JavaScript笔记整理

JavaScript笔记整理

> *JS：轻量级的客户端脚本编程语言*

1 基础语法

1.1 变量

变量（variable）

> *它不是具体的值，只是一个用来存储具体值的容器或者代名词，因为它存储的值可以改变，所以称为变量*

1.1.1 创建变量的方式

- var (ES3)
- function (ES3) 创建函数(函数名也是变量，只不过存储的值是函数类型的而已)
- let (ES6)
- const (ES6) 创建的是常量
- import (ES6) 基于ES6的模块规范导出需要的信息
- class (ES6) 基于ES6创建类

1.1.2 变量的命名规范

1. 严格区分大小写
2. 遵循驼峰命名法：按照数字、字母、下划线或者$来命名（数字不能做为名字的开头），命名的时候基于英文单词拼接成一个完整的名字（第一个单词字母小写，其余每一个有意义单词的首字母都大写）
3. 不能使用关键字和保留字：在JS中有特殊含义的叫做关键词，未来可能会成为关键字的叫做保留字

1.1.3 创建变量并且赋值的详细操作步骤

> *例如：var n = 10;*
>
> - *变量提升阶段（js代码执行之前），第一步先把n这个变量声明 var n;（默认值undefined）*

- 开辟一个位置或者空间，把值存储起来
- 让变量和值关联在一起（基于等号完成赋值），我们把赋值关联这一步称之为“变量的定义”

> *注意点： 当我们的值还没有彻底准备完成之前，值和变量是没有关系的*

【基本类型操作机制 =>值类型】

> *值类型操作都是“按照值来操作的”*
>
> - *赋值的时候，也是直接的把这个值赋值给变量（或者说和变量关联）*
> - *一个变量把自己的值赋值给另外一个变量的时候，也是重新开辟一个新的位置，把原有变量存储的值放到新位置一份（新老位置各有相同的值，但是是独立分开的，没有关联），在把新位置上的值赋值给新变量*
> - *...*

【引用数据类型】

> *引用数据类型，操作的时候，都是按照“空间的引用地址来操作的”*

1. 首先不能像基本基本值一样在作用域中开辟位置存储，需要额外单独开辟一个新的空间（有一个16进制的地址，通过地址可以找到空间）
2. 对于对象数据类型来说，它会把自己本身的键值对依次存储到这个空间中（对于函数来说，在空间中存储的是函数体中的代码字符串）
3. 引用类型是按照引用地址来操作的，所以给变量赋的值是空间的地址，而不是对象本身，以后的操作都是通过地址找到空间然后再操作

【堆占内存】 在JS中有两个重要的内存：堆内存/栈内存

- 栈内存
    a. 提供代码执行的环境
    b. 基本类型值都是直接的存储在栈内存中的
- 堆内存
    a. 引用类型值都存储在堆内存中

1.2 数据类型

- 基本数据类型（值类型）
    a. 数字number
    b. 字符串string
    c. 布尔boolean
    d. null
    e. undefined
- 引用数据类型
    a. 对象object
        i. 普通对象
        i. 数组对象
        ii. 正则对象
        iii. 日期对象
        iv. Symbol
        v. ...
    b. 函数function

1.3 输出

1.3.1 alert

> *基于alert输出的结果都会转换为字符串：把值(如果是表达式先计算出结果)通过toString这个方法转换为字符串，然后再输出*

```
var myname='BigSpinach';
alert(myname); //=>window.alert

//基于alert输出的结果都会转换为字符串：把值(如果是表达式先计算出结果)通过toString这个方法转换为字符串，然后再输出
alert(1+1); =>'2'
alert(true); =>'true'
alert([12,23]); =>'12,23'
alert({name:'xxx'}); =>'[object Object]'
```

1.3.2 confirm

```
var flag = confirm('确定要退出吗?');
if(flag){
  //=>flag:true 用户点击的是确定按钮
}else{
  //=>flag:false 用户点击的是取消按钮
}
```

1.3.3 prompt

> *在confirm的基础上增加输入框*

```
var inputValue = prompt('你好吗？','输入框中的默认值');
if(inputValue=='好'){
   document.write('好好好好好好');
}else{
   document.write('坏坏坏坏坏坏坏坏坏');
}
```

1.3.4 console

```
window.console
console {debug: ƒ, error: ƒ, info: ƒ, log: ƒ, warn: ƒ, ...}
  assert: ƒ assert()
  clear: ƒ clear()
  context: ƒ context()
  count: ƒ count()
  countReset: ƒ countReset()
  debug: ƒ debug()
  dir: ƒ dir()
  dirxml: ƒ dirxml()
  error: ƒ error()
  group: ƒ group()
  groupCollapsed: ƒ groupCollapsed()
  groupEnd: ƒ groupEnd()
  info: ƒ info()
  log: ƒ log()
  memory: (...)
  profile: ƒ profile()
  profileEnd: ƒ profileEnd()
  table: ƒ table()
  time: ƒ time()
  timeEnd: ƒ timeEnd()
  timeLog: ƒ timeLog()
  timeStamp: ƒ timeStamp()
  trace: ƒ trace()
  warn: ƒ warn()
  Symbol(Symbol.toStringTag): "Object"
  get memory: ƒ ()
  set memory: ƒ ()
  __proto__: Object
```

1.4 ECMAScript中的操作语句

> *`==` 相等比较,如果等号左右两边的类型不一样，首先会转换为一样的数据类型，然后再进行比较 `===` 绝对比较，如果两边的数据类型不一样，则直接不相等，它要求类型和值都完全一样才会相等 `!val` 先把其它数据类型转换为布尔类型，然后取反 `!!val` 取两次反，等于没取反，也就剩下转换为布尔类型了*

1.4.1 if / else if / else

只要有一个条件成立，后面不管是否还有成立的条件，都不再判断执行了

```
var num = 10;
if(num>5){
   num+=2;
}else if(num>8){
   num+=3;
}else{
   num+=4;
}
console.log(num); //=>12
```

1.4.2 三元运算符

> *语法：条件?成立做的事情:不成立做的事情; <=>相当于简单的**if/else**判断*

```
var num=12;
if(num>10){
   num++;
}else{
   num--;
}
//=>改写成三元运算符
num>10?num++:num--;
```

特殊情况

```
//=>如果三元运算符中的某一部分不需要做任何的处理，我们用
null/undeifned/void 0... 占位即可
var num = 12;
num>10?num++:null;

//=>如果需要执行多项操作，我们把其用小括号包裹起来，每条操作语句用逗号分
隔
num=10;
num>=10?(num++,num*=10):null;
```

1.4.3 switch case

> *switch case中每一种case情况的比较都是基于"==="绝对相等来完成的*

```
switch(num){
    case 10:
        num++;
        break;
    case 5:
        num--;
        break;
    default:
        num=0;
}
```

利用 不加BREAK，后面的条件不管是否成立，都会被执行 的机制

```
var num = 9;
switch (num) {
    case 10:
    case 5:
        num--;
        break;
    default:
        num = 0;
}
console.log(num);
//=>不加BREAK，后面的条件不管是否成立，都会被执行；利用此机制，我们可以完成一些特殊的处理，例如：如果num等于10和等于5都要做同一件事情，那么我们写在一起，不用加break即可
```

1.5 循环

1.5.1 for 循环

> *作用：按照一定的规律，重复去做某件事情，此时我们就需要使用循环来处理了for 循环遍历数组只能遍历得到数组的私有属性*

```
/*
 * 在FOR循环的循环体中，经常出现两个常用的关键字：
 *   1. continue：继续
 *   2. break：中断或者结束
 */

for (var i = 0; i < 10; i++) {
    if (i < 5) {
        i++;
        continue;//=>结束本轮循环（循环体中continue后面代码将不再执行），继续执行下一轮循环
    }
    if (i > 7) {
        i += 2;
        break;//=>强制结束整个循环，不做任何的处理
    }
    i += 1;
}
```

1.5.2 for...in

> *for...in 语句以原始插入顺序迭代对象的可枚举属性。 对于对象 `key` 值就是对象中的每一个 `属性名` 对于 数组 `key` 值就是 数组中的每一个 `索引`*

```
let arr=[1,2,3];
for(let key in arr){
   console.log(key);//0 1 2
   console.log(arr[key]);//1 2 3
}
```

提示： **for...in不应该用于迭代一个 Array，其中索引顺序很重要。**

```
Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];
iterable.foo = 'hello';

console.log(iterable);
//(3) [3, 5, 7, foo: "hello"]

for (let i in iterable) {
  console.log(i); // logs 0, 1, 2, "foo", "arrCustom", "objCustom"
}
/*
   此循环仅以原始插入顺序记录iterable 对象的可枚举属性。它不记录数组元素3, 5, 7 或hello，因为这些不是枚举属性。但是它记录了数组索引以及arrCustom和objCustom
*/



for (let i in iterable) {
  if (iterable. hasOwnProperty(i)) {
    console.log(i); // logs 0, 1, 2, "foo"

  }
```

```
}

for (let i of iterable) {
  console.log(i); // logs 3, 5, 7
}
/*
   该循环迭代并记录iterable作为可迭代对象定义的迭代值，这些是数组元素 3, 5, 7，而不是任何对象的属性。
*/
```

仅迭代自身的属性

> *使用 `getOwnPropertyNames()` 或执行 `hasOwnProperty()` 来确定某属性是否是对象本身的属性（也能使用 `propertyIsEnumerable` ）。或者，如果你知道不会有任何外部代码干扰，您可以使用检查方法扩展内置原型。*

1.5.3 for...of

> *for...of 语句遍历**可迭代**对象定义要迭代的数据。 在每次迭代中，将不同**属性的值**分配给变量。 被迭代枚举其属性的对象。 `语法`*

```
for (variable of iterable) {
   //statements
}

//variable
//在每次迭代中，将不同属性的值分配给变量。
//iterable
//被迭代枚举其属性的对象。
```

```
let arr=[1,2,3];
for(let val of arr){
   console.log(val);//1 2 3
}
```

1.5.4 数组的forEach()

IE6~8不兼容

> *forEach：用来遍历数组中的每一项*
>
> 1. *数组中有几项，那我们传递进去的匿名回调函数就会执行几次*
> 2. *每一次执行匿名函数的时候，还给器传递了三个参数值 数组中的当前项item 当前项的索引index*
>
>   *原始的数组 input*
> 3. *理论上forEach是没有返回值的*

```
var ary = [1,2,,3,4,5,6];
var return_forEach= ary.forEach(function(item,index,input){
   console.log(arguments);
});
console.log(return_forEach);//undefined
```

```
//3. 理论上forEach是没有返回值的
var ary = [1,2,,3,4,5,6];
var return_forEach= ary.forEach(function(item,index,input){
   input[index] = item*10;
});
console.log(return_forEach);//undefined
console.log(ary);//[ 10, 20, <1 empty item>, 30, 40, 50, 60 ]
```

1.5.5 数组的map()

map 方法在 forEach() 的基础上，实现可以修改数组中 的值的功能

> *map()方法 返回值： return xxx 就是将当前遍历项修改为 xxx 参数：item index input 原数组不变*

```
//map()方法
//返回值： return xxx 就是将当前遍历项修改为 xxx
//参数：item index input
//原数组不变
{
   let arr = [2, 3, 45, 6, 8, 9];
   arr.map(function() {
      console.log(arguments);
   });

   /*
      { '0': 2, '1': 0, '2': [ 2, 3, 45, 6, 8, 9 ] }
      { '0': 3, '1': 1, '2': [ 2, 3, 45, 6, 8, 9 ] }
      { '0': 45, '1': 2, '2': [ 2, 3, 45, 6, 8, 9 ] }
      { '0': 6, '1': 3, '2': [ 2, 3, 45, 6, 8, 9 ] }
      { '0': 8, '1': 4, '2': [ 2, 3, 45, 6, 8, 9 ] }
      { '0': 9, '1': 5, '2': [ 2, 3, 45, 6, 8, 9 ] }
    */
```

```
}
console.log("----------------------");
{
    let arr = [2, 3, 45, 6, 8, 9];

    arr.map(function(item,index,input) {
        //console.log(item);//2 3 45 6  8 9
        //console.log(index);//0 1 2 3 4 5
        //console.log(input);//输出6次  [ 2, 3, 45, 6, 8, 9 ]
    });
}

console.log("----------------------");
{
    let arr = [2, 3, 45, 6, 8, 9];

    let return_map = arr.map(function(item,index,input) {
        return 123;
    });
    console.log(arr);//[ 123, 123, 123, 123, 123, 123 ]
    console.log(return_map);//[ 123, 123, 123, 123, 123, 123 ]

}
```

1.6 JS中数据类型转换

1.6.1 把其它数据类型转换为number类型

1.发生的情况

- isNaN检测的时候：当检测的值不是数字类型,浏览器会自己调用Number方法把它先转换为数字，然后再检测是否为非有效数字

```
isNaN('3') =>false
 Number('3')->3
 isNaN(3)->false

isNaN('3px') =>true
 Number('3px')->NaN
 isNaN(NaN)->true
```

- 基于parseInt/parseFloat/Number去手动转换为数字类型
- 数学运算：+ - * / %，但是“+”不仅仅是数学运算，还可能是字符串拼接

```
'3'-1 =>2
 Number('3')->3
 3-1->2

'3px'-1 =>NaN

'3px'+1 =>'3px1' 字符串拼接

var i='3';
i=i+1; =>'31'
i+=1; =>'31'
i++; =>4  i++就是单纯的数学运算，已经摒弃掉字符串拼接的规则
```

- 在基于“==”比较的时候，有时候也会把其它值转换为数字类型
- …

2.转换规律

```
//=>转换的方法：Number（浏览器自行转换都是基于这个方法完成的）

【把字符串转换为数字】

只要遇到一个非有效数字字符，结果就是NaN
```

```
'' ->0
' ' ->0 空格(Space)
'\n' ->0 换行符(Enter)
'\t' ->0 制表符(Tab)



【把布尔转换为数字】
true ->1
false ->0

【把没有转换为数字】
null ->0
undefined ->NaN

【把引用类型值转换为数字】
首先都先转换为字符串（toString），然后再转换为数字（Number）
```

1.6.2 把其它类型值转换为字符串

1.发生的情况

- 基于alert/confirm/prompt/document.write等方法输出内容的时候，会把输出的值转换为字符串，然后再输出

```
alert(1) =>'1'
```

- 基于“+”进行字符串拼接的时候
- 把引用类型值转换为数字的时候，首先会转换为字符串，然后再转换为数字
- 给对象设置属性名，如果不是字符串，首先转换为字符串，然后再当做属性存储到对象中（对象的属性只能是数字或者字符串）
- 手动调用toString/toFixed/join/String等方法的时候，也是为了转换为字符串

```
var n=Math.PI;//=>获取圆周率:
n.toFixed(2) =>'3.14'

var ary=[12,23,34];
ary.join('+') =>'12+23+34'
```

- ...

2.转换规律

```
//=>调用的方法：toString

【除了对象，都是你理解的转换结果】
1 ->'1'
NaN ->'NaN'
null ->'null'
[] ->''
[13] ->'13'
[12,23] ->'12,23'
...

【对象】
{name:'xxx'} ->'[object Object]'
{} ->'[object Object]'
不管是啥样的普通对象，最后结果都一样
```

1.6.3 把其它值转换为布尔类型

1.发生的情况

- 基于!/!!/Boolean等方法转换
- 条件判断中的条件最后都会转换为布尔类型
- ...

```
if(n){
  //=>把n的值转换为布尔验证条件真假
}

if('3px'+3){
  //=>先计算表达式的结果'3px3'，把结果转换为布尔true，条件成立
}
```

2.转换的规律

> *只有“0/NaN/''/null/undefined”五个值转换为布尔的false,其余都是转换为true*

特殊情况：数学运算和字符串拼接“+”

```
//=>当表达式中出现字符串，就是字符串拼接，否则就是数学运算

1+true =>2 数学运算
'1'+true =>'1true' 字符串拼接

[12]+10 =>'1210' 虽然现在没看见字符串，但是引用类型转换为数字，首先会转换为字符串，所以变为了字符串拼接
({})+10 =>"[object Object]10"
[]+10 =>"10"

{}+10 =>10 这个和以上说的没有半毛钱关系，因为它根本就不是数学运算，也不是字符串拼接，它是两部分代码
 {} 代表一个代码块（块级作用域）
 +10 才是我们的操作
 严格写法：{}; +10;
```

思考题：

```
12+true+false+null+undefined+[]+'bigspinach'+null+undefined+[]+true
=>'NaNbigspinachnullundefinedtrue'

12+true ->13
13+false ->13
13+null ->13
13+undefined ->NaN
NaN+[] ->'NaN'
'NaN'+'bigspinach' ->'NaNbigspinach'
...
'NaNbigspinachtrueundefined'
'NaNbigspinachtrueundefined'+[] ->'NaNbigspinachtrueundefined'
...
=>'NaNbigspinachtrueundefinedtrue'
```

1.6.4 特殊情况：“==”在进行比较的时候，如果左右两边的数据类型不一样，则先转换为相同的类型，再进行比较

对象==对象：不一定相等，因为对象操作的是引用地址，地址不相同则不相等

```
{name:'xxx'}=={name:'xxx'} =>false
[]==[] =>false

var obj1={};
var obj2=obj1;
obj1==obj2 =>true
```

> *对象 == 数字：把对象转换为数字 对象 == 布尔：把对象转换为数字，把布尔也转换为数字 对象 == 字符串：把对象转换为数字，把字符串也转换为数字 字符串 == 数字：字符串转换为数字 字符串 == 布尔：都转换为数字 布尔 == 数字：把布尔转换为数字*

不同情况的比较，都是把其它值转换为数字，然后再进行比较的

> *null==undefined：true null===undefined：false null&&undefined和其它值都不相等*
>
> *NaN==NaN：false*

```
1==true =>true
1==false =>false
2==true =>false 规律不要混淆，这里是把true变为数字1


[]==true：false 都转换为数字 0==1
![]==true：false

[]==false：true 都转换为数字 0==0
![]==false：true 先算![]，把数组转换为布尔取反=>false =>false==false
```

2 数据类型

2.1 Number 数字类型

```
ƒ Number()
  EPSILON:
  2.220446049250313
  e-16
  MAX_SAFE_INTEGER: 9007199254740991
  MAX_VALUE:
  1.7976931348623157
  e+308
  MIN_SAFE_INTEGER: -9007199254740991
  MIN_VALUE:
  5
  e-324

  NEGATIVE_INFINITY: -Infinity
```

NaN: NaN
POSITIVE_INFINITY: Infinity
arguments: (...)
caller: (...)
isFinite: ƒ isFinite()
isInteger: ƒ isInteger()
isNaN: ƒ isNaN()
isSafeInteger: ƒ isSafeInteger()
length: 1
name: "Number"
parseFloat: ƒ parseFloat()
parseInt: ƒ parseInt()
prototype: Number {0, constructor: ƒ, toExponential: ƒ, toFixed: ƒ, toPrecision: ƒ, toString: ƒ, ...}
__proto__: ƒ ()
[[Scopes]]: Scopes[0]

2.1.1 isNaN([value])

isNaN检测的机制

isNaN检测的机制
1、首先验证当前要检测的值是否为数字类型的，如果不是，浏览器会默认的把值转换为数字类型

把非数字类型的值转换为数字
- 其它基本类型转换为数字：直接使用Number这个方法转换的

[字符串转数字]
Number('13') ->13
Number('13px') ->NaN 如果当前字符串中出现任意一个非有效数字字符，结果则为NaN
Number('13.5') ->13.5 可以识别小数

[布尔转数字]
Number(true) ->1
Number(false) ->0

[其它]
Number(null) ->0
Number(undefined) ->NaN

- 把引用数据类型值转换为数字：先把引用值调取toString转换为字符串，然后再把字符串调取Number转换为数字

[对象]
({}).toString() ->'[object Object]' ->NaN

[数组]
[12,23].toString() ->'12,23' ->NaN
[12].toString() ->'12' ->12

[正则]
/^$/.toString() ->'/^$/' ->NaN

Number('') ->0
[].toString() ->''
=> isNaN([])：false

2、当前检测的值已经是数字类型，是有效数字返回false，不是返回true（数字类型中只有NaN不是有效数字，其余都是有效数字）

2.1.2 parseInt / parseFloat

> *等同于Number，也是为了把其它类型的值转换为数字类型*

> *和Number的区别在于字符串转换分析上*

> *Number：出现任意非有效数字字符，结果就是NaN*

> *parseInt：把一个字符串中的整数部分解析出来，parseFloat是把一个字符串中小数(浮点数)部分解析出来*

```
parseInt('13.5px') =>13
parseFloat('13.5px') =>13.5

parseInt('width:13.5px') =>NaN 从字符串最左边字符开始查找有效数字字符，并且转换为数字，但是一但遇到一个非有效数字字符，查找结束
```

2.2 Boolean

> *true / false 规律：* ***在JS中只有“0/NaN/空字符串/null/undefined”这五个值转换为布尔类型的false，其余都转换为true***

2.3 object 对象数据类型

> *一个对象中的属性名不仅仅是字符串格式的，还有可能是数字格式的 当我们存储的属性名不是字符串也不是数字的时候，浏览器会把这个值转换为字符串（toString），然后再进行存储*

2.4 function 函数数据类型

> *在JS中，函数就是一个方法（一个功能体），基于函数一般都是为了实现某个功能*

```
var total=10;
total+=10;
total=total/2;


total=total.toFixed(2);//=>保留小数点后边两位（数字由一个方法toFixed用来保留
```

```
小数点后面的位数)

....

在后续的代码中，我们依然想实现相同的操作（加10除以2），我们需要重新编写代码
var total=10;
total+=10;
total=total/2;
total=total.toFixed(2);

...

这样的方式会导致页面中存在大量冗余的代码，也降低了开发的效率，如果我们能把实现这个功能的代码进行“封装”，后期需要这个功能执行即可，这样就好了！
```

函数诞生的目的就是为了实现封装：把实现一个功能的代码封装到一个函数中，后期想要实现这个功能，只需要把函数执行即可，不必要再次编写重复的代码，起到了**低耦合高内聚（减少页面中的冗余代码，提高代码的重复使用率）**的作用

```
function fn(){
   var total=10;
   total+=10;
   total/=2;
   total=total.toFixed(2);
   console.log(total);
}
fn();
fn();
...
想用多少次，我们就执行多少次函数即可

=====

ES3标准中：
```

```
//=>创建函数
function 函数名([参数]){
   函数体：实现功能的JS代码
}
//=>函数执行
函数名();

=====
ES6标准中创建箭头函数：
let 函数名(变量名)=([参数])=>{
   函数体
}
函数名();

let fn=()=>{
   let total=10;
   ...
};
fn();
```

函数作为引用数据类型中的一种，它也是按照引用地址来操作的， **函数的运行机制**

```
function fn(){
   var total=10;
   total+=10;
   total=total.toFixed(2);
   console.log(total);
}
fn();

【创建函数】


1. 函数也是引用类型，首先会开辟一个新的堆内存，把函数体中的代码当做“字符
```

串”存储到内存中（对象向内存中存储的是键值对）
2. 把开辟的堆内存地址赋值给函数名(变量名)

此时我们输出fn（切记不是fn()）代表当前函数本身
如果我们执行fn()，这是把函数执行
所以是否加小括号是两种不同本质的操作

【函数执行】
目的：把之前存储到堆内存中的代码字符串变为真正的JS代码自上而下执行，从而实现应有的功能

1.函数执行，首先会形成一个私有的作用域（一个供代码执行的环境，也是一个栈内存）
2.把之前在堆内存中存储的字符串复制一份过来，变为真正的JS代码，在新开辟的作用域中自上而下执行

函数中的参数

> *参数是函数的入口：当我们在函数中封装一个功能，发现一些原材料不确定，需要执行函数的时候用户传递进来才可以，此时我们就基于参数的机制，提供出入口即可*

```
//=>此处的参数叫做形参：入口，形参是变量（n/m就是变量）
function sum(n,m){
   //=>n和m分别对应要求和的两个数字
   var total = 0;
   total = n + m;
   console.log(total);
}

//=>此处函数执行传递的值是实参：实参是具体的数据值
sum(10,20);  //=>n=10 m=20
sum(10); //=>n=10 m=undefined
sum(); //=>n和m都是undefined
sum(10,20,30); //=>n=10 m=20  30没有形参变量接收
```

函数的运作机制： 【创建函数】

1. 函数也是引用类型值，首先开辟一个新的堆内存（16进制地址）
2. 把函数体中实现功能的代码“当做字符串”存储到堆内存中
3. 把堆内存的引用地址赋值给函数名（变量名）

【执行函数】 目的：把函数体中的JS代码执行，以此实现具体的功能和需求

1. 首先开辟一个新的栈内存（私有作用域），提供JS代码执行赖以生存的环境
2. 把原有在堆内存中存储的“字符串”，拿过来，放到新的栈内存中，变为真正的JS代码，自上而下执行

====

封装一个函数实现一些功能，有时候发现实现功能的原材料不足，需要执行函数的时候传递给我们才可以，此时我们可以给函数设置几个入口，我们把函数的入口称之为：“形参变量”(入口是变量)

2.5 Math

Math
  E: 2.718281828459045
  LN2: 0.6931471805599453
  LN10: 2.302585092994046
  LOG2E: 1.4426950408889634
  LOG10E: 0.4342944819032518
  PI: 3.141592653589793
  SQRT1_2: 0.7071067811865476
  SQRT2: 1.4142135623730951
  abs: *f* abs()
  acos: *f* acos()
  acosh: *f* acosh()
  asin: *f* asin()
  asinh: *f* asinh()
  atan: *f* atan()
  atan2: *f* atan2()
  atanh: *f* atanh()
  cbrt: *f* cbrt()
  ceil: *f* ceil()
  clz32: *f* clz32()
  cos: *f* cos()
  cosh: *f* cosh()
  exp: *f* exp()
  expm1: *f* expm1()
  floor: *f* floor()
  fround: *f* fround()
  hypot: *f* hypot()
  imul: *f* imul()
  log: *f* log()
  log1p: *f* log1p()
  log2: *f* log2()
  log10: *f* log10()
  max: *f* max()
  min: *f* min()

  pow: *f* pow()

```
random: ƒ random()
round: ƒ round()
sign: ƒ sign()
sin: ƒ sin()
sinh: ƒ sinh()
sqrt: ƒ sqrt()
tan: ƒ tan()
tanh: ƒ tanh()
trunc: ƒ trunc()
Symbol(Symbol.toStringTag): "Math"
__proto__: Object
```

2.5.1 Math中提供的常用方法

abs：取绝对值

ceil/floor：向上或者向下取整

round：四舍五入

sqrt：开平方

pow：取幂（N的M次方）

max/min：获取最大值和最小值

PI：获取圆周率

random：获取0~1之间的随机小数

2.6 String

```
console.dir(new String("ss"))
```

```
String
  0: "s"
  1: "s"
  length: 2
  __proto__: String
  anchor: ƒ anchor()
  big: ƒ big()
  blink: ƒ blink()
  bold: ƒ bold()
  charAt: ƒ charAt()
  charCodeAt: ƒ charCodeAt()
  codePointAt: ƒ codePointAt()
  concat: ƒ concat()
  constructor: ƒ String()
  endsWith: ƒ endsWith()
  fixed: ƒ fixed()
  fontcolor: ƒ fontcolor()
  fontsize: ƒ fontsize()
  includes: ƒ includes()
  indexOf: ƒ indexOf()
  italics: ƒ italics()
  lastIndexOf: ƒ lastIndexOf()
  length: 0
  link: ƒ link()
  localeCompare: ƒ localeCompare()
  match: ƒ match()
  normalize: ƒ normalize()
  padEnd: ƒ padEnd()
  padStart: ƒ padStart()
  repeat: ƒ repeat()
  replace: ƒ replace()
  search: ƒ search()
  slice: ƒ slice()
  small: ƒ small()

  split: ƒ split()
```

```
startsWith: ƒ startsWith()
strike: ƒ strike()
sub: ƒ sub()
substr: ƒ substr()
substring: ƒ substring()
sup: ƒ sup()
toLocaleLowerCase: ƒ toLocaleLowerCase()
toLocaleUpperCase: ƒ toLocaleUpperCase()
toLowerCase: ƒ toLowerCase()
toString: ƒ toString()
toUpperCase: ƒ toUpperCase()
trim: ƒ trim()
trimEnd: ƒ trimEnd()
trimLeft: ƒ trimStart()
trimRight: ƒ trimEnd()
trimStart: ƒ trimStart()
valueOf: ƒ valueOf()
Symbol(Symbol.iterator): ƒ [Symbol.iterator]()
__proto__: Object
[[PrimitiveValue]]: ""
[[PrimitiveValue]]: "ss"
```

2.6.1 字符串的常用方法和属性

> *在JS中所有用单引号或者双引号包起来的都是字符串，每一个字符串是由零到多个字符组成 字符串中的每一个字符都有一个自己对应位置的索引，也有类似于数组一样的length代表自己的长度*

```
let str= 'BigSpinach';
console.log(str.length);//10
console.log(str[str.length-1]);//"h"
console.log(str[100]);//undefined
```

> *字符串是基本数据类型，字符串的每一次操作都是值直接的进行操作，不像数组一样是基于空间地址来操作的，所以不存在原有字符串是否改变这一说，肯定都是不变的*

2.6.1.1 charAt/charCodeAt 和 String.fromCharCode()

> *作用：charAt根据索引获取指定位置的字符，charCodeAt不仅仅获取字符，它获取的是字符对应的Unicode编码值(ASCII码值) 参数：索引 返回：字符或者对应的编码*

```
str.charAt(0);//"B"
str[0]//"B"

str.charAt(100);//""
str[100]//ubdefined

//"B"的Unicode值是66
charCodeAt(0);//66
//十进制的Unicode值
String.fromCharCode(66);//"B"
```

2.6.1.2 indexOf/lastIndexOf

> *获取字符在字符串中第一次或者最后一次出现位置的索引，有这个字符，返回大于等于零的索引，不包含这个字符，返回的结果是-1， 所以可以基于这两个方法，验证当前字符串中是否包含某个字符*

```
let str= 'BigSpinach';
if(str.indexOf('@')>-1){
   //=>条件成立说明包含@符号
}
```

2.6.1.3 slice 、 substring 、 substr

> *作用：截取字符串 substr(index,number) 从索引 index 开始，截取 number 个*

```
//str.substr(n,m)
//从索引n开始，截取m个
//返回 截取到的字符串
//原字符串不变
let str= 'BigSpinach';
let return_substr = str.substr(0,6);
console.log(return_substr);//BigSpi
```

substring(indexStart,indexEnd)

> *从索引 indexStart 开始，截取到索引 indexEnd 位置，不包含 indexEnd*

```
//str.substring(n,m)
//从索引n开始，截取到索引m处（不包含索引m）
//返回 截取到的字符串
//原字符串不变
let str= 'BigSpinach';
let return_substring = str.substring(0,6);
console.log(return_substr);//BigSpi
```

str.slice(n,m)

```
//str.slice(n,m)
//从索引n开始，截取到索引m处（不包含索引m）
//返回 截取到的字符串
//原字符串不变
let str= 'BigSpinach';

let return_slice = str.substr(0,5);
```

```
console.log(return_slice);//BigSp
console.log(str);//BigSpinach
//slice和substring的区别
//slice支持负数索引
let return_slice_minus= str.slice(-3,-1);//ach
console.log(str.slice(7,9));//ac
//本质
//str.length-3到str.length-1位置
//    n          m
console.log(return_slice_minus);//ac
console.log(str);//BigSpinach
```

传参问题

1. 传递两个参数，并且第二个参数大于 str.length , 截取到字符串末尾
2. 只传递1个参数n,截取的结果是从 n到字符串末尾
3. 一个参数都没传，截取的是整个字符串（相当一克隆）

2.6.1.4 toUpperCase/toLowerCase

> *实现字母的大小写转换，toUpperCase小写转大写，toLowerCase大写转小写 返回新字符串 原字符串不发生改变*

```
let str= 'BigSpinach';
let return_toUpperCase = str.toUpperCase();
let return_toLowerCase = str.toLowerCase();
console.log(str,return_toUpperCase,return_toLowerCase);
//BigSpinach BIGSPINACH bigspinach
```

2.6.1.5 split 支持正则

`stringObject.split(separator,howmany)`

> ***separator*** *必需。字符串或正则表达式，从该参数指定的地方分割 stringObject。*
>
> ***howmany*** *可选。该参数可指定返回的数组的最大长度。如果设置了该参数，返回的子串不会多于这个参数指定的数组。如果没有设置该参数，整个字符串都会被分割，不考虑它的长度。*

> *和数组中的join相对应，数组中的join是把数组们一项按照指定的连接符变为字符串，而split是把字符串按照指定的分隔符，拆分成数组中每一项*

```
let str= 'Big-Spi-nach';
let return_split = str.split("-");
console.log(str,return_split);
//Big-Spi-nach [ 'Big', 'Spi', 'nach' ]
```

支持正则

```
let str= 'Big-Spi-nach';
let reg = /-/g;
let return_split_with_reg = str.split(reg,'');
//console.log(str,return_split_with_reg);
//Big-Spi-nach []

let return_split_with_reg_2 = str.split(reg,'');
console.log(str,return_split_with_reg_2);
//Big-Spi-nach [ 'Big', 'Spi', 'nach' ]
```

2.6.1.6 replace 支持正则

惰性，每次只替换一处 作用：替换字符串中的原有字符 参数：原有字符，要替换的新字符 返回：替换后的字符串 原字符串不发生改变

```
let str= 'Big-Spi-nach';
let return_replace= str.replace('-','哈哈');
console.log(str,return_replace);
//Big-Spi-nach Big哈哈Spi-nach
```

```
//使用正则，解决replace的惰性
let reg =/-/g;
let str= 'Big-Spi-nach';
let return_replace= str.replace('-','哈哈');
console.log(str,return_replace);
//Big-Spi-nach Big哈哈Spi哈哈nach
```

2.6.1.7 trim 、 trimLeft 、 trimRight

兼容性不太好

> *去除 首尾 的空格 返回新字符串 原字符串不发生改变*

```
let str= '      Big-    Spi    -nach       ';
let return_trim = str.trim();
console.log(return_trim);
//Big-    Spi    -nac
```

2.6.2 字符串中的常用案例

2.6.2.1 queryURLParameter

```
let url ='https://www.baidu.com/s?ie=utf-8&f=8&rsv_bp=1&ch=7';
   //使用字符串的方法
   //先从“？”处切割
   let index = url.indexOf("?");
   let newStr = url.slice(index+1);
   let obj={};
   //console.log(newStr);
   //ie=utf-8&f=8&rsv_bp=1&ch=7

   //以“&”切成数组
   let ary = newStr.split('&');
   //console.log(ary);
   //[ 'ie=utf-8','f=8',...]

   //遍历ary
   for(let i=0;i<ary.length;i++){
      //以“=”切
      let return_split = ary[i].split("=");
      obj[return_split[0]] = return_split[1];
   }

   //console.log(obj);
   /*
   {
      ie: 'utf-8',
      f: '8',
      ...
   }
    */

//封装
function queryURLParameter (url) {
      let obj={};
      let index = url.indexOf("?");

      if(index===-1){
```

```
        return obj;
    }
    let ary = newStr.split('&');
    for(let i=0;i<ary.length;i++){
        let return_split = ary[i].split("=");
        obj[return_split[0]] = return_split[1];
    }
    return obj;
}

//方案2：使用正则
String.prototype.myQueryURLParameter=function myQueryURLParameter() {//使用正则
    //(属性名)=(属性值)
    let obj={};
    let reg=/([^=?&]+)=([^=?&]+)/g;
    this.replace(reg,function(){
        //console.log(arguments);
        /*
            { '0': 'ie=utf-8',
              '1': 'ie',
              '2': 'utf-8',
              '3': 24,
              '4': 'https://www.baidu.com/s?ie=utf-8&f=8&rsv_bp=1&ch=7&tn=98012088_9_dg&wd=%E5%88%98%E5%87%AF&oq=bad' }
         */
        //let arg = arguments;
        //console.log(arg[1]);
        //console.log(arg[2]);
        //var attr = arguments[1];
        //var val= arguments[2];
        //console.log(arguments[1]);
        //console.log(arguments[2]);

        //obj[attr]= val;
```

```
        obj[arguments[1]] = arguments[2];
    });
    return obj;
}


String.prototype.myQueryURLParameter=function myQueryURLParameter() {
    let obj={};
    let reg=/([^=?&]+)=([^=?&]+)/g;
    this.replace(reg,function(){
        obj[arguments[1]] = arguments[2];
    });
    return obj;
}

//封装
~function (pro) {
    pro.queryURLParameter = function () {
        var obj = {},
            reg = /([^?=&#]+)(?:=([^?=&#]+)?)/g;
        this.replace(reg, function () {
            var key = arguments[1],
                value = arguments[2] || null;
            obj[key] = value;
        });
        return obj;
    }
}(String.prototype);
```

2.6.2.2 queryCode

```
function quertCode(n,str) {
    n=n||4;
```

```
str=str||'0123456789qwertyuiopasdfghjklzxcvbnmQAZXSWEDCVFRTGBNHYUJMKIOLP';
      let result='';
      for(let i=0;i<n;i++){
        //生成随机索引
        //获取随机索引的值
        //将随机索引的值拼接到 result变量上
        let index = Math.floor(Math.random()*(((str.length-1)+0)-0));
        let val = str.charAt(index);
        result+=val;

      }
      return result;
   }
```

2.6.2.3 时间字符串格式化

> *有一个时间字符串“2018-11-2 0:5:19”，我们想基于这个字符串获取到“2018年08月08日 23时24分24秒”*

```
let str_data='2018-11-2 0:5:19' ;
   let str_data2='2018-11 5:19' ;

   //目的：-> 2018年11月02日 00点05分19秒

   //思路一：
   //按照空格拆分成一个数组arr
   //arr[1] = '2018-11-2'
   //arr[2] = '0:5:19'
   //对这两项再进行拆分
   //arr[1]按照“-”拆分
   //   第一项拼接 年

   //   第二项拼接 月
```

```
  //   第三项拼接 日
  //   resault1
  //arr[2]按照“：”拆分
  //   第一项拼接 时
  //   第二项拼接 分
  //   第三项拼接 秒
  //    resault2
  //最后拼接result=result1+result2
  function formattingTime (str) {
    let result ='';
    let arr = str.split(" ");
    let arr1 = arr[0].split("-");
    let arr2 = arr[1].split(":");
    //["2018", "11", "2"] (3) ["0", "5", "19"]
    //这样写的好处是，下次想要换这里的关键字的时候可以直接更换这里的数组即可
    let str_date = ["年","月","日"];
    let str_time = ["时","分","秒"];
    for(let i=0;i<arr1.length;i++){
      let cur = arr1[i];
      if(cur<10){
        //先处理补0
        cur+="";
        let resault_pad0 = cur.padStart(2,"0");
        result +=(resault_pad0+str_date[i]);
      }else{
        result += (cur+str_date[i]);
      }
    }

    for(let i=0;i<arr2.length;i++){
      let cur2 = arr2[i];
      if(cur2<10){
        //先处理补0

        cur2+="";
```

```
            let resault_pad0 = cur2.padStart(2,"0");
            result +=(resault_pad0+str_time[i]);
        }else{
            result += (cur2+str_time[i]);
        }
    }
    return result;
}
console.log(formattingTime(str_data2));
```

```
~function (pro) {
    pro.formatTime = function (template) {
        template = template || '{0}年{1}月{2}日 {3}时{4}分{5}秒';
        var ary = this.match(/\d+/g);
        template = template.replace(/\{(\d+)\}/g, function () {
            var n = arguments[1],
                val = ary[n] || '0';
            val < 10 ? val = '0' + val : null;
            return val;
        });
        return template;
    }
}(String.prototype);
```

2.7 Array

```
[]
    length: 0
    __proto__: Array(0)
    concat: ƒ concat()
    constructor: ƒ Array()
    copyWithin: ƒ copyWithin()

    entries: ƒ entries()
```

```
    every: ƒ every()
    fill: ƒ fill()
    filter: ƒ filter()
    find: ƒ find()
    findIndex: ƒ findIndex()
    flat: ƒ flat()
    flatMap: ƒ flatMap()
    forEach: ƒ forEach()
    includes: ƒ includes()
    indexOf: ƒ indexOf()
    join: ƒ join()
    keys: ƒ keys()
    lastIndexOf: ƒ lastIndexOf()
    length: 0
    map: ƒ map()
    pop: ƒ pop()
    push: ƒ push()
    reduce: ƒ reduce()
    reduceRight: ƒ reduceRight()
    reverse: ƒ reverse()
    shift: ƒ shift()
    slice: ƒ slice()
    some: ƒ some()
    sort: ƒ sort()
    splice: ƒ splice()
    toLocaleString: ƒ toLocaleString()
    toString: ƒ toString()
    unshift: ƒ unshift()
    values: ƒ values()
    Symbol(Symbol.iterator): ƒ values()
    Symbol(Symbol.unscopables): {copyWithin: true, entries: true, fill: true, find:
true, findIndex: true, …}
    __proto__: Object
```

> *数组也是对象数据类型的，也是由键值对组成的 数组中每一项的值可以是任何数据类型的*

创建数组的方式

```
//直接量的方式
var arr = [0,1,2,3,4,5,6,7,8];
//构造函数的方式
var arr=new Array(1,2,3,4);
```

2.7.1 数组中的常用方法

2.7.1.1 增元素添加

push 作用：向数组“末尾”追加新的内容 参数：追加的内容（可以是一个，也可是多个） 返回值：新增后数组的长度 原有数组改变

```
let arr = [1,2,{'sname':"刘凯",'sage':25},["a","b","c"],"xxx"];
let return_push = arr.push(250);
console.log(arr);
console.log(return_push);//6
//[ 1, 2, { sname: '刘凯', sage: 25 }, [ 'a', 'b', 'c' ], 'xxx', 250 ]
```

unshift 作用：向数组开始位置追加新内容 参数：要新增的内容 返回：新增后数组的长度 原有数组改变

```
let arr = [1,2,{'sname':"刘凯",'sage':25},["a","b","c"],"xxx"];
let return_unshift = arr.unshift(250);
console.log(return_unshift);//6
console.log(arr);
//[ 250, 1, 2, { sname: '刘凯', sage: 25 }, [ 'a', 'b', 'c' ], 'xxx' ]
```

arr[arr.length]=x

```
let arr = [1,2,{'sname':"刘凯",'sage':25},["a","b","c"],"xxx"];
//末尾增
arr[arr.length] = 250;
console.log(arr);
//[ 1, 2, { sname: '刘凯', sage: 25 }, [ 'a', 'b', 'c' ], 'xxx', 250 ]
```

concat()

```
var arr = [0,1,2,3,4,5,6,7,8];
var arr2 = arr.concat([1,2,3], [22,33,44]);
console.log(arr2);  //0,1,2,3,4,5,6,7,8,1,2,3,22,33,44
```

2.7.1.2 元素的删除

pop 作用：删除数组最后一项 参数：无 返回：被删除的那一项内容 原有数组改变

```
let arr = [1,2,{'sname':"刘凯",'sage':25},["a","b","c"],"xxx"];
let return_pop = arr.pop();
console.log(return_pop);//xxx
console.log(arr);
//[ 1, 2, { sname: '刘凯', sage: 25 }, [ 'a', 'b', 'c' ] ]
}
```

shift 作用：删除数组中的第一项 参数：无 返回：被删除的那一项内容 原有数组改变

```
let arr = [1,2,{'sname':"刘凯",'sage':25},["a","b","c"],"xxx"];
let return_shift = arr.shift();
console.log(return_shift);//1
console.log(arr);
//[  2, { sname: '刘凯', sage: 25 }, [ 'a', 'b', 'c' ] ,xxx]
```

2.7.1.3 splice的增删改

splice(n,m,x,...)

> *基于 splice 可以对数组进行很多的操作：删除指定位置的内容、向数组指定位置增加内容、还可以修改指定位置的信息*

删除：ary.splice(n,m) 从索引n开始，删除m个内容，把删除的部分以一个新数组返回，原有数组改变

新增：ary.splice(n,0,x,...) 从索引n开始删除零项（没删除），把X或者更多需要插入的内容存放到数组中索引N的“前面”

修改：ary.splice(n,m,x,...) 修改的原理就是把原有内容删除掉，然后用新的内容替换这部分信息即可

```
//=>删除最后一项
ary.pop()
ary.splice(ary.length-1)
ary.length--

//=>向数组末尾追加新内容
ary.push(100)
ary.splice(ary.length,0,100)
ary[ary.length]=100
```

2.7.1.4 查

slice 作用：在一个数组中，按照条件查找出其中的部分内容 参数：两个参数（n/m），从索引n开始，找到索引m处，但是不包含m 返回：以一个新数组存储查找的内容 原有数组不会变

2.7.1.5 排序

reverse 作用：把数组倒过来排列 参数：无 返回：排列后的新数组 原有数组改变

sort 作用：给数组排序 参数：无/函数 返回：排序后的新数组 原有数组改变

```
//=>sort在不传递参数的情况下，只能处理10以内数字排序
var ary=[1,3,2,4,5,6,7,9,8];
ary.sort(); =>[1,2,3,4,5,6,7,8,9]

var ary=[18,1,23,27,2,35,3,56];
ary.sort(); =>[1, 18, 2, 23, 27, 3, 35, 56] 没有按照我们想象中的排序

//=>真实项目中，基于sort排序，我们都需要传递参数
var ary=[18,1,23,27,2,35,3,56];
ary.sort(function (a,b){
   return a-b;//=>升序  return b-a; 降序
```

```
});

//带单位增序降序的方法:
var aNum = ['33px','22px','67px','3px','543px'];
aNum.sort(function(a, b){
   return parseInt(a) - parseInt(b);
});
console.log(aNum);
```

2.7.1.6 数组转换

toLocaleString 作用：使用地区特定的分隔符来将生成的字符串连接起来 参数：无 返回：数组中的每一项用本地分隔符分隔的字符串 原有数组不变

```
   let arr =[1,3,4,56,999];
   let return_toLocaleString = arr.toLocaleString();
   console.log(arr,return_toLocaleString );
   //[ 1, 3, 4, 56, 999 ] '1,3,4,56,999'

````

`toString`
作用：把数组转换为字符串
参数：无
返回：数组中的每一项用逗号分隔的字符串
原有数组不变

```javascript
   let arr =[1,3,4,56,999];
   let return_toString = arr.toString();
   console.log(arr,return_toString);
   //[ 1, 3, 4, 56, 999 ] '1,3,4,56,999'
```

valueOf() 该方法可以返回Array对象的原数值，通常都是在后台隐式的调用该方法，一般不会显式的出现我们的代码中。 **valueOf() 方法定义在 Object 的原型上**

```
let arr =[1,3,4,56,999];
let return_valueOf = arr.valueOf();
console.log(arr,return_valueOf);
//(5) [1, 3, 4, 56, 999]0: 11: 32: 43: 564: 999length: 5__proto__: Array(0) (5) [1, 3, 4, 56, 999]

let str = "门前大桥下";
let return_valueOf_str = str.valueOf();
console.log(str,return_valueOf_str);

let num = 250250;
console.log(num.valueOf());//250250

let kong = null;
//console.log(kong.valueOf());//报错

let und = undefined;
//console.log(und.valueOf());////报错

let flag = true;
console.log(flag.valueOf());

let obj ={};
console.log(obj.valueOf());//{}
```

join 作用：和toString类似，也是把数组转换为字符串，但是我们可以设置变为字符串后，每一项之间的连接符 参数：指定的链接符 返回：字符串 原有数组不变

```
let arr = [12,3,4,5,6,7,8];
let return_join = arr.join("+");
//[ 12, 3, 4, 5, 6, 7, 8 ] '12+3+4+5+6+7+8'
console.log(arr,return_join);
```

2.7.1.7 位置方法

indexOf / lastIndexOf 这两个方法不兼容IE低版本浏览器(IE6~8) 作用：检测当前值在数组中第一次或者最后一次出现位置的索引 参数：要检测的值 返回：索引(没找到返回-1) 原有数组不变

> *indexOf() 和 lastIndexOf() 方法的第一个参数都是必需的，传入的是需要搜索的目标值，而第二个参数是可选的，即指定开始搜索的位置，如果不传的话， indexOf() 方法默认从头开始搜索， lastIndexOf 方法默认从尾开始搜索*
>
> *重要的是，第二个参数可以是一个负值，表示相对数组末尾的偏移量，所以这也使得以上两方法的使用没有特别明确的界限。*

```
//=>验证数组中是否包含某一项
if(ary.indexOf(100)>-1){
  //=>ARY中包含100这一项
}
```

2.7.1.6 Array.prototype 的迭代方法

- every
- filter
- find
- forEach
- includes

- keys
- map
- reduce / reduceRight
- some
- ...

forEach的实现原理

```
Array.prototype.forEach=function(fun,context){
   // forEach的实现原理
   var len=this.length;
   var context=arguments[1];//即使为undefined，call函数也正常运行。
   if(typeof fun !=="function"){
      throw "输入正确函数!";
   }
   for(var i=0;i<len;i++){
      fun.call(context,this[i],i,this);
//注意这四个参数，很关键，context是上下文，即作用对象，this[i]是各项元素的值，i为各项的索引值，this为执行该方法的主体
   }
};
arr.forEach(function(item,index,arr){
   console.log(item,index,arr);
   //item    arr的每一项的内容
   //index  各项的索引值
   //arr     即是输入的原数组值
});
```

every()

> *该方法对数组中的每一项都运行给定的函数直接量，如果该函数对每一项都返回true，则该方法返回true，注意是每一项都满足条件该方法才会返回true。*

```
var arr=[1,2,3,4,5,6,7];
var result=arr.every(function (item, index, array){ return item>5; });
alert(result);  //此时会返回的结果是 false;
var arr1=[3,4,5];
var result1=arr1.every(function (item, index, array){ return item>2; });
alert(result1); //此时会返回的结果是 true;
```

``filter()

> *该方法对数组中的每一项都运行给定函数，返回该函数返回true的项组成的数组，从单词字面意思理解，该方法是对数据执行一个过滤的作用，满足条件的返回，不满足条件的丢弃，仅此而已，用个例子说明：*

```
var arr=[1,2,3,4,5,6,7];
var result=arr.filter(function (item, index, array){ return item>5; });
alert(result);  //此时会返回的结果是 [6,7];
```

``forEach()

> *对数组中的每一项运行给定函数，需要注意的是，这个方法没有返回值。*

```

```

``map()

> *对数组中的每一项运行给定的函数，返回**每次函数调用的结果组成的*** `数组`

```

```

``some()

> *对数组中的每一项运行给定的函数，如果该函数对任一项都返回true，则该方法返回true，和every()方法有点相似，但是要区别啊亲，简单区别的话可以从单词下手，every()是传入的函数需要对每一项都需要满足条件，才会返回true；而some()方法是只要传入的函数对数组中的某一项返回true，该方法就会返回true。*

```

```

2.7.1.7 缩小方法

`educe()

> *该方法从数组的第一项开始逐个遍历至尾，使用指定的函数来将数组的元素进行整合，只生成单个的值，这就是缩小方法，很好理解吧。另外，需要对该方法的参数进行说明一下，reduce()需要两个参数，第一个参数是执行化简操作的函数，这个参数必需；第二个参数是一个传递给函数的初试值，这里需要理解一下，所谓初始值就是传给第一个函数参数执行操作的第一个值，在接下来的操作中，这个值就是上一次函数的返回值了，而当第二个传参不使用时，化简函数就使用数组的第一个元素和第二个元素作为其第一个和第二个参数进行计算。请结合以下代码理解：*

```
var arr=[1,2,3,4,4,5,6,6];
var sum=arr.reduce(function (x, y){ return x+y; }, 0);  //数组元素求和
var multi=arr.reduce(function (x, y){ return x*y}, 1);  //求出数组中各元素的积
var max=arr.reduce(function (x, y){ return (x>y)?x:y}); //求出最大值为6
```

reduceRight()

> *该方法的使用和reduce()是一样的，这里可以联想到lndexOf()方法和lastIndexOf()方法的关系，即reduceRight()方法是按照数组索引从高到低的处理数组。*

2.7.2 数组中的常用案例

2.7.2.1 数组去重

方案一: 基于双重for循环

> *1.依次拿出数组中的每一项（排除最后一项：最后一项后面没有需要比较的内容） 2.和当前拿出项后面的每一项依次比较 3.如果发现有重复的，我们把找到的这个重复项在原有数组中删除掉（splice）*

```
var ary = [1, 2, 3, 2, 2, 3, 4, 3, 4, 5];
//=>i<ary.length-1：不用拿最后一项

for (var i = 0; i < ary.length - 1; i++) {
    var item = ary[i];
    for (var k = i + 1; k < ary.length; k++) {
        //ary[k]：后面需要拿出来和当前项比较的这个值
        if (item === ary[k]) {
            //=>相等：重复了,我们拿出来的K这个比较项在原有数组中删除
            // ary.splice(k, 1);
            /*
             这样做会导致数组塌陷问题：当我们把当前项删除后，后面每一项都要向前进一位，也就是原有数组的索引发生了改变，此时我们k继续累加1，下一次在拿出来的结果就会跳过一位
             */
            ary.splice(k, 1);//=>删除后不能让k累加了
            k--;//=>删除后先减减，在加加的时候相当于没加没减
        }
    }
}
```

```
//解决数组塌陷
for (var i = 0; i < ary.length - 1; i++) {
   var item = ary[i];
   for (var k = i + 1; k < ary.length; ) {
    //将for循环中的 k++ 写在 else里边，说明只在没有删除的情况下走 k++
      if (item === ary[k]) {
         ary.splice(k, 1);
      }else{
         k++;
      }
   }
}
```

方案二：基于对象的属性名不能重复原理来实现数组去重

> *1.创建一个空对象 2.依次遍历数组中的每一项,把每一项存储的值，当做对象的属性名和属性值存储起来*

```
var obj = {};
for (var i = 0; i < ary.length; i++) {
   var item = ary[i];
   if (typeof obj[item] !== 'undefined') {
      /* ary.splice(i, 1);
       * i--;//=>防止数组塌陷
       *
       * 这种删除方式不好，如果数组很长，我们删除某一项，后面索引都需要重新计算，非常耗性能
       */

      /*
       * 1.我们把数组最后一项的结果获取到，替换当前项内容

       * 2.在把数组最后一项删除
```

```
        * [12,23,34,56] 想要删除23
        *   先让56替换23 [12,56,34,56]
        *   在把最后一项删除 [12,56,34]
        */
        ary[i] = ary[ary.length - 1];
        ary.length--;
        i--;
        continue;
    }
    obj[item] = item;//=>obj[1]=1 =>{1:1}
}
```

```
Array.prototype.myUnique = function myUnique(){
    let obj={};
    for(var i=0;i<this.length;i++){
        let cur = this[i];
        if(typeof obj[cur]!=='undefined'){
            this[i] = this[this.length-1];
            this.length--;
            i--;
            continue;
        }
        obj[cur] = cur;
    }
    obj = null;
    return this;
}

let arr = [1,23,4,5,6,1,2,3,45,2,63];
console.log(arr.myUnique());//[ 1, 23, 4, 5, 6, 63, 2, 3, 45 ]
```

方案三：使用indexOf() 不兼容IE6~8

> *1.遍历当前数组 2.用数组的每一项和它后边剩余的数组做对比， 利用 `indexOf` 方法判断数组后边是否还有相同的项 3.有就删除（没有就进行下一次循环）*

```
let ary = [1,23,4,5,6,7,2,12,3,5,6];
for(var i=0;i<ary.length;i++){
    let cur = ary[i];
    let cru_next_ary = ary.slice(i+1);
    if((cru_next_ary.indexOf(cur))>-1){
        ary.splice(i,1);
        i--;
    }
}
console.log(ary);//[ 1, 23, 4, 7, 2, 12, 3, 5, 6 ]
```

方案四： 相邻比较

> *1.先对数组进行排序 2.对排序后 的数组进项相邻元素比较，如果相同，删除一个*

```
Array.prototype.myUnique = function myUnique(){
    //1.先排序
    let arr = this.sort((a,b)=> a-b);
    //2.相邻比较
    for(var i=0;i<arr.length-1;i++){
        //let cur =arr[i];
        //let nextCur = arr[i+1];
        //if(cur==nextCur){
        if(arr[i]===arr[i+1]){
            arr.splice(i+1,1);
            i--;
            continue;
        }

    }
```

```
    return arr;

}

let arr = [1,2,4,5,6,1,2,3,4,5,2,63];
console.log(arr.myUnique());//[ 1, 2, 4, 5, 6, 3, 63 ]
```

方案五：基于ES6的set数据结构

```
Array.prototype.myUnique = function myUnique(){
    return  Array.from(new Set([...this]));
}

let arr = [1,2,4,5,6,1,2,3,4,5,2,63];
console.log(arr.myUnique());//[ 1, 2, 4, 5, 6, 3, 63 ]
```

方案六 forEach方法去重

```
Array.prototype.distinct=function (){
        var a=[],obj={},temp=this;
        temp.forEach(function (value, index, temp){
                if(!obj[typeof (value)+value])    {
                        a.push(value);
                        obj[typeof (value)+value]=true;
                }
        });
        return a;
};
```

2.7.3 数组和面向对象

数组和对象的关系

> □ ***数组也是对象***：*使用typeof对一个数组运算后会返回的是字符串“object”，这便足以说明数组就是对象。*
>
> □ ***数组是有序存储数据的集合***：*数组是一段线性分配的内存，它可以通过整数去计算偏移并访问其中的元素，由此可以看出数组是有序对象存储的集合。*
>
> □ ***数组是一种访问速度很快的数据结构***：*因为数组的属性都是以数字形式存放在栈中，从栈中获取数据是非常快捷的*
>
> □ ***类数组对象***：*而由于JavaScript中没有数组这样的数据结构，但却提供了一种拥有一些类数组特性的对象，将数组的下标转变成字符串，将其作为属性，这种我们称之为类数组对象。虽然类数组对象的访问速度比真正的数组慢，但它使用起来却更加的灵活方便，且其属性的检索和更新方式和对象都是一模一样的。*

解决对象去重问题

```
function multiTypeDistinct (arr){
  //判断对象类型的方法
  function isEqual(obj1,obj2){
    //判断两个对象的地址是否一样，地址一样则必相等，这里只为了优化性能
    if(obj1===obj2){
      return true;
    }
    if(typeof(obj1)=="object"&&typeof(obj2)=="object"){
      //判断两个对象类型一致且为object类型
      var count=0;
      for(var attr in obj1){
        count++;

        if(!isEqual(obj1[attr],obj2[attr])){
```

```
                return false;
            }
        }
        for(var attr in obj2){
            count--;
        }
        return count==0;
    }else if(typeof(obj1)=="function"&&typeof(obj2)=="function"){
        //判断两个对象类型一致且为function类型
        if(obj1.toString()!==obj2.toString()){
            return false;
        }
    }else { //判断两个对象类型不一致，再判断值是否相等
        if(obj1!=obj2){
            return false;
        }
    }
    return true;
};
//temp作为传入数组arr的备份，在不改变原数组的基础上进行去重操作
var temp=arr.slice(0);
for(var i=0;i<temp.length;i++){
    for(j=i+1;j<temp.length;j++){
        if(isEqual(temp[j],temp[i])){
            temp.splice(j,1);//删除该元素
            j--;
        }
    }
}
return temp;
}
```

将类数组型转化为数组

```
function listToArray(eles){
    try{
        return Array.prototype.slice.call(eles,0)
    }catch(e){//如果try里的代码出现了异常，则执行catch里的代码
        var a=[];
        for(var i=0;i<eles.length;i++){
            a.push(eles[i]);
        }
        return a;
    }
}
```

字符串转换成数组去重

```
var str='777748908883859999';
var arr=[].slice.call(str,0);//将字符串转化为数组
function stringDist(arr){
    var obj={},a=[];
    for(var i=0,len=arr.length;i<len;i++){
        var temp=arr[i];
        if(!obj.hasOwnProperty(temp)){  //判断obj对象是否具有temp属性
            obj[temp]=1;
            a.push(temp);
        }else{
            obj[temp]++;
        }
    }
    return a;
    //return obj;  //返回每一项及其重复的个数
}
console.log(stringDist(arr));
```

数组克隆的实现

```
Array.prototype.clone = function(){
    return this.slice(0);  //利用js内置的方法，执行效率特别高
};
Array.prototype.clone = function(){
    var a = [];
    for(var i=0;i<this.length;i++){
        a.push(this[i]);
    }
    return a;
};
//测试
var a = [3,44,5,6,66];
var a1 = a.clone();
alert(a1 == a);
```

3. 基础算法

3.1 数组的冒泡排序

> *冒泡排序*

- 冒泡排序（Bubble Sort），是一种计算机科学领域的较简单的排序算法。
- 它重复地走访过要排序的元素列，依次比较两个相邻的元素，如果他们的顺序（如从大到小、首字母从A到Z）错误就把他们交换过来。走访元素的工作是重复地进行直到没有相邻元素需要交换，也就是说该元素已经排序完成。

- 这个算法的名字由来是因为越大的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。

> *冒泡排序算法的原理如下:*

1. 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素做同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
/**
 * bubbleSort : 数组冒泡排序
 * @ parameter
 *     ary :  需要进行排序的数组
 * @ return
 *     [array] 返回一个排序后的数组（升序）
 *  by BigSpinach on 2018/11/03
 */
function bubbleSort(ary){
   for (var i = 0; i < ary.length-1; i++) {
      for(var j =0;j< ary.length-1-i;j++){
         if(ary[j]>ary[j+1]){
            /*
            实现a,b交换位置
            //方案1:
            a=a+b;
            b=a-b;     //-----> a+b-b=a
            a=a-b;     //-----> a=a-b=(a+b)-(a+b-b)=b

            //方案2:
            temp = a ;

            a=b;
```

```
            b=temp;
            */
            ary[j] =ary[j]+ary[j+1];
            ary[j+1] = ary[j]-ary[j+1] ;
            ary[j] = ary[j]-ary[j+1]  ;
          }
        }
    }
    return ary;
}

let ary= [12,34,5,67,33,34];
console.log(bubbleSort(ary));//[ 5, 12, 33, 34, 34, 67 ]
```

3.2 函数的递归算法

> *递归的定义*　　*程序调用自身的编程技巧称为递归（recursion）。递归做为一种算法在程序设计语言中广泛应用。 一个过程或函数在其定义或说明中有直接或间接调用自身的一种方法，它通常把一个大型复杂的问题层层转化为一个与原问题相似的规模较小的问题来求解，递归策略只需少量的程序就可描述出解题过程所需要的多次重复计算，大大地减少了程序的代码量。递归的能力在于用有限的语句来定义对象的无限集合。一般来说，递归需要有边界条件、递归前进段和递归返回段。当边界条件不满足时，递归前进；当边界条件满足时，递归返回。*

> *递归，就是在运行的过程中调用自己。*　*构成递归需具备的条件：*

1. 子问题须与原始问题为同样的事，且更为简单；
2. 不能无限制地调用本身，须有个出口，化简为非递归状况处理。

在数学和计算机科学中，递归指由一种（或多种）简单的基本情况定义的一类对象或方法，并规定其他所有情况都能被还原为其基本情况。

> *递归的缺点* *递归算法解题相对常用的算法如普通循环等，运行效率较低*

案例：求出1~100之间能同时被3 和4 整除的数，并求出这些数的和

```
//方案一：传统for循环
function getNumber() {
   let total = 0;
   let arr = [];
   for (var i = 0; i <= 100; i++) {
      if (i % 3 === 0 && i % 4 === 0) {
         arr.push(i);
         total += i;
      }
   }
   return {
         arr,
         total
      }
}
console.log(getNumber());
//{ arr: [ 0, 12, 24, 36, 48, 60, 72, 84, 96 ], total: 432 }


//方案二： 使用递归思想
let obj = {
      arr: [],
      total: 0
   };
function getNumberWithRecursion(i) {
   if (i > 100) {
      return;
   };
   if (i % 3 === 0 && i % 4 === 0) {
      obj.arr.push(i);
      obj.total += i;
      //console.log(obj);

      return i + getNumberWithRecursion(i + 1);
```

```
    }
    return getNumberWithRecursion(i + 1);


}


console.log(getNumberWithRecursion(1));
//{ arr: [ 12, 24, 36, 48, 60, 72, 84, 96 ], total: 432 }
```

> *1、编写一个方法用于验证指定的字符串是否为反转字符，返回true和false。请用递归算法实现。（反转字符串样式为'abcdedcba"）*

```
let str = 'abcdedcba';
// 1. 使用for循环
function isReverseStr(str) {
    for (var i = 0; i < str.length; i++) {
        if (str[i] === str[str.length - 1 - i]) {
            return true;
        }
        return false;
    }
}
//console.log(isReverseStr(str));

//2. 使用递归
let a=0;
let b=str.length-1;
function isReverseStrWithRecursion(str) {
    if(a===b){
        return true;
    }
    if(str[a]===str[b]){

        a+=1;
```

```
        b-=1;
        return isReverseStrWithRecursion(str);
    }
    return isReverseStrWithRecursion(str);
}

console.log(isReverseStrWithRecursion(str));//true

//上边的方法可移植性很差
function isReverseStrWithRecursion(str) {
    if(str.length<=1){
        return true;
    }
    let strFirst = str[0];
    let strLast = str[str.length-1];
    if(strFirst===strLast){
        //console.log(str);
        //str中删除第一项和最后一项
        str = str.slice(1,str.length-1);
        //console.log(str);
        return isReverseStrWithRecursion(str);
    }else{
        return false;
    }
}
```

3.3 快速排序

> *快速排序（Quicksort）是对冒泡排序的一种改进。 快速排序由C. A. R. Hoare在1962年提出。它的基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。*

> *设要排序的数组是A[0]……A[N-1]，首先任意选取一个数据（通常选用数组的第一个数）作为关键数据，然后将所有比它小的数都放到它前面，所有比它大的数都放到它后面，这个过程称为一趟快速排序*
>
> *一趟快速排序的算法是：*
>
> *1. 设置两个变量i、j，排序开始的时候：i=0，j=N-1；*

2. 以第一个数组元素作为关键数据，赋值给key，即key=A[0]；
3. 从j开始向前搜索，即由后开始向前搜索(j--)，找到第一个小于key的值A[j]，将A[j]和A[i]互换；
4. 从i开始向后搜索，即由前开始向后搜索(i++)，找到第一个大于key的A[i]，将A[i]和A[j]互换；
5. 重复第3、4步，直到i=j；（3,4步中，没找到符合条件的值，即3中A[j]不小于key,4中A[i]不大于key的时候改变j、i的值，使得j=j-1，i=i+1，直至找到为止。找到符合条件的值，进行交换的时候i， j指针位置不变。另外，i==j这一过程一定正好是i+或j-完成的时候，此时令循环结束）。

```
{
  let arr1 = [1, 2, 3, 4, 5, 2, 2, 1, 4, 5, 2];

  function quickSort(arr) {
    if(arr.length<=1){
      return arr;
    }

    // let key = arr[0];
    // arr.splice(0,1);
    let key = arr.splice(0,1);
    let arrLeft = [];
    let arrRight = [];
    for(var i=0;i<arr.length;i++){
      // if(arr[i]<key){
      //  arrLeft.push(arr[i]);
      // }else{

      //  arrRight.push(arr[i]);
```

```
        // }
        arr[i]<key?arrLeft.push(arr[i]):arrRight.push(arr[i]);
      }
      return quickSort(arrLeft).concat(key,quickSort(arrRight));
      //return [...quickSort(arrLeft)].concat(key,[...quickSort(arrRight)]);
    }
    console.log(quickSort(arr1));
}
```

3.4 插入排序

> *有一个已经有序的数据序列，要求在这个已经排好的数据序列中插入一个数，但要求插入后此数据序列仍然有序，这个时候就要用到一种新的排序方法——插入排序法,插入排序的基本操作就是将一个数据插入到已经排好序的有序数据中，从而得到一个新的、个数加一的有序数据，算法适用于少量数据的排序，时间复杂度为O(n^2)。是稳定的排序方法。插入算法把要排序的数组分成两部分：第一部分包含了这个数组的所有元素，但将最后一个元素除外（让数组多一个空间才有插入的位置），而第二部分就只包含这一个元素（即待插入元素）。在第一部分排序完成后，再将这个最后元素插入到已排好序的第一部分中。 插入排序的基本思想是：每步将一个待排序的记录，按其关键码值的大小插入前面已经排序的文件中适当位置上，直到全部插入完为止。*

---

4. DOM

DOM（文档对象模型(Document Object Model)）

> *文档对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展标志语言的标准编程接口。在网页上，组织页面（或文档）的对象被组织在一个树形结构中，用来表示文档中对象的标准模型就称为DOM。Document Object Model的历史可以追溯至1990年代后期微软与Netscape的“浏览器大战”，双方为了在JavaScript与JScript一决生死，于是*

> *大规模的赋予浏览器强大的功能。微软在网页技术上加入了不少专属事物，既有VBScript、ActiveX、以及微软自家的DHTML格式等，使不少网页使用非微软平台及浏览器无法正常显示。DOM即是当时蕴酿出来的杰作。*

DOM的分级

> *根据W3C DOM规范，DOM是HTML与XML的应用编程接口（API），DOM将整个页面映射为一个由层次节点组成的文件。有1级、2级、3级共3个级别。*

- 1级DOM在1998年10月份成为W3C的提议，由DOM核心与DOM HTML两个模块组成。DOM核心能映射以XML为基础的文档结构，允许获取和操作文档的任意部分。DOM HTML通过添加HTML专用的对象与函数对DOM核心进行了扩展。
- 鉴于1级DOM仅以映射文档结构为目标，DOM 2级面向更为宽广。通过对原有DOM的扩展，2级DOM通过对象接口增加了对鼠标和用户界面事件（DHTML长期支持鼠标与用户界面事件）、范围、遍历（重复执行DOM文档）和层叠样式表（CSS）的支持。同时也对DOM 1的核心进行了扩展，从而可支持XML命名空间。
  2级DOM引进了几个新DOM模块来处理新的接口类型：

- DOM视图：描述跟踪一个文档的各种视图（使用CSS样式设计文档前后）的接口；
- DOM事件：描述事件接口；
- DOM样式：描述处理基于CSS样式的接口；
- DOM遍历与范围：描述遍历和操作文档树的接口；

- 3级DOM通过引入统一方式载入和保存文档和文档验证方法对DOM进行进一步扩展，DOM3包含一个名为“DOM载入与保存”的新模块，DOM核心扩展后可支持XML1.0的所有内容，包括XML Infoset、 XPath、和XML Base。

0级别DOM

> *当阅读与DOM有关的材料时，可能会遇到参考0级DOM的情况。`需要注意的是并没有标准被称为0级DOM`，它仅是DOM历史上一个参考点（0级DOM被认为是在Internet Explorer 4.0 与Netscape Navigator4.0支持的最早的DHTML）。*

优点和缺点

> *DOM的优势主要表现在：易用性强，使用DOM时，将把所有的XML文档信息都存于内存中，并且遍历简单，支持XPath，增强了易用性。*
>
> *DOM的缺点主要表现在：效率低，解析速度慢，内存占用量过高，对于大文件来说几乎不可能使用。另外效率低还表现在大量的消耗时间，因为使用DOM进行解析时，将为文档的每个element、attribute、processing-instruction和comment都创建一个对象，这样在DOM机制中所运用的大量对象的创建和销毁无疑会影响其效率。*

DOM树(文档树)

> *当浏览器加载HTML页面的时候，首先就是DOM结构的计算，计算出来的DOM结构就是DOM树（把页面中的HTML标签像树桩结构一样，分析出之间的层级关系）*
>
> *DOM树描述了标签和标签之间的关系（节点间的关系），我们只要知道任何一个标签，都可以依据DOM中提供的属性和方法，获取到页面中任意一个标签或者节点*

4.1 JS中获取DOM元素的方法

`getElementById`

> *通过元素的ID获取指定的元素对象，使用的时候都是`document.getElementById('')`此处的document是限定了获取元素的范围，我们把它称之为“上下文(context)”*

> *在JS中，默认会把元素的ID设置为变量（不需要在即获取设置），而且ID重复，获取的结果就是一个集合，包含所有ID项，不重复就是一个元素对象（类似于ById获取的结果）*
>
> 1. *getElementById的上下文只能是document*
>   *因为严格意义上，一个页面中的ID是不能重复的，浏览器规定在整个文档中既可以获取这个唯一的ID*
> 2. *如果页面中的ID重复了，我们基于这个方法只能获取到第一个元素，后面相同ID元素无法获取*
> 3. *在IE6~7浏览器中，会把表单元素(input...)的name属性值当做ID来使用（建议：以后使用表单元素的时候，不要让name和id的值有冲突）*

getElementsByTagName

> `[context].getElementsByTagName` *在指定的上下文中，根据标签名获取到一组元素集合（HTMLCollection）*

1. 获取的元素集合是一个类数组（不能直接的使数组中的方法）
2. 它会把当前上下文中，子子孙孙（后代）层级内的标签都获取到（获取的不仅仅是儿子级的）
3. 基于这个方法获取到的结果永远都是一个集合（不管里面是否有内容，也不管有几项，它是一个容器或者集合），如果想操作集合中具体的某一项，需要基于索引获取到才可以

getElementsByClassName -不兼容IE6~8

> `[context].getElementsByClassName()` *在指定的上下文中，基于元素的样式类名（class='xxx'）获取到一组元素集合*
>
> 1. *真实项目中，我们经常是基于样式类来给元素设置样式，所以在JS中，我们也会经常基于样式类来获取元素，但是此方法在IE6~8下不兼容*
> 2. *兼容处理方案参考*

```
Node.prototype.queryElementsByClassName = function
```

```
queryElementsByClassName() {
    if (arguments.length === 0) return [];
    var strClass = arguments[0],
        nodeList = utils.toArray(this.getElementsByTagName('*'));
    strClass = strClass.replace(/^ +| +$/g, '').split(/ +/);
    for (var i = 0; i < strClass.length; i++) {
        var reg = new RegExp('(^| +)' + strClass[i] + '( +|$)');
        for (var k = 0; k < nodeList.length; k++) {
            if (!reg.test(nodeList[k].className)) {
                nodeList.splice(k, 1);
                k--;
            }
        }
    }
    return nodeList;
};
```

getElementsByName

> *document.getElementsByName() 它的上下文也只能是document，在整个文档中，基于元素的name属性值获取一组节点集合（也是一个类数组）*
>
> 1. *在IE浏览器中（IE9及以下版本），只对表单元素的name属性起作用（正常来说：我们项目中只会给表单元素设置name，给非表单元素设置name，其实是一个不太符合规范的操作）*
> 2. *上下文也只能是document*

querySelector ---->不兼容IE6~8 ------>没有DOM映射

> *[context].querySelector() 在指定的上下文中基于选择器（类似于CSS选择器）获取到指定的元素对象（获取的是一个元素，哪怕选择器匹配了多个，我们只获取第一个）*

querySelectorAll ---->不兼容IE6~8 ------>没有DOM映射

> *在querySelector的基础上，获取到选择器匹配到的所有元素，结果是一个节点集合（NodeList）*

获取浏览器一屏幕的宽度和高度（兼容所有的浏览器）

```
//=>需求：获取浏览器一屏幕的宽度和高度（兼容所有的浏览器）
document.documentElement.clientWidth || document.body.clientWidth

document.documentElement.clientHeight || document.body.clientHeight
```

获取当前页面中所有ID为HAHA的和元素（兼容所有的浏览器）

```
//=>不能使用querySelectorAll
/*
 * 1.首先获取当前文档中所有的HTML标签
 * 2.依次遍历这些元素标签对象，谁的ID等于HAHA，我们就把谁存储起来即可
 */
function queryAllById(id) {
    //->基于通配符*获取到整个文档中所有的HTML标签
    var nodeList = document.getElementsByTagName('*');

    //->遍历集合中的每一项，把元素ID和传递ID相同的这一项存储起来
    var ary = [];
    for (var i = 0; i < nodeList.length; i++) {
        var item = nodeList[i];
        item.id === id ? ary.push(item) : null;
    }
    return ary;
}
console.log(queryAllById('HAHA'));
```

4.2 节点 （node）

> *在一个HTML文档中出现的所有东西都是节点*
>
> - *元素节点（HTML标签）*
> - *文本节点（文字内容）*
> - *注释节点（注释内容）*
> - *文档节点（document）*
> - *...*

4.2.1 区分节点特征的方法

- [描述节点和节点之间关系的属性]
- nodeType nodeName nodeValue
- 元素节点 1 大写标签名 null
- 文本节点 3 #text   文本内容
- 注释节点 8 #comment 注释内容
- 文档节点 9 #document null

每一种类型的节点都会有一些属性区分自己的特性和特征

- nodeType：节点类型
- nodeName：节点名称
- nodeValue：节点值

`元素节点` nodeType：1 nodeName：大写标签名 nodeValue：null

`文本节点` nodeType：3 nodeName：'#text' nodeValue：文本内容

在标准浏览器中会把空格/换行等都当做文本节点处理

`注释节点` nodeType：8 nodeName：'#comment' nodeValue：注释内容

`文档节点` nodeType：9 nodeName：'#document' nodeValue：null

4.2.2 描述节点之间关系的属性

parentNode

> *获取当前节点唯一的父亲节点*

childNodes

> *获取当前元素的所有子节点*
>
> - *子节点：只获取儿子级别的*
> - *所有：包含元素节点、文本节点等*

children

> *获取当前元素所有的元素子节点*
>
> *在IE6~8中会把注释节点也当做元素节点获取到，所以兼容性不好*

previousSibling

> *获取当前节点的上一个哥哥节点（获取的哥哥可能是元素也可能是文本等）*
>
> *previousElementSibling：获取上一个哥哥元素节点（不兼容IE6~8）*

nextSibling

> *获取当前节点的下一个弟弟节点*
>
> *nextElementSibling：下一个弟弟元素节点（不兼容）*

firstChild

> *获取当前元素的第一个子节点（可能是元素或者文本等）*
>
> *firstElementChild*

lastChild

> *获取当前元素的最后一个子节点*
>
> *lastElementChild*

需求一：获取当前元素的所有元素子节点

> *基于children不兼容IE低版本浏览器（会把注释当做元素节点）*

```
/*
 * children：get all the element nodes of the current element
 * @parameter
 *    curEle：[object] current element
 * @return
 *    [Array] all the element nodes
 * by team on 2018/04/07 12:36
 */
function children(curEle) {
   //=>首先获取当前元素下所有的子节点,然后遍历这些节点,筛选出元素的(NODE-TYPE===1),把筛选出来的结果单独存储起来即可
   var nodeList = curEle.childNodes,
       result = [];
   for (var i = 0; i < nodeList.length; i++) {
       var item = nodeList[i];
       if (item.nodeType === 1) {
           result.push(item);
       }
   }
   return result;
}
console.log(children(course));
```

需求二：获取当前元素的上一个哥哥元素节点

> *previousSibling：上一个哥哥节点 previousElementSibling：上一个哥哥元素节点，但是不兼容*

```
/*
 * prev：get the last elder brother element node of the current element
 * @parameter
 *   curEle：[object] current element
 * @return
 *   [object] last elder brother element
 * by team on 2018/04/07 12:44
 */
function prev(curEle) {
  //=>先找当前元素的哥哥节点,看是否为元素节点,不是的话,基于哥哥,找哥哥的上一个哥哥节点...一直到找到元素节点或者已经没有哥哥了(说明我就是老大)则结束查找
  var pre = curEle.previousSibling;
  while (pre && pre.nodeType !== 1) {
    /*
     * pre && pre.nodeType !== 1
     *   pre是验证还有没有，这样写代表有，没有pre是null
     *   pre.nodeType是验证是否为元素
     */
    pre = pre.previousSibling;
  }
  return pre;
}
```

4.2.3 DOM的增删改

createElement（）、createAttibute（）、createTextNode（）、 createFragment

> *创建一个元素标签(元素对象)* `document.createElement([标签名])`

appendChild

> *把一个元素对象插入到指定容器的末尾* `[container].appendChild([newEle])`

insertBefore

> *把一个元素对象插入到指定容器中某一个元素标签之前*
> `[container].insertBefore([newEle],[oldEle])`

cloneNode

> *把某一个节点进行克隆*
>
> `[curEle].cloneNode()`*：浅克隆，只克隆当前的标签*
> `[curEle].cloneNode(true)`*：深克隆，当前标签及其里面的内容都一起克隆了*

removeChild

> *在指定容器中删除某一个元素*
>
> `[container].removeChild([curEle])`

set/get/removeAttribute

> *设置/获取/删除 当前元素的某一个自定义属性*

```
var oBox=document.getElementById('box');

//=>把当前元素作为一个对象，在对象对应的堆内存中新增一个自定义的属性
oBox.myIndex = 10;//=>设置
console.log(oBox['myIndex']);//=>获取
delete oBox.myIndex; //=>删除

//=>基于Attribute等DOM方法完成自定义属性的设置
oBox.setAttribute('myColor','red'); //=>设置
oBox.getAttribute('myColor'); //=>获取

oBox.removeAttribute('myColor'); //=>删除
```

上下两种机制属于独立的运作体制，不能互相混淆使用
- 第一种是基于对象键值对操作方式，修改当前元素对象的堆内存空间来完成
- 第二种是直接修改页面中HTML标签的结构来完成（此种办法设置的自定义属性可以在结构上呈现出来）

基于setAttribute设置的自定义属性值都是字符串

---

5. RegExp

> *正则：是一个用来处理字符串的规则*
>
> - *1.正则只能用来处理字符串*
> - *2.处理一般包含两方面：*
> - *A:验证当前字符串是否符合某个规则“正则匹配”*
> - *B:把一个字符串中符合规则的字符获取到“正则捕获”*

```
console.dir(new RegExp());
/(?:)/
   dotAll: false
   flags: ""
   global: false
   ignoreCase: false
   lastIndex: 0
   multiline: false
   source: "(?:)"
   sticky: false
   unicode: false
   __proto__:
      compile: ƒ compile()
      constructor: ƒ RegExp()
      dotAll: (...)
      exec: ƒ exec()
```

```
flags: (...)
global: (...)
ignoreCase: (...)
multiline: (...)
source: (...)
sticky: (...)
test: ƒ test()
toString: ƒ toString()
unicode: (...)
Symbol(Symbol.match): ƒ [Symbol.match]()
Symbol(Symbol.replace): ƒ [Symbol.replace]()
Symbol(Symbol.search): ƒ [Symbol.search]()
Symbol(Symbol.split): ƒ [Symbol.split]()
get dotAll: ƒ dotAll()
get flags: ƒ flags()
get global: ƒ global()
get ignoreCase: ƒ ignoreCase()
get multiline: ƒ multiline()
get source: ƒ source()
get sticky: ƒ sticky()
get unicode: ƒ unicode()
__proto__: Object
```

每一个正则都是由修饰“元字符”、“修饰符”两部分组成

5.1创建正则的两种方式

5.1. 1 字面量方式创建——不支持变量的匹配

```
//方式1：字面量方式创建------不支持变量的匹配
let reg =/^\d+$/;
console.log(reg.test(1234567890));//true

 //不支持变量的匹配
 let reg2 = /^\d+"name"+$/;
 //这个的结果是： \d+出现一次或多次，“name 出现 一次 ， ”+出现一次或多次
let name = "liukai";
console.log(reg2.test("2liukai6"));//false
console.log(reg2.test('22"name""'));//true
```

5.1. 2 构造函数的方式new RegExp(‘正则字符串’)

```
//方式2：构造函数方式创建------注意转义字符 \\d
let reg1 = new RegExp("^\\d+$");       console.log(reg1.test("123123"));//true
console.log(reg1.test(123123));//true

//支持变量的匹配
let name = "liukai";
let reg2 = new RegExp("^\\d+"+name+"+$");
console.log(reg2.test("2liukai"));//true
console.log(reg2.test('22"name""'));//false
```

5.2 正则的基本规则

> 1. *正则都是由修饰“元字符”、“修饰符”两部分组成*
> 2. *正则两个斜杠之间包起来的都是“元字符”，斜杠后面出现的都是“修饰符”*

5.2.1常用的修饰符和元字符

【常用的修饰符】

i：ignoreCase 忽略大写小匹配
m：multiline 多行匹配
g：global 全局匹配

【常用的元字符】

> *[量词元字符]*
> *+：让前面的元字符出现一到多次*
> *?：出现零到一次*
> **：出现零到多次 {n}：出现n次*
> *{n,}：出现n到多次*
> *{n,m}：出现n到m次*

> *[特殊意义的元字符]*
> *\：转义字符（把一个普通字符转变为有特殊意义的字符，或者把一个有意义字符转换为普 通的字符）*
> *.：除了\n（换行符）以外的任意字符*
> *\d：匹配一个0~9之间的数字*
> *\D：匹配任意一个非0~9之间的数字（大写字母和小写字母的组合正好是反向的）\w：匹配一个 `0~9` 或 `字母` 或 `_` 之间的字符*
> *\s：匹配一个任意空白字符*
> *\b：匹配一个边界符*
> *x|y：匹配x或者y 中的一个*
> *[az]：匹配az中的任意一个字符*
> *[^az]：和上面的相反，匹配任意一个非az的字符*
> *[xyz]：匹配x或者y或者z 中的一个字符*
> *[^xyz]：匹配除了xyz以外的任意字符*
> *()：正则的小分组，匹配一个小分组（小分组可以理解为大正则中的一个小正则）^：以某一个元字符开始*
> *$：以某一个元字符结束*
> *?:：只匹配不捕获*
> *?=：正向预查*
> *?!：负向预查*
> *……*

> *[普通元字符] 除了以上特殊元字符和量词元字符，其余的都叫做普通元字符：代表本身意义的元字符*

> *[量词元字符：让其左边的元字符出现多少次] *: 出现零到多次 ?: 出现零到一次 +: 出现一到多次 {n}: 出现N次 {n,}: 出现N到多次 {n,m}: 出现N到M次*

5.2.2 [] 的一些细节知识点

【 [] 的一些细节知识点】

```
[xyz]
[^xyz]
[a-z]
[^a-z]
```

> *1.中括号中出现的元字符一般都是代表本身含义的 2.中括号中出现的两位数，不是两位数，而是两个数字中的任意一个*

```
//1.里面出现的元字符一般都是代表本身意义的
[?.&=+]/ 这里出现的符号都是本身的意思
 /[\d]/ 此处的\d依然是0~9中的任意字符

//2.1 正则中的 . 通配符 （除了换行符等空白符）
let reg = /^.+$/;//=>一个正则设置了^和$，那么代表的含义其实就是只能是xxx
console.log(reg.test('n'));//=>true
console.log(reg.test('1'));//=>true
console.log(reg.test('nn'));//=>true
console.log(reg.test('\n'));//=>false

//2.2 [] 中的 . 通配符
let reg = /^[.]+$/;
console.log(reg.test('n'));//=>false

console.log(reg.test('1'));//=>false
```

```
console.log(reg.test('nn'));//=>false
console.log(reg.test('\n'));//=>false
console.log(reg.test('...'));//=>true

//2.3 [\d]或 [\w] 等都表示原 正则元字符的含义
let reg = /^[\d]+$/; //=>\d在这里依然是0~9中的一个数字
console.log(reg.test('0'));//=>true
console.log(reg.test('d'));//=>false

//2.4 []前后有^和$ : 表示以[] 中的字符作为开头或者结尾
let reg = /^[18]$/;//=>不加^和$代表字符串中只要包含xxx即可
//加^和$代表 以1开头或以8结尾
//所以实际意义就是： 出现1或者8
console.log(reg.test('18'));//=>false
console.log(reg.test('1'));//=>true
console.log(reg.test('8'));//=>true

//2.5 []中的 - 连字符
let reg = /^[12-65]$/;
console.log(reg.test('13'));//=>false 不是12~65
console.log(reg.test('7'));//=>false  这个正则的意思是 1或者2~6或者5
console.log(reg.test('3.5'));//=>false

//年龄：18~65之间
/*
 * 18~19  1[89]
 * 20~59  [2-5]\d
 * 60~65  6[0-5]
 */
let reg = /^((1[89])|([2-5]\d)|(6[0-5]))$/;

//=>需求：编写一个规则，匹配 "[object AAA]"
let reg = /^\[object .+\]$/;
console.log(reg.test('[object AAA]'));//=>true
```

> *3.中括号中出现 - 代表两种情况*

```
//1.[a-zA-Z0-9_]/这里的`-`是一个范围链接符（从某一个开始到某一个结束,获取范围中的任何一个）
/[a-zA-Z0-9_]/
//2./[a-z0-9_-]/ 如果`-`左右两边的任何一边并没有内容，它代表本身的意思
 /[a-z0-9_-]/
```

5.2.3 正则分组

> *分组的作用*
>
> - *1.改变默认的优先级 : (x|y)*
> - *2.分组捕获:*
> - *3.分组引用 : ()\1 出现和第一个分组一模一样的内容*

```
let reg = /^18|19$/;
console.log(reg.test('18'));//=>true
console.log(reg.test('19'));//=>true
console.log(reg.test('1819'));//=>true
console.log(reg.test('189'));//=>true
console.log(reg.test('181'));//=>true
console.log(reg.test('819'));//=>true
console.log(reg.test('119'));//=>true

reg = /^(18|19)$/;
console.log(reg.test('18'));//=>true
console.log(reg.test('19'));//=>true
console.log(reg.test('1819'));//=>false
console.log(reg.test('189'));//=>false
console.log(reg.test('181'));//=>false
console.log(reg.test('819'));//=>false

console.log(reg.test('119'));//=>false
```

```
let reg = /^([a-z])([a-z])\2\1$/;
//=>正则中出现的\1代表和第一个分组出现一模一样的内容...
console.log(reg.test('oppo'));
console.log(reg.test('poop'));

//=>分组引用：\1 或者 \2 ...出现和第N个分组一模一样的内容
var reg = /^([a-z])([a-z])\2([a-z])$/; //=> 符合的字符串：foot、book、week、attr、http...
```

【编写一个正则匹配身份证号码】

```
let reg = /^\d{17}(\d|X)$/;//=>简单：只能匹配是否符合格式，不能提取出身份证中的一些信息
// '130828199012040617'
//=>130828 地域
//=>19901204 出生年月
//=>0617 倒数第二位：奇数=男 偶数=女

let reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(?:\d|X)$/;
console.log(reg.exec('130828199012040617'));//=>EXEC实现的是正则捕获，获取的结果是一个数组，如果不匹配获取的结果是null，捕获的时候不仅把大正则匹配的信息捕获到，而且每一个小分组中的内容也捕获到了(分组捕获)：
["130828199012040617", "130828", "1990", "12", "04", "1", index: 0, input: "130828199012040617"]

/*
 * 正则捕获使用的是正则中的EXEC方法
 *  1.如果可以匹配获取的结果是一个数组，如果不能匹配获取的结果是NULL
 *  2.如果我们只在匹配的时候，想要获取大正则中部分信息，我们可以把这部分使用小括号包起来，形成一个分组，这样在捕获的时候，不仅可以把大正匹配的信息捕获到，而且还单独的把小分组匹配的部分信息也捕获到了(分组捕获)


 *  3.有时候写小分组不是为了捕获信息，只是为了改变优先级或者进行分组引
```

```
用，此时我们可以在分组的前面加上“?:”，代表只去匹配，但是不把这个分组内容捕获
 */
```

5.3 正则的一些方法提升

5.3.1 正则捕获 exec()

5.3.1 .1 【正则的懒惰性】

> *每次执行exec的时候，正则只捕获第一个匹配的内容，在不进行任何处理的情况下， 执行多次exec，都捕获的是第一次捕获的内容*

```
let reg =/\d+/;

//第1次执行exec
let arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
console.log(arr);//["1993", index: 5, input: "时间如流水1993我不能着急 0515欲速则不达", groups: undefined]
console.log(arr[0]);//1993
console.log(reg.lastIndex);//0

//第2次执行exec
arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
console.log(arr[0]);//1993
console.log(reg.lastIndex);//0

//第3次执行exec
arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
console.log(arr[0]);//1993
console.log(reg.lastIndex);//0
```

正则懒惰性的原因：

> *lastIndex是正则每次捕获的字符串中开始查找的位置*
> *懒惰模式下，这个lastIndex的值是不发生变化的*
> *所以：每次查找的都是相同的位置，结果也就相同了*

解决正则的懒惰性：

> *正则末尾加g(全局修饰符)*

```
//解决正则的懒惰性
let reg =/\d+/g;
//第1次执行exec
let arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
//console.log(arr);//["1993", index: 5, input: "时间如流水1993我不能着 急0515欲速则不达", groups: undefined]
console.log(arr[0]);//1993    console.log(reg.lastIndex);//0

//第2次执行exec
arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
console.log(arr[0]);//0515
console.log(reg.lastIndex);//18

//第3次执行exec
arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
console.log(arr);//null
console.log(reg.lastIndex);//0

//第4次执行exec  返回null之后，再次执行捕获的话，又会重新开始捕获
/*
   arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
   console.log(arr[0]);//null
   console.log(reg.lastIndex);//0
*/
```

一次性捕获所有的方法

```
function getAllReg(reg,str){
   //捕获一次
   let result=[];
   let arr =[];
   arr=reg.exec(str);//
   while(arr){//如果不为null，就继续
      result.push(arr[0]);
      arr=reg.exec(str);
   }    return result;
}

let reg =/\d+/g;
let str="时间如流水1993我不能着急0515欲速则不达";
//测试...
console.log(getAllReg(reg,str));//["1993", "0515"]
```

5.3..1.2 【正则的贪婪性】

> *正则的每一次捕获都是按照匹配的长的结果去捕获的*
> *解决贪婪性--在量词元素后边加上一个？ 即可*

正则的贪婪性

```
console.log("---------正则的贪婪性------------");
{
   let reg = /\d+/g;
   let arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
   console.log(arr[0]); //1993
   //贪婪：这样的正则匹配到1 ， 19 ， 199，1993都可以，但是它匹配的是1993，
长的那一个
}

   **********************************************************
```

```
    console.log("---------解决正则的贪婪性-----量词元素后加？----");    {
    //解决正则的贪婪
    let reg = /\d+?/g;
    let arr = reg.exec("时间如流水1993我不能着急0515欲速则不达");
    console.log(arr[0]); //1
}

*****************************************
console.log("---------自定义方法解决贪婪，匹配所有-------");
{
    function getNoGreedReg(reg, str) {
        //greed-贪婪
        let arr = [];
        let result = [];
        arr = reg.exec(str);
        while(arr) {
            result.push(arr[0]);
            arr = reg.exec(str);
        }
        return result;
    }

    //测试
    let reg = /\d+?/g;
    let str= "时间如流水1993我不能着急0515欲速则不达";
    console.log(getNoGreedReg(reg,str));//["1", "9", "9", "3", "0", "5", "1", "5"]
}
```

5.3.2 字符串的match方法实现捕获

> *str.match(reg) ——> 把所有的和正则匹配的内容都捕获到*
> *缺点：match只能捕获到大正则匹配的内容，小正则的内容捕获不到*

```
console.log("---------字符串的match方法捕获-----------");
{
   let reg = /\d+/g;
   let str = "时间如流水1993我不能着急0515欲速则不达";

   let result = str.match(reg);
   console.log(result); //Array["1993","0515"]
}
```

5.3.3 正则的分组捕获

```
/**
   * 正则分组的作用:
   * * 1.改变优先级
   * * 2.分组的引用
   * *    \2:代表的是和第2个分组出现的字符一模一样
   * *    \1:代表的是和第1个分组出现的字符一模一样
   * *    \n:代表的是和第n个分组出现的字符一模一样
   * * 3.分组的捕获：正则在捕获的时候，不仅仅把大正则的内容捕获到，而且还会把每 一个小分组的内容捕获到
   * *
* */
//身份证号码
let reg = /^(\d{6})(\d{4})(\d{2})(\d{2})\d{2}(\d)(X|\d)$/;      //         610126--1234---22-----22---55---5---x或2
//          区号---年-----月-----日---不知道-男女--不知道       let str = "12345612342222555X";
let str2 = "2345612342222555X2";

let arr = reg.exec(str);
let arr1 = reg.exec(str2);
console.log(arr);


//["12345612342222555X", "123456", "1234", "22", "22", "5", "X", index: 0, input:
```

```
"12345612342222555X", groups: undefined]       console.log(arr1); //null

//使用match
let arr3 = str.match(reg);
console.log(arr3);
//["12345612342222555X", "123456", "1234", "22", "22", "5", "X", index: 0, input:
"12345612342222555X", groups: undefined]
//发现match跟reg.xexc捕获到的一模一样
//区别在哪？ ------- 倒数第三个小正则和倒数第一个我们不想要
//不要匹配它们 -- （?:\d）小分组中加上？：  只匹配，不捕获
let reg2 = /^(\d{6})(\d{4})(\d{2})(?:\d{2})\d{2}(\d)(?:X|\d)$/;       let arr4 =
reg2.exec(str);
//(5) ["12345612342222555X", "123456", "1234", "22", "5", inde x: 0, input:
"12345612342222555X", groups: undefined]       console.log(arr4);
```

5.3.4 exec 和 match 的区别

> *在正则分组，并且字符串有多个匹配结果的情况下：*

- match捕获的结果只是大正则所匹配的所有结果，
- exec捕获的是大正则和小正则的结果

```
let reg =/liukai(\d+)/g;
let str = "liukai123liukai456liukai789"

//使用match捕获
let arr=str.match(reg); console.log(arr);//["liukai123", "liukai456", "liukai789"]

//使用exec捕获
let arr1= reg.exec(str);
console.log(arr1);// ["liukai123", "123", index: 0, input: "liukai123li
ukai456liukai789", groups: undefined] arr1= reg.exec(str);
console.log(arr1);// ["liukai456", "456", index: 9, input: "liukai123li
ukai456liukai789", groups: undefined]
arr1= reg.exec(str);
console.log(arr1);//["liukai789", "789", index: 18, input: "liukai123li
ukai456liukai789", groups: undefined]
```

5.3.5 正则捕获 replace

str.replace(a,b)的原理

```
//str.replace(a,b)的原理
//把字符串str中a样式的字符串用b替换掉-------他只替换一次
let str ="BigSpinach love program";
//需求把所有的i替换成“哈哈”
let result= str.replace("i","哈哈");
console.log(result);//B哈哈gSpinach love program

//replace他是懒惰的，不会搞后边的
let result2= str.replace("i","哈哈");
console.log(result2);//B哈哈gSpinach love program
```

replace中使用正则

```
//replace中使用正则
let str ="BigSpinach love program";
let reg = /i/g;
let result = str.replace(reg,"哈哈");
console.log(result);//B哈哈gSp哈哈nach love program
```

5.4 正则的一些应用场景

5.4.1 replace中使用匿名函数操作str

【replace的关键点】

> 1. *function()函数执行几次取决于字符串中的字符跟正则匹配成功了几次*
> 2. *function()中的arguments输出的结果跟reg.exec捕获到的结果非常的相似， 即使正则分组，我们也可以通过arguments获取到分组捕获的内容*
> 3. *在function中的return，return的是啥，就表示用啥替换掉当前大正则所捕获到的内容*

```
let str ="BigSpinach love program";
let reg = /i/g;
let result = str.replace(reg,function(){
   console.log(arguments);
   //这里会执行两次是因为正则匹配到了两次 i
   //Arguments(3) ["i", 1, "BigSpinach love program", callee: ƒ, Symbo
l(Symbol.iterator): ƒ]


   //Arguments(3) ["i", 5, "BigSpinach love program", callee: ƒ, Symbo
```

```
l(Symbol.iterator): ƒ]
    return "123";
});
console.log(result);//B123gSp123nach love program
/*replace的关键点
    1.function()函数执行几次取决于字符串中的字符跟正则匹配成功了几次
    2.function()中的arguments输出的结果跟 reg.exec捕获到的结果非常的相似，
        即使正则分组，我们也可以通过arguments获取到分组捕获的内容
    3.在function中的return，return的是啥，就表示用啥替换掉当前大正则所捕获
到的内容
*/
```

5.4.2 replace 练习——获取字符串中看每一个字符出现的次数以及那 个字符出现的次数最多

```
let str="I'm BigSpinach,BigSpinach love program";
//let reg = /(')\{1\}|(\D)\{1\}|(\s)\{1\}/;
let reg = /(\D)\{1\}|(\s)\{1\}/g;
var obj={};
let result =str.replace(reg,function(largeReg,smallReg,index,input){
    //获取到每一个字符
    //console.log(arguments);
    //["I", "I", undefined, 0, "I'm BigSpinach,BigSpinach love prog ram", callee: ƒ,
Symbol(Symbol.iterator): ƒ]
    var character = arguments[0];
    //console.log(smallReg);===arguments[1]
    //将每一个字符当做obj的一个属性添加进去

    /*
        if(obj[character]>=1){
            obj[character]+=1;
        }else{
            obj[character]=1;

        }
```

```
    */
    obj[character]>=1?obj[character]+=1:obj[character]=1;                    });

//console.log(obj);//{I: 1, ': 1, m: 2, " ": 3, B: 2, ...}    //console.log(result);

//用假设法找出大的value
let maxNum =0;
for(let key in obj){
    obj[key]>maxNum?maxNum=obj[key]:null;
}
//console.log(maxNum);
//把所有符合maxNum的key都找到
let arr = [];
for (let key in obj) {
    obj[key]==maxNum? arr.push(key):null
}
//console.log(arr);
console.log("多出现的字符是："+arr.toString()+";次数 是："+maxNum+"。");
```

5.4.3 replace 实现模板引擎的初步思想

```
let str ="My name is {0},我今年{1}岁,i love {2}。";
let reg = /{(\d+)}/g;
let arr=["刘凯", 26, "Program"];
str = str.replace(reg,function(){
    return arr[arguments[1]];
    //return arr[RegExp.$1];//IE不兼容+Chrom也不行了
});

console.log(str); //My name is Program,我今年Program岁,i love Program。 //My name is 刘凯,我今年26岁,i love Program。
```

5.4.4 使用replace拆分url

```
let url = ''http://202.110.112.57/wgdcnccdn.inter.qiyi.comn?
name=xx&age=12#ss';

//使用正则
let reg = /([^?=&#]+)=([^?=&#]+)/g;
let obj={};
url.replace(reg,function(){
   obj[arguments[1]]=  arguments[2]||null;
});
console.log(obj);
```

5.4.3 单词首字母大写

```
var str = 'my name is big-spinach,i am 26 years old,i love program,i will to be a
better programer!';
//var reg = /\b\w+\b/g; //=>\b代表的是边界：单词左右两边是边界，-的左右两边也
是边界，所以这里会把 'big-spinach' 算作二个单词（我们想把它当做一个）

//=>1、先把混淆边界符的中杠替换为下划线
str = str.replace(/-/g, '_');

//=>2、通过边界符匹配到每一个单词
str = str.replace(/\b(\w)(\w*)\b/g, function () {
   return arguments[1].toUpperCase() + arguments[2];
});

//=>3、在把之前替换的下划线重新赋值为中杠
str = str.replace(/_/g, '-');
console.log(str);
```

6. js深入

6.1 变量定义、预解释和闭包

6.1.1 声明变量有5中方式

- var
- let
- const
- function
- class

1.先声明一个变量a，没有赋值

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

let b;
console.log(b);//undefined

//const c;
//console.log(c);//报错

function fn(){}
console.log(fn);//[Function: fn]

class Student {
   constructor(){}
}
console.log(Student);//[Function: Student]
```

6.1.2 预解释(变量提升)

```
console.log(a);//=>undefined
var a = 12;
console.log(a);//12s
```

> *在JS里，这个 预解释阶段 是分为 **三步** 完成的*

- 第一步是在整段JS代码运行之前，浏览器引擎先找到所有的带var关键字的变量，在内存里分配好地址，这是个declare（声明）变量的过程，这个过程是在按顺序执行代码之前完成的，就是说代码在未执行之前，已经就把带var关键字定义的变量安排好了，这个机制叫“预解释”。
- 第二步：当代码正常执行的时候，在内存里分配一个空间保存 ”12” 这个数据。
- 第三步：让数据 ”12” 和 变量a 发生关联，就是把 ”12” 赋给 变量a 。这里相当于定义（defined）应该说这一步是叫“初始化”

【 预解释是在私有作用域内中】

```
var n=9;
var s=’str’;
function fn(){
   alert(n);
   alert(s);
   n=7;
   var n=6;
};
fn();//运行这个函数，会输出什么结果?
```

> ***js中的预解释：凡是带var和function关键字的，都会预解释。就是在执行代码之前，先执行***

> *在fn这个方法里，var n=6这个表达式相当于执行了三步，第一步是在脚本执行之前先定义一个变量n，第二步和第三步是当脚本执行到这一行的时候，出现一块内存来表示6并且把6赋值给n，即： var n和n=6;变量的定义是预解释的，但* ***赋值是不预解释的*** *。*
>
> *变量作用域：当一个函数在运行的时候，就形成了一个自己的私有作用域，这个私有的作用域会开成闭包（封闭的包）。预解释是在私用作用域里的预解释。在查找变量的时候先找自己的这个作用域里是不是定义了这个变量，自已这个作用域里如果没有，则再往上一级作用域里找。一直找到window，如果还找不到，则浏览器就要报错了。 JS里，window是顶级作用域。*

【 预解释是一种毫无节操的机制】

> *1.不管什么条件，这个fn方法都会被预解释的 2.预解释的代码不会受到return的影响。 3.所有的定义赋值操作都是先准备值，然后再赋值的*

```
f=function(){return true;};
g=function(){return false;};
(function(){
 if(g()&&[]==![]){
       f=function f(){return false;};
       function g(){return true;}
 }
})();
```

> ***解释*** *`g()&&[]==![]`* ***这个逻辑表达式：*** *（firefox除外） 这个g函数是在这个闭包作用域里定义，同样会解释。也就是说这里运行的这个g函数是下面定义的，和上面那个g没关系。所以当g()时，会返回true 后边的 `[]==![]` ，先执行![]这个运算，把![]转化为false，那现在就相当于 `[]==false` 了，数组和其它类型转化的时候，先隐式调用它的 `toString` 方法，把数组转为字符，然后再比较。空数组转为字符串，当然是空字符串了，那就相当于``''==false``;而空字符串和false做比较，就是true;*

6.1.3 闭包

【闭包的产生】

> *当一个方法在运行的时候，就会形成一个私有的作用域，在这个作用域里，里面定义的变量不会受到上一级作用域或其它作用域的影响，不会和全局或其它作用域里的变量有冲突。这个由方法运行而产生的私有作用域，就叫闭包。*

> ***闭包其实就是函数在运行的时候产生的那个私有作用域。***

【闭包的应用】 **闭包其实就是利用函数执行形成一个 私有的不销毁作用域** ，从而在这个 不销毁的私有所用域 中搞事情

JQuery里的变量、函数或其它对象，就是定义在闭包里的。

```
(function () {
   function jQuery() {
      //...
   }

   //...
   window.jQuery = window.$ = jQuery;//=>把需要供外面使用的方法，通过给WIN设置属性的方式暴露出去
})();
// jQuery();
// $();
```

Zepto这种方式：基于RETURN把需要共外面使用的方法暴露出去

```
var Zepto=(function () {
   //...
   return {
      xxx:function () {

      }
   };
})();
Zepto.xxx();
```

匿名函数中的闭包应用

```
var name = 'BigSpinach';
var age = 25;

//定义一个匿名函数再让这个匿名函数运行，并给匿名函数传参数
(function(myNamea, myAge) { //这里是给匿名方法定义的形式参数
   var name = 'liukai';
   var age = 25;
   alert("我是：" + myNamea + ",我今年" + myAge + "岁了!"
      +"2018年" + name + "已经都" + age + "岁了！");
})(name, age); //这里是给匿名方法传的参数
```

总结1：在这个例子里，定义了两个全局变量name和age，在下面的匿名函数里也定义了变量name和age,但这两组相同的变量互不影响，匿名方法里的变量属于自己的私有的作用域，这个运行的匿名方法就形成了闭包。如果在这个匿名方法想用到全局变量name和age,可以通过传参数的方式传给这个匿名函数

实名函数中的闭包

```
var name = 'BigSpinach';
var age = 25;
function fn(myNamea, myAge) { //这里是给实名函数定义的形参
   var name = 'liukai';
   var age = 25;
   alert("我是：  " + myNamea + ",我今年" + myAge + "岁了!"
      +"2018年" + name + "已经都" + age + "岁了！  ");
}
fn(name,age); //让函数运行
```

函数中访问全局变量的另一种方式

```
var name = 'BigSpinach';//相当于 window.name='BigSpinach';
var age = 25;
//注意只有var声明的变量才可以使用
console.log(window.name);//'BigSpinach'
console.log(window.age);//25
```

【让定义在闭包里的函数能在闭包外面使用】

> *需要给window暴露一个借口即可*

```
(function(chinaName,chinaAge){
    var name = 'liukai';
    var age = 25;
    function selfIntroduction(){//这个函数是私有，外部不能访问。这就形成了一个闭包方法
        alert("2018年" + name + "已经都" + age + "岁了！ ");
    window.selfIntroduction=selfIntroduction;
})();

//外部访问
selfIntroduction();
//或者
window.selfIntroduction();
```

【闭包的具体实例】

```
//有如下html代码，要求：点击下面的li，会弹出对应的索引号。
<ul>
        <li>列表一</li>
        <li>列表二</li>
        <li>列表三</li>
        <li>列表四</li>
        <li>列表五</li>
</ul>
```

错误的代码

```
var oLis=document.getElementsByTagName('li');
for(var i=0; i< oLis.length; i++){
  oLis [i].onclick=function() {//注意：这里的这个匿名方法，在循环运行的时候这个匿名方法本身并不运行，当点击某个li的时候，这个方法才运行呢。
   alert(i);//这里的这个i不是在这个匿名方法里定义的，而是上一级作用域里定义的。当这句代码运行的时候，循环早已经结束，并且i已经等于oLis.length了。
这里的问题出在这个方法里用到的是上一级作用域里定义的变量，如果要解决这个问题。
}
```

利用函数执行形成一个不销毁的私有作用域机制（闭包），很容易想到解决方案 方案1

匿名函数执行形成的私有作用域保存i的值

```
var oLis=document.getElementsByTagName('li');
for(var i=0; i< oLis.length; i++){
    //这里的i是就不是for循环完后的i了
     ;(function(i){
          oLis[i].onclick=function(){alert(i)}
     })(i);
}
```

实名函数执行形成的私有作用域保存i的值

```
function fn(i){
   oLis[i].onclick=function(){alert(i)}
}

for(var i=0; i< oLis.length; i++){
   fn(i);
}
```

方案2 利用自定义属性值不会反生改变

```
var oBox =document.getElementById('box');
var oLis=oBox.getElementsByTagName('li');

for(var i=0; i< oLis.length; i++){
    //留住当前的i的值
    //给每个li元素对象定义（添加）自定义属性
    //用那个自定义的属性值保存每个li的i的值
    oLis[i].myIndex = i+1;
    //console.log(oLis);
    oLis[i].onclick= function(){
        //alert(i);
        //此时就应该显示自定义属性的值了
        alert(this.myIndex);
    }
}
```

方案3 基于ES6的块级作用域

```
for(let i=0; i< oLis.length; i++){
//循环体也是块级作用域，初始值设置的变量是当前本次块级作用域中的变量(形成了 oLis.length; 个块级作用域，每个块级作用域中都有一个私有变量I，变量值就是每一次循环I的值)
    oLis[i].onclick= function(){
        alert(i);
    }
}
```

6.1.4 关于重名变量和预解释

```
function a(x) {
  return x * 2;
}
var a;
console.log(a);
```

【重名的处理】

> *带VAR和FUNCTION关键字声明相同的名字，这种也算是重名了（其实是一个FN，只是存储值的类型不一样）*
>
> *重名的处理：如果名字重复了，不会重新的声明，但是会重新的定义（重新赋值）[不管是变量提升还是代码执行阶段皆是如此]*
>
> *带var的在提升阶段只把声明处理了,赋值操作没有处理,所以在代码执行的时候需要完成赋值*

> ***在JavaScript中，使用var语句多次声明一个同名变量是合法的并且也没有什么害处。**带var 关键字或function关键字的是要预解释的（被提前声明）。但用var 去声明一个变量的时候，只是被提前声明，不会被赋值，就是只是被预解释，但不是执行等号的这个赋值运算。而用function去定义变量的时候，不但会被提前声明，还要被初始化一个值。*

```
var fn ;
function fn() {}
console.log(fn);//ƒ fn() {}
fn = 12;
console.log(fn);//12
```

6.1.5 关于全局变量和预解释

```
if (!("a" in window)) {
    var a ="BigSpinach" ;
}
alert(a);//
```

> *用var关键字声明的变量是预解释的，并且预解释是一种无节操的机制，所以在这段代码执行之前，a已经被声明了，也就是说全局下已经有a这么一个变量了，所以!("a" in window)的结果是false，所以就执行不到* `a ="BigSpinach" ;`

6.1.6 关于闭包和作用域

```
function fo(){
    var i=0;
    return function(n){
        return n+i++;
    }
};

var f=fo();
var a = f(15);
var b = fo()(15);
var c = fo()(20);
var d = f(20);

console.log(a,b,c,d);//
```

> *a和d执行使用的都是 `fo()` 函数执行形成的那个私有函数作用域，所以在第一次执行 （也就是a=f(15)的时候，fo()函数中声明的变量是不销毁的，当下一次执行 （也就是 d = f （20）） 的时候，里边的i的值已经被第一次执行f （15）的时候修改了，所以...*
>
> *b和c 的返回值，由于b和c都是 `fo()(15)` 先执行一次 `fo()` 函数，所以他们的宿主函数是不同的，所以他们后边修改i的值每一次都是新的，不会对其他的函数产生影响*

6.1.7 变量提升的细节知识点

> *当栈内存(作用域)形成，JS代码自上而下执行之前，浏览器首先会把所有带“VAR”/“FUNCTION”关键词的进行提前“声明”或者“定义”，这种预先处理机制称之为“变量提升”*

【变量提升阶段】

> *带 `“var”` 的只声明未定义 带 `“function”` 的声明和赋值都完成了*

```
console.log(a);//=>undefined
var a = 12;
console.log(a);//12

console.log(fn);//ƒ fn(){}
function fn(){}
console.log(fn);//ƒ fn(){}
```

【只对等号左边进行变量提升】

> *只对等号左边进行变量提升 只对等号左边进行变量提升（等号右边是值，是不进行变量提升的） 所以===>函数 `return` 后面的也是值，也是不需要变量提升的)*

```
console.log(a,b);//报错
//Uncaught ReferenceError: b is not defined at <anonymous>:1:15
var a=b=1;
console.log(a,b);
```

【条件判断下的变量提升】

> *在当前作用域下，不管条件是否成立都要进行变量提升 带 `function` 的在老版本浏览器渲染机制下，声明和定义都处理，但是为了迎合ES6中的块级作用域，新版浏览器对于函数（在条件判断中的函数），不管条件是否成立，都只是先声明，没有定义，类似于VAR*

```
console.log(fn);//=>undefined
if (1 === 1) {
  console.log(fn);//=>函数本身：当条件成立，进入到判断体中（在ES6中它是一个块级作用域）第一件事并不是代码执行，而是类似于变量提升一样，先把FN声明和定义了，也就是判断体中代码执行之前，FN就已经赋值了
  function fn() {
    console.log('ok');
  }
}
console.log(fn);//=>函数本身
```

> *变量提升只发生在当前作用域（例如：开始加载页面的时候只对全局作用域下的进行提升，因为此时函数中存储的都是字符串而已）*
>
> *在全局作用域下声明的函数或者变量是“全局变量”，同理，在私有作用域下声明的变量是“私有变量”*

【重名的处理】

> *函数名其实也是变量名 var fn / function fn 这两个是一个fn）变量提升阶段（以及代码执行阶段），不会重复声明，但是会重新赋值*

【带var 和不带 var的区别】

> *不加 `var` 的本质是 `window` 的属性 全局变量和 `window` 中的属性存在“映射机制” 全局变量值修改， `window` 的属性值也跟着修改*

```
console.log('a' in window);//返true/false
```

【let和const 和class】

> *在ES6中基于LET/CONST等方式创建变量或者函数,不存在变量提升机制 切断了全局变量和WINDOW属性的映射机制*

> *在相同的作用域中，基于LET不能声明相同名字的变量（不管用什么方式在当前作用域下声明了变量，再次使用LET创建都会报错*

ES6的语法检测机制

> *虽然没有变量提升机制，但是在当前作用域代码自上而下执行之前，浏览器会做一个重复性检测（语法检测）：自上而下查找当前作用域下所有变量，一旦发现有重复的，直接抛出异常，代码也不会在执行了（虽然没有把变量提前声明定义，但是浏览器已经记住了，当前作用域下有哪些变量）*

```
console.log(a);//=>Uncaught ReferenceError: a is not defined
let a = 12;
console.log(window.a);//=>undefined
console.log(a);//=>12
```

【暂时性死区】

> *基于LET创建变量，会把大部分{}当做一个私有的块级作用域（类似于函数的私有作用域），在这里也是重新检测语法规范，看一下是否是基于新语法创建的变量，如果是按照新语法规范来解析*

```
var a = 12;
if (true) {
    console.log(a);//=>Uncaught ReferenceError: a is not defined
    let a = 13;
}
```

【全局变量和私有变量】

> *变量提升*

- var
- function

> *在私有作用域中，只有以下两种情况是私有变量*
>
> - *声明过的变量(带var/function)*
> - *形参也是私有变量(相当于 在私有作用域中 `var 变量` ,并赋值)* ***注意：形参先赋值，后进行变量提升***
> - *剩下的都不是自己私有的变量，都需要基于作用域链的机制向上查找*

```
var ary = [12, 23];

function fn(ary) {
    console.log(ary);
    ary[0] = 100;
    ary = [100];
    ary[0] = 0;
    console.log(ary);
```

```
}

fn(ary);
console.log(ary);

//私有作用域中没有var
//    形参赋值 ary= 全局下的ary=[12,23]
      // ary[0]=100,改掉全局下的ary[0] 位置的值为100 全局下的ary变成 [100,23]
      // ary =[100];给私有变量的ary了一个地址
      // 修改新地址的值 ary[0]=0
      //console.log(ary);//输出的是新的ary修改后的值

//[12, 23]
//[0]
//[100,23]
```

6.1.8 综合测试

```
var number = 2;
var obj = {
   number : 4,
   fn1 : ( function() {
      this.number *= 2;
      number=number*2;
      var number=3;
      return function() {
         this.number *= 2;
         number*=3;
         console.log(number);
      }

   } )(),
```

```
    db2:function(){this.number*=2}
};

var fn1 = obj.fn1;
console.log(number);//2
fn1();//
obj.fn1();//

console.log(window.number); //
console.log(obj.number);   //
```

6.1.9 柯理化函数（闭包的高级应用）

柯理化函数其实就是一个函数的预处理的作用 【闭包】柯理化函数

> *函数执形成一个私有的作用域，保护里面的私有变量不受外界的干扰，这种保护机制称之为“闭包” 市面上的开发者认为的闭包是：形成一个不销毁的私有作用域（私有栈内存）才是闭包*

6.2 作用域

> *全局作用域 函数作用域（私有作用域）*

【查找上级作用域】

> *当前函数执行，形成一个私有作用域A，A的上级作用域是谁，和他在哪执行的没有关系，和他在哪创建（定义）的有关系，在哪创建的，它的上级作用域就是谁*

【堆栈内存释放】

> *堆内存：存储引用数据类型值（对象：键值对 函数：代码字符串） 栈内存：提供JS代码执行的环境和存储基本类型值*

> *[堆内存释放]*

- 让所有引用堆内存空间地址的变量赋值为null即可（没有变量占用这个堆内存了，浏览器会在空闲的时候把它释放掉）

> *[栈内存释放]*
>
> - *一般情况下，当函数执行完成，所形成的私有作用域（栈内存）都会自动释放掉（在栈内存中存储的值也都会释放掉），但是也有特殊不销毁的情况：*
>
> - *1.函数执行完成，当前形成的栈内存中，某些内容被栈内存以外的变量占用了，此时栈内存不能释放（一旦释放外面找不到原有的内容了）*
> - *2.全局栈内存只有在页面关闭的时候才会被释放掉*
> - *...*
> - *如果当前栈内存没有被释放，那么之前在栈内存中存储的基本值也不会被释放，能够一直保存下来*

```
var i = 2;

function fn() {
   var i = 3;
   return function (n) {
      console.log(n + (++i));
   }
}

var f = fn(4);
f(2);
fn(5)(2);
fn(6)(3);
f(3);
```

Alt text

```
let i = 1;
let fn = function (n) {
   i *= 2;
   return function (m) {
      i += n + m;
      console.log(i);
   }
};
let f = fn(2);
f(3);//=>7
fn(2)(3);//=>19
f(4);//=>25
f(5);//=>32
```

6.3 面向对象

> *面向对象(Object Oriented,OO)是软件开发方法。面向对象的概念和应用已超越了程序设计和软件开发，扩展到如数据库系统、交互式界面、应用结构、应用平台、分布式系统、网络管理结构、CAD技术、人工智能等领域。面向对象是一种对现实世界理解和抽象的方法，是计算机编程技术[1]发展到一定阶段后的产物。*

【早期发展】

> *早期的计算机编程是基于面向过程的方法，例如实现算术运算1+1+2 = 4，通过设计一个算法就可以解决当时的问题。随着计算机技术的不断提高，计算机被用于解决越来越复杂的问题。一切事物皆对象，通过面向对象的方式，将现实世界的事物抽象成对象，现实世界中的关系抽象成类、继承，帮助人们实现对现实世界的抽象与数字建模。通过面向对象的方法，更利于用人理解的方式对复杂系统进行分析、设计与编程。同时，面向对象能有效提高编程的效率，通过封装技术，消息机制可以像搭积木的一样*

> *快速开发出一个全新的系统。面向对象是指一种程序设计范型，同时也是一种程序开发的方法。对象指的是类的集合。它将对象作为程序的基本单元，将程序和数据封装其中，以提高软件的重用性、灵活性和扩展性。[2]*

> *起初，“面向对象”是专指在程序设计中采用封装、继承、多态等设计方法。[2]*

> *面向对象的思想已经涉及到软件开发的各个方面。如，面向对象的分析（OOA，Object Oriented Analysis），面向对象的设计（OOD，Object Oriented Design）、以及我们经常说的面向对象的编程实现（OOP，Object Oriented Programming）。[2]*

> *面向对象的分析根据抽象关键的问题域来分解系统。面向对象的设计是一种提供符号设计系统的面向对象的实现过程，它用非常接近实际领域术语的方法把系统构造成“现实世界”的对象。面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想，这与传统的思想刚好相反：传统的程序设计主张将程序看作一系列函数的集合，或者直接就是一系列对电脑下达的指令。面向对象程序设计中的每一个对象都应该能够接受数据、处理数据并将数据传达给其它对象，因此它们都可以被看作一个小型的“机器”，即对象。*

6.3.1 一个对象的设计模式:单例模式

【基本单例模式】 **对象字面量**

```
var liukai = {
    name:"liukai",
    age:25,
    englishName:"BigSpinach",
    sayHello(){},
    ...
}

//单例设计模式命名的由来
//每一个命名空间都是JS中Object这个内置基类的实例，而实例之间是相互独立互不干扰的，所以我们称它为“单例：单独的实例”
```

【高级单例模式】

> 1. *在给命名空间赋值的时候，不是直接赋值一个对象，而是先执行匿名函数，形成一个私有作用域AA（不销毁的栈内存），在AA中创建一个堆内存，把堆内存地址赋值给命名空间*
> 2. *这种模式的好处：我们完全可以在AA中创造很多内容（变量OR函数），哪些需要供外面调取使用的，我们暴露到返回的对象中（模块化实现的一种思想）*

```
var nameSpace = (function () {
    var n = 12;
    function fn() {
        //...
    }
    function sum() {

    }
    return {
        fn: fn,
        sum: sum
    }
})();
```

【模块化开发】

> *1.团队协作开发的时候，会把产品按照功能板块进行划分，每一个功能板块有专人负责开发 2.把各个版块之间公用的部门进行提取封装，后期在想实现这些功能，直接的调取引用即可（模块封装）*

```
var utils=(function () {
    return {
        aa:function () {

        }
    }
})();

//=>少帅
var skipRender = (function () {
    var fn = function () {
        //...
    };

    //...
```

```
    return {
        init: function () {

        },
        fn:fn
    }
})();
skipRender.init();

//=>敏洁
var weatherRender = (function () {
    var fn = function () {

    };
    return {
        init: function () {
            fn();//=>调取自己模块中的方法直接调取使用即可
            skipRender.fn();//=>调取别人模块中的方法
        }
    }
})();
weatherRender.init();
```

6.3.2 批量生产对象的方式：工厂模式

【工厂模式（Factory Pattern）】

> *1.把实现相同功能的代码进行“封装”，以此来实现“批量生产”（后期想要实现这个功能，我们只需要执行函数即可） 2.“低耦合高内聚”：减少页面中的冗余代码，提高代码的重复使用率*

```
function Factory(){
  var obj = {};//生产原料
  obj.name = '';  //加工过程
  obj.age = '';
  obj.writeCss = function(){ console.log('css')};
  obj.writeJs = function(){ console.log('js')};
  return obj;  //生产出来的产品
}
var a = [];
for(var i = 0;i<30;i++){
   a.push(Factory(i));
}
```

【缺点】

> *生产出来的对象单一，没有特殊性*

6.3.3 有区别的批量生成对象：构造函数

6.3.3.1 【基于构造函数创建自定义类（constructor）】

> 1. *在普通函数执行的基础上“new xxx()”，这样就不是普通函数执行了，而是构造函数执行，当前的函数名称之为“类名”，接收的返回结果是当前类的一个实例*
> 2. *自己创建的类名，最好第一个单词首字母大写*
> 3. *这种构造函数设计模式执行，主要用于组件、类库、插件、框架等的封装，平时编写业务逻辑一般不这样处理*

```
function Fn(name, age){
    this.name = name;
    this.age = age;
    this.writeCss = function(){};
    this.writeJs = function(){};
}

var f1 = new Fn("刘凯",25);var f1 = new Factory("刘凯",25);
var f2 = new Fn("bigspinach",25);
```

6.3.3.2 创建值的两种方式

【JS中创建值有两种方式】

> *1.字面量表达式 2.构造函数模式*

```
var obj = {};//=>字面量方式
var obj = new Object();//=>构造函数模式
//不管是哪一种方式创造出来的都是Object类的实例，而实例之间是独立分开的，所以 var xxx={} 这种模式就是JS中的单例模式
```

【基本数据类型基于两种不同的模式创建出来的值是不一样的】

> *基于字面量方式创建出来的值是基本类型值 基于构造函数创建出来的值是引用类型*

```
//num2是数字类的实例，num1也是数字类的实例，它只是JS表达数字的方式之一，都可以使用数字类提供的属性和方法
var num1 = 12;
var num2 = new Number(12);
console.log(typeof num1);//="number"
console.log(typeof num2);//="object"
```

6.3.3.3 普通函数执行VS构造函数执行

```
function Fn(name, age) {
    var n = 10;
    this.name = name;
    this.age = age + n;
}
```

普通函数执行

```
//1.形成一个私有的作用域
//2.形参赋值
//3.变量提升
//4.代码执行
//5.栈内存释放问题
Fn();
```

构造函数执行

```
var f1 = new Fn('xxx', 20);
var f2 = new Fn('aaa', 30);

console.log(f1 === f2);//=>false：两个不同的实例（两个不同的堆内存地址）
console.log(f1.age);//=>30
console.log(f2.name);//=>'aaa'
console.log("name" in f1);//=>true name&age在两个不同的实例都有存储，但是都是每个实例自己私有的属性
console.log(f1.n);//=>undefined 只有this.xxx=xxx的才和实例有关系,n是私有作用域中的一个私有变量而已（this是当前类的实例）*/
```

> *构造函数执行，不写 `return` ，浏览器会默认返回创建的实例，但是如果我们自己写了 `return` ？*

> 1. return是的一个基本值，返回的结果依然是类的实例，没有受到影响
> 2. 如果返回的是引用值，则会把默认返回的实例覆盖，此时接收到的结果就不在是当前类的实例了

> 所以 构造函数执行的时候，尽量减少RETURN的使用，防止覆盖实例

6.3.4 原型模式

【原型（prototype）、原型链（**proto**）】

> 1. 所有的函数数据类型都天生自带一个属性：prototype（原型），这个属性的值是一个对象，浏览器会默认给它开辟一个堆内存
> 2. 在浏览器给prototype开辟的堆内存中有一个天生自带的属性：constructor，这个属性存储的值是当前函数本身
> 3. 每一个对象都有一个**proto**的属性，这个属性指向当前实例所属类的prototype（如果不能确定它是谁的实例，都是Object的实例）

```
function Fn() {
    var n = 100;
    this.AA = function () {
        console.log(`AA[私]`);
    };
    this.BB = function () {
        console.log(`BB[私]`);
    };
}
Fn.prototype.AA = function () {
    console.log(`AA[公]`);
};

var f1 = new Fn;
var f2 = new Fn;
```

```
console.log(f1.n);
```

```
function Fn() {
    this.x = 100;
    this.y = 200;
    this.getX = function () {
        console.log(this.x);
    }
}
Fn.prototype.getX = function () {
    console.log(this.x);
};
Fn.prototype.getY = function () {
    console.log(this.y);
};
var f1 = new Fn;
var f2 = new Fn;
console.log(f1.getX === f2.getX);//=>false
console.log(f1.getY === f2.getY);//=>true
console.log(f1.__proto__.getY === Fn.prototype.getY);//=>true
console.log(f1.__proto__.getX === f2.getX);//=>false
console.log(f1.getX === Fn.prototype.getX);//=>false
console.log(f1.constructor);//=>Fn
console.log(Fn.prototype.__proto__.constructor);//=>Object
f1.getX();//=>this:f1  =>console.log(f1.x);  =>100
f1.__proto__.getX();//=>this:f1.__proto__  =>console.log(f1.__proto__.x);
=>undefined
f2.getY();//=>this:f2  =>console.log(f2.y); =>200
Fn.prototype.getY();//=>this:Fn.prototype  =>console.log(Fn.prototype.y);
=>undefined
```

6.3.5 js中的继承

6.3.5.1 原型继承

```
//第一种：原型链继承
  // 核心原理： 子类.prototype = new 父类
  // 结果：    子类继承父类所有的属性和方法(公有+私有)
  // 原型链继承的特点： 它把所有父类的公有和私有方法和属性都继承在子类的公有属性上

  function FatherClass(){
    //父类私有的属性
    this.fname = "我是爸爸";
  }

  //父类公有的属性
  FatherClass.prototype.fSay = function(){
    console.log(this.fname);
  }

  //var f = new FatherClass;
  //console.log(f);
  //f.fSay();//我是爸爸

  function SonClass(){
    //子类私有的属性
    this.sname = "我是儿子";
  }
  SonClass.prototype.sSay = function(){
    console.log(this.sname);
  }

  //原型继承核心
  SonClass.prototype = new FatherClass;
  //这就是为了防止上边子类继承父类后，子类的构造函数也变了
  SonClass.prototype.constructor = SonClass;
```

```
var son1 = new SonClass;
console.dir(son1);
```

6.3.5.2 call 继承

【call继承】

> *核心：把父类私有的属性和方法克隆一份一模一样的作为 子类的私有属性*
>
> *call继承的特点：子类只会继承父类的私有属性和方法，如果子类中有和父类私有属性或方法同名的属性或方法，那么子类中的优先，（重写父类的方法，但是不影响父类的本身）*

```
//第二种： call继承
// 核心：把父类私有的属性和方法克隆一份一模一样的作为 子类的私有属性
// call继承的特点： 子类只会继承父类的私有属性和方法，如果子类中有和父类私有属性或方法同名的属性或方法，那么子类中的优先，（重写父类的方法，但是不影响父类的本身）

function FatherClass(){
    this.x = 250;
    this.say = function(){
        console.log(this.x);
    }
}

FatherClass.prototype.getX = function(){
    console.log("我是FatherClass原型上的公有方法getX")
};

function SonClass(){
    //核心代码


    FatherClass.call(this);//把FatherClass执行，并且让FatherClass中的this变为
```

```
当前的this（也就是SonClass的实例）
      this.x = 88888888;
      this.say = function(){
         consoele.log(this.x);
      };
   };

   SonClass.prototype.getX = function(){
      console.log("我是SonClass原型上的公有方法getX")
   };

   var son1 = new SonClass;
   console.log(son1);
   son1.getX();//我是SonClass原型上的公有方法getX

   //call继承的特点： 子类只会继承父类的私有属性和方法，如果子类中有和父类私有属性或方法同名的属性或方法，那么子类中的优先，（重写父类的方法，但是不影响父类的本身）
   var f = new FatherClass;
   console.log(f.x);//250
   f.say();//250
   f.getX();//我是FatherClass原型上的公有方法getX
```

```
function Parent() {
   this.x = 100;
}
Parent.prototype.getX = function () {
   console.log(this.x);
};

function Children() {
   //=>this:child 子类的实例


   Parent.call(this);//=>让Parent执行,方法中的THIS依然是子类的实例（在父类构
```

```
造体中写的 THIS.XXX=XXX 都相当于在给子类的实例增加一些私有的属性和方法)
    this.y = 200;
}

var child = new Children();
```

6.3.5.3 寄生组合继承

> *Object.create(obj)：创建一个空对象（实例），把 `obj` 作为新创建对象的原型*

```
let obj = {name:"BigSpinach"};
let return_object_create = Object.create(obj);
console.log(return_object_create);
/*
    Object
        __proto__:
            name: "BigSpinach"
            __proto__: Object
                constructor: f Object()
                ...
 */
```

寄生组合式继承完成了一个需求：

> *子类公有的继承父类公有的（原型继承的变通） 子类私有的继承父类私有的（call继承完成*

Alt text

```
function Parent() {
```

```
    this.x = 100;
}
Parent.prototype.getX = function () {
    console.log(this.x);
};

function Children() {
    Parent.call(this);
    this.y = 200;
}
Children.prototype = Object.create(Parent.prototype);
Children.prototype.constructor = Children;
Children.prototype.getY = function () {
    console.log(this.y);
};
```

```
function Parent() {
        this.x = 100;
    }
    Parent.prototype.getX = function () {
        console.log(this.x);
    };
    //console.log(Parent);
    //console.dir(Parent.__proto__);
    //console.log(Parent.prototype);
    function Children() {
        Parent.call(this);
        this.y = 200;
    }
    //console.log(Children);
    //console.log(Children.prototype);
    Children.prototype = Object.create(Parent.prototype);
    console.log(Children);

    /*
```

```
ƒ Children() {
    Parent.call(this);
    this.y = 200;
}
 */
console.log(Children.prototype);
/*
Parent {}
    getY: ƒ ()
    __proto__:
        getX: ƒ ()
        constructor: ƒ Parent()
        __proto__: Object
 */


//将Parent.prototype 作为Children.prototype 的原型
//结构是这样子的
//
    /*
        Children
            x:100,
            y:  200,
            __proto__

    */
Children.prototype.constructor = Children;
Children.prototype.getY = function () {
    console.log(this.y);
};
console.log(Children);
console.log(Children.prototype);

let child1 = new Children();

console.log(child1);
```

```
/*
Children {x: 100, y: 200}
   x: 100
   y: 200
   __proto__: Parent
      constructor: ƒ Children()
      getY: ƒ ()
      __proto__:
         getX: ƒ ()
         constructor: ƒ Parent()
         __proto__: Object

 */
```

6.3.5.4 冒充对象继承

> *把父类私有的和父类共有的 克隆一份一模一样的给子类 操作细节： 在子类的构造函数中 遍历父类的实例，增添到子类的共有属性上*

```
//冒充对象继承
//目的： 子类继承的到父类的公有和私有属性
//怎么操作： 子类构造函数中遍历 父类的一个实例对象即可
function Parent() {
   this.x = 100;
}
Parent.prototype.getX = function () {
   console.log(this.x);
};

function Children() {
   var temp = new Parent();
   for(var key in temp){
      this[key] = temp[key];

   }
```

```
    temp = null;
    //子类自己的私有方法
    this.y = 200;
  }

  Children.prototype.constructor = Children;

  var child_1  = new Children();
  console.log(child_1);
  //Children { x: 100, getX: [Function], y: 200 }
```

```
Object.myCreate = function myCreate(obj) {
  var Fn = new Function();
  Fn.prototype = obj;
  return new Fn();
};

var oo = {name: 'oo'};
console.dir(Object.myCreate(oo));
```

6.3.5.5 混合模式继承

```
  //混合模式继承
  //目的:  子类的实例共有属性拿到父类公有属性
  //    子类的实例私有属性拿到父类的私有属性
  function Parent() {
    this.x = 100;
  }
  Parent.prototype.getX = function () {
    console.log(this.x);
  };


  function Children() {
```

```
    Parent.call(this);
    this.y = 200;
};

Children.prototype = new Parent;
Children.prototype.constructor = Children;

var child_one = new Children;
console.log(child_one);
//Children { x: 100, y: 200 }
/*
Children
    x: 100
    y: 200
    __proto__: Parent
        constructor: ƒ Children()
        x: 100
        __proto__: Object

 */
```

6.3.5.6 中间类继承--不兼容

> ***目的***：*其他类想使用别的类的公共方法*
>
> ***做法***: *用过改变其他类的父类指向(一般直接指向要使用的类的原型)，从而达到目的*

> ***原理***： *任何对象的无限多次* `obj.__proto__.__proto__` *最终都会回归基类*

```
//中间类继承

//目的：其他类想使用别的类的公共方法
```

```
//做法: 用过改变其他类的父类指向(一般直接指向要使用的类的原型)，从而达到目的
//原理： 任何对象的无限多次 obj.__proto__.__proto__ 最终都会回归基类

//案例： 实现任意参数求平均数
function avgFn () {
  //核心
  arguments.__proto__ = Array.prototype;

  //从此可以开心的玩耍了
  return (eval(arguments.join("+"))/arguments.length).toFixed(2)
}
console.log(avgFn(1,2,3,4,5,67,8,9));//12.38
```

6.3.5.7 ES6 的class继承

```
class Fn {
  constructor() {
    //=>给实例设置的私有属性
    this.xxx = 'xxx';
  }

  //=>getX & setX：都是给Fn.prototype设置方法
  getX() {  }

  setX() {  }

  //=>把Fn当做一个普通对象，增加的属性和方法
  static property() { }
}
```

```
class A {

  constructor() {
```

```
        this.x = 100;
    }

    getX() {
        console.log(this.x);
    }
}

class B extends A {
    constructor() {
        super();//=>CALL继承
        this.y = 200;
    }

    getY() {
        console.log(this.y);
    }
}

var b = new B();
```

6.3.4

7 JS盒子模型

> *在JS中通过相关的属性可以获取(设置)元素的样式信息,这些属性就是盒子模型属性（基本上都是有关于样式的）*

7.1

8.

9.

10

11