JavaScript

——有用请投 44 号，谢谢（部分有错误，请辩证看待，自行更正）

web 发展史

Mosaic，是互联网历史上第一个获普遍使用和能够显示图片的网页浏览器。于 1993 年问世。

1994 年 4 月，马克.安德森和 Silicon Graphics（简称为 SGI，中译为“视算科技”或“硅图”）公司的创始人吉姆·克拉克（Jim Clark）在美国加州设立了“Mosaic Communication Corporation”。

Mosaic 公司成立后，由于伊利诺伊大学拥有 Mosaic 的商标权，且伊利诺伊大学已将技术转让给 Spy Glass 公司，开发团队必须彻底重新撰写浏览器程式码，且浏览器名称更改为 Netscape Navigator，公司名字于 1994 年 11 月改名为“Netscape Communication Corporation”，此后沿用至今，中译为“网景”。

微软的 Internet Explorer 及 Mozilla Firefox 等，其早期版本皆以 Mosaic 为基础而开发。微软随后买下 Spy Glass 公司的技术开发出 Internet Explorer 浏览器，而 Mozilla Firefox 则是网景通讯家开放源代码后所衍生出的版本。

js 历史

JavaScript 作为 Netscape Navigator 浏览器的一部分首次出现在 1996 年。它最初的设计目标是改善网页的用户体验。

作者：Brendan Eich

期初 JavaScript 被命名为 **LiveScript**，后因和 Sun 公司合作，因市场宣传需要改名 **JavaScript**。后来 Sun 公司被 Oracle 收购，JavaScript 版权归 Oracle 所有。

浏览器组成

1.shell 部分——用户能操作部分(壳)

2.内核部分——用户看不到的部分

1)渲染引擎（语法规则和渲染）

2)js 引擎

3)其他模块（如异步）

js 引擎

2001 年发布 ie6，首次实现对 js 引擎的优化。

2008 年 Google 发布最新浏览器 Chrome，它是采用优化后的 javascript 引擎，引擎代号 V8，因能把 js 代码直接转化为机械码来执行，进而以速度快而闻名。

后 Firefox 也推出了具备强大功能的 js 引擎

Firefox3.5　TraceMonkey（对频繁执行的代码做了路径优化）

Firefox4.0　JeagerMonkey

js 的逼格（特有特色）

	编译型语言	解释性语言
怎么做	通篇翻译后，生成翻译完的文件，程序执行翻译后的文件	看一行翻译一行，不生成特定文件
代表语言	C,C++	JS,PHP，python 带尖角号
优点	快(常用于系统，游戏)	可以跨平台
缺点	移植性不好（不跨平台，window 和 Linux 不能混用）	稍微慢点

js 是解释性语言：(不需要编译成文件）跨平台

java 先通过 javac，编译成.class 文件，通过 jvm（Java 虚拟机）进行解释执行

.java→javac→编译→.class→jvm→解释执行（java 可以跨平台）（java 是 oak 语言）

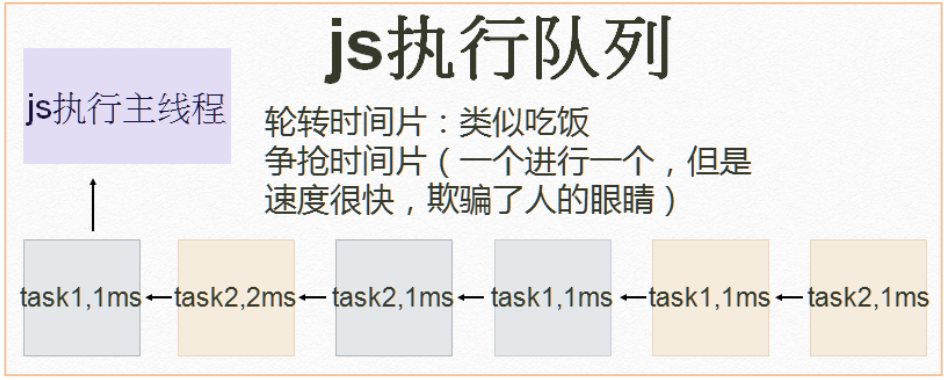
<link rel = “” >是异步加载

单线程：同一时间只能做一件事——js 引擎是单线程

（同一时间做很多事叫多线程）

ECMA（欧洲计算机制造联合会）标注：为了取得技术优势，微软推出了 JScript，CEnvi 推出 ScriptEase，与 JavaScript 同样可在浏览器上运行。为了统一规格 JavaScript 兼容于 ECMA 标准，因此也称为 ECMAScript。

js 是轮转时间片



主流浏览器（必须有独立内核）市场份额大于 3%	内核名称
IE	trident
chrome	webkit/blink
firefox	gecko
opera	presto
safari	webkit

开始学习 js

js 三大部分 ECMAScript、DOM、BOM

如何引入 js?

1、页面内嵌<script></script>标签,写 head 里面也行,写 body 里面也行

例<body>

```
    <script type="text/javascript">        //告诉浏览器我们是 js
    </script>
</body>
```

2、外部 js 文件，引入<script src="location"></script>

例如：以 lesson.js 保存文件，再引入到 html 中

为符合 web 标准（w3c 标准中的一项）结构（html）、行为（js）、样式（css）相分离，通常会采用外部引入。

一个文件中可以包括多个 css，js——不混用

特殊写页面，大部分写在外部——不混用

如果同时写了内部的 js 和外部的 js，那么是外部的 js 文件显示出来

js 基本语法

1、变量(variable)

HTML，css 不是编程语言，是计算机语言，编程语言需要有变量和函数

变量是存放东西，方便后续使用的框

1）变量声明

1.声明、赋值分解

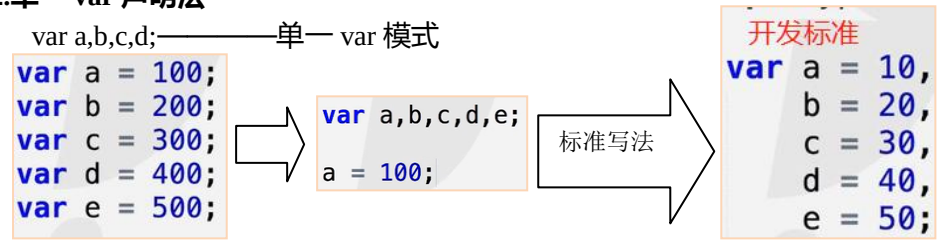
var a; 这个叫变量声明。我们向系统中申请了 var 这个框，命名叫 a

给 a 赋值 100，写作 a =100，这里不是等号是赋值

var a ；a =100；可以简化写成 var a=100；

2.单一 var 声明法

var a,b,c,d;————单一 var 模式



如写做：var a = 10；a=20;那么后面的 20 就会覆盖掉前面的 10

2）命名规则（用接近的英文单词）———— 起变量名一定要以英文语义化

1.变量名必须以英文字母、_、$ 开头

2.变量名可以包括英文字母、_、$、数字

3.**不可以**用系统的关键字、保留字作为变量名

关键字（有特殊语法含义的字）			
break	else	new	var
case	finally	return	void
catch	for	switch	while
default	if	throw	
delete	in	try	
do	instanceof	typeof	
保留字（未来可能当做关键字的词）			
abstract	enum	int	short
Boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronize
class	float	package	throws
const	goto	private	transient
debugger	implements	pritected	volatile
double	import	public	

基本语法

下面是变量,例：var a = 10;

```
var b = 20;
var c;
c = a + b;
```

先运算等号右边的 a+b，运算完后，再赋值给左边 c

先取值，再赋值

运算大于赋值的优先级

js 是动态语言，动态语言基本上都是解释性语言，解释性语言基本上都是脚本语言

js 是浮点型语言（带小数点）

值类型（数据类型）

1、不可改变的原始值（栈数据）栈 stack

Number,String,Boolean,undefined,null

已经放进去的值不可改变，只会改房间编号为 null(硬盘原理)

Number 数字,例 var a = 123;

String 字符串,语言放双引号里，例 var a=" 语言"，""是空串

Boolean 布尔数字,就两个值，false，true

undefined 是没有定义的，表示还没赋值,仅一个值 underfined

null 代表空，占位用，用空值来覆盖

例 var a =10;
var b = a;
a = 20;
document.write(b);
答案：10
原始值是我把一个值放到另一个值里面，改了第一个值，第二个值不变
2、引用值（堆数据）大致上放堆 heap 里面
array 数组, Object, function ... data,RegExp 正则
var arr = [1,2,3,4,5,false," abc"]; //这是数组
例：var arr = [1];
var arr1 = arr;
arr.push(2);
document.write(arr1);
答案：arr 是 1,2。arr1 是 1,2
引用值是把第一个值放到第二个值里面，改第一个值，第二个值也改变
js 由值决定类型。原始值和引用值唯一的不同是赋值形式不同

栈 stack 先进去的东西最后出来，有底没有顶。 在栈内存里面放东西，先放在底部		堆 heap 怎么放怎么拿，散列结构	
1005 旧 a 变成 null、 a arr	 10 heap1001	1001	[1,2] 在 arr.push(3) 就变成了[1,2,3] [1,2]
(旧 1004)b arr1	10 heap1001	1002	[1,3]
1003 新 a	 20	1003	
1002		1004	
序号倒序 1001		1005	

var a = 10;var b =a；是 a 先取出 10，copy 一份放到 b 里面，改变 a 的值，b 的值是不变的，再把 a=20;时 b 的值还是 10，不发生改变
var arr = [1,2];var arr1 =arr;arr.push(3);
答案：这往[1,2]放 3，arr 和 arr1 都是[1,2,3]
引用值是在栈内存里面放堆的地址，拷贝的也是地址，所以改变 arr，arr1 也变了
var arr = [1,2]; var arr1 =arr; arr = [1,3]; document.write(arr1)
答案：arr = [1,3];是新建了一个新的房间。arr1 是 1,2,现在是插入新引入值" 房间"，会在堆里面重新申请一间房，并指向新房间

js 语句基本规则
1、语句后面要用分号结束“；”但 function test(){}，for(){}，if(){}后面都不用加分号
2、js 语法错误会引发后续代码终止，但不会影响其它 js 代码块
错误分为两种
1）低级错误（语法解析错误），不能写中文
2）逻辑错误（标准错误，情有可原，错的那个执行不了）
3、书写格式要规范，“= + / -” 两边都应该有空格

js 运算符

一、运算操作符

+

1. “+”作用：数学运算、字符串链接
2.任何数据类型加字符串都等于字符串
例 var a = “a” + true + 1; //打印 atrue1
例 var a = 1 + “a” + 1 + 1; //打印 1a11
例 var a = 1 + 1 + “a” + 1 + 1; //打印 2a11，从左向右运算
例 var a = 1 + 1 + “a” +(1 + 2); //打印 2a3

\- * /

例 var a = 0 – 1； //等于-1
例 var a = 2 * 1； //等于 2
例 var a = 0 / 0； //答案是 NaN ,应该得出一个数字类型的数 ,但是没法表达，就用 NaN (NaN 是 Not a Number 非数，不是数，但是是数字类型
例 var a = 1 / 0; //是 infinity
例 var a = -1 / 0; /是-infinity

% ，摩尔，模，是取余数的意思
例 var a =5%2 //5%2 是五除二的余数，商二余一
例 var a =5%1 //是五除一的余数，结果是 0
例 var num = 1 % 5; //意思是 1 除以 5 的余数。商 0 余 1
例 var a =4%6 //是四除六的余数，结果是 4
例 var a = 4;a % = 5;document.write(a); // 4
例 var a = 0;a % = 5;document.write(a); //0
例 var a = 10;a %= 2;document.write(a); //0
例 var a = 3;a % = 4; //4
“-”，“*”，“/ “，“%”，” = “，“()”
优先级”=“最弱【赋值符号优先级最低】，”()”优先级较高

“++”，“- -”，” += “，“-=”，“/= “，“*=”，“%=”

++

例 var a = 10; a = a + 1; //结果 11

例 var a = 1;

a = a + 1;写成 a ++是一种简化形式“++”，是自身加一，再赋值给自身

a++是 a=a+1 的简化形式

例 var a =10；document.write(++a);document.write(a); //答案 11；11

是先执行++，再执行本条语句 document.write(++a)

例 var a =1；document.write(a ++);document.write(a); //答案 1；2。是先执行语句(document.write(a))，再++，所以第一次打印的还是 a，第二次打印 a++后的值

例 var a =10；var b=++a -1+a++;document.write(b + “ ” + a) //答案 21 12

先++a，这个时候 a=11，再-1，再加 a，b 就是 21，最后++，a 就是 12

赋值的顺序自右向左，计算的顺序自左向右（按数学来）

例 var a =1；var b = a ++ + 1；document.write(b)； //答案 2，先执行 var b =a+1，再 a++

例 var a =1；var b = a ++ + 1；document.write(a)；document.write(b)； //答案 2，2

例 var a =1；var b = ++a + 1；document.write(a)；document.write(b)； //答案 2，3

例 var i = 1;var a = i++; //答案 a = 1; 此时 i 先将值 1 赋给 a,然后自己+1，i=2;

var b = ++i; //答案 b = 3；此时 i 先自己+1 为 3.再给 b 赋值，b=3;

--

“- -”，是自身减一，在赋值给自身

例 var a = 1;var b = a-- + -- a;document.write(b); //答案 0 ,先执行--a;此时 a 变成 0，然后第一个 a 也变成 0，那么 b = 0-- + --a

例 var a = 1;var b = --a + --a;document.write(b); //答案-1

例 var a = 1;document.write(a++);document.write(a); //答案 1；2

例 var a = 1;document.write(++a);document.write(a); //答案 2；2

例 var a =1; var b = a ++ +1;document.write(b); //答案 2

a 写在后面就后运行，先计算 a+1=2 赋值给 b 后再++

例 var a = 1;var b= ++a + 1;document.write(a);document.write(b); //答案 2；3

+= -=

例 var a =10;a ++;a ++;a ++;加十个

简化写法：a +=10;也是 a = a+10;

例 var a =10;a += 10 + 1; //答案 21

例 var a = 1;a = a + 10;等于 a+=10

a++是 a +=1 的写法

/=

例 var a=10;a/=2; //答案 5，是除二赋给自身的意思

*=

例 var a =10;a *=2; //答案：20，是乘二赋给自身的意思

%=

例 var a=10;a%=2; //答案：0， 10 能整除 2,余数是 0,取余，余数赋给自身。

例 var a=3;a%=4; //答案：3，3 除以 4，余数为 3，余数赋给自身。

例 var a=0;a%=4; //答案：0，0 除以 4，余数为 0，余数赋给自身。

例 var a = 1;a% =10; //答案：1，1 除以 10，余数为 1，余数赋给自身。

作业：

1、写出打印结果

var a =(10 * 3 – 4 / 2 + 1)%2,b=3;b %= a + 3;

document.write(a++);

document.wtite(“
”);

document.write(--b);

```
var a = (10 * 3 - 4 / 2 + 1) % 2,
    b = 3;
b %= a + 3;
document.write(a++);
document.write("<br>");
document.write(--b);
```

2、var a =123; var b =234;经过计算交换 a，b 的值

作业答案

1、document.write(a++);是 1（先打出 1 再++）

document.write(--b);是 2

b % =a + 3,3 的模等于 4，除不开再赋值给 b，b 还是 3

2、方法一普通方法：var c = a; a = b; b = c;document.write(a ,b);

方法二 a = a + b; b = a – b; a = a –b; document.write(a ,b);

二、比较运算符

1、“>”，” <”，” ==”，“>=”，“<=”，” !=” 比较结果为 boolean 值

但凡是运算符，都是要有运算的

用到**布尔值，true 或 false**

字符串的比较，比的是 ASCII 码（七位二进制 0000000）

>， <

例 var a = "a">"b";document.write(a); //答案是 false

例 var a = 1 > 2;document.write(a); //答案是 false

例 var a = 1 < 2;document.write(a); //答案是 true

例 var a = "1">"8";document.write(a); //答案是 false

例 var a = "10">"8";document.write(a); //答案 false，不是十和八比，是字符串一零和八比，先用开头的一和八比，比不过就不看第二位了；一样的就拿零和八比

例 var a = 123;document.write(a); //答案 false

运算结果为真实的值

= =，等不等于

例 var a = 1 == 2; //答案是说 1 等不等于 2，因为 1 肯定不等于 2，所以值为 false

例 var a = NaN == NaN; //答案是 false，NaN 不等于任何东西，包括他自己

例 var a = undefined == underfined; //答案是 true
例 var a = infinity == infinity; //答案是 true
例 var a = NaN == NaN; //答案是 false。非数 NaN 是不等于自己的
NaN 得不出数，又是数字类型，就是 NaN

>=，<=

!=是否不等于，非等

比较结果为 boolean 值：true 和 false

三、逻辑运算符：“&&”，“||”，“!“运算结果为真实的值

“&&”与运算符

两个表达式：先看第一个表达式转换成布尔值的结果是否为真，如果结果为真，那么它会看第二个表达式转换为布尔值的结果，然后如果只有两个表达式的话，只看看第二个表达式，就可以返回该表达式的值了，如果第一位布尔值为 false，不看后面的，返回第一个表达式的值就可以了

运算符就是要求结果

例 var a = 1 && 2; //答案 2，如果第一位 1 为真，结果就为第二位的值 2
例 var a = 1 && 2 + 2; //答案 4，如果 1 为真，结果就为 4
例 var a = 0 && 2 + 2; //答案 0
例 var a = 1 && 1 && 8; //答案 8，先看第一个是否为真，为真再看第二个，中途如果遇到 false，那就返回 false 的值
例 var a =1 + 1 && 1 – 1;document.write(a); //答案 0

如果是三个或多个表达式，会先看第一个表达式是否为真，如果为真，就看第二个表达式，如果第二个也为真，就看第三个表达式（如果为真就往后看，一旦遇到假就返回到假的值），如果第三个是最后一个表达式，那就直接返回第三个的结果

如果第一个是假，就返回第一个值，当是真的时候就往后走，一旦遇到假，就返回

例：2>1 && document.write(‘成哥很帅’) //意思是如果 2 大于 1，那么就打印成哥很帅，如果前面真才能执行后面的（相当于短路语句使用）

&&与运算符是有中断作用的，当短路语句使用(如果。。那么。。)

例 var data = ...; //执行一个语句，会用到 data
data&&执行一个语句全用到 data
例 data && function(data);

&与运算 我们一般不用

例 var num = 1 & 2;document.write(num); //答案 0
例 var num = 1 & 1;document.write(num); //答案 1
例 var num = 1 & 3;document.write(num); //答案 1

上下一与，不同为 0，相同为 1		
1 在二进制中，是 1（为了对齐补的 0）	0	1
3 在二进制中，是 11	1	1
运算结果	0	1

“||”或运算符

例 var num = 1 || 3; //答案 1
例 var num = 0 || 3; //答案 3
例 var num = 0 || false; //答案是 false

看第一个表达式是否为真，如果为真，则返回第一个值，碰到真就返回

如果第一个表达式是假，就看第二个表达式，如果第二个是最后一个，就返回第二个的值

关注真假的说法：全假才为假，有一个真就为真

例 var num = 0 || false || 1;document.write(num); //答案 1
例 div .onclick = function(e){
非 IE 浏览器直接取 e 值
var event = e;
IE 浏览器存在 window.event;
}

```
div.onclick = function (e) {
    非IE浏览器

    var event = e;
    IE window.event;

}
```

写成下面这样就解决了兼容性。在所有的浏览器中都好使
div .onclick = function(e){var event = e || window.event;}

“!“非运算符，否的意思。

先转成布尔值，再取反

例 var a = ! 123;document.write(a); //答案 false。123 的布尔值是 true，取反是 false
例 var a = ! “” ;document.write(a); //答案 true。空串“”布尔值是 false，取反是 true
例 var a = ! ! “” ;document.write(a); //答案 false，取反后，再反过来，结果不变
例 var a = true;a =!a;document.write(a) //答案 false，自身取反，再赋值给自身
！=非等于是表达他们到底等不等的

四、被认定为 false 的值 转换为布尔值会被认定为 false 的值 undefined，null，NaN，“”（空串），0， false

条件语句

一、If 语句 if、if else if if <—> && 转换

if(条件判断){
当条件成立时，执行里面的执行语句
}
当 if 的条件成立时，才能执行{}内的语句
当条件转化为布尔值，如果为 true 就执行；如果为 false 就不执行
例 if(1 < 2){document.write(“老邓很丑”);}
例 if (1 > 0 && 8 > 9){}

&&放在 if 中的，全真才为真，&&是并且的意思

例 if (1 > 0 || 8 > 9){}

||放在 if 中是或者的意思，有一个是真就可以了

例

```
//90 - 100   alibaba
//80 - 90    tencent toutiao meituan 滴滴
//70 - 80    baidu eleme xiecheng 58赶集
//60 - 70    蘑菇街
//60 以下     你肯定不是我教的!!!
```

```
<script type="text/javascript">
    var score = parseInt(window.prompt('input'));
```

```
if(score > 90 && score <= 100) {
    document.write('alibaba');
}

if(score > 80 && score <= 90) {
    document.write('tencent');
}

if(score > 70 && score <= 80) {
    document.write('baidu');
}

if(score > 60 && score <= 70) {
    document.write('mogujie');
}

if(score < 60) {
    document.write('Oh my god!!! ');
}
```

//不能写 90<score<100,这样写会先比 90<score,等于 true 以后再跟 100 比

//else if 满足了第一条就不看第二条了，用 else if 要满足条件与条件之间互斥，不能有交叉点。除了上面所有以外的。else if 除了这个以外，满足第一个就不看了，不满足就看后面的

else if 除了这以外在看这个满不满足。满足条件后就不看了

下面的写法不够简洁

else 是上面这个条件的补集

简洁写法见右边

```
if(score > 90 && score <= 100) {
    document.write('alibaba');
}else if(score > 80 && score <= 90) {
    document.write('tencent');
}else if(score > 70 && score <= 80) {
    document.write('baidu');
}else if(score > 60 && score <= 70) {
    document.write('mogujie');
}else if(score < 60) {
    document.write('Oh my god!!! you g
}else{
    document.write('error');
}
```

IF 和&&的互相转化

```
if (1 > 2) {
        document.write.('a');
  }
```

上面与右边效果完全一样 1 > 2 && document.write('a');

&&和 II 比较常用在条件判断中

二、for 循环(for 循环不固定，非常灵活)

格式 for (var i = 0; i < 10; i++) {

}

for 是关键字，( ) 括号里面三个语句用两个分号隔开，{}里面是循环体

打印十个 a，写成：

```
for(var i = 0; i < 10; i++) {
    document.write('a');
}
```

```
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
document.write('a');
```

执行顺序如下：

(1)var i= 0;

(2)if(i <10){

document.write('a')

}

把条件判断放到 if 里面,条件判断成立，就执行{}中间的执行体

(3)i++

看 i++，此时 i 变成 1，i=1；

(4)if(i <10){

document.write('a')

}

把条件判断放到 if 里面,条件判断成立，就执行{}中间的执行体

(5)i++

看 i++，此时 i 变成 1，i=1；

————————如此反复

先执行一遍(1)，

判断(2)执行语句成不成立，条件成立就执行(3),

判断(2)执行语句成不成立，条件成立就执行(3),

.....当有一次判断不成立，就停止

因为看的是执行顺序，写外面也可以，执行顺序是一样，打印十个 a，也可以写成：

```
var i = 0;
for(; i < 10; ) {

    document.write('a');
    i++;
}
```

打印十个 a，也可以写成：

```
var i = 1;
var count = 0;
for(; i; ) {

    document.write('a');
    count ++;
    if(count == 10) {
        i = 0;
    }
}
```

```
var i = 1;

for(; i; ) {
    document.write('a');
    i++;
    if(i == 11) {
        i = 0;
    }
}
```

例打印 0-9

```
var i = 1;

for(var i = 0; i < 10; i++) {
    document.write(i);
}
```

求 0-9 的和：

```
var i = 1;
var count = 0;
for(var i = 0; i < 10; i++) {
    count += i;
}
```

例打印 100 以内能被 3 整除，或者能被 5 整除，或者能被 7 整除的数：

```
var i = 1;
var count = 0;
for(var i = 0; i < 100; i++) {
    if(i % 3 == 0|| i % 5 == 0 || i % 7 == 0) {
        document.write(i + " ");
    }
}
```

用 for 循环打印一百个数：

```
var i = 1;
var count = 0;
for(var i = 0; i < 100; i++) {
    document.write(i + " ");
}
```

例 for 循环中的三句只能写一句，打印 100 个数：

```
var i = 0
for(; i < 100; ) {
    document.write(i + " ");
    i++;
}
```

例 for 循环中的函数体里面只能写一句，打印 100 个数：

```
var i = 100;
for(; i --; ) {
    document.write(i + " ");
}
```

三、while, do while

while 循环

while 循环是 for 循环的简化版 for(； ；){}，while 循环底层机制是 for 循环。

for (; 只在这一部分写，前后不写 ;){}

```
var i = 0
for(;i < 10;) {
    document.write(i);
    i ++;
}
```

上下这两个完全相等

```
while(i < 10) {
    document.write(i);
    i ++;
}
```

死循环 never-ending loop 无限循环

```
while(1) {
    document.write(i);
    i ++;
}
```

例打印一百以内，7 的倍数就输出

```
var i = 0;
while(i < 100) {

    if(i % 7 == 0) {
        document.write(i + " ");
    }

    i ++;
}
```

例打印一百以内，7 的倍数或逢 7 就输出

```
var i = 0;
while(i < 100) {

    if(i % 7 == 0 || i % 10 == 7) {
        document.write(i + " ");
    }

    i ++;
}
```

do while 循环

do while 是不管满不满足条件都会先执行一次，再判断成不成立，如果成立才会执行第二次，不成立就停止　　一般没人用

```
do{
    document.write('a');
    i ++;
}while(i < 10)
```

```
var i = 0;
do{
    document.write('a');
    i ++;
}while(i < 10)
```

aaaaaaaaaa

作业：（先找规律，再写出来）

提示：var n = parseInt(window.prompt('input'));

JS 可以进行浮点计算

1.计算 2 的 n 次幂，n 可输入，n 为自然数。

2.计算 n 的阶乘，n 可输入。即 5！=5*4*3*2*1，最好写个 if

3.著名的斐波那契额数列(这个数列从第 3 项开始，每一项都等于前两项之和)

1 1 2 3 5 8 输出第 n 项

4.编写一程序，输入一个三位数的正整数，输出时反向输出。如：输入 456,输出 654

5.输入 a,b,c 三个数字，打印出最大的。

6.打印出 100 以内的质数（从 1 除到他本身，只能有两个因数）

作业答案:

1.原有结果*2,mul 是存上一个数的结果

```
2
2 * 2
2 * 2 * 2
2 * 2 * 2 * 2
```

```
// 1 * 2
// 1 * 2 * 2
// 1 * 2 * 2 * 2
// 1 * 2 * 2 * 2 * 2
```

```
<script type="text/javascript">
    var n = parseInt(window.prompt('input'));
var mul = 1;
for(var i = 0; i < n; i ++) {
    mul *= 2;
}
document.write(mul);
```

2.

```
<script type="text/javascript">
    var n = parseInt(window.prompt('input'));
var mul = 1;
for(var i = 1; i <= n; i ++) {
    mul *= i;
}
document.write(mul);
```

```
5! = 5 * 4 * 3 * 2 * 1 * 1;
4! = 4 * 3 * 2 * 1 * 1;
```

3. 第六位是四次运算，第七位是五次运算（第三位=第一位+第二位）

第一次计算的完整过程是把第一位和第二位相加，等于第三位，并且把游标向后挪一位。下面 for（里面的 i 是控制循环圈数的）

```
<script type="text/javascript">
    var n = parseInt(window.prompt('input'));
var n = parseInt(window.prompt('input'));
var first = 1,
    second = 1,
    third;

if(n > 2) {
    for(var i = 0; i < n - 2; i ++) {
        third = first + second;
        first = second;
        second = third;
    }

    document.write(third);
}else{
    document.write(1);
}
```

	f	s		
1	1	2	3	5
		f	s	t

4. 取模除减。先把 456 提出出来，再反过来。6 是取模除减%10 余 6。再用 456 减 6，剩下 450；450%100 是 50，50 除以 10，取出 5，450-50 剩下 400；400%100，再反着乘一遍，十位乘以 10，百位乘以 100。（中间的位数其实可以不动）

5.

```
var a = parseInt(window.prompt('input'));
var b = parseInt(window.prompt('input'));
var c = parseInt(window.prompt('input'));

if(a > b) {
    if(a > c) {
        document.write(a);
    }else{
        document.write(c);
    }
}else{
    if(b > c) {
        document.write(b);
    }else{
        document.write(c);
    }
}
```

6. (1)有一个判断质数的算法
(2)看看每一个 i 是不是质数
(3)从自身开始除一直除到 1，只能被自己和 1 整除，只能整除 2 次

```
var count = 0;
for(var i = 1; i < 100; i ++) {
    // 看看每一个i是否是质数
    //
    //
    for(var j = 1; j <= i; j++) {
        if(i % j == 0) {
            count ++;
        }
    }
    if(count == 2) {
        document.write(i + " ");
    }
    count = 0;
}
```

以下是最简单的方法：从 1 到 10 能整除，除平方数以下的

```
var count = 0;
for(var i = 2; i < 100; i ++) {
    // 看看每一个i是否是质数

    for(var j = 1; j <= Math.sqrt(i); j++) {
        if(i % j == 0) {
            count ++;
        }
    }

    if(count == 1) {
        document.write(i + " ");
    }
    count = 0;
}
```

用 console.log(a);在控制台输出，看看 a 被定义了没有

条件语句补充

一、switch case 条件判断语句

if(条件判断)
switch(条件){
    case 写条件：里面判是否相符：
        如果相符合就执行 case 后面的语句比如 console.log('a')
}
switch 不负责任，如果判断了 a 是符合条件的，也会把后面的连带打印出来
加个 break，就可以终止语句

例

```
<script type="text/javascript">
    var n = 2;
    switch(n) {
        case "a":
            console.log('a');
        case 2:
            console.log('b');
        case true:
            console.log('c');
    }
```

b
c

switch 找到满足要求的语句后，后面的语句虽然不判断了，但是也会执行出来
加个 break，就可以终止 swich case 语句

例

```
<script type="text/javascript">
    var n = 2;
    switch(n) {
        case "a":
            console.log('a');
            break;
        case 2:
            console.log('b');
            break;
        case true:
            console.log('c');
            break;
    }
```

例 if(score == 90){}else if (score == 100){}

例

```
var date = window.prompt('input');

switch(date) {
    case "monday":
        console.log('working');
        break;
    case "tuesday":
        console.log('working');
        break;
    case "wednesday":
        console.log('working');
        break;
    case "thursday":
        console.log('working');
        break;
    case "friday":
        console.log('working');
        break;
    case "周六":
        console.log('relaxing');
        break;
    case "周日":
        console.log('relaxing');
        break;

}
```

简化写法

```
var date = window.prompt('input');

switch(date) {
    case "monday":
    case "tuesday":
    case "wednesday":
    case "thursday":
    case "friday":
        console.log('working');
        break;
    case "周六":
    case "周日":
        console.log('relaxing');
        break;

}
```

二、break 的标准定义是终止循环，break 必须要放在循环里面

switch, for,while 都是循环

例

```
var i = 0;

while(1) {
    i ++ ;
    console.log(i);
    if(i > 100) {
        break;
    }
}
```

break 终止的是 while，对 if 没有影响

例从 0 开始加，加到 100 以上就停止

```
var i = 0;
var sum = 0;
for(var i = 0; i < 100; i++) {

    sum += i;
    console.log(i);
    if(sum > 100) {
        break;
    }

}
```

三、continue 继续

终止本次循环，后面的都不执行了，来进行下一次的循环

// js 里面是没有 goto 的，c 语言里面有

例当 i 是 7 的倍数，或尾数是 7 的时候，不打印

```
for ( var i = 0; i < 100; i++){
    if( i % 7 == 0 || i % 10 == 7){
}else{
console.log(i);
}
}
```

下面写法更好

```
for(var i = 0; i < 100; i++) {
    if(i % 7 == 0 || i % 10 == 7) {
        continue;
    }
    console.log(i);

}
```

continue 终止本次循环，来进行下一次循环

初识引用值

一、数组（下面方括号的），arr =也是数组

例 var arr = [1,2,3,4,5,6,7,″ abc″ ,undefined];

arr [0] 代表查数组的第一位，因为数字是从 0 开始的算的

arr [0] = 3; 是指把数组的第一位改成 3，显示 3,2,3,4,5,6,7, ″ abc″ ,undefined

arr.length;是数组的长度，有多少位就有多少

console.log(arr.length); //答案是 8 位

例：利用 for 循环把数组中的每一位都拿出来——遍历

```
var arr = [1,2,3,45,5,7,"abc",undefined];

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

例：把数组中的每一位都改成 1

```
var arr = [1,2,3,45,5,7,"abc",undefined];

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

```
> arr
< ▶ [1, 1, 1, 1, 1, 1, 1, 1]
```

例：把数组中的每一位都加 1

```
> arr
< ▶ [2, 3, 4, 4, 5, 6, 10, 11]
```

```
var arr = [1,2,3,3,4,5,9,10];

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

二、对象 object

面向对象的编程方法

var obj = {

里面存属性和方法

key 属性名：value 属性值;

}

在{}面用。属性与属性之间用逗号隔开

//属性值可以双引号或单引号；属性名是为了方便找到他，只是一个辅助

例

```
var deng = {

    lastName : "Deng",
    age : 40,
    sex : undefined,
    wife : "xiaoliu",
    father : "dengdaye",
    son : "xiaodeng",
    handsome : false


}
```

```
console.log(deng.lastName);
deng.lastName = "Old Deng";
console.log(deng.lastName);
```

console.log(deng.lastName); **//取值方式 deng.lastName**

赋值 deng.lastName = “old deng”;

console.log(deng.lastName);

编程形式的区别

1.面向过程，如 c

第一步干嘛，第二步干嘛

2.面向对象（对象 object）

现在 js 是一半面向过程，一半面向对象，前面学的都是面向过程

typeof 操作符

typeof 能返回的六种数据类型（区分数字类型）

number、string、boolean、undefined、object、function

例 var num = 123;console.log(typeof(num)); //返回 number

写成 console.log(typeof num);也可以不过最好加括号

例 var num = {}; console.log(typeof(num)); //泛泛的引入值都返回 object

例 var num = [];console.log(typeof(num)); //泛泛的引入值都返回 object

例 var num = null;console.log(typeof(num)); //答案 null 返回 object ,最早是代替空对象的

例 var num = undefined;console.log(typeof(num)); //答案返回 undefined

例 var num = fuction(){};console.log(typeof(num)); // 答案返回 function

类型转换

例 var num = 1 + “1”; //显示 11

例 var num = 1 * “1”;console.log(typeof(num) + “:” + num); //显示 number:1

例 var num = 1 - “1”;console.log(typeof(num) + “:” + num); //显示 number:0

例 var num = “2” - “1”;console.log(typeof(num) + “:” + num); //显示 number:1

例 var num = “2” * “1”;console.log(typeof(num) + “:” + num); //显示 number:2

以上例子说明 js 有类型转换

一、显示类型转换

Number(mix) 是想把里面的东西转换成数字

例 var num = Number(‘123’);

console.log(typeof(num) + “:” + num);

答案显示 Number:123，把字符串类型的 123 转换成了 number 类型

例 var demo = “123”;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:123，上面那一行的 Number 是为了把()里面转换成数字类型

例 var demo = true;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:1

例 var demo = false;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:0

例 var demo = null;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:0

例 var demo = undefined;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:NaN

例 var demo = “abc”;

var num = Number(demo);

console.log(typeof(num) + “:” + num);

答案显示 Number:NaN

例 var demo = “-123”;

var num = Number(demo);
console.log(typeof(num) + “:” + num);
答案显示 Number:-123
例 var demo = “123abc”;
var num = Number(demo);
console.log(typeof(num) + “:” + num);
答案显示 Number:NaN

parseInt(string,radix)

parse 是转化，Int 是整型，整数，目的是把里面转换成整数

例 var demo =” 123”;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number:123
例 var demo = true;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number: NaN
例 var demo = false;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number: NaN
例 var demo = 123.9;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 123，此处是直接去掉小数，不是四舍五入
例 var demo = “10”;
var num = parseInt(demo ,16);
console.log(typeof(num) + “:” + num);
答案显示 number: 16
var num = parseInt(demo ,radix); **//radix 是基底的意思**
radix 写成 16，系统会认为是以 16 进制为基底， 10（一零）是 16 进制的一零，是以 16 进制为基底 ,把他转成为 10 进制的数字(就是 16),上面是以目标进制为基底，转换成十进制（radix 范围是 2-36）
例 var demo = “3”;
var num = parseInt(demo ,2);
console.log(typeof(num) + “:” + num);
答案显示 number: NaN
例 var demo = “b”;
var num = parseInt(demo ,16);
console.log(typeof(num) + “:” + num);
答案显示 number: 11
例 var demo = “123abc”;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 123
例 var demo = “100px”;
var num = parseInt(demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 100
parseInt 从数字类开始看，看到非数字类为止，返回原来的数

parseFloat(string)

parseFloat(string)转换成浮点数字，就是正常小数
例 var demo = “100.2”;
var num = parseFloat (demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 100.2
例 var demo = “100.2.3”;
var num = parseFloat (demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 100.2
例 var demo = “100.2abc”;
var num = parseFloat (demo);
console.log(typeof(num) + “:” + num);
答案显示 number: 100.2
parseFloat 从数字类开始看，看到除了第一个点以外的非数字类为截止，返回前面的数

toString

例 var demo = 123;
var num = demo.toString();
console.log(typeof(num) + “:” + num);
答案显示 string: 123。相当于把 123 转换字符串。
想把谁转换成字符串，就写成谁.toString，上面是想把 demo 转换成 toString，写成 demo.toString

例 var demo = undefined;
var num = demo.toString();
console.log(typeof(num) + “:” + num);
答案显示报错，**undefined 和 null 不能用 toString**
例 var demo = 123;
var num = demo.toString(8);
console.log(typeof(num) + “:” + num);
答案 173，把 123 转成为八进制
这里的 radix 意思是以十进制为基底，转换成目标进制（即 8 进制）
例 var demo = 10;
var num = demo.toString(8);
console.log(typeof(num) + “:” + num);
答案 12
例 var demo = 20;
var num = demo.toString(8);
console.log(typeof(num) + “:” + num);
答案 24。**以十进制为基底，把 20 转换成 8 进制，就是 24**
例给你一个二进制的数，转换成十六进制，是先从二进制到十进制再到十六进制
var num = 10101010;
var test = parseInt(num, 2);
console.log(test.toString(16));

```
var num = 10101010;
var test = parseInt(num, 2);
console.log(test.toString(16));
```

答案 aa
例 var num = 10000;
var test = parseInt(num, 2);
console.log(test.toString(16));
答案 10

String(mix)

String(mix)转换成字符串，写什么都成了字符串
例 var demo = 123.234;
var num = String (demo);
console.log(typeof(num) + “:” + num);
答案显示 string: 123.234
例 var demo = undefined;
var num = String (demo);
console.log(typeof(num) + “:” + num);
答案显示 string: undefined

Boolean()

Boolean()转换成布尔值 false 和 true
例 var demo = “” ;
var num = String (demo);
console.log(typeof(num) + “:” + num);
答案显示 boolean: false

二、隐式类型转换

隐式类型转换是跟你转换了也不知道
隐式类型转换内部隐式调用的是显示的方法
隐式类型转换包括 isNaN () ，++，-- ， +/-（一元正负），+，*，% ，，&&，|| ，!，
<，>，<= ，>= ，== ，!=

isNaN ();

isNaN ();当你把一个数放到()里，它能判断是不是 NaN，先比括号里面的放到 number 里面转换，然后返回来
例 console.log(isNaN(NaN); //答案 true
例 console.log(isNaN(“123”); //答案 false
例 console.log(isNaN(“abc”); //答案 true。会调用 number，先把“abc”放 number 里面转换，通过 number 的转换再和 NaN 比对，如果相等就是 true
例 console.log(isNaN(null); //答案 false，在 number 里面放 null 是 0，不是 NaN
例 console.log(isNaN(undefined); //答案 true

++/--（加加减减） +/-（一元正负）

例 var a = “123” ;
a ++;
答案 124，++这个符号放到这里，还没运算之前，先把前面的 a 转换成 number 的 123
例 var a = “abc” ;
a ++;
答案 NaN

```
> a
< NaN
> typeof(a)
< "number"
```

+/-（一元正负）
+a;-a;正 a 和负 a 都会变换成数字
例 var a =+” abc” ;
console.log(a + “:” + typeof(a));
答案 NaN:number。尽管转换不成数字，也会转换成数字类型，因为里面隐式的调用了一个 number

+

+隐式类型会转换成 string，**当加号两侧有一个是字符串，就用调用 string**，把两个都变成字符串

例 var a = “a” +1
console.log(a + “:” + typeof(a));

*** %**

*和% 乘和模都会转换成 number.

例 var a = “1” *1; console.log(a + “:” + typeof(a));

答案 1：number

例 var a = “a” *1; console.log(a + “:” + typeof(a));

答案 1：number，先是 number(“a”)的结果乘以 number(1)的结果，最后是 NaN*1，还是 NaN，但是数据类型是 number

&& || !

与或非，都是有类型转换的，不过是返回的是表达式的值，不是隐士类型转换的值，但是判断是一个类型转换的值

< > <= >=

例 var a =1 > “2”; console.log(a + “:” + typeof(a));

答案 false:boolean，有数字相比较的，就会隐士类型转换成数字类型

例 var a =”3” > “2”; console.log(a + “:” + typeof(a));

答案这个没类型转换，这个比的是 ASCⅡ

例 var a =”3” > 2; console.log(a + “:” + typeof(a));

答案 true：boolean 会转换成数字，因为数字优先

== !=

例 var a = 1 == “1”; console.log(a + “:” + typeof(a));

答案 true:boolean，也有隐士类型转换

例 var a = 1 == true; console.log(a + “:” + typeof(a));

答案相等

！=也是这样

特殊东西，在控制台操作

例 false>true //**答案** false，会先转换成数字，0>1 当然是错的
例 2>1>3 //**答案** false
例 2>3<1 //**答案** true
例 10>100>0 //**答案** false
例 100>10>0 //**答案** true

以上都是挨个算的，先看前面的是 true 还是 false，再和后面的比，不是顺着下来

例 undefined>0 //**答案** false
例 undefined==0 //**答案** false
例 undefined<0 //**答案** false
例 null>0 //**答案** false
例 null==0 //**答案** false
例 null<0 //**答案** false
例 undefined == null //**答案** true
例 NaN ==NaN //**答案** false，NaN 是唯一一个连自己都不等于的

三、不发生类型转换

===绝对等于（三个等号） !==绝对不等于

例 1 === 1 //**答案** true
例 1 === “1” //**答案** false
例 1 !== “1” //**答案** true
例 1 !== 1 //**答案** false
例 NaN =NaN //**答案** false **特殊的**
例 console.log(a); //如果定量没定义就直接访问，就 a is not defined 报错；有一种特殊情况，当且仅当把未定义的变量放到 console.log(typeof(a));里面就访问，不报错，返回 undefined
例 console.log(typeof(a)); //**答案** undefined ,这个 undefined 是字符串
例 console.log(typeof(typeof(a))); //**答案** string。console.log(typeof(typeof(a)));可以先解析成 console.log(typeof(“undefined”));再返回一次就是 string 字符串

上面考的是 typeof(a)返回的六种类型的值（number、string、boolean、undefined、object、function）都是 undefined 字符串

作业

例 alert(typeof(a)); //返回 string
例 alert(typeof(undefined)); //返回 string，undefined
例 alert(typeof(NaN)); //返回 number
例 alert(typeof(null)); //返回 object
例 var a = “123abc”; //返回 string
例 alert(typeof(+a)); //返回 number，NaN
例 alert(typeof(!!a)); //返回 boolean
例 alert(typeof(a + “”)); //返回 string
例 alert(1 == “1”); //**显示** true
例 alert(NaN == NaN); //**显示** false
例 alert(NaN == undefined); //**显示** false
例 alert(“11” + 11); //**显示** 1111
例 alert(1 === “1”); //**显示** false
例 alert(parseInt(“123abc”)); //**显示** 123【parseInt 是截断数字】

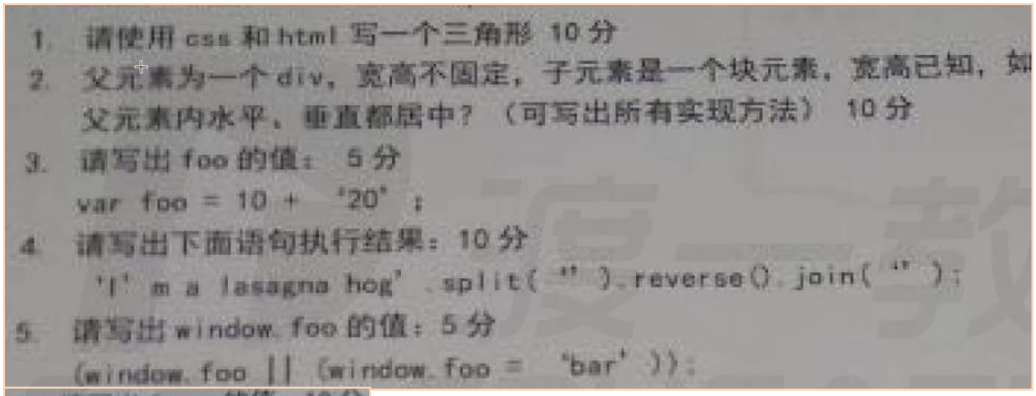
例 typeof(typeof(a)); //返回 string

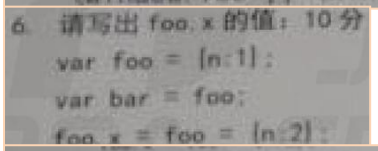
例 var num = 123123.345789;alert(num.toFixed(3));

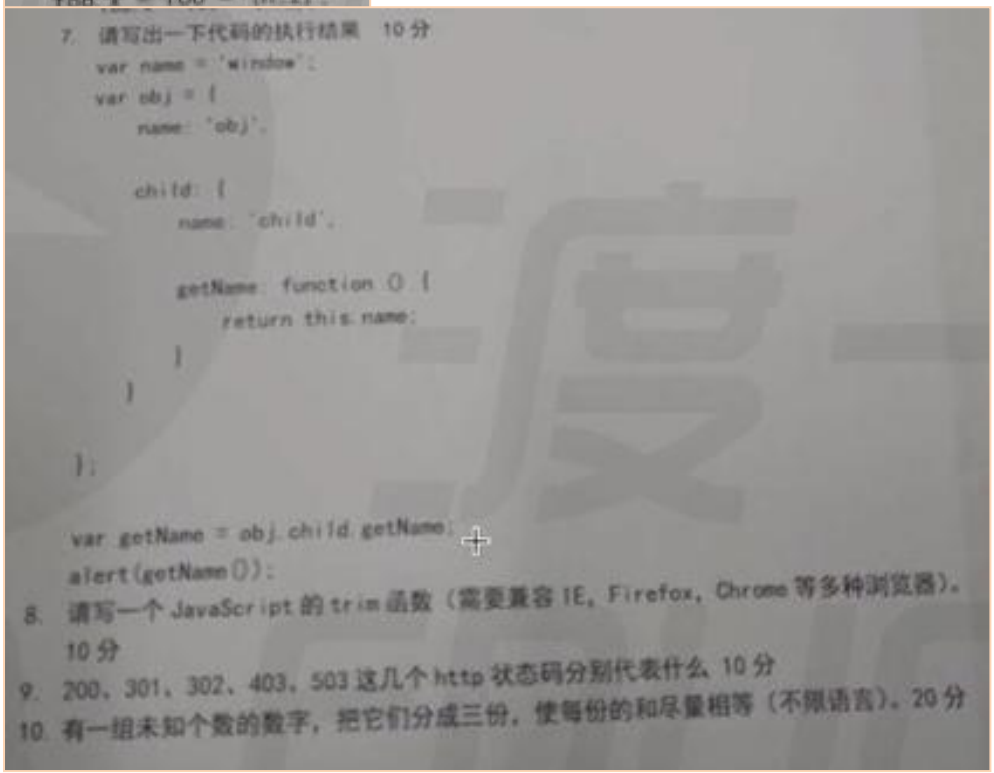
答案 123123.346【toFixed(3)是保留三位小数的意思，四舍五入】

提示：

alert('a');就是弹出框，相当于 console.log







函数 function

function 随便起个名(){}

例

```
if(1 > 0) {
    document.write('a');
    document.write('b');
    document.write('c');
}

if(2 > 0) {

    document.write('a');
    document.write('b');
    document.write('c');

}


if(3 > 0) {
    document.write('a');
    document.write('b');
    document.write('c');
}
```

```
function test() {
    document.write('a');
    document.write('b');
    document.write('c');
}


if(1 > 0) {
    test();
}

if(2 > 0) {
    test()
}

if(3 > 0) {
    test();
}
```

以上情况就是偶合，偶合度非常高，偶合代码就是低效代码

编程讲究高内聚，弱偶合

右上方是简便写法：可以用 test 调用执行，写几个 test 就调用执行几次

例

```
<script type="text/javascript">

    function test() {

        var a = 123;
        var b = 234;
        var c = a + b;
        document.write(c);


    }

    test();
```

答案 357。写了一个 test ( )；就执行了一遍函数语句，如果不写 test ( )；就相当于有一个框来存东西，但是不执行

例 function test (){

document.write('hello world');

}

test();

函数 function 可以先定义功能，之后再去用

一、定义

1、函数声明

定义一个函数可以先写一个 function,函数就是另一个类型的变量

我声明一个函数 test,test 是函数名。写成下面

```
function test(){
    函数体
}
```

函数名起名：开发规范要求，函数名和变量名如果由多个单词拼接，必须符合小驼峰原则（第一个单词首字母小写，后面的首字母大写）

例 function theFirstName(){}

document.write(theFirstName);

答案 function theFirstName(){}。打印出来的是函数体

这与 c 语言和 c++，他们打印指针，会输出指针的地址，而 js 这种弱数据语言（解释性语言）永远不输出地址，输出地址指向房间

2、函数表达式

例 var test = fuction test (){

document.write('a');

}

test();

答案 a。这种方式像定义一个变量

上面这种方式，可以演变成第三种，匿名表达式【不写 test 这种函数名】

例 var demo = fuction (){

document.write('a');

}

（1）命名函数表达式

例

```
var test = function abc() {
    document.write('a');
}
```

```
> test
< function abc() {
            document.write('a');
        }
> abc
✖ ▶ Uncaught ReferenceError: abc is not defined(…)
```

上面这个函数的函数名 name 是 abc

在控制台 console 直接输出 test 就会出现→

在控制台 console 直接输出 abc 会报错，表达式就会忽略他的名字 abc。

在上面例子中，fuction abc(){document.write('a');}这一部分叫表达式，是会忽略 abc 这个地方的名字，会变成匿名函数表达式，不如直接写成匿名函数

（2）匿名函数表达式（常用，一般说的函数表达式就是匿名函数表达式）

```
function test() {}
```

```
> test.name
< "test"
```

二、组成形式

1、函数名称

function test(){}其中 function 是函数关键字，test 是函数名，必须有(){}，参数可有可没有，参数是写在()括号里面的。

如果写成 function test(a，b){}，相当于隐式的在函数里面 var a，var b 申明了两个变量，( ) 括号里面不能直接写 var

例 function test(a, b){

  document.write(a + b)

}

test(1, 2)

答案 3。上面这个例子，1 就会到 a 里面去，2 就会到 b 里面去，这是传参的形式

2、参数(可有可没有，但是高级编程必有)

(1)形参（形式参数）：指的是 function sum（a，b）{}括号里面的 a 和 b

(2)实参（实际参数）：指的是 sum（1，2）；里面的 1，2

天生不定参，形参可以比实参多，实参也可以比形参多

例

```
function sum(a, b) {
    var c = a + b;
    document.write(c);
}

sum(1, 2);
sum(3, 4);
```

答案 37，参数把函数抽象了，可以组成很多形式

例如果第一个实参的数大于 10，就减第二个数的运算结果；如果第一个实参的数小于 10，就加第二个数的运算结果

```
// function test() {}
// 形式参数 -- 形参
function sum(a, b) {
    if(a > 10) {
        document.write(a - b);
    }else if(a < 10) {
        document.write(a + b);
    }else{
        document.write(10);
    }
}

// 实际参数 -- 实参
sum(1, 2)
```

答案 3

上面改成 sum(11,2)　　//答案 9

例 function sum(a,b){
document.write(a);
}
sum(11, 2, 3)
答案 11
例 function test(a, b, c,d) {
document.write(a);
document.write(d);
}
sum(11, 2, 3)
答案 11,undefined，上面这道题是形参多，实参少
js 参数不限制位置，天生不定参数
在每一个函数里面都有一个隐式的东西 arguments 这个是实参列表
console.log();是把信息展示在控制台
document.write();是把信息展示到网页
例 function test(a) {
console.log(arguments);
console.log(arguments.length);
}
sum(11, 2, 3)

```
▶ [11, 2, 3]
3
```

答案[11, 2, 3]，3
例 function test(a) {
for(var i = 0; i < argument.length; i++){
console.log(arguments[i]);
}
}
sum(11, 2, 3)

```
11
2
3
```

答案 11,2,3

```
function sum(a, b, c, d) {

    if(sum.length > arguments.length) {
        console.log('形参多了')
    }else if(sum.length < arguments.length) {
        console.log('实参多了')
    }else{
        console.log('相等');
    }

}


// 实际参数 -- 实参
sum(11, undefined, 3, "abc")
```

例：形参长度求法

```
function sum(a, b, c, d) {
    console.log(sum.length);
    }
sum(11, 2, 3)
```

答案 4
例任意个数求和(不定参才能求出来)

```
function sum() {

    //arguments [1,2,3,4,5,6,7];
    var result = 0;
    for(var i = 0; i < arguments.length; i++) {
        result += arguments[i];
    }
    console.log(result);

}

sum(1,2,3,4,5,6,7,8,9);
```

形参永远有尽头，要实现任意的数求和，无法定义形参。
例

```
function sum(a, b) {
    //arguments [1,2]
    //var a = 1;
    a = 2;
    console.log(arguments[0]);

}

sum(1,2);
```

```
2
```

答案是 2。a 变，arguments 跟着变。有一个映射关系。
例

```
function sum(a, b) {
    //arguments [1,2]
    //var a = 1;
    a = 2;
    arguments[0] = 3;
    console.log(a);

}

sum(1,2);
```

```
3
```

答案 3。arguments 里面一个变，一个跟着变，但是[1,2]是两个人，相当于映射关系。

例当形参两个，实参一个

```
function sum(a, b){
   //arguments[1]没值
   b = 2;
   console.log(arguments[1]);
}
sum(1);
```

答案 undefined ,实参列表出生时有几个 ,就有几个 ,在写 b=2 ,也不加在 arguments[1] 里面了，此处的 b 就当变量用，他和实参不映射。

形参实参完全对应上才映射

例

```
function sum(a, b){
   //arguments[1]没值
   a = 2;
   console.log(arguments[0]);
}
sum(1);
```

答案 2

3、返回值 return

结束条件和返回值 return，return 有终止函数的功能

没写 return，实际上是加上了一个隐式的 return

例

```
function sum(a, b){
    console.log( 'a' );
    console.log( 'b' );
    return;
}
```

答案 a，b

例

```
function sum(a, b){
    console.log('a');
    return;
    console.log('b');
}
```

答案 a

return 最常用的是返回值。本意是把一个值返回到函数以外

自己定义的函数也能返回,return 空格 123

例

```
function sum(){
   return 123;
   console.log( 'a' );
}
var num = sum();
```

答案这里的 num 就是 123 ,而且 console.log('a');无效 ,**这里的 return 又终止函数，又返回变量**

例把 target 转成数字

```
function myNumber(target){
   return +target;                    //利用+隐式的转换成了数字类型
}
var num = myNumber( '123' );
console.log(typeof(num) +  ""  + num);
```

答案 number 123

一般函数处理完一个参数，是为了返回

typeof()也是返回值，也是函数

typeof(123)也可以写成 typeof 123【typeof 空格 123】，只是看起来不方便

作业

1.写一个函数，功能是告知你所选定的小动物的叫声。

2.写一个函数，实现加法计数器。

3.定义一组函数，输入数字，逆转并输出汉字形式。

4.写一个函数，实现 n 的阶乘。

5.写一个函数，实现斐波那契数列。

答案

1、

```
function scream(animal) {

    switch(animal) {

        case "dog" :
            document.write('wang!');
            return;
        case "cat" :
            document.write('miao!');
            return;
        case "fish":
            document.write('o~o~o~');
            return;
    }
}
```

此处可用 break，也可以用 return

2、没写。之前笔记里面有

3、
```
function reverse() {
    var num = window.prompt('input');
    var str = "";
    for(var i = num.length - 1; i >= 0; i --) {
        str += transfer(num[i]);
    }
    document.write(str);
}

function transfer(target) {
    switch(target) {
        case "1" :
            return "壹";
        case "2" :
            return "俩";
        case "3" :
            return "仨";
    }
}
```

```
> var str = '123'
< undefined
> str.charAt(0)
< "1"
> str.charAt(1)
< "2"
```

```
> var str = "123";
< undefined
> str[0]
< "1"
> str[1]
< "2"
> str.length
< 3
```

```
> var str = "123";
< undefined
> str += "234"
< "123234"
```

reverse 是逆转

str.charAt(0)意思是把第 0 位拿出来

for(var i = num.length - 1; i >= 0)这是倒着拿的意思

一个字符串长度是 3，他的第三位字符串是 2

可以把第二个 function 放第一个里面，但是一般单独写，因为 transfer 是转换

str += transfer(num[i]);是先执行后面的 transfer(num[i])，再赋值给前面

4、方法一 function jc(n){

```
    //n 的阶乘
    for(var i = 1; i <= n; i ++){
        num *=i;
    }
}
```

方法二 function mul(n){

```
    if(n == 1 || n == 0){
        return 1;
}
        return n * mul(n - 1);
}
```

方法二叫递归：

1 找规律，

2 找出口（找停的点）

递归的优点是代码简洁，缺点慢

```
function mul(n) {

    //n的阶乘
    // var num = 1;
    // for(var i = 1; i <=n; i ++) {
    //     num *= i;
    // }
    if(n == 1) {
        return 1;
    }
    return n * mul(n - 1);

}
// 递归
mul(5);

mul(5) ==> 5 * 24;
mul(4) ==> 4 * 6;
mul(3) ==> 3 * 2;
mul(2) ==> 2 * 1
```

导图 return 5 * mul(5-1)

return (5 - 1) * mul(5 - 2)

return (3) * mul(5- 3)

return (5 - 3) * mul(5 - 4)

return 1

```
// n! = n * (n - 1)!

function mul(n) {

    if(n == 1 || n == 0) {
        return 1;
    }
    return n * mul(n - 1);
}
```

5、拿递归写斐波那契数列 fb(n)==fb(n - 1) + fb(n - 2);

答 function fb(n){

```
    if(n == 1 || n== 2){
        return 1;
    }
return fb(n - 1) + fb(n - 2);
}
```

```
// fb(n) == fb(n - 1) + fb(n - 2);
//
function fb(n) {
    if(n == 1 || n == 2) {
        return 1;
    }
    return fb(n - 1) + fb(n - 2);
}

fb(5) == > fb(4) + fb(3);
fb(4) == > fb(3) + fb(2); fb(3) == fb(2) + fb(1);
fb(3) == > fb(2) + fb(1);
```

递归返回的顺序：先执行的最后被返回，最后执行完（等最底层的一层层返回）

作用域初探

作用域定义：变量（变量作用于又称上下文）和函数生效（能被访问）的区域

全局、局部变量

作用域的访问顺序

作用域:函数里面的可以访问外面的全局变量

例

```
var a = 123;
// 全局变量
function test() {
    var b = 123;
    function demo() {
        var c = 234;
        document.write(b);
        document.write(a);
    }
    demo();
    document.write(c);
}
test();
```

函数外面不能用函数里面的。里面的可以访问外面的，外面的不能访问里面的，彼此独立的区间不能相互访问

test{}和 demo{}不能互相访问，但是可以访问外面的全局变量

例

彼此独立的区间不能相互访问

全局变量都可以访问

```
var global = 100;
function test() {
    var a = 123;
}

function demo() {
    var b = 234;
}
```

例 function test(){
    var a =123;
    function demo(){
      var b = 234;
    document.write(a);
    }
    demo();
    document.write(b);
}
上面的 document.write(b);不能访问 var b ，
上面的 document.write(a);可以访问 a
外层函数不能访问里层的，里层的可以访问外层的，越往里权限越大
作业
要求输入一串低于 10 位的数字，输入这串数字的中文大写。
例如：input :10000　　output:壹万
例如：input :1001010　　output:壹佰万壹仟零壹拾
千分位如果是 0 必须读零

js 运行三部曲
1 语法分析 → 2 预编译 → 3 解释执行

预编译前奏
例 function test(){
  console.log('a');
}
test();
上面能执行
例 test();
function test(){
  console.log('a');
}
也能执行，因为有预编译的存在
例 var a = 123;
console.log(a);
答案 123
例 console.log(a);
var a = 123;
答案 undefined
例只写 console.log(a);就会报错
函数声明整体提升：函数不管写到哪里，都会被提到逻辑的最前面。所以不管在哪里调用，本质上都是在后面调用
变量　声明提升：把 var a 提升到最前面
var a = 123;这是变量声明再赋值。
变量　声明提升是把他拆分成 var a; a = 123;然后把 var a 提升到最前面
上面这两句话没办法解决下面例子的问题
例 function a(a){
    var a = 234;
    var a = function(){
    }
    a();
}
var a =123;
1.imply global 暗示全局变量：即任何变量，如果变量未经声明就赋值，此变量就为全局对象(就是 window)所有。
全局对象是 window
例 window.a = 10;
例 a = 10; ===> windows.a = 10;
eg: var a = b = 123;
2.一切声明的全局变量，全是 window 的属性。
例 var a = 123; ===> window.a = 123;
window 就是全局的域
如果在全局变量在 var a = 123；那么就会返回到 window
例 var a = 123
console.log(a) ===> window.a
例 var a = b = 234;是把 234 的值赋给 b，在把 b 的值赋给 a
例 function test(){
    var a = b = 123;
}
test()
写 test()代表执行 test ,赋值是自右向左的 ,上面先把 123 赋给 b 的时候 ,b 未经声明，然后再声明 a，再 b 的值赋给 a，导致 b 未经声明，所以 b 归 window 所有
访问 window.a 是 undefined，访问 window.b 是 123
例 function test(){
  var b = 123;

}
test();
console.log(window.b);
答案 undefined
window 就是全局
例 var a = 123;
console.log(a); → console.log(window.a);
例 var a = 123;
var b = 234;
var c = 345;
window{
a : 123,
b : 234,
c : 345
}
如果 var a 对应会有 window.a

预编译（解决执行顺序问题）

例 function fn(a){
console.log(a);
var a = 123;
console.log(a);
function a (){}
console.log(a);
var b = function (){}
console.log(b);
function d() {}
}
fn(1);
答案是 function a(){}//123//123//function (){}
这个例子的形参是（a），变量声明也是 a
上面的例子按四部曲变化如下:
找形参和变量声明，将变量和形参（a）名作为 AO 属性名，值为 undefined ，AO{
a : undefined,
b : undefined,
}
（把实参值传到形参里）AO{
a : 1,
b : undefined,
}
function a () {}和 function d () {}都是函数声明，但是 var b = function (){}不是。AO{
a : function a () {},
b : undefined,
d : function d () {}
}
执行第一行 console.log(a);时，用的是 AO{
a : function a () {},
b : undefined,
d : function d () {}
}
执行 var a =123;改变的是 AO{
a : 123,
b : undefined,
d : function d () {}
}
在 b = function (){}时 AO{
a : 123,
b : function () {},
d : function d () {}
}
预编译发生在函数执行的前一刻

（函数）预编译的四部曲：

1.创建 AO 对象 Activation Object(执行期上下文，作用是理解的作用域，函数产生的执行空间库)
2.找形参和变量声明，将变量和形参名作为 AO 属性名，值为 undefined
相当于 AO{
a : undefined,
b : undefined
}
3.将实参值和形参统一（把实参值传到形参里）
4.在函数体里面找函数声明，值赋予函数体
（先看自己的 AO，再看全局的 GO）
例子 function test (a, b){
console.log(a);
c = 0;

```
function test(a, b) {
    console.log(a);
    c = 0;
    var c;
    a = 3;
    b = 2;
    console.log(b);
    function b () {}
    function d () {}
    console.log(b);
}

test(1);
```

var c;
a = 3;
b = 2;
console.log(b);
function b (){};
function d (){};
console.log(b);
}
test(1);

答题过程 :找形参和变量声明 ,将变量和形参名作为 AO 属性名 ,值为 undefined, AO{
a : 1,
b : undefined,
c : undefined
}
函数声明 function b(){}和 function d(){}，AO{
a : 1,
b : function b(){},
c : undefined,
d : function d(){}
}
执行 console.log(a);答案是 1
执行 c = 0;变 AO{
a : 1,
b : function b(){},
c : 0,
d : function d(){}
}
var c 不用管，因为 c 已经在 AO 里面了
执行 a = 3;改 AO{
a : 3,
b : function b(){},
c : 0,
d : function d(){}
}
执行 b = 2;改 AO{
a : 3,
b : 2,
c : 0,
d : function d(){}
}
执行 console.log(b);答案是 2
function b () {}和 function d(){}已经提过了，不用管
执行 console.log(b);答案是 2

例 function test(a , b){

```
function test(a, b) {
    console.log(a);
    console.log(b);
    var b = 234;
    console.log(b);
    a = 123;
    console.log(a);
    function a () {}
    var a;
    b = 234;
    var b = function ( ){}
    console.log(a);
    console.log(b);
}

test(1);
```

console.log(a);
console.log(b);
var b = 234;
console.log(b);
a = 123;
console.log(a);
function a (){}
var a;
b = 234;
var b = function(){}
console.log(a);
console.log(b);
}
test(1);

一旦有重名的，一但有 a 变量又有 a 函数【如 function a (){}】，又在第一条访问的是 a，一定是函数

答题过程： 将变量和形参名作为 AO 属性名，AO{
a : undefined,
b : undefined
}
将实参值和形参统一，AO{
a : 1,
b : undefined
}
找函数声明 function a (){}，AO{
a : function a (){},
b : undefined
}
执行 console.log(a);答案是 function a (){}
执行 console.log(b);答案是 undefined

执行 var b = 234;变 AO{
a : function a (){},
b : 234
}
执行 console.log(b);答案是 234
执行 a = 123;变 AO{
a : 123,
b : 234
}
执行 console.log(a);答案是 123
然后 function a (){};var a ;都可以不看了
执行 b = 234，b 值还是 234，不变
执行 var b = function (){}，变 AO{
a : 123,
b : function (){}
}
执行 console.log(a);答案是 123
执行 console.log(b);答案是 function (){}

下面开始讲全局的预编译

例 console.log(a);
var a = 123;
答案 undefined
例 console.log(a);
var a = 123;
function a (){}
答案是打印 a 是 function a (){}

全局的预编译三部曲：

1、生成了一个 GO 的对象 Global Object（window 就是 GO）
2、找形参和变量声明，将变量和形参名作为 GO 属性名，值为 undefined
3、在函数体里面找函数声明，值赋予函数体

例 console.log(a);
var a = 123;
function a (){}
答案过程，GO{
    a : undefined
}
函数声明 GO{
    a : function a (){}
}
执行 var a = 123;变 GO{
    a : 123
}
执行 console.log(a);就是 123

GO === window， GO 和 window 是一个东西

console.log(a);和 console.log(window.a);和 console.log(go.a);是一样

任何全局变量都是 window 上的属性
没有声明就是赋值了，归 window 所有，就是在 GO 里面预编译

例 function test(){
    var a = b =123;
    console.log(window.b);
}
test();
答案 a 是 undefined，b 是 123
先生成 GO{
  b : 123
}
再有 AO{
  a : undefined
}

先生成 GO 还是 AO?
想执行全局，先生成 GO，在执行 test 的前一刻生成 AO
在几层嵌套关系，近的优先，从近的到远的，有 AO 就看 AO，AO 没有才看 GO

例 console.log(test);
function test(test){
  console.log(test);
  var test = 234;
  console.log(test);
  function test(){
  }
}
test(1);
var test = 123;

```
console.log(test);
function test(test) {
    console.log(test);
    var test = 234;
    console.log(test);
    function test( ){
    }
}
test(1);
var test = 123;
```

答题过程:想执行全局,先有 GO,GO{
    test : undefined

}
发现有函数声明 GO{
    test : function (){
    .....
    }
}
执行 console.log(test)，
执行 test(1)之前生成 AO{
    test : function (){}
}
执行 var test = 234;变成 234
AO 上面有就用 AO 的，没有就看 GO 的
例 var global = 100;
function fn(){
    console.log(global);
}
fn();
答题过程 GO{
    global : undefined,
    fn : function(){.....}
}
执行 var global = 100;变 GO{
    global : 100,
    fn : function(){.....}
}
不看 function fn(){...}里面的东西
执行 fn()之前 AO{
    访问 GO 的 global
}
例

```
global = 100;
function fn() {
    console.log(global);
    global = 200;
    console.log(global);
    var global = 300;
}

fn();
var global;
```

答题过程，GO{
    global : undefined
    fn : undefined(没用可以不写)
}
变 GO{
    global : 100
    fn : undefined
}
执行 fn()之前，AO{
    global : undefined
}
执行结果是 undefined，200
例

```
function test (){
    console.log(b);
    if(a) {
        var b = 100;
    }
    c = 234;
    console.log(c);
}

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

答案//undefined //undefined //234 //234
过程 GO{
  a : undefined
  test:undefined(没用可以不写)
}
AO{
  b : undefined //不管 if（a）{}，可以提出 var b 的 b
}
执行到 c=234，GO{
  a : undefined
  c : 234
}
执行到 a=10，GO{
  a : 10
  c : 234}

```
function test (){
    console.log(b);//undefined
    if(a) {
        var b = 100;
    }
    console.log(b)//undefined
    c = 234;
    console.log(c);//234
}

var a;
test();
// AO{
//   b : undefined
// }
a = 10;
console.log(c);//234
```

例 function bar(){
    return foo;
    foo = 10;
    function foo(){}
    var foo = 11;
}
console.log(bar());
答案：function foo(){}
如果在第一行 return foo，下面有 foo 这个函数，一定打印这个函数
例 console.log(bar());
function bar(){
  foo = 10;
  function foo(){}
  var foo = 11;
  return foo;
}
答案 11
例 console.log(b);
var b = function (){}
答案是 undefined
例现在**在 if 里面定义函数声明 function 是不允许的**，但是过去可以，下面就是过去的旧题，按可以来做

```
a = 100;
function demo(e) {
    function e() {}
    arguments[0] = 2;
    document.write(e);
    if(a) {
        var b = 123;
        function c() {
            //猪都能做出来
        }
    }
    var c;
    a = 10;
    var a;
    document.write(b);
    f = 123;
    document.write(c);
    document.write(a);
}
var a;
demo(1);
document.write(a);
document.write(f);
```

GO{
  a : undefined
}
GO{
  a : undefined
  demo : function (){}
}
开始执行 a=100，GO{
  a : 100
  demo : function (){}
}
AO{
  e : undefined,
  b : undefined,
  c : undefined,
  a : undefined
}
形参实参相统一，AO{
  e : 1,
  b : undefined,
  c : undefined,
  a : undefined
}
赋值 AO{
  e : function e (){},
  b : undefined,
  c : undefined,------旧规则里面可以提出 function(){}
  a : undefined
}
执行 arguments[0] = 2;实参列表和传参是否相映射，变 AO{
  e : 2,
  b : undefined,
  c : undefined,
  a : undefined
}
执行 console.log(e);答案 2
if(a)由于 a 在 AO 里面是 undefined，所以不走 if

执行 a = 10;变 AO{
e : 2,
b : undefined,
c : undefined,
a : 10
}
执行 console.log(b)，答案 undefined
执行 f = 123，变 GO{
a : 100,
demo : function (){},
f : 123
}
执行 console.log(c);　　之前打印 function(){}，改语法后打印 undefined
执行 console.log(a);　　答案 10
执行 console.log(a);　　因为在外面是全局的，答案 100
执行 console.log(a);　　答案 123

例
```
var str = false + 1;
document.write(str);
var demo = false == 1;
document.write(demo);
if(typeof(a)&&-true + (+undefined) + ""){
    document.write('基础扎实');
}
if(11 + "11" * 2 == 33) {
    document.write('基础扎实');
}
!!" " + !!"" - !!false||document.write('你觉得能打印，你就是猪');
```

答题//1　　false+1 因为有+，两边都不是字符串，就转换成数字，false 是 0
//false　　false==1，false 肯定不等于 1，所以把 false 再赋给 demo
//undefined
typeof(a) 出现"undefined"　　-true 转换成数字是-1　　+undefined 显示"NaN"
-1 + NaN = NaN
-1 + NaN + "" = "NaN"

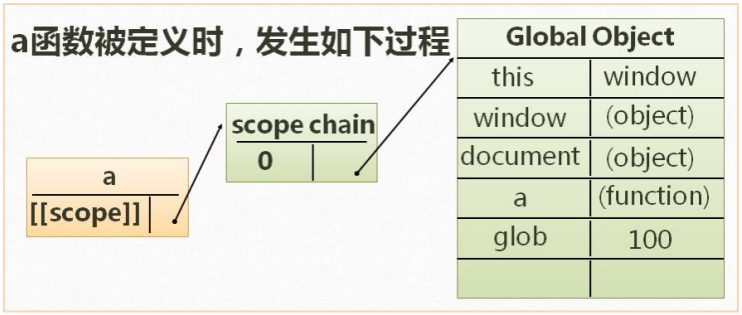
```
> typeof(a)
< "undefined"
```

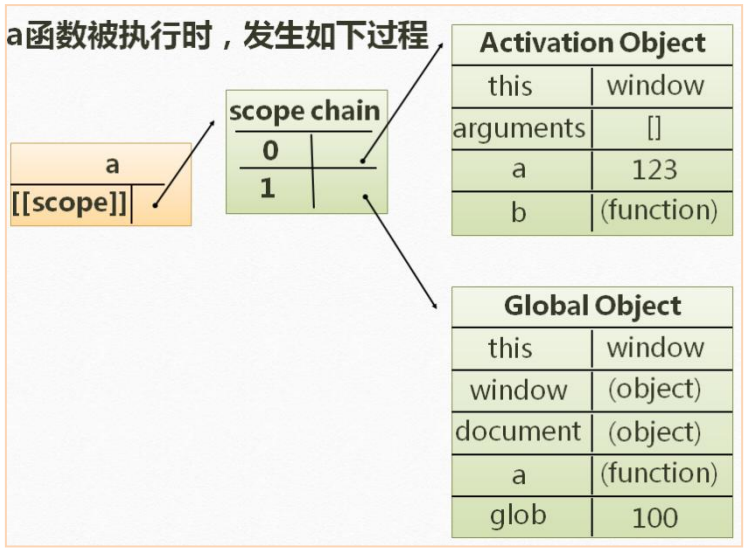
"undefined" && "NaN"转换成 boolean，就都是 true
"11" * 2 是*把两边转换成了数字，所以 11 +"11" * 2 =33，33 == 33，两边相等
!!非非就是正
" " 这不是空串，是空格字符串
!!" " 转换成 Boolean 为 true
!!"" 非非空串，转换为 Boolean 为 false
!!false 就是 false
true + false - false = 1 + 0 – 0 =1
11|| document.write('你觉得能打印？')
||遇到真就听，1 为真，所以返回 1
例(window.foo || (window.foo = 'bar'));求 window.foo
答案" bar"
这道题要先看(window.foo = 'bar')这一边的，再看左边的 window.foo，因为运算符的顺序；但是这道题错误的读法（从左到右）也是 bar
(window.foo || window.foo = 'bar');这么写就报错；||或运算符优先级高于=等号

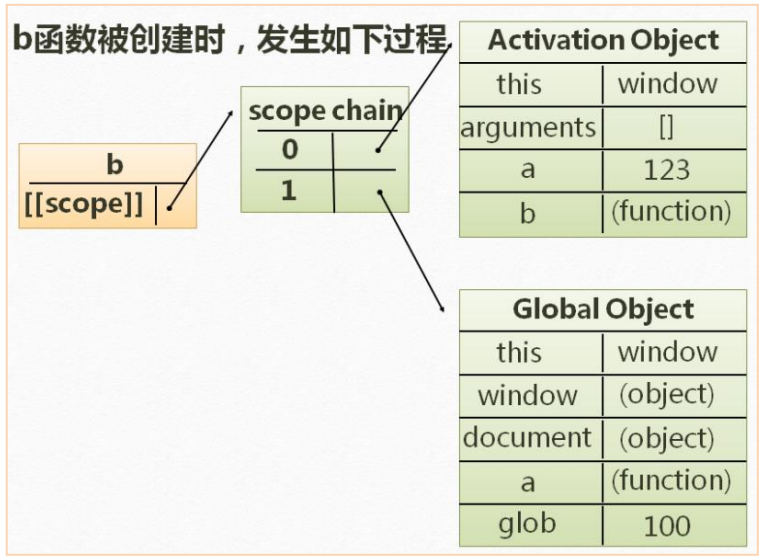
作用域精解

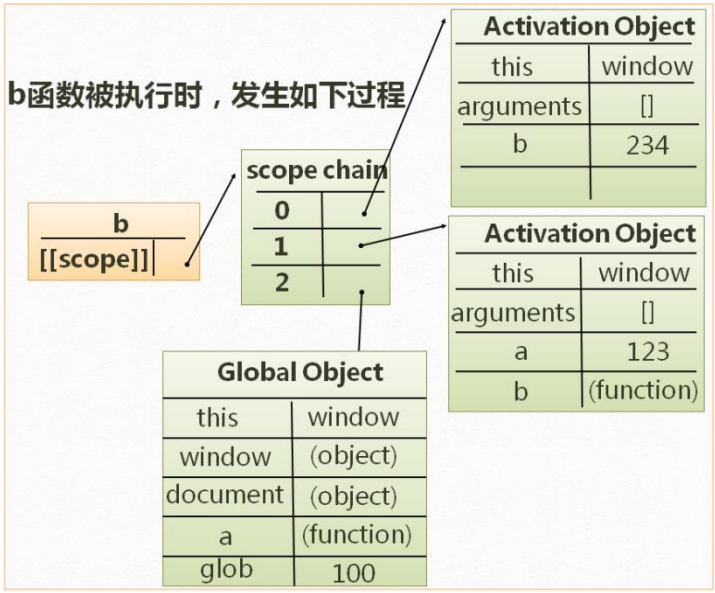
[[scope]]:每个 javascript 函数都是一个对象，对象中有些属性我们可以访问，但有些不可以，这些属性仅供 javascript 引擎存取，[[scope]]就是其中一个。[[scope]]指的就是我们所说的作用域,其中存储了运行期上下文的集合。
作用域链：[[scope]]中所存储的执行期上下文对象的集合，这个集合呈链式链接，我们把这种链式链接叫做作用域链。
运行期上下文:当函数在执行的前一刻，会创建一个称为执行期上下文的内部对象。一个执行期上下文定义了一个函数执行时的环境，函数每次执行时对应的执行上下文都是独一无二的，所以多次调用一个函数会导致创建多个执行上下文，当函数执行完毕，执行上下文被销毁。
查找变量 :在哪个函数里面查找变量，就从哪个函数作用域链的顶端依次向下查找。
函数类对象，我们能访问 test.name
test.[[scope]]隐式属性——作用域
例 function test (){
}
第一次执行 test(); → AO{} //AO 是用完就不要的
第二次执行 test(); → AO{} //这是另外的 AO
例 function a (){
　　function b (){
　　var bb = 234;
　　aa = 0;
　　}
　　var aa = 123;
　　b();
　　console.log(aa)
}
var glob = 100;
a();

0 是最顶端，1 是次顶端，查找顺序是从最顶端往下查









例

```
function a() {
    function b() {
        var bb = 234;
        aa = 0;
    }
    var aa = 123;
    b();
    console.log(aa);
}
var glob = 100;
a();
```

答案：0

理解过程：bb 的 AO 是拿到 aa 的 AO，就是同一个 AO，bb 只是引用了 aa 的 AO，GO 也都是同一个。function b(){}执行完，干掉的是 b 自己的 AO（销毁执行期上下文）（去掉连接线），下次 function b 被执行时，产生的是新的 b 的 AO。b 执行完只会销毁自己的 AO，不会销毁 a 的 AO。function a(){}执行完，会把 a 自己的 AO 销毁【会把 function b 也销毁】，只剩 GO（回归到 a 被定义的时候），等下次 function a 再次被执行时，会产生一个全新的 AO，里面有一个新的 b 函数。。。。。周而复始

例

```
function a() {

    function b () {

        function c() {

        }
        c();
    }
    b()

}
a();
```

理解过程

a 被定义 a.[[scope]] → 0 : GO{}

a 被执行 a.[[scope]] → 0 : aAO{}
1 : GO{}

b 被定义 b.[[scope]] → 0 : aAO{}
1 : GO{}

b 被执行 b.[[scope]] → 0 : bAO{}
1 : aAO{}
2 : GO{}

c 被定义 c.[[scope]] → 0 : bAO{}
1 : aAO{}
2 : GO{}

c 被执行 c.[[scope]] → 0 : cAO{}
1 : bAO{}
2 : aAO{}
3 : GO{}

当 c 执行完后，会干掉自己的 cAO，回到 c 被定义的状态，当 c 再被执行时，会生成一个新的 newcAO{}，其余都一样，因为基础都是 c 的被定义状态

c 被执行 c.[[scope]] → 0 : newcAO{}
1 : bAO{}
2 : aAO{}
3 : GO{}

如果 function a 不被执行，下面的 function b 和 function c 都是看不到的（也不会被执行，被折叠）。只有 function a 被执行，才能执行 function a 里面的内容

a();不执行，根本看不到 function a (){}里面的内容

闭包

当内部函数被保存到外部时，将会生成闭包。闭包会导致原有作用域链不释放，造成内存泄露。

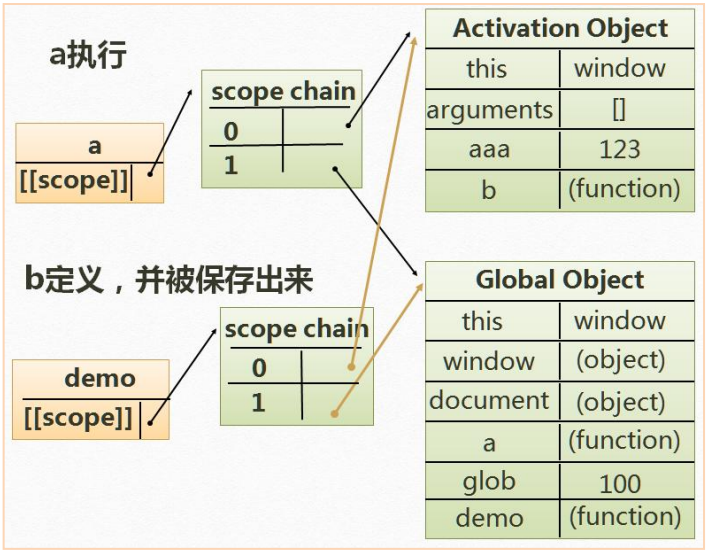
内存泄漏就是内存占用，内存被占用的越多，内存就变得越来越少了，就像内存被泄露了一样

例

return b 以后，就返回出去，再销毁 fn a

```
function a () {
    function b( ) {
        var bbb =  234;
        console.log(aaa);
    }
    var aaa = 123;
    return b;
}

var glob = 100;
var demo = a();
demo();
```

答案 123。因为没有 b();此时 b 还是被定义的状态，和 a 执行的状态是一样的。function a(){}是在 return b 之后才执行完，才销毁。return b 让 a 执行时的 AO 被保存在外面。



return b 是把 b（包括 a 的 AO）保存到外部了（放在全局）

当 a 执行完砍掉自己的 AO 时，b 依然可以访问到 a 的 AO(因为 return b)

但凡是内部的函数被保存到外部，一定生成闭包

例

```
function a() {

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

}

var demo = a();
demo();
demo();
```

答案 101,102　　理解过程

a 被执行 0 : a AO: num = 100;
1 : GO: demo = a();

b 被执行 0 : bAO :
1 : aAO : num = 100;
2 : GO: demo = a();

在第一次执行 function b 时，num ++就把 aAO 变成{num : 101}，当 function b 执行完毕时，剪断的是 bAO，而 aAO 不变，当执行 function a 的 return b 时就把 aAO,GO 都

存在了外部，执行完 a 销毁 scope 时去掉 a 的连接线，但是因为 return b 把 aAO,GO 存在了外部，所以依然还是可以访问值

在第二次执行 function b 时，aAO{num : 101}，在 num ++就是 102

例

```
function a() {
    var aa = 345;
    function b() {
        var bb = 234;
        function c() {
            var cc = 123;
        }
        c();
    }
    b();
}
a();
```

执行过程：

先执行 function a(){ var aa = 345;function b (){}b();}

想要执行完上面的 b();就需要执行完 function b(){var bb = 234; function c(){}c();}，

想要执行 c();就要先执行完 function c(){var cc =123;},

b();是一个复合语句，执行完 b 里面的每一句话，才能执行 b();

销毁顺序：

哪个先被执行完，哪个先被销毁

当 c 执行完，先销毁 c 自己的执行期的上下文，

当 c();执行完，那么 b 也执行完了，就销毁 b 的执行期上下文；

当 b();执行完，那么 a 也执行完了，就销毁 a 的执行期上下文

闭包的作用

一、实现公有变量

例函数累加器

```
function add() {
    var count = 0;
    function demo() {
        count ++;
        console.log(count);
    }
    return demo;
}

var counter = add();
counter();
counter();
counter();
counter();
counter();
counter();
counter();
```

每回调用 counter 就会在原有基础上加一次

二、可以做缓存（存储结构）

eg:eater。缓存是外部不可见的，但是确实有存储结构

例

```
function test() {
    var num = 100;
    function a () {
        num ++;
        console.log(num);
    }
    function b () {
        num --;
        console.log(num);
    }
    return [a,b];
}
var myArr = test();
myArr[0]();
myArr[1]();
```

答案 101 和 100，**思考**过程：说明两个用的是一个 AO

test doing test[[scope]] 0：testAO
1：GO

a defined a.[[scope]] 0 : testAO
1 : GO

b defined b.[[scope]] 0 : testAO
1 : GO

return[a, b]将 a 和 b 同时被定义的状态被保存出来了

当执行 myArr[0]();时

a doing a.[[scope]] 0 : aAO
1 : testAO
2 : GO

当执行 myArr[1]();时

b doing b.[[scope]] 0 : bAO
1 : a 运行后的 testAO
2 : GO

a 运行后的 testAO， 与 a doing 里面的 testAO 一模一样

a 和 b 连线的都是 test 环境，对应的一个闭包

function a 和 function b 是并列的，不过因为 function a 在前，所以先执行 num ++，在执行 num --

myArr[0]是数组第一位的意思，即 a，myArr[0]();就是执行函数 a 的意思；

myArr[1]是数组第二位的意思，即 b，myArr[1](); 就是执行函数 b 的意思

例缓存的应用，对象里面可以用属性和方法

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

var eater1 = eater();

eater1.push('banana');
eater1.eat();
```

答案 i am eating banana，eat 和 push 操作的是同一个 food

在 function eater(){里面的 food}就相当于一个隐式存储的机构

obj 对象里面是可以有 function 方法的,也可以有属性，方法就是函数的表现形式

三、可以实现封装，属性私有化。

eg: Person();

四、模块化开发，防止污染全局变量

立即执行函数

定义：此类函数没有声明，在一次执行过后即释放（被销毁）。适合做初始化工作。

针对初始化功能的函数：只想让它执行一次的函数

立即执行的函数也有参数，也有返回值，有预编译

例(function (){ //写成(function abc(){}())也调用不到

```
    var a = 123;
    var b = 234;
    console.log(a + b);
}())
```

例(function (a, b, c){

```
    console.log(a + b + c * 2);
}(1, 2, 3))
```

这一行里面的(1,2,3)是实参

例 var num = (function (a, b, c){

```
    var d = a + b + c * 2 – 2;
    return d;
}(1, 2, 3))
```

答案 num = 7

立即执行函数的两种写法

一(function (){}()); //在 W3C 建议使用这一种

二(function (){})();

只有表达式才能被执行符号执行

能被执行符号执行的表达式，这个函数的名字就会被自动忽略（放弃名字）

能被执行符号执行的表达式基本上就是立即执行函数

函数声明和函数表达式是两个东西，虽然都能定义函数

函数声明：function test ( ){}　　函数表达式：var test = function( ){}

例 function (){

```
  var a = 123;
}()
```

答案这是函数声明，不能执行，报语法错误，因为**只有表达式才能被执行符号执行**

例 function　test(){

```
    console.log( 'a' );
}
```

答案这也是函数声明

例 function (){

```
  var a = 123;
}
test();
```

答案 test()；就是表达式，所以能执行

例 var test = function (){

```
    console.log( 'a' );
}()
```

答案这是表达式，可以被执行，此时在控制台执行 test 的结果是 undefined，**这个函数的名字就会被放弃**

例+ function test(){

```
    console.log( 'a' );
}()
```

答案加了个“正“，在趋势上要把他转换成数字，就是表达式了，既然是表达式就能被执行，就会放弃名字，此时 console.log (test)，就会报错；这就是立即执行函数

同样放了正号，负号，！就会放弃函数名字，转换成表达式；但是*和/不行，&&||前面放东西也行

例 var test = function (){}

其中= function (){}把 function 赋到 test 里面去叫表达式，var test 是声明

在执行时，会放弃这个函数，储存到 test 里面储存引用，让这个 test 恢复到被声明的状态

例(function test(){console.log('a');})()
这个被()包起来的 function 函数声明变表达式了，就能被外面的最后的()执行
例(function test(){console.log('a');} ())
最外面的大括号是数学运算符号，是最先被执行，其余的括号都是有语法意义的，就把函数变表达式了
()也是数学执行符，能**打印** a，但是执行 test 就报错，所以干脆就不写 test
例 function test (a, b, c, d){
    console.log(a + b + c + d);
}(1, 2, 3, 4); //写成(1)也是这种效果
理论上不能执行，只写()就会被当成执行符，但是(1, 2, 3, 4);这样写暂时不会当成运算符，没意义，但是不会执行，也不报错。还能调用 test
例先定义一个 10 位数的数组，就是在 var arr = [function () {console.log(i);}有十个]并且把数组返回

```
function test() {

    var arr = [];
    for(var i = 0; i < 10; i ++) {
        arr[i] = function () {
            document.write(i + " ");
        }
    }
    return arr;
}

var myArr = test();
for(var j = 0; j < 10; j++) {
    myArr[j]();
}
```

```
function test (){  //定义个函数 test
    var arr = [];  //定义一个空数组
    for (var i = 0; i < 10; i ++){
//丰满空数组，让空数组添加十条数组，
每一条都是一个 function(){}
        arr[i] = function () { //随着 for 循环 i 变，
数组 i 也变，arr 每一次都等于一个全新的函数体
            document.write(i + " ");
        }
    }
    return arr;  //把 arr 返回到外部
}
var myArr = test();
for (var j = 0; j < 10; j ++){ //分别执行十个函数体，函数体里面定义了 document.write
    myArr[j]();
}
```

答案 10 10 10 10 10 10 10 10 10 10
第二个 for 是为了**打印**这个数组，麻烦写法 myArr[0](); myArr[1](); 。。。myArr[9]();
过程 for (var i = 0; i < 10; i ++){}执行了十次，产生了十个彼此独立的函数。并且把这十个函数放在数组里面去，还把数组返回了，这十个函数和 test 一起产生了一个闭包。
既然是闭包，那么访问 test 里面的变量时，实际上访问的是同一套，而 test 产生了 arr 和 i 变量（写在 for 循环里面的 i 变量），而这十个函数在外边要访问 i 变量，其实访问的是同一个 i。

什么时候访问的？在 test 执行完以后，在下面 for(j)访问的
第一个 i=0，转到 9 的时候，i ++变 10 终止 for 循环，结束的时候 i=10，结束之后把 return arr 返回，arr；
这十个函数都是为了**打印** i 的，在外部访问 i 的时候 i=10，所以**打印**的是 10

```
arr[i] = function () {
        document.write( i + " " );
}
```

理解过程：在这个函数体中，当 arr[0] 时，document.write(i)的 i 是不变的，还是 i，等函数保存到外部之后，等执行的时候，才会去找 i 的值。
这个赋值语句中 ,arr[0] = 函数;把一个函数体或者说是一个函数引用赋给数组的当前位，数组的当前位需要马上被索取出来的（数组现在是当前第几位，我们是知道的，因为这个是执行语句），当 for(var i = 0)时，arr[i]会变成 arr[0]，但是这个 i 跟函数体里面的 d.w(i+ "") 里面的 i 是没有关系的，因为函数体 function(){}不是现在执行，不会在意函数里面写的是什么，不是现在执行那么里面的 document.write 不会变成现实的值，不是现在执行就是函数引用（函数引用就是被折叠起来的，系统不知道里面写的是什么）
在执行 myArr[j]();的时候，系统才会读 document.write(i +"")里面的语句
在定义函数的时候是不看里面的，在执行的时候才看
例我们让上面这个变成**打印** 0,1,2,3,4,5,6,7,8,9，用立即执行函数解决

```
function test (){
    var arr = [];
    for (var i = 0; i < 10; i ++){
        (function (j) {
            arr[j] = function () {
                document.write( j + " " );
            }
        }(i));
    return arr;
}
var myArr = test();
for (var j = 0; j < 10; j ++){
    myArr[j]();
}
```

```
function test() {
    var arr = [];
    for(var i = 0; i < 10; i ++) {

        (function (j) {

            arr[j] = function () {
                document.write(j + " ");
            }

        }(i));

    }
    return arr;
}

var myArr = test();
for(var j = 0; j < 10; j++) {
    myArr[j]();
}
```

理解过程：相当于在 for 循环里面有十个立即执行函数 function(j){}
在第一圈 i 是 0，j 也是 0，function(){document.write(j + " ")}拿着 j=0，进行循环
的第二圈 i 是 1，又有了一个新的 j 是 1，反复循环
形成了十个立即执行函数，有十个 j 对应

例 for(var i = 0; i < 10; i ++){
    console.log(i);
}
答案 0,1,2,3,4,5,6,7,8,9
例 for(var i = 0; i < 10; i ++){
  (function(){
}())
}
中间 function 这个会执行 10 次

闭包的防范

闭包会导致多个执行函数共用一个公有变量，如果不是特殊需要，应尽量防止这种情况发生。

对象

1.用已学的知识点，描述一下你心目中的对象。

例

```
var mrDeng = {

    name : "MrDeng",
    age : 40,
    sex : "male",
    health : 100,
    smoke : function () {
        console.log('I am smoking ! cool!!!');
        mrDeng.health --;
    },
    drink : function () {
        console.log('I am drink');
        mrDeng.health ++;
    }
}
```

```
> mrDeng.health
< 100
> mrDeng.drink()
  I am drink
< undefined
> mrDeng.drink()
  I am drink
< undefined
> mrDeng.health
< 102
```

```
> mrDeng.smoke()
  I am smoking ! cool!!!
< undefined
> mrDeng.health
< 99
> mrDeng.drink()
  I am drink
< undefined
```

灰色的 undefined 是返回值，因为没有设置返回值，所以就是灰色的 undefined

.代表函数引用

改 mrDeng.health 为 this.health ,此处 this 指代的是自己 ,是第一人称 ,指的就是 mrDeng。因为 this 是在一个方法里面，所以指的这个方法。

```
smoke : function (){
    console.log( 'I am good' )
    this.health --;
},
```

2.属性的增、删、改、查

例在上面的基础上 mrDeng.wife = "xiaoliu"
也可以在控制台操作

```
> mrDeng.wife = "xiaoliu"
< "xiaoliu"
> mrDeng
< ▶Object {name: "MrDeng", age: 40, sex: "male",
  health: 100, wife: "xiaoliu"}
```

= "" 等号后面的引号里面需要有值才可以

```
> mrDeng
< ▶Object {name: "MrDeng", age: 40, sex: "female",
  health: 100}
> mrDeng.sex = "male"
< "male"
> mrDeng.sex
< "male"
> mrDeng.sex = "midmale"
< "midmale"
```

删除必须**借助 delete** mrDeng.sex

```
> delete mrDeng.sex
< true
> mrDeng.sex
< undefined
> mrDeng.abc
< undefined
>
```

例

```
var deng = {
    prepareWife : "xiaowang",
    name : "laodeng",
    sex : "male",
    gf : "xiaoliu",
    wife : "",
    divorce : function () {
        delete this.wife;
        this.gf = this.prepareWife;
    },
    getMarried : function () {
        this.wife = this.gf;
    },
    changePrepareWife : function (someone) {
        this.prepareWife = someone;
    }
}
```

```
> deng.getMarried()
< undefined
> deng.wife
< "xiaoliu"
> deng.divorce()
< undefined
> deng.wife
< undefined
```

```
> deng.getMarried()
< undefined
> deng.wife
< "xiaowang"
> deng.changePrepareWife('xiaozhang')
< undefined
```

```
> deng.divorce()
< undefined
> deng.wife
< undefined
> deng.getMarried()
< undefined
```

```
> deng
< ▶Object {prepareWife: "xiaozhang", name:
  "laodeng", sex: "male", gf: "xiaozhang", wife:
  "xiaozhang"}
```

3.对象的创建方法

(1) var obj = {}　　对象字面量/对象直接量 plainObject

(2)构造函数

1)系统自带的构造函数 Object()

new Object();Array();Number();Boolean();Date();

系统自带的构造函数 Object()可以批量生成对象，每一个对象都一样，但是彼此相互独立。

在 Object()前面加个 new，变成 new Object()的执行，就会真正的返回一个对象，通过 return 返回，拿变量接受。var obj = new Object();

var obj = new Object();和 var obj = {};这样写区别不大

例 var obj = new Object();
obj.name = 'abc';
obj.sex = "male";
双引号和单引号都是表示的字符串，写双引号也可以写单引号，但是为了跟后端 php 配合最好写单引号。如果要打印一个单个的引号，用正则表达式转义字符\
注意等号和冒号的用法 obj.say = function(){}　　var obj = { name : 'abc' }

2）自定义

Object.create(原型)方法

例 function Person(){}
Person 是可以随便写的，也是构造函数
构造函数跟函数结构上没有任何区别
例 var person1 = new person();

```
> person1
< ▶person {}
> person1.name = 123;
< 123
> person1
< ▼person
    name: 123
    ▶__proto__: Object
```

必须用 new 这个操作符，才能构造出对象

构造函数必须要按照大驼峰式命名规则，但凡是构造函数就要大写，例如 TheFirNa

例

```
function Car() {
    this.name = "BMW";
    this.height = "1400";
    this.lang = "4900";
    this.weight = 1000;
    this.health = 100;
    this.run = function () {
        this.health --;
    }
}

var car = new Car();
var car1 = new Car();
```

```
car.name = "Maserati";
car1.name = "Merz";
```

```
> car.run()
< undefined
> car1.health
< 100
> car.health
< 99
```

```
> car
< ▶Car {name: "Maserati", height: "1400", lang: "4900", weight: 1000}
> car1
< ▶Car {name: "Merz", height: "1400", lang: "4900", weight: 1000}
```

car1 和 car 是长得一样，但是是不同的两个 car。**方法名和对象名尽量不一样**
a 和 A 变量是两个变量，var car = new Car 里面 car 和 Car 是两个变量

例
这里的 color 可以和上面的重复

```
function Car(color) {
    this.color = color;
    this.name = "BMW";
    this.height = "1400";
    this.lang = "4900";
    this.weight = 1000;
    this.health = 100;
    this.run = function () {
        this.health --;
    }
}

var car = new Car('red');
var car1 = new Car('green');
```

//这里的color可以和上面的重复

通过参数，
使函数发生变化，变成自定义

例

```
function Student(name, age, sex) {

    this.name = name;
    this.age = age;
    this.sex = sex;
    this.grade = 2017;

}


var student = new Student('zhangsan', 18, 'male');
```

构造函数内部原理

前提必须要加 new，以下三步都是隐式的：
1.在函数体最前面隐式的加上 var this = {}　　空对象
2.执行 this.xxx = xxx;
3.隐式的返回 return this

例

```
function Student(name, age, sex) {
    //var this = {}; AO { this : {}}
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.grade = 2017;

}


var student = new Student('zhangsan', 18, 'male');
```

```
function Student(name, age, sex) {
    //var this = {
    //  name : ""
    //  age :
    //};
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.grade = 2017;
    //return this;
}

var student = new Student('zhangsan', 18, 'male');
```

例 function Person(name, height){
//**隐式的 var this = {}，下面正常执行 this**
this.name = name;
this.height = height;
this.say = funtion (){
console.log(this.say);
//**此处的 this 和外面的 this 不同**
}
// **隐式的 return this;**
}
console.log(new Person('xiaowang', 180).name);

```
function Person(name, height) {
    // var this = {}
    this.name = name;
    this.height = height;
    this.say = function () {
        console.log(this.say);
    }
    // return this;
}


console.log(new Person('xiaowang', 180).name);
```

例也可以这样显式的写出来
function Person (name, height){
var that = {};　　//显式写出来
that.name = name;
that.height = height;
return that;　　//显式写出来

例
```
function Person(name, height) {
    // var this = {}
    this.name = name;
    this.height = height;
    this.say = function () {
        console.log(this.say);
    }
    return 123;
    // return this;
}


var person = new Person('xiaowang',180);
var person1 = new Person('xiaozhang', 175);
```

```
> person
  ▶ Person {name: "xiaowang", height: 180}
> person1
  ▶ Person {name: "xiaozhang", height: 175}
>
```

答案现在的 peson 和 person1 都是 Object{}

如果 return 写成 return 123，会使 return 失效，如右上图

x 有 **new 了以后就不能返回原始值**，例如 123

包装类

new String();　　new Boolean();　　new Number();

var num =123;　→　原始值数字

只有原始值数字是原始值，**原始值不能有属性和方法**

属性和方法只有对象有，包括对象自己，数组，function

```
> num
  ▶ Number {[[PrimitiveValue]]: 123}
> num.abc = 'a'
  "a"
> num.abc
  "a"
> num
  ▶ Number {abc: "a", [[PrimitiveValue]]: 123}
```

var num = new number 123;　→　构造函数。是对象 123，不是原始值数字

```
> num
  ▶ Number {[[PrimitiveValue]]: 123}
> num * 2
  246
```

```
> var str = new String('abcd')
  undefined
> str.a = 'bcd'
  "bcd"
> str.a
  "bcd"
> str.sayValue = function ( ){ return this.a};
  function ( ){ return this.a}
```

```
> str.sayValue()
  "bcd"
```

字符串类型的对象

var num = new Nunber(123);　　//数字类型对象

var str = new String('abcd');　　//字符串类型对象

var bol = new Boolean('true');　　//布尔类型对象

```
> bol
  ▶ Boolean {[[PrimitiveValue]]: true}
```

undefined 和 null 不可以有属性

不能写成 undefined.abc = 123;会报错

```
var str = "abcd";
str.abc = 'a';
```

```
> str.abc
  undefined
```

```
var str = "abcd";
> str.length
  4
```

```
var str = "abcd";
str.abc = 'a';
```

```
> str.abc
  undefined
```

例原始值不可能有属性和方法，但经过了包装类(加隐式)可以调用一些属性与方法

var num = 4 ;

num.len = 3;

//系统隐式的加上 new Number(4).len = 3; 然后 delete

console.log(num.len);

//系统隐式的加上 new Number(4).len; 但是这个 new number 和上面的 new number 不是同一个，所以返回 undefined

而上面这些隐式的过程就是包装类

例 var str = "abcd" ;

str.length = 2;

//隐式的加上 new string('abcd').length = 2;　delete

console.log(str);

console.log(str.length);

答案是 abcd，4

例
```
var str = "abc";
str += 1;
var test = typeof(str);
if(test.length == 6) {
    test.sign = "typeof的返回结果可能为String";
}
console.log(test.sign);
```

str += 1;　　**//abc1**

var test = typeof(str);　　**//test == "string"，返回 string，string 长度是 6**

if(test.length == 6){

　test.sign = "typeof 的返回结果可能为 String" ;　　**//这是原始值,原始值要赋属性值需要调用包装类，赋了跟没赋值是一样的，new String(test).sign=' xxx' ;**

}

conlogo.log(test.sign);　　**//new String(test).sign**

答案 undefined

例

```
// 2.分析下面的JavaScript代码段:
// function employee(name,code) {
//     this.name="wangli";
//     this.code="A001";
// }
// var newemp = new employee("zhangming",'A002');
// document.write("雇员姓名:"+ newemp.name+ "<br>");
// document.write("雇员代号:"+ newemp.code +"<br>");
// 输出的结果是().(选择一项)

// A.  雇员姓名:wangli 雇员代码:A001
// B.  雇员姓名: zhangming 雇员代码: A002
// C.  雇员姓名: null, 雇员代码: null
// D.  代码有错误, 无输出结果
```

答案 A，里面并没有用参数被写死了，传参不成功

例

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

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

答案 1，2，1

例

```
var x = 1, y = z = 0;

function add(n) {
    return n = n + 1;
}
y = add(x);
function add(n) {
    return n = n + 3;
}
z = add(x);
//x y z
```

答案 1，4，4

同一个函数，后面会覆盖前面的

fn add 会提升到前面

例下面代码中 console.log 的结果是[1,2,3,4,5]

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

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

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

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

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

```
请问以下表达式的结果是什么？：
parseInt(3, 8)
parseInt(3, 2)
parseInt(3, 0)

3, 3, 3
3, 3, NaN
3, NaN, NaN
other
```

答案 ACD（枚举后面有清晰版）

例：请问以下表达式的结果是什么？

parseInt(3, 8)

parseInt(3, 2)

parseInt(3, 0)

答案选 3 或 4，值为 3，NaN，3（有的浏览器遇到 0 是报 NaN）

例：以下哪些是 JavaScript 语言 typeof 可能返回的结果：

A.string　B.array　C.object　D.null

答案：A

例：看看下面 alert 的结果是什么？

```
function b (x, y, a) {
    arguments[ 2 ] = 10;
    alert( a );
}
b(1, 2, 3);
```

如果函数体改成下面，结果又会是什么？

```
a = 10；
alert(arguments[ 2 ]);
```

答案 10，10

例

8.写一个方法，求一个字符串的字节长度。（提示：字符串有一个方法 charCodeAt(); 一个中文占两个字节，一个英文占一个字节.

定义和用法

charCodeAt() 方法可返回指定位置的字符的 Unicode 编码。这个返回值是 0 - 65535 之间的整数。(当返回值是<=255 时，为英文, 当返回值 > 255时为中文)

语法

stringObject.charCodeAt(index)

eg:

```
<script type="javascript/text">
        var str="Hello world!"
        document.write(str.charCodeAt(1));//输出101
</script>
)
```

```
var str = "hello world 邓哥身体好";

str.charCodeAt(0);
```

```
str.charCodeAt(0);
104
```

```
var str = "hello world 邓哥身体好";

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

小于等于 255 是一个字节，

大于是两个字节

返回值大于 255 是中文

右边是两种方法→

Unicode 编码涵盖 asc 码

```
var str = "abcd邓";

function bytesLength(str) {
    // var count = str.length;
    // for(var i = 0; i < str.length; i++) {
    //     if(str.charCodeAt(i) > 255) {
    //         count ++;
    //     }
    // }
    var count = 0;
    for(var i = 0; i < str.length; i++) {
        if(str.charCodeAt(i) > 255) {
            count += 2;
        }else {
            count ++;
        }
    }
    return count;

}
```

原型

1.定义：原型是 function 对象的一个属性，它定义了构造函数制造出的对象的公共祖先。通过该构造函数产生的对象，可以继承该原型的属性和方法。原型也是对象。
2.利用原型特点和概念，可以提取共有属性。
3.对象属性的增删和原型上属性增删改查。
4.对象如何查看原型 ==> 隐式属性 __proto__。
5.对象如何查看对象的构造函数 ==> constructor。

例 person.prototype　　//原型（描述一种继承关系），出生时就被定义好了
person.prototype ={}　　//是祖先

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.name = "hehe";
function Person() {


}
var person = new Person();
```

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.name = "hehe";
function Person() {


}
var person = new Person();
var person1 = new Person();
```

```
> person
< ▶Person {}
> person.name
< "hehe"
```

右上的 person 和 person1 都有一个共有的祖先 Person.prototype

例

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.LastName = "Deng";
Person.prototype.say = function () {
    console.log('hehe');
}
function Person() {

    this.LastName = "Ji";

}
var person = new Person();
var person1 = new Person();
```

```
> person.LastName
< "Ji"
>
```

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.LastName = "Deng";
Person.prototype.say = function () {
    console.log('hehe');
}
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;

}
var person = new Person('xuming', 35, 'male');
```

```
> person
< ▶Person {name: "xuming", age: 35, sex: "male"}
> person.name
< "xuming"
> person.LastName
< "Deng"
```

自己身上有属性，原型上也有属性，取近的，用自己的

例

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
function Car(color, owner) {

    this.owner = owner;
    this.carName = "BMW";
    this.height = 1400;
    this.lang = 4900;
    this.color = color;

}

var car = new Car('red', 'prof.ji');
```

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Car.prototype.height = 1400;
Car.prototype.lang = 4900;
Car.prototype.carName = "BMW";
function Car(color, owner) {

    this.owner = owner;
    this.color = color;

}

var car = new Car('red', 'prof.ji');
var car1 = new Car('green', 'laodeng');
```

```
> car1.carName
< "BMW"
> car.carName
< "BMW"
```

将左上的共有的东西提取出来放在原型里面，如右上图

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.lastName = "Deng";
function Person(name) {
    this.name = name;
}
var person = new Person('xuming');

person.lastName
```

```
> person.lastName = "James";
< "James"
> person
< ▶Person {name: "xuming", lastName: "James"}
> person.lastName
< "James"
```

上面通过对象（后代）改原型（祖先）是不行的，在对象里面修改，只作用给自己
改原型都不行，增加肯定也不行。对象可以删除属性

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Person.prototype.height = 1400;
Person.prototype.lang = 4900;
Person.prototype.carName = "BMW";
function Car() {
}
var person = new Person('xuming');
```

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先

Car.prototype = {
    height : 1400,
    lang : 4900,
    carName : "BMW"
}

function Car() {
}
var car = new Car();
```

```
> delete person.lastName
< true
> person.lastName
< "Deng"
```

左上的**简化**写法见右上　　// construnctor 是构造的意思（隐式是浅粉色）

```
function Car() {
}
var car = new Car();
```

```
> car.constructor
< function Car() {
  }
```

```
> Car.prototype
< ▼Object
    ▶constructor: Car()
    ▶__proto__: Object
```

浅粉色是系统帮你设置的，深紫色是自己设置的

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先
Car.prototype.abc = '123';
function Car() {
}
var car = new Car();
```

```
> Car.prototype
< ▼Object
    abc: "123"
    ▶constructor: Car()
    ▶__prot .constructor
```

在原型内部自带 constructor，指的是 Car。通过 constructor 能找的谁构造的自己

例

```
// Person.prototype     -- 原型
// Person.prototype = {}    是祖先

function Person() {

}

Car.prototype = {
    constructor : Person
}

function Car() {
}
var car = new Car();
```

```
> car.constructor
< function Person() {
      }
```

constructor 可以被人工手动更改

例

```
Person.prototype.name = 'abc';
function Person() {
    //var this = {
    //      __proto__ : Person.prototype
    //};

}


var person = new Person();
```

```
> person
< ▼Person
    ▼__proto__: Object
      ▶constructor: Person()
      ▶__proto__: Object
```

```
> person.__proto__
< ▼Object
    ▶constructor: Person()
     name: "abc"
    ▶__proto__: Object
```

浅粉色的__proto__是系统的隐式的属性,前面两个_后面两个_,可以修改 尽量不改。
在开发的时候，如果很私人可以写成_private，告诉同事别动。
上面的__proto__放的是原型。__proto__存的对象的原型
上面的 var this ={__proto__:person.prototype};这个对象并不是空的，这个 proto，当你访问这个对象的属性时，如果对象没有这个属性，那么就会访问 proto 索引，看看有没有。有一个连接的关系，原型和自己连接到一起

例

```
Person.prototype.name = 'abc';
function Person() {
    //var this = {
    //
    //   __proto__ : Person.prototype
    //}

}

var obj = {
    name : "sunny"
}

var person = new Person();
person.__proto__ = obj;
```

```
> person.__proto__
< ▶Object {name: "abc"}
```

```
> person.__proto__ = obj
< ▶Object {name: "sunny"}
> person.name
< "sunny"
```

Person 的原型是可以被修改的

例

```
Person.prototype.name = 'sunny';

function Person() {

}

var person = new Person();

Person.prototype.name = 'cherry';
```

```
> person.name
< "cherry"
```

例

```
Person.prototype.name = 'sunny';

function Person() {

}
Person.prototype.name = "cherry";
var person = new Person();
```

```
> person.name
< "cherry"
```

例

```
Person.prototype.name = 'sunny';

function Person() {
    //var this = {__proto__ : Person.prototype}
}


var person = new Person();

Person.prototype = {
    name : 'cherry'
}
```

```
> person.name
< "sunny"
```

Person.prototype.name 这种.的写法是在原有的基础上把值改了。改的是属性，也就是房间里面的东西。
而 Person.prototype={name:’ cherry’ }是把原型改了，换了新的对象。改了个房间。
上面在 new 的时候 var this = {__proto__:Person.prototype}里面的指向 Person，此时 Person.prototype 与__proto__指向的是一个空间，把他返回给 var person。
先 new 再 Person.prototype={name:’ cherry’ }已经晚了
在 Person.prototype={name:’ cherry’ }时，Person.prototype 空间改了，但是__proto 指向的空间不变。
上面的步骤实际上是→

```
Person.prototype = {name : "a"};
__proto__ = Person.prototype;
Person.prototype = {name : "b"};
```

例

```
var obj = {name : "a"};
var obj1 = obj;
obj = {name : "b"};
```

```
> obj1
< ▶Object {name: "a"}
> obj
< ▶Object {name: "b"}
```

例

```
Person.prototype.name = 'sunny';

function Person() {
    //var this = {__proto__ : Person.prototype}
}
Person.prototype = {
    name : 'cherry'
}

var person = new Person();
```

上面这种**思考**过程：程序执行顺序

1.先把 function Person(){}在预编译的过程中提到最上面

2.再执行 Person.prototype.name = 'sunny' 这一样行

3.再执行 Person.prototype = {name:' cherry' }

4.最后执行 var person = new Person();执行到 new 的时候，才会发生//var this ={__proto__:Person.prototype}

5.下面的把上面的覆盖了

6.**答案**是 cherry

例

```
function Person() {

}
```

```
> Person.prototype
Object
    constructor: Person()
    __proto__: Object
```

这说明原型里面有原型

例

原型链

```
Grand.prototype.lastName = "Deng";
function Grand() {

}

var grand = new Grand();

Father.prototype = grand;
function Father() {
    this.name = 'xuming';
}

var father = new Father();

Son.prototype = father;
function Son () {
    this.hobbit = "smoke";
}
var son = new Son();
```

```
> son.hobbit
"smoke"
> son.name
"xuming"
```

```
> son.lastName
"Deng"
```

```
> Grand.prototype
Object
    constructor: Grand()
    lastName: "Deng"
    __proto__: Object
        __defineGetter__:
        __defineSetter__:
        __lookupGetter__:
```

```
> Object.prototype
Object {}
> Object.prototype.__proto__
null
```

执行 son.toString //**返回** function toString( ){ [native code] } ,这里返回的是原型链终端的 toString

Grand.prototype.__proto__ = Object.prototype // Object.prototype 是原型链的终端

原型链

1、如何构成原型链?（见上一个例子）

2、原型链上属性的增删改查

原型链上的增删改查和原型基本上是一致的。只有本人有的权限，子孙是没有的。

3、谁调用的方法内部 this 就是谁-原型案例

4、**绝大多数**对象的最终都会继承自 Object.prototype

5、Object.create(原型);

6、原型方法上的重写

例

```
Grand.prototype.lastName = "Deng";
function Grand() {

}

var grand = new Grand();

Father.prototype = grand;
function Father() {
    this.name = 'xuming';
    this.fortune = {
        card1 : 'visa'
    }
}

var father = new Father();

Son.prototype = father;
function Son () {
    this.hobbit = "smoke";
}
var son = new Son();
```

```
> son.fortune
Object {card1: "visa"}
> son.fortune = 200;
200
> son
Son {hobbit: "smoke", fortune: 200}
> father.fortune
Object {card1: "visa"}
```

```
> son.fortune.card2 = 'master'
"master"
> son
Son {hobbit: "smoke"}
> father
Grand {name: "xuming", fortune: Object}
```

```
Grand
    fortune: Object
        card1: "visa"
        card2: "master"
        __proto__: Object
    name: "xuming"
    __proto__: Grand
```

son.fortune.card2=' master' 这种改，这是引用值自己的修改。属于 fortune.name 给自己修改，这是一种调用方法的修改

例

```
Grand.prototype.lastName = "Deng";
function Grand() {

}

var grand = new Grand();
```

```
function Father() {
    this.name = 'xuming';
    this.fortune = {
        card1 : 'visa'
    };
    this.num = 100;
}

var father = new Father();

Son.prototype = father;
function Son () {
    this.hobbit = "smoke";
}
var son = new Son();
```

```
> son.num ++;
100
> father.num
100
> son.num
101
```

son.num++是 son.num=son.num+1 是先把父级的取过来再赋值+1，所以爹的没变

例

```
Person.prototype = {
    name : "a",
    sayName : function () {
        console.log(this.name);
    }
}

function Person () {

}

var person = new Person();
```

```
> person.sayName()
  a
< undefined
```

console.log(this.name); //如果写成 name 就会错，没有这个变量

例

```
Person.prototype = {
    name : "a",
    sayName : function () {
        console.log(this.name);
    }
}

function Person () {
    this.name = "b";
}

var person = new Person();
```

```
> person.sayName()
  b
< undefined
```

```
> Person.prototype.sayName()
  a
< undefined
```

//a.sayName ( )方法调用 ,就是 say.Name 里面的 this 指向 ,是谁调用的这个方法 ,this 就指向谁

例

```
Person.prototype = {
    height : 100
}

function Person () {
    this.eat = function () {
        this.height ++;
    }
}

var person = new Person();
```

```
> person.eat()
< undefined
> person
< ▶ Person {height: 101}
> person.__proto__
< ▶ Object {height: 100}
```

this.height ++; //这后面默认有一个 return undefined

例

```
Person.prototype = {
    height : 100
}

function Person () {
    this.eat = function () {
        this.height ++;
        return 123;
    }
}

var person = new Person();
```

```
> person.eat()
< 123
```

例 var obj = { };也是有原型的

var obj = { };与 var obj1 = new Object( );效果是一样的

写 var obj = { }; 系统会在内部来一个 new Object( );

obj1.__proto__ → Object.ptototype;

但是在构造对象时，能用对象自变量 var obj = { };就不要用 var obj1 = new Object( );

例

```
Person.prototype = {} --> Object.prototype
function Person() {

}
```

对像自变量的原型就是 Object.ptototype;

Object.create(原型);

//var obj = Object.create(原型);

Object.create 也能创建对象。var obj = Object.create（这里必须要有原型）

例

```
// var obj = Object.create(原型);
var obj = {name : "sunny", age : 123};
var obj1 = Object.create(obj);
```

```
> obj1
< ▶ Object {}
```

```
> obj1.name
< "sunny"
```

例

```
// var obj = Object.create(原型);
Person.prototype.name = "sunny";
function Person() {

}
var person = Object.create(Person.prototype);
```

绝大多数对象的最终都会继承自 Object.prototype

例 html 里面没有添加任何东西 这样就报错

```
> Object.create()
✖ ▶ Uncaught TypeError: Object prototype may only be an Object or null: undefined
      at Function.create (<anonymous>)
      at <anonymous>:1:8
                                                    VM2870:1
```

例 html 里面没有添加任何东西 只在控制台加上 null

```
> Object.create(null)
< ▼ Object
    No Properties
```

例

```
// var obj = Object.create(原型);

var obj = Object.create(null);
```

```
> obj.toString()
✖ ▶ Uncaught TypeError: obj.toString is not a function
      at <anonymous>:1:5
```

```
> obj.name = 123;
< 123
> obj
< ▼ Object
    name: 123
```

```
> obj.__proto__ = {name : "sunny"}
< ▶ Object {name: "sunny"}
> obj
< ▶ Object {}
> obj.name
< undefined
```

原型是隐式的内部属性，你加是没有用的

例

```
// var obj = Object.create(原型);

var obj = Object.create(123);
```

```
▶ Uncaught TypeError: Object                    demo.htm
prototype may only be an Object or null: 123
    at Function.create (<anonymous>)
    at demo.html:16
```

Object.create()在括号里面只能放 null 或者 Object，其余会报错

例 undefined 和 null 没有原型，也就不可能有 toString 方法

```
> undefined.toString()
▶ Uncaught TypeError: Cannot read
property 'toString' of undefined
    at <anonymous>:1:10
```

```
> null.toString()
▶ Uncaught TypeError: Cannot read
property 'toString' of null
    at <anonymous>:1:5
```

例下面 123.toString 首先会识别成浮点型，所以在后面直接加.toString 是不行的

```
> true.toString()
"true"
```

```
> 123.toString();
Uncaught SyntaxError: Invalid or
unexpected token
```

```
> var num = 123;
undefined
> num.toString()
"123"
```

```
> var obj = {};
undefined
> obj.toString();
"[object Object]"
```

数字想用 toString 方法，要经过包装类包装 new Number(num)然后. toString

例加深上面的理解

```
var num = 123;
// num.toString(); -- > new Number(num).toString();
Number.prototype.toString = function () {

}

// Number.prototype.__proto__ = Object.prototype
```

而 new Number(num). toString 的原型是 Nunber.prototype ,而 Nunber.prototype 上面有一个.toString 方法，Nunber.prototype 也有原型 Nunber.prototype.__proto__，原型是 Object.prototype

假如 new Number 上面的 prototype 上面有这个 toString，那么就不用 Object.prototype 的 toString。而这个 number 上面有这个 toString。

然后 number 上面的 toString 调用的是自己重写的 toString。

原型上有这个方法，我自己又写了一个和原型上同一名字，但不同功能的方法，叫做重写（同一名字的函数，不同重写方式）

通过返回值，形参列表不同传参

同样的名实现不同功能的，就是重写

```
var num = 123;
// num.toString(); -- > new Number(num).toString();
Number.prototype.toString = function () {


}

// Number.prototype.__proto__ = Object.prototype


// Object.prototype.toString = function () {

// }
```

例

```
// var obj = Object.create(原型);

// Object.prototype.toString = function () {

// }
Person.prototype = {

}
function Person () {

}
var person = new Person();
```

```
> person.toString()
"[object Object]"
```

和原型链上终端方法名字一样，但实现不同的功能，叫做方法的重写。也就是覆盖

```
// var obj = Object.create(原型);

// Object.prototype.toString = function () {

// }
Person.prototype = {
    toString : function () {
        return 'hehe';
    }
}
function Person () {

}
var person = new Person();
```

```
> person.toString()
"hehe"
```

下面这个也是重写

```
// var obj = Object.create(原型);

Object.prototype.toString = function () {
    return 'haha';
}
Person.prototype = {

}
function Person () {

}
var person = new Person();
```

```
> person.toString()
"haha"
```

例让 object 上面的 toString 重写了。

所以 num.toString()调用的是 number. prototype.toString。

```
// Object.prototype.toString
// Number.prototype.toString
// Array.prototype.toString
// Boolean.prototype.toString
// String.prototype.toString
```

```
> var num = 123;
undefined
> num.toString()
"123"
```

如果调用的是 object. prototype.toString 结果会不一样。

```
> Object.prototype.toString.call(123)
"[object Number]"
```

```
> Object.prototype.toString.call(true)
"[object Boolean]"
```

例

```
Number.prototype.toString = function () {
    return '老邓身体好';
}
```

```
> var num = 123;
< undefined
> num.toString()
< "老邓身体好"
```

例

```
var obj = 123;
document.write(obj);
```

123

例

```
var obj = {};
document.write(obj);
```

[object Object]

例

```
var obj = Object.create(null);
document.write(obj);
```

```
▶ Uncaught TypeError: Cannot convert object to primitive value
    at demo.html:20
```

document.write 会隐式的调用 toString 方法，其实打印的是 toString 的结果

```
var obj = Object.create(null);
document.write(obj.toString());
```

没有原型就不能 toString

例

```
var obj = Object.create(null);
obj.toString = function () {
    return '老邓身体好';
}
document.write(obj);
```

file:///Users

老邓身体好

上面这个例子表示：我要打印的是 obj，实际上打印出来的是 toString 方法，也证明了 document.write 调用的是 toString 方法

call/apply

作用，改变 this 指向。

区别，后面传的参数形式不同。

toFixed 是保留两位有效数字

例有个 bug，在控制台 0.14*100

出现 14.0000000000002，是 js 开发精度不准

```
> 0.14 * 100
< 14.000000000000002
```

例向上取整 Math.ceil(123.234)

答案 124

例向下取整 Math.floor(123.999)

答案 123

```
> Math.ceil(123.234)
< 124
> Math.floor(123.99999)
< 123
```

例 Math.random()是产生一个 0 到 1 区间的开区间 随机数

例

```
for(var i = 0; i < 10; i ++) {
    var num = Math.random().toFixed(2) * 100;
    console.log(num);
}
```

```
72
11
37
70
28.000000000000004
99
72
98
40
16
```

所以一般在这种情况下，我们不用 toFixed，因为精度不准确

例用这种方法取整更好，就不会精度不准确

```
for(var i = 0; i < 10; i ++) {
    var num = Math.floor(Math.random() * 100);
    console.log(num);
}
```

例

```
10000000000000000001 + 10000000000000000001
20000000000000000000
```

```
> 0.1000000000000001 + 0.1000000000000001
< 0.2000000000000002
> 0.10000000000000001 + 0.10000000000000001
< 0.2
```

```
> 0.000000000000001 + 0.000000000000001
< 2e-15
```

注意：之前 js 在小数点后面最多能容纳 15-17 位，但是升级后能用科学计数法表示如 2e-17（是 2 乘以 10 的负 17 次方）

而小数点前面只能容纳 16 位的运算，**可正常计算的范围是小数点前后 16 位**

例 任何一个方法都可以.call

.call 才是一个方法执行的真实面目

```
function Person(name, age) {
    this.name = name;
    this.age = age
}

var person = new Person('deng', 100);
var obj = {
}
Person.call();
```

```
function Person(name, age) {
    //this == obj
    this.name = name;
    this.age = age
}

var person = new Person('deng', 100);

var obj = {
}
Person.call(obj, 'cheng', 300);

// test() --->   test.call();
```

直接执行 Person.call ( )和 Person ( )没有区别

Person.call( );括号里面可以传东西

```
> obj
< ▶ Object {name: "cheng", age: 300}
```

如果 Person.call(obj);里面的 call 让 person 所有的 this 都变成 obj

不 new 的话，this 默认指向 window。call 的使用必须要 new

call 的第一位参数用于改变 this 指向，第二位实参（对应第一个形参）及以后的参数都当做正常的实参，传到形参里面去

借用别人的方法，实现自己的功能。

例写 test（ ）和写 test.call（ ）是一样的，→

```
function test() {

}

// test() --->   test.call();
```

例

如右图

两个人的需求有重复部分

```
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}

function Student(name, age, sex, tel, grade) {
    this.name = name;
    this.age = age;
    this.sex = sex;
    this.tel = tel;
    this.grade = grade;
}

var student = new Student('sunny', 123, 'male', 139, 2017);
```

call 改变 this 指向，借用别人的函数，实现自己的功能。
只能在你的需求完全涵盖别人的时候才能使用
如果不想要 age 这个，就不能使用这种方法

```
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}

function Student(name, age, sex, tel, grade) {
    //var this = {name : "", age : "", sex : ""}
    Person.call(this, name, age, sex);
    this.tel = tel;
    this.grade = grade;

}

var student = new Student('sunny', 123, 'male', 139, 2017);
```

Person.call(this, name, age, sex);里面的 this 现在是 new 了以后的 var this={}
利用 Person 方法，实现了 Student 自己的封装
例 function Wheel (wheelSize,

```
function Wheel(wheelSize, style) {
    this.style = style;
    this.wheelSize = wheelSize;
}

function Sit(c, sitColor) {
    this.c = c;
    this.sitColor = sitColor;
}
```

```
function Model(height, width, len) {
    this.height = height;
    this.width = width;
    this.len = len;
}
```

```
> car
< ▼Car
    c: "真皮"
    height: 1800
    len: 4900
    sitColor: "red"
    style: "花里胡哨的"
    wheelSize: 100
    width: 1900
  ▶__proto__: Object
```

```
function Car(wheelSize, style, c, sitColor, height, width, len)
    {
    Wheel.call(this, wheelSize, style);
    Sit.call(this, c, sitColor);
    Model.call(this, height, width, len);
}
var car = new Car(100, '花里胡哨的', '真皮', 'red', 1800, 1900,
    4900);
```

apply 也是改变 this 指向的，只是传参列表不同，第一位也是改变 this 指向的人，第二位，apply 只能传一个实参，而且必须传数组 argunments
call 需要把实参按照形参的个数传进去
new 以后才有意义

```
Wheel.apply(this, [wheelSize, style]);
Sit.apply(this, [c, sitColor]);
Model.apply(this, [height, width, len]);
```

继承发展史

1.传统形式 ==> 原型链 //问题：过多的继承了没用的属性

```
Grand.prototype.lastName = "Ji";
function Grand() {

}

var grand = new Grand();

Father.prototype = grand;
function Father() {
    this.name = 'hehe';
}

var father = new Father();

Son.prototype = father;
function Son (){

}

var son = new Son();
```

2.借用构造函数 ==>利用 call、apply 所以不算标准的继承模式
1）不能继承借用构造函数的原型
2）每次构造函数都要多走一个函数 ==>浪费效率
this 放进去的前提，这个函数必须是 new 来的

```
function Person(name, age, sex) {
    this.name = name;
    this.age = age;
    this.sex = sex;
}

function Student (name, age, sex, grade) {
    Person.call(this, name, age, sex);
    this.grade = grade;
}

var student = new Student();
```

3.共享原型（较好的继承方法）不能随便改动自己的原型
Son.prototype=Father.prototype
让 son 和 father 共享原型

```
> son.lastName
< "Deng"
```

```
> father.lastName
< "Deng"
```

```
Father.prototype.lastName = "Deng";
function Father() {


}

function Son() {

}

Son.prototype = Father.prototype
var son = new Son();
var father = new Father();
```

例可以用上面的方式封装函数，实现一个继承

extend 和 inherit 都是继承的意思。

inherit 是 css 的一个值，也是继承的意思。

文字类属性都有一个传递的特性：子元素没有设置文字类属性，子元素默认继承父元素的属性。　　font-size:inheit;我没有就继承父亲的

在 inherit（Target，Origin）里面传进去的值是构造函数，需要大驼峰式书写，origin 是原始的意思，让 target（目标）继承 origin

```
Father.prototype.lastName = "Deng";
function Father() {


}

function Son() {

}
```

```
function inherit(Target, Origin) {
    Target.prototype = Origin.prototype;
}

inherit(Son, Father);
var son = new Son();
```

```
> son.lastName
<- "Deng"
```

上面这种方式让 son 里面有了 father 原型的属性

应该是先 inherit 继承，后使用

下面这种方式就是先继承了，后改变原型已经晚了，因为他继承的还是原来的空间

```
Father.prototype.lastName = "Deng";
function Father() {
}
function Son() {
}
function inherit(Target, Origin) {
    Target.prototype = Origin.prototype;
}

var son = new Son();
inherit(Son, Father);
```

```
> son.lastName
<- undefined
```

例下面这种写法，son.prototype 和 father.prototype 指向的是一个房间，改 son 就改了 father。我们希望 son 用的 father 的原型，但是改变 son 自己的属性不影响 father

```
Father.prototype.lastName = "Deng";
function Father() {
}
function Son() {
}
function inherit(Target, Origin) {
    Target.prototype = Origin.prototype;
}
inherit(Son, Father);

Son.prototype.sex = "male";


var son = new Son();
var father = new Father();
```

```
> son.sex
<- "male"
```

```
> father.sex
<- "male"
```

4.圣杯模式

圣杯模式是在方法三的共有原型，但是在共有原型的基础上有改变。

共享原型是：son.prototype=father.prototype

圣杯模式是：另外加个构造函数 function F（）{}当做中间层，然后让 F 和 father 共有一个原型 F.prototype=father.prototype，然后 son.prototype = new F（）；使用原型链形成了继承关系，现在改 son.prototype 就不会影响 father.prototype

```
//                Father.prototype
//
//                        function F() {}
//                        F.prototype = Father.prototype
//
//                        Son.prototype = new F();
//
// Father                                 Son
```

```
function inherit(Target, Origin) {
    function F() {};
    F.prototype = Origin.prototype;
    Target.prototype = new F();
}
Father.prototype.lastName = "Deng";
function Father() {
}
function Son() {
}

inherit(Son, Father);
var son = new Son();
var father = new Father();
```

```
> father.lastName
<- "Deng"
```

```
> son.lastName
<- "Deng"
```

```
> Son.prototype.sex = "male";
<- "male"
> son.sex
<- "male"
> father.sex
<- undefined
> Father.prototype
<- ▶Object {lastName: "Deng"}
```

son.__proto__ --> new F().__proto__ --> Father.prototype

例原型上默认有个 constructor

constructor 默认指向他的构造函数

son. constructor 应该指向 Son

```
> son.constructor
<- function Father() {
    }
```

指向 father 就是混乱了

所以要指一下　==>

我们希望我们构造出的对象，能找到自己的超类，超级父级（究竟继承自谁）应该起名为 super 但这个是保留字，我们就以 uber

```
function inherit(Target, Origin) {
function F() {};
    F.prototype = Origin.prototype;
    Target.prototype = new F();
    Target.prototype.constuctor = Target;
    Target.prototype.uber = Origin.prototype;
}
Father.prototype.lastName = "Deng";
function Father() {
}
function Son() {
}

inherit(Son, Father);
var son = new Son();
var father = new Father();
```

例：左下这种方法就不好使了，相当于右下。还是原型指向有问题，new 的时候用的是原来的原型，再 F.prototype =father.prototype 没用，son.prototype 没发生改变

```
function inherit(Target, Origin) {
    function F() {};
    Target.prototype = new F();
    F.prototype = Origin.prototype;
    Target.prototype.constuctor = Target;
    Target.prototype.uber = Origin.prototype;
}

Father.prototype.lastName = "Deng";
function Father() {
}
function Son() {
}

inherit(Son, Father);
var son = new Son();
var father = new Father();
```

```
function F() {}
Son.prototype = new F();


F.prototype = Father.prototype
```

例，在雅虎时代，封装了 YUI3 库来解决方法三的不足，与圣杯模式相似。现在不用 YUI3 库，现在用 jq

```
function inherit(Target, Origin) {
    function F() {};
    F.prototype = Origin.prototype;
    Target.prototype = new F();
    Target.prototype.constuctor = Target;
    Target.prototype.uber = Origin.prototype;
}

var inherit = (function () {
    var F = function () {};
    return function (Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
        Target.prototype.constuctor = Target;
        Target.prototype.uber = Origin.prototype;
    }
}());
```

上面的 var inherit 与右边的 var inherit 是一样的意思
建议写上面的这种

```
var inherit = (function () {
    var F = function () {};
    function demo(Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
        Target.prototype.constuctor = Target;
        Target.prototype.uber = Origin.prototype;
    }
    return demo;
}());
```

联系到闭包作用：可以实现封装，属性私有化。

例为什么在外部执行的 divorce 能用内部的变量？能换成 xiaozhang？
因为 this.divorce 在对象上，由于对象被返回了，这个方法也被返回了。
因为闭包。这个函数被储存到了外部，所以储存了这个函数的执行期上下文。所以可以用这个闭包。所以 var prepareWife 被下面的 this=fn 三个函数共用，这三个函数分别与 fn Deng 形成了闭包，共同用 Deng 的 AO,所以可以在外部随意存取。

```
function Deng(name, wife) {

    var prepareWife = "xiaozhang";


    this.name = name;
    this.wife = wife;
    this.divorce = function () {
        this.wife = prepareWife;
    }
    this.changePrepareWife = function (target) {
        prepareWife = target;
    }
    this.sayPraprewife = function () {
        console.log(prepareWife);
    }
}

var deng = new Deng('deng', 'xiaoliu');
```

```
> deng
▼ Deng
  ▶ changePrepareWife: (target)
  ▶ divorce: ()
    name: "deng"
  ▶ sayPraprewife: ()
    wife: "xiaoliu"
  ▶ __proto__: Object
```

```
> deng.divorce()
< undefined
> deng.wife
< "xiaozhang"
```

```
> deng.prepareWife
< undefined
```

deng.prepareWife 是 undefined 的，表面上看起来不是自己的，但是实际上只有对象自己通过对象自己设置的方法可以去操作他。外部用户通过对象. prepareWife 是看不到的。只有自己能看到，就是闭包的私有化的运用。

例

```
var inherit = (function () {
    var F = function () {};
    return function (Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
        Target.prototype.constuctor = Target;
        Target.prototype.uber = Origin.prototype;
    }
}());
```

最后执行完是这个样的

```
var inherit = function (Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
        Target.prototype.constuctor = Target;
        Target.prototype.uber = Origin.prototype;
    }
```

上面的 var F 这个 F 形成了闭包，成为了这个函数的私有化变量，而且变成私有化变量就更好

命名空间（其实就是对象）

管理变量，防止污染全局，适用于模块化开发

多人开发，对象命名容易重复，就要
解决命名空间的问题

右边是命名空间老旧的解决方 ==>

用的时候，用下面的写法

```
var jicheng = org.department1.jicheng;
jicheng.name
```

```
var org = {
    department1 : {
        jicheng : {
            name : "abc",
            age : 123
        },
        xuming : {

        }
    },
    department2 : {
        zhangsan : {

        },
        lisi : {

        }
    }

}
```

下面是现在公司最常见的方法 :用闭包来解决(也可用 webpack),返回方法的调用。
init 是初始化，入口函数，入口名字。init 调用了这个函数的功能

```
var name = 'bcd';
var init = (function () {

    var name = 'abc';


    function callName() {
        console.log(name);
    }

    return function () {
        callName();
    }

}())

init();
```

```
abc
>
```

```
var name = 'bcd';
var init = (function () {

    var name = 'abc';


    function callName() {
        console.log(name);
    }

    return function () {
        callName();
    }

}())
```

```
> init()
  abc
< undefined
> initDeng(0
✖ Uncaught Synt
  argument list
> initDeng()
  123
< undefined
```

```
var initDeng = (function () {

    var name = 123;
    function callName() {
        console.log(name);
    }

    return function () {
        callName();
    }


}())
```

思考问题

如何实现链式调用模式（模仿 jquery）

obj.eat().smoke().drink().eat().sleep();

例

```
<script type="text/javascript" src="jQuery.js"></script>
```

```
<body>
    <div></div>
    <script type="text/javascript">


        $('div').css('background-color','red').width(100).height(100).
            html(123).css('position','absolute').css('left','100px').
            css('top', '100px');
```

例

```
var deng = {
    smoke : function () {
        console.log('Smoking,... xuan cool!!!');
    },
    drink : function () {
        console.log('drinking..., ye cool!');
    },
    perm : function () {
        console.log('preming..., cool!');
    }
}
deng.smoke();
deng.drink();
deng.prem();
```

不能像右边这样连续调用 `deng.smoke().drink();`

上面改成下面的写法 : 用 return this，就可以连续调用和执行了。deng.smoke().drink()

```
var deng = {
    smoke : function () {
        console.log('Smoking,... xuan cool!!!');
        return this;
    },
    drink : function () {
        console.log('drinking..., ye cool!');
        return this;
    },
    perm : function () {
        console.log('preming..., cool!');
        return this;
    }
}
deng.smoke().drink().perm().smoke().drink();
```

```
Smoking,... xuan cool!!!
drinking..., ye cool!
preming..., cool!
Smoking,... xuan cool!!!
drinking..., ye cool!
```

上面是用 return this 连续调用

属性的表示方法（查看属性）

obj.prop　　查看就用.XXXX　　obj[“prop”]　　中括号也是访问属性的方法

例想要传序号几，就会调用几

```
var deng = {
    wife1 : {name : "xiaoliu"},
    wife2 : {name : "xiaozhang"},
    wife3 : {name : "xiaomeng"},
    wife4 : {name : "xiaowang"},
    sayWife : function (num){
        switch(num) {
            case 1:
                return this.wife1
        }
    }
}
```

wife1

老旧办法看右边，
然后 case1。。。　case2。。。连着往下写
这种方法不好，下面的方法更好

var obj = { name :　“abc”　}

用方括号来访问属性也是一样的（里面必须是字符串）

这两种基本上完全相同 obj.name　→　obj [　‘name’　]

想实现属性名的拼接，只能用方括号的形式

```
var deng = {
    wife1 : {name : "xiaoliu"},
    wife2 : {name : "xiaozhang"},
    wife3 : {name : "xiaomeng"},
    wife4 : {name : "xiaowang"},
    sayWife : function (num){
        return this['wife' + num];
    }
}
```

```
> obj.name
"abc"
> obj['name']
"abc"
```

对象的枚举 enumeration

for in 循环(简化版 for 循环)，目的是便利对象，通过对象属性的个数来控制循环圈数，这个对象有多少属性循环多少圈，而且在每一圈时，都把对象的属性名放到 Prop 里面　　在枚举里面，一定要写成 obj[prop]不能加字符串

1.hasOwnProperty

2.in

3.instanceof

枚举也就是遍历：挨个知道信息的过程就叫这个数据组的遍历

例

```
var arr = [1,3,3,4,5,6,7,8,9];
// 遍历 枚举 enumeration
for(var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```

例

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75
}

for(var prop in obj) {
    console.log(prop + " " + typeof(prop));
}
```

```
name string
age string
sex string
height string
weight string
```

//上面就是 for in 循环，就是遍历用的。通过对象的属性个数来控制循环圈数，有多少个属性就会循环多少圈。
for（var prop in obj）在循环每一圈的时候，他会把对象的属性名放在 prop 里面。想遍历谁就 in 谁，prop 可以写别的，obj 就是我们想要遍历的对象。
var XX in XX 的格式是固定的。

var 也可以写在外面，写成 var key；　for（key in obj1）效果是一样的

```
var obj1 = {
    a : 123,
    b : 234,
    c : 345
}
var key;
for(key in obj1) {
    obj1.key ++;
}
```

写成下面这样会出错

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75
}

for(var prop in obj) {
    console.log(obj.prop);
}
```

5 undefined

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75,
    prop : 123
}

for(var prop in obj) {
    console.log(obj.prop);
}
```

6 123

上面 obj.prop 系统以后我们写的是 obj[‘prop’]，系统会以为我们是在让他访问 prop 这个属性，不会把 prop 当成一个变量来使用。写成 obj[prop]就可以成功访问了。

```
for(var prop in obj) {
    // console.log(obj.prop  ---> obj['prop']     );
    console.log(obj[prop]);
}
```

```
13
123
male
180
75
```

写对象时用方括号的形式不容易犯错

例

右边写法会把原型上面的东西也拿出来

```
13
123
male
180
75
deng
```

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75,
    __proto__ : {
        lastName : "deng"
    }
}

for(var prop in obj) {
    console.log(obj[prop]);
}
```

如果在遍历的时候，我们不想把原型上面的属性拿出来，可以用 hasOwnProperty，一般与 for in 循环成套出现

hasOwnProperty 是一个方法，来判断这个对象是你自己的还是原型的， 任何一个对象里面都有 hasOwnProperty，里面是需要传参的，把属性名传进去（如 prop）。下面达到了如果不是自己的属性，是原型上的属性，就不会返回。

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75,
    __proto__ : {
        lastName : "deng"
    }
}

for(var prop in obj) {
    if(obj.hasOwnProperty(prop)) {
        console.log(obj[prop]);
    }
}
```

```
13
123
male
180
75
```

注明：for in 循环理论上可以返回原型和原型链上的东西，一旦这个原型链延展到了的 object.prototype 上，不会打印系统的，只会打印自带的。

```
var obj = {
        __proto__ : {
            lasrName : "deng ",
            __proto__ : Object . prototype      //不会打印这个 object.prototype
        }
    }
```

```
__proto__ : {
    lastName : "deng",
    __proto__ : Object.prototype
}
```

例 加 Object . prototype.abc = '123'； 加个！变成只有不是他的才打印

```
var obj = {
    name : '13',
    age : 123,
    sex : "male",
    height : 180,
    weight : 75,
    __proto__ : {
        lastName : "deng"
    }
}

Object.prototype.abc = '123';

for(var prop in obj) {
    if(!obj.hasOwnProperty(prop)) {
        console.log(obj[prop]);
    }
}
```

```
deng
123
```

in 操作符：很少用

in 操作符你的也是你的，你父亲的也是你的，只能判断这个对象能不能访问到这个属性，包括原型上；不是判断属性属不属于这个对象的

```
> height in obj
✖ ▶ Uncaught ReferenceError: height is not defined
      at <anonymous>:1:1
```

```
> 'height' in obj
< true
```

```
> 'lastName' in obj
< true
```

判断一个属性属不属于这个对象的只能用 hasOwnProperty

instanceof 操作用法类似于 in，但是完全不同

A instanceof B 的意思是 A 对象是不是 B 构造函数构造出来的；记住是：看 A 对象的原型链上有没有 B 的原型

解决了

例

```
function Person() {

}
var person = new Person();
```

```
> person instanceof Object
< true
```

```
> [] instanceof Array
< true
> [] instanceof Object
< true
```

判断这个变量是数组还是变量

例

```
function Person() {

}
var person = new Person();
var obj = {};
```

```
> obj instanceof Person
< false
```

```
> typeof([])
< "object"
> typeof({})
< "object"
```

例区别传的变量是数组还是对象的方法：一是 constructor，二是 instanceof

```
var arr = {};// [];
```

```
> obj.constructor
< function Object() { [native code] }
```

```
> [] instanceof Array
< true
> var obj = {};
< undefined
> obj instanceof Array
< false
```

第三种区分数组还是对象的方法：想让数组调用他的 toString 方法

Object . prototype . toString . call ([ ]) ; //[]会替换 this

Object . prototype . toString = function ( ) {

识别 this（谁调的他，这个 this 就是谁）

返回相应的结果

}

obj . toString ( ) ;

```
Object.prototype.toString.call([]);

Object.prototype.toString = function () {

    // 识别 this
    // 返回相应的结果


}

// obj.toString();
```

```
> Object.prototype.toString.call([]);
< "[object Array]"
>       Object.prototype.toString.call(123);
< "[object Number]"
>       Object.prototype.toString.call({});
< "[object Object]"
```

例：下面这段 js 代码执行完毕后 x，y，z 的值分别是多少？

```
var x = 1, y = z = 0;
function add(n){
    return n = n + 1;
}
y = add(x);
function add(n){
    return n = n + 3;
}
z = add (x);
```

答案 1，4，4，同一个函数，后面会覆盖前面的，fn add 会提升到前面

例：下面代码中 console.log 的结果是[1, 2, 3, 4, 5]的选项是：

```
A、function foo (x){
    console.log(arguments)
    return x
}
foo(1, 2, 3, 4, 5)
B、function foo (x){
    console.log(arguments)
    return x
} (1, 2, 3, 4, 5)
C、( function foo (x){
    console.log(arguments)
    return x
})(1, 2, 3, 4, 5)
D、function foo(){bar .apply (null, arguments);}
function bar (x) {console.log(arguments);}
foo(1, 2, 3, 4, 5);
```

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

答案 A, C,D　其中 b 执行不了,但是不报错

bar 里面传了一下参数。bar .apply (null, arguments);写成 bar (arguments);

例：一行文本，水平垂直居中

答案：height = line height　text-align:center

例：请问以下表达式的结果是什么？

parseInt(3, 8)　parseInt(3, 2)　parseInt(3, 0)

A.3，3，3　B.3，3，NaN　C.3，NaN，NaN　D.other

答案 C 或 D（有的浏览器 0 进制报错，有的不报错）

进制

十六进制 0 1 2 3 4 5 6 7 8 9 a b c d e f

十六进制的中 10 是十进制的 16，　1f = 16 + 15

二进制中的 10 是十进制的 2，11 是十进制的 3

在十进制中：	在二进制中：
1 = 1	1 = 1
10 = 10	10 = 2
100 = 10 ^ 2	100 = 2 ^ 2
1000 = 10 ^ 3	1000 = 2 ^ 3
10000 = 10 ^ 4	10000 = 2 ^ 4

例：以下哪些是 JavaScript 语言 typeof 可能返回的结果

A.string　B.array　C.object　D.null

例：JavaScript 的 call 和 apply 方法是做什么的，两者有什么区别？

例：看看下面 alert 的结果是什么？

```
function b (x, y, a){
    arguments [ 2 ] = 10;
    alert( a );
}
b(1, 2, 3);
```

如果函数体改成下面，结果又会是什么？

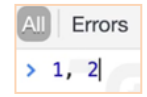
```
a = 10;
alert(arguments[2]);
```

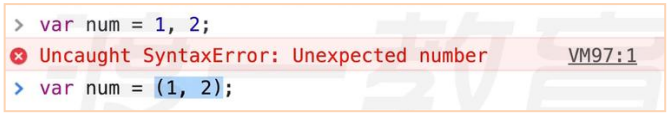
答案两个都是 10

逗号操作符，这种情况，会看一眼 1，在看一眼 2，然后返回第二个，就是 2

例：

```
var f = (
    function f() {
        return "1";
    },
    function g() {
        return 2;
    }
)();
typeof f;
```





答案是 number

例

```
var x = 1;
if(function f() {}) {
    x+= typeof f;
}

console.log(x);
```

用括号把 function f （ ）{}转换成表达式，就被立即执行了，就找不到了。
因为 function f （ ）{}肯定是 true，所以会执行{}，但 typeof f 中的 f 已经找不到了
答案是 1undefined
下面是用友的题目
例：以下哪些表达式的结果为 true（ ）
A.undefined == null　B.undefined === null　C.isNaN(“100”)　D.parseInt(“la”) ==1
答案 A、C、D　//isNaN（“ 100”）意思是这个数经过 number 转换后是不是 NaN
C

```
function isNaN(num) {

    var ret = Number(num);
    ret += "";
    if(ret == "NaN") {
        return true;
    }else {
        return false;
    }

}
```

```
> myIsNaN('123')
< false
```

```
> myIsNaN('NaN')
< true
```

parseInt（“1a”）==1。

引用值比的是地址

```
> {} == {}
< false
```

```
> var obj = {};
< undefined
> var obj1 = obj;
< undefined
> obj1 == obj
< true
> obj1 === obj
< true
```

this

1.函数预编译过程 this —>指向 window

```
function test(c) {
    //var this = Object.create(test.prototype);
    //{
    //__proto__ : test.prototype
    //}
    var a = 123;
    function b() {}

}
AO {
    arguments : [1],
    this : window,
    c : 1,
    a : undefined,
    b : function () {}
}

test(1);
new test();
```

前面的 var this = Object.creat(test.prototype);是最标准的写法
new test( ) ;就会让 var this = Object.creat(test.prototype);如果不 new ,this 指向 window

例

```
function test() {
    console.log(this);
}
test();
```

```
▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo…}
```

```
> window
< ▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo…}
```

2.全局作用域里 this —> 指向 window

```
> this
< ▶ Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo…}
```

3.call/apply 可以改变函数运行时 this 指向
4.obj.func(); func()里面的 this 指向 obj
谁调用这个方法,这个方法里的 this 就是指向谁 →

```
var obj = {
    a : function () {
        console.log(this.name)
    },
    name : 'abc'

}
obj.a();
```

例

a.say 是 function 函数体
function (fun){
//this → b
//console.log(this)→b
fun( );
}
fun()空执行，走预编译

在 b.say(a.say)中
a.say 当做参数传进来了

```
var name = "222";
var a = {
    name : "111",
    say : function () {
        console.log(this.name);
    }
}
var fun = a.say;
fun()//222 姚式          222 孙式          222 陈 + 马
a.say()//111             111               111

var b = {
    name : "333",
    say : function (fun) {
        fun();
    }
}
b.say(a.say);//333       222               333      111
b.say = a.say;
b.say();//333            333               333
```

答案 fun( )是 222　a.say 是 111　b.say(a.say);是 222　b.say( );是 333

arguments

arguments.callee 指向函数的引用（函数自己）

function.caller

例

```
function test() {
    console.log(arguments.callee);
}

test();
```

```
function test() {                    lesson1
        console.log(arguments.callee);
    }
```

例

```
function test() {
    console.log(arguments.callee == test);
}

test();
```

```
true
```

例我们要初始化数据，是 100 以内的阶乘，用立即执行函数找到自己的引用来解决

```
var num = (function (n) {

    if(n == 1) {
        return 1;
    }
    return n * arguments.callee(n - 1);

}(100))
```

例

```
function test( ){
    console.log(arguments.callee);
    function demo() {
        console.log(arguments.callee);
    }
    demo();
}
test();
```

```
function test(){                     lesson14.html:14
        console.log(arguments.callee);
        function demo() {
            console.log(arguments.callee);
        }
        demo();
    }
function demo() {                    lesson14.html:16
        console.log(arguments.callee);
    }
```

在哪个函数里面的 arguments.callee 就指代了哪个函数

例 caller 谁叫他，caller 不能用在 arguments 里面

```
function test () {
    demo();
}

function demo() {
    console.log(demo.caller);
}

test();
```

```
function test() {
        demo();
    }
```

demo 被调用的环境是 test

所以这个 caller 指代的 test

例

1、 请阅读以下代码，写出以下程序的执行结果：

```
var foo = '123';
function print(){
    var foo = '456';
    this.foo="789";
    console.log(foo);
}
print();
```

执行结果：_____

答案是 456

把上面这题变形：

```
var foo = 123;
function print() {
    this.foo = 234;
    console.log(foo);
}

print();
```

答案打印 234，要 console.log(foo);是全局的 foo，但是这里 this.foo 的 this 是指向全局 window 的，相当于就把外面的 123 改成了 234

例

```
var foo = 123;
function print() {
    // var this = Object.create(print.prototype)
    this.foo = 234;
    console.log(foo);
}
new print();
```

答案 123，new 了以后，隐式 var this = Object.create(print.prototype)这时候 this.foo 不再指向 window，转而指向 var this，所以**打印**的时候找不到 234，就到全局找到 123

例

```
1.运行 test() 和 new test()的结果分别是什么?
var a = 5;
function test() {
    a = 0;
    alert(a);
    alert(this.a);
    var a;
    alert(a);
}
```

答案运行 test( )是 0，5，0【之前其中 this.a 指代的是 window，所以是 5】
运行 new test( )是 0，undefined ，0【因为 this 上没有 a，所以打印 undefined
在执行 test( )之前，AO 被定义{a:undefined}，执行之后，AO 被执行{a:0,this:window}
在执行new test( )时会隐式的var this = Object.create(test.prototype)简化就是var this ={ __proto__: test prototype}一个对象上面没有的属性，打印出来就是 undefined
new test( );执行的时候 AO 是{a:0，this:{} }

例

```
2、 请阅读以下代码，写出以下程序的执行结果；
    function print(){
        console.log(foo);
        var foo=2;
        console.log(foo);
        console.log(hello);
    }
    print();                    执行结果：_____
```

答案 undefined，2，报错 hello is no defined
因为 hello 没有被定义，所以报错

例

```
3、 请阅读以下代码，写出以下程序的执行结果；
    function print(){
        var test;
        test();
        function test (){
            console .log(1);
        }
    }
    print();                    执行结果：_____
```

答案 1

例

```
4、 请阅读以下代码，写出以下程序的执行结果；
    function print(){
        var x = 1;
        if(x == "1") console.log("One!");
        if(x === "1") console.log("Two!");
    }
    print();                    执行结果：_____
```

答案打印 One！

例

```
5、 请阅读以下代码，写出以下程序的执行结果；
    function print(){
        var marty = {
            name:"marty",
            printName:function(){ console.log(this.name);}
        }
        var test1={   name:"test1" };
        var test2={   name:"test2" };
        var test3={   name:"test3" };
        test3.printName=marty.printName;
        var printName2 = marty.printName.bind({name:123});
        marty.printName.call(test1);
        marty.printName.apply(test2);
        marty.printName();
        printName2();
        test3.printName();
    }
    print();                    执行结果：_____
```

答案：test1，test2，marty，做不了有 bind，test3

例

bar.a=’ a’ ;
把{a:” 002” }
变成
{a:’ a’ }

```
6、 请阅读以下代码，写出以下程序的执行结果；
    var bar = {a:"002"};
    function print() {
        bar.a = 'a';
        Object.prototype.b = 'b';
        return function inner() {
            console.log(bar.a);
            console.log(bar.b);
        }
    }
    print()();                  执行结果：_____
```

答案 a，b 其中 print( ) ( )第一个括号返回的是一个函数，第二个再来函数执行

克隆

浅层克隆　　深层克隆

例这个是做的浅层克隆

```
var obj = {
    name : 'abc',
    age : 123,
    sex : 'female'
}

var obj1 = {}
```

```
function clone(origin, target) {

    for(var prop in origin) {
        target[prop] = origin[prop];
    }
}
```

```
> obj1
◂ ▶Object {name: "abc", age: 123, sex: "female"}
```

把上面做一个兼容性的写法，为了防止用户不传 target（容错），给了参数就直接用，不给就当空对象，见下方

```
function clone(origin, target) {
    var target = target || {};
    for(var prop in origin) {
        target[prop] = origin[prop];
    }
    return target;
}

clone(obj, obj1);
```

```
> obj
◂ ▶Object {name: "abc", age: 123, sex: "female"}
> obj1
◂ ▶Object {name: "abc", age: 123, sex: "female"}
> obj.name = 'bcd';
◂ "bcd"
> obj1.name
◂ "abc"
```

例

```
var obj = {
    name : 'abc',
    age : 123,
    sex : 'female',
    card : ['visa','unionpay']
}

var obj1 = {}


function clone(origin, target) {
    var target = target || {};
    for(var prop in origin) {
        target[prop] = origin[prop];
    }
    return target;
}

clone(obj, obj1);
```

```
> obj1
◂ ▶Object {name: "abc", age: 123, sex: "female", card: Array[2]}
> obj1.card.push('master');
◂ 3
> obj1.card
◂ ▶["visa", "unionpay", "master"]
> obj
◂ ▶Object {name: "abc", age: 123, sex: "female", card: Array[3]}
```

```
> obj.card
◂ ▶["visa", "unionpay", "master"]
```

现在我们想实现深度克隆（只考虑数组和对象），copy 过去后，我改，但是你不会改，引用值不能直接拷贝

思考上一个题做深度克隆，分析他是什么，建立是什么，收尾需要一个递归，提示：

```
    card : ['visa','unionpay',[1,2]]
}

var obj1 = {
    card : [obj.card[0],obj.card[1],[]]
}
```

作业：做一个深度克隆（copy 后，各自独立，互不影响）

思路：需要一个分析环节，分析是什么，是原始值就按原来的方法拷贝过来，是引用值就分析是数组还是对象。如果是数组，就新建一个数组；如果是对象，就新建一个对象。再一层层看，里面有一个递归。引用值不能直接拷贝，引用值拷贝的是地址

答案

```
var obj = {
    name : "abc",
    age : 123,
    card : ['visa', 'master'],
    wife : {
        name : "bcd",
        son : {
            name : "aaa"
        }
    }
}

var obj1 = {

}
```

```
> deepClone(obj, obj1)
◂ undefined
> obj1
◂ ▶Object {name: "abc", age: 123, card: Array[2], wife: Object}
> obj.card.push('abc')
◂ 3
> obj1.card
◂ ▶["visa", "master"]
```

```
function deepClone(origin, target) {
    var target = target || {},
        toStr = Object.prototype.toString,
        arrStr = "[object Array]";
    for(var prop in origin) {
        if(origin.hasOwnProperty(prop)) {
            if(origin[prop] !== "null" && typeof(origin[prop])
                == 'object') {
                if(toStr.call(origin[prop]) == arrStr) {
                    target[prop] = [];
                }else{
                    target[prop] = {};
                }
                deepClone(origin[prop], target[prop]);
            }else{
                target[prop] = origin[prop];
            }
        }
    }
    return target;
}
```

```
var obj = {
        name :  “abc” ,
        age : 123,
        card : [ ‘visa’ ,  ‘master’ ], //原始对象
        wife : {
            name :  “bcd” ,
            son : {
                name :  “aaa”
          }
        }
}
var obj1 = {
        name :  “abc” ,
        age : 123,
        card : [ obj.card[ 0 ],obj.card[ 1 ] ],
//要拷贝的对象，进行 obj 里面的数组，对这个数组的拷贝再一次拷贝
        wife : {
            name :  “bcd” ,
            son : {
                name :  “aaa”
          }
        }
}
function deepClone(origin, target){
        var target = target | | {},   //有就用你的，没有就用后面的
            toStr = Object.prototype.toString,   //引用，目的是简化
            arrStr =  “ [ Object Array ] ” ;   //引用，目的是简化比对
      for ( var prop in origin ) {    //从原始 origin 拷贝到 target
          if( origin . hasOwnProperty( prop ) ){   //先判断是不是原型上的属性，如果是
false 就是原型上的
              if( typeof (origin [ prop ] ) !==  “null”  && typeof ( origin [ prop ] ) ==
 ‘object’ ) {
                  if ( toStr.call( origin  [ prop ] ) ) == arrStr ){
                      target [ prop ] = [ ];
                  }else{
                      target [ prop ] = { };
                  }
                  deepClone ( origin [ prop ], target [ prop ]);
              }else{     //else 后面是原始值
                  target [ prop ] = origin [ prop ];
              }
          }
      }
      return target;
}
```

深度克隆的步骤

1、先把所有的值都遍历一遍（看是引用值和原始值）

用 for (var prop in obj)，对象和数组都可以使用

2、判断是原始值，还是引用值？用 typeof 判断是不是 object

1）如果是原始值就直接拷贝

2）如果是引用值，判断是数组还是对象

3、判断是数组还是对象？（方法 instanceof【看 a 的原型链上有没有 b 的原型】、toString、constructor，建议用 toString,另外两个有个小 bug——跨父子域不行）

1）如果是数组，就新建一个空数组；

2）如果是对象，就新建一个空对象。

4、建立了数组以后，如果是挨个看原始对象里面是什么，都是原始值就可以直接考过来了；或者，建立了对象以后，挨个判断对象里面的每一个值，看是原始值还是引用值

5、递归

三目运算符 ？ ：

形式：?问号前面是一个条件判断，判断 true 就走：冒号前面的；false 就走：冒号后面的，并且会返回值 条件判断？ 是 ： 否 并且会返回值

三目运算符是简化版的 if（条件判断）{是在这里}else{否在这里}

例 var num = 1 > 0 ? 2 + 2 : 1 + 1;

//答返回值为 4

例 var num = 1 < 0 ? 2 + 2 : 1 + 1;

//答返回值为 2

例 ar num = 1 > 0 ? ("10" > 9 ? 1 : 0) : 2;

//答 1

例 var num = 1 > 0 ? (“10” > “ 9 “ ? 1 : 0) : 2;

//答 0，因为“10” > “ 9 “ 比的是 ASCII 码，一位位比，一零小于 9，先用 1 和 9 比，在 asc 妈里面 1 小于 9；

当“10” > 9 字符串和数字比，会先转换成数字再比较

例**可利用三目运算符简化克隆的代码**

```
    for ( var prop in origin ) {
        if( origin . hasOwnProperty( prop ) ){
            if( typeof (origin [ prop ] ) !==  “null”  && typeof ( origin [ prop ] ) ==
‘object’ ) {
                target [ prop ] = toStr.call (origin [ prop ] ) ==arrStr ? [] : {};
                deepClone ( origin [ prop ], target [ prop ]);
            }else{      //else 后面是原始值
                target [ prop ] = origin [ prop ];
            }
        }
    }
return target;
}
```

```
function deepClone(origin, target) {
    var target = target || {},
        toStr = Object.prototype.toString,
        arrStr = "[object Array]";
    for(var prop in origin) {
        if(origin.hasOwnProperty(prop)) {
            if(origin[prop] !== "null" && typeof(origin[prop])
                == 'object') {
                // if(toStr.call(origin[prop]) == arrStr) {
                //     target[prop] = [];
                // }else{
                //     target[prop] = {};
                // }
                target[prop] = toStr.call(origin[prop]) ==
                    arrStr ? [] : {};
                deepClone(origin[prop], target[prop]);
            }else{
                target[prop] = origin[prop];
            }
        }
    }
    return target;
}
```

封装 type 方法

定义对象方式：自变量，构造函数，自定义的构造函数，Object.create

定义数组的方式：var arr = [ ] ;数组自变量；var arr = new Array ( ) ;系统提供。两者区别就只一位数的情况

数组能用的方法来源于 Array.prototype

例 var arr = [ ] ;

如果写 var arr = [1, ,1]，出来就是 1，undefined*1，1

数组不是每一位都有值，稀疏数组

例 var arr = new Array ( ) ;

var arr = new Array(1,2,3,4,5);跟上面效果一样

例 var arr = new Array (10);长度为 10 的稀疏数组，括号里面只有一位数，就代表着长度，并且里面每一位都没有值，console 里面会是 undefined*10。并且里面不能写小数，会报错。

var arr1 = [10] ;

数组的读和写，基本上没有报错的情况，除非是引用了没有的方法

js 数组是基于对象的，数组是一种特殊的对象

```
> arr[10] = "abc"
< "abc"
> arr
< ▶ [undefined × 10, "abc"]
> arr.length
< 11
```

数组

一、数组的定义(来源于 Array.prototype)

1）new Array(length/content);　　**var arr = new Array(1,2,3,4,5);**

2）字面量　　**var arr = [1,2,3,4,5];**

二、数组的读和写

arr [num]　　**//不可以溢出读，结果 undefined**

arr[num] = XXX;　　**//不可以溢出读**

arr[num] = xxx;　　**//可以溢出写**

es3.0 最标准最基础　es5.0　es6.0，最新的 es7.0 还没有普及，今天讲的都是 es3.0

对象的定义方式 1 字面量.2 构造函数,3 自定义构造函数 4object.create

数组是一种特殊的对象，在本质上两者没有太大的区别

例可以写 var arr = [,]　//稀松数组，相当于定了两个位置，conlogo 结果是 undefined

例

```
var arr = [1,2,,,,3,4];
```

```
> arr
< ▶ [1, 2, undefined × 3, 3, 4]
```

```
> arr.length
< 7
```

例 var arr = [ ];和 var arr = new Array( );唯一的区别是在 var arr = new Array( );只传了一个参数的情况下，会被当成长度，并且成为一个稀松数组

```
var arr = new Array(10);
var arr1 = [10];
```

```
> arr
< ▶ [undefined × 10]
```

如传进去的参数是一个小数，就非法

```
var arr = new Array(10.2);
```

```
✖ ▼ Uncaught RangeError: Invalid array length
```

例

```
var arr = [];
```

```
> arr[10]
< undefined
```

```
> arr[10] = "abc"
< "abc"
> arr
< ▶ [undefined × 10, "abc"]
> arr.length
< 11
```

数组常用的方法

一、改变原数组（在原来数组基础上去改变）

1）reverse,sort,push,pop, unshift, shift,

2）splice

二、不改变原数组

1）forEach filter map reduce reduceRight

2）slice concat,join—>split,toString

在控制台操作

例 push 是在数组的最后一位添加数据，可以添加一个，也可以添加很多个

```
var arr = [];
```

```
> arr.push(10)
< 1
> arr
< ▶ [10]
> arr.push(11)
< 2
> arr
< ▶ [10, 11]
```

```
> arr.push(9)
< 3
> arr
< ▶ [10, 11, 9]
```

```
> arr.push(1,2,3,4,4,6,7)
< 7
> arr
< ▶ [1, 2, 3, 4, 4, 6, 7]
```

例说明能重写

```
var arr = [];
Array.prototype.push = function () {
    return 'haha';
}
```

```
> arr.push()
< "haha"
```

例数组有三位，想在数组最后一位添加东西

var arr = [1, 2, 3];如果在第四位加东西写成 arr = [3]。 //length-1 位添加东西

例

```
var arr = [1,2,3];
Array.prototype.push = function () {
    for(var i = 0; i < arguments.length; i ++) {
        this[this.length] = arguments[i];
    }
    return this.length;
}
```

```
> arr.push(4,5,6)
< 6
> arr
< ▶ [1, 2, 3, 4, 5, 6]
```

Array . prototype.push = function ( ){ //不能写形参

例 pop 是剪切方法（把最后一位数剪切出去）。在 pop( )括号里面不能传参，写了会忽略

```
var arr = [1,2,3];
```

```
> arr.pop()
< 3
> arr
< ▶ [1, 2]
```

```
> arr.pop()
< 2
> arr
< ▶ [1]
```

```
> var num = arr.pop()
< undefined
> num
< 3
```

```
> arr.pop(2)
< 3
> arr
< ▶ [1, 2]
```

例 unshift 是从第一位加东西

```
var arr = [1,2,3];
```

```
> arr.unshift(0)
< 4
> arr
< ▶ [0, 1, 2, 3]
```

```
> arr.unshift(-1,0)
< 5
> arr
< ▶ [-1, 0, 1, 2, 3]
```

例 shift 是从第一位开始减

```
var arr = [1,2,3];
```

```
> arr.shift()
< 1
```

```
> arr
< ▶ [2, 3]
```

例数组不能从-1 位插入东西

```
> arr[-1] = 0
< 0
> arr
< ▶ [1, 2, 3]
```

可以用两个数组拼接成一个数组的方式添加东西

例 reverse 逆反

```
var arr = [1,2,3];
```

```
> arr.reverse()
< ▶ [3, 2, 1]
> arr.reverse()
< ▶ [1, 2, 3]
```

```
> arr
< ▶ [1, 2, 3]
```

例 splice 一种剪切，切片

```
var arr = [1,1,2,2,3,3];
```

// 这是从第零位到第五位

/arr.splice(从第几位开始，截取多少长度，传参在切口处添加新的数据)

arr.splice(1,2); //从第 1 位开始截取 2 位，传参可以不填

截取的是[1,2]

```
> arr.splice(1,2);
< ▶ [1, 2]
> arr
< ▶ [1, 2, 3]
```

```
> arr.splice(0, 3)
< ▶ [1, 1, 2]
> arr
< ▶ [2, 3, 3]
```

```
> arr.splice(1,1,0,0,0)
< ▶ [1]
> arr
< ▶ [1, 0, 0, 0, 2, 2, 3, 3]
```

例 arr.splice(1,1,0,0,0);

//意思是从第 1 位起截取 1 位，然后加上 0,0,0 这三个数

例 var arr = [1,2,3,5]; //1 是第 0 位，2 是第一位，3 是第二位，5 是第三位

arr.splice(3,0,4);

变成 arr ==> 1,2,3,4,5

```
> arr
< ▶ [1, 2, 3, 4, 5]
```

例 var arr = [1,2,3,4];

arr.splice(-1,1);

//这里的-1 是倒数第一位

数组一般的方法都可以带负数

```
> arr.splice(-1, 1)
< ▶ [4]
```

例下面是系统内部解决负数的问题的兼容

```
var arr = [1,2,3,4];

splice = function (pos) {

    pos += pos > 0 ? 0 : this.length;

}

-1 + 4 = 3
```

例

```
function( i ) {
var len = this.length,
    j = +i + ( i < 0 ? len : 0 );
return this.pushStack( j >= 0 && j < len ? [ this[ j ] ] : []
```

```
// splice = function (pos) {

//     pos += pos > 0 ? 0 : this.length;
    pos >= 0 || pos <
// }
```

例 sort 给数组排序（按照从小到大），改变原数组

```
var arr = [1,3,4,0,-1,9];
```

```
> arr.sort()
◂ ▶ [-1, 0, 1, 3, 4, 9]
```

在 sort 后面加 reverse 就是降序

```
> arr.sort().reverse()
◂ ▶ [9, 4, 3, 1, 0, -1]
```

例下面这个是按 asc 码排序的

```
var arr = [1,3,5,4,10];
```

```
> arr.sort()
◂ ▶ [1, 10, 3, 4, 5]
```

所以给我们留了个接口，如下图

sort 按 asc 码排序的

1 必须写两形参

2 看返回值 return

1）当返回值为负数时，那么前面的数放在前面，

2）当返回值为正数时，那么后面的数在前，

3）为 0，不动

右边的

return 1;　　//此处的 1 代表返回正的

return -1;　　//此处的-1 代表返回负的

```
var arr = [1, 3, 5, 4, 10];

arr.sort(function (a, b) {

    if(a > b) {
        return 1;
    }else {
        return -1;
    }

});
```

思维方式：上面就控制了升序

这个函数第一次调动时，会把数组的第一位和第二位传进来，也就是 a=1，b=3，然后通过规则比较，当你把返回值返回为正，为负，为 0。

传参的顺序（但是这个顺序是位子的顺序，不是按数字比），第一次是 1,3，第二次 1,5,第三次 1,4,第四次 1,10，第五次 3,5,第六次 3,4,第七次 3,10，第八次 5,4，第九次 5,10，第十次 4,10；依次传参（符合**冒号排序**的算法）

是以换位置的方式改变顺序

例 var arr = [2,13,19,4];

//到 13 这一位时，换位置变成了[2,4,19,13]

例下面这种方式变成了降序

```
var arr = [2,10,20,13,4,8,9];

arr.sort(function (a, b) {

    if(a < b) {
        return 1;
    }else {
        return -1;
    }

});
```

例

```
var arr = [20,2,10,13,4,8,9];

arr.sort(function (a, b) {

    if(a - b > 0) {
        return a - b
    }else {
        return a - b
    }

});
```

```
var arr = [20,2,10,13,4,8,9];

arr.sort(function (a, b) {

    return a - b;

});
```

简化版

记住升序 return a － b；　降序 return b － a；

直接调用 arr.sort( )比的是 asc 码，要在里面填函数才可以

```
var arr = [20,2,10,13,4,8,9];

arr.sort(function (a, b) {

    // return a - b; 升序
    // return b - a; 降序
    return b - a;

});
```

```
> arr
◂ ▶ [20, 13, 10, 9, 8, 4, 2]
> arr.sort(function () {})
```

例给一个有序的数组，乱序。当我们没规律可以遵循的时候，返回的是随机的

```
var arr = [1,2,3,4,5,6,7];
arr.sort(function() {

    return Math.random() - 0.5;

});
```

// Math.random() 会生成一个 0 到 1（包括 0，但是不包括 1）的随机数

例给他们三个按照年龄升序

```
var cheng = {
    name : "cheng",
    age : 18,
    sex : 'male',
    face : "handsome"
}
var deng = {
    name : "deng",
    age : 40,
    sex : undefined,
    face : "amazing"
}
var zhang = {
    name : "zhang",
    age : 20,
    sex : "male"
}
```

```
var arr = [cheng, deng, zhang];
arr.sort(function (a, b) {

    if(a.age > b.age) {
        return 1;
    }else{
        return -1;
    }

});
```

可以把右上改成简化版

```
var arr = [cheng, deng, zhang];
arr.sort(function(a, b) {

    return a.age - b.age;

});
```

简化版

例按字符串长度排序

```
var arr = ['ac','bcd','cccc','dddd','asdfkhiuqwe','
    asdoifqwoeiur','asdf'];

arr.sort(function (a, b) {

    return a.length - b.length;

});
```

例按字节长度排序

```
function retBytes(str) {

    var num = str.length;
    for(var i = 0; i < str.length; i ++ ) {
        if(str.charCodeAt(i) > 255) {
            num ++;
        }
    }
    return num;

}

var arr = ['ac','bcd','cccc','dddd','asdfkhiuqwe','
    asdoifqwoeiur','asdf'];

arr.sort(function (a, b) {

    return a.length - b.length;

});
```

下面略微修改一下

```
    var num = str.length;
    for(var i = 0; i < str.length; i ++ ) {
        if(str.charCodeAt(i) > 255) {
            num ++;
        }
    }
    return num;

}

var arr = ['a邓','ba邓','cc邓cc','老邓','残邓','
    asdoifqwoeiur','asdf'];

arr.sort(function (a, b) {

    return retBytes(a) - retBytes(b);

});
```

```
> arr
< ▶ ["a邓", "ba邓", "老邓", "残邓", "asdf", "cc邓cc",
    "asdoifqwoeiur"]
```

例 concat 连接， 把后面的数组拼到前面，并成立一个新的数组，不影响之前的两个数组。不能改变原数组

```
var arr = [1,2,3,4,5,6];
var arr1 = [7,8,9];
```

```
> arr.concat(arr1)
< ▶ [1, 2, 3, 4, 5, 6, 7, 8, 9]
> arr
< ▶ [1, 2, 3, 4, 5, 6]
> arr1
< ▶ [7, 8, 9]
```

例 toString 是把数组当做字符串展示出来

```
var arr = [1,2,3,4,5,6];
var arr1 = [7,8,9];
```

```
> arr.toString()
< "1,2,3,4,5,6"
```

例 slice 从该位开始截取，截取到该位，并不改变原数组，这里也**可以写负数**

```
var arr = [1,2,3,4,5,6];
//slice(从该位开始截取，截取到该位)
```

```
> arr.slice(1, 2)
< ▶ [2]
> arr
< ▶ [1, 2, 3, 4, 5, 6]
```

slice 并不改变原数组 slice 完了以后需要有东西接收，不然没有意义

```
var arr = [1,2,3,4,5,6];
//slice(从该位开始截取，截取到该位)
var newArr = arr.slice(1, 3)
```

```
> newArr
< ▶ [2, 3]
```

slice 里面可以填 0 个参数，也可以填 1 个参数，也可以填两个参数

1、如果填两个参数，slice（从该位开始截取，截取到该位）

如 arr.slice(1,2)从第一位开始截取，截取到第二位

2、如果填一个参数，从第几位开始截取，一直截取到最后。

如果 arr.slice(1) ，从第 1 位开始截取，截取到最后一位

3、不写参数就是整个截取数组（把类数组转换成数组的时候经常使用到）

例 **join 括号里面需要用字符串形式（标准语法规定），就会用加的东西连接起数组**

```
var arr = [1,2,3,4,5,6];
```

```
> arr.join("-")
< "1-2-3-4-5-6"
> arr.join("!")
< "1!2!3!4!5!6"
> arr.join("~")
< "1~2~3~4~5~6"
> arr.join("老邓")
< "1老邓2老邓3老邓4老邓5老邓6"
```

例 join 可逆的东西：split( )是 string 字符串方法

```
var arr = [1,2,3,4,5,6];
var str = arr.join('-');
```

```
> str
< "1-2-3-4-5-6"
> str.split("-")
< ▶ ["1", "2", "3", "4", "5", "6"]
> str.split("4")
< ▶ ["1-2-3-", "-5-6"]
```

split 按照什么拆分为数组。用什么拆，什么就没了，按-拆就去掉-，按 4 拆就去掉 4。

split 可以返回数组，数组可以返回字符串

例把下面字符串拼到一起

```
var str = "alibaba";
var str1 = 'baidu';
var str2 = 'tencent';
var str3 = 'toutiao';
var str4 = 'wangyi';
var str5 = 'xiaowang';
var str6 = 'nv';
```

下面这种写法不好，字符串是在栈内存里面的，先进后出

```
var strFinal = "";
var arr = [str, str1, str2, str3, str4];
for(var i = 0; i < arr.length; i++) {
    strFinal += arr[i];
}
```

用下面这种方式更好，join 里面不传参默认用逗号连接，传空串如下图

```
// 散列
var arr = [str,str1,str3,str4,str5];
console.log(arr.join(""));
```

```
alibababaidutoutiaowangyixiaowang
```

类数组

1、可以利用属性名模拟数组的特性

2、可以动态的增长 length 属性

3、如果强行让类数组调用 push 方法，则会根据 length 属性值的位置进行属性的扩充。

例这个看着像数组，但是数组有的方法，他全部都没有，所以他是类数组

```
function test() {

    console.log(arguments);
    arguments.push(7);
}

test(1,3,3,4,5,6);
```

```
▶ [1, 3, 3, 4, 5, 6]
▶ Uncaught TypeError:
  arguments.push is not a function
    at test (lesson15.html:14)
    at lesson15.html:17
```

类数组长得像数组，但是没有数组所拥有的的方法。

例

```
var obj = {
    "0" : 'a',
    "1" : 'b',
    "2" : 'c'
}

var arr = ['a','b','c'];
```

```
> obj[0]
< "a"
> arr[0];
< "a"
```

例下面是类数组的基本形态

```
var obj = {
    "0" : 'a',
    "1" : 'b',
    "2" : 'c',
    "length" : 3,
    "push" : Array.prototype.push
}
```

```
> obj.push('d')
< 4
> obj
< ▶Object {0: "a", 1: "b", 2: "c", 3: "d", length: 4}
```

在控制台 push('d')以后，obj 的 object 多了一个 3：d，长度也变成了 4

类数组：属性要为索引（数字）属性，**必须要有 length 属性**，最好加上 push 方法。

例：如果给一个对象加上 splice 方法，那么这个对象就长得像数组了。但他仍然是对象，但是可以当做数组来用，需要自己添方法。（如下图）

```
var obj = {
    "0" : 'a',
    "1" : 'b',
    "2" : 'c',
    "length" : 3,
    "push" : Array.prototype.push,
    "splice" : Array.prototype.splice
}
```

```
> obj
< ▶["a", "b", "c"]
```

```
Array.prototype.push = function (target){
    this[this.length] = target;
    this.length ++;
}
```

如果对象 obj 调用这个方法，那么 this 变成了 obj

```
Array.prototype.push = function (target){
    obj[obj.length] = target;
    obj.length ++;
}
```

例阿里巴巴题目，问这个 obj 长什么样子？

```
var obj = {
    "2" : "a",
    "3" : "b",
    "length" : 2,
    "push" : Array.prototype.push
}
obj.push('c');
obj.push('d');
obj -- >
```

答案：

```
> obj
< ▶Object {2: "c", 3: "d", length: 4}
```

关键点在 length 上面，根据 length 改变而改变，走一下 length，即：

```
Array.prototype.push = function (target){
    this[obj.length] = target;
    this.length ++;
}
```

```
Array.prototype.push = function (target){
    obj[obj.length] = target;
    obj.length ++;
}
```

例

```
var obj = {
    "1" : 'a',
    "2" : 'c',
    "3" : 'd',
    "length" : 3,
    "push" : Array.prototype.push
}
obj.push('b');
```

答案："1"："a"，"2"："c"，"3"："b"，"length"：4

例

```
var obj = {
    "0" : "a",
    "1" : "b",
    "2" : "c",
    name : "abc",
    age : 123,
    length : 3,
    push : Array.prototype.push,
    splice : Array.prototype.splice
}
```

```
> obj
< ▶["a", "b", "c"]
> obj.name
< "abc"
> obj.age
< 123
```

```
> obj.length
< 3
```

```
> for(var prop in obj) { console.log(obj[prop]) }
a                                                    VM2805:1
b                                                    VM2805:1
c                                                    VM2805:1
abc                                                  VM2805:1
123                                                  VM2805:1
3                                                    VM2805:1
function push() { [native code] }                    VM2805:1
function splice() { [native code] }                  VM2805:1
< undefined
```

作业 1、封装 type 方法达到下面效果

```
// 封装type
typeof([]) -- array
typeof({}) -- object
typeof(funtion) -- object
typeof(new Number()) -- number Object
typeof(123) --- number
```

```
> Object.prototype.toString.call(new Number(123))
< "[object Number]"
> Object.prototype.toString.call(123)
< "[object Number]"
```

2、数组去重：要求在原型链上编程→

```
Array.prototype.unique = function () {
}
```

```
var arr = [1,1,1,1,0,0,0,'a','b','a','b'];
arr.unique() -- > [1,0,'a','b'];
```

作业答案：

1、封装 type（这个方法是一个工具类方法，可以存在库里面），区分 typeof 方法

1）先分类，原始值，引用值

2）区分引用值，先判断是不是 null

数组，对象，包装类(new number)会返回 object，通过 Object.prototype.toString

```
function type(target) {
    var template = {
        "[object Array]" : "array",
        "[object Object]" : "object",
        "[object Number]" : "number - object",
        "[object Boolean]" : 'boolean - object',
        "[object String]" : 'string - object'
    }

    if(target === null) {
        return null;
    }
```

```
> type(123)
"number"
> type(new Number(123))
"number - object"
```

```
> type([])
"array"
> type(null)
null
> type(undefined)
"undefined"
```

```
    if(typeof(target) == 'function') {
        return 'function';
    }else if(typeof(target) == "object") {
        var str = Object.prototype.toString.call(target);
        return template[str];
    }else{
        return typeof(target);
    }
```

去掉

简化写法

```
function type(target) {
    var template = {
        "[object Array]" : "array",
        "[object Object]" : "object",
        "[object Number]" : "number - object",
        "[object Boolean]" : 'boolean - object',
        "[object String]" : 'string - object'
    }
    if(target === null) {
        return null;
    }
    if(typeof(target) == "object") {
        var str = Object.prototype.toString.call(target);
        return template[str];
    }else{
        return typeof(target);
    }
}
```

进一步**简化**

```
function type(target) {
    var ret = typeof(target);
    var template = {
        "[object Array]" : "array",
        "[object Object]" : "object",
        "[object Number]" : "number - object",
        "[object Boolean]" : 'boolean - object',
        "[object String]" : 'string - object'
    }
    if(target === null) {
        return "null";
    }else if(ret == "object") {
        var str = Object.prototype.toString.call(target);
        return template[str];
    }else{
        return ret;
    }
}
```

2、数组去重（利用对象的特性做数组去重，去重就是去掉重复）

思路：写一个对象，把数组的每一位当做对象的属性名。利用对象的特性（同一属性名不可以出现两次），先把第一位当做属性名添加进去，属性值随便写个，再看第二位，如果第二位在对象里面有属性名，就看下一位，如果对象里面没有属性名，就把这个值添加进去当属性名。只看对象的属性名，就去重了，这个方法叫 hash。

思考过程见下面，**答案**见右侧

```
var arr = [1,1,1,1,1,2,2,2,2,2,1,1,1,1];

// var obj = {
//     1 : "abc",
//     2 : "abc"
// }
// obj[1] --> undefined;
// obj[1] --> 'abc'
// obj[2] --> undefined;
```

```
Array.prototype.unique = function () {
    var temp = {},
        arr = [],
        len = this.length;
    for(var i = 0; i < len; i ++ ) {
        if(!temp[this[i]]) {
            temp[this[i]] = this[i];
            arr.push(this[i]);
        }
    }
    return arr;
}
```

当数组有0时就有bug

```
Array.prototype.unique = function () {
    var temp = {},
        arr = [],
        len = this.length;
    for(var i = 0; i < len; i ++ ) {
        if(!temp[this[i]]) {
            temp[this[i]] = "abc";
            arr.push(this[i]);
        }
    }
    return arr;
}
```

```
> arr.unique()
▶ [1, 2]
```

每一圈循环都要 this.length; 写成 var len = this.length;这样直接放值能少一些运算。

[this[i]]代表数组的第几位。 “abc” 可以随便填值（要求是字符串，但是不为 false），但是填[this[i]]= [this[i]];在数组有 0 的时候就会有 bug。

if 里面取到值什么都不敢，取到 undefined 才开始操作，所以写非！，没有值，才处理

复习：包装类

引用值就是一种对象（泛泛的概括类对象），包括数组，函数，对象。在内存里面存储。原始值不能有属性和方法，引用值才可以有。但是经过包装类，原始值就能有属性和方法。

通过原始值访问属性和方法，系统为了让语法好用，不报错，系统会帮我们进行一个 js 内部机制包装类。

例

```
var str = "abc";

//new String('abc').length
console.log(str.length);
```

思路：即隐式的 new String 构造出一个字符串对象，然后把字符串内容与原来保持一致 new String（‘abc’），因为我们进行了 str.length 操作，那么他也加上.length，变成了隐式 new String（‘abc’).length。这里虽然写的是 console.log(str.length)，实际上执行的是 console.log(new String（‘abc’).length)　这样隐式的执行过程就是包装类。

例

```
var num = 123;
num.abc = 'abc';
//new Number(num).abc = 'abc';  --> delete
//
//new Number(num).abc
console.log(num.abc);
```

是两个new

```
undefined
```

答案 undefined **思路**：当 num.abc =　“abc”时，系统会进行包装类，隐式的 new Number(num).abc =　“abc”；执行完这一步以后就会 delet 掉这个隐式的值，并不保留。等下一步又遇到 num.abc 时，又隐式的 new 了一个 number。但是这个和上一个是两个 new Number，是两个彼此独立的对象。

new.Number(123).abc 和 var num = new Number(123); num.abc 是一样的

复习：原型

例任何函数上都有原型，包括构造函数，这是一个构造函数,原型需要基于构造函数，没有原型的构造函数没有意义，任何一个函数都会有 prototype

```
Person.prototype.lastName = 'deng';
function Person() {
    //var this = {
    //
    //    __proto__ : Person.prototype
    //}
}
var person = new Person();
console.log(person.lastName);
```

复习：create

Object.create( );是创建的对象，对象必须要有原型，Object.create( );需要指定创建对象的原型是谁，括号里面就要填谁（所以括号里面一定要填值）

例

```
var demo = {
    lastName : "deng"
}

var obj = Object.create(demo);
obj = {
    __proto__ : demo
}
```

Object.create(prototype,definedProperty)还能填第二个参数。第一个填的 prototype 表示你的原型是谁，第二个参数 definedProperty 是特性。（可读可写都是特性）

例

```
var num = 123;
```

```
> delete num
< false
```

```
> delete window.num
< false
```

这个 num 算 window 的属性。写在全局的属性,一旦经历了 var 的操作,所得出的属性 window,这种属性叫做不可配置的属性。不可配置的属性,delete 不掉。

直接增加的属性卡叫可配置属性，delete 只能删除可配置的属性

例

```
var obj = {

}
obj.num = 234;
```

```
> delete obj.num
< true
> obj
< ▶Object {}
```

例直接在控制台操作对比，发现 var 过的属性是不可配置的属性,delete 不掉

```
> window.num = 123;
< 123
> delete window.num
< true
> window.num
< undefined
```

```
> var num = 123;
< undefined
> window.num
< 123
> delete window.num
< false
> window.num
< 123
```

复习：this　call

1.预编译　this==>window

2.谁调用的，this 指向谁

3.call 和 apply 能改变 this 指向

4.全局　this ==>window

例注释掉的是预编译的过程

```
function test() {
    var num = 123;
    function a () {

    }
}
test() -- > AO {
    arguments : {},
    this : window,
    num : undefined,
    a : function ( ){}
}
```

例 test( );完全等于 test.call( );执行。其实 test( )执行会内部转换成 test.call( );执行

```
function test() {
    console.log(this);
}
test();
```

```
Window {speechSynthesis: SpeechSynthesis, caches: CacheStorage, localStorage: Storage, sessionStorage: Storage, webkitStorageInfo: DeprecatedStorageInfo…}
```

例如果我们在 test.call( );里面传值，第一个值就会作为函数执行时的 this 环境

```
function test() {
    console.log(this);
}
// test();
test.call({name : "deng"});
// test() -- > AO {
//     arguments : {},
//     this : {name: "deng"},
//     num : undefined,
//     a : function ( ){}
// }
```

```
Object {name: "deng"}
```

例

```
var name = 'window';
var obj = {
    name : "obj",
    say : function () {
        console.log(this.name);
    }
}

obj.say();
```

打印obj

```
var name = 'window';
var obj = {
    name : "obj",
    say : function () {
        console.log(this.name);
    }
}

obj.say.call(window);
```

打印window

obj.say.call(window);有 call 就打破一切规则，call()里面传的是谁，就是谁

例

```
var name = 'window';
var obj = {
    name : "obj",
    say : function () {
        console.log(this.name);
    }
}

var fun = obj.say;
fun();
```

只能走预编译

```
window
```

var fun = obj.say 相当于 varfun= say：function(){}里面的函数体

fun ( )；相当于让 say:function 在这函数在全局范围内自调用，不是谁调用的，就只能走预编译，this 就是 window

例

```
var name = 'window';
var obj = {
    name : "obj",
    say : function () {
        console.log(this.name);
    }
}

var fun = obj.say;
fun.call(obj);
```

打印obj

例想让 Person 实现 Student 的功能

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

function Student(name, age, sex) {
    //var this = Object.create(Student.prototype);
    this.name = name;
    this.age = age;
    this.sex = sex;
}

var student = new Student('cheng', 18, 'male');
```

简化成Person.call(this, name, age);

```
function Student(name, age, sex) {
    //var this = Object.create(Student.prototype);
    Person.call(this, name, age);
    this.sex = sex;
}
```

简化

复习：闭包

闭包表象：一个函数套着另外一个函数，你把被嵌套的函数保存到套他的函数外面（a 套着 b，你把 b 弄出 a 里面），就形成了闭包（不一定要 return）

例右边两种方法都能实现闭包

```
function a () {
    function b() {

    }
    return b;
}
```

```
var obj = {}
function a () {
    function b() {

    }
    obj.fun = b;
}
```

例

```
var obj = {}
function a () {
    var aa = 123;
    function b() {
        console.log(aa);
    }
    obj.fun = b;
}
a();
```

```
> obj.fun()
  123
< undefined
```

例

```
var a = 123;
// this call
go {
    a : 123
}
ao{
    a : undefined,
}
function test() {
    a = 1;
    var a;
}
test();
```

复习：构造函数

通过构造函数构造对象的时候用 new，我执行函数的时候就不用 new

构造对象必须是 new 加上构造函数执行（如 person();）才能构造出对象

有了 new 之后，才会发生两步隐式变化（var this = {};　return this）

例

```
function Person() {
    //var this = {}
    this.name = 'abc';
    this.age = 123;
    //return this;
}

var person = new Person();
```

没有 var person = new Person（ ）；只 Person 会走预编译，此时 this 指向 window

例

私有化属性看不到 var money=100;

外部看不到 var money

就是闭包

```
function Person(name) {
    //var this = {
    //
    //  makeMoney : function () {}
    //  offer : function () {}
    //}
    var money = 100;
    this.name = name;
    this.makeMoney = function () {
        money ++;
    }
    this.offer = function () {
        money --;
    }
    //return this;
}

var person = new Person();
```

例

```
var inherit = (function () {

    var F = function () {};
    return function(Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
    }

}())
```

立即执行函数执行完就成下面这样了

```
var inherit = function(Target, Origin) {
        F.prototype = Origin.prototype;
        Target.prototype = new F();
    }
```

例引用值也能进行类型转换

```
> [] + ""
< ""
```

```
> [] + 2;
< "2"
```

```
> Number([])
< 0
```

```
> [] == [];
< false
```

```
> String([]) + 1
< "1"
```

数组不等于数组，因为里面的地址不一样

复习：克隆

```
var obj = {
    name : "abc"
}

var obj1 = {

}

for(var prop in obj) {
    obj1[prop] = obj[prop];
}
```

浅克隆：

当拷贝引用值的时候就不行了

```
var obj = {
    name : "abc",
    card : ['visa','master']
}

var obj1 = {

}

obj1['card'] = obj['card']
```

这样浅克隆，克隆的是地址，缺点是你改我也改

深度克隆解决的就是引用值

```
var obj = {
    name : "abc",
    wife : {
        name : "xiaoliu",
        son : {
            name : "xiaodeng"
        }
    }
}
```

深度克隆不拷贝地址，是新建一个来拷贝对象

```
var obj1 = {
    name : obj.name,
    wife : {
        name : obj.wife.name,
        son : {
            name : obj.wife.son.name
        }
    }
}
```

```
var obj = {
    name : "abc",
    wife : {
        name : "xiaoliu",
        son : {
            name : "xiaodeng"
        }
    }
}

var obj1 = {

}
```

这是引用值

```
> undefined < 1
< false
```

null 和 undefined 不能和数字进行比较，不会进行类型转换，他们不作为比较值存在

例下面考私有化变量

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

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

答案：打印 1,2,1

例

```
(function (x) {
    delete x;
    return x;
})(1);
```

答案 1，删不掉 x，该是什么是什么

例

```
function test() {
    concole.log(typeof(arguments));
}

test();
```

答案：返回 object

例

```
var h = function a() {
    return 23;
}

console.log(typeof a());
error : a is not a defined;
```

答案报错

例：选择你熟悉的一种方式实现 JavaScript 对象的继承

答案：声明模式

例：实现 object 类型的 clone()方法

答案：深度克隆

例

12、尝试优化以下代码，使得代码看起来更优雅（5 分）

```
functiongetDay(day){
    switch(day){
        case 1:
            document.write("Monday");  break;
        case 2:
            document.write("Tuesday");  break;
        case 3:
            document.write("Wednesday");  break;
        case 4:
            document.write("Thursday");  break;
        case 5:
            document.write("Friday");  break;
        case 6:
            document.write("Saturday");  break;
        case 7:
            document.write("Sunday");  break;
        default:
            document.write("Error");
    }
}
```

答案：可以放数组里面，也可以放类数组里面

```
function retDate(date) {

    var arr = ['一','二','三'];
    var ret = arr[date - 1];
    if(ret === undefined) {
        return 'error';
    }else{
        return ret;
    }
}
```

seo 是搜索引擎优化

例：HTML 布局实现：头和尾固定，中间自适应。
答案：三栏布局，头尾是指左右
例：在页面中增加一个 div（宽度 400px，高度 400px，背景颜色蓝色，边框颜色红色），该 div 要求在页面中居中显示。
例：简单画图描述 CSS 盒模型
例：css 中的选择器有哪些？
答案：id 选择器是 xxx，先说类型在举例
例：JavaScript 中有哪些数据类型？
答案：原始值里面有 XXX，引用值里面有 XXXX
例：什么是 rem 布局？
例：html 顶部的 DOCTYPE 有什么作用？有什么影响？
例：Display 的参数值及其含义
例：描述一下盒模型
答案：需要讲两种才满分
例：css3 可用伪类都有那些？
例：position 属性有哪些值，有什么区别？
例：如何创建一个 div，并添加到页面里。
例：写一个正则表达式，检验字符串首尾是否含有数字
例：跨域请求数据的方法都有哪些？
例：编写一个类和类继承，类名为 Person，含有属性 name，age，含有方法$。一个 student 类，继承自 Person，自由属性 score，自有方法 study
（类指的就是构造函数）

作业：
1、一个字符串[a-z]组成，请找出该字符串第一只出现一次的字母。
2、字符串去重
提示

```
'pqoiwuerpaoiqyweproiuqwpoeirupqiwyepoiutqpoweuropquweporiyqpwoeiytpoiqwueoruqwpoeibyrqwiuytoiquwyerpoiuqwpoeiru';
```

```
> var str = "abc";
< undefined
> str[0]
< "a"
> str.charAt(0)
< "a"
```

都是字符串第一位的意思

```
> var str = "abc";
< undefined
> str.split("")
< ▶ ["a", "b", "c"]
```

打散

try…catch 防止我们报错

try 花括号{里面会正常执行，但是遇到 b 报错时 b 就执行不出来，后面的代码 c 就不执行了，但是外面的代码 d 还能执行}catch(e)，这个 e 可以随便写，写 abc 都可以，也是个形参

例

```
try{
    console.log('a');
    console.log(b);
    console.log('c');

}catch(e) {

}

console.log('d');
```

报错后，不执行后续的代码，但是依然可以执行外面的d

结果a，d

```
a
d
```

在 try 里面的发生错误，不会执行错误后的 try 里面的代码

catch

例

```
try{
    console.log('a');
    console.log('b');
    console.log('c');

}catch(e) {
    console.log('e');
}

console.log('d');
```

```
a
b
c
d
```

```
try{
    console.log('a');
    console.log(b);
    console.log('c');

}catch(e) {//error  error.message error.name --> error
    console.log('e');
}

console.log('d');
```

如果 try 里面的代码不出错，在 catch 里面的代码就不执行；
如果 try 里面的代码出错，catch 负责补抓到错误信息封装到里面（error.massage error.name），错误对象只有 message 和 name。

例

```
try{
    console.log('a');
    console.log(b);
    console.log('c');

}catch(e) {//error  error.message error.name --> error
    console.log(e.message + " " + e.name);
}

console.log('d');
```

```
a
b is not defined ReferenceError
d
```

例

```
try{
    console.log('a');
    console.log(b);
    console.log('c');

}catch(e) {//error  error.message error.name --> error
    console.log(e.name + " : " + e.message);
}

console.log('d');
```

```
a
ReferenceError : b is not defined
d
```

try{}catch(e) {}finally{}

Error.name 的六种值对应的信息：

（前面是错误名称，后面是错误信息）

1. EvalError：eval()的使用与定义不一致

//eval 是不被允许使用的

2. RangeError：数值越界

3. ReferenceError：非法或不能识别的引用数值

//未经声明就使用，没有定义就使用

4. SyntaxError：发生语法解析错误

// Syntax 是语法解析（ ）

5. TypeError：操作数类型错误

6. URIError：URI 处理函数使用不当

//引用地址错误

大部分都是 3 和 4 这种错误

伪代码也可以写了，可以写 var 老邓 = 123；这就是伪代码

var str = avs ==> **ReferenceError**

es5.0 严格模式

（这章就是讲 es3.0 和 es5.0 产生冲突的部分）

浏览器是基于 es3.0 和 es5.0 的新增方法使用的。

如果两种发生了冲突，就用 es3.0。

```
"use strict";
```

es5.0 严格模式是指 es3.0 和 es5.0 产生冲突发部分就是用 es5.0，否则就用 es3.0。

es5.0 严格模式的启动 “use strict”；

用法在整个页面的**最顶端**写 “use strict”，可以写在全局的最顶端，也可以写在某函数（局部）的最顶端，推荐使用局部的。

例在 es5.0 不能用 argument.callee，但是 es3.0 可以用

```
//es5.0严格模式的启动

"use strict";

function test() {
    console.log(arguments.callee);
}


test();
```

```
▶ Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them    lesson17.html:23
    at test (lesson17.html:23)
    at lesson17.html:27
```

例 function test( ){ }里面加 **“use strict” ;**是局部启动严格模式

```
function demo() {
    console.log(arguments.callee);
}

demo();

function test() {
    "use strict";
    console.log(arguments.callee);
}


test();
```

这个是es3.0

只有这个使用es5.0的方法

```
function demo() {
    console.log(arguments.callee);
}                                                    lesson17.html:29
▶ Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them    lesson17.html:36
    at test (lesson17.html:36)
    at lesson17.html:41
```

“use strict” ;

不再兼容 es3 的一些不规则语法。使用全新的 es5 规范。两种用法：

1、全局严格模式

2、局部函数内严格模式（推荐）

就是一行字符串，不会对不兼容严格模式的浏览器产生影响。

不支持 with，arguments.callee，function.caller，变量赋值前必须声明，局部 this 必须被赋值(Person.call(null/undefined) 赋值什么就是什么),拒绝重复属性和参数

例

浏览器升级到 es5.0 才好用

写 strict( );就有风险

写成 **“use strict” ;**有一个向后兼容的作用

```
"use strict";
strict();
```

写字符串模式

老版本不报错，新版本能使用

例 es5.0 严格模式不让使用 with。

with 可以改变作用域链

with ( ) 括号里面的代码会按照正常顺序执行，但是如果在括号里面添加了对象，就会把对象当做 with 要执行的代码体的作用域链的最顶端（最直接的最近的 AO）。

例下面这个 with 看到的就是 var obj 的 name （with 改变作用域链）

```
var obj = {
    name : "obj"
}

var name = 'window';

function test() {
    var age = 123;
    var name = 'scope';
    with(obj) {
        console.log(name);
        console.log(age);
    }
}
test();
```

```
obj
123
```

```
var obj = {
    name : "obj",
    age : 234
}

var name = 'window';

function test() {
    var age = 123;
    var name = 'scope';
    with(obj) {
        console.log(name);
        console.log(age);
    }
}
test();
```

```
obj
234
```

例命名空间应该像下面这样用的，with 可以简化代码

```
var org = {
    dp1: {
        jc : {
            name : 'abc',
            age : 123
        },
        deng : {
            name : "xiaodeng",
            age : 234
        }
    },
    dp2: {

    }
}

with(org.dp1.jc) {

    console.log(name);
}
```

```
document.ATTRIBUTE_NODE
    createElement
    createElementNS
    createEvent
    createExpression
    createNSResolver
    createNodeIterator
    createProcessingInstruction
```

```
with(document) {
    write('a');
}
```

就是document.write

这也是 with 的运用方式

用 with 表示 document.write

with 过于强大，可以改作用域链，失去效率，所以 es5.0 不能用

例 **arguments.callee**

```
function test() {
    console.log(arguments.callee);
}
test();
```

```
▶ Uncaught TypeError: 'caller', 'callee', and 'arguments' properties may not be accessed on strict mode functions or the arguments objects for calls to them
    at test (lesson17.html:47)
    at lesson17.html:49
```

例 **arguments.callee 在 es5.0 严格模式下报错**

```
"use strict";

function test() {
    console.log(test.caller);
}
function demo() {
    test();
}

demo();
```

```
▶ Uncaught TypeError: 'caller' and 'arguments' are restricted function properties and cannot be accessed in this context.
    at test (lesson17.html:47)
    at demo (lesson17.html:50)
    at lesson17.html:53
```

es5.0 严格模式中：变量赋值前必须声明，局部 this 必须被赋值(Person.call(null/undefined) 赋值什么就是什么),拒绝重复属性和参数(this 不在指向 window)

例

```
var a = b = 3;
```

```
▶ Uncaught ReferenceError: b is not defined
```

例

```
a = 3;
```

```
▶ Uncaught ReferenceError: a is not defined
```

例局部 this 必须被赋值(Person.call(null/undefined) 赋值什么就是什么)

```
"use strict";

function test () {
    console.log(this);
}

test();
```

```
undefined
```

例 该对象的 constructor 是 Test，

```
"use strict";

function Test () {
    console.log(this);
}

new Test();
```

```
▶ Test {}
```

construction

例

```
"use strict";

function Test () {
    console.log(this);
}

Test.call({});
```

```
▶ Object {}
```

例

```
"use strict";

function Test () {
    console.log(this);
}

Test.call(123)
```

```
123
```

例在 es3.0 里面是不能这样的

```
// "use strict";

function Test () {
    console.log(this);
}

Test.call(123)
```

```
▶ Number {[[PrimitiveValue]]: 123}
> new Number(123) 这是包装类的显示形式
▶ Number {[[PrimitiveValue]]: 123}
```

例全局的 this 指向 window

```
"use strict";

console.log(this);
```

全局范围

```
指向window                                    lesson17.html:46
Window {speechSynthesis: SpeechSynthesis, caches:
CacheStorage, localStorage: Storage,
sessionStorage: Storage, webkitStorageInfo:
DeprecatedStorageInfo…}
```

例**拒绝重复属性和参数（this 不在指向 window）**

```
function test(name, name) {
    console.log(name);
}
test(1);
```

```
undefined
```

```
function test(name, name) {
    console.log(name);
}
test(1, 2);
```

```
2
```

在 es3 里面重复的参数是不报错的，但是在 es5 里面是会报错的

例

```
'use strict';
function test(name, name) {
    console.log(name);
}
test(1, 2);
```

```
▶ Uncaught SyntaxError: Duplicate parameter name not allowed in this context    lesson17.html:44
```

例重复的**属性名**在 es5 也不行，但是不报错（后面会覆盖上面的）

```
'use strict';


var obj = {
    name : "123",
    name : "234"
}
```

```
>
```

eval 很强大，**能把字符串当成代码来执行**

但是约定俗成在 es3.0 中都不能使用 eval。eval 是魔鬼，因为会改变作用域

例

```
"use strict";
var a = 123;
eval('console.log(a)');
```

```
123
```

例

```
// es3.0 都不能eval();  eval 是 魔鬼
var global = 100;
function test() {
    global = 200;
    eval('console.log(global)');
}

test();
```

```
200
```

如果改变 global，他改变的是全局的

当情况不同，eval 改变的作用域是不同的

eval 还有自己独立的作用域

理解：可以把回调函数理解为先定义好了函数，执行的时候再回头调用

dom 这章之前是笔试面试的重点

什么是 DOM

1.DOM — > Document Object Model(文档对象模型)

2.DOM 定义了表示和修改文档所需的方法（对象、这些对象的行为和属性以及这些对象之间的关系。）DOM 对象即为宿主对象，由浏览器厂商定义，用来操作 html 和 xml 功能的一类对象的集合。

也有人称 DOM 是对 HTML 以及 XML 的标准编程接口。

```
<student>
    <name>老邓</name>
    <age>100</age>
</student>  数据集合
```

xml ==> xhtml ==> html

xml 是最早的版本，xml 里面的标签是可以自定义的，被 js 里面的 Jason 取代了

dom 不能改变 css 样式表,可以间接改变行间样式的 css

例：说的改变不了 css 是指改变不了 css 的样式表，但是可以通过间接方式改变 html 的行间样式来改变

```
<body>
    <div></div>
    <script type="text/javascript">

        var div = document.getElementsByTagName('div')[0];
        div.style.width = "100px";
        div.style.height = "100px";
        div.style.backgroundColor = "red";


    </script>
```

1、找到 html 的方法：如 document.getElementsByTagName（‘div’）就能把所有的 div 都选出来。

如果想拿到第一个 div，写成 document.getElementsByTagName（‘div’）后面就要加一个[0];就可以实现了

2、div.style 选出来代表行间样式，选出来的就是 dom 对象

3、js 不能写-，只能用小驼峰方法写。如 background-color 写成 backgroundColor

例

有 dom 操作以后就变成动态交互可以操作的了（你动一下，他给你一个反应）

div.onclick 是加一个交互效果的事件监听

```
<body>
    <div></div>
    <script type="text/javascript">
        //dom 对象
        var div = document.getElementsByTagName('div')[0];
        div.style.width = "100px";
        div.style.height = "100px";
        div.style.backgroundColor = "red";

        div.onclick = function () {
            this.style.backgroundColor = 'green';
            this.style.width = "200px";
            this.style.height = "50px";
            this.style.borderRadius = "50%";
        }
```

例实现点击一下改变一下

```
<div></div>
<script type="text/javascript">
    //dom 对象
    var div = document.getElementsByTagName('div')[0];
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";


    var count = 0;
    div.onclick = function () {

        count ++;
        if(count % 2 == 1) {
            this.style.backgroundColor = "green";
        }else{
            this.style.backgroundColor = 'red';
        }


    }
```

例写个选项卡，点第一个按钮出现第一对应的信息，点第二个按钮，第一个消失，出现第二个的信息

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">

        .content{
            display: none;
            width:200px;
            height:200px;
            border:2px solid red;
        }
        .active {
            background-color: yellow;
        }

    </style>
</head>
```

下面的 button 就是个按钮

```
<body>
    <div class="wrapper">
        <button class="active">111</button>
        <button>222</button>
        <button>333</button>
        <div class="content" style="display:block">1111</div>
        <div class="content">邓哥..2222.</div>
        <div class="content">3333</div>
    </div>
    <script type="text/javascript">
```

```
<script type="text/javascript">

    var btn = document.getElementsByTagName('button');
    var div = document.getElementsByClassName('content');
    for(var i = 0; i < btn.length; i++) {

        btn[i].onclick = function () {

            for(var j = 0; j < btn.length; j++) {
                btn[j].className = "";
                div[j].style.display = "none";
            }

            this.className = "active";
            div[i].style.display = "block";

        }

    }
```

最好用下面这种立即执行函数的写法：

```
<script type="text/javascript">

    var btn = document.getElementsByTagName('button');
    var div = document.getElementsByClassName('content');
    for(var i = 0; i < btn.length; i++) {

        (function (n) {
            btn[n].onclick = function () {
                for(var j = 0; j < btn.length; j++) {
                    btn[j].className = "";
                    div[j].style.display = "none";
                }
                this.className = "active";
                div[n].style.display = "block";

            }
        }(i))
    }
```

例写个小方块应用

document.body.appendChild(div);意思是在 body 里面放个 div

setInterval(function () {},100);是一个定时器功能，意思是每隔 100 毫秒就执行一次

```
var div = document.createElement('div');

document.body.appendChild(div);
div.style.width = "100px";
div.style.height = "100px";
div.style.backgroundColor = "red";
div.style.position = "absolute";
div.style.left = "0";
div.style.top = "0";

setInterval(function (){

    div.style.left = parseInt(div.style.left) + 2 + "px";
    div.style.top = parseInt(div.style.top) + 2 + "px";
}, 50);
```

把上面 setInterval(function){}改成以下样式：

```
var speed = 1;
setInterval(function (){

    speed += speed/7;
    div.style.left = parseInt(div.style.left) + speed + "px";
    div.style.top = parseInt(div.style.top) + speed + "px";
}, 10);
```

把最上面 setInterval(function){}改成以下样式，可以让定时器停止

```
var speed = 1;
var timer = setInterval(function (){

    speed += speed/20;
    div.style.left = parseInt(div.style.left) + speed + "px";
    div.style.top = parseInt(div.style.top) + speed + "px";
    if(parseInt(div.style.top) > 200 && parseInt(div.style.left
        ) > 200) {
        clearInterval(timer);
    }
}, 10);
```

例

```
var div = document.createElement('div');

document.body.appendChild(div);
div.style.width = "100px";
div.style.height = "100px";
div.style.backgroundColor = "red";
div.style.position = "absolute";
div.style.left = "0";
div.style.top = "0";
```

```
document.onkeydown = function(e) {
    switch(e.which) {
        case 38:
            div.style.top = parseInt(div.style.top) - 5 + "px";
            break;
        case 40:
            div.style.top = parseInt(div.style.top) + 5 + "px";
            break;
        case 37:
            div.style.left = parseInt(div.style.left) - 5 + "px
                ";
            break;
        case 39:
            div.style.left = parseInt(div.style.left) + 5 + "px
                ";


    }
}
```

例：现在要实现按住加速，不好写没写，看 js 运动课

```
var time = new Date().getTime();
document.onkeydown = function(e) {
        div.style.left = parseInt(div.style.left) + speed +
            "px";
}
new Date.getTime() - time
```

按住加速：就是这一次按下与下一次按下的速度间隔时间十分短暂，就认为他加速了。

上面**思路**：每一次按下的时候都记录一个新的时间片段，都减去上一个执行的时间片段，如果时间片段都小于一定的毫秒数的话，我们让一个计数器去++，当连续小于的时候，就让计数器连续++,如果++到一定数的时候，我们认为是连续按了，再按就加速了，让每一次按的时候都判断一下,如果时间间隔过大的话，就让计数器重新归 0。——在 js 运动课里面讲

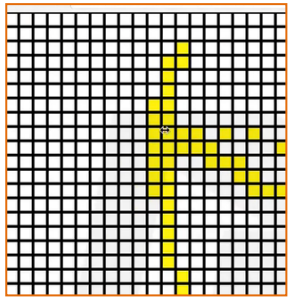
例 点了加速的按钮以后，移动的速度变快

```
<body>
    <button style="width:100px;height:50px;background:linear-gradient(
    to left, #999, #000, #432, #fcc)
    ;position:fixed;right:0;top:50%;text-align:center;line-
    height:50px;color:#fff;font-size:25px;font-family:arial;">加速</
    button>
    <script type="text/javascript">
        var btn = document.getElementsByTagName('button')[0];

        var div = document.createElement('div');

        document.body.appendChild(div);
        div.style.width = "100px";
        div.style.height = "100px";
        div.style.backgroundColor = "red";
        div.style.position = "absolute";
        div.style.left = "0";
        div.style.top = "0";
        var speed = 5;
```

```
var speed = 5;
btn.onclick = function ( ) {
    speed ++;
}
document.onkeydown = function(e) {
    switch(e.which) {
        case 38:
            div.style.top = parseInt(div.style.top) - speed + "
                px";
            break;
        case 40:
            div.style.top = parseInt(div.style.top) + speed + "
                px";
            break;
        case 37:
            div.style.left = parseInt(div.style.left) - speed +
                "px";
            break;
        case 39:
            div.style.left = parseInt(div.style.left) + speed +
                "px";
    }
}
```

例 box-sizing:border-box;启动了另外一种盒模型

```
        *{
            margin:0;
            padding:0;
        }

        li{
            box-sizing:border-box;
            float:left;
            width:10px;
            height:10px;
            border:1px solid black;
        }
        ul{
            list-style:none;
            width:200px;
            height:200px;
        }


    </style>
</head>
<body>
```

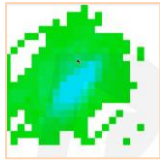


中间有<li img-date=" 0" >
*400，onmouseover 是鼠标滑过变成

```
    <li img-date="0"></li>
</ul>

<script type="text/javascript">

    var ul = document.getElementsByTagName('ul')[0];

    ul.onmouseover = function (e) {
        var event = e || window.event;
        var target = event.target || event.srcElement;

        target.style.backgroundColor = "rgb(255, 255," + target.
            getAttribute('img-date') +")";
        target.setAttribute('img-date', parseInt(target.get
            Attribute('img-date')) + 6);

    }
```

把左边的案例，如图去掉边线

```
<style type="text/css">
    *{
        margin:0;
        padding:0;
    }

    li{
        box-sizing:border-box;
        float:left;
        width:10px;
        height:10px;
        /*border:1px solid black;*/
    }
    ul{
        list-style:none;
        width:200px;
        height:200px;
    }
```



emmet 插件提供的方法：(打字操作)

例 div*3

例 div.demo 就是直接设置了 class 名字 `<div class="demo"></div>`

例 div.demo#only `<div class="demo" id="only"></div>`

例 div.demo#only > p 就是让 div 下面包含一个子元素 p

```
<div class="demo" id="only">
    <p></p>
</div>
```

例 div.demo#only > p[style=' background-color:red;width:100px;height:100px;']

```
<div class="demo" id="only">
    <p style="background-color:red;width:100px;height:100px;"></p>
</div>
```

例 div>(p^span.demo)

```
<div>
    <p></p>
    <span class="demo" id="only"></span>
</div>
```

例加内容就用{}花括号

div>(p^span.demo{123})

```
<div>
    <p></p>
    <span class="demo" id="only">123</span>
</div>
```

例 ul>li{$}*10 这里面的$代表变量 ==========>

```
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
    <li>6</li>
    <li>7</li>
    <li>8</li>
    <li>9</li>
    <li>10</li>
</ul>
```

例 ul>li{$*2}*10 ============================>

```
<ul>
    <li>1*2</li>
    <li>2*2</li>
    <li>3*2</li>
    <li>4*2</li>
    <li>5*2</li>
    <li>6*2</li>
    <li>7*2</li>
    <li>8*2</li>
    <li>9*2</li>
    <li>10*2</li>
</ul>
```

可以看看 emmet 插件教程

在安装插件之前要按一个 package control，在百度里面找，复制代码后，按 ctrl+~再把复制的代码粘贴到这里，按回车就可以了

在 sublime text 里面 preferences 里面找到 package control ,在搜索框找到 install package 后打 emmet，Jsprettify（整理排序用的）

编辑器还有 vscode，webstrom，atom

DOM 基本操作(大部分都是类数组)——方法类操作

1.对节点的增删改查　　()括号里面都不用写.或#

查

查看元素节点

document 代表整个文档（如果给 html 标签上面再套一层标签就是 document）

document.getElementById() **//元素 id 在 Ie8 以下的浏览器，不区分 id 大小写，而且也返回匹配 name 属性的元素，通过 id 标识我们来选择这个元素，一一对应**

例

```
<div name="only">123</div>
<script type="text/javascript">

    var div = document.getElementById('only');
```

除了 id 以外，其余选择出来的都是一组，很少用 id 选择器

.getElementsByTagName()　**//标签名，这是一个类数组，最主流的用法，经常用**

例把页面里面所有的 div 都选择出来

```
<div id="only"></div>
<script type="text/javascript">

    // var div = document.getElementById('only');

    var div = document.getElementsByTagName('div')
```

```
> div
▶ [div#only]
> div.push(1)
▶ Uncaught TypeError: div.push is not a function
    at <anonymous>:1:5
```

从 dom 开始，我们所学的一切系统给我们生成的成组的东西，基本上都是类数组

例加个[0]就选中了第一个 div，如果不加 0，那个 div 表示的是一个数组，设置背景颜色就会报错

```
<div id="only">123</div>
<script type="text/javascript">

    // var div = document.getElementById('only');

    var div = document.getElementsByTagName('div')[0];
```

getElementsByName(); **//IE 不支持需注意，只有部分标签 name 可生效（表单，表单元素，img，iframe），不是在所有的浏览器都能用——开发一般不用**

例

```
<input name="fruit">
<script type="text/javascript">

    // var div = document.getElementById('only');

    var div = document.getElementsByName('fruit');
```

例把下面所有的 div 都拿出来

```
<div></div>
<div></div>
<div></div>
<p></p>
<script type="text/javascript">

    // var div = document.getElementById('only');

    var div = document.getElementsByTagName('div');
```

```
> div
▶ [div, div, div]
> div.length
3
```

```
> div[1]
  <div></div>
> div[1].innerHTML = 123
123
```

只拿出第二个 div，加上[1] ，或者在控制台上打 div[1]

例选择第一个 p 的方式如下

```
<div>
    <p></p>
</div>
<p></p>
<script type="text/javascript">

    // var div = document.getElementById('only');

    var p = document.getElementsByTagName('p')[0];
```

例只拿出 demo 的 p 标签写法如下：

```
<div>
    <p class="demo"></p>
</div>
<p></p>
<script type="text/javascript">

    // var div = document.getElementById('only');

    var p = document.getElementsByClassName('demo')[0];
```

注意哪怕整个文档只有一个 demo，也要加[0]，不然选出来的就是一个组

.getElementsByClassName()　**// 类名 ->缺点： ie8 和 ie8 以下的 ie 版本中没有，可以多个 class 一起，不是所有浏览器都能用**

.querySelector()　**// css 选择器，只能选一个，在 ie7 和 ie7 以下的版本中没有**

.querySelectorAll()　**// css 选择器，全选，选一组，在 ie7 和 ie7 以下的版本中没有**

.querySelectorAll()和.querySelector()选出来的元素不是实时的（是静态的），所以一般不用，其他的再怎么修改，跟原来的没有关系

id 选择器不能太依赖，一般当顶级框架存在，在 css 中一般用 class 选择器

query 是一个词条

例.querySelector()和.querySelectorAll()选出来的元素不是实时的，是静态的，是副本

```
<body>
    <div>
        <strong></strong>
    </div>
    <div>
        <span>
            <strong class="demo">123</strong>
        </span>
    </div>
    <script type="text/javascript">

        // var div = document.getElementById('only');

        var strong = document.querySelector('div > span strong.demo');
        var strong1 = document.querySelectorAll('div > span strong.demo
            ');
```

```
> strong
  <strong class="demo">123</strong>
> strong1
▶ [strong.demo]
```

例下面的都是实时的

```
<body>

    <div></div>
    <div class="demo"></div>
    <div></div>


    <script type="text/javascript">


        var div = document.getElementsByTagName('div');
        var demo = document.getElementsByClassName('demo')[0];
        var newDiv = document.createElement('div');
        document.body.appendChild(newDiv);
```

```
> div
▶ [div, div.demo, div, div]
```

例.querySelectorAll()选中了所有的 div，但是实时操作不能实时反馈

```
<div></div>
<div class="demo"></div>
<div></div>


<script type="text/javascript">


    var div = document.querySelectorAll('div');
    var demo = document.getElementsByClassName('demo')[0];
    var newDiv = document.createElement('div');
    document.body.appendChild(newDiv);
```

```
> demo.remove()
undefined
> div
▶ [div, div.demo, div]
```

DOM 基本操作——非方法类的操作

遍历节点树：(灵活，**兼容好**)——关系类的选择

parentNode → 父节点 (最顶端的 parentNode 为#document);

childNodes → 子节点们（直接的节点数）节点包括文本节点，属性节点

firstChild → 第一个子节点

lastChild → 最后一个子节点

nextSibling →后一个兄弟节点

previousSibling → 前一个兄弟节点

例 **parentNode** → **父节点**

```
<div>
    <span></span>
    <div>
        <p></p>
    </div>
</div>
```

div

span　　div

p

```
> strong.parentNode
▶ <div>…</div>
> strong.parentNode.parentNode
▶ <body>…</body>
> strong.parentNode.parentNode.parentNode
  <html lang="en">
  ▶ <head>…</head>
  ▶ <body>…</body>
  </html>
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

```
> strong.parentNode.parentNode.parentNode.parentNo
  de
▶ #document
> strong.parentNode.parentNode.parentNode.parentNo
  de.parentNode
null
```

例 **childNodes** → 子节点们

```
<div>
    <strong>
        <span>1</span>
    </strong>
    <span></span>
    <em></em>
</div>
```

```
> div.childNodes.length
7
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

节点为什么是 7？

答案第一个节点是 后面的文本节点（空格回车），第二个节点是元素节点1，第三个节点是跟着的文本节点（空格回车），第四个节点是 ,第五个是跟着的文本节点(空格回车),第六个是，第七个是跟着的文本节点（空格回车）

节点的类型

后面的数字是调用 **nodeType** 返回的数字

元素节点 —— **1**

属性节点 —— 2 （基本没用，）

文本节点 —— **3**

注释节点 —— 8

document —— 9

DocumentFragment —— **11**

例

```
<div>
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
> div.childNodes
◀ ▶ [text, comment, text, strong, text, span, text]
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

答案七个，第一个节点是 后面的文本节点（空格回车），第二个节点是注释节点，第三个节点是跟着的文本节点（空格回车），第四个节点是元素节点，第五个是跟着的文本节点（空格回车），第六个是，第七个是跟着的文本节点（空格回车）

例

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

7个节点

```
> div.childNodes
◀ ▶ [text, comment, text, strong, text, span, text]
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

答案七个，123 和空格等是一个文本

例 **firstChild** 和 **lastChild**

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

7个节点

```
> div.firstChild
◀ "
    123
    "
> div.lastChild
◀ ▶ #text
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

例 **nextSibling** →后一个兄弟节点 ，**previousSibling** → 前一个兄弟节点

```
<div>
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
<script type="text/javascript">

    var strong = document.getElementsByTagName('strong')[0];
```

```
> strong.nextSibling
◀ ▶ #text
> strong.nextSibling.nextSibling
◀   <span></span>
> strong.nextSibling.nextSibling.nextSibling
◀ ▶ #text
> strong.nextSibling.nextSibling.nextSibling.nextS
  ibling
◀ null
> strong.nextSibling.previousSibling
◀   <strong></strong>
```

基于元素节点树的遍历（不含文本节点）

（**除 children 外，其余 ie9 及以下不兼容**）

parentElement -> 返回当前元素的父元素节点 (IE 不兼容)

例

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

```
> div.parentElement
◀ ▶ <body>…</body>
> div.parentElement.parentElement
◀   <html lang="en">
    ▶ <head>…</head>
    ▶ <body>…</body>
    </html>
> div.parentElement.parentElement.parentElement
◀ null
```

例 children -> 只返回当前元素的元素子节点

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
> div.children
< ▶ [strong, span]
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

例 node.childElementCount === node.children.length 当前元素节点的子元素节点个数(IE 不兼容)——基本不用，因为与 length 相等

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
> div.childElementCount
< 2        ||  等于
> div.children.length
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

例 firstElementChild -> 返回的是第一个元素节点(IE 不兼容)
lastElementChild -> 返回的是最后一个元素节点(IE 不兼容)

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
</div>
```

```
> div.firstElementChild
<   <strong></strong>
> div.lastElementChild
<   <span></span>
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

例 nextElementSibling / previousElementSibling ->返回后一个/前一个兄弟元素节点(IE 不兼容)

```
<body>

    <div>
        123
        <!-- This is comment -->
        <strong></strong>
        <span></span>
        <em></em>
        <i></i>
        <b></b>
    </div>
```

```
    <script type="text/javascript">

        var strong = document.getElementsByTagName('strong')[0];
```

```
> strong.nextElementSibling
<   <span></span>
> strong.nextElementSibling.nextElementSibling
<   <em></em>
> strong.nextElementSibling.nextElementSibling.previousElementSibling
<   <span></span>
> strong.nextElementSibling.nextElementSibling.previousElementSibling.previousElementSibling
<   <strong></strong>
> strong.nextElementSibling.nextElementSibling.previousElementSibling.previousElementSibling.previousElementSibling
< null
```

除 children 外，parentElement、node.childElementCount、firstElementChild、lastElementChild、nextElementSibling、 previousElementSibling 在 **ie9 及以下不兼容**

真正常用的就是 children，兼容性好

每一个节点的四个属性

1、nodeName

元素的标签名，以大写形式表示,只读，不能写

例 nodeName

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
    <em></em>
    <i></i>
    <b></b>
</div>
```

```
> div.childNodes[1].nodeName
"#comment"
```

```
> div.childNodes[3].nodeName = "abc"
"abc"
> div.childNodes[3].nodeName
"STRONG"
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

2、nodeValue

Text 文本节点或 Comment 注释节点的文本内容,可读写

例 nodeValue

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
    <em></em>
    <i></i>
    <b></b>
</div>
```

```
> div.childNodes[0].nodeValue
"
        123
    "
> div.childNodes[0].nodeValue = 234
234
> div.childNodes[0].nodeValue
"234"
> div.childNodes[0]
  "234"
```

```
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
```

```
> div.childNodes[1].nodeValue
" This is comment "
> div.childNodes[1].nodeValue = "That is comment";
"That is comment"
> div.childNodes[1]
  <!--That is comment-->
```

3、nodeType（最有用）

该节点的类型，只读返回这个 div 的所有的元素节点

数字是调用 nodeType 返回的数字

元素节点 —— 1

属性节点 —— 2 （基本没用，class=“demo”就是一个属性节点）

文本节点 —— 3

注释节点 —— 8

document —— 9

DocumentFragment —— 11

```
> div.childNodes[0].nodeType
3
> div.childNodes[3].nodeType
1
```

例把 div 下面所有的直接子元素节点挑出来，放在数组里面返回，不能用 children

```
<div>
    123
    <!-- This is comment -->
    <strong></strong>
    <span></span>
    <em></em>
    <i></i>
    <b></b>
</div>
```

```
var div = document.getElementsByTagName('div')[0];

function retElementChild(node) {

    //no children

    var temp = {

            length : 0,
            push : Array.prototype.push,
            splice : Array.prototype.splice

        },
        child = node.childNodes,
        len = child.length;

    for(var i = 0; i < len; i++) {

        if(child[i].nodeType === 1) {
            temp.push(child[i]);
        }

    }
    return temp;

}

console.log(retElementChild(div));
```

```
▶ [strong, span, em, i, b]
```

4、attributes

Element 节点的属性集合

例

```
<div id="only" class="demo"></div>   属性节点
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
```

```
> div.attributes
▶ NamedNodeMap {0: id, 1: class, length: 2}
> div.attributes[0]
  id="only"
> div.attributes[0].nodeType
2
```

```
> div.attributes[0].value
"only"
> div.attributes[0].name
"id"
> div.attributes[0].value = 'abc'
"abc"
> div
  <div id="abc" class="demo"></div>
```

```
> div.attributes[0].name = 'abc'
"abc"
> div
  <div id="only" class="demo"></div>
```

属性名不能改，属性值可以改，但是我们一般不用这种方法

我们一般用 getAttribute 和 setAttribute 去取

节点的一个方法 Node.hasChildNodes();——他有没有子节点，返回值是 true 或 false

例

```
<div id="only" class="demo">
    <span></span>
</div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
```

```
> div.hasChildNodes()
true
```

例

```
<div id="only" class="demo">
    <!-- this is comment -->
</div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
```

```
> div.hasChildNodes()
true
```

例

```
<div id="only" class="demo">
</div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
```

```
> div.hasChildNodes()
true
```

例当且仅当这种情况下是 false（没空格，没回车）

```
<div id="only" class="demo"></div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];
```

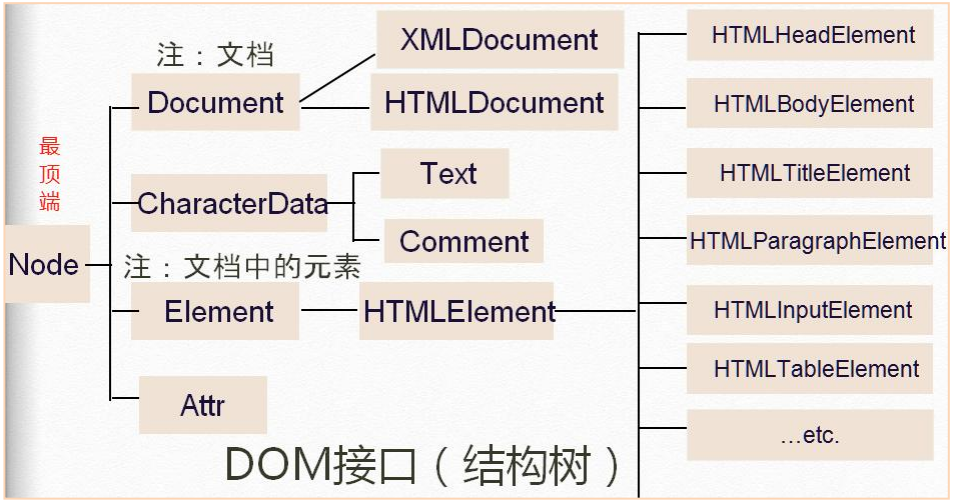
```
> div.hasChildNodes()
false
```

例属性的节点是 div 上面的，不是 div 里的

```
<div>
    <p></p>          七个节点
    <span></span>    两个元素节点
    <!-- this is comment -->
</div>
```

DOM 接口

dom 结构树代表的是一系列继承关系



例 Document 和 document 的关系

```
> Document
function Document() { [native code] }
```

可以理解成构造函数，但是不能new

```
> document
▶ #document
```

```
> Document.prototype.abc = 'abc';
"abc"
> document.abc
"abc"
```

```
document --> HTMLDocument.prototype --> Document.prototype
```

document 继承自 HTMLDocument.prototype。

HTMLDocument.prototype 继承自 Document.prototype

原型是不是一个对象？

```
HTMLDocument.prototype = {

    __proto__ : Document.prototype
}
```

例

```
document --> HTMLDocument.prototype --> Document.prototype
```

```
> HTMLDocument.prototype.bcd='123';
"123"
> document
▶#document
> document.bcd
"123"
```

```
> Document.prototype.abc='1';
"1"
> HTMLDocument.prototype.abc='2';
"2"
> document.abc
"2"
```

DOM 结构树中，Node 也是构造函数，comment 是注释

HTMLDocument 和 HTMLElement 实际上并列了对应的 XML ,但是因为不用了 XML 就省略了。HTMLHeadElement 就是 head 标签，其余类推。

例

```
<script type="text/javascript">


    // document --> HTMLDocument.prototype --> Document.
    prototype


    HTMLBodyElement.prototype.abc='demo';
    var body = document.getElementsByTagName('body')[0];
    var head = document.getElementsByTagName('head')[0];
```

```
> body.abc
"demo"
```

```
> head.abc
undefined
```

上面如果 HTMLElement.prototype.abc=’ demo’ ;定义，则右上方两个都能用

例看看一层返回的是什么东西

```
> document.__proto__
▶HTMLDocument {Symbol(Symbol.toStringTag):
"HTMLDocument"}
> document.__proto__.__proto__
▶Document {Symbol(Symbol.toStringTag):
"Document", Symbol(Symbol.unscopables): Object}
> document.__proto__.__proto__.__proto__
  Node {ELEMENT_NODE: 1, ATTRIBUTE_NODE: 2,
▶TEXT_NODE: 3, CDATA_SECTION_NODE: 4,
ENTITY_REFERENCE_NODE: 5…}
```

```
> document.__proto__.__proto__.__proto__.__proto__
▶EventTarget {Symbol(Symbol.toStringTag):
"EventTarget"}
> document.__proto__.__proto__.__proto__.__proto__
.__proto__
▶Object {}
```

例

```
> document.body
▶<body>…</body>
```

```
> document.body.toString()
"[object HTMLBodyElement]"
```

DOM 基本操作

1.getElementById 方法定义在 Document.prototype 上，即 Element 节点上不能使用。

2.getElementsByName 方法定义在 HTMLDocument.prototype 上，即非 html 中的 document 以外不能使用(xml document,Element)

3.getElementsByTagName 方法定义在 Document.prototype 和 Element.prototype 上

例选 div 里的 span

```
> span
<span>1</span>
```

```
<body>
    <div>
        <span>1</span>
    </div>
    <span></span>
    <script type="text/javascript">

        var div = document.getElementsByTagName('div')[0];
        var span = div.getElementsByTagName('span')[0];
```

开发中，经常利用先选择的父级，在他父级里面再次选什么元素来定位一个元素

例

```
var div = document.getElementsByTagName('*')[0];
```

通配符-选择所有

4.HTMLDocument.prototype 定义了一些常用的属性 ,body,head,分别指代 HTML 文档中的<body><head>标签。

```
> document.body
▶<body>…</body>
> document.head
▶<head>…</head>
```

5.Document.prototype 上定义了 documentElement 属性 ,指代文档的根元素 ,在 HTML 文档中，他总是指代<html>元素

```
> document.documentElement
<html lang="en">
▶<head>…</head>
▶<body>…</body>
</html>
```

6.getElementsByClassName、querySelectorAll、querySelector 在 Document,Element 类中均有定义 `div.getElementsByClassName('')`

作业 1.遍历元素节点树，要求不能用 children 属性（在原型链上编程）

答案：低级方法，给一个父节点，把子节点全部遍历出来，并打印

高级方法：打印树形结构 → 分层打印出来

先看 div 再看子元素节点，再挨个判断

```
<div>
    <p></p>
    <span>
        <strong></strong>
        <b></b>
    </span>
</div>
```

2.封装函数，返回元素 e 的第 n 层祖先元素

答案

```
<body>
    <div>
        <strong>
            <span>
                <i></i>
            </span>
        </strong>
    </div>
    <script type="text/javascript">
function retParent(elem, n) {

    while(elem && n) {

        elem = elem.parentElement;
        n --;
    }
    return elem;
}

var i = document.getElementsByTagName('i')[0];
```

```
> retParent(i, 1)
<· ▶<span>…</span>
```

3.封装函数，返回元素 e 的第 n 个兄弟节点，n 为正，返回后面的兄弟节点，n 为负，返回前面的，n 为 0，返回自己。

答案

```
<div>
    <span></span>
    <p></p>
    <strong></strong>
    <!-- this is comment-->
    <i></i>
    <address></address>
</div>
<script type="text/javascript">
function retSibling(e, n) {
    while(e && n) {
        if(n > 0) {
            e = e.nextElementSibling;
            n --;
        }else{
            e = e.previousElementSibling;
            n ++;
        }
    }
    return e;
}

var strong = document.getElementsByTagName('strong')[0];
```

```
> retSibling(strong, 1)
<·   <i></i>
> retSibling(strong, 2)
<·   <address></address>
> retSibling(strong, 3)
<· null
> retSibling(strong, 1000)
<· null
> retSibling(strong, -1)
<·   <p></p>
> retSibling(strong, -2)
<·   <span></span>
> retSibling(strong, -3)
<· null
> retSibling(strong, -300)
<· null
> retSibling(strong, 0)
<·   <strong></strong>
```

上面在 ie9 以下不能用。考虑到兼容性，可以按照下面**思路**写

```
function retSibling(e, n) {
    while(e && n) {
        if(n > 0) {
            if(e.nextElementSibling) {
                e = e.nextElementSibling;
            }else{
                e = e.nextSibling;
                if(e.nodeType != 1) {
                    e = e.nextSibling;
                    if(e.nodeType != 1) {
                        e = e.nextSibling;
                    }
                }
            }

            n --;
        }else{
            e = e.previousElementSibling;
            n ++;
        }
    }
    return e;
```

1 是元素节点的返回值，用 for 循环和三目运算：

```
function retSibling(e, n) {
    while(e && n) {
        if(n > 0) {
            if(e.nextElementSibling) {
                e = e.nextElementSibling;
            }else{
                for(e = e.nextSibling; e && e.nodeType != 1
                    ; e = e.nextSibling);
            }
            n --;
        }else{
            if(e.previousElementSibling) {
                e = e.previousElementSibling;
            }else{
                for(e = e.previousSibling; e && e.nodeType
                    != 1; e = e.previousSibling);
            }
            n ++;
        }
    }
    return e;
}
```

如果 for ( ) {}循环的执行体是空的，那么可以不写{}

4.编辑函数，封装 children 功能，解决以前部分浏览器的兼容性问题

答案

```
<div>
    <b></b>
    abc
    <!-- this is comment -->
    <strong>
        <span>
            <i></i>
        </span>
    </strong>
</div>
```

```
> div.myChildren()
< ▶ [b, strong]
```

```
Element.prototype.myChildren = function () {

    var child = this.childNodes;
    var len = child.length;
    var arr = [];
    for(var i = 0; i < len; i ++) {
        if(child[i].nodeType == 1) {
            arr.push(child[i]);
        }
    }
    return arr;
}
```

```
var div = document.getElementsByTagName('div')[0];
```

5.自己封装 hasChildren()方法，不可用 children 属性

答案

```
// var i = document.getElementsByTagName('i')[0];

// Element.prototype.myChildren = function () {

//     var child = this.childNodes;
//     var len = child.length;
//     var arr = [];
//     for(var i = 0; i < len; i ++) {
//         if(child[i].nodeType == 1) {
                return true;
//         }
//     }
//     return false;
// }
// var div = document.getElementsByTagName('div')[0];
```

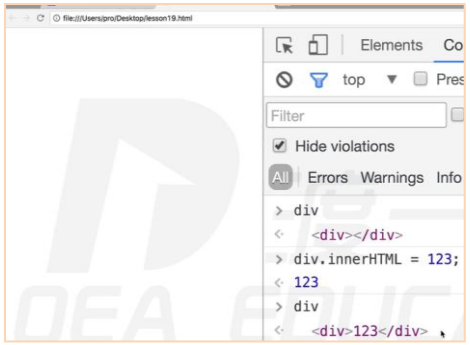
DOM 基本操作

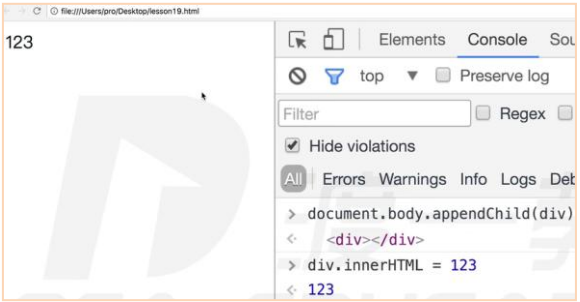
1、增

document.createElement(); //增加或创建元素节点（标签）——常见

例在括号里面写什么字符串，就创建什么标签

```
var div = document.createElement('div');
```





左上加了 123，但是页面里面没有显示，用右上方法即可

document.body.appendChild(div)

document.createTextNode(); //创建文本节点

document.createComment(); //创建注释节点

document.createDocumentFragment(); //创建文档碎片节点，最后讲

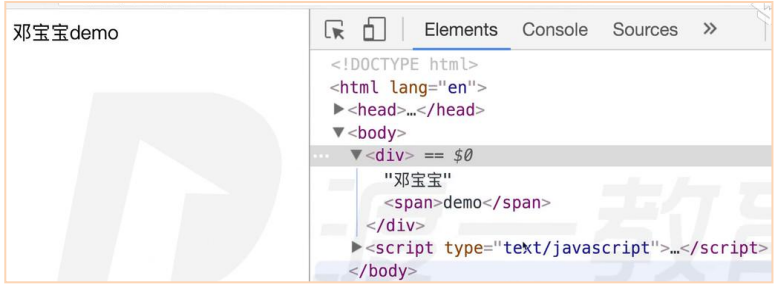
```
var div = document.createElement('');创建元素节点
var text = document.createTextNode('邓宝宝');创建文本节点
var comment = document.createComment('this is comment');
```

2、插——剪切操作

PARENTNODE.appendChild(); 可以理解成.push

```
<div></div>
```

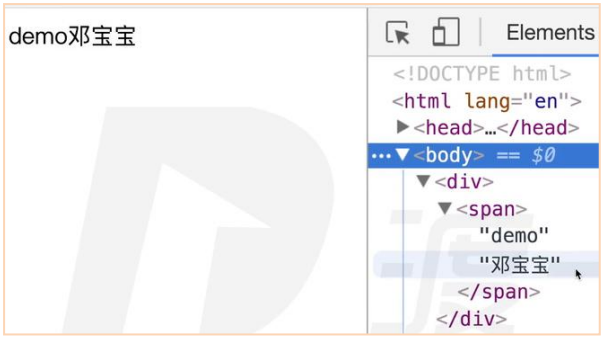
```
var div = document.getElementsByTagName('div')[0];
var text = document.createTextNode('邓宝宝');
var span = document.createElement('span');
div.appendChild(text);
div.appendChild(span);
var text1 = document.createTextNode('demo');
span.appendChild(text1);
```



例选择我们把邓宝宝插入到 span 里面去

```
<div></div>
```

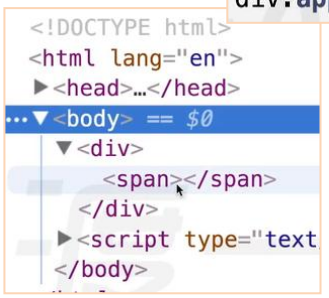
```
var div = document.getElementsByTagName('div')[0];
var text = document.createTextNode('邓宝宝');
var span = document.createElement('span');
div.appendChild(text);
div.appendChild(span);
var text1 = document.createTextNode('demo');
span.appendChild(text1);
span.appendChild(text);
```



例把 span 放 div 里面

```
<div></div>
<span></span>
```

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
div.appendChild(span);
```



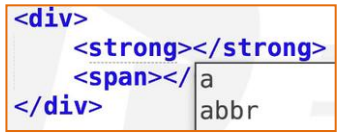
PARENTNODE.insertBefore(a, b);一定是 div 先 insert a，before b

例

```
<div>
    <span></span>
</div>
```

```
var strong = document.createElement('strong');

div.insertBefore(strong, span);
```

strong 插入效果见右侧

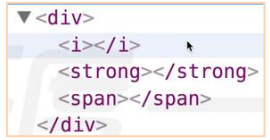


例在 div 后，span 前插入 strong

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
var strong = document.createElement('strong');

div.insertBefore(strong, span);
```

例在 div 后，span 前插入 strong 的基础上，把 i 放 strong 前面

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
var strong = document.createElement('strong');
var i = document.createElement('i');
div.insertBefore(strong, span);
div.insertBefore(i, strong);
```



3、删

parent.removeChild();　　//就是被剪切出来了

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
var strong = document.createElement('strong');
var i = document.createElement('i');
div.insertBefore(strong, span);
div.insertBefore(i, strong);
```

```
> div.removeChild(i);
<- <i></i>
```

```
> var ii = div.removeChild(i);
<- undefined
> ii
<- <i></i>
```

child.remove();　　//自尽，完全删除

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
var strong = document.createElement('strong');
var i = document.createElement('i');
div.insertBefore(strong, span);
div.insertBefore(i, strong);
```

```
> i.remove()
<- undefined
> strong.remove()
<- undefined
> span.remove()
<- undefined
```

4、替换

parent.replaceChild(new, origin);　　//用新的 new 去置换旧的 origin

```
var div = document.getElementsByTagName('div')[0];
var span = document.getElementsByTagName('span')[0];
var strong = document.createElement('strong');
var i = document.createElement('i');
div.insertBefore(strong, span);
div.insertBefore(i, strong);
var p = document.createElement('p');

// parentNode.replaceChild(new, origin);
```

```
> div
<- ▶<div>…</div>
> div.replaceChild(p, strong)
<-   <strong></strong>
> div
<- ▼<div>
    <i></i>
    <p></p>
    <span></span>
  </div>
```

DOM 基本操作

1、Element 节点的一些属性

innerHTML ==> 可取，可写，可赋值

innerText ==> 可取，可赋值 (老版本火狐不兼容) / textContent(火狐使用这个，老版本 IE 不好使)

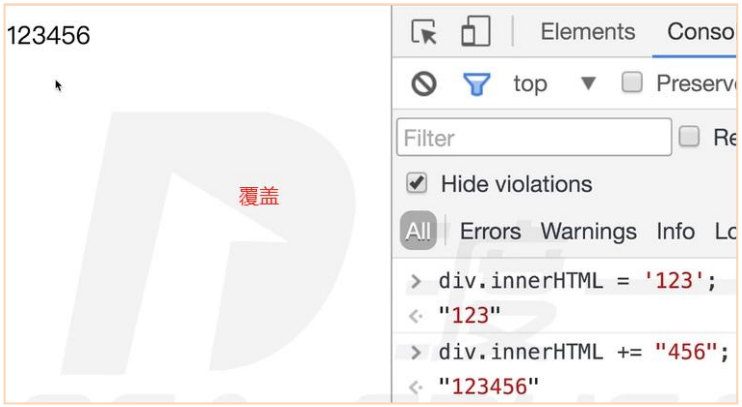
例 div.innerHTML 可以改变 div 里面的 HTML 的内容

```
<div>
    <span>123</span>
    <strong>234</strong>
</div>
```

```
> div.innerHTML    取
< "
        <span>123</span>
        <strong>234</strong>
    "
```

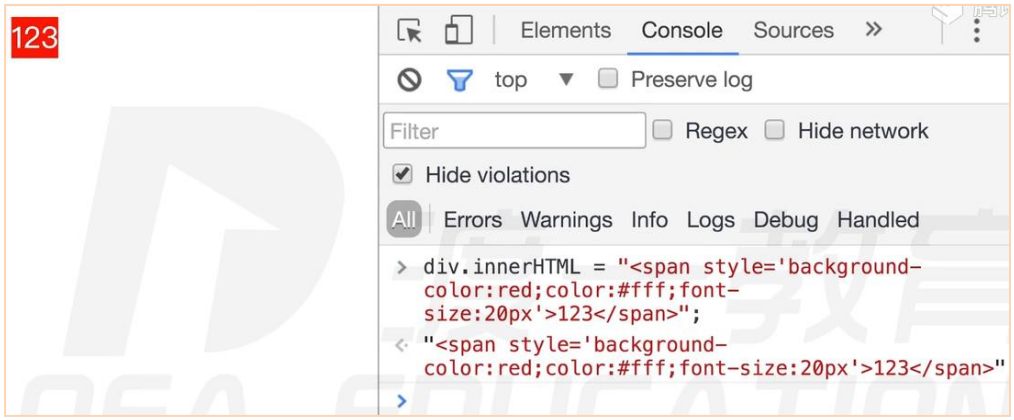
```
var div = document.getElementsByTagName('div')[0];
```

div.innerHTML = “123”;是覆盖



可赋值，可写值的才能+=

下面是用 innerHTML 改变 css 行间样式



例 innerText 可取可赋值

赋值会覆盖掉（能覆盖标签）

```
> div.innerText
< "123 234"
```

```
> div.innerText = '123'
< "123"    赋值
```

例下面的 innerText 让 span 没有了，所以赋值要谨慎

```
<div>
    <span>234</span>
</div>
```

```
> div.innerText = 123;
< 123
```

```
<body>
    <div>123</div> == $0
```

2、Element 节点的一些方法

ele.setAttribute() //设置

ele.getAttribute(); //取这个值

行间属性可以设置系统没有的

例

```
<div>
    <span>234</span>
</div>
```

用.setAttribute()设置属性

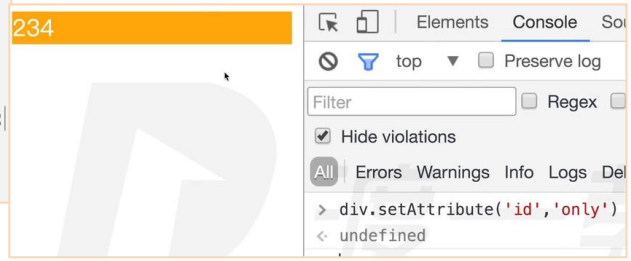
```
> div.setAttribute('class','demo');
< undefined    设置行间样式
> div
< ▶<div class="demo">…</div>
> div.setAttribute('id','only')
< undefined
> div
< ▶<div class="demo" id="only">…</div>
```

用.getAttribute()取这个值

```
> div.setAttribute('id','only')
< undefined
> div.getAttribute('id')
< "only"
```

例通过 div.setAttribute(‘id’ ,’ only’)动态的去改 div 的值

```
<style type="text/css">
    #only{
        font-size:20px;
        color:#fff;
        background-color: orange;
    }
</style>
<div>
    <span>234</span>
</div>
```



当满足一定条件时，if 就可以动态操作了，配合事件连成整体操作

这个行间属性可以设置系统没有的。data-log 是打点，点击率，这是人工设置的行间属性，不是系统定义的

例如何取这个 data-log？

```
<div>
    <a href="#" data-log="0">hehe</a>
</div>
```

```
var div = document.getElementsByTagName('div')[0];
var a = document.getElementsByTagName('a')[0];
a.onclick = function () {
    console.log(this.getAttribute('data-log'));
}
```

例给三个标签，让他们行间有一个属性 this-name，比如第一个 DIV，第二个 SPAN

```
<div></div>
<span></span>
<strong></strong>

<script type="text/javascript">
    var all = document.getElementsByTagName('*');
    for(var i = 0; i < all.length; i++) {
        all[i].setAttribute('this-name',all[i].nodeName);
    }
</script>
```

作业：

请编写一段JavaScript脚本生成下面这段DOM结构。要求：使用标准的DOM方法或属性。

```
<div class="example">
    <p class="slogan">姬成，你最帅!</p>
</div>
```

- 提示 dom.className 可以读写class

答案

```
<script type="text/javascript">
    var div = document.createElement('div');
    var p = document.createElement('p');
    div.setAttribute('class','example');
    p.setAttribute('class','slogan');
    var text = document.createTextNode('最帅');
    p.appendChild(text);
    div.appendChild(p);
    document.body.appendChild(div);
```

low 方法 div.innerHTML,剩下全手写

```
div.innerHTML = "
<>"
```

或

如果想改变一个 div 结构或 dom 结构或 html 结构的 class，直接用 div.className=""就可以了，不用 setAttribute

作业

1.封装函数 insertAfter()；功能类似 insertBefore();

insertAfter 是系统没有提供的

提示:可忽略老版本浏览器，直接在 Element.prototype 上编程

```
<div>
    <i></i>
    <b></b>
    <span></span>
</div>

<script type="text/javascript">
Element.prototype.insertAfter = function (targetNode,
    afterNode) {
    var beforeNode = afterNode.nextElementSibling;
    if(beforeNode == null) {
        this.appendChild(targetNode);
    }else{
        this.insertBefore(targetNode, beforeNode);
    }
}
var div = document.getElementsByTagName('div')[0];
var b = document.getElementsByTagName('b')[0];
var span = document.getElementsByTagName('span')[0];
var p = document.createElement('p');
```

2.将目标节点内部的节点顺序，逆序。（标签逆序）

eg: <a>

<a>

利用 appendChild 和剪切，第一次先操作倒数第二个，第二次操作倒数第三个，写个 for 循环

3.封装 remove(); 使得 child.remove()直接可以销毁自身

日期对象 Date()——就是一种对象，是系统提供好的

var date = new Date()大写的 Date 是系统提供的一个构造函数，通过 new Date 的方法会给我们返回一个对象，这个对象就是一个日期对象。日期对象有很多属性和方法。小的 date 代表此时此刻的时间。用小的 date 调用方法，如 date.getDate( )

Date 对象属性(不够标准)

属性	描述
constructor	返回对创建此对象的 Date 函数的引用。原型上的属性。
prototype	使您有能力向对象添加属性和方法。prototype 是构造函数的属性

Date 对象方法

方法	描述
Date()	返回当日的日期和时间。
getDate()	制造出对象,从 Date 对象返回一个月中的某一天 (1 ~ 31)。
getDay()	今天是一周的第几天,如果是 2 是星期二,但是是指第三天(第一天是周日,也就是 0)。从 Date 对象返回一周中的某一天 (0 ~ 6)。
getMonth()	**一月份返回值是 0**，从 Date 对象返回月份 (0 ~ 11)。
getFullYear()	从 Date 对象以四位数字返回年份。
getYear()	已废弃。请使用 getFullYear() 方法代替。
getHours()	返回 Date 对象的小时 (0 ~ 23)。
getMinutes()	返回 Date 对象的分钟 (0 ~ 59)。
getSeconds()	返回 Date 对象的秒数 (0 ~ 59)。
getMilliseconds()	返回 Date 对象的毫秒(0 ~ 999)。
getTime() **最有用**	返回 1970 年 1 月 1 日（纪元时刻）至今的毫秒数。经常用于项目的计算时间。获取时间戳
setDate()	设置 Date 对象中月的某一天 (1 ~ 31)。
setMonth()	设置 Date 对象中月份 (0 ~ 11)。
setFullYear()	设置 Date 对象中的年份（四位数字）。
setYear()	请使用 setFullYear() 方法代替。
setHours()	设置 Date 对象中的小时 (0 ~ 23)。
setMinutes()	设置 Date 对象中的分钟 (0 ~ 59)。
setSeconds()	设置 Date 对象中的秒钟 (0 ~ 59)。
setMilliseconds()	设置 Date 对象中的毫秒 (0 ~ 999)。
setTime()	以毫秒设置 Date 对象。**机械之间交换时间**
toSource()	返回该对象的源代码。
toString()	把 Date 对象转换为字符串。
toTimeString()	把 Date 对象的时间部分转换为字符串。
toDateString()	把 Date 对象的日期部分转换为字符串。

getUTC 一类，parse()的没用，不放表格里面

在控制台调用 date.getSeconds 就是 date 创建时间的毫秒数 ,是静止的 ,不是动态的。这个 date 对象记录的是出生的那一刻的时间，不是实时的。

例 **getTime()**

```
var firstTime = new Date().getTime();

for(var i = 0; i < 1000000; i++) {
}
        程序计算的毫秒数
var lastTime = new Date().getTime();
console.log(lastTime - firstTime);
```

例 **setDate()**

```
> date.setDate(15)
< 1487142638547
> date          跟着变化
< Wed Feb 15 2017 15:10:38 GMT+0800 (CST)
```

例**循环执行用 setInterval**

```
var date = new Date();
date.setMinutes(17);

setInterval(function () {

    if(new Date().getTime() - date.getTime() > 1000) {

        console.log('老邓还是个宝宝')

    }
},1000);
```

例 **setTime()**

```
> date.setTime(12345678900)
< 12345678900
> date
< Sun May 24 1970 05:21:18 GMT+0800 (CST)
```

例 **toString()和 toTimeString()**

```
> date.toString()
< "Wed Mar 22 2361 03:15:00 GMT+0800 (CST)"
> date.toTimeString()
< "03:15:00 GMT+0800 (CST)"
```

作业：封装函数，打印当前是何年何月何日何时，几分几秒。

js 定时器

一、setInterval(); **//注意：setInterval(“func()”,1000);定时循环器**

例 setInerval(function () {},1000);定时器，意思是 1000 毫秒执行一次这个函数

```
var time = 1000;
setInterval(function () {
    console.log('a');

}, time);


time = 2000;
```

time=2000不能改变函数里面的时间，只能执行写在time的时间数

例如果先定义 1000 毫秒，在后面改成 2000 毫秒，程序仍按 1000 毫秒执行，因为他只识别一次，不能通过改变 time 改变 setInterval 的快慢

```
var time = 1000;
setInterval(function () {
    console.log('a');

}, time);


time = 2000;
```

例查数

```
var i = 0;
setInterval(function () {
    i++;
    console.log(i);
}, 1000);
```

查数

例定时器到底准不准？见下方：

```
var firstTime = new Date().getTime();
setInterval(function () {

    var lastTime = new Date().getTime();
    console.log(lastTime - firstTime);
    firstTime = lastTime;

}, 1000);
```

setInerval计算时间并不准确

```
1001
999
1001
999
1000
```

setInterval 计算时间非常不准

注意：setInterval();是 window 的方法，在全局上就算不写 window. setInterval();他也会上全局的 GO 里面查找，所以不写 window.也行。

每一个 setInterval();都会返回一个一个数字，作为唯一的标识，有唯一标识就可以把他清除掉（利用 clearInerval 清除）

例用 timer2 接收返回值。timer 是逐一罗列放下排序

```
var timer = setInterval(function() {

}, 1000);

var timer2 = setInterval(function (){},2000);
```

```
> timer
< 1
> timer2
< 2
```

二、clearInterval(); **//能让 setInterval 停止**

例一般写了 setInterval 就要写 clearInterval

```
// clearInterval();
var i = 0;
var timer = setInterval(function() {
    console.log(i++);
    if(i > 10) {
        clearInterval(timer);
    }
}, 10);
```

```
2
3
4
5
6
7
8
9
10
```

三、**setTimeout();** **//正真的定时器,隔了一段时间后再执行（起推迟作用），并且只执行一次**

例隔了 1000 毫秒才执行，并且只执行一次

```
setTimeout(function() {

    console.log('a');

}, 1000);
```

```
a
```

常应用于一个电影试看期是 5 分钟

四、clearTimeout(); //清除 setTimeout();让他停止执行

例这种写法，setTimeout();还没执行就被清除了，就执行不了

```
var timer = setTimeout(function() {
    console.log('a');
}, 1000);

clearTimeout(timer);
```

例这个 timer= setTimeout();返回的唯一标识和 setInterval 返回的唯一标识是不会重叠的，他们两个是依次的

```
var timer = setTimeout(function() {
    console.log('a');
}, 1000);

var timer2 = setInterval(function () {},10000);

clearTimeout(timer);
```

```
> timer
<- 1
> timer2
<- 2
```

setInterval();setTimeout();clearInterval();clearTimeout();这四个都是全局对象，都是 window 上的方法，内部函数 this 指向 window

例 setInterval("func()",1000);和 setTimeout();都有另一种形式展现，里面可以写成字符串，例如" console.log(a);"。但是一般用 function( ){}

```
setInterval("console.log('a');", 1000);
```

特殊形式，但是一般不这样写，意思是1000毫秒执行一次console.log

作业

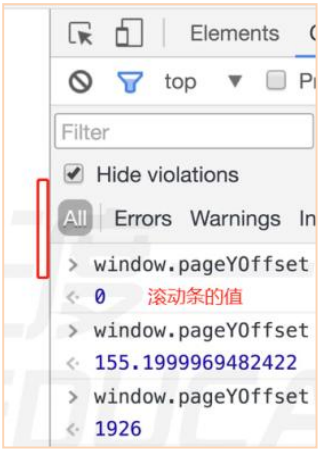
1.计时器，到三分钟停止


```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
        input{
            border:1px solid rgba(0,0,0,0.8);
            text-align:right;
            font-size:20px;
            font-weight:bold;
        }

    </style>
</head>
```

```
bold
bolder
bold (value)
bolder (value)
```

定时器要计数，计完数以后要填到结构里面，下面的起的两个变量 minutes 和 seconds 用于计数，计完数以后要填到 dom 结构里面（元素标签最正确的说法是 dom 结构，因为他可以被 dom 操作）

```
<script type="text/javascript">

    var minutesNode = document.getElementsByTagName('input')[0]
        ;
    var secondsNode = document.getElementsByTagName('input')[1]
        ;

    var minutes = 0,
        seconds = 0;
    var timer = setInterval(function () {

        seconds ++;
        if(seconds == 60) {
            seconds = 0;
            minutes ++;
        }

        secondsNode.value = seconds;
        minutesNode.value = minutes;
        if(minutes == 3) {
            clearInterval(timer);
        }

    }, 10);
```



DOM/BOM 基本操作

这一部分都是实战用的，考试面试一般不考

脚本化就是操作的意思

一、查看滚动条的滚动距离

1、window.pageXOffset 横向/pageYOffset 纵向滚动条

IE8 及 IE8 以下不兼容（IE9 部分兼容）IE9 以上能用

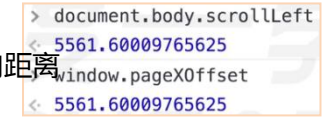
例：滚动条往下滚动了 400px，求浏览器最顶端到滚动条滚动的位置的像素

答案 400px+首屏像素（此时这个屏幕的底端距离整个网页的最顶端也是这样算）

2、IE8 及 IE8 以下的使用方法

1）document.body.scrollLeft/scrollTop——求横向距离和纵向距离

2）document. documentElement.scrollLeft/scrollTop



上面两个兼容性比较混乱，其中一个有值，另外一个的值一定是 0。这两个最好的用法是取两个值相加，因为不可能存在两个同时有值

如 document.body.scrollLeft + document. documentElement.scrollLeft

3、封装兼容性方法（哪个浏览器都好用），求滚动轮滚动距离 getScrollOffset()

——求滚动条的位置

例 Offset 是尺寸的意思（可以封装一个代码库，放在 js 文件里面）

```
function getScrollOffset() {

    if(window.pageXOffset) {
        return {
            x : window.pageXOffset,
            y : window.pageYOffset
        }
    }else{
        return {
            x : document.body.scrollLeft + document.doc
                umentElement.scrollLeft,
            y : document.body.scrollTop + document.doc
                umentElement.scrollTop
        }
    }

}
```

二、查看视口的尺寸

可视区窗口就是编写的 html 文档可以看到的部分，不含菜单栏、地址栏、控制台

1、window.innerWidth/innerHeight 可视区域的宽高 （加上 滚动条宽度 / 高度）

IE8 及 IE8 以下不兼容 //w3c 标准方法

例求可视区窗口的尺寸。

```
<script type="text/javascript">
    function getViewportOffset() {
        if(window.innerWidth) {
            return {
                w : window.innerWidth,
                h : window.innerHeight
            }
        }else{
            if(document.compatMode === "BackCompat") {
                return {
                    w : document.body.clientWidth,
                    h : document.body.clientHeight
                }
            }else{
                return {
                    w : document.documentElement.clientWidth,
                    h : document.documentElement.clientHeight
                }
            }
        }
```

如果窗口放大页面了，页面的尺寸也会拉伸了，尺寸就会变小。

注意渲染模式：

1 标准模式：<！DOCTYPE html>是 html5 的（在 emmet 插件下 html:5 就出来了）

2 怪异/混杂模式：试图去兼容之前的语法，去掉<！DOCTYPE html>这一行即可开启（向后兼容）

2、document.documentElement.clientWidth/clientHeight

标准模式下，任意浏览器都兼容 client 是客户端的意思

3、document.body.clientWidth/clientHeight

适用于怪异模式下的浏览器

4、封装兼容性方法，返回浏览器视口尺寸 getViewportOffset()

例 document.compatMode 是用于判断是怪异模式还是标准模式的

```
> document.compatMode
< "CSS1Compat"   标准模式
```

```
> document.compatMode
< "BackCompat"   怪异模式向后兼容
```

例封装查看可视窗口的兼容性方法

```
<br>
<script type="text/javascript">
    function getViewportOffset() {
        if(window.innerWidth) {
            return {
                w : window.innerWidth,
                h : window.innerHeight
            }
        }else{
            if(document.compatMode === "BackCompat") {
                return {
                    w : document.body.clientWidth,
                    h : document.body.clientHeight
                }
            }else{
                return {
                    w : document.documentElement.clientWidth,
                    h : document.documentElement.clientHeight
                }
            }
        }
```

```
> getViewportOffset()
< ▶ Object {w: 500, h: 825}
```

```
> getViewportOffset()
< ▶ Object {w: 500, h: 825}
```

三、查看元素的几何尺寸

1、domEle.getBoundingClientRect();　　//这是 es5.0 的方法，但是只用了解

例

```
<body>
    <div style="width:100px;height:100px;background-
    color:red;position:absolute;left:100px;top:100px;"></div>
    <script type="text/javascript">
        // function getViewportOffset() {
        // }
        var div = document.getElementsByTagName('div')[0];
```

```
> div.getBoundingClientRect();
< ▼ClientRect
    bottom: 200
    height: 100
    left: 100
    right: 200
    top: 100
    width: 100
  ▶ __proto__: ClientRect
```

求得是四个边和窗口之间的像素距离，right 是右边离 document 的边有 200px　，也可以理解成求的是左上点和右下点的位置

2、兼容性很好

3、该方法返回一个对象，对象里面有 left,top,right,bottom 等属性。left 和 top 代表该元素左上角的 X 和 Y 坐标，right 和 bottom 代表元素右下角的 X 和 Y 坐标

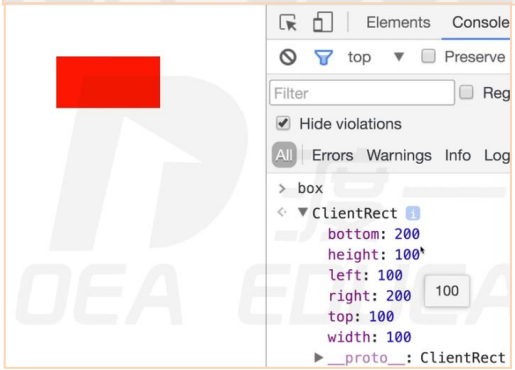
4、height 和 width 属性老版本 IE 并未实现

在老版本的 ie 里面，利用右侧边减左侧面解决

5、返回的结果并不是“实时的”

例

```
<body>
    <div style="width:100px;height:100px;background-
    color:red;position:absolute;left:100px;top:100px;"></div>
    <script type="text/javascript">
        // function getViewportOffset() {
        // }
        var div = document.getElementsByTagName('div')[0];

        var box = div.getBoundingClientRect();

        div.style.width = "200px";
```



四、查看元素的尺寸

dom.offsetWidth，dom.offsetHeight

求得值是包括 padding 的

例我们求的是实际内容区的宽高，还是长得看起来的宽高（视觉尺寸）？

```
<body>
    <div style="padding:100px;width:100px;height:100px;background-
    color:red;position:absolute;left:100px;top:100px;"></div>
    <script type="text/javascript">
        // function getViewportOffset() {
        // }
        var div = document.getElementsByTagName('div')[0];
```

```
> div.offsetWidth
< 300
> div.getBoundingClientRect()
< ▶ ClientRect {top: 100, right: 400, bottom: 400,
  left: 100, width: 300…}
```

```
> div.style.height
< "100px"
```

dom.offsetWidth 和 domEle.getBoundingClientRect();出现的值是一样的，求的值都是 padding+content（视觉尺寸），可以被代替。右上是间接的求这个 div 的宽高

五、查看元素的位置

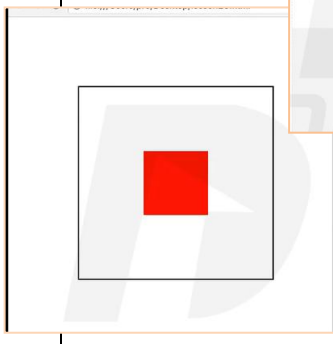
1、dom.offsetLeft, dom.offsetTop

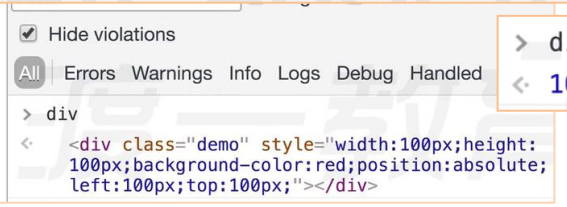
对于无定位父级的元素，返回相对文档的坐标。

对于有定位父级的元素，返回相对于最近的有定位的父级的坐标。(无论是 left 还是 margin-left 等都是距离。)

例.offsetLeft 忽略自身是不是定位元素，求出来的是这个元素距离自己有定位的父级元素的距离，不管距离是 margin 生成还是定位生成或者是其他方法（圆形，五角星都是这样）

```
<body>
    <div style="width:300px;height:300px;border:2px solid
    black;position:relative;top:100px;left:100px;">
        <div class="demo" style="
        width:100px;height:100px;background-
        color:red;position:absolute;left:100px;top:100px;"></div>

    </div>
    <script type="text/javascript">
        // function getViewportOffset() {
        // }
        var div = document.getElementsByClassName('demo')[0];
```






```
> div.offsetLeft
< 100 相对于父级的位置
```

例 position:static;是 position 是默认值，是不定位的写法，是静态的意思
这种情况下 margin 塌陷了（因为加了 border 又解决了塌陷的情况）

```
<body>
    <div style="width:300px;height:300px;border:2px solid
    black;margin-left:100px;margin-top:100px">
        <div class="demo" style="
        width:100px;height:100px;background-
        color:red;position:absolute;margin-left:100px;margin-
        top:100px"></div>

    </div>
    <script type="text/javascript">
        // function getViewportOffset() {
        // }
        var div = document.getElementsByClassName('demo')[0];
```

```
> div.offsetLeft
< 210
```

```
> div.offsetTop
< 202
```

body 有个默认的 margin：8px;存在，横向的，是相加
body 有个默认的 margin：8px;存在，纵向的 body 的 margin 和外面的 div 的 margin 重叠了，实现了 margin 塌陷

2、dom.offsetParent

返回最近的有定位的父级，如无，返回 body, body.offsetParent 返回 null
这个方法能求有定位的父级
例沿用上面的例子，在控制台上面操作 →

```
> div.offsetParent
< ▶<body>…</body>
> div.offsetParent.offsetParent
< null
```

例加了个 relative 定位

```
<div style="width:300px;height:300px;border:2px solid
black;margin-left:100px;margin-top:100px;position:relative;">
    <div class="demo" style="
    width:100px;height:100px;background-
    color:red;position:absolute;margin-left:100px;margin-
    top:100px"></div>

</div>
<script type="text/javascript">
    // function getViewportOffset() {
    // }
    var div = document.getElementsByClassName('demo')[0];
```

```
> div.offsetParent
< ▶<div style="width:300px;height:300px;border:2px
  solid black;margin-left:100px;margin-top:100px;
  position:relative;">…</div>
```

作业：求元素相对于文档的坐标 ，以 getElementPosition()命名
思路：先看有没有 有定位的父级，如果有，先求他与有定位的父级的距离，然后把视角换到他有定位的父级上，这他这个有定位的父级上还有没有有定位的父级，一段一段的加，一直加到最后，是一个循环递归的过程

六、让滚动条滚动

1、window 上有三个方法 scroll(), scrollTo(),两个功能一样，scrollBy();累加滚动距离
scroll(x，y), scrollTo(x，y),功能是一样的，里面能填两个参数
scroll(x 轴滚动条的距离，y 轴滚动条的距离)，里面的 xy 可以填负数
scrollBy(x，y);是累加滚动距离，填负数就往上滚动
2、三个方法功能类似，用法都是将 x,y 坐标传入。即实现让滚动轮滚动到当前位置。
3、区别：scrollBy()会在之前的数据基础之上做累加。
eg：利用 scrollBy() 快速阅读的功能
例利用上一个例子的代码，在控制台操作 scroll 如右：

```
> window.scroll(0, 100)
< undefined
> window.scroll(0, 500)
< undefined   不动，不是累加
> window.scroll(0, 500)
< undefined
> window.scroll(0, 500)
< undefined
> window.scroll(0, 500)
```

scroll 的滚到当前距离（某个固定的鼎），
不是累加使用，除非移动了滚动条的位置，
才会从新计算距离

例利用上一个例子的代码，在控制台操作 scrollBy();如右：

```
> window.scrollBy(0,10)
< undefined
> window.scrollBy(0,10)
< undefined
> window.scrollBy(0,10)
< undefined
```

scrollBy();是累加滚动距离，填负数就往上滚动
scrollBy(0,10);是向下滚动 10 个距离
scrollBy(0,-10);是向上滚动 10 个距离

作业：点完收起，页面回到点展开时候的位置

思路：求滚动条的位置，记录点击展开时候的位置，点收起回到位置，记录是查看 getScrollOffset()封装方法，回去是 window.scroll，在执行
例做个自动翻页的阅读（当点击 start 就自动阅读）
答案

```
<div style="width:100px;height:100px;background-
color:orange;color:#fff;font-size:40px;font-weight:bold;text-
align: center;line-
height:100px;position:fixed;bottom:200px;right:50px;border-
radius:50%;opacity:0.5;">start</div>
<div style="width:100px;height:100px;background-
color:#0f0;color:#fff;font-size:40px;font-weight:bold;text-
align: center;line-
height:100px;position:fixed;bottom:50px;right:50px;border-
radius:50%;opacity:0.5;">stop</div>
```

```
var start = document.getElementsByTagName('div')[0];
var stop = document.getElementsByTagName('div')[1];
var timer = 0;
start.onclick = function () {

    timer = setInterval(function () {

        window.scrollBy(0, 10);

    }, 100);

}
stop.onclick = function () {
    clearInterval(timer);
}
```

上面有个问题：多次点击 start 就会加速，并且停不下来了，按 stop 就只能停止一个加速，并不能使他停止

右侧是让多次点击 star 不能加速

所以加个锁 key

stop 以后再把锁打开

```
var start = document.getElementsByTagName('div')[0];
var stop = document.getElementsByTagName('div')[1];
var timer = 0;
var key = true;
start.onclick = function () {
    if(key) {

        timer = setInterval(function () {

            window.scrollBy(0, 10);

        }, 100);
        key = false;
    }
}
stop.onclick = function () {
    clearInterval(timer);
    key = true;
}
```

对比上一个增加了一个调速的按钮

增加一个 num 变量，通过调整变量来调速

注意不能通过控制 timer 来调整速度

也不能通过 100 来调速

```
var start = document.getElementsByTagName('div')[0];
var stop = document.getElementsByTagName('div')[1];
var timer = 0;
var key = true;
start.onclick = function () {
    if(key) {

        timer = setInterval(function () {

            window.scrollBy(0, num);   设置一个变量后，通过调整变量来调整速度

        }, 100);   调速不能通过这里调整
        key = false;
    }
}
stop.onclick = function () {
    clearInterval(timer);
    key = true;
}
```

脚本化 CSS

dom 不能操作 css，是间接操作 css，这一部分需要记下

一、读写元素 css 属性（间接控制）

1、dom.style.prop //常用，只有这个可读可写，其余只能读

1）可读写行间样式，没有兼容性问题，碰到 float 这样的关键字属性，前面应加 css(行间样式以外没用)

eg:float —> cssFloat

2）符合属性必须拆解（建议），组合单词变成小驼峰式写法

3）写入的值必须是字符串格式

例 dom.style 属性。能拿，能写（通过写间接改变了 css 属性）

```
<div style="width:100px;height:100px;background-color:red;"></div>

<script type="text/javascript">


    var div = document.getElementsByTagName('div')[0];
```

```
> div.style.width
< "100px"
> div.style.width = "200px"
< "200px"
> div.style.height = "300px";
< "300px"
> div.style.backgroundColor
< "red"
> div.style.backgroundColor = "green"
< "green"
```

```
> div.style
< CSSStyleDeclaration {0: "width", 1: "height", 2: "background-color", alignContent: "", alignItems: "", alignSelf: "", alignmentBaseline: "", all: ""…}
```

空的，没有值

CSSStyleDeclaration 是 css 样式表声明（类数组，有索引类的属性），把你能够用的所有的 css 都展示出来了，里面不填写就有没有值是空串。

可以用 div.style[‘width’]拿出属性，也可以用 div.style.width，效果一样。

在 js 访问属性的时候没有-杠的形式，不能写 background-color，要写小驼峰。

不在 html 文件里面写的值也可以利用 js 调用，如 div.style.borderRadius=”50”；

例 js 只能操作 css 行间样式

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
        div{
            width:200px;
        }


    </style>
</head>
<body>
    <div style="height:100px;background-color:red;"></div>

    <script type="text/javascript">


        var div = document.getElementsByTagName('div')[0];
```

```
> div.style.width
< ""
```

```
> div.style.float
< "left"
> div.style.cssFloat
< "left"
```

推荐使用

```
> div.style.border="2px solid black";
< "2px solid black"
> div.style.borderWidth = "5px";
< "5px"
> div.style.borderStyle = "double";
< "double"
```

推荐这样写

二、查询计算样式

1）**window.getComputedStyle(ele,null);** //原生底层的方法。展示权重最高的

获取伪元素的方法：window.getComputedStyle(ele,null);括号里面要填两个东西，第一个 ele 是填的是你要获取谁，第二个先填写 null（null 解决的就是伪元素的问题，用它可以获取伪元素的样式表）。

2）**计算样式只读，不可以写入**

3）**返回的计算样式的值都是绝对值，没有相对单位**

4）**IE8 及 IE8 以下不兼容**

例在控制台操作就能有一个 css 样式表（类数组）

```
    <style type="text/css">
        div{
            width:200px;
        }


    </style>
</head>
<body>
    <div style="height:100px;background-color:red;"></div>

    <script type="text/javascript">



        var div = document.getElementsByTagName('div')[0];
```

```
> window.getComputedStyle(div,null)
< CSSStyleDeclaration {0: "animation-
  "animation-direction", 2: "animatio
261: "rx"
262: "ry"
alignContent: "stretch"
alignItems: "stretch"
alignSelf: "stretch"
alignmentBaseline: "auto"
```

不管填不填都有值，这里面的值都是默认值，这会获取的是这个当前元素所展现出的一切css属性的显示值

window.getComputedStyle 不管填不填都有值，这里面的值都是默认值，这会**获取的是这个当前元素所展现出的一切 css 属性的显示值（显示值是你最终看到的值）**

例以上面的 HTML 页面为基础，在控制台操作

```
> window.getComputedStyle(div,null).width
< "200px"
```

例根据上面例子，增加行间样式，width：100px，在控制台操作

```
> window.getComputedStyle(div,null).width
< "100px"
```

例 div.style.width 获取的是行间样式。取一个元素显示样式 getComputedStyle 更准

```
    <style type="text/css">
        div{
            width:200px!important;
        }


    </style>
</head>
<body>
    <div style="width:100px;float:left;height:100px;background-
    color:red;"></div>

    <script type="text/javascript">



        var div = document.getElementsByTagName('div')[0];
```

```
> window.getComputedStyle(div,null).width =
"300px";
不可以写入
▶ Uncaught DOMException: Failed to set    VM1205:
the 'width' property on 'CSSStyleDeclaration':
These styles are computed, and therefore the
'width' property is read-only.
    at <anonymous>:1:41
```

```
> window.getComputedStyle(div,null).width
< "200px"
> div.style.width
< "100px"
```

例返回的计算样式的值都是绝对值，没有相对单位，相对值会转换成绝对值显示

```
    <style type="text/css">
        div{
            width:10em;
        }


    </style>
</head>
<body>
    <div style="float:left;height:100px;background-color:red;"></div>

    <script type="text/javascript">



        var div = document.getElementsByTagName('div')[0];
```

```
> window.getComputedStyle(div,null).width
< "160px"
> window.getComputedStyle(div,null).backgroundColor
< "rgb(255, 0, 0)"
```

三、查询样式

1）**ele.currentStyle** //展示权重最高的

2）**计算样式只读，不可以写入**

3）**返回的计算样式的值不是经过转换的绝对值，是原封不动的**

4）**IE 独有的属性**

例['width']写成.width 也可以。两种写法都可以，推荐写.点

```
// var div = document.getElementsByTagName('div')[0];

// window.getComputedStyle(div, null)[prop]
// div.currentStyle --> CSSStyleDeclaration;
div.currrentStyle['width']
```

`div.currrentStyle['width']` 等于 `div.currrentStyle.width`

作业：封装兼容性方法 getStyle(obj,prop);是常用的 ，通用的

```
var div = document.getElementsByTagName('div')[0];

function getStyle(elem, prop) {

    if(window.getComputedStyle) {
        return window.getComputedStyle(elem, null)[prop];
    }else{
        return elem.currentStyle[prop];
    }

}
```

必须中括号

function getStyle (elem,prop){}

//elem 是指获取的谁（dom 元素），prop 是获取的是什么属性

例：理解 window.getComputedStyle(ele,null);第二个值是 null

获取伪元素的方法：window.getComputedStyle(ele,null);括号里面要填两个东西，第一个 ele 是填的是你要获取谁，第二个先填写 null（null 解决的就是伪元素的问题，用它可以获取伪元素的样式表）。

```
    <style type="text/css">
        div{
            width:10em;
        }
        div::after{
            content:"";
            width:50px;
            height:50px;
            background-color: green;
            display: inline-block;
        }

    </style>
</head>
<body>
    <div style="float:left;height:100px;background-color:red;"></div>

    <script type="text/javascript">
        var div = document.getElementsByTagName('div')[0];
        window.getComputedStyle(div, "after")
```

window.getComputedStyle(div, “after”);就获取了伪元素的样式表，在控制台操作

```
> window.getComputedStyle(div, "after").width
< "50px"
```

例怎么改变伪元素（下面是举例，通过改变 class）

如图这个上尖角号和下尖角号就是用伪元素写的，在操作的实际会发生变化

```
    div{
        width:10em;
    }
    .green::after{
        content:"";
        width:50px;
        height:50px;
        background-color: green;
        display: inline-block;
    }
    .yellow::after{
        content:"";
        width:50px;
        height:50px;
        background-color: yellow;
        display: inline-block;
    }
</style>
```

当我们点击这个红色的 div 时，绿色方块变黄

```
<body>
    <div class="green" style="float:left;height:100px;background-color:red;"></div>

    <script type="text/javascript">

        var div = document.getElementsByTagName('div')[0];
        div.onclick = function () {

            div.className = "yellow";

        }
```

虽然我们讲了通过.style 改变 css，但是我们一般不像改

例实现我们点击一次，这个 div 就变成宽高 200px

```
<style type="text/css">
    div{
        width:100px;
        height:100px;
        background-color: red;
    }
```

```
<body>
    <div></div>

    <script type="text/javascript">

        var div = document.getElementsByTagName('div')[0];
        div.onclick = function () {

            div.style.width = "200px";
            div.style.height = "200px";
            div.style.backgroundColor = "green";

        }

    </script>
```

效果是点击后，

红方块变大变绿

例凡是需要像上面这样状态切换的情况，我们可以提前把状态编辑好，再通过权重覆盖形式改掉状态（好维护，节省效率）

通过改变 class 来实现改变，改状态位的操作

class 是保留字，要避开

所以起名 className

```
<style type="text/css">
    div{
        width:100px;
        height:100px;
        background-color: red;
    }

    .active {
        width:200px;
        height:200px;
        background-color: green;
    }
```

```
<body>
    <div></div>

    <script type="text/javascript">

        var div = document.getElementsByTagName('div')[0];
        div.onclick = function () {

            // div.style.width = "200px";
            // div.style.height = "200px";
            // div.style.backgroundColor = "green";
            div.className = "active";
        }
```

小练习：让方块运动。默认是 left 是默认值是 auto

```
<div style="width:100px;height:100px;background-
color:red;position:absolute;left:0;top:0;"></div>

<script type="text/javascript">
    function getStyle(elem, prop) {
        if(window.getComputedStyle) {
            return window.getComputedStyle(elem, null)[prop];
        }else{
            return elem.currentStyle[prop];
        }
    }

    var div = document.getElementsByTagName('div')[0];
    setInterval(function () {
        div.style.left = parseInt(getStyle(div, 'left')) + 10 + 'px
            ';
    }, 100);
```

```
> window.getComputedStyle(div, null).left
"0px"
```

上面这个例子中，setInterval 详细解释

```
var div = document.getElementsByTagName('div')[0];
setInterval(function () {
    div.style.left = parseInt(getStyle(div, 'left')) + 1 + 'px'
    ;
}, 10);
```

控制速度　数字　控制流畅度

下面是设置到一个时候就停止了

```
function getStyle(elem, prop) {
    if(window.getComputedStyle) {
        return window.getComputedStyle(elem, null)[prop];
    }else{
        return elem.currentStyle[prop];
    }
}

var div = document.getElementsByTagName('div')[0];
var speed = 3;
var timer = setInterval(function () {
    speed += speed/7;
    div.style.left = parseInt(getStyle(div, 'left')) + speed +
        'px';

    if(parseInt(div.style.left) > 500) {
        clearInterval(timer);
    }
}, 10);
```

作业：轮播图（仿照优酷电影主页）每隔几秒动一次，无缝运动
看起来是四张图，其实是五张

脚本化样式表

查找，操作样式表

1）document.styleSheets

2）该属性存储了一个 html 文档里面的所有 css 样式表的集合

事件（所有事件都是用的小写）

交互是你对页面动一下，页面给一个反馈

1.何为事件　— 就是一个动作，没效果也是事件

2.重要吗？ — 交互体验的核心功能

演示 demo — 拖拽，和点击

例

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];


    div.onclick = function () {
        console.log('a');
    }
```

如何绑定事件

1、ele.onxxx = function (event) {}

1）兼容性很好，但是一个元素只能绑定一个事件处理程序

例 div.onclick = function(){}

div.oncliccck 就叫做可以被点击的事件（绑定事件类型），function(){}是反馈，一旦事件被触发，就要执行 function 里面的函数（绑定的是一个事件处理函数）

例

```
div.onclick = function () {
    console.log('a');
}
```

b覆盖了a，这是赋值的原因

```
div.onclick = function () {
    console.log('b');
}
```

2）基本等同于写在 HTML 行间上，如下图

例

```
<div style="width:100px;height:100px;background-color:red;" onclick
="console.log('a')"></div>
<script type="text/javascript">
    // 句柄
    var div = document.getElementsByTagName('div')[0];


    // div.onclick = function () {
    //
    //     console.log('a');
    // }
```

两种方式效果一样

onclick=”consolo.log(‘a’)”是句柄的绑定方式，写在行间不用写 function(){}

2、**ele.addEventListener(type, fn, false);里面可以填三个参数**

IE9 以下不兼容，可以为一个事件绑定多个处理程序

例 div.addEventListener（‘ 事件类型’，处理函数，false ）

div.addEventListener（‘click’，function(){}，false ）

function(){} 是函数引用，和外面定义一个 function test(){}直接写 test 是一样的

例
```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
    // 句柄
    var div = document.getElementsByTagName('div')[0];

    div.addEventListener('click', function (){

        console.log('a');

    },false);
```

事件有一个事件监听机制

例一个事件绑定了两个处理函数。

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
    // 句柄
    var div = document.getElementsByTagName('div')[0];

    div.addEventListener('click', function (){

        console.log('a');

    }, false);

    div.addEventListener('click', function() {
        console.log('b');
    }, false);
```

若 console.log 两个都是(‘a ‘)打印出来是两个 a。这是两个处理函数，是两个地址

例下面只执行一个 a。下面和上面的结果是不一样的。这种写法是一个地址，一个人

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
    // 句柄
    var div = document.getElementsByTagName('div')[0];

    div.addEventListener('click', test, false);

    div.addEventListener('click', test, false);


    function test(){

        console.log('a');

    }
```

一个a

ele.addEventListener 不能给同一个函数绑定多次，重复的绑定一个函数就不能用了

3、**ele.attachEvent(‘on’ + type, fn);**

IE 独有，一个事件同样可以绑定多个处理程序，同一个函数绑定多次都可以

例 div.attachEvent(‘on’ + 事件类型，处理函数);

div.attachEvent(‘onclick’ ，function ( ) {});

例想给三个 li 都绑定这个事件

```
<body>
    <ul>
        <li>1</li>
        <li>2</li>
        <li>3</li>
    </ul>
    <script type="text/javascript">
        // 句柄

        var li = document.getElementsByTagName('li');
        for(var i = 0; i < li.length; i ++) {
            li[i].addEventListener('click', function() {

            }, false);
        }
```

例

```
<ul>
    <li>a</li>
    <li>a</li>
    <li>a</li>
    <li>a</li>
</ul>
```

使用原生 JS，addEventListener 给每个 li 元素绑定一个 click 事件，输出他们的顺序

答案绑定事件出现在循环里面（用到 i），考虑是否形成闭包，用立即执行函数来写

```
<ul>
    <li>a</li>
    <li>a</li>
    <li>a</li>
    <li>a</li>
</ul>
```

```
<script type="text/javascript">


    var liCol = document.getElementsByTagName('li'),
        len = liCol.length;
    for(var i = 0; i < len; i++) {
        (function(i){
            liCol[i].addEventListener('click', function () {

                console.log(i);

            }, false);

        }(i))
    }
```

事件处理程序的运行环境

1、ele.onxxx = function (event) {}

程序 this 指向是 dom 元素本身（指向自己）

例
```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">



    var div = document.getElementsByTagName('div')[0];
    div.onclick = function () {
        console.log(this);
    }
```

2、obj.addEventListener(type, fn, false);

程序 this 指向是 dom 元素本身（指向自己）

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
```

```
var div = document.getElementsByTagName('div')[0];
div.addEventListener('click', function () {

    console.log(this);

}, false);
```

3、obj.attachEvent('on' + type, fn);

程序 this 指向 window

例让 obj.attachEvent 指向自己，function handle(){}里面是事件处理程序

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">



    var div = document.getElementsByTagName('div')[0];
    div.attachEvent('onclick', function () {

        handle.call(div);

    });

    function handle() {
        this.
    }
```

4、封装兼容性的 addEvent(elem, type, handle);方法（必须会）

addEvent 是给一个 dom 对象添加一个**该事件类型**的处理函数

例
```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
```

```
    function addEvent(elem, type, handle) {
        if(elem.addEventListener) {
            elem.addEventListener(type, handle, false);
        }else if(elem.attachEvent) {
            elem.attachEvent('on' + type, function() {
                handle.call(elem);
            })
        }else{
            elem['on' + type] = handle;
        }
    }
```

解除事件处理程序

1、ele.onclick = false/''/null;　　==>　　解除 ele.onxxx = function (event) {}

2、ele.removeEventListener(type, fn, false);==>解除 addEventListener(type, fn, false)

3、ele.detachEvent('on' + type, fn);　　==> obj.attachEvent('on' + type, fn);

注:若绑定匿名函数，则无法解除

例用 div.onclick = null;解除事件，

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">



    // function addEvent(elem, type, handle) {
    // }
    var div = document.getElementsByTagName('div')[0];
    div.onclick = function () {
        console.log('a');
    }

    div.onclick = null;
```

例只能执行一次的事件的写法

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">



    // function addEvent(elem, type, handle) {
    // }
    var div = document.getElementsByTagName('div')[0];
    div.onclick = function () {
        console.log('a');
        this.onclick = null;
    }
```

例 ele.removeEventListener(type, fn, false);

```
var div = document.getElementsByTagName('div')[0];
div.addEventListener('click', test, false);
function test() {
    console.log('a');
}

div.removeEventListener('click', test, false);
```

一模一样

test 如果填 function(){}别人就找不到他，就没办法清除，所以此时的函数体要写在外面

例 ele.detachEvent('on' + type, fn);清除绑定时一摸一样的 obj.attachEvent('on' + type, fn);

事件处理模型 — 事件冒泡、捕获

事件处理的两个模型：事件冒泡、捕获（不能同时存在）

例

```
<style type="text/css">
    .wrapper{
        width:300px;
        height:300px;
        background-color: red;
    }

    .content{
        width:100px;
        height:100px;
        background-color: green;
    }

    .box{
        width:100px;
        height:100px;
        background-color: orange;
    }
</style>
<body>

    <div class="wrapper">
        <div class="content">
            <div class="box"></div>
        </div>
    </div>
    <script type="text/javascript">
```


```
<script type="text/javascript">

    var wrapper = document.getElementsByClassName('wrapper')[0];
    var content = document.getElementsByClassName('content')[0];
    var box = document.getElementsByClassName('box')[0];


    wrapper.addEventListener('click', function () {
        console.log('wrapper')
    }, false);
    content.addEventListener('click', function () {
        console.log('content')
    }, false);
    box.addEventListener('click', function () {
        console.log('box')
    }, false);
```

只点了黄色区域，但是出现了 box,content,wrapper，往下漏了，就是事件冒泡

1、事件冒泡：

结构上（非视觉上）嵌套关系的元素，会存在事件冒泡的功能，即同一事件，自子元素冒泡向父元素。（自底向上）

结构上存在父子关系的元素，如果点击到子元素，会一级级向父元素传递这个事件（从代码的角度是自底向上一层层冒泡的）

例加了 margin，只点黄色的，还是出现了 box,content,wrapper。所以与视觉无关

```
<style type="text/css">
    .wrapper{
        width:300px;
        height:300px;
        background-color: red;
    }

    .content{
        margin-left:300px;
        width:200px;
        height:200px;
        background-color: green;
    }

    .box{
        margin-left:200px;
        width:100px;
        height:100px;
        background-color: orange;
    }
</style>
```

```
<body>

    <div class="wrapper">
        <div class="content">
            <div class="box"></div>
        </div>
    </div>
    <script type="text/javascript">
```

2、事件捕获：（只有谷歌有，最新火狐有）

1）结构上（非视觉上）嵌套关系的元素，会存在事件捕获的功能，即同一事件，自父元素捕获至子元素（事件源元素）。（自底向上）

2）IE 没有捕获事件

一个对象的一个事件类型，只能存在一个事件处理模型（冒泡或捕获）

obj.addEventListener(type, fn, true);第三个参数为 true 就是事件捕获

例点击黄的：先红的捕获事件并且执行，再绿的捕获事件并且执行，最后只执行事件黄的。捕获是把结构的最外面先抓住。最外面先捕获，再一层层向里面捕获，最里面的是按常规执行。

例把 false 改成 true，就变成了事件捕获

html 部分沿用左侧冒泡情况的代码，只把 false 改成 true 就变成事件捕获了

```
wrapper.addEventListener('click', function () {
    console.log('wrapper')
}, true);
content.addEventListener('click', function () {
    console.log('content')
}, true);
box.addEventListener('click', function () {
    console.log('box')
}, true);
```

改为true就变成了捕获

红的wrapper
绿的content
黄色box

思考：同一个对象的同一个事件类型，上面绑定了两个事件处理函数，一个符合冒泡，一个符合捕获，点击一个元素后，是先捕获，还是先冒泡？

3、触发顺序，先捕获，后冒泡

同一个对象的一个事件处理类型，上面绑定了两个事件处理，分别执行事件冒泡和事件执行

例 html 部分沿用上一页冒泡情况的代码

```
wrapper.addEventListener('click', function () {
    console.log('wrapper')
}, true);
content.addEventListener('click', function () {
    console.log('content')
}, true);
box.addEventListener('click', function () {
    console.log('box')
}, true);

wrapper.addEventListener('click', function () {
    console.log('wrapperBubble')
}, false);
content.addEventListener('click', function () {
    console.log('contentBubble')
}, false);
box.addEventListener('click', function () {
    console.log('boxBubble')
}, false);
```

```
wrapper
content
box
boxBubble
contentBubble
wrapperBubble
```

先捕获，后冒泡

例　改变了冒泡和捕获的顺序

```
wrapper.addEventListener('click', function () {
    console.log('wrapperBubble')
}, false);
content.addEventListener('click', function () {
    console.log('contentBubble')
}, false);
box.addEventListener('click', function () {
    console.log('boxBubble')
}, false);

wrapper.addEventListener('click', function () {
    console.log('wrapper')
}, true);
content.addEventListener('click', function () {
    console.log('content')
}, true);
box.addEventListener('click', function () {
    console.log('box')
}, true);
```

```
wrapper
content
boxBubble
box
contentBubble
wrapperBubble
```

这个的顺序是先捕获红色，再捕获绿色，boxBubble 黄色区域事件执行，box 黄色区域事件执行，冒泡到绿色，冒泡到红色（谁先绑定，谁先执行，boxBubble 先绑定，所以先执行）

4、focus，blur，change，submit，reset，select 等事件不冒泡

取消冒泡和阻止默认事件

例不给 div 绑定事件处理函数，依然会冒泡（document 冒泡到 div 上）

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.onclick = function () {
        console.log('你闲的呀');
    }
```

例给 div 也帮个事件（点红色，也冒泡到 document）

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.onclick = function () {
        console.log('你闲的呀');
    }
    var div = document.getElementsByTagName('div')[0];
    div.onclick = function () {
        this.style.background = "green";
    }
```

在每一个事件处理函数中【div.onclick=function(){}】，我们可以写一个形参（如 e），系统可以传递**事件对象（记载了数据发生时的状态和信息）**到这个参数里面去

1、取消冒泡：

1）W3C 标准 event.stopPropagation();但不支持 ie9 以下版本

例事件对象上有一个 **event.stopPropagation();**取消冒泡事件

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.onclick = function () {
        console.log('你闲的呀');
    }
    var div = document.getElementsByTagName('div')[0];

    div.onclick = function (e) {

        e.stopPropagation();

        this.style.background = "green";
    }
```

2）IE 独有 event.cancelBubble = true;【实际上谷歌也实现了】

例 ie 里面事件对象上有一个 **event.cancelBubble = true;**能取消冒泡事件

```
div.onclick = function (e) {

    // e.stopPropagation();
    e.cancelBubble = true;

    this.style.background = "green";
}
```

3）封装取消冒泡的函数 stopBubble(event)

例

```
<div class="wrapper">
</div>
document.onclick = function () {
    console.log('你闲的呀');
}
var div = document.getElementsByTagName('div')[0];

div.onclick = function (e) {

    // e.stopPropagation();
    stopBubble(e);

    this.style.background = "green";
}
function stopBubble(event) {
    if(event.stopPropagation) {
        event.stopPropagation();
    }else{
        event.cancelBubble = true;
    }
}
```

2、阻止默认事件:

1）默认事件 — 表单提交，a 标签跳转，右键菜单等

例浏览器点右键出菜单，是一个事件（默认事件）

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.oncontextmenu = function () {
        console.log('a');
    }
```

2）return false; 兼容性非常好，以对象属性的方式注册的事件才生效（这是句柄的方式阻止默认事件，只有句柄的方式绑定事件才好使）

例

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.oncontextmenu = function () {
        console.log('a');
        return false;
    }
```

ele.onxxx = function (event) {}是句柄的绑定方式，才能用 return false;

3）event.preventDefault(); W3C 标注，IE9 以下不兼容

例

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.oncontextmenu = function (e) {
        console.log('a');
        e.preventDefault();
    }
```

4）event.returnValue = false; 兼容 IE

例

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.oncontextmenu = function (e) {
        console.log('a');
        e.returnValue = false;
    }
```

5）封装阻止默认事件的函数 cancelHandler(event);

例

```
<div class="wrapper">
</div>
<script type="text/javascript">

    document.oncontextmenu = function (e) {
        console.log('a');
        cancelHandler(e);
    }

    function cancelHandler(event) {
        if(event.preventDefault) {
            event.preventDefault();
        }else{
            event.returnValue = false;
        }
    }
```

阻止默认事件

不能用 return false

例 a 标签有一个跳转的默认时间，如何取消看下面

```
<a href="#">www.baidu.com</a>
br*100
<script type="text/javascript">

    // document.oncontextmenu = function (e) {
    //     console.log('a');
    //     cancelHandler(e);
    // }
    var a = document.getElementsByTagName('a')[0];
    a.onclick = function () {
        return false;
    }
    function cancelHandler(event) {
        if(event.preventDefault) {
            event.preventDefault();
        }else{
            event.returnValue = false;
        }
    }
```

右侧也能取消 a 标签的默认值

<a href = “avascript:void( )” >

```
<a href="javascript:void()" >demo</a>
<script type="text/javascript">
```

相当于写return

事件对象

非 ie 浏览器会把事件对象(记载了数据发生时的状态和信息)打包传到参数里面去。

ie 浏览器在 window.event 里面储存事件对象。

例

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
    div.onclick = function (e) {

        console.log(e);

    }
```

在非ie浏览器会记录对象

事件对象

lesson19

▶ MouseEvent {isTrusted: true, screenX: 83, screenY: 123, clientX: 55, clientY: 32…}

1、event || window.event 用于 IE

window.event 用于 IE，event 只能用于非 ie 浏览器

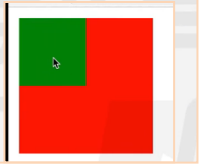
例这是储存事件对象的兼容性写法

```
<div style="width:100px;height:100px;background-color:red;"></div
<script type="text/javascript">

    var div = document.getElementsByTagName('div')[0];
    div.onclick = function (e) {

        var event = e || window.event;
        console.log(event);

    }
```

cancelBubble: false
cancelable: true
clientX: 76 鼠标坐标点
clientY: 77

例

```
<div class="wrapper" style="width:100px;height:100px;background-color:red;">
    <div class="box" style="width:50px;height:50px;background-color:green"></div>
</div>
<script type="text/javascript">

    var wrapper = document.getElementsByClassName('wrapper')[0];
    var box = document.getElementsByClassName('box')[0];

    wrapper.onclick = function (e) {

        var event = e || window.event;
        console.log(event);

    }
```



点红色会执行，点绿色会冒泡执行。点红色是点击到他自己来执行；点绿色身上，触发事件的点在绿色身上，是绿色传递的，我们**把触发事件的地方叫事件源。**

例事件对象上有个专门的信息是存储事件源的。

点击之后查看控制台 srcElement:这就是储存事件源的地方

▶ srcElement: div.wrapper

▶ srcElement: div.box

2、事件源对象:（找事件源对象的方法）

event.target 火狐独有的

event.srcElement Ie 独有的

这俩 chrome 都有

例事件源对象的兼容性写法

```
<div class="wrapper" style="width:100px;height:100px;background-color:red;">
    <div class="box" style="width:50px;height:50px;background-color:green"></div>
</div>
<script type="text/javascript">

    var wrapper = document.getElementsByClassName('wrapper')[0];
    var box = document.getElementsByClassName('box')[0];
    //事件源对象
    wrapper.onclick = function (e) {

        var event = e || window.event;
        var target = event.target || event.srcElement;
        console.log(target);

    }
```

求事件源对象

事件委托

例我们给每个 li 绑定事件，要求点哪个 li 就输出哪个内容，这不涉及闭包问题

```
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
    <li>4</li>
    <li>5</li>
    <li>6</li>
    <li>7</li>
    <li>8</li>
    <li>9</li>
    <li>10</li>
</ul>
<script type="text/javascript">

    var li = document.getElementsByTagName('li');
    var len = li.length;
    for(var i = 0; i < len; i ++) {
        li[i].onclick = function () {
            console.log(this.innerText);
        }
    }
```

上面的写法不好（如果是三千亿个 li 就没效率），不能动态，要用事件源和事件冒泡

事件委托：利用事件冒泡，和事件源对象进行处理

优点：

1. 性能 不需要循环所有的元素一个个绑定事件

2. 灵活 当有新的子元素时不需要重新绑定事件

例给每个 li 绑定事件，再增加 li 也能使用

```
    <li>6</li>
    <li>7</li>
    <li>8</li>
    <li>9</li>
    <li>10</li>
</ul>
<script type="text/javascript">

    var ul = document.getElementsByTagName('ul')[0];
    ul.onclick = function (e) {
        var event = e || window.event;
        var target = event.target || event.srcElement;
        console.log(target.innerText);

    }
```

扫雷的雷也可以放 ul 上

作业：预习下面的三个词，写拖拽功能（鼠标按住方块跟着动，松开就不跟着走）

```
onmouseenter onmouseleave onmousemove
      over         out
```

老版本

答案

```
<div style="width:100px;height:100px;background-color:red;position:absolute;left:0;top:0;"></div>
<script type="text/javascript">
    var div = document.getElementsByTagName('div')[0];

    div.onmousedown = function () {



        div.onmousemove = function (e) {
            var event = e || window.event;
            div.style.left = e.pageX + "px";
            div.style.top = e.pageY + "px";
        }

        div.onmouseup = function () {
            div.onmousemove = null;
        }


    }
```

鼠标在左上角

上面这个写法有问题，鼠标在方块的左上角，不在你点击他时候的位置

思路：上面是把点击鼠标的点设置成方块的 left 和 top（就是把鼠标的点设置成了左顶点，这样就少了个距离），我们把这个距离算上就可以了，加上鼠标第一次点击时候的离方块上边和左边的距离再赋给左上角了

之前是把 div.onmousemove 写在 div 上面，有一个 bug，我们按住鼠标迅速离开方块，方块就不会跟着动，一直按住鼠标，再一进去，又可以带着方块动了，是一个事件监听的问题（鼠标挪动频次大于事件监听频次），我们写成 document.onmousemove 就可以了

```
<div style="width:100px;height:100px;background-color:red;position:absolute;left:0;top:0;"></div>
<script type="text/javascript">
var div = document.getElementsByTagName('div')[0];
var disX,
    disY;
div.onmousedown = function (e) {
    disX = e.pageX - parseInt(div.style.left);
    disY = e.pageY - parseInt(div.style.top);
    document.onmousemove = function (e) {
        var event = e || window.event;
        div.style.left = e.pageX - disX + "px";
        div.style.top = e.pageY - disY + "px";
    }
    document.onmouseup = function () {
        div.onmousemove = null;
    }
}
```

上面这个写的很粗糙，没有封装也没有绑定。document.onmousemove=function(){}这种写法是不可以的，要用 **addEventListener** 绑定。作业就是把上面绑定成函数，函数名叫 function drag (elem) {}

例

```
function drag(elem) {
    var disX,
        disY;
    addEvent(elem, 'mousedown', function (e) {
        var event = e || window.event;
        disX = event.clientX - parseInt(getStyle(elem, 'left'));
        disY = event.clientY - parseInt(getStyle(elem, 'top'));
        addEvent(document, 'mousemove', mouseMove);
        addEvent(document, 'mouseup', mouseUp);
        stopBubble(event);
        cancelHandler(event);
    });
    function mouseMove(e) {
        var event = e || window.event;
        elem.style.left = event.clientX - disX + "px";
        elem.style.top = event.clientY - disY + "px";
    }
    function mouseUp(e) {
        var event = e || window.event;
        removeEvent(document, 'mousemove', mouseMove);
        removeEvent(document, 'mouseup', mouseUp);
    }
```

面试问题，什么是事件捕获，一个是冒泡，一个是捕获 **obj.addEventListener(type, fn, true);**他所说的第二种捕获不是事件处理模型，而是一种真实的事件获取的过程，用于解决拖拽鼠标出方块的问题

仅在 ie 好使，利用 div.setCapture();会捕获页面上发生的所有事情，都获取到自己身上。对应的用 div.releaseCapture();释放。但是方法比较老旧，一般不用。

事件分类

1、鼠标事件（不需要小驼峰和大驼峰）

click、mousedown、mousemove、mouseup、contextmenu、mouseover、mouseout、mouseenter、 mouseleave

例 click=mousedown+mousemove

```
document.onclick = function () {
    console.log('click');
}

document.onmousedown = function () {
    console.log('mousedown');
}

document.onmouseup = function () {
    console.log('mouseup')
}
```

```
mousedown
mouseup
click
```

这三个事件的触发顺序是 mousedown，mouseup，click

例 contextmenu 右键取消菜单，mousemove 是鼠标移动的事件

例相对应 mouseover、mouseout 鼠标覆盖区域与 mouseenter 、mouseleave 鼠标离开但是 mouseenter 、mouseleave 是 html5 的，都是鼠标进去，出来发生的变化

```
var div = document.getElementsByTagName('div')[0];
div.onmouseover = function () {
    div.style.background = "yellow";
}

div.onmouseout = function () {
    div.style.background = "green";
}
```

2、用 button 来区分鼠标的按键，0/1/2

只有 mouseup、mousedown 两个能区分鼠标垫左右键

button 返回值，右键是 0，左键是 2，中间是 1

例

```
document.onmousedown = function (e) {
    if(e.button == 2) {
        console.log('right');
    }else if(e.button == 0){
        console.log('left');
    }
}
```

```
MouseEvent {isT
screenY: 328, c
  altKey: false
  bubbles: true
  button: 0
```

3、DOM3 标准规定:click 事件只能监听左键,只能通过 mousedown 和 mouseup 来判断鼠标键

例 click 不能监听右中

```
document.onclick = function (e) {
    console.log(e);
}
```

4、如何解决 mousedown 和 click 的冲突

事件练习作业

1、拖拽应用：实现拖拽正常拖，但是点击就正常跳转

思路：拖拽由 down+move+up，但是不管隔多久都算一个 click，可以理解成拖拽不等于点击 click。看时间差解决：按下+抬起的时间差>多少，就能知道是拖拽了。

答案：按下之后才绑定 move 事件，抬起了 move 事件就解除

```
<div style="width:100px;height:100px;background-color:red;"></div>
<script type="text/javascript">
    var firstTime = 0;
    var lastTime = 0;
    var key = false;
    document.onmousedown = function () {
        firstTime = new Date().getTime();

    }
    document.onmouseup = function () {
        lastTime = new Date().getTime();
        if(lastTime - firstTime < 300) {
            key = true;
        }

    }

    document.onclick = function () {
        if(key) {
            console.log('click');
            key = false;
        }
    }
```

2、应用 mousedown mousemove mouseup

3、随机移动的方块（完全随机）

思路：当把鼠标挪到方块里面去 onmouseover 的时候，方块随机向八个方向挪动

4、mouseover

移动端 onmousedown 不能用，只能用 touchstart,touchmove,touchend

事件分类

一、键盘事件

1、keydown，keyup，keypress

2、触发顺序是 keydown > keypress > keyup

3、keydown 和 keypress 的区别

1）keydown 可以响应任意键盘按键，keypress 只可以相应字符类键盘按键

检测字符类不准确，keypress 检测字符很准。但是 keydown 能监控所有，包括上下左右都能监控，但是 keypress 只能监视字符。

用法：如果你想监控字符类按键，并想区分大小写，就用 keypress，如果是操作类按键的话，就用 keydown（which：39 是给按键牌号 39，不是 asc 码）

2）keypress 返回 ASCII 码，可以转换成相应字符

例连续按键盘按键的时候就是连续触发 keydown 和 keypress，松开触发 keyup

```
document.onkeypress = function () {
    console.log('keypress');
}

document.onkeydown = function () {
    console.log('keydown');
}
document.onkeyup = function () {
    console.log('keyup');
}
```

```
keydown
keypress
keydown
keypress
keydown
keypress
keydown
keypress
keyup
```

注意：游戏触发设置在 keydown 上，机械键盘抬起速度快反馈力量大，对游戏没用

小写的 a 的 charCode 返回的是

例利用下面，把 Unicode 编码转成对应值

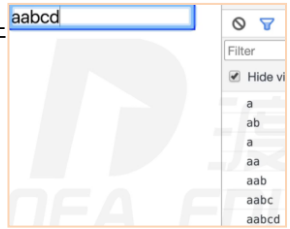
```
document.onkeypress = function (e) {
    console.log(String.fromCharCode(e.charCode));
}
```

二、文本操作事件

input，change，focus，blur

例框里面所有变化（增删改）都会触发 input 事件

```
<style type="text/css">

    input{
        border:1px solid #01f;
    }

</style>
```




```
var input = document.getElementsByTagName('input')[0];
input.oninput = function (e) {
    console.log(this.value);
}
```

例 change 对比鼠标聚焦，或失去焦点的时，两个状态是否发生改变，如果两个状态没有改变就不触发，如果发生改变就触发

```
var input = document.getElementsByTagName('input')[0];
input.onchange = function (e) {
    console.log(this.value);
}
```

例 focus，blur 聚焦和失去焦点

```
<input type="text" value="请输入用户名" style="color:#999" onfocus="
if(this.value=='请输入用户名'){this.value='';this.style.
color='#424242'}" onblur="if(this.value==''){this.
value='请输入用户名';this.style.color='#999'}">
```

三、窗体操作类(window 上的事件) scroll，load

例 **scroll** 当滚动条一滚动，scroll 事件就触发了

```
window.onscroll = function () {

    console.log(window.pageXOffset + " " + window.pageYOffset);

}
```

ie6 没 fixed 定位，就用 position 的 top=原来的 top+滚动条的位置来写。absolute 定位相对于文档定位，用 absolute 定位模拟 fixed 定位，挪动距离加载 absolute 的 top 上读到 js 的时候就阻断页面，所以必须把 js 写在下面才能把上面的东西读出来。

例 **load** 重要但是不用。window.onload 发生在什么时候

```
<script type="text/javascript">
window.onload = function () {
    var div = document.getElementsByTagName('div')[0];
    console.log(div);
    div.style.width = "100px";
    div.style.height = "100px";
    div.style.backgroundColor = "red";
}
</script>

<div></div>
```

利用了 onload 就能操作写在下面的 div 了，但是我们不能这样用。

理由：html 和 css 是一起解析的，在解析的时候会有 html 有 domTree，css 有 cssTree 生成（树形图的顶底是 document，然后是 html，然后是 head，body），两个树拼在一起是 renderTree。

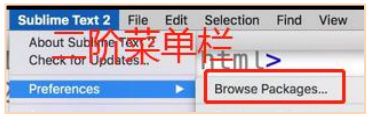
什么时候把节点放在树里？dom 节点解析，如确定是 img 标签就把他放到树里。（先解析完 img，同时开启一个线程异步的去下载里面的内容，后下载完）。
我们把 js 的 script 标签写在最下面的好处是，这些刚刚解析完 js 就能操作页面了，就更快了。而 window.onload 要等整个页面解析完，下载完才能操作 js，才能触发事件（效率很差）。onload 能提醒我们什么时候整个页面解析完毕。在设计广告时，就要用 onload，等整个页面下载完了才开始用，但是 onload 绝对用于主程序里面。

例
```
<div></div>
<span></span>
<img src="">

<script type="text/javascript">

    window.onload = function () {

    }

</script>
```
所有的页面加载完毕，才能执行 onload

小练习:fixed 定位 js 兼容版 //ie6 没有 fixed 定位，用 position:absolution 是相对于文档定位。模拟 fixed 定位 跟着解决文档拖的问题。position-top= 原来的 position-top + 滚动条的位置

作业

1、完善轮播图，加按钮 //提示：用留的接口，加两个按钮，改参数
2、提(qie)取密码框的密码 //提示：监听键盘事件
3、输入框功能完善
4、菜单栏（二阶菜单栏） //提示：先写一个 div，开始的时候 display：none，等鼠标挪上去，变成 display：block



5、贪食蛇游戏 //提示：蛇头动，蛇尾怎么跟着动，控制蛇头操作就可以了，蛇尾的每一节跟着前一节动
6、扫雷游戏 //注意闭包，有个扩散算法。
二维数组就是数组里面套数组(像矩阵)，可以表示坐标点

arr[1][1]说的是第二行第二列，也就是 22

```
var arr = [
[1  ,2  ,3  ],
[11 ,22 ,33 ],
[111,222,333]
    ]
arr[1][1]
```

可以尝试做：球击方块（板子）的游戏

json

1、JSON 是一种传输数据的格式（以对象为样板，本质上就是对象，但用途有区别，对象就是本地用的，json 是用来传输的）
2、JSON.parse(); string — > json
3、JSON.stringify(); json — > string

例 json 的属性名必须加双引号(传的是二进制文本)

```
json
    属性名必须双引号
{
    "name" : "deng",
    "age" : 123
}
```

例
```
var obj = {
    "name" : "abc",
    "age" : 123
}
```
```
> JSON.stringify(obj)
"{"name":"abc","age":123}"
```

例
```
<script type="text/javascript">
    var obj = {
        "name" : "abc",
        "age" : 123
    }

    var str = JSON.stringify(obj);
```
```
> str
"{"name":"abc","age":123}"
> JSON.parse(str)
▶ Object {name: "abc", age: 123}
```

例
```
<meta charset="UTF-8">
<title>Document</title>
<style type="text/css">
    div{
        width:100px;
        height:100px;
        background-color: red;
    }

</style>
```
```
    <div>
        <strong></strong>
    </div>
    <span></span>
    domTree
```


思路：绘制 dom 树，符合深度优先（纵向）原则，比如先看 head → title → meta → body → div → strong → span。
dom 树是节点解析，dom 树解析完毕代表 dom 数所有的节点解析完毕，不代表加载（下载完毕）完毕。如看到 img 标签就放到 dom 树上，然后同时下载。
dom 树形成完了以后，就等 css 树形成【cssTree 也是深度优先原则。】。
domTree + cssTree = randerTree，randerTree 形成以后才，渲染引擎才会绘制页面，
domTree 改变，randerTree 也会改变，会重排，影响效率,要尽量避免重排。
randerTree 触发重排（reflow）的情况：dom 节点的删除，添加，dom 节点的宽高变化，位置变化，display none ==> block，offsetWidth，offsetLeft
repaint 重绘：效率也比较低，效率影响较小。触发情况：改颜色，图片

异步加载 js

js 是单线程的，会阻断 HTML，css 加载（因为 js 会修改 html 和 css 一起加载会乱），所以是同步加载 js。先下载 js，再下载 HTML 和 css。**常规来说 js 是同步加载的，所以我们讲讲 js 异步加载的情况**

一、js 加载的缺点：加载工具方法没必要阻塞文档，过得 js 加载会影响页面效率，一旦网速不好，那么整个网站将等待 js 加载而不进行后续渲染等工作。

二、有些工具方法需要按需加载，用到再加载，不用不加载。

三、javascript 异步加载的三种方案

1、defer 异步加载，但要等到 dom 文档全部解析完（dom 树生成完）才会被执行。只有 IE 能用。

dom 文档全部解析完，不代表整个页面加载完

2、async 异步加载，加载完就执行，async 只能加载外部脚本，不能把 js 写在 script 标签里。ie9 以上可以用，w3c 标准

1 和 2 执行时也不阻塞页面

3、创建 script，插入到 DOM 中，加载完毕后 callBack（按需加载，方便）→常用

例 js 异步加载，属性名和属性值相同可以只写一个 defer=“defer”

```
<script type="text/javascript" src="tools.js" defer="defer"></
    script>
```

执行异步加载，属性名和属性值相同可以只写一个

例这样也可以异步加载

```
<script type="text/javascript" defer="defer">
    var a = 123;
</script>
```

例 async 异步加载

```
<script type="text/javascript" aysnc="aysnc" src="tools.js">
        这里面不能写var等等脚本
</script>
```

例创建 script，插入到 DOM 中，加载完毕后 callBack

```
var script = document.createElement('script');
script.type = "text/javascript";
script.src = "tools.js";
如果不写这一段，就是只加载，不执行。像这样写了执行以后才执行
document.head.appendChild(script);
```

例预加载机制：img 的灯塔模式

```
<body>

    <script type="text/javascript">

        var script = document.createElement('
            script');
        script.type = "text/javascript";
        script.src = "demo.js";
                                    异步加载

        document.head.appendChild(script);
```

```
1 alert('老邓不是特别很磕碜')
```

例

```
<script type="text/javascript">
                                    创建
    var script = document.createElement('
        script');
    script.type = "text/javascript";  设置
    script.src = "demo.js";          下载

    document.head.appendChild(script);


    test();


</script>
```

```
function test() {
    console.log('a');
}
```

```
▶ Uncaught ReferenceError: test is not defined
    at lesson23.html:16
```

上面这种方式不能执行，会报错

例

```
<script type="text/javascript">

    var script = document.createElement('
        script');
    script.type = "text/javascript";
    script.src = "demo.js";

    document.head.appendChild(script);


    setTimeout(function () {

        test();

    },1000);
```

```
function test() {
    console.log('a');
}
```

```
a
>
```

思考：为什么上一个例子当前执行，执行不了，但是设置定时器以后就能执行了？

答案：因为还没有下载完。因为程序执行是非常快的，当程序读到 document.head 读到 test()时上面的 script.type 和 script.src 还没有下载完，所以执行不了。

所以，能不能有一个东西提示我们，他下载完了，等他下载完了我们在用？

方法一：非 ie 方法 script.onload = function ( ) { }，触发 script.onload 事件就代表他下载完了，当他们下载完了再执行 test。例

```
<script type="text/javascript">

    var script = document.createElement('
        script');
    script.type = "text/javascript";
    script.src = "demo.js";

    script.onload = function () {
        test();
    }
    document.head.appendChild(script);


</script>
```

```
function test() {
    console.log('a');
}
```

方法二：ie 上有一个状态码，script.readyState，功能与 script.onload 相似

script.readyState = “loading”;最开始的值

script.readyState = “complete”;或者“leaded”表示加载完成

例

```
<script type="text/javascript">

    var script = document.createElement('
        script');
    script.type = "text/javascript";
    script.src = "demo.js";

    script.readyState = "loading";
```

例监听这个方法的事件

```
<script type="text/javascript">

    var script = document.createElement('script');
    script.type = "text/javascript";
    script.src = "demo.js";
    if(script.readyState) {
        script.onreadystatechange = function () {//Ie
            if(script.readyState == "complete" || script.re
                adyState == "loaded") {
                test();
            }
        }
    }else{

        script.onload = function () {//Safari chrome firefox
        opera
            test();
        }
    }

    document.head.appendChild(script);
```

例我们把以上两种异步加载 js 方法封装成函数：(**左下是方法三的封装库**)

```
<script type="text/javascript">

    function loadScript(url, callback) {
        var script = document.createElement('script');
        script.type = "text/javascript";
        if(script.readyState) {
            script.onreadystatechange = function () {//Ie
                if(script.readyState == "complete" || script.
                    readyState == "loaded") {
                    callback();
                }
            }
        }else{
            script.onload = function () {//Safari chrome
            firefox opera
                callback();
            }
        }
        script.src = url;
        document.head.appendChild(script);
    }


</script>
```

事件里面有一个绑定的事件处理函数，当满足一定执行条件才执行的函数叫做回调函数。回调函数叫 callback

例把上面那个函数折叠，加上下面

```
<script type="text/javascript">

    function loadScript(url, callback) {
    }

    loadScript('demo.js', test);
</script>
```

此刻未被定义

```
function test() {
    console.log('a');
}
```

```
▶ Uncaught ReferenceError: test
is not defined
```

执行顺序：先 function loadScript(){}【不会看里面的代码是什么】，再 loadScript()【这一步的时候不知道 test 是什么】，然后执行 function 里面的内容

为了解决上面的问题，如下例：

```
<script type="text/javascript">

    function loadScript(url, callback) {
    }

    loadScript('demo.js', function () {
        test();
    });
</script>
```

```
function test() {
    console.log('a');
}
```

利用 callback 变成字符串形式，（不让用 eval，只做拓展），例

```
function test() {
    console.log('a');
}
```

```
function loadScript(url, callback) {
    var script = document.createElement('
        script');
    script.type = "text/javascript";
    if(script.readyState) {
        script.onreadystatechange = function
            () {//Ie
            if(script.readyState == "
                complete" || script.re
                adyState == "loaded") {
                eval(callback);
            }
        }
    }else{
        script.onload = function () {//
        Safari chrome firefox opera
            eval(callback);
        }
    }
    script.src = url;
    document.head.appendChild(script);
}

loadScript('demo.js', "test");
```

右下是更好的方法，需要配合库实现，例

```
var tools = {

    test : function () {
        console.log('a')
    },
    demo : function () {
    }
}
```

```
function loadScript(url, callback) {
    var script = document.createElement('
        script');
    script.type = "text/javascript";
    if(script.readyState) {
        script.onreadystatechange = function
            () {//Ie
            if(script.readyState == "
                complete" || script.re
                adyState == "loaded") {
                tools[callback]()
            }
        }
    }else{
        script.onload = function () {//
        Safari chrome firefox opera
            tools[callback]();
        }
    }
    script.src = url;
    document.head.appendChild(script);
}

loadScript('demo.js', "test");
```

js 加载时间线（可以理解成浏览器加载时间线）背

js 加载时间线：依据 js 出生的那一刻起，记录了一系列浏览器按照顺序做的事（就是一个执行顺序）

js 时间线步骤（创建 document 对象==>文档解析完==>文档解析完加载完执行完）

1、创建 Document 对象，开始解析 web 页面。解析 HTML 元素和他们的文本内容后添加 Element 对象和 Text 节点到文档中。这个阶段 document.readyState = 'loading'。

2、遇到 link 外部 css，创建线程，进行异步加载，并继续解析文档。

3、遇到 script 外部 js，并且没有设置 async、defer，浏览器同步加载，并阻塞，等待 js 加载完成并执行该脚本，然后继续解析文档。

4、遇到 script 外部 js，并且设置有 async、defer，浏览器创建线程异步加载，并继续解析文档。

对于 async 属性的脚本，脚本加载完成后立即执行。（异步禁止使用 document.write()，因为当你整个文档解析到差不多，再调用 document.write()，会把之前所有的文档流都清空，用它里面的文档代替）

5、遇到 img 等（带有 src），先正常解析 dom 结构，然后浏览器异步加载 src，并继续解析文档。 看到标签直接生产 dom 树，不用等着 img 加载完 scr。

6、当文档解析完成（domTree 建立完毕，不是加载完毕），document.readyState = 'interactive'。

7、文档解析完成后，所有设置有 defer 的脚本会按照顺序执行。（注意与 async 的不同,但同样禁止使用 document.write()）；

8、document 对象触发 DOMContentLoaded 事件，这也标志着程序执行从同步脚本执行阶段，转化为事件驱动阶段。

9、当所有 async 的脚本加载完成并执行后、img 等加载完成后（页面所有的都执行加载完之后），document.readyState = 'complete'，window 对象触发 load 事件。

10、从此，以异步响应方式处理用户输入、网络事件等。

例异步禁止使用 document.write()

```
<body>
    <div style="width:100px;height:100px;background-color:red;"></div>

    <script type="text/javascript">
        document.write('a');
    </script>
```

例用 window.o nload 会等整个页面加载完才执行，消除文档流（把自己 script 删了）

```
<body>
    <div style="width:100px;height:100px;background-color:red;"></div>

    <script type="text/javascript">
        window.onload = function () {
            document.write('a');
        }
    </script>
```

```
<html>
  <head></head>
  <body>a</body>
</html>
```

例执行到 document.readyState 时，整个 dom 树还没有解析完，所以不会是 interactive

```
<body>
    <div style="width:100px;height:100px;background-color:red;"></div>

    <script type="text/javascript">
        console.log(document.readyState);
    </script>


    <script type="text/javascript">
        console.log(document.readyState);
    </script>

</body>
```

```
loading
loading
```

例我们就利用 window.onload 事件，看到的是 complete，代表执行加载完

```
<div style="width:100px;height:100px;background-color:red;"></div>

<script type="text/javascript">
    console.log(document.readyState);
    window.onload = function () {
        console.log(document.readyState);
    }
</script>
```

```
loading
complete
```

例如果想看到 interactive，**所有事件都是用的小写**

```
<div style="width:100px;height:100px;background-color:red;"></div>

<script type="text/javascript">
    console.log(document.readyState);

                事件全部用小写
    document.onreadystatechange = function () {
        console.log(document.readyState);
    }

</script>
```

```
loading
interactive
complete
```

例

```
<div style="width:100px;height:100px;background-color:red;"></div>

<script type="text/javascript">
    console.log(document.readyState);

    document.onreadystatechange = function () {
        console.log(document.readyState);
    }

    document.DOMContentLoaded = function () {
        console.log('a');
    }
</script>
```

```
loading
interactive
complete
```

例这个事件只在 addEventListener 上面能用

```
<div style="width:100px;height:100px;background-color:red;"></div>

<script type="text/javascript">
    console.log(document.readyState);

    document.onreadystatechange = function () {
        console.log(document.readyState);
    }

    document.addEventListener('DOMContentLoaded',function () {
        console.log('a');
    },false);
</script>
```

只在这一块好用

```
loading
interactive
a
complete
```

例通用写法是把 JS 的 script 写在最下面，
为什么我们要把他写在最下面？
写在最下面意味着上面的 dom 已经处理完毕了。
window.onload 是整个页面加载完才执行，慢
等 dom 解析完毕就执行完就执行，比较快。
如右侧 →

```
window.onload 区别：慢

当dom解析完就执行的部分
$(document).ready(function () {


                解析完就执行，快
})
```

只要有一个图片没加载完，window.onload 就不能用，所以效率非常低下

例 script 标签这样写在上面。又能操作 div，又能 dom 解析完就处理，效率很高。但是最好还是写在最下面

```
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript">
        document.addEventListener('DOMContentLoaded', function () {
            var div = document.getElementsByTagName('div')[0];
            console.log(div);
        },false);
    </script>
</head>
<body>
<div style="width:100px;height:100px;background-color:red;"></div>
```

dom解析完就完事。效率更高

BOM（权限过大，不让用，见就业班）

定义：Browser Object Model，定义了操作浏览器的接口

BOM 对象: Window, History,Navigator,Screen, Location 等

由于浏览器厂商的不同，Bom 对象的兼容性极低。一般情况下，我只用其中的部分功能。

Window

History 对象

Navigator 对象

http://www.w3school.com.cn/jsref/dom_obj_navigator.asp

Screen 对象

Location 对象

location.hash

“#”后是对浏览器操作的，对服务器无效，实际发出的请求也不包含“#”后面的部分

“#”被算作历史记录

课前补充

转义字符 “\”，反斜杠

例 var str = “abcdedf”

var str = “abcd\ “edf”

会把 \ 后面接的字符取消原来的意思，变成文本

想打一个转移符号\文本，就用\\就可以了

\n 是换行

例 **\n** 只在 console.log 里面有效，在 document.write 里面没有用，会变成一个空格

\r 是行结束符

例\r 代表行结束符，但是写成\r\n 才看的出来是回车了。

\t 是 table，缩进符

例\t 是 table，代表四个空格，缩进符

例系统不允许字符串多行显示：

```
<script type="text/javascript">
    document.body.innerHTML = "
    <div></div>
<span></span>";
    var str = "abcdedf";

    console.log(str);
```

```
▼ Uncaught SyntaxError: Invalid or
unexpected token
```

例我又想写多行字符串，又不能让系统报错，就加速转义字符\，意思是把后面的回车转成字符串，如右下方：

```
<script type="text/javascript">
    document.body.innerHTML = "\
    <div></div>\
    <span></span>\
    ";

    var str = "abcdedf";

    console.log(str);
```

意思是把后面的回车转换成字符串

或者用字符串连接，左下（low）：

```
<script type="text/javascript">
    var test =
    "<div></div>" +
    "<span</span>" +


    var str = "abcdedf";

    console.log(str);
```

low

RegExp 正则表达式

正则表达式的作用：匹配特殊字符或有特殊搭配原则的字符的最佳选择。

RegExp 对象表示正则表达式，它是对字符串执行模式匹配的强大工具（判断字符串满不满足要求）。正则表达式其他语言也有，规则是一样的，在这个基础上，js 增加了自己独特的方法。

例

```
> var input =
  document.getElementsByTagName('input')[0]
< undefined
> input.value
< "cheng.ji@alibaba-inc.com"
```

㈠. 基础语法

创建一个正则表达式

第一种方法：正则表达式字面量//　　【推荐使用这种】

例 var reg = /pattern/;

```
var reg = /abc/;

var str = "abcd";
```

```
> reg.test(str)
< true
```

例：意思是正则表达式测验一下字符串含不含有我规定的片段

```
var reg = /abce/;

var str = "abcd";
```

```
> reg.test(str)
< false
```

例像下面这种，虽然都有 abce，但是没挨着，或者排序不对，都不算

```
var reg = /abce/;

var str = "abcde";
```

```
> reg.test(str)
< false
```

```
var reg = /ab/;
var str = "abababababab";
```

```
> str.match(reg)
< ▶ ["ab"]
```

第二种方法：new RegExp();

var reg = new RegExp(“pattern”);

例：括号里面写的是规则

```
var str = "abcd";
var reg = new RegExp("abc");
```

```
> reg.test(str)
< true
```

例下面是给正则表达式增加属性的方法：

```
var str = "abcd";
var reg = new RegExp("abc", "m");
```

例在正常情况下，下面两个虽然是形式一样，但是是两个不同的正则表达式

```
var reg = /abce/m;

var reg1 = new RegExp(reg);
```

```
> reg
< /abce/m
> reg1
< /abce/m
> reg.abc = 123
< 123
> reg1.abc
< undefined
```

```
> reg
< /abce/m
> reg1
< /abce/m
> reg.abc = 123
< 123
> reg1.abc
< 123
```

例但是有一个非正常的情况：

```
var reg = /abce/m;

var reg1 = RegExp(reg);
```

下面是给正则表达式增加属性（修饰符）：

i 是 ignoreCase，是忽视大小写的意思，例 var reg = /abce/i;

g 是全局匹配的意思，例 var reg = /abce/ g;

m 执行多行匹配，例 var reg = /abce/ m;

例

```
var reg = /abce/i;

var str = "ABCEd";
```

```
> reg.test(str)
< true
```

```
var reg = /ab/;
var str = "abababababab";
```

```
> str.match(reg)
< ▶ ["ab"]
```

```
var reg = /ab/g;
var str = "abababababab";
```

```
> str.match(reg)
< ▶ ["ab", "ab", "ab", "ab", "ab"]
```

^a 的意思是必须是这个 a 开头，那么下面这串就只有一个

例

```
var reg = /^a/g;
var str = "abcdea";
```

```
> str.match(reg)
< ▶ ["a"]
```

```
var reg = /^a/g;
var str = "abcde\na";
```

```
> str.match(reg)
< ▶ ["a"]
```

例因为上面这种写法没有多行匹配的功能，直到给他加上了 m

```
var reg = /^a/gm;
var str = "abcde\na";
```

```
> str.match(reg)
< ▶ ["a", "a"]
```

上面这三种方法还能组合到一起，例 var reg = /abce/igm;

正则表达式的表达式：

reg.test( );　只能判断这个字符串有没有符合要求的片段 ,返回结果只有 true 和 false

str.match( );　可以把所有东西都匹配出来，返回给你，比上一种方法更直观，还能告诉你返回了多少个

例 :想把第一、二、三位都是数字都匹配出来，不同，可变化的数字，可以用[]来写，一个[]代表一位数，[]里面放的是范围，是一个区间，如[1234567890，如下图：

```
var reg = /[1234567890][1234567890][1234567890]/g;
var str = "12309u98723zpoixcuypiouqwer"
```

```
> str.match(reg)
< ▶ ["123", "987"]
```

例

```
var reg = /[ab][cd][d]/g;
var str = "abcd";
```

```
> str.match(reg)
< ▶ ["bcd"]
```

例 0-9 是指 0 到 9，A-Z 是指大 A 到大 Z，a 到 z 是指小 a 到小 z

```
var reg = /[0-9A-Za-z][cd][d]/g;
var str = "abcd";
```

这个是按照 ASC 码排序的，**从大写到小写可以写成 A-z**

```
var reg = /[0-9A-z][cd][d]/g;
var str = "ab1cd";
```

```
> str.match(reg)
< ▶ ["1cd"]
```

例^在表达式里面和外面的意思是不一样的，在里面是非的意思

```
var reg = /[^a][^b]/g;
var str = "ab1cd";
```

```
> str.match(reg)
< ▶ ["b1", "cd"]
```

例正则表达式中 | 是或的意思，下面是匹配 abc 或 bcd

```
var reg = /(abc|bcd)/g;
var str = "abc";
```

```
> str.match(reg)
< ▶ ["abc"]
```

```
var reg = /(abc|bcd)[0-9]/g;
var str = "bcd2";
```

```
> str.match(reg)
< ▶ ["bcd2"]
```

正则表达式的元字符和表达式是一个东西

元字符（Metacharacter）是拥有特殊含义的字符：

\w 代表一位，w 意思是 world， \w===完全等于[0-9A-z_]

\W === [^\w]　大写\W 是非的\w

例
```
var reg = /\Wcd2/g;
var str = "bcd2";
```
```
> str.match(reg)
null
```

```
var reg = /\Wcd2/g;
var str = "b*cd2";
```
```
> str.match(reg)
▶ ["*cd2"]
```

\d === [0-9]

\D ==== [^\d] ， \D ==== [^0-9]

例
```
var reg = /[\w\d]/g;
var str = "s";
```
```
> str.match(reg)
▶ ["s"]
```

表达式里面可以写元字符

\s === [空白字符，即\t\n\r\v\f]　最常用的是空格和换行

在正则表达式中写空格就代表空格

\S ==== [非空白字符]

空白字符可以是： 最常用的是空格和换行	
制表符(tab character)	**\t**
回车符(carriage return character)	**\r**
换行符 (new line character)	**\n**
垂直换行符 (vertical tab character)	**\v**
换页符 (form feed character)	**\f**

\b === 单词边界

\B ==== 非单词边界

单词边界在字符串里面可以看成 （空格）

例下面 str 有六个单词边界，写成 var reg = /\bc/g;意思是 c 前面要有一个单词边界
```
var reg = /\bcde/g;
var str = "abc cde fgh";
```
```
> str.match(reg)
▶ ["cde"]
```

```
var reg = /\bc/g;
var str = "abc cde fgh";
```
```
> str.match(reg)
▶ ["c"]
```

例 cde 前面要是一个单词边界，后面也要上一个单词边界
```
var reg = /\bcde\b/g;
var str = "abc cde fgh";
```
```
> str.match(reg)
▶ ["cde"]
```

```
var reg = /\bcde\b/g;
var str = "abc cdefgh";
```
```
> str.match(reg)
null
```

例
```
var reg = /\bcde\B/g;
var str = "abc cdefgh";
```
```
> str.match(reg)
▶ ["cde"]
```

例下面的　　是一个制表符\t 打出来的，但是系统是不能识别的
```
var reg = /\tc/g;
var str = "abc  cdefgh";
```
```
> str.match(reg)
null
```

要写成以下形式才能识别\t
```
var reg = /\tc/g;
var str = "abc\tcdefgh";
```
```
> str.match(reg)
▶ ["\tc"]
```

例\r 匹配回车
```
var reg = /\nc/g;
var str = "ab\nccdefgh";
```
```
> str.match(reg)
▶ ["↵c"]
```

\uxxxx　这是简单的四位的 Unicode 编码（包含了汉字）

查找以**十六进**制数 xxxx 规定的 Unicode 字符。

还有**六位数 Unicode 编码（前两位代表的是层，后面代表的是范围）**

\u010000 - \u01ffff　第一层 Unicode 编码的区间

\u020000 - \u02ffff　第二层 Unicode 编码的区间

\u100000 - \u10ffff　第十六层 Unicode 编码的区间

经常用的是第一层，一般会忽略 01，写成\uxxxx。

例
```
var reg = /\u8001\u9093\u8eab\u4f53\u597d/g;
var str = "老邓身体好";
```
```
> str.match(reg)
▶ ["老邓身体好"]
```

例 **Unicode 编码也可以写成区间，如下**
```
var reg = /[\u3000-\ua000]/g;
var str = "老邓身体好";
```
```
> str.match(reg)
▶ ["老", "邓", "身", "体", "好"]
```

注意：代表一切的集合 var reg = /[/s/S]/;或者/[/d/D]/这样类似的

.	查找单个字符，除了换行和行结束符。/. ==== [^\r\n]

例
```
var reg = /./g;
var str = "老 邓身体好";
```
```
> str.match(reg)
▶ ["老", " ", "邓", "身", "体", "好"]
```

正则表达式的量词（代表数量的词）

n+　匹配任何包含至少一个 n 的字符串。这个变量可以出现 1 到无数次。

n*　匹配任何包含零个或多个 n 的字符串。这个变量可以出现 0 到无数次。

n 是一个变量，n+代表这个变量可以重复出现 1 次到无数次，n*代表{0 到正无穷}

例
```
var reg = /\w+/g;
var str = "abc"
```
```
> str.match(reg)
▶ ["abc"]
```

```
var reg = /\w*/g;
var str = "abc"
```
```
> str.match(reg)
▶ ["abc", ""]
```

光标在 c 后面，c 后面有逻辑上的距离，*如果是零，匹配的是空。\w 会先把能识别值的先识别，到最后识别不了，才试一下*零，匹配空

例
```
var reg = /\d*/g;
var str = "abc"
```
```
> str.match(reg)
▶ ["", "", "", ""]
```
因为光标在 a 前面，逻辑上有距离，匹配了一个零，空串；然后光标移动 a 后面，逻辑上有距离，又匹配了一个零，空串，所以有多少个光标定位点就有多少个

例
```
var reg = /\w+/g;
var str = "aaaaaaaaaaaaaaa"
```
```
> str.match(reg)
▶ ["aaaaaaaaaaaaaaa"]
```

正则表达式有一个贪婪匹配原则，能多就不少

n?　匹配任何包含零个或一个 n 的字符串。这个变量 0 或 1 个一匹配。

例最后会匹配一个空
```
var reg = /\w?/g;
var str = "aaaaaaaaaaaaaaa"
```
```
> str.match(reg)
▶ ["a", "a", "a", "a", ""]
```

n{X}　匹配包含 X 个 n 的序列的字符串。

例**如 n{3}就是三个三个一匹配**
```
// n{X}  {x}
var reg = /\w{3}/g;
var str = "aaaaaaaaaaaaaaa"
```
```
> str.match(reg)
▶ ["aaa", "aaa", "aaa", "aaa", "aaa"]
```

n{X,Y}　匹配包含 X 至 Y 个 n 的序列的字符串。

例下面要符合贪婪匹配原则，/\w{3,5}/就是能 5 个就不 4 个，能 4 个就不 3 个
```
var reg = /\w{3,5}/g;
var str = "aaaaaaaaaaaaaaaaaa"
```
```
> str.match(reg)
▶ ["aaaaa", "aaaaa", "aaaaa", "aaa"]
```

n{X,}　匹配包含至少 X 个 n 的序列的字符串。

例/\w{1,}/　{1,}和 n+是一样的，指匹配 1 到无数个。

例/\w{2,}/　{2,}是匹配 2 到无数个

思考：量词，量的是什么？

答案：n*是 n 乘以这个量词，/\w{2,}/不是\w 的结果乘以量词，如果\w 的结果乘以这个量词，那么匹配的都是一致的。量词量的是 w，是\w 乘很多东西，再每个 w 再随机匹配。而不是\w 先匹配结果，再乘以量词（例如先匹配一个 a，再乘以那么多 a 就匹配不成来了，所以不能这么理解）

例
```
var reg = /\w{2,}/g;
var str = "abcded"
```
```
> str.match(reg)
▶ ["abcded"]
```

n$　匹配任何结尾为 n 的字符串。

例
```
var reg = /abc/g;
var str = "abcded"
```
```
> str.match(reg)
▶ ["abc"]
```

例^abc 是以 a 开头的 abc
```
var reg = /^abc/g;
var str = "abcded"
```
```
> str.match(reg)
▶ ["abc"]
```

例$是以 d 结尾，这个匹配不出来
```
var reg = /abcd$/g;
var str = "abcded"
```

例以 ed 结尾
```
var reg = /ed$/g;
var str = "abcded"
```
```
> str.match(reg)
▶ ["ed"]
```

例
```
var reg = /^abc$/g;
var str = "abcabc"
```
```
> str.match(reg)
null
```

上面这道题应该理解成：以当前这个 abc 开头，以这个 abc 结尾的字符串

例
```
var reg = /^abc$/g;
var str = "abc";
```
```
> str.match(reg)
▶ ["abc"]
```

作业：写一个字符串，检验一个字符串首尾是否含有数字。

首尾都含有的意思是需要加“都”；首尾的意思是首有，或，尾有

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

如果是首和尾都有：
```
var reg = /^\d[\s\S]*\d$/g;
var str = "123abc123";
```
```
> str.match(reg)
▶ ["123abc123"]
```

RegExp 对象属性（FF 是火狐浏览器，IE 的 ie 浏览器，从哪个版本开始兼容）

属性	描述	FF	IE
global	RegExp 对象是否具有标志 g。	1	4
ignoreCase	RegExp 对象是否具有标志 i。	1	4
lastIndex 最有用	一个整数，标示开始下一次匹配的字符位置。	1	4
multiline	RegExp 对象是否具有标志 m。	1	4
source	正则表达式的源文本。	1	4

```
> reg
/^\d[\s\S]*\d$/g
> reg.ignoreCase
false
> reg.global
true
> reg.multiline
false
> reg.source
"^\d[\s\S]*\d$"
```

RegExp 对象方法

方法	描述	FF	IE
compile	编译正则表达式。	1	4
exec 厉害的方法	检索字符串中指定的值。返回找到的值，并确定其位置。	1	4
test	检索字符串中指定的值。返回 true 或 false。	1	4

reg.exec( );是一个匹配的方法

例
```
var reg = /ab/g;
var str = "abababab";
```

```
> reg.exec(str)
< ▶ ["ab"]
> reg.exec(str)
< ▶ ["ab"]
> reg.exec(str)
< ▶ ["ab"]
> reg.exec(str)
< null
```

例
```
var reg = /ab/g;
var str = "abababab";

// 检验一个字符串首尾是否含有数字
//
// reg.exec();
console.log(reg.exec(str));
```

这是一个类数组 → `▶ ["ab", index: 0, input: "abababab"]`

例说明他在第一个 ab 后面接着匹配

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.exec(str));
console.log(reg.exec(str));
```

```
▶ ["ab", index: 0, input: "abababab"]
▶ ["ab", index: 2, input: "abababab"]
```

例

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
```

```
▶ ["ab", index: 0, input: "abababab"]
▶ ["ab", index: 2, input: "abababab"]
▶ ["ab", index: 4, input: "abababab"]
▶ ["ab", index: 6, input: "abababab"]
```

例

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
console.log(reg.exec(str));
```

```
▶ ["ab", index: 0, input: "abababab"]
▶ ["ab", index: 2, input: "abababab"]
▶ ["ab", index: 4, input: "abababab"]
▶ ["ab", index: 6, input: "abababab"]
null
▶ ["ab", index: 0, input: "abababab"]
```

相当于游标一直在变，一圈一圈的转

例：游标的属性是 lastIndex。lastIndex 是为了 exec 而存在的

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
console.log(reg.exec(str));
console.log(reg.lastIndex);
```

```
0
▶ ["ab", index: 0, input: "abababab"]
2
▶ ["ab", index: 2, input: "abababab"]
4
▶ ["ab", index: 4, input: "abababab"]
6
▶ ["ab", index: 6, input: "abababab"]
8
null
0
```

例如果把游标归 0

```
var reg = /ab/g;
var str = "abababab";
console.log(reg.lastIndex);
console.log(reg.exec(str));
```

```
0
▶ ["ab", index: 0, input: "abababab"]
> reg.lastIndex = 0;
< 0
> console.log(reg.exec(str))
▶ ["ab", index: 0, input: "abababab"]
< undefined
> reg.lastIndex
< 2
```

例如果不加 g，lastIndex 就不会动，如下

```
var reg = /ab/;
var str = "abababab";
console.log(reg.lastIndex);
console.log(reg.exec(str));
```

```
0
▶ ["ab", index: 0, input: "abababab"]
0
▶ ["ab", index: 0, input: "abababab"]
```

拓展知识

例 var str = “aaaa”； **//想要匹配四个一样 xxxx 的**

var reg = /(a)/g; **//这个括号还有一个子表达式的意思，正常来说括号写了没有用，也没影响，但是在特殊情况，当你把式子当特殊括起来以后，这个括号会记录里面匹配的内容，记录完以后利用\几可以反向引用出来，如下**

var reg = /(a)\1/g; **//这个\1 意思是反向引用第一个子表达式里面的内容，这里的意思是匹配 a 和后面同样的 a**

如果换成 var reg = /(\w)\1/g; **//意思就变成了\w 匹配出来的东西，后面要 copy 一个一样的**

例想要匹配四个一样的写法如下：

```
var str = "aaaa";
var reg = /(\w)\1\1\1/g;
```

```
> str.match(reg)
< ▶ ["aaaa"]
```

例

```
var str = "aaaabbbb";
var reg = /(\w)\1\1\1/g;
```

```
> str.match(reg)
< ▶ ["aaaa", "bbbb"]
```

例

```
var str = "aabb";
var reg = /(\w)\1\1\1/g;
```

```
> str.match(reg)
< null
```

例现在想匹配出 aabb，应该怎么写？/(\w)\1(\w)\2/里面\1 是反向引用第一个表达式里面的内容，(\w)\2 是反向引用第二个表达式里面的内容。

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/g;
```

```
> str.match(reg)
< ▶ ["aabb"]
```

例

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/g;

console.log(reg.exec(str));
```

```
▶ ["aabb", "a", "b", index: 0, input: "aabb"]
```

上面多出了第一个字表达式和第二个子表达式的内容，而且是正式的数据位，虽然是类数组，但是能当数组用。

例就算不加 g，也会这样匹配出来，只是不改变 index

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/;
```

```
▶ ["aabb", "a", "b", index: 0, input: "aabb"]
```

例 **match　找到一个或多个正则表达式的匹配。**

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/;

console.log(str.match(reg));
```

```
▶ ["aabb", "a", "b", index: 0, input: "aabb"]
```

例但是加 g 了以后，只把匹配了多少个给你

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/g;

console.log(str.match(reg));
```

```
▶ ["aabb"]
```

例 **search　检索与正则表达式相匹配的值。**

返回的不是-1 都匹配成功了，返回的是匹配的这个东西的位置。

例

```
var str = "aabb";
var reg = /(\w)\1(\w)\2/g;

console.log(str.search(reg));
```

```
0
```

例

```
var str = "edbaabbbbee";
var reg = /(\w)\1(\w)\2/g;

console.log(str.search(reg));
```

```
3
```

例加不加 g，没有区别

```
var str = "edbaabbbbee";
var reg = /(\w)\1(\w)\2/g;

console.log(str.search(reg));
console.log(str.search(reg));
```

```
3
3
```

例如果匹配不到就返回-1

```
var str = "abc";
var reg = /(\w)\1(\w)\2/g;

console.log(str.search(reg));
console.log(str.search(reg));
```

```
-1
-1
```

split　把字符串分割为字符串数组。

例：按双重复的拆分（加不加 g 返回都一样）。下面这种失败

```
var str = "qpoweiuaaroqpiwuerl;mzxvvcpvoiyoiquweeeknpoasiyudofiuqwer";
var reg = /(\w)\1/g;

console.log(str.split(reg));
```

```
▶ ["qpoweiu", "a", "roqpiwuerl;mzx", "v", "cpvoiyoiquw", "e", "eknpoasiyudofiuqwer"]
```

例

```
var str = "qpoweiuaaroq0piwuerl;mzxvvcp0voiyoiquweeeknpoasiyudofi0uqwer";
var reg = /(\d)/g;

console.log(str.split(reg));
```

```
▶ ["qpoweiuaaroq", "0", "piwuerl;mzxvvcp", "0", "voiyoiquweeeknpoasiyudofi", "0", "uqwer"]
```

例如果写()子表达式就会把子表达式的东西返回，如果不写就不能反向引用了

```
var str = "qpoweiuaaroq0piwuerl;mzxvvcp0voiyoiquweeeknpoasiyudofi0uqwer";
var reg = /\d/g;

console.log(str.split(reg));
```

```
▶ ["qpoweiuaaroq", "piwuerl;mzxvvcp", "voiyoiquweeeknpoasiyudofi", "uqwer"]
```

replace　替换与正则表达式匹配的子串。　非常实用

例

```
var str = "aa";
console.log(str.replace("a", "b"));
```

```
ba
```

上面是非正则表达式的缺点，想要返回 bb，但是返回的是 ba。没有返回全局的能力。

例

```
var reg = /a/;
var str = "aa";
console.log(str.replace(reg, "b"));
```

```
ba
```

例

```
var reg = /a/g;
var str = "aa";
console.log(str.replace(reg, "b"));
```

```
bb
```

replace 的精华是正则表达式

例：想把 aabb 匹配出来，再倒过来

答案方法一

```
var reg = /(\w)\1(\w)\2/g;
var str = "aabb";
console.log(str.replace(reg, "$2$2$1$1"));
```

```
bbaa
```

上面 str.replace(reg,"")替换的信息不管写什么都要写成字符串，字符串里面“$1”代表第一个子表达式的内容，“$2”代表第二个子表达式的内容

方法二，写成 function，而 function（ ， ，）{}里面传的第一个参数是正则表达式匹配的全局（结果）（可随便起名），第二个参数是第一个表达式匹配的内容，第三个参数是第二个表达式匹配的内容（写 return$1$2 不适合语法）

```
var reg = /(\w)\1(\w)\2/g;
var str = "aabb";
console.log(str.replace(reg, function ($, $1, $2) {

    return $2 + $2 + $1 + $1;
}));
```

```
bbaa
```

例
```
var reg = /(\w)\1(\w)\2/g;
var str = "aabb";
console.log(str.replace(reg, function ($, $1, $2) {

    return $2 + $2 + $1 + $1 + $1 + 'abc';
}));
```
bbaaaabc

提示
```
> str
"aabb"
> str.toUpperCase()
"AABB"
> str.toUpperCase().toLowerCase()
"aabb"
```
变大写
变小写

例把 the-first-name 变成小驼峰式写法 theFirstName

思路：是把-f 变成 F，把-n 变成 N，先匹配再转换
```
var reg = /-(\w)/g;
var str = "the-first-name";
console.log(str.replace(reg, "_"));
```
the_irst_ame

答案
```
var reg = /-(\w)/g;
var str = "the-first-name";
console.log(str.replace(reg, function ($, $1) {
    return $1.toUpperCase();
}));
```
theFirstName

上面的 function 匹配了多少次 就有多少次 function 的执行 ,而上面的 reg 找了两次，找一次匹配一次

正向预查/正向断言

?=n　匹配任何其后紧接指定字符串 n 的字符串。

例我要选一个 a，后面跟着 b，就是正向预查/正向断言
```
var str = "abaaaaa"
var reg = /a(?=b)/g;
```
```
> str.match(reg)
▶ ["a"]
```
/a(?=b)/意思是 a 后面跟着 b，但是 b 不参与选择

?!n　匹配任何其后没有紧接指定字符串 n 的字符串。后面不跟着 n

例后面不跟着 b 的 a
```
var str = "abaaaaa"
var reg = /a(?!b)/g;
```
```
> str.match(reg)
▶ ["a", "a", "a", "a", "a"]
```

下面是正则表达式讲义内容：

贪婪匹配，变成非贪婪匹配（能少就不多），在量词后面加个?

例
```
var str = "aaaaaa";
var reg = /a+?/g;
```
```
> str.match(reg)
▶ ["a", "a", "a", "a", "a", "a"]
```

例 var reg = /a{1,3}?/g;　//这个 1 到 3，意思是有 1 就不取 2，3

思考：如果在量词是??

例 var reg = /a??/g;　//??第一个问号代表量词，第二个问号叫取消他的匹配，第一个问号是 0-1 的意思，加个问号就是能取 0 就不取 1

***?意思是能取 0 就不取多**

例
```
var str = "aaaaaa";
var reg = /a*?/g;
```
```
> str.match(reg)
▶ ["", "", "", "", "", "", ""]
```

注意：

1、想匹配空格，直接写 就可以了

2、想把选中的字符替换成$，直接写$是不行的，需要加上转义字符$

由于在替换文本里$有了特殊的含义（$代表反向引用），如果想要是替换$这个字符的话，需要写成$$，充当成转义字符

3、正则表达式要匹配一个反向引用的东西，写\\

例
```
var str = "aa\\aaaa";
var reg = /\\/g;
```
```
> str.match(reg)
▶ ["\"]
```

4、在正则表达式里面要匹配问号？写\？　其余*+-（ ）星号加号减号括号都类似

例
```
var str = "aa?aaaa";
var reg = /\?/g;
```
```
> str.match(reg)
▶ ["?"]
```

例下面是想去重，变成 abc，先匹配一串，再取一
```
var str = "aaaaaaaaaabbbbbbbbbbccccccc";
var reg = /(\w)\1*/g;
console.log(str.replace(reg, "$1"));
```
abc

例百度招聘 14 年最后一题：

给 var str =　“100000000000”；　这个数字科学计数法，每隔三位打个点

思路：找规律，从后往前查，每三位打个点，换个空进去【什么样的空？后面的数一定是 3 的倍数，(\d{3})+是 3 的倍数位个数字，这里面的东西一到多个，第一个/后面的空后面加上正向预查，后面是 3 的倍数位个数字，$以什么结尾】空的后面跟着 3 的倍数位个数字并且以这个结尾。\B 非单词边界

答案：
```
var str = "100000000000";

var reg = /(?=(\d{3})+$)/g;
console.log(str.replace(reg, "."));
```
.100.000.000.000

```
var str = "100000000000";

var reg = /(?=(\B)(\d{3})+$)/g;
console.log(str.replace(reg, "."));

// 100.000.000;
```
100.000.000.000