函数
        // 5. 定时器函数
        // setInterval(function() {}, 1000);  这个函
数是定时器自动1秒钟调用一次
        // 6. 立即执行函数
        (function() {
            console.log('人生的巅峰');
        })();
        // 立即执行函数是自动调用
    </script>
</body>
19202122232425262728293031 32
```

5.3、函数内this的指向

- `this` 指向，是当我们调用函数的时候确定的，调用方式的不同决定了 `this` 的指向不同，一般我们指向我们的调用者

调用方式	this指向
普通函数调用	window
构造函数调用	实例对象，原型对象里面的方法也指向实例对象
对象方法调用	该方法所属对象
事件绑定方法	绑定事件对象
定时器函数	window
立即执行函数	window

```
<body>
    <button>点击</button>
    <script>
        // 函数的不同调用方式决定了this 的指向不同
        // 1. 普通函数 this 指向window
        function fn() {
            console.log('普通函数的this' + this);
        }
        window.fn();
        // 2. 对象的方法 this指向的是对象 o
        var o = {
            sayHi: function() {
                console.log('对象方法的this:' +
this);
            }
        }
        o.sayHi();
        // 3. 构造函数 this 指向 ldh 这个实例对象 原型
对象里面的this 指向的也是 ldh这个实例对象
        function Star() {};
        Star.prototype.sing = function() {

        }
        var ldh = new Star();
        // 4. 绑定事件函数 this 指向的是函数的调用者
btn这个按钮对象
        var btn = document.querySelector('button');
        btn.onclick = function() {
            console.log('绑定时间函数的this:' +
this);
        };
        // 5. 定时器函数 this 指向的也是window
        window.setTimeout(function() {
```

```
            console.log('定时器的this:' + this);

        }, 1000);
        // 6. 立即执行函数 this还是指向window
        (function() {
            console.log('立即执行函数的this' + this);
        })();
    </script>
</body>
19202122232425262728293031323334353637 38
```

5.4、改变函数内部this指向

- JavaScript 为我们专门提供了一些函数方法来帮我们处理函数内部 this 的指向问题，常用的有 `bind(),call(),apply()` 三种方法

5.4.1、call() 方法

- `call()` 方法调用一个对象，简单理解为调用函数的方式，但是它可以改变函数的 `this` 指向
- `fun.call(thisArg,arg1,arg2,.....)`
- `thisArg` : 在 fun 函数运行时指定的 this 值
- `arg1,arg2` : 传递的其他参数
- 返回值就是函数的返回值，因为它就是调用函数
- 因此当我们想改变 this 指向，同时想调用这个函数的时候，可以使用 call，比如继承

```
<body>
    <script>
        // 改变函数内this指向  js提供了三种方法
call()  apply()  bind()

        // 1. call()
        var o = {
            name: 'andy'
        }

        function fn(a, b) {
            console.log(this);
            console.log(a + b);

        };
        fn.call(o, 1, 2);
        // call 第一个可以调用函数 第二个可以改变函数内
的this 指向
        // call 的主要作用可以实现继承
        function Father(uname, age, sex) {
```

```
            this.uname = uname;
            this.age = age;
            this.sex = sex;
        }

        function Son(uname, age, sex) {
            Father.call(this, uname, age, sex);
        }
        var son = new Son('刘德华', 18, '男');
        console.log(son);
    </script>
</body>
19202122232425262728293O
```

5.4.2、apply()方法

- `apply()` 方法调用一个函数，简单理解为调用函数的方式，但是它可以改变函数的 `this` 指向
- `fun.apply(thisArg,[argsArray])`
- thisArg: 在 fun 函数运行时指定的 this 值
- argsArray : 传递的值，必须包含在数组里面
- 返回值就是函数的返回值，因为它就是调用函数
- 因此 apply 主要跟数组有关系，比如使用 Math.max() 求数组的最大值

```
<body>
    <script>
        // 改变函数内this指向  js提供了三种方法
call()  apply()  bind()

        // 2. apply()  应用 运用的意思
        var o = {
            name: 'andy'
        };

        function fn(arr) {
            console.log(this);
            console.log(arr); // 'pink'

        };
        fn.apply(o, ['pink']);
        // 1. 也是调用函数 第二个可以改变函数内部的this
指向
        // 2. 但是他的参数必须是数组(伪数组)
        // 3. apply 的主要应用 比如说我们可以利用
apply 借助于数学内置对象求数组最大值
        // Math.max();
```

```
        var arr = [1, 66, 3, 99, 4];
        var arr1 = ['red', 'pink'];
        // var max = Math.max.apply(null, arr);
        var max = Math.max.apply(Math, arr);
        var min = Math.min.apply(Math, arr);
        console.log(max, min);
    </script>
</body>
1920212223242526 27
```

5.4.3、bind()方法

- `bind()` 方法不会调用函数。但是能改变函数内部 `this` 指向
- `fun.bind(thisArg,arg1,arg2,....)`
- 返回由指定的 `this` 值和初始化参数改造的 原函数拷贝
- 因此当我们只是想改变 this 指向，并且不想调用这个函数的时候，可以使用bind

```
<body>
    <button>点击</button>
    <button>点击</button>
    <button>点击</button>
    <script>
        // 改变函数内this指向  js提供了三种方法
call()  apply()  bind()

        // 3. bind()  绑定 捆绑的意思
        var o = {
            name: 'andy'
        };

        function fn(a, b) {
            console.log(this);
            console.log(a + b);


        };
        var f = fn.bind(o, 1, 2);
        f();
        // 1. 不会调用原来的函数    可以改变原来函数内部
的this 指向
        // 2. 返回的是原函数改变this之后产生的新函数
        // 3. 如果有的函数我们不需要立即调用,但是又想改
变这个函数内部的this指向此时用bind
        // 4. 我们有一个按钮,当我们点击了之后,就禁用这
个按钮,3秒钟之后开启这个按钮
```

```
        // var btn1 =
document.querySelector('button');
        // btn1.onclick = function() {
        //     this.disabled = true; // 这个this 指
向的是 btn 这个按钮
        //     // var that = this;
        //     setTimeout(function() {
        //         // that.disabled = false; // 定时
器函数里面的this 指向的是window
        //         this.disabled = false; // 此时定
时器函数里面的this 指向的是btn
        //     }.bind(this), 3000); // 这个this 指向
的是btn 这个对象
        // }
        var btns =
document.querySelectorAll('button');
        for (var i = 0; i < btns.length; i++) {
            btns[i].onclick = function() {
                this.disabled = true;
                setTimeout(function() {
                    this.disabled = false;
                }.bind(this), 2000);
            }
        }
    </script>
</body>
1920212223242526272829303132333435363738394041424344
```

5.4.4、总结

call apply bind 总结：

相同点：

- 都可以改变函数内部的 `this` 指向

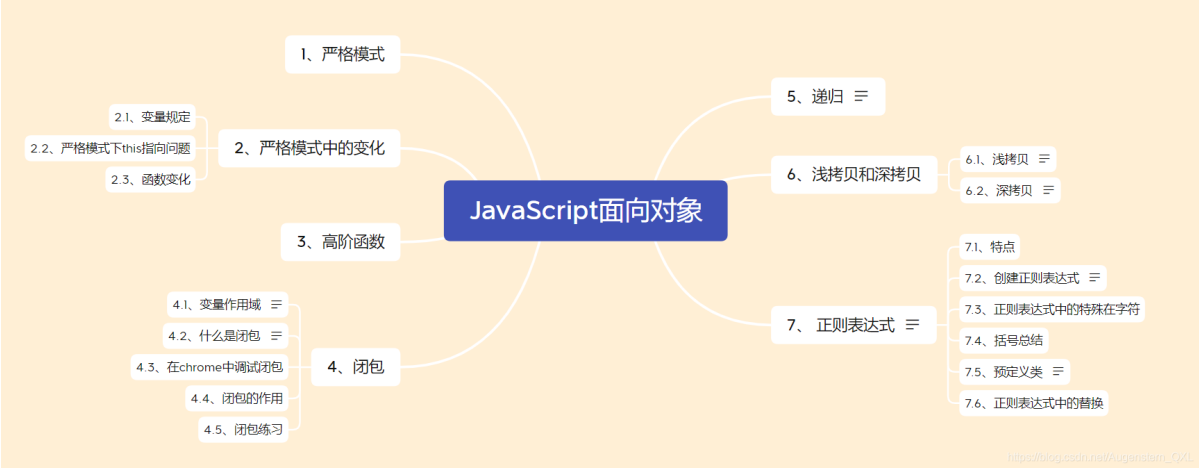
区别点：

- `call` 和 `apply` 会调用函数，并且改变函数内部的 `this` 指向
- `call` 和 `apply` 传递的参数不一样，call 传递参数，apply 必须数组形式
- `bind` 不会调用函数，可以改变函数内部 `this` 指向

主要应用场景

1. `call` 经常做继承
2. `apply` 经常跟数组有关系，比如借助于数学对线实现数组最大值与最小值
3. `bind` 不调用函数，但是还想改变this指向，比如改变定时器内部的this指向

✍目录总览



1、严格模式

- JavaScript 除了提供正常模式外，还提供了严格模式
- ES5 的严格模式是采用具有限制性 JavaScript 变体的一种方式，即在严格的条件下运行 JS 代码
- 严格模式在IE10 以上版本的浏览器才会被支持，旧版本浏览器会被忽略
- 严格模式对正常的JavaScript语义做了一些更改：
  - 消除了Javascript 语法的一些不合理、不严谨之处，减少了一些怪异行为
  - 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
  - 禁用了在 ECMAScript 的未来版本中可能会定义的一些语法，为未来新版本的 Javascript 做好铺垫。比如一些保留字如：class, enum, export, extends, import, super 不能做变量名

1.1、开启严格模式

- 严格模式可以应用到整个脚本或个别函数中。
- 因此在使用时，我们可以将严格模式分为为脚本开启严格模式和为函数开启严格模式两种情况

1.1.2、为脚本开启严格模式

- 为整个脚本文件开启严格模式，需要在所有语句之前放一个特定语句
- `"use strict"` 或 `'use strict'`

```
<script>
    'user strict';
    console.log("这是严格模式。");
</script>
1234
```

因为 `"use strict"` 加了引号，所以老版本的浏览器会把它当作一行普通字符串而忽略。

有的 script 基本是严格模式，有的 script 脚本是正常模式，这样不利于文件合并，所以可以将整个脚本文件放在一个立即执行的匿名函数之中。这样独立创建一个作用域而不影响其他 script 脚本文件。

```
<script>
    (function (){
        'use strict';
         var num = 10;
         function fn() {}
    })();
</script>
67
```

1.1.2、为函数开启严格模式

- 若要给某个函数开启严格模式，需要把 `"use strict"` 或 `'use strict'` 声明放在函数体所有语句之前

```
<body>
    <!-- 为整个脚本(script标签)开启严格模式 -->
    <script>
        'use strict';
        //   下面的js 代码就会按照严格模式执行代码
    </script>
    <script>
        (function() {
            'use strict';
        })();
    </script>
    <!-- 为某个函数开启严格模式 -->
    <script>
        // 此时只是给fn函数开启严格模式
        function fn() {
            'use strict';
            // 下面的代码按照严格模式执行
        }

        function fun() {
```

```
            // 里面的还是按照普通模式执行
        }
    </script>
</body>
192021222324
```

- 将 `"use strict"` 放在函数体的第一行，则整个函数以 "严格模式"运行。

2、严格模式中的变化

- 严格模式对JavaScript的语法和行为，都做了一些改变

2.1、变量规定

- 在正常模式中，如果一个变量没有声明就赋值，默认是全局变量
- 严格模式禁止这种用法，变量都必须先用var 命令声明，然后再使用
- 严禁删除已经声明变量，例如，`` `delete x `` 语法是错误的

```
<body>
    <script>
        'use strict';
        // 1. 我们的变量名必须先声明再使用
        // num = 10;
        // console.log(num);
        var num = 10;
        console.log(num);
        // 2.我们不能随意删除已经声明好的变量
        // delete num;
    </script>
</body>
1112
```

2.2、严格模式下this指向问题

1. 以前在全局作用域函数中的 `this` 指向 `window` 对象
2. 严格模式下全局作用域中函数中的 `this` 是 undefined
3. 以前构造函数时不加 `new` 也可以调用，当普通函数， `this` 指向全局对象
4. 严格模式下，如果构造函数不加 `new` 调用， `this` 指向的是 `undefined` ，如果给它赋值，会报错
5. `new` 实例化的构造函数指向创建的对象实例
6. 定时器 `this` 还是指向 `window`
7. 事件、对象还是指向调用者

```
<body>
```

```
    <script>
        'use strict';
        //3. 严格模式下全局作用域中函数中的 this 是
undefined。
        function fn() {
            console.log(this); // undefined。

        }
        fn();
        //4. 严格模式下,如果 构造函数不加new调用,
this 指向的是undefined 如果给他赋值则 会报错.
        function Star() {
            this.sex = '男';
        }
        // Star();
        var ldh = new Star();
        console.log(ldh.sex);
        //5. 定时器 this 还是指向 window
        setTimeout(function() {
            console.log(this);

        }, 2000);

    </script>
</body>
192021222324
```

2.3、函数变化

1. 函数不能有重名的**参数**
2. 函数必须声明在顶层，新版本的JavaScript会引入“块级作用域”（ES6中已引入）。为了与新版本接轨，**不允许在非函数的代码块内声明函数**

```
<body>
    <script>
        'use strict';
        // 6. 严格模式下函数里面的参数不允许有重名
        function fn(a, a) {
           console.log(a + a);

        };
        // fn(1, 2);
        function fn() {}
    </script>
</body>
1112
```

3、高阶函数

- 高阶函数是对其他函数进行操作的函数，它接收函数作为参数或将函数作为返回值输出

接收函数作为参数

```
<body>
    <div></div>
    <script>
        // 高阶函数- 函数可以作为参数传递
        function fn(a, b, callback) {
            console.log(a + b);
            callback && callback();
        }
        fn(1, 2, function() {
            console.log('我是最后调用的');

        });

    </script>
</body>
1415
```

将函数作为返回值

```
<script>
    function fn(){
        return function() {}
    }
</script>
```

- 此时 fn 就是一个高阶函数
- 函数也是一种数据类型，同样可以作为参数，传递给另外一个参数使用。最典型的就是作为回调函数
- 同理函数也可以作为返回值传递回来

4、闭包

4.1、变量作用域

变量根据作用域的不同分为两种：全局变量和局部变量

1. 函数内部可以使用全局变量
2. 函数外部不可以使用局部变量
3. 当函数执行完毕，本作用域内的局部变量会销毁。

4.2、什么是闭包

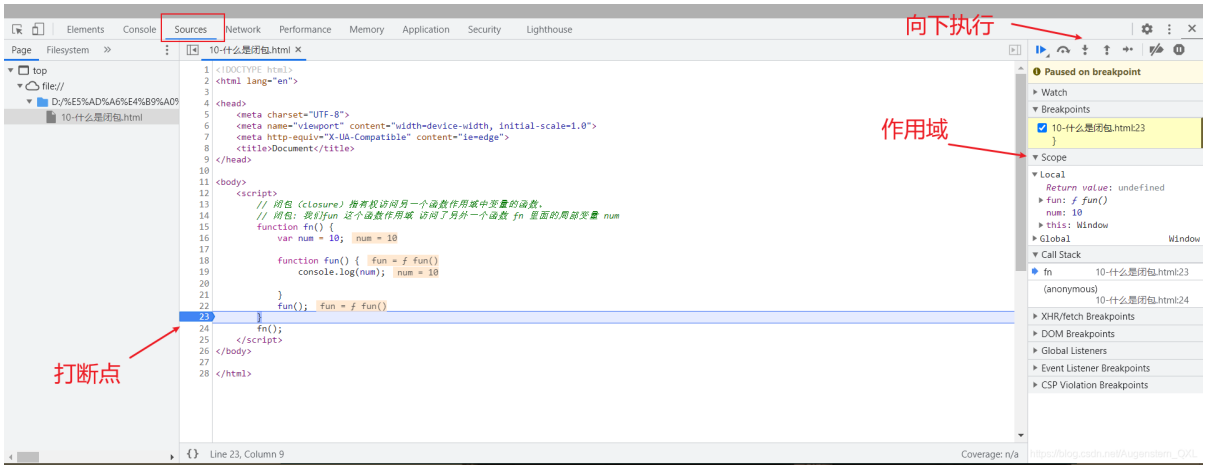
闭包指有权访问另一个函数作用域中的变量的函数

简单理解：一个作用域可以访问另外一个函数内部的局部变量

```
<body>
    <script>
        // 闭包（closure）指有权访问另一个函数作用域中变量的函数。
        // 闭包：我们fn2 这个函数作用域 访问了另外一个函数 fn1 里面的局部变量 num
        function fn1() {          // fn1就是闭包函数
            var num = 10;
            function fn2() {
                console.log(num);    //10
            }
            fn2();
        }
        fn1();
    </script>
</body>
14
```

4.3、在chrome中调试闭包

1. 打开浏览器，按 F12 键启动 chrome 调试工具。
2. 设置断点。
3. 找到 Scope 选项（Scope 作用域的意思）。
4. 当我们重新刷新页面，会进入断点调试，Scope 里面会有两个参数（global 全局作用域、local 局部作用域）。
5. 当执行到 fn2() 时，Scope 里面会多一个 Closure 参数 ，这就表明产生了闭包。



4.4、闭包的作用

- 延伸变量的作用范围

```
<body>
    <script>
        // 闭包（closure）指有权访问另一个函数作用域
中变量的函数。
        // 一个作用域可以访问另外一个函数的局部变量
        // 我们fn 外面的作用域可以访问fn 内部的局部
变量
        // 闭包的主要作用: 延伸了变量的作用范围
        function fn() {
            var num = 10;
            return function() {
                console.log(num);
            }
        }
        var f = fn();
        f();
    </script>
</body>
141516
```

4.5、闭包练习

4.5.1、点击li输出索引号

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>臭豆腐</li>
        <li>鲱鱼罐头</li>
        <li>大猪蹄子</li>
    </ul>
    <script>
        // 闭包应用-点击li输出当前li的索引号
        // 1. 我们可以利用动态添加属性的方式
        var lis =
document.querySelector('.nav').querySelectorAll('
li');
        for (var i = 0; i < lis.length; i++) {
            lis[i].index = i;
            lis[i].onclick = function() {
                // console.log(i);
                console.log(this.index);
```

```
            }
        }
        // 2. 利用闭包的方式得到当前小li 的索引号
        for (var i = 0; i < lis.length; i++) {
            // 利用for循环创建了4个立即执行函数
            // 立即执行函数也成为小闭包因为立即执行函数里面的任何一个函数都可以使用它的i这变量
            (function(i) {
                // console.log(i);
                lis[i].onclick = function() {
                    console.log(i);

                }
            })(i);
        }
    </script>
</body>
1920212223242526272829303132 33
```

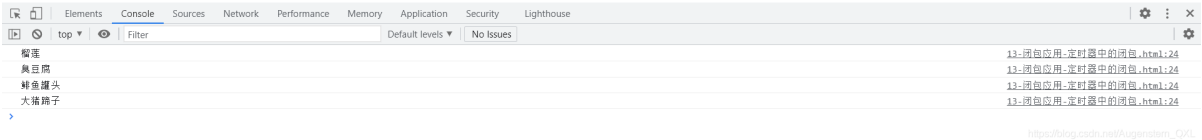
- 榴莲
- 臭豆腐
- 鲱鱼罐头
- 大猪蹄子

4.5.2、定时器中的闭包

```
<body>
    <ul class="nav">
        <li>榴莲</li>
        <li>臭豆腐</li>
        <li>鲱鱼罐头</li>
        <li>大猪蹄子</li>
    </ul>
    <script>
        // 闭包应用-3秒钟之后,打印所有li元素的内容
        var lis = document.querySelector('.nav').querySelectorAll('li');
        for (var i = 0; i < lis.length; i++) {
            (function(i) {
                setTimeout(function() {

 console.log(lis[i].innerHTML);
                }, 3000)
            })(i);
        }
```

```
    </script>
</body>
19
```

- 榴莲
- 臭豆腐
- 鲱鱼罐头
- 大猪蹄子



5、递归

如果一个函数在内部可以调用其本身，那么这个函数就是递归函数

简单理解： 函数内部自己调用自己，这个函数就是递归函数

由于递归很容易发生"栈溢出"错误，所以必须要加退出条件 return

```
<body>
    <script>
        // 递归函数 : 函数内部自己调用自己, 这个函数就是递归函数
        var num = 1;

        function fn() {
            console.log('我要打印6句话');

            if (num == 6) {
                return; // 递归里面必须加退出条件
            }
            num++;
            fn();
        }
        fn();
    </script>
</body>
```

6、浅拷贝和深拷贝

1. 浅拷贝只是拷贝一层，更深层次对象级别的只拷贝引用
2. 深拷贝拷贝多层，每一级别的数据都会拷贝
3. `Object.assign(target,....sources)` ES6新增方法可以浅拷贝

6.1、浅拷贝

```
// 浅拷贝只是拷贝一层，更深层次对象级别的只拷贝引用
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
};
var o = {}
for(var k in obj){
    // k是属性名，obj[k]是属性值
    o[k] = obj.[k];
}
console.log(o);
// 浅拷贝语法糖
Object.assign(o,obj);
141516
```

6.2、深拷贝

```
// 深拷贝拷贝多层，每一级别的数据都会拷贝
var obj = {
    id: 1,
    name: 'andy',
    msg: {
        age: 18
    }
    color: ['pink','red']
};
var o = {};
// 封装函数
function deepCopy(newobj,oldobj){
    for(var k in oldobj){
        // 判断属性值属于简单数据类型还是复杂数据类型
        // 1.获取属性值   oldobj[k]
        var item = obldobj[k];
        // 2.判断这个值是否是数组
        if(item instanceof Array){
            newobj[k] = [];
            deepCopy(newobj[k],item)
        }else if (item instanceof Object){
                // 3.判断这个值是否是对象
            newobj[k] = {};
            deepCopy(newobj[k],item)
```

```
        }else {
            // 4.属于简单数据类型
            newobj[k] = item;

        }
    }
}
deepCopy(o,obj);
1920212223242526272829303132
```

7、 正则表达式

正则表达式是用于匹配字符串中字符组合的模式。在JavaScript中，正则表达式也是对象。

正则表通常被用来检索、替换那些符合某个模式（规则）的文本，例如验证表单：用户名表单只能输入英文字母、数字或者下划线， 昵称输入框中可以输入中文(匹配)。此外，正则表达式还常用于过滤掉页面内容中的一些敏感词(替换)，或从字符串中获取我们想要的特定部分(提取)等 。

7.1、特点

- 实际开发，一般都是直接复制写好的正则表达式
- 但是要求会使用正则表达式并且根据自身实际情况修改正则表达式

7.2、创建正则表达式

在JavaScript中，可以通过两种方式创建正则表达式

1. 通过调用 RegExp 对象的构造函数创建
2. 通过字面量创建

7.2.1、通过调用 RegExp 对象的构造函数创建

通过调用 RegExp 对象的构造函数创建

```
var 变量名 = new RegExp(/表达式/);
1
```

7.2.2、通过字面量创建

通过字面量创建

```
var 变量名 = /表达式/;
1
```

注释中间放表达式就是正则字面量

7.2.3、测试正则表达式 test

- `test()` 正则对象方法，用于检测字符串是否符合该规则，该对象会返回 `true` 或 `false` ,其参数是测试字符串

```
regexObj.test(str)
1
```

- `regexObj` 写的是正则表达式
- `str` 我们要测试的文本
- 就是检测 `str` 文本是否符合我们写的正则表达式规范

示例

```
<body>
    <script>
        // 正则表达式在js中的使用

        // 1. 利用 RegExp对象来创建 正则表达式
        var regexp = new RegExp(/123/);
        console.log(regexp);

        // 2. 利用字面量创建 正则表达式
        var rg = /123/;
        // 3.test 方法用来检测字符串是否符合正则表达式要求的规范
        console.log(rg.test(123));
        console.log(rg.test('abc'));
    </script>
</body>
1415
```

7.3、正则表达式中的特殊在字符

7.3.1、边界符

正则表达式中的边界符(位置符)用来提示字符所处的位置，主要有两个字符

边界符	说明
^	表示匹配行首的文本(以谁开始)
$	表示匹配行尾的文本(以谁结束)

如果^ 和 $ 在一起，表示必须是精确匹配

```
// 边界符 ^ $
var rg = /abc/;    //正则表达式里面不需要加引号，不管是数字型还是字符串型
// /abc/只要包含有abc这个字符串返回的都是true
console.log(rg.test('abc'));
console.log(rg.test('abcd'));
console.log(rg.test('aabcd'));

var reg = /^abc/;
console.log(reg.test('abc'));    //true
console.log(reg.test('abcd'));  // true
console.log(reg.test('aabcd')); // false

var reg1 = /^abc$/
// 以abc开头，以abc结尾，必须是abc
14
```

7.3.2、字符类

- 字符类表示有一系列字符可供选择，只要匹配其中一个就可以了
- 所有可供选择的字符都放在方括号内

①[] 方括号

```
/[abc]/.test('andy');      // true
1
```

后面的字符串只要包含 `abc` 中任意一个字符,都返回 `true`

②[-]方括号内部 范围符

```
/^[a-z]$/.test()
1
```

方括号内部加上 - 表示范围，这里表示 a - z 26个英文字母都可以

③[^] 方括号内部 取反符 ^

```
/[^abc]/.test('andy')    // false
1
```

方括号内部加上 ^ 表示取反，只要包含方括号内的字符，都返回 `false`

注意和边界符 ^ 区别，边界符写到方括号外面

④字符组合

```
/[a-z1-9]/.test('andy')    // true
1
```

方括号内部可以使用字符组合，这里表示包含 a 到 z的26个英文字母和1到9的数字都可以

```
<body>
    <script>
        //var rg = /abc/;  只要包含abc就可以
        // 字符类: [] 表示有一系列字符可供选择，只要匹配其中一个就可以了
        var rg = /[abc]/; // 只要包含有a 或者 包含有b 或者包含有c 都返回为true
        console.log(rg.test('andy'));
        console.log(rg.test('baby'));
        console.log(rg.test('color'));
        console.log(rg.test('red'));
        var rg1 = /^[abc]$/; // 三选一 只有是a 或者是 b  或者是c 这三个字母才返回 true
        console.log(rg1.test('aa'));
        console.log(rg1.test('a'));
        console.log(rg1.test('b'));
        console.log(rg1.test('c'));
        console.log(rg1.test('abc'));
        console.log('-------------------');

        var reg = /^[a-z]$/; // 26个英文字母任何一个字母返回 true  - 表示的是a 到z 的范围
        console.log(reg.test('a'));
        console.log(reg.test('z'));
        console.log(reg.test(1));
        console.log(reg.test('A'));
        // 字符组合
        var reg1 = /^[a-zA-Z0-9_-]$/; // 26个英文字母(大写和小写都可以)任何一个字母返回 true
        console.log(reg1.test('a'));
        console.log(reg1.test('B'));
        console.log(reg1.test(8));
        console.log(reg1.test('-'));
        console.log(reg1.test('_'));
        console.log(reg1.test('!'));
        console.log('-----------------');
        // 如果中括号里面有^ 表示取反的意思 千万和 我们边界符 ^ 别混淆
        var reg2 = /^[^a-zA-Z0-9_-]$/;
        console.log(reg2.test('a'));
```

```
        console.log(reg2.test('B'));
        console.log(reg2.test(8));
        console.log(reg2.test('-'));
        console.log(reg2.test('_'));
        console.log(reg2.test('!'));
    </script>
</body>
```

```
192021222324252627282930313233343536373839404 1
```

7.3.3、量词符

量词符用来设定某个模式出现的次数

量词	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复n次
{n,}	重复n次或更多次
{n,m}	重复n到m次

```
<body>
    <script>
        // 量词符: 用来设定某个模式出现的次数
        // 简单理解: 就是让下面的a这个字符重复多少次
        // var reg = /^a$/;


        //  * 相当于 >= 0 可以出现0次或者很多次
        // var reg = /^a*$/;
        // console.log(reg.test(''));
        // console.log(reg.test('a'));
        // console.log(reg.test('aaaa'));



        //  + 相当于 >= 1 可以出现1次或者很多次
        // var reg = /^a+$/;
        // console.log(reg.test('')); // false
        // console.log(reg.test('a')); // true
        // console.log(reg.test('aaaa')); // true
```

```
        //  ?  相当于 1 || 0
        // var reg = /^a?$/;
        // console.log(reg.test('')); // true
        // console.log(reg.test('a')); // true
        // console.log(reg.test('aaaa')); //
false

        //  {3 } 就是重复3次
        // var reg = /^a{3}$/;
        // console.log(reg.test('')); // false
        // console.log(reg.test('a')); // false
        // console.log(reg.test('aaaa')); //
false
        // console.log(reg.test('aaa')); // true
        //  {3, }  大于等于3
        var reg = /^a{3,}$/;
        console.log(reg.test('')); // false
        console.log(reg.test('a')); // false
        console.log(reg.test('aaaa')); // true
        console.log(reg.test('aaa')); // true
        //  {3,16}  大于等于3 并且 小于等于16
        var reg = /^a{3,6}$/;
        console.log(reg.test('')); // false
        console.log(reg.test('a')); // false
        console.log(reg.test('aaaa')); // true
        console.log(reg.test('aaa')); // true
        console.log(reg.test('aaaaaaa')); //
false
    </script>
</body>
4748
```

7.3.4、用户名验证

功能需求：

1. 如果用户名输入合法, 则后面提示信息为 : 用户名合法,并且颜色为绿色
2. 如果用户名输入不合法, 则后面提示信息为: 用户名不符合规范, 并且颜色为绿色

分析：

1. 用户名只能为英文字母,数字,下划线或者短横线组成, 并且用户名长度为 6~16位.
2. 首先准备好这种正则表达式模式 /$[a-zA-Z0-9-_]{6,16}^/
3. 当表单失去焦点就开始验证.
4. 如果符合正则规范, 则让后面的span标签添加 right 类.
5. 如果不符合正则规范, 则让后面的span标签添加 wrong 类.

```
<body>
    <input type="text" class="uname"> <span>请输入用户名</span>
    <script>
        //  量词是设定某个模式出现的次数
        var reg = /^[a-zA-Z0-9_-]{6,16}$/; // 这个模式用户只能输入英文字母 数字 下划线 短横线但是有边界符和[] 这就限定了只能多选1
        // {6,16}  中间不要有空格
        // console.log(reg.test('a'));
        // console.log(reg.test('8'));
        // console.log(reg.test('18'));
        // console.log(reg.test('aa'));
        // console.log('--------------');
        // console.log(reg.test('andy-red'));
        // console.log(reg.test('andy_red'));
        // console.log(reg.test('andy007'));
        // console.log(reg.test('andy!007'));
        var uname = document.querySelector('.uname');
        var span = document.querySelector('span');
        uname.onblur = function() {
            if (reg.test(this.value)) {
                console.log('正确的');
                span.className = 'right';
                span.innerHTML = '用户名格式输入正确';
            } else {
                console.log('错误的');
                span.className = 'wrong';
                span.innerHTML = '用户名格式输入不正确';
            }
        }
    </script>
</body>
19202122232425262728293O
```

7.4、括号总结

1. 大括号 量词符 里面面表示重复次数
2. 中括号 字符集合 匹配方括号中的任意字符
3. 小括号 表示优先级

```
// 中括号 字符集合 匹配方括号中的任意字符
var reg = /^[abc]$/;
// a || b || c
// 大括号 量词符 里面表示重复次数
var reg = /^abc{3}$/;    // 它只是让c 重复3次 abccc
// 小括号 表示优先级
var reg = /^(abc){3}$/;  //它是让 abc 重复3次
67
```

在线测试正则表达式：https://c.runoob.com/

7.5、预定义类

预定义类指的是 某些常见模式的简写写法

预定类	说明
\d	匹配0-9之间的任一数字，相当于[0-9]
\D	匹配所有0-9以外的字符，相当于[^ 0-9]
\w	匹配任意的字母、数字和下划线,相当于[A-Za-z0-9_]
\W	除所有字母、数字、和下划线以外的字符，相当于[^A-Za-z0-9_]
\s	匹配空格（包括换行符，制表符，空格符等），相当于[\t\t\n\v\f]
\S	匹配非空格的字符，相当于[^ \t\r\n\v\f]

7.5.1、表单验证

分析：

1.手机号码: `/^1[3|4|5|7|8][0-9]{9}$/`

2.QQ: `[1-9][0-9]{4,}` (腾讯QQ号从10000开始)

3.昵称是中文: `^[\u4e00-\u9fa5]{2,8}$`

```
<body>
    <script>
        // 座机号码验证:  全国座机号码  两种格式:
010-  或者  0530-67
        // 正则里面的或者 符号  |
        // var reg = /^\d{3}-\d{8}|\d{4}-\d{7}$/;
        var reg = /^\d{3,4}-\d{7,8}$/;
    </script>
</body>
```

7.6、正则表达式中的替换

7.6.1、replace 替换

`replace()` 方法可以实现替换字符串操作，用来替换的参数可以是一个字符串或是一个正则表达式

```
stringObject.replace(regexp/substr,replacement)
1
```

1. 第一个参数: 被替换的字符串或者正则表达式
2. 第二个参数：替换为的字符串
3. 返回值是一个替换完毕的新字符串

```
// 替换 replace
var str = 'andy和red';
var newStr = str.replace('andy','baby');
var newStr = str.replace(/andy/,'baby');
1234
```

7.6.2、正则表达式参数

```
/表达式/[switch]
1
```

`switch` 按照什么样的模式来匹配，有三种

- `g` : 全局匹配
- `i` :忽略大小写
- `gi` : 全局匹配 + 忽略大小写