ready

2017年4月24日 9:20

A=A+A=A+A+A

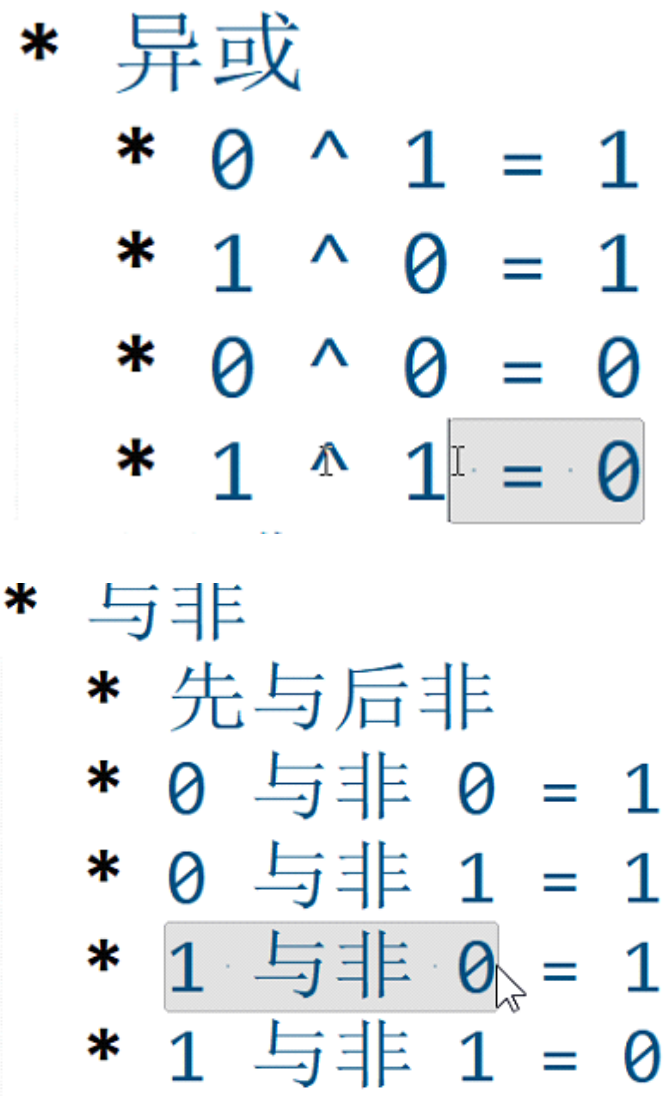
A+1=1

A*1=A

A*0=0

AB+AC=A(B+C)

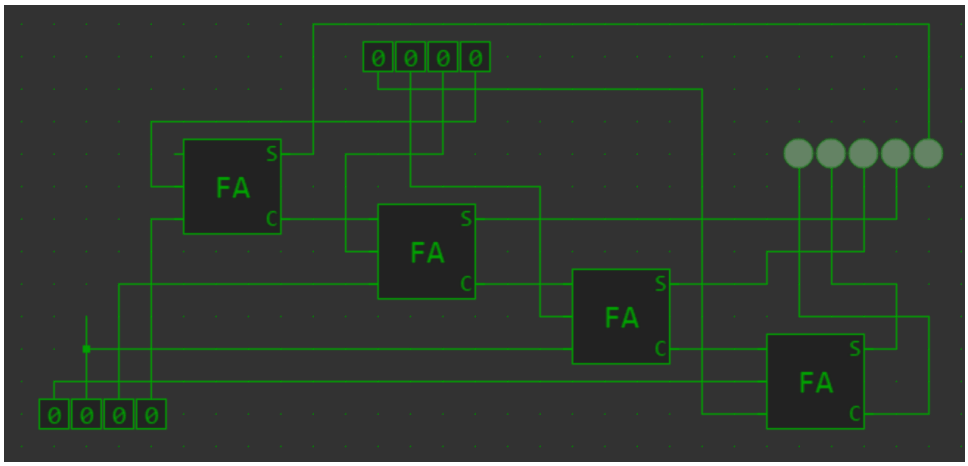
(A+B)(A+C)=A+BC



* 摩根定律/反演律
  * !(A * B) = !A + !B
  * !(A + B) = !A * !B

* 分配律
  * A + (B * C) = (A + B) * (A + C)
  * A * (B + C) = (A * B) + (A * C)

加法进位器https://simulator.io/board



位运算

2017年4月25日　　17:08

整数在计算机中的表示：

最高位为符号位

剩余位为数值位

正数的数值位：原码，首位为0

负数的数值位：原码取反加1，首位为1，即为补码

``反引号字符

| 按位或

&按位与

^按位异或

~按位非（先与后非）

二进制移位运算：

>>按位右移，保留符号位，负数（即有符号）在左边补1，正数补0

>>>按位右移，不保留符号位，左边都补0

<<按位左移

如2>>1 得到 00000001 即为1

-2>>1 得到 11000001 即为-1，需要把-2转换为补码之后移位再转换为原码

2>>>1 得到 00000001 即为1

位运算:

一个字节8位为256种，负数在电脑中用补码表示为，负数到-128，正数到127，还有一个0

正数即到256

如13的原码：00001101

-13的原码 ： 10001101

-13的反码： 11110010

-13的补码： 11110011 即为计算机内部-13的表达值

计算机中只有加法，但是可以把减法转化为加法：若溢出，则直接忽略溢出位

例一：1-2 即为1+（-2）

1为00000001

-2为10000010 原码

11111101 反码

11111110 补码

所以相加为

11111111补码

11111110 减一得到反码
10000001取反得到原码 即为-1

任何数与自己做位异或都将得到0
任何数与0做位异或都将得到自己

例二：1^-1
1为00000001
-1为10000001 原码
11111110 反码
11111111 补码
00000001
^11111111 得到
11111110（补码）
11111101 (减一得到反码)
10000010 取反得到原码 即-2

```
> 1^-1
<- -2
```

判断一个数是否为偶数，只需判断最后一位是否为1，与00000001取与
11110001
&00000001
为 00000001即为奇数

取一个数的前四位，与11110000取与
10110011
& 11110000
为 10110000 即为前四位，但是希望得到00001011
即可以10110000>>4 即可，（不过因为负数为补1，所以会有问题）

写出使用16位二进制表示168与-200时的形式。然后计算它们进行按位与，按位或，按位非，按位异或的结果
168 00000000 10101000
-200 10000000 11001000(原) 11111111 00110111(反) 11111111 00111000(补)
00000000 10101000(168补)
11111111 00111000(-200补)
00000000 00101000(按位与)40
11111111 10111000(按位或)11111111 10110111/10000000 01001000/-72
11111111 10010000(按位异或)11111111 10001111/10000000 01110000/-112
00000000 10101000(168补)
11111111 01010111(按位非)

11111111 00111000(-200补)
00000000 11000111(按位非)

值、类型、操作符

2017年4月26日　　19:33

```
> 0.1+0.2
< 0.30000000000000004
```

计算机中都转化为二进制在计算，二进制转小数即除以2，
最小即为1/2=0.5，以此类推0.25,0.125，0.0625.....
所以小数相加会有精度

分号前置
当一行的第一个字符是 +-/[(` 这几个字符时，他前面一行必须加分号
其余都可以不加
如;(2+8)

特殊数值（Number）：
Infinity　正无穷大　如1/0
-Infinity　负无穷大　如 -1/0
NaN　　not a number　如 0/0　'a'*2

undefined values
null　可以参与数学运算，相当于0
undefined　不能参与运算，否则为NaN
（函数中参数传多被忽略，传少系统默认为undefined）

null==undefined为true
typeof null=="object"
null或undefined与其他比都为false，如null==0 返回false

String
转义：\ '\\' 即为'\'
回车：\n
tab：\t
反引号内部加shift+enter可以直接换行：
·b
a· 输出即为"a
　　b"

字符串比较大小，从左往右一个字符一个字符比较，比较的是其ascii码

运算符：

一元运算符

typeof 4　　输出 number

- 1

二元运算符

+-*/%><=

==　确保两边类型相同时，可以使用，否则js会自动转换类型，如8*null返回为0，null被强制转换为了0

===　全等，不会对两边做类型转换

==! 全不等

三元运算符（ 条件运算符 ）

a?b:c

boolean

除了0为false,其他包括负数也是true

只有NaN自己不等于自己

即NaN==NaN　返回为false

当需要判断一个变量是否为NaN,直接a==a，如果返回为false,a即为NaN

如

0/0==0/0

false

转换类型函数

String(123)

Boolean()

Number()

转换成10进制

parseInt(10,2)　=》 左边的为数，右边为该数为2进制需要转换为10进制， 返回2

5转换为2进制

5..toString(2)　=> “101”

或

(5).toString(2)

或

var a=5

a.toString(2)

十六进制转换为二进制

0x11.toString(2)

0b1111为二进制

0x1111为十六进制

运行时间戳

```
console.time("aaa")
console.timeEnd("aaa")
```

注：浮点数做比较，两个浮点数的绝对值小于某个很小的数，不要直接用等号

```
  name: 'wangzibo',
  age: 25,
  height: 150.03723924,
}].sort(function(a,b){
  if (a.age > b.age) {
    return 1
  }
  if (Math.abs(a.height - b.height) < 0.0001) {
    return 0
  }
  return -1
})
```

math

2017年5月16日 17:07

Math:

Math.floor 向下取整返回小于等于x的最大整数

var a=Math.floor(0.60); 0

var b=Math.floor(0.40); 0

var d=Math.floor(5.1); 5

var e=Math.floor(-5.1); -6

var f=Math.floor(-5.9); -6

Math.ceil 向上取整 对一个数进行上舍入,大于等于他的最小整数

```
var a=Math.ceil(0.60);   1
var b=Math.ceil(0.40);   1
var c=Math.ceil(5);       5
var d=Math.ceil(5.1);     6
var e=Math.ceil(-5.1);   -5
var f=Math.ceil(-5.9);   -5
```

Math.round(2.5) 3 把一个数字四舍五入为最接近的整数

```
Math.trunc(42.84) 42 只保留整数部分
```

Number:

2..toFixed(2) 2.00 保留小数位数

(1234).toExponential(2) 1.23e+3 指数形式，四舍五入

Number.prototype.toLocaleString.call(1233457) 1,233,457

2017年5月3日 12:03

2**6 2的6次方

胖箭头，简单的函数可以如下：当函数体是一个表达式（而不是段落）的时候，可以隐式return

var f=x=>2*x 即为 var f=function(x){return 2*x}
var f=(x,y)=>x+y
var f=(x,y)=>{
  return x+y
}

箭头函数与普通函数的区别
箭头函数无this作用域
无arguments
不能作为构造函数调用
都可以写成异步
箭头函数不能写成生成器函数

若函数只想取参数的某一个属性，可以如下
function a({born,age},{name}){ //第一个参数的born，age属性，第二个参数的name属性
  return born>1990&&age>10&&name=="123"
}
var f=({born})=>born>12

P95页书上例子可以这么写

```
ancestry.filter(({died,born})=>died-born>90).map(({name})=>name)
▶ (3) ["Clara Aernoudts", "Emile Haverbeke", "Maria Haverbeke"]
```

解构 distructing
const [first, ...rest] = [1, 2, 3, 4, 5] // first是1，rest是[2, 3, 4, 5]

var a=([m,...n])=>m===undefined?0:m+a(n) 同[1,2,3].reduce((a,b)=>a+b,0)
a([1,2,3])==>6
{a,b=1}={a:2} ==>a=2,b=1，b为默认值
{a,b:{c}}={a:1,b:{c:1}} ==>c=1
function f([a,b]){console.log(a,b)} ==>f([1,2])
function f(){yield {a:1,b:2}} ==》for(var {a,b} of f())console.log(a,b)

let 同var，只在最近的语句块有效,且不是window的属性
如

```
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

TDZ Temporal Dead Zone
在let赋值语句未结束之前，let的变量都不能访问，不能访问的那块区域叫TDZ

var 只在函数作用域有效
声明的变量会被提升（hoist）
console.log(a)
var a=8
同
var a
console.log(a)
a=8
用let定义则不会，于是在声明之前使用该变量的话会报错
如

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // same variable!
    console.log(x);  // 2
  }
  console.log(x);  // 2
}
```

来自 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

逗号表达式 外可不加括号，返回最后那一项
return stack.pop(), stack

Number.isNaN(NaN) =》true 括号内必须为NaN才为true
IsNaN('abc') =》true

`` 反引号内部可以打回车
`${x+y}` 反引号内部$内可以写表达式

...args 表示把剩余的参数放入args数组，args为数组类型,只能放在最后或只有该项，等同function.apply

注：[...a]=[1,2,3,4] a为有迭代器的类数组对象，可以转换为数组

```
> function test(m,...args){
  console.log(args)
    }
<· undefined
> test(1,2,3,4,5)
  ▶(4) [2, 3, 4, 5]
```

...展开运算符
var a=[1,2,3]
Math.min(...a) ==>1

function.apply等同于...展开运算符
如：Math.max.apply(null,[1,2,3]) ===》Math.max(...[1,2,3])

call apply bind

f.call(this,1,2,3)　　直接调用函数，并设定其内部的**this**，同时依次传入函数的参数

f.apply(this,[1,2,3]) 直接调用函数，但是参数是放在一个数组里传入的，这就意味着其可为被调用函数传入可变量数量的参数

f.bind(this,1,2)(3) 返回函数，绑定this和参数

f.bind(1).bind(2).bind(3)(4,5)===>bind(1,4,5)，this始终为1，后来的bind不会改变this

bind函数实现：(可以看出bind一次之后无法再重新指定this)

```
Function.prototype.bind = function bind(thisValue, ...fixedArgs) {
    var self = this //该this即为调用bind2的函数，保存以返回函数中用
    return function(...restArgs) {
      console.log(this)
      self.call(thisValue, ...fixedArgs, ...restArgs)
    }
 }
```

柯里化：curry 相当于自动bind

```
f1=_.curry(fn)
```

f1(1)(2,3)(4)　===>fn(1,2,3,4)

arguments 所有函数都有的属性，除了箭头函数，所有参数对象

```
function test(){
console.log(arguments)
  }
undefined
test(2,3,4,5,6)
▶(5) [2, 3, 4, 5, 6, callee: function, Symbol(Symbol.iterator): function]
```

属性：callee，其函数自身

坑点：

```
function f(a){
   a={b:1};
   arguments[0].b=9;
}
var obj={b:0}
f(obj)
obj=> {b:0}  //调用时，obj与a解绑之后，argumengts也与obj解绑，a与arguments指向同一个地址，因此obj不会改变
```

```
function f(a){
   "use strict"  //增加严格模式，则修复了这个坑点，最后obj为{b:9}
   a={b:1};
   arguments[0].b=9;
}
```

const 声明常量，不允许重新再赋值

```
> const a=8
< undefined
> a=1
✖ ▶Uncaught TypeError: Assignment to constant variable.
      at <anonymous>:1:2
```

交换位置 [a,b]=[b,a]

```
a=1
1
b=2
2
[a,b]=[b,a]
▶(2) [2, 1]
a
2
b
1
```

集合 Set　无重复项 同lodash中uniq [object Set]

数组去重一般方法：排序然后依次遍历即可

无法增加重复项，也无法增加多项NaN，以对该坑进行修改

var a=new Set()

a.add(1)==>{1}

a.add(1)==>{1}

a.has(1)==>true

a.delete(1)

var a=new Set([1,1,2])

a==>{1,2}

a.size===>2

Map 任何值都可映射，key可为任何值，不像对象那样key值为字符串

a=new Map([[1,2],[3,4]])

a.set(5,6)

a.delete(5)

a.has(5)

a.keys

a.values

a.size

new.target 判断调用构造函数时是否用new

instanceof 会在其原型链上找，所以除了原始类型，其他所有对象都是Object的实例 ， a instanceof A

```
function Complex(real,imag){
  if(new.target!==Complex){//ES6,判断是否是用new来创造的实例
    return new Complex(real,imag)
  }
  if(!this instanceof Complex){
     return new Complex(real,imag)
  }
  this.real=real
  this.imag=imag
}
```

Symbol() 符号类型，为原始类型，一旦生成就唯一，没办法产生跟他相同的

一般用来作为Map或对象的key，该key无法被枚举，可以通过Object.getOwnPropertySymbols得到

也可用来做私有变量

如

```
var obj={}
```

obj[a]=8
obj[a] ===>8

```
> Symbol("symbol2") === Symbol("symbol2");
false
> Symbol.for("symbol2")===Symbol.for("symbol2")
true
>
```

迭代器 [Symbol.iterator] Symbol.iterator 为每一个对象定义了默认的迭代器。该迭代器可以被 for...of 循环结构使用

function*生成函数，蒙太奇

箭头函数不能写成生成函数

生成函数的对象有next方法

自定义number的迭代器函数

```
Number.prototype[Symbol.iterator] = function * (){
 for(var i = 0; i < this; i++) {
      yield i
}}
for(var i of 9) { console.log(i)}
```

此时[...9] ===>[1,2,3....9],...即调用了for of

或如下写法

```
Number.prototype={
  *[Symbol.iterator](){
     for(var i=0;i<this;i++){
    yield i
    }
  },
 }
```

Tagged String f`` 反引号内部为正常字符串，\符号数量必须为偶数，因为其会对其进行转义，但raw属性中保存的是未转义的字符所以可以利用这一点得到输入即输出而不考虑转义

String.raw`\\a` ==>"\\a"

普通：`\\a`==>"\a"

f`aa\\b${a}cc${d}`

f会运行，接收的第一个参数是一个数组

数组的每一项是被插值符分隔出来的若干个字符串

同时这个数组还有一个raw属性，也是一个数组，里面是所有片段里内容未被转义之前的内容

f还会接收若干个剩余的参数，其值与插值符号内的表达式的值一一对应

自己实现String.raw函数，即把接收到的第一个参数里的raw属性连接即得到原字符串

function raw(...arg){return arg[0].raw.join("")}

raw `123\n`

function f(...args){console.log(...args)}

f`123` => ["123", raw: Array(1)]

Proxy 代理

```
var proxyObj=new Proxy({},{
  get(obj,key,proxy){  //proxyObj.a  ary=obj,key=a
    return obj[key]+"11"
  },
  set(obj,key,value,proxy){  //proxyObj.a=2
    obj[key]=value+"lala"
    return obj[key]
  },
  has(obj,key){
    return true
  },
  deleteProperty(obj,key){
    delete obj[key]
    console.log("delete success")
  },
  getPrototypeOf(){ //Object.getPrototypeOf(proxyObj)
    return {a:1}
  }
})
```

对象简写

```
x="test"
y="tt"
obj={
  [x]:function f(){},  ===>obj.test=f
   y,   ===>y:y
}
```

```
a={a:1,b:2,c:3}
b={...a,d:4} =》 b={a:1,b:2,c:3,d:4}   浅复制
```

尾递归，尾循环调用 即可以通过循环优化，不是真正的递归

```
function f(n){
      if(n<0){
            return 0
      }
```

```
    else{
        return 1+f(n-1)  //最后一次执行不需要保存整个作用域环境，算完直接返回，可以转化为循环优化，否则会爆栈
                         //如果换成return f(n-1)+1 则还需要回到之前的作用域，无法优化
}
```

最后执行的代码是函数且不再依赖当前闭包内的任何变量，可以优化为循环

a::b::c Function bind syntax 浏览器暂时不支持，提案阶段

a的结果作为b的this，b的结果作为c的this

```
a=2
f=function(a,b){
  return this+a+b
}
a::f(1,2)  ==>5
```

等同于f.bind(a,1,2)

```
$(".some-link").on("click", ::view.reset);
$(".some-link").on("click", view.reset.bind(view));
```

TypedArray 一个TypedArray 对象描述一个底层的二进制数据缓存区的一个类似数组(array-like)视图

具体指

Unit8Array

Unit16Array

共同拥有的 ArrayBuffer

分配一个8位的内存片段，把它以一个8位整数读取

```
a=new ArrayBuffer(8)
b=new Int8Array(a)
```

```
> a=new ArrayBuffer(8)
◂ ▶ ArrayBuffer {}
> b=new Int8Array(a)
◂ ▶ Int8Array(8) [0, 0, 0, 0, 0, 0, 0, 0]
> c=new Int16Array(a)
◂ ▶ Int16Array(4) [0, 0, 0, 0]
> b[0]
◂ 0
> b[0]=20
◂ 20
> b
◂ ▶ Int8Array(8) [20, 0, 0, 0, 0, 0, 0, 0]
> c
◂ ▶ Int16Array(4) [20, 0, 0, 0]
> |
```

Number.EPSILON 用来判断两个浮点数的最小差值

export import

a.js

```
export default let a=8
export default let b=8
setTimeout(()=>a=10,1000)
```

b.js

```
import a from '/a.js'
console.log(a)   ==>a=8
```

setTimeout(()=console.log(a),2000) ==>a=10

Tree shaking 压缩时会把不用的依赖树去掉，如export default let b=8这句会被去掉，因为没有引用

Object.assign 对象浅复制 所有可枚举的自有属性

Object.assign({},{a:1,b:2},{c:3}) ===》{a: 1, b: 2, c: 3}

2017年5月4日 16:08

变量的读取比读属性要快很多

数组

var a=[]

var a=new Array(8)

var a=[1,"abc",function m(){return 8},[1,2,3]]

数组内部可以放函数，数组，字符串，等任意值

a.push() 往数组后插入

a.pop() 弹出最后的值，数组长度减一

a.unshift() 往数组前插入

a.shift() 往数组前删除

a.indexOf(1,1) 在数组中找1，从下标为一开始找；同样可用于字符串，a.indexOf("ed") 在字符串中找ed

a.fill(true)数组都填充true 注：不能fill对象，否则数组所有项都指向同一个对象，而不是new的，如a.fill([])

```
arr.fill(value)
arr.fill(value, start)
arr.fill(value, start, end)
```

来自 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/fill?v=control>

注：数组长度会随着设置长度或者增加一项而改变，中间若有空余项则自动填充undefined,如

```
var fruits = [];
fruits.push('banana', 'apple', 'peach');
3
fruits.length=10
10
fruits
▶ (10) ["banana", "apple", "peach", undefined × 7]
Object.keys(fruits)
▶ (3) ["0", "1", "2"]
fruits[5]="orange"
"orange"
fruits
▶ (10) ["banana", "apple", "peach", undefined × 2, "orange", undefined × 4]
```

slice 数组切割 返回新数组，若参数为负数，则加上数组的长度直到为正数即是相对项

a.slice(start) 从start位置开始切割

a.slice(start,end) 包括start ,不包括end

生成的新数组（若数组中有对象）还是指向原来的对象，是浅拷贝

isEqual是深对比，a===b是浅对比

深对比 deep copy，对比值

浅对比 shallow copy ，对比指针，在内存在是否为同一地址

深复制：

```
var deepCopy= function(source) {
        var result={};
        for (var key in source) {
          result[key] = typeof source[key]===‘ object‘ ? deepCoyp(source[key]): source[key];
        }
    return result;
}
```

来自 <http://www.cnblogs.com/yichengbo/archive/2014/07/10/3835882.html>

splice 数组项删除，返回删除项

a.splice(start,length,newvalue) 从start位置开始删除length长度项，替换为newvalue

a.splice(2,1,-1) ==>删除下标为2的项，删除一项，替换为-1
a.splice(2,1,-1,-2) ==>删除下标为2的项，删除一项，替换为-1,-2

var a=[1,2,3,4]
a.splice(1) =>从1的位置开始删除，返回删除项[2, 3, 4]，a变成[1]
a.splice(-1) ==>[4]，-1项即从最后开始找

如何把arguments对象变成数组
Array.from(arguments)
[].slice.call(arguments)
[].splice.call(arguments)
[...arguments]
[].map.call(arguments,it=>it)

concat 数组连接 返回新数组,相对于对原来的数组做了一次浅复制
可接收多个参数，参数可以为数组（相当于数组降一次维度塞进新数组）
var c=a.concat(b)

```
> [2].concat([3],[4,[5]],[6])
< ▶(5) [2, 3, 4, Array(1), 6]
```

Sort 排序 默认按字符排序
有一个可选参数，是回调函数，函数应以两个数组元素为参数，两个参数相等时返回0，第一个大返回正数即排在后面，第二个大返回负数

```
function customSort(){
    if(a>b)return 1
    if(a<b)return -1
    return 0
}
var numbers = [4, 2, 5, 1, 3];
numbers.sort(function(a, b) {
  return a - b;     //小于0，a排在b前面，大于0，b排在a前面，等于0，不换位置
});
 //[1, 2, 3, 4, 5]
numbers.sort(function(a, b) {
  return b - a;
});
 //[5,4,3,2,1]

var items = [
  { name: 'Edward', value: 21 },
  { name: 'Sharpe', value: 37 },
  { name: 'And', value: 45 },
  { name: 'The', value: -12 },
  { name: 'Magnetic', value: 13 },
  { name: 'Zeros', value: 37 }
];
// sort by value
items.sort(function (a, b) {
  return a.value - b.value;
});
```

排序算法的稳定性，不改变一开始就有序的数值顺序，如[3,2,2,1]，排序后中间两个2未交换顺序则稳定
稳定性一般在多级排序时比较有用，第二级排序是在第一级的基础上
插入排序稳定排序，因为移动是整片移动，不会交换位置
火狐下sort为稳定排序
谷歌下
数量17以内为插入排序，大于17为快速排序，快速排序是否稳定即取决于浏览器的实现，也取决于数据的顺序

例子，火狐下可直接用，数据量少时谷歌里也OK
var list=[{math:1,sum:2},{math:2,sum:5},{math:3,sum:6},{math:3,sum:5}]
list.sort((a,b)=>b.math-a.math)
.sort((a,b)=>b.sum-a.sum) 先按总分排，再按数学排，优先级最高的在最下面

注：浮点数做比较，两个浮点数的绝对值小于某个很小的数，不要直接用等号

```
  name: 'wangzibo',
  age: 25,
  height: 150.03723924,
}].sort(function(a,b){
  if (a.age > b.age) {
    return 1
  }
  if (Math.abs(a.height - b.height) < 0.0001) {
    return 0
  }
  return -1
})
```

Array基本函数

[[1,2,3],[4,5]].toString() ===》 "1,2,3,4,5" 将 Array 的每个元素转换为字符串，并将它们依次连接起来，两个元素之间用英文逗号作为分隔符进行拼接。

a.reverse()

Array.isArray(a) true

```
Array.from('foo'); 把类似数组的对象转换为数组
// ["f", "o", "o"]

function f() {
  return Array.from(arguments);
}
f(1, 2, 3);// [1, 2, 3]

Array.from([1, 2, 3], x => x + x);// [2, 4, 6]

Array.from({1:'a',2:'b',3:'c',0:'x',length:4})
▶ (4) ["x", "a", "b", "c"]
```

用slice.call实现将类数组转换为数组
var a={0:1,1:2,length:2}
[].slice.call(a)

```
Array.of 把参数放到数组里
Array.of(1);         // [1]
Array.of(1, 2, 3);   // [1, 2, 3]
```

Array迭代函数

迭代函数基本都可以传三个参数val,index,ary本身，包括map,filter,forEach等， 如

```
[1,2,3].forEach(function(val,index,ary){console.log(ary,index,val)})
▶ (3) [1, 2, 3] 0 1
▶ (3) [1, 2, 3] 1 2
▶ (3) [1, 2, 3] 2 3
```

a.some(cb,thisArg)　数组里的值是否有满足函数条件

```
[2, 5, 8, 1, 4].some(x => x > 10);  // false
[12, 5, 8, 1, 4].some(x => x > 10); // true
```

a.every(cb,thisArg)　所有的都满足

```
[12, 5, 8, 130, 44].every(x => x >= 10); // false
[12, 54, 18, 130, 44].every(x => x >= 10); // true
```

a.entries() 返回新数组对象，键值对

如

```
var a = ['a', 'b', 'c'];
var iterator = a.entries();
for (let e of iterator) {
  console.log(e);
}
// [0, 'a']
// [1, 'b']
// [2, 'c']
```

a.forEach(cb,thisArg)

如

```
var words = ['one', 'two', 'three', 'four'];
words.forEach(function(word) {
  console.log(word);
  if (word === 'two') {
    words.shift();
  }
});
// one
// two
// four
```

a.map(cb,thisArg) 每个值调用传入的函数，建立新的映射关系，返回新数组，注：原数组若有undefined值，map到那项无效

```
var numbers = [1, 4, 9];
var roots = numbers.map(Math.sqrt);
// roots is now [1, 2, 3]
// numbers is still [1, 4, 9]

['1', '2', '3'].map(Number); // [1, 2, 3]
```

['1','2','3'].map(parseInt) =>[1, NaN, NaN]

注：因为parseInt需要两个参数，还有一个参数为下标

即parseInt("1",0),parseInt("2",1),parseInt("3",2)，后面两个失效 1作为0进制来转为十进制即1,2作为1进制来转为NaN...

```
var kvArray = [{key: 1, value: 10},
               {key: 2, value: 20},
               {key: 3, value: 30}];
var reformattedArray = kvArray.map(function(obj) {
   var rObj = {};
   rObj[obj.key] = obj.value;
   return rObj;
});

// reformattedArray is now [{1: 10}, {2: 20}, {3: 30}],

// kvArray is still:
// [{key: 1, value: 10},
//  {key: 2, value: 20},
//  {key: 3, value: 30}]
```

a.filter(cb,thisArg)

```
var a = ['a', 'b', 'c'];
var b=a.filter(function(obj){return obj==="c"})
```

```
b => ["c"]
```

a.reduce(cb,start)　cb(res,item,i,ary)　reduceRight

从传入的初始值（未传初始值则从数组第一项开始，数组至少两项，否则出错）开始与数组的第一个值开始调用函数，结果继续传给函数的第一个值，然后与数组的下一个值继续计算

```
var total = [ 0, 1, 2, 3 ].reduce(( acc, cur ) => acc + cur,0);=>6
```

```
var flattened = [[0, 1], [2, 3], [4, 5]].reduce(
  function(a, b) {
    return a.concat(b);
  },
  []
);
// flattened is [0, 1, 2, 3, 4, 5]
```

来自 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every?v=control>

展平数组

```
a=(ary)=>ary.reduce((res,item)=>{return res.concat(Array.isArray(item)?a(item):item)},[])
a([[1,2,3],[4,5,[6,7]]]) ===》 [1, 2, 3, 4, 5, 6, 7]
```

或

```
[[1,2,3],[4,5,[6,7]]].toString().split(",").map(item=>+item)  ===》 [1, 2, 3, 4, 5, 6, 7]
```

2017年5月15日 11:34

对象/关联数组

var m={"a":1,"b":2,"c":3,} var m={a:1,b:2,c:3,}

读取a属性

m["a"]

m.a

a={m:1}

a.n=2 =>{m:1,n:2} 增加属性

o={a(m){return m},"b":1}

对象中的函数可以直接简写为a(m){}，a为属性名称，o.a返回的函数没有名称

delete a.m => a={} 删除对象

'm' in a ==>true ,判断属性是否在对象里，注：in会检查属性是否在其原型链上

'toString' in a==> true ,对象自带的方法

{a:1}.a ==>VM328:1 Uncaught SyntaxError: Unexpected token .

浏览器读到“{”以为后面为语句块，如if(){}的大括号一样，读到“}”后语句块结束，然后出现个 "."，浏览器会认为前面没有对象

({a:1}).a ==>1， “(”里认为本身是个表达式，所以浏览器把他当对象处理，

如 (function a(x){console.log(x)})(3) ==》3

var a=function(x){return x}(3) ==>3， 此时a右边的为表达式，不在需要括号

任意两个不同的对象都是不同，除非a=b赋值，指向同一个对象，改变b的同时a也会发生改变

但是a={},b={} a===b ==> false,此时不同，他们指向的不是同一个地址

注：

对象没有length属性

对象的属性和值都可以写为表达式

对象可以在数组里,如[{x:2},{y:2}]

数组与对象没有本质区别，数组用数字编号，对象用字符串编号，一个是中括号，一个是大括号。

数组与对象的属性名称只能是字符串，虽然数组里index为数字，但运行时也是把它转成字符串之后再输出，如a[0]=1 ，a['0']=1，对象也同

```
> function f(a,b){console.log()}
<· undefined
> f.name
<· "f"
> f.length
<· 2
> var m=function(a,b){console.log()}
<· undefined
> m.length
<· 2
> m.name
<· "m"
```

length是参数个数，name属性是函数的名字

对象对比

基本数据类型按值对比（number,string,boolen）

对象按内存中的位置对比（object,array）

Object.assign 对象浅复制 所有可枚举的自有属性

Object.assign({},{a:1,b:2},{c:3}) ===》{a: 1, b: 2, c: 3}

isEqual是深对比，a===b是浅对比

深对比 deep copy，对比值

浅对比 shallow copy ，对比指针，在内存在是否为同一地址

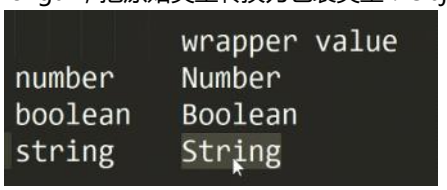
深复制：

```
var deepCopy= function(source) {
        var result=Array.isArray(source)?[]:{};
        for (var key in source) {
          if(Object.prototype.hasOwnProperty.call(source,key)){//只拷贝对象自身的属性
            result[key] = typeof source[key]==='object'? deepCopy(source[key]): source[key];
        }
    }
    return result;
}
```

或者(前提：JSON安全对象，即内部无死循环)

```
newObj=JSON.parse(JSON.stringify(obj))
```

基本类型与包装类型（对象），基本类型本没有属性可言，读取基本类型的一些属性时，浏览器自动把它变成包装类型再进行读取，如length，把原始类型转换为包装类型：Object("123"), 包装类型转换为原始类型 ：Object("123").valueOf()



valueOf 返回对象原始值

默认情况下, valueOf() 会被每个对象Object继承。每一个内置对象都会覆盖这个方法，为了返回一个合理的值，如果对象没有原始值，valueOf() 就会返回对象自身

自定义对象的valueOf函数：

```
function myNumberType(n) {
    this.number = n;
}
myNumberType.prototype.valueOf = function() {
    return this.number;
};

myObj = new myNumberType(4);
myObj + 3; // 7
```

传参只是把参数等于那个传的值，此时两者指向同一个对象，除非其中一个解绑，即重新定义，

如

```
function f(o){  --传参即o=c
  o.x=1     --c.x=1
  o={}      --此时o与c已解绑
  o.y=2
}
var c={}
f(c)
```

c => Object {x: 1}

遍历对象的属性，可枚举属性（包括原型链上）

```
var a={a:1,b:2,c:3}
for(key in a){
  console.log(key,a[key])
}
```

输出：

a 1

b 2

c 3

循环数组：(es6) 原理：向被访问对象请求一个迭代器【Symbol.iterator】对象，然后通过调用迭代器对象的next方法来遍历所有返回值

```
for(value of [1,2,3]){
 console.log(value)
}
```

=>> 1 2 3

获取对象属性列表与值列表

```
a={a:1,b:2}
Object.keys(a)    => ["a", "b"]  只获取自身的key,同Object.getOwnProperty
Object.values(a) => [1, 2]
```

Map

与对象的区别

一个对象的键只能是字符串或者 Symbols，但一个 Map 的键可以是**任意值。**

你可以通过size属性很容易地得到一个Map**的键值对个数**，而对象的键值对个数只能手动确认。

```
var myMap = new Map();
myMap.set(0, 'zero');
myMap.set(1, 'one');
for (var [key, value] of myMap) {
  console.log(key + ' = ' + value);
}
// 0 = zero
// 1 = one
for (var key of myMap.keys()) {
  console.log(key);
}
// 0
// 1

for (var value of myMap.values()) {
  console.log(value);
}
// zero
// one

for (var [key, value] of myMap.entries()) {
  console.log(key + ' = ' + value);
}
// 0 = zero
// 1 = one

myMap.get('key1')  => 返回value
```

来自 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map>

2017年5月31日 20:55

__proto__ ，es6中规范为Object.getPrototypeOf

所有对象都有__proto__属性，

obj.__proto__等同于Object.getPrototypeOf(obj)

意为obj的构造函数，即其原型

如下：

```
Number.__proto__ === Function.prototype  // true
Boolean.__proto__ === Function.prototype // true
String.__proto__ === Function.prototype  // true
Object.__proto__ === Function.prototype  // true
Function.__proto__ === Function.prototype // true
Array.__proto__ === Function.prototype   // true
RegExp.__proto__ === Function.prototype  // true
Error.__proto__ === Function.prototype   // true
Date.__proto__ === Function.prototype    // true
Math.__proto__ === Object.prototype  // true
JSON.__proto__ === Object.prototype  // true
```

Object.prototype为最根部的构建函数，其不再有原型

Object.prototype.__proto__===null //true

来自 <http://www.cnblogs.com/snandy/archive/2012/09/01/2664134.html>

原型链：任何时候，若在对象本身无法找到某属性，则会在其__proto__属性上查找，找不到则继续在其__proto__的__proto__上查找

每个对象都有属于自己的构造器属性，其所引用的就是用于创建该对象的那个函数，在对象自身上找不到某个属性时，就会到用于创建当前对象的构造器函数的原型上去找，依此类推

即obj.name === obj.constructor.prototype.name === obj.__proto__.name

(注：若重置对象的prototype之后，constructor属性不可靠)

```
//重写对象的prototype时，需要重置相应的constructor属性
 var monkey={a:1,b:2}
 function m(){}
 m.prototype=monkey

 c=new m()
 c.a==1
 typeof c.contructor === function Object(){[native code]}
 c.constructor.prototype.hasOwnProperty("a")===false
//需要重置
 m.prototype.constructor=m
 c.contructor === m
 c.constructor.prototype.hasOwnProperty("a")===true
```

prototype与__proto__有什么区别和联系

prototype是在函数上的(obj.constructor.prototype)，__proto__是在对象上的(obj.__proto__)

其实没有任何联系

只不过由构造函数创建出来的实例的__proto__会指向构造函数的prototype属性

a.__proto__=A.prototype

prototype 原型对象

函数的属性，即意为其为构造函数，所有对象都是由其产生，由该构造函数生成的对象拥有构造函数里的所有对象和属性

自定义的构造函数

当一个函数用new操作符作为构造函数来调用时，在函数内部，this初始值指向一个空对象这个空对象以构造函数的prototype属性作为其原型。

New的工作流程

1、创建一个新对象
2、将构造函数的作用域赋给新对象（因此this就指向了这个新对象）
3、执行构造函数中的代码（为这个新对象添加属性）
4、返回新对象

```
function new1(f,...arg){
   var obj=Object.create(f.prototype)
   var res=f.call(obj,...arg)
   if(res!=undefined&&res!=null){
     return res
   }
   return obj
}
```

例子

```
function Person() {}
console.log(Person.__proto__ === Function.prototype)   //true
console.log(Person.prototype.__proto__ === Object.prototype) //true
```

```
Person.prototype
▼Object {constructor: function}
  ▶constructor: function Person()
  ▶__proto__: Object
p=new Person()
▶Person {}
p.__proto__==Person.prototype
true
p.__proto__
▶Object {constructor: function}
```

每个构造函数自带prototype属性，prototype属性里都有constructor属性，这个属性即指向函数自己
如：

```
function Person() {}
undefined
Person.prototype
▶Object {constructor: function}
Person.prototype.constructor
function Person() {}
Person.prototype.constructor===Person
true
```

a instanceof Foo:在a的整条原型链中是否有指向Foo.prototype的对象，左边为对象，右边为函数

b.isPrototypeOf(a) 在a的整条原型链中是否出现过b ，用来判断两个对象是否通过原型关联

Object.create Object.setPrototypeOf 设置对象的原型
var a=Object.create(obj) --以obj为原型建对象，同a.__proto__=obj
或者 Object.setPrototypeOf(a,obj)

Object.create(null) --没有原型的对象，即其属性都为自身的属性，可以不需要再担心for in的时候出来曾经在原型上增加的属性

object.definePrоperty() 设置对象属性是否可枚举等（需要配置这些的时候才使用，否则直接把属性挂在对象上即可）
object.definePrоperty(obj,property,{enumerable:false,value:"hi"})为obj设置属性property,不可枚举(即不能在for in中获取)，值为hi
Object.defineProperty(a,"prop",{deleteable:false,value:123}) 为a对象设置其prop属性不可删除

```
Object.defineProperty(obj,'propName', {
```

```
  get:
  set:
  value:
  writable:
  enumerable:
  configurable:
})
```

Object.preventExtensions(*obj*) 不允许增加新属性，但是原型上还是可以增加

Object.defineProperties() 设置对象多个属性

```
Object.defineProperties(Vector.prototype,{
    length:{
      set(v){
        this.x=v.x
        this.y=v.y
      },
      get(){
        return this.x**2+this.y**2
      }
    },
    X:{
      set(x){
        this.x=x
      },
      get(){
        return this.x
      }
    },
```

Object.getOwnPropertyDescriptor(*obj*, *prop*)　属性描述符 如getter/setter函数，是否可写，是否可以枚举，是否可配置

返回指定对象上一个自有属性对应的属性描述符。（自有属性指的是直接赋予该对象的属性，不需要从原型链上进行查找的属性）

```
Object.getOwnPropertyDescriptor(a,"mm")
▶{value: 6, writable: false, enumerable: false, configurable: false}
```

数据描述符：

defineProperty时，默认值都为false，所以需要显式的设他们为true；否则一般对象如{m:1}的属性的描述符都是true

writable

enumerable

configurable：是否可配置，是否可删除

即如果第一次用defineProperty设置为false,之后就无法再对该属性重新defineProperty，是单向的

例外：即便configurable为false,writable状态可由true改为false,但是无法由false改为true

访问描述符：有getter/setter的属性

obj.propertyIsEnumerable("a") 判断自有属性是否可枚举

Object.preventExtensions(obj) 禁止扩展对象，不能添加新属性

Object.seal(obj)　密封对象，不能添加新属性，也不能重新配置或删除，可以修改；即在禁止扩展的基础上把configurable:false

Object.freeze(obj) 冻结对象，在密封的基础上，writable:false

注：只对基础数据类型的对象属性有效，对引用类型的对象（如value是对象）不受影响，要想深度冻结，可以遍历他引用的所有对象上调用freeze

hasOwnProperty 对象自己身上的属性，非原型上

注：in会检查属性是否在其原型链上

obj.hasOwnProperty("toString") //false
一般调用 Object.prototype.hasOwnProperty.call(object,key) ,以防止obj自身就有该属性或者没有该属性（Object.create(null)）时调用会出错

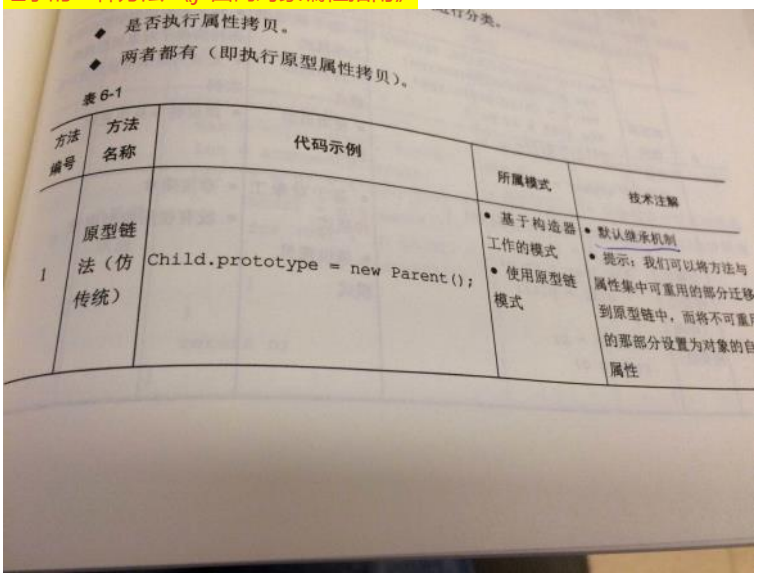
Object.keys 获取自身可枚举的key
Object.keys({a:1,b:2}) => ["a", "b"]

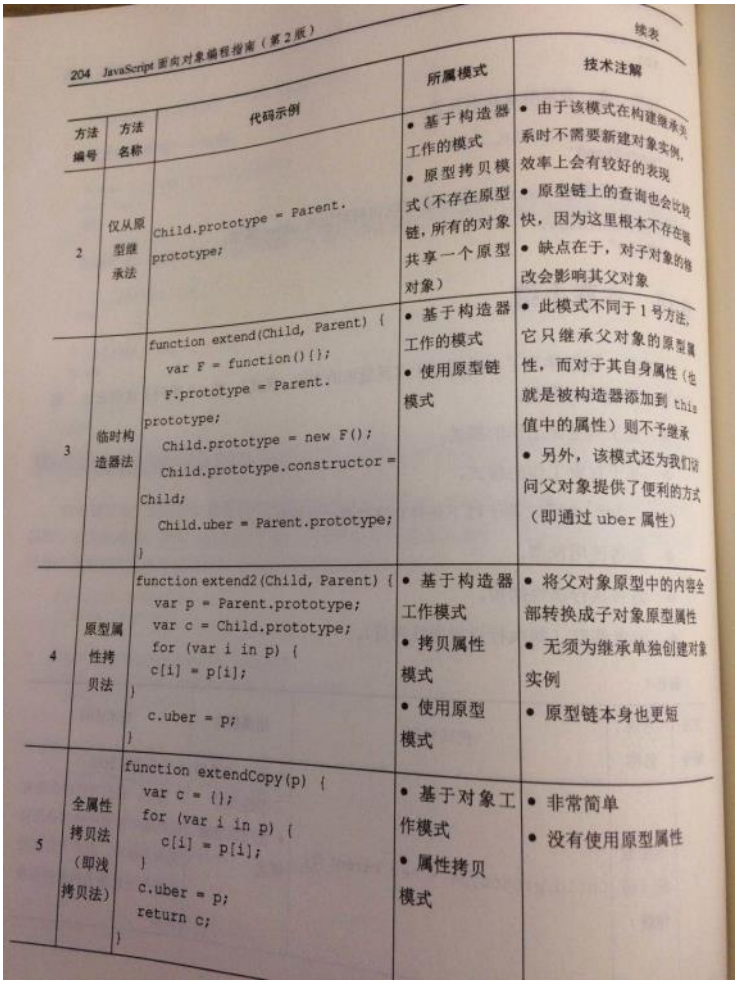
Object.getOwnPropertyNames 获取自身属性，不论属性是否可枚举

扩展
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object

js中私有变量或函数，利用闭包或约定 this._a=a
创建一个闭包：从一个函数A内返回一个函数B,只有这个函数B可以访问到A内的变量，返回闭包函数，这样外部就访问不到那个变量
如下，private里面可以访问foo

继承的12种方法 《js面向对象编程指南》





204 JavaScript 面向对象编程指南（第2版）

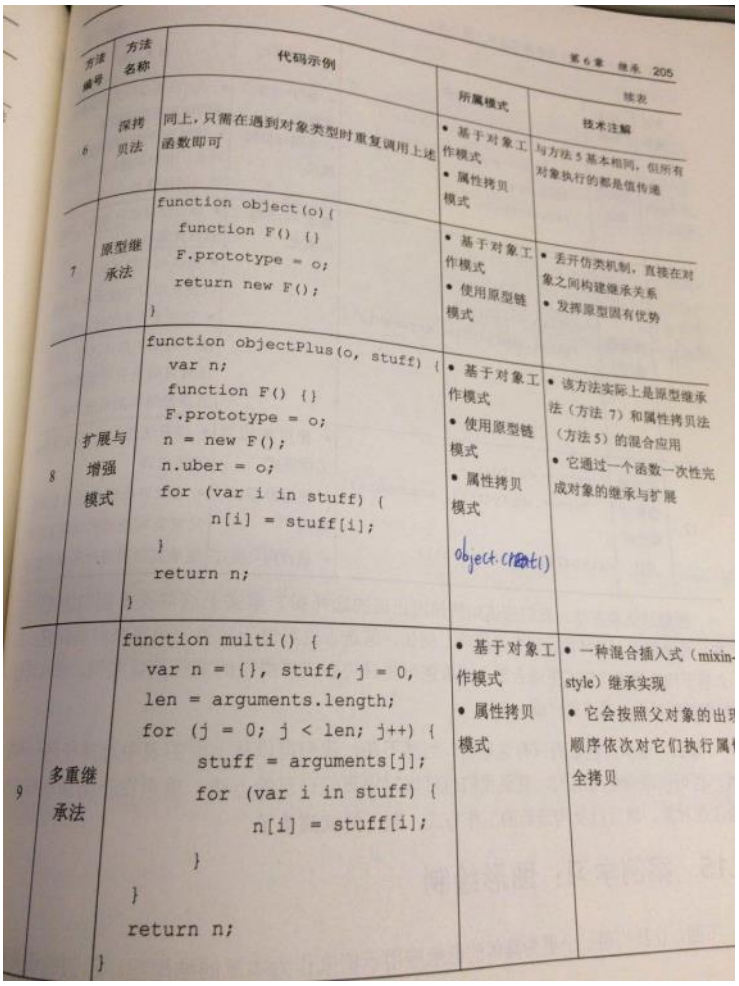
续表

方法编号	方法名称	代码示例	所属模式	技术注解
2	仅从原型继承法	`Child.prototype = Parent.prototype;`	• 基于构造器工作的模式 • 原型拷贝模式（不存在原型链，所有的对象共享一个原型对象）	• 由于该模式在构建继承关系时不需要新建对象实例，效率上会有较好的表现 • 原型链上的查询也会比较快，因为这里根本不存在链 • 缺点在于，对子对象的修改会影响其父对象

3	临时构造器法	`function extend(Child, Parent) {`
		`  var F = function(){};`
		`  F.prototype = Parent.prototype;`
		`  Child.prototype = new F();`
		`  Child.prototype.constructor = Child;`
		`  Child.uber = Parent.prototype;`
		`}`
			• 基于构造器工作的模式
			• 使用原型链模式
				• 此模式不同于1号方法，它只继承父对象的原型属性，而对于其自身属性（也就是被构造器添加到 this 值中的属性）则不予继承
				• 另外，该模式还为我们访问父对象提供了便利的方式（即通过 uber 属性）

4	原型属性拷贝法	`function extend2(Child, Parent) {`
		`  var p = Parent.prototype;`
		`  var c = Child.prototype;`
		`  for (var i in p) {`
		`  c[i] = p[i];`
		`  }`
		`  c.uber = p;`
		`}`
			• 基于构造器工作模式
			• 拷贝属性模式
			• 使用原型模式
				• 将父对象原型中的内容全部转换成子对象原型属性
				• 无须为继承单独创建对象实例
				• 原型链本身也更短

5	全属性拷贝法（即浅拷贝法）	`function extendCopy(p) {`
		`  var c = {};`
		`  for (var i in p) {`
		`  c[i] = p[i];`
		`  }`
		`  c.uber = p;`
		`  return c;`
		`}`
			• 基于对象工作模式
			• 属性拷贝模式
				• 非常简单
				• 没有使用原型属性



第6章 继承 205

续表

方法编号	方法名称	代码示例	所属模式	技术注解
6	深拷贝法	同上，只需在遇到对象类型时重复调用上述函数即可	• 基于对象工作模式 • 属性拷贝模式	与方法5基本相同，但所有对象执行的都是值传递

7	原型继承法	`function object(o){`
		`  function F() {}`
		`  F.prototype = o;`
		`  return new F();`
		`}`
			• 基于对象工作模式
			• 使用原型链模式
				• 丢开仿类机制，直接在对象之间构建继承关系
				• 发挥原型固有优势

8	扩展与增强模式	`function objectPlus(o, stuff) {`
		`  var n;`
		`  function F() {}`
		`  F.prototype = o;`
		`  n = new F();`
		`  n.uber = o;`
		`  for (var i in stuff) {`
		`  n[i] = stuff[i];`
		`  }`
		`  return n;`
		`}`
			• 基于对象工作模式
			• 使用原型链模式
			• 属性拷贝模式
			object.create()
				• 该方法实际上是原型继承法（方法7）和属性拷贝法（方法5）的混合应用
				• 它通过一个函数一次性完成对象的继承与扩展

9	多重继承法	`function multi() {`
		`  var n = {}, stuff, j = 0,`
		`  len = arguments.length;`
		`  for (j = 0; j < len; j++) {`
		`  stuff = arguments[j];`
		`  for (var i in stuff) {`
		`  n[i] = stuff[i];`
		`  }`
		`  }`
		`  return n;`
		`}`
			• 基于对象工作模式
			• 属性拷贝模式
				• 一种混合插入式（mixin-style）继承实现
				• 它会按照父对象的出现顺序依次对它们执行属性全拷贝

续表

方法编号	方法名称	代码示例	所属模式	技术注解

10	寄生继承法	function parasite(victim) {
		  var that = object(victim);
		  that.more = 1;
		  return that;
		}
			• 基于对象工作模式
			• 使用原型链模式
				• 该方法通过一个类似构造器的函数来创建对象
				• 该函数会执行相应的对象拷贝，并对其进行扩展，然后返回该拷贝

11	构造器借用法	function Child() {
		  Parent.apply(this, arguments);
		}
			基于构造器工作模式	• 该方法可以只继承父对象的自身属性
				• 可以与方法1结合使用，以便从原型中继承相关内容
				• 它便于我们的子对象继承某个对象的具体属性（并且还有可能是引用类属性）时，选择最简单的处理方式

12	构造器借用与属性拷贝法	function Child() {
		  Parent.apply(this, arguments);
		}
		extend2(Child, Parent);
			• 使用构造器工作模式
			• 使用原型链模式
			• 属性拷贝模式
				• 该方法是方法11与方法4的结合体
				• 它允许我们在不重复调用父对象构造器的情况下同时继承其自身属性和原型属性

面对这么多方法，我们应该如何做出正确的选择呢？事实上这取决于我们的设计风、性能需求、具体项目任务及团队。例如，您是否更习惯于从类的角度来解决问题？那基于构造器工作模式更适合您。或者您可能只关心该“类”的某些具体实例，那么可能用基于对象的模式更合适。

那么，继承实现是否只有这些呢？当然不是，我们可以从上面的表中选择任何一种模也可以混合使用它们，甚至我们也可以写出我们自己的方法。重点在于必须理解并熟对象、原型以及构造器的工作方式，剩下的就简单了。

案例学习：图形

2017年5月9日 16:32

'a'.charCodeAt() 97
String.fromCharCode(97) a
"abc".charAt(0) a
"abc"[0] => a
[1,2,3,4].join(",") =>"1,2,3,4"
"1,2,3,4".split(",")=>["1","2","3","4"]
"1,2,3,4".split(",",2)=>["1","2"] 第二个参数表示数组长度,第一个参数也可以为正则表达式
"abced".indexOf("ed") 在字符串中找ed
"123".replace("1","2") =>"223"
"123".concat("456") =>"123456"
"123".includes("12") =>true
"123".endsWith("23") =>true startsWith()
"123".repeat(2) =>"123123"
"123".substr(1,2) =>"23" substr(start,length)
toLowerCase()
.toUpperCase()
trim()

"abc".length==="abc"["length"] =>3

字符串用作为数组时为只读，赋值无效
如var a="abc" a[0]=b无效，返回a还是"abc"

document.write()是生成html到页面

执行字符串 eval(string)
eval("1+1")===>2

```
a=5
function f(){
  var a=4
  return eval("a+a") //8,作用于局部
}
globalEval=eval
```

globalEval("a+a") //10,作用于全局

严格模式下eval不再为当前所处代码块的范围引入新变量，或改变变量，即其自生产生闭包

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x");
console.assert(x === 17); true
console.assert(evalX === 42);
```

执行字符串 new Function

前面的参数为形参，最后一个为函数内代码，要求都用字符串

f=new Function("a","b=1","return a+b")

相当于

```
function anonymous(a,b=1) {
   return a+b
}
```

2017年11月9日 10:35

月份：0-11
日期：1-31
时数：0-23
分钟：0-59
秒钟：0-59
毫秒数：0-999
星期：0（星期日）-6（星期六）

new Date(2015,0,1,13,30,35,505) Thu Jan 01 2015 13:30:35 GMT+0800 (中国标准时间) 年月日时分秒,可缺省任意一个，会默认为0
new Date(1420147835505) 传时间戳
new Date() Thu Nov 09 2017 10:41:33 GMT+0800 (中国标准时间)
new Date('May 4,2015') 传字符串
Date() 返回字符串 "Thu Nov 09 2017 10:41:28 GMT+0800 (中国标准时间)"
new Date().toString()===Date() true

setTime/getTime 设置获取时间，以时间戳的形式
setDate/getDate 设置日期，d.setDate(31)
setMonth/getMonth new Date().setMonth(2) 将月份设置为三月
setFullYear/getFullYear 设置获取年份
getDay 获取星期数，从0开始
getTimezoneOffset 获取本地时间与UTC时间差距，以分钟为单位，480/60=8

toUTCString()
toDateString() 获取日期
toTimeString() 获取时间
toLocaleString() 格式更为友好
toLocaleDateString()
toLocaleTimeString()

Date.now() 返回时间戳 ，等同于new Date().getTime()
Date.parse(string) 返回时间戳
Date.UTC() UTC(Universal Time Coordinated,通用协调时)，比北京时间慢8个小时；格林尼治平均时(GMT, Greenwich Mean Time)，同UTC
可传入年月日时分秒，返回时间戳
new Date(Date.UTC(2018,0,11))==>Thu Jan 11 2018 08:00:00 GMT+0800 (中国标准时间) utc时间转为本地时间快8个小时

```
a=new Date(2017,1,1)
Wed Feb 01 2017 00:00:00 GMT+0800 (中国标准时间)
a.toString()
"Wed Feb 01 2017 00:00:00 GMT+0800 (中国标准时间)"
a.toUTCString()
"Tue, 31 Jan 2017 16:00:00 GMT"
```

北京为东八区，领先UTC八个小时，所以本地时间后面有+0800意为领先utc8个小时
https://zhidao.baidu.com/question/584155849371920925.html

例：
2016年6月20日是星期几？
d=new Date(2016,5,20)
d.getDay()==>1 所以是星期1
d.toDateString() => "Mon Jun 20 2016"

2016年到3016年有多少个6月20日是星期一，统计一周中6.20号出现的分布情况

```
var starts=new Array(7).fill(0)
for(var i=2016;i<3016;i++){
    starts[new Date(i,5,20).getDay()]++
}
starts  ==》[140, 146, 140, 145, 142, 142, 145]
所以有140个星期天和145个星期六
```

类型判断

2017年5月12日 20:40

原始类型包括三种数据类型。

- 数值（number）
- 字符串（string）
- 布尔值（boolean）

合成类型也包括三种数据类型。

- 对象（object）
- 数组（array）
- 函数（function）

还有两个特殊值`null`和`undefined`

来自 <http://imjeen.github.io/javascript/2015/07/26/judge-type>

获取对象类型

typeof [1,2,3]==>"object"

toString.call([1,2,3]) =>"[object Array]"

var a={"1":2}

a instanceof Object =>true

typeof 运算符

原子类型字符串、数值、布尔值的typeof运算返回相应的类型值；而合成类型中只有函数的typeof运算返回‘function’，其他都返回‘object’。

两个特殊值里， typeof undefined 为 ‘undefined’，而typeof null 为 ‘object’。

instanceof 运算符

`instanceof` 是用来判断一个对象是否为某个构造函数的实例。值得注意的是原子类型的值不是对象，所以不能用instanceof。

数组、函数和正则表达式等对象，也是`Object`对象的实例。

另外，不同window或iframe之间对象类型检测不能用instanceof。

Object.prototype.toString.call()/toString.call()

实例对象的toString方法返回一个字符串“[object Object]”，其中第二个Object表示该值的准确类型。这是一个十分有用的判断数据类型的方法。

注意：如果在 `function` 上调用`toString` 则会返回当前源代码的字符串。

结合使用function的call方法并返回相应的结果：

- 数值：返回[object Number]。
- 字符串：返回[object String]。
- 布尔值：返回[object Boolean]。
- undefined：返回[object Undefined]。
- null：返回[object Null]。
- 对象：返回”[object “ + 构造函数的名称 + “]” 。

```
Object.prototype.toString.call(2) // "[object Number]"
Object.prototype.toString.call('') // "[object String]"
Object.prototype.toString.call(true) // "[object Boolean]"
Object.prototype.toString.call(undefined) // "[object Undefined]"
Object.prototype.toString.call(null) // "[object Null]"
Object.prototype.toString.call(Math) // "[object Math]"
Object.prototype.toString.call({}) // "[object Object]"
Object.prototype.toString.call([]) // "[object Array]"
```

值得注意的是 `null`：返回[object Null] ， `undefined`：返回[object Undefined]

可以利用上面的特性，写出一个比typeof运算符更准确的类型判断函数。

```
var type = function (o){
    var s = Object.prototype.toString.call(o);
        return s.match(/\[object (.*?)\]/)[1].toLowerCase();
};
type({}); // "object"
type([]); // "array"
type(5); // "number"
type(null); // "null"
type(); // "undefined"
type(/abcd/); // "regex"
type(new Date()); // "date"
['Null',
'Undefined',
'Object',
'Array',
'String',
'Number',
'Boolean',
'Function',
'RegExp',
'Element',
'NaN',
'Infinite'].forEach(function (t) {
    type['is' + t] = function (o) {
        return type(o) === t.toLowerCase();
    };});
type.isObject({}); // true
type.isNumber(NaN); // false
type.isElement(document.createElement('div')); // true
type.isRegExp(/abc/); // true
```

http://imjeen.github.io/javascript/2015/07/26/judge-type

2017年5月11日 12:17

递归条件

1.必须要有结束条件，其他情况总是把问题往结束条件推导

2.实现递归函数时，认为此函数已经被正确完整的实现

在实现过程中，就把它当成一常规函数调用

--

对半且不需管另一半 O(log2 n)

需要管为O(n*log2 n)

2017年5月8日 11:53

switch/case语句注意点

switch里的表达式与case后面的表达式的对比是使用===

最好是能在每个case内加上break,否则进入某个case后，会一直执行遇到break才会退出switch语句

default最好写在最后

JS代码在什么情况下会发生（隐式/自动）类型转换

当运算符两边的类型不符合该运算符的期望时，两边的值就会发生类型转换

不同类型在进行运算时，会产生类型转换，该回答不够严密，python就可以，如'abc' * 3 ---> 'abcabcabc'

JS语言的【所有】运算符

+-*/

== != === !== < > <= >= =

&& || ！逻辑运算

~ & | ^ 位运算

>> 移位运算

>>>

<<

a?b:c 选择运算

%

a = (1,2,3,4)逗号运算符 最终得到4

typeof(类型检测运算符)

a[b] property access operator(属性存取运算符)

a.b

in

delete

O 算法复杂度中的大O符号的具体定义是什么

如果存在正常数c和n0,使得当N>=n0时T(N)<=cf(N),则记为T(N)=O(f(N))

表示能够找到一个n0,以及c,让n>=n0的时候，cF(n)>T(n)

词法作用域

对于任何一个变量的访问，都从代码中书写该变量的位置开始查找，逐级往上层作用域查找

词法作用域最重要的特征是它的定义过程发生在代码的书写阶段

动态作用域的作用域链是基于调用栈的，如this指向是在运行函数时确定的

词法作用域的作用域链是基于代码中的作用域嵌套

闭包

内层函数可以看到外层函数内的局部变量，但外层看不到内层函数里的局部变量。

一个函数a运行返回另一个函数b，只有返回的这个函数b才可以访问a内的变量

函数每次运行时创建的局部作用域都是不相关的

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

来自 <https://www.zhihu.com/question/34547104/answer/198016466?utm_source=qq&utm_medium=social>

垃圾回收/内存泄露
程序在运行过程中会创建很多值
当这些值不再使用时，其所占用的内存就会被自动回收，一般只在解释型语言中存在（由虚拟机执行的语言如java,js,py），可以用引用标记法来解决
没用的变量或其他一直没用回收，如不断产生的闭包内的变量

事件循环：虚拟机自身会每隔一段时间监测有没有需要他执行的程序，在执行一段代码时，不会因为另外一段代码而被打断，而是完成之后，再处理其他代码
虚拟机每隔一段时间去检查有无任务（代码）需要执行
任务来至于事件被触发，事件被触发后，事件的回调就会放入任务队列中等待执行或者是异步任务结束也会触发主线程上的回调
事件循环正在执行某些任务（代码）的时候，新加入的任务是无法执行的，会等待正在执行的任务结束

异步函数在执行结束后，会在事件队列中添加一个事件（回调函数）(遵循先进先出原则)，主线程中的代码执行完毕后（即一次循环结束），下一次循环开始就在事件队列中“读取”事件，然后调用它所对应的回调函数。这个过程是循环不断的，所以整个的这种运行机制又称为Event Loop（事件循环）

同步：代码按顺序执行，必须等到上面的执行完才可以执行下面的代码，期间不可以执行任何其他的程序
是指某个函数在当前这一次事件循环中被调用

异步：程序在等待结果的同时，可以执行下面的代码，结果返回后，再执行其回调函数
某函数在本次事件循环结束后的某个时间点被调用

SVG　几何，dom元素
canvas 美术 ，无法控制一个画好的点，只能重新画
canvas与svg的区别
svg　数据量小，dom元素，所以复杂度稿会减慢渲染速度，不依赖分辨率，支持事件处理器，不适合游戏应用
canvas 时时更新，适合图像密集型的游戏，其中许多对象会被重绘
如何避免canvas卡顿
不要刷新那么频繁，降帧（fps）,setInterval时间设置长一点，浏览器自身降帧，rAF（requestAnimateFrame）也随之降帧

2017年5月23日 12:04

JSON：JavaScript Object Notation

只是符合某种标准的字符串，不是js代码

格式要求：

属性必须用双引号，字符串也用双引号

只能出现直接量，不能有函数、表达式、变量，没有undefind值

不能有注释

数组或对象最后不能有多余的逗号

双引号内部不能出现明文tab，可以{"a":"\\t"}

```
> JSON.parse('{"a":"  "}')
✖ ▶Uncaught SyntaxError: Unexpected token in JSON at position 6
    at JSON.parse (<anonymous>)
    at <anonymous>:1:10
```

```
JSON.parse('{"a":"\\t"}')
▶ Object {a: "    "}

JSON.parse('"\\t"')
"    "
```

JSON.stringify(value,cb,white)对象或值转换为JSON字符串，内部可以为{}，[], value

可以传第二个参数,可为函数或数组（白名单），定制输出；为数组时，即只出现数组里的key值

第三个参数为空白字符数量，容易阅读

```
JSON.stringify({a:1,b:2},["a"],4)
"{
    "a": 1
}"
```

例：JSON.stringify对象内的function会被过滤为null,可以用定制输出来解决这一点

```
JSON.stringify([1,()=>console.log(1),3])
"[1,null,3]"
```

```
JSON.stringify([1,()=>console.log(1),3],
(key,value)=>{
  if(typeof value=="function")
    return value.toString()
  else return value
})
"[1,"()=>console.log(1)",3]"
```

```
JSON.stringify([1,()=>console.log(1),3],(key,value)=>{
  if(typeof value=="function")
     return value.toString()
   else return value
})
"[1,"()=>console.log(1)",3]"
```

JSON.parse(text,cb) cb获得key,value键值对，可以修改value,或删除value（返回undefined即可）

JSON字符串转换为对象或值

```
JSON.parse('{"a":1,"b":2}',(k,v)=>{if(k=="a")return undefined;return v})
▶ {b: 2}
```

2017年6月6日 9:46

多态 如toString()

```
var a=new Complex(1,2)
alert(a)    //alert里面参数会默认转换为string,所以它会先找a的原型里的toString方法，直到找到Object.toString()
```

for of也是一种多态

自定义number的迭代器函数

```
Number.prototype[Symbol.iterator] = function * (){ for(var i = 0; i < this; i++) {
        yield i
}}
for(var i of 9) { console.log(i)}
```

"use strict"

可以利用闭包避免多个js文件合并时，导致严格模式失效的问题

```
(function(){
  "use strict"
  function f1(a){b=1; console.log(a)}
  function f2(b){console.log(b)}
  f1(1)
  f2(2)
 })()
```

如果函数的形参用了es6的新语法，
则函数体的第一句不能使用use strict触发严格模式。
但可以把这个函数放在一个触发了严格模式的js或者script标签里。
es6的class语法是默认严格模式的

严格模式下eval不再为当前所处代码块的范围引入新变量，或改变变量，即其自生产生闭包

```
var x = 17;
var evalX = eval("'use strict'; var x = 42; x");
console.assert(x === 17); true
console.assert(evalX === 42);
```

坑点：

```
function f(a){
    a={b:1};
    arguments[0].b=9;
}
var obj={b:0}
f(obj)
obj=> {b:0}  //调用时，obj与a解绑之后，argumengts也与obj解绑，a与arguments指向同一个地址，因此obj不会改变
```

```
function f(a){
    "use strict"   //增加严格模式，则修复了这个坑点，最后obj为{b:9}
    a={b:1};
    arguments[0].b=9;
}
```

断言 assert

就是在程序的某些位置（一般是函数的开头）来判断程序所要运行的必要条件是否达成，如果没有达成，则直接抛出一个错误

一般用在函数的开头，来确保函数接收了正确的参数

用在测试框架里面，来告诉测试框架此条测试没通过

```
console.assert(1===1)
```

```
function assert(expr) {
  if (!expr){
    throw new Error()
  }
}
```

2017年6月9日 10:47

堆：（完全二叉树，最后一层可以不满）
大堆：根节点比每个叶子节点都大
小堆：根节点比每个叶子节点都小
以下都考虑用大堆

排序渐进方法：

维护堆，把堆中最后的叶子节点去掉放到根节点之后，再调整为堆：
把新的根节点与下面的叶子节点做比较，若叶子节点有比根大的，则根与较大的叶子节点交换位子，然后用换下来的根与他的左右叶子节点继续做比较，直到比到最后一层。

把数组转换为堆：
从最后的结点开始，依次往上走，每走到一个结点，比较他的左右子节点，
左右子节点为空则直接返回，
若只有左结点，则比较根与左结点的大小，然后换位
然后将交换到下面的子节点作为根再重新调整他为堆，即调用上面的方法
整体思路：从最后的结点开始，依次往上走，每走到一个结点，调整它到最后一个结点为堆，即调用上面的方法

堆排序：
先转换为堆，然后每次取出最后一个结点与根节点换位置，即把根（最大的元素）放在数组的最后
然后再调整除了最后那个元素之外的数组为堆，即调用上面的方法，依次类推，直到移动到第一个元素为止

霍夫曼编码

队列：先进先出
两个栈模拟一个队列：要出栈的时候，把所有栈里的内容放到另外一个栈，然后在另外一个栈里出栈，依此类推

栈：先进后出
两个队列模拟一个栈：要出队列的时候，把前面的全入另外一个队列，然后出队列，
再有入队列，则继续入到另外一个队列，要出队列，则同理除最后一个把前面的全入到那个空的队列，依此类推

图
深度优先遍历：一条道走到黑，走迷宫，走到死路往回退，直到走出；二叉树先序遍历；八皇后
广度优先遍历：二叉树按层遍历，考虑每一种情况的情况，直到走完

字符串匹配的KMP算法

来自 <http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.html>

并查集

['abc', 'foo', 3, '7', {2}] value

[0　　1　　2　3　4] index

[-1,　 -1,　 -1, -1,　-1] uset

给要合并的元素生成一个值都为-1的数组，数组长度即集合长度

-1表示当前项是以自己为代表的集合

后比如第0项与第一项合并union[0,1],即下标为1的值变成0，下标为0的变为-2表示以0为代码的集合长度为-2，合并之后还剩下的负数的长度即为集合数量

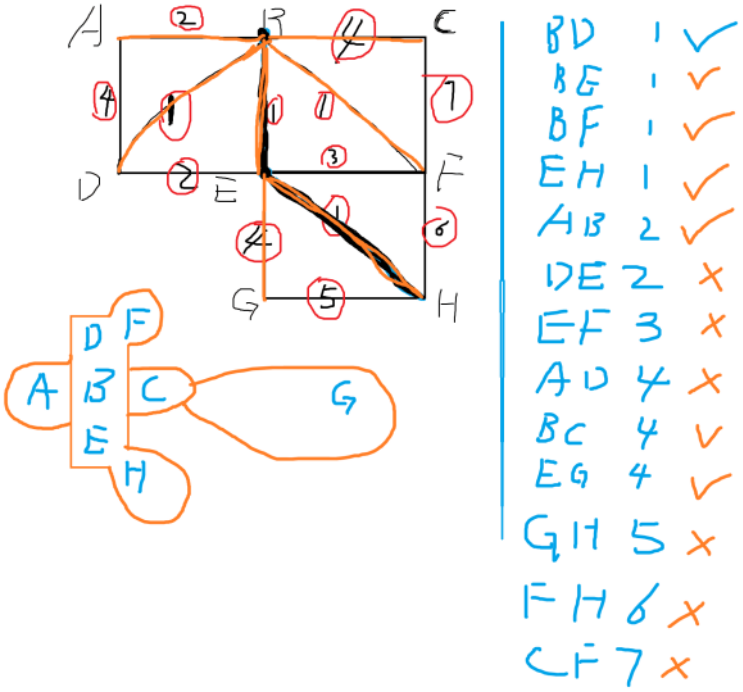
union[1,2],即下标为2的值为1，下标为1的值-1

[0　　1　　2　3　4] index

[-2,　 -1,　 1, -1,　-1] uset

并查集还可以用于图的算法，如计算图中需要能到每个点，但权值要最小的路径（不能构成环即实现最优路径）

如下，圆圈表示每条路径的权值，按照权值大小给每条路径排序，然后取各个点的集合，不能有点重复即可，最后剩下的路径即为得到的最优路径



布隆过滤器　在大数组中查某个项，时间复杂度O(1)

　用一个很大的按位实现的数组， 存放原数组里的每一项用哈希函数算出值对应在该大数组中的位置，查找的时候再算一遍哈希值，直接从数组里取，可以理解为直接用下标取。

正则

2017年6月20日 14:22

两种写法：
var re1 = new RegExp("abc"，"gim")；第二个参数可省略
var re2 = /abc/; ---是对象，不是原始类型

匹配过程：回溯

具体语法
[ab] [0-9] 出现任意一个
\d 任意数字
\w 任意字符，包括下划线 [0-9a-zA-Z_]
\s space,tab,换行,全角空格

\D 非数字字符
\W 非字符
\S 非space,tab,换行,全角空格
. 任意字符，除回车，[.]为出现点，此时失去特殊意义
^ 不在[]里的字符，只有在[]里有特殊意义
[^] 匹配任意符号，包括回车， 同[\s\S]
| 选择性匹配
\uHHHH 匹配unicode，HHHH为16进制 /[\u6211-\u6266]/.test("我")
\rHH 匹配ascii码

出现次数匹配
? 0次或1次
* 0次或多次 ，.*即为出现任意字符任意次
+ 至少出现一次
{1,2} 出现次数1到2次
{1,} 1次以上

组合
(ab) 出现ab组合，如cababd即为true

匹配类型
i 大小写不敏感，如/ab/i,默认敏感，
g,即全局匹配
m,多行匹配
如 /ab/igm 顺序无关

零宽字符匹配，匹配出来的是位置
^ $ ^表示开始位置，$表示结束位置，/^\d+$/ 即匹配数字，如果对正则设置了多行修饰符m,则匹配的是每一行的开始或结束位置
\b 匹配位置，单词的开始结束位置，即\w|\W中间的位置，即一边为\w，一边非\w的位置
单词边界符可以是字符串的开始或者结束位置，也可以是任意一个单词字符与非单词字符的中间位置
如："fwe f !fa@w".replace(/\b/g,"8") ==》"8fwe8 8f8 !8fa8@8w8"

以零宽位置为结束结果总为null，必须以正常字符为结尾
如'1'.match(/^(.*)(?=.)$/) ，要以.前面的位置为结尾，.肯定在位置后，所以匹配结果永远为null

/cat/.test ("concatenate") // true

/\bcat\b/.test ("concatenate")) //false
如去掉单词后面的叹号
"Hi! Hi!".replace(/\b!+/g, '') //单词的结束位置为叹号则替换

不捕获组合，但其内部的括号可以捕获，如下，数组下标为1项不存在

```
"abc".match(/(?=.)/)
▶["", index: 0, input: "abc"]
```

"fewaf fwef".replace(/(?!^)(?=.)(?!$)/g," ") 所有字符前后加空格，除了开始结束位置

扩展：

(?=expr) 正预测，断言某个位置的右边满足expr
(?!expr) 负预测，断言某个位置的右边不满足expr
(?<=expr) 正回顾，断言某个位置的左边满足expr（js不支持）
(?!=expr) 负回顾，断言某个位置的左边不满足expr（js不支持）
可以在js里使用(?!expr)(....)(desired)来模拟回顾型零宽断言
即(desired)的左边不能为expr，且需要四个字符
如匹配如下左边为a的字符，即意为位置的右边不为a的字符
"abacad".match(/(?!a)(.)/g) ==》 ["b", "c", "d"]

例子：实现从右边开始三个字符一个逗号
"bacfCFasCdsafsa"
.replace(/(?!^)(?=(...)+$)/g,',') =》 "bac,fCF,asC,dsa,fsa"
//(?!^)意为该位置的右边不是开始的位置，(?=(...)+$)意为该位置的右边至少有三个或3的倍数个字符为结尾

数字格式化，整数部分三位一个逗号
'12333.12'.replace(/(?!^)(?=([^.]{3})+(\.|$))/g,',') 三个三个组合的最右边为 . 或者为结束位置

如
/ab/i.test("AB") ==》true
/\\/.test("\\") ==》true 反斜杠匹配

RegExp 参数中的符号需要转义
a=new RegExp("\\b\\d\\b") ===> /\b\d\b/
es6新语法 Tagged String f` ， 输入什么即是什么
a=new RegExp(String.raw`\b\d\b`) ==》/\b\d\b/

其属性：
source 返回原始匹配串
lastIndex
全局匹配时(g)，要用reg.exec()从lastIndex（初始为0）位置开始找，找完后lastIndex置为匹配结束位置的下一个位置，匹配不到则重置为0
非全局匹配，则一直从0开始，此时exec方法等同match
该值可设置
reg.exec(string)
返回数组，index为第一次出现的位置，每次只找一次，找完lastIndex值设为匹配结束位置的下个位置

```
/123/.exec("1123")
▶["123", index: 1, input: "1123"]
```

string.match(reg)，同exec，
若reg为全局匹配，即加g,则返回所有匹配到的,此时不捕获组合
否则只匹配到第一次的，返回捕获数组，index，input等

未匹配到组合的返回undefined，匹配到多个组合的返回最后一项

如匹配年月日

[,year,month,day]='20182017-10-20'.match('(....)-(..)-(..)')

year ==>2017

month==>10

day==>20

String.search(/\d/)

匹配到则返回位置，否则返回-1

无法指定开始位置 ，indexOf可以，"1231".indexOf("1",2) ===》3

string.replace

replace(string,string) 只替换一次

replace(reg,"$&$1$2") $&即匹配结果，$1为第一个捕获，依次类推，最大到9

replace(reg,function(match,b1,b2){ ...}) 第二个参数可以为函数，函数内部参数依次为匹配结果，第一个捕获，第二个捕获等，匹配多少次，函数就调用多少次

如

'af2we3ef'.replace(/\w(\d)(\w)/g,function(m,k){console.log(m,k)})

f2w 2

e3e 3

实现 render('我是{{name}},年龄{{age}},性别{{set}}',{name:'a',age:18})

"我是a,年龄18,性别undefined"

```
function render(str,obj){
  return str.replace(/{{([^{}]+)}}/g,function(m,k){
    return obj[k]
  })
}
```

string.split(reg) 按正则分，若里面有分组，则分组也会放到结果数组中的相应位置

尽量少的匹配 ，非贪婪匹配 后加问号即可 ？

+?

*?

??

{}?

如下，匹配到单词的单数形式，否则因为s可以出现0次，所以捕获的到肯定包括s

"2 eggs,3 ifawefs".replace(/\b\d (\w+?)s?\b/g,"$1")

"egg,ifawef"

只分组，不捕获 (?:foo|bar)

如下

不捕获

"aabbcc".match(/(?:a|b)/)

["a", index: 0, input: "aabbcc"]

捕获
"aabbcc".match(/(a|b)/)

["a", "a", index: 0, input: "aabbcc"] 此时$1记住的不是r而是e
"regu expre".replace(/(?:r)(e)/g,'$1$1') ==> "eegu expee"

反向引用，匹配与前面捕获内容相同的内容 /(.)\1/g \1匹配与前面第一个括号一样的内容，最大到\9，即第9个括号
如匹配abb类型的字符
"abb cdd ad".match(/\b.(.)\1\b/g) 注：加了\b无法匹配到中文，下例中就可以
["abb", "cdd"]

"abbb cdd ad 1高佳佳1".match(/.(.)\1/g)
["abb", "cdd", "高佳佳"]

Tagged String f`` 反引号内部为正常字符串，带\符号必须为偶数，因为其会对其进行转义，但raw属性中保存的是未转义的字符
所以可以利用这一点得到输入即输出而不考虑转义

String.raw`\\a` ==>"\\a"
普通：`\\a`==>"\a"

f`aa\\b${a}cc${d}` ${a}里面的值可以是任意合法字符串，如可以继续嵌套反引号或f``
f会运行，接收的第一个参数是一个数组
数组的每一项是被插值符分隔出来的若干个字符串
同时这个数组还有一个raw属性，也是一个数组，里面是所有片段里内容未被转义之前的内容
f还会接收若干个剩余的参数，其值与插值符号内的表达式的值一一对应

自己实现raw函数，同
function raw(...arg){return arg[0].raw.join("")}

function f(...args){console.log(...args)}
f`123` => ["123", raw: Array(1)]

EDM:Email Direct Marketing
不能使用浮动、定位，基本上只能用表格做布局，不能使用外部样式表，伪元素，伪类，
所有样式只能写成内联的，标签的style里，不能有js
可能会泄露隐私

总结：
match,不考虑正则表达式的lastIndex属性。
当正则表达式有g标志的时候，匹配出所有能够满足整条正则表达式的内容
当没g标志的时候，匹配出第一条能满足的内容，同时把分组捕获到的内容也放入结果数组,

返回的数组有属性index，表面匹配的位置，input表明输入的字符串

replace(RegExp|String, String|Function)

两个参数都是字符串：匹配第一次出现的内容为目标内容

首参为正则

次参为字符串，里面的$&,$1,$2是特殊内容，表示匹配到匹配到的内容以及各个分组捕获到的内容

次参为函数，把整个匹配到的内容以及各分组捕获到的内容传给函数做为参数，把函数返回值插入被替换位置

有多少次匹配，函数就会调用多少次

search(reg)

匹配到则返回位置，否则返回-1

无法指定开始位置 ，indexOf可以，"1231".indexOf("1",2) ===》3

re.exec(str)方法

如果re不带g标志，则完全等同于str.match(re)

如果re带有g标志，则从str的re.lastIndex位置开始查找

查找成功后把re.lastIndex置为匹配位置的后一个位置

查找不成功的时候，返回null，把lastIndex置为0

str.split(String|RegExp)

当参数是字符串时，按字符串把原字符串拆成数组

当参数为正则时，按正则把原字符串拆成各部分的数组，但是

当正则里有捕获分组时，分组捕获到的内容也会出现在结果数组的相应位置

2017年6月26日 19:03

事件绑定 addEventListener 默认为window下的方法，默认参数即绑定的这个事件
第三个参数默认为false，即执行冒泡事件模式，如果传true，则执行事件捕获模式
btn.addEventListener("click",function(event){console.log(event.type)},false)
btn.onclick=function(event){console.log(event.type)}
<buttton onclick="alert(1)"></button>
若标签上绑定了函数，js代码里又用onclick绑定，则onclick覆盖标签上的
标签上的事件永远先于用addEventListener绑定事件的执行

（早期的IE浏览器不会以参数的形式传而是在事件发生的过程中放在event一个全局变量中，但是事件结束又无法访问）
移除事件 removeEventListener
btn.removeEventListener("cllick",funcName)

如，绑定一次就解绑
```
btn.addEventListener("click",function once(){
  console.log(1)
  this.removeEventListener("click",once)  //this相当于btn
})
```

ie6下 attachEvent

事件event
共同属性：
type 事件类型；target/srcElement 事件执行来源;detail 连续执行次数；
isTrusted，若为用户真实操作，则该值为true，若为模拟事件，则为false，如el.click()即为false，
或自定义事件 el.dispatchEvent ，具体查看
https://developer.mozilla.org/zh-CN/docs/Web/Guide/Events/Creating_and_triggering_events

click/dblclick 点击/双击
beforeunload 页面关闭之前执行 (提示是否要离开页面等)
focus/blur 不冒泡，但支持捕获,同focusin/focusout,冒泡
鼠标点击时，可以调用select（文字全选中）,click,focus,blur事件
tabindex可以让div有focus事件
鼠标事件
mousedown/mouseup 鼠标按下/鼠标松开，冒泡
该事件之后，会在最小包含down和up事件元素上触发click事件
属性：which 1/2/3 左中右
pageX/pageY 鼠标相对页面的位置
clientX/clinetY 鼠标相对视口的位置
mousemove 鼠标移动事件，可以实现鼠标拖曳元素功能
属性：which 鼠标单按值
buttons 1/4/2 哪些鼠标按钮被按下，组合的值，二进制形式最低位为左，一次为右，中
如同时按中键与右键得到6
大多数浏览器至少支持以上一种属性
mouseover/mouseout 冒泡

属性：relatedTarget over时为来源元素，out时为去向元素

mouseenter/mouseleave 不冒泡 ，可以用来替换mouseover/mouseout（冒泡），实现hover效果

scroll 滚动条滚动时触发，不冒泡，可以实现滚动条的进度条等，类似插件有progress.js,progressbar.js
用preventDefault无效，因为滚动总在发生之后才调用事件
mousewheel/DOMMouseScroll（firefox） 鼠标滚动就触发，兼容性不好
同scroll，可以阻止默认事件，可以用来实现子元素滚动而父元素不滚动，滚动距离需要模拟实现
contextmenu 右键菜单事件
键盘事件 都冒泡 ，获取焦点focus位置为触发该事件的源（可用tabindex），若没有，则默认为document.body
公共属性 ：key 即为输入键
keydown
属性
keyCode 输入键的Unicode，都为大写的值
shiftKey,ctrlKey,altKey,metaKey，即是否按下这些键，bool类型
keypress 对shift，ctrl等按钮无效
属性
charCode 即为输入键的Unicode，区分大小写
keyup

事件冒泡
从事件元素一直到他最外层父元素，如果绑定跟他同一种冒泡事件，则所有外层元素都会运行，且在一个事件冒泡中，在冒泡的过程中事件对象都是同一个
利用冒泡实现事件代理，父元素中右若干个元素都需要绑定类似的方法，则可以只在父元素上绑定，如下

```
< button >A </ button >
< button >B </ button >
< button >C </ button >
< script >
document.body.addEventListener("click", function( event ) {
if( event . t a r g e t . n o d e N a m e == " B U T T O N ")
  console . log (" C l i c k e d ", event . t a r g e t . t e x t C o n t e n t ) ;
}) ;
</ script
```

阻止冒泡 event.stopPropagation() 阻止其所有父元素
事件捕获
若addEventListener函数中传了第三个参数为true，则事件元素会先执行最外层元素的该事件，依次往里找有捕获绑定的执行，直到到target元素；
若target元素上即有冒泡模式，又有捕获模式，则按照绑定顺序执行；
若其父元素同时又有绑定冒泡事件，则先从外捕获到里，再冒泡到最外层

阻止元素默认行为 event.preventDefault()
如id为a的a标签阻止其点击跳转事件的执行

```
a.onclick=function(e){
  e.preventDefault()
}
```

进程与线程的区别：

不同进程之间不能共享内存

同一个进程的多个线程是可以共享这个进程的内存的

cup的时间片轮转是以线程为基本单位的

拖拽事件

拖动至目标元素，通过 dataTransfer来获取拖放的数据

dragover ==>拖文件时不希望因为浏览器的默认行为被直接打开则需要preventDefault

drop ==>拖文件时不希望因为浏览器的默认行为被直接打开则需要preventDefault

dragenter

dragleave

被拖元素上要增加 `draggable="true"`

dragstart

drag

dragend

移动端事件

touchstart

touchmove

touchend

e.touches 得到触摸到所所有点， 支持多点触摸

onorientationchange 手机方向旋转时触发

`passive` 改善滚屏性能，提前通知浏览器执行函数内部是否有阻止默认行为的函数，为true则代表内部不调用preventDefefault，即使调用了也无效

```
elem.addEventListener('touchmove', function listener() { /* do something */ }, { passive: true });
```

2017年6月13日　　16:50

document为全局变量，同window,以下都为document属性

documentElement　html标签
head
body
nodetype
 标签节点　　ELEMENT_NODE　值为 1
 文本节点　　TEXT_NODE　　3
 描述节点，即<!--and-->，COMMENT_NODE　　8

如下，a标签前与后因有空格等，自动生成两个文本节点

```
> $0
<- ▼<body>
       "
       123123
         "
       <a href>aaa</a>
     </body>
> $0.childNodes
<- ▶(3) [text, a, text]
```

返回多个的都是类数组对象（有下标，有长度），不是数组，没有数组上的一些方法
node.hasChildNodes()
childNodes 包括文本节点的所有子节点
children　所有标签子节点
firstChild　第一个子节点，包括文本节点
firstElementChild　第一个标签子节点
previousSibling　前一个兄弟元素，包括文本节点
previousElementSibling 前一个兄弟标签节点
nextSibling　后一个兄弟节点
lastChild　　最后一个子元素,包括文本节点
lastElementChild 最后一个标签子节点
parentNode　父节点
nodeValue　text类型的node的值，所以需要先获取其子元素然后再调用该属性

```
$0
  <a href>aaa</a>
$0.firstChild
  "aaa"
$0.firstChild.nodeValue
"aaa"
$0.innerText
"aaa"
```

createDocumentFragment：批量创建元素，然后插入dom中

cloneNode() 默认为false,浅拷贝，即只拷贝第一个元素，不拷贝其子元素，为true则为深拷贝（无法拷贝绑定在元素上的事件）

getElementsByTagName

getElementById

getElementsByClassName

removeChild

appendChild　append某个页面上的元素，元素将失去在原来的位置，会在新位置上出现

replaceChild　父元素调用，把后面的元素替换成前面的元素

insertBefore　，父元素调用，把前面的元素插入到后面元素的前面

```
parentNode.removeChild(childNode)
parentNode.appendChild(childNode)
parentNode.insertBefore(newNode, baseNode)
parentNode.replaceChild(newNode, oldNode)
```

节点样式获取与设置

node.style.fontSize　可读写，只能获取内联样式，不能获取样式表样式，读出来都为字符串类型

如node.style.zIndex=9，读出来为"9"

```
$0.getElementsByTagName("a")
▶ [a]
$0.getElementsByTagName("a")[0].style
▶ CSSStyleDeclaration {0: "color", alignContent: "", alignItems: "", alignSelf:
$0.getElementsByTagName("a")[0].style.color="blue"
"blue"
$0.getElementsByTagName("a")[0].style.zIndex=5
5
$0.getElementsByTagName("a")[0].style.zIndex
"5"
```

window.getComputedStyle(node).fontSize　读该元素上所有样式，包括样式表上的

parseFloat(div.style.left)//可以直接得到值，不需要再做字符串截取

节点的class属性设置与获取

node.className　　"a b c"

node.classList

```
$0.classList
▶(3) ["a", "b", "c", value: "a b c"]
```

有如下方法：

add

remove

contains

toggle　触发器，没有则增加某类，有则移除该类

如：

```
.hidden{
  display:none
}
```

```
$0.classList.toggle("hidden")
true
$0.classList.toggle("hidden")
false
```

属性设置与获取

node.id/node.htmlFor/node.className 标准属性直接获取<label for="xx" class="a b"></label>

node.style["font-style"] 同 node.style.fontStyle

node.style.cssText 即把css样式当字符串来处理

node.hasAttributes()

node.getAttribute('foo') 自定义属性获取，都为字符串类型

node.setAttribute('foo','123') 自定义属性设置

注：以上两个方法获取或设置的都是html元素内的，而非直接反应到页面上，直接获取或直接设置，则是直接显示到页面上的内容，两种方法不一致

如node=><input value="123">

node.value=222 =>页面上显示222，html中还是123

node.setAttribute("value",333)=>html中变成333，页面上还是222

node.removeAttribute

node.dataset.id="2" data-开头的属性设置，设置data-id属性为2

node.dataset.fooBar ==> data-foo-bar

attributes

获取节点内部文本

node.textContent 返回去掉标签元素后所有该元素内部的文本，原格式返回（不合并空格，回车）

innerHTML 标签内部html

innerText 标签内部文本，只保留换行，返回从页面上看到的样子，如果用css改变了换行等，innerText得到的内容也会改变

textContent 返回标签内部文本，包括script标签，保留空行，空格，换行符
在不支持textContent的浏览器，我们可以使用text与innerHTML代替。
来自 <http://www.cnblogs.com/rubylouvre/archive/2011/05/29/2061868.html>

outerHTML 包括标签自身
outerText 包括标签自身

document.createTextNode('123') ==> 123
document.createElement('a') ==> <a>

以下方法返回的值都是时时的，做移除或增加时，其值会实时变化，因此可以倒着循环
或者用a=[].slice.call(node.childNodes),循环a的长度即可
childNodes
getElementsByTagName
getElementsByClassName

节点选择器（sizzle.js），document与节点本身都有如下方法
querySelectorAll("a[b]") 选中所有带有b属性的a标签 ,控制台中可直接用$$(str)
querySelector(".a") 选中a类的节点，控制台中可直接用$(str)
注：选择器在全局范围内匹配
可以选择一部分伪类
不能选择伪元素
返回静态集合，不会动态更新
document.all[id]

文档流
document.open() --打开新的文档流
document.write("123") --往页面上写入123，若原来有内容会被覆盖
document.write("456") --继续写入456，因为此时为同一个文档流，所以之前的内容还在
document.close() --关闭该文档流
document.open()
document.write("789") --此时之前的内容都被覆盖为789

注：若页面与文档流写在一起，则不会被覆盖，因为处于同一个文档流中

例：遍历dom

```
function walkDom(n){
    do{
        console.log(n)
        if(n.hasChildNodes()){
```

```
                walkDom(n.firstChild)
            }
        }
        while(n=n.nextSibling)
}
```

交换节点位置

```
function swapNode(node1, node2) {
  //判断两个节点是否为父子关系，是的话直接return
  for (let temp1 = node1; temp1 != null; temp1 = temp1.parentNode) {
    if (temp1 === node2) return
  }
  for (let temp2 = node2; temp2 != null; temp2 = temp2.parentNode) {
    if (temp2 === node1) return
  }
  //创建一个辅助节点即可
  var par1 = node1.parentNode
  var par2 = node2.parentNode
  var newNode = document.createElement("my")
  par1.insertBefore(newNode, node1)
  par1.insertBefore(node1, node2)
  par2.insertBefore(node2, newNode)
  newNode.remove()

  //由于克隆不管是js原生还是jq，都没办法完全把绑定在其元素身上的事件克隆完全，所以不能用克隆的办法
  // var newNode1 = node1.cloneNode(true)
  // var newNode2 = node2.cloneNode(true)
  //node1.before(newNode2)
  // node2.before(newNode1)
  // node1.remove()
  // node2.remove()
 }
```

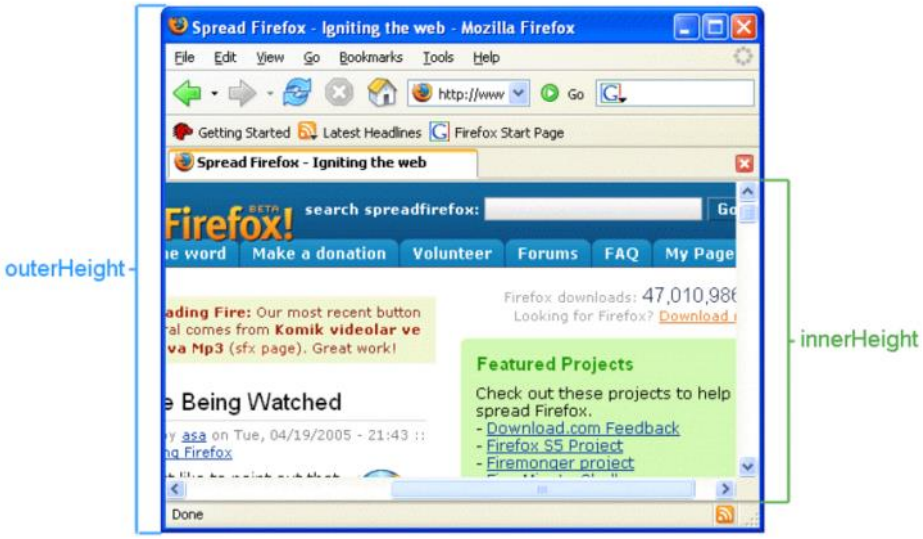
2017年6月14日　　21:33

浏览器重绘总是在页面的js代码运行结束才重绘，若js中改变布局后需要马上获取布局结果，这时候由于页面还没有重绘，js会暂停然后计算出结果之后，再继续运行，所以尽量不要一边改布局一边获取，会很慢

视口宽高

window.innerHeight

window.innerWidth



节点宽度高度，包括border，不包括margin

offsetWidth

offsetHeight

offsetLeft 相对于offsetParent的位置

offsetTop

节点宽度高度，不包括border

clientWidth

clientHeight

scrollHeight 没有滚动条时，等同于clientHeight，否则为整个页面高度，包括滚动条

body.scrollHeight-innerHeight　整个页面需要滚动的高度

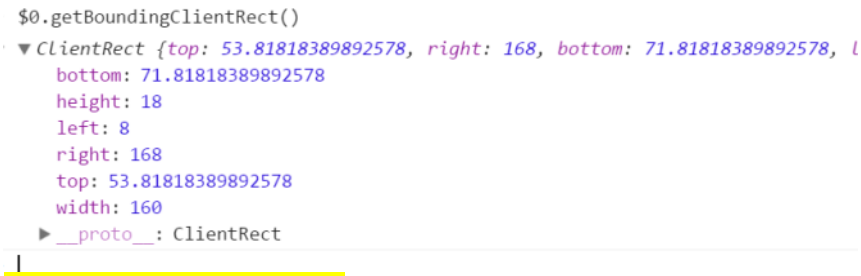
pageYOffset/(body.scrollHeight-innerHeight) 即为滚动条当前进度,body 没有margin或padding的前提下，否则要取html元素的高度，即

document.documentElement.scrollHeight　html页面高度，当body上方有margin或pading时，就用此

window.scrollY　当前Y轴方向上滚动的距离

元素四个边距离视口左上角的距离

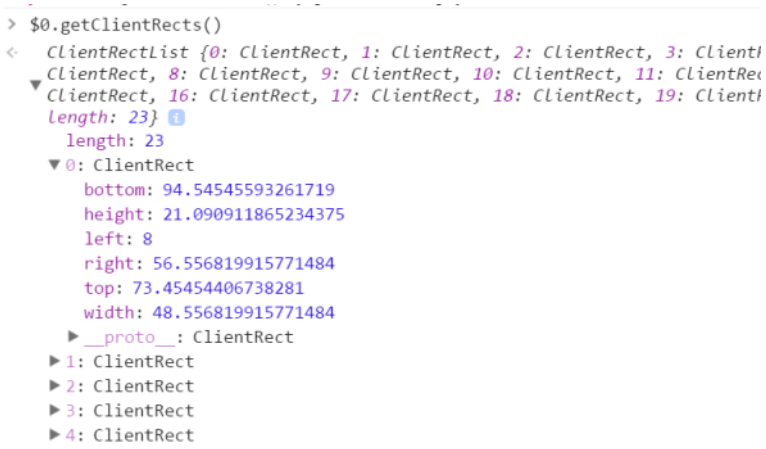
getBoundingClientRect()　返回的heigth与width即offsetHeight与offsetWidth



组成元素的所有矩形框大小，如span元素，因为span元素会断行

（underline.js中即利用每一行的位置定位画自定义下划线）

getClientRects()



滚动条位置，相对页面左上角的位置,window的属性

pageXOffset

pageYOffset

getBoundingClientRect+滚动条的位置　即得到离页面位置

计算滚动条按钮高度

若元素高度145px（即视口右边的滚动条），元素内容高度1000px，则滚动条按钮高度？

x/145=145/1000 ===>x=145*145/1000

设置滚动距离

window.scrollTo(0,100)　-->x轴0，y轴向下滚动100px

window.scrollBy(0,100)　-->在原来的基础上再往y轴向下滚动100px

scrollTop 上方看不见的区域的高度

scrollLeft 左方看不见的区域的宽度

鼠标事件位置event的属性

鼠标相对页面的位置

pageX

pageY

鼠标相对视口的位置
clientX
clientY

2017年6月15日　　19:01

requestAnimationFrame(func)　逐帧动画，在下一帧之前调用func,并传入一个高精度时间，即从页面打开到现在经历的毫秒时间

cancelAnimationFrame(id) 取消事件

如下代码为元素绕着椭圆轨迹运动

此处angle用时间来计算，离开该页面动画停止，回来时，动画会一下跳到该段停止时间该运动到的地方，给人一种一直在运动的错觉，若改成angle+=1，则每次切换回来就是从原来的位置开始

```
var div=document.querySelector("div")
var angle=0,lastTime=null;
function animate(time){
  if(lastTime!=null)
    angle+=(time-lastTime)*0.001
   lastTime=time
   div.style.top=(Math.sin(angle)*50)+"px"
   div.style.left=(Math.cos(angle)*50)+"px"
   id=requestAnimationFrame(animate)
   console.log(oldtime)
   console.log(time)
   if(time>oldtime+5000)cancelAnimationFrame(id)
}
oldtime=requestAnimationFrame(animate)
```

注：该动画如果用循环，将不会有动画在页面上，因为当js还没运行完，浏览器不会更新页面上的显示内容！！

id=setTimeout(func,time)　　错过后还会执行一次

clearTimeout(id)

id=setInterval(func,time)　　错过后不会再执行，会执行下一次

clearInterval(id)

node中的事件循环

process.nextTick(fn)　把任务放在当前事件循环结尾执行，事件一结束就执行 ，相对其他两个最早执行

setImmediate(fn)　　把任务放在下一次事件循环的开头

setTimeout(fn)　　两个setTimeOut之间可能执行其他任务 ，相对其他两个最晚执行，下一次事件循环中执行

EventLoop 事件循环，浏览器总是按照一定的频率在检测有没有需要他做的事，可以通过测试得到平率大概为4ms,在执行一个事件时，不会因为另个事件而被打断，而是完成之后，再处理其他事件

所以以上函数若放在其他事件之前执行，浏览器都会最后处理

如

console.log(1)

setTimeout(function(){console.log(2)},0) -->检测到该事件发现需要等待0ms所以先执行下面的事件

console.log(3)

==》 1 3 2

Date.now() --1970年至今经历的毫秒数

简述使用JS实现页面中动画的原理 ,直接使用for循环为什么出不来动画

为什么要使用 rAF/setTimeout/setInterval?

1、浏览器在执行js，解析DOM，绘制DOM到屏幕这三件事中，任何时候都只能做其中之一 所以修改完dom后要给浏览器机会出绘制DOM到屏幕上 所以我们修改dom的代码要间歇性运行

2、之所以rAF要比另外两个更合适， 第一、它会在页面不显示的时候不运行 第二、它会给所调用的函数传一个高精度时间 第三、它跟浏览器的FPS（帧率）同步

load

2017年7月3日 22:27

页面第一次加载时浏览器先解析代码为dom树，若语法有问题，直接出错，解析过程中如果有外部文件如图片等则会先下载到缓存， 解析中若遇到js则解析完js后执行js ，因此如果js代码中有循环的话，会造成页面显示的卡顿，然后在加载页面，若js文件加载需要的时间比较久，浏览器就会把已经解析的内容先渲染到页面。

load 不冒泡，可捕获 window上的事件
所有图片、脚本、css加载完之后执行
支持onload事件的标签：
script/audio/video/iframe/object
script里的js是先于其onload事件执行的

注：css中的2D/3D变化(不影响布局的操作)是在另一个线程中执行，就算js中有代码死循环，它也不受影响
async defer script 属性
此二属性都是允许让js延迟执行
defer：用了该属性的script标签会按照脚本顺序执行，如两个script ，即同步执行
async：哪个js先加载（运行）完了则哪个先执行 ，即异步执行
一般用在页面中间的广告js，希望其不影响页面加载，所以可以用defer属性来让js延迟执行

defer async 的script标签里不要写document.write
因为他们执行的时候很有可能dom已经解析完毕并且文档流已经关闭（document.close()），再write将会重新开启文档流，会覆盖已解析的dom树

beforeunload/onunload 关闭页面前触发，确保了用户是可以强制退出页面的

error 不冒泡，捕获 window.onerror ，或 元素.onerror
不是事件，只是一个出错时的处理程序
也可用在其他img，script标签，一般用于加载错误，加载的非图片文件
多用于网站错误上报：

```
window.onerror = function(...args){
  var errorJson = JSON.stringify(args)
  uploadError('http://a.com/error_report', errorJson)

  console.log('window on error error', arguments)
  return true
}
```

domready 事件 浏览器支持依次退化
DOMContentLoaded
readstatechange/document.readyState：complete/interactive

load

进程与线程的区别
不同进程之间不能共享内存
同一个进程的多个线程是可以共享这个进程的内存的
CPU的时间片轮转是以线程为基本单位的

Worker 多线程 内参数必须为文件路径，不能为函数
a=new Worker("a.js")
worker之间不能共享数据，只能通过事件与postMsg来发生消息，所以不存在锁，线程安全等问题
所发送的消息是复制以后发过去的，所以修改接收到的消息是不会改变源消息的
worker内不能访问dom，以及任何与UI相关的接口

storage

2017年7月5日 16:28

localStorage 存储在浏览器

同个域（协议主机名端口）且同个浏览器下的localStorage为同一个，每个域可以存储5M

只要用户不主动删除，该值就一直存在

不管存的是什么类型的数据，取到的都是字符串，可以用Json来转换

localStorage.setItem('a',8) 或 localStorage.a=8

localStorage.getItem('a') 或 localStorage.a

delete localStorage.a

storage事件

localStorage发生改变时，打开同个域的页面的storage事件会触发，可以用作页面之间的通信

sessionStorage

作用域是限定在文档源中，顶级窗口，如同个页面的两个ifram可以共享，但两个相同的页面就不共享

一旦窗口或标签页被永久关闭，那么存储的数据被删除

cookie 不能超过4kb

默认值持续在web浏览器的会话期间，一旦关闭，数据就丢失，作用域不局限于单个页面，而是整个浏览器进程

max-age，单位s，一旦设置就会存储在一个文件中，直到过期才删除

作用域默认通过文档源和文档路径来确定，也可通过path和domain属性

值不允许有分号，逗号，空白符，因此可以用encodeURIComponent()进行编码后再存储

访问cookie的限制条件：

httpOnly: 只有http请求才能访问到cookie,js代码获取不到，防止植入js代码盗取cookie

path和domain:同个domain和path下才能获取到cookie，否则发请求不带该cookie

$

2017年7月10日　　14:38

prop()　获取dom元素的属性，而非html里的，attr则是获取与设置html标签上的属性
来自 <http://www.jquery123.com/prop/>

特性（attribute）实际对应的是defaultChecked属性（property），而且仅用于设置复选框最初的值。checked特性（attribute）值不会因为复选框的状态而改变，而checked属性（property）会因为复选框的状态而改变。

prop("checked")
is(':checked')　两者同，都会同步变化

```
$("input[type='checkbox']").prop("checked", function( i, val ) {
        return !val;
});
```

trigger()　触发事件，如$("div").trigger("click")　所有div的click事件都会被触发

jQuery(function($){})　即相当于window.ready方法
默认jquery会传参数jquery给里面的function，为了以防$符号在使用jquery之前已经被其他地方使用而出错，所以可以在外面用jquery,里面的参数写$，该$即表示默认参数jquery
最后，第4种调用方式是传入一个函数给$()方法。此时，当文档加载完毕且DOM可操作时，传入的函数将被调用。这是例13-5中onLoad()函数的jQuery版本。在jQuery程序中，在jQuery()里定义一个匿名函数非常常见：

$四种调用
$(selector,context)
$(ele/document/window)
$(" ",{class:a,click:fn})　新建元素
$(function)
最后，第4种调用方式是传入一个函数给$()方法。此时，当文档加载完毕且DOM可操作时，传入的函数将被调用。这是例13-5中onLoad()函数的jQuery版本。在jQuery程序中，在jQuery()里定义一个匿名函数非常常见：

动画
show(5000)
show("fast/slow")
slideUp("slow")

get()　类数组转换为数组
each(fn)　fn如果返回false，遍历终止
map(function(index,item){})　与Array.map中的默认参数顺序(value,key,arry)相反
$("input").is("input#a")　匹配到一个就返回true
index()　返回匹配对象的索引值
$(selector).index(selector) 在后者里的前者的index

getter　setter
用作getter时，只会查询元素集中第一个元素，返回单个值，要操作多个，用map或其他
setter内部都可以传函数，或对象（键值对）
css()
css({key:value})
css(key,value)
css(key,function(){})
不能获取复合样式如margin,只能margin-left，可以设置复合样式

class
addClass("c1 c2"/function)
removeClass("c1 c2"/function)
toggleClass("c1 c2"/function) 切换类，可以传入函数以每次切换不同的类
hasClass("c1") ==>is("c1") 匹配到任意一个元素返回true，只接受单个类名

val()
select多选返回数组 $("select").val() ==> ["1", "2"]
$("[name=a]").val(["1","3","2"]) 选中带有这些名字或值的复选框
val(function)

text() html()
text:不传参数返回所有匹配元素的所有子孙文本节点的纯文本内容
html:不传参数返回第一个匹配元素的html内容
都可以传函数

位置
offset() 相对页面位置，返回对象带有top，left属性
offset({top:1,left:2})
position() 只读，相对其偏移父元素
滚动条位置 可用在window以及doucument元素上，不可传函数
scrollTop()
scrollLeft()
例：根据页面数n来滚动窗口

```
function page(n){
    var w=$(window)
    var pagesize=w.height()
    var current=w.scrollTop()
    w.scrollTop(current+n*pagesize)
}
```

宽高
width() height() 元素基本宽高
$(window).width() ==>视口宽高
$(document).width() ==>文档宽高
$("div").width(10).width() ==>10 //在box-sizing为content-box时成立，否则border-box返回值包括内边距与边框
下面两方法都只适用于元素，不适用于窗口或文档
innerWidth() innerHeight() 包含内边距
outerWidth() outerHeight() 包含内边距，边框，如果传入true,返回包含外边距

data() 元素数据，非属性
$(a).data("x",1)
$(a).data({"x":1,"y":2})
$(a).removeData("x")

插入替换元素 返回结果dom append某个页面上的元素，元素将失去在原来的位置，会在新位置上出现
append() 在元素内部最后插入 $(selector/tag).appendTo() 注：前面不能传入存文本，否则会被当为选择器
prepend() 在元素内部最前插入 prependTo()
after() 在元素外部最后插入，包括文本节点 insertAfter()
before() 在元素外部最前插入，包括文本节点 insertBefore()
replaceWith() 替换 replaceAll()

clone() 复制元素，默认不复制事件处理程序，可以传入true来复制事件 原生：el.cloneNode()

```
$("a").clone().appendTo("div")
```

包装元素
wrap() 包装选中元素
wrapInner() 包装每一个选中元素内容

wrapAll() 将选中元素作为一组包装

删除元素

empty() 删除选中元素的所有子节点

remove() 移除整个节点 ，包括节点数据与事件，否则可以用detach，不移除事件与数据，可用来做临时移除

$('div').remove('.hello'); 移除有hello类的div

filter() 在指定的集合中移除

unwrap() 移除父元素

事件

load() 可以用来加载新内容，爬虫

$("p").load("a.txt")

$("p").load("http://www.baidu.com/ div") 只加载百度页面的div标签到当前页面的p标签

hover(f,g)

同mouseenter(f) mouseleave(g)

如果传一个参数，即两个事件调同一个处理程序

toggle()

用来切换show()或hide()

delegate 事件代理,现在用on，在body上代理事件，后续添加的元素也可以照样执行

$("body").delegate("div","click",console.log) 用body代理p标签的click事件，就算接下来增加p标签，也可以触发该事件

$("body").on("click","p",console.log) 同上，现在多用on

on 事件注册,若在事件中返回false，即相当于阻止了冒泡与默认事件

$("p").on("click movemove",function(){})

命名空间：定义处理程序组，方便后续触发或卸载特定命名空间下的处理程序，

.+命名空间，可以有多个如click.a.b，表示同时在a,b命名空间下

类似定义了个别名

$("p").on("click.a",f)

$("p").on("click.a",f1)

$("p").on("click.b",f2)

$("p").off("click.a") 解绑事件f,f1 ，不传参时即全部移除

$("p").off(event) ===>$("p").off(e.type,e.handler)

传对象

$("p").on({click:f1,"mouseenter mouseleave":f2}) //表示click的时候执行f1，鼠标事件时执行f2

one() 只处理一次，之后自动注销

触发事件，事件名称可自定义(同原生dispatchEvent)，可用来广播

$("p").click() ===》$("p").trigger("click")

$("p").trigger("click.a",[1,2,3]) 触发a下的click,传参数1,2,3

$("p").trigger("click!") 触发不带命名空间的click

$("p").triggerHandler("click") 不冒泡，不执行默认操作

```
// button1的单击处理程序触发button2上的相同事件
$('#button1').click(function(e) { $('#button2').trigger(e); });
```

http

2017年7月11日 19:59

http请求：\r\n为回车 请求方法： get/post/put/delete put做替换，delete删除资源
例如当http请求方法为delete时，因为电信会拦截该方法，所以一般会在增加请求头x-http-method-override来上传方法名

GET /17 _http . html HTTP /1.1 \r\n //请求方法/请求路径/协议版本，get 幂等，即多次执行都是一样的结果
Host : eloquentjavascript . net \r\n //表明浏览器用什么域名连接到此服务器
User-Agent : Your browser ' s name\r\n\r\n
//空行
.... //若为post方法，会有请求体，Get方法一般请求体为空
服务器返回：
HTTP /1.1 200 OK //协议版本/状态码/状态文字 2开头即表示成功，3表示跳转、缓存，
Content-Length: 65585 //返回内容长度字节
Content-Type: text / html //内容类型为text/html，MIME
Last-Modified: Wed , 09 Apr 2014 1 0 : 4 8 : 0 9 GMT //可以转化为标准的Date
//空行
<! doctype html > //请求返回的内容
... the rest of the d o c u m e n t

状态码：
2xx 成功 3xx 重定向 4xx 客户端访问错误或者网站接口定义错误 5xx 服务器内部错误
401 未授权（未登录）
403 Forbidden隐藏
404 未找到Not Found
soft 404，返回一200的页面，但是内容是404页面的样子，IE浏览器对于小于512字节的404页面是不显示的
417 由于某些政治原因，不显示

301 move permanently 永久移动到了新位置
302 moved Temporarily 暂时移动到了新位置
304 not modified 未改变，所以可以取浏览器缓存

501 not implemented
502 Internal Server Error

MIME content-Type类型可以为：
js：text/javascript
css: text/stylesheet
text: text/plain
jpg: image/jpeg
png: image/png

同一个域名可同时发送6-10个请求，所以把页面用到的资源放到不同域名下
会增加页面加载速度（domain sharding）

动态网站：早期意为服务器动态生成网站，即每次请求都根据请求内容，服务器动态生成返回内容
后期也被意为网站上的动态交互，即非静态网站
表单
表单提交用get，http请求头的方法即为get，且后面url即为提交时的url+querystring
若用post，http请求头的方法即为post, 此时请求体的内容为querystring
此时若querystring中有特殊符号，则提交时会自动编码
（UTF-8，使用一个百分号接着一个两位的被编码字符的十六进制数的形式，转换为十进制即为其ascii码）
encodeURIComponent
decodeURIComponent
（base64编码转换：atob,btoa）

prefetch 预加载

XMLHttpRequest XHR https://www.html5rocks.com/zh/tutorials/file/xhr2/
按照用户输入自动意见的插件：autocomplete/auto suggestion/type ahead
可以发送文本、文件

var req=new XMLHttpRequest()
老版本没有onload事件，通过onreadystatechange事件的state值来判断是否加载完成。0未初始化，1载入请求，2载入完成，3请求交互，4请求完成

```
req.open("get","https://www.baidu.com",false)  //大小写不敏感
 //内部的url如果没有带协议头，则认为为相对路径，如href="mi.com"会在本地找该路径
//默认为true,即异步，即send未返回也可执行接下来的代码，false为同步，即在send完成之后，才会执行下面的代码
req.addEventListener("load",function(){   //send完成之后触发req的load事件
      console.log(req.responseText)
 })
req.addEventListener("error",function(){   //send失败，TCP连接为建立等错误
      console.log("error")
 })
req.send(null)
```

```
function request(url,cb){
 var xhr=new XMLHttpRequest()
  xhr.onreadystatechange=(function (xhr){
    if(xhr.readtstate===4&&xhr.state==200){
       cb(xhr)
   }
  })(xhr)
xhr.open("get",url,true)
xhr.send('')
}
```

post时则需要设置头content-type:application/x-www-form-urlencoded，才能解析出正确的content，否则默认为文本类型

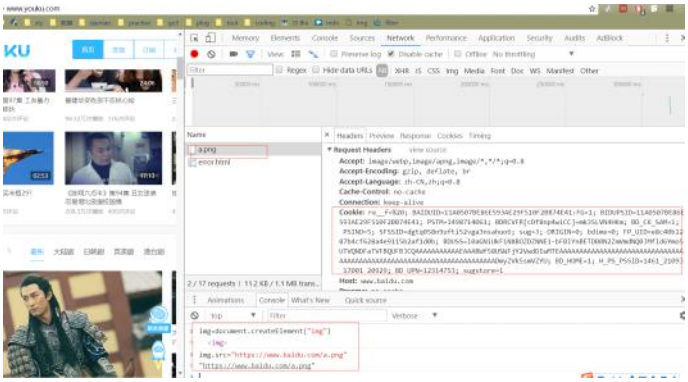
```
xhr.setRequestHeader('Content-Type',
```

fetch(url) 请求url，返回promise，请求默认不带cookie，发送之后不能取消，xhr可以取消（时abort事件）
fetch(url,{credentials:"include"}) 带cookie

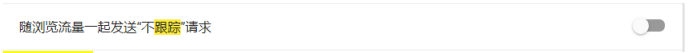
封装过的xhr
axios
superagent

setRequestHeader 可自定义设置请求头，如req.setRequestHeader ("a","b")，则服务器收到的请求头会多一组值，即a:b

cookie 安插广告
在A域名下请求B域名的资源时，若B的账号已经登录，则请求会带上B域名的cookie
安插广告：
    许多广告如淘宝就是利用浏览器这样的设计，在腾讯上有淘宝的ad.js，请求该js时，若你有登录淘宝，则请求信息中会自动带上了你淘宝上的cookie，即账号信息，然后安插跟你相关的广告
如下在百度账号登录的情况下，在优酷下请求百度的图片，请求头自动带上了登录百度的cookie



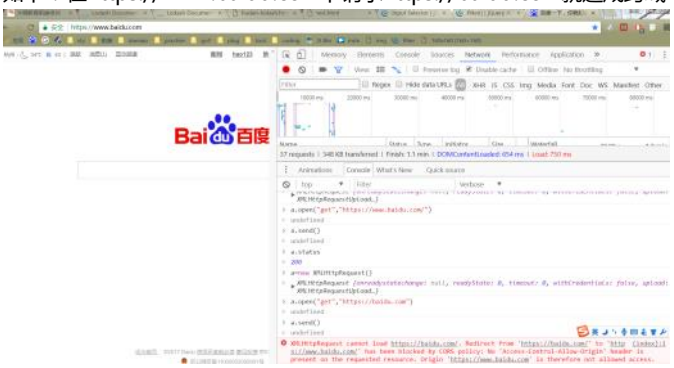
可以在浏览器中设置如下项，即不会再带上cookie



跨域问题

注：http://www.baidu.com 该前面有任意部分不一样，就跨域

如下：在https://www.baidu.com下请求https://baidu.com就造成跨域



在不同域名之间请求访问浏览器会报错，CORS 跨域资源共享，所以若要模拟服务器的发送，接收，需要用node来搭建服务器，并且请求其页面

或者

通过谷歌浏览器自带跨域测试功能，在新的谷歌快捷方式下改变其原地址至如下即可

"C:\Program Files (x86)\Google\Chrome\Application\chrome.exe" --disable-web-security --user-data-dir="C:/ChromeData/disable-web-security/"

//--user-data-dir后项内容若不写，则需要重新关闭已打开的谷歌才可以，否则，需要写成新路径让谷歌创建一个新实例才可以

解决跨域问题：CORS cross-origin sharing standard 跨域资源共享 或者jsonp,iframe

跨域资源共享标准新增了一组 HTTP 首部字段，允许服务器声明哪些源站有权限访问哪些资源。另外，规范要求，对那些可能对服务器数据产生副作用的 HTTP 请求方法（特别是 GET 以外的 HTTP 请求，或者搭配某些 MIME 类型的 POST 请求），浏览器必须首先使用 OPTIONS 方法发起一个预检请求（preflight request），从而获知服务端是否允许该跨域请求。服务器确认允许之后，才发起实际的 HTTP 请求。在预检请求的返回中，服务器端也可以通知客户端，是否需要携带身份凭证（包括 Cookies 和 HTTP 认证相关数据）。

在响应头增加以下头即可实现跨域，需要服务器支持该功能

```
Access-Control-Allow-Origin: http://foo.example  //允许该地址来跨域请求本服务器资源，或者可以写成*通配符，表示任意域都可以向服务器请求
Access-Control-Allow-Methods: POST, GET, OPTIONS  //允许该地址通过这些方法请求
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type //可以发送这些头，如req.setRequestHeader ("Content-Type","text/html")
Access-Control-Max-Age: 60        //跨域在一分钟之内有效，即在请求后的60s内该请求一直有效，即不会再发OPTION请求，只发post请求，超过60s又重新发两条请求
```

具体跨域办法：

XHR请求

https://www.example.com:8080/xxxx.html

这个页面只能请求以https://www.example.com:8080开头url

JSONP,CORS

页面间的交互

两个页面不是同域的，是无法交互的

拿到另一个页面的window引用，可以用postMessage.on('message')

window.name页面跳转时不发生变化

iframe页面的hash内容

上层页面可以修改下层页面的iframe的url，只改hash不会刷新，

如页面A 8080 套B 9090 套C 8080，A可以修改B的hash，B可以修改C的hash，A跟C因为同域可以互通，所以B对C的修改可以通过top属性（即给顶层窗口挂属性）发送给A

CSP:Content Security Policy 内容安全策略

比如不允许其他域名用iframe嵌套本网站，除了某些地址的js不允许请求其他服务器的js等规定

可以在响应头增加Content-Security-Policy

如github的，

block-all-mixed-content ;意为https的页面不能访问http页面

child-src url; 运行被作为子页面的地址，即只能被该url嵌套iframe

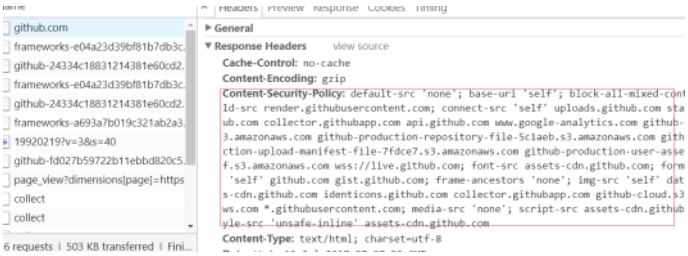
connect-src url1 url2;加载的 js只能访问这些服务器地址

form-action 'self' url;当前表单可以被提交到哪些地址

frame-ancestors 'none';不能被嵌套在任何页面下

srcipt-src:url;只能加载该地址的js，且不允许行内的js

style-src:'unsafe-inline' url; 行内样式允许以及该url下



HTTP协议总结

是一个文本协议。请求/响应模型，一次完整的HTTP协议过程必须是包含这两个过程的，

这两个过程完成以后请求就结束了。不同于TCP可以一直收一直发，直到一端或两端把连接断掉。

由于请求和响应都可以带上若干HTTP首部，次协议也非常容易扩展。

integrity SRI

浏览器拿到资源内容之后，会使用 `integrity` 所指定的签名算法(此例中为sha256)计算结果，并与 `integrity` 提供的摘要签名比对，如果二者不一致，就不会执行这个资源

用来减少由「托管在 CDN 的资源被篡改」而引入的 XSS 等风险

```
<link crossorigin="anonymous" href="https://assets-cdn.github.com/assets/github-a.css" integrity="sha256-rvMIhRfGASjhDFzOjTkphVBAGHRaWKE2kfGieJUYUrs=" media="all" rel="stylesheet" />
```

来自 <https://imququ.com/post/subresource-integrity.html>

https://imququ.com/post/subresource-integrity.html

请求相关首部：

User-Agent，浏览器标识，可以用js通过navigator.userAgent读到

Host，用来告诉服务器，浏览器是通过什么域名来连接到服务器的，带上端口号，

可用此特性来实现同一个服务器服务多个网站，coding/github pages就是这么做的

Referer，告诉服务器，浏览器是在哪个页面请求的这个资源，页面完整地址

可以用来防盗链，即盗用别的网站的图片时，请求地址时，服务器发现请求的地址不是自己的服务器网站，就可以把图片内容替换掉，换成防盗链的图片。

If-None-Match（请求）/ETag（响应）：'hash of the resource'

If-Modified-Since（请求）/Last-Modified(响应)：ISO Time String，浏览器会发请求把最后修改信息与服务器上的对比，

如果一样服务器就不需要重新发，服务器会响应304请求，Not Modified；不一样，服务器再发送新的响应。

cache-control

if-none-match

cookie

accept

Range：bytes=200-1000, 2000-6576, 19000- 断点续传，请求文件的某些部分

accept-Encoding:通知服务器客户端可以接受的内容编码方式，通常是某种压缩算法gzip，deflate

服务的会选择一个方式，使用并在响应头首部Content-Encoding中通知客户端该选择

响应相关首部：

Content-Security-Policy：告诉浏览器，页面里的JS，甚至是其他资源能否被执行

CORS

Access-Control-Allow-Origin:http://www.baidu.com:8080 允许哪些域下的页面里的js请求此资源

Access-Control-Max-Age:86400 表明此次允许跨域访问的时效是多久，过期后浏览器会重新发送OPTIONS请求，以重新询问服务器是否还可以跨域访问。

即若请求一个其他网站的js时，会发送一个OPTION请求，若他允许你访问他的资源，会继续发送一个POST请求，会设置一个过期时间，在这段时间内可以任意请求其网站上的资源，这段时间之内的请求一直发post请求，这段时间之后，若又请求，则又会发送一个OPTION请求，依此重复

Content-Length：配合Connection:keep-alive可以实现Http pipe line

Content-Type： type/substype ; charset=utf-8

Expires:Sun,1 Jan 2006 01:00:00 GMT 本资源的过期时间，过期之前，浏览器可以选择完全不发请求，直接使用本地缓存的版本。过后必须发请求，但可以带上If-Modify-Since，响应也可能是304

Content-Disposition:attachment; filename="download.txt" 弹出下载对话框

Accept-Ranges:bytes 断点续传，支持范围请求

Http pipeline

在同一个tcp连接上发送多个请求，不等服务器返回，就发下一个服务器也是顺次按发送顺序来响应

需配合Content-Length头，这样发送请求的时候可以按长度发送request内容（如果是post请求），否则无法从TCP流里解析出请求/响应

必须按接收到的请求顺序返回相应的响应

Connection:keep-alive 保证下次连接不需要再重新连

http 缓存策略 ctl+F5 强制刷新时，请求不会带任何条件头，直接全部重新请求

Cache-Control (no-cache/private/public),强缓存，第一次请求时服务器返回的响应头，按照max-age单位为秒来缓存到浏览器，之后只有在刷新的时候才会重新发请求，否则重新打开就直接从缓存获取

Last-Modified 协商缓存，向服务器发送请求带If-Modified-Since请求头，如果修改时间变了，则会重新发请求，否则响应304

Etag，同Last-Modified，

响应头表示资源的版本，浏览器在发送请求时会带if-None-Match头字段，来询问服务器该版本是否仍然可用,如果是最新的，则返回304

http://harttle.com/2017/04/04/using-http-cache.html

多线程

Worker() 内参数必须为文件路径，不能为函数

URL.createObjectURL(new Blob(['a','b','c'])) 得到一个链接路径，打开内容为abc

Blob: Binary Large Object，Blob参数数组里为每一个文件片段

一个 **Blob**对象表示一个不可变的, 原始数据的类似文件对象。Blob表示的数据不一定是一个JavaScript原生格式。 **File** 接口基于Blob，继承 blob功能并将其扩展为支持用户系统上的文件。

```
with(document.createElement('a')){
  href = URL.createObjectURL(new Blob(['aaaaaaaa']))
  download = 'aaa.txt'
  click()
}
```

http的两种封装：

RPC Remote Procedure Call 远程调用

http等请求都被封装为函数，直接调用方法

Rest Representational State Transfer 具象状态传输

RESTful API Document

把网络另一边的都当成资源来看待,一般通过url访问返回得到json格式

如http://example*com/products?color=green

https http走在了TLS(SSL)上，中间传输过程进行了加密 ,tls/ssl加密的tcp连接

HTTPS必须要通过域名的形式访问才可以建立，不能用服务器地址

浏览器通过自己的根证值来验证服务器的证书是否为CA机构颁发

CA certificate

SNI Server Name Identify
在TLS（SSL）建立连接前，声明自己要连接哪个域名
当一个服务器服务多个不同域名的网站时，想要这些网站都支持HTTPS的话，那么就需要在TLS连接建立的过程中，指明想要跟哪个域名建立HTTPS连接，服务器会把对应域名的HTTPS证书发送给服务器，因为证书是按域名发放的

XSS：跨站脚本攻击 提交表单时提交脚本，页面中显示该内容以至于每次访问这个页面都会执行该脚本
CSRF/XSRF 跨站请求伪造 不要用post或get,可以通过img标签等伪造请求，用csrf-token,或**检查Referer字段**

长连接 comet
客户端发送请求，待服务器端响应或者连接超时时立刻又发送请求，来达到信息时时接收传送的目的
缺点：维护成本高，某些语言不好处理，对于php每一次请求都需要单开一个线程
缓存会比较占内存
前端一直需要轮询发请求
websoket
在tcp的基础上建立的连接，即承载http的tcp连接一直没断，消息模型，不是字节流
websoket的http请求通过一个upgrade头转变为websoket协议
且只在服务器的upgrade事件上才监听的到
允许跨域
对websocket的封装：
socket.io npm包，服务器端，自动连接服务器，就算服务器重启
socket.io.js 客户端

WebRTC Real Time Comunication 页面间时时通信
peerjs
对webRTC的封装
第一次建立连接时需要服务器，之后的信息传输不需要再经过服务器
传输过程中已经加密

网站上线 https://www.zhihu.com/question/20790576
应用服务器吐出来的html里面引用的静态资源总是与当前版本的html是对应的
静态资源总是以“文件名.hash.扩展名”存储的，因此每次有修改hash就会变化
静态资源服务器会保存每个静态资源的所有版本
每个版本的文件名里都带有文件hash
文件名仅与文件内容相关
所有的静态资源都是强缓存

所以，大公司的静态资源优化方案，基本上要实现这么几个东西：

> 1. 配置超长时间的本地缓存 —— 节省带宽，提高性能
> 2. 采用内容摘要作为缓存更新依据 —— 精确的缓存控制
> 3. 静态资源CDN部署 —— 优化网络请求
> 4. 更资源发布路径实现非覆盖式发布 —— 平滑升级

上线过程： 非覆盖式发布,这样客户端如果是访问缓存的请求老的文件，如果是新打开页面的请求新的文件
应用服务器上线一半
有请求是到新版本服务器
有些请求时到老板本服务器的
新版HTML请求新版静态资源
老版html请求老版静态资源
如果新版html对静态资源的访问时打到老版的静态服务器上，
所以先把所有的静态资源服务器升级并上线
如果老版html对静态资源的访问打到了新版的静态资源服务器上
新版静态服务器存有老版本的静态资源

wenjm jingjyu

2017年7月14日 10:12

disable属性：作为按钮禁用时，不能触发focusd、changed事件，就算js代码中修改该属性，事件还是无法触发

```
[disabled],:disabled{  //因为老浏览器没有伪类，所以要加上[disabled]属性；
    color:gray;
    cursor:not-allowed
}
```

可以利用这一点防止用户不断的重复提交

form elements dom属性 都是类数组

form有属性elements

其内的element有form属性

```
var form=document.querySelector("form")
form.elements.password.form==form  ===>true ，password为name属性
```

selectionStart selectionEnd dom属性 选中文本的开始index与结束index

失去焦点之后也存着最后的值

可以用这个来实现用户选中文字后，按某些快捷键，直接把文字替换为指定内容

粘贴图片也是这么操作，需要上传到服务器，再在用户页面显示

如下，按f2插入当前时间

```
input.addEventListener("keydown",function(e){
    console.log(e.key)
        if(e.key=="F2"){
            var from=this.selectionStart//要提前把之前的保存，否则赋值后selectionStart已经为新值
            var to=this.selectionEnd
            $(this).val($(this).val().slice(0,from)+new Date().toString()+$(this).val().slice(to))
            this.selectionStart=this.selectionEnd=from+res.length //最后光标的位置
        }
});
```

事件：复制剪切粘贴

past 粘贴

e.clipboardData.getData('text/plain') 获取到粘贴内容

cut 剪切

copy 复制

oncontextmenu 右键菜单事件

label
<label><input></input></label>
若把事件绑定在label上，动作在label上，则会触发两次事件
第一次为label自身被触发，第二次为input冒泡到label上，
所以最好把事件绑定到input上

change事件 input事件可时时变化
input中内容改变且失去光标时触发
checkbox/radiobutton/select点击时就触发

change事件在变化发生之后触发，读到的也是变化以后的内容，preventDefault无效

input type=file FileReader
属性 input.files保存选中的文件
如果需要获取文件内容，若文件过大会导致进程卡顿，所以需要异步获取文件内容

```
var input=document.querySelector("input")
input.addEventListener("change",function(){
    var file=input.files[0]
    var reader=new FileReader()
    reader.addEventListener("load",function(){
        img.src=reader.result
    })
    reader.reaAsDataURL(file)//把文件内容读成url，该方法即转换为base64编码
})
```

```
readAsArrayBuffer    FileReader
readAsBinaryString
readAsDataURL
readAsText
```

URL.createObjectURL(input.file[0])===>返回一个可用的url
URL.createObjectURL(new Blob([11,22,33]))

<form action="https://www.baidu.com/s" method="get" target="_blank">
method不填是默认get
提交文件需要增加配置 enctype="multipart/form-data" 到form

CKEditor 富文本编辑器
codeMirro 代码编辑器
IDE

2017年7月17日 16:28

sizzle 先找符合最右边条件的元素，依次往左找

过滤器
:checkbox $("input:checkbox")
:checked
:selected
:eq(1) 效果同$("div").eq(1)，但是后者更高效，因为不需要再解析选择器内的字符串

选取方法
first()
last()
eq(-1) 匹配倒数第一个
slice(2,4) 选取下标为2，3
slice(2) 选取下标为2开始的所有项
filter(selector/jqueryObj/function) 选区过滤，若传入jquery对象，则返回交集
$("div").filter(function(id){return id%2}) ==> 同$("div:even")
has() 在子元素中查找
not("#header") ==> 除了#header以外的div元素
add() 扩充选区，会移除重复并按照文档内容排序
find() 在子元素中查找
children() 返回每个元素的直接子元素
contents() 返回每个元素的所有后代元素，还会返回inframe内容的文档对象
next() nextAll()
prev() prevAll()
siblings() 返回每一个选中元素的所有兄弟元素，除了它自己，一般可用于切换active类
nextUntil() 选取选中元素后面的所有兄弟，直到找到某个匹配该选择器的兄弟元素为止
prevUntil() 同上
parent() 返回每一个选中元素的父节点
parents() 返回选中元素的所有祖先节点
parentsUntil() 返回每一个选中元素的祖先元素 ，直到出现匹配指定选择器的第一个祖先元素
closet() 返回最近的一个祖先元素
end() 匹配元素集到之前的状态
toggleClass(".a .b") 在类a,b切换
jquery 插件扩展
jQuery.fn.println=function(){} ===>$.prototype.println=...

工具函数
jQuery.support ==>检测特性的支持性，可以用Modernizr.js
jQuery.proxy() 同bind

Ajax工具函数
load 传入url时 异步加载该URL的内容，后可选参数是回调函数
$("div").load(function(){}) ==>里面传函数为注册load事件
$("div").load("a.html #price",callbackFn) 异步加载该URL的内容，空格后为选择器，即只加载匹配内容，然后将内容插入每个选中的元素
$("div").load("a.html","search=123") ==>get请求
$("div").load("a.html",{a:1,b:2}) ==>post请求

getScript
//同源 直接执行脚本内容，eval()，回调函数三个参数：脚本内容、success状态码，用来获取脚本内容的XMLHttpRequest对象

jQuery.getScript("a.js",function(content,stateCode,req){
  $("div").my_plugin() //使用加载的类库
})
//跨源 用script标签，只能获取到一个返回状态
getJSON
会对返回数据进行parse操作，返回参数同上，可接受第二个参数为字符串或对象，
若为对象会通过jQuery.param()被转化为querystr添加到URL的问号后
jQuery.getJSON("data.json"，{a:1,b:2}，function(data){
//data参数是对象{x:1,y:2}
})
序列化
$(this.form).serialize() ==>序列化，转换为querystring
jQuery.param() ==>将对象转化为字符串querystring，若第二个参数传入true，可以阻止对对象内的数组进一步序列化
//提交表单时，将表单内容加载到url后来提交

```
$('#submit_button').click(function(event) {
    $(this.form).load(              // 通过加载新内容来替换表单
        this.form.action, '         // 表单url
        $(this.form).serialize());  // 将表单数据附加到表单url后面
    event.preventDefault();         // 取消掉表单的默认提交
    this.disabled = "disabled";     // 防止多次提交
});
```

通用 jQuery.get() jQuery.post()
一个通过http get请求，一个post请求
参数同以上方法，还有个可选的第四个参数，被指定请求数据的类型，如getScript里为“script”,getJson里为“json”
数据类型有text/html/xml/script/json/jsonp

jQuery.ajax() 所有以上都是调用该函数
都返回jqXHR对象，上面定义了success和error方法

```
jQuery.ajax({
    type: "GET",          // HTTP请求方法
    url: url,             // 要获取数据的url
    data: null,           // 不给url添加任何数据
    dataType: "script",   // 一旦获取到数据，立刻当做脚本执行
    success: callback     // 完成时调用该函数
});
```

jQuery.ajaxSetup({//设置任意选项的默认值
  type:"GET",
  dataType:"jsonp",
  timeout:2000,//2s后取消所有ajax请求
  cache:false,//通过给url添加时间戳来禁用浏览器缓存，如a.com?a=b&_=12332123，每次都不一样，所以浏览器会每次都重新加载
  processData:false, //默认为true，表示传入的对象会通过param转换为querystring,为false则不转换为querystring，比如需要发post请求时
})
回调：
beforeSend
success
error
complete sucess或error后执行

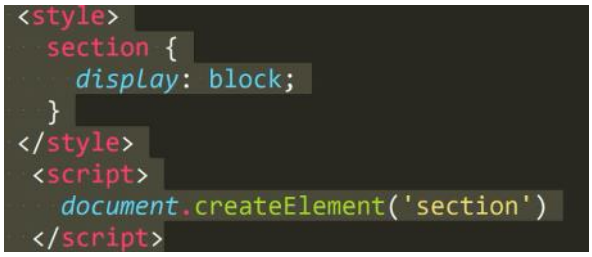
polyfill

2017年7月18日 13:47

polyfill是在老浏览器里把新api完全实现，如实现Promise,Map,Set,Array.from/of

shim则是让无法完全实现的新的api尽量在老浏览器不报错，如console

shiv /shim 老版本浏览器不支持的标签需要通过以下方式生成



polyfill

```
if (!Function.prototype.bind) {
  Function.prototype.bind=...
```

来自 <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind>

Brower Object Model

2017年7月26日 17:18

浏览器提供的接口

domain 获取或设置域，可以用来把两个地址设置为同域而实现跨域
a.taobao.com
b.taobao.com
document.domain=taobao.com 在两个页面下都设置为同一个则实现跨域

cookie
document.cookie="a=b"
document.cookie="a=b;expires=Thu Jan 01 1979 08:00:00 GMT+0800" 删除cookie，设置过期时间

referrer
来源地址，比如从百度中打开雅虎，则在雅虎页面该值即为百度页面

Screen
百度统计通过该属性得到各操作系统的分辨率的使用情况

location 自身及其值都可读写
属性：
host：包括端口号
href:整个url
hostName:不包括端口号
origin:域，域不同则需要跨域
hash:url#后的部分，不会发送到服务器，只发#前部分，
hash改变时，会触发window.onhashchange，可以在handle中请求ajax或刷新
前端路由，:target选择器
search：第一个 ‘?’ 后到第一个 ‘#’ 前的部分
方法：
reload() 刷新页面
assign(url) 即location.href=url，同location=url
replace(url) 替换当前地址，且不把跳转记录保存到浏览器的前进后退按钮中
如下：

```
location
▼Location {href: "http://localhost:8000/a/b.html?a=2&b=3#m",
 assign: function…} i
  ▶ancestorOrigins: DOMStringList
  ▶assign: function ()
    hash: "#m"
    host: "localhost:8000"
    hostname: "localhost"
    href: "http://localhost:8000/a/b.html?a=2&b=3#m"
    origin: "http://localhost:8000"
    pathname: "/a/b.html"
    port: "8000"
    protocol: "http:"
  ▶reload: function reload()
  ▶replace: function ()
    search: "?a=2&b=3"
```

a标签上也都拥有类似属性

```
> var a=document.createElement("a")
< undefined
> a.charset
    charset                    HTMLAnchorElemer
    constructor
    coords
    download
    hash
    host
    hostname
    href
    hreflang
    name
    origin
    password
    pathname
    ping
    port
    protocol
```

history 当前窗口的跳转记录

forward() 按一次前进按钮 ，同理back()按一次后退按钮

go(2) 按两次前进按钮

history.pushState(state,pagetitle,url)：老浏览器不支持，就用hash实现前端路由，即path前加#

改变页面url的path部分，但是页面内容不会刷新，向浏览器历史添加了一个状态，

可以做通过ajax来请求数据并更新页面内容的效果，以便用户把地址在另外一个浏览器打开时，能出现对应的新内容，此时也需要后端的配合才可以，window.onpopstate 事件在history.state的改变切换时后退前进才触发

history.replaceState() 替换当前历史记录，即如登录成功之后应该把跳转页面替换到历史记录的当前登录页面，以保证后退不再到登录页面

可以通过这个方法实现URL的改变但页面不刷新，即pjax=>pushState+ajax，如谷歌浏览器的应用商店的效果

如下效果：点击后退前进时，地址发生变化，同时触发onpopstate事件，state状态回到当前地址的值

navigator

userAgent:会在发送http请求时放进User-Agent首部里，

根据这个头来判断请求来自手机还是PC,以此返回不同的页面内容

如小米的手机网站会自动从mi.com跳转到m.mi.com

（通过浏览器上prevent-log来查看完整的跳转记录）

window

open(url,title,'left=200,top=200')　可以做bookmarklet

opener:在同一个域中打开内部的链接才有值，且值相同，可以实现跨页面的颜色选择器

name：可以通过页面跳转做跨域的数据传递

如从a页面上的链接跳到b页面，b页面依然可以得到a页面上的window.name值，b页面修改window.name然后再回退到a页面，a页面上的值也得到了更新

resize　重新调整页面大小，或放大缩小时触发，触发的比较频繁

可以使用lodash的debounce/throttle 来减少事件执行的频率

debounce 执行事件时，若没到wait时间又执行了，则上次取消，开始新的一次计算时间再执行

throttle　每次事件执行的间隔时间至少为wait，否则取消事件

onhashchange　hash改变时触发，会被记录到历史记录，location.hash改变时，触发

online/offline　电脑上网时触发/下网时触发

onmessage 用于postMessage跨页面通信，同个域下的页面

小米内部链接打开的页面： 给该域发送信息aaa

window.opener.postMessage("aaa","https://www.mi.com")

小米页面：

window.onmessage=console.log　==>即会打印出e

promise

2017年7月31日　　21:32

promise

规则：

1、Promise构造函数内是操作函数，会立即执行，要执行后面的then需要回调resolve或reject，否则就算return了值也不会往后执行

可以理解为request.send()之后的回调函数load与error

2、从 then() 回调中返回某些内容时，如果返回一个值，则会以该值调用下一个 then()。但是，如果返回类似于 promise 的内容，下一个 then() 则会等待，并仅在 promise 产生结果（成功/失败）时调用

3、最终返回的promise由最后执行的then中的返回值决定，若then中返回的是promise，则由该promise的状态决定

then内函数若没返回，则默认返回undefined，返回的promise值也为undefined

4、then若没有对应的函数，则相当于then(it=>{return it},it=>{throw it })，那promise的状态就由上一个promise决定

　　所以没有then即相当于promise状态值即为传入的参数或者抛出异常为该值

如下：

```
> a=Promise.resolve(8)
◂ ▶Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: 8}
> b=Promise.resolve(a)
◂ ▶Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: 8}
> a===b
◂ true
```

```
> Promise.resolve(8)
◂ ▶Promise {[[PromiseStatus]]: "resolved", [[PromiseValue]]: 8}
> Promise.reject(8)
◂ ▶Promise {[[PromiseStatus]]: "rejected", [[PromiseValue]]: 8}
⊗ ▶Uncaught (in promise) 8
```

5、如果在promise中直接throw，即reject throw的值(如下看构造函数)，不需要调reject

promise的构造函数:

```
function promise(executor){
  try{
    executor(resolve,reject)
  }
  catch(e){
    reject(e)
  }
}
```

注：如果在异步函数里throw，promise没办法捕捉到该错误，因为throw是在程序运行了n行之后才执行的，所以不要这么做，直接在异步函数里reject就可

以下两者区别 catch
p.then(f2).catch(f3)　//p成功则执行f2,p失败执行f3，f2若失败也执行f3
p.then(f2, f3)　// p成功则f2,失败则f3，f2失败则往后找第一个then(null,fn)

unhandledrejection事件，在promise reject时，没有reject事件处理时触发

例：

```
new Promise((resolve, reject) => {
        reject(1)
  }).then(value => {
        console.log(value)
  }).catch(reason => {
      console.log(reason)
      Promise.resolve(3).then
      (value=>{
      console.log(value)
        })
  }).then(value => {
        console.log(value)
  }, reason => {
        console.log(reason)
  })
```

输出：

```
1
3
undefined　//因为前一个resolve中没有返回值，所以then中的value为undefined
```

异步函数　async 让异步代码看起来就像同步执行，让代码更容易维护

```
async function (){　//返回promise，await后可以为promise或者async函数或其他异步函数
  var a=await  delay(1000)
 ...
  var b=await  geturl(url)
 ...
}
```

多个async函数同步执行时，看起来就像多线程在执行内部的函数，事实上只是协程

callback实现了promise

genrator与promise实现了async 原理：next之后执行promise等到状态确定，即执行then表示完成，同时把promise的状态通过next传回到yield语句，然后返回到yield语句的左边

```
async function f(){
   for(let i=0;i<4;i++){
      var m=await new Promise((res)=>setTimeout(()=>{res(i);},1000))
       console.log(m)
   }
}
```

```
function getValueAfter(v, d) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve.bind(null, v), d)
  })
}
async function f1(){
  var a = await getValueAfter(4, 1000)
  var b = await getValueAfter(5, 1000)
  return a + b
}
async function f2(){
  var a = getValueAfter(5, 1000)
  var b = getValueAfter(5, 1000)
  return await a + await b
}
f1().then(v => {
  console.log(v)
})
f2().then(v => {
  console.log(v)
})
```

10 一秒后

9 二秒后

2017年8月11日 14:38

减少HTTP请求
合并请求 多个js拼成一个
css sprite 小图片合成大图
icon font / svg

cdn服务器 Content Delivery Network
静态资源
物理距离近
浏览器在同个时间往同个服务器发送6-8个请求，以域名为单位的同一个服务器，域名分片（Domain sharding）可以使同时发送的请求更多

使用缓存
if-modified-since/last-modified
协商缓存，发请求，响应304，只响应响应头，不需要响应体
ETag : RvUixJl3I8 / If-None-Match :RvUixJl3I8 //
协商缓存，发请求，响应304
Cache-Control
强缓存，资源不过期就不发请求的，一般不用于htm资源，可以用在jquey.js等不变的资源上
点击链接，重开页面，地址栏输入地址，强缓存不发请求
f5 会发协商缓存的请求
前端缓存（把资源存在localStorage，一个页面最多存储5M数据）
一般用在移动端，节省流量
压缩
在http连接上启用gzip压缩/或其他压缩方式
一般只针对文本型资源做压缩
html css js
jpg png 无法再压缩
信息熵
信息论

将css放页面顶部，尽早开始下载
把css直接内联到页面里，一般首页
将js放底部
<script nodefer noasync src="xxx.js"></script>
js的下载会阻塞页面DOM的解析（如果下载很久，那么已经解析出的dom是可以渲染出来的）
js的运行会阻塞页面的渲染
解析：解析html并构建dom树
渲染：把页面画到屏幕上
为啥js下载的时候会阻塞？
因为页面解析，渲染，和js运行，都在同一个线程里执行
执行：JS执行线程跟UI线程是同一个线程
defer async 的script标签里不要写document.write

因为他们执行的时候很有可能dom已经解析完毕并且文档流已经关闭（document.close()），再write将会重新开启文档流，会覆盖已解析的dom树

页面里各个资源下载的优先级： css>js>视口内的img元素>img>font

避免css表达式

width:expression('document.body.innerWidth-100')

表达式可能在各种条件下都会触发执行，执行次数太多导致影响性能

使用外部的js和css

因为不同页面使用同个文件的话，缓存就会起作用

分散域名（Domain sharding）

移动端不适合，连接不复用，要重新连接多个连接，费流量

减少DNS查找

DNS prefetch

<link rel="dns-prefetch" href="https://www.taobao.com"/>

会提前解析带该值的地址，之后就不需要再发DNS请求

精简代码

uglify.js ：nmp模块，代码压缩

去除死代码

避免重定向

307 内部跳转

4xx 客户端错误 401 UnAuthorized 未登录

5xx

移除重复脚本

配置ETag

使ajax可缓存

HTTP2

2017年8月10日 17:45

vue响应式

所有data变成setter与getter
getter 让computed能检测到
setter 设置值的时候能够响应到dom

v-model

input type=text 对应 js中string类型
input number 对应 num类型
checkbox 对应bool类型

```
<input type="text" v-model="userInput"></input>
//等同于 <input :value="userInput"  @input="val=>userInput=val"></input >
```

nextTick vue中操作dom，需要在事件循环末尾，否则不会有效果

Promise.then相当于nextTick，
以及MutationObserver 检测dom元素的属性变化，放在当前事件循环末尾执行

setTimeout是在下一次事件循环中（开始之前）执行，也可以

Vue中是 `Vue.nextTick` ，可以在更新数据之后操作dom元素或其他时用此函数

```
var vm = new Vue({
      el: '#example',
      data: {
      message: '123'
      }
})
vm.message = 'new message' // 更改数据
vm.$el.textContent === 'new message' // false 无效
Vue.nextTick(function () {
vm.$el.textContent === 'new message' // true 有效
})
```

来自 <https://cn.vuejs.org/v2/guide/reactivity.html>

数组更新检测

由于 JavaScript 的限制， Vue 不能检测以下变动的数组：
当你利用索引直接设置一个项时，例如： `vm.items[indexOfItem] = newValue`
当你修改数组的长度时，例如： `vm.items.length = newLength`
为了解决第一类问题，以下两种方式都可以实现和 `vm.items[indexOfItem] = newValue` 相同的效果， 同时也将触发状态更新：

```
// Vue.set
Vue.set(example1.items, indexOfItem, newValue)
```

```
// Array.prototype.splice
example1.items.splice(indexOfItem, 1, newValue)
```

为了解决第二类问题，你可以使用 `splice`：

```
example1.items.splice(newLength)
```

来自 <https://cn.vuejs.org/v2/guide/list.html#数组更新检测>

- Vue的是如何实现双向绑定的（即响应式数据更新）？
- 几种表单控件的双向绑定默认都绑定什么类型的数据？能否换为绑定成其它类型的数据？如何做？
- Vue实例是如何实现代理了其$data属性的访问的？
- 为什么vm.$data.ary[0] = 8这样的代码不会触发视图的更新？为什么vm.$data.ary = [8]又可以呢？
- 计算属性是什么？它会在何时更新？它如何知道自己应该在何时更新？计算属性最终实际被Vue转换成了什么？
- v-model的本质是什么？组件如何实现一个自己自定义的v-model？
- 递归组件写过吗？
- 父组件如何向子组件传递数据？反过来呢？

vuex

vue父子通信：父组件通过props 向子组件传递数据，子组件通过自定义事件向父组件传递数据。

如果是组件与组件之间的通信非常复杂，不光是父子组件，还有兄弟组件，那就需要用到状态管理，vuex

来自 <http://www.cnblogs.com/SamWeb/p/6391373.html>

数据中心，本来vue中的computed只能检测data里的数据变更，但是用了vuex里的Store，Store里的数据也可以被computed检测到，相当于一个全局数据可以被检测到，这样父子通信就不需要通过props来通信，可以直接修改store里的数据即可

flux

redux ---------- Vuex

mapStateToProps =》 mapState,mapGettersToComputed

mapDispatchToProps =》 mapMutationsToMethods,mapActionsToMethods

reducer返回新state =》 mutations修改state为新的

mapStateToProps= state=>({})

mapDispatchToProps=dispatch=>({})

ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(Component)

Vuex里则是自动响应的，不需要connect

两个框架都不需要显式的去调用subscribe，即监测数据的修改

2017年8月30日 11:53

react 纯组件 ,组件必须为大写首字母
var comp=React.createClass({}) 生成组件，对象内部调用方法直接用this.meth即可，现已弃用该方法，
现用继承React.Component的类来实现组件，内部调用方法需要bind，或者用箭头函数，否则获取直接传this.meth调用时，内部this失效
React.createElement(type,props,children) jsx内部实际都在调用该函数，标签内部插入代码用{}表示插值，内部只能是表达式，且jsX只能返回一个对象，即要包在一个元素内
ReactDom.render(<comp /> ,ele) 把生成的组件渲染到某个元素内，第一个参数必须只有一个对象，如果有多个组件嵌套，就在外部包个div，因为其实际就是调用createElement函数

在对象内部调用this，this即指对象本身，若调用this的方法，该方法是已经经过bind过的方法，
所以执行方法时，方法内部的this也为对象
在类内部调用this，this即指对象本身，若调用this的方法，该方法内部的this失效，需要重新bind

修改更新数据
this.state
this.setState (注：内部不要调用this.state.xxx或者this.props.xxx，因为这些是异步获取的，
会有延迟，可以通过this.setState((preState,props)=>({a:preState.a+props.increase}))来实现)
this.props
this.props.children 内容分发

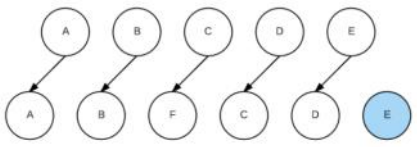
生命周期

https://codepen.io/eduardoboucas/details/jqWbdb

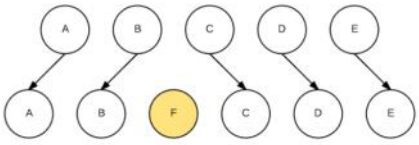
父子组件通信方式
父组件通过添加属性传到子组件
子组件通过读取this.props得到父组件的传值，若该属性是方法，则调用该方法（该方法一般都会用来处理父组件中的数据）来通知父组件修改数据
子组件不能直接改的this.props的值，这一点通vue！！
兄弟组件无法直接通信，只能通过子组件通知父组件修改数据之后，可能会引起兄弟组件的 重新渲染

Dom diff 算法复杂度O(n) http://www.infoq.com/cn/articles/react-dom-diff/
同一层级相同类型的dom元素，更新dom上的不同项，如属性
不同类型或不同层级，删除原来层级的元素新建新元素到原来位置，删除原来层级元素时，会触发componentWillUnmount
列表节点的比较，通过给列表增加属性key，react通过比较key值来确定要更新哪一项，否则会更新所有不对应的项

这时如果每个节点都没有唯一的标识，React无法识别每一个节点，那么更新过程会很低效，即，
将C更新成F，D更新成C，E更新成D，最后再插入一个E节点。效果如下图所示：



可以看到，React会逐个对节点进行更新，转换到目标节点。而最后插入新的节点E，涉及到的DOM操作非常多。而如果给每个节点唯一的标识（key），那么React能够找到正确的位置去插入新的节点，入下图所示：



Context 用于组件通信，不推荐用，因为还在测试阶段，最好用redux

Portals modal组件,模态窗组件，类似messagebox

redux

数据中心，相当于一个全局数据可以被检测到，这样父子通信就不需要通过props来通信，可以直接修改store里的数据即可

flux

redux ---------- Vuex
mapStateToProps =》 mapState,mapGettersToComputed
mapDispatchToProps =》 mapMutationsToMethods,mapActionsToMethods
reducer返回新state =》 mutations修改state为新的

mapStateToProps= state=>({})
mapDispatchToProps=dispatch=>({})
ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(Component)

Vuex里则是自动响应的，不需要connect
两个框架都不需要显式的去调用subscribe，即监测数据的修改

为什么用react！！
react 发布的时候 vue 还没火
react 开始火的时候 angular1 已火炸了
组件化在 ng1 里没有特别重的体现
web components
polymer polyfill
足够简单，完全组件化
还有一个有钱的爹：Facebook
社区繁荣

2017年8月30日 12:05

嵌套函数的this和外层函数的this
不一样，如果内层函数是箭头函数，则是一样的， 否则不一样

```
obj={
 f:function(){
   return this.x-1   //this即为obj自身
 },
 x:1
};
obj.f()  ===>即得到0
[1,2,3,4].map(obj.f);//[NaN,NaN,NaN,NaN] ，里面的this在全局
[1,2,3,4].map(obj.f,obj);//[0,0,0,0]，指定第二个参数为上下文的this
```

this属于哪个函数，包裹this的最近一个function关键字声明的函数
这个函数是如何被调用，有没有bind，是不是传给其他库

从this外面找大括号外部，大括号外部即为this对象
jQuery绑定事件时，默认是把传入的函数call当前的dom元素，所以如果是调用另一个函数，则需要bind

```
class one{
 constructor(){
   this.m=1
   //$("div").click(this.f)//错误！！this.f中的this在大括号里找就是one,调用f时f里的this默认就是dom元素，需哦一需要绑定
   $("div").click(this.f.bind(this))
  //或 $("div").click(()=>this.f())
   /*

    */
 }
 f(){
   console.log(this)
   console.log(this.m)
 }
}
o=new one()
```